University of Sheffield

Lab Report For COM6015- 2022

Group Number 5 :

Jagpreet Jakhar, Rohanshu Sharma

Department of Computer Science

May 18, 2023

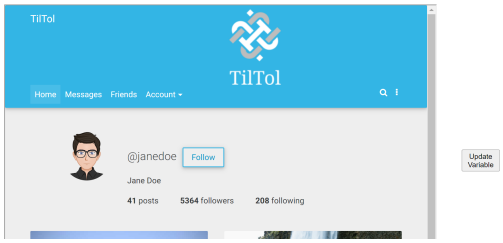
Contents

List of Figures

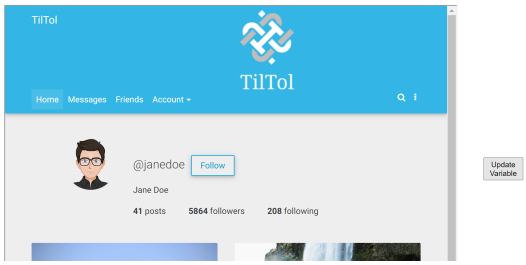
List of Tables

Chapter 1

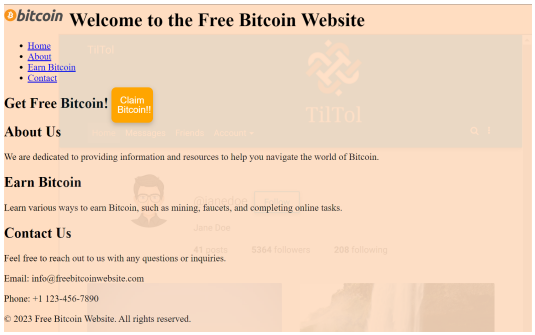
Question 1:Clickjacking



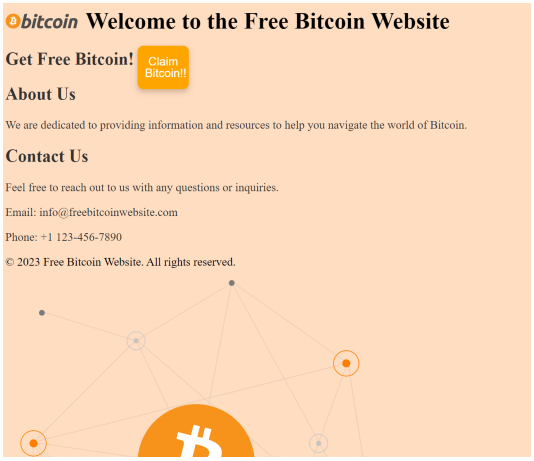
(a) *Before Update: followers:5364*



(b) *After update: followers:5864 (+500 on every press)*



(c) *Pre-Final*



(d) *Final Clickbait*

Figure 1.1: *Scam Chain*

The code files are attached as test.html for clickbait webpage, and joan.html for Joan Doe's tiltolpage.

```
<!DOCTYPE html>
/**
 * Retrieves and updates followers count of Jane Doe
 *
 * The access() function retrieves a reference to the iframe element
    containing Jane Doe's Webpage.
 *
 * The updateVariable() function accesses the document object of the iframe
    and then retrieves the element with the ID "followers_no", which is the
    number of followers  and
 * it updates its value by incrementing the current value by 500.
 * Clicking the button triggers the updateVariable() function.
 */
<script>
    function access() {
      var iframe = document.getElementById("victim_website");

      }
      function updateVariable() {
      var iframe = document.getElementById("jane");
      var iframeDocument = iframe.contentDocument || iframe.contentWindow.
    document;

      // Access and update the variable within the iframe
        var followers_no = iframeDocument.getElementById("followers_no");
        if (followers_no) {
          var currentValue = parseInt(followers_no.textContent);
          var newValue = currentValue + 500;
          followers_no.textContent = newValue;
        }
      }
    </script>
<iframe id="jane" src="joan.html" onload="access()" class="scam_frame"></
    iframe>
    <button onclick="updateVariable()" class="scam">Claim Bitcoin!!</button>

```

Code 1.1: Clickbait Code

The Clickbait puts the Joan Doe's page in an Iframe and then using JavaScript extracts the variable "followers_no" from the joan.html. Everytime the Claim Bitcoin button is clicked the number of followers increase by 500.

Chapter 2

Question 2: Wireshark

2.1 What is the victim MAC address?

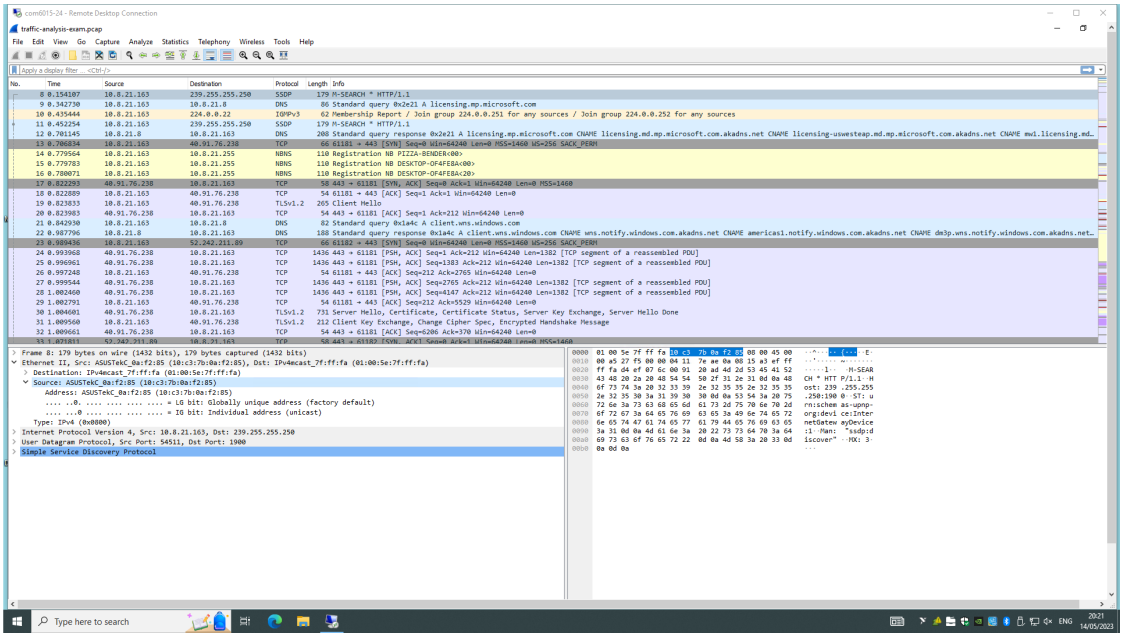


Figure 2.1: *Victim MAC Address*

As can be seen here the MAC address of the victim with the IP address - 10.8.21.163 is 10:C3:7B:0A:F2:85. This was found under the Source field of Ethernet.

2.2 What is the victim Host Name?

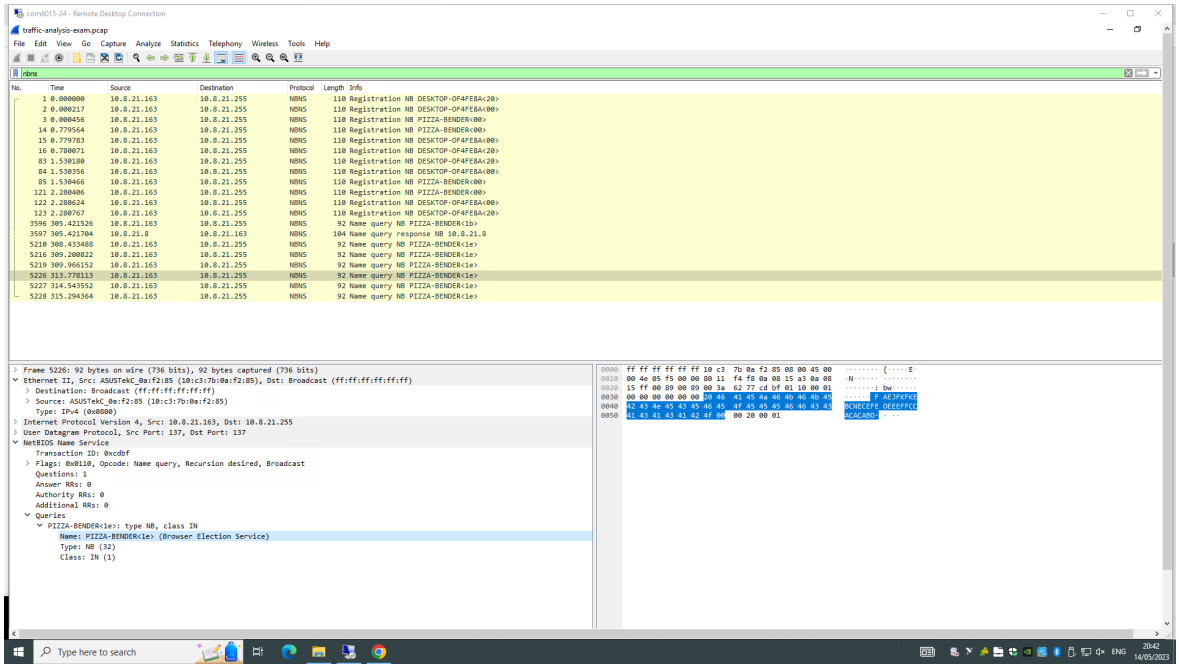


Figure 2.2: *Victim Host Name*

The host name is PIZZA-BENDER. This was found by checking the NetBIOS which is used by Microsoft Windows for its name resolution function.

2.3 What is the client operating system, and what day and time the malware was executed?

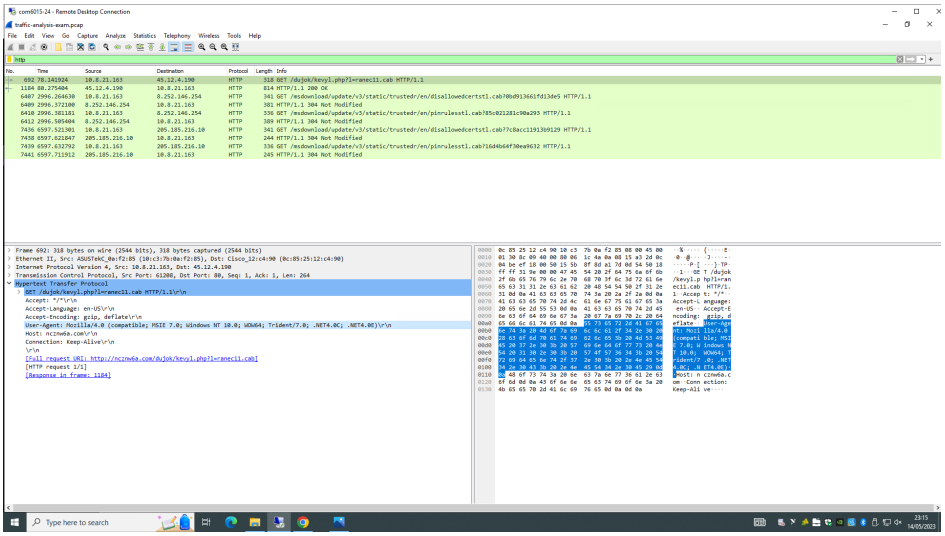


Figure 2.3: *Victim Client OS*

Here HTTP filter was used and as can be seen that there is only one request that gave code 200(success). So the malware might have been downloaded from here. Now as can be seen the HTTP field in the User-Agent that the client operating system is Windows NT 10.0.

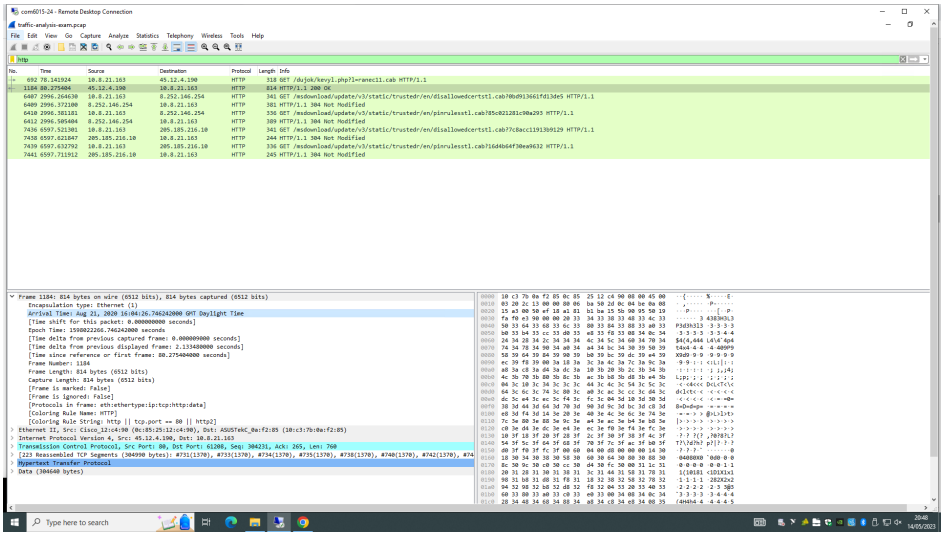


Figure 2.4: *Date and Time of attack*

Also the day and time on which the attack was executed, which can be seen in the particulars of the frame, is August 21, 2020 at 16:04.

2.4 Where was the intruder file location?

As can seen in the full request URI field the HTTP request is as follows : http://ncznw6a.com/dujok/kevyl.php?1=ranec11.cab .

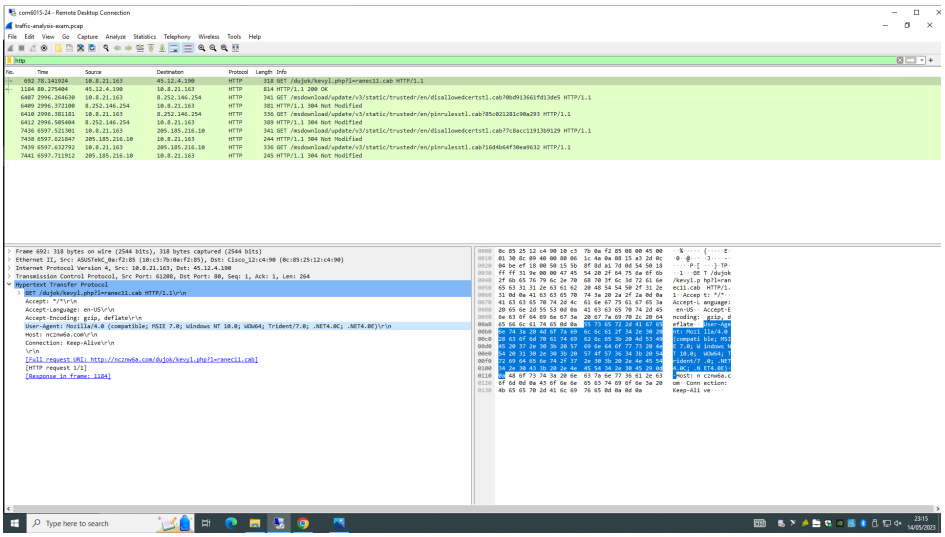


Figure 2.5: *File Location*

2.5 What is the intruder IP address? Show the geographical location of the intruder using the Wireshark endpoint feature.

As we can see in the above images, the IP address is 45.12.4.190. Now using the Wireshark endpoint feature we got the geographical location as:

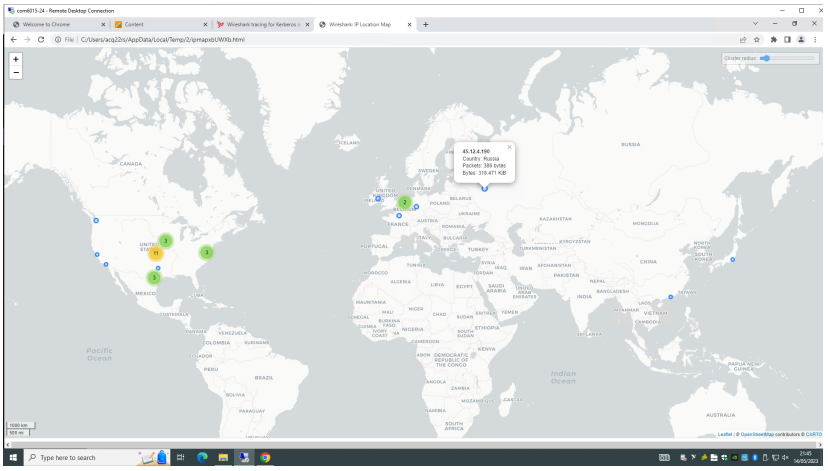


Figure 2.6: *IP Address*

2.6 Extract the malicious file (in your VM) and extract its hash key using Linux terminal/ windows cmd (SHA 256).

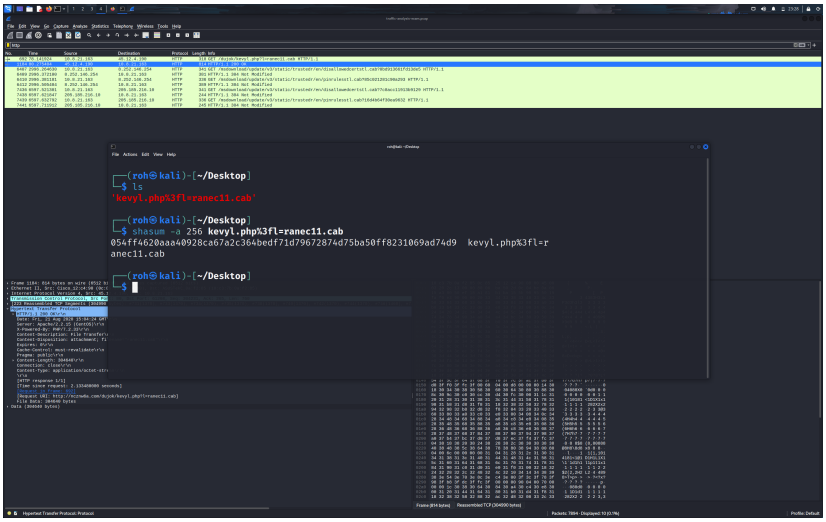


Figure 2.7: *Malicious File*

The pcap was opened in Kali to see the hash as the as the vm system did not have rights to download malicious files. The file was exported from Wireshark and the hash(SHA 256) as seen was extracted.

2.7 Use the extracted SHA 256 code, search on the Internet and provide the malware description.

We can see that when the hash was entered on the internet the following details were found. The malware's name is IcedID and it is a trojan that is designed to steal banking credentials of the victims. It steals payment information and can also deliver another viruses. This explains the unauthorised withdrawal of $5000.

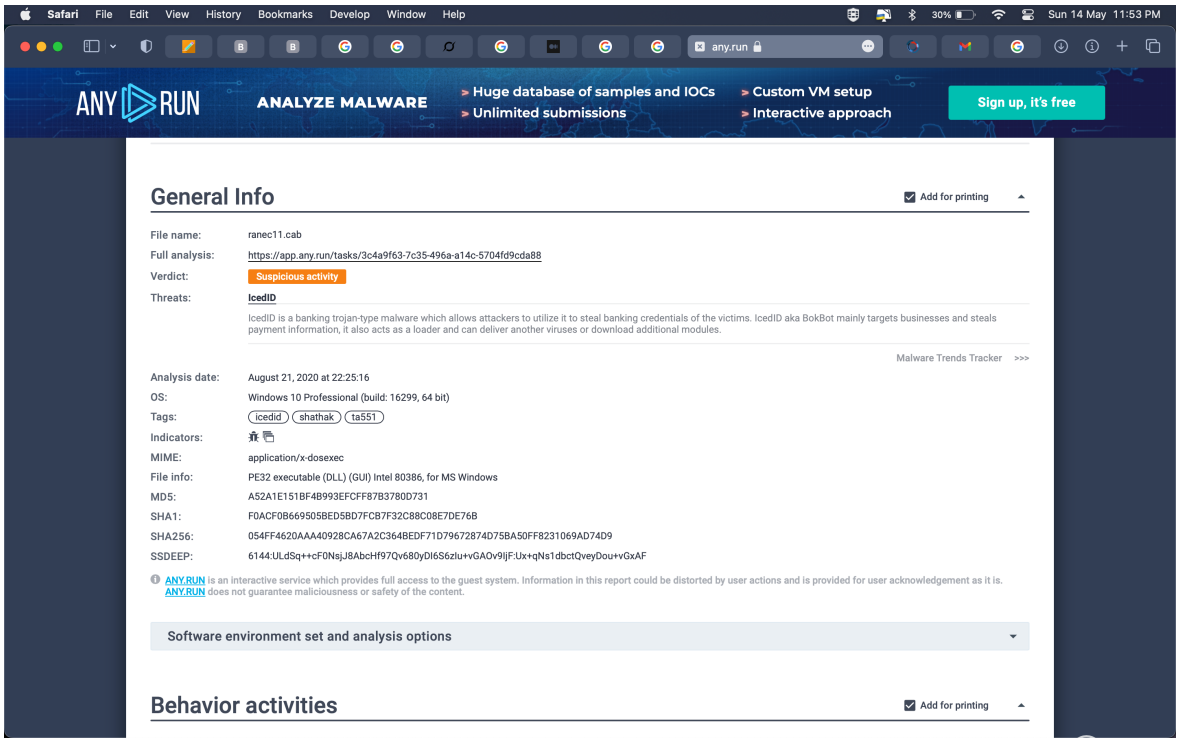


Figure 2.8: *Malicious File*

2.8 Which user is accountable for downloading the malicious malware from the Internet? provide Account Name by searching inside Kerberos packets (hint: find the answer in the KRBS packet (237) 'CNameString' section)

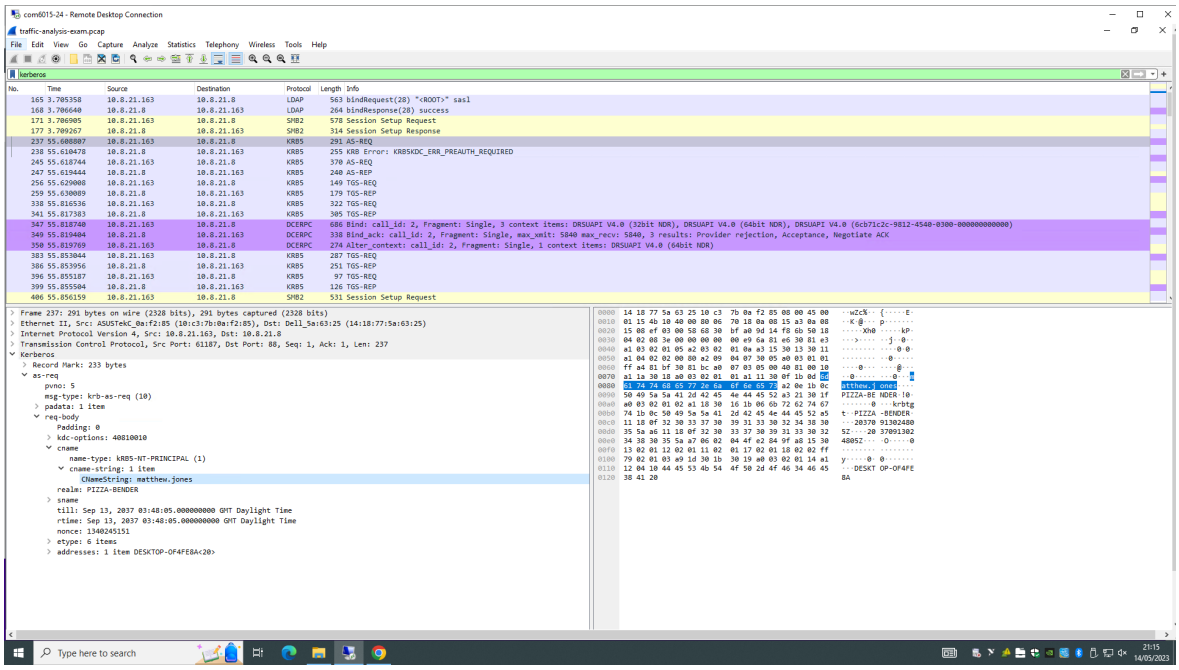


Figure 2.9: *User Accountable*

As can be seen the kerberos packets were explored. In the cname string the name Matthew.jones is seen. It can be inferred that he is responsible for downloading the malicious malware.

Chapter 3

Question 3 : SQL Injection

3.1 Extract the column names of the table that contains the user data

The DVWA was set to low security before attempting this question. And the following statement was put into the text box:

```
1' OR 1=1 union select null, table_name from information_schema.tables #
```

Code 3.1: Extract Tables

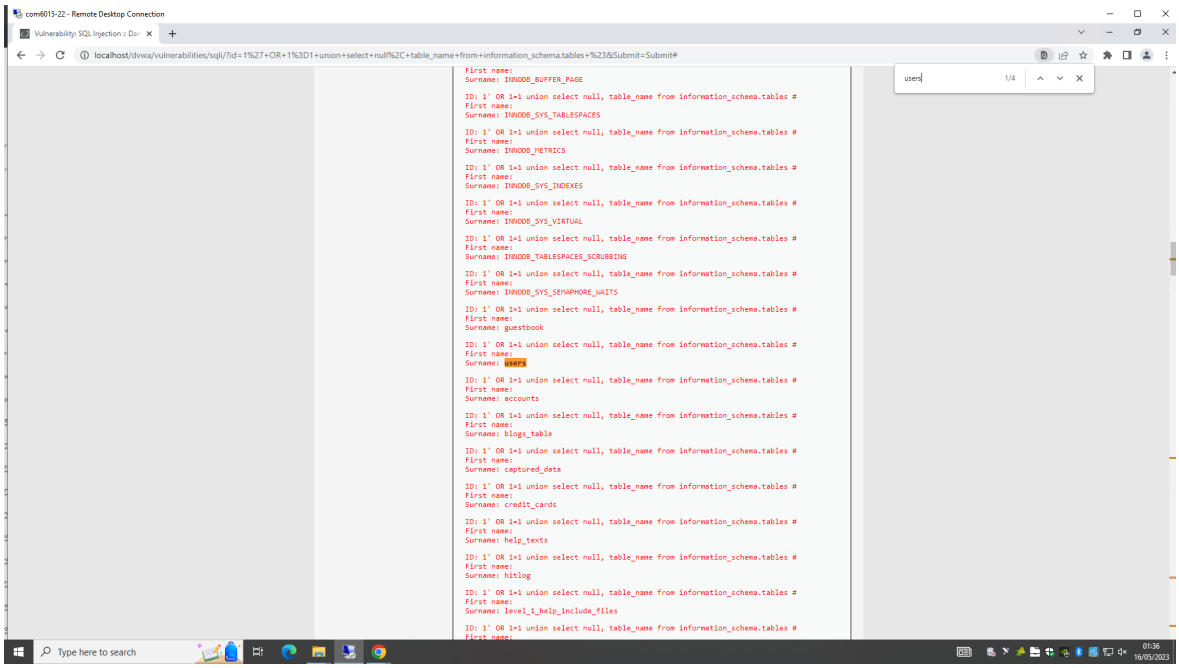


Figure 3.1: *Tables Present*

This shows all tables in database

- The 1' OR 1=1 is used to manipulate the logic of the SQL query such that regardless of the original query the condition after OR is always true.

- The union is used to combine the results of the select and return the output.
- The null returns in the first column field(First name)
- The second request: table_name returns in the second column field(surname).

This logic will remain the same throughout the answers in this section. Now as can be seen we are seeing which tables are present there in the databases and as per the question Users table is the one we are interested in.

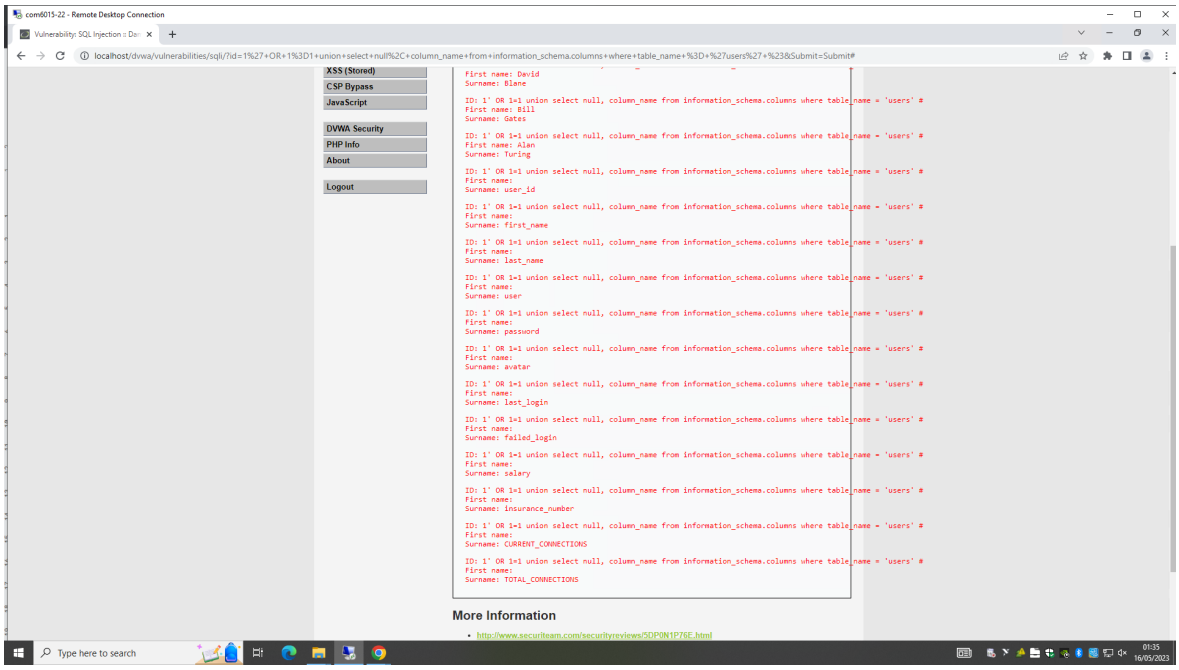


Figure 3.2: *Column Names*

Now as we know the target we select the columns from that users tables and return that in the second field of the site by using the following method:

```
1 1' OR 1=1 union select null, column_name from information_schema.columns
  where table_name = 'users' #
```

Code 3.2: Extract Column names

We are successfully able to extract the tables.

3.2 Attempt the previous injection in medium security, update your query to bypass the new security measures

Now as required, the setting was first changed to medium security in the DVWA. For this OWASP ZAP was used to do the SQL Injection. Now as we know mysqli_real_escape_string prepends a backslash to potentially dangerous characters, so we can just remove the “ ‘ “ from places and it should still work. But for where it is necessary we are using the hex code values of the text instead of the word instead which helps us to successfully inject the site. So the command now looks like:

```
1 1 or 1=1 union select null, column_name from information_schema.columns
  where table_name=0x7573657273
```

Code 3.3: Extract Column names Medium Security

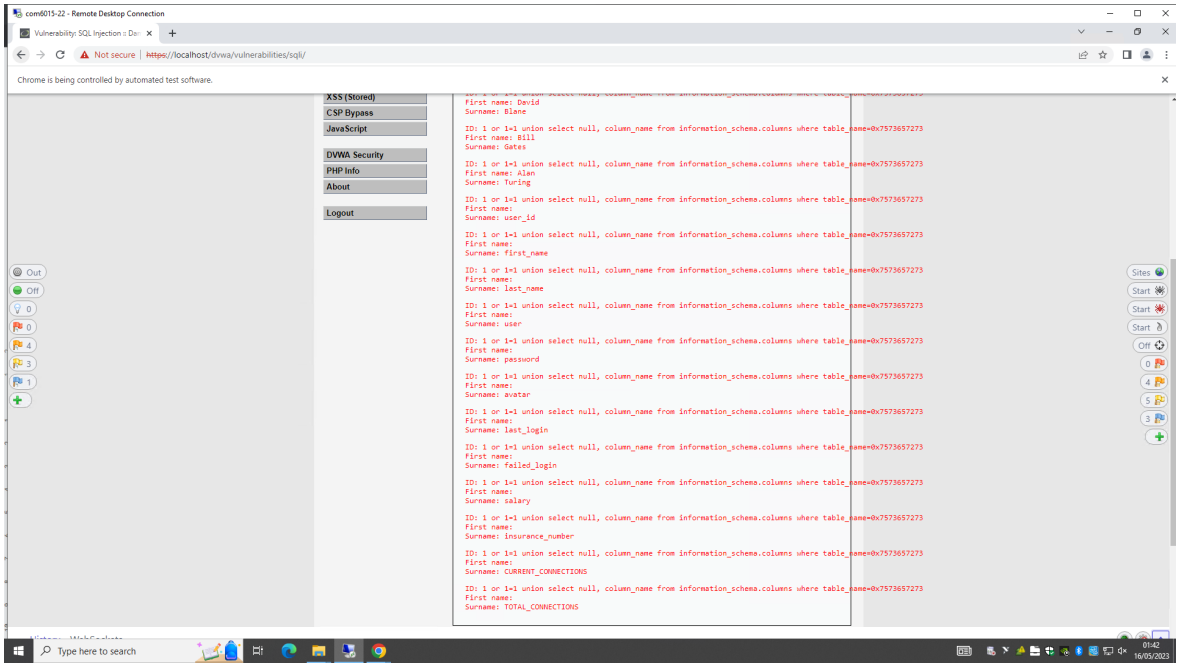


Figure 3.3: *Column Names Medium Security*

Here we changed the “users” to its corresponding hex code value which is 0x7573657273.

3.3 Locate the location of the database on the remote system (File path the database is stored)

The following command is used for the Injection:

```
1 or 1=1 union select database(), @@datadir
```

Code 3.4: Locate Database

Here the database() returns the name of the database in the first column field. Then in the next column field we use @@datadir which returns the location of the database. So the location that is obtained is the following as seen in the image: `C:\xampp\mysql\data\`

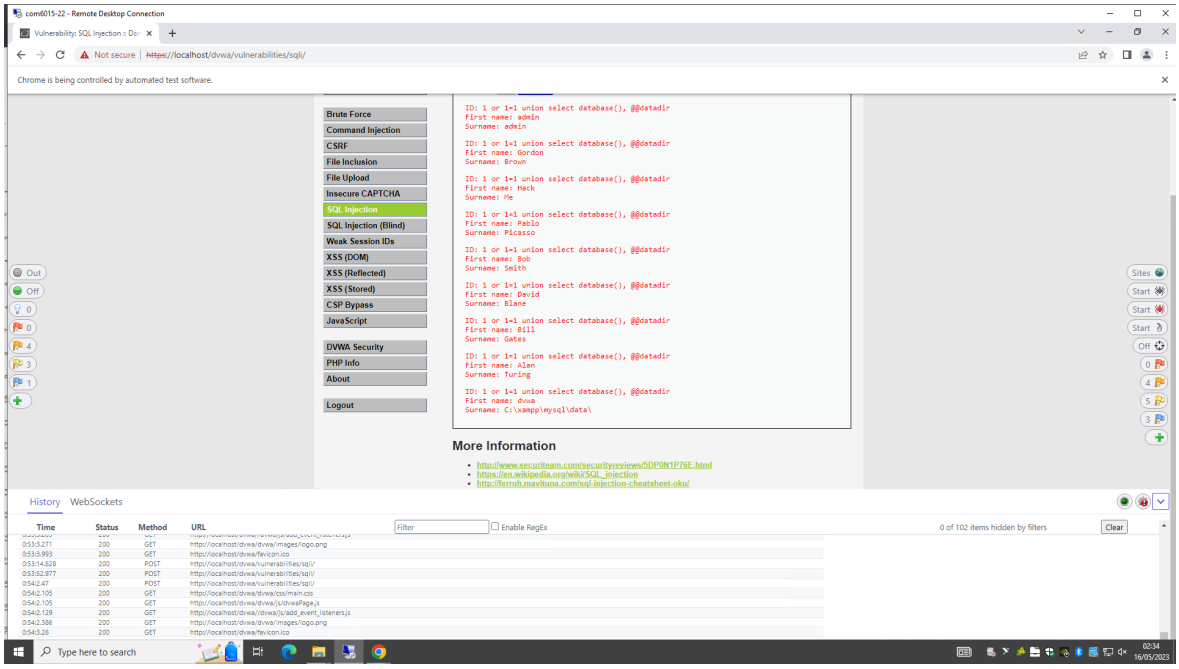


Figure 3.4: *Database Location*

3.4 Read a file (e.g. passwords) from the discovered path

The following command is used to perform the SQL Injections:

```
1 or 1=1 union all select null, load_file(0
   x433a5c78616d70705c6d7973716c5c646174615c70617373776f7264732e747874)
```

Code 3.5: Read File

Now as seen, the first column field returns a null value while the second column field returns the contents of the file that is mentioned by using the function load_file() which reads the contents of a file on the server. The absolute path of the file is converted to hex code value to avoid getting errors due to mysqli_real_escape_string function. So the `C:\xampp\mysql\data\passwords.txt` is converted to

`0x433a5c78616d70705c6d7973716c5c646174615c70617373776f7264732e747874`

Now as seen in the image we are successfully able to read the contents of the file.

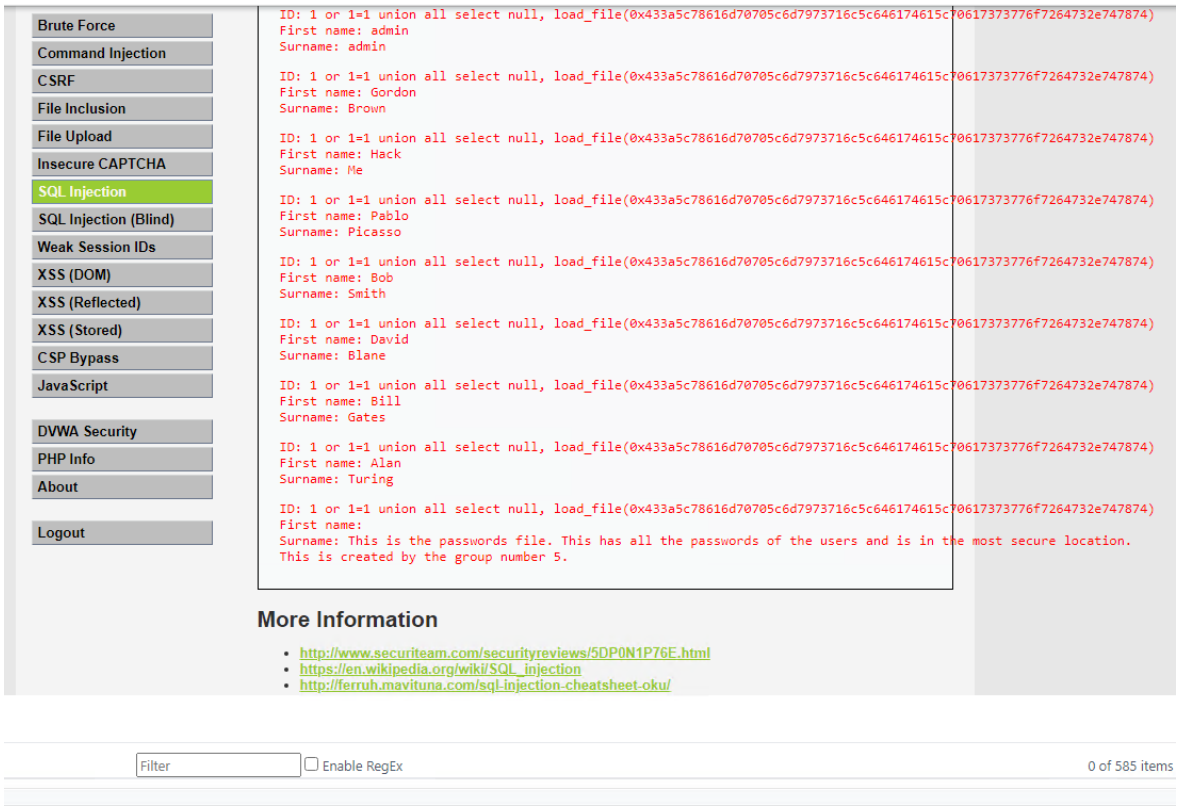


Figure 3.5: *Contents of File*

3.5 Discover the users with the highest and lowest salaries

We used the following command for the SQL Injection:

```
1 or 1=1 union select concat(first_name,0x20,last_name),salary from users
    where salary = (select MAX(salary) from users)or salary = (select MIN(
    salary) from users)
```

Code 3.6: Salaries

- It is seen that it returns the full name of the user and that is done by concatenating the content of the columns first_name and last_name and a space character is used which is represented in hex code value which is 0x20. The second column field returns the salary of the chosen user.
- The other part of command selects the required variables from the users table where the conditions are met. It sets the condition for the salary column. It retrieves the rows where the salary is equal to the maximum salary or the minimum salary from the same users table.

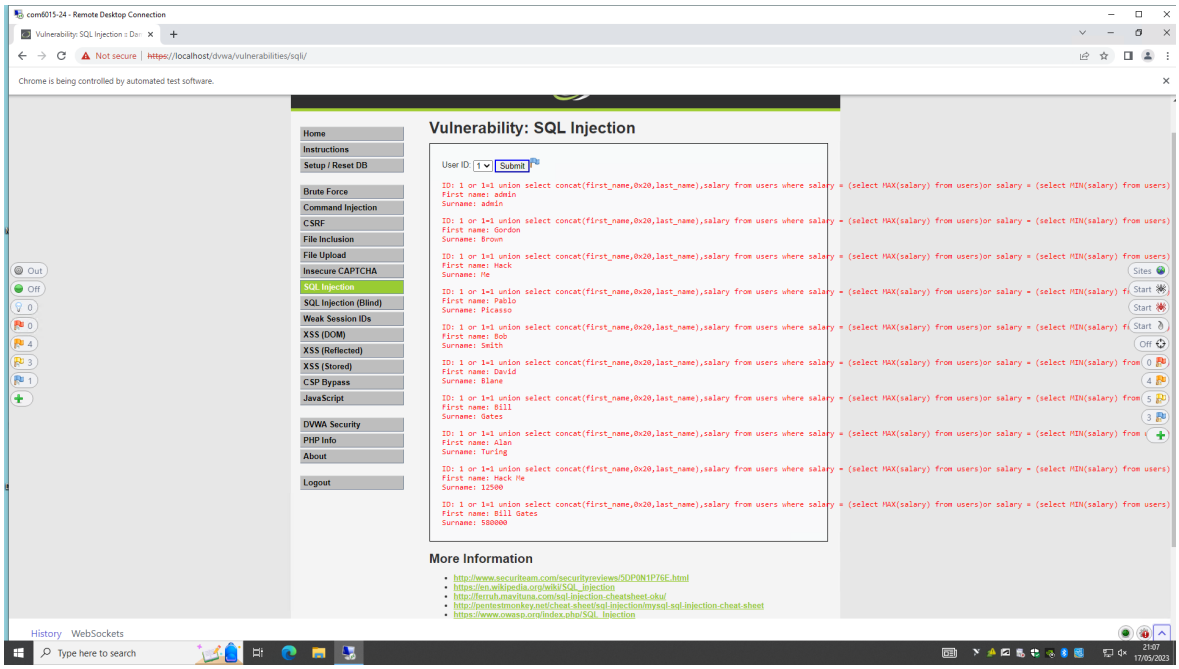


Figure 3.6: *Highest Salary*

We can see that the max salary is of Bill Gates with a salary of 580000 while Hack Me has the lowest salary of 12500. This means the injection is successful.

3.6 Who has the insurance number: 53779132

The following command is used for the SQL Injection:

```
1 or 1=1 union select concat(first_name,0x20,last_name),insurance_number from
    users where insurance_number = 53779132
```

Code 3.7: Insurance Number

- It is seen that it returns the full name of the user and that is done by concatenating the content of the columns first_name and last_name and a space character is used which is represented in hex code value which is 0x20. The second column field returns the insurance_number of the chosen user for the users table.
- The other part of command selects the required variables from the users table where the conditions are met. It sets the condition for the insurance_number column. It retrieves the rows where the insurance_number is equal to 53779132.

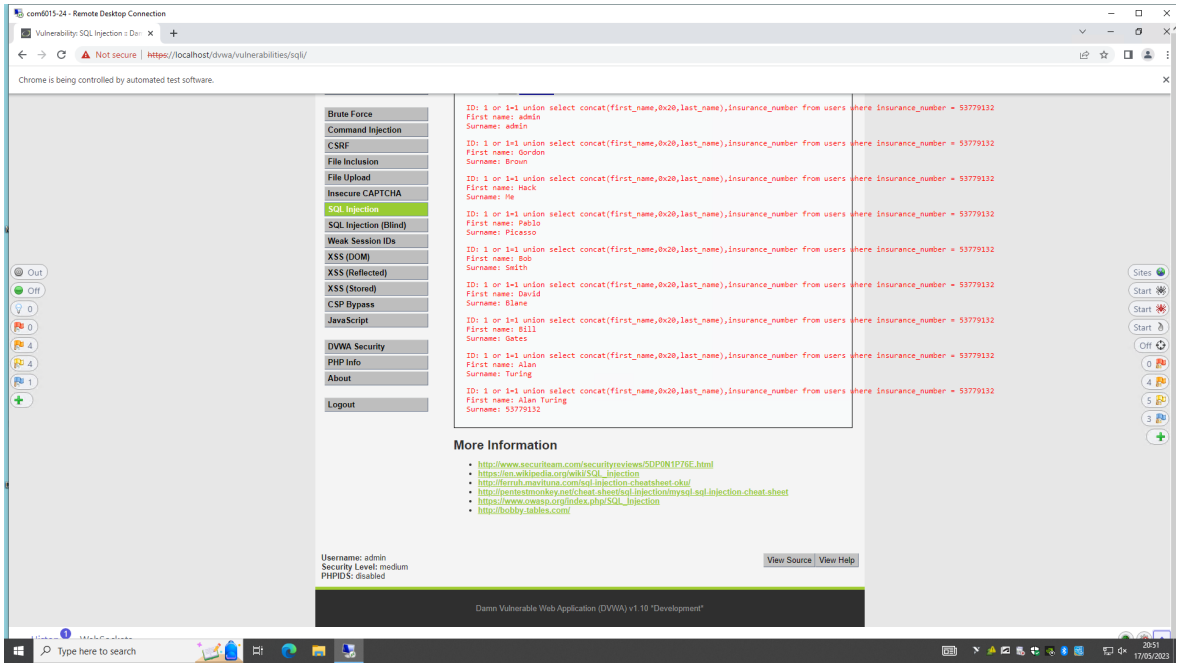


Figure 3.7: *Insurance Number*

We can see that Alan Turing had the insurance number as 53779132. Which says the injection was successful.

Chapter 4

Question 4 : XSS

4.1 Analyse the PHP source code and perform an attack that works on Reflected, Stored and DOM XSS pages. Provide a description for each relating to the source code as to why your given attack works.

REFLECTED XSS:

Figure 4.1: *Reflected XSS Code*

As we can see the "name" variable will strip the <script>tag and makes it impossible to use the same. To attack we can simply use another tag that has some event handlers. So we are going to use:

```
<body onload = "alert('TEAM 5 HAS HACKED YOU!!!')"></body>
```

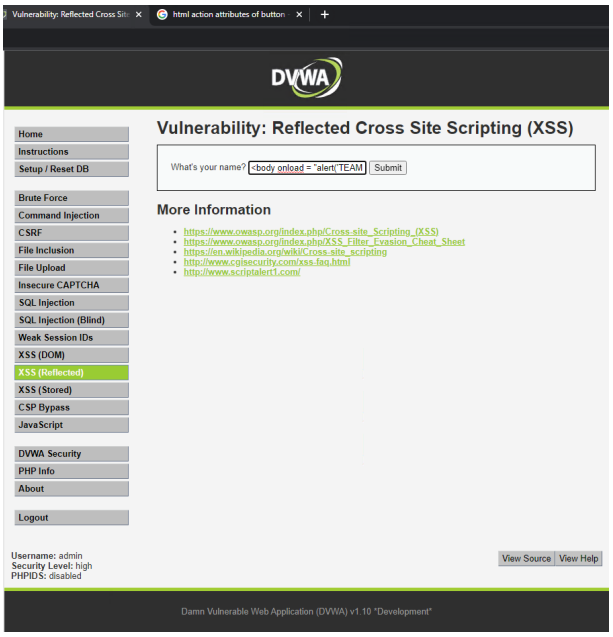


Figure 4.2: *Reflected XSS attack*

This is a simple body tag which when loaded gives an alert message to the user. As shown below:

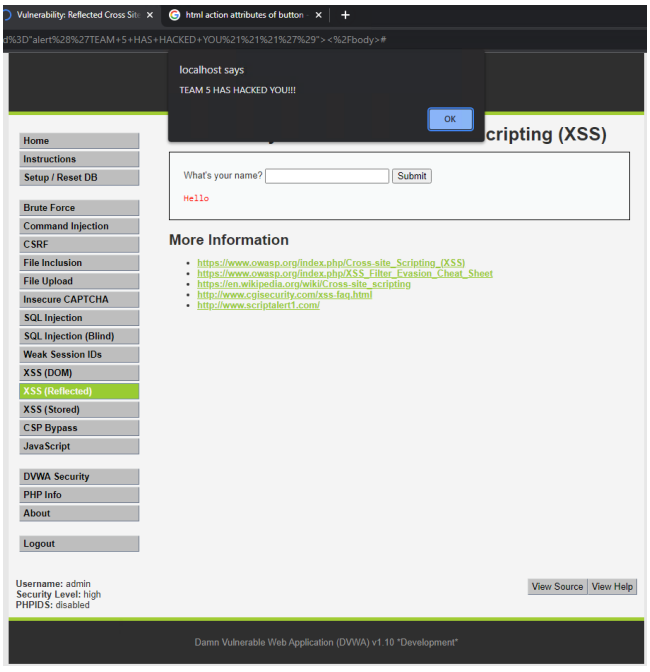


Figure 4.3: *Reflected XSS hack*

This code will be reflected back to the victim as part of the website's response and will not be on the server.

STORED XSS:

We will analyse the code first:

Figure 4.4: *Stored XSS code*

As can be seen there are two fields, message and name. We can see the message field is using strip_tags() and htmlspecialchars() was used which prevents us from using all the html tags here so we can not attack using this field as proper sanitation methods are incorporated. While in the name field similar to the reflected one the sanitation of the <script>tag is done which means we can use other tags to perform the injection of our code. So we will be using the same attack here too. And the following code was used:

```
<body onload = "alert('TEAM 5 HAS HACKED YOU!!!!')"></body>
```

But before that when we inspect the page we can see that the maxlength of the input field for the name was just 10 characters long which will need to be changed so we changed it to 1000 so we can input as many words we want to in the field.

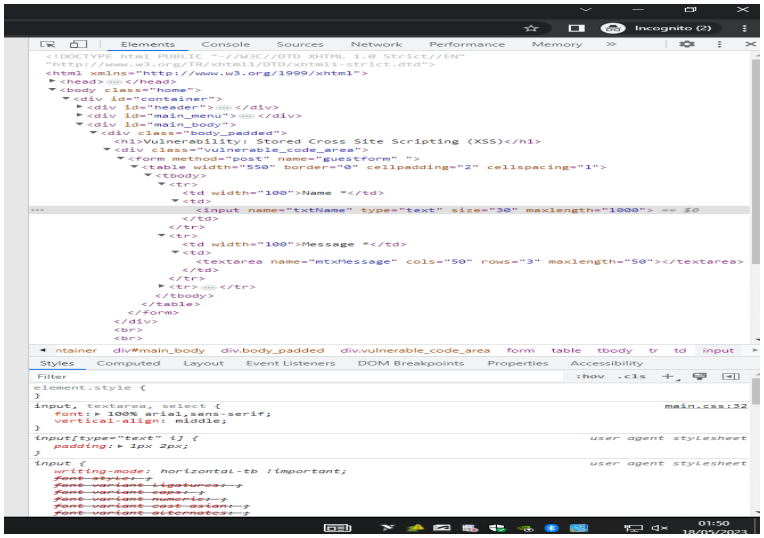


Figure 4.5: *Stored XSS Inspect*

Now we will attack the page using the name input field by putting our code there.

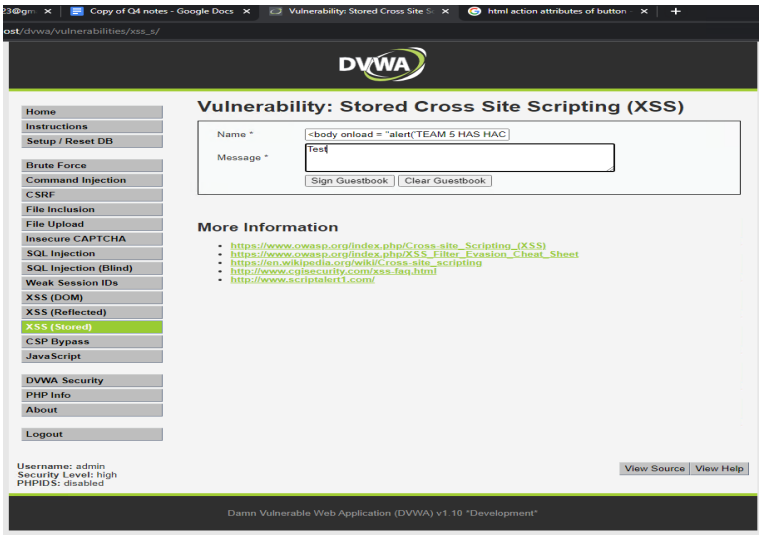


Figure 4.6: *Stored XSS Attack*

And now when we press the Sign Guestbook button we can see that the alert box has appeared. But the difference this time is that the malicious code is now sent to the server and will be permanently stored there and now anytime any user tries to access the site this alert box will be presented to them as this will persist and this is why it is the most dangerous XSS out of all 3.

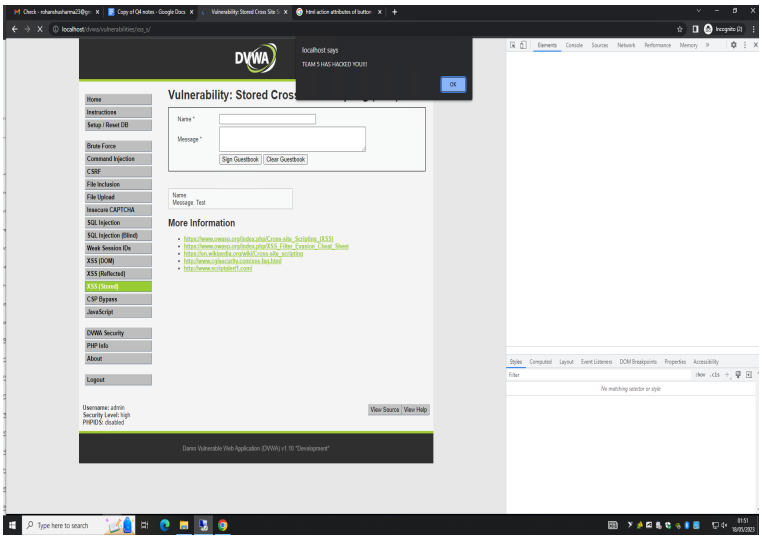


Figure 4.7: *Stored XSS Hacked*

DOM XSS:

We will first look at the code. We can see that the code uses switch-case and only these 4 languages will be selected from so we can not make changes.

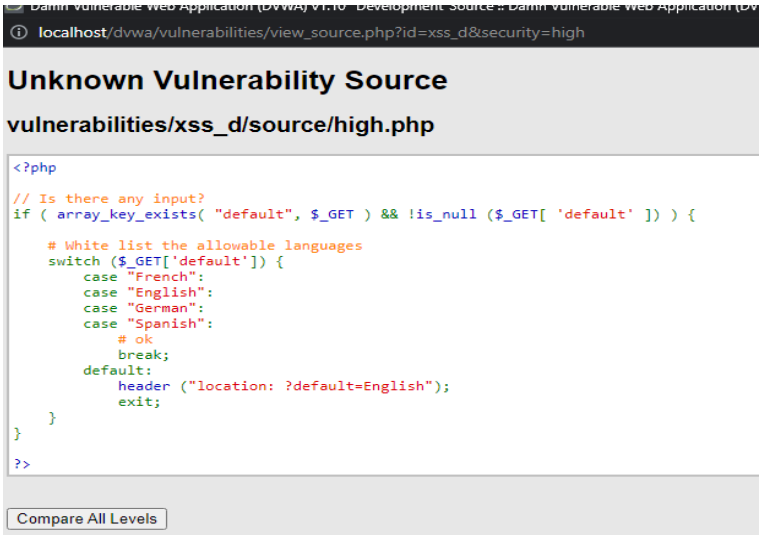


Figure 4.8: *DOM XSS Code*

Also now when we inspect the page we can see that the script takes 8 characters as input after the "default=" when the URI is sent to the server. So we can terminate the query by using "#" and then inject our code in the URI as the server will receive the correct input but the rest will not be sent to be processed but the browser will show the output of the script.

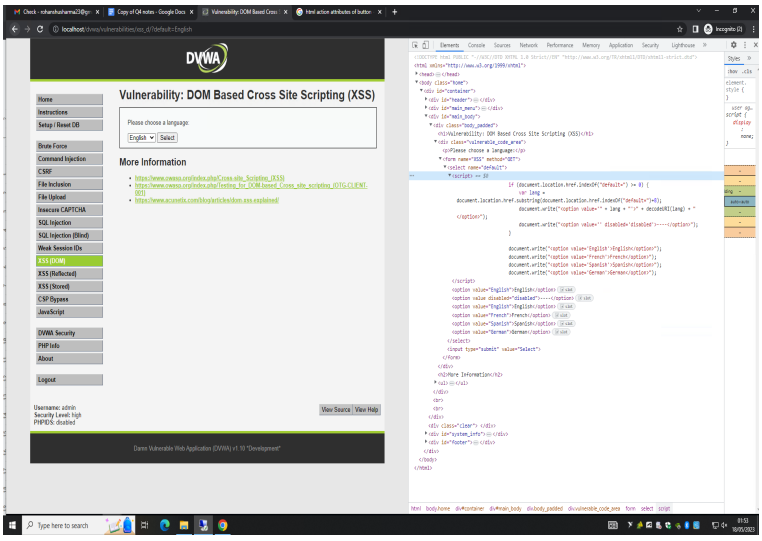


Figure 4.9: *DOM XSS Before*

Now we inject the code into the search bar after appending a hashtag. The following code was used:

```
<script>(alert('TEAM 5 HAS HACKED YOU!!!'))</script>
```

The following message is seen when executed.

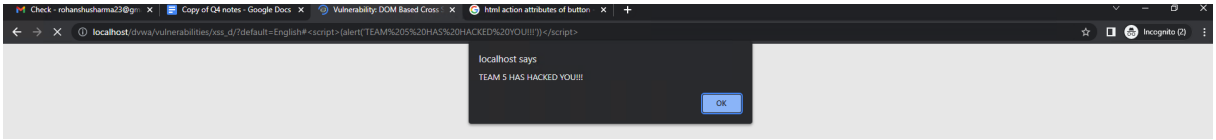


Figure 4.10: *DOM XSS Attack*

DOM XSS targets the user's browser and the script is not sent to the server. We can again inspect the page and see that the following changes are made in the code. We can see that the script that we used is not triggering the filter that was used against it.

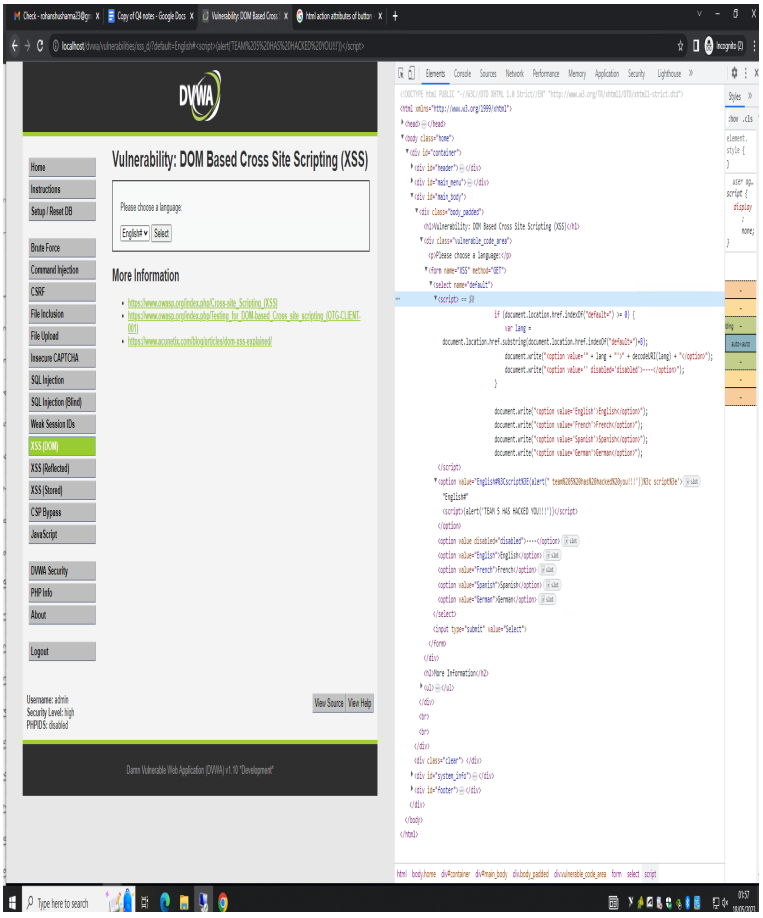


Figure 4.11: *DOM XSS After*

Chapter 5

Question 5 : Defence

5.1 In a language of your choice, write a short function which handles input from a user which would prevent an SQL injection attack and a Stored XSS attack. Provide a short description of where/why your function is safe against these vulnerabilities.

SQL Injection works by mixing data and code in user input, which is achieved by using symbols and Conditionals. The code removes all such characters from the user input.

Other measures to ensure strong protection is to use Custom error handling, Prepared Statements etc

```
function sanitizeHtml(input) {
   // Remove all special characters that could be used to for SQL Injection.
   input = input.replace(/[;|,|'|"|-]/g, '');
```

Code 5.1: Code Sanitizer SQL

Stored XSS attacks work by including HTML code and javascript scripts to attack the system, The Function ignores and escapes all HTML characters, It also escapes all Javascript characters to ensure protection against Stored XSS attack.

```
   // Escape all HTML characters so that they are not interpreted as markup
    to facilitate Cross side scripting.
   input = input.replace(/&/g, '&amp;').replace(/</g, '&lt;').replace(/>/g, '
    &gt;');

   // Escape all JavaScript characters so that they are not interpreted as
    code.
   input = input.replace(/</g, '&lt;').replace(/>/g, '&gt;').replace(/"/g, '&
    quot;').replace(/'/g, '&#039;');
    //&#039 == '
   return input;
}
```

Code 5.2: Code Sanitizer XSS

Chapter 6

Question 6: Buffer Overflow

6.1 Choose one of the known buffer overflow vulnerabilities and write a half-page description of it. Aspects that you should cover are:

The FORCEDENTRY buffer overflow vulnerability, also known as CVE-2021-30860, was an integer overflow vulnerability when collating referenced segments[1] resulting in triggering a heap buffer overflow in the ImageIO JBIG2 decoder, used by ImageIO library in CoreGraphicsAPI used in Apple. The entry point for attack was a fake gif containing JBIG2(decoder codec) for PDF which was sent as iMessage, the JBIG2 codec contained code which was Turing Complete i.e. operators like AND, OR, XOR, and XNOR, enabled the attacker to perform bit-level operations on memory regions at arbitrary offsets. Tallowing for arbitrary memory access and computation of any computable function.

6.1.1 Which systems were affected by the vulnerability?

The vulnerability affected all versions of iOS and macOS prior to iOS 14.8 and macOS Big Sur 11.3.1. It is estimated that over 1 billion devices were vulnerable to the attack at the time it was discovered.

6.1.2 When was it discovered, reported and fixed?

This vulnerability was discovered in March 2021 by Citizen Lab, The vulnerability was patched by Apple in iOS 14.8 and macOS Big Sur 11.3.1 updates released for its devices.

6.1.3 What were the known attacks exploiting it, and what were their consequences?

This vulnerability was used by NSO Group for its zero-click iMessage exploit to spy on Saudi activist and Al-jazeera journalists[2].The affected phone was analysed by Citizen Lab and further analysis was done by Ian Beer of Project Zero[1] from Google.NSO was added the "Entity List" by the US government, severely restricting the ability of US companies to do business with NSO.

6.2 Propose a modification of the C programming language that would mitigate buffer overflow vulnerabilities, and discuss the implications of this modification.

Buffer Overflow vulnerabilities have plagued C language since the 1980s, as C allows for direct access to memory and lacks strong object typing.

One Modification that would help in mitigating buffer overflows would be the use of canaries or stack cookies[3] during runtime, They are random values which are placed on stack after each buffer to check for buffer overflows.They may contain other bytes, such as newline characters, that frequently terminate the copying responsible for string-based buffer overflows[4] or some bits unknown to the attacker to prevent return-address clobbering with an integer overflow.

However, they cannot protect against a direct overwrite (by exploiting an indexing error) as they only check for corruption at function exit.

Implications:

- Overhead: Adding stack canaries may add some overhead to the program and therefore slow execution.
- Compatibility issues may arise due to so many legacy code not using stack canaries.
- Using Stack Canaries leads to Reduced Vulnerability Surface and Enhanced Security

Other modifications that are more robust and can guard against more sophisticated attacks are :

1. Address Space Layout Randomisation(ASLR), it provides most comprehensive protection against attacks and is only vulnerable to information leakage attacks.
2. Using Data execution Prevention(DEP) for non-executable stacks, However this approach is vulnerable to Return-Oriented Programming Attacks.
3. Rearrangement of local variables, so that scalar variables are above array variables, so that in case of overflow, static variables are not affected.
4. Bound Checking to check for variables are in appropriate buffers
5. Null termination for handling string buffers
6. Using Static Analysis tools and using secure alternatives to vulnerable functions such as strcpy(), cand writing secure and high quality code.

Bibliography

[1] Google Project Zero. A deep dive into nso zero-click exploits. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html, December 2021.

[2] Citizen Lab. Forcedentry: Nso group imessage zero-click exploit captured in the wild. *Citizen Lab Research Publication*, September 2021. URL https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/.

[3] Laszló Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013. URL https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf.

[4] Steve Anderson. Low-level security by example. *University of Pennsylvania*, 2019. URL https://www.cis.upenn.edu/~sga001/classes/cis331f19/resources/low-level-security-by-example.pdf.

Appendices