Buffer Overflow Vulnerability Lab

Austin Hansen

1001530325

2.1 Turning Off Countermeasures

Text

Description automatically generated

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

Text

Description automatically generated

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.

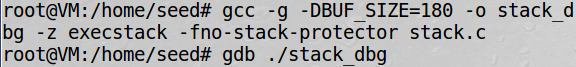
2.3 The Vulnerable Program

Text

Description automatically generated

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



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)

Text

Description automatically generated

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.

Text

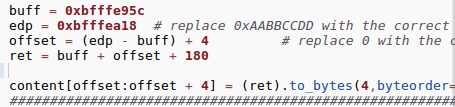
Description automatically generated

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?

Text

Description automatically generated

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.



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?

Text

Description automatically generated

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.**

Text, letter

Description automatically generated

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

Text, letter

Description automatically generated

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

2.5 (Task 3): Defeating dash’s Countermeasure

Text

Description automatically generated with medium confidence

Text

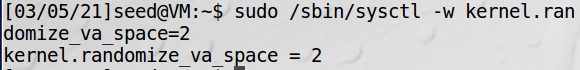
Description automatically generated

Text

Description automatically generated

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



Text

Description automatically generated

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.

Text

Description automatically generated

Compilation and changing ownership yield no errors.

Text

Description automatically generated

Text

Description automatically generated

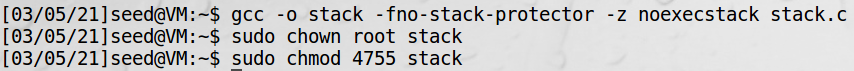
The address of our points of interest changed, no errors though. Updated exploit.py…

Text

Description automatically generated

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.



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.

Text

Description automatically generated

Our addresses changed and have been updated in the program.

Text

Description automatically generated

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.