НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
Завідувач кафедри

_____ Дмитро ЛАНДЕ
(підпис)

«_____»_____2022 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи, технології та математичні методи кібербезпеки»
спеціальності 125 «Кібербезпека»

на тему: **Методи та засоби тестування захищеності REST API. Фазінг REST API**

Виконав (-ла): здобувач вищої освіти **IV** курсу, групи ФБ-83
(шифр групи)

Гах Валерій Олександрович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник к.т.н., доцент кафедри Інформаційної безпеки

Коломицев Михайло Володимирович
(посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім'я, по батькові) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Здобувач вищої освіти _____
(підпис)

Київ – 2022 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Дмитро ЛАНДЕ

(підпис) (ініціали, прізвище)

«___»_____20__ р.

ЗАВДАННЯ
на дипломну роботу студенту

Гах Валерій Олександрович
(прізвище, ім'я, по батькові)

1. Тема роботи:

Методи та засоби тестування захищеності REST API. Фазінг REST API,

керівник роботи: к.т.н., доцент кафедри Інформаційної безпеки
Коломицев Михайло Володимирович,

затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Строк подання студентом проекту (роботи) **13 червня 2022 р.**

3. Вихідні дані до проекту (роботи)

1) Література на тему фазінгу в тестуванні веб-застосунків
2) Відомості про типові вразливості REST API
3) Відомості про існуючі інструменти фазінгу веб-застосунків

4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити)

Аналіз існуючих рішень фазінгу веб-застосунків, зокрема фазінгу REST API. Розробка власного методу фазінгу REST API на основі відомого переліку вразливостей REST API. Порівняльний аналіз практичного застосування існуючих інструментів фазінгу REST API, в тому числі із застосуванням розробленої методики.

5. Перелік графічного (ілюстративного) матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо)

Презентація

6. Дата видачі завдання

Календарний план

№ з/п	Назва етапів роботи та питань, які мають бути розроблені відповідно до завдання	Термін виконання	Позначки керівника про виконання завдань
1	Вибір напрямку роботи.	01.09.2021 – 24.09.2021	
2	Аналіз тематичної літератури, звуження напряму роботи.	25.09.2021 – 13.10.2021	
3	Створення плану роботи, погодження плану з керівником.	14.10.2021 – 02.02.2022	
4	Ознайомлення з вихідними даними до роботи.	03.02.2022 – 01.04.2022	
5	Аналіз існуючих підходів до тестування захищеності веб-застосунків, розгляд існуючих підходів застосування фазінгу.	02.04.2022 – 14.04.2022	
6	Аналіз типових вразливостей і багів REST API.	15.04.2022 – 21.04.2022	
7	Вибір інструментів фазінгу для практичного дослідження.	22.04.2022 – 24.04.2022	
8	Розробка методики фазінгу REST API	25.04.2021 – 01.05.2021	

9	Переддипломна практика.	02.05.2022 – 24.05.2022	
10	Проведення практичного дослідження фазінгу REST API. Проведення порівняльного аналізу застосованих підходів до фазінгу.	25.05.2022 – 12.06.2022	
11	Передзахист дипломної роботи.	13.06.2022	
12	Доопрацювання дипломної роботи, створення презентації.	14.06.2022 – 19.06.2022	
13	Захист дипломної роботи.	20.06.2022	

Здобувач вищої освіти _____ **Валерій ГАХ**
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ **Михайло Коломицев**
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Обсяг роботи 67 сторінки, 6 ілюстрацій, 7 таблиць, 18 джерел літератури та посилань на матеріали.

RESTful веб-застосунки та хмарні сервіси, що імплементують REST API є дуже цінними компонентами бізнесів сучасності. Методи та засоби тестування захищеності таких програмних рішень має невпинно розвиватися у відповідь постійному вдосконаленню атак з боку зловмисників. Тестування REST API вимагає специфічних засобів та підходів, зокрема фазінг REST API має певні тонкощі застосування на практиці. Фазінг API є незамінним методом тестування якості та захищеності результуючого програмного рішення, тому вимагає більше уваги до себе та до вдосконалення існуючих інструментів, що імплементують фазінг.

Об'єктом дослідження є баги і вразливості в імплементації REST API веб-застосунків.

Предметом дослідження є існуючі інструменти фазінгу REST API та підходи з їх використанням при пошуку багів REST API.

Метою дослідження є порівняння існуючих підходів до фазінгу REST API та розробка власного методу тестування фазінгом API, проведення порівняльного аналізу застосування розробленого методу та існуючих інструментів на практиці.

Ключові слова: тестування захищеності веб-застосунків, REST API, фазінг, Restler fuzzer, Radamsa.

ABSTRACT

The work contains 67 pages, 6 illustrations, 7 tables, 18 references to materials and sources of literature.

RESTful web applications and cloud services that implement the REST API are very valuable components of today's business. Methods and tools for testing the security of such software solutions must be constantly evolving in response to the constant improvement of attacks by malicious actors. REST API testing requires specific tools and approaches, in particular, REST API fuzzing has certain subtleties in practice. Fuzzing the APIs is an indispensable method of testing the quality and security of the resulting software solution, so it requires more attention to themselves and to improve existing tools that implement phasing.

The object of the study is the bugs and vulnerabilities in the implementation of the REST API web applications.

The subject of the research is the existing REST API fuzzing tools and approaches with their use in finding REST API bugs.

The aim of the research is to compare the existing approaches to REST API fuzzing and to develop one's own testing method by API phasing, to conduct a comparative analysis of the application of the developed method and existing tools in practice.

Keywords: Web Application Security Testing, REST API, Fuzzing, Restler fuzzer, Radamsa.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

"Точка входу" в ПЗ – окрема функція ПЗ, яка безпосередньо приймає вхідні дані від користувача.

ВСТУП

Сучасні бізнеси зазвичай містять в собі веб-застосунки, якими користуються їх клієнти. Такі системи обробки веб-контенту, що часто створені на швидку і з економією на заходах безпеки, містять вразливості, що зрештою призводять до масових крадіжок даних та відмови від обслуговування. Для багатьох бізнесів створення і супровід цих систем, як вторинна задача після налагодження основних бізнес-процесів, має бути максимально дешеве і швидке а також не менш захищене від можливих атак.

Об'єктом дослідження є процес тестування захищеності REST API, зокрема тонкощі методу фазінгу API.

Предметом дослідження є існуючі інструменти фазінгу REST API та підходи з їх використанням при пошуку багів REST API.

Метою дослідження є порівняння існуючих підходів до фазінгу REST API, розробка вдосконаленої методики фазінгу із застосуванням існуючих інструментів ручного фазінгу, проведення порівняльного аналізу інструментів за переліком критеріїв.

Завдання на дослідження для досягнення поставленої мети:

1) Ознайомитись із літературними джерелами теорії характерних вразливостей та тестування захисту веб-застосунків, зокрема RESTful застосунків.

2) Визначити популярні існуючі підходи та стеки інструментів тестування REST API, зокрема інструментів, що підтримують фазінг API.

3) Розробити власний метод фазінгу API на основі даних про поширені баги і вразливості REST.

4) Розгорнути навмисно вразливий веб-застосунок із REST API, встановити стеки інструментів тестування.

5) Провести фазінг цільового застосунку за власним підходом та вже існуючим за допомогою визначених інструментів, отримати результати у вигляді знайдених багів у REST API.

6) Оцінити застосовність та практичність визначених підходів та інструментів окремо та порівняти їх за переліком критеріїв.

Практична частина дослідження полягає у проведенні процесів тестування за допомогою обраних існуючих програмних рішень на вразливому веб-застосунку із усіма затребуваними попередніми кроками, порівняння результатів окремо для пари програмних рішень та для результатів, отриманих дослідником із даними про наявні вразливості від розробника піддослідного застосунку.

1 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ДО ТЕСТУВАННЯ ЗАХИЩЕНОСТІ ВЕБ-ЗАСТОСУНКІВ

1.1 Типи та підходи до тестування захищеності коду веб-застосунків

У веб-застосунках існують характерні вразливості і ризики, на зменшення яких і направлене тестування захисту, специфічне для веб-застосунків. Ці вразливості визначають підходи до тестування коду веб-застосунків – вони взаємопов'язані. На основі відкритих джерел інформації про найпоширеніші атаки [1] на веб-застосунки та відомої класифікації вразливостей [2], характерних для цих сервісів, сформуємо таблицю відповідності атак на веб-застосунки та використовуваних ними вразливостей:

Таблиця 1.1 – Атаки і використовувані в них вразливості у веб-застосунках

Вразливості	Класи атак, що їх експлуатують
Надлишковий функціонал веб-застосунку, надлишкові об'єкти в середовищі роботи веб-сервісу Відсутність оновлень, наявність ПЗ з закінченим терміном підтримки	Майже усі атаки використовують інформацію про веб-застосунок, отриману завдяки цим вразливостям
Публічно доступна критична інформація, що оброблюється у веб- застосунку і вимагає захисту	Крадіжка облікового запису, підвищення привілеїв

Продовження таблиці 1.1

Вразливості	Класи атак, що їх експлуатують
Вразливості механізмів обміну та зберігання критичної інформації на клієнтській і серверній частині	Крадіжка облікового запису, підвищення привілеїв
Слабкість механізмів автентифікації/авторизації	Перерахування облікових записів, підвищення привілеїв, крадіжка облікового запису, spoofing, прямий доступ до ресурсів, запуск довільного коду, DoS
Слабка криптографія у використанні веб-застосунком	Злам хешів, цифрових підписів, розшифрування критичних даних
Слабкі механізми перевірки вхідних даних від недовірених джерел	Injection-атаки, XSS, атаки на механізми десеріалізації, DoS
Слабкий механізм реалізації сесій	Крадіжка сесії, підвищення привілеїв, CSRF
Вразливості механізмів завантаження файлів на веб-сервер, відсутність перевірок даних від недовірених джерел	Code-injection атаки, XSS-атаки, запуск довільного коду, DoS
Вразливості механізмів обробки форматів даних	XML injection, XXE, XPath injection, XQuery injection атаки, DoS
Вразливості механізмів виділення ресурсів на обробку клієнтських запитів	DoS

Класи вразливостей визначають підходи і засоби тестування веб-застосунків, в той час як статистичні дані про успішні атаки на веб-застосунки визначають потреби в покращенні мір захисту і тестування навколо вразливостей, що все частіше експлуатуються зловмисниками.

Навіть для перевірки захисту від одного класу атак або для перевірки на наявність певної вразливості існують різні підходи та інструменти різної витратності, складності в застосуванні і ефективності. Тестування захищеності веб-застосунку має бути інтегроване в процес розробки так як вразливості, що з'являються на ранніх стадіях життєвого циклу програмного продукту, вимагатимуть непомірних витрат для їх усунення на більш пізніх етапах. Такі вразливості як помилки дизайну застосунку чи вразливості його бізнес-логіки означатимуть однозначну незахищеність застосунку від відповідних атак, а для усунення цих вразливостей необхідно розпочинати процес розробки спочатку. OWASP Web Security Testing Guide[3] – приклад професійного посібника з вичерпною інформацією про принципи тестування захищеності веб-застосунків на різних стадіях їх життєвого циклу. Також в даному ресурсі висвітлено підходи і методи тестування захищеності окремих функціональних компонентів веб-застосунків, завдяки чому розробники і спеціалісти з безпеки використовують його і як практичний посібник з тестування веб- застосунків.

Велике різноманіття методів тестування захищеності концептуально можна об'єднати в наступні типи тестування:

- Сканування на наявність вразливостей(Vulnerability Scanning) – автоматизований процес перевірки цільової системи на наявність відомих вразливостей в цілому, або окремих її компонентів.
- Аналіз ризиків(Risk Assessment) – ручний процес оцінки ймовірності настання загрози і ймовірних збитків в разі її реалізації в результаті якого приймається рішення про імплементацію тих чи інших заходів безпеки в певних місцях цільової системи. Процес оцінки заснований на урахуванні потенційних загроз винятково цінним компонентам системи і направлений на створення економічно ефективної стратегії забезпечення захисту.

- Аудит безпеки(Security Audit) – процес перевірки відповідності будь-яких процесів всередині цільової системи визначеним стандартам і практикам безпеки.
- Тестування безпеки застосунку(Web Application Security Testing) – за аналогією тестування якості застосунку дане тестування призначене для виявлення помилок коду і слабкостей реалізації застосунку посередництвом статичного тестування коду застосунку, динамічного тестування компонентів та їх інтеграцій, і т. д. Як вже було сказано, цей тип тестування захисту веб-застосунків тісно переплітається із тестуванням якості веб-застосунку.
- Тестування на проникнення(Penetration Testing) – ручний процес імітації спроб реалізації загроз веб-застосунку, в результаті якого тестер виявляє слабкості цільової системи і передає інформацію про них замовнику. Процес тестування на проникнення повністю слідує практикам зловмисників.

Кожний тип тестування захищеності веб-застосунку гарантує свою визначену частину захисту застосунку. Тобто лише використання усіх типів тестування надає максимальні гарантії протидії ймовірним атакам, проте не гарантує ідеальної захищеності в силу неможливості врахувати усі ризики. Дійсно, імплементація усіх типів тестування є дуже витратним процесом для бізнесу, що створює свій веб-застосунок, тому спеціалісти із тестування безпеки розробляють все нові і більш ефективні методи і інструменти для зменшення цих витрат.

Як вже було сказано, типи тестування об'єднують методи забезпечення захисту веб-застосунку(те ж саме, що і тестування захисту веб-застосунку) і перелічити усі ці методи є дуже трудомісткою задачею. На практиці при перевірці захищеності веб-застосунку встановлюється перелік тестів, по успішному виконанні яких і оцінюється захищеність. Розглянемо ці тести захищеності веб-застосунків на прикладі документу із рекомендаціями - Certified Secure Web Application Security Checklist[4]. На основі визначених Certified Secure тестах та згаданих вище типах

тестування захищеності веб-застосунків встановимо між ними відповідність. Для визначеного типу тестування зазначимо певні методи за допомогою яких реалізуються тести захисту із CS Web Application Checklist:

Таблиця 1.2 – Тести та відповідні методи проведення тестування захищеності веб-застосунків

Тест на захищеність веб-застосунку від певної загрози	Типи тестування та їх методи, що застосовуються в цих тестах
Тестування конфіденційності та нерозкриття інформації	
Тестування наявності зайвих файлів та директорій в середовищі веб-застосунку, наявності в них критичної інформації про веб-застосунок	WAST[1)]: Сканування файлової системи; SA[2)]: Аудит;
Тестування наявності критичної інформації веб-застосунку в публічному доступі, в лог-файлах, повідомленнях про помилки	WAST: Сканування файлової системи; PT[3)]: OSINT;
Тестування можливості розкриття інформації про середовище обслуговування веб-застосунку(топології внутрішніх мереж, дані про хостів, і т. д.)	PT: OSINT, Сканування портів, Тестування на проникнення у внутрішню мережу; SA: Аудит;
Тестування наявності критичної інформації в URL-запитах веб-застосунку	WAST: Статичний аналіз вихідних кодів, фазінг, URL маніпуляції; PT: фазінг;
Тестування захищеності збереження на клієнті та передачі критичної інформації від клієнта до веб-сервера	WAST: Статичний аналіз вихідних кодів, динамічний аналіз;

1) Скорочення від "Web Application Security Testing"
2) Скорочення від "Security Audit"
3) Скорочення від "Penetration Testing"

Продовження таблиці 1.2

Тест на захищеність веб-застосунку від певної загрози	Типи тестування та їх методи, що застосовуються в цих тестах
Тестування наявності вразливості до "cache poisoning"	WAST: Динамічний аналіз; SA: Аудит;
Тестування наявності вразливостей використання протоколів SSL/TLS	SA: Перевірка стандартами безпечного застосування криптографії;
Тестування механізмів автентифікації та авторизації	
Тестування наявності і адекватності механізмів автентифікації та механізмів авторизації на стороні сервера	WAST: Статичний аналіз вихідних кодів, динамічне тестування; PT: Фазінг, Ручний аналіз;
Тестування механізмів автентифікації та авторизації на стороні клієнта	WAST: Динамічне тестування;
Тестування складності використовуваних паролів	WAST: Статичний аналіз вихідних кодів; SA: Перевірка стандартами; PT: Підбір паролів, злам хешів паролів, фазінг;
Тестування наявності передбачуваних та відомих за замовчуванням автентифікаційних даних облікових записів, токенів	PT: Ручне тестування на проникнення, злам паролів, фазінг;
Тестування наявності обходів механізмів автентифікації/авторизації(наявність прямого доступу до об'єктів веб-застосунку)	WAST: Динамічний аналіз, фазінг; PT: Ручне тестування на проникнення;
Тестування наявності захисту від brute-force атак	PT: фазінг;

Продовження таблиці 1.2

Тест на захищеність веб-застосунку від певної загрози	Типи тестування та їх методи, що застосовуються в цих тестах
Тестування захищеності механізмів зміни автентифікаційних даних, механізму виходу з облікового запису	WAST: Статичний аналіз вихідних кодів, динамічний аналіз; PT: Ручне тестування на проникнення; SA: Перевірка стандартами;
Тестування використовуваної криптографії	
Тестування механізму збереження паролів на стороні сервера	WAST: Статичний аналіз вихідних кодів; SA: Аудит;
Тестування наявності використання сумнівних криптографічних алгоритмів, некоректного використання перевірених алгоритмів	WAST: Статичний аналіз вихідних кодів; SA: Перевірка стандартами;
Тестування захищеності механізмів використання сертифікатів	SA: Аудит;
Тестування механізму формування криптографічних секретів(генерування випадкових чисел)	WAST: Статичний аналіз вихідних кодів; SA: Перевірка стандартами;
Тестування обробки вводу користувача	
Тестування наявності вразливості до SQL-injection	PT: Фазінг, Ручне тестування на проникнення; WAST: Фазінг;
Тестування наявності вразливості до XSS	PT: Фазінг, Ручне тестування на проникнення;
Тестування наявності вразливості до command-injection	PT: Фазінг, Ручне тестування на проникнення;

Продовження таблиці 1.2

Тест на захищеність веб-застосунку від певної загрози	Типи тестування та їх методи, що застосовуються в цих тестах
Тестування наявності вразливості до XML-injection	PT: Фазінг, Динамічний аналіз;
Тестування наявності вразливості до HTTP-injection	PT: Фазінг, динамічний аналіз;
Тестування наявності вразливостей в механізмах десеріалізації	PT: Фазінг, динамічний аналіз;
Тестування наявності вразливості до Path-traversal	PT: Фазінг, Ручне тестування на проникнення; WAST: Перевірка конфігурацій веб-сервера;
Тестування механізму керування сесій	
Тестування механізмів зміни та відкликання сесій при зміні автентифікаціних даних та виході з облікового запису, при закінченні терміну дії сесії	WAST: Статичний аналіз вихідних кодів, динамічний аналіз; PT : Ручне тестування на проникнення;
Тестування наявності вразливості до CSRF	PT: Ручне тестування на проникнення; SA: Перевірка стандартами;
Тестування захищеності алгоритму генерування ідентифікаторів сесій	WAST: Статичний аналіз вихідних кодів, динамічний аналіз;
Тестування вразливості до Session fixation	PT: Ручне тестування на проникнення; SA: Перевірка стандартами;
Тестування захищеності cookies на стороні клієнта	WAST: Динамічний аналіз; SA: Перевірка стандартами;

Продовження таблиці 1.2

Тест на захищеність веб-застосунку від певної загрози	Типи тестування та їх методи, що застосовуються в цих тестах
Тестування алгоритмів завантаження файлів на веб-сервер	
Тестування місця завантаження файлів на веб-сервері, атрибутів їх доступу та правил їх обробки	WAST: Статичний аналіз вихідних кодів, динамічний аналіз; SA: Аудит;
Тестування наявності вразливості до завантаження файлів в непризначені директорії	PT: Фазінг, Ручне тестування на проникнення; WAST: Динамічний аналіз;
Тестування наявності перевірок розміру та формату завантажуваних файлів	WAST: Статичний аналіз вихідних кодів; PT: Фазінг, Ручне тестування на проникнення;
Тестування механізмів обробки XML-контенту	
Тестування захищеності механізмів парсингу XML	WAST: Динамічний аналіз, фазінг;
Тестування наявності небезпечних XML розширень	SA: Аудит;
Тестування захищеності клієнтської сторони веб-застосунку	
Тестування наявності вразливостей до clickjacking-у	WAST: Динамічний аналіз;
Тестування наявності вразливостей інсталяції веб-застосунку, доставки і встановлення оновлень	WAST: Динамічний аналіз;

Кінець таблиці 1.2

Тест на захищеність веб-застосунку від певної загрози	Типи тестування та їх методи, що застосовуються в цих тестах
Тестування захищеності серверної сторони веб-застосунку	
Тестування захисту від DoS атак	PT: Фазінг; WAST: Тестування під навантаженням;
Тестування наявності вразливості до SSRF	
Тестування наявності вразливостей перенаправлення користувачів у веб-застосунку	PT: Ручне тестування на проникнення;
Тестування наявності зайвих сервісів в середовищі роботи веб-сервера	SA: Аудит; PT: Сканування портів;

В результаті бачимо наступне – деякі методи тестування захищеності маючи одну назву належать одразу декільком типам тестування. Один із таких методів – фазінг. В наступних підрозділах буде детальніше розглянуто його суть і роль в застосуванні усіх типів тестування захищеності веб-застосунків, до яких він належить.

1.2 Фазінг – визначення і класифікація

Фазінг(англ. - “fuzzing”) – метод тестування програмних продуктів, що полягає в формуванні величезних масивів вхідних даних для піддослідного ПЗ і аналізі результатів його обробки цих даних. Масиви вхідних даних можуть містити як звичайні, «правильні» дані, так і аномальні – такі що призведуть до помилкових

ситуацій. Таким чином, застосовуючи величезну кількість різноманітно сформованих даних до ПЗ і аналізуючи його роботу на цих вхідних даних, можна робити висновки про стійкість даного ПЗ до різних класів аномальних вхідних даних(тобто, про відсутність відповідних вразливостей у ПЗ), або можна судити про присутність вразливості, якщо на деякому наборі даних поведінка ПЗ не відповідала очікуванням.

На практиці послідовність дій у виконанні даного методу тестування така:

1) Вибір цілі тестування
2) Опис специфікацій «точок входу» у ПЗ та вибір із них точок, що будуть протестовані
3) Створення масивів вхідних даних для заданих «точок входу»
4) Запуск на виконання ПЗ із кожним набором вхідних даних з масиву
5) Перехоплення помилкових ситуацій, логування пар запит-відповідь
6) Створення звіту та аналіз

Інструменти, що імплементують фазінг(іменуються "фазерами"), та підходи до тестування, частиною яких є фазінг, можна класифікувати за багатьма ознаками:

- Класифікація за ціллю тестування:

File format fuzzer – фазери парсерів певних файлових форматів. Такі фазери виявляють помилки і баги програм обробки файлів, подаючи їм на обробку навмисно помилково сформовані дані.

Web-App fuzzer – спеціалізовані фазери веб-застосунків. Фазінгу піддаються усі інтерфейси веб-застосунку і відповідно виявляються вразливості його фронтенду і бекенду.

Web-browser fuzzer – фазери програм веб-браузерів. Здебільшого фазінг направлений на виявлення вразливостей обробки HTML, виконання скриптів і підтримки форматів медіа-контенту.

Network-Protocol fuzzer – фазери протоколів нижніх рівнів стеку мережевих протоколів – Physical, Data link, Network, Transport рівнів OSI.

General-Purpose fuzzer – фазери загального призначення. Такі фазери використовуються для ПЗ, що не належить до перелічених вище прикладів, і окремо вимагають налаштувань фазерів на цільовий інтерфейс та формат формованих вхідних даних.

- Класифікація за наявністю апріорних знань про ціль тестування:

White box fuzzing – при налаштуванні фазера використовуються дані про вихідний програмний код цільового ПЗ.

Gray box fuzzing – фазер імплементує попередній аналіз цільового ПЗ для збору релевантної інформації; не використовує дані про вихідний програмний код.

Black box fuzzing – фазінг не залежить від інформації про ціль тестування.

- Класифікація за алгоритмом генерації вхідних даних:

Generation-Based fuzzer – фазер генерує масиви вхідних даних ітераціями. Використовуючи побудовану модель цільового ПЗ, або на основі конфігурації фазер "пристосовує" згенеровані дані з попередньої ітерації для отримання "кращих" результатів на наступній. "Кращі" результати означають, що нові згенеровані дані можуть виявити нову і глибшу помилку в тому самому інтерфейсі.

Mutation-Based fuzzer – фазер використовує певні зразки вхідних даних і за заданим попередньо алгоритмом змінює ці зразки в окремих місцях, генеруючи множину нових вхідних даних.

Protocol-Based fuzzer – фазер генерує вхідні дані виключно на основі специфікації цільового протоколу обробки даних і використовує відомі зловмисно сформовані послідовності, що можуть виявити помилки у відповідних місцях у цільовому ПЗ.

Hybrid fuzzer – фазер, що не належить до перелічених вище класів; фазер довільного алгоритму генерації зразків вхідних даних, чи той, що імплементує принципи декількох інших класів фазерів.

- Класифікація за тестовим призначенням:

Directed fuzzing – фазінг направлений на дослідження певного функціоналу цільового ПЗ, ігноруючи інші розгалуження логіки його коду.

Coverage-based fuzzing – фазінг направлений на перевірку якомога більшої частини функціоналу цільового ПЗ.

Із вразливостей, що тестери шукають за допомогою фазінгу можна виділити: вразливості коду, що призводять до Injection-атак, помилки обробки вхідних даних чи вразливості, що викликають необроблювані ситуації, вразливості коду, що призводять до успішних атак переповнення буферу, пошкодження пам'яті чи розкриття інформації. В наступному підрозділі буде розкрито роль фазінгу в різних типах тестування захищеності а також деталі його застосування на практиці.

1.3 Роль фазінгу у виявленні багів у веб-застосунках

Як окремий метод тестування захищеності веб-застосунків фазінг належить типам тестування: сканування на наявність вразливостей, тестування безпеки застосунку та тестування на проникнення. Таким чином фазінг підходить для систематичного застосування на багатьох стадіях життєвого циклу ПЗ.

При скануваннях на наявність вразливостей фазінг стає рішенням автоматизації тестування відповідних вхідних точок веб-застосунку – відкритих сервісів, автентифікаційних форм та форм вводу користувача таких як, наприклад,

текстові поля блог-сайтів. Brute-force тестування, наприклад, відкритих сервісів застосунку чи логін форми в ньому, є нічим іншим як фазінгом цих "вхідних точок". Сканування на наявність відомих вразливостей може відбуватись без втручання людини і проводитись регулярно із створенням відповідних звітів і потребує лише налаштування фазінгу із вихідними даними про відомі вразливості цільових "вхідних точок".

Крім того, фазінг може виступати в якості симуляції діяльності зловмисника із виявлення вразливостей ПЗ – його застосовують як одноіменний інструмент тестування на проникнення – фазер. А на стадії супроводу програмного продукту фазінгом виявляють непомічені раніше вразливості, що вимагають оновлень безпеки, тим самим запобігаючи Zero-Day атакам. Фазінг обирають тестери при динамічному тестуванні безпеки веб-застосунку завдяки його автоматизації і широкому покритті тесткейсів. Виявлені тестерами на проникнення вразливості помічаються як баги і підлягають виправленню, а процедура такого тестування регулярно повторюється.

Для тестування безпеки коду веб-застосунку фазінг є компромісом між витратністю і ефективністю між статичним та динамічним тестуванням коду ПЗ. Також фазінг став широко використовуваним завдяки своїй ідейній простоті та ефективності в загальному випадку. Фазінг виявляє помилки, що були пропущені на попередніх тестування через ті чи інші обставини, значно доповнюючи результати ручного тестування, а іноді фазінг може бути єдиним варіантом перевірки якості програмного продукту – наприклад при закритості розробки програмного коду, коли інформація про ціль тестування обмежена("black box", або "grey box" тестування).

Одного лише фазінгу, однак, недостатньо для гарантій захищеності. Результати такого тестування все рівно вимагають людського аналізу і можуть нараховувати багато false-positive результатів. Також людського втручання вимагає пошук конкретної помилки в коді, що викликає результати, зафіксовані фазінгом як

баг. Фазінг знаходить баг, але не його причину. Серед інших складностей застосування фазінгу виділяються витрати часу і оперативна складність етапу налаштування фазінгу. Встановлення алгоритму генерування вхідних даних для поточного цільового інтерфейсу може бути трудомістким, проте існують інструменти, що вирішують компроміс між складністю налаштування та кількістю надаваних результатів.

Висновки до розділу 1

В першому розділі було підсумовано множини вразливостей, притаманних веб-застосункам та типам тестування їх безпеки. Встановивши відповідність необхідних тестів захищеності веб-застосунків із застосовуваними методами тестування було виявлено, що одним із найпопулярніших методів є фазінг. Фазінг має деякі відмінності реалізації і призначення серед різних типів тестування захищеності, а саме тестування на проникнення, тестування безпеки застосунку та сканування на наявність вразливостей. Тим не менш, наведені в цьому розділі факти свідчать про те, що фазінг гідний уваги і глибших досліджень з метою підвищення його ефективності як самостійного підходу тестування. Перший розділ вмістив у себе вичерпний, але загальний опис фазінгу і деталі існуючої класифікації цього методу тестування. Були розглянуті сильні і слабкі сторони фазінгу. Проте головною темою даної роботи є тестування захищеності RESTful веб-застосунків – широко поширений клас веб-застосунків, що відповідно вимагає уваги з боку забезпечення інформаційного захисту. Наступний розділ буде містити аналіз цієї технології REST, як і існуючі підходи до захисту веб-сервісів, що їх застосовують. Зокрема буде проаналізовано вклад фазінгу в захист REST API веб-застосунків.

2 ВРАЗЛИВОСТІ ТА ТЕСТУВАННЯ REST API

2.1 Характеристичні властивості REST API, призначення та відомі вразливості

REST - це програмне архітектурне рішення для розподілених систем доставки медіа-контенту. Опис технологій і принципів нового підходу до створення систем доставки медіа-контенту був вперше представлений на дисертації Р. Т. Фіелдінга[5] 2000 року. Згідно з нею виділимо характеристичні властивості REST.

REST систему визначає дотримання специфічних для цього рішення обмежень і принципів. Обмеження в основі REST:

- Універсальний інтерфейс доступу до ресурсів – центральна властивість REST систем. Сумісність інтерфейсів доступу до ресурсів у систем різного призначення дозволяє спростити процеси їх розробки і взаємодії між ними.
- Розділення клієнтської та серверної програмних рішень – дозволяє простіше масштабувати мережу взаємодії клієнтів із серверами, розділяючи проблеми оновлень обох сторін.
- Взаємодія клієнт-сервер без запам'ятовування стану – спрощує керування станами взаємодій між клієнтом і сервером, перекладаючи за це відповідальність на клієнта, а сервер звільняється від підтримки станів.
- Кешування – основа концепції збільшення ефективності мережевої взаємодії.

- Ієрархічна система компонентів – введення рівнів компонентів всієї системи і обмеження їх взаємодії дозволяє спростити сам функціонал цієї взаємодії, а також дозволяє просто розширювати ланцюги компонентів, задіяних в цій взаємодії.
- Код на вимогу – обов'язкова можливість розповсюдження програмних розширень серверами системи, що применшує вимоги до пре-інстальованих компонентів на клієнтах.

REST можливий у будь-якій системі доставки медіа. В даній роботі, однак, увага приділятиметься застосуванню REST до програмних рішень у веб-застосунках, а отже буде детальніше розглянуто імплементації принципів REST у бекенді веб-застосунків та протоколі взаємодії клієнт-сервер HTTP. Імплементований архітектурний програмний стиль у веб-застосунках, що відповідає принципам і обмеженням REST називають RESTful.

З точки зору клієнта RESTful веб-застосунок дозволяє отримати доступ до його ресурсів за їх унікальними ідентифікаторами, які одночасно є спеціально сформованими URL. Поля запиту в протоколі HTTP та формат навантаження повністю визначають маніпуляцію із ресурсом, яку хоче здійснити клієнт.

HTTP метод: GET визначає запит на зчитування ресурсу з сервера, POST визначає запит на створення нового ресурсу, PUT – оновлення вмісту існуючого ресурсу, а DELETE – видалення ресурсу.

Дані в тілі запиту в RESTful застосунку мають формат JSON, як і дані в тілі відповіді сервера. Функціонал обробки таких клієнтських запитів представляє собою RESTful API. Головне призначення RESTful веб-сервісів і застосунків – завдяки відкритості своїх RESTful API в будь-який момент часу без проблем сумісності обмінюватись ресурсами і доставляти медіа-контент клієнтам на вимогу. API призначено для розробників застосунків з функціями доступу до ресурсів інших

застосунків, що відповідають принципам REST, причому сумісність і простота доступу до існуючого API досягається створенням специфікацій API, мова про які піде в наступному підрозділі. Такі веб-сервіси як Facebook, Twitter і Google підтримують RESTful API. RESTful став стандартом побудови веб-сервісу поряд із такими рішеннями як SOAP, WSDL і подібними. Стандартизація архітектурних програмних рішень позитивно позначається на швидкості розробки, масштабованості результуючих систем та якості обслуговування клієнтів, проте RESTful рішення також містять відомі вразливості.

За даними джерела OWASPS-Top-10 [6] вразливостей, специфічних для REST, існують можливі слабкості цих систем:

- Слабкий механізм автентифікації користувачів REST API: найпоширенішими причинами існування такої вразливості є використання завідомо слабких алгоритмів автентифікації користувачів REST API, використовуваної в них криптографії або слабкості зберігання автентифікаційних даних, проте список причин зазначеними не обмежений.
- Слабкий механізм авторизації на рівні ресурсів: при слабкій реалізації контролю доступу до ресурсів автентифікованим користувачам вони можуть отримати доступ до певних ресурсів в обхід механізмів авторизації.
- Слабкий механізм авторизації на рівні REST API: зазвичай дана вразливість з'являється в комплексних моделях доступу до ресурсів, контроль доступу в яких був реалізований з помилками, внаслідок чого стають можливі порушення встановлених правил доступу – вертикальне чи горизонтальне підвищення привілеїв.
- Витік інформації у відповідях клієнту REST API: внаслідок помилок розробників програмного коду виклики функцій повертають надлишкову інформацію, в тому числі і деяку критичну інформацію, як наприклад інформацію про помилки обробки запиту, конфіденційні дані користувача, і т. д.

- Слабкі механізми квотування доступу до ресурсів
- Слабкі конфігурації: при неналежному налаштуванні веб-сервера зловмисники можуть використати незаборонені функції, запити чи дії на ньому.
- Надлишковий функціонал: якщо деякі функції REST API були залишені розробниками для службового користування або експлуатаційного тестування, без належного захисту, то ними можуть скористатися зловмисники. Тобто головною причиною даної вразливості стає рішення розробника приховати від зловмисника ці функції.
- Вразливості до Injection-атак: при відсутності валідації вхідних даних із ненадійних джерел зловмисник може сформувати запит, що викличе непередбачену розробником дію на веб-сервері. Зазвичай зловмисники таким запитом виконують на веб-сервері довільний програмний код.
- Слабкості механізмів логування і звітності в REST API
- Інше, включаючи "баги" програмного коду, помилки логіки і т. д.

Аналогічно розглянутій відповідності методів тестування певним вразливостям веб-застосунку в наступному підрозділі окрім іншого буде розглянуто тестування REST API, направлене на розглянуті вище класи вразливостей.

2.2 Стислий опис складових частин REST API та опис конструкцій, пов’язаних із тестуванням захищеності коду

При імплементації серверної частини RESTful застосунку розробники обирають між програмними рішеннями на .NET, Python, Ruby або Java. Існує безліч фреймворків для розробки API з нуля, проте в даній роботі вони не розглядаються. В результаті розробки обов'язковими компонентами результуючої системи(і водночас

компонентами, цікавими з точки зору захисту) є механізми автентифікації та авторизації користувачів API, механізм логування, логіко-специфічні механізми, в тому числі і обробник вхідних запитів та документування API.

Структурно на схемі ці компоненти і їх взаємодія між собою та з клієнтом виглядатиме так:

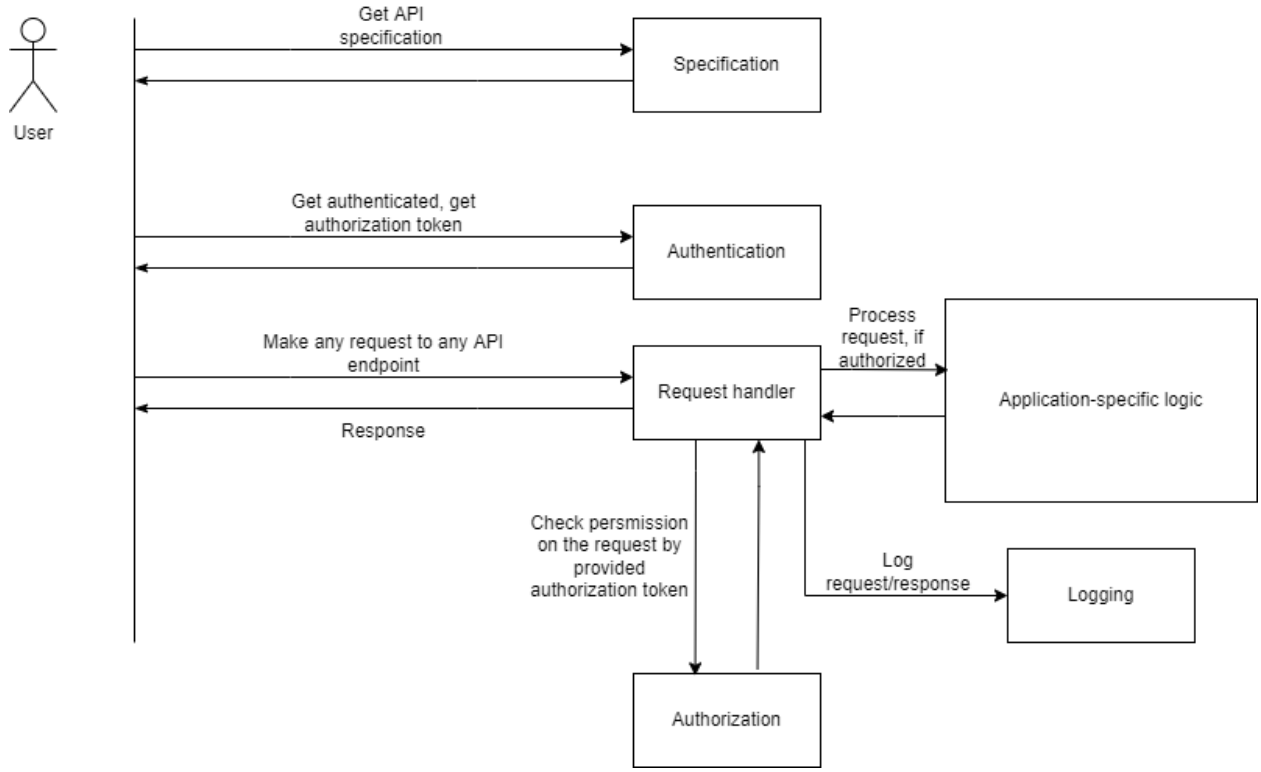

Рисунок 2.1 – Структурна схема взаємодії вразливих компонент REST API

Розглянуті в попередньому підрозділі вразливості REST API з'являються в результаті помилок імплементації цих компонентів і для кожної з них в даному підрозділі буде розглянуто можливі підходи та інструменти тестування API.

1) Автентифікація

У RESTful веб-застосунках для доступу до довільної функції API користувач має автентифікуватись перед базою даних облікових записів. Автентифікаційні дані передаються в запиті HTTP – в передбачених для цього заголовках, користувацьких заголовках або в тілі запиту. Поширеною практикою є передача пари ім'я користувача-пароль в заголовку Authorization(наприклад Basic із base64

закодованими ім'ям користувача та паролем), або передача ключа "apiKey" в URL або тому ж заголовку Authorization. Автентифікація може відбуватись як на самому сервері застосунку так і на окремому сервісі автентифікації. Можливими відповідями на запит можуть бути HTTP коди 200(Успіх) або 401(Провал автентифікації) із подальшими роз'ясненнями відповіді в тілі запиту, або в заголовку WWW-Authenticate HTTP. З точки зору безпеки автентифікація REST API непридатна для використання без допоміжних таких засобів захисту передачі HTTP пакетів від сніфінгу та MITM атак, як наприклад SSL.

Тестування автентифікації зводиться до перевірок застосованих алгоритмів за перевіреними стандартами захищеності, статичних та динамічних тестів коду або фазінгу при тестуванні на проникнення. Слабкості автентифікації призводять до підвищення привілеїв і виконання функцій API від імені вкраденого облікового запису.

2) Авторизація

Як наступний крок після успішної автентифікації веб-застосунок формує авторизаційні дані для користувача, які той застосовує до кожного свого наступного запиту і які визначають дозволені йому дій із функціями API. Існують поширені практики формування цих авторизаційних даних: Basic Auth, JWT, OAuth та інші алгоритми специфічні для певних систем(таких як AWS, Facebook і т. д.) Аналогічно автентифікації даний механізм є недостатнім без таких засобів захисту потоку HTTP запитів від сніфінгу та MITM атак, наприклад SSL.

Тестування авторизації на рівні ресурсів найефективніше при підходах тестування на проникнення, а тестування авторизації на рівні функцій API також імплементується динамічними тестами, що створюються на основі імплементованої в системі моделі доступу до функцій(зазвичай рольової моделі).

3) Логування

В цілях безпеки усі спроби автентифікації та акти доступу до ресурсів та функцій API мають документуватися. Лог-файли надзвичайно важливі у реагуванні на інциденти безпеки(детектування та запобугання за допомогою автоматизованих систем захисту) а також в пост-інцидентних розслідуваннях.

Тесування логування полягає в перевірках імплементованого механізму за стандартами захисту і аудиті безпеки.

4) Специфічна логіка API

Вразливості логіки програмного коду чи алгоритмів їх взаємодії становлять одну з найбільших загроз RESTful веб-застосунків. Ресурси доступу в таких застосунках представляються як об'єкти із набором властивостей і функціями змін цих властивостей, а логіка застосунку представляє собою можливі послідовності виконання функцій API так, що результат залежить від цієї послідовності. Непередбачені послідовності викликів, відсутність коректної обробки логічно помилкових запитів, некоректні виклики функцій API сторонніх веб-застосунків, і подібні помилки реалізації API повинні тестуватися в першу чергу розробником. Також в специфічну логіку API можна включити імплементовані заходи захисту такі як захист від DoS атак на виклики функцій та квотування пам'яті виділеної під створювані ресурси.

Тестер захисту, що перевіряє код веб-застосунку до стадії впровадження, застосовує динамічні тести, а при тестах на виробничій версії тестер застосовує фаззінг викликів функцій для перевірок захисту від DoS атак та нездатності застосунку обробити певні послідовності викликів.

5) Обробка вхідних запитів

Програмний модуль обробки запитів до REST може мати вразливості різної серйозності, починаючи від помилок форматування чи типізації вхідних даних, що викликає програмні виключення, і закінчуючи вразливостями переповнення буфера і т. п.

6) Документація

Документація REST API потрібна розробникам клієнтських частин застосунків, що будуть викликати функції з API. Опис синтаксису і семантики викликів можливо сформувати довільним чином, проте в цілях підвищення ефективності розробки клієнтських частин були створені стандарти документацій API – специфікації. Специфікації задають строгий формат опису викликів функцій API і призначені для автоматизованої генерації придатної для читання людиною документації, автоматизованого генерування клієнтської частини RESTful веб-застосунку і т. д. Тестер API створює тесткейси на основі парсингу специфікації, тобто вона є ключовою структурою даних в тестуванні захисту API. Із найпопулярніших стандартів створення специфікацій виділяються RAML, API Blueprint та OpenAPI. В подальшому розділі – в практичній частині – буде розглядатися стандарт OpenAPI(у минулому Swagger) і його застосування при фазінгу API.

Висновки до розділу 2

В даному розділі було зібрано відомості про будову RESTful веб-застосунку, програмно архітектурні принципи їх побудови та будову REST API з точки зору тестера безпеки. Було розглянуто перелік вразливостей, притаманних REST API і цей перелік буде застосований в практичній роботі для оцінки застосовності різних типів фазингу API. Із виділених у другому підрозділі складових компонентів REST API ті, що підлягають тестуванню фазінгом, а саме: механізми автентифікації, авторизації, захисту від DoS атак на функції API та алгоритми, що імплементують

логіку застосунку – усі ці компоненти будуть розглянуті як цілі для фазінгу в практичній роботі. Із корисних структур даних в розпорядженні тестера виступить специфікація цільового REST API. Далі ці відомості будуть використані в практичній частині даної роботи – фазінг тестуванні веб-застосунку, що імплементує REST API.

3 РОЗРОБКА МЕТОДИКИ ФАЗІНГУ REST API ІЗ BURP SUITE І RADAMSA. ПОРІВНЯННЯ ІНСТРУМЕНТАРІЮ ДЛЯ РУЧНОГО ФАЗІНГУ ІЗ RESTLER

3.1 Розробка методики фазінгу REST API

На основі розглянутих в попередньому розділі характеристичних властивостей REST API та їх поширених вразливостей побудуємо методику фазінгу функцій API доступу до певного ресурсу.

1) Підготовка розробки

У якості вихідних даних для розробки дана структура функціонуючого RESTful веб-додатка, описана у другому розділі(Рисунок 2.1):

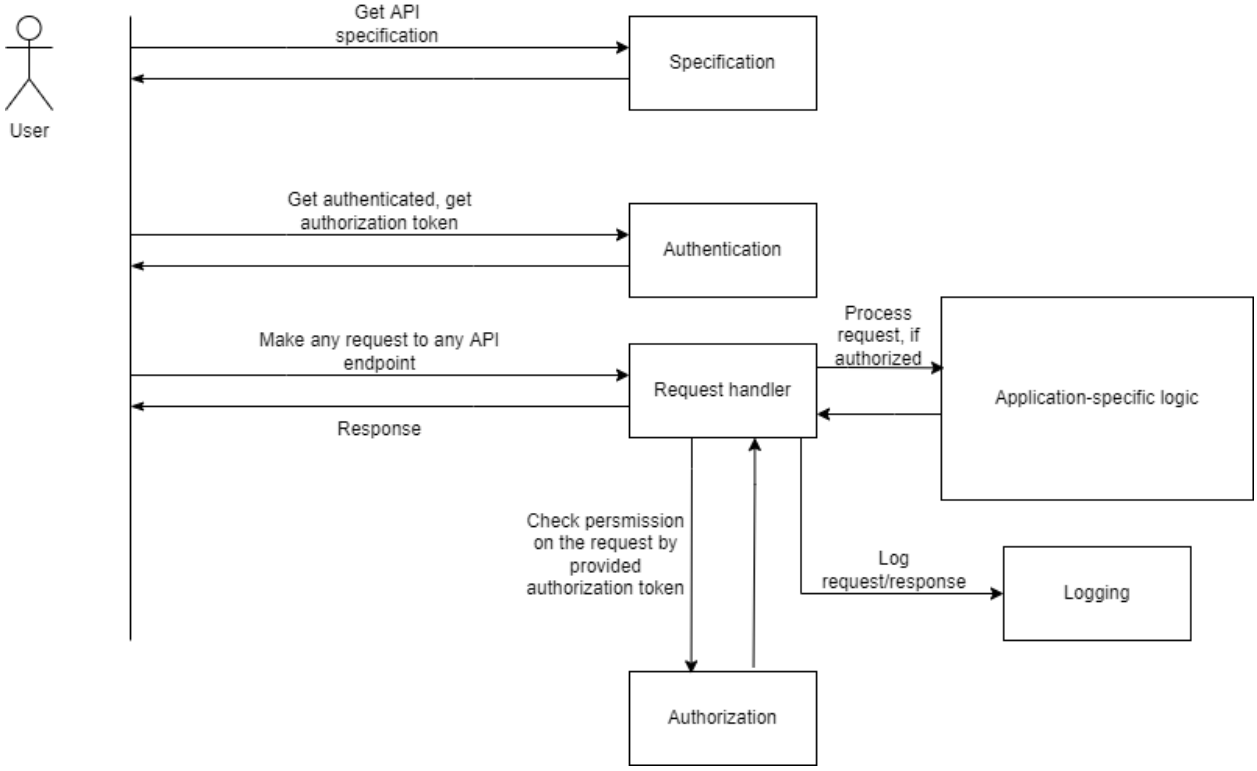

Рисунок 3.1 – Структурна схема взаємодії вразливих компонент REST API

Також з другого розділу з відомостей про можливості фазінгу та його призначення можна зазначити класи помилок реалізації REST API, які даним методом тестування виявляють:

- Недостатню фільтрацію вхідних даних
- Небезпечні практики програмування(відомо вразливий код)
- Помилки бізнес логіки функцій в цілому
- Помилки оброблення програмних виключень
- Небезпечну маніпуляцію ресурсами системи (пам'ять, процесорний час, кількість одночасно підтримуваних з'єднань)
- Помилки при яких відбувається розкриття інформації
- Нестійкі до атак механізми автентифікації/авторизації та слабкі логіни/паролі користувачів

2) Початкові умови та необхідні засоби

Для поточної розробки методу фазінгу API початкові умови будуть аналогічні black-box фазінгу, що означає, що тестер отримує лише специфікацію цільового API. Також для тестування автентифікації та авторизації тестер не має інформації про зареєстрованих у системі користувачів та внутрішню будову відповідних механізмів.

3) Розробка методики

Окремий ресурс в системі REST має як мінімум 4 методи доступу – POST, PUT, GET та DELETE. У вигляді скінченного автомата Мілі життєвий цикл ресурсу буде виглядати так:

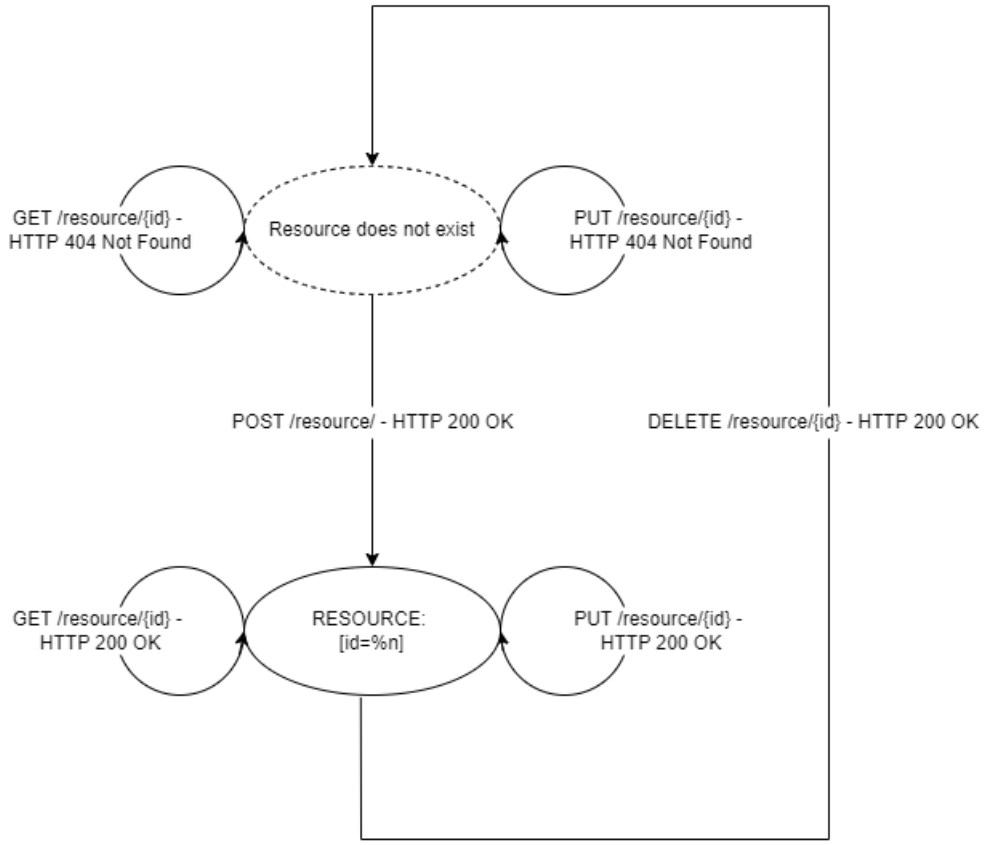

Рисунок 3.1 – Схема життєвого циклу ресурсу у REST

Неочікувані відповіді на запити окремих функцій в цьому життєвому циклі свідчать про наявність багу конкретно в реалізації даної функції.

Рисунок-демонстрація прикладу наявності цих багів на схемі життєвого циклу ресурсу:

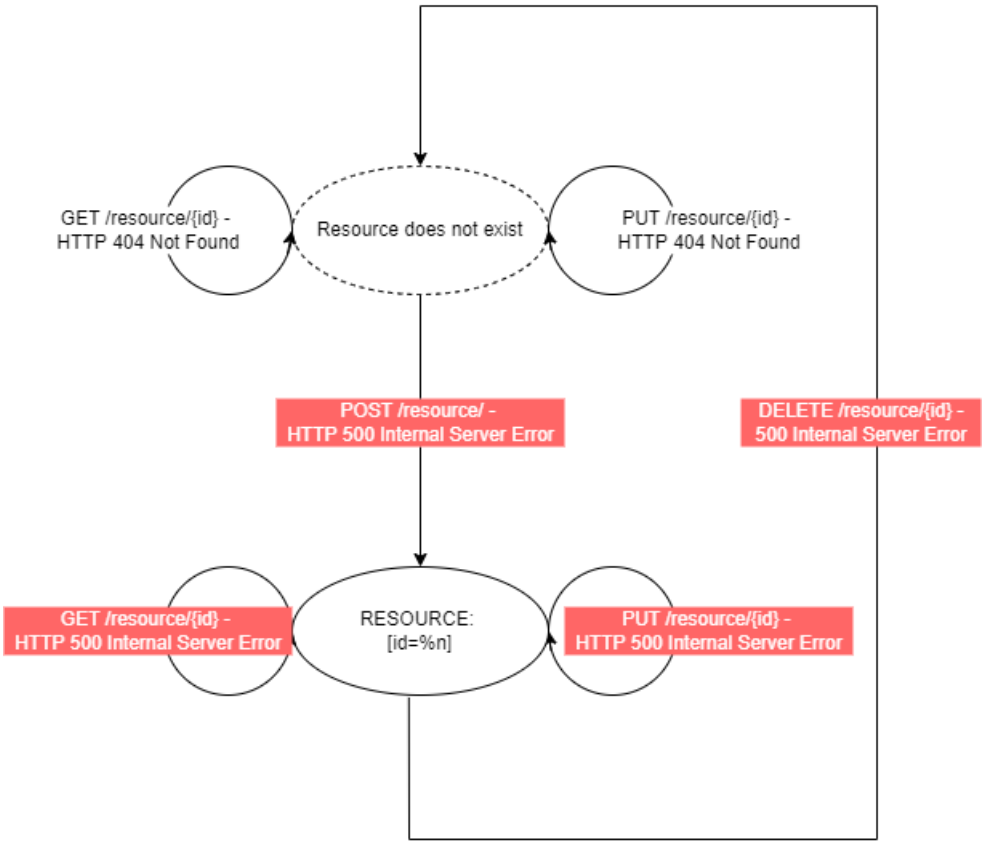

Рисунок 3.2 – Баги окремих функцій доступу до ресурсу

Ці баги зазвичай відповідають вразливостям:

- Недостатня фільтрація вхідних даних
- Помилки оброблення програмних виключень

Про це завжди свідчить відповідь HTTP 500 Internal Server Error, тому при фазінгу окремих функцій тестер шукає такі відповіді.

Вразливі практики коду та небезпечну маніпуляцію ресурсами виявити можна лише послідовністю тестових запитів.

Один приклад - недоступність ресурсу після його створення. Створений, але недоступний на запит ресурс є витоком пам'яті і свідченням того, що програмно доступ до ресурсів реалізований з помилками.

Ситуація 1: Перший запит на створення ресурсу був виконаний успішно, але хоча б один із запитів на доступ до новоствореного ресурсу повертає помилку. Демонстрація на рисунку:

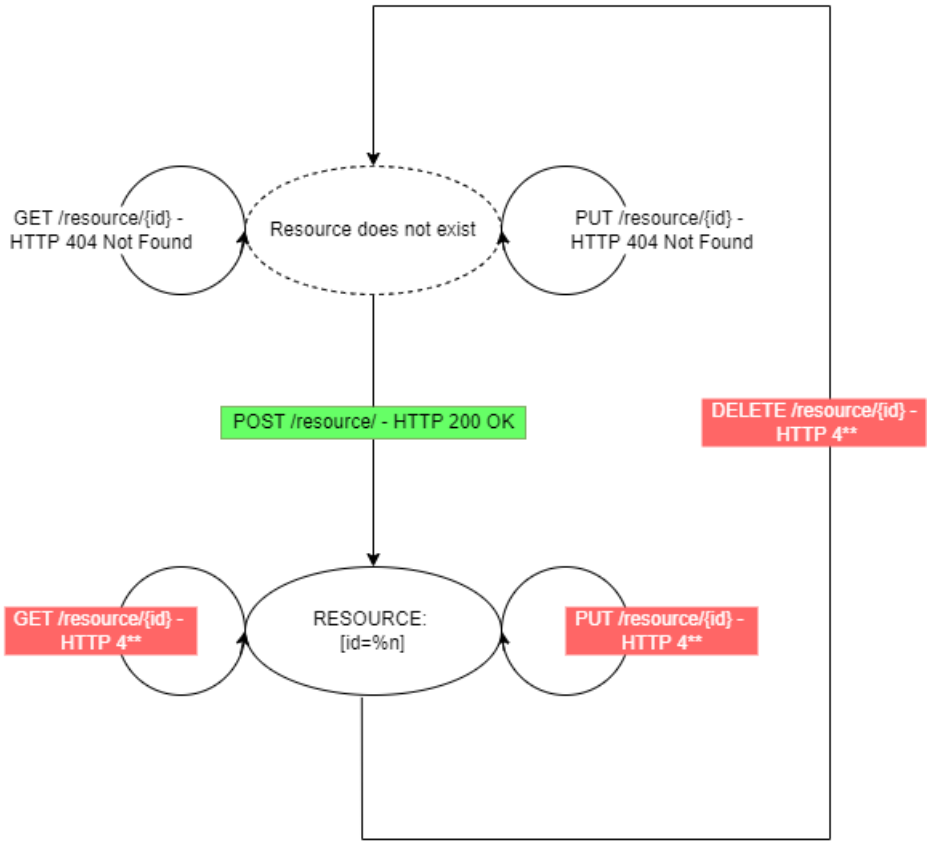

Рисунок 3.3 – Баг, ресурс недоступний після створення

Другий приклад - успішно видалений ресурс залишається доступним функціям API, або недоступний за API, але не звільнений з системи (наприклад, тіло ресурсу не було видалено з бази даних остаточно).

Ситуація 2: Перший запит на видалення ресурсу був виконаний успішно, але хоча б один із запитів на доступ до видаленого ресурсу за ідентифікатором, що був йому присвоєний, відбувається успішно. Оскільки перший та інші запити відбуваються послідовно, можна виключити можливість повторного використання ідентифікаторів для новостворених ресурсів.

Такий сценарій виглядатиме так:

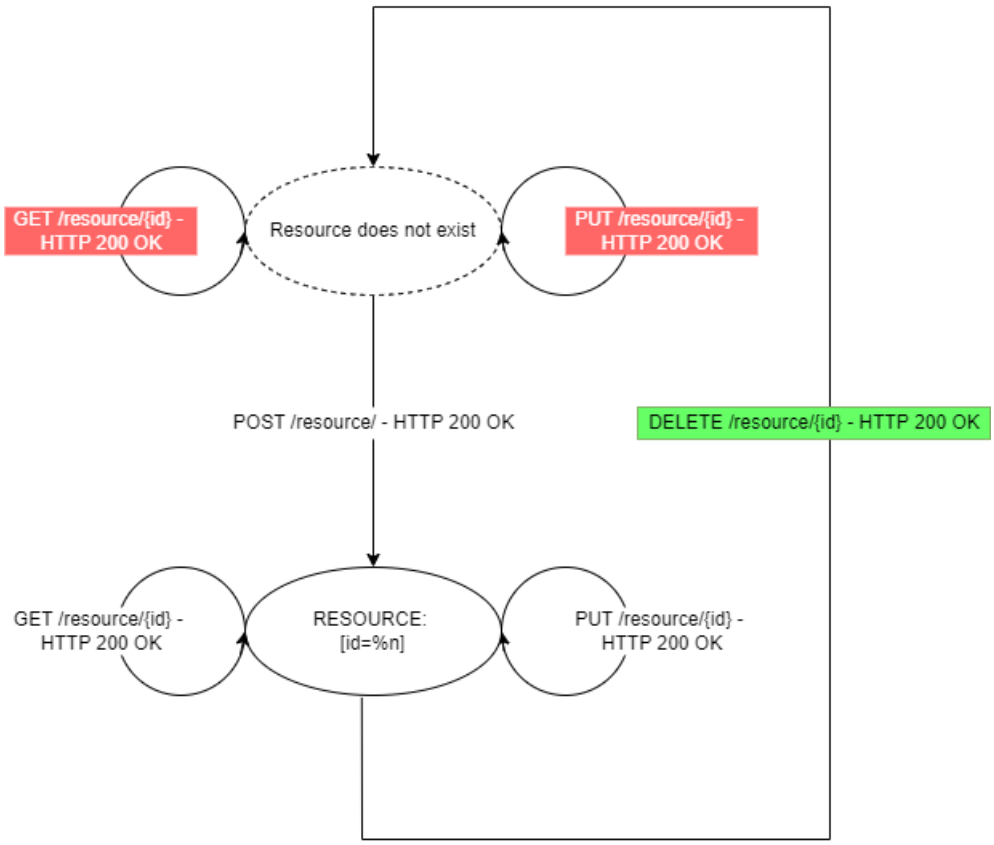

Рисунок 3.4 – Баг, ресурс доступний після видалення

Помилки бізнес логіки функцій в цілому вимагають більше інформації про ціль, ніж дано при "black box" тестуванні, а саме дані про потоки даних та потоки виконання коду в системі. Із заданими для розробки умовами провести тестування цього класу багів фазінгом не є можливим.

Помилки, при яких відбувається розкриття інформації, виявляються шляхом пошуку у тілах відповідей певних ключових слів, або просто порівняння їх із очікуваними. Такі перевірки можна проводити паралельно із пошуком інших багів, реєструючи нетипові відповіді на запити.

Вразливості автентифікації та авторизації виявляються фазінгом відповідних функцій API. Самі вразливості полягають в можливості вгадати, передбачити чи

обійти відповідні алгоритми перевірки введених користувачем даних. Сценарій наявності вразливостей автентифікації/авторизації виглядає так:

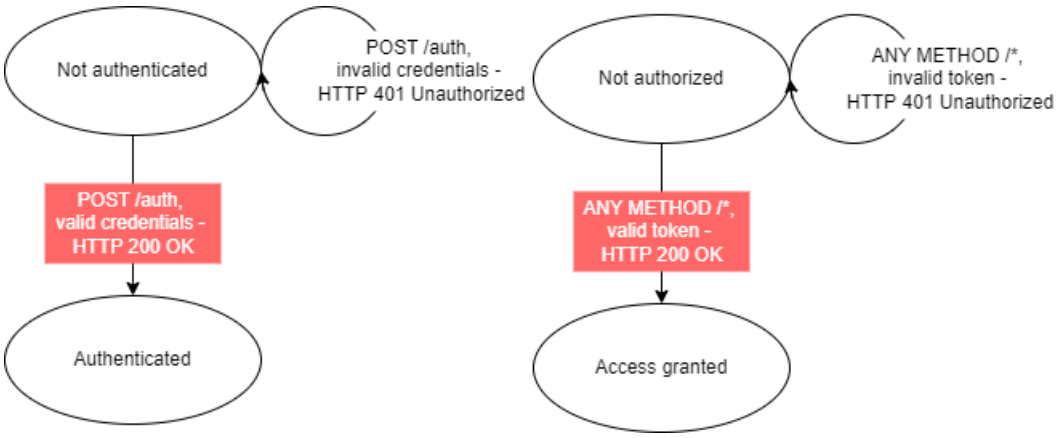

Рисунок 3.5 – Успішна компрометація автентифікації/авторизації

При тестуванні окремих функцій – тестуванні одного запиту – фазінг полягатиме в підстановці навантаження в усі поля тіла та заголовків цього запиту для будь-яких їх комбінацій. У випадку тестування послідовністю запитів поля фазінгу та їх комбінації залежать від конкретного випадку.

Для отримання ідеального результату комбінація фазінгу полів має бути кожне поле з кожним іншим. Також додається тестування відсутності окремих полів у комбінації кожне з кожним. Проте така методика означає побудову неймовірно великої кількості тестових запитів, а отже і неймовірно довготривалий процес фазінгу. Для розробленої методики комбінації фазінгу будуть оптимізовані для отримання максимальної результативності (шансу виявлення багу) при допустимій кількості побудованих запитів.

Розроблену методику фазінгу API можна сформулювати у вигляді таблиці визначених запитів та відповідей, що свідчитимуть про наявність багу/вразливості а також полів застосування фазінгу в них:

Таблиця 3.1 – Порядок проведення фазінгу в розробленій методиці

№	Сценарій	Тестові запити	Поля фазінгу	Комбінації фазінгу полів	Послідовності відповідей, що свідчать про баг
1.	Тестування окремої функції доступу до ресурсу	Відповідний запит доступу до ресурсу(POST, PUT, GET чи DELETE)	Параметри URL, поля тіла запиту(поля JSON схеми чи HTTP навантаження)	• усі поля одночасно; • кожне обов'язкове поле окремо;	500 Internal Server Error
2.	Тестування недоступності новоствореного ресурсу	POST	Поля тіла запиту(поля JSON схеми чи HTTP навантаження)	• усі поля одночасно; • кожне обов'язкове поле окремо;	200 OK
		PUT, GET, DELETE	Параметри URL, поля тіла запиту(поля JSON схеми чи HTTP навантаження)	• поле ідентифікатора визначене; • усі інші поля одночасно;	403 Forbidden, 404 Not Found, 500 Internal Server Error
3.	Тестування доступності щойно видаленого ресурсу	DELETE	Параметри URL, поля тіла запиту(поля JSON схеми чи HTTP навантаження)	• поле ідентифікатора;	200 OK
		GET, PUT	Параметри URL, поля тіла запиту(поля JSON схеми чи HTTP навантаження)	• поле ідентифікатора визначене; • усі інші поля одночасно;	200 OK, 500 Internal Server Error

Кінець таблиці 3.1

№	Сценарій	Тестові запити	Поля фазінгу	Комбінації фазінгу полів	Послідовності відповідей, що свідчать про баг
4.	Тестування стійкості автентифікації	GET, або POST	Заголовок Authorization, поля тіла запиту	• кожне поле окремо; • кожне поле з кожним;	200 OK, 500 Internal Server Error
5.	Тестування стійкості авторизації	Будь-який запит (при наявності краще запит на доступ до даних користувача за одним лише токеном авторизації)	Заголовок Authorization, поля тіла запиту	• кожне поле окремо; • кожне поле з кожним;	200 OK, 500 Internal Server Error

3.2 Практичне застосування розробленої методики на основі можливостей Burp Suite і Radamsa

Radamsa [7] – утиліта командного рядка, єдиним призначенням якої є перетворення вхідних даних випадковим чином – перестановки символів, додавання спеціальних символів чи зміни окремих бітів тексту і т. п. перетворення. Вона не є самостійним інструментом тестування, проте вважається корисним mutation-based фазером, яким ефективно користуються дослідники і тестери. Radamsa використовується для автоматизованого генерування навантаження для тестування кінцевих точок API, що будуть застосовані в спеціалізованих інструментах тестування веб-застосунків таких як wfuzz, OWASP ZAP, Burp Suite, і т. д. Всі ці

інструменти мають функції встановлення HTTP взаємодії із цільовим веб-сервером і перехоплення пакетів запитів-відповідей для подальшого їх розгляду, проте їх практична простота і комфорт у використанні відрізняються.

Burp Suite [8] є повноцінними інструментом тестування захищеності, проте тестування API в ньому є ручним і полягає в покроковому фазінгу певної функції API із заданим при налаштуванні навантаженням. Конкретно тестування захищеності REST API шляхом його фазінгу в Burp Suite виглядатиме так:

1) Використовуючи доповнення OpenAPIParser [9] із BAppStore тестер зчитує специфікацію цільового API і отримує перелік кінцевих точок API, які підлягають фазінгу.

2) Для кожної із кінцевих точок тестер коректує заголовки і тіло HTTP запиту, налаштовує поля, що будуть змінюватись при фазінгу та обирає підхід в застосуванні навантажень до полів для фазінгу.

3) Тестер надає джерела навантаження для полів фазінгу (список із файлу, випадкова генерація і т. д.) та обирає функції перетворення навантажень перед їх застосування до полів. Наприклад, джерелом навантаження для фазінгу може бути результат роботи Radamsa – у BAppStore наявне доповнення BRadamsa [10], що дозволяє в декілька кліків налаштувати генерацію даних для фазінгу через встановлену локально Radamsa.

4) Після вибору полів для фазінгу, навантаження для фазінгу та правил застосування навантаження при фазінгу тестер запускає фазінг та спостерігає HTTP відповіді. Тестер робить висновок про наявність вразливості, багу і т. д. по параметрам відповідей від сервера – по коду статусу, розміру пакета та наявності певних слів в пакеті відповіді.

Фазінг в Burp Suite налаштовується в розділі Intruder.

Щоб реалізувати розроблену методику із існуючими інструментами, основним з яких є Burp Suite, тестер має виконати наступні кроки:

1) Отримані за допомогою розширення OpenAPIParser згенеровані із специфікації запити перенаправити у розділ Intruder та запустити фазінг кожного з них. Навантаження для полів фазінгу обрати або генероване BRadamsa, або як було застосовано при практичних дослідженнях – вручну згенеровані локально встановленою Radamsa файли із навантаженнями, вихідними даними для яких обрати дані відповідного типу та формату для відповідного поля(тип дата, час, ім'я, uid, цілі числа, і т. д.). Комбінації для фазінгу виставити відповідно таблиці застосування розробленої методики з попереднього підрозділу. Комбінація усіх полів одночасно відповідає методу Pitchfork у Burp Suite; фазінг одного поля за раз – Sniper, а кожний з кожним – Cluster Bomb. Застосування Cluster Bomb обмежене вимогами допустимого часу на фазінг.

За таблицею застосування розробленої методики з попереднього підрозділу виконати пошук відповідей, що свідчать про наявність багів у функціях.

2) Із переліку запитів визначити множину наявних в цільовій системі ресурсів. Далі підкроки для кожного із визначених ресурсів:

2.1) Перенаправити усі запити на доступ до певного ресурсу *R* із OpenAPIParser до Intruder.

2.2) Виконати фазінг на запиті POST по полю ідентифікатора доступу до ресурсу R.

2.3) Виконати фазінг аналогічний у пункті (1) для запитів GET, PUT, потім знову GET ресурсу R.

2.4) Виконати фазінг на запиті DELETE по полю ідентифікатора доступу до ресурсу R з тим же навантаженням, що і на підкроці (2.2).

2.5) Повторити підкрок (2.3).

2.6) Отриманий список послідовностей відповідей від застосунку на запити POST→GET→PUT→GET→DELETE→GET→PUT→GET або залишити у вигляді відкритих вікон, або зберегти у файли.

2.7) Перезавантажити цільовий застосунок з чистого зображення, щоб скинути зроблені попереднім фазінгом зміни в ньому.

3) Для форми автентифікації провести фазінг відповідного поля із навантаженнями з поширеними логінами та паролями. Також провести фазінг того ж поля автентифікації із згенерованим Radamsa навантаженням. Приклад: для HTTP Basic Authentication застосувати тип навантаження Custom_iterator із попереднім кодуванням base64_encode, що дозволить задати списки логінів і паролів окремо і щоб вони автоматично конкатенувались з роздільником (":"), а результат кодувався base64.

4) При наявності обрати запит, що в якості змінного поля містить лише авторизаційне (поле авторизаційного токену) і застосувати фазінг з навантаженням з Radamsa. В залежності від формату авторизаційного токена, при виявленні (вгадуванні) формули його формування застосувати фазінг із спеціальним навантаженням(логіном, чи іншими даними про користувача, з яких може обчислюватися авторизаційний токен).

3) По кожному ресурсу оформити знайдені баги і вразливості у таблицю; вручну, або застосовуючи автоматизований аналіз шукати послідовності відповідей, що свідчитимуть про баг. Шаблони послідовностей відповідей у таблиці розробленої методики, POST→GET→PUT→GET – перевірка багу недоступності після створення, DELETE→GET→PUT→GET – доступності після видалення.

3.3 Аналіз практичного застосування RESTler fuzzer

Restler fuzzer від Microsoft Research [11] є stateful (враховує стан взаємодії із сервером) фазером REST API. За словами авторів це перший в своєму роді інструмент фазінгу REST API, що підтримує становість. Даний інструмент є

результатом дослідження в області ефективного фазінгу REST API хмарних сервісів і він імплементує спеціалізовані алгоритми пошуку багів в API. Опис цих алгоритмів та документацію можна знайти за посиланнями на Github репозиторій[12] та роботами Microsoft Research[13], [14]. В даній же роботі буде розглянуто застосування на практиці даного інструменту – описаний процес фазінгу від початку і до кінця і порівняно переваги і недоліки із іншими підходами до фазінгу REST API.

Із важливого про роботу інструмента Restler, що стосується застосування на практиці, Restler підтримує 4 режими-етапи роботи – compile, test, fuzz-lean і fuzz – названі у хронологічному порядку їх виконання.

- *Compile* – найважливіший етап налаштування фазінгу. Restler приймає на вхід OpenAPI специфікацію цільового REST API а на виході створює граматику для HTTP запитів в форматі .py, словник для фазінгу за замовчуванням в форматі .json, .json файл параметрів тестування та декілька інших json файлів конфігурацій.

Граматика та словник – основні структури, що цілком визначають фазінг REST API в Restler. Правильність згенерованої граматики перевіряється на етапі Test і після виявлення помилок правки вносяться у специфікацію API. Словник представляє собою розділені за типами даних списки вхідних даних, що будуть вставлені в поля для фазінгу в самому процесі тестування. Restler передбачає динамічну (в момент безпосереднього формування тіла HTTP запиту) випадкову генерацію контенту для словника, або збереження в .json файлі попередньо згенерованих даних для кожного із типів даних.

- *Test* – етап перевірки згенерованої на етапі Compile граматики. Цей етап представляє собою перевірку відповідей на HTTP запити до кінцевих точок API, що містяться в граматиці. Відповідь HTTP 404 означають відсутність певної кінцевої точки, і тестер дізнається про кількість таких помилкових описів функцій в граматиці за результатами Test. Тестер рекурсивно вносить правки в специфікацію, генерує граматику в Compile знову і знову виконує Test, допоки "покриття" (відсоток

існуючих і правильно описаних до усіх заданих кінцевих точок в граматиці) не буде задовільним.

На вхід даного етапу Restler вимагає файл із граматикою, словник, опис команди відновлення автентифікаційних даних для функцій API та параметри цільового веб-серверу, що потрібні для доступу до нього в мережі (IP та порт призначення).

- *Fuzz* - безпосередньо етап фазінгу. Тестер назначає вихідні параметри та конфігурації аналогічно етапу Test а також обирає застосовні "чекери" при фазінгу, ідейний метод обходу послідовностей запитів на вибір із тих, що пропонує сам Restler та обмеження по часу на фазінг.
- *Fuzz-lean* – режим повністю аналогічний режиму Fuzz, окрім меншої кількості та інтенсивності надсилання запитів – менш агресивний режим фазінгу, призначений для найшвидкого пошуку оптимальної кількості багів.

Restler для глибшого пошуку багів імплементує різні алгоритми побудови послідовностей запитів та їх контенту. На даний момент інструмент підтримує розроблені Microsoft Research алгоритми bfs-fast, bfs-cheap, bfs та random-walk. На практиці має сенс застосовувати різні алгоритми по черзі, керуючись витратами часу на фазінг а також різницю між можливими результатами у різних алгоритмів.

Пошук багів із Restler розділений на модулі (так звані "checkers") – кожний модуль призначений для пошуку багів і вразливостей певного класу, властивого API, які були наведені в попередньому розділі. Наявні модулі із самоописовими назвами: leakagerule, resourcehierarchy, useafterfree, namespacerule, invaliddynamicobject та payloadbody. На практиці має сенс застосовувати усі модулі одночасно, оскільки результати їх роботи не перетинаються і лише додають до множини виявлених багів.

Результати фазінгу із Restler оформлені у вигляді текстових файлів логів запитів, на яких був зафіксований баг та .json файл із описом структури директорії із цими логами та їх форматами. Логи містять необроблені описи HTTP запитів та

описи помилок, що вони викликали на веб-сервері та інструкції із відтворення цих тесткейсів засобами Restler.

Практичні результати тестування піддослідного RESTful веб-застосунку будуть розглянуті в останньому підрозділі даного розділу.

Забігаючи наперед, на підставі проведеного фазінгу із Restler на практиці було зроблено висновок про певні добрі та погані сторони практичного застосування даного інструменту для фазінгу REST API:

Таблиця 3.2 – Позитивні та негативні якості Restler, виявлені під час застосування на практиці

Позитивні якості	Негативні якості
Простота дебагу помилок при попередньому налаштуванні фазінгу (етап Test і аналіз логів після нього)	Відсутність вбудованих засобів попередньої генерації словників різних типів, або інтеграції з такими; відсутність засобів копіювання існуючих списків навантажень у файл словника – тестер самостійно має створювати .json словник програмно, або копіюючи вручну контенти існуючих файлів в потрібні поля
Автоматизованість пошуку багів окремих функцій а також більш глибших вразливостей роботи всього API	Відсутність графічного інтерфейсу, а особливо відсутність відображення прогресу фазінгу
Представлення результатів фазінгу у форматі придатному для автоматизованого пост-аналізу	Відсутність вбудованих засобів створення файлу налаштувань для майбутніх запусків тестування (settings.json)
	Відсутність гнучкості в налаштуванні автентифікації/авторизації запитів при фазінгу – тип та значення автентифікуючих/автеризуючих даних задаються одні для усіх кінцевих точок

	граматики, що тестуються

3.4 Порівняльний аналіз тестування із RESTler та тестування розробленою методикою із Burp Suite та Radamsa

В якості піддослідного RESTful застосунку буде взято навмисно розроблений вразливим Hackazon [16] від AppSpider. Після встановлення за надаваною інструкцією з репозиторію на Github застосунок буде розгорнутий локально за адресою http://localhost:80/api (базовий URL REST API застосунку).

Примітка – при встановленні застосунку з'являються помилки створення бази даних, пов'язані із тим, що mysql вважає за помилку застосування до двох і більше стовпчиків однієї таблиці параметрів DEFAULT CURRENT_TIMESTAMP, тому для успішного встановлення ці суперечності були видалені із скриптів mysql. На подальше тестування API дана зміна не впливає.

Для налаштування фазінгу потрібно завантажити специфікацію API – swagger.json у робочій директорії застосунку. В останній версії репозиторію в специфікації (файлі swagger.json) також наявні помилки: тег "baseURL" має бути встановленим "", теги "description" певних кінцевих точок не відповідають дійсності. Для оптимізації ручного аналізу із Burp Suite аналітичним методом було виведено коректні описи тих функції, в яких "describtion" був неправильний.

Після налаштування Restler так, як було описано в попередніх підрозділах, та проведення фазінгу (метод пошуку random-walk та bfs, причому перший дав більше результатів) було отримано результати у вигляді виявлених багів REST API:

Таблиця 3.3 – Виявлені баги при фазінгу із Restler

№	Застосований модуль	Кінцева точка з виявленим багом	HTTP відповідь	Можлива причина багу
1.	main_driver_500	GET /api/order? page={}& per_page={}	HTTP code 500 – Division by 0	Відсутність перевірки наявності вхідних даних та помилки типізації
		GET /api/product? page={}& categoryID={}	HTTP code 500 – Error	Відсутність перевірки наявності вхідних даних та помилки типізації
		GET /api/customerAddress? page={}& per_page={}	HTTP code 500 – Division by 0	Відсутність перевірки наявності вхідних даних та помилки типізації
		GET /api/category? page={}& per_page={}	HTTP code 500 – Error	Відсутність перевірки наявності вхідних даних та помилки типізації
		POST /api/orderAddresses	HTTP code 500 – Error	Відсутність фільтрування вхідних даних та перевірок типів перед обробкою
2.	PayloadBodyChecker_500	POST /api/order	HTTP code 500 – Error	Множинні некоректні обробки полів json схеми
		POST /api/orderAddresses	HTTP code 500 – Error	Множинні некоректні обробки полів json схеми
		GET /api/category? page={}& per_page={}	HTTP code 500 – Division by 0	Відсутність перевірки наявності вхідних даних
		GET /api/customerAddress? page={}& per_page={}	HTTP code 500 – Division by 0	Відсутність перевірки наявності вхідних даних та помилки типізації
		GET /api/order? page={}& per_page={}	HTTP code 500 – Division by 0	Відсутність перевірки наявності вхідних даних та помилки типізації

Кінець таблиці 3.3

№	Застосований модуль	Кінцева точка з виявленим багом	HTTP відповідь	Можлива причина багу
3.	LeakageRule	-		
4.	ResourceHierarchy	-		
5.	UseAfterFree	-		
6.	NamespaceRule	-		
7.	InvalidDynamicObject	-		

Також застосовуючи розроблену методику при тестуванні цільового веб-застосунку із Burp Suite, використовуючи навантаження, згенеровані із Radamsa, було отримано дані про баги цільового REST API:

Таблиця 3.4 – Результати тестування із Burp Suite і Radamsa

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
1.	/api/category	GET category: ?page={}&per_page={}	page, per_page	400, 200, 500("Division by zero", "Error")	null
		GET category: /{categoryID}	categoryID	400, 404, 200	Даний метод доступу був не задокументований в специфікації.

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
		POST category: схема json	усі поля схема json	200	Даний метод доступу був не задокументований в специфікації. Некоректні дані запитів були коректно перетворені при створенні нових ресурсів.
		DELETE category: /{categoryID}	categoryID	400, 404, 405, 200	Даний метод доступу був не задокументований в специфікації. categoryID при створенні ресурсу призначає не користувач API, а сам сервер.
		PUT category: /{categoryID},схема json	categoryID, усі поля схема json	400, 404, 405, 200	Даний метод доступу був не задокументований в специфікації.
Тестування наявності багів життєвого циклу ресурсу багів не виявило.					
2.	/api/product	GET product: ?page={}&categoryID={}	page, categoryID	400, 404, 200, 500("Error")	page=0&categoryID=0 викликає HTTP Error 500.
		GET product: /{productID}? categoryID={}	productID, categoryID	400, 404, 200	null
		DELETE product: /{productID}	productID	400, 404, 405, 200	null

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
		POST category: схема json	categoryID, name, description, Price, items_sold, managerID	200	Майже усі поля не обов'язкові – при фазінгу створилося безліч "порожніх" ресурсів
		PUT category: /{productID}, схема json	categoryID	400, 405, 405, 200	null
Тестування наявності багів життєвого циклу ресурсу багів не виявило.					
3.	/api/customerAddress	GET customer address: ?page={}&per_page={}	page, per_page	400, 500 ("Division by zero", "Error")	Різні запити повертають одну і ту ж відповідь – список із одного ресурсу, хоча в БД їх існує багато.
		POST customer address: схема json	усі поля json крім phone, region та zip	200	Усі запити були оброблені та ресурси створені із попередньою валідацією вхідних даних.
Тестування наявності багів життєвого циклу ресурсу не було можливим – для ресурсу відсутні за специфікацією та не виявлені вручну методи видалення та редагування.					

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
4.	/api/cart	GET cart: /{cartID}	cartID	404	Навіть коректні cartID повертають помилку 404. Існує URL /api/cart?page={}& per_page={} – не задокументований у специфікації. Ця кінцева точка при фазінгу повертає відповіді HTTP 500.
		GET cart: /my?uid={cartUID}	cartUID	400, 414 та 200	Запити з відповіддю 200 успішно створювали нові ресурси.
		POST cart: схема json	id, uid, customerID, customer_email, customer_is_guest	200	Даний метод доступу був не задокументований в специфікації.
		PUT cart: /{cartID}, схема json	id, uid, customerID, customer_email, customer_is_guest	400, 414, 200	Цільовий ресурс визначається cartID, а cartID призначає сам застосунок – це завжди послідовні цілі числа.
		DELETE cart: /{cartID}	cartID	400, 404, 200	Аналогічно PUT cart. Тестування подальшого доступу до видалених ресурсів не виявило багів.
Тестування наявності багів життєвого циклу ресурсу багів не виявило.					

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
5.	/api/cartItems	POST cartItems: ?uid={ cartUID }, схема json	uid, cart_id, name, product_id, id, price, qty	400, 414, 200, 500("Error")	Запити додавання нових items до неіснуючих cart повертали 400 та 500 замість очікуваних 404.
		PUT cartItems: /{cartItemsID}, схема json	cartItemsID, cart_id, name, product_id, price, qty	400, 404, 405 414, 200	Даний метод доступу був не задокументований в специфікації.
		DELETE cartItems: /{cartItemsID}	cartItemsID	400, 404, 200	Даний метод доступу був не задокументований в специфікації.
		GET cartItems: /{cartItemsID}	cartItemsID	400, 404, 200	Даний метод доступу був не задокументований в специфікації.
Тестування наявності багів життєвого циклу ресурсу ускладнене залежністю створення даного ресурсу із існуванням іншого ресурсу(cart), а точніше – залежить від невідомого uid.					
6.	/api/order	POST order: схема json	comment, coupon_id, customer_firstname, customer_lastname, customer_id, orderItems, id	200, 500("Error")	null

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
		GET order: ?page={} &per_page={}	page, per_page	400, 200, 500 ("Division by zero", "Error")	null
		GET order: {orderID}?page={} &per_page={}	orderID, page, per_page	400, 404, 500 ("Division by zero", "Error")	null
		DELETE order: {orderID}	orderID	400, 404, 405, 200	Не задокументований метод в специфікації.
		PUT order: /{orderID}, схема json	orderID	400, 404, 405, 200	Не задокументований метод в специфікації.
Тестування наявності багів життєвого циклу ресурсу помилкових ситуацій не виявило.					
7.	/api/orderItems	POST oderItems: схема json	cart_id, created_at, name, order_id, product_id, price, qty, id	200	Неправильно сформовані дані у всіх випадках були коректно зведені до відповідних типів/форматів, не викликаючи помилкових ситуацій.
		PUT oderItems: /{orderItemsID}, схема json	orderItemsID , cart_id, created_at, name, order_id, product_id, price, qty	400, 404, 405, 200	Не задокументований метод в специфікації.

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
		GET oderItems: /{orderItemsID}	orderItemsID	400, 404, 405, 200	Не задокументований метод в специфікації.
		DELETE oderItems: /{orderItemsID}	orderItemsID	400, 404, 405, 200	Не задокументований метод в специфікації.
Тестування наявності багів життєвого циклу ресурсу помилкових ситуацій не виявило.					
8.	/api/orderAddresses	POST order address: схема json	address_line_1, address_type, country_id, customer_id, full_name, order_id, id	200, 500 ("Error")	null
		PUT order address: /{ orderAddressesID}, схема json	orderAddressesID , address_line_1, address_type, country_id, customer_id, full_name, order_id, id	400, 404, 200	null
		GET order address: /{ orderAddressesID }	orderAddressesID	400, 404, 200	null
		DELETE order address: /{orderItemsID}	orderAddressesID	400, 404, 200	null
Тестування наявності багів життєвого циклу ресурсу помилкових ситуацій не виявило.					
9.	/api/contactMessages	POST contact message: схема json	усі поля схеми json	200	За специфікацією в запиті відсутній заголовок авторизації.
		PUT contact message: /{ contactMessagesID}, схема json	contactMessagesID	400, 404, 405, 200	Не задокументований метод в специфікації.

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
		GET contact message: /{ contactMessagesID}	contactMessagesID	400, 404, 405, 200	Не задокументований метод в специфікації.
		DELETE contact message: /{ contactMessagesID}	contactMessagesID	400, 404, 405, 200	Не задокументований метод в специфікації.
Тестування наявності багів життєвого циклу ресурсу помилкових ситуацій не виявило.					
10.	/api/user	GET user: /me	Токен авторизації	400, 401	Отримати коректний токен авторизації дозволить лише повний перебір (brute force)
		PUT user: /{user_id}, схема json	user_id, id, username, email, oauth_uid, active	400, 404, 405, 200	Вхідні дані типізуються та форматуються коректно, проте відсутня валідація неадекватних вхідних даних, що автентифікують користувача(наприклад – порожній логін, і т. д.)
		POST user: схема json	user_id, id, username, active	200	Не задокументований метод в специфікації. Нові користувачі створюються із порожнім значенням хешу пароля – тобто вони недоступні для входу

Продовження таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
		GET user: /{userID}	userID	404, 200	Не задокументований метод в специфікації. Метод не дозволяє перелічити всіх існуючих користувачів (повертає відповідь 200, але список порожній) а також не надає інформацію про користувача навіть по коректному userID (крім даних користувача, що авторизує запит).
		DELETE user: /{userID}	userID	403, 404	Не задокументований метод в специфікації. Користувачам заборонено видаляти ресурс user.
По факту ресурс user невидаляємий, що може свідчити про наявність багу життєвого циклу. Насправді ж можливо це передбачено моделлю обмеження доступу до ресурсів особливо важливих для забезпечення безпеки REST.					

Кінець таблиці 3.4

№	Кінцева точка	Методи доступу до ресурсів	Поля для фазінгу	HTTP відповіді	Примітки
11.	/api/auth	GET auth token with basic auth	Authorization: Basic	400, 401	Навантаження використовувалось в необробленій формі(raw), закодоване base64 а також крім згенерованого Radamsa застосовувались списки популярних логінів/паролів.
Тестування наявності багів життєвого циклу ресурсу неможливе, оскільки відповідні функції доступу до ресурсу не відповідають концепціям.					

Крім тестування фазінгом інструментарій BurpSuite + розширення до нього дозволяють перевірити наявність інших вразливостей, що не виявляються одним лише фазінгом. Наприклад, для даної цілі тестування вручну було виявлено протиріччя форматів контенту, очікуваного при запиті та отриманого у відповіді(при певних запитах до REST API сервер відповідає у форматі HTML, а не в очікуваному JSON); було виявлено відсутність будь-якої моделі розмежування доступу – будь-який користувач API має доступ до будь-яких ресурсів, в тому числі і до даних інших користувачів;

На основі результатів тестування обома підходами та отриманому досвіду з їх практичного застосування можна провести порівняльний аналіз:

Таблиця 3.5 – Порівняльний аналіз фазінгу двома інструментаріями

Критерій	Restler fuzzer	Burp Suite з Radamsa
Простота та практичність налаштування	• Невелика кількість налаштувань • Обмеженість сумісності з іншими інструментами • Відсутність графічного інтерфейсу налаштування • Недостатня документація	• Майже повні можливості налаштування фазінгу окремого запиту • Доповнені можливості у вигляді розширень та сумісність з іншими інструментами • Відсутність налаштування послідовності запитів
Витрати часу на налаштування та фазінг	• Час налаштування залежить від якості специфікації API та сторонніх засобів генерації навантаження для фазінгу • Час фазінгу налаштовується	• Довготривале налаштування при методичному підході • У версії Burp Suit Community швидкість фазінгу обмежена
Покриття типів багів та вразливостей REST API	• "Checkers" покривають 7 типів багів	• Реалізація методики дає змогу покрити 6 класів вразливостей REST API

Кінець таблиці 3.5

Критерій	Restler fuzzer	Burp Suite з Radamsa
Результативність	• PayloadBodyChecker_500: 49 та main_driver_500: 5 – всього 54. Кожний баг – окрема помилка в реалізації функції, що викликає помилкову ситуацію	• 8 помилок реалізацій окремих функцій, 20 незадокументованих в специфікації функцій та 2 функції задокументовані, але недоступні ні при яких умовах. 1 запит із зайвим функціоналом та 1 запит із багом недоступності новоствореного ресурсу (недоступість новоствореного користувача).
Простота та практичність аналізу результатів фазінгу	• Результати автоматично зберігаються у формі, придатній для автоматизованої генерації звіту • Графічної візуалізації виявлених багів немає • Можливе відтворення багу з ручним налаштуванням	• У версії Burp Suite Community аналіз результатів ручний, проте швидкий за рахунок сортування відповідей. У платній версії Professional – збереження у файл та автоматизований аналіз • Результати придатні для пошуку помилок і за межами тих, на які направлена розроблена методика. Опис помилкової ситуації при появі багу повний.

Висновки до розділу 3

Було розроблено методику фазінгу REST API і адаптовано до застосування із набором інструментів Burp Suite та Radamsa. Методика була сформульована у вигляді послідовності кроків із налаштування та проведення фазінгу окремих цільових функцій та послідовностей їх викликів, направленого на виявлення типових багів та вразливостей REST API. В результаті практичного дослідження застосування цієї методики із зазначеними інструментами ручного аналізу, а також існуючим інструментом автоматизованого тестування Restler fuzzer було сформовано звіти та проведено порівняльний аналіз двох підходів до фазінгу REST API. Два інструментарії мають свої переваги та недоліки в практичному застосуванні, проте вони знаходять ефективне застосування на певних стадіях тестування веб-застосунків, а саме Restler можна позиціонувати як засіб автоматизованої перевірки наявності багів розроблюваних версій та оновлень API застосунків, в той час як застосування методики із інструментами ручного аналізу в силу своєї витратності часу слід застосовувати як частину тестування на проникнення експлуатаційної версії застосунку. На основі отриманого досвіду із практичного фазінгу REST API можна зробити висновок, що повна свобода налаштування при ручному аналізі для тестера більш цінна, ніж автоматизоване тестування, оскільки вона дає більше інформації про виявлені баги та більше результатів в цілому.

ВИСНОВКИ

В ході виконання даної роботи було розглянуто різні типи та окремі методи тестування захищеності веб-застосунків та виділено роль фазінгу серед інших існуючих методів. В якості об'єкту дослідження було розглянуто REST API веб-застосунків та баги і вразливості, що тестери виявляють в ньому. Було визначено множину вразливостей та багів, що можливо виявити в REST API із застосуванням фазінгу та виокремлено цільові компоненти, що стають ціллю тестера в ході тестування. Практична частина робота полягала в розробці методики застосування фазінгу до REST API для виявлення попередньо визначених багів та виконанню фазінгу на практиці за допомогою двох різних існуючих інструментів. В результаті було сформульовано порядок процедури фазінгу для розробленої методики, виконано фазінг із застосуванням Restler fuzzer та із застосуванням стеку Burp Suite, Radamsa з подальшим формуванням звітів про результати аналізу. На основі отриманого досвіду застосування розглянутих інструментів було проведено порівняльний аналіз інструментів та підходів, визначено їх поточні добрі якості та бажані майбутні покращення. В результаті мета даної роботи була досягнута, а поставлені завдання успішно виконані. Літературні джерела, що безпосередньо вплинули на написання даної роботи: [16], [17], [18].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1) Топ 10 ризиків безпеки веб-застосунків за даними OWASP Foundation [Електронний ресурс] - Режим доступу: https://owasp.org/www-project-top-ten/ - 6.06.2022 р.

2) Savita B. Chavan, Dr. B. B. Meshram// Classification of Web Application Vulnerabilities: International Journal of Engineering Science and Innovative Technology(IJESIT), volume 2, issue 2 – березень 2013 р.

3) Збірник практик безпеки веб-застосунків Web Security Testing Guide(WSTG) від OWASP Foundation [Електронний ресурс] - Режим доступу: https://owasp.org/www-project-web-security-testing-guide/ - 6.06.2022 р.

4) Список тестів безпеки веб-застосунків від Certified Secure (Certified Secure Checklists) [Електронний ресурс] - Режим доступу: https://www.certifiedsecure.com/checklists/cs-web-application-security-test.pdf - 6.06.2022 р.

5) Fielding, Roy Thomas.// Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

6) За даними джерела OWASPS-Top-10 [6] вразливостей, специфічних для REST, існують можливі слабкості цих систем

7) Домашня сторінка проекту Radamsa на Gitlab [Електронний ресурс] – Режим доступу: https://gitlab.com/akihe/radamsa – 29.05.2022 р.

8) Домашня сторінка Port Swigger, Burp Suite [Електронний ресурс] – Режим доступу: https://portswigger.net/burp – 29.05.2022 р.

9) Домашня сторінка проекту Swurg на Github (OpenAPI парсера у вигляді розширення для Burp Suite)[Електронний ресурс] – Режим доступу: https://github.com/portswigger/openapi-parser – 29.05.2022 р.

10) Домашня сторінка проекту Bradamsa на Github (Генератор навантаження з Radamsa у вигляді розширення для Burp Suite)[Електронний ресурс] – Режим доступу: https://github.com/portswigger/bradamsa – 29.05.2022 р.
11) Сторінка на Microsoft Research, присвячена RESTler fuzzer Suite [Електронний ресурс] – Режим доступу: https://www.microsoft.com/en-us/research/publication/restler-stateful-rest-api-fuzzing/ – 29.05.2022 р.
12) Домашня сторінка проекту Restler fuzzer на Github/ вихідний код та інструкція з встановлення [Електронний ресурс] – Режим доступу: https://github.com/microsoft/restler-fuzzer – 29.05.2022 р.
13) Vaggelis Atlidakis, Patrice Godefroid, Marina Polishchuk// RESTler: Stateful REST API Fuzzing [Електронний ресурс] - Режим доступу: https://patricegodefroid.github.io/public_psfiles/icse2019.pdf – 29.05.2022 р.
14) Vaggelis Atlidakis, Patrice Godefroid, Marina Polishchuk// Checking Security Properties of Cloud Service REST APIs [Електронний ресурс] - Режим доступу: https://patricegodefroid.github.io/public_psfiles/icst2020.pdf – 29.05.2022 р.
15) Домашня сторінка проекту Hackazon (A modern vulnerable web app) на Github/ вихідний код та інструкція з розгортання [Електронний ресурс] – Режим доступу: https://github.com/rapid7/hackazon – 29.05.2022 р.
16) Tony Hsiang-Chih Hsu Practical Security Automation and Testing [Текст]/ Tony Hsiang-Chih Hsu – 2019 р. – 438 с.
17) Richa Gupta Hands-on Penetration Testing for Web Applications [Текст]/ Richa Gupta – 2021 р. – 341 с.
18) Ari Takanen Fuzzing: the Past, the Present and the Future [Текст]/ Ari Takanen – 11 с.