Using the Juliet Test Suite to compare Static Security Scanners

Andreas Wagner[1], Johannes Sametinger[2]
[1]*GAM Project, IT Solutions, Schwertberg, Austria*
[2]*Dept. of Information Systems – Software Engineering, Johannes Kepler University Linz, Austria*
andreas.wagner@gam-project.at, johannes.sametinger@jku.at

Keywords: Juliet test suite, Security scanner, Scanner comparison, Static analysis

Abstract: Security issues arise permanently in different software products. Making software secure is a challenging endeavor. Static analysis of the source code can help eliminate various security bugs. The better a scanner is, the more bugs can be found and eliminated. The quality of security scanners can be determined by letting them scan code with known vulnerabilities. Thus, it is easy to see how much they have (not) found. We have used the Juliet Test Suite to test various scanners. This test suite contains test cases with a set of security bugs that should be found by security scanners. We have automated the process of scanning the test suite and of comparing the generated results. With one exception, we have only used freely available source code scanners. These scanners were not primarily targeted at security, yielding disappointing results at first sight. We will report on the findings, on the barriers for automatic scanning and comparing, as well as on the detailed results.

1 INTRODUCTION

Software is ubiquitous these days. We constantly get in touch with software in different situations. For example, we use software for online banking; we use smartphones, we drive cars, etc. Security of software is crucial in many, if not most situations. But news about security problems of software systems continues to appear in the media. So the question arises: how can security errors be avoided or at least minimized? Security is complex and difficult to achieve. It is commonly agreed on that security has to be designed into software from the very start. Developers can follow Microsoft's secure software life-cycle (Howard, Lippner 2006) or adhere to the security touch points (McGraw 2009). Source code reviews depict an important piece of the puzzle in the direction of secure software. Source code scanners provide a means to automatically review source code and to detect problems in the code. These scanners typically have built-in, but mostly extensible sets of errors to look for in the code. The better the scanner and its rule set, the better the results of its scan. We have used a test suite that contains source code with security weaknesses to make a point about the quality of such scanners. We have analyzed several scanners and compared their results with each other.

The paper is structured as follows: Section 2 gives an overview of the Juliet Test Suite. In Section 3, we introduce security scanners. The process model for the analysis and comparison is shown in Section 4. Section 5 contains the results of our study. Related work is discussed in Section 6.

2 JULIET TEST SUITE

The Juliet Test Suite was developed by the Center for Assured Software (CAS) of the US American National Security Agency (NSA) (Center for Assured Software 2011). Its test cases have been created in order to test scanners or other software. There are two parts of the test suite. One part covers security errors for the programming languages C and C++. The other one covers security errors for the language Java. Code examples with security vulnerabilities are given in simple form as well as embedded in variations of different control flow- and data-flow patterns. The suite contains around 57,000 test cases in C/C++ and around 24,000 test cases in Java

Table 1: Top 10 Security Errors (MITRE 2011)

No	Score	ID	Name
1	93.8	CWE-89	Improper Neutralization of Elements used in an SQL Command (SQL Injection)
2	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)
3	79.0	CWE-120	Buffer Copy without Checking Size of Input (Classic Buffer Overflow)
4	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)
5	76.9	CWE-306	Missing Authentication for Critical Function
6	76.8	CWE-862	Missing Authorization
7	75.0	CWE-798	Use of Hardcoded Credentials
8	75.0	CWE-311	Missing Encryption of Sensitive Data
9	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
10	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision

(Boland, Black 2012). A test suite can only cover a subset of possible errors. The Juliet Test Suite covers the top 25 security errors defined by SANS/MITRE (MITRE 2011). MITRE is a non-profit organization operating research and development centers funded by the US government. The SANS Institute is a cooperative research and education organization and is a trusted source for computer security training, certification and research (http://www.sans.org/). CWE is a community-developed dictionary for software weakness types (http://cwe.mitre.org/). These types have been used for the classification of the security errors in the Juliet test Suite. Each CWE entry describes a class of security errors. For example, CWE-89 describes "Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)". This happens to be the top 1 security error according to SANS/MITRE. Table 1 shows the first 10 of the top 25 security errors by SANS/MITRE (MITRE 2011).

2.1 Structure of the Test Suite

The Juliet Test Suite contains source code files that are structured in different folders. Each folder covers one CWE entry. Therefore, there are several source code files in every folder that contain a collection of errors for the specific CWE entry. Every source code file targets one error. In most cases, this error is located in a function called "Bad-Function". But there are also cases, where the error is contained in some helper functions called "Bad-Helper". Additionally, "Class-based" errors arise from class inheritance. Besides bad functions, there are also good functions and good helper functions. These functions contain nearly the same logic as the bad functions but without the security errors. The good functions can be used to prove the quality of security scanners. They should find the errors in the bad functions and its helpers but not in the good functions (National Institute of Standards and Technology 2012). In version 1.1, the Juliet Test Suite covers 181 different kinds of flaws, including authentication and access control, buffer handling, code quality, control-flow management, encryption and randomness, error handling, file handling, information leaks, initialization and shutdown, injection, and pointer and reference handling (Boland, Black 2012).

2.2 Natural code vs. artificial code

We can distinguish two types of source code, i.e., artificial code and natural code. Natural code is used in real software like, for example, the Apache Webserver or Microsoft Word. Artificial code has been generated for some specific purpose, for example, to test security scanners. The Juliet Test Suite contains only artificial code, because such code simplifies the evaluation and the comparison of security scanners. In order to determine whether an error reported by a security scanner is correct, it is necessary to know where exactly there are any security bugs in the source code. This is a difficult task for natural code, because the complete source code would have to be subject of close scrutiny. For artificial code this is a much easier task, because the code had been generated and documented with specific errors in mind anyway.

Any security scanner may not find specific security errors in natural code. A manual code review is necessary to find such problems, provided the availability of personnel with sufficient security knowledge. Otherwise, the existence of these security errors in the code may remain unknown. In contrast, the number of errors in artificial code is known. Only if the errors in the test suite are known, we can check whether scanners find all of these. But even for artificial code, it is hard to determine the exact source code line of a specific security error.

Different scanners typically report these errors at different source code lines. Thus, authors of artificial code have to pay close attention in order to define the exact locations of any errors they include in their code. Control flow and data flow can appear in many different combinations. Natural code does not contain all of these combinations. With artificial code, security scanners can be tested whether they find all these combinations (National Institute of Standards and Technology 2012). Artificial code has advantages, but it also has its limitation. Artificial test cases are typically simpler than what can be found in natural code. In fact, test cases in the Juliet Test Suite are much simpler than natural code. Therefore, security scanners may find something in these test cases but fail at real programs that are much more complex. In addition, the frequency of flaws in the test suite may not reflect the frequency of the bugs of real programs. Again, a security scanner may find many errors in the test suite but not in natural code. And, even if less likely, it can also be just the opposite (National Institute of Standards and Technology 2012).

3 SECURITY SCANNERS

Source code scanners are programs that analyze the static source code of other programs to identify flaws. They typically check the source code, but some may also scan byte code or binaries. Every scanner has a built-in set of weaknesses to look for. Most also have some means of adding custom rules (Black 2009). The rules can target specific weaknesses, e.g., security, but may also check for programming style.

3.1 Software security

Security is about protecting information and information systems from unauthorized access and use. The core goals are to retain confidentiality, integrity and availability of information. Besides IT security, other security terms that are often used include network security, computer security, web security, mobile security, and software security. Software security is "the idea of engineering software so that it continues to function correctly under malicious attack" (McGraw 2004).

Prominent software security bugs include buffer overflows, SQL injections and cross-site scripting. There are many examples, where these bugs have occurred and caused damage. While security bugs are problems at the implementation level, security flaws are located at the architecture or design level. Security flaws are much harder to detect and typically need more detailed expert knowledge. Security scanners target security bugs. Thus, they can help to greatly enhance the security of software, but they are not capable of finding all security problems that may exist in the software they are scanning.

3.2 Security problems

Software security problems have to be uniquely identified and classified. For example, when different scanners use their own classification scheme, comparison is difficult for customers who use more than one tool from different vendors. The databases CVE and CWE have been created for that purpose. CVE stands for Common Vulnerabilities and Exposures (http://cve.mitre.org/). It provides a standard for security vulnerability names and is a dictionary of publicly known information security vulnerabilities and exposures. CWE stands for Common Weakness Enumeration (http://cwe.mitre.org/). It provides a unified, measurable set of software weaknesses for effective discussion, description, selection, and use of software security tools. Both CWE and CVE are free for public use. They roughly contain 1,000 and 60,000 entries, respectively.

If a specific security problem gets known, it is assigned its own CVE number for identification and description. The vulnerability relates to a specific weakness, i.e., to a CWE number. CWE entries are more abstract than CVE entries and identify a class of problems. For example if a heap overflow is found in a software product, this problem gets a CVE number. The CVE number relates to a CWE entry that describes buffer overflows in general. Additionally, security vulnerabilities get scores to classify the severity of the problems. For that purpose, CVSS is used. CVSS stands for The Common Vulnerability Scoring System. It is a "vendor agnostic, industry open standard designed to convey vulnerability severity and help determine urgency and priority of response" (http://www.first.org/cvss).

3.3 Selected scanners

As mentioned above, the Juliet Test Suite contains a Java and a C/C++ part. Thus, it can be used to either test Java or C/C++ scanners. A list of security scanners is given in http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html. We have chosen scanners that are available for free, and that can be started via command line. Free scanners have been chosen because we had done this project only for demonstration purposes.

Commercial scanners may hopefully provide better results than the scanners we have chosen. However, they can be used by the same approach that we will present here. Additionally, we have only chosen scanners that can be started via command line. This requirement was necessary for the automation process. However, with some programming or scripting, tools without a command line interface will be able to be integrated, too. As a result, we have used the following source code scanners for our case study:

- PMD (Java) – version 4.0.3
  PMD is an open source static analysis tool. We have used the GDS PMD Secure Coding Ruleset, a custom set of security rules intended to identify security violations that map to the 2010 OWASP Top 10 application security risks (https://github.com/GDSSecurity/GDS-PMD-Security-Rules). More information on PMD can be found at http://pmd.sourceforge.net.
- FindBugs (Java) – version 2
  FindBugs is also an open source static analysis tool for the Java programming language. It is interesting that FindBugs processes the compiled class files rather than the source code files. More information can be found at http://findbugs.sourceforge.net.
- Jlint (Java) – version 3.0
  Jlint is an open source static analysis tool like PMD and FindBugs. It mainly detects inconsistent software routines and synchronizations problems. Version 3.0 has been released in 2011, indicating that its development may have been discontinued. For more details see http://jlint.sourceforge.net.
- Cppcheck (C/C++) – version 1.58
  Cppcheck is a static analysis scanner for the programming languages C and C++. It detects several different security errors, which are described on the tool's website. See http://cppcheck.sourceforge.net for more information.
- Visual Studio (C/C++) – version 2010
  The VisualStudio Compiler is shipped with Microsoft Visual Studio, Microsoft's development environment. The compiler has an option "/analyze" that lets it analyze the source code for errors. A list of all detected problems is given in the documentation of the compiler. See http://msdn.microsoft.com/vstudio for more information.

We have used these scanners because they were freely available. The only exception is Microsoft Visual Studio Compiler, for which a license had been available. It is important to note that all these scanners are not dedicated security scanners. They detect non-security issues like bad programming habits or violation of programming guidelines. Security is not their primary target. Naturally, the Juliet Test Suite is best suited to compare dedicated security scanners. Our goal was to demonstrate the usefulness of the test suite for that purpose, and also to provide a methodology for an automatic assessment of such scanners.

4 APPROACH

In order to compare different scanners, we have to simply let them scan the test suite and then compare their results. As simple as it sounds, this can be a tedious and cumbersome process for several reasons.

4.1 Error types

First of all, different security scanners report different error types, making it hard to compare the results. For example, one tool may report an SQL injection as an error with type "SQL Injection". Another tool may report an error of "type=2". Therefore, we have a transformation tool that transforms the scanner results into a uniform format that can later be used for comparison. For that purpose, we have mapped the scanner specific error types to CWE numbers that already are used in the test suite. The transformation is split into two steps. First, all the scanner result files are converted to CVS files. For that purpose, a tool called Metric Output Transformer (MOT) is used. This tool is part of a research project called Software Project Quality Reporter (SPQR) (Ploesch et al. 2008). Every line in the CVS files represents one detected error. Each line contains several columns with the name of the scanner, an abbreviation of the error set by the transformer, the path to the file with the error, and the line number where the error was detected. The line ends with the scanner's error message. Listing 1 shows one such line, i.e., the description of one error. Due to space limitations, each column entry is presented in a separate line or two with a comment on the right side.

In the second step, output in the CVS files is used in combination with CWE mapping files. CWE mapping files map scanner-specific results to CWE numbers. Listing 2 shows a small part of the CWE mapping file for the scanner Jlint. As can be seen, CWE mapping files contain tags for the scanner codes. These codes are assigned to CWE numbers.

```
JLint                                   name of scanner
CCNP                                    abbreviation of error
CWE113_HTTP_Response_Splitting          file
__connect_tcp_addCookieServlet_15.java
148                                     line number
Switch case constant 6 can't be         error message
produced by switch expression
```

Listing 1: SPQR output

Multiple CWE numbers can be assigned to a single scanner code, be-cause scanner results are sometimes more abstract than its CWE numbers. For example, a scanner result with code "CPSR" can either be of type CWE 570 or CWE 571.

4.2 Error locations

Another problem for comparison is the fact that security errors are indeed documented in the test suite, but their exact location, i.e., the specific source code line, is not always known. We have therefore developed a small utility program that identifies the appropriate lines in the source code files. For some of the test suite's source code files, we have been able to find additional descriptions that specify the line of the error in an XML file. These additional files had been used whenever they were available. See http://samate.nist.gov for more information. Without these files we were only able to define a range of lines in the source code in which the error was located. With the additional files we were able to determine the exact location where the error occurred. These files have lead to roughly 2,800 (out of 24,000) exactly located errors in the Java version of the test suite and about 23,000 (out of 57,000) errors in the C/C++ version.

4.3 Automated analysis and comparison

For automated comparison, we have developed a simple utility program that is used to start the security scanners and to compare their results. The process is structured into four steps:

1. We scan the Juliet Test Suite and locate the lines of errors as a reference. This needs to be done only once.
2. We start the security scanners to analyze the test suite. The scanners report their results in specific formats.

```
<mappings scanner="JLINT">
  <scannerCode name="AWSMCD"
   desc="* invocation of synchronized method *">
    <cwe id="833" />
  </scannerCode>
  <scannerCode name="AWWID"
   desc="Method wait\(\) can be invoked with monitor of other object locked">
    <cwe id="833" />
  </scannerCode>
  ...
  <scannerCode name="CPSR"
   desc="Comparison always produces same result">
   <cwe id="570" />
   <cwe id="571" />
  </scannerCode>
</mappings>
```

Listing 2: CWE mapping file

3. We convert the result files of the scanners into a uniform format (as shown in Listing 1) for comparison.
4. We compare the results and generate a report that shows the results in human readable form.

Details including the configuration of the individual scanners can be found at https://github.com/devandi/AnalyzeTool.

4.4 Scanner results

When scanners analyze the test suite, they report found errors, cf. step 2. These errors do not always match the errors that in fact are in the test suite, cf. step1. We get a false positive when a scanner reports a problem when in fact there is none. A false negative occurs when a scanner does not report a problem that in fact is existent. Scanners may also report a wrong location for a problem. Thus, we distinguish the following types of reported errors:

- True positive.
  The scanner reports an existing error.
- False positive.
  The scanner reports a non-existing error.
- Wrong positive.
  The scanner reports a specific error at the location of an error with a different type.
- True negative.
  The scanner does not report an error where in fact there is none.
- False negative.
  There is an error in the source code, but the scanner does not find and report it.

In fact, wrong positives are a combination of a false positive and a false negative. We consider them separately, because the reported errors may come close to the real errors. Of course, false positives and false negatives are the most important categories when evaluating a scanner. True positive means that a scanner did report an existing error. Security errors are typically spread over several lines. The question is whether a scanner reports the right location. We distinguish the following cases:

- True positive+: Correct location
  The scanner has reported an error at the same source code line where this error had been documented, assuming that we know the exact location of the error.
- True positive: Unspecified location
  The scanner has correctly reported a specific error. The exaction location is unknown in the source code or has not been specified. We typically know a line range it the error is contained in a (bad) function.
- True positive–: Incorrect location
  The scanner has correctly reported an error, but given a wrong location, which is close enough to be counted as a true positive.

In the Juliet test suite, we have errors where we know either the exact source code line or a range of source code lines, i.e., we have a list of True positive+ and True positive. A scanner should report as many True positives as possible. It is better to have a report of an error with an incorrect location than no report of that error at all, i.e., a True positive- reported by a scanner is much better than a True negative. We can only count a true positive as a true positive+, when we ourselves know the exact location in the test suite.

4.4 Security model

We have used a security model in which CWE entries were combined to more abstract categories. For example, a category "Buffer Overflow" represents different CWE entries that describe types of buffer overflows. We have used these categories according to (Center for Assured Software 2011), to generate a security model as part of a more general software quality model (Ploesch et al. 2008). This security model helps to interpret the scanner results. For example, with the model we can determine a scanner's weak-ness or strength in one or more areas. This information can be used to choose a particular scanner for a software project. Thus, if a scanner is weak in identifying authentication problems but is strong in other areas, this scanner can be used for projects that do not have to deal with authentication. Alternatively, an additional scanner can be used with strong authentication results. In general, the security model helps to analyze found errors in more detail. Another advantage of having a security model is to provide a connection between generic descriptions of software security attributes and specific software analysis approaches (Wagner et al. 2012). This allows us to automatically detect security differences between different systems or subsystems as well as security improvements over time in a software system.

5 RESULTS

The Juliet Test Suite consists of a Java and a C/C++ test suite. We will start with Java. Subsequently the results for C/C++ will be presented. Finally, an overall discussion of the findings will follow.

5.1 Java

In the Java test suite we had far more true positives with unspecified location (True Positive) than such with a conclusive location (True Positive+), which were determined with the additional 'SAMATE' files. Consequently, the scanners can only deliver a few conclusive locations. Figure 1 contrasts the percentage of errors with a conclusive location, i.e., True Positives+, to errors with an unspecified location, i.e., True Positives. Figure 2 shows the distribution of the test cases by the security model. We can see that the numbers of test cases are not balanced for every category. As a result, scanners, which find many issues in categories with many test cases, are better in this comparison than other scanners. Apparently, the category "Buffer overflow" has no test cases at all. This should not come as a surprise as the Java Runtime Environment prevents buffer overflows in Java programs. Figure 3 shows an overview of all errors that were detected by the different scanners. The entry Juliet on top shows the actual number of documented errors in the test suite. We can see that for a small percentage the exact location of the error is known, but for most errors this is not the case. Apparently, FindBugs has detected the most errors, followed by Jlint and PMD. However, PMD has found more exact locations than Jlint and FindBugs. As Fig. 3 shows, the numbers of True Positives+, True Positives, True Positives- and Wrong Positives are higher than the numbers of Jlint and FindBugs. Thus, it can be said that PMD has found the most accurate errors regarding to the test

suite. A deeper analysis of the results has shown that the GDS rule set used within PMD were responsible for the results of PMD. Without them, PMD would not have found any errors in the test suite. Nevertheless, the overall results were poor. The scanners have detected only less than half of the errors of the test suite. The figures do not explicitly show the false negatives of the scanners. They can be determined by comparing a scanner's true positives to the test-suite's true positives.

5.2 C/C++

In the C/C++ version there were more errors with a conclusive location (True Positives+) than in the Java version but more than 60 % had an unspecified location (True Positives). Figure 4 shows the distribution of the True Positives+ and the True Positives of the C/C++ version. Figure 5 shows the distribution of the test cases by the security model. As we can see the number of test cases per category is not balanced either. The category Buffer Handling has the most test cases because problems in this area are common security issues in C/C++ programs. Figure 6 shows an overview of all detected errors by the tested scanners. The Visual Studio Compiler found far the most errors. But most of them were not covered by the test suite and were non-security issues. A deeper analysis of the results had shown that the scanner had found many issues where a function was used to allocate memory in the test cases, which should not be used. As such functions were used many times within the test cases, this has lead to the high number of found errors. Despite that, the Microsoft Scanner found more errors than Cppcheck. However, very few errors were found in the test suite. It should be mentioned that this scanner had the longest run time. We have not taken exact runtime measurements, but most of the scanners took approximately one hour to scan the test suite. The Visual Studio Compiler took about 11 hours. We had used a virtual machine on Windows 7, 32-bit with one CPU on an Intel i5-2500 @ 3.3 GHz with 2 GB assigned main memory.

5.3 Discussion

The results of the analysis may lead to the conclusion that such security scanners should not be used or that they are of only little value. We have to keep in mind that these scanners are not dedicated security scanners. Also, the example of PMD shows that if special rule sets for security are used, then the results improve considerably. Furthermore, these scanners are not a replacement to security reviews from security experts. The scanners cost less and are much quicker than such reviews. Moreover, the scanners can be integrated in an automated build process for software development. The Microsoft Visual Studio Compiler can be used in a way that the source code is analyzed every time the file gets compiled during development. Even though the results are behind expectations, every found and corrected error reduces the attack surface of the software under construction. For example, a third-party source code analyzer could easily have found Apple's recent 'goto fail' bug in its encryption library (McCullagh 2014). In the results, we have seen large number of false positives, i.e., scanners report errors that are not documented in the test suite. In fact, the scanners often did not report real false positives, but rather they reported issues that were not unusual in artificial code like 'Local declaration hides declaration of same name in outer scope'. Such issues are not documented in the test-suite, because the focus is on security. Therefore, many of these false positives are only false positives in relation to the documented security issues of the test-suite.

6 RELATED WORK

Rutar et al. have tested several source code scanners including PMD and FindBugs (Rutar et. al 2004). For that purpose, they wrote a meta program to automatically start the scanners. For their study, they used open source programs containing natural code. Therefore, they additionally had to perform security reviews to determine errors in the tested programs.

Due to the effort it takes, they were only able to review part of the scanned software. The reviewed parts were then used to extrapolate an overall result. The findings showed that scanners found different results that did not overlap.

The Center for Assured Software also performed a study in which they tested different security source code scanners (Center for Assured Software 2001). For their analysis they also used the Juliet Test Suite. The names of the used security scanners were not published. For the comparison they also transformed the scanner specific results into a uniform format. Furthermore, they generated some key figures, which were used to compare the scanners. They used "weakness classes" to classify the different errors. These weakness classes have been used as the foundation for the security model in our approach.

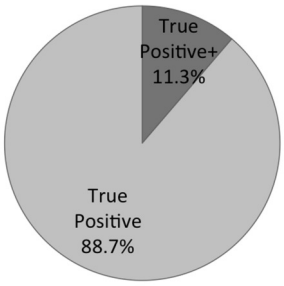

Figure 1: Percentage of Java errors with conclusive location

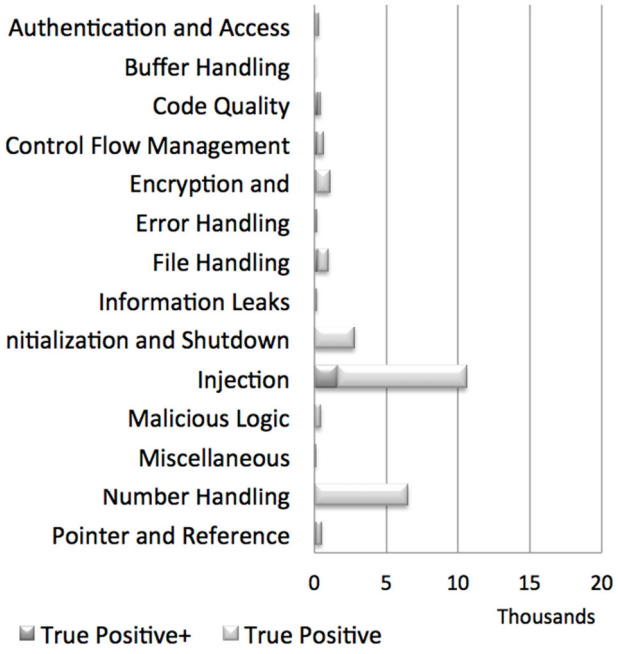

Figure 2: Java test case distribution

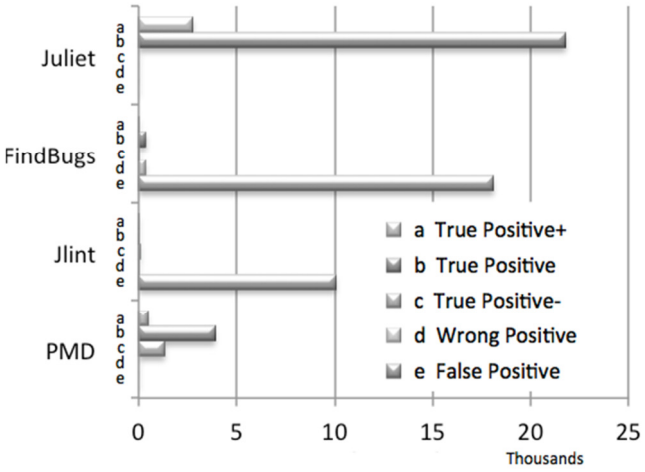

Figure 3: Java scanner results

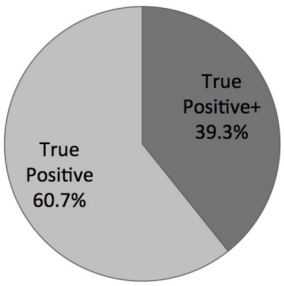

Figure 4: Percentage of C/C++ errors with conclusive location

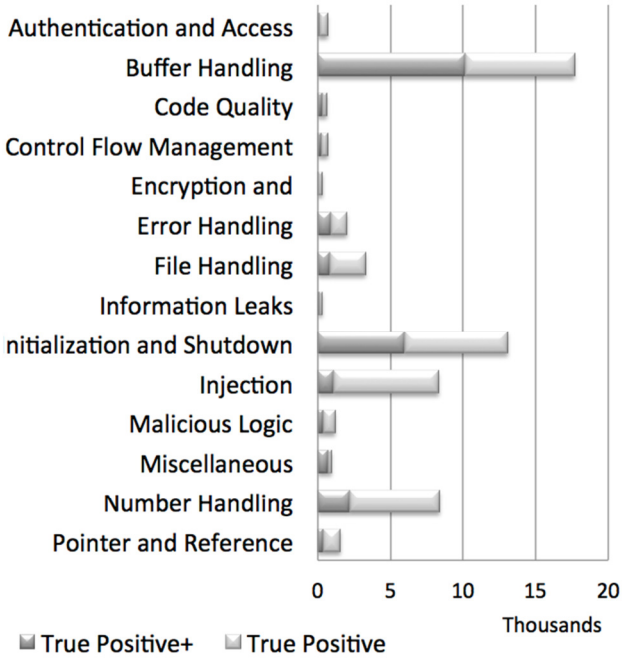

Figure 5: C/C++ test case distribution

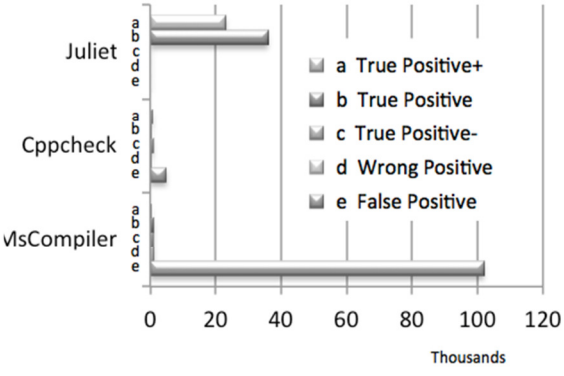

Figure 6: C/C++ scanner results

A student project had been performed at CERN, the European Organization for Nuclear Research (Hofer 2010). Different scanners for several programming languages like Java, Python and C/C++ were analyzed. The goal of the analysis was to make recommendations for scanner usage and to increase software security considerations at CERN. But locating the errors in software was only one of five classes, which were used for the classification of the scanners. They also used classes like installation and configuration to evaluate the scanners. The problem was that they did not analyze how accurate the scanners found some errors. It was only measured how much errors where found but not whether found errors were correct.

7 CONCLUSIONS

We have tested static source code analyzers with the Juliet test suite that contains security bugs in Java and C/C++ code. We have used scanners that were not primarily targeted at security, because dedicated security scanners require expensive licenses. The general results of the scanners were disappointing with the exception of the open source static analysis tool PMD that we had used with secure coding rules. We draw the following conclusions from our simple test. First, static source code analysis is a simple and efficient means to detect bugs in general and security bugs in particular. Second, it is advisable to use more than one tool and, thus, combine the strengths of different analyzers, And third, dedicated test suites like the Juliet Test Suite provide a powerful means of revealing the security coverage of static scanners, which in turn enables us to pick the right set of scanners for improved software coding. We have also shown that scanner results can be used to assess the security quality of software systems. Scanner results can automatically be used to generate a security model and, for example, indicate security improvements or degradation over time.

REFERENCES

Black, P.E., 2009. Static Analyzers in Software Engineering. *CROSSTALK–The Journal of Defense Software Engineering*. https://buildsecurityin.us-cert.gov/resources/crosstalk-series/static-analyzers-in-software-engineering

Boland, T., Black, P.E., 2012. Juliet 1.1 C/C++ and Java Test Suite, *Computer*, vol. 45, no. 10, pp. 88-90, DOI: 10.1109/MC.2012.345

Center for Assured Software, 2011. *CAS Static Analysis Tool Study – Methodology*, December 2011. http://samate.nist.gov/docs/CAS%202011%20Static%20Analysis%20Tool%20Study%20Methodology.pdf

Hofer T., 2010. *Evaluating Static Source Code Analysis Tools*, Master Thesis, InfoScience 2010. http://infoscience.epfl.ch/ record/153107

Howard M., Lipner S., 2006. *The Security Development Life-Cycle*, Microsoft Press.

McCullagh D., 2014. Klocwork: Our source code analyzer caught Apple's 'gotofail' bug, *clnet* February 28. http://news.cnet.com/8301-1009_3-57619754-83/klocwork-our-source-code-analyzer-caught-apples-gotofail-bug/

McGraw G., 2004. Software Security, *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80-83. doi:10.1109/MSECP.2004.1281254

McGraw G., 2009. *Software Security: Building Security In*, 5th edition, Addison-Wesley.

MITRE 2011. CWE/SANS Top 25 Most Dangerous Software Errors, Version 1.0.3. http://cwe.mitre.org/top25/

National Institute of Standards and Technology 2012. *SAMATE Reference Dataset*. http://samate.nist.gov/SRD/testsuite.php.

Plösch, R., et al., 2008. Tool Support for a Method to Evaluate Internal Software Product Quality by Static Code Analysis, *Software Quality Professional Journal, American Society for Quality*, Volume 10, Issue 4.

Rutar, N., Almazan, C.B., Foster, J.S., 2004. *A Comparison of Bug Finding Tools for Java*, IEEE, ISBN 0–7695–2215–7, 245–256.

Wagner, S., et al., 2012. The Quamoco Product Quality Modelling and Assessment Approach, *Proceedings of 34th International Conference on Software Engineering* (ICSE 2012), Zurich.