Type-based Verification at Scale

miTLS: a verified reference implementation of TLS

Antoine Delignat-Lavaud
with
Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella Beguelin

https://www.miTLS.org
Transport Layer Security (1994—)

The most widely deployed cryptographic protocol? HTTPS, 802.1x, VPNs, files, mail, VoIP, ...

20 years of attacks, fixes, and extensions
1994  Netscape’s Secure Sockets Layer
1995  SSL3
1999  TLS1.0 (RFC2246, ≈SSL3)
2006  TLS1.1 (RFC4346)
2008  TLS1.2 (RFC5246)

Many implementations
SChannel, OpenSSL, NSS, GnuTLS, JSSE, PolarSSL
many patches every year; Snowden allegations

Many papers
Well-understood, detailed specs
many security theorems...
mostly for small simplified models of TLS
What can still possibly go wrong?
**Infrastructure**
certificate management (PKI)

**Protocol Logic**
e.g. ambiguous messages
- cause clients and server to negotiate weak sessions

**TLS DESIGN**

**Cryptography**
e.g. not enough randomness
- write applet to realize adaptive attack (BEAST)

**Implementation Bugs**
many critical errors

**Weak Algorithms**
MD5, PKCS1, RC4, ...

**Application**
HTTPS clients & servers
ASN.1

Binary encoding standard

Ancient (1984)

<Tag, Length, Value>

Distinguished rules (DER): unique serialization

Infrastructure

Certificates are hard to check
NSS Signature Forgery (August 2014)

PKCS#1 Padding

Sign: \( S = (\text{padding}||\text{oid}||h)^d \mod N \)

Verify: \( S^e \mod N \) (e.g. \( e=3 \))

Infrastructure

Certificates are hard to check
NSS Signature Forgery (August 2014)

Infrastructure
Certificates are hard to check

PKCS#1 Padding + hash algorithm OID
Injection of junk bytes
Ignored by ASN.1 parser

CA Certificate

Signed hash

Bleichenbacher attack on low public exponents (e=3)
Cubic root of padding + Fermat theorem for hash
gotofail bug
iOS, Feb’14

then GnuTLS, Mar’14
then Heartbleed, OpenSSL, April’14

Implementation Bugs
many critical errors

The duplicate goto always branches to the end of the function with err = 0.
The key is not bound to the server signing-key certificate.
SMACK: State Machine AttaCKs

Implementation Bugs
What gets really implemented?
Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS

Karthisayan Bhargavan*, Antoine Delignat-Lavaud*, Cédric Fournet†, Alfredo Pironti* and Pierre-Yves Strub‡
*INRIA Paris-Rocquencourt †Microsoft Research ‡IMDEA Software Institute

Abstract—TLS was designed as a transparent channel abstraction to allow developers with no cryptographic expertise to protect their application against attackers that may control some clients, some servers, and may have the capability to tamper with network connections. However, the security guarantees of TLS fall short of those of a secure channel, leading to a variety of attacks.

We show how some widespread false beliefs about these guarantees can be exploited to attack popular applications and defeat several standard authentication methods that rely too naively on TLS. We present new client impersonation attacks against TLS renegotiations, wireless networks, challenge-response protocols, and channel-bound cookies. Our attacks exploit combinations of RSA and Diffie-Hellman key exchange, session resumption, and renegotiation to bypass many recent countermeasures. We also demonstrate new ways to exploit known weaknesses of HTTP over TLS. We investigate the root causes for these attacks and propose new countermeasures. At the protocol level, we design and implement two new TLS extensions that strengthen the authentication guarantees of the handshake. At the application level, we develop an exemplary HTTPS client library that implements several mitigations, on top of a previously verified TLS implementation, and verify that their composition provides strong, simple application security.

sessions, validating certificates, etc. Meanwhile, TLS applications continue to rely on URLs, passwords, and cookies; they mix secure and insecure transports; and they often ignore lower-level signals such as handshake completion, session resumption, and truncated connections.

Many persistent problems can be blamed on a mismatch between the authentication guarantees expected by those actually provided by TLS. To illustrate our point, we list below a few myths about those guarantees, which we debunk in this paper. Once a connection is established:

1) the principal at the other end cannot change;
2) the master secret is shared only between the two peers, so it can be used to derive fresh application-layer keys;
3) the \texttt{tls-unique} channel binding [6] uniquely identifies the connection;
4) the connection authenticates the whole data stream, so it is safe to start processing application data as it arrives.

The first is widely believed to be ensured by the TLS renegotiation extension [49]. The second and third are used for man-in-the-middle protections in tunneled protocols like PAP and some authentication modes in SASL and GSS-API. The fourth
Many web services rely on session tokens to authenticate their users.

The `secure` cookie attribute tells the client browser that the cookie is HTTPS-only.

Many browsers silently process truncated HTTP (e.g. images).

After truncation, any fake HTTP query leaks the authentication token.

**Application**

HTTPS clients & servers

---

**HTTP/1.1 302 Redirect**

Location: https://x.com/P

Set-Cookie: SID=\[SessionToken\]; secure

Content-Length: 0

Protected by TLS

---

<table>
<thead>
<tr>
<th>Browser vulnerable to truncations?</th>
<th>Header</th>
<th>Body (Length)</th>
<th>Body (Chunked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android 4.2.2</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Chrome 27</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Chrome 28</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Firefox 24</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Safari Mobile 7.0.2</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Opera Mini 7.5</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Opera Classic 12.1</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Internet Explorer 10</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
SPDY Connection Pooling Attack

Application
HTTPS clients & servers
SPDY Connection Pooling Attack

The site's security certificate is not trusted!

You attempted to reach localhost, but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which Chrome cannot rely on for identity information, or an attacker may be trying to intercept your communications.

You should not proceed, especially if you have never seen this warning before for this site.

Proceed anyway  Back to safety

Help me understand

Application
HTTPS clients & servers
SPDY Connection Pooling Attack

This should be impossible because of certificate pinning in Chrome
Protocol Logic
Bad compositions of protocol features
# Triple Handshake Attack

A server-in-the-middle, using 3 related handshakes

<table>
<thead>
<tr>
<th>Client</th>
<th>TLS library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium</td>
<td>NSS</td>
</tr>
<tr>
<td>Opera 15+</td>
<td></td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>SChannel</td>
</tr>
<tr>
<td>Safari &amp; Apple mail</td>
<td>Secure Transport</td>
</tr>
<tr>
<td>Apple Mail</td>
<td>Secure Transport</td>
</tr>
<tr>
<td>CURL</td>
<td>OpenSSL</td>
</tr>
<tr>
<td>CURL</td>
<td>GnuTLS</td>
</tr>
<tr>
<td>Wget</td>
<td>OpenSSL</td>
</tr>
<tr>
<td>NodeJS HTTPS</td>
<td>OpenSSL</td>
</tr>
<tr>
<td>PHP SSL Transport</td>
<td>OpenSSL</td>
</tr>
<tr>
<td>Apache HttpClient</td>
<td>JSSE 1.7</td>
</tr>
<tr>
<td>SVN / Neon</td>
<td>OpenSSL</td>
</tr>
</tbody>
</table>

## Protocol Logic

Bad compositions of protocol features
Infrastructure
certificate management (PKI)

Protocol Logic
- e.g. ambiguous messages
- cause clients and server to negotiate weak sessions

TLS DESIGN

Cryptography
- e.g. not enough randomness
- write applet to realize adaptive attack (BEAST)

Implementation Bugs
- many critical errors

Weak Algorithms
- MD5, PKCS1, RC4, ...

Application
- HTTPS clients & servers
To get application security, we must capture **all** these aspects within the same model

- We build a verified reference implementation
- We use automated proof tools to scale up
A cryptographically verified reference implementation of TLS
We develop and verify a **reference implementation** for SSL 3.0—TLS 1.2

1. **Standard compliance**: we closely follow the RFCs
   - concrete message formats
   - support for multiple ciphersuites, sessions and connections, re-handshakes and resumptions, alerts, message fragmentation,...
   - interop with other implementations such as web browsers and servers

2. **Verified security**: we structure our code to enable its modular verification, from its main API down to concrete assumptions on its base cryptography (e.g. RSA)
   - probabilistic computational security theorems for a 7000-line functionality (automation required)

3. **Experimental platform**: for testing corner cases, trying out attacks, studying application-level protocols, analysing new extensions and patches, ...
Ciphersuites & Crypto Agility

TLS negotiates its use of cryptography

Not all algorithms are equal!
Cautionary tale: ECDHE considered safest, open to attack for 2 years due to bug in elliptic curve fast multiplication

Clients and servers should get security for the ciphersuite they prefer, not the weakest they support
Circular dependency: TLS relies on the ciphersuites being negotiated

We verify TLS generically, for multiple ciphersuites & algorithms
This requires new cryptographic models
Verification Method: Type-Based Cryptography

Cryptographic algorithms
- types express cryptographic assumptions

Cryptographic constructions
- types express security guarantees

Security protocols
- types express attacker models

Applications & Adversaries

- symmetric encryption (AES-CBC)
- symmetric encryption (RC4)
- message authentication (SHA1)
- cipher (INT-CMA)
- IND-CPA
- encrypt then-MAC
- fragment-MAC-encode-then-encrypt
- authenticated encryption
- Secure RPC
- TLS 1.2
- some application code
- some attack