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, 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

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
  ◆ Badly broken
◆ 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

Let’s Get Going...
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

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

C

S

ClientHello (RFC)

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

S

ServerHello (RFC)

struct {
  ProtocolVersion server_version;
  CipherSuite cipher_suites;
  CompressionMethod compression_methods;
} 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

end;

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;

“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 \( k \)
   • \( \text{sig}(M, k) \) 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 (RFC)

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

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

“Core” SSL

If the protocol is correct, C and S share some secret key material secretc, at this point
switch to key derived from secretc

Server responds with his public-key certificate containing either his RSA, or
his Diffie-Hellman public key (depending on chosen crypto suite)
Participants as Finite-State Machines

Murphy rules define a finite-state machine for each protocol participant.

Intruder Model

Informal Protocol Description
Formal Protocol
Intruder Model

Find error
Analysis Tool

Intruder Model and Cryptography

There is no actual cryptography in our model
- Messages are marked as "encrypted" or "signed", and the intruder rules respect these markers
- 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
Running Murϕ Analysis

Intruder Model

Intruder Analysis Tool

Formal Protocol

Informal Protocol Description

Find error

Specify security conditions and run Murϕ

Secrecy

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

```murϕ
def ruleset i: ClientId do
    def 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;
```

Shared Secret Consistency

- After the protocol has finished, client and server should agree on their shared secret

```murϕ
def ruleset i: ServerId do
    def 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

```murϕ
def ruleset i: ServerId do
    def 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

- Murϕ rules for protocol participants and the intruder define a nondeterministic state transition graph
- Murϕ will exhaustively enumerate all graph nodes
- Murϕ 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

C, Version=3.0, suitec, Nc

Version=3.0, suitec, Nc, sigca(S,Ks), "ServerHelloDone"

(Secretc)cKs

If the protocol is correct, C and S share some secret key material secretc at this point
switch to key derived from secretc

S

A Case of Bad Abstraction

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

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

This piece matters! Need to add it to the model.

Fixed "Core" SSL

C, Version=3.0, suitec, Nc

Version=3.0, suitec, Nc, sigca(S,Ks), "ServerHelloDone"

{Versionc, Secretc}Ks

If the protocol is correct, C and S share some secret key material secretc at this point
switch to key derived from secretc

Prevent version rollback attack
Add rule to check that received version is equal to version in ClientHello

Version Consistency Fails!

C, Version=2.0, suitec, Nc

Version=2.0, suitec, Nc, sigca(S,Ks), "ServerHelloDone"

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

Basic Pattern for Doing Your Project

◆ Read and understand protocol specification
  - Typically an RFC or a research paper
  - We’ll put a few on the website: take a look!
◆ Choose a tool
  - Murph by default, but we’ll describe many other tools
  - Play with Murph 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
  - Actual Murph model available
◆ 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