# Group Project 5

(Group 1)

Sushruti Bansod

Joshua Fishman

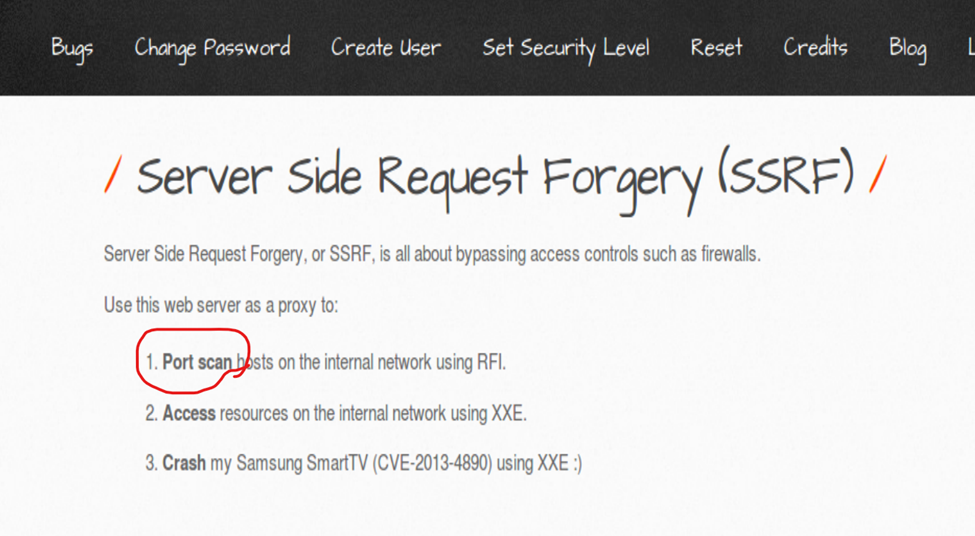
Nick Forleo

Mario Amaro

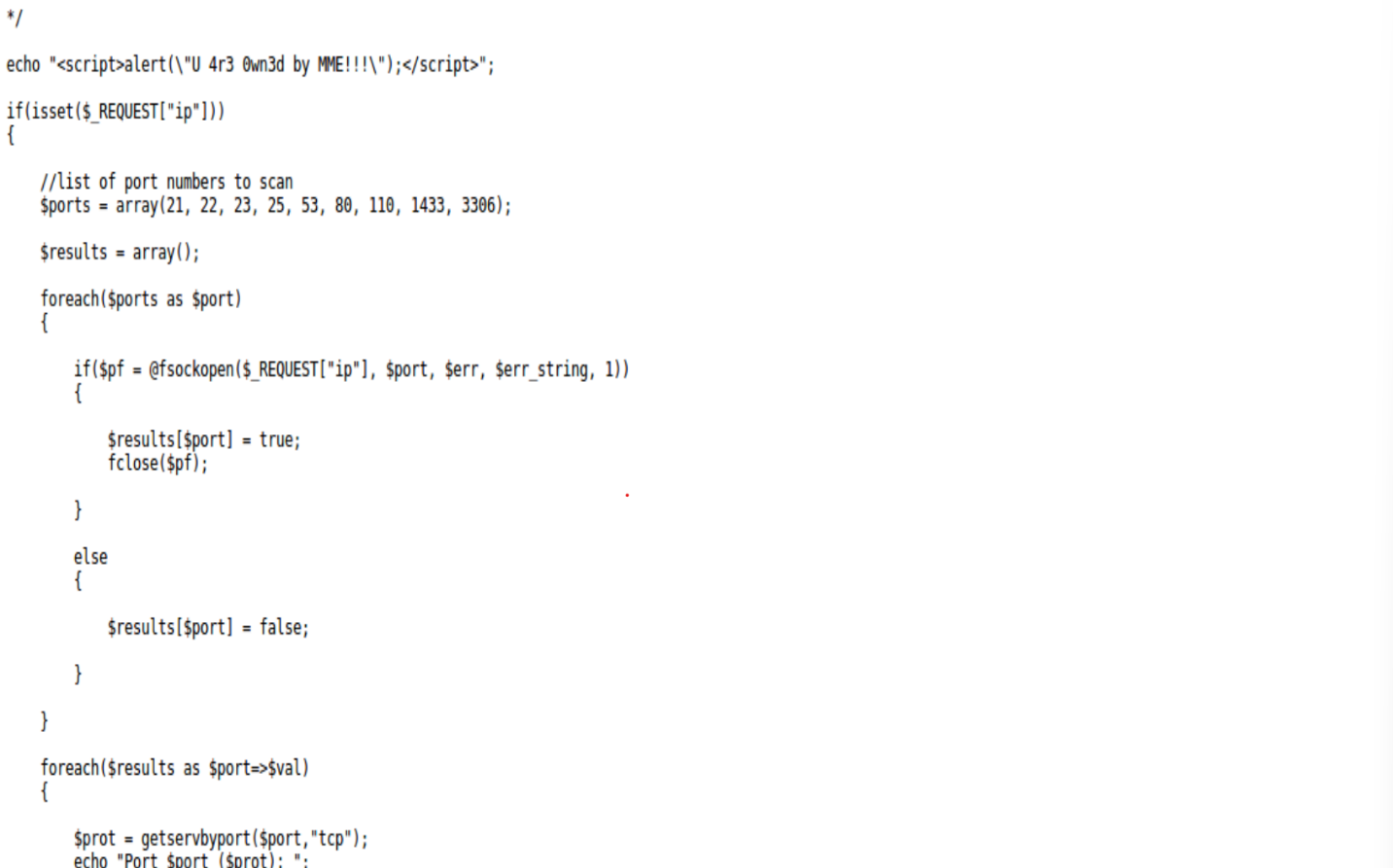
**Report Guidelines (please follow this guideline to create your report)**:

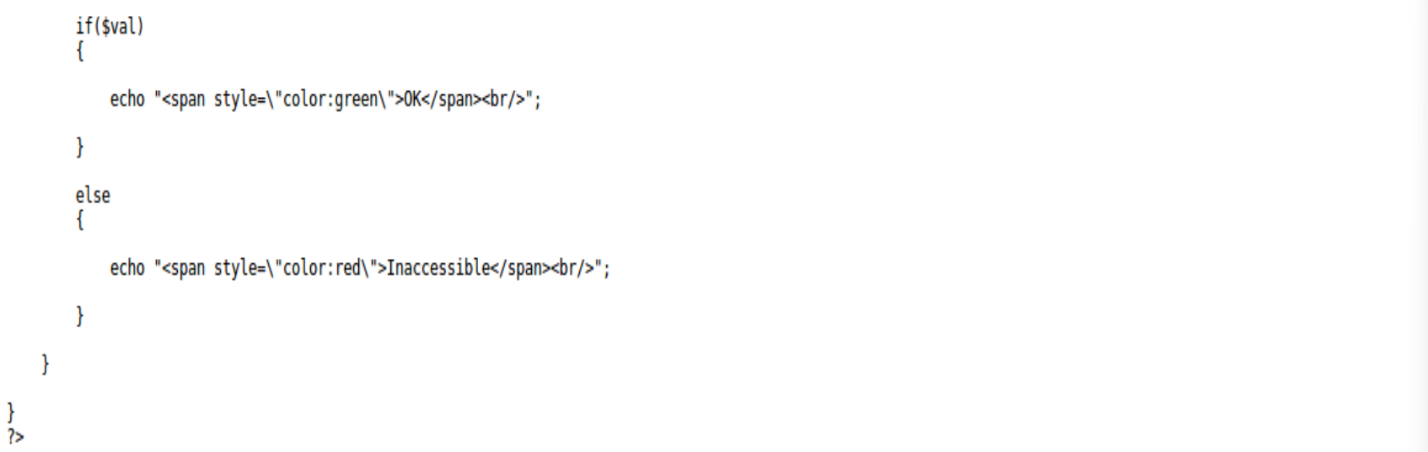
**Task 1:** Show a scenario that attacks a server-side vulnerability (a system/component or application logic flaw)

The target web application was bWAPP using HTML and a script provided by bWAPP. The vulnerability is the remote file inclusion. On figure 1.1 we are going to port scan host on the internal network using the Remote File Inclusion (RFI).

Figure 1.1 SSRF

A script was written(ssrf-1.txt) to trick the server into displaying its port information.



Figure 1.2 Request Forgery Script

We went to the Remote & Local File Inclusion page of bWAPP. The page asked to select a language and we left it as default ‘English’ and hit go.

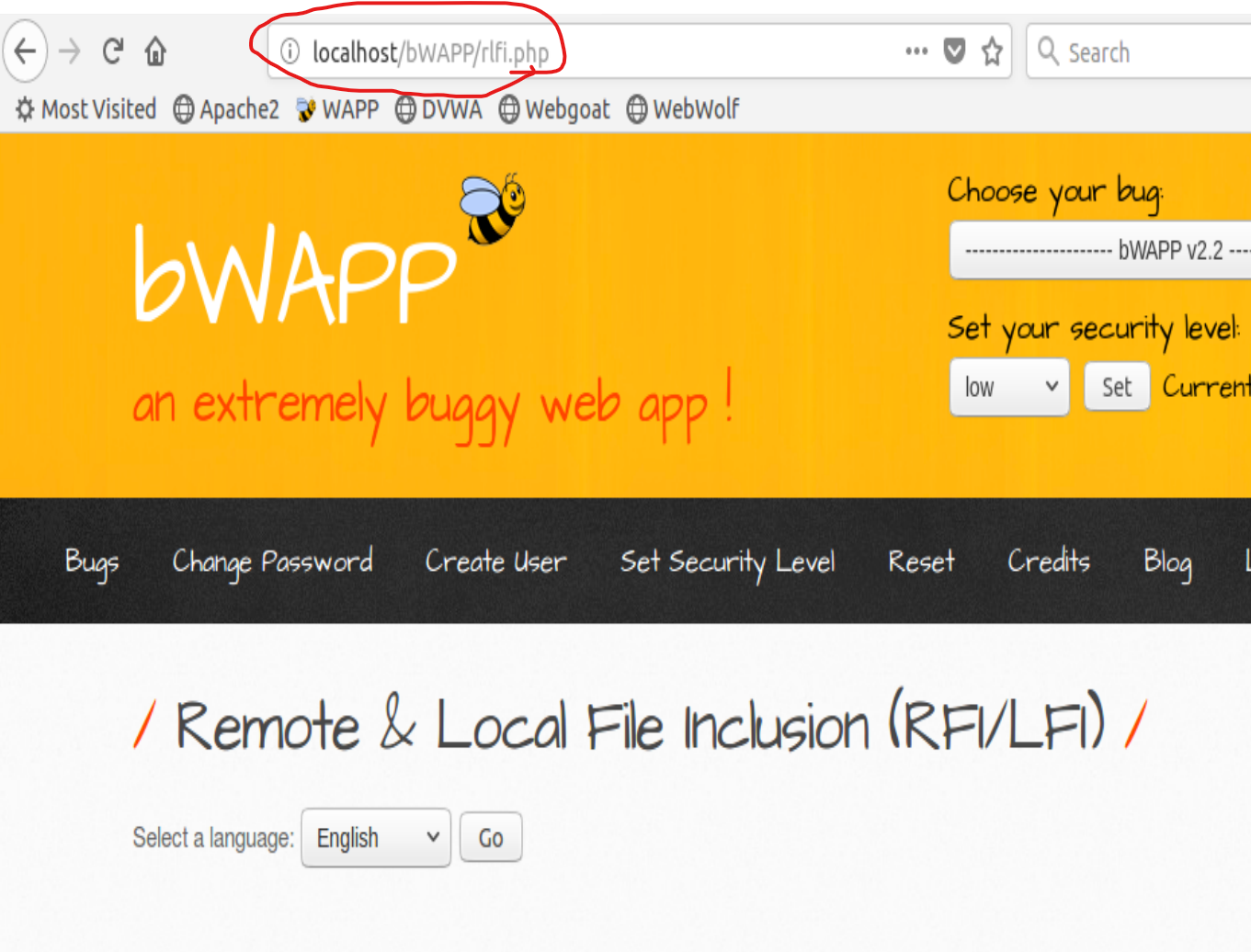


Figure 1.3 RFI/LFI

Observing the URL, we see that a ‘lang\_en.php’ script was submitted to have bWAPP in English.

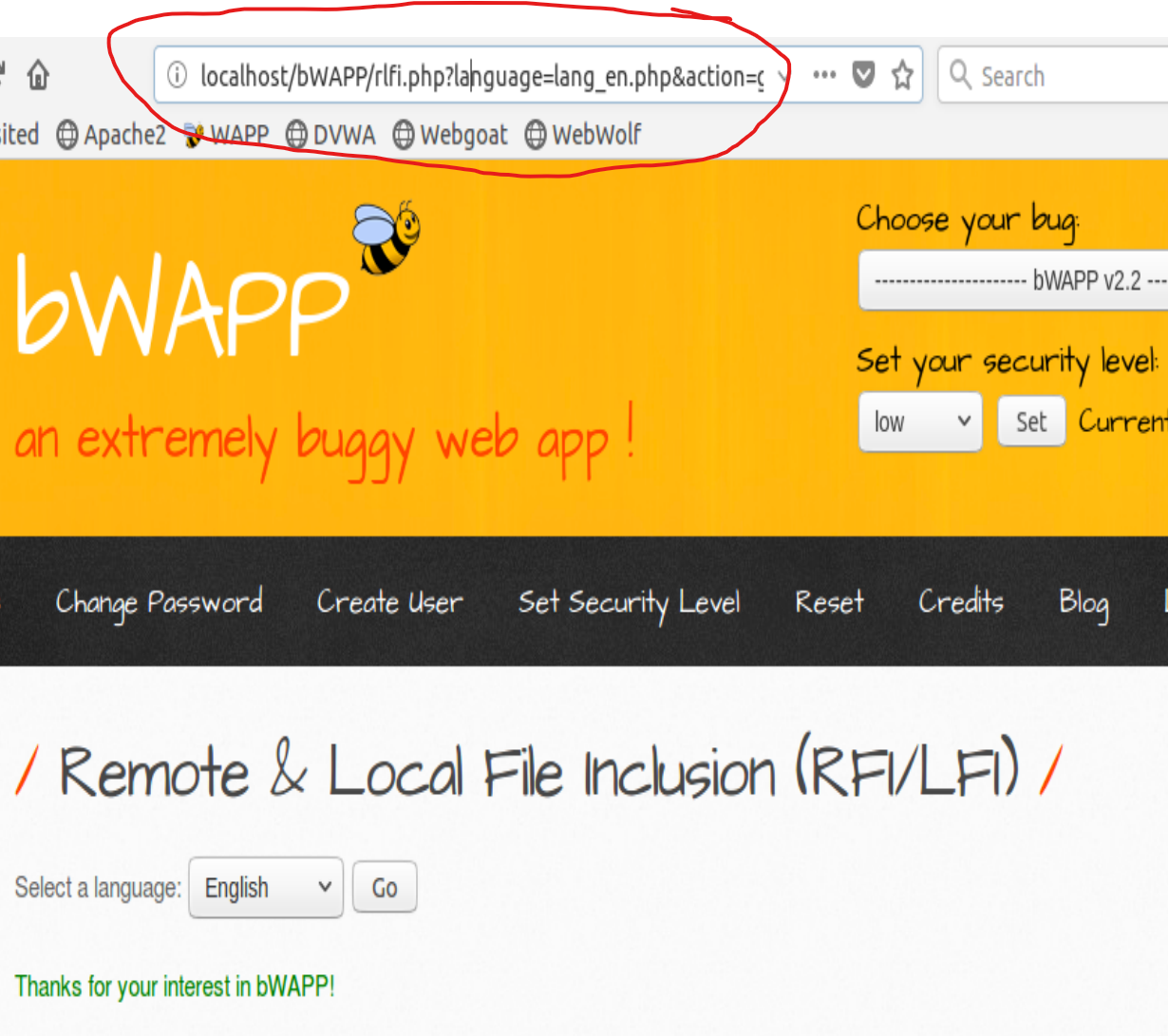


Figure 1.4 Hitting go on language.

We then took note of the location of ‘ssrf-1.txt’

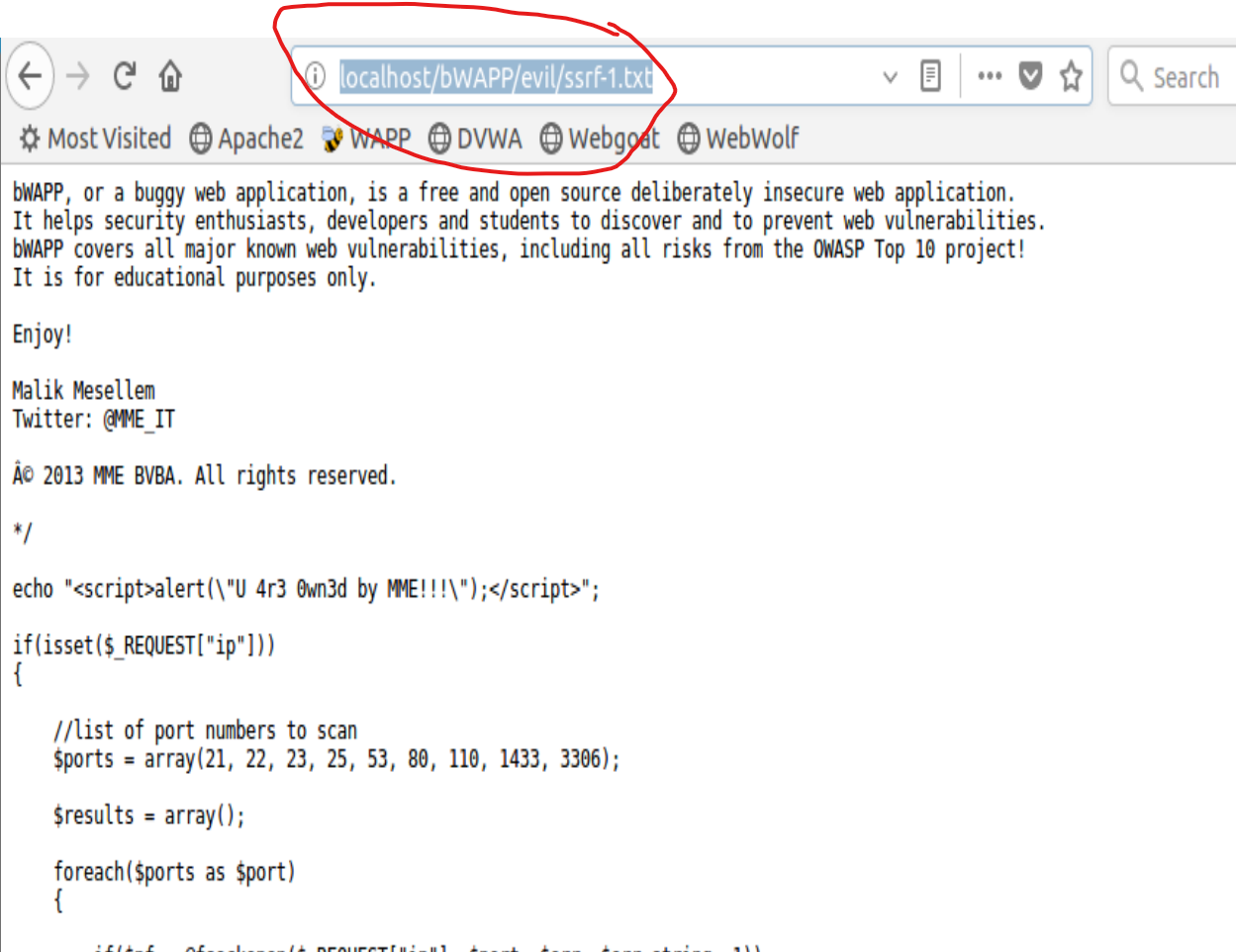
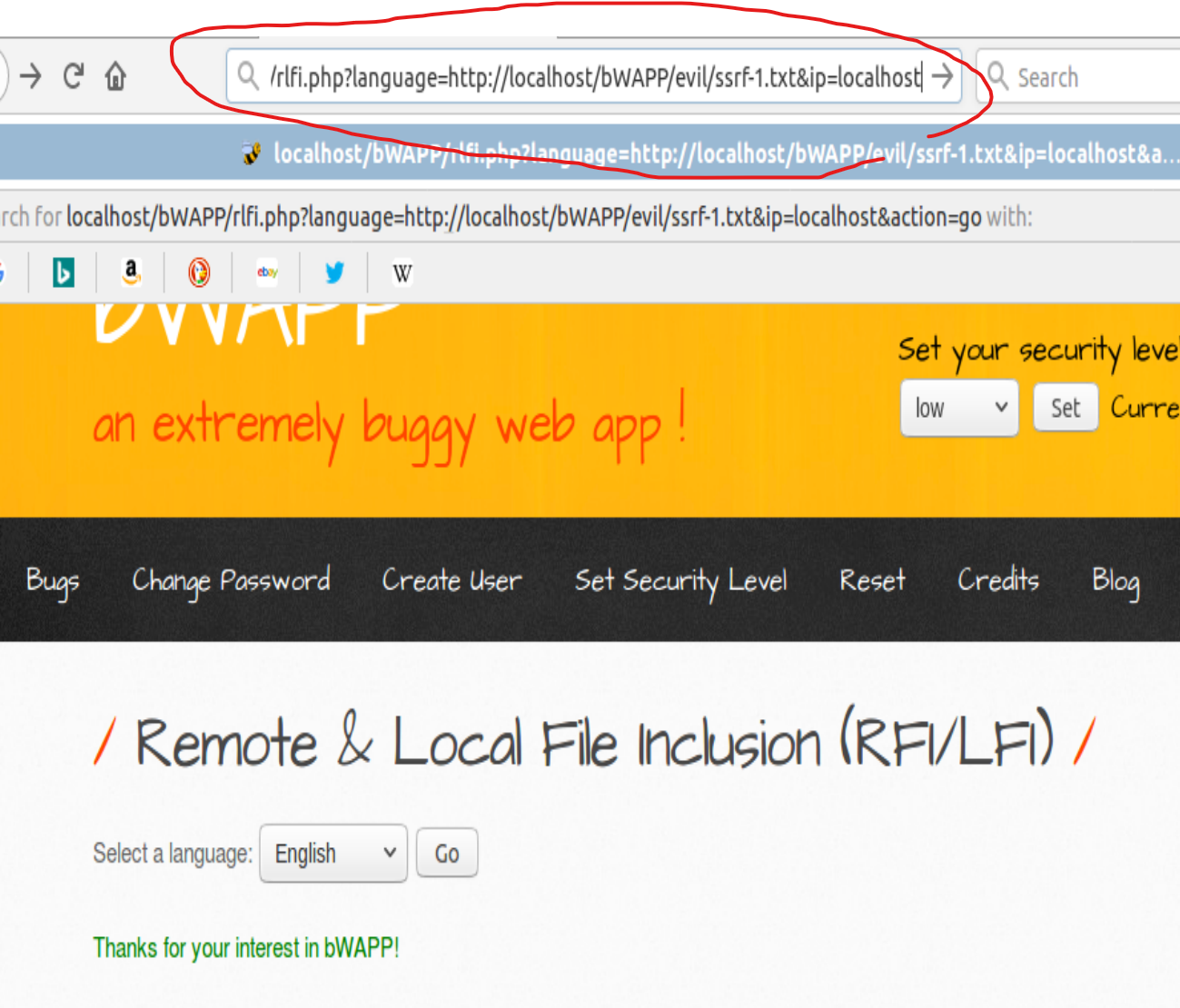


Figure 1.5 SSRF script URL

We replaced ‘lang\_en.php’ with the location of the ‘ssrf-1.txt’ script and the IP address to our local host address.

Figure 1.6 Replace HTML with script

Finally, we refreshed the web page and received our prompt that our script ran successfully. We are now able to view the ports on the server.

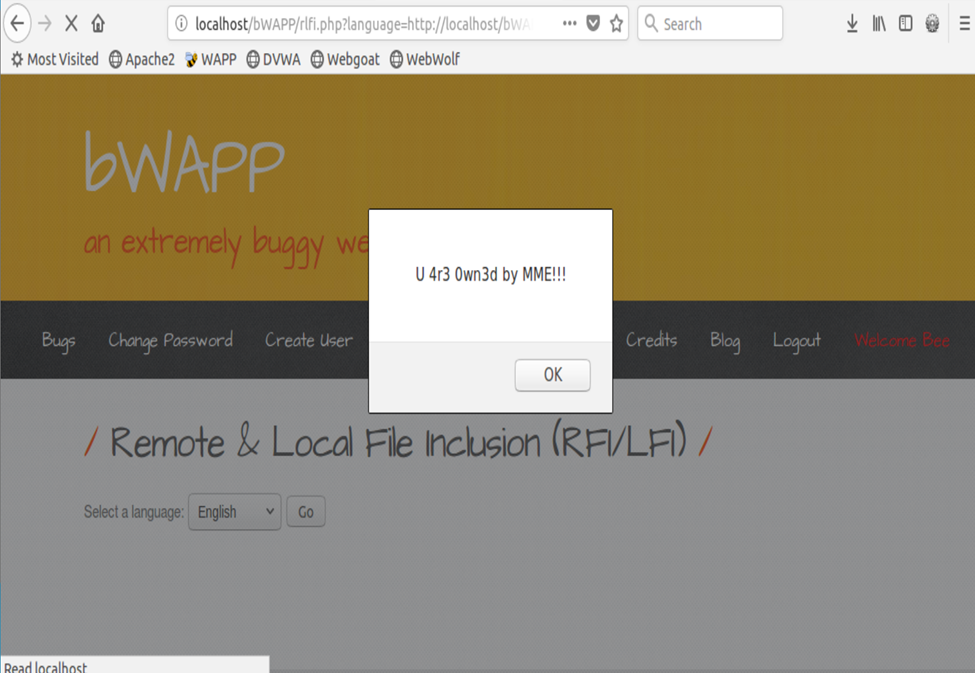
Figure 1.8 Prompt for successful attack

Figure 1.9 Port Scan

**Task 2:** Show a sophisticated attack to compromise the target through exploiting a combination of at least 3 web vulnerabilities discussed throughout this course.

**Task 2.0.1: Attack Setup**

For this sophisticated three-part attack, we will be compromising the DVWA website with security set to low using Zed Attack Proxy (ZAP) to reset the passwords of multiple users and then brute force attack the usernames to be able to log in with multiple accounts.

To perform this three-part attack, we need to be able to simulate legitimate users using the DVWA. The DVWA is initialized with multiple username and password combinations, but since the passwords are masked (possibly hashed, encrypted, or encoded). We will change all the passwords to “password” (the admin account’s default password) so that we can log into them, but we will pretend that these passwords are not known to us and connect be identified. Additionally, to set up the third attack targeting predictable usernames, we will change the usernames of these accounts as if they were generated in a predictable way.

These changes are done by opening a terminal on the VM and opening an interactive mysql shell (when prompted, the password “ubuntumysql” is used):

mysql –u root dvwa -p

Then run the following MySQL commands:

-- change all passwords to "password" so that we can simulate other users logging in and getting attacked

UPDATE users, (SELECT password FROM users WHERE user="admin") as admin

SET users.password = admin.password;

-- change usernames so that we can exploit a predictable usernames vulnerability

UPDATE users SET user = "u0000002" WHERE user\_id = 2;

UPDATE users SET user = "u0000003" WHERE user\_id = 3;

UPDATE users SET user = "u0000004" WHERE user\_id = 4;

UPDATE users SET user = "u0000005" WHERE user\_id = 5;

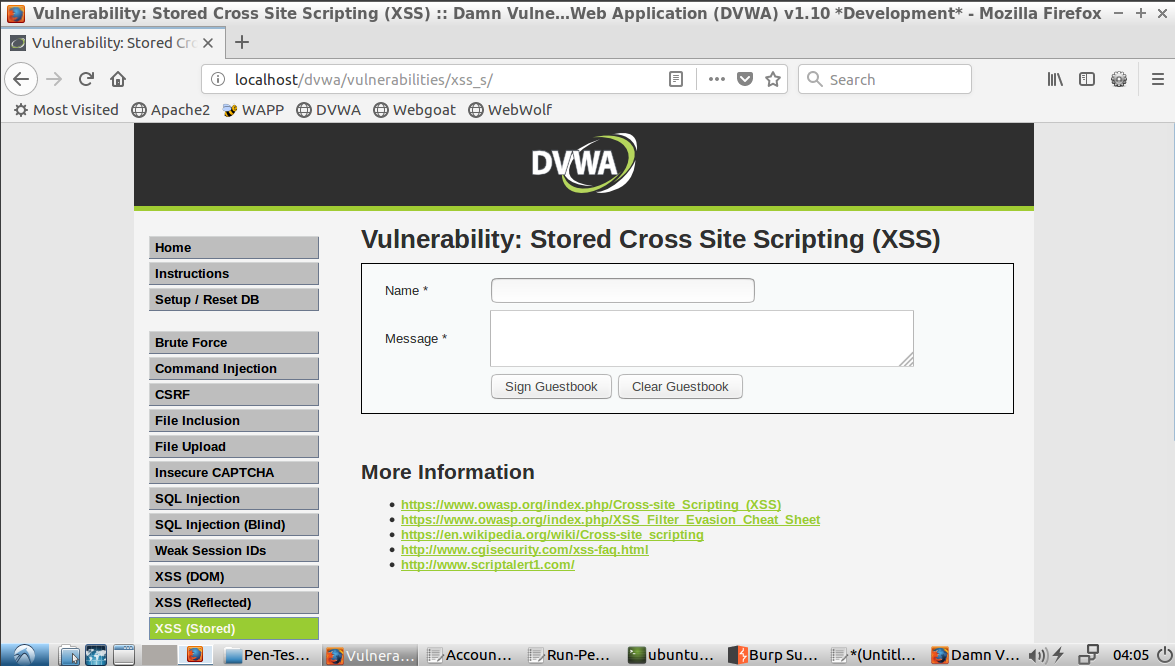
-- show the users table so that we know everything worked

SELECT \* FROM users;

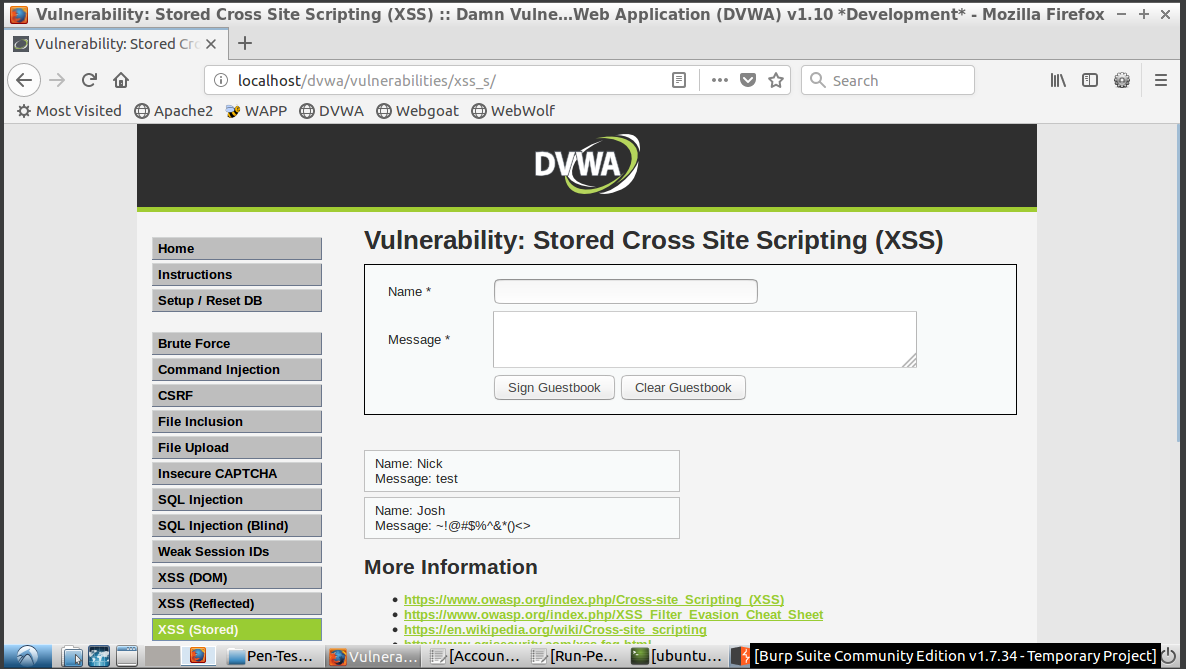
As the attacker, we are granted the username of “u0000005”, so we will use that to log in.

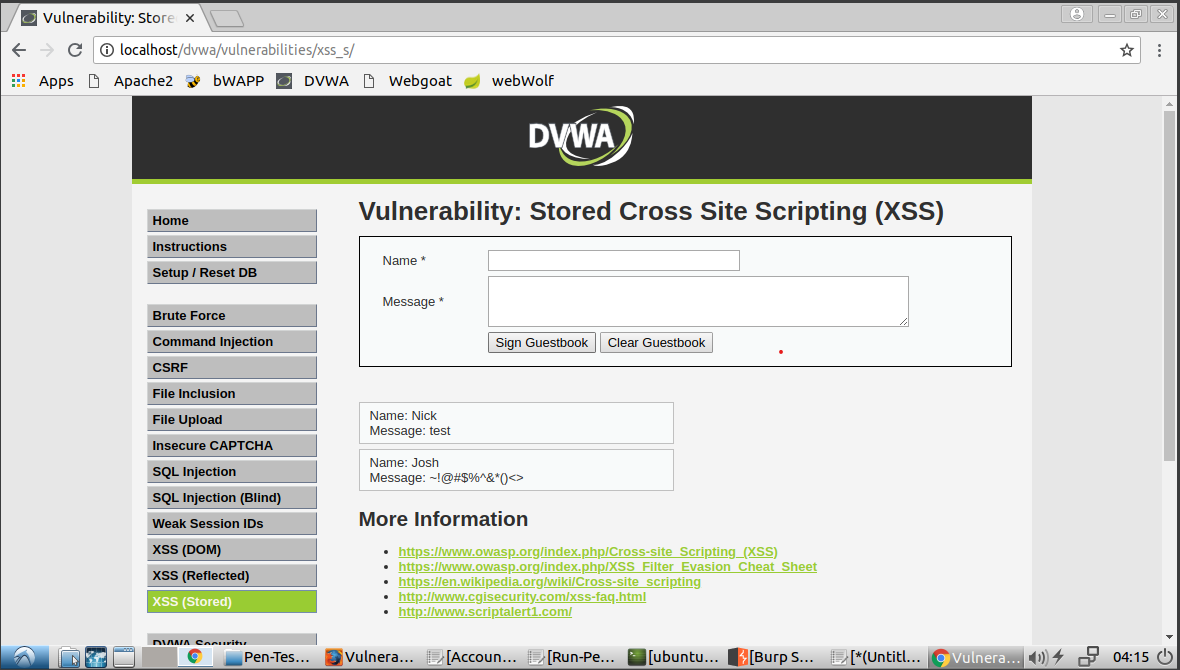
## Task 2.1: Stored XSS attack

This section describes how we discovered this vulnerability, which was done during a previous exercise. To uncover an XSS vulnerability, we looked through DVWA. We were able to uncover this vulnerability without the use of a testing toolkit. We did, however, use the toolkit (in this case, Burp Suite) to verify the requests and the data being transferred. This is where we found a form that accepts user input and prints it back out on the web page. Figure 2.1 shows what that page looks like before adding any data.

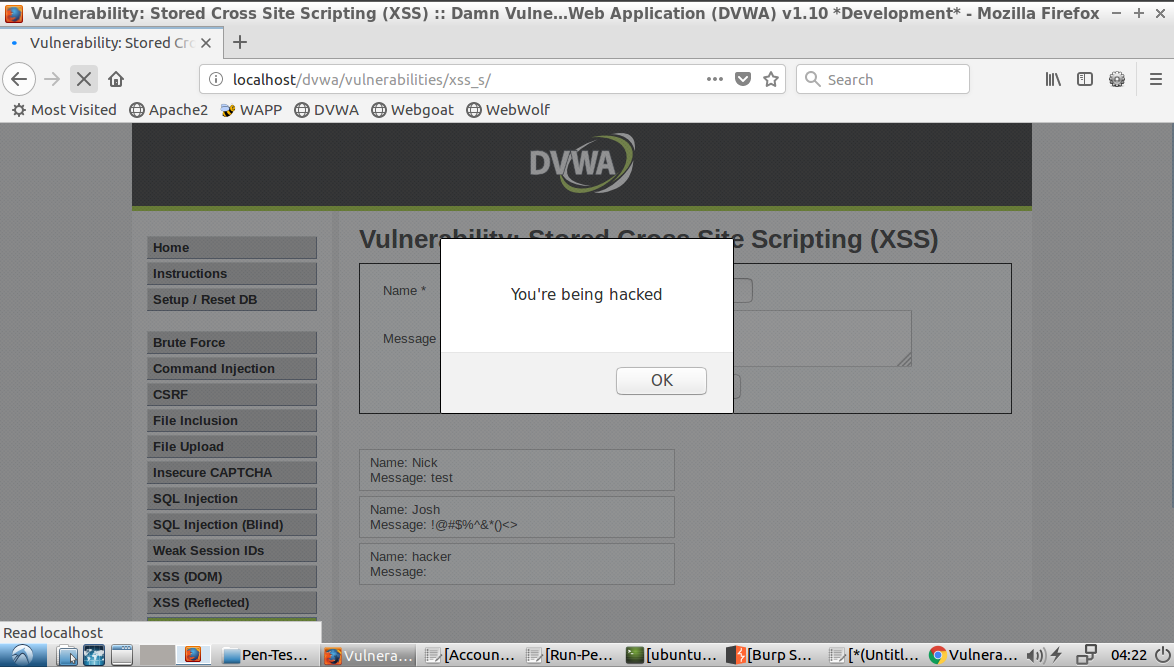
*Figure 2.1 - Form to enter data*

After inputting some dummy data, we discovered two things. The first being that special characters are not stripped out. This means that the user input does not have basic input sanitization. We show this in Figure 2.2. The second discovery was that the data persists through a reload and a new browser. This tells us that the data is being stored in a database and that other users have the same data returned from the database, which is shown in Figure 2.3.

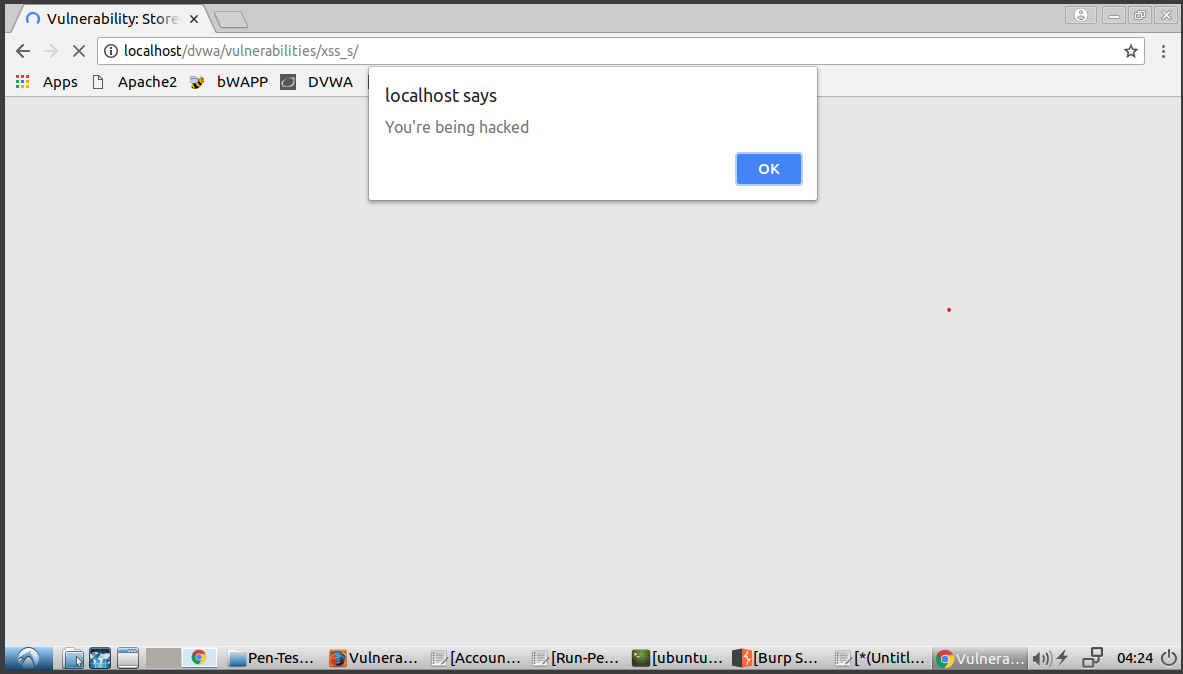
*Figure 2.2 - Special characters are rendered*

*Figure 2.3 - The data input from Mozilla Firefox is shown in a new browser (Chrome) and session*

Now that we have discovered this, we can craft a simple stored XSS attack. This is a stored XSS attack because we can save our malicious input onto the server which can then affect other users. We demonstrated this by inserting `<script>alert(“You’re being hacked”)</script>` in the message field. Once this is saved and the browser retrieves the content from the database, the browser tries to render it, causing our code to be executed. Figure 2.4 shows the results of the script when initially input (in Firefox) while Figure 2.5 shows how the separate browser (Chrome)/session is affected by the attack.



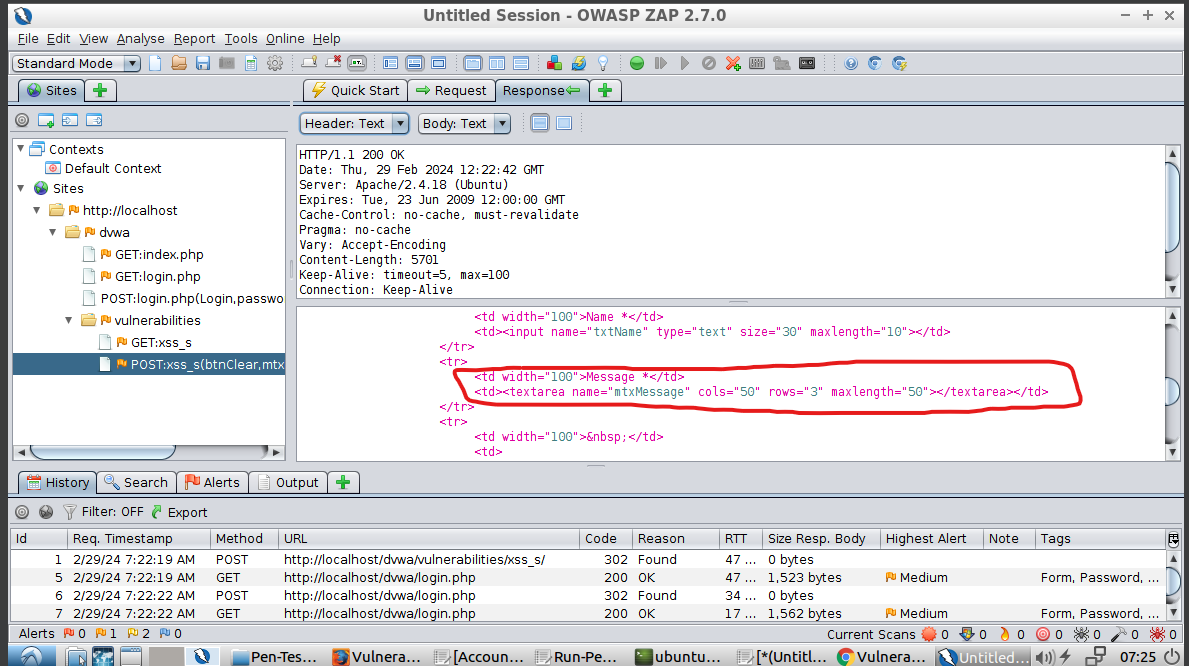
*Figure 2.4 - Original input in Firefox showing results of script saved to database*



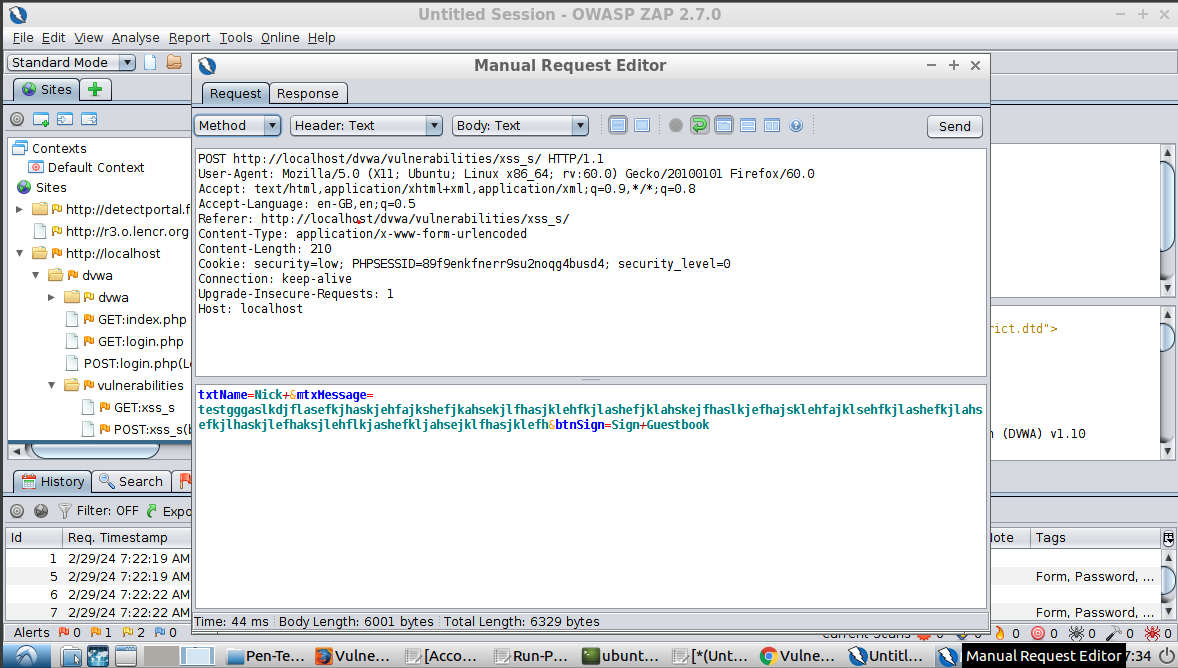
*Figure 2.5 - Second browser (Chrome) on reload showing the script being retrieved from the database*

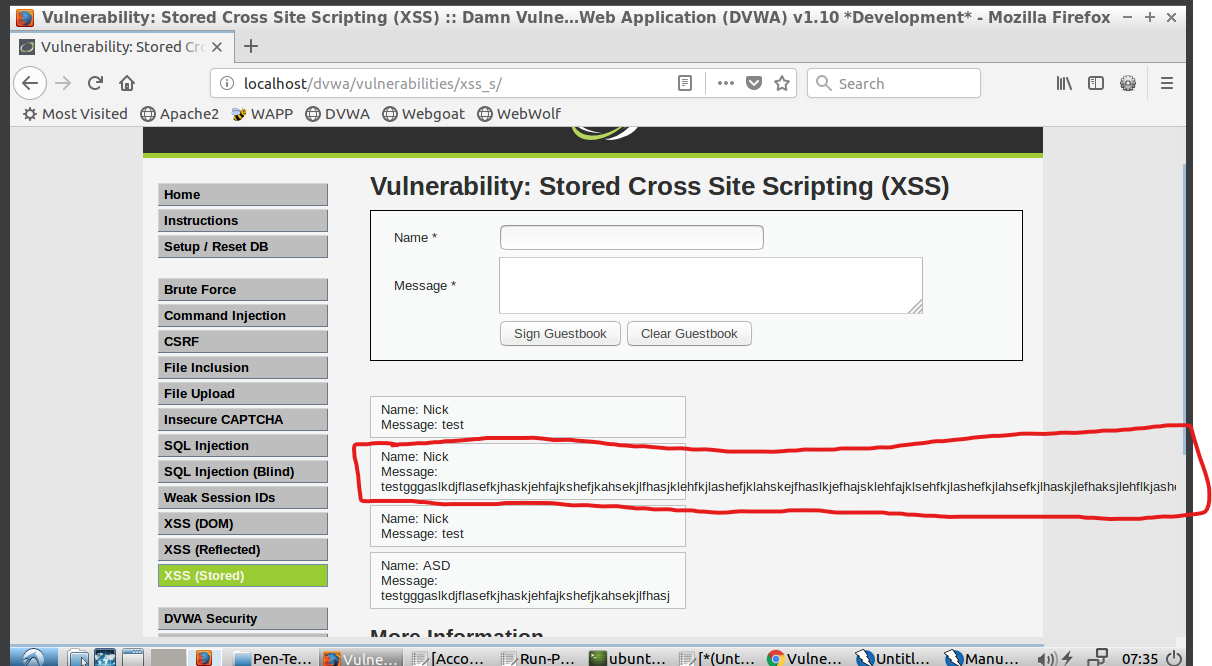
This shows that this attack can range anywhere from a mild inconvenience to a dangerous security breach depending on the complexity and/or intention of the bad actor taking advantage of this vulnerability.

We will take advantage of this exploit to execute forgery requests on other users. More specifically, the stored XSS exploit will deliver the request forgery payload. This will be described in more detail below. One thing that noticed during discovery that would affect our ability to load executable code was the `maxlength` attribute of 50 in the message input text area – which means our code would need be equal or less than that in order to have a successful upload (Figure 2.6).

*Figure 2.6 - maxlength=50*

To bypass this, we used ZAP to execute a modified request without this attribute, thus allowing us to save a larger value. In Figure 2.7, you can see that we sent a request without the maxlength attribute and Figure 2.8 shows that all characters were saved to the database, even though there are more than 50.

*Figure 2.7 - Modified request with more than 50 characters*

*Figure 2.8 - Message larger than 50 characters rendered in GUI.*

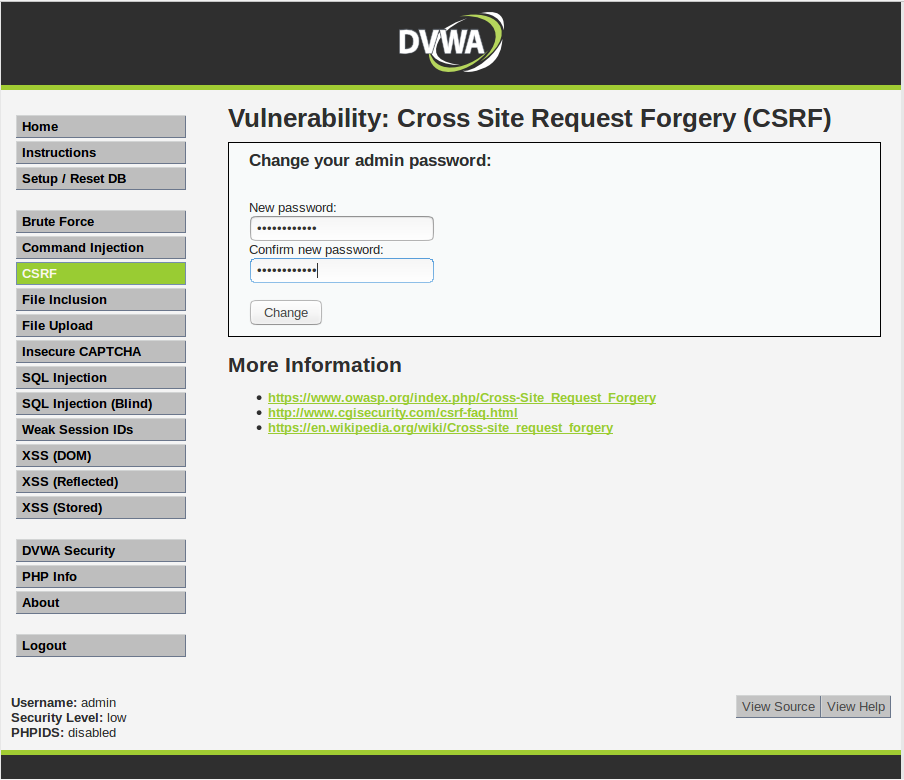
## Task 2.2: Reset a user’s password with request forgery.

Using ZAP and DVWA, we were able to simulate a request forgery attack on a user. This attack could change the user’s password to a value known by the attacker and the attacker would be able to log into the application as the attacked user.

The idea for this attack comes from learning how to execute a Cross-Site Request Forgery which was done in a previous lesson.

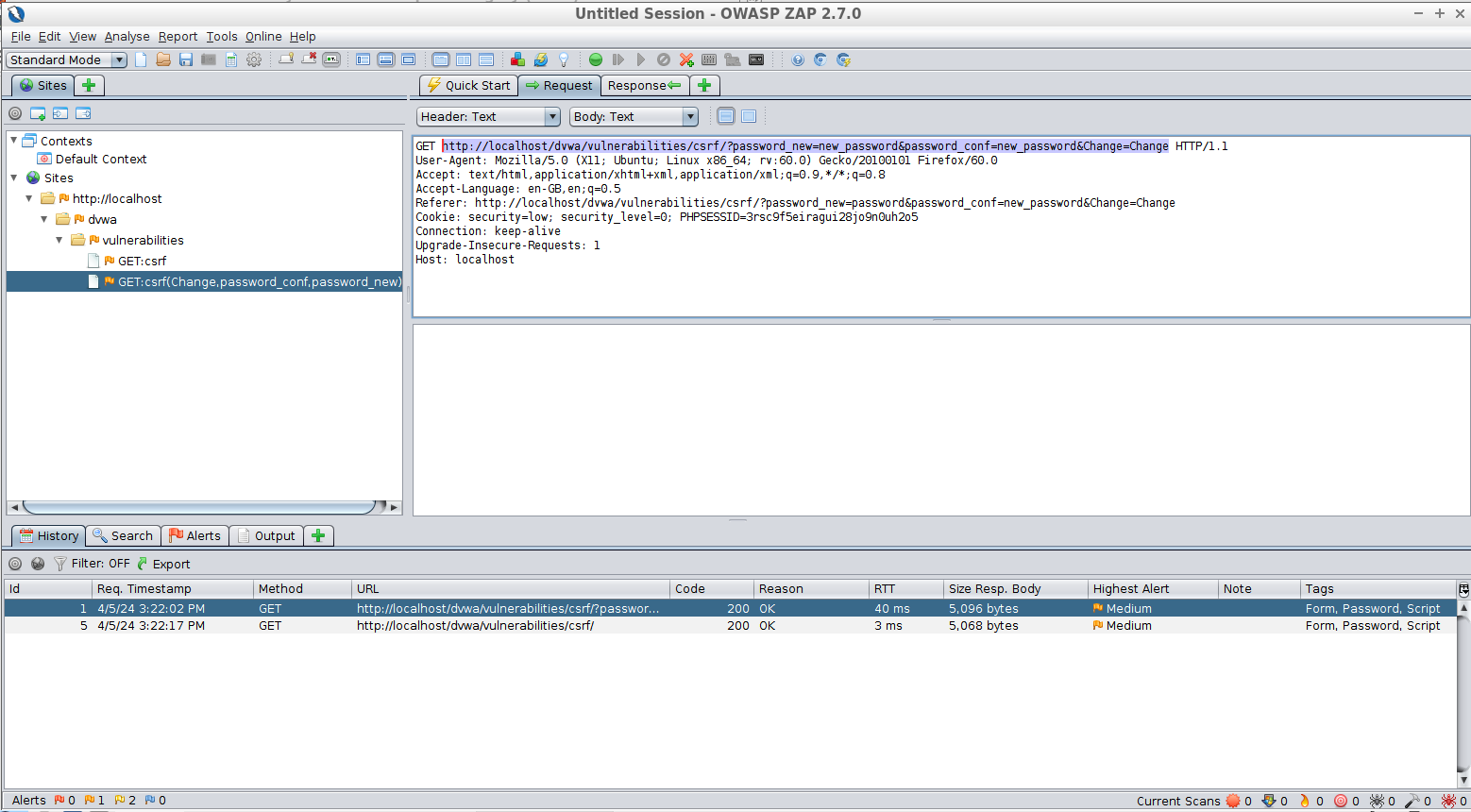
**Uncovering the CSRF vulnerability**

Using the CSRF page, a user can change their password by inputting a new password and then confirming that new password. We used the password “new\_password” in this case.



*Figure 2.9 - Searching for CSRF vulnerability by testing a password change request in DVWA.*

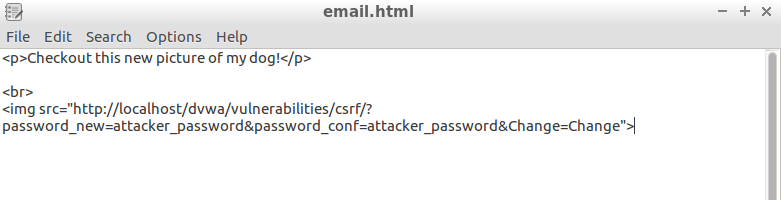
This HTTP conversation between the client and the server was captured via ZAP. It appears to be a GET request. The URL of this request was noted.



*Figure 2.10 - Analyzing the change password GET request for DVWA and taking note of the URL.*

**Compromising the web application**

The URL was changed so that the new password arguments in the URL will be “attacker\_password” so that we (the attackers) can log in with that password once the attack is complete. This new URL was put in the src attribute of an image in an email to a user we could attack.



*Figure 2.11 - Crafting an email with an image referencing the malicious URL we created.*

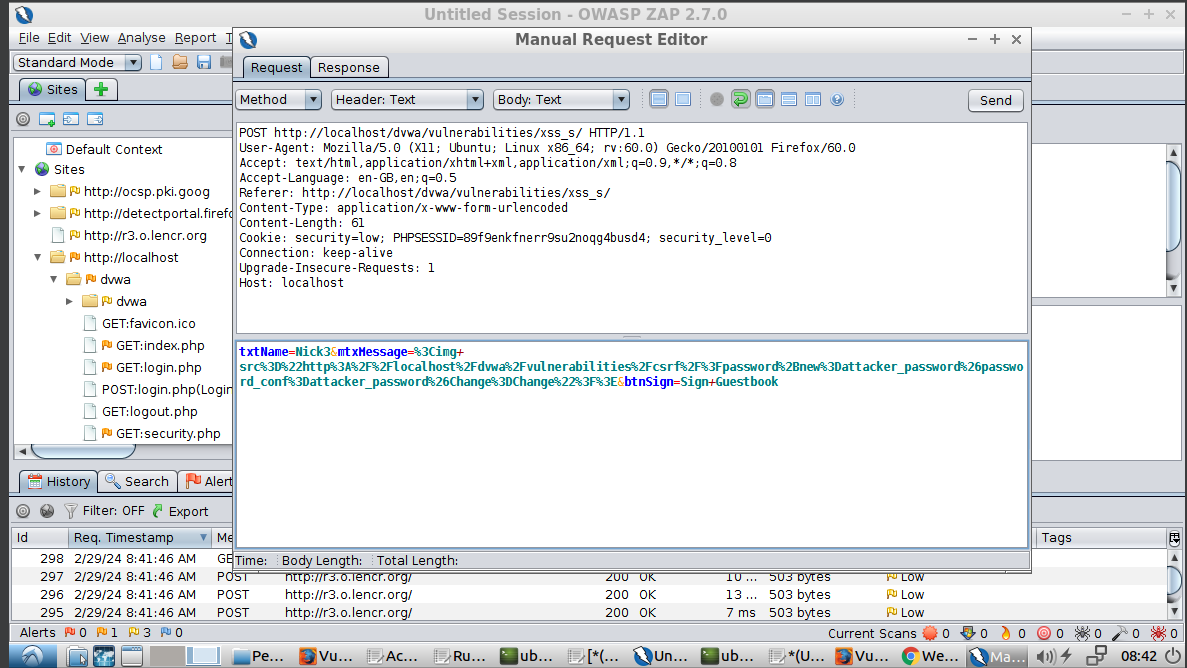
This email can now be sent to the user whose password we want to change. To them it will look like this:



*Figure 2.12 - Malicious email from the perspective of the attacked user.*

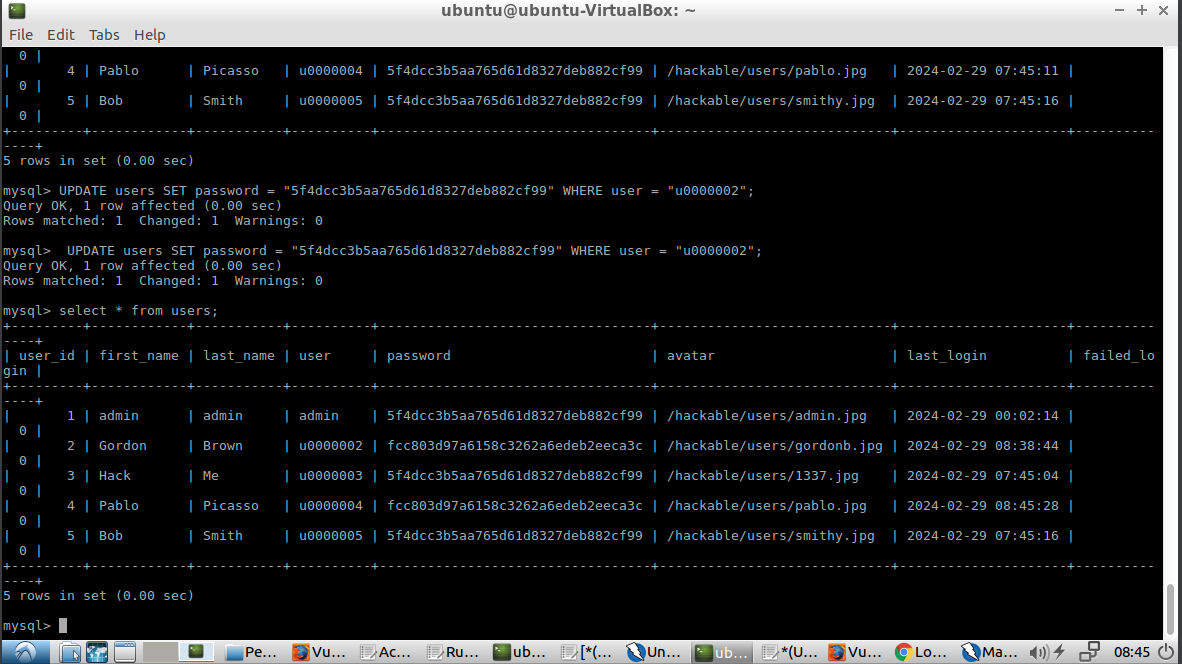
The email provider (gmail/outlook/etc.) will attempt to download the image by calling the GET request that we put as the image source. The GET request won’t return an image. If the user is logged in while viewing this email, this request will reset their password without their knowledge, and we (the attackers) can login with their username and the new password “attacker\_password”.

For our specific use case, we modified this attack slightly since we will be using the stored XSS exploit to deliver the payload instead of a different site. Therefore, it isn’t quite a “Cross-Site” request forgery, but the core idea of the exploit remains the same. We will use a modified request as seen in Figure 2.7 to save our payload to the DVWA database, as seen in Figure 2.13. The payload in an `<img>` html tag with the `src` attribute defined as a URL encoded endpoint to change the user’s password.

*Figure 2.13 - Modified request to bypass 50 max character limit and with a URL encoded endpoint to change the user’s password.*

Once a user logs in and views the guestbook, the browser will try to render the img tag, which executes the call to the endpoint and changes the user’s password automatically.

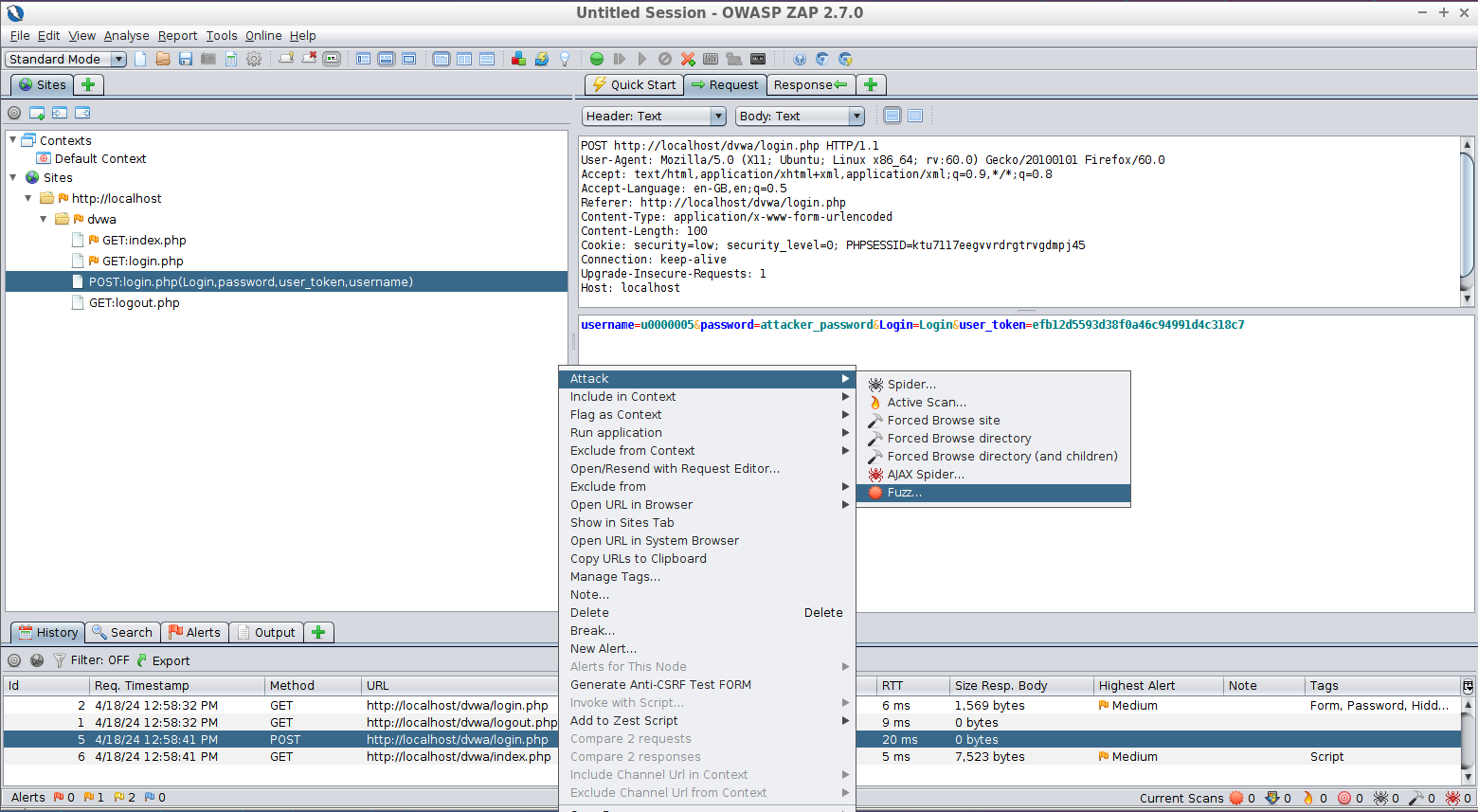
After the message in the guestbook was created, we simulated users u000002 and u000004 logging in and viewing the guestbook. Upon viewing the guestbook, their passwords were reset to “attacker\_password” due to this attack. While we simulated u000002 and u000004 falling victim to this attack, we (the attackers) wouldn’t know which users were affected, so in the next step, we will apply a brute force attack to determine which users’ passwords we reset. Figure 2.14 shows that the passwords in the database were actually changed.

*Figure 2.14 - Passwords for users u000002 and u000004 were changed (which is why they have the same hash value, but are different from the rest of the users)*

## Task 2.3: Guess username and passwords with brute force attack

In the previous steps, we were able to attack users into changing their password to the known value of “attacker\_password”. In this step, we will determine which users have fallen victim to this attack with a brute force on predictable usernames.

To uncover this, we attempted a login request using our attack account and recorded it using ZAP. Note that upon viewing the guestbook, this account also had its password changed to “attacker\_password”. Once recorded, we began a fuzzer attack using that request.

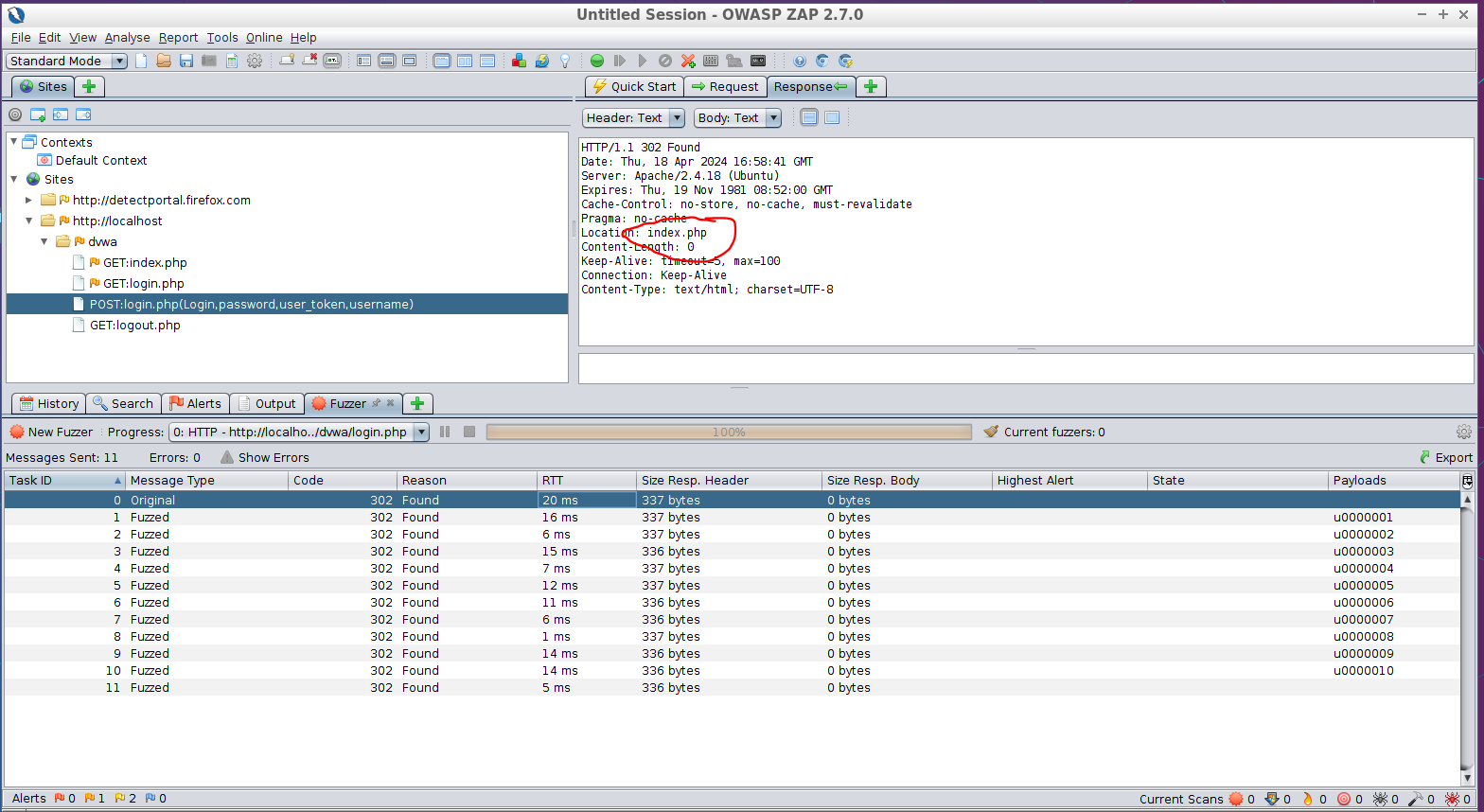
*Figure 2.15 - Capture login request and begin fuzzer attack based on the request*

In the fuzzer attack, we replaced the username with a payload which is a list of usernames that we expected to be generated sequentially: u0000001, u0000002, etc.



*Figure 2.16 - Setup of fuzzer attack with a list of usernames to attempt a login on other accounts.*

Then, we executed the fuzzer attack. For each request, successfully compromised username and password combinations will redirect to index.php as shown below.



*Figure 2.17 - Example of a successful login as a result of the fuzzer attack.*

Unsuccessful logins will redirect back to login.php as shown below.



*Figure 2.18 - Example of an unsuccessful login as a result of the fuzzer attack.*

As a result of the brute force fuzzer attack, we noted that u0000002 and u0000004 use the new password of “attacker\_password” and in addition to u0000005 which is the account owned by us, the attackers. Now we have control of 2 legitimate user accounts: u0000002 and u0000004. We can now access any information in their accounts that is only accessible to them.

**Task 3:** Based on the found vulnerable issues, what should you do to prevent attackers from hacking your web application?

**Task 3.1** Protection/prevention from a server-side hacking based on the uncovered vulnerabilities in Task 1

To keep your web application safe, start by staying up to date with software updates. That means regularly checking and applying patches for the web server, frameworks, and any plugins used. These updates often include important security fixes that can help plug vulnerabilities that hackers might exploit.

Next, tighten up the application's defenses by being selective about what it accepts. Implement robust input validation and sanitization techniques to block malicious code injections. Avoid including files from external sources unless absolutely necessary, and carefully manage user access to sensitive functions and data. By taking these proactive measures, you can significantly reduce the risk of server-side hacking and protect your web application from potential threats.

**3.2** Protection/prevention from the attack based on the uncovered vulnerabilities in Task 2

To safeguard against the multi-faceted attack outlined in Task 2, start with rigorous input validation and sanitization protocols to thwart XSS vulnerabilities. Employ robust password policies coupled with secure authentication methods like multi-factor authentication (MFA) to fortify access controls. Implement anti-CSRF measures such as token validation and utilize rate limiting and account lockout mechanisms to deter brute force attacks. Regularly audit the system for vulnerabilities using tools like ZAP or Burp Suite, educate users on safe browsing practices, and stay updated with evolving security trends to build a resilient defense against sophisticated web threats.

By integrating these security layers and fostering a culture of proactive vigilance, we can effectively shield the web application from intricate attacks, ensuring data integrity, user privacy, and overall system resilience against malicious exploits.

**Team peer-review table**

|  |  |
| --- | --- |
| **Name** | **Contributing Efforts in this project (0 ~ 100%)** |
| Sushruti | 100% |
| Joshua | 100% |
| Nick | 100% |
| Mario | 100% |