Validating Security Protocols Through Type Checking

Eric Allen and Brian Stoler
COMP 527: Computer Systems Security
December 4, 2001

Protocol validation is hard.

♦ We’d like to prove that our protocols are secure.
  – Requires a particular notion of proof and security.

♦ Automated theorem proving (e.g. BAN Logic).
  – May not converge!
  – Can be slow.
  – No explicit model for the set of possible attacks.

♦ Model Checking.
  – Can be slow.

Validation via Type Checking

User annotates protocol with types.

– Types direct the proof of security properties.

– Localized analysis: components can be verified independently.

– Takes time linear in the size of the protocol!

Mathematical Foundations

♦ Pi calculus (Milner, 1999).

♦ Spi calculus (Abadi and Gordon, 1999).

♦ Cryptyc (Gordon and Jeffrey, 2001).
A program has one client process and any number of server processes.

- **client name** { **statement*** }
- **server name socket** { **statement*** }

What are we trying to prove?

- **Protocol is safe.**
  - Protocol works correctly on its own (without an opponent).
- **Protocol is robustly safe.**
  - Protocol works correctly even in the presence of an arbitrary opponent process.
  - The opponent needn’t adhere to our type system!!!
- Still need to formalize “works correctly”.

An Example Process

```java
client Sender {
    connect Receiver socket;
    new msg;
    output socket {msg}key;
}

server Receiver socket {
    input socket ctext;
    decrypt ctext {msg}key;
}
```
An Example Process

```java
client Sender {
    connect Receiver socket;
    new msg;
    output socket {msg}key;
}
server Receiver socket {
    input socket ctext;
    decrypt ctext {msg}key;
}

Is this protocol safe?
Is it robustly safe?
```

Correspondence Assertions

- Annotate protocol with labeled events.

```java
begin message*
end message*
```

- Safety: For every run of the protocol, for every message L, there is a distinct begin L event for every end L event.

- Robust Safety: The protocol is safe even in the presence of arbitrary opponent processes.

Adding Assertions to the Example

```java
client Sender {
    connect Receiver socket;
    new msg;
    begin msg;
    output socket {msg}key;
}
server Receiver socket {
    input socket ctext;
    decrypt ctext {msg}key;
    end msg;
}

Not robustly safe:
An attacker can replay encrypted messages.
```

How can we check robustness?

- Must prevent replay.
  - Use nonces.
  - Nonces allow transfer of knowledge that an event occurred.

- Syntax addition: check nonce is nonce'.

- There should be exactly one check per new nonce statement.
Adding nonce to example

```java
client Sender {
    connect Receiver socket;
    input socket nonce;
    new msg;
    begin msg;
    output socket {msg, nonce}key;
}
server Receiver socket {
    new nonce;
    output socket nonce;
    input socket context;
    decrypt context {msg, nonce'}key;
    check nonce is nonce';
    end msg;
}
```

Types to the Rescue

- Need more annotations to help with proof.
- Need to correlate nonces with effects: events occurring during process execution.
- Nonces are assigned types parameterized by effects.
- A Nonce type specifies the event it validates.

Types to the Rescue

- Added syntax:
  - cast nonce to nonce' : Nonce(effect*)
  - check nonce is nonce' : Nonce(effect*)

Typing Rules

- Each process is examined independently.
- Statements are considered in reverse order, with the effects of each statement accumulated.
- A process type checks if the accumulated effect set is empty.
Effect Rules

- \( \text{end } e \rightarrow \text{add } [\text{end } e] \)
- \( \text{begin } e \rightarrow \text{subtract } [\text{end } e] \)
- \( \text{cast } n \text{ to } n' : \text{Nonce}(e) \rightarrow \text{add } [e] \)
- \( \text{check } n \text{ is } n' : \text{Nonce}(e) \rightarrow \text{subtract } [e] \)
  \[ \text{add } [\text{check } n] \]
- \( \text{new } n \rightarrow \text{subtract } [\text{check } n] \)

Type Checking Sender

```plaintext
client Sender |
connect Receiver socket;
input socket nonce;
new msg;
begin msg;
check nonce is nonce':Nonce(end msg);
output socket {msg, nonce'}key;
}
```

Type Checking Receiver

```plaintext
server Receiver socket |
new nonce;
output socket nonce;
input socket ctext;
derrypt ctext {msg, nonce'}key;
check nonce is nonce':Nonce(end msg);
end msg; }{end msg}
}
```

Implementation

- Gordon and Jeffrey’s implementation: cryptyc.cs.depaul.edu.
- We have reimplemented Cryptyc.
  - Separated parsing and type-checking phases.
  - More sophisticated error reporting.
  - More extensible.
Future Extensions

- Add asymmetric cryptographic primitives.
- Simplify the language.
- Include type inference.
- Enhance error reporting to give examples of how an attacker would exploit type errors.
- Include a protocol compiler/interpreter.

Adding Asymmetric Keys

- In symmetric protocols, data is either
  - Secret and untainted (Most data in Crytyc)
  - Public and tainted (Un)

Adding Asymmetric Protocols

- In asymmetric protocols, public keys allow for addition types:
  - Public and untainted (sent with honest agent’s private key)
  - Secret and tainted (encrypted with honest agent’s public key)
  - Use subtyping to represent these new types

Adding Asymmetric Protocols

- In symmetric protocols, trust is fixed
  - An agent using the symmetric key is either trusted, or is replaying an old message
Adding Asymmetric Protocols

- In asymmetric protocols, trust may increase as new information arises.
- An agent may prove identity by sending back a nonce encrypted with a public key
- Add *trust effects* to account for state of trust during a protocol
- Trust effects added by *trust* statements, removed by *witness* statements

Adding Asymmetric Protocols

- In asymmetric protocols, nonce handshakes may always proceed as follows:
  - Challenge in the clear
  - Response encrypted