Characterizing and understanding security risks through Security-Aware Mutation Testing of security configuration in RESTful APIs

Carlos Andres Delgado Saavedra
carlos.andres.delgado@correounivalle.edu.co
Advisors:
Jesus A. Aranda, PhD. Universidad del Valle
Gills Perrouin, PhD. Universite de Namur
James Ortiz. PhD, Universite de Namur

Universidad del Valle

UNIVERSITÉ DE NAMUR

AVISPA
AMBIENTES VISUALES DE PROGRAMACIÓN APLICATIVA

Overview

1. Introduction
2. Problem statement
3. Literature Review
4. Research Objectives
5. Methodology
6. Expected Results

Introduction

Introduction I

Background

RESTful APIs are essential for enabling communication between web applications and services, facilitating operations such as reading, updating, creating, and deleting data. However, the exchange of sensitive information introduces significant security risks.

Significance

Security challenges, such as misconfigurations and vulnerabilities in RESTful APIs, can lead to unauthorized access and data breaches. Testing mechanisms, like mutation testing, help address these challenges by evaluating the robustness of security configurations.

Introduction II

Key Idea

Mutation testing introduces deliberate changes (mutations) into code to assess the effectiveness of security tests. This research explores security-aware mutation operators to improve the security configuration of RESTful APIs.

Problem statement

Problem Statement I

Challenges in API Security

- RESTful APIs handle sensitive information, such as passwords and personal data.
- Common vulnerabilities include:
  - Authorization issues
  - Data encryption weaknesses
  - Security misconfigurations

Current Limitations

- Existing security tools often fail to uncover configuration-based vulnerabilities.
- High dependency on developers to manually ensure secure configurations.

Problem Statement II

Research Question

How can security-aware mutation operators be designed to improve the coverage of security testing for vulnerabilities in the configuration of security policies in RESTful APIs?

Hypothesis

Designing security-aware mutation operators for RESTful APIs can enhance the validation of security configuration policies, reducing the risk of vulnerabilities.

Literature Review

Literature Review I

Overview of Mutation Testing

- Introduced in 1972 as a method to evaluate software reliability by modifying code to create faults.
- Recent advancements include:
    - Mutation operators for specific languages like Java and Python.
    - Use of machine learning techniques to enhance fault detection.

Literature Review II

Security Testing in RESTful APIs

- Focuses on detecting vulnerabilities in API endpoints.
- Common testing methods:
  - Black-box and white-box testing
  - Penetration testing and property-based testing
- Challenges include managing various data formats (e.g., JSON, XML) and ensuring comprehensive test coverage.

Literature Review III

Q1

What are the existing mutation operators for security testing of restful APIs?

1. Test case generation
2. Source code based
3. Model based mutation testing

Literature Review IV

Q2

How effective are these mutation operators in detecting security vulnerabilities?

1. Test case generation operators are effective in detecting vulnerabilities and easy to generalize.
2. Model-based mutation depends of the abstraction level of the language.
3. Test case generation mutation operators are specific to the language, it is complex to generalize.

Literature Review V

Q3

What strategies are there for improving security practices in restful APIs?

1. CORS
2. Auth based authentication
3. Encryption of the query parameters

Literature Review VI

Q4

What elements define common security misconfigurations in restful APIs?

1. CORS missconfigurations
2. Bad configuration of the authentication, using leak credentials.
3. Not using encryption for the query parameters.

Literature Review VII

Gaps Identified

- Mutation operators are often language-specific, limiting general applicability.
- Insufficient focus on security-aware mutation testing for RESTful APIs.
- Lack of standardized frameworks for evaluating security tests.

Research Objectives

Research Objectives I

General Objective

- Develop a collection of security-aware mutation operators designed for evaluating the configuration of security policy files within RESTful APIs.

Research Objectives II

Specific Objectives

1. Identify the key elements of security policies in RESTful APIs.
2. Design a set of security-aware mutation operators for testing security policies.
3. Develop the security-aware mutation operators and integrate them into testing tools.
4. Evaluate the proposed operators against existing security testing frameworks, focusing on their effectiveness and coverage.

Research Objectives III

Expected Results

- A comprehensive set of mutation operators tailored for RESTful API security.
- Detailed reports on the performance of these operators compared to current tools.
- Contribution to the development of frameworks for automated security testing.

Methodology

Proposed Methodology I

The research is divided into four key phases:

1. **Systematic Literature Review:**
   - Identify existing vulnerabilities and security-aware mutation operators.
   - Analyze current tools and techniques used for testing RESTful APIs.
2. **Design of Mutation Operators:**
   - Define strategies to introduce vulnerabilities into code.
   - Specify and describe security-aware mutation operators.
3. **Development of Mutation Operators:**

Proposed Methodology II

- Implement operators in mutation testing tools (e.g., MutPy, MutMut).
- Refactor the implementation for efficiency and maintainability.

4. **Evaluation:**
    - Apply operators to case studies.
    - Measure coverage, fault detection, and mutation score.

Proposed Methodology III

Techniques Used

- **Test-Driven Development (TDD):** Ensure mutation operators function as intended.
- **Snowballing Methodology:** Review recent surveys and track relevant studies.
- **Evaluation Metrics:** Analyze coverage, redundancy, and effectiveness of the operators.

Expected Deliverables

- A set of validated mutation operators for RESTful API security testing.
- A framework for automating security-aware mutation testing.
- Reports detailing the findings and contributions.

Timeline I

The project timeline is structured over 24 months, covering the following phases:

1. **Systematic Literature Review:** Months 1-6
   - Review recent surveys and identify vulnerabilities in RESTful APIs.
   - Analyze security-aware mutation operators and existing tools.
2. **Design of Mutation Operators:** Months 7-12
   - Define strategies to introduce vulnerabilities.
   - Specify and describe the proposed mutation operators.
3. **Development of Mutation Operators:** Months 13-18

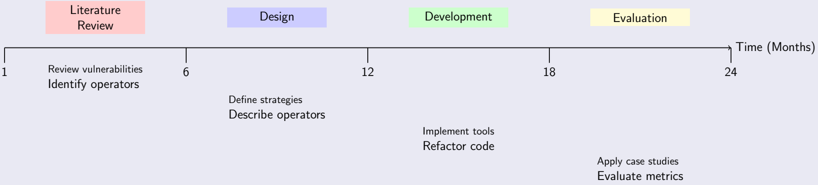
Timeline II

- Implement the mutation operators in selected testing tools.
- Refactor code to ensure efficiency and maintainability.

4. **Evaluation:** Months 19-24
   - Apply the mutation operators to case studies.
   - Evaluate their effectiveness using coverage and redundancy metrics.
   - Generate detailed reports and summarize findings.

Timeline III

Visual Timeline



Literature Review

Design

Development

Evaluation

Time (Months)

1 6 12 18 24

Review vulnerabilities
Identify operators

Define strategies
Describe operators

Implement tools
Refactor code

Apply case studies
Evaluate metrics

Figure 2: Timeline of Methodology

Expected Results

Expected Results I

Impact on RESTful API Security

- Provide a systematic approach for evaluating the robustness of RESTful API security configurations.
- Enhance the ability of developers to detect vulnerabilities during the development lifecycle.

Contributions to the Field

- Specification of a comprehensive set of security-aware mutation operators applicable to RESTful APIs.
- Introduction of a generic framework for automated security testing tools.
- Empirical evidence showcasing improvements in test coverage and fault detection rates.

Expected Results II

Anticipated Challenges

- Balancing the trade-off between test coverage and execution time.
- Addressing redundancy in mutation operators to avoid excessive equivalent mutants.
- Ensuring scalability and applicability across different frameworks and programming languages.