Buffer Overflow Lab

1 Background

The learning objective of this lab is to gain the first-hand experience on buffer-overflow vulnerability by putting what you have learned about the vulnerability from class into action. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses). An overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, you are given a program with a buffer-overflow vulnerability. Your task is to develop a scheme to exploit the vulnerability and finally execute what you want. In addition to the attack, you will experience the effect of several protection schemes that have been implemented in the operating system to counter against the buffer-overflow attacks. You will evaluate whether the schemes work or not and understand why or why not.

2 Lab Tasks

2.1 Initial setup

You can execute the lab tasks using `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:

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

StackGuard Protection. 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* flag. For example, to compile a program test.c with Stack Guard disabled, you may use the following command:

```
$ gcc -fno-stack-protector test.c
```

Non-Executable Stack. In `Ubuntu` 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
```

2.2 The Vulnerable Program

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

int shell(){

printf("YOU GOT IT !!\n");

return 1;
}

int vuln(char *str){

     char buffer[24];

     strcpy(buffer, str);

     return 1;
}

int main(int argc, char **argv){

     char str[500];
     FILE *input;

     input = fopen("input", "r");
     fread(str, sizeof(char), 500, input);

     vuln(str);

     printf("YOU REALLY GOT IT !!\n");
     return 1;
}
```

Compile the above vulnerable program by following commnd (note the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
  $ gcc -o stack -z noexecstack -fno-stack-protector stack.c
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called “input”, and then passes this input to another buffer in the function `vuln()`. The original input can have a maximum length of 50 bytes, but the buffer in `vuln()` has only 12 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. It should be noted that the program gets its input from a file called “input”. This file is under users control. Your objective is to create the contents for “input”, such that when the vulnerable program copies the contents into its buffer, we print ”YOU GOT IT !!”.

2.3 Task 1: Exploiting the Vulnerability

You are required to use a python script or a C program (exploit.c/exploit.py) to generate the ”input”. You need to develop the exploit by creating properly crafted inputs to overwrite the return address with the address you want. After you finish creating the input from a C or python program, compile and run it. This will generate the contents for “input”. Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to print ”YOU GOT IT !!”. In case you implement a safe exit then you will also be able to print ”YOU REALLY GOT IT !!”.

Important: Please compile your vulnerable program first. Please note that the program which generates the input, can be compiled with the default Stack Guard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in stack.c, which is compiled with the Stack Guard protection disabled.

```
$ gcc -o exploit exploit.c
$./exploit        // create the input
$./stack          // run the vulnerable program
YOU GOT IT !!
NOW YOU REALLY GOT IT !!
$
```

2.4 Task 2: Address Randomization

Now, please turn on the Ubuntu’s address randomization. We run the same attack developed in Task 1. Can you print the output? If not, what is the problem? How does the address randomization make your attacks difficult? You should make some good observations here which will help you in further labs. You can use the following instructions to turn on the address randomization:

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

If running the vulnerable code once does not get the output, how about running it for many times? You can run `./stack` in the following loop , and see what will happen. If your exploit program is designed properly, you should be able to get the output after a while. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait).

```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

2.5 Task 3: Stack Guard

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In your previous tasks, we disabled the “Stack Guard” protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the *-fno-stack-protector’* option. For this task, you will recompile the vulnerable program, stack.c, to use GCC’s Stack Guard, execute task 1 again, and make your observations. You may discuss any error messages you observe.

In the GCC compilers, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the flag mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable Stack Guard.

2.6 Task 4: Non-executable Stack

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, you intentionally made stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in Task 1. Can you get the output? If not, what is the problem? How does this protection scheme make your attacks difficult. You should understand how the protection works. You can use the following instructions to turn on the non-executable stack protection.

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

It should be noted that non-executable stack only makes it impossible to run code on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The *return-to-libc* attack is an example.

If you are using the Ubuntu 12.04 VM, whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the non-executable stack protection does not work, see whether you can figure out the problem.

2.7 Task 5: Running a Shellcode (Own time work)

if you want to attack and get a shell, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

int main( ) {
   char *name[2];

   name[0] = ``/bin/sh'';
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

The shellcode is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code, and see whether a shell is invoked.

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

const char code[] =
  "\x31\xc0"         /* Line 1:  xorl    %eax,%eax              */
  "\x50"             /* Line 2:  pushl   %eax                   */
  "\x68""//sh"       /* Line 3:  pushl   $0x68732f2f            */
  "\x68""/bin"       /* Line 4:  pushl   $0x6e69622f            */
  "\x89\xe3"         /* Line 5:  movl    %esp,%ebx              */
  "\x50"             /* Line 6:  pushl   %eax                   */
  "\x53"             /* Line 7:  pushl   %ebx                   */
  "\x89\xe1"         /* Line 8:  movl    %esp,%ecx              */
  "\x99"             /* Line 9:  cdq                            */
  "\xb0\x0b"         /* Line 10: movb    $0x0b,%al              */
  "\xcd\x80"         /* Line 11: int     $0x80                  */
;

int main(int argc, char **argv){
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Please use the following command to compile the code (note the `execstack` option):

```
  $ gcc -z execstack -o call_shellcode call_shellcode.c
```

A few places in this shellcode are worth mentioning. First, the third instruction pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol. Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and `%edx` registers, respectively. Line 5 stores `name[0]` to `%ebx`; Line 8 stores `name` to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one (`cdq`) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting `%edx` to 0.

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack 49*, Volume 7, Issue 49. Available at http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html