Buffer Overflow

Computer Security CSE 481

Tatiana Ensslin - October 5, 2015

Introduction

In this lab, one will use the given program which contains a buffer-overflow vulnerability in order to exploit the vulnerability and gain root privilege. Throughout this lab, one will be guided through several schemes that have been implemented in the operating system in order to learn how to counter attack the buffer-overflow attacks and whether they were successful or not.

Task 2.1
Initial Set Up

In order to execute the following tasks within this lab, we must first turn off the randomize memory, stop the execution of stacks and turn off the stack guard.

```
[10/05/2015 16:58] seed@ubuntu:~$ su root
Password:
su: Authentication failure
[10/05/2015 16:58] seed@ubuntu:~$ #sysctl -w kernel.randomiz
e_va_space=0
```

Above we stop the random memory. We turn off the stack guard and stop the execution of the stacks for specific programs. Therefore, you will see these steps throughout the remainder of the tasks for the lab. Completing these commands allows for us to simplify the attacks.

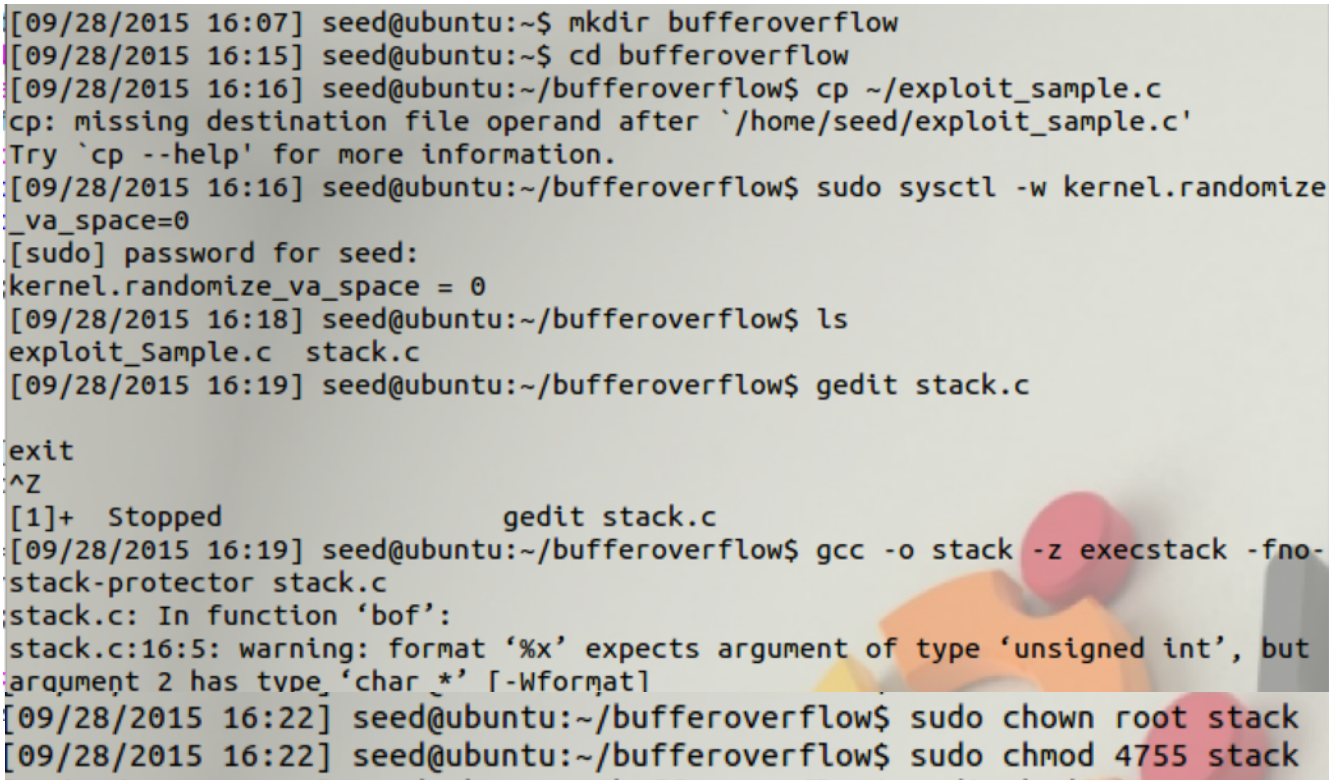
Task 2.2
Shellcode

Shellcode is used to launch the shell. It must be loaded into the memory so that one can force the vulnerable program to jump to it. For this code, we turn off the stack guard and make the stack non-executable via the execstack option and run the following code which was given. It must be compiled in assembly because the C code contains NULL characters:

```
[10/05/2015 17:12] seed@ubuntu:~/bufferoverflow$ gcc -fno-st
ack-protector call_shellcode.c
[10/05/2015 17:15] seed@ubuntu:~/bufferoverflow$ gcc -z ecec
stack -o call_shellcode call_shellcode.c
/usr/bin/ld: warning: -z ececstack ignored.
[10/05/2015 17:15] seed@ubuntu:~/bufferoverflow$ gcc -z exec
stack -o call_shellcode call_shellcode.c
```

Task 2.3
The Vulnerable Program

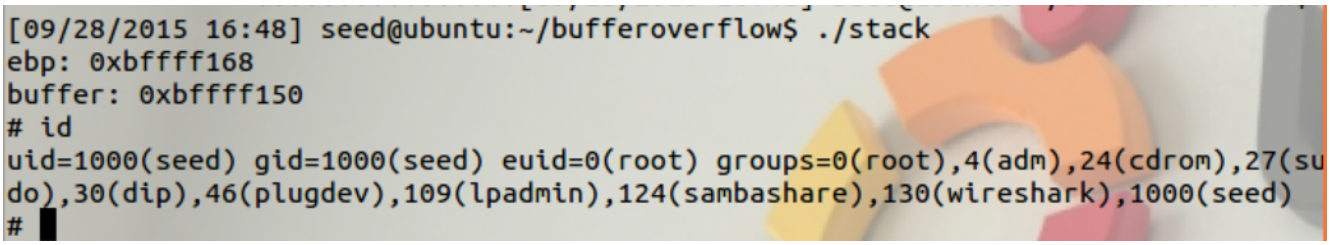
In this task, we attack the vulnerable program by overloading the buffer and creating a buffer overflow. We are exploiting its vulnerability. First we start by compiling the stack.c program and making it a Set-Root-UID program.



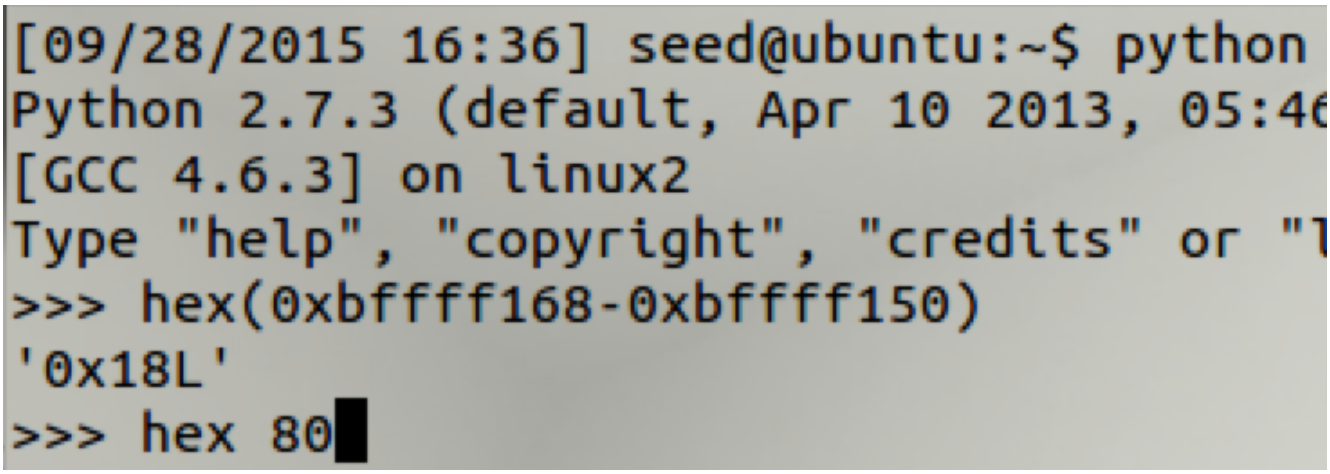
Because strcpy() in stack.c does not check boundaries, a buffer overflow will occur. Since this program is a Set-Root-UID program, if a normal user is capable of exploiting the vulnerability, than the normal user might be able to get a root shell—meaning that a lot of damage can be done! This program receives its input from a file names "badfile" which is under the users' control. The next step in this task is to create this "badfile" so that when the vulnerable program copies the contents into its buffer, a root shell can be spawned. At first, we used touch badfile to create a file which contained less than 24 characters. This allowed the program to run completely fine as if nothing was going to occur.

Task 2.4
Task 1. Exploiting the Vulnerability

We were given partially exploited code named exploit.c. The goal of this code is to construct the contents for "badfile". Because my exploit was implemented correctly, I was able to obtain a root shell and check the root id seen below (note buffer and bp address):



This additionally overflowed the buffer in stack.c, obtaining the results shown below. In order to overflow the buffer properly, edits must be made to the exploit_sample.c which contains the malicious shell code address. We need to find return address so we need to point to where the current function call is. This is done with the ebp, which points to previous function pointer. Because I am running a 32 bit machine (= 4 byte) this means that the next address is ebp +4, which will be start address of the return address (the previous function pointer). If one can obtain the ebp then we can find the other return address.. *****distance +4 = return address****. Therefore, distance= the distance between the ebp and the buffer (if its 12 its 24 bytes). My computers difference is hex 80, which was calculated using python on another terminal, seen below:

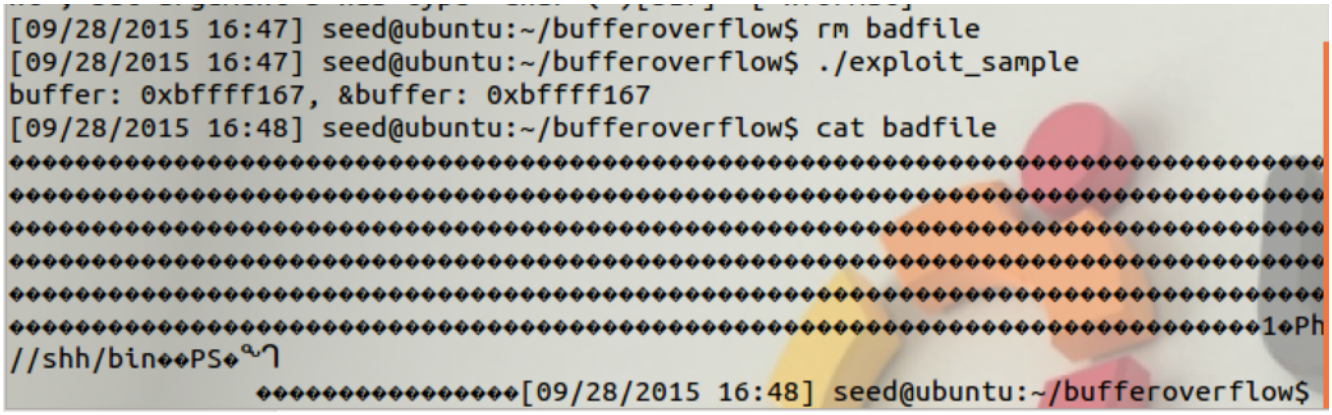


So we add buffer +0x18+0x04=RA= malicious code), which equals 0x1c. That means overall, the ebp+4 is the return address (RA). Ebp=168+4 in hex becomes 0xbffff16c=RA. RA + 0x150 (to get above the RA in order to overflow) = 0xbfffff2BC this is our malicious address. We then modify exploit_sample.c to the correct malicious code address (which was only able to

be calculated due to the fact that randomized memory was turned off). This modification is shown below in the C code:

```
    *((long *) (buffer + 0x1c)) = 0xbffff2bc; //
uffer + distance = malicious shell code start
ddress
```

When we re-run the code exploit_sample.c, the buffer is overflown and the file becomes gibberish:



Then, in order to allow our commands to behave properly and not as a normal user, we set the real user id to root by running a setuid(0); program for the shell:

```
2015 17:54] seed@ubuntu:~/bufferoverflow$ gcc set_root.c -o set_root
```

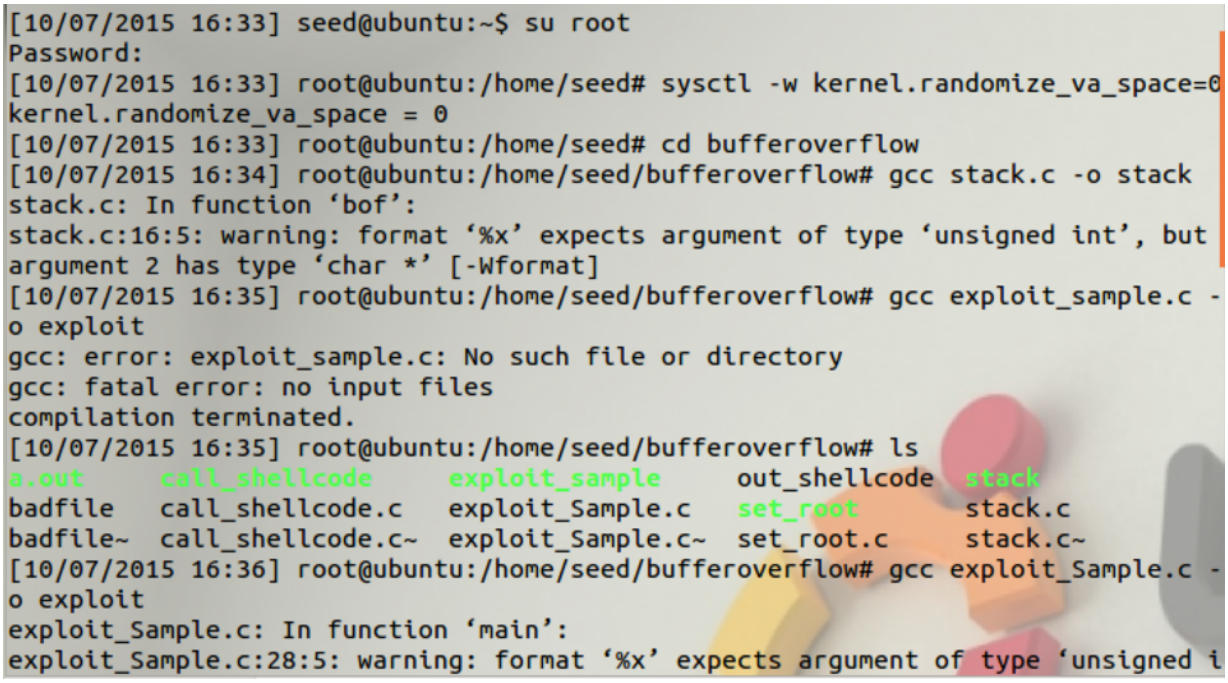
Task 2. Address Randomization

Once Ubuntu's randomization was turned on, we run the same attack developed in Task 1. I am still able to get a Shell, however the address for the malicious code no longer works. The address randomization makes my attacks more difficult because in order to create a malicious address I would now have to guess the address in the buffer rather than be able to calculate it. Once running my vulnerable code, I was able to get the root shell after a day:

The address randomization makes my attacks difficult because we can no longer pinpoint the address to point to to place the malicious code.

Task 3. Stack Guard

For this section, we turn off address randomization and turn on the the presence of Stack Guard. I recompiled the vulnerable program, stack.c, as well as the exploit_sample.c, and executed task 1 again. I received an error message of "stack smashing detected." It then terminated the entire program via abort and dumped the core. This happened because stack guard is now on and is protecting the return address on the stack from being altered.



```
exploit_Sample.c:28:5: warning: format '%x' expects argument of type 'unsigned i
nt', but argument 3 has type 'char (*)[517]' [-Wformat]
[10/07/2015 16:36] root@ubuntu:/home/seed/bufferoverflow# ./exploit
buffer: 0xbffff187, &buffer: 0xbffff187
[10/07/2015 16:36] root@ubuntu:/home/seed/bufferoverflow# ./stack
ebp: 0xbffff168
buffer: 0xbffff150
*** stack smashing detected ***: ./stack terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
./stack[0x8048581]
[0xbffff2bc]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 1582848    /home/seed/bufferoverflow/stack
08049000-0804a000 r--p 00000000 08:01 1582848    /home/seed/bufferoverflow/stack
0804a000-0804b000 rw-p 00001000 08:01 1582848    /home/seed/bufferoverflow/stack
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0b000-b7e0c000 r--p 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0c000-b7e0d000 rw-p 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
```

```
.1
b7e0b000-b7e0c000 r--p 0001b000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e0c000-b7e0d000 rw-p 0001c000 08:01 2360149    /lib/i386-linux-gnu/libgcc_s.so
.1
b7e1f000-b7e20000 rw-p 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc3000-b7fc5000 r--p 001a3000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc5000-b7fc6000 rw-p 001a5000 08:01 2360304    /lib/i386-linux-gnu/libc-2.15.s
o
b7fc6000-b7fc9000 rw-p 00000000 00:00 0
b7fd8000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 2364405    /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)
[10/07/2015 16:36] root@ubuntu:/home/seed/bufferoverflow#
```

Task 4. Non-executable Stack

For this task, we also had address randomization, and we recompiled our vulnerable program using the noexecstack option while repeating task 1. I was unable to get a shell, but instead received a segmentation fault (core dump).

```
[10/07/2015 16:36] root@ubuntu:/home/seed/bufferoverflow# gcc -o stack -fno-stack-protector -z noexecsta
ck stack.c
stack.c: In function ‘bof’:
stack.c:16:5: warning: format ‘%x’ expects argument of type ‘unsigned int’, but argument 2 has type ‘cha
r *’ [-Wformat]
[10/07/2015 16:45] root@ubuntu:/home/seed/bufferoverflow# gcc exploit_Sample.c -o exploit
exploit_Sample.c: In function ‘main’:
exploit_Sample.c:28:5: warning: format ‘%x’ expects argument of type ‘unsigned int’, but argument 2 has
type ‘char *’ [-Wformat]
exploit_Sample.c:28:5: warning: format ‘%x’ expects argument of type ‘unsigned int’, but argument 3 has
type ‘char (*)[517]’ [-Wformat]
[10/07/2015 16:45] root@ubuntu:/home/seed/bufferoverflow# ./exploit
buffer: 0xbffff187, &buffer: 0xbffff187
[10/07/2015 16:46] root@ubuntu:/home/seed/bufferoverflow# ./stack
ebp: 0xbffff178
buffer: 0xbffff160
Segmentation fault (core dumped)
[10/07/2015 16:46] root@ubuntu:/home/seed/bufferoverflow#
```

The non-executable stack only makes it impossible to run shelled on the stack, but it does not prevent the buffer-overflow. Therefore, this averts the original attack, yet does not make it impossible to still insert the malicious code. For example, it is still possible to attack with a buffer overflow using a return-to-libc attack.