Pseudo Random Number Generation Lab

Modified 2025 for COMP 435: Computer Security Concepts at UNC

Overview

Generating random numbers is a common task in security-focused software. In many cases, encryption keys are not provided by users, but are instead generated inside the software. The keys' randomness is important; otherwise, attackers may be able to predict the encryption key, and thus defeat the encryption. Many developers know how to generate random numbers from their experiences in software development (e.g., for Monte Carlo simulation), and may try to use similar methods to generate random numbers for security purposes. Unfortunately, a sequence of random numbers that is good enough for Monte Carlo simulation may not be good enough for producing encryption keys. Mistakes like this have been made in some well-known products, including Netscape and Kerberos.

In this lab, students will learn why the typical random number generation method is not appropriate for generating cryptographically secure encryption keys. They will further learn a standard way to generate pseudo random numbers that is appropriate for security purposes. This lab covers the following topics:

- Pseudo random number generation
- Mistakes in random number generation
- Generating encryption keys
- The /dev/random and /dev/urandom device files

Generative AI

The use of **ANY** Generative AI tool such as ChatGPT, Microsoft Copilot, etc. is strictly prohibited.

Lab environment

We'll be using VMs provided by the department. Instructions for logging into your VM are here: https://help.cs.unc.edu/en/blog/classvm. The VM's password is `dees`. Use the VM for your lab; do not use your home machine. Note that you must be either on campus or logging on through a

VPN. Instructions for connecting to the campus VPN for Mac, Windows, or Linux are available here.

Once you have logged into your VM, you can clone the starter code for the assignment here. Before starting on the assignment you are encouraged to read through the README within the starter code for instructions about the setup of the starter code as well as how to compile and run each task.

Submission

You will submit your finalized code to Gradescope using the assignment “Lab 1 Code”. This assignment has an autograder, and will autograde the code you write / change for tasks 1, 2, and 5 (the other tasks do not involve writing / modifying code). All 5 tasks for this assignment also have corresponding written questions in the gradescope assignment “Lab 1 Written Questions” that you should answer as you complete the lab.

The screenshots that you submit to the Gradescope assignment must include a unique identifier. The easiest way to do this is to take a screenshot that encapsulates the VM interface itself (which should already happen before cropping), as it includes your onyen/name on the left and top-right.

Lab Tasks

Task 1: Generate an Encryption Key the Wrong Way

To generate good pseudo random numbers, we need to start with something that is random; otherwise, the outcome will be quite predictable. After accepting the assignment from Github Classroom, clone the starter code to your VM. Next open `task1.c` to view the starter code for this task. `task1.c` uses the current time as a seed for the pseudo random number generator.

Note that the main function for the starter code is located in the file `lab1.c` within the starter code. Within `lab1.c`, you will find several lines of code that you can use to test the function that is in `task1.c`, along with commented out code that you may use to test your solutions to task 2 and task5 later in the lab. Within `lab1.c`, the starter code uses the library function `time()`. This function returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

Run the starter code multiple times, and describe your observations. Then, comment out Line A, run the program multiple times again, and describe your observations. Use the observations in both cases to explain the purpose of the `srand()` and `time()` functions in the code.

As a refresher, you can compile the starter code for this lab using the `make` command:
`$ make task1`
You can then run the compiled code from the terminal
`./task1`

If you get a permission denied error, update the file permissions of the executable:
`$ chmod +x ./task1`

After completing the instructions below for answering the written questions on Gradescope, make sure that the line marked `// Line A` within task1.c is **uncommented**. You may need this function in future tasks.

For this task, make sure to answer the question on the Gradescope assignment Lab 1 Written Questions that are under the section “Task 1”.

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 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 months later, Bob broke into her computer and got a copy of the encrypted tax return. Since Alice is the CEO of a large company, this file is very valuable.

Bob could not 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 have been generated by the program. He also noticed the timestamp of the encrypted file, which was “2018-04-17 23:08:49". He guessed that the key may have been generated within a two-hour window before the file was created.

PDF files have a header and 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. In other words, the header of the tax return likely 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 guessed the first 16 bytes of the plaintext. Based on the metadata of the encrypted file, he knew that the file was encrypted using aes-128-cbc. Since AES is a 128-bit cipher, the 16-byte plaintext consists of one block of plaintext, so Bob had a block of plaintext and its matching ciphertext. Moreover, Bob also had the Initial Vector (IV) from the encrypted file (the IV is never encrypted). Here is what Bob knew:

```
Plaintext: 35100a4624312e35025d0d4c5d35a30a
Ciphertext: 9938580ddce5f778daf6a75c7e627f65
IV: 0908070605040302010 0A2B2C2D2E2F2
```

Your job is to help Bob figure 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 had been generated correctly, this task would not be possible. However, since Alice used `time()` to seed her random number generator, you should be able to figure out her key.

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
```

In the starter code, open the file `task2.c`. Within the file, you will see the starter code for this task. Modify the `findkey()` function so that it takes as input the plaintext, ciphertext, and IV, and

outputs the correct key once it is found. Your program should run with the following syntax and output the correct key:

```
$  ./task2 <plaintext> <ciphertext> <iv>
```

For example, we would expect to run your program like this:

```
$  ./task2 35100a4624312e35025d0d4c5d35a30a 9938580ddce5f778daf6a75c7e627f65
090807060504030201 00A2B2C2D2E2F2
```

To complete this task you will need to install the tiny-AES-C library on your VM:

```
$ git clone https://github.com/kokke/tiny-AES-c && cd tiny-AES-c
$ make
$ cd ..
```

You should also visit the tiny-AES-c library's repo (https://github.com/kokke/tiny-AES-c) and look at their README. The documentation here will give you insight into the datatypes the plaintext, ciphertext, IV and key should have. This is specified in the first code block that has the function signatures.

Note

1. To compile it:
   ```
   $ make task2
   ```
2. To run it:
   ```
   $ ./task2 <plaintext> <ciphertext> <iv>
   ```

For this task, make sure to answer the question on the Gradescope assignment Lab 1 Written Questions that are under the section "Task 2".

Task 3: Investigating Randomness and /dev/urandom

Note: For this and the next task, ensure that you use the virtual machine rather than VSCode or Terminal SSH tools.

Background

In the digital world, it is difficult to create randomness, i.e., using software alone it is hard to create random numbers. Most systems resort to the physical world to gain randomness. Linux gains randomness from the following physical resources to seed a cryptographically secure pseudorandom number generator (CSPRNG):

```
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.

Randomness is measured using *entropy*, which is different from the meaning of entropy in 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
```

Linux stores the random data collected from the physical resources (e.g., interrupts, timing events) into a random pool, and then uses two devices to turn the randomness into pseudo random numbers.

Historically:

- `/dev/random` : would block if the entropy pool was empty. Only returned data when "enough" entropy had been collected. This meant every byte read was backed by "fresh" physical entropy, which deemed it more secure historically.
- `/dev/urandom`: never blocked, always returning random numbers from the CSPRNG seeded by the pool.

On modern kernels, both `/dev/random` and `/dev/urandom` now behave the same way. In practice, `/dev/urandom` is the recommended source for applications.

Let us try generating and printing random bytes from `/dev/urandom`. We use `cat` to get pseudo random numbers from this device. Please run the following command, and then observe 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 numbers from `/dev/urandom` and save them in a file. Then we run `ent` on the file. Observe 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
```

The random numbers provided by the Linux kernel are generated by a CSPRNG that is seeded with unpredictable data from the system's environment. In our program, we can obtain random values by reading directly from the device file. In the starter code for this assignment, open the file called task3.c, which contains sample code that generates a **128-bit encryption key**. Modify this code so that it generates a **256-bit encryption key** instead. You can test that your output is correct by uncommenting the code in lab1.c that is for task 3.

For this task, you will also need to answer the written questions for task 3 with the Lab 1 Written Questions assignment on Gradescope.

Code Submission (Gradescope Autograder)

Once you are confident that your code for this assignment is correct, you should upload the code to the gradescope assignment the lab1 code. After submitting, you will get feedback on the correctness of your solutions for task1, task2, and task3.