SSL / TLS Case Study
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, 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
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…
Let’s Get Going…

Informal protocol description

RFC (request for comments)

Analysis tool

Find error

Formal specification

Intruder model
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 (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
From RFC to Mur$\varphi$ Model

- Informal protocol description
- RFC (request for comments)
- Find error
- Formal specification
- Mur$\varphi$ code
- Intruder model
- Analysis Tool
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

ClientHello

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

[Certificate],
ClientKeyExchange,
[CertificateVerify]

switch to negotiated cipher

Finished

switch to negotiated cipher

Finished
Abbreviated Handshake

- 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, subprotocols 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 Step by Step: ClientHello

ClientHello

Client announces (in plaintext):
• Protocol version he is running
• Cryptographic algorithms he supports
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)
- Cryptographic algorithms supported by the client (e.g., RSA or Diffie-Hellman)
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, Version_{C}, suite_{C}, N_{C}

ServerHello

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

C, Version$_C$, suite$_C$, $N_C$

$\rightarrow$

$\leftarrow$

Version$_S$, suite$_S$, $N_S$, sig$_{ca}(S,K_S)$, "ServerHelloDone"

ClientKeyExchange

Client generates some secret key material and sends it to the server encrypted with the server’s public key.
ClientKeyExchange (RFC)

```c
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 $\{\text{Secret}_c\}_K$
If the protocol is correct, C and S share some secret key material (secret_subscript_c) at this point.

C, Version_subscript_c, suite_subscript_c, N_c

Version_subscript_s, suite_subscript_s, N_s, sigsubscript_ca(S,K_s), "ServerHelloDone"

{Secret_subscript_c}_K_s

switch to key derived from secret_subscript_c

switch to key derived from secret_subscript_c
Murφ rules define a finite-state machine for each protocol participant.
Intruder Model

Informal protocol description

RFC (request for comments)

Formal specification

Murφ code

Find error

Analysis tool

Intruder model

Murφ code, similar for all protocols
Intruder Can Intercepts

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

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;
    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[i].fromIntruder = false
        ==> begin
            alias msg: cliNet[i] 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;
            end;
        end;
    end;
```
Intruder Can Create New Messages

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

```
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

- There is no actual cryptography in this model
  - Messages are marked as “encrypted” or “signed”, and the intruder rules respect these markers

- The assumption that cryptography is perfect is reflected by 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
Running Murϕ Analysis

Informal protocol description

Formal specification

Intruder model

RFC (request for comments)

Murϕ code

Find error

Murϕ code, similar for all protocols

Specify security conditions and run Murϕ
Secrecy

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

ruleset i: ClientId do
  ruleset j: IntruderId do
    rule "Intruder has learned a client's secret"
      cli[i].state = M_DONE &
      multisectcount(s: int[j].secretKeys,
          keyEqual(int[j].secretKeys[s], cli[i].secretKey)) > 0
    ==> begin
      begin
        error "Intruder has learned a client's secret"
      end;
      end;
  end;
end;
Shared Secret Consistency

◆ 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;
  end;
end;
Version and Crypto Suite Consistency

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

```
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
      error "Server has not learned the client's version or suite correctly"
    end;
  end;
end;
```
Finite-State Verification

- Mur$\varphi$ rules for protocol participants and the intruder define a nondeterministic state transition graph.
- Mur$\varphi$ will exhaustively enumerate all graph nodes.
- 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.

Correctness condition violated
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
  ● Yay! Hooray!
  ● 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

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

\text{Version\textsubscript{s}=2.0, suite\textsubscript{s}, N\textsubscript{s},
\text{sig}_{\text{ca}}(S,K\textsubscript{s}),
"ServerHelloDone"

\{Secret\textsubscript{c}\}_{K\textsubscript{s}}

C and S end up communicating using SSL 2.0 (weaker earlier version of the protocol)
A Case of Bad Abstraction

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

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

Model this as \( \{\text{Version}_c, \text{Secret}_c\}_K \)

This piece matters! Need to add it to the model.
Fixed “Core” SSL 3.0

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

\textbf{Prevents version rollback attack}

\{Version\textsubscript{C}, Secret\textsubscript{C}\}\_{K\textsubscript{S}}

Version\textsubscript{S}=3.0, suite\textsubscript{S}, N\textsubscript{S}, sig\textsubscript{ca}(S,K\textsubscript{S}), “ServerHelloDone”

\textbf{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\textsubscript{C} at this point

switch to key derived from secret\textsubscript{C}  \hspace{1cm} \text{switch to key derived from secret\textsubscript{C}}
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
Basic Pattern for Doing Your Project

◆ Read and understand protocol specification
  • Typically an RFC or a research paper
  • Website has some protocol specs, I’ll put up more

◆ Choose a tool
  • Murϕ by default, but I’ll describe many other tools
  • Play with Murϕ now to get some experience (installing, running simple models, etc.)

◆ Start with a simple (possibly flawed) model
  • Rational reconstruction is a good way to go

◆ Give careful thought to security conditions
Background Reading on SSL 3.0

Optional, for deeper understanding of SSL / TLS

  • Nice study of an early proposal for SSL 3.0
  • Murϕ analysis of SSL 3.0 (similar to this lecture)
  • Actual Murϕ model is in /projects/shmat/Murphi3.1/ex/secur
  • Cryptography is not perfect: this paper breaks SSL 3.0 by directly attacking underlying implementation of RSA