MH4311 Cryptography

Lecture 16

Key Generation

Wu Hongjun

Lecture Outline

- Classical ciphers
- Symmetric key encryption
- Hash function and Message Authentication Code
- Public key encryption
- Digital signature
- **Key generation, establishment and management**
  - **Key generation**
  - Key establishment
  - Secret sharing
- Introduction to other cryptographic topics

Recommended Reading

- Wikipedia

  Key generation:

  - http://en.wikipedia.org/wiki/Random_number_generation
  - http://en.wikipedia.org/wiki//dev/random
  - http://en.wikipedia.org/wiki/CryptGenRandom
  - http://en.wikipedia.org/wiki/Hardware_random_number_generator

Random Number

- **Secret** and **random** numbers are important for cryptography
  - used to generate the cipher keys
  - used for the padding in the RSA encryption scheme
  - used to generate the per-message random number in ElGamal encryption/digital signature, Digital Signature Algorithm
  - …….
- Generating secure random number is a big challenge in cryptographic applications
  - There are many applications that failed due to the weak random numbers

Random Number Generation

- Common approach to generate random number
  - Step 1.
    - Get random number from entropy source
      - May not be uniformly distributed
    - Examples of source of entropy:
      - dice, coin flipping, roulette wheels
        - » Inconvenient for cryptographic applications
      - radioactive decay, thermal noise, clock drift, the timing of actual movements of a hard disk read/write head, and radio noise …..
      - Human use of system
        - » Mouse movements, timing of keystrokes …

Random Number Generation

- Common approach to generate random number
  - Step 2.
    - Use randomness extractor to generate uniformly distributed random numbers from the random sequence from Step 1.
    - Example of random extractor: cryptographic hash function

Random Number Generation

- In practice, we should **not** use the random number generator **rand( )** of any programming language to generate secret key
  - The rand( ) function generates fake random number from the current time, so there is little entropy
- In general, we should not use any random generator of any programming language unless it has been explicitly stated that the random number generator is for cryptographic applications

Random Number Generation

- In practice, we should be extremely careful when we are generating random numbers for cryptographic applications
  - Try to avoid writing your own random number generator for cryptographic applications (the chance is high that it may be insecure)
  - Should know the details of the random number generator being used

Random Number Generation

- To generate random number for cryptographic applications, we should use the random numbers generated by an operating system (software) or use the random numbers generated from a trusted hardware device
  - Two main types of operating systems
    - UNIX-like (UNIX, Linux, Android, macOS, iOS …)
    - Microsoft Windows
  - Hardware random number generator

Random Number Generator: UNIX-like System

- On the UNIX-like system, the operating system generates random numbers and store the random numbers into the files **/dev/random** and **/dev/urandom**
  - Random numbers are generated by collecting information from the hard to predict events: keyboard and mouse, network traffic packet timings, disk drive timings …, hash these information to generate random number, then store the random number in the files /dev/random and /dev/urandom
  - When reading random number from the file /dev/random, if insufficient entropy are available, have to wait for sufficient entropy.
  - When reading random number from the file /dev/urandom, if insufficient entropy are available, just read the random number with insufficient entropy

Random Number Generator: UNIX-like System

- On the current Linux, it is better to use the system call **getrandom** to generate random number (so there is no need to open and read the random or urandom file)
  - Example given on the next slide.

Example of getrandom( ) in C/C++ code*

```
#include <sys/random.h>
#include <stdlib.h>

int main( )
{
   char buf[1024];
   int ret;

   // get 1024 random bytes, store them at the address buf
   // getrandom() returns the number of bytes written at the address buf;
   // otherwise, it returns zero to indicate error.
   ret = getrandom(buf, 1024, GRND_RANDOM);
   if (ret != 1024)  perror("getrandom failed");
}

/* GRND_RANDOM is a constant value which is set to select between
/dev/random and /dev/urandom */
```

Random Number Generator: Microsoft Windows

- On Microsoft Windows, the system call **CryptGenRandom** (or RtlGenRandom) should be used to generate random numbers for cryptographic applications
  - CryptGenRandom collects the following information to generate random number
    - The current process ID (GetCurrentProcessID).
    - The current thread ID (GetCurrentThreadID).
    - The tick count since boot time (GetTickCount).
    - The current time (GetLocalTime).
    - Various high-precision performance counters (QueryPerformanceCounter).
    - An MD4 hash of the user's environment block, which includes username, computer name, and search path. [...]
    - High-precision internal CPU counters, such as RDTSC, RDMSR, RDPMC
    - ……
  - A simple example of using CryptGenRandom to generate random numbers in C/C++:

    https://gist.github.com/pmachapman/aa79a4f58378ec96d6fae0402efd0670

Random Number Generator: Hardware

- We may also use hardware random number generator (if we trust the hardware device)
  - Example: random numbers are generated in hardware on the latest Intel and AMD CPUs. Instruction RDRAND is used to access the hardware random number generator
    - In Linux, RDRAND is used together with other entropy source to generate /dev/random

Random Number Generator

- Many entropy sources are normally quite fragile, and fail silently. Statistical tests on their output should be performed continuously.
  - Many random number generator devices include some statistical tests into the software that reads the device.

Key Generation

- Security requirements for key generation
  - **Secret**
  - **Random**
    - Enough entropy/uncertainty
  - **Independent** (at least computationally)
    - The disclosure of one key should not affect the security of other keys
- We should generate keys from secret random numbers

Key Generation Flaws

- Security flaws in key generation
  - Type 1 flaw: Random number generator is weak
  - Type 2 flaw: Random number generator is strong, but key is not generated from the random number properly

Key Generation Flaws

- Example 1 (weak random number generator)
  - 10+ years ago, some local bank used the function rand() in Java to generate keys for symmetric ciphers
- Example 2 (weak random number generator)
  - In the early implementation of secure socket layer (SSL) in Netscape web browser, the symmetric keys are generated from the current time (as accurate as millisecond) and process ID

Key Generation Flaws

- Example 3
  - In 2012, Arjen K. Lenstra collected around 6 million RSA pubic keys from the Internet
    - Some different parties are using the same modulus *n* (weak random number generator)
    - 1,995 modulus *n* share a prime factor with another *n* (weak random number generator)
    - Around 50 modulus *n* contain small prime factors, and can be factorized easily
      (improper generation of RSA keys from random number)

Key Generation Flaws

- Example 4 (weak random number generator)
  - In 2014, it was found that the key generations in all the versions of Android phones are flawed. The keys can be recovered with around 2^{30} computations
    - In the Java Cryptography Architecture (JCA) on Android phone, an algorithm is used to generate keys and random numbers from a random seed. But the random seed is not properly initialized, so there is little entropy in the seed. (Note that there are /dev/random and /dev/urandom on Android, but JCA did not use the random numbers from /dev/random and /dev/urandom)

Key Generation Flaws

- Example 5 (weird random number source)
  - Bitcoin software “Blockchain for Android” uses the random number retrieved from the website random.org to generate private keys of ECDSA
    - it is a bad idea to generate key in this way since these random numbers are known to the website random.org
  - random.org provides the normal http connection, so the random number being received is not secret
  - In 2015, random.org changed to the secure https connection. But “Blockchain for Android” still used the http connection, so the app received the faulty response from random.org, then used the faulty response as random number to generate private keys. A number of users generated the same private keys.

Summary

- Key generation
  - Good entropy source is needed
    - Avoid using the function “random( )” to generate random number for cryptographic applications
  - Apply randomness extractor to enhance randomness
- Unix-like system
  - /dev/random, /dev/urandom
  - On the current Linux, use the system call getrandom
- Microsoft Windows
  - Use the system call CryptGenRandom