C3: Pseudo Random Number Generation
due: 10/14/2022

1 Overview

Generating random numbers is a quite common task in security software. In many cases, encryption keys are not provided by users, but are instead generated inside the software. Their randomness is extremely important; otherwise, attackers can predict the encryption key, and thus defeat the purpose of encryption. Many developers know how to generate random numbers, but they may be bad for encryption keys. Developers need to know how to generate secure random numbers, or they will make mistakes. Similar mistakes have been made in some well-known products, including Netscape and Kerberos.

Through this assignment, you will learn why the typical random number generation method is not appropriate for generating secrets, such as encryption keys. You will further learn a standard way to generate pseudo random numbers that are good for security purposes.

2 Preliminaries

1. Install Docker Desktop
2. Open docker, click on the gear button on the top right-hand side, select `Settings`, then `Resources > File Sharing`, and add a newly created folder where you will keep all your files to run in Linux (this folder will be shared between your computer and the Linux docker container).
3. Open VSCode and press `Command-Shift-P` to open the `Command Palette`. Type and select the `Remote-Containers: Open Folder in Container` and open the folder created and linked in **Step 2** in a `Ubuntu 22` container.
4. In the terminal inside VSCode, type `sudo apt-get install gcc` to install `gcc` and,
5. `sudo apt-get install ent` to install the `ent` tool described and used later in this assignment.
6. To compile and run the provided c-code, type in the terminal `gcc program.c -o program` to compile and `./program` to run.

For more details refer to this VSCode article.

3 Tasks

3.1 Task 0: LFSR attack

We want to perform an attack on another LFSR-based stream cipher. In order to process letters, each of the 26 uppercase letters and the numbers 0, 1, 2, 3, 4, 5 are represented by a 5-bit vector according to the following mapping:

$$\begin{gathered} \text{A - 0} = 000002 \\ \ldots \\ \text{Z - 25} = 110012 \\ \text{0 - 26} = 110102 \\ \ldots \\ \text{5 - 31} = 111112 \end{gathered}$$

We happen to know the following facts about the system:
The degree of the LFSR is $\mathbf{m = 6}$.
Every message starts with the header **WPI**.
We observe now on the channel the following message (the fourth letter is a zero):

```
j5a0edj2b
```

1. What is the initialization vector?
2. What are the feedback coefficients of the LFSR?
3. Write a program in your favorite programming language which generates the whole sequence, and find the whole plaintext.

4. Where does the thing after WPI live?

5. What type of attack did we perform?

3.2 Task 1: Generate Encryption Key in a Wrong Way

To generate good pseudo random numbers, we need to start with something that is random; otherwise, the outcome will be quite predictable. The following program uses the current time as a seed for the pseudo random number generator.

Listing 1: Generating a 128-bit encryption key

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  #define KEYSIZE 16
6
7  void main()
8  {
9    int i;
10   char key[KEYSIZE];
11
12   printf("%lld\n", (long long) time(NULL));
13   srand (time(NULL));
14
15   for (i = 0; i< KEYSIZE; i++){
16      key[i] = rand()%256;
17      printf("%.2x", (unsigned char)key[i]);
18   }
19   printf("\n");
20 }
```

The library function `time()` returns the time as the number of seconds since the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`. Run the code above, and describe your observations. Then, comment out **Line 13**, run the program again, and describe your observations. Use the observations in both cases to explain the purpose of the `srand()` and `time()` functions in the code.

3.3 Task 2: Guessing the Key

On April 17, 2018, Alice finished her tax return, and she saved the return (a PDF file) on her disk. To protect the file, she encrypted the PDF file using a key generated from the program described in Task 1. She wrote down the key in a notebook, which is securely stored in a safe. A few month later, Bob broke into her computer and gets a copy of the encrypted tax return. Since Alice is CEO of a big company, this file is very valuable.

Bob cannot get the encryption key, but by looking around Alice's computer, he saw the key-generation program, and suspected that Alice's encryption key may be generated by the program. He also noticed the timestamp of the encrypted file, which is `"2018-04-17 23:08:49"`. He guessed that the key may be generated within a two-hour window before the file was created.

Since the file is a PDF file, which has a header. The beginning part of the header is always the version number. Around the time when the file was created, PDF-1.5 was the most common version, i.e., the header starts with `%PDF-1.5`, which is 8 bytes of data. The next 8 bytes of the data are quite easy to predict as well. Therefore, Bob easily got the first 16 bytes of the plaintext. Based on the meta data of the encrypted file, he knows that the file is encrypted using `aes-128-cbc`. Since AES is a 128-bit cipher, the 16-byte plaintext consists of one block of plaintext, so Bob knows a block of plaintext and its matching ciphertext. Moreover, Bob also knows the Initial Vector (IV) from the encrypted file (IV is never encrypted). Here is what Bob knows:

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

Your job is to help Bob find out Alice's encryption key, so you can decrypt the entire document. You should write a program to try all the possible keys. If the key was generated correctly, this task will not be possible. However, since Alice used `time()` to seed her random number generator, you should be able to find out her key easily. You can use the `date` command to print out the number of seconds between a specified time and the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`. See the following example.

```
$ date -d "2018-04-15 15:00:00" +%s
1523818800
```

3.4 Task 3: Measure the Entropy of Kernel

In the virtual world, it is difficult to create randomness, i.e., software alone is hard to create random numbers. Most systems resort to the physical world to gain the randomness. `Linux` gains the randomness from the following 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 first two are quite straightforward to understand: the first one uses the timing between key presses; the second one uses mouse movement and interrupt timing; the third one gathers random numbers using the interrupt timing. Of course, not all interrupts are good sources of randomness. For example, the timer interrupt is not a good choice, because it is predictable. However, disk interrupts are a better measure. The last one measures the finishing time of block device requests.

The randomness is measured using *entropy*, which is different from the meaning of entropy in the information theory. Here, it simply means how many bits of random numbers the system currently has. You can find out how much entropy the kernel has at the current moment using the following command.

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

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

`Linux` stores the random data collected from the physical resources into a random pool, and then uses two devices to turn the randomness into pseudo random numbers. These two devices are `/dev/random` and `/dev/urandom`. They have different behaviors. The `/dev/random` device is a blocking device. Namely, every time a random number is given out by this device, the entropy of the randomness pool will be decreased. When the entropy reaches zero, `/dev/random` will block, until it gains enough randomness.

Let us design an experiment to observe the behavior of the `/dev/random` device. We will use the `cat` command to keep reading pseudo random numbers from `/dev/random`. We pipe the output to `hexdump` for nice printing.

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

Please run the above command and at the same time use the `watch` command to monitor the entropy. What happens if you do not move your mouse or type anything. Then, randomly move your mouse and see whether you can observe any difference. Please describe and explain your observations.

3.6 Task 5: Get Random Numbers from `/dev/urandom`

`Linux` provides another way to access the random pool via the `/dev/urandom` device, except that this device will not block. Both `/dev/random` and `/dev/urandom` use the random data from the pool to generate pseudo random numbers. When the entropy is not sufficient, `/dev/random` will pause, while `/dev/urandom` will keep generating new numbers. Think of the data in the pool as the "seed", and as we know, we can use a seed to generate as many pseudo random numbers as we want.

Let us see the behavior of `/dev/urandom`. We again use `cat` to get pseudo random numbers from this device. Please run the following command, and the describe whether moving the mouse has any effect on the outcome.

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

Let us measure the quality of the random number. We can use a tool called `ent`, which has already been installed in our VM. According to its manual, "`ent` applies various tests to sequences of bytes stored in files and reports the results of those tests. The program is useful for evaluating pseudo-random number generators for encryption and statistical sampling applications, compression algorithms, and other applications where the information density of a file is of interest". Let us first generate 1 MB of pseudo random number from `/dev/urandom` and save them in a file. Then we run `ent` on the file. Please describe your outcome, and analyze whether the quality of the random numbers is good or not.

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

Theoretically speaking, the `/dev/random` device is more secure, but in practice, there is not much difference, because the "seed" used by `/dev/urandom` is random and non-predictable (`/dev/urandom` does re-seed whenever new random data become available). A big problem of the blocking behavior of `/dev/random` is that blocking can lead to denial of service attacks. Therefore, it is recommended that we use `/dev/urandom` to get random numbers. To do that in our program, we just need to read directly from this device file. The following code snippet shows how.

Listing 2: Generating 128 random bits using `/dev/urandom`

```
#define LEN 16  // 128 bits

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);
```

Please modify the above code snippet to generate a 256-bit encryption key. Please compile and run your code; print out the numbers and include the screenshot in the report.

4 Submission

You need to submit a detailed report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credit.