Analyzing Real-World Crypto Protocols: from attacks to proofs

Karthik Bhargavan

+ many, many others.

(INRIA, Microsoft Research, LORIA, IMDEA, Univ of Pennsylvania, Univ of Michigan, JHU)
Analyzing Real-World Protocols

Internet protocols (TLS, SSH, IPsec) seemingly implement textbook cryptographic protocols... yet, not exactly the same protocols

- Modeling gaps between paper proofs and real protocol
- Implementation gaps between protocol and deployment

These gaps lead to many attacks, new questions

- Can we prove the deployed protocol correct?
- Can we show that a theoretical attack can be exploited?
- Important to understand where these gaps come from, so we can close them in new protocol designs
Example: HTTPS for Web Security

Secure connection to bank’s website
Nobody other than the bank can read what I type (confidentiality)

My secret login Information
Nobody other than me can access my account page (authentication)

Goal: Prevent unauthorized access to data even if an unknown attacker controls the network and some other bank clients.
Many recent attacks on HTTPS

- **BEAST** CBC predictable IVs [Sep’11]
- **CRIME** Compression before Encryption [Sep’12]
- **RC4** Keystream biases [Mar’13]
- **Lucky 13** MAC-Encode-Encrypt CBC [May’13]
- **3Shake** Insecure resumption [Apr’14]
- **POODLE** SSLv3 MAC-Encode-Encrypt [Dec’14]
- **SMACK** State machine attacks [Jan’15]
- **FREAK** Export-grade 512-bit RSA [Mar’15]
- **LOGJAM** Export-grade 512-bit DH [May’15]
- **SLOTH** RSA-MD5 signatures [Jan’16]
- **DROWN** SSLv2 RSA-PKCS#1v1.5 [Mar’16]
Many recent attacks on HTTPS

High-profile attacks, with Logos!
What’s going on?
How do we prevent this in the future?
Lecture Plan

Part 1-2: Attacks on Transport Layer Security
Part 3-4: Towards High-Assurance Crypto Software
Part I:

Attacks on Transport Layer Security
Reading Materials

• **TLS 1.2.** IETF RFC 5246.


• *Messy State of the Union: Taming the Composite State Machines of TLS.* IEEE Security and Privacy 2015.


• *Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH.* ISOC NDSS 2016.
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 SSLv3
1999 TLS1.0 (RFC2246)
2006 TLS1.1 (RFC4346)
2008 TLS1.2 (RFC5246)
2018 TLS1.3 (RFC8446)

Many implementations
OpenSSL, SecureTransport, NSS, SChannel, GnuTLS, JSSE, PolarSSL, ...
many bugs, attacks, patches every year

Many security theorems
mostly for simplified models of TLS
Goal: a secure channel

Security Goal: As long as the adversary does not control the long-term credentials of the client and server, it cannot
- Inject forged data into the stream (authenticity)
- Distinguish the data stream from random bytes (confidentiality)


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

```
accept(port);
d1' = recv();
d2' = recv();
d3' = recv();
...
```
Secure channels for the Web

**Security Goal**: As long as the client is honest and the adversary does not know the server’s private key, it cannot
- Inject forged data into the data stream (authenticity)
- Distinguish the data stream from random bytes (confidentiality)

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

**Hello**
- Client to Server: Protocol negotiation
  - Version, Ciphersuite
  - DH groups, Auth mode
- Server to Client: No interaction

**AKE**
- Client to Server: Authenticated Key Exchange
  - Verify peer identity
  - Generate session key
- Server to Client: No interaction

**Finished**
- Client to Server: Transcript & key confirmation
  - Completes authentication
  - Matches transcripts
- Server to Client: No interaction

**AppData**
- Application data streams
  - Full duplex channel
  - Authenticated encryption
- No interaction

---

Client

Server
TLS negotiation

**ClientHello (m₃)**
nonce₃, TLS 1.2,
[RSA, DHE, ECDHE],
[AES_GCM, AES_CBC, RC4],

**ServerHello (m₂)**
nonceₛ, TLS 1.0,
ECDHE, AES_GCM
The many, many modes of TLS

Protocol versions
- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges
- ECDHE, FFDHE, RSA, PSK, ...

Authentication modes
- ECDSA, RSA signatures, PSK, ...

Authenticated Encryption Schemes
- AES-GCM, CBC MAC-Encode-Encrypt, RC4, ...

100s of possible protocol combinations!
RSA Key Transport

Client

Server

ServerCertificate (m₃)

ClientKeyExchange (m₄)

ClientFinished (m₅)

ServerFinished (m₆)

Session Key

K = PRF(pms, nonceᵥ, nonceₛ)

cert(pkₛ)

rsa-encrypt(pms, pkₛ)

mac(m₁-m₄, K)

mac(m₁-m₅, K)

Session Key

K = PRF(pms, nonceᵥ, nonceₛ)
RSA Key Transport

- Client chooses secret \( pms \), encrypts it with server’s public key \( pk_s \)

\[
\text{rsa-pkcs1-encrypt}(pms, pk_s) = [\text{pad} \mid pv_{\text{max}} \mid pms]^e \mod pq
\]

- Server decrypts, checks pad and protocol version, computes session key from \( pms \)

Security: In theory, relies on hardness of factoring \( pq \)
RSA Key Transport: Attacks and Proofs

• [1994] Classic protocol, many proofs
• [1998] **Chosen Ciphertext attack** on PKCS#1
• [2002] Mitigations in TLS and other protocols
• [2013] Proof of TLS assuming mitigation
• [2016] **DROWN**: downgrade to SSLv2 + Bleichenbacher + software bugs

---

**DROWN: Breaking TLS using SSLv2**

Nimrod Aviram$^1$, Sebastian Schinzel$^2$, Juraj Somorovsky$^3$, Nadia Heninger$^4$, Maik Dankel$^2$, Jens Steube$^5$, Luke Valenta$^4$, David Adrian$^6$, J. Alex Halderman$^6$, Viktor Dukhovni$^7$, Emilia Käsper$^8$, Shaanan Cohney$^4$, Susanne Engels$^3$, Christof Paar$^3$ and Yuval Shavitt$^1$

$^1$Department of Electrical Engineering, Tel Aviv University
$^2$Max Planck University of Applied Sciences
$^3$University of Applied Sciences
(EC)DHE Key Exchange

Client

Server

ServerKeyExchange (m_3)
cert(pk_s), rsa-sign(G | g^y, sk_s)

ClientKeyExchange (m_4)
g^x

ClientFinished (m_5)
mac(m_1-m_4, K)

ServerFinished (m_6)
mac(m_1-m_5, K)

Session Key
K = PRF( g^{xy}, nonce_C, nonce_s)
(EC)DHE Key Exchange

- Server chooses group \((p,g)\) and a public value \(g^y\) and signs it with its certificate signing key \(sk_S\):
  \[
  \text{rsa-sign}([\text{nonce}_C \mid \text{nonce}_S \mid \ p \mid g \mid g^y], \ sk_S)
  \]

- Client sends \(g^x\) and both parties derive D-H key
  \[
  \text{pms} = g^{xy} \mod p
  \]

*Security*: In theory, relies on (some) D-H assumption
(EC)DHE Key Exchange Analysis

- [1994] Classic protocol, many proofs
- [2011] Proof of mutually-authenticated DHE
- [2013] Proof of server-authenticated RSA+DHE
- [2015] **Logjam:** Downgrade to DHE_EXPORT + discrete logarithm + configuration bugs

---

**Imperfect Forward Secrecy:**
How Diffie-Hellman Fails in Practice

David Adrian*, Karthikeyan Bhargavan*, Zakir Durumeric*, Pierrick Gaudry†, Matthew Green§
J. Alex Halderman*, Nadia Heninger‡, Drew Springall*, Emmanuel Thomé†, Luke Valenta‡
Benjamin VanderSloot*, Eric Wustrow*, Santiago Zanella-Béguelin‡, Paul Zimmermann†

* INRIA Paris-Rocquencourt † INRIA Nancy-Grand Est, CNRS, and Université de Lorraine
‡ Microsoft Research § Johns Hopkins University

For additional materials and contact information, visit WeakDH.org.
TLS Record Protocol

Hello
- Protocol negotiation
  - Version, Ciphersuite
  - DH groups, Auth mode

AKE
- Authenticated Key Exchange
  - Verify peer identity
  - Generate session key

Finished
- Transcript & key confirmation
  - Completes authentication
  - Matches transcripts

AppData
- Application data streams
  - Full duplex channel
  - Authenticated encryption
TLS Record Protocol

Application Data (aead)

- $\text{encrypt}(d,K)$
- $\text{authenticate}([d,seq_{n},PV],K)$
MAC-Encode-Encrypt

- AEAD via AES-CBC and HMAC:
  \( \text{aes-cbc}([\text{hmac}(d,K_m) \mid \text{pad}], K_e) \)

- Padding up to CBC block boundary

- Separate keys for encryption and MAC derived for both directions from session key (\text{pms})

- Recipient decrypts message, then verifies padding, then verifies MAC

\textbf{Security:} In theory, composes UF-CMA and IND-CPA
MAC-Encode-Encrypt Analysis

- [2001] Dangerous, but secure in SSL
- [2002] Padding oracle attacks on MEE-CBC
- [2010] Proof assuming side-channel mitigation
- [2011] Attack on short MACs, new proof
- [2016] First proof of MEE-CBC implementation

Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC

José Bacelar Almeida¹², Manuel Barbosa¹³, Gilles Barthe⁴, and François Dupressoir⁴

¹ HASLab – INESC TEC
² University of Minho
³ DCC-FC, University of Porto
Specification Gap → Crypto Attacks

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 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
Implementation Gap $\rightarrow$ Software Bugs

Memory safety
Buffer overruns leak secrets

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

Certificate validation
ASN.1 parsing, wildcard certificates
Optional Protocol Features

Client authentication with signatures or PSK
  • Authentication may be required for some resources

Renegotiation for key refresh and (re-)authentication
  • A new TLS handshake tunneled within old one

Session resumption for fast connections
  • A new connection using previous session key
Examples of Modeling Gaps

Crypto construction specified in an execution model, proved secure for some goals under a threat model

- Authenticated Key Exchange + Authenticated Encryption = Secure Channel Protocol

Protocols and their implementations need to adapt construction to real-world execution & threat model

- Infinite precision arithmetic → Bignum arithmetic
- Atomic algorithm steps → Side-channel attacks
- Protocol runs in isolation → Multiple protocols in parallel
- Honest participants send a stream of secrets → JavaScript attackers mix known plaintext with secrets
- Prove Authenticity + Confidentiality → Channel Binding?
Formalizing Secure Channels for TLS

[Paterson et al. '11] Stateful LHAE security for MEE-CBC

[Jager et al. '11] ACCE security for TLS-DHE with client auth

[Krawczyk et al. '13] SACCE security for TLS-DHE + TLS-RSA

[Giesen et al. '13] ACCE security for TLS-DHE + renegotiation

[Bhargavan'14 et al.] Verified security for a reference implementation of TLS 1.0-1.2 with TLS-RSA, TLS-DHE, client authentication, resumption, and renegotiation

[Almeida et al. '16] Verified security for MEE-CBC code
Cryptographic security goals

If a client completes with an honest server’s certificate and (all) strong algorithms, then

**Agreement**: there must be a server that agrees on all handshake variables (e.g. key, server cert)

**Authenticity**: each endpoint only accepts a prefix of the data sequence send by its peer

If connection is gracefully closed, then all sent data has been accepted

**Confidentiality**: the data sequences in both directions are indistinguishable from random

(vice versa for server if client is authenticated)
# Still, many, many attacks

<table>
<thead>
<tr>
<th>Attack</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEAST</td>
<td>CBC predictable IVs</td>
<td>Sep’11</td>
</tr>
<tr>
<td>CRIME</td>
<td>Compression before Encryption</td>
<td>Sep’12</td>
</tr>
<tr>
<td>RC4</td>
<td>Keystream biases</td>
<td>Mar’13</td>
</tr>
<tr>
<td>Lucky 13</td>
<td>MAC-Encode-Encrypt CBC</td>
<td>May’13</td>
</tr>
<tr>
<td>3Shake</td>
<td>Insecure resumption</td>
<td>Apr’14</td>
</tr>
<tr>
<td>POODLE</td>
<td>SSLv3 MAC-Encode-Encrypt</td>
<td>Dec’14</td>
</tr>
<tr>
<td>SMACK</td>
<td>State machine attacks</td>
<td>Jan’15</td>
</tr>
<tr>
<td>FREAK</td>
<td>Export-grade 512-bit RSA</td>
<td>Mar’15</td>
</tr>
<tr>
<td>LOGJAM</td>
<td>Export-grade 512-bit DH</td>
<td>May’15</td>
</tr>
<tr>
<td>SLOTH</td>
<td>RSA-MD5 signatures</td>
<td>Jan’16</td>
</tr>
<tr>
<td>DROWN</td>
<td>SSLv2 RSA-PKCS#1v1.5</td>
<td>Mar’16</td>
</tr>
</tbody>
</table>
What goes wrong in TLS?

Cryptographic Weaknesses in Legacy Constructions
- Weak hash functions, weak DH groups, short block ciphers, leaky PKCS#11v1.5 padding

Logical Flaws in Handshake Protocol
- Cross-Protocol Attacks, Downgrade Attacks, Transcript Synchronization/Collision Attacks

Implementation Bugs in TLS Libraries
- Bugs in crypto library, Buffer overflows in packet parsing, Composition bugs in state machines, Bad configurations

Sometimes, a mix of all of the above!
Exploiting Crypto Weaknesses: Weak DH Groups
Anonymous Diffie-Hellman (ADH)

\[ A \]

Knows \( G = (g, p) \)

\[ k = \text{kdf}(g^{xy} \mod p) \]

\[ B \]

Knows \( G = (g, p) \)

\[ g^x \mod p \]

\[ g^y \mod p \]

\[ k = \text{kdf}(g^{xy} \mod p) \]
Man-in-the-Middle attack on ADH

Active Network Attacker or Malicious Peer
Authenticated DH (SIGMA)

$A$ knows $sk_A, pk_B$

$G = (g, p)$

$m_1 = g^x \mod p$

$m_2 = g^y \mod p$

$k = kdf(g^{xy} \mod p)$

$B$ knows $sk_B, pk_A$

$G = (g, p)$

$k = kdf(g^{xy} \mod p)$

Sign$(sk_A, \text{hash}(m_1 | m_2)), \text{mac}(k, A)$

Sign$(sk_B, \text{hash}(m_1 | m_2)), \text{mac}(k, B)$

Sign-and-MAC the transcript: prevents most MitM attacks
Weak Diffie-Hellman Groups

Diffie-Hellman shared secret computation

\[ k = \text{kdf}(g^{xy} \mod p) \]

**Theoretical Security:**
- Relies on some DH assumption (CDH, Gap, PRF-ODF,...)
- Attacker cannot compute \( k \) without knowing \( x \) or \( y \)

**Attacks:**
- Best known attacks rely on discrete log:
  \[ y = \log(g^y \mod p) \]
Discrete Log Attack on SIGMA

A

Knows $sk_A, pk_B$

$G = (g, p)$

$m_1 = g^x \mod p$

$m_2 = g^y \mod p$

$k = \text{kdf}(g^{xy} \mod p)$

sign$(sk_A, \text{hash}(m_1 \mid m_2))$, mac$(k, A)$

sign$(sk_B, \text{hash}(m_1 \mid m_2))$, mac$(k, B)$

MitM

$m_1 = g^x \mod p$

$m_2 = g^y \mod p$

$k = \text{kdf}(g^{xy} \mod p)$

sign$(sk_A, \text{hash}(m_1 \mid m_2))$, mac$(k, A)$

sign$(sk_B, \text{hash}(m_1 \mid m_2))$, mac$(k, B)$

B

Knows $sk_B, pk_A$

$G = (g, p)$

$b = \text{dlog}(g^y \mod p)$

$k = \text{kdf}(g^{xy} \mod p)$
How likely is a discrete log-based attack?

Discrete Log Computation Records

- [Joux et al. 2005] 431-bit prime
- [Kleinjung et al. 2007] 530-bit prime
- [Bouvier et al. 2014] 596-bit prime
- [Kleinjung et al. 2017] 768-bit prime
- + other results for special groups

Best known generic technique: Number Field Sieve (NFS) and variants
How long does the number field sieve take?

Answer 1:

\[ L\left(1/3, 1.923\right) = \exp\left(1.923\left(\log N\right)^{1/3}\left(\log \log N\right)^{2/3}\right) \]
Computing Discrete Logs with NFS

How long does the number field sieve take?

**Answer 2:**

512-bit DH: $\approx 10$ core-years.

768-bit DH: $\approx 35,000$ core-years.

1024-bit DH: $\approx 45,000,000$ core-years.

2048-bit DH: Minimum recommended key size today.
Exploiting Pre-computation
(slide from N. Heninger)

<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: 3 hours
- 15 hours
- 120 hours
- 70 seconds
TLS-DHE in practice

Internet-wide scan of HTTPS servers using Zmap (2015)
- 14.3M hosts, 24% support DHE
- 70,000 distinct groups \((p,g)\)

Small-sized prime groups
- 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

What percentage of the internet does our TLS-DHE cryptographic proofs apply to?
- Depends on how powerful your adversary is
Exploiting Crypto Weaknesses: Weak Hash Functions
Authenticated DH (SIGMA)

$A$ knows $sk_A, pk_B$

$G = (g, p)$

$B$ knows $sk_B, pk_A$

$G = (g, p)$

$m_1 = g^x \mod p$

$m_2 = g^y \mod p$

$k = kdf(g^{xy} \mod p)$

Sign-and-MAC the transcript: prevents most MitM attacks
Authentication via Transcript Signatures

• Sign the full transcript
  – \texttt{sign}(sk_B, \texttt{hash}(m_1 \mid m_2))
  – \textit{Example}: TLS 1.3, SSH-2, TLS 1.2 client auth

• How weak can the \texttt{hash} function be?
  – do we need collision resistance?
  – do we only need 2\textsuperscript{nd} preimage resistance?
Quick Primer on Hash Functions

- Hash function: public function \( \{0, 1\}^* \rightarrow \{0, 1\}^n \)
  - Maps arbitrary-length message to fixed-length hash

- Mekle-Damgård mode: \( n \)-bit chain value
  - Process message iteratively
  - Use the message length in the padding (MD strengthening)

- Hash function should behave like a random function
  - Hard to find collisions, preimages
  - Hash can be used as a fingerprint
Hash Function Cryptanalysis

Collision attack

- Find $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$
- Generic attack with complexity $2^{n/2}$ (expected security)
- Shortcut attacks
  - MD5: complexity $2^{16}$
  - SHA1: complexity $2^{61}$

[Wang & al.'05, Stevens & al.'09]
[Wang & al.'05, Stevens '13]

Arbitrary common prefix/suffix, random collision blocks
Hash Function Cryptanalysis

**Chosen-prefix collision attack**

- Given $P_1, P_2$, find $M_1 \neq M_2$ such that $H(P_1 || M_1) = H(P_2 || M_2)$
- Generic attack with complexity $2^{n/2}$ (expected security)
- Shortcut attacks
  - MD5: complexity $2^{39}$
  - SHA1: complexity $2^{77}$

[Stevens & al.'09]
[Stevens '13]
2\textsuperscript{nd} preimage attack

- Given $M_1, H(M_1)$, find $M_2 \neq M_1$ s.t. $H(M_1) = H(M_2)$
- Generic attack with complexity $2^n$ (expected)
  - MD5: complexity $2^{128}$
  - SHA1: complexity $2^{160}$
  - No practical attacks
- Protocols that rely only on 2\textsuperscript{nd} preimage resistance can safely use even MD5
  - E.g. public key fingerprints in SSH
Hash Function Attack Complexity

- **MD5**: known attack complexities
  - **MD5 second preimage**: $2^{128}$ hashes (infeasible)
  - **MD5 generic collision**: $2^{64}$ hashes (months?)
  - **MD5 chosen-prefix collision**: $2^{39}$ hashes (1 hour)
  - **MD5 common-prefix collision**: $2^{16}$ hashes (seconds)

- **SHA1**: estimated attack complexities
  - **SHA1 second preimage**: $2^{160}$ hashes (infeasible)
  - **SHA1 generic collision**: $2^{80}$ hashes (infeasible)
  - **SHA1 chosen-prefix collision**: $2^{77}$ hashes (?)
  - **SHA1 common-prefix collision**: $2^{61}$ hashes (months)
Authentication via Transcript Signatures

• Sign the full transcript
  – \texttt{sign}(sk_B, \texttt{hash}(m_1 \mid m_2))
  – \textit{Example}: TLS 1.3, SSH-2, TLS 1.2 client auth

• How weak can the \texttt{hash} function be?
  – do we need collision resistance?
  – do we only need 2\textsuperscript{nd} preimage resistance?

• Is it still safe to use MD5, SHA-1 in TLS, IKE, SSH?
  – Cryptographers and practitioners \textit{disagree}
    (e.g. see Schneier vs. Hoffman, RFC4270)
Transcript Collisions on SIGMA

Can the attacker find and exploit collisions in this transcript hash?
Hash Collisions in SIGMA

Can the attacker find and exploit collisions in this transcript hash?
SLOTH: Transcript Collision Attacks

- Server Impersonation
- Client Impersonation

- Parameter Downgrade

Keys $k_a = \text{kdf}(g^{xy'} \mod p)$
Transcript $t_a = T(m_1, m'_2)$

Signatures:
- $\text{sign}(sk_A, \text{hash}(t_a)), \text{mac}(k, A)$
- $\text{sign}(sk_B, \text{hash}(t_a)), \text{mac}(k, B)$

- MitM

Keys $k_b = \text{kdf}(g^{x'y} \mod p)$
Transcript $t_b = T(m'_1, m_2)$

Signatures:
- $\text{sign}(sk_A, \text{hash}(t_b)), \text{mac}(k, A)$
- $\text{sign}(sk_B, \text{hash}(t_b)), \text{mac}(k, B)$

- Collision $\text{hash}(t_a) = \text{hash}(t_b)$
Computing a Transcript Collision

\[
\text{hash}(m_1 \mid m'_2) = \text{hash}(m'_1 \mid m_2)
\]

- We need to compute a collision, not a pre-image
  - Attacker controls parts of both transcripts
  - If we know the black bits, can we compute the red bits?
  - This can sometimes be set up as a generic collision

- If we’re lucky, we can set up a shortcut collision
  - Common-prefix: collision after a shared transcript prefix
  - Chosen-prefix: collision after attacker-controlled prefixes
Computing Transcript Collisions

A

hash

len₁

gₓ

paramsₐ

len₂′

gᵧ′

paramsₐ

MitM

Challenge: compute m₁′ before seeing m₂

m₁

m₁′

m₂′

m₂

B

hash

len₁′

gₓ′

paramsₐ

len₂

gᵧ

paramsₐ
Generic Transcript Collisions

A

hash

len₁

\(g^x\)

nonceₐ

len₂'

\(g^{y'}\)

nonceₙ

MitM

Try random nonces until collision

B

hash

len₁'

\(g^{x'}\)

nonceₙ

len₂

\(g^{static}\)

nonceₐ

N = 2^{\mid\text{hash}\mid/2}

MD5: 2^{64}

SHA-1: 2^{80}

HMAC/96: 2^{48}
Chosen-Prefix Transcript Collisions

A

\[
\text{len}_1 \\
g^x \\
\text{blob}_A
\]

MitM

B

\[
\text{len}_2 \\
g^y \\
\text{blob}_B
\]

Ephemeral DH key, arbitrary blob, known length \(\text{len}_2\)
Find Chosen-Prefix Collision $C_1$, $C_2$

$N = 2^{\text{CPC}(\text{hash})}$

MD5: $2^{39}$
SHA-1: $2^{77}$
Weak Hash Functions in TLS

TLS $\leq 1.1$ uses MD5 and SHA-1 for signatures

- RSA signatures over $\text{MD5}(t) \ || \ \text{SHA}-1(t)$
- DSA signatures over $\text{SHA}-1(t)$

TLS 1.2 introduces signatures with SHA-2 but allows negotiation of MD5, SHA-1

- RSA signatures over $\text{MD5}(t)$, or $\text{SHA}-1(t)$, or $\text{SHA}-256(t)$, or $\text{SHA}-224(t)$, or $\text{SHA}-384(t)$, or $\text{SHA}-512(t)$
- (EC)DSA signatures only over $\text{SHA}-1(t)$

TLS 1.2 client signatures using RSA-MD5 are vulnerable to transcript collision attacks
Exploiting Logical Flaws: Downgrade Attacks on Agile Key Exchange
Authenticated DH (SIGMA)

(A)

Knows $sk_A, pk_B$
$G = (g, p)$

$k = kdf(g^{xy} \mod p)$

(B)

Knows $sk_B, pk_A$
$G = (g, p)$

$m_1 = g^x \mod p$
$m_2 = g^y \mod p$

$k = kdf(g^{xy} \mod p)$

sign$(sk_A, \text{hash}(m_1 | m_2))$, mac$(k, A)$

sign$(sk_B, \text{hash}(m_1 | m_2))$, mac$(k, B)$

(Exercise)
DO YOU SEE THE BUG?
Agility: Negotiating DH Groups

Why? backwards compatibility, export regulations,...
Logjam: DH Group Downgrade Attack

The Logjam Attack [2015]

Remove Strong Groups

A

Knows $sk_A, pk_B$

$G_{2048}, G_{512}$

\[ [G_{2048}, G_{512}] \rightarrow [G_{512}] \]

\[ m_1 = g^x \mod p_{512} \]

\[ m_2 = g^y \mod p_{512} \]

\[ k = kdf(g^{xy} \mod p_{512}) \]

MitM

B

Knows $sk_B, pk_A$

$G_{2048}, G_{512}$

\[ [G_{512}] \rightarrow [G_{512}] \]

\[ b = dlog(g^y \mod p_{512}) \]

\[ k = kdf(g^{xy} \mod p_{512}) \]

sign($sk_A$, hash($m_1 | m_2$)), mac($k, A$)

sign($sk_B$, hash($m_1 | m_2$)), mac($k, B$)
TLS Variant of SIGMA

A

Knows $sk_A, pk_B$

$G_{2048}, G_{512}$

$m_1 = [G_{2048}, G_{512}]$

$m_2 = B, \text{sign}(sk_B, G_{2048} | g^y \mod p_{2048})$

$m_3 = A, g^x \mod p_{2048}$

$k = \text{kdf}(g^{xy} \mod p_{2048})$

B

Knows $sk_B, pk_A$

$G_{2048}, G_{512}$

Transcript MAC covers negotiation

Signature covers group
MACing the Handshake Transcript

TLS 1.2: mac the full transcript to prevent tampering

\[ \text{mac}(k, [G_{2048}, G_{512}] \mid G_{512} \mid m_1 \mid m_2) \]
Logjam Still Works
MACing the Handshake Transcript

TLS 1.2: mac the full transcript to prevent tampering

– \( \text{mac}(k, \left[ G_{2048}, G_{512} \right] | G_{512} | m_1 | m_2) \)

– but it is too late, because we already used \( G_{512} \)

\[ k = \text{kdf}(g^{xy} \mod p_{512}) \]

– so, the attacker can forge the \text{mac}

• The TLS 1.2 downgrade protection mechanism itself depends on downgradeable parameters.

– hence, the only fix is to find and disable all weak parameters: groups, curves, mac algorithms,...
What went wrong?

• Cryptographic weakness
  – Problem: Continued support for weak DH groups
  – Countermeasure: Ban all weak groups

• Logical protocol flaw
  – Problem: Downgrade attack on agile key exchange
  – Countermeasure: Protect integrity of key exchange even if the negotiated DH group is weak
Signing the Handshake Transcript

- **IKEv1**: both A and B sign the offered groups
  - \( \text{sign}(sk_B, \text{hash}([G_{2048}, G_{512}] \mid m_1 \mid m_2)) \)

- **IKEv2**: each signs its own messages
  - \( \text{sign}(sk_A, \text{hash}([G_{2048}, G_{512}] \mid m_1)) \)
  - \( \text{sign}(sk_B, \text{hash}(G_{512} \mid m_2)) \)

- **SSH-2 and TLS 1.3**: sign everything
  - \( \text{sign}(k, \text{hash}([G_{2048}, G_{512}] \mid G_{512} \mid m_1 \mid m_2)) \)
IKEv2 Variant of SIGMA

Exercise: show a variant of Logjam on this protocol

\[ m_1 = [G_{2048}, G_{512}] \]
\[ m_2 = G_{2048}, g^y \mod p_{2048} \]
\[ m_3 = g^x \mod p_{2048} \]

\[ k = \text{kdf}(g^{xy} \mod p_{2048}) \]

Sign your own messages
Signing the Handshake Transcript

• **IKEv1**: both A and B sign the offered groups
  – \( \text{sign}(sk_B, \text{hash}([G_{2048}, G_{512}] \mid m_1 \mid m_2)) \)
  – no agreement on chosen group!

• **IKEv2**: each signs its own messages
  – \( \text{sign}(sk_A, \text{hash}([G_{2048}, G_{512}] \mid m_1)) \)
  – \( \text{sign}(sk_B, \text{hash}(G_{512} \mid m_2)) \)
  – no agreement on offered groups!

• **SSH-2 and TLS 1.3**: sign everything
  – \( \text{sign}(k, \text{hash}([G_{2048}, G_{512}] \mid G_{512} \mid m_1 \mid m_2)) \)
  – works! (only if hash is collision-resistant)
SLOTH: Hash Function Downgrade

TLS 1.2 introduces signatures with SHA-2 but allows negotiation of MD5, SHA-1

• Attacker can downgrade TLS 1.2 connection from SHA-256 to MD5, and then apply transcript collision attacks (SLOTH)

What went wrong?

• Crypto Weakness:
  Continued support for RSA-MD5 signatures

• Logical Protocol flaw:
  Downgrade attack on signature algorithms extension

• Implementation bug:
  OpenSSL, GnuTLS, NSS accept MD5 signatures even if disabled
Exploiting Logical Flaws: Triple Handshake Attacks

(maybe later)
User authentication over TLS

Application-level Authentication
- **Outer**: server-authenticated TLS
- **Inner**: user authentication

Many examples of this pattern
- SASL, GSSAPI, EAP, ...
- TLS Renegotiation with client certificate

Inner authentication *endorses* unauthenticated TLS channel
- *Need to strongly bind the two protocol layers together!"
Generic credential forwarding attack
Simplified version of [Asokan, Niemi, Nyberg’02]

- Suppose $u$ uses same authentication credential at both $M$ and $S$
- $M$ forwards $S$’s authentication challenge to $C$
- $M$ forwards $C$’s response to $S$

- $M$ can log in as $u$ at $S$!
TLS renegotiation attack [2009]

Martin Rex’s Version

- Suppose $u$ uses same client cert to log in to both $M$ and $S$
- $M$ forwards $S$’s renegotiation request to $C$
- $M$ forwards renego handshake between $C$ and $S$
- $S$ concatenates data sent by $M$ to data sent by $u$!
Binding user auth to TLS channels

Computing a channel identifier ($cid$):
- $f(master\ secret)$ (EAP)
- $f(handshake\ log)$ (Renegotiation Indication, SASL)

Extract TLS-level channel identifier $cid$

Bind $cid$ to User authentication

Does not work if $M$ can ensure that $cid = cid'$
Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS

Karthikeyan Bhargavan*, Antoine Delignat-Lavaud*, Cédric Fournet†, Alfredo Pironti* and Pierre-Yves Strub‡

*INRIA Paris-Rocquencourt †Microsoft Research ‡IMDEA Software Institute

Details, demos at:
http://secure-resumption.com
**Triple Handshake attack: step 1**

**Key Synchronization Attack**

A malicious server $M$ can ensure that the master secrets in two different connections from $C-M$ and $M-S$ are the same.

**RSA Key Synchronization**

$M$ re-encrypts $C$’s premaster secret under $S$’s public key.

$M$ forces same ciphersuite and nonces on the two handshakes.

**DHE Key Synchronization**

$M$ chooses a “bad” (non-prime) Diffie-Hellman group.

Does not break single handshake theorems

“If a client completes with an honest server...”

**Breaks EAP compound authentication** (reenables 2002 attack)

The master secret is not a good channel identifier (it isn’t *contributive*).

Renegotiation indication channel identifier (handshake log) still works.
Transcript Synchronization Attack
After resumption, a malicious server $M$ can ensure that the master secrets, keys, and handshake logs on two different connections from $C-M$ and $M-S$ are the same.

Abbreviated agreement
Transcript depends only on master secret, ciphersuite, session ID (no certificates).

Does not break session resumption theorem
“If the server in the original handshake was honest...”

Breaks transcript-based channel identifiers
After resumption, handshake log is not a good channel identifier
Breaks tls-unique (SASL), renegotiation indication
User Impersonation Attack (reenables 2009 attack)

\(\text{cid} = \text{hash(abbreviated handshake log)}\) same on both connections
So \(M\) can forward renegotiation between \(C\) and \(S\) unchanged.

Surely this must break Giesen’s multi-handshake theorem?
Renegotiation with honest peer implies agreement on abbreviated handshake, but not on original handshake
Theorem needs honest peer in original handshake for agreement on all three

Impact
A malicious website can impersonate any user who uses client certificates on any other website that requires client certificate auth, and supports resumption and renegotiation
What went wrong?

• Logical protocol flaw
  – **Problem:** Key synchronization attack on RSA/DHE
  – **Countermeasure:** Independent keys per connection

• Logical protocol flaw
  – **Problem:** Transcript synchronization after resumption
  – **Countermeasure:** Independent master secrets per session
Exploiting Implementation Bugs: State Machine Attacks
TLS Implementation Bugs

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 attacks
Confusions between modes
Recall: the many modes of TLS

Protocol versions
- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges
- ECDHE, FFDHE, RSA, PSK, ...

Authentication modes
- ECDSA, RSA signatures, PSK, ...

Authenticated Encryption Schemes
- AES-GCM, CBC MAC-Encode-Encrypt, RC4, ...

100s of possible protocol combinations!
Implementing RSA Handshake

**Client**

- **ServerCertificate** ($m_3$)
  - $\text{cert}(pk_s)$

- **ClientKeyExchange** ($m_4$)
  - $\text{rsa-encrypt}(pms, pk_s)$

- **ClientFinished** ($m_5$)
  - $\text{mac}(m_1-m_4, K)$

- **ServerFinished** ($m_6$)
  - $\text{mac}(m_1-m_5, K)$

**Server**

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

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

- **ServerCertificate**($\text{cert}_s$)

- **ServerHelloDone**

- **ClientKeyExchange**($\text{rsaenc}(pms, pk_s)$)

- **ClientCCS**

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

- **ServerCCS**

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

- **ApplicationData**
Implementing DHE Handshake

Client

Server

ServerKeyExchange (m₃)
cert(pkₛ), rsa-sign(G | gʸ, skₛ)

ClientKeyExchange (m₄)

ServerKeyExchange (sign((G, gʸ), skₛ)

ClientFinished (m₅)
mac(m₁-m₄, K)

ServerFinished (m₆)
mac(m₁-m₅, K)

ClientHello(v, [kx₁, kx₂, ...])

ServerHello(v, kx = DHE|ECDHE)

ServerCertificate(certₛ)

ServerHelloDone

ClientKeyExchange(gₓ)

ClientCCS

ClientFinished(mac(log, gˣʸ))

ServerCCS

ServerFinished(mac(log', gˣʸ))

ApplicationData*
Composing Handshakes

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

$\Rightarrow$ RSA

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

$\Rightarrow$ ServerCertificate$(certs)$

$\Rightarrow$ ServerHelloDone

ClientKeyExchange$(rsaenc(pms, pk_S))$

$\Rightarrow$ ClientCCS

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

$\Rightarrow$ ServerCCS

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

$\Rightarrow$ ApplicationData$
$

(\text{EC})$DHE$

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

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

$\Rightarrow$ ServerCertificate$(certs)$

$\Rightarrow$ ServerKeyExchange$(\text{sign}((G, g^v), sk_S))$

$\Rightarrow$ ServerHelloDone

ClientKeyExchange$(g^x)$

$\Rightarrow$ ClientCCS

$\Rightarrow$ ClientFinished$(\text{mac}(log, g^x))$

$\Rightarrow$ ServerCCS

$\Rightarrow$ ServerFinished$(\text{mac}(log', g^{x'})$

$\Rightarrow$ ApplicationData
TLS State Machine

RSA + DHE + ECDHE
+ Session Resumption
+ Client Authentication

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

Do implementations conform to this state machine?
Many, Many Bugs

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

- Required messages can be skipped
- Unexpected messages can be received
Many, Many Bugs

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

- Required messages can be skipped
- Unexpected messages can be received

How come all these bugs?

- In independent code bases, sitting in there for years
- CVEs for many libraries
- Are they exploitable?
Culprit: Underspecified State Machine

TLS specifies a ladder diagram with optional messages

- Relies on the Finished messages to ensure agreement

![Ladder Diagram of TLS Handshake](image)

Figure 1. Message flow for a full handshake
Composing Key Exchanges

ClientHello($v, [kx_1, kx_2, \ldots]$) → RSA → ServerHello($v, kx = RSA$) → ServerCertificate($cert_S$) → ServerHelloDone → ClientKeyExchange($rsaenc(pms, pk_S$)) → ClientCCS → ClientFinished($mac(log, pms)$) → ServerCCS → ServerFinished($mac(log', pms)$) → ApplicationData

(ES)DHE

ClientHello($v, [kx_1, kx_2, \ldots]$) → ServerHello($v, kx = DHE|ECDHE$) → ServerCertificate($cert_S$) → ServerKeyExchange(sign($\langle G, g^v \rangle, sk_S$)) → ServerHelloDone → ClientKeyExchange($g^x$) → ClientCCS → ClientFinished($mac(log, g^xv)$) → ServerCCS → ServerFinished($mac(log', g^xv)$) → ApplicationData

ClientHello($v, [kx_1, kx_2, \ldots]$) → ServerHello($v, kx$) → ServerCertificate($cert_S$) → ServerKeyExchange(\ldots) → ServerHelloDone → ClientKeyExchange(\ldots) → ClientCCS → ClientFinished($mac(log, \ldots)$) → ServerCCS → ServerFinished($mac(log', \ldots)$) → 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 for the Internet: "be liberal in what you accept" (not for security!)

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

Clients should reject these cases
- But they don’t, so we are not running the TLS handshake any more
Recall: DHE Handshake

Client

Server

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

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

ServerCertificate($\text{cert}_S$)

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

ServerHelloDone

ClientKeyExchange($g^x$)

ClientKeyExchange($g^x$)

ClientFinished ($m_5$)

$\text{mac}(m_1-m_4, K)$

ServerFinished ($m_6$)

$\text{mac}(m_1-m_5, K)$

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

ServerCCS

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

ApplicationData*
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
A network attacker can impersonate any server (Paypal, Amazon, Google) to any Java TLS client (built with JSSE)

Affects all versions of Java until Jan 2015 CPU (CVE-2014-6593)

Other state machine bugs found in a dozen popular TLS libraries
FREAK:
State Machine Bugs
+ Downgrade Attacks
+ Crypto Weaknesses
RSA Key Transport

Client

ServerCertificate ($m_3$)
  cert($pk_s$)

ClientKeyExchange ($m_4$)
  rsa-encrypt($pms$, $pk_s$)

ClientFinished ($m_5$)
  mac($m_1$-$m_4$, $K$)

ServerFinished ($m_6$)
  mac($m_1$-$m_5$, $K$)

Session Key
$K = \text{PRF}(\ pms,\ nonce_C,\ nonce_s)$
RSA Key Transport

- Client chooses secret \( pms \), encrypts it with server’s public key \( pk_s \)

\[
\text{rsa-pkcs1-encrypt}(pms, pk_s) = [\text{pad} \mid p_{v_{max}} \mid pms]^e \mod pq
\]

- Server decrypts, checks pad and protocol version, computes session key from \( pms \)

*Security*: In theory, relies on hardness of factoring \( pq \)
# RSA Factoring Challenge

<table>
<thead>
<tr>
<th>RSA Number</th>
<th>Decimal digits</th>
<th>Binary digits</th>
<th>Cash prize offered</th>
<th>Factored on</th>
<th>Factored by</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA-140</td>
<td>140</td>
<td>463</td>
<td>US$17,226</td>
<td>February 2, 1999</td>
<td>Herman te Riele et al.</td>
</tr>
<tr>
<td>RSA-160</td>
<td>160</td>
<td>530</td>
<td></td>
<td>April 1, 2003</td>
<td>Jens Franke et al., University of Bonn</td>
</tr>
<tr>
<td>RSA-170[*]</td>
<td>170</td>
<td>563</td>
<td></td>
<td>December 29, 2009</td>
<td>D. Bonenberger and M. Krone[*][**]</td>
</tr>
<tr>
<td>RSA-576</td>
<td>174</td>
<td>576</td>
<td>$10,000 USD</td>
<td>December 3, 2003</td>
<td>Jens Franke et al., University of Bonn</td>
</tr>
<tr>
<td>RSA-190[*]</td>
<td>190</td>
<td>629</td>
<td></td>
<td>November 8, 2010</td>
<td>A. Timofeev and I. A. Popovyan</td>
</tr>
<tr>
<td>RSA-640</td>
<td>193</td>
<td>640</td>
<td>$20,000 USD</td>
<td>November 2, 2005</td>
<td>Jens Franke et al., University of Bonn</td>
</tr>
<tr>
<td>RSA-200[*]</td>
<td>200</td>
<td>663</td>
<td></td>
<td>May 9, 2005</td>
<td>Jens Franke et al., University of Bonn</td>
</tr>
<tr>
<td>RSA-210[*]</td>
<td>210</td>
<td>696</td>
<td></td>
<td>September 26, 2013</td>
<td>Ryan Propper</td>
</tr>
<tr>
<td>RSA-704[*]</td>
<td>212</td>
<td>704</td>
<td>$30,000 USD</td>
<td>July 2, 2012</td>
<td>Shi Bai, Emmanuel Thomé and Paul Zimmermann</td>
</tr>
</tbody>
</table>

**Best Generic Technique:** Number Field Sieve (NFS)

- Try CADO-NFS: http://cado-nfs.gforge.inria.fr/
How long does factoring take with the number field sieve?

**Answer 3**

- **512-bit RSA**: 7 months — large academic effort [Cavallar et al., 1999]
- **768-bit RSA**: 2.5 years — large academic effort [Kleinjung et al., 2009]
- **512-bit RSA**: 2.5 months — single machine [Moody, 2009]
- **512-bit RSA**: 72 hours — single Amazon EC2 machine [Harris, 2012]
- **512-bit RSA**: 7 hours — Amazon EC2 cluster [Heninger, 2015]
- **512-bit RSA**: < 4 hours — Amazon EC2 cluster [this work]
Factoring RSA keys in TLS

RSA encryption used in TLS 1.0-1.2

- If $pq$ can be factored into $p$ and $q$, an attacker can break TLS encryption, integrity
- 512-bit keys and 768-bit keys can be factored

Browsers now reject < 1024-bit RSA certs

- They will soon require $\geq$ 2048 bits
- So nobody today accepts 512-bit RSA keys, right?
Export-Grade Ciphers in TLS

In the 1990s, cryptography exports were controlled

- All software had two versions: domestic and export
- Export RSA keys, Diffie-Hellman groups limited to 512 bits
- Export symmetric crypto limited to 40 bit keys

International Traffic in Arms Regulations [April 1, 1992 version]

Category XIII--Auxiliary Military Equipment ...

(1) Cryptographic (including key management) systems, equipment, assemblies, modules, integrated circuits, components or software with the capability of maintaining secrecy or confidentiality of information or information systems...

Commerce Control List [current]

a.1.b.1. Factorization of integers in excess of 512 bits (e.g., RSA);
Export-Grade Ciphers in TLS

TLS 1.0 included many Export-grade ciphers

- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_RSA_EXPORT_WITH DES40_CBC_SHA
- TLS_DHE_RSA_EXPORT_WITH DES40_CBC_SHA
- TLS_DHE_DSS_EXPORT_WITH DES40_CBC_SHA

To support these, every TLS server had two sets of keys

- A 2048-bit RSA key for TLS_RSA + a 512-bit RSA key for TLS_RSA_EXPORT
- A 1025-bit DH group for TLS_DHE + a 512-bit DH group for TLS_DHE_EXPORT
- E.g. OpenSSL created a 512-bit RSA_EXPORT on startup
RSA_EXPORT support on the Web

In 2000, EXPORT deprecated in TLS 1.1, not used since

- (Dead) code still exists in OpenSSL and other libraries

In Mar 2015, many TLS servers still allow RSA_EXPORT!

- 8.9M (26.3%) HTTPS servers support EXPORT ciphers
- 36.7% of HTTPS servers with browser-trusted certificates
- 9.6% of Alexa top 1M HTTPS servers
- Reason: backwards compatibility with old TLS clients

Modern browsers do not support or offer RSA_EXPORT

- EXPORT ciphers are never negotiating, so problem solved?
- An implementation bug reenables RSA_EXPORT in clients!
RSA Key Transport

Client

ServerCertificate (m₃)
cert(pkₛ)

ClientKeyExchange (m₄)
rsa-encrypt(pms, pkₛ)

ClientFinished (m₅)
mac(m₁-m₄, K)

ServerFinished (m₆)
mac(m₁-m₅, K)

Session Key
K = PRF( pms, nonceₖ, nonceₛ)
RSA_EXPORT Key Transport

Client

ServerCertificate \((m_3)\)
\[
\text{cert}(pk_s), \text{rsa-sign}(pk_{512}, sk_s)
\]

ClientKeyExchange \((m_4)\)
\[
\text{rsa-encrypt}(pms, pk_{512})
\]

ClientFinished \((m_5)\)
\[
\text{mac}(m_1-m_4, K)
\]

ServerFinished \((m_6)\)
\[
\text{mac}(m_1-m_5, K)
\]

Server

Session Key
\[K = \text{PRF}(\text{pms, nonce}_c, \text{nonce}_s)\]
Client accepts 512-bit RSA_EXPORT keys during regular RSA handshake!
RSA_EXPORT State Machine Bugs in TLS

Affected Software:
- OpenSSL
- SecureTransport: Windows
- SecureTransport: Safari, iOS
- Oracle Java JSSE
- IBM Java JSSE
- Mono TLS

OpenSSL State Machine
Java State Machine
FREAK: Downgrade to RSA_EXPORT

A man-in-the-middle attacker can:

- impersonate servers that support RSA_EXPORT,
- at buggy clients that accept RSA_EXPORT keys in RSA handshakes
What went wrong?

• Cryptographic weakness
  – **Problem:** Continued support for RSA_EXPORT
  – **Countermeasure:** Disable EXPORT ciphersuites

• Logical protocol flaw
  – **Problem:** Signature ambiguity between RSA/RSA_EXPORT
  – **Countermeasure:** Signatures should cover transcript

• Implementation bug
  – **Problem:** Clients accept EXPORT even if disables
  – **Countermeasure:** Fix state machine composition
Part I: Summary

Real-world attacks exploit a combination of:
- Cryptographic weaknesses
- Logical protocol flaws
- Implementation bugs

Vulnerabilities in less-studied modes can break strong provably secure modes of the protocol
- Too many modes and corner cases to prove by hand

A need for automated protocol verification
- Tools for finding protocol flaws and implementation bugs
- Machine-checked proofs for real-world protocols
End of Part I