PROTOCOL STATE FUZZING OF TLS IMPLEMENTATIONS

Joeri de Ruiter
University of Birmingham
Short introduction to TLS

Client
ClientHello
ClientKeyExchange
ChangeCipherSpec
Finished
ApplicationData

Server
ServerHello
Certificate
ServerHelloDone
ChangeCipherSpec
Finished
ApplicationData
State machines

- Every application that implements a protocol has to implement the corresponding state machine

- **Mealy machines**
  - Set of states
  - Input alphabet
  - Output alphabet

- Specify in all states for each input
  - Returned output
  - Next state

- It is unambiguous
State machine inference

- Extract state machines from implementations by communicating with them
- Fuzzing of message order
- Discover bugs
- Provides interesting insights in the code
- Will not find carefully hidden backdoors
State machine inference
State machine inference

→ ClientHello
← ServerHello
State machine inference

→ ClientHello
← ServerHello
State machine inference

→ ClientHello
← ServerHello

→ Other messages
← Fatal alert / Connection close
State machine inference

→ ClientHello
← ServerHello

→ Other messages
← Fatal alert / Connection close

Other messages
Fatal alert / Connection close

ClientHello
ServerHello
State machine inference

→ ClientHello
← ServerHello

→ Other messages
← Fatal alert / Connection close

→ ClientHello, ClientHello
← Fatal alert / Connection close
State machine inference

→ ClientHello
← ServerHello

→ Other messages
← Fatal alert / Connection close

→ ClientHello, ClientHello
← Fatal alert / Connection close

Other messages
Fatal alert / Connection close
Automated learning

- Deterministic Mealy machine
- Learner
  - Adapted L* algorithm by Niese
- Teacher
  - Equivalence queries approximated
    - Random traces
    - Chow’s W-method

![Diagram showing the interaction between Learner and Teacher](attachment:image.png)
Automated learning

- LearnLib by TU Dortmund
  - Implementation of adapted L* and equivalence algorithms
- Equivalence checking using modified W-method
  - Given an upper bound it is guaranteed to find the correct state machine
  - Depth specified to search for counter-examples
  - After a socket is closed no data will be received
- Custom test harness for TLS
- Manual analysis if we see unexpected behavior
Test harness

- (Almost) stateless TLS implementation
- Minimal state in test harness to handle encryption
- Support to test clients and servers
- All regular TLS messages and Heartbeat extensions
  - RSA and DH key exchange
  - Client authentication
  - Some special symbols that correspond to exceptions in the test harness
Analysis of TLS servers

- 9 TLS implementations
  - OpenSSL
  - GnuTLS
  - Java Secure Socket Extension
  - mbed TLS (previously PolarSSL)
  - NSS
  - RSA BSAFE for C
  - RSA BSAFE for Java
  - miTLS
  - nqsb-TLS

- Every learned model different
Learned models
Results

- Used demo applications when provided
- 6 to 16 states
- 6 minutes to over 8 hours
  - Under 1 hour if connections are properly closed
  - Dependent on implementation specific time-outs (100ms to 1.5s)
- Several new flaws in different implementations
Java Secure Socket Extension

- Possible to skip ChangeCipherSpec message
- Server will accept plaintext data
- Problem also present in client
- Also found by the Prosecco group at INRIA
- Fixed in January 2015
GnuTLS

- Shadow path after sending HeartbeatRequest during handshake
- Buffer handshake messages for hash in Finished reset
- Same problem present in the client
OpenSSL

- Sending a ChangeCipherSpec after successful handshake gets server in invalid state
- Client key set to server key
- Same keys used for both directions
- Fixed in 1.0.1k
- Same issue present in LibreSSL
OpenSSL

- Able to detect EarlyCCS bug by Kikuchi
- By modifying the test harness we can successfully exploit this flaw
nqsb-TLS

- Plaintext alerts returned after ChangeCipherSpec
- No security flaw
- Quickly fixed
- Shows it is a useful technique during development
- Different interpretation of the specification
nqsb-TLS

- Plaintext alerts returned after ChangeCipherSpec
- No security flaw
- Quickly fixed
- Shows it is a useful technique during development
- Different interpretation of the specification
Conclusions

- Protocol state fuzzing is a useful technique to find security flaws and other bugs related to the implementation of state machines.
- Everybody interprets specifications differently and makes different design decisions.
- It would be good to include state machines in specifications.
Conclusions

- Protocol state fuzzing is a useful technique to find security flaws and other bugs related to the implementation of state machines.

- Everybody interprets specifications differently and makes different design decisions.

- It would be good to include state machines in specifications.

Thank you for your attention!