Skip, Freak, and Logjam: Finding and Preventing attacks on TLS

http://smacktls.com
http://weakdh.org
http://mitls.org

Karthikeyan Bhargavan
+ many, many others.
(INRIA, LORIA, Microsoft Research, IMDEA, Univ of Pennsylvania, Univ of Michigan, JHU)
INRIA Prosecco

Our goal is to verify implementations of mainstream cryptographic protocols

- Computational model of cryptography
- Semi-automated verification tools
- Account for messy details of protocol in practice

This talk: new proofs and attacks on TLS

- **miTLS**: formal security theorems [Crypto’14]
- **Skip, Freak**: state machine attacks [Oakland’15]
- **Logjam**: imperfect forward secrecy (submitted)
- How to reduce the gap between formal theorems and concrete attacks?
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
**Goal: a secure channel**

- **connect (server, port);**
- **send (d1);**
- **send (d2);**
- **send (d3);**
- **...**

- **accept (port);**
- **d1' = recv ();**
- **d2' = recv ();**
- **d3' = recv ();**
- **...**

**Security Goal:** A network attacker cannot
- Impersonate the client or the server or inject data (authenticity)
- Distinguish the data stream from random bytes (confidentiality)

Secure channels for the Web

Security Goal: A network attacker cannot
- Impersonate the server or inject server data (authenticity)
- Distinguish user data from random bytes (confidentiality)

More formally: SACCE [Krawczyk et al. ’13] + sLHAE
TLS protocol overview

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

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

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

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

Hello
- Client
- Server
- cr
- sr
- TLS 1.2 (mandatory cipher suite)
- Client and server exchange fresh nonces

KEM
- cert_S
- rsa-enc(pms,pk_S)
- 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` to Server
- Server sends `sr` to Client
- Client and Server exchange fresh nonces
- **TLS 1.2** (Google’s cipher suite)

**KEM**
- Server sends `cert_s`
- Server signs `((G, g^y), sk_s)` using `rsa-sign`
- Server sends `g^x`
- **Server** picks group/curve
- **Server** signs group, key share
- **Server** calculates `pms = g^{xy}`
- **Server** sends `ms`, keys (k) derived from `pms, cr, sr`

**Finished**
- Client sends `ae(0||tag_c,k)` to Server
- Server sends `ae(0||tag_s,k)` to Client
- **Server** derives `tag_c, tag_s` from `ms + SHA-256 hash of handshake log`

**AppData**
- Client encrypts `ae(i||d_i,k)`
- Server decrypts `ae(i||d_i,k)`
- **Authentic**ated encryption
Cryptographic weaknesses

Many obsolete crypto constructions

• RSA encryption with PKCS#1 v1.5 padding (Bleichenbacher)
• MAC-then-Pad-then-Encrypt with AES-CBC (Padding oracle)
• Compress-then-MAC-then-Pad-then-Encrypt (CRIME)
• Chained IVs in TLS 1.0 AES-CBC (BEAST)
• RC4 key biases

Countermeasures

• Disable these features: SSL3, compression, RC4
• Implement ad-hoc mitigations very very carefully:
  • empty fragment to initialize IV for TLS 1.0 AES-CBC
  • constant time mitigation for Bleichenbacher attacks
  • constant-time plaintext length-hiding HMAC to prevent Lucky 13
Other implementation challenges

Memory safety
Buffer overruns leak secrets

Missing checks
Forgetting to verify signature/MAC/certificate bypasses crypto guarantees

Certificate validation
ASN.1 parsing, wildcard certificates

State machine bugs
Most TLS implementations don’t conform to spec
Unexpected transitions break protocol (badly)
Implementing TLS correctly

Use formal methods!

• Use a type-safe programming language
  • F#, OCaml, Java, C#,…
  • (No buffer overruns, no Heartbleed)
• Verify the logical correctness of your code
  • Use a software verifier: F7/F*, Why3, Boogie, Frama-C,…
• Link software invariants to cryptographic guarantees
  • Use a crypto verifier: EasyCrypt, CryptoVerif, ProVerif
  • Hire a cryptographer!
miTLS: a verified implementation

miTLS
A verified reference TLS implementation

miTLS
miTLS is a verified reference implementation of the TLS protocol. Our code fully supports its wire formats, ciphersuites, sessions and connections, re-handshakes and resumptions, alerts and errors, and data fragmentation, as prescribed in the RFCs; it interoperates with mainstream web browsers and servers. At the same time, our code is carefully structured to enable its modular, automated verification, from its main API down to computational assumptions on its cryptographic algorithms.

News
3 October 2014
miTLS 0.8.1 released. See the download page.

20 August 2014
miTLS 0.7.0 released. See the download page.

• How does this verification link to crypto assumptions and the secure channel goal?
State of the art

[Jager et al. '11] Security for TLS-DHE + authenticated encryption in the standard model
Monolithic proof (ACCE model), does not cover TLS-RSA

[Krawczyk, Paterson, Wee '13] Security for TLS-DHE + TLS-RSA + authenticated encryption
KEM abstraction (SACCE model), single ciphersuite, does not cover resumption, renegotiation

[Bhargavan et al. '14] Comprehensive modular treatment of a TLS handshake implementation
Multi-ciphersuite, multi-handshake security
Cryptographic core of TLS

\[ \ell_c \leftarrow \$; \]

\[ \text{ClientHello}[\ell_c] \rightarrow \ell_s \leftarrow \$; \ell := \ell_c \parallel \ell_s; \]

\[ \ell := \ell_c \parallel \ell_s; \]
\[ pk := pk(\text{cert}_S) \]
\[ c, ms \leftarrow \text{Enc}(pk, \ell) \]
\[ k := \text{KDF}(ms, \ell) \]

\[ \text{ServerHello}[\ell_S] \]

\[ \leftarrow \text{ServerCertificate}[\text{cert}_S] \rightarrow \text{ServerHelloDone} \]

\[ \leftarrow \text{ClientKeyExchange}[c] \rightarrow ms \leftarrow \text{Dec}(sk, \ell, c) \]

\[ \log_C := \{\text{all prior messages}\} \]
\[ \text{tag}_C := \text{MAC}(ms, \text{“C”}, \log_C) \]

\[ \text{ClientFinished}[\text{tag}_C] \rightarrow \log_C := \{\text{all prior messages}\} \]
\[ \text{tag}_C \leftarrow \text{MAC}(ms, \text{“C”}, \log_C) \]
\[ k := \text{KDF}(ms, \ell) \]

\[ \log_S := \{\text{all prior messages}\} \]
\[ \text{tag}_S := \text{MAC}(ms, \text{“S”}, \log_S) \]

\[ \leftarrow \text{ServerFinished}[\text{tag}_S] \leftarrow \text{ServerFinished}[\text{tag}_S] \rightarrow \text{tag}_S := \text{MAC}(ms, \text{“S”}, \log_S) \]
Standard socket API with embedded security specification

- Abstract types for confidentiality (a la information flow)
- Refinements for authenticity (a la contracts/pre-/post-conditions)

```plaintext
type Connection // for each local instance of the protocol
type (;c:Connection) AppData

// creating new client and server instances
val connect: TcpStream -> Params -> Connection
val accept: TcpStream -> Params -> Connection

// reading data
type (;c:Connection) IOResult_i =
| Read of c':Connection * data:(;c) AppData
| CertQuery of c':Connection
| Complete of c':Connection { Agreement(c') }
| Close of TcpStream
| Warning of c':Connection * a:AlertDescription
| Fatal of a:AlertDescription
val read : c:Connection -> (;c) IOResult_i

// writing data
type (;c:Connection,data:(;c) AppData) IOResult_o =
| WriteComplete of c':Connection
| WritePartial of c':Connection * rest:(;c') AppData
| MustRead of c':Connection
val write: c:Connection -> data:(;c) AppData -> (;c,data) IOResult_o

// triggering new handshakes, and closing connections
val rehandshake: c:Connection -> Connection Result
val request: c:Connection -> Connection Result
val shutdown: c:Connection -> TcpStream Result
```
Main crypto result: concrete TLS & ideal TLS are computationally indistinguishable.

We prove that ideal miTLS meets its secure channel specification using standard program verification (typing).
Security theorem

Proof automation
7,000 lines of F#
checked against
3,000 lines of F7
type annotations
+
3,000 lines of EasyCrypt
for the core key exchange

Ongoing work
ECDHE, GCM, Certificates, Side-channels
Mission accomplished?
Composing Key Exchanges

1. \text{ClientHello}(v, [kx_1, kx_2, \ldots])
2. \text{ServerHello}(v, kx = \text{RSA})
3. \text{ServerCertificate}(certs)
4. \text{ServerKeyExchange}(\text{sign}((G, g^v), sK_s))
5. \text{ServerHelloDone}
6. \text{ClientKeyExchange}(\text{rs}a\text{enc}(\text{p}ms, pK_s))
7. \text{ClientCCS}
8. \text{ClientFinished}(\text{mac}(log, pms))
9. \text{ServerCCS}
10. \text{ServerFinished}(\text{mac}(log', pms))
11. ApplicationData*

\text{ClientHello}(v, [kx_1, kx_2, \ldots])

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

\text{ServerCertificate}(certs)

\text{ServerKeyExchange}(\text{sign}((G, g^v), sK_s))

\text{ServerHelloDone}

\text{ClientKeyExchange}(g^x)

\text{ClientCCS}

\text{ClientFinished}(\text{mac}(log, g^{xy}))

\text{ServerCCS}

\text{ServerFinished}(\text{mac}(log', g^{xy}))

ApplicationData*

\text{ClientHello}(v, [kx_1, kx_2, \ldots])

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

\text{ServerCertificate}(certs)

\text{ServerKeyExchange}(\text{sign}((G, g^v), sK_s))

\text{ServerHelloDone}

\text{ClientKeyExchange}(\ldots)

\text{ClientCCS}

\text{ClientFinished}(\text{mac}(log, \ldots))

\text{ServerCCS}

\text{ServerFinished}(\text{mac}(log', \ldots))

ApplicationData*
TLS State Machine

RSA + DHE + ECDHE
+ Session Resumption
+ Client Authentication

• Covers most features used on the Web
• Composition implemented and proved for miTLS [IEEE S&P’13, CRYPTO’14]
• Only works for reference code written for verification, in F# (dialect of OCaml)

Can this proof technique be applied to OpenSSL?
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

State machine for common Web configurations
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 Key Exchanges

ClientHello($v, [kx_1, kx_2, \ldots]$)  \rightarrow  RSA

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

\rightarrow  \text{ServerCertificate($certs$)}

\rightarrow  \text{ServerKeyExchange(sign($G, g^y, sk_S$))}

\rightarrow  \text{ServerHelloDone}

\rightarrow  \text{ClientKeyExchange(rsaenc($pms, pk_S$))}

\rightarrow  \text{ClientCCS}

\rightarrow  \text{ClientFinished(mac($log, pms$))}

\rightarrow  \text{ServerCCS}

\rightarrow  \text{ServerFinished(mac($log, pms$))}

\rightarrow  \text{ApplicationData*}

\rightarrow  \text{ClientHello($v, [kx_1, kx_2, \ldots]$)}  \rightarrow  \text{(EC)DHE}

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

\rightarrow  \text{ServerCertificate($certs$)}

\rightarrow  \text{ServerKeyExchange(\ldots)}

\rightarrow  \text{ServerHelloDone}

\rightarrow  \text{ClientKeyExchange(\ldots)}

\rightarrow  \text{ClientCCS}

\rightarrow  \text{ClientFinished(mac($log, g^y$))}

\rightarrow  \text{ServerCCS}

\rightarrow  \text{ServerFinished(mac($log, g^y$))}

\rightarrow  \text{ApplicationData*}

\rightarrow  \text{ClientHello($v, [kx_1, kx_2, \ldots]$)}

\rightarrow  \text{ServerHello($v, kx$)}

\rightarrow  \text{ServerCertificate($certs$)}

\rightarrow  \text{ServerKeyExchange(\ldots)}

\rightarrow  \text{ServerHelloDone}

\rightarrow  \text{ClientKeyExchange(\ldots)}

\rightarrow  \text{ClientCCS}

\rightarrow  \text{ClientFinished(mac($log, \ldots$))}

\rightarrow  \text{ServerCCS}

\rightarrow  \text{ServerFinished(mac($log, \ldots$))}

\rightarrow  \text{ApplicationData*}
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**
FREAK: Downgrade to RSA_EXPORT

A man-in-the-middle attacker can:
- impersonate servers that support RSA_EXPORT,
- at buggy clients that allow ServerKeyExchange in RSA
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
Export-Grade DHE in TLS

Yet another export-grade cipher in TLS

- Diffie-Hellman groups limited to 512 bits

- Protocol flaw: Messages look the same as regular DHE!
A man-in-the-middle attacker can:
- impersonate servers that support DHE_EXPORT,
- at **ALL** clients that accept 512-bit DH groups

512-bit discrete logs a bit harder than factoring RSA-512
First broken with CADO-NFS in 2014
Now: 2 weeks of precomputation on Grid5000 + 90 seconds for each key
Logjam: Exploit and Impact

Of course, many servers still offer DHE_EXPORT
- 8.4% of Alexa Top 1M websites in March 2015
- Vulnerable sites included fbi.gov, tcl.tk, …
- See demos at weakdh.org

New worry: 768-bit, 1024-bit discrete logs are feasible
- 768-bits would require months of precomputation
- 1024-bits would require supercomputers
- IPsec, SSH, TLS all use 768-bit and 1024-bit primes!

Security updates to major browsers and websites
- Disabling 512-bit, then 768-bit, then 1024 bit
- Move to 20148-bit freshly-generated safe primes
Long-term Solutions?
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!
A new protocol: TLS 1.3

Stronger key exchanges, fewer options
- ECDHE and DHE by default, no RSA key transport
- Strong DH groups (> 2047 bits) and EC curves (> 255 bits)
- Only AEAD ciphers (AES-GCM), no CBC, no RC4

Signatures, session keys bound to handshake params
- Session hash for key derivation (proposed by us)
- Server signature covers ciphersuite (preventing Logjam)

Faster: lower latency with 1 round-trip
- 0-round trip mode also available
- Security analysis ongoing
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
Questions?

mitls.org
smacktls.com
weakdh.org