Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Tools for static code analysis: A survey

by

Patrik Hellström

LIU-IDA/LITH-EX-A--09/003--SE

2009-02-06

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Avdelning, Institution
Division, Department

Division of Computer an Information Science
Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2009-02-06

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LIU-IDA/LITH-EX-A--09/003--SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version

Titel
Title

En undersökning av verktyg för statisk kodanalys

Tools for static code analysis: A survey

Författare
Author

Patrik Hellström

Sammanfattning
Abstract

This thesis has investigated what different tools for static code analysis, with an emphasis on security, there exist and which of these that possibly could be used in a project at Ericsson AB in Linköping in which a HIGA (Home IMS Gateway) is constructed. The HIGA is a residential gateway that opens up for the possibility to extend an operator's Internet Multimedia Subsystem (IMS) all the way to the user's home and thereby let the end user connect his/her non compliant IMS devices, such as a media server, to an IMS network.

Static analysis is the process of examining the source code of a program and in that way test a program for various weaknesses without having to actually execute it (compared to dynamic analysis such as testing).

As a complement to the regular testing, that today is being performed in the HIGA project, four different static analysis tools were evaluated to find out which one was best suited for use in the HIGA project. Two of them were open source tools and two were commercial.

All of the tools were evaluated in five different areas: documentation, installation & integration procedure, usability, performance and types of bugs found. Furthermore all of the tools were later on used to perform testing of two modules of the HIGA.

The evaluation showed many differences between the tools in all areas and not surprisingly the two open source tools turned out to be far less mature than the commercial ones. The tools that were best suited for use in the HIGA project were Fortify SCA and Flawfinder.

As far as the evaluation of the HIGA code is concerned some different bugs which could have jeopardized security and availability of the services provided by it were found.

Nyckelord
Keywords Static analysis, Software security, IMS, HIGA

Institutionen för datavetenskap
Department of Computer and Information Science

Master's Thesis

Tools for static code analysis: A survey

Patrik Hellström

Reg Nr: LIU-IDA/LITH-EX-A--09/003--SE
Linköping 2009

Supervisor: **Mattias Törnqvist**
Cybercom Group AB

Examiner: **Nahid Shahmehri**
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Abstract

This thesis has investigated what different tools for static code analysis, with an emphasis on security, there exist and which of these that possibly could be used in a project at Ericsson AB in Linköping in which a HIGA (Home IMS Gateway) is constructed. The HIGA is a residential gateway that opens up for the possibility to extend an operator's Internet Multimedia Subsystem (IMS) all the way to the user's home and thereby let the end user connect his/her non compliant IMS devices, such as a media server, to an IMS network.

Static analysis is the process of examining the source code of a program and in that way test a program for various weaknesses without having to actually execute it (compared to dynamic analysis such as testing).

As a complement to the regular testing, that today is being performed in the HIGA project, four different static analysis tools were evaluated to find out which one was best suited for use in the HIGA project. Two of them were open source tools and two were commercial.

All of the tools were evaluated in five different areas: documentation, installation & integration procedure, usability, performance and types of bugs found. Furthermore all of the tools were later on used to perform testing of two modules of the HIGA.

The evaluation showed many differences between the tools in all areas and not surprisingly the two open source tools turned out to be far less mature than the commercial ones. The tools that were best suited for use in the HIGA project were Fortify SCA and Flawfinder.

As far as the evaluation of the HIGA code is concerned some different bugs which could have jeopardized security and availability of the services provided by it were found.

Acknowledgments

I would like to thank my tutors at the company, Mattias, Johan and Jimmy for helping me with all sorts of problems, both theoretical and practical.

I would also like to thank all the people at the office for being so supportive and helpful throughout this project.

Finally I send a big thank you to my girlfriend, Sandra, for her great support!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter will give the reader an introduction to the thesis. It contains a background, which questions that are to be answered and the goal of the thesis. It also contains an overview of the thesis layout.

1.1 Background

Imagine that you are sitting on the bus and have nothing to do but staring out the window when suddenly you remember that yesterday you downloaded the latest episode of your favorite TV show to your home multimedia server. You pick up your IMS (Internet Multimedia Subsystem) compliant mobile phone and connect to your multimedia server at home and after a minute or two the TV show is streaming to your phone with perfect quality, making your bus ride a little more enjoyable. This is a scenario that could be happening in a not so distant future.

The IMS is an architectural framework for delivering multimedia over the Internet protocol (IP) to mobile users and was initially standardized by the 3rd Generation Partnership Program (3GPP) as a new service layer on top of IP-based 3G networks back in 2003 [15]. Since then a lot has happened and today services such as Push to Talk over Cellular (PoC) and TV over IP (IPTV) are some of the features available in the IMS.

One new service that today experiences heavy research on Ericsson AB is the Home IMS Gateway (HIGA). This is a residential gateway which will act as a bridge between the outer IMS services mentioned before and the inner, non-IMS home devices, such as TVs, PCs or media servers. In other words it will give the users the freedom of accessing their home computers etc. with the help of their phones wherever they might be. However, this freedom comes with the drawback that it opens up yet another way into a user's private network and it is of crucial importance that the gateway is secure and robust against all types of attacks[1] . But how can one make sure that it really is secure and is able to withstand most

[1]Not only attacks that aim at breaking in to a users private network, but also attacks whose goal is to bring down the HIGA and thereby bringing down the availability of the services it provides.

attacks? The simple answer is that one can not, but in order to get as close as possible to a secure system one should always have security in mind throughout the whole Software Development Life Cycle (SDLC) and make continuous tests of the system.

1.2 Goal

The goal of this thesis is to investigate what different static code analysis tools, with an emphasis on security, there exist and find alternatives that can be used in the future work with the HIGA.

Furthermore a static analysis should be made on part of the HIGA source code, and a statement concerning security in the present implementation should be made.

1.3 Question at issue

The question at issue is: What alternatives considering tools for static code analysis, with an emphasis on security, are there that can be used in the work with the development of the HIGA? The following five areas concerning the evaluation of the tools will be reviewed:

- Documentation
- Installation and integration procedure
- Usability
- Performance
- What types of bugs the tools are able to find

1.4 Restrictions

Since this thesis does not have any financial budget the tools that are to be examined should be either open source or commercial with an evaluation license that can be obtained for free.

Since most code in the HIGA project is written in the language C, the chosen tools have to support this language.

1.5 The company

This thesis is commissioned by the company Cybercom Sweden West AB. Cybercom is a high-tech consulting company that concentrates on selected technologies and offers business-critical solutions, in a number of different segments such as portals, mobile solutions, embedded systems, e-commerce, and business support system.

The company was launched in 1995; it was listed on the Stockholm stock exchange in 1999. Today, the Cybercom Group has offices in eleven countries: China, Denmark, Dubai (UAE), Estonia, Finland, India, Poland, Romania, Singapore, Sweden and UK, with about 2000 employees [1].

1.6 Structure

The second chapter of the report contains a theoretical background which covers IMS and software security (with an emphasis on static analysis). Chapter three brings up the survey of the different tools which were evaluated and the fourth chapters describes how the tests of the HIGA source code were performed and the results of the tests. In the fifth chapter the results of the work as well as the conclusions reached are presented.

Chapter 2

Theoretical Background

This chapter serves as an introduction to the IMS and software security with an emphasis on static analysis.

2.1 IMS

Poikselkä et al. [17] define the IMS as:

> *a global, access-independent and standard-based IP connectivity and service control architecture that enables various types of multimedia services to end-users using common Internet-based protocols*

More simply put one can say that the IMS is an architectural framework for delivering multimedia to the end-users with the help of the Internet Protocol (IP).

The IMS has its foundation in the 3G mobile system. The first release of 3G was specified by the 3GPP in the year of 1999. After this release the 3GPP started investigating the possibility to include the IMS in the next specification but it took as long as to the fifth release, which was released in March 2002, before IMS became part of the specification. This stated that the IMS should be a standardized access-independent IP-based architecture that should interwork with existing voice and data networks for both fixed (e.g. PSTN, ISDN, Internet) and mobile users (e.g. GSM ,CDMA) [17]. As the word access-independent suggests it does not matter how a user entity connects to the system as long as it uses the IP protocol. Furthermore the system should provide quality of service (QoS), support for charging, security, roaming, registration and more.

2.1.1 Architecture

The IMS-system is a very complex distributed system with a lot of different servers and databases. A generalized view of an "IMS system" is depicted in figure 2.1.

As seen in the figure the IMS has a layered design consisting of three different layers: "Transport Layer", "Control Layer" and "Application Layer".

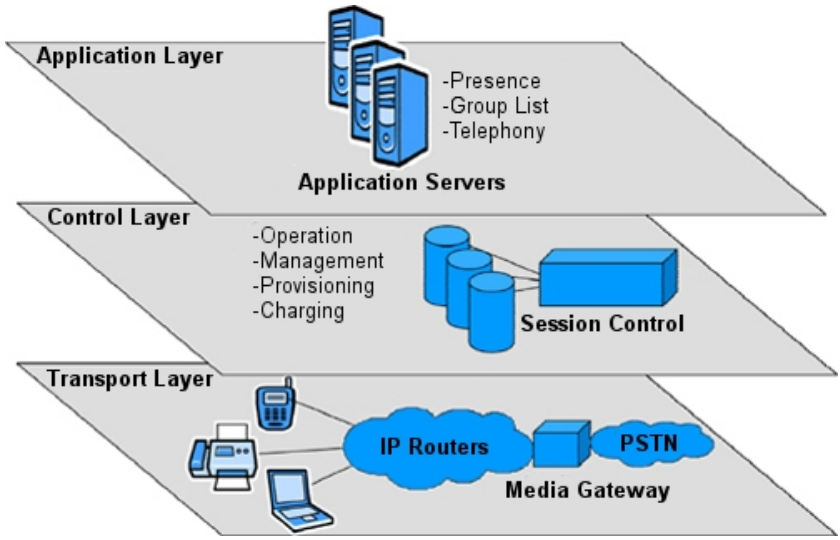

Figure 2.1. Generalized view of the IMS

The transport layer consists of for example an IP network or a non-IMS network such as the Public Switched Telephony Network (PSTN).

The control layer consists of Session Initiation Protocol servers (SIP servers) which handle all session management and routing of SIP messages, boundary gateways which provide the functionality of translating SIP messages and connecting IMS sessions to non-IMS networks. This layer also holds various databases such as the Home Subscriber Server (HSS) which contains information about the users.

The application layer contains a number of different application servers which provide the users with services such as for example Push to Talk over Cellular (PoC), presence and conferencing.

2.1.2 HIGA

The Home IMS Gateway (HIGA) is an approach to extend an IMS network all the way to a customer's home (the connected home is often heard in this context). Products such as TV, computers and "regular" telephones can thus be connected via a HIGA to an operator's IMS network. The functionality the HIGA provides is simply a translation between the IMS/SIP protocols used in the IMS network and home network protocols such as uPnP (Universal Plug and Play) [3].

2.2 Software Security

Software security is the practice of building software to be secure and to function properly under malicious attacks [16]. This is not to be confused with security

software which is software whose purpose is to secure a computer system or a computer network. The aphorism "*software security is not security software*" is often heard in software security contexts and it is of essential importance to get people aware of the difference; it does not matter if you have a really good anti-virus program or a fancy firewall if the programs themselves are not written with security in mind and hence contain a lot of flaws and bugs.

When talking about software security, words as *bug, flaw, defect* etc. are often used in an inconsistent manner and quite often it is easy to misinterpret what the author of an article or a book really means when he/she for example writes the word *bug*. The following list presents a short basic terminology (with an emphasis on security) as seen in [16] which will be used throughout this thesis.

Bug A bug is an implementation-level software problem which might exist in code but never be executed. In many cases the term bug is applied quite generally but in this thesis it will be used to describe fairly simple implementation errors such as a buffer overflow.

Flaw In contrast to a bug, a flaw is a problem at a deeper level and is present already at the design level.

Defect Both implementation vulnerabilities (bugs) and design vulnerabilities (flaws) are defects.

In order to detect defects a variety of methods can be used of which code auditing as well as testing are some of the methods used actively in the field. If one want to automate the process and use tools, only bugs can be discovered as there do not yet exist any automated technologies to discover design-level flaws [16].

2.2.1 Auditing Source Code

A code audit is an analysis of source code which is performed with the intent of discovering bugs, flaws, security issues or violations of programming conventions [4].

Gartner vice president, John Pescatore, once said that

> *Removing only 50 percent of software vulnerabilities before use will reduce patch management and incident response costs by 75 percent [8].*

With this in mind one realizes that auditing software for defects is very important and should start in an early stage of the Software Development Life Cycle.

When auditing source code one can use a number of different methods of which manual code review and automated source code analysis are two of the more common ones [8].

- **Manual code auditing**
  In manual code auditing the developer, or whoever is doing the audit, examines the code by hand in order to find defects. The advantage of this method

is that it can be used to find both design flaws and bugs. However, the downside are that it takes a huge amount of time to conduct and it requires very good programming skills as well as security knowledge (if this is one of the reasons that the audit is being performed) from the one performing the audit. Another big disadvantage is that it is a very tedious task with the consequence that the result might not be very accurate since most people get tired of it rather fast [16].

- **Automated source code analysis**
  In order to make the process of code auditing more efficient (and affordable) one can use a tool to perform the audit. There are plenty of tools that are capable of inspecting both source code, binary code and byte code and present the result of the scan in a user friendly environment. One thing to keep in mind though is that it is not a fully automatic process as the result of the scan still has to be analyzed by a person in the end. A drawback with using a tool, as already pointed out, is that it can not be used in order to find flaws.

When performing a code audit with the purpose to improve security it is a good idea to know what kind of coding mistakes there are and how they affect security. A person who is making a manual audit needs to know what a specific code error might look like and he/she also needs to know what the consequence(s) are of that specific error. Also when using a tool this knowledge is of importance, not only to the person conducting the audit, but also to the tool that needs knowledge about what to search for. Since there are a lot of different coding errors it is a good idea to somehow categorize them, and there exist a lot of different taxonomies of coding errors on the Internet, some better than others. The following list presents a taxonomy (The Seven Pernicious Kingdoms Taxonomy) consisting of seven categories as seen in [16][1]. For a list of specific coding errors for each category please refer to Appendix A.

1. **Input Validation and Representation**
   Problems in this category are caused by trusting input to the program and not making any validation of it. Examples on consequences can be buffer overflows, cache poisoning and SQL injections.

2. **API Abuse**
   An API is a contract between a caller and a callee. The most common form of API abuse are caused by the caller failing to honor its end of this contract.

3. **Security Features**
   Handles topics as authentication, access control, confidentiality, cryptography and more. These things are hard to get right and cause devastating consequences if they are not.

[1] The reason this taxonomy was chosen was of the simple fact that it is very straightforward and only consists of seven categories, making it easy to adopt.

4. **Time and State**
   With multi-core, multi-CPU or distributed systems, two (or more) events may take place at exactly the same time. Problems in this category concern the ones with interaction between threads, processes, time and information.

5. **Error Handling**
   Error handling not performed the right way, or not at all, is a major concern when talking about software security. Even if handled correctly they might be a problem. What if the error message provides too much information which can be used by a potential attacker?

6. **Code Quality**
   Poor code quality leads to unpredictable behavior of the program. From a users point of view this often results in bad usability but for an attacker it can be exploited to stress the system in unexpected ways.

7. **Encapsulation**
   Concerns the boundaries between for instance data and users. Examples might be that an applet on the Internet can not get access to your hard drive or that data might leak between users in a shared system.

As mentioned earlier this list concerns different coding mistakes/design errors and how they affect security of a program. This is of course not the only reason why audits of source code are performed. As John Pescatore said it is of crucial importance to find as many defects as possible before a product is shipped out and those defects do not necessarily have to result in security breaches.

2.3 Static Analysis

Static analysis is the process of examining the text of a program statically, without trying to execute it (in contrast to dynamic analysis, e.g. testing) [12]. With this definition manual auditing also falls under the category of static analysis. However, this thesis focuses solely on automated static analysis.

Static analyzers can be used for a variety of purposes such as finding security related vulnerabilities, bugs, type checking, style checking, program understanding, property checking and software metrics [12].

When using static analysis in order to find bugs it is good to know what types of bugs they might be able to detect. Emanuelsson et al. [14] presents a (non-exhaustive) list that contains four main areas where static analysis might be used successfully:

- **Improper resource management:** Resource leaks of various kinds such as dynamically allocated memory that is not freed, files, sockets etc. which are not properly deallocated when no longer used.

- **Illegal operations:** Things like division by zero, over- or underflow in arithmetic expressions, addressing arrays out of bounds, dereferencing of null pointers etc.

- **Dead code and data:** Code and data that cannot be reached or is not used.

- **Incomplete code:** The use of uninitialized variables, functions with unspecified return values and incomplete branching statements (for example no else branch in conditional statement).

Static analysis could also be used in order to check for deadlocks, non-termination and race conditions etc. [14].

2.3.1 History of Static Analysis

The first tools that were developed to scan source code in order to find security related problems were all very simple. The only thing they did was to perform a lexical analysis of the code. This type of analysis can in many ways be compared to just performing a simple search through the code, looking for functions and code constructs that might lead to a security related error [16]. However, performing only a simple search with for example the Linux utility Grep[2] together with a list of "good search strings" to search the code after known dangerous function calls etc. comes with the problem that it does not understand anything about the file it scans, making it impossible to separate actual code from comments. In order to get around this issue the lexical rules of the programming language have to be considered and this is what tools like ITS4[3], Flawfinder[4] and RATS[5] all do [12]. These are three of the earliest static analysis tools that can be used to search for security vulnerabilities (ITS4 was released in the year 2000) and they all perform a basic lexical analysis. All of them begin with preprocessing and tokenizing the source file that is to be analyzed and then match the token stream against a library of known vulnerable constructs.

The next step taken to enhance the tools was to equip them with even more intelligence. Even though that they now knew how to separate code from comments they did not account for the target code's semantics and as a result the outcome of the analysis contained a lot of false positives. By implementing more compiler technology, such as the ability of building an abstract syntax tree (AST)[6] as well as a symbol table from the source code, some basic semantics of the program could be taken into account [12]. Most modern tools today have gone one step further and in addition to just supplying source code audit functionality most of them give the user a whole suite of different functionalities such as code-browsing and paths-exploring.

[2]Global Regular Expression Print. Used to search for lines of text that match one or many regular expressions.

[3]It's The Software Stupid Security Scanner, http://www.cigital.com/its4/

[4]http://www.dwheeler.com/flawfinder/

[5]Rough Auditing Tool for Security, http://www.fortify.com/security-resources/rats.jsp

[6]An AST is a data structure that represents something that has been parsed. It is often used by compilers and interpreters as an internal representation of a program and acts as the base from which code generation is performed [2].

2.3.2 How does it work?

Most of today's tools for static analysis function in basically the same way. They receive code as input, build a model that represents the program, perform some sort of analysis in combination with knowledge about what to search for and finally present the result to the user. Figure 2.2 shows the process that all static analysis tools that target security make use of [13].

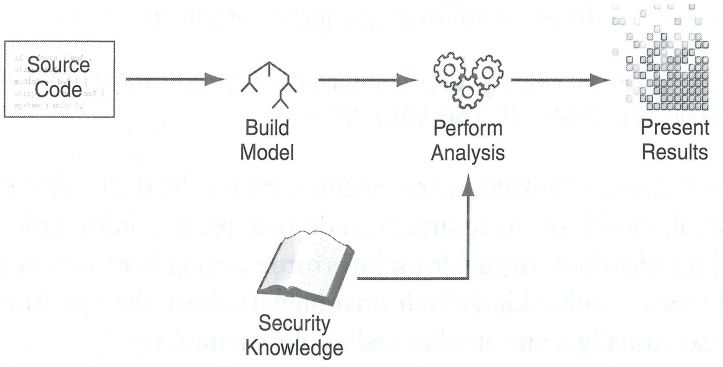

Figure 2.2. Generalized view of the process of a static analysis

The first step taken by the tools when performing an analysis is to build a model of the program. In this way an abstract representation of the program is created, which is better suited to be used for the actual analysis. What kind of model that is created depends largely on what kind of analysis that is to be performed. The simplest model, used in a lexical analysis, is a stream of tokens. This is created by taking the source code, discarding all unimportant features such as whitespaces and comments and finally translate all parts of the code into tokens. Figure 2.3 shows a simple example [13].

```
Code:
if (test) //TODO: Fix this!
 index[i] = temp;

Stream of tokens:
IF LPAREN ID(test) RPAREN ID(index) LBRACKET ID(i)
RBRACKET EQUAL ID(temp) SEMI
```

Figure 2.3. Example of a translation of code into a stream of tokens

This stream can then be used to perform a simple lexical analysis on. This is the way ITS4, RATS and Flawfinder all work.

The next step is to translate the stream of tokens into a parse tree by using a parser that matches the token stream against a set of production rules. The parse tree contains the most direct representation of the code just as the programmer wrote it and can thus be used as a good base to perform analysis on [13]. However, since the production rules might introduce various symbols in order to make the parsing easy, a parse tree is not the best representation to perform more complex analyzes on. The next step is instead to translate the parse tree into an AST as well as creating a symbol table alongside it. These can then be used as input to a semantic analysis of the program [13]. Another thing that the AST might be used for is to scan it by using predefined patterns in order to find errors that require data-flow analysis of the code [11]. This is not possible to perform on for example a token stream.

The procedure of performing a static analysis is until now very alike the procedure taken by a compiler when compiling a program. But as a compiler as a next step will use the AST to generate an intermediate representation of the code, which can be used for optimization and later on translation to platform-specific code, the processes now separates.

Most of the tools instead continue by building a control flow graph, which can be used to inspect the various paths in the program or to track data flow, on top of the AST. The way the graph is used by a tool depends largely on which techniques the tool makes use of and this has a big impact on efficiency, speed and result of a scan.

Another thing that influences the result of a scan is how the tool makes certain simplifications of the program. The very essence of a static analyzer is that the program being analyzed is not run. However, in order to perform a thorough scan the tool, as already described, translates the code into an abstract representation and then "executes" the program on an abstract machine with a set of non-standard values replacing the standard ones. This is where a concept refereed to as *states* is introduced. States are a collection of program variables and the association of values to those variables [14]. An example of where state information is used is when determining if a statement like **x/y** may result in a division by zero.

For every program statement the state of a variable may change in some way and the aim for a static analysis is to associate the set of all possible states with all program points. Yet another thing that influences the state is the scope of the analysis. In an intra-procedural analysis the state only considers local variables whilst a context-sensitive analysis also takes account for global variables and contents of the stack and the heap.

All this makes the number of sets of states very big (sometimes even infinite), leading to that the tool must make approximations and make use of simplified descriptions of sets, which in turn lead to a less-than-perfect output.

The use of states and how a tool makes the approximations and the simplified descriptions of sets is something that makes the tools differ greatly from one and another. Some make very sophisticated and accurate decisions in order not to make unjustified assumptions about the result of an operation whilst others resort to a more simple approach.

2.3.3 Techniques and Precision

As mentioned in the previous part all tools take their own approach to how to make use of the control flow graph, and there exist a number of different techniques a static analysis tool can make use of. All influencing the precision of the tool as well as the time it takes to conduct an analysis. If a tool makes a *flow-sensitive* analysis it means that it takes into account the control-flow graph of the program in contrast to a *flow-insensitive* which does not. The advantage with a flow-sensitive analysis is that it usually gets more precise since it for example "knows" *when* a certain variable may be aliased whereas a flow-insensitive only knows *that* the variable may be aliased, the drawback is of course that a flow-sensitive analysis takes more time and cpu power.

Another technique concerns if the analyzer is *path-sensitive* which means that it considers only valid paths trough the program. This requires the tool to keep track of values of variables and boolean expressions in order to avoid branches that are not possible. A *path-insensitive* analyzer on the other hand takes all execution paths into account, even the impossible ones. As in the case with a flow-sensitive analyzer a path-sensitive analyzer implies higher precision but at the cost of time and cpu power.

The third technique is known as *context-sensitive* and deals with such things as global variables and parameters of a function call in order to make a more precise analysis. This is sometimes referred to as *interprocedural* analysis in contrast to *intraprocedural* analysis which analyzes a function without any assumptions about the context. The intraprocedural analysis is faster but at the cost of a more imprecise analysis. Most tools that make use of an advanced analysis strategy usually begins with an intraprocedural analysis for every individual function and then performs an interprocedural analysis for analyzing interaction between the functions [13].

When talking about precision the concept of *false-positives* and *false-negatives* are of great importance. A false-positive in the context of static analysis means a bug, reported by the tool, that does not exist in the code. A false-negative on the other hand is a bug in the code that the tool fails to discover. The presence of false positives and false negatives in the result of a scan is consequently not wished for since they both have a negative impact on the result of an analysis. If the result contains many false positives the auditor will have real problem with finding the real bugs. However, if the tool on the other hand produces a lot of false negatives this will lead to a false sense of security which is even more dangerous.

Sometimes one talks about if a static analysis tool is *sound*. A sound static analysis tool produces no false-negatives (probably at the cost of more false-positives [14]), i.e. all defects in the code are found by the tool.

2.3.4 Advantages and disadvantages with static analysis

Even though that the use of static analysis can find a lot of bugs before the program is run it is not supposed to replace testing of the program. A lot of bugs and particularly flaws may be found better and easier with extensive testing compared to static analysis. However, the advantages with static analysis include

for example that no test cases are necessary, no test oracle is needed, it can find "hard" bugs such as for example memory leaks, and the analyzed program does not have to be complete (still it is not recommended as it probably leads to a lot of false positives [14]). On the downside the method usually produces false positives, which in turn have to be analyzed, and to understand the reports produced by the tool the auditor will need good programming competence.

2.3.5 Key characteristics of a tool

When choosing a tool to be used for static analysis in a project one has to ask oneself what the focus of the analysis should be. If for example security is one of the main concerns one should choose one that have a security module. McGraw [16] mentions six characteristics that a tool should have (and three that it should not have) in order to be useful and cost effective:

1. **Be designed for security**
   Tools that focus purely on software quality is good to some extent when it comes to robustness, but tools with a security module have more critical security knowledge built in to them and the bigger the knowledge base a tool have the better. In many cases a security breach might also have more costly business impacts than do standard-issue software risks.

2. **Support multiple tiers**
   Today not many programs are written solely in one language or targeted to a single platform. More often the application is written in a number of different languages and runs on many different platforms. For a tool to be successful it must have the capability of supporting many languages and platforms.

3. **Be extensible**
   Nothing ever stays exactly the same and this goes for security problems as well. They evolve, grow and new ones are discovered every now and then. A good tool needs a modular architecture that supports a number of different analysis techniques. In that way when a new attack is discovered the tool can be expanded to find them as well. Furthermore the tool should have the ability to let the users add their own rules.

4. **Be useful for security analysts, QA teams and developers alike**
   The tools should make it possible for the analyst to focus their attention directly on the most important issues. Furthermore it should support not only analysts but also the developer who need to fix the problems discovered by the tool.

5. **Support existing development processes**
   It should be easy to integrate the tool with existing build processes and Integrated Development Environments (IDEs). In order to make the tool accepted it has to interoperate well with used compilers and build tools.

6. **Make sense to multiple stakeholders**
   The tool needs to support the business. Different views for release managers, development managers and executives can support for example release decisions and help control rework costs.

Three characteristics that should be avoided are “too many false positives”, “spotty integration with IDEs” and “single-minded support for C”.

This list is of course not the one truth to follow when choosing a tool for static analysis. Instead it could be used as input to a discussion about what characteristics are the most important in one particular project/company.

Chapter 3

Survey

Today there exist a number of different vendors of static analysis tools. This chapter presents six of the major commercial tools as well as three open source tools[1]. Out of these, four (two commercial and two open source) were evaluated thoroughly.

3.1 Available tools today

3.1.1 Commercial Tools

Fortify 360 by Fortify Software Inc.

Fortify Software Inc. was founded in 2003 and has since then provided their customers with tools for finding security vulnerabilities in software applications. The company runs their own security group which among other things maintains the Fortify taxonomy of security vulnerabilities[2]. The security group also provides the company's customers with quarterly updates, which give the tool information about new types of vulnerabilities and support for more programming languages.

The main product is Fortify 360 which is a suite of tools consisting of one static analyzer as well as two dynamic analyzers, working together in order to find as many vulnerabilities as possible. Furthermore it supports most of the most common programming languages and platforms.

The tool in the suite suitable for this thesis is the *The Fortify Source Code Analyzer* (SCA) which is the tool performing a static analysis. It can be run on most operating systems and provide functionality for scanning C/C++, Java, .Net, PL-SQL, T-SQL and ColdFusion code.

[1] There exists a number of open source tool for static analysis. The reason these three were chosen in an initial step was that they seemed like perfect candidates for this thesis since they all have functionality for finding security related bugs.

[2] http://www.fortify.com/vulncat/

Coverity Prevent by Coverity™ Inc.

During the years of 1998-2002 a research group at Stanford University began developing a static analyzer which later on would be the foundation for Coverity Prevent. In 2003 the team released their first version of the tool and it did not take long as it got recognized by experts and industry. Not long afterwards the group began to apply the technology for commercial products.

Coverity Prevent can be used for static analysis of C/C++ and Java source code and it supports most platforms and compilers. It also supports many IDEs in order to be used directly by developers when coding or it can be used as part of the central build system.

Ounce 5 by Ounce Labs Inc.

Ounce Labs was founded in 2002 and their tool Ounce 6.0 (released in July 2008) is a tool that focus purely on software security. The tool supports both developers with support for integration in different IDEs as well as audit and quality assurance (QA) teams. The tool supports C/C++/C#, Java, VB.NET, ASP.NET and more.

CodeSonar by GrammaTech Inc.

GrammaTech's CodeSonar has been available for three years and is used to find vulnerabilities in C/C++ source code. The functionality in CodeSonar makes use of the company's other tool called CodeSurfer. CodeSurfer is a code browser which can be used when performing a manual code review and it is based on research conducted at the University of Wisconsin.

Klockwork Insight by Klockwork Inc.

Klockwork Insight can be used to check C/C++ and Java source code and it can be used directly by developers on their desktops or at system build. The tool supports a wide variety of IDEs, platforms and builds environments.

PolySpace™ by The MathWorks Inc.

The MathWorks provide PolySpace client and server respectively for C/C++. The client is used for management and visualization as it is used to submit jobs to the server and to review test results. The benefit with a client/server solution is that a user can use multiple servers in order to accelerate the analysis or let several users use one server and let many individuals or teams view the result at the same time.

3.1.2 Open Source Tools

Two of the tools chosen in this category are lexical analyzers that focus on finding security vulnerabilities. The tools are not as advanced as the commercial tools but they are very fast and can be used as an initial step to identify dangerous code areas etc. The drawback is that they produce a lot of false positives.

The third tool performs the scan on the semantic level and is thus more advanced than the former two tools. It is not comparable to the commercial ones though.

RATS by Fortify Software Inc.

RATS, short for Rough Auditing Tool for Security, is a tool for scanning C/C++, Perl, PHP and Python source code for common programming errors concerning security. Since the tool only performs a rough analysis of the source code, manual auditing is still necessary but the tool might help the analyst to focus on the most critical parts.

RATS works in the way that it searches through the source code after known function calls which might give rise to a vulnerability. It then presents a list of potential problems as well as a description of each problem. In some cases it also suggests a solution to the problem.

The latest release of RATS, release 2.1, was in 2002 and is available as a binary for Windows as well as a source tarball.

Flawfinder

Flawfinder functions in the same way as RATS in that it searches for known security vulnerabilities using a built-in database containing known problems. It then produces a list with the problem sorted by risk. The supported programming languages that can be analyzed are C and C++.

As in the case with RATS, Flawfinder does not understand the semantics of the code and it has no capabilities of doing a control- or data flow analysis.

The latest release of Flawfinder, version 1.27, was in 2007 and works on Unix-like systems (but should be easy to port to Windows according its the homepage).

Splint

Splint version 3.1.2 was released in 2007 and is used to check C source code. The program runs on Unix-like systems and comes with more intelligence than RATS and Flawfinder in that it works on the semantic level and is able to take the control flow in consideration when making the analysis. Furthermore it makes use of annotations which are a special form of comment. These are used to document assumptions about functions, variables, parameters and types, making Splint capable of performing a more precise analysis of the source code.

Splint does in contrast to RATS and Flawfinder not only focus on finding purely security related vulnerabilities but on coding mistakes that affect general code quality as well.

3.2 Requirements for Further Evaluation

In order to be further evaluated some requirements had to be fulfilled by the suggested tools in the previous part.

The first requirement that had to be fulfilled concerned what languages and platforms the tool supported. The main part of the code for the HIGA is written in C in a Linux environment and consequently the tool had to support (at least) the C language as well as provide support for the *gcc* compiler and the *make* build tool.

The tool should have some sort of possibility to examine the code from a security perspective.

As this thesis does not have any budget one last requirement was that the tool should have an evaluation license if it was commercial.

Below is a summary of the requirements.

- The tool must support analysis of code written in C, the gcc compiler and the make build tool.
- The tool must provide functionality for finding security vulnerabilities.
- The tool must be either open source or have an evaluation license if it is commercial.

The tools fulfilling the requirements are presented below:

- RATS
- Flawfinder
- Splint
- The Fortify Source Code Analyzer
- Coverity Prevent
- CodeSonar
- Klockwork Insight

Out of these Flawfinder, Splint, The Fortify Source Code Analyzer and CodeSonar were chosen to be further evaluated. For a detailed review on how the tools were chosen refer to Appendix B.

Ounce 5 did not meet the requirement of supplying an evaluation license and PolySpace did not provide support for performing an analysis with an emphasis on security.

3.3 Tool Theory and Background

3.3.1 Flawfinder

Flawfinder is one of the earliest lexical analysis tools and the most recent version, 1.27, uses a built-in database containing 160 C/C++ functions that might be dangerous from a security perspective. The types of vulnerabilities it scans for are

buffer overflow, format string problems, meta character risks, race conditions and poor random number generation.

Flawfinder begins with matching the source code against the names in the built-in database and produce a list of "hits" which could be potential security vulnerabilities. The list is then sorted by risk. The risk level is determined by combining the risk of the function and the values of the parameters of the function. If the value for example is a constant string this is calculated as less risky than a fully variable string.

Since Flawfinder is a lexical analysis tool it produces a hefty amount of false positives. In order to reduce them to some extent Flawfinder is able to tell the difference between comments, strings and the actual code [18].

3.3.2 Splint

In 1979 the tool Lint became part of seventh version of the Unix operating system. Lint was used to review C source code in order to find suspicious constructs which could be bugs.

LCLint, developed by David Evans et al. on University of Virginia, is a successor of Lint, and when it in 2002 was enhanced to also detect security bugs, it became known as Splint [9].

Compared to Flawfinder, Splint also checks for a lot of errors that are not strictly security vulnerabilities. Errors scanned for include type mismatch, memory leaks, null dereference, use of un-initialized formal parameters, undocumented use of global variables, buffer overflows and more.

Another difference between Flawfinder and Splint is that it works on the semantic level of the code in contrast to just doing a lexical analysis. In this way the analysis gets more powerful and extensive than in the case with just searching the code for functions that might be risky to use.

David Evans et al. says that Splint makes use of lightweight analysis techniques that require more effort than using a compiler but not nearly as much effort as to perform a full program verification. In order to do this, compromises has to be made and as an example of this Splint makes use of heuristics to assist in the analysis. As a consequence Splint is not sound and it produces false positives [12].

Annotations

Annotations are stylized comments (or semantic comments) and are used to specify certain pre-conditions and post-conditions about functions, variables, parameters and types. The more effort that is put into annotating the source code, the better the analysis gets. An example of an annotation could be to declare that a pointer value may be null by using the */*@null@*/* annotation. One common cause of program failure is dereferencing of a null pointer but if a pointer is annotated with the */*@null@*/* annotation this implies that the code must check that it is not null on all paths leading to a dereference of the pointer. Figure 3.1 holds two different functions returning a char pointer which is annotated to that it may be null. The function `firstChar1()` does not check if the returned value might be null and this

will make Splint produce a warning. In the other function, `firstChar2()`, `s` is checked not to be null before returned and since this is ok Splint will not produce a warning.

```
char firstChar1 (/*@null@*/ char *s)
{
   return *s;
}
char firstChar2 (/*@null@*/ char *s)
{
   if (s == NULL) return '\0';
      return *s;
}
```

Figure 3.1. Splint /*@null@*/ annotation example

Annotations are also used by Splint to reduce the number of false positives. A "simple" lexical analysis produces a lot of false positives since it only searches for "dangerous" functions, and produces a warning every time it finds one. Splint on the other hand has its own annotated standard library that have more information about function interfaces than those in a system header file [10]. Consider for example the strcpy function that takes two `char` * parameters and copies the string pointed at by one of the pointers to the location pointed at by the other pointer. If the destination to which the string is copied is not as big as the string itself, a buffer overflow will occur. This is checked for by Splint by using extra annotations on the strcpy interface in Splint's library *standard.h* which is the ISO C99 Standard Library modified for Splint. Figure 3.2 shows the annotated strcpy function in *standard.h*.

```
Void /*@alt char * @*/
  strcpy (/*@unique@*/ /*@out@*/ /*@returned@*/ char *s1, char *s2)
     /*@modifies *s1@*/
     /*@requires maxSet(s1) >= maxRead(s2) @*/
     /*@ensures maxRead(s1) == maxRead (s2) /\ maxRead(result) ==
     maxRead(s2) /\ maxSet(result) == maxSet(s1); @*/;
```

Figure 3.2. Splint strcpy interface in standard.h

The interesting annotations that deals with the buffer overflow problem is the pre-condition *requires* which uses two buffer attribute annotations, `maxSet` and `maxRead`. The value of `maxSet(b)` gives the highest index `i` such that `b[i]` can be set to a value. `maxRead(b)` on the other hand gives the highest index `i` such that `b[i]` can be read. When a call to the function strcpy is being made, Splint checks to see if the pre-condition *requires* is met and if not it produces a warning.

Consider for example the function call `strcpy(d, s)`. If Splint cannot determine that `maxSet(d) >= maxRead(s)` this might indicate a possible buffer overflow and a warning will be produced.

3.3.3 Fortify SCA

The Fortify Source Code Analyzer (SCA) is part of Fortify Software's solution Fortify 360 which is a complete suite for finding security vulnerabilities in software. Other than SCA, Fortify 360 also includes functionality for program trace analysis and real-rime analysis.

The most recent version of SCA, version 5.0, was released October 2007. Due to limitations in obtaining an evaluation license from Fortify, the version investigated in this thesis is a demo version of Fortify SCA 4.0, which was released in 2006.

Fortify SCA is an analyzer that focuses on finding security related vulnerabilities that could be exploited by a potential attacker. Since new attacks and hacking techniques are getting more and more sophisticated by the day, Fortify releases updates to there customer every quarter of the year. The updates come in the form of something referred to as *Fortify Secure Coding Rulepack*. These Rulepacks build up the core of the Fortify solution and contains information needed by the tools in order to find possible bugs in the code. The rulepacks are released by *The Security Research Group* which is an internal group at Fortify consisting of researchers trying to find out how real-world system fails. The results of their research are then integrated in the Secure Coding Rulepack, allowing the tools of doing an even more thorough scan of the program being analyzed.

The SCA is built-up from five different engines: Data flow, Semantic, Control Flow, Configuration and Structural analyzers. Vulnerabilities checked for include among others buffer overflow, cross-site scripting, information leakage, log forging, memory leakage, process control and SQL injection.

When performing a scan, Fortify SCA begins with translating the source code into an intermediate format. This is then used as input to the actual scan of the code in which an inter-procedural scan is performed in order to make the evaluation of the code as accurate as possible. The tool is not sound and produces false positives [5].

3.3.4 CodeSonar

CodeSonar is one of the major products of GrammaTech. The other major tool they provide is called CodeSurfer which is code browser. CodeSurfer is best used when doing a manual audit of the code in that it can help the auditor of getting a more complete view of the code and how different things are connected to one and another. The technology in CodeSurfer also serves as a foundation to the one in CodeSonar which is the automatic static analysis tool developed by GrammaTech. CodeSonar does, unlike Fortify 360, not take an equally strong emphasis on trying to find security related vulnerabilities but focuses more on code quality (even so CodeSonar has support for finding a lot of possible security vulnerabilities). For a complete list of the bugs found by CodeSonar refer to Appendix D.

When CodeSonar examines a program it starts of by monitoring a build of the program being audited and in this way learns the build process. This results in that the one performing the audit does not have to concern himself/herself with trying to replicate the build process for the tool. The next step CodeSonar takes is parsing the code and generating an AST, symbol table information, call graphs and control-flow graphs of the program. This abstract representation of the program is then linked together. As an example of the linking procedure the linking of control-flow graphs (CFG) can be mentioned. Each procedure in the program has its own CFG and the linking procedure merges all of the individual CFGs into one whole-program CFG.

When the linking is done, CodeSonar starts performing an interprocedural analysis of program paths. This analysis aims at finding out about feasible paths, program variables and how they relate. The analysis is then followed by an interprocedural, path-sensitive path exploration and when an anomaly is detected CodeSonar generates a warning [6].

CodeSonar 3.1 was released in 2008 and is the version evaluated in this thesis.

3.4 Evaluation Criteria

The procedure of choosing what to evaluate consisted of two major parts. The first part concerned the six different key characteristics of a tool that were introduced in chapter 2.3.5. The second part was then performed by using these points as a base for a discussion with team members on the HIGA project, aiming at finding out which things that should be of interest of investigating and evaluating about the chosen tools.

The questions that were decided to be investigated can be categorized in a total of five major categories: documentation, installation & integration, usability, performance and types of defects found.

Since the key characteristics from chapter 2.3.5 were used only as a base to determine what to evaluate, not all of them are actually satisfied. The first characteristic which states that a tool should be designed for security is not evaluated. The reason for this is that this characteristic in this thesis has been interpreted such that a tool should have some “functionality of finding” security related bugs as opposed to be “designed” for security. Since one of the criterions all of the tools had to fulfill in order to be further evaluated was exactly this, this characteristic was determined to already be satisfied. Furthermore the fourth characteristic states that a tool should be useful for security analysts, QA teams and developers alike. This was not considered to be of interest since the only user of the tool in the HIGA project would be the developer. Because of this, this characteristic is rewritten to that the tool should be useful from a developer’s point of view. The sixth characteristic which says that a tool should make sense to multiple stakeholders is due to the same reason not evaluated at all.

3.4.1 Documentation

The availability of good and thorough documentation in the form of user manuals as well as specifications on exactly what different bugs that can be found by the tool was the main focus when reviewing the documentation. Another thing that was investigated was if there existed any documentation that provided information about the different bugs searched for, in the forms of for example the impact of a certain bug as well as what could be done to avoid them. The quality of the documentation was measured according to the answers of the following questions.

- Was the manual easy to understand, that is, was it clear?
- Was there any information on how to get started with the tool?
- Was there a list of the different bugs searched for by the tools?
- If such a list exists, does it provide information about the bugs?

This category does not relate to any of the six characteristics but was nevertheless considered to be of interest to evaluate.

3.4.2 Installation & Integration

The installation procedure of the tool was considered a relatively small issue but was nevertheless evaluated to make sure that they did not account for any negative surprises. The procedure of integrating the tool with an existing build environment was on the other hand considered a more important issue. Which programming languages the tool could scan was also considered to be of interest. The evaluation aimed at answering the questions below.

- Was the installation procedure straightforward?
- What platforms are supported?
- Does the tool support for integration with a build environment?
- What integrated development environments (IDEs) are supported?
- What programming languages are supported?

This category relates to the characteristic that a tool should support the existing development process[3] and the characteristic saying that a tool should support multiple tiers.

[3]According to the sixth characteristic described in chapter 2.3.5, development process in this case is the same as build process.

3.4.3 Usability

The usability of the tools were decided to be investigated in the way that the author should use all of the tools on a few different sized/complex programs as well as on the HIGA code in order to get a feel for each of the tools respectively and then give a summary of his experiences. Even though that this does not result in a statement whether the usability of a tool is good or bad per se, it still gives some insight about the user experience. In addition to this general question about how a scan/audit is performed, three other questions were put together to serve as a base for the investigation:

- Does the size and complexity of a program influence the process of performing an audit?
- Does the tool have any restrictions on the amount of code a program can contain?
- How is the result presented to the auditor?
  - Are there any sorting alternatives?
  - Can the result be written to various file-types?

This category relates to the rewritten fourth characteristic. That is that a tool should be useful from a developer's point of view.

3.4.4 Performance

The tools ability to find bugs, the rate of false positives, rate of false negatives and the analysis time were determined to be of interest for evaluation.

This category does not relate to any of the characteristics that a tool should have but is still very interesting when making a comparison of the tools.

3.4.5 Types of Bugs Found

What sorts of bugs the tools are able to find was also considered to be of interest. To find out what kinds of bugs the tools could find and if they supported for the user to add custom checks were considered to be very important. The following questions were answered:

- What categories of bugs does the tool search for?
- Does the tool provide functionality of permitting the user to add custom checks?
- If such functionality exists, is there documentation to explain how it works?

Even if the first characteristic from chapter 2.3.5 was already determined to be satisfied this category has its roots in it. It also evaluates (parts of) the third characteristic, saying that a tool should be extensible.

3.5 Documentation

The first step in the evaluation of the tools was to review the documentation they provided.

The first thing that was examined was the actual documentation such as user manuals that came with the distribution of the tools and information on the tool's website. As a second step the web was searched for Internet forums and books about each tool.

3.5.1 Flawfinder

User manuals

The actual documentation about Flawfinder is a nine page user manual which comes with the installation of Flawfinder. It can also be downloaded from the homepage.

The manual begins with a description of Flawfinder bringing up questions like what it is, how it works, some basic annotations that can be made to the code and explanations to some expressions that are used in the output from a scan. This is followed by a brief tutorial on how to perform a scan. The next part explains all of the options that can be made when using Flawfinder. Furthermore it brings up how to integrate Flawfinder with Emacs[4]. The manual is concluded with some instructions on how to use Flawfinder in a secure way.

Neither the manual nor the homepage (*http://www.dwheeler.com/flawfinder/*) include a list of the different "dangerous" functions Flawfinder searches for and how these functions might compromise security. The output of a scan on the other hand explains why a certain function should not be used and in some cases suggests what to use instead.

All together the manual provides very good information and is easy to understand.

Internet forums

No Internet forums were found that discussed Flawfinder.

Books

There are no specific books about Flawfinder but it is referenced to in a lot of books about information technology, software security and hacking to mention some.

[4] A text editor commonly used by programmers since it has an extensive set of features often used by programmers.

3.5.2 Splint

User manuals

The Splint manual can be viewed either as a html page or as a pdf document. The manual is very large and thorough, consisting of 14 chapters and 5 appendices in a total of 121 pages. The first chapter brings up how to operate Splint, by explaining the basic commands, warnings produced by Splint, various flags that can be used, what annotations are and how they might be used. The remaining chapters deal with the various problems (bugs) Splint can detect. Each problem is given an individual chapter in which all of its sub problems are very well explained (in many cases with good examples). Furthermore which annotations that can be used for a specific problem, in order to make the scan even more accurate, are described.

The user manual does not contain any tutorial on how to use Splint but on Splint's web page (found at *http://www.splint.org/*) there exists a very good tutorial that covers many aspects on how to use LCLint (the predecessor of Splint). Since Splint is LCLint with some added functionality it serves as a very good foundation when beginning to learn how to use Splint.

As a whole the manual is very clear, explains all of the functionality very well and explains all of the problems searched for in an educational manner.

Internet forums

There is no Internet forum dedicated to Splint that the author of this thesis has found. However, there are two mailing lists that one can join: *Splint Announce*, which is used to announce new version of Splint, and *Splint Discuss* which is dedicated to informal discussions about Splint usage and development. The contents of the latter one can also be found at the *Splint Discuss Archives*[5] and it has some posts every month with response times at often just a few days.

Books

No books were found about Splint but it is referenced to in quite a lot of books concerning software security.

3.5.3 Fortify SCA

User manuals

Since only a demo version of Fortify SCA 4.0 could be obtained and since no actual documentation could be found on the website (*http://www.fortify.com*), the documentation that came with the demo version is what is being evaluated in this thesis.

Fortify SCA comes with two different user manuals. The first one is about the SCA and describes what it is, what different methods are being used when doing a scan, why they are made and finally how to use it. There are chapters explaining

[5] http://www.cs.virginia.edu/pipermail/splint-discuss/

how to use the SCA for Java, .NET, C/C++, PL/SQL, T-SQL and ColdFusion code, each making a step by step description of the procedure of performing a scan.

The other manual describes AuditWorkbench which is a GUI that comes with the Fortify SCA. This is used to organize the result of a scan and to make it easier to investigate it and prioritize different bugs. The AuditWorkbench manual begins with a getting stared guide that in a good way explains the various parts of the program and how to use them. The next chapter deals with how to perform an audit and how to generate various reports. This is followed by a part that explains more advanced functions and a troubleshooting and support chapter.

A list of what checks are being made to the code is not present in the user manuals. However, on the homepage a very thorough list is available that not only presents what checks are being made, but also how each problem searched for can affect security.

Both of the manuals are well-written and it is very easy to follow the instructions about how to install the tool, perform a scan with it and finally making an audit using AuditWorkbench.

Internet forums

No Internet forums about Fortify products were found.

Books

Both [16] and [13] comes with a demo version of Fortify SCA and chapters with tutorials and exercises on how to use it. The exercises concerns code written in both JAVA and C and are very good and educational and give the user a feel about how to use the tools and what they can be used for.

3.5.4 CodeSonar

User manuals

The user manual for CodeSonar has two major parts. The first is about setting up CodeSonar and the second is about how to use it.

The setting up part begins with a guide on how to install CodeSonar on Windows, Linux and Solaris. The guide is very straightforward and easy to follow. The next part of this chapter is a quick start guide on how to use CodeSonar on the command line or in Windows. The chapter is concluded with a good tutorial that describes how to perform a scan.

The second part that describes how to use CodeSonar brings up a lot of theory on how the tool actually works and what settings that can be made. It also holds a list of all the warning classes CodeSonar checks for, together with links to more specific descriptions of all the bugs that fall into each class. The descriptions of the bugs are taken from The Common Weakness Enumeration (CWE) which is an

initiative that focuses on creating a common set of software security vulnerabilities descriptions[6].

All together the user manual is very clear and in addition to explaining how to use CodeSonar it also gives very good descriptions, with a lot of code samples, of the various bugs searched for by the tool and how these may impact code quality and security of a program.

Internet forums

No Internet forums about CodeSonar were found.

Books

No books about CodeSonar were found.

	Flawfinder	**Splint**	**Fortify SCA**	**CodeSonar**
Clearness	Very clear and straightforward.	Very clear and straightforward.	Very clear and straightforward.	Very clear and straightforward.
Beginner's guide?	Some examples which is enough to get the user started.	Some basic examples on how to get started. Homepage also has a tutorial on how to use LCLint.	Tutorials describing how to perform a scan and inspect results using Audit Workbench.	Tutorials on all parts about how to perform an analysis.
List of bugs searched for?	No	Yes	Yes	Yes
Information about the bugs searched for?	Not in manual but in result of scan.	Yes. Very good.	Yes. Very good.	Yes. Very good.

Table 3.1. Evaluation of Documentation

[6] http://cwe.mitre.org/

3.6 Installation & Integration Procedure

This part of the evaluation focuses on the installation procedure of the different tools. The tools were installed on a computer running Ubuntu 8.04 with the goal to get the tool functional so that it could be used on the desktop by a developer in his/her daily work.

In the case with this thesis the tool is being integrated in the project rather late in the SDLC, making the question about the integration of the tool with the build environment very important.

3.6.1 Flawfinder

The installation of Flawfinder can be carried out in several ways. The easiest and most straightforward is to install the program with the help of a package manager like Red Hat Package Manager (RPM) or Debian's APT to install a binary of the program. This will install the latest version of Flawfinder and setup the program to be ready for use in an instant.

If one doesn't want the latest release or would like to compile the program oneself, tarballs of every version are also available for manual installation. The platforms supported are all kinds of UNIX-like systems but according to the developer of Flawfinder, porting it to Windows should not be very hard. The only programming language that can be scanned with Flawfinder is C.

Flawfinder does not provide support for letting it be integrated in an IDE and since it is a basic tool, which is best suited to be used on small code parts, the question concerning how to integrate the tool with an existing build environment was not investigated. However, if one want to scan a full program (consisting of multiple files) Flawfinder provides the functionality of accepting a folder that contains all of the source code files.

3.6.2 Splint

The latest release of Splint is at the time when writing this thesis not available as a binary and must thus be installed from source. There are some older versions available that can be installed thorough a package manager but if one want the latest release one has to compile the program oneself. The platforms supported are UNIX/Linux for the newest version but binaries of some older versions exist that can be used on Windows, OS/2, BSD and Solaris. Splint supports scanning of programs written in C.

When installing Splint 3.1.2 on a fresh Ubuntu install one minor problem was encountered: the installation process complained about not finding a function called *yywrap*. This was solved relatively fast by installing a program called flex[7] which contained the missing function. Other than that the installation went smoothly.

To integrate Splint with an existing build environment some manual editing of the Makefile has to be done. Splint does not come with a special function

[7]The fast lexical analyzer generator

which lets it be easily used in a large project as some of the more sophisticated commercial tools do. In order to run Splint on a whole project the Makefile has to be manually extended with calls to Splint with the necessary flags, include paths and paths to all of the source files. Another way of doing the scan is to make some sort of script that performs it.

Splint does not provide for being integrated in an IDE.

3.6.3 Fortify SCA

The installation of The Fortify Source Code Analyzer Suite 4.0.0 demo version was like any other installation of a commercial program. A nice installation GUI was presented and it was more or less click-and-go (command line installation in UNIX/Linux was also possible through an additional flag to the installation program). During the installation the most up to date coding rules were downloaded from the fortify web page. Fortify SCA can be installed and used on UNIX/Linux, Windows, Solaris, AIX and Mac OS X and supports the build tools Ant and Make as well as the Visual Studio (2003. 6.0 and .NET) IDEs. The programming languages supported are Java, .NET, C/C++, PL-SQL, T-SQL and ColdFusion.

The Fortify SCA comes with support to easily integrate it with the make build tool. There are two options in which this can be done. The first one is altering the Makefile to tell it to run the Fortify SCA every time a compiler is launched. The second method is to use the Fortify SCA built-in functionality (refereed to as the "Fortify Touchless Build Adapter" in the user manual) to automatically recognize the make environment. With this method every time make invokes a call to what Fortify SCA determines is a compiler, a scan is being performed. As this does not include any altering of the Makefile the integration becomes very straightforward.

3.6.4 CodeSonar

As in the case with Fortify SCA the installation of CodeSonar was also very simple. In Windows an installation guide was presented and the program was installed after a minute or two. In Unix all that had to be done was to untar a tarball. CodeSonar can also be installed on the Solaris and OS X platform and supports the build tools make, nmake as well as the Visual Studio IDE. The programming languages that can be scanned are C and C++.

The integration of CodeSonar with an existing build environment is as in the case with the Fortify SCA quite automatic. CodeSonar recognizes the make command as well as most of the most common compilers. If one is building the project with a Makefile, one simply calls CodeSonar with a call to make and the project name. CodeSonar then automatically performs a scan on the project and no editing of the Makefile is thus necessary.

If using an IDE, a wizard is available to make the integration just as easy. The wizard guides the user through a number of steps that among other things let CodeSonar "record" the build of the project as it is performed in the IDE. The result of this recording is then used by CodeSonar to perform a scan on.

	Flawfinder	Splint	Fortify SCA	CodeSonar
Installation procedure straightforward?	Yes. Install using packet manager or build from source.	Yes. Newest version: build from source. Some old versions available through packet manager system.	Yes. Installation guide with a "click-and-go" approach.	Yes. Unzip/untar and then ready for use
Integrating with existing build environment	No explicit support.	No explicit support.	Ant & Make	Make & Nmake
Supported platforms	UNIX-like systems.	Newest version: UNIX/Linux. Older versions: Windows, OS/2, BSD, Solaris	Linux, Windows, Solaris, AIX, Mac OS X	Linux, Windows, Solaris
Supported IDEs	None	None	Visual Studio	Visual Studio
Programming languages	C	C	Java, .NET, C/C++, PL-SQL, T-SQl & ColdFusion	C/C++

Table 3.2. Evaluation of Installation & Integration

3.7 Usability

This part of the evaluation will bring up the process of performing an audit as well as the author's view of the usability of the tools. The way this was done was by running each tool on three different programs of various sizes. The reason of this is twofold. The first thing to find out was to see if the size and complexity of a program influenced the process of performing an audit. A program consisting only of one file is most likely compiled directly on the command line whilst a larger program might use some sort of build tool in order to perform the compiling. Does this influence the way of performing an audit? The second thing concerned the result of the analysis. An analysis of a large program obviously generates much more data than an analysis of a small program and if the result is not presented

to the auditor in a good way it will most likely influence the result of the audit in a bad way.

The first program consisted of a single file with only 19 source lines of code (SLOC) written in C. The second program was Siproxd[8], a proxy/masquerading daemon for SIP. This was chosen since it is written mainly in C and uses some implementation of the SIP protocol which is the backbone of the HIGA. Siproxd was the mid size program, consisting of ~5000 SLOC in a total of 30 files. The last program that was analyzed was pure-ftpd, an ftp daemon written in C consisting of ~22000 SLOC in 115 files. The reason pure-ftpd was chosen was mainly because of the fact that it has been developed with security in mind[9].

3.7.1 Flawfinder

Single File Program

Performing a scan with Flawfinder is very straightforward. The only thing that has to be done is to call Flawfinder with the file to be analyzed as a parameter. Every possible bug found is presented with a reference to the file in which it is found, the line where it occurs and the degree of the severity of the bug. One also gets a reference to what category the bug belongs to, the name of the "dangerous" function as well as a short explanation of why it might be dangerous to use that specific function.

Figure 3.3 shows the output from the scan of the single file program. Flawfinder tells the user that a call to the printf function is being made on line 15 of the file *single_file.c*.

Siproxd 0.7.0

In contrast to the first case, where only one file was analyzed, Siproxd 0.7.0 consists of around 30 different files. Even so the process of performing a scan is still very simple. The only thing that has to be done is to give Flawfinder the folder with all the source files as an argument and it will begin scanning all of the files in it. Flawfinder does in other words not have any restrictions on the size of the program that is scanned. The result will be sorted on risk level with the highest ranked issues first. No other sorting alternatives are available.

One problem that arises when Flawfinder scans many files at once is that the result gets rather overwhelming and to just output it on the console is not recommended. Flawfinder comes with some functionality to format the output in various ways as for example as HTML or to display a single line of text for each hit. One can also set the lowest risk level from which Flawfinder should output the result or activate a function that reduces the number of false positives[10] (at the cost of that some issues might not get found) in order to some extent reduce the result.

[8] http://siproxd.sourceforge.net/

[9] http://www.pureftpd.org/project/pure-ftpd

[10] This works in the way that function names are ignored if they're not followed by "(".

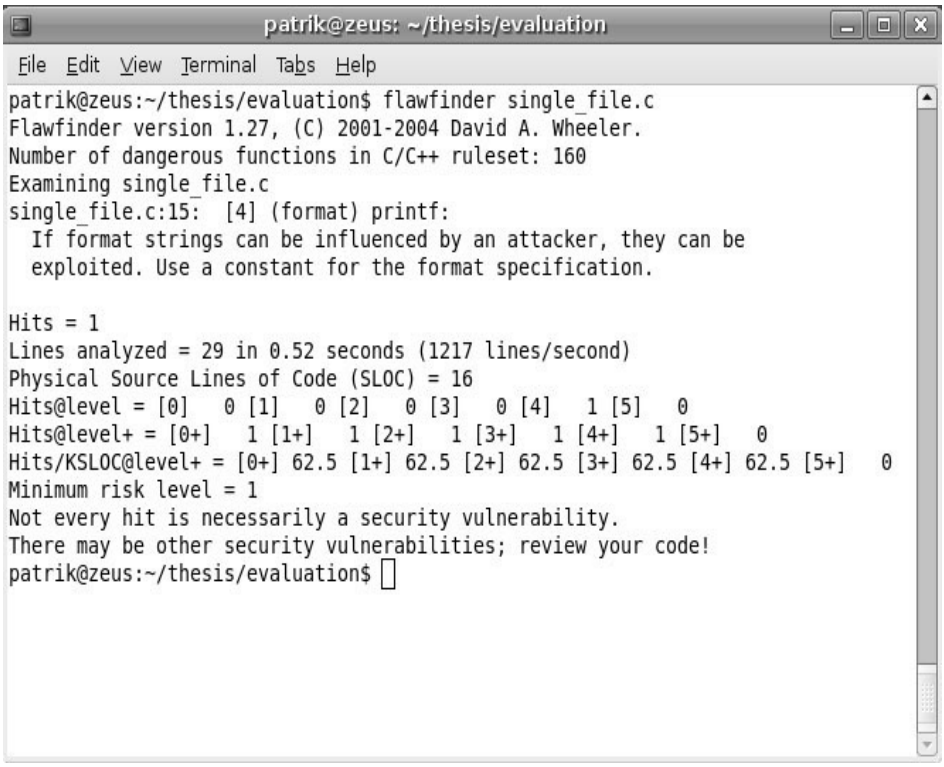

Figure 3.3. Scan with Flawfinder

Pure-ftpd 1.0.21

As in the case with Siproxd 0.7.0, Pure-ftpd 1.0.21 contains a lot of source files. However, the simplicity of performing a scan remains as just the folder with the source code has to be sent as an argument to Flawfinder.

A scan of Pure-ftpd 1.0.21 without any additional flags will give a result with 645 possible bugs and even if they are sorted at risk it is a very large job inspecting every single one of them. Furthermore no information is given on where the source of the problem arises in the code, only where the actual "dangerous" function call is being made. All this makes the result quite hard to sift through.

3.7.2 Splint

Single File Program

To perform a scan of a single file with Splint is as in the case with Flawfinder very easy. A call to Splint with the file that is to be scanned as a parameter is the simplest way. Figure 3.4 shows the output from a scan of the file *single_file.c*.

As seen in the figure Splint tells the user that a possible bug has been found

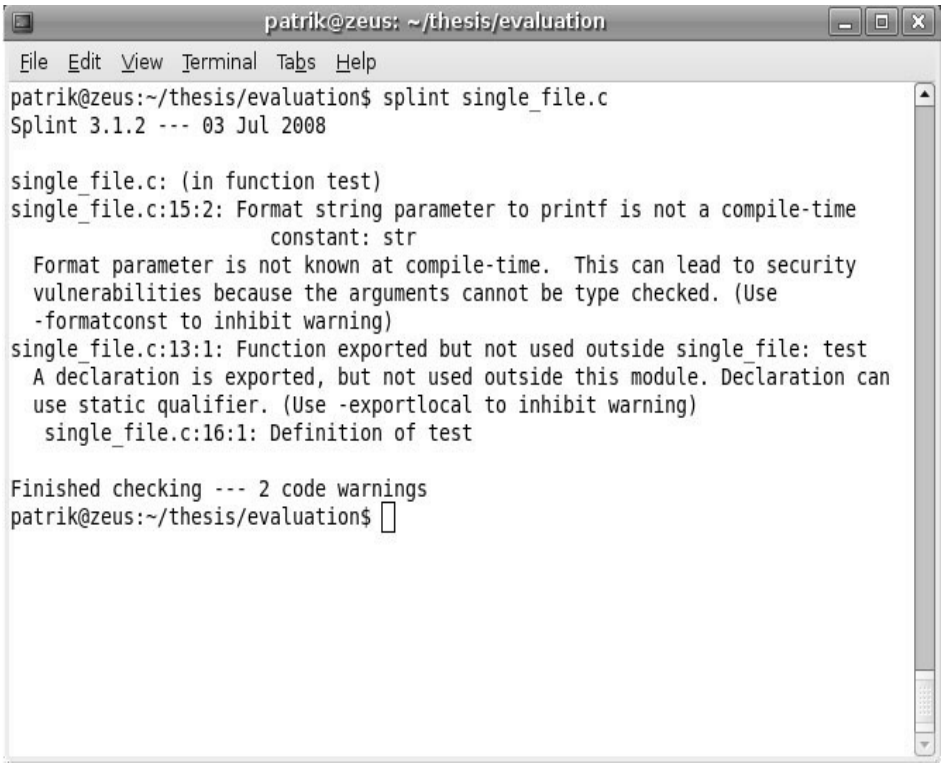

Figure 3.4. Scan with Splint

in the function test at row 15, column 2 in the file *single_file.c*. It also states the name of the variable causing the bug and why this might be a bug.

The last thing that is presented for each found bug is a flag that can be used in order to inhibit these kinds of warnings. If for example the user do not want to get any warnings about format string parameters he can give Splint the additional flag `-formatconst` to inhibit any warnings of that type.

Siproxd 0.7.0

Splint does not have any limitations on the size of a program that will be scanned but in order to perform an analysis on a whole project some additional work is required compared to the case when just scanning a single file. Splint does not support the option of just passing the folder containing the source code as input, as Flawfinder does, and thus another approach has to be taken.

The first, and probably the most simple way (in terms of implementation), is to make some sort of script that makes a call for Splint for each of the source code files and saves the output in some kind of result file. This makes the process of performing a scan of a rather complex program quite simple but with the drawback

that a lot of work has to be done before the scan can begin.

Another way is to extend the Makefile to also make calls to Splint and in that way combine the build process with the scan. As in the case with the "script method" this also involves some preparatory work and furthermore requires knowledge about how the make build tool works. However, in the long run this method might still be better since the Makefile in an easy way can be extended to create various special cases on which files to scan according to different pre-requisites (such as for example only scanning one specific module in a large project).

When Splint was used to analyze Siproxd the biggest problem to get it to succeed and produce a good result was the use of additional flags that can be given to Splint. Splint comes with a lot of flags that can be used to tune the scan into giving a more precise result. Furthermore some flags had to be used just to make Splint actually succeed with a scan. The first scan that was performed on Siproxd only had the smallest amounts of flags which more or less only made it possible for Splint to process all of the files (almost[11]). This resulted in an output of over 1000 warnings.

Another thing that was noticed when running a scan with Splint on a large project was that the output from a scan (with no extra flags given to Splint and no annotations made to the code) was rather heavy and when compared to the code many of them were unnecessary. In order to reduce the rate of false positives extra flags have to be given to Splint and insertion of annotations into the code have to be made.

The different output formats supported by Splint include a text file, a csv (comma separated value) file and a "semi-html" (the references to the files scanned are written in html) file format. No sorting of the output is possible.

Pure-ftpd 1.0.21

The scan of Pure-ftpd 1.0.21 was very much like the scan of Siproxd 0.7.0 in that a lot of preparatory work had to be done in order to get the scan to succeed and to generate a good output (the first scan generated over 1600 warnings).

As a whole a scan with Splint of a large project where no annotations to the code has been made is a hard job. A lot of preparatory work has to be carried out before the scan succeeds in running and produces a good output. But if on the other hand Splint is being used throughout the whole SDLC (which of course is the right way to go when using a static analysis tool) and annotations have been made to the code during development, a lot of time can be saved. This of course demands that the developers have to learn all about the annotations that are possible to make and how they are used.

3.7.3 Fortify SCA

Single File Program

The call to the Fortify SCA as well as the result of the scan of *single_file.c* can be seen in figure 3.5.

[11] Splint encountered an internal bug and crashed in one of the files

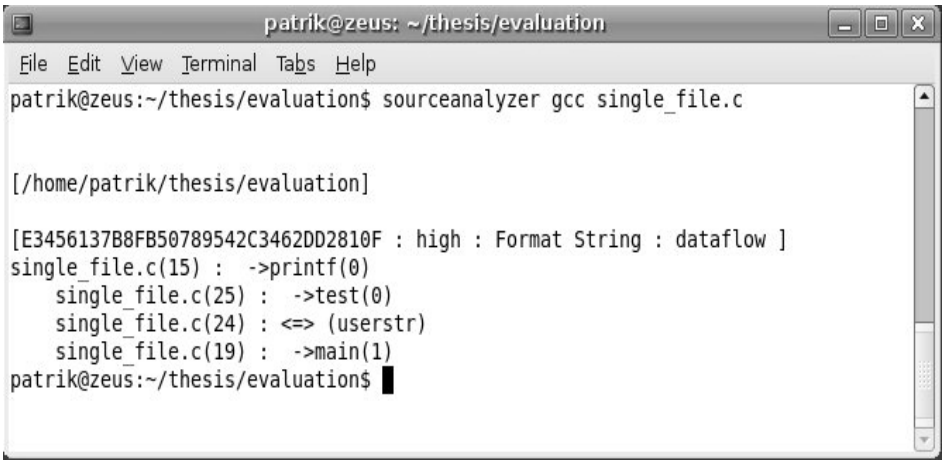

Figure 3.5. Scan with the Fortify SCA with result in console

As seen in the figure the call to sourceanalyzer is followed by the compiler that is to be used to compile the file. The result of the scan of this small file is quite simple and can be presented directly in the console. If one is doing a scan of a more complex program which generates a lot of output one can tell the tool to write the result in a "fpr format" which is the format used by AuditWorkbench. Other possible output formats are text files and FVDL (Fortify's Vulnerability Description Language which is an xml-dialect).

The result of the scan in figure 3.5 begins with a "header" consisting of four different fields. The long line of letters and digits is a globally unique identifier that all of the found bugs will get. This identifier is computed in a way that makes it stay the same even if minor changes are made to the code, and it can thus be is used in order to track the same issue across multiple analyses and code versions. The next field tells the severity of the bug and the field after that tells what vulnerability category the found bug belongs to. The last field tells which analyzer that found the issue. In this case it was the dataflow analyzer.

The next thing that is presented is the dataflow that shows the path leading to the issue. As seen in the figure the source of the problem begins at row 19 and at row 15 the function call that causes the problem is located.

Siproxd 0.7.0

When making an analysis of a complete program, and not only of a single file, the easiest way is to use the Fortify SCA's built-in functionality of automatically make use of the Makefile of the program that is to be scanned. The first thing to do is to run the SCA with a flag, `-b buildid`, which assigns an id to this specific build of the program. Furthermore the make command has to be given as a parameter. The Fortify SCA will build the complete program and at the same time translate the source files into an intermediate format that will be used as input for the actual

scan.

The next step is to call the Fortify SCA with the scan flag, the build id that was assigned in the previous step and choose an output format.

To work with the Fortify SCA is very straightforward and it is not very hard to understand how the different flags work. However, the status bar indicating the process of the scan only gives the percentage of how much that has been done. Unfortunately this means that even if the status bar tells the user that 90% are done, the scan might not have come halfway measured in time. This might be a problem since the time to conduct a scan might be in the range from a couple of seconds for a small project to hours in a larger project and if the user does not get an estimate on when the scan is finished this might impact the possibilities for the user to organize his/her work.

Fortify SCA does not have any restrictions on the size of the analyzed program.

When the scan is finished the next step is to analyze the result by using AuditWorkbench. The first thing that is presented when beginning a new audit is a summary of the project. The summary gives the user an idea of the magnitude of the scan in that it tells how many issues that have been found and which files that contains the most of them. The user is also given a choice to continue the audit by using the AuditGuide which is a guide that helps the user to choose what kind of warnings he/she wants to be notified about.

Another way of choosing what to display of the result is by customizing the Rulepack security level as seen in figure 3.6. AuditWorkbench comes with three different security levels: *Broad, Medium* and *Targeted*. If the audit should be used to discover a broad set of possible security issues the *Broad* choice is to be preferred but if one only wants to be notified about issues that are proven to be high priority issues the *Targeted* choice is better.

The next step is to inspect all of the issues. Figure 3.7 shows the standard view of the AuditWorkbench. At the top left corner a panel presenting all of the issues is present. The issues are divided in three different categories: *Hot*, *Warning* and *Info* which all are color coded with red, orange and yellow respectively. The hot issues are the one that should be given immediate attention and when clicking on the "Hot" button all of those bugs are presented grouped by vulnerability category. Grouping/sorting can also be made according to analyzer, audit analysis, filename, sink, source, taint flag and new issues.

When clicking on an issue the source file will be presented on the right side of the workspace and the row with the issue will be highlighted. At the bottom left the analysis trace of the issue is presented so that one can easily follow how the issue arises.

At the bottom of the screen a summary of the current issue is presented. The auditor can make choices such as the status of the issue, what impact it has, what kind of issue it is and to what category (Hot, Medium or Info) the issue should belong to. Furthermore the auditor can obtain details about the specific issue such as an explanation of it, examples of how it might characterize itself and some recommendations on how to avoid it.

As a whole AuditWorkbench is a very good support for the auditor in that it comes with many features that might help to get the analyze of the result of a scan

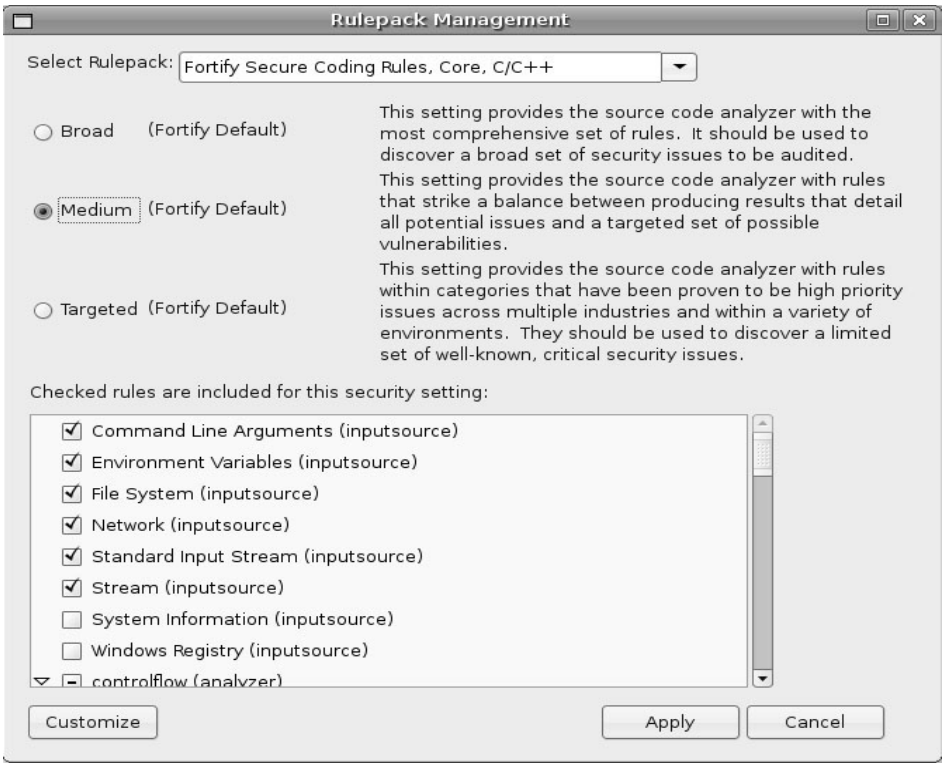

Figure 3.6. Customizing the Rulepack security level in AuditWorkbench

to become more smoother than if only a text file with the result was presented to the user.

Pure-ftpd 1.0.21

Pure-ftpd 1.0.21 is an even more complex program than Siproxd 0.7.0 but the procedure of performing a scan is all the same. As long as the program will compile on the system using the Makefile, Fortify SCA will have no problem in interacting with the make tool and perform a scan of the program. The audit of the result is then performed using AuditWorkbench as described in the previous part.

3.7.4 CodeSonar

Single File Program

To perform a scan with CodeSonar on a single file is just as straightforward as with all of the other tools evaluated in this thesis. A call to the program with a few additional flags is all that is needed in order to start the scan and later on

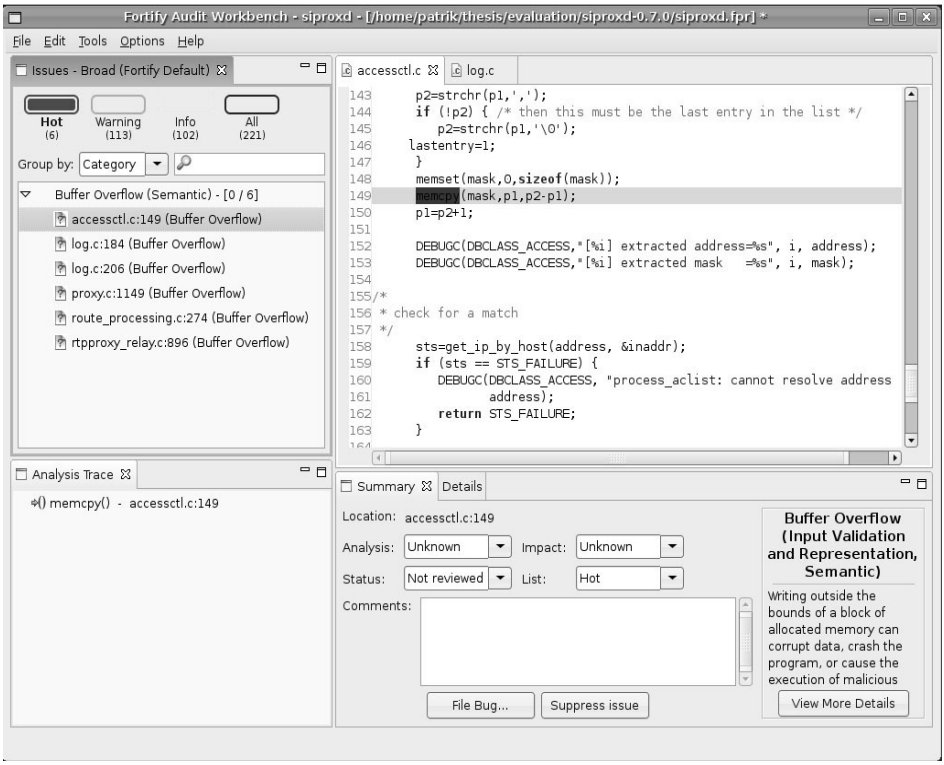

Figure 3.7. Standard view of AuditWorkbench

the result can be inspected using a web browser. Figure 3.8 shows the scan of *single_file.c*.

The flag `hook-html` is used to tell CodeSonar to build the program, run the analysis and at last output the analysis results to a series of HTML-files (which is the only output format supported by CodeSonar). The next argument is the project name that the user would like the project to have followed by a command telling CodeSonar where to send the result (see below). As a last argument a call to the compiler that is to be used for the actual build as well as the file to be scanned is sent.

When the scan is completed the result is sent to the *CodeSonar-hub* which is a small web server that comes with CodeSonar. The CodeSonar-hub can be configured to act as a service on any computer, receiving analysis results from many different users and scans. If this is the case yet another flag has to be sent to CodeSonar, telling it the IP address as well as the port number of where the CodeSonar-hub is running. If this flag is not passed CodeSonar automatically starts a hub on the local machine on which the scan is being performed.

The approach of presenting the result of a scan online through a dedicated web server opens up major opportunities for collaboration between many auditors.

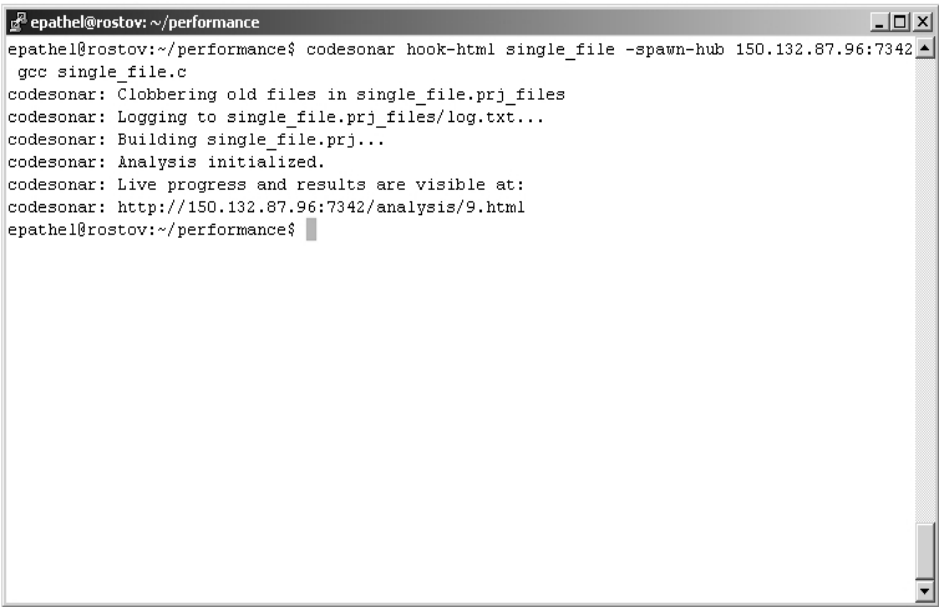

Figure 3.8. Scan of single file with CodeSonar

To inspect the result of a scan the user simply visits the hub homepage and chooses the current project. Figure 3.9 depicts the front page of the hub whose main purpose is to present all of the projects that are active. The state column contains a progress bar telling the user how far the analyze has come and an estimation on how much time is left. Unfortunately the progress bar does not move unless the whole page is being reloaded which can be a bit frustrating.

When a project is chosen all of the issues found for the specific project is presented as warnings to the user. As all of the other tools in this survey CodeSonar makes no exception in that it also presents some sort of rank of the severity of the found issues. With this the user quickly can get a feel for how many "high-level" risks the code contains. However, the way that the risk is calculated is not described anywhere which in turn makes one wonder why the risk is presented in numbers in the first place. Why not use different categories? Furthermore, since no explanation is given to the numbers one has no idea what counts as a very high risk issue, a high risk issue, a medium risk and so on. Figure 3.10 shows the view presenting the result of the scan of *single_file.c*. Most of it should be self-explanatory but it should be mentioned that all of the warnings can be sorted according to any of the columns.

The next step in the inspection of the result would be to examine the issues. Figure 3.11 shows the format string issues found in *single_file.c* with the warning location highlighted. As seen in the legend on the right hand side CodeSonar uses different color codes to explain how a certain issue arises. On the left hand side in

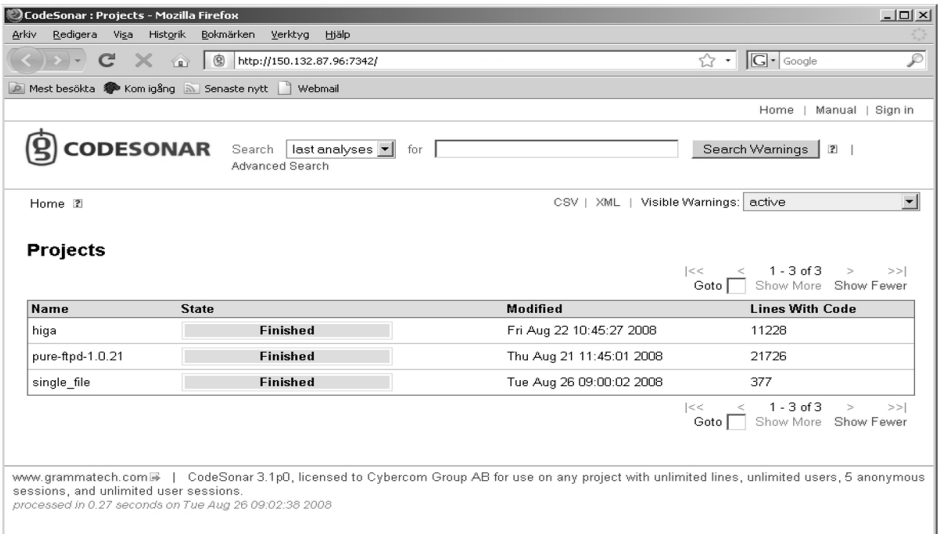

Figure 3.9. CodeSonar-hub front page

the *problem column* various hints can be given to the user about what is happens on a row that contributes to the warning (see the part about using CodeSonar to scan pure-ftpd for a more detailed explanation on how to follow the path leading to an issue).

If the auditor is not sure why/how the issue might give rise to an error or a security breach the links presented next to *Categories* leads to good and educational descriptions of the issue.

Siproxd 0.7.0

CodeSonar does not have any restrictions on the code size of a program, and when a scan of a more complex program than only a single file is to be performed not much is different from the previous case. If a Makefile is used, all that has to be done is to perform a `make clean` command (since CodeSonar "watches" a build of the program subject to a scan as a way of learning about different dependencies etc. it is of great importance that a complete build is performed) and after that call CodeSonar in the same way as in the case with *single_file.c* but with a call to `make` instead of a compiler followed by a file.

The remaining part of the work, such as inspecting the issues, is all the same as described in the previous part.

Pure-ftpd 1.0.21

Pureftpd 1.0.21 is the most complex program used in this part of the thesis but it does not influence the procedure of performing a scan with CodeSonar at all

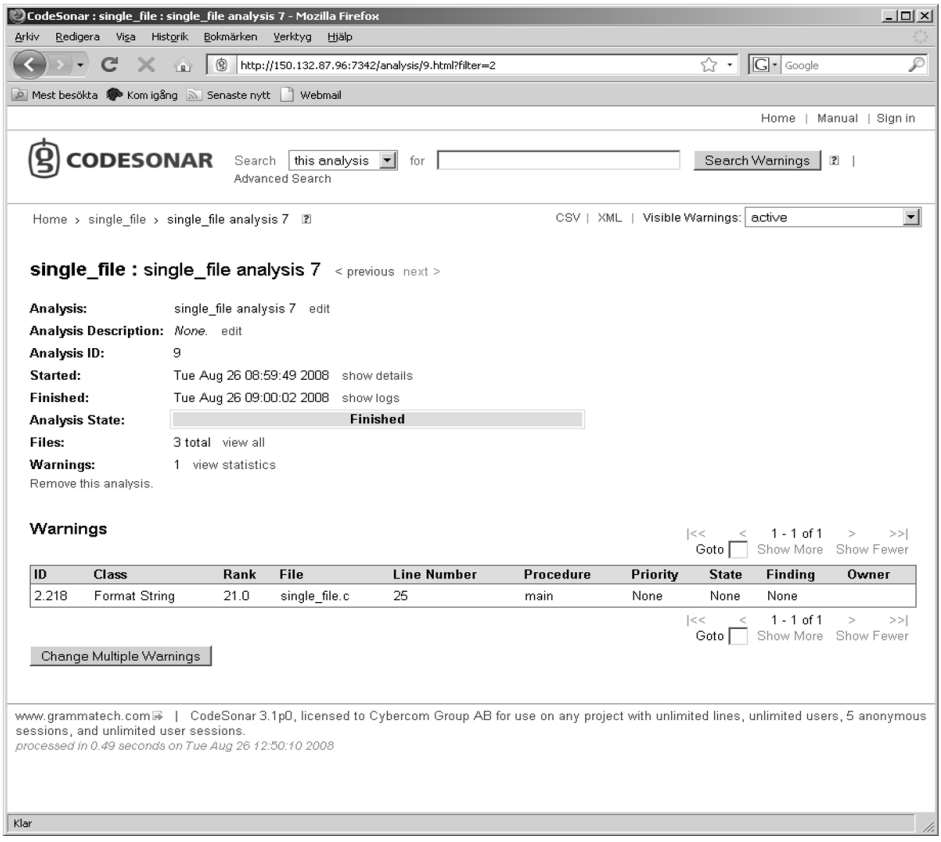

Figure 3.10. Analysis result presented in CodeSonar-hub

compared to the case with Siproxd. As long as the program builds correctly with the help of the Makefile, CodeSonar will have no problem in performing an analysis.

One of the issues found by CodeSonar when scanning Pureftpd 1.0.21 is illustrated in figure 3.12. This serves as good example on how CodeSonar presents the path that leads to the issue and what different parts of the code that contributes to it. In this case CodeSonar warns that opening of a file on row 49 may not be closed under some circumstances which will give rise to a memory leak. The actual command that give rise to the issue is highlighted in yellow (the command `open` in this case) and a comment is inserted by CodeSonar on the current row, describing what happens. In this case also `read` is highlighted, but in green, telling the user that this command contributes to the issue. All of the read colored text is code that is on the execution path leading to the issue. Here CodeSonar tells the user that if `r <= (ssize_t) 0` is true, the function will return a value and the file will not be closed, leading to a leak.

In order to follow data flow CodeSonar provides the functionality of presenting

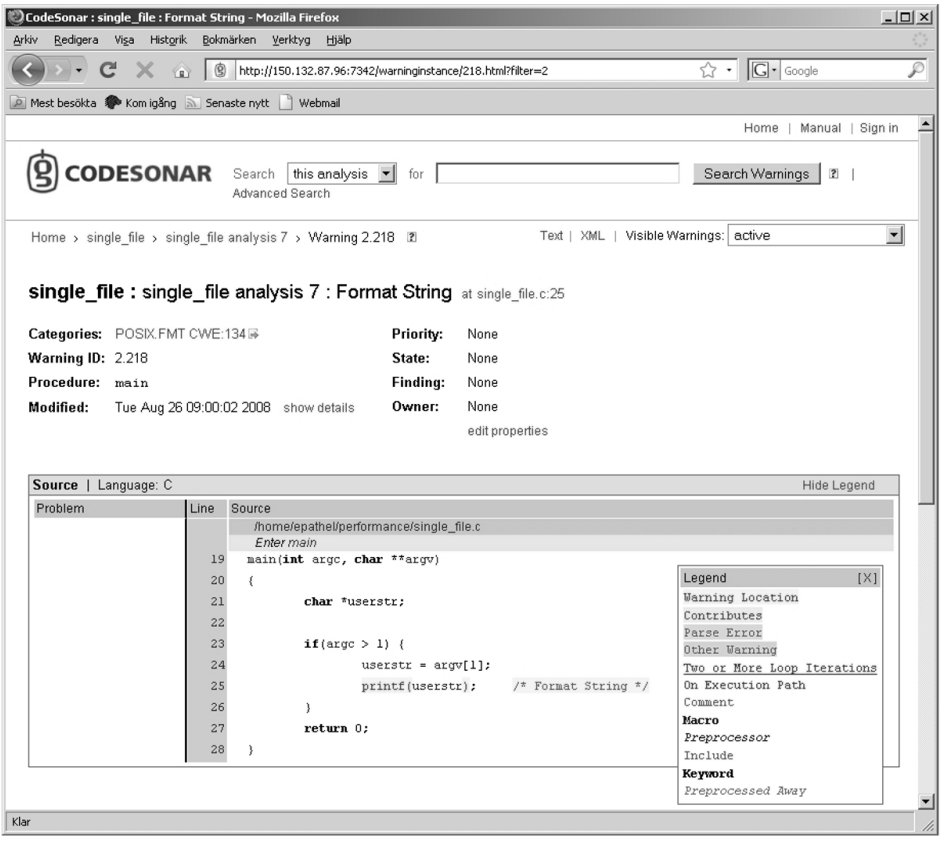

Figure 3.11. Format string issue found by CodeSonar

all of the variables and function calls as links to the file in which they are declared. Furthermore if an issue has its root in a different function CodeSonar opens up that function inside the current source and in that way builds a “tree” of nested functions that eventually leads to the root of the problem.

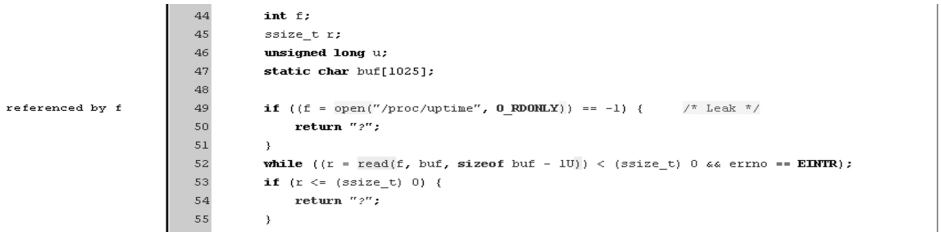

Figure 3.12. Analysis trace in CodeSonar

	Flawfinder	Splint	Fortify SCA	CodeSonar
Impact on audit process from code size and complexity	None. Pass folder containing source code and Flawfinder scans the appropriate files.	Big. Manually edit the Makefile or write script to perform scan. All paths to header files etc. has to be passed as input, many flags has to be set in order for it to work.	None. Pass the Makefile as input.	None. Pass the Makefile as input.
Restriction on amount of code?	No	No	No	No
Results presentation	Command line, text file, html file. Sorted on risk level.	Command line, text file, csv file, "semi-html" file. No sorting.	Command line, text file, fpr (for Audit Workbench), FVDL (an xml dialect). Many sorting alternatives.	Presented as a web page in a stand-alone web server provided by CodeSonar. Many sorting alternatives

Table 3.3. Evaluation of Usability

3.8 Performance

This part of the evaluation focuses on reviewing how good the tools are at finding bugs, if they generate a lot of false positives/false negatives and how fast they are.

The first test concerned the rate of false positives/false negatives. Two different test suites were obtained from the NIST[12] SAMATE[13] Reference Dataset Project[14] (SRD). A project who's purpose is to provide users with sets of known security vulnerabilities and in that way allow them to evaluate for instance tools for code analysis.

[12]National Institute of Standards and Technology
[13]Software Assurance Metrics and Tool Evaluation
[14]http://samate.nist.gov/SRD/

The first test suite (number 45 on SRD) contained 77 different code snippets, each containing some sort of bug that could affect security. This was used to find out the rate of false negatives for each tool.

The second test suite on the other hand (number 46 on SRD) contained 74 code snippets that all were coded in a "correct way" and thus could be used to find out the number of false positives generated by each tool.

Out of these two test suites, two specific ones were put together for each tool respectively (one testing for false positives and one testing for false negatives). In this way the test suites for each tool only contained the types of bugs that the tools claimed they could find, and the result of this test thereby gives a rate of how good each tool is at finding the various sorts of bugs they claim they can find. All of the test suites used as well as the results of the tests are found in appendix C.

As a complement to this test yet another test was performed. This one was used to find out the tools ability of finding bugs.

The third test concerned how much time the tools needed for a scan. To find out this Pure-ftpd-1.0.21 (~22000 SLOC) as well as the code base for the HIGA (~12000 SLOC) was used as test programs.

All of the analyses in the "time test" were performed three times at different times of the day[15] on a computer with double Intel Xeon processors at 3.0GHz, 16 GB of Ram and running Ubuntu 7.10. The times presented in this chapter are the mean-values of the three different scans.

3.8.1 Flawfinder

The test suite put together for Flawfinder when testing it for false negative consisted of 36 test cases. Out of these 2 were missed by the tool, giving a false negative rate of 5.56%. As seen in table C.1 in appendix C, the two test cases that were missed were number 1612 and 1751.

The first one tests for a heap overflow by using the function `strncpy` to copy 80 characters into a buffer, that can only hold 40 characters. Since the function `strncpy` is used Flawfinder does not warn since this is not listed as a dangerous function, such as `strcpy`. This kind of type of error perfectly describes the weakness of only searching after function names that might be dangerous to use.

The other test case that were missed concerned a simple buffer overflow. In this case an array with a size of 10 was overflowed by simply giving an out of bound index to the array and try to assign a value to it. Since no "dangerous" function was used to perform the overflow Flawfinder is unable to find the error.

The suite used when determining the false positive rate consisted of 37 test cases of which Flawfinder reported 30 to contain a possible bug. This gives a false positive rate of 81.08%.

[15] The reason of this is because the computer the analyses were performed on was a distributed build server. By running the analyses on different times of the day, and then calculate the mean time of the runtimes of the scans, the effect of the workload on the server, affecting the runtimes, is reduced.

The very high rate of false positives is a direct effect from that Flawfinder only performs a lexical analysis and consequently not actually knows anything about what "happens" in the code. A more detailed result of the tests can be seen in table C.2 in Appendix C.

The time Flawfinder needed in order to finish the scan of Pure-ftpd was just about 1 to 2 seconds and the scan of the HIGA code took about 1 second.

3.8.2 Splint

None of the test cases analyzed with Splint did contain any sort of annotations. This was simply due to the fact that annotating all of them would have taken to much time. Furthermore Splint was run with no flags at all.

When evaluating the false negative rate of Splint a test suite with 44 test cases was used. Splint missed 22 of these giving a false negative rate of 50.00%.

The false positive test suite consisted of 42 test cases. Splint reported 3 of them to contain a possible bug leading to a false positive rate of 7.14%.

The exact test results are found in table C.3 and C.4 in appendix C.

The reason that Splint performed so badly in finding many of the issues is probably because of the reason that no annotations were done to the code. In order for Splint to perform a really good scan annotations should have been made.

Splint performed a scan of Pure-ftpd in about 2 to 3 seconds and of HIGA in about 1 second.

3.8.3 Fortify SCA

The false negative rate of Fortify SCA was tested with a suite containing 66 different test cases. 10 of them were missed giving a false negative rate of 15.15%.

Two interesting things in this test is that 3 of the 10 false negatives were tests for TOCTOU[16] race conditions. Fortify SCA did in other words not find any of the issues, concerning race conditions, tested for. What is even more interesting is that in all of the tests, concerning this kind of issue, which were performed to find out the false positive rate, SCA reported a warning. The other interesting thing is that Fortify SCA did not find any of the bugs in the four test cases belonging to the "Hard Coded Passwords" group.

The test suite used for finding out the rate of false positives consisted of 65 test cases. Fortify SCA reported that 27 of them contained a bug which gives a false positive rate of 41.54%.

The table C.6 shows that Fortify SCA especially has problems with command injection, resource injection and SQL injection issues. All of the tests performed on these issues were reported to contain an error when they actually were coded in a correct way. Other than that it is hard to find any other category of bugs that Fortify SCA has a great deal of trouble with telling if the bug indeed is a true positive or a false positive.

A scan of Pure-ftpd with Fortify SCA took about 80 seconds and a scan of the HIGA about 65 seconds.

[16] Time Of Check Time Of Use

3.8.4 CodeSonar

CodeSonar was tested for false negatives with a test suite with 50 test cases. 19 of them were missed, giving a false negative rate of 38.00%.

The false positive rate was tested with 48 test cases of which 3 were reported to contain bugs, resulting in a false positive rate of 6.00%.

C.7 and C.8 in appendix C show what kinds of types of bugs that CodeSonar missed as well as what types of bugs it is having problem with finding out if it is a true one. As seen one cannot determine any specific category of issues that CodeSonar is having problems with. Instead it seems to find one type of issue in one case and then miss the same kind of issue in another test case.

The rate of false positives on the other hand is very good.

CodeSonar performed the scan of Pure-ftpd in a about 4 minutes and of the HIGA in about 11 minutes.

	Flawfinder	**Splint**	**Fortify SCA**	**CodeSonar**
Bugs found	44.74%	28.95%	73.68%	39.47%
False Positive Rate	81.08%	7.14%	41.54%	6.00%
False Negative Rate	5.56%	50.00%	15.15%	38.00%
Time of analysis of Pure-ftpd 1.0.21 & HIGA	1-2 & 1 seconds	2-3 & 20 seconds	80 & 65 seconds	4 & 11 minutes

Table 3.4. Evaluation of Performance

3.8.5 Comparative Test

As a complement to the previous test suites, one more test suite was put together. This one contained 76 different test cases and was used to compare the tools ability to find various bugs. The test suite can be found in table C.9 in appendix C.

As seen in the table no respect was given to whether a tool should be able to find a particular type of bug or not. In this way a more fair comparison between the tools could be made and the result can also be used for comparison with future tests.

The tool that succeeded best in this test was Fortify SCA which was able to find 73.68% of the bugs. This is a far better result than any of the other tools managed to produce: Flawfinder, CodeSonar and Splint found 44.74%, 39.47% and 28.95% respectively.

From the table one can identify four categories in which all of the tools failed in finding the bugs: the XSS (Cross-site scripting) category. the Hard-Coded Password category, the Unintentional pointer scaling category and the Resource

locking problems category. In all other categories at least one of the tools managed to find the bugs.

3.9 Types of Bugs Found

This part describes more in detail what different sorts of vulnerabilities the different tools find. Since all of the tools find very many different bugs this part will only bring up the different categories that the vendors of the tools have chosen to divide them into.

3.9.1 Flawfinder

Flawfinder focuses on finding "dangerous" functions from a security point of view. The most recent version contains a built-in database containing 160 C/C++ functions, categorized in nine categories, with well-known problems, all of which could be used to breach security in some way if not used correctly. The categories of the different rules include:

- **Buffer**, deals with buffer overflows etc.
- **Format**, deals with format strings attacks
- **Race**, deals with race conditions
- **Misc**, miscellaneous functions that could be used to breach security
- **Access**, deals with access to objects etc.
- **Shell**, deals with execution of new programs and processes
- **Integer**, deals with the problem of signed and unsigned integers
- **Random**, deals with "bad" random number generators
- **Crypt**, deals with weak cryptographic functions

Since Flawfinder is open source it is also possible for the user to add his/her own checks. The way this is performed is by expanding the dictionary containing the names of the "dangerous" functions that Flawfinder searches after. Flawfinder is written in python, and for a person with basic programming skills the procedure of adding yet another check should not be very hard. An entry in the dictionary (also referred to as "the ruleset") used by Flawfinder looks like `''function_name'' : (hook, level, warning, suggestion, category, {other})`. The entry for the function `strcpy` will serve as an example:

```
"strcpy" :
  (c_buffer, 4,
  "Does not check for buffer overflows when copying to destination",
  "Consider using strncpy or strlcpy (warning, strncpy is easily
```

```
misused)",
"buffer", "", {}
)
```

It should be mentioned that the manual does not include any information about how to include a custom check.

3.9.2 Splint

As mentioned earlier Splint does not solely focus on finding bugs that affects the security of the program but on general code quality as well. The following list is taken from [10] and describes the different categories of vulnerabilities that Splint searches for.

- Dereferencing a possibly null pointer
- Using possibly undefined storage or returning storage that is not properly defined
- Type mismatches, with greater precision and flexibility than provided by C compilers
- Violations of information hiding
- Memory management errors including uses of dangling references and memory leaks
- Dangerous aliasing
- Modifications and global variable uses that are inconsistent with specified interfaces
- Problematic control flow such as likely infinite loops, fall through cases or incomplete switches and suspicious statements
- Buffer overflow vulnerabilities
- Dangerous macro implementations or invocations
- Violations of customized naming conventions

In addition to these categories it is also possible for the user to add his/her own checks and annotations in order to enhance the scan to be more precise and to better fit for a specific project. The manual contains a specific chapter that brings up how this is done and it also contains an example to explain in detail how it works.

The user manual states that many useful checks can be described as constraints on *attributes*[17] associated with program objects. To add a new check in Splint

[17] An attribute can for example be associated with a specific program object or a specific type of pointer etc.

implies to creating a new attribute, defining which objects or data types that can have the attribute, and as a last step defining possible values and transfer rules for that specific attribute. In order to do so, Splint provides a general language that is used to produce a so called "metastate file" containing the attribute and its rules.

Since a special kind of programming language is used, some programming skills would probably be beneficial if trying to create a custom rule.

3.9.3 Fortify SCA

The Fortify SCA is a tool that focuses solely on finding security vulnerabilities. Fortify's research group has divided the vulnerabilities scanned for into seven different categories, each consisting of several sub-categories. This is also the same taxonomy that is being used in chapter two in this thesis and for a description of each category the reader is referred to page eight. The seven categories are seen below.

- Input Validation and Representation
- API Abuse
- Security Features
- Time and State
- Error Handling
- Code Quality
- Encapsulation

Fortify SCA also provides functionality of letting the users specifying their own rules. However, the user manual that was included with the demo version of Fortify SCA 4.0.0 (which was tested in this thesis) did not contain any information on the subject but both [13] and [16] has a chapter that describes how this is done (both these book contains a demo version of Fortify SCA).

Fortify SCA makes use of rulepacks (see chapter 3.3.3), and to create a new rule, a new rulepack has to be created. A rulepack is an xml file containing information about what kind of rule it contains and the actual rule (for example a semantic rule or a dataflow rule etc.). The rule itself is constructed by using various xml tags, specific for the Fortify SCA, and is quite an advanced task to produce. Due to the lack of information about this topic in the user manuals, and only a tutorial in [13] and [16] were available to the author about this topic, no further discussion is being made.

3.9.4 CodeSonar

CodeSonar provides functionality of finding bugs that affect security and bugs that affect code quality in general. The categorization of vulnerabilities searched for is quite wide-ranged and can be viewed in Appendix D.

CodeSonar supports for implementation of custom checks and offers two different alternatives in which this can be achieved. The first method is by inserting the code for the check directly into the program files that is to be scanned. The second method is to write a special code wrapper in a separate file, which is later on linked into the CodeSonar project. If the latter method is used, the language in that the wrapper has to be written is C.

To add a new check is listed as an advanced topic in the manual but the chapter describing how it is done is very clear and also contains a good tutorial on how to do it.

	Flawfinder	**Splint**	**Fortify SCA**	**CodeSonar**
Types of bugs	Security	Code quality and security.	Security	Code quality and security.
Possibility to create new rules?	Yes	Yes	Yes	Yes
Documentation about how to create new rules?	No	Yes	No	Yes

Table 3.5. Evaluation of Types of Bugs Found

Chapter 4

Test of HIGA Source Code

This chapter discusses the actual testing that was performed on the HIGA product developed at Ericsson AB in Linköping. The chapter includes a description of how the tests were performed, a test specification that was used to organize the tests and finally the results of the tests with a corresponding summary.

4.1 Test plan

The HIGA consists of many different modules of which two were tested in this project. The first one will be referred to as Module A (~12000 LOC) and the second as Module B (~23000 LOC). In both these modules good code quality that implies robustness and a secure execution is of great importance.

The first thing that was done before the tests were started was to put together a test specification consisting of different things to test the code for. This list was put together in collaboration with personnel on the HIGA project and with *The Fortify Taxonomy of coding errors that affects security*[1] as a base.

The tools were then used to test for all of the specific vulnerabilities in the test specification and in those cases in which more than one tool warned about the same issue, an in-depth analysis was done by the author in order to find out if it was indeed a correct warning or a false positive. This was also done for those issues that the tools ranked as the highest risks.

Both of the modules tested had a Makefile which could be used to compile it. In the case with the commercial tools where support was given to make use of a Makefile this was the way the tests were performed. In the case with Flawfinder and Splint a script was written for each tool and module respectively to perform the scan. The scripts called the tool in question and sent the files that were specified in the Makefile as input. In this way all of the tools scanned the same files.

[1] http://www.fortify.com/vulncat/en/vulncat/index.html (also represented in Appendix A)

4.2 Test specification

The following weaknesses was checked for:

- API Abuse
  - Missing check against NULL[2]
  - Unchecked Return Value
- Code Quality
  - Dead Code
  - Double Free
  - Memory Leak
  - Null Dereference
  - Divide by Zero
  - Uninitialized Variable
  - Unreleased Resource
  - Use After Free
- Input Validation and Representation
  - Buffer Overflow
  - Buffer Underrun
  - Command Injection
  - Format String
  - Illegal Pointer Value
  - Integer Overflow
  - Log Forging
  - String Termination Error
- Security Features
  - Insecure Randomness
- Time and State
  - File System Race Condition: TOCTOU
  - Deadlocks

[2]Fortify SCA is the only tool that has an explicit check for this type of error. CodeSonar performs a check for cases where a NULL pointer is freed which implicitly might indicate a missing check for NULL. In the tests the result from this check is what is reported in the "Missing check against NULL" for CodeSonar. This deliberation was made as this is a very important category. As a consequence all of the issues found by Fortify SCA in this category were thoroughly inspected.

4.3 Practical issues

When trying to run Splint on the HIGA code it repeatedly failed, complaining about parse errors. This turned out to be due to the fact that Splint is written in C89 standard and thus only supports scanning of C89 compliant code. One of the drawbacks with the C89 standard is that new variables can not be declared at other places than at the beginning of a function. Since the HIGA code was not written with this standard Splint failed to analyzing it. However, after some time searching the issue on the Internet a patch was found that solved this problem. The patch[3] changed part of a file called `cgrammar.y` which is part of Splint source code and contains grammar declarations used by Splint. The problem with this patch was that it was not (yet) approved to be part of Splint. Even so it was used as the other alternative would have been to totally exclude Splint from this part of the thesis which seemed as a rather harsh solution. After Splint had been patched all of the HIGA code went through right away.

4.4 Test result of Module A

As seen in table 4.1 the two issues that by far received the most warnings were "Null Dereference" and "Buffer Overflow" (seen from the point of view that more than one tool had to report the same issue). However, a vast majority of them turned out to be false positives only causing a rather tedious inspection of the code[4].The reason that so many of them were false positives was probably due to the fact that many variables were checked for (for example null pointer checks and buffer sizes) using a user defined macro which none of the tools seemed to understand. However, some true positives were found and one of the more interesting was that arguments that could be sent to one of the programs created in the module (which was later on used by another part of the module) were not validated to be correct. In this way an attacker easily could have performed a buffer overflow attack.

Since all of the issues in the category "Missing check against NULL" were inspected (see footnote on previous page) this group in the end held the greatest number of true positives. All of the warnings concerned some kind of string function and the function that caused more or less all of the warnings was `strdup()`. `Strdup()` is a function that duplicates a string and insert the copy in a newly allocated memory space. This memory space is allocated via the function `malloc()` which in turn might return NULL.

Another category in which all of the issues reported by more than one tool were true positives was "Command Injection". One of theses warnings occurred since part of a path to an executable that was executed in the code using the `execv()`

[3]http://www.mail-archive.com/splint-discuss@mail.cs.virginia.edu/msg00168.html, http://www.mail-archive.com/splint-discuss@mail.cs.virginia.edu/msg00168/cgrammar.y.patch 2008-08-22

[4]It should be mentioned that an even more strict selection on which issues that should be analyzed was done in these two groups. In order to be thoroughly inspected an issue belonging to any of these two groups had to be found by three of the tools. This narrowed down the numbers of issues to inspect to a more feasible amount considering the time aspect of this thesis.

call consisted of an environment variable. If somehow this environment variable could have been changed by an attacker, an arbitrary executable could have been launched. Since this part of the code was executed with root privileges an attacker in other words could have done more or less anything he/she would have wanted to. The other two warnings in this category occurred since an executable was called by the `system()` function without an absolute path. If an attacker somehow could have changed the $PATH-variable or inserted a malicious executable early on the search path security would have been breached.

The issue that got ranked as the biggest threat was a memory leak issue. A total of two memory leaks were found. Both due to the use of (once again) the function `strdup`. The reason that `strdup` causes so much problems is most likely due to the fact that the developers really does not how it works. If they did know that `strdup` calls the function `malloc` one would expect that they used the function more carefully.

The category "Dead Code" also contained some true positives and all of the issues had to do with a `case` statement in which four return statements, that never would have been executed, were made . This does not imply a very serious risk but the fact that the issues exist should raise a question whether the developer really knows exactly how this particular code construct works. If not, this could result in a more serious issue.

The last category that contained a true positive was "String Termination Error". The function `recvfrom` was used to read from a connection and this function does not NULL terminate the string variable into which the data are read. If later on this string would have been used in a function that requires it to be NULL terminated, the program most likely would have crashed.

4.5 Test result of Module B

As in the case with Module A also Module B had the most warnings in the two categories concerning null dereferences and buffer overflows and also in this case the majority of the warnings were false positives (also in this module an issue belonging to these groups had to be found by three tools in order to be inspected by the author).

The categories containing the most true positives in this module were "Missing Check Against Null" and "Unchecked Return value" and once again it was the `strdup()` function that generated most of the true positives in the missing check against null category. The true positives in the group concerning unchecked return values was generated by calls to `mkdir()` and `read()`.

The last category that held a true positive was "String Termination Error". Also this time it was the function `recvfrom` that caused the issue.

As a whole Module B did not contain many true errors and the majority of the ones found were not of a very critical manner.

Issue	Flawfinder	Splint	Fortify SCA	Codesonar	Found by more than one tool
Missing check against NULL	-	-	14	(15)	3
Unchecked Return Value	-	-	5	7	4
Dead Code	-	4	0	7	4
Double Free	-	0	0	0	0
Memory Leak	-	730	6	4	2
NULL Dereference	-	113	102	21	103
Divide by Zero	-	-	0	0	0
Uninitialized Variable	-	24	16	21	18
Unreleased Resource	-	-	0	-	0
Use After Free	-	6	0	0	0
Buffer Overflow	75	14	45	0	43
Buffer Underrun	0	0	0	0	0
Command Injection	3	-	4	-	3
Format String	1	0	2	0	0
Illegal Pointer Value	-	-	0	-	0
Integer Overflow	37	39	0	0	0
Log Forging	-	-	2	-	0
String Termination Error	21	-	54	0	7
Insecure Randomness	0	-	0	-	0
TOCTOU	0	-	0	0	0
Deadlocks	-	-	0	0	0

Table 4.1. Test Result of Module A

Issue	Flawfinder	Splint	Fortify SCA	Codesonar	Found by more than one tool
Missing check against NULL	-	-	8	(2)	1
Unchecked Return Value	-	-	8	6	6
Dead Code	-	0	0	4	0
Double Free	-	0	0	0	0
Memory Leak	-	421	3	1	1
NULL Dereference	-	44	52	8	37
Divide by Zero	-	-	0	0	0
Uninitialized Variable	-	3	2	5	2
Unreleased Resource	-	-	0	-	0
Use After Free	-	0	0	0	0
Buffer Overflow	36	12	19	0	15
Buffer Underrun	0	0	0	0	0
Command Injection	5	-	1	-	1
Format String	3	0	5	7	3
Illegal Pointer Value	-	-	0	-	0
Integer Overflow	7	11	0	0	2
Log Forging	-	-	3	-	0
String Termination Error	44	-	3	0	2
Insecure Randomness	0	-	4	-	0
TOCTOU	1	-	0	0	0
Deadlocks	-	-	0	1	0

Table 4.2. Test Result of Module B

4.6 Comparison of the results of the tools

When looking at table 4.1 and table 4.2 there are some different things that are of particular interest when one compares the results of the tools between one and another.

The first thing that distinguishes itself is the number of warnings produced by Splint in the memory leak category. Compared to Fortify SCA and CodeSonar, that also perform this check, Splint finds an enormous numbers of memory leaks. One reason of this is probably due to the lack of annotations to the code, making Splint unaware about special presumptions that would have made the scan more accurate. Another reason is also probably due to the fact that Splint does not build an equally exact model of the program, as do the commercial tools.

Another interesting observation is the difference between the tools ability of finding buffer overflows. Flawfinder finds a lot of buffer overflows, but as mentioned earlier in this thesis Flawfinder produces a lot of false positives. However, the most interesting about this category is that CodeSonar does not report one single warning. This is rather strange considering that it should be able to find this kind of bug and all of the other tools manage to find quite many. As pointed out in the previous part in this chapter many of the warnings for buffer overflows were false positives, but some true positives were found, making one wonder if CodeSonar actually performs a scan for buffer overflows, and if it does, how good it actually is at finding them.

Even if CodeSonar performs badly in finding buffer overflows it certainly makes a better job in the null dereference category. Both Splint and Fortify SCA reports a large number of warnings in this category (both in module A and module B). Unfortunately most of them turned out to be false positives. CodeSonar on the other hand reports a much lower number of warnings but compared to Splint and Fortify SCA a bigger share of them are actually true positives.

Chapter 5

Discussion

5.1 Results

This thesis has investigated what different tools for static analysis with an emphasis on security and robustness are available and which of these that possibly could be used in a project at Ericsson AB in a project whose goal is to develop a HIGA. A number of tools were found of which four were chosen to be thoroughly evaluated, two of them were open source tools while two were commercial. As a second goal an analysis should be made on part of the HIGA source code using the tools chosen to be evaluated.

The different categories that were evaluated were the documentation the tools provided, the installation and integration procedure, usability and how an analysis was performed, performance in terms of the tool's ability of finding bugs, amount of false positives/false negatives and the time it takes to perform an analysis. The different sorts of vulnerabilities checked for were also investigated. A summary of the result of the evaluation is seen in table 5.1. The complete survey is found in chapter three.

All together the evaluation showed that the two commercial tools by far are more mature than the two open source tools in all of the categories compared.

The second goal with this thesis was to perform a scan of the HIGA source code with the four chosen tools in order to some extent determine the code quality in terms of robustness and security. Two modules were analyzed and in both of them a number of different coding mistakes were found of which some could lead to an abnormal termination of the program whereas others possibly could have been used to breach security.

The scan of the HIGA source code also confirmed the results from the evaluation chapter in that the two commercial tools did not only produced better results but also made the whole process of an audit a lot easier compared to the two open source tools.

	Flawfinder	Splint	Fortify SCA	CodeSonar
Documentation				
Clearness	Very good and straightforward.	Very good and straightforward.	Very good and straightforward.	Very good and straightforward.
Beginner's guide?	Some examples which is enough to get the user started.	Some basic examples on how to get started. Homepage also has a tutorial on how to use LCLint.	Tutorials on how to perform a scan and inspect results using Audit Workbench.	Tutorials on all of the parts of how to perform an analysis.
List of bugs searched for?	No	Yes	Yes	Yes
Information about the bugs searched for?	Not in manual but in result of scan.	Yes. Very good.	Yes. Very good.	Yes. Very good.
Installation				
Installation procedure straightforward?	Yes. Install using packet manager or build from source.	Yes. Newest version: build from source. Some old versions available through packet manager system.	Yes. Installation guide with a "click-and-go" approach.	Yes. Unzip/untar and then ready for use
Supported platforms	UNIX-like systems.	Newest version: UNIX/Linux. Older versions: Windows, OS/2, BSD, Solaris	Linux, Windows, Solaris, AIX, Mac OS X	Linux, Windows, Solaris
Integrating with existing build environment	No explicit support.	No explicit support.	Ant & Make	Make & Nmake
Supported IDEs	None	None	Visual Studio	Visual Studio
Programming languages	C	C	Java, .NET, C/C++, PL-SQL, T-SQl & ColdFusion	C/C++

Usability				
Impact on audit process from code size and complexity	None. Pass folder containing source code and Flawfinder scans the appropriate files.	Big. Manually edit the Makefile or write script to perform scan. Paths to header files etc. has to be passed as input, many flags has to be set in order for it to work.	None. Pass the Makefile as input.	None. Pass the Makefile as input.
Restriction on amount of code?	No	No	No	No
Results presentation	Command line, text file, html file. Sorted on risk level.	Command line, text file, csv file, "semi-html" file. No sorting.	Command line, text file, fpr (for Audit Workbench), FVDL (an xml dialect). Many sorting alternatives.	Presented as a web page in a stand-alone web server provided by CodeSonar. Many sorting alternatives
Performance				
Bugs found	44.74%	28.95%	73.68%	39.47%
False positive Rate	81.08%	7.14%	41.54%	6.00%
False Negative Rate	5.56%	50.00%	15.15%	38.00%
Time of analysis of Pure-ftpd 1.0.21 & HIGA	1-2 & 1 seconds	2-3 & 20 seconds	80 & 65 seconds	4 & 11 minutes
Bugs searched for				
Types of bugs	Security	Code quality and security.	Security	Code quality and security.
Possibility to create new rules?	Yes	Yes	Yes	Yes
Documentation about how to create new rules?	No	Yes	No	Yes

Table 5.1. Result of survey

5.2 Conclusions

5.2.1 Tools evaluation

The authors conclusions about which of the four tools evaluated is better suited to be used in the HIGA project greatly depends on in what context they are being used. The open source tool Flawfinder is a very simple tool (both to use and how it performs a scan), more or less only doing a sophisticated search in the code for functions that are known to, if used incorrectly, affect security of a program in a bad way. This implies that Flawfinder produces a rather hefty amount of false positives which in turn leads to a slow and time consuming analysis of the result. With this in mind my conclusion about using Flawfinder in the HIGA project is that it should only be used on a single file basis by the developers responsible for the file in question. Perhaps just before check in to a version control system (VCS) as a way of being sure that no "dangerous" functions are being used in the code and in those cases they are, the developer can make sure that they are being used in a correct way.

The other open source program, Splint, is more advanced than Flawfinder in more or less all aspects. From a user perspective this is most noticeable when first trying to perform an analysis on a large project consisting of a large number of files. The first attempts to scan the HIGA source code with Splint were all failures in that Splint did not find include files, did not understand different code constructs and reported a lot of parse errors due to that Splint is written in C89 standard and of that reason is only able to check C89 standard compliant code. All of this, together with the fact that Splint on an initial scan of the HIGA code reported around 5000 warnings, made me come to the conclusion that Splint is not a tool that can be used in the HIGA project. If this thesis would have been written when the HIGA project just was started and thus in the initial phase of the SDLC another conclusion might have been reached. In that case code annotations could have been made to the code as a natural step of the coding process which in turn probably would have had a great impact on the result.

As already pointed out in the results both of the commercial tools keeps a very mature level compared to the open source tools, but still they both have things that can be improved.

Fortify SCA showed very good results if one sees to the ability of finding bugs, rate of false negatives but the rate of false positives (both established in the tests in the performance chapter and during the test of the HIGA) makes the inspection of the issues to a rather slow task. Furthermore Fortify SCA does not provide any functionality for collaborative working between multiple of auditors and developers. However, the Fortify 360 suite includes a newer version of the SCA as well as modules that opens up for just that possibility. It is also possible that the newest version of the SCA (5.0) would have managed to suppress the numbers of false positives better than version 4.0 that was tested in this thesis. However, this is only reflections made by the author and should not be taken as hard facts.

Codesonar, which was the other commercial tool evaluated comes with support for collaborative as part of the tool and not as a separate module as in the case with

Fortify 360. The idea of presenting the results in a web environment was really nice and worked very well and when working with the HIGA project it shoved that this was the tool that by far had the best functionality of presenting the code to the auditor. The downside with Codesonar was that it missed one to many weaknesses when tested against the test suites put together in the performance chapter.

My final conclusion about what tools that are best suited to be used in the HIGA project would have to be the Fortify SCA together with the 360 suite in order to open up for things such as collaborative working, and Flawfinder to be used on a single file basis. However, introducing a tool this late in the SDLC might be hard and might give rise to a lot of extra work.

5.2.2 Result of scan of HIGA

My conclusions about the code quality in the HIGA project from a security point of view are as a whole very good. There were not a lot of true positives found by the tools, probably indicating that the code holds a high level. However, some different kinds of bugs were found of which some can jeopardize the stability of the product as well as being exploited in order to gain access to the underlying system running on the HIGA. Since the HIGA code is running as a built in system, a first step that has to be taken before being able to exploit some of the vulnerabilities is to actually gain access to the gateway. This could be for example through a Secure Shell (SSH) connection. This of course to some extent prevents the possibilities to exploit some of the bugs but all of the true positives found should nevertheless be fixed. Since most of them are rather simple this should not be a very hard task.

Bibliography

[1] About us - cybercom group. URL: http://www.cybercomgroup.com/en/Home-/About-us/ (2008-06-12).

[2] abstract syntax tree - definitions from dictionary.com. URL: http://dictionary.reference.com/browse/abstract%20syntax%20tree (2008-06-24).

[3] Bridging the ims gap to the home. URL: http://www.ericsson.com/solutions/news/2007/q1/20070124_ims.shtml (2008-06-18).

[4] Code audit. URL: http://en.wikipedia.org/wiki/Code_audit (2008-07-29).

[5] Fortify sca. URL: http://www.fortify.com/servlet/downloads/public/Fortify_SCA.pdf (2008-06-24).

[6] Overview of grammatech static-analysis technology. URL: http://www.grammatech.com/products/codesonar/GrammaTechCodeSonarOverview.pdf (2008-06-24).

[7] Representative codesonar checks. URL: http://www.grammatech.com/products/codesonar/listofchecks.html (2008-07-24).

[8] Source code audit - faq - source code security. URL: http://ouncelabs.com/resources/code-audit-faq.asp (2008-06-12).

[9] Splint - release 3.0.1. URL: http://www.splint.org/release.html.

[10] *Splint Manual - Version 3.1.1-1*, 2003. URL: http://www.splint.org/manual/. Included with every Splint distributions.

[11] Amit Chaturvedi. Java & static analysis - finding bugs early with automated code reviews. URL: http://www.ddj.com/java/184406143?pgno=1 (2008-06-25), 2005.

[12] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[13] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison Wesley, 2007. ISBN 0-321-42477-8.

[14] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools (extended version). Technical report, Department of Computer and Information Science, Linköping University, 2008.

[15] Thomas Magedanz. Ims- ip multimedia subsystem, towards a unified platform for multimedia services, 2006. URL: `http://www.eurescom.de/message/messagemar2006/IMS_%20IP_Multimedia_Subsystem.asp` (2008-06-12).

[16] Gary McGraw. *Software Security - Building Security In*. Addison-Wesley, 2006. ISBN 0-321-35670-5.

[17] Miika Poikselkä, Gerog Mayer, Hisham Khartabil, and Aki Niemi. *The IMS IP Multimedia Concepts and Services*. John Wiley and Sons, 2 edition, 2006. ISBN 0-470-01906-9.

[18] David A. Wheeler. Flawfinder home page. URL: `http://www.dwheeler.com/flawfinder/`.

Acronyms

3GPP	Third Generation Partnership Program
AST	Abstract Syntax Tree
CDMA	Code Division Multiple Access
CFG	Control Flow Graph
GSM	Global System for Mobile communications
IDE	Integrated Development Environment
IMS	Internet Multimedia Subsystem
ISDN	Integrated Services Digital Network
IPTV	Internet Protocol Television
HIGA	Home IMS Gateway
HSS	Home Subscriber Server
PoC	Push to talk Over Cellular
PSTN	Public Switched Telephony Network
QA	Quality Assurance
SDLC	Software Development Life Cycle
SIP	Session Initiation Protocol
SLOC	Source lines of code
SSH	Secure Shell
QoS	Quality of Service
uPnP	Universal Plug and Play
VCS	Version Control System

Appendix A

Taxonomy of security vulnerabilities

1. **Input validation and representation**

   - **Buffer Overflow.** Writing outside the bounds of allocated memory can corrupt data, crash the program, or cause the execution of an attack payload.
   - **Command Injection.** Executing commands from an untrusted source or in an untrusted environment can cause an application to execute malicious commands on behalf of an attacker.
   - **Cross-Site Scripting.** Sending unvalidated data to a Web browser can result in the browser executing malicious code (usually scripts).
   - **Format String.** Allowing an attacker to control a function's format string may result in a buffer overflow.
   - **HTTP Response Splitting.** Writing unvalidated data into an HTTP header allows an attacker to specify the entirety of the HTTP response rendered by the browser.
   - **Illegal Pointer Value.** This function can return a pointer to memory outside of the buffer to be searched. Subsequent operations on the pointer may have unintended consequences.
   - **Integer Overflow.** Not accounting for integer overflow can result in logic errors or buffer overflows.
   - **Log Forging.** Writing unvalidated user input into log files can allow an attacker to forge log entries or inject malicious content into logs.
   - **Path Manipulation.** Allowing user input to control paths used by the application may enable an attacker to access otherwise protected files.

- **Process Control.** Executing commands or loading libraries from an untrusted source or in an untrusted environment can cause an application to execute malicious commands (and payloads) on behalf of an attacker.
- **Resource Injection.** Allowing user input to control resource identifiers may enable an attacker to access or modify otherwise protected system resources.
- **Setting Manipulation.** Allowing external control of system settings can disrupt service or cause an application to behave in unexpected ways.
- **SQL Injection.** Constructing a dynamic SQL statement with user input may allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.
- **String Termination Error.** Relying on proper string termination may result in a buffer overflow.
- **Struts: Duplicate Validation Forms.** Multiple validation forms with the same name indicate that validation logic is not up-to-date.
- **Struts: Erroneous validate() Method.** The validator form defines a validate() method but fails to call super.validate().
- **Struts: Form Bean Does Not Extend Validation Class.** All Struts forms should extend a Validator class.
- **Struts: Form Field Without Validator.** Every field in a form should be validated in the corresponding validation form.
- **Struts: Plug-in Framework Not In Use.** Use the Struts Validator to prevent vulnerabilities that result from unchecked input.
- **Struts: Unused Validation Form.** An unused validation form indicates that validation logic is not up-to-date.
- **Struts: Unvalidated Action Form.** Every Action Form must have a corresponding validation form.
- **Struts: Validator Turned Off.** This Action Form mapping disables the form's validate() method.
- **Struts: Validator Without Form Field.** Validation fields that do not appear in forms they are associated with indicate that the validation logic is out of date.
- **Unsafe JNI.** Improper use of the Java Native Interface (JNI) can render Java applications vulnerable to security bugs in other languages.
- **Unsafe Reflection.** An attacker may be able to create unexpected control flow paths through the application, potentially bypassing security checks.
- **XML Validation.** Failure to enable validation when parsing XML gives an attacker he opportunity to supply malicious input.

2. **API Abuse**

   - **Dangerous Function.** Functions that cannot be used safely should never be used.
   - **Directory Restriction.** Improper use of the chroot() system call may allow attackers to escape a chroot jail.
   - **Heap Inspection.** Do not use realloc() to resize buffers that store sensitive information.
   - **J2EE Bad Practices: getConnection().** The J2EE standard forbids the direct management of connections.
   - **J2EE Bad Practices: Sockets.** Socket-based communication in web applications is prone to error.
   - **Often Misused: Authentication.** Do not rely on the name the getlogin() family of functions returns because it is easy to spoof.
   - **Often Misused: Exception Handling.** A dangerous function can throw an exception, potentially causing the program to crash.
   - **Often Misused: File System.** Passing an inadequately-sized output buffer to a path manipulation function can result in a buffer overflow.
   - **Often Misused: Privilege Management.** Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities.
   - **Often Misused: Strings.** Functions that manipulate strings encourage buffer overflows.
   - **Unchecked Return Value.** Ignoring a method's return value can cause the program to overlook unexpected states and conditions.

3. **Security Features**

   - **Insecure Randomness.** Standard pseudo-random number generators cannot withstand cryptographic attacks.
   - **Least Privilege Violation.** The elevated privilege level required to perform operations such as chroot() should be dropped immediately after the operation is performed.
   - **Missing Access Control.** The program does not perform access control checks in a consistent manner across all potential execution paths.
   - **Password Management.** Storing a password in plaintext may result in a system compromise.
   - **Password Management: Empty Password in Config File.** Using an empty string as a password is insecure.
   - **Password Management: Hard-Coded Password.** Hard coded passwords may compromise system security in a way that cannot be easily remedied.

- **Password Management: Password in Config File.** Storing a password in a configuration file may result in system compromise.
- **Password Management: Weak Cryptography.** Obscuring a password with a trivial encoding does not protect the password.
- **Privacy Violation.** Mishandling private information, such as customer passwords or social security numbers, can compromise user privacy and is often illegal.

4. **Time and State**

- **Deadlock.** Inconsistent locking discipline can lead to deadlock.
- **Failure to Begin a New Session upon Authentication.** Using the same session identifier across an authentication boundary allows an attacker to hijack authenticated sessions.
- **File Access Race Condition: TOCTOU.** The window of time between when a file property is checked and when the file is used can be exploited to launch a privilege escalation attack.
- **Insecure Temporary File.** Creating and using insecure temporary files can leave application and system data vulnerable to attack.
- **J2EE Bad Practices: System.exit().** A Web application should not attempt to shut down its container.
- **J2EE Bad Practices: Threads.** Thread management in a Web application is forbidden in some circumstances and is always highly error prone.
- **Signal Handling Race Conditions.** Signal handlers may change shared state relied upon by other signal handlers or application code causing unexpected behavior.

5. **Errors**

- **Catch NullPointerException.** Catching NullPointerException should not be used as an alternative to programmatic checks to prevent dereferencing a null pointer.
- **Empty Catch Block.** Ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed.
- **Overly-Broad Catch Block.** Catching overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.
- **Overly-Broad Throws Declaration.** Throwing overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.

6. **Code Quality**

- **Double Free.** Calling free() twice on the same memory address can lead to a buffer overflow.
- **Inconsistent Implementations.** Functions with inconsistent implementations across operating systems and operating system versions cause portability problems.
- **Memory Leak.** Memory is allocated but never freed leading to resource exhaustion.
- **Null Dereference.** The program can potentially dereference a null pointer, thereby raising a NullPointerException.
- **Obsolete.** The use of deprecated or obsolete functions may indicate neglected code.
- **Undefined Behavior.** The behavior of this function is undefined unless its control parameter is set to a specific value.
- **Uninitialized Variable.** The program can potentially use a variable before it has been initialized.
- **Unreleased Resource.** The program can potentially fail to release a system resource.
- **Use After Free.** Referencing memory after it has been freed can cause a program to crash.

7. **Encapsulation**

- **Comparing Classes by Name.** Comparing classes by name can lead a program to treat two classes as the same when they actually differ.
- **Data Leaking Between Users.** Data can "bleed" from one session to another through member variables of singleton objects, such as Servlets, and objects from a shared pool.
- **Leftover Debug Code.** Debug code can create unintended entry points in an application.
- **Mobile Code: Object Hijack.** Attackers can use Cloneable objects to create new instances of an object without calling its constructor.
- **Mobile Code: Use of Inner Class.** Inner classes are translated into classes that are accessible at package scope and may expose code that the programmer intended to keep private to attackers.
- **Mobile Code: Non-Final Public Field.** Non-final public variables can be manipulated by an attacker to inject malicious values.
- **Private Array-Typed Field Returned From a Public Method.** The contents of a private array may be altered unexpectedly through a reference returned from a public method.
- **Public Data Assigned to Private Array-Typed Field.** Assigning public data to a private array is equivalent giving public access to the array.

- **System Information Leak.** Revealing system data or debugging information helps an adversary learn about the system and form an attack plan.
- **Trust Boundary Violation.** Commingling trusted and untrusted data in the same data structure encourages programmers to mistakenly trust unvalidated data.

Appendix B

Selection of tools for further evaluation

The tools that were chosen to be further evaluated had to fulfill the requirements that they should support analysis of code written in C, the gcc compiler and the make build tool. Most of the development in the HIGA project are being done in a Linux environment so if the tools could be used in Linux this was beneficial.

Another requirement stated that the tool should have some support to perform a security audit of the code. One can argue about that also a general audit of code quality concerns security since availability is one of the three pillars that computer security stands on[1]. Even so, it seemed that the tools should provide some distinct functionality for finding security vulnerabilities since this thesis focuses mainly on software security.

The last requirement was that the tool had to be open source or offer an evaluation license if it was commercial.

One last thing that played a role in the selection of the tools (even if it was not a requirement per se) was that it would be interesting to choose tools with different approaches to how the scans are performed and how advanced the tools are.

Flawfinder was chosen since this is one of the earliest tools and its approach of doing only a lexical analysis gives a perfect foundation for a comparison of the different tools. Furthermore since the goal of this work was to find alternatives of tools Flawfinder is probably one of the easiest to get started with even if it does not produce the best results. Flawfinder is used in order to scan the code for security vulnerabilities, is free and supports scanning of C code and thus makes a perfect candidate for further evaluation.

The reason that Flawfinder was chosen and not RATS was that it had a much more up to date version (version 1.27 released in 2007 compared to RATS v. 2.1 which was released in 2002).

The next chosen tool, Splint, is one step up from Flawfinder in that it performs

[1]Where confidentiality and integrity are the other two

an analysis on the semantic level. Splint is open source and even if it is not as advanced as the two commercial tools that were chosen, it still is well known among tools for static analysis. The tool has support for finding security vulnerabilities as well as other coding errors. The approach of making use of annotations seemed like an interesting thing to bring up when looking for alternatives and since the tool supports C code and can be used on Linux it was chosen to be further evaluated.

The Fortify Security Analyzer is as the name implies a tool that focuses mainly on security vulnerabilities. Since an evaluation license (even if this did not include full functionality) could be obtained and all other requirements were fulfilled this tool was chosen as one of the commercial ones.

CodeSonar is a tool that focuses on both code quality and security and since all requirements were met it was chosen to be the last tool to evaluate.

Klockwork Insight, which was the third commercial tool that fulfilled all of the requirements, was not chosen since no evaluation copy of the tool was offered to the author even if they had one. Somewhat contradictive to Klockwork's homepage which clearly stated that a free evaluation copy of the software could be obtained after registration.

Coverity Prevent did also meet all of the requirements. However, due to that a rather comprehensive questionnaire had to be answered in order to obtain the evaluation license, and people on the HIGA project all were very busy, this tool was not chosen to be further evaluated.

Appendix C

Detailed Test Results of FP/FN Tests

In this appendix the test results of the tests that were performed in order to find out the tools ability of finding bugs and rates of false positives/false negatives are presented.

The test case ID:s are reffering to the ones at the SAMATE Reference Dataset Project [1].

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

Test Case ID	Weakness	Found
10	Format String Vulnerability	x
11	Command Injection	x
92	Format String Vulnerability	x
93	Format String Vulnerability	x
102	TOCTOU	x
1544	Stack Overflow	x
1548	Stack Overflow	x
1563	Stack Overflow	x
1565	Stack Overflow	x
1612	Heap Overflow	
1751	Stack Overflow	
1780	Command Injection	x
1806	TOCTOU	x
1808	TOCTOU	x
1831	Format String Vulnerability	x
1833	Format String Vulnerability	x
1843	Heap Overflow	x
1845	Heap Overflow	x
1847	Heap Overflow	x
1849	Improper NULL termination	x
1850	Improper NULL termination	x
1854	Improper NULL termination	x
1857	Improper NULL termination	x
1865	String Based BO	x
1867	String Based BO	x
1869	String Based BO	x
1871	String Based BO	x
1873	String Based BO	x
1881	Command Injection	x
1883	Command Injection	x
1885	Command Injection	x
1907	Stack Overflow	x
1909	Stack Overflow	x
2009	Stack Overflow	x
2010	Improper NULL termination	x
2074	Heap Overflow	x
Total number of cases:		36
Found		**34**
Missed		**2**
% FN		**5,56%**

Table C.1. Flawfinder False Negative Rate

Test Case ID	Weakness	Warning
1545	Stack Overflow	x
1547	Stack Overflow	x
1549	Stack Overflow	x
1556	Format String Vulnerability	
1560	Format String Vulnerability	
1562	Format String Vulnerability	x
1566	Stack Overflow	x
1574	Heap Overflow (Integer overflow)	
1602	Stack Overflow	x
1613	Heap Overflow	
1615	Heap Overflow	x
1807	TOCTOU	x
1809	TOCTOU	x
1832	Format String Vulnerability	
1834	Format String Vulnerability	
1844	Heap Overflow	
1848	Heap Overflow	x
1851	Improper Null Termination	x
1855	Improper Null Termination	x
1856	Improper Null Termination	x
1858	Improper Null Termination	x
1866	String Based BO	x
1868	String Based BO	x
1870	String Based BO	x
1872	String Based BO	x
1874	String Based BO	x
1882	Command Injection	x
1884	Command Injection	x
1886	Command Injection	x
1892	Race Condition	x
1894	Race Condition	x
1906	Stack Overflow	x
1908	Stack Overflow	x
1910	Stack Overflow	x
1931	Command Injection	x
2006	Heap Overflow	x
2012	Improper Null Termination	x
Total number of cases:		**37**
Warnings		**30**
% FP		**81,08%**

Table C.2. Flawfinder False Positive Rate

Test Case ID	Weakness	Found
6	Use After Free	x
10	Format String Vulnerability	x
92	Format String Vulnerability	x
93	Format String Vulnerability	x
99	Double Free	x
1508	Double Free	x
1544	Stack Overflow	x
1548	Stack Overflow	
1563	Stack Overflow	x
1565	Stack Overflow	
1588	Memory Leak	x
1590	Double Free	x
1612	Heap Overflow	
1751	Stack Overflow	
1757	Uninitialized Variable	x
1827	Double Free	x
1829	Double Free	x
1831	Format String Vulnerability	x
1833	Format String Vulnerability	x
1843	Heap Overflow	x
1845	Heap Overflow	
1847	Heap Overflow	
1865	String Based BO	
1867	String Based BO	
1869	String Based BO	
1871	String Based BO	
1873	String Based BO	
1875	Null Dereference	
1877	Null Dereference	
1879	Null Dereference	
1907	Stack Overflow	
1909	Stack Overflow	
1911	Use After Free	
1913	Use After Free	x
1915	Use After Free	x
1917	Use After Free	x
1928	Unchecked Error Condition	x
2001	Null Pointer Dereference	x
2003	Uninitialized Variable	x
2009	Stack Overflow	
2074	Heap Overflow	
Total number of cases:		**44**
Found		**22**
Missed		**22**
% FN		**50,00%**

Table C.3. Splint False Negative Rate

Test Case ID	Weakness	Warning
1545	Stack Overflow	
1547	Stack Overflow	
1549	Stack Overflow	
1556	Format String Vulnerability	
1560	Format String Vulnerability	
1562	Format String Vulnerability	x
1566	Stack Overflow	
1574	Heap Overflow (Integer overflow)	
1586	Memory leak	
1589	Memory leak	x
1591	Double Free	
1602	Stack Overflow	
1613	Heap Overflow	
1615	Heap Overflow	
1828	Double Free	
1830	Double Free	
1832	Format String Vulnerability	
1834	Format String Vulnerability	
1844	Heap Overflow	
1848	Heap Overflow	
1866	Often Misused: String Management	
1868	Often Misused: String Management	
1870	Often Misused: String Management	
1872	Often Misused: String Management	
1874	Often Misused: String Management	
1880	Null Dereference	
1882	Command Injection	
1906	Stack Overflow	
1908	Stack Overflow	
1910	Stack Overflow	
1912	Use After Free	
1914	Use After Free	
1918	Use After Free	
1925	Memory leak	x
1926	Memory leak	
1929	Unchecked Error Condition	
1932	Double Free	
1933	Memory leak	
2002	Null Dereference	
2004	Uninitialized Variable	
2006	Heap Overflow	
2008	Use After Free	
Total number of cases:		**42**
Warnings		**3**
%FP		**7,14%**

Table C.4. Splint False Positive Rate

Test Case ID	Weakness	Found
6	Use After Free	x
10	Format String Vulnerability	x
11	Command Injection	x
92	Format String Vulnerability	x
93	Format String Vulnerability	x
99	Double Free	x
102	TOCTOU	
1508	Double Free	x
1544	Stack Overflow	x
1548	Stack Overflow	x
1563	Stack Overflow	x
1565	Stack Overflow	x
1585	Memory Leak	x
1588	Memory Leak	
1590	Double Free	x
1612	Heap Overflow	x
1737	Heap Inspection (API Abuse)	x
1751	Stack Overflow	x
1757	Uninitialized Variable	x
1780	Command Injection	x
1796	SQL Injection	x
1798	SQL Injection	x
1800	SQL Injection	x
1806	TOCTOU	
1808	TOCTOU	
1827	Double Free	x
1829	Double Free	x
1831	Format String Vulnerability	x
1833	Format String Vulnerability	x
1835	Hard Coded Password	
1837	Hard Coded Password	
1839	Hard Coded Password	
1841	Hard Coded Password	
1843	Heap Overflow	x
1845	Heap Overflow	x
1847	Heap Overflow	x
1849	Improper NULL termination	x
1850	Improper NULL termination	x
1854	Improper NULL termination	x
1857	Improper NULL termination	x
1865	String Based BO	x
1867	String Based BO	x
1869	String Based BO	x

1871	String Based BO	x
1873	String Based BO	x
1875	Null Dereference	x
1877	Null Dereference	x
1879	Null Dereference	
1881	Command Injection	x
1883	Command Injection	x
1885	Command Injection	x
1895	Resource Injection	x
1897	Resource Injection	x
1899	Resource Injection	x
1901	Resource Injection	x
1907	Stack Overflow	x
1909	Stack Overflow	x
1911	Use After Free	x
1913	Use After Free	x
1915	Use After Free	
1917	Use After Free	x
2001	Null Pointer Dereference	x
2003	Uninitialized Variable	x
2009	Stack Overflow	x
2010	Improper NULL termination	x
2074	Heap Overflow	x
Total number of cases:		**66**
Found		**56**
Missed		**10**
% FN		**15,15%**

Table C.5. Fortify SCA False Negative Rate

Test Case ID	Weakness	Warning
1545	Stack Overflow	x
1547	Stack Overflow	
1549	Stack Overflow	
1556	Format String Vulnerability	
1560	Format String Vulnerability	x
1562	Format String Vulnerability	x
1566	Stack Overflow	
1574	Heap Overflow (Integer overflow)	x
1586	Memory leak	
1589	Memory leak	
1591	Double Free	
1602	Stack Overflow	x
1613	Heap Overflow	
1615	Heap Overflow	x
1797	SQL Injection	x
1799	SQL Injection	x
1801	SQL Injection	x
1807	TOCTOU	x
1809	TOCTOU	x
1828	Double Free	
1830	Double Free	
1832	Format String Vulnerability	
1834	Format String Vulnerability	
1836	Hard Coded Password	
1838	Hard Coded Password	
1840	Hard Coded Password	
1822	Hard Coded Password	
1844	Heap Overflow	x
1848	Heap Overflow	x
1851	Improper Null Termination	x
1855	Improper Null Termination	x
1856	Improper Null Termination	
1858	Improper Null Termination	
1866	String Based BO	
1868	String Based BO	
1870	String Based BO	
1872	String Based BO	
1874	String Based BO	
1880	Null Dereference	
1882	Command Injection	x
1884	Command Injection	x
1886	Command Injection	x
1892	Race Condition	x
1894	Race Condition	x
1896	Reource Injection	x

1898	Reource Injection	x
1900	Reource Injection	x
1902	Reource Injection	x
1906	Stack Overflow	
1908	Stack Overflow	
1910	Stack Overflow	
1912	Use After Free	
1914	Use After Free	
1918	Use After Free	
1925	Memory leak	
1926	Memory leak	
1930	SQL Injection	x
1931	Command Injection	x
1932	Double Free	
1933	Memory leak	
2002	Null Dereference	
2004	Uninitialized Variable	
2006	Heap Overflow	x
2008	Use After Free	
2012	Improper Null Termination	
Total number of cases:		**65**
Warnings		**27**
%FP		**41,54%**

Table C.6. Fortify SCA False Positive Rate

Test Case ID	Weakness	Found
6	Use After Free	x
10	Format String Vulnerability	x
92	Format String Vulnerability	
93	Format String Vulnerability	x
99	Double Free	x
102	TOCTOU	x
1508	Double Free	x
1544	Stack Overflow	
1548	Stack Overflow	
1563	Stack Overflow	x
1565	Stack Overflow	x
1585	Memory Leak	
1588	Memory Leak	
1590	Double Free	x
1612	Heap Overflow	x
1751	Stack Overflow	x
1757	Uninitialized Variable	x
1806	TOCTOU	
1808	TOCTOU	
1827	Double Free	x
1829	Double Free	x
1831	Format String Vulnerability	x
1833	Format String Vulnerability	
1843	Heap Overflow	
1845	Heap Overflow	x
1847	Heap Overflow	x
1849	Improper NULL termination	
1850	Improper NULL termination	
1854	Improper NULL termination	
1857	Improper NULL termination	x
1865	String Based BO	x
1867	String Based BO	
1869	String Based BO	
1871	String Based BO	
1873	String Based BO	
1875	Null Dereference	x
1877	Null Dereference	x
1879	Null Dereference	x
1907	Stack Overflow	x
1909	Stack Overflow	x
1911	Use After Free	x
1913	Use After Free	x
1915	Use After Free	x

1917	Use After Free	x
1928	Unchecked Error Condition	x
2001	Null Pointer Dereference	x
2003	Uninitialized Variable	x
2009	Stack Overflow	
2010	Improper NULL termination	
2074	Heap Overflow	
Total number of cases:		**50**
Found		**31**
Missed		**19**
% FN		**38,00%**

Table C.7. CodeSonar False Negative Rate

Test Case ID	Weakness	Warning
1545	Stack Overflow	
1547	Stack Overflow	
1549	Stack Overflow	
1556	Format String Vulnerability	
1560	Format String Vulnerability	
1562	Format String Vulnerability	
1566	Stack Overflow	
1574	Heap Overflow (Integer overflow)	
1586	Memory leak	
1589	Memory leak	
1591	Double Free	
1602	Stack Overflow	
1613	Heap Overflow	
1615	Heap Overflow	
1807	TOCTOU	
1809	TOCTOU	
1828	Double Free	
1830	Double Free	
1832	Format String Vulnerability	
1834	Format String Vulnerability	
1844	Heap Overflow	
1848	Heap Overflow	x
1851	Improper Null Termination	
1855	Improper Null Termination	
1856	Improper Null Termination	
1858	Improper Null Termination	
1866	String Based BO	
1868	String Based BO	
1870	String Based BO	
1872	String Based BO	
1874	String Based BO	
1880	Null Dereference	
1892	Race Condition	x
1894	Race Condition	x
1906	Stack Overflow	
1908	Stack Overflow	
1910	Stack Overflow	
1912	Use After Free	
1914	Use After Free	
1918	Use After Free	
1925	Memory leak	
1926	Memory leak	
1929	Unchecked Error Condition	
1932	Double Free	
1933	Memory leak	

2002	Null Dereference	
2004	Uninitialized Variable	
2006	Heap Overflow	
2008	Use After Free	
2012	Improper Null Termination	
Total number of cases:		**50**
Warnings		**3**
%FP		**6,00%**

Table C.8. CodeSonar False Positive Rate

Test Case ID	Weakness	Flawfinder	Splint	Fortify SCA	CodeSonar
6	Use After Free		x	x	x
10	Format String Vulnerability	x	x	x	x
11	Command Injection	x		x	
92	Format String Vulnerability	x	x	x	
93	Format String Vulnerability	x	x	x	x
99	Double Free		x	x	x
102	TOCTOU	x			x
1508	Double Free		x	x	x
1544	Stack Overflow	x	x	x	
1548	Stack Overflow	x		x	
1563	Stack Overflow	x	x	x	x
1565	Stack Overflow	x		x	x
1585	Memory Leak			x	
1588	Memory Leak		x		
1590	Double Free		x	x	x
1612	Heap Overflow			x	x
1737	Heap Inspection (API Abuse)			x	
1751	Stack Overflow			x	x
1757	Uninitialized Variable		x	x	x
1780	Command Injection	x		x	
1781	XSS				
1782	Unintentional pointer scaling				
1792	XSS				
1794	XSS				
1796	SQL Injection			x	
1798	SQL Injection			x	
1800	SQL Injection			x	
1806	TOCTOU	x			

Test Case ID	Weakness	Flawfinder	Splint	Fortify SCA	CodeSonar
1808	TOCTOU	x			
1810	Hard-Coded Password				
1827	Double Free		x	x	x
1829	Double Free		x	x	x
1831	Format String Vulnerability	x	x	x	x
1833	Format String Vulnerability	x	x	x	
1835	Hard-Coded Password				
1837	Hard-Coded Password				
1839	Hard-Coded Password				
1841	Hard-Coded Password				
1843	Heap Overflow	x	x	x	
1845	Heap Overflow	x		x	x
1847	Heap Overflow	x		x	x
1849	Improper NULL termination	x		x	
1850	Improper NULL termination	x		x	
1854	Improper NULL termination	x		x	
1857	Improper NULL termination	x		x	x
1861	Left over debug code				
1863	Resource Locking problems				
1865	String based BO	x		x	x
1867	String based BO	x		x	
1869	String based BO	x		x	
1871	String based BO	x		x	
1873	String based BO	x		x	
1875	Null Dereference			x	x
1877	Null Dereference			x	x
1879	Null Dereference				x
1881	Command Injection	x		x	
1883	Command Injection	x		x	

Test Case ID	Weakness	Flawfinder	Splint	Fortify SCA	CodeSonar
1885	Command Injection	x		x	
1895	Resource Injection			x	
1897	Resource Injection			x	
1899	Resource Injection			x	
1901	Resource Injection			x	
1907	Stack Overflow	x		x	x
1909	Stack Overflow	x		x	x
1911	Use After Free			x	x
1913	Use After Free		x	x	x
1915	Use After Free		x		x
1917	Use After Free		x	x	x
1919	XSS				
1921	XSS				
1928	Unchecked Error Condition		x		
2001	Null Pointer Dereference		x	x	x
2003	Uninitialized Variable		x	x	x
2009	Stack Overflow	x		x	
2010	Improper NULL termination	x		x	
2074	Heap Overflow	x		x	
Total number of cases:	**76**	**34**	**22**	**56**	**30**
% Found		**44,74%**	**28,95%**	**73,68%**	**39,47%**

Table C.9. Tools ability of finding bugs

Appendix D

Bugs found by CodeSonar

This list contains the bugs that CodeSonar checks for. It was obtained from [7]

- **Buffer Overrun:** A read or write to data after the end of a buffer.
- **Buffer Underrun:** A read or write to data before the beginning of a buffer.
- **Type Overrun:** An overrun of a boundary within an aggregate type.
- **Type Underrun:** An underrun of a boundary within an aggregate type.
- **Null Pointer Dereference:** An attempt to dereference a pointer to the address 0.
- **Divide By Zero:** An attempt to perform integer division where the denominator is 0.
- **Double Free:** Two calls to free on the same object.
- **Use After Free:** A dereference of a pointer to a freed object.
- **Free Non-Heap Variable:** An attempt to free an object which was not allocated on the heap, such as a stack-allocated variable.
- **Uninitialized Variable:** An attempt to use the value of a variable that has not been initialized.
- **Leak:** Dynamically allocated storage has not been freed.
- **Dangerous Function Cast:** A function pointer is cast to another function pointer having an incompatible signature or return type.
- **Delete[] Object Created by malloc:** An attempt to release memory obtained with malloc using delete[]
- **Delete[] Object Created by new:** An attempt to release memory obtained with new using delete[]

- **Delete Object Created by malloc:** An attempt to release memory obtained with malloc using delete
- **Delete Object Created by new[]:** An attempt to release memory obtained with new[] using delete
- **Free Object Created by new[]:** An attempt to release memory obtained with new[] using free
- **Free Object Created by new:** An attempt to release memory obtained with new using free
- **Missing Return Statement:** At least one path through a non-void return-type function does not contain a return statement.
- **Redundant Condition:** Some condition is either always or never satisfied.
- **Return Pointer To Local:** A procedure returns a pointer to one of its local variables.
- **Return Pointer To Freed:** A procedure returns a pointer to memory that has already been freed.
- **Unused Value:** A variable is assigned a value, but that value is never subsequently used on any execution path.
- **Useless Assignment:** Some assignment always assigns the value that the variable being modified already has.
- **Varargs Function Cast:** A varargs function pointer is cast to another function pointer having different parameters or return type.
- **Ignored Return Value:**The value returned by some function has not been used.
- **Free Null Pointer:** An attempt to free a null pointer.
- **Unreachable Code:** Some of the code in a function is unreachable from the function entry point under any circumstances.
- **Null Test After Dereference:** A pointer is NULL-checked when it has already been dereferenced.
- **Format String:** A function that should have a format string passed in a particular argument position has been passed a string that either is not a format string or is from an untrusted source. (Potential security vulnerability.)
- **Double Close:** An attempt to close a file descriptor or file pointer twice.
- **TOCTTOU Vulnerability:** A time-of-check-to-time-of-use race condition that can create a security vulnerability.

- **Double Lock:** An attempt to lock a mutex twice.
- **Double Unlock:** An attempt to unlock a mutex twice.
- **Try-lock that will never succeed:** An attempt to lock a mutex that cannot possibly succeed.
- **Misuse of Memory Allocation:** Incorrect use of memory allocators.
- **Misuse of Memory Copying:** Incorrect use of copying functions.
- **Misuse of Libraries:** Misuse of standard library functions.
- **User-Defined Bug Classes:** Checks for arbitrary bug classes can be implemented through the CodeSonar extension functions.