Return-to-libc Attack Lab

Derived from ©2006 - 2014 Wenliang Du, Syracuse University. Do not redistribute with explicit consent from MAJ Benjamin H. Klimkowski (usma@benklim.org) or CPT Michael Kranch, United States Military Academy

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 to this location 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 64-bit 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.

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 simply 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 in this type of buffer-overflow attack. In this lab, we disable these features using the following commands:

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

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard", also more generally known as stack cookies or stack canaries, 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.

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), 52, badfile);
    return 1;
}
int main(int argc, char **argv){
    setuid(0);
    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 with `sudo`, and `chmod` the executable to `4755`:

```
sudo gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chmod 4755 retlib
ls -l retlib #should have the setuid bit set and should be owned by root
```

The above program has a buffer overflow vulnerability. It first reads an input of size 52 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

In this task we will write a program `exploit.c` that will form the payload in the **badfile**. The payload will operate much like a traditional stack-based buffer overflow, except that instead of directing the instruction pointer to code on the stack, we will direct the code the `system()` function in `libc`, which is in executable memory. Further complicating matters, our code will need to account for the `System V` calling convention. In order to do write the code to make the payload, we need to determine the follow:

1. Determine the offset between `buffer` and `rbp`
2. Determine the location of `system()`
3. Create an environmental variable to hold the string `/bin/sh`
4. Find the address of our environmental variable
5. Find a gadget to load the address of the environment variable into a register

2.3.1 Determine the offset between `buffer` and `rbp`

You can determine the offset in `gdb` per the standard buffer overflow lab and the generic way for a 32 bit process. Remember, the offset caculates the difference between `buffer and rbp`, and a 64-bit address is 8 bytes long.

2.3.2 Determine the location of `system()`

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
```

2.3.3 Create an environmental variable to hold the string `/bin/sh`

One of the challenges 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 `sh`

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

2.3.4 Find the address our environmental variable

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 vulnerabile 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.

Another technique is to print off all the environmental variables in `gdb`. With either technique, you may find it necessary to tweak the string in the debugger–HINT: you need to remove the `MYSHELL=` portion of the string.

2.3.5 Find a gadget to load the address of the environment variable into a register

This step consists of multiple smaller steps, and it is arguably the toughest to complete. First complete the following question:

> **Question 1:** Draw an illustration of what a normal function call is supposed to look like using the `Standard V` convention for a function with one argument.

Notice that unlike our 32-bit `CDECL` executables, we need to find a way to load something into `rdi` in order for this attack to work. One way we can do this is to find a sequence of instructions ending in `RET` in memory. These sequences are referred to as *gadgets* in return-oriented programming. An ideal gadet would be:

```
pop %rdi
ret
```

> **Question 2:** Explain what this sequence of instructions above would do.

The following command is one way to find this gadet:[1]

```
xxd -c1 -p /lib/x86_64-linux-gnu/libc.so.6 | grep -n -B1 c3 |
grep 5f -m1 | awk '{printf"%x\n",$1-1}'
```

> **Question 3:** Explain what each part of the above command does.

We now have the relative address from the our gadget to the beginning of the `libc` library. Now we have to find the absolute address. There a few ways to do this subtask, but one of the easiest is to start the

[1] http://crypto.stanford.edu/ blynn/rop/

`retlib` program in the debugger and break on `main`. Open another terminal and look for the `retlib` process and find the start of `libc` in `/proc/<pid>/maps`.

```
eecs@client_victim:~$ sudo ps -aux | grep retlib
eecs      39401  0.1  1.5  81344 31040 pts/19   S+   15:43   0:00 gdb retlib
eecs      39441  0.0  0.0   4356   756 pts/19   t    15:46   0:00 /home/eecs/retlib
eecs      39447  0.0  0.0  21292   932 pts/18   S+   15:46   0:00 grep --color=auto
eecs@client_victim:~$ cat /proc/39441/maps
00400000-00401000 r-xp 00000000 fc:00 131694          /home/eecs/retlib
00600000-00601000 r--p 00000000 fc:00 131694          /home/eecs/retlib
00601000-00602000 rw-p 00001000 fc:00 131694          /home/eecs/retlib
00602000-00623000 rw-p 00000000 00:00 0               [heap]
0xDEADBEEF-0XDEADDEAD r-xp 00000000 fc:00 266204
                                /lib/x86_64-linux-gnu/libc-2.23.so
```

NOTE: the last two lines above are actually together and the addresses are made-up. With the base of `libc` and the distance to our gadet determined we can now complete the the exploit code.

2.3.6 Writing the exploit code

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[52];
 FILE *badfile;
 badfile = fopen("./badfile", "w");
 /* You need to decide the addresses and
    the values for X, Y, Z.*/
  *(long *) &buf[X] = address;
  *(long *) &buf[Y]= address;
  *(long *) &buf[Z] = address;
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

You need to figure out the values for the addresses above, as well as to find out where to store those addresses. Use gdb to help your troubleshooting.

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, you should get the root shell at this point.

It should be noted that after you exit the shell, the code may crash, which may cause the user to grow suspicious.

Question 4: While not necessary for this attack, how could you get the program to terminate quietly and thus make the attack more stealthy?

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

In your report, please answer the following questions:

Question 5: Please describe how you decide the values for `X`, `Y` and `Z` Either show us your reasoning, or if you use trial-and-error approach, show your trials.

Question 6: 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.

2.4 Task 2: Address Randomization

In this task, let us turn on the Ubuntu's address randomization protection. 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 return-to-libc attack 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=2
```

2.5 Task 3: Stack Guard Protection

In this task, let us turn on the Ubuntu's Stack Guard protection. Please remember to turn off the address randomization protection. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the Stack Guard protection make your return-to-libc attack difficult? You should describe your observation and explanation in your lab report. 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
```

3 Submission

Your group needs to submit a detailed lab report to describe what you have done and what you have observed. This includes

- explanations and supporting evidence for all specified questions and submission items

- time spent
- any challenging points or thoughts on what you found interesting during the lab
- a reflection on security princples
- a separate individual report for the optional lab task, if you chose to complete.

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] Lynn, Ben 64-bit Linux Return-Oriented Programming Available at http://crypto.stanford.edu/ blynn/rop/

[3] Saif El-Sherei Return-to-libc https://www.exploit-db.com/docs/28553.pdf

[4] 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