

Verifying the SSH TLP with ProVerif

A Demo

Alfredo Pironti Riccardo Sisto

Politecnico di Torino, Italy
{alfredo.pironti,riccardo.sisto}@polito.it

CryptoForma
Bristol, 7-8 April, 2010

Outline

- 1 Introduction
- 2 The SSH Transport Layer Protocol (TLP)
- 3 Modelling and Verifying the SSH TLP
 - Version One
 - Version Two
- 4 Conclusion

Introduction

Goals

- Show how to model a real protocol in ProVerif
 - So that an interoperable implementation can be derived
- Learn how to tweak the model to let it verify
- Tips to help spotting bugs in the model
 - Safety properties turn true on a protocol doing nothing!

The SSH TLP – I

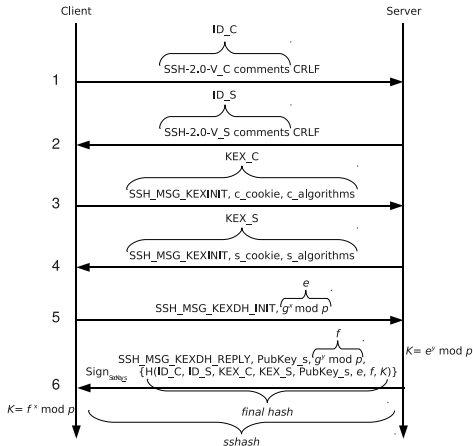
Definition

- RFC 4253
 - Also related to RFC 4250 and 4251

Description

- 1–2: Identifiers exchange
- 3–4: Supported algorithms exchange
- 5: Client's DH public key
- 6: Server's DH public key; digitally signed hash of relevant session data

SSH Transport Layer Protocol



The SSH TLP – II

Algorithms negotiation

- Several algorithms are negotiated
 - E.g. hashing, signing, encryption ...
 - Ten algorithms are negotiated in total
- For each algorithm to be negotiated
 - Each actor sends a list of supported algorithms
 - Preferred algorithms come first
- The most preferred client algorithm that is supported by the server is chosen
 - Negotiation is client-driven

Hashings

- Digital signature implies hashing before signing
 - Hashing is explicitly represented in the model

The SSH TLP – III

Server authentication

- Through public key mechanism
 - But not certificates
- Public key must be trusted by other means
 - Usually by a local key store
 - “First session trust” problem

Specification Outline

Description

- Separate processes for client and server
- One “instance” process coordinating them
 - Thus generating protocol sessions

Example

- See `sshBoth.V1.proverif` file

Notes about the Model

Protocol sessions

- Unbounded independent sessions handled
 - Each actor is replicated; not the composition of the two

Cryptographic algorithms and parameters

- Not taken into account in this version
 - Not even algorithm negotiation

Server trusted public key

- The model “knows” the correct server key
- The server provided key must match the correct one

Experiments with the Model

Tip (valid for all models)

- Do the actors eventually reach the end of the protocol (at least once)?
 - Very relevant to safety properties
 - Use fictitious “debug” events
 - Not liveness, yet some information about protocol execution

Example

- See `sshBoth.V1.debug.proverif` file

Preliminary Algorithms Support – I

Representing algorithms in the model

- Algorithms lists are included in exchanged messages
- No explicit algorithm agreement
- No usage of those algorithms in the specification
- Partially included in authentication (via agreement on exchanged messages)

Example

- See `sshBoth.V2.proverif` file

Preliminary Algorithms Support – II

Enhancing cryptographic primitives

- Algorithms are represented as constructor/destructor parameters
 - The used cryptographic algorithms must match

Representing algorithms in the model

- Use server provided algorithms everywhere
 - Like the server chose the algorithms (based on client preferences)
 - Useful to let the model formally verify
 - Not really close to the RFC

Preliminary Algorithms Support – III

Note

- Always in a Dolev-Yao model
- Adding algorithms to spot logical errors in their negotiation and usage
 - Not able to spot bad algorithms interactions or specific weaknesses

Example

- See `sshBoth.V3.proverif` file

Improving the Model

Addressed issues

- Full algorithms handling
 - Including algorithms negotiation
- Trusted server key handling
 - By means of a fully modeled key store
- Protocol errors handling

Also modeled

- Type casts
 - Unnecessary in untyped languages such as π -calculus and derivatives
 - Needed in strongly typed derived implementations
 - Actually the main source of verification troubles
 - But solved with semantically equivalent constructs

Algorithms Negotiation – I

Strategy

- Both client and server exchange their preferred algorithms lists
 - Remember: several lists; each for one algorithm negotiation
- Each list is opaque (i.e. a name); not an expanded list
- For each algorithm to be negotiated, we have
 - Client list
 - Server list
 - These form a “list-pair”
- Both client and server apply the same negotiation function to each list-pair
 - They get the same negotiated algorithm

Algorithms Negotiation – II

Discussion

- Realistic
 - Each actor gets its own negotiated algorithms independently
 - The negotiated algorithms can be formally shown to be equal
 - Both client and server implement the same negotiation algorithm
- Generic
 - No knowledge on the negotiation algorithm internals
 - Modeled as a hash
 - Output (negotiated algorithm) depends on input (algorithms lists)
 - Link between input and output is lost
 - But the attacker knows the input

Trusted Server Key – I

Strategy

- A key store stores the “known” server key pair
- A honest server accesses the key store to retrieve its key pair
- A honest client compares the server-provided public key with the trusted key looked up from the key store
- One key store running in parallel with each client and each server instance

Trusted Server Key – II

Interacting with the key store

- Request/Response (Output/Input) policy
 - Send request (*OP, DATA*)
 - Receive response *OUTCOME*
- Several available operands
 - *CHECK, CHECK_IP*: check whether the given alias is associated with any key
 - *OUTCOME* is [*YES, NO*]
 - *GET, GET_IP*: retrieve the key associated with the given alias
 - *OUTCOME* is the key, or the process is stuck if there is no such key

Trusted Server Key – III

Discussion

- Design: key store must be implementable
 - We provide such an implementation
 - Backed by the Java key store
- Client does not need to know the “correct” key
 - The key store actually knows it
 - “First session trust” problem still open
- Each session is “independent”
 - Keys in key stores do not depend from previous runs of the protocol
 - Very helpful in verification
 - Reason why “first session trust” problem still open
- *Vision*: Automatically find and exploit protocol session invariants

Protocol Errors

Modeling

- Use “else” branches to handle errors or unexpected data
- Send SSH prescribed error messages
- (Also use a “state” private channel to return data to the caller of the protocol)

Discussion

- More complete specification
- (Dolev-Yao) Information leakage through error messages is ruled out

Type Casts – I

Motivation

- Allow strongly typed implementations be derived from this specification
- Each spi calculus term gets a statically assigned type
- Two types for the same term \rightarrow Two different terms

Design

- Send term M over the *cast* channel: $\text{out}(\text{cast}, M)$
- Read M back from the *cast* channel; bind it to x :
 $\text{in}(\text{cast}, x)$
- Several casts sequentially over the same channel

Type Casts – II

Issue 1

- Over-approximations
 - Message order lost on private channels

Tentative solution

- Rename cast channel to preserve order
- Very verbose specification
- Not always working
 - Not actually working in the current version of the SSH model

Type Casts – III

Issue 2

- According to Blanchet: private channels stress ProVerif a lot
 - We could not reach ProVerif termination

Working solution

- Convert casts to “let” assignments
- Semantically equivalent
 - But avoids duplicated terms (syntactical substitution)
 - Enables verification of the model

Demo

Example

- See `ssh2/*.proverif` files

Conclusion – I

Results

- Quite detailed SSH model
 - Adheres to the RFC
 - Interoperable implementations have been derived from it
 - Include details sometimes omitted from high-level models
 - Algorithms negotiation
 - Key stores
 - Error handling
- Sometimes over-complicated (see casts)
 - Needed to derive an implementation
 - Semantically equivalent simplifications available
- Still in a Dolev-Yao symbolic context

Conclusion – II

Lessons learnt

- Interpretation of a model
 - Usage of cryptography can *mean* something at a higher level
 - E.g. hashing for algorithms negotiation
- Safety properties can be tricky
 - At least ensure some form of “liveness”, especially in complex models
- Tweaking the model may be necessary to enable verification
 - Strongly dependent on verification tool internal algorithms

Future work

- Extend to several (unbounded?) client/server pairs
- Handle the “first session trust” problem

The End

Thank you!

Questions?