Return-to-libc Attack Lab

1   Overview

The learning objective of this lab is for students to gain the 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

Readings and videos.   Detailed coverage of the return-to-libc attack can be found in the following:

   • Chapter 5 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.

   • Section 5 of the SEED Lecture at Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Note for instructors.   Instructors can customize this lab by choosing a value for the buffer size in the vulnerable program. See Section 2.3 for details.

2   Environment Setup

2.1   Note on x86 and x64 Architectures

The return-to-libc attack on the x64 machines (64-bit) is much more difficult than that on the x86 machines (32-bit). Although the SEED Ubuntu 20.04 VM is a 64-bit machine, we decide to keep using the 32-bit programs (x64 is compatible with x86, so 32-bit programs can still run on x64 machines). In the future, we may introduce a 64-bit version for this lab. Therefore, in this lab, when we compile programs using `gcc`, we always use the `-m32` flag, which means compiling the program into 32-bit binary.

2.2   Turning off countermeasures

You can execute the lab tasks using our pre-built Ubuntu virtual machines. Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

Address Space Randomization.   Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of heap and stack, making guessing the exact addresses difficult. Guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable this feature using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

The StackGuard Protection Scheme.   The `gcc` compiler implements a security mechanism called *StackGuard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks do not work. We can disable this protection during the compilation using the *-fno-stack-protector* option. For example, to compile a program `example.c` with StackGuard disabled, we can do the following:

```
$ gcc -m32 -fno-stack-protector example.c
```

Non-Executable Stack.   Ubuntu used to allow executable stacks, but this has now changed. The binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of `gcc`, and by default, stacks are set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -m32 -z execstack  -o test test.c

For non-executable stack:
$ gcc -m32 -z noexecstack  -o test test.c
```

   Because the objective of this lab is to show that the non-executable stack protection does not work, you should always compile your program using the `"-z noexecstack"` option in this lab.

Configuring /bin/sh.   In Ubuntu 20.04, the `/bin/sh` symbolic link points to the `/bin/dash` shell. The `dash` shell has a countermeasure that prevents itself from being executed in a `Set-UID` process. If `dash` is executed in a `Set-UID` process, it immediately changes the effective user ID to the process's real user ID, essentially dropping its privilege.
   Since our victim program is a `Set-UID` program, and our attack uses the `system()` function to run a command of our choice. This function does not run our command directly; it invokes `/bin/sh` to run our command. Therefore, the countermeasure in `/bin/dash` immediately drops the `Set-UID` privilege before executing our command, making our attack more difficult. To disable this protection, we link `/bin/sh` to another shell that does not have such a countermeasure. We have installed a shell program called `zsh` in our Ubuntu 16.04 VM. We use the following commands to link `/bin/sh` to `zsh`:

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

It should be noted that the countermeasure implemented in dash can be circumvented. We will do that in a later task.

2.3 The Vulnerable Program

Listing 1: The vulnerable program (retlib.c)

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);    ← buffer overflow!

    return 1;
}

int main(int argc, char **argv)
{
   char input[1000];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   int length = fread(input, sizeof(char), 1000, badfile);
   printf("Address of input[] inside main():  0x%x\n", (unsigned int) input);
   printf("Input size: %d\n", length);

   bof(input);

   printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
   return 1;
}

// This function will be used in the optional task
void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
```

```
}
```

The above program has a buffer overflow vulnerability. It first reads an input up to `1000` bytes from a file called `badfile`. It then passes the input data to the `bof()` function, which copies the input to its internal buffer using `strcpy()`. However, the internal buffer's size is less than `1000`, so here is potential buffer-overflow vulnerability.

This program is a root-owned `Set-UID` program, so if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called `badfile`, which is provided by users. Therefore, we can construct the file in a way such that when the vulnerable program copies the file contents into its buffer, a root shell can be spawned.

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 (for turning off the StackGuard protection) and the `"-z noexecstack"` option (for turning on the non-executable stack protection). 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`.

```
// Note: N should be replaced by the value set by the instructor
$ gcc -m32 -DBUF_SIZE=N -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

For instructors. To prevent students from using the solutions from the past (or from those posted on the Internet), instructors can change the value for BUF_SIZE by requiring students to compile the code using a different BUF_SIZE value. Without the `-DBUF_SIZE` option, BUF_SIZE is set to the default value `12` (defined in the program). When this value changes, the layout of the stack will change, and the solution will be different. Students should ask their instructors for the value of `N`. The value of `N` can be set in the provided `Makefile` and `N` can be from 10 to 800.

3 Lab Tasks

3.1 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. We use the `p` command (or `print`) to print out the address of the `system()` and `exit()` functions (we will need `exit()` later on).

```
$ touch badfile
$ gdb -q retlib     ← Use "Quiet" mode
Reading symbols from ./retlib...
(No debugging symbols found in ./retlib)
gdb-peda$ break main
```

```
Breakpoint 1 at 0x1327
gdb-peda$ run
......
Breakpoint 1, 0x56556327 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ 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.

Running `gdb` in batch mode.   If you prefer to run `gdb` in a batch mode, you can put the `gdb` commands in a file, and then ask `gdb` to execute the commands from this file:

```
$ cat gdb_command.txt
break main
run
p system
p exit
quit
$ gdb -q -batch -x gdb_command.txt ./retlib
...
Breakpoint 1, 0x56556327 in main ()
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

3.2   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
MYSHELL=/bin/sh
```

   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:

```
void main(){
   char* shell = getenv("MYSHELL");
   if (shell)
      printf("%x\n", (unsigned int)shell);
}
```

Compile the code above into a binary called `prtenv`. If the address randomization is turned off, you will find out that the same address is printed out. When you run the vulnerable program `retlib` inside the same terminal, the address of the environment variable will be the same (see the special note below). You can verify that by putting the code above inside `retlib.c`. However, 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`.

Note. You should use the `-m32` flag when compiling the above program, so the binary code `prtenv` will be for 32-bit machines, instead of for 64-bit ones. The vulnerable program `retlib` is a 32-bit binary, so if `prtenv` is 64-bit, the address of the environment variable will be different.

3.3   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 following, with the essential parts left for you to fill out.

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

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

X = 0
sh_addr = 0x00000000       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0x00000000   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0x00000000     # 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)
```

You need to figure out the three addresses and the values for X, Y, and Z. If your values are incorrect, your attack might not work. In your report, you need to describe how you decide the values for X, Y and Z. Either show us your reasoning or, if you use a trial-and-error approach, show your trials.

A note regarding gdb. If you use gdb to figure out the values for X, Y, and Z, it should be noted that the gdb behavior in Ubuntu 20.04 is slightly different from that in Ubuntu 16.04. In particular, after we set a

break point at function `bof`, when `gdb` stops inside the `bof()` function, it stops before the `ebp` register is set to point to the current stack frame, so if we print out the value of `ebp` here, we will get the caller's `ebp` value, not `bof`'s `ebp`. We need to type `next` to execute a few instructions and stop after the `ebp` register is modified to point to the stack frame of the `bof()` function. The SEED book (2nd edition) is based on Ubuntu 16.04, so it does not have this `next` step.

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.

Attack variation 2: After your attack is successful, 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 `badfile`). Will your attack succeed or not? If it does not succeed, explain why.

3.4 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. 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 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 describe how you construct your input.

4 Guidelines: Understanding the Function Call Mechanism

4.1 Understanding the stack layout

To know how to conduct Return-to-libc attacks, we need to understand how stacks work. We use a small C program to understand the effects of a function invocation on the stack. More detailed explanation can be found in the SEED book and SEED lecture.

```
/* foobar.c */
#include<stdio.h>
void foo(int x)
{
  printf("Hello world: %d\n", x);
}

int main()
{
  foo(1);
  return 0;
}
```

We can use `"gcc -m32 -S foobar.c"` to compile this program to the assembly code. The resulting file `foobar.s` will look like the following:

```
    ......
 8 foo:
 9         pushl   %ebp
10         movl    %esp, %ebp
11         subl    $8, %esp
12         movl    8(%ebp), %eax
13         movl    %eax, 4(%esp)
14         movl    $.LC0, (%esp)  : string "Hello world: %d\n"
15         call    printf
16         leave
17         ret
    ......
21 main:
22         leal    4(%esp), %ecx
23         andl    $-16, %esp
24         pushl   -4(%ecx)
25         pushl   %ebp
26         movl    %esp, %ebp
27         pushl   %ecx
28         subl    $4, %esp
```

```
29          movl     $1, (%esp)
30          call     foo
31          movl     $0, %eax
32          addl     $4, %esp
33          popl     %ecx
34          popl     %ebp
35          leal     -4(%ecx), %esp
36          ret
```

4.2 Calling and entering `foo()`

Let us concentrate on the stack while calling `foo()`. We can ignore the stack before that. Please note that line numbers instead of instruction addresses are used in this explanation.
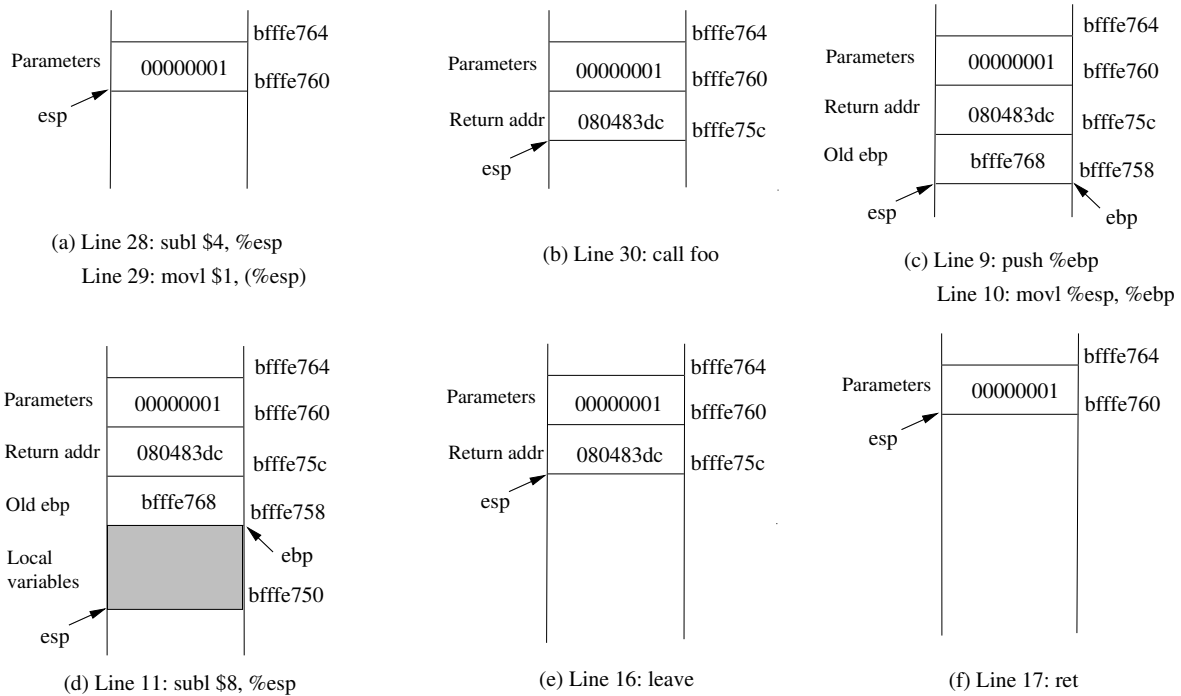

Figure 1: Entering and Leaving `foo()`

- **Line 28-29::** These two statements push the value 1, i.e. the argument to the `foo()`, into the stack. This operation increments `%esp` by four. The stack after these two statements is depicted in Figure 1(a).

- **Line 30: `call foo`:** The statement pushes the address of the next instruction that immediately follows the `call` statement into the stack (i.e the return address), and then jumps to the code of `foo()`. The current stack is depicted in Figure 1(b).

- **Line 9-10:** The first line of the function `foo()` pushes `%ebp` into the stack, to save the previous frame pointer. The second line lets `%ebp` point to the current frame. The current stack is depicted in Figure 1(c).

- **Line 11: subl $8, %esp**: The stack pointer is modified to allocate space (8 bytes) for local variables and the two arguments passed to printf. Since there is no local variable in function foo, the 8 bytes are for arguments only. See Figure 1(d).

4.3 Leaving foo()

Now the control has passed to the function foo(). Let us see what happens to the stack when the function returns.

- **Line 16: leave**: This instruction implicitly performs two instructions (it was a macro in earlier x86 releases, but was made into an instruction later):

  ```
  mov  %ebp, %esp
  pop  %ebp
  ```

  The first statement releases the stack space allocated for the function; the second statement recovers the previous frame pointer. The current stack is depicted in Figure 1(e).

- **Line 17: ret**: This instruction simply pops the return address out of the stack, and then jump to the return address. The current stack is depicted in Figure 1(f).

- **Line 32: addl $4, %esp**: Further restore the stack by releasing more memories allocated for foo. As you can see that the stack is now in exactly the same state as it was before entering the function foo (i.e., before line 28).

5 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.