Lecture 13

Transport Layer Security/
Secure Socket Layer (TLS/SSL)

(Chapter 9 in KPS)
SSL: Secure Sockets Layer

- widely deployed security protocol
  - supported by almost all browsers, web servers
  - the “s” in https
  - billions $/year over SSL
- mechanisms: [Woo 1994], implementation: Netscape
- variation - TLS: transport layer security, RFC 2246
- provides
  - confidentiality
  - integrity
  - authentication
- original goals:
  - Web e-commerce transactions
  - encryption (especially credit-card numbers)
  - Web-server authentication
  - optional client authentication
  - minimum hassle in doing business with new merchant
- available to all TCP applications
  - secure socket interface
SSL and TCP/IP

- SSL provides application programming interface (API) to applications
- C and Java SSL libraries/classes readily available

```
<table>
<thead>
<tr>
<th>Application</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>SSL</td>
</tr>
<tr>
<td>IP</td>
<td>TCP</td>
</tr>
</tbody>
</table>

normal application  application with SSL
Could do something like PGP:

- Pretty Good Privacy (PGP) used to secure email
- $K_A$ is Alice’s private key used to sign
- $K_B$ is Bob’s public key used to encrypt to Bob a session key $K_S$

- but want to send byte streams & interactive data
- want set of secret keys for entire connection
- want certificate exchange as part of protocol: handshake phase
Toy SSL: a Simple Secure Channel

- **handshake**: Alice and Bob use their certificates, private keys to authenticate each other and exchange a shared secret

- **key derivation**: Alice and Bob use shared secret to derive set of keys

- **data transfer**: data to be transferred is broken up into series of records

- **connection closure**: special messages to securely close connection
Toy: a Simple Handshake

$K_B(MS) = EMS$

$MS$: master secret

$EMS$: encrypted master secret
Toy: Key Derivation

- considered bad to use same key for more than one cryptographic operation
  - use different keys for message authentication code (MAC) and encryption

- four keys:
  - $K_c = \text{encryption key for data sent from client to server}$
  - $M_c = \text{MAC key for data sent from client to server}$
  - $K_s = \text{encryption key for data sent from server to client}$
  - $M_s = \text{MAC key for data sent from server to client}$

- keys derived from key derivation function (KDF)
  - takes master secret and (possibly) some additional random data and creates the keys
Toy: Data Records

- why not encrypt data in constant stream as we write it to TCP?
  - where would we put the MAC? If at end, no message integrity until all data processed.
  - e.g., with instant messaging, how can we do integrity check over all bytes sent before displaying?
- instead, break stream in series of records
  - each record carries a MAC
  - receiver can act on each record as it arrives
- issue: in record, receiver needs to distinguish MAC from data
  - want to use variable-length records
Toy: Sequence Numbers

- **problem**: attacker can capture and replay record or re-order records

- **solution**: put sequence number into MAC:
  - \( \text{MAC} = \text{MAC}(M_x, \text{sequence}||\text{data}) \)
  - note: no sequence number field, \( M_x = \text{MAC} \text{ key} \)

- **problem**: attacker could replay all records

- **solution**: use nonce
**Toy: Control Information**

- **Problem**: truncation attack:
  - attacker forges TCP connection close segment
  - one or both sides thinks there is less data than there actually is

- **Solution**: record types, with one type for closure
  - type 0 for data; type 1 for closure

- MAC = MAC($M_x$, sequence||type||data)

```
<table>
<thead>
<tr>
<th>length</th>
<th>type</th>
<th>data</th>
<th>MAC</th>
</tr>
</thead>
</table>
```
Toy SSL: Summary

- hello
- certificate, nonce
- $K_B(MS) = EMS$
- type 0, seq 1, data
- type 0, seq 2, data
- type 0, seq 1, data
- type 0, seq 3, data
- type 1, seq 4, close
- type 1, seq 2, close

encrypted

bob.com
Toy SSL isn’t complete

- how long are fields?
- which encryption algorithms to use?
- want negotiation?
  - allow client and server to support different encryption algorithms
  - allow client and server to choose together specific algorithm before data transfer
SSL Cipher Suite

- cipher suite
  - public-key algorithm
  - symmetric encryption algorithm
  - MAC algorithm

- SSL supports several cipher suites

- negotiation: client, server agree on cipher suite
  - client offers choice
  - server picks one

common SSL symmetric ciphers
- DES – Data Encryption Standard: block
- 3DES – Triple strength: block
- RC2 – Rivest Cipher 2: block
- RC4 – Rivest Cipher 4: stream

SSL Public key encryption
- RSA
Real SSL: Handshake (1)

**Purpose**

1. server authentication
2. negotiation: agree on crypto algorithms
3. establish keys
4. client authentication (optional)
Real SSL: Handshake (2)

1. client sends list of algorithms it supports, along with client nonce
2. server chooses algorithms from list; sends back: choice + certificate + server nonce
3. client verifies certificate, extracts server’s public key, generates pre_master_secret, encrypts with server’s public key, sends to server
4. client and server independently compute encryption and MAC keys from pre_master_secret and nonces
5. client sends a MAC of all the handshake messages
6. server sends a MAC of all the handshake messages
last 2 steps protect handshake from tampering

- client typically offers range of algorithms, some strong, some weak
- man-in-the middle could delete stronger algorithms from list
- last 2 steps prevent this
  - last two messages are encrypted
Real SSL: Handshake (4)

- why two random nonces?
- suppose Trudy sniffs all messages between Alice & Bob
- next day, Trudy sets up TCP connection with Bob, sends exact same sequence of records
  - Bob (Amazon) thinks Alice made two separate orders for the same thing
  - solution: Bob sends different random nonce for each connection. This causes encryption keys to be different on the two days
  - Trudy’s messages will fail Bob’s integrity check
SSL Record Protocol

record header: content type; version; length

MAC: includes sequence number, MAC key $M_x$

fragment: each SSL fragment $2^{14}$ bytes (~16 Kbytes)
SSL Record Format

- Content type: 1 byte
- SSL version: 2 bytes
- Length: 3 bytes
- Data
- MAC

Data and MAC encrypted (symmetric algorithm)
Real SSL Connection

handshake: ClientHello

handshake: ServerHello

handshake: Certificate

handshake: ServerHelloDone

handshake: ClientKeyExchange

ChangeCipherSpec

handshake: Finished

ChangeCipherSpec

handshake: Finished

application_data

application_data

Alert: warning, close_notify

tcp FIN message follows

everything henceforth is encrypted
Key Derivation

- client nonce, server nonce, and pre-master secret input into pseudo random-number generator (PRG).
  - produces master secret
- master secret and new nonces input into another random-number generator: “key block”

- key block sliced and diced:
  - client MAC key
  - server MAC key
  - client encryption key
  - server encryption key
  - client initialization vector (IV)
  - server initialization vector (IV)