Visual Analysis of Code Security

John R. Goodall
Secure Decisions division of Applied Visions, Inc.
6 Bayview Ave.
Northport NY 11768
631-754-4920
JohnG@SecureDecisions.avi.com

Hassan Radwan, Lenny Halseth
Applied Visions, Inc.
6 Bayview Ave.
Northport NY 11768
631-754-4920
{ HassanR, LennyH } @avi.com

ABSTRACT
To help increase the confidence that software is secure, researchers and vendors have developed different kinds of automated software security analysis tools. These tools analyze software for weaknesses and vulnerabilities, but the individual tools catch different vulnerabilities and produce voluminous data with many false positives. This paper describes a system that brings together the results of disparate software analysis tools into a visual environment to support the triage and exploration of code vulnerabilities. Our system allows software developers to explore vulnerability results to uncover hidden trends, triage the most important code weaknesses, and show who is responsible for introducing software vulnerabilities. By correlating and normalizing multiple software analysis tools' data, the overall vulnerability detection coverage of software is increased. A visual overview and powerful interaction allows the user to focus attention on the most pressing vulnerabilities within huge volumes of data, and streamlines the secure software development workflow through integration with development tools.

Categories and Subject Descriptors
H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Graphical user interfaces (GUI).*

General Terms
Security, Human Factors.

Keywords
Security visualization, software visualization, software assurance, data fusion, software analysis.

1. INTRODUCTION
DHS listed Software Assurance as the number one hard problem for Cyber Security: "poorly written software is at the root of all of our security problems."[10] There are a huge number of vulnerabilities out there, and bad or malicious coding practices are at the heart of the problem. However, tools exist today to help make software more secure. These software security analysis tools include open-source and commercial solutions. None of these tools on their own, however, are capable of finding all bugs or vulnerabilities. The NSA tested five different software security analysis tools on eight different applications and found that 84% of the vulnerabilities were identified by one tool and one tool alone. Kris Britton, technical director at NSA's Center for Assured Software, said: "No tool stands out as an uber-tool. Each has its strengths and weaknesses."[4] Different tools identify different weaknesses in software; using only one tool ensures that not all vulnerabilities will be found. Our own development and tests found that many of these tools produce enormous result sets and that many of the identified vulnerabilities are false positives. Coverity, a market leader in software analysis, targets a 20% false positive rate. [2] Assuming other tools have similar ratios of false positives, even the best case is that only 80% of the identified vulnerabilities are likely to be accurate. These tools also tend to present results in a view oriented around vulnerability hierarchies, but developers think in a different hierarchy – that of the source code itself. Current tools lack overviews of the results, making it difficult to understand the overall security of an application.

Better software analysis tools are only part of improving code security, because these tools:

- Identify different vulnerabilities;
- Use different semantics for results;
- Produce sizable datasets with numerous false positives;
- Present a vulnerability-centric view; and
- Offer no big picture overviews.

Our technical approach for this project was to develop a visual analysis environment, shown in Figure 1, that brings together the output of disparate commercial and open-source software analysis tools into a visual analysis environment that supports a natural workflow for developers to triage and explore the security state of their code. This solution integrates, correlates, and normalizes software analysis tools' data to increase vulnerability detection coverage, provides a visual overview and interaction methods to focus users' attention on the most pressing vulnerabilities within huge volumes of data, and streamlines secure development workflow through integration with software development tools.

The benefits of our visual analysis approach to Software Assurance include:

- More software analysis tools mean more vulnerabilities will be detected, and vulnerabilities that are detected by multiple tools have a higher confidence that they are accurate;
- Providing interactive information visualization presents results in an understandable format and grants the ability to focus on the most important vulnerabilities; and
- Integration with Systems Development Lifecycle tools provide a streamlined workflow that gives developers more time for coding and less time trying to interpret results.

VizSec '10, September 14, 2010, Ottawa, Ontario, Canada.

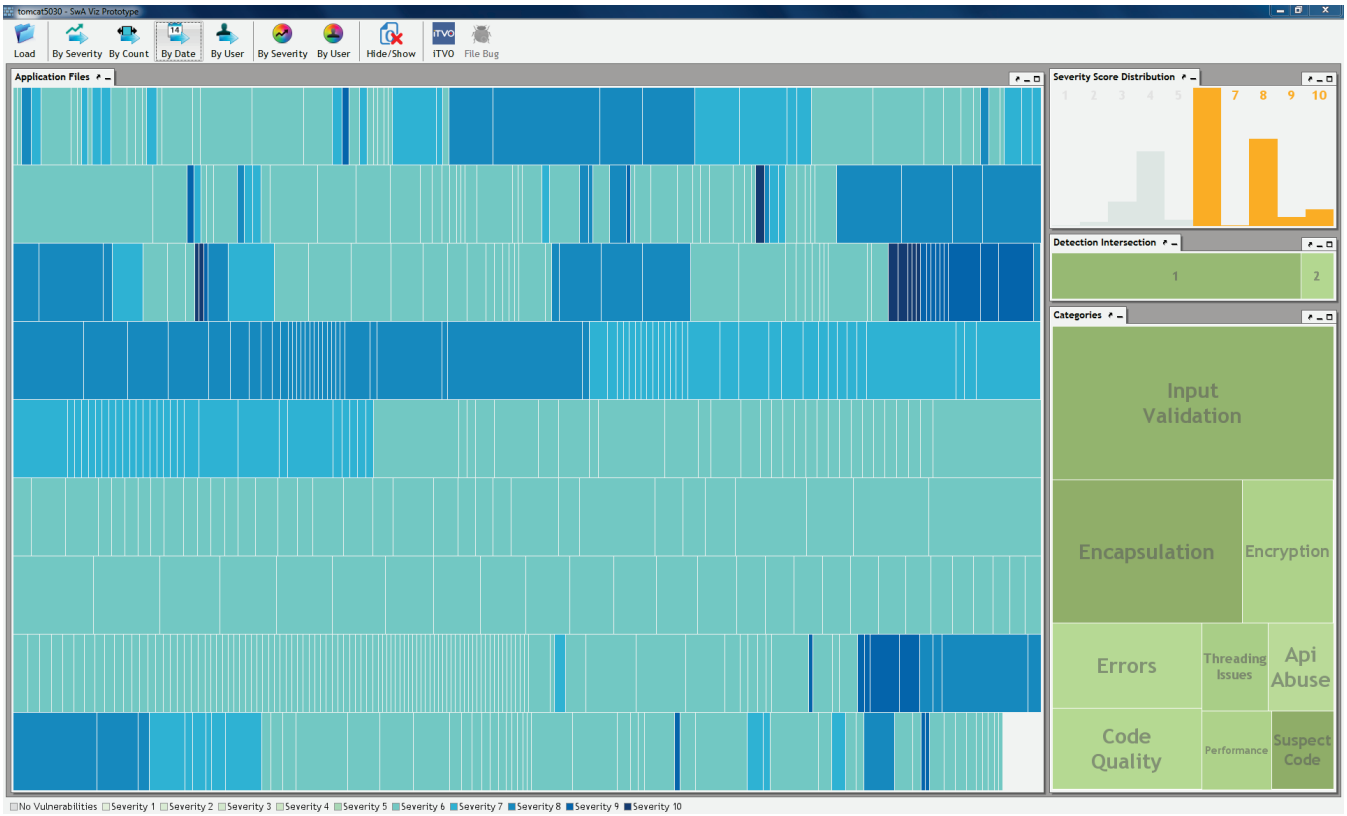

Figure 1. Visual analysis system for software security triage and analysis. Each block in the main display (left) represents a source code file, colored here by a weighted severity of the vulnerabilities within that file. Visual filter widgets (right) are used to interactively filter the data, and are linked together; here severity is filtered to only show high severity vulnerabilities.

2. LITERATURE REVIEW

There is a rich history in applying information visualization to the domains of software development (e.g. ACM Symposium on Software Visualization and IEEE International Workshop on Visualizing Software for Understanding and Analysis) and, more recently, cyber security (e.g. Symposium on Visualization for Cyber Security). However, little work has been applied to the visualization of software security. Recently, researchers have begun to look at applying visualization to the specific issues of reverse engineering and understanding malware behavior. Quist and Liebrock use visualization to aid in reverse engineering by identification of major functional areas and de-obfuscation through a node-link visualization in which nodes represent addresses and links represent state transitions between addresses.[11] Trinius, *et al.* use treemaps to display the distribution of operations and threadgraphs to display the sequence of operations.[13] These efforts are focused on facilitating the understanding of unknown code. Our work is focused on facilitating the understanding of how secure code is from a developer's standpoint.

In this sense, because our work is more in line with the software visualization community, we are providing a tool for developers to better understand their own code, rather than the security visualization community, which tends to look at the adversary's data, be it malware or network intrusions. However, this is an overlooked need with the security visualization community.

Software security analysis tools scan software for potential vulnerabilities and weaknesses within the code. These tools cover a broad range of categories in terms of how systems are checked and what is checked. NIST maintains a web site to support the Software Assurance community, Software Assurance Metrics And Tool Evaluation,[1] and has compiled a taxonomy of tools. Our focus was on the class of tools identified as Source Code Security Analyzers, which simplify the process of identifying potential security vulnerabilities by automating the process. Understanding the cause of software vulnerabilities is generally well understood (e.g. most developers can recognize a null pointer exception); however, vulnerabilities cannot be avoided without incorporating tools into the development process.[5]

Tools use a variety of techniques for detecting vulnerabilities within software. Many tools use a white box approach, statically analyzing the source against predefined patterns for rules. Some tools use a black box approach, exercising the program without internal knowledge. Although both these approaches reveal different sets of security flaws in software, the combination of both black and white box testing is even more powerful than either by itself.[8] To get the best picture possible of the overall security status of a system, the various techniques available for testing and exploiting a system should be utilized in conjunction with each other.

[1] http://samate.nist.gov/

While the number of potential vulnerabilities increases because tools greatly reduce the effort required in finding vulnerabilities in software, the potential vulnerabilities discovered need to be reviewed by an actual person due to both high false-positive and high false-negative rates among the different techniques.[6]

3. SYSTEM DESIGN

The goal of this project is to improve software developers' understanding of the security state of code by integrating the results from disparate software analysis tools into a unified system that facilitates visual analysis and integrates with existing software development tools. New capabilities for software developers include improved decision-making about the criticality and characteristics of found vulnerabilities and folding security into the software development process.

3.1 Data

We used two different software code bases for testing: a small, in-house graph visualization and Apache Tomcat.[2] The internally developed graph visualization is about 100 files and 15,000 lines of code. Apache Tomcat is about 1,500 files and 200,000 lines of code. We ran analysis tools on multiple versions of each of these code bases. We collected data using three different software security analysis tools. Two are commercial products; the other is an open-source tool. XML output produced from all three tools was correlated together.

Vulnerabilities detected by different tools identifying the same source code section – overlap of line and column numbers – are deemed to be equivalent. The data for all runs against each test code base yielded similar results to the NSA's[4]; there was very little overlap between the results of the three test tools. One of the commercial tools identified 40,000 potential vulnerabilities in a version of Tomcat, while the other found 2,500; the open source tool found 1,000. Only about 2,600 vulnerabilities were identified by two tools, and no vulnerabilities were found by all three tools.

3.2 Use Case

The primary use case driving the design of our system is triage, in which a developer or quality assurance analyst must prioritize the vulnerability results from multiple detection tools. Two challenges in this use case are the large number of vulnerabilities identified (e.g. the 40,000 vulnerabilities detected for one of the snapshots of Tomcat) and the high percentage of false positives (which were found when analyzing the data, but attempting to quantify is outside the scope of the project). The questions we attempted to address when designing for the triage use case were:

- Which vulnerabilities are noise / most important?
- What vulnerability categories are most common?
- What vulnerabilities are found by multiple tools?
- Where in the code are the vulnerabilities?
- Who do confirmed vulnerabilities get assigned to?

3.3 Visualization and Interaction

Our primary visualization is based on a block metaphor, similar to bargrams[14]. Each source code file is represented as a block. Each block aggregates the vulnerabilities found by the analysis tools for the file it represents. To move away from the traditional vulnerability-centric views we aggregated the vulnerability information in blocks the developers would recognize: the source files they created/maintained. The width of a file block corresponds to the number of potential vulnerabilities or bugs found within that file. The structure of the visualization is described in Figure 2. This method produces a very compact, space-filling visualization that allows the system to scale to very large code bases while still providing a useful data overview. Our implementation provides flexible data to visual mappings; for example, color can represent the developer that last modified the file or an average severity score. The color palettes used in the examples in this paper were derived from ColorBrewer [3]. The sort order can be customized by creation date, number of vulnerabilities, average severity score or the username of the developer that last modified the file.

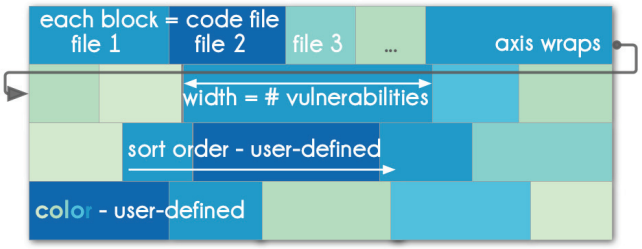

Figure 2. Overview visualization mockup: vulnerabilities are aggregated by source code file, represented as blocks and sized according to the number of vulnerabilities within each.

Our system takes advantage of multiple, coordinated views, where there are several supporting views, which we refer to as visual filter widgets, for the main visualization. Each supporting view is linked with each other and the main overview display so that interacting in one filters data in or out of the others.

The workflow for our prototype follows the Visual Information Seeking Mantra: "Overview first, zoom & filter, details on demand."[12] This process is outlined below:

- Overview: see and compare all of the vulnerabilities within a project across source code files;
- Filter: visual filter widgets for vulnerability category, weighted severity, analysis tool, and detection intersection to interactively and iteratively filter out data irrelevant to the user's current task;
- Zoom: zoom into a source code file to see the hierarchy of classes, methods and vulnerabilities within each file; and
- Details: view meta-data about individual vulnerabilities.

This workflow is supported by visual filter widgets and zooming. The visual filter widgets can be seen at the right of Figure 1. Selection on each of these widgets will filter the main display as well as the other visual widgets. This gives the user immediate response to see how different attributes interact together.

The first of these filter widgets is a severity distribution histogram for the vulnerability data. Each severity level in the distribution histogram uses the same severity-to-color mapping used in the main file display view. The y axis represents vulnerability count within the dataset, using a square root scale. Although a square root scale can be more difficult for users to grasp than a linear scale, or even a logarithmic scale, we wanted to ensure that smaller counts would not be obscured, while not skewing the differences as much as a logarithmic scale would have. Perceptually, identifying relative differences within a small

[2] http://tomcat.apache.org/

display area was more important than accuracy for our target use case.

The second filter widget summarizes the intersection detection among analysis tools of the vulnerabilities. The intersection detection is the number of software analysis tools that identified a vulnerability. It shows and allows the user to filter on the overlap among different analysis tools. Each block represents the number of tools that identified a vulnerability, sized by the number of vulnerabilities detected. This filter may be useful to initially investigate vulnerabilities that are found by multiple tools, on the theory that false positives will be less likely if multiple tools identify the same weakness in the code.

The final filter widget is a single level, squarified treemap visualization [1] of the categories of detected vulnerabilities. Each block within the treemap is sized by the vulnerability count of that category. Because categories are modeled as a single-level tree, a treemap may not be the best visualization, but we wanted the flexibility to add more complex categorical structures in the future such as the Common Weakness Enumeration (CWE).[9] This widget can help the user quickly narrow in on common types of issues.

3.4 Development Tool Integration

One of the goals is to fit our prototype visualization within the workflow of the software developer or quality assurance analyst. In order to do so, it is necessary to integrate the tool with existing software development tools. Integration with the Subversion Source Code Management system maps user information to source code files. Additionally, integration with the Bugzilla Issue Tracking system allows a user to click a button while an individual vulnerability is selected to create a new Bugzilla entry without leaving the prototype. These integrations are only a first step, intended to be a proof of concept to demonstrate the utility of integrating with different kinds of development tools.

3.5 Implementation

Our system is implemented in Java. The prefuse visualization toolkit is used as the basis for visualizations.[7] A custom parser facilitates the standardization and normalization of the different output formats produced by the software security analysis tools that our system leverages. A directory analyzer also pairs the files listed by the vulnerability analysis tools to actual source files and directories on the system, in case the security analysis took place on a different system. Software development tool integration is achieved by gathering user information from the Subversion[3] blame of specific revisions of source code. XML RPC was leveraged within Java to connect to a Bugzilla[4] server, file new bugs with relevant information, and assign them to the user indicated by the Subversion blame command.

4. SCENARIO WALKTHROUGH

To better describe our system and to demonstrate how it can facilitate the understanding of large data sets, this section will walk through how a developer or quality assurance analyst can use our system to triage and prioritize a massive vulnerability dataset. This scenario utilizes data from three software analysis tools that were run against an old version of Apache Tomcat.

[3] http://subversion.tigris.org/

[4] http://www.bugzilla.org/

After loading a data set, the initial display, shown in Figure 3, presents the user with an overview of all of the source code files within the project. The overview display on the left shows each individual file as a rectangular block. Each file is colored by the average weighted severity of all the vulnerabilities detected within it; the darker the color, the higher the average severity of the file. Thin, gray blocks within the overview display shows files that had no vulnerabilities.

Figure 3. Visualization of Apache Tomcat, which consists of 1,160 source code files. The data here shows nearly 34,000 vulnerabilities identified by 3 software analysis tools.

There are several sorting options for this overview display: by severity, by vulnerability count, by creation date, or by user. Sorting by severity and vulnerability count are probably the most useful for triage. We added the ability to sort by creation date in order to have a stable layout if we were to add an updating capability, in which newly created files from a previous analysis snapshot would be shown at the upper right of the display, and the oldest files on the bottom. Two coloring options are also available: by severity and by user. Sorting and coloring by user, i.e. the developer that last modified the source code file, enables a different view into the data and may facilitate different use cases, such as identifying developers who regularly check in code updates with vulnerabilities.

The series of smaller views vertically aligned to the right of the display area are the visual filter widgets the user can act on to filter the vulnerability data. For this dataset, the severity widget shows that most of the vulnerabilities are low (3 or 4), medium (6) or high (8); there are very few very high (9 or 10) severity vulnerabilities in relation to all of the identified vulnerabilities. Also, the vast majority of the vulnerabilities were detected by only one detection tool, a small percentage were detected by two tools, and none were detected by three tools. This finding is similar to the NSA study that also found there to be very littler overlap among tools.[4] The category with the largest number of vulnerabilities is Input Validation.

The overview display can reduce the dataset size an order of magnitude by aggregating vulnerabilities into files, as shown in Figure 3, which depicts 1,160 blocks representing each of the source code files, which include 33,907 vulnerabilities. Even with this large data reduction, the user needs the ability to further narrow the scope of their analysis. In this triage scenario, the user

is interested in finding the high-impact vulnerabilities first. To do that, the user selects the vulnerabilities with severity levels 6 through 10 from the severity distribution widget. Because different tools report vulnerabilities differently, and our normalization does not account for these semantic differences, a larger range of severities needs to be included to ensure vulnerabilities that certain tools may underestimate are not missed. Figure 1 shows the resulting display.

The selected severities in the distribution histogram are colored in orange while the ignored severities are colored in gray. Each of the visual filter widgets is linked; modifying one adjusts the others to show how the applied filter effects the distribution of those data attributes. Thus, the colors in both the detection intersection and category treemap filter widgets have changed. When a filter is active, the filter widgets show the percentage of vulnerabilities that match the active filters using color. White indicates that there are no matches and a green-hued sequential color palette is used to show the match percentage with darker shades indicating a higher percentage. As shown in Figure 1, with severity filters set to 6-10, the vulnerabilities mostly fall in the Input Validation, Encapsulation, and Suspect Code categories.

Although applying the severity filters begins to reduce the visible data, more filtering is necessary to drill down to a more manageable set of source code files. In order to further reduce the data, the user utilizes the detection intersection widget to only show the vulnerabilities that are detected by two tools, giving more confidence that the detected vulnerabilities are not false positives. In addition, sort order can be changed from file creation date to severity to aid in prioritizing. Figure 4 shows the result after applying the new filters and sort order. The overview display now shows the aggregated source code file information for only 227 vulnerabilities of the original nearly 34,000. This example shows how the user can quickly combine filters to remove less important data to begin to know what vulnerabilities within the code need to be dealt with first.

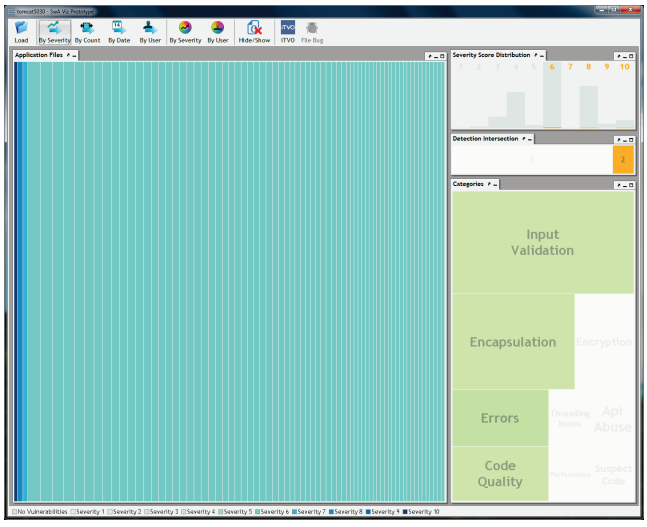

Figure 4. Filtered overview display that reduces the visible source code files to just a handful, with files having higher severity vulnerabilities at the left.

Choosing the highest average severity as a mechanism to prioritize on, the user can zoom in to the detailed view for it with a double click. The detail view, shown in Figure 5, contains a tree-table of the classes and methods contained in the file, each with entries for vulnerability count, average severity, and the severity distribution histogram for the class or method. The user-selected file, StandardWrapperValve.java, contains only two vulnerabilities that match the filter settings. As expected by first examining those vulnerabilities found by multiple tools, the vulnerabilities show a Null Pointer Exception on the same line found by two tools. Reviewing the method in the source code shows that this Null Pointer Exception is likely a true positive and could indeed be a vulnerability.

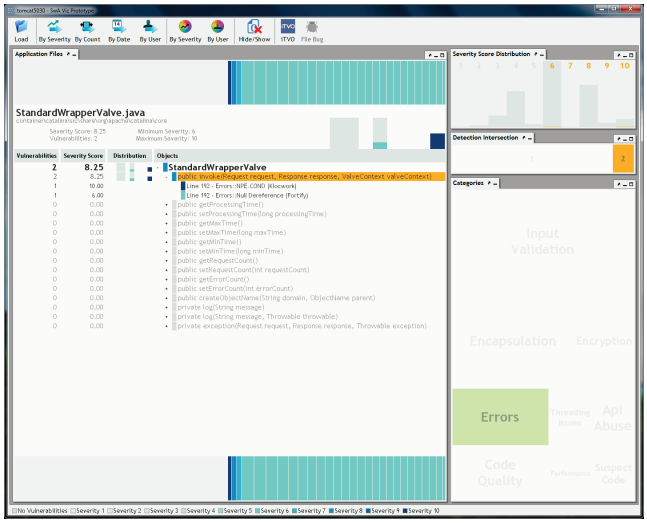

Figure 5. Detailed view, which shows the selected file's classes, the methods within those classes and the vulnerabilities within each method as an expandable tree-table that uses visual cues to help the user quickly find high severity methods.

With only a few clicks, our system provided the ability to drill down from a listing of nearly 34,000 vulnerabilities to look at the details of two overlapping vulnerabilities. From this point, the user could choose to look at the file listing by user – shown in Figure 6 – to see if there are any patterns that jump out.

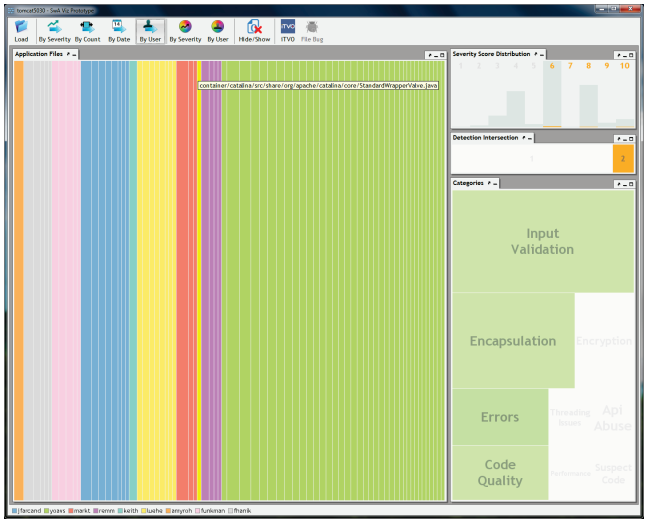

Figure 6. Overview display that colors and sorts source code files by the user that last checked in the file, showing that about half of the visible files were checked in by the same developer (in green, at right).

In this case the user represented by the light green takes up nearly the entire right half of the display; this could mean the programmer regularly checks in code with vulnerabilities, or just that she is the main developer that modifies more code than anyone else. An understanding of the context would be required to fully understand these types of patterns.

The user may also wish to assign the vulnerability to a developer by using the embedded ability to click on a vulnerability and automatically enter an issue into the issue tracking system based on the developer that last modified the code and the attributes of the vulnerability itself.

5. CONCLUSIONS

This research effort has resulted in a prototype system that correlates and normalizes the output of multiple software analysis tools that automatically detect potential weaknesses and vulnerabilities in software code. The system visualizes this output and provides an overview-to-details workflow to triage vulnerabilities utilizing a developer-centric view. This workflow allows the system to scale to large code bases with tens of thousands of vulnerabilities. Several integration points were made with software development tools to associate users with vulnerabilities and to submit bug reports to an issue tracking system.

Our future plans include extending the detail view to show source code with cross-reference vulnerability information as opposed to source code metadata only, as shown in Figure 7.

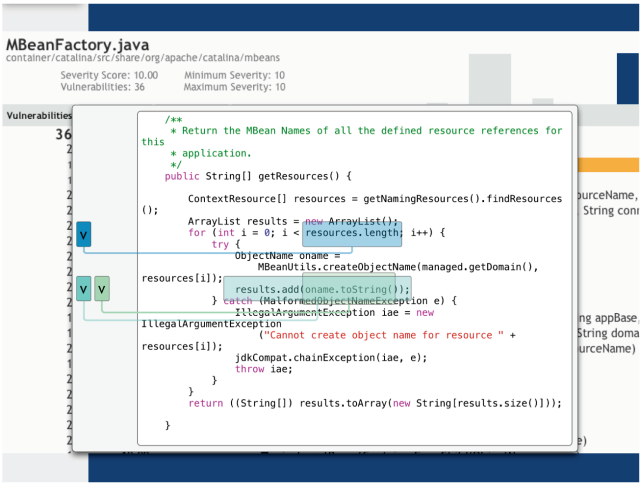

Figure 7. Mockup integrating source code decorated with vulnerability information within the detailed view.

Additionally, the current system shows a point in time snapshot of the vulnerability state of a software code base. In the future we plan on extending the system to look at vulnerability trends across time; for example, we will explore the use of animation to show new files being added to the code base and changes in the number of vulnerabilities found in existing code files. We also plan on integrating additional software analysis tools that can be used as input data to our visual analysis system and increasing the variety of Source Code Management and Issue Tracking systems. We plan on investigating other development tools for potential integration with our system that developers would benefit from.

We would also like to explore additional use cases, such as visual analysis of vulnerability trends over time, and design visual analysis support for them. Finally, we would like to validate our approach through user evaluations of the prototype system.

6. ACKNOWLEDGMENTS

This research and development effort is supported by DHS S&T through a Small Business Innovative Research grant, under contract no. N10PC20014 .

7. REFERENCES

1. Bederson, B.B., Shneiderman, B. and Wattenberg, M. 2002. Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies. *ACM Transactions on Graphics*, *21* (4). 833-854.
2. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S. and Engler, D. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, *53* (2). 66-75.
3. Brewer, C.A. http://www.ColorBrewer2.org/. Accessed 6/10/2010.
4. Buxbaum, P. http://gcn.com/articles/2007/03/18/all-for-one-but-not-one-for-all.aspx. Accessed 6/10/2010.
5. Evans, D. and Larochelle, D. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, *19* (1). 42-51.
6. Goertzel, K.M., Winograd, T., McKinley, H.L., Oh, L., Colon, M., McGibbon, T., Fedchak, E. and Vienneau, R. 2007. Software Security Assurance: A State of the Art Report.
7. Heer, J., Card, S.K. and Landay, J.A. 2005. prefuse: a toolkit for interactive information visualization *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, Portland, Oregon, USA, 421-430.
8. Janardhanudu, G. 2005. White Box Testing *Build Security In*.
9. Martin, R.A., Christey, S.M. and Jarzombek, J. 2005. The Case for Common Flaw Enumeration. In *NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics*.
10. Maughan, D. 2010. The need for a national cybersecurity research and development agenda. *Communications of the ACM*, *53* (2). 29-31.
11. Quist, D.A. and Liebrock, L.M. 2009. Visualizing compiled executables for malware analysis. In *International Workshop on Visualization for Cyber Security (VizSec)*, 27-32.
12. Shneiderman, B. 1996. The eyes have it: A task by data type taxonomy of information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, 336-343.
13. Trinius, P., Holz, T., Gobel, J. and Freiling, F.C. 2009. Visual analysis of malware behavior using treemaps and thread graphs. In *International Workshop on Visualization for Cyber Security (VizSec)*, 33-38.
14. Wittenburg, K., Lanning, T., Heinrichs, M. and Stanton, M. 2001. Parallel bargrams for consumer-based information exploration and choice *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, ACM, Orlando, Florida, 51-60.