Title: Design and Development of a Custom Browser Fuzzing Tool for Detecting Security Vulnerabilities
Name: Dhiaeddine Kraini
Student Number: 19185073
Supervisor: Hong Zhu

Table of Contents

---

1. Introduction

1.1 Background

Web browsers are the primary gateway for internet access and have become essential tools in modern computing. They accept and handle a wide range of untrusted inputs, including HTML, CSS, and JavaScript, making them a tempting target for malicious actors. Attackers often attack vulnerabilities in core browser subsystems such as JavaScript engines, HTML parsers, and DOM rendering systems, resulting in serious breaches such as data theft, unauthorised system access, and remote code execution [1].

In order to tackle these security issues, fuzzing has become a very successful method for identifying software flaws. A program can be made to display unexpected states, crashes, or other abnormal behaviours that are a sign of hidden faults by giving it a series of randomly generated or mutated inputs. Fuzzing has been crucial in exposing parser and execution mechanism flaws in browsers [2]. For instance, the Domato tool from Google Project Zero has successfully found browser vulnerabilities that were previously unknown by producing unusual DOM structures [3]. However, a

lot of fuzzers now in use have trouble developing realistic or extremely complicated test cases, which can lead to the failure to detect minor logic mistakes and memory corruption problems.

This project aims to address these constraints by creating a custom browser fuzzing tool from the ground up. By integrating advanced input generation and mutation techniques with automated execution pipelines for essential browser components, the tool aims to find hidden vulnerabilities that could be missed by existing solutions. The focus will be on producing inputs with high levels of complexity, scaling test execution across several engines, and carefully recording anomalies for further examination. This comprehensive approach seeks to increase browser fuzzing's application while also enhancing its ability to detect complex security vulnerabilities.

1.2 Aim

The main aim of this project is to create and develop a custom browser-fuzzing tool that automatically generates, mutates, and runs test cases in popular web browsers. It will focus on finding security flaws in key browser parts, such as JavaScript engines, HTML parsers, and DOM rendering systems, which might not be caught by current testing tools.

1.3 Objectives

The main objective of this project is to design and build a custom browser-fuzzing tool from scratch that can automatically generate, mutate, and execute test cases to find security weaknesses in popular browsers like Chrome and Firefox. By focusing on input mutation, the tool aims to produce realistic and complex inputs that uncover flaws in critical components such as DOM rendering systems, HTML parsers, and JavaScript engines which might be missed by existing solutions.

To ensure scalability and efficiency, the tool will include an execution engine capable of running tests in headless browser setups, along with a crash detection mechanism to log and analyze unusual behaviors. Finally, the project will evaluate the tool’s effectiveness by testing it on multiple widely used browsers, measuring how well it detects previously undiscovered vulnerabilities.

1.4 Product Overview

1.4.1 Scope

This project will deliver a browser fuzzing tool that automatically generates and modifies test inputs for key browser components namely HTML/CSS parsers, DOM rendering systems, and JavaScript engines. By creating faulty or malformed inputs and running them in headless environments (Firefox and Chrome), the tool supports large-scale, automated testing. It will also include a crash detection system that monitors for segmentation faults, memory corruption, and other errors. All crashes and anomalies will be logged and analyzed, giving security researchers and developers the data they need to identify and fix underlying vulnerabilities.

1.4.2 Audience

The tool will primarily benefit:

- **Security Researchers:** To assist in discovering new vulnerabilities in browsers and improve browser security.
- **Browser Developers**: To integrate the fuzzer into their development pipelines, ensuring the robustness and security of browser components.
- **Academic Researchers**: To provide a foundation for further research into fuzzing techniques and browser security.

---

2. Background Review

2.1 Existing Approaches

Several existing fuzzers and fuzzing approaches serve as foundations and comparators for this project:

1. **Domato (Google Project Zero)**:
    - Focuses on generating random HTML, CSS, and JavaScript to trigger DOM-related issues.
    - Strength: Good at quickly testing broad scenarios in JavaScript/DOM engines.
    - Limitation: Often lacks advanced mutation and seeding strategies, focusing instead on random generation. Struggles with generating deeply complex or context-specific test cases.
2. **IFuzzer**:
    - Uses invalid HTML generation to test rendering engines.
    - Advantage: Targets HTML parsing and rendering specifically.
    - Limitation: Insufficient coverage of other browser components, such as advanced JavaScript concurrency or cross-component interactions.
3. **Coverage-Guided Fuzzers (AFL, libFuzzer)**:
    - Provide systematic exploration of code paths by observing feedback from instrumentation.
    - Strength: High code coverage over time, guided by real-time performance metrics. They typically rely on a “seed corpus” to begin generating additional inputs.
    - Limitation: Browser environments are large and complex, requiring specialized harnesses and seed handling to run coverage-guided fuzzing effectively.
4. **FuzzBench**:
    - A benchmarking platform that compares various fuzzers on standard targets.
    - Strength: Offers reproducible metrics to measure fuzzers’ efficiency.
    - Limitation: Focuses on measuring fuzzers rather than providing a specialized solution for browser fuzzing, and it typically relies on structured seed corpora.

These tools highlight the need for a robust fuzzer capable of producing more complicated

inputs and thoroughly testing many browser components, ranging from JavaScript engines to HTML parsers and DOM renderers.

2.2 Literature Review

Fuzzing as a discipline is well-documented in both industry and academia. Key papers and findings include:

- **Whitebox Fuzzing (Godefroid et al.):** Proposed symbolic execution to systematically explore paths in software [2]. This technique, while thorough, can be computationally heavy for large-scale systems like browsers.
- **Mutation-based Techniques (Klees et al.):** In-depth comparison showing that mutation-based fuzzing often uncovers subtle memory corruption more efficiently [6].
- **Coverage-Guided Fuzzing (Zalewski's AFL):** Innovated the idea of dynamically focusing on inputs that trigger new code paths [7]. AFL highlights the role of a *seed corpus* for bootstrapping the fuzzer; from these seeds, further mutations can explore deeper and more complex functionality.

In parallel to these mainstream fuzzing approaches, **data mutation testing** has gained attention for its ability to generate structurally diverse inputs from seed data:

- **JFuzz (Zhu):** A Java testing framework that combines data mutation testing with metamorphic testing to automate both test case generation and result checking [8]. By mutating existing seeds, JFuzz can expose functional errors that simpler test frameworks might overlook.
- **Data Mutation Testing Applied to a Modelling Tool (Zhu):** Demonstrates how complex input formats such as modeling diagrams can be effectively tested by systematically mutating their structural elements [9]. This approach reveals faults that remain hidden when only using manual or random test generation.
- **Testing Software Modelling Tools Using Data Mutation (Shan & Zhu):** Extends the concept of data mutation testing to UML-based modeling tools, achieving high fault detection rates through large-scale structural mutations of seed diagrams [10]. Their findings underscore how the creation of numerous "mutants" from a small number of seed models can reveal subtler consistency and parsing flaws.

All of these data mutation testing initiatives support the more general fuzzing objectives of automation, robust logging, randomised test creation, and carefully selected seed inputs. They also show how deeper coverage in big, complicated systems may be obtained by changing structural or semantically significant aspects of incoming data. Lessons learnt from data mutation testing are especially relevant given the complexity of current browsers, where little changes to HTML, CSS, or JavaScript can result in radically altered execution pathways. A browser fuzzer may systematically reveal vulnerabilities that simple random input generators might overlook by applying mutation operators after being seeded with both valid and minimally invalid web pages.

---

3. Technical Progress

This section summarizes the technical progress made so far, including an updated project plan and a discussion of the work completed in alignment with the original proposal.

3.1 Project Plan Update

Originally, the project plan was defined in four main sprints (each spanning about 2–3 weeks). Below is an update:

- **Sprint 1: Input Generation Module**
  Status: Completed basic implementation.
  Details:
  - Designed a Python-based generator capable of producing random HTML/CSS constructs and minimal JavaScript.
  - Integrated a config-driven approach to define different tags, attributes, event handlers, CSS properties, and basic JS statements.
  - Implemented preliminary mutation strategies like random insertion of new HTML elements, altering existing CSS values, nested event listeners.
  - Established a basic seed corpus from small, well-formed HTML/JS/CSS files to guide the initial generation.
- **Sprint 2: Execution Engine**
  Status: Ongoing.
  Details:
  - I am going to be using *Selenium* for headless browser interaction.
  - Basic functionality to automatically launch browsers, feed them test inputs (including seeded ones), and track crashes or console errors.
  - Partial coverage instrumentation set up for the JavaScript engine. Next steps include refining instrumentation hooks and merging coverage data with test logs.
- **Sprint 3: Crash Detection**
  Status: Not Started.
  Details:
  - I am going to be deploying Google's AddressSanitizer builds for Chrome/Firefox to catch memory corruption.
  - Browser logs and system logs are going to be captured in separate files.
  - Basic triage mechanism is going to be implemented to categorize crashes by type.
- **Sprint 4: Testing and Evaluation**
  Status: Not started.
  Details:
  - The plan is to systematically run large-scale tests, collect coverage data, and measure crash frequency.
  - Will compare discovered vulnerabilities or crash triggers against those identified by established fuzzers.

3.2 System Requirements and Design

The system requirements drawn from the project's aims are:

1. **Functional Requirements**
    - The system must generate both valid and intentionally malformed HTML, CSS, and JavaScript inputs.
    - Must support headless execution in both Chrome and Firefox.
    - Must log all anomalies (crashes, unusual console logs, memory errors).
    - Must provide coverage metrics where available.
    - Must integrate a seeding strategy to kickstart fuzzing with known or curated inputs.
2. **Non-Functional Requirements**
    - **Performance**: Should handle generating and testing thousands of inputs each day while rotating through seed-based inputs to maintain diversity.
    - **Scalability**: Modular design so that new input mutators, coverage instrumentation, or seed files can be introduced with minimal effort.
    - **Reliability**: Should robustly handle browser restarts, ensuring that crashes do not halt the entire testing campaign.
    - **Maintainability**: Code is organized into modules: Input Generator, Execution Engine, Crash Manager, and Logger.

Design Overview

- **Input Generator Module**: Written in Python, uses configuration files to define a library of valid HTML/CSS/JS constructs, then introduces random and guided mutations. A curated "seed corpus" is used as a foundation to ensure coverage of common and historically vulnerable patterns.
- **Execution Engine:** A Python-based controller that launches headless browsers (using Selenium), provides them with newly created and mutated inputs, and then tracks console output and runtime behavior.
- **Crash Detection:** Employs environment variables and AddressSanitizer (ASan) instrumentation to detect memory leaks, use-after-free issues, and other memory-related faults.
- **Logging and Reporting:** Gathers crash data, coverage metrics (when possible), and console output in a structured format (JSON or CSV).
- **Seed Management:** Maintains an evolving library of baseline HTML, CSS, and JavaScript inputs drawn from real-world scenarios (such as open-source projects and W3C examples), ensuring each new mutation cycle starts with well-understood foundations.

3.3 Test Plan

Test Objective: Evaluate the fuzzing tool's ability to cause and detect unusual browser behavior or outright crashes.

- **Unit Tests**: Check the correctness of random and mutated input generation, including validating seed files to ensure they are syntactically correct.

- **Integration Tests**: Validate the interplay between the Input Generator, Seed Manager, and Execution Engine.
- **System Tests**:
    - Run full-scale fuzz campaigns on multiple browsers.
    - Monitor code coverage, memory usage, crash frequency.
    - Compare results when using different seed sets (minimal seeds vs. complex seeds from real websites).
- **Regression Tests**: Re-run known “bad inputs” that previously triggered crashes whether randomly generated or seeded to confirm that subsequent code changes have not regressed the detection mechanism.

3.4 Implementation Progress

- **Implemented**:
    - Basic version of the Input Generator, including an initial seed corpus of minimal HTML/JS/CSS.
    - Selenium-based pipeline for headless Chrome/Firefox.
- **Next Steps**:
    - Initial Logging: Collect basic information about crashes, console errors, and each test cycle.
    - Coverage Integration: Strengthen the link between coverage data and the seed-based approach (focusing on seeds that cover more code).
    - Mutation Improvements: Extend mutation algorithms to handle advanced JavaScript features (like concurrency, WebAssembly, and complex DOM interactions).
    - Test Cycle Performance: Make sure the system resets browsers efficiently and cycles through seed inputs in a way that increases coverage.
    - Crash Detection Completion: Fully integrate AddressSanitizer (ASan) builds for both browsers.
    - Large-Scale Testing: Run a major test campaign (using seeded inputs) to gather data on performance, coverage, and overall effectiveness.
    - Seed Corpus Refinement: Add real-world HTML examples to better reflect typical user scenarios and increase the chance of finding real-world bugs.

---

4. Professionalism and Risk

4.1 Risk Analysis and Mitigation

Managing risk is essential to ensure a successful outcome. The primary categories considered in this project are:

1. **Technical Risks**
    - **Complexity**: Browsers are highly complex; coverage instrumentation may be difficult to maintain.

        - Mitigation: Use existing instrumentation frameworks and keep the tool modular.
    - **Performance**: Fuzzing thousands of inputs daily can be time consuming, especially if seed-based campaigns grow large.
        - Mitigation: Optimize generation and use a powerful machine and cloud solution for scalability, prioritize seeds that yield the best coverage.
2. **External Risks**
    - **Dependencies on Browser Builds**: Relying on specialized builds of Chrome or Firefox with ASan can be delayed if new versions are not released promptly.
        - Mitigation: Maintain local copies of stable instrumented builds, track releases.
3. **Organizational Risks**
    - **Time Constraints**: Balancing academic deadlines and research is challenging.
        - Mitigation: Adhere to sprint schedules; allocate at least 10-15 hours per week specifically to project tasks, as recommended.
4. **Project Management Risks**
    - **Coordination with Supervisor**: Risk of missing crucial feedback.
        - Mitigation: Regularly attend supervisor meetings, maintain a log of feedback and tasks to address.

4.2 Professional, Legal, Social, Ethical, Environmental, and IP Issues

1. **Legal**
    - **Computer Misuse Act (1990), Data Protection Act (2018), and GDPR**: Relevant primarily if user data or personal data were ever used in test inputs. Currently, the fuzzer only uses synthetic data or publicly available open-source examples for seeding, so no direct conflict is expected.
    - **Copyright and Intellectual Property**: The code for the custom fuzzing tool is original, with references to open-source projects. I observe relevant open-source licenses, seeding from open-source code is done in compliance with each project's license.
2. **Social & Ethical**
    - **Security Research Ethics**: Fuzzing can potentially uncover vulnerabilities that could be exploited maliciously if disclosed irresponsibly. As per responsible disclosure practices, any discovered vulnerabilities will be reported to the relevant browser vendor and to my my supervisor for guidance.
    - **Ethical Considerations:**
      The project ensures that all tests are conducted in controlled, isolated environments, avoiding risks to live systems or sensitive information. No harm is done to users or external systems during the research.
3. **Professional Codes of Conduct**
    - **BCS, ACM, IEEE**: Emphasize integrity, competence, and respect for privacy. My project ensures that no personal user data is used. I also commit to responsible and private handling of any discovered vulnerabilities.

4. **Environmental Impact**
    - **Energy Usage**: Repeated fuzzing can consume significant computing resources. By leveraging seeding strategies and coverage guidance, the project aims to minimize wasted.
    - **Green Computing**: Whenever possible, we run the tool on energy-efficient hardware and only for intervals necessary to achieve meaningful coverage gains.
5. **Intellectual Property (IP)**
    - The resulting tool will be stored in a private GitHub repository until completion. Seed files derived from external sources will include notices regarding their original authorship and licensing.

4.3 Equality, Diversity, and Inclusion (EDI)

- The tool development and testing process remain accessible to a wide range of users and contributors.
- Documentation will be clear, and the project environment is designed to encourage collaboration.
- Where possible, the project references best practices in open-source communities to ensure inclusive language and accessibility.

---

5. Bibliography

[1] Fonseca, J., Vieira, M., and Madeira, H., “Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks,” *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, Melbourne, Australia, 2007, pp. 365-372.

[2] Godefroid, P., Levin, M. Y., and Molnar, D., “Automated whitebox fuzz testing,” *Communications of the ACM*, vol. 51, no. 6, pp. 107-113, Jun. 2008.

[3] Google Project Zero, “Domato: DOM Fuzzing Tool,” GitHub. [Online]. Available: https://github.com/googleprojectzero/domato. [Accessed: Oct. 15, 2024].

[4] Veggalam, S., Rawat, S., Haller, I., and Bos, H., “IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming,” in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds. Cham, Switzerland: Springer, 2016, pp. 527-548. [Online]. Available: https://doi.org/10.1007/978-3-319-45744-4_29.

[5] Metzman, J., Szekeres, L., Simon, L., Sprabery, R., and Arya, A., “FuzzBench: An open fuzzer benchmarking platform and service,” *Proceedings of the 30th USENIX*

Security Symposium (USENIX Security '21), 2021, pp. 1393-1403. doi: 10.1145/3468264.3473932.

[6] Klees, G. et al., “Evaluating fuzz testing,” in *Proceedings of the IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2018, pp. 530-545.

[7] Zalewski, M., *American Fuzzy Lop: A Security-Oriented Fuzzer*, 1st ed., Google Inc., 2014.

[8] H. Zhu, "JFuzz: A Tool for Automated Java Unit Testing," Technical Report CCT-AFM-2015-01, Dept. of Computing and Communication Technologies, Oxford Brookes University, Oxford, UK, Feb. 2015.

[9] H. Zhu, "Data Mutation Testing Applied to a Modelling Tool," Dept. of Computing, Oxford Brookes University, Oxford, UK, 2015.

[10] L. Shan and H. Zhu, "Testing Software Modelling Tools Using Data Mutation," in *Proc. of AST’06*, Shanghai, China, May 2006, pp. 1–10. DOI: 10.1145/1126008.1126018.