ECE 411: Final Project Report

Author:

Rain D. Spann

UIN: 01060463

Project Due: April 28th , 2021, 11:00 PM

Honor Code:

ODU Honor Code must be adhered to. This means that homework assignments and design projects are to be the work of an individual student group only. Evidence such as identical results and/or wording of sections of a report, if strong enough, will be reported to the University Hearing Officer in charge of administering the ODU Honor Code. If the violation is deemed sufficient, a permanent record of this infraction will be placed on the student's official university transcript for the first offense! Subsequent offenses could result in dismissal from the university.

Abstract

The objective of this project was to use the Ubuntu software, my knowledge of virtual machines, and the information gained from research in order to learn more about Random numbers, Encryption Keys, and their importance in the field of Network Security. While learning about these topics, there were three primary goals: 1.) Generate Encryption Keys incorrectly using srand() and time() commands, 2.) Use encryption techniques to guess an encryption key from the lab I used as a learning basis, and 3.) Observe entropy as it is affected by physical interaction through the interface. As mentioned above, the first part of this assignment dealt with generating an encryption key incorrectly using the c programming language. The second part dealt with using given information (Cipher Text, Plain Text, IV, and a time stamp) to figure out the encryption key for a set of tax documents. The third portion of this project actually had a few sub-portions that highlighted different ways to observe entropy.

Table of Contents

List of Figures:

Introduction

The purpose of this project was to use the Ubuntu software, my knowledge of virtual machines, and the information gained from research in order to learn more about Random numbers, Encryption Keys, and their importance in the field of Network Security. The Ubuntu software would be used for the use of the Virtual Machine in SEED Labs. The Virtual Machines would be used so that I could use linux and the terminal more efficiently. Since SEED Labs and Ubuntu have python, c, and other various programming language compilers already installed this made the entire process much easier. The difference in using the virtual machine terminal as opposed to using the terminal on my PC is that one operates on linux and the other on windows. As we all know, linux is definitely the better of the two when it comes to file systems.

In order to complete the parts mentioned in the Abstract part of this report, the student would have to have some knowledge of python, c, and terminal controls. As mentioned before, this project required the use of several different commands to observe different ways in which random numbers are created within the Network Security Field. In the next section of this report, *Design Methodology*, I will discuss the steps that I took in the implementation of the programs, what I accomplished, and what I did not accomplish.

Objective:

The objective of this assignment was to further my understanding of random numbers, encryption keys, and how they are created in the Network Security Field. As a computer engineer, I have had a lot of experience in using random numbers for various purposes, but after learning more about encryption keys I wanted to know if the process for generating these random numbers or keys was the same or different. To answer my question, I used a lab as a reference for random numbers since it highlighted many details for me. As well as this lab reference, I used Ubuntu seed labs, python programming, and c-programming in order to compile/run several lines of code that would aid in my research of random numbers and their relation to Network Security.

Why Are Random Numbers Important?

Random numbers are very important in Network Security since they are used to create specific keys called *encryption keys*. These keys are used to decrypt important information such as tax documents, classified information, etc. Since they are used for a variety of purposes, creating encryption keys is a really common thing. Unlike many people may think, these encryption keys are not created by the user. They are instead generated inside the program/software that the user may be using at the time. I will be discussing the different ways in which they are generated by the system later on in this report. For now, I would like to put the importance of randomness into perspective. As mentioned before, since these keys are used to decrypt important data and other kinds of information, they are extremely important. For these keys to protect the user from being a victim of theft, the randomness is extremely important. If the randomness of the software is predictable then that puts the user at risk of having their property stolen by attackers.

Since randomness is of utmost importance, the traditional method of generating random numbers using methods such as Monte Carlo or Srand() have proven themselves to be inefficient

when it comes to randomness and security. As I move forward with this report, I will discuss Pseudo random number generation, mistakes made in generating random numbers (encryption keys), the correct way to generate random numbers (encryption keys), and also the */dev/random* and */dev/urandom* device files.

Part 1: *Generating Encryption Keys Incorrectly*

As mentioned before, one of the topics I want to discuss is how to generate encryption keys incorrectly. Using the lab as a main reference I was able to use a c-program to actually create random numbers. After I created the random numbers, I was then able to actually see why this method of creation is typically not used in the generation of encryption keys. Before I discuss why it is not used, I will first talk about the program, my observations, and then finalize this section with the reason why it is not commonly used. To generate encryption keys, I used the below c-program (one to the left) first. Results are shown to the right.

```c
#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");
}
```

```
[04/21/21]seed@VM:~$ gcc -o nocomment wrongkey.c
[04/21/21]seed@VM:~$ nocomment
1619018069
364ba546bb47b3406c1dd4cacd23b2b2
[04/21/21]seed@VM:~$ nocomment
1619018071
1c8283dc171f507e94229c418d476e0a
[04/21/21]seed@VM:~$ no comment
no: command not found
[04/21/21]seed@VM:~$ nocomment
1619018077
72b543faf0931dcacda7cb89cece589f
```

Image 1:

Generating keys(w/ *srand()*)

Image 2:

Generating keys (*Results)*

In the first image, I use the program to create random numbers or encryption keys every time the program is run in the terminal. This program works by using KEYSIZE as an int value, implementing it within the for loop, and then using an array in conjunction with the s*rand()* function. Everytime the number is generated, it is randomly generated, it is printed out in the terminal. In image 2, you can see how every time the program was run in the terminal a time stamp (in seconds) would be displayed to the user and afterwards the actual encryption key would be displayed. The time stamp is circled or marked in red and the encryption key is highlighted. Now, this looks like it would work fine, but actually it would not. With a simple program, an attacker could easily find a copy and run the program until they found the desired encryption key. In this next part, I will show how fickle the program is by taking out the *srand*() function.

```c
#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");
}
```

```
[04/21/21]seed@VM:~$ gcc -o comment wrongkey.c
[04/21/21]seed@VM:~$ comment
1619018134
67c6697351ff4aec29cdbaabf2fbe346
[04/21/21]seed@VM:~$ comment
1619018135
67c6697351ff4aec29cdbaabf2fbe346
[04/21/21]seed@VM:~$ comment
1619018139
67c6697351ff4aec29cdbaabf2fbe346
[04/21/21]seed@VM:~$
```

Image 3:

Generating keys(w/o *srand()*)

Image 4:

Generating keys (*Results)*

In this test case, you can see how when the *srand(time(NULL))* line of code is commented out, the program no longer generates random numbers. Instead the key is exactly the same every time it is run. This is not the desired result that we would want in any case of network security.

Part 2: *Guessing the Key!*

For the second part of this project, I challenged myself to actually find the encryption key of a set of tax files. This part involves a scenario in which a woman has just saved a PDF file on her disk and has created an encryption key to protect the file. The encryption key is said to have been generated from another program. The program itself is to be stolen and the thief wants to decrypt the file with the key. My job is to write a program that would find the encryption key using the Plaintext code: 255044462d312e350a25d0d4c5d80a34, Ciphertext code: d06bf9d0dab8e8ef880660d2af65aa82, and IV code: 09080706050403020100A2B2C2D2E2F2.

Once the encryption key is found, I will then be able to decrypt the entire document. This program would try all the possible keys and if the key was generated correctly, the encryption key would be found. Since the woman who encrypted the key used the program in Part 1 to actually generate the key then I would be able to find it. Besides the codes given, I was also given a time stamp in which the actual encrypted file was created: "2018-04-17 23:08:49". I was also given this command line (*$ date -d "2018-04-15 15:00:00" +%s*) so that I would be able to print the number of seconds between specified time and the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

For the first part of this part, I used the date command ($ date -d "2018-04-17 23:08:49" +%s) to find the timestamps in seconds for the encrypted file. The first time stamp was at 2018-04-17 23:08:49 and the second one was 2018-04-17 23:06:49. This gives me two

timestamps within a 2 hour period of time. Next, I took the time in seconds that I received from the code and used them in the program below.

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

#define KEYSIZE 16

void main()
{
        int i;
        char key[KEYSIZE];

        long long j;
        for (j = 1524020809 -60 * 60 * 2; j < 1524020809; j++){
                        srand(j);
                        for(i = 0; i< KEYSIZE; i++){
                                key[i] = rand()%256;
                                printf("%.2x", (unsigned char)key[i]);
                        }
                        //print("\n");
                }

}
```

Image 5: Program for Part 2

The program above takes the lower time and uses it in a program that generates a random number string. Because this string is so large, I used an additional line of code to place the string in a text document. This line of code is: "*./name of compiled c program > name of .txt doc.*". In this case, my compiled c program was called *guess* and the name of the text document was *guesskey.txt.* Once this line of code is run in the terminal, the text document is created in the directory of the c-program. Below is a screenshot of the text document that was created. It also shows part of the string that was generated from the c-program.

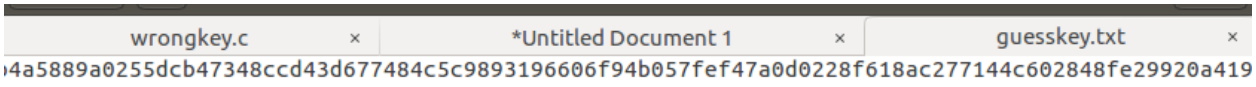

Image 6: Text Document

After the text document was created, I could then move on to the next part of part 2. This part consisted of using the text file that was generated in the previous step in conjunction with the other codes given. All of these used together would then be used in a for loop to generate the encryption key. The encryption key would be generated with the AES_CDC_MODE which encrypts and shows the key we are looking for. AES stands for Advanced Encryption Standard and is a symmetric book cipher standardized by NIST. It has a fixed data of 16 bytes. Another mode used is MODE_CBC which stands for "Cipher Block Chaining". It XOR's plain text with a given ciphertext to create an encryption key. Below is a screenshot of the program used for finding the encryption key. The results are shown in image 8. These results were not correct though. I have a screenshot of what the results are supposed to look like in image 9.

```python
from Crypto.Cipher import AES
data = bytearray.fromhex('255044462d312e350a25d0d4c5d80a34')
ciphertext = bytearray.fromhex('d06bf9d0dab8e8ef880660d2af65aa82')
iv = bytearray.fromhex('09080706050403020100A2B2C2D2E2F2')
with open('guesskey.txt') as f:
    keys = f.readlines()
for k in keys:
    k = k.rstrip('\n')
    key = bytearray.fromhex(k)
    AES = AES.new(key=key,mode=AES.MODE_CBC,iv=iv)
    guess = AES.encrypt(data)
    if guess == ciphertext:
        print("the key is : ", k)
        break
```

Image 7: Program to find the Encryption Key

```
[04/21/21]seed@VM:~$ gcc task2guess.c -o task2guess
[04/21/21]seed@VM:~$ gedit task2guess.c
[04/21/21]seed@VM:~$ gcc task2guess.c -o task2guess
[04/21/21]seed@VM:~$ ./task2guess > key.txt
[04/21/21]seed@VM:~$ python guesskey2.py
Traceback (most recent call last):
  File "guesskey2.py", line 19, in <module>
    cipher = AES.new(key=key,mode=AES.MODE_CBC,iv=iv)
  File "/usr/local/lib/python2.7/dist-packages/Crypto/Cipher/AES.py", line
95, in new
    return AESCipher(key, *args, **kwargs)
  File "/usr/local/lib/python2.7/dist-packages/Crypto/Cipher/AES.py", line
59, in __init__
    blockalgo.BlockAlgo.__init__(self, _AES, key, *args, **kwargs)
  File "/usr/local/lib/python2.7/dist-packages/Crypto/Cipher/blockalgo.py",
 line 141, in __init__
    self._cipher = factory.new(key, *args, **kwargs)
TypeError: argument 1 must be string or read-only buffer, not bytearray
[04/21/21]seed@VM:~$
```

Image 8: Compiling the guesskey file

incorrect results

```
$ python guess_key.py
the key is : 95fa2030e73ed3f8da761b4eb805dfd7
```

Image 9: Correct Results

Part 3: Entropy,Random Numbers, and Pseudo Random Numbers

I: Entropy and Random Numbers

For the third part of this project, I explored entropy and how it relates to random numbers in the world of Network Security. Entropy is the measure of randomness or diversity of a data generating function. It is very important for generating random encryption keys. After doing some research and also working with the lab, I learned that it is hard producing random numbers using regular programming techniques (like the one seen in task 1). Mainly it is hard to use these traditional methods because the random numbers that they generate do not suit the needs for efficiency that encryption keys require. It is this efficiency that makes an encryption key effective and worth having. Otherwise, you might as well use a regular password. To combat this obstacle, network security programmers created other random number generating functions to effectively create random numbers.

These functions are built into the actual system, unlike the program that we saw before. The main thing that I found really interesting about these functions is that they are created to react based on interaction from the physical world. For example, some functions actually track mouse movement and generate random numbers based on the actual movement. Another example can be seen from interaction tracked between the user and the keys they press while the program is actually encrypting the file. Below are some of the functions created:

1. Void add_keyboard_randomness(unsigned char scancode);
2. Void add_mouse_randomness(_u32 mouse_data);
3. Void add_interrupt_randomness(int irq);
4. Void add_blkdev_randomness(int major);

So in a sense, entropy is used to measure the randomness of each interaction! So where does the entropy actually come from? How is it measured? Well, everyone's system has entropy. To actually find out how much entropy your system has you can use this command in your terminal: *cat /proc/sys/kernel/random/entropy_avail.* To monitor the changes in entropy as you move around and interact with your system you can use this command in your terminal:

watch -n .1 cat /proc/sys/kernel/random/entropy_avail… In the presentation, I actually ran the second command and moved my mouse around, typed things, and even opened a browser just to show my fellow classmates how the entropy would change as movement/interaction was detected by the system.

II: **Pseudo Random Numbers from the Terminal**

We have discussed how to actually generate random numbers the wrong way using a c-program, we have actually used a program to find an encryption key created from the same program used in task 1, and we have discussed entropy and how network security uses it in conjunction with several kinds of randomness functions to generate random numbers based on user interaction. Now, I will discuss one more method that is commonly used to generate random numbers. Much like the Part I in this section, this is done using the terminal and terminal commands. This method is with Pseudo Random Numbers.

Pseudo Random Numbers can be generated using the commands ***/dev/random*** and ***/dev/urandom.*** As discussed before, random data is collected in the actual system from physical sources. This random data is then put into a random pool. Once the random data is put into a pool, the system uses the devices */dev/random* and */dev/urandom* to turn those random numbers into pseudo random numbers. The command ***/dev/random*** is known as a blocking device. This means that as a random number is generated the entropy of the system will decrease until it reaches zero. Once it reaches zero, the command will block until it gains enough randomness. The images below show the random numbers generated and also show the decreasing entropy below.

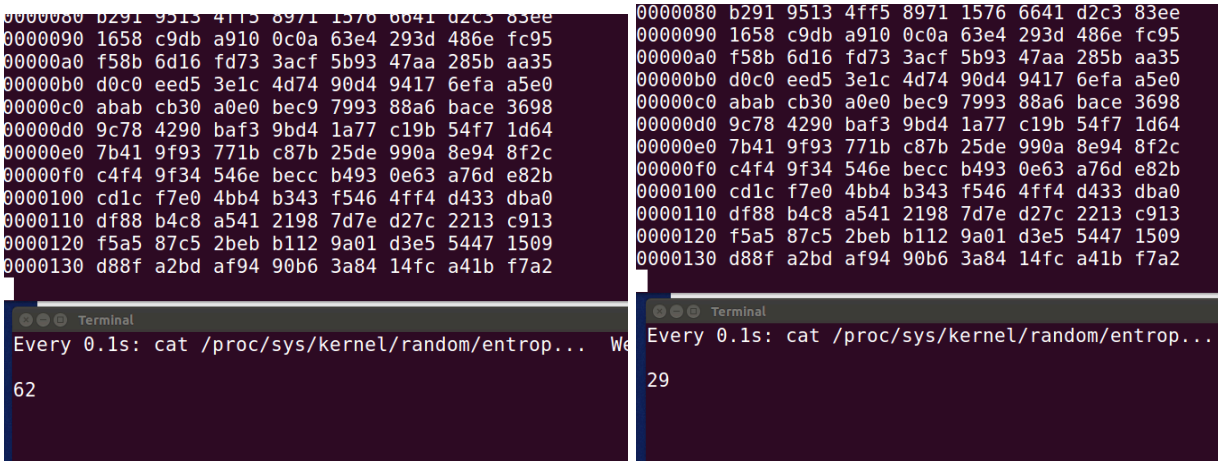

Image 10: /dev/random Results

That is not the only way to actually use the random numbers from the random pool though, we can also use the command ***/dev/urandom*** . Unlike the other command that was mentioned earlier, the urandom command will not pause once entropy is not available. Instead, it will keep generating random numbers.

Conclusion

In conclusion there are many ways to create random numbers for encryption key development. Unfortunately, all of these methods are not the most secure or efficient for high security, or even regular security, usage. Because of the difficulty that comes with developing random number keys with software, physical movement has been used to further assist software developers. Entropy, which is the measure of randomness or diversity of a data generation function, works with the physical movement such as typing, movement, etc. to generate random numbers. As well as entropy and functions that track movement and user interaction, the random number pool can also be accessed in order to generate effective encryption keys. These encryption keys can be accessed using the commands mentioned in Part 3, Task I & Task II.