A messy state of the union: Taming the Composite State Machines of TLS

http://smacktls.com

Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Jean Karim Zinzindohoue
Agile Cryptographic Protocols

Modern protocols negotiate crypto parameters
- Many key exchanges (RSA, DHE, PSK)
- Many authentication mechanisms (Cert, Password)
- Many encryption schemes (AEAD, RC4-HMAC)
- Much of the complexity of TLS, IKEv2, SSH is in the composition of these mechanisms

How do we implement such protocols correctly?
- What can go wrong? Can we prove them correct?
Transport Layer Security (1994—)

The default secure channel protocol?
HTTPS, 802.1x, VPNs, files, mail, VoIP, …

20 years of attacks, and fixes
1994  Netscape’s Secure Sockets Layer
1996  SSL3
1999  TLS1.0 (RFC2246)
2006  TLS1.1 (RFC4346)
2008  TLS1.2 (RFC5246)
2015  TLS1.3?

Many implementations
OpenSSL, SecureTransport, NSS, SChannel, GnuTLS, JSSE, PolarSSL, …

many bugs, attacks, patches every year

Many security theorems
mostly for small simplified models of TLS
TLS protocol overview

**Hello**
- Protocol negotiation
  - Agree on version
  - Agree on ciphersuite
  - Determines all crypto algos

**KEM**
- Authenticated Key Exchange
  - Verify server/client identity
  - Generate master secret
  - Derive connection keys

**Finished**
- Key, transcript confirmation
  - Completes authentication
  - Matches transcripts
  - Authenticated encryption

**AppData**
- Application data streams
  - Full duplex channel
  - Authenticated encryption
**RSA Key Transport**

**Hello**
- Client sends **cr**
- Server sends **sr**
- Client and server exchange fresh nonces

**KEM**
- Server certificate **cert_s** supports RSA encryption
- Client generates **pms**
- **ms**, keys **(k)** derived from **pms**, **cr**, **sr**

**Finished**
- **ae(0||tag_c,k)**
- **tag_c**, **tag_s** derived from **ms** + SHA-256 hash of handshake log

**AppData**
- **ae(i||d_i,k)**
- Authenticated encryption
(EC)DHE Key Exchange

**Hello**
- Client sends `cr`
- Server sends `sr`
- Server waits for `cr`
- Server waits for `sr`
- Server sends `cert_S`
- Server sends `rsa-sign((G, g^y), sk_S)`
- Server sends `g^x`
- Server waits for `ae(0||tag_C, k)`
- Server waits for `ae(0||tag_S, k)`

**KEM**
- Server picks group/curve
- Server signs group, key share
- Server computes `pms = g^{xy}`
- Server computes `ms, keys (k)` derived from `pms, cr, sr`

**Finished**
- Client sends `ae(0||tag_C, k)`
- Server sends `ae(0||tag_S, k)`
- Client and server exchange fresh nonces

**AppData**
- Authenticated encryption

**Notes**
- TLS 1.2 (Google’s cipher suite)
- Server picks group/curve
- Server signs group, key share
- Server computes `pms = g^{xy}`
- Server computes `ms, keys (k)` derived from `pms, cr, sr`
- `tag_C, tag_S` derived from `ms + SHA-256 hash of handshake log`
- Authenticated encryption
Composing Key Exchanges

ClientHello\( (v, [kx_1, kx_2, \ldots]) \)

\( \text{RSA} \)

ServerHello\( (v, kx = \text{RSA}) \)

ServerCertificate\( (certs) \)

ServerHelloDone

ClientKeyExchange\( (rsaenc(pms, pk_s)) \)

ClientCCS

ClientFinished\( (mac(log, pms)) \)

ServerCCS

ClientFinished\( (mac(log', pms)) \)

ApplicationData

(\( EC \))DHE

ServerHello\( (v, kx = \text{DHE|ECDHE}) \)

ServerCertificate\( (certs) \)

ServerKeyExchange\( (sign((G, g^v), sk_s)) \)

ServerHelloDone

ClientKeyExchange\( (g^x) \)

ClientCCS

ClientFinished\( (mac(log, g^sv)) \)

ServerCCS

ServerFinished\( (mac(log', g^s)) \)

ApplicationData

ServerCCS

ServerFinished\( (mac(log', \cdots)) \)

ApplicationData
Can this proof technique be applied to OpenSSL?

TLS State Machine

RSA + DHE + ECDHE + Session Resumption + Client Authentication

- Covers most features used on the Web
- Composition proved secure for miTLS implementation [IEEE S&P’13, CRYPTO’14]
  http://mitls.org
- Reference code written for verification, in F#

State machine for common Web configurations
OpenSSL State Machine

+ Fixed_DH
+ DH_anon
+ PSK
+ SRP
+ Kerberos
+ * _EXPORT
+ …

We cannot ignore all these because they share code/keys with RSA/DHE
Fuzzing TLS

Does OpenSSL conform to the miTLS state machine?

- There are known attacks if it doesn’t [EarlyCCS 2014]

We built a test framework

- FlexTLS, based on miTLS
- Generates 100s of non-conforming traces from a state machine specification
- We tested many TLS libraries
Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, ...

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?
Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, …

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?
TLS specifies a ladder diagram with optional messages

- Handshake ends with agreement on transcript

---

**Figure 1.** Message flow for a full handshake
Composing with Optional Messages

Treat ServerKeyExchange as optional
• Server decides to send it or not
• Client tries to handle both cases
• Consistent with Postel’s principle: “be liberal in what you accept”

Unexpected cases at the client
• Server skips ServerKeyExchange in DHE
• Server sends ServerKeyExchange in RSA

Clients should reject these cases
• In practice: clients accept and perform unexpected cryptographic computations, breaking the security of TLS
Network attacker impersonates S.com to a Java TLS client

1. Send S’s cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone
4. SKIP ServerCCS (bypass encryption)
5. Send ServerFinished using uninitialized MAC key (bypass handshake integrity)
6. Send ApplicationData (unencrypted) as S.com
TLS 1.0 supported weakened ciphers to comply with export regulations in 1990s

- RSA keys limited to 512 bits
- Export keys are sent in a signed ServerKeyExchange
- Client uses the 512-bit key instead of S’s public key

EXPORT deprecated in 2000

- (Dead) code still exists in OpenSSL and many other libraries
- Can be triggered by sending an unexpected ServerKeyExchange
A man-in-the-middle attacker can:

- impersonate servers that support RSA_EXPORT,
- at buggy clients that allow ServerKeyExchange in RSA.

**Diagram Details:**

- **Client Hello:**
  - ClientHello($c_r$, $[\ldots, \text{RSA}, \ldots]$)
  - ServerHello($s_r$, RSA)

- **MitM:**
  - ClientHello($c_r$, [RSA_EXPORT])
  - ServerHello($s_r$, RSA_EXPORT)
  - ServerCertificate($cert_S$)
  - ServerKeyExchange($\text{sign}(c_r | s_r | p_{512}, sk_S)$)

- **Client Key Exchange:**
  - ClientKeyExchange($\text{rsaenc}(pms, p_{512})$)
  - $(m_s, k_1, k_2) = \text{kdf}(pms, c_r | s_r)$
  - $s_{512} = \text{factor}(p_{512})$

- **Log:**
  - $log'_{C}$

- **Client CCS:**
  - ClientFinished($\text{mac}(log'_C, m_s)$)
  - ServerCCS
  - $\text{ServerFinished}(\text{mac}(log'_C, m_s))$
  - authenc($k_1, \text{Data}$)
  - authenc($k_2, \text{Data'}$)
FREAK: Exploit and Impact

Many servers in 2015 offer RSA_EXPORT
• 37% of browser-trusted servers in March 2015
• After FREAK: came down to 6.5% [Zmap team, 2015]
• See: www.smacktls.com/#freak
• Vulnerable sites included nsa.gov, hsbc.com, …

Factoring 512-bit RSA keys is easy
• First broken with CADO-NFS in 2000 [EuroCrypt’00]
• Now: 12 hours and $100 on Amazon EC2 [N. Heninger]

Client-side state machine bugs are widespread
• Same bug in SChannel, SecureTransport, IBM JSSE, …
• CVEs for all major libraries and web browsers
A Verified State Machine for OpenSSL
A Verified State Machine for OpenSSL

OpenSSL has two state machines (client/server)
  • A bit of a mess: many protocol versions, extensions, optional, and experimental features

We rewrote this code and verified it with Frama-C
  • 750 lines of code, 460 lines of specification
  • 1 month of a PhD student’s time
  • Reused logical specification from miTLS
  • Eliminates all state machine bugs in OpenSSL
  • No impact on performance.
Conclusions

Cryptographic protocol testing needs work
• We used a specification-driven fuzzing tool to find critical state machine bugs in a number of libraries
• This should be done systematically by developers

Open source code is not immune from attack
• Security bugs can hide in plain sight for years

Verification of production code is feasible
• We focused on the core state machine, one small step towards verifying OpenSSL

Beware of deliberately weakened cryptography
• Backdoors come back to bite you even decades later