Pseudo Random Number Generation Lab

In this lab, students will understand

- How to generate secure random numbers
- Why the typical random number generation method is not appropriate for generating secrets, such as encryption keys

These labs particularly states a standard way to generate pseudo random numbers that are good for security purposes

Table of Contents

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 (e.g. for Monte Carlo simulation) from their prior experiences, so they use the similar methods to generate the random numbers for security purpose. Unfortunately, a sequence of random numbers may be good for Monte Carlo simulation, 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.

Lab environment: This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website. https://seedsecuritylabs.org/lab_env.html. Download the June 2019 version of ubuntu 16.04

Lab Tasks

Task 1: Generate Encryption Key in a Wrong Way

In this task we run the code provided in the lab description which generates 128 bit encryption key based on the random value (which is time in this case). Below is the code provided in the lab description

Step 1: Please compile and run the following program and describe your observation.

```
/*  task1.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");
}
```

Compile the code and run it 2-3 times

Commands:
$gcc task1.c -o task1
$./task1
$./task1
$./task1

Give your observation with screen shot.

Step 2: Now comment out the statement srand (time(NULL)) and then compile and run the program 2-3 times.

Give your observation with screen shot.

Task 2: Guessing the Key

In this task we as Bob need to break the encryption done by Alice by guessing the key she used for encryption. Since Alice did not use safe and efficient way to generate the key we assume that Bob can guess the key. We do require some reconnaissance before we can generate the key, results of which are already provided in lab description.

Plaintext: 255044462d312e350a25d0d4c5d80a34
Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
IV: 09080706050403020100A2B2C2D2E2F2

Timestamp: 2018-04-17 23:08:49
Time range: 2018-04-17 21:08:49 - 2018-04-17 23:08:49

After looking at the source code of the key generation program used by Alice we can see that the time() function is used as seed value given to the srand() function. We also know that the time() returns the time elapsed in secs. Therefore for the malicious key generation program we need to give the same seed value as Alice gave while generating the key. From our reconnaissance we know the time range in which Alice generated the key, so we can brute force all the keys possible during the time range.

Step 1: We can use the date command as mentioned in the lab description to get the time elapsed in seconds.

Commands:

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

<value1>
$date -d "2018-04-17 23:08:49" +%s
<value2>

Give your observation with screen shot. Make a note of value1 and value 2.

Step 2: Modify the program used by Alice to generate all the keys in this time range (value1 – value2). Write all the keys to the file called keys.txt
The program used for key generation is given below:

```
/*    task2.c   */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

void main()
{
        int i, j;
        FILE *f;
        char key[KEYSIZE];
        int value1, value2;

        /* use the output of the previous step as value1 and value2 respectively*/
        // value1 = output of date -d "2018-04-17 21:08:49" +%s ;
        // value2 = output of date -d "2018-04-17 23:08:49" +%s ;

        f = fopen("keys.txt", "w");
        for (j = value1; j <= value2; j++)
        {
                srand (j);
                for (i = 0; i< KEYSIZE; i++)
                {
                        key[i] = rand()%256;
                        fprintf(f, "%.2x", (unsigned char)key[i]);
                  }
                  fprintf(f,"\n");
         }
}
```

Compile and run the program.

Commands:

```
$gcc task2.c -o task2
$./task2
```

Give your observation with screen shot.

Step 3: All the possible keys generated are written to file "keys.txt". Compile and run the following python script that will encrypt the plaintext with keys in "keys.txt" and compares it with cipher text.

Install python cryptography package

```
sudo apt-get update
sudo apt install python-pip
pip install cryptography
```

```
/*   decrypt.py  */
from Crypto import Random
from Crypto.Cipher import AES

file = open("keys.txt", "r")
ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
for i in range(0,7200):
   str = file.readline()
   key = (str[:-1]).decode("hex")
   IV = "090807060504030201 00A2B2C2D2E2F2".lower().decode("hex")
   plaintext1 = "255044462d312e350a25d0d4c5d80a34".decode("hex")
   cipher = AES.new(key, AES.MODE_CBC, IV)
   encrypted =  cipher.encrypt(plaintext1)
   print("Encrypted: " + (encrypted).encode("hex"))
   if ciphertext == encrypted.encode("hex")[0:32]:
      print("")
      print("Match found")
      print("key: "+str[:-1])
      print("Ciphertext: " + ciphertext)
      print("Encrypted: " + (encrypted).encode("hex"))
      print("")
```

Command:

$python decrypt.py

Give your observation with screen shot.

Task 3: Measure the Entropy of Kernel

The 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. Find out how much entropy the kernel has at the current moment.

Step 1: Execute the command:

Commands:
$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail
Observe the output on the screen by moving the mouse and pressing keys

Give your observation with screen shot.

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.

Step 1: Execute both the following commands simultaneously

Commands:
$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail
$ cat /dev/random | hexdump

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.

Give your observation with screen shot.

Task 5: Get Pseudo 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.

Step 1: Execute both the following commands simultaneously
Commands:
$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail
$ cat /dev/urandom | hexdump

Give your observation with screen shot.

Step 2: Measure the quality of the random number using a tool called ent.

Install the package : sudo apt install ent

Please note that when we run /dev/urandom, a lot of random numbers are generated and since is it non blocking process the number will keep generating and may not be able to observe the effect of cursor movement immediately. For that we can write random numbers to file and check its quality using ent command.

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

Give your observation with screen shot.

Step 3: The program from Task 1 can be modified to write a new 128 bit key using /dev/urandom

```
/*   task5.c   */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

void main()
{
 int i;
 FILE *random;
 unsigned char *key = (unsigned char *) malloc (sizeof (unsigned char) * KEYSIZE);

 random = fopen("/dev/urandom", "r");
 for (i = 0; i< KEYSIZE; i++)
 {
                fread(key, sizeof(unsigned char) * KEYSIZE, 1, random);
                printf("%.2x", *key);
```

```
 }
 printf("\n");
 fclose(random);
}
```

Commands:

```
$gcc task5.c -o task5
$./task5
$./task5
```

Give your observation with screen shot.

Submission

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanation to the observations that are interesting or surprising. You are encouraged to pursue further investigation, beyond what is required by the lab description. Please submit in word or PDF format only.