Homework Exercise 3

ENGR 201, 2017

Dylan Chong - 300373593

Topic: Available methods for finding security issues in software

### Overview of the SimFuzz approach

Zhang et al ([7]) proposed a new approach for generating and selecting test cases for deep execution paths. Deep execution paths are many in number so are very time consuming to test manually, vulnerabilities are likely to be found in these deep execution paths. Existing Fuzz and Concolic testing approaches generate a large number of test cases, but most of these involve using inputs that the program under test (PUT) would consider invalid. Invalid inputs do not allow the program to follow deep execution paths because the program would not progress far enough from the initial states (i.e. the program can only follow a shallow execution path). The proposed approach uses a Test Case Similarity metric (TCS) in order to find similar execution paths to some original deep path - in order to have high coverage of deep program states, a set of similar paths are required to since the program into related states. The proposed approach has been implemented in a tool called SimFuzz for experimentation [7].

### Phases

The approach can be broken down into two phases, with an optional third phase [7].

The purpose of the first phase is to create a set of similar inputs that cause similar deep execution paths, using a proposed Incremental Mutation process. Begin with a valid input that creates a deep execution path, and break it into small segments. For each segment, perform a slight alteration and if the resulting input creates an execution with a high TCS to the original, add this to the set of test cases, and then repeat by performing a larger alteration. This process creates a range of inputs that are as broad as possible while still being able to focus on testing related execution path [7].

The second phase involves combining multiple test cases from the first phase in order to create additional test case inputs. This is necessary for finding vulnerabilities that are only possible when multiple parts of the input have been malformed in specific ways, and also to make sure that related execution paths are explored by the test cases [7].

The optional third phase is required only when there are not enough test cases generated from the previous two stages in order to provide sufficient coverage for the deep execution path. Because having a strictly high TCS requirement can filter out a large number of generated test cases in the first phase, retrying the entire process again with a lower TCS requirement can help to increase the coverage of the deep execution path [7].

### Advantages and Limitations

It is a very efficient process because test cases are created intentionally and selectively because only the important test cases are executed - the existing Concolic testing approach creates a large number of test cases and don't filter any out, so the test generation execution process can be very time-consuming, especially for large programs. In comparison with the existing Fuzz testing approach, it has a much higher coverage of deep execution paths because most test cases generated by fuzzing only test shallow execution paths, due to the randomness and therefore largely invalid inputs. Vulnerabilities are likely to be found in these deep execution paths and some vulnerabilities are only possible when certain conditions are met (i.e. the program must be in a deep state) [7].

The experiments showed that the approach can be effective, but they may not provide results that are representative of real programs - vulnerabilities were intentionally added to the experimental programs, rather than accidentally being introduced [7]. More testing is needed on large, real world software in order to prove its effectiveness and reveal any flaws.

### Discussion (this will be moved into the discussion section of the final report)

(NOTE: This paragraph refers to the summary in the second homework exercise, which has not been included in this document.) The proposed process by Zhang et al can be combined with the previously mentioned risk analysis and threat modelling approach in order to prioritise the testing of execution paths that involve high risk operations. Prioritised paths can then be tested more thoroughly in order to increase the probability of finding hidden vulnerabilities.

Zhang et al intend to extend the SimFuzz tool to work with binary executables and web applications [7]. The application of the tool to the web could be particularly useful especially in a language such as JavaScript which uses weak typing - a feature that can cause unexpected issues to arise. The tool could also be extended to simulate network connectivity quality, in order to test states that can only be reached by corrupting incoming data. Potentially it may be possible to simulate concurrent systems which are hard to test thoroughly and prone to unpredictable behaviour at runtime - therefore it may be worth testing these systems for vulnerabilities.

### References

[7] D. Zhang et al, "SimFuzz: Test case similarity directed deep fuzzing,"

Journal of Systems and Software, vol. 85, pp. 102-111, Jan. 2012.