CSCI 400 Cryptography Lab - 3

Prof. Faheem Abdur-Razzaaq

September 19, 2019

(Priya Thapa, Emranul Hakim, Lakpa Shona Sherpa, Kenneth Asamoah)

MD5 Collision Attack Lab

Task 1: Generating Two Different Files with the Same MD5 Hash

Emranuel

The purpose of this task is to generate two different files with the same MD5 hash for which we have performed the following

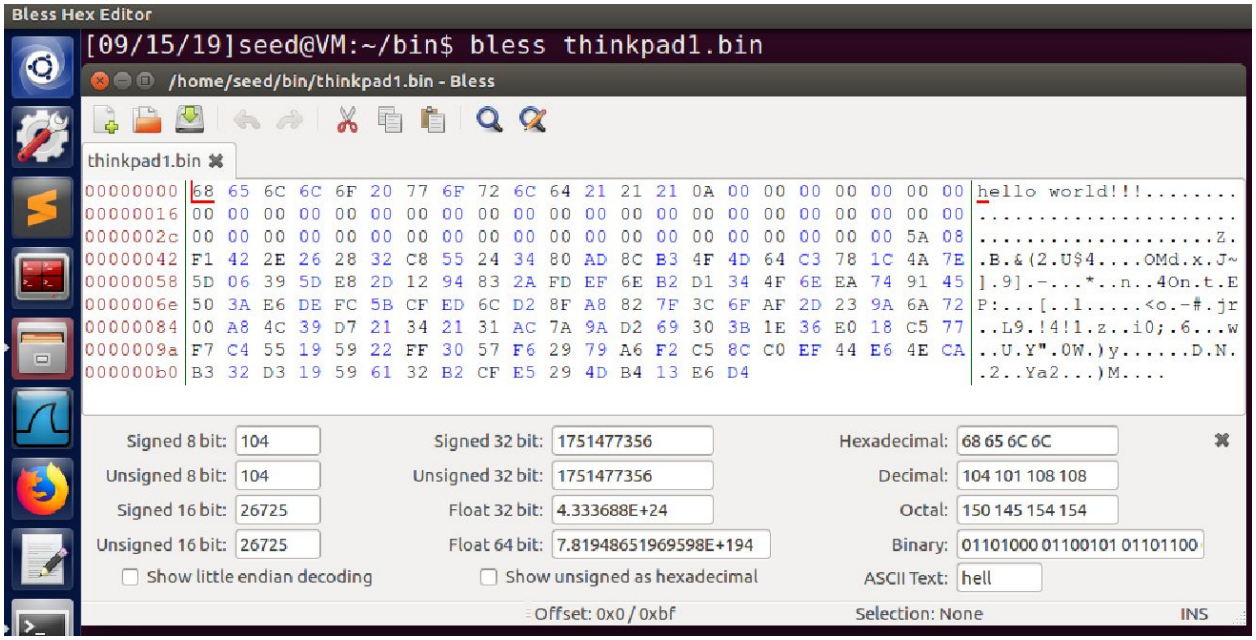
First we created a file name thinkpad.txt which contain a plain text and we run the md5collegen **-p thinkpad.txt -o thinkpad1.bin thinkpad2.bin**

Where the thinkpad1.bin and thinkpad2.bin are two files with the same MD5 hash as we have shown below:

```
[09/18/19]seed@VM:~/bin/lep$ truncate -s 64 thinkpad.txt
[09/18/19]seed@VM:~/bin/lep$ md5collgen -p thinkpad.txt -o thinkpad1.bin thinkpad2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'thinkpad1.bin' and 'thinkpad2.bin'
Using prefixfile: 'thinkpad.txt'
Using initial value: 1e5e975169e9393df3b8d64b98df5fd0

Generating first block: ..........
Generating second block: S10.....................................
Running time: 9.56944 s
[09/18/19]seed@VM:~/bin/lep$
```

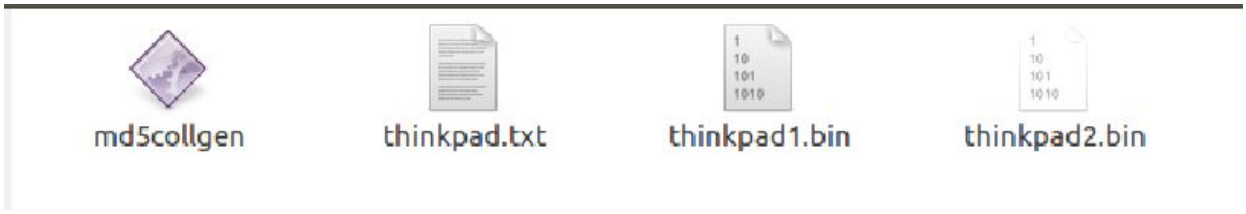
Question1. If the length of your prefix file is not a multiple of 64, what is going to happen?

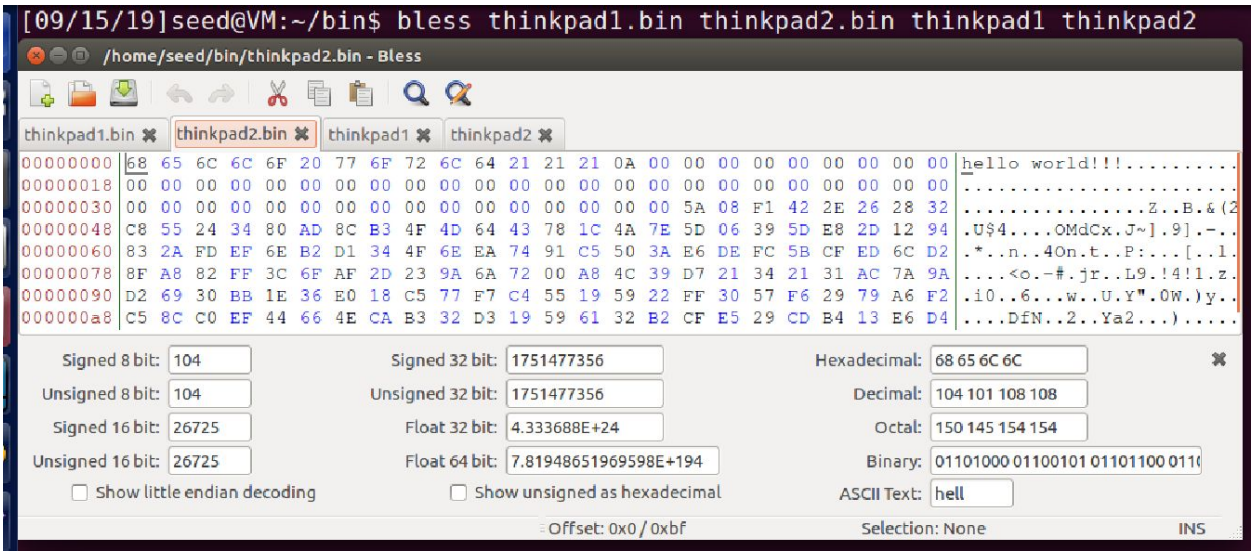
Answer- It will be padded with the Zeros(0's). From the result bless, we notice that it has been padded with zeros. This is because MD5 processes blocks of size 64 bytes.



Question 2. Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.

Answer- No zeros padding is observed to see this we use bless tool and observe the action





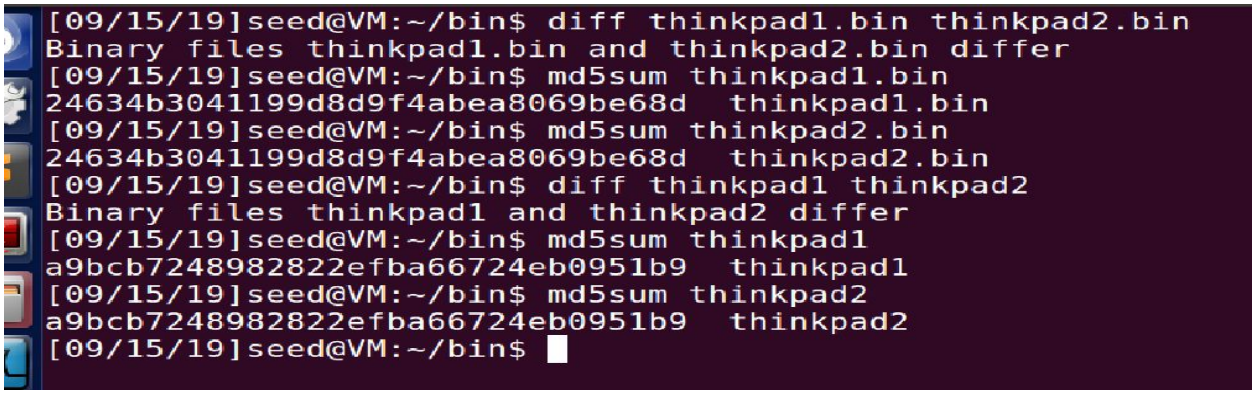
Question 3. Are the data (128 bytes) generated by md5collgen completely different for the two output files? Please identify all the bytes that are different.

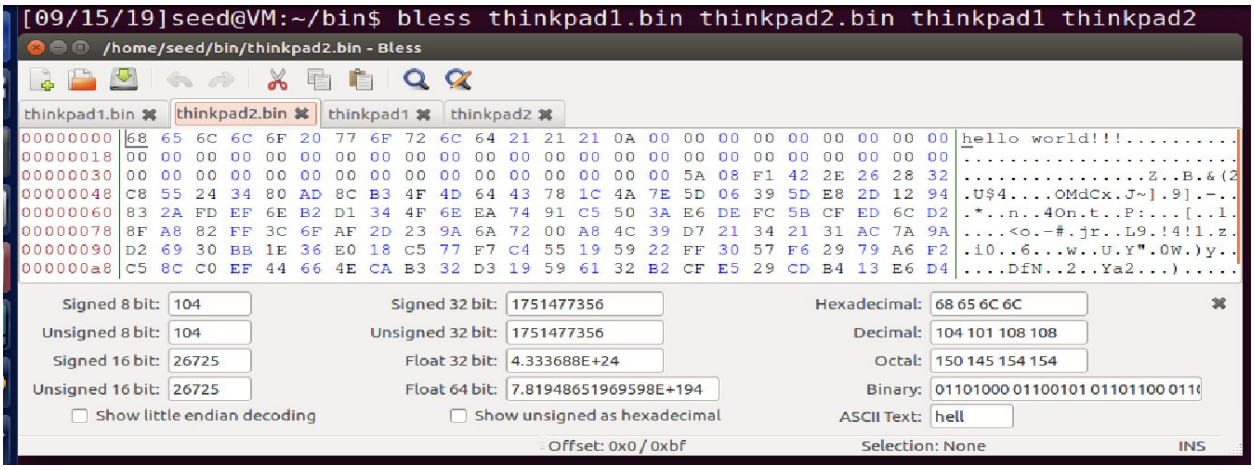
Answer- No, not all bytes are different. We observe some bytes are different which are highlighted below.Even though these are two different files their MD5 hash values are the same. However, we observe that their SHA-256 hashes are different as shown below.

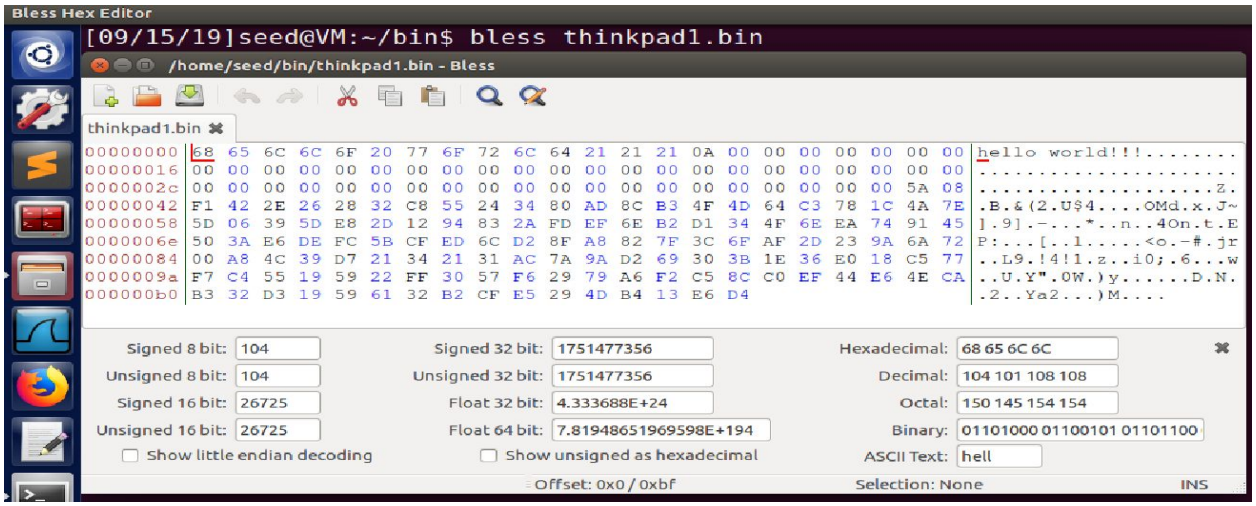
```
hello world!!!..........
........................
..............Z..B.&(2.U$
4....OMd.x.J~].9].-...*..
n..4On.t.EP:...[..l.....<
o.-#.jr..L9.!4!1.z..i0;.6
...w..U.Y".0W.)y......D.N
..2..Ya2...)M....
```

```
hello world!!!..........
........................
..............Z..B.&(2.U$
4....OMdCx.J~].9].-...*..
n..4On.t..P:...[..l.....<
o.-#.jr..L9.!4!1.z..i0..6
...w..U.Y".0W.)y.....DfN
..2..Ya2...).....
```

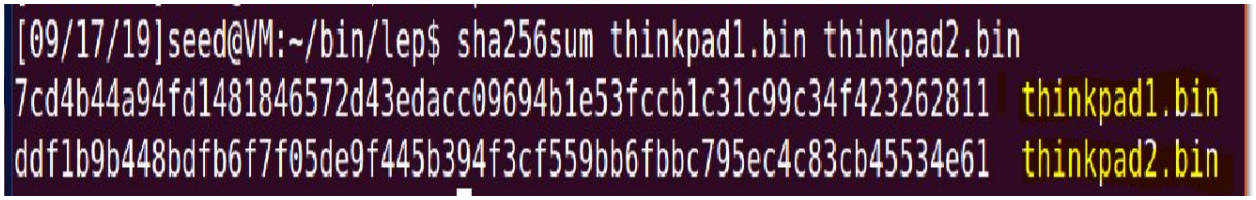
MD5 hash value:







SHA-256 hash value:



Task 2: Understanding MD5's Property

Lakpa

To understand the MD5 property we take the file thinkpad

```
[09/15/19]seed@VM:~/bin$ truncate -s 64 thinkpad
[09/15/19]seed@VM:~/bin$ md5collgen -p thinkpad -o thinkpad1 thinkpad2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'thinkpad1' and 'thinkpad2'
Using prefixfile: 'thinkpad'
Using initial value: 1e5e975169e9393df3b8d64b98df5fd0

Generating first block: ....
Generating second block: S00.............
Running time: 3.7069 s
[09/15/19]seed@VM:~/bin$
```

Verify that the MD5 hashes are same,

```
[09/15/19]seed@VM:~/bin$ md5sum thinkpad1 thinkpad2
a9bcb7248982822efba66724eb0951b9  thinkpad1
a9bcb7248982822efba66724eb0951b9  thinkpad2
[09/15/19]seed@VM:~/bin$
```

```
[09/15/19]seed@VM:~/bin$ cat thinkpad1 thinkpad2 > thinkpad3
[09/15/19]seed@VM:~/bin$ md5sum thinkpad1 thinkpad2 thinkpad3
a9bcb7248982822efba66724eb0951b9  thinkpad1
a9bcb7248982822efba66724eb0951b9  thinkpad2
dd89f560498f6d96839f9b61986d0901  thinkpad3
```

We append the random string to the end of both the file to check MD5 hashes as below and see that their MD5 hashes remain identical.

```
[09/15/19]seed@VM:~/bin$ echo thinkpad >> thinkpad1
[09/15/19]seed@VM:~/bin$ echo thinkpad >> thinkpad2
[09/15/19]seed@VM:~/bin$ md5sum thinkpad1 thinkpad2
3604e5152328fd753f1405e0740ac1e3  thinkpad1
3604e5152328fd753f1405e0740ac1e3  thinkpad2
[09/15/19]seed@VM:~/bin$
```

Task 3: Generating Two Executable Files with the Same MD5 Hash

Priya

The goal for this tasks is to create two different programs with the same hash value for which we perform the following tasks:

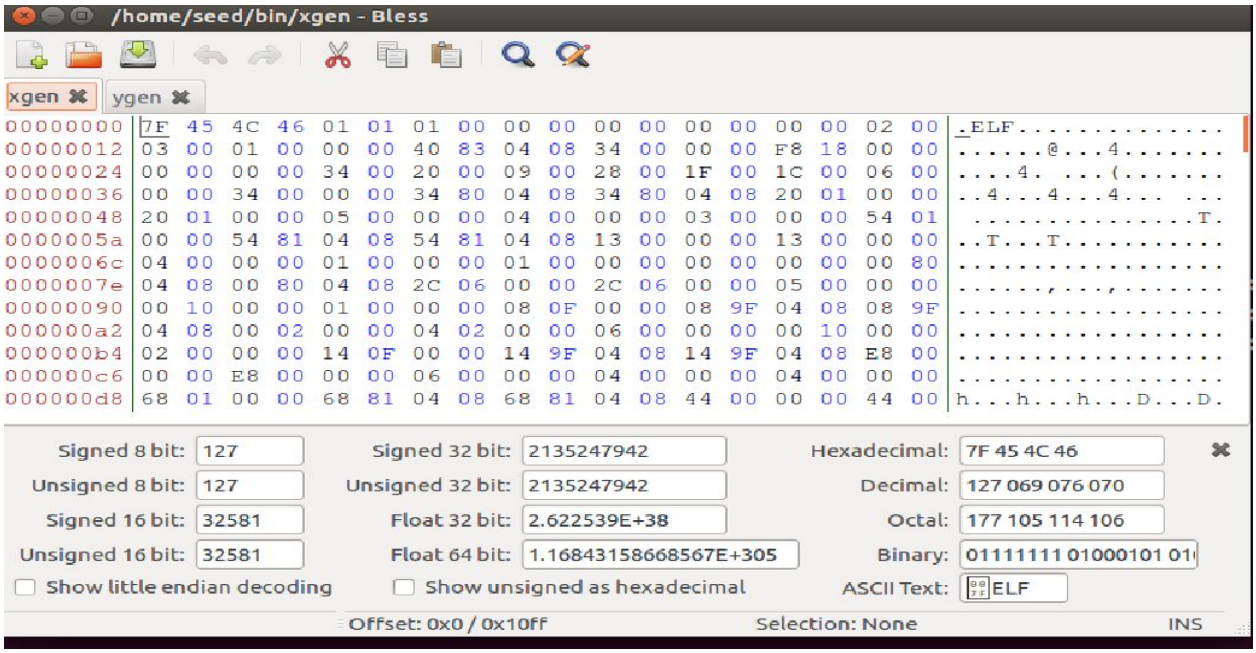
Created a C program and saving a file as guess.c

We compiled the program gcc guess.c -o guess.out,

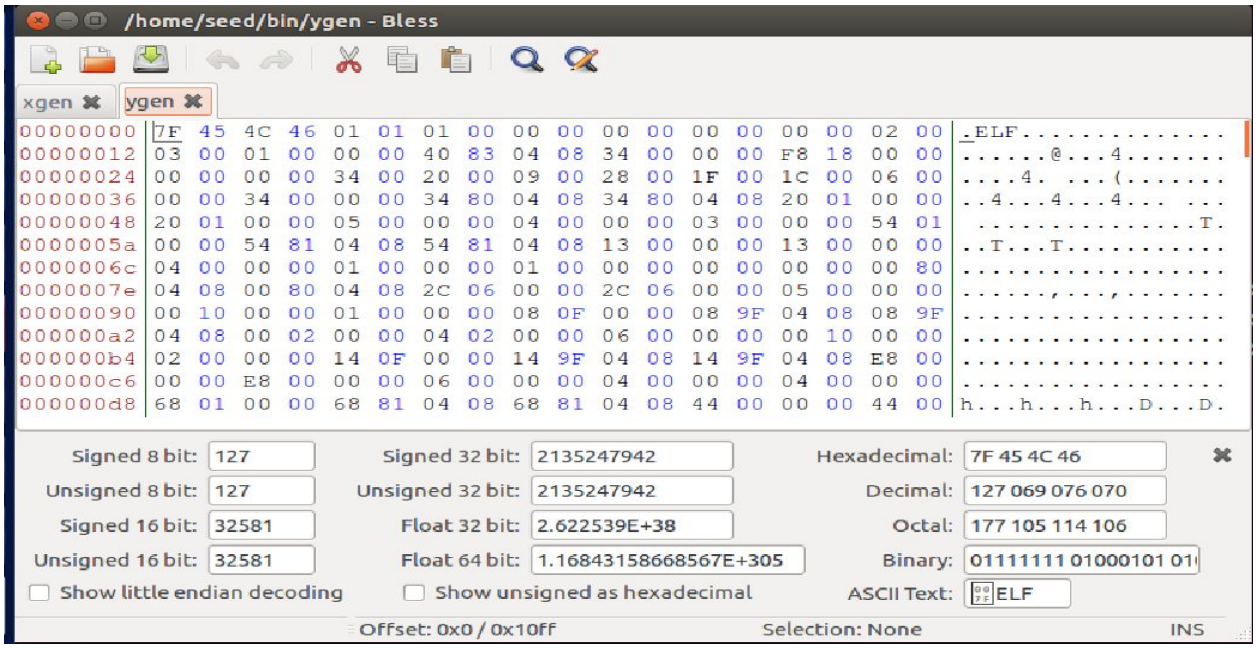
```
[09/15/19]seed@VM:~/bin$ gcc guess.c -o guess.out
[09/15/19]seed@VM:~/bin$ ls
guess.c     md5collgen  thinkpad1      thinkpad2      thinkpad3
guess.out   thinkpad    thinkpad1.bin  thinkpad2.bin  thinkpad.txt
[09/15/19]seed@VM:~/bin$ head -c 4224 guess.out > prefix
[09/15/19]seed@VM:~/bin$ md5collgen -p prefix -o xgen ygen
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'xgen' and 'ygen'
Using prefixfile: 'prefix'
Using initial value: e48516573394fa1a9b6005d9dec8c990

Generating first block: .
Generating second block: W...!.................
Running time: 1.32897 s
[09/15/19]seed@VM:~/bin$
```

Now we have two files with the same MD5 hash but different suffixes ‘xgen’ and ‘ygen’ And we used the bless tool to see the difference in the bytes as shown below:

Xgen:



Ygen:



With tail-c, commonend,executable permission to both files and run them. Note that the output differs :

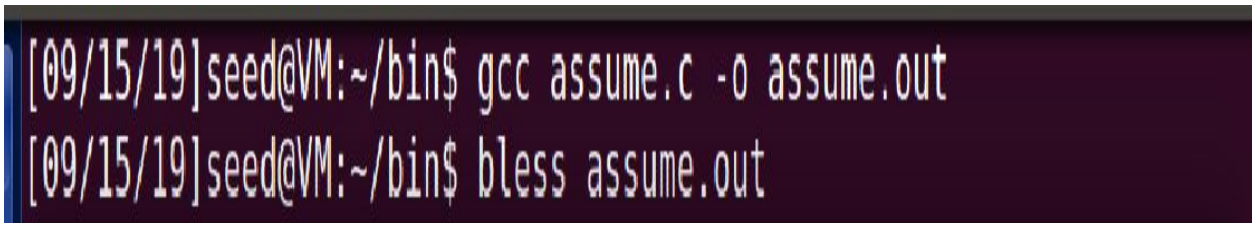
```
[09/15/19]seed@VM:~/bin$ tail -c 4353 guess.out > commonend
[09/15/19]seed@VM:~/bin$ cat commonend >> xgen
[09/15/19]seed@VM:~/bin$ cat commonend >> ygen
[09/15/19]seed@VM:~/bin$ chmod +x xgen
[09/15/19]seed@VM:~/bin$ chmod +x ygen
[09/15/19]seed@VM:~/bin$ ./xgen
4841414141414141414148414141414141414141484141414141414141414841414141414141414148414141414141414141484141414141
4141414148414141c58ceef374bc148d21335f0a964850bf85a6184541a88a28f681fac63e1dcb3bcdb7e3f249224aeb6bd73cf597d44719
3dfbdb37637db7f722794d87669651a4c32423d31181cafdd5c2d58d9ec9bd58917cbff21c1e3976c5cce9ac9fad4ff7f9eb5b8ba8fabfda
990e81c3def105899bebf63d08adae7633a384eac00000454e44
[09/15/19]seed@VM:~/bin$ ./ygen
4841414141414141414148414141414141414141484141414141414141414841414141414141414148414141414141414141484141414141
4141414148414141c58ceef374bc148d21335f0a964850bf85a6984541a88a28f681fac63e1dcb3bcdb7e3f249224aeb6bd73cf517d54719
3dfbdb37637db7f722714d87669651a4c32423d31181cafdd5c2d58d9ec9bd58997cbff21c1e3976c5cce9ac9fad4ff7f9eb5b8ba8fabfda
990681c3def105899bebf63d08adae7e33a384eac00000454e44
[09/15/19]seed@VM:~/bin$ ./xgen > xoutput
[09/15/19]seed@VM:~/bin$ ./ygen > youtput
[09/15/19]seed@VM:~/bin$ diff -q xoutput youtput
Files xoutput and youtput differ
[09/15/19]seed@VM:~/bin$
```

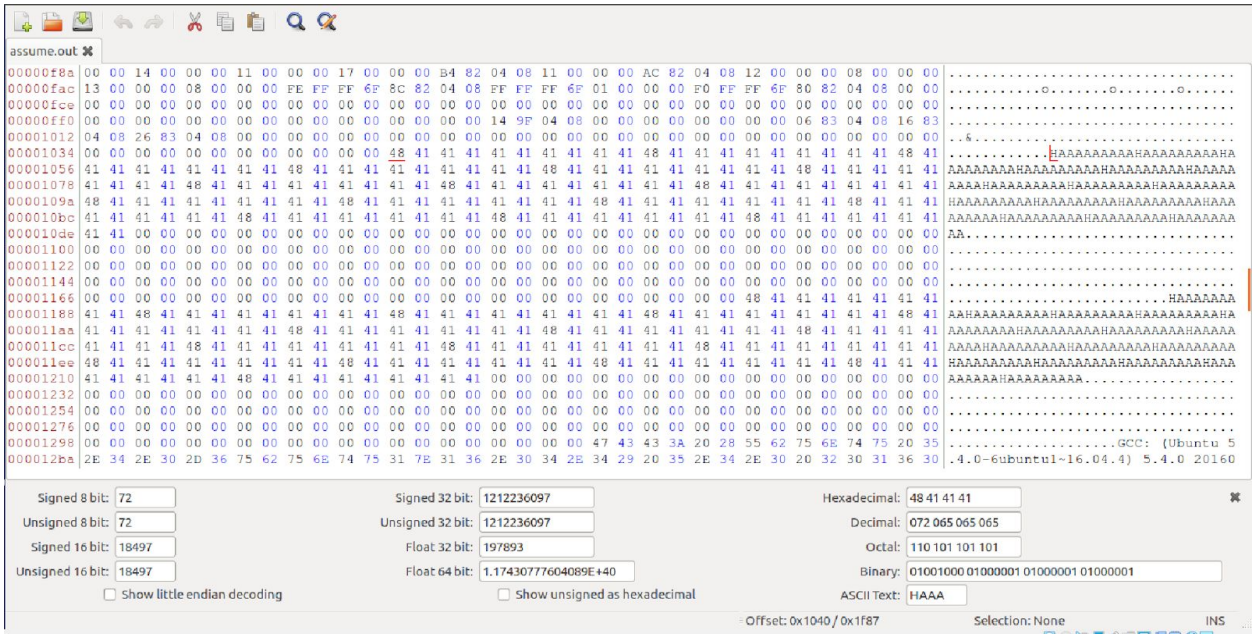
<u>Task 4: Making the Two Programs Behave Differently</u>

<u>Emranuel , Lakpa, Priya</u>

The task goal is to produce two programs that behaves completely different for which we started first with creating the the c program and save a file as assume.c

Now we compile the program using gcc assume.c and the output binary is stored as assume.out and run the bless command to view the assume.out content as shown below:



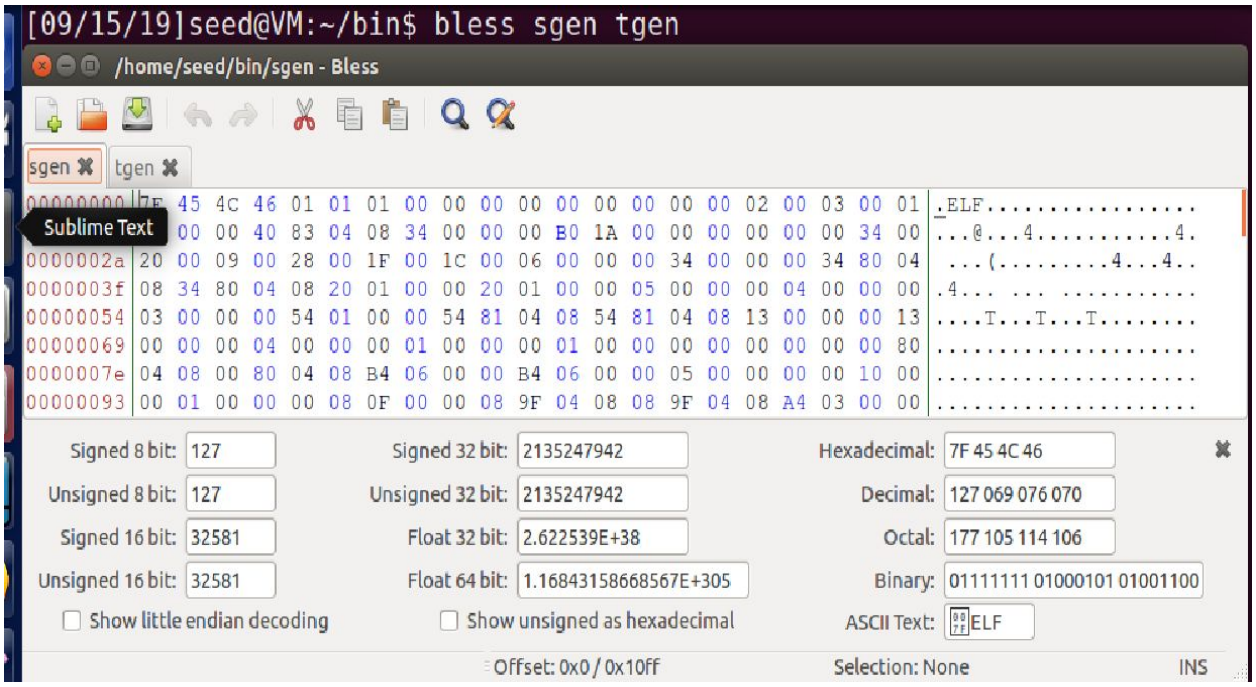


We note that the first array starts at position 0x1040 = 4160. As before, the prefix can extend until there, but I will continue to use the position 0x1080 so that the generated data will start in the middle of the array.

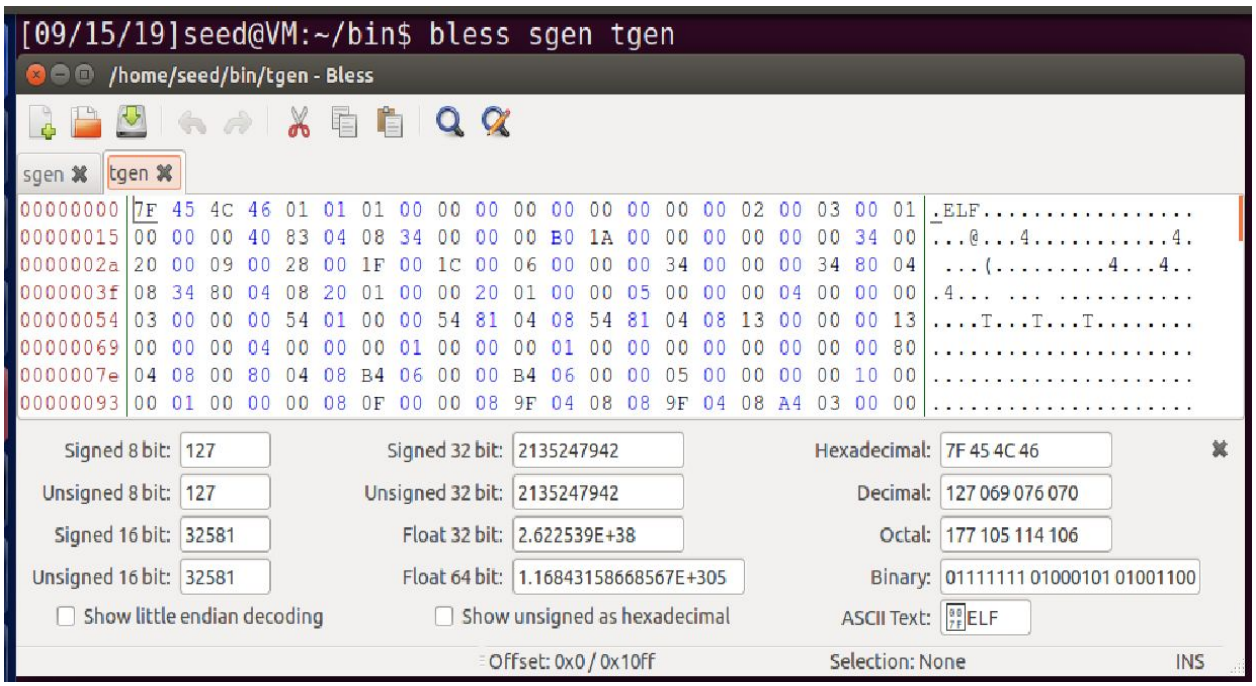
Since 0x1080 will refer to the first 4224 bytes of the data, we will run head -c 4224 assume.out > prefix to store the first 4224 bytes in prefix. Then we will run md5collgen -p prefix -o sgen tgen which will generate two binaries with the same MD5 sum but they will differ for which we used SHA-256 hashes

```
[09/15/19]seed@VM:~/bin$ head -c 4224 assume.out > prefix
[09/15/19]seed@VM:~/bin$ md5collgen -p prefix -o sgen tgen
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'sgen' and 'tgen'
Using prefixfile: 'prefix'
Using initial value: e55d99cfde856518972950b5b6e844c4

Generating first block: ................
Generating second block: S11..................................
Running time: 19.3478 s
[09/15/19]seed@VM:~/bin$
```

Sgen:


[09/15/19]seed@VM:~/bin$ bless sgen tgen
/home/seed/bin/sgen - Bless
sgen
tgen
Sublime Text
00000000 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 02 00 03 00 01 .ELF.....
00000015 00 00 00 40 83 04 08 34 00 00 00 B0 1A 00 00 00 00 00 00 34 00 ...@...4.....4.
0000002a 20 00 09 00 28 00 1F 00 1C 00 06 00 00 00 34 00 00 00 34 80 04 ...(.....4...4..
0000003f 08 34 80 04 08 20 01 00 00 20 01 00 00 05 00 00 00 04 00 00 00 .4... ... .
00000054 03 00 00 00 54 01 00 00 54 81 04 08 54 81 04 08 13 00 00 00 13 ....T...T...T.....
00000069 00 00 00 04 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00 00 80 .....
0000007e 04 08 00 80 04 08 B4 06 00 00 B4 06 00 00 05 00 00 00 00 10 00 .....
00000093 00 01 00 00 00 08 0F 00 00 08 9F 04 08 08 9F 04 08 A4 03 00 00 .....
Signed 8 bit: 127
Signed 32 bit: 2135247942
Hexadecimal: 7F 45 4C 46
Unsigned 8 bit: 127
Unsigned 32 bit: 2135247942
Decimal: 127 069 076 070
Signed 16 bit: 32581
Float 32 bit: 2.622539E+38
Octal: 177 105 114 106
Unsigned 16 bit: 32581
Float 64 bit: 1.16843158668567E+305
Binary: 01111111 01000101 01001100
Show little endian decoding
Show unsigned as hexadecimal
ASCII Text: ELF
Offset: 0x0 / 0x10ff
Selection: None
INS

Tgen:

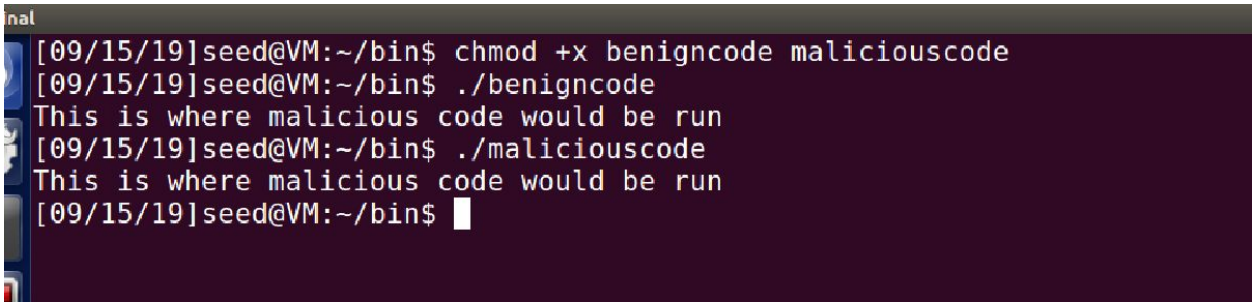

[09/15/19]seed@VM:~/bin$ bless sgen tgen
/home/seed/bin/tgen - Bless
sgen
tgen
00000000 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 02 00 03 00 01 .ELF.....
00000015 00 00 00 40 83 04 08 34 00 00 00 B0 1A 00 00 00 00 00 00 34 00 ...@...4.....4.
0000002a 20 00 09 00 28 00 1F 00 1C 00 06 00 00 00 34 00 00 00 34 80 04 ...(.....4...4..
0000003f 08 34 80 04 08 20 01 00 00 20 01 00 00 05 00 00 00 04 00 00 00 .4... ... .
00000054 03 00 00 00 54 01 00 00 54 81 04 08 54 81 04 08 13 00 00 00 13 ....T...T...T.....
00000069 00 00 00 04 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00 00 80 .....
0000007e 04 08 00 80 04 08 B4 06 00 00 B4 06 00 00 05 00 00 00 00 10 00 .....
00000093 00 01 00 00 00 08 0F 00 00 08 9F 04 08 08 9F 04 08 A4 03 00 00 .....
Signed 8 bit: 127
Signed 32 bit: 2135247942
Hexadecimal: 7F 45 4C 46
Unsigned 8 bit: 127
Unsigned 32 bit: 2135247942
Decimal: 127 069 076 070
Signed 16 bit: 32581
Float 32 bit: 2.622539E+38
Octal: 177 105 114 106
Unsigned 16 bit: 32581
Float 64 bit: 1.16843158668567E+305
Binary: 01111111 01000101 01001100
Show little endian decoding
Show unsigned as hexadecimal
ASCII Text: ELF
Offset: 0x0 / 0x10ff
Selection: None
INS

Sum of sgen & tgen:

```
[09/15/19]seed@VM:~/bin$ md5sum sgen tgen
ffd3080c2f487b73d02d736e198d8aaf  sgen
ffd3080c2f487b73d02d736e198d8aaf  tgen
[09/15/19]seed@VM:~/bin$
```

```
[09/15/19]seed@VM:~/bin$ tail -c 4353 assume.out > commonend
[09/15/19]seed@VM:~/bin$ cat commonend >> sgen
[09/15/19]seed@VM:~/bin$ cat commonend >> tgen
[09/15/19]seed@VM:~/bin$ chmod +x sgen
[09/15/19]seed@VM:~/bin$ chmod +x tgen
[09/15/19]seed@VM:~/bin$ ./sgen
This is where malicious code would be run
[09/15/19]seed@VM:~/bin$ ./tgen
This is where malicious code would be run
[09/15/19]seed@VM:~/bin$ ./sgen > soutput
[09/15/19]seed@VM:~/bin$ ./tgen > toutput
```

```
[09/15/19]seed@VM:~/bin$ head -c +4353 assume.out > end
[09/15/19]seed@VM:~/bin$ tail -c +321 end > commonend
[09/15/19]seed@VM:~/bin$ cp sgen benigncode
[09/15/19]seed@VM:~/bin$ cp tgen maliciouscode
[09/15/19]seed@VM:~/bin$ cat middle >> benigncode
cat: middle: No such file or directory
[09/15/19]seed@VM:~/bin$ cat middle >> maliciouscode
cat: middle: No such file or directory
[09/15/19]seed@VM:~/bin$ cat s >> benigncode
cat: s: No such file or directory
[09/15/19]seed@VM:~/bin$ cat s >> maliciouscode
cat: s: No such file or directory
[09/15/19]seed@VM:~/bin$ cat commonend >> benigncode
[09/15/19]seed@VM:~/bin$ cat commonend >> maliciouscode
[09/15/19]seed@VM:~/bin$ md5sum benigncode maliciouscode
52f3db4559d39d9c29ed9a4d8430ad29  benigncode
52f3db4559d39d9c29ed9a4d8430ad29  maliciouscode
[09/15/19]seed@VM:~/bin$
```



SHA-256 hash value:

Pseudo Random Number Generation Lab

The purpose of this lab is to learn to generate random numbers for security purpose and why we don't really prefer to use typical random number generator for generating secret key . We used pseudo random number generator for some security purpose like encryption key.

Task 1: Generate Encryption Key in a Wrong Way

Priya

In the code:

srand() and time() functions :

```
[09/15/19]seed@VM:~$ cd Documents
[09/15/19]seed@VM:~/Documents$ gcc try.c -lcrypto
[09/15/19]seed@VM:~/Documents$ a.out
1568577125
05042f205f470868d4e9c70ea19c15e4
```

```
[09/15/19]seed@VM:~/Documents$ gcc try.c -lcrypto
[09/15/19]seed@VM:~/Documents$ a.out
1568577498
0a376371c822f0c22c79de01528e4c40
```

No srand() and time() functions :

```
[09/15/19]seed@VM:~/Documents$ gcc try.c -lcrypto
[09/15/19]seed@VM:~/Documents$ a.out
1568577231
67c6697351ff4aec29cdbaabf2fbe346
[09/15/19]seed@VM:~/Documents$ gcc try.c -lcrypto
[09/15/19]seed@VM:~/Documents$ a.out
1568577475
67c6697351ff4aec29cdbaabf2fbe346
```

Result:

For the same seed we will have the same sequence of values.

If we need a different random value every time and we set the same seed every time, then we will get the same "random" value.

Hence, seed is often generated from the current time, in seconds time (NULL).
If we set the seed before taking the random number, we will get the same number unless we call the srand/rand combo multiple times in the same second.

Using of time(NULL) returns the second value when we initialize the seed value.

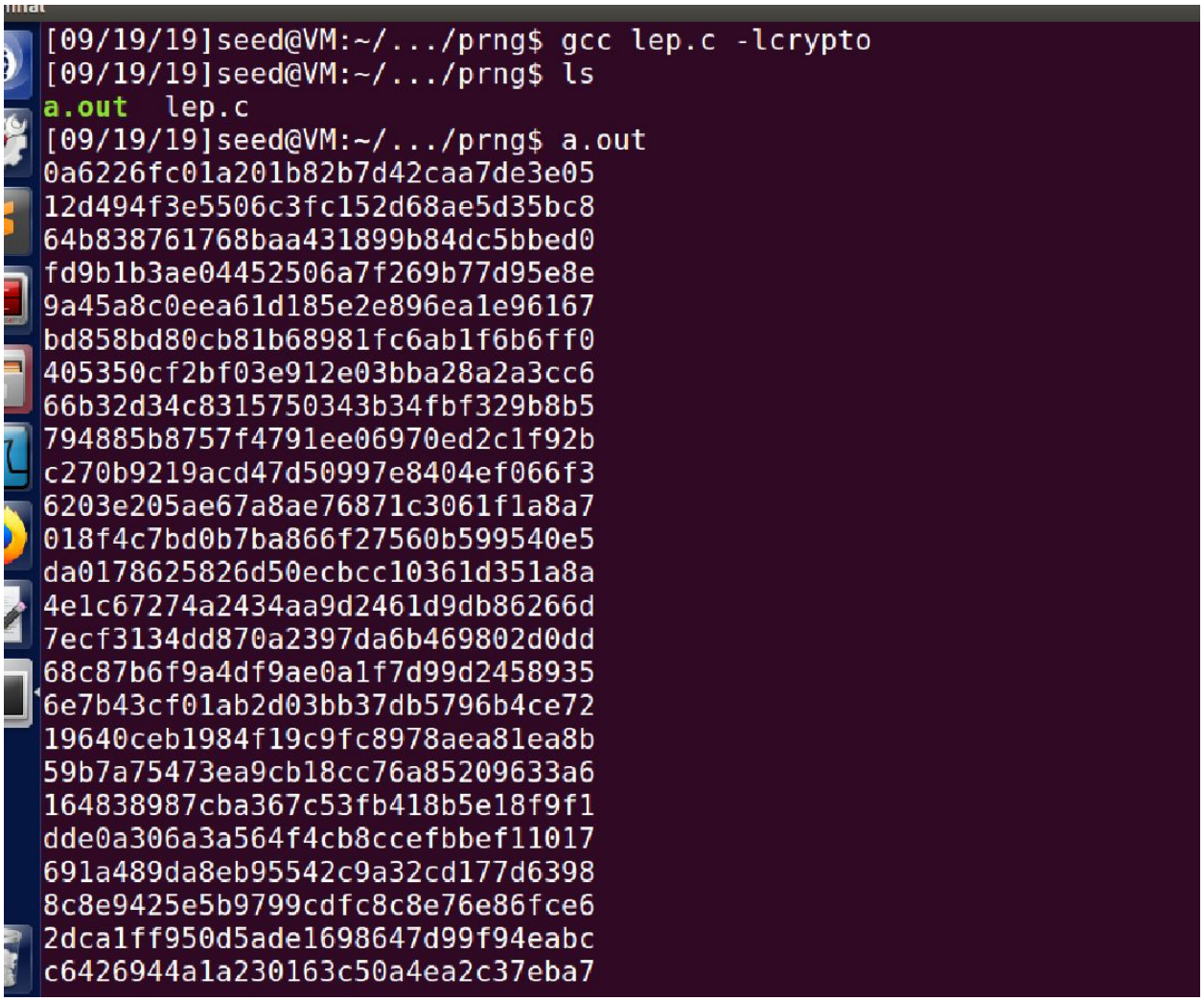
If srand(time(NULL)) is there then, the starting point for producing a series of pseudo-random integers is set and we are getting different random value every time because seconds value is changing.

So, commenting on the srand() and time() functions, seed is not taken from the current time and we are getting the same number.

Task 2: Guessing the Key

Priya Lakpa Emranuel

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

#define KEYSIZE 16

void main()
{
        int i;
        int j;
        FILE *f;
        char key[KEYSIZE];
        f = fopen("keyfile.txt","w");
        for (j = 1524013729; j<= 1524020929; j++){
                srand (j);
                for (i = 0; i< KEYSIZE; i++){
                        key[i] = rand()%256;
                        fprintf(f, "%.2x", (unsigned char)key[i]);
                }
                fprintf(f,"\n");
        }
}
```



We saved our keys in keyfile.txt.

We tried to use this command to guess our key.

```
IV='09080706050403020100A2B2C2D2E2F2=='
KEY= open("keys.txt", "r")
INPUT= '255044462d312e350a25d0d4c5d80a34'


DECRYPTED=$(php -r "print(openssl_decrypt('$ENCRYPTED','aes-128-
cbc',base64_decode('$KEY'),OPENSSL_ZERO_PADDING,base64_decode('$IV')));")
echo '$DECRYPTED='$DECRYPTED
```

Task 3: Measure the Entropy of Kernel
Lakpa

Entropy the kernel has at the current moment :

```
[09/15/19]seed@VM:~/Documents$  cat /proc/sys/kernel/ra
ndom/entropy_avail
2895
```

The change of the entropy:

Mouse Movement:

```
Every 0.1s: cat /proc/sys/...  Sun Sep 15 17:03:39 2019

3055
```

Visiting a Website:

```
Every 0.1s: cat /proc/sys/...  Sun Sep 15 17:05:12 2019

3232
```

Result:

The entropy is generated from the mouse movements and other activities in hardware and in linux kernel makes the random character data available to other operating system processes through the special files /dev/random and /dev/urandom

.

The main purpose of gathering up entropy was to seed a good PRNG.
RNG will stop giving the entropy if there is a stop in mouse movement but PRNG reads the initial entropy and continue the sequence and produce random numbers.
Higher the entropy, the less certainty found in the result.

Task 4: Get Pseudo Random Numbers from /dev/random
Priya

/dev/random is suitable to use when we require the high randomness like in key generation. It will return only the maximum number of entropy from the pool.

It's disadvantage is blocking is the kernel's entropy pool is exhausted and will start working only if entropy is fill up.

After running this command, we are burning up our entropy pool:

```
Every 0.1s: cat /proc/sys/...  Sun Sep 15 17:11:24 2019

20
```

If you do not move your mouse or type anything, the random numbers are generated very slowly, almost drops to 0. The device is waiting for kernel to fill entropy for randomness.

```
[09/15/19]seed@VM:~/Documents$ cat /dev/random | hexdum
p
0000000 75b4 6e6e ce81 b2c5 14fc 747f d23d a6f0
0000010 d830 1c26 1efb 5c15 487c 9445 9f04 54d9
0000020 7398 dd2b 23e6 d954 9028 27ec 7e5f a1ab
0000030 5297 9da0 b20b 03a4 c229 5f1d c4f2 6169
0000040 e86b 4964 8fdb 0d51 5b46 6d7f cd41 b7c9
0000050 1661 4278 78be d39e 924d 2b73 37b8 fdbb
0000060 25f5 174d 382c 3018 e4e8 d8da 8ff5 e599
0000070 36b6 035a 3972 62a0 616b b5d3 a534 ca39
```

Result:

If a server uses /dev/random to generate the random session key with a client, then when client sent request insanely in large number, it floods the network traffic of a server. The main target

is to have kernel /dev/random's entropy exhausted.This request will exhaust the entropy. Once it is exhausted, it will pause and cannot produce enough randomness to seed PRNG.
Therefore, we can launch a Denial-Of-Service (DOS) attack on such a server.

Task 5: GetRandomNumbersfrom /dev/urandom

```
[09/15/19]seed@VM:~/Documents$  head -c 1M /dev/urandom
 > output.bin
[09/15/19]seed@VM:~/Documents$ ent output.bin
Entropy = 7.999853 bits per byte.

Optimum compression would reduce the size
of this 1048576 byte file by 0 percent.

Chi square distribution for 1048576 samples is 214.51,
and randomly
would exceed this value 96.91 percent of the times.

Arithmetic mean value of data bytes is 127.5688 (127.5
= random).
Monte Carlo value for Pi is 3.142880031 (error 0.04 per
cent).
Serial correlation coefficient is -0.001867 (totally un
correlated = 0.0).
```

This is the analysis of random.org.

John Walker's Ent program:

```
Entropy = 7.999805 bits per character.
Optimum compression would reduce the size of this
   1048576 character file by 0 percent.
 Chi square distribution for 1048576 samples is
   283.61, and randomly would exceed this value
   25.00 percent of the times.
 Arithmetic mean value of data bytes is 127.46
   (127.5 = random).
 Monte Carlo value for PI is 3.138961792 (error
   0.08 percent).
 Serial correlation coefficient is 0.000417
   (totally uncorrelated = 0.0
```

Compared to the above picture and our results, we can see that entropy is 7.999853, which is almost to perfect 8 bits per byte.
It also shows the crypto pseudo-random-number-generator with really good randomness property.

Generating a 256-bit encryption key:
Using the commands given in seed lab project:

```
[09/16/19]seed@VM:~/Documents$ gcc urandom.c -lcrypto
[09/16/19]seed@VM:~/Documents$ a.out
67c6697351ff4aec29cdbaabf2fbe3467cc254f81be8e78d765a2e6
3339fc99a[09/16/19]seed@VM:~/Documents$ ent a.out
Entropy = 3.650456 bits per byte.

Optimum compression would reduce the size
of this 7536 byte file by 54 percent.

Chi square distribution for 7536 samples is 666830.85,
and randomly
would exceed this value less than 0.01 percent of the t
imes.

Arithmetic mean value of data bytes is 35.7156 (127.5 =
 random).
Monte Carlo value for Pi is 3.837579618 (error 22.15 pe
rcent).
Serial correlation coefficient is 0.577199 (totally unc
orrelated = 0.0).
```

Using command openssl rand -rand /dev/urandom 256 > p.out

```
[09/16/19]seed@VM:~/Documents$ openssl rand -rand /dev/
urandom 256 > p.out
2048 semi-random bytes loaded
[09/16/19]seed@VM:~/Documents$ ent p.out
Entropy = 7.119441 bits per byte.

Optimum compression would reduce the size
of this 256 byte file by 11 percent.

Chi square distribution for 256 samples is 274.00, and
randomly
would exceed this value 19.75 percent of the times.

Arithmetic mean value of data bytes is 117.9609 (127.5
= random).
Monte Carlo value for Pi is 2.571428571 (error 18.15 pe
rcent).
Serial correlation coefficient is -0.003018 (totally un
correlated = 0.0).
```

Result:

In the 1st figure;

Entropy is 3.65, hence the Entropy, randomness is less. So, this might not be a good seed for PRNG.

In the 2nd figure;
Entropy is 7.11, hence the randomness is more . So, this is a good seed for PRNG.