Building an arbitrary secure E-commerce system: A Case Study in the Applicability of Spi Calculus to Application Development

Chad R. Meiners
Gokhan Gokoz

December 4, 2002

Overview

- Objectives
- Procedure
- Spi Calculus
- Cryptic
- Issues and Satisfactions
- Conclusions

Objectives

To study the applicability of formal techniques to protocol development

- Defining an arbitrary banking protocol
- Defining a development procedure
- Recording progress and obstacles
- Identify benefits received from the formal technique
Banking Protocol

- The protocol was developed before experience was gathered in the tool support and Spi Calculus extensions.

- The protocol was designed to allow atomic transactions over an public network.

- We wanted the transaction information to remain secret.

- We wanted the transactions to be invulnerable to replay attacks.

Banking Protocol

Message A:
\[
\text{client}(ID) \rightarrow \text{Bank}:
\{(\text{TransactionType}, ID, \text{Nonce}', \text{Challenge})\} K_{ID}
\]
Identify self and request transaction.

Message B:
\[
\text{Bank} \rightarrow \text{client}(ID):
\{(\text{TransactionType}, ID, C\text{Nonce}',
\text{BankChallenge})\} K_{ID}
\]
Request authentication for given transaction.

Message C:
\[
\text{client}(ID) \rightarrow \text{Bank}:
\{(\text{Transaction}, ID, \text{Nonce}', \text{Challenge})\} K_{ID}
\]
Make authenticated transaction.

Message D:
\[
\text{Bank} \rightarrow \text{client}(ID):
\{(\text{Receipt}, ID, C\text{Nonce}'')\} K_{ID}
\]
Give receipt of transaction.

Procedure

Spiral technique with four stages:

- Requirements / Informal Description
- Translate into Spi Calculus
- Translate into Cryptic
- Typecheck

When issues were found we backtracked through our previous stages to keep the artifacts from each stage consistent.
Requirements

- Required secrecy of transaction details
- Required authentication of transactions
- Satisfied secrecy via message encryption
- Satisfied authentication with nonces

Translating into Spi Calculus

\[
\text{Client}(K_{ID}, ID, Socket, Transaction) \triangleq \text{new} \text{Socket} \{\text{BankChallenge}\};
\]

\[
\text{out} \text{Socket} \{(TTyp, ID, BankChallenge)\}_{K_{ID}};
\]

\[
\text{in} \text{Socket} \{(BankAcceptance)\}_{K_{ID}};
\]

\[
\text{decrypt} \text{BankAcceptance is} \{(bTyp, bID, BankChallenge2)\}_{K_{ID}};
\]

\[
\text{new} \text{ClientChallenge};
\]

\[
\text{out} \text{Socket} \{(Transaction, ID, BankChallenge2, ClientChallenge)\}_{K_{ID}};
\]

\[
\text{in} \text{Socket} \{(BankReceipt)\}_{K_{ID}};
\]

\[
\text{decrypt} \text{BankReceipt is} \{(Receipt, ID, ClientChallenge)\}_{K_{ID}};
\]

Translating into Spi Calculus

\[
\text{BankSession}(ID, K_{ID}, Socket) \triangleq \text{new} \text{Socket} \{\text{StartupChallenge}\};
\]

\[
\text{out} \text{Socket} \{\text{StartupChallenge}\};
\]

\[
\text{in} \text{Socket} \{\text{ClientRequest}\};
\]

\[
\text{decrypt} \text{ClientRequest is} \{(TransactionType, ID, StartupChallenge)\}_{K_{ID}};
\]

\[
\text{new} \text{TransactionChallenge};
\]

\[
\text{out} \text{Socket} \{(TransactionType, ID, TransactionChallenge)\}_{K_{ID}};
\]

\[
\text{in} \text{Socket} \{\text{Trans}\};
\]

\[
\text{decrypt} \text{Trans is} \{(Transaction, ID, TransactionChallenge, ClientNonce)\}_{K_{ID}};
\]

\[
\text{new} \text{Receipt};
\]

\[
\text{out} \text{Socket} \{(Receipt, ID, ClientNonce)\}_{K_{ID}};
\]
Correspondence Relations

- **begin** and **end** correspondence assertions were added to simplify authentication proofs.

- All processes are typed according to which correspondence assertions they start and end.

- Processes when combined can change type.
  
  if \( L[\text{begin } X_1 \text{ and } P[\text{end } X_1 } \) then \( L; P[] \)

- A process is **robustly safe** if it is not typed to end a correspondence.

Safe Composition of Processes

The common technique is to use nonces to compose parallel processes safely.

- For each new communication, the receiver sends the sender a unique nonce.

- Upon receiving the nonce, the sender starts the message's correspondence relation and associates the nonce with the correspondence relation.

- The sender can then send the message to the receiver with the nonce.

- The receiver then verifies the nonce before ending the correspondence relation the sender started.

Translating into Cryptyc

```plaintext
client Sender at Customer is {
  /* Message A */
  new (id : MyID);
  new (TType : TransactionType);
  new (trans : Payload);

  establish Receiver at Bank is (Socket : Socket);
  input socket is (nonce : Challenge);

  new (chal1 : Challenge);
  begin (Sender TType);
  cast nonce is
    (nonce' : BankNonce(TType));
  outputSocket is
    ([TType, id, nonce', chal1]SAKey);
```
Translating into Cryptyc

/* Message B */
input socket is
   (\{ tag : TransactionType,
      id2 : MyID,
      nonce' : ClientNonce1(tag),
      nonce2' : Challenge\}SBKey);
check cha1 is nonce';
end (Receiver tag);

/* Message C */
new (cha2 : Challenge);
begin (Sender trans);
cast nonce2 \$ (nonce2' : BankNonce2(trans));
output socket is
   (\{ trans, id, nonce2', cha2\}SCKey);

Translating into Cryptyc

/* Message D */
input socket is
   (\{ Receipt : ReceiptType,
      id3 : MyID,
      nonce2' : ClientNonce2(Receipt)
      \}SDKey);
check cha1 is nonce2';
end (Receiver Receipt);

Translating into Cryptyc

server Receiver at Bank is (socket : Socket) {
   /* Message A */
   new (BankChallenge : Challenge);
   output socket is (BankChallenge);
   input socket is
      (\{ tag : TransactionType,
         id : MyID,
         nonce' : BankNonce1(tag),
         Cnonce1 : Challenge\}SAKey);
   check BankChallenge is nonce';
   end (Sender tag);

   /* Message B */
   new (BankChallenge2 : Challenge);
   begin (Receiver tag);
cast Cnonce1 is
      (Cnonce1' : ClientNonce1(tag));
output socket is
   (\{ tag, id, Cnonce1', BankChallenge2
      \}SBKey);
Translating into CryptoC

/* Message C */
input socket is
  { trans : Payload,
    id2 : MyID,
    nonce2' : BankNonce2(trans),
    Cnonce2 : Challenge } SCKey;
check BankChallenge2 is nonce2';
end (Sender trans);

/* Message D */
new (Receipt : ReceiptType);
begin (Receiver Receipt);
cast Cnonce2 is
  { Cnonce2' : ClientNonce2(Receipt) };
output socket is
  { {Receipt, id2, Cnonce2'} } SCKey;

Issues

- While Spi Calculus can hold state information, we removed state information from the protocol to simplify the write-up
- No support for public key primitives or any sort of arithmetic operators (in CryptoC)
- Lack of formalization of the CryptoC tool.
- Lack of a defined grammar (it exists but it is not defined anywhere except in the implementation code)

Requirement Satisfactions

- Secrecy
  - By typed message inspection we know that no details such as the customer ID or transaction amount is leaked
  - We however have NO guarantee that an attacker cannot perform a statistical attack on the protocol
Conclusions

- No magic involved
  - Benefits are a result of understanding what the tools allow to be checked

- Translation yielded benefits
  - Incremental translation allow us to focus on understanding the role of each part of the protocol

- Method integration deserves more focus
  - Formal security analysis should be perceived as satisfying system requirements.
  - Research into integrating rigorous requirement methodologies (such as Reveal) with formal security method needed.

References

