SSVChecker: Unifying Static Security Vulnerability Detection Tools in an Eclipse Plug-In

Josh Dehlinger
Dept. of Computer Science
Iowa State University
dehlinge@iastate.edu

Qian Feng
ABC Virtual Communications
qfeng@abcv.com

Lan Hu
Dept. of Computer Science
Utah State University
lanhu@cc.usu.edu

ABSTRACT
The increasing complexity of secure software applications has given rise to static analysis security tools to alert developers to potential security flaws within source code. However, these static security vulnerability detection tools tend to be difficult to use and are not integrated with common software development environments. The contribution of this work is *SSVChecker*, an Eclipse plug-in that unifies existing static security vulnerability detection tools into a powerful, intuitive tool. We make three fundamental claims for *SSVChecker*. First, it contains functionality not found in other static security vulnerability detection tools (e.g., union and intersection of multiple tool results). Second, the tool can adapt to the results of user-performed analysis to prevent repeatedly reporting user-dismissed security vulnerabilities. Lastly, it operates on a user-friendly, generic framework allowing for the inclusion of future static security vulnerability detection tools. To illustrate these claims, we use *SSVChecker* on a security-sensitive networking package. Results show the benefits of the tool in identifying potential security vulnerabilities.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – V*alidation*.

General Terms
Security

Keywords
Software security, secure programming, security auditing

1. INTRODUCTION
The necessity for software developers to consistently produce secure code for security-critical software applications continues to increase as software becomes progressively more immersed in our lives (e.g., *e*-commerce, online banking, etc.). The recent surge of interest in developing and improving security vulnerability tools is one response by which researchers are trying to cope with the demand for secure applications. Despite existing security vulnerability detection tools (e.g. RATS [10], ITS4 [14], Splint [12], MOPS [2], etc.), extensive software security literature (e.g., [3, 11, 13]) and documented security attacks used to exploit software systems (e.g., [1, 7, 8, 9]), a large amount of software produced continues to have security vulnerabilities that have been repeatedly exploited for nearly 20 years (e.g., the format-string vulnerability in C) [8, 9, 16].

Potential security vulnerabilities are often introduced into software by commonly used library functions and language-specific constructs unknowingly by software developers [4]. There are two likely contributing factors as to why software developers have failed to adequately mitigate known security vulnerabilities in today's software. First, software developers may not be aware that they are introducing potentially devastating security vulnerabilities into software [4, 14]. This likely stems from a lack of education and awareness of common software attacks and proper secure programming. Second, software developers do not have adequate, easy-to-use tools in a familiar environment containing desired features to flag potential security risks in their developed code and provide explanations and possible solutions during development [4, 14].

If software developers were warned of potential security vulnerabilities, provided with an explanation and given possible remedies within a software IDE, we believe that even novice software developers could produce applications devoid of commonly exploited, known security vulnerabilities. Until then, it is futile to expect secure software in everyday applications.

This work addresses these two problems by providing a tool that alerts software developers to potential security vulnerabilities in their source code. Moreover, we hope to bridge the gap between existing static analysis security vulnerability detection tools and software developers by unifying existing security vulnerability detection tools into a single interface, *SSVChecker* (**S**tatic **S**ecurity **V**ulnerability **Checker**). *SSVChecker* is an Eclipse plug-in to fuse existing static security vulnerability detection tools into a powerful, developer-friendly tool. Specifically, *SSVChecker* offers three fundamental advantages to static analysis security vulnerability detection:

- Provides features not found in other security vulnerability detections tools (e.g., union and intersection of multiple tool results) that better aids developers in identifying potential security vulnerabilities.
- Adapts to the results of user-performed analysis, without altering the original source code, to prevent repeatedly

Eclipse Technology Exchange Workshop at OOPSLA'06, October 22–23, 2006, Portland, OR, USA.

```
<?xml version="1.0" ?>
<Vulnerability-List>
        <Vulnerability>
                <Filename>C:\its4\inet.c</Filename>
                <Line-Number>92</Line-Number>
                <Priority>Very Risky</Priority>
                <Function>strcpy</Function>
                <Source-Code>strcpy(name, np->n_name);</Source-Code>
                <Description>This function is high risk for buffer overflows.</Description>
                <Suggestion>Use strncpy instead.</Suggestion>
        </Vulnerability>
</Vulnerability-List>
```

Figure 1. *SSVChecker's* XML format for potential security vulnerabilities.

reporting user-dismissed security vulnerabilities allowing developers to concentrate on those flagged security vulnerabilities that still warrant attention.

- Operates on a user-friendly, generic framework allowing the inclusion of future static security vulnerability detection tools.

The remainder of this paper is organized as follows. Section 2 reviews related work in existing static security vulnerability detection tools. Section 3 presents an overview of *SSVChecker* and describes its interface. Section 4 presents an evaluation and discussion of *SSVChecker*. Finally, Section 5 provides concluding remarks and planned future work.

2. RELATED WORK

Static analysis security tools attempt to find security vulnerabilities without executing the software by scanning the source code for known potentially security-compromising functions. They then perform analyses to try to determine if, indeed, a function call could be maliciously attacked. These tools can not guarantee to find all security vulnerabilities in a program and often report many false positives (those potential vulnerabilities reported by a tool which are not actual vulnerabilities).

This work illustrates *SSVChecker* by integrating three existing static security vulnerability detection tools into an Eclipse plug-in: ITS4 [14], RATS [10] and Splint [12]. Although these tools were used here to illustrate the features of *SSVChecker*, Section 3 briefly discusses how *SSVChecker* can be used with any static security vulnerability detection tool because of the use of a generalized XML format for security vulnerabilities.

ITS4 was one of the first available static security analysis tools to search C source code looking for potentially dangerous function calls [14]. ITS4 performs limited analysis to determine how risky a function call is and, for every problem reported provides suggestions how to mitigate the security vulnerability.

RATS is similar to ITS4 in its approach but performs additional analysis to attempt to reduce the number of false positives reported [10]. Unlike ITS4, however, RATS performs analysis to discover Time Of Check, Time Of Use race conditions.

Splint (Secure Programming Lint) is an improvement over another static security analysis tool, Lint [12] that does additional analysis on potential security vulnerabilities beyond both ITS4 and RATS.

Other tools perform different analysis techniques to try and discover a different type of security vulnerability or eliminate a different type of false positives. For example, BOON [15] performs analysis focusing primarily on the detection of the buffer overflow security vulnerability whereas FlawFinder [5] uses a vulnerability database as does ITS4 and RATS. Thus, different tools often produce different sets of results. *SSVChecker* allows users to exploit the differences in analysis by providing the potential security vulnerabilities of multiple tools' results.

3. *SSVCHECKER* DESCRIPTION

This section provides an overview and description of the interface and features of *SSVChecker*. A full demonstration of *SSVChecker* is available online at http://www.cs.iastate.edu/~dehlinge/ssvchecker/SSVCheckerDemo.htm.

SSVChecker relies on reading the results from external static analysis security vulnerability detection tools in an XML format, shown in Figure 1. Although existing tools do not provide output in this XML format, it is trivial to provide a wrapper to convert a tool's existing output to our XML schema.

The XML schema was designed after studying the results produced by a number of existing tools (including ITS4 [14], RATS [10], Splint [12], FlawFinder [5], MOPS [2] and BOON [15]). This was done to generalize the results reported by the various static analysis security tools currently available and to give the software developer adequate information pertaining to a potential security vulnerability. The XML schema is also intended to generalize a file format in which future static analysis tools could produce output allowing *SSVChecker* to unify all static analysis security tools in a user-friendly software IDE.

SSVChecker provides software developers with an interface within Eclipse to run existing static analysis security vulnerability detection tools (e.g., ITS4 [14], RATS [10] and Splint [12]) to find potential security vulnerabilities in source code during development. Within *SSVChecker*, software developers have the option of executing a single or multiple static analysis security vulnerability detection tools on the source code in development.

Software developers electing to run a single static analysis security vulnerability detection tool within *SSVChecker* have access to all the features of the desired tool (i.e., parameters can still be provided to the tool to perform specialized/concentrated analysis). However, the software developer benefits by getting the results displayed in Eclipse for simultaneous viewing of the source code and flagged potential security vulnerabilities.

Moreover, *SSVChecker* allows software developers to execute multiple static analysis security vulnerability detection tools and calculate and return either the union or intersection of the results. Users select the tools they desire to run on their source code and select intersection or union. *SSVChecker* will automatically execute the appropriate tools, calculate the intersection/union and present the results to the user within Eclipse.

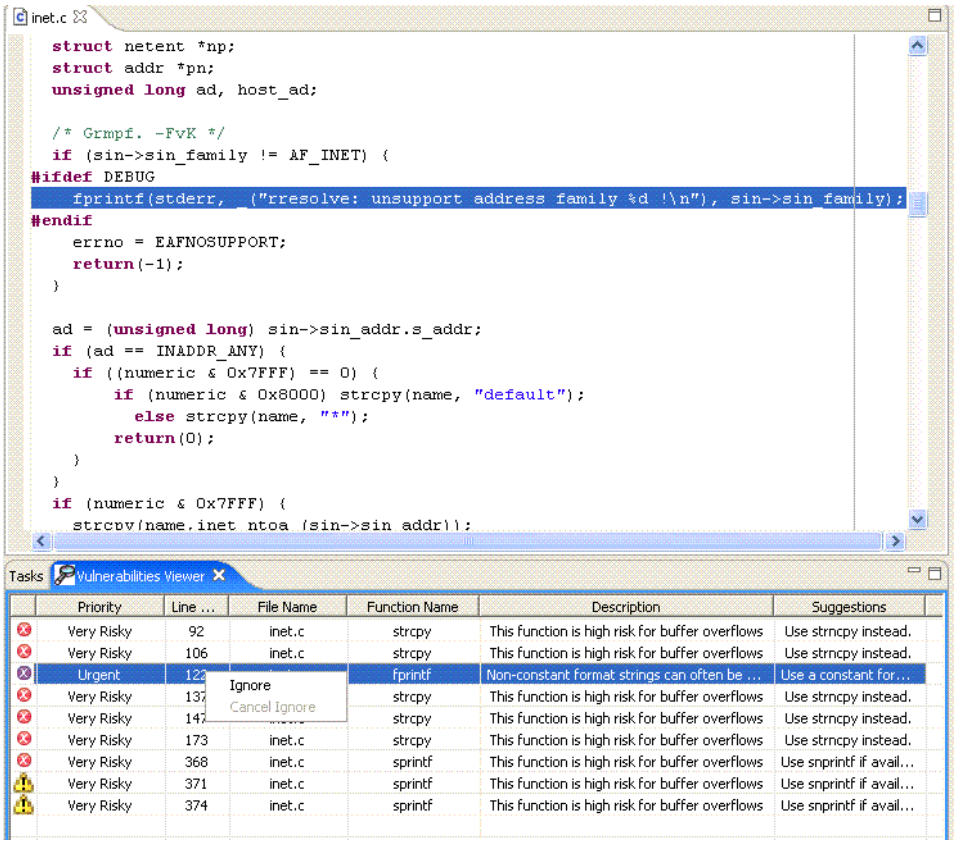

Figure 2. *SSVChecker's* interface in Eclipse.

Figure 2 presents a screenshot within the Eclipse IDE illustrating the results of running a single static analysis security vulnerability detection tool from *SSVChecker*. The results are presented in the Vulnerability Viewer allowing for each column to be sorted by the user. Double-clicking any potential vulnerability automatically focuses on the associated line of code, as shown in Figure 2.

The Vulnerability Viewer, shown in Figure 2, provides a summary of the results by listing the number of vulnerabilities and ignored vulnerabilities, discussed next, above the list of potential security vulnerabilities. These features allow software developers to easily use static analysis security vulnerability detection tools to analyze their source code, quickly identify high priority potential security vulnerabilities and efficiently mitigate them using the provided suggestions. Thus, we envision *SSVChecker* as a tool that software developers can utilize on a semi-frequent basis much like a compiler, during security-critical software development.

SSVChecker provides functionality to be able to adapt to the results of user-performed analysis to prevent repeatedly reporting user-dismissed security vulnerabilities. This allows developers to concentrate on those flagged security vulnerabilities that still warrant attention (analogous to Microsoft Word's "Ignore Once" option for misspelled words). To achieve this, *SSVChecker* provides users with the option of ignoring a potential vulnerability if the user has manually determined that the reported vulnerability does not pose a security risk. This prevents repeatedly drawing the developers' attention to the same vulnerability on subsequent executions of *SSVChecker*. As shown in Figure 2, previously ignored vulnerabilities are filtered to the bottom of the list and are tagged as a warning (with a different icon). *SSVChecker* also provides users with the ability to stop ignoring (called "Cancel Ignore") a specific vulnerability as is shown in Figure 2.

4. EVALUATION AND DISCUSSION

This section provides an evaluation of *SSVChecker* by examining the results of running three different static security vulnerability detection tools separately using *SSVChecker* and then using *SSVChecker* to calculate the intersection and union of the tools to justify the advantage of such features to a software developer.

We evaluated *SSVChecker* using portions of the net-tools 1.46 networking package. Net-tools 1.46 is an open source package written in C for the Linux operating system consisting of several commands related to networking [6]. Specifically, we used *SSVChecker* on the netstat.c (approximately 1,400 LOC) and inet.c (approximately 400 LOC) files of the net-tools 1.46 package.

4.1 SSVChecker's Intersection and Union Results

We provide the results and analysis from preliminary tests showing the results of using *SSVChecker* to calculate and return the intersection and union sets of potential security vulnerabilities on portions of the net-tools 1.46 package. The results, shown in Table 1 and discussed below, illustrate the value of *SSVChecker's* intersection/union calculation feature.

Table 1. Number of potential security vulnerabilities reported by various static security vulnerability detection tools as well as intersection/union results using *SSVChecker*.

	ITS4	RATS	Splint	ITS4 ∪ RATS	ITS4 ∩ RATS	ITS4 ∪ RATS ∪ Splint	ITS4 ∩ RATS ∩ Splint
inet.c	12	16	51	16	12	58	9
netstat.c	111	94	485	139	66	581	43

When using *SSVChecker* to run ITS4, RATS and Splint separately on the inet.c source file 12, 16 and 51 potential security vulnerabilities, respectively, were reported. For the netstat.c code 111, 94 and 485 potential security vulnerabilities were reported from ITS4, RATS and Splint, respectively, by *SSVChecker*.

Retrieving the intersection of ITS4 and RATS for the inet.c code yielded 12 potential vulnerabilities. Unfortunately, this set represented the same 12 vulnerabilities originally reported by ITS4. Thus, for this particular file, the results reported by ITS4 were a subset of the results reported by RATS and the intersection feature of *SSVChecker* did not provide any additional insight or advantage for a software developer.

Retrieving the intersection for ITS4 and RATS for the netstat.c code yielded in 66 potential vulnerabilities reported, representing a reduction of 45 and 28 possible security vulnerabilities from executing ITS4 and RATS separately, respectively. In this case, there were vulnerabilities reported by ITS4 that were not reported by RATS and vice versa. This reduction is advantageous in that it assists a software developer in identifying likely security vulnerabilities since more than one static security vulnerability detection tool, each using different analysis methods, flagged them as being a potential security vulnerability.

Calculating the intersection of the ITS4, RATS and Splint results for the inet.c code yielded only 9 potential vulnerabilities. Likewise, using *SSVChecker* to calculate the intersection of the ITS4, RATS and Splint results for the netstat.c code yielded 43 potential security vulnerabilities and is discussed further in Section 4.2.

Using *SSVChecker* to calculate the union of the ITS4 and RATS results for the inet.c and netstat.c source files yielded 16 and 139 potential security vulnerabilities, respectively. Similarly, the union set of potential vulnerabilities for ITS4, RATS and Splint for the inet.c and netstat.c source files are 58 and 581 potential security vulnerabilities, respectively.

4.2 Discussion

From the experimental results, it appears that *SSVChecker* can benefit software developers when developing security-critical software. The convenience of a generalized tool allowing a developer to execute any existing static security vulnerability detection tools gives software developers the desired flexibility within the Eclipse IDE. Further, *SSVChecker* integrates the results of static security vulnerability detection tools to allow software developers to simultaneously view the source code and potential security vulnerabilities, allowing for a shorter vulnerability mitigation cycle.

Through the evaluation results, we believe that *SSVChecker's* feature to calculate the intersection set of potential security vulnerabilities from the results of several executed tools can help software developers concentrate on an initial set of likely security vulnerabilities. Assuming that a security vulnerability is more likely an actual vulnerability if multiple tools flag it, *SSVChecker's* ability to calculate the intersection set provides software developers with a feature that is not found elsewhere. Thus, the calculation of intersection sets in *SSVChecker* provides developers with a useful asset during the development of security-critical code or during a security audit/code review by circumventing the poor precision (i.e., the high rate of false positives) of a single static security vulnerability detection tool.

The ability to calculate the union of potential security vulnerabilities in *SSVChecker* from the results of several executed tools may be valuable when security is a high priority and a listing of all possible vulnerabilities is more desirable than a list of likely security vulnerabilities. While the resulting union sets always increased the number of potential security vulnerabilities compared to the results of any single tool, it did not give any indication which potential security vulnerabilities were more or less likely to be actual vulnerabilities. Rather, the union set gave a more comprehensive list of potential security risks in the source code. This may be a valuable asset during code reviews or security audits when security is a high priority.

Although not measured in the evaluation results, *SSVChecker's* ability to "ignore" previously flagged potential security vulnerabilities allows the tool to adapt to the manual analysis performed by the user so that they are not repeatedly bothered by previously dismissed flagged vulnerabilities. In our experience from using ITS4 and RATS when securing C code, repeatedly being warned about a specific potential security vulnerability even after, through a manual inspection of the code, we know it can not be exploited by a malicious user is annoying and distracting from those other potential vulnerabilities that still warrant our attention. Thus, we believe that the ability to temporarily ignore a vulnerability (and, thus, having *SSVChecker* filter it to the bottom of the list) is a valuable feature that focuses developers attention on those potential security vulnerabilities that have not yet been considered by other static security vulnerability detection tools.

Finally, this work illustrated *SSVChecker's* application to C/C++ source code with static security vulnerability detection tools targeted for performing analysis to detect vulnerabilities in C/C++ code. However, there is nothing preventing *SSVChecker* from being applied in the same manner to other types of source code using static security vulnerability detection tools for other languages. For example, the Eclipse IDE can be used to develop an application in Python and *SSVChecker* can be used to execute RATS [10], which can also check for security vulnerabilities in Python, to identify and report potential security vulnerabilities.

5. CONCLUSION AND FUTURE WORK

This paper described *SSVChecker*, an Eclipse plug-in to unify existing static security vulnerability detection tools into a powerful, developer-friendly tool. *SSVChecker* provides software developers the ability to analyze their code using existing static security vulnerability detection tools within the Eclipse IDE and displays the results in a familiar, customizable manner. This work bridges the gap between research-oriented, command-line-based, static security analysis tools and development IDE's so that both novice and advanced software developers can take advantage of security analysis tools when developing security-critical code.

SSVChecker allows software developers to run either a single tool or multiple tools and provides a generic framework such that any existing or future static security vulnerability detection tool can be used within *SSVChecker*. When executing multiple tools, *SSVChecker* provides the user with the ability to return either an intersection or union of the results of multiple tools. This allows the software developer to view a narrowed set of likely security vulnerabilities (i.e., the intersection set of multiple tools) or a large set of possible security vulnerabilities (i.e., the union set of multiple tools). Finally, *SSVChecker* is able to adapt to user-performed analysis by temporarily ignoring previously reported security vulnerabilities so that the user's attention is focused on those security vulnerabilities not yet considered.

Planned future work is threefold. First, *SSVChecker* was initially developed as a proof-of-concept tool. Thus, *SSVChecker's* calculation of intersection/union potential vulnerability sets of multiple tools is currently more time-consuming than necessary. Depending on the size of the source code and the number of tools selected to analyze the code and the number of potential security vulnerabilities, *SSVChecker* may take up to 10 seconds to display results. We have identified a number of areas in which *SSVChecker* efficiency can be improved so that its calculation of intersection/union sets is faster. Thus, our first planned future work includes tuning *SSVChecker's* efficiency as well as performing an analysis of the length of time required by *SSVChecker* to return intersection/union sets as the number of tools and vulnerabilities increase. We would like to keep the time performance fast enough that software developers will use *SSVChecker* during the development of security-critical code.

Second, we plan on introducing *SSVChecker* into a classroom to study its use as a learning tool to teach secure programming practices to upper-level undergraduate students. We believe that its integration into the Eclipse IDE along with its ease-of-use can help novice developers with little security programming experience the ability to quickly adopt secure programming practices. By following the guidelines in the suggestions given by the static security vulnerability detection tools shown in *SSVChecker's* Vulnerability Viewer, their programs will be more secure. In this study, we would also like to identify how *SSVChecker* is used by novice software developers as well as its usefulness in producing secure code. In particular, the usefulness of the ability to calculate intersection/union sets of potential security vulnerabilities in identifying and prioritizing possible security vulnerabilities for the user.

Finally, we would eventually like to include a "Quick Fix" feature (analogous to Eclipse's "Quick Fix" feature for some compiler errors). Such a feature could help expedite and partially automate the process of transforming insecure code to secure code.

6. ACKNOWLEDGEMENTS

The authors would like to thank Dr. Suraj C. Kothari for his valuable suggestions throughout this work and Dr. Robyn R. Lutz for her helpful feedback on an earlier version of this work.

Josh Dehlinger and Qian Feng were supported by the National Science Foundation under grants 0204139 and 0205588.

7. REFERENCES

[1] CERT Coordination Center. http://www.cert.org/. (current August 2006).

[2] Chen, H. and Wagner, D. MOPS: Model Checking Programs for Security Properties. http://www.cs.berkeley.edu/~daw/mops/. (current August 2006).

[3] Engler, D., Chen, D. Y., Hallem, S., Chou, A. and Chelf, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proc. 18th ACM Symposium on Operating System Principles*, pp. 57-72, Banff, Alberta, Canada, 2001.

[4] Evans, D. and Larochelle, D. Improving Security Using Extensible Lightweight Static Analysis. In *IEEE Software*, pp. 42-51, January, 2002.

[5] FlawFinder Home Page. http://www.dwheeler.com/flawfinder/. (current August 2006).

[6] Linux.com – Net-tools. http://howtos.linux.com/guides/html/appendixa/net-tools.shtml. (current August 2006).

[7] Linux Security Resources – The Community's Center for Security. http://www.linuxsecurity.com/content/view/101892/155/. (current August 2006).

[8] Microsoft TechNet Security Center. http://www.microsoft.com/technet/security/default.mspx. (current August 2006).

[9] Open Source Vulnerability Database. http://www.osvdb.org/. (current August 2006).

[10] RATS: Rough Auditing Tool for Security. http://www.securesoftware.com/resources/tools.html. (current August 2006).

[11] Seacord, R. C. *Secure Coding in C and C++*. Addison-Wesley Professional, 2005.

[12] Splint: Secure Programming Lint. http://www.splint.org/. (current August 2006).

[13] Tsipenyuk, K., Chess, B. and McGraw, G. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. In *IEEE Security and Privacy*, pp. 81-84, 2005.

[14] Viega, J., Bloch, J. T., Kohno, T. and McGraw, G. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proc. 16th Computer Security Applications Conferences* pp. 257-266, New Orleans, LA, 2000.

[15] Wagner, D. BOON: Buffer Overrun Detection. http://www.cs.berkeley.edu/~daw/boon/. (current August 2006).

[16] Wagner, D., Foster, J. S., Brewer, E. A. and Aiken, A. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pp. 3-17, San Diego, CA, 2000.