FuzzSlice
Technical Documentation

Version 1.0.0

Aniruddhan Murali | Noble Saji Mathews

December 29, 2023

Contents

List of Code Listiångs

1 Introduction

This document is a technical documentation for FUZZSLICE tool. The tool aims to tackle the challenge of a very large number of false positives in static analysis tools. FUZZSLICE is an approach that aims to identify these false positives in static analysis reports in an efficient manner. Instead of fuzzing the entire program end-to-end from the `main` function, FUZZSLICE only fuzzes the function containing the warning to identify possible false positives. FUZZSLICE hinges on the novel idea that a warning fuzzed at the function level and not yielding a crash in a given time budget is most likely a false positive. Unlike typical methods that reduce fuzzing cost by independently fuzzing modules and libraries, FUZZSLICE generates minimal compiled slices enclosing any arbitrary warning location from any static analysis tool. Fuzzslice generates and fuzzes a seperate binary for each warning which facilitates the coverage of most warnings. In the case of warnings that do crash in the given time budget, these are worth manually checking or confirming for true positives by developers.

In this manual we color certain text to highlight and differentiate from other text. For example, commands to be executed are in dark purple. Also functions, files, paths, repositories are in light blue. The configuration options and fields of results are colored in dark blue.

2 Motivating Example

```c
 /* Glue an array of strings together and return it as an allocated
   string.
 */
char *glue_strings(const char *list[])
{
    size_t len = 0;
    char *p, *ret;
    int i;

    for (i = 0; list[i] != NULL; i++)
        len += strlen(list[i]);

    if (!(ret = p = OPENSSL_malloc(len + 1)))
        return NULL;

    for (i = 0; list[i] != NULL; i++)
        p += strlen(strcpy(p, list[i])); //False positive

    return ret;
19 }
```

Listing 1: Code snippet from Openssl project flagged by RATS as buffer overflow.

The goal of this tool is to identify possible false positives efficiently. In this section, we provide an example to motivate the FuzzSlice approach. An example code listing of the repository openssl in the C language is shown in Listing 1. The code listing describes a function glue_strings that joins an array of strings (the function argument) into a single string (the return value). On line 10, the variable `len` is updated in a loop to hold the sum of length of all input array strings. On line 12, the variable ret is dynamically allocated with size of 'len+1'. On line 16, the string is joined together in pointer variable ret by iterating the pointer p and copying each input string one by one.

When a static analysis tool such as RATS is run on this code it flags line 16 as a possible heap buffer overflow. However, this is clearly a false positive because the strcpy on line 16 can never exceed bounds of allocated pointer ret. The reason behind this is that the size of allocated pointer ret will always be one plus the length of all input strings. RATS is not capable of this kind of value flow analysis for variable len. Therefore the tool cannot be sure that line 16 will never cause a heap buffer overflow.

Fuzzing has become a popular solution for the verification of static analysis reports. It is possible to compile the whole openssl binary and guide the fuzzing towards line 16 in the code snippet from the main method. However, this often take several days, several CPU cycles and requires the help of appropriate fuzzing dictionaries. Inspite of this, there is no guarantee that the given static analysis warning can be covered by fuzzing within given time budget.

FuzzSlice is an approach that is primarily targeted towards false positives in static analysis warnings. It takes advantage of the fact that the function glue_strings in Listing 1 can be fuzzed directly to identify it is a false positive. Given an arbitrary warning location, it automatically constructs a function slice of the program, compiles it including necessary dependencies, generates it's own fuzzing wrapper and fuzzes the function slice. When the function slice is fuzzed on its own, the static warning on line 16 can be easily reached. Let us assume that fuzzing this function slice gives no crash on line 16. This implies that fuzzing from main will also not result in a crash. This can be derived by the fact that caller functions can only constrain the input to a given function through the function arguments. On the other hand if a crash is observed on the static analysis warning line then we cannot comment on it being an actual bug or a true positive. This is because a caller function can invalidate the input that causes the observed crash in the slice. FuzzSlice aims to identify all false positives within a static analysis report similar to Listing 1.

3 Prerequisites

FuzzSlice works on C code only as of current version. Additionally the project must be able to build on its own in the same environment that FuzzSlice operates in. This is necessary because FuzzSlice will attempt to capture this project build information in order to make it's own compiled slice capturing the warning before fuzzing.

4 Docker setup for existing test repositories in FuzzSlice

If the user is interested in a quick run to see how the tool works then a docker version is readily available. The existing projects for testing in FuzzSlice are tmux, openssh-portable, openssl, Little-CMS and C (Juliet) dataset. We describe the procedure to run any of these projects below:

These instructions will get a docker of the tool up and running on your local machine:

1. (Required) Prerequisite: Install Docker.
   sudo apt install docker-ce

2. (Required) Building docker image: First clone this repo, and enter the root directory of the project. Recursively fetch the submodules by running:
   git submodule update –init –recursive

3. (Required) You can now build the environment required for the tool from using the Dockerfile included by running the following command:
   docker build -t sf .

4. (Required) Launching docker image with tool: The build command will both setup the environment and the test repositories. To enter the container invoke:
   docker run -it sf

5. (Optional) Set the repository to analyze as test_library option within config.yaml. This means you can choose from one of tmux, openssh-portable and openssl as the test_library to analyze. The default repository is openssl.

6. (Optional) Modify the static analysis warnings to classify within info_lib/<test_library>/targets.txt. Each repository already has a default set of static analysis warnings to choose from.

7. (Required) Run main script: **python3 main.py**

5 Manual setup for new project

Let us assume that user is interested in using FuzzSlice on a static analysis warning in a new project different from the ones used as test repositories in FuzzSlice. The following steps are critical:

1. (Required) First the user must copy the root directory of the project into the test_lib directory within FuzzSlice.
   iie. **cp -R <project_name> test_lib/**

2. (Required) Ensure that the static analysis warnings to be analyzed are of the format: <path_to_file_with_warning>:<line_no>:<issue>

3. (Required) The directory info_lib contains individual directories for each project in test_lib. These directories each contain a file called targets.txt which is the file with the static analysis warnings to be analyzed. Therefore please ensure that there exists a file called: info_lib/project_name/targets.txt that contains the warning in required format.
   iie. **echo <warning> » info_lib/project_name/targets.txt**

4. (Warning) At this stage, FuzzSlice requires the build information for the new project. FuzzSlice expects the configuration to look like '../configure' and the build command to be make command. A different build process is not yet supported.

5. (Required) Now to set FuzzSlice to examine the given project, open the config.yaml file in the FuzzSlice root directory. This file contains a field called test_library. This field must be set to project_name. The timeout for fuzzing each warning can be changed in the `timeout` option.

6. (Required) Finally run **python3 main.py** in the root directory.

6 FuzzSlice Configuration Options

In this section we discuss the different options available for configuring FuzzSlice in detail. The configuration file for FuzzSlice is available within config.yaml file in the top level FuzzSlice directory. The different options available for modification in this file are listed below:

1. **test_library** : This corresponds to the project that is to be fuzzed. There must exist a directory with the same project name within the test_lib directory corresponding to the actual project. Similarly a directory with same name should also be present in info_lib directory containing a file called targets.txt containing all the static analysis warnings to be analyzed.

2. **fuzz_tool** : The fuzzing tool to be used - AFL(1) or Libfuzzer(0)

3. **timeout** : The fuzzing timeout for each static analysis warning

4. **hard_timeout** : The total timeout for a warning including compilation and fuzzing.

5. **max_length_fuzz_bytes** : The maximum bytes used to fuzz a warning.

6. **parallel_execution** : When set to 1, fuzzing is done in parallel for multiple issues only when batch mode is active.

7. **crash_limit** : The maximum number of crashes to analyze after fuzzing an issue.

The above parameters in the config file of FuzzSlice control all important internal properties of FuzzSlice. The user can use these configuration variables to apply FuzzSlice as they see fit for any new project.

7 Viewing sliced source code files

FuzzSlice stores the compiled slice for each warning within workspace/<project_name>/test_files directory. In the batch mode (described in the configuration) all sliced files for all warnings are available for viewing. If the batch mode is not enabled then each warning is created and fuzzed. Subsequently the workspace is cleaned before fuzzing the next warning.

Additionally the workspace/<project_name>/test_files directory contains several subdirectories for each additional required dependency required to compile a slice containing the warning. These dependencies are also compiled before finally linking to obtain the binary for the sliced code.

8 Result logs

The result of FuzzSlice is the classification of static analysis warnings into true positives, possible false positives, not executed warnings, and not compiled static analysis warnings. The classification of static analysis warnings as obtained in the FuzzSlice report is shown in Listing 2.

```
===========================FUZZING STARTS============================
Total number of source files : 3
The number of binaries to be fuzzed: 3
=========================FUZZ REPORT=================================
Number of possible True positives: 2
 ./test_lib/usrsctp/usrsctplib/netinet/sctp_auth.c:2555:SEGV
 ./test_lib/usrsctp/usrsctplib/netinet/sctp_indata.c:1252:SEGV

Number of possible False positives: 1
./test_lib/usrsctp/usrsctplib/user_mbuf.c:1270:SEGV

Number of unreachable issues: 0

Number of files that are not compiled :0
======================FUZZ REPORT END================================
===========================FUZZING ENDS============================
```

Listing 2: Fuzzing report of FuzzSlice which classifies the static analysis warnings.

More detailed results will be stored as a log file in info_lib/project_name_log.txt. Within the log file each line corresponds to different warning and looks as follows:

{"target_line_hit": "9", "coverage_ratio": 0.44, "type": "TP", "time_compile": 9.0, "time_fuzz": 461.27, "time_total": 470.29, "issue": "./test_lib/openssl/params_test.c:660"}

We describe each field in detail below:

1. **target_line_hit**: Number of times warning was executed.
2. **type**: This refers to the classification of the warning. This can be 'TP' or true positive if a crash was observed at warning location, 'FP' or false positive if no warning was observed at warning location and the warning was executed and finally 'NR' if the warning could not be executed.
3. **coverage_ratio**: Coverage ratio within minimized slice
4. **time_compile**: Time taken to create a minimized sice around warning.

5. **time_fuzz**: Time taken to fuzz
6. **time_total**: total time taken for warning
7. **issue**: Warning path and line number