ECS 235A Computer Information and Security

Aditya Parab; aparab@ucdavis.edu
Kshitij Sinha; kxsinha@ucdavis.edu

Introduction

In the contemporary digital landscape, web applications play a pivotal role in facilitating numerous online activities, from financial transactions and social networking to collaborative workspaces. However, the growing reliance on web applications has made them prime targets for individuals with malicious intent. These attackers exploit vulnerabilities in web applications to compromise data security, steal identities, and disrupt essential services. This report aims to comprehensively document the prevalent vulnerabilities in web applications and propose effective mitigation strategies.

As our reliance on web applications continues to grow, addressing and mitigating common vulnerabilities is imperative to safeguarding sensitive information and ensuring the seamless functionality of online services. By implementing robust security measures, staying informed about emerging threats, and fostering a culture of security awareness, organizations can significantly reduce the risk of web application vulnerabilities and enhance overall cybersecurity.

SQL injection and Cross-Site Scripting (XSS) are prevalent forms of injection attacks where malicious code is inserted into input fields, leading to unauthorized access or manipulation of data. Cross-Site Scripting (XSS) is a type of security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. These scripts can be executed in the context of a user's browser, potentially leading to the theft of sensitive information, session hijacking, or defacement of websites. SQL Injection is a technique where attackers exploit vulnerabilities in a web application's database layer to execute arbitrary SQL code. This can lead to unauthorized access, manipulation, or deletion of data, as well as potential exposure of sensitive information.

To mitigate these vulnerabilities, employing parameterized queries and input validation to sanitize user inputs, helps in reducing the risk of injection attacks. Input validation is a critical aspect of securing web applications, and it involves verifying and sanitizing user inputs to ensure that they meet specific criteria before being processed by the application.

By implementing robust input validation practices, web developers can significantly reduce the attack surface of their applications and enhance overall security. It's essential to adopt a comprehensive approach, combining different validation techniques and staying informed about emerging threats to create a strong defense against malicious inputs.

In this project, we have implemented an input validation wrapper upon a locally hosted website. This website is immune to any SQL injection as well as any XSS attacks. We first make a vulnerable web application and simulate SQL injection and XSS attacks. These attacks were successful which helped us understand the key complexities of these attacks and propelled us to make an Input Validation Wrapper that sanitizes the input before processing it thus avoiding any SQL injection and XSS attacks. Additionally, we analyze the performance tradeoffs with or without the wrapper and compare the results. Our code can be found here: https://github.com/aditya-parab/input-validation-ecs235a

Project Objectives and Methodology

Web application security is vital for protecting data and services, preventing unauthorized access, data breaches, financial losses, reputational harm, and legal issues. Web applications face various threats like DDoS attacks, XSS, SQL injection, CSRF, and Clickjacking, which can result in data theft, service interruptions, website defacement, and sensitive information being compromised.

"*How does the implementation of an input validation wrapper in a web application impact the security of web applications in mitigating common vulnerabilities, and what are the performance trade-offs?*"

We have tried to answer the above question in detail during this project and elaborately discuss our findings and results. Let us talk about the vulnerabilities.

Types of vulnerabilities

Here we will focus on SQL injections and Cross-Site Scripting (XSS), which are the two most common types of attacks that target web applications.

1. **SQL Injection**
   SQL injection (SQLi) is a type of web security vulnerability that may permit an unauthorized party to control the queries an application sends to its database. Exploiting this vulnerability allows the attacker to access sensitive data that would normally be inaccessible. This may involve viewing information belonging to other users or any data accessible to the application. Additionally, the attacker can potentially alter or delete this data, leading to lasting modifications in the application's content or functionality.

2. **XSS Attack**
   Cross-site scripting (XSS) attacks occur when untrusted data enters a web application, usually via a web request (attackers can inject malicious scripts into web pages viewed by other users), and is subsequently included in dynamic content sent to a web user without prior validation for malicious content.[1] Malicious content, often in the form of JavaScript or executable code, can compromise user data, and session information, and take control of the user's browser. This leads to various nefarious activities by attackers using the vulnerable website as a disguise.

The Role of Input Validation

One approach to enhancing the security of web applications is the implementation of input validation. Input validation is the process of examining and verifying user inputs to ensure they conform to expected formats and data ranges before they are processed by the application. Input validation can help to prevent XSS and SQL injection attacks in the following ways:

SQL injection: Input validation can be used to validate all user input that is used in database queries. This prevents malicious SQL code from being injected into the queries.

XSS: Input validation can be used to escape special characters in user input. This prevents malicious JavaScript code from being executed.

Our Objectives

We make a web application that verifies if a user is a valid admin and allows the user to record their address.

- Implement a Vulnerable Web Application
  - basic login page with known security vulnerabilities, especially to SQL injections and XSS attacks.
- Design and Develop an Input Validation wrapper
  - Middleware to sanitize and validate user input before processing → Mitigate SQL injection & XSS attacks
- Conduct experiments and record outcomes
- Measure and analyze

Old version

We first make a web application that is vulnerable to SQL injection. Let's call this version as “Old version”. The old version consists of two files: index.html and app.js. Let's break down the key sections of the code:

index.html

Form structure

```
<form action="/login" method="post" onsubmit=" return validateForm(event)">
  <!-- Username input -->
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" required>
  <!-- Password input -->
  <label for="password">Password:</label>
  <input type="password" id="password" name="password" required>
  <!-- Address input -->
  <label for="address">Address:</label>
  <input type="text" id="address" name="address" required>
  <!-- Submit button -->
  <input type="submit" value="Login">
</form>
```

- Username, password, and address input fields within a form.
- Basic HTML structure with required attributes.

JavaScript Validation

```
function validateForm(event) {
  event.preventDefault();  // Prevent form submission by default

  // Get form elements
  var usernameInput = document.getElementById('username');
  var passwordInput = document.getElementById('password');
  var addressInput = document.getElementById('address');

  // Simulate server-side validation
  // ...

  return false; // Prevent the form from submitting
}
```

JavaScript function validateForm prevents the form from submitting by default. Gets form elements and simulates server-side validation.

app.js

Database initialization

```
const db = new sqlite3.Database(':memory:');
db.serialize(() => {
  db.run('CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT, password TEXT)');
  db.run('INSERT INTO users (username, password) VALUES ("admin",
"admin123")');
});
```

Sets up an in-memory SQLite database and creates a users table and inserts a sample user with username “admin” and password “admin123".

Vulnerable Login Route:

```
app.post('/login', (req, res) => {
  const params = new URLSearchParams(req.body);
  const username = params.get('username');
  const password = params.get('password');
  const address = params.get('address');

  db.all(`SELECT * FROM users WHERE username = '${username}' AND password =
'${password}'`, (err, rows) => {
    if (err) {
      return console.error(err.message);
    }

    if (rows.length > 0) {
      res.send(`Login successful \nUsername: ${username} and Address: ${address}`);
    } else {
      res.send('Login failed');
    }
  });
});
```

Vulnerabilities expected in this version:

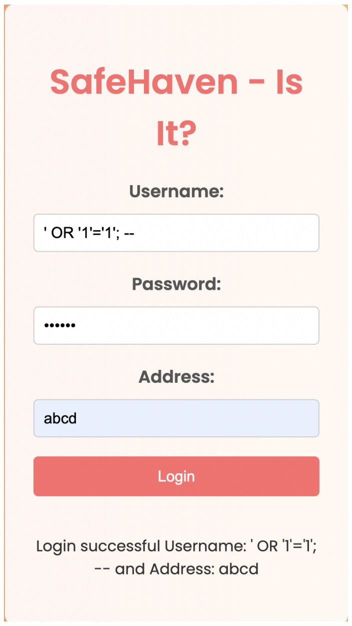
1. SQL Injection:

The SQL query in the login route uses string concatenation to include user input directly in the query. If an attacker inputs `' OR '1'='1'; --` in the username field, it would make the query always true, allowing unauthorized access.

2. XSS Attack:

The address field in the form is not sanitized or validated on the server side. If an attacker inputs `<img src="https://i.imgur.com/bCVv2A2.jpg" onerror="alert('XSS Attack!')" width="100px">`, it will be rendered as HTML when displayed, executing the embedded JavaScript.

Example of SQL injection


SafeHaven - Is It?

Username:

' OR '1'='1'; --

Password:

••••••

Address:

abcd

Login

Login successful Username: ' OR '1'='1'; -- and Address: abcd

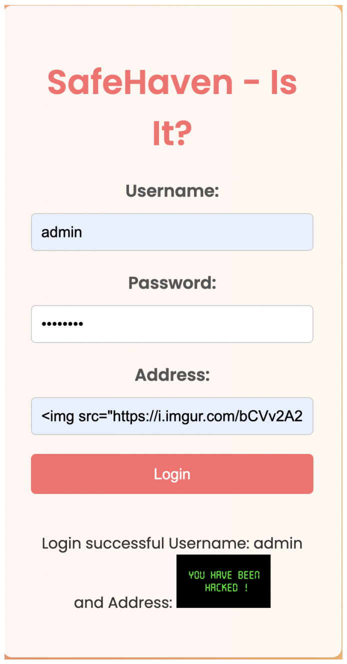
Fig: Login page with a successful SQL injection

Here the attacker is allowed entry despite inputting an invalid username, because the input is rendered as SQL and it would make the query always true, allowing unauthorized access.
The original SQL query was:
`SELECT * FROM users WHERE username = '${username}' AND password = '${password}'`

The SQL query in our program after the vulnerable username input becomes:
SELECT * FROM users WHERE username = '' OR '1'='1'; --' AND password = '${password}';

In this altered query, the condition '1'='1' always evaluates to true, effectively bypassing the original authentication check. The web application is susceptible to a SQL injection attack, wherein the attacker exploits a vulnerability in the input processing mechanism. When a user attempts to log in with a potentially invalid username, the application doesn't effectively validate or sanitize the input. Consequently, an attacker can input specially crafted characters that manipulate the SQL query's logic, making it always evaluate to true. This manipulation effectively bypasses the intended authentication mechanism, granting the attacker unauthorized access to the application.

Example of XSS attack



SafeHaven - Is It?

Username: admin

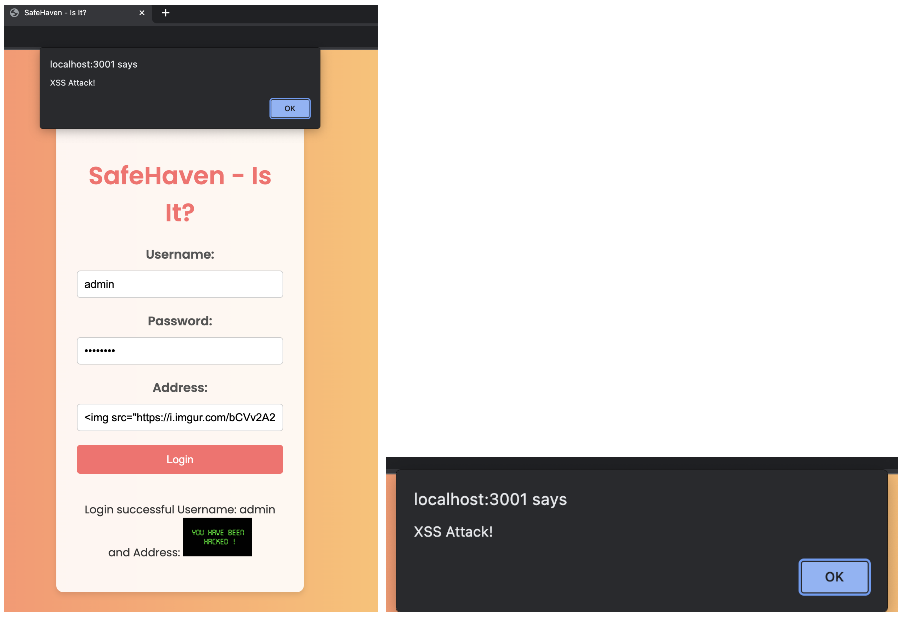
Password: ••••••••

Address: <img src="https://i.imgur.com/bCVv2A2

Login successful Username: admin and Address:

Fig: Login page with a successful XSS attack

Here the attacker includes site-altering html data in the address field, which allows the user to make unauthorized changes to the UI of the site. In this case an image with "YOU HAVE BEEN HACKED !" has been rendered. If you click on the loaded image, an onclick prompt will appear that says "XSS attack!" signifying that an XSS attack has occurred.



localhost:3001 says

XSS Attack!

SafeHaven - Is It?

Username: admin

Password: ••••••••

Address: <img src="https://i.imgur.com/bCVv2A2

Login successful Username: admin and Address:

localhost:3001 says

XSS Attack!

New version

Effect of the Input Validation Wrapper
In app.js, a middleware named sanitizeInput is introduced using the express-validator library. The wrapper enhances the security of the application by validating and sanitizing user inputs. The “sanitizeInput” middleware ensures that potentially malicious content in the username, password, and address fields is sanitized before processing. This helps prevent SQL injection and XSS attacks.

1. **Usage of Parameterized Queries:**

```
db.all(
  'SELECT * FROM users WHERE username = ? AND password = ?',
  [username, password],
  (err, rows) => {
    // ...
  }
);
```

In the vulnerable login route, parameterized queries (user inputs are treated as data rather than executable code) are employed to prevent SQL injection.

2. **Sanitizing for XSS Attack**

The “xss” library is utilized for HTML entity encoding. This library helps encode special characters in user inputs, ensuring that they are treated as plain text rather than executable code.

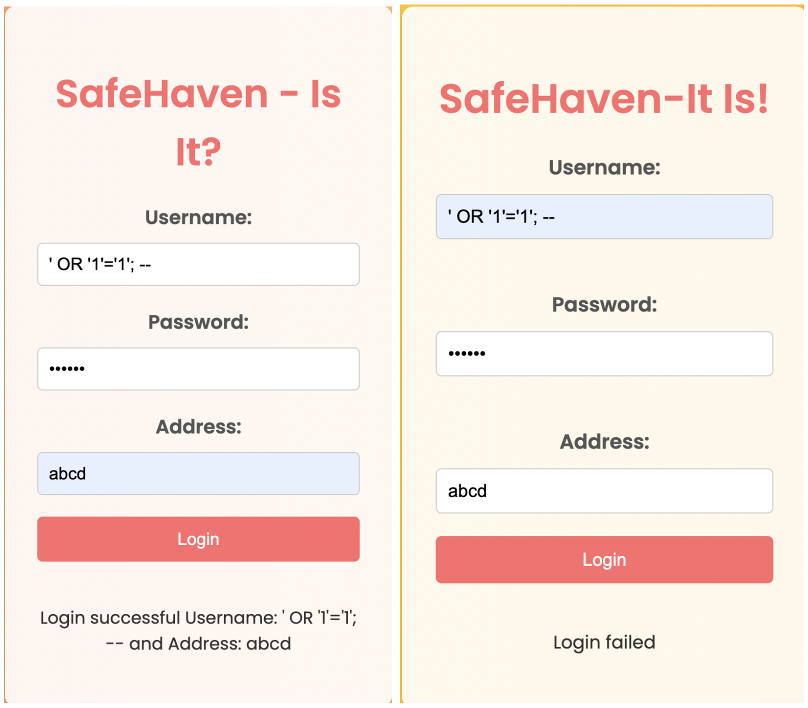
```
const xss = require('xss');
// Extract the username, password, and address
const username = params.get('username');
const password = params.get('password');
const address = params.get('address');

// ...

// Prevent XSS by encoding HTML entities before sending to the client
const safeUsername = xss(username);
const safeAddress = xss(address);
```

The “xss” function takes the user input and transforms it, replacing or encoding characters that could be interpreted as HTML or JavaScript. This ensures that even if a user enters malicious script code, it will be treated as plain text when displayed in the client.

Recording Outcomes

Testing SQL injection:

The password entered is incorrect, and the username is an SQL command. This should not be a valid command and should not be executed at all. On the left without the middleware, this fails and the attacker has gained access. Here on the right, we can see that the use of the middleware and the parameterized queries has prevented an SQL injection.



SafeHaven - Is It?

Username: ' OR '1'='1'; --

Password: ••••••

Address: abcd

Login

Login successful Username: ' OR '1'='1'; -- and Address: abcd

SafeHaven-It Is!

Username: ' OR '1'='1'; --

Password: ••••••

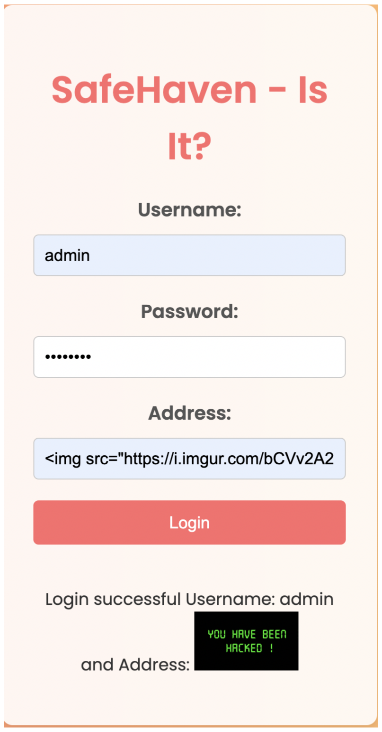
Address: abcd

Login

Login failed

Testing XSS attack:

The address field has been inputted with an HTML tag which is meant to make malicious unauthorized changes to the web page UI. On the left, we can see that the attacker has succeeded in displaying the malicious img tag and conducted an XSS attack. On the right, we notice that the use of the “xss” library is used to encode HTML entities and stop XSS attacks.



SafeHaven - Is It?

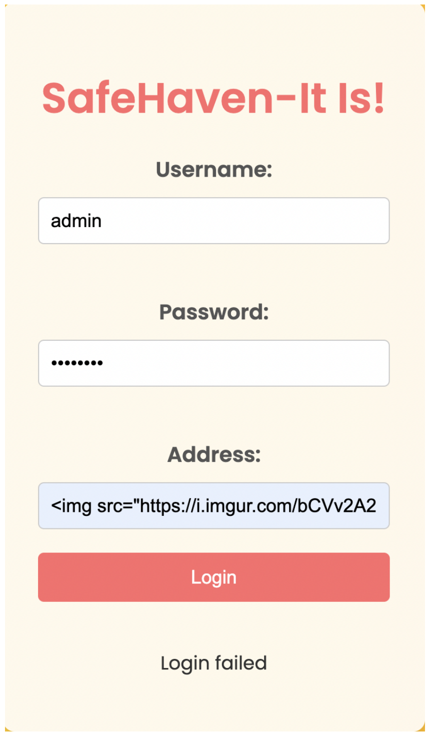
Username: admin

Password: ••••••••

Address: <img src="https://i.imgur.com/bCVv2A2

Login

Login successful Username: admin and Address:



SafeHaven-It Is!

Username: admin

Password: ••••••••

Address: <img src="https://i.imgur.com/bCVv2A2

Login

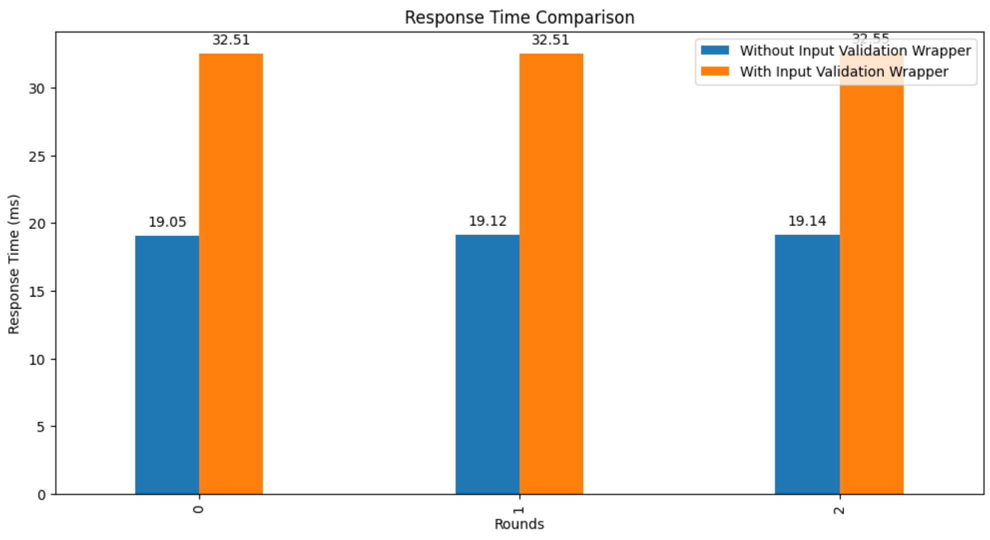
Login failed

Analyzing performance tradeoffs

By logging the response times of the web page for both versions, we can estimate the performance tradeoff with and without the input validation wrapper.

Test Steps:

- Ran tests three times on the old version (without input validation wrapper).
- Took the average response time for these three runs.
- Repeated the process for the new version (with the input validation wrapper).



Average Response Times:

Without wrapper	With wrapper
19.105ms (avg of 3 tries)	32.519ms (avg of 3 tries)

<u>Potential future steps</u>

1. As user traffic increases, the impact of the input validation wrapper on performance becomes more pronounced. Scalability considerations are essential for applications expecting high user loads. We aim to use Apache JMeter (JMeter enables you to assess how well your application scales by increasing the number of virtual users).

2. In resource-constrained environments, such as limited server capabilities, and in scenarios where the login web form is dealing with large data, the additional processing introduced by the validation wrapper may strain available resources. We aim to use Chrome dev tools and Apache Jmeter.

3. By analyzing performance, developers can optimize the middleware logic, ensuring that validation and sanitization processes are efficient and streamlined.

BIBLIOGRAPHY

[1] Cross site scripting (XSS). Cross Site Scripting (XSS) | OWASP Foundation. (n.d.). https://owasp.org/www-community/attacks/xss/#:~:text=Cross%2DSite%20Scripting%20(XSS)%20attacks%20occur%20when%3A,being%20validated%20for%20malicious%20content.

[2] Robert, Shreedhar, 3xhale, &amp; Vivek. (2023, August 14). Cross site scripting (XSS) attack tutorials with examples, types &amp; prevention. Software Testing Help. https://www.softwaretestinghelp.com/cross-site-scripting-xss-attack-test/

[3] Banach, Z. (2022, August 30). Input validation errors: The root of all evil in web application security. Invicti. https://www.invicti.com/blog/web-security/input-validation-errors-root-of-all-evil/

[4] GeeksforGeeks. (2023, February 23). Authentication bypass using SQL injection on login page. GeeksforGeeks. https://www.geeksforgeeks.org/authentication-bypass-using-sql-injection-on-login-page/

[5] A. Alzahrani, A. Alqazzaz, Y. Zhu, H. Fu and N. Almashfi, "Web Application Security Tools Analysis," 2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids), Beijing, China, 2017, pp. 237-242, doi: 10.1109/BigDataSecurity.2017.47.

[6] Google. (n.d.). The basics of web hacking. Google Books. https://books.google.com/books?hl=en&amp;lr=&amp;id=qFIJaYv5bI4C&amp;oi=fnd&amp;pg=PP1&amp;dq=web%2Bsecurity%2Btechniques%2Band%2Btools&amp;ots=7YjziMm3hT&amp;sig=2jpTosNhWoIggZKlSXG9E6rJZ18#v=onepage&amp;q=web%20security%20techniques%20and%20tools&amp;f=false