Return-to-libc Attack Lab

Austin Hansen

1001530325

2.1 Turning Off Countermeasures:

Text, letter

Description automatically generated

For the first portion of the lab, test that each command that is used in the following parts works correctly and produces no errors, test in this case is a c program that returns nothing and is simply used to ensure that execstack and noexecstack work on the compiled program as intended. Bottom line links /bin/sh to /bin/zsh.

2.2 The Vulnerable Program

Text

Description automatically generated

We have now compiled the program in a vulnerable state, given ownership to the root user and set its ownership, it is time to begin our first attack.

2.3 Task 1: Finding out the addresses of libc functions.

Text

Description automatically generated

Using the instructions from the assignment, we can get the address of our system () call in our libc program, in this case **0xb7e42da0** to 0**xb7e369d0** from this we can figure out where we can jump to in the next task.

2.4 Task 2: Putting the shell string in memory:

A picture containing text

Description automatically generated

Using this program, we can get the address of our shell, which will be inserted as the “X” value in the next task. I called this program **get\_memory.c.**

A picture containing graphical user interface

Description automatically generated

After compiling the **get\_memory.c** using **gcc -o get\_memory get\_memory.c** (not picture here), we make an environmental variable **MYSHELL** for the shell variable, then use **get\_memory** to get the address of /bin/sh. Also, I had to swap to a different terminal, as this terminal began behaving strangely, but all data from the previous parts is the same. Our new address is **bffffe14.**

**(Strangely after running this a few times using different names for the program, I got slightly different results)**

2.5 Task 3: Exploiting the buffer-overflow vulnerability:

Text

Description automatically generated

After running the debugger and setting a breakpoint at bof, we look at the line with lea eax, [ebp-0x9e], 0x9e (158) in this case is where our start function needs to be placed at in our **exploit.c** buffer.

Text

Description automatically generated

something curious to note about our address given in Task 2 is that it does not match the same address in gdb, this is because the environmental variables differ in gdb from the regular shell. In fact we can see the environment variable at 0xbffffe20, but in a fragmented state.

Text, letter

Description automatically generated

After getting this information, I updated the exploit.c program to have 158 as the start of our attack string, after which we step up 4 bytes (size of a pointer), until reaching 166, where after 170 we have only the ASCII value of ‘Q’. The size of our file only needs to be bigger than the size of the buffer in the other program.

Text

Description automatically generated

It only half works and I have no idea why this is, I have retried all the steps above 20+ times, I get the same results, what this is doing is calling the shell, but not executing it in root (I have made sure that it is still owned by root), yielding only a local shell, all countermeasures are off, all steps are completed, I am just not sure why it is doing this.

Text

Description automatically generated

Removing exit from exploit.c results in a seg fault, this makes sense given the nature of this attack. So, yes exit () is necessary for the attack to succeed, as without it we essentially don’t have a complete program.

Text

Description automatically generated with medium confidence

It should return more than this, as I implied above, the addresses should change, but the environment variable will not change its address, it should try to execute a command, but fail due to having an incomplete or nonexistent command string.

2.6 Task 4: Turning on address randomization.

Text

Description automatically generated

Using get\_memory to test the addresses, we can see that they are quite different from before, this would also apply to the other addresses as well, regardless of gdb keeping addresses static, also of the 6 values present in exploit.c, all addresses would be unreliable to figure out or calculate.

Text

Description automatically generated

Intriguingly, we will always get 158, when we calculate for where in the attack buffer, we would need to place the address, meaning the indices of our attack array remain the same.

2.7 Task 5: Defeat Shell’s countermeasureText

Description automatically generated

To defeat shells countermeasures, I used a program from the book called **stack\_rop.c,** it is pictured here, it will be used for this task and the next task. Critically it uses fread () which is vulnerable to a buffer overflow attack. I pretty much used the same program for both task 5 and task 6, task 5 was just a little shorter. (I actually did task 6 first, and both the attack function and chain\_attack function are both from the book, at about page 127) Also, I had to swap terminals again.

Text

Description automatically generated

To begin, we address hunt, making note of what addresses we get, with this output we no know the address of /bin/sh, and our frame pointer for our foo function.



After disassembling foo in gdb, we write down the address next to leave, this is the start of our leave return function.

Text, letter

Description automatically generated

This portion was done for task 6 but is still relevant to task 5 (indeed I kept them in for the task 5 function), but we fill in all addresses needed for our program.

Text

Description automatically generated

Here are all our addresses, in the program for task 5 we invoke sprintf 4 times, padding with zeroes as we go, which precipitates into us setting the uid to zero, calling the shell and then exiting when necessary. (see below for sprintf portion)

Graphical user interface, text, application, chat or text message

Description automatically generated

(Note that for the Part 5 adaptation sprint\_arg2 does not exist)

Text

Description automatically generated

We have bypassed using 0s, it still gives the strange shell that does not list euid, but it looks a bit better than the one far above.

2.8 Task 6: Defeat Shell’s countermeasure without putting zeros in input.

Text

Description automatically generated

With familiar addresses, we invoke sprintf 4 times to set bits to 0 starting at sprintf\_addr + 12 + 5\*0x20 (not picture here), we then set that address till we get to 4 bytes above it, this allows us to then call setuid(0), but importantly we are not padding with zeroes as we are going, following up with a call to system(/bin/sh), bypassing the countermeasures in place. Important parts of program below:

Text

Description automatically generated

While I did slam you a bit with this large image of just code, I want to explain it a bit more. See, the reason that sprinf\_arg1 is calculated the way that it is because each function is always edp+8, which is always 12, the next portion simply indicates a call chain, in this case a call chain of five. Sprinf\_arg2 is our zero byte but it is set to the address of /bin/sh, for later use. The rest is a lot like task 5, mostly doing padding of 0x20.

Text

Description automatically generated

Here’s the result of running this version of the attack, we still get the weird shell, I might need to look at reinstalling the VM.