Chapter 5

Considerations for Secure Coding and Testing

5.1 Introduction

This chapter provides an overview of key security practices that project managers should include during software coding and testing. A number of excellent books and Web sites also provide detailed guidance on software security coding and software security testing. Thus the intent here is to summarize considerations for project managers and provide references for further reading.

Software security is first and foremost about identifying and managing risks. Assuming that appropriate requirements engineering, design, and architecture practices have been implemented, the next most effective way to identify and manage risks for a software application is to iteratively analyze and review its code throughout the course of the SDLC. In fact, many project managers start here because code analysis and review is better defined, more mature, and, therefore, more commonly used than some of the earlier life-cycle practices. This chapter identifies some of the more common software code vulnerabilities and effective

practices for conducting source code review. It also briefly introduces the topic of practices for secure coding, and provides supporting references for further investigation.

The description of software security testing compares and contrasts software testing with testing software with security in mind. It describes two accepted approaches for software security testing: functional testing and risk-based testing. The chapter closes by describing practices and approaches to be considered when addressing security during unit test (including white-box testing), the testing of libraries and executable files, integration testing, and system testing (including black-box and penetration testing).

5.2 Code Analysis[1]

Developing robust software applications that are predictable in their execution and as vulnerability free as possible is a difficult task; making them completely secure is impossible. Too often software development organizations place functionality, schedules, and costs at the forefront of their concerns and make security and quality an afterthought. Nearly all attacks on software applications have one fundamental cause: The software is not secure owing to defects in its design, coding, testing, and operations.

A vulnerability is a software defect that an attacker can exploit. Defects typically fall into one of two categories: bugs and flaws.

A bug is a problem introduced during software implementation. Most bugs can be easily discovered and corrected. Examples include buffer overflows, race conditions, unsafe system calls, and incorrect input validation.

A flaw is a problem at a much deeper level. Flaws are more subtle, typically originating in the design and being instantiated in the code. Examples of flaws include compartmentalization problems in design, error-handling problems, and broken or illogical access control.

In practice, we find that software security problems are divided 50/50 between bugs and flaws [McGraw 2006]. Thus discovering and

1. This material is extracted and adapted from a more extensive article by Steven Lavenhar of Cigital, Inc. [BSI 19]. That article should be consulted for additional details and examples.

eliminating bugs during code analysis takes care of roughly half of the problem when tackling software security. Attack patterns, as discussed in Chapter 2, can also be used effectively during coding to help enumerate specific weaknesses targeted by relevant attacks, allowing developers to ensure that these weaknesses do not occur in their code.

This section focuses on implementation-level security bugs that can be addressed during source code analysis. Design-level flaws are discussed in Chapter 4, Secure Software Architecture and Design.

5.2.1 Common Software Code Vulnerabilities

The use of sound coding practices can help to substantially reduce software defects commonly introduced during implementation. The following types of security bugs are common. More details are available in [McGraw 2006] and [Tsipenyuk 2005] as well as the Common Vulnerabilities and Exposures Web site [CVE 2007], the Common Weakness Enumeration Web site [CWE 2007], and the National Vulnerability Database [NIST 2007].

Common Security Bugs and Attack Strategies with Known Solution Approaches

- Incorrect or incomplete input validation
- Poor or missing exception handling
- Buffer overflows
- SQL injection
- Race conditions

Input Validation

Trusting user and parameter input is a frequent source of security problems. Attacks that take advantage of little to no input validation include cross-site scripting, illegal pointer values, integer overflows, and DNS cache poisoning (refer to the glossary for definitions of these

types of attacks). In addition, inadequate input validation can lead to buffer overflows and SQL defects as described below. All of these types of attacks can pose risks to confidentiality and integrity. One of the more effective approaches for input validation is to use a whitelist, which lists all known good inputs that a system is permitted to accept and excludes everything else (including characters used to perform each type of attack).

Exceptions

Exceptions are events that disrupt the normal flow of code. Programming languages may use a mechanism called an exception handler to deal with unexpected events such a divide-by-zero attempt, violation of memory protection, or a floating-point arithmetic error. Such exceptions could be handled by the code by checking for conditions that can lead to such violations. When such checks are not made, however, exception handling passes control from the function with that error to a higher execution context in an attempt to recover from that condition. Such exception handling disrupts the normal flow of the code. The security concerns that arise from exception handling are discussed in [McGraw 2006].

Buffer Overflows

Buffer overflows are a leading method used to exploit software by remotely injecting malicious code into a target application [Hoglund 2004; Viega 2001]. The root cause of buffer overflow problems is that commonly used programming languages such as C and C++ are inherently unsafe. No bounds checks on array and pointer references are carried out, meaning that a developer must check the bounds (an activity that is often overlooked) or risk encountering problems.

When writing to buffers, C/C++ programmers must take care not to store more data in the buffer than it can hold. When a program writes past the bounds of a buffer, a buffer overflow occurs and the next contiguous chunk of memory is overwritten. C and C++ allow programs to overflow buffers at will. No runtime checks are performed that might prevent writing past the end of a buffer, so developers have to perform the checks in their own code.

Reading or writing past the end of a buffer can cause a number of diverse (and often unanticipated) behaviors: (1) Programs can act in

strange ways, (2) programs can fail completely, and (2) programs can proceed without any noticeable difference in execution. The side effects of overrunning a buffer depend on the following issues:

- How much data is written past the buffer bounds
- What data (if any) is overwritten when the buffer gets full and spills over
- Whether the program attempts to read data that is overwritten during the overflow
- Which data ends up replacing the memory that gets overwritten

The indeterminate behavior of programs that have overrun a buffer makes them particularly tricky to debug. In the worst cases, a program may overflow a buffer and not show any adverse side effects at all. As a result, buffer overflow problems often remain invisible during standard testing. The important thing to realize about buffer overflows is that any data that happens to be allocated near the buffer can potentially be modified when the overflow occurs.

Memory usage vulnerabilities will continue to be a fruitful resource for exploiting software until languages that incorporate memory management schemes enter into wider use.

SQL Injection

SQL injection is currently the principal technique used by attackers to take advantage of nonvalidated input defects to pass SQL commands through an application for execution by a database. The security model used by many applications assumes that a SQL query is a trusted command. In this case, the defect lies in the software's construction of a dynamic SQL statement based on user input.

Attackers take advantage of the fact that developers often chain together SQL commands with user-provided parameters, meaning that the attackers can, therefore, embed SQL commands inside these parameters. As a result, the attacker can execute arbitrary SQL queries and/or commands on the database server through the application. This ability enables attackers to exploit SQL queries to circumvent access controls, authentication, and authorization checks. In some instances, SQL queries may allow access to commands at the level of the host operating system. This can be done using stored procedures.

Race Conditions

Race conditions take on many forms but can be characterized as scheduling dependencies between multiple threads that are not properly synchronized, causing an undesirable timing of events. An example of a race condition that could have a negative outcome on security is when a specific sequence of events is required between Event A and Event B, but a race occurs and the proper sequence is not ensured by the software program. Developers can use a number of programming constructs to control the synchronization of threads, such as semaphores, mutexes, and critical sections. Race conditions fall into three main categories:

- Infinite loops, which cause a program to never terminate or never return from some flow of logic or control
- Deadlocks, which occur when the program is waiting on a resource without some mechanism for timeout or expiration and the resource or lock is never released
- Resource collisions, which represent failures to synchronize access to shared resources, often resulting in resource corruption or privilege escalations (see [Bishop 1996])

Additional security concerns that arise from these and other types of software vulnerabilities are discussed in [McGraw 2006].

5.2.2 Source Code Review

Source code review for security ranks high on the list of sound practices intended to enhance software security. Structured design and code inspections, as well as peer review of source code, can produce substantial improvements in software security. You can easily integrate these reviews into established software development processes. In this type of review, the reviewers meet one-on-one with developers and review code visually to determine whether it meets previously established secure code development criteria. Reviewers consider coding standards and use code review checklists (refer to Section 5.3.1) as they inspect code comments, documentation, the unit test plan, and the code's compliance with security requirements. Unit test plans detail how the code will be tested to demonstrate that it meets security requirements and design/coding standards intended to reduce design flaws and implementation bugs. The test

plan includes a test procedure, inputs, and expected outputs [Viega 2001]. (See also Section 5.5.1.)

Manual inspection of code for security vulnerabilities can be time-consuming. To perform a manual analysis effectively, reviewers must know what security vulnerabilities look like before they can rigorously examine the code and identify those problems. The use of static analysis tools is preferred over manual analysis for this purpose because the former tools are faster, can be used to evaluate software programs much more frequently, and can encapsulate security knowledge in a way that does not require the tool operator to have the same level of security expertise as a human reviewer. Nevertheless, these tools cannot replace a human analyst; they can only speed up tasks that are easily automated.

Static Code Analysis Tools[2]

Static source code analysis is the process by which software developers check their code for problems and inconsistencies before compiling it. Developers can automate the analysis of source code by using static analysis tools. These tools scan the source code and automatically detect errors that typically pass through compilers and can cause problems later in the SDLC.

Many modern static analysis tools generate reports that graphically present the analysis results and recommend potential resolutions to identified problems.

Identifying security vulnerabilities is complicated by the fact that they often appear in hard-to-produce software states or crop up in unusual circumstances. Static analysis has the advantage of being performed before a program reaches a level of completion where dynamic analysis or other types of analysis can be meaningfully used. However, static code analyzers should not be viewed as a panacea to all potential problems. These tools can produce false positives and false negatives, so their results should be taken with the proverbial "grain of salt." That is, results indicating that zero security defects were found should not be taken to mean that your code is completely free of vulnerabilities or 100 percent secure; rather, these results simply mean that your code has none of the patterns found in the analysis tool's rulebase for security defects.

2. See [McGraw 2006, appendix A], [Chess 2004], [Chess 2007], and http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis for further details and several examples.

What Static Analysis Tools Find

Static analysis tools look for a fixed set of patterns or rules in the code in a manner similar to virus-checking programs. While some of the more advanced tools allow new rules to be added to the rulebase, the tool will never find a problem if a rule has not been written for it.

Here are some examples of problems detected by static code analyzers:

- Syntax problems
- Unreachable code
- Unconditional branches into loops
- Undeclared variables
- Uninitialized variables
- Parameter type mismatches
- Uncalled functions and procedures
- Variables used before initialization
- Non-usage of function results
- Possible array bound errors
- Misuse of pointers

The greatest promise of static analysis tools derives from their ability to automatically identify many common coding problems. Unfortunately, implementation bugs created by developer errors are often only part of the problem. Static analysis tools cannot evaluate design and architectural flaws. They cannot identify poorly designed cryptographic libraries or improperly selected algorithms, and they cannot point out design problems that might cause confusion between authentication and authorization. They also cannot identify passwords or magic numbers embedded in code. One further drawback to automated code analysis is that the tools are prone to producing false positives when a potential vulnerability does not exist. This is especially true of older freeware tools, most of which are not actively supported; many analysts do not find these tools to

be useful when analyzing real-world software systems.[3] Commercial tool vendors are actively addressing the problem of false positives and have made considerable progress in this realm, but much remains to be done.

Static code analysis can be used to discover subtle and elusive implementation errors before the software is tested or placed into operation. By correcting subtle errors in the code early, project managers can reduce testing efforts and minimize operations and maintenance costs. Static code analysis tools can be applied in a variety of ways, all of which lead to higher-quality software. This said, static analysis tools can identify only a subset of the vulnerabilities leading to security problems. These tools must always be used in conjunction with manual analysis and other software assurance methods to reduce vulnerabilities that cannot be identified based on patterns and rules.

Metric Analysis

Metric analysis produces a quantitative measure of the degree to which the analyzed code possesses a given attribute. An attribute is a characteristic or a property of the code. For example,

> When considered separately, "lines of code" and "number of security breaches" are two distinct measures that provide very little business meaning because there is no context for their values. A metric made up as "number of breaches/lines of code" provides a more interesting relative value. A comparative metric like this can be used to compare and contrast a given system's "security defect density" against a previous version or similar systems and thus provide management with useful data for decision making. [McGraw 2006, p. 247]

The process of using code metrics begins by deriving metrics that are appropriate for the code under review. Then data is collected, and metrics are computed and compared to preestablished guidelines and historical data (such as the number of defects per 1000 lines of code). The results of these comparisons are used to analyze the code with the intent of improving the measured qualities.

Two classes of quantitative software metrics are distinguished: absolute and relative. Absolute metrics are numerical values that represent

3. See Cigital's ITS4 software security tool (http://www.cigital.com/its4) and Fortify Software's RATS (Rough Auditing Tool for Security) (http://www.fortifysoftware.com/security-resources/rats.jsp).

a characteristic of the code, such as the probability of failure, the number of references to a particular variable in an application, or the number of lines of code. Absolute metrics do not involve uncertainty. There can be one and only one correct numerical representation of a given absolute metric. In contrast, relative metrics provide a numeric representation of an attribute that cannot be precisely measured, such as the degree of difficulty in testing for buffer overflows. There is no objective, absolute way to measure such an attribute. Multiple variables are factored into an estimation of the degree of testing difficulty, and any numeric representation is just an approximation.

Code Analysis Process Diagrams

The BSI Web site provides a number of code analysis process flow diagrams for source code review, static code analysis, and metric analysis, as well as for dynamic analysis, fault injection, cryptanalysis, and random-number generator analysis. We encourage you to consult the Web site [BSI 19] and [McGraw 2006] for further details.

5.3 Coding Practices[4]

Coding practices typically describe methods, techniques, processes, tools, and runtime libraries that can prevent or limit exploits against vulnerabilities. These measures may include the development and technology environment in which the coding practice is applied, as well as the risk of not following the practice and the type of attacks that could result.

Secure coding requires an understanding of programming errors that commonly lead to software vulnerabilities and the knowledge and use of alternative approaches that are less prone to error. Secure coding can benefit from the proper use of software development tools, including compilers. Compilers typically have options that allow increased or specific diagnostics to be performed on code during compilation. Resolving these warnings (by correcting the problem or determining that the warning is superfluous) can improve the security of the

4. This material is extracted and adapted from a more extensive article by Robert Seacord and Daniel Plakosh of Carnegie Mellon University [BSI 20]. That article should be consulted for additional details and examples.

deployed software system. In addition, compilers may provide options that influence runtime settings. Understanding available compiler options and making informed decisions about which options to use and which to omit can help eliminate vulnerabilities and mitigate against runtime exploitation of undiscovered or unresolved vulnerabilities.

As one example, CERT has observed through an analysis of thousands of vulnerability reports that most vulnerabilities stem from a relatively small and recurring number of common programming errors that could be easily avoided if developers learned to recognize them and understand their potential harm. In particular, the C and C++ programming languages have proved highly susceptible to these classes of errors. Easily avoided software defects are a primary cause of commonly exploited software vulnerabilities. By identifying insecure coding practices and developing secure alternatives, software project managers and developers can take practical steps to reduce or eliminate vulnerabilities before they are deployed in the field.

5.3.1 Sources of Additional Information on Secure Coding

We encourage readers to review the Coding Practices area of the BSI Web site for additional coding practices that can be used to mitigate common problems in C and C++ [BSI 20]. An example of the use of compiler checks to minimize integer vulnerabilities is described in the "Compiler Checks" section of the Web site. Examples of using other static and dynamic analysis tools to discover and mitigate vulnerabilities are described in "Runtime Analysis Tools" and "Heap Integrity Detection."

The CERT Secure Coding Initiative (http://www.cert.org/secure-coding) works with software developers and software development organizations to reduce vulnerabilities resulting from coding errors before they are deployed in products. The initiative's work includes identifying common programming errors that lead to software vulnerabilities, establishing standard secure coding standards, educating software developers, and advancing the state of the practice in secure coding.

Table 5–1 provides a description of a number of recent and excellent books on the subject.

Table 5–1: *Books to Consult for Secure Coding Approaches and Practices*

Secure Programming with Static Analysis [Chess 2007]	Describes how static source code analysis can be used to uncover errors and the most common types of security defects that result in security vulnerabilities. The book describes how this method works, explains how to integrate it into your software development process, and explores how to conduct effective code reviews using the method.
Software Security: Building Security In [McGraw 2006]	Describes in detail how to put software security into practice. It presents the topic from the two sides of software security—attack and defense, exploiting and designing, breaking and building—including a description of seven essential "touchpoints" for software security. Excerpts and citations from *Software Security* are included throughout this chapter and on the BSI Web site.
The Secure Development Lifecycle [Howard 2006]	Describes Microsoft's Security Development Lifecycle (SDL) as one proven way to help reduce the number of software security defects during each phase of the development process. This process has been used effectively in many Microsoft products.
Secure Coding in C and C++ [Seacord 2005]	Provides a comprehensive description of common programming errors (for example, in string manipulation, integer operations, and dynamic memory management), the vulnerabilities that result from them, and mitigation strategies for minimizing their impact.
Exploiting Software: How to Break Code [Hoglund 2004]	Describes how to design software so that it is as resistant as possible to attack. This book describes how malicious hackers go about writing exploit scripts that can be used to cause software to fail; in this way, it provides software designers with an understanding of the types of attacks their software may be forced to deal with.

Table 5–1: *Books to Consult for Secure Coding Approaches and Practices (Continued)*

Secure Coding: Principles and Practices [Graff 2003]	Describes good and bad practices to consider during architecture, design, code, test, and operations, along with supporting case studies. Good practices for secure coding identified in this book include handling data with caution (perform bounds checking, set initial values for data), reusing good code whenever practicable, insisting on a sound review process (peer reviews, independent verification and validation), using checklists and standards, and removing obsolete code.
Writing Secure Code, second edition [Howard 2002]	Provides developers with detailed practices for designing secure applications, writing robust code that can withstand repeated attacks, and testing applications for security flaws. The book provides proven principles, strategies, and coding techniques.
Building Secure Software: How to Avoid Security Problems the Right Way [Viega 2001]	"Helps people involved in the software development process learn the principles necessary for building secure software. It is intended for anyone involved in software development, from managers to coders, although it contains the low-level detail that is most applicable to developers. Specific code examples and technical details are presented in the second part of the book. The first part is more general and is intended to set an appropriate context for building secure software by introducing security goals, security technologies, and the concept of software risk management" [Viega 2001, p. xxiii].

5.4 Software Security Testing[5]

M L L4

Security test activities are primarily performed to demonstrate that a system meets its security requirements and to identify and minimize

5. This material is extracted and adapted from a more extensive article by C. C. Michael and Will Radosevich of Cigital, Inc. [BSI 21]. That article should be consulted for additional details.

the number of security vulnerabilities in the software before the system goes into production. Additionally, security test activities can aid in reducing overall project costs, protecting an organization's reputation or brand once a product is deployed, reducing litigation expenses, and complying with regulatory requirements.

The goal of security testing is to ensure that the software being tested is robust and continues to function in an acceptable manner even in the presence of a malicious attack. Security testing is motivated by probing undocumented assumptions and areas of particular complexity to determine how a software program can be broken. The designers and the specification might outline a secure design, and the developers might be diligent and write secure code, but ultimately the testing process determines whether the software will be adequately secure once it is fielded.

Testing is laborious, time-consuming, and expensive, so the choice of testing approaches should be based on the risks to the software and the system. Risk analysis provides the right context and information to make tradeoffs between time and effort to achieve test effectiveness (see Section 7.4.2). An effective testing approach balances efficiency and effectiveness to identify the greatest number of critical defects for the least cost.

This section is not intended to serve as a primer on software testing. Anyone responsible for security testing should be familiar with standard approaches to software testing such as those described in these books:

- *Testing Object-Oriented Systems: Models, Patterns, and Tools* [Binder 1999]
- *Automated Software Testing* [Dustin 1999]
- *Software Test Automation* [Fewster 1999]
- *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing* [Marick 1994]
- *Black-Box Testing: Techniques for Functional Testing of Software and Systems* [Beizer 1995]
- *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Second Edition* [Black 2002]
- *Testing Computer Software, Second Edition* [Kaner 1999]

5.4.1 Contrasting Software Testing and Software Security Testing

At one time, it was widely believed that security bugs in a software system were just like traditional programming bugs and that traditional quality assurance and testing techniques could be applied equally well to secure software development. Over time, however, developers have learned that security-related bugs can differ from traditional software bugs in a number of ways. These characteristics, in turn, influence the practices that you should use for software security testing [Hoglund 2004].

- Users do not normally try to search out software bugs. An enterprising user may occasionally derive satisfaction from making software break, but if the user succeeds, it affects only that user. Conversely, malicious attackers *do* search for security-related vulnerabilities in an intelligent and deliberate manner. One important difference between security testing and other testing activities is that the security test engineer needs to emulate an intelligent attacker. An adversary might do things that no ordinary user would do, such as entering a 1000-character surname or repeatedly trying to corrupt a temporary file. Test engineers must consider actions that are far outside the range of normal activity and might not even be regarded as legitimate tests under other circumstances. A security test engineer must think like the attacker and find the weak spots first.
- Malicious attackers are known to script successful attacks and distribute exploit scripts throughout their communities. In other words, a single, hard-to-find vulnerability can be exploited by a large number of malicious attackers using publicly available exploit scripts. This proliferation of attacker knowledge can cause problems for a large number of users, whereas a hard-to-find software bug typically causes problems for only a few users.
- Although most developers are not currently trained in secure programming practices, developers can (and do) learn from experience to avoid poor programming practices that can lead to software bugs in their code. However, the list of insecure programming practices is long and continues to grow, making it difficult for developers to keep current on the latest exploits and attack patterns (see also Section 2.3.2).

- Security testing differs from traditional software testing in that it emphasizes what an application should *not* do rather than what it *should* do. While it sometimes tests conformance to positive requirements such as "User accounts are disabled after three unsuccessful login attempts" and "Network traffic must be encrypted," more often it tests negative requirements [Fink 1997] such as "Outside attackers should not be able to modify the contents of the Web page" and "Unauthorized users should not be able to access data." This shift in emphasis from positive to negative requirements affects the way testing is performed (see Section 5.4.3). The standard way to test a positive requirement is to create the conditions in which the requirement is intended to hold true and verify that the requirement is satisfied by the software. By contrast, a negative requirement may state that something should never occur. To apply a standard testing approach to negative requirements, one would need to create every possible set of conditions, which is not feasible.
- Many security requirements, such as "An attacker should never be able to take control of the application," would be regarded as untestable in a traditional software development setting. It is considered a legitimate practice for testers to ask that such requirements be refined or perhaps dropped altogether. Many security requirements, however, can be neither refined nor dropped even if they are untestable. For example, one cannot reliably enumerate all of the ways in which an attacker might gain control of an application (which would be one way to make it more testable), and obviously one cannot drop the requirement either. Thus the challenge is to find both a way to specify these types of requirements and a way to adequately test them.

Project managers and security test engineers must ask which kinds of vulnerabilities can exist for the software being tested and which kinds of problems are likely to have been overlooked by the developers. Often the most important types of vulnerabilities to consider are the most common ones (described in Section 5.2.1), which are targeted by security scanners and reported in public forums.

Many traditional software bugs can have security implications. Buggy behavior is almost by definition unforeseen behavior, and as such it presents an attacker with the opportunity for a potential exploit. Indeed, many well-known vulnerabilities could cause software to crash if they were triggered. Crashing software can expose confidential information

in the form of diagnostics or data dumps. Even if the software does not crash as the result of a bug, its internal state can become corrupted and lead to unexpected behavior at a later time. For this reason, error-handling software is a frequent target of malicious attacks. Attackers probing a new application often start by trying to crash it.

> ***Ninety Percent Right***
>
> Finding 90 percent of a software program's vulnerabilities does not necessarily make the software less vulnerable; it merely reduces the cost of future fixes and increases the odds of finding the remaining problems before attackers do. The result is that secure software development is intrinsically more challenging than traditional software development. Given this fact, security testing needs to address these unique considerations and perspectives to the extent possible and practical.

Security Testing Methods

Two common methods for testing whether software has met its security requirements are functional security testing and risk-based security testing [McGraw 2006]. Functional testing is meant to ensure that software behaves as specified and so is largely based on demonstrating that requirements defined in advance during requirements engineering (see Chapter 3) are satisfied at an acceptable level. Risk-based testing probes specific risks that have been identified through risk analysis. The next two sections discuss how functional and risk-based testing can be used to enhance confidence in the software's security.

5.4.2 Functional Testing

Functional testing usually means testing the system's adherence to its functional requirements. A functional requirement usually has the following form: "When a specific thing happens, then the software should respond in a certain way." This way of specifying a requirement is convenient for the tester, who can exercise the "if" part of the requirement and then confirm that the software behaves as it should.

Examples of functional security requirements are that a user's account is disabled after three unsuccessful login attempts and that only certain characters are permitted in a URL. These positive functional requirements can be tested in traditional ways, such as attempting three unsuccessful login attempts and verifying that the account is disabled, or by supplying a URL with illegal characters and making sure that those characters are stripped out before the URL is processed.

When risks are identified early in the SDLC, developers have adequate time to include *mitigations* for those risks (also known as countermeasures). Mitigations are meant to reduce the severity of the identified risks, and they lead to positive requirements. For example, the risk of password-cracking attacks can be mitigated by disabling an account after three unsuccessful login attempts or by enforcing long passphrases. Passphrases are largely immune to cracking and have the added benefit of often being easier to remember than complex passwords. The risk of SQL injection attacks from a Web interface can be mitigated by using an input validation whitelist (a list of all known good inputs that a system is permitted to accept) that excludes all other characters. These mitigations have to be tested not only to confirm that they are implemented correctly, but also to determine how well they actually safeguard the system against the risks they were designed to address.

A common software development practice is to ensure that every requirement can be mapped to a specific software artifact meant to implement that requirement. As a consequence, the tester who is probing a specific requirement knows exactly which code artifact to test. Generally, there is a clear mapping between functional requirements, code artifacts, and functional tests.

Some Caveats

Software engineers may not understand how to implement some security requirements. In one example, a Web application was found to be vulnerable to a directory traversal attack, where a URL containing the string ".." was used to access directories that were supposedly forbidden to remote clients. To counter this possibility, developers used a blacklist technique, in which a list is created and used to exclude or filter out bad input data and bad characters. URLs that contained this string were added to the blacklist and thus disallowed. However, blacklists are not infallible:

> [Blacklists] often fail because the enumeration is incomplete, or because the removal of bad characters from the input can result in the production of another bad input which is not caught (and so on recursively). Blacklists fail also because they are based on previous experience, and only enumerate known bad input. The recommended practice is the creation of whitelists that enumerate known good input. Everything else is rejected. [Meunier 2006]

Testing cannot demonstrate the absence of software problems; it can only demonstrate (sometimes) that problems are present [Dijkstra 1970]. The problem is that testers can try out only a limited number of test cases; the software might work correctly for those cases and fail for other cases. Therefore, testing a mitigation measure is not enough to guarantee that the corresponding risk has truly been eliminated, and this caveat is especially important to keep in mind when the risk in question is a severe one.

Also, when bugs are fixed, the fix is sometimes not subjected to the same scrutiny as those features that were part of the original software design. For example, a problem that should normally be detected in design reviews might slip through the cracks if it shows up as part of a bug fix. Sometimes software that has been repaired is retested simply by running the original test suite again—but that approach works poorly for the caveats described here.

Testing Beyond Requirements

Functional testing is meant to probe whether software behaves as it should, but so far we have focused only on requirements-based testing. A number of other functional testing techniques (as described in 170 on pages 170–171) do not rely on defined requirements. These techniques are described in more detail in [BSI 21].

5.4.3 Risk-Based Testing

Risk-based testing addresses negative requirements, which state what a software system should not do. Tests for negative requirements can be developed in a number of ways. They should be derived from a risk analysis, which should encompass not only the high-level risks identified during the design process but also low-level risks derived from the software itself.

Table 5–2: *Functional Testing Techniques*

Ad hoc testing (experience-based testing) and exploratory testing	Tests are based on the tester's skill, intuition, and experience with similar programs to identify tests not captured in more formal techniques.
Specification-based and model-based testing	Tests are derived automatically using a specification created in a formal language (rare) or through the use of a model of program interfaces.
Equivalence partitioning	Tests are derived by dividing the input domain into a collection of subsets or equivalence classes (such as output path or program structure) and then selecting representative tests for each class.
Boundary values analysis	Tests are selected on or near the boundaries of the input domain of variables, given that many defects tend to concentrate near the extreme values of inputs.
Robustness and fault-tolerance testing	Test cases are chosen outside the domain to test program robustness in the face of unexpected and erroneous inputs.
Decision table (also called logic-based) testing	Tests are derived by systematically considering every possible combination of conditions (such as inputs) and actions (such as outputs).
State-based testing	Tests are selected that cover states and transitions from a finite state machine model of the software.
Control-flow testing	Tests are selected to detect poor and incorrect program structures. Test criteria aim at covering all statements, classes, or blocks in a program (or some specified combinations).
Data-flow testing	This form of testing is often used to test interfaces between subsystems. It is accomplished by annotating a program-control flow graph with information about how variables are defined and used and then tracing paths from where the variable is defined to where it is used.

Table 5–2: *Functional Testing Techniques (Continued)*

Usage-based and use-case-based testing	Tests are derived by developing an operational scenario or set of use cases that describe how the software will be used in its operational environment. (See also Section 3.2.)
Code-based testing (also called white-box testing; see Section 5.5.1)	This approach is a superset of control-flow and data-flow testing. Tests are designed to cover the code by using the control structure, data-flow structure, decision control, and modularity.
Fault-based testing	Tests are designed to intentionally introduce faults to probe program robustness and reliability [Whittaker 2003].
Protocol conformance testing	Tests are designed to use a program's communication protocol as the test basis. In combination with boundary values testing and equivalence-based testing, this method is useful for Web-based programs and other Internet-based code.
Load and performance testing	Tests are designed to verify that the system meets its specified performance requirements (capacity and response time) by exercising the system to the maximum design load and beyond it.

When testing negative requirements, security test engineers typically look for common mistakes and test suspected weaknesses in the software. The emphasis is on finding vulnerabilities, often by executing abuse and misuse tests that attempt to exploit software weaknesses (see Section 3.2). In addition to demonstrating the actual presence of vulnerabilities, security tests can assist in uncovering symptoms that suggest potential vulnerabilities.

Requirements can be expected to contain mitigations for many risks. Mitigations generally result in positive requirements, but the fact that a risk has a mitigation does not imply that it should be ignored during risk-based testing. Even if a mitigation measure is correctly implemented, there is still a need to ask whether it really does safeguard against the risk it serves as a countermeasure for and to what extent. Each mitigation generates a positive requirement—the correct

implementation of the mitigation strategy—but it also generates a negative requirement stating that the mitigation must not be circumventable. To put it another way, the mitigation might not be sufficient for avoiding the underlying risk, and this possibility constitutes a risk in and of itself.

Unfortunately, the process of deriving tests from risks is as much an art as a science, such that it depends a great deal on the skills and security knowledge of the test engineer. Many automated tools can be helpful during risk-based testing (for example, see the description of black-box testing in Section 5.5.4), but these tools can perform only simple tasks; the difficult tasks remain the responsibility of the test engineer. You might also consider the use of commercial tools for identifying vulnerabilities in Web applications such as those from SPI Dynamics and Watchfire.

Defining Tests for Negative Requirements

As a basis for defining test conditions, past experience comes into play in two ways. First, a mature test organization typically has a set of test templates that outline the test techniques to be used for testing against specific risks and requirements in specific types of software modules. Test templates are usually created during testing projects, and they accumulate over time to capture the organization's past experience. This book does not provide test templates, but attack patterns appropriate for this purpose are described in several other sources [Hoglund 2004; Whittaker 2002, 2003].

Another way to derive test scenarios from past experience is to use incident reports. Incident reports can simply be bug reports, but in the context of security testing they can also be forensic descriptions of successful intruder activity. Furthermore, vulnerability reports are often followed by proofs of concept to demonstrate how the reported vulnerability can be exploited. Sometimes these proofs of concept are actual exploits; at other times they simply show that a vulnerability is likely to be exploitable. For example, if a buffer overflow can be made to cause a crash, then it can usually be exploited by an attacker as well. Sometimes it is sufficient to find evidence of vulnerabilities as opposed to actual exploits, so that the resulting proofs of concept can be used as the basis for test scenarios. When devising risk-based tests, it can be useful to consult IT security personnel, as their jobs involve keeping up-to-date on vulnerabilities, incident reports, and security threats.

Attack patterns (as discussed in Chapter 2) can be used effectively during software security testing to craft test cases that reflect attacker behavior and to help identify test cases that validate secure behavior.

Finally, threat modeling can be leveraged to help create risk-based tests. For example, if inexperienced intruders (e.g., script kiddies) are expected to pose a major threat, then it might be appropriate to probe the software under test with automated tools; intruders often use the same tools (see the description of black-box testing in Section 5.5.4).

Additional thought processes that might be helpful in creating new tests for negative requirements include (1) understanding a software component and its environment, (2) understanding the assumptions of the developers, and (3) building a fault model (hypotheses about what might go wrong). Consult [BSI 21] for further details.

5.5 Security Testing Considerations Throughout the SDLC[6]

Activities related to testing take place throughout the software life cycle, not just after coding is complete. Preparations for security testing can begin even before the planned software system has definite requirements and before a risk analysis has been conducted. For example, past experience with similar systems can provide a wealth of information about relevant attacker activity.

As part of a preliminary risk analysis, you might consider which environment factors the software will be subjected to, what its security needs are, and what kinds of effects a breach of security might have. This information provides useful and early inputs for test planning. If risk analysis starts early in the SDLC, it becomes possible to take a security-oriented approach when defining requirements.

During the requirements phase, test planning focuses on outlining how each requirement can and will be tested. Some requirements may initially appear to be untestable. If test planning is already under way, then those requirements can be identified and possibly revised to make them more testable. Testing is driven by both risks and requirements, and

6. This material is extracted and adapted from a more extensive article by C. C. Michael and Will Radosevich of Cigital, Inc. [BSI 21]. That article should be consulted for additional details.

risks are especially important to consider in security testing. While traditional non-security-related risks are linked to what can go wrong if a requirement is not satisfied, security analysis often uncovers severe security risks that were not anticipated in the requirements phase. In fact, a security risk analysis (as discussed in Chapters 4 and 7) is an integral part of secure software development, and it should drive requirements derivation and system design as well as security testing.

Risks identified during this phase may inspire additional requirements that call for features to mitigate those risks. The software development process can be expected to go more smoothly if these security measures are defined early in the SDLC, when they can be more easily implemented. If the development team faces intense time pressure, it is often a legitimate strategy to spend less time testing against a risk that has a known countermeasure, on the assumption that a mitigated risk is less severe.

Functional security testing generally begins as soon as software is available to test. Given this timeline, a test plan should be established at the beginning of the coding phase and the necessary infrastructure and personnel should be determined before testing starts.

Software is tested at many levels in a typical development process. This section cannot hope to catalog every possible software test activity. Instead, it describes several broader activities that are common to most test processes, some of which are repeated at different times for software artifacts at different levels of complexity. We discuss the role of security testing in each of these activities:

- Unit testing, where individual classes, methods, functions, or other relatively small components are tested
- Testing libraries and executable files
- Functional testing, where software is tested for adherence to requirements (as described in Section 5.4.2)
- Integration testing, where the goal is to test whether software components work together as they should
- System testing, where the entire system is under test

5.5.1 Unit Testing

Unit testing is usually the first stage of testing that a software artifact goes through. This type of testing involves exercising individual

functions, methods, classes, or stubs. As a functional-based approach to unit testing, white-box testing is typically very effective in validating design decisions and assumptions and in finding programming errors and implementation errors. It focuses on analyzing data flows, control flows, information flows, coding practices, and exception and error handling within the system, with the goal of testing both intended and unintended software behavior. White-box testing can be performed to validate whether code implementation follows the intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities.

White-box testing requires knowing what makes software secure or insecure, how to think like an attacker, and how to use different testing tools and techniques. The first step in such testing is to comprehend and analyze the source code (see Section 5.2.2), so knowing what makes software secure is a fundamental requirement. In addition, to create tests that exploit software, a tester must think like an attacker. Finally, to perform testing effectively, testers need to know what kinds of tools and techniques are available for white-box testing. The three requirements do not work in isolation, but together.

Further details on how to conduct white-box testing and what sorts of benefits it confers are available at [BSI 22].

5.5.2 Testing Libraries and Executable Files

In many development projects, unit testing is closely followed by a test effort that focuses on libraries and executable files. Usually test engineers who are experienced in testing security—rather than software developers—perform testing at this level. As part of this testing, there may be a need for specialized technology that crafts customized network traffic, simulates fault and stress conditions, allows observation of anomalous program behavior, and so on.

Coverage analysis (which measures the degree to which the source code has been fully tested, including all statements, conditions, paths, and entry/exit conditions) can be especially important in security testing [Hoglund 2004]. Because a determined attacker will probe the software system thoroughly, security testers must do so as well. 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 test tools that simulate error conditions during testing so as to exercise the error handlers in a dynamic environment.

Libraries need special attention in security testing, because components found in a library might eventually be reused in ways that are not anticipated in the current system design. For example, a buffer overflow in a particular library function might seem to pose little risk because attackers cannot control any of the data processed by that function; in the future, however, this 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 such reuse was not planned during the design of the current system.

5.5.3 Integration Testing

Integration testing focuses on a collection of subsystems, which may contain many executable components. Numerous software bugs are known to appear only because of the way components interact, and the same is true for security bugs as well.

Integration errors often arise when one subsystem makes unjustified assumptions about other subsystems. For example, an integration error can occur if the calling function and the called function each assume that the other is responsible for bounds checking and neither one actually does the check. The failure to properly check input values is one of the most common sources of software vulnerabilities. In turn, integration errors are one of the most common sources of unchecked input values, because each component might assume that the inputs are being checked elsewhere. (Components should validate their own data, but in many systems this ideal is sacrificed for reasons of efficiency.) During security testing, it is especially important to determine which data flows and controls flows can and cannot be influenced by a potential attacker.

5.5.4 System Testing

Certain activities relevant to software security, such as stress testing, are often carried out at the system level.[7] Penetration testing is also carried out at the system level, and when a vulnerability is found in this way, it provides tangible proof that the vulnerability is real: A vulnerability that

7. See also Chapter 6, Security and Complexity: System Assembly Challenges.

can be exploited during system testing will be exploitable by attackers. In the face of schedule, budget, and staff constraints, these problems are the most important vulnerabilities to fix.

Stress Testing for Security

Stress testing is relevant to security because software performs differently when it is under stress. For example, when one component is disabled because of insufficient resources, other components may compensate in insecure ways. An executable that crashes 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 (see Section 5.2.1) might become easier to exploit. Stress testing may also exercise error handlers, which are often fraught with vulnerabilities. Security testers should look for unusual behavior during stress testing that might signal the presence of unsuspected vulnerabilities.

Black-Box Testing

One popular approach to system testing is black-box testing. Black-box testing uses methods that do not require access to source code. Either the test engineer does not have access or the details of the source code are irrelevant to the properties being tested. As a consequence, black-box testing focuses on the externally visible behavior of the software, such as requirements, protocol specifications, APIs, or even attempted attacks. Within the security test arena, black-box testing is normally associated with activities that occur during the predeployment test phase (system test) or on a periodic basis after the system has been deployed.

Black-box test activities almost universally involve the use of tools, which typically focus on specific areas such as network security, database security, security subsystems, and Web application security. For example, network security tools include port scanners to identify all active devices connected to the network, services operating on systems connected to the network, and applications running for each identified

service. Vulnerability scanning tools identify specific security vulnerabilities associated with the scanned system based on information contained within a vulnerability database. Potential vulnerabilities include those related to open ports that allow access to insecure services, protocol-based vulnerabilities, and vulnerabilities resulting from poor implementation or configuration of an operating system or application.

For more information on black-box testing and test tools, refer to [BSI 23].

Penetration Testing

Another common approach for conducting certain aspects of system security testing is penetration testing, which allows project managers to assess how an attacker is likely to try to subvert a system. At a basic level, the term "penetration testing" refers to testing the security of a computer system and/or software application by attempting to compromise its security—in particular, the security of the underlying operating system and network component configurations.

Conventional penetration testing tools come in a variety of forms, depending on which sort of testing they can perform. A key distinguishing factor is the perspective from which each type of tool operates—that is, whether a testing tool evaluates its target from afar or from relatively close up (i.e., at least within the same computer system). Popular classes of tools used in penetration testing today include host-based, network-based, and application scanning [Fyodor 2006].

For example, most organizations, when doing network-based penetration testing, follow a process that looks something like this (Steps 1–3 constitute the vulnerability scanning approach mentioned earlier):

1. *Target acquisition.* The test engineer identifies legitimate test targets. This step is most often performed using a combination of manual and automated approaches in which the person responsible for the system under test provides a starting list of network addresses and the test engineer uses software tools to look for additional computers in the network vicinity.
2. *Inventory.* The test engineer uses a set of tools to conduct an inventory of available network services to be tested.
3. *Probe.* The test engineer probes the available targets to determine whether they are susceptible to compromise.
4. *Penetrate.* Each identified vulnerability (or potential vulnerability) is exploited in an attempt to penetrate the target system. The level

of invasiveness involved in exploiting a vulnerability can influence this step dramatically. For example, if a vulnerability can result in the attacker (in this case, the test engineer) having the ability to overwrite an arbitrary file on the target system, great care should be taken in how the vulnerability is exploited.

5. *Host-based assessment.* This step is typically carried out for any system that is successfully penetrated. It enables the test engineer to identify vulnerabilities that provide additional vectors of attack, including those that provide the ability to escalate privileges once the system is compromised.
6. *Continue.* The test engineer obtains access on any of the systems where identified vulnerabilities were exploited and continues the testing process from the network location(s) of each compromised system.

For more information on penetration testing and pitfalls to avoid, refer to [BSI 24]. For more information on penetration testing tools, refer to [BSI 25].

5.5.5 Sources of Additional Information on Software Security Testing

Articles in the *IEEE Security & Privacy* "Building Security In" series provide excellent guidance on software security testing. Articles titled "Software Penetration Testing," "Static Analysis for Security," and "Software Security Testing" are available on the BSI Web site under Additional Resources [BSI 26].

The Art of Software Security Testing [Wysopal 2006] reviews software design and code vulnerabilities and provides guidelines for how to avoid them. This book describes ways to customize software debugging tools to test the unique aspects of any software program and then analyze the results to identify exploitable vulnerabilities. Coverage includes the following topics:

- Thinking the way attackers think
- Integrating security testing into the SDLC
- Using threat modeling to prioritize testing based on risk
- Building test labs for conducting white-, gray-, and black-box testing
- Choosing and using the right tools

- Executing today's leading attacks, from fault injection to buffer overflows
- Determining which flaws are most likely to be exploited

Exploiting Software: How to Break Code [Hoglund 2004] provides examples of real attacks, attack patterns, tools, and techniques used by attackers to break software. It discusses reverse engineering, classic attacks against server software, surprising attacks against client software, techniques for crafting malicious input, buffer overflows, and rootkits.

How to Break Software Security: A Practical Guide to Testing [Whittaker 2003] defines prescriptive techniques (attacks that software test engineers can use on their own software) that are designed to reveal security vulnerabilities in software programs. The book's chapters discuss fault models for software security testing, the creation of unanticipated user input scenarios, and ways to attack software designs and code that focus on the most common places where software vulnerabilities occur (e.g., user interfaces, software dependencies, software design, and process and memory).

5.6 Summary

It is no secret that common, everyday software defects cause the majority of software vulnerabilities. The most widely used operating systems and most application software contain at least one or two defects per thousand lines of code and, therefore, may include hundreds to thousands of defects. While not every software defect is a security defect, if only 1 or 2 percent lead to security vulnerabilities, the risk is still substantial. Understanding the sources of vulnerabilities and learning to program securely are essential for protecting the Internet, your software, and your systems from attack. Reducing security defects, and thereby security vulnerabilities, requires a disciplined engineering approach based on sound coding practices [Howard 2006; McGraw 2006; Seacord 2005].

The key secure coding practices highlighted in this chapter include these approaches:

- Using sound and proven secure coding practices to aid in reducing software defects introduced during implementation

- Performing source code review using static code analysis tools, metric analysis, and manual review to minimize implementation-level security bugs

Security testing relies on human expertise to an even greater extent than does ordinary testing, so full automation of the test process is even less feasible when focusing on security issues than in a traditional testing environment. Although tools are available that automate certain types of tests, organizations using these tools should not be lulled into a false sense of security, because they cover only a small part of the spectrum of potential vulnerabilities. Instead, test tools should be viewed as aides for human testers, automating many tasks that are time-consuming or repetitive.

Creating security tests other than ones that directly map to security requirements is challenging, especially tests that intend to exercise the non-normative behavior of the system. When creating such tests, it is helpful to view the software under test from multiple angles, including the data the system will handle, the environment in which the system will operate, the users of the software (including software components), the options available to configure the system, and the error-handling behavior of the system. There is an obvious interaction and overlap between the different views; however, treating each one individually and specifically provides unique perspectives that are very helpful in developing effective tests.

This chapter has highlighted the following key software security testing practices:

- Understanding the differences between software security testing and traditional software testing, and planning how best to address these (including thinking like an attacker and emphasizing how to exercise what the software should not do)
- Constructing meaningful functional test cases (using a range of techniques) that demonstrate the software's adherence to its functional requirements, including its security requirements (positive requirements)
- Developing risk-based test cases (using, for example, misuse/abuse cases, attack patterns, and threat modeling) that exercise common mistakes, suspected software weaknesses, and mitigations intended to reduce or eliminate risks to ensure they cannot be circumvented (negative requirements)

- Using a complement of testing strategies, including white-box testing (based on deep knowledge of the source code), black-box testing (focusing on the software's externally visible behavior), and penetration testing (identifying and targeting specific vulnerabilities at the system level)

An organization should not rely exclusively on security test activities to build security into a system. This said, security testing—when it is coupled with other security activities performed throughout the SDLC—can be very effective in validating design assumptions, discovering vulnerabilities associated with the software environment, and identifying implementation issues that may lead to security vulnerabilities.

Chapter 6

Security and Complexity: System Assembly Challenges

6.1 Introduction

The primary theme of this chapter is how aspects of complexity due to technical difficulty, size, and conflicting objectives affect security as systems expand to support multiple processes within and across organizations.[1] Mitigation strategies and project management approaches are suggested for each area, including examples of "planning for failure" in the context of Web services and identity management.

System development has always encountered new and often complex problems that were not represented in project plans. Often, the hard-to-solve problems are not new. Not many years ago, for example, the Common Object Request Broker Architecture (CORBA) received considerable

1. Robert Ferguson at the SEI has been studying the effects of systems engineering complexity on project management. The discussion of complexity factors in this chapter reflects discussions with him and was also influenced by the Incremental Commitment Model (ICM) [Boehm 2007].