Random numbers with OpenSSL

CATALDO BASILE

< CATALDO.BASILE@ POLITO.IT >

POLITECNICO DI TORINO

Agenda

- pseudo-random number generators in OpenSSL
- sample programs
- ...then with error handling

Pseudo-Random Number Generator (PRNG)

not easy generating strongly unpredictable random numbers

- nonetheless, indispensable for several cryptographic operations
  - e.g., key, IV, salt, etc.

OpenSSL RAND provides a cryptographically-secure Pseudo-Random Number Generator (PRNG)

- it's a Deterministic Random Bit Generator (DRBG)
- not truly random (impossible with deterministic algorithms)
  - but indistinguishable from a truly random one

- cryptographically secure PRNGs need a *seed* to be properly initiated
  - starting from the seed, the PRNG uses mathematical and cryptographic transforms to ensure that its output cannot be predicted
  - the seed is a source of entropy to initialize the generator

PRNG vs. seeds

ideally, the seed should be high in entropy but…

- difficult for a deterministic machine to produce true entropy
  - …collected from unpredictable events
    - low-order bits of time, keystrokes, screen data, threads and hard-disk interrupts, thermal noise, imperfections in the chip manufacture

if an insecure seed is used, the PRNG output is predictable

- bad seed = bad pseudorandom numbers
  - a predictable seed leads to a predictable sequence
- if we use PRNG to generate keys that are predictable
- …thus, the security of even a correctly designed app. is compromised
- *rand()* and even */dev/random* are not good PRNGs for cryptography purposes

OpenSSL RAND

on Unix-like and Windows OSes

- OpenSSL properly seeds the PRNG by connecting to the best source of randomness
- not guaranteed on other OSes and systems
  - manually seed your PRNG (see next slide)

OpenSSL provides three PRNGs

- public PRNG: the main one
- private PRNG: ideally used to generate random numbers that will stay private
  - only be seen inside the program
  - every time you share random numbers, PRNG vulnerabilities could be exploited
- primary: only used to seed the public and private one

RAND package functions: generate

*int RAND_bytes(unsigned char *buf, int num);*
- generates num random bytes and store them in the buffer pointed by *buf*

*int RAND_priv_bytes(unsigned char *buf, int num);*
- as RAND_bytes

- https://wiki.openssl.org/index.php/Random_Numbers

RAND_load_file

in practice

- on systems that have /dev/random available, seeding the PRNG with RAND_load_file from /dev/random is the most convenient thing to do
- be sure to limit the number of bytes read from /dev/random to some reasonable value (e.g., max 1024 bytes, 32 or 64 bytes more balanced)
  - if you put -1 to read the entire file, RAND_load_file will read data forever and never return!

```
int rc = RAND_load_file("/dev/random", 32);
if(rc != 32) {
    /* RAND_load_file failed */
}
```

consider hardware random number generators (HRNG)
true random number generator (TRNG)

Alternative libraries in C: libsodium

◦ https://github.com/jedisct1/libsodium/blob/master/src/libsodium/randombytes/sysrandom/randombytes_sysrandom.c

```
#include "sodium.h"

int foo() {
        char myString[32];
        uint32_t myInt;

        if (sodium_init() < 0) {
         /* panic! the library couldn't be initialized, it is not safe to use */
                return 1;
        }
        randombytes_buf(myString, 32);
        /* myString will be an array of 32 random bytes, not null-terminated */

        myInt = randombytes_uniform(10);
        /* myInt will be a random number between 0 and 9 */
}
```

Further reading and examples

some examples

- https://github.com/saju/misc/blob/master/misc/openssl_aes.c

how the EVP_RAND is used in practice

- https://www.openssl.org/docs/manmaster/man7/EVP_RAND.html

a more comprehensive tutorial

https://wiki.openssl.org/index.php/Random_Numbers

random reference material

- https://www.openssl.org/docs/man1.1.1/man3/RAND_priv_bytes.html

interesting paper: random primes are not so random

- https://eprint.iacr.org/2012/064.pdf