

cryptocoding



JP Aumasson



@veorq / <http://aumasson.jp>

academic background

principal cryptographer at Kudelski Security, .ch

applied crypto research and outreach

BLAKE, BLAKE2, SipHash, NORX

Crypto Coding Standard

Password Hashing Competition

Open Crypto Audit Project board member

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, \
                     3 + payload + padding);
```





bugs are bad
software crashes, incorrect output, etc.



crypto bugs are really bad
leak of private keys, secret documents,
past and future communications, etc.

threats to
individuals' privacy, sometimes lives
organizations' strategies, IP, etc.

```
sslKeyExchange.c      x  100%
```

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;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(ctx,
                    ctx->peerPubKey,
                    dataToSign,
                    dataToSignLen,
                    signature,
                    signatureLen);
/* plaintext */
/* plaintext length */

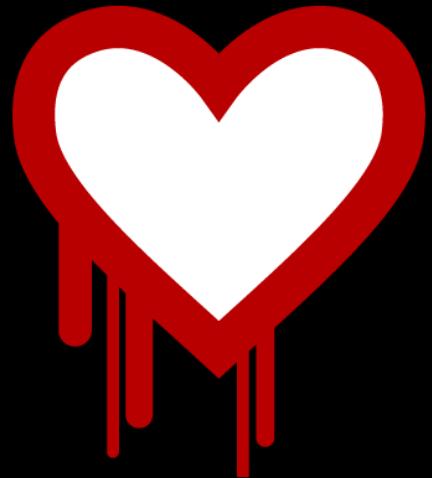
if(err) {
    sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
               "returned %d\n", (int)err);
    goto fail;
}
```

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)



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

detected

(like plenty of other unreachable code)

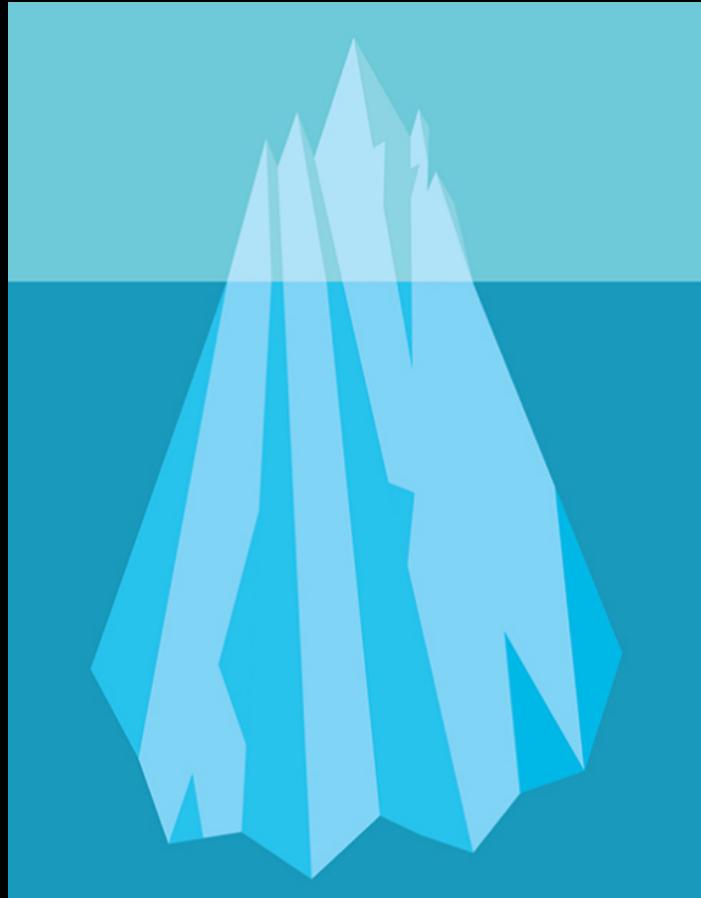
crypto bugs (and bugs in crypto)
vs "standard" security bugs:

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. the poster child: OpenSSL
2. secure crypto coding guidelines
3. conclusion

"OpenSSL s**"?**



A I M H I G H

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,
CCM, GCM, HMAC, GOST*, PKCS*,
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

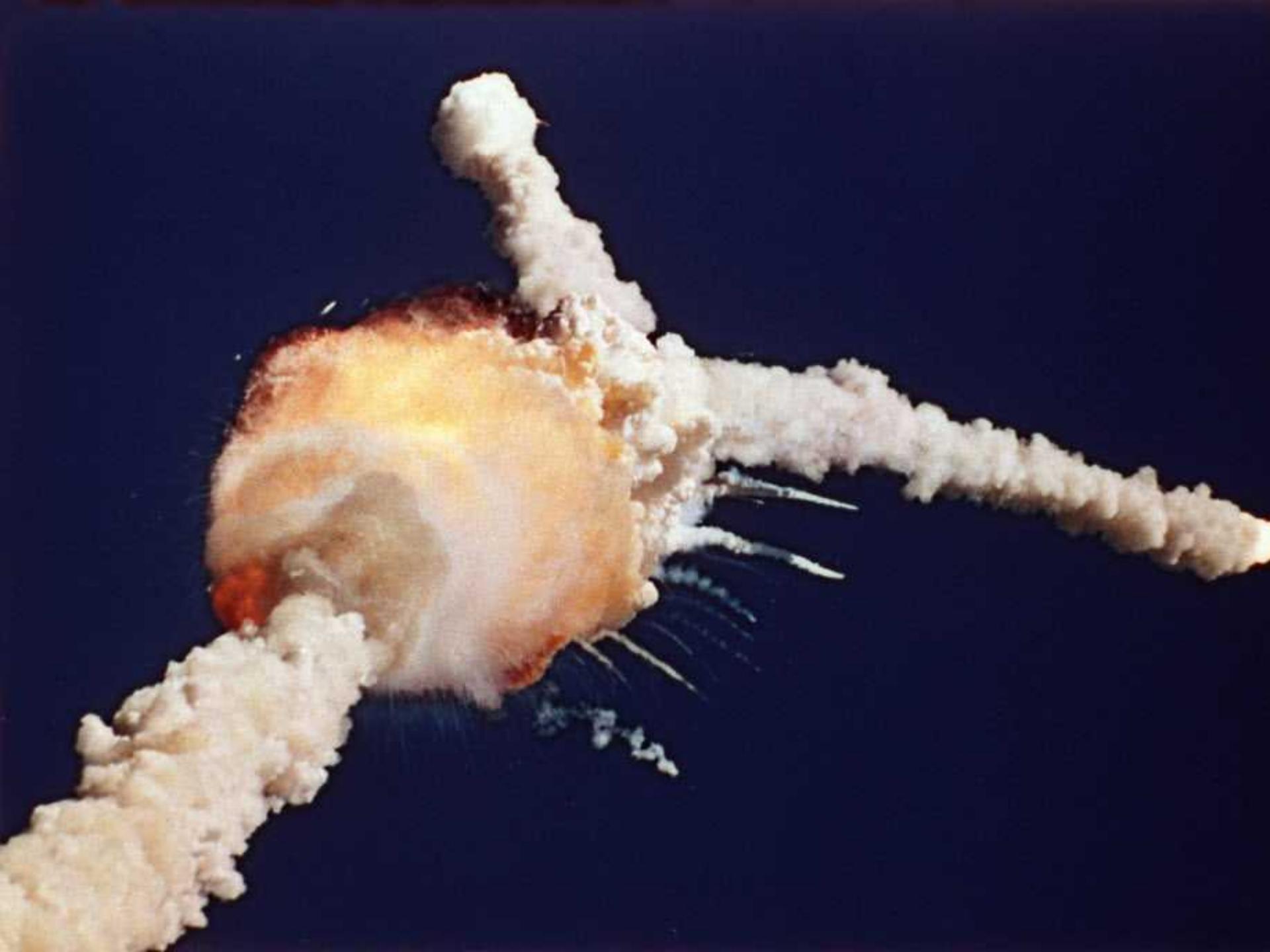
Win* (including 16-bit, CE)

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)

courtesy of @OpenSSLFact (Matt Green)

```
/* BIG UGLY WARNING! This is so damn  
ugly I wanna puke ... ARGH! ARGH! ARGH!  
Let's get rid of this macro package. Please?
```

```
/* HAS BUGS! DON'T USE - this is only  
present for use in des.c */  
void DES_3cbc_encrypt(...)
```

```
#else  
if (0)  
;  
#endif
```

```
"user_pwd = NULL; /* abandon responsibility
```

```
/* FIXME: the cast of the function seems  
unlikely to be a good idea */  
(void)BIO_set_info_callback(dbio,  
(bio_info_cb *)data->info_callback)
```

1. Is OpenSSL thread-safe?

Yes (with limitations: an SSL connection may not concurrently be used by many Unix systems, OpenSSL automatically uses the multi-threaded version of the library). If your platform is not one of these, consult the INSTALL file.

in the RNG:

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

(crypto/rand/md_rand.c)

RFC 5246: TLS 1.2

1. Generate a string R of 46 random bytes  (1)
2. Decrypt the message to recover the plaintext M  (2)
3. If the PKCS#1 padding is not correct, or the length of message M is not exactly 48 bytes:

```
pre_master_secret = ClientHello.client_version || R
else If ClientHello.client_version <= TLS 1.0, and version
number check is explicitly disabled:
    pre_master_secret = M
else:
    pre_master_secret = ClientHello.client_version || M[2..47]
```

 (3)

OpenSSL 1.0.1c

```
i=RSA_private_decrypt((int)n,p,p,rsa,RSA_PKCS1_PADDING);  (2)

al = -1;

  
  
if (al != -1)
{
    /* Some decryption failure -- use random value instead as countermeasure
     * against Bleichenbacher's attack on PKCS #1 v1.5 RSA padding
     * (see RFC 2246, section 7.4.7.1). */
    ERR_clear_error();
    i = SSL_MAX_MASTER_KEY_LENGTH;
    p[0] = s->client_version >> 8;
    p[1] = s->client_version & 0xff;
    if (RAND_pseudo_bytes(p+2, i-2) <= 0)  (1)
        /* should be RAND_bytes, but we cannot work around a failure */
        goto err;
}

s->session->master_key_length=
    s->method->ssl3_enc->generate_master_secret(s,
        s->session->master_key,
        p,i);  (3)
```

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

we should not 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?)

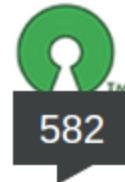
Implementation	Developed by	Open source	Software license	Copyright owner
Botan	Jack Lloyd	Yes	Simplified BSD License	Jack Lloyd
cryptlib	Peter Gutmann	Yes	Sleepycat License and commercial license	Peter Gutmann
CyaSSL	wolfSSL	Yes	GPLv2 and commercial license	wolfSSL Inc.
GnuTLS	GnuTLS project	Yes	LGPL	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?

[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?

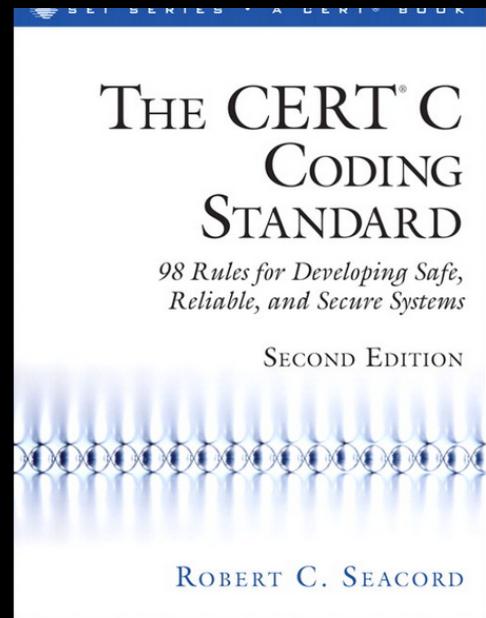
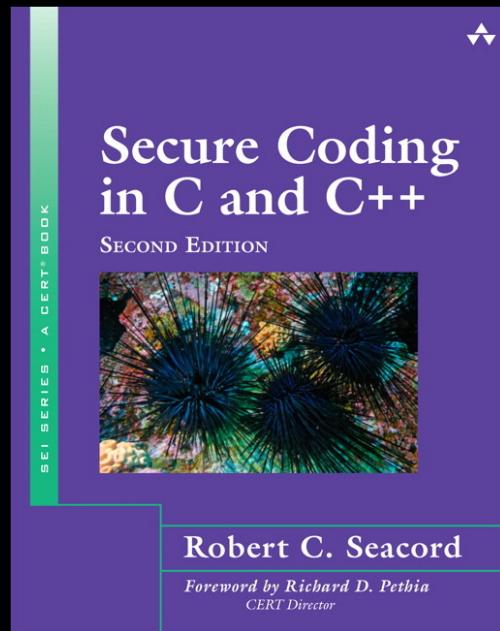
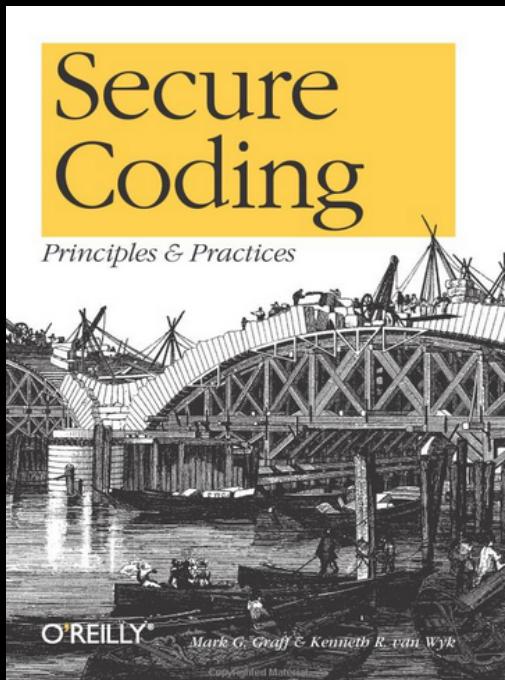
write secure code!

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](#)).



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





Page

Discussion

Read

Edit

View history

▼

Search

G

Cryptography Coding Standard

Welcome to the Cryptography Coding Standard homepage.

The Cryptography Coding Standard (CCS) is a set of coding rules to prevent the most common weaknesses in software cryptographic implementations. CCS was first presented and discussed at the [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 addressed 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) */
```

```
}
```

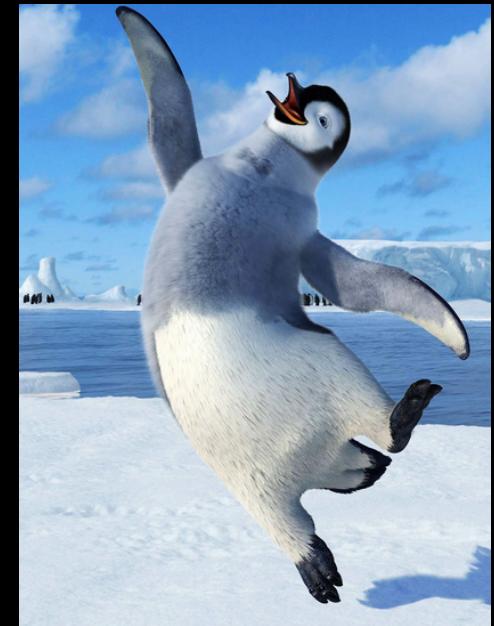
```
return v;
```



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;  
}
```



avoid other 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



R A N D O M N E S S

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

```
/**  
 * 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 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';  
    $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)

“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

```
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.)

cryptographers (and scientists, etc.)

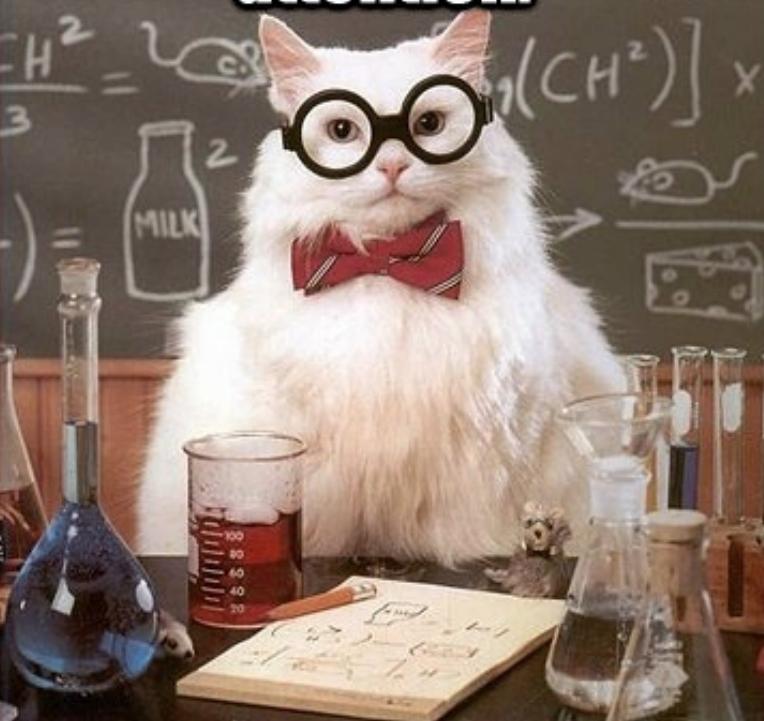
- acknowledge that you **suck at coding**
- either go the extra mile and learn, or
- get help from a real programmer

programmers

- acknowledge that you **suck at crypto**
- either go the extra mile and learn, or
- get help from a real cryptographer

in any case: get **third-party reviews/audits!**

**Merci pour votre
attention.**



Y a-t-il des questions ?"