Understanding Security Goals Provided by Cryptoprotocol Implementations

Jan Jürjens

Software & Systems Engineering
Technical University of Munich

juerjens@in.tum.de

http://www4.in.tum.de/~juerjens
Software Engineering & Security

„Penetrate-and-patch“ (aka „banana strategy“):

- insecure
- disruptive

Traditional formal methods: limited adoption in industry.

- training people
- constructing formal specifications.
Model-based Security

Increase security with bounded investment in time, costs:

- Extract models from artefacts arising in industrial development and use of security-critical systems (UML models, source code, configuration data).
- Tool-supported, theoretically sound, efficient automated security analysis.
Model-based Security Engineering

- Analyze (UMLsec) models against security requirements.
- Generate code (or tests) from models.
- Generate models from evolving or legacy code.

Goal: model-based = source-code based.
Security Analysis of C-Programs

Goal: Logic-based security analysis of C programs which is as
• automatic and
• complete
as possible.

Note: can’t be both perfectly automated and complete: Security in general undecidable.

Here: emphasize automation.
Abstraction

Enable efficient automated analysis by abstraction.

Use lookup tables for 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.
Control Flow Graph

Use aicall (Absint, SB) to generate Mealy Machines from code:

\[ \text{trans(state, inpattern, condition, action, nextstate)} \]

where action can be outpattern or localvar:=value.

Loops: Abstract away or break up after maximal iteration number.
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.
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.

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

Define \( \text{knows}(E) \) for any \( E \) initially known to the adversary.

Define cryptosystem. E.g.: \( \text{Dec}_{K^{-1}}(\{E\}_k) = E \)

For evolving knowledge define

\[
\forall E_1, E_2. (\text{knows}(E_1) \land \text{knows}(E_2) \Rightarrow \\
\text{knows}(E_1 :: E_2) \land \text{knows}(\{E_1\}_{E_2}) \land \\
\text{knows}(\text{Dec}_{E_2}(E_1)) \land \text{knows}(\text{Sign}_{E_2}(E_1)) \land \\
\text{knows}(\text{Ext}_{E_2}(E_1)))
\]

\[
\forall E. (\text{knows}(E) \Rightarrow \\
\text{knows}(\text{head}(E) \land \text{knows}(\text{tail}(E)))
\]
Given Control Flow Diagram ...

```
routine: s_xchd_3_message

void s_xchd_3_message(SOCKET stream)

if(strcmp(head (ext (s_res_1_2_arg,"pubkey_ca")),"S") == 0)
    if(strcmp(head (tail (ext (dec (s_res_1_1_arg,"privkey_c"),head
tail (ext (s_res_1_2_arg,"pubkey_ca"))))),"N") == 0)
        char* local_xchd_2_1= enc ("secret","k");
        SendMessage(stream, local_xchd_2_1);

exit
```
... Translate to 1st Order Logic

Graph transition

\[ TR1 = (in(msg_in), cond(msg_in), out(msg_out)) \]

followed by \( TR2 \) gives predicate

\[ PRED(TR1) = \forall msg_in. \left[ \text{knows}(msg_in) \land \neg \text{cond}(msg_in) \right] \]

\[ \Rightarrow \text{knows}(msg_out) \land \neg PRED(TR2) \]

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

Check whether can derive threat conjecture (e.g. \( \text{knows}(s) \) for a secret \( s \)) from axioms.
Example: Proposed Variant of TLS (SSL)

Presented at IEEE Infocom 1999.

Goal: send secret protected by session key using fewer server resources.
Example: Translation to Logic

\[ \text{knows}(N) \models \text{knows}(K_C) \models \text{knows}(\text{Sign}_{K_{C^{-1}}}(C::K_C)) \]

\[ \forall \text{init}_1, \text{init}_2, \text{init}_3. [\text{knows}(\text{init}_1) \models \text{knows}(\text{init}_2) \models \text{knows}(\text{init}_3) \models \text{snd}(\text{Ext}_{\text{init}_2}(\text{init}_3)) = \text{init}_2] \]

\[ \Rightarrow \text{knows}(\{\text{Sign}_{K_{S^{-1}}}(\ldots)\} \models \ldots \models \ldots \Rightarrow \ldots \ldots) \]
Can derive $\text{knows}(s)$. That is: Protocol does not preserve secrecy of $s$ against adversaries.

$\Rightarrow$ Completely insecure wrt stated goals.

But why? Use prolog-based attack generator.
Man-in-the-Middle Attack

\[
\begin{align*}
C & \xrightarrow{N_i::K_C::\text{Sign}_{K_C^{-1}}(C::K_C)} A & N_i::K_A::\text{Sign}_{K_A^{-1}}(C::K_A) \\
& & \xrightarrow{\{\text{Sign}_{K_S^{-1}}(K_j::N_i)\} K_A::\text{Sign}_{K_{CA}^{-1}}(S::K_S)} S \\
A & \xleftarrow{\{\text{Sign}_{K_S^{-1}}(K_j::N_i)\} K_C::\text{Sign}_{K_{CA}^{-1}}(S::K_S)} C \\
& & \xleftarrow{\{s\}_K_j} A \xrightarrow{\{s\}_K_j} S
\end{align*}
\]
Biometric Authentication System

In development by large German company.

In joint project, use presented security analysis tools at given UML specification.

So far, have discovered three major attacks against subsequently improved versions (misuse counter circumvented by dropping / replaying messages, smart-card insufficiently authenticated by recombing sessions).
Conclusions

Understanding Security Goals using First-Order-Logic:

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

More information:
http://www4.in.tum.de/~juerjens
Cryptographic Expressions I

\(\text{Exp} \): quotient of term algebra generated from sets \(\text{Data} \), \(\text{Keys} \), \(\text{Var} \) of symbols using

- \(\_::\_\) (concatenation), \(\text{head}(\_) \), \(\text{tail}(\_)\),
- \(\_^{-1}\) (inverse keys)
- \(\{_\_\}_\) (encryption)
- \(\text{Dec}(\_)\) (decryption)
- \(\text{Sign}(\_)\) (signing)
- \(\text{Ext}(\_)\) (extracting from signature)

under equations …
Cryptographic Expressions II

∀∀E,K.\text{Dec}_K^{-1}(\{E\}_K)=E
∀∀E,K.\text{Ext}_K(\text{Sign}_K^{-1}(E))=E
∀∀E_1,E_2.\text{head}(E_1::E_2)=E_1
∀∀E_1,E_2.\text{tail}(E_1::E_2)=E_2

• Associativity for ::.
Write $E_1::E_2::E_3$ for $E_1::(E_2::E_3)$ and $\mathrm{fst}(E_1::E_2)$
for $\text{head}(E_1::E_2)$ etc.

Can include further crypto-specific primitives
and laws (XOR, ⋯).
Resources

TUM TB Dez. 2004 (mit T. Kuhn)
International Conference for Software Maintenance 2005
Memocode 2005
Models (UML) 2005 (mit S. Houmb)

More information (papers, slides, tool, …):
http://www.umlsec.org

Note: International Symposium on Secure Software Engineering (ISSSE 06 - IEEE)
input_formula(tls_abstract_protocol, axiom,( ![ArgS_11, ArgS_12, ArgS_13, ArgC_11, ArgC_12] :
    ![DataC_KK, DataC_k, DataC_n] :
        % Client -> Attacker (1. message)
        ( knows(n) & knows(k_c) & knows(sign(conc(c, k_c), inv(k_c) )) ) ) &
        % Server -> Attacker (2. message)
        ( ( knows(ArgS_11) & knows(ArgS_12) & knows(ArgS_13) & (? [X] : equal( sign(conc(X, ArgS_12), inv(ArgS_12) ), ArgS_13 ) ) )
    => ( knows(enc(sign(conc(kgen(ArgS_12), ArgS_11), inv(k_s) ), ArgS_12 ) ) & knows(sign(conc(s, k_s), inv(k_ca) )) ) ) )
The Fix

e-Setheo: \textit{knows}(s) not derivable. Thus secure.
Industrial Application

Verisoft project. Goal: Practical application of formal methods.

Planned for 8 years from 7/2003; 10 industrial + academic partners.

Integrated formal verification from application software down to operating system and processor on C-code level. Security-relevant:

• Biometric access control system
• Automotive emergency application

Apply UMLsec approach, tools.
& % Client -> Attacker (3. message)
  (  (  knows(ArgC_11)
       & knows(ArgC_12)
       & equal(sign(conc(s, DataC_KK), inv(k_ca)), ArgC_12 )
       & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC_KK) ),
                  k_c), ArgC_11 )
     & (  ? [DataC_ks] : equal(sign(conc(s, DataC_ks), inv(k_ca) ),
                                 ArgC_12 ) )
     & equal(enc(sign(conc(DataC_k, n), inv(DataC_KK) ), k_c),
            ArgC_11 )
     & equal(enc(sign(conc(DataC_k, DataC_n), inv(DataC_KK) ), k_c),
            ArgC_11 )
   )
=> (  knows(symenc(secret, DataC_k))  )
)) ).
Adversary: Simulation

**Adversary knowledge:**

- $k^{-1}, y, x$
- $\{z\}_k, z$

- $\forall e, k. Dec_{k^{-1}}(\{e\}_k) = e$
... and Physical Layer Model ...

Deployment diagram.
Derived adversary model: read, delete, insert data.
Execute in System Context

Activity diagram.
Formulate Data Security Requirements

Class diagram.
Gives conjecture: \( \text{knows}(s) \) derivable?
Application: OpenSSH

Apply to OpenSSH handshake.

Can find security flaws from earlier version.

```c
/* #ATP# state(1121, ssh_client_key_with_diffie_hellman) */
/* #ATP# set(1121,1122,g_c,generator(c),-) */
/* #ATP# set(1122,1123,p_c,getmodulus(c),-) */
/* #ATP# set(1123,1124,e_c,dh(g_c,x_c),-) */
/* #ATP# io(1124,1125,-,e_c,-) */
/* #ATP# io(1125,1126,ks,-,-) */
/* #ATP# set(1126,1127,k_c,key(f_c,x_c),-) */
/* #ATP# cond(1127,verify_host_key(k_c)==true,1128,0,-) */
/* #ATP# io(1128,1129,f,-,-) */
/* #ATP# io(1129,1130,sig,-,-) */
/* #ATP# set(1130,1131,k,dhcomputekey(f),-) */
/* #ATP# set(1131,1132,hh_c,hash(conc(VERC_c,conc(VERC_c,conc(KINITC_c,conc(KINITS_c,conc(f_c,conc(e_c,k_c))))))))),-) */
/* #ATP# cond(1132,verify(sig,inv(ks),hh_c)==true,1133,0,-) */
```
UMLsec: Goals

Extension for secure systems development.
• evaluate UML specifications for weaknesses in design
• encapsulate established rules of prudent secure engineering as checklist
• make available to developers not specialized in secure systems
• consider security requirements from early design phases, in system context
• make certification cost-effective
UMLsec: How

Recurring security requirements, adversary scenarios, concepts offered as stereotypes with tags on component-level. Use associated constraints to verify specifications using automated theorem provers and indicate possible weaknesses. Ensures that UML specification provides desired level of security requirements. Link to code via round-trip engineering etc.
Formal semantics for UML: How

Diagrams in context (using subsystems).
Model actions and internal activities explicitly.
Message exchange between objects or components (incl. event dispatching).
For UMLsec: include adversary arising from threat scenario in deployment diagram.
Use Abstract State Machines (pseudo-code).
Adversaries

Model classes of adversaries.

May attack different parts of the system according to threat scenarios.

Example: insider attacker may intercept communication links in LAN.

To evaluate security of specification, verify against adversary model.
Send reference template and signature to host system

Send reference template and signature to host system.
Crypto Protocol Part 2: Problem

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

\[
\begin{align*}
\text{snd}(& \text{arg}_{sc,5,1}) = \text{Mac}_{sksc} (\text{fst}(& \text{arg}_{sc,5,1})) \\
\text{Dec}_{sksc} (\text{fst}(& \text{arg}_{sc,5,1})) &= \text{"getFBZ2"}
\end{align*}
\]

\[
\begin{align*}
\text{thd}(& \text{arg}_{sc,6,1}) = \text{Mac}_{sksc} (\text{snd}(& \text{arg}_{sc,6,1})) \\
\text{fst}(& \text{arg}_{sc,6,1}) &= \text{"writeFBZ2"}
\end{align*}
\]

\[
\text{FBZ2} := \text{fst}(\text{argsc}, 5, 1)
\]

\[
\begin{align*}
\text{Dec}_{sksc} (\text{arg}_{sc,8,1}) &= \text{"getData"}
\end{align*}
\]

\[
\begin{align*}
\text{9:send(&gt;FBZ2&quo;,:)&gt;\text{"getFBZ2"})\text{skh} &gt; \\
\text{10:return(FBZ2::Mac}_{sksc} (FBZ2))\text{skh} &gt; \\
\text{11:send(&gt;writeFBZ2&quo;::FBZ2::Mac}_{skh} (FBZ2))\text{skh} &gt; \\
\text{14:send(&gt;getData")sksc} &gt; \\
\text{15:return({user::Sign}_{inv}ka)(Hash(idsc::user)}) &gt; \\
\text{mh} := \text{Dec}_{skh} (\text{arg}_{h,7,1}) &gt; \\
t'\text{user} := \text{fst}(\text{mh}) &gt; \\
[\text{Hash}(&\text{arg}_{h,1,1} : t'\text{user}) = \text{Ext}_{ka} (\text{snd}(&\text{mh}))]
\end{align*}
\]
Crypto Protocol Part 2: Improvement

10: return(FBZ2:: MacSKSC(FBZ2))

11: send("writeFBZ2::FBZ2:: Macskh(FBZ2')")

12: send("getFBZ2*::*("getFBZ2**") skkh")

13: return(FBZ2:: MacSKSC(FBZ2))

14: send("getData") sksc

15: return({tuser::Signinv(ka)(Hash(idsc::tuser))} sksc)

Check whether FBZ decreased

Fred::{MacKH(fst(arg h,5,1))}
[fst(arg h,5,1) > 0] FBZ2' := fst(arg h,5,1) - 1
Crypto Protocol Part 2: Improvement?

Note:

$skh = sksc$

$FBZ2 = FBZ2'$
Crypto Protocol Part 2: Problem

Replay MAC_{skh} (FBZ2')
CEPS: Load Protocol

Load «data security»

Card
  «critical»
  \{ secrecy=\{K_{CI}\}\}
  \{ integrity=\{K_{CI},cep,nt,rc_{nt}\}\}
cep,nt,rc_{nt}: Data; K_{CI}: Keys

Init(lda,m)
Credit(s2,r1)

LSAM
  «critical»
  \{ secrecy=\{K_{LI}\}\}
  \{ integrity=\{K_{LI},lda,n,r_l,n,r_2l,n,m_n\}\}
lda,n,r_l,n,r_2l,n,m_n: Data
K_{LI},r_n: Keys
RespL(cep,nt,s1,hc)
RespC(s3,rc)
RespL(s2)

Issuer
  «critical»
  \{ secrecy=\{K_{CI},K_{LI},rc_{nt}\}\}
  \{ integrity=\{K_{CI},K_{LI},rc_{nt}\}\}
rc_{nt}: Data; K_{LI},K_{CI}: Keys

Load(cep,lda,m,nt,s1,ml,h
  hl,h2l)
Comp(cep,lda,m,nt,r_2l,s3)
CEPS Load and Purchase Protocol (three weaknesses)

\( m_l_n : \) „Proof“ for bank that loading machine received money
But: \( r_n \) shared between bank, loading machine
Theorem: Corrected version secure.
CSDUML Framework

Framework for analysis plug-ins to access UML models on conceptual level over various UI’s. Exposes a set of commands. Has internal state (preserved between command calls). Framework and analysis tools accessible and available at http://www4.in.tum.de/~umlsec. Upload UML model (as .xmi file) on website. Analyse model for included critical requirements. Download report and UML model with highlighted weaknesses.
Beyond Specification Analysis

Model-based test generation.
Configuration analysis.

• Analyze permission data using Prolog (e.g. SAP R/3). Integrated in CSDUML framework.
• Analyze firewall configurations using model-checkers

Source-code analysis (C).
\begin{align*}
\text{enc}_S(E, E') & \quad \text{(encryption)} \\
\text{dec}_S(E, E') & \quad \text{(decryption)} \\
\text{hash}_S(E) & \quad \text{(hashing)} \\
\text{sign}_S(E, E') & \quad \text{(signing)} \\
\text{ver}_S(E, E', E'') & \quad \text{(verification of signature)} \\
\text{kgen}_S(E) & \quad \text{(key generation)} \\
\text{inv}_S(E) & \quad \text{(inverse key)} \\
\text{conc}_S(E, E') & \quad \text{(concatenation)} \\
\text{head}_S(E) \text{ and } \text{tail}_S(E) & \quad \text{(head and tail of concat.)}
\end{align*}
/* C_Encrypt encrypts single-part data. */
CK_PKCS11_FUNCTION_INFO(C_Encrypt)
#ifdef CK_NEED_ARG_LIST
(
    CK_SESSION_HANDLE hSession,        /*session handle*/
    CK_BYTE_PTR pData,                 /*plaintext data*/
    CK ULONG ulDataLen,                /*plaintext bytes*/
    CK BYTE_PTR pEncryptedData,       /*gets ciphertext*/
    CK ULONG_PTR pulEncryptedDataLen   /*gets ctext size*/
);
#endif
input_formula(protocol, axiom, 

! [ S, Init_1, Init_2, Init_3, Resp_1, Resp_2, Xchd_1 ] : ( 

% C -> Attacker
( ( true & true )
  => knows( conc(S, n, conc(S, k_c, sign(S, conc(S, c, conc(S, k_c, eol)), inv(S, k_c))) ) ))

& ( knows(Resp_1) & knows(Resp_2)
  & equal( fst(S, ext(S, Resp_2, k_ca)), s) & equal( snd(S, ext(S, dec(S, Resp_1, inv(S, k_c)),
    snd(S, ext(S, Resp_2, k_ca)))), n ) )
  => knows( enc(S, secret, fst(S, ext(S, dec(S, Resp_1, inv(S, k_c)), snd(S, ext(S, Resp_2, k_ca)))))))
)

) & % S -> Attacker
( ( knows(Init_1) & knows(Init_2) & knows(Init_3)
    & equal( snd(S, ext(S, Init_3, Init_2)), Init_2 ) )
  => knows( conc(S, enc(S, sign(S, conc(S, kgen(S, Init_2), conc(S, Init_1, eol)), inv(S, k_s)), Init_2),
    sign(S, conc(S, s, conc(S, k_s, eol)), inv(S, k_ca))) )

& ( ( knows(Xchd_1)
    & true )
  => true ) ) ) ) ) .
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_GetAttributeValue</td>
<td>obtains attribute value</td>
</tr>
<tr>
<td>C_SetAttributeValue</td>
<td>modifies attribute value</td>
</tr>
<tr>
<td>C_Encrypt</td>
<td>encrypts single-part data</td>
</tr>
<tr>
<td>C_Decrypt</td>
<td>decrypts encrypted data</td>
</tr>
<tr>
<td>C_Digest</td>
<td>digests single-part data</td>
</tr>
<tr>
<td>C_Sign</td>
<td>signs single-part data</td>
</tr>
<tr>
<td>C_VerifyRecover</td>
<td>verifies a signature</td>
</tr>
<tr>
<td></td>
<td>where the data is recovered</td>
</tr>
<tr>
<td>C_GenerateKey</td>
<td>generates a secret key</td>
</tr>
<tr>
<td>C_GenerateKeyPair</td>
<td>generates a key pair</td>
</tr>
<tr>
<td>C_GenerateRandom</td>
<td>generates random data</td>
</tr>
</tbody>
</table>
(STEP 1) Generation of function call graph from code p_i

(STEP 2) Marking of relevant functions with object relation numbers j

(STEP 3) Read all p_i and abstraction code (Library Tables).

(STEP 4) Security Property Definition (e.g. for confidentiality): Marking of data structures within the library table.

(STEP 5) Generation of n-separation from marked p_i.

(STEP 6) For each part j of the n-separation and all programs:
- Transform source code to minimal code
- Transform minimal code to ATP marking language.

(STEP 7) Transform to logical representation for each part j of the n-separation

(STEP 8) Transform data structures with marked security tag.

Verification with ATPs e.g. E-Setheo, SPASS, ...

Logical .tptp File (Representation)
Annotated Code

\begin{align*}
&\text{n: } \text{set}(n,n+1,a,b,-) \\
&\text{n+1: } \text{...} \\
&\text{n: } \text{set}(n,n+1,a(i),b,-) \\
&\text{n+1: } \text{...} \\
&\text{n: } \text{<atpmarking of line 1>} \\
&\text{n+1: } \text{<atpmarking of line 2>} \\
&\text{n: } \text{cond}(n,a==b,i,j,-) \\
&i: \text{...} \\
&j: \text{...} \\
&\text{n: } \text{cond}(n,a==b,i,n+1,-) \\
&i: \text{...} \\
&n+1: \text{...} \\
&\text{n: } \text{set}(n,n+1,local,a,-) \\
&n+1: \text{set}(n+1,n+2,localb,b,-) \\
&\text{...} \\
&n+26: \text{set}(n+26,i,localz,z,-) \\
&z: \text{<atpmarking of function start>} \\
&\text{a = b} \\
&a[i] = b \\
\{ <\text{line1}>; <\text{line2}>; \ldots \} \\
\text{if (a==b) then } \{ \text{...} \} \text{ else } \{ \text{...} \} \\
\text{while (a==b) do } \{ \text{...} \} \\
\text{f(a, b, ..., z);}
\end{align*}
void simplecall(char *shared_secret, char *kbuf, int klen, char *secret)
/* #ATP# state(1,2) */
{
    char *signature;
    int slen;

    /* read input */
in_init();
in_read(server_host_key);
/* #ATP# io(2,3,serverhostkey,-,-) */
in_read(signature);
/* #ATP# io(3,4,sig),-,--) */
in_read(slen);

    /* compute hash of server_host_key and length with shared secret */
tmp_hash = simple_crypto_hash(
    server_host_key,   /* server public key (ks) */
    shared_secret     /* shared secret for the crypto hash (k) */
/* #ATP# set(4,5,tmphash,hash(conc(serverhostkey,sharedsecret)),-,--) */

    /* check signature where hash is the signed value */
if (key_verify(server_host_key, signature, slen, tmp_hash, 20) == false)
fatal("key_verify failed for server_host_key");
/* #ATP# cond(5,verify(sig,serverhostkey,tmphash)==false,0,6,--) */

    /* send out data 'secret' encrypted with server_host_key */
out_init();
out_write(DES_encryption(secret,server_host_key));
/* #ATP# io(6,7,-,enc(secretdata,serverhostkey),-) */

out_close();
in_close();
xfree(kbuf);
}
Translation to Logic

Similar as explained earlier.
Loops: Abstract away or break up after maximal iteration number.

Example:

\[(true \land true \Rightarrow true \ % \ state \ 1 \\
   \land (knows(serverhostkey) \land true \Rightarrow true \ % \ state \ 2 \\
   \land (knows(sig) \land true \Rightarrow true \ % \ state \ 3 \\
   \land (true \land \text{tmphash} = \text{hash(conc(serverhostkey,sharedsecret))} \Rightarrow true \ % \ state \ 4 \\
   \land (true \land \neg(\text{verify(sig,serverhostkey,tmphash)}=false) \Rightarrow true \ % \ state \ 5 \\
   \land (true \land true \Rightarrow knows(enc(secretdata,serverhostkey))))))) \ % \ state \ 6\]