(Pseudo)Random Data

PA193 – Secure coding

Petr Švenda, Zdeněk Říha, Marek Sýs
Faculty of Informatics, Masaryk University, Brno, CZ

CROCS
Centre for Research on Cryptography and Security

Need for “random” data

- Games
- Simulations, …
- Crypto
  - Symmetric keys
  - Asymmetric keys
  - Padding/salt
  - Initialization vectors
  - Challenges (for challenge – response protocols)
  - …

“Random” data

- Sometimes (games, simulations) we only need data with some **statistical properties**
  - Uniformly distributed numbers (from an interval)
  - Long and complete cycle
    - Large number of different values
    - All values can be generated
- In crypto we **also** need **unpredictability**
  - Even if you have seen all the “random” data generated until now you have no idea what will be the random data generated next

Evaluation of randomness

- What is more random 4 or 1234?
  - **source** of randomness is evaluated (not numbers).
- Evaluation of source:
  - based on **sequence** of data it produces
- Entropy
  - measure of randomness
  - says “How hard is to guess number source generates?”

Entropy

- FIPS140-3: “Entropy is the measure of the disorder, randomness or variability in a closed system. The entropy of a random variable X is a mathematical measure of the amount of information provided by an observation of X. “
- Determined by set of values and their probabilities
- Shannon entropy:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i)$$

  - Regular coin gives **1** bit of entropy per observation
  - Biased coin **99%:1%** gives only **0.08** bits of entropy

Entropy estimates

- Difficulty of measurement/estimates
  - depends how we model the source i.e. set of values (and probabilities) we consider – single bits, two-bit blocks etc.
  - different models results in different estimation - **minimum** is the resulting estimate

- Definition Min-entropy $$H_\infty(X) \doteq \min_{i=1}^{n}(-\log p_i) = -(\max_i \log p_i) = -\log \max_i p_i \,.$$
  - lower bound for estimate

Entropy estimate - example

Source (https://qrng.anu.edu.au/RainBin.php):

```
0110110101010011000001011000001011010000
1001111111011100001010110001000111111111
```

Model (1 bit):

Stats: '0': 40, '1': 40

Entropy: H(X) =1bit info per symbol (Min = 1)

Model (2 bit):

Stats:'01': 12, '10': 6, '11': 11, '00': 11

Entropy: H(X) =1.95 per symbol (Min = 1.73)

Random number generator (RNG)

- True RNG (TRNG) – physical process

  Pros: **unpredictable**, aperiodic

  Cons: slow
- Pseudo RNG (PRNG) – software func.

  Pros: **fast**

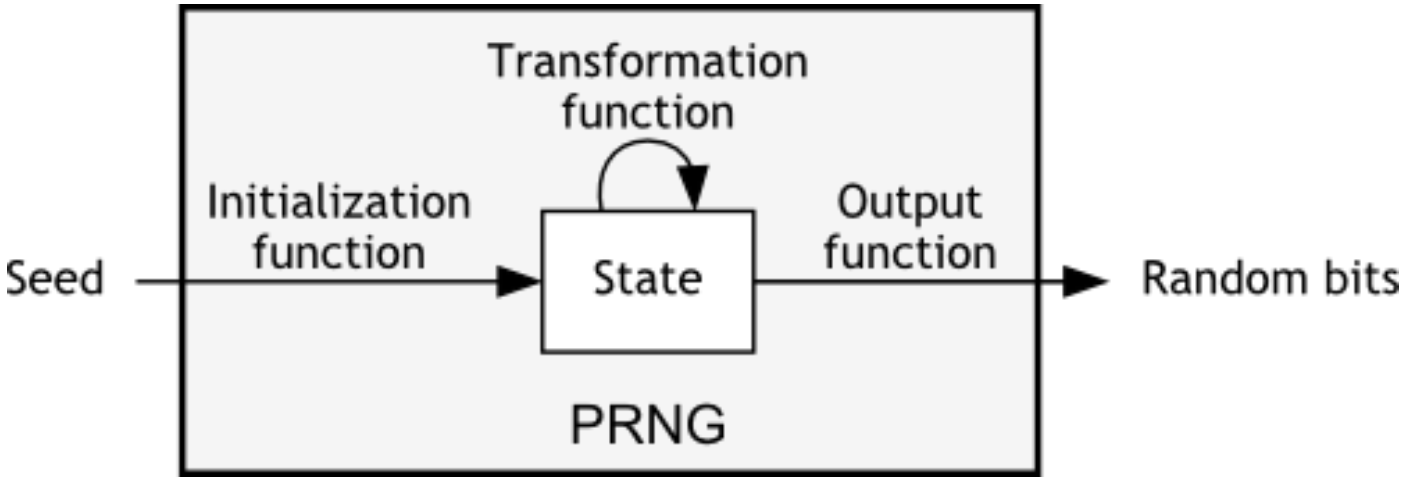
  Cons: deterministic (hence pseudo), periodic

- Combined TRNG + PRNG:

  – fast and unpredictable (depends on what attacker knows)

PRNG in general

- Initialised by **seed**

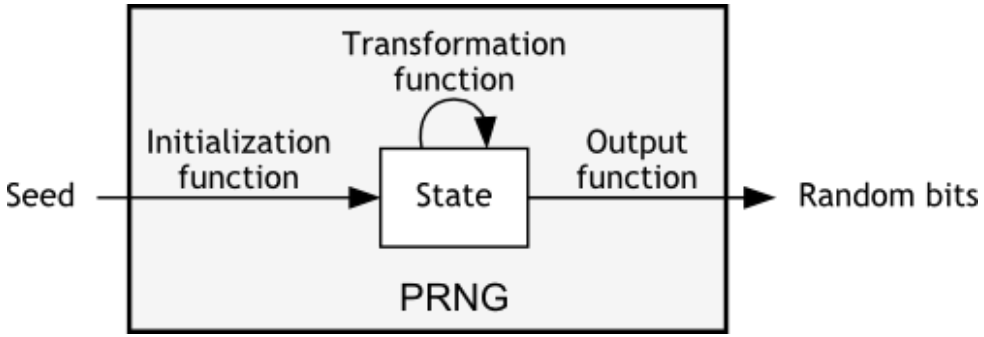


Source: http://pit-claudel.fr/clement/blog/how-random-is-pseudo-random-testing-pseudo-random-number-generators-and-measuring-randomness/

PRNG example: gcc random()

Generator type:

– LCG i.e. $S_{n+1} = a\,S_n + c\ mod\ m$ (a,c,m are constants)



State – 31 bits

Functions:

- Init, output: identity (seed → state, state → rnd)
- Transform: $S_{n+1} = 1103515245\,S_n + 12345\ mod\ 2^{31}$

https://en.wikipedia.org/wiki/Linear_congruential_generator

Secure PRNG requirements

- Secure design of function
  - No info leakage about internal state or seed
- Seed must be protected
  - seed determine whole sequence
- Internal state must be protected
  - internal state determine next random data

- Guessing the seed and internal state must be hard
  - **entropy** of seed must be large enough

Entropy of the seed

How much entropy do we need to seed a cryptographic generator securely?

Give as much entropy as the random number generator can accept. The entropy you get sets the maximum security level of your data protected with that entropy, directly or indirectly.

E.g. If a 256-bit AES key is obtained with a PRNG seeded with 56 bits of entropy, then any data encrypted with the 256-bit AES key will be no more secure than encrypted with a 56-bit DES key.

Source: Secure programming Cookbook

How to assess RNGs?

- Tests of randomness (**not sufficient** !!!)
  - Tests statistical properties (e.g. #0≈ #1, data can not be compressed, etc.)
  - Automated continuous tests – FIPS 140-1

- Cryptanalysis (human):
  - test unpredictability (statistically random ≠ unpredictable), period length, internal state leakage

Insecure PRNGs

- Linear Feedback Shift Registers (LFSR)
  - Berlekamp-Massey can compute IS from 2n values
- Standard random functions:
  - these all are LCG generators
- RC4 – Pr(2^{nd} Byte=0) =1/128 (instead 1/256)
- Mersenne Twister – IS from 624 values
- Anything else not labeled as CSPRNG…

- Not to be used for most purposes….

TRNG

- TRNG typically produce
  - biased data
  - entropy is spread across long blocks

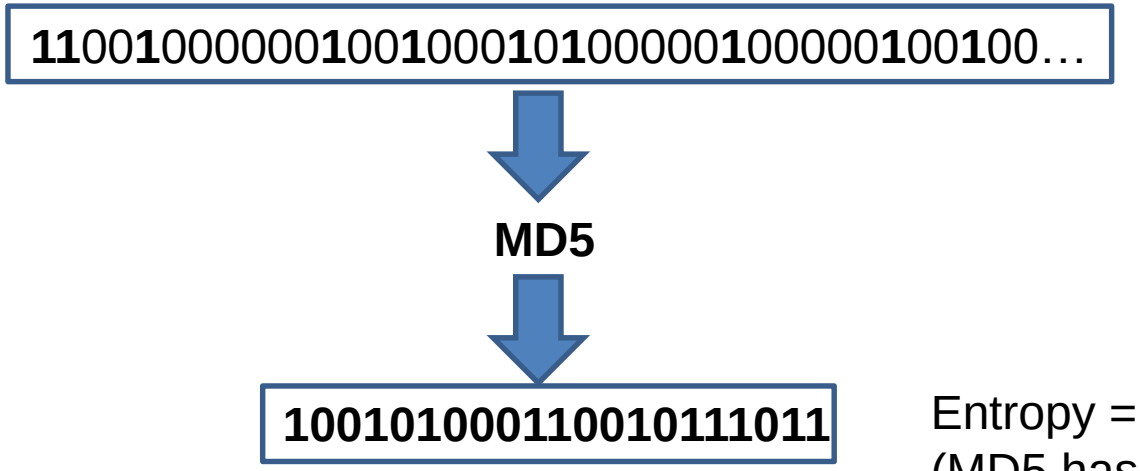
} **Postprocessing needed**

- Postprocessing: entropy extraction
  - **whitening** – hash func. applied produce unbiased data of the same entropy but with fixed size
  - entropy estimation – wise to divide estimate by factor **4** or **8** to be conservative (estimation often overestimated)

TRNG whitening example

```
00001000001010010010100000010000010100111000
00001010100000011010000000101000010010010010
00110000000100010100001010000100001010010100
```

Stats: '0': 110, '1': 42 H(X) = 0.85 info per bit



Entropy =129 bits
=0.85*152

Entropy =128 bits
(MD5 hash only 128 bits)

Tips on collecting entropy

- Make sure that any data coming from an entropy-producing source is **postprocessed with cryptography** to remove any lingering statistical bias and to help ensure that your data has at least as many bits of entropy input as bits you want to output.
- Make sure you use **enough entropy to seed** any pseudo-random number generator securely. Try not to use less than 128 bits.
- When choosing a pseudo-random number generator, make sure to pick one that explicitly advertises that it is **cryptographically strong**. If you do not see the word “cryptographic” anywhere in association with the algorithm, it is probably not good for security purposes, only for statistical purposes.
- When selecting a PRNG, prefer solutions with a refereed **proof of security** bounds. Counter mode, in particular, comes with such a proof, saying that if you use a block cipher bit with 128-bit keys and 128-bit blocks seeded with 128 bits of pure entropy, and if the cipher is a pseudo-random permutation, the generator should lose a bit of entropy after 264 blocks of output.
- Use postprocessed **entropy for seeding pseudo-random number generators** or, if available, for picking highly important cryptographic keys. For everything else, use **pseudo-randomness, as it is much, much faster**.

Source: Secure programming Cookbook

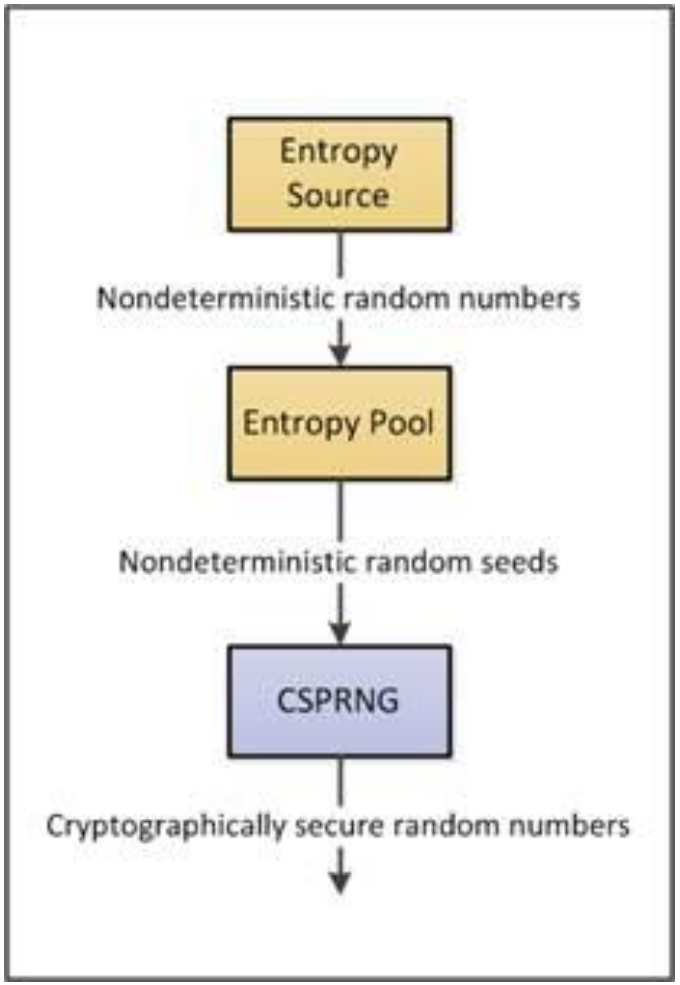
Insecure vs Secure RNGs

- Insecure RNG:
  - non-cryptographic PRNG
  - often leak information about their internal state with each output
  - TRNG without postprocessing

- Secure RNG
  - CSPRNG (Cryptographically secure PRNGs)
  - TRNG with appropriate **postprocessing**
  - combination of previous two types

Combined TRNG + PRNG

- TRNG (entropy source)
- Entropy Pool (optional)
  - collects entropy
  - new entropy is added (not replaced) to state – depends on **all** values generated by TRNG in the past
- PRNG (CSPRNG)
  - seeded by pool or TRNG directly



Secure RNGs and state compromise

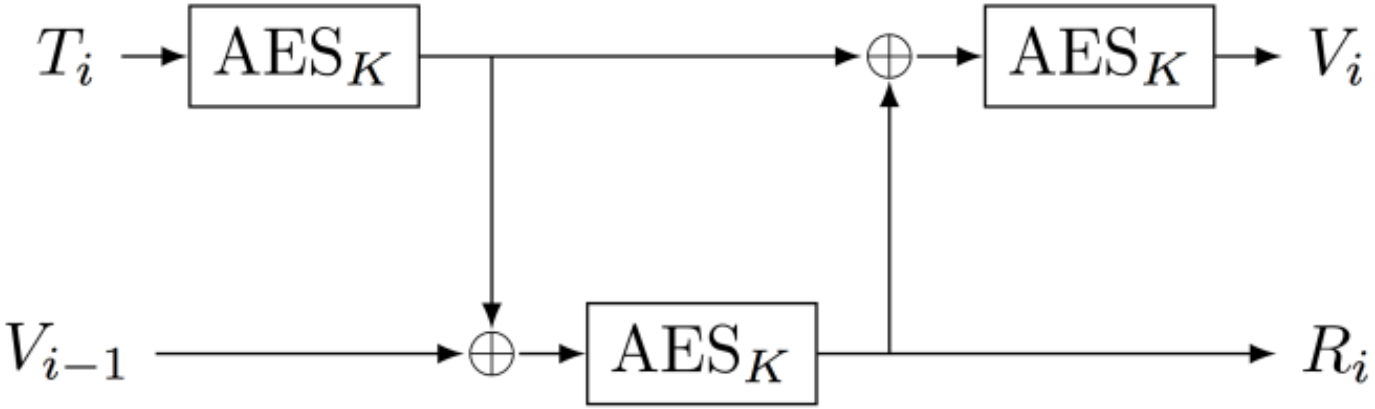
- No recovery – stream ciphers, counter mode of block ciphers, etc.
  - data are determined only by seed / state - no entropy added after initialisation
- Weak recovery - ANSI X9.17, ANSI X9.31
  - time stamps (small amount of entropy) are added in each iteration
- Good recovery – **Fortuna**, Yarrow
  - more entropy added periodically

ANSI X9.17

- ANSI X9.17 standard
  - It takes as input a TDEA (with 2 DES keys) key bundle k and (the initial value of) a 64bit random seed s. Each time a random number is required it:
    - Obtains the current date/time D to the maximum resolution possible.
    - Computes a temporary value $t = TDEA_k(D)$
    - Computes the random value $x = TDEA_k(s \otimes t)$
    - Updates the seed $s = TDEA_k(x \otimes t)$
  - Obviously, the technique is easily generalized to any block cipher
    - AES has been suggested…

ANSI 9.31

- T_i - timestamps
- V_i - temporary value
- R_i - random number



ANSI X9.31

- Security of X9.31 is not considered sufficient
- Bad recovery after internal state compromise
- The only entropy added later are the timestamps
- The entropy of timestamps is problematic
- Too much dependent on the entropy of initial values of
  - The seed
  - The symmetric encryption keys (3DES or AES)

Fortuna

- Designed by Bruce Schneier and Niels Ferguson
- Follower of the Yarrow algorithm
- Efforts to recover quickly from the internal state compromise
- Adding entropy frequently
- Fortuna is state of the art

Fortuna

- It is composed of :
  - Generator: produces pseudo-random data.
    - Based on any good block cipher (e.g. AES, Serpent,Twofish). Cipher is running in counter mode, encrypting successive values of an incrementing counter. Key is changed periodically (no more than 1 MB of data + key changed after every data request).
  - Entropy accumulator: collects genuinely random data and reseeds the generator.
    - The entropy accumulator is designed to be resistant against injection attacks thanks to the use of 32 pools of entropy (at the n^{th} reseeding of the generator, pool k is used only if 2^k divides n).
  - Seed file: stores state

Unix infrastructure

- Special files – reading files provides (pseudo)random data
  - **/dev/random**
    - Always produces entropy
    - Provides random data
    - Can block the caller until entropy available (blocking)
  - **/dev/urandom**
    - Based on cryptographic pseudorandom generator
    - Amount of entropy not quaranteed
    - Always returns quickly (non-blocking)

https://www.2uo.de/myths-about-urandom

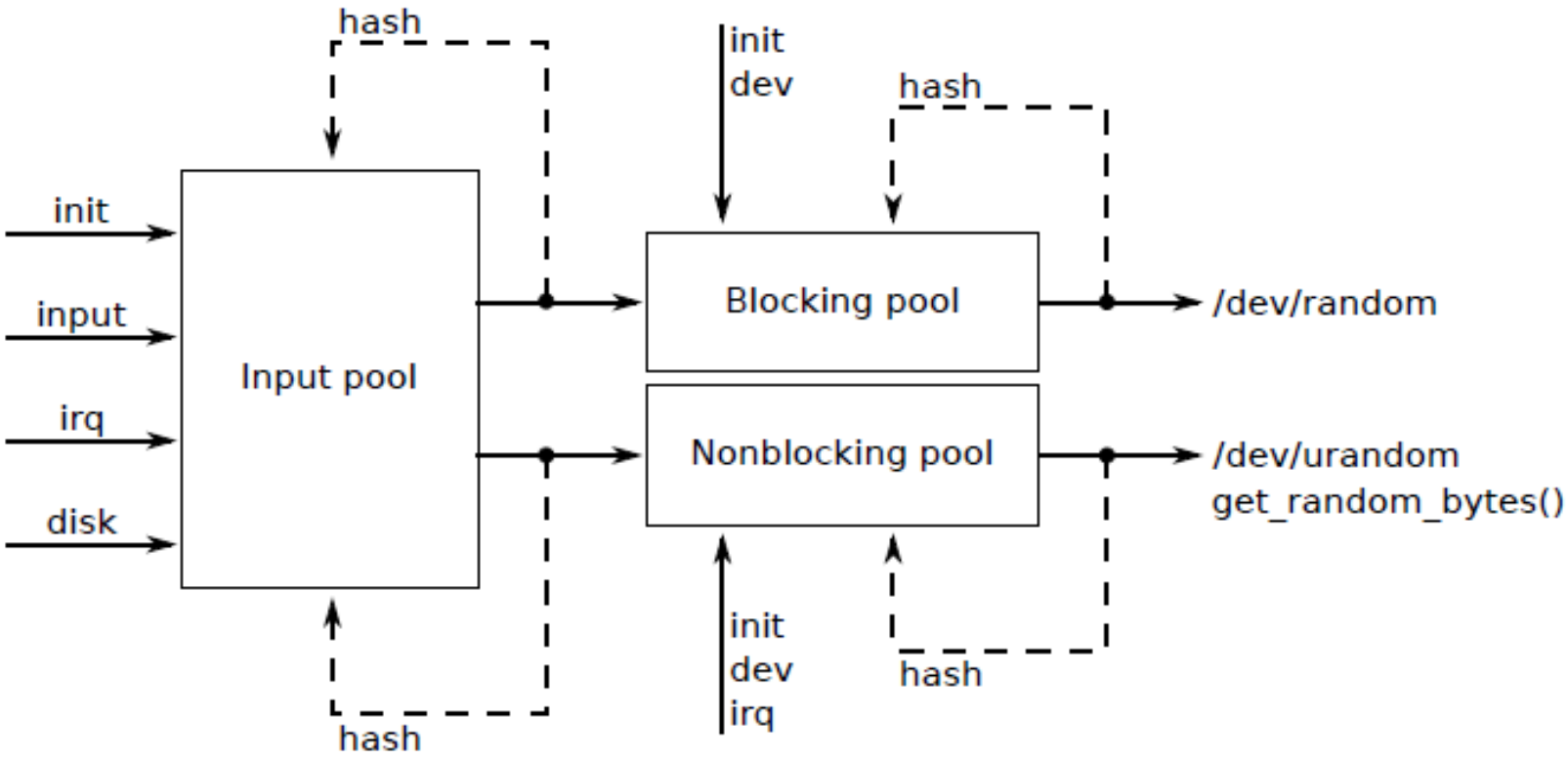
Unix infrastructure

- Available on most modern Unix-like OS
  - Including Linux, *BSD, etc.
- Each OS implements the functionality independently
  - Quality of the implementation can vary from OS to OS
- Usually no need to worry
- The core of the system is the seed of PRNG
  - The entropy of the seed may be low during/just after booting (in particular at diskless stations, virtual HW etc.)
  - The seed is often saved at shutdown

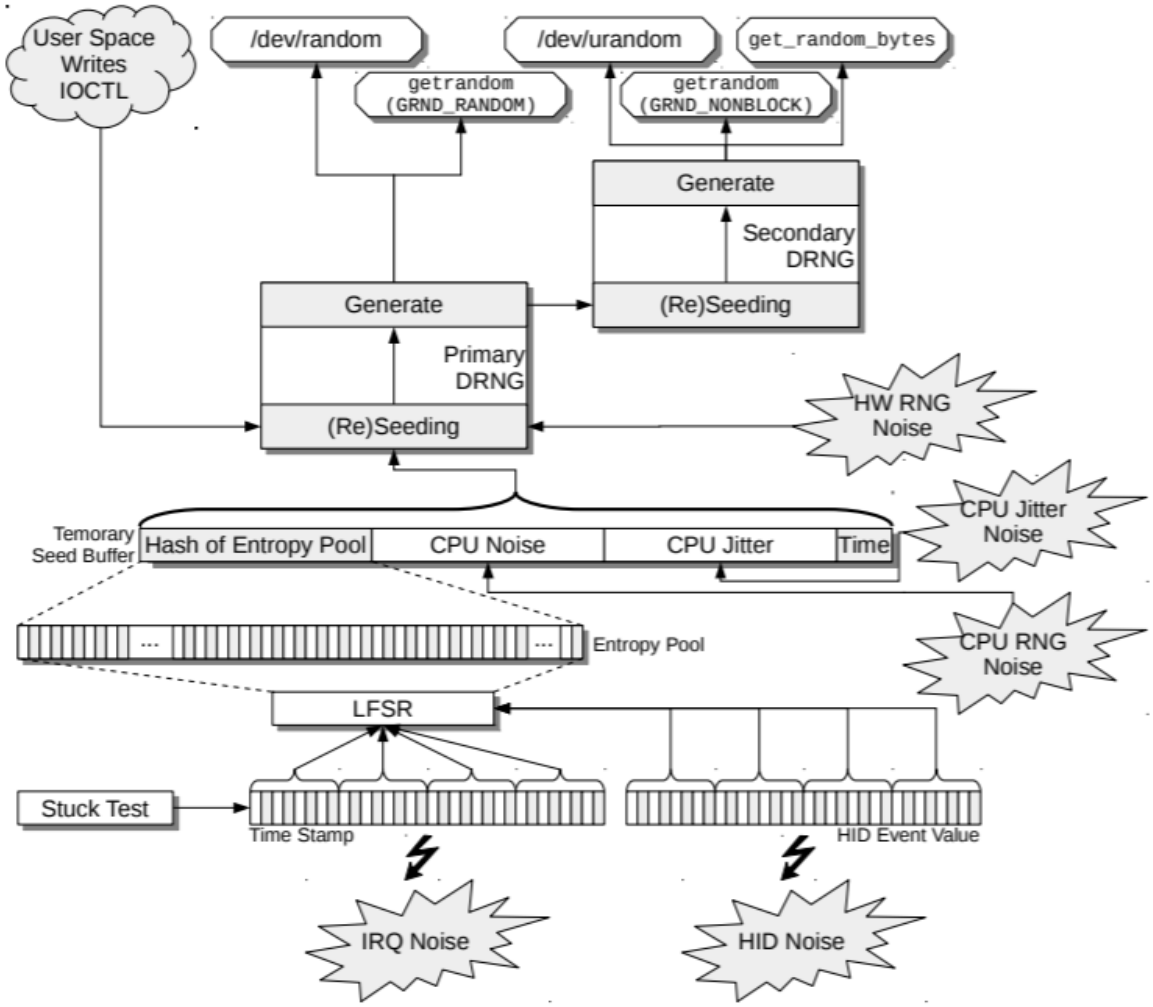
Unix infrastructure

- Operation on files
  - To get entropy use open the file and read it
    - use read(2)
    - it returns number of bytes read
    - short read (even 0 if interrupted by a signal)
- It is also possible to write to **/dev/random**.
  - This allows any user to mix random data into the pool.
  - Non-random data is harmless, because only a privileged user can issue the ioctl needed to increase the entropy estimate.
- Linux
  - The current amount of entropy and the size of the Linux kernel entropy pool are available in **/proc/sys/kernel/random/.**

Example: Linux RNG



Example: Linux RNG in details



Example: Linux

```
[root@labak ~]# more /proc/sys/kernel/random/*
::::::::::::::
/proc/sys/kernel/random/boot_id
::::::::::::::
74f01edd-e251-46fe-80a7-6cc9d264179b
::::::::::::::
/proc/sys/kernel/random/entropy_avail
::::::::::::::
159
::::::::::::::
/proc/sys/kernel/random/poolsize
::::::::::::::
4096
::::::::::::::
/proc/sys/kernel/random/read_wakeup_threshold
::::::::::::::
64
::::::::::::::
/proc/sys/kernel/random/uuid
::::::::::::::
872c22ff-a916-4cc6-a124-5e4f7ee2e932
::::::::::::::
/proc/sys/kernel/random/write_wakeup_threshold
::::::::::::::
128
```

MSDN: CryptGenRandom()

CryptGenRandom

The **CryptGenRandom** function fills a buffer with cryptographically random bytes.

```
BOOL WINAPI CryptGenRandom(
  HCRYPTPROV hProv,
  DWORD dwLen,
  BYTE* pbBuffer
);
```

Parameters

hProv
[in] Handle of a *cryptographic service provider* (CSP) created by a call to **CryptAcquireContext**.

dwLen
[in] Number of bytes of random data to be generated.

pbBuffer
[in, out] Buffer to receive the returned data. This buffer must be at least *dwLen* bytes in length.

Optionally, the application can fill this buffer with data to use as an auxiliary random seed.

Source: MSDN

MSDN: RtlGenRandom()

Syntax

C++ Copy

```
BOOLEAN RtlGenRandom(
  _Out_  PVOID RandomBuffer,
  _In_   ULONG RandomBufferLength
);
```

Parameters

RandomBuffer [out]
A pointer to a buffer that receives the random number as binary data. The size of this buffer is specified by the *RandomBufferLength* parameter.

RandomBufferLength [in]
The length, in bytes, of the *RandomBuffer* buffer.

Return value

If the function succeeds, the function returns **TRUE**.

If the function fails, it returns **FALSE**.

Source: MSDN

CryptGenRandom() vs. RtlGenRandom()

"Historically, we always told developers not to use functions such as rand to generate keys, nonces and passwords, rather they should use functions like CryptGenRandom, which creates cryptographically secure random numbers. The problem with CryptGenRandom is you need to pull in CryptoAPI (CryptAcquireContext and such) which is fine if you're using other crypto functions.

On a default Windows XP and later install, CryptGenRandom calls into a function named ADVAPI32!RtlGenRandom, which does not require you load all the CryptAPI stuff. In fact, the new Whidbey CRT function, rand_s calls RtlGenRandom".

Source:http://blogs.msdn.com/b/michael_howard/archive/2005/01/14/353379.aspx

New: BCryptGenRandom()

C++

```
NTSTATUS BCryptGenRandom(
  BCRYPT_ALG_HANDLE hAlgorithm,
  PUCHAR            pbBuffer,
  ULONG             cbBuffer,
  ULONG             dwFlags
);
```

`hAlgorithm`

The handle of an algorithm provider created by using the BCryptOpenAlgorithmProvider function. The algorithm that was specified when the provider was created must support the random number generator interface.

`pbBuffer`

The address of a buffer that receives the random number. The size of this buffer is specified by the *cbBuffer* parameter.

`cbBuffer`

The size, in bytes, of the *pbBuffer* buffer.

`dwFlags`

A set of flags that modify the behavior of this function. This parameter can be zero or the following value.

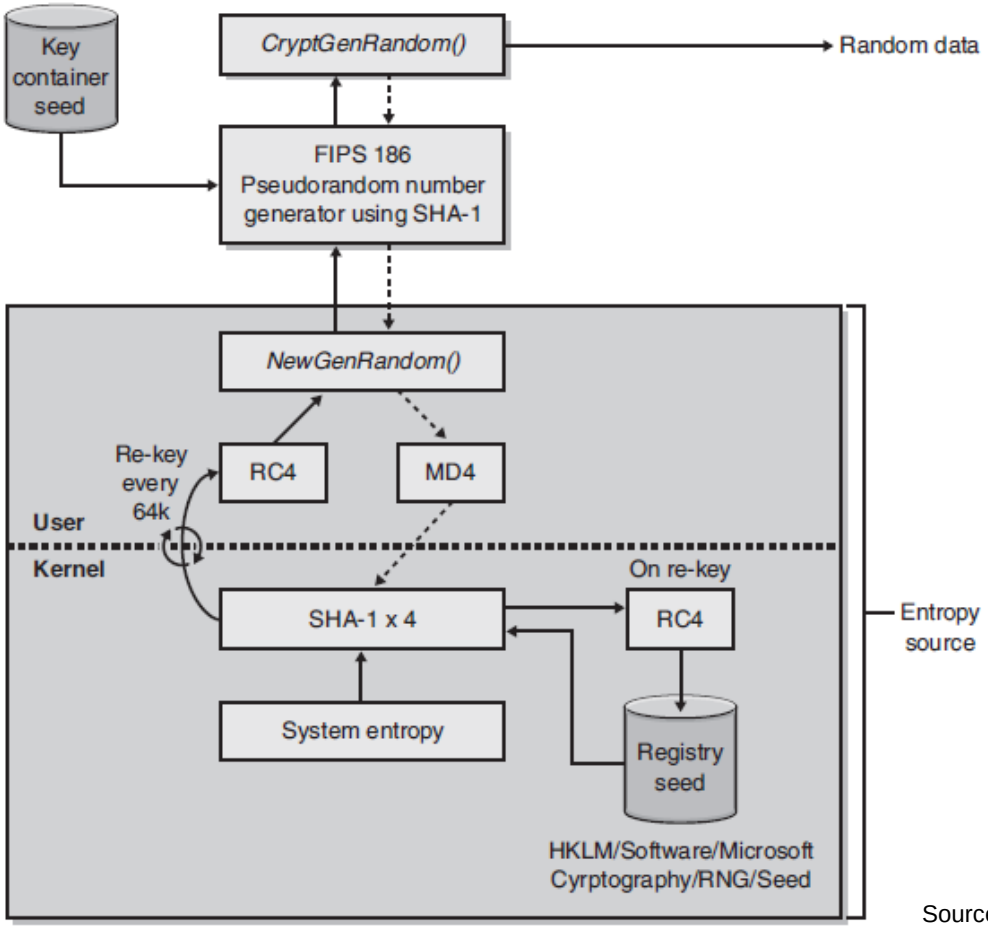
Source: MSDN

CryptGenRandom() documentation

With Microsoft CSPs, **CryptGenRandom()** uses the same random number generator used by other security components. This allows numerous processes to contribute to a system-wide seed. CryptoAPI stores an intermediate random seed with every user. To form the seed for the random number generator, a calling application supplies bits it might have—for instance, mouse or keyboard timing input—that are then combined with both the stored seed and various system data and user data such as the process ID and thread ID, the system clock, the system time, the system counter, memory status, free disk clusters, the hashed user environment block. This result is used to seed the pseudorandom number generator (PRNG). In Windows Vista with Service Pack 1 (SP1) and later, an implementation of the AES counter-mode based PRNG specified in NIST Special Publication 800-90 is used. In Windows Vista, Windows Storage Server 2003, and Windows XP, the PRNG specified in Federal Information Processing Standard (FIPS) 186-2 is used. If an application has access to a good random source, it can fill the *pbBuffer* buffer with some random data before calling **CryptGenRandom()**. The CSP then uses this data to further randomize its internal seed. It is acceptable to omit the step of initializing the *pbBuffer* buffer before calling **CryptGenRandom()**.

Source: MSDN

Design of the old Windows PRNG (up to Vista)



Source: Writing secure code, 2nd edition

MS Windows – (pseudo)random data

- Function BCryptGenRandom()
  - Part of MS CryptoAPI of next generation (CNG)

```
NTSTATUS BCryptGenRandom(
  BCRYPT_ALG_HANDLE hAlgorithm,
  PUCHAR            pbBuffer,
  ULONG             cbBuffer,
  ULONG             dwFlags
);
```

- Typical usage:

  BcryptGenRandom(NULL, (BYTE*) buffer, buffer_size, BCRYPT_USE_SYSTEM_PREFERRED_RNG)

Random data in openSSL

- OpenSSL exports its own API for manipulating random numbers. It has its own cryptographic PRNG, which must be securely seeded.
- To use the OpenSSL randomness API, you must include *openssl/rand.h* in your code and link against the OpenSSL crypto library.
- void RAND_seed(const void *buf, int num);
- void RAND_add(const void *buf, int num, double entropy);
- int RAND_load_file(const char *filename, long max_bytes);
  - Pure entropy expected
- int RAND_write_file(const char *filename);
  - To save the state of PRNG
- int RAND_bytes(unsigned char *buf, int num);

Random numbers in various languages (C, Java, Python)

- C: systems having dev/urandom
  – just open and read from file - **fopen**(“dev/urandom“, "r"))
- Python - **secrets** module
  - SystemRandom() – highest-quality of sources provided by the OS
- Java: **java.security.SecureRandom**

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[20];
random.nextBytes(bytes);
```

HW random number generators

- Require specific devices
  - More or less common
  - Price
- LavaRnd (Lava Lamp)
- Random.org
- Special devices
- Crypto devices
  - Smartcard
  - HSM

Entropy sources

- Linux:
  - default: keybord timings, mouse movement, IDE timings, network interrupts (some systems), etc
  - other can be added: audio (Fedora), video
  - Havege alg. – can be used to increase entropy pool (exec time, interrupts)
- Windows:
  - TPMs, UEFI interface, RdRand CPU instruction, hw system clock (RTC), OEM0 ACPI table content

RNG disasters

- Debian OpenSSL (2008)
  - User might not seed RNG
  - >1% of https host affected at dislosure time
- Linux boot-time entropy hole (2012)
  - User might request output before seeding RNG
  - ~1% of https and SSH servers affected
- Netscape SSL RNG (1996)
  - RNG is seeded with low entropy inputs
- ANSI X9.31
  - RNG is seeded with low entropy inputs

RNG & entropy disasters

- Dual_EC_DRBG (2014)
  - User might use backdoored PRNG design
- Juniper Dual EC Incident (2016)
  - User might use backdoored PRNG design
- Weak Infineon RSA keys (2017)
  - entropy loss of RSA keys, not RNG problem,
  - Private keys can be computed

Source: Nadia Heninger: Random number generation failures from Netscape to DUHK

Debian random number generator flaw

On May 13th, 2008 the Debian project announced that Luciano Bello found an interesting vulnerability in the OpenSSL package they were distributing. The bug in question was caused by the removal of the following line of code from md_rand.c

```
MD_Update(&m,buf,j);
[ .. ]
MD_Update(&m,buf,j); /* purify complains */
```

These lines were removed because they caused the Valgrind and Purify tools to produce warnings about the use of uninitialized data in any code that was linked to OpenSSL. You can see one such report to the OpenSSL team here. Removing this code has the side effect of crippling the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only "random" value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32,768, resulting in a very small number of seed values being used for all PRNG operations.

Source: https://www.schneier.com/blog/archives/2008/05/random_number_b.html

Debian flaw- impact

This is a Debian-specific vulnerability which does not affect other operating systems which are not based on Debian. However, other systems can be indirectly affected if weak keys are imported into them.

It is strongly recommended that all cryptographic key material which has been generated by OpenSSL versions starting with 0.9.8c-1 on Debian systems is recreated from scratch. Furthermore, all DSA keys ever used on affected Debian systems for signing or authentication purposes should be considered compromised; the Digital Signature Algorithm relies on a secret random value used during signature generation.

The first vulnerable version, 0.9.8c-1, was uploaded to the unstable distribution on 2006-09-17, and has since that date propagated to the testing and current stable (etch) distributions. The old stable distribution (sarge) is not affected.

Affected keys include SSH keys, OpenVPN keys, DNSSEC keys, and key material for use in X.509 certificates and session keys used in SSL/TLS connections. Keys generated with GnuPG or GNUTLS are not affected, though.

Source: http://www.debian.org/security/2008/dsa-1571

ECC NIST random number generator (Dual_EC_DRBG)

- Problematic
- Even more problematic after Snowden

The Guardian and The New York Times have reported that the National Security Agency (NSA) inserted a CSPRNG into NIST SP 800-90 that had a backdoor which allows the NSA to readily decrypt material that was encrypted with the aid of Dual_EC_DRBG. Both papers report that, as independent security experts long suspected, the NSA has been introducing weaknesses into CSPRNG standard 800-90; this being confirmed for the first time by one of the top secret documents leaked to the Guardian by Edward Snowden. The NSA worked covertly to get its own version of the NIST draft security standard approved for worldwide use in 2006. The leaked document states that "eventually, NSA became the sole editor.“In spite of the known potential for a backdoor and other known significant deficiencies with Dual_EC_DRBG, several companies such as RSA Security continued using Dual_EC_DRBG until the backdoor was confirmed in 2013.

Source:http://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator#NSA_backdoor_in_the_Dual_EC_DRBG_PRNG

Paper: Lousy Random Numbers Cause Insecure Public Keys

In this paper we complement previous studies by concentrating on computational and randomness properties of actual public keys, issues that are usually taken for granted. Compared to the collection of certificates considered in [12], where shared RSA moduli are "not very frequent", we found a much higher fraction of duplicates. More worrisome is that among the 4.7 million distinct 1024-bit RSA moduli that we had originally collected, more than 12500 have a single prime factor in common. That this happens may be crypto-folklore, but it was new to us, and it does not seem to be a disappearing trend: in our current collection of 7.1 million 1024-bit RSA moduli, almost 27000 are vulnerable and 2048-bit RSA moduli are affected as well. When exploited, it could act the expectation of security that the public key infrastructure is intended to achieve.

We checked the computational properties of millions of public keys that we collected on the web. The majority does not seem to suffer from obvious weaknesses and can be expected to provide the expected level of security. We found that on the order of 0.003% of public keys is incorrect, which does not seem to be unacceptable. We were surprised, however, by the extent to which public keys are shared among unrelated parties. For ElGamal and DSA sharing is rare, but for RSA the frequency of sharing may be a cause for concern. What surprised us most is that many thousands of 1024-bit RSA moduli, including thousands that are contained in still valid X.509 certificates, offer no security at all. This may indicate that proper seeding of random number generators is still a problematic issue....

Source: https://www.schneier.com/blog/archives/2012/02/lousy_random_nu.html

Best practices

- Zeroize seed – replace seed by 0‘s or rand data
- Recover from potential state compromise - reseed frequently, change keys in X9.31 but not too often
- use as much entropy sources as possible
- use more entropy 4x - 8x than alg can accept
  – especially after boot wait until enough entropy is collected
- use whitening if system does not postprocess trng data

PRNG Standards

- FIPS 186-2 (replaced later by -3 and -4)
- NIST SP 800-90A
  - Recommendation for Random Number Generation Using Deterministic Random Bit Generators
  - Hash_DRBG
  - HMAC_DRBG
  - CTR_DRBG
  - Dual EC DRBG (problematic)
- Fortuna
- ANSI X9.17-1985, Appendix C
- ANSI X9.31-1998, Appendix A.2.4
- ANSI X9.62-2005, Annex D

(P)RNG Standards

- NIST SP 800-90B
  - Recommendation for the Entropy Sources Used for Random Bit Generation
- NIST SP 800-90C
  - Recommendation for Random Bit Generator (RBG) Constructions

NIST SP 800-90A

- NIST Special Publication 800-90A
  - Recommendation for Random Number Generation Using Deterministic Random Bit Generators
- Mechanisms based on hash functions
  - Hash_DRBG
  - HMAC_DRBG
- Mechanisms based on block ciphers
  - CTR_DRBG
- Mechanisms Based on Number Theoretic Problems
  - Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG)

Testing randomness

- 3 important test suites
  - NIST STS
  - Dieharder
  - TestU01

NIST tests

- NIST Special Publication 800-22rev1a
  - "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications"
  - Revised in April 2010
  - Textual description of the tests (+ mathematics/statistics behind)
  - Software implementation
    - STS-2.1.2

Source: NIST Special Publication 800-22rev1a

NIST tests

- The 15 tests are:
  - The Frequency (Monobit) Test,
  - Frequency Test within a Block,
  - The Runs Test,
  - Tests for the Longest-Run-of-Ones in a Block,
  - The Binary Matrix Rank Test,
  - The Discrete Fourier Transform (Spectral) Test,
  - The Non-overlapping Template Matching Test,
  - The Overlapping Template Matching Test,
  - Maurer's "Universal Statistical" Test,
  - The Linear Complexity Test,
  - The Serial Test,
  - The Approximate Entropy Test,
  - The Cumulative Sums (Cusums) Test,
  - The Random Excursions Test, and
  - The Random Excursions Variant Test.

Source: NIST Special Publication 800-22rev1a

NIST test – examples of tests

2.4 Test for the Longest Run of Ones in a Block

2.4.1 Test Purpose

The focus of the test is the longest run of ones within M-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence. Note that an irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeroes. Therefore, only a test for ones is necessary. See Section 4.4.

2.11 Serial Test

2.11.1 Test Purpose

The focus of this test is the frequency of all possible overlapping m-bit patterns across the entire sequence. The purpose of this test is to determine whether the number of occurrences of the 2^m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity; that is, every m-bit pattern has the same chance of appearing as every other m-bit pattern. Note that for $m = 1$, the Serial test is equivalent to the Frequency test of Section 2.1.

Source: NIST Special Publication 800-22rev1a

Diehard tests

- Set of statistical tests to verify the quality of random number generators.
- Developed by George Marsaglia.
- Description of the test and implemetation
- Alternative GPL implemetation “Dieharder”
  – Contains also implementation of NIST STS tests

Diehard tests

- Birthday spacings: Choose random points on a large interval. The spacings between the points should be asymptotically exponentially distributed. The name is based on the birthday paradox.
- Overlapping permutations: Analyze sequences of five consecutive random numbers. The 120 possible orderings should occur with statistically equal probability.
- Ranks of matrices: Select some number of bits from some number of random numbers to form a matrix over {0,1}, then determine the rank of the matrix. Count the ranks.
- Monkey tests: Treat sequences of some number of bits as "words". Count the overlapping words in a stream. The number of "words" that don't appear should follow a known distribution. The name is based on the infinite monkey theorem.
- Count the 1s: Count the 1 bits in each of either successive or chosen bytes. Convert the counts to "letters", and count the occurrences of five-letter "words".
- Parking lot test: Randomly place unit circles in a 100 x 100 square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully "parked" circles should follow a certain normal distribution.
- Minimum distance test: Randomly place 8,000 points in a 10,000 x 10,000 square, then find the minimum distance between the pairs. The square of this distance should be exponentially distributed with a certain mean.
- Random spheres test: Randomly choose 4,000 points in a cube of edge 1,000. Center a sphere on each point, whose radius is the minimum distance to another point. The smallest sphere's volume should be exponentially distributed with a certain mean.
- The squeeze test: Multiply 231 by random floats on [0,1) until you reach 1. Repeat this 100,000 times. The number of floats needed to reach 1 should follow a certain distribution.
- Overlapping sums test: Generate a long sequence of random floats on [0,1). Add sequences of 100 consecutive floats. The sums should be normally distributed with characteristic mean and sigma.
- Runs test: Generate a long sequence of random floats on [0,1). Count ascending and descending runs. The counts should follow a certain distribution.
- The craps test: Play 200,000 games of craps, counting the wins and the number of throws per game. Each count should follow a certain distribution.