Return-to-libc Attack Lab

CMSC 426 - Computer Security

February 12, 2021

1 Lab Overview

The objective of this lab is to gain first-hand experience with an interesting variant of buffer-overflow attack; this attack can bypass an existing protection scheme implemented in major Linux operating systems. A common way to exploit a buffer-overflow vulnerability is to overflow the buffer with malicious shellcode and then cause the vulnerable program to jump to the shellcode on the stack. To prevent this type of attack, some operating systems allow programs to specify that the stack be "non-executable," thereby, preventing the shellcode from running.

Unfortunately, this 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 and, in fact, does not use shellcode at all. 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, you will be given given a program with a buffer-overflow vulnerability; your task is to develop a `return-to-libc` attack to exploit the vulnerability and gain root privileges. In addition to the return-to-libc attack, you will be introduced to protection schemes that have been implemented in `Ubuntu` to counter buffer-overflow attacks.

2 Lab Tasks

2.1 Initial Setup

To complete the lab, you will need to run the `SEEDUbuntu20.04` virtual machine in Virtual Box or a cloud provider (e.g. Amazon AWS or Microsoft Azure). The VM and set-up instructions are available from the SEED website (`https://seedsecuritylabs.org/labsetup.html`).

`Ubuntu` and other Linux distributions have implemented several security mechanisms to defend against buffer-overflow attacks. As part of the set-up for the lab, you will need to *disable* two specific protections and *enable* one.

Note on 32-bit and 64-bit Architectures. Although the SEEDUbuntu20.04 VM is a 64-bit machine, the return-to-libc lab still uses 32-bit programs. When compiling lab programs using GCC, the `-m32` flag must be used to produce a 32-bit binary.

Address Space Layout Randomization. `Ubuntu` and several other Linux-based systems use address space layout randomization (ASLR) to randomize the starting addresses of the heap and stack. This makes

guessing addresses of data in memory difficult, but guessing addresses is an important component of many buffer-overflow attacks. For this lab to succeed, you need to disable ASLR in the kernel:

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

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called *StackGuard* to prevent modification of key information on the stack. StackGuard can be disabled during compilation using the *-fno-stack-protector* switch. For example, the following command would compile the program `example.c` with StackGuard disabled, producing a 32-bit binary:

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

Non-Executable Stack. In `Ubuntu`, the binary images of programs and shared libraries must declare whether they require executable stacks or not by setting or clearing a flag in the program header. The kernel or dynamic linker uses this flag to determine whether to make the stack of the running program executable or non-executable. Recent versions of GCC set the stack to be non-executable by default, but this may be overridden during compilation using the *-zexecstack* switch:

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

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

Since the objective of this lab is to show that the non-executable stack protection can be bypassed using a return-to-libc attack, you should always compile the vulnerable program using the `-znoexecstack` option.

Configuring `/bin/sh`**.** in `Ubuntu` 20.04, `/bin/sh` is a symbolic link to the `/bin/dash` shell. Unfortunately, the `dash` shell in `Ubuntu` 20.04 includes a countermeasure that blocks the shell from executing as a set-uid process, which will prevent your return-to-libc attack from succeeding. For the attack to work, it is necessary to link `/bin/sh` to another shell program that does not include the set-uid countermeasure:

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

Techniques exist to bypass the `dash` countermeasure, but they are outside the scope of this lab.

2.2 The Vulnerable Program

```
#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(): %p\n", (void *) buffer);
    printf("Frame Pointer value inside bof(): %p\n", (void *) 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(): %p\n", (void *) input);
   printf("Input size: %d\n", length);
   bof(input);

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

You should compile the program using the provided Makefile, as it includes all the required compiler and linker flags and sets the `BUF_SIZE` variable to the correct value (128).

The program contains a buffer overflow vulnerability. It reads 1000 bytes from a data file called `badfile` and saves the data into a buffer of size `BUF_SIZE`, which is less than 1000 bytes, causing the overflow. The function `fread()` does not check array boundaries. Since this is a set-uid root program, a normal user can exploit the vulnerability to obtatin a root shell. Note that the contents of `badfile` are under the user's control. Our goal is to construct `badfile` so that when the vulnerable program copies the contents into its buffer, important control information on the stack is overwritten, and when the vulnerable function returns, a root shell is opened.

2.3 Task 1: Exploiting the Vulnerability

Use the following Python framework to create the file `badfile`.

Python Framework

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

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

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

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

Z = 0
exit_addr = 0xaaaaaaaa     # 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)
```

To complete the exploitation program, you will need to determine the addresses of two library functions — `system()` and `exit()` — and the address of the string "/bin/sh" in memory. In addition, you will need to determine where in the buffer to store these addresses. If either the addresses or their locations are incorrect, the attack will not work.

Once you have completed the exploitation program, run the program to generate `badfile`. Run the vulnerable program `retlib`. If your exploit is implemented correctly, when the function `bof()` finishes, it will return to the `system()` library function and execute `system("/bin/sh")`. If the vulnerable program is running with root privileges, the result will be a shell with root privileges.

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

It should be noted that the `exit()` function is not really necessary for this attack, but without this function, when `system()` returns, the program might crash; in a real attack, crashing the program is undesirable as it may lead an administrator to suspect an attack.

Requirements.

- Provide a screenshot demonstrating the successful execution of your attack.

- Describe the process you used to determine the values for `X`, `Y` and `Z`, including an explanation of the stack layout. Describe your reasoning *in detail*, or if you used a trial-and-error approach, describe your method and show your trials.
- After your attack is successful, change the file name of `retlib` to a different name, making sure that the lengths 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? If it does not succeed, explain why. Include a screenshot of your results.

2.4 Task 2: Address Space Layout Randomization

For this task, turn on the `Ubuntu`'s ASLR protection. You can use the following command to turn-on ASLR:

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

Now run the attack developed in Task 1 with ASLR enabled.

Requirements.

- Provide one or more screenshots demonstrating your results.
- Were you able to get a shell? If not, describe *in detail* the mechanisms preventing the attack from succeeding. You will need to investigate how ASLR is implemented in Linux to fully answer the question.

2.5 Task 3: Stack Guard Protection

For this task, turn *off* ALSR and turn *on* GCC's StackGuard protection. To turn on StackGuard, modify the Makefile, replacing `-fno-stack-protector` with `-fstack-protector`, and rebuild the program:

```
$ make clean
$ make
```

Now run the attack developed in Task 1 with ASLR disabled and StackGuard enabled.

Requirements. Describe and explain your observations. Include a screenshot demonstrating your results.

- Provide one or more screenshots demonstrating your results.
- Were you able to get a shell? If not, describe *in detail* the mechanisms preventing the attack from succeeding. You will need to investigate how stack protection is implemented in GCC to fully answer the question.

3 Suggestions and Hints

3.1 Find the addresses of libc functions

To find the address of any libc function, use the `print` command in `gdb`. Here `a.out` is an arbitrary executable:

```
$ gdb a.out

  gdb-peda$ b main
  gdb-peda$ run
  gdb-peda$ p system
   $1 = {<text variable, no debug info>} 0xf7e12420 <system>
  gdb-peda$ p exit
   $2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

From the `gdb` output, we can see that in this case the address of the `system()` function is `0xf7e12420`, and the address for the `exit()` function is `0xf7e04f80`. The addresses on your system will probably be different.

3.2 Putting the shell string in memory

One of the challenge in this lab is to put the string `"/bin/sh"` into the memory and get its address. This can be achieved using environment variables. When a program is run, it inherits all the exported environment variables from the shell that executes it. We can introduce a new shell variable, say `MYSHELL`, with value `"/bin/sh"`:

```
$ export MYSHELL=/bin/sh
```

We will use the address of this variable as an argument to the `system()` function. The location of this variable in memory can be found using the program `find_myshell.c`:

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

If ASLR is turned off, you will find that you can run `find_myshell` multiple times, and the same address will be printed each time. However, when you run the vulnerable program `retlib`, the address of the environment variable might not be exactly the same as that which you found by running `find_myshell` since the address is affected by the value of other environment variables: even the number of characters in an executable's file name can affect the location of an environment variable. However, the address of the shell in `retlib` should be quite close to what you find using `find_myshell`; therefore, you might need to try a range of addresses to get the attack to succeed.

3.3 Understanding the stack

To complete the `return-to-libc` attack, it is essential to understand how the stack works. We use a small C program to study the affect of a function call on the stack.

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

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

We can use the command `gcc -S foobar.c` to compile this program to assembly code. The resulting file `foobar.s` will look something 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
```

3.3.1 Calling and entering `foo()`

We can focus on what happens to the stack when `foo()` is called, ignoring previous stack frames. Please note that we refer to lines of assembly code using line numbers rather than instruction addresses in this

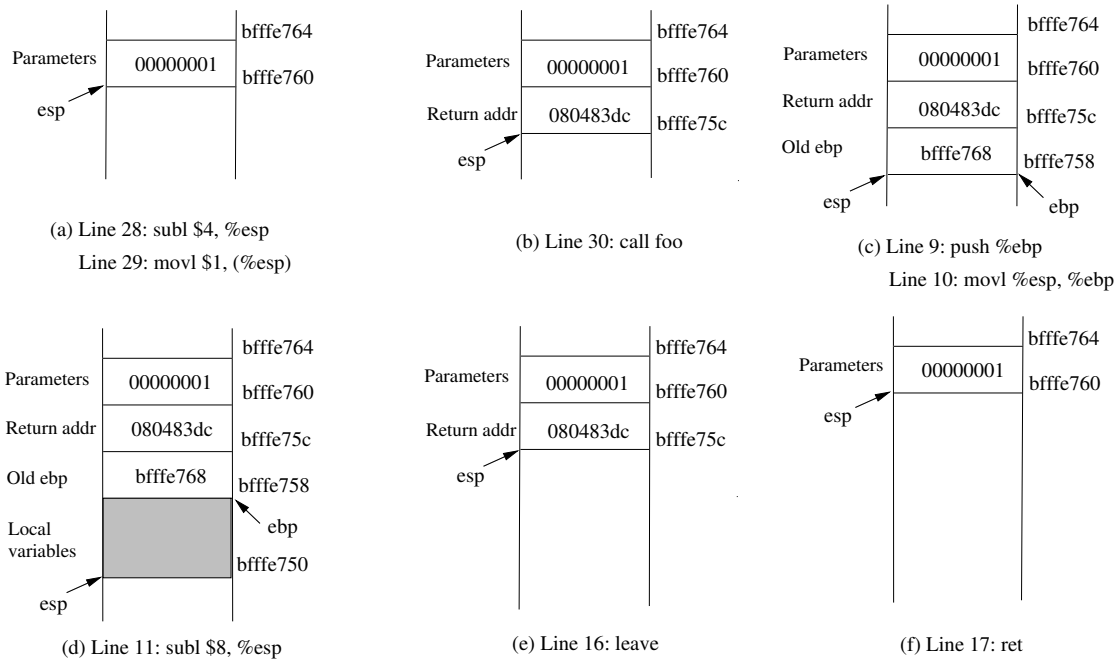

Figure 1: Entering and Leaving `foo()`

explanation.

- **Line 28-29:**: These two statements push the value 1, i.e. the argument to the `foo()`, ont the stack. Line 28 decrements `%esp` by four to make room for the integer value, and line 29 writes the integer literal 1 to the newly created location on the stack. The stack after these two statements is depicted in Figure 1(a).
- **Line 30: `call foo`**: The statement pushes the return address, the address of the instruction that immediately follows the `call` statement, onto the stack. It then then jumps to the code of `foo()`. The stack after execution of line 30 is depicted in Figure 1(b).
- **Line 9-10**: The first line of the function `foo()` pushes `%ebp` onto the stack, saving the previous frame pointer. The second line sets `%ebp` point to the current frame. The updated stack is depicted in Figure 1(c).
- **Line 11**: 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).

3.3.2 Leaving `foo()`

Next we consider how the stack changes when `foo()` returns.

- **Line 16**: 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 stack is depicted in Figure 1(e).

- **Line 17**: This instruction pops the return address from the stack and then jumps to the return address. The stack after this instructions executes is depicted in Figure 1(f).
- **Line 32**: Further restores the stack by releasing memory used to store arguments for `foo`. At this point, the stack is in exactly the same state as it was before entering the function `foo`.

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