Software Security Analysis - Execution Phase Audit

Bengt Carlsson[*] and Dejan Baca[#]
[*] *School of Engineering, Blekinge Institute of Technology ; PO Box 520, S-372 25 Ronneby, SWEDEN;*
bengt.carlsson;@bth.se
[#] *Ericsson AB*
P.O. Box 518, S-371 33 Karlskrona, SWEDEN;
dejan.baca@ericsson.com

Abstract

Code revision of a leading telecom product was performed, combining manual audit and static analysis tools. On average, one exploitable vulnerability was found for every 4000 lines of code. Half of the located threats in the product were buffer overflows followed by race condition, misplaced trust, and poor random generators. Static analysis tools were used to speed up the revision process and to integrate security tests into the overall project process. The discussion analyses the effectiveness of automatic tools for auditing software. Furthermore, the incorporation of the software security analysis into the development process, and the results and costs of the security analysis is discussed. From the initial 42 workdays used for finding all vulnerabilities, approximately 16 days were needed for finding and correcting 91,5 % of the vulnerabilities. So, proportionally small investments improve the program code security by integrating an automatic auditing tool into the ordinary execution of source code revision.

1. Introduction

During recent years software developers have changed focus from only reliability measurement to include aspects of security threats and risks. Both security tests and function tests look for weaknesses but not necessarily at the same time. By definition a reliability threat or test will sooner or later manifest itself, while a security threat may remain undetected. Recently static analysis tools have been used to automate source-code security analysis by measuring the amount of weaknesses or vulnerabilities at hand. As a result, improved "error-free" code for future versions of the application may be developed.

Manual audit done by experienced programmers is a time consuming but otherwise efficient method for conducting secure code revision of software. The main reasons for introducing automated auditing tools are to decrease manual audit time and to integrate automatic tools as part of a revision update.

The first issue has its background in program checkers [8] followed by several generations of automated auditing tools, from "rule" based to more flexible context based. Static analysis tools use a database of keywords to find vulnerabilities and output a vulnerability report by doing a syntactic matching [14]. These tools report a large number of false positives since they lack a deepened context analysis, and therefore manual examinations to exclude the false positives are necessary. More recent global analysis tools perform an analysis of program semantics (for an overview see [4]) in an effort to minimize the amount of false positives.

The second issue is the possibility to integrate automated tools into the software development life cycle as improvement and effectiveness factors. Development cost savings are part of a more general return on investment (ROI) calculation for the investigated project. ROI involves factors for identifying the context of the life cycle, techniques for setting up the measurement, the actual requirement and comparison of the software, and finally the set of measures involved. So, the obvious calculation of number of bug fixes should be supplemented by calculating revision time, to complete the development cost discussion.

This study discusses what sort of security flaws that have been found, how they can be avoided and how to use automated tools to increase the level of security during software revision. In Section 2, different security

exploits are described and in Section 3 the product and its development process is explained. The investigation, described in Section 4, uses different automated auditing tools combined with manual auditing applied on a real telecom application. The results, in Section 5, compare the effectiveness of different automated auditing tools applied on the mobile telecom server software. The development cost is discussed as an improvement on the original investigation. In the discussion section open and close source software are compared followed by a conclusion.

2. Securing unsecured software

Like most commercial applications, the code investigated is a closed source software project. The main issue in general is to protect the software from being copied, consequently it is difficult for an outsider to find and correct security vulnerabilities. In contrast to close source, open source is released publicly and interested users can perform audits and post corrections.

Information about closed source security bug-tests is insufficiently discussed in the literature whereas classification of various security taxonomies is abundantly discussed (see Mc Graw [9] for an overview).

Some security vulnerabilities occur repeatedly on both the examined application and when exploring different security sites. Recent software auditing of security vulnerabilities in Windows NT with a susceptibility matrix classification is presented by Jiwnani and Zelkowitz [7]. Unlike this investigation, the result presented here is based on a limited subset of all detectable security flaws.

Four common types of security flaws that often arise in software are described below.

- Buffer overflow - a study presented by Wagner et al. [14], showed that as much as 50% of all large exploits were buffer overflows. Buffer overflows occur when unsafe programming languages like C and C++ are used. The insecurity of C and C++ are a consequence of not performing any bound checks on arrays or pointer references. A buffer overflow occurs when the amount of data exceeds the size of the buffer. There is a lack of bound checks to confirm that the data will fit in the buffer. This can result in several dangerous function calls in the C programming language and lead to a buffer overflow.
- Misplaced trust - when developing new software there is always the question of whom to trust and what input to consider safe. Misplaced trust creates many security risks and is often the base of exploitable code. During the investigation most of the misplaced trusts was found in the input validation. Often, the developers consider input data from in-house software as safe and therefore no input validation is performed. They may try to hide information inside the code, e.g. cryptographic keys and other in-house authentications [13]. With programs like IDApro [6] that helps reverse engineer binaries, an attacker can recreate the design of the software and find any hidden algorithms. Placing the trust in client software and not on the server side also creates a security issue, i.e. it is possible for the end user to circumvent the client and any client validation.
- Race condition - in a multi thread environment there is always a probability of a race condition. Because of multi threading, the program can perform the same or different tasks at the same time. 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. To avoid race conditions the developer has to remove the time between verifying a resource and using it. Even in microscopic timeframes a race condition can be exploited [13].
- Poor random number generator - a simple routine like generating a random number is a daunting task for a computer. Developers use non-true random calls where the results can be predicted. From a security point of view, random numbers are important when dealing with encryption. Without a good random number the cryptography can be predicted and broken. Most UNIX systems have their own built in random number generator.

3. Product and development process

The results of this paper where obtained through an investigation at a software development department at Ericsson AB. The most recent stable version of a telecom product was used as a case study. The examined product had a component-based architecture and was built on a shared platform, which allows use and reuse of the same component architecture. Only the C++ components where examined during the investigation but the product also used HTML and Java for GUI. For communication the product and the components used a common socket connection interface and all the data was transmitted in XML format. The product runs on a SPARC/Solaris 8 platform, a server application com-

posed of approximately 100000 lines of code. During every production cycle the product passes several tests but no source code security analysis had been undertaken. Figure 1 describes the principal steps within the investigated project.

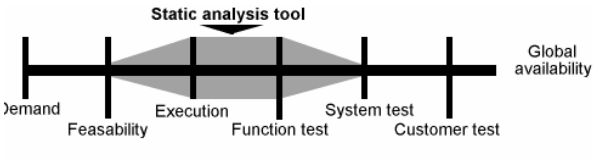

Figure 1 Program progress and security analysis

The development department had a project cycle of 1-1.5 years with an average of 50 participants. After an initial demand the design structures were decided. If security gaps, as a result of design errors, were undetected no later analysis would typically capture these deficiencies. After the design (feasibility) was completed, the product was divided into different function parts. Execution and function tests were then done before being reunited at the system level again. No security specific tests had been done through the development process. During every phase of the development cycle measures were taken to detect faults in the code, e.g. basic tests during the execution and automated tests, TTCN [11], during function and system test. Damm et al. [5] performed a case study at the same development department and concluded that a fault had almost a twenty fold increase in cost if it was found during system tests instead of the execution test. Since neither function nor system test had any specific security tests, any security flaw introduced during the execution would likely pass unnoticed, e.g. it will not be detected until the product had ended its development cycle.

4. The investigation

During the investigation both an automatic and a manual audit of the source code were performed. The static analysis tools were used in the execution part as pure code revision tools. The possible exploits that appeared in the audit were confirmed or dismissed in the practical test environment. The proof of concept investigation used the compiled code at the system level to show the effect of the security risks.

The purpose of the manual source code auditing was to reveal the security risks in the code. Reading and understanding the source code was time demanding and relied on the analyst's knowledge in the programming language and the system the software is used on. In order to find these security vulnerabilities the analyst had to understand the source code and make sure it did what it was designed for and nothing more.

Since no prior security test had been undertaken, the automatic auditing tools were used to examine vulnerabilities from a large database rather than performing a semantics analysis. The comprehensive task was to find methods and rules for integrating security analysis into program revision on closed source software. The automatic auditing of the source code was performed with RATS, a tool for static analysis. To confirm the findings of RATS, the tools Flawfinder and ITS4 were used. Any differences between the tools' findings were noted.

Confirmed threats and possible security issues were examined and categorized. The outcome was divided into the following categories:

1. False positives. These warnings, generated by the automated auditing tool, do not pose any risk at all.
2. False negative. All vulnerabilities, found during the manual auditing, not reported by the automated auditing tool.
3. Possible security improvement. A possible security risk without confirmed consequence, i.e. the code is not properly written but other parts of the program will take care of the vulnerability.
4. Security risk with consequence. A security risk that had a consequence and was confirmed in the test environment.

The purpose with these categories was to separate different types of threats, aiming to find the most risky ones. To remove the false positive warnings, each warning were examined and determined if it could pose a threat or not. All inspected threats found during the manual examination were labeled as possible security improvements. Finally, all security warnings confirmed in the practical test environment were re-examined to be certain that no new or further exploitation was possible.

The security warnings were also examined at a system level; why they had occurred, how they could have been avoided and what kind of threats they posed. 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.

Besides checking all the warnings that were generated by the audit tools, a manual audit of selected components was conducted. The combined size of these components represents 10% of the overall source code used by the product. The purpose of the manual audit was to examine the effectiveness of the static analysis

tools. 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 proofs of concept programs had different goals in their attack; siege the system, sabotage or espionage.

5. Results

The findings were divided into four parts. First the efficiency of the automated tools, was investigated. Secondly, these findings were applied on the telecom product where the security risks are categorized into four groups. Third, different proof of concepts programs illustrated the possibility to create attacks. Finally an estimation of revision time and effectiveness was done, i.e. a ROI factor was decided.

5.1. The Static analysis tools

The different tools showed a mixed list of warnings depending on what level they were configured to report, lowest, standard or highest. Figure 2 below, showed the results from the tools at the different levels. At the lowest level every warning was reported compared to the highest level where only the most dangerous warnings should be reported.

	Lowest	Standard	Highest
Flawfinder	587	587	0
ITS4	504	310	36
RATS	520	306	242

Figure 2 Number of security warnings

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. Flawfinder found one unique warning that neither RATS nor ITS4 found. Flawfinder did not find a single insecurity dangerous enough to be labelled at the highest level.

ITS4 had the lowest overall report of warnings and worst correct findings of security risks. At the highest level, ITS4 reported 36 warnings where non of them was reported by the other tools at this level. All the warnings reported at the highest level in ITS4 were correctly reported by the other tools at a lower level of danger.

RATS was the tool with the least configurable levels of warnings. As a result RATS had the largest number of warnings, 242, at the highest level. All the warnings at the highest level were also unique when compared to the other tools at the highest level. RATS reported altogether 520 warnings of which 101 were unique.

5.2. Manual Examination of the Automated tools findings

All warnings reported by the automated auditing tools were manually examined. By doing a manual auditing on entire components, instead of auditing warnings separately, the automated tools insecurities could be better verified. During this manual component audit no new security risks were found, i.e. the security risks had already been reported by the automatic tools.

So, from the reported warnings by the three tools, a manual examination determined if the warnings were possible security improvements or if they were false alarms. Together the tools reported 823 unique warnings. The manual examination of these warnings was a time consuming task, resulting in the findings in Figure 3. In all there existed 59 possible security improvements (7.2% of the total warnings) when combining the different automatic tools with manual auditing. Of these 25 represented a security risk with consequences, i.e. almost half of the possible risks were real security threats. For the entire application there existed one possible security improvement per 1700 lines of source code. For every 4000 lines of source code there exists one security risk that could be exploited.

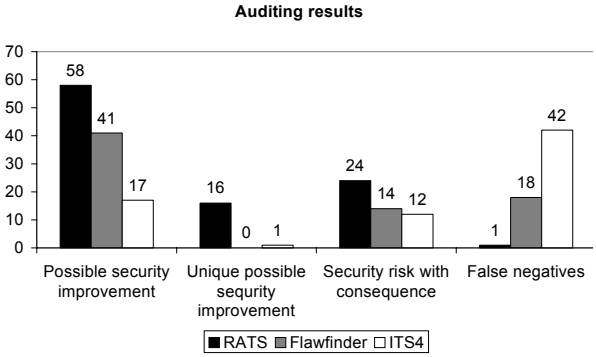

Figure 3 Auditing results after manual examination

RATS had the best results with just one false negative; this false negative was found by ITS4. RATS was also the only tool that found possible security improvements at its highest level, where 80% of all the possible security improvements were found.

All automated tools checked had false negatives because of deficiencies in the vulnerability database. Two reasons were found, either the risks were not present in the vulnerabilities database or the tools where not capable to perform the analysis required for finding these false negatives. The tool with the lowest number of false negatives was RATS. A missing registry in its database caused RATS single false negative. The false negatives in Flawfinder were also missing vulnerabilities in the database combined with deficiency inside the database, Flawfinder did not report known vulnerabilities. The false negatives from ITS4 were also missing vulnerabilities, but the larger portions were deficiency reporting fixed local buffers that were not used properly.

5.3. Security vulnerabilities

The security risks in the code were categorized into four groups: buffer overflow, misplaced trust, race condition and poor random generators, see Figure 4. Security risks with unsafe buffer handling were placed in the buffer overflow group. Incorrect input validations and the lack of validations were placed in the misplaced trust group. The race conditions were security risks where an attacker would be able to alter the information the software was going to use. Predictable random generators in a security function were placed in the poor random generator group. This was also the smallest of all groups.

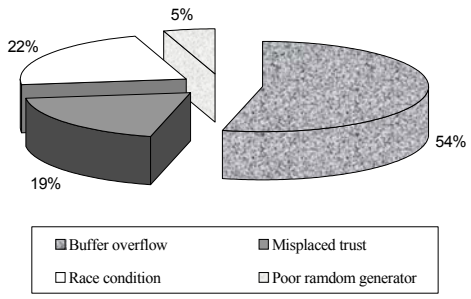

Figure 4 Proportions of different vulnerabilities

5.4. Proof of concepts

As already mentioned, three security risks were chosen and proof of concept programs were written to show the effect of the security risks.

- Siege the system with a buffer overflow: the buffer overflow created a opportunity to crash the program and create a denial-of-service attack. It also created the possibility of remote shell access, a far more dangerous risk. The proof of concept program created 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 as the most severe security risks.
- Sabotaging the system with a race condition: the race condition made it possible for an attacker to damage the system and more importantly, the ability to cover any track of the intrusion. The proof of concept program exploited a race condition present in the product to force the software to overwrite and delete log files. it is the software itself that destroys the log files, an Intrusion Detection System (IDS) would not detect the intrusion.
- Conducting espionage on the system due to poor random generators: the poor random generator created the possibility to break the encryption, thus enabeling an attacker to read any information important enough to encrypt. The proof of concept resulted in a decryption and encryption tool that may be used for breaking whatever encrypted data the attacker wants. The exploit made it possible to read other customers encrypted data and change them.

5.5. Return on investment

Since the automated tools used during the audit were open source tools, there was no significant investment cost to acquire the automated tool. There are more powerful tools that are not free and are better at finding security issues but for this article these high-end tools were excluded. The remaining investment cost is usage of the tool and learning the tool. The man-hours used to audit the findings of the tool are the only large investment cost worth calculating. So, the ROI depends on the ease of use and the targeted users, e.g. making the learning cost negligible compared to the overall project study costs.

	All warnings	RATS total	RATS standard
Warnings	843	520	306
Work days	42	27	16
Security warnings	100%	98%	91,5%
Improvement /day	2,38%	3,63%	5,72%

Figure 5 Security warnings investigated

In the investigation 843 warnings were examined, to fully comprehend the warning the surrounding code had to be understood. A total of 42 workdays were spent on manually investigating the warnings. This figure can be reduced if only one tool is used at a higher level. By only using RATS and at a standard level, RATS would have found 91.5% of all of the possible security improvements with 16 workdays of examination done by developers unfamiliar with security issues. The improvement per day was more than doubled compared to investigating all warnings.

For a relative small increase of time during the execution phase, it would be possible to lower the amount of undetected security faults. On the examined product the execution phase lasted approximately 720 days, when the time spent by all developers is combined. With an extra 16 days, used to review the source code with the assistance of RATS, a maximum of 54 security related faults were detected, resulting in decreased development costs. Since the 54 faults found by RATS were security related, neither function nor system tests were able to detected them. Consequently, the faults would have occasionally been detected by the product users, resulting in a twenty fold cost increase to correct the discovered faults. The code reviews during execution are done by the developers themselves, and there are no records or calculations of the corrections made. Therefore it was not possible to determine the exact gain from introducing static analysis tools into the execution phase.

6. Discussion

Manual audit done by experienced programmers is a time consuming but efficient method for conducting secure code revision of software. By comparing manual and automatic audit it was possible to estimate both the degree of audit tool improvements, and the deficiencies of program developers. The chosen rule based audit tools were not effective in finding security risks, but instead they showed where the manual auditing should be focused. The tools saved time and eased the burden on those performing the security analysis.

When looking at all 823 unique warnings from the three investigated tools, 7.2 % probable security warnings were found. RATS was the tool with the least false negatives and the most unique possible security improvements. During highest level report, 80% of these improvements were found which constituted a success rate of 19.5 %. Despite the low success rate, the cost to perform the audit with static analysis tools, was much lower then the fault correction cost the security issues would have costed if they were found later in the development process. When looking at all the warnings from all the three tools, the manual audit did not find any security risks that the tools had not already reported.

The open source project Apache [12] performed a similar test [2] with the tool RATS for about 110000 lines of code. The Ericsson 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, 12 warnings related to security risks. The apache project benefited from a large user base that has examined the code and made source code corrections. The Ericsson product had been restricted to the time available to the developers and had not undergone specific security testing, i.e. the initial security quality should for a true comparison be measured after the conducted test.

The security arguments for and against closed source code compared to open source code, are decreased exposure to risks and fewer persons involved in security improvements. At best, closed source developers have a security team or persons that examines the code. In principle, under the assumption of the standard reliability growth model, opening a system enables the attacker to discover vulnerabilities more quickly, but it helps the defenders exactly as much [1]. More efforts on both sides imply exploring and correcting more vulnerability, but the probability of finding a new security failure is not altered. The mean time to failure depends only on the initial quality of the code and the time spent testing it thus far. This has been discussed for general code revision [3] and security vulnerabilities [10]. The static analysis tool could help a project to discover its vulnerabilities by adding a security audit to the project code review. In the investigated products, the Ericsson product bene-

fited more from integrating static analysis tools to its development process.

Industrial software projects have time and resource constraints, i.e. delivering on time with a restricted amount of money. Improving security revisions at the same time as ongoing function tests fulfill these requirements. By using RATS standard settings about 90% of all security warnings found, could be corrected. Due to programmers experience in the source code and recent exposure, the audit time could have been further reduced if conducted as part of an execution process.

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 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.

For a more precise ROI a new audit should be done during the execution phase on the software's next project cycle. As a rule of thumb for an Ericsson project, the estimated cost for correcting vulnerability is increased twenty fold during a system test. This growth is mainly dependent on new persons involved for a function test and a new department involved for the system test, both generating heavy loads of people involved, new documentations and other overheads.

Today the customers, buying the product, may also demand security tests aside of function tests before the product is given global availability. It is, as shown above, much cheaper to do this during an ongoing execution phase. The automated tools generate warnings where insecure code exists 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 execution phase there will not be a great number of warnings to examine, compared to a complete examination of a finished product. The developer also learns from warnings and may avoid insecure code in the future. Since automated tools are fast, tests could be performed every day by sending new warnings to the developers for examination.

7. Conclusions

The audit of the Ericsson product showed one possible security improvement per 1700 lines of source code. Furthermore, there existed one security improvement that could be exploited every 4000:th lines of code. In all 823 potential security vulnerabilities were examined during 42 days of work. Half of the found threats in the product were buffer overflows followed by race condition, misplaced trust and to minor degree poor random generators.

Static analysis tools were used to speed up the revision process and to integrate security tests into the project process. Different tools with different levels of security warnings were examined where RATS at standard level setting produced the most efficient result. The results showed that RATS found 91.5% of all real security threats spending less than 40% of total time for finding them.

Bigger industrial projects usually include different subgroups for developing specific functions of the program in progress, before merging into the final system. All vulnerabilities with an action plan at the execution level are cheaper to handle than those vulnerabilities at the function level or exceedingly at the system level.

Today, customers demand security tests aside of function tests, making an action plan for correcting vulnerabilities indispensable. This work shows that proportionately small investments improve the program code security, by integrating an automatic auditing tool into the execution of source code revision. For the software's next project cycle a more precise return on investment calculation should be done focusing on optimizing the ratio between found vulnerabilities and time spent during the execution phase.

8. References

[1] Anderson, R.J. "Security in Open Versus Closed Systems - the Dance of Boltzmann, Coase and Moore", at Open Source Software Economics, 2002

[2] Audit for apache 1.3.26 (2002, October 04), URL http://www.sardonix.org/audit/apache-45.html, 2005-06-01

[3] Bishop, P., and Bloomfield, R., "A Conservative Theory for Long-Term Reliability-Growth Prediction", IEEE Transactions on Reliability v 45 no 4 pp 550-560 1996

[4] Chess, B., and McGraw, G., "Static Analysis for Security" Security & Privacy vol. 2, No 6, pp. 76-79, 2004

[5] Damm, L.O., Lundberg, L., and Wohlin, C., "Phase-Oriented Process Assessment using Fault Analysis", Proceedings of the 11th European Conference on Software Process Improvement, Springer-Verlag, 2004

[6] IDApro (2004), URL http://www.datarescue.com/idabase/ida.htm 2005-06-01

[7] Jiwnani K. and Zelkowitz M. , "Susceptibility Matrix: A New Aid to Software Auditing" Security & Privacy vol. 2 No 2 pp. 18-21 2004

[8] Johnson, S.C., “Lint, a C program checker” Computer Science Tech. report 65, Bell Laboratories, 1978

[9] Mc Graw G. Software security Security & Privacy vol. 2, No 2, pp. 80-83, 2004

[10] Rescorla, E., “Is finding security holes a good idea?”, in Workshop on Economics and Information Security, May 13-15, Minneapolis 2004

[11] Testing & Test control Notation, http://www.ttcn-3.org/ 2005-06-01

[12] The Apache Software Foundation (2004), URL http://www.apache.org 2005-06-01

[13] Viega, J. and McGraw G., Building Secure Software, Indianapolis, IN: Addison-Wesley, 2001

[14] Wagner D., Froster J., Brewer E.and Aiken A, “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, 2000