Pseudorandom Number Generation

1 Overview

Random number generation is commonly required in cryptography. Many software developers know how to generate random numbers for non-security applications; however, a common pitfall is to use the same methods to generate random numbers for security purposes, which leads to security vulnerabilities.

In this assignment, we will learn why conventional pseudorandom number generation methods are not appropriate for generating secrets such as encryption keys. We will then see a secure method of generating pseudorandom numbers for cryptographic applications. We cover the following topics:

- Pseudorandom number generation
- Mistakes in random number generation
- Generating encryption key
- The `/dev/random` and `/dev/urandom` device files

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Submission

Please submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. Please answer the questions in detail, and explain any observations you make that are interesting or surprising. Please also explain any code that you write. Code submitted without any accompanying explanations will result in grade deductions.

3 Lab Tasks

3.1 Task 1: Generate an Encryption Key the Wrong Way

To generate good pseudorandom numbers, we need to start with something that is random; otherwise, the outcome will be predictable. The following program uses the current time as a seed for the pseudorandom number generator. The `time()` library function returns the time as the number of seconds since the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`.

Listing 1: ”Generating a 128-bit encryption key”

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

```
#define KEYSIZE 16

void main()
{
  int i;
  char key[KEYSIZE];
  time_t t = time(NULL);

  printf("%lld\n", (long long) t);
  srand (t);       ①

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

Complete the following tasks:

1. Run the code above and describe your observations.
2. Then, comment out ①, run the program again, and describe your observations.
3. Use the observations above to explain **how** and **why** both `srand()` and `time()` are used.

3.2 Task 2: Guess the Key

On January, 15, 2024, Alice finally concluded her years-long research to develop an algorithm that outputs the biomechanical steps for the funniest silly walk that is physiologically feasible for a human to execute. She believes that her research will revolutionize the field of comedic locomotion, but she must keep her work under wraps until she prepares her patent filing. Alice saved her work in a PDF file on her disk. To protect the PDF file, she encrypted it using a key generated from the program in Task 1. She wrote the key in a notebook, which is securely stored in a safe. A few months later, Bob broke into her computer and retrieved the encrypted PDF file.

Bob cannot find the encryption key, but by snooping around on 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 `"2025-01-13 23:45:49"`. 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, it begins with a standardized header. The start of the header is always the version number. Around the time that 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 easy to predict as well. Therefore, Bob easily guesses the first 16 bytes of the plaintext. By looking at Alice's command history, he figures out that the file is encrypted using `aes-128-cbc`[1]. Since AES is a 128-bit block cipher, the 16-byte plaintext consists of one block of plaintext, so Bob knows a single block of plaintext and its matching ciphertext. Moreover, Bob also knows the Initial Vector (IV) from the encrypted file (the IV is stored in plaintext). Here is what Bob knows:

[1]For this assignment, assume that no padding is used. We will cover block ciphers and padding in class soon.

```
Plaintext: 2550 4446 2d31 2e35 0a25 bff7 a2fe 0a33
Ciphertext: bd51 16ad d7a7 8849 8ea5 263a b96c aa62
IV: 6f6d 6567 616c 756c 3132 3334 3536 3738
```

Your task is to help Bob find out Alice's encryption key, as follows:

1. Write a program to try all the possible keys. Exploit the fact that Alice used an insecure key generation method (otherwise this task would not be feasible).
2. **Explain** the strategy that your program follows and highlight the important steps in the process.

You may 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)` as follows.

```
$ date -d "2025-01-15 21:08:49" +%s
1736993329
```

> **Hint:** Be mindful of how time zone configurations might affect your results.

3.3 Task 3: Measure the Entropy of Kernel

Extracting randomness from software alone is difficult. Most systems extract randomness by taking measurements from the "real world". The following are functions used by the Linux kernel to extract randomness from 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 function uses timing information between key presses; the second uses mouse movement and interrupt timing; the third uses interrupt timing. Not all interrupts are good sources of randomness, e.g., the timer interrupt is unsuitable since it is predictable, whereas disk interrupts are more suitable since disk activity is less predictable. The fourth function measures the finishing time of block device requests (block devices typically represent storage devices).

> **Note:** In 2020, there was a major change to the Linux kernel's random number generation interface. The rest of this assignment thus requires a pre-5.6 version of the Linux kernel. The SEED VM comes with Linux 5.4, which satisfies this requirement.

Randomness is measured using *entropy*, which in simple terms represents 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
```

Let us monitor the change of the entropy by running the above command via `watch`, which executes a program periodically, showing the output in fullscreen. The following command runs the `cat` program every `0.1` seconds.

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

Please run the above command, and:

1. While it is running, move and click your mouse, type on your keyboard, or visit a website. What do you observe?
2. Report which activities increase the entropy more significantly.
3. Experiment further and report at least one more activity that you observe that has an impact on the reported entropy.

3.4 Task 4: Get Pseudorandom Numbers from `/dev/random`

Linux saves the random data collected from physical resources into a random pool, and then uses two virtual devices to turn the randomness into pseudorandom 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 output 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 repeatedly read pseudorandom numbers from `/dev/random`. We pipe the output to `hexdump` to "pretty print" it.

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

Please run the above command and at the same time use the `watch` command to monitor the entropy.

1. What happens if you do not move your mouse or type anything?
2. Share any differences you observe when you move your mouse around.
3. Suppose a server uses `/dev/random` to generate cryptographic keys each time it communicates with a client. Describe how you can launch a Denial-Of-Service (DoS) attack on such a server.

3.5 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 pseudorandom 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 pseudorandom numbers as we want.

Let us see the behavior of `/dev/urandom`. We again use `cat` to read pseudorandom numbers from this device, as follows:

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

1. Does moving or not moving your mouse have any effect when running the command above? Why or why not?

3.6 Task 6: Measuring Random Number Quality

We can use a tool called `ent`, which has already been installed in our VM, to measure random number quality. According to its documentation, "`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 pseudorandom number from `/dev/urandom` and save them in a file. Then, run `ent` on the file as follows.

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

Run `ent` on **three** other types of files. Share all of your results and answer the following questions:

1. How can we interpret the output provided by the `ent` tool, including all the statistical tests provided?
2. What conclusions can you draw from `ent` about the comparative "randomness" of the different types of data that you examined? Link your conclusions with the information provided by `ent`.

3.7 Task 7: Generating Cryptographic Keys

Theoretically, `/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 becomes available), and the pseudorandom number generator that is used is considered suitable for cryptographic purposes. It is usually recommended to use `/dev/urandom` to avoid blocking (and thus DoS attacks). To do that in our original program, we just need to read directly from this device file. The following code snippet shows how.

```
#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.