Report 2: Pseudo Random Generation of Numbers

Generating random numbers is very important for cryptography. So, a computer who utilizes these techniques very often must have a robust and secure way of generating them. In this report we will explore a miriad of techiniques and see how secure an entropic they are.

Task 1: Generating an Encription Key the Wrong Way

Firstly, we will generate an **Encryption key** the wrong way, and for that, we will use a program that uses the random library of c and the time to generate random numbers. The following is the program used:

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

#define KEYSIZE 16

void main()
{
    int i;
    char key[KEYSIZE];
    printf("%lld\n", (long long) time(NULL));
    srand (time(NULL));
    for (i = 0; i< KEYSIZE; i++){
        key[i] = rand()%256;
        printf("%.2x", (unsigned char)key[i]);
    }
    printf("\n");
}
```

The program generates a 128 bit char random **Encryption key** using srand and a seed.

As we can see, we generate the seed for the random function in the line **srand (time(NULL));**. We use the current time as the seed, and then in the line **key[i] = rand()%256;** we generate a number between 0 and 256 to add to the **Encryption key**. **time(NULL)** returns the time in seconds elapsed from January 1, 1970 to now. By running the program we generate the following keys:

```
[03/05/24]seed@VM:~/.../Lab3$ ./task1
1709638435
7c319c868cf47f8429dab249acbf153f
[03/05/24]seed@VM:~/.../Lab3$ ./task1
1709638504
709c3bdc349e3a4f7ea991cde3e0ce05
[03/05/24]seed@VM:~/.../Lab3$
```

If we remove the line defining the seed as the current time, the program uses the default seed for every execution, which is **1**. So the resulting **Encryption keys** are always the same, as we can see in the image bellow:

```
[03/05/24]seed@VM:~/.../Lab3$ ./task1
1709638173
67c6697351ff4aec29cdbaabf2fbe346
[03/05/24]seed@VM:~/.../Lab3$ ./task1
1709638175
67c6697351ff4aec29cdbaabf2fbe346
[03/05/24]seed@VM:~/.../Lab3$ ./task1
1709638176
67c6697351ff4aec29cdbaabf2fbe346
[03/05/24]seed@VM:~/.../Lab3$ 
```

Task 2: Guessing the Key

So, what is the problem with this way of doing it this way? Let us say that Alice generated an **Encryption key** using the program above between the date of **2018-04-17 21:08:49** and **2018-04-17 23:08:49** and used it to encrypt some very important documents. As we saw before, the program above uses time as a seed to generate the **Encryption key**, so, as we know the timeframe of the generation we can execute the program and obtain the same seed as Alice! We can then obtain her **Encryption key** and decrypt all her important files! We will now attempt to get Alice's documents. Let us assume that the documents are in the **pdf** format. The document is encrypted using **AES**, which is a 128-bit cypher that uses blocks to encrypt and decrypt documents. A block consists of 16 bytes and so we will need 16 bytes of plaintext to decrypt it, so how do we get them? But first we need to get Alice's **Encryption Key**, and so we need to generate every possible key in that 2 hour period and test againts some plain text we can find in the **pdf** file. We are lucky as we can get 16 bytes of the header of the files. The beggining part of a **pdf** header is always the version number. At the time the file was created **PDF-1.5** was widely used so we now have 8 bytes of clear data in the form of the version number **%PDF-1.5**. The next 8 bytes are also quite easy to predict and so we now have 16 bytes of plain text. We compare the result to the encrypted data of those 16 bytes and see if it matches. The **plaintext** and **ciphertext** are bellow:

```
Plaintext: 255044462d312e350a25d0d4c5d80a34
Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
IV: 09080706050403020100A2B2C2D2E2F2
```

We also have access to the **Initial Vector(IV)** as it is never encrypted. With all that done we can finally start the brute force attack using the following programs:

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

```c
#define KEYSIZE 16

void main()
{
    int i;
    FILE* f1;
    f1 = fopen("enckeys.txt", "a+");

    for (i = 1524013729; i <= 1524020929; i++) {
        char key[KEYSIZE];
        printf("%lld\n", (long long) i);
        srand(i);

        int j;

        for (j = 0; j < KEYSIZE; j++) {
            key[j] = rand()%256;
            printf("%.2x", (unsigned char) key[j]);
            fprintf(f1, "%.2x", (unsigned char) key[j]);
        }
        fprintf(f1, "\n");
        printf("\n");
    }
}
```

This c program computes from every possible seed (the 2 hour window) starting in **1524013729** and ending in **1524020929**, computes every **Encryption Key** and saves it to the file **enckeys.txt**. Now with all the possible keys we can check which one is actually the right one. And for this we use the python script bellow:

```python
from Crypto.Cipher import AES

file = open("enckeys.txt", "r")

Lines = file.readlines()

for key in Lines:
    aes = AES.new(bytearray.fromhex(key), AES.MODE_CBC,
bytearray.fromhex("09080706050403020100A2B2C2D2E2F2"))
    data = aes.encrypt(bytearray.fromhex("255044462d312e350a25d0d4c5d80a34"))
    if data == bytearray.fromhex("d06bf9d0dab8e8ef880660d2af65aa82"):
        print("Key gotten!\n")
        print(key)
```

This python scrip imports the **AES** algorithm, gets all the keys from the **enckeys.txt** and creates an **AES** with each key and with the **IV** collected, it then encrypts our 16 bytes of plaintext and compares it with the equivalent cyphertext. If they are the same that means we found the actual key used in the encryption! We can see the python script execute in the image bellow:

```
[03/05/24]seed@VM:~/.../Lab3$ python3 cracker.py
Key gotten!

95fa2030e73ed3f8da761b4eb805dfd7
```

So the Encryption Key is **95fa2030e73ed3f8da761b4eb805dfd7**! We found Alice's key and can now decrypt all her encrypted files.

Task 3: Measure the Entropy of Kernel

It is difficult for computers to create randomness. So most Operating Systems get their randomness from the physical world. Linux gets the randomness from these physical resources:

```
void add_keyboard_randomness(unsigned char scancode);
void add_mouse_randomness(__u32 mouse_data);
void add_interrupt_randomness(int irq);
void add_blkdev_randomness(int major);
```

The **OS** is using the timing between keypresses to generate random numbers, the movement of the mouse and interrupt timing, the interrupt timing of the disk and finally the finish time of block device requests. But how can we judge the quality of the randomness? We can do it with **Entropy**, it measures how many bits of random numbers the system has. In Linux you can check the system's entropy in the file **/proc/sys/kernel/random/entropy_avail**. We can check it by doing the following command:

```
$ cat /proc/sys/kernel/random/entropy_avail
```

This command returns the number of bytes of entropy available for random number generation, we can see how many we have in the image bellow:

```
[03/05/24]seed@VM:~/.../Lab3$ cat /proc/sys/kernel/random/entropy_avail
1523
[03/05/24]seed@VM:~/.../Lab3$ cat /proc/sys/kernel/random/entropy_avail
1534
[03/05/24]seed@VM:~/.../Lab3$ cat /proc/sys/kernel/random/entropy_avail
1543
[03/05/24]seed@VM:~/.../Lab3$
[03/05/24]seed@VM:~/.../Lab3$ cat /proc/sys/kernel/random/entropy_avail
1586
[03/05/24]seed@VM:~/.../Lab3$
```

As we can see, we have arround 1500 bytes of entropy available. But, as stated before, the levels of entropy can be raised by using the keyboard, mouse and other methods. By executing the following command we can check how the number of bytes change by using the keyboard and mouse.

```
$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail
```

The command checks the changes in the **/proc/sys/kernel/random/entropy_avail** file every 0.1 seconds. We can see in the image bellow that after we move the mouse and press some keys the entropy value increases.

```
Every 0.1s: cat /proc/sys/kernel/random/entropy...  VM: Tue Mar  5 12:48:59 2024

2317
```

Through this expirement we can see how the system obtains a reliable source of entropy to generate resilient pseudo random numbers.

Task 4: Get Pseudo Random Numbers From /dev/random

Linux uses two devices to use the physical numbers collected by **/proc/sys/kernel/random/entropy_avail**. Those two being **/dev/random** and **/dev/urandom**. They behave diffently, as **/dev/random** blocks and does not generate any number when the **/proc/sys/kernel/random/entropy_avail** has 0 bytes of collected data. **/dev/random** will only resume operation when it finds that the physical data is enough for generating pseudo random numbers. We can see the behaviour of the **/dev/random/** by executing the command:

```
$ cat /dev/random | hexdump
```

Hexdump makes the output more understandable. When we run the output we see the following:

```
[03/05/24]seed@VM:~$ cat /dev/random | hexdump
0000000 4de8 6d5f a70b 2ba7 b14c 6883 6d89 4cca
0000010 f139 6ad8 3b54 f312 eb12 767b 9517 f628
0000020 504d 6b92 9e90 e0b5 7067 35da 3839 3751
0000030 2880 9d35 2db8 2a1c f9c3 a571 46b9 05f6
0000040 f2b7 87d7 529d e908 fe19 374f da66 b3ae
0000050 7c6d 2982 4f11 8ca2 1170 6b99 bd2e f185
0000060 2d19 d773 c046 dfcd 6db7 7b39 3e5c 5b0c
```

When we execute the watch on **cat /proc/sys/kernel/random/entropy_avail** and see the entropy increas, we also see that more lines are added to **/dev/random** until it goes to 0. When enough bytes are collected another entry is made on **/dev/random** and the entropy goes back to 0 again. We can see the blocking behaviour here, as entries are only made when there are enough bytes collected. **/dev/random** should be used in limited amounts because of this blocking behaviour. For example, if we use **/dev/random** in a server to generate random session keys we can cause a **DOS**. If the number of clients is greater than the number of lines in **/dev/random** has and can produce, the server will block waiting for the entropy and the clients are left waiting. This can then constitutes a **DOS**. A malicius attacker can also send many false client requests and can bring the server down to legitimate clients.

Task 5: Get Pseudo Random Numbers From /dev/urandom

Linux also provides another device for pseudo random numbers and that is the **/dev/urandom**. The difference to **/dev/random** is that it will not block when the entropy in **/proc/sys/kernel/random/entropy_avail** is 0. **/dev/urandom** uses the data pooled to create a seed for its random number generation. Let us see the behaviour of **/dev/urandom** by, once again, executing the following command:

```
$ cat /dev/urandom | hexdump
```

```
030ad90 73c2 6e32 2c5e 2e23 fb37 c204 ebbb f5e7
030ada0 ff77 b192 def6 8be0 8ab3 34fa 10e9 c0ff
030adb0 2064 9b03 dc2d a4a9 94e8 409b e495 8f54
030adc0 78f9 b3f9 38ef 83af ef37 a400 c4e8 55f1
030add0 634d cffd 39bb 37e2 2c49 dc1d c244 2c34
030ade0 4b38 1355 6b00 cd2b c17f cfbf d246 3753
030adf0 37bd 55b6 2111 674f 9494 1614 a4d1 7fd8
030ae00 30c4 32ac c160 1721 866c 7c6b 406b 3ebe
030ae10 c5fe 1a2b cb70 f743 8bc3 13cb 5fcb 72be
030ae20 dd57 f885 b5e8 a4de 4cb8 f325 05c9 6655
030ae30 37af e46a 1fa6 5d15 cfbe 1109 fd89 15f3
030ae40 8692 b1cf a2b1 c749 f253 5a5a 7b4e 454d
030ae50 e14c c54c ff0f 761d 45c0 5df9 09f2 27e1
030ae60 92e6 c3a9 5f58 6189 f7b5 57f3 70a6 28fa
030ae70 b519 530b 8365 21c4 c83c ed77 712f 44a0
030ae80 9033 b5bd 321b a1a1 8d33 477a 58dc 1ffd
030ae90 47a0 c05a d1ed 189e 0b4d f0aa 52bb b1cc
030aea0 3072 f3c3 70bd af3e c839 f424 9bab d7e6
030aeb0 7d59 bc40 e8d8 4436 e039 2ba3 3d30 dd59
030aec0 e598 4b6e 5479 b798 a979 b2a0 4bac 0822
030aed0 067c a5d8 d21f f6fe 4248 8622 2c5a 69b3
030aee0 438a 8219 7e22 3445 f29b b925 1633 c5a3
030aef0 7d05 a235 e6e2 a259 7064 bbb1 82d2 490b
```

Immediatly we see that it generates numbers non-stop, as it is non blocking, when the entropy runs out it starts using the pool instead. But are the results any good? We can check their quality using a tool called **ENT**, it analyses the numbers generated in a miriad of tests. Let's test 1 MB of the pseudo numbers generated by **/dev/urandom**. To test the 1 MB we will use the following commands:

```
$ head -c 1M /dev/urandom > output.bin
$ ent output.bin
```

The first command generates 1 MB of data and saves it to **output.bin**. The second tests it using **ENT**. The following image shows the results:

```
[03/05/24]seed@VM:~$ head -c 1M /dev/urandom > output.bin
[03/05/24]seed@VM:~$ ent output.bin
Entropy = 7.999832 bits per byte.

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

Chi square distribution for 1048576 samples is 244.91, and randomly
would exceed this value 66.39 percent of the times.

Arithmetic mean value of data bytes is 127.4410 (127.5 = random).
Monte Carlo value for Pi is 3.153156865 (error 0.37 percent).
Serial correlation coefficient is -0.001467 (totally uncorrelated = 0.0).
[03/05/24]seed@VM:~$
```

The test shows that the results are quite good! They are not totally random but are very close as we can see in the **arithmetic mean value**, **monte carlo** and **serial correlation coeficient**. They are close to the best value possible. In theory, **/dev/random** is more secure as it gets the best results but, for practical applications, we use **/dev/urandom** as the blocking penalty is very costly because it causes **Denial of Service**. So, using what we learned, let's modify the program that generates **Encryption Keys** using **/dev/urandom**. The following is the modified code:

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

#define KEYSIZE 32

void main()
{
    int i;
    unsigned char* key = (unsigned char *) malloc(sizeof(unsigned char) *
KEYSIZE);
    FILE* random = fopen("/dev/urandom", "r");

    fread(key, sizeof(unsigned char) * KEYSIZE,1 , random);
    fclose(random);

    for (i = 0; i < KEYSIZE; i++) {
        printf(".2x", (unsigned char) key[i]);
    }

    printf("\n");
}
```

The script uses **/dev/urandom** to generate a 256-bit Encryption Key. It accesses the device and reads 32 bytes of data. This way the **Encryption key** is resilient to bruteforce attacks of the type used before due to using a pseudo random number and not a predictable seed. We can see the key generated in the following image:

```
[03/05/24]seed@VM:~/.../Lab3$ sudo ./task1
754b7054d640d15da701e8ac250ee771d4626e40d007b88b7c9c685845f3c27b
[03/05/24]seed@VM:~/.../Lab3$ sudo ./task1
0c2c407a819d6a9a6158579db869d57199d1942b667dfce40130ad3a2ea3358c
```

Authored by:

```
Eduardo Ramos, up201906732
Leandro Silva, up202008061
Luís Paiva, up202006094
```