Return-to-libc Attack Lab

Preparation

- This lab should be finished on **Ubuntu 20.04 VM**.
- **Your username format of your VM should be :** `<name>+<student_ID>(such as 'LiuShuo201800131111')`

Report

- 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.
- The report should be in English.
- The name format of your report should be `<lab3>+<name>+<student ID>.pdf(such as lab1_LiuShuo_201800131111.pdf)`
- You need submit the report before 2023-4-14-24:00 (upload it to SDU icloud: https://icloud.qd.sdu.edu.cn:7777/link/F4ABCDCBBB06077EDD166DA8A9F7360B).

Environment Setup

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

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.

The Vulnerable Program

```
//retlib.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);

    return 1;
}

void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}

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;
}
```

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.

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>
```

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.

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.

```
#!/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.

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.

Task 5 (Optional): Return-Oriented Programming

There are many ways to solve the problem in Task 4. Another way is to invoke setuid(0) before invoking system(). The setuid(0) call sets both real user ID and effective user ID to 0, turning the process into a non-Set-UID one (it still has the root privilege). This approach requires us to chain two functions together. The approach was generalized to chaining multiple functions together,

and was further generalized to chain multiple pieces of code together. This led to the Return-Oriented Programming (ROP).

Using ROP to solve the problem in Task 4 is quite sophisticated, and it is beyond the scope of this lab. However, we do want to give students a taste of ROP, asking them to work on a special case of ROP. In the retlib.c program, there is a function called foo(), which is never called in the program. That function is intended for this task. Your job is to exploit the buffer-overflow problem in the program, so when the program returns from the bof() function, it invokes foo() 10 times, before giving you the root shell. In your lab report, you need to describe how your input is constructed. Here is what the results will look like.

```
$ ./retlib
...
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
bash-5.0# →Got root shell!
```

Guidelines. Let's review what we did in Task 3. We constructed the data on the stack, such that when the program returns from bof(), it jumps to the system() function, and when system() returns,the program jumps to the exit() function. We will use a similar strategy here. Instead of jumping to system() and exit(), we will construct the data on the stack, such that when the program returns from bof, it returns to foo; when foo returns, it returns to another foo. This is repeated for 10 times. When the 10th foo returns, it returns to the execv() function to give us the root shell.

Further readings. What we did in this task is just a special case of ROP. You may have noticed that the foo() function does not take any argument. If it does, invoking it 10 times will become signficantly more complicated. A generic ROP technique allows you to invoke any number of functions in a sequence, allowing each function to have multiple arguments. The SEED book (2nd edition) provides detailed instructions on how to use the generic ROP technique to solve the problem in Task 4. It involves calling sprintf() four times, followed by an invocation of setuid(0), before invoking system("/bin/sh") to give us the root shell. The method is quite complicated and takes 15 pages to explain in the SEED book.