COEN225 Lab 4: Return-to-libc

Overview

The learning objective of this lab is for students to gain experience with basics of a code reuse attack. In lab 3, students exploited a buffer overflow vulnerability by overflowing a buffer with malicious shellcode, and then causing the vulnerable program to jump to the shellcode that is stored in the stack. To prevent these types of attacks, modern operating systems mark the stack as non-executable, which causes the program to exit if the CPU attempts to execute code in the stack. Code reuse attacks can bypass non-executable memory protections. Instead of executing the attacker's shellcode in the stack, 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 memory in a region where execution is allowed. In this lab, students will develop a return-to-libc exploit.

Lab Submission

Perform the tasks below and submit your responses to questions noted in the task submission sections. Also include any source code you used to generate exploits or other tests.

The lab work and submission must be completed individually but working with other students is allowed. The Camino message board is the preferred method for asking questions so that the instructor and students can respond and allow all students to learn from it.

Lab Tasks

Initial Setup

We will use the same virtual machine setup from previous labs. Modern Linux distributions have implemented several security mechanisms to make buffer overflow attacks more difficult. In order to focus on the attack technique itself, these defenses (ASLR, NX, and Stack Protector) will be disabled. See the Initial Setup section from Lab 3 for details on how to enable/disable each of these.

Task 1: Return-to-libc attack

Look at lab4.c, the vulnerable program for this lab. Compile the program with protections disabled.

```
$ gcc -z execstack -fno-stack-protector lab4.c -o lab4
```

This program reads a file called `lab4file` into memory which results in a buffer overflow. The objective is to create an exploit file that uses the return-into-libc technique to execute an arbitrary command and then gracefully exit.

An incomplete program called `gen_exploit.c` is prepared to create a basic file. Like the buffer overflow lab, your challenge is to determine the locations and values of key data to place in your exploit file to make the attack successful.

In this simple exploit generator, we need 3 more bits of information.

- The location of the `system()` function
- The location of a string to use as the argument for `system()`
- The location of the `exit()` function to exit gracefully

Note that the `exit()` function is not necessary for the attack itself to work; however, without this function, when `system()` returns after executing a command, the program will likely crash due to a corrupted stack. By using the `exit()` function as the return address of `system()`, the program will exit gracefully, making it less likely for the attack to be detected.

After you finish the program, compile and run it; this will generate `lab4file`. When `lab4` is executed, it will read your exploit file and, if created correctly, will execute your command.

At this point, you may begin constructing the exploit file, or use the guidance sections below if you need assistance. Don't forget to use the techniques used in earlier labs, such as finding out how to overwrite a return address with your exploit. Good luck!

Guidance 1: Finding addresses of libc functions
Common libraries such as libc are dynamically linked, unless the compiler was told to link statically. This means that the library code is not in the program itself, but they often come with the operating system. Because the library is not available at compile time, the program cannot resolve the addresses of libc functions. Instead, they are stored as symbols in the executable which are then linked at run-time by the OS that runs the program.

In earlier labs, you were able to use tools like objdump in order to find addresses of other functions. However, you won't be able to analyze just lab4.c to figure out the addresses of libc functions because they are unknown to it until the dynamic linker connects them. On Linux, you can use the ld command to invoke the dynamic linker.

However, to get addresses of functions, it may be easier to run the lab4 program and let the OS do the work for you, and then simply look up addresses using a debugger. To find out the address of libc functions, you can use the print (`p`) command to lookup the symbol you are looking for.

```
$ gdb lab4
...
(gdb) start
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e95e80 <system>
(gdb) b *0xb7e95e80
Breakpoint 1 at 0xb7e95e80
```

Guidance 2: Getting the address of a command string
One of the challenges in this lab is to get a string to be used as the command to pass to the `system()` function. The `system()` function takes any string and executes it as a command.

You may include the string in your exploit file, find the absolute address of it when it is in the stack, then modify your code again to use the address. Alternatively, we can find a string already in memory and repurpose it for our attack. One source of strings we could use is the environment variables. Use the `printenv` command to print out environment variables in the current context.

When a program is executed, it inherits all environment variables from the shell context that it was executed from and stores them in the process's memory space. In the Bash shell, the environment variable `SHELL` has a value of `/bin/bash` to indicate the path of the current shell. This could be used when an attacker wants to spawn a shell.

You can use the `/bin/bash` string to spawn another shell, but it may be difficult to determine if the attack was successful. Another string that can be repurposed is the `TERM` variable, which in most cases has the value `xterm` (a graphical terminal emulator). Launching `xterm` may be a more suitable demonstration of the attack.

Once you have figured out the command that you would like to use, you can search for its address in GDB using the `find` command.

```
(gdb) find $esp, +3000, "xterm"
0xbffe46c7
1 pattern found.
(gdb) x/s 0xbffe46c7
0xbffe46c7: "xterm"
(gdb) x/s 0xbffe46c2
0xbffe46c2: "TERM=xterm"
```

You can also add your own environment variables for an easy way to inject strings into memory for other commands.

```
$ export MYCMD=/bin/mycmd
```

Guidance 3: Building the exploit file for function chaining
Like earlier labs, this lab involves overwriting the return address to change the flow of code being executed. In a return-to-libc attack, we are calling functions that will eventually return control to the calling function. However, because the libc function isn't actually being properly called and instead being *returned to* the start of the function, the stack's layout is a bit different for chaining function calls (or rather, returns).

Start with the `system()` function address. This should overwrite the return address in the stack so that it is executed first. Because the function was invoked using a ret rather than a call, the stack frame does not get created how it would normally. You will have to think about where to place the command string so that the `system()` function can find it as its first argument.

Finally, determine where to place the `exit()` function's address so that it is executed when `system()` returns.

Task 1 Submission

Submit an explanation of how your exploit works, and include the source code. How did you find the addresses you used in your exploit? In what order did you put them in the exploit file?

Task 2: Testing Defenses Again

Re-run your attack 3 more times, once with each of the protections enabled.

Stack Protector enabled:

```
$ gcc -z execstack -fstack-protector lab4.c -o lab4
```

Non-executable stack enabled:

```
$ gcc -z noexecstack -fno-stack-protector lab4.c -o lab4
```

ASLR enabled (be sure to recompile with protections disabled):

```
# sysctl -w kernel.randomize_va_space=2
```

(if using GDB) `(gdb) set disable-randomization off`

Task 2 Submission

Report whether your attack was successful or not when Stack Protector, NX, or ASLR was enabled. Two of the attacks should have failed but one of them should have been successful even when the protection was enabled. Which protection failed to detect your attack? Why is this technique different from lab 3?

END OF LAB