TOOL EVALUATION REPORT: FORTIFY

Derek D'Souza, Yoon Phil Kim, Tim Kral, Tejas Ranade, Somesh Sasalatti

ABOUT THE TOOL

Background

The tool that we have evaluated is the Fortify Source Code Analyzer (Fortify SCA) created by Fortify Software. Fortify software is a software security vendor of choice of government and Fortune 500 companies in a wide variety of industries. They provide products that identify and remediate security vulnerabilities in software in order to mitigate enterprise security risks.

The Fortify SCA tool attempts to protect systems from security flaws in business-critical software applications. Fortify SCA drives down cost and risk by automating and enhancing key software audit, development, testing and deployment processes. Unlike traditional network security, Fortify SCA strengthens the software applications themselves so that hackers and malicious insiders cannot access vital assets or disrupt business processes.

In addition to the Fortify SCA tool some of their other products are as follows: Fortify Tracer - the first and only automated solution that makes Black Box security testing actionable and measurable, Fortify Tester - the first black box security testing plug-in for Microsoft Visual Studio, Fortify Defender - the first and only intrusion prevention software solution for web applications already in deployment, and Fortify Manager - the definitive software security risk management dashboard.

(www.fortifysoftware.com)

Languages/Platform supported

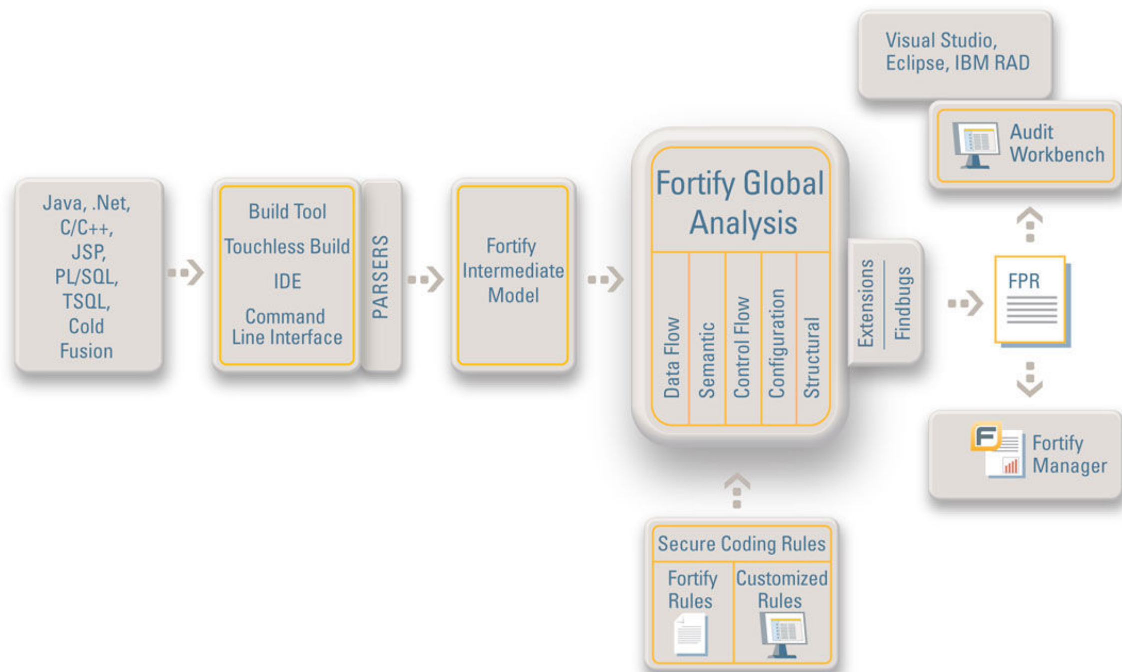
Fortify SCA supports a wide variety of languages, frameworks and operating systems.

- Languages: ASP.NET, C/C++, C#, ColdFusion, Java, JSP, PL/SQL, T-SQL, XML, VB.NET and other .NET languages
- Platforms: Windows, Solaris, Linux, Mac OS X, HP-UX, AIX
- Frameworks: J2EE/EJB, Struts, Hibernate

- IDEs: Microsoft Visual Studio, Eclipse, Web Sphere Application Developer, IBM Rational Application Developer

How it Works

Fortify SCA is a static analysis tool and it processes code in a manner similar to a code compiler. It uses a build tool that runs on a source code file or set of files and converts it into an intermediate model that is optimized for security analysis by Fortify. This model is put through a series of analyzers (Data flow, Semantic, Control Flow, Configuration, and Structural). Fortify SCA also uses Fortify Secure Coding Rule Packs to analyze the code base for violations of secure coding practices. Fortify Rules Builder allows to extend and expand analysis capabilities to include custom rules as well. The results can be viewed in a number of ways using the Audit Workbench and the Fortify Manager.



(http://www.fortifysoftware.com/products/sca/scaHowItWorks.jsp)

Capabilities/Features

Fortify SCA is used to find and fix following software vulnerabilities at the root cause: Buffer Overflow, Command Injection, Cross-Site Scripting, Denial of Service, Format String, Integer Overflow, Log Forging, Password, Management, Path Manipulation, Privacy Violation, Race Conditions, Session Fixation, SQL Injection, System Information Leak, and Unreleased Resource. It sorts, filters, prioritizes

and categorizes the issues found in various forms for easy viewing and analysis through the Audit Workbench and Fortify Manager. The results can also be exported as a report in formats such as html and xml. The SCA tool supports integration with black box security testers such as Watchfire's Appscan and it also supports integration with Findbugs for central security and quality issue reporting.

TOOL AND EXPERIMENTS SETUP

Tool Setup

The SCA tool comes in three editions: Enterprise, Team, and Developer. We used the Team Edition which is intended for smaller organizations, and consists of Fortify SCA, Fortify Audit Workbench, and Fortify Rules Builder. We believe this edition will be the most relevant and interesting to other MSE and MSIT students because the majority of studio and practicum projects are being done in a Windows environment with small teams. The tool was downloaded with the evaluation license from www.fortifysoftware.com

Artifacts

We ran Fortify SCA on the following artifacts:

- OpenSSH (Large C/C++ code base) – we used the tool on this project to test its analysis on a large well-known code base.
- Small C/C++ application injected with bugs – we created a small test file that we injected with bugs to test precisely for false positives, false negatives, and effectiveness of the tool in finding bugs.
- JSPWiki (Large Java code base) – we used the tool on this project to evaluate it on a different language (Java) and to compare it with existing bug reports on the tool to test its efficacy.
- Crystal (Small Java code base) – we used the tool on Crystal to evaluate a project that we have actively been using this semester and have some familiarity with.

How to Use

An analysis can be performed with the Fortify SCA tool in two steps:

1) Use the command line to run the sourceanalyzer on the project source files and obtain a .fpr file.

2) Use the Audit workbench or Fortify Manager on the .fpr file to explore the results of the analysis.

The first step in performing an analysis involves running a series of commands at the command line prompt to build the source files. For Java projects, the classpath and the source folder or file must be provided. For C/C++ projects, a makefile is required. Once the build is successful, a .fpr file is generated on which we ran the Audit Workbench to analyze the results.

The screenshot shows the Fortify Audit workbench used to analyze the Crystal code base:

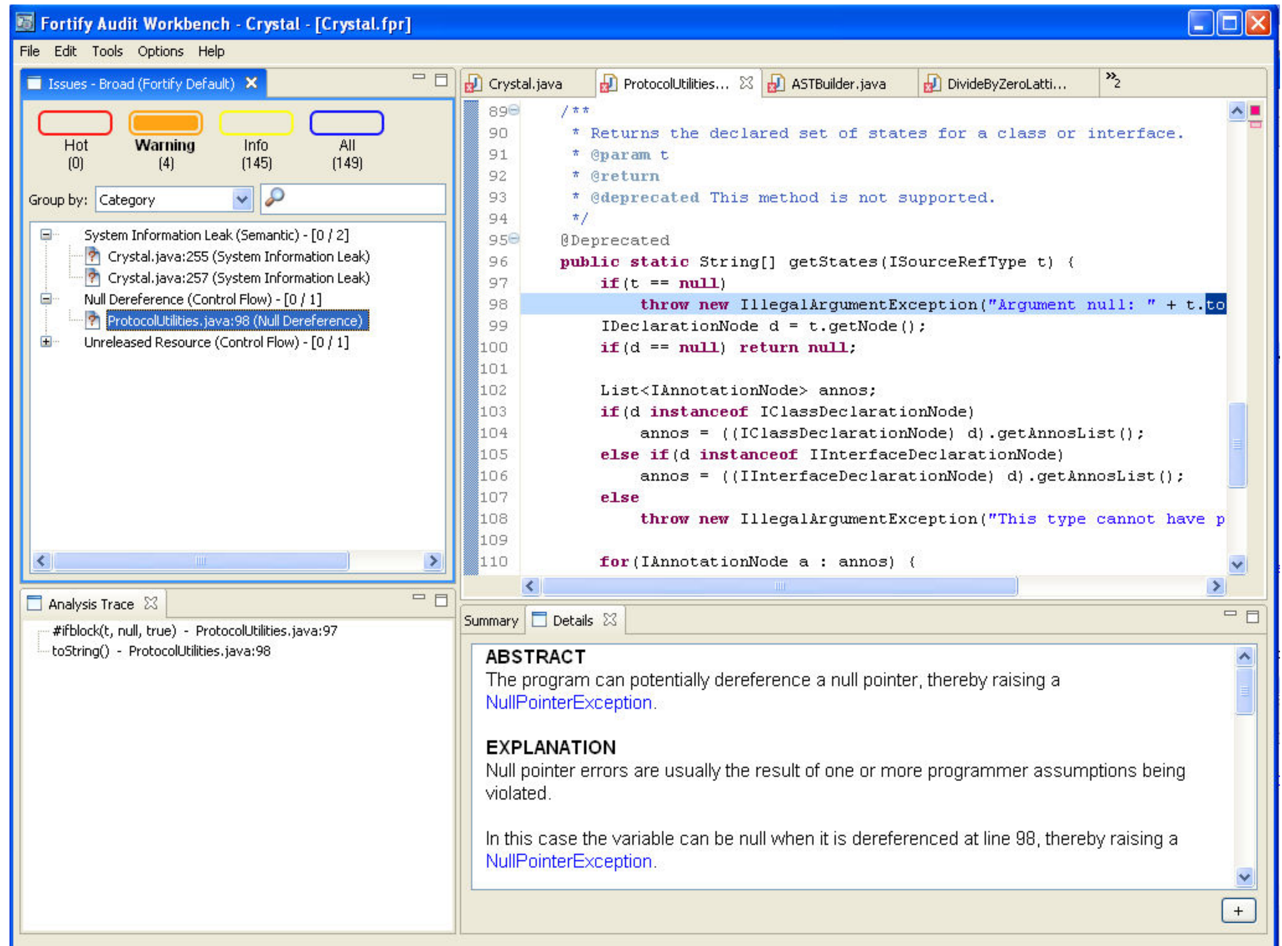


Fig 1: Screenshot of Fortify SCA Audit Workbench

The list of issues grouped as "Warnings" can be viewed here with the Null Deference bug highlighted. It zooms in on the piece of code that contains the bug. The bottom right box shows a summary of the bug as well as details in separate tabs. The bottom left box gives the bug trace found in the analysis.

EVALUATION CRITERIA

Performance

We measured the performance in terms of the time required for the analysis to complete, memory usage, and CPU usage. These criteria helped us determine if the tool was reasonable in its usage of resources, and how it performed on different codebases. Our evaluation was limited to a large C codebase, and a medium sized Java codebase. An interesting observation was that even on a large code base like OpenSSH, SCA's performance was quite impressive. Although the custom build took long, the actual scan time of 2.45 minutes to scan 53000 lines of code across 152 files is noteworthy.

Criteria		OpenSSH	JSPWiki
Size of the code base (KLOC)		53	35
Language		C	Java
CPU Usage		30-50%	50%
Memory usage (K bytes)		8000-10000	2500
Execution time (minutes)	**Compilation**	17	0.3
	Scanning	3	1

Number of bugs and quality of the findings

The tool reports bugs in four categories: (1) semantic, (2) data flow, (3) structural and (4) control flow. In addition, bugs are categorized as "Hot issues", "Warnings" and "Information". Although the number of bugs reported, particularly for OpenSSH and JSPWiki is high, the categorization of bugs makes it much easier to understand and prioritize them. The tool also allows the user to configure the prioritization of bugs, which makes it possible to specify what you believe is important for the system being analyzed.

On OpenSSH and JSPWiki, the bug lists have entries in the order of a few hundred. One can notice from the bug list that there several types of errors found. Each of them relates to some security vulnerability that can be exploited. Some of the common ones are buffer overflows, unreleased resource errors, memory leaks etc. In addition to these, some bugs that can be very important in security analysis were also reported. For example, the tool reports denial of service bugs, which can occur when a thread sleeps for variable time. Another useful report was that of command injections for command executions that use relative paths. Such a bug can be exploited by a malicious user by changing the environment variable to point to malicious code. Fortify identifies parts of the code where such paths are used.

On Crystal, the tool reported 4 bug warnings and 145 miscellaneous notes. The finding that was most useful was a null deference bug that would have caused a null pointer exception every time that piece of code was run. There were two instances where getStackTrace() was called that the tool reported as System Information warnings since it could potentially provide important system information which can be exploited by an attacker. In addition, the tool found an instance of an unreleased resource after

getPrintWriter() was being called. The 145 notes provided useful information in categories of issues such as poor error handling, poor logging practices, model violations, and dead code found in Crystal.

Level of Detail in the Reports

For all the issues the tool finds, the level of detail is sufficient to gain an understanding of what the problem is. Double clicking on a specific issue zooms in on the portion of the problematic code. Fortify provides both summary and detailed descriptions of all bugs found. Detailed summaries include a full explanation about the type of bug found, why it is a bug, where in the current source code it is occurring (including data flow traces if applicable), fully explanatory examples of the bug in sample code and in some cases, references for further reading about the bug. The tool also provides sample code to illustrate how the issue can potentially be exploited by an attacker of the system.

Suggested fixes, fixing time

For each of the issues found, the tool provides a list of recommendations and tips. Some of these are helpful, but some are not. For example, for the null deference bug the tool's recommendation was

"Security problems caused by dereferencing null pointers are almost always related to the way in which the program handles runtime exceptions. If the software has a solid and well-executed approach to dealing with runtime exceptions, the potential for security damage is significantly diminished."

This does not provide any particularly useful information for fixing the bug and seems more like background information on the issue. However, for some other bugs, the recommendations also contained useful sample code which can be used as reference for fixing the bug.

Given that some of the recommendations are quite specific, the fix time for bugs is low. Even for the reports that do not provide very useful suggestions, the location of the bug can be pin-pointed, which makes it a lot simpler to debug. We found Fortify to be more intuitive for fixing bugs than most of the analysis tools we have used during this course.

False Positives:

On the whole, the tool does not report a large percentage of false positives in comparison to the number of bugs reported. It does; however, report some false positives. The Fortify tool website also acknowledges this in a disclaimer.

Among the false positives we noticed, here is an example:

- The tool assumes that dynamic memory is freed within the scope of a function. As a result, if custom free methods are called, it throws a potential memory leak. A code snippet that illustrates this is shown below:

```
{
        int *ptr = (int*)malloc(2*sizeof(int));
```

```
        Myfree(ptr);
}
Myfree(int *ptr){
free(ptr);
}
```

In this snippet, the memory allocated to *ptr* is released in the method *Myfree(int *).* However, the tool does not understand that the memory is freed in another method which is outside the scope of *ptr.* This is a common source of false positives in the SSH source code.

- Another source of false positive that we observed was the report of a SQL injection at places where we did not find any hint of a dynamic SQL query generation. This type of result was observed in JSPWiki code.

False Negatives / Overlooks:

One of our interesting investigations involved trying to find false negatives. Our approach, as documented in the proposal, was to compare the results of the tool with the bugs reported on the bug list in an open source forum. We did not have a substantial degree of success finding open source software which we could use that had well documented bug-lists. However, we did find an instance of a false negative in the analysis of JSPWiki. Here is the sample.

```
1 String password = profile.getPassword();
2 String existingPassword = ( existingProfile == null ) ? null : existingProfile.getPassword();
3 if ( "".equals( password ) ){
4           password = null;
5 }
6 if ( password == null){
7 password = existingPassword;
8 }

9 // If password changed, hash it before we save
10 if ( !password.equals( existingPassword ) )
…
```

There is a null dereference error in line 10 of the code displayed above. Although *password* is being checked previously for null value, it is actually being assigned to *existingPassword* (line 7). Now *existingPassword* can very well be null, as indicated by line 2. Therefore, there is a possibility of a null dereference in line 10, but Fortify fails to catch it. This bug was one of the bugs noticed in the forum, and corrected thereafter.

(Ref. http://jspwiki.org/wiki/BugNullPointerExceptionWhenSavingTheUserProfileJDBC)

Another observation was that the tool does not identify shallow copy as an illegal operation. If an alias to a pointer is created, then the tool does not warn that an inappropriate operation might result.

Usability:

Good	Bad
An audit workbench is available for viewing the results and analyzing the issues. The audit workbench we used was an Eclipse plug-in. The workbench is very easy to use, and it reports the bugs in an understandable fashion; including color-coding for severity, hierarchies based on bug types and file structures. It also allows easy location of bugs in the source code.	The intermediate files and the analysis files need to be built from a command line interface. The analysis file can then be opened in the workbench. This process is especially painful on machines that are not configured as test machines. We spent a considerable time trying to 'make the analysis work', particularly on C and C++ codebases.
Other than the files that need to be generated automatically, the tool does not require any annotations, specification or customization to the source code or the tool. Thus, the time spent in preparing for the analysis is very low.	A usability bug exists in the audit workbench. If the source file is edited while it is being viewed in the workbench, then the line numbers for the errors can get skewed.

Interesting bug types:

Some of the more interesting bugs caught by the tool are as follows:

- Least Privilege Violation: There is a point in the code where the privilege level to execute the code was escalated, but was not reverted back immediately after the operation. This vulnerability can be exploited by an attacker by injecting code which can be run in an escalated privilege.
- Insecure temporary file. There is a point in the code of SSH where an insecure temporary swap file is being created. Fortify identifies that this temporary file is not secured.
- Command Injection: There are several places in the code where relative paths are being specified for files and commands. If an attacker changes the $PATH env variable which is being used, then he can get a malicious binary to execute.

Practicality:

The bugs reported by Fortify are very relevant particularly from a security standpoint. Also, they are good indicators of overall code quality and coding practices. The bug list cannot be considered complete in all respects. However, we believe that it is worthwhile to spend time fixing the bugs that the analyst is concerned about. There might be certain classes of bugs, or certain areas in the application that need to be given attention to. Fortify SCA presents the bugs by order of configurable importance, as well as by location. Also as noticed before, the bug reports are comprehensive ensuring relatively low fix time. Considering the large number of bugs, it will be an overhead to try to fix each one. Therefore, the categorization can be used to identify what to fix. This definitely makes the tool very practical and cost effective.

LESSONS LEARNED

Benefits

- The Fortify SCA tool provides powerful analysis techniques, including patent pending multi-tier data flow analysis and interactive results viewing. It helps prioritize security risks and thus helps to bring high risk vulnerabilities to the forefront.
- Fortify SCA efficiently processes large, complex code bases and presents discovered security issues in a well categorized form that makes reporting and remediation significantly faster.
- Fortify SCA uses the knowledge base of secure coding guidelines and thus brings required expertise and consistency to developers and allows security experts to leverage their knowledge across more projects.
- The tool provides support for multiple languages and platforms.
- The tool is easy to setup and does not require any extra work other than building the source files.

Drawbacks

- The SCA tool cannot catch design intentions or analyze the existing design for soundness.
- There are some usability issues in the build phase of using the tool since this has to be performed through the command line and with the availability of makefiles in some instances.
- Fortify SCA suffers from what is called the "black box problem". For reactive applications and heterogeneous systems, execution does not always take place in available application code. For instance, in reaction to a mouse click, a reactive application can start executing in kernel code to pass the event over and around the operating system. This part of its execution can rarely be analyzed and therefore, static analysis tools can hardly determine what type of data comes out of these calls. Thus, this prevents true inter-procedural analysis.
- Fortify SCA has issues with discovering bugs from aliasing and considering method calls (it does not cover all execution paths).

Scope of application

The SCA tool is a useful tool that can be used during the implementation phase while code is being written and also during the testing phases of the software lifecycle. Fortify also provides other tools such as the Fortify Defender which can be used during the production phase.

CONCLUSIONS

If there are no trial restrictions we plan to continue using this tool to analyze our source code. The tool can be used to supplement code inspections since it gives a list of critical security flaws in the code as well as bad coding practice issues. Security is an attribute that is not very easy or tangible to test for with traditional methods, and this tool can significantly help in this area while at the same time helping developers gain an understanding of how certain issues in code can result in security flaws. Identifying these issues early in the lifecycle with the help of this tool can reduce cost of fixing bugs later on down the line once the implementation is complete. This tool is especially useful for teams whose projects will be using or will be used in distributed systems that are open to remote access. For example, the Google team is developing protocol code for an open source library called Libjingle that can be used for file transfers between distributed machines. One of the MSIT practicum teams is developing an online version of the Risk board game that will involve multiple players remotely accessing the game. In these cases, security of the system is critical and this tool can help ensure that the flaws in code that can be exploited by attackers are minimized. In general, this is a very useful tool that can provide a robust and easy to examine analysis that should be used in projects where security is a high priority quality attribute.