UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Triennale in Informatica

Leonardo Cantarella

Benchmarking Stateful Fuzzers over Lighttpd

Final Project Report

Supervisior: Giampalolo Bella

Advisor: Marcello Maugeri

External Advisor: Cristian Daniele

Academic Year 2023 - 2024

Abstract

Abstract goes here.

Contents

1 Introduction **3**

2 State of Art **4**
2.1 Introduction to Fuzzing 4
2.1.1 Types of Fuzzing Techniques 4
2.2 Stateful Fuzzing: Concepts and Challenges 5
2.2.1 Understanding Stateful Applications 5
2.2.2 Key Techniques in Stateful Fuzzing 6
2.2.3 Challenges in Stateful Fuzzing 6
2.3 Lighttpd: A Case Study for Stateful Fuzzing 7
2.3.1 Architectural Overview of Lighttpd 7
2.3.2 Relevance of Lighttpd for Stateful Fuzzing 8
2.4 Overview of Benchmarking Tools: AFLNet, ChatAFL, and Fallaway 8
2.4.1 AFLNet 8
2.4.2 ChatAFL 8
2.4.3 Fallaway 9
2.5 Summary 9

3 Fallaway Setup **10**

4 Comparison with AFLNet and ChatAFL **11**

5 Results **12**

Conclusion **13**

Bibliography **14**

Chapter 1

Introduction

Here goes the introduction.

Chapter 2

State of Art

2.1 Introduction to Fuzzing

Fuzzing, also known as fuzz testing, is a software testing technique used to discover coding errors and security vulnerabilities by inputting large volumes of random data, known as fuzz, into the system to cause unexpected behaviors. The primary objective of fuzzing is to find vulnerabilities such as buffer overflows, memory leaks, and other security weaknesses that could be exploited by attackers.

Fuzzing is effective due to its ability to automate the generation of test cases and to focus on areas of code that may not be exercised by traditional testing methods. It is particularly useful for applications with complex input grammars, where manually generating test cases would be infeasible.

2.1.1 Types of Fuzzing Techniques

There are several fuzzing techniques, each tailored to specific applications and objectives:

- **Black-box Fuzzing**: Generates inputs without any knowledge of the application's internal structure. It is easy to deploy but often less efficient due to the lack of internal feedback.
- **White-box Fuzzing**: Uses detailed information about the application's source code, such as control flow and data flow, to generate test cases that maximize code coverage. This approach often involves complex static and dynamic analysis techniques.
- **Grey-box Fuzzing**: Combines elements of both black-box and white-box fuzzing, using partial knowledge about the application's inter-

nals (e.g., code coverage feedback) to guide input generation. It balances simplicity with effectiveness and is exemplified by tools like AFL (American Fuzzy Lop).

- **Mutation-based Fuzzing**: Creates new inputs by randomly mutating existing ones, without requiring knowledge of the input format. It is versatile but may struggle with applications that require highly structured inputs.
- **Generation-based Fuzzing**: Generates inputs from scratch based on a formal specification of the input format, such as a grammar or protocol description. It is particularly effective for applications that require complex or structured inputs.

2.2 Stateful Fuzzing: Concepts and Challenges

Stateful fuzzing extends traditional fuzzing techniques by incorporating state-awareness. Traditional fuzzing often treats each input as an independent test case, but stateful fuzzing considers the application's internal state and how this state may affect the processing of subsequent inputs. This is particularly important for applications that maintain complex state information, such as network servers, databases, and interactive applications.

2.2.1 Understanding Stateful Applications

Stateful applications maintain a state across multiple interactions or sessions. For instance, a network server may keep track of the connection status, authentication states, session identifiers, or application-specific state information. These states influence how inputs are processed and can affect the application's behavior over time. Proper handling of state transitions is critical for both security and reliability, as state-dependent bugs can lead to vulnerabilities such as unauthorized access, denial of service (DoS), or data corruption.

Stateful fuzzers aim to model and explore these state transitions by generating sequences of inputs that mimic real-world usage scenarios, while tracking the state changes to ensure comprehensive testing of all possible state transitions.

2.2.2 Key Techniques in Stateful Fuzzing

Stateful fuzzing involves several advanced techniques that distinguish it from traditional fuzzing approaches:

- **State Modeling**: Construction of a model representing the application's state machine, either by analyzing source code, observing real interactions, or employing machine learning techniques. This model helps guide test case generation.
- **State Tracking**: Monitoring the application's state throughout multiple inputs, which may involve tracking network traffic, system calls, or internal state variables.
- **Feedback Mechanisms**: Utilizing feedback mechanisms to prioritize test cases that explore new states or code paths, thereby improving the efficiency of the fuzzing process.
- **Sequence Generation**: Creating sequences of inputs that accurately reflect real-world interactions with the application, essential for discovering vulnerabilities that depend on specific input sequences or state changes.
- **Learning-Based Approaches**: Some fuzzers employ machine learning or heuristic approaches to dynamically learn the application's state machine structure, allowing the fuzzer to adapt and improve over time.

2.2.3 Challenges in Stateful Fuzzing

Stateful fuzzing poses several challenges that need to be addressed for effective testing:

- **State Explosion**: As the number of possible states in an application grows, the complexity of the fuzzing process increases exponentially. This *state explosion* problem necessitates strategies such as state abstraction, pruning, or prioritization to manage the complexity.
- **Protocol Complexity**: Understanding complex protocols or state machines is often required to generate meaningful input sequences, which demands significant domain-specific knowledge or advanced learning algorithms.

- **Performance Overhead**: The need to track state and generate sequences of inputs can lead to substantial computational overhead, impacting the performance of the fuzzing process.
- **Handling Non-Deterministic Behavior**: Many stateful applications exhibit non-deterministic behavior due to factors such as concurrency, external inputs, or timing variations, complicating the reproduction of bugs and consistency of fuzzing results.

2.3 Lighttpd: A Case Study for Stateful Fuzzing

Lighttpd is an open-source web server optimized for high performance with a low memory footprint. Designed to handle a large number of parallel connections, Lighttpd is well-suited for environments with limited resources or high concurrency requirements. Its modular architecture and support for advanced web protocols make it popular for embedded devices, cloud platforms, and high-traffic websites.

2.3.1 Architectural Overview of Lighttpd

Lighttpd follows an event-driven architecture, which enables it to handle numerous requests concurrently. It uses an asynchronous I/O model that reduces the overhead associated with network connections, allowing it to scale efficiently under load. Some of the key features of Lighttpd include:

- **Modular Design**: Offers a range of modules that provide various functionalities, such as URL rewriting, HTTP compression, SSL/TLS, and WebSockets. This modular approach enables customization based on specific needs.
- **Protocol Support**: Supports multiple protocols, including HTTP/1.1, HTTPS, FastCGI, SCGI, and HTTP/2, making it versatile for different web applications and services.
- **Security Features**: Built-in security features include support for TLS/SSL encryption, defense against denial-of-service attacks, and various authentication mechanisms.

2.3.2 Relevance of Lighttpd for Stateful Fuzzing

Lighttpd's state management complexity, particularly in protocol handling, makes it an ideal candidate for stateful fuzzing:

- **State Management Complexity**: Handles multiple client connections concurrently, each with potentially different states based on protocol version, authentication, request type, and other factors.
- **Protocol Vulnerabilities**: Due to its support for multiple web protocols, Lighttpd is vulnerable to a range of potential attacks that could arise from improper state handling, such as request smuggling, race conditions, and state confusion attacks.
- **Modular Configuration**: Its modular architecture and numerous configurations present unique states or behaviors, which can be thoroughly tested using stateful fuzzing.

2.4 Overview of Benchmarking Tools: AFLNet, ChatAFL, and Fallaway

For this thesis, three stateful fuzzers—AFLNet, ChatAFL, and Fallaway—will be benchmarked over Lighttpd to evaluate their effectiveness in uncovering state-dependent vulnerabilities.

2.4.1 AFLNet

AFLNet is an extension of the American Fuzzy Lop (AFL) designed specifically for fuzzing stateful network protocols. It integrates state-awareness into AFL's grey-box fuzzing framework, allowing it to target specific protocol states and maximize state coverage during fuzzing.

2.4.2 ChatAFL

ChatAFL is a stateful fuzzer that leverages machine learning techniques to improve the fuzzing of stateful applications. It is particularly effective for applications involving natural language processing or interactive protocols, using reinforcement learning to adapt and optimize its input generation strategies.

2.4.3 Fallaway

Fallaway is a stateful fuzzer that combines static analysis with dynamic state tracking to explore the state space of the target application. It constructs a preliminary model of the application's state machine through static analysis and refines this model during dynamic fuzzing sessions. Fallaway's hybrid approach is particularly suited for applications with large or poorly documented state machines.

2.5 Summary

Stateful fuzzing represents a significant advancement in software testing and security, particularly for applications with complex state management requirements like Lighttpd. By considering the internal state of the application, stateful fuzzers such as AFLNet, ChatAFL, and Fallaway can more effectively identify vulnerabilities that would be missed by traditional fuzzing techniques. The following chapters will provide a detailed benchmarking of these tools over Lighttpd to evaluate their effectiveness and identify areas for further improvement.

Chapter 3

Fallaway Setup

All the info about the setup for fallaway. mutator, settings, corpus, etc.

Chapter 4

Comparison with AFLNet and ChatAFL

How they works, how they are different, how they are similar, how they are better, how they are worse, etc. Script, dockerfile, etc. Replayer.

Chapter 5

Results

Discussions and comparisions of graphs and results.

Conclusion

Bibliography