SECURITY TESTING USING MODELS AND TEST PATTERNS

Presented by [Bruno Legeard, Elizabeta Fournieret]
MODEL-BASED SECURITY TESTING

Positionning with respect to the state of the art
Model-Based Testing

- Model-Based Testing (MBT) is based or involved on models, called MBT models.

- It extends and supports classic test design techniques integrating closely with the existing lifecycle in an enterprise.
MBT Process

Functional needs
Business needs
Requirements

Test design and implementation

Test Repository
(Excel, HP/ALM…)

Functional tests

Manual execution
& scripts for automation

User Conference on
Advanced Automated Testing
MBT Process

Functional needs
Business needs
Requirements

Modeling for test generation

Automatic or manual test conception

Test Repository (Excel, HP/ALM…)

Functional tests

Manual execution
& scripts for automation

User Conference on Advanced Automated Testing

26-28/10/2016
What are the benefits of MBT from an industry point of view?

What do you expect from a model-based approach to testing?

- Our test design shall become more efficient (“cheaper tests”).
- Our tests shall become more effective (“better tests”).
- Models shall help us to manage the complexity of the system with respect to testing.
- We wish to improve the communication between stakeholders.
- Models shall help us to start test design earlier.

Expectations of MBT practitioners (from 2014 MBT User Survey)
Model-Based Security Testing

• What are the challenges in Security Testing?

• Where MBT stands for Security Testing?
Security Testing Approaches

Manual Techniques

- Code review
- Manual Penetration Testing
- Intrusive proxies (Burp suite, Webscarab, ...)

Automated Techniques

- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- Vulnerability Scanners, Fuzzing tools, ...

Static Techniques

- Static Techniques
- Dynamic Techniques
Security Testing Approaches

Static Techniques
- Code review
- Static Application Security Testing (SAST)

Dynamic Techniques
- Manual Penetration Testing
- Dynamic Application Security Testing (DAST)
- Intrusive proxies (Burp suite, Webscarab, …)
- Vulnerability Scanners, Fuzzing tools, …

Manual Techniques
- Automated Techniques

Static Techniques vs Dynamic Techniques: Static Techniques are more manual and review-based, while Dynamic Techniques involve more automated and intrusive methods.
DAST Approaches

Model-based Security Testing (MBST) Approaches

- Rely on a variety of techniques to compute black-box test cases
  - Patterns,
  - Fuzzing,
  - Model-checking, etc.

- Promising results for pattern-based techniques
  - Better detection rates than scanners
  - Less time consuming than manual penetration testing
  - Test execution integrated within large-scale testbeds
IN PRACTICE

Pattern-driven and Model-Based Security Testing
Objectives in theory and in practice

- **Objective 1**: Improve the coverage of security requirements, keeping overall traceability

- **Objective 2**: Increase the fault detection capability of the test suite

- **Objective 3**: Cost-effectiveness
Pattern-based process to reach the objectives

1. Identification of vulnerabilities & security requirements
2. Definition of security test requirements
3. Formalization of security test requirements

Security Test Repository
MBT Process for security testing

Security needs & requirements

Risk Analysis

Modeling for test generation

Automatic or manual test conception

Test Repository (Excel, TTCN-3, ALM…)

Functional tests

Manual execution & scripts for automation

Security Test Patterns

Security Test Purposes

User Conference on Advanced Automated Testing

26-28/10/2016
MBT Process for security testing

Security needs & requirements

Modeling for test generation

Automatically generated tests

Manual execution & scripts for automation

Test Repository (Excel, TTCN-3, ALM…)

User Conference on Advanced Automated Testing

26-28/10/2016
Background on the MBT Approach
Behavioral modeling notation (UML4MBT) based on UML metamodel:

- **Class diagrams** specify the static structure
- **Object diagrams** specify concrete entities and initial state
- **OCL** (Object Constraint Language) to describe its behavioral characteristics
Background on the MBT Approach

Test selection depending on two test characteristics:

➢ **Functional behavioral testing** → activate all behaviors
➢ **Security testing** → formalization of test scenarios using temporal properties (TOCL) and test patterns (TP)

Test generation relies on symbolic state exploration of the model:

➢ **Functional behavioral testing** → A test target per behavior to activate
➢ **Security testing** → Test targets derived by unfolding each TOCL and TP
Test cases must be **concretized** to be made executable:

- Conformity table between abstract data and concrete data
- Implementation of class operations

Test cases may be published for test management and execution tools:

- HP Quality Center, TestLink, etc.
TOCL and TP test selection criteria

- **TOCL and TP** make possible to generate tests that exercise *corner cases*, relevant when testing security properties and vulnerabilities.

- **TOCL** allows to express *temporal properties*, for instance of succession or precedence, contributing to the MBT process with:
  - Evaluation of the existing tests coverage
  - Verification of the model’s conformance to these properties
    - Simplifying the model debugging

- **TP** allow to express in terms of *procedures of tests* based on a verbose representation and using the experts experience and knowledge on the system vulnerabilities.
Design of Temporal Properties using TOCL

• TOCL = Temporal OCL
  • overlay of OCL to express temporal properties
  • based on Dwyer et al. property patterns [DAC99]
  • does not require the use of a complex formalism (e.g. LTL, CTL)

• TOCL Property = Pattern + Scope
  • Pattern: describes occurrences or orderings of events
    (always, never, eventually k times, precedes, follows)
  • Scope: describes the observation window on which the pattern is supposed to hold
    (globally, between, after, before)

Test Patterns

• Test Purpose Language
  • relies on the use of *keywords* to represent a test scenario expressing a combinations of test steps and test input parameters
  • powerful and easy to read by test engineers
  • does not require the use of a complex formalism (e.g. LTL, CTL)

• Test Purpose (TP) = Quantifiers + Blocks
  • *Quantifiers*: describes the context in which an action defined by the block will be activated
    • *(for_each behavior $X$ from {list})*
  • *Blocks*: describes the actions to be taken in order to activate a state in the model
    • *(use any_operation any_number_of_times to_active $X)*
MBT Process for Security Testing

1. Functional Requirements
2. Security Properties
3. MBT Model
4. MBT Tool
5. System Under Test

- Structural test selection
- Dynamic test selection
- Coverage monitoring
- Evaluation
- Publication
- Test repository
- Execution
MBT Process for Security Testing

1. Functional Requirements
2. Security Properties
3. MBT Tool
   - Structural test selection
   - Dynamic test selection
4. Publication
5. System Under Test
   - Execution
   - Evaluation

Coverage monitoring

TOCL

TP

User Conference on Advanced Automated Testing

26-28/10/2016
MBT Process for Security Testing

1. Functional Requirements
2. Security Properties
3. MBT Tool
4. Publication
5. System Under Test

- Structural test selection
- Dynamic test selection

- Coverage monitoring
- Execution
- Evaluation

MBT Tool

System Under Test

Test repository

User Conference on Advanced Automated Testing

26-28/10/2016
MBT Process for Security Testing

1. Functional Requirements
2. Security Properties
3. MBT model
4. MBT Tool
5. System Under Test

Structural test selection
Dynamic test selection
Coverage monitoring
Execution
Publication
Evaluation

User Conference on Advanced Automated Testing
26-28/10/2016
MBT Process for Security Testing

1. Functional Requirements
2. Security Properties
3. MBT Tool
4. Publication
5. System Under Test

- Structural test selection
- Dynamic test selection
- Coverage monitoring

Evaluation
MBT Process for Security Testing

1. Functional Requirements
2. Security Properties
3. MBT Tool
4. Publication
5. System Under Test
6. Coverage monitoring

- Structural test selection
- Dynamic test selection
- Evaluation
- Execution
EXPERIENCE IN SECURITY COMPONENTS TESTING
Experience in security components testing

Security Components have two categories of test requirements:
- Functional Requirements
- Security Requirements
Experience in security components testing

- **PKCS#11** is an RSA standard that defines an interface called *Cryptoki* to promote interoperability and security of cryptographic tokens.

- **Scope:** 24 functions most commonly present in the tokens, such as session, token, key and user management functions, as well as cryptographic functions for signing messages and verifying signatures.

- To ensure the repeatability of the MBT process we chose **SoftHSM** - virtual cryptographic store largely used for exploring PKCS#11 without the necessity to possess an HSM (created by the group OPENDNSSEC).
PKCS#11: Functional description

Specification documents:

PKCS #11 v2.20: Cryptographic Token Interface Standard

RSA Laboratories
28 June 2004

Table of Contents
1 INTRODUCTION
2 SCOPE
3 REFERENCES
4 DEFINITIONS
5 SYMBOLS AND ABBREVIATIONS
6 GENERAL OVERVIEW
6.1 INTRODUCTION

♦ C_OpenSession

```c
CK_DEFINE_FUNCTION(CK_RV, C_OpenSession)(
    CK_SLOT_ID slotID,
    CK_FLAGS flags,
    CK_VOID_PTR pApplication,
    CK_NOTIFY Notify,
    CK_SESSION_HANDLE_PTR phSession
)
```

C_OpenSession opens a session between an application and a token in a particular slot. `slotID` is the slot’s ID; `flags` indicates the type of session; `pApplication` is an application-defined pointer to be passed to the notification callback; `Notify` is the address of the notification callback function (see Section 11.17); `phSession` points to the location that receives the handle for the new session.

When opening a session with C_OpenSession, the `flags` parameter consists of the logical OR of zero or more bit flags defined in the CK_SESSION_INFO data type. For legacy reasons, the CKF_SERIAL_SESSION bit must always be set; if a call to C_OpenSession does not have this bit set, the call should return unsuccessfully with the error code CKR_PARALLEL_NOT_SUPPORTED.
## PKCS#11: Functional requirements

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C_DigestFinal</td>
<td>C_DigestFinal finishes a multiple-part message-digesting operation, returning the message digest.</td>
<td>Section 11.10 p.151</td>
<td>C_DigestFinal</td>
<td>Effet : OPERATION_NOT_INITIALIZED</td>
<td>Effet : BUFFER_TOO_SMALL</td>
<td>Effet : OK</td>
</tr>
<tr>
<td>16</td>
<td>C_SignInit</td>
<td>C_SignInit initializes a signature operation</td>
<td>Section 11.11 p.152</td>
<td>C_SignInit</td>
<td>Effet : USER_NOT_LOGGED_IN</td>
<td>Effet : OPERATION_ACTIVE</td>
<td>Effet : MECHANISM_INVALID - the mechanism is not valid</td>
</tr>
<tr>
<td>17</td>
<td>C_Sign</td>
<td>C_Sign signs data in a single part</td>
<td>Section 11.11 p.153</td>
<td>C_Sign</td>
<td>Effet : OPERATION_NOT_INITIALIZED</td>
<td>Effet : BUFFER_TOO_SMALL</td>
<td>Effet : USER_NOT_LOGGED_IN</td>
</tr>
<tr>
<td>18</td>
<td>C_SignUpdate</td>
<td>C_SignUpdate continues a multiple-part signature operation</td>
<td>Section 11.11 p.154</td>
<td>C_SignUpdate</td>
<td>Effet : OPERATION_NOT_INITIALIZED</td>
<td>Effet : USER_NOT_LOGGED_IN</td>
<td>Effet : ARGUMENTS_BAD - the data to sign is null</td>
</tr>
<tr>
<td>19</td>
<td>C_SignFinal</td>
<td>C_SignFinal finishes a multiple-part signature operation</td>
<td>Section 11.11 p.154-155</td>
<td>C_SignFinal</td>
<td>Effet : OPERATION_NOT_INITIALIZED</td>
<td>Effet : BUFFER_TOO_SMALL</td>
<td>Effet : USER_NOT_LOGGED_IN</td>
</tr>
</tbody>
</table>
PKCS#11: Test model

The **test model** is a System Under Test **abstraction**, representing its expected behavior.

### Class Diagram:
- Represents the business objects that can be used by the System Under Test
- Classes own operations that can be called on the system under test (control and observation points)

### OCL:
- Represents the expected behavior of an operation on the System Under Test, regarding the system state and the operation parameters

### Instance Diagram:
- Represents the initial state of the System Under Test
PKCS#11: Test model

The **test model** is a System Under Test **abstraction**, representing its expected behavior.

### Class Diagram:
- represents the business objects that can be used by the System Under Test
- classes own operations that can be called on the system under test (control and observation points)

### OCL:
- represents the expected behavior of an operation on the System Under Test, regarding the system state and the operation parameters

### Instance Diagram:
- represents the initial state of the System Under Test
PKCS#11: Class Diagram
PKCS#11: Test model

The **test model** is a System Under Test **abstraction**, representing its expected behavior.

**Class Diagram:**
- \( \rightarrow \) represents the business objects that can be used by the System Under Test
- \( \rightarrow \) classes own operations that can be called on the system under test (control and observation points)

**OCL:**
- \( \rightarrow \) represents the expected behavior of an operation on the System Under Test, regarding the system state and the operation parameters

**Instance Diagram:**
- \( \rightarrow \) represents the initial state of the System Under Test
### PKCS#11:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>@REQ</td>
<td>Requirement description</td>
<td>Spec v2.0</td>
<td>Main function</td>
<td>Effet</td>
</tr>
<tr>
<td>C_DigestFinal</td>
<td>C_DigestFinal finishes a multiple-part message-digesting operation, returning the message digest</td>
<td>Section 11.10 p.151</td>
<td>C_DigestFinal</td>
<td>OPERATION_NOT_INITIALIZED</td>
</tr>
<tr>
<td>C_SignInit</td>
<td>C_SignInit initializes a signature operation</td>
<td>Section 11.11 p.152</td>
<td>C_SignInit</td>
<td>USER_NOT_LOGGED_IN</td>
</tr>
<tr>
<td>C_Sign</td>
<td>C_Sign signs data in a single part</td>
<td>Section 11.11 p.153</td>
<td>C_Sign</td>
<td>OPERATION_NOT_INITIALIZED</td>
</tr>
</tbody>
</table>

```plaintext
if((self.C_SignInitialized=CK_BOOLEAN::CK_FALSE) and (self.mechanism.oclIs Undefined()))=false then
  self.CKR = CK_RV::CKR_OPERATION_ACTIVE
  /**@AIM:OPERATION_ACTIVE*/

else
  if((session.slot.logged_user.oclIsUndefined()) then
    self.CKR = CK_RV::CKR_USER_NOT_LOGGED_IN
    /**@AIM:USER_NOT_LOGGED_IN*/
  else
    let mech : Mechanism = mechs->any(true) in
    self.C_SignInitialized = CK_BOOLEAN::CK_TRUE and
    mech.key = key and
    self.mechanism = mech and -- creating the link "is currently used"
    self.CKR = CK_RV::CKR_OK
    /**@CKR:OK*/
    /**@AIM:OK*/
  endif
endif
```
PKCS#11: Test model

The **test model** is a System Under Test **abstraction**, representing its expected behavior.

**Class Diagram:**
- \( \rightarrow \) represents the business objects that can be used by the System Under Test
- \( \rightarrow \) classes own operations that can be called on the system under test (control and observation points)

**OCL:**
- \( \rightarrow \) represents the expected behavior of an operation on the System Under Test, regarding the system state and the operation parameters

**Instance Diagram:**
- \( \rightarrow \) represents the initial state of the System Under Test
PKCS#11: Initial State

Instances of SmartTestingModel: Initial Data

- SmarTestingModel
  - CryptoKI
    - CryptoKIInfo
      - CKR = CKR_NONE
      - C_DigestInitialized = CK_FALSE
      - C_DigestUpdateCalled = CK_FALSE
      - C_INITIALIZED = CK_FALSE
      - C_SignInitialized = CK_FALSE
      - C_SignUpdateCalled = CK_FALSE
      - C_VerifyInitialized = CK_FALSE
      - C_VerifyUpdateCalled = CK_FALSE
      - C_VerifyUpdate_Data = DATA_NULL (default)
      - mechanism = Ø
      - slots = {SlotEmptyInstance, SlotWithNonInitTokenInstance, SlotWithTokenInstance}
      - tokens = {TokenInitializedInstance, TokenNotInitializedInstance}
  - Key
    - KeyInfo
      - CKA_PRIVATE = CKA_TRUE
      - CKA_SING = CKA_TRUE
      - CKA_VERIFY = CKA_TRUE
      - key_handle = KEY_ID1
      - mechanism = Ø
      - pDataSigned = Ø
      - session_handle = HANDLE_INVALID_SESSION (default)
      - token = Ø
      - user = Ø
      - KeyInfo2
      - KeyInfo3

Instance manager
Exercise Functional vs Security Functional Requirements

• Functional Requirement
  • Cryptoki signs data if the user logged, otherwise it responds with an error code USER_NOT_LOGGED_IN

• Security Functional Requirement
  • When the logout successfully executes, any of the application’s handles to private objects become invalid (even if a user is later logged back into the token). In addition, all private session objects from sessions belonging to the application are destroyed.

How will you test these requirements?
How you will express them, using TP or TOCL?
Exercise Solution (1/4)

- Functional Requirement
  - The tool creates a test case by choosing the shortest path
Exercise Solution (2/4)

- Functional Requirement
  - The tool exports the test into a specific format

```cpp
class pkcs11_testsuite : public CPPUNIT::TestFixture
{
    CPPUNIT_TEST_SUITE(pkcs11_testsuite);
    CPPUNIT_TEST(C_SignInit_7f3_ffca_c75);
    CPPUNIT_TEST_SUITE_END();

    public:
    void setUp();
    void tearDown();
    void C_SignInit_7f3_ffca_c75();
};

#undef __SOFTHSM_V2

void pkcs11_testsuite::C_SignInit_7f3_ffca_c75()
{
    string argument1[]="VOID_NULL_PTR";
    BenchmarkTools::op_adaptor("C_Initailize",argument1);

    string argument2[]="CKR_OK";
    CPPUNIT_ASSERT_MESSAGE("Operation C_Initailize Actual result is "+BenchmarkTools::get_Last_CKR()+" Expected result is CKR_OK",
                           BenchmarkTools::op_adaptor("checkResult",argument2));

    string argument3[]="SLOT_VALID_TOKEN_NOTINIT","PIN_USER_TOKEN_NOT_INIT","TOKEN_NOT_INITIALIZED";
    BenchmarkTools::op_adaptor("C_InitToken",argument3);
```
Exercise Solution (3/4)

• Security Functional Requirement

\[
\text{for} \ \text{each literal } $KEY \ \text{from} \ \text{KEY}_{1} \ \text{or} \ \text{KEY}_{2} \ \text{or} \ \text{KEY}_{4} \ \text{or} \ \text{KEY}_{5},
\]

\[
\text{use any_operation any_number_of_times then}
\]

\[
\text{use cryptoki.C_OpenSession(\_)}
\]

\[
\quad \text{to activate behavior_with_tags \{} \text{CKR:OK} \}\ \text{then}
\]

\[
\text{use cryptoki.C_Login(\_)}
\]

\[
\quad \text{to activate behavior_with_tags \{} \text{AIM:C_Login/CKU_USER_RW} \}\ \text{then}
\]

\[
\text{use cryptoki.nominal_generateKey(\_,\_,\_CKU_TRUE,\_)}
\]

\[
\quad \text{to activate behavior_with_tags \{} \text{AIM:GENERATE_KEY/OK} \}\ \text{then}
\]

\[
\text{use cryptoki.C_Finalize(\_)} \ \text{then}
\]

\[
\text{use any_operation any_number_of_times then}
\]

\[
\text{use cryptoki.C_Login(\_)}
\]

\[
\quad \text{to activate behavior_with_tags \{} \text{AIM:C_Login/CKU_USER_RW} \}\ \text{then}
\]

\[
\text{use cryptoki.C_SignInit(\_,\_, $KEY)}
\]
Exercise Solution (4/4)

• Security Functional Requirement

<table>
<thead>
<tr>
<th>Steps</th>
<th>Test detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default model instance</td>
<td></td>
</tr>
<tr>
<td>Initialized model instance</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.setUp()</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.C_Initialize(VOID_NULL_PTR) = CKR_OK</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.C_Login(HANDLE_RW_PUBLIC_SESSION, CKU_USER, PIN_USER_TOKEN_INIT) = CKR_OK</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.nominal_generateKey(HANDLE_RW_PUBLIC_SESSION, CK_TRUE, CK_TRUE, KEY_ID1) = CKR_OK</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.C_Finalize(VOID_NONNULL_PTR) = CKR_ARGUMENTS_BAD</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.C_CloseAllSessions(SLOT_VALID) = CKR_OK</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.C_OpenSession(SLOT_VALID, CKF_SERIAL_SESSION, CKF_RW_SESSION, VOID_NONNULL_PTR, VOID_NONNULL_PTR, HANDLE_RW_PUBLIC_SESSION_PTR) = CKR_OK</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.C_Login(HANDLE_RW_PUBLIC_SESSION, CKU_USER, PIN_USER_TOKEN_INIT) = CKR_OK</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.C_SignInit(HANDLE_RW_PUBLIC_SESSION, CKM_SHA512_PTR, KEY_ID1) = CKR_OBJECT_HANDLE_INVALID</td>
<td></td>
</tr>
<tr>
<td>CryptokiInstance.tearDown()</td>
<td></td>
</tr>
</tbody>
</table>
# PKCS#11 in numbers

## PKCS#11 set up metrics

<table>
<thead>
<tr>
<th>Test Requirement category</th>
<th>#FR</th>
<th>#SFR</th>
</tr>
</thead>
<tbody>
<tr>
<td>general purpose</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>slot and token management</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>session management</td>
<td>32</td>
<td>9</td>
</tr>
<tr>
<td>object management</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>digesting</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>signing</td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>verifying signatures</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>158</strong></td>
<td><strong>48</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PKCS#11 model element</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#classes</td>
<td>9</td>
</tr>
<tr>
<td>#enumerations</td>
<td>20</td>
</tr>
<tr>
<td>#enum. literals</td>
<td>123</td>
</tr>
<tr>
<td>#associations</td>
<td>17</td>
</tr>
<tr>
<td>#class attributes</td>
<td>34</td>
</tr>
<tr>
<td>#operations</td>
<td>24</td>
</tr>
<tr>
<td>#observations</td>
<td>1</td>
</tr>
<tr>
<td>#behaviors</td>
<td>206</td>
</tr>
<tr>
<td>#tcl properties</td>
<td>50</td>
</tr>
<tr>
<td>#test purposes</td>
<td>5</td>
</tr>
<tr>
<td><strong>#LOC</strong></td>
<td><strong>1308</strong></td>
</tr>
</tbody>
</table>

LOC: Lines of OCL constraints
PKCS#11 in numbers

PKCS#11 test generation coverage and execution metrics

<table>
<thead>
<tr>
<th>Test Selection Criterion</th>
<th>#Test targets</th>
<th>#Test cases</th>
<th>Cov. in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>206</td>
<td>184</td>
<td>100</td>
</tr>
<tr>
<td>TOCL</td>
<td>311</td>
<td>90</td>
<td>31</td>
</tr>
<tr>
<td>Test Purpose</td>
<td>24</td>
<td>24</td>
<td>9</td>
</tr>
<tr>
<td>Manual</td>
<td>24</td>
<td>24</td>
<td>45</td>
</tr>
</tbody>
</table>

Fig. Distinct fault detection capabilities per coverage requirement
To learn more about this case study

Chapter 11 – PKCS #11 case study

Published in 2016 – Related to the ISTQB® Model-Based Tester Certification
EXPERIENCE IN SECURITY TESTING FOR IOT SYSTEMS
IoT Existing Security Frameworks

- **OWASP IoT** defines a framework that gathers information on security issues associated to the IoT development, deployment or technology assessment. However, it remains **too high level and lacks a specific methodology** that could be used in a systemic way - for instance in security audits.

- **GSMA** provides a set of security guideline documents that target all IoT involved entities (service providers, device manufacturers, developers, network operators etc.). However, the GSMA Security Framework only points to **currently available solutions, standards and best practices**.

- **oneM2M** identifies 4 security domains (*Application, Intra Common Services, Inter Common Services, Underlying Network*), and 3 layers (*Security Functions, Security Environment Abstraction, Secure Environment*). Selected as **a starting point for ARMOUR risk analysis and mitigation methodology**.
IoT Security Framework [H2020 ARMOUR Project]
IoT Security Framework [H2020 ARMOUR Project]

The ARMOUR security framework takes as entry the oneM2M vulnerabilities, threats and risk assessment methodology, and for each experiment:

• performed an analysis of the **risks/vulnerabilities** to be considered during all phases of the experimentation.
• defined a **set of countermeasures** in order to address the vulnerabilities and reduce associated risk.
• described the **scenarios and methodology** that will be the basis for the experiments execution.

Vulnerability classification and patterns in IoT systems

• Built on existing frameworks and adapt them if necessary,
• A vulnerability pattern intends to describe vulnerabilities, their conditions of occurrence and impacts.
• CVE (Common Vulnerabilities and Exposure) framework: dictionary of publicly known information security vulnerabilities and exposures
MBT Methodology and Framework for Large-Scale IoT Testing

- MBT models
- Security
- Standards
- Vulnerability patterns
- automated MBT approach
- manual test conception
- in-house approach
- Keeping overall traceability
- Security tests
  - TTCN-3
  - TPLan
- automated execution
- Local or FIRE testbeds

Keeping overall traceability
MBT Methodology in 5 steps applied to oneM2M

1. Vulnerability patterns identification
2. MBT model based on ARMOUR guidelines
3. Security test pattern formalization using the Smartesting Test Purpose Language
4. MBT test generation based on ARMOUR test strategies using CertifyIt
5. Publication in TPLan – test description and TTCN-3 test scripts

TPLan & TTCN-3

testbed execution and results
Results from the oneM2M experience

1. Security test pattern formalization for test generation
2. Definition of generic MBT models for TTCN-3 and TPLan production
3. TTCN-3 publisher
4. TPLan publisher
5. Detection of inconsistencies in the IoT platform under test with the specification during the oneM2M interoperability event in South Korea.
CONCLUSION

Lessons learnt from experience
LESSONS LEARNT

Based on MBT pitfalls and drawbacks

• MBT is not just a matter of tooling
  • careful evaluation of the organisation is necessary for a successful adoption
  • efficient & effective test strategy on long term scale

• MBT models are not always correct
  • adequate tools are necessary to measure the quality of the models, as it leads to quality of test cases

• MBT generates a myriad of test cases
  • to deal with it adequate formalism are necessary to benefit from domain experts experience
LESSONS LEARNT

Benefits from the MBT approach based on our experience

- it produced cheaper tests (average time spent of modelling - 2 days)
- it produces better tests (increased fault detection, better model quality)
- MBT models are clear
- early test design (detection of inconsistencies in specification)
CONCLUSION AND FUTURE WORK

• Model-Based Security Testing position at the research and industry state of the art
• MBT tooling extended to security testing
• Initial MBT Framework for Security Testing of IoT systems at different levels
• OneM2M experience ➔ A. Ahmad presentation on Friday
• Security is number one challenge in the IoT domain ➔ Looking forward for new proof of concepts
“Testing is always model-based!”
Robert Binder

THANK YOU
QUESTIONS?

Source - http://model-based-testing.info