SEED Security Labs : Cryptography
Pseudo Random Number Generator
Lab / Project Report

Matthew Chin

August 2025

If you are reading this, this lab report and project are still In-Progress

1 Introduction

This project is a lab guided by *SEED Security Labs*, exploring pseudo-random number generation, and when/how it is appropriate for certain situations, such as encryption keys and secret generation.
Random number generation is fundamental in modern computing and is necessary in various situations like simulations, statistical modeling, and secure/private communications. This is a lab focused on increasing security, because in encryption we desire not just any sequence of random numbers (which may be adequate for situations like the Monte Carlo Simulation), we want some control over that sequence to actually be able to use it. Developers should know how to generate *secure* random numbers, avoiding mistakes like weak or predictable randomness in well-known products like *Netscape* and *Kerberos*.
This lab and project explores the differences between catch-all-type random number generation and cryptographically safe, secure random number generation. By exploring common mistakes and failed algorithms for random number generation, attempting key-recovery attacks, and analyzing entropy sources within a Linux OS (Ubuntu 20.4 VM), this entire process investigates the challenges of achieving **true randomness** in software. Experiments are also conducted using `/dev/random` and `/dev/urandom` to provide some insight into how certain operating systems manage entropy, the pros and cons of blocking or not blocking random sources, and their effects on security.
The objective of the lab is to build an understanding of how pseudo-random numbers are generated within software, why the insecure methods are inadequate, and how to properly use system-level randomness to strengthen cryptographic systems through hands-on experiments within a controlled Linux OS.

2 Pseudo-Random

2.1 Generating Incorrect Encryption Keys

To generate good pseudo-random numbers, we need to start with the opposite: bad pseudo-random number generation.

```
#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));                 //What difference does this line make?
    for (i = 0; i< KEYSIZE; i++){
    key[i] = rand()%256;
    printf("%.2x", (unsigned char)key[i]);
}
printf("\n");
}
```

2.1.1 Observations

The code above uses the library function `time()`, which returns the number of seconds from the *Epoch*, January 1, 1970. Therefore, if you run the program multiple times within the same second, you will get the same number because time()'s return value has a smallest unit of seconds.
Here, the current time is used as a `seed` for the random-number generation. When the marked line is commented out, the first line still prints the system time in seconds, but the random number generator is never reseeded, meaning the generator always starts from the same state and moves towards the same states. As a result, the key sequence will be identical if two are run at the same time (the same second). Although the timestamp will change throughout runs, the key values become predictable and no longer "truly random."

2.2 Guessing Lazy Encrypted Key-Gen

The story here is that Alice used the above insecure `keygen.c` program to generate her AES key after filing her taxes. Because she used the time(null) seed, there is a limited number of possible keys, so here the task is to write a script to try all of the seeds in the likely time window to find hers.

2.2.1 The Problem

We know from the lab description that the file was created on `2018-04-17 23:08:49`
Also, Alice's time window for her key generation was within two hours before that.
We know:
- Plain text:
255044462d312e350a25d0d4c5d80a34
- Cipher text:
d06bf9d0dab8e8ef880660d2af65aa82
- IV:
09080706050403020100A2B2C2D2E2F2

2.2.2 The Process

Remember that time(NULL) returns the seconds since the `Epoch`, which is **Jan 1, 1970**. We need to know all of the possible "seed" values within that 2-hour window before the file creation.
Alice created the file at `2018-04-17 23:08:49`, so we need to check all the values between 2018-04-17 23:08:49 and 2018-04-17 21:08:49. To do this, first run the following commands in the terminal:

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

To write the program, I will use Python which is easier for AES because of the `Crypto.Cipher` library. Here is what a general script should look like:

```
from Crypto.Cipher import AES
import time
import struct
import random

# Known values
plaintext = bytes.fromhex("255044462d312e350a25d0d4c5d80a34")
ciphertext = bytes.fromhex("d06bf9d0dab8e8ef880660d2af65aa82")
iv = bytes.fromhex("09080706050403020100A2B2C2D2E2F2")

# Range of times checked earlier in bash commands above.
start = 1524008929   # 2018-04-17 21:08:49
end   = 1524016129   # 2018-04-17 23:08:49

for seed in range(start, end+1):
    random.seed(seed)
    key = bytes([random.randint(0,255) for _ in range(16)])

    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted = cipher.decrypt(ciphertext)

    if decrypted == plaintext:
        print(f"Found! Seed = {seed}")
        print("Key =", key.hex())
        break
```

Although there is a text break because of the pages, I think the idea should get across

This becomes more difficult if you decide to write the script in C for multiple reasons. 1) You have to manually convert the hex string to bytes and return the number of bytes written for the 128-bit key. 2) In C, you have to manually size the arrays correctly, avoid out of bounds errors, overflows, just overall memory management. 3) Python is simply a higher level language so in this case it is easier to complete this task because it includes more complex and specialized libraries compared to C, which takes much more work or more convoluted strategies to complete the same task.

Essentially, Python is just more convenient for cryptography here because it includes less code and higher level language and libraries. However, this task specifically requires reproducing tasks that are trivial in C such as reproducing C's `rand()` stream used in `time(NULL)`.