# Code Pulse: Real-Time Code Coverage for Penetration Testing Activities

Hassan Radwan, Kenneth Prole
Secure Decisions Division
Applied Visions, Inc.
Northport, NY, USA
{Hassan.Radwan, Ken.Prole} @securedecisions.com

***Abstract*— A continuous challenge facing software penetration testers is ensuring adequate coverage of a target application. Many dynamic application security testing tools and manual pen-testing techniques test only part of the exposed code base, leaving much of the attack surface untested. A purely black box approach, used by most DAST tools, makes it almost impossible to accurately identify how much of the attack surface of an application was tested for penetration during assessment. Glass box testing techniques, as described in this paper, significantly improve the insight that penetration testers have into the coverage and makeup of the applications they are targeting. This paper reports on DHS-funded research which resulted in an innovative open source tool called Code Pulse that provides real-time code coverage for pen-testing Java web applications. Code Pulse leverages the Java instrumentation libraries to provide a real-time glass box perspective of method calls as they are exercised during security testing activities. While the concept of glass box testing is not new, Code Pulse delivers a novel real-time approach to the challenge while maintaining a tool-agnostic approach. In this paper we will outline the code coverage challenges facing penetration testers, describe the state-of-the-art in software assurance code coverage, the innovative aspects of our approach and its contribution to the state-of-the-art, the feedback we have received since releasing it as an Open Web Application Security Project (OWASP) pen-testing application in May 2014, and the planned improvements to Code Pulse.**

***Keywords: penetration testing, code coverage, instrumentation, application security, software assurance, software visualization, cybersecurity, software testing, DAST, OWASP***

## I. INTRODUCTION

Penetration testing is increasingly becoming a cornerstone process in securing applications prior to operational deployment. Penetration testers, also know as ethical hackers or white hat testers, have a variety of techniques and tools in their vulnerability discovery toolbox. Some testers rely on manual probing of a target system's attack surface, whilst others leverage the ever-increasing availability of automated dynamic application security testing (DAST) tools. The majority, however, rely on a combination of both manual and automated approaches to identify vulnerabilities – such as the ones listed in the OWASP Top 10 [1] – before the not-so-friendly attackers exploit them resulting in the types of data breaches that are unfortunately ever prevalent in today's news cycle.

From a penetration tester's perspective, the system being probed for vulnerabilities is opaque. In the case of a web application, a tester might be aware of some of the site's entry points via link navigation, however, that is a far cry from understanding what the complete attack surface is or which parts of the system internals are exercised with the varying test inputs. This is why this type of testing is colloquially known as black box testing. There is little to no insight into the makeup of the system being tested and which parts are accessible to the outside. The testing interaction is thereby reduced to a series of test inputs that are based more on best practices and instinct than direct knowledge of a system or an effective feedback loop. The resulting process is akin to a game of "Who am I?" that we've all played on road trips, albeit significantly more sophisticated. This by no means is meant as a shot at the incredibly valuable results yielded by effective penetration testers. It is meant to demonstrate to the reader the degree of the challenge facing our ethical hackers.

One might argue that attackers face the same challenges. However, an important distinction is that whilst the goal of an attacker is to find just a single exploitable vulnerability, the goal of our friendly penetration tester is to ensure there are none to be exploited across the entire system. Test coverage is therefore a critical measure for the penetration testing process. Gaining insight into which parts of an application remain to be tested and which parts of the system react to test inputs provide testers with the information they need to create an effective feedback loop for the testing process. Understanding coverage is important not just to better guide the testing process for individual assessments, but also for getting a broader coverage perspective across all tools and techniques, helping identify the coverage overlaps and more importantly the coverage gaps. Unfortunately given the inherent challenges of black box testing, getting an accurate measure of the test coverage is a difficult and sometimes even impossible task.

Getting the insight into the makeup and coverage of an application transforms the process from black box to *glass box testing* – a runtime testing process where the internal composition and behavior of the test application are observable. We've contributed Code Pulse, an open source glass box tool focused specifically at identifying the code coverage of penetration testing activities in real-time. In this paper we will describe the glass box approach taken by Code

Pulse, a sample usage scenario, and discuss the benefits and challenges of this work.

## II. System Design

Two key decisions were made in the early phases of our work. The first was to scope our efforts to identifying coverage data for web applications, and the second was to limit our coverage identification to Java applications. These requirements pointed our attention at a large user population that would benefit from the Code Pulse solution while focusing our attention on an achievable goal within our constraints. As such, the discussion in the remainder of the paper, while we believe it to be applicable to a broad range of software, will be colored by these two decisions.

In designing the Code Pulse system, there were two primary concerns: how best to identify the coverage data; and how best to communicate it to the users.

### *A. Coverage Identification*

A challenge when considering the nature of relevant coverage data is the disparity between the viewpoint of a penetration tester and the actual source code composition of an application. In the case of web applications, penetration testers are interfacing with the various web pages that form the entry points for the application. However, the sitemap and its corresponding URL-set does not necessarily translate directly to the basic code building blocks (classes, methods, etc.) that ultimately process the inputs being passed by the penetration testers. This is largely dependent on the platform and web frameworks in use and more often than not a URL routing mechanism will exist to dispatch URL requests to specific code dependent on the provided inputs. Consider a hypothetical web application with a URL to generate a report with the format (PDF or XML for instance) specified as an input parameter to the same URL. The PDF and XML report generation will result in different code being called, despite the same URL entry point.

Identifying the full set of URLs forming the attack surface of a web application is a non-trivial problem due to a significant disparity in URL route dispatching between web frameworks. For Java alone, a quick cursory search reveals that there are at least 35 established web frameworks [2], each with its independent mechanism for URL handling. But these same 35 frameworks ultimately all call Java code elements regardless of their built-in URL routing mechanism. Therefore creating a set of URL coverage monitors, one per supported web framework, was a non-starter due to the lack of scalability and specificity of the solution. A more generic approach is to observe the code elements as they are called for each request.

In addition, taking a step back to reflect on the penetration testing process, after vulnerabilities are detected the next step in the process is to notify the development team of the issues.

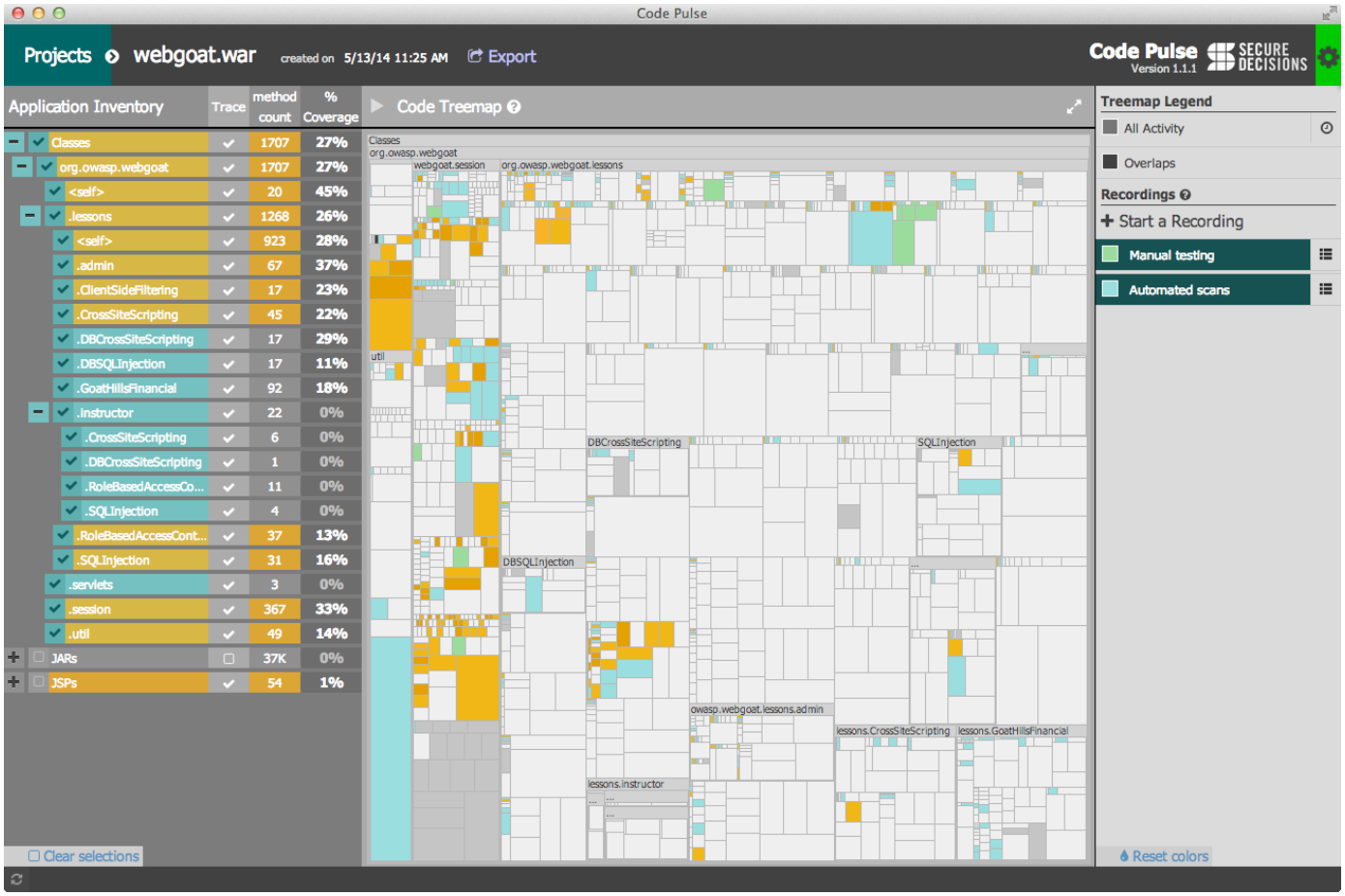


Figure 1. Code Pulse real-time highlighting of recently called Java packages and methods for an application under testing

Understanding the coverage data at the source code level gives developers the best data on which parts of the application are impacted by the identified vulnerabilities as they determine the best remediation recourse. Therefore, from a developer user perspective the most effective coverage data would represent the relevant source code entities that need their attention.

For these reasons, we decided to focus the Code Pulse coverage identification at the source code level, providing a tool-agnostic approach at coverage collection. This does not nullify the validity of URLs as coverage data, and certainly from a penetration tester user's perspective, the ideal data would identify both the source code coverage along with the URL site map of an application and its expected inputs. We aspire to reconcile that in future efforts.

### B. *Coverage Communication*

Identifying the timing for communicating the coverage data was a key design challenge. Collecting the data and presenting it in summary form at the end of the testing process does not empower penetration testers with quick adjustments of their testing activity based on application response. On the other hand presenting coverage in real-time is also a challenge due to the sheer volume of data. A single web request can result in tens or even hundreds of thousands of methods calls. Overwhelming the user with too much coverage information is arguably more harmful than not presenting it at all since the abundant data results in a cognitive load that takes away from the testing process instead of enhancing it.

Visualizations have been used to great effect to summarize large volumes of data in a meaningful manner. When facing large data volumes or data of a complex nature, we, as humans, process data visually more effectively than text-based alternatives [3]. In addition when used in combination with relevant data filters and interactions, we excel at identifying data patterns visually. The effectiveness of presenting a software hierarchy in a treemap visualization [4] has been repeatedly [5] demonstrated over the years [6] and is increasingly becoming a familiar visualization within commercial software analysis tools [7].

The conclusion of our explorations on how to communicate the coverage data was to present it as quickly as possible to the penetration testers, and to do so leveraging visualizations to improve the data readability.

## III. Approach

Our efforts in Code Pulse had the ambition of turning the black box perspective facing penetration testers into a glass box one. The Code Pulse approach to achieve this glass box perspective is to leverage software instrumentation to represent the code coverage visually in real-time to penetration testers whilst conducting their tests. There are two top-level components to the Code Pulse system architecture: the instrumentation component responsible for monitoring code coverage of target applications at runtime; and the front-end user interface and visualization responsible for representing the coverage information in an easily digestible manner.

### A. *Instrumentation*

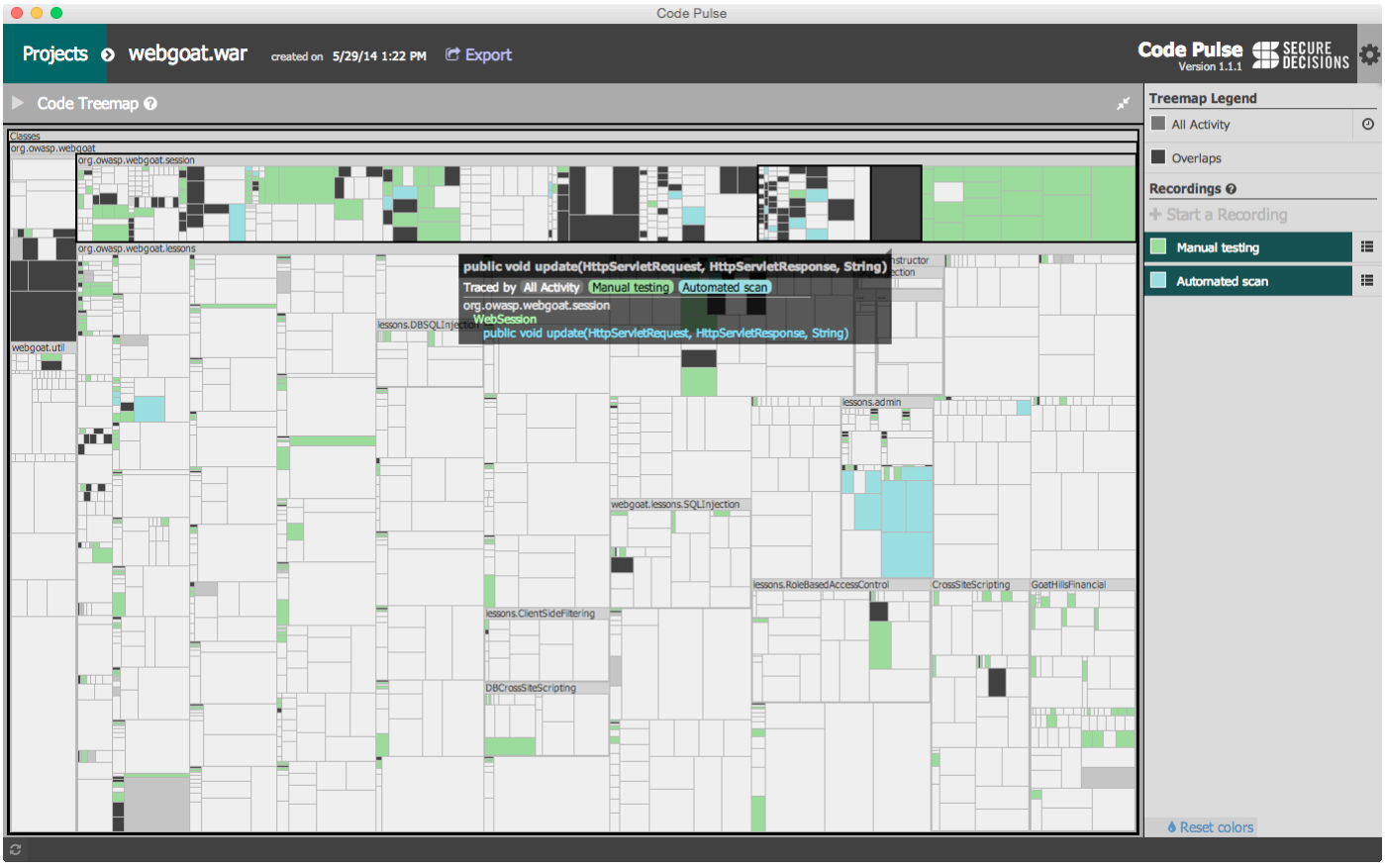


Figure 2. Highlighting the distinct and overlapping coverage for a manual penetration test and an automated one

To identify coverage data Code Pulse leverages Java's instrumentation libraries and uses an agent-based approach. Using a Java Virtual Machine (JVM) directive, the Code Pulse agent is loaded prior to loading any other libraries and classes. Once initialized, the Code Pulse agent starts monitoring the JVM class loaders and injects monitoring bytecode in the classes of interest. By default, third-party libraries are not instrumented in this phase, although users have the option to override that and select libraries, or even specific packages within them for coverage monitoring.

A key constraint of the instrumentation component is the requirement to have minimal impact on the resources of the target application. To satisfy that constraint, the system had to be designed to perform minimal work in the same execution context as the target application and instead rely on another context to process the coverage data. Therefore the monitoring component was set up into two distinct pieces using a client / server model. The agent (client) runs in the same JVM as the target application that will be tested. As the target application runs, the agent listens in on the execution and sends the coverage information to the server for processing and storage. This high-level separation is shown in Figure 3. The separation in responsibility between the observer and data is key to limiting the impact on the traced application and reduce the footprint of the agent to the lowest possible condition. Note that nothing prevents the agent and server from running on the same machine, and in fact is anticipated to be a frequent use case.

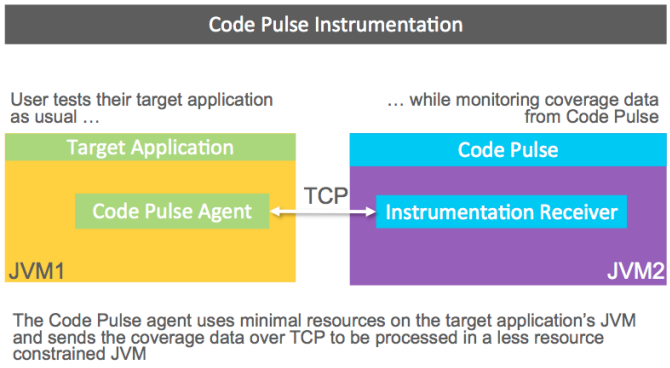


Figure 3. Overview of the instrumentation client/server model

A number of tests were conducted on the performance impact of the instrumentation on target applications. The results varied significantly depending on the nature of the application. Overhead was significant for graphics intensive applications conducting many sustained rendering method calls. Conversely for web applications where execution patterns followed short quick bursts, the overhead, while still measurable, was not noticeable in normal usage. The average instrumentation overhead range for web applications using our current implementation is a slowdown factor of about 1.5-2.5.

### B. *Visualization*

To represent the coverage data in Code Pulse a treemap visualization was used. Two types of nodes were represented in the treemap. Java package nodes were shown as slim labels and only served as a point of reference within the visualization. The other nodes all represented either Java classes or Java Server Pages (JSP) files. These node types were the focus of the visualization and were sized by bytecode instruction count. In a default state, prior to any coverage activity, nodes are colored in a light grey color. As coverage activity occurs for a method or JSP file, the node shading changed to indicate that it had been covered during the testing activity.

The treemap served a variety of purposes. The first was real-time activity highlighting. As methods were called, they were highlighted in real-time within the treemap to indicate to the user which parts of the application their testing activity impacted. An example of this real-time highlighting can be seen in Figure 1 with the orange shaded nodes. The second purpose was to serve as the persistent coverage indicator for tested methods and files. As methods are called as a result of testing activity, its color indicator changed from the default light grey color. Finally, the treemap was used to drive the coverage overlap analysis. Coverage data in Code Pulse can be segmented using labeled markers. When multiple markers are selected, the treemap changes the coloring of the nodes to reflect which ones were covered within a single segment, and which ones had overlapped coverage. The distinct coverage for a manual penetration test (green) and an automated one (blue) along with the shared coverage overlaps (dark grey) can be seen in Figure 2.

## IV. Coverage Scenario

To test the utility of Code Pulse at representing coverage data we set up a timed test to identify how much test coverage can be improved using a DAST tool. We limited ourselves to 20 minutes of testing and used a community test application designed explicitly with known vulnerabilities to test DAST tools. The test application was Web Application Vulnerability Scanner Evaluation Project (WAVSEP) [8] and the DAST tool used to test it was OWASP Zed Attack Proxy (more commonly known as ZAP) [9].

Within the allotted 20 minutes, three separate scans were conducted, each progressive one with additional tuning applied in reaction to the coverage results from the previous scan. The purpose of the tuning is to provide ZAP with a better understanding of the test application. The result of the three scans is summarized in Figure 4.

The first scan was conducted with no configuration to the DAST tool other than pointing it at the host and port number to test. WAVSEP's sitemap is intentionally obscure, so it's not surprising that the tool only found the main index page (the single blue square in the top left of the 1st scan in Figure 4).

The second scan was conducted after seeding ZAP with the key top-level WAVSEP entry pages. Despite the seeding process only a small subset of the pages was uncovered by ZAP's discovery mechanisms and was tested. The covered pages are once again colored in blue.

The third and final scan was conducted after further seeding of the sitemap. With the emergence of visual patterns in the 3rd scan treemap results from Figure 4 several quick observations can be derived:

- A large block of the code, in the left portion of the treemap, remains undiscovered and untested.

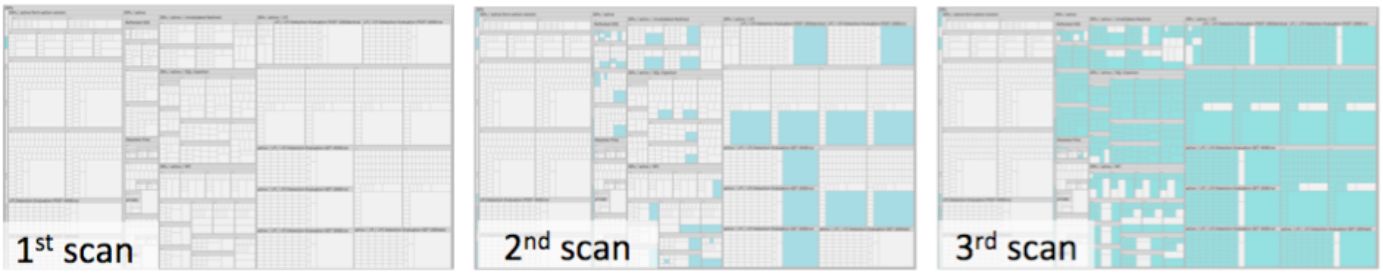


Figure 4. Progressive tuning of test coverage for an automated penetration test

- Despite getting good test coverage, there are several isolated grey nodes in the sea of blue in the right portion of the treemap.
- The middle section of the treemap was fully covered.

A number of conclusions can be drawn from this exercise:

1. The turnaround between DAST scans was incredibly quick with Code Pulse. Visually processing the coverage data after each scan was a quick exercise that made inadequate sitemap configuration immediately obvious.
2. Small coverage gaps as seen in the 3rd scan would have been very challenging if not impossible to identify via manual processing of the test logs. With the treemap visualization a quick scan revealed coverage insight that would have most likely remained unnoticed otherwise.
3. The treemap visualization is a great communication tool. The whole testing process was summarized in three screenshots showing the coverage progression and coverage gaps at the conclusion of the test scenario.

## V. Related Work

Shay Chen, an information security researcher and blogger, has published several comparison studies of DAST tools [10]. There exists over 60 open-source and commercial DAST tools aimed at identifying security flaws in web applications. The majority of these tools are purely black box, meaning by definition they have no insight into the internals of the application under test, and therefore have no way to provide code coverage insight. A few commercial tools use glass box techniques similar to Code Pulse to make the DAST scanning more intelligent. Most notable are IBM's AppScan [11] and HP Fortify's WebInspect Real-Time [12]. These solutions however do not offer code coverage metrics nor interactive visualizations, and are tied to *their* DAST tool, unlike Code Pulse, which is tool agnostic.

Several open-source and commercial tools exist that focus on providing code coverage information [13]. Some of these include Atlassian Clover [7], JaCoCo [14], and JCov [15]. These solutions are used by developers to improve the code coverage of unit tests for the purposes of improving the quality of an application. This differs from the objective of Code Pulse where the focus is on software assurance code coverage for use by penetration testers. The other key difference is none of the existing solutions offer real-time code coverage. The output of these tools is provided after the coverage analysis is complete. There is no real-time interactive feedback loop as provided in Code Pulse, which allows penetration testers to alter their testing technique based on coverage results and compare coverage between multiple tools.

## VI. Conclusion

In this paper we presented a new approach for glass box testing that marries real-time instrumentation with real-time visualization to improve penetration testing coverage. This technique supports manual and automated testing approaches, and is tool-agnostic. We also presented the resultant tool, Code Pulse, released as an open source tool and joins the diverse project inventory of the Open Web Application Security Project (OWASP). It is freely available for download and extension at http://code-pulse.com. Whilst the initial reception for the tool and approach has been very positive, the feedback we've received thus far has helped us identify a number of future directions for improvement.

First, we will investigate adding support for additional platforms and language. The Java-based instrumentation limits the utility of this tool to Java applications. .NET support is currently high on our priority list, although we've also received requests for a variety of dynamic languages.

Second, we will investigate increasing the precision of the coverage instrumentation from its current method level-of-detail to block-level coverage. Although providing coverage insight at the current precision has proven to be valuable, being able to distinguish between statements that were executed vs others that were not within a method due to conditional logic or other flow control mechanisms would further increase the utility of the resulting tool.

Finally, to better support increased levels of precision and other use cases, we intend to investigate improving the instrumentation performance.

## ACKNOWLEDGMENT

This work was partially funded by the Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD), BAA 11-02 and Air Force Research Laboratory information Directorate via contract number FA8750-12-C-0219.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Department of Homeland Security, Air Force Research Laboratory or the U.S. Government.

## REFERENCES

[1] OWASP Top Ten Project, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[2] Wikipedia listing of Java Web Frameworks, http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#Java.

[3] S. Card, J. Mackinlay, B. Shneiderman, “Readings in information visualization: using vision to think”. Morgan Kaufmann Publishers. 1999.

[4] B. Shneiderman, "Tree visualization with tree-maps: 2-d space-filling approach." ACM Transactions on graphics (TOG) 11.1 (1992): 92-99.

[5] G. Langelier, H. Sahraoui, P. Poulin, "Visualization-based analysis of quality for large-scale software systems." Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005.

[6] R. Wettel, "Visual exploration of large-scale evolving software." Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on. IEEE, 2009.

[7] Atlassion Clover , https://www.atlassian.com/software/clover/overview.

[8] Web Application Vulnerability Scanner Evaluation Project, https://code.google.com/p/wavsep/.

[9] OWASP Zed Attack Proxy, https://code.google.com/p/zaproxy/.

[10] S. Chen, Comparision of open-source and commercial web application scanners, http://sectoolmarket.

[11] R. Saltzman, A. Sharabani, “Glass box testing : Thinking inside the box”, http://public.dhe.ibm.com/common/ssi/ecm/en/raw14283usen/RAW14283USEN.PDF.

[12] HP WebInspect, http://www8.hp.com/us/en/software-solutions/software.html?compURI=1341991#.VFU29cmnc8Y.

[13] K. Alemerien, K. Magel, “Examining the Effectiveness of Testing Coverage Tools: An Empirical Study,” International Journal of Software Engineering and Its Applications, Vol.8, No.5 (2014), pp.139-162.

[14] JaCoCo, http://www.eclemma.org/jacoco/.

[15] JCov , https://wiki.openjdk.java.net/display/CodeTools/jcov.