BearSSL: SSL for all Things

Thomas Pornin

BSides Edinburgh, April 7th, 2017

Outline

- Why yet another SSL library?
- SSL attacks and defences
- Constant-time implementations
- Constrained RAM, streaming and buffering
- X.509 certificate validation
- Why SSL sucks and how to fix it

SSL

A family of protocols:

- Uses a *reliable* bidirectional transport for bytes (e.g. TCP).
- Provides a *secure* bidirectional transport for bytes.
- Used in HTTPS, SMTP, FTPS, some VPN…
- Netscape: SSL 1.0, 2.0 and 3.0
- IETF: TLS 1.0, 1.1, 1.2 (draft 1.3)

We use “SSL” to designate SSL 3.0 to TLS 1.2.

SSL Handshake

```
Client                                               Server

ClientHello                 -------->
                                                ServerHello
                                               Certificate*
                                         ServerKeyExchange*
                                        CertificateRequest*
                            <--------       ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                    -------->
                                         [ChangeCipherSpec]
                            <--------              Finished
Application Data            <------->      Application Data
```

Things


Kwikset
MY LOCKS & KEYS
Home - Front Door
Home - Back Door
Beach House


easybulb
easybulb

Unfulfilled Needs

An SSL/TLS library that:

- is correct and secure (TLS 1.2, modern crypto...);
- works with very little RAM;
- has a small ROM footprint;
- has no OS dependency;
- is compatible with an embedded C world.

BearSSL

- Written from scratch in C.
- State-machine API, streamed processing.
- No `malloc()`.
- Should fit in about 25 kB RAM.
- Static linking model, down to about 20 kB code (minimal server).

BearSSL

Extra Goals

- Pluggable crypto (optimised, constant-time...).
- Clean documented structure, and comments.
- Reusable opensource.
- Support for many cipher suites and features.
- Should work well on big machines as well.

BearSSL

Secure Crypto

- RSA (up to 4096 bits).
- ECC (P-256, P-384, P-521, X25519).
- ChaCha20+Poly1305.
- AES/GCM and AES/CBC.
- Legacy support for SHA-1, 3DES.

SSL Attacks

SSL Attacks

Version Rollback

- Attacker forces client and server to negotiate a lower version than what they both support.
- Requires the client to do something “stupid”.
- Modern protection: `TLS_FALLBACK_SCSV`
  - Sent by client when downgrading.
  - Allows server to detect undue downgrade.

SSL Attacks

RSA: Bleichenbacher Attack

RSA key exchange (encryption):

- $m = \texttt{00 02 xx xx} \ldots \texttt{xx 00} \,||\, \textit{pre-master}$
- $z = m^e \pmod{n}$

Decryption:

- $m = z^d \pmod{n}$
- Check and remove padding.

SSL Attacks

RSA: Bleichenbacher Attack

Attacker sends carefully crafted, invalid messages z and expects the server to respond differently when the padding is valid.

Solution: when decryption fails, use a random value.

SSL Attacks

rsa_ssl_decrypt.c

```
x = core(data, sk);
x &= EQ(data[0], 0x00);
x &= EQ(data[1], 0x02);
for (u = 2; u < (len - 49); u ++) {
        x &= NEQ(data[u], 0);
}
x &= EQ(data[len - 49], 0x00);
memmove(data, data + len - 48, 48);
return x;
```

ssl_hs_server.t0

```
x = (*ctx->policy_vtable)->do_keyx(
        ctx->policy_vtable, epms, &len);
br_enc16be(epms, ctx->client_max_version);
br_hmac_drbg_generate(&ctx->eng.rng, rpms, sizeof rpms);
br_ccopy(x ^ 1, epms, rpms, sizeof rpms);
```

SSL Attacks

Forward Secrecy

If an attacker steals a server private key, he can decrypt past recorded sessions.

Solution: use *ephemeral* keys for key exchange.

- Server generates new Diffie-Hellman key pair.
- Server *signs* its DH public key.
- Server "forgets" its DH private key after use.

SSL Attacks

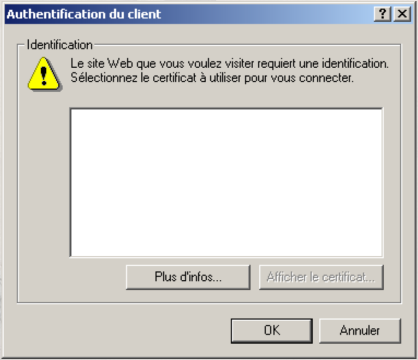
Forward Secrecy

Some issues:

- Performance: `TLS_ECDH_ECDSA` requires one point multiplication, `TLS_ECDHE_ECDSA` needs three.
- Larger code (ECDH *and* ECDSA).
- Extra ServerKeyExchange message.

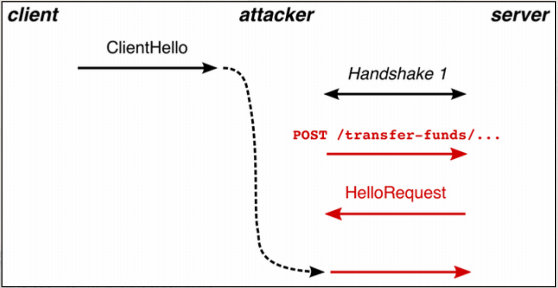
SSL Attacks

Secure Renegotiation



SSL Attacks

Secure Renegotiation



SSL Attacks

Secure Renegotiation

Solution 1: Secure Renegotiation extension (RFC 5746)

- Extension in ClientHello, distinguishes between first handshake and subsequent handshakes.
- BearSSL refuses renegotiations without the extension.

Solution 2: reject all renegotiations

- Use flag `BR_OPT_NO_RENEGOTIATION`.

SSL Attacks

Bad (EC)DHE Parameters

DHE: server sends p, g and $g^s \pmod{p}$. Client responds with $g^c \pmod{p}$. Shared secret is $g^{sc} \pmod{p}$.

ECDHE: server selects curve E, with generator G, and sends sG. Client responds with cG. Shared secret is scG.

SSL Attacks

Bad (EC)DHE Parameters

- Client cannot validate DHE parameters (e.g. p is not prime, order of g has small divisors...).
- Client may send wrong values to obtain information about server secret (if server reuses that secret):
  - Low-order value not in the subgroup.
  - Point not on the curve.

SSL Attacks

Bad (EC)DHE Parameters

Countermeasures in BearSSL:

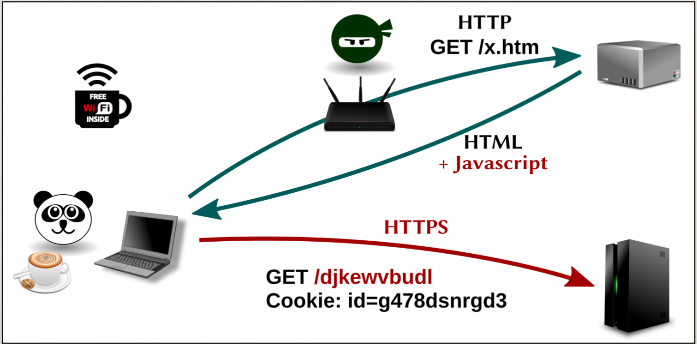
- No DHE support, only ECDHE.
- Only known, named curves.
- No secret reuse (*ephemeral*: we mean it).
- Validation of incoming curve points:

$$Y^2 = X^3 + aX + b$$

(Overhead: about +0.5%)

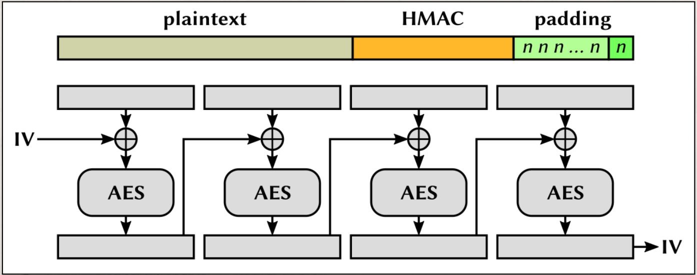
SSL Attacks

Chosen-Plaintext and the Web



SSL Attacks

CBC Woes



SSL Attacks

CBC Woes

POODLE: in SSL 3.0, padding bytes can have arbitrary values. Attacker replaces last block with another encrypted block to test an hypothesis on the last plaintext byte.

- Attacker injects some plaintext to "phase" record for a full-length padding block.
- If peer does not mind, then last decrypted byte was equal to 15.

Solution: don't support SSL 3.0; use TLS 1.0+ only.

SSL Attacks

CBC Woes

Padding Oracle: attacker modifies the last two blocks and tries to know whether the *padding* was correct (not the MAC).

- Explicit error message (Vaudenay 2002).
- Timing (recomputation of HMAC).
- Lucky13: timing again (*length* of HMAC source data).

SSL Attacks

CBC Woes

Solution:

- Constant-time padding check.
- Always compute HMAC.
- Constant-time HMAC computation (even with regards to *length* of data).
- Report generic error, only at the end.

SSL Attacks

```
v = 0;
for (u = min_len; u < max_len; u ++) {
        tmp1[v] |= MUX(GE(u, len_nomac) & LT(u, len_withmac),
                buf[u], 0x00);
        rot_count = MUX(EQ(u, len_nomac), v, rot_count);
        if (++ v == cc->mac_len) {
                v = 0;
        }
}
/* ... */
for (i = 5; i >= 0; i --) {
        uint32_t rc;

        rc = (uint32_t)1 << i;
        cond_rotate(rot_count >> i, tmp1, cc->mac_len, rc);
        rot_count &= ~rc;
}
```

SSL Attacks

BEAST

In TLS 1.0, IV for next record is last block from previous record.

- Attacker sends long request, observes IV x.
- Attacker sends plaintext $x \oplus y$, observes $E(y)$.
- This tests an hypothesis on y given $E(y)$.
- Cookie recovery, byte by byte.

SSL Attacks

BEAST

Solution 1: use TLS 1.1+ (per-record random IV).

Solution 2: the $1/n-1$ split.

- When sending a record with n bytes, send *two* records with 1 and $n-1$ bytes, respectively.
- This reuses the HMAC output on first record as IV randomization.
- Do this only for application data records (compatibility issues).

SSL Attacks

CRIME

Encryption hides *contents* but not *length*. Compression makes length depend on contents.

Solution: don't compress.

SSL Attacks

SWEET32

"Bad things" happen when you encrypt more than $2^{n/2}$ blocks with a block cipher with *n*-bit blocks.

SWEET32: encrypt hundreds of gigabytes with 3DES. Collisions reveal cookie elements.

<u>Solution:</u> don't use 3DES if you can avoid it.

SSL Attacks

Weak Crypto is Weak

- "Export" cipher suites, with 40-bit encryption meant to be breakable (it works!).
- 512-bit RSA (FREAK).
- 512-bit DHE (Logjam).

Solution: don't do that.

Constant-Time Cryptography

Constant-Time Cryptography

Timing attacks are side-channel attacks than can be exploited remotely (over a network).

- Algorithmic (conditional execution).
- Cache-based (lookup tables, code path).
- Non-constant-time opcodes.

Constant-Time Cryptography

Constant-Time RSA

Classical square-and-multiply leaks secret key information.

Solution 1: use random masking.

$$r^{-1}(mr^e)^d = m^d \pmod{n}$$

Solution 2: always multiply, use a constant-time conditional copy (BearSSL).

Constant-Time Cryptography

```
if (win_len > 1) {
        uint64_t *base;

        memset(t2, 0, mw62num * sizeof *t2);
        base = t2 + mw62num;
        for (u = 1; u < ((uint32_t)1 << k); u ++) {
                uint64_t mask;
                size_t v;

                mask = -(uint64_t)EQ(u, bits);
                for (v = 0; v < mw62num; v ++) {
                        t2[v] |= mask & base[v];
                }
                base += mw62num;
        }
}
```

Constant-Time Cryptography

```
for (i = 0; i < k; i ++) {
        montymul(t1, x, x, m, mw62num, m0i);
        memcpy(x, t1, mw62num * sizeof *x);
}
montymul(t1, x, t2, m, mw62num, m0i);
mask1 = -(uint64_t)EQ(bits, 0);
mask2 = ~mask1;
for (u = 0; u < mw62num; u ++) {
        x[u] = (mask1 & x[u]) | (mask2 & t1[u]);
}
```

Constant-Time Cryptography

Cache-Based Attacks

- Algorithm makes secret-dependent memory accesses, that hit various cache lines.
- Attacker then times its own read accesses, that exercise the same cache lines, and sees which have been evicted.
- Can work from another process or even another virtual machine.
- Lab demonstrations against AES, RSA, ECC...

Constant-Time Cryptography

Cache-Based Attacks

Microarchitecture defence: extra accesses to hit other cache lines.

- Fast and cheap.
- Fragile, can break on other hardware versions.

"True" constant-time: no secret-dependent memory access.

- Also no secret-dependent conditional jump.

Constant-Time Cryptography

Bitslicing

(Re)discovered by Biham in 1997.

- Decompose algorithm into a circuit with boolean operations.
- One data bit per variable.
- With 64-bit registers, compute 64 instances in parallel.

Constant-Time Cryptography

Bitslicing

Operation: XOR *x* with *y* (6-bit values), then rotate left by 1 bit.

```
/* classical */
z = x ^ y;
z = ((z << 1) & 31)
    | (z >> 5);
```

```
/* bitslice */
z1 = x0 ^ y0;
z2 = x1 ^ y1;
z3 = x2 ^ y2;
z4 = x3 ^ y3;
z5 = x4 ^ y4;
z0 = x5 ^ y5;
```

Constant-Time Cryptography

Bitslicing

Advantages:

- Uses the full register width.
- Data routing (e.g. rotations) is free.
- Naturally constant-time.

Constant-Time Cryptography

Bitslicing

Disadvantages:

- Larger code.
- More RAM/register traffic (expensive on non-multiscalar architectures).
- Lookup tables become complicated circuits.
- Copes poorly with non-parallel contexts (e.g. CBC encryption).

Constant-Time Cryptography

Bitslicing

Mixed strategies: use bitslicing between similar operations within a single algorithm instance (e.g. 16 identical S-boxes in an AES round).

- Less total state, so a better fit in registers.
- Better at non-parallelism.
- Some routing is no longer free.
- In BearSSL: `aes_ct`, `aes_ct64`, `des_ct`

Constant-Time Cryptography

Tricky Opcodes

- Memory accesses and conditional jumps
- Integer divisions
- Shifts and rotations
- Multiplications

`https://www.bearssl.org/ctmul.html`

Streaming and Buffering

Streaming and Buffering

ClientHello

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

Streaming and Buffering

X.509 Certificate

```
Certificate  ::=  SEQUENCE  {
     tbsCertificate       TBSCertificate,
     signatureAlgorithm   AlgorithmIdentifier,
     signatureValue       BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
     extensions      [3]  EXPLICIT Extensions OPTIONAL
     }
```

Streaming and Buffering

Buffering

Solution 1: buffering.

- Maximum message / certificate size: 16 MB.
- In practice: several kilobytes.
- OpenSSL uses a maximum 64 kB buffer.

Streaming and Buffering

Callbacks

Solution 2: streaming with callbacks.

- Decode "on the fly".
- Use callback functions to obtain new data.
- Typical of OOP languages (e.g. Java, C#).
- Blocking operations (needs threads).
- Uses more stack space.

Streaming and Buffering

Coroutines

Solution 3: run decoder in a coroutine.

- Decoder is "on the fly" in its own dedicated interruptible context.
- Library offers state-machine API (push/pull network and application data).
- Supports parallel runs (`select()` / `poll()`).

Problem: standard C does not support coroutines.

Streaming and Buffering

State-Machine API

```
unsigned char *br_ssl_engine_sendapp_buf(
        const br_ssl_engine_context *cc, size_t *len);
void br_ssl_engine_sendapp_ack(br_ssl_engine_context *cc, size_t len);

unsigned char *br_ssl_engine_recvapp_buf(
        const br_ssl_engine_context *cc, size_t *len);
void br_ssl_engine_recvapp_ack(br_ssl_engine_context *cc, size_t len);

unsigned char *br_ssl_engine_sendrec_buf(
        const br_ssl_engine_context *cc, size_t *len);
void br_ssl_engine_sendrec_ack(br_ssl_engine_context *cc, size_t len);

unsigned char *br_ssl_engine_recvrec_buf(
        const br_ssl_engine_context *cc, size_t *len);
void br_ssl_engine_recvrec_ack(br_ssl_engine_context *cc, size_t len);
```

Streaming and Buffering

T0

Standard C does not have coroutines.

- Can be done on many architectures with a bit of in-line assembly or dark tricks with `longjmp()`.
- Not portable.
- Requires an extra stack (+4 kB).

Solution: create a new language.

Streaming and Buffering

T0

- Forth dialect, with very non-Forth features.
- Separate interpreter/compiler (written in C#).
- Runtime: interpreter loop (*token-threaded code*).
- General metaprogramming.
- Coroutines.
- Static stack usage analysis.

Streaming and Buffering

```
: process-alerts ( -- bool )
      0
      begin has-input? while read8-native process-alert-byte or repeat
      dup if 1 addr-shutdown_recv set8 then ;

: process-alert-byte ( x -- bool )
      addr-alert get8 case
            0 of
                  dup 1 <> if drop 2 then
                  addr-alert set8 0
            endof
            1 of
                  0 addr-alert set8
                  dup 100 = if 256 + fail then
                  0=
            endof
            \ Fatal alert implies context termination.
            drop 256 + fail
      endcase ;
```

Streaming and Buffering

T0

Static analysis: compute stack depth at any point.

- Restriction on computing model (no recursion).
- Infers or verifies stack usage.
- No data type analysis (all values are 32-bit words).

```
[src/x509/asn1.t0]
[src/x509/x509_minimal.t0]
main: ds=17 rs=25
code length:   2778 byte(s)
data length:    286 byte(s)
total words: 200 (interpreted: 139)
```

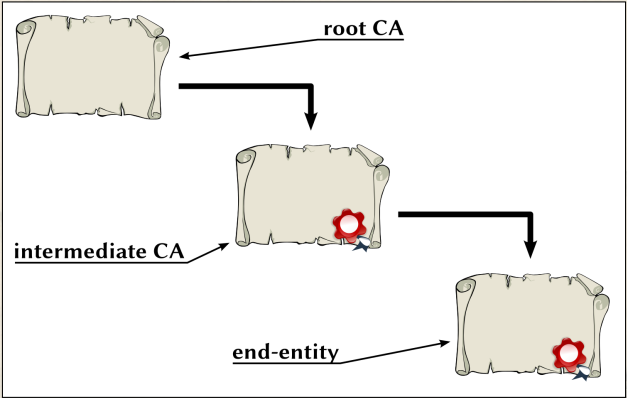
Streaming and Buffering

```
: read-length ( lim -- lim length )
        read8
        \ Lengths in 0x00..0x7F get encoded as a single byte.
        dup 0x80 < if ret then

        \ If the byte is 0x80 then this is an indefinite length, and we
        \ do not support that.
        0x80 - dup ifnot ERR_X509_INDEFINITE_LENGTH fail then

        \ Masking out bit 7, this yields the number of bytes over which
        \ the value is encoded. Since the total certicate length must
        \ fit over 3 bytes (this is a consequence of SSL/TLS message
        \ format), we can reject big lengths and keep the length in a
        \ single integer.
        { n } 0
        begin n 0 > while n 1- >n
                dup 0x7FFFFF > if ERR_X509_INNER_TRUNC fail then
                8 << swap read8 rot +
        repeat ;
```

X.509 Certificates



X.509 Certificates

BearSSL has a pluggable support for X.509 certificate validation:

- Input: the certificate chain from the peer (by chunks).
- Output: a public key, or an error code.
- Two provided implementations:
  - `br_x509_knownkey`
  - `br_x509_minimal`

X.509 Certificates

br_x509_knownkey

- Peer public key is already known.
- Certificate chain is ignored.
- Implements a security model close to SSH.

X.509 Certificates

br_x509_minimal

- Validates chain as sent (no path rebuilding).
- Stops on matching trust anchor (both CA and "direct trust").
- Checks:
  - Subject/issuer DN equality.
  - Expiration dates.
  - Basic Constraints.
  - Key Usage.

X.509 Certificates

`br_x509_minimal`

Name Extraction:

- Elements from subjectDN and from SAN extension.
- Normalisation to UTF-8.
- SAN: email address, DNS name, URI, and arbitrary `otherName` (e.g. Microsoft's UPN).
- Server name match: exact, and with a leading wildcard.

X.509 Certificates

`br_x509_minimal`

Features NOT supported:

- Revocation (CRL, OCSP).
- Path building (AIA extension).
- Name constraints.
- Certificate policies.

(Unsupported critical extensions imply validation failure.)

SSL Sucks

Large Buffers

- Records may contain up to 16 kB of plaintext.
- No clear half-duplex policy, so shared input/output buffer may be difficult.
- Max Fragment Length (RFC 6066): unusable:
  - Client-driven only.
  - Same maximum length in both directions.
  - Very few implementations support it.

SSL Sucks

Legacy Cruft

- Non-AEAD cipher suites.
- Cipher suites mix concepts (`ECDH_RSA`…).
- Forced buffering (hash function choice).
- Renegotiations.
- Asynchronous alerts, but synchronous closure.

SSL Sucks

Other Issues

- X.509.
- Length+value nested structures.
- Modern emphasis on the Web:
  - TLS 1.3 cookies, session tickets, new Certificate message structure.
  - Enforced ECDHE.
  - Non-streamable Ed25519 and Ed448 (in certificates).

SSL Sucks

Fixing SSL

SSL for the embedded world:

- Start with TLS 1.2, with AEAD cipher suites.
- Use known key model when possible.
- Normalise on SHA-256 only.
- Use smaller buffers on both sides.

In the long run: new protocol with easier encoding.

Questions?