Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice

http://weakdh.org
http://mitls.org

Karthikeyan Bhargavan
+ many, many others.
(CNRS, INRIA, Microsoft Research, IMDEA, Univ of Pennsylvania, Univ of Michigan, Johns Hopkins)
If deployed correctly, \( k \) enjoys many properties:

- authenticity, confidentiality, forward secrecy
- + resistance to UKS, KCI? future secrecy?
Who chooses the group \((p,g)\)?
- client? server? standard writers?

What other protocols are running?
- do they use the same long-term keys \((sk_A, sk_B)\)?

Can the DH key shares be reused?
- do we need to validate public values \((g^x, g^y)\)?

How does the application use \(k\)?
- does \(k\) need to be unique for each session?
DH key exchange is well-understood, but real-world protocols based on DH often broken
• buggy ADH implementations (SKIP)
• weak DH groups (Logjam)
• unexpected security requirements (3Shake)

Understanding protocol details can make DLP-based attacks more practical

Case study: Transport Layer Security (TLS)
• Only modp groups, not elliptic curves
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**
  - Client: Protocol negotiation
    - Agree on version
    - Agree on ciphersuite
    - Determines all crypto algos
  - Server: Authenticated Key Exchange
    - Verify server/client identity
    - Generate master secret
    - Derive connection keys

- **Finished**
  - Server: Key, transcript confirmation
    - Completes authentication
    - Matches transcripts
    - Authenticated encryption
  - Client: Application data streams
    - Full duplex channel
    - Authenticated encryption

- **AppData**
(EC)DHE Handshake in TLS

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Client Actions</th>
<th>Server Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hello</strong></td>
<td>- Send <code>cr</code>&lt;br&gt;- Send <code>sr</code></td>
<td>- Send TLS 1.2 (Google’s cipher suite)&lt;br&gt;- Client and server exchange fresh nonces</td>
</tr>
<tr>
<td><strong>KEM</strong></td>
<td>- Get <code>cert_s</code>&lt;br&gt;- Get <code>rsa-sign((G, g^y), sk_s)</code>&lt;br&gt;- Get <code>g^x</code></td>
<td>- Server picks group/curve signs group, key share&lt;br&gt;- <code>pms = g^{xy}</code>&lt;br&gt;- <code>ms, keys (k)</code> derived from <code>pms, cr, sr</code></td>
</tr>
<tr>
<td><strong>Finished</strong></td>
<td>- Get `ae(0</td>
<td></td>
</tr>
<tr>
<td><strong>AppData</strong></td>
<td>- Get `ae(i</td>
<td></td>
</tr>
</tbody>
</table>
Who chooses the group \((p, g)\) ?
- server sends: \(\text{sign}(sk_s, cr | sr | p | g | g^y)\)

What other protocols are running?
- RSA key transport using same \((sk_A, sk_B)\)?

Can the DH key shares be reused?
- yes, and public values are not usually validated

How does the application use \(k\) ?
- fast session resumption, unique channel ids, …
TLS State Machine

RSA + DHE + ECDHE
+ Session Resumption
+ Client Authentication

• Covers most features used on the Web
• Already quite a complex combination of protocols!

• Composition proved secure for reference implementation
  [S&P’13, Crypto’14]
  [see http://mitls.org]
Full SSL/TLS State Machine?

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

All implemented in OpenSSL
Implementation Bugs

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

• Required messages can be skipped
• Unexpected messages may 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: \(rsa\text{-}sign(sk_S, cr | sr | p | g | g^y)\)
3. SKIP ServerCCS
   bypass encryption
4. Send ServerFinished
   using uninitialized MAC key
   bypass handshake integrity
5. Send ApplicationData
   unencrypted as S.com
A network attacker can impersonate *any* server (Paypal, Amazon, Google) to *any* Java TLS client (built with JSSE) until Jan 2015 (CVE-2014-6593).

Similar bugs also found in: OpenSSL, wolfSSL, mono TLS, GNU classpath.

Reality check: our efforts in securing ADH can be made irrelevant by ridiculous implementation bugs.
Choosing Good DH Groups
TLS-DHE in practice

Internet-wide scan of HTTPS servers (Zmap)
- 14.3M hosts, 24% support DHE
- 70,000 distinct groups \((p,g)\)

Composite-order groups with short exponents
- 4,800 groups where \((p-1)/2\) was not prime
- Applied ECM to opportunistically factor \((p-1)/2\)
- Got prime factors for 750 groups (40K connections)
- Some servers used short exponents: 128/160 bits
- Used Pohlig-Hellman to compute: full secret exponent for 159 servers (partial exponent for 460 servers)
Internet-wide scan of HTTPS servers (using Zmap)

- 14.3M hosts, 24% support DHE
- 70,000 distinct groups \((p,g)\)

Small-sized safe primes

- 84% (2.9M) servers use 1024-bit primes
- 2.6% (90K) servers use 768-bit primes
- 0.0008% (2.6K) servers use 512-bit primes
- But 512-bit DLP is solved since 2014, so can we break these connections?
Who uses 512-bit DHE?
Export-grade DHE in TLS

TLS 1.0 supported weakened ciphers to comply with export regulations in 1990s

- DHE_EXPORT: groups limited to 512 bits

DHE_EXPORT deprecated in 2000

- 8.4% (489K) HTTPS servers still support it
- … but only when client asks for it
- Web browsers never negotiate DHE_EXPORT, we should be safe, yes?

DHE_EXPORT handshake looks just like DHE

- Server uses same long-term signing key for both
- Difference is prime-size, which clients don’t check
- Opens the way to a downgrade attack!
Logjam: Downgrade to DHE_EXPORT

A man-in-the-middle attacker can:
- impersonate ANY server that supports DHE_EXPORT,
- at ANY client that accept 512-bit DHE groups

\[
(m_s, k_1, k_2) = \text{kdf}(g^{ab}, cr | sr) \]

512-bit discrete log needs to be computed in real-time!
512-bit Discrete Logs with CADO-NFS

<table>
<thead>
<tr>
<th></th>
<th>Sieving</th>
<th>Linear Algebra</th>
<th>Descent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
<td>log $B$</td>
<td>core-years</td>
</tr>
<tr>
<td>RSA-512</td>
<td>14</td>
<td>29</td>
<td>0.5</td>
</tr>
<tr>
<td>DH-512</td>
<td>15</td>
<td>27</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Times for cluster computation:

- polysel: 2000-3000 cores
- sieving: 288 cores
- linalg: 36 cores
- descent: DH-512
  - 3 hours
  - 15 hours
  - 120 hours
  - 70 seconds
**Logjam: Exploiting pre-computation**

Most DHE_EXPORT servers use the same groups
- 92% of these use one of two 512-bit primes

<table>
<thead>
<tr>
<th>Source</th>
<th>Popularity</th>
<th>Prime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>82%</td>
<td>9fbd8b8a004544f0045f1737d0ba2e0b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>274c0f1a9f588218fb435316a16e3741</td>
</tr>
<tr>
<td></td>
<td></td>
<td>71fd19d8d8f37c39bf863fd60e3e3006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80a3030c6e4c3757d08f70e6aa871033</td>
</tr>
<tr>
<td>mod_ssl</td>
<td>10%</td>
<td>d4bcd52406f69b35994b88de5db89682</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c8157f62d8f33633ee5772f11f05ab22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d6b5145b9f241e5acc31ff090a4bc711</td>
</tr>
<tr>
<td>(other)</td>
<td>8%</td>
<td>(463 distinct primes)</td>
</tr>
</tbody>
</table>

- We performed pre-computation for these primes
- About 1 week each one 2000-3000 cores
- Per-connection descent computation: 30-150 seconds
Logjam: Exploiting False Start

Some web browsers start sending data too early
- **Reason**: optimize TLS performance for PFS ciphersuites
- But now no need to wait 150 seconds for DLP
- We can capture this early application data and compute DLP at leisure to read password/cookies
For DHE_EXPORT connections

- Connections between Chrome/Firefox/IE and 8.4% of websites can be broken offline (no forward secrecy)

For regular DHE, we need to break bigger groups

- For academics, probably needs algorithmic improvements
- For governments, 768 bits is definitely reachable.

<table>
<thead>
<tr>
<th></th>
<th>Sieving</th>
<th>Linear Algebra</th>
<th>Descent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I</td>
<td>log B</td>
<td>rows</td>
</tr>
<tr>
<td>RSA-512</td>
<td>14</td>
<td>29</td>
<td>4.3M</td>
</tr>
<tr>
<td>DH-512</td>
<td>15</td>
<td>27</td>
<td>2.1M</td>
</tr>
<tr>
<td>RSA-768</td>
<td>16</td>
<td>37</td>
<td>250M</td>
</tr>
<tr>
<td>DH-768</td>
<td>17</td>
<td>35</td>
<td>150M</td>
</tr>
<tr>
<td>RSA-1024</td>
<td>18</td>
<td>42</td>
<td>8.7B</td>
</tr>
<tr>
<td>DH-1024</td>
<td>19</td>
<td>40</td>
<td>5.2B</td>
</tr>
</tbody>
</table>
Impact of breaking bigger groups

IKEv1, IKEv2, SSH all use 768-bit/1024-bit groups
- 6% of IKEv2 servers use Oakley 1 (768-bits)
- 64% of IKEv2 servers use Oakley 2 (1024-bits)
- 26% of SSH servers use Oakley 2 (1024-bits)
- 13% of HTTPS servers use 1024-bit Apache group

<table>
<thead>
<tr>
<th></th>
<th>all 512-bit groups</th>
<th>all 768-bit groups</th>
<th>one 1024-bit group</th>
<th>ten 1024-bit groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPS Top 1M w/ active downgrade</td>
<td>45,100 (8.4%)</td>
<td>45,100 (8.4%)</td>
<td>205,000 (37.1%)</td>
<td>309,000 (56.1%)</td>
</tr>
<tr>
<td>HTTPS Top 1M</td>
<td>118 (0.0%)</td>
<td>407 (0.1%)</td>
<td>98,500 (17.9%)</td>
<td>132,000 (24.0%)</td>
</tr>
<tr>
<td>HTTPS Trusted w/ active downgrade</td>
<td>489,000 (3.4%)</td>
<td>556,000 (3.9%)</td>
<td>1,840,000 (12.8%)</td>
<td>3,410,000 (23.8%)</td>
</tr>
<tr>
<td>HTTPS Trusted</td>
<td>1,000 (0.0%)</td>
<td>46,700 (0.3%)</td>
<td>939,000 (6.56%)</td>
<td>1,430,000 (10.0%)</td>
</tr>
<tr>
<td>IKEv1 IPv4</td>
<td>–</td>
<td>64,700 (2.6%)</td>
<td>1,690,000 (66.1%)</td>
<td>1,690,000 (66.1%)</td>
</tr>
<tr>
<td>IKEv2 IPv4</td>
<td>–</td>
<td>66,000 (5.8%)</td>
<td>726,000 (63.9%)</td>
<td>726,000 (63.9%)</td>
</tr>
<tr>
<td>SSH IPv4</td>
<td>–</td>
<td>–</td>
<td>3,600,000 (25.7%)</td>
<td>3,600,000 (25.7%)</td>
</tr>
</tbody>
</table>
Solutions?
Short-term fixes

Security updates to major TLS libraries, web browsers, websites, mail servers, …

• Disabling 512-bit, then 768-bit, then 1024 bit
• We recommend 2048-bit safe primes

Fixes are surprisingly hard to deploy

• Many libraries hard-code DH parameters
• Hardware devices difficult to update
• May be easier to move to ECDHE
A new protocol: TLS 1.3

Stronger key exchanges, fewer options

- ECDHE and DHE by default, no RSA key transport
- Fixed 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

- Server signature covers ciphersuite (preventing Logjam)

Faster: lower latency with 1 round-trip

- 0-round trip mode also available
- Many security analyses ongoing (!)
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.

• A strong security theorem links software invariants to standard cryptographic assumptions
Conclusions

Protocols use and compose Diffie-Hellman key exchanges in various (weird) ways
• Complex compositions lead to implementation bugs, downgrade attacks, …

Don’t assume that servers know how to generate good DH groups or keys
• Most don’t validate groups or keys
• Off-curve and small-subgroup attacks are feasible

Beware of cryptographic front-doors (EXPORT)
• Obsolete crypto can bite you decades later
Questions?

weakdh.org
mitls.org
smacktls.com

Papers:

• A Messy State of the Union: Taming the Composite State Machines of TLS. IEEE S&P, 2015