В. М. Савченко О. В. Мнушка

СУЧАСНІ ТЕХНОЛОГІЇ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

В. М. Савченко, О. В. Мнушка

СУЧАСНІ ТЕХНОЛОГІЇ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ

Навчально-методичний посібник
для самостійної роботи студентів другого (магістерського) рівня
денної, заочної та дуальної форм навчання
за спеціальністю 123 «Комп’ютерна інженерія»

Затверджено
редакційно-видавничою
радою НТУ «ХПІ»,
протокол № 1 від 13.02.2025 р.

Харків
НТУ «ХПІ»
2025

УДК 004.42/.49:004.056(075)
С 13

Рецензенти:

В. Д. Ковальов, д-р техн. наук, проф., ректор Донбаської державної машинобудівної академії, лауреат Державної премії України в галузі науки і техніки;

К. А. Трубчанінова, д-р. техн. наук, проф., Український державний університет залізничого транспорту

Савченко В. М.

С 13 Сучасні технології безпечного програмування : навчально-методичний посібник для самостійної роботи студентів другого (магістерського) рівня денної, заочної та дуальної форм навчання за спеціальністю 123 «Комп'ютерна інженерія» / В. М. Савченко, О. В. Мнушка. – Харків : НТУ «ХПІ», 2025. – 112 с.

ISBN 978-617-05-0553-8

У навчально-методичному посібнику розглядаються ключові концепції кіберзлочинності, методи захисту від загроз та стандарти безпечного програмування. Містить необхідний теоретичний мінімум та завдання для самостійної роботи студентів.

Призначено для самостійної роботи здобувачів вищої освіти другого (магістерського) рівня денної, заочної та дуальної форм навчання за спеціальністю 123 «Комп'ютерна інженерія».

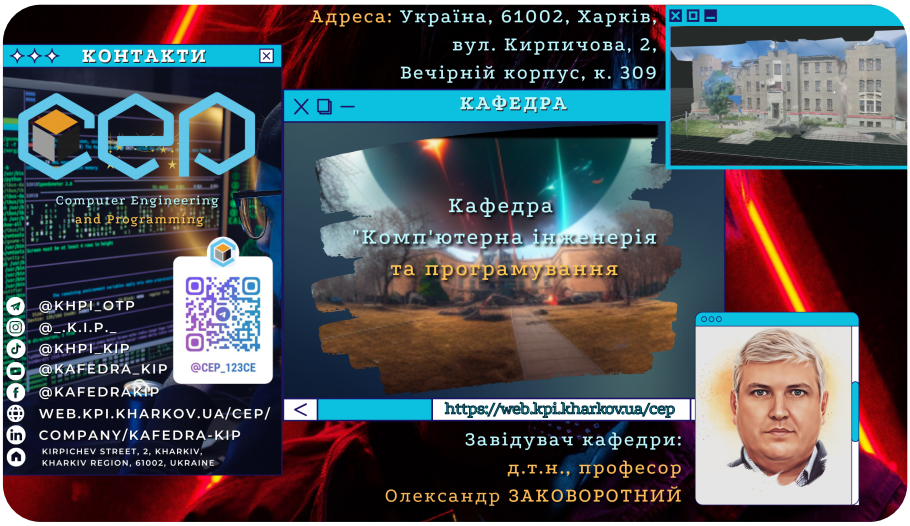
Іл. 16. Табл. 8. Бібліогр. 37 назв.

УДК 004.42/.49:004.056(075)

ISBN 978-617-05-0553-8
DOI 10.20998/978-617-05-0553-8

ЗМІСТ

Кафедра **«Комп'ютерна інженерія і програмування»** НТУ «ХПІ»
Department of **Computer Engineering and Programming** NTU «KhPI»



123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ
Програма включає вивчення архітектури апаратного та програмного забезпечення комп'ютерів, комп'ютерної схемотехніки, розробку локальних і компонент глобальної мереж, комп'ютерних ігор, інтелектуальний аналіз даних, веб розробку й фронтенд, з акцентом на математичні основи процесу обробки інформації.

123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ
Спеціальність поєднує інженерію програмного забезпечення, комп'ютерні науки та інженерію комп'ютерних компонентів та систем. Комп'ютерна інженерія — це інженерна дисципліна, яка вимагає глибоких знань у галузі теоретичних основ інформаційних технологій, інженерії та комп'ютерних наук.

СУЧАСНЕ ПРОГРАМУВАННЯ, МОБІЛЬНІ ПРИСТРОЇ ТА КОМП'ЮТЕРНІ ІГРИ
Підготовка фахівців з розробки програмного забезпечення та апаратно-програмних засобів у комп'ютерній індустрії, робототехніці, контролю та управлінні процесами; зі створення комп'ютерних ігор. Це творчий і складний процес, що вимагає високого рівня підготовки в програмуванні, алгоритмах, апаратному забезпеченні й мережних технологіях, який дозволить розробити щось дуже унікальне і популярне. Освітні траєкторії: ***«Програмування мобільних пристроїв та комп'ютерних систем»*** та ***«Інноваційний кампус»***

ПРИКЛАДНА КОМП'ЮТЕРНА ІНЖЕНЕРІЯ
Підготовка фахівців з проектування сайтів й веб-додатків та їх інтерфейсів; розробки креативної концепції сайту; створення дизайну сайту, макетів сторінок, мультимедіа-об'єктів; програмування (розробка функціональних інструментів) або інтеграції у систему управління вмістом (CMS); оптимізації та розміщення матеріалів сайту; тестування та обслуговування сайту; публікації проекту на хостингу; 3D-моделювання; програмного та інформаційного забезпечення для розумних об'єктів. Освітні траєкторії: ***«Web-дизайн та Internet-програмування»*** та ***«Інженерія захищених систем та мереж»***

ПЕРЕДМОВА

Розроблення сучасного програмного забезпечення вимагає не лише глибокого розуміння методів забезпечення його якості та ефективності, а й усвідомлення питань безпеки на всіх етапах життєвого циклу – від проєктування до підтримки. Опанування принципів безпечного програмування є важливим завданням для майбутніх фахівців у галузі *комп'ютерної інженерії*, і його успішне засвоєння значною мірою залежить від самостійної роботи студента.

Навчально-методичний посібник «Сучасні технології безпечного програмування» створено для організації самостійного опрацювання матеріалу. Він спрямований на розвиток практичних навичок роботи з науковими публікаціями, фаховою літературою, а також на застосування методів, інструментів і технологій розроблення безпечного програмного забезпечення. У ньому представлено методи аналізу баз даних вразливостей, принципи кодування, гешування та шифрування даних, а також використання цифрових підписів.

Самостійне вивчення посібника дозволяє студентам детально розглянути питання аналізу поширених загроз безпеці застосунків на основі рейтингу *OWASP Top 10*. Цей рейтинг містить узагальнені результати досліджень вразливостей програмного забезпечення за останні роки та є авторитетним незалежно від технологій і мов програмування. Усвідомлення механізмів виникнення загроз і методів їх усунення сприятиме формуванню навичок проєктування безпечного ПЗ відповідно до принципів *Secure Software Development Life Cycle (SSDLC)*. Самостійна робота з базами даних вразливостей (*Common Vulnerabilities and Exposures (CVE)*) дозволяє студентам отримати досвід пошуку, аналізу та узагальнення інформації про потенційні загрози безпеці програмних систем, вивченням історії рохзробки програмного забехзпечення.

У посібнику також розглядаються основи роботи із інструментами аналізу виконуваного коду (*YARA*, *radare2* та ін.), основ криптографії, зокрема застосування криптографічних бібліотек, таких як *OpenSSL*, використання *HMAC* (коду автентифікації повідомлень на основі гешування) та *ЕЦП* (електронного цифрового підпису). Самостійне опрацювання цих матеріалів дозволяє студентам навчитися реалізовувати механізми гешування та цифрового підпису, що є основою сучасної інформаційної безпеки.

Тема 1
ВСТУП ДО КУРСУ

Метою є ознайомлення з основами безпечного програмування, набуття навичок встановлення та налаштування інструментальних засобів.

Завдання для самостійної роботи

1. *Встановіть та налаштуйте Oracle VirtualBox* для роботи з віртуальними машинами (створіть нову віртуальну машину, оберіть потрібну операційну систему (ОС), налаштуйте мережу).
2. *Встановіть QEMU* та запустіть віртуальну машину, порівняйте її продуктивність із *VirtualBox* (використайте базові налаштування).
3. *Створіть знімок (snapshot)* у *VirtualBox* або *QEMU*, щоб мати можливість швидкого відновлення стану віртуальної машини.
4. *Ознайомтеся з базовими командами* для керування віртуальними машинами (`start`, `stop`, `pause`, `clone` тощо) у *VirtualBox* та *QEMU*.
5. *Підготуйте тестове середовище*: встановіть усі необхідні інструменти (компілятори, інтерпретатори, мережеві утиліти) у віртуальній машині для подальших завдань.

Короткі теоретичні відомості

1. Загальні відомості про дисципліну.
2. Кіберзлочинність та кіберзлочинці.
3. Комп’ютерна безпека та технічна безпека.
4. Вплив генеративного штучного інтелекту на технології безпечного програмування.

Курс сучасних технологій безпечного програмування охоплює основні принципи та практики, що стосуються розробки програмного забезпечення, яке мінімізує вразливості до кіберзагроз. Програмне забезпечення часто стає ціллю атак з боку кіберзлочинців, що робить важливим впровадження механізмів безпеки на всіх етапах розробки. Важливість проактивного підходу до виявлення та усунення можливих вразливостей є ключовою для створення надійного програмного забезпечення.

Курс також розглядає цикли розробки програмного забезпечення (*SDLC*), що включають кроки для впровадження безпеки, такі як аналіз ризиків і тестування

вразливостей. Вивчення прикладів атак допомагає уникнути подібних ситуацій в майбутньому.

Рекомендовано використовувати наступну літературу: [1–9]

Кіберзлочинність та кіберзлочинці

Кіберзлочинність включає різноманітні дії, такі як несанкціонований доступ до інформаційних систем, крадіжка конфіденційних даних, розповсюдження шкідливого програмного забезпечення, *DDoS*-атаки (атаки на відмову в обслуговуванні) та інші злочини в інтернеті. За даними звіту [10], кількість кіберзлочинів продовжує стрімко зростати. Втрати від таких злочинів у 2022 році оцінюються в більш ніж 10 мільярдів доларів, що демонструє масштаб загроз.

Кіберзлочинці активно використовують вразливості в програмному забезпеченні, такі як неоновлені системи або незахищені конфігурації серверів. Одним із яскравих прикладів стала *атака WannaCry* 2017 року, яка скористалася уразливістю в протоколі SMB для поширення шкідливого програмного забезпечення на тисячі комп'ютерів у понад 150 країнах [11].

Безпечне програмування грає ключову роль у зниженні ризиків, пов'язаних з кіберзлочинністю. *Належне проєктування* систем і регулярне оновлення програмного забезпечення значно знижує ймовірність того, що злочинці зможуть використовувати вразливості.

Сучасні кіберзлочинці використовують різноманітні підходи та техніки, серед яких можна виділити:

• *фішинг* – розповсюдження підроблених електронних листів або використання підроблених вебсайтів для крадіжки облікових даних користувачів. Згідно зі статистикою майже 1,2% всіх надісланих електронних листів є шкідливими, що в цифрах означає *3,4 мільярда фішингових листів щодня*. Очікується, що до 2024 року буде викрадено понад 33 мільйони записів, а фішингові атаки з вимогою викупу відбуватимуться кожні 11 секунд [1];

• *атаки на ланцюги постачання* – злам програмного забезпечення або обладнання під час виробничого процесу, що здійснюється, використовуючи довіру між компанією-виробником та її клієнтами. Відомий приклад – атака на *SolarWinds* у 2020 році, яка вплинула на тисячі організацій, включно з урядовими агентствами США [12];

• *експлуатація вразливостей «нульового дня»*, які ще не були виявлені розробниками, тому вони не мають виправлень безпеки, так атака на *Microsoft Exchange* у 2021 році використовувала кілька вразливостей «нульового дня» для викрадення конфіденційних даних [13].

[1] Astra Security: 81 Phishing Attack Statistics 2024: The Ultimate Insight

Розробка програмного забезпечення з акцентом на безпеку допомагає мінімізувати ризики таких атак. Впровадження *захисту на рівні коду*, на кшталт перевірки вхідних даних, обробка виключень і дотримання принципів мінімізації прав доступу, значно знижує вразливість системи.

Комп'ютерна безпека та технічна безпека

В англійській мові існує два терміни «*safety*» та «*security*», що мають український аналог «безпека».

Комп'ютерна безпека або security – це комплекс заходів, спрямованих на захист інформаційних та комп'ютерних систем від несанкціонованого доступу, модифікації або викрадення даних. Основні принципи комп'ютерної безпеки відомі як *CIA*-тріада (*Confidentiality, Integrity, Availability*):

- *конфіденційність* гарантує, що дані доступні лише тим, хто має на це право. Це досягається через шифрування даних та належну політику доступу;
- *цілісність* гарантує, що дані не можуть бути змінені без дозволу. Методи, як-от хешування і цифрові підписи, дозволяють перевіряти цілісність даних;
- *доступність* гарантує, що інформаційні системи доступні в будь-який час, коли це потрібно. Захист від *DDoS*-атак і налаштування серверів з резервними копіями допомагають підтримувати доступність.

Технічна безпека або safety включає апаратні та програмні засоби, які допомагають захищати системи:

- *мережеві брандмауери* захищають системи від несанкціонованого мережевого доступу, фільтруючи вхідний та вихідний трафік;
- *антивірусне та антишпигунське програмне забезпечення* виявляє та усуває шкідливе програмне забезпечення;
- *системи виявлення вторгнень (IDS)* та *системи попередження про вторгнення (IPS)* допомагають виявляти та запобігати атакам на ранніх етапах.

Спеціалісти у галузі *комп'ютерної інженерії*, як розробники апаратного та програмного забезпечення, повинні впроваджувати ці інструменти та застосовувати сучасні методи шифрування, автентифікації й контролю доступу, щоб забезпечити надійний захист даних і систем.

Вплив генеративного штучного інтелекту на технології безпечного програмування

Генеративний штучний інтелект (ШІ), особливо на основі таких моделей, як *GPT-3, GPT-4*, має значний потенціал у програмуванні завдяки можливості швидко розв'язувати типові задачі та автоматично генерувати код, що може суттєво пришвидшити процес розробки. Проте цей ШІ також може також генерувати код із вразливостями, наприклад, код, який генерує *GPT-3* та *GPT-4*

без належних заходів безпеки, може містити вразливості типу *SQL*-ін'єкцій або переповнення буфера.

Крім того, ШІ може бути використаним для автоматизації атак, наприклад:

• *Deepfakes* можуть бути використані для створення фальшивих відео або аудіо, що використовуються у фішингових кампаніях [14; 15].

• ШІ може допомагати автоматично сканувати програми на наявність уразливостей.

• ШІ може генерувати експлойти.

Тому однією з актуальних задач у галузі комп'ютерних наук та кібербезпеки є розробка нових підходів до забезпечення безпеки програмного забезпечення, враховуючи можливості та ризики, які несе генеративний ШІ. Такими напрямками досліджень можуть бути:

• автоматизоване тестування коду для виявлення вразливостей;

• ШІ-моделі для виявлення мережевих атак, наприклад, для аналізу мережевого трафіку та виявлення аномальної активності, яка може свідчити про атаку;

• постійний моніторинг та навчання ШІ-моделей для безперервного оновлення систем захисту в реальному часі.

Зазначимо, що потенційною та серйозною проблемою використання ШІ є *приватність даних*, що виражається у використанні даних користувача для навчання моделей ШІ, а також можливий доступ до таких даних у розробників та дослідників:

• більшість сучасних моделей ШІ, включаючи великі мовні моделі (такі як *GPT*), мають навчатися на дуже великих обсягах даних, в т.ч. зібраних з Інтернету. Це викликає серйозні питання щодо конфіденційності, оскільки ці дані можуть включати особисту інформацію або чутливі дані користувачів. Відомо про численні приклади, коли моделі ШІ генерували контент, що містив особисті дані користувачів, які не призначалися для публічного використання;

• особисті дані користувачів, зібрані під час навчання моделей, можуть бути доступними розробникам або дослідникам, що порушує приватність, відповідно до результатів досліджень [16; 17], що машинні моделі можуть «пам'ятати» фрагменти конфіденційної інформації;

• правила використання мовних моделей безпосередньо звертають увагу на можливість використання приватних даних для навчання. *OpenAI*, розробник *GPT-4*, має спеціальний розділ, де обговорюються питання безпеки та приватності даних [2].

Таким чином, генеративний ШІ може бути не тільки потужним інструментом для розробників у різних галузях, але й серйозною загрозою для кібербезпеки та приватності.

[2] https://openai.com/policies/privacy-policy/

Контрольні запитання

1. Що таке безпечне програмування і чому воно важливе?
2. Як різні етапи *SDLC* інтегруються з методами безпеки?
3. Які види кіберзлочинності є найпоширенішими сьогодні?
4. Як кіберзлочинці використовують вразливості програмного забезпечення для проведення атак?
5. Що таке вразливості «нульового дня» і як їх можна уникнути під час розробки програмного забезпечення?
6. Які основні принципи комп'ютерної безпеки повинні впроваджуватися під час розробки програмного забезпечення?
7. Як технічні засоби захисту допомагають зменшити кіберзагрози?
8. Як генеративний ШІ впливає на процес розробки безпечного програмного забезпечення?
9. Які ризики використання генеративного ШІ для створення програмного забезпечення?
10. Які заходи безпеки можна застосовувати для захисту від атак, що використовують ШІ?

Тема 2

ОГЛЯД МЕТОДІВ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ

Метою є вивчення основних підходів, методів і принципів безпечного програмування, а також набуття навичок аналізу програмного коду.

Завдання для самостійної роботи

1. *Проаналізуйте два проєкти з відкритим кодом* та визначте, які методи безпечного програмування в них використовуються (скористайтеся *GitHub*-репозиторіями та зверніть увагу на безпекові практики, що описані у `README` або `CONTRIBUTING`).
2. *Напишіть коротку настанову із безпеки з п'ятьма основними методами безпеки* для початківців (наприклад, принцип найменших привілеїв, регулярне оновлення залежностей тощо).
3. *Порівняйте статичний та динамічний аналіз коду* у вигляді зведеної таблиці (розгляньте інструменти, переваги та недоліки обох підходів).
4. *Розробіть сценарій навчання для команди розробників* щодо впровадження методів безпеки (план тренінгу, теми, приклади коду).
5. *Створіть план для аудиту безпеки програмного коду*, який можна використовувати перед кожним релізом (доступність, наявність тестів, використання статичних аналізаторів коду (лінтерів) тощо).

Короткі теоретичні відомості

1. Методи безпечного програмування.
2. Принципи захисту даних і стандарти.
3. Правові та етичні аспекти.

Методи безпечного програмування

Безпечне програмування включає кілька підходів та методів, що допомагають знизити ризики вразливостей і атак на програмне забезпечення. Одним із ключових принципів безпечного програмування є перевірка вхідних даних. Усі дані, які отримуються ззовні, повинні бути ретельно перевірені для запобігання ін'єкціям коду, таким як *SQL*-ін'єкції, або інших видів атак. Важливо враховувати

можливість вбудовування коду у команди або запити, особливо для вебзастосунків, де введення даних користувачем може стати джерелом шкідливого коду, а обробка даних відбувається, зокрема, на стороні сервера.

Інший важливий метод – фільтрація вихідних даних. Коректне кодування даних перед їхнім використанням або відправкою на зовнішні системи знижує ризик *XSS* (міжсайтових скриптових атак) та інших вразливостей, пов'язаних з небезпечними вихідними даними.

Використання криптографії також є базовим елементом безпечного програмування. Шифрування даних гарантує їхню конфіденційність та цілісність, особливо під час передачі через мережу, за умови використання сучасних та надійних алгоритмів шифрування, таких як *AES* або *RSA* [18].

Загалом, можна виділити такі основні методи безпечного програмування, які є важливими для фахівців у галузі комп'ютерної інженерії:

- перевірка вхідних даних;
- кодування вихідних даних;
- аутентифікація та управління паролями;
- керування сесіями;
- контроль та управління доступом до ресурсів;
- використання криптографічних методів, алгоритмів та протоколів;
- обробка помилок та логування;
- захист даних;
- безпека передавання даних;
- рецензування коду та статичний аналіз;
- тестування коду, зокрема на безпеку (*pen-testing*).

Принципи захисту даних і стандарти

Програмування з урахуванням безпеки повинно слідувати чітко визначеним принципам і стандартам. Наприклад, стандарт *OWASP* (*Open Web Application Security Project*) надає розробникам практичні рекомендації для виявлення та усунення найпоширеніших вразливостей у вебзастосунках. Перелік *OWASP Top 10* (Табл. 2.1) містить найбільш поширені загрози, такі як *SQL*-ін'єкції та *XSS*, і надає рекомендації для їх запобігання [18].

У версії *OWASP Top 10* за 2021 рік були внесені певні зміни до категорій (Рис. 2.1[1]). Очікується, що нова версія *OWASP Top 10* за 2025 рік буде опублікована в першій половині 2025 року.

Крім *OWASP*, рекомендації щодо безпечної розробки коду надає *SEI CERT* (*Software Engineering Institute's Computer Emergency Response Team*). *SEI CERT* розробляє стандарти для безпечного програмування на різних мовах, таких як *C*,

[1] https://owasp.org/Top10/assets/mapping.png

Таблиця 2.1 – Порівняння *OWASP Top* 10 за 2017 та 2021 роки

***OWASP Top 10* (2017)**	***OWASP Top 10* (2021)**
A01. Ін’єкції	A01. Недоліки контролю доступу
A02. Недоліки аутентифікації	A02. Криптографічні збої
A03. Розголошення конфіденційних даних	A03. Ін’єкції
A04. Зовнішні сутності *XML* (*XXE*)	A04. Небезпечний дизайн
A05. Недоліки контролю доступу	A05. Некоректна конфігурація безпеки
A06. Неправильна конфігурація безпеки	A06. Вразливі та застарілі компоненти
A07. Міжсайтове виконання сценаріїв (*XSS*)	A07. Помилки ідентифікації та автентифікації
A08. Небезпечна десеріалізація	A08. Порушення цілісності програмного забезпечення та даних
A09. Використання компонентів з відомими вразливостями	A09. Недоліки ведення журналів безпеки та моніторингу
A10. Недоліки журналювання та моніторингу	A10. Підробка запитів на стороні сервера (*SSRF*)

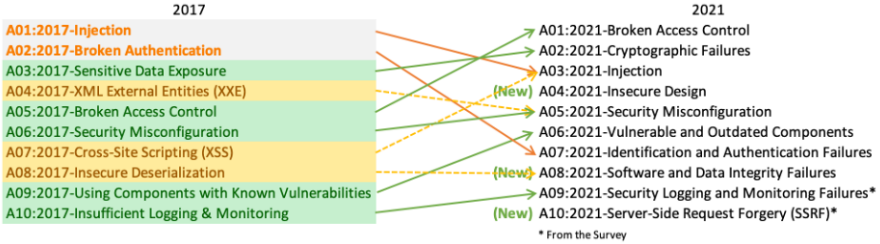


Рисунок 2.1 – Зіставлення категорій *OWASP Top 10* (2021)

C++ та *Java*, зокрема *SEI CERT C Coding Standard* [19], *SEI CERT C++ Coding Standard* [20] та *SEI CERT Oracle Coding Standard for* Java [21], які надають правила та рекомендації розробникам щодо уникнення поширених помилок безпеки під час написання коду. Ці стандарти охоплюють питання управління пам’яттю, уникнення переповнення буфера, роботу з файлами й процесами, обробку виключень тощо. Актуальні версії стандартів можна знайти на сайті *SEI* [22].

Ці правила та рекомендації, разом з іншими стандартами, такими як *MITRE*, *ISO/IEC TS 17961:2013*, використовуються в статичних аналізаторах коду, таких

як *Cppcheck*[2] або *clang-tidy*[3], що дозволяє виявляти потенційні проблеми на ранніх етапах розробки.

Впровадження стандартів безпечної розробки підвищує якість коду, безпеку систем та зменшує кількість уразливих місць у програмному забезпеченні [23].

До інших важливих стандартів належать *ISO/IEC 27001*, який встановлює вимоги до систем управління інформаційною безпекою, та *PCI DSS* (*Payment Card Industry Data Security Standard*), що регулює безпеку даних платіжних карток [24].

Правові та етичні аспекти

Безпечне програмування має також важливі правові та етичні аспекти. Закони різних країн, такі як Загальний регламент про захист даних (*GDPR*) в Європейському Союзі, вимагають від розробників дотримання жорстких вимог щодо захисту персональних даних користувачів [23].

Розробники повинні не лише дотримуватись цих законів, але й діяти етично. Відповідальна поведінка включає оперативне виправлення вразливостей і відповідальне розкриття інформації про них. Наприклад, етичний хакінг (або *white-hat hacking*) сприяє покращенню безпеки програмного забезпечення через виявлення вразливостей до їхнього використання злочинцями.

Контрольні запитання

1. Які основні методи безпечного програмування?
2. Чому потрібно перевіряти вхідні дані та як це впливає на безпеку?
3. Що таке фільтрація вихідних даних і які атаки вона допомагає запобігти?
4. Яке значення має криптографія у забезпеченні безпеки даних?
5. Що таке *OWASP Top 10* і які загрози вона включає?
6. Які основні рекомендації *SEI CERT* щодо безпечного програмування?
7. Як правові регламенти, такі як *GDPR*, впливають на безпечне програмування?
8. Чому важливі етичні принципи у контексті безпечного програмування?
9. Які основні правові вимоги для захисту даних у різних юрисдикціях?
10. Як відповідально розробники повинні діяти у випадку виявлення вразливостей?

[2] https://cppcheck.sourceforge.io/
[3] https://clang.llvm.org/extra/clang-tidy/

Тема 3

ПОШКОДЖЕННЯ ПАМ'ЯТІ ТА ПЕРЕПОВНЕННЯ

Метою є дослідження вразливостей, пов'язаних із переповненням буфера та помилками в роботі з пам'яттю, а також набуття практичних навичок аналізу вразливостей програмного коду.

Завдання для самостійної роботи

1. *Знайдіть приклад коду з переповненням буфера* (на *C* і *C++*) та виправте його, пояснивши, де була помилка та як її уникнути.
2. *Напишіть програму на* C *і* C++, яка демонструє контроль меж масивів (використовуйте перевірку індексів під час доступу до елементів масиву).
3. *Протестуйте написаний код за допомогою Valgrind* для виявлення витоків пам'яті та переповнень буфера.
4. *Дослідіть, як ASLR (Address Space Layout Randomization)* допомагає запобігти експлуатації переповнень (поясніть механізм та його переваги).
5. *Створіть презентацію про найпоширеніші помилки, пов'язані з керуванням пам'яттю*, та дайте рекомендації щодо їх уникнення.

Короткі теоретичні відомості

1. Пошкодження пам'яті.
2. Переповнення буфера, стека та купи.
3. Методи статичного та динамічного аналізу.

Пошкодження пам'яті

Пошкодження пам'яті є однією з найсерйозніших вразливостей безпеки, що виникає через некоректне управління пам'яттю в програмному забезпеченні. Така вразливість може призводити до непередбачуваних збоїв у роботі програм або надавати зловмисникам можливість виконувати атаки, включно з виконанням довільного коду або ескалацією привілеїв. Найчастіше пошкодження пам'яті виникає внаслідок доступу до пам'яті, яка не була коректно виділена або вже звільнена. Це призводить до типових вразливостей, таких як переповнення буфера (*buffer overflow*), стека (*stack overflow*) та купи (*heap overflow*).

Ці вразливості можуть бути експлуатовані через недоліки бібліотек програмування або бібліотек часу виконання операційних систем, таких як *libc*

або *libc++*. Наприклад, класичні атаки типу переповнення буфера дозволяють зловмисникам переписати дані в пам'яті, що може призвести до виконання шкідливого коду на рівні операційної системи. Такі атаки, як *Return-Oriented Programming (ROP)* або *Jump-Oriented Programming (JOP)*, маніпулюють послідовністю інструкцій програми для виконання шкідливих дій, що може призвести до повного контролю над системою.

Атаки на пошкодження пам'яті є одними з найнебезпечніших, оскільки дозволяють зловмисникам маніпулювати критичними даними або змінювати потік виконання програми. Це робить систему вразливою до експлойтів, які можуть використовуватись для виконання привілейованого коду або витоку конфіденційних даних. Багато сучасних систем безпеки, таких як *Data Execution Prevention (DEP)* і *Address Space Layout Randomization (ASLR)*, були розроблені, щоб запобігати подібним атакам, але їх обхід можливий у випадку складних експлойтів, які комбінують кілька вразливостей.

Переповнення буфера, стека та купи

При виклику функції стек використовується для збереження додаткових аргументів, які не вміщаються у виділені регістри (Рис. 3.1[1]). Також у стеку завжди зберігається адреса повернення (`return address`), яка дозволяє функції після завершення повернутися до місця виклику. Стек вирівнюється на 16 байтів, щоб забезпечити коректність викликів і доступ до даних.

Calling convention визначає, як передаються аргументи функцій і як функції повертають значення на низькому рівні в програмуванні. У *Windows x64* (*Microsoft ABI*) перші 4 аргументи передаються через регістри `RCX`, `RDX`, `R8`, `R9`, а решта – через стек. Також виділяється 32 байти «*shadow space*» на стеку для збереження цих регістрів, навіть якщо вони не використовуються. Повернення значень здійснюється через `RAX` для цілочислових і `XMM0` для рухомих точок. Стек вирівнюється на 16 байтів перед викликом функції.

***Windows x64* (*Microsoft ABI*)**

- перші чотири цілочислові або вказівникові аргументи передаються через `RCX`, `RDX`, `R8`, `R9`;
- аргументи після 4-го передаються через стек;
- виділяється 32 байти «*shadow space*» на стеку для збереження перших чотирьох аргументів, навіть якщо вони не використовуються;
- повернення значень: `RAX` для цілочислових і `XMM0` для рухомої точки;
- стек вирівнюється на 16 байтів перед викликом функції.

[1] Eli Bendersky. Stack frame layout on x86-64

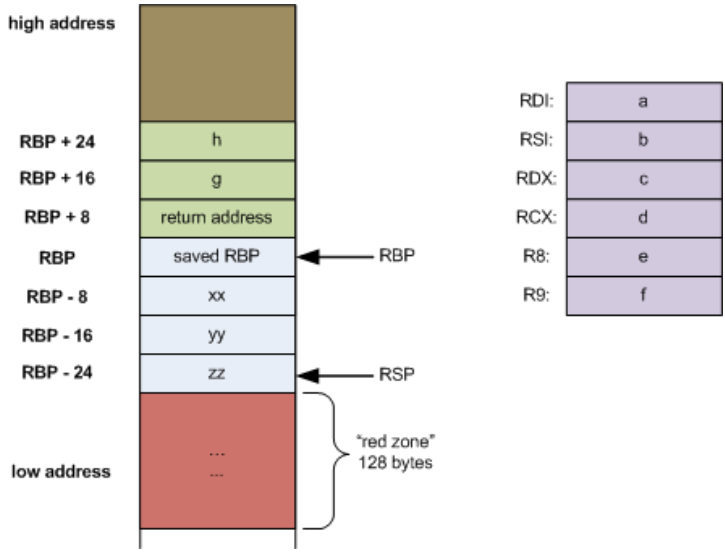


Рисунок 3.1 – *x86_64* стек (*Linux*)

***Linux x64* (*System V ABI*)**

• перші шість цілочислових аргументів передаються через `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`;

• аргументи після шостого передаються через стек;

• немає «*shadow space*»;

• повернення значень: `RAX` для цілочислових і `XMM0` для рухомої точки;

• стек також вирівнюється на 16 байтів;

• у Linux використовується *128-бітна red zone*, яка знаходиться під `RSP` і може бути використана для тимчасових змінних без модифікації `RSP`. Вона не використовується під час викликів функцій.

Переповнення стека є поширеним типом вразливостей, пов'язаних із неправильним управлінням пам'яттю, але воно не є прямим наслідком переповнення буфера. *Переповнення стека* відбувається, коли кількість даних або функцій, що зберігаються у стеку, перевищує його розмір, що може призвести до порушення логіки виконання програми або до атак на безпеку, зокрема переписування вказівників на повернення (*return pointers*). Це може дозволити зловмисникам отримати контроль над потоком виконання програми.

Переповнення буфера – це вразливість, при якій дані, що записуються в буфер, перевищують його розмір, що спричиняє запис за межі відведеної пам'яті

(Рис. 3.2[2]). Це може призвести до змін у важливих структурах програми, таких як змінні, вказівники або адреси повернення, що може використовуватись для атаки типу *виконання довільного коду*. Зазвичай такі вразливості призводять до значних загроз безпеці, як-от атаки типу *stack smashing* або *heap corruption*.

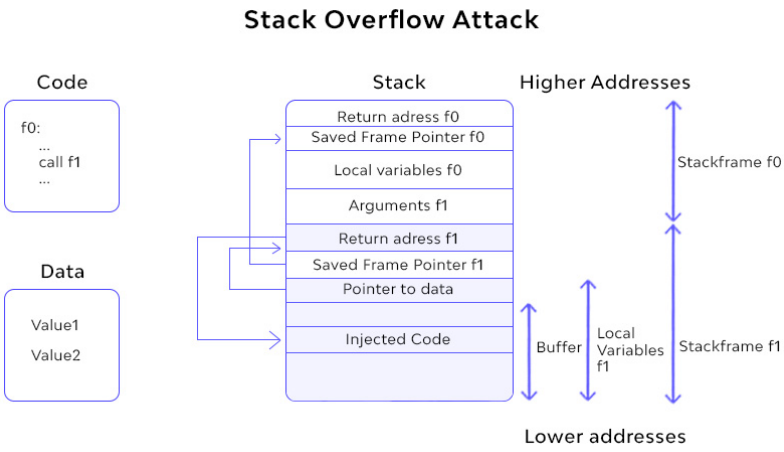


Рисунок 3.2 – Переповнення стека

Переповнення купи виникає у мовах програмування *C* і *C++*, що дозволяють прямий доступ до пам'яті. Купа використовується для динамічного виділення пам'яті під час виконання програми, і якщо пам'ять не управляється правильно (наприклад, через недоліки в коректному виділенні та звільненні динамічної пам'яті), це може призвести до вразливостей, таких як *використання не проініціалізованих вказівників* або *double free* (повторне звільнення пам'яті).

Поширені види атак:

• *Stack smashing* – це метод атаки, при якому зловмисник змінює вміст стека через переповнення буфера, що може призвести до виконання шкідливого коду;

• *Heap corruption* – це порушення цілісності даних у купі, що може виникати через неправильне управління динамічною пам'яттю;

• *ASLR* (Randomization of memory layout) та *Stack Canaries* (контроль цілісності стека) є механізмами захисту, які зменшують ризик успішної експлуатації переповнень стека та буфера.

[2]Wallarm: What is a Buffer Overflow Attack?

Приклад переповнення стека

Розглянемо простий приклад функції на мові *C*, яка викликає переповнення стека через те, що всі локальні змінні та адреси повернення під час виклику функції, зазвичай, зберігаються у стеку:

```
void vulnerable_function() {
        char buffer[10]; // C-рядок
        gets(buffer);    // небезпечна функція
    }
```

Якщо користувач введе більше ніж 10 символів, це призведе до переповнення буфера, що може дозволити зловмиснику перезаписати вказівник на повернення і виконати шкідливий код.

Приклад переповнення купи.

У випадку переповнення купи, зловмисник може використовувати неправильне виділення динамічної пам'яті для зміни важливих даних у програмі. Наприклад:

```
void vulnerable_heap() {
        char *buffer1 = (char *)malloc(10);
        char *buffer2 = (char *)malloc(10);
        strcpy(buffer1, "123456789012345");  // переповнення буфера
        ↪  buffer1
        free(buffer1);
    }
```

Тут функція `strcpy()` записує більше даних у `buffer1`, ніж було виділено, що може призвести до пошкодження інших змінних або вказівників у купі.

Зазначимо, що стандарти мов *C* та *C++* не ігнорують проблему переповнення буфера. Деякі небезпечні функції, такі як `gets`, були видалені у стандарті *C11* через їх потенційну небезпеку. Інші функції мають «безпечні» варіанти з контролем розміру буфера. Наприклад, замість `strcpy` рекомендується використовувати `strncpy`, а замість `sprintf` – `snprintf`, які дозволяють вказувати максимальний розмір буфера і запобігають переписуванню пам'яті за межами буфера (Табл. 3.1[3]).

[3]Список залежить від стандарту та версії компілятора

Крім того, деякі функції були позначені як *deprecated* (застарілі). Наприклад, у стандарті *C++* `gets` також було позначено як застаріле і видалене через відсутність можливості контролювати розмір введених даних. Такі функції, як `strcat`, можуть також бути небезпечними й рекомендується використовувати їх «безпечні» варіанти (`strncat`) для запобігання переповненню буфера.

Стандарти *C11* і *C++11* суттєво підвищили безпеку управління пам'яттю шляхом введення нових безпечних функцій та вилучення старих, небезпечних підходів до обробки рядків.

Таблиця 3.1 – Небезпечні функції та їх безпечні аналоги у C *і* C++

Небезпечна функція	***ANSI* C *і* C++**	***Microsoft VS* C *і* C++**
`strcpy`	`strncpy`	`strcpy_s`, `strlcpy`
`strcat`	`strncat`	`strcat_s`, `strlcat`
`sprintf`	`snprintf`	—
`vsprintf`	`vsnprintf`	—
`gets`	`fgets`	`gets_s`
`makepath`	—	`_makepath_s`
`_splitpath`	—	`_splitpath_s`
`scanf`	—	`sscanf_s`
`sscanf`	—	`sscanf_s`
`snscanf`	—	`_snscanf_s`
`strlen`	—	`strnlen_s`

Методи статичного та динамічного аналізу

Щоб виявити та запобігти пошкодженню пам'яті й переповненню, використовуються методи статичного та динамічного аналізу. Статичний аналіз передбачає перевірку вихідного коду програми без її виконання. Інструменти статичного аналізу можуть виявити потенційні проблеми, як-от небезпечне управління пам'яттю або використання неперевірених даних. Це дозволяє виправити помилки до того, як вони призведуть до вразливостей у реальному середовищі.

Динамічний аналіз, на відміну від статичного, проводиться під час виконання програми. Цей метод дозволяє виявляти проблеми в реальному часі, наприклад, неправильне звільнення пам'яті або використання пошкоджених вказівників. Такі інструменти, як *Valgrind*[4] або *AddressSanitizer*[5], активно вико-

[4]https://valgrind.org/

[5]https://github.com/google/sanitizers/wiki/addresssanitizer

ристовуються для пошуку проблем, пов'язаних із пошкодженням пам'яті, під час тестування програмного забезпечення. Поєднання статичного та динамічного аналізу є ефективним підходом для створення безпечних програм.

Контрольні запитання

1. Що таке пошкодження пам'яті і які його основні причини?
2. Як пошкодження пам'яті може бути використано зловмисниками для атак на програмне забезпечення?
3. Яка різниця між переповненням стека і переповненням купи?
4. Як переповнення буфера може призвести до порушення безпеки програми?
5. Які безпечні практики управління пам'яттю можуть запобігти пошкодженню пам'яті?
6. Що таке статичний аналіз і як він допомагає виявляти вразливості в програмному забезпеченні?
7. Які інструменти використовуються для статичного аналізу коду?
8. Що таке динамічний аналіз і в яких випадках він є ефективним?
9. Які інструменти динамічного аналізу допомагають виявляти проблеми управління пам'яттю?
10. Як поєднання статичного і динамічного аналізу підвищує безпеку програмного забезпечення?

Тема 4

ІН’ЄКЦІЇ КОДУ

Метою є аналіз природи типових атак із використанням ін’єкції коду(*SQL*, *XSS* тощо) та методів їхнього запобігання, набуття практичних навичок виявлення ін’єкцій коду.

Завдання для самостійної роботи

1. *Напишіть SQL-запит з параметризованими даними* (*Prepared Statement*) для запобігання *SQL*-ін’єкціям.
2. *Проаналізуйте журнали вебсервера* (наприклад, Apache чи Nginx) та виявіть можливі спроби *SQL*-ін’єкцій.
3. *Виправте Python скрипт*, який вразливий до командних ін’єкцій (перевірте спосіб виконання системних команд і замініть його безпечнішим варіантом).
4. *Проведіть демонстрацію XSS-атаки* на тестовому вебзастосунку (створіть просту сторінку з формою, що відображає введені дані).
5. *Розробіть фільтр для вхідних даних*, який блокує підозрілі символи та перевіряє дані на валідність перед обробкою (регулярні вирази, whitelisting тощо).

Короткі теоретичні відомості

1. Ін’єкції коду – загальний огляд.
2. *SQL*-ін’єкції.
3. Безпека скриптових мов та командних оболонок.

Ін’єкції коду – загальний огляд

Ін’єкції коду – це один із найбільш поширених і небезпечних видів атак на програмне забезпечення. Ці атаки відбуваються, коли зловмисник вносить шкідливий код у поле вводу, яке не було належно захищене, і цей код виконується системою. Найбільш поширеними прикладами таких атак є ін’єкції команд операційної системи та *SQL*-ін’єкції.

Ін’єкції команд операційної системи (Рис. 4.1[1]) дозволяють зловмиснику

[1]MITRE: CWE-78

виконувати небажані команди безпосередньо в операційній системі, тоді як *SQL*-ін’єкції спрямовані у бази даних, де виконуються шкідливі SQL-запити.

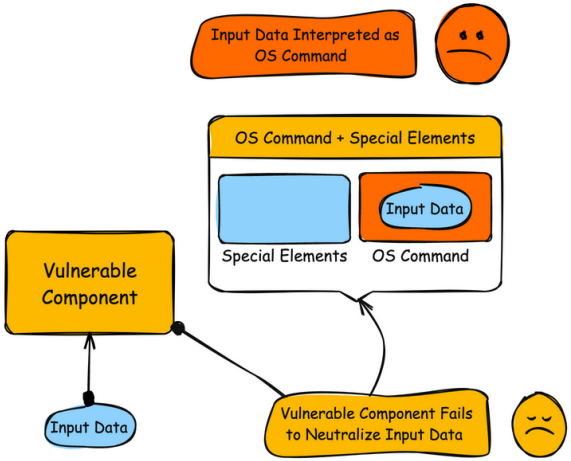


Рисунок 4.1 – Ін’єкція команди ОС

Прикладом ін’єкції команд операційної системи може бути ситуація, коли вебзастосунок приймає введення користувача для виконання команди системи, наприклад, запиту на команду `ping`:

```
ping 127.0.0.1
```

Якщо введення не було коректно перевірено, зловмисник може додати до команди власний шкідливий код:

```
ping 127.0.0.1; rm -rf /
```

Це призведе до виконання команди видалення файлів на сервері.

SQL-ін’єкції

SQL-ін’єкція – це техніка, при якій зловмисник додає або змінює *SQL*-запити через введення у вебформи або *URL*-параметри, що призводить до виконання небажаних дій на базі даних (Рис. 4.2[2]).

[2] xkcd: Exploits of a Mom

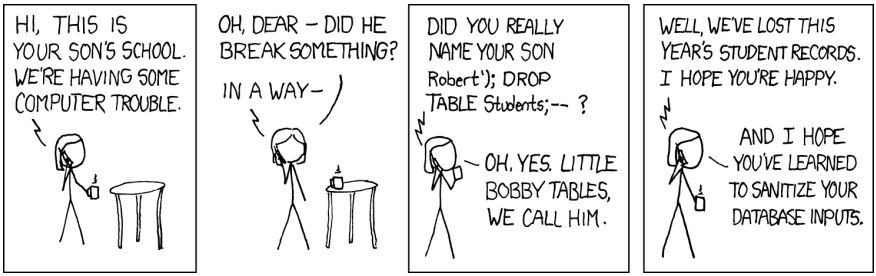


Рисунок 4.2 – *SQL*-ін’єкція

Наприклад, якщо *SQL*-запит виглядає так:

```
SELECT * FROM users WHERE username = '$username' AND password =
↪  '$password';
```

Зловмисник може вставити в поле вводу наступний код:

```
' OR '1'='1
```

Це перетворить запит у:

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '';
```

У такому випадку запит поверне всі записи з бази даних, оскільки умова завжди буде правдивою. Це дозволить зловмиснику обійти автентифікацію та отримати доступ до конфіденційних даних.

Щоб запобігти *SQL*-ін’єкціям, необхідно використовувати параметризовані запити та уникати прямого включення введених користувачем даних у SQL-запити [3]. Далі наведено приклад безпечного використання *SQL* у *PHP* з підготовленими виразами:

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE username = :username
↪  AND password = :password');
$stmt->execute(['username' => $username, 'password' => $password]);
```

[3] OWASP: SQL Injection Prevention Cheat Sheet

Тепер ми можемо розглянути приклад, наведений на рис. 4.2. Нехай для вставлення даних у таблицю використовується наступна команда:

```
INSERT INTO Students (firstname) VALUES ('Elaine');
```

Тоді, якщо ім’я буде `name = 'Robert'); DROP TABLE STUDENTS; --'` як на рис. 4.2, буде виконано наступний запит:

```
INSERT INTO Students (firstname) VALUES ('Robert'); DROP TABLE
↪  STUDENTS; --'
```

Який трансформується у три команди SQL та призведе до видалення таблиці `Students`:

```
INSERT INTO Students (firstname) VALUES ('Robert');
DROP TABLE Students;
--';
```

ORM-фреймворк (від англ. *Object-Relational Mapping*) – це бібліотека або інструмент, який дозволяє працювати з базами даних використовуючи об’єктно-орієнтований підхід, а не прямі *SQL*-запити. *ORM*-фреймворк *транслює* об’єкти та класи мови програмування у таблиці бази даних – і навпаки.

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    ...
new_user = User(name="Volodymyr")
```

Популярні *ORM*-фреймворки:
- *Python*: *SQLAlchemy*, *Django ORM*
- *Java*: *Hibernate*
- *C#/.NET*: *Entity Framework*
- *PHP*: *Eloquent* (*Laravel*)
- *JavaScript*: *Sequelize* (для *Node.js*)

Безпека скриптових мов та командних оболонок

Скриптові мови програмування, такі як *PHP*, *Python* і *JavaScript*, часто є мішенню для ін'єкцій коду через їхню гнучкість і широке застосування в вебзастосунках. Небезпека полягає у можливості виконання шкідливого коду при взаємодії зі сторонніми даними. Наприклад, вразливість до *XSS*, якщо на вебсторінці відображається неперевірений введений користувачем код.

Також ін'єкції команд через командні оболонки, як-от *Bash* або *PowerShell*, можуть призвести до серйозних наслідків. Зловмисники можуть отримати доступ до системи, виконуючи небезпечні команди з використанням вразливих скриптів. Один зі способів захисту – це уникати прямого виконання зовнішніх команд, використовувати функції для ізоляції введених даних або повністю відмовитися від виконання команд на стороні сервера.

Контрольні запитання

1. Що таке ін'єкція коду та як її використовують в атаках на програмне забезпечення?
2. Як працюють ін'єкції команд операційної системи? Наведіть приклад.
3. Які потенційні наслідки можуть мати *SQL*-ін'єкції для баз даних?
4. Як можна захиститися від *SQL*-ін'єкцій? Які техніки є найбільш ефективними?
5. Які ризики виникають при використанні скриптових мов програмування та командних оболонок?
6. Що таке міжсайтові скриптові атаки (*XSS*) і як вони пов'язані з ін'єкціями коду?
7. Як підготовлені (prepared) вирази в *SQL* допомагають запобігти ін'єкціям?
8. Які загальні принципи безпеки слід застосовувати у скриптах, що обробляють введення користувача?
9. Які методи можна використовувати для захисту командних оболонок від ін'єкцій?
10. Як можна виявити ін'єкції коду за допомогою статичного та динамічного аналізу?

Тема 5

ПРОБЛЕМА КОНКУРЕНТНОСТІ (ПЕРЕГОНІВ)

Метою є вивчення загроз, пов'язаних з умовами перегонів, та способів їх уникнення у багатопотокових програмах, набуття практичних навичок аналізу багатопотокових програм.

Завдання для самостійної роботи

1. *Знайдіть race condition у коді мовою Java* та усуньте його за допомогою ключового слова `synchronized` або інших механізмів синхронізації.
2. *Напишіть приклад коду з використанням м'ютексів* (mutex) у *C* і *C++* або *Python* для керування доступом до спільного ресурсу.
3. *Протестуйте багатопотоковий застосунок* (написаний мовою *C* і *C++* або *Java*) за допомогою інструменту *ThreadSanitizer*[1] та проаналізуйте результати.
4. *Порівняйте рішення для уникнення race condition у мовах C++ та* Python, пояснюючи особливості підходів (синхронізація, *GIL* у *Python* тощо).
5. *Створіть схему взаємодії потоків* для складного багатопотокового сценарію (наприклад, взаємодія між чергою завдань та робочими потоками).

Короткі теоретичні відомості

1. Загальний огляд проблеми конкурентності.
2. Методи запобігання перегонам.
3. Наслідки проблеми конкурентності.

Загальний огляд проблеми конкурентності

Проблема конкурентності виникає, коли кілька потоків або процесів одночасно звертаються до одного й того ж ресурсу (наприклад, пам'яті або файлу) без належної синхронізації. Це може призвести до некоректної поведінки програми та серйозних вразливостей, таких як пошкодження даних або витік конфіденційної інформації (Табл. 5.1).

[1] https://github.com/google/sanitizers/wiki/threadsanitizercppmanual

Таблиця 5.1 – Основні типи вразливостей, пов’язані з конкурентними умовами

Тип вразливості	**Опис**	***CWE***
Race Conditions (Конкурентні умови)	Декілька потоків чи процесів одночасно намагаються змінити або отримати доступ до спільного ресурсу (пам’ять, файли, змінні) без належної синхронізації.	*CWE-362*
Data Races Перегони даних)	Декілька потоків одночасно змінюють одну й ту ж змінну або область пам’яті без належної синхронізації, що призводить до непередбачуваних результатів.	*CWE-367*
Deadlocks (Взаємні блокування)	Декілька потоків чи процесів блокують ресурси, які вони чекають один від одного, що призводить до повної зупинки їхньої роботи.	*CWE-833*
Live Locks (Живі блокування)	Ситуація, коли процеси продовжують виконувати дії, що блокують один одного, без можливості завершення.	*CWE-667*
Priority Inversion (Інверсія пріоритетів)	Процес із нижчим пріоритетом утримує ресурс, необхідний процесу з вищим пріоритетом, що призводить до затримок у роботі системи.	*CWE-410*
Double-Fetch (Подвійне отримання)	Дані з ресурсу читаються двічі, й між цими операціями зловмисник може змінити ці дані, що призводить до непередбачуваних результатів.	*CWE-367*
Lock Contention (Конкуренція за блокування)	Конкуренція за блокування відбувається, коли кілька потоків змагаються за один й той самий ресурс, що спричиняє затримки й зниження продуктивності.	*CWE-667*
Permission Races (Перегони прав доступу)	Декілька процесів намагаються змінити чи використовувати ресурси без відповідних прав доступу.	*CWE-275*
Races with Temporary Files (Перегони тимчасових файлів)	Вразливості виникають, коли тимчасові файли створюються і використовуються без належної синхронізації.	*CWE-377*
Directory Position Race (Перегони при обробці каталогів)	Перегони виникають при змінах у каталогах між моментами перевірки та використання.	*CWE-346*

Атака *TOCTOU*

TOCTOU (*Time-of-Check to Time-of-Use*) – це клас вразливостей, які виникають у багатозадачних і багатопотокових системах через вікно між перевіркою

ресурсу (*time of check*) та його використанням (*time of use*). Це дозволяє зловмиснику змінити стан ресурсу в проміжок між перевіркою та його використанням, що може призвести до небажаних наслідків.

Основна ідея *TOCTOU*:

Time of Check – це момент часу, коли програма перевіряє доступність або стан ресурсу (файл, пам'ять, мережеве з'єднання тощо) і приймає рішення на основі цієї перевірки. *Time of Use* – це момент часу, коли програма виконує операцію на ресурсі, вважаючи, що його стан не змінився після перевірки. *Проблема* – між цими двома моментами часу інший процес або потік може змінити стан ресурсу, що призведе до виконання небажаних або небезпечних дій.

Уявімо програму, яка перевіряє, чи існує файл, а потім відкриває його для запису:

```
if (access("somefile.txt", W_OK) == 0) {
    // файл існує і можна записувати
}
fd = open("somefile.txt", O_WRONLY);
```

Між цими операціями (перевіркою та відкриттям) інший процес може замінити файл на символьне посилання до іншого файлу, до якого програма не повинна мати доступ, що може призвести до некоректного запису даних.

TOCTOU-вразливості можуть призвести до таких наслідків:

- некоректний доступ до даних;
- експлуатація системи через зміну стану ресурсів;
- потенційна ескалація привілеїв.

Способи запобігання *TOCTOU*

1. *атомарні операції*. Потрібно використовувати системні виклики, які виконують перевірку та використання ресурсу в одній операції. Наприклад, можна гарантувати, що файл буде створений лише якщо він не існує, уникаючи *TOCTOU*-атаки:

```
fd = open("somefile.txt", O_WRONLY | O_CREAT | O_EXCL);
```

2. *принцип найменших привілеїв* шляхом обмеження доступу до файлів і ресурсів;

3. *міжпроцесна синхронізація* шляхом блокування ресурсів (*locking*), щоб уникнути некоректного доступу між процесами чи потоками;

4. *фіксація шляху файлів* шляхом безпосереднього визначення шляхів до файлів (`fdstat()` безпечніший за `stat()`).

До цієї ж групи вразливостей відносять *Permission Races* (перегони прав доступу), *Races with temporary files* (перегони тимчасових файлів), *Directory position race* (перегони при обробці каталогів) та ін. (Табл. 5.2).

Таблиця 5.2 – Основні вразливості, пов'язані з конкурентними умовами

Вразливість	**Опис**
CWE-367: *Time-of-check Time-of-use (TOCTOU)*	Файл або ресурс перевіряється перед використанням, але змінюється до операції використання.
CWE-362: *Concurrent Execution using Shared Resource*	Ресурси одночасно використовуються кількома потоками без належної синхронізації.
CWE-364: *Signal Handler Race Condition*	Обробник сигналів викликається під час зміни глобального стану, що може призвести до корупції даних або збоїв.
CWE-363: *Race Condition Enabling Link Following*	Стан файлу чи каталогу перевіряється до його доступу, але ресурс може бути замінений посиланням перед використанням.
CWE-368: *Context Switching Race Condition*	Виникає під час перемикання контексту між потоками, коли можливі атаки на цілісність або конфіденційність даних.

Перегони даних

Одним із найбільш поширених сценаріїв конкурентності є умови перегонів (*race conditions*). Вони виникають, коли два або більше потоків одночасно змінюють спільний ресурс, і результат їхньої роботи залежить від порядку виконання цих потоків. Оскільки порядок виконання не завжди передбачуваний, це може призвести до непередбачуваних результатів.

Доступ до спільної змінної декількома потоками може призвести до виникнення стану перегонів даних якщо:

- здійснюється (потенційно) одночасно;
- принаймні один із доступів є записом.

Стан перегонів даних є умовою перегонів на рівні атомарних операцій з пам'яттю. Він є основною причиною багатьох складних помилок у багатопотокових програмах.

Перегони даних зазвичай є випадковими помилками:

- призводять до недетермінованої поведінки програми
- помилкова поведінка може бути дуже рідкісною (навіть у тестах);
- відтворення помилки може бути складною задачею.

Ці підходи зазвичай застосовуються лише в експертному коді бібліотек або при розробці ядра ОС. Звичайні розробники застосунків повинні прагнути писати програми, вільні від перегонів даних.

Приклад умови перегонів

Розглянемо приклад на мові C++, де дві функції одночасно працюють з однією змінною:

```
int shared_resource = 0;

void increment() {
        shared_resource++;
    }

void decrement() {
        shared_resource--;
    }

int main() {
        std::thread t1(increment);
        std::thread t2(decrement);
        t1.join();
        t2.join();
        printf("%d\n", shared_resource);
    }
```

У прикладі потоки одночасно звертаються до змінної `shared_resource`. Через відсутність синхронізації результат може бути непередбачуваним – значення змінної може бути 0, 1 або -1, залежно від порядку виконання потоків.

У разі виникнення стану перегонів в операційній системі з'являються потенційні можливості їх використання зловмисниками для спричинення помилкового розрахунку або непослідовного значення шляхом:

- впливу на планування потоків;
- багаторазового виконання певної дії.

Методи запобігання перегонам

Для запобігання умовам перегонів використовуються різні методи синхронізації, такі як м'ютекси (*mutex*), семафори або блокування. М'ютекси дозволяють гарантувати, що лише один потік може працювати з ресурсом у певний момент часу, тоді як інші потоки чекають свого доступу.

Приклад використання м'ютекса для запобігання перегонам.

Розглянемо той самий приклад, але з використанням м'ютекса для синхронізації доступу до спільного ресурсу:

```
#include <mutex>

int shared_resource = 0;
std::mutex mtx;

void increment() {
        mtx.lock();
        shared_resource++;
        mtx.unlock();
    }

void decrement() {
        mtx.lock();
        shared_resource--;
        mtx.unlock();
    }

int main() {
        std::thread t1(increment);
        std::thread t2(decrement);
        t1.join();
        t2.join();
        printf("%d\n", shared_resource);
    }
```

У цьому прикладі м'ютекс (`mtx`) гарантує, що лише один потік може змінювати `shared_resource` в один і той самий час. Це дозволяє уникнути умов перегонів і забезпечує правильний результат.

Інші методи, такі як семафори та блокування читання/запису, також можуть бути використані для розв'язання проблеми конкурентності, залежно від конкретної ситуації. Семафори забезпечують обмежений доступ до ресурсів, тоді як блокування читання/запису дозволяє кільком потокам одночасно читати дані, але тільки одному потоку записувати їх.

Наслідки проблеми конкурентності

Якщо проблема конкурентності не вирішена належним чином, це може призвести до серйозних вразливостей у програмному забезпеченні. Наприклад, у багатопотокових вебсерверах чи базах даних відсутність синхронізації може призвести до некоректної обробки запитів, втрати даних або витоку конфіденційної інформації. Зловмисники можуть використовувати ці вразливості для запуску атак або доступу до неавторизованих ресурсів.

Приклад атаки на умови перегонів.

Зловмисник може намагатися використати вразливість у доступі до файлів, де два потоки одночасно намагаються змінити один і той самий файл:

```
open("/tmp/file", O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
write(fd, buffer, sizeof(buffer));
close(fd);
```

Якщо зловмисник зуміє змінити файл під час обробки операцій, він може змусити програму працювати з іншим файлом, що призведе до втрати або викрадення даних.

Контрольні запитання

1. Що таке умови перегонів і як вони виникають у програмному забезпеченні?
2. Як проблема конкурентності може призвести до некоректної роботи програми?
3. Які методи можна використовувати для запобігання умовам перегонів?
4. Що таке м'ютекси і як вони допомагають розв'язувати проблему умов перегонів?
5. Як семафори можуть використовуватися для синхронізації доступу до ресурсів?
6. Які є наслідки для безпеки при неправильному управлінні конкурентними потоками?

7. Як умови перегонів можуть бути використані зловмисниками для атак на програмне забезпечення?

8. Які інші методи синхронізації потоків можуть бути використані для запобігання перегонам?

9. Які проблеми можуть виникати в багатопотокових програмах через відсутність належної синхронізації?

10. Які інструменти аналізу можна використовувати для виявлення умов перегонів у програмному забезпеченні?

Тема 6

ШКІДЛИВЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ТА СОЦІАЛЬНА ІНЖЕНЕРІЯ

Метою є ознайомлення з видами шкідливого програмного забезпечення та методами соціальної інженерії, набуття практичних навичок протидії цим методам.

Завдання для самостійної роботи

1. *Проаналізуйте зразок шкідливого програмного забезпечення (ПЗ)* за допомогою *Sandbox* (наприклад, *Any.Run*, *Cuckoo*) та складіть звіт про його поведінку. Рекомендується використовувати безпечні навчальні зразки шкідливого ПЗ, такі як *EICAR test file* або контрольовані зразки з проєктів на кшталт *theZoo*, виключно в ізольованих середовищах.
2. *Напишіть сценарій тренування для співробітників щодо виявлення фішингових листів*, включно з порадами з перевірки відправника, посилань і вкладень.
3. *Дослідіть, як антивірусне ПЗ визначає шкідливі файли* (наприклад, сигнатурний аналіз, евристичний підхід, поведінковий аналіз).
4. *Створіть інфографіку про основні методи соціальної інженерії* (*phishing*, *pretexting*, *baiting*, *tailgating* тощо).
5. *Проведіть імітацію атаки соціальної інженерії* у своїй команді (з дозволу учасників) та проаналізуйте результати.

Короткі теоретичні відомості

1. Шкідливе програмне забезпечення.
2. Методи програмування стійкості проти шкідливого коду.
3. Соціальна інженерія та її вплив на кібербезпеку.
4. Методи захисту від соціальної інженерії.

Шкідливе програмне забезпечення

Щкідливе програмне забезпечення (*malware*) є одним із найпоширеніших інструментів зловмисників для здійснення атак на системи та мережі. Таке програмне забезпечення може включати *віруси* (*viruses*), *хробаки* (*worms*), *троянські коні* (*trojan horses*), *руткіти* (*rootkits*), *рекламне* (*adware*) та *шпигунське*

(*spyware*) програмне забезпечення. Основною метою таких програм є експлуатація уразливостей (*vulnerabilities*) у комп'ютерних системах для отримання несанкціонованого доступу (*unauthorized access*), викрадення даних (*data theft*) або пошкодження ресурсів (*resource damage*).

Віруси можуть поширюватися через заражені файли або макроси в документах *Microsoft Office*. Троянські програми маскуються під звичайні застосунки, але насправді відкривають доступ до системи зловмисникам. Руткіти можуть приховувати активність інших шкідливих програм, надаючи зловмисникам тривалий доступ до системи.

Одними з найбільш небезпечних є віруси-вимагачі (*ransomware*). Це шкідливе програмне забезпечення блокує доступ до файлів або цілої системи та вимагає певне відшкодування за їх розблокування. Такий вірус шифрує дані користувача на локальних носіях, після чого робота комп'ютера блокується, а за розблокування (отримання ключа для розшифрування) зловмисники вимагають викуп, зазвичай у криптовалюті, на що дається обмежений час. У разі невиконання вимог, через певний час дані можуть бути втрачені назавжди. Ці атаки спрямовані як на звичайних користувачів, так і на великі корпорації, уряди та інші організації, що призводить до паралічу їхньої роботи.

Однією з найвідоміших атак вірусів-вимагачів є атака вірусу *WannaCry* у 2017 році, яка вразила понад 200 тисяч комп'ютерів у 150 країнах. Цей вірус використовував уразливість в операційних системах *Windows* для швидкого поширення через мережі, завдаючи великих збитків компаніям і державним органам. Інший відомий приклад – вірус *NotPetya*, який вразив корпоративні мережі в Україні та поширився по всьому світу, завдавши збитків на мільярди доларів.

Методи програмування стійкості проти шкідливого коду

Під час проєктування архітектури застосунків архітектори програмного забезпечення мають враховувати всі потенційні ризики, в т.ч. загрози кібербезпеки. Як правило, безпечне програмування передбачає дотримання типових практик:

- коректна обробка помилок і винятків;
- застосування принципу найменших привілеїв;
- використання сучасних і безпечних бібліотек та фреймворків;
- ведення якісної документації та коментування коду;
- проведення якісного рецензування коду.

Аналіз кодової бази та тестування

Для забезпечення високого рівня кібербезпеки IT-продукту використовують наступні види аналізу:

• *інтеграційне та функціональне тестування*, яке перевіряє взаємодію між різними компонентами програми та дозволяє виявити проблеми, пов'язані з інтеграцією та логікою програми;

• *статичний аналіз коду*, що виконується до компіляції програми і дає змогу виявити розповсюджені помилки, такі як витоки пам'яті, переповнення буфера або недотримання стандартів програмування. Для цього використовуються спеціалізовані інструменти – статичні аналізатори;

• *динамічний аналіз коду*, який перевіряє програму під час її виконання і дозволяє оцінити її реальний стан в умовах навантаження, використання ресурсів, та витоків пам'яті. Такий аналіз також включає інструментування коду та емуляцію кібератак для перевірки стійкості програми до атак типу *SQL Injection*, *XSS* та *CSRF*.

Щоб запобігати, виявляти та своєчасно усувати вразливості в програмному забезпеченні, варто дотримуватися наступних рекомендацій та стандартів:

Дотримання стандартів кіберзахисту

Розробка та підтримка IT-продуктів повинні відповідати міжнародним стандартам безпеки:

• використання у технологіях розробки

1. *Open Web Application Security Project (OWASP)* у галузі розробки вебзастосунків;

2. *Payment Card Industry Data Security Standard (PCI DSS)* у галузі розробки систем електронних платежів;

• побудова процесів з урахуванням питань кібербезпеки

1. *NIST Cybersecurity Framework* загальні рекомендації з безпеки від Національного інституту стандартів і технологій США;

2. *ISO 27001* міжнародний стандарт управління інформаційною безпекою, що дозволяє побудувати процеси у компанії;

3. *GDPR (General Data Protection Regulation)*, що визначає загальний регламент захисту персональних даних в Європейському Союзі.

Дотримання цих вимог та стандартів позитивно впливає на запобігання ризикам та забезпечення безпеки коду в умовах сучасних кіберзагроз.

Також слід відзначити важливу роль *тестування* для забезпечення безпеки продуктів. Тестування має бути безперервним процесом, незалежно від стадії життєвого циклу продукту. Тестування, засноване на використанні *CI/CD*(*Continuous Integration/Continuous Delivery*), має на меті перевірку безпеки

та усунення вразливостей як до, так і після релізу продукту, оскільки загрози постійно еволюціонують, і нові вразливості можуть бути виявлені навіть у добре протестованому коді.

Соціальна інженерія та її вплив на кібербезпеку

Соціальна інженерія – це техніка, за допомогою якої зловмисники маніпулюють людьми для отримання доступу до конфіденційної інформації або систем. Замість того, щоб атакувати безпосередньо технічну сторону системи, зловмисники можуть використовувати людський фактор як слабку ланку безпеки. Типовими прикладами соціальної інженерії є фішинг, коли користувачів обманом змушують розкрити паролі або інші конфіденційні дані через підроблені вебсайти або електронні листи.

Фішинг

За час існування *фішінгу* кіберзлочинці розробили чимало методів та продовжують їх удосконалювати увесь час[1]:

- *фішинг в електронних листах* (*email phishing*), в якому зловмисники підробляють гіперпосилання в електронних листах, мімікруючи під відомі компанії, банки, компанію жертви тощо;
- *фішинг за допомогою зловмисних програм* (*malware-based phishing*), в якому використовують шкідливе програмне забезпечення, замасковане під безпечні файли, наприклад, резюме або банківські виписки, у вкладеннях електронних листів. Відкриття таких файлів може завдати шкоди IT-системам;
- *цільовий фішинг* (*spear phishing*) є атакою на конкретних обраних осіб, робочі чи особисті дані яких були зібрані заздалегідь. Така атака може обходити засоби кібербезпеки, оскільки спеціально націлена на жертву;
- *полювання на велику здобич* (*whaling*) орієнтоване на бізнесменів, знаменитостей чи інших осіб, які мають значні активи. Фазі атаки передує фаза ретельного аналізу жертви атаки та чекають сприятливого моменту для крадіжки облікових даних або іншої конфіденційної інформації;
- *смсшинг* (*smishing*) є фішингом через *SMS*-повідомлення, де шахраї маскуються під відомі компанії-надавачі сервісних та інших послуг – *Amazon*, *Нова пошта*, *Монобанк* тощо. Такі повідомлення можуть виглядати дуже особистими, що робить їх особливо ефективними;
- *вішинг* (*vishing*) є атакою через телефонні дзвінки, де зловмисники маскуються під співробітників інформаційно-довідкових служб і виманюють персональну інформацію.

[1] Microsoft: Що таке фішинг?

Приклад атаки фішингу

Користувач отримує електронний лист, що імітує повідомлення від банку:

```
Dear Customer,

Please click on the link below to verify your account information:
http://fakebank.com/login

Thank you,
Your Bank
```

Після переходу за посиланням користувач потрапляє на підроблений вебсайт, який виглядає як легітимний. Якщо користувач введе свої дані для входу, вони потраплять до зловмисників.

Методи захисту від соціальної інженерії

Для захисту від соціальної інженерії важливо навчати користувачів про загрози й методи атак. Окрім цього, необхідно впроваджувати багатофакторну автентифікацію, яка ускладнює зловмисникам отримання доступу навіть за наявності вкрадених даних.

Приклад багатофакторної автентифікації

```
if (user_authenticated && sms_code_verified) {
        // Надати доступ до системи
    }
```

Навіть якщо зловмисник отримає пароль користувача, йому все одно потрібен буде код підтвердження, відправлений на телефон користувача.

Фреймворк *Cyber Kill Chain*

Cyber Kill Chain фреймворк (розроблений Lockheed Martin[2]) є моделлю для виявлення та попередження активності кібервторгнень. Модель визначає, що саме має зробити противник, щоб досягти своєї мети. Основна ідея – допомогти кращому розумінню ролей сторін атаки та захисту (Табл. 6.1).

[2]https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html

Заслуговують на увагу та вивчення наступні фреймворки та моделі кібербезпеки:

1. *MITRE ATT&CK* (*Adversarial Tactics, Techniques, and Common Knowledge*) – база знань, яка містить інформацію про тактики, техніки та процедури (*TTPs*) кіберзлочинців. Використовується для аналізу загроз, оцінки безпеки та розробки захисних заходів;

2. *Diamond Model of Intrusion Analysis* – модель аналізу кібератак, що описує чотири основні компоненти будь-якої атаки: зловмисник (*adversary*), жертва (*victim*), інфраструктура (*infrastructure*) та методи (*capability*). Модель допомагає виявляти зв'язки між цими компонентами для кращого реагування на загрози;

3. *OODA Loop* (*Observe, Orient, Decide, Act*) – модель прийняття рішень, яка використовується в кібербезпеці. Вона описує процес реагування на атаки;

4. *Purdue Enterprise Reference Architecture* (*PERA*) – архітектурна модель для опису взаємодії різних рівнів систем у промислових мережах. Вона допомагає проєктувати та захищати системи автоматизації на всіх рівнях;

5. *NIST Cybersecurity Framework* – стандарт кібербезпеки від Національного інституту стандартів і технологій (*National Institute of Standards and Technology, NIST*). Фреймворк включає етапи ідентифікації, захисту, виявлення, реагування та відновлення для управління ризиками кібербезпеки.

Таблиця 6.1 – Етапи *Cyber Kill Chain*

Етап	**Опис**
Розвідка (*Reconnaissance*)	Збір максимально можливої інформації про ціль.
Озброєння (*Weaponization*)	Створення шкідливого інструменту, такого як вірус або хробак.
Доставлення (*Delivery*)	Доставлення шкідливого ПЗ жертві через вразливості в системі.
Експлуатація (*Exploitation*)	Використання вразливості для виконання шкідливого коду на системі жертви.
Інсталяція (*Installation*)	Самоінсталяція шкідливого ПЗ та його поширення в середовищі.
Управління та контроль (*Command and Control, C2*)	Створення зворотного каналу до сервера зловмисника.
Дії на об'єкті (*Actions on Objectives*)	Ексфільтрація даних, їх знищення або шифрування для викупу.

Контрольні запитання

1. Що таке шкідливе програмне забезпечення і які його основні види?

2. Як працює вірус або троянська програма, і як вони можуть експлуатувати уразливості?

3. Які методи програмування можуть забезпечити стійкість до шкідливого коду?

4. Як шифрування може допомогти захистити інформацію від атак?

5. Що таке соціальна інженерія і як вона використовується в атаках на користувачів?

6. Які приклади атак на основі соціальної інженерії найбільш поширені?

7. Як фішинг може використовуватися для викрадення конфіденційної інформації?

8. Які методи можна використовувати для захисту від соціальної інженерії?

9. Як працює багатофакторна автентифікація і чому вона важлива для безпеки?

10. Як можна навчати користувачів протидії атакам на основі соціальної інженерії?

Тема 7

МЕТОДИ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ У МОВАХ ПРОГРАМУВАННЯ

Метою є вивчення особливостей забезпечення безпеки в різних мовах програмування, набуття навичок аналізу методів безпечного програмування у цих мовах.

Завдання для самостійної роботи

1. *Порівняйте вбудовані механізми безпеки в Rust та C++* (borrow checker у Rust, виключення у C++ тощо).
2. *Напишіть код на* Java *з використанням SecurityManager* та поясніть, як він обмежує доступ до системних ресурсів.
3. *Використайте статичний аналізатор (наприклад, ESLint)* для перевірки коду *Java*Script на поширені помилки та вразливості.
4. *Реалізуйте обробку помилок у* Python з використанням блоків `try-except`, перевіривши, як це впливає на стабільність програми.
5. *Дослідіть, як мова Go запобігає переповненням буфера*, звернувши увагу на особливості управління пам'яттю.

Короткі теоретичні відомості

1. Методи безпечного програмування у мовах програмування:
- *C* і *C++*.
- *Java*.
- *Python*.
- *Java*Script.
2. Порівняння підходів до забезпечення безпечного коду.

Методи безпечного програмування у мовах програмування

Кожна мова програмування має свої особливості та вразливості, через що підходи до забезпечення безпеки можуть відрізнятися. У цьому розділі ми розглянемо методи безпечного програмування для таких мов як C, C++, *Java*, *Python* та *Java*Script. Порівняння цих методів допоможе краще зрозуміти специфіку кожної мови та найефективніші підходи до запобігання вразливостям.

C і *C++*: управління пам'яттю та захист від переповнення буфера

Мови *C* і *C++* забезпечують програмістам низькорівневий доступ до пам'яті, що робить їх потужними, але водночас вразливими до помилок у керуванні пам'яттю. Однією з найбільших проблем у цих мовах є переповнення буфера, яке може призвести до пошкодження пам'яті або навіть до виконання шкідливого коду.

Оновними документами, що визначають методи безпечного програмування у цих мовах є *SEI CERT C Coding Standard* [19], *SEI CERT C++ Coding Standard* [20]. Як зазначено у [20], «метою …є розробка безпечних, надійних та захищених систем, наприклад, шляхом усунення невизначених поведінок, які можуть призвести до непередбачуваних дій програми та вразливостей, що можуть бути використані зловмисниками.»

SEI CERT C++ Coding Standard містить наступні групи правил, що відносяться для певних особливостей мови програмування *C++*:

1. *DCL* – оголошення та ініціалізація;
2. *EXP* – вирази;
3. *INT* – цілі числа;
4. *FLP* – числа з рухомою крапкою (комою);
5. *ARR* – масиви;
6. *STR* – рядки;
7. *MEM* – управління пам'яттю;
8. *FIO* – введення/виведення;
9. *ENV* – оточення;
10. *SIG* – сигнали;
11. *ERR* – обробка помилок;
12. *OBJ* – об'єктноорієнтоване програмування;
13. *CON* – паралелізм;
14. *MSC* – різне;
15. *API* – інтерфейс програмування застосунків;

Розглянемо приклад вимоги *SEI CERT C++ Coding Standard* із розділу керування пам'яттю.

MEM53-CPP. Явно створюйте та знищуйте об'єкти при ручному управлінні життєвим циклом об'єктів. Створення динамічно виділених об'єктів у *C++* відбувається у два етапи. Перший етап відповідає за виділення достатньої кількості пам'яті для зберігання об'єкта, а другий етап – за ініціалізацію нововиділеного блоку пам'яті залежно від типу об'єкта, що створюється.

Цей тип помилок є одним із найпоширеніших, коли об'єкт містить поля даних, ресурси під які виділяються за допомогою `new()`.

Оцінка ризиків для *MEM53-CPP*[20]

- Правило: *MEM53-CPP*.
- Серйозність: Висока.
- Ймовірність: Ймовірна.
- Вартість виправлення: Середня.
- Пріоритет: *18*.
- Рівень: *L1*.

Особливо цікавими для розробників є приклади, які поділяються на два типи: такі, що задовольняють вимоги *compliant*, та такі, що не задовольняють вимоги *noncompliant*, що допомагає краще зрозуміти сутність правила.

SEI CERT C Coding Standard містить правила та рекомендації, в іншому він схожий на варіант для *C++*:

- *правила* – це обов'язкові вимоги, яких *необхідно* дотримуватися для уникнення вразливостей та невизначеної поведінки програми. Вони зазвичай можуть бути автоматично перевірені за допомогою інструментів, і порушення правила призводить до серйозних наслідків для безпеки та стабільності програми;
- *рекомендації* – це *рекомендовані* практики, які покращують якість і безпеку коду, але не завжди є обов'язковими. Вони менш суворі, і їх порушення може призвести до погіршення якості коду, але не становить такої критичної загрози, як порушення правил.

Обидва стандарти для мов *C* і *C++* є у вільному доступі (версія 2016 року є найновішою).

Java

Java є мовою з керованим середовищем виконання (*JVM*), що забезпечує підвищений захист у контексті управління пам'яттю, оскільки автоматичне прибирання сміття (*garbage collection*) запобігає витокам пам'яті. *Java* є однією з основних мов програмування бізнес-застосунків, тому помилки в програмах на *Java* можуть мати критичні наслідки для користувачів. Саме тому існує версія стандарту *SEI CERT Oracle Coding Standard for Java* [21].

Оригінальна версія була оприлюднена у 2011 році, а актуальна присутня на сайті *SEI CERT* [22]. Його структура близька до структури *SEI CERT C Coding Standard*. Правила також поділяються на різні категорії, кожна з яких відображає окремий аспект програмування. Ці категорії охоплюють теми від управління ресурсами та пам'яттю до контролю за потоком виконання та безпечної роботи з даними. Правила є обов'язковими для виконання. Їх порушення може призвести до серйозних вразливостей, які зловмисники можуть використати для атаки на програму або систему

Java також має специфічні виклики безпеки, зумовлені її використанням.

Розглянемо приклад правила із розділу *Input Validation and Data Sanitization (IDS)*[1].

IDS00-J. Запобігайте SQL-ін'єкціям. Уразливості *SQL*-ін'єкцій виникають у програмах, де елементи *SQL*-запиту надходять із ненадійного джерела. Без належних заходів безпеки ненадійні дані можуть зловмисно змінити запит, що призведе до витоку інформації або модифікації даних. Основними засобами запобігання *SQL*-ін'єкціям є *санітизація* (*sanitization*) і *валідація* (*validation*) даних, які зазвичай реалізуються через *параметризовані запити* (*parameterized queries*) та *збережені процедури* (*stored procedures*).

Фрагмент прикладу коду із *SEI CERT Oracle Coding Standard for Java*, який показує, як запобігати *SQL*-ін'єкціям. Його особливістю є використання методів `set*()` класу `PreparedStatement` для забезпечення жорсткої перевірки типів. Це зменшує ризик *SQL*-ін'єкцій, оскільки введені дані автоматично екрануються. *Підготовлені запити* (*prepared statements*) можна використовувати для вставлення даних у базу даних.

```
...
String pwd = hashPassword(password);
// Validate username length
if (username.length() > 8) {
        // Handle error
    }
String sqlString =
"select * from db_user where username=? and password=?";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, username);
stmt.setString(2, pwd);
ResultSet rs = stmt.executeQuery();
if (!rs.next()) {
        throw new SecurityException("User name or password
        ↪  incorrect");
    }
...
```

Оцінка ризиків для IDS00-J [22]:

- *правило*: IDS00-J;
- *серйозність*: висока;
- *ймовірність*: ймовірна;

[1] SEI: IDS00-J. Prevent SQL injection

- *вартість виправлення*: середня;
- *пріоритет*: *P18*;
- *рівень*: *L1*.

Python

Аналогічно до *C*, *C++* та *Java*, *Python* та програми на його основі через свою популярність, можуть бути об'єктами атаки.

Для *Python* також створено стандарт безпечного програмування [25], але під егідою *Open Source Security Foundation (OpenSSF)*, яка продовжила розробку, започатковану за ініціативою компанії *Ericsson* для покращення безпечного кодування на *Python* (для цілей навчання). Структура стандарту безпосередньо базується на *CWE Pillar Weakness* [26]. Проєкт перебуває на стадії розробки та наразі містить лише приклади коду.

Розглянемо приклад[2].

CWE-197. Numeric Truncation Error *(Помилка числового усічення)*. Забезпечуйте передбачувані результати в циклах, використовуючи змінні типу `int` замість `float` як лічильники. Арифметика з рухомою комою може представляти лише скінченну підмножину дійсних чисел за стандартом *IEEE 754*, наприклад, число $0.\overline{5}$ може представлятись як `0.5555555555555556`, що також обговорюється в *CWE-1339:* Precision Loss or Inaccuracy of Real Number *(Недостатня точність або похибка дійсного числа)*.

Ця помилка також актуальна для *Java* (*NUM09-J*) та *C* (*FLP30-C*).

Нижче наведено фрагменти коду, що показують правильний та хибний підходи при використанні циклів.

```
""" Non-compliant Code Example """
counter = 0.0
while counter <= 1.0:
if counter == 0.8:
print("we reached 0.8")
break  # never going to reach this
counter += 0.1
```

```
""" Compliant Code Example """
counter = 0
while counter <= 10:
```

[2] OpenSSF: CWE-197: Numeric Truncation Error

```
value = counter/10
if value == 0.8:
print("we reached 0.8")
break
counter += 1
```

JavaScript

JavaScript широко використовується у вебзастосунках, що робить його мішенню для атак, зокрема *міжсайтових скриптових атак* (*cross-site scripting*, *XSS*). *XSS* виникає, коли зловмисник вставляє шкідливий код у вебсторінку, який виконується у браузері користувача.

На сьогодні існує стандарт безпечного програмування на *JavaScript* від *Checkmarx* [27]. Стандарт *JavaScript Web Application Secure Coding Practices* організований у розділи, що охоплюють критичні аспекти безпеки у веброзробці за допомогою *JavaScript*. До цих розділів входять *валідація введених даних* (*input validation*), *кодування вихідних даних* (*output encoding*), *управління автентифікацією* (*authentication management*), *робота з сесіями* (*session handling*) та *обробка помилок* (*error handling*). Стандарт пропонує логічний підхід до забезпечення безпеки: починаючи з перевірки вхідних даних і закінчуючи безпечними шаблонами кодування для роботи з чутливою інформацією та зовнішніми системами.

Одним із ключових акцентів стандарту є *валідація введених даних*, що передбачає обробку всіх введених даних як ненадійних за замовчуванням. Підкреслюється важливість розділення даних за джерелом – на надійні та ненадійні, а також надаються рекомендації щодо безпечних практик валідації. Цей розділ охоплює захист від таких поширених вразливостей, як *SQL*-ін'єкції та *міжсайтовий скриптинг* (*XSS*), пропонуючи правильні методи валідації та кодування.

Стандарт описує основні типи вразливостей, які виникають у вебзастосунках:

- *SQL-ін'єкції* (*SQL injection*): ризик виникає через включення ненадійних даних у *SQL*-запити. Для запобігання цьому рекомендується використовувати *параметризовані запити* (*parameterized queries*) та *збережені процедури* (*stored procedures*);
- *Міжсайтовий скриптинг (XSS)* (*Cross-Site Scripting*): для запобігання *XSS*-вразливостям слід застосовувати *кодування вихідних даних* (*output encoding*), щоб дані безпечно відображалися на сторінках без виконання шкідливих скриптів;
- *NoSQL-ін'єкції* (*NoSQL injection*): для захисту *NoSQL*-баз даних наводяться найкращі практики безпечного формування запитів до бази даних.

Окремий розділ присвячений *автентифікації* та правильному управлінню сесіями. Наголошується на важливості безпечного зберігання облікових даних і

дотримання політики безпечних паролів. Також детально описуються методи управління сесіями, такі як безпечна обробка cookie та завершення сесій для захисту від захоплення сесії.

У стандарті також розглядаються *криптографічні практики*, що стосуються використання безпечних алгоритмів та управління ключами для захисту чутливої інформації. Крім того, стандарт наголошує на правильній обробці помилок, а саме, необхідності уникнення виведення системних даних через повідомлення про помилки та застосування надійних механізмів логування.

Порівняння підходів до забезпечення безпечного коду

• мови *C* і *C++* надають найбільше можливостей для низькорівневого доступу до пам'яті, але це також робить їх найбільш вразливими до проблем управління пам'яттю (переповнення буфера, витоки пам'яті). Використання сучасних функцій для роботи з пам'яттю та перевірка введених користувачами даних є критично важливими;

• *Java* менш вразлива до проблем пам'яті завдяки автоматичному керуванню пам'яттю, але вимагає обережної обробки виключень та серіалізації, щоб уникнути витоків інформації та несанкціонованого доступу;

• *Python* зі своєю динамічною типізацією є зручною, але вимагає додаткової обережності при роботі з бібліотеками для запобігання вразливостей;

• *JavaScript* особливо вразливий до *XSS*-атак, тому розробники повинні забезпечити правильне екранування введених користувачами даних і уникати небезпечних конструкцій, що дозволяють виконання неперевіреного коду.

Контрольні запитання

1. Які основні проблеми безпеки виникають при роботі з *C* і *C++*?
2. Як *Java* обробляє пам'ять і які методи безпеки потрібно враховувати?
3. Чому використання `eval` у *Python* є небезпечним і як цього уникнути?
4. Що таке *XSS*-атаки у *JavaScript* і як їх можна запобігти?
5. Як *параметризовані запити* (*parameterized queries*) можуть допомогти уникнути ін'єкцій у *SQL*-запитах?
6. Як серіалізація в *Java* може призвести до вразливостей і як цього уникнути?
7. Чому управління пам'яттю є важливим у *C* і *C++* і які методи можна використовувати для запобігання витокам пам'яті?
8. Які переваги й недоліки використання автоматичного прибирання сміття (*garbage collection*) у *Java*?
9. Як можна забезпечити безпеку під час роботи з введеними користувачами даними в *Python*?

10. Які методи автентифікації та авторизації можуть використовуватися для захисту хмарних застосунків?

11. Які ризики виникають при використанні небезпечних функцій у *JavaScript* і як їх можна уникнути?

12. Які уразливості можуть виникнути при використанні сторонніх бібліотек і пакетів у *Python*, *Java* або *C++*?

13. Як працює механізм перевірки типів у *TypeScript* і як це підвищує безпеку?

14. Чому важливо обмежувати доступ до середовищ виконання (*runtime environments*) у мовах типу *JavaScript* або *Node.js*?

15. Які кращі практики варто застосовувати для зберігання та перевірки паролів у багатомовних програмних системах?

Тема 8

ВІРТУАЛІЗАЦІЯ ТА ХМАРНІ ЗАСТОСУНКИ

Метою є ознайомлення з питаннями безпеки у хмарних технологіях та при використанні віртуалізації, набуття практичних навичок використання інструментів для аналізу вразливостей.

Завдання для самостійної роботи

1. *Налаштуйте ізольоване середовище за допомогою Docker* для тестування застосунків (створіть `Dockerfile`, `Docker Compose` та зберіть образ).
2. *Проаналізуйте ризики безпеки* в хмарній інфраструктурі (*AWS* або *Azure*) і складіть список можливих векторів атаки.
3. *Дослідіть політики доступу для віртуальних машин у VMware*, використовуючи рольові моделі доступу (*RBAC*).
4. *Проведіть тест на проникнення хмарного застосунку* за допомогою *OWASP ZAP* та задокументуйте знайдені вразливості.
5. *Порівняйте безпеку контейнерів Docker та віртуальних машин*, зосередившись на рівнях ізоляції та гіпервізорах.

Короткі теоретичні відомості

1. Віртуальні машини та їхні засоби забезпечення безпеки.
2. Ізоляція процесів та дозволи як методи захисту.
3. Архітектура хмарних застосунків.
4. Проблеми приватності даних на хмарних сервісах.

Віртуальні машини та їхні засоби забезпечення безпеки

Віртуалізація – це технологія, яку використовують для створення віртуальних представлень серверів, сховищ, мереж та інших фізичних машин (Табл. 8.1 Рис. 8.1). Віртуальне програмне забезпечення імітує функції фізичного обладнання для одночасного запуску кількох віртуальних машин на одній фізичній машині. Використання віртуалізації дозволяє сучасним компаніям оптимізувати використання їхніх обчислювальних ресурсів, включаючи перехід на повне використання хмарних сервісів. За допомогою віртуалізації у вигляді хмарних сервісів компанії можуть повністю відмовитися від використання фізичних серверів, що спрощує обслуговування інфраструктури [28].

Таблиця 8.1 – Типи віртуалізації

Тип віртуалізації	Короткий опис
Віртуалізація застосунків	Дозволяє запускати програми в ізольованих середовищах, незалежно від ОС або апаратного забезпечення.
Віртуалізація мережі	Створення віртуальних мережевих інфраструктур на основі фізичних мереж для гнучкого управління трафіком.
Віртуалізація робочих столів	Дає змогу користувачам отримати доступ до віддалених робочих столів з будь-якого пристрою.
Віртуалізація сховища	Об'єднує фізичні ресурси сховищ у віртуальні, щоб спростити управління даними.
Віртуалізація серверів	Дозволяє запустити кілька віртуальних серверів на одному фізичному сервері для підвищення ефективності.
Віртуалізація даних	Дає можливість отримувати доступ до даних незалежно від їхнього фізичного місцеперебування або формату.

Віртуальні машини (VM) є інструментом для створення ізольованих середовищ виконання на одному фізичному комп'ютері. Вони дозволяють одночасно виконувати декілька операційних систем (*guest OS*) на одному сервері (*host OS*). Кожна з гостьових ОС працює у власному ізольованому середовищі, яке забезпечується гіпервізором. Віртуалізація надає певні переваги у питаннях безпеки, оскільки ізоляція між віртуальними машинами запобігає втручанню однієї системи в роботу іншої (Рис. 8.1).

Технічні аспекти ізоляції включають використання різних рівнів віртуалізації, таких як гіпервізори типу 1 (*bare-metal*) і типу 2 (що працюють на базі основної операційної системи). Гіпервізори типу 1, наприклад, *VMware ESXi* або *Microsoft Hyper-V*, забезпечують вищий рівень ізоляції та продуктивності завдяки прямому доступу до апаратного забезпечення. Натомість гіпервізори типу 2, такі як *VirtualBox* або *VMware Workstation*, працюють у середовищі ОС, що може створювати додаткові накладні витрати.

Крім того, апаратна віртуалізація, підтримувана сучасними процесорами, такими як *Intel VT-x* або *AMD-V*, забезпечує ще більший рівень безпеки та ізоляції між віртуальними машинами, що мінімізує ризик виходу за межі ізольованого середовища (*VM escape*).

Основні засоби безпеки віртуальних машин включають:

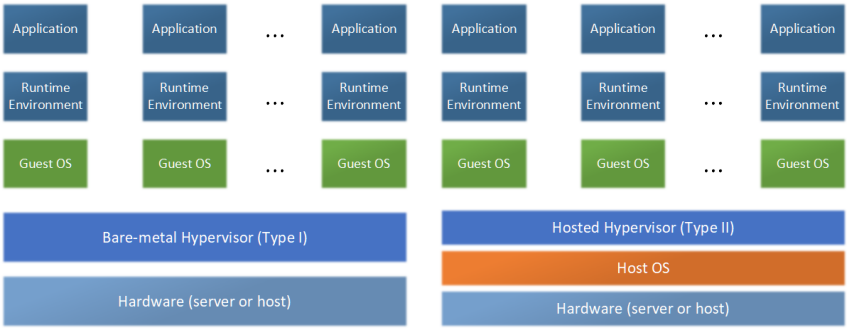


Рисунок 8.1 – Віртуалізація

• безпека гіпервізора, зокрема захист від «*breakout machine*»;

• ізоляція, сегментація та мікросегментація ресурсів, що забезпечується використанням кожною VM певної частки процесорного часу, пам'яті та інших ресурсів, що унеможливлює прямий доступ між *VM*;

• мережева безпека шляхом використання систем виявлення (*IDS*) та запобігати вторгненням (*IPS*);

• безпечні засоби зберігання та відновлення даних;

• регулярне оновлення виправлень безпеки.

Зазначені методи забезпечують «тонке налаштування та адміністрування» безпеки віртуальних середовищ та їх стабільну роботу.

Ізоляція процесів та керування пам'яттю у захищеному режимі

Ізоляція процесів є базовим елементом забезпечення безпеки як у віртуалізованих середовищах, так і на фізичних серверах. В операційних системах кожен процес працює у власному адресному просторі, що захищає його від несанкціонованого доступу з боку інших процесів. Це означає, що кожен процес не має доступу до пам'яті інших процесів, а доступ до апаратних ресурсів суворо контролюється.

Ізоляція процесів підтримується як на рівні апаратного, так і програмного забезпечення. Сучасні процесори мають кілька *кілець захисту* (*protection rings*), що забезпечують виконання процесів з різними рівнями привілеїв. Наприклад, у процесорах архітектури x86 використовується чотири кільця:

• *кільце 0* – це найбільш привілейований рівень, де виконується ядро операційної системи. Цей рівень має повний доступ до всіх апаратних ресурсів;

• *кільця 1 і 2* – використовуються для драйверів і служб з середнім рівнем привілеїв;

• *кільце 3* – призначене для непривілейованих процесів користувача, які не мають прямого доступу до апаратного забезпечення і можуть взаємодіяти з ресурсами лише через виклики до ядра.

Керування пам'яттю

Для реалізації ізоляції процесів в операційних системах застосовуються такі технології, як *віртуальна пам'ять*. Віртуальна пам'ять дозволяє кожному процесу мати власний *віртуальний адресний простір*, який відображається на фізичну пам'ять за допомогою таблиць сторінок (*page tables*). Ця схема захищає пам'ять одного процесу від доступу з боку інших процесів і дозволяє ефективно використовувати доступну фізичну пам'ять.

У *захищеному режимі* (*protected mode*), який підтримується більшістю сучасних процесорів архітектури *x86-64*, операційні системи мають повний контроль над розподілом і керуванням пам'яттю. Процеси виконуються в ізольованих середовищах, і доступ до критично важливих ресурсів (наприклад, пристроїв введення/виведення або системних ресурсів) здійснюється лише через виклики до ядра ОС.

Процеси у захищеному режимі виконуються з різними рівнями привілеїв, що регулюються через *кільця захисту*. Це забезпечує надійну ізоляцію між привілейованими і непривілейованими процесами. Наприклад, якщо користувацький процес потребує доступу до привілейованих ресурсів, він повинен викликати відповідну функцію ядра, яка працює на *кільці 0*, забезпечуючи контрольований доступ.

У контексті віртуалізації, наприклад за допомогою *гіпервізорів* і *контейнерів* (наприклад, Docker), цей механізм ізоляції доповнюється додатковими рівнями розподілу ресурсів на апаратному рівні. Контейнери дозволяють виконувати ізольовані процеси з власними системними ресурсами, такими як файлові системи та мережеві інтерфейси, що забезпечує додатковий рівень безпеки.

Окрім ізоляції, важливою частиною безпеки є правильне налаштування дозволів для користувачів та процесів, щоб уникнути ескалації привілеїв.

Методи забезпечення ізоляції включають:

• *Контейнери* (наприклад, *Docker*) надають ізольоване середовище, що дозволяє кожному контейнеру працювати незалежно від інших. Контейнери також забезпечують сталий стан середовища та готовий набір інструментів для розробника;

• *Віртуалізація застосунків* (наприклад, *VMware ThinApp*, *Turbo Studio* (раніше *Spoon Studio* та *Xenocode Studio*) у *Windows*, *AppImage* у *Linux*). Такі застосунки виконуються у контейнері, але при цьому виглядають як звичайні програми і не потребують встановлення;

• *Списки контролю доступу* (*Access Control Lists, ACL*) використовують для встановлення дозволів на файли та процеси. Це гарантує, що тільки авторизовані користувачі можуть отримати доступ до певних ресурсів.

Попри переваги віртуалізації, ця технологія також постійно зазнає атак, наприклад, у таблиці 8.2 наведено деякі типові вразливості різних гіпервізорів за останні роки.

Архітектура хмарних застосунків

Хмарні застосунки та сервіси можуть бути побудовані на декількох рівнях, що суттєво впливає на можливості такого застосунку, а також на кваліфікацію користувача цього застосунку або сервісу.

На нижньому, фізичному рівні (*physical layer*) використовується реальне апаратне забезпечення, включно із серверами, сховищами даних, мережевою інфраструктурою тощо.

На наступному, інфраструктурному рівні (*infrastructure layer*) надаються базові обчислювальні ресурси, необхідні для функціонування хмарних застосунків, наприклад, кількість процесорів, обсяг пам'яті та місце на диску.

На платформному рівні (*platform layer*) надається готова до розгортання система, що є основою для розробки, наприклад, налаштована операційна система із встановленими бібліотеками та фреймворками для розробників.

І на найвищому рівні – рівні застосунків (*application layer*) – функціонують прикладні програми користувача.

Зазначимо, що архітектура хмарних обчислень є складною і включає не лише сервіси, які ми обговоримо нижче, але й багато технічних деталей реалізації (рис. 8.2, [29]).

Платформи можуть бути класифіковані залежно від моделі доставлення сервісів:

• *IaaS* (*Infrastructure as a Service*, інфраструктура як сервіс) – надає базові обчислювальні ресурси. Прикладами таких платформ є *Amazon Web Services (AWS)* від *Amazon*, *Microsoft Azure* від *Microsoft* і *Google Cloud Platform* від *Google*;

• *PaaS* (*Platform as a Service*, платформа як сервіс) – створює середовище для розробки та управління програмним забезпеченням. Прикладами *PaaS*-платформ є *AWS Elastic Beanstalk* від *Amazon*, *Azure App Service* від *Microsoft* та *Google App Engine* від *Google*;

• *SaaS* (*Software as a Service*, програмне забезпечення як сервіс) – надає готові застосунки, які можна використовувати без необхідності їх встановлення або управління, наприклад, *Microsoft 365* і *Google Workspace*.

Зазначимо, що наданий перелік сервісів не є вичерпним – сьогодні існує

Таблиця 8.2 – Вразливості ПЗ віртуалізації

CVE	**Продукт**	**Опис**
CVE-2024-43575	*Hyper-V*	Уразливість типу *Denial of Service (DoS)* у *Microsoft Hyper-V*, яка може спричинити аварійне завершення системи через спеціально сформовані запити
CVE-2023-21990	*VirtualBox*	Уразливість в *Oracle VirtualBox*, яка дозволяє привілейованому користувачу локально скомпрометувати *VirtualBox*, що може призвести до повного контролю над системою
CVE-2022-39423	*VirtualBox*	Уразливість у *VirtualBox* (версії до 6.1.38), яка дозволяє отримати несанкціонований доступ до даних або повне захоплення системи
CVE-2021-28476	*Hyper-V*	Критична уразливість у *Hyper-V*, яка дозволяє віддаленим зловмисникам виконувати довільний код через компонент *vmswitch*
CVE-2022-23825	*Hyper-V*	Уразливість типу *speculative execution* (подібна до *Spectre*), яка може призвести до витоку інформації у віртуальних машинах *Hyper-V*
CVE-2023-2085	*VirtualBox*	Уразливість у *VirtualBox*, яка дозволяє *втечу з віртуальної машини* (*VM escape*) та доступ до основної системи
CVE-2022-21900	*Hyper-V*	Уразливість у *Hyper-V*, яка дозволяє обійти засоби безпеки та призвести до витоку даних між віртуальними машинами
CVE-2023-21982	*VirtualBox*	Уразливість у *VirtualBox*, яка дозволяє обійти обмеження безпеки та отримати несанкціонований доступ до основної системи з гостьової ОС
CVE-2020-4006	*VMware*	Уразливість віддаленого виконання коду (*remote code execution*) у *VMware ESXi*, яка дозволяє зловмиснику виконувати довільні команди через службу *OpenSLP*
CVE-2022-29900	*Hyper-V*	Уразливість, пов’язана з *speculative execution* (*RETbleed*), яка може призвести до витоку інформації через *атаки на побічні канали* (*side-channel attacks*)

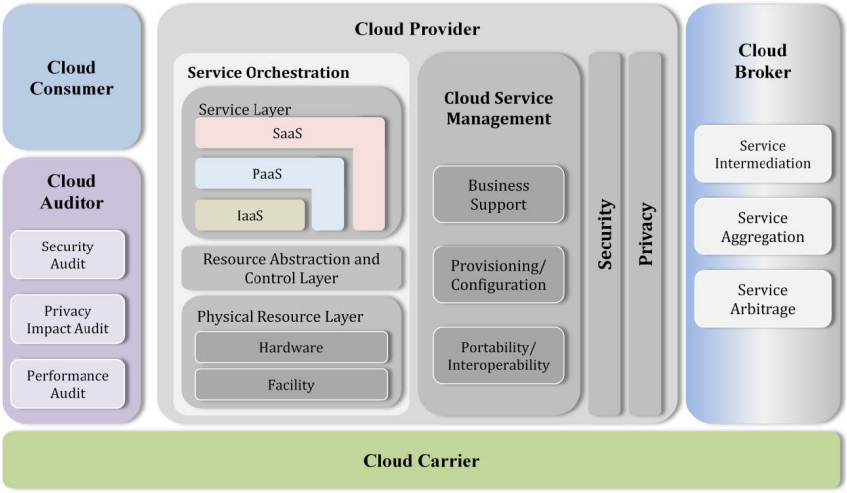


Рисунок 8.2 – Референсна архітектура хмарних обчислень *NIST* [29]

тенденція переносити більшість сервісів у хмари.

Хмарні застосунки *SaaS*: багатошарова архітектура та безпека

Хмарні застосунки *SaaS* зазвичай побудовані за багатошаровою архітектурою, що включає фронтенд (інтерфейс користувача), бекенд (сервери обробки даних), бази даних та мережеві ресурси. Основним елементом захисту в хмарних застосунках є чіткий розподіл ролей і дозволів, а також забезпечення належної автентифікації та шифрування.

Типова архітектура хмарних застосунків *SaaS*:

- *Фронтенд* – це інтерфейс, через який користувач взаємодіє з хмарним застосунком, зазвичай через веббраузер або мобільний застосунок. Безпека фронтенду забезпечується використанням *TLS* (*Transport Layer Security*) для захисту трафіку.
- *Бекенд* – сервери, які обробляють запити користувачів, керують бізнес-логікою та доступом до баз даних. Безпека бекенду полягає в контролі доступу, автентифікації та використанні ізольованих середовищ для виконання коду.
- *Бази даних* – зберігають конфіденційні дані користувачів. Дані повинні бути зашифровані як під час зберігання, так і під час передачі.

Приклад архітектури *AWS*

На платформі *AWS* хмарний застосунок може використовувати *Elastic Load Balancer* для розподілу трафіку між серверами *EC2*, *Amazon RDS* для зберігання даних і *S3* для статичних файлів. Кожен компонент ізольований і має власні політики безпеки, що забезпечує належний рівень захист

Проблеми приватності даних на хмарних сервісах

Однією з основних проблем використання хмарних сервісів є забезпечення приватності та безпеки даних. Дані користувачів можуть бути збережені на віддалених серверах, розташованих у різних частинах світу, що створює питання щодо того, хто має доступ до цих даних та як їх можна захистити (Fig. 8.3[1]).

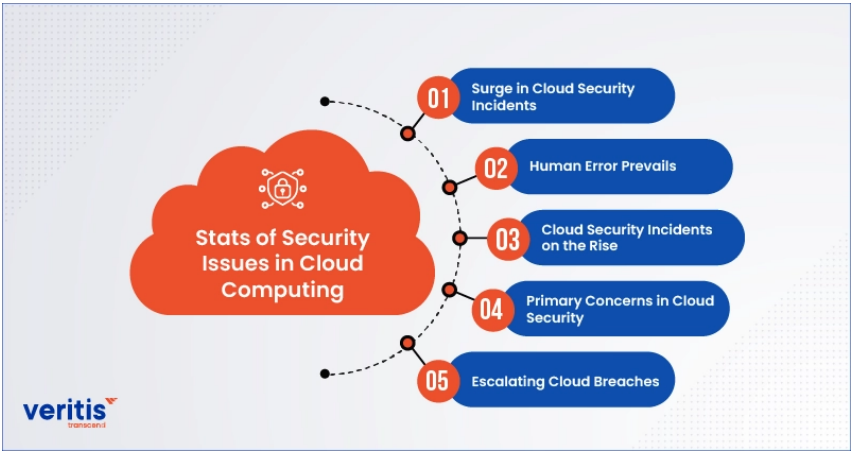


Рисунок 8.3 – Проблеми безпеки даних у хмарних середовищах

Основні проблеми приватності включають:

• контроль над даними у хмарному середовищі покладений на постачальника хмарних послуг, що може створювати ризики несанкціонованого доступу;

• шифрування даних як під час передачі (*TLS*), так і під час зберігання (*AES*) може забезпечити конфіденційність даних;

• різні країни мають специфічні закони щодо обробки та зберігання персональних даних, тому постачальники хмарних послуг повинні дотримуватися таких стандартів, як GDPR.

[1] Veritis: Top 10 Security Issues in Cloud Computing

Приклад шифрування даних на платформі AWS.

На AWS можна використовувати KMS (Key Management Service) для шифрування конфіденційних даних:

```
aws kms encrypt --key-id alias/my-key --plaintext fileb://data.txt
↪  --output text --query CiphertextBlob
```

Цей приклад показує, як зашифрувати файл, використовуючи ключі, збережені в KMS. Це допомагає захистити дані навіть у випадку несанкціонованого доступу до фізичних носіїв.

Контрольні запитання

1. Що таке віртуальні машини і як вони допомагають забезпечити безпеку?
2. Які переваги та недоліки ізоляції процесів у порівнянні з віртуальними машинами?
3. Які основні компоненти типової архітектури хмарного застосунку і як вони взаємодіють?
4. Які проблеми приватності виникають під час використання хмарних сервісів?
5. Як шифрування даних допомагає захистити конфіденційну інформацію в хмарі?
6. Які технології можна використовувати для шифрування даних на хмарних платформах?
7. Які основні методи автентифікації використовуються в хмарних застосунках?
8. Які переваги та ризики використання контейнерів у порівнянні з віртуальними машинами?
9. Як забезпечити відповідність хмарного застосунку регуляторним вимогам (наприклад, *GDPR*)?
10. Як налаштувати ізоляцію та безпеку між віртуальними машинами на хмарній платформі?

Тема 9

ЗАГАЛЬНІ ПИТАННЯ БЕЗПЕКИ ВЕБЗАСТОСУНКІВ

Метою є вивчення типових вразливостей вебзастосунків та способів їх усунення, набуття практичних навичок аналізу вразливостей у сфері веббезпеки.

Завдання для самостійної роботи

1. *Налаштуйте HTTPS* для локального веб-сервера за допомогою *Let's Encrypt*. Наприклад, використайте утиліту *Certbot* для автоматичного отримання та оновлення *TLS*-сертифіката.
2. *Перевірте вебзастосунок на вразливості* згідно з *OWASP Top 10*. Проведіть тестування на такі типові загрози, як *SQL*-ін'єкції (*SQLi*), міжсайтове скриптування (*XSS*), підробка міжсайтових запитів (*CSRF*) тощо. Підготуйте звіт з виявленими вразливостями та рекомендаціями щодо їх усунення.
3. *Реалізуйте політики CORS* (*Cross-Origin Resource Sharing*) для вашого *API*. Налаштуйте відповідні заголовки, зокрема *Access-Control-Allow-Origin*, *Access-Control-Allow-Methods*, *Access-Control-Allow-Headers* тощо.
4. *Проаналізуйте заголовки безпеки* вебсайту. Перевірте наявність та коректність заголовків *Content-Security-Policy*, *X-Frame-Options*, *X-Content-Type-Options* та інших. Запропонуйте покращення відповідно до сучасних практик безпеки.
5. *Напишіть сценарій резервного копіювання бази даних* вебзастосунка. Автоматизуйте процес резервного копіювання за допомогою *Bash*/*Python*-скрипта або відповідного інструмента. Забезпечте щоденне збереження копій із контролем цілісності та можливістю відновлення.

Короткі теоретичні відомості

1. Платформи вебпрограмування та методи забезпечення їх безпеки.
2. Протокол *HTTP* та його безпечне використання.
3. Форми і їх захист від атак.
4. Загрози на стороні клієнта та серверу.
5. Способи уникнення загроз.

Платформи вебпрограмування та методи забезпечення їх безпеки

Вебпрограмування включає використання платформ і фреймворків, таких як *Django* (*Python*), *Spring* (*Java*), *Express* (*Node.js*) та *Laravel* (*PHP*). Незалежно від вибору платформи, ключовими аспектами безпеки є автентифікація користувачів, шифрування даних і захист від поширених атак, таких як *SQL*-ін'єкції та *XSS*. Більшість сучасних фреймворків мають вбудовані механізми безпеки, зокрема автоматичний захист від *CSRF* та підтримку параметризованих *SQL*-запитів для запобігання ін'єкціям.

Основними джерелами інформації про загрози безпеці є *OWASP Top 10* (табл. 2.1), *MITRE Common Weakness Enumeration (CWE)* [30], *MITRE CVE Program* [31] тощо. Вебзастосунки постійно перебувають під атакою, що підтверджують дані таблиці 9.1, яка містить приклади деяких обраних вразливостей.

Процес розробки має враховувати відомі проблеми безпеки і дотримуватися методів безпечної розробки та розгортання протягом усього життєвого циклу програмного забезпечення, наприклад, шляхом застосування *Secure Development Lifecycle (SDLC)*.

Забезпечення безпеки вебзастосунків вимагає використання інструментів для виявлення та усунення вразливостей. Статичні інструменти тестування безпеки (*SAST*) аналізують вихідний код для виявлення проблем на етапі розробки. Динамічні інструменти (*DAST*), такі як *OWASP ZAP*, моделюють реальні атаки для перевірки застосунків в реальному часі. Вебзастосункові міжмережеві екрани (*WAF*), такі як *Cloudflare WAF*, забезпечують захист від мережевих загроз, а інструменти для управління *SSL/TLS*-сертифікатами спрощують шифрування трафіку, забезпечуючи конфіденційність.

Необхідно регулярно оновлювати програмне забезпечення, включаючи сторонні бібліотеки. Оновлення містять *патчі* для усунення відомих вразливостей.

Важливо використовувати складні паролі та двофакторну автентифікацію для захисту облікових записів. Захист сервера повинен включати міжмережеві екрани (*firewalls*), системи виявлення вторгнень (*IDS*) та вебзастосункові міжмережеві екрани (*WAF*).

Для забезпечення актуальності безпеки слід постійно стежити за новими кіберзагрозами та вразливостями, використовуючи відповідні ресурси та інструменти.

Протокол *HTTP* та його безпечне використання

HTTP (*Hypertext Transfer Protocol*) є базовим протоколом для передачі даних у вебзастосунках (Рис. 9.1[1]). Однак, він не забезпечує шифрування даних.

[1] https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-http/

Таблиця 9.1 – Вразливості вебфреймворків

CVE	**Фреймворк**	**Опис**
CVE-2023-37979	*Ninja Forms* (*WordPress*)	Уразливість *XSS* (*Cross-Site Scripting*) у плагіні *Ninja Forms* до версії 3.6.26, що дозволяє виконання *JavaScript*-коду в контексті вебінтерфейсу, що може призвести до компрометації сайту.
CVE-2023-28708	*Apache Tomcat*	Уразливість у *Tomcat* дозволяє витік сесійних *cookie* через незахищений канал, якщо атрибут *secure* не встановлений при використанні заголовка *X-Forwarded-Proto*.
CVE-2023-29450	*Zabbix*	Уразливість у *Zabbix* дозволяє несанкціонованому користувачу отримати доступ до конфіденційної інформації через *JavaScript*-обробку перед початковим виконанням.
CVE-2022-47148	*WooCommerce* (*WordPress*)	Уразливість *CSRF* (*Cross-Site Request Forgery*) у плагіні *WooCommerce PDF Invoice and Packing Slips* до версії 3.2.6, яка дозволяє зловмисникам модифікувати або видаляти конфіденційні дані користувача.
CVE-2023-3247	*PHP*	Уразливість у механізмі *SOAP Digest Authentication* у версіях *PHP* до 8.1.20, яка може призвести до витоку незаповненої пам'яті між клієнтом і сервером.
CVE-2023-3824	*PHP*	Уразливість у *PHP* дозволяє переповнення буфера при завантаженні файлів *PHAR*, що може призвести до виконання довільного коду або пошкодження пам'яті.
CVE-2022-41040	*Microsoft Exchange*	Уразливість *SSRF* (*Server-Side Request Forgery*) у *Microsoft Exchange*, яка дозволяє виконання довільного коду через *PowerShell*, використовуючи серію запитів на сервері.
CVE-2022-41082	*Microsoft Exchange*	Уразливість *Remote Code Execution (RCE)* у *Microsoft Exchange*, яка використовує уразливість *ProxyNotShell* для виконання атак з використанням *PowerShell*.
CVE-2023-3823	*PHP*	Уразливість у *PHP* через проблеми в глобальному стані *libxml*, що може призвести до витоку конфігураційних даних між модулями у процесах.

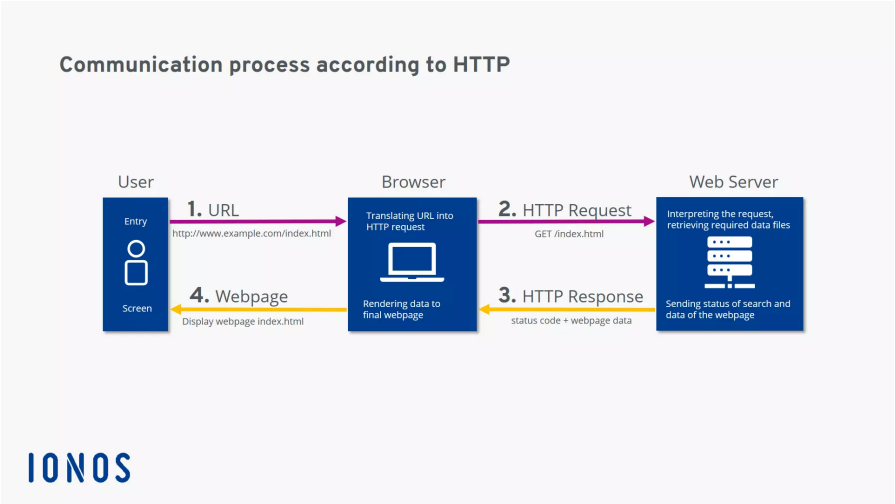


Рисунок 9.1 – *HTTP* протокол

Використання *HTTPS* (*HTTP Secure*) з *SSL/TLS*-шифруванням є обов'язковим для захисту конфіденційної інформації під час передачі, що запобігає атакам типу «людина посередині» (*Man-in-the-Middle*).

HTTP GET та *POST* з точки зору безпеки

HTTP-запити *GET* та *POST* мають важливі відмінності щодо безпеки. Запит *GET* передає дані через *URL*, що робить його небезпечним для чутливої інформації, оскільки *URL* може бути збережено в історії браузера або передано в *referrer'ax*. *GET*-запити також мають обмеження на об'єм переданих даних. Наприклад:

```
GET /search?q=keyword HTTP/1.1
```

Запит *POST* використовує тіло запиту для передачі даних, що робить його більш придатним для конфіденційних даних, таких як паролі або інформація про користувачів, оскільки дані не відображаються в *URL*. Ось приклад *POST*-запиту:

```
POST /login HTTP/1.1
Host: example.com
```

```
Content-Type: application/x-www-form-urlencoded

username=user&password=pass
```

Проблеми безпеки *GET*:

- дані передаються через URL і можуть зберігатися в історії браузера;
- дані можуть ставати доступними через заголовок Referrer або журнали сервера;
- існують обмеження на розмір переданих даних.

Проблеми безпеки *POST*:

- хоча дані не зберігаються в *URL*, вони все ще можуть бути перехоплені, якщо не використовується *HTTPS*;
- можливі атаки *CSRF*, тому слід використовувати токени *CSRF* для захисту.

Зазначимо, що використання *HTTPS* для шифрування запитів є обов'язковим для обох методів. Також важливо впроваджувати перевірку введених даних та захист від атак *CSRF*.

Налаштування *HTTPS* у *Apache*

```
<VirtualHost *:443>
ServerName example.com
SSLEngine on
SSLCertificateFile /etc/ssl/certs/example.com.crt
SSLCertificateKeyFile /etc/ssl/private/example.com.key
</VirtualHost>
```

***Base64* у вебпрограмуванні та кібербезпеці**

Base64-кодування широко використовується у вебпрограмуванні та кібербезпеці для перетворення бінарних даних у текстовий формат. Це забезпечує безпечну передачу даних через текстові протоколи, такі як *HTTP*. *Base64* часто застосовується для кодування зображень, файлів та іншої бінарної інформації у форматах *JSON*, *HTML* та електронній пошті, щоб уникнути пошкодження даних під час передачі.

Нелатинські символи в *URL* та фішинг

Кодування нелатинських символів у *URL* за допомогою *UTF-8* та *percent*-кодування (наприклад, «к» як `%D0%BA`) дозволяє передавати символи через системи, що не підтримують їх напряму. Однак, це також може використовуватися

для фішингових атак. Зловмисники можуть створювати *URL* з нелатинськими символами, що виглядають схожими на легітимні сайти (наприклад, використання символу з кирилиці «а» замість латинської «a»), щоб обманути користувачів та змусити їх перейти на шкідливі ресурси. Для захисту слід перевіряти *URL* перед клацанням на покликання та використовувати антивірусні програми.

Форми і їх захист від атак

Форми є ключовим елементом взаємодії користувача з вебзастосунком, тому їх захист має бути надійним. Основні загрози – *SQL*-ін'єкції, *XSS* та *CSRF*. Для безпеки слід:

- валідовувати всі вхідні дані;
- захищати від *CSRF* за допомогою токенів;
- екранувати введені дані для захисту від *XSS*.

Приклад захисту у *Django*.

Django автоматично забезпечує захист від *CSRF* через вбудовані механізми:

```
from django.views.decorators.csrf import csrf_protect

@csrf_protect
def my_view(request):
return HttpResponse('Захищена сторінка')
```

Приклад захисту від *XSS* у *PHP*

```
<?php
function escape($input) {
            return htmlspecialchars($input, ENT_QUOTES, 'UTF-8');
        }
    ?>
    <form method="post" action="submit.php">
    <input type="text" name="username" value="<?php echo
    ↪  escape($_POST['username']); ?>">
<input type="submit" value="Submit">
</form>
```

Загрози на стороні клієнта та серверу

Основними загрозами на стороні клієнта є *XSS*, фішинг та недовірені скрипти. Для захисту слід використовувати політики безпеки контенту (*CSP*) та уникати динамічного генерування *HTML* без належної валідації.

Основними загрозами на стороні сервера є *SQL*-ін'єкції, атаки на файли та експлуатація вразливостей баз даних. Параметризовані запити та обмеження прав доступу допомагають захистити сервери.

Приклад параметризованого *SQL*-запиту у *PHP*

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE username =
↪  :username');
$stmt->execute(['username' => $username]);
```

Способи уникнення загроз

Для уникнення типових загроз на клієнтській та серверній стороні, слід використовувати такі методи:

- використання *HTTPS* для шифрування переданих даних;
- захист від *CSRF* за допомогою токенів;
- параметризовані *SQL*-запити для захисту від *SQL*-ін'єкцій;
- регулярні оновлення фреймворків та бібліотек для усунення вразливостей;
- використання хешування паролів (*bcrypt*, *Argon2*) та багатофакторної автентифікації.

Контрольні запитання

1. Які основні загрози безпеці виникають на вебплатформах, таких як *Django* чи *Spring*?
2. Як *HTTPS* забезпечує безпеку передачі даних у вебзастосунках?
3. Що таке токени *CSRF* і як вони допомагають захистити вебформи?
4. Як працюють *XSS*-атаки і які методи захисту можна використовувати на стороні клієнта?
5. Які методи можна застосувати для захисту серверних баз даних від *SQL*-ін'єкцій?
6. Як можна захистити вебформи від шкідливих введень користувачів?
7. Які заходи безпеки слід впроваджувати на сервері для захисту від несанкціонованого доступу?

8. Які методи автентифікації можуть допомогти підвищити безпеку вебзастосунків?

9. Як регулярне оновлення фреймворків допомагає забезпечити безпеку вебзастосунків?

10. Як багатофакторна автентифікація допомагає запобігти несанкціонованому доступу до вебзастосунків?

11. Що таке *CSP* і як політика безпеки контенту допомагає боротися з *XSS*?

12. Як можна виявити спроби підміни вебсервера, наприклад через аналіз *SSL*-сертифікатів?

13. Чому важливо використовувати заголовки безпеки, такі як *X-Frame-Options* чи *Content-Security-Policy*?

14. Що таке *Open Redirect* і як уникнути цієї вразливості у вебзастосунках?

15. Які засоби виявлення вразливостей можна використати для тестування вебзастосунків (наприклад, *OWASP ZAP*)?

Тема 10

БЕЗПЕКА ВЕБЗАСТОСУНКІВ – COOKIES, СЕСІЇ ТА АТАКИ

Метою є вивчення механізмів автентифікації, cookies, сесій та пов'язаних із ними атак, набуття практичних навичок виявлення й усунення вразливостей керування сесіями.

Завдання для самостійної роботи

1. *Налаштуйте безпечні cookies* з прапорцями `HttpOnly` і `Secure`, поясніть, у чому їх перевага.
2. *Виявіть вразливі сесії* у вебзастосунку за допомогою Burp Suite (перевірте повторне використання ідентифікаторів сесій тощо).
3. *Реалізуйте механізм CSRF-токенів* у формі входу та перевірте, чи працює захист.
4. *Проведіть атаку session hijacking* у тестовому середовищі (поясніть, як це можливо та як захиститися).
5. *Дослідіть, як JWT (JSON Web Tokens)* покращує безпеку сесій порівняно з традиційними сесіями на базі cookie.

Короткі теоретичні відомості

1. Cookies та сесії: основні поняття та безпека.
2. Атаки *Cross-Site Request Forgery*.
3. Атаки *XML External Entity*.
4. Перенаправлення та їх безпека.

Cookies та сесії: основні поняття та безпека

Cookies – це невеликі текстові файли, що зберігаються у браузері користувача та містять інформацію, яка може використовуватися для автентифікації, збереження налаштувань або відстеження активності. Сесії – це механізм, який дозволяє серверу ідентифікувати користувача між кількома запитами. Cookies часто використовуються для зберігання сесійних ідентифікаторів.

Google планує поступово відмовитися від використання сторонніх (*third-part*) cookie у своєму браузері Chrome в рамках ініціативи *Privacy Sandbox*. Основною метою є захист конфіденційності користувачів і надання їм більшого контролю над тим, як їхні дані використовуються. Відмова від сторонніх

cookie буде відбуватися поетапно, з початком у 2024 році. Хоча сторонні cookie будуть поступово відмовлятися, *first-party cookis* залишаться активними для забезпечення внутрішніх функцій вебсайтів, таких як збереження сесій та налаштувань користувача. Крім того, Google розробляє нові технології, такі як *FLoC* (Federated Learning of Cohorts) та *Topics API*, для підтримки рекламних кампаній без порушення приватності користувачів [32–34].

Основні проблеми безпеки, пов'язані з cookies та сесіями:

• *викрадення сесій* – зловмисники можуть викрасти сесійний ідентифікатор і отримати доступ до облікового запису користувача;

• *cookies без захисту* – незахищені cookies можуть бути передані через незашифроване з'єднання або використовуватися для шкідливих цілей.

Приклад налаштування захищених cookies у PHP.

```
setcookie('session_id', $sessionId, [
        'secure' => true,        // Тільки для HTTPS
        'httponly' => true,      // Недоступний для \emph{Java}Script
        'samesite' => 'Strict'   // Запобігання CSRF
    ]);
```

Цей приклад показує, як налаштувати cookies для забезпечення безпеки: атрибути `secure`, `httponly` та `samesite` допомагають уникнути атак, пов'язаних із cookies.

З погляду кібербезпеки, використання cookie може скомпрометувати безпеку вебзастосунків:

• *CVE-2023-26136*: Вразливість у пакеті `tough-cookie` дозволяє виконання «Prototype Pollution» через неправильне оброблення файлів cookie при використанні режиму rejectPublicSuffixes=false. Це може дозволити зловмисникам маніпулювати об'єктами в системі, що призведе до непередбачуваних наслідків під час виконання коду на сервері;

• *CVE-2023-46218*: Ця вразливість у `curl` дозволяє встановлювати «суперфайли cookie» через змішані великі та малі регістри у домені. Це може призвести до того, що файли cookie будуть передані на інші, не пов'язані між собою сайти та домени, що створює серйозні проблеми безпеки;

• *CWE-539: Use of Persistent Cookies Containing Sensitive Information*: Вразливість, пов'язана зі зберіганням конфіденційної інформації у постійних файлах cookie. Такі файли можуть зберігати критичні дані, такі як сесійні ідентифікатори чи навіть паролі, що може призвести до витоку конфіденційної інформації, якщо файли cookie будуть перехоплені чи зламані

• *CWE-1275: Sensitive Cookie with Improper SameSite Attribute*: Вразливість пов'язана з неправильним налаштуванням атрибута `SameSite` у файлах cookie, що дозволяє здійснювати атаки типу *CSRF*. Якщо атрибут `SameSite` не налаштовано належним чином, файли cookie можуть бути використані для відправки шкідливих запитів від імені користувача.

Атаки *Cross-Site Request Forgery*

CWE-352: Cross-Site Request Forgery (CSRF). Якщо вебсервер спроєктований так, що він приймає запити від клієнта без механізму перевірки того, чи було запит надіслано навмисно, то існує можливість, що зловмисник може обдурити клієнта, змусивши його ненавмисно надіслати запит до вебсервера, який буде оброблено як автентичний. Це може бути зроблено через URL, завантаження зображень, *XMLHttpRequest* тощо, що може призвести до витоку даних або виконання небажаного коду[1].

Таким чином, CSRF – це атака, при якій зловмисник змушує користувача виконати небажані дії на вебсайті, на якому користувач вже автентифікований. Наприклад, зловмисник може створити підроблену форму, яка надсилає запит на зміну пароля користувача без його прямого доступу до облікового запису.

Приклад атаки *CSRF*

```
<form action="http://victim.com/change-password" method="POST">
<input type="hidden" name="password" value="newpassword">
<input type="submit" value="Submit">
</form>
```

Користувач, який вже авторизований на `victim.com`, може випадково натиснути на це посилання, і його пароль буде змінено на `newpassword`, якщо сайт не захищений від CSRF.

Методи захисту від *CSRF*

- використання CSRF-токенів;
- використання атрибута `SameSite` для cookie-файлів;
- перевірка HTTP-заголовків для виявлення підозрілих запитів.

[1] https://cwe.mitre.org/data/definitions/352.html

Приклад захисту від *CSRF* у *PHP*

```
if ($_POST['csrf_token'] !== $_SESSION['csrf_token']) {
        die('CSRF attack detected!');
    }
```

Цей код перевіряє наявність *CSRF*-токена у запиті і порівнює його з токеном у сесії.

Атаки *XML External Entity*

CWE-611: Improper Restriction of XML External Entity Reference. *XML*-документи можуть опціонально містити визначення типу документа (*DTD*), яке, серед іншого, дозволяє визначати *XML*-сутності. Можна визначити сутність, надавши рядок підставляння у вигляді *URI*. *XML*-парсер може отримати доступ до вмісту цього *URI* і вставити цей вміст назад у *XML*-документ для подальшої обробки[2].

Зловмисник може використати *XML*-файл із зовнішньою сутністю, яка посилається на *URI* формату `file://`, щоб змусити застосунок отримати вміст локального файлу. Наприклад, *URI* `file:///c:/winnt/win.ini` може вказувати на файл конфігурацій у *Windows*, або `file:///etc/passwd` – на файл паролів у *Unix*. Також можливо використати *URI* інших протоколів, таких як `http://`, для відправлення запитів до віддалених серверів, що дозволяє обходити обмеження або приховувати джерело атак. Вміст таких файлів може бути випадково відображений у застосунку, розкриваючи конфіденційні дані.[3].

Таким чином, атака *XXE* (*XML External Entity*) виникає, коли *XML*-процесор обробляє небезпечний зовнішній ресурс, визначений у *XML*-документі. Зловмисник може скористатися цією уразливістю для отримання доступу до конфіденційних даних або запуску шкідливих запитів.

Приклад уразливого *XML*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
        <!ELEMENT foo ANY >
        <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
```

[2]https://cwe.mitre.org/data/definitions/611.html
[3]https://cwe.mitre.org/data/definitions/611.html

```
<foo>&xxe;</foo>
```

У цьому прикладі XML-документ намагається отримати доступ до файлу `/etc/passwd`, що може призвести до витоку інформації.

Захист від атак типу *XXE*

Вимкнення зовнішніх ресурсів у *XML*-парсерах та використання безпечних бібліотек для обробки *XML* є основними методами захисту від таких уразливостей.

Приклад захисту від *XXE* у *Java*

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
↪  true);
```

Цей код вимикає обробку зовнішніх сутностей у *XML*-документах, захищаючи систему від *XXE*-атак.

Перенаправлення та їх безпека

Перенаправлення є загальною функцією вебзастосунків, яка може бути використана для скерування користувача на інші сторінки або сайти. Проте, якщо перенаправлення не контролюються належним чином, зловмисники можуть використовувати їх для фішингових атак або перенаправлення користувачів на шкідливі сайти.

CWE-601: URL Redirection to Untrusted Site ('Open Redirect') стосується вразливості, при якій вебзастосунок приймає введений користувачем *URL* і використовує його для перенаправлення без належної перевірки. Це може призвести до фішингових атак, коли зловмисник підміняє *URL*, змушуючи користувачів перейти на шкідливий сайт, який виглядає легітимно, але краде їхні дані або запускає шкідливий код.[4]

Приклад уразливого перенаправлення

```
header('Location: ' . $_GET['url']);
```

[4] https://cwe.mitre.org/data/definitions/601.html

Цей код перенаправляє користувача на будь-який *URL*, переданий у запиті, що може бути використано для атаки.

Захист від небезпечних перенаправлень

- перевіряти URL-адреси перед перенаправленням;
- використовувати дозволений список (whitelist) безпечних URL.

Приклад безпечного перенаправлення

```
$allowed_urls = ['home.php', 'profile.php'];
    if (in_array($_GET['url'], $allowed_urls)) {
    header('Location: ' . $_GET['url']);
}
```

У цьому прикладі перенаправлення виконується тільки на безпечні *URL* зі списку дозволених.

Контрольні запитання

1. Що таке *cookies* і як їх можна налаштувати для безпеки вебзастосунка?
2. Які проблеми безпеки виникають під час використання сесій?
3. Що таке *CSRF*-атака і як можна захистити вебзастосунок від неї?
4. Що таке *XXE*-атака і як запобігти її виникненню?
5. Як небезпечні перенаправлення можуть використовуватися у фішингових атаках?
6. Як можна захиститися від викрадення сесій?
7. Які атрибути *cookies* можуть допомогти запобігти атакам, пов’язаним з безпекою?
8. Які методи можна використовувати для перевірки безпеки перенаправлень у вебзастосунках?
9. Як можна забезпечити правильне використання *CSRF*-токенів у вебзастосунках?
10. Які *XML*-бібліотеки можна використовувати для запобігання *XXE*-атакам?

Тема 11

БЕЗПЕКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ МОБІЛЬНИХ ПРИСТРОЇВ

Метою є вивчення загроз і особливостей захисту мобільних застосунків, набуття практичних навичок аналізу мобільних застосунків з огляду на принципи безпечного програмування.

Завдання для самостійної роботи

1. *Проаналізуйте дозволи Android-застосунку* за допомогою *adb* (перевірте, які дозволи вимагає застосунок і навіщо).
2. *Напишіть код для безпечного зберігання паролів у iOS-застосунку*, використовуючи *Keychain*.
3. *Виявіть вразливості в мобільному застосунку* за допомогою *MobSF* (*Mobile Security Framework*) та надайте рекомендації щодо їх усунення.
4. *Реалізуйте біометричну аутентифікацію* (*Face ID* або *Touch ID*) у тестовому мобільному застосунку.
5. *Порівняйте безпеку нативних застосунків та Progressive Web Apps (PWA)*, зосередившись на зберіганні даних та доступі до апаратних ресурсів.

Короткі теоретичні відомості

1. Платформи мобільних операційних систем та архітектура мобільного застосунку.
2. Безпечне програмування для *OS Android*.
3. Безпечне програмування для *iOS*.
4. Захист мережевих операцій.

Платформи мобільних операційних систем та архітектура мобільного застосунку

Мобільні пристрої працюють на різних операційних системах (Рис. 11.1), серед яких найбільш поширеними є *Android* та *iOS*. Ці операційні системи мають різні механізми захисту, такі як контроль дозволів, ізоляція застосунків і шифрування даних. *Мобільна платформа* – це загальна назва технології, що дозволяє створювати застосунки, які працюють на мобільних пристроях під управлінням операційних систем *Android* та *iOS*.[1]

[1] One Service (oneservice.in.ua)

ОС	%
Android	70.95
iOS	28.36
Samsung	0.37
Unknown	0.14
KaiOS	0.12
Windows	0.02
Linux	0.01
Other	0.03

Рисунок 11.1 – Використання мобільних ОС за даними StatCounter.

Типова архітектура мобільного застосунку включає кілька шарів:

• *інтерфейс користувача (*UI*)* забезпечує зручну та інтуїтивну взаємодію користувача із застосунком через віджети, анімації та інші візуальні елементи;

• *бізнес-логіка* відповідає за функціональність застосунку та забезпечує керування основними процесами для розв'язання прикладних задач;

• *доступ до даних* забезпечує збереження та отримання інформації через локальні бази даних або взаємодіє з віддаленими серверами, використовуючи *REST*, *GraphQL API* тощо;

• *аутентифікація, безпека та мережеві сервіси* забезпечують безпечну комунікацію через шифрування *SSL/TLS*, а також реалізують механізми аутентифікації, такі як *OAuth* або *JWT*.

Існують щонайменше чотири типові архітектури для мобільних застосунків[2], а вибір архітектури суттєво залежить від призначення застосунку, досвіду розробника, вибраних технологій тощо:

• *MVC (Model-View-Controller)*: Це класична модель, де *Model* відповідає за дані, *View* – за інтерфейс, а *Controller* керує логікою взаємодії між ними. Використовується для чіткого розподілу логіки, інтерфейсу та даних, що полегшує розробку та підтримку коду;

• *MVP (Model-View-Presenter)*: У цій архітектурі *Presenter* працює як посередник між *Model* та *View*. *View* лише відображає дані, тоді як *Presenter* відповідає за всю логіку взаємодії. Це дозволяє тестувати бізнес-логіку незалежно

[2]IT-компанія WEZOM (wezom.com.ua)

від інтерфейсу;

• *MVVM (Model-View-ViewModel)*: Тут *ViewModel* відокремлює бізнес-логіку від інтерфейсу (*View*), взаємодіючи з *Model*. Ця архітектура підходить для застосунків із великою кількістю даних і складною логікою відображення;

• *Clean Architecture*: Базується на принципах відокремлення бізнес-логіки, процесів і технологій. Мета – створити незалежну архітектуру, що не залежить від інтерфейсів, баз даних або зовнішніх сервісів, що робить її гнучкою та легкою для тестування.

Основні питання безпеки мобільних застосунків включають контроль доступу до ресурсів пристрою, захист локально збережених даних та безпечне виконання мережевих операцій.

Безпечне програмування для ОС *Android*

Android є операційною системою з відкритим кодом, яка використовує моделі дозволів для захисту доступу до критичних ресурсів пристрою (камери, мікрофона, контактів тощо). Застосунки на *Android* працюють у ізольованих середовищах, що запобігає несанкціонованому доступу до даних одного застосунку з боку іншого.

Android-застосунки розробляють кількома мовами, таких як *Java*, *Kotlin*, *C* і *C++*, а також *HTML5* (для мобільних вебзастосунків). Застосунки *Android* можуть використовувати нативний код, написаний на *C* і *C++*, для реалізації певних функціональних можливостей, наприклад, при роботі з ядром *Linux*. Слід зазначити, що взаємодія з такими застосунками через *Java Native Interface (JNI)* потенційно піддається вразливостям, таким як переповнення буфера та інші проблеми, які можуть бути присутні в реалізації нативного коду та характерні для *C* і *C++*.

SEI Android Secure Coding Standard містить правила для безпечного кодування застосунків на платформі *Android*, основна мета яких полягає в створенні безпечних, надійних і стійких до атак програмних систем. Основний акцент робиться на усунення невизначеної поведінки, яка може призводити до вразливостей у програмному забезпеченні. Відповідність цим правилам необхідна для безпеки систем, але недостатня – також важливий безпечний дизайн системи. Для критичних систем, де безпека є першорядною, передбачені ще суворіші вимоги, наприклад, статичне виділення пам'яті. Стандарт *CERT* для *Android* частково інтегрує правила з інших мов програмування, таких як *C* і *Java*, але з урахуванням специфіки платформи *Android* [35].

У цьому стандарті розрізняють правила та рекомендації в чотирьох категоріях:

• правила, специфічні лише для *Android*;

- правила та рекомендації для *C* і *C++* (*NDK*);
- найкращі практики (рекомендації) безпечного програмування на *Java*;
- правила для *Java*.

Приклад верифікації сертифікатів *SSL/TLS* у *Android*-застосунках

Rule 04. Network – SSL/TLS (NET). DRD19. Properly verify server certificate on SSL/TLS.

Android-застосунки, які використовують протоколи *SSL/TLS* для безпечної комунікації, повинні правильно верифікувати серверні сертифікати. Основна перевірка включає:

- перевірку, що суб'єкт (*CN*) сертифіката *X.509* відповідає *URL*;
- перевірку, що сертифікат підписаний довіреним центром сертифікації (*CA*);
- перевірку правильності підпису;
- перевірку, що термін дії сертифіката не закінчився.

Існують також стандарти для розробників *Android* від *Japan Smartphone Security Association (JSSEC)* [36; 37].

Android Application Secure Design/Secure Coding Guidebook, як і розглянута вище *SEI Android Secure Coding Standard*, містить правила, рекомендації та найкращі практики для безпечного дизайну та кодування, спеціально розроблену для розробників застосунків на платформі *Android*.

З огляду на стрімке зростання ринку смартфонів та відкритість мобільних платформ, таких як *Android*, розробники мають доступ до функцій, які раніше були обмеженими. Однак така відкритість приносить із собою підвищену відповідальність для розробників, оскільки неправильно розроблені або закодовані застосунки можуть призвести до витоку особистої інформації або створити вразливості в системі безпеки. З огляду на відкритий характер Android, розробники повинні бути особливо пильними щодо питань безпеки, оскільки існує менше обмежень на випуск застосунків порівняно з iOS.

Android Application Secure Design/Secure Coding Guidebook містить велику кількість прикладів коректного дизайну саме застосунків *Android* з докладними прикладами. Уся інформація розбита на чотири великих групи:

- базові знання з безпечного проєктування та безпечного програмування;
- безпечне використання технологій;
- як користуватися функціями безпеки;
- складні проблеми.

Контроль дозволів в *Android*

Починаючи з *Android* 6.0 (*API level 23*), користувачі можуть дозволяти або забороняти доступ до певних функцій застосунку під час його використання. Це дозволяє користувачам краще контролювати, які дані доступні застосунку.

Приклад запиту дозволу на доступ до камери.

```
if (ContextCompat.checkSelfPermission(this,
↪  Manifest.permission.CAMERA)
!= PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
        new String[] { Manifest.permission.CAMERA }, REQUEST_CAMERA);
    }
```

Цей код перевіряє, чи має застосунок дозвіл на доступ до камери, і, якщо ні, запитує його у користувача.

Шифрування локальних даних в *Android*

Для захисту конфіденційних даних, що зберігаються на пристрої, застосунки можуть використовувати *Android Keystore* для безпечного збереження ключів шифрування.

```
KeyGenerator keyGenerator =
↪  KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
↪  "AndroidKeyStore");
keyGenerator.init(new KeyGenParameterSpec.Builder("myKeyAlias",
KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
.setBlockModes(KeyProperties.BLOCK_MODE_GCM)
.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
.build());
SecretKey key = keyGenerator.generateKey();
```

Цей приклад показує, як створити ключ у Keystore для шифрування та розшифрування даних.

Приклад використання *HTTPS* у *Android* з `HttpURLConnection`

```
URL url = new URL("https://example.com/api");
HttpsURLConnection connection = (HttpsURLConnection)
↪  url.openConnection();
connection.setRequestMethod("GET");
InputStream inputStream = connection.getInputStream();
// Обробка відповіді
```

Безпечне програмування для *iOS*

iOS – це операційна система закритого типу з високим рівнем контролю доступу до системних ресурсів. *Apple* надає розробникам суворі вказівки щодо захисту даних користувачів і використовує моделі ізоляції застосунків для забезпечення безпеки.

Контроль дозволів в iOS. Як і *Android*, *iOS* вимагає від застосунків запитувати дозволи на доступ до конфіденційних ресурсів, таких як камера або геолокація. Запити дозволів виконуються через файл конфігурації *Info.plist*, де вказується пояснення для користувача, навіщо застосунку потрібен доступ.

Приклад конфігурації дозволу на доступ до геолокації.

```
<key>NSLocationWhenInUseUsageDescription</key>
<string>We need access to your location for navigation
↪  purposes.</string>
```

Шифрування даних в *iOS*

Apple забезпечує шифрування даних, що зберігаються у застосунках, за допомогою механізму *Data Protection*. Крім того, розробники можуть використовувати *Keychain* для збереження конфіденційної інформації, такої як паролі.

Приклад збереження даних у *Keychain*

```
let keychain = Keychain(service: "com.example.myapp")
keychain["password"] = "supersecretpassword"
```

Цей код зберігає пароль у *Keychain*, який використовує шифрування для захисту конфіденційної інформації.

Захист мережевих операцій

Обидві платформи, *Android* та *iOS*, підтримують використання *HTTPS* для забезпечення безпеки передачі даних через мережу. Використання *HTTPS* запобігає перехопленню даних зловмисниками під час їх передачі між застосунком і сервером.

Приклад використання *HTTPS* у *iOS* з *URLSession*

```
let url = URL(string: "https://example.com/api")!
let task = URLSession.shared.dataTask(with: url) { data, response,
↪  error in
        // Обробка відповіді
    }
task.resume()
```

Контрольні запитання

1. Яка основна архітектура мобільного застосунку і які її основні рівні?
2. Які методи контролю дозволів використовуються в *Android* та *iOS*?
3. Як можна забезпечити захист даних, що зберігаються локально на пристрої *Android*?
4. Що таке *Android Keystore* і як він допомагає у забезпеченні безпеки?
5. Як вs *iOS* можна використовувати *Keychain* для безпечного збереження конфіденційної інформації?
6. Як можна забезпечити безпеку мережевих операцій у мобільних застосунках?
7. Які переваги і недоліки надає ізоляція застосунків у *Android* та *iOS*?
8. Як захистити дані під час передачі між мобільним застосунком і сервером?
9. Як працює система дозволів в *Android* 6.0 і вище?
10. Які основні механізми захисту мобільних платформ *Android* та *iOS* від несанкціонованого доступу?

Тема 12

ЖИТТЄВИЙ ЦИКЛ БЕЗПЕЧНОЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

Метою є вивчення концепції *Secure Software Development Lifecycle (SSDLC)* та її етапів, набуття практичних навичок застосування методології *SSDLC*.

Завдання для самостійної роботи

1. *Створіть план впровадження* SSDLC у вигаданій компанії (визначте етапи, відповідальних, інструменти перевірки безпеки).
2. *Проведіть* Threat Modeling для нового функціоналу застосунку (використайте підходи *STRIDE* або *DREAD*).
3. *Напишіть політику безпеки рецензування коду* для команди розробників (хто перевіряє, що перевіряється, які інструменти потрібні).
4. *Проаналізуйте етапи* SSDLC *у контексті* Agile-*методології*, розглянувши, як часто слід виконувати перевірки безпеки.
5. *Розробіть план для тестування безпеки* на кожному етапі *SSDLC* (планування, дизайн, розробка, тестування, реліз).

Короткі теоретичні відомості

1. Огляд структури та компонентів *SSDLC*.

Огляд *SSDLC*

SSDLC – це інтеграція практик безпеки на кожному етапі розробки програмного забезпечення. *SSDLC* допомагає ідентифікувати, уникнути та виправити вразливості на ранніх стадіях розробки, що дозволяє зменшити ризики, пов'язані з безпекою (Рис. 12.1[1]).

Основними етапами *SSDLC* є

- визначення вимог (*Requirements*);
- проєктування (*Design*);
- розробка (*Development*);
- тестування (*Testing*);
- розгортання (*Deployment*);

[1] Checkmarx: Secure SDLC

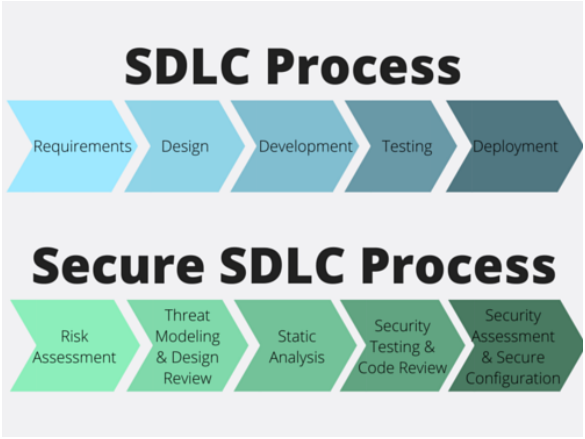


Рисунок 12.1 – *SSDLC*

- підтримка та моніторинг (*Maintenance and Monitoring*).

Визначення вимог (*Requirements*)

На першому етапі визначаються вимоги до програмного забезпечення, включаючи вимоги до безпеки. Важливо ідентифікувати потенційні загрози та ризики. Фахівці з безпеки повинні брати участь у процесі визначення вимог. Основні дії:

- ідентифікація загроз та вразливостей;
- визначення вимог безпеки для функціональності та даних.

Проєктування (*Design*)

Цей етап включає проєктування архітектури програмного забезпечення з урахуванням вимог до безпеки. Проводиться аналіз загроз та визначаються захисні механізми:

- проведення аналізу загроз (*Threat Modeling*);
- проєктування засобів безпеки: шифрування, автентифікація, контроль доступу.

Розробка (*Development*)

Під час розробки застосовуються принципи безпечного програмування, такі як контроль введених даних, управління дозволами, використання перевірених

бібліотек:

- використання безпечних шаблонів кодування для запобігання ін'єкціям;
- статичний аналіз коду для виявлення вразливостей.

Тестування (*Testing*)

На цьому етапі проводяться різні види тестування для виявлення вразливостей, включаючи тестування на проникнення та автоматизований аналіз:

- тестування на проникнення (*Penetration Testing*) для імітації реальних атак;
- використання інструментів динамічного аналізу для виявлення вразливостей під час виконання.

Розгортання (*Deployment*)

Розгортання програмного забезпечення має включати захист середовища виконання, налаштування шифрування та контроль доступу:

- використання *DevSecOps* для автоматизації безпечного розгортання;
- налаштування брандмауерів, *VPN* та інших мережевих засобів захисту.

Підтримка та моніторинг (*Maintenance and Monitoring*)

Після розгортання забезпечується постійний моніторинг на предмет нових вразливостей. Проводиться оновлення та виправлення для підтримки безпеки:

- регулярне оновлення та виправлення вразливостей програмного забезпечення;
- використання інструментів для моніторингу загроз (*IDS/IPS*).

Інструменти та методи *SSDLC*

- *рецензування коду (Code Review)* – ручний перегляд коду на предмет можливих вразливостей;
- *статичний аналіз коду (Static Code Analysis)* – автоматизований аналіз коду для виявлення вразливостей;
- *тестування на проникнення (Penetration Testing)* – імітація атак на програму для виявлення вразливих місць;
- *виправлення помилок та оновлення (Patching and Updates)* – регулярне оновлення для закриття вразливостей.

CVE* та *CWE* у контексті *SSDLC

CVE (Common Vulnerabilities and Exposures) – це система унікальних ідентифікаторів відомих вразливостей, яка допомагає виявляти та виправляти вра-

зливості у програмному забезпеченні.

CWE (Common Weakness Enumeration) – це класифікація типів вразливостей, яка допомагає розробникам уникати поширених помилок під час розробки програмного забезпечення.

Принципи *SSDLC*

Основні принципи *SSDLC* включають:

• *безпеку за замовчуванням (Security by Default)* – програмне забезпечення повинно бути захищене без додаткових налаштувань;

• *принцип найменших привілеїв (Least Privilege)* – кожен користувач або процес повинен мати мінімальні необхідні привілеї;

• *реагування на інциденти (Incident Response)* – необхідність мати план дій у разі виявлення загрози;

• *постійне навчання та вдосконалення (Continuous Improvement)* – загрози безпеці постійно еволюціонують, тому процеси SSDLC мають постійно вдосконалюватися.

Контрольні запитання

1. Які етапи включає *SSDLC* і чому вони важливі?
2. Яка роль статичного аналізу коду у процесі *SSDLC*?
3. Що таке *CVE* і як воно впливає на підтримку безпеки програмного забезпечення?
4. Як можна інтегрувати безпеку на етапі розгортання програмного забезпечення?
5. Що таке *Penetration Testing* і як його результати допомагають покращити безпеку?
6. Які основні принципи безпеки в контексті *SSDLC*?
7. Як *Threat Modeling* допомагає у проєктуванні безпечного програмного забезпечення?
8. Які методи моніторингу безпеки можна застосувати на етапі підтримки системи?
9. Чим відрізняються *CVE* та *CWE* і як вони застосовуються у *SSDLC*?
10. Як автоматизовані інструменти *DevSecOps* можуть підвищити рівень безпеки у процесі розробки?

Тема 13

ПРОГРАМУВАННЯ БЕЗПЕКИ НА ВИСОКОМУ РІВНІ

Метою є вивчення методів інтеграції безпеки на рівні архітектури програмного забезпечення, набуття навичок роботи з протоколами шифрування.

Завдання для самостійної роботи

1. *Напишіть модуль шифрування даних з використанням AES-256* (наприклад, на *Python* з бібліотекою *PyCryptodome*).
2. *Дослідіть реалізацію протоколу OAuth 2.0* у вебзастосунку (як працює потік авторизації, які токени видаються).
3. *Реалізуйте систему ролей та прав доступу (Role-Based Access Control(RBAC))* у застосунку (визначте ролі, дозволи та механізми перевірки).
4. *Порівняйте алгоритми хешування паролів*: *bcrypt*, *scrypt*, *Argon2* (продуктивність, безпека, налаштування).
5. *Створіть схему багатофакторної аутентифікації* для *API* (наприклад, паролі + *OTP* або апаратні ключі).

Короткі теоретичні відомості

1. Методи забезпечення безпеки на високому рівні.
2. Хешування та криптографія за допомогою криптографічних бібліотек.
3. Автентифікація через *GSSAPI*.
4. Використання цифрових підписів.

Методи забезпечення безпеки на високому рівні

Безпека на високому рівні в корпоративних системах передбачає використання складних засобів захисту, які забезпечують цілісність, конфіденційність і автентичність даних. Це включає використання хешування, шифрування, цифрових підписів, а також протоколів для автентифікації та авторизації.

Основні методи забезпечення безпеки включають:

• *хешування* – односторонній процес перетворення даних у фіксовану довжину, який використовується для перевірки цілісності даних;

• *шифрування* – процес перетворення даних у зашифрований вигляд для забезпечення конфіденційності;

• *цифрові підписи* – використання криптографічних алгоритмів для забезпечення автентичності даних та їх цілісності;

• *автентифікацію* – перевірку автентичності користувача або сервісу, зокрема за допомогою таких протоколів, як *GSSAPI*.

Хешування та криптографія за допомогою криптографічних бібліотек

Хешування є критично важливим для забезпечення цілісності даних. Одним із найбільш поширених алгоритмів хешування є *SHA-256*, який забезпечує високий рівень безпеки. Криптографічні бібліотеки, такі як *OpenSSL*, надають інструменти для використання різних алгоритмів хешування та шифрування.

Приклад хешування з використанням *OpenSSL* (*SHA-256*).

```
#include <openssl/evp.h>
#include <string.h>

unsigned char hash[EVP_MAX_MD_SIZE];
unsigned int hash_len;

EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
EVP_DigestInit_ex(mdctx, EVP_sha256(), NULL);
EVP_DigestUpdate(mdctx, data, strlen(data));
EVP_DigestFinal_ex(mdctx, hash, &hash_len);
EVP_MD_CTX_free(mdctx);
```

Цей код показує, як за допомогою *OpenSSL* можна створити хеш для масиву даних з використанням алгоритму *SHA-256*.

Шифрування є ще одним важливим інструментом для забезпечення конфіденційності даних. Криптографічні алгоритми, такі як AES (Advanced Encryption Standard), широко використовуються для захисту даних у корпоративних системах.

Приклад шифрування з використанням *AES*.

```
EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len);
EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
```

```
    EVP_CIPHER_CTX_free(ctx);
```

Цей приклад використовує *AES-256* для шифрування даних у режимі *CBC*, забезпечуючи високий рівень захисту.

Автентифікація через *GSSAPI*

GSSAPI (*Generic Security Services Application Programming Interface*) є стандартом для автентифікації та встановлення захищених сеансів між користувачами та сервісами. Це *API* широко використовується у корпоративних мережах для підтримки безпечної автентифікації за допомогою таких протоколів, як *Kerberos*.

GSSAPI дозволяє програмам використовувати різні механізми безпеки для автентифікації без необхідності детального налаштування криптографічних параметрів. Наприклад, за допомогою *GSSAPI* можна реалізувати автентифікацію користувачів через *Kerberos*, що дозволяє надійно ідентифікувати користувачів у корпоративних мережах.

Приклад автентифікації через *GSSAPI* (*Kerberos*).

```
    OM_uint32 major_status, minor_status;
    gss_name_t server_name;
    gss_buffer_desc name_buf;
    gss_ctx_id_t context = GSS_C_NO_CONTEXT;

    name_buf.value = "service@hostname";
    name_buf.length = strlen((char *)name_buf.value);
    major_status = gss_import_name(&minor_status, &name_buf,
    GSS_C_NT_HOSTBASED_SERVICE, &server_name);

    major_status = gss_init_sec_context(&minor_status,
    ↪  GSS_C_NO_CREDENTIAL,
    &context, server_name, GSS_C_NO_OID,
    GSS_C_MUTUAL_FLAG | GSS_C_REPLAY_FLAG,
    0, NULL, GSS_C_NO_BUFFER, NULL, NULL, NULL);
```

Цей приклад демонструє процес ініціалізації контексту автентифікації за допомогою *GSSAPI* для встановлення захищеного сеансу.

Використання цифрових підписів

Цифрові підписи забезпечують автентичність і цілісність даних. Вони використовують асиметричне шифрування для перевірки джерела даних і гарантують, що дані не були змінені під час передачі. У корпоративних системах цифрові підписи широко використовуються для підпису електронних документів або захисту програмного забезпечення.

Приклад створення цифрового підпису з використанням *OpenSSL*.

```
EVP_PKEY *private_key = load_private_key();
EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
EVP_SignInit(mdctx, EVP_sha256());
EVP_SignUpdate(mdctx, message, message_len);
EVP_SignFinal(mdctx, signature, &sig_len, private_key);
EVP_MD_CTX_free(mdctx);
```

Цей код показує, як можна створити цифровий підпис для повідомлення за допомогою алгоритму *SHA-256* і приватного ключа.

Контрольні запитання

1. Що таке хешування і яке його основне призначення у системах безпеки?
2. Які криптографічні алгоритми використовуються для шифрування даних у корпоративних системах?
3. Як працює *GSSAPI* і яке його значення у процесі автентифікації?
4. Яку роль відіграють цифрові підписи в забезпеченні безпеки даних?
5. Як використовувати криптографічні бібліотеки для реалізації шифрування і хешування?
6. Які ризики можуть виникнути при неправильному використанні шифрування?
7. Що таке *AES* і чому цей алгоритм широко використовується для захисту даних?
8. Як можна використовувати *GSSAPI* для захисту корпоративних мереж?
9. Які типи цифрових підписів використовуються в корпоративних системах і як вони допомагають гарантувати цілісність даних?
10. Як *OpenSSL* допомагає реалізувати криптографічні рішення в програмному забезпеченні?

Тема 14

АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ БЕЗ ДОСТУПУ ДО ВИХІДНОГО КОДУ ПРОГРАМИ

Метою є ознайомлення з принципами реверс-інжинірингу та його застосуванням у сфері безпеки, набуття навичок роботи з інструментами зворотної розробки.

Завдання для самостійної роботи

1. *Проаналізуйте бінарний файл* за допомогою *IDA Pro* або *Ghidra* (знайдіть точки входу, основні функції).
2. *Виявіть функції шифрування* у зворотно розробленому коді та спробуйте зрозуміти їх алгоритм.
3. *Напишіть код (patch)* для модифікації поведінки програми (наприклад, змініть перевірку ключа активації).
4. *Дослідіть техніки обфускації коду* (*пакувальники*, *шифрування рядків*, *віртуалізація коду*) для захисту від *зворотної розробки*.
5. *Проведіть аналіз шкідливого ПЗ* з використанням *Debugger* (*OllyDbg*, *x64dbg*), зосереджуючись на ключових точках виконання.

Короткі теоретичні відомості

1. Визначення та важливість зворотної розробки.
2. Основні методи аналізу програмного забезпечення без вихідного коду.
3. Інструменти для зворотної розробки.
4. Етичні та правові аспекти зворотної розробки.

Методи зворотної розробки

Зворотна розробка (*reverse engineering*) – це процес аналізу програмного забезпечення для виявлення його структури та функціональності без доступу до вихідного коду. Цей метод використовується для аналізу підозрілого програмного забезпечення, виявлення вразливостей та виправлення помилок у скомпільованих програмах.

Основні методи зворотної розробки включають:

• *декомпіляція* – перетворення виконуваного коду назад у код високого рівня, що дозволяє дослідникам зрозуміти логіку програми;

• *дизасемблювання* – перетворення виконуваного коду у низькорівневий асемблерний код, який дозволяє аналізувати роботу програми на машинному рівні;
• *аналіз бінарних файлів* – дослідження скомпільованих бінарних файлів для виявлення прихованих функцій або вразливостей.

Приклад використання декомпілятора *Java*.

```
javap -c MyClass.class
```

Цей приклад демонструє використання інструменту `javap` для декомпіляції байт-коду *Java*, що дозволяє досліджувати методи класу.

Пакування та його аналіз

Пакування (*packing*) – це процес, який застосовується для стискання або шифрування виконуваних файлів з метою приховування їх справжнього коду. Паковані програми часто використовуються для приховування шкідливого програмного забезпечення або для захисту коду від зворотної розробки. Розпакування таких файлів є ключовим кроком для проведення аналізу.

Приклад інструменту для аналізу упакованих файлів (*UPX*).

```
upx -d packed_binary
```

Цей приклад показує використання *UPX* для розпакування виконуваного файлу, що дозволяє аналізувати його оригінальний код.

Декомпіляція

Декомпіляція – це процес перетворення виконуваного коду у код високого рівня. Цей метод допомагає відновити логіку програми та досліджувати її поведінку. Для різних мов програмування існують декомпілятори, що дозволяють аналізувати програми без доступу до вихідного коду.

Приклад використання декомпілятора для *.NET*.

```
ildasm MyProgram.exe
```

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.8.3928.0
//  Copyright (c) Microsoft Corporation.  All rights reserved.
// Metadata version: v4.0.30319
.assembly extern System.Runtime
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  ↪  // .?_....:
  .ver 9:0:0:0
}
.assembly extern System.Console
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  ↪  // .?_....:
  .ver 9:0:0:0
}
.assembly HelloIldasm
{
  ...
}
.module HelloIldasm.dll
// MVID: {8FFEB72D-1EED-43DF-A9A2-FB2113C5FD7D}
.custom instance void [System.Runtime]System.Runtime.CompilerServices.R⌋
↪  efSafetyRulesAttribute::.ctor(int32) = ( 01 00 0B 00 00 00 00 00 )
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00000197D5C60000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit HelloIldasm.Program
       extends [System.Runtime]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
```

```
      .entrypoint
      .custom instance void [System.Runtime]System.Runtime.CompilerServic
      ↪  es.NullableContextAttribute::.ctor(uint8) = ( 01 00 01 00 00 )
      // Code size       11 (0xb)
      .maxstack  8
      IL_0000:  ldstr      "Hello from ILDASM!"
      IL_0005:  call       void
      ↪  [System.Console]System.Console::WriteLine(string)
      IL_000a:  ret
    } // end of method Program::Main

    .method public hidebysig specialname rtspecialname
            instance void  .ctor() cil managed
    {
      // Code size       7 (0x7)
      .maxstack  8
      IL_0000:  ldarg.0
      IL_0001:  call       instance void
      ↪  [System.Runtime]System.Object::.ctor()
      IL_0006:  ret
    } // end of method Program::.ctor

  } // end of class HelloIldasm.Program

...
```

Цей приклад показує використання інструменту `ildasm` для декомпіляції .NET-застосунків, що дозволяє переглядати структуру коду і методи програми.

Аналіз мережевого трафіку

Аналіз мережевого трафіку дозволяє дослідити, як програма взаємодіє з мережею, а також виявити можливі вразливості або небезпечні дії. Наприклад, можна визначити, чи передаються конфіденційні дані через незахищені з'єднання або чи встановлює програма зв'язок із підозрілими серверами. Однією з найбільш поширених утиліт для такого аналізу є *Wireshark* (рис. 14.1).

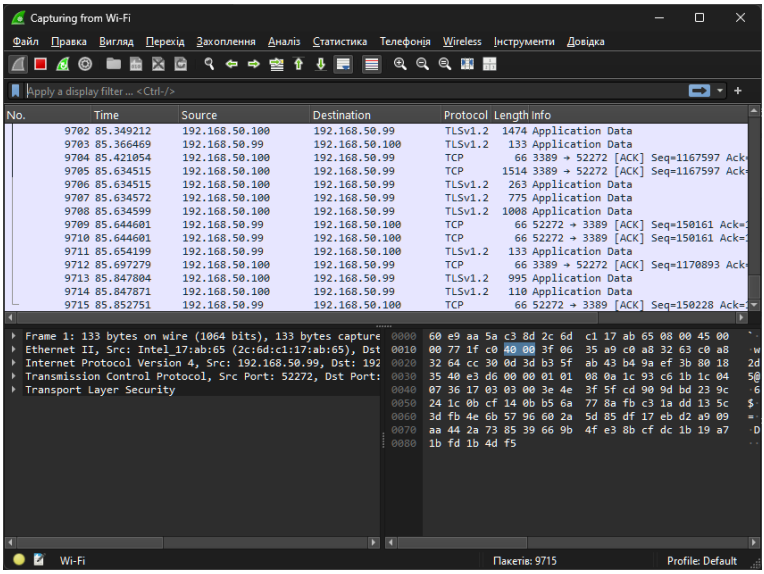


Рисунок 14.1 – *Wireshark*

Приклад аналізу трафіку за допомогою *Wireshark*.

```
wireshark -i eth0 -k
```

Цей приклад демонструє, як можна використати Wireshark для моніторингу трафіку мережевого інтерфейсу, дозволяючи вивчити пакети, які передає або отримує програма.

Спроби підміни вебсервера та методи їх виявлення

Зловмисники можуть спробувати підмінити вебсервер для викрадення даних користувачів або впровадження шкідливого коду. Такі атаки можуть бути реалізовані за допомогою *DNS*-спуфінгу або атак типу «людина посередині» (*Man-in-the-Middle, MitM*). Виявлення спроб підміни вебсервера включає аналіз *SSL/TLS*-сертифікатів та перевірку відповідності *IP*-адрес оригінальному серверу.

Приклад перевірки *SSL*-сертифіката за допомогою `openssl`

```
openssl s_client -connect example.com:443
```

Ця команда відкриває *SSL*-з'єднання з вебсервером і виводить інформацію про його сертифікат. Це дозволяє перевірити, чи відповідає сертифікат очікуваному.

Методи виявлення підміни вебсервера.

Для виявлення підміни вебсервера можна використовувати різні методи, такі як:

- перевірку *SSL/TLS*-сертифікатів;
- використання інструментів моніторингу мережі для виявлення змін у *DNS*-записах;
- аналіз *IP-адрес* і маршрутів для виявлення відхилень від нормальних параметрів мережі.

Контрольні запитання

1. Що таке зворотна розробка і які її основні методи?
2. Як працює декомпіляція і які інструменти використовуються для цього процесу?
3. Що таке пакування програм і як можна провести аналіз пакованого файлу?
4. Як можна використовувати *Wireshark* для аналізу мережевого трафіку і виявлення загроз?
5. Що таке *DNS*-спуфінг і як він використовується для підміни вебсервера?
6. Як перевірка *SSL*-сертифікатів допомагає виявити підміну вебсервера?
7. Які методи виявлення підміни вебсервера можна використовувати для захисту системи?
8. Що таке дизасемблювання і як воно допомагає в процесі зворотної розробки?
9. Які основні ознаки можуть свідчити про можливу підміну вебсервера?
10. Як аналіз бінарних файлів може допомогти виявити шкідливе програмне забезпечення?

Тема 15

МЕТОДИ ЗАХИСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА КОМП'ЮТЕРНИХ ІГОР

Метою є вивчення методів захисту програмного забезпечення та комп'ютерних ігор від несанкціонованого доступу й модифікацій, набуття навичок аналізу типових загроз безпеки комп'ютерних ігор.

Завдання для самостійної роботи

1. *Реалізуйте захист від шахрайства в онлайн-грі* (наприклад, перевірка цілісності клієнта або виявлення відомих читерських програм).
2. *Забезпечте цілісність даних між клієнтом і сервером*, застосувавши механізми шифрування або цифрового підпису.
3. *Налаштуйте систему виявлення модифікованих файлів гри*, яка сигналізує про зміни в ресурсах чи виконуваних файлах.
4. *Створіть механізм ліцензування* для десктопного ПЗ (генерація ключів, валідація підпису).
5. *Проведіть тестування гри на стійкість до DDoS-атак*, змоделювавши сценарії високого навантаження.

Короткі теоретичні відомості

1. Важливість захисту програмного забезпечення.
2. Основні методи захисту програмного забезпечення.
3. Захист комп'ютерних ігор від піратства та зломів.
4. Правові аспекти захисту програмного забезпечення.

Методи захисту програмного забезпечення

Захист програмного забезпечення – це набір методів і технологій, які використовуються для запобігання несанкціонованому використанню, модифікації та зворотній розробці програм. Вони включають підписування коду, обфускацію, водяні знаки, а також механізми виявлення модифікацій.

Основні методи захисту:

• *підписування коду (Digital Signing)* – використання цифрових підписів для підтвердження цілісності та автентичності програмного забезпечення;

• *обфускація коду (Software Obfuscation)* – техніка, що ускладнює читання вихідного коду, щоб ускладнити зворотну розробку;

• *захист від модифікацій (Anti-Tampering and Binary Integrity Checking)* – технології, які виявляють зміни у програмному забезпеченні і можуть запустити механізми припинення роботи програми;

• *апаратні ключі (Software Protection Dongles)* – спеціальні апаратні пристрої, які надають доступ до програмного забезпечення лише при їх підключенні до комп'ютера, що забезпечує захист від несанкціонованого копіювання або використання;

• *сервери ліцензування (Cloud Licensing Location)* – віддалені сервери, що керують ліцензіями програмного забезпечення і перевіряють їх дійсність при запуску або використанні програмного забезпечення;

• *White-box Cryptography* – метод захисту криптографічних алгоритмів, де ключі залишаються захищеними навіть при доступі до коду і даних, що забезпечує безпечну роботу криптографічних операцій в умовах потенційного зловмисного середовища.

Підписування коду

Криптографічні методи відіграють ключову роль у захисті програмного забезпечення, надаючи гарантії цілісності та походження. Використання хешування та цифрових підписів дозволяє перевірити, що програмне забезпечення не було модифіковано, а його походження є автентичним.

Основні механізми захисту

Основні механізми захисту програмного забезпечення включають:

• *хешування (*Hashing*)* – використовує алгоритми, такі як *SHA-256*, для створення унікального хешу, що дозволяє перевірити цілісність файлу;

• *шифрування (*Encryption*)* – перетворює дані у недоступний вигляд, що ускладнює їх читання без відповідного ключа;

• *цифрові підписи (*Digital Signatures*)* – дозволяють перевірити автентичність джерела програмного забезпечення.

Підписування коду – це процес цифрового підпису виконуваних файлів та скриптів для підтвердження їхньої автентичності операційній системі та користувачу. Цей метод використовує криптографічний підпис та сертифікат для перевірки справжності програмного забезпечення, але не гарантує захист від змін в полі.

Підписування коду зазвичай використовує цифровий підпис для підтвердження автора або системи збірки та контрольну суму для виявлення модифікацій.

Підпис обов'язково перевіряється операційною системою під час запуску

системи, а для плагінів та доповнень – застосунком, що їх завантажує, під час ініціалізації. Підписування коду є обов'язковим у багатьох екосистемах, оскільки гарантує, що програмне забезпечення надходить із довіреного джерела. Однак воно не повинно бути єдиним засобом захисту, оскільки може бути обійдене у випадку компрометації системи.

Приклад підписування коду за допомогою *OpenSSL*

```
openssl dgst -sha256 -sign private.pem -out signature.bin program.exe
openssl dgst -sha256 -verify public.pem -signature signature.bin
↪  program.exe
```

Цей код демонструє використання OpenSSL для підписування та верифікації виконуваного файлу, що забезпечує його цілісність та автентичність. Перший рядок виконує підписування файлу `program.exe` за допомогою приватного ключа `private.pem`, створюючи файл підпису `signature.bin`. Другий рядок перевіряє підпис, використовуючи відповідний публічний ключ `public.pem`.

Захист від зворотної розробки

Захист програмного забезпечення включає методи, які ускладнюють або роблять економічно невигідною зворотну розробку (reverse engineering). Обфускація коду, використання алгоритмів шифрування або спеціальних інструкцій можуть суттєво ускладнити зворотну розробку. Сюда також відносят метолди захисту від динамічного аналізу,

Обфускація коду

Обфускація коду – це техніка захисту, яка змінює структуру та вигляд вихідного коду, зберігаючи його функціональність, але роблячи його важким для аналізу та зворотного інжинірингу (Рис. 15.1[1]). Це популярний спосіб захисту комерційного програмного забезпечення від несанкціонованого доступу і злому.

Основні методи обфускації:

• *перейменування символів (Symbol Renaming):* імена класів, методів і змінних змінюються на випадкові або короткі значення (наприклад, `Class123`, `MethodXYZ`, або `varA1`), що робить їх незрозумілими для людини. Наприклад,

[1] JSObfusDetector

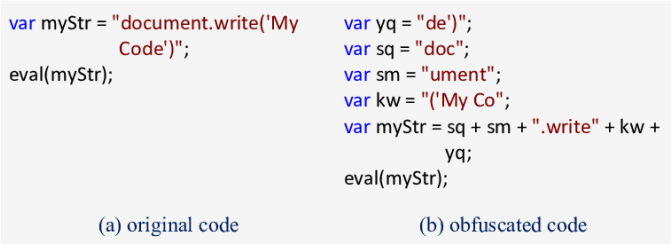


Рисунок 15.1 – Обфускація коду

змінна `customerBalance` може бути перейменована в `a1`, що ускладнює розуміння її функціоналу;

• *видалення метаданих:* обфускатор може видаляти метадані, що знижує кількість контексту для аналізу вихідного коду;

• *структурна обфускація:* логіка коду змінюється на складнішу форму. Наприклад, умовні вирази можуть бути перетворені на більш заплутані, але еквівалентні;

• *інлайнінг (Inlining) та видалення абстракцій:* абстракції, такі як функції або класи, можуть бути «інлайнові» або об'єднані, що ускладнює аналіз і зворотний інжиніринг;

• *поліморфізм та динамічний контроль потоку:* логічна структура стає динамічною, що робить її важчою для передбачення та розуміння.

• *шифрування рядків:* статичні рядки з важливою інформацією, наприклад *URL* або повідомлення, можуть бути зашифровані та розшифровуються лише під час виконання;

Популярні інструменти для обфускації:

• *ProGuard* – інструмент для обфускації *Java* та Android-застосунків;
• *Dotfuscator* – інструмент для обфускації *.NET*-застосунків;
• *JavaScript Obfuscator* – інструмент для обфускації *Java*Script-коду;
• *Stunnix* C *і* C++ *Obfuscator* – інструмент для обфускації *C* і *C++* коду.

Міжнародний конкурс обфускації *C*-коду

Міжнародний конкурс обфускації коду на *C*(*IOCCC*) є цікавим ресурсом для дослідження технік обфускації коду. Результати переможців демонструють нескінченну кількість творчих ідей та нестандартних підходів до програмування мовою *C*, надаючи уявлення про те, як код може бути навмисно змінений до невпізнанності та бути складним для читання та розуміння.

Аналізуючи роботи переможців, дослідники, розробники і просто ентузіасти програмування можуть глибше зрозуміти методи обфускації, включаючи складний контроль потоку, нестандартний синтаксис і неочевидне використання можливостей мови.

Варто зазначити, що подібна обфускація є можливою тільки завдяки унікальним можливостям та гнучкості мови програмування *C*.

Нижче наведено програму, яка «розв'язує класичну задачу про n ферзів на шахівниці розміром до 99x99, зберігаючи програму якомога коротшою та використовуючи лише цикли `for`»[2]:

```
v,i,j,k,l,s,a[99];
main()
{
    for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&(!k&&!!pr
    ↪  intf(2+"\n\n%c"-(!l<<!j),"
    ↪  #Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),
    ↪  v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i])
        ;
}
```

Це приклад обфускованої програми мовою C, яка реалізує алгоритм зворотного пошуку (backtracking) для генерування числової конфігурації на основі введеного значення `s`. У програмі використано мінімалістичний синтаксис, щільні логічні вирази та вбудовані виклики `printf` для відображення проміжного результату.

```
#include <stdio.h>
int v, i, j, k, l, s, a[99];
int main() {
    scanf("%d", &s);
    while (*a - s) {
        v = a[j *= v] - a[i];
        k = (i < s);
        j += (
            v = (j < s) &&
            (
                !k &&
```

[2] https://www.ioccc.org/1990/baruch.c

```
                !!printf(2 + "\n\n%c" - (!l << !j), " #Q"[(l ^ v) ?
                ↪  ((l ^ j) & 1) : 2]) &&
                ++l || (a[i] < s && v && v - i + j && v + i - j)
            )
        ) && !(l %= s);
        if (!v) {
            if (i == j)
                a[i += k] = 0;
            else if (++a[i] >= s * k)
                ++a[--i];
        }
    }
    return 0;
}
```

Ще один цікавий приклад обфускованого коду програми одного з переможців конкурсу *IOCCC* 2020 року[3]:

```
#define/**/Q(x,y)char*/*                                */q=y#x","#y")",*p,s[x;}
  /*IOCCC'20*/#include/*                                */<stdio.h>/*-Qlock-*/
    int(y),x,i,k,r;Q(9/*             12                */<<9];float(o)[03];
      void(P)(){*o=r<0/*         11         1           */?r:-r;o[1]=39.5;
       o[2]=22.5;for(k/*      10               2       */=0;++k<39;*o*=i
        /6875.5/(k%2?k/*                                */:-k))y=o[1+k%2
        ]+=*o;k=o[2];/*     9         o-----> 3       */p=s+y+k/2*80;
         }int(main)()/*              /                  */{for(p=s;+i<
         1839;*q>32?k/*       8     L         4         */=i++/80-11,y
          =(750>r*r+k/*          7         5             */*k*4)*4+y/2
          ,*p++=r<41?/*               6                 */y?"0X+0X+!"
          [y-1]-1:+*q/*                                 */++:10:*q++)
          r=i%80-38;;/*                                 */;for(x=13,r
          =20;i=3600*/*     \       /    -----+         */--x,i;*p++=
         "OISEA2dC8e"/*      \     /     ---- |         */[x%10],*p+=x
         /10*41)P();r/*       \   /      ---- |         */=10;;sscanf(
        __TIME__,"%d"/*        \ /       ---- |         */":%d:%d",&k,&
       x,&i);for(i+=(/*         X        ---- |         */k*60+x)*60;18+
      r;*p=k%2?*p%2?+/*      __/ \__         |  |       */59:44:*p>39?59:
     39,i=!r--?i%3600/*     /  \ /  \        |  |       */*12:i)P();puts(s
   ),"#define/**/Q(x"/*     \__/ \__/        +--+       */",y)char*q=y#x\","
  "\"#y\")\",*p,s[x;}"/*                                */"/*IOCCC'20*/#inclu"
"de<stdio.h>/*-Qlock-"/*                                */"*/int(y),x,i,k,r;Q(")
```

[3]https://www.ioccc.org/2020/endoh3/index.html

Обмеження обфускації

Обфускація суттєво ускладнює зворотний інжиніринг, але досвідчені розробники, в т.ч. зловмисники, можуть використовувати автоматизовані інструменти для декомпіляції та покрокового виконання коду програми. Проте, вона створює додаткові бар'єри, які підвищують захист програмного забезпечення від злому.

Методи виявлення змін та припинення роботи

Іншою задачею є захист від несанкціонованого доступу та змін бінарного коду, метою якого є захистити програмне забезпечення від модифікацій та використання способами, що не передбачені розробниками. Захист від несанкціонованого втручання розроблений так, щоб під час виконання програми відбувався плавний збій без надання жодних підказок щодо причин, чому модифікований код не працює. Хоча він не заважає іншим розробникам вивчати код виконуваного файлу, цей метод забезпечує ефективний захист і суттєво ускладнює динамічний аналіз програмного забезпечення.

Технології виявлення змін у коді дозволяють ідентифікувати модифікації програмного забезпечення. Програми можуть періодично перевіряти цілісність своїх файлів або використовувати механізми хешування для перевірки критичних компонентів. У разі виявлення змін такі системи можуть припинити роботу програми або заблокувати доступ до певних функцій.

Захист від несанкціонованого доступу може бути реалізований як усередині, так і ззовні програми, що захищається.

Програмне забезпечення для захисту від несанкціонованого доступу застосовується в багатьох галузях: вбудовані системи, фінансові застосунки, програмне забезпечення для мобільних пристроїв, системи мережевих пристроїв, боротьба з шахрайством в іграх тощо.

Зазначимо, що ті самі методи часто використовують розробники шкідливого програмного забезпечення з тією ж метою – ускладнити динамічний аналіз коду вірусів, руткітів, хробаків тощо.

Приклад виявлення змін у файлі

```
String originalHash = "abcd1234...";
String currentHash = calculateHash("program.exe");
if (!originalHash.equals(currentHash)) {
        System.exit(1);  // Закінчити роботу програми
    }
```

Апаратні ключі (Software Protection Dongles)

Апаратні ключі безпеки, наприклад флешнакопичувачі, – це двоінтерфейсні маркери, що використовуються для захисту програм від несанкціонованого використання. За відсутности таких ключів програмне забезпечення може працювати в обмеженому режимі або не запускатися взагалі.

Сучасні типові ключі мають містити енергонезалежну пам'ять що дозволяє, наприклад, зберігати та виконувати певні частини програмного забезпечення на ключі, також вони мають вбудоване стійке шифрування та використовують технології виготовлення, що унеможливлюють зворотне проєктування.

Теоретично такі ключі можна клонувати використовуючи методи клонування апаратного забезпечення, щоб запобігти цьому можуть використовувати спеціальні smart-картки.

Сервери ліцензування (Cloud Licensing Location)

Зважаючи на недоліки традиційних методів захисту, перехід на хмарні технології у сфері ліцензування забезпечує вищий рівень безпеки, ніж системи з апаратними ключами. *Хмарне ліцензування* програмного забезпечення усуває ризик втрати або пошкодження ключів чи локальних серверів ліцензування, а також суттєво ускладнює несанкціоноване використання. Проте, попри всі переваги, хмарне ліцензування може бути зламане на клієнтських комп'ютерах, тому клієнтське програмне забезпечення також потребує захисту від зворотного проєктування та модифікації.

Криптографія «білого ящика» (White-box cryptography)

Основна ідея методів криптографії «білого ящика» полягає в об'єднанні ключа і коду криптоалгоритму в новий, перетворений код. Ключ ефективно прихований у коді та не може бути легко виділений. Ці методи зосереджені на захисті криптографічних алгоритмів і ключів у середовищах, де є доступ до виконуваного коду. Їх метою є ускладнення аналізу криптографічних операцій для збереження конфіденційності ключів і захисту алгоритмів, навіть якщо зловмисник має доступ до коду.

На сьогодні відомі комерційні реалізації методів криптографії «білого ящика» для основних симетричних блокових шифрів (*AES* і *DES*). Додатково вони можуть включати реалізацію *хешування*, *RSA* та методи еліптичної криптографії.

Ці методи можна вважати вдосконаленими техніками обфускації програмного коду.

Захист програмного забезпечення комп'ютерних ігор

У питанні захисту комп'ютерних ігор існують дві основні проблеми – захист інтелектуальних прав на зміст (юридичні питання) та захист ігрової екосистеми (технічні та організаційні питання).

Захист інтелектуальних прав поширюється на такі компоненти ігор, як:

- мультимедійна підсистема гри – аудіовізуальні ефекти, музичний супровід, відеоряд, аудіоряд, анімація, графічні елементи (логотипи, налаштування, персонажі, об'єкти, інтерфейс, шрифти тощо);
- сценарій;
- персонажі;
- бази даних, включно з їхніми структурами;
- керівництво користувача;
- …та багато інших компонентів.

Для захисту ігрової екосистеми застосовують розглянуті вище методи, але сьогодні, зважаючи на поширення онлайн-ігор та мобільних застосунків, існують додаткові специфічні проблеми, пов'язані з ігровим процесом.

Історично для захисту комп'ютерних ігор використовували складні апаратні методи захисту, які обмежували клонування носіїв із іграми, певним чином впливаючи на носій, а також апаратні ключі захисту, що запобігали несанкціонованому запуску гри.

Мережеві ігри вразливі до атак через мережу, тому важливо застосовувати принципи безпечного програмування:

- *шифрування мережевого трафіку* – використання протоколів шифрування, таких як TLS або DTLS, для захисту ігрових транзакцій від перехоплення або модифікації;
- *аутентифікація гравців* – перевірка автентичності кожного гравця для запобігання несанкціонованому доступу до серверів гри та запобігання використанню підроблених облікових записів;
- *захист персональних даних* – забезпечення відповідності локальному законодавству, такому як GDPR, для збереження конфіденційності та безпеки особистої інформації гравців;
- *виявлення аномальної активності*– використання журналів, систем моніторингу та інструментів аналізу поведінки для виявлення підозрілої активності, такої як шахрайство або атаки типу *DDoS*.

Правові аспекти захисту програмного забезпечення

Правові аспекти захисту програмного забезпечення охоплюють різні частини ПЗ, пов'язані з авторським правом, патентами, ліцензуванням та іншими юридичними питаннями. Вони включають:

• *авторське право (Copyright)* захищає програмне забезпечення як твір, надаючи виключне право на відтворення, поширення та модифікацію;

• *патенти на програмне забезпечення* дозволяють захистити нові та корисні технічні рішення, якщо вони відповідають вимогам патентного законодавства;

• *ліцензійні угоди (EULA і SLA)* регулюють використання програмного забезпечення, встановлюючи права та відповідальність сторін;

• *закони про захист персональних даних* вимагають забезпечення конфіденційності даних користувачів відповідно до локального законодавства (наприклад, *GDPR*);

• *захист торгових секретів* передбачає збереження унікальних алгоритмів або технологій у таємниці як комерційної інформації;

• *антипіратські закони й заходи* передбачають покарання за незаконне копіювання та розповсюдження програмного забезпечення. У багатьох країнах існує відповідальність за використання торентів як інструменту піратства;

• *захист прав споживачів* передбачає дотримання прав споживачів на отримання якісного програмного забезпечення, а також можливість повернення або обміну продукту у разі невідповідності заявленим характеристикам;

• *управління цифровими правами (DRM)* для обмеження доступу та контролю використання програмного забезпечення та мультимедіа.

Контрольні запитання

1. Що таке підписування коду і як воно захищає програмне забезпечення?
2. Як працює обфускація і яким чином вона ускладнює зворотну розробку?
3. Що таке водяні знаки у програмному забезпеченні і як вони допомагають захищати програми?
4. Які криптографічні гарантії забезпечують цілісність та походження програмного забезпечення?
5. Які методи виявлення модифікацій використовуються для захисту програмного забезпечення?
6. Як можна захистити мережеві комп'ютерні ігри від атак через мережу?
7. Які методи використовуються для захисту комп'ютерних ігор від піратства?
8. Що таке DRM і як воно допомагає захистити програмне забезпечення від несанкціонованого використання?
9. Як антишахрайські (античитінгові) засоби захищають мережеві ігри від сторонніх програм?
10. Які принципи безпечного програмування важливо застосовувати для багатокористувацьких ігор?

Тема 16

ЗАХИСТ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ HMAC ТА ЕЛЕКТРОННОГО ЦИФРОВОГО ПІДПИСУ

Метою є ознайомлення з механізмами *HMAC* і електронного цифрового підпису для забезпечення цілісності та автентичності даних, набуття навичок роботи з цифровим підписом у мовах програмування.

Завдання для самостійної роботи

1. *Реалізуйте* HMAC *(Hash-Based Message Authentication Code)* у вибраній мові програмування (наприклад, *Python*). Поясніть принцип роботи *HMAC*, зосередьтеся на механізмі ключа та способі формування підпису.
2. *Згенеруйте ЕЦП для файлу за допомогою* OpenSSL:
   - Створіть пару ключів (приватний і публічний).
   - Підпишіть обраний файл приватним ключем.
   - Перевірте дійсність підпису публічним ключем.
   - Задокументуйте всі команди та опції, які використовувалися.
3. *Реалізуйте простий механізм цифрового підпису на Python* (використовуючи бібліотеку `cryptography` або *PyOpenSSL*). Продемонструйте процес підпису та перевірки (наприклад, для текстового файлу).
4. *Порівняйте алгоритми ЕЦП – RSA, ECDSA, EdDSA* – звернувши увагу на:
   - Продуктивність (швидкість підпису та перевірки);
   - Розмір ключів і підписів;
   - Рівень безпеки (стійкість до криптоаналітичних атак).

Зробіть висновки про доцільність використання кожного алгоритму в різних сценаріях.
5. *Поясніть різницю між HMAC та ЕЦП*, зверніть увагу на:
   - Спосіб створення підпису (симетричний або асиметричний);
   - Безпекові переваги та обмеження;
   - Типові області застосування (захист вебзапитів, підпис документів тощо).

Наведіть приклади конкретних ситуацій, коли варто обрати *HMAC* або ЕЦП.

Короткі теоретичні відомості

1. Огляд *HMAC* (*Хеш-код із використанням секретного ключа*).

2. Використання *НМАС* для забезпечення цілісності та автентичності.
3. Огляд електронного цифрового підпису (*ЕЦП*).
4. Використання *ЕЦП* для захисту програмного забезпечення.
5. Порівняння *НМАС* та *ЕЦП* для різних типів застосунків.

НМАС: забезпечення цілісності та автентичності даних

HMAC (*Hashed Message Authentication Code*) – це метод автентифікації повідомлень, який використовує хеш-функцію разом із секретним ключем для забезпечення цілісності та автентичності даних. *HMAC* гарантує, що повідомлення не було змінено під час передачі, і що його відправник є автентичним (Рис. 16.1[1]).

Основні властивості *HMAC*:

• *цілісність даних* – *HMAC* забезпечує виявлення змін у даних під час їх передачі;

• *автентичність* – лише сторони, що мають спільний секретний ключ, можуть створити або перевірити *HMAC*.

HMAC працює за принципом комбінування повідомлення з секретним ключем та обчислення хеш-функції, такої як *SHA-256* або *MD5*, для створення унікального коду автентифікації повідомлення.

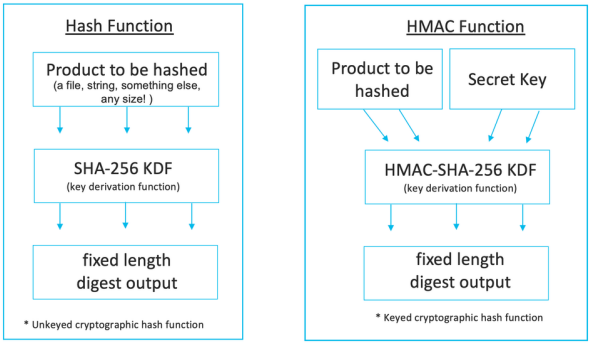


Рисунок 16.1 – *HMAC*

[1]CISCO Community: Configuring & Understanding OSPF HMAC Authentication

Приклад використання *HMAC* з *SHA-256*.

```
#include <openssl/hmac.h>

unsigned char *result;
unsigned int len = 32;
result = HMAC(EVP_sha256(), key, strlen(key), data, strlen(data),
↪  NULL, NULL);
```

Цей код демонструє використання *HMAC* з алгоритмом *SHA-256* для обчислення хешу повідомлення із секретним ключем.

HMAC широко використовується для забезпечення безпеки *API*-запитів, автентифікації повідомлень у протоколах (наприклад, *TLS*) та захисту конфіденційних даних від модифікацій.

Електронний цифровий підпис: автентичність та невідкличність повідомлень

Електронний цифровий підпис (*Digital Signature*) – це криптографічний метод, що забезпечує автентичність, цілісність та невідкличність електронних документів або повідомлень (Рис. 16.2[2]). Цей метод використовує пару ключів: приватний ключ, яким підписується документ, та публічний ключ, яким цей підпис можна перевірити.

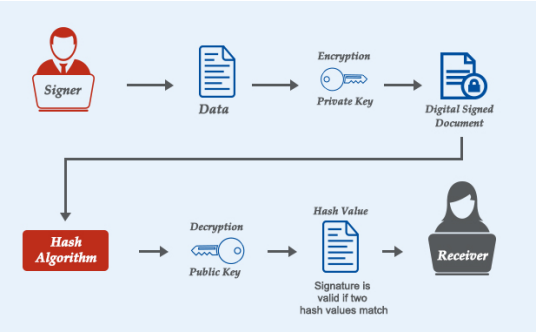


Рисунок 16.2 – Цифровий підпис

Основні властивості електронного цифрового підпису:

[2]Comodo: What is Digital Signature? How does it Work?

• *автентичність* – підпис підтверджує, що документ був підписаний власником приватного ключа;

• *цілісність* – підпис також підтверджує, що документ не був змінений після його підписання;

• *невідкличність* – відправник не може заперечувати факт підписання документа, оскільки підпис є унікальним і пов'язаний з приватним ключем відправника.

Процес створення цифрового підпису включає обчислення хешу повідомлення та шифрування цього хешу приватним ключем відправника. Одержувач може перевірити підпис, розшифрувавши його публічним ключем відправника та порівнявши отриманий хеш з хешем самого повідомлення.

Приклад створення електронного підпису за допомогою *OpenSSL*.

```
EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
EVP_PKEY *private_key = load_private_key("private_key.pem");

EVP_SignInit(mdctx, EVP_sha256());
EVP_SignUpdate(mdctx, message, strlen(message));
EVP_SignFinal(mdctx, signature, &sig_len, private_key);
EVP_MD_CTX_free(mdctx);
```

Цей код демонструє, як створити електронний підпис повідомлення за допомогою *SHA-256* та приватного ключа.

Цифровий підпис широко використовується у таких сферах, як електронна комерція, юридичні документи, та для захисту програмного забезпечення від несанкціонованої модифікації. Наприклад, програмне забезпечення може бути підписане розробником, і користувачі можуть перевіряти підпис перед встановленням, щоб переконатися в його автентичності та цілісності.

Роль цифрового підпису та *HMAC* у забезпеченні безпеки

Обидва методи – *HMAC* і цифровий підпис – є інструментами у забезпеченні безпеки програмного забезпечення та даних. *HMAC* ефективний у сценаріях, коли обидві сторони мають спільний секретний ключ і необхідно забезпечити автентичність і цілісність повідомлень без потреби в асиметричному шифруванні. Цифровий підпис, зі свого боку, надає гарантії невідкличності та автентичності за допомогою асиметричних ключів.

HMAC і цифровий підпис використовуються в різних сценаріях:

• *HMAC* – підходить для автентифікації повідомлень у захищених каналах зв'язку (наприклад, у *VPN* або *TLS* (Рис. 16.3[3])));

• *цифровий підпис* – підходить для автентифікації і захисту документів, програмного забезпечення, а також для довгострокової гарантії автентичності.

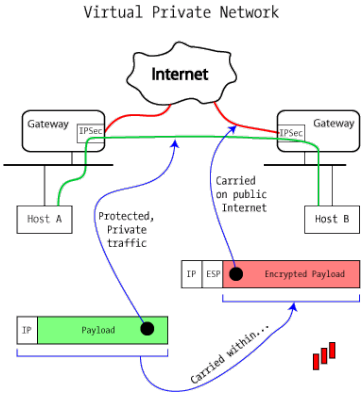


Рисунок 16.3 – *IPSec*

Контрольні запитання

1. Що таке *HMAC* і як він забезпечує цілісність та автентичність даних?
2. Як використовується *HMAC* у захищених комунікаціях, таких як *TLS* або *VPN*?
3. Що таке електронний цифровий підпис і як він забезпечує автентичність та невідкличність повідомлень?
4. У чому полягає різниця між *HMAC* і цифровим підписом?
5. Як працює асиметричне шифрування в контексті електронного підпису?
6. Як забезпечується перевірка цілісності підписаного повідомлення або документа?
7. Які алгоритми хешування можуть використовуватися в *HMAC* та електронному підписі?
8. У яких сценаріях *HMAC* є більш доцільним для використання, ніж цифровий підпис?
9. Як *OpenSSL* допомагає у створенні *HMAC* та електронних підписів?
10. Які переваги і недоліки використання електронного підпису в порівнянні з іншими методами автентифікації?

[3] An Illustrated Guide to IPsec (www.unixwiz.net)

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Савченко В., Мнушка О.* Сучасні технології безпечного програмування : Навчальний посібник. – Харків : НТУ «ХПІ», 2024. – 180 с.
2. *Савченко В., Мнушка О.* Сучасні технології безпечного програмування. Практикум : Навчально-методичний посібник. – Харків : ФОП Бровін О.В., 2024. – 132 с. – ISBN 978-617-8238-77-3. – URL: https://repository.kpi.kharkov.ua/handle/KhPI-Press/85832 (дата зверн. 26.11.2024).
3. *Остапов С., Євсеєв С., Король О.* Кібербезпека: сучасні технології захисту : Навчальний посібник. – Харків : Новий світ, 2020. – 678 с. – URL: http://library.kpi.kharkov.ua/en/inftechnologies/кібербезпека-сучасні-технології-захисту (дата зверн. 26.11.2024).
4. *Лісовська Ю.* Кібербезпека: ризики та заходи : Навчальний посібник. – Київ : Видавничий дім «КОНДОР», 2019. – 272 с. – URL: http://library.kpi.kharkov.ua/files/new_postupleniya/kibrtz.pdf (дата зверн. 26.11.2024).
5. *Тарнавський Ю. А.* Технології захисту інформації : Підручник. – Київ : КПІ ім. Ігоря Сікорського, 2018. – 162 с. – URL: https://ela.kpi.ua/handle/123456789/23896 (дата зверн. 26.11.2024).
6. *Helfrich J. N.* Security for Software Engineers. – Boca Raton : CRC Press, Taylor & Francis Group, 2020. – URL: http://repo.darmajaya.ac.id/4635/1/SecurityforSoftwareEngineers(PDFDrive).pdf (дата зверн. 26.11.2024).
7. *Winkler I., Gomes A. T.* Advanced Persistent Security. – Amsterdam : Syngress / Elsevier, 2017. – URL: https://www.sciencedirect.com/book/9780128093160/advanced-persistent-security (дата зверн. 30.10.2024).
8. *Paulsen C., Byers R.* Glossary of Key Information Security Terms. – National Institute of Standards, Technology, 2019. – URL: https://nvlpubs.nist.gov/nistpubs/ir/2013/nist.ir.7298r2.pdf (дата зверн. 25.11.2024).
9. *Committee on National Security Systems*. CNSS Glossary. – 2022. – URL: https://rmf.org/wp-content/uploads/2017/10/CNSSI-4009.pdf (дата зверн. 25.11.2024).
10. *Federal Bureau of Investigation*. Internet Crime Report 2023. – 2023. – URL: https://www.ic3.gov/Media/PDF/AnnualReport/2023_IC3Report.pdf (дата зверн. 30.04.2024).
11. *Samani R.* An Analysis of the WannaCry Ransomware Outbreak. – 2017. – URL: https://www.mcafee.com/blogs/other-blogs/executive-perspectives/analysis-wannacry-ransomware-outbreak/ (дата зверн. 30.04.2024).

12. *Microsoft Security Response Center*. Analyzing Solorigate, the compromised DLL used in the SolarWinds attack. – 2020. – URL: https://www.microsoft.com/en-us/security/blog/2020/12/18/analyzing-solorigate-the-compromised-dll-file-that-started-a-sophisticated-cyberattack-and-how-microsoft-defender-helps-protect/ (дата зверн. 30.04.2024).

13. *Microsoft Security*. Guidance for responders: Investigating and remediating on-premises Exchange Server vulnerabilities. – 2021. – URL: https://msrc.microsoft.com/blog/2021/03/guidance-for-responders-investigating-and-remediating-on-premises-exchange-server-vulnerabilities/ (дата зверн. 30.04.2024).

14. *Miller M.* Deepfakes: A Real Threat to Cybersecurity. – 2023. – URL: https://kpmg.com/kpmg-us/content/dam/kpmg/pdf/2023/deepfakes-real-threat.pdf (дата зверн. 30.04.2024).

15. *European Union Agency for Law Enforcement Cooperation*. Facing reality?: law enforcement and the challenge of deepfakes : an observatory report from the Europol innovation lab. – Publications Office, 2024. – DOI: 10.2813/158794.

16. Proceedings of the 32nd USENIX Security Symposium : USENIX Security Symposium. – [Berkeley, CA] : USENIX Association, 2023. – 490 с. – ISBN 978-1-939133-37-3.

17. Language Models are Few-Shot Learners / T. B. Brown [та ін.]. – 2020. – DOI: 10.48550/ARXIV.2005.14165.

18. *OWASP Foundation*. OWASP Top 10 - 2021. – 2021. – URL: https://owasp.org/www-project-top-ten/ (дата зверн. 30.10.2024).

19. *Software Engineering Institute*. SEI CERT C Coding Standard. – Вер. 01. – Carnegie Mellon University, 2016. – URL: https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf (дата зверн. 15.05.2024).

20. *Ballman A.* SEI CERT C++ Coding Standard. – Вер. 01. – Carnegie Mellon University, 2016. – URL: https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf (дата зверн. 15.05.2024).

21. The CERT Oracle Secure Coding Standard for *Java* / за ред. F. W. Long. – Upper Saddle River, NJ : Addison-Wesley, 2012. – 699 с. – (The SEI series in software engineering). – ISBN 0-13-288284-1.

22. *Software Engineering Institute*. CERT Secure Coding Standards. – 2021. – URL: https://www.securecoding.cert.org (дата зверн. 30.04.2024).

23. *European Union*. General Data Protection Regulation (GDPR). – 2016. – URL: https://gdpr.eu/ (дата зверн. 30.10.2024).

24. *PCI Security Standards Council*. PCI Security Standards Overview. – 2021. – URL: https : / / www . pcisecuritystandards . org / standards/ (дата зверн. 30.10.2024).

25. *OpenSSF*. Secure Coding Guide for *Python*. – 2024. – URL: https://github.com/ossf/wg-best-practices-os-developers/tree/main/docs/Secure-Coding-Guide-for-Python (дата зверн. 12.05.2024).

26. *MITRE Corporation*. 2024 CWE Top 25 Most Dangerous Software Weaknesses. – 2024. – URL: https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html (дата зверн. 15.12.2024).

27. *Checkmarx*. JavaScript Secure Coding Practices (JS-SCP). – 2024. – URL: https://github.com/Checkmarx/JS-SCP (дата зверн. 12.05.2024).

28. *Amazon Web Services*. What is Virtualization? – 2024. – URL: https://aws.amazon.com/what-is/virtualization/ (дата зверн. 15.05.2024).

29. NIST cloud computing reference architecture / F. Liu [та ін.]. – 2011. – DOI: 10.6028/nist.sp.500-292.

30. *The MITRE Corporation*. Common Weakness Enumeration (CWE). – 2024. – URL: https://cwe.mitre.org/index.html (дата зверн. 15.05.2024).

31. *The MITRE Corporation*. Common Vulnerabilities and Exposures (CVE). – 2024. – URL: https://www.cve.org/ (дата зверн. 15.05.2024).

32. *Google*. Prepare for the phaseout of third-party cookies. – 2024. – URL: https://support.google.com/google-ads/answer/1033961 (дата зверн. 15.05.2024).

33. *Wagner K.* Google Third Party Cookies Phase Out in 2024: Timeline and Tips. – 03.2024. – URL: https://kattwagner.com/google-third-party-cookies-phase-out-in-2024 (дата зверн. 15.05.2024).

34. *Google*. A more private web: Privacy Sandbox. – 2023. – URL: https : / / privacysandbox.com/ (дата зверн. 15.05.2024).

35. *Software Engineering Institute*. Android Secure Coding Standard. – 2024. – URL: https://wiki.sei.cmu.edu/confluence/display/android/Android+Secure+Coding+Standard (дата зверн. 25.05.2024).

36. *JSSEC*. Android Secure Coding. – 2018. – URL: https://www.jssec.org/dl/android_securecoding_en.pdf (дата зверн. 15.05.2024).

37. *JSSEC*. Security Guidelines 2012. – 2012. – URL: https://www.jssec.org/dl/guidelines2012Enew_v1.0.pdf (дата зверн. 15.05.2024).

Навчальне видання

САВЧЕНКО Володимир Миколайович
МНУШКА Оксана Василівна

СУЧАСНІ ТЕХНОЛОГІЇ БЕЗПЕЧНОГО ПРОГРАМУВАННЯ

Навчально-методичний посібник
для самостійної роботи студентів другого (магістерського) рівня
денної, заочної та дуальної форм навчання
за спеціальністю 123 «Комп’ютерна інженерія»

Відповідальний за випуск проф. Заковоротний О. Ю.
Роботу до видання рекомендував проф. Заполовський М. Й.

В авторській редакції

План 2025 р., поз. 17
Гарнітура Times New Roman. Ум. друк. арк. 5.09

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, м. Харків, вул. Кирпичова, 2

Електронне видання

Кафедра
«Комп'ютерна
інженерія і
програмування»
НТУ «ХПІ»