Model-Based Fuzzing for Security Testing

Ina Schieferdecker, Jürgen Großmann, Martin Schneider
Fraunhofer FOKUS

21 April 2012, Montreal, Canada
My Testing Context

- Research and Teaching at FU Berlin
- Applied research at Fraunhofer FOKUS
- Standardization at ETSI and OMG
- ITEA Project DIAMONDS
Outline

• Introduction
• The Project Context
• Behavior Fuzzing with Sequence Diagrams
• Data Fuzzing with TTCN-3
Do you have a feeling about security?

A CRYPTO NERD’S IMAGINATION:
His laptop’s encrypted. Let’s build a million-dollar cluster to crack it.

WHAT WOULD ACTUALLY HAPPEN:
His laptop’s encrypted. Drug him and hit him with this $5 wrench until he tells us the password.

No good! It’s 4096-bit RSA!
Blast! Our evil plan is foiled!

Got it.
**Terminology**

- **Security testing**
  Testing whether a system under test meets the specified security objectives.

- **Model-based testing**
  Model-based testing is an umbrella of approaches that generate tests from models.

- **Model-based security testing**
  Is an umbrella of approaches that generate tests from model, where the tests check if a system under test meets the specified security objectives. Model-based fuzzing is one of the approaches to randomize tests in a smart manner.
Outline

• Introduction
• The Project Context
• Behavior Fuzzing with Sequence Diagrams
• Data Fuzzing with TTCN-3
DIAMONDS Objectives

• Security engineering is increasingly challenged by
  - the openness,
  - dynamics, and
  - distribution of networked systems

• Most verification and validation techniques for security have been developed in the framework of static or known configurations, with full or well-defined control of each component of the system

• This is not sufficient in networked systems, where control and observation of remote (sub) systems are dynamically invoked over the network
  → Combination of active and passive security testing
  → Usage of fuzz tests (for unknown issues) and functional tests (for security measures)
  → Combination of risk analysis and test generation
  → Integration of automated test generation, test execution and monitoring
ITEA2 DIAMONDS Project

- **Diamonds** will enable efficient and automated security testing methods of industrial relevance for highly secure systems in multiple domains (incl. e.g. banking, transport or telecommunication)

- **Business Value**
  - Multiple Domains
  - Pre-Standardization Work
  - Novel Integration of Testing, Security and Risk-Analysis

- **Expected Results**
  - Risk-driven Security Testing Methodology
  - Model-Based Security Test Techniques
  - Security Test Patterns Catalogue
Model-Based Testing in a Nutshell

Test models enable
- objective test procedures,
- knowledge sustainment,
- test quality assessment,
- test reuse, and
- technology-independence
Model-Based Testing for Security

- **Security objectives**
- **Security targets & scenarios**

**Requirements**
- System Model
- Testmodel

- Represents
- Validates
- Realize

**System Model**
- System Code Derivation
- Represents

**Testmodel**
- Test System Generation
- Represents

**System (Under Test)**
- Validates

**Test Execution System**
- Represents

Hypotheses
Randomization
Metrics

Performance & Evaluation

© DIAMONDS Consortium 2010-2013
Outline

• Introduction

• The Project Context

• Behavior Fuzzing with Sequence Diagrams

• Data Fuzzing with TTCN-3
Fuzz Testing

• Fuzzing originally describes the random generation of test vectors (Miller et. al. in the early 1990s).

• Fuzzing is about **injecting invalid or random inputs** in order
  - to reveal *unexpected behaviour*
  - to identify *errors* and expose potential *vulnerabilities*.

• Ideally, fuzzers generate **semi-valid input data**, i.e. input data that is *invalid only in small portions*.

• Depending on fuzzer’s knowledge about the protocol, fuzzers can generate totally *invalid to semi-valid* input data.

Categorization of Fuzzers

• **Random-based fuzzers** generate randomly input data. They don’t know nearly anything about the SUT’s protocol.
  
  fuzzed input: HdmxH&k dd**&%

• **Template-based fuzzers** use existing traces (files, …) and fuzz parts of the recorded data.
  
  template: GET /index.html
  fuzzed input: GE? /index.html, GET /inde?.html

• **Block-based fuzzers** break individual protocol messages down in static (grey) and variable (white) parts and fuzz only the variable part.
  
  fuzzed input: GET /inde?.html, GET /index.&%ml

• **Dynamic Generation/Evolution-based fuzzers** learn the protocol of the SUT from feeding the SUT with data and interpreting its responses, e.g. using evolutionary algorithms. While learning the protocol, these fuzzers implicitly run fuzzing.

• **Model-based fuzzers** employ a model of the protocol. The model is executed on- or offline to generate complex interactions with the SUT. Thus, it is possible to fuzz data after passing a particular point (e.g. after authentication).
Model-Based Fuzz Testing

- **Random fuzzing**: has close to zero awareness of the tested interface
- **Mutation-based fuzzing**: mutate existing data samples to create test data, breaks the syntax of the tested interface into blocks of data, which it semi-randomly mutates.

- **Model-based fuzz testing**:
  - Uses models of the input domain (protocol models, e.g. context-free grammars), for generating systematic non-random test cases
  - In security testing purposes, the models are augmented with intelligent and optimized anomalies that will trigger the vulnerabilities in code.
  - Finds defects which human testers would fail to find
Mutation Testing & Fuzzing

Mutation testing is used for two purposes:
• to assess a **software component for anomalous inputs**,  
• to assess the **quality of test cases** for finding specific defects,  
• to **simulate usage situations** that are difficult to test.

There are two types of mutation testing:

a) **Data Fault Injection**  
   This is traditional data fuzzing submitting invalid or semi-valid input data to the SUT.

b) **Code Fault Injection on the side of the SUT**  
   Code Fault Injection is the modification of the SUT’s code by adding typical programming errors. That way quality of test suites can be assessed.

Besides the above-mentioned types of mutation testing, the **operational interface of the SUT can be stimulated in an invalid way.**

This type of mutation testing is called **behaviour fuzzing.**
Banking Domain Case Study:
Banknote Processing Machines
Banking Domain Case Study:
Olymp Currency Processing Network

CP  = Currency Processor
RS  = Reconciliation Station
CC  = Control Center
VMS = Vault Management System

Data Flow:
- VMS: deposit data
- CP: configure & monitor CP
- CC:
  - Shift + Reject data
  - updated Shift + Reject data

Giesecke & Devrient
© DIAMONDS Consortium 2010-2013
Banking Domain Case Study: TTCN-3 based Test Architecture

- **Test Goal**
  Integration testing of any configurable subsystem.

- **Overview**
  Components not part of the system under test (SUT) are replaced by the tester.

  The system is stimulated and observed by sending and receiving the C# objects.

- **Challenges**
  C/C# interoperability between tester and SUT.
  C# object type representation in TTCN-3
  Transfer between TTCN-3 messages and C# object representations.
Banking Domain Case Study:
Processing Steps

Risk Analysis
Modeling
Test Generation
TTCN-3 Code Generation
Test Execution

Attacker sends “SQL injection” crafted messages over the Message Router

: TC
   1: f_CP_login(p_user="OP1")

: CP
   2: f_CP_selectProcessingModeUS(p_mode=Real)
   3: f_CP_selectDenominationUS(p_denom=USD5)

---

testcase ModeTest_GD08_3_3_7_fuzzed_TestCase_219 ()
runs on Comp_CP_RS
system System_CP_RS
{
  var integer i, v_total, v_rjc;
  _mctSetup_CP_RS(CFRSStartingMode:All);
  f_CP_login("OP1");
  f_CP_selectProcessingModeUS(ProcessingModeUS:Reel);

Development:
Behaviour Fuzzing of UML Sequence Diagrams

• In current state of the art, behaviour fuzzing is done only in small portions using state machines:
  - by fuzzing the message type,
  - by reordering messages and
  - by inserting, repeating and dropping messages.

• The motivation for behaviour fuzzing is that vulnerabilities can be revealed not only when invalid input data is accepted and processed by the SUT, but also by stimulating the SUT with invalid sequences of messages.

• To start with, we fuzz existing functional test cases.

• A real-world example is given in [Tak10] where a vulnerability in Apache web server was found by repeating the host header message in an HTTP request.


© DIAMONDS Consortium 2010-2013
Development:
Behaviour Fuzzing of UML Sequence Diagrams

- Test cases are generated by **fuzzing valid sequence diagrams**, e.g. **functional test cases**.
- Behaviour fuzzing is realized by changing the order and appearance of messages in two ways:
  - **By rearranging messages.** This enables straight-lined sequences to be fuzzed. Fuzzing operations are for example remove, move or repeat a message.
  - **By utilising control structures of UML 2 sequence diagrams**, such as combined fragments, guards, constraints and invariants. This allows more sophisticated and specific behaviour fuzzing.
- By applying one or more fuzzing operations to a valid sequence, **invalid sequences (= behaviour fuzzing test cases)** are generated.
Original vs. Fuzzed TTCN-3 Code
Starting with Known Behaviour

Original Test Case

SYSTEM INITIALIZATION
var integer i, v_total, v_rjc;
f_mtcSetup_CP_RS(CPRSStartingMode:All);

AUTHENTICATION
f_CP_logon("OP1");

CONFIGURATION
f_CP_selectProcessingModeUS (...);
f_CP_selectDenominationUS (...);
f_CP_selectOperationModeUS (...);
f_CP_selectStackingModeUS (...);

COUNTING
f_CP_bnProcessing(...);
f_CP_bnProcessingWithBarcode (...);
f_CP_bnProcessingWithWarning (...);
...

COMPLETION
f_CP_endShift();

Fuzzed Test Case

SYSTEM INITIALIZATION
var integer i, v_total, v_rjc;
f_mtcSetup_CP_RS(CPRSStartingMode:All);
f_CP_selectStackingModeUS (...);

AUTHENTICATION
f_CP_logon("OP1");

CONFIGURATION
f_CP_selectProcessingModeUS (...);
f_CP_selectDenominationUS (...);
f_CP_selectOperationModeUS (...);

COUNTING
f_CP_bnProcessing(...);
f_CP_bnProcessingWithBarcode (...);
f_CP_bnProcessingWithWarning (...);
...

COMPLETION
f_CP_endShift();
Behavior Fuzzing
Banking Domain Case Study

Results so far
• Running successfully over 30 test cases, no weaknesses found

Challenges
• Test execution takes more than 15 minutes per test case, optimization is needed
• Test evaluation and assessment is difficult (system crashes, error modes, etc.)
• Large number of generated test cases incl. duplicates (commutative equal combinations, e.g. the combination of the fuzzing operations “move message A”, “remove message A” is equal to “remove message A”)

Outlook
• Optimized test generation (risk and security-oriented test generation, avoiding duplicates)
• Combination with data fuzzing methods
• Defining metrics and coverage criteria

© DIAMONDS Consortium 2010-2013
Outline

• Introduction
• The Project Context
• Behavior Fuzzing with Sequence Diagrams
• Data Fuzzing with TTCN-3
Test Execution with TTCN-3

- TTCN-3: Testing and Test Control Notation
  - Abstract test specification
    - Data templates allow unlimited structuring and reusability of test data
    - Matching mechanism to compare an oracle to response data
    - Communication paradigms: message and procedure oriented ports
    - Parallel test components
  - Concrete test implementation
    - adapter and codec (coder/decoder of data types)

TTCN-3 Multi-Part Standard

- ETSI ES 201 873-1 TTCN-3 Core Language (CL)
- ETSI ES 201 873-2 TTCN-3 Tabular Presentation Format (TFT)
- ETSI ES 201 873-3 TTCN-3 Graphical Presentation Format (GFT)
- ETSI ES 201 873-4 TTCN-3 Operational Semantics
- ETSI ES 201 873-5 TTCN-3 Runtime Interface (TRI)
- ETSI ES 201 873-6 TTCN-3 Control Interfaces (TCI)
- ETSI ES 201 873-7 Integration of ASN.1
- ETSI ES 201 873-8 Integration of IDL
- ETSI ES 201 873-9 Integration of XML
- ETSI ES 201 873-10 TTCN-3 Documentation

- TTCN-3 Extension packages ETSI ES 202 78x
  - Advanced Parametrization
  - Behaviour Types
  - Configuration and Deployment Support
  - Performance and Real Time Testing
  - Extended Runtime Adaptation

- Standard available for download at http://www.ttcn-3.org
- Also standardized by the ITU-T as ITU-T Z.140
TTCN-3 designed for interface, service, protocol, ..., system testing

System Under Test (SUT)

Port

TTCN-3 Test Case

Port.send(Stimulus)  Port.receive(Response)

• Assignment of a Test Verdict

© DIAMONDS Consortium 2010-2013
TTCN-3 static and dynamic test configurations
## Major language elements of TTCN-3 notation (1)

<table>
<thead>
<tr>
<th>Module Definitions</th>
<th>Imports</th>
<th>Importing definitions from other modules defined in TTCN-3 or other languages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Types</td>
<td>User defined data types (messages, PDUs, information elements, …)</td>
</tr>
<tr>
<td></td>
<td>Test Data</td>
<td>Test data transmitted/expected during test execution (templates, values)</td>
</tr>
<tr>
<td></td>
<td>Test Configuration</td>
<td>Definition of the test components and communication ports</td>
</tr>
</tbody>
</table>
Major language elements of TTCN-3 notation (2)

Type definitions:
- boolean, integer, float, bitstring, charstring, octectstring, hexstring
- record, set, enumeration, union

Programming constructs:
- message: send/receive
- procedure: call/getcall, reply/getreply, raise/catch
- if-then-else, loops: for, while, do-while
- functions, alternatives
- component/port/timer control

Predefined functions:
- type conversion, lengthof (string), sizeof (records), ..

Overview: e.g. TTCN-3 Quick Reference Card

xxx → relevant for data fuzzing
Define test data for information sent to and received from the SUT

- Type-based templates
- Signature templates
- Global and local templates
- Inline templates
- Modified templates
- Template matching mechanisms
Main TTCN-3 Matching Mechanisms

Syntactical Structure

```
cmit |  
"(" { TemplateInstance [",",] } ")"
complement "(" { TemplateInstance [",",] } ")"
"?" |  
"*" |  
"(" ( ConstantExpression | -infinity ) "." ( ConstantExpression | infinity ) ")"
superset "(" { ConstantExpression [",",] } ")" |  
subset "(" { ConstantExpression [",",] } ")" |  
pattern Cstring
```

Syntactical Structure

```
length "(" ConstantExpression [ "." ( ConstantExpression | infinity ) ] ")" [ ifpresent ] |  
ifpresent
```

Syntactical Structure

```
template "(" ( cmit | present | value ) ")" Type
```
General Considerations

- Providing a **light-weight extension for TTCN-3** to maximize the usability of the extension for existing TTCN-3 users
  - Matching mechanisms can be used with a new special mechanisms `fuzz(...)` for templates to be sent

- Supporting **repeatability of fuzz test cases** performed via TTCN-3
  - a **seed** can be used to support repeatability when generating fuzzed data
New Mechanism `fuzz(...)`

- `fuzz` can be used instead of values to define a list of values or templates
  - Only for templates to be sent
  - A single value will be selected when
    - sending a template or
    - invoking the `valueOf` operation
  - Has one or two arguments:
    1. shall be of the type return by `fuzz` and can be a concrete value or a matching mechanism
    2. optional `charstring` denoting a fuzzing generator or operator.
  - Loop to repeat a test case with different fuzz data to be realized in the module control part
New Mechanism fuzz(…)

Example:

```plaintext
type record myData {
    charstring field1, octetstring field2, integer field3
}
template myType myData := {
    field1 := fuzz(?, UnicodeUtf8ThreeCharGenerator"),
    field2 := '12AB'O,
    field3 := fuzz(123)
}
```

- field1 retrieves a value that is generated by `UnicodeUtf8ThreeCharGenerator`
- field3 retrieves a value generated by applying appropriate fuzzing operators to the number 123
Maintaining Repeatability of Test Cases

- Fuzz data is often **randomly generated** which interferes with repeatability of test cases.

- In order to maintain **repeatability of test cases**, a seed is used for randomly generated or mutated fuzz testing values:
  - The seed is optional.
  - It can be read and set using two predefined functions `getSeed` and `setSeed`. 
Data Fuzzing Approach

• Make traditional data **fuzzing widely available**
  - allow an **easy integration** into tools
  - **without deep knowledge** about fuzz data generation
• Allow data fuzzing **without the need for**
  - **making familiar** with a specific fuzzing tool
  - **integrating further fuzzing tools** into the test process

• Approach:
  - don’t reinvent the wheel, **use the potential of existing fuzzing tools:**
    - Peach
    - Sulley
    - OWASP WebScarab
  - extract their **fuzzing generators and operators into a library**
    (reimplementation in Java)
Data Fuzzing Library with TTCN-3

- Allows **generation and mutation-based fuzzing**
- Platform independent: the library is implemented on **Java** running on many platforms
- Language independent: the library provides an **XML-based interface**
- Efficient: the user can decide
  - **which fuzzing operators** shall be used
    - allows adjusting of the library to fit specific requirements, e.g. only generate fuzz testing values based on valid values (for instance taken from functional testing)
  - **how many values** shall be generated by the library
    - avoids transmission of billions of values at once
- Communicative: the fuzzing **library tells you which fuzzing operators are used, so you can request more values only from this fuzzing operator** if one fuzz testing value generated by a specific fuzzing operator found bugs in your software
Tool Architecture

XML
XML schema file for requests to (fuzzingRequest.xsd) and responses from the library (fuzzingResponse.xsd)

Data Fuzzing Library

Fuzzing Generators/Operators
- Bad Strings Generator
- String Repetition Operator
- RegEx Expander

Processing
- String Request
- String Response
- Number Request
- Number Response

Fuzz Testing Values of Type
- String
- Number
# Generators and Operators

<table>
<thead>
<tr>
<th>Generators</th>
<th>Peach</th>
<th>Sulley</th>
<th>OWASP WebScarab</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringCaseMutator</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UnicodeStringsMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UnicodeBomMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UnicodeBadUtf8Mutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UnicodeUtf8ThreeCharMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StringMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PathMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HostnameMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FilenameMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BadIpAddress</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BadNumbers</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BadDate</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BadTime</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FiniteRandomNumbersMutator</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Repetition</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL Injection</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Command Injection</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format String</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delimiter</td>
<td>G</td>
<td></td>
<td>G/O</td>
</tr>
<tr>
<td>RegExExpander</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical Edge Case Mutator</td>
<td>G</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>Numerical Variance Mutator</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LongString</td>
<td></td>
<td></td>
<td>G</td>
</tr>
</tbody>
</table>
Results yet to come ...
Summary

• Security testing
  - Is needed and challenging
  - Requires additional approaches for MBT
  - Ongoing work

• Systematic and automated security testing
  - **Model-based fuzzing** using models on the data and behaviour that is being mutated in such a way that the number of test cases are significantly reduced – our approach is to fuzz functional tests
  - **Risk-oriented testing** uses risk analysis results for test identification, test selection and test assessment to prioritize and optimize the test process – our approach is to extend CORAS and combine it with Fokus!MBT
  - **Security test pattern** catalogue capturing expert knowledge on what to test in which context (kind of system, security goal) and allow the reuse of this knowledge within a slightly different context – our approach is to relate vulnerabilities and test patterns
Selected References

Thank you for your attention!

Further Questions?
Contact

Prof. Dr.-Ing. Ina Schieferdecker

Phone:    +49 30 34 63 7241
Mobile:   +49 175 260 30 21
Email:    ina.schieferdecker@fokus.fraunhofer.de

**FOKUS**
Fraunhofer Institute for Open Communication Systems FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany

Tel:       +49 (30) 34 63 – 7000
Fax:       +49 (30) 34 63 – 8000
Web:       www.fokus.fraunhofer.de

© DIAMONDS Consortium 2010-2013