George, Levi

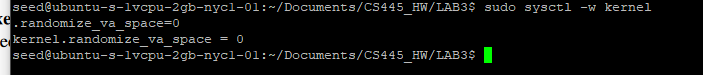
CS44500 – Computer Security

02/04/23

Lab 3

Env Setup.

Randomization Turned off



Removing dash



Task1. Invoking Shellcode

Shellcode can’t be written in regular code (High-level languages). It doesn’t provide the level of control that we get with shellcode and it includes numerous possible points where ‘\0’ is in the code, and if that gets load into strcpy it will not load the full code.

The Shellcode displayed essentially loads a new shell into execve. The only major differences between 64 bit and 32 bit versions of the code shown are the register names and how we call execve and the system call. When I ran them both, I got a shell based on the file owner’s permissions.

32 bit code

Text

Description automatically generated

64 bit code

Text

Description automatically generated

I didn’t notice any major differences, I was able to run both versions of the code and get access to a new shell. If the code were Set-UID I could get a root shell as seen below. (We see LOGNAME=root in my printenv command since I logged into the server through root then ran “su seed” to use the seed account.

Text

Description automatically generated

Task2. Understanding the Vulnerable Program

The L1 and L2 values have been set according to the lab instructions.

Text

Description automatically generated

The code we are given for this task has a vulnerability in it’s code, strcpy(), that we can use to buffer-overflow the program when it is running.

Running the makefile will produce the following output and files.

A screenshot of a computer

Description automatically generated with medium confidence

Task3. Launching Attacks on 32-bit programs



The empty badfile has been made

Ebp for the first stack program is

Text

Description automatically generated

Address of the buffer

Text

Description automatically generated



(difference in memory)

Shellcode inserted – I chose the 32 bit shellcode from Task 1 since we are running all the code as 32 bit.

Text

Description automatically generated

I set the start of the shellcode to be at the very end of our NOP flood. 517 being the end of the file – len(shellcode) will give use enough space to insert our payload at the end of payload.

A screenshot of a computer

Description automatically generated with medium confidence

I set the return address value to my $ebp value + 4 to get my return address value, I then appended my guess to that value.

I also set my offset to 158 + 4 to insert it into the Return Address, 158 being ebp, +4 putting it ontop of the RA register.

Text

Description automatically generated

My root access (thank you so much Prof. Chen, I literally could not have done this without you).

Text

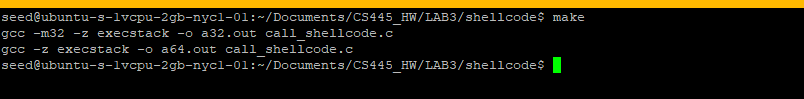
Description automatically generated

Task7. Defeating Dash

I return my shell from zsh to dash.



I make the shellcode executables.



I make the programs setuid

A screenshot of a computer

Description automatically generated with medium confidence

I run both the 64 bit and 32 bit code.

Text

Description automatically generated

Neither get me the golden root shell, but both are able to open the shell.

So, I add the setuid binary code to my shellcode.

Text

Description automatically generated

I remake the code

Text

Description automatically generated

Then I make the programs setuid, and run the code, both award me root access.

Graphical user interface, text

Description automatically generated

Task8. Defeating Address Randomization

I run the command to turn on ASLR.



I also ran commands to reset zsh as my shell.

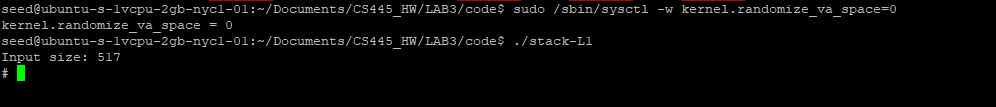
My first step is to check and make sure the exploit.py has been run and badfile is up to date. I then run stack-l1. Only after running it three times I realize that I already turned on ASLR.

I ran the bash shell script to brute force the code. However, it ran for almost an hour without returning root.

Text

Description automatically generated

Task9. Defeating other countermeasures



I begin by testing my Level 1 exploit.

Then I recompile my stack.c code with no stack guard

Text

Description automatically generated

Upon re-running my stack-L1-noguard code, I get the message the “stack smashing” was detected and my code is aborted before I can get root access.

A screenshot of a computer

Description automatically generated with medium confidence

Task 9b: Non-executable stack protection

I created a version of the code without executable stack, using the makefile and ran the code. I only got a seg fault. This can be reasonably assumed since we just keep running no op instructions until we reach illegal addresses.

Text

Description automatically generated

Task4 – Should I choose to accept.