Resolving Input Validation Vulnerabilities by Retracing Taint Flow Through Source Code

Justin Smith
Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27606
Email: jssmit11@ncsu.edu

***Abstract*—Various security-oriented static analysis tools are designed to detect potential input validation vulnerabilities early in the development process. To verify and resolve these vulnerabilities, developers must retrace problematic data flows through the source code. My thesis proposes that existing tools do not adequately support the navigation of these traces. In this work I will explore the strategies developers use to navigate tainted data flow in source code and work toward solutions that support successful strategies.**

I. Introduction

Security defects pose a pressing threat to modern software systems. Compared with other types of defects, security defects are more likely to impact company profits [1]. One class of particularly damaging defects are *input validation vulnerabilities*. According to Tsipenyuk and colleagues, input validation vulnerabilities cover four of the top ten most critical security risks [2].

There are a variety of tools designed to detect potential input validation vulnerabilities early in the development process, even before the code executes. Detecting these defects early is important, because long-lingering defects are more expensive to fix [3]. There are two broad types of tools that can detect these defects statically. One type, know as static taint analyzers, detect defects by tracking the propagation of user input through a system. The other approach, pattern-based static analysis, detects potential defects by matching the code to predefined criteria.

Regardless of the underlying defect detection approach, developers play a key role in resolving security defects. Unfortunately, analysis tools do not effectively support developers in resolving the defects they detect [4]. Consider, for example, how Joelle – a persona inspired by participants in my previous study [5] – attempts to resolve a defect detected by Find Security Bugs (FSB).[1] Initially, she reads the notification generated by FSB; potentially tainted data has reached a sensitive context. Now she must determine whether that data has been properly sanitized. She tries to find a tool to help her retrace data back to its source, eventually settling on the navigation tools built into her integrated development environment. The tools help Joelle navigate her program's call graph, but she must manually track the flow of data across calls. She finally locates sanitation methods along the few program paths she retraces. Tired of tracing data through obscure corners of the system, she decides to ignore the warning. Two weeks later, an attacker executes a SQL injection attack along a vulnerable path she did not consider, causing her company to loose valuable data.

Though different tools might have helped Joelle navigate more easily, little is known about which tool interfaces support trace navigation most effectively. Some tools provide little support by simply generate textual descriptions of problematic data flows, leaving developers to use their choice of general purpose navigation tool. At the other end of the spectrum, other tools provide an array of various custom navigation interfaces, including code annotations and visualizations.

In my work, I will explore the following research questions:

- **RQ1**: What strategies do developers use to successfully resolve input validation vulnerabilities?
- **RQ2**: To what extent do the interfaces of existing static analysis tools support these strategies?
- **RQ3**: How can static analysis tools better support effective strategies?

II. How do Developers Resolve Security Defects?

My previous work identified developers' information needs while resolving security defects using a pattern-based static analysis tool [5]. I observed ten developers performing four tasks, three of which required them to resolve input validation vulnerabilities. Some of the vulnerabilities participants encountered were false positives. Participants in this study used FSB, which locates suspicious data sources and sinks, as well as provides textual notifications. As part of determining whether the vulnerabilities were false positives, participants investigated code surrounding suspicious data sources and sinks, using general purpose navigation tools and techniques.

I extended the study and performed a second round of analysis. By doing so, I was able to identify the strategies developers used to acquire the information they need.

Applying these analyses to the three input validation vulnerability tasks produced two relevant results: (1) Retracing the flow of data by navigating source code was important part of participants' resolution strategies, and (2) participants struggled to select and use program navigation tools to retrace the flow of data through source code.

[1] http://find-sec-bugs.github.io/

978-1-5090-0252-8/16/$31.00

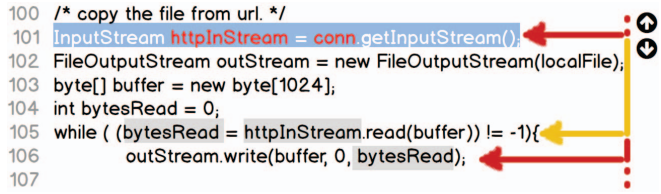

Fig. 1: Trace navigation interface: (1) Code highlighting helps developers locate and trace tainted data. (2) Yellow arrows draw developers' attention to the steps where the tool used a potentially inaccurate heuristic. (3) Arrows (⬆⬇) enable navigation between relevant lines and across files.

Building on these results, I began exploring designs for tools that better support developers in retracing dataflow through their programs. One tool, Flower,[2] is designed to enable structural program navigation with a minimal interface. Another design, depicted in Figure 1, helps developers navigate individual taint traces and quickly identify tainted data. It also helps developers hone in on locations where the taint tracking tool used heuristics in its analysis. Those locations are important to developers, because they may require additional verification.

III. Proposed Evaluation Plan

Thus far, I have studied developers' defect resolution strategies (**RQ1**) while using a tool that provided little support (**RQ2**) for navigating taint traces. Recognizing the importance of navigating taint traces, I have begun to explore designs for tools that might help developers perform this task more effectively. To further inform the design of such tools and complement my initial study, I propose to evaluate the existing tools that do attempt to support trace navigation.

First, I will inspect existing taint trackers that more or less support the navigation of taint traces, specifically examining their interfaces (not their underlying defect detection algorithms). This process will produce a categorization of interface elements. Next, I will select a representative subset of the tools covering all the categories and evaluate those tools in a user study. By observing users interacting with the tools, I will gain insight into which interface elements most effectively help developers thoroughly navigate taint traces and accurately resolve input validation vulnerabilities (**RQ1/RQ2**).

Based on my findings, I intend to derive design guidelines and apply those guidelines by either implementing a new navigation tool or modifying an existing taint tracking tool (**RQ3**).

IV. Related Work

Existing literature has primarily focused on improving the algorithms that detect input validation vulnerabilities. Researchers have provided tools and techniques for detecting new vulnerabilities [6] in new contexts [7]. These techniques are assessed on how accurately and efficiently they detect vulnerabilities [8]. However, the interfaces for these tools have not been systematically evaluated. My work aims to amplify the impact of existing tools by reviewing and improving their interfaces, with the end goal of assisting developers in resolving the defects the tools detect.

Program navigation is a critical task for resolving input validation vulnerabilities and a well studied research area. Several approaches enable developers to navigate source code structurally, between arbitrary points in a program (e.g. [9], [10]). Unlike other tasks, retracing taint flow is unique in that the developer must navigate between two fixed program points (potential sources of data and sensitive sinks). In this work, I will tailor applicable findings from the program navigation community to better suit the task of retracing taint flow.

V. Conclusion

Rethinking the way tools present taint traces to developers, I propose to continue to perform a systematic evaluation of the static analysis tools that detect input validation vulnerabilities. The algorithms and analysis techniques underlying these tools have already been throughly examined. However, I contend that by bettering these tools' interfaces, we can help developers leverage the advances in analysis techniques to resolve vulnerabilities more accurately and efficiently.

VI. Acknowledgments

I would like to thank Dr. Emerson Murphy-Hill for his advice and the rest of the Developer Liberation Front for their support. This material is based upon work supported by the National Science Foundation under grant number 1318323.

References

[1] H. Chen and D. Wagner, "Mops: an infrastructure for examining security properties of software," in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 235–244.

[2] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: A taxonomy of software security errors," *Security & Privacy, IEEE*, vol. 3, no. 6, pp. 81–84, 2005.

[3] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[4] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 672–681.

[5] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 248–259. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786812

[6] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 171–180.

[7] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages." in *USENIX Security*, vol. 6, 2006, pp. 179–192.

[8] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 97–106.

[9] T. D. LaToza and B. A. Myers, "Visualizing call graphs," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sept 2011, pp. 117–124.

[10] P. Anderson and M. Zarins, "The codesurfer software understanding platform," in *13th International Workshop on Program Comprehension (IWPC'05)*, May 2005, pp. 147–148.

[2] tinyurl.com/flowerTool