

2009

PACIFIC NW SOFTWARE
QUALITY
CONFERENCE



MOVING
QUALITY
FORWARD

OCTOBER 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

BEST PRACTICES FOR SECURITY TESTING

TOP 10 RECOMMENDED PRACTICES

Aarti Agarwal

McAfee Software P. Ltd (Enterprise Security Team), Bangalore, India
Tel +1 9980951530 • Email Aarti.Agarwal@gmail.com

Abstract

This White Paper is intended to outline and define the process, procedures and methods that should be used to evaluate the product from a security perspective. The “Top 10” recommended security practices have been prioritized for building confidence in your product, focusing on the most critical areas every enterprise needs to adopt.

About The Author

Aarti Agarwal is a Software Quality Assurance Engineer in McAfee’s Enterprise Security team. She has an experience in Testing and Quality Assurance with a special focus on security testing of client and server applications.

In her present role, she is the research and analysis owner for security testing of McAfee’s Host Intrusion Prevention System that preserves desktops’ and servers’ integrity with signature and behavioral protection from attacks. Her role involves identification of architectural risks, design and implementation risks, information gathering, user interface attacks, file system attacks, design attacks, implementation attacks, test planning, threat model preparation, research on security tools, penetration testing, execution and analysis of results and report metrics preparation.

She has also been associated with Accenture Services Pvt. Ltd, where she gained expertise in Data Warehouse quality assurance for a Financial Organization project.

Copyright Aarti Agarwal August 15, 2009

From a technical and project management perspective, security test activities are primarily performed to validate a system's conformance to security requirements and to identify potential security vulnerabilities within the system. From a business perspective, security test activities are often conducted to reduce overall project costs, protect an organization's reputation or brand, reduce litigation expenses, or conform to regulatory requirements. Identifying and addressing software security vulnerabilities prior to product deployment assists in accomplishing these business goals. Security issues are among the highest concerns to many organizations. Despite this fact, security testing is often the least understood and least defined task.

There are two major aspects of security testing: testing security functionality to ensure that it works and testing the subsystem in light of malicious attack. Security testing is performed by probing undocumented assumptions and areas of particular complexity to determine how a program can be broken. In addition to demonstrating the presence of vulnerabilities, security tests can also assist in uncovering symptoms that suggest vulnerabilities might exist.

This White Paper is intended to outline and define the process, procedures and methods that should be used to evaluate the product from a security perspective. Here, I'll prioritize the "Top 10" recommended security practices for building confidence in your product. These recommendations can be used as a guideline for security testing of system as well as web based products. These guidelines aim to bridge the gap in knowledge and process between vulnerability researchers and quality assurance professionals by bringing together software security and software testing expertise.

1 Changing the Old Paradigm: Creating a better, more secure application development process

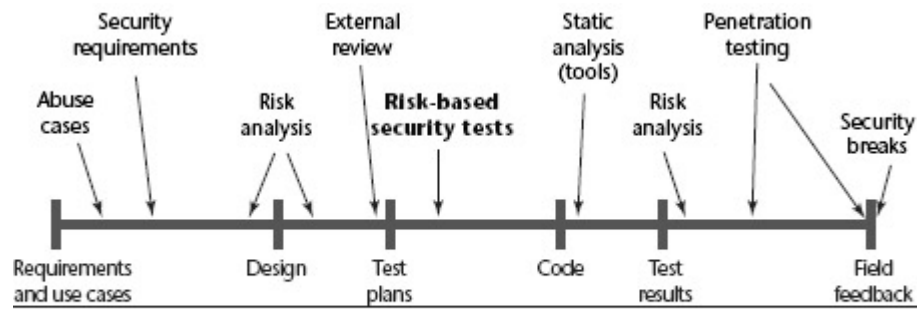
Software security testing is very different from software functionality testing and hence, simple "black-box" testing is not enough. Security best practices should be integrated into the software development lifecycle because they reduce overall costs by increasing efficiency and improve customer satisfaction.

Testing for security vulnerabilities requires a full understanding of how the application is designed, and the ability to creatively think about what is going on inside the application as it is operating. Simple "black-box" testing is not enough.

In order to break this traditional approach, we must fundamentally change the way that we approach application security. We must integrate security throughout the Software Development Life Cycle (SDLC), not just hastily add it to the end.

Security testing in the Software Life Cycle

Integrating security best practices into the software development lifecycle actually reduces overall costs by increasing efficiency and also boosts customer satisfaction. Since risk analysis is an ongoing and fractal process throughout software development, new information is always becoming available for the test plan and test planning becomes an ongoing process as well. The figure below shows the security practices in different phases of the software development life cycle:



Integration of security practices through SDLC

The Start of the Software Life Cycle

The preparations for security testing can start even before the planned software system has definite requirements and before a risk analysis has begun. *For example, past experience with similar systems can provide a wealth of information about how attackers have typically tried to subvert similar systems.*

More generally, the initiation phase makes it possible to start a preliminary risk analysis, asking what environment the software will be subjected to, what its security needs are, and what impact a breach of security might have. If risk analysis starts early, it will provide an early focus for the test planning and security-oriented approach when defining requirements.

Security Testing in the Requirements and Design Phases

The process of software development starts by gathering requirements and developing use cases. In this phase, *test planning* focuses on outlining how each requirement will be tested. Mapping out the elements of the security test plan should begin in the requirements phase of the development life cycle in order to avoid surprises later on. Details on the test plan are to follow later in this paper.

It is also useful for a more detailed risk analysis to begin during the requirements phase. Security analysis often uncovers severe security risks that were not anticipated in the requirements process. The risks identified during this phase may lead to additional requirements that call for features to mitigate those risks. The software development process goes more smoothly if mitigations are defined early in the life cycle, when they can be more easily implemented.

Security Testing in the Test/Coding Phase

Functional security testing generally begins as soon as there is software available to test. A test plan should be in place at the start of the coding phase along with the necessary infrastructure and personnel.

This document does not attempt to catalog every possible testing activity. Instead, it will discuss several broader activities and the role of security testing in each. The activities discussed are as below:

Unit Testing

Unit testing is usually the first stage of testing that a software code goes through. It is where individual classes, methods, functions, or other relatively small components are tested.

When evaluating potential threats to a software component, one should keep in mind that many attacks have two stages: one stage where a remote attacker gains initial access to the machine, and one stage where the attacker gains control of the machine after gaining access. *For example, an attacker who subverts a web server will initially have only the privileges of that web server, but the attacker can now use a number of local attack strategies such as corrupting system resources and running privileged programs in corrupt environments.* Therefore, defense in

depth demands that local threats be addressed on systems that are assumed not to have malicious users, and even on machines that supposedly have no human users at all. Also, assumptions a component makes about its environment should be tested for vulnerability to exploits.

Testing Libraries and Executable Files

In many development projects, unit testing is closely followed by a testing effort that focuses on libraries and executable files.

A greater level of testing expertise is needed for this testing as the testers need to be up to date on the latest vulnerabilities and exploits¹. The test environment can also be complex, encompassing databases, stubs for components that are not yet written, and complex test drivers used to set up and tear down individual test cases. It is useful during functional testing to carry out *code coverage analysis* using a code coverage tool. This helps identify program parts not executed by functional testing. These program parts may include functions, statements, branches, conditions, etc. Such an analysis helps to

- focus and guide testing priorities
- assess the thoroughness of testing
- identify code not executed by tests
- check adequacy of regression test suites
- keep test suites in sync with code changes

Error handling routines are difficult to cover during testing, and they are also notorious for introducing vulnerabilities. Good coding practices can help reduce the risks posed by error handlers, but it may still be useful to have testware² that simulates error conditions during testing in order to exercise error handlers in a dynamic environment.

Libraries also need special attention in security testing keeping in mind: just because a library function is protected by other components in the current design does not mean that it will always be protected in the future. *For example, a buffer overflow³ in a particular library function may seem to pose little risk because attackers cannot control any of the data processed by that function, but in the future the function might be reused in a way that makes it accessible to outside attackers.*

Furthermore, libraries may be reused in future software development projects, even if this was not planned during the design of the current system. This creates additional problems.

1. People who developed the library code might not be available later, and the code may not be well understood anymore
2. Vulnerabilities in the library will have a greater negative impact if it is reused in many systems.
3. If the library is used widely, malicious hackers might have exploits already at hand.

An example of a vulnerable library function is the `strcpy()` (string copy) function in the standard C library, which is susceptible to buffer overflows. For many years, calls to `strcpy()` and similarly vulnerable functions were one of the chief causes of software vulnerabilities in deployed software. By the time `strcpy()`'s vulnerability was discovered, the function was in such widespread use that removing it from the standard C library was not feasible. The story of `strcpy()` contains a moral about the value of catching vulnerabilities as soon as possible and the expense of trying to address vulnerabilities late in the life cycle.

Integration Testing

Integration testing focuses on a collection of subsystems, which may contain many executable components. There

¹ A few recommendations for information on the latest security vulnerabilities: <http://www.securityfocus.com>, <http://www.microsoft.com/technet/security>

² Testware is an umbrella term for all utilities and application software that serve in combination for testing a software package but not necessarily contribute to operational purposes. Testware is produced by both verification and validation testing methods.

³ A buffer overflow occurs when a program or process tries to store more data in a data storage area than it was intended to hold. The extra information can overflow into the runtime stack, which contains control information such as function return addresses and error handlers.

are numerous software bugs that appear only because of the way components interact, and this is true for security bugs as well as traditional ones. Integration errors are often the result of:

- One subsystem making unjustified assumptions about other subsystems
- Failure to properly check input values. During security testing, it is especially important to determine what data can and cannot be influenced by a potential attacker.
- Error handlers because of unusual control and data-flow patterns during error handling. Error handling is one more form of component interaction, and it must be addressed during integration testing.

System Testing

In this stage, the complete system is the artifact that will actually be attacked. *Penetration testing* involves manually or automatically trying to mimic an attacker's actions and checking if any tested scenarios result in security breaches. Penetration testing makes the most sense here, because any vulnerability it uncovers will be a real vulnerability.

Two levels of penetration testing are recommended:

- To determine whether the application is vulnerable to **common attacks**, use penetration testing, which simulates the types of attacks that could be launched on any application.
- To determine whether the application is vulnerable to **unique attacks** (such as attacks on application logic), design and execute penetration tests that attempt to subvert your application's unique security mechanisms. This is critical to verify that an application can resist the application logic attacks that are now becoming increasingly common.

Stress testing is also relevant to security because software performs differently when under stress and many vulnerabilities come out because of conditions that developers were not expecting. *For example, when one component is disabled due to insufficient resources, other components may compensate in insecure ways.* An executable that crashes altogether may leave sensitive information in places that are accessible to attackers. Attackers might be able to spoof subsystems that are slow or disabled, and race conditions might become easier to exploit. Stress testing may also exercise error handlers. Unusual behavior during stress testing might also signal the presence of unsuspected vulnerabilities. *Metasploit* is a good tool in this space.

The Operational Phase

The operational phase of the software life cycle begins when the software is deployed. Often, the software is no longer in the hands of the developing organization or the original testers. Traditionally, beta testing is associated with the early operational phase, and it has its counterpart in the security arena, since white-hat hackers may examine the software in search of vulnerabilities.

However, *beta testing* is not the last word because the software system may *become* vulnerable after deployment. This can be caused by configuration errors or unforeseen factors in the operational environment. Assumptions that were true about those components during development may no longer be true after new versions are deployed. It may also be that only some components are updated while others are not, creating a *mélange* of software versions whose exact composition cannot be foreseen in any development setting. This makes the vulnerabilities of the composite system unforeseeable as well.

The risk profile of the system might also change over time, and this creates a need for continuing *security audits* on systems deployed in the field. This can happen when the system is used in ways that were not foreseen during development, or when the relative importance of different assets grows or diminishes. *For example, an organization might unknowingly protect critical information with encryption methods that are no longer considered secure.*

For new vulnerabilities that are uncovered in existing software versions. A responsible development organization will release a patch, but software users might not apply the patch. This creates a drastic shift in the system's risk profile.

A related issue is that *encryption techniques* used by a software system can become obsolete, either because increasing computational power makes it possible to crack encryption keys or because researchers have discovered ways of breaking encryption schemes previously thought to be secure. These issues create the need for *security audits* in deployed systems.

2 Information Gathering: Software Components and Their Environment

A thorough understanding of the software and its environment is necessary to evaluate the attack surface. Information can be gathered by Runtime Analysis, Application footprinting, OS footprinting, Environmental Interactions and with the help of some appropriate tools.

A thorough understanding of the product's design is necessary. You need to know where all the entry points are so that you can understand and diagram the product's *attack surface*.

Evaluating the attack surface lets you target the areas where an attacker might find an entry point. To understand the high-risk areas of a business application, you also need to know the valuable functions that the program performs and its assets/resources. Typically this requires reviewing the program's design documents, designing a threat model, including data flow diagrams if they exist, and interviewing the program's designers or architects.

Additional attack surface information can be gathered by ***Runtime Analysis***—executing the program and analyzing it with debugging and diagnostic tools to create a block diagram of the system, sketching the major components and the major data flows between them. Of special importance is the data that comes from outside the program that must be considered untrusted. These are the inputs to the system where an attacker can strike. Here are some external data flows:

- Network I/O
- Remote procedure calls (RPCs)
- Querying external systems: domain name system (DNS), database lookups, Lightweight Directory Access Protocol (LDAP)
- File I/O
- Registry
- Named pipes, mutexes, shared memory, any OS object
- Windows messages
- Other operating system calls

Application footprinting

It is the process of discovering what system calls and system objects an application uses. It lets you know how an application receives input from its environment via operating system calls. It also helps to find what system objects the application uses, such as

- Network ports
- Files
- Registry keys

You can use the application footprinting process to validate and add to the information gathered from the runtime analysis.

Windows footprinting

Process Explorer shows you what DLLs and handles a process has opened. As the program executes, handles are created and destroyed. So *Process Explorer* is good at discovering the system objects that the process uses for long periods of time. For shorter-lived objects or object accesses, you need to use other tools.

Process Explorer can view the following handles:

- Desktop
- Directory
- Event
- File
- Process
- Section
- Semaphore
- KeyedEvent
- WindowStation
- Port
- Token
- SymbolicLink
- Timer
- Thread
- Key
- Mutant

Other useful tools for Windows footprinting are *Filemon*, *Regmon* and *Procmon* from the Windows Sysinternals Suite. These tools intercept calls to the Windows file and Registry APIs. *Procmon* also shows real-time file system, registry and process/thread activity.

Unix footprinting

strace, *ktrace*, and *truss* are debugging tools that print a trace of all the system calls made by an application. *strace* is available on Linux. *ktrace* is available on FreeBSD, NetBSD, OpenBSD, and OS X. *truss* is available on Solaris.

It is important to understand the software itself, but it is also important to understand the **environment**. A software module may interact with the user, the file system, and the system memory in fairly obvious ways, but behind the scenes many more interactions may be taking place without the user's knowledge.

Two types of environmental interactions have to be considered, namely, corruption of the software module by the environment and corruption of the environment by the software module. Some of the environmental interactions to be considered include:

- interactions with the user
- interactions with the file system
- interactions with memory
- interactions with the operating system via system calls
- interactions with other components of the same software system
- interactions with dynamically linked libraries
- interactions with other software via APIs
- interactions with other software via interprocess communication
- interactions with global data such as environment variables and the registry
- interactions with the network
- interactions with physical devices
- dependencies on the initial environment

The biggest challenge is to avoid overlooking an interaction altogether. Still, it is appropriate to prioritize these interactions according to how realistic it is to view them as part of an attack.

Some good tools in this space are *Log Parser* for information gathering, *Application Verifier* for runtime analysis, and *Total Uninstall* for monitoring registry and file system changes.

3 Threat evaluation with Data Flow Diagrams

Building and analyzing a DFD helps in understanding how the system works and the threats that your system faces. The STRIDE model is a good way to get started in this space.

A data flow diagram (DFD) is a block diagram. A DFD with security-specific annotations will help to describe how data enters, traverses and leave the system. It shows data sources and destinations, relevant processes that data goes through and trust boundaries in the system.

Be particularly sensitive to data that flows into your control from *external sources*, including: script from web pages, files read from the local computer, user input, and data read from the registry. One good strategy is to consider every kind of threat possible at each data entry point. Microsoft's **STRIDE⁴** model provides good help in evaluating the threats.

4 Prioritizing Security Testing with Threat Modelling

Threat modelling is an iterative procedure for prioritizing security testing. It involves identifying threat paths, threats, vulnerabilities, ranking the risk associated with vulnerability and determining exploitability. Tools can assist in defining strategies for dealing with security issues.

Focus testing on areas where difficulty of attack is least and the impact is highest.

—Chris Wysopal

It is necessary to prioritize the tests to be performed to maximize the quantity and severity of the security flaws uncovered given a fixed amount of time and resources. A technique for prioritizing security testing is *threat modelling*. It is an iterative procedure for optimizing network security by identifying objectives and vulnerabilities, and then defining countermeasures to prevent, or mitigate the effects of, threats to the system.

The threat-modelling process has four main steps:

Identifying Threat Paths

The first stage of the process is to identify the highest-level risks and protection mechanisms for the application you will test. First, overall security strengths of the application platform and implementation language are noted because these will be relevant throughout the threat-modelling process. Next, you need to identify the different user access categories. The table below shows sample user access categories ranked from highest-risk to lowest-risk.

⁴ Spoofing identity, Tampering with data, Repudiation, Information disclosure, Denial of service and Elevation of privilege.

Access Categories	Risk Access Category
Very high	Anonymous remote user
High	Authenticated remote user with file manipulation capability
Medium	Authenticated remote user
Low	Local user with execute privileges
Very low	Administrative local user

Most client/server applications have similar user access categories. Every application needs security testing performed on the threat paths that anonymous and authenticated remote users can access. The next step in identifying the threat paths starts with a DFD as explained earlier.

Identifying Threats

Starting with the highest-risk threat path, the processing performed along the path is analyzed. The following is a list of questions to ask for each processing component along the path:

- What processing does the component perform?
- How does it determine identity?
- Does it trust data or other components?
- What data does it modify?
- What external connections does it have?

Then high-risk activities need to be marked along the path:

- Data parsing
- File access
- Database access
- Network access
- Authentication
- Authorization
- Synchronization or session management
- Handling private data
- Spawning of processes

Use this information to write out the threats along each threat path. This list of threats will be used to drive the vulnerability finding process.

Identifying Vulnerabilities

A threat becomes a vulnerability when the designers fail to build any security features into the application that mitigate that threat. Some of the security mitigations to look for are *data validation testing*, *resource monitoring*, and *access control* for critical functions.

The vulnerability hunt can branch in several directions at this point: detailed security design review, security code review, or security testing. The security design review is best at finding design level vulnerabilities. The security code review and security testing are best at finding coding errors where the programmer did not follow secure coding practices. These three methods of security analysis each benefit from the prioritization done during the threat-modelling procedure.

Ranking the Risk Associated with a Vulnerability

A technique for ranking a threat's severity is to use the DREAD model (introduced by Michael Howard and David Leblanc). You rank a threat using DREAD as follows:

- Damage potential - The extent of the damage if vulnerability is exploited.
- Reproducibility - How often an attempt at exploiting vulnerability works.
- Exploitability - How much effort is required? Is authentication required?
 - authentication required,
 - no authentication but hard-to-determine knowledge required,
 - no authentication and no special knowledge required.

- Affected users - How widespread could the exploit become?
 - only rare configurations,
 - a common case,
 - the default for most users.
- Discoverability (you might not want to use this one if you believe all vulnerabilities are eventually discovered)

Determining Exploitability

When software is in development, it is typically easier to just fix an issue that looks likely to be exploitable than to take the time to determine if it actually is.

Threat Analysis & Modelling (TAM) Tool

Microsoft's TAM tool helps build a picture of threats the systems may face and assists in defining strategies for dealing with security issues. TAM also aids in collectively increasing the security awareness of a team and customers.

Create Test Cases based on Threats Identified in the Threat Model

For each threat identified in your threat models, you should create a test case. The test cases should confirm that the identified threat has been mitigated by the design or implementation of your control.

5 Identifying Vulnerabilities in Source Code

Well-tested code that includes security tests results in an end product that is more robust, easier to maintain and more secure. This can be implemented through security code reviews, source code analysis (can be automated) and observing the behavior of running applications.

The best time to begin looking for security vulnerabilities during development is when writing code. There are coding techniques that produce functional results, yet have unresolved errors in the application and allow intruders to gain unauthorized access.

A few ways to detect vulnerabilities and violations of secure coding practices are through:

- Security code reviews
- Automating the code review process with source code analysis
- Observing the behavior of the running application to discover poor practices such as unencrypted passwords being transferred over the network, storing weakly encrypted passwords in the system, database connections that are opened but not closed and even unhandled exceptions within the application.

Here are a few pointers to keep in mind while identifying vulnerabilities in source code (C/C++):

- For memory management, unbound API functions are particularly dangerous like:
 - strcpy()/tcscopy(), strcat()/tcscat(), the *sprintf()/_tsprintf() family of functions, the *scanf() family of functions, Gets(), strncpy()/_tcsncpy() – these do not guarantee NULL termination
 - strncat()/_tcsncat() - deceptive size argument. Size is not the size of the buffer, it is the size (amount of data) left in the buffer-1 (it does NOT account for NULL byte)
 - (v)snprintf()/_tsnprintf() - subject to return value misinterpretation.
- In pointer arithmetic, the most common error is "off-by-one" mistakes which results in a single byte being written out of bounds.

- In case of (v)sprintf(), using format specifiers like "%n" might extend the amount of output data you can write, thus triggering a buffer overflow.
- To avoid buffer overflows, developers usually enforce a max limit on the input and do not copy more than that. This methodology has the potential side-effect of data being lost.
- In Program Flow, the size parameter usually indicates the size of a buffer in wide characters, not bytes. Incorrectly supplying a size in bytes is easy to do and can result in memory corruption. E.g.: MultiByteToWideChar(), wcsncpy(), wcsncat().

Automating this type of analysis can be faster and more comprehensive than a manual effort and is possible, using a product that observes both the internal calls and data transfers within the application, the operating system and software environment in which it operates.

6 Testing with Known Intrusions

Test the product with the known intrusions, as no security testing regime is complete until you actually try to break into the running application. Automating attack simulation is a good approach.

Even if you have a degree of assurance that the application was developed with good coding practices and security in mind, inadvertent errors or unforeseen consequences of specific practices can create security weaknesses that is not apparent using other types of analyses. So, test the product with the known intrusions.

A better approach is to automate attack simulation, a process whereby software pretends to attack the application with known paths of intrusion. The value of attack simulation is that it verifies the application cannot be breached by known means. If it can, then the problem can be fixed and verified before the application goes into production. Standard tools like *Metasploit* can be used to test the software with known and current intrusions.

7 Creating a Security Test Plan

The Security Test Plan should incorporate a high-level outline of which artifacts are to be tested and what methodologies are to be used, including a general description of the tests themselves, prerequisites, setup, execution, and a description of what to look for in the test results.

The main purpose of a test plan is to organize the testing process, including security testing. It outlines which components of the software system are to be tested and what test procedure is to be used on each one. Often, the test plan has to account for the fact that time and budget constraints prohibit testing every component of a software system; risk analysis is one way of prioritizing the tests.

Test planning is an ongoing process. In creating the test plan, inter-component dependencies must be taken into account so that the potential need for retesting is minimized. The planning process also includes validation of the test environment and the test data, which should be verified during test design and execution.

Within each cycle, the test case specification should map out the *test cases* that should run during the test execution process. A test case typically includes information on the preconditions and post conditions of the test, how the test will be set up, how it will be torn down, and how the output of the test will be evaluated. The test case also defines a test condition, which is the actual state of affairs that is going to be tested. The process of actually devising test conditions is a challenging one for security testing. The test requirements, test case specifications and test procedures can be separate documents. It aids in maintenance, since the test plan changes less than the test specifications, which change less than the test cases.

A typical test plan that includes security considerations might contain the following elements:

- Purpose
- Software Under Test Overview
 - Software and Components
 - Test Boundaries
 - Test Limitations
- Risk Management
 - Synopsis of Risk Analysis
 - Mitigation Plan
 - Contingency Plan (Back-up plans or Worst-case scenario plans)
- Test Strategy
 - Assumptions
 - Test Approach
 - Items Not To Be Tested
- High Level Test Requirements
 - Functionality Test Requirements
 - Security Test Requirements
 - Installation/Configuration Test Requirements
 - Stress and Load Test Requirements
 - User Documentation
 - Personnel Requirements
 - Facility and Hardware Requirements
- Test Environment
- Test Case Specifications
 - Unique Test Identifier
 - Requirement Traceability (what requirement number from requirement document does test case validate)
 - Input Specifications
 - Output Specifications/Expected Results
 - Environmental Needs
 - Special Procedural Requirements
 - Dependencies Among Test Cases
- Test Automation and Testware
 - Testware Architecture
 - Test Tools
 - Testware
- Test Execution
 - Test Entry Criteria
 - QA Acceptance Tests
 - Regression Testing
 - Test Procedures(Special requirements, Procedure steps)
 - Test Schedule
 - Test Exit Criteria
- Test Management Plan
- Definitions and Acronyms

8

Choosing the Right Tool for Security Testing

Choosing an adequate tool is very important. Automated security testing is a comprehensive approach that spiders the entire application to identify security holes but can also produce false positives. Manually crawling a web application can be time consuming, but it also helps ensure that specific pages are tracked and analyzed.

The QA department will need application security testing software that can perform three types of testing—as a *non-authenticated user*, an *authenticated user* and an *administrative user*—to determine the vulnerabilities inherent in each user class. Additionally, the web application security tool should be able to perform crawling/spidering⁵ of your Web application.

Manual security testing allows a user to focus on specific pathways or tasks on a web site while the software follows silently behind, tracking the process. The program can then audit the particular path that the user has taken for security vulnerabilities and provide a report. Manually crawling an application can be time consuming, but it also helps ensure that specific pages are tracked and analyzed.

Automated security testing will spider the entire application by clicking every button and link, filling out data fields to identify the structure of the program, and then audit each page for vulnerabilities. It should do this from the outside in, reviewing each portion of the site the way an external hacker might, ideally from behind the scenes. On the down side, it can also produce false positives, and it may not be able to access all of your Web pages due to the way certain pages are coded.

Here is a special mention of a tool, *BackTrack*, which contains a free suite of open source security tools and is very logically structured according to the work flow of security professionals. Currently it provides a graphical environment with **more than 300 different up-to-date tools**. The major tool categories are:

- Information Gathering
- Network Mapping
- Vulnerability Identification
- Penetration
- Privilege Escalation
- Reverse Engineering
- Maintaining Access
- Radio Network Analysis
- VOIP & Telephony Analysis
- Digital Forensics
- Miscellaneous

⁵ A Web crawler is a computer program that browses the World Wide Web in a methodical, automated manner. This process is called Web crawling or spidering.

9 Relevant Metrics

Both the quantity and quality of testing needs to be measured to help make a quantifiable assessment about the efficiency of the testing performed on the software. The metrics that are useful include test requirements criteria, test coverage criteria, test case metrics, defect metrics and test case quality metrics.

Unfortunately, there are no industry-standard metrics for measuring test quantity and test quality. In general, test metrics are used as a means of determining when to stop testing and when to release the software to the customer. The test criteria to be considered and the rationale behind using these criteria are described below.

• **Requirements test criteria:** One of the purposes of testing is to demonstrate that the software functions as specified by the requirements. Thus every software requirement must be tested by at least one corresponding test case. By having traceability from test cases to functional requirements, one can determine the percentage of requirements tested to the total number of requirements. Thus, *test requirement metric* is defined as the ratio of requirements tested to the total number of requirements.

• **Test coverage criteria:** Testing should attempt to exercise as many "parts" of the software as possible, since code that is not exercised can potentially hide faults. Exercising code comprehensively involves covering *all* the execution paths through the software. Unfortunately, the number of such paths in even a small program can be astronomical, and hence it is not practical to execute all these paths. The next best thing is to exercise as many statements, branches, and conditions as possible. Thus, we can define the test coverage criteria such as statement coverage, branch coverage, etc. *Commercial off-the-shelf (COTS) coverage tools* can be used to monitor coverage levels as testing progresses.

• **Test case metrics:** This metric is useful when it is plotted with respect to time. Essentially, such a plot will show the total number of test cases created, how many have been exercised, passed and failed over time. These project-level metrics provide a snapshot of the progress of testing where the project is today and what direction it is likely to take tomorrow.

• **Defect metrics:** These metrics are very useful when plotted with respect to time. It is quite informative to plot cumulative defects found since the beginning of testing to the date of the report. The slope of this curve provides important information. If the slope of the curve is rising sharply, large numbers of defects are being discovered rapidly, and certainly testing efforts must be maintained, if not intensified. In contrast, if the slope is leveling off, it might mean fewer failures are being found. It might also mean that the quality of testing has decreased. Defect metrics also include measures of the following criteria as an information for analysis:

- total open, by priority
- show-stopper / stop-ship problems
- find rate, by severity (daily and 15-day rolling average)
- resolution type versus severity for all resolved problems (total and last 15 days)
- unreviewed problems, by priority
- unverified resolutions (QA backlog)
- reopen rate, over time (number of problems with resolutions that were rejected by QA, and thus reopened rather than concluded).

• **Test case quality metric:** Test cases "fail" when the application under test produces a result other than what is expected. This can happen due to several reasons, one being a true defect in the application. Other reasons could be a defect in the test case itself, a change in the application, or a change in the environment. Test

case quality metric can be defined as the ratio of test case failures resulting in a defect to the total number of test case failures.

The most important of these metrics are defect metrics. Defect metrics must be collected and carefully analyzed in the course of the project.

10 Reporting

All time-oriented metrics should be measured on more or less a daily scale. All problems found by anyone during any phase of testing must be logged in the database. Defect metrics are vital to the successful management of a high assurance test project and hence needs to be used properly.

All time-oriented metrics should be measured on a daily scale. These metrics should be automated so that they are updated on a daily basis. The reports should be available for the whole system and on the level of individual product areas.

The major reports to produce include Test case metrics, Defect metrics and Test case quality metrics (as detailed above in section 9). Others may be useful, depending on the circumstances

Problem Tracking

The problem database should be maintained primarily by the test team. All problems found by anyone during and after functional testing must be logged in the database. This protocol can be enforced easily by requiring that all changes made after code freeze be associated with a problem number. Prior to each build, the change list is then compared to the problem tracking system. Problems found by customer engineering, and ultimately the customer, are also important to track.

How To Use the Metrics

Defect metrics are vital to the successful management of a high assurance test project. Here are a few important points to keep in mind:

- Defect metrics should be used on more or less a daily basis, to review the status of the product, scan for risks, and guide the testing effort toward areas of greatest risk.
- The critical and severe open problem reports should be resolved and verified before the product is shipped.
- A find rate requirement (e.g., no defects found for a week) is not recommended for shipping the product. Reviewing each problem found and using that information to question the efficacy of the test effort and to re-analyze technical risk is recommended. Keeping the important problem find rate as high as possible throughout the project is an important goal of the test team.
- The reopen rate should start low (less than 1 out of 10 resolutions rejected) and get steadily lower over the course of the project.

Conclusion

The importance of security is no secret in the world of software today. The stakes are incredibly high and as products grow in complexity and sophistication, so has securing and operating software become more challenging and demanding.

The pressure and more importantly the need to build software products with improved security features to fulfill their economic promise and protect organizations against liability and loss is more than it has ever been.

Software security, today, needs a highly efficient methodology to plan, develop, test and maintain the softwares which are under constant attack by the hackers with malicious intent. My "Top 10 best practices" recommendations is an effort towards the development of one such methodology and focuses on some of the most critical areas that every enterprise needs to adopt. As the technology and thus security threats are constantly evolving, no solutions can promise a foolproof secure system but following some best practices like the ones mentioned here, we can definitely take a respectable stand against, and possibly even get ahead of, the cyber criminals who are already planning their next moves.

References

- MCGRAW, Gary: *Software Security* (Addison-Wesley; 2006).
- J. Whittaker and H. Thompson: *How to Break Software Security*, Addison-Wesley, 2003.
- <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box/261-BSI.html>.
- <http://opensource-testing.org>: *Open source testing tools, news and discussion*.
- <http://msdn.microsoft.com/security/securecode/threatmodeling/acetm/>.
- Ghosh, Anup K.; O'Connor, Tom; & McGraw, Gary: *An Automated Approach for Identifying Potential Vulnerabilities in Software*.
- Oakland, California, May 3-6, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998: *Security and Privacy*.
- TechRepublic Publication: *Risk Based Security Testing*.
- Grance, T.; Myers, M.; & Stevens, M: *Security Considerations in the Information System Development Life Cycle*, (NIST Special Publication 800-64), 2004.
- December 2007, HP: *Pillars of Application Quality: Security, Functional, and Performance Testing*.
- Peter Varhol, Technology Strategy Research, LLC: *Building Secure Web Applications*.
- Bruce potter and Gary McGraw: *Software Security Testing*.
- C. C. Michael and Will Radosevich: *Risk-Based and Functional Security Testing*.
- Guttman, Barbara; Roback, Edward: *An Introduction to Computer Security*.
- Gearhead Product Security Trainings for Whitebox Testing.
- Wenliang Du & Aditya P. Mathur: *Testing for Software Vulnerability Using Environment Perturbation*. Center for Education and Research in Information Assurance and Security (CERIAS), CERIAS Center and Software Engineering Research Center (SERC).
- Du, W. & Mathur, A. P.: *Vulnerability Testing of Software System Using Fault Injection* (COAST technical report). West Lafayette, IN: Purdue University, 1998.
- Dustin, Elfriede; Rashka; Jeff; McDiarmid, Douglas; & Nielson, Jakob: *Quality Web Systems: Performance, Security, and Usability*.
- Wack, J.; Tracey, M.; & Souppaya, M.: *Guideline on Network Security Testing* (NIST Special Publication 800-42), 2003.
- NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Planning Report 02-3). Gaithersburg, MD: National Institute of Standards and Technology, 2002.
- Howard, Michael & LeBlanc, David: *Writing Secure Code*, 2nd ed. Redmond, WA: Microsoft Press, 2002.
- Hogg, Greg & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley, 2004.
- Grance, T.; Myers, M.; & Stevens, M. *Security Considerations in the Information System Development Life Cycle* (NIST Special Publication 800-64), 2004.
- Graff, Mark G. & Van Wyk, Kenneth R. *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596002424).
- Viegas, John & Messier, Matt. *Secure Programming Cookbook for C and C++*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596003943).
- G. Hogg and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.