CS220 - Computer System II
Lab 10 – return-to-libc

Due: 11/09/2016, 11:59pm

1 Introduction

In this lab, you will write a program that will spawn a shell, but with a twist. You will not call a function that will actually create the shell, but you will corrupt the return address to *return* to the function that creates a shell. Attacks that invoke functionality by returning to beginning of a function rather than calling a function are called return-to-libc attacks. The libc here is only due to historic reasons, but the attack itself works on any function. `libc` is a library that contains several useful juicy functions worth returning to!

Note: the instructions in this lab will refer to addresses in our address space when we compiled and ran the lab. Your addresses may differ due to the different versions of `libc` or Address Space Layout Randomization (ASLR). These differences do not affect the idea of this lab, but will require your attention to detail.

2 Getting Started

Create a folder **Lab10** with four files: **prelim.c**, **ret2libc.c**, **get_ebp.S** and a Makefile. The Makefile should have three targets:

prelim This executable will demo key ideas discussed in §3.

ret2libc This executable will build your attack using code found in both **ret2libc.c** and **get_ebp.S**.

clean Removes the two above executables and any temporary files. Exactly what you're expecting.

3 Background

Function `int system (char *command)`[1] is a powerful libc function that allows execution of commands from within a C program. Try the following code in **prelim.c**, build it with the Makefile, and you will see the directory listing:

[1]Make it a habit to check the man pages whenever you encounter a new library function (i.e. `$ man system`).

```c
#include <stdlib.h>
int main() {
  return system("ls");
}
```

Similarly, replace "ls" with "/bin/bash", and you will spawn a shell. So, the goal is to invoke `system("/bin/bash");` without actually invoking it.

Strategy: First, we will identify the stack layout when the control enters `system`. Then, we will corrupt the return address of the function and replace it with address of the `system` function. At the same time, we will set up the stack in such a way that when the function returns, it is equivalent to calling `system` with "/bin/bash" as its argument.

4 Implementation

Our first task is to implement a function that will return the value of the `ebp` register as a 32-bit `uint32_t` value (this type is defined in `<stdint.h>`). Add the function prototype `extern uint32_t* get_ebp(void);` to **ret2libc.c**, and implement it in `get_ebp.S`. (Hint: This is a very simple assembly function of only two instructions.)

Our next task is to examine the stack layout when the control enters `system`. Compile **prelim.c** (do not forget the `-m32` flag in your Makefile) and start it in gdb. Insert a breakpoint at system:

```
(gdb) b system
```

When you hit the breakpoint, examine the contents of the stack. Command `x/16x $esp` tells gdb to print 16 DWORDS in hex starting from `$esp`.

```
(gdb) x/16x $esp
```

Remember `esp` grows towards lower address. So, this will actually give you the first 16 DWORDS on the stack. The first DWORD must be the return address where system will return to. Confirm that it is within the `main()` function.

```
(gdb) x/16x $esp
0xffffdb3c: 0x08048431  0x080484d0  0xf7fee560  0x0804845b
0xffffdb4c: 0xf7fa4ff4  0x08048450  0x00000000  0xffffdbd8
0xffffdb5c: 0xf7e57e46  0x00000001  0xffffdc04  0xffffdc0c
0xffffdb6c: 0xf7fdc860  0xf7ff47f1  0xffffffff  0xf7ffcff4
(gdb) disas main
Dump of assembler code for function main:
0x0804841c <+0>:   push   %ebp
0x0804841d <+1>:   mov    %esp,%ebp
0x0804841f <+3>:   and    $0xfffffff0,%esp
0x08048422 <+6>:   sub    $0x10,%esp
0x08048425 <+9>:   movl   $0x80484d0,(%esp)
0x0804842c <+16>:  call   0x8048300 <system@plt>
0x08048431 <+21>:  leave
0x08048432 <+22>:  ret
End of assembler dump.
(gdb)
```

As expected, 0x08048431 is the first entry, and that is the address in main that system returns to. Next entry must be the argument to system. Confirm it.

```
(gdb) x/s 0x080484d0
0x80484d0:   "ls"
```

So, now we know what the stack layout must be. Within ret2libc.c, write a function `void ret2libc()` that will implement your code.

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

extern uint32_t* get_ebp();

void ret2libc(int dummy) {
  /* TASK 1: Get the value of ebp. Make use of the get_ebp function you have
    previously written. */
  /* TASK 2: Overwrite written address with address of system */
  /* TASK 3: Set up the argument to system as a pointer to string "/bin/bash"
    */
}
```

```
int main() {
  ret2libc(0);
  printf("Done!\n");
  return 0;
}
```

TASK 1: Notice how we have declared that `get_ebp` returns a pointer to an unsigned 32 bit value. This allows us to treat the return value as a base of an array and access each DWORD on the stack as an array access. For example, suppose the return value of get_ebp is `curr_ebp`, then `curr_ebp[0]` is the old ebp of the previous frame that was saved using the `push ebp` instruction. Similarly `curr_ebp[1]` is the address where ret2libc returns to, and `curr_ebp[2]` is the dummy variable.

TASK 2: Now, go ahead and replace the return address of ret2libc with `&system`.

```
  curr_ebp[1] = &system; /* Compiler will complain. Fix it. */
```

TASK 3: We know that when control enters system, the first DWORD will be the address that system returns to, but the second DWORD is where the argument is stored. So, create a string that contains the argument, and set it to curr_ebp[3]:

```
char *str = "/bin/bash";
curr_ebp[3] = str;
```

Compile and run, and you should get a bash prompt. The `ret2libc()` function actually "returned" to system. The system function promptly fed off of the arguments on the stack, and executed a shell. Go ahead and exit the shell.

```
$ exit
exit
Segmentation Fault
```

The program crashed because the system function tried to return to whatever was present in curr_ebp[2], which was not to where `ret2libc()` was originally to return. In fact, ret2libc must return to main. In order to fix it, we simply copy the original return address of ret2libc function to curr_ebp[2]. This should happen at the beginning before we replace it with address of system.

```
curr_ebp[2] = curr_ebp[1];
...
```

Compile and run the program. You should get a shell, and upon exiting from the shell, you should see "DONE!" printed from main.

5 Adding a little spice!

Now, based on what you have learned so far, implement `void ret2libc_generic(char *cmd);`. Your goal is to supply `cmd` as argument to system, as opposed to a fixed string "/bin/bash" in the previous case. The challenge here is that cmd resides at curr_ebp[2]. So, be careful when you overwrite it! Move it first to the location where argument to system goes.

6 Submitting the result

Once you have implemented and tested `ret2libc_generic()`, you are ready to submit your code. Remove binaries and intermediate files from Lab10/. Create a tar.gz of Lab10/ folder with only the files listed in §2.

```
$ tar -cvzf lab12_submission.tar.gz ./Lab10
```

Submit lab10_submission.tar.gz to MyCourses.