Formal Security Analysis of Cryptographic Protocol Code

Karthikeyan Bhargavan
INRIA

IIT Delhi, Fall 2010
Lecture 1:
Writing Secure Web Applications

Introduction
Writing Secure Web Applications

• An increasing number of security-critical services can now be accessed from the web
  • Online banking, Google Health, Electronic tax returns

• A parallel increase in attacks on web applications
  • Credit card fraud, Identity theft

• Cryptography, if used correctly, can help
  • Data encryption, Secure hardware, Access control
  • Mathematically proven guarantees

• But easy to get wrong, even for experts
  • Need for training, verification tools
  • Many recent advances in provers, program analysis tools
**Example: Secure Online Banking**

**Secure connection to bank’s website**
Nobody other than the bank can read what I type (confidentiality)

**My secret login information**
Nobody other than me can access my account page (authentication)

**Goal:** Prevent unauthorized access to data even if an unknown attacker controls the network and some other bank clients.
Application-Level Vulnerabilities

- Bank holds records in a Database
  - Is the database secure? And the passwords?
  - Can one bypass the password mechanism?
- Client keeps copies of her bank records
  - Is her computer secure?
  - Can social engineering reveal her password?
Network-level Attacks

- An attacker can hijack the web session
- **Impersonation**: Pretend to be the client
- **Replay**: Resend an old client message
- **Redirection**: Divert a message meant for one server to another
Cryptographic Protections

- Secure password database
- No access to secrets
- Strong crypto primitives
- No side-channel information leakage

Client

- Web Browser (Firefox+JS)
  - Cryptographic Protocol (HTTPS/TLS)
  - Networking (Windows)

Bank

- Web Server (Apache+SQL)
  - Cryptographic Protocol (HTTPS/TLS)
  - Networking (Linux)

Online Banking Procedure

Web Session

Secure Connection

Insecure Network

Networking (Linux)

Networking (Windows)
Security Verification Goal

Given a

- **Web application** with **strong security goals**, protected using a
- **Cryptographic protocols** and **secure databases**, against an
- **Arbitrary attacker** who controls the **network** and **some clients and servers**

**Goal:** Mathematical proof that the web application preserves its security goals in all runs.
Potential Attacks at All Levels

**Client**

- Web Browser (Firefox+JS)
- Cryptographic Protocol (HTTPS/TLS)
- Networking (Windows)

**Bank**

- Web Server (Apache+SQL)
- Cryptographic Protocol (HTTPS/TLS)
- Networking (Linux)

**Social engineering**

- Misconfiguration: 10%

**Web application bugs Browser, Server bugs**

- 40%

**Crypto protocol errors Protocol software bugs**

- 20%

**Operating system bugs**

- 30%

---

* 2010 CWE/SANS Top 25 Most Dangerous Software Errors

---
A Problem of Scale

**Client**
- **Web Browser** (Firefox+JS)
- **Cryptographic Protocol** (HTTPS/TLS)
- **Networking** (Windows)

**Bank**
- **Web Server** (Apache+SQL)
- **Cryptographic Protocol** (HTTPS/TLS)
- **Networking** (Linux)

---

**Insecure Network**

**A**: Web application code 100,000 lines (in Java, Javascript)

**D**: Client and Bank share 100 lines of data

**P**: Security-related code 10,000 lines (in C, Java)

**L**: System libraries 1,000,000 lines (in C)

**O**: Unknown Opponent
A Simpler Verification Goal

Write and verify a
• Simple web application in F#, that uses a
• Cryptographic protocol written in F#, on top of
• Libraries provided by the .NET Framework

Under the following trust assumptions
• *Fully Trusted*: databases, cryptographic algorithms
  – Assume they are implemented correctly
• *Partially Trusted*: operating systems, web browsers
  – Assume they do not leak secret data
• *No Trust*: network, compromised clients and servers
  – Assume they are under the control of the attacker
A Simpler Verification Goal

Client

Web Browser (Firefox+JS)
Cryptographic Protocol (Windows TLS)
Networking (Windows)

Bank

Web Server (Apache+SQL)
Cryptographic Protocol (OpenSSL)
Networking (Linux)

Password-based Authentication
HTTPS Session
TLS Connection

Client and Bank share 100 lines of data

A: Web application code 100,000 lines
P: Security-related code 10,000 lines
L: System libraries 1,000,000 lines
O: Opponent <unknown> lines

100,000 lines of code held by Client
10,000 lines of code held by Bank
1,000,000 lines of code held by Server
<unknown> lines held by Opponent

Insecure Network
If you cannot verify, find an attack

- Protocols are designed by experienced cryptographers
  - And implemented by skilled programmers
- Still, serious flaws both in designs and implementations
  - Most standards got it wrong once or twice (SSL, SSH, IPSEC)
  - Just in 2009, bugs in Google single-sign-on, Microsoft Kerberos, and OpenSSL certificate verification

---

If you cannot verify, find an attack

- Protocols are designed by experienced cryptographers
  - And implemented by skilled programmers
- Still, serious flaws both in designs and implementations
  - Most standards got it wrong once or twice (SSL, SSH, IPSEC)
  - Just in 2009, bugs in Google single-sign-on, Microsoft Kerberos, and OpenSSL certificate verification
Course Outline

• Write web applications in F#
  – Using simple libraries for Networking, Databases, Cryptography, and Key Management
• Security analysis of F# programs
  – Syntax and semantics of F, a core subset of F#
  – Understand and implement attacks
  – Formalize security properties of programs
• Automated proofs of security for F# programs
  – Syntax and semantics of applied pi calculus
  – Translation from F to applied pi
  – Verify F# programs using the ProVerif theorem prover
• Other security verification techniques for programs
Course Keywords

• Functional Programming (F#, lambda calculus)
  – Ease of analysis, well-defined semantics
• Models of concurrency (pi calculus)
  – Precisely model web applications, attacker
• Cryptographic Primitives (AES, HMAC, RSA)
  – Provide strong security guarantees
• Authentication, Secrecy, Authorization
  – Typical security goals
• Formal Verification Tools (Coq, ProVerif)
  – Semi-automated proofs of correctness, security
Lecture 1:
Writing Secure Web Applications

A client-server application in F#
Data Encodings

- Conversions between strings and bytestings
  - bytes are byte arrays, string is Unicode
- Implemented using .NET Framework libraries
  - System.Convert, System.Text.Encoding
A Simple Networking Library

- Network operations in functional style (stateless)
- Sends and receives bytestrings
  - Can be extended to other messaging patterns
- Implemented using .NET Framework (System.Net)
A “Hello, World!” Web Application

```

let rec hellosserver () : unit =
  let c = Net.listen p in
  let r = ("<xml><html>Hello, World!</html>") in
  Net.send c (utf8 r)
```

![Hello, World!](http://127.0.0.1:8080/)

Hello, World!
An Echo Client-Server Application

let echoclient (s:string) : unit =
  let c = Net.connect p in
  Net.send c (utf8 s);
  let r = iutf8 (Net.recv c) in
  print_string r

let echoserver () : unit =
  let c = Net.listen p in
  let s = iutf8 (Net.recv c) in
  let r = ("Received " ^ s) in
  Net.send c (utf8 r)
Running the Echo Client and Server

Terminal — bash — 80x24

dhcp89:src karthik$ mono rpc.exe echoclient TestMessage
Connecting to localhost:8080
Sending: {VGVzdE1lc3NhZ2U=}
containing 11 bytes
Received TestMessage
dhcp89:src karthik$ 

dhcp89:src karthik$ mono rpc.exe echoserver
Listening at 127.0.0.1:8080
Sending: {UmVjZWhZ2Z1cVGVzdE1lc3NhZ2U=}
containing 21 bytes
dhcp89:src karthik$ 

Network-based Attacker

```
module Attack
val send: bytes -> bytes
val recv: bytes -> bytes
```

- We explicitly allow a network-based attacker to modify each message that is sent or received.
- On every message, Net.send and Net.recv call Attack.send and Attack.recv, respectively.
- By default, they do not modify the message.

```
let send (b:bytes) = b
let recv (b:bytes) = b
```
An Injection Attack

- Sent message is modified en route
- Client and server are unaware of modification

```
module Attack
let send (b:bytes) =
  if b = utf8 "TestMessage" then
    utf8 "ModifiedMessage"
  else b
let recv (b:bytes) = b
```
Network-based Attacks

Generally called Man-in-the-Middle attacks

- **Injection**: Send a false message to server
- **Replay**: Send an old message again
- **Redirection**: Forward message to wrong server

Both client and server messages may be targeted, unpredictably over time
Exercises

• Install F# (Windows, or Mono/Linux)
• Write a client-server application of your choice
  – Libraries will be available on the course web page
  – Choose an application with an interesting security property, e.g. something that requires either providing or retrieving secret data
• Exhibit injection, replay, and redirection attacks
  – How do these attacks violate the security property?