ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КІБЕРБЕЗПЕКИ ТА ЗАХИСТУ ІНФОРМАЦІЇ

КАФЕДРА СИСТЕМ ТА ТЕХНОЛОГІЙ КІБЕРБЕЗПЕКИ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Технології тестування і захисту мобільних додатків»

зі спеціальності *125 Кібербезпека та захист інформації*
(код, найменування спеціальності)

освітньо-професійної програми *Інформаційна та кібернетична безпека*
(назва програми)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Анастасія ЧАПЛІЄВА
(підпис)

Виконала: здобувачка вищої освіти групи БСДМ-61
ЧАПЛІЄВА Анастасія
(прізвище, ім'я)

Керівник д.т.н., професор ГАЙДУР Галина
(науковий ступінь, вчене звання, прізвище, ім'я)

Рецензент _____
(науковий ступінь, вчене звання, прізвище, ім'я)

Київ 2024

ЗМІСТ

Стор.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ …… 9

ВСТУП ……..…… 10

1 **АНАЛІЗ ПРОБЛЕМИ ЗАБЕЗПЕЧЕННЯ КІБЕРБЕЗПЕКИ МОБІЛЬНИХ ДОДАТКІВ** ……..……. 12

1.1 Аналіз проблеми забезпечення захисту мобільних додатків ……. 12
1.2 Аналіз життєвого циклу мобільних додатків……... 22
1.3 Аналіз існуючих нормативних вимог та рекомендацій ...……..... 30

2 **АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ ЗАХИСТУ ТА ТЕСТУВАННЯ МОБІЛЬНИХ ДОДАТКІВ**…..…….…….....…. 40

2.1 Захист під час розробки мобільного додатку ……...….. 40
2.2 Захист на етапі тестування ……..… 49
2.3 Захист на етапі розгортання додатку …… 53
2.4 Захист під час експлуатації додатку (використання користувачем)…….. 51
2.5 Забезпечення підтримки та оновлення безпеки після розгортання. 55

3 **РОЗРОБЛЕННЯ ВАРІАНТІВ ТЕХНОЛОГІЙ ТЕСТУВАННЯ ТА ЗАХИСТУ МОБІЛЬНИХ ДОДАТКІВ**……..…. 65

3.1 Технологія тестування SAST на базі рішення Snyk…….…… 65
3.2 Технологія перевірки середовища запуску додатку як частини RASP……. 67
3.3 Технологія захисту обфускації коду JavaScript……..… 70

ВИСНОВКИ …….. 77

ПЕРЕЛІК ПОСИЛАНЬ 79

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація) 81

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API - Application Programming Interface

BOM - Bill of Materials

CI/CD - Continuous Integration and Continuous Delivery

CVE/CVSS – Common Vulnerabilities and Exposures / Common Vulnerability Scoring System

CWE - Common Weakness Enumerations

DAST - Dynamic application security testing

DevOps - Development & Operations

DevSecOps - Development & Security & Operations

DIVA – Damn insecure and vulnerable App

IAC - Inter-Application Communication

IAST - Interactive Application Security Testing

ISO/IEC – the International Organization for Standardization / the International Electrotechnical Commission

ISP – Information security policy

NDA – Non-Disclosure Agreement

NIST – National Institute of Standards and Technology

NVD - National Vulnerability Database від NIST

OSINT - Open-source intelligence

OWASP - Open Worldwide Application Security Project

RASP - Runtime Application Self-Protection

SAST – Static application security testing

SCA - Software composition analysis

SDLC - Software Development Lifecycle

ВСТУП

Актуальність дослідження. Сьогодні мобільні додатки стали невід'ємною частиною нашого життя. Ми використовуємо їх для спілкування, фінансових операцій, купівлі товарів, доступу до державних послуг, управління бізнесом та багатьох інших задач. З кожним роком залежність від мобільних технологій зростає, і це створює нові виклики у сфері кібербезпеки.

Статистика свідчить, що більшість мобільних додатків має суттєві вразливості. Згідно з даними аналітичних компаній, близько 70% таких додатків містять критичні помилки безпеки, які зловмисники можуть використати для крадіжки даних або несанкціонованого доступу до ресурсів користувачів. Основною причиною цього є недостатня увага до питань безпеки під час розробки та тестування.

Особливу небезпеку становлять мобільні додатки, які використовуються у фінансовій сфері, охороні здоров’я, державному управлінні чи інших критичних галузях. Атаки на такі програми можуть призвести не лише до фінансових втрат, але й до порушення роботи цілих систем. Крім того, зловмисники постійно вдосконалюють свої методи: від соціальної інженерії до експлуатації вразливостей у бібліотеках і сторонніх модулях.

Саме тому важливо забезпечити безпеку мобільних додатків на всіх етапах їх розробки та подальшої експлуатації. Застосування міжнародних стандартів, таких як OWASP MSTG чи ISO 27034, допомагає формувати надійні підходи до захисту додатків. А використання сучасних технологій статичного аналізу коду (SAST), динамічного аналізу (DAST) та інші, впровадження безпеки у процес розробки (DevSecOps) та самозахисту додатку у реальному часі (RASP) на етапі експлуатації користувачами, дозволяє вчасно виявляти, усувати та попереджати потенційні загрози.

Дослідження даної кваліфікаційної роботи у цій галузі є особливо актуальним, оскільки дозволяє вирішувати не лише поточні, але й майбутні виклики у сфері кібербезпеки мобільних додатків. Результати такого дослідження

можуть бути корисними як для розробників, так і для організацій, що прагнуть захистити свої мобільні платформи від кіберзагроз.

Об'єкт дослідження – безпечна робота мобільних додатків.

Предмет дослідження – технології статичного тестування на базі рішення Snyk з використанням технології перевірки середовища запуску додатку на базі скриптів та обфускації коду JavaScript.

Мета роботи – розробити рекомендації застосування технологій тестування та безпеки мобільних додатків на всіх етапах їхнього життєвого циклу (від розробки до використання).

Наукові завдання:

дослідити сутність проблеми забезпечення безпечної роботи мобільних додатків;

проаналізувати підходи до забезпечення безпечної роботи мобільних додатків;

проаналізувати існуючі рішення для тестування та забезпечення безпечної роботи мобільних додатків на різних етапах розробки та подальшої експлуатації;

проаналізувати методи та засоби забезпечення безпечної роботи мобільних додатків на базі технології SAST, RASP та обфускації коду;

розкрити порядок реалізації технологій безпечної роботи мобільних додатків відповідно до циклу розробки та експлуатації.

Методи дослідження – опрацювання літератури за даною темою, аналіз експлуатаційної документації, міжнародних стандартів та їх порівняння.

Практичне значення одержаних результатів: запропоновано порядок застосування технологій забезпечення безпечної роботи мобільних додатків, а також розроблено рекомендації фахівцям з кібербезпеки щодо їх застосування впродовж циклу розробки та експлуатації додатків.

Результати кваліфікаційної роботи апробовані на Всеукраїнській науково-практичній конференції «Цифрова трансформація кібербезпеки», яка відбулася 26 квітня 2024 року в Державному університеті інформаційно-комунікаційних технологій, м. Київ[1].

1 АНАЛІЗ ПРОБЛЕМИ ЗАБЕЗПЕЧЕННЯ КІБЕРБЕЗПЕКИ МОБІЛЬНИХ ДОДАТКІВ

1.1. Аналіз проблеми забезпечення захисту мобільних додатків

Сьогодні, навіть в умовах війни, ми не можемо вже уявити своє життя без використання мобільних додатків. Сучасні мобільні додатки здатні покрити більшість наших запитів у найрізноманітніших сферах життя. Спілкування з друзями, купівля продуктів харчування, оплата комуналки, мобільний банкінг, та навіть відслідковування новин про повітряну тривогу, відключення світла – це все на відстані декількох натискань у телефоні.

Та попри надзвичайну різноманітність, зручність і доступність мобільних додатків, ми маємо пам’ятати про кібербезпеку:

- Понад 75% додатків мають принаймні одну вразливість[2];
- Вектором для багатьох атак сімействами програм-вимагачів є використання відомих вразливостей у публічних додатках[3];
- Понад 300 000 користувачів Android завантажили банківські троянські програми через Google Play Store[4].

Через недостатньо захищений або вразливий мобільний додаток зловмисник може:

- Отримати доступ до конфіденційної інформації, такої як ім'я, адреса, дата народження або паспортні дані;
- Отримати доступ до платіжної інформації (номера карток, CVV-коди, банківські рахунки);
- Отримати доступ до облікових записів користувача, включаючи електронну пошту, соціальні мережі або хмарні сервіси, якщо у додатку використовуються слабкі методи автентифікації;
- Завантажити через додаток шкідливе ПЗ на пристрій;
- Перехопити дані за допомогою атаки Man-In-The-Middle (MITM), які

передаються між додатком і сервером без використання шифрування (TLS/SSL);

- Перенаправляти користувачів на підроблені сторінки для збору конфіденційної інформації, такої як логіни та паролі;
- Отримати доступ до камери, мікрофона, місцезнаходження або списку контактів, через недостатній контроль дозволів додатку;
- Використовувати незахищені додатки для атак на інші пристрої, підключені до тієї ж мережі (наприклад, в корпоративному середовищі).

Ураховуючи вищевказане дуже важливо звертати увагу на безпеку мобільного додатку упродовж усього циклу розробки та подальшої експлуатації, щоб якомога раніше виявити та виправити можливі вразливості.

1.2. Аналіз життєвого циклу мобільних додатків

Безпека додатків[5] починається задовго до того, як компанія-розробник програмного забезпечення укладе контракт на створення продукту або ж почне власну розробку. На початку компанія-розробник повинна прийняти та задокументувати конкретні процедури та принципи безпеки та викласти їх у своїй політиці інформаційної безпеки.

Політика інформаційної безпеки, або ISP, — це основні принципи, яких компанія дотримується для забезпечення належних заходів безпеки щодо програмного забезпечення, яке вони створюють.

Політика інформаційної безпеки компанії включає вимоги до процесу розробки та діяльності співробітників щодо безпеки:

- *Вимоги безпеки для розробки рішень* представляють вказівки щодо забезпечення відповідності програми сучасним стандартам безпеки. Політика безпеки включає погляди компанії на кібербезпеку та включає в себе управління ролями та обов’язками, політику витоку даних, плани аварійного відновлення та безперервності бізнесу, методи безпечного кодування, а також практики та стандарти моніторингу безпеки додатків.
- *Внутрішні вимоги компанії* стосуються робочої поведінки та дій

співробітників. Вони можуть включати захист персональних пристроїв, вимоги NDA, політики віддаленого доступу тощо, а також дисциплінарні санкції за порушення вимог.

Після того як в компанії визначена політика інформаційної безпеки, а з нею основні вимоги до розробки, починається фактична розробка.

Життєвий цикл розробки програмного забезпечення (SDLC)[6] — це економічно ефективний процес, який використовується групами розробників для розробки та створення високоякісного програмного забезпечення. Метою SDLC є мінімізація проєктних ризиків шляхом перспективного планування, щоб програмне забезпечення відповідало очікуванням клієнтів під час виробництва та після нього. Ця методологія описує низку кроків, які поділяють процес розробки програмного забезпечення на завдання, які можна призначати, виконувати та вимірювати.

Управління розробкою програмного забезпечення може бути складним через мінливі вимоги, оновлення технологій та міжфункціональну співпрацю. Методологія життєвого циклу розробки програмного забезпечення (SDLC) забезпечує системну структуру управління з конкретними результатами на кожному етапі процесу розробки програмного забезпечення. В результаті всі зацікавлені сторони заздалегідь узгоджують цілі та вимоги до розробки програмного забезпечення, а також мають план досягнення цих цілей.

Переваги використання підходу SDLC:

- Підвищена прозорість процесу розробки для всіх зацікавлених сторін;
- Ефективне оцінювання, планування та складання розкладу;
- Покращене управління ризиками та оцінка витрат;
- Систематичне надання оновлень програмного забезпечення та краще задоволення потреб клієнтів.

Розглянемо основні кроки життєвого циклу розробки більш детально (рис.1.1.).

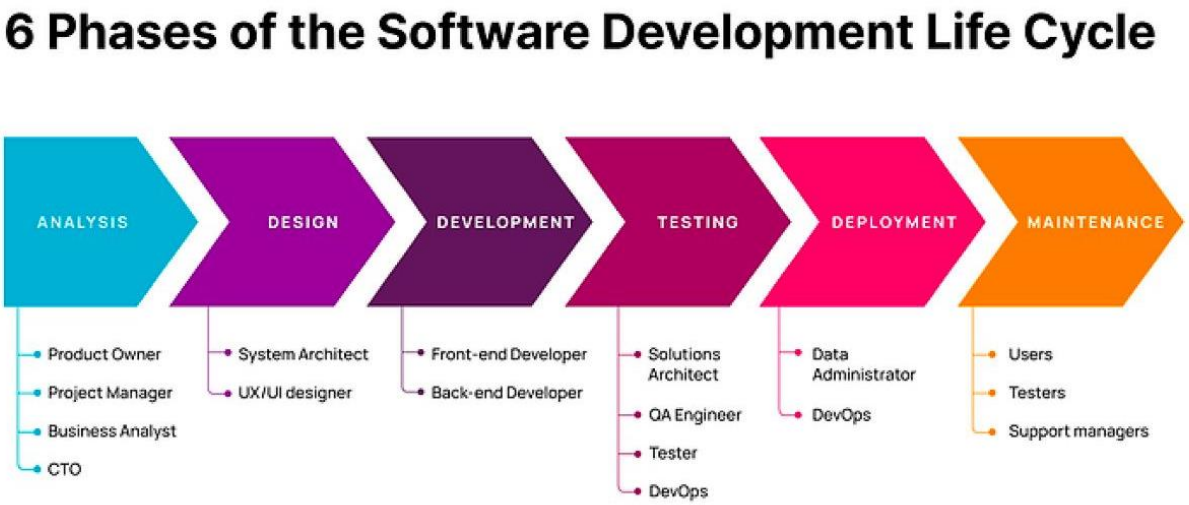

Рис. 1.1. Основні кроки SDLC [7]

Планування. Етап планування зазвичай включає такі завдання, як аналіз витрат і вигоди, планування, оцінка та розподіл ресурсів. Команда розробників збирає вимоги від кількох зацікавлених сторін, таких як замовники, внутрішні та зовнішні експерти та менеджери, щоб створити документ специфікації вимог до програмного забезпечення.

Документ встановлює очікування та визначає загальні цілі, які допомагають у плануванні проекту. Команда оцінює витрати, створює графік і має детальний план досягнення своїх цілей.

Дизайн. На етапі проектування інженери-програмісти аналізують вимоги та визначають найкращі рішення для створення програмного забезпечення. Наприклад, вони можуть розглянути можливість інтеграції вже існуючих модулів, зробити вибір технології та визначити інструменти розробки. Також розглядають, як найкраще інтегрувати нове програмне забезпечення в існуючу IT-інфраструктуру, яку може мати організація.

Розробка. На етапі реалізації команда розробників пише код продукт. Аналізуються вимоги, щоб визначити менші завдання з кодування, які можна розпланувати для щоденної розробки та досягнення кінцевого результату.

Тестування. Команда розробників поєднує автоматизацію та ручне тестування, щоб перевірити програмне забезпечення на наявність помилок. Аналіз

якості включає перевірку програмного забезпечення на наявність помилок і перевірку відповідності вимогам замовника. Оскільки багато команд одразу тестують код, який вони пишуть, етап тестування часто проходить паралельно з етапом розробки.

Впровадження. Коли команди розробляють програмне забезпечення, вони кодують і тестують на іншій копії програмного забезпечення, ніж та, до якої мають доступ користувачі. Програмне забезпечення, яке використовують користувачі, називається робочою версією, тоді як інші копії знаходяться в середовищі збірки, або середовищі тестування.

Наявність окремих середовищ збірки та виробництва гарантує, що клієнти можуть продовжувати користуватися програмним забезпеченням, навіть коли воно змінюється або оновлюється. Етап розгортання включає кілька завдань для перенесення останньої копії збірки у виробниче середовище, таких як пакування, конфігурація середовища та інсталяція.

Підтримка. На етапі обслуговування, серед інших завдань, команда виправляє помилки, вирішує проблеми клієнтів та керує змінами в програмному забезпеченні. Крім того, команда відстежує загальну продуктивність системи, безпеку та користувацький досвід, щоб визначити нові шляхи вдосконалення існуючого програмного забезпечення.

Різні моделі розташовують фази SDLC у різному хронологічному порядку для оптимізації циклу розробки. Популярними моделями SDLC є:

- *Водоспад*. Модель водоспаду розташовує всі фази послідовно так, що кожна нова фаза залежить від результату попередньої фази. Концептуально, дизайн перетікає з однієї фази в іншу, подібно до водоспаду. Модель водоспаду забезпечує дисципліну в управлінні проектом і дає відчутний результат в кінці кожної фази. Однак після того, як фаза вважається завершеною, залишається мало місця для змін, оскільки зміни можуть вплинути на час доставки програмного забезпечення, його вартість та якість. Тому ця модель найкраще підходить для невеликих проектів з розробки програмного забезпечення, де завдання легко організувати та керувати ними, а вимоги можуть бути точно визначені заздалегідь.

- *Ітеративний*. Ітеративний процес передбачає, що команди починають розробку програмного забезпечення з невеликого набору вимог. Потім вони ітеративно покращують версії з плином часу, поки повне програмне забезпечення не буде готове до виробництва. Наприкінці кожної ітерації команда створює нову версію програмного забезпечення. При даному підході легко виявляти та управляти ризиками, оскільки вимоги можуть змінюватися між ітераціями. Однак повторювані цикли можуть призвести до зміни обсягу робіт і недооцінки ресурсів.
- *Спіраль*. Спіральна модель поєднує невеликі повторювані цикли ітеративної моделі з лінійним послідовним потоком моделі водоспаду для визначення пріоритетів аналізу ризиків. Ви можете використовувати спіральну модель для забезпечення поступового випуску та вдосконалення програмного забезпечення шляхом створення прототипів на кожному етапі. Спіральна модель підходить для великих і складних проектів, які потребують частих змін. Однак вона може бути дорогою для невеликих проектів з обмеженою сферою застосування.
- *Гнучкий*. Гнучка модель розподіляє фази SDLC на кілька циклів розробки. Команда швидко проходить через фази, вносячи лише невеликі інкрементні зміни в програмне забезпечення в кожному циклі. Вони постійно оцінюють вимоги, плани та результати, щоб швидко реагувати на зміни. Гнучка модель є одночасно ітеративною та інкрементальною, що робить її більш ефективною, ніж інші моделі процесів. Швидкі цикли розробки допомагають командам виявляти та вирішувати проблеми в складних проектах на ранніх стадіях, до того, як вони стануть значними проблемами. Вони також можуть залучати клієнтів та зацікавлені сторони для отримання зворотного зв'язку протягом усього життєвого циклу проекту. Однак надмірна залежність від відгуків клієнтів може призвести до надмірних змін в обсязі або завершення проекту на півдорозі.

У традиційній розробці програмного забезпечення тестування безпеки було окремим процесом від життєвого циклу розробки програмного забезпечення (SDLC). Команда розробників виявляла недоліки безпеки лише після того, як вони створили програмне забезпечення. Це призводило до великої кількості багів, які залишалися прихованими, а також до збільшення ризиків безпеки.

DevOps (комбінація «розробки» та «операцій») — це поєднання практик та інструментів, призначених для підвищення здатності організації надавати програми та послуги швидше, ніж традиційні процеси розробки програмного забезпечення. Така швидкість дозволяє організаціям краще обслуговувати своїх клієнтів і ефективніше конкурувати на ринку.

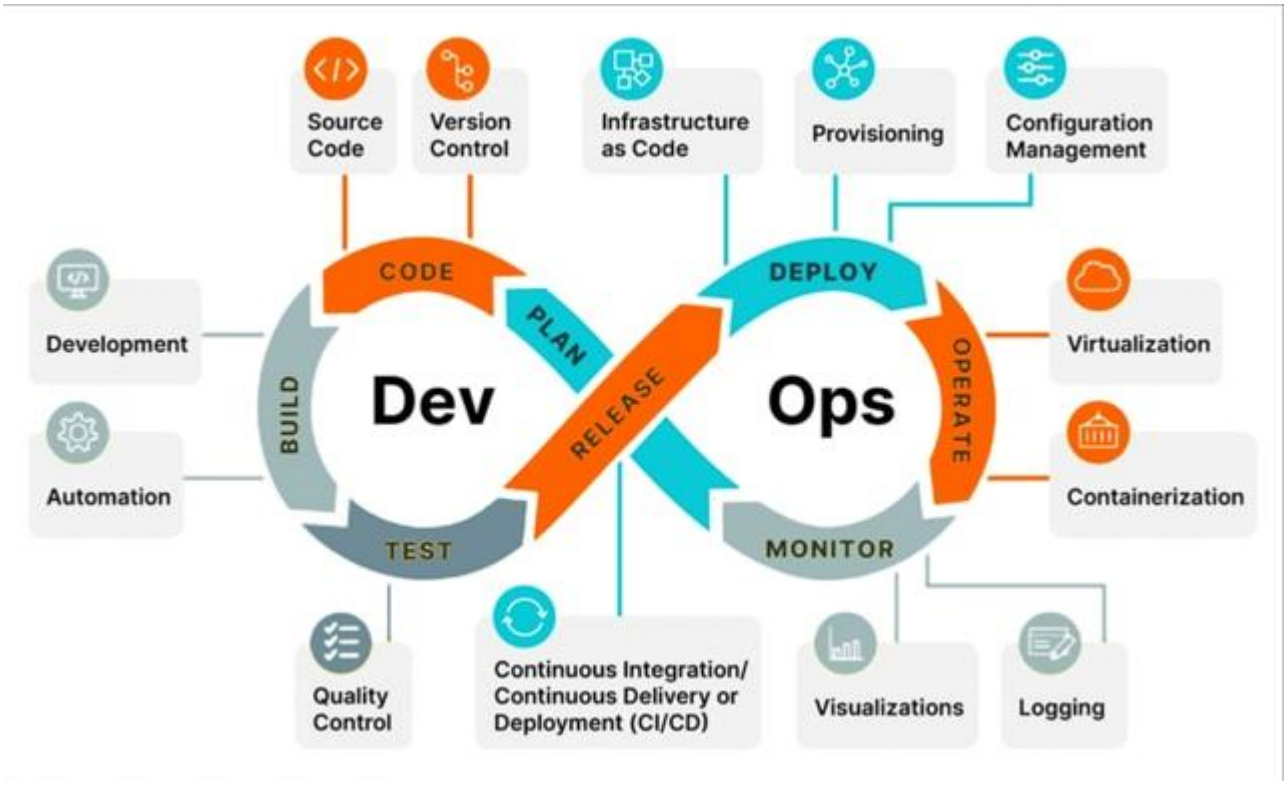

Рис. 1..2 Етапи розробки додатку при DevOps підході [8]

Робочий процес DevOps складається з етапів (рис. 1.2).

- Планування наступної ітерації розробки продукту;
- Побудова коду;
- Тестування та розгортання у робочому середовищі;
- Доставка оновлень продукту;
- Моніторинг і журнал продуктивності програмного забезпечення;
- Збір відгуків клієнтів.

Планування . Інструменти планування та відстеження завдань необхідні для того, щоб команди DevOps досягли безперебійного та ефективного циклу управління проектами та своєчасно випускали продукт. Популярні інструменти для

цього етапу включають Confluence та Jira.

Збірка та доставка. Розробникам потрібне швидке розгортання середовищ розробки та тестування, і вони не можуть довго чекати ремонту, коли щось піде не так. Контейнерізація дозволяє розбивати великі програми на менші окремі частини, що, у свою чергу, дозволяє оновлювати ці частини без необхідності повторного розгортання всієї програми. Розробники можуть просто замінити або оновити пошкоджений контейнер. Перехід до контейнеризації дозволив DevOps прискорити цикли розробки. Популярні інструменти для цього етапу включають Docker, Kubernetes і Terraform.

Тестування. Автоматизація тестування є важливим елементом впровадження DevOps, оскільки вона дозволяє використовувати інструменти автоматизації для запуску тестових сценаріїв і виконувати функціональне тестування способами, які не перешкоджають швидкості, яку досягли завдяки переходу на DevOps. Популярні інструменти для цього етапу включають Jenkins , CircleCI та GitLab CI.

Моніторинг та журналювання програмного забезпечення. Щойно програмне забезпечення переміщується у виробництво, його потрібно контролювати, щоб забезпечити стабільну продуктивність і підвищити задоволеність клієнтів. Цей етап також включає в себе аналіз ефективності та ведення журналів, створення інтелектуальних сповіщень про різні проблеми та збір відгуків клієнтів. Популярні інструменти для цього етапу включають Prometheus, Grafana, Elastic (ELK) Stack, Splunk і Sumo Logic.

Безпека DevOps, яку частіше називають *DevSecOps*, відноситься до дисципліни та практики захисту всього середовища DevOps за допомогою стратегій, політик, процесів і технологій. Філософія DevSecOps(рисунок 1.3) полягає в тому, що безпека повинна бути вбудована в кожну частину життєвого циклу DevOps, включаючи написання коду, компіляція, тестування, випуск, впровадження, підтримку, моніторинг тощо.

Рис. 1.3. Інтеграція безпеки на всіх етапах при підході DevSecOps [9]

Традиційна безпека ґрунтується на тому, що після розробки системи її дефекти безпеки можна визначити та виправити перед випуском. У моделі DevOps традиційні методи безпеки застосовуються надто пізно в циклі розробки та надто повільні для розробки та випуску програмного забезпечення, створеного шляхом ітерації.

DevSecOps включає в себе інструменти та процеси, які заохочують співпрацю між розробниками, фахівцями з безпеки та операційними командами для створення програмного забезпечення, яке може протистояти сучасним загрозам. Крім того, це гарантує, що заходи із забезпечення безпеки, такі як перегляд коду, аналіз архітектури та тестування на проникнення, є невід'ємною частиною процесу розробки.

Складнощі впровадження DevOps включають інтеграцію різноманітних технологій розробки та тестування, посилення конвеєрів розробки, щоб вони не порушувалися під час збільшення навантаження, і запровадження надійного тестування безпеки.

Середовища розробки складаються з безлічі фреймворків, мов і архітектур, кожна зі своїми унікальними способами взаємодії та роботи. Вирівняти всі рухомі частини одна з одною може бути складно.

Хоча автоматизація є ключем до стандартизації та централізації вашої

стратегії, політик, SLA та метрик ризику, усі ці взаємопов’язані компоненти можуть зробити вас вразливими до нестабільних конвеєрів, які можуть вийти з ладу або зупинитися. Коли одна з частин ламається або автоматизація не спрацьовує чи блокує процеси, доводиться зупиняти все, щоб знайти проблему та вирішення, перш ніж продовжити. Керування всіма цими подіями, що спрацьовують, і політиками, які їх регулюють, може нагадувати роботу з постійно увімкненим гальмом.

Оскільки ризики безпеки мобільних додатків можуть виникати на будь-якому етапі конвеєра, отримати повну видимість ризиків у такій різноманітності фреймворків, мов програмування та архітектур може бути складно як і управління ними.

CI/CD, що означає Continuous Integration and Continuous Delivery (або Continuous Deployment), — це практика розробки програмного забезпечення, що характеризується такими ключовими елементами:

- *Безперервна інтеграція*: ця практика передбачає часте об’єднання змін коду в спільне сховище. Після кожної інтеграції коду виконуються автоматичні тести, щоб перевірити якість коду та виявити потенційні проблеми;
- *Безперервна поставка*: ґрунтуючись на безперервній інтеграції, Безперервна доставка автоматизує процес підготовки коду для розгортання. Це гарантує, що код завжди в стані розгортання та може бути випущений у будь-який час;
- *Безперервне розгортання*: продовжуючи крок далі, безперервне розгортання автоматично розгортає зміни коду у робочому середовищі після того, як вони пройдуть автоматизовані тести та перевірку якості. Такий підхід виключає ручне втручання в процес розгортання.

Інструменти CI/CD автоматизують і підключають весь процес доставки програмного забезпечення. Вони створюють код, запускають тести та розгортають у середовищах узгодженим і повторюваним способом. Ця автоматизація забезпечує швидкий зворотний зв’язок, зменшує кількість помилок і забезпечує швидку доставку високоякісного програмного забезпечення.

Ключові переваги безперервної інтеграції та розвитку:

- Автоматизація збірки – усуває ручну роботу та прискорює цикли випуску.
- Автоматизація тестування – раннє виявлення дефектів.
- Автоматичне розгортання – Дозволяє відпускати за допомогою кнопки.
- Наскрізні конвеєри – об’єднують етапи збірки, тестування та розгортання.
- Аналітика/звіти – надають уявлення про процес доставки.

DevOps зосереджено на швидшій доставці програмного забезпечення шляхом інтеграції розробки та операцій. CI/CD є важливою практикою DevOps (рисунок 1.4), яка автоматизує інтеграцію та доставку змін коду.

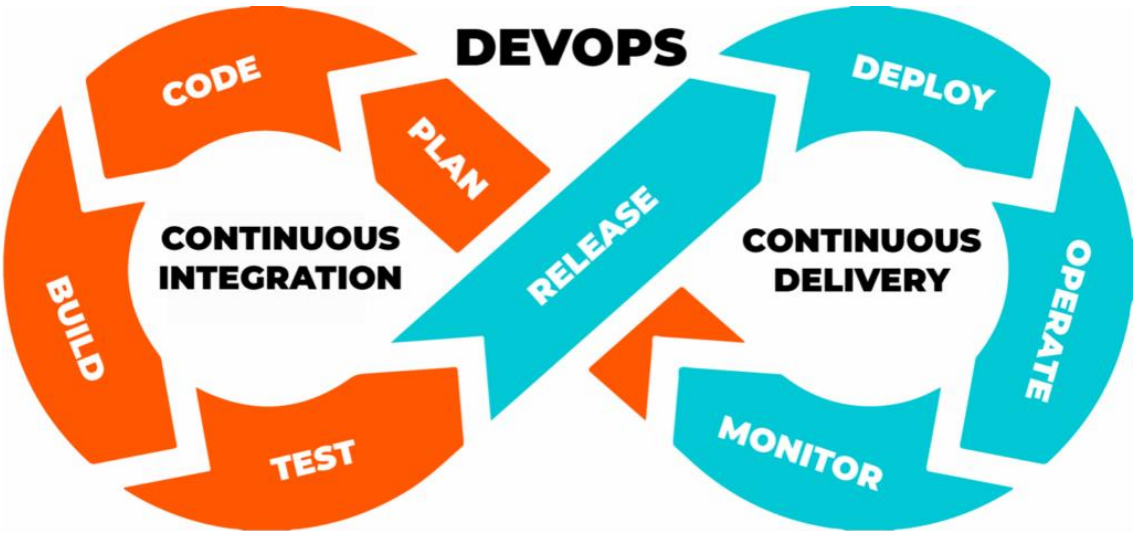

Рис. 1.4. Поєднання DevOps та CI/CD [10]

Інструменти CI/CD допомагають розробникам швидко створювати, тестувати та розгортати оновлення програм за допомогою автоматизованих конвеєрів. Замість окремих ручних процесів, CI/CD поєднує всі етапи – від фіксації до виробництва – у безперебійний робочий процес.

Така автоматизація прискорює цикли випуску, покращує якість за рахунок раннього виявлення дефектів і забезпечує надійне розгортання в одне натискання кнопки.

Конвеєри CI/CD також є критично важливими для DevSecOps, який впроваджує безпеку в практики DevOps. Сканування безпеки та елементи керування впроваджуються безпосередньо в конвеєр CI/CD DevOps. Це «зміщує ліворуч» безпеку на ранніх етапах життєвого циклу розробки, а не лише на етапі попереднього виробництва.

Конвеєр CI/CD — це автоматизований наскрізний робочий процес, який вносить зміни в код через повний цикл доставки програмного забезпечення. Конвеєр поєднує етапи безперервної інтеграції, тестування та розгортання, щоб забезпечити безперервну розробку, тестування та випуск оновлень програм без уповільнення швидкості.

Знаючи який підхід до розробки мобільного додатку використовується, ми можемо краще спрогнозувати можливі ризики та підготувати відповідні рекомендації щодо їх усунення або мінімізації за допомогою технологій безпеки та тестування.

1.3. Аналіз існуючих нормативних вимог та рекомендацій

Важливим етапом при підготовці рекомендацій щодо застосування технологій безпеки та тестування мобільних додатків є аналіз існуючих нормативних вимог та рекомендацій.

Для розробки у банківському секторі необхідно слідувати постановам Національного Банку України (НБУ). Основним збірником вимог з інформаційної безпеки, в тому числі вимог до розробки, є *Постанова НБУ №95*[11].

Постанови НБУ з безпеки є адаптацією сімейства міжнародних стандартів безпеки ISO 27000 для банківського сектору та реалій України. Тому дана постанова може бути цікава не тільки для банків і виступати орієнтиром та прикладом з організації безпеки для інших сфер. Також постанови НБУ є у відкритому доступі.

Міжнародний стандарт ISO/IEC 27001[12] - найвідоміший у світі стандарт для систем управління інформаційною безпекою (СУІБ) та вимог до них. Додаткові

найкращі практики захисту даних та кіберстійкості охоплюються більш ніж десятком стандартів сімейства ISO/IEC 27000. Разом вони дають змогу організаціям усіх галузей і розмірів управляти безпекою таких активів, як фінансова інформація, інтелектуальна власність, дані співробітників та інформація, довірена третіми сторонами. Міжнародні стандарти ISO/IEC надаються тільки на платній основі.

ISO/IEC 27002-2022 «Інформаційна безпека, кібербезпека та захист приватності - Контроль інформаційної безпеки»[13] містить практичні правила менеджменту інформаційної безпеки та є розширеною версією вимог та підходів ISO/IEC 27001. У рамках вивчення вимог та рекомендацій щодо безпечної розробки мобільного додатку, стандарт містить наступні розділи:

- 5.8 Інформаційна безпека в управлінні проектами;
- 8.25 Безпечний життєвий цикл розробки;
- 8.26 Вимоги до безпеки додатків;
- 8.28 Безпечне кодування.

Міжнародні стандарти та рекомендації від NIST[14] (Національний інститут стандартів і технологій, США). NIST це нерегуляторна державна установа, створена з метою стимулювання інновацій та підвищення промислової конкурентоспроможності в галузі науки, техніки та технологій.

Основна функція NIST полягає у створенні найкращих практик (також відомих як стандарти) для покращення стану безпеки державних установ та приватних компаній, що мають справу з державними даними. Вони також відомі під назвою NIST Cybersecurity Framework (CSF), що являє собою набір керівних принципів і передових практик, щодо вдосконалення стратегії кібербезпеки. Стандарти та рекомендації NIST надаються у відкритому доступі.

NIST SP 800-163 «Перевірка безпеки мобільних додатків»[15] охоплює стандарти та найкращі практики для розробки безпечних додатків (відповідно до їхнього передбачуваного використання), а також формулювання процедур для перевірки цих додатків. Надаються вказівки щодо вибору відповідних інструментів для оцінки та перевірки безпеки додатків, а також методи оцінки того, чи може

додаток бути безпечно розгорнутий на обладнанні організації кінцевого користувача.

Аналітика та рекомендації від OWASP. OWASP – це некомерційний фонд, який працює над покращенням безпеки програмного забезпечення. Також OWASP є відкритою спільнотою, яка допомагає організаціям створювати, розробляти, купувати, експлуатувати та підтримувати додатки, яким можна довіряти. Всі проекти, інструменти, документи, форуми та розділи є безкоштовними і відкритими для всіх, хто зацікавлений у підвищенні безпеки додатків.

Флагманський проект OWASP Mobile Application Security (MAS)[16] надає стандарт безпеки для мобільних додатків (OWASP MASVS)[17] і комплексний посібник з тестування (OWASP MASTG)[18], який охоплює процеси, методи та інструменти, що використовуються під час тестування безпеки мобільних додатків, а також вичерпний набір тестових кейсів, що дозволяє тестувальникам отримувати послідовні та повні результати.

Стандарт OWASP MASVS розділений на різні групи елементів управління, позначені MASVS-XXXXX, які представляють найбільш критичні ділянки мобільної поверхні атаки:

- MASVS- STORAGE: Безпечне зберігання конфіденційних даних на пристрої (дані при зберіганні).
- MASVS-CRYPTO: Криптографічна функціональність, що використовується для захисту конфіденційних даних.
- MASVS-AUTH: Механізми аутентифікації та авторизації, що використовуються мобільним додатком.
- MASVS-NETWORK: Безпечний мережевий зв'язок між мобільним додатком і віддаленими кінцевими точками (дані при передачі).
- MASVS- PLATFORM: Безпечна взаємодія з базовою мобільною платформою та іншими встановленими додатками.
- MASVS-CODE: Найкращі практики безпеки для обробки даних та підтримання актуальності програми.
- MASVS- RESILIENCE: Стійкість до спроб зворотного інжинірингу та

втручання.

- MASVS-PRIVACY: Контроль конфіденційності для захисту приватності користувачів.

Посібник з тестування безпеки мобільних додатків OWASP (MASTG) - це всеосяжний посібник з тестування безпеки мобільних додатків та зворотного інжинірингу. Він описує технічні процеси для перевірки засобів контролю, перелічених в OWASP MASVS, через слабкі місця, визначені OWASP MASWE[19].

Перелік вразливостей безпеки мобільних додатків (Mobile Application Security Weakness Enumeration, MASWE) - це список найпоширеніших вразливостей безпеки та конфіденційності в мобільних додатках. Він призначений для використання в якості довідника для розробників, дослідників безпеки та фахівців з безпеки. Він є сполучною ланкою між MASVS та MASTG.

MASWE має на меті доповнити CWE, зосередивши увагу на слабких місцях безпеки в мобільних додатках.

Слабке місце - це проблема безпеки або конфіденційності, яка може бути впроваджена в мобільний додаток. Слабкі місця класифікуються за категоріями MASVS та засобами контролю. Наприклад, вразливість, пов'язана з використанням незахищених генераторів випадкових чисел, відноситься до категорії MASVS-CRYPTO-1.

Кожна вразливість містить наступну інформацію:

- Огляд: Короткий опис вразливості.
- Вплив: Потенційний вплив вразливості на безпеку або конфіденційність програми.
- Способи впровадження: Способи, якими вразливість може бути впроваджена в програму.
- Зменшення вразливостей: Рекомендації щодо усунення вразливості.

Чеклист безпеки мобільних додатків OWASP[20] містить посилання на тестові кейси MASTG для кожного елементу управління MASVS:

- Оцінки безпеки/Пентести: переконатися, що покривається принаймні стандартна поверхня атак, і початок дослідження.

- Відповідність стандартам: включає версії MASVS і MASTG та ідентифікатори коммітів.
- Вивчення та практика навичок мобільної безпеки.
- Винагороди за виправлення помилок: крок за кроком покриття поверхні мобільних атак.

На сайті доступні до завантаження згаданий стандарт безпеки OWASP MASVS, посібник з тестування OWASP MASTG, beta-доступ до ресурсу з Переліком вразливостей безпеки мобільних додатків (MASWE) та чекліст для перевірки відповідності стандарту.

Також у процесі формування рекомендацій паралельно для аналізу можуть використовуватися документація популярних інструментів для тестування та захисту, сучасні практики DevSecOps та професійні блоги, ресурси з кібербезпеки.

Висновки до 1 розділу

У цьому розділі було проведено аналіз проблеми забезпечення захисту мобільних додатків. Було детально розглянуто життєвий цикл розробки (SDLC), основні підходи до організації процесу розробки такі як DevOps, DevSecOps, CI/CD та різницю між ними. Це дало змогу виявити особливості які необхідно враховувати при інтеграції безпеки в процес розробки. Також були розглянути сучасні вимоги до безпеки мобільних додатків, що включають в себе захист від зовнішніх загроз та забезпечення конфіденційності користувачів. Висновки цього розділу підкреслюють необхідність комплексного підходу до розробки та тестування мобільних додатків, де безпека виступає як один з основних пріоритетів.

2 АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ ЗАХИСТУ ТА ТЕСТУВАННЯ МОБІЛЬНИХ ДОДАТКІВ

2.1. Захист під час розробки мобільного додатку

Як правило, організації проводять моделювання загроз на етапі проектування (але це може відбуватися і на інших етапах) нового додатку, щоб допомогти розробникам знайти вразливі місця, дізнатися про наслідки для безпеки їхніх проектів, коду та конфігураційних рішень та врахувати їх при розробці.

Моделювання загроз – це структурований підхід, спрямований на виявлення та визначення пріоритетів потенційних загроз і вразливостей у додатках. Він включає виявлення потенційних зловмисників, їх мотивації та методи, які вони можуть використовувати для використання вразливостей у системі. Мета полягає в тому, щоб визначити потенційні ризики безпеки на ранніх етапах життєвого циклу розробки програмного забезпечення (SDLC), щоб їх можна було вирішити до розгортання програмного забезпечення.

Виконуючи моделювання загроз, організації виконують ретельний аналіз архітектури програмного забезпечення, бізнес-контексту та інших артефактів (наприклад, функціональних специфікацій, документації користувача). Цей процес дозволяє глибше зрозуміти та відкрити важливі аспекти системи.

Зазвичай розробники виконують моделювання загроз у чотири етапи.

- *Діаграма*. Що ми будуємо?
- *Визначити загрози*. Що може піти не так?
- *Пом'якшити*. Що ми робимо для захисту від загроз?
- *Перевірити*. Чи виконали ми кожен з попередніх кроків?

При правильному виконанні, моделювання загроз може забезпечити чітку лінію видимості в проекті програмного забезпечення, допомагаючи перевірити заходи безпеки. Процес моделювання загроз допомагає організації документувати загрози безпеці програми та приймати раціональні рішення щодо їх усунення.

Моделювання загроз сприяє розумінню безпеки в усій команді. Це перший крок до того, щоб безпека стала відповідальністю кожного. Концептуально моделювання загроз є простим процесом. Основні рекомендації під час створення або оновлення моделі загроз:

- *Визначте обсяг і глибину аналізу*. Визначте обсяг із зацікавленими сторонами, а потім розділіть глибину аналізу для окремих груп розробників, щоб вони могли моделювати загрози для програмного забезпечення.
- *Створіть візуальну інтерпретацію зібраних даних*. Створіть діаграму основних компонентів системи (наприклад, сервер додатків, сховище даних, товстий клієнт, база даних) і взаємодії між цими компонентами.
- *Застосуйте контекст безпеки до створеної діаграми*. Визначте активи програмного забезпечення, елементи керування безпекою та агенти загроз, а також зобразіть їх розташування, щоб створити модель безпеки системи. Змоделювавши систему, ви зможете визначити, що може піти не так.
- *Визначте загрози*. Зверніться до моделей загроз, таких як STRIDE, capec, att&ck тощо. Задокументуйте можливості, складіть список потенційних атак і поставте такі запитання, як
    - Що може зробити зловмисник?
    - Який тип доступу вони мають?
    - Де вони розташовані?
    - Чи є шляхи, якими агент загрози може дістатися до активу, не проходячи контроль?
    - Чи може агент загрози подолати цей контроль безпеки?
    - Що повинен зробити агент загрози, щоб здолати цей контроль?
    - Які можуть бути різні потенційні сценарії та вектори атак?
- *Створіть матрицю простежуваності*. Задокументуйте перелік цілей, яких зловмисник може захотіти досягти, разом із тим, як ці цілі можна досягти. Також додайте список заходів безпеки, щоб запобігти ризику.

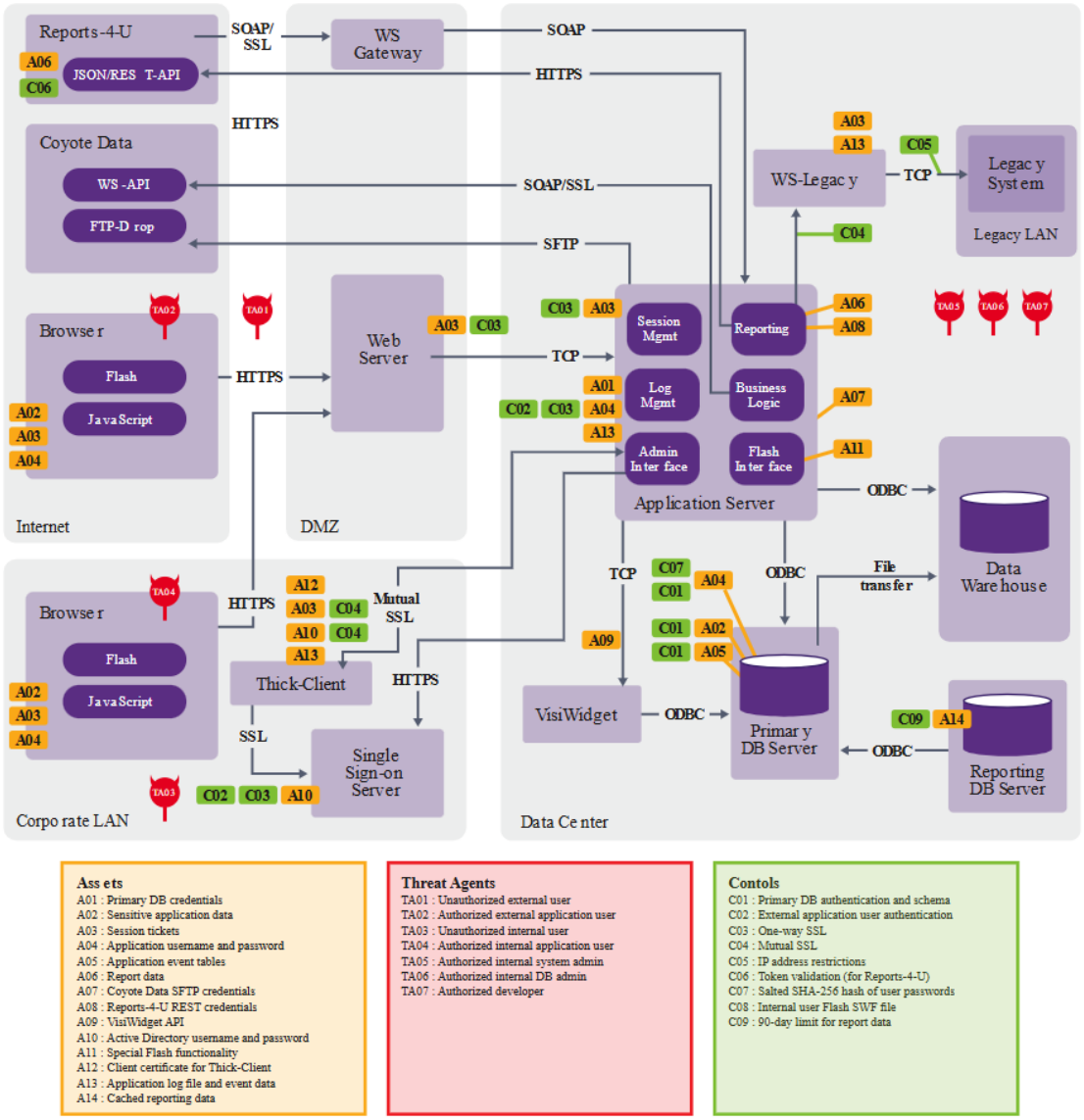

Рис. 2.1. Приклад моделі безпеки системи [21]

Під час побудови моделі загроз (рис. 2.1) також враховується обраний стек технологій, який був обраний на базі вимог до продукту, галузі та місця для зберігання та збору даних. Це рішення важливе, оскільки вибір неправильного стека технологій, який не відповідає вимогам, може відкинути команду розробників назад до початку розробки. В інших випадках розробники можуть змінити шлях розробки прямо посередині процесу, оскільки вже написаний код не відповідатиме вимогам безпеки.

Технологічний стек (рис. 2.2) — це комбінація програмних інструментів і мов програмування, які використовуються для втілення в життя веб- або мобільного додатку. Веб-програми та програми для мобільних пристроїв складаються з

інтерфейсу та серверної частини, які є клієнтською програмою та прихованою частиною на сервері відповідно.

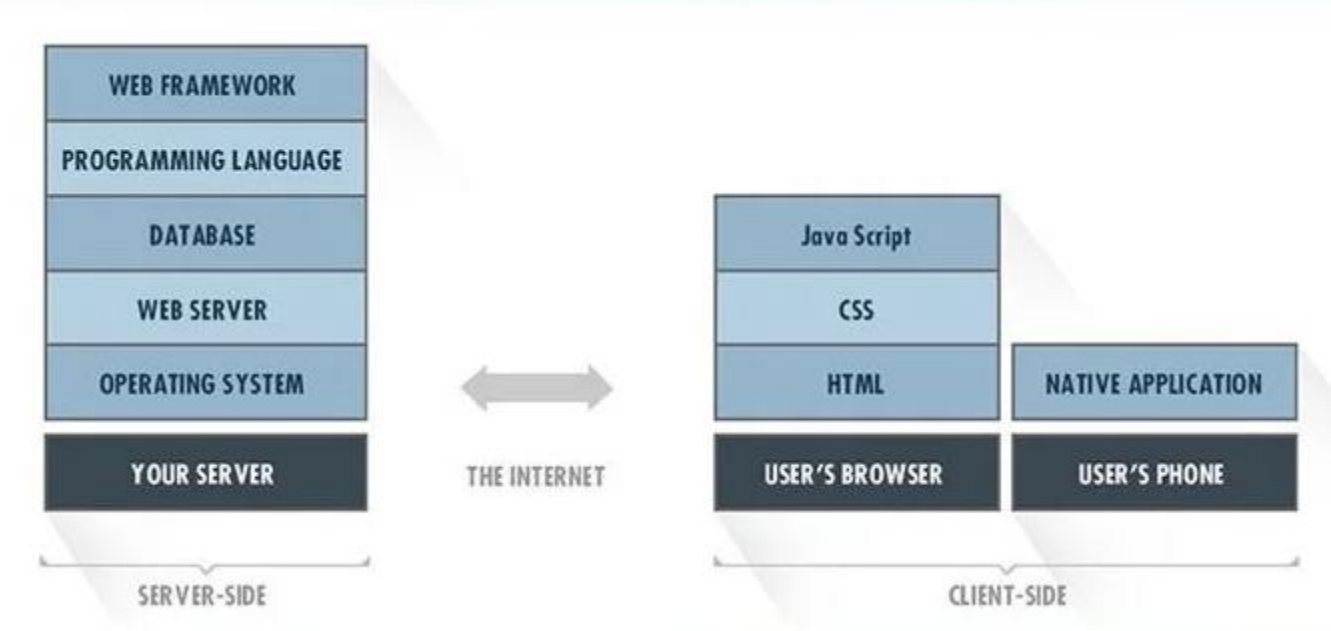

Рис. 2.2. Приклад стеку технологій програми [22]

Кожен рівень програми побудовано поверх нижнього, утворюючи стек. Це робить технології веб-стеку сильно залежними одна від одної. На зображенні вище показано основні будівельні блоки типового технологічного стеку; однак можуть бути задіяні інші допоміжні елементи.

Часто при розробці додатку розробники використовують відкритий код і при цьому повинні знати про ліцензійні обмеження та зобов'язання. Відстеження цих зобов'язань вручну стало надто важким завданням, часто не помічали наявність відкритого коду і його супутні вразливості. Щоб запобігти подібним ризикам було розроблено автоматизоване рішення SCA, яке з початкового варіанту використання розширилося для аналізу безпеки та якості коду.

Аналіз складу програмного забезпечення (SCA) – це автоматизований процес, який ідентифікує програмне забезпечення з відкритим кодом у кодовій базі. Цей аналіз виконується для оцінки безпеки, відповідності ліцензії та якості коду.

Інструменти SCA перевіряють менеджери пакетів, файли маніфестів, вихідний код, двійкові файли, зображення контейнерів тощо. Виявлене відкрите джерело компілюється в опис матеріалів (BOM), який потім порівнюється з

різними базами даних, включаючи Національну базу даних уразливостей (NVD)[23].

Ці бази даних містять інформацію про відомі та поширені вразливості. NVD є сховищем вразливостей уряду США.

Прикладами рішень SCA є: Snyk Open Source, Veracode, Black Duck Software Composition Analysis, Mend, GitLab, Nexus Repository та інші.

OWASP Чеклист безпечних практик кодування[24] - документ визначає набір загальних практик кодування безпеки програмного забезпечення у форматі чеклисту, який можна інтегрувати в життєвий цикл розробки програмного забезпечення. Застосування цих практик зменшить більшість поширених вразливостей програмного забезпечення.

Використовуючи цей посібник, команди розробників повинні почати з оцінки зрілості свого життєвого циклу розробки безпечного програмного забезпечення та рівня знань свого персоналу розробників. Оскільки цей посібник не охоплює деталей того, як реалізувати кожну практику кодування, розробникам потрібно або мати попередні знання, або достатньо ресурсів, які надають необхідні вказівки. У цьому посібнику наведено методи кодування, які можна перевести у вимоги до кодування без необхідності, щоб розробник мав глибоке розуміння вразливостей системи безпеки та експлойтів. Однак інші члени групи розробників повинні нести відповідальність, відповідну підготовку, інструменти та ресурси, щоб підтвердити, що дизайн і впровадження всієї системи безпечні.

Чеклист включає такі пункти як:

- Перевірка вхідних даних;
- Вихідне кодування;
- Автентифікація та керування паролями;
- Керування сеансами;
- Контроль доступу;
- Криптографічні методи;
- Обробка помилок та ведення журналу;
- Захист даних;

- Безпека зв'язку;
- Конфігурація системи;
- Безпека баз даних;
- Керування файлами;
- Керування пам'яттю;
- Загальні практики кодування.

Також OWASP публікує OWASP Top-10 що є визнаною світовою методологією оцінки вразливостей веб-додатків у всьому світі і відображає сучасні тренди безпеки веб-додатків, є першим кроком організації до створення культури більш безпечного коду програмного забезпечення.

На момент написання кваліфікаційної роботи OWASP збирають дані від компаній-користувачів і планують випуск OWASP Top 10:2025[25] у першій половині 2025 року. Останнім актуальним на момент написання є версія 2021-го року. Порівняно з версією 2017-го року з’явилися три нові категорії, чотири категорії зі змінами в назві та охопленні.

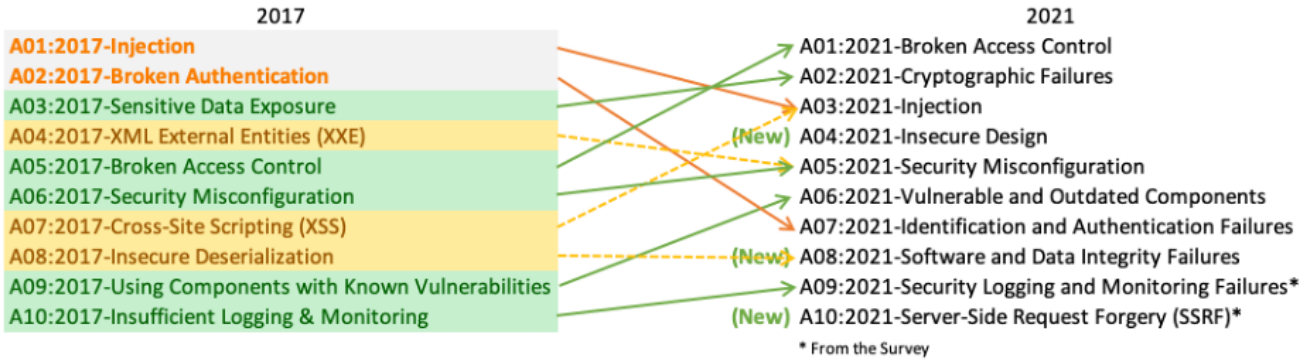

Рис. 2.3. OWASP Top 10:2021 порівняно до минулої версії [25]

Тож 10 найбільших ризиків для безпеки веб-додатків за версією OWASP:2021 порівняно з 2017 роком (рисунок 2.1.3) є:

- *A01:2021-Порушений контроль доступу* піднімається з п'ятої позиції; 94% додатків було перевірено на певну форму зламаного контролю доступу. 34 CWE, зіставлені з порушенням контролю доступу, зустрічались частіше у програмах, ніж будь-яка інша категорія.
- *A02:2021-Криптографічні збої* зміщуються на одну позицію вгору до

другого місця, раніше відомі як розкриття конфіденційних даних, що було загальним симптомом, а не основною причиною. Тут зосереджені збої, пов'язані з криптографією, які часто призводять до розкриття конфіденційних даних або компрометації системи.

- *A03:2021-Ін'єкції* зсувається на третю позицію. 94% додатків було перевірено на певну форму ін'єкції, і 33 CWE, віднесені до цієї категорії, займають друге місце за кількістю випадків у додатках.
- *A04:2021-Незахищений дизайн* – це нова категорія для 2021 року, яка зосереджена на ризиках, пов'язаних із недоліками дизайну. Якщо розробники прагнуть «рухатися ліворуч» як галузь, це потребує більшого використання моделювання загроз, безпечних шаблонів і принципів проектування та еталонних архітектур.
- *A05:2021-Неправильна конфігурація безпеки* переміщається з шостої позиції у попередньому виданні; 90% додатків перевірено на наявність неправильної конфігурації. Зі збільшенням кількості переходів до програмного забезпечення з широкими можливостями налаштування не дивно, що ця категорія просувається вгору. Колишня категорія зовнішніх сутностей XML (XXE) тепер є частиною цієї категорії.
- *A06:2021-Вразливі та застарілі компоненти* раніше називався «Використання компонентів із відомими вразливими місцями» і займав 2 місце в опитуванні спільноти, але також мав достатньо даних, щоб потрапити до топ 10 за результатами аналізу даних. Ця категорія піднялася з дев'ятого місця у 2017 році, і це відома проблема, яку OWASP намагається перевірити та оцінити ризик.
- *A07:2021-Помилки ідентифікації та автентифікації* раніше мав назву «Порушена автентифікація» та займав другу позицію, а тепер включає CWE, які більше пов'язані з помилками ідентифікації.
- *A08:2021-Порушення цілісності програмного забезпечення та даних* — це нова категорія для 2021 року, яка зосереджується на припущеннях щодо оновлень програмного забезпечення, критичних даних і конвеєрів CI/CD без перевірки цілісності.

- *A09:2021-Помилки в журналі безпеки та моніторингу* раніше мав назву «Недостатнє журналювання та моніторинг» та десяту позицію. Ця категорія розширена, щоб охопити більше типів збоїв, її важко перевірити, і вона недостатньо представлена в даних CVE/CVSS. Однак збої в цій категорії можуть безпосередньо вплинути на видимість, попередження про інциденти та криміналістику.
- *A10:2021- Підробка запитів на стороні сервера* додано з опитування спільноти. Дані свідчать про відносно низький рівень знаходження з охопленням тестування вище середнього, а також вищими за середні оцінками потенціалу використання та впливу. Ця категорія представляє сценарій, коли учасники спільноти безпеки повідомляють нам, що це важливо, навіть якщо на даний момент це не показано в даних.

Також OWASP публікує схожий топ 10 ризиків для мобільних пристроїв на базі Android та iOS.

2.2. Захист на етапі тестування

Тестування безпеки мобільного додатка передбачає тестування мобільного додатка таким чином, щоб зловмисний користувач спробував його атакувати. Ефективне тестування безпеки починається з розуміння бізнес-цілей програми та типів даних, які вона обробляє. Звідси комбінація статичного аналізу , динамічного аналізу та тестування на проникнення дає результати в ефективну цілісну оцінку для пошуку вразливостей, які були б упущені, якби ці методи не використовувалися разом ефективно. Процес тестування включає:

- Взаємодію з програмою та розуміння того, як вона зберігає, отримує та передає дані;
- Розшифровку зашифрованих частин програми;
- Декомпіляцію програми та аналіз отриманого коду;
- Використання статичного аналізу для визначення слабких місць безпеки в декомпільованому коді;
- Застосування знань, отриманих під час зворотного проектування та

статичного аналізу, для динамічного аналізу та тестування на проникнення;

- Використання динамічного аналізу та тестування на проникнення для оцінки ефективності елементів керування безпекою (наприклад, елементів керування автентифікацією та авторизацією), які використовуються в програмі.

Статичне тестування безпеки додатків (SAST) або статичний аналіз — це методологія тестування, яка аналізує вихідний код для виявлення вразливостей безпеки, які роблять програми організації сприйнятливими до атак. SAST сканує програму перед компіляцією коду. Це також відоме як тестування білого ящика.

SAST проводиться на дуже ранньому етапі життєвого циклу розробки програмного забезпечення (SDLC), оскільки для нього не потрібна працююча програма і може відбуватися без виконання коду. Інструменти SAST надають розробникам зворотній зв’язок у режимі реального часу під час кодування, допомагаючи їм виправляти проблеми, перш ніж вони передадуть код на наступний етап SDLC. Це запобігає тому, що проблеми, пов’язані з безпекою, не розглядатимуться після випуску релізу. Інструменти SAST надають графічне представлення виявлених проблем від джерела до поглинача, вказують на точне розташування вразливостей, виділяють ризикований код, та можуть надавати докладні вказівки щодо закриття вразливості і найкраще місце в коді для їх вирішення, не вимагаючи глибоких знань у сфері безпеки.

Розробники також можуть створювати налаштовані їм звіти за допомогою інструментів SAST; ці звіти можна експортувати офлайн і відстежувати за допомогою інформаційних панелей. Упорядковане відстеження всіх проблем безпеки, про які повідомляє інструмент, може допомогти розробникам швидко усунути ці проблеми та випустити програми з мінімальними проблемами. Цей процес сприяє створенню безпечного SDLC.

Важливо зауважити, що інструменти SAST необхідно запускати в програмі на регулярній основі, наприклад під час щоденних/щомісячних збірок, кожного разу, коли перевіряється код, або під час випуску коду.

Прикладами рішень SAST є: Snyk Code, Fortify, Checkmarx, Synopsys, Veracode та інші.

Динамічне тестування безпеки додатків (DAST) — це метод тестування безпеки додатків, за якого тестувальники перевіряють застосунок під час його роботи, але не мають знань про внутрішню взаємодію додатку чи дизайну на системному рівні, а також не мають доступу чи видимості вихідної програми. Це тестування «чорної скриньки» розглядає програму ззовні всередину, перевіряє її запущений стан і спостерігає за її реакцією на імітовані атаки, зроблені інструментом тестування. Відповіді програми на ці симуляції допомагають визначити, чи є програма вразливою та чи може бути чутливою до справжньої зловмисної атаки.

DAST працює, імітуючи автоматизовані атаки на програму, імітуючи зловмисника. Мета полягає в тому, щоб знайти результати або результати, які не очікувалися і тому могли бути використані зловмисниками для компрометації програми. Оскільки інструменти DAST не мають внутрішньої інформації про програму чи вихідний код, вони атакують так само, як і зовнішній хакер, маючи ті самі обмежені знання та інформацію про програму.

Прикладами рішень DAST є: Synopsys, Snyk, Veracode, Checkmarx, Invicti, Acunetix та інші.

Існує низка безкоштовних і комерційних інструментів безпеки мобільних додатків, які оцінюють додатки за допомогою методологій статичного або динамічного тестування з різним ступенем ефективності. Однак жоден інструмент не забезпечує комплексної оцінки програми. Щоб забезпечити найкраще покриття, потрібна комбінація як статичного, так і динамічного тестування з переглядом вручну.

Для підвищення ефективності використання SAST та DAST рішень, рекомендують їх одразу інтегрувати в процес розробки додатку.

Прикладом таких рішень, які дозволяють поєднати безпеку та розробку в рамках однієї платформи це GitLab CI/CD з інтегрованими SAST/DAST та Jenkins.

IAST (Interactive Application Security Testing) – це метод тестування безпеки додатків, який тестує додаток під час його запуску автоматизованим тестом, людиною-тестувальником або будь-якою іншою активністю, що «взаємодіє» з

функціоналом додатку.

Ядром інструменту IAST є сенсорні модулі, програмні бібліотеки, включені в код програми. Ці сенсорні модулі відстежують поведінку програми під час виконання інтерактивних тестів. Якщо буде виявлено вразливість, буде надіслано сповіщення.

Процес і зворотній зв'язок відбуваються в режимі реального часу у інтегрованому середовищі розробки (IDE), середовищі безперервної інтеграції (CI), забезпеченні якості або під час виробництва. Датчики мають доступ до:

- Всього коду;
- Потоку даних і потоку керування;
- Даних конфігурації системи;
- Веб-компонентів;
- Внутрішніх даних про з'єднання.

Прикладами таких вразливостей можуть бути жорстке кодування ключів API відкритим текстом, відсутність санітарної обробки даних, що вводяться користувачами, або використання з'єднань без SSL-шифрування.

Прикладами рішень IAST є: Acunetix, Veracode та інші.

Тестування API для мобільних пристроїв — це критично важливий процес, який допомагає мобільним програмам ефективно працювати з серверами, базами даних та іншими службами через API (інтерфейс програмування додатків). Мобільні програми використовують API для об’єднання інформації, надсилання даних і виконання багатьох важливих операцій. Таким чином, ці взаємодії повинні бути перевірені, щоб працювати ефективно та безпечно. Тестування мобільного API перевіряє, чи кінцеві точки правильно надають відповідну відповідь.

Прикладами рішень тестування API є: Invicti, Cloudflare,

Тестування мобільного додатку на проникнення – це процес визначення вразливостей у стані кібербезпеки мобільного додатка iOS або Android шляхом стимулювання атак у реальному житті. Мета полягає в тому, щоб проаналізувати, визначити пріоритети та виправити вразливості, перш ніж вони будуть зловмисно використані хакерами або ботами.

Це допомагає підвищити рівень безпеки для конфіденційних даних і різних функцій програми, щоб забезпечити добре захищену програму, яка захищає користувачів і адміністраторів. Під час цієї процедури перевіряються код, архітектура, сховище даних, підключення до мережі та методи автентифікації.

Тестування на проникнення можуть проводити як окрема команда всередині компанії так і зовнішні компанії.

Методи тестування на проникнення:

- *Тестування чорного ящика* проводиться з точки зору користувача, щоб перевірити функціональні та нефункціональні аспекти роботи програми. Цей тип тестування здійснюється шляхом прямої взаємодії з інтерфейсом. Тестери не знають, як система працює зсередини, і шукають проблеми та вразливості, перевіряючи вручну всі кнопки, форми, переходи тощо.
- *Тестування білого ящика* — це тестування зі знанням внутрішньої структури програми. Тестувальники враховують код програми, архітектуру та конфігурації. На основі цього вони аналізують очікувану відповідь на кожну команду та фактичний результат. Таким чином оцінюється коректна робота програми та виведення даних.
- *Тестування сірого ящика* - це комбінація методів чорного ящика та білого ящика. У тестуванні сірого ящика тестувальники мають мінімальну кількість інформації про внутрішню роботу програми та оцінюють її з точки зору користувача. Таким чином, вони можуть ефективно перевірити функціональність програми.

Зовнішнє тестування на проникнення поєднують в собі:

- Статичний аналіз мобільних додатків (SAST);
- Розвідку з відкритим кодом (OSINT) - збір загальнодоступної інформації про додаток, його творців та інфраструктуру, яка його підтримує, та навіть обговорення в соціальних мережах, форуми розробників і списки додатків;
- Аналіз трафіку мобільної мережі - вивчення мережевого трафіку, створеного програмою, допомагає визначити протоколи передачі даних (HTTPS проти HTTP), кінцеві точки, з якими здійснюється зв’язок, і потенційно

конфіденційну передачу даних;

- Динамічний аналіз мобільних додатків (DAST);
- Аналіз архітектури - неправильно налаштована політика безпеки, слабка автентифікація та авторизація, незахищене зберігання даних;
- Зворотне проектування - розбирання коду програми, щоб зрозуміти її внутрішню роботу та виявити приховані функції чи заплутану логіку, яка може містити вразливі місця. Цей процес схожий на декомпіляцію програми, щоб зрозуміти її логіку та функціональність на більш глибокому рівні;
- Аналіз файлової системи - мобільні програми часто зберігають дані локально на пристрої для таких функцій, як офлайн-доступ або налаштування користувача. Перевірка локального сховища програми на наявність конфіденційних даних, які можуть бути захищені неналежним чином або доступні для неавторизованих програм;
- Аналіз взаємодії між програмами (IAC) - дослідження того, як програма взаємодіє з іншими програмами на пристрої, зокрема механізми обміну даними та потенційні вразливості, якими можна скористатися для отримання доступу до даних або функцій інших програм;
- Експлуатація - на основі вразливостей, виявлених командою раніше, фаза експлуатації моделює атаки в реальному світі, такі як зловмисне корисне навантаження, як-от експлойти оболонки/кореневого доступу, щоб зрозуміти, як вони поводитимуться, якщо та коли станеться атака;
- Звітування та повторне сканування - після завершення етапу експлуатації команда готує детальний звіт про здійснені атаки, та знайдені вразливості. Крім пентесту, рекомендують також розглянути можливість налаштування періодичного повторного сканування після виправлення виявлених вразливостей. Це допомагає забезпечити ефективність заходів із відновлення, а також допомагає виявити й усунути будь-які потенційні тривалі проблеми.

На GitHub виклали у відкритому доступі Шпаргалку з тестування мобільних додатків на проникнення[26], якою також можна скористатись при проведенні тестів на проникнення.

Приклади зовнішніх команд з проведення тестів на проникнення: Integrity360, Astra, LX, LRQA та інші.

2.3. Захист на етапі розгортання додатку

Розгортання мобільного додатка в захищеній інфраструктурі є ключовим етапом для забезпечення його безпеки. Це включає впровадження заходів спрямованих на захист серверів, баз даних, API та інших компонентів, що підтримують роботу додатка.

Основні аспекти безпеки середовища на які необхідно звертати увагу:

- *Використання безпечних хмарних платформ*. Вибір хмарних провайдерів, які відповідають міжнародним стандартам безпеки, таким як ISO 27001, SOC 2, або PCI DSS. Наприклад, AWS, Google Cloud, Microsoft Azure пропонують рішення з вбудованими засобами безпеки. Використання сервісів хмарної безпеки, таких як AWS Shield або Azure Security Center, для захисту від DDoS-атак і моніторингу загроз.
- *Сегментація інфраструктури*. Розділення робочих навантажень через окремі віртуальні мережі, щоб зменшити вплив потенційного компрометування. Налаштування сегментації даних та доступу до сервісів за принципом найменших привілеїв (Least Privilege Principle).
- *Захищений мережевий доступ*. Використання брандмауерів для контролю мережевого трафіку, шифрування всіх переданих даних за допомогою TLS/SSL, регулярне оновлення сертифікатів шифрування та використання віртуальних приватних мереж (VPN) для захисту адміністративного доступу до серверів.
- *Контроль автентифікації та доступу*. Використання багатофакторної автентифікації (MFA) для всіх користувачів та адміністраторів інфраструктури та сервісів управління ідентичностями, таких як AWS IAM або Azure Active Directory, для забезпечення точного контролю доступу.
- *Захист API*. Захист API додатку за допомогою OAuth 2.0, OpenID

Connect, або інших стандартів автентифікації, використаннч рейтингу обмеження запитів (Rate Limiting), щоб запобігти атакам типу Brute Force чи DoS та регулярне тестування API за допомогою спеціалізованих інструментів, таких як Postman або OWASP ZAP.

- *Інтеграція засобів моніторингу*. Встановлення систем моніторингу, таких як Prometheus, Datadog, або Splunk, для виявлення та усунення аномальної активності в реальному часі та використання SIEM (Security Information and Event Management), наприклад, Splunk або IBM QRadar, для аналізу логів і виявлення підозрілих дій.
- *Ізоляція середовищ*. Розділення середовищ розробки (Development), тестування (Testing), і продакшну (Production), заборона доступу тестовим обліковим записам і конфігураціям у продакшн-середовищі.
- *Резервне копіювання та відновлення*. Регулярне створення резервних копій даних у захищених місцях та впровадження політики швидкого відновлення після збоїв (Disaster Recovery Plans).
- *Захист серверів*. Автоматичне оновлення серверного ПЗ та операційних систем, використання механізмів захисту від шкідливого ПЗ, таких як встановлення Endpoint Protection та налаштування автоматичного аудиту серверів для перевірки відповідності стандартам безпеки.
- *Аудит і відповідність стандартам*. Регулярна перевірка відповідності системи стандартам, наприклад, OWASP MASVS або PCI DSS, залежно від типу додатка та залучення сторонніх спеціалістів для проведення аудиту безпеки або тестування на проникнення.

Зворотне проектування або зворотний інжинирінг — це процес аналізу продукту чи системи для розуміння їх дизайну, функцій і базових принципів. Він включає в себе розбирання продукту або системи для вивчення його компонентів і їх зв’язків один з одним, щоб зрозуміти, як це працює.

Зворотне проектування часто використовується для створення нових продуктів або вдосконалення існуючих. Наприклад, компанія може провести зворотне проектування продукту конкурента, щоб визначити його сильні та слабкі

сторони або визначити, як його можна вдосконалити чи модифікувати.

Процес зворотного проектування зазвичай включає кілька етапів, зокрема:

- Визначення продукту або системи для аналізу та збирання інформації про них. Може включати вивчення маркетингових матеріалів, посібників користувача або інших джерел інформації про продукт або систему.
- Розбирання продукту або системи та вивчення його компонентів та їхніх зв’язків один з одним.
- Створення детальної схеми або моделі продукту чи системи, показуючи взаємозв’язки між її компонентами та те, як вони взаємодіють один з одним. Може включати створення фізичної моделі, моделі автоматизованого проектування (САПР) або моделювання програмного забезпечення.
- Аналіз продукту або системи, щоб зрозуміти їх дизайн і функції, а також визначити потенційні області для вдосконалення або модифікації. Може включати проведення тестів, моделювання або інші форми аналізу.
- Використання інформації, зібраної під час процесу зворотного проектування, щоб створити новий продукт або вдосконалити існуючий. Це може передбачати внесення змін до дизайну, функцій або базових принципів продукту чи системи, щоб зробити їх більш ефективними чи зручнішими для користувача.

Загалом, зворотне проектування є цінним інструментом для розуміння складних продуктів або систем, для розробки нових або вдосконалених продуктів на основі цього розуміння. Та зворотне проектування так само часто використовується зловмисниками щоб на етапі аналізу знайти слабкі місця, вразливості та використати отриману інформацію для атаки.

Для захисту самого коду від зворотного проектування рекомендується використовувати обфускацію коду.

Обфускація (заплутування) коду – це перетворення читабельного JavaScript або іншої мови програмування у нечитабельний формат, який важко зрозуміти і модифікувати, але який працює так само, як вихідний код. Програмний код часто маскується, щоб захистити інтелектуальну власність або комерційну таємницю, а також щоб запобігти зловмиснику зворотній інжиніринг запатентованої програми.

Шифрування частини або всього коду програми є одним із методів обфускації. Інші підходи включають видалення потенційно відкритих метаданих , заміну імен класів і змінних безглуздими мітками та додавання невикористаного або безглуздого коду до сценарію програми. Інструмент під назвою обфускатор автоматично перетворює звичайний вихідний код у програму, яка працює належним чином, але її складніше читати, розуміти та, отже, скомпрометувати потенційно зловмисниками.

Обфускація в комп'ютерному коді використовує складні обхідні фрази та надлишкову логіку, щоб зробити код складним для розуміння читачеві, зберігаючи при цьому властиву йому функціональність. Зчитувач може бути людиною (справжнім користувачем або суб'єктом кіберзагрози), комп'ютерним пристроєм або іншою програмою (наприклад, шкідливим ПЗ). Мета полягає в тому, щоб відволікти читача складним синтаксисом і ускладнити йому аналіз повідомлення та визначення його справжнього змісту.

Приклад обфускації коду (рисунок 2.3.1):

```
1.  // До обфускації
2.  function greeting(name) {
3.    console.log('Hello ' + name);
4.  }
5.
6.  // Після обфускації
7.  function _0x4a12(_0x5980d5){
8.    console['log']('Hello\x20'+_0x5980d5);
9.  }
```

Рис. 2.4. Приклад обфускації коду

Як бачимо, в обфусцированому коді:

- імена функцій і змінних замінені на беззмістовний набір символів
- рядки зашифровані в шістнадцятковий формат
- структура коду змінена і заплутана

Але функціональність коду залишилася незмінною. При виклику _0x4a12('Anastasiia') ми отримаємо той самий результат, що і при greeting('Anastasiia').

Деякі з найпоширеніших методів обфускації:

- *Перейменування*. Обфускатор змінює методи та імена змінних. Нові імена можуть включати нерозбірливі, недруковані або невидимі символи. Цей метод зазвичай використовується обфускаторами Java, iOS , Android і .NET;
- *Упаковка* - стискає всю програму, щоб зробити код нечитабельним;
- *Контроль потоку* – змінення логічної структури коду, щоб зробити його менш простежуваним. Декомпільований код дає недетерміновані семантичні результати та виглядає як спагетті-логіка . Ця логіка є неструктурованою, і тому хакеру важко зрозуміти чи скористатися нею;
- *Перетворення шаблону інструкції*. Цей підхід використовує загальні інструкції, створені компілятором, і замінює їх більш складними, менш поширеними інструкціями, які ефективно виконують ті самі операції, а також посилюють код;
- *Перетворення арифметичних і логічних виразів*. Прості арифметичні та логічні вирази замінено складними еквівалентами, які важко зрозуміти;
- *Вставка фіктивного коду*. До програми можна додати фіктивний або допоміжний код, щоб ускладнити її читання та виконати зворотне проектування. Як і інші методи обфускації, це не впливає на логіку або результат програми;
- *Видалення метаданих або невикористаного коду*. Метадані надають додаткову інформацію про програму, подібно до анотацій у документі Word, які можуть допомогти читачам зрозуміти її вміст, історію, творця тощо. Видалення метаданих, а також невикористаного коду залишає хакеру менше інформації про програму та її код, зменшуючи ймовірність того, що вони зможуть зрозуміти її логіку для цілей зворотного проектування. Ця техніка також може покращити продуктивність програми під час виконання;
- *Бінарне зв'язування*. Об’єднання кількох вхідних виконуваних файлів або бібліотек в один (чи більше) вихідний бінарний файл зменшує кількість інформації, доступної кіберзлочинцям для можливого використання програми. Це також зменшує розмір програми та спрощує її розгортання;
- *Непрозора вставка предиката*. Предикат у коді - це логічний вираз,

який є істинним або хибним. Непрозорі предикати — це умовні розгалуження — або оператори «якщо-тоді» — де результати не можна легко визначити за допомогою статистичного аналізу. Вставлення непрозорого предиката вводить непотрібний код, який ніколи не виконується, але може спантеличити когось, хто намагається зрозуміти декомпільований вихід;

- *Шифрування рядка*. Цей метод використовує шифрування, щоб приховати рядки у виконуваному файлі, і відновлює їх значення лише тоді, коли вони потрібні для запуску програми. Це ускладнює проходження програми та пошук певних рядків. Тим не менш, розшифровка рядків під час виконання може негативно вплинути на продуктивність виконання, хоча ефект зазвичай досить незначний;
- *Транспонування коду*. Це зміна порядку підпрограм і гілок у коді без видимого впливу на його поведінку.

Також доступні інструменти для кращого захисту програм від спроб зворотного проектування коду. Одним із прикладів є засіб захисту від помилок . Законні розробники програмного забезпечення та хакери використовують інструменти налагодження, щоб перевіряти код рядок за рядком і виявляти проблеми безпеки. Однак хакери також можуть використовувати ці інструменти для зворотного проектування коду, пошкодження даних, до яких програма отримує доступ, або виклику випадкових збоїв. Інструменти захисту від налагодження дозволяють фахівцям з IT-безпеки визначати, коли хакер запускає програму налагодження як частину атаки, зупиняючи їх спроби зворотного проектування та інші зловмисні дії.

Приклади інструментів для обфускації коду: ProGuard, DexGuard, ConfuserEx та інші.

2.4. Захист під час експлуатації додатку (використання користувачем)

Захист під час експлуатації додатку може містити:

- *Шифрування даних на пристрої*. Використовується для захисту

збережених даних у додатку, зокрема чутливої інформації. Прикладом інструментів для шифрування є Keychain Services для iOS та Keystore для Android.

- *Захист під час передачі даних*. Використання протоколів шифрування (наприклад, HTTPS з SSL/TLS) для захисту даних, що передаються між додатком і сервером.

- *Використання механізмів аутентифікації та авторизації*. Для забезпечення захисту облікових записів користувачів необхідне впровадження аутентифікації (наприклад, двофакторної) та контролю доступу до даних. Прикладом рішень є OAuth 2.0, OpenID Connect.

- *Захист від атак у реальному часі(RASP).*

Самозахист програм під час виконання або RASP — це технологія безпеки, яка працює в програмних програмах для автоматичного моніторингу, виявлення, аналізу та захисту від зловмисної діяльності в режимі реального часу під час роботи програми.

RASP відрізняється від інших інструментів захисту коду, таких як статичне тестування безпеки додатків (SAST) або брандмауери веб-додатків (WAF), тим, що виходить за рамки статичного коду. RASP аналізує та адаптує логіку та поведінку програми під час виконання, щоб забезпечити більш детальний, точний і комплексний захист, ніж традиційні рішення безпеки.

RASP вбудовано в програмне забезпечення та працює як імунна система для програми. Безпека RASP забезпечує контекстний моніторинг і виявлення в реальному часі, а також забезпечує миттєвий захист шляхом автоматичної зупинки атак у міру їх виникнення.

Основними компонентами, завдяки яким працює RASP, є:

- *Інтеграція*. Датчики, вбудовані в програму, активуються, щойно програма запускається, що називається часом виконання.

- *Постійний контекстний моніторинг*. RASP розуміє контекст, у якому працює програма, і відстежує контекстну інформацію всередині програмного забезпечення під час виконання. RASP аналізує трафік і поведінку користувачів, вхідні запити, потоки даних і шляхи виконання, щоб виявити аномальність або

вразливість у коді.

- *Виявлення та сповіщення*. У разі виявлення вразливості системи безпеки або атаки технологія RASP видає сповіщення та може надавати звіти безпеки щодо стану та продуктивності програми.
- *Блокування*. RASP одночасно й автоматично блокує зловмисний запит і віртуально виправляє програму, щоб запобігти подальшим атакам.
- *Захист у реальному часі*. Оскільки RASP вбудовано в програму, він миттєво реагує на загрози, щойно вони виникають, і на вразливі місця, коли вони виявляються.

Існує два основні методи інтеграції та впровадження RASP.

Пряме вбудовування в додаток:

- RASP безпосередньо вбудовано в код програми.
- Розробникам не потрібно суттєво змінювати логіку програми.
- Вплив на продуктивність мінімальний.
- Пряме вбудовування полегшує глибоке бачення контексту та поведінки програми.
- Захист можна адаптувати до конкретних вимог програми.

Інтеграція на основі агентів:

- Інструменти RASP можуть використовувати легкі агенти, які працюють разом із програмою.
- Агенти постійно відстежують і аналізують поведінку програми під час виконання.
- Коли така подія, як впровадження SQL, порушує правила безпеки, агент діє.
- Агенти інтегруються з інструментами оркестровки та існуючими системами, такими як SIEM або DAST.
- Використовуючи API та веб-перехоплювачі, RASP може включати канали аналізу загроз.

Приклад рішень RASP: LIAPP, Guardsquare, Pradeo та інші.

Та при всіх перевагах використання, перед впровадженням RASP необхідно врахувати сумісність з фреймворками додатку, інтеграцію з існуючими засобами безпеки, потенційні проблеми з затримкою обробки запитів та перевірку працездатності програми.

2.5. Забезпечення підтримки та оновлення безпеки після розгортання

Та навіть після випуску додатку необхідно дотримуватись наступних рекомендацій:

- *Моніторинг безпеки та реагування на інциденти*. Забезпечення постійного моніторингу стану додатку, виявлення аномальної активності та запобігання кібератакам за допомогою SIEM систем. Прикладами SIEM є Splunk, IBM, Securonix, Exabeam та інші.
- *Регулярні оновлення та патчі*. Важливо регулярно випускати оновлення безпеки для додатку задля усунення нових вразливостей.
- *Резервне копіювання та відновлення*. Для захисту даних необхідно забезпечити регулярне резервне копіювання додатку та можливість його відновлення в разі інциденту. Прикладами таких інструментів можуть бути Google Firebase для Android, iCloud для iOS.

Дані кроки допоможуть підтримувати безпеку додатку на постійній основі навіть після його випуску.

Висновки до розділу 2

У цьому розділі було виконано завдання розробки рекомендацій для якісної інтеграції безпеки та тестування на всі етапи життєвого циклу мобільного додатку. Ці рекомендації базуються на вже відомих стандартах, методологіях, програмних застосунках та утилітах, що дозволяють розробникам та тестувальникам комплексно забезпечити кібербезпеку додатку. Було розглянуто особливості захисту на етапах розробки, тестування, розгортання, експлуатації, підтримки та оновлення безпеки після розгортання.

У підсумку, розділ підкреслює важливість комплексного підходу до організації безпеки та тестування захищеності мобільних додатків, який включає в себе як теоретичний аналіз, так і практичне застосування спеціалізованих інструментів. Такий підхід до безпеки мобільного додатку дозволяє не тільки виявити потенційні вразливості, захистити від можливих загроз, а й мінімізувати нанесену шкоду при успішній атаці.

3 РОЗРОБЛЕННЯ ВАРІАНТІВ ТЕХНОЛОГІЇ ТЕСТУВАННЯ ТА ЗАХИСТУ МОБІЛЬНИХ ДОДАТКІВ

3.1. Технологія тестування SAST на базі рішення Snyk

Для прикладу застосування статичного аналізу коду ми розглянемо платформу Snyk[27], а саме рішення Snyk Code[28].

Snyk - це платформа, яка дозволяє сканувати, визначати пріоритети та виправляти вразливості безпеки у коді, залежностях з відкритим кодом, образах контейнерів та інфраструктурі у вигляді конфігурацій коду.

Платформа Snyk представляє собою сукупність різних продуктів, а саме:

- *Snyk Open Source and Snyk Code;*
- *Snyk Container;*
- *Snyk Infrastructure as Code;*
- *Snyk AppRisk.*

Snyk Code - це швидкий і точний інструмент безпеки, який видає менше помилкових спрацьовувань, що полегшує розробникам усунення проблем і створення безпечного програмного забезпечення.

Snyk Code надає можливість просканувати код за допомогою наступних опцій:

- Snyk Web UI (включаючи перевірку PR);
- Snyk IDE;
- Snyk CLI;
- Snyk API.

У наступній таблиці показано можливості Snyk Code, включаючи аналіз, управління проблемами безпеки у коді та полегшення виправлення у середовищі розробки.

Таблиця 3.1.

Опис функціоналу Snyk Code

Функціонал	Опис
Фільтрація, сортування та групування повідомлень	Щоб виявити найпоширеніші проблеми, є можливість відфільтрувати проблеми на основі їх серйозності, мови програмування, оцінки пріоритету та інших критеріїв.
Оцінка пріорітету	Сортування та визначення пріоритетів найбільш важливих проблем шляхом включення таких факторів, як поширеність проблеми, легкість її вирішення та фактор ризику, в єдиний показник ризику.
Потік даних	Візуалізація шляху проблеми від джерела до поглинача за допомогою поетапного потоку.
Вразливості	Більше інформації про вразливість за допомогою кураторського контенту, який пояснює, як була створена вразливість, які фактори ризику та популярні стратегії її усунення.
Аналіз виправлення	Надання розуміння та контекст, вивчаючи приклади з посиланнями на реальний код, який виправляє ті ж самі проблеми у схожих потоках даних.
Створення тікету в Jira	Відстеження та експорт проблеми Snyk до проекту Jira.
Ігнорування проблем	Можливість налаштувати Snyk на ігнорування запропонованих виправлень для проблеми, щоб приховати певні попередження. Наприклад, ви навмисно використовували жорстко закодовані паролі для перевірки процедур у тестовому коді,

	або знаєте про проблему, але вирішили не виправляти її.
Виключення файлів з процесу імпорту	Перевірка наявності ігнорованих DeepCode/Snyk файлів .gitignore .dcignore і прочитання їх, якщо вони існують. Використовуючи інформацію з цих файлів, Snyk фільтрує їх, щоб виявити лише файли з підтримуваними розширеннями у каталозі проекту, але не вище поточного каталогу проекту. Snyk Code об'єднує ці файли, розмір яких менший за 4 МБ, у пакети і надсилає їх до Snyk. Винятки gitignore враховуються CLI командою snyk code test..
Міжфайловий аналіз	Даний аналіз доступний для всіх мов, що підтримуються Snyk Code, окрім Ruby.

Snyk Code працює на основі семантичного механізму аналізу, заснованого на штучному інтелекті, і може аналізувати у коді наступне:

- *Використання API*: Виявляє численні потенційні проблеми, включаючи зловживання API, нульові посилання та невідповідності типів, моделюючи використання пам'яті у змінних та посиланнях. Цей механізм також може виявити використання небезпечних функцій.
- *Проблеми кодування*: Знаходить такі проблеми, як мертвий код, наперед визначені гілки та гілки з однаковим кодом з обох боків.
- *Потік керування*: Виявляє нульове розіменування або умови гонки, моделюючи кожен можливий потік керування у програмі.
- *Потік даних*: Відстежує потік даних у додатку від джерела до приймача. У поєднанні з навчанням на основі штучного інтелекту про зовнішні небезпечні джерела даних, поглиначі даних і функції санітарії це дає змогу проводити потужний аналіз на предмет забруднення.
- *Жорстко закодовані секрети*: Жорстко закодовані правила виявлення

секретів застосовуються під час сканування SAST, але не діють як окремий інструмент сканування секретів, оскільки це робиться через наше партнерство зі сторонніми інструментами.

- *Точковий аналіз*: Виявляє численні потенційні проблеми, зокрема вичерпання буфера, нульові посилання та невідповідності типів, моделюючи використання пам'яті у змінних та посиланнях.
- *Виведення типів*: Визначає початковий тип та його зміни. Це особливо важливо для динамічно типізованих мов.
- *Діапазони значень*: Вказує можливі значення змінних, які використовуються для виклику функцій, щоб відстежувати помилки на одиницю в масивах, помилки ділення на нуль та нульові посилання.

Кодом для аналізу я обрала код додатку *DIVA*[29].

DIVA (Damn insecure and vulnerable App) - це Android додаток, який навмисно розроблений таким чином, щоб бути незахищеним. Мета додатку - навчити розробників/контролерів якості/професіоналів з безпеки недолікам, які зазвичай присутні в додатках через погані або небезпечні практики кодування.

Даний додаток можна використовувати як навчальну платформу для практичного ознайомлення з вразливостями мобільних додатків, але я проаналізую сам код додатку на наявність вразливостей за допомогою Snyk Code.

Код додатку Diva[30] я скопіювала з вказаного GitHub ресурсу до свого новоствореного акаунту (рисунок 3.1.).

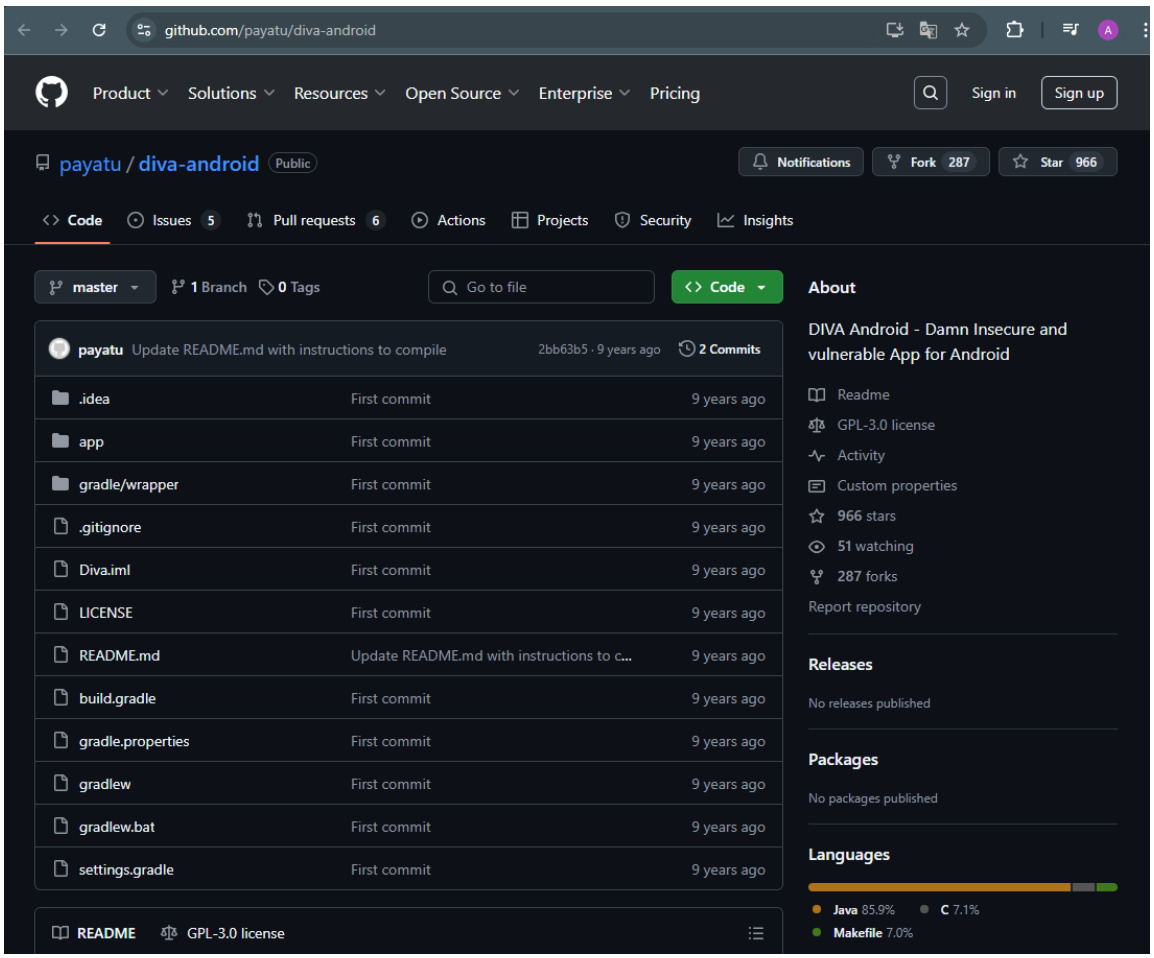

Рис. 3.1. GitHub-ресурс, де розміщено код додатку DIVA

Заходимо в акаунт Snyk (рис. 3.2):

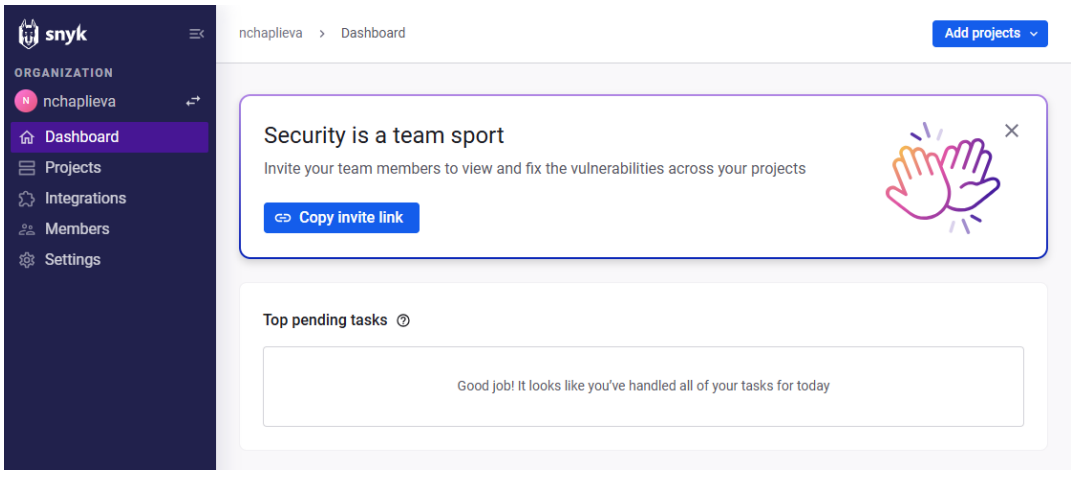

Рис. 3.2. Вхід до акаунту на платформі Snyk

Налаштовуємо інтеграцію з моїм GitHub акаунтом (рис. 3.3):

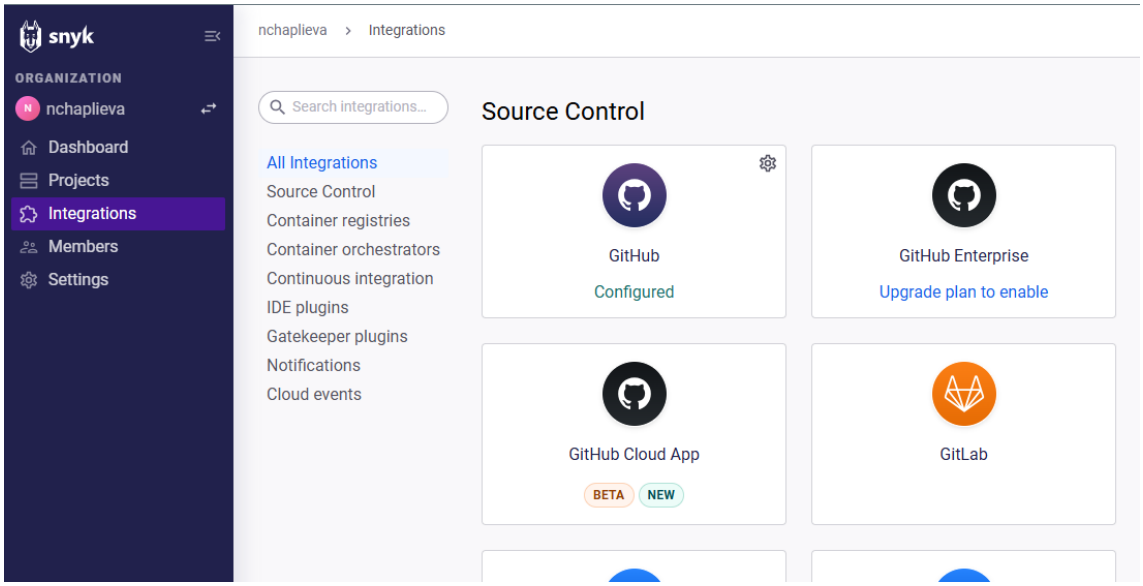

Рис. 3.3. Налаштування інтеграцій Snyk Code

Рішення дозволяє аналізувати як локальні проекти так і проекти, які знаходяться на GitHub. Я обрала другий варіант та додала репозиторій Diva до платформи Snyk (рис. 3.4):

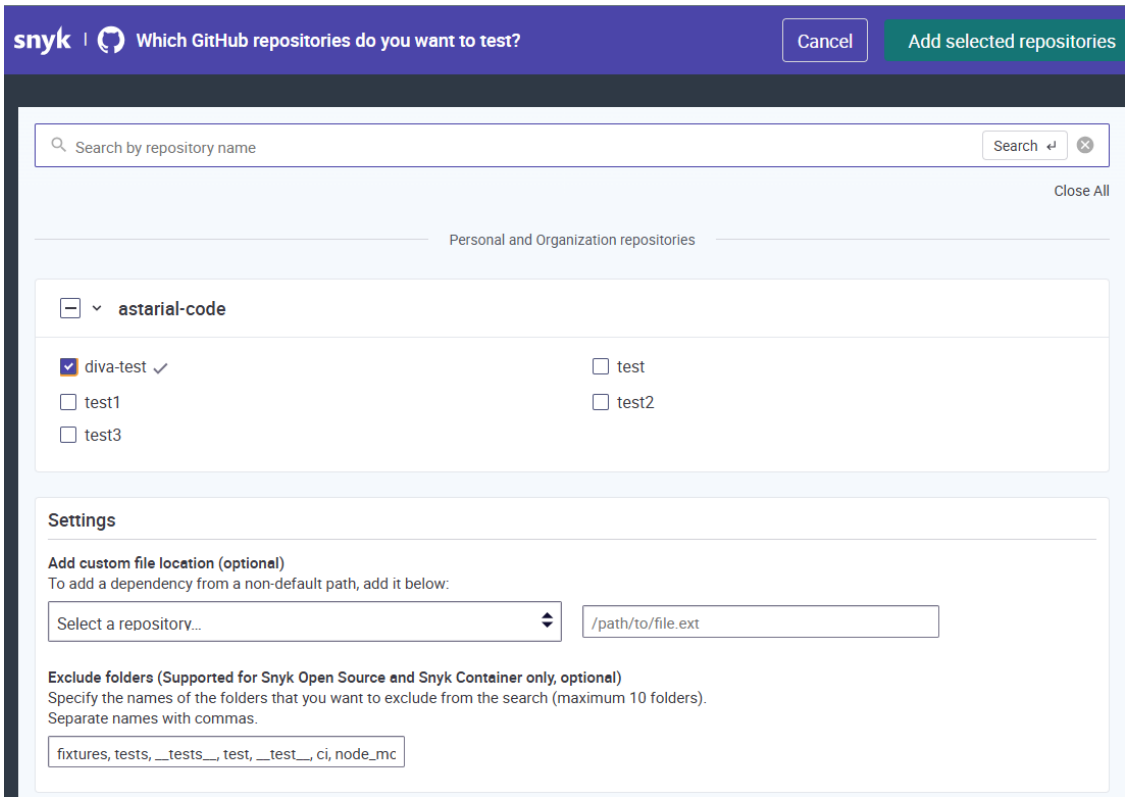

Рис. 3.4. Імпорт репозиторію DIVA з GitHub

Після того як проєкт підвантажився ми одразу отримаємо результат сканування на вразливості (рисунок 3.5):

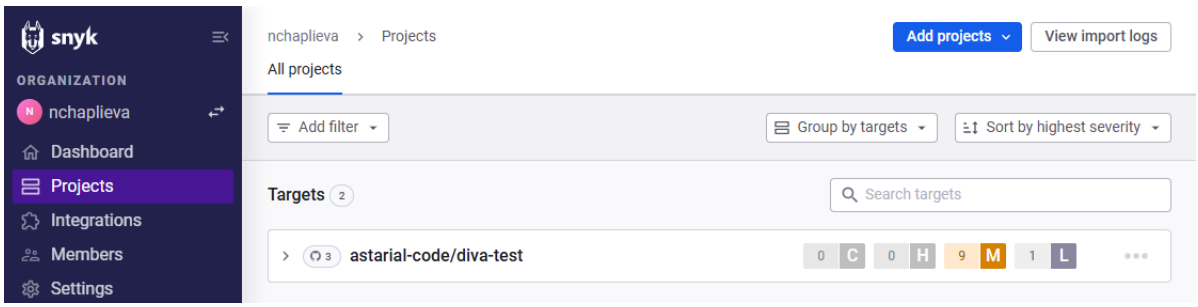

Рис. 3.5. Результат імпорту репозиторію з GitHub

Результат сканування (рисунок 3.6.) показав наявність 9 вразливостей середньої критичності та 1 низької, використані мови програмування Java та C/C++, а також знайдені типи вразливостей SQL ін’єкції, потенційне перевантаження буферу та активування JavaScript:

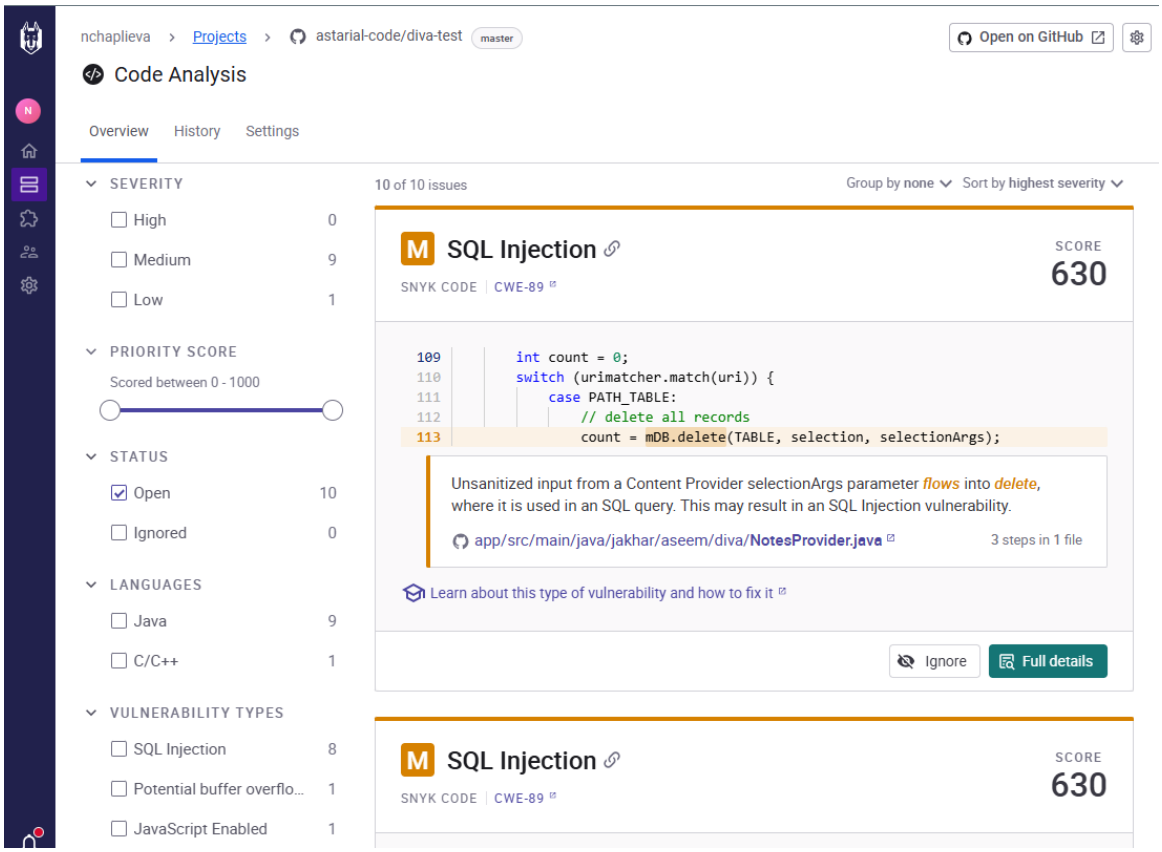

Рис. 3.6. Сторінка результату сканування на вразливості додатку DIVA

На прикладі знайденої SQL ін’єкції (рисунок 3.7) ми можемо побачити, що дана вразливість CWE-89 (з посиланням на ресурс MITRE ATT&CK[31] (рисунок 3. 8)) та яка саме частина коду є вразливою:

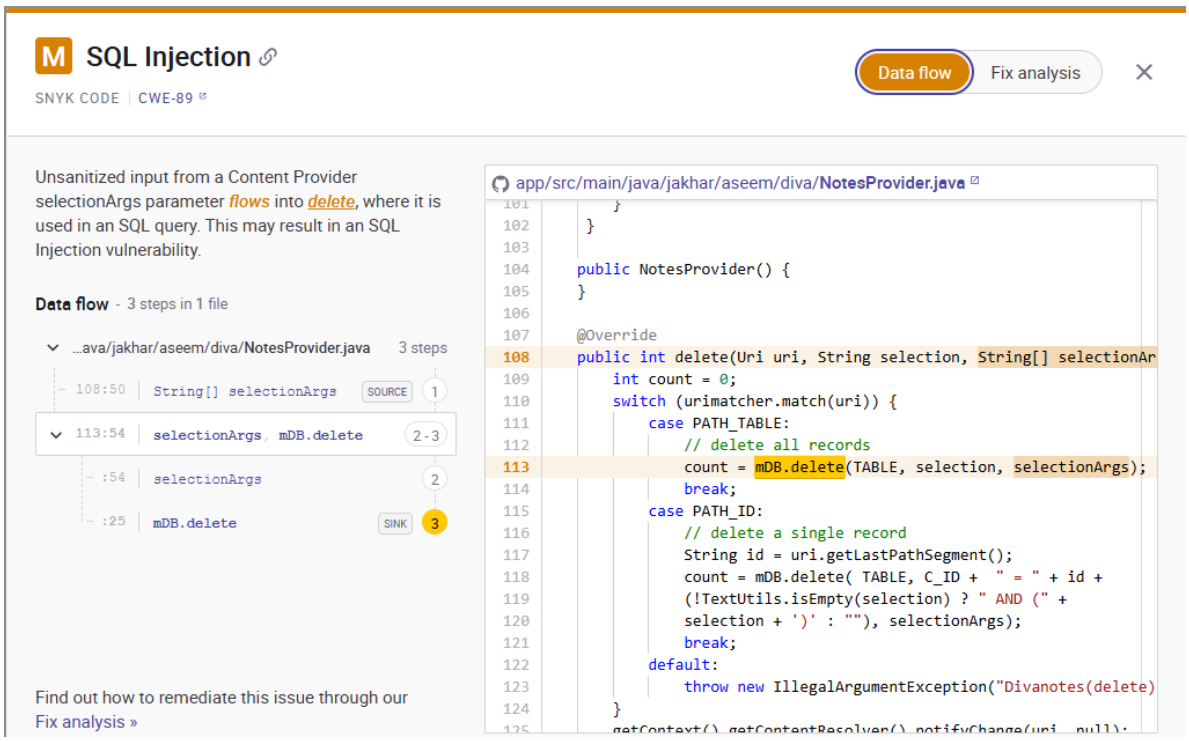

Рис. 3.7. Приклад знайденої SQL ін’єкції у коді

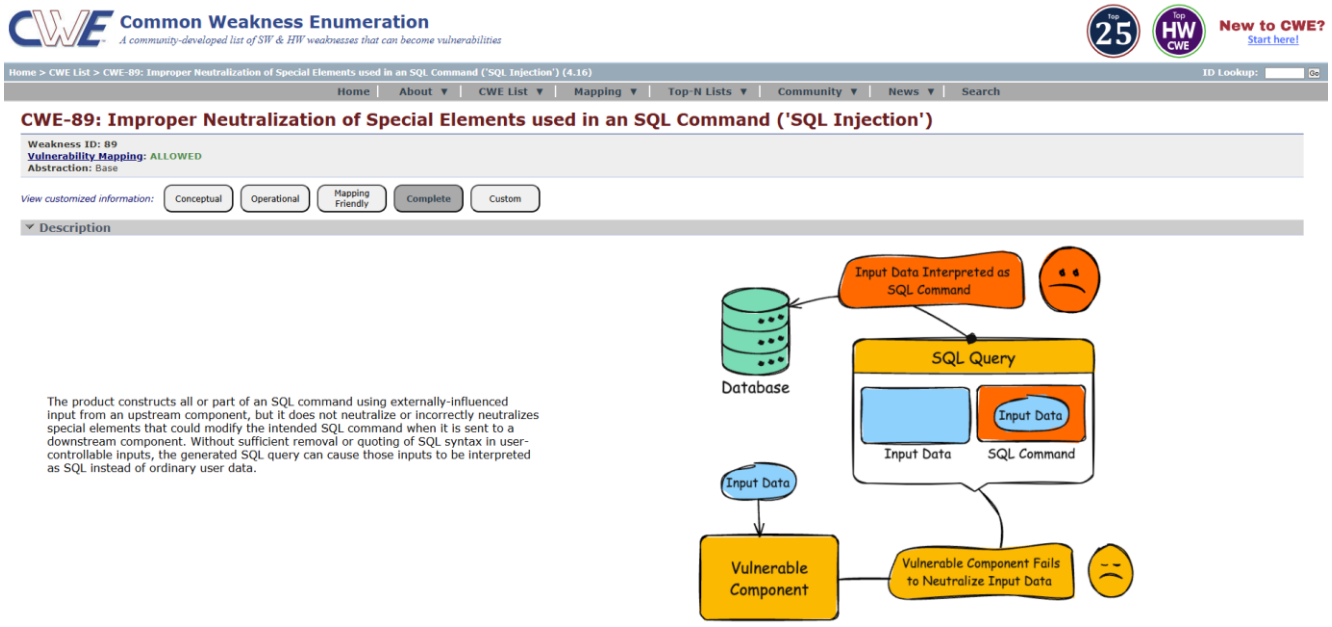

Рис. 3.8. Опис знайденої SQL ін’єкції від MITRE за посиланням

Також Snyk надає опис потенційної небезпеки знайденої вразливості та рекомендації щодо її усунення (рисунок 3.9):

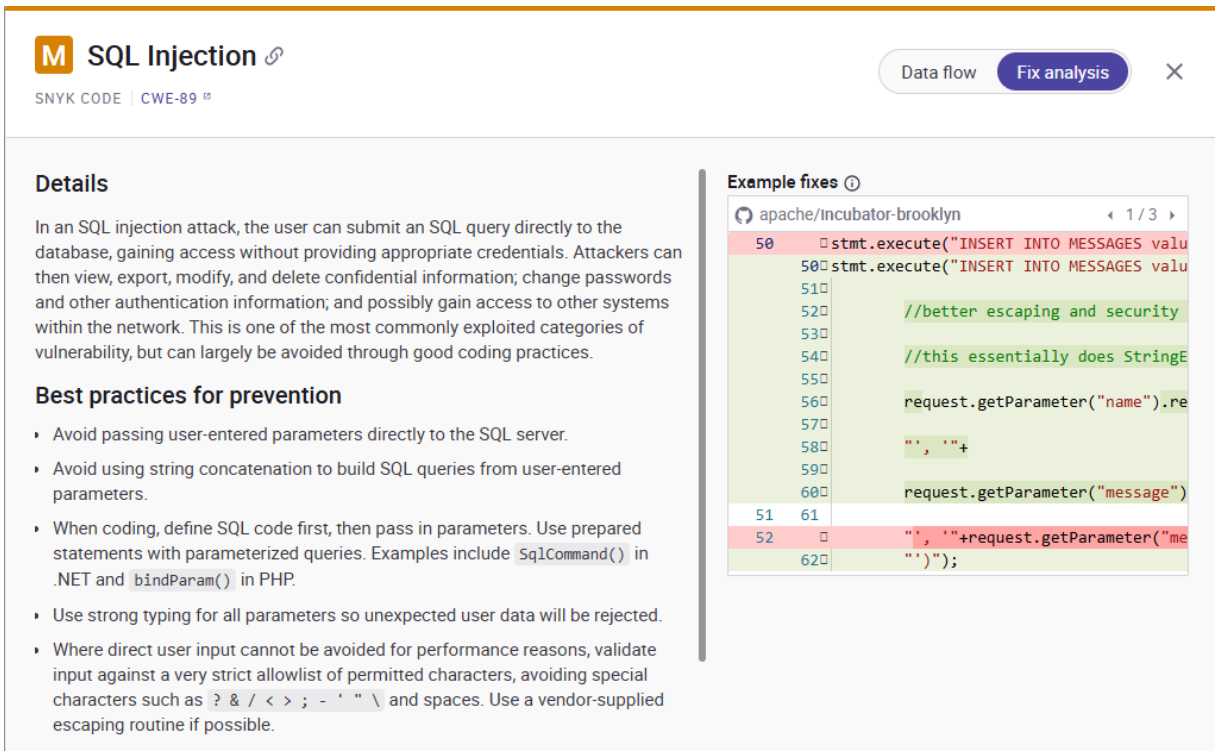

Рис. 3.9. Приклад опису та рекомендацій щодо усунення вразливості в коді

Також Snyk за замовченням надсилає щотижневі звіти сканування на вразливості коду доданих проєктів (рис. 3.10 та 3.11):

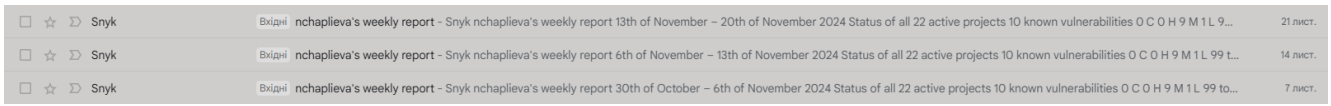

Рис. 3.10. Приклад щотижневих листів від Snyk

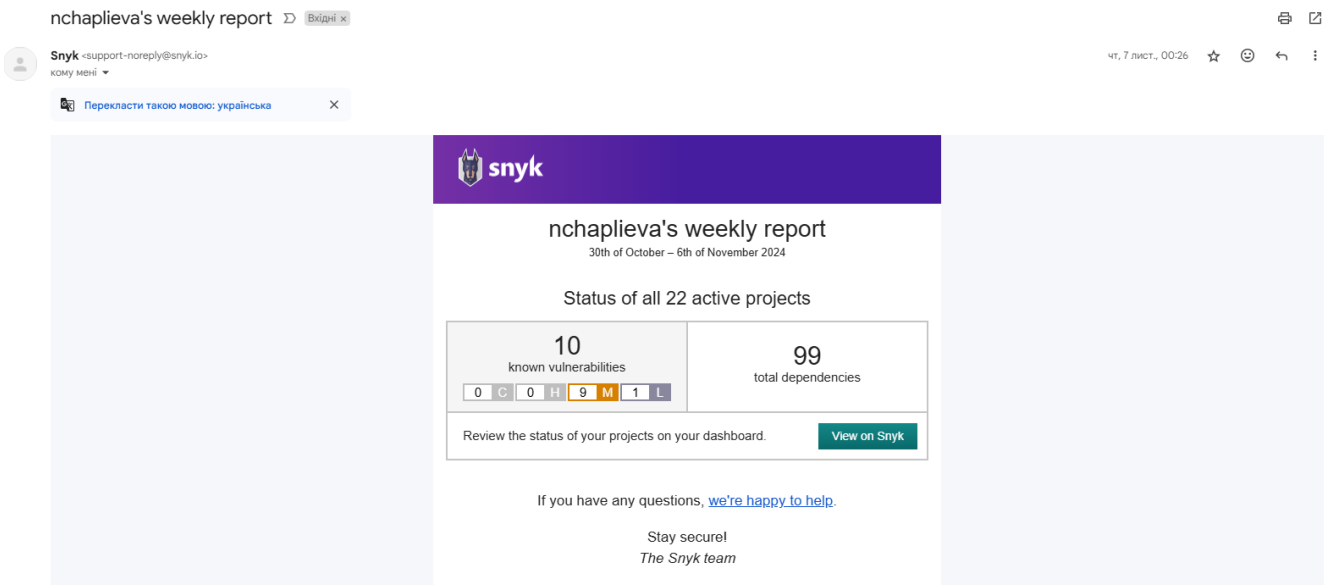

Рис. 3.11. Приклад вмісту щотижневих листів від Snyk

Дані сповіщення дозволяють отримувати актуальну інформацію щодо стану коду додатку та вчасно реагувати на знайдені вразливості.

Таким чином ми можемо проаналізувавши код додатку ще на етапі розробки, зменшити кількість вразливостей і тим самим підвищити його безпеку.

3.2. Технологія перевірки середовища запуску додатку як частини RASP

Виявлення кореневих прав доступу на мобільному пристрою є одним з способів ускладнення запуску додатку на пристрої з «root» правами, якими користуються реверс-інженери.

Як і більшість інших засобів захисту, виявлення кореня не є дуже ефективним саме по собі, але впровадження кількох перевірок кореня, розкиданих по всій програмі, може покращити ефективність загальної схеми захисту від несанкціонованого доступу.

SafetyNet — це API Android, який надає набір послуг і створює профілі пристроїв відповідно до інформації про програмне та апаратне забезпечення. Потім цей профіль порівнюється зі списком прийнятних моделей пристроїв, які пройшли перевірку сумісності з Android. Google рекомендує[32] використовувати цю функцію як «додатковий сигнал поглибленого захисту як частину системи протидії зловживанням».

Для використання API, програма викликає SafetyNetApi.attest метод (який повертає повідомлення JWS із результатом атестації), а потім перевірити такі поля:

- *nonces*: відповідати відповіді на запит.
- timestampMs: щоб перевірити, скільки часу минуло з моменту подання запиту та отримання відповіді. Затримка відповіді може свідчити про підозрілу активність.
- *apkPackageName*, *apkCertificateDigestSha256*, *apkDigestSha256*: надати інформацію про файл APK, який використовується для перевірки ідентичності

програми, що викликає. Ці параметри відсутні, якщо API не може надійно визначити інформацію про APK.

- *ctsProfileMatch*: якщо значення true, профіль пристрою відповідає одному з пристроїв Google у списку.
- *basicIntegrity*: якщо значення true, пристрій, на якому запущено програму, швидше за все, не було змінено.

Приклад результату атестації за допомогою виклику SafetyNetApi.attest (рисунок 3.2.1):


```
{
  "nonce": "R2Rra24fVm5xa2Mg",
  "timestampMs": 9860437986543,
  "apkPackageName": "com.package.name.of.requesting.app",
  "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the
                                  certificate used to sign requesting app"],
  "apkDigestSha256": "base64 encoded, SHA-256 hash of the app's APK",
  "ctsProfileMatch": true,
  "basicIntegrity": true,
}
```


Рис. 3.12. Приклад результату виклику SafetyNetApi.attest

Також існує мобільний додаток SafetyNet для перевірки працездатності і безпеки середовища пристрою та програми з PlayMarket(рисунок 3.13):

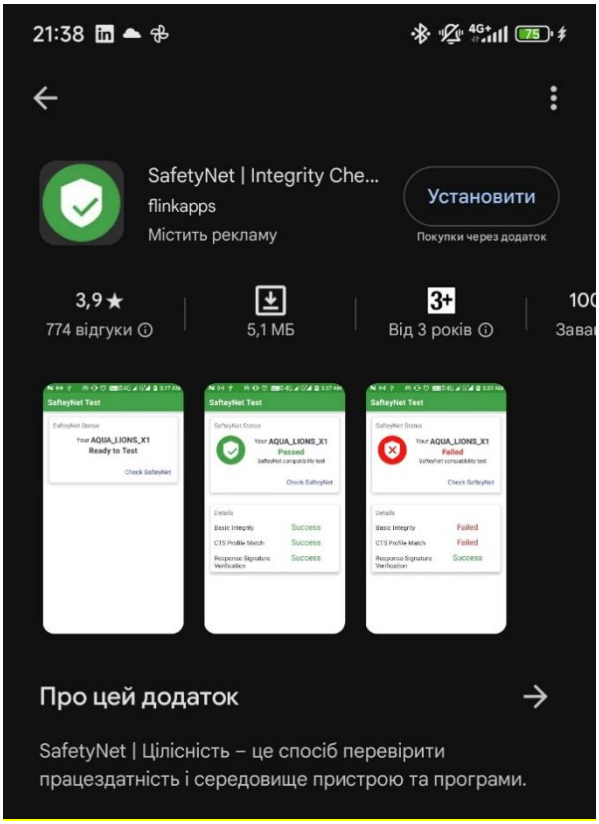

Рис. 3.13. Додаток SafetyNet у PlayMarket

Для запуску перевірки необхідно встановити додаток.ю запустити його та натиснути «Check Play Integrity» (рисунок 3.14):

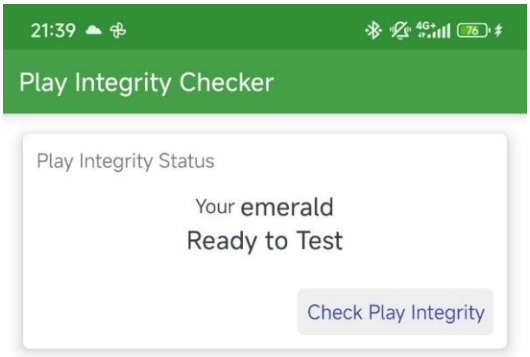

Рис. 3.14. Запуск перевірки середовища у додатку SafetyNet

Результат запуску перевірки на телефоні (рисунок 3.15):

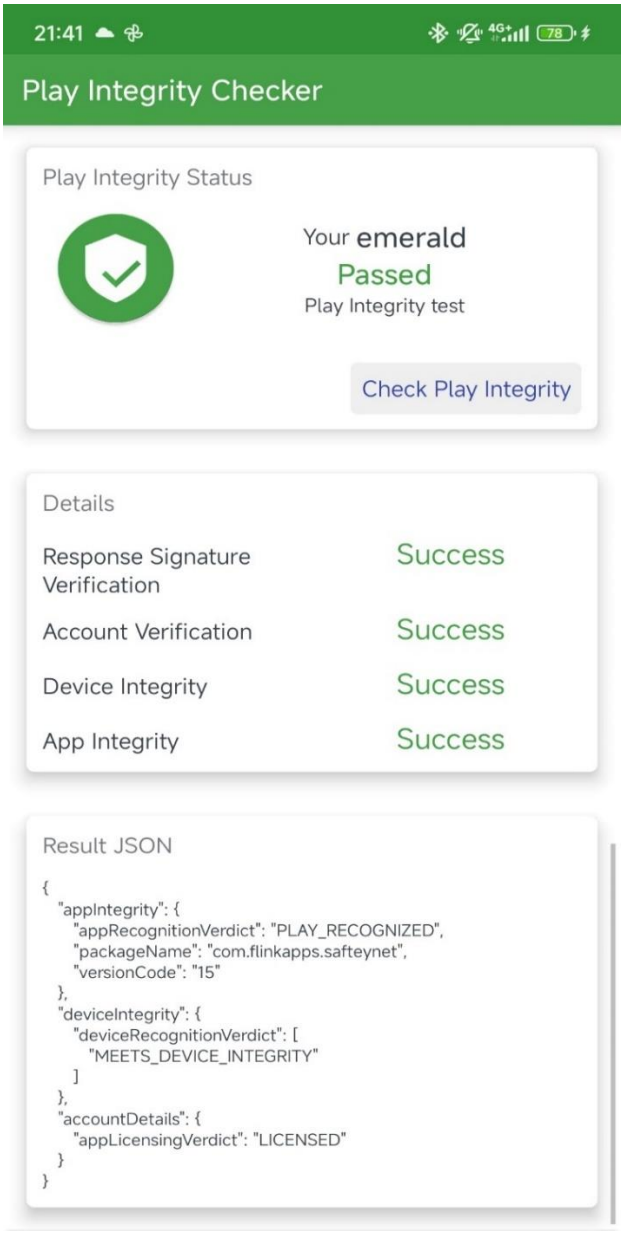

Рис. 3.15. Результат перевірки середовища у додатку SafetyNet

Та приклад негативного результату (рисунок 3.16):

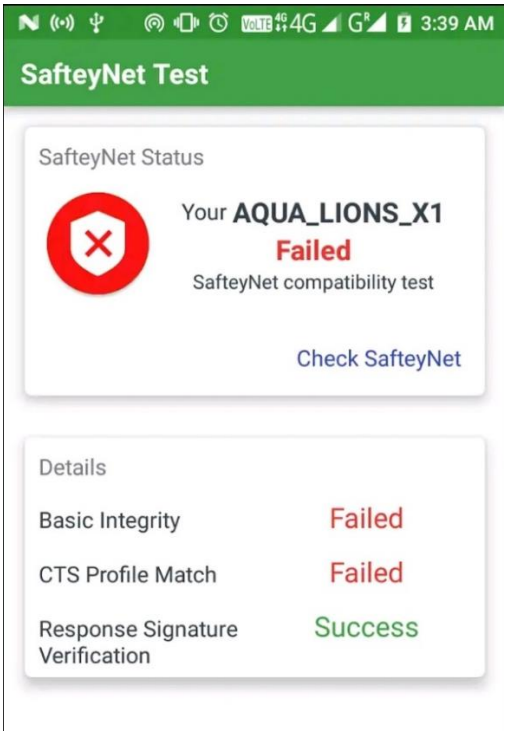

Рис. 3.17 Приклад негативного результату перевірки середовища у додатку SafetyNet

Таким чином ми можемо перевірити середовище як додавши інтеграцію в додатку з API запитом SafetyNetApi.attest на перевірку, так і запустивши мобільний додаток SafetyNet на телефоні.

3.3. Розроблення та приклад застосування технології захисту обфускації коду JavaScript

В роботі розглянемо декілька базових підходів до обфускації коду.

Перейменування ідентифікаторів. Імена змінних і функцій замінюються на випадковий набір символів, що не має сенсу (рисунок 3.18):

```
// До обфускації
var number = 10;
function add(x, y) {
  return x + y;
}

// Після обфускації
var _0x7daa1 = 0xa;
function _0x514a(_0x2abf1a, _0x12c6a2) {
  return _0x2abf1a + _0x12c6a2;
}
```

Рис. 3.18. Приклад застосування обфускації коду

Кодування рядків. Рядкові літерали конвертуються в юнікод, шістнадцятковий код або інші уявлення (рисунок 3.19):

```
console.log("Hello World");

// Шістнадцяткове кодування
console.log("\x48\x65\x6C\x6C\x6F\x20\x57\x6F\x72\x6C\x64");

// Юнікод
console.log("\u0048\u0065\u006C\u006C\u006F\u0020\u0057\u006F\u0072\u006C\u0064");

// Base64
console.log(atob("SGVsbG8gV29ybGQ="));
```

Рис. 3.19. Приклад застосування підходу «Кодування рядків»

Мертвий код. У код вставляються безглузді операції, які не впливають на результат. Це заплутує логіку роботи скрипта (рисунок 3.20):

```
function add(x, y) {
  return x + y;
}
// З мертвим кодом
function add(x, y) {
  var _0x52ba = 0xffff;
  if (_0x52ba > 0) {
    console.log("Dummy");
  }
  return x + y;
}
return x + y;
}
```

Рис. 3.20. Приклад застосування підходу «Мертвий код»

Заплутування потоку управління. Додаються зайві умови, цикли і переходи.

Це маскує реальну послідовність виконання функцій (рисунок 3.21):

```
function factorial(num) {
  if (num < 0) return;
  if (num === 0) return 1;
  return num * factorial(num - 1);
}
// Після обфускації
function _0x4a2a(_0x3b9439) {
  if (_0x3b9439 < 0) {
    if (false) {
      return;
    } else {
      return;
    }
  }
  if (_0x3b9439 === 0) {
    if (true) {
      return 1;
    }
  }
  return _0x3b9439 * _0x4a2a(_0x3b9439 - 1);
}
```

Рис. 3.21 Приклад застосування підходу «Заплутування потоку управління»

Динамічна генерація коду. Код збирається по шматочках прямо під час виконання за допомогою eval() і Function() (рисунок 3.22):

```
var add = new Function('x', 'y', 'return x + y');

var result = add(2, 3);

console.log(result);
```

Рис. 3.22. Приклад застосування підходу «Динамічна генерація коду»

Пакування/стиснення коду. Код мінімізується, шифрується і записується в один рядок. Перед виконанням він розпаковується (рисунок 3.23):

```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){whil
```

Рис. 3.23. Приклад застосування підходу «Пакування/стиснення коду»

Також існують безкоштовні онлайн-ресурси для обфускації коду, такі як JavaScript Obfuscator Tool.

JavaScript Obfuscator Tool[33] – онлайн-сервіс з великим набором налаштувань, включаючи контроль імен змінних, методи шифрування рядків і

мертвий код.

Для прикладу візьмемо код звичайного калькулятора та додамо до сервісу (рисунок 3.24):

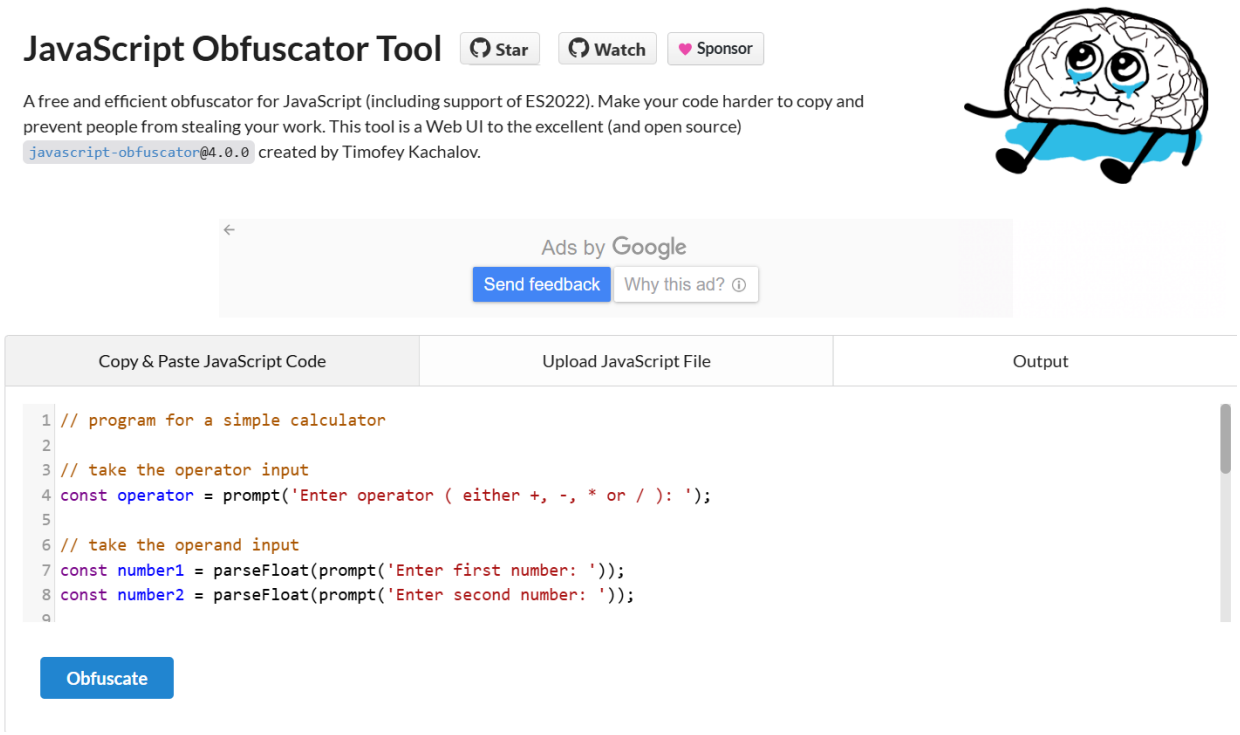

Рис. 3.24. Додавання коду до JavaScript Obfuscator Tool

І натиснувши кнопку «Obfuscate» ми отримаємо наш код вже після обфускації (рисунок 3.25):

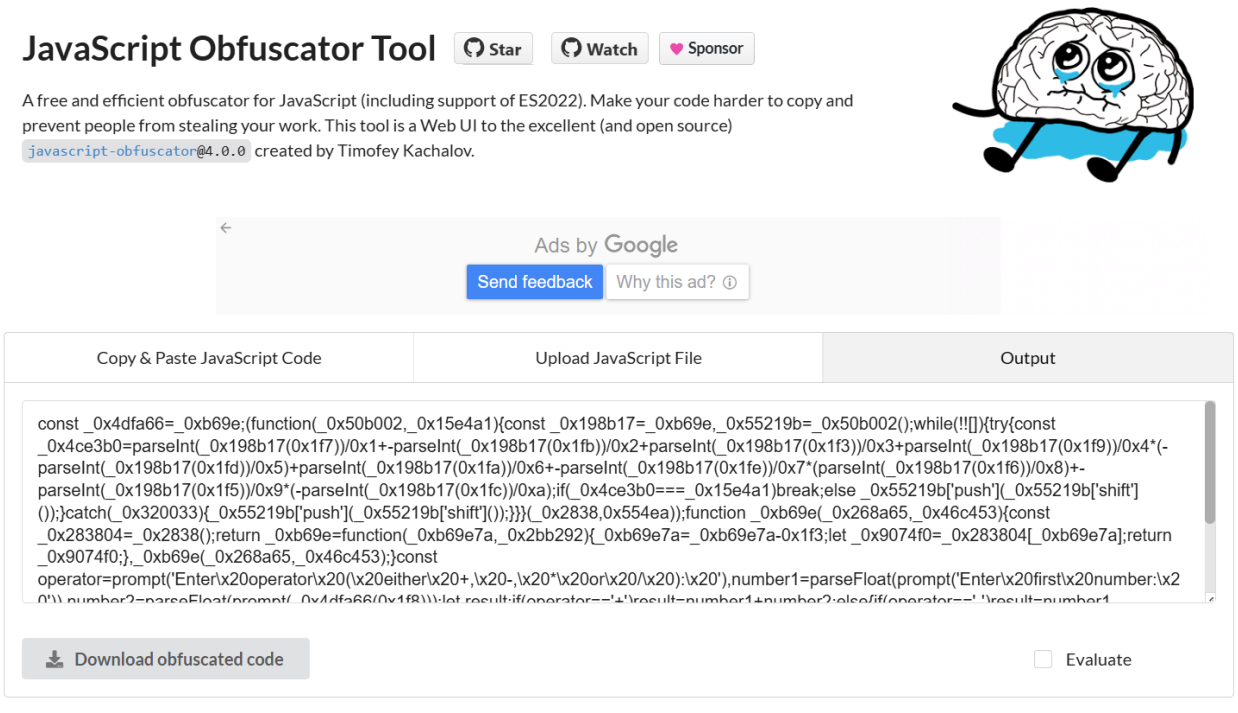

Рис. 3.25 Результат обфускації коду в JavaScript Obfuscator Tool

При необхідності сервіс має також додаткові налаштування (рисунок 3.26):

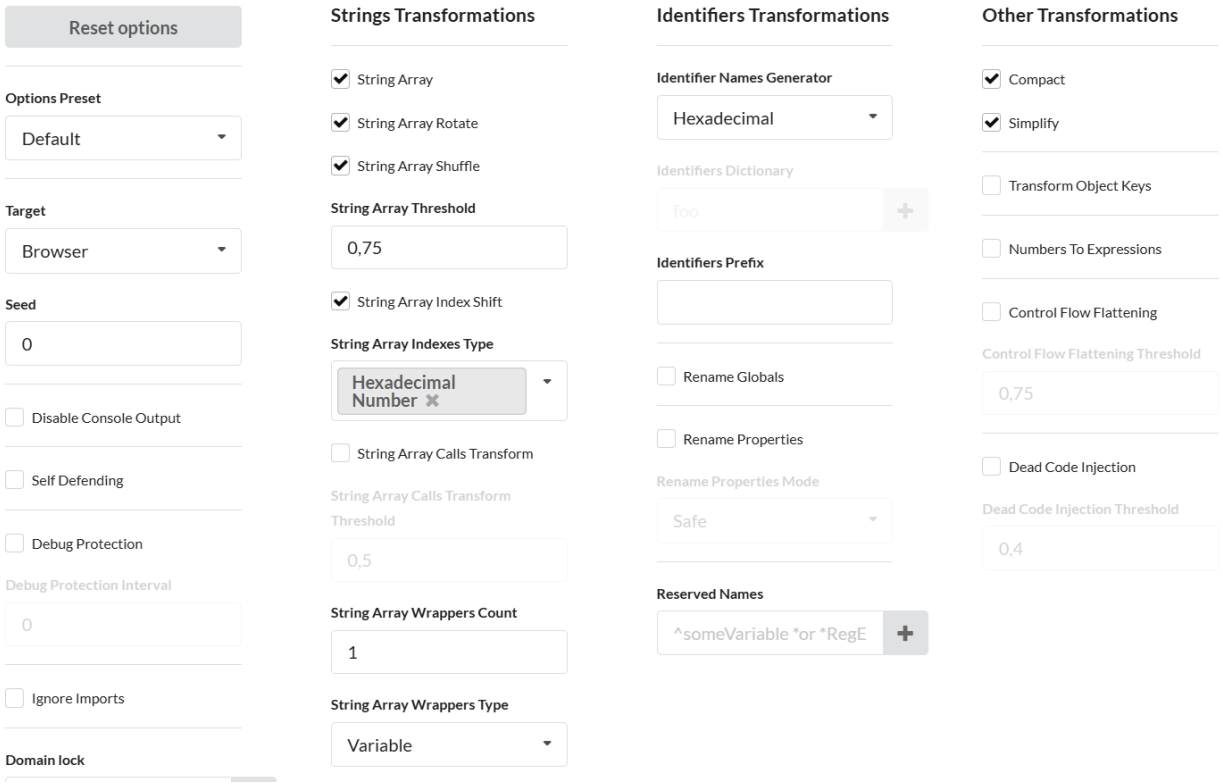

Рис. 3.26. Додаткові налаштування для обфускації коду у JavaScript Obfuscator Tool

Вибір конкретного інструменту залежить від вимог до рівня обфускації, розміру коду і сумісності з використовуваними фреймворками і системою збірки.

Однак обфускація має свої мінуси – вона уповільнює виконання і ускладнює підтримку коду. Тому обфускацію слід застосовувати зважено і багаторівнево, комбінуючи різні підходи.

Висновки до 3 розділу

У цьому розділі кваліфікаційної роботи було розглянуто приклади застосування таких технологій як статичне сканування коду на базі Snyk Code, технології самозахисту додатку в реальному часі (RASP) на базі застосування скрипта перевірки середовища запуску додатку та технології обфускації коду на базі JavaScript.

Згадані технології можливо інтегрувати в життєвий цикл розробки та

автоматизувати як тестування так і захист.

Такий підхід до тестування та безпеки мобільних додатків дозволяє проводити глибокий аналіз коду, виявляти складні вразливості, автоматизувати процес виявлення загроз та захистити код додатку від зворотнього проектування.

ВИСНОВКИ

У роботі проведено дослідження проблеми забезпечення безпечної роботи мобільних додатків, визначено його мету та завдання.

Проведено аналіз існуючих технологій безпеки та тестування мобільних додатків на всіх етапах їхнього життєвого циклу (від розробки до використання).

Досліджено методи та засоби забезпечення безпеки та тестування додатків при комплексному підході до кібербезпеки на кожному з етапів життєвого циклу розробки. Визначено призначення, основні функції, переваги та недоліки застосування технологій SAST на базі Snyk, RASP на базі скриптів перевірки середовища запуску додатку та обфускації коду.

Сьогодні інструменти безпеки повинні бездоганно вписуватися в робочий процес розробки та конвеєр CI/CD, щоб йти в ногу з DevOps і не сповільнювати швидкість розробки.

Ключовою практикою DevSecOps є інтеграція безпеки в усі робочі процеси DevOps. Проводячи заходи безпеки на ранній стадії та послідовно протягом усього життєвого циклу розробки програмного забезпечення (SDLC), організації можуть гарантувати, що вони виявляють уразливості якомога раніше, і мають змогу приймати обґрунтовані рішення щодо ризиків і пом’якшення.

Також необхідно не забувати про комплексний підхід щодо забезпечення безпечної роботи додатку на всіх етапах життєвого циклу.

На основі досліджень проведених в роботі запропоновано рекомендації щодо порядку застосування технологій безпеки та тестування додатків на всіх етапах їхнього життєвого циклу (від розробки до використання) та розглянуто приклади застосування технологій SAST на базі Snyk, RASP на базі скрипту перевірки середовища запуску додатку та обфускації коду.

Розроблено рекомендації фахівцям з кібербезпеки щодо застосування технологій забезпечення безпечної роботи мобільних додатків.

Середовища розробки складаються з безлічі фреймворків, мов і архітектур,

кожна зі своїми унікальними способами взаємодії та роботи. Тому важливо адаптувати описані вище рекомендації задля ефективного захисту мобільних додатків без шкоди на швидкість, зручність та гнучкість для користувачів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Чаплієва А. О. Інструменти тестування безпеки мобільних додатків. Всеукраїнська науково-практична конференція «Цифрова трансформація кібербезпеки». Державний університет інформаційно-комунікаційних технологій. 26 квітня 2024. Тези доповідей. С. 231-233. URL: https://duikt.edu.ua/uploads/n_12581_11703414.pdf
2. Veracode, 2024 https://www.veracode.com/state-software-security-2024-report
3. Symantec, 2024 https://www.symantec.broadcom.com/hubfs/Symantec_Ransomware_Threat_Landscape_2024.pd
4. Threat Fabric, 2021 https://www.threatfabric.com/blogs/google-play-droppers
5. https://yalantis.com/blog/secure-application-development/
6. https://aws.amazon.com/what-is/sdlc/?nc1=h_ls
7. https://medium.com/@s.atmaramani/quick-recap-of-all-sdlc-models-drawbacks-comparision-when-to-use-limitations-f3a487b6074f
8. What Is DevOps? By Tyler Charboneau. Orangematter. March 21, 2022. URL: https://orangematter.solarwinds.com/2022/03/21/what-is-devops/
9. DevSecOps: What is It & Why is It Important. Bigstep. May 20, 2022. URL: https://bigsteptech.com/blog/devsecops-what-is-it-why-is-it-important/ (дата звернення)
10. The Difference Between CI/CD and DevOps—and How They Work Together to Drive Innovation. Navisite. URL: https://www.navisite.com/blog/insights/ci-cd-vs-devops/
11. Постанова №95 Правління Національного банку України Про затвердження Положення про організацію заходів із забезпечення інформаційної безпеки в банківській системі України [Чинна від 2018-03-01]
12. ДСТУ ISO/IEC 27001:2023 (ISO/IEC 27001:2022, IDT) Інформаційна безпека, кібербезпека та захист конфіденційності. Системи керування інформаційною

безпекою [Чинний від 2023-08-22]

13. ДСТУ ISO/IEC 27002:2023 (ISO/IEC 27002:2022, IDT) Інформаційна безпека, кібербезпека та захист конфіденційності. Засоби контролювання інформаційної безпеки [Чинний від 2023-08-22]
14. The National Institute of Standards and Technology (NIST) URL: https://www.nist.gov/
15. NIST SP 800-163 Rev. 1 Vetting the Security of Mobile Applications. April 2019. URL: https://csrc.nist.gov/pubs/sp/800/163/r1/final
16. OWASP Mobile Application Security. URL: https://mas.owasp.org/
17. OWASP MASVS. URL: https://mas.owasp.org/MASVS/
18. OWASP MASTG. URL: https://mas.owasp.org/MASTG/
19. Mobile Application Security Weakness Enumeration (MASWE). OWASP. URL: https://mas.owasp.org/MASWE/
20. OWASP MAS Checklist. URL: https://mas.owasp.org/checklists/
21. What is threat modeling? BlackDuck. URL: https://www.blackduck.com/glossary/what-is-threat-modeling.html
22. How to Choose a Technology Stack for Startups. OSsystem. 05.10.20. URL: https://os-system.com/blog/how-to-choose-technology-stack-for-startups-and-mvp/
23. National Vulnerability Database. NIST. URL: https://nvd.nist.gov/
24. Secure Coding Practices Checklist. OWASP. URL: https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/stable-en/02-checklist/05-checklist
25. OWASP Top Ten 2025. OWASP. URL: https://owasp.org/www-project-top-ten/
26. MobileApp-Pentest-Cheatsheet. GitHub. URL: https://github.com/tanprathan/MobileApp-Pentest-Cheatsheet
27. Snyk. URL: https://snyk.io/
28. Snyk Code. Snyk. URL: https://snyk.io/product/snyk-code/
29. DIVA. Payatu. January 1, 2019. URL: https://payatu.com/blog/damn-insecure-and-vulnerable-app-diva/

30. diva-android. GitHub. URL: https://github.com/payatu/diva-android
31. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). CWE by MITRE. URL: https://cwe.mitre.org/data/definitions/89.html
32. SafetyNet. Google Play services. URL: https://developers.google.com/android/reference/com/google/android/gms/safetynet/SafetyNet
33. JavaScript Obfuscator Tool. URL: https://obfuscator.io/#code