Assessing the Robustness and Security of Web Services
State-of-the-art and Research Opportunities

Marco Vieira
mvieira@dei.uc.pt
University of Coimbra – Portugal

Outline
- Part #1: Robustness Testing in Web Services
- Part #2: Detecting Security Vulnerabilities in WS
- Part #3: Open Discussion

#1: Robustness Testing in Web Services
- What is software robustness?
  - Robustness testing
  - Web Services robustness testing
  - Case studies
  - Research challenges and opportunities

#2: Detecting Security Vulnerabilities in WS
- What are vulnerabilities?
  - Vulnerabilities detection approaches and tools
  - Vulnerabilities in public Web Services
  - How effective are vulnerabilities detection tools?
  - Can we do better?
  - Research challenges and opportunities

What is software robustness?
- Software is said robust if it performs well under unusual conditions
  - Conditions that stress designers' assumptions
- Software is typically buggy and fragile
  - i.e., contains errors
- Large because programs are usually too big and too complex
  - It is difficult to discover and eliminate all the bugs
  - This is especially true with regard to subtle errors that only appear in unusual circumstances
Software bugs

Studies show a clear prevalence of software faults as the root cause of computer failures
- [Kalyanakrishnam99]
- [Lee95]

Due to the complexity of today’s software, the weight of software faults will tend to increase

Interface faults are related to problems in the interaction among components/modules
- Particularly relevant when considering component based development and service based SW development

Robustness testing?

Characterize the behavior of a system in presence of erroneous input conditions

- Robustness tests stimulate the system in a way that triggers internal errors

And in that way expose both programming and design errors in the error detection or recovery mechanisms

- Systems can be differentiated according to the number of errors uncovered

Two most important robustness testing tools:
- Ballista [Koopman99]
- MAFALDA [Arlat99]

Classical robustness testing approach

Specified behavior | Input space | Response space
--- | --- | ---
Should work | Valid input | Robust operation
Unspecified | Invalid input | Interface faults
Should return error | | Failure

Has been mainly applied to operating systems and kernels

- Ballista [Koopman99]
- Mafalda [Arlat99]
- Xception [Carreira98]

Classical robustness upgrading approach

Specified behavior | Input space | Response space
--- | --- | ---
Should work | Valid input | Robust operation
Unspecified | Invalid input | Interface faults
Should return error | | Failure

Wrapper

Has been mainly applied to operating systems and kernels

- Ballista [Koopman99]
- Mafalda [Arlat99]
- Xception [Carreira98]

Ballista [Koopman99] (1)

Tool that combines software testing and fault injection techniques

- The main goal is to test software components for robustness

- Focusing specially operating systems

- Tests use combinations of exceptional and acceptable input values

- Parameter values are extracted randomly from a database of predefined tests

- For each parameter a set of values of a certain data type is associated

Ballista [Koopman99] (2)

Robustness is classified using the CRASH scale:

- Catastrophic
  - OS becomes corrupted or the machine crashes or reboots

- Restart
  - Application hangs and must be terminated by force

- Abort
  - Abnormal termination of the application

- Silent
  - No error is indicated by the OS on an operation that cannot be performed

- Hindering
  - The error code returned is not correct
MAFALDA [Arlat99]

- Microkernel Assessment by Fault injection Analysis and Design Aid
- Allows the characterization of the behavior of microkernels in the presence of faults
- Supports fault injection into the parameters of system calls and into the memory segments
  - In what concerns to robustness testing, only the fault injection into the parameters of system calls is relevant

Questions?

The problem

Web Services must provide a robust service to the client applications

- Development tools lack mechanisms to:
  - Characterize the robustness of Web Services code
  - Compare the robustness of alternative Web Services

Robustness Testing in WS

Testing Approach

Web Services robustness testing

- Erroneous Web Services call parameters
  - Generated using a set of predefined rules
  - Based on the data types of each parameter
  - Injected during the Web Services execution

GetWeather (city, day) → GetWeather ("Bangalore", null)

Key components needed:
- Workload
- Robustness tests
- Failure modes classification

The need

An approach to support the evaluation of the robustness of Web Services code

- Useful for programmers and testers:
  - Help providers evaluating the robustness of their Web Services code before deployment
  - Help consumers picking the Web Services that best fit their requirements by comparing different alternatives
  - Help providers and consumers identifying the need for Web Service wrappers
Key components

- **Workload**
  - Work that the service must perform during the benchmark run

- **Robustness tests**
  - Faultload consisting of a set of invalid call parameters
  - Applied to the target services to expose robustness problems

- **Failure modes classification**
  - Characterize the behavior of the service while executing the workload in the presence of the robustness tests

Additional components

- **Experimental setup**
  - Describes the setup required to run the tests
  - Typically includes two key elements:
    - Tests Target (TT) that represents the service that the benchmark user wants to characterize
    - The Tests management system (TMS) that is in charge of managing all the experiments
  - The TMS performs three key tasks
    - Experiments control
    - Workload emulation
    - Robustness test execution
  - The goal is to make it a completely automated process

- **Testing procedure**

Testing procedure

- **Description of the steps and rules that must be followed during tests execution**
- Closely related to the class of services being targeted, but includes:
  - Tests preparation
    - Analysis of the services under testing to gather information
    - Workload generation
  - Tests execution
    - Execution of the workload to understand the service behavior
    - Execution of the robustness tests to trigger faulty behaviors
  - Failure modes classification
    - Robustness problems identification

Workload

- **Defines the work that the system must perform during the benchmark execution**
- Three different types of workload can be considered:
  - Real workloads
  - Realistic workloads
  - Synthetic workloads

Robustness tests

- **Applied during the execution of the workload**
- **Try to activate robustness issues**
- **Involves parameter tampering at some level**
- **A set of rules must be defined for parameter mutation**
  - Must focus limit conditions that typically represent difficult validation aspects
  - Which are normally the source of robustness problems

Rules for parameter mutation (1)

- **Null and empty values**
  - e.g., null string, empty string

- **Valid values with special characteristics**
  - e.g., nonprintable characters in strings, valid dates by the end of the millennium

- **Invalid values with special characteristics**
  - e.g., invalid dates with using different formats

- **Maximum and minimum valid values in the domain**
  - e.g., maximum value valid for the parameter, minimum value valid for the parameter
Rules for parameter mutation (2)

- Values exceeding the maximum and minimum valid values in the domain
  - e.g., maximum value valid for the parameter plus one
- Values that cause data type overflow
  - e.g., add characters to overflow string and replace by maximum number valid for the type plus one
- These rules are similar to the ones used in classical robustness testing
- Each rule is a mutation that is applied at testing time to an incoming parameter

Example of parameters robustness tests

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Replace by null value</td>
</tr>
<tr>
<td></td>
<td>Replace by empty string</td>
</tr>
<tr>
<td></td>
<td>Replace by predefined string</td>
</tr>
<tr>
<td></td>
<td>Replace by string with nonprintable characters</td>
</tr>
<tr>
<td></td>
<td>Add nonprintable characters to the string</td>
</tr>
<tr>
<td></td>
<td>Replace by alphanumeric string</td>
</tr>
<tr>
<td></td>
<td>Add characters to overflow max size</td>
</tr>
<tr>
<td>Number</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td></td>
</tr>
</tbody>
</table>

Failure modes classification

- Services robustness can be classified according to the severity of the exposed failures
- There are several ways to classify failure modes
- An alternative is to use CRASH scale \([15]\) as basis for services characterization
  - The scale must be tailored according to the specificities of the class of services targeted

Tailoring the CRASH scale

- Example:
  - Catastrophic
    - The service supplier (i.e., the underlying middleware) becomes corrupted, or the server or the OS crashes or reboots
  - Restart
    - The service supplier becomes unresponsive and restart must be forced
  - Abort:
    - Abnormal termination when executing the service
    - e.g., unexpected exception is thrown
  - Silent:
    - No error is indicated by the service on an operation that cannot be concluded or is concluded in an abnormal way
  - Hindering:
    - The returned error code is incorrect

Questions?
**Case Studies**

- [Siblini05]
- [Xu05]
- [Laranjeiro08]
- [Vieira07]

**[Siblini05]**
- One of the first examples of robustness testing applied to Web Services
- Proposes a technique to test Web Services using parameter mutation analysis
- The WSDL document is parsed and mutation operators are applied to it
  - Resulting in several mutated documents that will be used to test the service
- Very good approach
  - But, the parameter mutation operators are limited!

**[Xu05]**
- Similar approach
- Represents a more complete study
- Coupling with the XML (eXtensible Markup Language) technology
  - Invalidates any kind of test generalization
  - i.e., it does not apply to other technologies as it is tightly coupled to XML

**[Laranjeiro08]**
- State-of-the-art on JMS robustness testing
- Practical way to evaluate the robustness of JMS middleware
  - Based on a fault injection approach
  - Set of robustness tests (i.e., invalid parameters)
- Help providers evaluate robustness of their implementations
- Help clients to evaluate and select alternative JMS middleware platforms

**[Vieira07]**
- State-of-the art on SOAP Web Services robustness testing
  - **Key components:**
    - Workload, robustness tests, failure modes classification
  - **Main steps:**
    - Preparing the tests
    - Executing the tests
    - Characterizing the Web Services
  - **Experimental evaluation:**
    - 100 publicly available Web Services
    - Two different implementations of the TPC-App services

**Preparing the tests**

- Obtain Web Service definitions
  - List of operations
  - Parameters
  - Data types
  - Domains
  - The WSDL file is processed automatically to obtain the required information
  - The domain for each parameter cannot be deduced from the WSDL description
  - Must be provided by the benchmark user
Workload

- A workload is needed to exercise each operation of the Web Service
- A generic workload that fits all Web Services is not feasible
  - We need to generate a workload for each Web Service tested
- Workload generation:
  - User defined workload
  - Random workload

Executing the tests

- Two steps:
  - Step 1: workload is run without considering invalid call parameters
  - Step 2: workload is run in the presence of invalid call parameters (robustness tests)

Parameters values mutation

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Replace by null value</td>
</tr>
<tr>
<td></td>
<td>Replace by empty string</td>
</tr>
<tr>
<td></td>
<td>Replace by predefined string</td>
</tr>
<tr>
<td></td>
<td>Replace by string with nonprintable characters</td>
</tr>
<tr>
<td></td>
<td>Add nonprintable characters to the string</td>
</tr>
<tr>
<td></td>
<td>Replace by alphanumeric string</td>
</tr>
<tr>
<td></td>
<td>Add characters to overflow max size</td>
</tr>
<tr>
<td>Number</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td></td>
</tr>
</tbody>
</table>

Characterizing the Web Services (1)

- wsCRASH scale:
  - Catastrophic: Application server crashes or reboots
  - Restart: Web Service execution hangs
  - Abort: Abnormal termination of the Web Service
  - Silent: After a timeout no error is indicated
  - Hindering: Incorrect error code or delayed response
- Consumers cannot distinguish between a catastrophic and a restart failure
  - Do not have access to the server where the service is running

Characterizing the Web Services (2)

- wsAS scale
  - Simplification for consumers
    - Abort: Abnormal termination of the Web Service
    - Silent: After a timeout no error is indicated

Complement the characterization

- Complement the failure modes with the analysis of the Web Service behavior
  - Understand the source of the failures
- e.g. of some sources for robustness problems:
  - Database access operations
  - Arithmetic operations
  - Null References
- Source of the failures depends on the WS
  - No predefined classification
  - An excellent alternative is to use the Orthogonal Defect Classification (ODC)
Orthogonal Defect Classification (ODC)

- [Chillarege92]
- Important contribution to promote the collection and study of observed faults
- Classification schema for software faults
  - Defects are classified into non-overlapping attributes
  - Used as a source of information to understand and improve the product and the development process
- Intended to provide feedback on to the development process
  - But, it also provides a useful defect classification concerning the problem of the emulation of SW faults

The wsrbench tool...

- Implements the Web Services testing approach presented before
- Available online
  - http://wsrbench.dei.uc.pt

Experimental evaluation

- Two different scenarios
  - Evaluation of Web Services publicly available in the Internet
  - Comparison of two different implementations of the TPC-App Web Services
- Three key questions:
  - Can robustness benchmarking be used by providers and consumers to test Web Services?
  - Can robustness benchmarking be used to improve the robustness of Web Services code?
  - Can the benchmark be used to compare different implementations of a given Web Service?

Public Web Services evaluation (1)

- Tested 100 Web Services publicly available
  - The majority is listed at http://xmethods.net/
- Several technologies:
  - .NET (83%)
  - Visual Dataflex (6%)
  - Axis (5%)
  - MS Soap (2%)
  - SOAP/AM (1%)
  - Visual Fox-Pro (1%)
  - 4D (1%)
  - Visual Basic (1%)

Public Web Services evaluation (2)

- Web Services are owned by different relevant parties
  - E.g., Microsoft and Xara
- Some Web Services implement the same functionality
  - E.g., Text Disguise and Free Captcha Service
- Some Web Services are used in real businesses in the field
  - E.g., Portuguese Postal Office Orders Cost and UPS Online Tracking Web-Service

Sources of robustness problems

- Null references
  - Null pointers that reflect none or poor input validation
- Database access operations
  - Exceptional behavior caused by invalid SQL commands
- Arithmetic operations
  - Typically data type overflow in numeric operations
- Conversion problems
  - Class cast exceptions or numeric conversion problems
- Other causes
  - Arguments out of range, invalid characters, etc
Results overview

Some details on the results...

- About 35% of the Web Services tested presented abort failures
  - 30% null references
  - 30% SQL problems
  - 13% conversion problems
  - 7% arithmetic problems
  - 20% others

- No Silent failures were observed

- A huge percentage of Web Services revealed robustness problems related to null references
  - Most of the services assume non-null input parameters

More details on the results...

- Database related failures are mostly related to null parameters
  - This is more than a simple SQL construction error
  - Provider does not validate the SQL inputs, which opens a door for SQL injection attacks...

- Arithmetic problems were exposed using the maximum value for a numeric parameter
  - Resulted in several overflows in arithmetic operations

Are Web Services robust?

- Faulty Web Services are frequently deployed
  - Unacceptable situation for providers and also for clients

- Robustness problems may lead to security issues
  - Some services are vulnerable to SQL injection attacks

- Robustness testing is essential when developing an infrastructure for Web Services
  - Test and fix Web Services code
  - Select alternative Web Services
  - Build wrappers to mitigate robustness problems

Questions?

Robustness Testing in WS
Research Challenges and Opportunities
Challenges and opportunities (1)

- Fact: Robustness testing seems to be effective when applied to Web Services
- Workload generation
  - Increase representativeness of the workload
  - Guarantee high coverage
- Focus on different types of services
  - Other technologies
- Automated identification of problems
  - In most cases human intervention is needed

Challenges and opportunities (2)

- Web Services inputs are not the ones provided during invocation
  - Responses from other services (including databases, gateways, etc.) may also be invalid and cause robustness problems
  - How to deal with these?
- Fact: Web Services are typically deployed with robustness problems
- How to improve the current situation?

Challenges and opportunities (3)

- Improve the software development process?
  - More effective testing?
  - More effective code reviews and inspections?
    - Use targeted checklists?
- Provide tools for the automated removal of robustness problems?
  - How to specify input domains?
    - These are not in the WSDL file
- How to take advantage of Web Services diversity to change the current situation?

Questions?

Detecting Security Vulnerabilities in WS
Principles

Outline revisited

- What are vulnerabilities?
- Vulnerabilities detection approaches and tools
- Vulnerabilities in public Web Services
- How effective are vulnerabilities detection tools?
- Can we do better?
- Research challenges and opportunities
What is a vulnerability?

A vulnerability is a weakness that may allow attackers to gain access to the system or info

- Complexity
- Password and privileges management flaws
- Operating system design flaws
- Software bugs
- Unchecked user input

There are many causes:

- Complexity
- Password and privileges management flaws
- Operating system design flaws
- Software bugs
- Unchecked user input

SQL Injection

- It is possible to alter the construction of backend SQL statements
- An attacker can read or modify database data and
- In some cases, execute database administration operations or commands in the system

XPath Injection

- It is possible to modify an XPath query to be parsed in a way differing from the programmer's intention
- Attackers may gain access to information in XML documents

Vulnerabilities in Web Applications (1)

Vulnerabilities in Web Applications (2)

Code Execution

- It is possible to manipulate the application inputs to trigger server-side code execution
- An attacker can exploit this vulnerability to execute malicious code in the server machine

Buffer Overflow

- It is possible to manipulate inputs in such a way that causes buffer allocation problems
- Including overwriting of parts of the memory
- An attacker can exploit this causing Denial of Service
- Or, in worst cases, alter application flow and force unintended actions

Vulnerabilities in Web Applications (3)

Username/Password Disclosure

- A response contains information related to usernames and/or passwords
- An attacker can use this information to get access to private data

Server Path Disclosure

- A response contains a fully qualified path name to the root of the server storage system
- An attacker can use this info to discover the server file system structure and devise other security attacks

Examples of SQL Injection vulnerability

```java
public String auth(String login, String pass) throw SQLException {
    String sql = "SELECT * FROM users WHERE username='" + login + "' AND password='" + pass + "'";
    ResultSet rs = statement.executeQuery(sql);
    ...
}
```

```java
public void delete(String str) throw SQLException{
    String sql = "DELETE FROM table WHERE id='" + str + "'";
    statement.executeUpdate(sql);
}
```

Examples of SQL Injection vulnerability

```java
" OR 1=1 --
SELECT * FROM users WHERE username='" OR 1=1 -- AND "
password='" OR 1=1 --
DELETE FROM table WHERE id='" OR 1=1 --
WHERE " OR 1=1 --
```

Questions?
Detecting Security Vulnerabilities in WS

Vulnerabilities detection approaches and tools

Are Web Services vulnerable?

- Web Services are widely exposed
- Any existing vulnerability will most probably be uncovered/exploited
- Both providers and consumers need to assess Web Services' security

Web Services security

- Security threats
  - Hackers are moving their focus to applications’ code
  - Traditional security mechanisms (Firewall, IDS, encryption) cannot mitigate these attacks
- Developers must
  - Apply best coding practices
  - Perform code analysis
    - Manual code analyses (reviews, inspections)
    - Automated static code analysis
  - Do tests
    - Manual penetration testing
    - Automated penetration testing (vulnerability scanners)

Penetration testing

- Widely used by developers
- Consists in stressing the application from the point of view of an attacker
  - "black-box" approach
  - Uses specific malicious inputs
    - e.g., for SQL Injection: `' or 1=1`
- Can be performed manually or automatically
- Does not require access to the source code (or bytecode)

Penetration testing tools

- Provide an automatic way to search for vulnerabilities
- Avoid the repetitive and tedious task of doing hundreds or even thousands of tests by hand
- Many tools available
  - Including commercial and open-source
- Different tools target different types of vulnerabilities
- The effectiveness of penetration testing tools is doubtful

Examples of penetration testing limitations

```java
public void operation(String str) {
    try {
        String sql = "DELETE FROM table" + "WHERE id=" + str + "";
        statement.executeUpdate(sql);
    } catch (SQLException se) {}
}
```

No return value and exceptions related with SQL mal-formation do not leak out to the invocator

```java
public String dumpDepositInfo(String str) {
    try {
        String path = "//DepositInfo/Deposit" + "[@accNum=" + str + ""]";
        return csvFromPath(path);
    } catch (XPathException e) {
        return null;
    }
}
```

This lack of output information
Examples of penetration testing tools

- WSDigger
- WSFuzzer
- HP WebInspect
- IBM Rational AppScan
- Acunetix Web Vulnerability Scanner
- Static code analysis

HP WebInspect (1)

- “Web application security testing and assessment for complex web applications
- Built on emerging Web 2.0 technologies
- Fast scanning capabilities, broad security assessment coverage
- Accurate web application security scanning results”

HP WebInspect (2)

- Includes pioneering assessment technology
  - Including simultaneous crawl and audit (SCA) and concurrent application scanning
- Broad scanning capabilities
  - Targets many different types of vulnerabilities
- Can be applied for penetration testing in web-based applications
  - Including Web Services

IBM Rational AppScan

- “Is a leading suite of automated Web application security and compliance assessment tools
- Scan for common application vulnerabilities”
- Suitable for users ranging from non-security experts to advanced users
- Supports extensions for customized scanning environments
- Can be used for security testing in web applications, including Web Services

Acunetix Web Vulnerability Scanner

- “Is an automated web application security testing tool
- Audits your web applications by checking for exploitable hacking vulnerabilities”
- Broad scanning capabilities
  - Targets many different types of vulnerabilities
- Can be applied for security testing in web applications in general
  - Including Web Services

Static code analysis

- “white-box” approach
- Consists in analyzing the source code of the application
  - Without execution it
- Looks for potential vulnerabilities
  - Among other types of software defects
- Can be performed manually or automatically
- Does not require access to the source code (or bytecode)
Static code analysis tools

- Analyze the code without actually executing it
- The analysis varies depending on the tool sophistication
  - Ranging from tools that consider only individual statements and declarations
  - To others that consider the complete code
- Have other usages
  - e.g., model checking and data flow analysis
- These tools provide an automatic way for highlighting possible coding errors

Examples of static code analysis tools

- FindBugs
- Yasca (Yet Another Source Code Analyzer)
- IntelliJ IDEA
- ...

Examples of static analysis limitations

- Analyzers identify the vulnerability because the SQL query is a non-constant string
- Depending on the complexity of csvFromPath method A static analysis tool may not be able to find the vulnerability

FindBugs

- "A program which uses static analysis to look for bugs in Java code"
- It is able to scan the bytecode of Java applications
- Detects, among other problems, security issues
- It is one of the most used tools for static code analysis

Yasca (Yet Another Source Code Analyzer)

- "A framework for conducting source code analyses"
- Wide range of programming languages, including java.

Yasca includes two components:
  - A framework for conducting source code analyses
  - An implementation of that framework that allows integration with other static code analyzers
    - e.g., FindBugs, PMD, Jlint

IntelliJ IDEA

- A commercial tool that provides a powerful IDE for Java development
- Includes "inspection gadgets" plug-ins with automated code inspection functionalities
- IntelliJ IDEA is able to detect security issues in java source code
Detecting Security Vulnerabilities in WS
Vulnerabilities in public Web Services

Experimental study
- [Vieira09]
- Question: Are Web Services vulnerable?
- Apply leading commercial scanners in public Web Services
- 300 Web Services tested
  - Randomly selected
- 4 Scanners used
  - Including two different versions of a brand

Experimental procedure
- Preparation
  - Select services and scanners
- Execution
  - Test the services using the scanners
- Verification
  - Identify false positives
- Analysis
  - Analysis and systematization of results

Scanners
- HP WebInspect™
- IBM Rational Application Test
- Web Vulnerability Scanner

Vulnerabilities Found
- SQL injection
- XPath Injection
- Code Execution
- Possible Parameter Based Buffer Overflow
- Possible Username or Password Disclosure
- Possible Server Path Disclosure
### Overall results analysis

<table>
<thead>
<tr>
<th>Vulnerability Types</th>
<th>VS1.1</th>
<th>VS1.2</th>
<th>VS1.3</th>
<th>VS2</th>
<th>VS3</th>
</tr>
</thead>
<tbody>
<tr>
<td># Vuln. # WS # Vuln. # WS # Vuln. # WS # Vuln. # WS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL Injection</td>
<td>16</td>
<td>21</td>
<td>16</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>XPath Injection</td>
<td>21</td>
<td>10</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code Execution</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Possible Remarks Based</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Possible Username or</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Password Disclosure</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Possible Server Path</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Disclosure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>228</td>
<td>40</td>
<td>236</td>
<td>40</td>
<td>5</td>
</tr>
</tbody>
</table>

### SQL Injection

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 27 Vulns, 27 Vuln.
- VS1.3: 171 Vulns, 171 Vuln.
- VS2: 24 Vulns, 24 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 21 Vulns, 21 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.

- VS1.1: 19 Vulns, 19 Vuln.
- VS1.2: 24 Vulns, 24 Vuln.
- VS1.3: 5 Vulns, 5 Vuln.
- VS2: 5 Vulns, 5 Vuln.
- VS3: 5 Vulns, 5 Vuln.
False positives examination

- False positive when
  - The error/answer obtained is related to an application robustness problem.
  - The same problem occurs when the service is executed with valid inputs

- Confirmed vulnerability when
  - It is possible to observe that a SQL command was invalidated by the "injected" values
  - The "injected" values lead to exceptions raised by the database server
  - It is possible to access unauthorized resources

False positives examination

- False positive when
  - The error/answer obtained is related to an application robustness problem.
  - The same problem occurs when the service is executed with valid inputs

- Confirmed vulnerability when
  - It is possible to observe that a SQL command was invalidated by the "injected" values
  - The "injected" values lead to exceptions raised by the database server
  - It is possible to access unauthorized resources

False positives results

![False positives results chart]

<table>
<thead>
<tr>
<th>VS1.1</th>
<th>VS1.2</th>
<th>VS2</th>
<th>VS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>166</td>
<td>71</td>
<td>26</td>
</tr>
</tbody>
</table>

- False Positives 40%
- Doubtful 37%
- Confirmed Vulnerabilities 11.6%

SQL Injection without False Positives

VS1.1: 142
VS1.2: 127

SQL Injection without False Positives

VS1.1: 103
VS1.2: 24
VS3: 2

SQL Injection without False Positives

VS1.1: 102
VS1.2: 21
VS3: 3

SQL Injection without False Positives

VS1.1: 3
VS1.2: 15
VS3: 15
**SQL Injection without False Positives**

- VS1.1
  - 1
  - 2
  - 3
  - 102
- VS1.2
  - 1
  - 15
- VS2
  - 21
  - 3
- VS3
  - 1

**Coverage analysis**

- Real number of vulnerabilities unavailable
- It is possible to make a comparative analysis
- Overestimated Coverage values!!

<table>
<thead>
<tr>
<th>Scanner</th>
<th># SQL Injection Vulnerabilities</th>
<th>Coverage %</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS1.1</td>
<td>130</td>
<td>87.2%</td>
</tr>
<tr>
<td>VS1.2</td>
<td>142</td>
<td>95.3%</td>
</tr>
<tr>
<td>VS2</td>
<td>25</td>
<td>16.8%</td>
</tr>
<tr>
<td>VS3</td>
<td>26</td>
<td>17.4%</td>
</tr>
<tr>
<td>Total</td>
<td>149</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Common vulnerabilities**

- SQL Injection (149)
- Possible Server Path Disclosure (16)
- XPath Injection (10)
- Code Execution (1)
- Possible Parameter Based Buffer Overflow (1)

**Are Web Services secure?**

- A large number of vulnerabilities was observed
- SQL Injection vulnerabilities are prevalent
- Selecting a scanner for Web Services seems to be a very difficult task
- Different scanners detect different types of vulnerabilities
- High false positives rates
- Low coverage rates
- How effective are vulnerabilities detection tools?

**Questions?**

**Detecting Security Vulnerabilities in WS**

How effective are vulnerabilities detection tools?
Experimental study [Antunes09a]

- Evaluate several automatic penetration testing tools and static analysis tools
  - In a controlled environment
- Focus on two key measures of interest:
  - Coverage
    - Portrays the percentage of existing vulnerabilities that are detected by a given tool
  - False positives rate
    - Represents the number of reported vulnerabilities that in fact do not exist
- Target only SQL Injection vulnerabilities
  - Extremely relevant in Web Services

Steps

- Preparation
  - Select the penetration testers and static code analyzers
  - Select the Web Services to be considered
- Execution
  - Use the tools to identify potential vulnerabilities
- Verification
  - Perform manual verification to confirm that the vulnerabilities identified by the tools do exist
    - i.e., are not false positives
- Analysis
  - Analyze the results obtained and systematize the lessons learned

Web Services tested

- Eight Web Services
  - A total of 25 operations
- Four of the services are based on the TPC-App performance benchmark
- Four other services have been adapted from code publicly available on the Internet
- Implemented in Java and use a relational database

Web Services characterization

<table>
<thead>
<tr>
<th>Service</th>
<th>Short Description</th>
<th>#Op</th>
<th>LA(C)</th>
<th>LA(Op)</th>
<th>Avg. C.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductDetail</td>
<td>Get details about a product</td>
<td>1</td>
<td>105</td>
<td>105.0</td>
<td>6.0</td>
</tr>
<tr>
<td>NewProduct</td>
<td>Add new product to the database</td>
<td>1</td>
<td>126</td>
<td>126.0</td>
<td>6.0</td>
</tr>
<tr>
<td>NewCustomer</td>
<td>Add new customer to the database</td>
<td>1</td>
<td>194</td>
<td>194.0</td>
<td>9.0</td>
</tr>
<tr>
<td>ChangePaymentMethod</td>
<td>Change customer’s payment method</td>
<td>1</td>
<td>97</td>
<td>97.0</td>
<td>11.0</td>
</tr>
<tr>
<td>JamesSmith</td>
<td>Manages personal data about students</td>
<td>5</td>
<td>270</td>
<td>54.0</td>
<td>6.0</td>
</tr>
<tr>
<td>PhoneBill</td>
<td>Place a phone bill</td>
<td>5</td>
<td>132</td>
<td>26.4</td>
<td>2.8</td>
</tr>
<tr>
<td>Bank</td>
<td>Manages bank operations</td>
<td>5</td>
<td>175</td>
<td>35.0</td>
<td>3.4</td>
</tr>
<tr>
<td>Bank1</td>
<td>Manages bank operations (different from the Bank service)</td>
<td>6</td>
<td>377</td>
<td>62.8</td>
<td>9.0</td>
</tr>
</tbody>
</table>

Tools studied

- Penetration testing
  - HP WebInspect
  - IBM Rational AppScan
  - Acunetix Web Vulnerability Scanner
- Static code analysis
  - FindBugs
  - Yasca
  - IntelliJ IDEA
- Decided not to mention the brand of the tools to assure neutrality
  - VS1, VS2, VS3 (without any order in particular)
  - SA1, SA2, SA3 (without any order in particular)

Tools and environment configuration

- Penetration-testing
  - Underlying database restored before each test
  - This avoids the cumulative effect of previous tests
  - Guarantees that all the tools started the service testing in a consistent state
- If allowed by the testing tool, information about the domain of each parameter was provided
  - If the tool requires an exemplar invocation per operation, the exemplar respected the input domains of operation
  - All the tools in this situation used the same exemplar
- Static code analysis
  - Configured to fully analyze the services code
  - For the analyzers that use binary code, the deployment-ready version was used
Web Services manual inspection

- It is essential to correctly identify the vulnerabilities that exist in the services code
- A team of experts was invited to review the source code looking for vulnerabilities
  - False positives were eliminated by cross-checking the vulnerabilities identified by different people
- A key difficulty is that different tools report (and count) vulnerabilities in different ways
  - Penetration testing: a vulnerability for each vulnerable parameter
  - Static analysis: a vulnerability for each vulnerable line in the service code

Vulnerabilities found

<table>
<thead>
<tr>
<th>Service</th>
<th>#Vuln. Inputs</th>
<th>#Vuln. Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductDetail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NewProducts</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NewCustomer</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>ChangePaymentMethod</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JamesSmith</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>PhoneDir</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Bank</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Bank3</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>61</strong></td>
<td><strong>28</strong></td>
</tr>
</tbody>
</table>

Penetration testing results

- **Tool**
  - VS1: 14.0%
  - VS2: 4.0%
  - VS3: 0.0%
  - VS4: 0.0%

- **Experts**
  - VS1: 1/22 = 4.5%
  - VS2: 0/62 = 0.0%
  - VS3: 63/63 = 100.0%
  - VS4: 0/63 = 0.0%

Static code analysis results

- **Tool**
  - SA1: 23.3%
  - SA2: 26.3%
  - SA3: 26.7%

Penetration testing vs Static analysis (1)

- **Coverage**
  - VS1: 23%
  - VS2: 28%
  - VS3: 11%
  - VS4: 28%
  - SA1: 23%
  - SA2: 28%
  - SA3: 11%

Penetration testing vs Static analysis (2)

- **False positives**
  - VS1: 2%
  - VS2: 10%
  - VS3: 6%
  - VS4: 28%
  - SA1: 23%
  - SA2: 28%
  - SA3: 11%
Key observations

- The coverage of static code analysis is typically higher than that of penetration testing.
- False positives are a problem for both approaches.
  - But have more impact in the case of static analysis.
- Different tools report different vulnerabilities in the same piece of code.
  - Even tools implementing the same approach frequently.
- Poor results?
  - Can we do better?

Questions?

Detecting Security Vulnerabilities in WS

Can we do better?

Yes, we can!

- [Antunes09b]
  - Proposes a new penetration testing approach to detect SQL Injection vulnerabilities in Web Services code.
- Main improvements:
  - A representative workload to exercise the services and understand the expected behavior.
  - A broader set of attacks.
  - Well defined rules to analyze the service’s responses.
  - To improve coverage and remove false positives.
  - Completely automatic.

Execution steps

1. Prepare the tests
   1.1. Gather information about the Web Service’s operations, call parameters, data types, and input domains
   1.2. Generate the workload
2. Execute the tests
   2.1. Execute the workload to understand the expected behavior of the service in the absence of attacks
   2.2. Perform the attacks to trigger faulty behaviors and disclose SQL Injection vulnerabilities
3. Analyze the responses to detect and confirm the vulnerabilities

Prepare the tests: Gather information

- Web Service interfaces are described as a WSDL file.
- This file is processed automatically to obtain:
  - Operations
  - Call parameters
  - Data types
- The valid values for each parameter (i.e., input domains) have to be provided by the user.
Prepare the tests: generate the workload

- Two options:
  - User-defined workload
  - Random workload

- Random workload is generated automatically
  - Generate test values for each input parameter
  - Generate test calls for each operation
  - Select test calls for each operation
    - It may be unfeasible to use a workload based on all the test calls generated (e.g., due to time constraints)
    - It is up to the user to specify the size of this subset

Execute the tests: Configuration

- Penetration Testing Tool
- SOAP

Execute the tests: Type of attacks

- Examples:
  - SQL Injection Attack
    - "or 1=1"
    - "or exists (select * from employees"
    - "or name like %%
    - "or exist (select char(23)+23+23+23+23)"
    - UNION ALL SELECT

- A total of 137 types
  - The list can be continuously improved
  - Just add new attack patterns to a configuration file

Execute the tests: Attacks generation

- Mutation of the workload test calls
  - Valid values are replaced by malicious values

- Number of attacks can be extremely large, e.g.:
  - 3 operations with 5 parameters each
  - A workload with 25 test calls per operation
  - 137 attack types \( \Rightarrow \) 52500 attacks

- The tool allows specifying the number of test calls to be used for the attack load generation
  - The original test calls are ranked based on their ability to help us detecting vulnerabilities
  - e.g. test calls that lead to valid Web Service responses (i.e., no error) are in the top of the list

Analyze the responses

- W:
  - Valid call

- A:
  - Attack call

Experimental evaluation

- Web Services tested
  - 262 public Web Services

- Four steps:
  - Preparation: select a large set of Web Services.
  - Execution: use the vulnerability scanners to scan the services to identify potential vulnerabilities
  - Verification: perform manual testing to confirm that the vulnerabilities identified do exist
  - Analysis: analyze the results and compare the effectiveness of our tool to the commercial ones
Scanners

Raw results for public Web Services

After removing false positives...

Detection coverage

Can we do better?

Questions?
Detecting Security Vulnerabilities in WS
Research challenges and opportunities

Challenges and opportunities (1)
- Fact: The effectiveness of vulnerability detection tools is very low
- How to improve penetration testing?
  - Increase representativeness of the workload
  - Guarantee high coverage
  - Improve the attacks performed
  - Improve the vulnerability detection algorithms
- How to improve static analysis?
  - Include new vulnerable code patterns
  - How to identify those patterns?

Challenges and opportunities (2)
- Merge penetration testing and static code analysis techniques?
- Fact: Web Services are typically deployed with vulnerabilities
- How to improve the current situation?
  - Better tools for vulnerability detection
  - Approaches to automatically remove vulnerabilities
- What about attack detection?
  - Can we include in the Web Services attack detection mechanisms?

Challenges and opportunities (3)
- Improve the software development process?
  - More effective testing?
  - More effective code reviews and inspections?
    - Use targeted checklists?
- How to take advantage of Web Services diversity to change the current situation?

Questions?

Open Discussion
Your Time!
Key references (1)


Key references (2)


Slides available at:

http://eden.dei.uc.pt/~mvieira

Questions?

CMU/UC Master of Software Engineering

- Explicitly training graduates for:
  - Effectively managing teams, projects and processes
  - Creating architectures for large-scale systems
  - Implementing change inside of organizations
  - Becoming team leaders, project managers, software architects

- Program that top organizations send their engineers to do before becoming team and section leads

- Widely recognized as being the Number One Program in Software Engineering worldwide

Questions?

Change your life!

http://mse.dei.uc.pt
Thanks for your participation!

Marco Vieira  
Center for Informatics and Systems  
University of Coimbra  
mvieira@dei.uc.pt