“**Random numbers are absolutely essential** for a crypto library, if they suck we don’t even have to get started with encryption or anything else, because it all collapses to something trivially deterministic and therefore predictable.”

Martin Boßlet


randomness

Are these bits random?

01001101110101101010

NO

“10 is random”

makes no sense

(without a context)

Talk instead of **random variables** with a given ***distribution***

An object may have been

(pseudo)randomly generated

(we talk, **a posteriori**, of (pseudo)random bits)

“Randomness means different things in various fields. Commonly, it means **lack of pattern or predictability in *events***.”

Wikipedia

Have these bits been (pseudo)randomly generated?

01001101110101101010

Have these bits been (pseudo)randomly generated?

01001101111010101101010

Probability = 2^{-20}

Have these bits been (pseudo)randomly generated?

00000000000000000000

Have these bits been (pseudo)randomly generated?

00000000000000000000

Probability = 2^{-20}

Don’t be “fooled by patterns”

PRNG are not RNGs

“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

John von Neumann

RNGs produce **random** bits

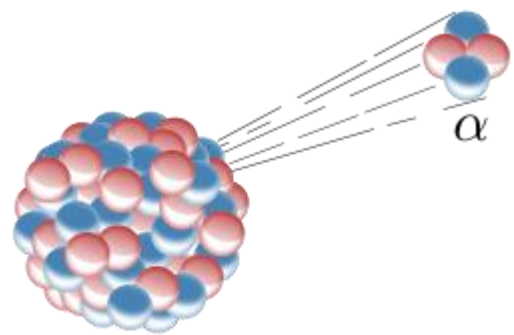
- *non-deterministically*
- *thanks to* ***analog sources***
- *with a deterministic postprocessing*

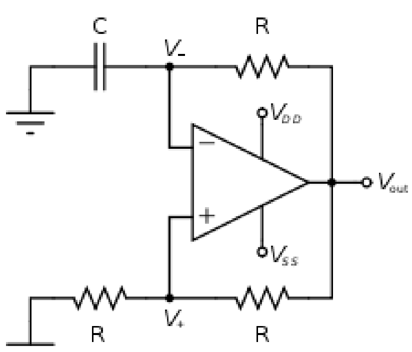
PRNGS produce **pseudorandom** bits

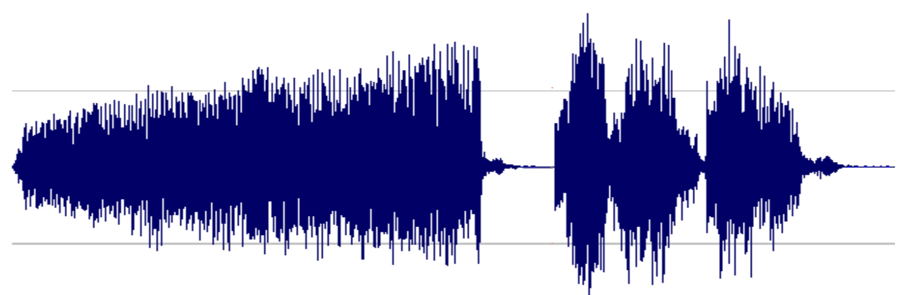
- *deterministically*
- *from a digital seed (taken from an RNG)*

Both produce "unpredictable" bits from "uncertainty"

Uncertainty comes from the real (analog/physical) world







Uncertainty quantified with the notion of

entropy

Defined for a **random variable**, for example

- Symmetric keys (should have as much entropy as bits)
- Public keys (as much entropy as $\log_2$ #choices)

$$\log_2 \#\text{choices}$$

= minimal entropy required for secure generation

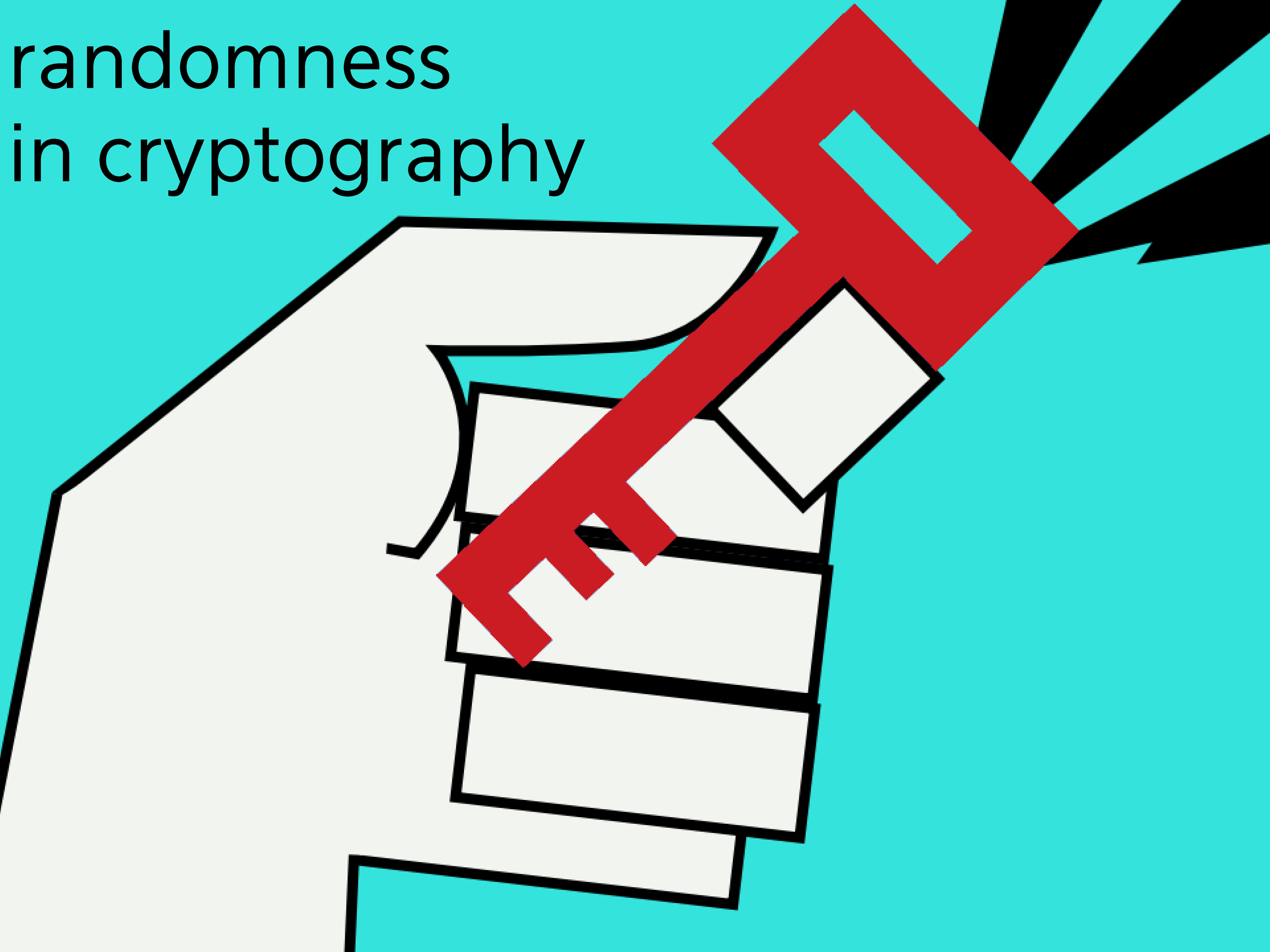
= minimal size of the RNG internal state

"I group random with stochastic or chancy, taking a random process to be one which does not operate wholly capriciously or haphazardly but in accord with stochastic or probabilistic laws."

John Earman *A Primer on Determinism*, 1986

Any distinction, given physics' laws?

randomness
in cryptography



Key generation

(symmetric and asymmetric)

Challenge-response
authentication protocols

Semantically secure encryption

IVs, nonces in block and stream ciphers

Padding in RSA-OAEP, El Gamal, etc.

Probabilistic signatures

DSA, ECDSA, etc.

Key agreement

authenticated Diffie-Hellman, MQV, etc.

Side-channel defenses

Blinding, masking, jitter, etc.

Etc. etc.

Where does randomness come from?

"Entropy" + postprocessing to eliminate biases

(bits sampled from analog sources are often not uniformly distributed, i.e;. entropy of ‹1 per bit)

Implemented in OS' as

- **`/dev/random`** and **`/dev/urandom`** on unices
- **`CryptGenRandom`** on Windows
- Ad hoc tricks...

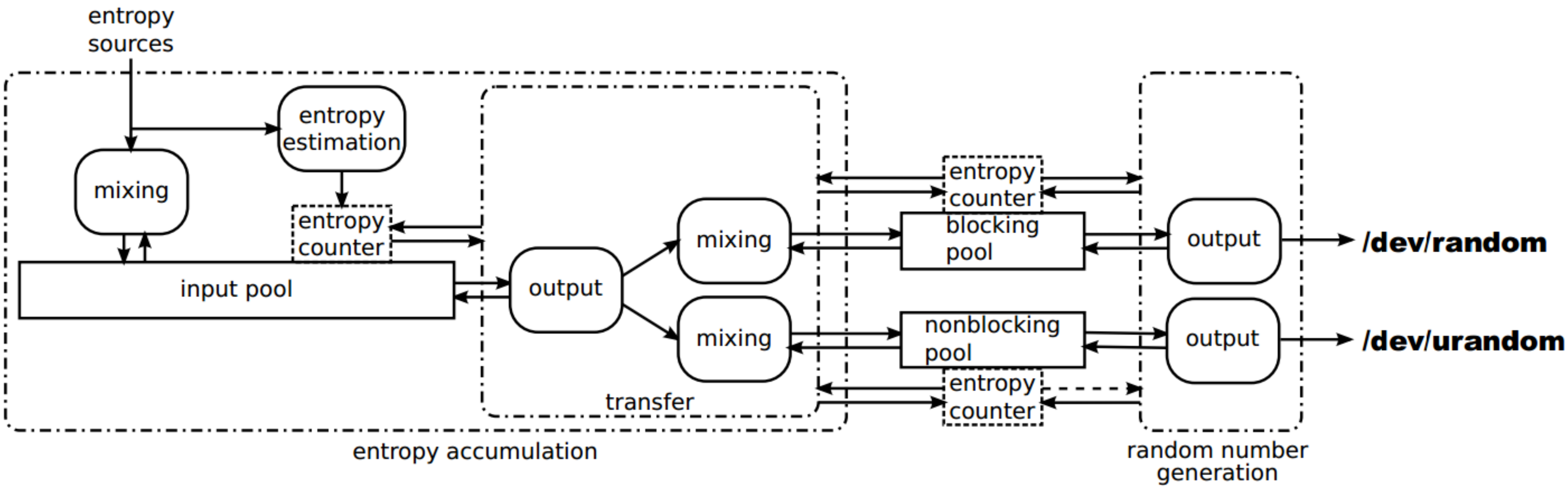
/dev/urandom and /dev/random

- Device file probing analog sources to gather entropy and generate random bytes
- Implemented differently on different OS'
  - Linux
  - FreeBSD
  - OpenBSD
  - etc.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
  int randint;
  int fd = open("/dev/urandom", O_RDONLY);
  if (fd != -1) {
    read(fd, &randint, sizeof randint);
  }
  printf("%08x\n", randint);
  close(fd);
  return 0;
}
```

`/dev/(u)random` on Linux



- Entropy from keyboard/mouse/interrupts/disk
- 4kB entropy pool, internal mixing (linear)
- Postprocessing based on SHA-1

`/dev/(u)random` on Linux

- Current entropy `/proc/sys/kernel/random/`
- Entropy appears to decrease (!) on inactivity

```
$ cat /proc/sys/kernel/random/entropy_avail
3459
$ dd if=/dev/random bs=1024 count=1 2>/dev/null | od -t x1 -An
 e4 f0 78 d7 21 21 ed b2 39 e1 1e ec 70 b5 a7 77
 52 bd d6 04 85 c9 0c 48 78 d3 b0 71 ef 1c a1 f6
 c6 f2 dc 58 4b 76 cf 1f 61 97 ba 50 26 58 5b ad
 5f fa 95 21 df 53 85 26 a0 90 ce f6 af 08 cd b2
 df 4b bf 3e c9 f7 99 10 55 e2 ec e4 32 c7 88 08
 09 73 8f d1 80 d2 f7 1e 3e db f1 a2 64 15 ea d0
 d1 7b 50 45 64 18 71 88 12 24 5d f4 1a ee 94 70
 7d 34 31 29 8a cb 2f a3 2e 7a b7 d6 89 76 3a b3
$ cat /proc/sys/kernel/random/entropy_avail
2216
$ cat /proc/sys/kernel/random/entropy_avail
2112
$ cat /proc/sys/kernel/random/entropy_avail
2005
```

`/dev/(u)random` on Linux

- `/dev/random` blocks when insufficient entropy
- This is why `gpg --gen-key` can complain

```
$ cat /proc/sys/kernel/random/entropy_avail
644
$ dd if=/dev/random bs=1024 count=1 2>/dev/null | od -t x1 -An
 4f cd fc f9 45 63 44 db 0e b8 02 e4 a6 2a 93 ef
 68 73 a1 cf cd 7c 43 87 9f ee 4c 52 60 77 d8 59
 af fb 06 4f e9 0c 9d 67 6d cd 16 68 88 f6 c0 01
 ef 96 13 25
$ cat /proc/sys/kernel/random/entropy_avail
29
$ dd if=/dev/random bs=1024 count=1 2>/dev/null | od -t x1 -An
 d7 ec 27 75 61 17 81 02
$ ^C
```

Attempting to dump 1KB blocks from `/dev/random`

`/dev/(u)random` on Linux

- `/dev/urandom` is fine in most cases
- Most libs default to `/dev/urandom` for usability

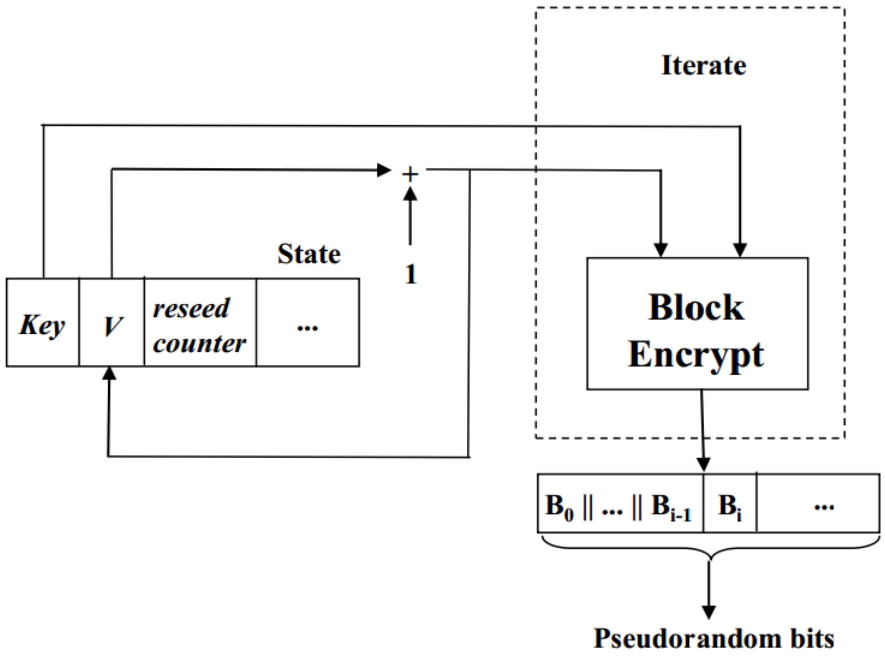
```
#ifdef DEVRANDOM
        memset(randomstats,0,sizeof(randomstats));
        /* Use a random entropy pool device. Linux, FreeBSD and OpenBSD
         * have this. Use /dev/urandom if you can as /dev/random may block
         * if it runs out of random entries.  */

        for (i = 0; (i < sizeof(randomfiles)/sizeof(randomfiles[0])) &&
                        (n < ENTROPY_NEEDED); i++)
                {
                if ((fd = open(randomfiles[i], O_RDONLY
```

`openssl-1.0.1/crypto/rand/rand_unix.c`

CryptGenRandom

- Based on AES-CTR on Vista/7 (NIST SP 900-90)
- Less straightforward use than `/dev/random`
  need to create/release a “cryptographic context”
- Harvesting entropy from
  - Environment variables
  - Process / thread IDs
  - Time and date
  - CPU counters
  - etc.



CryptGenRandom

```
#include <iostream>
#include <windows.h>
#pragma comment(lib, "advapi32.lib")

int main()
{
  HCRYPTPROV hProvider = 0;

  if (!::CryptAcquireContextW(&hProvider, 0, 0, PROV_RSA_FULL, \
                              CRYPT_VERIFYCONTEXT | CRYPT_SILENT))
    return 1;

  const DWORD dwLength = 8;
  BYTE pbBuffer[dwLength] = {};

  if (!::CryptGenRandom(hProvider, dwLength, pbBuffer))
  {
    ::CryptReleaseContext(hProvider, 0);
    return 1;
  }

  for (DWORD i = 0; i < dwLength; ++i)
    std::cout << std::hex << static_cast<unsigned int>(pbBuffer[i]) << std::endl;

  if (!::CryptReleaseContext(hProvider, 0))
    return 1;
}
```

When none is available?

1. Collect entropy from the most sources:
   - environment (e.g. `env`, `ps aux`), time, etc.
   - CPU (RDTSC, RDPMC, temperature, etc.)
   - logs (`dmesg`, `access_log`, etc.)
2. Hash the data collected with (say) SHA-2
3. Seed a strong PRNG with the result

Be. very. careful.

Bug #1: Netscape (1996)

```
global variable seed;

RNG_CreateContext()
    (seconds, microseconds) = time of day; /* Time elapsed since 1970 */
    pid = process ID;  ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12));
     seed = MD5(a, b);

mklcpr(x) /* not cryptographically significant; shown for completeness */
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

MD5() /* a very good standard mixing function, source omitted */

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

Bug #1: Netscape (1996)

An attacker who has an account on the UNIX machine running the Netscape browser can easily discover the pid and ppid values used in RNG_CreateContext() using the ps command (a utility that lists the process IDs of all processes on the system).

All that remains is to guess the time of day. Most popular Ethernet sniffing tools (including tcpdump) record the precise time they see each packet. Using the output from such a program, the attacker can guess the time of day on the system running the Netscape browser to within a second. It is probably possible to improve this guess significantly. This recovers the seconds variable used in the seeding process. (There may be clock skew between the attacked machine and the machine running the packet sniffer, but this is easy to detect and compensate for.)

Of the variables used to generate the seed in Figure 2 (seconds, microseconds, pid, ppid), we know the values of seconds, pid, and ppid; only the value of the microseconds variable remains unknown. However, there are only one million possible values for it, resulting in only one million possible choices for the seed. We can use the algorithm in Figure 3 to generate the challenge and

http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html

Bug #1: Netscape (1996)

Our second attack assumes the attacker does not have an account on the attacked UNIX machine, which means the pid and ppid quantities are no longer known. Nonetheless, these quantities are rather predictable, and several tricks can be used to recover them.

The unknown quantities are mixed in a way which can cancel out some of the randomness. In particular, even though the pid and ppid are 15bit quantities on most UNIX machines, the sum pid + (ppid << 12) has only 27 bits, not 30 (see Figure 2). If the value of seconds is known, a has only 20 unknown bits, and b has only 27 unknown bits. This leaves, at most, 47 bits of randomness in the the secret key-a far cry from the 128-bit security claimed by the domestic U.S. version.

Bug #1: Netscape (1996)

What happened

- RNG using only weak entropy sources

Lessons

- Don’t *only* use weak entropy sources
  (IDs, timestamps, machine configuration, etc.)
- Estimate entropy
- A stronger post-processing doesn’t matter
  (problem doesn’t change if MD5 is replaced with SHA-2)

Bug #2: Debian (2008)

```
char buf[100];
fd = open("/dev/random", O_RDONLY);
n = read(fd, buf, sizeof buf);
close(fd);
RAND_add(buf, sizeof buf, n);
```

```
void RAND_add(const void *buf, int num, double entropy)
```

```
  /* DO NOT REMOVE THE FOLLOWING CALL TO MD_Update()! */
  MD_Update(&m,buf,j);
  /* We know that line may cause programs such as
     purify and valgrind to complain about use of
     uninitialized data.  The problem is not, it's
     with the caller.  Removing that line will make
     sure you get really bad randomness and thereby
     other problems such as very insecure keys. */
```

Bug #2: Debian (2008)

```
Subject:    Random number generator, uninitialised data and valgrind.
Date:       2006-05-01 19:14:00

When debbuging applications that make use of openssl using
valgrind, it can show alot of warnings about doing a conditional
jump based on an unitialised value.  Those unitialised values are
generated in the random number generator.  It's adding an
unintialiased buffer to the pool.

The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif
```

Bug #2: Debian (2008)

```
Because of the way valgrind works (and has to work), the place
where the unitialised value is first used, and the place were the
error is reported can be totaly different and it can be rather
hard to find what the problem is.

...

What I currently see as best option is to actually comment out
those 2 lines of code.  But I have no idea what effect this
really has on the RNG.  The only effect I see is that the pool
might receive less entropy.  But on the other hand, I'm not even
sure how much entropy some unitialised data has.

What do you people think about removing those 2 lines of code?
```

Bug #2: Debian (2008)

2 responses on the mailing list...

Both essentially said, "go ahead, remove the `MD_update` line." The Debian maintainer did, causing `RAND_add` not to add anything to the entropy pool but still update the entropy estimate. There were other `MD_update` calls in the code that didn't use `buf`, and those remained. The only one that was a little unpredictable was one in `RAND_bytes` that added the current process ID to the entropy pool on each call. That's why OpenSSH could still generate 32,767 possible SSH keys of a given type and size (one for each pid) instead of just one.

Bug #2: Debian (2008)

What happened

- "Optimization" of sloppy code dramatically reduced the entropy of the PRNG

Lessons

- Beware of "optimizations" and of "clever code"
- Do not tweak OpenSSL (unless you really have to)
- Have the crypto code review by experts

Randomness for randomization

Often to generate **random objects**, like

- Numbers in an arbitrary range
- RSA moduli

How is this done?

- How to generate a random number in {1,2,3} ?
- How to generate a random 1k RSA modulus?

Bug #3: Cryptocat (2013)

This generates a string of 16 digits in {0,1,..,9}:

```
Cryptocat.random = function() {
        var x, o = '';
        while (o.length < 16) {
                x = state.getBytes(1);
                if (x[0] <= 250) {
                        o += x[0] % 10;
                }
        }
        return parseFloat('0.' + o);
```

What's wrong?

Bug #3: Cryptocat (2013)

`getBytes` is a strong PRNG, based on Salsa20

```
Cryptocat.random = function() {
	var x, o = '';
	while (o.length < 16) {
		x = state.getBytes(1);
		if (x[0] <= 250) {
			o += x[0] % 10;
		}
	}
	return parseFloat('0.' + o);
```

Selects integers in {0,1,..,250} => 251 possible values:

- 25 values give a **1**, 25 values a **2**, ... , 25 values a **9**
- **26 values give a 0**

Bug #3: Cryptocat (2013)

`getBytes` is a strong PRNG, based on Salsa20

```
Cryptocat.random = function() {
        var x, o = '';
        while (o.length < 16) {
                x = state.getBytes(1);
                if (x[0] <= 250) {
                        o += x[0] % 10;
                }
        }
        return parseFloat('0.' + o);
```

=› 16-digit string has **entropy 45 bits instead of 53**

=› Bruteforce would take on average 2^{44} instead of 2^{52}

Bug #3: Cryptocat (2013)

What happened

- A bias was introduced in the postprocessing of strong random bits

Lessons

- Distinguish PRNG from sampling algorithms
- Make sure the algorithm implements a uniformly random sampling
- Test, test, test

Bug #4: Routers, firewalls, switches...

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_randomness(bits)
q = prng.generate_random_prime()
N = p*q
```

What can go wrong?

Bug #4: Routers, firewalls, switches...

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_randomness(bits)
q = prng.generate_random_prime()
N = p*q
```

What if two devices start with a same seed?

(and later gather distinct entropy bits from their activity)

Bug #4: Routers, firewalls, switches...

Even more alarmingly, we are able to compute the private keys for 64,000 (0.50%) of the TLS hosts and 108,000 (1.06%) of the SSH hosts from our scan data alone by exploiting known weaknesses of RSA and DSA when used with insufficient randomness. In the case of RSA, distinct moduli that share exactly one prime factor will result in public keys that appear distinct but whose private keys are efficiently computable by calculating the greatest common divisor (GCD). We implemented an algorithm that can compute the GCDs of all pairs of 11 million distinct public RSA moduli in less than 2 hours

https://factorable.net

Bug #4: Routers, firewall, switches...

What happened

- Entropy reuse + structured sampling => Fail

Lessons

- RSA key generation is not failure-friendly
- On low-entropy platforms, wait until sufficient entropy before generating crypto secrets
- Test, test, test

Which PRNG should we use?

C(++): OpenSSL, NaCl (TODO)

Python/Ruby/Perl/Go etc.: a crypto-strong module seeding from `/dev/(u)random` or `CryptGenRandom`

If a hardware RNG is available, use it (RdRand..)

Need to rely on a specific algorithm?

- Secure-blockcipher-CTR, secure stream cipher
- Better to implement reseeding (with FS/BS)
- **Be very careful** – get your code reviewed, tested

Bug #5: Mediawiki (2012)

```php
/**
 * Return a random password. Sourced from mt_rand, so it's not particularly secure.
 * @todo hash random numbers to improve security, like generateToken()
 *
 * @return \string New random password
 */
static function randomPassword() {
        global $wgMinimalPasswordLength;
        $pwchars = 'ABCDEFGHJKLMNPQRSTUVWXYZabcdefghjkmnpqrstuvwxyz';
        $l = strlen( $pwchars ) - 1;

        $pwlength = max( 7, $wgMinimalPasswordLength );
        $digit = mt_rand( 0, $pwlength - 1 );
        $np = '';
        for ( $i = 0; $i < $pwlength; $i++ ) {
                $np .= $i == $digit ? chr( mt_rand( 48, 57 ) ) : $pwchars{ mt_rand( 0, $l ) };
        }
        return $np;
}
```

Bug #5: Mediawiki (2012)

```
/**
 * Generate a looking random token for various uses.
 *
 * @param $salt \string Optional salt value
 * @return \string The new random token
 */
public static function generateToken( $salt = '' ) {
        $token = dechex( mt_rand() ) . dechex( mt_rand() );
        return md5( $token . $salt );
}
```

This generates password-reset tokens…

Bug #5: Mediawiki (2012)

`mt_srand(seed)`/`mt_rand(min, max)`: `mt_rand` is the interface for the Mersenne Twister (MT) generator [15] in the PHP system. In order to be compatible with the 31 bit output of `rand()`, the LSB of the MT function is discarded. The function takes two optional arguments which map the 31 bit number to the [`min`, `max`] range. The `mt_srand()` function is used to seed the MT generator with the 32 bit value `seed`; if no seed is provided then the seed is provided by the PHP system.

(Argyros/Kiayias, 2012)

19937-bit state, but fully linear update and 32-bit seed:

$$x_{k+n} = x_{k+m} \oplus ((x_k \wedge 0x80000000)|(x_{k+1} \wedge \mathrm{0x7fffffff}))A$$

$$xA = \begin{cases} (x \gg 1) & \text{if } x^{31} = 0 \\ (x \gg 1) \oplus a & \text{if } x^{31} = 1 \end{cases}$$

Bug #5: Mediawiki (2012)

A session identifier preimage completely determines the seed of the mt_rand() and rand() PRNGs!

(Argyros/Kiayias, 2012)

$\Rightarrow$ Weak RNG can be exploited to

- Hijack sessions
- Predict temporary passwords

Bug #5: Mediawiki (2012)

What happened

- Weak RNG used for security purposes

Lessons

- Avoid non-crypto PRNGs, even for other applications than key generation
- In PHP, better use `openssl_random_pseudo_bytes` (and check the `$crypto_strong` flag)
- Don't use PHP's `rand()` or `mt_rand()`; C's `random(3)`, `rand(3)`; LFSRs; etc.

Can I make sure that my entropy is OK?
That my PRNG implementation is OK?

TESTING

will detect many randomness bugs

will NOT detect many randomness bugs

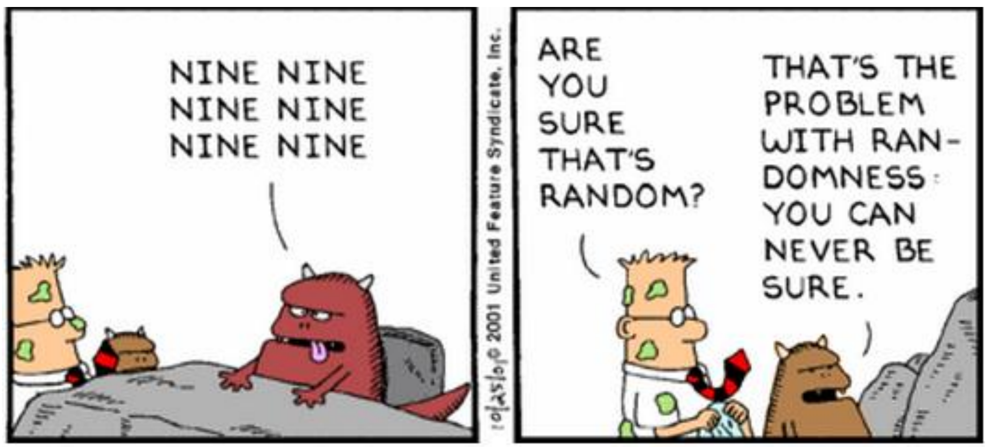
Testing (P)RNGs

Test as much as possible

- The **PRNG** function
  - Determinism, through test vectors in different settings
  - Usual software bugs, memory leaks, etc.
  - *Pseudorandomness statistical tests*
- The underlying **entropy source**
  - How much bits should be expected? Does if fail securely?
  - Is the entropy quality consistent accross OSs/hardware?
- The **usage** of pseudorandom bits
  - Sampling algorithm for software bugs
  - Distribution of sampled objects (is it really uniform?)

Statistical tests: background

Can provide evidence that a PRNG is weak, but
NOT evidence that a PRNG is *cryptographically* strong



Based on **hypothesis testing** methods

result = likelihood that the hypothesis was correct

Typically hypothesis: uniform distribution

Statistical tests: background

Remember that

1. Cryptographic weaknesses will NOT be detected by statistical tests

2. It is easy to design a weak PRNG that will successfully pass all statistical tests

Corollary: backdoors in hardware (P)RNGs can remain undetectable by statistical tests and black-box query

Statistical test suites

Test with deterministic seeds

Use large enough samples (a few megabytes)

Check the encoding in your test sample
(e.g., base-64 will look non-random if tested as binary data)

If not sure how to interpret the result, compare with a reliable source of randomness:

```
dd if=/dev/urandom of=buf bs=1014 count=1024
```

Ent (www.fourmilab.ch/random)

Easiest to use, limited set of tests

```
$ ./ent onekb
Entropy = 7.756598 bits per byte.

Optimum compression would reduce the size
of this 1014 byte file by 3 percent.

Chi square distribution for 1014 samples is 315.48, and randomly
would exceed this value 0.59 percent of the times.

Arithmetic mean value of data bytes is 125.1844 (127.5 = random).
Monte Carlo value for Pi is 3.266272189 (error 3.97 percent).
Serial correlation coefficient is 0.023406 (totally uncorrelated = 0.0).
```

Results for **1 KB** from `/dev/urandom`

Ent (www.fourmilab.ch/random)

Easiest to use, limited set of tests

```
$ ./ent onemb
Entropy = 7.999806 bits per byte.

Optimum compression would reduce the size
of this 1028196 byte file by 0 percent.

Chi square distribution for 1028196 samples is 277.22, and randomly
would exceed this value 16.21 percent of the times.

Arithmetic mean value of data bytes is 127.4411 (127.5 = random).
Monte Carlo value for Pi is 3.135441103 (error 0.20 percent).
Serial correlation coefficient is 0.001117 (totally uncorrelated = 0.0).
```

Results for **1 MB** from `/dev/urandom`

Ent (www.fourmilab.ch/random)

Easiest to use, limited set of tests

```
$ ./ent tenmb
Entropy = 7.999984 bits per byte.

Optimum compression would reduce the size
of this 10383360 byte file by 0 percent.

Chi square distribution for 10383360 samples is 224.86, and randomly
would exceed this value 91.33 percent of the times.

Arithmetic mean value of data bytes is 127.5012 (127.5 = random).
Monte Carlo value for Pi is 3.141561113 (error 0.00 percent).
Serial correlation coefficient is -0.000111 (totally uncorrelated = 0.0).
```

Results for **10 MB** from `/dev/urandom`

Diehard (http://www.stat.fsu.edu/pub/diehard/)

Suite of statistical tests, less simple to interpret

```
Chi-square with 5^5-5^4=2500 d.of f. for sample size: 256000
                       chisquare  equiv normal  p value
  Results for COUNT-THE 1's in specified bytes:
           bits  1 to  8  2536.11        .511       .695213
           bits  2 to  9  2612.60       1.592       .944354
           bits  3 to 10  2542.02        .594       .723845
           bits  4 to 11  2443.76       -.795       .213197
           bits  5 to 12  2557.82        .818       .793223
           bits  6 to 13  2512.44        .176       .569843
           bits  7 to 14  2491.77       -.116       .453666
           bits  8 to 15  2483.29       -.236       .406576
           bits  9 to 16  2452.23       -.676       .249667
           bits 10 to 17  2483.80       -.229       .409383
           bits 11 to 18  2503.36        .048       .518962
           bits 12 to 19  2590.39       1.278       .899430
           bits 13 to 20  2572.86       1.030       .848574
           bits 14 to 21  2508.75        .124       .549239
           bits 15 to 22  2650.99       2.135       .983630
           bits 16 to 23  2698.42       2.806       .997493
           bits 17 to 24  2545.39        .642       .739515
           bits 18 to 25  2428.31      -1.014       .155331
           bits 19 to 26  2376.08      -1.753       .039841
           bits 20 to 27  2696.23       2.775       .997240
           bits 21 to 28  2623.88       1.752       .960108
           bits 22 to 29  2432.46       -.955       .169760
           bits 23 to 30  2452.18       -.676       .249444
           bits 24 to 31  2427.26      -1.029       .151814
           bits 25 to 32  2414.97      -1.203       .114581
```

```
Chisquare with  6 d.o.f. =    14.77 p-value=  .977872
 :::::::::::::::::::::::::::::::::::::::::
                  For a sample of size 500:     mean
         buf              using bits  3 to 26   2.086
 duplicate       number       number
 spacings       observed     expected
       0           56.        67.668
       1          135.       135.335
       2          137.       135.335
       3           96.        90.224
       4           44.        45.112
       5           25.        18.045
 6 to INF           7.         8.282
Chisquare with  6 d.o.f. =     5.31 p-value=  .495244
 :::::::::::::::::::::::::::::::::::::::::
                  For a sample of size 500:     mean
         buf              using bits  4 to 27   1.992
 duplicate       number       number
 spacings       observed     expected
       0           67.        67.668
       1          139.       135.335
       2          133.       135.335
       3           93.        90.224
       4           40.        45.112
       5           20.        18.045
```

p-values should be uniformly distributed
(e.g., a p-value consistently close to zero indicates a bias)

CONCLUSION:

The 10 commandments

Do not rely only on **predictable entropy sources** like timestamps, PIDs, temperature sensors, etc.

Do not rely only on
non-crypto functions
like `stdlib`'s `rand()`, `random()`,
Python's `random` module,
PHP's `rand()` and `mt_rand()`,
Java's `java.util.Random`,
etc.

Do not use **non-crypto PRNGS** like LFSRs, LCGs, Mersenne Twister, etc.

Do not use RaaS

(things like www.random.org)

-› random bits may be shared or reused

Do not design **your own PRNG**, even if it’s based on strong crypto (unless you know what you’re doing)

Do not **reuse** bits accross applications

Do not conclude that a PRNG is secure just because it passes all the **statistical tests** (Ent, Diehard, etc.)

Do not assume that a crypto-secure PRNG does necessarily provide **forward or backward** secrecy, would the internal state leak to an attacker.

Do not **directly use "entropy"** as pseudorandom data (entropy from analog sources is often biased)

Do not use random bits

if you don’t have to

(it’s safer to use a counter as a nonce, for example)

TEST TEST TEST!

THANK YOU!