ECE455 – Final Project Proposal

Jonathan Lam and Daniel Tsarev

2021/11/04

1 Interests

We both have found an interest in programming languages and verification. In particular, both of us took the Compilers course last semester with Prof. Hakner, and we both have been looking at functional programming languages like Haskell (Dan is interested in its use in cryptocurrency, while Jon is interested in its use in formal analysis). Jon is also currently taking an I.S. in formal analysis. We are also both interested in LLVM.

2 Proposal

We would like to develop a (very) simple static analyzer for (blatant) buffer overflow issues in a language like C. This would involve a dataflow analysis and an interprocedural analysis to track the common causes of buffer overflow (e.g., unchecked-length buffer functions such as `strcat`) and `printf` attacks. The analysis would also reveal information about possible attack vectors, and would be conservative in the analysis of unknown (i.e., library) functions.

For the implementation, we may use a tool such as clang or WLLVM to parse a C file to LLVM, to avoid manual parsing. The analysis may be conducted in a functional language such as OCaml or Haskell, which are often used in program analysis scenarios. (This would also allow both of us to learn about LLVM and apply functional programming.)

3 Justification

One may wonder if this exercise is useful, since static analyzers can do this already and to a better degree, and modern OSes already have protection against these common attacks. Despite this, it would be a tremendous learning experience that ties together our experiences from this course, formal analysis, and compilers.

4 Project deliverables/Evaluation

The end result is a tool that, when run on a C program, will be able to detect and report simple buffer overflow vulnerabilities. Additionally, this tool should be able to have report some additional information about the vulnerability, such as the possible sizes of the buffer(s), the vulnerable function(s), the calling context, etc.

The end result will be limited in many ways – it may not be able to detect a vulnerability such as the url_decode function, which ranges over the input buffer without considering the length of the destination buffer. Essentially, what we hope to do is look for usages of known vulnerable functions and trace information that may be useful for protecting against or exploiting buffer overflow vulnerabilities. If there is time, the same may be adapted for detecting simple printf vulnerabilities.

To evaluate this project, we will test the program on the Lab 1 program, and perhaps other C programs with known vulnerabilities (to check for detecting of known vulnerabilities) or programs with unknown vulnerabilities (to detect unknown vulnerabilities).

5 Timeline

The following dates are approximate proposed deadlines for major steps of the project. The first three weeks will also be onboarding time for learning LLVM, OCaml, and program analysis techniques.

- Nov 4 – Finish final project proposal.
- Nov 11 – Implement clang/WLLVM parser, understand basics of LLVM and create debugging framework (design helper functions to print out AST/CFG)
- Nov 18 – Design analysis framework (e.g., which abstract variables to track)
- Nov 25 – Turkey Day, begin implementation of analysis framework (implementing the lattice, tracking abstract variables)
- Dec 2 – Continue implementation ....
- Dec 9 – Finish implementation, work on presentation
- Dec 16 – Finalize presentation

6 References/Materials

6.1 LLVM

We will use a LLVM IR (generated by clang or a similar tool) as the representation of a program. The language of choice for our implementation is OCaml.

- LLVM tutorial: Kaleidoscope. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend

6.2 OCaml

We are both exploring functional programming in OCaml.

- Real World OCaml. https://dev.realworldocaml.org/
- OCaml Llvm module tutorial. https://www.wzdftpd.net/blog/ocaml-llvm-01.html
- OCaml Llvm module documentation. https://llvm.moe/ocaml/Llvm.html

6.3 Program Analysis

The first few chapters of this book provide a method (data-flow analysis) for performing an analysis on code using a lattice data structure. In our case we will customize the dataflow analysis to track buffers and vulnerable function calls.

- Program Analysis. https://cmu-program-analysis.github.io/2021/resources/program-analysis.pdf

6.4 Static analysis for buffer overflows

The following papers are used to get an idea for previous buffer-overflow detection using static analyzers, but they are probably too complex to follow in-depth.

- Wikman, Eric C. Static Analysis Tools for Detecting Stack-Based Buffer Overflows. NAVAL POSTGRADUATE SCHOOL MONTEREY CA MONTEREY United States, 2020.
- Shahriar, Hossain, and Mohammad Zulkernine. "Classification of static analysis-based buffer overflow detectors." 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion. IEEE, 2010.
- Zitser, Misha, Richard Lippmann, and Tim Leek. "Testing static analysis tools using exploitable buffer overflows from open source code." Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering. 2004.