Software Vulnerabilities, Prevention and Detection Methods: A Review[1]

Willy Jimenez , Amel Mammar, Ana Cavalli

Telecom SudParis. 9, Rue Charles Fourier
91000 Evry, France
{name.lastname}@it-sudparis.eu

Abstract. Software is a common component of the devices or systems that form part of our actual life. These systems are usually complex and are developed by different programmers. Usually programmers make mistakes in the code which could generate software vulnerabilities. A software vulnerability is a flaw or defect in the software construction that can be exploited by an attacker in order to obtain some privileges in the system. It means the vulnerability offers a possible entry point to the system. Despite the knowledge about vulnerabilities nowadays there is still a growing tendency in the number of reported vulnerabilities, reason why software security has become an important field of research. The presence of vulnerabilities in the production of software makes necessary to have tools that can help programmers to avoid or detect them in the development of the code. Thus, in relation to our on-going research on vulnerability detection, this article presents an overview of software vulnerabilities and their prevention and detection methods.

Keywords: Software vulnerability, Prevention/Detection Methods, Testing.

1 Introduction

A software vulnerability can be seen as a flaw, weakness or even an error in the system that can be exploited by an attacker in order to alter the normal behavior of the system. Because the number of software systems increases everyday also the number of vulnerabilities. Additionally, if we consider that most of the systems are exposed to multiple users (internet) and environments (operating systems for example) then it is just a matter of time that someone can launch an attack (sequence of actions) whose consequences are unpredictable in damages and cost. Usually the goal of an attacker is to gain some privileges in the system to take control of it or to obtain valuable information for its own benefit. Then it is important for the developers and general public to know about vulnerabilities and their prevention and detection.

[1] The research leading to these results has received funding from the European Community's Seventh Framework Program (FP 7/2007-203) under the grant agreement number 215995 (http://www.shields-project.eu/)

Under the context of the European project SHIELDS, whose main objective is to bridge the gap between security experts and software developers and thereby reduce the occurrence of security vulnerabilities; we present a review of different methods to detect and prevent software vulnerabilities as well as some well known software vulnerabilities. In most of the cases vulnerabilities are caused by improper validation of the data supplied by the user. This undesired condition is used by attackers to inject faults and malicious code into the system that allows them to run their own code and applications.

For better understanding vulnerabilities the creation of models that express the set conditions that could lead or originate them is very helpful; additionally when models are well understood they could also be used for prevention. But since it is impossible to guarantee the absence of vulnerabilities in a piece of code during its creation, then it is necessary to have methods to detect them. One possibility is security software inspections, or simply manual review of the code or related documents. Also some more automated methods for vulnerability detection can be applied, which are classified into two main categories: static, when the detection is performed without running the source code; and dynamic when the program is executed in order to detect vulnerabilities. Actually, in Telecom SudParis we are doing some research about the use of models in the detection of vulnerabilities.

The organization of this paper is as follow. In section 2 we introduce some known vulnerabilities and we mention possible consequences of their exploit. Section 3 contains methods to prevent vulnerabilities. In section 4 we study vulnerability detection techniques, classified into static and dynamic according to the execution of the source code. Section 5 describes our work in progress and finally in section 6 conclusions and perspectives of this work are presented.

2 Software Vulnerabilities

As we have mentioned a vulnerable software system can be exploited by attackers and the system could be compromised, the attacker might take control of the system to damage it, to launch new attacks or obtain some privileged information that he can use for his own benefit. Considering this, it is important to know the different types of vulnerabilities, their prevention and detection in order to try to avoid their presence in the final software version of the system and then reduce the possibility of attacks and costly damages.

2.1 Examples of vulnerabilities

Most of the known vulnerabilities are associated to an incorrect manner of dealing with the inputs supplied by an user of the system, if these inputs are not correctly processed before using them inside the program they can generate unexpected behavior of the system. For instance, some known and frequent vulnerabilities are [1]:

- Buffer overflow: it occurs usually with fixed length buffers when some data is going to be written beyond the boundaries of the current defined capacity.

This could lead to mal functioning of the system since the new data can corrupt the data of other buffers or processes. The buffer overflow can be used also to inject malicious code, and then the execution sequence of the program could be altered in order to execute the injected code and take control of the system.

- XSS or cross site scripting: usually associated to web applications, consists in the injection of code in the pages accessed by other users. If exploited an attacker can bypass access controls, perform phishing, identity theft or expose connections.
- SQL injection: it consists in the injection of code with the intension of exploiting the content of a database. Usually happens because the inputs are not handled correctly, the attacker can get sensitive information from the database.

However, some others common vulnerabilities that can be mentioned:

- Format string bugs: it happens when external data is given to an output function as format string argument. The output function, for instance, *printf* in C language, generates an output according to the specifications of the format string, some directives can write to memory locations, thus the attacker can use the *printf* to write malicious code and change the control flow to execute it.
- Integer overflows: can be of two different types, sign conversion bugs and arithmetic overflows. The first occurs when a signed integer is converted to an unsigned integer; while in the second the result of an arithmetic operation is an integer larger than the maximum integer and it is stored in an integer variable.

2.2 Vulnerability Modeling

Most of the vulnerabilities presented in the previous section could be prevented if the software is developed more carefully, avoiding the introduction of vulnerabilities that could be exploited by attackers. One solution is in the improvement of the knowledge and understanding of software developers about: known vulnerabilities, causes, threats, attacks and counter measures. Models are in fact adequate to implement such solution.

There is for instance a vulnerability model called *Vulnerability Cause Graph* (VCG) [2,3] which "is *a directed acyclic graph that contains one exit node representing the vulnerability being modeled, and any number of cause nodes, each of which represents a condition or event during software development that might contribute to the presence of the modeled vulnerability*". An example of a VCG representing a known buffer overflow in *xpdf* (CVE-2005-3192) taken from [2], is shown in figure 1. In this graph we can observe the different causes and possible scenarios or sequence actions that could lead to the introduction of this kind of vulnerability. The VCG is helpful to understand what can cause the vulnerability. If causes are well understood then they could be avoided in the development process.

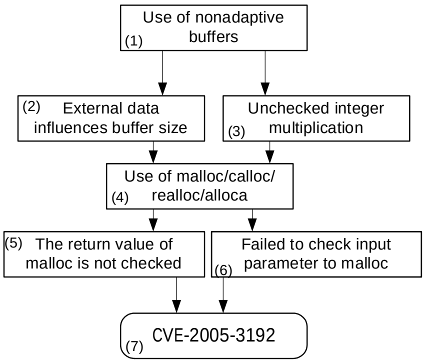
Use of nonadaptive buffers
(1)
(2) External data influences buffer size
Unchecked integer multiplication
(3)
Use of malloc/calloc/ realloc/alloca
(4)
(5) The return value of malloc is not checked
Failed to check input parameter to malloc
(6)
CVE-2005-3192
(7)

Fig. 1. Vulnerability Cause Graph

3 Preventing Software Vulnerabilities

Models are a first approach to deal with vulnerabilities and their understanding. However it is necessary to count on methods or procedures to prevent any risks related to vulnerabilities. In this section two possible vulnerability prevention methods developed in the literature are presented. The purpose is to evaluate the code during the construction process to detect any security defect and correct it on time without the need of performing intensive test at the end when the whole program is finished.

3.1 Software Inspection

The software inspection process consists in reading or visually inspecting the program code or documents in order to find any defects and correct them early in the development process. When the defect is found soon the less expensive it becomes to fix. However, a good inspection depends then on the ability and expertise of the inspector, and the kind of defects he is looking for. Usually during the software inspection, it is necessary to look for any possible defects during the security inspections. In the following sections we introduce two inspection methods that intend to codify the implicit knowledge of security experts regarding how to check for correct implementation of security goals and how to search for vulnerabilities.

3.1.1 Security Goal Indicator Trees

Security Goal Indicator Trees (SGIT) [4] focus on positive features of the software which can be verified during the inspection process. A SGIT is then a graph where the root is a security goal and its subtree are indicators or properties that can be checked for achieving that goal. However, since not all properties can be positively expressed it is possible to have also negative indicators (something that should not occur). These indicators have Boolean relations with the goal and have to be checked in order to validate the security goal. SGIT are created by security experts. A SGIT for the goal Audit Data Generation, taken from [4], is presented in figure 2, showing some dependency relations, and positive and negative indicators. Also the small box pointing to the indicator “An audit component exists” means that a specialization tree can be deployed for this indicator.

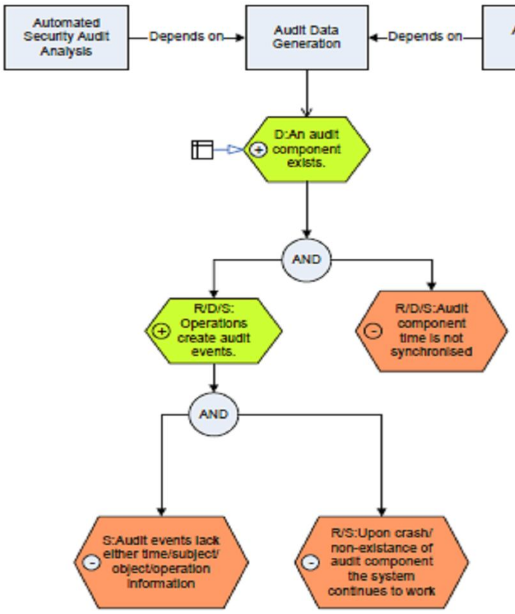
Automated Security Audit Analysis
Depends on
Audit Data Generation
Depends on
Audit Data Storage
+
D:An audit component exists.
AND
+
R/D/S: Operations create audit events.
-
R/D/S:Audit component time is not synchronised
AND
-
S:Audit events lack either time/subject/ object/operation information
-
R/S:Upon crash/ non-existance of audit component the system continues to work

Fig. 2. Security Goal Indicator Tree

3.1.2 Vulnerability Inspection Diagram

Vulnerability Inspection Diagram (VID) is also a manual inspection introduced in [5], the purpose is to benefit developers from the knowledge and experience of security experts in the detection of problems in the development process. Thus a VID is a flowchart-like graph that guides developers to check the software to detect the presence of vulnerabilities based on the knowledge of experts. There is a specific VID for each vulnerability class.

3.2 Security Activity Graph

Security Activity Graphs (SAGs) [3,6] are also helpful in the prevention of vulnerabilities. SAGs are a graphical representation that is associated with causes in a VCG. SAGs indicate how a particular cause can be prevented following a combination of security activities during the development process. To illustrate this, in figure 3 there is a SAG [6] showing different alternatives to address the cause "Lacking design to implementation traceability".

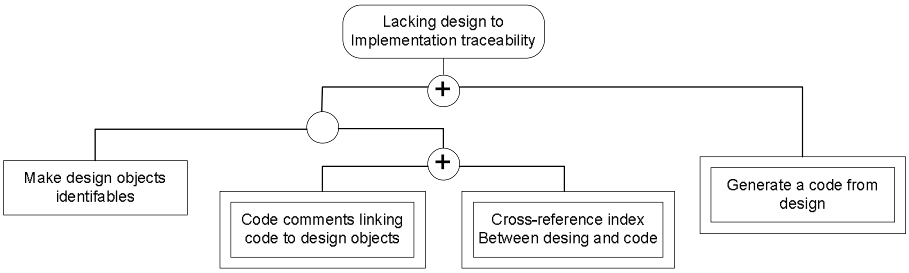
Lacking design to Implementation traceability
+
Make design objects identifables
+
Code comments linking code to design objects
Cross-reference index Between desing and code
Generate a code from design

Fig. 3. Security Activity Graph

4 Detecting Software Vulnerabilities

Models and inspections are useful to understand and prevent vulnerabilities; nevertheless it is also necessary to count on tools that can be used by programmers in order to detect vulnerabilities during the process of software construction.

Some of these tools are based on static methods, thus it is not necessary to run the code to perform the detection. In the case of dynamic methods, the code is run inside a controlled environment to perform the detection or collect program traces that can be use for such purpose. In the next section we present some static and dynamic techniques to detect vulnerabilities.

4.1 Static Techniques

Static techniques are those applied directly to the source code without running the application, the objective is to evaluate or get specific information directly from the source code without executing it. There are different techniques to perform static analysis; here we mention some of them.

4.1.1 Pattern Matching

Consists in searching a "pattern" string inside the source code and give as results the number of occurrences of it. For instance if we consider C language, the pattern could be any call to possible dangerous functions (vulnerable) like "getc". Pattern matching can be implemented using a simple tool like the Unix command "grep", however this method generates much false positives because there is no analysis of the results, additionally its effectively is limited since depends on the exact writing of the strings, thus additional white spaces will limit the results.

Flawfinder also uses a more elaborated pattern matching process to find possible vulnerabilities and sort them by risk level [7]. This risk level depends on the function and on the values of the parameters of the function.

4.1.2 Lexical Analysis

Lexical analysis adds an additional step before applying a pattern match. In fact, the source code is transformed into a sequence of tokens, which are later compared with a vulnerability database in order to identify them. False positives number is still high because they do not consider the syntax or grammar of the program. The ITS4 [8] tools use lexical analysis.

4.1.3 Parsing

Parsing is more complex than lexical analysis, thus when the source code is parsed, a representation of the program is built using a parsing tree in order to analyze the syntax and the semantics of the program. For example the parsing technique is used to detect SQL command injection attacks [9].

4.1.4 Type Qualifier

Type qualifiers are used to qualify types and modify the properties of variables in the programming language, as in the case of the tool Cqual [10]. Cqual is used to specify and check properties of C programs using user-defined *type qualifiers*, which are added to the program. The modified program is analyzed to find vulnerabilities.

4.1.5 Data Flow Analysis

The purpose is to determine the possible values a variable or an expression can have during the execution of the program, specially suited for buffer overflow detection. For instance data flow analysis is used in [11]. The authors take rules describing vulnerability patterns and the source code to detect locations and paths of the pattern in the program. The process is executed in three parts: pattern matching, control and data flow and flow analyzer.

4.1.6 Taint Analysis

It is a special case of data flow analysis where any data coming from un-trusted sources, e.g. introduced by a user, is a potential problem to the system, thus it is marked as tainted. Tainted data flow is monitored because it can not reach critical functions unless it is processed and changed to untainted.

Livshits and Lam [12] propose a static analysis framework to find vulnerabilities in Java applications. They define a Tainted Object Propagation problem class to deal with improper user input validation. Java bytecode and vulnerability specifications are employed to perform a taint object propagation and find vulnerabilities using the Eclipse platform.

4.1.7 Model Checking

Model Checking is a technique to automatically test if the model of a system meets its specification and it can be used to detect vulnerabilities. Usually model checking is a complex technique because the elaboration of the model is difficult, however once obtained it is easier to test the properties of the system.

A security verification framework with multi-language support was developed [13] based on GCC compiler. Their approach uses a conventional push down system model checker for reach ability properties to verify software security properties; it is composed of three phases: security property specifications, program model extraction and property model checking, this last has as output the detected errors with execution traces.

Constraint analysis is combined with model checking [14] in order to detect buffer overflow vulnerabilities. They trace the memory size of buffer-related variables and the code instrumented with constrains assertions before the potential vulnerable points. The vulnerability can be detected with the reach ability of the assertion using model checking. They decrease the cost of model checking slicing the program.

4.2 Dynamic Techniques

In order to dynamically detect vulnerabilities it is necessary to execute the program code, and then analyze the behavior or the answers of the system and gives a verdict. In the next part we study some of the techniques to perform dynamic detection.

4.2.1 Fault Injection

Fault injection is a testing technique that introduces faults in order to test the behavior of the system, some knowledge about the system is required to generate the possible faults. With fault injection is possible to find security flaws in the system [15], in this work faults are injected into the system under test and the system behavior is observed, the failure to tolerate faults is an indicator of a potential security flaw in the system, a model is used to decide what faults to inject.

4.2.2 Fuzzing Testing

The idea of this test is to provide random data as input to the application in order to determine if the application can handle it correctly. Fuzzing testing is easier to implement than fault injection because the test design is simpler and previous knowledge about the system to test is not always required, additionally it is limited to the entry points of the program. Web scanners are in this tool category.

Fuzzing testing can also be improved to have a better coverage of the system. For instance recording real user inputs to fill out web forms and then utilize the collected data in the fuzz testing process to better explore web applications (reach ability) [16].

4.2.3 Dynamic Taint

Similar to taint analysis, however in this case the tainted data is monitored during the execution of the program to determine its proper validation before entering sensitive functions. It enables the discovering of possible input validation problems which are reported as vulnerabilities [17].

4.2.4 Sanitization

One possibility to avoid vulnerabilities due to the use of user supply data is the implementation of new incorporated functions or custom routines whose main idea is to validate or sanitize any input from the users before using it inside a program. In [18] they present an approach using static and dynamic analysis to detect the correctness of sanitization process in web applications that could be bypass by an attacker. They use data flow techniques to identify the flows of input values from sources to sensitive sinks or the places where the value is used. Later they apply the dynamic analysis to determine the correct sanitization process.

5 Our approach

Our research in Telecom SudParis evolves around the use of models for tool-based detection of vulnerabilities. In our approach a vulnerability model, the Vulnerability Cause Graph are considered as an input in order to derive a formalism called Vulnerability Detection Condition [19], with the goal of automatically test the source code to detect vulnerabilities. The main idea behind this concept is to use the information provided by VCGs to point out in the code the use of a dangerous action under some particular conditions, for instance “it is dangerous to use unallocated memory”. The checking for vulnerabilities is performed on execution traces of the program using the TestInv tool [20]. Another tool, TestGen [20], might be adapted to generate test cases for the detection of vulnerabilities.

Another possibility considered in our research is the extension of Martins et al work [21] which uses attacks trees to generate test cases to uncover protocol vulnerabilities. The objective is to extend this work toward more general programs. Additionally we will evaluate a possible integration with the modeling tool Seamonster [22], it uses attack trees, in order to have a more powerful tool.

6 Conclusions

As we can see vulnerabilities are not a new topic on software field; however it is also noted that they still appear in the source code, thus it means that programmers still do not know how to deal with vulnerabilities. In order to help programmers to build better code we could use vulnerability cause graphs to teach them how the vulnerabilities are introduced thus they could learn to avoid the presence of vulnerabilities in their source code. However, in the mean time the source code should be inspected to guarantee there are no vulnerabilities, this method can be applied several times during the construction phase as advantage but requires specialists to perform the task as drawback.

Also a number of tools are available in order to detect vulnerabilities, some of them are based on static techniques, and it means the source code is analyzed without running the application while on dynamic techniques it is necessary to run it. The selection of the tools is related to the type of application to evaluate, the programming language and the type of vulnerability to detect. The static techniques cover all possible execution paths but require the source code while dynamic techniques have the difficulty of requiring the preparation of test cases and the possibility that not all paths in the program are covered, but the advantage that the problems if any, are found in the running code. Dynamic techniques have also less false positives than statics.

Finally, our current research intends to create new vulnerability detection methods based on models. In this manner we could guarantee the reusability of the tests cases and facilitate the transformation of these formal representations into the specific programming language of the tool used to perform the vulnerability detection.

References

1. S. Christey. Unforgivable Vulnerabilities. The MITRE Corpotarion.2007.
2. D. Byers, S. Ardi, N. Shahmehri, C. Duma. Modeling Software Vulnerabilities with Vulnerability Cause Graphs. In Proceedings of the International Conference on Software Maintenance, Philadelphia, PA, USA, 2006.
3. S. Ardi, D. Byers, and N. Shahmehri. Towards a structured unified process for software security. In Proceedings of the ICSE 2006 Workshop on Software Engineering for Secure Software (SESS06), Shanghai, China, 2006.
4. H. Peine, M. Jawurek, S. Mandel. Security Goal Indicator Trees: A Model of Software Features that Supports Efficient Security Inspection. HASE, pp.9-18, 2008 11th IEEE High Assurance Systems Engineering Symposium.
5. SHIELDS Project Consortium. D2.1 Formalism definitions and representation schemata. SHIELDS Project Deliverable D2.1. http://www.shields-project.eu/.
6. D. Byers, N. Shahmehri. A Cause-Based Approach to Preventing Software Vulnerabilities. ARES, pp.276-283, 2008 Third International Conference on Availability, Reliability and Security.
7. D. Wheeler. Flawfinder, April 2007. http://www.dwheeler.com/flawfinder/.
8. ITS4: Software Security Tool. http://www.cigital.com/its4/.

9. Z. Su, G. Wassermann. The Essence of Command Injection Attacks in Web Applications. Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp.372-382, 2006.
10. Cqual, A tool for adding type qualifiers to C. http://www.cs.umd.edu/~jfoster/cqual/.
11. H. Kim, T. Choi, S. Jung, H. Kim, O. Lee, K. Doh. Applying Dataflow Analysis to Detecting Software Vulnerabilities. ICACT, pp.255-258. 10th International Conference on Advanced Communication Technology, 2008.
12. V. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis in Proceedings of the 14th conference on USENIX Security Symposium - Volume 14, 2005, pp.18.
13. R. Hadjidj, X. Yang, S. Tlili, M. Debbabi. Model-Checking for Software Vulnerabilities Detection with Multi-Language Support. PST, pp.133-142, 2008 Sixth Annual Conference on Privacy, Security and Trust.
14. L. Wang, Q. Zhang, P. Zhao. Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking. SCAM, pp.165-173, 2008 Eight IEEE International Working Conference on Source Code Analysis and Manipulation.
15. W. Du, A. Mathur. Vulnerability Testing of Software System Using Fault Injection. in Proceeding of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop On Dependability Versus Malicious Faults.
16. S. McAllister, E. Kirda, C. Kruegel. Expanding Human Interactions for In-Depth Testing of Web Applications. RAID 2008, 11th Symposium on Recent Advances in Intrusion Detection.
17. B. Chess, J. West. Dynamic Taint Propagation: Finding Vulnerabilities without Attacking. Information Security Technical Report. Volume 13, Issue 1, 2008, Pages 33-39.
18. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel§, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. SP, pp.387-401, 2008 IEEE Symposium on Security and Privacy.
19. A. Mammar, A. Cavalli, E. Montes de Oca, S. Ardi, D. Byers, N. Shahmehri. Modélisation et Détection Formelles de Vulnérabilités Logicielles par le Test Passif. 4ème conférence sur la Sécurité des Architectures réseaux et des Systèmes d'information. June 2009.
20. A. Cavalli, E. Montes De Oca, W. Mallouli, M. Lallali. Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints, The 12-th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2008).
21. E. Martins, A. Morais, A. Cavalli. Generating Attack Scenarios for the Validation of Security Protocol Implementations. The 2nd Brazilian Workshop on Systematic and Automated Software Testing (SBES 2008 -SAST), October 2008, Brazil.
22. Meland P., Spampinato D., Hagen E., Baadshaug E., Krister K., Velle K. (2008). SeaMonster: Providing tool support for security modeling. NISK 2008. National Conference on Information Security. November 2008.