Software Security Analysis, Metrics, and Test Design Considerations

LJUBOMIR LAZIĆ, DŽENAN AVDIĆ AND ALDINA PLJASKOVIĆ
Department for Mathematics and Informatics
State University of Novi Pazar
SERBIA
llazic@np.ac.rs,dzavdic@np.ac.rs,apljaskovic@np.ac.rs http://www.np.ac.rs

Abstract: - Software security addresses the degree to which software can be exploited or misused. Software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit. Security is a blend of -enhanced processes and practices—and the skilled people to perform them— which are required to build software that can be trusted not to increase risk exposure. Three categories of analysis provide such a blend: threat modeling, risk analysis, and security assessment and testing. This article discusses the role of software testing in a security-oriented software development process. It focuses on two related topics: functional security testing and risk-based security testing. Any endeavor worth pursuing is worth measuring, but software security presents new measurement challenges: there are no established formulas or procedures for quantifying the security risk present in a program. This paper details the importance of measuring software security and discusses the less-than satisfying approaches that are prevalent today. A new set of metrics is then proposed for ensuring an accurate and comprehensive view of software projects ranging from legacy systems to newly deployed web applications. Many of the new metrics make use of source code analysis results.

Key-Words: - Security issues, security testing, security metrics, security risks

1 Introduction

Software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit. Today, as with cancer, vulnerable software can be invaded and modified to cause damage to previously healthy software, and infected software can replicate itself and be carried across networks to cause damage in other systems. Like cancer, these damaging processes may be invisible to the lay person even though experts recognize that their threat is growing. And as in cancer, both preventive actions and research are critical, the former to minimize damage today and the latter to establish a foundation of knowledge and capabilities that will assist the cyber security professionals of tomorrow reduce risk and minimize damage for the long term.

The software development process offers opportunities to insert malicious code and to unintentionally design and build software with exploitable weaknesses. Security-enhanced processes and practices—and the skilled people to perform them—are required to build software that can be trusted not to increase risk exposure. Our research [1][1] concluded that Software testing, as well as Security testing, as a process, has technical, as well as financial aspects [1,2,6,24]. The financial aspects are related to the fact that the available time and resources given to the team are limited. In many cases, thorough security (complete) testing is not achievable because of financial limitations. A company dedicated to software development must complete the project in time, within the budget limits, and satisfy the client's demands [29,30].

Secure software is software that is resistant to intentional attack as well as unintentional failures, defects, and accidents. Software security is the ability of software to resist, tolerate, and recover from events that intentionally threaten its dependability. Security in the Software Lifecycle [5] defines software security as follows:

- Protection against intentional subversion or forced failure.

[1] This work was supported in part by the Ministry of Education and Science of the Republic of Serbia under Grant No. TR-35026 entitled as:“Software Development Environment for optimal software quality design“.

- Preservation of the three subordinate properties that make up security—availability, integrity, and confidentiality.
- Security manifests as the ability of the system to protect itself from external faults that may be accidental or deliberate (attacks).

The objective of software security is to design, implement, configure, and support software systems in ways that enable them to:

1. continue operating correctly in the presence of most attacks by either resisting the exploitation of faults or other weaknesses in the software by the attackers or tolerating the errors and failure that result from such exploits
2. isolate, contain, and limit the damage resulting from any failures caused by attack-triggered faults that the software was unable to resist or tolerate and recover as quickly as possible from those failures

Software security and secure software are often discussed in the context of software assurance. Software assurance is broader than software security, encompassing the additional disciplines of software safety and reliability.

A key objective of software assurance is to provide justifiable confidence that software is free of vulnerabilities. Another is to provide justifiable confidence that software functions in the "intended manner" and the intended manner does not compromise the security and other required properties of the software, its environment, or the information it handles.

This paper discusses the role of software testing in a security-oriented software development process. It focuses on two related topics: functional security testing and risk-based security testing.

Functional testing is meant to ensure that software behaves as it should. Therefore, it is largely based on software requirements. For example, if security requirements state that the length of any user input must be checked, then functional testing is part of the process of determining whether this requirement was implemented and whether it works correctly.

Analogously, risk-based testing is based on software risks, and each test is intended to probe a specific risk that was previously identified through risk analysis. A simple example is that in many web-based applications, there is a risk of injection attacks, where an attacker fools the server into displaying results of arbitrary SQL queries. A risk-based test might actually try to carry out an injection attack, or at least provide evidence that such an attack is possible. For a more complex example, consider the case where risk analysis determines that there are ambiguous requirements. In this case, testers must determine how the ambiguous requirements might manifest themselves as vulnerabilities. The actual tests are then aimed at probing those vulnerabilities.

This paper focuses on how risk-based and functional security testing mesh into the software development process. Many aspects of software testing are discussed, especially in their relationship to security testing. Nonetheless, this article is not intended as a primer on software testing per se.

2 Approaches For Integrating Security Into The SDLC

Regardless of the SDLC model used (e.g. waterfall, spiral, Rational Unified Process), the SDLC represents a phased approach to the development of a system. An appropriate model should facilitate the reanalysis and validation of the plans, requirements, and design at multiple points throughout its life cycle. Whether regarded as a phase or discipline, an SDLC is composed of several common groupings of activities: requirements, design and build, test and deploy, operations and maintenance, and disposal, with full lifecycle support activities such as risk management, CM, and training. Security can be integrated into these different points of the SDLC independent of the model.

Figure 1 describes each phase or discipline of an SDLC with the associated security activities.

Full Life-Cycle Support Activities Risk Management Risk management includes performing security risk analyses at different points of the life cycle. Security risk analysis serves to identify and mitigate security-related risks.

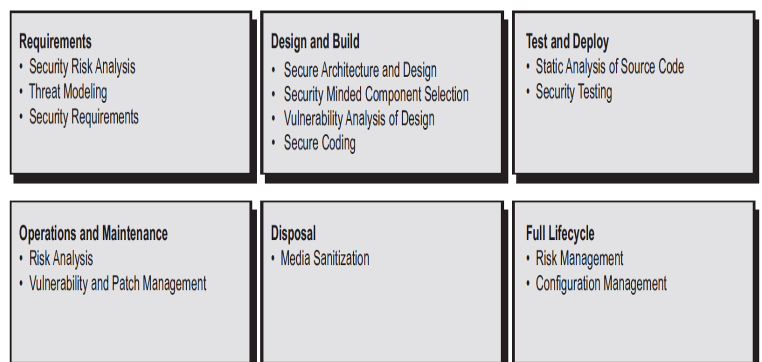

Fig. 1 Phases of an SDLC

The results of the risk analysis feed into the risk management process of identifying, controlling, and eliminating or minimizing uncertain events that may affect the system. Risk analysis should be repeated

iteratively throughout the system's life cycle as different activities allow opportunities to identify new or changing risks. For instance, as the project progresses forward and activities shift from requirements development to high-level system design, additional information will be uncovered about the application. This new information may reveal risks not previously identified such as use of vulnerable components or a flawed authentication model. We also know that changes to design during the build phase are almost always certain to occur. That is why it is important to also perform a risk analysis on the system after it has been built.

Inaccurate or incomplete CM may enable malicious developers to exploit the shortcomings in the CM process in order to make unauthorized or undocumented changes to the software. Lack of proper software change control, for example, could allow rogue developers to insert or substitute malicious code, introduce exploitable vulnerabilities, or remove or modify security controls implemented in the software. Good CM practices also prevent the introduction of unintentional flaws into software code. For example, a developer makes a seemingly harmless modification to the application's interface before deployment and is able to bypass the CM process. This change unintentionally gives normal, restricted users elevated privileges to view information they normally would not be allowed to access. Since the CM process was bypassed, this change was not analyzed or tested for its security impact as it normally should have been.

By tracking and controlling all of the artifacts of the system development process, CM helps ensure that changes made to those artifacts cannot compromise the trustworthiness of the software
as it evolves through each phase of the process. Coming from a systems development background, I have had the opportunity to practice CM hands on. Now that I am involved in security, I have found that good CM is no different than CM for security. Thus, practicing good CM is good security.

3 Metrics That Quantify Software Security Risk

What would happen if your company cut its security budget in half? What if the budget was doubled instead? In most companies today, no one knows the answers to these questions. Security remains more art than science, and nothing is more indicative of this fact than the inability of security practitioners to quantify the effects of their work.

Software security is no exception: nearly every major business critical application deployed today contains vulnerabilities— buffer overflow and cross-site scripting are commonplace, and so are many other, less well-known, types of vulnerabilities.

These problems can be exploited to cause considerable harm by external hackers or malicious insiders. Security teams know that these errors exist, but are, for the most part, will equipped to quantify the problem. Any proposed investment in improving this situation is bound to bring up questions such as:

- Are the applications more secure today than yesterday—or less secure?
- Does security training really make a difference?
- How will we know when our systems are secured?

This paper examines the current state of practice for measuring software security. It then suggests two new approaches to the problem: quantifying the secure development lifecycle, and focusing on the root cause of much vulnerability using metrics built with source code analysis results.

3.1 Approaches to Measuring Security

3.1.1. Build Then Break: Penetration Testing as a Metric

The de facto method that most organizations use for measuring software security today can be summarized as "build then break." Developers create applications with only a minimum of attention paid to security, and the applications are deployed. The operations team then attempts to compensate for the problematic software with perimeter security. When the team takes inventory of all of the ways that data moves through and around the perimeter defenses, it becomes clear that the perimeter security is insufficient. At this point, the operations team may bring in penetration testers to find the problems before hackers or malicious insiders do. The penetration testers generally have a fixed schedule for performing their work and their goal is to find a small number of serious problems to justify their consulting fee. Once these problems are resolved, everyone is happy. But there's no reason to believe that the penetration test revealed all of the problems with the application. In fact, subsequent audits usually prove that it did not. There's also very little feedback to the developers, so penetration tests often find the same types of problems over and over again.

3.1.2. Measure Software Security as Part of Software Quality

A naive approach to software security calls for treating security as just another aspect of software quality. The problem is that traditional quality assurance is aimed at verifying a set of features against a specification. Software security, however, requires much more than well-implemented security features.
The reality is that a typical process for achieving good results with respect to traditional quality issues does not guarantee good results with respect to security issues. In other words, you have to focus specifically on security in order to improve it. Good security is not a byproduct of good quality. Further complicating this approach, the majority of Quality Assurance (QA) departments lack the requisite security expertise to carry out adequate security tests. Finally, any approach to quality that is based on the behavior of regular users will leave many untested opportunities for attackers.

3.1.3. The Feel-Good Metric: If it Hasn't Been Hacked Yet, it's Probably Okay

Because security so often goes un-quantified, the bottom-line measure for security is often gut-feel. Human nature and the nature of security are in conflict on this point: people and organizations tend to gain comfort with the status quo over time, but security may actually degrade as time passes. New types of attacks and new applications for old types of attacks can harm a program's security—even as an organization becomes more and more complacent because security "hasn't been a problem yet!"

A similar fallacy holds that the security of a program can be correlated to the breadth of its adoption. Interestingly, this line of reasoning always seems to work in favor of the status quo.

For applications with a small user base, people assume that attackers will not take an interest. For applications with a large user base, people assume that any security issues will be flushed out of the system shortly after release. In truth, security is no more related to breadth of adoption than it is to longevity. The BugTraq mailing list (where news of many new vulnerabilities debuts) is filled with entries about small and obscure applications. Furthermore, the long history of buffer overflows in widely adopted programs as varied as Send Mail and Internet Explorer shows that neither age nor a large install base prevents attackers from finding new exploits.

3.2. Software Security Metrics You Can Use Imediately

Having explained the measurement problem and how not to solve it, we now turn to two practical methods for measuring software security.

3.2.1. Quantify The Secure Development Lifecycle

Software security must be addressed as part of the software development lifecycle [2,3]. There are practical steps that development groups can take during each phase of the lifecycle in order to improve the security of the resulting system. These steps include:

• Evaluate the current state of software security and create a plan for dealing with it throughout the development life cycle (see Figure 2).
• Specify the threats, identify both business and technical risks, and plan countermeasures.
• Review the code for security vulnerabilities introduced during development.
• Test code for vulnerabilities based on the threats and risks identified earlier.
• Build a gate to prevent applications with vulnerabilities from going into production. Require signoff from key development and security personnel.
• Measure the success of the security plan so that the process can be continually improved. Yes, your measurement efforts should be measured!
• Educate stakeholders about security so they can implement the security plan effectively.

Each of these steps can be measured. For example, if your security plan includes educating developers, you can measure what percentage of developers have received software security training. Of course, not all organizations will adopt all steps to the same degree. By tracking and measuring the adoption of secure development practices, you will have the data to draw correlations within your organization. For example, you will likely find that the up-front specification of threats and risks correlates strongly to a faster and easier security signoff prior to release.

3.2.2. Use Source Code Analysis to Measure Security

All software organizations, regardless of programming language, development methodology, or product category, have one thing in common: they all have source code. The source code is a very direct embodiment of the system, and many vulnerabilities manifest themselves in the source [19]. It follows that the source code is the one key artifact to measure as part of assessing software security. Of course, source code review is useful for more than just metrics. The following sections

discuss some source code analysis fundamentals and then look at how source code analysis results can provide the raw material for powerful software security metrics.

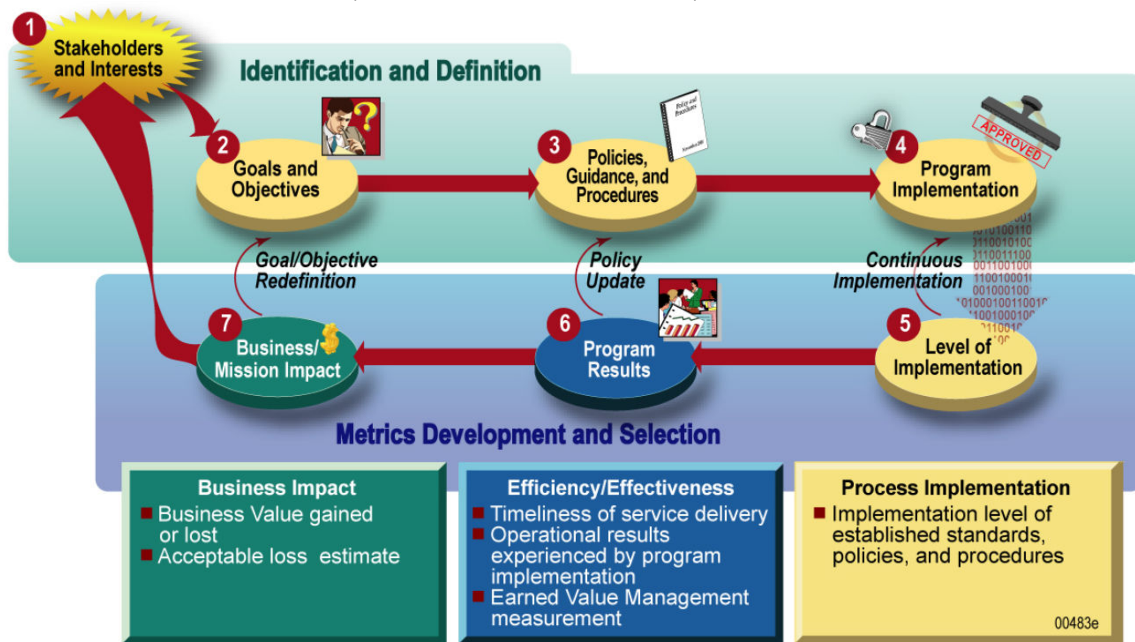

Fig. 2 Information Security Metrics Development Process

Source code analyzers process code looking for known types of security defects. In an abstract sense, a source code analyzer searches the code for patterns that represent potential vulnerabilities and presents the code that matches these patterns to a human auditor for review. The three key attributes for good source code analysis are accuracy, precision, and robustness.

A source code analyzer should accurately identify vulnerabilities that are of concern to the type of program being analyzed. For example, web applications are typically at risk for SQL injection, cross-site scripting, and access control problems, among others. Further, the analysis results should indicate the likely importance of each result.

The source code analyzer must also be precise, pointing to a manageable number of issues without generating a large number of false positives. Furthermore, if a program is analyzed today, and subsequently re-analyzed tomorrow, it is likely that only a small amount of code will have changed. The source code analyzer must be able to give the same name to the same issue today and tomorrow, allowing for the ability to track when issues appear and disappear. This capability is critical for extracting meaningful metrics from source code analysis results.

Finally, the source code analyzer must be robust: it must be able to deal with large, complex bodies of code. Of course, not every issue the source code analyzer identifies will be a true vulnerability. Therefore, part of being robust is allowing human auditors to evaluate and prioritize potential issues. A preferred scenario has a human auditor classify the output from the analyzer into 1) severe vulnerabilities that must be corrected immediately, 2) bad practices, and 3) issues that are not relevant to the organization. An even better application of source code analysis allows developers to analyze their own code as they write it, making source code analysis part of the daily process of program development.

3.2.3. Security Metrics Based on Source Code Analysis

The best metrics that can be derived from source code analysis results are, to a certain extent, dependent upon the way in which an organization applies source code analysis. We will consider the following scenarios:

1. Developers use the source code analyzer on a regular basis as part of their development work. They are proactively coding with security in mind.
2. A software security team uses the source code analyzer as part of a periodic code review process. A large body of code has been created with little regard for security. The organization plans to remediate this code over time.

Of course, the first scenario is preferable, but most organizations cannot achieve that overnight. For the near future, it is likely that both scenarios will co-exist in most organizations.

ISBN: 978-1-61804-126-5

4 The Security Causes and Errors Taxonomy

Software acquirers want assurance that the software products they are obtaining are reviewed for known types of exploitable security weaknesses, and the acquisition groups in large government and private organizations are moving forward to use these types of reviews as part of future contracts. Until recently the tools and services that could be used for this type of review were new at best and there were no nomenclature, taxonomies, or standards to define the capabilities and coverage of them. That made it difficult to comparatively decide which tool or service was best suited for a particular job. What was needed was a standard list and classification of software security weaknesses to serve as a unifying language of discourse and a measuring stick for tools and services. Common Vulnerabilities and Exposures (CWE) [28] is a community-developed formal list or dictionary of common software weaknesses. Leveraging the diverse thinking on this topic from academia, the commercial sector, and government, CWE unites the most valuable breadth and depth of content and structure to serve as a unified standard. Our objective is to help shape and mature the code security assessment industry and also dramatically accelerate the use and utility of software assurance capabilities for organizations in reviewing the software systems they acquire or develop.

4.1 Initial Weaknesses, Idiosyncrasies, Faults, Flaws (WIFFs) Enumeration

The following section introduces the current content we have derived through studying a large portion of the CWE list. The listing below, which is comprised of 290 specific types of weakness, idiosyncrasies, faults and flaws (WIFFs) is not exhaustive and will certainly evolve.

Our purpose in coining the term “WIFFs” is avoid the use of the term “vulnerability” for these items. The term “vulnerability” is frequently used in the community to apply to other concepts including bugs, attacks, threats, risks, and impact. Also, there are widely varying opinions regarding what “risk level” must be associated with a problem in order to call it a vulnerability, e.g. in terms of denial-of-service attacks and minor information leaks.

Finally, not every instance of the items listed below, or those collected in this overall effort, will need to be removed or addressed in the applications they reside in. While they most certainly need to be examined and evaluated for their potential impact to the application, there will certainly be a large number of these items that could be safely left as is, or dealt with by making some minimal adjustments or compensations to keep them from manifesting into exploitable vulnerabilities. If we went forward using the term “vulnerability” for these items, there would be a built-in bias and predisposition to remove and eliminate each and every one of them, which would be a massive and unnecessary waste of time and resources.

The items below have not been categorized except in the most obvious and expeditious manner. With the incorporation of the other contributions from academia and industry sources we will most certainly reorganize these groupings as more examples and specifics are added. With this caveat we provide the following summary of the 28 main categories which contain the 290 individual types of WIFFs we have enumerated to-date.

1. Buffer overflows, format strings, etc. [BUFF] (10 types)

These categories cover the increasingly diverse set of WIFFs that are generally referred to as “buffer overflows.”

The specific types in this group are: Buffer Boundary Violations (“buffer overflow”), Unbounded Transfer (“classic overflow”), Boundary beginning violation (“buffer underflow”), Out-of-bounds Read, Buffer over-read, Buffer under-read, Array index overflow, Length Parameter Inconsistency, Other length calculation error, Format string vulnerability

2. Structure and Validity Problems [SVM] (10 types)

These categories cover certain ways in which “wellformed” data could be malformed. The specific types in this group are: Missing Value Error, Missing Parameter Error, Missing Element Error, Extra Value Error, Extra Parameter Error, Undefined Parameter Error, Undefined Value Error, Wrong Data Type, Incomplete Element, Inconsistent Elements.

3. Special Elements (Characters or Reserved Words) [SPEC] (19 types)

These categories cover the types of special elements (special characters or reserved words) that become security relevant when transferring data between components. The specific types in this group are: General Special Element Problems, Parameter Delimiter, Value Delimiter, Record Delimiter, Line Delimiter, Section Delimiter, Input Terminator, Input Leader, Quoting Element, Escape, Meta, or Control Character / Sequence, Comment Element, Macro Symbol, Substitution Character, Variable Name Delimiter, Wildcard or Matching Element, Whitespace, Grouping Element / Paired Delimiter,

ISBN: 978-1-61804-126-5

Delimiter between Expressions or Commands, Null Character / Null Byte.

4. Common Special Element Manipulations [SPECM] (11 types)

These categories include different ways in which special elements could be introduced into input to software as it operates. The specific types in this group are: Special Element Injection, Equivalent Special Element Injection, Leading Special Element, Multiple Leading Special Elements, Trailing Special Element, Multiple Trailing Special Elements, Internal Special Element, Multiple Internal Special Element, Missing Special Element, Extra Special Element, Inconsistent Special Elements

5. Technology-Specific Special Elements [SPECTS] (17 types)

These categories cover special elements in commonly used technologies and their associated formats. The specific types in this group are: Cross-site scripting (XSS), Basic XSS, XSS in error pages, Script in IMG tags, XSS using Script in Attributes, XSS using Script Via Encoded URI Schemes, Doubled character XSS manipulations, e.g. “<<script”, Null Characters in Tags, Alternate XSS syntax, OS Command Injection, Argument Injection or Modification, SQL injection, LDAP injection, XML injection (aka Blind Xpath injection), Custom Special Character Injection, CRLF Injection, Improper Null Character Termination

6. Pathname Traversal and Equivalence Errors [PATH] (47 types)

These categories cover the use of file and directory names to either “escape” out of an intended restricted directory, or access restricted resources by using equivalent names. The specific types in this group are: Path Traversal, Relative Path Traversal, “/directory/../filename”, “../filedir”, “/../filedir”, “directory/../../filename”, “..\filename” (“dot dot backslash”), “\..\filename” (“leading dot dot backslash”), “\directory\..\filename”, “directory\..\..\filename”, “...” (triple dot), “....” (multiple dot), “....//” (doubled dot dot slash), Absolute Path Traversal, /absolute/pathname/here, “.../...//”, \absolute\pathname\here (“backslash absolute path”), “C:dirname” or C: (Windows volume or “drive letter”), “\\UNC\share\name\” (Windows UNC share), Path Equivalence, Trailing Dot - “filedir.”, Internal Dot -“file.ordir”, Multiple Internal Dot - “file...dir”, Multiple Trailing Dot - “filedir....”, Trailing Space - “filedir “, Leading Space - “ filedir”, file[SPACE]name (internal space), filedir/ (trailing slash, trailing /), //multiple/leading/slash (“multiple leading slash”), /multiple//internal/slash (“multiple internal slash”), /multiple/trailing/slash// (“multiple trailing slash”), \multiple\\internal\backslash, filedir\ (trailing backslash), /./ (single dot directory), filedir* (asterisk / wildcard), dirname/ fakechild /../realchild/filename, Windows 8.3 Filename, Link Following, UNIX symbolic link (symlink) following, UNIX hard link, Windows Shortcut Following (.LNK), Windows hard link, Virtual Files, Windows MSDOS device names, Windows ::DATA alternate data stream, Apple “.DS_Store”, Apple HFS+ alternate data stream

7. Channel and Path Errors [CP] (13 types)

These categories cover the ways in which the use of communication channels or execution paths could be security-relevant. The specific types in this group are:

Channel Errors, Unprotected Primary Channel, Unprotected Alternate Channel, Alternate Channel Race Condition, Proxied Trusted Channel, Unprotected Windows Messaging Channel (“Shatter”), Alternate Path Errors, Direct Request aka “Forced Browsing”, Miscellaneous alternate path errors, Untrusted Search Path, Mutable Search Path, Uncontrolled Search Path Element, Unquoted Search Path or Element

8. Cleansing, Canonicalization, and Comparison Errors [CCC] (16 types)

These categories cover various ways in which inputs are not properly cleansed or canonicalized, leading to improper actions on those inputs. The specific types in this group are:

Encoding Error, Alternate Encoding, Double Encoding, Mixed Encoding, Unicode Encoding, URL Encoding (Hex Encoding), Case Sensitivity (lowercase, uppercase, mixed case), Early Validation Errors, Validate-Before-Canonicalize, Validate-Before-Filter, Collapse of Data into Unsafe Value, Permissive Whitelist, Incomplete Blacklist, Regular Expression Error, Overly Restrictive Regular Expression, Partial Comparison

9. Information Management Errors [INFO] (19 types)

These categories involve the inadvertent or intentional publication or omission of sensitive data, which is not resultant from other types of WIFFs. The specific types in this group are: Information Leak (information disclosure), Discrepancy Information Leaks, Response discrepancy infoleak, Behavioral Discrepancy Infoleak, Internal behavioral inconsistency infoleak, External behavioral inconsistency infoleak, Timing discrepancy infoleak, Product-Generated Error Message Infoleak, Product-External Error Message Infoleak, Cross-Boundary Cleansing Infoleak, Intended information leak, Process information infoleak to other processes, Infoleak Using Debug

ISBN: 978-1-61804-126-5

Information, Sensitive Information Uncleared Before Use, Sensitive memory uncleared by compiler optimization, Information loss or omission, Truncation of Security-relevant Information, Omission of Security relevant Information, Obscured Security-relevant Information by Alternate Name 10. Race Conditions [RACE] (6 types)

These categories cover various types of race conditions. The specific types in this group are: Race condition enabling link following, Signal handler race condition, Time-of-check Time-of-use race condition, Context Switching Race Condition, Alternate Channel Race Condition, Other race conditions

11. Permissions, Privileges, and ACLs [PPA] (20 types)

These categories include the improper use, assignment, or management of permissions, privileges, and access control lists. The specific types in this group are: Privilege /sandbox errors, Incorrect Privilege Assignment, Unsafe Privilege, Privilege Chaining, Privilege Management Error,

Privilege Context Switching Error, Privilege Dropping /Lowering Errors, Insufficient privileges, Misc. privilege issues, Permission errors, Insecure Default Permissions, Insecure inherited permissions, Insecure preserved inherited permissions, Insecure execution-assigned permissions, Fails poorly due to insufficient permissions, Permission preservation failure, Ownership errors, Unverified Ownership, Access Control List (ACL) errors, User management errors.

12. Handler Errors [HAND] (4 types)

These categories, which are not very mature, cover various ways in which “handlers” are improperly applied to data.

The specific types in this group are: Handler errors, Missing Handler, Dangerous handler not cleared/disabled during sensitive, Raw Web Content Delivery, File Upload of Dangerous Type.

13. User Interface Errors [UI] (7 types)

These categories cover WIFFs in a product's user interface that lead to insecure conditions. The specific types in this group are: Product UI does not warn user of unsafe actions, Insufficient UI warning of dangerous operations, User interface inconsistency, Unimplemented or unsupported

feature in UI, Obsolete feature in UI, The UI performs the wrong action, Multiple Interpretations of UI Input, UI Misrepresentation of Critical Information.

5 Software Assessments and Security Testing Framework

Testing traditionally involves exercising an application to see if it works as it should. In contrast, security testing entails identifying and removing vulnerabilities that could result in security violations. It also validates the effectiveness of security measures that are in place [21].

Most of the testing methodologies used fall into one of two categories: black-box or white-box testing. Black-box tests are those whose data are derived from the specified functional requirements in which attention is not given to the final program structure [22, 23]. Commonly used black-box testing approaches for software security are penetration, functional, risk-based, and unit testing.

White-box tests are those tests and assessment activities where the structure and flow of the software under review are visible to the tester. Testing plans are made based on the details of the software implementation and test cases are based on the program structure [21-23].

Commonly used white-box assessment approaches that can assess security are source code analysis and profiling.

The method by which security assessment and testing is carried out depends on the perspective of the tester relative to the software component. We developed OptimalSQM Test Framework Architecture [1] which follows the three-tier architecture:

front-end, middle and target system tiers (Figure 3).

Front-end tier: This provides a user-friendly interface (GUI and/or Web browser), and allows the tester to specify the test scenarios/cases derived from requirements using the textual/graphical representations, query the database, and review test results.

Middle tier: This is divided into two internal tiers: front-middle and back-middle tier. The front-middle tier:

- Organizes scenario specifications in an OO fashion such as creating testing scenario objects, test case objects, input data objects, method signature objects, and complex scenario objects;
- Performs a variety of analysis, such as completeness and consistency check, dependency analysis;
- Executes tests (such as regression tests, functional tests) by sending commands cross the network using TCP/IP or SOAP;
- Performs runtime verification;
- Performs dynamic simulation.

ISBN: 978-1-61804-126-5

The back-middle tier facilitates access to the database for storing test specifications and results.

Target system tier: In this tier, test agents act as proxies of the test master and perform tests on the target system on behalf of the master. The systems can be existing ones or prototypes. Test agents carry out test execution by collaborating with each other, and report the test results to the test master. Our aim is to assist clients in assessing both threat agents and threat events that may affect the organisation as well as identify possible causes and scenarios for each event.

The following formula can be used as a method to identify the level of threat for each threat agent. *Threat Agent = Capabilities + Intentions + Past Activities*

- Capability assessment is based upon the Threat Agent's Resources and Knowledge.
- Intentions assessment is based upon the Threat Agent's Motivation and Incentive.
- Past Activities assessment is based upon historical data relevant to the Threat Agent.

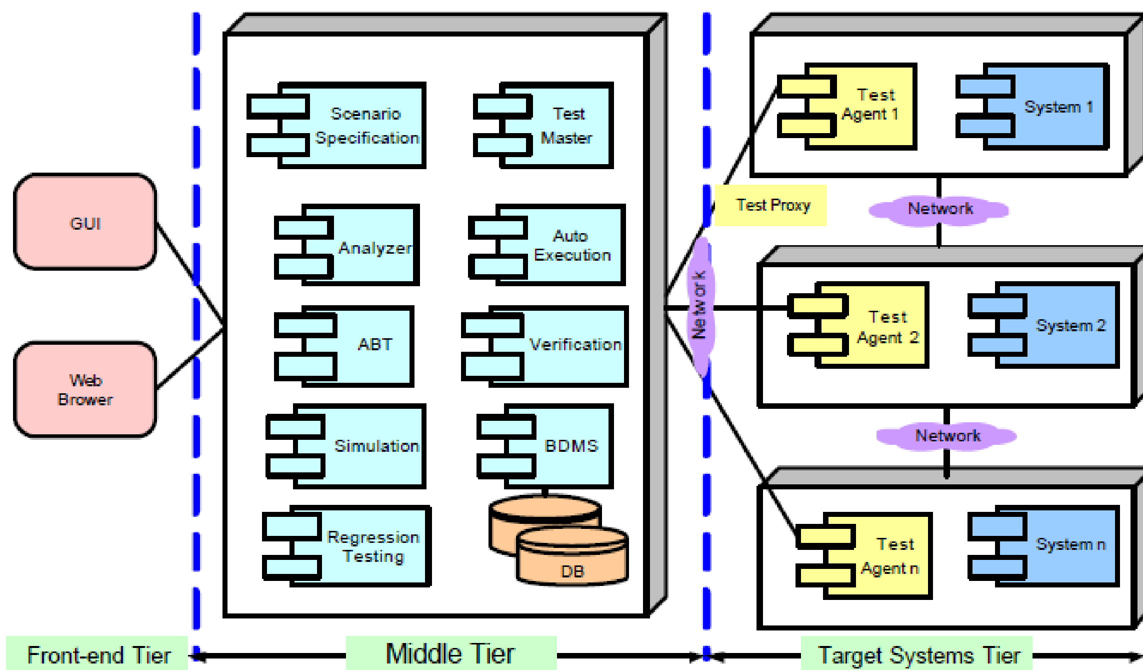

Fig. 3 Overall OptimalSM Architecture of Test Framework

Test cases that are constructed based on functional requirements without regard to specific knowledge about software internals are known as black-box tests; test cases that take advantage of internal structure are known as white-box tests. Often, the information gathered during risk analysis is used to develop white-box and black-box test cases. In particular, flaws identified during risk analysis can be purposely added to a software component to forcibly change the program state and demonstrate the effects of a successfully exploited vulnerability. This approach, known as fault injection, allows for absolute worst-case prediction [24]. It gives an insight into predictive measures such as mean-time-to-hazard, minimum-time-to-hazard, and meantime-to-failure; all of which quantify risk.

Three approaches are commonly taken to test the security of a component in a black-box fashion. Risk-based testing demonstrates that security functionalities work as intended [25].

Penetration testing examines the ease with which a component can be infiltrated.

Unit security testing assumes that adversaries will take a two-stage approach to attack: First, they get access to the software, then second, control the software after access. As such, the assumptions that developers make about the environment and incorporate into the components should be checked at the unit testing level. Attack trees have been used by many as a method for identifying and modeling security threats, especially those that involve many stages for implementation [26].

Two high-leverage white-box techniques for assessing and validating security are source code analysis and profiling. Static analysis tools are used to look at the text of a program while it is not executing so that it can discover vulnerabilities

ISBN: 978-1-61804-126-5

within the program. A fixed set of patterns or rules are used as basis for scanning the source code. For example, many vulnerabilities are known to come from reusable library functions such as stropy() and stat (); so, a static analyzer could scan the programs to see if they contain any calls to those functions. The result of the source code analysis aids in the development of test cases and gives a good perspective of the security posture of the application. White-box testing should be used to verify that the potential vulnerabilities uncovered by the static analysis tool will not lead to security violations [27].

Profiling tools enable the tester to observe the performance of an application while it is running. This provides insight into where performance bottlenecks may be occurring. It also enables the tester to see and understand the sequence of function calls and the time spent in different areas of the software, and thereby brings it to the open areas of vulnerability that are not apparent when using static code analyzers [23].

Although security aspects of software should be tested, it is also important to understand that security is not just a function that can be checked off but is an emergent property of the application. In other words, this would be analogous to saying that being dry is an emergent property of being inside a tent during a rainstorm. The tent only keeps a person dry if the poles are made stable, vertical, and able to support the weight of the wet fabric; the tent also must have waterproof fabric
(without any holes) and be large enough to protect all those who want to remain dry.

4.1. Test Suites

Test Suites are generated by Scenario Specification. One of key activities in testing distributed systems is functional testing, which often involves specification of system behavior scenarios and development of test cases/scripts based on the specified scenarios. Distributed systems often have clear interface through which they interact with each other. For example, in a supply chain system, entities (such customers, retailers, manufacturers, and suppliers) interact with each other through well-defined API (Application Programming Interface). The Web Services Interoperability (WS-I) defines three interaction scenarios in supply-chain web services:

- **One-Way**: a consumer sends a request message to a provider without response from the provider.
- **Synchronous Request/Response**: a consumer sends a request message to a provider. The provider receives the message, processes it and sends back a response.
- **Basic Callback**: At runtime a consumer sends the initial SOAP request in a request/response sequence to the provider, which in turn sends back an immediate acknowledgement. At a later time the provider will initiate the final request/response sequence to the consumer containing the response data for the initial request sent by the consumer.

To derive scenarios of distributed systems, a tester can use the following steps:

Step 1: Derive scenario specification for each sub-system from security point of view, and formalize the scenario specification by annotating each scenario as a sequence of events, actions, and associated pre-/post-conditions vulnerabilities;

Step 2: Specify the interaction between each pair of subsystems;

Step 3: Derive the overall scenarios for the distributed system by combining the scenarios for individual sub-systems with the interaction from security point of view.

Derive Scenarios for Each Sub-System

This step derives scenario specification from subsystem requirements. Each scenario can be classified as an atomic scenario, a sub-scenario, or a complex scenario. The derived scenarios are organized into a tree structure with each sub-tree represent a group of functionally related scenarios that the tester can analyze them together in a hierarchical manner. Scenarios are annotated with pre -conditions, events, actions, and post conditions, and specified using OCL (Object Constraint
Language) and XML. The information specified is useful in various analyses such as dependency analysis, consistency analysis and concurrency analysis. For example, "A customer accesses to the retailer system with a valid customer ID" is a scenario in Retailer system.

Test designer, who test applications that model real security features and vulnerabilities of each identified scenario provide:

• Configurable to be vulnerable to one or many types of attack

• Ability to provide increasing level of defense for a vulnerability.

4.2 Creating Security Assurance Cases

Developing a security assurance case is not a trivial matter. In any real system the number of claims involved and the amount of evidence required will be significant. The effort involved is offset by an expected decrease in effort required to find and fix

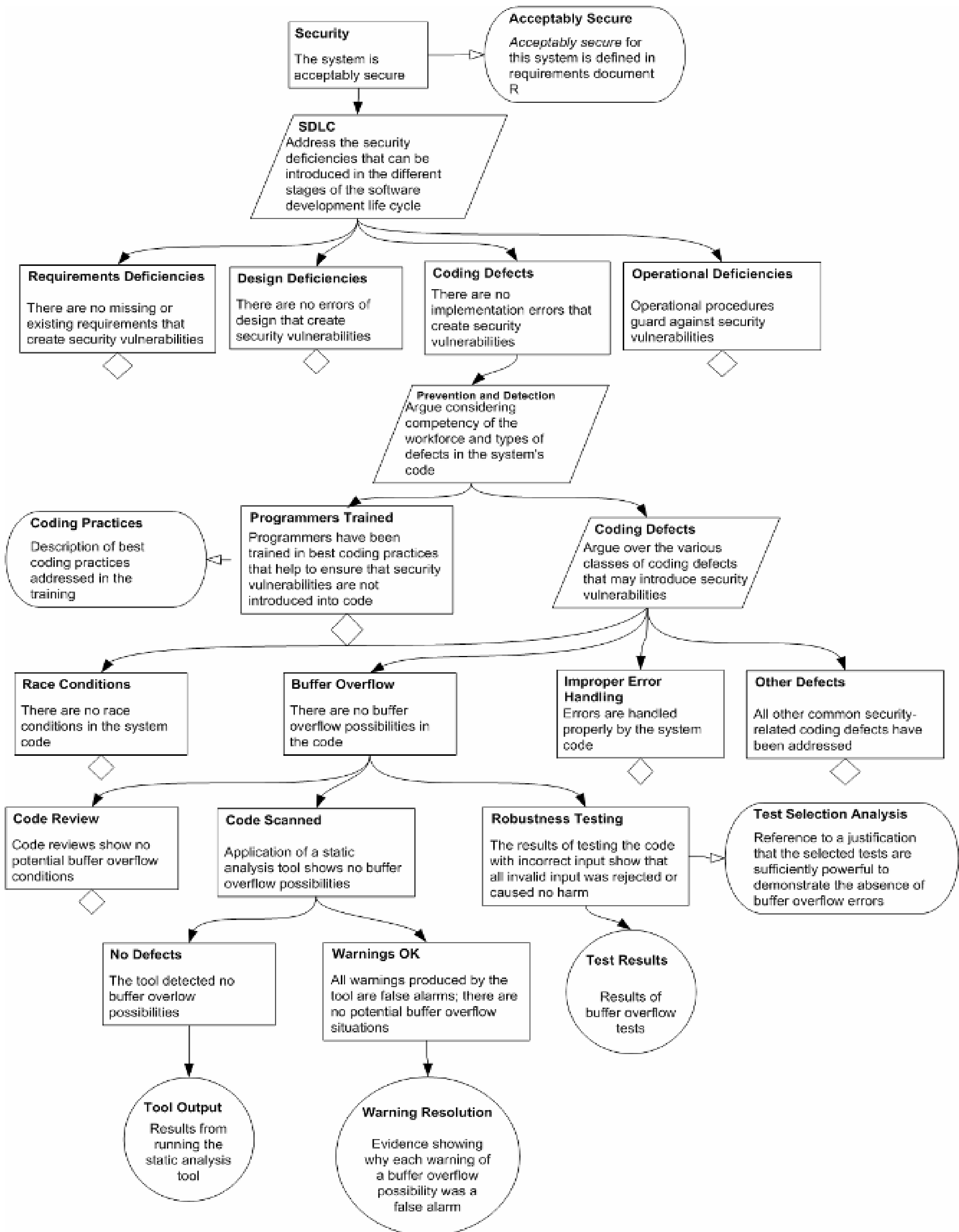

Fig. 4. Partially expanded security assurance case that focuses on buffer overflow

security-related problems at the back end of product development and by a reduced level of security breaches with their attendant costs. Although we believe that the return on investment (ROI) for developing security cases would typically be substantial, work is needed in the community to gather the hard evidence necessary to support this assumption.

ISBN: 978-1-61804-126-5

Creating and evolving the security case as the system is being developed is highly recommended. Developing even the preliminary outlines of an assurance case as early as possible in the software development life cycle (SDLC) can lead to improvement in the development process by focusing attention on what needs to be assured and what evidence needs to be developed at each subsequent stage of the SDLC. Attempting to gather or generate the necessary security case evidence once development is complete may not only be much more costly, it may be impossible.

5 Conclusion

To develop software systems with security as an emergent feature entails that the high leveraged techniques discussed be incorporated into the whole software development life cycle. Threat modeling that drives risk analysis begins with the garnering of requirements and use cases. Risks generated from the threat modeling activities act as a barometer for design, development of tests, and development of rules for software code assessment and as one of the benchmarks for testing.

Software security demands a balance of reactive and proactive measures, and it requires that more time be spent in determining the risks that can or will affect the system. Software systems have to be designed from a high enough level of abstraction with security of the system as an emergent feature of the system in question. The processes utilized to create secure systems need more refinement so that the ubiquity of software is not hampered by inherent insecurity due to poor design. While software security has been a universally recognized risk, there has been an absence of established procedures for quantifying the security risk present software. Only by measuring can organizations conquer the software security problem.

The first step in this journey is the adoption of security-focused activities and deliverables throughout each phase of the software development process. These activities and deliverables include risk analysis during software design, code review during development, and security-oriented testing that targets the risks that are specific to the application at hand. By tracking and measuring the security activities adopted into the development process, an organization can begin to quantify their software security risk.

The data produced by source code analysis tools can be particularly useful for this purpose, giving insight into whether or not code review is taking place and whether or not the results of the review are being acted upon.

This paper presents a scenario-based test framework (OptimalSQM) for rapid distributed system testing. Using the framework, a tester does not need to write testing code, instead focuses on scenario identification and specification. The frame work generates test cases/scripts, and executes them automatically. Whenever a change occurs, the tester just needs to re-specify the modified scenarios so that new test scripts can be generated to test the modified feature. The framework can also perform regression testing by identifying those affected scenarios by dependency analysis. This paper used a SCM application for illustration.

References

[1] Lj. Lazić, N. Mastorakis. ” OptimalSQM:Integrated and Optimized Software Quality Management”, WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS, Issue 10, Volume 6, pp 1636-1664, ISSN: 1790-0832, October 2009.

[2] Burnstein, I., “Practical software testing : a process-oriented approach”, 2003 Springer-Verlag New York, Inc.

[3] Jiwnani K. and Zelkowitz M., “Maintaining Software with a Security Perspective”, IEEE International Conference on Software Maintenance, Montreal Canada, October 2002.

[4] Madan B., Gosseva-Popstojanova K., Vaidyanathan K., and Trivedi K.S.. “Modeling and quantification of security attributes of software systems”. In Proc. Int. Conf. DSN, (IPDS stream), volume 2, pages 505–514, 2002.

[5] Department of Homeland Security. Security in the Software Lifecycle: Making Software Development Processes – and Software Produced by Them – More Secure, draft version 1.2. DHS, August 2006.

[6] Verdon, Denis, and Gary McGraw. “Risk Analysis in Software Design.” IEEE Security and Privacy 2.4 (2004).

[7] Cheswick, B, Paul Kocher, G. McGraw, and A. Rubin. “Bacon Ice Cream: The Best Mix of Proactive and Reactive Security?” IEEE Security and Privacy 1.4 (2003).

[8] McGraw, Gary. “Building Secure Software: Better Than Protecting Bad Software.” IEEE Software 5.7 (2002).

[9] G. Sindre, and A.L. Opdahl. Templates for Misuse Case Description. Proc. Of the Seventh International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ 2001), 4-5 June 2001, Switzerland.

[10] United States. Department of Homeland Security (DHS). National Vulnerability Database 7 Dec. 2006, <http://nvd.nist.gov/>.

[11] OSVDB. Open Source Vulnerability Database. 8 Dec. 2006 <www.osvdb.org>.

[12] Dianxiang Xu, and Kendall Nygard. "A Threat-Driven Approach to Modeling and Verifying Secure Software." Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering ASE, Nov. 2005, Long Beach, CA. New York: ACM Press, 2005.

[13] Schneier, B. "Attack Trees: Modeling Security Threats." Dr. Dobb's Journal Dec. 1999.

[14] McDermott, J.P. "Attack Net Penetration Testing." Proc. of the 2000

[15] Workshop on New Security Paradigm, Sept. 2000. Ballycotton, County Cork, Ireland. New York: ACM Press, 2000.

[16] Gegick, M., and L. Williams. "Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs." Proc. of the 2005 Workshop on Software Engineering For Secure Systems; Building Trustworthy Applications, 15-16 May 2005, St. Louis, MS. New York: ACM Press, 2005 <http://doi.acm.org/10.1145/1083200.1083211>.

[17] Hernan, Shawn, Scott Lambert, Tomasz Ostwald, and Adam Shostack. "Threat Modeling – Uncover Security Design Flaws Using The STRIDE Approach." MSDN Magazine Nov. 2006.

[18] Steel, Christopher, Ramesh Nagappan, and Ray Lai. Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management. Prentice Hall, 2005.

[19] McGraw, Gary. Software Security: Building Security In. Addison-Wesley Professional, 2006.

[20] United States. Department of Commerce. An Introduction to Computer Security – The NIST Handbook. NIST, 1995.

[21] Pan, Jiantao. "Software Testing – 18- 849 Dependable Embedded Systems."

[22] Carnegie Mellon University, 1999 <www.ece.cmu.edu/~koopman/des_s99/sw_testing>.

[23] Howard, Michael, and David C. LeBlanc. Writing Secure Code. 2nd ed. Redmond, WA: Microsoft Press, 2002.

[24] Hetzel, William C. The Complete Guide to Software Testing. 2nd ed. Wellesley, MA: QED Information Sciences, 1988.

[25] Voas, Jeffrey M., and Gary McGraw. Software Fault Injection: Inoculating Programs Against Errors. New York, NY: John Wiley & Sons, 1998.

[26] Michael, C.C., and Will Radosevich. "Risk-Based and Functional Security Testing." DHS. Build Security In Portal, <https://buildsecurityin.us-cert.gov/portal/article/bestpractices/security_testing/overview.xml#Risk-Based-Testing>.

[27] Schneier, B. Secrets and Lies: Digital Security in a Networked World. New York: John Wiley & Sons, 2000.

[28] cwe.mitre.org.

[29] Hope, Paco, Gary McGraw, and Annie I. Anto'n. "Misuse and Abuse Cases: Getting Past the Positive." IEEE Security and Privacy 2.3 (2003).

[30] Defense Information Systems Agency, "Application SecurityAssessment Tool Market Survey," Version 3.0 July 29, 2004.