Overview

- Introduction to the SSL / TLS protocol
  - Widely deployed, “real-world” security protocol

- Protocol analysis case study
  - Start with the RFC describing the protocol
  - Create an abstract model and code it up in Murϕ
  - Specify security properties
  - Run Murϕ to check whether security properties are satisfied

- This lecture is a compressed version of what you will be doing in your project!
What is SSL / TLS?

- Transport Layer Security protocol, ver 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

Wells Fargo Account Summary - Microsoft Internet Explorer

Account Summary

Wells Fargo Accounts   OneLook Accounts

Tip: Select an account's balance to access the Account History.

Cash Accounts

<table>
<thead>
<tr>
<th>Account</th>
<th>Account Number</th>
<th>Available Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To end your session, be sure to Sign Off.

Account Summary | Brokerage | BillPay | Transfer | My Message Center | Sign Off

©1995 - 2003 Wells Fargo. All rights reserved.
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 problems (next slide)

- **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
SSL 2.0 Vulnerabilities

- **Short key length**
  - In export-weakened modes, SSL 2.0 unnecessarily weakens the authentication keys to 40 bits.

- **Weak MAC construction**

- **Message integrity vulnerability**
  - SSL 2.0 feeds padding bytes into the MAC in block cipher modes, but leaves the padding-length unauthenticated, may allow active attackers to delete bytes from the end of messages.

- **Ciphersuite rollback attack**
  - An active attacker may edits the list of ciphersuite preferences in the hello messages to invisibly force both endpoints to use a weaker form of encryption.
Let’s get going with SSL/TLS

Intruder Model

Formal Protocol

Informal Protocol Description

Find error

Analysis Tool

RFC (request for comments)

Security Properties
Request for Comments

- Network protocols are defined in 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 (that’s you!)
  - Mixture of informal prose and pseudo-code
- Read some RFCs to get a flavor of what protocols look like when they emerge from the committee
Evolution of the SSL/TLS RFC

- SSL 2.0
- SSL 3.0
- TLS 1.0

Page count

Bar chart showing the page count for SSL 2.0, SSL 3.0, and TLS 1.0.
TLS Basics

- TLS consists of two 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

C

ClientHello

ServerHello,
[Certificate],
[ServerKeyExchange],
[CertificateRequest],
ServerHelloDone

C

[Certificate],
ClientKeyExchange,
[CertificateVerify]

switch to negotiated cipher

Finished

S

switch to negotiated cipher

Finished
The handshake protocol may be executed in an abbreviated form to resume a previously established session

- No authentication, key material not exchanged
- Session resumed from an old state

For complete analysis, have to model both full and abbreviated handshake protocol

- This is a common situation: many protocols have several branches, sub-protocols for error handling, etc.
Rational Reconstruction

- Begin with simple, intuitive protocol
  - Ignore client authentication
  - Ignore verification messages at the end of the handshake protocol
  - Model only essential parts of messages (e.g., ignore padding)
- Execute the model checker and find a bug
- Add a piece of TLS to fix the bug and repeat
  - Better understand the design of the protocol
Protocol Model

- Informal Protocol Description
- Formal Protocol
- Intruder Model
- Murphi code
- Analysis Tool
- Security Properties
Protocol Step by Step: ClientHello

Client Hello

Client announces (in plaintext):
- Protocol version he is running
- Cryptographic algorithms he supports
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites;
    CompressionMethod compression_methods;
} ClientHello
ClientHello (Murφ)

ruleset i: ClientId do
ruleset j: ServerId do
  rule "Client sends ClientHello to server (new session)"
    cli[i].state = M_SLEEP &
    cli[i].resumeSession = false
  ==> 
  var
    outM: Message; -- outgoing message
  begin
    outM.source := i; outM.dest := j;
    outM.session := 0; outM.mType := M_CLIENT_HELLO;
    outM.version := cli[i].version; outM.suite := cli[i].suite;
    outM.random := freshNonce(); multisetadd (outM, cliNet);
    cli[i].state := M_SERVER_HELLO;
  end; end; end;
ServerHello

\[ C, \text{ Version}_C, \text{ suite}_C, N_C \]

Server responds (in plaintext) with:

- Highest protocol version both client & server support
- Strongest cryptographic suite selected from those offered by the client
ServerHello (Murφ)

ruleset i: ServerId do
  choose l: serNet do
  rule “Server receives ServerHello (new session)”
    ser[i].clients[0].state = M_CLIENT_HELLO &
    serNet[l].dest = i &
    serNet[l].session = 0
  ==> var
    inM: Message; -- incoming message
    outM: Message; -- outgoing message
  begin
    inM := serNet[l]; -- receive message
    if inM.mType = M_CLIENT_HELLO then
      outM.source := i;          outM.dest := inM.source;
      outM.session := freshSessionId();          outM.mType := M_SERVER_HELLO;
      outM.version := ser[i].version;         outM.suite := ser[i].suite;
      outM.random := freshNonce(); multisetadd (outM, serNet);
      ser[i].state  := M_SERVER_SEND_KEY;
    end; end; end;
ServerKeyExchange

C, Version\textsubscript{c}, suite\textsubscript{c}, N\textsubscript{c} → Server responds with his public-key certificate containing either his RSA, or his Diffie-Hellman public key (depending on chosen crypto suite)

Version\textsubscript{s}, suite\textsubscript{s}, N\textsubscript{s}, ServerKeyExchange → S
Symbolic Cryptography

- We will use abstract data types to model cryptographic operations
  - Assumes that cryptography is perfect
  - No details of the actual cryptographic schemes
  - Ignores bit length of keys, random numbers, etc.

- Simple notation for encryption, signatures, hashes
  - \( \{M\}_k \) is message M encrypted with key k
  - \( \text{sig}_k(M) \) is message M digitally signed with key k
  - \( \text{hash}(M) \) for the result of hashing message M with a cryptographically strong hash function
Client generates some secret key material and sends it to the server encrypted with the server’s public key
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
    } exchange_keys
} ClientKeyExchange

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

Let’s model this as \{Secret_c\}_Ks
If the protocol is correct, C and S share some secret key material $\text{secret}_c$ at this point.

Switch to key derived from $\text{secret}_c$. 
Participants as Finite-State Machines

Mᵦᵣᵦᵦ rules define a finite-state machine for each protocol participant

Client state

- M_SLEEP
- M_SERVER_HELLO
- M_SERVER_KEY
- M_SEND_KEY

Server state

- M_CLIENT_HELLO
- M_SEND_KEY
- M_CLIENT_KEY
- M_DONE

Transitions:
- ClientHello: M_SLEEP → M_SERVER_HELLO
- ServerHello: M_CLIENT_HELLO → M_SEND_KEY
- ServerKeyExchange: M_SEND_KEY → M_CLIENT_KEY
- ClientKeyExchange: M_SEND_KEY → M.Done
Intruder Model

Informal Protocol Description

Formal Protocol

Intruder Model

Murphi code: similar for all protocols

Find error

Analysis Tool

Security Properties
Intruder Can Intercept

- Store a message from the network in the data structure modeling intruder’s “knowledge”

```plaintext
ruleset i: IntruderId do
    choose l: cliNet do
        rule "Intruder intercepts client's message"
            cliNet[l].fromIntruder = false
        ==> begin
            alias msg: cliNet[l] do -- message from the net
                ...
                alias known: int[i].messages do
                    if multisetcount(m: known, msgEqual(known[m], msg)) = 0 then
                        multisetadd(msg, known);
                    end; end; end;
```
Intruder Can Decrypt if Knows Key

- If the key is stored in the data structure modeling intruder’s “knowledge”, then read message

```
ruleset i: IntruderId do
    choose l: cliNet do
        rule "Intruder intercepts client's message"
            cliNet[l].fromIntruder = false
        ==>begin
            alias msg: cliNet[l] do -- message from the net
                ...
                if msg.mType = M_CLIENT_KEY_EXCHANGE then
                    if keyEqual(msg.encKey, int[i].publicKey.key) then
                        alias sKeys: int[i].secretKeys do
                            if multisetcount(s: sKeys,
                                keyEqual(sKeys[s], msg.secretKey)) = 0 then
                                multisetadd(msg.secretKey, sKeys);
                            end;
                        end;
                    end;
                end;
```
Intruder Can Create New Messages

- Assemble pieces stored in the intruder’s “knowledge” to form a message of the right format

```plaintext
ruleset i: IntruderId do
ruleset d: ClientId do
ruleset s: ValidSessionId do
  choose n: int[i].nonces do
ruleset version: Versions do
  rule "Intruder generates fake ServerHello"
    cli[d].state = M_SERVER_HELLO
  ==> var
    outM: Message; -- outgoing message
begin
  outM.source := i; outM.dest := d; outM.session := s;
  outM.mType := M_SERVER_HELLO; outM.version := version;
  outM.random := int[i].nonces[n]; multisetadd (outM, cliNet);
end; end; end; end;
```
Intruder Model and Cryptography

- Perfect/symbolic cryptography model
- Our assumption that cryptography is perfect is reflected in the absence of certain intruder rules
  - There is no rule for creating a digital signature with a key that is not known to the intruder
  - There is no rule for reading the contents of a message which is marked as “encrypted” with a certain key, when this key is not known to the intruder
  - There is no rule for reading the contents of a “hashed” message
Specify Security Properties

- Informal Protocol Description
- Formal Protocol
- Intruder Model
- Analysis Tool
- Find error
- Security Properties

Murphi invariants
Secrecy

- Intruder should not be able to learn the secret generated by the client

```plaintext
ruleset i: ClientId do
  ruleset j: IntruderId do
    rule "Intruder has learned a client's secret"
      cli[i].state = M_DONE &
      multisetcount(s: int[j].secretKeys,
        keyEqual(int[j].secretKeys[s], cli[i].secretKey)) > 0
      ==> begin
        error "Intruder has learned a client's secret"
      end;
  end;
end;
```
After the protocol has finished, client and server should agree on their shared secret

```
ruleset i: ServerId do
  ruleset s: SessionId do
    rule "Server's shared secret is not the same as its client's"
      ismember(ser[i].clients[s].client, ClientId) &
      ser[i].clients[s].state = M_DONE &
      cli[ser[i].clients[s].client].state = M_DONE &
      !keyEqual(cli[ser[i].clients[s].client].secretKey,
                ser[i].clients[s].secretKey)
    ==>
    begin
      error "S's secret is not the same as C's"
    end;
```
Version and Crypto Suite Consistency

- Client and server should be running the highest version of the protocol they both support

```plaintext
ruleset i: ServerId do
ruleset s: SessionId do
    rule "Server has not learned the client's version or suite correctly"
        !ismember(ser[i].clients[s].client, IntruderId) &
        ser[i].clients[s].state = M_DONE &
        cli[ser[i].clients[s].client].state = M_DONE &
        (ser[i].clients[s].clientVersion != MaxVersion | ser[i].clients[s].clientSuite.text != 0)
    ==> begin
        begin
            error "Server has not learned the client's version or suite correctly"
        end; end; end;
```
Finite-State Verification

- $\text{Mur}\varphi$ rules for protocol participants and the intruder define a nondeterministic state transition graph.
- $\text{Mur}\varphi$ will exhaustively enumerate all graph nodes.
- $\text{Mur}\varphi$ will verify whether specified security conditions hold in every reachable node.
- If not, the path to the violating node will describe the attack.
When Does Murφ Find a Violation?

 Bad abstraction
- Removed too much detail from the protocol when constructing the abstract model
- Add the piece that fixes the bug and repeat
- This is part of the rational reconstruction process

 Genuine attack
- Success!
- Attacks found by formal analysis are usually quite strong: independent of specific cryptographic schemes, OS implementation, etc.
- Test an implementation of the protocol, if available
“Core” SSL 3.0

C, Version$_{C}$=3.0, suite$_{C}$, N$_{C}$

Version$_{S}$=3.0, suite$_{S}$, N$_{S}$, sig$_{ca}(S,K_{S})$, “ServerHelloDone”

{Secret$_{C}$)$_{K_{S}}$

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

switch to key derived from secret$_{C}$

switch to key derived from secret$_{C}$
Version Consistency Fails!

C, Version = 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,K\textsubscript{s}), “ServerHelloDone”

{Secret\textsubscript{c}}\textsubscript{Ks}

C and S end up communicating using SSL 2.0 (weaker earlier version of the protocol)
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: ClientDiffieHellmanPublic;
    } exchange_keys
} ClientKeyExchange

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

Model this as \( \{Version_c, Secret_c\}_{K_s} \)

This piece matters! Need to add it to the model.
Fixed “Core” SSL

C, Version\_c=3.0, suite\_c, N\_c

Version\_s=3.0, suite\_s, N\_s, sig\_ca(S,K\_s), “ServerHelloDone”

Prevents version rollback attack

\{Version\_c, Secret\_c\}\_K_s

Add rule to check that received version is equal to version in ClientHello

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

switch to key derived from secret\_c
Summary of Reconstruction

- A = Basic protocol
- C = A + certificates for public keys
  - Authentication for client and server
- E = C + verification (Finished) messages
  - Prevention of version and crypto suite attacks
- F = E + nonces
  - Prevention of replay attacks
- Z = “Correct” subset of SSL
Anomaly (Protocol F)

C

... Suite ... Modify

... Suite ...

Switch to negotiated cipher

C

Finished

data

S

Finished

data

X

X
Protocol Resumption

SessionId, $\text{Ver}_C = 3.0, N_C, \ldots$

$\text{Ver}_S = 3.0, N_S, \ldots$

Finished

data

C

S

Finished

data
Version Rollback Attack

SSL 2.0 Finished messages do not include version numbers or cryptosuites
Basic Pattern for Doing Your Project

- Read and understand protocol/system specification
  - Typically an RFC or a research paper
- Choose a tool
  - Murϕ by default, but we’ll describe other tools
- Start with a simple (possibly flawed) model
  - Rational reconstruction is a good way to go
- Give careful thought to security conditions, intruder model
Project suggestions

- Electronic Voting Protocols/Systems
  - 2 projects
- IKEv2 – IPSec key exchange standard
  - 2 projects
- Trusted Computing TPM Spec
  - 2 projects
- Secure routing protocols
- Protocols for anonymous communication
  - Tor may be worth a look
- Secure sensor network protocols
- Formalization of privacy laws
  - GLBA is a good candidate

Project discussion: Sept 18, 20 (Tue-Th): 4:30-6PM in CIC 2118
Background Reading on SSL 3.0

Optional, for deeper understanding of SSL / TLS

  - Nice study of an early proposal for SSL 3.0
- D. Bleichenbacher. “Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1”. CRYPTO ’98.
  - Cryptography is not perfect: this paper breaks SSL 3.0 by directly attacking underlying implementation of RSA
Questions?