Master Thesis
Computer Science
Thesis no: MCS-2004:13
June 2004

Software Security Analysis

- Managing source code audit

Daniel Persson, Dejan Baca

Department of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Engineering at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science with a major in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author: Daniel Persson
Address: Karlskronavägen 11, 372 37 Ronneby
E-mail: daniel@dp.nu

Author: Dejan Baca
Address: Sångvägen 23, 284 37 Perstorp
E-mail: me@dejan.se

External advisor: Perolof Bengtsson
Ericsson AB
Address: Ölandsgatan 1-3, 371 31 Karlskrona
Phone: +46 455 39 50 00

University advisor: Bengt Carlsson
Department of Engineering

Department of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.tek.bth.se/
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Software users have become more conscious of security. More people have access to Internet and huge databases of security exploits. To make secure products, software developers must acknowledge this threat and take action. A first step is to perform a software security analysis. The software security analysis was performed using automatic auditing tools. An experimental environment was constructed to check if the findings were exploitable or not. Open source projects were used as reference to learn what patterns to search for. The results of the investigation show the differences in the automatic auditing tools used. Common types of security threats found in the product have been presented. Four different types of software security exploits have also been presented. The discussion presents the effectiveness of the automatic tools for auditing software. A comparison between the security in the examined product and the open source project Apache is presented. Furthermore, the incorporation of the software security analysis into the development process, and the results and cost of the security analysis is discussed. Finally some conclusions were drawn.

Keywords: Software security, audit, exploit, closed source, open source, buffer overflow.

CONTENTS

1 INTRODUCTION

1.1 Background

The increasing popularity of the Internet has made more people affected by deficient software security. Because of the poor security in today's software large groups of people and companies have been disturbed by malicious software [1], therefore increasing their costs. The increased cost caused by malware has caught the media's interest, resulting in more exposure on software insecurities.
The Internet has made a lot of information on software hacking publicly available. Because of this, a user does not need any deep knowledge of programming or security to exploit known security flaws. There are several hundreds of websites that supply users with security exploit programs. When a security exploit is found there are web pages and mailing lists that will report the security risk to thousands if not millions of readers.
With the increased cost, exposure and interest in security, the pressure on the developers to create secure software increases and at the same time the threats grow.

1.2 Securing software

The development of secure software is a hard task at best. It is very difficult for the developer to remember all known security problems and avoid them during programming. Adding strict deadlines and a schedule that is already falling behind and security does not seem that important for the programmer anymore. The security analysis of the product is also often conducted on the finished product after customers have asked hard security questions that need answers. Needless to say this make the task even harder [2].

To conduct a security test is not the same as conducting a function test. They both look for bugs but not of the same type. The people writing the software should not be part of the actual security auditing. They tend to read what they thought they wrote instead of what is actually written [3].

Code example:

```
char username[256];
readInput(&username[0],256);
databaseSubmit(&username[0]);
```

This function reads a username and writes it to the database. The function call readInput reads the username and prevents buffer overflows. The next function call sends the username to the database.
The code example is vulnerable to SQL-injection because it accepts all characters as input. With the input "Schneier" the function would work, but with the altered input "Schneier';drop database Main" the database would be deleted.

1.3 Differences in Closed and Open source

Projects in closed source differ a lot from open source projects. The main issue of closed source is to protect the software from being copied. If the software can be copied freely the closed source concept of licensing the use of copies for profit is lost.
In contrast to closed source, open source is released publicly and interested users perform audits and post corrections. The cost is often not an issue and many people voluntarily inspect the source code. Even the fact that many people have the source code available may make the developers take security in serious consideration

when they write the program, since it is possible that someone discovers a security flaw.

Closed source disadvantages:

- the source code in closed source get less exposure than open source code gets.
- The closed source project is limited to the security knowledge of the company or the individuals working with the project.
- A software security analysis cost time and resources and it can be very hard to convince management that the security analysis is worth the cost.

Open source disadvantages:

- There are no guarantees that open source products have sufficient exposure and examination.
- The quality and quantity of the security examination is determined by the interest of the project members and users.
- Harder to gain revenue from the use of the product

1.4 Research questions

- What security improvements can be found in a large closed source product? What types of threats are present in the product? Which are the most common types?
- How can a software security analysis be made part of the development process? What is the cost of a security analysis?
- What automated tools can be used and how effective are they?

1.5 Purpose

The thesis was written to examine the effectiveness of automated auditing tools. At the same time software security auditing had been performed on a mobile telecom server application. The thesis reports what security flaws have been found, how they can be avoided and how automated tools could be used to increase the level of security while developing new software.

1.6 The product

The study was performed on a corporate closed source product that has had several revisions and has existed for a few years. It is a leading product in its field and is used by several large companies. The software runs on a SPARC/Solaris architecture and is a server application. The server uses authentication and has a built-in web server that receives all input. The input is in XML format.

The product contains about 100.000 lines of source code and is written in C++. The developers have used, among others, the third party library rogue wave [4].

The product has had several function tests but until now no source code security analysis.

1.7 Limitations of this thesis

- No source code corrections have been performed. The focus of this work is code analysis and not on corrections. Examples will be given on source code improvements but there is no permission to implement the improvements or test them.
- Only stability problems caused by security risks in the source code have been addressed, this thesis covered security issues only.
- All possible security improvements were probably not found. The examination of the findings is restricted to the automatic audit tools and

the manual auditing of the source code. It was impossible to determine if all security improvements have been found.

- The source code auditing was performed in a C/C++ environment on SPARC architecture as follows from the product definition.
- No security analysis on the operating system has been performed. It did not fall within the scope for this thesis and has been performed by others.
- No general product risk analysis was preformed on the product. Others have already done a general risk analysis in the organization.

2 THEORY

2.1 Manual source code auditing

When the security analyst has access to the source code of the software, that should be examined instead of trying out attacks at random. This way of analyse is called white box. Reading and understanding the source code is a time consuming task. Even if the manual source code auditing does take a long time it will give valuable information about the software and any security risks. The time the auditing takes depends a lot on the analyst's knowledge in the programming language used and the system the software is used on. The purpose of a manual source code auditing is to reveal the security risks in the code. To find these security risks the analyst has to read what the design says the software should do and then examine if the source code gives room for other operations too. To perform an attack on a remote server the attacker has to send input to the software in order to exploit a security flaw. Therefore it is important to make sure the programs input interface does what it is supposed to do *and nothing more* [2].

2.2 Automatic source code auditing

To save time during a source code auditing there are automated tools that scan the source code and search for vulnerabilities. These automated tools use a database of keywords to find vulnerabilities and output a vulnerabilities report. The automated audit tools report a large number of *false positives* and therefore manual examinations to exclude the false positives are necessary. There are automated tools that perform an analysis on the vulnerabilities in an effort to minimize the amount of false positives. A real security risk often results in several warnings and not just one [2].

2.3 Common security risks and improvements

There are some common types of security issues that often arise in software. These types have been found on the examined product and are often reported on security sites.

2.3.1 Buffer overflow

A study that was presented in the NDSS 2000 conference [5], showed that as much as 50% of all large exploits were buffer overflows. Buffer overflows are made possible by unsafe programming languages like C and C++. The insecurity of C and C++ are due to their design for execution speed. To gain as much speed as possible the C and C++ programming languages do not perform any bounds checks on arrays or pointer references. Modern safe languages like Java are immune to buffer overflows at the expense of speed. A buffer overflow occurs when more data is written to a buffer then it has room for. The only way this is possible is if the programmer did not write bounds checks to confirm that the data will fit in the buffer. There are several dangerous function calls in the C programming language that can lead to a buffer overflow. These function calls overflow the buffer if no proper input validation has been performed [2].

Vulnerable C calls and replacements:

	Unsafe	**Safer**
	gets(buf);	fgets(buf, size, stdin);
	strcpy(destination, source);	strncpy(destination, source, size);
	sprintf(buf, format, source);	snprintf(buf, size, format, source);

When the input data overflows the buffer it continues to write to the buffer. This can cause the program to crash or behave strangely. Some buffer overflows might not even affect the system and therefore were not found during a functions test. The result of a random buffer overflow is often that the software crashes. Even if this is a devastating fact it is not as dangerous as the results of a more planned buffer overflow. If the buffer resides in the stack memory, the target of the overflow is the return address. Whenever execution finishes a function or a program itself, the CPU reads the return address and jumps to the location of the address [3]. This means that the program will continue running and execute the code that is written on the destination of the return address. An attacker can do two things; jump to other parts of the code that seem interesting or enter his own code in the buffer and then execute it by jumping to the start of the buffer [6].
The process to prevent a buffer overflow is very straightforward;

No data that is written to a buffer shall be allowed to be bigger then the buffer itself.

If the data is bigger, unsafe function like *sprintf* should be replaced by safe function like *snprintf* that truncates the data at a given length.

Code examples:

	Unsafe	**Safe**
	char buf[20]; sprintf(&buf,“%s”,&bigbuffer);	const int length = 20; char buf[length]; snprintf(&buf,length,“%s”,&bigbuffer);

The problem with sprintf is that the function can overflow buf and then overwrite the return address. The solution is to use snprintf and specify the length n of the buffer.
In the C language there are more function calls that are safe even if the data is larger then the buffer. These functions truncate the data to fit the buffer. These safer C function calls shall be used whenever possible to increase the security level of the program [7].

2.3.2 Denial of Service

Denial of Service (DoS) is the name of an attack that influences the availability of a system or network. The attack is conducted through a flood of requests or an incorrect error handling in the software. If the software itself contains bugs, then the bugs can be used to crash the software or create exceptions that would render the software unusable. Another way of creating a DoS attack is to generate so many processes that the process id numbers will be out of limit. Then the system cannot start more processes and cannot serve more clients. The result of a successful denial of service attack is total shutdown of the service the application is meant to provide [3].

2.3.3 Misplaced trust and input validation

When developing new software there is always the question of whom to trust and what input to consider safe. The software developers have to make several decisions about trust and often they are not even aware that they have made any decisions. Misplaced trusts create many security risks and are often the base of exploitable code. Most misplaced trusts can be found in the input validation. Often, input data from in-house software are considered by the developers as safe. Therefore, no input validation is performed and exploitable code is created. The reason why many developers trust their own software is because the software is compiled and closed source. With programs like IDApro [8] that help reverse engineer binaries, an attacker can recreate the design of the software and find any hidden algorithms. The misplaced trust in machine-code binaries often result in developers trying to hide secrets inside the code, e.g. cryptographic keys and other

in-house authentications [2]. Because a client cannot be trusted, all its input must be validated before being handled by any server application, especially if the server accepts input from all sources.
Conducting a good input validation can be very hard. The input validation shall always validate that the input follows a set of rules and never try to find illegal input [9]. It is easier to define secure and valid input then to find all forms of insecure input. All input that does not pass the input validation should be discarded [7].

2.3.4 Race condition

In a multi thread environment there is always a chance of a race condition. Because of multi threading, the program can perform the same or different tasks at the same time. The trouble occurs when these threads depend on the same resource or variable. Most race conditions are not security vulnerabilities but instead stability issues. Race conditions are becoming a bigger problem and they are often hard to correct even after they have been found. In a multi thread program it can be very hard to avoid all race conditions, therefore it is better to use single threading whenever possible. To avoid race conditions the developer has to remove the time between verifying a resource and using it. Even in microscopic timeframes it can be exploited [2].

2.3.5 Random number generator

A simple thing like generating a random number is a daunting task for a computer. It is easy to forget that something that easy is hard to accomplish during programming. The result is often that developers use non-true random calls like *random()*. Some developers might even think that *random()* does give a true random value and others might think that with a time seed the value will be random. But all of these results can be predicted. A call to *random()* follows a pattern and seeds can be predicted [2].
There are several situations were a real random number is needed. From a security point of view, random numbers are important when dealing with cryptographic. Without a good random number generator the cryptographic can be predicted and broken. Therefore it is important to gather a lot of information that can be used as data for generating random number, this is called entropy. Sufficient entropy is necessary to create a good seed for the *random()* call. Most UNIX systems have their own built in random number generator (*/dev/random*). If the built in generator creates enough bits of random numbers then it should be used. If more bits are needed a different random number generator has to be created [7].

3 METHODOLOGY

3.1 Auditing method

To find possible security improvements we adopted a white box method and examined the source code. Software security expert John Viega [2] suggests in his book to use white box methods. White box auditing yields more results when searching for security issues. Furthermore, we assumed it was more efficient work when the testers have the required documentation of the project. We have conducted practical experiments on the product where we confirmed the security warnings.

3.2 Auditing source code

We have performed both automatic and manual source code auditing. The possible exploits that appeared in the auditing were confirmed or dismissed in our practical test environment. The test system was built on Sun Ultra SPARC with Solaris 8, all according to the product specification.

3.2.1 Automatic auditing

The automatic auditing of the source code was performed with RATS, a tool for automatic auditing. This tool gave us a rough image of the security state.
RATS was chosen as our primary auditing tool because it had a large database with vulnerabilities. To confirm the findings of RATS, the tools Flawfinder and ITS4 was used. Any differences between the tools' findings were noted.
The automated auditing tools reported how likely a warning really is a security threat and not a false positive. The different tools have different configurable levels. We used the lowest level where all warnings were reported and we examined all the unique warnings. With unique warning it is meant non-duplicates of the warnings. To remove the false positive warnings we examined each warning and determined if it could pose a threat. All the warnings that were found during the manual examination were labelled as possible security improvements.

3.2.2 Flawfinder

Flawfinder was written by the author of "Secure Programming for Linux and Unix HOWTO", David Wheeler. Flawfinder scans C/C++ code and has a database of 128 C/C++ vulnerabilities. Flawfinder has 6 levels of reporting warnings, level 0 to 5, where level 5 only shows the most dangerous warnings. Standard level in Flawfinder is level 1. Flawfinder version 1.24 was used [10].

3.2.3 ITS4

Cigital developed ITS4, "It's The Software Stupid! Security Scanner". ITS4 scans C/C++ code and its database has 144 vulnerabilities. ITS4 has 6 levels of reporting warnings, level 0 to 5 where level 5 only showed the most dangerous warnings. Standard level in ITS4 is level 2. Version 1.1.1 of ITS4 was used [11].

3.2.4 RATS

RATS, "Rough Automatic Tool for Security". Secure Software, Inc developed RATS and some of the programmers of RATS have also worked with ITS4. RATS scans C/C++, Python, PHP and Perl source code. Rats has 3 levels of reporting warnings, Low, Medium and High where High shows the most dangerous warnings. The standard level is Medium. Version 2.1 was used and it has a database with 334 C/C++ vulnerabilities [12].

3.2.5 Manual auditing

We have performed a manual audit on some components of the product. The combined size of these components represents 10% of the overall source code used in the product. The purpose of the manual audit was to examine the effectiveness of the automated auditing tools.

3.3 Security categories

Initially all warnings that were found were called *Software Security Warning* (SSW). The purpose of the SSW category was to keep track of the warnings found in the automatic and manual audit.
Confirmed threats and possible security issues were examined and categorized. The outcome was divided into the following categories:

1. *False Positive* (FP). These are safe and were generated by the automated auditing tool. They are no risk at all.
2. *False Negative* (FN). These are risks but were not reported by the automated auditing tool. These types of risks could be found by manual audit or when analysis the differences between the tools.
3. *Possible Security Improvement* (PSI). A possible security risk without confirmed consequence.
4. *Security Risk with Consequence* (SRC). A security risk that had a consequence and was confirmed in the test environment. A proof of concept may have been created but not published.

The purpose with these categories was to put the different types of threats in order and take more care of the most risky threats. These categories also showed how in the beginning a harmless programming error turned into a full-grown exploit.

3.4 Practical experiments

The practical experiments were performed on a test system with the latest stable version of the product. The test system was used to confirm the possible security improvements and determine their consequence. Some of the confirmed security risks were also selected for more complex proof of concept programs. The confirmed security risks found were exploited as much as possible. These proof of concepts were important to strengthen our arguments and show the consequence of poor security. All of our tests were performed as users of the server application without any shell access.

3.5 Open source security reports

The open source community has an immense resource of security reports, source code reviews and bug reports. By looking at similar open source programs that have had bugs that resemble the warnings we have found, we got an idea what to look for.

3.6 Risk evaluation

To manage the warnings from the automatic auditing tool, a risk evaluation had to be performed and the warnings put in different categories. It was necessary so the findings could be traced from warnings to improvement possibilities. The automatic auditing tools generated a large set of warnings but many of these were excluded by studying patterns in the source code.

The main issue of the risk evaluation was to focus on the most crucial risks. When it is time to incorporate the security improvements into the source code there can be a discussion on how much time should be spent on programming on a product that already works. Some decide to implement corrections on all confirmed

security risks while other like the OpenBSD [13] project decide to fix even the smallest security warning. The result of the OpenBSD project's hard policy has created what today is considered one of the most secure operating systems available. Often, security flaws in other BSD distributions have already been patched in OpenBSD without knowing it was a security risk [3].

3.7 Bug tracking

All security warnings that were confirmed in the practical experiment were re-examined to be certain no new or further exploitation was possible. The security warnings were also examined why they have occurred, how they could have been avoided and what kind of threats they pose. This re-examination was used to create a guideline for how the threats could be avoided in the future and how to avoid insecurities while developing new software.

3.8 Third party libraries

The product used a third party library that introduced secure strings that protected the buffer and stopped buffer overflows. Because of this string protection most insecure buffer handlings was not a threat. While the third party library made the development of the product easier and at the same time more secure it also added extra elements the developers had to be aware of while programming the software. The data that is copied between the different libraries has to be validated or copied with secure functions. Overall in the product the third party library had defused several security issues and made the software more secure, but it was not an adequate replacement of input validation and secure functions. The security of the third party library itself could not be determined.

4 INVESTIGATION

The target software we examined only received requests from the network interface and no user was allowed access to the server itself. Nevertheless, we still have reported shell based security risk where the attack would need access to a shell.

4.1 Results – automated tools

The different tools showed a mixed list of warnings depending on what level they were configured to report. The number of warnings per level varied a great deal depending on the tool. The total number of warnings also varied depending on the tool. The tables below show the results from the automated tools at the lowest level where every warning was reported, the default level and the highest level where only the most dangerous warnings should be reported.

- Flawfinder was the tool that showed the highest number of warnings at the standard level, in fact it showed all the warnings it could find.
- At the lowest possible level where all warnings were reported, Flawfinder found one unique warning that neither RATS nor ITS4 had found.
- At the highest level, Flawfinder did not report a single warning. Flawfinder did not find any insecurity that where dangerous enough to be label at the highest level. Most of Flawfinder's warnings were at level one and two.
- Many of the warnings Flawfinder reported on standard level, the other tools reported only at the lowest level.

Flawfinder warnings	Lowest	Standard	Highest
Not present in ITS4	213	414	0
Not present in RATS	200	326	0
All warnings	587	587	0
Unique warnings	1	313	0

- ITS4 had the lowest overall report count of warnings.
- ITS4 also reported the largest number of unique warnings at the lowest level.
- At the highest-level, ITS4 reported 36 warnings and no other tool reported these 36 warnings at that level.
- All the warnings reported at the highest level in ITS4 were reported by the other tools at a lower level of danger.
- At standard level, ITS4 and RATS had a similar number of warnings but they where not the same type of warnings.
- The perception of dangerous warnings differs in ITS4 compare to Flawfinder and RATS.

ITS4 warnings	Lowest	Standard	Highest
Not present in Flawfinder	144	137	36
Not present in RATS	297	174	36
All warnings	504	310	36
Unique warnings	112	133	36

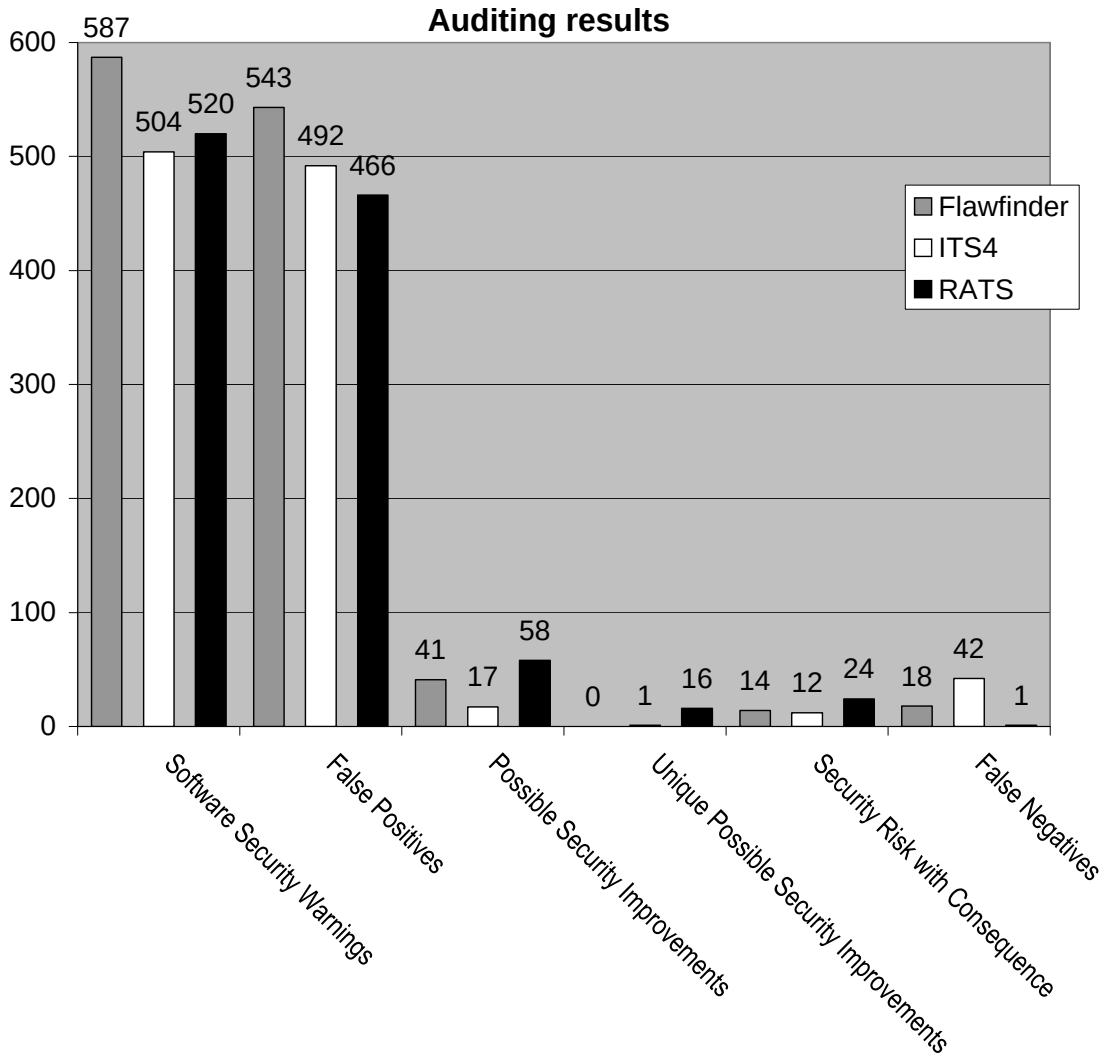
- RATS was the tool with the least configurable levels of warnings. As a result RATS had the largest number of warnings at the highest level.
- Even at the highest level 242 warnings were reported by the tool.

- All the warnings at the highest level were also unique when comparing with the other tools at the highest level.
- At standard level, RATS reported a similar number of warnings as ITS4 but the type of warnings corresponded more with the warnings from Flawfinder.
- RATS reported altogether 520 warnings and from these 101 were unique.
- The biggest variation was with the tool ITS4.

RATS warnings	Lowest	Standard	Highest
Not present in Flawfinder	133	45	242
Not present in ITS4	313	170	242
All warnings	520	306	242
Unique warnings	101	41	242

4.1.1 Manual examination of the automated tools findings

From the reports of warnings that the three tools reported, a manual examination determined if the warnings were possible security improvements or if they were false positives. Together the tools reported 823 unique warnings. The manual examination of these warnings was a very time consuming task. The outcome from the manual examination was 59 possible security warnings. Resulting in that 7.2% of the warnings were real security issue or a part of a real security issue. The different tools had mixed results of the numbers of possible security improvements they did find and how many they did not find (False Negative). RATS had the best results with just one false negative, this false negative was found by ITS4. The worse false negatives result came from ITS4. ITS4 found one unique possible security improvement from a total of 17 unique possible security improvements. ITS4 had the largest number of unique warning but at the same time only one unique possible security improvement. This showed that ITS4 had a high number of false positives that no other tool had reported. RATS that had the second highest number of unique warnings had the highest number of unique possible security improvement and at the same time the highest number of warnings that where real threats to security. Flawfinder had a better result than ITS4 but because of it lack of unique findings it did not give rise to any new results. When the tools were used at their highest level Flawfinder did not find any warnings at all. ITS4 found some warning but they were all false positives. RATS was the only tool that found possible security improvements at its highest level. 80% of all the possible security improvements where found by RATS at Maximum level.



4.1.2 Manual auditing

During the automated auditing there was a need to perform manual auditing on all the warnings that where reported. Many of the warnings were just point of insecurity that had to be examined if any insecure action was performed with them. To verify that the automated tools had found most of the insecurities, we performed a manual auditing on whole components and not just on the warnings. During this manual component audit we did not find any new security risks. The security risks that were found in the manual audit had already been reported by the automatic tools. Therefore, all the security issues that where found in the manual audit had already been found in the warnings by the automated tools and the manual examination of these warnings.

4.1.3 Automated tools false negative

All automated tools checked had false negatives. These risks were not present in the vulnerabilities database or the tools where not capable to perform the analysis that where required to find these false negatives. The tool with the lowest number of false negatives was RATS. Its false negatives were caused by missing vulnerabilities in its database. If the vulnerability was added to the database then RATS reported zero false negatives, according to this investigation. An email with the vulnerability was sent to Secure Software, Inc [12] and they responded that they would enter it in to the database. The false negatives in flawfinder where also

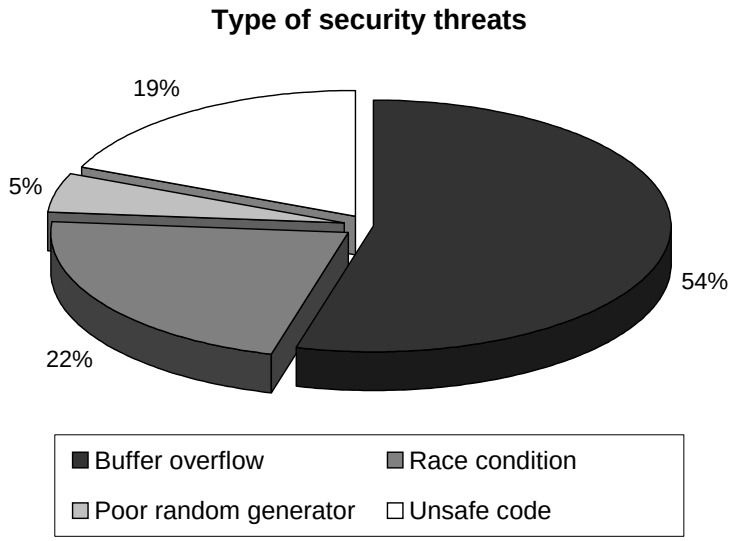
missing vulnerabilities in the database but some of them were vulnerabilities that where in the database but flawfinder did not report them. The false negatives from ITS4 were also missing vulnerabilities but the larger portions were fixed local buffers that were not used properly. ITS4 was the only tool that did not report them. ITS4 also had trouble examining the code like Flawfinder. ITS4 failed to find insecure function calls like 'return crypt(password, key)'.

4.2 Results – application

The security analysis shows that there existed one possible security improvement per 1700 lines of source code. For every 4000 lines of source code there existed one security risk that can be exploited.

Possible Security Improvements	Security Risk with Consequence
59	25 (42% of PSI)

The security risks in the code were categorized into four groups: buffer overflow, race condition, poor random generators and unsafe code. Security risks with unsafe buffer handling were placed in the buffer overflow group. The poor random generators group were security issues that used a predictable random generator in a security function. The race conditions were security risks where an attacker would be able to alter the information the software was going to use. Unsafe code was a group of security risks that could not be placed as a special type.



4.2.1 Security risks

These security risks were found in the product. All source code examples have been altered to support anonymity but the end result of the source code is still the same.

Buffer overflow and denial of service

Incorrect buffer sizes that could create uncontrollable buffer overflows.

Source:

```
int theVersionNr;
char *tmpVersionNr = new char[1];
sprintf( tmpVersionNr, %i , theVersionNr);
```

This operation would have been safe if theVersionNr could not receive a value greater then nine (10 creating two characters not one). If the value would have been greater then nine the extra number would overflow parts of the stack buffer

and rewrite other variables. This could have devastating effects on the stability of the program or no effect whatsoever. The effect would depend on what variables are overwritten. To counter this security risk the developers would have to make sure the buffer could hold the maximum size from the input or have used snprintf and truncated the input to fit the buffer.

Buffer overflow and input validation
A misplaced input validation combined with a buffer overflow would create a dangerous security issue.

Source:	

	void newuser(){
	if (strlen(username) < 9)
	…
	}
	
	void login(){
	char LogInfo[512];
	if (!authorized(username, passwd)
	//if authorized fails report the login attempt to the log.
	sprintf(LogInfo, “user: %s pass: %s”, &username[0], &passwd[0]);

The input validation for usernames and passwords resides in the wrong function and not in the function that receives input, therefore it is not an input validation at all. The length of the username and password is not were it should be and when unauthorized users try to login. The login attempt is sent to the log. This creates a buffer overflow that can be controlled. Resulting in full control to the attacker. To prevent this there has to be a real input validation or the buffer has to be copied with a safe function like snprintf and not with sprintf.

Poor random generator
Below follows some examples of random generators that creates predictable random numbers.

Source:	
	rand, drand48, erand48, lrand48, nrand48

All the standard ANSI C random functions create predictable random numbers if no adequate seed is used. If the poor random number has been used for important matters then the level of security has decreased. The use of /dev/random creates a better random number.

Race condition and poor random generator
Insecure temp file handling that create security exploits on the operating system.

Source:	

	char *aTempFile
	aTempFile = randomname(“/tmp”, “temp”);
	int aFD = creat(aTempFile, 0666);
	…
	unlink(aTempFile);

The randomname() function described in the source code uses the predictable rand function mentioned above. If the filename could be predicted then it would be possible to create links to other files and redirect all action performed to the tempfile. The source code showed that an attacker would be able to delete files. The permission 0666, according to the source code, gives every user on the system permissions to alter and read the file. The temporary files should have a true

random name and their permission should not allow anyone but the owner of the file to read or write to it.

Unsafe code
Incorrect programming could create unsafe code.

Source:	

	int MaxPasswordLength=20;
	RWCString aKey = thePlainTextPW(0,2);
	return crypt(thePlainTextPW, akey);

This security risk arises from the incorrect use of crypt. At first, crypt only supported eight characters and therefore the last 12 were discarded and not used when authorizing a user. The salt used was the two first letters in the password. Therefore if an attacker acquired the encrypted password only six letters were left to decrypt. The password should be limited to the limitation of crypt. The salt to the function crypt should have been a true random character and not a part of the password.

4.2.2 Proof of concepts

Three security risks were chosen and proof of concept programs were written to show the effect of the security risks. The three security risks were buffer overflows, race conditions and poor random generators. The three proof of concept programs had different goals in their attack, sieging the system, sabotage and espionage.

Sieging the system with a buffer overflow.
The buffer overflow created a possibility to crash the program and create a denial of service attack. The buffer overflow also created the possibility of remote shell access, a far more dangerous risk. The proof of concept program did create a remote shell. By overwriting the return address, the software was redirected to the injected machine code and a shell was bound to a specific port. With this shell access an attacker would have had access to the system and gateway into the rest of the systems in the network. Because of the dangerous results from buffer overflows they were considered the most severe security risks.

Source:	
	sprintf(LogInfo, “user: %s pass: %s”, &username[0], &passwd[0]); Login: <Shell code> Password: AAAAA(512-shell code characters)AA(return address)

Sabotaging the system with a race condition.
The race condition made possible for an attacker to damage the system and more important, the ability to cover his tracks. The proof of concept program exploited a race condition present in the product to fool the software to overwrite and delete log files. Because it is the software itself that destroys the log files, an Intrusion Detection System (IDS) would not detect the intrusion.

Source:	
	int aFD = creat(aTempFile, 0666); # ln –s aTempFile /var/log/importantLog

Conducting espionage on the system due to poor random generators.
The poor random generator created the possibility to break the encryption. If the encryption could be broken then an attacker could read any information that was important enough to encrypt. The proof of concept resulted in a decryption and encryption tool that could have been used to create whatever encrypted data the attacker would want. The exploit also made it possible to read other customers encrypted data and change them.

Source:	
	int Crypto::getRandomDigit () { double tmp = drand48(); # cryptcracker Encrypted.file

5 DISCUSSION

5.1 The tools

The automated tools were not effective in finding security risks. Instead they showed where the manual auditing should be focused. The tools saved time and lessened the burden on those performing the security analysis. Looking at the areas of the product that historically has had most security flaws can increase the effectiveness of the automated tools. Kanta Jiwnani and Marvin Zelkowitz mention in their article [14] what parts of a product often is responsible for security risks. Using their study, and with this study, the number of false positives that have to be examined could be reduced. The developers of the auditing tools have had different ideas of what was considered a major threat and what was a less likely threat. Therefore, it is important to use the tools at the right level. When comparing the tools' different levels, RATS had the smallest configurations and as a consequence reported the most warnings at the highest level. But RATS was also the only tool that found possible security improvements with the highest level. For ITS4 and Flawfinder, the highest level did not contribute to the security analysis. When looking at the standard level, RATS found 91.5% of all the possible security improvements with just 3/8 of the security warnings all the tools had together. RATS was the tool with the least false negatives and the most unique possible security improvements. All the tools had false negatives, even if the false negatives were just a missing vulnerability in the database, the fact remains that the tools will not report every security danger in the code. Therefore the effectiveness of the tools cannot be considered 100%. The tools also reported a variety of false positives. ITS4 had the worst ratio of false positives with a high number of unique false positives. When looking at all the warnings from all the three tools, the manual audit did not find any security risk that the tools had not already reported. With the tools ITS4 and RATS, using the lowest level, all the warnings we could find would be reported but the total number of warnings would be much larger then just one tool at standard level.

5.2 The application

According to an article in "*The Wall Street journal"* [15] source code contains one bug per 1.000 lines of code and every one of them a potential security risk. The audit of the product showed one possible security improvement per 1700 lines of source code. The application also used third party libraries that help increase the level of security. The third party library had increased the level of security in the product. At the same time it had created new security issues. The string handling was safe while it used the third party library but it became very insecure when copied to a standard ANSI C variable. The copying of data between the third party library and the standard ANSI C libraries created several security risks and was directly responsible for several exploitable security issues. The third party libraries are small and specialised in specific tasks and therefore they should be secure, but no source code security audit on the third party library was possible to perform. Looking at the general security and the usage of the third party library the product had good security for software that has not had any security analysis.

Comparing the application with the open source project Apache [16] showed some interesting figures. An automated security audit was performed in 2002 by an open source community user [17] with the tool RATS. The audit was performed on Apache version 1.3.26 and it contained about 110.000 lines of code. The apache software also handles more user input than this product does. In the apache audit the analysis found 6 risks and exploits. These risks and exploits consisted of 12

warnings from RATS. The apache project has had the benefit of a large user base that has examined the code and made source code corrections. The target product has been restricted to the time available to the developers. Comparing the two products, the application had after several years of use and bug fixes a total of 59 warnings that where related to security risks while the apache project had, after several years of use and open source bug reports, 12 warnings related to security risks. Even if the open source analysis is not as structured and performed as accurately as a corporate study, the difference in the number of security risks is large. The closed source product that handled less input than the open source project still had six times more security risks. When looking at the different threats the open source project also had buffer overflows and race conditions but it did not have any poor random generators. In the examined product buffer overflows were the most common security threats. The buffer overflows were also responsible for the most dangerous security exploits.

5.3 Incorporating security

Manage software security from the beginning of a project the developers need to have large knowledge of all known possible security risks. The developers must have knowledge about all available ways to exploit software products. Furthermore, the developers have to keep track of every aspect of the software they develop. These are not reasonable goals, unless the project has an unlimited budget and no timeframe. In open source projects the developers use the eyes of their users to discover security issues they might have created.

Closed source developers do not have the same exposure of source code that open source have since closed source want to protect the code. At best, closed source developers have a security team or person that examines the code. To replace the eyes of the open source community, the closed source developers can use automated auditing tools. The automated tools generate warnings where insecure code exist and the developer can then examine the warnings and correct the code or determine the warnings as false positives. If this is performed during the development phase there will not be a great number of warnings to examine. The developer also learns from his warnings and can avoid insecure code. Because the automated tools are fast these tests could be performed every day and any new warnings sent to the developers for examination. The developer that added the code has to be the one to determine if it is a security risk or not. He has to prove it is not dangerous or if he cannot it shall be considered a security risk and the source code altered. The cost of incorporating security in an existing process is possible to determine. The cost depends on size of the source code and which tools used. Calculations indicate that when securing software of 100.000 lines of code with 823 warnings, it costs 42 workdays including analysis and report every warnings. The investigation showed difference in the automatic auditing tools. According to the investigation, RATS found 91.5% of all possible security improvements at standard level. The examination of RATS at standard level would take 16 workdays. At standard level, RATS had found the dangerous buffer overflow threats. Buffer overflows are also easy to secure when they have been found. To examine the remaining warnings RATS had missed at standard level would cost another 26 workdays. Furthermore, if more examinations would be performed on the source code it is possible that even more security risks could be found at a greater cost than the 42 workdays. With large closed source projects, the development of new source code could be too expensive for a full source code security analysis. With very large projects, even a good enough result could be too hard to achieve with the sparse resources spent on security. To find all security risks could be impossible to achieve within a reasonable timeframe.

6 CONCLUSIONS

In the application there are six times as much security risks than in an open source project of same size. The open source has even more intensive user inputs than the application. The result of the investigation showed that the application contained one possible security improvement every 1700 lines of code. Furthermore, there existed one security improvement that could be exploited every 4000 lines of code. Quite half of the found threats in the product were buffer overflows and according to a study that was presented in NDSS 2000 conference [5] half of all large exploits were buffer overflows. Fifth parts each of the threats were on race condition and unsafe code. The remaining 5% of the threats were poor random generators.

The automatic tools reported a high number of false positives and the real warnings were often part of a larger security threat. Therefore all warnings had to be manual examined. When we compared the different level of the tools it was found that RATS standard level produced the most efficient result. When comparing Flawfinder, ITS4 and RATS we found that RATS produced the most effective report of security warnings. The result of RATS findings was 91.5% of all real security threats.

Incorporate security into software development could be a proportionately small cost when comparing the development time for the whole product. The analysis with RATS in standard level took 16 workdays and the tool caught 91.5% of the security issues. The security threats included in RATS standard level were the most dangerous risks. It is possible to reach good enough security, according to us, within 16 workdays, which is the only additional cost to acquire the risks from the warnings.

Functions tests do not increase the level of security of an application to a good enough result. Therefore, closed source applications that have relied on functions tests are exploitable or at least vulnerable to security related attacks. A security analysis on the closed source application could create good enough security but for every 4000 lines of code added to the application, a new exploit would be created. Thereby requiring a new security analysis.
For the security analysis to find all security risks, 150% more time has to be spent when comparing to the good enough result. With a new security risk per 1700 lines of code the need for more frequent security analysis increases.
With the higher cost for a full security analysis and the introduction of new security risks with new code a large closed source project that relies on its own developers for security might never be safe.

7 REFERENCES

1. The Word Spy (2004), URL http://www.wordspy.com/words/malware.asp
2. Viega, J. and McGraw G. (2001), Building Secure Software, Indianapolis, IN: Addison-Wesley, ISBN: 020172152X
3. Russell R. and Cunningham S. (2000), Hack Proofing Your Network: Internet Tradecraft, Rockland, MA: Syngress Media, ISBN: 1-928994-15-6
4. Rouge Wave (2004), URL http://www.roguewave.com/
5. Wagner D., Froster J., Brewer E.and Aiken A (2000), A first step toward automated detection of buffer over-run vulnerabilities, In Proceedings of the year 2000 Network and Distributed system Security Symposium (NDSS), San Diego, CA.
6. Aleph1 (1996), Smashing the Stack for Fun and Profit [Online], Phrack Magazine 49-14, URL http://www.phrack.org/phrack/49/P49-14.
7. Viega J. and Messier M. (2003), Secure Programming Cookbook for C and C++, Sebastopol, CA: O'Reilly & Associates, ISBN: 0-596-00394-3
8. IDApro (2004), URL http://www.datarescue.com/idabase/ida.htm
9. Wheeler D., Secure Programming for Linux and Unix HOWTO. v3.010. (2003, March), URL http://www.dwheeler.com/secure-programs
10. Flawfinder (2004), URL http://www.dwheeler.com/flawfinder/
11. ITS4 (2004), URL http://www.cigital.com/its4/
12. Rough Auditing Tool for Security (2004), URL http://www.securesw.com/rats
13. OpenBSD (2004), URL http://www.openbsd.org
14. Jiwnani K. and Zelkowitz M. (2004, April), “Susceptibility Matrix: A New Aid to Software Auditing” [Online], URL http://www.cs.umd.edu/~kanta/ieee.pdf
15. Wingfield N., Mangalindan M., Swisher K., Bank D., Hamliton D. and Clark D. (2004, Feburary 09), “Safety, Blogs and Protocols” [Online], URL http://online.wsj.com/article_print/0,,SB107582325268319257,00.html
16. The Apache Software Foundation (2004), URL http://www.apache.org
17. Audit for apache 1.3.26 (2002, October 04), URL http://www.sardonix.org/audit/apache-45.html