Quality Assurance for Security-Critical Systems
Jan Jürjens

Computing Department
The Open University

http://www.umlsec.org

J.Jurjens@open.ac.uk
3 Key Points

- Quality Assurance for security-critical systems is very difficult
- Some specific security properties (e.g. wrt brute-force attacks) cannot be established using conventional testing methods
- Model-based Security Testing is a corner-stone of Security QA
Personal Introduction

- Senior Lecturer (comp. US Assoc. Prof.) at The Open University (British distance university)
- Prev. leading Competence Center for IT-Security at Software & Systems Engineering, TU Munich
- Extensive collaboration with industry (BMW, HypoVereinsbank, Munich Re, O2, Deutsche Bank, Siemens, Infineon, Allianz, …)
- PhD in Computer Science from Oxford Univ., Masters in Mathematics from Bremen Univ.
- Numerous publications incl. 2 books on the subject
- Founding Chair of GI WG on Formal Methods & Software Engineering for Safety & Security (FoMSESS)
- Extensive consulting on security and quality assurance
Testing Security-Critical Systems

Very challenging.

Given motivated adversary, would need full coverage (test every possible execution).

Usually infeasible (especially open systems).

Need heuristics for trade-off between development effort and reliability.

Need to ask yourself:

- How complete is the heuristic?
- How can I validate it?
Security: Some Causes for Problems

„Blind“ use of security mechanisms:

- Security usually compromised by circumventing (rather than breaking) them.
- Assumptions on system context, physical environment.
- Attacker may use unintended/unnoticed functionality

„Those who think that their problem can be solved by simply applying cryptography don`t understand cryptography and don`t understand their problem“ (R. Needham).
Security: Need to Consider Complete Picture
Basic Security Requirements

Secrecy

Integrity

Availability

Secrecy

Information

Integrity

Sender

Availability

Nonrepudiability

Sender
Problem: Security is Elusive

- Classical weakness in old Unix systems: “wrong password” message at first wrong letter in password. Using **timing attack**, reduce password space from $26^n$ to $26^n$ (n = password length)

- More recent weakness on smart-card: reconstruct secret key by timed measurement of power consumption during crypto operations

→ How do you find these weaknesses using classical testing? (You don’t.)
Problem: Untrustworthy Programmer

- For security assurance, may not even trust the programmer of the code.
- May have intentionally built in back-door into code.
- May be impossible to find by random or black-box testing (e.g. hard-coded special password).
- Even worse when elusive weaknesses are used (previous slide).

What is your precaution?

(Probably none.)
Special Problem: Crypto

- Cryptography plays important role in many security-critical applications
- By definition, needs to be secure against brute-force attacks
- **Paradox**: How do you get sufficient test coverage (for inputs accessible to a given attacker) of a system that needs to be secure against brute-force attacks on that input?

→ What's your answer? (Not using classical testing.)
Model-based Security Assurance

Idea: Extract models from artefacts in development and use of software.

Requirements

- Weave
- Analyze against

UML Models

- Code-/Testgen.
- Reverse Engin.

Source Code

Configurations

- Verify
- Gener.
- Configure

Idea: Extract models from artefacts in development and use of software.
Secure System Lifecycle

Critical requirements

External review

Static analysis (tools)

System Monitoring

Abuse cases

Risk analysis

Risk-based tests

Risk analysis

System breaks

Requirements and use cases

Design

Test plans

Code

Test results

Field feedback

Model-based Security Assurance

[McGraw 2003]
Security Assurance: Model vs Code

Model:

+ earlier (less expensive to fix flaws)
+ more abstract ➔ more efficient

➔ Analyze both model and code:
  - construct interface spec model
  - analyze interface spec for critical requirements
  - verify that software satisfies interface spec
**UMLsec**

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 Tool Support

For example:

- consistency checks
- automated analysis of complicated requirements on model level (bindings to model-checkers, constraint solvers, automated theorem provers, ...)
- code generation
- test-sequence generation
- configuration data analysis against models.
Tool Architecture

Snapshot

Jan Jürjens, OU


Model Verification

Jessie – using RSA & Server authentication

C:Client
ClientHello(R_{i,3})
ServerHello(R_{i,3})
Certificate(sig_{K_{i}}(G_{i}, R_{i,3}))
ClientKeyExchange(enc_{K_{i}}(prf_{i}))
Send(K_{i}, enc_{K_{i}}(G_{i}, R_{i,3}))
Send(K_{i}, enc_{K_{i}}(G_{i}, R_{i,3}))
HandshakeDone

S:Server

R_{i,3} \equiv \text{msg3,1}
K_{i} \equiv \text{msg1,1}
K_{i} \equiv \text{msg3,1} \cdot (G_{i}, R_{i,3})
K_{i} \equiv \text{msg1,1} \cdot (G_{i}, R_{i,3})
K_{i} \equiv \text{msg3,1} \cdot (G_{i}, R_{i,3})
K_{i} \equiv \text{msg1,1} \cdot (G_{i}, R_{i,3})
K_{i} \equiv \text{msg3,1} \cdot (G_{i}, R_{i,3})

\text{FOL}

\text{knows} (\text{ArgC}_3)
\& \text{equal} (\text{fst} (\text{ArgC}_3), \text{type_serverkeyexchange})
\& \text{equal} (\text{snd} (\text{ext} (\text{snd} (\text{snd} (\text{ArgC}_3), k_{\text{ca}}), skey))
\& \text{equal} (\text{snd} (\text{ext} (\text{snd} (\text{ArgC}_2), k_{\text{ca}}), \text{fst} (\text{snd} (\text{ArgC}_3))))

=>
\{\text{knows} (\text{ArgC}_4, k_{\text{ca}})
\& \text{equal} (\text{ArgC}_4, 1, \text{type_serverhelldone})
\}

=>
\{ \text{true} \& \text{equal} (\text{ClientKeyExchange}, \text{enc} (\text{premasterkey}, skey)) \}

%--------------------------------- Conjecture --

\text{input\_formula} (\text{attack}, \text{conjecture}, \left[ \begin{array}{l}
\text{knows} (\text{mastersecret}) \end{array} \right]).

analyzing results ...

model found/total failure

time limit information: 19 total / 18 strategy
(leave wrapping).
task myUML_PID1491 on atbroy1 has status SUCCESS
(model found by strategy 300) consuming 1 seconds
deleting temporary files.
e-SETHEO done. exiting
Model vs. Implementation

Sent / received data

Automated verification

Test cases

Trace

Generate

Consistent?

Implementa-
tion

Jessie – using RSA & Server authentication

Client

C:Client

ClientHello(R_{C})

ServerHello(R_{S})

Certificate(sign_{K_{C}}(S:K_{S}))

ClientKeyExchange(enc_{K_{C}}(pms))

Finished(symenc_{md5_{C}}, symenc_{sha_{C}}))

S:Server
Model-based Security Testing: Strategies

**Internal:** Ensure test-case selection from models does not miss critical cases: Select according to information on criticality / security.

**External:** Test code against possible environment interaction generated from parts of the model (e.g. deployment diagram with information on physical environment).
To address problems mentioned earlier, test cases generation needs to be informed by actual code.

- Generate control flow graph.
- Analyze for criticality requirements (e.g. security).
- Use to generate critical test-cases.
Example: SSL

I) Identify program points:
   value (r), receive (p), guard (g), send (q)

II) Check guards enforced
Checking Guards

Guard $g$ enforced by code?

b) Generate runtime check for $g$ at $q$ from diagram: simple + effective, but performance penalty.

c) Testing against checks (symbolic abstractions for crypto).

d) Automated formal local verification: conditionals between $p$ and $q$ logically imply $g$ (uses Prolog).
msg = Handshake.read(din, certType);

session.trustManager.checkServerTrusted(peerCerts, suite.getAuthType());

msg = new Handshake(Handshake.Type.CLIENT_KEY_EXCHANGE, ckex);

msg.write(dout, version);

only possible way without throwing exception
Some Applications

Analyzed designs / implementations / configurations e.g. for

- Biometry- or smart-card-based identification
- authentication (crypto protocols)
- authorization (user permissions, e.g. SAP systems)

Analyzed security policies, e.g. for privacy regulations.
Common Electronic Purse Spec.

Global elec. purse standard (Visa, 90% market).

Smart card contains account balance, performs crypto operations securing each transaction.

Formal analysis of load and purchase protocols: three significant weaknesses: purchase redirection, fraud bank vs. load device owner.
Biometric Authentication System

In development by company in joint project.

Uses bio-reference template on smart-card.

Analyze given UML spec.

Discovered three major weaknesses in subsequently improved versions (misuse counter circumvented by dropping / replaying messages, smart-card insufficiently authenticated by mixing sessions).

Finally developed version secure by our analysis.
Bank Application

Security analysis of web-based banking application, to be put to commercial use (clients fill out and sign digital order forms).

Layered security protocol (first layer: SSL protocol, second layer: client authentication protocol)

Security requirements: confidentiality, authenticity

Successfully verified using model-based security.
Intranet Information System

MetaSearch Engine: Personalized search in company intranet (including password protected).

Some documents highly security-critical.

More than 1,000 potential users, index 280,000 documents, allow 20,000 queries per day.

Seamlessly integrated in enterprise-wide security reference architecture. Provides security services to applications, including user authentication, role-based access control, global single-sign-on and hook-up of new security apps.

Successfully analyzed using model-based security.
Mobile Communications

- Application of Model-based Security Assurance at Mobile Communication Systems at O2 (Germany)
- All 62 relevant security requirements derived from the security policy could successfully be established using the approach
How does it compare?

- Empirical study to compare classical vs. model-based testing: embedded software / Automotive (window controller). In cooperation with colleagues from BMW / Elektrobit. From the comparison:

<table>
<thead>
<tr>
<th>Modelchecking</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examines an abstract model</td>
<td>Examines a physical or concrete system</td>
</tr>
<tr>
<td>Cheap and early verification (without setting up complex in-the-loop-test environments)</td>
<td>In-the-loop-tests take place in an environment near to the real one</td>
</tr>
<tr>
<td>Proof of correctness of properties possible</td>
<td>No proof of correctness of properties possible</td>
</tr>
<tr>
<td>Uses selected user specific properties</td>
<td>Uses often many, superficial test cases</td>
</tr>
</tbody>
</table>
3 Things to Remember

- Quality Assurance for security-critical systems is very difficult.
- Some specific security properties (e.g. wrt brute-force attacks) cannot be established using conventional testing methods.
- Model-based Security Testing based on UMLsec is a corner-stone of Security QA.

... don’t get lost in the security maze...
More information: www.umlsec.org

Questions?

J.Jurjens@open.ac.uk
Backup
Unlinked, cash-based load transaction (on-line).

Load value onto card using cash at load device.

Load device contains Load Security Application Module (LSAM): secure data processing and storage.

Card account balance adjusted; transaction data logged and sent to issuer for financial settlement.

Uses symmetric cryptography.
Load «data security»

Card «critical»
- secrecy: \{K_{CI}\}
- integrity: \{K_{CI}, cep, nt, rc_{nt}\}

cep, nt, rc_{nt} : Data; K_{CI} : Keys

Init(lดา,n) Credit(s2,r1)

LSAM «critical»
- secrecy: \{K_{LI}\}
- integrity: \{K_{LI}, l�다, n, r1l_n, r2l_n, r_n, m_n\}

l�다, n, r1l_n, r2l_n, m_n : Data
K_{LI}, r_n : Keys

RespI(cep, nt, sl, 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, lดา, m, nt, sl, ml, h
hl, h2l)
Comp(cep, lดา, m, nt, r2l, s3)

Load
- Load a security

InCard
- Load input data

InLSAM
- Load input data

InIssuer
- Load input data

InLink
- Load input data

InComp
- Load input data
Audit Security

No direct communication between card and cardholder.
Manipulate load device display.

Use post-transaction settlement scheme.
Relies on secure auditing.

Verify this here (only executions completed without exception).
Flaw I

$\text{Proof}^{"}$

for bank that load machine received money. But: $r_n$ shared between bank and load machine.
Flaw II

$rc_{nt}$: „Proof“ for LSAM that load device received only amount $m_n$.

But: LSAM cannot prove validity of $rc_{nt}$. 