SSL/TLS

EJ Jung
10/18/10

Authenticity of Public Keys

**Problem**: How does Alice know that the public key she received is really Bob’s public key?
Distribution of Public Keys

- Public announcement or public directory
  - Risks: forgery and tampering

- Public-key certificate
  - Signed statement specifying the key and identity
    - $\text{sig}_{\text{Alice}}(\text{“Bob”, PK}_B)$

- Common approach: certificate authority (CA)
  - Single agency responsible for certifying public keys
  - After generating a private/public key pair, user proves his identity and knowledge of the private key to obtain CA’s certificate for the public key (offline)
  - Every computer is pre-configured with CA’s public key

Using Public-Key Certificates

Unassigned certificate: contains user ID, user’s public key

Generate hash code of unassigned certificate

Signed certificate: Recipient can verify signature using CA’s public key.

Authenticity of public keys is reduced to authenticity of one key (CA’s public key)
Hierarchical Approach

- Single CA certifying every public key is impractical
- Instead, use a trusted root authority
  - For example, Verisign
  - Everybody must know the public key for verifying root authority’s signatures
- Root authority signs certificates for lower-level authorities, lower-level authorities sign certificates for individual networks, and so on
  - Instead of a single certificate, use a certificate chain
    - \( \text{sig}_{\text{Verisign}}("UI", \text{PK}_{UI}) \), \( \text{sig}_{UI}("EJ Jung", \text{PK}_E) \)
  - What happens if root authority is ever compromised?

Alternative: “Web of Trust”

- Used in PGP (Pretty Good Privacy)
- Instead of a single root certificate authority, each person has a set of keys they “trust”
  - If public-key certificate is signed by one of the “trusted” keys, the public key contained in it will be deemed valid
- Trust can be transitive
  - Can use certified keys for further certification

Alice \[\rightarrow\] Friend of Alice \[\rightarrow\] Friend of friend \[\rightarrow\] Bob

\( \text{sig}_{\text{Alice}}("Friend", \text{Friend’s key}) \)
\( \text{sig}_{\text{Friend}}("Foaf", \text{Foaf’s key}) \)

I trust Alice
X.509 Authentication Service

- ITU-T standard
- Specifies certificate format
  - X.509 certificates are used in IPSec and SSL/TLS
- Specifies certificate directory service
  - For retrieving other users’ CA-certified public keys
- Specifies a set of authentication protocols
  - For proving identity using public-key signatures
- Does not specify crypto algorithms
  - Can use it with any digital signature scheme and hash function, but hashing is required before signing

X.509 Version 1

- Encrypt, then sign for authenticated encryption
  - Goal: achieve both confidentiality and authentication
  - E.g., encrypted, signed password for access control
- Does this work?
Attack on X.509 Version 1

- Receiving encrypted password under signature does not mean that the sender actually knows the password!
- Proper usage: sign, then encrypt

Denning-Sacco Protocol

- Goal: establish a new shared key $K_{AB}$ with the help of a trusted certificate service
**Attack on Denning-Sacco**

- Alice’s signature is **insufficiently explicit**
  - Does not say to whom and why it was sent
- Alice’s signature can be used to impersonate her

**Authentication with Public Keys**

- **Only Alice can create a valid signature**
- **Signature is on a fresh, unpredictable challenge**

Potential problem: Alice will sign anything
Mafia-in-the-Middle Attack [from Anderson’s book]

Early Version of SSL (Simplified)

Bob’s reasoning: I must be talking to Alice because...

- Whoever signed $N_B$ knows Alice’s private key... Only Alice knows her private key... Alice must have signed $N_B$... $N_B$ is fresh and random and I sent it encrypted under $K_{AB}$... Alice could have learned $N_B$ only if she knows $K_{AB}$... She must be the person who sent me $K_{AB}$ in the first message...
Breaking Early SSL

Charlie uses his legitimate conversation with Alice to impersonate Alice to Bob
  • Information signed by Alice is not sufficiently explicit

What is SSL / TLS?

- Transport Layer Security protocol, version 1.0
  • De facto standard for Internet security
  • "The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications"
  • In practice, used to protect information transmitted between browsers and Web servers

- Based on Secure Sockets Layers protocol, ver 3.0
  • Same protocol design, different algorithms

- Deployed in nearly every Web browser
SSL / TLS in the Real World

Application-Level Protection

- Protects against application-level threats (e.g., server impersonation), **NOT** against IP-level threats (spoofing, SYN flood, DDoS by data flood)

<table>
<thead>
<tr>
<th>physical</th>
<th>data link</th>
<th>network</th>
<th>transport</th>
<th>session</th>
<th>presentation</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>email, Web, NFS</td>
</tr>
</tbody>
</table>
History of the Protocol

- **SSL 1.0**
  - Internal Netscape design, early 1994?
  - Lost in the mists of time

- **SSL 2.0**
  - Published by Netscape, November 1994
  - Several weaknesses

- **SSL 3.0**
  - Designed by Netscape and Paul Kocher, November 1996

- **TLS 1.0**
  - Internet standard based on SSL 3.0, January 1999
  - **Not** interoperable with SSL 3.0
    - TLS uses HMAC instead of MAC; can run on any port

---

“Request for Comments”

- Network protocols are usually disseminated in the form of an RFC
- **TLS version 1.0** is described in RFC 2246
- Intended to be a self-contained definition of the protocol
  - Describes the protocol in sufficient detail for readers who will be implementing it and those who will be doing protocol analysis
  - Mixture of informal prose and pseudo-code
TLS Basics

- TLS consists of two protocols
  - Familiar pattern for key exchange protocols
- Handshake protocol
  - Use public-key cryptography to establish a shared secret key between the client and the server
- Record protocol
  - Use the secret key established in the handshake protocol to protect communication between the client and the server
- We will focus on the handshake protocol
TLS Handshake Protocol

- Two parties: client and server
- Negotiate version of the protocol and the set of cryptographic algorithms to be used
  - Interoperability between different implementations of the protocol
- Authenticate client and server (optional)
  - Use digital certificates to learn each other’s public keys and verify each other’s identity
- Use public keys to establish a shared secret

Handshake Protocol Structure
ClientHello

C

Client announces (in plaintext):
Protocol version he is running
Cryptographic algorithms he supports

S

ClientHello (RFC)

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites;
    CompressionMethod compression_methods;
} ClientHello

Highest version of the protocol supported by the client
Session id (if the client wants to resume an old session)
Set of cryptographic algorithms supported by the client (e.g., RSA or Diffie-Hellman)
ServerHello

Server responds (in plaintext) with:
Highest protocol version supported by both client and server
Strongest cryptographic suite selected from those offered by the client

ServerKeyExchange

Server sends his public-key certificate containing either his RSA, or his Diffie-Hellman public key (depending on chosen crypto suite)
ClientKeyExchange

C, Version\(_c\), suite\(_c\), N\(_c\) → Version\(_s\), suite\(_s\), N\(_s\), sig\(_ca\)(S, K\(_s\)), “ServerHelloDone”

Client generates some secret key material and sends it to the server encrypted with the server’s public key (if using RSA)

ClientKeyExchange (RFC)

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
    } exchange_keys
} ClientKeyExchange
```

```
struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret
```

Random bits from which symmetric keys will be derived (by hashing them with nonces)
"Core" SSL 3.0 Handshake

C, Version\_\textsubscript{C}=3.0, suite\_\textsubscript{C}, N\_\textsubscript{C}

Version\_\textsubscript{S}=3.0, suite\_\textsubscript{S}, N\_\textsubscript{S}, sig\_\textsubscript{CA}(S_{Ks}), “ServerHelloDone”

\{Secret\_\textsubscript{C}\}_{Ks}

If the protocol is correct, C and S share some secret key material (secret\_\textsubscript{C}) at this point

switch to key derived from secret\_\textsubscript{C}, N\_\textsubscript{C}, N\_\textsubscript{S}

switch to key derived from secret\_\textsubscript{C}, N\_\textsubscript{C}, N\_\textsubscript{S}

Version Rollback Attack

C, Version\_\textsubscript{C}=2.0, suite\_\textsubscript{C}, N\_\textsubscript{C}

Server is fooled into thinking he is communicating with a client who supports only SSL 2.0

Version\_\textsubscript{S}=2.0, suite\_\textsubscript{S}, N\_\textsubscript{S}, sig\_\textsubscript{CA}(S_{Ks}), “ServerHelloDone”

\{Secret\_\textsubscript{C}\}_{Ks}

C and S end up communicating using SSL 2.0 (weaker earlier version of the protocol that does not include “Finished” messages)
SSL 2.0 Weaknesses (Fixed in 3.0)

- Cipher suite preferences are not authenticated
  - “Cipher suite rollback” attack is possible
- Weak MAC construction
- SSL 2.0 uses padding when computing MAC in block cipher modes, but padding length field is not authenticated
  - Attacker can delete bytes from the end of messages
- MAC hash uses only 40 bits in export mode
- No support for certificate chains or non-RSA algorithms, no handshake while session is open

“Chosen-Protocol” Attacks

- Why do people release new versions of security protocols? Because the old version got broken!
- New version must be backward-compatible
  - Not everybody upgrades right away
- Attacker can fool someone into using the old, broken version and exploit known vulnerability
  - Similar: fool victim into using weak crypto algorithms
- Defense is hard: must authenticate version early
- Many protocols had “version rollback” attacks
  - SSL, SSH, GSM (cell phones)
**Version Check in SSL 3.0**

- C, Version C = 3.0, suite C, N_C
- Version S = 3.0, suite S, N_S
- \( \text{sig}_{C}(S, K_s) \)
- "ServerHelloDone"

If the protocol is correct, C and S share some secret key material secret C at this point.

- Switch to key derived from secret C, N_C, N_S
- Switch to key derived from secret S, N_C, N_S

**SSL/TLS Record Protection**

- Application Data
- Fragment
- Compress
- Add MAC
- Encrypt
- Append SSL Record Header

Use symmetric keys established in handshake protocol.