

Return-to-libc Attack Lab

Copyright © 2006 Wenliang Du, Syracuse University.

The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

1 Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on an interesting attack on buffer-overflow vulnerability; this attack can bypass an existing protection scheme currently implemented in 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 this kind of attacks, some operating systems, such as Fedora Linux, 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 another type of attacks, 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 Fedora to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

2 Lab Tasks

2.1 Initial setup

We will conduct the attack on Fedora Linux¹. There are several protection mechanisms in Fedora that make the attacks much more difficult. Fedora uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult. Since guessing the right addresses is one of the critical steps of buffer-overflow attacks, we disable the randomization feature in this lab. The following `sysctl` command disables the feature.

```
$ su root
Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=0
```

Moreover, to further protect against buffer overflow attacks and other attacks that use shell programs, many shell programs automatically drop their privileges when invoked. Therefore, even if you can “fool” a privileged `Set-UID` program to invoke a shell, you might not be able to retain the privileges within the

¹We have tested this lab in Fedora 4, 5, and 6. It should also work for the most recent Fedora versions.

shell. This protection scheme is implemented in `/bin/bash`. In Fedora, `/bin/sh` is actually a symbolic link to `/bin/bash`. To see the life before such protection scheme was implemented, we use another shell program (the `zsh`), instead of `/bin/bash`. The following instructions describe how to set up the `zsh` program.

```
$ su
Password: (enter root password)
# wget ftp://rpmfind.net/linux/fedora/(continue on the next line)
    core/4/i386/os/Fedora/RPMS/zsh-4.2.1-2.i386.rpm
# rpm -ivh zsh-4.2.1-2.i386.rpm
# cd /bin
# rm sh
# ln -s zsh sh
```

You can also use `yum` to install the `zsh` package:

```
# yum install zsh
```

2.2 The Vulnerable Program

```
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);

    return 1;
}

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

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

    printf("Returned Properly\n");

    fclose(badfile);
}
```

```
    return 1;
}
```

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:

```
$ su root
Password (enter root password)
# gcc -o retlib retlib.c
# chmod 4755 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: Exploiting the Vulnerability

Create the **badfile**. You may use the following framework to create one.

```
/* exploit_1.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);
}
```

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.

It should be noted that the `exit()` function is not very necessary for this attack; however, without this function, when `system()` returns, the program might crash, causing suspicions.

```
$ gcc -o exploit_1 exploit_1.c
$ ./exploit_1          // create the badfile
$ ./retlib            // launch the attack by running the vulnerable program
# <---- You've got a root shell!
```

2.4 Task 2: Protection in /bin/bash

Now, we let `/bin/sh` point back to `/bin/bash`, and run the same attack developed in the previous task. Can you get a shell? Is the shell the root shell? What has happened? It appears that there is some protection mechanism in `bash` that makes the attack unsuccessful. Actually, `bash` automatically downgrade its privilege if it is executed in `Set-UID` root context; this way, even if you can invoke `bash`, you will not gain the root privilege.

```
$ su root
Password: (enter root password)
# cd /bin
# rm sh
# ln -s bash sh // link /bin/sh to /bin/bash
# exit
$ ./retlib          // launch the attack by running the vulnerable program
```

However, there are ways to get around this protection scheme. Although `/bin/bash` has restriction on running `Set-UID` programs, it does allow the real root to run shells. Therefore, if you can turn the current `Set-UID` process into a real root process, before invoking `/bin/bash`, you can bypass that restriction of `bash`. The `setuid(0)` system call can help you achieve that. Therefore, you need to first invoke `setuid(0)`, and then invoke `system("/bin/sh")`; all of these have to be done using the return-to-libc mechanism. The incomplete exploit code is given in the following:

```
/* exploit_2.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 W, X, Y, Z */
/* You need to decide the addresses and
   the values for W, X, Y, Z. The order of the following
   three statements does not imply the order of W, X, Y, Z. */
*(long *) &buf[W] = some address ;    // system()
*(long *) &buf[X] = some address ;    // address of "/bin/sh"
*(long *) &buf[Y] = some address ;    // setuid()
*(long *) &buf[Z] = 0;                // parameter for setuid

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

2.5 Task 3: Address Randomization

Now, we turn on the Fedora's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

```
$ su root
Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=1
```

3 Guidelines: Understanding the function call mechanism

3.1 Find out the addresses of libc functions

To find out the address of any libc function, you can use the following gdb commands (a.out is an arbitrary program):

```
$ gdb a.out

(gdb) b main
(gdb) r
(gdb) p system
$1 = {<text variable, no debug info>} 0x9b4550 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0x9a9b70 <exit>
```

From the above gdb commands, we can find out that the address for the `system()` function is `0x9b4550`, and the address for the `exit()` function is `0x9a9b70`. The actual addresses in your system might be different from these numbers.

3.2 Putting the shell string in the 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 C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable **SHELL** points directly to `/bin/bash` and is needed by other programs, so we introduce a new shell variable **MYSHELL** and make it point to `zsh`

```
$ export MY_SHELL=/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("MY_SHELL");
    if (shell)
        printf("%x\n", 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 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.

3.3 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.

```
/* 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
```

3.4 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).
- **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 the local variables. See Figure 1(d).

3.5 Leaving `foo()`

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

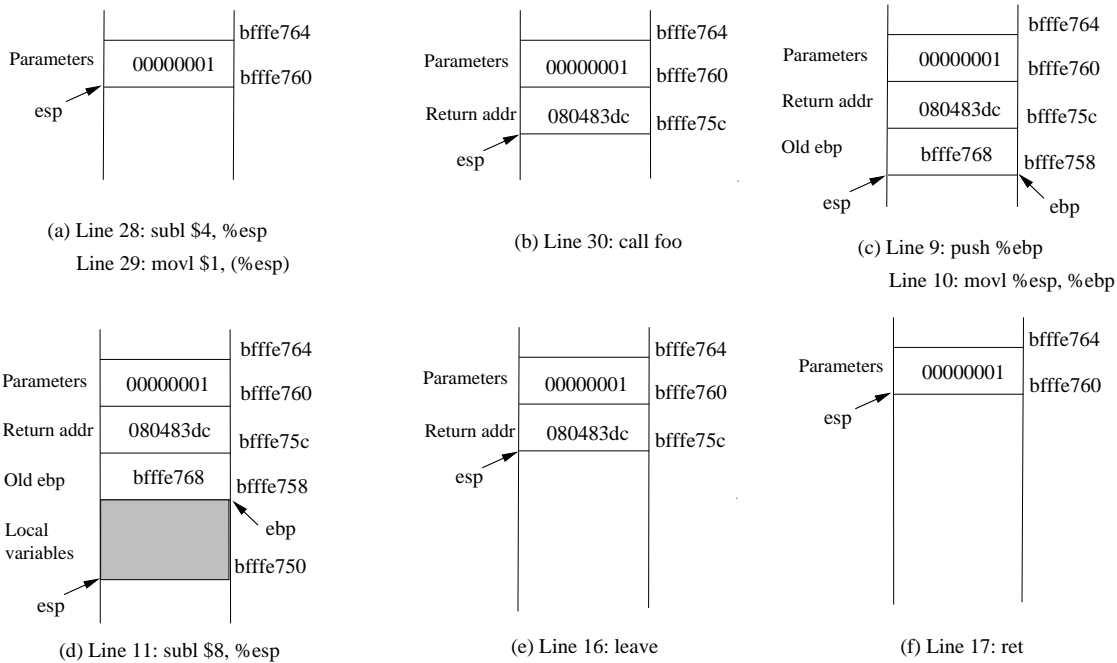


Figure 1: Entering and Leaving `foo()`

- **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).

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>