Will Pond

Change the N value to 252 based on your instructions for the Makefile.

A screenshot of a computer

Description automatically generated

Turning off the countermeasures.

A screenshot of a computer

Description automatically generated

Compiling the retlib.c program and changing it into a SetUID program.

A screenshot of a computer

Description automatically generated

**TASK 1**

Create a bad file and start to debug.

A screen shot of a computer code

Description automatically generated

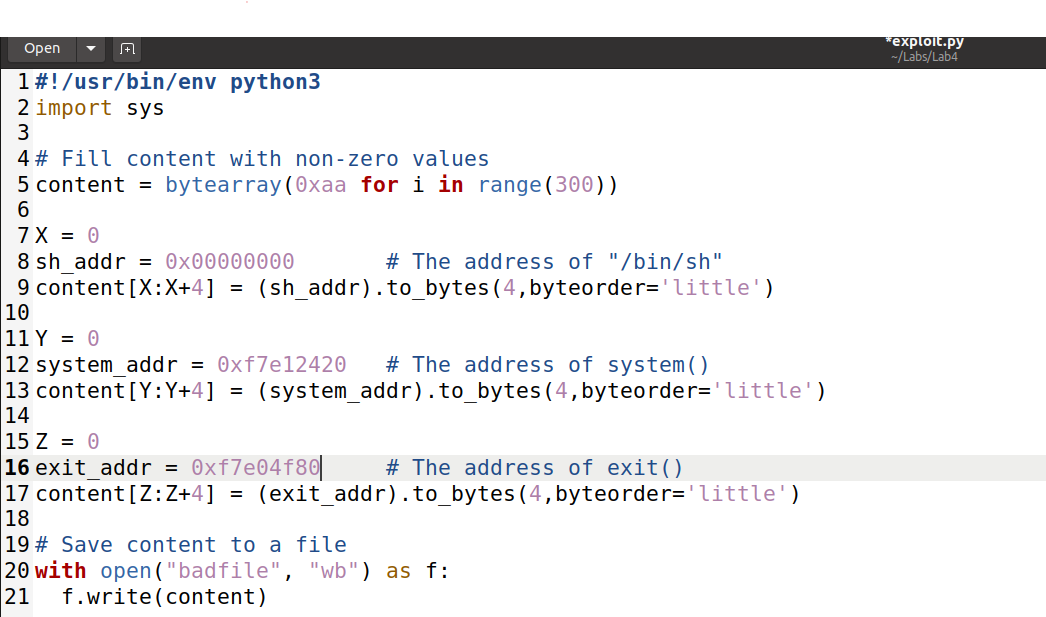
Running the debugging and getting the address of system and exit to use in exploit.py

A screen shot of a computer

Description automatically generated

**TASK 2**

Putting the address inside the exploit.py



**Creating an environment variable and creating a prtenv.c file to get the address**

**A screenshot of a computer

Description automatically generated**

**The code of prtenv.c**

**A black and white screen

Description automatically generated**

**Compiling the program in a 32-bit program and making it 6 characters long because retlib is 6 characters long. As a result, the address was being retrieved.**

**A screenshot of a computer

Description automatically generated**

**TASK3**

Putting the address of /bin/sh into exploit.py

**A screenshot of a computer program

Description automatically generated**

Running retlib to get the buffer address frame pointer address

**A screen shot of a computer

Description automatically generated**

Frame pointer – buffer address = buffer size

A black and green rectangular object with numbers and a black and white background

Description automatically generated

Editing the exploit.py file to add the calculated values to it. I figure out the decimal value is the ebp value and system() is the lowers in the hierarchy so it only get puts +4, then exit() function is above +8 and bin/sh get +12 because it is highest in stack.

A screenshot of a computer

Description automatically generated

Running the exploit.py file and to generate the badfile then running the retlib to get the root.

A screenshot of a computer

Description automatically generated

Removing the exit function from exploit.py

**A screenshot of a computer program

Description automatically generated**

Running the program again and the attack is successful but when it exits it cashes because it missing the exit function.

**A screen shot of a computer code

Description automatically generated**

Running the attack again after changing retlib to newretlib. What I have notice is that the privilege escalation failed which shows that the length of the program needs to be the same. If not it cannot find the file which will lead to Segmentation fault

A screenshot of a computer screen

Description automatically generated

**TASK4**

Enabling the counter measure

A black screen with white text

Description automatically generated

Running the debugger and getting address values of exit and execv

A screenshot of a computer program

Description automatically generated

Putting the address in the exploit\_Task4\_skeletion.py

A screenshot of a computer

Description automatically generated

**Creating an environment variable and creating a prtenv.c file to get the addresses**

A screenshot of a computer

Description automatically generated

**The code of prtenv.c**

A screen shot of a computer

Description automatically generated

**Getting the address “-p” and “/bin/bash”**

A screenshot of a computer

Description automatically generated

Putting the address in the exploit\_Task4\_skeletion.py file

A screenshot of a computer

Description automatically generated

Getting the address of the buffer and frame pointer

A screenshot of a computer program

Description automatically generated

Put the address into the exploit\_Task4\_skeletion.py

A screenshot of a computer

Description automatically generated

Frame pointer – buffer address

A screenshot of a calculator

Description automatically generated

Using the previous values from task3 and simply adding + 12 to the first excev argument and + 16 to second argument being they are higher rank on the stack and that is how I got the offset range. I tried and error on guessing the offset range, so I got to work on 430 for the offset.

A screenshot of a computer

Description automatically generated

Running the exploit\_Task4\_skeleton.py file and then the retlib program

A screenshot of a computer program

Description automatically generated

**LAB 4 REFLECTION**

In this lab I learned about another variant buffer-overflow attack called Return-to-libc which does not need executable stack or shellcode for that matter. It uses system() function in the lib library in the process memory. For beginning the lab, I had turn off the countermeasures, adding the require values and make a badfile. Then use the make command to make SETUID program of retlib. Which contains the vulnerability of the buffer overflow.

TASK1 required me to find the address of libc functions by debugging retlib and getting address of exit and system. TASK2 required me to put shell strings into memory by using environment variables and writing a program to put out the string address. Then using running retlib to get the buffer address and pointer frame. After that do calculate the hexadecimal using a website to get the size of the buffer.

In TASK3 put all the addresses in the correct spots and do calculations for the 3 values to get the root privilege. Then commenting out the exit function and rerunning the attack will still be successful but exiting it will make it crash and changing the name of the program will cause to fault because the length of the file. In TASK4 I had to turn on the counter measures of dash and repeat TASK1-TASK3 but with some changes. Instead of using the system() I had to use execv().

To do this I have to figure out the argv[] array place on the stack. In the first step I had to the two-string address from the environment variables. Next, I had to find the address of libc functions excev() and exit from the debugger. Finally finding the buffer addresses the running retlib. Putting all the addresses into the exploit\_Task4\_skeletion.py. I had to do some calculations and guessing to find the offset then apply to the rest of the values to complete the skeleton code. Next, running the program to get the root privilege to beat the dash countermeasure.