ISSN 2047-3338

A Survey on Software Security Testing Techniques

Abdullah Saad AL-Malaise AL-Ghamdi
Department of Information Systems, Faculty of Computing & Information Technology,
King Abdulaziz University, Kingdom of Saudi Arabia
aalmalaise@kau.edu.sa

***Abstract*– This article briefs a survey on software security techniques. Software security testing is not the identical as testing the correctness and competence of security functions implemented by software, which are most frequently verified through requirements-based testing. These tests are important; they expose only a small piece of the depiction needed to verify the security of the software. Security testing is necessary because it has a distinct relationship with software quality. Software meets quality requirements related to functionality and performance, it does not necessary mean that the software is secure.**

***Index Terms*– Software Security, Assurance, Reliability and Recoverability**

I. INTRODUCTION

SOFTWARE behaves in the presence of a malicious attack, even though in the real world, software failures usually happen spontaneously-that is, without intentional mischief. Security is always next of kin to the information and services being protected, the skills and resources of adversaries, and the costs of potential assurance remedies; security is executed in risk management. Risk analysis, especially at the design level, can help us identify potential security problems and their impact. Once recognized and categorized, software risks can then assist software security testing.

Software assurance is comprised of reliability, recoverability, and resiliency aspects of the software. Software testing must address all of these. Software testing for functionality should always be improved with security testing for resiliency [1]. Unit testing is performed by developers and has the benefit of detecting functional and software assurance issues early on in the life cycle because it breaks the software into small convenient units. Regression testing is essential when code changes. It can be used to compute the relative attack surface from one version to another and provide imminent into whether the state of software security is improving or deteriorating. The difference between software safety and software security is presence of an smart adversary twisted on breaking the system. Software quality, reliability and security belong to one family. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more ruthless and excruciating. Many critical software applications and services need integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially guide to security violations, and validating the effectiveness of security measures.

A vulnerability tools is a program that performs the analytical phase of a vulnerability analysis, and assessment. Vulnerability analysis defines, identifies, and classifies the security holes and their weakness in computer systems includes network, server, or communications channel. Also vulnerability analysis can predict the effectiveness of proposed countermeasures, and evaluate how well they work after they are put into use. The tools relies on a database that contains all the information required to check a system for security holes in services and ports, anomalies in packet construction, and potential paths to exploitable programs. Vulnerability assessments that verify the presence of security controls, and penetration testing which is used to determine if those security controls are effectively working, are common security Testing techniques.

II. REQUIREMENT FOR SECURITY TESTING

Basically there are three key quality components to software assurance. These are reliability, resiliency, and recoverability. Reliable software is that which functions as needed by the end user. Resilient software is that which is able to endure the attempts of an attacker to compromise impact integrity, confidentiality and availability. We can say a secure software should achieves these security requirements

A. Possible Attacks on Software

This section describes the various possible attacks. The large systems are typically most susceptible to, due to malicious outsiders and an insider includes users, processes and applications. The possible attacks are

- Information Disclosure Attacks Applications can often be forced to reveal sensitive or useful data. Error

messages generated by the application often contain information useful to attackers. Attacks of this type include directory indexing attacks, path traversal attacks and determination of whether the application allocates resources from a conventional and accessible location. The target with this set of attacks is to segregate any and all cases of information leakage.

- System Dependency Attacks: Vital system resources can be identified by monitoring the environment of use of the application and targeted. A system must have the ability to securely process corrupt, missing and Trojaned files. Large systems are often vulnerable to input strings that tend to cause insecure behaviors. Attacks in this class include large strings, command injection, LDAP injection, OS commanding, SQL injection, SSI injection, format strings, escape characters, and special/problematic character sets.
- Logic/Implementation (business model) Attacks The hardest attacks to apply are often the most profitable for an attacker. These include broadcast temporary files for sensitive information, attempts to mall-treatment internal functionality to expose secrets and cause insecure behavior, checking for faulty process validation and testing the application's ability to be remote-controlled. Users may get in between the time-of-check and time-of-use of sensitive data and perform denial of service at the component level.
- Authentication/Authorization Attacks These attacks is comprise both dictionary attacks and common account/password strings) and credentials, exploiting key materials , insufficient and poorly implemented protection and recovery of passwords, key material both in memory and at component boundaries.

III. SECURITY TESTING TECHNIQUES

We have reviewed many articles on security testing techniques and brief here. Basically in software engineering the:

- Code reviews
- Automated static analysis
- Binary code analysis
- Fuzz testing
- Source and binary code fault injection
- Risk analysis
- Vulnerability scanning
- Penetration testing

A. Risk Analysis

To review security requirements and to identify security risks, risk analysis is carried out during the design phase of development. Threat modeling is a methodical process that is used to identify threats and vulnerabilities in software. It helps system designers to analyze and think about the security threats that their system might face. Therefore, threat modeling is carried out as risk assessment for software development. In fact, it enables the designer to develop mitigation strategies for potential vulnerabilities and helps them focus their limited resources and attention on the parts of the system most at risk. It is recommended that all applications have a threat model developed and documented. Threat models should be created as early as possible in the SDLC, and should be revisited as the application evolves and development progresses. To develop a threat model, implement a simple approach that follows the NIST 800-30 [7] standard for risk assessment. This approach involves:

- Decomposing the application - understand, through a process of manual inspection, how the application works, its assets, functionality, and connectivity.
- Defining and classifying the assets - classify the assets into tangible and intangible assets and rank them according to business importance.
- Exploring potential vulnerabilities - whether technical, operational, or management.
- Exploring potential threats - develop a realistic view of potential attack vectors from an attacker's perspective, by using threat scenarios or attack trees.
- Creating mitigation strategies - develop mitigating controls for each of the threats deemed to be realistic.

B. Code Review

Source code review is carried by static analysis. It is the process of manually checking source code for security vulnerability. Many serious security weaknesses cannot be detected with any other procedure of analysis or testing. According to the security community there is no alternate for actually looking at code for detecting subtle vulnerabilities. Unlike testing third party closed software such as operating systems, when testing applications the source code should be made available for testing purposes. Many unintentional but significant security problems are also extremely difficult to discover with other forms of analysis or testing, such as penetration testing, making source code analysis the technique of choice for technical testing. The advantages of code review are Completeness, effectiveness, and Accuracy. Disadvantages are not practical for large code bases, requires highly skilled reviewers, labor intensive, and infeasible to detect runtime errors.

With the source code, a tester can accurately determine what is happening and remove the speculation work of black box testing. The issues includes concurrency problems, time bombs, logic bombs, flawed business logic, access control problems, and cryptographic weaknesses as well as back doors, Trojans and other forms of malicious code can be exposed by source code reviews. These issues often visible themselves as the most harmful vulnerabilities in web sites. Source code analysis can also be extremely efficient to find implementation issues such as sections of the code where input validation was not performed or where fail open control procedures may be present.

Operational procedures need to be reviewed as well, since the source code being deployed might not be the same as the one being analyzed. Code review is highly work exhaustive, but can, when reviewers with appropriate levels of experience perform the review, produce the most complete, accurate

results early in the review process, before the reviewer fatigues. It is common for the reviewer to begin by very scrupulously checking every line of code, then to gradually skip larger and larger portions of code, so that by the end of the review, the inconsistent and decreasing amount of ”code coverage” is insufficient to determine the true scenery of the software. It is important to note, that as the size of the code-base increases it becomes less feasible to perform a complete manual review. Code reviews are also useful for detecting indicators of the presence of malicious code. For example, if the code is written in C, the reviewer might seek out comments that indicate exploit features, and/or portions of code that are complex and hard-to-follow, or that contain embedded assembler code.

C. Automated Static Analysis

Automated static analysis is any analysis that examines the software without executing it, and it involves the use of a static analysis tool. In most cases, this means analyzing the program’s source code, although there are a number of tools for static analysis of binary executables. Because static analysis does not require a fully integrated or installed version of the software, it can be performed iteratively throughout the software’s implementation. Automated static analysis does not require any test cases and does not know what the code is intended to do [11]. The main objective of static analysis is to find out security flaws and to identify their potential fixes. The static analysis tool output should provide enough detailed information about the software’s possible failure points to enable its developer to classify and prioritize the software’s vulnerabilities based on the level of risk they pose to the system.

Static analysis testing should be performed as early and as often in the life cycle as possible. The most effective tests are performed on granularly small code units-individual modules or functional-process units-which can be corrected relatively easily and quickly before they are added into the larger code base. Iteration of reviews and tests ensures that flaws within smaller units will be dealt with before the whole system code review, which can then focus on the ”seams” between code units, which represent the relationships among and interfaces between components. Static analysis tools are effective at detecting language rules violations such as buffer overflows, incorrect use of libraries, type checking and other flaws.

Static analysis tools can scan very large code bases in a relative short time when compared to other techniques. The reviewer’s job is limited to running the tool and interpreting its results. Static analysis tools are not efficient enough to detect anomalies that a human reviewer would determine. The tools can provide additional benefits, by allowing developers to run scans as they are developing-addressing potential security vulnerabilities early in the process. Similarly, the level of expertise required for an automated review is less than that required for a manual review. In many cases, the tool will provide detailed information about the vulnerability found, including suggestions for mitigation.

1) Limitation of Static analysis tools

Following are the limitations of Static analysis tools

- Limited number of path to analyze since a full exploration of all possible paths through the program could be very resource intensive.
- Third party code - If part of a source code is not available, such as library code, OS, etc., the tool has to make assumptions about how the missing code operates.
- Incapability to trace out unexpected flaws - Flaw categories must be predefined.
- Incapability to trace out architectural errors.
- Incapability to trace out system administration or user mistakes.
- Incapability to find vulnerabilities introduced or exacerbated by the execution environment.

D. Source and binary code fault injection

Source code fault injection is a testing technique originated by the software safety community where as Binary fault injection is an adjunct to security penetration testing to enable the tester to obtain a more complete picture of how the software responds to attacks. .Source code fault injection is used to induce stress in the software, create interoperability problems among components, simulate faults in the execution environment, and thereby reveal safety-threatening faults that are not made apparent by traditional testing techniques. Security fault injection extends standard fault injection by adding error injection, thus enabling testers to analyze the security of the behaviors and state changes that result in the software when it is exposed to various perturbations of its environment data. Software programs interact with their execution environment though operating system calls, remote procedure calls, application programmatic interfaces, man machine interfaces, etc. Binary fault injection involves monitoring the fault injected software’s execution at runtime.

For example, by monitoring system call traces, the tester can decipher the names of system calls and the call’s return code/value (which reveals success or failure of the access attempt. In binary fault injection, faults are injected into the environment resources that surround the program. Environmental faults in particular are useful to simulate because they are most likely to reflect real world attack scenarios. However, injected faults should not be limited to those simulating real world attacks. As with penetration testing, the fault injection scenarios exercised should be designed to give the tester as complete as possible an understanding of the security of the behaviors, states, and security properties of the software system under all possible operating conditions.

IV. FUZZ TESTING

Fuzzing is a technique for finding security-critical flaws in any software in a very less computational cost and time. Fuzz testing takes random invalid data to the software under test through its environment or another software component. Fuzzing means a random character generator for testing applications by injecting random data at their interfaces. In other ward it means injecting noise at program interfaces. Fuzz testing is implemented by a program or script that

submits a combination of inputs to the software to disclose how that software responds. The idea is to look for interesting program behavior that results from noise injection and may indicate the presence of vulnerability or other software fault. Fuzzers are generally specific to a particular type of input, such as HTTP input, and are developed to test a specific program; they cannot be reused. Their value is their specificity, because they can often reveal security vulnerabilities that generic testing tools such as vulnerability scanners and fault injectors cannot.

Fuzzing might be characterized as a blind fishing mission that hopes to uncover completely unsuspected problems in the software. For example, suppose the tester intercepts the data that an application reads from a file and replaces that data with random bytes. If the application crashes as a result, it may indicate that the application does not perform needed checks on the data from that file but instead assumes that the file is in the right format. The missing checks may (or may not) be exploitable by an attacker who exploits a race condition by substituting his or her own file in place of the one being read, or an attacker who has already subverted the application that creates this file. The main focus of fuzzing is on functional security assessment. As fuzzing is essentially functional testing, it can be conducted in various steps during the overall development and testing process.

V. BINARY CODE ANALYSES

In Binary code analysis the technique of reverse engineering and analysis of binary is used. This executes as decompiles, disassembles, and binary code scanners, reflecting the varying degrees of reverse engineering that can be performed on binaries.

The least intrusive technique is binary scanning. Binary scanners, analyze machine code to model a language-neutral representation of the program's behaviors, control and data flows, call trees, and external function calls. Such a model may then be traversed by an automated vulnerability scanner in order to locate vulnerabilities caused by common coding errors and simple back doors. A source code emitter can use the model to generate a human-readable source code representation of the program's behavior, enabling manual code review for design level security weaknesses and subtle back doors that cannot be found by automated scanners. The most intrusive reverse engineering technique is de-compilation, in which the binary code is reverse engineered all the mode back to source code, which can then be subjected to the same security code review techniques and other white box tests as original source code. Note, however, that de-compilation is technically problematical: the quality of the source code generated through de-compilation is often very poor.

Such code is rarely as navigable or comprehensible as the original source code, and may not accurately reflect the original source code. This is particularly true when the binary has been obfuscated or an optimizing compiler has been used to produce the binary. Such measures, in fact, may make it impractical to generate meaningful source code. In any case, the analysis of decompiled source code will always be significantly more difficult and time consuming than review of original source code. For this reason, de-compilation for security analysis only makes sense for the most significant of high effective components. The next least intrusive technique is disassembly, in which binary code is reverse engineered to intermediate assembly language [3]. The drawback of disassembly is that the resulting assembler code can only be meaningfully analyzed by an expert who both thoroughly understands that particular assembler language and who is skilled in detecting security-relevant constructs within assembler code.

VI. VULNERABILITY SCANNING

Application vulnerability scanners are a very important software security testing technique. These tools scan the executing application software for input and output of known patterns that are associated with known vulnerabilities In application level software, automated vulnerability scanning is used. Also uses for Web servers, database management systems, and some operating systems. These vulnerability patterns, or "signatures", are comparable to the signatures searched for by virus scanners, or the "dangerous coding constructs" searched for by automated source code scanner, making the vulnerability scanner, in essence, an automated pattern-matching tool. While they can find simple patterns associated with vulnerabilities, automated vulnerability scanners are unable to pinpoint risks associated with aggregations of vulnerabilities, or to identify vulnerabilities that result from unpredictable combinations of input and output patterns.

In addition to signature-based scanning, some Web application vulnerability scanners attempt to perform "automated state full application assessment" using a combination of simulated reconnaissance attack patterns and fuzz testing techniques to "probe" the application for known and common vulnerabilities. Like signature-based scans, state full assessment scans can detect only known classes of attacks and vulnerabilities [10].

Most vulnerability scanners do attempt to provide a mechanism for aggregating vulnerability patterns. The current generation of scanners is able to perform fairly unsophisticated analyses of risks associated with aggregations of vulnerabilities. In many cases, especially with commercial off the-shelf (COTS) vulnerability scanners, the tools also provide information and guidance on how to mitigate the vulnerabilities they detect.

Archetypical application vulnerability scanners are able to recognize only some of the types of vulnerabilities that exist in large applications: they focus on vulnerabilities that need to be truly remedied versus those that can be mitigated through patching. As with other signature-based scanning tools, application vulnerability scanners can report false positives, unless re-calibrated by the tester. The tester must have enough software and security expertise to meaningfully interpret the scanner's results to weed out the false positives and negatives, so as not to identify as vulnerability what is actually a benign issue, and not to ignore a true vulnerability that has been overlooked by the tool. This is why it is important to combine different tests techniques to examine the software for weakness in a variety of ways, none of which is adequate on

its own, but which in combination can greatly increase the likelihood of vulnerabilities being found [8]. Because automated vulnerability scanners are signature-based, as with virus scanners, they need to be frequently updated with new signatures from their vendor. Two important evaluation criteria for selecting a vulnerability scanner are: (1) how extensive the tool's signature database is, and (2) how often the supplier issues new signatures. Before penetration testing, in order to locate straightforward common vulnerabilities, and thereby eliminate the need to run penetration test scenarios that checks for such vulnerabilities.

VII. PENETRATION TESTING

The alternate name of Penetration testing is ethical hacking. It is a very common technique for testing network security. While penetration testing has proven to be effective in network security, the technique does not naturally translate to applications. Penetration testing is, for the purposes of this guide, the"art" of testing a running application in its"live" execution environment to find security vulnerabilities. Penetration testing observes whether the system resists attacks successfully, and how it behaves when it cannot resist an attack. Penetration testers also attempt to exploit vulnerabilities that they have detected and ones that were detected in previous reviews [2]. Types of penetration testing include black-box, white box and grey box. In black-box penetration testing, the testers are given no knowledge of the application [9]. White-box penetration is the opposite of black-box in that complete information about the application may be given to the testers. Grey-box penetration testing, the most commonly used, is where the tester is given the same privileges as a normal user to simulate a malicious insider [5] Penetration testing should focus on those aspects of system behavior, interaction, and vulnerability that cannot be observed through other tests performed outside of the live operational environment. Penetration testers should subject the system to sophisticated multi-pattern attacks designed to trigger complex series of behaviors across system components, including non-contiguous components. These are the types of behaviors that cannot be forced and observed by any other testing technique.

Penetration testing is used to find security problems that are likely to originate in the software's architecture and design as it is this type of vulnerability that tends to be overlooked by other testing techniques.

VIII. CONCLUSIONS

In this paper, we have described various software security testing techniques that may help to the community of software developer. Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming very challenging job. Many critical software applications and services need integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to find vulnerabilities.

REFERENCES

[1] J. Whittaker and H. Thompson, How to Break Software Security, Addison-Wesley, 2003. Brown Jeremy. "Fuzzing for Fun and Prot." Krakow Labs Literature, 02 Nov. 2009.

[2] Caballero, Yin, Liang, Song, "Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis", In Proceedings of the 14th ACM Conference on Computer and Communication Security, Alexandria, VA, October 2007.

[3] P. Godefroid, M. Levin, D. Molnar. Automated Whitebox Fuzz Testing. NDSS Symposium 2008.San Diego, CA. 10-13 February, 2008.

[4] Charles Miller The legitimate vulnerability market: the secretive world of 0-day exploit sales. Work-shop on the Economics of Information Security (WEIS) 2007. The Heinz School and CyLab at Carnegie Mellon University Pittsburgh, PA (USA). June 7-8, 2007.

[5] P. Godefroid. Compositional Dynamic Test Generation, In Proceedings of POPL-2007, 34th ACM Symposium on Principles of Programming Languages, pages 47-54, Nice, January 2007.

[6] G. McGraw "Software Security," IEEE Security & Privacy, vol. 2, no. 2, 2004, pp. 80-83.

[7] M. A. Hadavi, H. M. Sangchi, V. S. Hamishagi, H. Shirazi "Software Security, A Vulnerability- Activity Revisit", The Third International Conference on Availability, Reliability and Security, 2008.

[8] G. Hoglund and G. McGraw, Exploiting Software, Addison-Wesley, 2004.

[9] Tina R. Knuston "Building Privacy into Software Products and Services", IEEE Security and Privacy, pp.72-74, Mar-Apr 2007.

[10] D. Verndon and G. McGraw, "Risk Analysis in Software Design," IEEE Security & Privacy, vol. 2, no. 4, 2004, pp. 79-84.

[11] Donald G.Firesmith "Security Use Cases", JOURNAL OF OBJECT TECHNOLOGY, Vol. 2, No. 3, May-June 2003.