Pseudo Random Number Generation Lab

Random number generation is crucial in security software, especially for encryption keys. This lab focuses on secure methods, addressing common mistakes and introducing special device files like `/dev/random` and `/dev/urandom`.

Task 1: Generate Encryption Key in a Wrong Way

With the provided code, we found that the `srand(time(NULL))` instruction seeds the pseudo-random number generator with the current time, ensuring that each time the program runs, a different sequence of random numbers is generated.

```
#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");
}
```

However, if the previous instruction is commented out, the random number generator remains unseeded. Consequently, the sequence of random numbers generated by `rand()` remains constant across multiple executions of the program.

Then, there is a crucial role of `srand()` and `time()` that introduces randomness and variability into programs that depend on pseudo-random number generation.

Task 2: Guessing the Key

According to the text, the key of the file encryption was generated with a code similar to the previous one between `2018-04-17 21:08:49` and `2018-04-17 23:08:49`. The first step was extracted all the seeds from these values, using the `date` command to print out the number of seconds between a specified time and the Epoch:

```
$ date -d "2018-04-17 21:08:49" +%s # 1524013729
$ date -d "2018-04-17 23:08:49" +%s # 1524020929
```

And then we generated all possible keys within that range:

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

#define KEYSIZE 16

void main() {

    int i;
    char key[KEYSIZE];

    for (time_t t = 1524013729 ; t < 1524020929 ; t++) {

        srand(t);

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

We stored everything in a file for easier manipulation:

```
$ gcc -o task2.c task2
$ ./task2 > keys.txt
```

The text indicates the use of the `aes-128-cbc` algorithm to encypher the desired file. Next, we proceed to perform brute-force with the previously calculated seeds, as we already knew the values of the Initial Vector (IV), the plaintext, and the ciphertext:

```
from Crypto.Cipher import AES

data = bytearray.fromhex('255044462d312e350a25d0d4c5d80a34')
ciphertext = bytearray.fromhex('d06bf9d0dab8e8ef880660d2af65aa82')
iv = bytearray.fromhex('09080706050403020100A2B2C2D2E2F2')

def bruteforce(allKeys):
    for line in allKeys:
        key = key.rstrip('\n')
        hexvalue = bytearray.fromhex(key)
        cipher = AES.new(key = hexvalue, mode=AES.MODE_CBC, iv = iv)
        guess = cipher.encrypt(data)
        if guess == ciphertext:
            print(f"The key is: {key}")
            return
```

```
    print("Key not found")

def main():

    keys = []
    with open('keys.txt', 'r') as file:
        keys = file.readlines()
        file.close()

    bruteforce(keys)

main()
```

With this, we discover the value of the key used to encrypt the document, highlighting that the use of time in order to get randomenss does not guarantee much security:

```
$ python3 task2.py
> The key is: 95fa2030e73ed3f8da761b4eb805dfd7
```

Task 3: Measure the Entropy of Kernel

In this task, we were asked to check the variation in entropy of the kernel according to various actions we can perform based on the physical components of the system.

We ran the following command in a distinct terminal in order to check how much entropy the kernel had and to monitor the changes:

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

At the time, the value observed of entropy was around 3250, but it could be in any state depending on wether the system was using that entropy, the importance was on how much this value changes, and we concluded that the following actions listed in ascending order (from the one that causes slower changes to the one that causes faster changes) were relevant:

1. Time (derived from interrupts);
2. Network/disk activity;
3. Mouse movement;
4. Mouse click;
5. Keyboard press.

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

Just like in the previous task, we used the `watch` command to observe the system's entropy, but this time using `/dev/random` as the randomness source. This source uses physical factors as random seeds, but every time a random number is generated by this device, the entropy of the randomness pool will be decreased.

```
```

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

After some monitoring, we noticed that the entropy values never exceeded the value 50, and the device always blocked whenever reached zero, waiting for some physical entropy source such as a mouse movement. If we keep moving the mouse, the program is able to output random numbers more quickly.

If a server uses `/dev/random` to generate the random session key with a client, we could launch a DOS (Deniel Of Service) on the server by repeatedly requesting connections. This continuous connection requests will consume the available source entropy, causing the server's random number generator to become blocked, thus preventing it from fulfilling legitimate requests.

Task 5: Get Random Numbers from /dev/urandom

Exploring /dev/urandom

Firstly, they asked us to run the following command:

```
$ cat /dev/urandom | hexdumps
```

The objective was to try to notice any variation while actively moving the mouse versus keeping it stationary.

The outcome was that the `/dev/urandom` consistently generates random numbers, never blocking, so it's not possible to see visually any difference. However, the generated numbers surely have higher levels of randomization when we are moving the mouse.

Quality of /dev/urandom

The second question is to analyze the quality of the random numbers generated.

For that, we generated 1MB of random numbers with the command:

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

And then used the command `ent` on the generated file that applies various tests to the sequences of bytes generated and reports the results.

Below is the output from the command:

```
Entropy = 7.999819 bits per byte.

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

Chi square distribution for 1048576 samples is 262.74, and randomly
would exceed this value 35.61 percent of the times.
```

```
Arithmetic mean value of data bytes is 127.5738 (127.5 = random).
Monte Carlo value for Pi is 3.144688205 (error 0.10 percent).
Serial correlation coefficient is 0.000455 (totally uncorrelated = 0.0).
```

We can conclude that the bytes have high randomness quality, all metrics above show good results, the only one that could be questioned is the Chi square, however, when reading the man manual of the `ent` command we can see that 35.61 is far from a suspect value.

```
If the percentage is greater than 99% or less than 1%, the sequence is almost
certainly not random. If the percentage is between 99% and 95% or between 1% and
5%, the  sequence is suspect. Percentages between 90% and 95% and 5% and 10%
indicate the sequence is "almost suspect".
```

Final task

The final task was to modify the code snippet they provided in order to generate a 256-bit encryption key:

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

#define LEN 32 // 256 bits

int main(){
    unsigned char* key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);
    FILE* random = fopen("/dev/urandom", "r");
    fread(key, sizeof(unsigned char)* LEN, 1, random);
    fclose(random);

    for (int i = 0; i < LEN; i++)
        printf("%.2x", key[i]);
    printf("\n");
    return 0;
}
```

```
[03/07/24]seed@VM:~/PseudoRandomNumberGenerationLab$ cat gen_key.c
#include <stdio.h>
#include <stdlib.h>

#define LEN 32 // 256 bits

int main(){
    unsigned char* key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);
    FILE* random = fopen("/dev/urandom", "r");
    fread(key, sizeof(unsigned char)* LEN, 1, random);
    fclose(random);

    for (int i = 0; i < LEN; i++)
        printf("%.2x", key[i]);
    printf("\n");
    return 0;
}
[03/07/24]seed@VM:~/PseudoRandomNumberGenerationLab$ gcc gen_key.c -o gen_key
[03/07/24]seed@VM:~/PseudoRandomNumberGenerationLab$ gen_key
abf425ba9db3601461a7f335af58eddbd023ac5bf72fd3c64d9b6bee606ce024
[03/07/24]seed@VM:~/PseudoRandomNumberGenerationLab$
```

Author

G1:

- Alexandre Nunes (up202005358)
- Fábio Sá (up202007658)
- Inês Gaspar (up202007210)