Ethical Hacking for Cyber Security (INTE2102)

Technical Report 1

Sai Teja Bathula (S4038096)

A. Introduction:

Through the set of challenges in this report, I define and discuss the risks that come with Cross Site Scripting (XSS) and SQL Injection (SQLi). The goal was essentially to find vulnerabilities in web applications and learn how various inputs and server-side filters work. To do so, I employed tools like; Burp Suite used in intercepting and manipulative traffic flow, Kali Linux used as operating system environment for penetration testing, SQL Map in Linux for automating the attack of SQL injection and Chromium for manual web traffic testing. These tools were selected based on their performances in revealing any security issues and so enabled the evaluation of every challenge.

B. Tools Used

Towards this assignment, many security testing tools were employed to accomplish the XSS and SQLi vulnerabilities testing comprehensively. They all served different purpose helping in the identification of weaknesses of the web application.

- Burp Suite: Originally designed as web traffic analysis and manipulation tool, Burp Suite allowed for manual and automated testing of XSS and SQLi attacks. It was very useful for checking reflected inputs, changing request body and watch the responses from the server.
- Kali Linux: Similar to Kali Linux, Metasploit served as a penetration testing platform where, along with running tools like Burp Suite, vulnerability exploits were performed with SQLMap.
- SQLMap: This tool was used to minimize the process of conducting an SQL injection by providing means to find the accessible queries. It was especially useful to detect the vulnerability in databases and to extract flags which must be connected with SQLi CTF.
- Chromium: A browser of choice while conducting a manual web inputs testing. It was used in the addition, through post request, of XSS payloads and the analysis of the patch's reaction to the injections.

C. Tools Comparison and Contrast

During testing, several tools were employed to identify XSS and possibly SQLi vulnerabilities in the web applications under test. For example, each of them would have a specific purpose that could make it possible for both usable handheld and automated testing.

- Burp Suite: In its role as a web proxy, Burp Suite enabled the manual interception, alteration and retransmission of requests, which was necessary for observing how the

web application processes inputs in real time. Sit allowed for manual injection of payloads that are not possible using automated tools and detailed analysis of the server responses especially in XSS testing.

- SQLMap: SQL injection was found out by using SQLMap to automate the presumptive stage. Even though SQLMap can handle a lot of complicated SQLi testing scenarios it was not able to find any Positives in this case. Finally, after performing several tests, it has been identified that the website was not being exposed to SQL injection. SQLMap failed to discover any inputs that could be exploited or database flags which could yield information about possible SQL injection vulnerability such as parameterized query or input validation.
- Chromium: When performing XSS tests manually, Chromium allowed to compare the outcome of a web application's behaviour with a real browser. It enabled the examination of reflected inputs in terms of how the page reacted towards different payloads including the successful <script>alert(1)</script> payload.

Thus, these tools, when used in unison, comprised the necessary infrastructural foundation for the detection of security weaknesses.

D. Cross Site Scripting (XSS) Challenges:

1. Challenge 1: My Awesome Website:

- While performing the first try to detect if the web page was prone to Cross-Site Scripting (XSS), I used the technique with a simple script tag containing a payload which makes an alert to see how the page reacted. After that for simple payloads I have referred the basic pay load as **<script>alert(1)</script>**
- Response: The server replied with an HTTP 200 OK status, and the script was present within the HTML response neither encoded nor sanitized. The payload worked and displayed a JavaScript alert box with the message 1, further proving that this page is prone to Cross Site Scripting. As a result of executing this payload, the XSS vulnerability was verified, and I obtained the flag using validation page:
  flag{a99b3071–5ef5–4f1c–bed5–c576360335aa}
- **Cookie Flag:**
  Once I ensured that the page was vulnerable to simple XSS, I used an advanced XSS payload to dump all the cookies that are present in the browser validating the impacts of an XSS attack. The payload used was: **<script>alert(console.log(document.cookie))</script>**
  As a result of executing this payload, the XSS vulnerability was verified, and I obtained the flag using validation page:
  flag{23463f7e–710a–4138–bee6–335c49947293}
- Conclusion for XSS Challenge 1:
  When the <script>alert(1)</script> payload is executed, it demonstrates a reflected cross site scripting flaw. There is no input sanitization or encoding implemented on the server side which permits the attacker to inject and execute JavaScript code on the user's browser. This could lead to exploits for perform actions such as do steal session cookies or modifying the page content for evil purposes.

2. **Challenge 2: Login:**
    - The second XSS challenge was delivered on a login page which involved entering a username and password. I initially tested the username field for XSS vulnerability by injecting a simple script payload**: "<script>alert(1)</script>".**
    - Response: The server responded with a 302 redirect with an 'Invalid login' message in the error parameter meaning that the payload was reflected but not executed here. Further, much to my surprise, I found that the error parameter was appended in the URL query string which was prone to XSS. To test this hypothesis, I used the same payload by passing it in the error parameter of the URL. This time the payload was executed as it popped a JavaScript alert box with the message 1, meaning that the error parameter had XSS vulnerability on the given page. Executing the payload in the error parameter yielded me the following flag: "flag{532495a5–f773–4c60–953a–d10abd176bd1}".
    - Cookie Payload: Exploiting More I decided to try for more attacks, and I wanted to see if the application was storing cookies so I could retrieve them. I crafted a payload to retrieve the cookies stored in the browser using the following script:
      "<script>alert(console.log(document.cookie))</script>".
      As a result of this payload execution, I retrieved the following cookie flag: "Flag {23463f7e-710a-4138-bee6-335c49947293}".
    - Conclusion for XSS Challenge 2:
      This challenge revealed that the login page was vulnerable to XSS, particularly in the error parameter of the URL. I was able to execute both basic and advanced XSS payloads, resulting in the execution of JavaScript and the retrieval of cookies.
3. **Challenge 3: MATH Magic**
    - In this challenge, the tab labelled "Math Magic" let people type in equations. During XSS attacks, I tried the most basic XSS payload on the name field: <script>alert(1)</script>. This was displayed as an invalid equation, although the script was not executed correctly because of issue related to filtering or context problems while the input was reflected on the page.
    - Next, I sent the payload alert(1) with no <script> tags attached to it in any way. While it is reflected, it still was not written as valid JavaScript meaning that only parts of the input were processed.
    - Due to the above filters, I decided to encode the alert(1) payload using JSFuck encoding technique. As for the further work of this obfuscated form of JavaScript, it executed successfully and revealed the presence of XSS vulnerability, providing the flag flag{067c0555-deda-44cb-93b8-48d84909ce96}.
    - Subsequently, based on the same approach, I ran the alert(document.cookie) codded in JSFuck to see the cookies. It also worked fine to show the cookies & giving the cookie flag, flag{fcb277ea-4e30-433b-b081-2507a1b95f9f}.

- Conclusion:
  Math Magic has been susceptible to XSS; however, general filters, which were used initially, did not allow executing JavaScript. Thanks to JSFuck encoding, I were able to perform both alert and cookie flag here, meaning that the application suffers from improper input validation.

4. **Challenge 4: Contact US:**
   - For the fourth XSS challenge, I forwarded the “Contact Us” page and entered <script>alert(100)</script> into the name, email, subject, and message body fields. Although, all the inputs were introduced in the page, none of scripts were working as all the scripts appeared in the safe mode.
   - For my second attempt, I used a more complex payload with nested <script> tags in the subject field: It then returned an alert box of 100 <script>ript>. This worked and escaped the filter, which fired an alert with the message “100” and proving the existence of the XSS vulnerability. Consequently, I got the flag flag{faab1c7a-04be-4328-83e3-839886fd8bd5}.
   - Next, I used a similar nested script payload to retrieve the cookies with the following payload: Here we have <sc<script>ript>alert(document.cookie)</sc</script>ript>. This was done effectively, and the cookies were shown and the cookie flag flag{c4a34181-74f2-4554-a061-48699832dd3c} was obtained.
   - Conclusion:
     There was an XSS in the Contact Us page from the side of the subject field. While basic payloads were cleanly sanitized, nested scripts slip through the filter and both the alert and cookie flags were fetched indicating the dangers of lack of input validation.
5. **Challenge 7: Button Generator:**
   - I examined the sanitizer by exploiting the Button Generator page using the following SVG-based injection attack on the text parameter "><svg><animate onbegin=alert(1) attributeName=x></svg>.
   - I received an alert flag: “flag{a52051ac-e24b-47ee-8222-25f4bd48c726}”
   - I then used another payload comprised of an SVG where the payload can fetch cookies "><svg><animate onbegin=alert(console.log(document.cookie)) attributeName=x></svg>. This payload proved effective; it returned the cookies: “flag{d2caf966-60b3-4e81-8a94-1fcb4517af99}”.
   - Conclusion:
     The Button Generator page was affected by an XSS through the text parameter and the execution of scripts through SVG animations.
6. **Challenge 8: My New Website:**
   - When testing for XSS in the page for the first time, I used the simplest script payload, <script>alert(‘XSS’)</script> hoping it would generate an alert. However, the script was reflected as plain text, which appeared to suggest that the server was cleansing the input by either stripping out or escaping characters such as < and >. That sanitization stopped the script

from running and at the same time, made it powerless. For my second try, I just decided to use a more complex payload, <noscript><script>alert(20)</script></noscript>, to try and remove any actively closing <noscript> tag that turns off JavaScript and then includes a new <script> tag. The concept was that the </noscript> would bring an end to any <noscript> block so that the script could be run. However, in print, the input was exhibited in plain text, the characters being turned to noscriptscriptalert20script, meaning that the server cleared every input vigorously before passing them to the browser.

- While having the ability to execute code on the client side was a viable attempt for cases where JavaScript could be disabled, the server side sanitization process eliminated this possibility also with the help of </nosscript>. This means that no matter what the user inputs or introduces on the page, they cannot be directed to be executed by the browser as a JavaScript code since the server side of the system sanitizes the input. In this case the attempt to bypass the JavaScript disabling utilizing </noscript> failed because the server side filtering was very effective.

E. SQL Challenges:

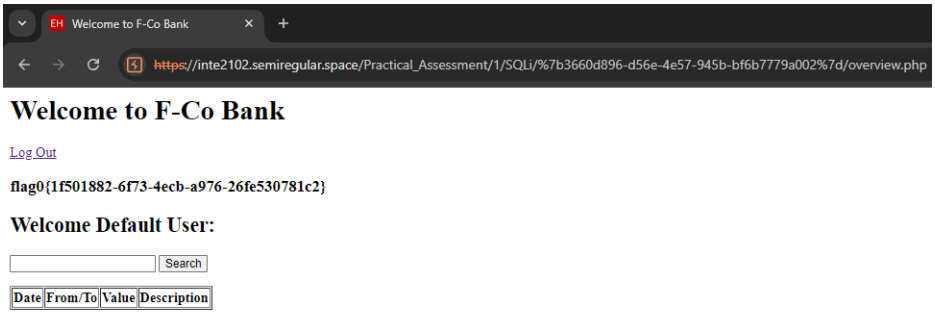
1. flag0{1f501882-6f73-4ecb-a976-26fe530781c2}:

During the SQL Injection challenge of the Internet Banking website, I first tried the SQL injection attack on the login form with a simple SQL Injection Payload. The first one was to try to input: admin' -- into the User ID field, which includes a SQL. However, to do so I got the message, "Nice try but not good enough!", meaning maybe the input was sanitized or the payload was not compelling enough to allow one log in.

In the second attempt, I used the User ID field with a payload of 1' OR '1' = '1 which is a typical SQL injection technique used for login system cracking with success. This payload attempts to test whether 1=1 is true to allow potential access without having to type in the correct username and password. However, I got an error message which stated that a password is required, this gave a hint that the input was partly processed but did not take over the complete validation full file.

For my third and successful attempt, I applied the same payload to both the User ID and Password fields: 1' OR '1' = '1. Another technique used which made both fields entered the condition statement, therefore passing the authentication system as true. As a result, I successfully logged into the system and retrieved the first flag.

This successful SQL injection demonstrates a major weakness by the login form of not sanitizing inputs received, as well as, not implementing parameterized queries. Direct inputting of SQL statements into the fields allowed me to get past the authentication part of the application and access certain secure areas.



2. **flag1{e2f52896-070d-4c68-aa09-f6ba88383c16}:**

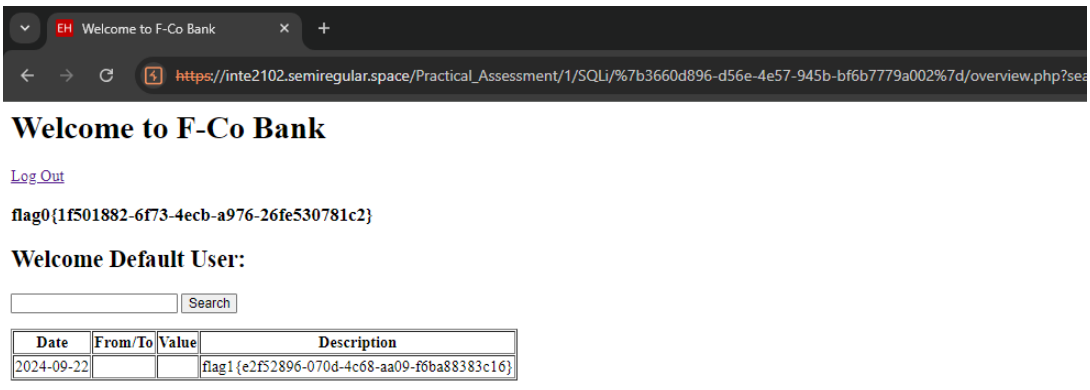
   For the second example, I wanted to investigate some possible SQL Injection scenarios, so I entered the vulnerabilities search bar input. First, I was testing for SQL injections and for this, I used payload ‘ OR ‘1’=’1 which refreshed the page, but I never got an error or useful output from it. I then tried to run additional SELECT statements in order to parse the rest of the schema data from the database. Employing this problem, I utilized the payload example as an aid in my representation.

   But when using this query: select null, null, null, table_name from information_schema.tables -- instead, I get through with returned results without encountering SQL syntax errors. This meant that the query structure did not follow the Formatting expected by the application.

   To avoid this, I changed the number of columns in the SELECT statement, but I was still struggling with other column count problems. Finally, I made some modifications on the payload to correctly match all the query structure. The final payload that successfully extracted the database name was:

   'UNION SELECT NULL, NULL, NULL, NULL, database() FROM information_schema.schemata –

   This payload allowed me to retrieve the name of the database and obtain **Flag 1**:



3. **flag2{5f91f157-8f8e-46f9-9590-d56527a83923}:**

   I started this SQL Injection challenge by attempting to extract table names from the database using the search option. Making use of the payload

'UNION SELECT NULL, NULL, NULL, NULL, table_name FROM information_schema.tables --

2024-09-22			INNODB_BUFFER_PAGE
2024-09-22			INNODB_SYS_VIRTUAL
2024-09-22			user_variables
2024-09-22			INNODB_TABLESPACES_ENCRYPTION
2024-09-22			INNODB_LOCK_WAITS
2024-09-22			THREAD_POOL_STATS
2024-09-22			ed_gebruiker
2024-09-22			transactions

Two table names that I was able to obtain were transactions and ed_gebruiker. It appeared that sensitive information about user credentials and transaction details was contained in these tables. Next, I used the payload below to retrieve the column names from these tables:

2024-09-22			POLLS_BY_LISTENER
2024-09-22			POLLS_BY_WORKER
2024-09-22			DEQUEUES_BY_LISTENER
2024-09-22			DEQUEUES_BY_WORKER
2024-09-22			gebruiker_ed
2024-09-22			wagwoord_gr
2024-09-22			phone
2024-09-22			date_created
2024-09-22			txn_date
2024-09-22			txn_from
2024-09-22			txn_to

The following crucial columns were found by the query: phone, wagwoord_gr, gebruiker_ed, txn_date, txn_from, and txn_to. These columns seemed to contain important information that could be used in the future. I used the payload to obtain access to the data in the ed_gebruiker table:

'UNION SELECT NULL, NULL, NULL, NULL, gebruiker_ed FROM ed_gebruiker --

Date	From/To	Value	Description
2024-09-22			34436606
2024-09-22			42353243
2024-09-22			72474194

This made the usernames and related information visible. I then used a similar query on the wagwoord_gr field to collect passwords:

Date	From/To	Value	Description
2024-09-22			anvilreporter
2024-09-22			*ashyhappily
2024-09-22			dryhundredth

Three usernames (anvilreporter, *ashyhappily, and dryhundredth) together with their corresponding passwords were displayed in the results. I finally tried logging in with several permutations using the credentials. The first username and third password were the right combination, validating access to and recovery of Flag 2.