Code Security Analysis of a Biometric Authentication System Using Automated Theorem Provers

Jan Jürjens

Software & Systems Engineering
Technical University of Munich

juerjens@in.tum.de

http://www4.in.tum.de/~juerjens
Security Analysis of C Programs

Logic-based program understanding of crypto protocols in C which is as
• automatic and
• complete
as possible.

Note: can‘t be both perfectly automated and complete: Security in general undecidable.

Abstract and approximate safely.
Security Analysis

Following Dolev, Yao (1982): To analyze system, verify against attacker model from threat scenarios in deployment diagrams who

- may **participate** in some protocol runs,
- **knows** some data in advance,
- may **intercept** messages on some links,
- **injects** messages that it can produce in some links
- may access certain nodes.
Abstraction, Preprocessing

Enable efficient automated analysis by abstraction (e.g. functions or code-blocks):

- **symbolic** representation of cryptographic or arithmetic routines
- **technical infrastructure** (packet_send, buffer_copy, …)
- **data structures** (e.g. a->b)

Factor out **pointers** usage. Transform to SSA. Eliminate side effects.
Security Analysis in First-order Logic

Approximate set of possible data values flowing through system from above.

Predicate $\text{knows}(E)$ meaning that the adversary may get to know $E$ during the execution of the protocol.

E.g. secrecy: For any secret $s$, check whether can derive $\text{knows}(s)$ using automated theorem prover.
First-order Logic: Basic Rules

Define $knows(E)$ for any $E$ initially known to the adversary.

For evolving knowledge define

\[ \forall E_1, E_2, S. (knows(E_1) \land knows(E_2) \Rightarrow \\
knows(E_1 ::_S E_2) \land knows(\{E_1; S\}_{E_2}) \land \\
knows(Dec_{E_2}(E_1; S)) \land knows(Sign_{E_2}(E_1; S)) \land \\
knows(Ext_{E_2}(E_1; S))) \]

\[ \forall E, S. (knows(E) \Rightarrow \\
knows(head(E; S)) \land knows(tail(E; S))) \]
Control Flow Graph

Generate control flow graph (e.g. with aicall (Absint)).

Transform to state machine:
trans(state, inpattern, condition, action, nextstate)
where action can be outpattern or localvar:=value.
Graph transition

\[ TR1 = (in(msg_{in}), \text{cond}(msg_{in}), out(msg_{out})) \]

followed by \( TR2 \) gives predicate

\[ PRED(TR1) = \forall msg_{in}. [\text{knows}(msg_{in}) \land \text{cond}(msg_{in}) \Rightarrow \text{knows}(msg_{out}) \land PRED(TR2)] \]

Abstraction (e.g. from senders, receivers): find all attacks, may have false positives.

Analyze with automated prover.
Example: Proposed Variant of TLS (SSL)

Presented at IEEE Infocom 1999.

Goal: send secret protected by session key using fewer server resources.
TLS Overview

TLS Client Routine:

- `void TLS_Client(char* secret)`
- Allocate and prepare buffers:
  - `char* Resp_1;`
  - `char* Resp_2;`
- `memset(Resp_1, 0x00, MESSAGEBUFF_MAXLEN);`
- `memset(Resp_2, 0x00, MESSAGEBUFF_MAXLEN);`
- Send `n`:
  - `send(n);`
  - `send(k_c);`
- Send `sign(conc(c, k_c), inv(k_ca)))`:
- Receive Server's respond:
  - `recv(Resp_1);`
- `recv(Resp_2);`
- Check guards:
  - `if (memcmp(fat(ext(Resp_2, k_c)), n, MESSAGEBUFF_MAXLEN) == 0) && (memcmp(snd(ext(dec(Resp_1, k_c)), n, MESSAGEBUFF_MAXLEN) == 0))`
  - Send Secret:
    - `send(symenc(secret, fat(ext(dec(Resp_1, k_c), inv(k_c)), snd(ext(Resp_2, k_c)))));`
  - Free buffers:
    - `free(Resp_1);`
    - `free(Resp_2);`

TLS Server Routine:

- `char* TLS_Server()`
- Receive Init from client:
  - `recv(Init_1);`
- Generate temporary symmetric key:
  - `k_imp = kenc(Init_2);`
  - Send Server's respond:
    - `send(sign(conc(c, k_c), inv(k_ca)))`; `recv(EncSecret);`
  - Receive Secret:
    - `recv(EncSecret);`
  - Get secret:
    - `GetSel = symdec(EncSecret, k_imp);`
  - Free buffers:
    - `free(Init_1);`
    - `free(Init_2);`
    - `free(Init_3);`
    - `free(EncSecret);`
Example: Translation to Logic

\[ \text{knows}(N) \land \text{knows}(K_C) \land \text{knows}(\text{Sign}_{K_C^{-1}}(C::K_C)) \]
\[ \land \forall \text{init}_1, \text{init}_2, \text{init}_3. [\text{knows} (\text{init}_1) \land \text{knows} (\text{init}_2) \land \text{knows} (\text{init}_3) \land \text{snd}(\text{Ext}_{\text{init}_2}(\text{init}_3)) = \text{init}_2 \]
\[ \Rightarrow \text{knows} (\{\text{Sign}_{K_S^{-1}}(\ldots)\} \ldots) \land [\ldots] \land [\ldots \Rightarrow \ldots] \ldots] \]
Surprise …

Can derive *knows(s)*. That is: Protocol does not preserve secrecy of \( s \) against adversaries.

⇒ Completely insecure wrt stated goals.

But why? Use prolog-based attack generator.
Tool Support

ANSL C editor

C code

Annotations

Text Report

data flow

"uses"

seCse Suite

preprocessed annotated code

Control flow graph

Analyzer

Automated Theorem Prover

ATP result

FOL formula

Formula generator
Biometric Authentication System

In development by company in joint project.

Store bio-reference template on smart-card.

Discovered three major attacks against subsequently improved versions (misuse counter circumvented by dropping / replaying messages, smart-card insufficiently authenticated by recombining sessions).
Authent. Protocol Pt. 2: Problem?

Decrease misuse counter

Message order?

\[
\text{sksc} := \text{sessionKey}(Z_1, Z_{2sc})
\]

\[
\text{snd}(\text{arg}_{sc,5,1}) = \text{Mac}_{sksc}(\text{fst}(\text{arg}_{sc,5,1}))
\]

\[
\text{Dec}_{sksc}(\text{fst}(\text{arg}_{sc,5,1})) = \text{"getFBZ2"}
\]

\[
\text{snd}(\text{arg}_{sc,5,1}) = \text{Mac}_{sksc}(\text{fst}(\text{arg}_{h,4,1}))\]

\[
\text{skh} := \text{Dec}_{kh}(\text{fst}(\text{arg}_{h,4,1}))
\]

\[
9: \text{send}(\text{"getFBZ2"}::\{\text{"getFBZ2"}\}_{\text{skh}})
\]

\[
10: \text{return}(\text{FBZ2}::\text{Mac}_{sksc}(\text{FBZ2}))
\]

\[
11: \text{send}(\text{"writeFBZ2"}::\text{FBZ2}'::\text{Mac}_{skh}(\text{FBZ2}'))
\]

\[
14: \text{send}(\{\text{"getData"}\}_{sksc})
\]

\[
15: \text{return}(\{\text{tuser}::\text{Sign}_{inv(ka)}(\text{Hash(idsc::tuser)})\}_{sksc})
\]

\[
\text{Drop message 11} \ldots
\]
Authent. Protocol Pt. 2: Improvement

Check whether FBZ decreased
Authent. Prot. Pt. 2: Improvement?

Note:
\[ \text{skh}=\text{sksc} \]
\[ \text{FBZ2}=\text{FBZ2}' \]
Authent. Prot. Pt. 2: Problem

Replay MAC_{skh} (FBZ2')

\[ \text{snd}(\text{arg}_{sc,5,1}) = \text{mac}_{sksc}(\text{fst}(\text{arg}_{sc,5,1})) \]
\[ \text{Dec}_{sksc}(\text{fst}(\text{arg}_{sc,5,1})) = \text{"getFBZ2"} \]
\[ \text{thd}(\text{arg}_{sc,6,1}) = \text{mac}_{sksc}(\text{snd}(\text{arg}_{sc,6,1})) \]
\[ \text{fst}(\text{arg}_{sc,6,1}) = \text{"writeFBZ2"} \]
\[ \text{FBZ2} := \text{fst}(\text{arg}_{sc,5,1}) \]
\[ \text{snd}(\text{arg}_{sc,7,1}) = \text{mac}_{sksc}(\text{fst}(\text{arg}_{sc,7,1})) \]
\[ \text{Dec}_{sksc}(\text{fst}(\text{arg}_{sc,7,1})) = \text{"getFBZ2"} \]
\[ \text{Dec}_{sksc}(\text{arg}_{sc,8,1}) = \text{"getData"} \]

10: return(FBZ2:: Mac_{sksc}(FBZ2))

11: send("writeFBZ2": FBZ2:: Mac_{skh}(FBZ2'))

12: send("getFBZ2": ["getFBZ2"]_{skh})

13: return(FBZ2:: Mac_{sksc}(FBZ2))

14: send("getData"}_{sksc}

15: return([tuser::Sign_{inv[ka]}(Hash(idsc::tuser)_{sksc})

\[ \text{mh} := \text{Dec}_{skh}(\text{arg}_{h,7,1}) \]
\[ t'\text{user} := \text{fst}(\text{mh}) \]
\[ \text{Ext}_{ka}(\text{snd}(\text{r})) \]
Authent. Prot. Pt. 2: Improvement (?)

Subst. $\text{MAC}_{\text{skh}}(\text{FBZ2}')$ by $\text{MAC}_{\text{skh}}$ ("write"::FBZ2')
Authentic Protocol Part 1: Problem ?

Mutual authentication with challenge & response

Authentic. vs. key gen. ?

Generate shared key

$$\text{sksc} := \text{sessionKey}(Z_h, Z_{2sc})$$
Mutual authentication with challenge & response

Forged smart-card after authentic.; replay old session key

Generate shared key

Authentic. Protocol
Part 1: Problem.
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Authentic Protocol
Part 1: Improvement (!)

Mutual authentication with challenge & response
Use (both) random numbers in Macs
Generate shared key

```

```
Conclusions

Understanding Security Goals using First-Order-Logic:

• formally based approach
• automated, powerful tool support
• successful use in industrial projects

Further work: assertions.

More information:

http://www4.in.tum.de/~juerjens