

PV181 Laboratory of security and applied cryptography



Random values and Random Number Generators

Marek Sýs

syso@mail.muni.cz, A405

CRCS

Centre for Research on
Cryptography and Security

You will learn

- What types of RNG you can find in libraries.
- What is entropy and why it is important.
- What RNGs are (in)appropriate for crypto.
- How to generate secure random values:
 - in *python*, *C*
- Why standard **rand()** and others (e.g. Mersenne Twister) are insecure.

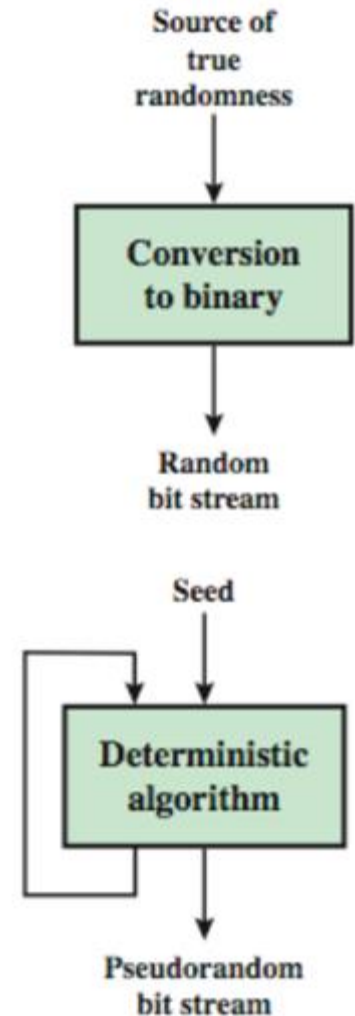
RNG types

True random (TRNG)

- Source: physical device (noise)
radio decay, thermal noise, ...
- non-deterministic, aperiodic, **slow**

Pseudo random (PRNG)

- Source: software function
- **deterministic**, periodic, very fast



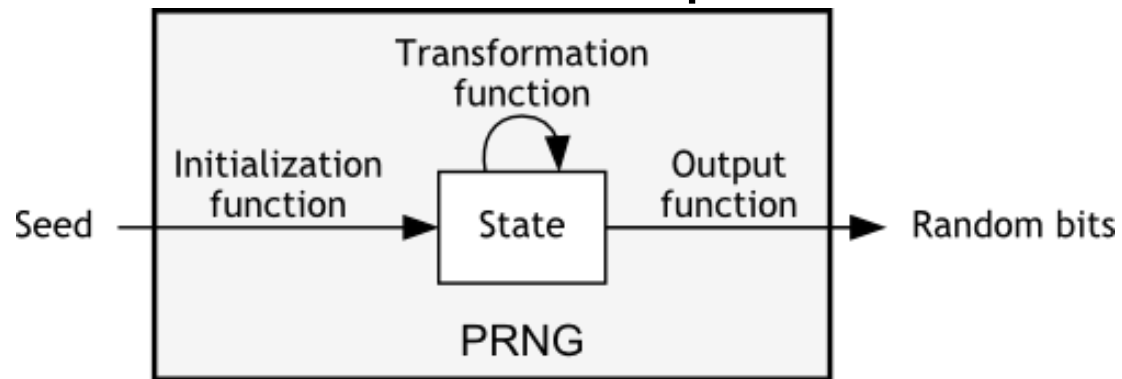
PRNG

defined by 3 functions: Init, Transform, Output

State = Init(Seed)

State = Trans(State)

rnd = Out(State)



Cryptographically secure PRNG (CSPRNG)

- **generated data** leaks no information about **next** or **previous** values \Rightarrow no info about Seed, State

Example

ANSI C portable functions

```
static unsigned long int next = 1;

int rand(void)    // RAND_MAX assumed to be 32767
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next = seed;
}
```

Standard library functions

[ANSI C\(rand\), Java\(java.util.random\),...](#)

- very fast but **very insecure** LCG generator

Linear Congruential Generator(LCG)

- $s_{n+1} = a * s_n + b \text{ mod } m$ (fixed constants a, b, c)
 1. rnd value = **State** \Rightarrow next rnd easily computed
 2. Trans is simple: s_{n+1} is **linear func** of $s_n \Rightarrow$ previous states (hence rnd values) easily computed
$$s_n = (s_{n+1} - b) / a \text{ mod } m$$
 (/a is inverse modulo!)

Weak generators

Python [random\(\)](#) - Mersenne Twister

- seed can be reconstructed from generated values
 - see [tool](#) for gclib, mt, java, etc.

C rand(): [LCG](#) generators (+ some tweaks)

- [glibc](#) (used by [GCC](#)) [rand\(\)](#) - LCG and “linear additive feedback” ($r[i] = r[i-31] + r[i-3]$)

C++: [LCG or MT or Lagged fibonacci](#)

- `minstd_rand(0 or 1)`, `mt19937(_64)`

Entropy

- measure of uncertainty
 - related to probability, attack complexity, unpredictability
- Examples:
 - 2 random bytes A,B
 - 16 bits of entropy = 2^{16} possibilities for A,B
 - 2 random bytes A, B with additional information $A \oplus B = 0$ (gained 8 bits of e.)
 - system A,B has 8 bits of e. = 2^8 possibilities
 - with additional information $A > 128$ (gained 1 bit)
 - system has only 7 bits of entropy

Practice

CSPRNG:

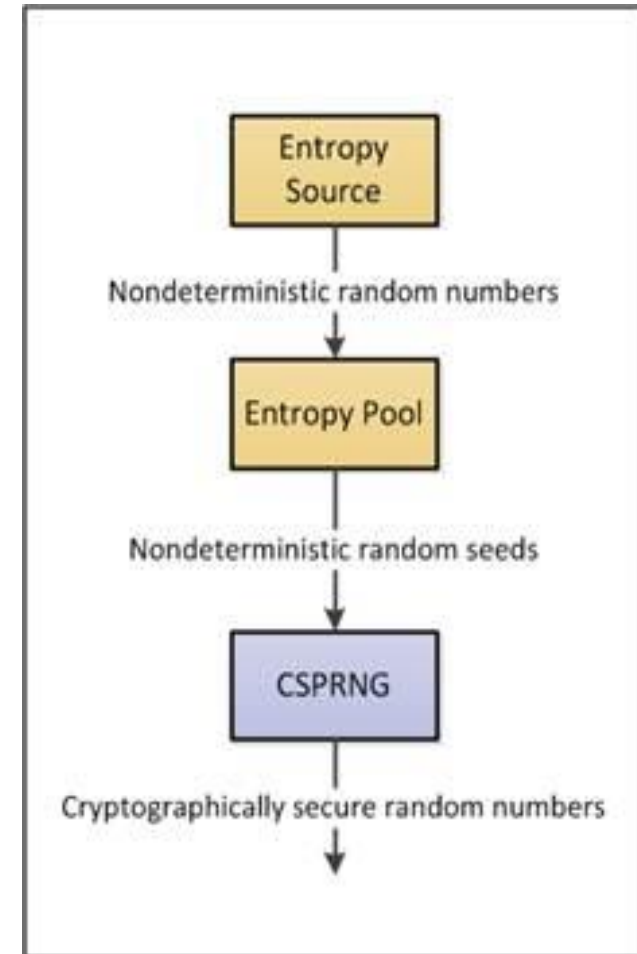
- seeded from entropy pool

Entropy pool:

- stores entropy
- usage decreases entropy in pool

TRNG (entropy source):

- repeatedly adds entropy to pool



TRNG and pools

Linux: two entropy pools (files) *dev/(u)random*

- [keyboard](#) timings, [mouse](#) movements, IDE timings

Windows: similar to Linux

- binary register *HKEY_LOCAL_MACHINE\SYSTEM\ RNG\Seed*

Additional entropy sources (if available):

- TPM, RNRAND instruction, hardware system clock (RTC), [Interrupt](#) timings, [havege](#) daemon, jitter RNG

Unix infrastructure

- pool of entropy - 2 files connected with the pool
 - pool saved at shut down!
- /dev/random
 - always produces some entropy but,
 - blocking - can block the caller until entropy available (entropy estimation)
- /dev/urandom
 - amount of entropy not guaranteed
 - always returns quickly (non blocking)

Operations

- open and read from the file to get entropy
 - use `read(2)` but **always** check if returned value == requested number of bytes (reading can be interrupted!!!)
- It is also possible to write to **`/dev/random`**
 - privileged (harmless) user can mix random data into the pool - entropy is increased (but not entropy counter)
- information about the pool in files
 - see content of `proc/sys/kernel/random/*`

Facts and recommendations

- Not all info on internet are true/reflects reality!
 - It is not necessary that dev/random blocks.
 - dev/random is more secure than dev/urandom (see [Myths about dev/urandom](#))
- dev/(u)random accessing same pool
 - when pool initialized (entropy collected in the past) files provide same quality => use /dev/urandom
- things are dynamically changing – “All of these functions provide the **same bytes**. No difference in behavior *after initialization*.” (Inside the kernel Linux, 13.09.22 😊)

Unix: methods and quality

Good sources(C):

- **initialized** random/urandom
- [getrandom\(\)](#) + flags:
 - source: random or urandom
 - blocking or non-blocking (also blocks until initialized)
- `get_random_bytes()` - kernel space
- similar in Python: [os.urandom\(\)](#), [os.getrandom\(\)](#), [secrets.token_bytes\(\)](#)

Weak sources:

- `rand`, `time(rdtsc instruction, clock func,...)`, uninitialized `urandom`

How to generate key

Good sources of entropy:

- initialized dev/urandom,
- CSPRNG seeded by dev/urandom,
- stream cipher with key generated by dev/urandom,

Implementation matters!

- seed should be protected (e.g. erased after usage)
- dev/urandom could be interrupted – always check number of obtained bytes
- use library functions to generate key – do not implement mechanism – many checks needed

Practice (python)

Follow the instructions in `install.txt` to install jupyter notebook.

Open the notebook:

1. PV181_RNG_python
2. PV181_RNG_C

Practice C

Use [Jupyter notebooks](#) is just description of tasks – not as executable notebook you used in python!

- JNs are available also on github
- 1. You can work with python JN remotely here
 - https://mybinder.org/v2/gh/sysox/PV181_RNG_2024/HEAD
- 2. Use **putty** and go to **aisa.fi.muni.cz**:
 - xlogin + secondary password

For uploading files to aisa use **winscp** or **wget**

Linux RNG design

- 3 entropy pools (store random data)
 - can be viewed as PRNG - “Init” func **mixes** (using ChaCha20) input rnd data to the state \Rightarrow state depends input data and **all** previous states!!
- input_pool (state of 4096 bits)
 - accumulate (collects, compress) the entropy from hardware events to the state
 - feeds exclusively (no access to this pool)
 - blocking_pool (state of 1024 bytes)
 - non-blocking_pool (ChaCha20 stream cipher)
 - only key (256) is fed by true rnd values
 - state (“seed” for other pools) is saved at shutdown

See [Gauvrit's blog](#) with nice scheme

Assignment 2

- 10 points
- Upload:
 - Text with the solution - change provided description.txt
 - Scripts you used
- Deadline 24.10. 8:00 (Thursday) = two weeks

Assignment 2 - Tasks

- Files (1.bin, 2.bin, 3.bin, 4.bin, 5.bin) were generated using following commands:
 - `python3 LCG.py -m 2**31 -a 1103515245 -c 12345 -l 0 -u 31 -n 10000 --file name1`
 - `python3 LCG.py -m 2**48 -a 25214903917 -c 11 -l 16 -u 47 -n 10000 --file name2`
 - `dd if=/dev/urandom of=name3 bs=4 count=10000`
 - `openssl rand -out name4 40000`
 - `rand_file.c => compile => ./a.out name5` (HINT: <https://www.mathstat.dal.ca/~selinger/random/>)
- Identify the generator and provide the mapping (replace ?):
 - 1.bin \Leftrightarrow name?, 2.bin \Leftrightarrow name?, 3.bin \Leftrightarrow name?, 4.bin \Leftrightarrow name?, 5.bin \Leftrightarrow name?
- Explain briefly how you identified the files. If it is not possible to identify the source explain why.