Will Pond

Turning off the countermeasures

A screenshot of a computer code

Description automatically generated

Task1

Code for the call\_shellcode.c to be use later.

A screenshot of a computer program

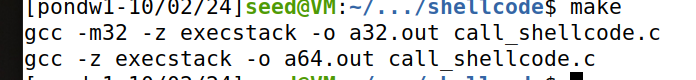
Description automatically generated

Code for the Makefile and in shellcode folder to be used later.

A screenshot of a computer

Description automatically generated

Ran the make command to produce a32.out and a64.out from the code above.



Gets the seed user from both executable file

A screen shot of a computer code

Description automatically generated

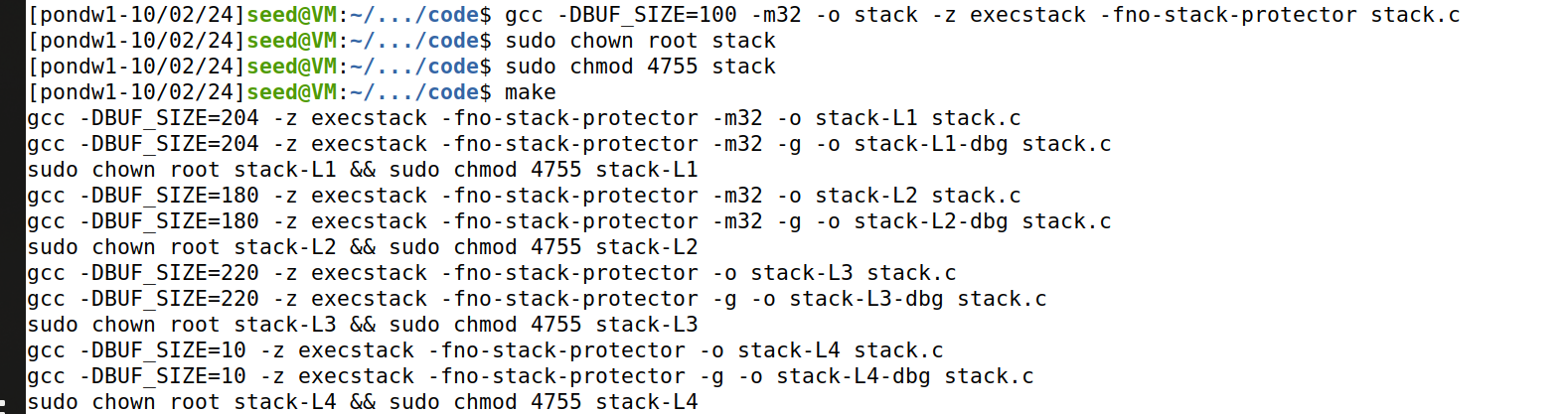
Task2

Changing the values of Makefile to your specific instructions for Last Names M-Z.

A screenshot of a computer

Description automatically generated

Make stack a root owned Set-UID program by changing the ownership and permission and then running make to execute the command in the Makefile to make the compilation



Task3

Code for stack.c to be use later.

A screenshot of a computer

Description automatically generated

Created the badfile to be use later then starting debugging and place a break point when it started.

A screenshot of a computer program

Description automatically generated

After the run command you get this following output the buffer overflow

A screen shot of a computer program

Description automatically generated

The Next Command get this following output and the string copy down below to know it is finished

A screenshot of a computer program

Description automatically generated

Gets the ebp value the and the buffers’ address value, then getting the $3 or offset bit value by subtracting $1-2. Finally exits the debug

A screen shot of a computer code

Description automatically generated

Code for exploit.py for Attack level-1 by changing the shellcode for the 32 bits, then changing the start to 400 to get the shellcode because we want to start at 400 and go to 517 next changing ret to the ebp value and adding 100 to address by guessing. finally using the above calculating for getting the difference of 212 and taking 212 +4 to get the offset to get the return address.

A screenshot of a computer code

Description automatically generated

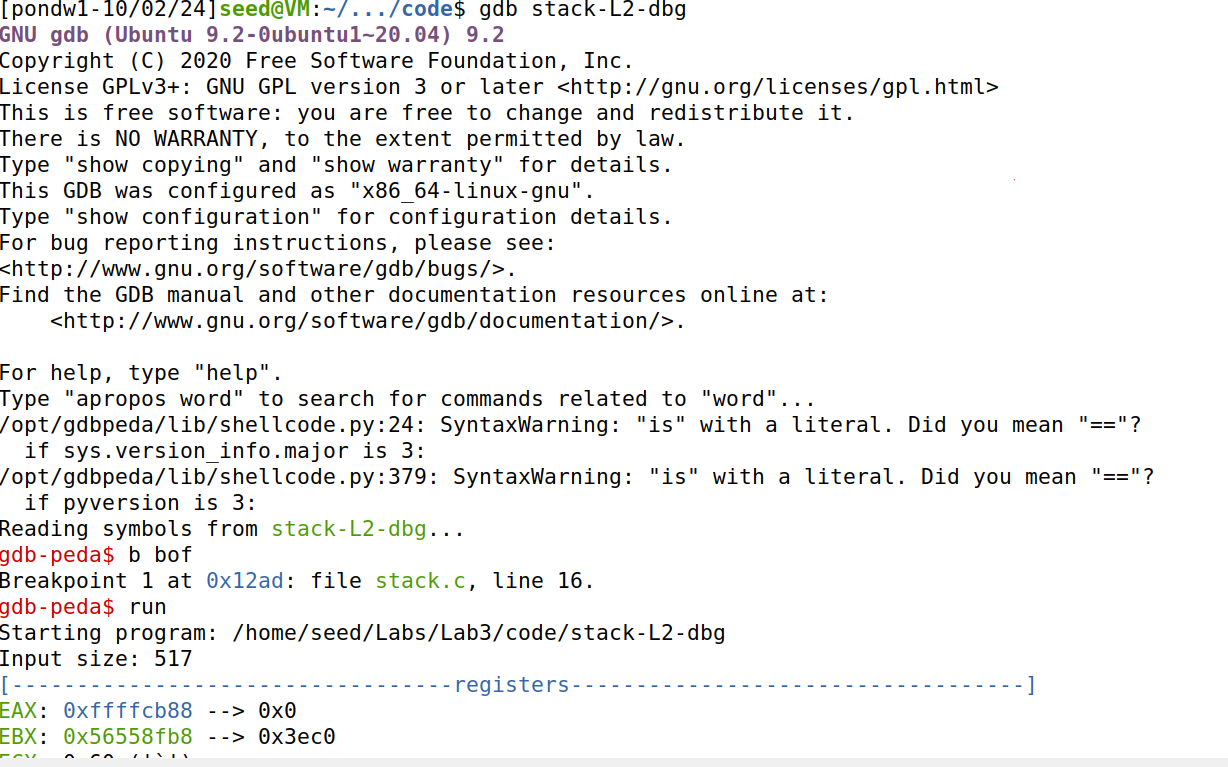
Running the exploit.py and then executing the stack-L1 to get root shell.

A screenshot of a computer

Description automatically generated

Task4

Started the debug on stack-L2-dbg after creating a new badfile and running the programing to get the buffer overflow.



The Next Command get this following output and the string copy down below to know it is finished



Then getting the buffer address and exit the debug

A screenshot of a computer code

Description automatically generated

Changing the exploit.py Attack level2 by adding the 32-bit code to the shellcode and changing the content variable by putting the shellcode at the end of the badfile, changing the ret variable by putting in the buffer address value from above and 400 to it to starting out when jumping. Finally, we are spraying the buffer with the return address by adding the for loop to loop through the offset and changing the contents

A screenshot of a computer

Description automatically generated

Running the exploit.py and executing the stack-L2 to get root shell

A screen shot of a computer

Description automatically generated

Task7

Disabling the setuid privileges to add a countermeasureA screenshot of a computer code

Description automatically generated

Adding the binary code for setuid to the shellcode by adding to the beginning of call\_shellcode.c

A screenshot of a computer code

Description automatically generated

Making the call\_shellcode.c and making it into root-owned binary to be use later

A close-up of a computer code

Description automatically generated

Executing a32.out and a64.out getting out normal privilege shell after making modifications to the program.

A screenshot of a computer program

Description automatically generated

Relaunching Level1 attack and making changes to the shellcode by adding the binary code for setuid.

A screenshot of a computer program

Description automatically generated

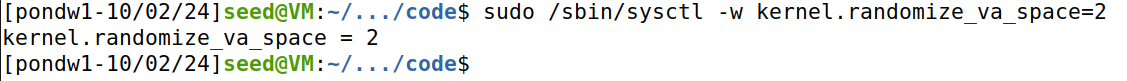
Then rerunning the attack and getting root shell while also verifying that the countermeasure is turned on by getting output

A screen shot of a computer code

Description automatically generated

Task8

The code for the brute-force method and making executable by chmod +x the file. Then turning on the address randomization



Code for brute-force.sh

A screenshot of a computer program

Description automatically generated

After running brute-force.sh the attack succeeds after 31 seconds, and it ran 13777 times. That ran faster that I thought maybe because I am using a local VM to speed up the process.

A screenshot of a computer code

Description automatically generated

Task9

Repeat the level-1 attack without the stackGuard to make sure I get the root shell

A black and white rectangular object with a black stripe

Description automatically generated

Turning on the stack Guard protection by removing the -fno-stack-protector flag in the gcc command and blocking the attacker

A screenshot of a computer

Description automatically generated

Turning on the non-executable stack protection by not adding the -z noexecstack flag in the compilation which lead to segment failure.

A screenshot of a computer

Description automatically generated

Same thing happened down here when we don’t add the -z noexecstack flag when running make command and we get Segmentation failure

A screenshot of a computer

Description automatically generated

Reflection

In this buffer overflow lab, I was able to run attacks from a buffer-overflow vulnerability program with many different protection schemes. I ran command that turns off countermeasures for the operating system to implemented. Then I was able to inject malicious code by the help of the shellcode using the 32-bit binary code or the 64-bit binary code from call\_shellcode.c. Then I learn the make command that can help execute the files in that directory to make more executable files depending on what file being executed. While doing this it was able to get the root shell but if it didn’t have execstack option the programing running would fail.

Next in my opinion the fun part of the lab launching the attacks by getting the debugging and running the program to get the buffer address. Then doing calculations to get the offset and guessing where to jump to get the return address. Then run the attack programs to test if the attack work or not. The hardest part lab was not knowing the Buffer Size is because if you have to guess more and the more guess you do the easier you get detected, meaning you have to start over again. Then defeating the countermeasure of dash by changing the UID == the effective UID when creating Set UID programs. To do this have to add setuid binary code to the call\_shellcod.c and using make setuid in that directory.

The other part I thought was going have hard was the Defeating Address Randomization when you have turn on address randomization and use the brute-force program to attack it. I thought my program will take a long time, but it only took 31 seconds to complete it. Finally experimenting with other Countermeasures was another part in the lab I had fun with by test out the StackGuard to see if your attack worked or not. Which executable nonexecutable in the complication does not work because buffer is mistaken it for data.