Journal of Mobile, Embedded and Distributed Systems, vol. VII, no. 2, 2015
ISSN 2067 – 4074

Software Tools for Source Code Analysis

Alexandru GROSU
IT&C Security Master
Department of Economic Informatics and Cybernetics
The Bucharest University of Economic Studies
ROMANIA
alex.grosu@gmail.com

Abstract: This article aims to look at the risks derived from vulnerabilities introduced by the source-code and present the means to detect them. The software tools that are often used in such situations are called analyzers and can be categorized in static and dynamic analyzers. This article will present the main advantages and disadvantages of each software tool used and subsequently show the results and a comparison between these utilities. Finally, conclusions will be drawn explaining how source code introduced vulnerabilities can be handled and avoided.

Key-Words: Static, Dynamic, Analysis, Valgrind, Memory, Security, Source-Code, Vulnerabilities

1. Introduction

The quick expansion of computer and internet usage [1] is well known and its effects on the world have been more and more significant, as the number of people having access to technology, either at work, or at home, have increased constantly over the last decades. With population growth come administrative difficulties which have been overcome by correct implementation of technology, especially software solutions destined to centralize information. Health care systems are now heavily based on complex computer architectures, while public services are mostly available online, especially in developed countries. Another very important impact of technology spread is access to information that is simplified and developing countries have benefited greatly from the possibility of easy education solutions available through the use of computers and internet.
There are no signs of the expansion of technology slowing down, and estimations show that the end of 2015 might see around 50% of the population having a computer at home and almost 45% having an active internet connection. This has a direct impact over software development in the sense that while the demand for software is increasing, the developers must be aware of their responsibility to deliver clean code, with a low number of vulnerabilities, especially in those applications that handle sensitive data.

2. Problem Formulation

While the expansion of technology has served communities greatly, this tendency is becoming more and more of a challenge for software developers [2] who can no longer consider that only experienced users will have access to their products. In other words, software security is relying less on additional components, such as antiviruses, firewall, etc. and more on the inner security, mostly ensured by secure coding, which results in less vulnerabilities introduced by the source code.
This does, by no means, imply that end-users should rely solely on the security of their software products and not take all necessary measures to protect themselves against attackers, the use of third party security tools being always indicated. Still, users should never undervalue the importance of verifying the authenticity of information providers and not trusting unknown sources, since a large part of regular attacks [3] are still done by users installing infected software.
The purpose of this article is to show how

developers can improve the quality of the source code, by avoiding common mistakes and identifying the more complex ones by using software tools to that analyze both the source code itself and the resulting program

2.1 Source Code. Importance of source code in security matters

Computer science defines source code as the human-readable text written in a programming language, representing a series of computer instructions. For the computer to be able to understand the source code and execute the commands, it must be translated into machine code. This process is done by using a compiler and sometimes an interpreter, as is the case for .Net or Java programming languages.

Within organizations, where multiple developers must have synchronized access to the source code, the latter is stored using source code management or versioning systems, such as SVN, git, etc. Access to these systems must be highly secured, to prevent intruders from getting unwanted access to the source code itself. Another important matter in handling source code is that it should not be shared outside the organization, except for cases where common development alongside the client is to be done. Users should always have access to the binary files and not the source-code, with the exception of open platforms, where the community is assisting in the development process. Regarding the importance of source code in security matters, there are several observations to be made:

Firstly, most frequent intrusions are based on errors in the source code, allowing the intruder to benefit from code weakness to take control of the program or even of the platform on which the program is running. With sensitive data becoming more and more important in applications, it is essential that information is correctly protected from any possible break-in.

Furthermore, as shown in the previous section, technology has spread and it is now available to a very wide range of users, under its different forms. A company developing a software product cannot and should not rely on the fact the users will have the necessary skills and resources to invest in additional layers of security to protect their products. A clean product should be delivered by guaranteeing that it will not introduce any liabilities on the system on which it will run, by behaving as designed and not needing further protection. This is, of course, an ideal and it does not mean that users will not and cannot further protect themselves; it simply states that they should not be obliged to do so.

2.1.1 Vulnerabilities issues introduced by source code

Source-code vulnerabilities [4] allow attackers to take advantage of weaknesses introduced by developers, either by error, or by not following secure programming best practices, in order to change the behavior of the program or the supporting platform. These vulnerabilities are specific to each area of programming and we will take a look at some of the most frequent situations. Of course, it is impossible to list all the possible vulnerabilities, but a quick overview of the most common ones is a good starting point in understanding the importance of writing clean code.

The vast majority of vulnerabilities introduced by the source-code [6] are based on an incorrect assessment of the possible input received from users. Programmers have to take into consideration all cases, especially limit cases, when dealing with input data, in order to ensure that the program's behavior is not affected by unexpected incoming data. Below are the most common such vulnerabilities:

A) Buffer Overrun [5]

Buffer overruns occur when data is written beyond the capacity limit of the buffer. They are most common in C and C++ which don't have built-in mechanisms to ensure correct memory writing and addressing, leaving these tasks to the programmer. Languages like Java, on the other hand, have built-in bounds checking and exceptions would be thrown, should the checks fail.

In order to determine buffer overruns, attackers need to load the addressing

ISSN 2067 – 4074

space with the attacking code and then trigger the execution of that code, thus taking control over the system. Generally, there are several types of buffer overruns:

A. Stack Overrun

Each section of a code function will determine the creation of a stack frame containing local variables, the return address to main and the arguments. Stack overflow attacks are based on overwriting the return address to point to different instructions, in order to take control of the system.

Stack overruns have been used by attackers for a long time and they are considered the most exploited source-code vulnerability.

B. Heap Overrun

Heap overruns are more difficult to exploit, because they are highly dependent on the system state. While the heap is the area where memory allocated variables are stored, the main goal of heap overruns is to overwrite variables stored after an overflowable buffer, by taking advantage of the fact that free heap sections are merged when applications free used memory.

C. String format errors

While some pieces of documentation present them as a separate type of vulnerability, they are arguably a type of buffer overflow, since the latter is the end result of such an attack. String format errors can result in the attacker being able to send incompatible data types as input parameters to functions and thus determining buffer overflows. Some functions like "printf" and "sprintf" are highly vulnerable to such attacks since they are dependent on the declared input type being the same with the parameter and they are also easy to identify in the binary file. On the other hand, such situations are easily detected during audit, reviews and code inspection, so release versions of applications should never be open to these vulnerabilities. Furthermore, many programming languages have built-in verifications to ensure correct input parameters are provided. Still, languages like C, C++ and Assembly are highly vulnerable and extra care is to be taken when implementing sensible function calls.

D. SQL Injection

SQL Injection attacks can take place when incorrect or incomplete verification of the input data is made and the attacker's goal is to determine the application to run SQL commands that were not designed by the developers. This kind of attacks are often encountered in applications that run SQL commands through strings created using data collected from the user. In such situations, the application must be designed to have several verifications of the input data, as well as other best practice guidelines that will be detailed in the next chapters. SQL injection attacks have been used for many years, since they are relatively easy to put in place with unsafe applications.

There are other vulnerability categories besides the ones above, which are less complex and derive from common mistakes made while implementing applications:

E. Integer Overflows

a) Sign conversion

Sign conversion vulnerabilities appear when the programmer does not take into consideration the return type of some functions and assumes implicit correct conversions will be made. A common mistake is using "sizeof" in C, which returns an unsigned integer. If the programmer uses this output in logical operations with variables that are signed, then the result will be the implicit conversion of the signed integer to unsigned, not vice-versa. This way, some memory addresses might be left uninitialized, leaving room for more critical errors and even leading to buffer overflows.

b) Arithmetic overflows or underflows.

Arithmetic overflows are a result of incorrect evaluation of the upper limits that a variable can and will have to store, in the sense that through some operations, the program tries to store into a variable a value greater than its limit. Such a situation will lead to buffer overflows.

Similar to arithmetic overflows, arithmetic underflows are a result of incorrect evaluation of the lower limits that a variable can and will have to store. A

possible result will be reserving too much memory space in situations where less would suffice.
Both situations are most common with the integer type and are often referred to as integer overflow and integer underflow.

F. Memory issues,

They can vary from trivial memory leaks determined by not freeing allocated memory, to using uninitialized variables in system calls leaving the possibility for attackers to take control over the system by executing random code.

2.1.2 Source code static and dynamic analysis

There are generally two types of analysis [7]:
Static analysis is based on the examination of the source code without executing the program (compile-time). Both security and performance issues can be detected through static analysis, although several tools might be needed, as each of these tools can identify certain categories of issues.
Dynamic analysis is based on the examination of the source code using certain criteria during run-time. Compared to static source code analysis more complex risks can be identified, but since running the program is required in order to execute dynamic analysis, the critical errors found through static analysis must be fixed, so that the code can be compiled and executed.

3. Software Tools for Source Analysis

In this section we will present the software tools that will be used to test the code

3.1 Static analysis tools

The following software analysis tools have been chosen in order to fulfill the tests:

3.1.1 Flawfinder

Flawfinder [12] is a source-code analyzer designed to run on Linux systems and can also be ported to run on Windows systems. Like any static tool, it does the checking at compile-time, so its results are independent of the compiler.

3.1.2 CppCheck

CppCheck [10] is static analysis tool used to identify errors and vulnerabilities in C and C++ code. It is significantly different from Flawfinder, since it focuses on more complex situations and does not check for syntax errors. This is done in order to minimize the high number of false positives, which is one of the main causes of developers not using static analyzers enough.
Although CppCheck does the checking at compile time, it should be run after trying to compile the code, in order to make sure that basis errors, such as syntax ones, are eliminated beforehand.
CppCheck advantages:

- Can output in XML format and other formats;
- Light tool, which runs fast;
- Very easy to use;
- Very low number of false positive;
- Users can define their own rules;
- Excellent documentation.

CppCheck disadvantages:

- Detects a lower amount of errors;
- Needs extensive configuration to be able to return complete results;
- Only for C/C++ (this can hardly be considered unexpected though, considering its name).

3.2 Dynamic analysis tools

Two utilities have been chose to run the tests and report the detected vulnerabilities

3.2.1 Valgrind

Valgrind [11] is one of the most spread and used dynamic analyzers. It is very complex and distributed under the form of a framework offering a number of tools that can check run-time parameters and developers can also build their own tools using the Valgrind framework.
It can be used for both vulnerability and performance testing, since it offers a wide range of parameters that it can monitor, check and report. With tools as complex as Valgrind, users need to take extra care when configuring them, since this will

ISSN 2067 – 4074

have a strong influence on the results obtained. Also, when try to test different versions of the same program and decide which one manages resources better, it is indicated to use the same configurations, in order to avoid inconsistencies. The built-in tools offer: memory error detection, thread error detection, cache and branch prediction profiling, heap profiling and three other experimental tools, including an array overrun detector.
Valgrind advantages:

- Can be installed using yum on CentoS, Fedora and Red-Hat or apt-get on Debian;
- Very well maintained code, with constant updates;
- Extensive documentation, which is clear and concise;
- Tests for complex, hard to identify situations and provides important feedback;
- Can perform multiple checks on the same run.

Valgrind disadvantages:

- Due to its complexity, configuration is very important and not trivial;
- Does not have a Windows version.

3.2.2 Dr. Memory

Dr. Memory [9] is tool used to monitor and identify memory-related errors. Running on both Windows and Linux, it can check for accesses of uninitialized memory, accesses to un-addressable memory, accesses to freed memory, double frees, memory leaks and other memory related errors.
Dr. Memory Advantages:

- Faster than Valgrind. This can be useful when testing programs with complex operations that require a lot of resources;
- Can run on both Windows and Linux systems;
- Can run more types of programs than Valgrind;
- Good, easy to understand documentation;

Dr. Memory Disadvantages:

- Although it can be installed on 64-bit operating systems, it can only check 32-bit applications;
- Detects less vulnerabilities and profiles less parameters than Valgrind.

Several benchmarks have been made using Dr. Memory and other dynamic analyzers and results show that, Dr. Memory has significantly lower run time, mostly because it is a lighter tool and its code is better optimized.

4. Running the Software Analysis Tools. Interpretation of Results

In order to demonstrate the use of software analysis tools, an environment has been setup as a virtual machine. This is extremely important because the source codes that have been used to test the tools introduce multiple risks, leaving the system vulnerable to attacks.

4.1 Running the Static Analysis Tools

The first step is to create the context for a static analysis tool, this being source codes that introduce multiple vulnerabilities. The goal is to test these codes with multiple tools and examine the results. The source codes that have been used include vulnerabilities such as incorrect string formatting, no boundary checking, signed-unsigned comparisons, altogether missing format strings, unused variables, undeclared variables, memory leaks (some trivial and some more complex).
The most important conclusion to be drawn out of the results presented is that each tool is able to detect different vulnerabilities.

Table 4.1-1 Test results.

Vulnerability	gcc – Wall	CppCheck	Flawfinder
Array out of bounds	No	Yes	No
Boundary checking missing - strings	No	Yes	Yes
Buffer Overrun	No	Maybe	Maybe
Division by zero	No	Yes	No
File path - no check	No	No	Yes
Incorrect use of function	Yes	Yes	Yes
Integer overflow/underflow	Yes	No	No
Memory leak (complex)	No	No	No
Memory leak (trivial)	No	Yes	No
Printf and Scanf risks	No	Yes	Yes
String formatting - incorrect	Yes	Yes	Yes
String functions parameters	Yes	Yes	Yes
Other Syntax errors	Yes	No	Maybe
Undefined variable	Yes	No	Yes
Uninitialized variable used	Yes	Maybe	No
Unused variable/function	Yes	Yes	No
Validation of input data - missing	No	No	Yes

As shown in Table 4.1-1, only two cases (both of unused variables) were reported by two different tools, while the other situations were identified by one single tool. As expected, Flawfinder is the most verbose, while CppCheck reports fewer errors, but also has no false positives. It is to be noted that the trivial memory leaks were only reported by CppCheck, thus demonstrating it is a tool built to analyze more diverse situations, although it ignores syntax errors. Also, we must stress that a number of vulnerabilities were only identified by the compiler.
The results of the tests come to prove how different one tool can be from another and stress the importance of using more than one tool in order to ensure that as many vulnerabilities as possible are identified before the application is released.

4.2 Running the Dynamic Analysis Tools

Similar to the static analysis scenarios, test source-codes that introduce vulnerabilities have been designed and implemented. The chosen risks that have been inserted in the programs include trivial memory leak (not freeing allocated memory at all), possible memory leaks (freeing memory leaks inside a conditional statement or before a return statement), using uninitialized variables, using un-addressable values in system calls, illegal memory frees of unallocated memory, illegal memory frees by trying to de-allocate the same memory twice.
The first tool used is Memcheck, the main tool provided by the Valgrind utility package. With a very low false positive return rate, Memcheck is one of the highly praised dynamic memory analyzers on the market. The second analyzer is Dr. Memory, which can only check for memory issues, unlike Valgrind which has several tools. The test results are shown by Table 4.2-1.

Table 4.2-1 Test results.

Vulnerability	Memcheck	Dr. Memory
Using uninitialized variables	Yes	Yes
Memory leak (trivial)	Yes	Yes
Memory leak (complex)	Yes	Yes
Illegal frees	Yes	Maybe
Using uninitialized variables	Yes	Yes
Memory handling warnings	Yes	Yes
Using invalid heap arguments	Yes	Yes

As we can see, the most important vulnerabilities have been detected by both programs, only the error messages being different. The complete reports will be presented in the complete work, to which this document refers to. The comparison above shows Valgrind's Memcheck versus Dr. Memory. Still, Valgrind also provides users with other useful tools, which will also be treated in more detail in the full study. These tools also help improve program performance, which can have a direct impact over the security of the application. This can occur in situations where the application uses significant resources which at some point are depleted, thus resulting either in a system crash, or system freeze.

5. Conclusions

Firstly, it is important to note that the applications tested in this study have been written in the C language because this is one of the programming languages

ISSN 2067 – 4074

that introduce a high number of vulnerabilities. When deciding to implement a solution, one way of avoiding some vulnerabilities can be to choose languages such as Java or .Net, if they are suitable for that project.

As regards the software tools used in this study, they were chosen as different from each other as possible in order to outline the importance of having the right tools at hand. The results were able to confirm this hypothesis: during static analysis Flawfinder and CppCheck were able to detect different kinds of errors and furthermore, the gcc compiler identified some vulnerabilities that neither of the tool had. Dynamic analysis using Valgrind and Dr. Memory showed the importance of thoroughly investigating the program at run-time, with more complex and especially memory-related issues unable to be detected at compile-time. With Valgrind being a more complex utility and Dr. Memory running a lot faster, it has been pointed out that dynamic analysis tools also behave very different from each other.

In order to ensure a high level of code security, the development work-flow should be well planned, with code reviews and inspections done after each major change. Furthermore, it has been proven that both analysis methods have to be used and it is recommended to make the most of the available tools by implementing more than one analyzer for both static and dynamic analysis. As for picking the right tools, this is a matter that has to be treated individually for each project by assessing the requirements, the risks and the budget, since commercial tools offer better results and are more flexible while open-source or freeware ones might lack support and be less adaptive to the solution's architecture.

Acknowledgement

Parts of this paper were presented at The 7th International Conference on Security for Information Technology and Communications (SECITC 2014), Bucharest, Romania, 12-13 June 2014.

References

[1] Computer and internet penetration statistics: http://www.itu.int/en/ITU-D/Statistics/Pages/default.aspx http://www.internetworldstats.com/emarketing.htm

[2] "The Evolution of Cyber Attacks and Next Generation Threat Protection Presentation" by Ashar Aziz, RSA Conference 2013

[3] "History of attacks", Lewis University, http://online.lewisu.edu/the-history-of-cyber-warfare.asp

[4] "Software Vulnerabilities, Prevention and Detection Methods: A Review", Willy Jimenez, Amel Mammar, Ana Cavalli,Telecom SudParis

[5] Stack and Heap Overrun http://drdeath.myftp.org:881/books/Exploiting/Stack.and.Heap.Overflow.pdf

[6] "Source Code Security" presentation, I. Smeureanu, University of Academic Studies of Bucharest, 2014

[7] "Program Analysis presentation", by Mario Barrenechea for Colorado University

[8] Commercial dynamic Analyser: http://www.viva64.com/en/d/

[9] Practical Memory Checking with Dr. Memory - Derek Bruening and Qin Zhao, http://www.burningcutlery.com/derek/docs/drmem-CGO11.pdf

[10] CppCheck: http://cppcheck.sourceforge.net/

[11] Flawfinder: http://www.dwheeler.com/flawfinder/

[12] Valgrind: http://www.valgrind.org/

[13] Dr. Memory http://www.drmemory.org/