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

This thesis compare three stateful fuzzers—Fallaway, AFLNet, and ChatAFL—by employing them against Lighttpd, a high-performance web server. The significance of stateful fuzzing lies in its ability to identify vulnerabilities in applications characterized by intricate internal states, which may be overlooked by conventional fuzzing techniques. This research compares these instruments with regard to code coverage, crash detection, and the different kinds of vulnerabilities. These results provide enlightening insights into the strengths and weaknesses of each fuzzer, hence guiding selection and improvements of stateful fuzzing approaches for modern software systems.

Contents

1 Introduction

As the software systems are getting increasingly complex, ensuring their robustness and security has turned out to be a serious challenge. In this scenario, fuzzing has emerged as a powerful technique in the identification of security vulnerabilities and defects, which may remain elusive for traditional testing techniques. Fuzzing involves the generation of random test inputs in order to see how the software reacts to unexpected or malformed input data, looking for problems, such as crashes, unexpected behaviour, or security vulnerabilities.
The evolution of fuzzing methodologies has significantly enhanced their effectiveness for a wide range of applications. Traditional fuzzers typically focus on generating random inputs and observing the software responses. However, for applications that maintain internal states across several interactions, such as web servers or networked applications, this approach can be insufficient. These stateful applications call for more sophisticated fuzzing techniques that take into account the interaction between different states and transitions.
Stateful fuzzing is an advanced approach for solving the problems of applications that rely on state management. Whereas in stateless fuzzing, each input is considered a unique event, stateful fuzzing emulates the flow of activities along with the succeeding changes in the state of an application. It includes the generation of inputs which consider previous interactions and what these have done to the state of the application, hence providing a more realistic and deeper testing process.
An instance of a stateful application is Lighttpd, an open-source web server recognized for its effectiveness and ability to scale. Lighttpd is engineered to manage a substantial number of concurrent connections and accommodates multiple network protocols, rendering it a sophisticated system with considerable state management demands. Assessing Lighttpd offers a chance to scrutinize stateful fuzzing methodologies owing to its complex architecture and variable behaviour.
This thesis focuses on the benchmarking of stateful fuzzers to ascertain their efficiency in detecting all the existing vulnerabilities in Lighttpd. Our evalu-

ation targets three such prominent stateful fuzzers: Fallaway, AFLNet, and ChatAFL. Each of them has a different approach toward stateful fuzzing:

- **Fallaway**: Fallaway couples static analysis with dynamic state tracking, aiming at an effective exploration of large state spaces.
- **AFLNet**: AFLNet extends the capabilities of the AFL fuzzer for stateful network protocols, increasing its power in testing complex network interactions.
- **ChatAFL**: ChatAFL uses machine learning methodologies to improve fuzzing strategies and adapt to the application behaviour.

The key contribution of this paper is to compare these fuzzers based on several metrics, including code path coverage, different crashes, and other types of vulnerabilities they were able to find. By analyzing the performance of all these tools, we aim to report on various strengths and weaknesses of each, which gives necessary suggestions for improving stateful fuzzing techniques and enhancing the security of modern software systems. This introductory section creates a base for deep research in stateful fuzzing methodologies and their realization in real systems, setting the base for further chapters examining extensive tests and results.

2 Background

2.1 Introduction to Fuzzing

Fuzzing, or fuzz testing, is a software testing technique that includes feeding a huge amount of random data into the system, called SUT (System Under Test), to find unprecedented responses and reveal major programming errors, along with key security vulnerabilities. The primary objective of fuzzing is to identify vulnerabilities such as buffer overflows, memory leaks, and other security weaknesses that can be exploited by attackers.
The success of fuzzing is based on its capabilities for automatic test case generation and for focusing its attention on portions of programs that otherwise would not have been tested by other more traditional testing technique.
Indeed, it is particularly effective for applications with complex input grammars, where manual test case creation would be impracticable.

2.1.1 Types of Fuzzing Techniques

Fuzzing methodologies vary, and there exist a lot for different applications and purposes:

- **Black-box Fuzzing**: This is a technique of generating inputs without prior knowledge of the internal structure of an application. It is easy to deploy but often less efficient as there is no internal feedback.
- **White-box Fuzzing**: This is one of those techniques that rely heavily on source code intuition, such as control flow and data flow, to provide maximum code coverage with test case generation. The approach often employs some sort of complex static and dynamic analysis methodologies.
- **Grey-box Fuzzing**: It's a strategy that combines the various merits of black-box and white-box fuzzing. It brings in partial knowledge about internal application details, and code coverage feedback guiding

the generation of inputs. It balances simplicity with effectiveness, and an example might be the family of tools called AFL (American Fuzzy Lop).

- **Mutation-based Fuzzing**: This generates new inputs through random mutations of existing inputs. It requires no knowledge about the structure of the inputs but is often a lot weaker compared with other generations for applications requiring highly structured inputs.
- **Generation-based Fuzzing**: This builds the inputs from scratch, based on a formal characterization of the input format, grammar or protocol specification. It has proved quite effective in applications where the inputs have to be complex or systematically structured.

2.2 Stateful Fuzzing: Concepts and Challenges

Stateful fuzzing adds state awareness to traditional fuzzing methods. Traditional fuzzing approaches treat every input individually as a test case, but stateful fuzzing considers an application's internal state and how that state might affect subsequent inputs handling. This becomes particularly critical for applications that handle complex state information, such as network servers, databases, and interactive applications.

2.2.1 Understanding Stateful Applications

Stateful applications maintain some sort of state across multiple interactions or sessions. Examples include network servers maintaining connection states, authentication states, session identifiers, or other state information specific to an application. These states could contribute significantly to the processing of inputs and, therefore, influence the behavior of the application over time. Good practice in state transition management is important for security as well as reliability: bugs dependent on state can lead to vulnerabilities such as unauthorized access, DoS, or corruption of data.
Stateful fuzzers attempt to model and explore such state transitions through the generation of input sequences that mimic valid usage scenarios while monitoring state changes concurrently to ensure adequate coverage 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**: The process of building a model representing an application state machine by reverse engineering source code, observing real interactions, or guide test case generation in conjunction with machine learning techniques.
- **State Tracking**: This involves tracking the state of the application across successive inputs; indeed, tracking of network traffic, system calls, or internal state variables.
- **Feedback Mechanisms**: With feedback mechanisms, one can prioritize those test cases that tend to explore new states or code paths; hence, the general efficiency of fuzzing can be improved.
- **Sequence Generation**: It is the need to generate input sequences to properly model actual use, since the findings of vulnerabilities often depend on specific sequences or state transitions.
- **Learning-Based Approaches**: Certain fuzzers utilize machine learning or heuristic methodologies to dynamically ascertain the structural configuration of the application's state machine, thereby enabling the fuzzer to adjust and enhance its efficacy progressively.

2.2.3 Challenges in Stateful Fuzzing

Successfully performing testing is fraught with several challenges in stateful fuzzing:

- **State Explosion**:As in real life, an application itself may have a number of possible states, and with more states, a risk for exponential growth in process complexity increases. In this case, state abstraction, pruning, or prioritization counters the *state explosion* in an essential way.
- **Protocol Complexity**: Generating meaningful input sequences can involve deep knowledge of complex protocols or state machines. This often includes much domain-specific knowledge or even advanced algorithms.

- **Performance Overhead**: To date, state tracking performed by the application and input sequence generation can cause significant computational costs, hence slowing down the fuzzing process.
- **Handling Non-Deterministic Behavior**: The nondeterministic behavior of stateful applications often results from concurrency, differences in external inputs, or even timing variations. These factors therefore make the reproduction of bugs and receiving consistent fuzzing results usually difficult.

2.3 Lighttpd: A Case Study for Stateful Fuzzing

Lighttpd is an open-source web server optimized for performance with very low memory usage. It is designed to handle huge volumes of parallel connections with minimal overhead, making it particularly useful on systems with limited resources or those requiring a high degree of concurrency. Its modular design and support for advanced web protocols make it a popular choice for embedded systems, cloud computing platforms, and high-traffic websites.

2.3.1 Overview of Lighttpd Architecture

Lighttpd operates on an event-driven architecture, which enables it to serve many requests concurrently. An asynchronous I/O framework is employed to minimize overhead in network connections, allowing the server to scale efficiently under varying workloads. The key features of Lighttpd include:

- **Modular Design**: Provides a series of modules for implementing functions like URL rewriting, HTTP compression, SSL/TLS, and WebSockets. The modular design allows for customization based on specific needs.
- **Protocol Support**: Out of the box, it supports HTTP/1.1, HTTPS, FastCGI, SCGI, and HTTP/2, making it suitable for a wide range of web applications and services.
- **Security Attributes**: Advanced integrated security features include TLS/SSL encryption, prevention of denial-of-service attacks, and multiple authentication options.

2.3.2 Relevance of Lighttpd for Stateful Fuzzing

The complexity of state management in Lighttpd, especially in protocol processing, makes it an ideal candidate for stateful fuzzing:

- **State Management Complexity**: Responsible for managing numerous concurrent client connections, each with its own state depending on protocol versioning, authentication, and request type.
- **Protocol Vulnerabilities**: The support for multiple web protocols exposes Lighttpd to a range of potential attacks due to improper state management, such as request smuggling, race conditions, and state confusion attacks.
- **Modular Configuration**: Its modular architecture and extensive array of configuration options facilitate the emergence of distinct states or behaviors, which can be effectively evaluated through stateful fuzzing techniques.

2.4 Fuzzers Overview: 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.

3 Fallaway Setup

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

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.

5 Results

Discussions and comparisions of graphs and results.

Conclusion

Bibliography