Related Work

Scott Kuroda

March 11, 2012

As software has taken on a more prevalent role in society, the security and robustness of applications has become an increasingly important area of study. Though there are those that choose to specify in the domain of computer security, it is a challenge that every developer in industry should be concerned with. Today, a large amount of work goes into ensuring the reliability, stability, and security of application code in industry. However, it is often an afterthought in academics.

Because of the already high demand on students in an academic setting, the challenge that presents itself is how to incorporate secure coding practice into existing courses. One way to do this is to incorporate similar tools and techniques used in industry to validate code, in an academic setting. By doing this, the code that students already submit in there course work will be scrutinized to some extent by security analysis. Though this will not necessarily generate experts in the field of security, it will, at a minimum, expose students to a variety of potential security flaws in their code.

1 Existing Code Analysis

Code analysis in industry is done in a number of ways. Of particular interest are static analysis methods. These offer lower computational overhead, compared to dynamic testing. It would allow for more rapid feedback on submitted code. The following sections will discuss static code analysis methods, its accuracy, and service oriented testing architecture.

1.1 Static Code Analysis

A popular method for analyzing code is through static code analysis. Static code analysis involves analysis of code, in a variety of methods, without actually executing any code. Static code analysis can be broken into three distinct stages. The first involves parsing the code. Because code is never actually run, this parsed code is then used to create a representational model that the tool understands. The tool then provides some analysis on this model, finding a variety of bugs and errors in the submitted code. The types of analysis done vary widely, and are beyond the scope of this paper. Though tools do find a variety of issues, they often have a large number of false positive and false negative results. This can be partially mitigated as some tools allow for the incorporation of end user generated rules into the analysis set. [1]

1.2 Java Code Analysis

One particular study that involved combining a variety of Java static code analysis tools [6]. In this study, Ware and Fox use 8 existing static code analysis tools against a series of test cases. These test cases together provided 115 unique violations. In this study, violations ranged from security violations to simple coding standards. To provide further analysis, there was some overlap in the test cases provided. That is, a single test may encompass 1 to many unique violations.

After running all the test cases through the analysis tools, Ware and Fox found that the tools performed in a less than impressive manner. As a whole, the tools successfully identified 50 of the unique violations introduced. Also, no single violation was found by all of the tools. In addition, the tools also had a variety of false positive results. Though the set of tools only found 50 of the 115 violations, the most violations an individual tool found was 30. From these results we can draw two distinct conclusions. First that static

code analysis alone is not enough for code analysis. Second, though the set of tools together did not provide great results, as a set they found almost twice as many unique violations than any given tool alone. [6]

1.3 Static Code Analysis and Inferred Information Flow

As previously discussed, static code analysis has its limitations. Beyond simple static code analysis methods is dynamic analysis. This involves things like tainting code. However, this is very computationally intensive. Though this may, at some point be incorporated into an analysis test suite, at the moment the incorporation of dynamic analysis methods is beyond the scope of this paper. [3] To help bridge the gap between dynamic and static analysis methods, [4] proposes using static code analysis to infer the information flow. If done properly, this would provide another avenue for code analysis, with a relatively low amount of overhead.

In [4], the authors propose a method to analyze explicit data flows. These flows are the results of variable assignments. Not covered by this tool, but covered by many dynamic tools in implicit flows. Implicit flows are the result of some conditional statement. For example, if a variable i' were to be defined as j if j is greater than some value. To complete this analysis, this method employs several different distinct analysis methods, Point-to and fragment analysis. The results of these analysis methods are then used to create the inferences. To validate their work, the authors used the described static analysis method on a set of Java components. The tool found a variety of issues in the existing code base. The authors argue that the results are both precise and practical. Though this analysis method only infers on explicit flow, it is still a step in further utilizing static analysis methods beyond some of the current static code analysis tools. This type of tool, combination with the other existing tools would allow for a more thorough final report to be generated. Of course, even the combination of tools, as previously discussed, may not catch everything. But it would give a more complete picture to the end user.

1.4 Testing as a Service

In addition to work being done regarding analysis methods, another related area is service oriented systems. These systems leverage cloud computing services. By utilizing this architecture, a given system is much more widely available. In addition, the pure computational load needed to perform a given task is removed from a users machine, and pushed to a much more dynamic architecture, the cloud.

This architecture is generally referred to as Testing as a Service (TaaS). As previously stated, this moves the testing of software from local machines, or in house testing by a company, and moves it to a cloud based service. Because code analysis methods can be computationally intense, most developers would rather not slow their development process with these additional checks. However, with a service oriented testing architecture, a developer could continue development, as they test their existing code. Because this testing would not occur on their machine, they would take a much smaller, if any, performance hit. By testing continuously, and early on in the development process, it may be possible to identify bugs quickly. This would allow for bugs to be fixed early on in the development cycle when, arguably, a given problem will be easier to fix. [2]

However, cloud based systems are not without their challenges. Thoroughly testing software often requires some expertise in the domain. With this in mind, TaaS is not the end all solution for testing. Rather, it would handle a certain set of test, allowing in house testers to focus more on implementation details. In addition, as with any cloud based service, the code being tested, as well as the results must be kept secure. In [5], companies interviewed stated that they would need a certain level of guaranteed security. Because of how their code base may integrate with the service, an attacker may be able to work from the testing service, back into the companies system. With these issues in mind, cloud based testing, in industry, may not be the end all solution. However, it can provide a level of testing, allowing in house testers to be much more focused.

Though not necessarily required by all systems, depending on the scope, some TaaS systems allow for individual configuration to test a set of code. This would allow additional flexibility for an end user to take advantage of while leveraging the service based system. This type of system could be leverage through creating a set of common rules that can be utilized across various submitted programs. Then, more detailed testing can be done on a program by program basis. [2]

References

[1] D. Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[2] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 155–160, New York, NY, USA, 2010. ACM.

[3] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[4] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, pages 50–56, New York, NY, USA, 2008. ACM.

[5] L. Riungu, O. Taipale, and K. Smolander. Software testing as an online service: Observations from practice. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 418 –423, april 2010.

[6] M. S. Ware and C. J. Fox. Securing java code: heuristics and an evaluation of static analysis tools. In *Proceedings of the 2008 workshop on Static analysis*, SAW '08, pages 12–21, New York, NY, USA, 2008. ACM.