Workshop Maputo

Random values and Random Number Generators

Marek Sýs
syso@mail.muni.cz, Masaryk University

Centre for Research on Cryptography and Security

Overview

- Seminar (PV181 Masaryk Uni, Brno, Czech)
  - Preparation for testing
  - Example how courses look like at Masaryk Uni.
- Random number generators – theory
- Installation of the environment (Windows)
  - Copy repository
  - Installation python packages and JN
- Tasks
  - Example how to solve first task
  - Your work on the next tasks (hands on)

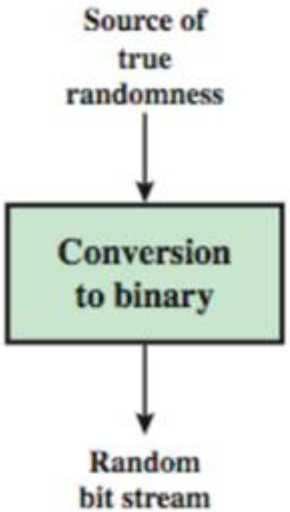
You will learn

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

RNG types

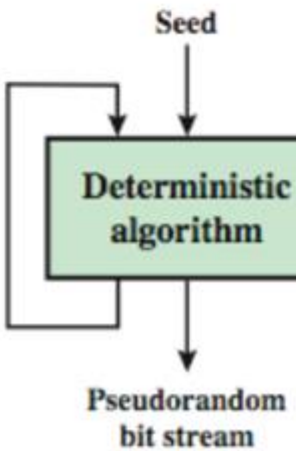
True random (TRNG)

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



Pseudo random (PRNG)

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



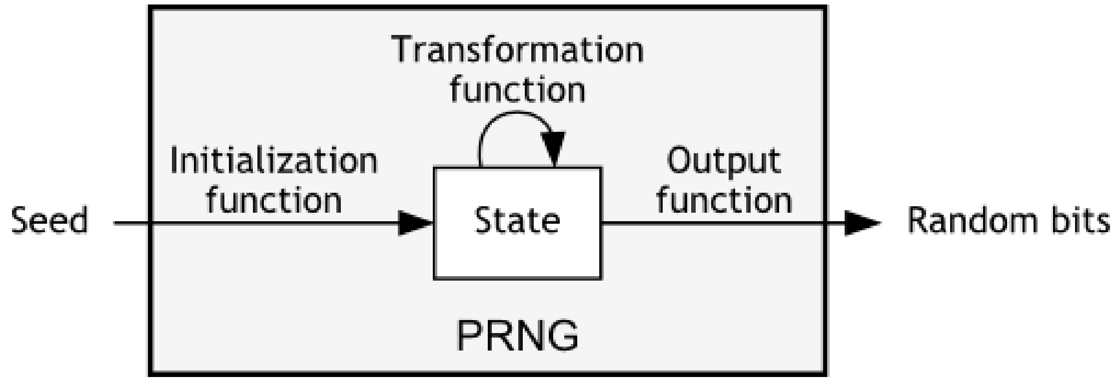
PRNG

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 ⇒ no info about Seed, State

Tasks

- Determinism of PRNG (Tasks 1-7)
  - Seed or internal determine entire sequence
- Standard PRNG
  - Implement standard ANSI C generator (Task 8)
  - Verify your implementation (Task 9)
  - Generate from different starting point (Task 10)
  - Invert the generation process (Task 11)
- Small state attack
  - Generate key, encrypt and decrypt message (Task 12,13)
  - Find the key and decrypt the message (Task 14)

Example
ANSI C portable functions

```c
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 mod m (fixed constants a,b,c)

1. rnd value = *State* ⇒ next rnd easily computed
2. Trans is simple: s_{n+1} is *linear func of* s_n ⇒ previous states (hence rnd values) easily computed

   s_n = (s_{n+1} –b)/a mod m (/a is inverse modulo!)

Tasks

- Determinism of PRNG (Tasks 1-7)
  - Seed or internal determine entire sequence
- Standard PRNG
  - Implement standard ANSI C generator (Task 8)
  - Verify your implementation (Task 9)
  - Generate from different starting point (Task 10)
  - Invert the generation process (Task 11)
- Small state attack
  - Generate key, encrypt and decrypt message (Task 12,13)
  - Find the key and decrypt the message (Task 14)

Practice

CSPRNG:

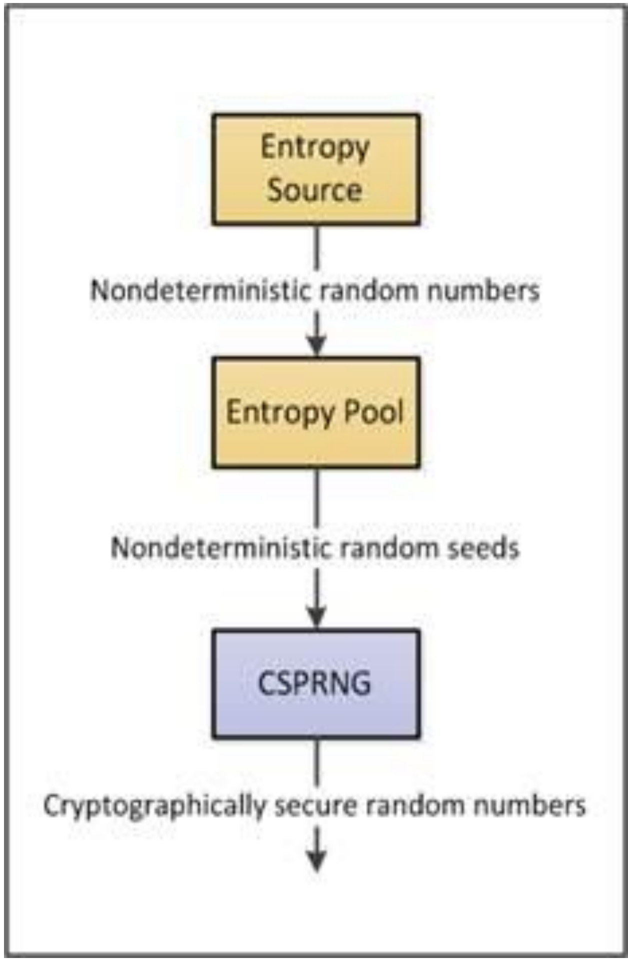
- seeded from entropy pool

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
- **/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)

Windows: similar to Linux

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

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

Tasks

- Determinism of PRNG (Tasks 1-7)
  - Seed or internal determine entire sequence
- Standard PRNG
  - Implement standard ANSI C generator (Task 8)
  - Verify your implementation (Task 9)
  - Generate from different starting point (Task 10)
  - Invert the generation process (Task 11)
- Small state attack
  - Generate key, encrypt and decrypt message (Task 12,13)
  - Find the key and decrypt the message (Task 14)

Practice (python)

- Clone the repository
  - *git clone https://github.com/sysox/Maputo_Workshop.git*
- Open powershell and go to repository
- Install
  - *pip install -r .\requirements.txt*
- Execute the jupyter notebook
  - *python -m notebook*
- Open JN with tasks
  - *PV181_RNG_python.ipynb*
- Solve the tasks:
  - If stacked 1. ask me 2. use JN with the solution