The plain simple reality of **entropy**

Or how I learned to stop worrying and love urandom

Filippo Valsorda — @FiloSottile

Cryptography and systems engineering at CLOUDFLARE

Heartbleed test

FAQ/status

Fork me on GitHub

If there are problems, head to the FAQ

Results are now cached globally for up to 6 hours.

Enter a URL or a hostname to test the server for CVE-2014-0160.

example.com

Go!

You can specify a port like this `example.com:4433`. 443 by default.

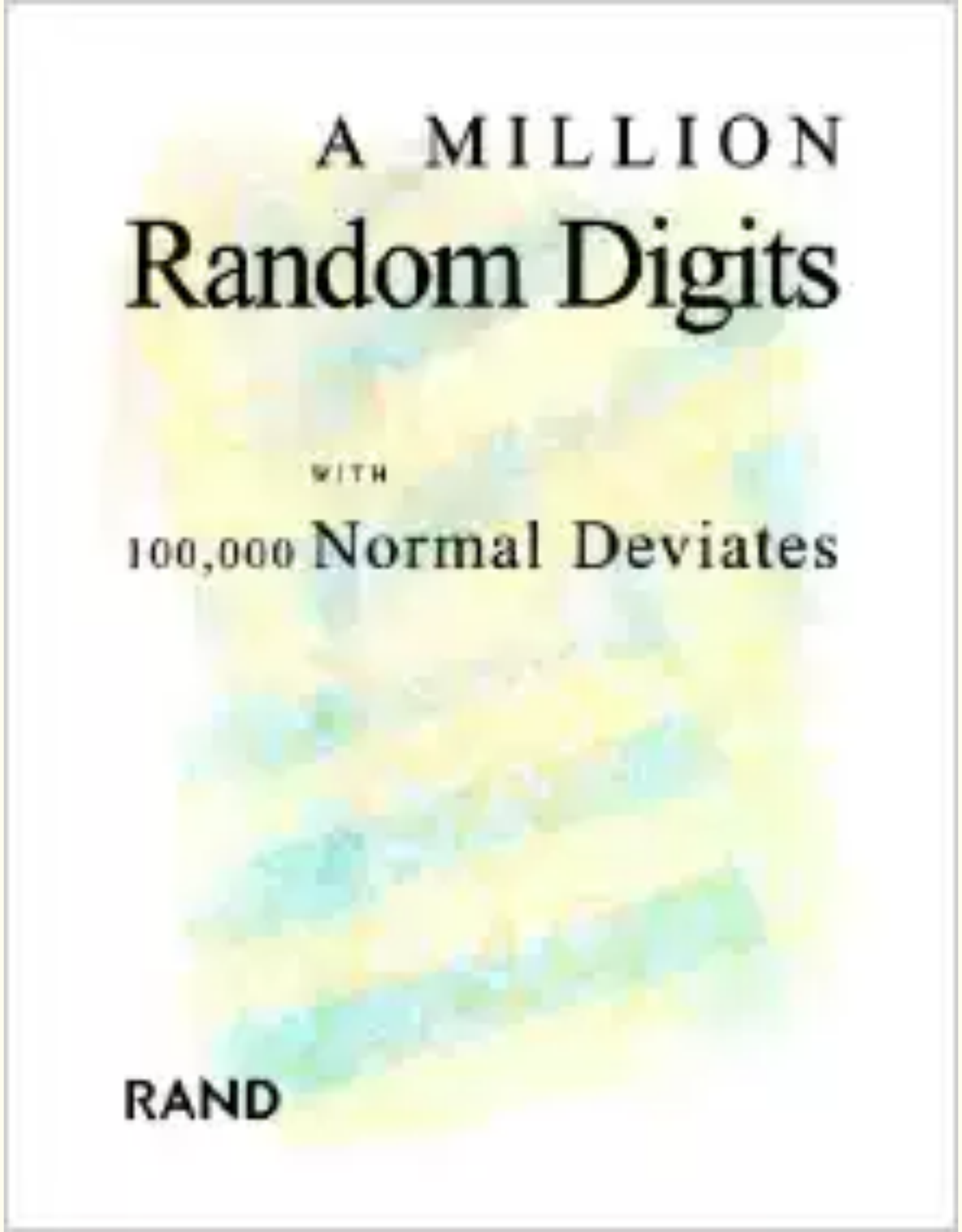
Go here for all your Heartbleed information needs.

If you want to **donate** something, I've put a couple of buttons here.

Built in a frenzy by Filippo Valsorda. Twitter, Github. <3

Random bytes

```
c6146e72e6c83e8b68799a532aec5618e0350dac638439dea033b98e5bb6
378622872a31fec37042ab0d9710cd01cd698f8c02274bb72beed5833726
83474f73b9f9a08e7629f36ac767f7ad7d759576e8b2d5cfaf9f6d386185
b782d44975a7df4360f6edfa1fb25c0dbeadaa265da6b6294897b20d54e9
3829d13e8642ad42ef34b75999dc92e29863c90dde8741e420f5938f6ce6
2f481a5a4e986c5245716417ff52ed37bf11bcd041bc83b4a980a1d75664
49a93cadfc0c2b732429ac5114ce4a2f567290b3c3ebebb9189b81d20417
14f7eba1e34717b5dde19a0a482e6333eccb1b41e80873bfd3250f5247df
599d6a7ac7b925205cadfa99189d1d5bbd2f28bcae444ad8a869e3a84dc1
df50e1fd764b81633d31249f6939735f82aa2b177fe7a437b2c559940f88
6998af04c847fa4926ee01c64ab2820ca4ef8c60b30ac65123e35ef4be9c
```



http://www.amazon.com/Million-Random-Digits-Normal-Deviates/dp/0833030477

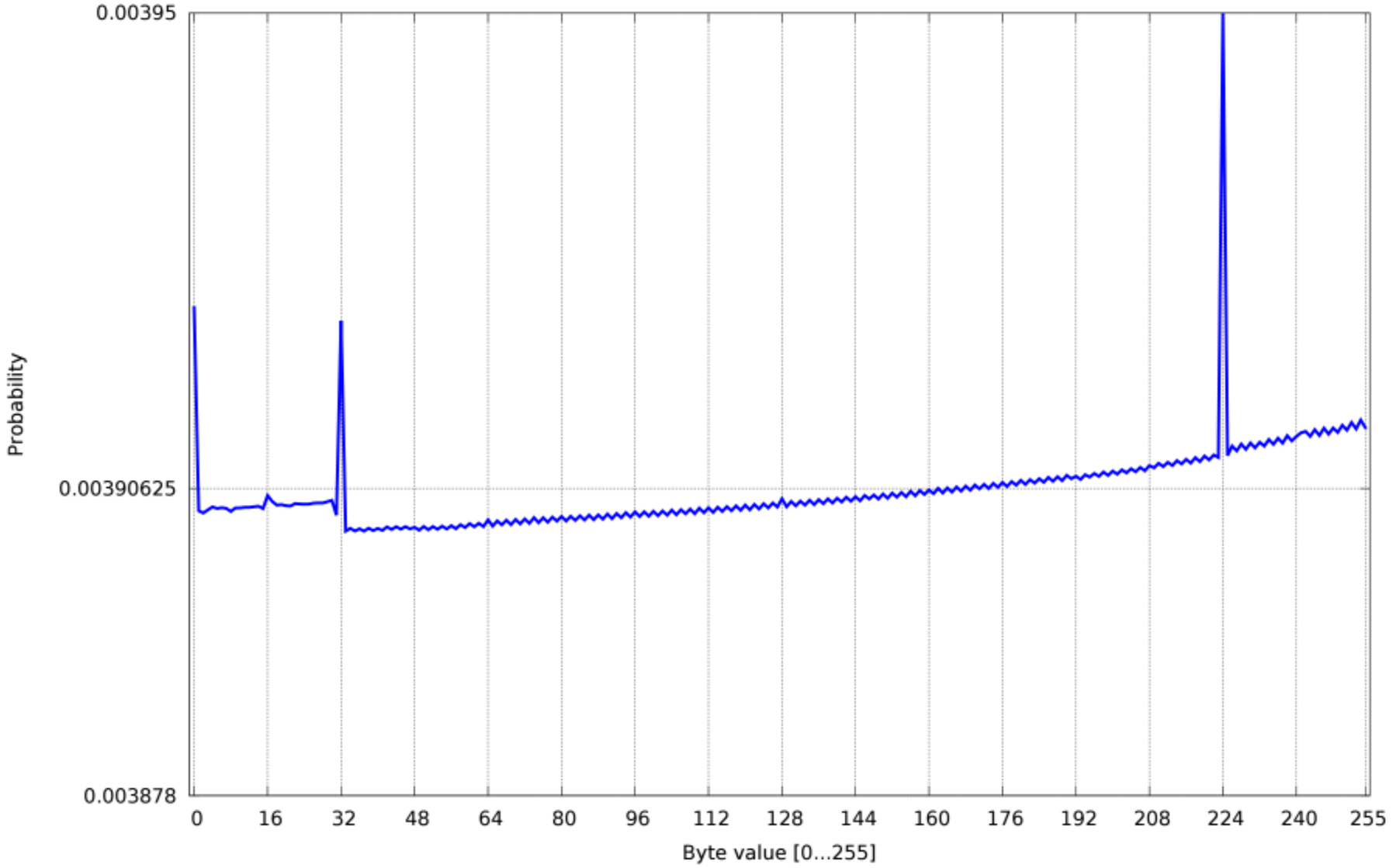
Random bytes

- **Keys**
  `ssh-keygen -b 2048`
  SSL/TLS symmetric keys
- Nonces, (EC)DSA signatures
- TCP Sequence number, DNS Query ID
- ASLR (Address space layout randomization)
- Passphrases, tokens, …
- Simulations

Random bytes

Uniform

Keystream distribution at position 32



Random bytes

Unpredictable

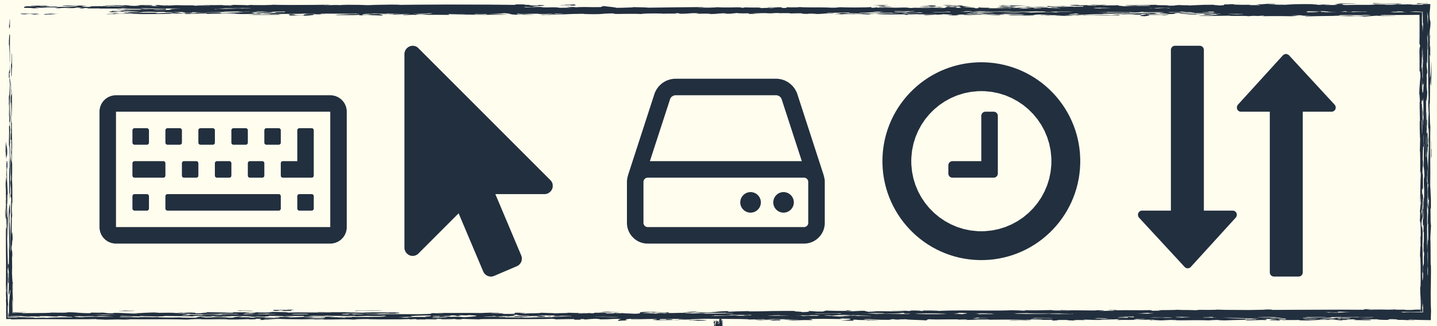
Unpredictable events

Unpredictable events

- Visible to the kernel
- Carrying variable amounts of **entropy**
- Not uniform and usually not enough

CSPRNG

cryptographically secure pseudo-random number generator



CSPRNG

```
d5a067d6be0b0a5dc90695946706e0940f0cf3821f056056e756df138f8c
29fb57acc4d21f382de3df24fbb6f3f145c3d9f194285d01ac05a44f4705
40a83c0aeeb1ed32320c072711074151c866fb8f66066bcead84edd84a49
e8dd7c02c76aeb1b15b573143d8caa49267d1a1c4b9a0fa089759583....
```

Example hash-based CSPRNG

SHA2-512(

```
1492bcf6062118a27098c426122651
805958a9b7149a8b4c534fb8721d81
d59029df69ddb0624fb07ecd55d4e0
6a74e0dcdf8b209576d2705e520eb5
9f1a3212b0e30d445cd08d06b3ccf8
fedb56c946266cb56d0df18dd2c79f
a09087f6a580f7f1dc8a1840de5483
75aeebc228421a1dadc091b9088b99
```

)

```
1b38b2f77e478cff4cc92ff99fa06d9029a2cf8a10f5cfee83ea2e7bd8a123f7
31ff26c51e048c5030cb3469349fe221835f7ffc70893c5b2674691b7dafc744
```

Example hash-based CSPRNG

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

(15,235)

SHA2-512

```
68d2cc2dc357f722f2b6fef1e99f86f022e9b2a3fcc104b55084393448c5cfee
ec9b6d165f2409f7f230bc22d72fb28664acd2e4f22eb3d5ff57097c52754f10
```

Example hash-based CSPRNG

```
68d2cc2dc357f722f2b6fef1e99f86f0
22e9b2a3fcc104b55084393448c5cfee
ec9b6d165f2409f7f230bc22d72fb286
64acd2e4f22eb3d5ff57097c52754f10
```

1.27589

SHA2-512

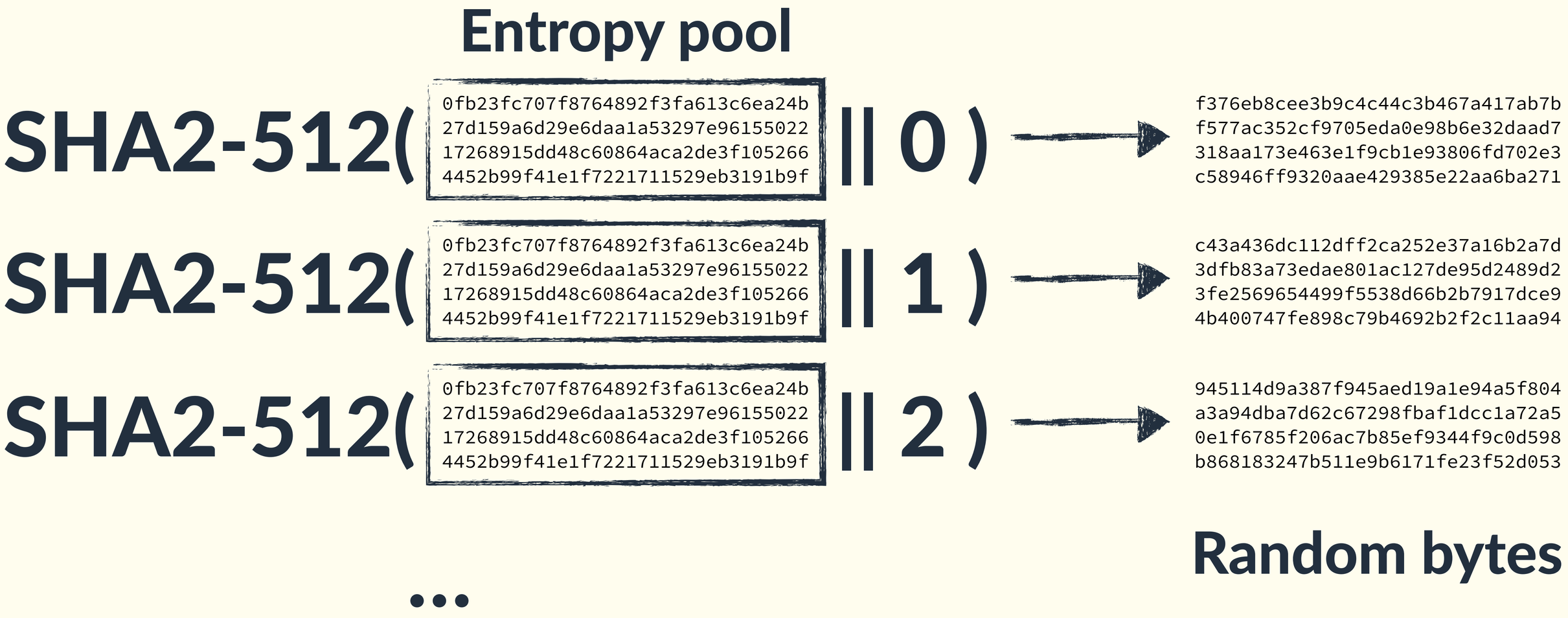
```
0fb23fc707f8764892f3fa613c6ea24b27d159a6d29e6daa1a53297e96155022
17268915dd48c60864aca2de3f1052664452b99f41e1f7221711529eb3191b9f
```

Example hash-based CSPRNG

```
0fb23fc707f8764892f3fa613c6ea24b27d159a6d29e6daa1a53297e96155022
17268915dd48c60864aca2de3f1052664452b99f41e1f7221711529eb3191b9f
```

Entropy pool

Example hash-based CSPRNG



Example hash-based CSPRNG

- ☑ Uniform
- ☑ Unpredictable
- ☑ Unlimited

(Still, it’s an example, don’t use this in practice)

Kernel CSPRNG

/dev/urandom

(Linux)

/dev/urandom

```
$ head -c 300 /dev/urandom | xxd
0000000: f60d 4bda 67a1 83b4 d095 0db9 5366 0bb7  ..K.g.......Sf..
0000010: bf20 7474 2b62 8a61 88f5 7938 52ed f77a  . tt+b.a..y8R..z
0000020: c2e7 6fa9 3c66 2998 dc54 a6cb 8c59 caa6  ..o.<f)..T...Y..
0000030: ac37 9640 81d5 1691 09ca 1d64 6d4f 7e9f  .7.@.......dmO~.
0000040: 6749 8674 4df6 e6d3 85de 4e19 e979 63f2  gI.tM.....N..yc.
0000050: de44 09c5 d6c7 b26b 6407 35e9 5bd3 cbd6  .D.....kd.5.[...
0000060: 1a02 10b8 6111 9713 57a6 191c 5e27 601c  ....a...W...^'`.
0000070: 6965 1fc2 5798 8faf 5f6b 104f 351c b2b5  ie..W..._k.O5...
0000080: 573f 9bb9 10bf 16f6 fe0d fdff 2e49 2d86  W?.........I-.
0000090: c183 1cc1 25f1 923e 54ec e235 7ff4 db05  ....%..>T..5....
00000a0: 56bd 2b26 4e87 a7ad 6542 f01e 183c 718f  V.+&N...eB...<q.
00000b0: 7437 6f31 4af6 17e7 7870 ccc9 61e3 dd94  t7o1J...xp..a...
00000c0: 72d1 1b46 bf17 c8ed 2b67 f440 3c34 c22e  r..F....+g.@<4..
```

Kernel CSPRNG

/dev/random

(OS X, BSD)

Kernel CSPRNG

CryptGenRandom()

(Windows)

Kernel CSPRNG

>

Userspace CSPRNG

(OpenSSL, etc.)

This talk could be over now

Linux

/dev/urandom
vs.
/dev/random

/dev/[u]random

```
#define INPUT_POOL_SHIFT 12
#define INPUT_POOL_WORDS (1 << (INPUT_POOL_SHIFT-5))

static __u32 input_pool_data[INPUT_POOL_WORDS];

static struct entropy_store input_pool = {
   .poolinfo = &poolinfo_table[0],
   .name = "input",
   .limit = 1,
   .lock = __SPIN_LOCK_UNLOCKED(input_pool.lock),
   .pool = input_pool_data
};
```

pool of 4096 bit

https://github.com/torvalds/linux/blob/85051e295fb7487fd2254/drivers/char/random.c

/dev/[u]random

The pool is mixed with unpredictable bytes from various sources using a fast CRC-like hash

```
/*
 * This function adds bytes into the entropy “pool”. The pool is
 * stirred with a primitive polynomial of the appropriate degree,
 * and then twisted.  We twist by three bits at a time because
 * it’s cheap to do so and helps slightly in the expected case
 * where the entropy is concentrated in the low-order bits.
 */
static void _mix_pool_bytes(struct entropy_store *r, const void *in, int nbytes)
```

https://github.com/torvalds/linux/blob/85051e295fb7487fd2254/drivers/char/random.c

/dev/[u]random

Random numbers are generated by hashing the pool with **SHA1**

```
static void extract_buf(struct entropy_store *r, __u8 *out)
{
    sha_init(hash.w);

    /* Generate a hash across the pool, 16 words (512 bits) at a time */
    for (i = 0; i < r->poolinfo->poolwords; i += 16)
        sha_transform(hash.w, (__u8 *)(r->pool + i), workspace);

    __mix_pool_bytes(r, hash.w, sizeof(hash.w));

    hash.w[0] ^= hash.w[3]; hash.w[1] ^= hash.w[4]; hash.w[2] ^= rol32(hash.w[2], 16);

    memcpy(out, &hash, EXTRACT_SIZE);
}
```

https://github.com/torvalds/linux/blob/85051e295fb7487fd2254/drivers/char/random.c

/dev/[u]random

/dev/random and /dev/urandom use the same code, sizes and entropy sources

```
static ssize_t random_read(struct file *file, char __user *buf,
     size_t nbytes, loff_t *ppos)

     extract_entropy_user(&blocking_pool, buf, nbytes);

static ssize_t urandom_read(struct file *file, char __user *buf,
     size_t nbytes, loff_t *ppos)

     extract_entropy_user(&nonblocking_pool, buf, nbytes);
```

https://github.com/torvalds/linux/blob/85051e295fb7487fd2254/drivers/char/random.c

/dev/[u]random

Only difference:

/dev/random

- tries to guess how many bits of entropy were mixed in the pool
- decreases that number when bytes are read
- blocks if number is low

/dev/[u]random

This is **useless.**

Entropy does not decrease.

Entropy does not run out.

/dev/[u]random

Once unpredictable,
forever unpredictable.

(Unless the CSPRNG leaks secret bits)

/dev/[u]random

If CSPRNGs didn’t work (leaked secret bits):

- Stream ciphers wouldn’t work
- CTR wouldn’t work
- TLS wouldn’t work
- PGP wouldn’t work

Cryptography relies on this.

/dev/[u]random

/dev/random blocking

- is useless
- can be unacceptable (TLS)
- can be dangerous (side channel)

/dev/[u]random

/dev/urandom is safe for crypto

- Google’s BoringSSL uses it
- Python, Go, Ruby use it
- Sandstorm replaces /dev/random with it
- Cryptographers say so, too
- other OSes don’t have /dev/random

/dev/[u]random

- https://www.imperialviolet.org/2015/10/17/boringssl.html
- http://sockpuppet.org/blog/2014/02/25/safely-generate-random-numbers/
- http://www.2uo.de/myths-about-urandom/
- http://blog.cr.yp.to/20140205-entropy.html
- https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man4/random.4.html
- https://en.wikipedia.org/wiki//dev/random#FreeBSD
- https://docs.sandstorm.io/en/latest/developing/security-practices/
- http://lists.randombit.net/pipermail/cryptography/2013-August/004983.html

You don’t need /dev/random

- You don’t need to keep measuring your entropy pool
- You don’t need to “refill” the pool (like haveged, etc.)
- Random numbers “quality” does not decrease

The early boot problem

At early boot, there’s no pool yet. And you often have keys to generate.

Embedded devices, containers often fall prey to this.

The early boot problem

“Widespread Weak Keys in Network Devices”

Heninger, Durumeric, Wustrow, Halderman

https://factorable.net/

The early boot problem

“Predictable SSH host keys”

https://www.raspberrypi.org/forums/viewtopic.php?f=66&t=126892

The early boot problem

Very early at boot, /dev/urandom might not be seeded yet, predictable.

This is a Linux shortcoming. The OS must save the entropy pool at poweroff.

Your distribution probably does already.

To sum up:

CSPRNGs are cool.
You don’t need /dev/random.
Avoid userspace CSPRNG.

Use /dev/urandom

Thank you! Q/A

Doubts? Unconvinced? Ask!

Filippo Valsorda — @FiloSottile
filippo@cloudflare.com

CLOUDFLARE

Slides: https://filippo.io/entropy-talk-ccc

getrandom(2)

It’s equivalent to OpenBSD getentropy(2)

Behaves like urandom, but blocks at boot until the pool is initialized

https://lwn.net/Articles/606552/

getrandom(2)

```
#include <linux/random.h>

int getrandom(void *buf, size_t
buflen, unsigned int flags);
```

[...] when reading from /dev/urandom, it blocks if the entropy pool has not yet been initialized.

The boot problem

If you

- can’t use getrandom(2)
- run early at boot
- don’t trust your distribution

you might want to first read 1 byte from /dev/random, then use urandom

Unseeded CSPRNG

Userspace CSPRNG are more dangerous: it’s easy to entirely forget to seed them.

It happened: http://android-developers.blogspot.it/2013/08/some-securerandom-thoughts.html

Unseeded CSPRNG

“Exploiting ECDSA Failures
in the Bitcoin Blockchain”
https://speakerdeck.com/filosottile/exploiting-ecdsa-failures-in-the-bitcoin-blockchain

http://android-developers.blogspot.it/2013/08/some-securerandom-thoughts.html

Unseeded CSPRNG

Or to seed them with predictable information, like the timestamp.

"Linux Ransomware Debut Fails on Predictable Encryption Key"

http://labs.bitdefender.com/2015/11/linux-ransomware-debut-fails-on-predictable-encryption-key/

Broken CSPRNG

Between 2006 and 2008, in Debian, the OpenSSL CSPRNG was crippled, seeding only with the PID.

All outputs, keys, etc. of anything using it were easily predictable.

https://www.debian.org/security/2008/dsa-1571

Broken CSPRNG

```
--- openssl/trunk/rand/md_rand.c   2006/05/02 16:25:19 140
+++ openssl/trunk/rand/md_rand.c   2006/05/02 16:34:53 141
@@ -271,7 +271,10 @@ static void ssleay_rand_add(
      else
         MD_Update(&m,&(state[st_idx]),j);

+/*
+ * Don't add uninitialised data.
      MD_Update(&m,buf,j);
+*/
      MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
      MD_Final(&m,local_md);
      md_c[1]++;
```

Fork()ed entropy pool

The state of a userspace CSPRNG is duplicated on fork(), so the child and the parent will generate identical streams if not reseeded.

https://www.agwa.name/blog/post/libressls_prng_is_unsafe_on_linux

Non-CS PRNG

Not all PRNG are Cryptographically Secure

Normal PRNG are usually fast and uniform, but **not unpredictable.**

Non-CS PRNG

Mersenne Twister (MT19937)
Python: `random.random()`
Ruby: `Random::rand()`
PHP: `mt_rand()`

C (depends on stdlib impl, but non-CS): `rand(3)`, `random(3)`

Non-CS PRNG

For example, MT19937 core is a state of 624 numbers, and a mixing function that iterates over them.

Given an output, it’s easy to reconstruct the state number from which it was generated.

After seeing just 624 outputs, **we can predict all future outputs**.

Non-CS PRNG

```
def untemper_MT19937(y):
    x = y
    for i in range(32 // 18):
        y ^= x >> (18 * (i + 1))
    for i in range(32 // 15):
        y ^= (((y >> (i*15)) % (2**15)) << ((i+1)*15)) & 0xefc60000
    for i in range(32 // 7):
        y ^= (((y >> (i*7)) % (2**7)) << ((i+1)*7)) & 0x9d2c5680
    x = y
    for i in range(32 // 11):
        y ^= x >> (11 * (i + 1))
    return y
```

A word about HWRNG

Many CPUs now have built-in **hardware random number generators**.

Intel (RDRAND), Broadcom (on the **Raspberry Pi**), many others.

When/if loaded in the kernel, they seed the pool and every SHA1 extraction.

A word about HWRNG

```
/*
 * If we have an architectural hardware
 * random number generator, use it for SHA's
 * initial vector
 */

sha_init(hash.w);
for (i = 0; i < LONGS(20); i++) {
        unsigned long v;
        if (!arch_get_random_long(&v)) break;
        hash.l[i] = v;
}
```