crypto coding (bis)

JP Aumasson

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload); /* pl length never checked */
r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, \
                                3 + payload + padding);
```

in RFC 6520 “TLS/DTLS Heartbeat Extension“:

“TLS is based on reliable protocols, but there is not necessarily a feature available to keep the connection alive without continuous data transfer.

The Heartbeat Extension as described in this document overcomes these limitations. The user can use the new HeartbeatRequest message, which has to be answered by the peer with a HeartbeatResponse immediately.”

in RFC 6520 “TLS/DTLS Heartbeat Extension“:

“The Heartbeat protocol messages consist of their type and an arbitrary payload and padding.

The total length of a HeartbeatMessage MUST NOT exceed 2^14 or max_fragment_length when negotiated as defined in [RFC6066].”

“arbitrary” => freedom to choose payload **content and length**
=> larger attack surface

(2^14 limit not enforced by OpenSSL… 2^16 is the actual bound)

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
/* bp (buffer+3) will contain data from local memory! */
memcpy(bp, pl, payload);
/* buffer with potentially local secrets is sent back */
r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, \
                                3 + payload + padding);
```

patch

```
+    /* Read type and payload length first */
+    if (1 + 2 + 16 > s->s3->rrec.length)
+        return 0; /* silently discard */
+    hbtype = *p++;
+    n2s(p, payload);
+    if (1 + 2 + payload + 16 > s->s3->rrec.length)
+        return 0; /* silently discard per RFC 6520 sec. 4 */
+    pl = p;
+
```

sslKeyExchange.c

opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c

```
    hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,                    /* plaintext */
                       dataToSignLen,                 /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
            sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                "returned %d\n", (int)err);
            goto fail;
    }
```

```
goto fail;
goto fail;
```

the “fail” label is actually **not a fail**:

it checks err, and fails iff err != 0

sslRawVerify() being bypassed by the goto, “verification” always succeeds

Heartbleed, gotofail:
“silly bugs” by “experts”

not pure "crypto bugs", but
bugs in the crypto

missing bound check
unconditional goto

"But we have static analyzers!"

not detected

(in part due to OpenSSL's complexity)

detected

(like plenty of other unreachable code)

crypto bugs (and bugs in crypto) vs "standard" security bugs:

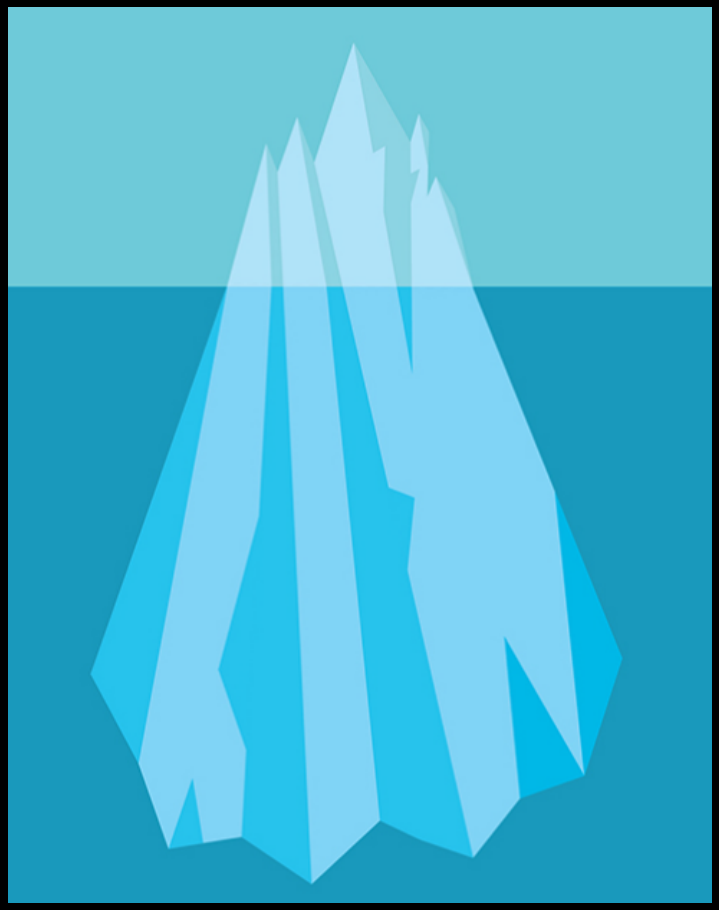
less understood

fewer experts

fewer tools

everybody uses OpenSSL, Apple

sometimes, some read the code



many more bugs in code that noone reads

Agenda

1. OpenSSL
2. guidelines for secure crypto coding
3. conclusion

"OpenSSL s****"?

AIM HIGH

What's the worst that could happen?

ASN.1 parsing, CA/CRL management

crypto: RSA, DSA, DH*, ECDH*; AES, CAMELLIA, CAST, DES, IDEA, RC2, RC4, RC5; MD2, MD5, RIPEMD160, SHA*; SRP, GCM, HMAC, GOST*, PKCS*; etc.),

PRNG, password hashing, S/MIME

X.509 certificate management, timestamping

some crypto accelerators, hardware tokens

clients and servers for SSL2, SSL3, TLS1.0, TLS1.1, TLS1.2, DTLS1.0, DTLS1.2

SNI, session tickets, etc. etc.

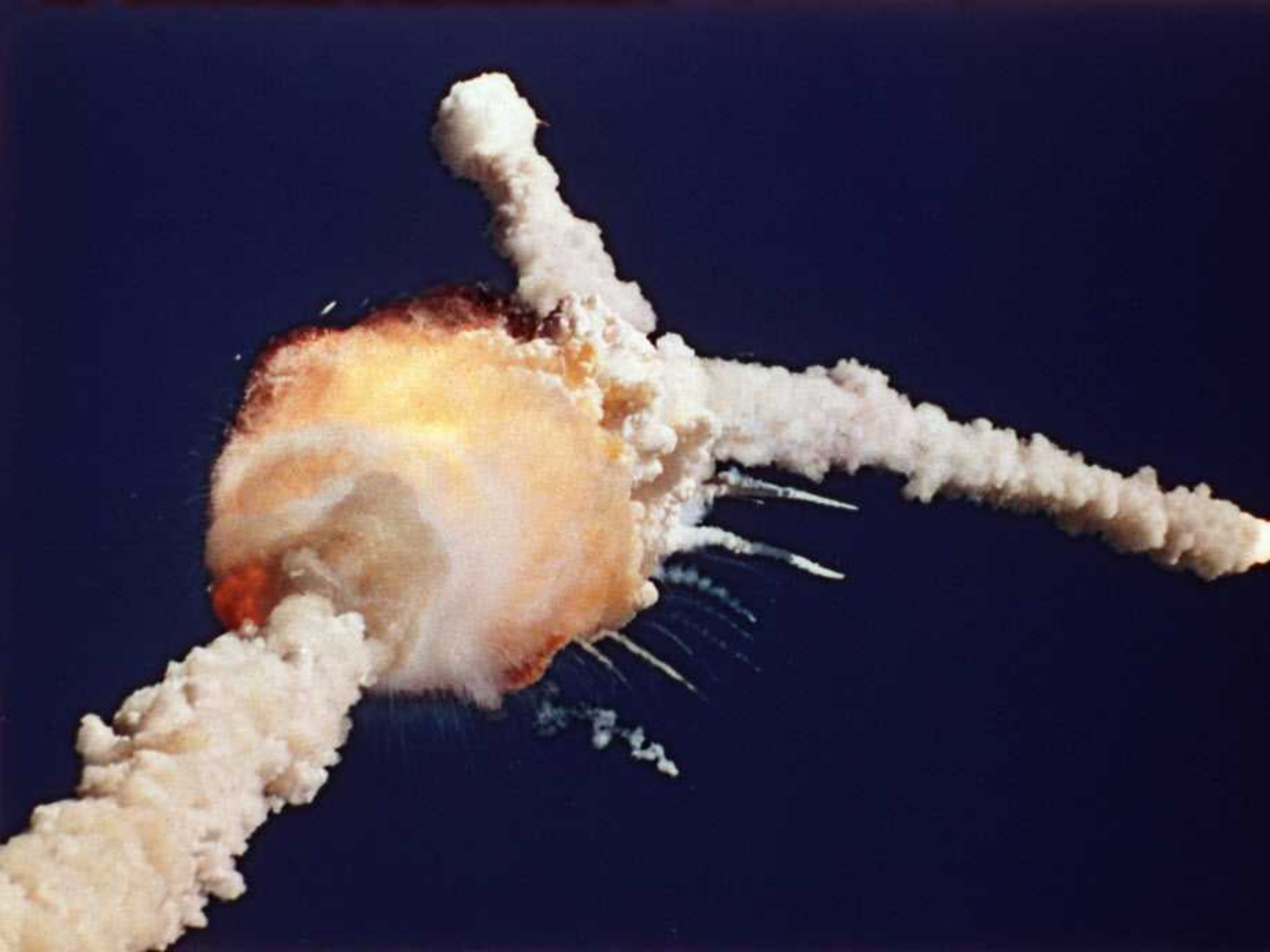
*nix
BeOS
DOS
HP-UX
Mac OS Classic
NetWare
OpenVMS
ULTRIX
VxWorks
Win* (including 16-bit, CE)

OpenSSL is the space shuttle of crypto libraries. It will get you to space, provided you have a team of people to push the ten thousand buttons required to do so.

— Matthew Green

I promise nothing complete; because any human thing supposed to be complete, must not for that very reason infallibly be faulty.

— Herman Melville, in Moby Dick



OpenSSL code

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
*bp++ = TLS1_HB_RESPONSE;
        payload, bp);
memcpy(bp, pl, payload);
r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, \
                                3 + payload + padding);
```

payload is not the payload but its length (pl is the payload)

```
#ifdef _OSD_POSIX
        /* In the BS2000-OSD POSIX subsystem, the compiler
generates
         * path names in the form "*POSIX(/etc/passwd)".
         * This dirty hack strips them to something sensible.
         * @@@ We shouldn't modify a const string, though.
         */
```

(crypto/err/err.c)

in the thread-unsafe RNG:

```
/* may compete with other threads */
state[st_idx++]^=local_md[i];
```

(crypto/rand/md_rand.c)

I TOLD YOU SO!

I have been getting a ton of requests to make more comments so here goes. I told you so, la la la, I told you so!

Joking aside, this is the worst security bug I have ever dealt with. Who knew that running crypto was **worse** than not running it at all? This is **NOT** the last catastrophic bug lurking in this code. Buyer beware, this will happen again. I was in NYC when the Internet went into full meltdown and could not respond earlier. Once things calm down I might do another round of pointing out amazing things I ran across in OpenSSL. There is no end to the amount of awe when reading through that code. For now, enjoy the old rant that is going around the tubes, again.

OpenSSL is written by monkeys

https://www.peereboom.us/assl/assl/html/openssl.html

ranting about OpenSSL is easy

don't blame the devs

let's try to understand

So, Why did "We" the community let OpenSSL happen..

Nobody Looked.

Or nobody admitted they looked.

http://www.openbsd.org/papers/bsdcan14-libressl/mgp00004.html

(slide credit: Bob Beck, OpenBSD project)

OpenSSL prioritizes

speed

portability

functionalities

at the price of "best efforts" and "dirty tricks"...

/* Quick and dirty OCSP server: read in and parse input request */

/* Quick, cheap and dirty way to discard any device and directory

/* kind of dirty hack for Sun Studio */

#ifdef STD_ERROR_HANDLE /* what a dirty trick! */

/* Dirty trick: read in the ASN1 data into a STACK_OF (ASN1_TYPE):

of lesser priority

usability

security

consistency

robustness

Who should design cryptographic libraries

In order to create a proper SSL/TLS implementation you need to be a master of:

- Cryptographic algorithms.
- Cryptographic practice.
- Software engineering.
- Software optimization.
- The language(s) used.
- Domain specific knowledge.

http://insanecoding.blogspot.gr/2014/04/libressl-good-and-bad.html

crypto by "real programmers" often yields cleaner code, but dubious choices of primitives and/or broken implementations (cf. messaging apps)

it's probably unrealistic to build a better secure/fast/usable/consistent/certified **toolkit+lib** in reasonable time

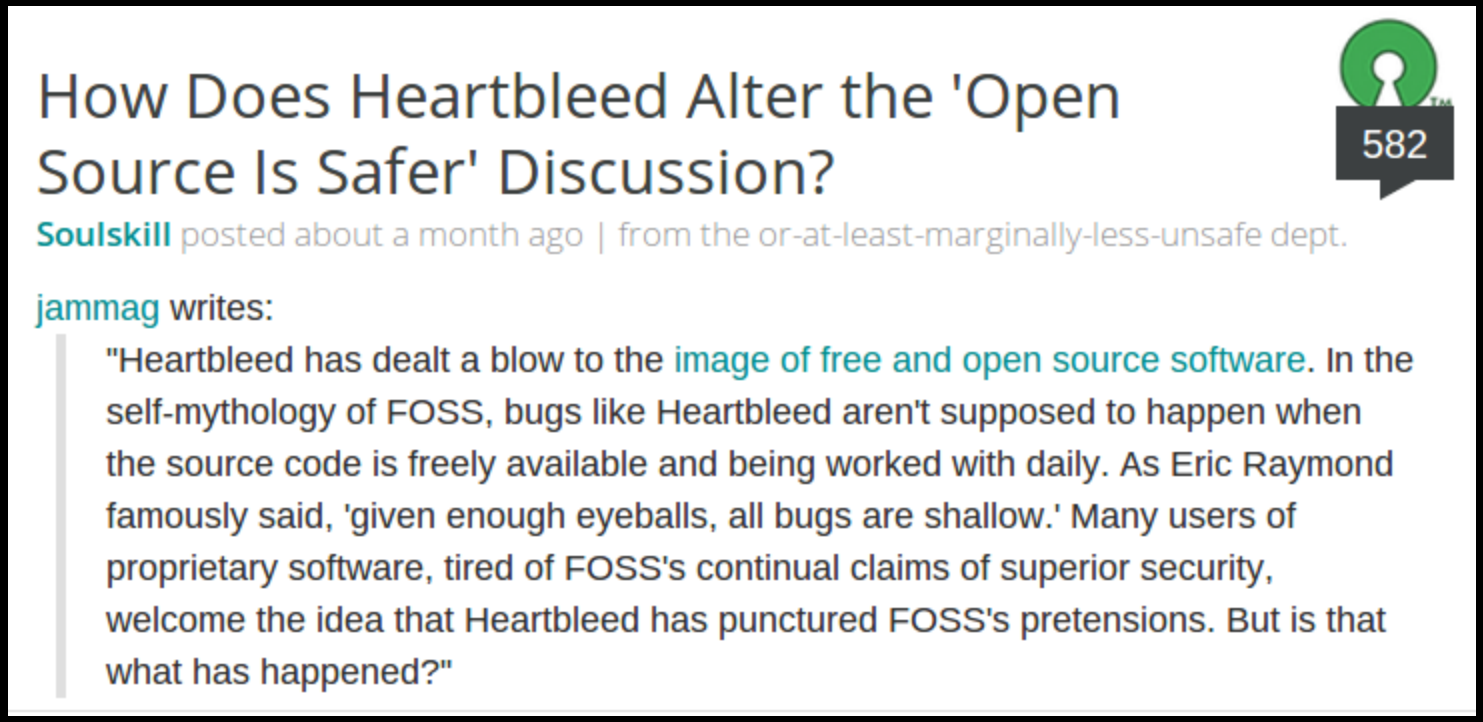
what are the alternatives?

Really better? (maybe TLS itself is the problem?)

Implementation	Developed by	Open source	Software license	Copyright owner
Botan	Jack Lloyd	Yes	Simplified BSD License	Jack Lloyd
cryptlib	Peter Gutmann	Yes	Sleepycat License and commercial license	Peter Gutmann
CyaSSL	wolfSSL	Yes	GPLv2 and commercial license	wolfSSL Inc.
GnuTLS	GnuTLS project	Yes	LGPL	Free Software Foundation
MatrixSSL	PeerSec Networks	Yes	GPLv2 and commercial license	PeerSec Networks
Network Security Services		Yes	Mozilla Public License	NSS contributors
OpenSSL	OpenSSL project	Yes	OpenSSL / SSLeay dual-license	Eric Young, Tim Hudson, Sun, OpenSSL project, and others
PolarSSL	Offspark	Yes	GPLv2 and commercial license	Brainspark B.V. (brainspark.nl)
SChannel	Microsoft	No	Proprietary	Microsoft Inc.
Secure Transport	Apple Inc.	Yes	APSL 2.0	Apple Inc.
SharkSSL	Realtimelogic LLC[1]	No	Proprietary	Realtimelogic LLC
JSSE	Oracle	Yes	GPLv2 and commercial license	Oracle
Bouncy Castle	The Legion of the Bouncy Castle Inc.	Yes	MIT License	Legion of the Bouncy Castle Inc.
LibreSSL	OpenBSD	Yes	OpenSSL / SSLeay dual-license	Eric Young, Tim Hudson, Sun, OpenSSL project, and others

http://en.wikipedia.org/wiki/Comparison_of_TLS_implementations

let's just use closed-source code!



How Does Heartbleed Alter the 'Open Source Is Safer' Discussion?

582

Soulskill posted about a month ago | from the or-at-least-marginally-less-unsafe dept.

jammag writes:

> "Heartbleed has dealt a blow to the image of free and open source software. In the self-mythology of FOSS, bugs like Heartbleed aren't supposed to happen when the source code is freely available and being worked with daily. As Eric Raymond famously said, 'given enough eyeballs, all bugs are shallow.' Many users of proprietary software, tired of FOSS's continual claims of superior security, welcome the idea that Heartbleed has punctured FOSS's pretensions. But is that what has happened?"

It’s not just OpenSSL, it’s not an open-source thing.

— Bob Beck

open- vs. closed-source software security:

- well-known debate
- no definite answer, depends on lots of factors; see summary on http://en.wikipedia.org/wiki/Open-source_software_security

for **crypto**, OSS has a better track record

- better assurance against "backdoors"
- flaws in closed-source can often be found in a "black-box" manner

LibreSSL

LibreSSL is a **FREE** version of the SSL/TLS protocol forked from OpenSSL

At the moment we are too busy deleting and rewriting code to make a decent web page. No we don't want help making web pages, thank you.

http://www.libressl.org/

initiative of the OpenBSD community

big progress in little time (lot of code deleted)

adoption unclear if it remains BSD-centric

ports expected, but won't leverage BSD security features

OpenSSL patches unlikely to directly apply

how to write secure crypto code?

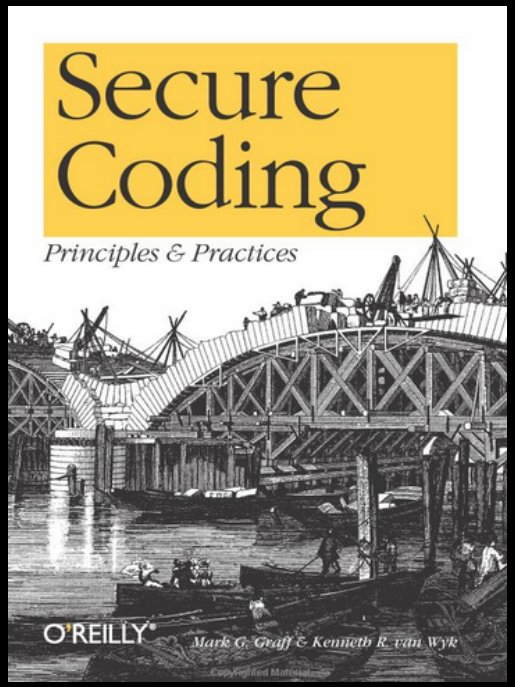
write secure code!

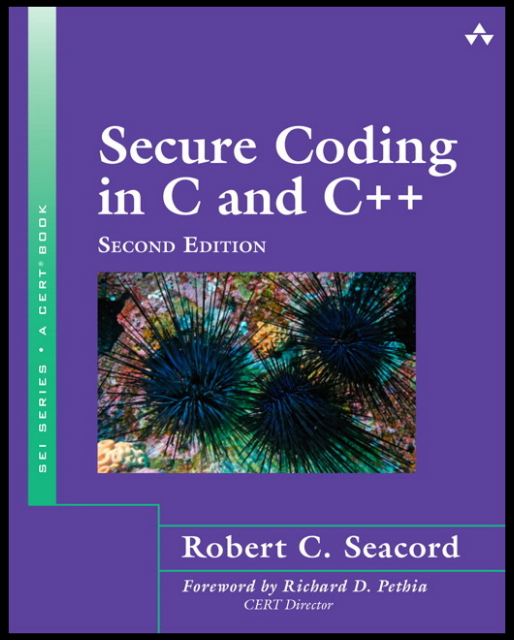
The Power of Ten
10 Rules for Writing Safety Critical Code

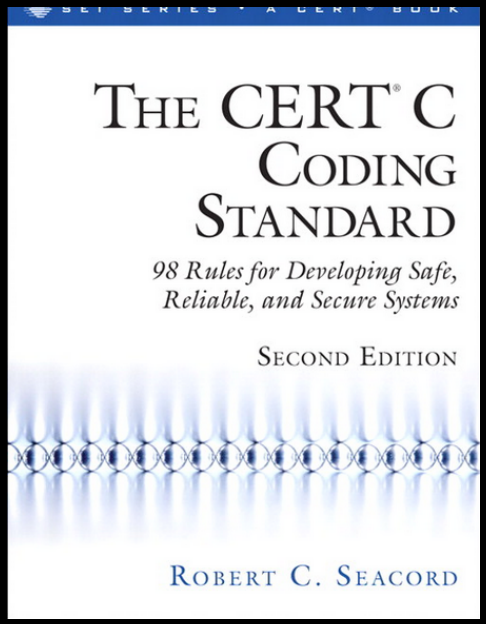
1. Restrict to simple control flow constructs.
2. Give all loops a fixed upper-bound.
3. Do not use dynamic memory allocation after initialization.
4. Limit functions to no more than 60 lines of text.
5. Use minimally two assertions per function on average.
6. Declare data objects at the smallest possible level of scope.
7. Check the return value of non-void functions, and check the validity of function parameters.
8. Limit the use of the preprocessor to file inclusion and simple macros.
9. Limit the use of pointers. Use no more than two levels of dereferencing per expression.
10. Compile with all warnings enabled, and use one or more source code analyzers.

Based on: "The Power of Ten -- Rules for Developing Safety Critical Code," *IEEE Computer*, June 2006, pp. 93-95 (PDF).

http://spinroot.com/p10/


Secure
Coding
Principles & Practices
O'REILLY
Mark G. Graff & Kenneth R. van Wyk


SEI SERIES • A CERT® BOOK
Secure Coding
in C and C++
Second Edition
Robert C. Seacord
Foreword by Richard D. Pethia
CERT Director


The CERT® C
Coding
Standard
98 Rules for Developing Safe,
Reliable, and Secure Systems
Second Edition
Robert C. Seacord

etc.

write secure crypto!

=

defend against algorithmic attacks, timing attacks, "misuse" attacks, etc.

?

the best list I found: in NaCl [salt]

Branches

Do not use secret data to control a branch. In particular, do not use the `memcmp` function to compare secrets. Instead use `crypto_verify_16`, `crypto_verify_32`, etc., which perform constant-time string comparisons.

Even on architectures that support fast constant-time conditional-move instructions, always assume that a comparison in C is compiled into a branch, not a conditional move. Compilers can be remarkably stupid.

Array lookups

Do not use secret data as an array index.

Early plans for NaCl would have allowed exceptions to this rule inside primitives specifically labelled `vulnerable`, in particular to allow fast `crypto_stream_aes128vulnerable`, but subsequent research showed that this compromise was unnecessary.

Dynamic memory allocation

Do not use heap allocators (`malloc`, `calloc`, `sbrk`, etc.) or variable-size stack allocators (`alloca`, `int x[n]`, etc.) in C NaCl.

http://nacl.cr.yp.to/internals.html

so we tried to help

Jpa Talk Preferences Watchlist Contribu

Page Discussion Read Edit View history Search G

Cryptography Coding Standard

Navigation

Main page
Coding rules
References
FAQ

Toolbox

What links here
Related changes

Welcome to the Cryptography Coding Standard homepage.

The Cryptography Coding Standard (CCS) is a set of coding rules to prevent the most commo weaknesses in software cryptographic implementations. CCS was first presented and discuss Internet crypto workshop on Jan 23, 2013 (our slides are available).

The following pages are available:

- Coding rules: the list of coding rules, with for each rule a statement of the problem address or more proposed solutions
- References: a list of external references
- FAQ: the usual Q&As page

These pages can also be accessed with the navigation bar on the left.

https://cryptocoding.net

with help from Tanja Lange, Nick Mathewson, Samuel Neves, Diego F. Aranha, etc.

we tried to make the rules simple,
in a **do-vs.-don't** style

secrets should be kept secret

=

do not leak information on the secrets
(timing, memory accesses, etc.)

compare strings in constant time

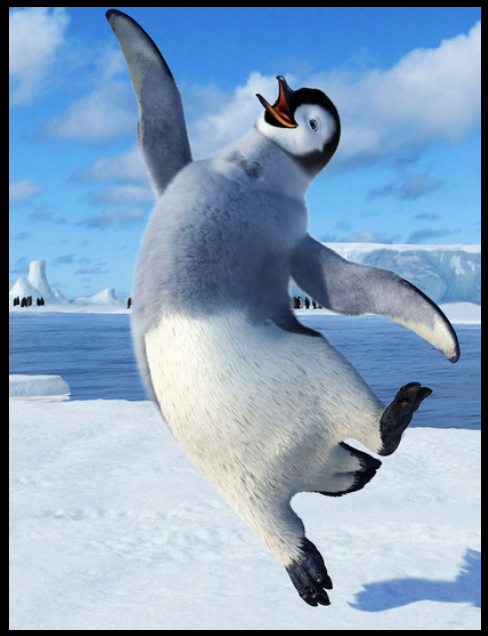
Microsoft C runtime library memcmp implementation:

```
EXTERN_C int __cdecl memcmp(const void *Ptr1, const void *Ptr2, size_t
Count) {
    INT v = 0;
    BYTE *p1 = (BYTE *)Ptr1;
    BYTE *p2 = (BYTE *)Ptr2;

    while(Count-- > 0 && v == 0) {
        v = *(p1++) - *(p2++);
       /* execution time leaks the position of the first difference */
       /* may be exploited to forge MACs (cf. Google Keyczar's bug) */
    }

    return v;
```

compare strings in constant time

Constant-time comparison function

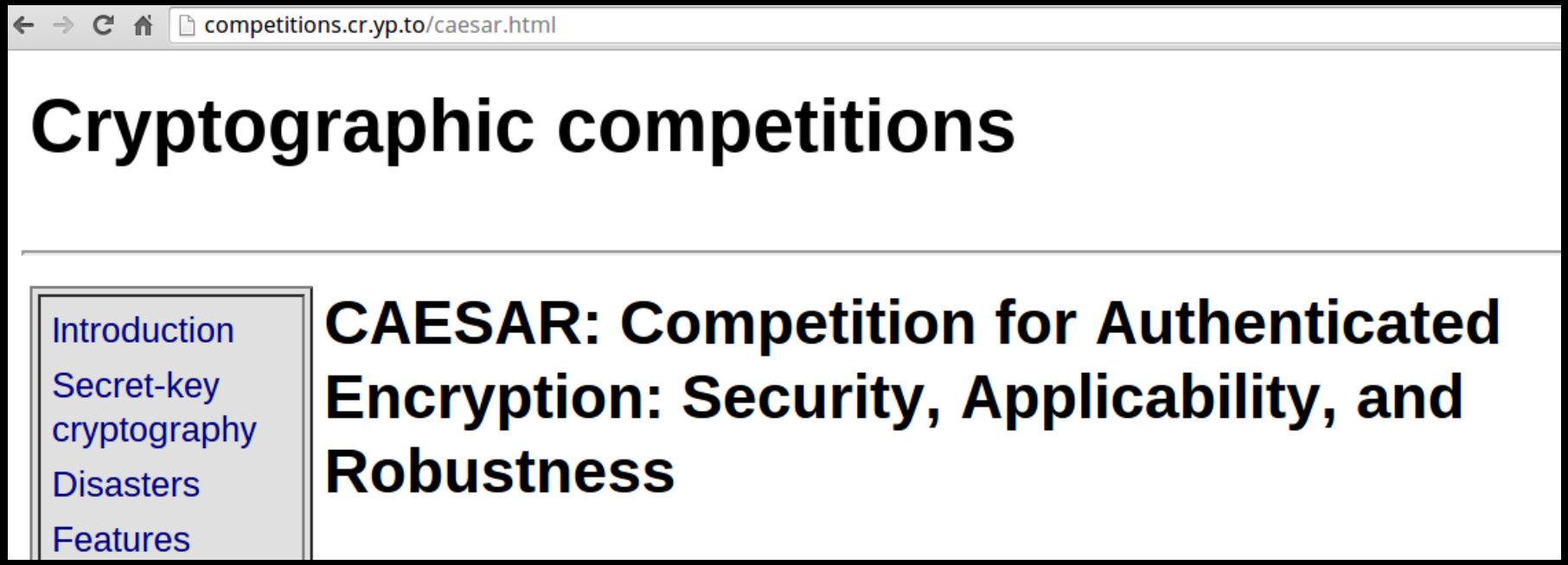
```
int util_cmp_const(const void * a, const void *b, const size_t size) {
  const unsigned char *_a = (const unsigned char *) a;
  const unsigned char *_b = (const unsigned char *) b;
  unsigned char result = 0;
  size_t i;

  for (i = 0; i < size; i++)
    result |= _a[i] ^ _b[i];

 /* returns 0 if equal, nonzero otherwise */
  return result;
}
```



compare strings in constant time

counter-examples in CAESAR candidates

(collected by Samuel Neves)



Ascon

```
// return -1 if verification fails
for (i = 0; i < klen; ++i)
  if (c[clen - klen + i] != S[rate + klen + i])
    return -1;
```

Julius

```
for(cur = reg_index; cur < reg_index + BLK_SIZE + pres;
cur++){
        if(pad[cur] != 0){
            return -1;
        }
    }
```

PAEQ

```
for(unsigned i=0; i<CRYPTO_ABYTES; ++i) {
    if(c[(*mlen)+i] != Tag[i]) {
        for(unsigned j=0; j< (*mlen); ++j)//Erasing decryption result
            m[j]=0;
        return -1; //Invalid
    }
}
```

Enchilada

```
/* see if we have a match */
if( memcmp((void *) accum, (void *) ptr, AUTH_BYTES) != 0)
    return -2;
```

Joltik

```
/* If the tag does not match, return error */
if(0!=memcmp(Final, ciphertext+c_len-8, 8)) return -1;
```

Etc. etc.

avoid branchings
(or make them constant-time)

standard square-and-multiply exponentiation:

$x = m$
for $i = n - 2$ **downto** 0
  $x = x^2$
  if $(k_j == 1)$ **then**
    $x = x \cdot m$
endfor
return x

several attacks in the 1998 paper by Dhem et al.
http://users.belgacom.net/dhem/papers/CG1998_1.pdf

avoid table look-ups

fast AES implementations use large LUTs…

Efficient Cache Attacks on AES, and Countermeasures

Eran Tromer[1,2], Dag Arne Osvik[3] and Adi Shamir[2]

Cache Games – Bringing Access-Based Cache Attacks on AES to Practice

Endre Bangerter
Bern University of Applied Sciences
endre.bangerter@bfh.ch

David Gullasch
Bern University of Applied Sciences, Dreamlab Technologies
david.gullasch@bfh.ch

Stephan Krenn
Bern University of Applied Sciences, University of Fribourg
stephan.krenn@bfh.ch

solutions for AES: NIs, bitslicing, SIMD ops, etc.

avoid potential timing leaks

make

- branchings
- loop bounds
- table lookups
- memory allocations

independent of secrets or user-supplied value
(private key, password, heartbeat payload, etc.)

prevent compiler interference with security-critical operations

Tor vs MS Visual C++ 2010 optimizations

```
int
crypto_pk_private_sign_digest(...)
{
  char digest[DIGEST_LEN];
  (...) /* operations involving secret digest */
  memset(digest, 0, sizeof(digest));
  return r;
}
```

a solution: C11’s memset_s()

clean memory of secret data
(keys, round keys, internal states, etc.)

Data in stack or heap may leak through crash dumps, memory reuse, hibernate files, etc.

Windows' SecureZeroMemory()

OpenSSL's OPENSSL_cleanse()

```
void burn( void *v, size_t n )
{
  volatile unsigned char *p = ( volatile unsigned char * )v;
  while( n-- ) *p++ = 0;
}
```

last but not least

RANDOMNESS

You never saw it coming.

Randomness everywhere

key generation and key agreement

symmetric encryption (CBC, etc.)

RSA OAEP, El Gamal, (EC)DSA

side-channel defenses

etc. etc.

Netscape, 1996: ~ **47-bit** security thanks to

```
RNG_GenerateRandomBytes() {
  return (..) /* something that depends only on
      • microseconds time
      • PID and PPID */
}
```

Mediawiki, 2012: **32-bit** Mersenne Twister seed

```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;
}
```

*nix: /dev/urandom

example: get a random 32-bit integer

```
int randint, bytes_read;
int fd = open("/dev/urandom", O_RDONLY);
if (fd != -1) {
  bytes_read = read(fd, &randint, sizeof(randint));
  if (bytes_read != sizeof(randint))  return -1;
}
else { return -2; }
printf("%08x\n", randint);
close(fd);
return 0;
```

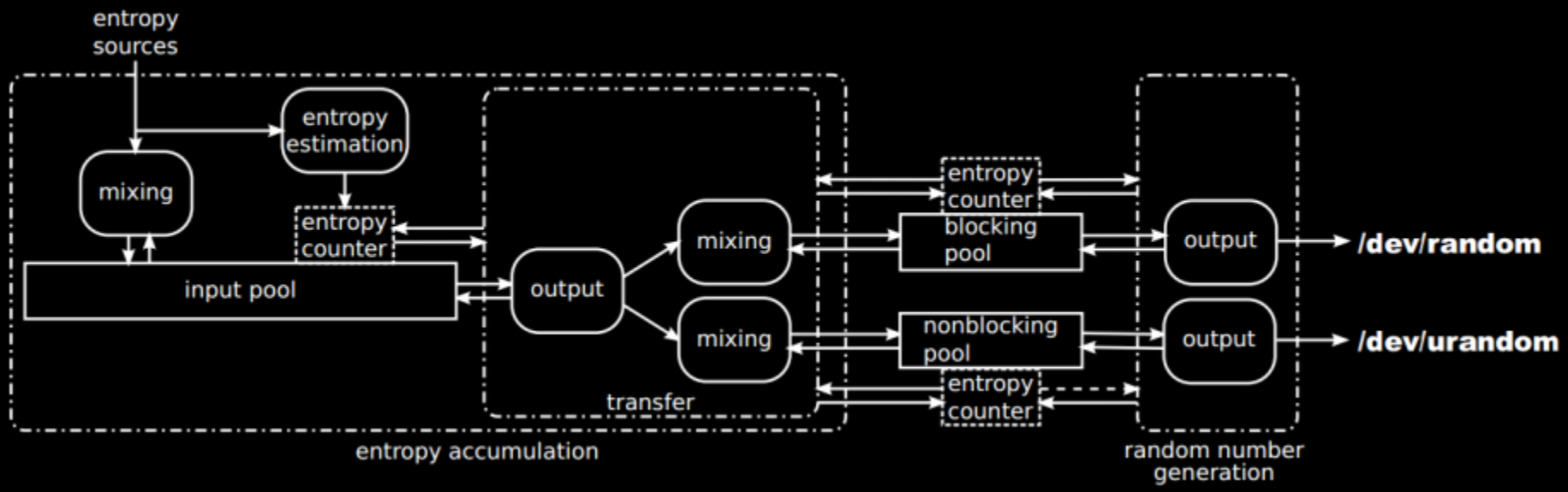
(ideally, there should be a syscall for this)

what is /dev/urandom?

device file probing analog sources to gather entropy and generate random bytes

implemented differently on different OS' (Linux, FreeBSD, OpenBSD, etc.)

/dev/urandom in Linux



entropy from keyboard/mouse/interrupts/disk

4kB entropy pool, linear internal mixing

postprocessing using SHA-1

“but /dev/random is better! it blocks!”

/dev/random may do more harm than good to your application, since

- blockings may be mishandled
- /dev/urandom is safe on reasonable OS’

Win*: CryptGenRandom

(based on the CTR_DRBG of NIST SP 800-90)

```
int randombytes(unsigned char *out, size_t outlen) {
  static HCRYPTPROV handle = 0;
  if(!handle) {
    if(!CryptAcquireContext(&handle, 0, 0, PROV_RSA_FULL,
                            CRYPT_VERIFYCONTEXT | CRYPT_SILENT))
      return -1;
  }
  while(outlen > 0) {
    const DWORD len = outlen > 1048576UL ? 1048576UL : outlen;
    if(!CryptGenRandom(handle, len, out)) { return -2; }
    out += len;
    outlen -= len;
  }
  return 0;
}
```

it’s possible to fail in many ways, and *appear* to succeed in many ways

non-uniform sampling

no forward secrecy

randomness reuse

poor testing

etc.

Thou shalt:

1. compare secret strings in constant time
2. avoid branchings controlled by secret data
3. avoid table look-ups indexed by secret data
4. avoid secret-dependent loop bounds
5. prevent compiler interference with security-critical operations
6. prevent confusion between secure and insecure APIs
7. avoid mixing security and abstraction levels of cryptographic primitives in the same API layer
8. use unsigned bytes to represent binary data
9. use separate types for secret and non-secret information
10. use separate types for different types of information
11. clean memory of secret data
12. use strong randomness

Learn the rules like a pro, so you can break them like an artist.

— Pablo Picasso

conclusion

let’s stop the blame game

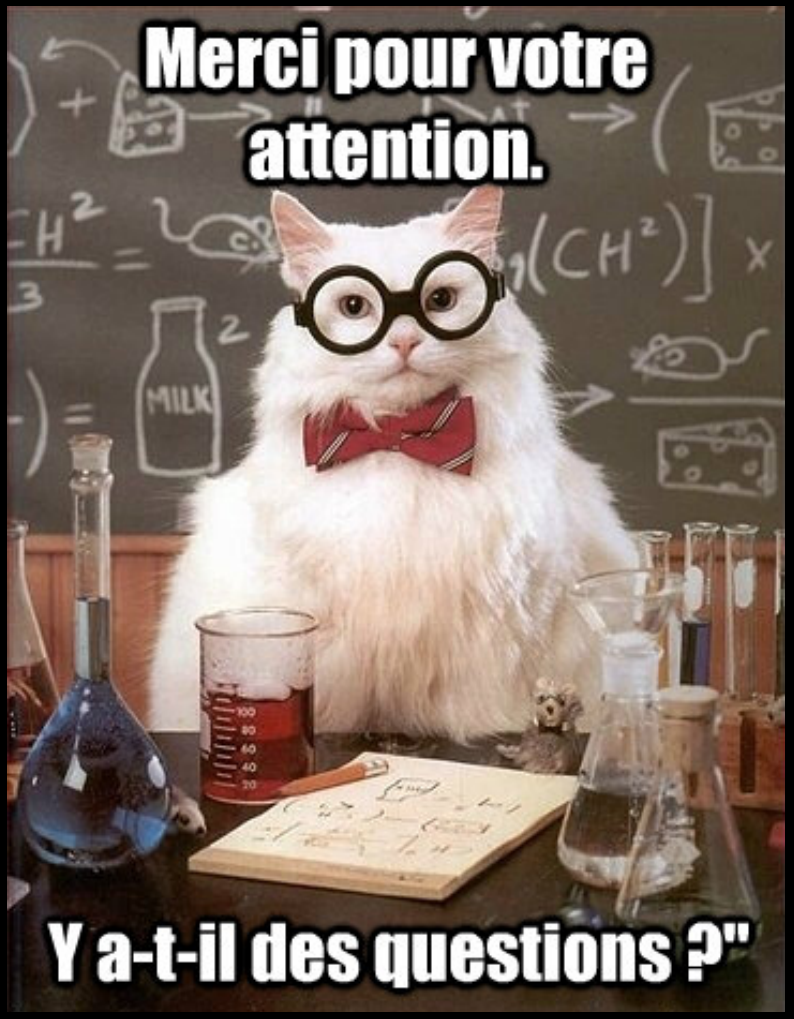
(OpenSSL, “developers”, “academics”, etc.)

researchers

add disclaimers to experimental code
(or it may be copypasted to a production code base)

try to learn secure (crypto) coding practices

publish your code! (GitHub etc.)


Merci pour votre attention.
MILK
Y a-t-il des questions ?"