https://doi.org/10.31891/2219-9365-2025-83-46
УДК 004.415.25:004.451

СЕМЕНЕЦЬ Олександр
Національний аерокосмічний університет ім. М.Є. Жуковського «Харківський авіаційний інститут»
o.y.semenets@csn.khai.edu
ТЕЦЬКИЙ Артем
Національний аерокосмічний університет ім. М.Є. Жуковського «Харківський авіаційний інститут»
https://orcid.org/0000-0003-1745-2452
a.tetskiy@csn.khai.edu

ДОСЛІДЖЕННЯ ВИНИКНЕННЯ ЛОГІЧНИХ ПОМИЛОК В РОЗРОБЦІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА МЕТОДИКИ ЇХ ВИЯВЛЕННЯ

Предметом вивчення в даній статті є вплив наявності логічних помилок на кібербезпеку веб-додатків.

Метою роботи є вивчення впливу логічних помилок на кібербезпеку веб-додатку, створення класифікації помилок, ознайомлення з методиками їх виявлення та побудова моделі логічної помилки.

Завдання: проаналізувати вплив логічних помилок на кібербезпеку веб-додатків, причини їх виникнення. Розглянути помилки, що з'являються на різних етапах розробки застосунків, класифікувати логічні помилки, привести методи протидії виникнення логічних помилок. Відповідно до поставлених завдань, були отримані наступні результати. Була розглянута каскадна модель розробки програмного забезпечення, на кожній з фаз котрої можуть виникати логічні помилки в архітектурі додатку. Для кращого розуміння шляхів виявлення логічних помилок була створена модель і введена класифікація. Наведено рекомендації і інструменти по виявленню похибок на кожній фазі.

Висновки: Виявлення логічних помилок в програмному забезпечені є досить складною і недостатньо вивченою проблемою. Отримані результати дозволяють краще зрозуміти природу логічної помилки, роблячи акцент на всі фази розробки програмного забезпечення. Наведено рекомендації з приводу методик і інструментів для виявлення логічних помилок.

Ключові слова: розробка програмного забезпечення, логічні помилки, кібербезпека, веб-додатки.

SEMENETS Oleksandr, TETSKYI Artem
National Aerospace University "Kharkiv Aviation Institute"

STUDY OF THE OCCURRENCE OF LOGICAL ERRORS IN SOFTWARE DEVELOPMENT AND METHODS FOR THEIR DETECTION

The study presented in this paper focuses on the phenomenon of logical errors in software development and their impact on the cybersecurity of web applications. Logical errors represent one of the most elusive categories of software defects, since they are not related to the technical correctness of code syntax or compilation issues, but rather to the incorrect realization of the intended logic of the system. Unlike typical coding errors, logical flaws may remain undetected through conventional debugging or testing procedures, yet they often lead to critical vulnerabilities that can be exploited by attackers. This makes their identification and prevention a matter of significant importance for secure and reliable software engineering.

The aim of the work is to examine the influence of logical errors on web application cybersecurity, to develop a classification system of such errors, to analyze their occurrence across different phases of the software development lifecycle, and to explore methodologies and tools that can improve their detection. The study also proposes a conceptual model of a logical error that provides a structured understanding of how such flaws emerge and propagate in application architectures.

The research objectives include: identifying the most common sources of logical errors in the software development process; analyzing their impact on system architecture, data processing, and user interaction; classifying errors based on their origin and manifestation; and considering methodological approaches for their detection and mitigation. Special attention is paid to the cascade (waterfall) model of software development, where logical flaws may appear at each phase — from requirements analysis and system design to implementation, testing, and maintenance. By introducing a classification framework and error model, the paper contributes to a more systematic approach to understanding and handling logical errors.

The results of the study highlight that logical errors not only affect the reliability and stability of software systems, but also play a critical role in weakening the cybersecurity posture of web applications. Logical vulnerabilities can open paths for unauthorized access, data leakage, privilege escalation, or violation of business logic, which attackers often exploit. Therefore, their timely detection is essential for both software quality assurance and cyber defense. Based on the conducted analysis, the paper provides recommendations on methodological practices and specialized tools that can assist developers, testers, and security analysts in identifying logical errors at different stages of development.

Logical error detection remains a complex and insufficiently studied problem in modern software engineering. The results of this research contribute to a deeper understanding of the nature and lifecycle of logical flaws, providing a foundation for further methodological development in this field. The findings emphasize that considering logical errors throughout the entire software development process allows for more resilient application design and enhances cybersecurity protection.

Keywords: software development, logical errors, cybersecurity, web applications.

Стаття надійшла до редакції / Received 16.07.2025
Прийнята до друку / Accepted 14.08.2025

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

На відміну від синтаксичних помилок, логічні помилки не завжди призводять до негайного припинення роботи програмного забезпечення, однак вони спричиняють некоректне виконання функціоналу або неправильну обробку даних. Це, у свою чергу, створює потенційні умови для експлуатації вразливостей або порушення безпеки. Виявлення логічних помилок на етапах тестування або під час експлуатації системи є суттєво ускладненим через відсутність явних ознак порушення роботи. У контексті критичних галузей, таких як медичні системи, авіаційне програмне забезпечення чи фінансові сервіси, наслідки таких помилок можуть бути надзвичайно серйозними - від значних фінансових збитків до безпосередньої загрози життю користувачів.

Оскільки проєктування логіки та реалізація алгоритмічної складової відбувається переважно на етапі розробки, саме ця фаза є найбільш вразливою до виникнення логічних помилок. Вони часто є наслідком недостатнього опрацювання вимог, їх неправильної інтерпретації, помилок у проектних рішеннях або відсутності належного контролю. З метою мінімізації ризиків, пов'язаних із логічними помилками, доцільним є застосування комплексного підходу, що включає формалізацію вимог, детальне проєктування, рецензування коду, впровадження практик захищеного програмування та системне тестування.

Етапи розробки програмного забезпечення

З метою абстрактного аналізу етапів розробки програмного забезпечення, на яких потенційно можуть виникати логічні помилки, у цій роботі використано спрощену каскадну (waterfall) модель розробки. Методологія Waterfall, запропонована доктором Вінстоном В. Ройсом у 1970 році, є послідовним процесом проектування, що широко застосовується в галузі розробки програмного забезпечення та інженерних продуктів. Вона передбачає поетапне проходження проекту через низку логічно впорядкованих фаз - від початкового планування до фінального впровадження - подібно до руху води у водоспаді [1].

Застосування цієї моделі дозволяє репрезентувати процес розробки як чітку послідовність фаз, у межах яких можливо виокремити критичні точки виникнення логічних помилок. Такий підхід сприяє структурованому аналізу потенційних ризиків і дозволяє акцентувати увагу на етапах, що потребують особливої уваги з погляду забезпечення коректності логіки програмного забезпечення.

Рис. 1. Каскадна модель (waterfall model) розробки програмного забезпечення

Причини виникнення логічних помилок в фазі "Затвердження вимог"

Це початковий етап планування, під час якого команда розробки збирає максимально повну інформацію для забезпечення успішної реалізації проекту. [1] Він вимагає ретельного попереднього аналізу та значного обсягу підготовчої роботи. На цьому етапі можлива недостатня або некоректна формалізація вимог з боку замовника [2], що може призвести до неповного охоплення необхідного функціоналу.

Не врахування специфічних сценаріїв використання або помилки в інтерпретації потреб користувача здатні спричинити логічні помилки, які проявляться вже під час реалізації або тестування. Крім того, на цьому

етапі можуть з'явитися суперечливі або неоднозначні вимоги, а їх некоректне об'єднання в єдину проєктну специфікацію створює ризик логічних помилок у подальшій архітектурі системи.

Відсутність повного розуміння предметної області або недостатньо детальне документування вимог [2] є одним із критичних чинників, що ускладнюють забезпечення логічної цілісності програмного забезпечення. Обмежена або неефективна комунікація між розробниками, системними аналітиками та замовниками також може призвести до хибного трактування вимог, що суттєво підвищує ймовірність помилок логіки на наступних етапах розробки.

Причини виникнення логічних помилок в фазі "Проектування"

Етап, на якому команда проекту визначає технічні аспекти реалізації програмного забезпечення, зокрема вибір апаратного забезпечення, мов програмування, засобів модульного тестування, типів інтерфейсів користувача та інших архітектурних параметрів. Ця фаза розробки є критично важливою для забезпечення відповідності майбутнього програмного продукту встановленим функціональним вимогам та очікуваним показникам продуктивності.

Архітектурне проектування традиційно поділяється на два етапи: високорівневе (архітектурне) проектування та низькорівневе (детальне) проектування. [1] У межах високорівневого проектування формується загальна структура системи - визначаються основні компоненти, способи обміну інформацією між ними, а також взаємодія із зовнішніми системами. На етапі низькорівневого проектування деталізуються конкретні алгоритми, модулі та механізми обробки даних, які реалізують функціональність кожного компонента.

Помилки, допущені на цьому етапі - зокрема в розумінні взаємодії між компонентами або з користувачем - можуть спричинити порушення логіки функціонування системи. Невірно визначені інтерфейси взаємодії, некоректна інтеграція з базами даних або сторонніми сервісами, а також неправильне структурування логіки обробки запитів можуть зумовити логічні збої, які виявляються лише на пізніших етапах розробки або в процесі експлуатації.

Причини виникнення логічних помилок в фазі "Розробка"

На цьому етапі команда розпочинає безпосередню реалізацію програмного забезпечення відповідно до специфікацій, сформульованих у фазі визначення вимог та детально описаних у фазі проектування системи [1]. Основними орієнтирами для реалізації є документація з вимог (створена на першому етапі) та технічний дизайн (розроблений на другому етапі).

Під час фази розробки розробники створюють програмні модулі та проводять їх модульне тестування з метою перевірки відповідності функціоналу визначеним вимогам. Проте на цьому етапі можуть виникати логічні помилки, зумовлені неоднозначною інтерпретацією вимог [2], особливо у випадках, коли вони недостатньо формалізовані або містять прогалини.

Крім того, типовими джерелами логічних помилок на етапі розробки є відсутність належної валідації вхідних даних, некоректна обробка виняткових ситуацій, помилки у реалізації алгоритмів, а також використання застарілих технологій [2], що можуть призвести до відхилення від очікуваної бізнес-логіки або порушення внутрішньої цілісності системи.

Причини виникнення логічних помилок в фазі "Тестування"

Це етап, на якому команда розробки передає програмний продукт команді забезпечення якості для проведення всебічного тестування [1]. Основною метою є виявлення дефектів або помилок, які необхідно усунути до етапу розгортання системи.

Тестувальники здійснюють перевірку програмного забезпечення відповідно до технічних та функціональних вимог, використовуючи різні типи тестів (функціональні, інтеграційні, регресійні тощо). Усі виявлені відхилення мають бути ретельно задокументовані із зазначенням умов їхнього виникнення. Це сприяє підвищенню ефективності процесу виправлення помилок, оскільки дозволяє іншим розробникам швидко ідентифікувати та відтворити проблему.

Однак навіть за умов якісного тестування не всі логічні помилки можуть бути виявлені. Це може бути зумовлено як обмеженим охопленням тест-кейсів [2], так і часовими обмеженнями, через які не вдається перевірити всі можливі сценарії використання системи. Внаслідок цього частина логічних дефектів може залишатися невиявленою до моменту експлуатації програмного забезпечення.

Причини виникнення логічних помилок в фазі "Підтримка"

Після розгортання програмного забезпечення на етапі його експлуатації можуть виникати нові помилки або з'являється потреба в оновленні функціоналу. Фаза підтримки передбачає постійний моніторинг, усунення дефектів, удосконалення системи та адаптацію до змін у середовищі її функціонування [1].

Внесення змін до програмного забезпечення, зокрема додавання нового функціоналу або оновлення зовнішніх залежностей [2], може призводити до порушення вже реалізованої логіки системи. Це особливо актуально у випадках, коли модифікації не супроводжуються повним регресійним тестуванням. Нові компоненти можуть вступати у конфлікт з існуючими механізмами, змінювати передумови їх роботи або порушувати логіку взаємодії між модулями.

Крім того, несвоєчасне оновлення зовнішніх бібліотек і залежностей може стати причиною логічних помилок, спричинених несумісністю або змінами в поведінці зовнішніх компонентів. Таким чином, фаза супроводу є критично важливою для забезпечення цілісності логіки програмного забезпечення впродовж усього життєвого циклу системи.

Модель логічної помилки

Для кращого розуміння джерел і природи логічних помилок у програмному забезпеченні доцільно враховувати стандартизовану класифікацію, зокрема **MITRE CWE-840: Business Logic Errors [3]**, яка описує помилки в реалізації бізнес-логіки, що не забезпечують належної відповідності очікуваній поведінці системи.

Вхідні дані які використовує веб додаток

Базуючись на вводі інформації в веб додаток, можна перевірити

- Дані з форм користувача, сторонніх сервісів або авторизації, що можуть бути неправильно оброблені через відсутність перевірок граничних умов (CWE-840.1) або неочікувані переходи станів (CWE-841);
- Зловмисник може змінити дані запиту, обходячи логіку перевірки.

Вихідні дані

Логічні помилки можуть призвести до неправильної відповіді API, некоректного форматування або трансформації даних для інтерфейсу користувача або доступу до конфіденційних даних.

Обробка даних

Стадія обробки даних вимагає суворої перевірки вхідних значень і реалізації бізнес-логіки. Основними джерелами логічних помилок на цьому етапі можуть бути:

- Неправильна авторизація. (CWE-843);
- Непослідовність правил, або конфлікт між частинами системи щодо логіки доступу або обмежень. (CWE-840.2);
- CWE-842 - система приймає рішення на основі ненадійного вводу.

Інтерфейс користувача (UI)

Треба перевірити навігацію в додатку, чи має користувач доступ до всіх сторінок доступних для данного користувача, та як інформація відображається на сторінках.

Вразливості логіки можуть виявитися у навігації, випадки коли користувач отримує доступ до недозволених сторінок, якщо відсутні перевірки ролей або станів.

Логування та зворотній зв'язок

В рамках цього етапу, треба перевірити чи можуть користувачі залишити відгук про проблеми в роботі додатку, та чи в повному обсязі забезпечений запис подій що викликали помилку в роботі додатку.

Відсутність належного логування дій користувача або зворотного зв'язку про помилки може ускладнити виявлення логічних вразливостей.

Класифікація логічних помилок

Для кращого розуміння по виявленню та виправленню логічних помилок їх треба **класифікувати**.

У статті [4] автори провели ручний аналіз коду створеного програмістами-початківцями та класифікували помилки таким чином:

Алгоритмічні - це помилки в абстрактних термінах, що не обов'язково вказує на фундаментальний брак знань у програмуванні.

Неправильне тлумачення - помилки які виникають внаслідок неправильного розуміння до вимог логіки.

Неправильне розуміння - логічна помилка, яка відображає фундаментальну відсутність знань з програмування.

Також автори виділили 2 метрики, при вимірюванні складності помилок.

Час на виправлення помилки - це час, що минув між першим виправленням логічної помилки та успішним вирішенням задачі.

Повторні помилки - наявність одних й тих самих помилок.

У дослідженні [5], авторами запропоновано поділ логічних помилок на такі категорії:

- **Помилки умовних виразів** - некоректне використання умов, що призводить до неправильного виконання логіки програми;
- **Помилки ітераційних конструкцій** - помилки в циклах, які можуть спричинити нескінченні або неправильні ітерації;
- **Помилки обчислень** - неправильні математичні операції або обчислення;
- **Помилки у викликах функцій** - використання функцій з неправильними аргументами або в неправильному контексті;
- **Помилки в обробці даних** - некоректна маніпуляція з даними, що може призвести до неправильних результатів.

Враховуються взаємозв'язки між помилками, це дозволяє ефективніше їх виявляти та виправляти.

Автори у дослідженні [6] представили класифікацію причин збоїв у програмному забезпеченні, що включає:

- **Дефекти в коді** - помилки, які виникають безпосередньо в коді програми;
- **Умови, що активують помилки** - специфічні стани або вхідні дані, які призводять до появи помилки;
- **Стан помилки** - результат активації помилки, який може призвести до збою системи.

Використання даної класифікації дозволяє зрозуміти, причини виникнення логічних помилок, сприяє ефективному виявленню та запобіганню.

У дослідженні [7] логічні помилки класифікуються за їх джерелом:

- **Алгоритмічні помилки** - виникають через неправильне розуміння або реалізацію алгоритмів;
- **Помилки тлумачення** - результат неправильного розуміння вимог або специфікацій;
- **Концептуальні помилки** - свідчать про глибоке нерозуміння основ програмування або принципів побудови алгоритмів.

Класифікація допомагає виявити основні причини помилок та розробити стратегії для їх запобігання.

Khan Academy у своїй статті [8] пропонує класифікацію помилок за контекстом їх виникнення.

- **Синтаксичні помилки** - виникають через порушення правил синтаксису мови програмування;
- **Помилки часу виконання** - виникають під час виконання програми, наприклад, ділення на нуль;
- **Логічні помилки** - програма виконується без збоїв, але результат не відповідає очікуванням.

Розділення помилок за часом виникнення та контекстом сприяє більш ефективному їх виправленню.

В моєму дослідженні я пропоную розширити класифікацію взявши за основу причини виникнення логічних помилок в каскадній моделі розробки ПЗ та модель логічної помилки.

Семантичні - коли логіка обчислень не відповідає очікуванням.

Помилки бізнес-логіки - не правильна реалізація вимог функціональності.

Помилки вхідних даних - дані, що ввів користувач недостатньо перевіряються або не перевіряються взагалі.

Критичні - ті що призводять до повного збою роботи додатка, або блокують його.

Неочікувані - коли результат виконання обчислення не відповідають очікуванням.

Помилки в авторизації - недоліки в перевірці прав доступу.

Помилки в управлінні сесіями - помилки зберігання, або перевірок роботи сесій.

Помилки в кешуванні - неправильна реалізація управління кешуванням.

Помилки в роботі з базами даних - неправильні запити, або проблеми транзакцій.

Клієнтські - проблеми, що виникають в браузері.

Серверні - проблеми, що виникають під час обробки запитів на сервері.

Вразливі модулі - використання не перевірених бібліотек.

Таблиця 1

Класифікація логічних помилок

Критерій	**Тип помилки**
за типом	семантичні, помилки бізнес логіки
за стадією взаємодії з користувачем	помилки вхідних даних та відображення інформації
за впливом на роботу програми	критичні, неочікувані
за рівнем доступу	помилки авторизації, в управлінні сесіями
за обробкою даних	помилки в кешуванні, в роботі з базами даних
за джерелом виникнення	клієнтські, серверні, вразливі модулі
за часом виявлення	на етапі розробки, на етапі експлуатації

Методи виявлення логічних помилок

Інженер з кібербезпеки може виявляти логічні помилки на різних етапах розробки шляхом ретельного аналізу вимог, дизайну, коду, тестування та моніторингу системи. Чим раніше будуть виявлені помилки, тим

легше їх виправити, тим менше витрат буде на виправлення, а також знижується ризик серйозних проблем на етапі впровадження чи експлуатації.

На етапі планування та проектування програмного забезпечення інженер з кібербезпеки має на меті оцінити загрози та визначити вразливості в архітектурі, щоб забезпечити належний рівень безпеки на всіх етапах розробки.

В статті [9] автори дослідили використання SLA-generator, Microsoft Threat Modeling Tool та OWASP Threat Dragon, кожен з цих інструментів має свої особливості в використані для різних завдань. Авторами був зроблений висновок, що **OWASP Threat Dragon** - чудовий для швидкого та легкого моделювання загроз, буде найкращими варіантом для моделювання загроз на етапі проектування.

Загрози та вразливості виявляються під час життєвого циклу розробки програмного забезпечення, інженер з кібербезпеки може за допомогою сучасних технік статичного та динамічного аналізу програм виявити їх. Ці техніки виявилися ефективними для виявлення відомих заздалегідь дефектів, але вони не достатньо охоплюють таку проблему, як виявлення логічних помилок. [10]

У цій статті автори розглянули такі методи:

- статичний аналіз коду - це аналіз, що включає перевірку вихідного коду програми без її виконання. Метою є пошук патернів або аномалій у коді. Метод допомагає виявити помилки, які можуть бути наслідком неправильного підходу до вирішення задачі або некоректних логічних конструкцій;
- динамічне профілювання - аналіз який потребує виконання програми з набором вхідних даних, і на основі отриманих результатів виявляються логічні помилки. Цей метод дозволяє спостерігати за поведінкою програми під час її виконання та виявляти проблеми, які можуть виникати в процесі її роботи з конкретними даними.

В статті [11] авторка оцінила ефективность різних інструментів SAST, які використовуються для автоматизованого виявлення вразливостей на етапі розробки.

SonarQube є гарним вибором, якщо потрібно швидко перевіряти код на загальні помилки. Також авторка прийшла до висновку, що для більш глибокого аналізу помилок в бізнес-логіці, **Checkmarx** або **Fortify** будуть більш ефективними інструментами.

Під час фази тестування інженер з кібербезпеки має перевірити функціональність програми та її поведінку в різних умовах, щоб знайти логічні помилки, які можуть виникнути при взаємодії з іншими системами.

В статті [12] автор порівнюють інструменти **Burp Suite**, **OWASP ZAP (Zed Attack Proxy)** і **Wireshark** і приходять до висновку, що OWASP ZAP добре підходить для DevOps/DevSecOps, так як він легше інтегрується через API і має хорошу підтримку. У той час як Burp Suite більше орієнтований на фактичну оцінку вразливостей і проведення пенетраційного тестування веб-додатків. Burp Pro коштує 399 доларів США за користувача на рік, тоді як OWASP ZAP є безкоштовним і відкритим проектом під ліцензією Apache 2.0.

Wireshark більше підходить для аналізу мережевого трафіку, але він не є спеціалізованим інструментом для виявлення логічних помилок на рівні додатка. Проте, **Wireshark** може допомогти виявити проблеми, що виникають під час мережевої комунікації, які можуть свідчити про логічні помилки в тому, як програма обробляє запити чи відповіді.

В типі запропонованої мною класифікації за джерелом виникнення я вказав використання вразливих модулів враховуючи npm-модулі, котрі встановлюються при створенні SPA веб-додатків або серверного коду на node.js. Ці модулі можуть містити шкідливий код, який буде впливати на безпеку веб-додатку. Зловмисники можуть публікувати пакети в **npm registry**, які виглядають як корисні утиліти чи бібліотеки, але насправді містять шкідливий код. Цей код може спричиняти:

- Крадіжку конфіденційних даних (паролів, токенів, особистої інформації);
- Виконання команд на сервері;
- Зниження рівня безпеки або встановлення бекдорів;
- Поширення вірусів і троянів.

Іноді опубліковані пакети не містять шкідливого коду на момент початкової публікації, але можуть бути змінені через деякий час. Це може трапитися з широко розповсюдженими пакетами, коли зловмисник отримає доступ до цього пакету і внесе шкідливий код в більш пізні версії.

Також під цю категорію потрапляє використання пакетів, які використовують застарілі або небезпечні залежності, що можуть ставати вразливими до атак. Зловмисники можуть впровадити вразливості в залежності, які будуть використовуватися під час установки пакету, що може призвести до виконання довільного коду на машині користувача.

Автори [13] моніторять популярні репозиторії відкритого програмного забезпечення (OSS) за допомогою своїх автоматизованих інструментів для запобігання потенційним загрозам безпеки та повідомляють про будь-які виявленні вразливості або шкідливі пакети невідповідним технічним фахівцем та спільноті розробників. Нещодавно ними було виявлено 25 шкідливих пакетів у репозиторії npm. Новачки-хакери продовжують використовувати npm з метою атак, оскільки зусилля для розробки та публікації

шкідливого пакета є мінімальними. Прогнозується, що ця тенденція буде тільки зростати, оскільки щодня сканери npm виявляють десятки нових шкідливих пакетів.

Для виявлення потенційно проблемних модулів розробникам треба використовувати команду npm audit. npm audit - це вбудований інструмент командного рядка, який дозволяє аналізувати залежності проекту на наявність відомих вразливостей в залежностях та надає рекомендації щодо їх усунення. Він використовує дані з публічної бази даних вразливостей Node Security Platform (NSP), яка включає інформацію про пакети з відомими вразливостями.

Package	Payload	Infection Method
node-colors-sync	Discord token stealer	Masquerading (colors)
color-self	Discord token stealer	Masquerading (colors)
color-self-2	Discord token stealer	Masquerading (colors)
wafer-text	Environment variable stealer	Typosquatting (wafer-*)
wafer-countdown	Environment variable stealer	Typosquatting (wafer-*)
wafer-template	Environment variable stealer	Typosquatting (wafer-*)
wafer-darla	Environment variable stealer	Typosquatting (wafer-*)
lemaaa	Discord token stealer	Hidden functionality
adv-discord-utility	Discord token stealer	Unknown
tools-for-discord	Discord token stealer	Unknown
mynewpkg	Environment variable stealer	Unknown
purple-bitch	Discord token stealer	Unknown
purple-bitchs	Discord token stealer	Unknown
noblox.js-addons	Discord token stealer	Masquerading (noblox.js)
kakakaakaaa11aa	Connectback shell	Unknown
markedjs	Python remote code injector	Masquerading (marked)
crypto-standarts	Python remote code injector	Masquerading (crypto-js)
discord-selfbot-tools	Discord token stealer	Masquerading (discord.js)
discord.js-aployscript-v11	Discord token stealer	Masquerading (discord.js)
discord.js-selfbot-aployscript	Discord token stealer	Masquerading (discord.js)
discord.js-selfbot-aployed	Discord token stealer	Masquerading (discord.js)
discord.js-discord-selfbot-v4	Discord token stealer	Masquerading (discord.js)
colors-beta	Discord token stealer	Masquerading (colors)
vera.js	Discord token stealer	Unknown
discord-protection	Discord token stealer	Unknown

Рис. 2. Досліджені модулі та вразливості виявленні в них

Окрім npm audit, на етапі розробки варто розглянути впровадження більш надійної платформи для безпеки додатків, яка орієнтована на розробників і надає їм інструменти, необхідні для захисту їхніх додатків. Snyk - це платформа безпеки, орієнтована на розробників, яка в першу чергу створена з урахуванням робочих процесів розробників. [14]

Також для запобігання використання вразливих пакетів можна використовувати бази CVE (Common Vulnerabilities and Exposures), NVD (National Vulnerability Database) та Node Security Platform.

Після розгортання програмного забезпечення важливо продовжувати моніторинг і перевірку роботи системи, щоб виявляти логічні помилки, які можуть проявлятися під час реального використання.

На етапі підтримки інженер з кібербезпеки зазвичай займається моніторингом, аналізом і реагуванням на події безпеки та аномалії в роботі системи. Для виявлення логічних помилок в рамках цього етапу важливо мати інструменти, які дозволяють аналізувати великі об'єми логів, виявляти аномалії, відстежувати події, а також автоматизувати процеси реагування.

На етапі підтримки інженер з кібербезпеки зазвичай займається моніторингом, аналізом і реагуванням на події безпеки та аномалії в роботі системи. Для виявлення **логічних помилок** в рамках цього етапу важливо мати інструменти, які дозволяють аналізувати великі об'єми логів, виявляти аномалії, відстежувати та корелювати події, а також автоматизувати процеси реагування.

Компоненти стеку ELK - Elasticsearch, Logstash і Kibana - працюють разом, забезпечуючи гнучку та масштабовану платформу. ELK знаходить застосування в різних сферах, включаючи управління логами, безпеку та бізнес-аналітику. Відкритий код ELK, велика підтримка спільноти та ефективність з точки зору витрат роблять його привабливим вибором для багатьох організацій, що шукають надійні можливості для керування логами. [15]

Посібник OWASP Web Security Testing Guide (WSTG) [16] (розділ 4.10 "Business Logic Testing") надає практичні підходи до виявлення логічних вразливостей (logical vulnerabilities). Ці підходи відрізняються від традиційного технічного тестування тим, що вони зосереджені на контекстуальному розумінні бізнес-процесів, очікуваної поведінки системи та способів її обходу. До цих підходів можна віднести:

- тестування дозволів та обмежень. Під час данного тестування інженер виконує спроби виконати дії від імені іншого користувача та перевіряє обмеження за кількістю запитів;
- проводить ручне чи автоматизоване тестування;
- виконує дії котрі не передбачені під час проектування;
- аналізує відповіді при хибних запитах.

На відміну від технічних помилок (SQLi, XSS тощо), логічні вразливості не завжди можна виявити автоматичними сканерами. Вони залежать від контексту і того, як саме має працювати система.

У своїй роботі [17] крім вже розглянутих методик, автори рекомендують залучати користувачів під час розробки, для проведення тестування, та постійно підвищувати кваліфікацію розробників.

У роботі [18], автор розглядає різні помилки проектування, які можуть виникати на різних етапах розробки програмного забезпечення. Автор приділяє увагу використанню діаграм для візуалізації вимог та використання математичних моделей для опису вимог.

Також рекомендується застосувати рецензії коду та моделювання системи, для раннього виявлення потенційних проблем.

У своїй роботі С. Макконнелл "Code Complete" [19] розглядає практичний підхід до підвищення якості програмного забезпечення шляхом глибокого аналізу коду. Зокрема, автор акцентує увагу на застосуванні принципу "Проектування за контрактом", який передбачає чітке визначення формальних умов для кожного програмного модуля: передумов, що мають бути виконані перед викликом функції; після умов, які повинні бути істинними після її виконання; а також інваріантів, що залишаються незмінними протягом усього життєвого циклу модуля. Такий підхід сприяє підвищенню надійності та передбачуваності поведінки програмних компонентів.

ВИСНОВКИ З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

У подальшому буде розглянуто, як інженер з кібербезпеки може виявити логічні помилки, не маючи доступу до проекту. Це дуже складна задача, яку можна вирішити за допомогою різноманітних підходів та інструментів, орієнтуючись на аналіз поведінки системи в процесі її експлуатації, тестування та взаємодії з іншими компонентами. Можна використати Black-box testing для перевірки функціональності програми. Наступним етапом можна провести атаку на проникнення, щоб виявити вразливості такі як SQL injection, XML injection, або command injection, котрі можуть бути викликані наявністю логічних помилок в обробці запитів. Також гарною рекомендацією, щодо виявлення логічних помилок без доступу до вихідного коду є аналіз роботи API, так як багато сучасних програм взаємодіють через API. В нагоді може бути Postman. Postman - це безкоштовний інструмент для тестування API, який можна використовувати перевіряючи правильність відповідей.

References

1 Sarah Laoyan. Guide to waterfall methodology: Free template and examples. URL - https://asana.com/resources/waterfall-project-management-methodology;

2 Pressman R. S., Maxim B. R. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.);

3 MITRE CWE-840: *Business Logic Errors*. URL - https://cwe.mitre.org/data/definitions/840.html;

4 Andrew Ettles, Andrew Luxton-Reilly, Paul Denny. Common logic errors made by novice programmers. URL - https://dl.acm.org/doi/abs/10.1145/3160489.3160493;

5 Yanggyu Lee, Suchae Jeong, Jihie Kim. Improving LLM Classification of Logical Errors by Integrating Error Relationship into Prompts. URL -

https://www.researchgate.net/publication/381089186_Improving_LLM_Classification_of_Logical_Errors_by_Integrating_Error_Relationship_into_Prompts;

6 Lena Feinbube, Peter Tröger, Andreas Polze. The landscape of software failure cause models.
URL - https://www.researchgate.net/publication/301839557_The_landscape_of_software_failure_cause_models;

7 Zihe Zhou, Shijuan Wang, Yizhou Qian. Learning From Errors: Exploring the Effectiveness of Enhanced Error Messages in Learning to Program.
URL - https://www.researchgate.net/publication/356704967_Learning_From_Errors_Exploring_the_Effectiveness_of_Enhanced_Error_Messages_in_Learning_to_Program;

8 Vanessa Leite, Gustavo Soares, Fernando Castor. Detecting Logical Errors in Haskell.
URL - https://llp.dcc.ufmg.br/Publications/Art2021/2021-detectionErrorHaskell-Vanessa.pdf?utm_source=chatgpt.com;

9 Daniele Granata, Massimiliano Rak. Systematic analysis of automated threat modelling techniques: Comparison of open-source tools. URL - https://link.springer.com/article/10.1007/s11219-023-09634-4

10 George Stergiopoulos, Panagiotis Katsaros, Dimitris Gritzalis. Source code profiling and classification for automated detection of logical errors. URL - https://www.infosec.aueb.gr/Publications/PAS-2014%20Logical%20Error.pdf;

11 Alexandra de Barros Reigada. Generic SAST tool Comparer. URL - https://repositorium.sdum.uminho.pt/bitstream/1822/84173/1/Alexandra%20de%20Barros%20Reigada.pdf;

12 Ekene Joseph. Burp Suite vs OWASP ZAP — a Comparison series. URL - https://medium.com/@Ekenejoseph/burp-suite-vs-owasp-zap-a-comparison-series-8e34162c42e6;

13 Andrey Polkovnychenko, Shachar Menashe. Malware Civil War – Malicious npm Packages Targeting Malware Authors. URL - https://jfrog.com/blog/malware-civil-war-malicious-npm-packages-targeting-malware-authors

14 How to use npm audit. URL - https://www.nodejs-security.com/blog/how-to-use-npm-audit

15 Musmanqureshi. Exploring Logging Tools: Comparision of ELK with Splunk and Datadog & Deep Dive into ELK Stack. URL - https://medium.com/@musmanqureshi109/popular-logging-tools-used-in-devops-db5b18b85fb7;

16 OWASP Testing Guide. URL - https://owasp.org/www-project-web-security-testing-guide;

17 Pressman R. S., & Maxim B. R. Software Engineering: A Practitioner's Approach (8th ed.) - 2014;

18 Sommerville I. Software Engineering (10th ed.) - 2016;

19 McConnell S. Code Complete (2nd ed.) Microsoft Press. - 2004;

20 Dovetail Editorial Team. What is the Waterfall methodology in project management? URL - https://dovetail.com/product-development/waterfall-project-management

21 Indeed Editorial Team. A Complete Guide to the Waterfall Methodology. URL - https://www.indeed.com/career-advice/career-development/waterfall-methodology

22 Patrick Icasas. What are the Six Phases of Waterfall Project Management? URL - https://birdviewpsa.com/blog/six-phases-waterfall-project-management