(Pseudo)Random Data

PA193 – Secure coding

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

CRO CS

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
  - Evenly 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

"Random" data generators

- Insecure random number generators
  - noncryptographic pseudo-random number generators
  - Often leak information about their internal state with each output
- Cryptographic pseudo-random number generators (PRNGs)
  - Based on seed deterministically generate pseudorandom data
- "True" random data generators
  - Entropy harvesters
  - gather entropy from other sources and present it directly

What (pseudo)random data to use?

- Avoid using noncryptographic random number generators
- For many purposes the right way is to get the seed from the true random number generator and then use it in the pseudorandom number generator (PRNG)
  - PRNG are deterministic, with the same seed they produce the same pseudorandom sequence
- There are situations, where PRNG are not enough
  - E.g. one time pad

Noncryptographic generators

- Standard rand()/srand(), random ()/srandom() functions
  - libc
- "Mersenne Twister"
- linear feedback shift registers
- Anything else not labeled as cryptographic PRNG…

- Not to be used for most purposes….

Noncryptographic generators

```
int __cdecl rand (void) {
    return(((holdrand =
            holdrand * 214013L + 2531011L)  >> 16) & 0x7fff);
}
```

```
unsigned long int next = 1;
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
```

Source: Writing secure code, 2nd edition

```
int getRandomNumber()
{
    return 4;   // chosen by fair dice roll.
                // guaranteed to be random.
}
```

Source: http://xkcd.com/

PRNG

- Cryptographic pseudo-random number generators are still predictable if you somehow know their internal state.

- Assuming the generator was seeded with sufficient entropy and assuming the cryptographic algorithms have the security properties they are expected to have, cryptographic generators do not quickly reveal significant amounts of their internal state.

- Protect the seed of the PRNG!

- Entropy of the seed matters!

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

Entropy estimates

- Entropy
  - Definition Shannon
  - Definition Min-entropy

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

$$H_\infty(X) \doteq \min_{i=1}^{n}(-\log p_i) = -(\max_i \log p_i) = -\log \max_i p_i.$$

- Difficulty of measurement/estimates
  - For example, the digits of π appear to be a completely random sequence that should pass any statistical test for randomness. Yet they are also completely predictable.

Entropy estimates

- After figuring out how much entropy is in a piece of data (e.g. expected entropy is 160 bits), it is wise to divide the estimate by a factor of 4 to 8 to be conservative.

- Because entropy is easy to overestimate, you should generally cryptographically postprocess any entropy collected (a process known as *whitening*) before using it.
  - E.g. use hash functions (SHA2)

- As most PRNG take a fixed-size seed, and you want to maximize the entropy in that seed. However, when collecting entropy, it is usually distributed sparsely through a large amount of data.
  - E.g. use hash functions (SHA2)

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

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)

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
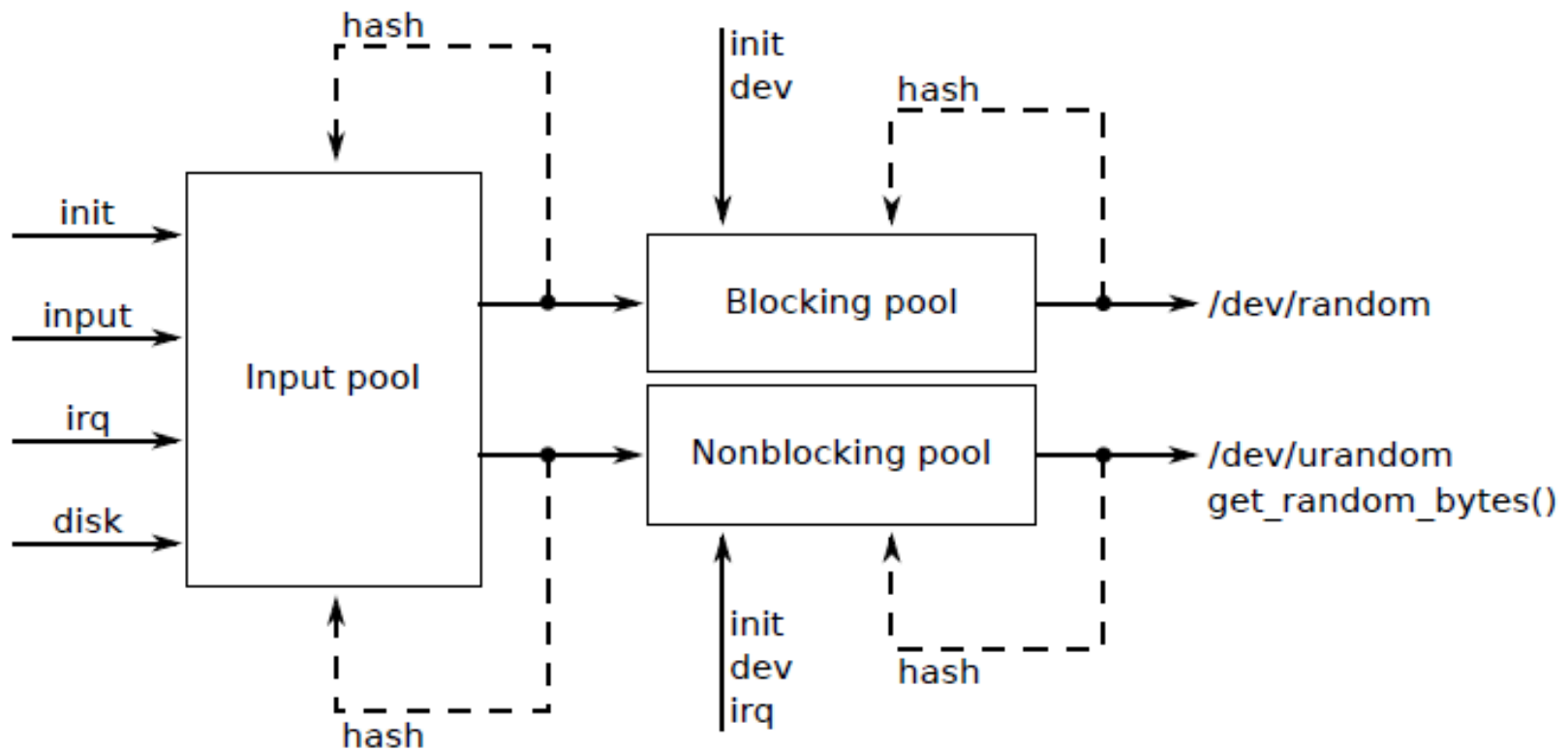
Example: Linux

```
[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
```

Example: FreeBSD

- FreeBSD implements a 256-bit variant of the Yarrow algorithm, intended to provide a cryptographically secure pseudorandom stream—this replaced a previous Linux style random device. Unlike the Linux /dev/random, the FreeBSD /dev/random device never blocks. Its behavior is similar to the Linux /dev/urandom, and /dev/urandom on FreeBSD is linked to /dev/random.

- Yarrow is based on the assumptions that modern PRNGs are very secure if their internal state is unknown to an attacker, and that they are better understood than the estimation of entropy. Whilst entropy pool based methods are completely secure if implemented correctly, if they overestimate their entropy they may become less secure than well-seeded PRNGs. In some cases an attacker may have a considerable amount of control over the entropy, for example a diskless server may get almost all of it from the network—rendering it potentially vulnerable to man-in-the-middle attacks. Yarrow places a lot of emphasis on avoiding any pool compromise and on recovering from it as quickly as possible. It is regularly reseeded; on a system with small amount of network and disk activity, this is done after a fraction of a second.

Source: Wikipedia

MS Windows – (pseudo)random data

- Function CryptGenRandom()
  - Part of MS CryptoAPI
    - First use CryptAcquireContext( )
    - and then CryptGenRandom()
  - Based on PRNG
- Internally CryptGenRandom() is using RtlGenRandom()
  - Direct call of RtlGenRandom() possible
  - Does not require loading Crypto API

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

```cpp
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

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)
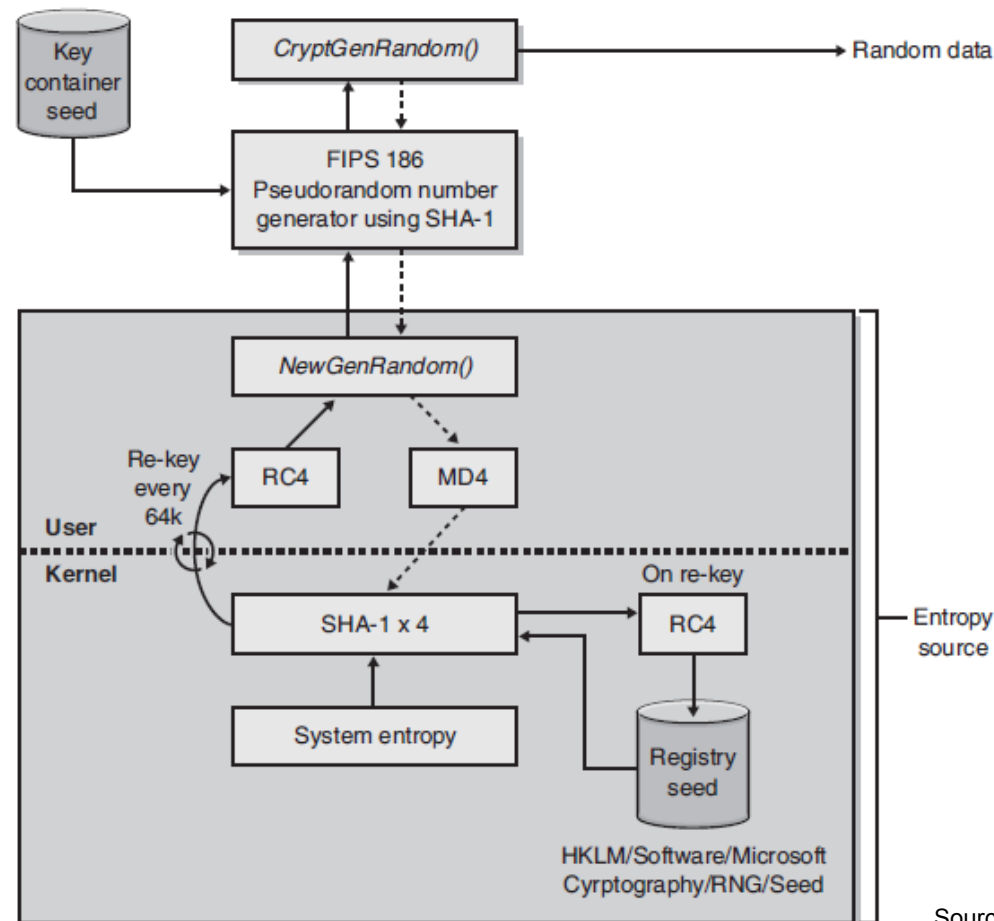
Source: Writing secure code, 2nd edition

The entropy in Windows comes from …

Low-level system information: Idle Process Time, Io Read Transfer Count, I/O Write Transfer Count, I/O Other Transfer Count, I/O Read Operation Count, I/O Write Operation Count, I/O Other Operation Count, Available Pages, Committed Pages, Commit Limit, Peak Commitment, Page Fault Count, Copy On Write Count, Transition Count, Cache Transition Count, Demand Zero Count, Page Read Count, Page Read I/O Count, Cache Read Count, Cache I/O Count, Dirty Pages Write Count, Dirty Write I/O Count, Mapped Pages Write Count, Mapped Write I/O Count, Paged Pool Pages, Non Paged Pool Pages, Paged Pool Allocated space, Paged Pool Free space, Non Paged Pool Allocated space, Non Paged Pool Free space, Free System page table entry, Resident System Code Page, Total System Driver Pages, Total System Code Pages, Non Paged Pool Lookaside Hits, Paged Pool Lookaside Hits, Available Paged Pool Pages, Resident System Cache Page, Resident Paged Pool Page, Resident System Driver Page, Cache manager Fast Read with No Wait, Cache manager Fast Read with Wait, Cache manager Fast Read Resource Missed, Cache manager Fast Read Not Possible, Cache manager Fast Memory Descriptor List Read with No Wait, Cache manager Fast Memory Descriptor List Read with Wait, Cache manager Fast Memory Descriptor List Read Resource Missed, Cache manager Fast Memory Descriptor List Read Not Possible, Cache manager Map Data with No Wait, Cache manager Map Data with Wait, Cache manager Map Data with No Wait Miss, Cache manager Map Data Wait Miss, Cache manager

■ System lookaside information consisting of Current Depth, Maximum Depth, Total Allocates, Allocate Misses, Total Frees, Free Misses, Type, Tag, and Size.

■ System interrupt information consisting of context switches, deferred procedure call count, deferred procedure call rate, time increment, deferred procedure call bypass count, and asynchronous procedure call bypass count.

■ System process information consisting of Next Entry Offset, Number Of Threads, Create Time, User Time, Kernel Time, Image Name, Base Priority, Unique Process ID, Inherited from Unique Process ID, Handle Count, Session ID, Page Directory Base, Peak Virtual Size, Virtual Size, Page Fault Count, Peak Working Set Size, Working Set Size, Quota Peak Paged Pool Usage, Quota Paged Pool Usage, Quota Peak Non Paged Pool Usage, Quota Non Paged Pool Usage, Page file Usage, Peak Page file Usage, Private Page Count, Read Operation Count, Write Operation Count, Other Operation Count, Read Transfer Count, Write Transfer Count, and Other Transfer Count.

Source: Writing secure code, 2nd edition

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);

HW random number generators

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

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

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 64 bit 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 X9.17

ANSI X9.31

Pseudorandom data
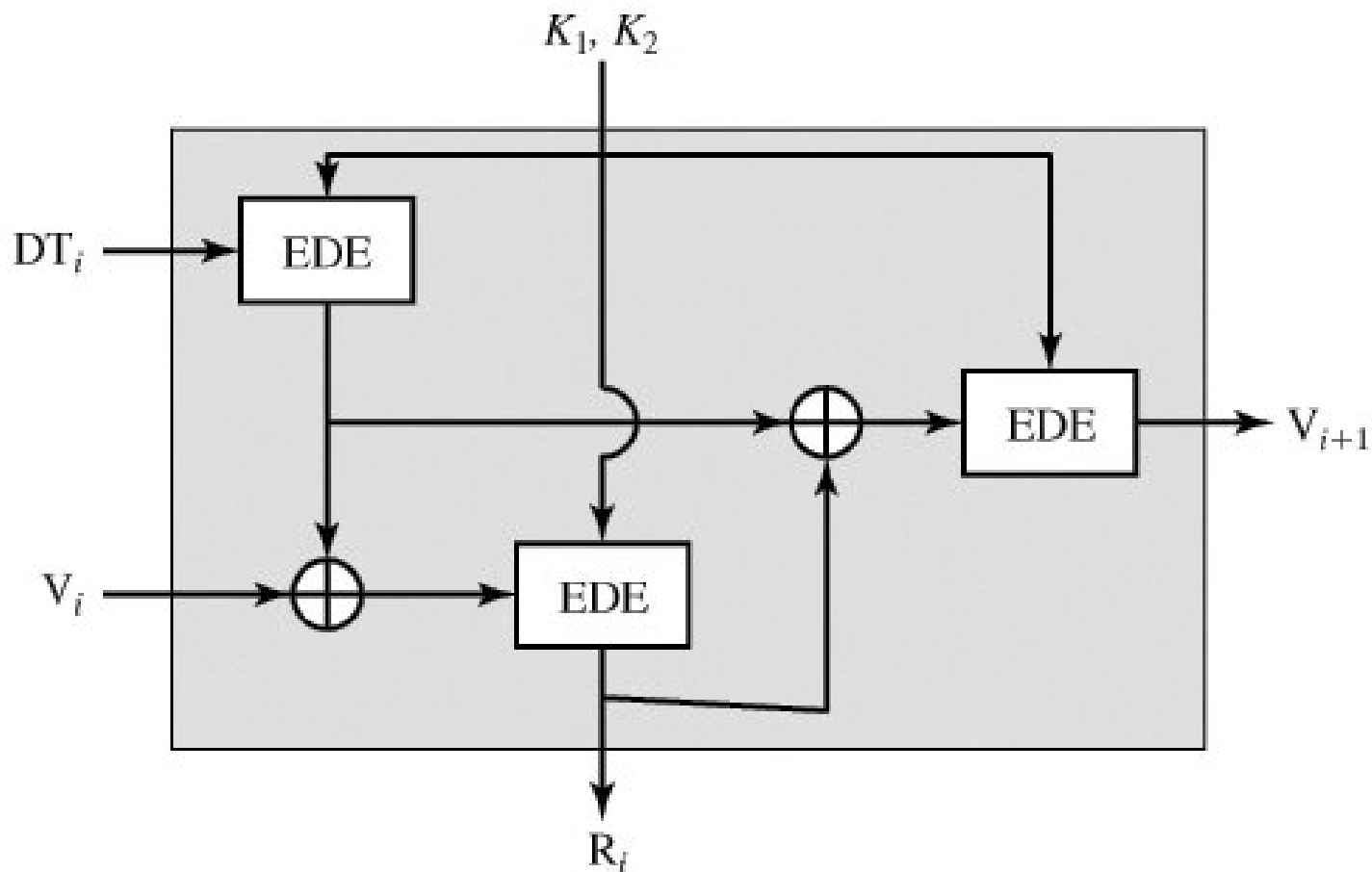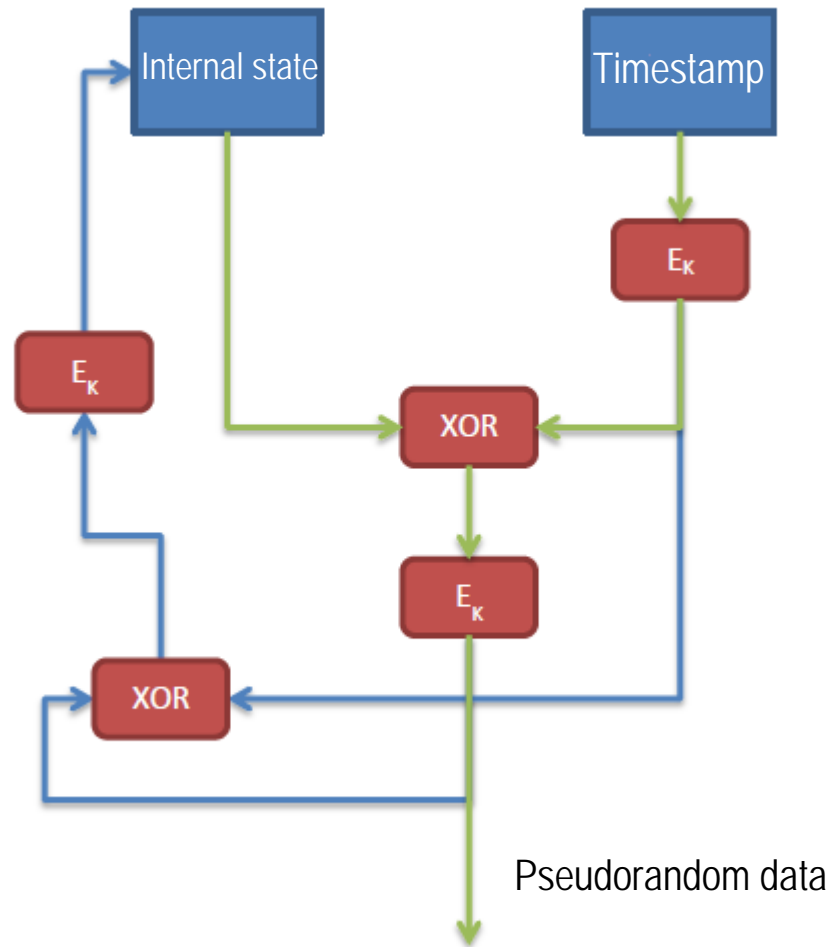
ANSI X9.31

- 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 nth reseeding of the generator, pool k is used only if 2^k divides n).
  - Seed file: stores state

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)

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.

Testing randomness

- Testing whether the generated sequence of bits "looks random", i.e. has got some statistical properties
  - E.g. the number of 0s versus the number of 1s in the sequence of bits.
- 2 important test suits
  - NIST
  - Diehard

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

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.

Using Password to derive cryptokeys

- Entropy of the password
  - Length
  - Character set
- Do not use the password directly as key
- Cryptographically process the password
  - E.g. hash it
- Derivation should slow (e.g. 1 second)
  - To slow down brute force attacks

PKCS#5

- **PBKDF1 (Password-Based Key Derivation Function 1)**
  - Up to 160 bits
  - Old, replaced by newer function
- **PBKDF2 (Password-Based Key Derivation Function 2)**

PBKDF2

- DK = PBKDF2(PRF, Password, Salt, c, dkLen)
  - PRF is a pseudorandom function (output of hlen)
  - c is the number of iterations
  - dkLen is the length of the derived key
- DK = T_1 || T_2 || ... || $T_{dklen/hlen}$
  - T_i = F(Password, Salt, Iterations, i)
    - F(Password, Salt, Iterations, i) = U_1 ^ U_2 ^ ... ^ U_c
      - U1 = PRF(Password, Salt || INT_32_BE(i))
      - U2 = PRF(Password, U1)
      - ...

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

CROCS

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.

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

Netscape <2: SSL random number weakness

```
RNG_CreateContext()
    (seconds, microseconds) = time of day;
    pid = process ID; ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12) );
    seed = MD5(a, b);

mklcpr(x)
    return((0xDEECE66D*x + 0x2BBB62DC) >> 1)

RNG_GenerateRandomBytes()
    x = MD5(seed);
    seed = seed+1;
    return x;

create_key()
    RNG_CreateContext();
    RNG_CreateRandomBytes(); RNG_CreateRandomBytes();
    challenge = RNG_CreateRandomBytes();
    secret_key = RNG_CreateRandomBytes();
```

Source: http://www.hit.bme.hu/~buttyan/courses/Revkomarom/prng.pdf

Netscape <2: SSL random number weakness

- Access to the machine with browser
  - pid, ppid is known
  - time can guessed +- 1 second
  - microsecond unknown: 20 bits
- No access to machine with browser
  - Entropy of the seed increases to max. 47 bits
- Contrast with 128 bit session key

Source: http://www.hit.bme.hu/~buttyan/courses/Revkomarom/prng.pdf

Code red worm: IP list generator

On July 12, 2001, a worm began to exploit the aforementioned buffer-overflow vulnerability in Microsoft's IIS webservers. Upon infecting a machine, the worm checks to see if the date (as kept by the system clock) is between the first and the nineteenth of the month. If so, the worm generates a random list of IP addresses and probes each machine on the list in an attempt to infect as many computers as possible. However, this first version of the worm uses a static seed in its random number generator and thus generates identical lists of IP addresses on each infected machine. The first version of the worm spread slowly, because each infected machine began to spread the worm by probing machines that were either infected or impregnable. The worm is programmed to stop infecting other machines on the 20th of every month. In its next attack phase, the worm launches a Denial-of-Service attack against www1.whitehouse.gov from the 20th-28th of each month.

Code-Red version 2 uses a random seed, so each infected computer tries to infect a different list of randomly generated IP addresses. This seemingly minor change had a major impact: more than 359,000 machines were infected with Code-Red version 2 in just fourteen hours.
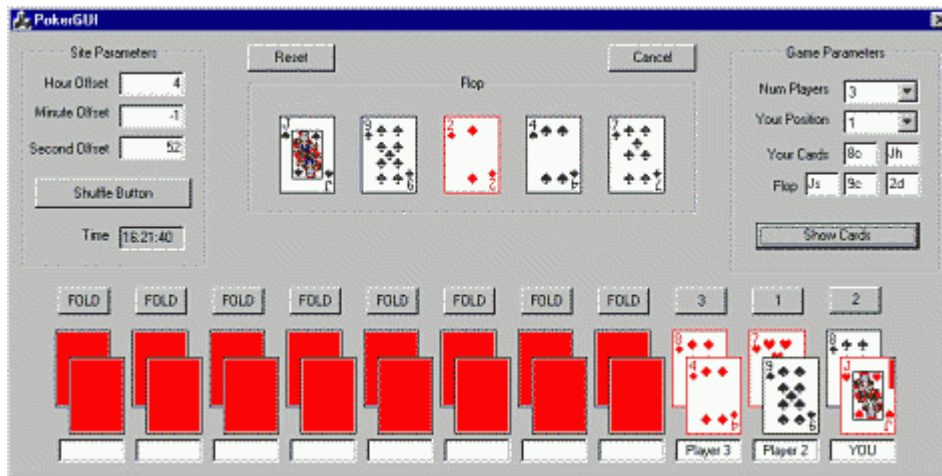
Source: http://www.caida.org/research/security/code-red/

CVE-2014-9293

- When no authentication key is set in the configuration file, ntpd(8) would generate a random key that uses a non-linear additive feedback random number generator seeded with very few bits of entropy. [CVE-2014-9293]
- The ntp-keygen(8) utility is also affected by a similar issue. [CVE-2014-9294]

Texas hold'em Poker application

- Based random number generation on standard borland random number generator
- "Reliable Software Technologies" developed a tool that required five cards from the deck to be known.

Source: http://www.ibm.com/developerworks/library/s-playing/

Dice-o-matic ☺