In the name of God

Sharif University of Technology
Department of Computer Engineering

CE 815: Secure Software Systems

Mehdi Kharrazi *Azar 12^{th}, 1404*

Homework 2*

1 FUZZING*

In this part of the assignment, you are asked to fuzz a codebase. The target code is a MessagePack library, and the program file along with an example of how to use it is provided in the handouts. Your task is to first fuzz the code to identify any bugs and then fix these bugs. You are required to perform fuzzing using the Atheris fuzzer.Your work consists of the following three parts:

- **Fuzzing the Code:** Use Atheris to fuzz the `msgpacker.py` implementation. In addition to simply running Atheris, you must:
  - build an initial **seed corpus** (at minimum: valid MessagePack structures, boundary cases, malformed inputs),
  - monitor **edge coverage** during execution,
  - log which inputs produce **new coverage** and why,
  - analyze how different input classes expand program exploration.
- **Optimizing the Fuzzer:** Improve your fuzzer so that it discovers deeper program paths with fewer test cases. Your optimization results will be evaluated compared to the best optimization results in other hw submissions.
- **Fixing Bugs:** Identify and resolve all bugs triggered by the fuzzer . Provide clear explanations of:
  - the failing input,
  - the root cause of the bug,

Note: You may omit implementing support for floats, which the provided `msgpacker.py` does not support.

1.1 Delivery

Submit:

- your corrected `msgpacker.py`,
- all fuzzing scripts (Atheris driver, corpus generator),
- all corpora used (initial seeds + minimized final corpus, i.e., the corpus reduced by removing redundant inputs while preserving the same coverage)
- and a complete report.

*Acknowledgement: This homework was orginally developed by Iman Hosseini and Solmaz Salimi and edited by Zahra Fazli

*https://61600-labs.csail.mit.edu/lab5.html

Report Requirements (Very Important)

Your report is a major part of your grade and **must** include, at minimum:

- **Fuzzing Methodology:**
  - how the corpus was constructed and why each seed matters,
  - explanation of your optimizations and how they improved exploration.
- **Coverage Analysis:**
  - functions with the highest and lowest coverage,
  - which code regions were *never* covered and your explanation for why,
  - comparison of coverage before and after optimizer improvements.
- **Bug Analysis and Fixes:**
  - include the crashing input (or differential-triggering input),
  - explain the root cause,
  - describe the fix.

Quality of the report is heavily weighted. A clear, structured, and technically deep report is required for full credit. Poor or shallow reports will receive significant penalties regardless of code quality.

2 SAT: Source Code Analysis Tool

In this assignment, you will implement a simple static analysis tools for Java programs enabling vulnerability detection. First, you should generate an AST (abstract syntax tree) for a Java program in order to perform these analyses.

2.1 Instructions

The goal of this part is to implement a simple analyzer to detect:

a) division by zero,

b) negative array index,

c) bad shift *(i.e., shift by a constant that is greater than 31 or less than 0)*,

d) integer overflow,

e) taint-based vulnerabilities.

As you see in the handout examples, you should also handle cases in which a statement may execute zero or more times, using control-flow structures (i.e., `if`, `for` and `while`). Actually, you should statically parse the code and derive a **symbolic formula** for each integer variable. Then, based on the symbolic execution, you can determine how the value of each variable changes based on input arguments, and whether that value may trigger one of the above vulnerabilities.

A) Suppose that the main function calls other function with some concrete values. In this case, your task is very straigh-forward: just replace the concrete values in your symbolic formula (i.e., symbolic execution) of each statement to check whether the aforementioned errors/vulnerabilities may occur in the program.

B) Now, suppose a more realistic case: the main function calls other functions with user-provided input. For this part, you should treat input values as symbolic unknowns. You should use Z3 as an SMT solver to:

- solve your symbolic constraints,
- determine inputs that lead to division by zero, negative index, bad shift, overflow or a taint flow into a sensitive sink.

Hint 1: You can use any available Java parser or develop your own.
Hint 2: Sample Java files are included in the handout repository.

C) You must extend your symbolic execution engine to additionally perform a simple **taint analysis**. A value is considered **tainted** if it originates from an untrusted source. In this assignment:

- **Taint Sources**:
  - user-provided inputs (`args[]`),
  - values read from user input (e.g., via `Scanner`, `BufferedReader`)
  - return values of functions whose outputs are already tainted.
- **Taint Propagation:**
  - If any operand of an expression is tainted, the entire expression becomes tainted.
  - If a tainted variable is passed as an argument to a function, the corresponding parameter becomes tainted.
  - The return value of a function becomes tainted if any of its internal computations use tainted values.
- **Sensitive Sinks:**
  - `runtime.getRuntime().exec()`,
  - `system.exit()`,
  - file/OS operations,

2.2 Delivery

You should submit your code, which takes a Java file as input and prints the symbolic execution of statements. At the end of each function, it should print:

- the final symbolic formulas for every variable (sorted alphabetically),
- the concrete (evaluated) values of variables in part A,
- detected vulnerabilities from both symbolic execution and taint analysis,
- in part B, the input values generated by the SMT solver that trigger each vulnerability.

Note: The format of your output must match the structure and style of the sample output provided in the handout, including:

- the "Symbolic formulas" block,
- step-by-step symbolic execution log (e.g., *Symbolically Declare*, *Symbolically Assign*, *Symbolically call functionn*, etc.),
- two-phase execution (AST exploration rounds),
- final result summary.

Failure to follow the expected output format will result in point deductions.

LLM Usage Policy

You may use Large Language Models (LLMs) such as ChatGPT, DeepSeek, etc. as auxiliary tools. Any use of LLMs must be clearly disclosed in the report, including:

- which services were used,
- for which tasks, and
- which parts of the code or report were LLM-assisted.

You are fully responsible for the correctness of all submitted work, regardless of LLM usage.