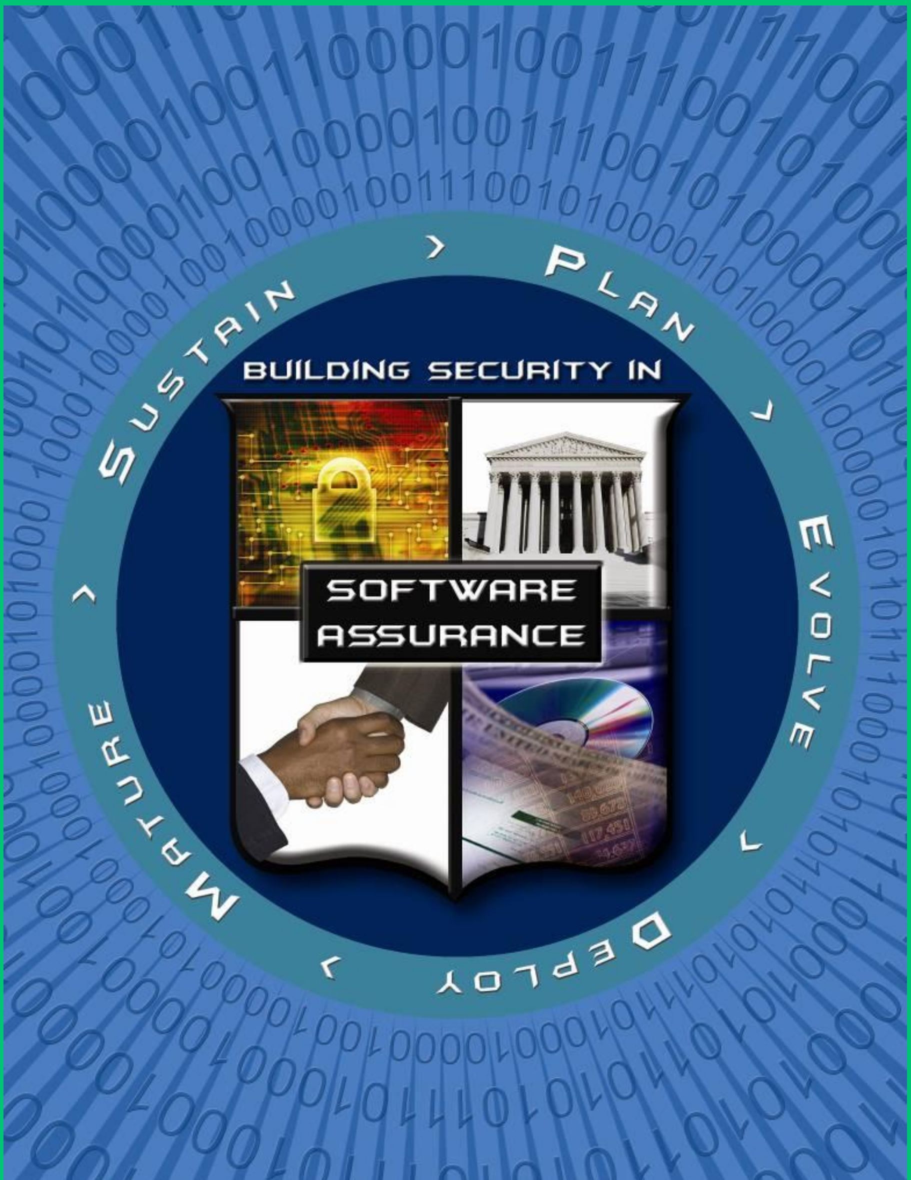
Software Security Testing

Software Assurance Pocket Guide Series:
Development, Volume III
Version 1.0, May 21, 2012

3



Software Assurance (SwA) Pocket Guide Resources

This is a resource for 'getting started' in selecting and adopting relevant practices for delivering secure software. As part of the Software Assurance (SwA) Pocket Guide series, this resource is offered for informative use only; it is not intended as directive or presented as being comprehensive since it references and summarizes material in the source documents that provide detailed information. When referencing any part of this document, please provide proper attribution and reference the source documents, when applicable.

At the back of this pocket guide are references, limitation statements, and a listing of topics addressed in the SwA Pocket Guide series. All SwA Pocket Guides and SwA-related documents are freely available for download via the SwA Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa.

Executive Summary

Software security testing validates the secure implementation of a product thus reducing the likelihood of security flaws being released and discovered by customers or malicious users. The goal of software security testing is to find vulnerabilities, keep them from the final product, and confirm that the security of the software product is at an acceptable level. This pocket guide describes the most effective security testing techniques, their strengths and weaknesses, and when to apply them during the Software Development Life Cycle (SDLC).

Target Audience

This guide has been written with software developers in mind. It is meant as an introduction to the many techniques of software security testing and how they can be applied throughout the SDLC. The goal is that the reader will use the information from this guide, and the references it points to, to become more involved in the software security testing process.

Acknowledgements

SwA Pocket Guides are developed and socialized by the SwA community as a collaborative effort to obtain a common look and feel and are not produced by individual entities. SwA Forum and Working Groups function as a stakeholder meta-community that welcomes additional participation in advancing software security and refining. All SwA-related information resources that are produced are offered free for public use. Inputs to documents for the online resources are invited. Please contact Software.Assurance@dhs.gov for comments and inquiries. For the most up to date pocket guides, check the website at https://buildsecurityin.us-cert.gov/swa/.

The SwA Forum and Working Groups are composed of government, industry, and academic members and focuses on incorporating SwA considerations in the acquisition and development processes relative to potential risk exposures that could be introduced by software and the supply chain.

Participants in the SwA Forum's Processes & Practices Working Group collaborated with the Technology, Tools and Product Evaluation Working Group in developing the material used in this pocket guide as a step in raising awareness on how to incorporate SwA throughout the SDLC.

Information contained in this pocket guide is primarily derived from **"*Enhancing the Development Life Cycle to Produce Secure Software*"** and other documents listed in the *Resources* box that follows.

Special thanks to the Department of Homeland Security (DHS) National Cyber Security Division's Software Assurance team who provided much of the support to enable the successful completion of this guide and related SwA documents. Additional thanks to the EC-Council for their input and review of the material.

Resources

» Enhancing the Development Life Cycle to Produce Secure Software***",*** DHS SwA Forum Process and Practices Working Group, 2008; at https://buildsecurityin.us-cert.gov/swa/procwg.html**.**

» "Software Security Assurance: A State-of-the-Art Report**"***,* Goertzel, Karen Mercedes, *et al*, Information Assurance Technology Analysis Center (IATAC) of the DTIC at http://iac.dtic.mil/iatac/download/security.pdf**.**

» "Multi-perspective Application Security Risk Analysis: A Toolbox Approach", Sean Barnum, et al, March 2009.

» **"**OWASP Testing Guide v3", The Open Web Application Security Project at http://www.owasp.org/index.php/Category:OWASP_Testing_Project#OWASP_Testing_Guide_v3_2.

» **"**Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today**",** The Software Assurance Forum for Excellence in Code (SAFECode), 2008 at http://www.safecode.org/publications/SAFECode_Dev_Practices1008.pdf**.**

» **'**Microsoft Security Development Lifecycle (SDL) – Process Guidance" at http://msdn.microsoft.com/en-us/library/84aed186-1d75-4366-8e61-8d258746bopq.aspx**.**

» **"**Software Security: Being Secure in an Insecure World", Mano Paul, (ISC)^{2}.

Overview

Security testing of software is performed regardless of the type of functionality that software implements. Its function is to assess the security properties and behaviors of that software as it interacts with external entities (human users, its environment, and other software), and as its own components interact with each other.

> ***Security Testing Verifies that the Software:***
>
> » Behavior is predictable and secure,
> » Exposes no vulnerabilities or weaknesses,
> » Error and exception handling routines enable it to maintain a secure state,
> » Satisfies all of its specified and implicit nonfunctional security requirements, and
> » Does not violate any specified security constraints.

The objectives of software security testing are threefold:

» To verify that the software's dependable operation continues even under hostile conditions, such as receipt of attack-patterned input, and intentional (attack-induced) failures in environment components;

» To verify the software's trustworthiness, in terms of its consistently safe behavior and state changes, and its lack of exploitable flaws and weaknesses; and

» To verify the software's survivability, by verifying that its anomaly, error, and exception handling can recognize and safely handle all anticipated security-relevant exceptions and failures, errors, and anomalies; this means minimizing the extent and damage impact that may result from intentional (attack-induced) failures in the software itself, and preventing the emergence of new vulnerabilities, unsafe state changes, *etc.* Secure software should not react to anomalies or intentional faults by throwing exceptions that leave it in an unsafe (vulnerable) state.

Software security testing is predicated on the notion of "tester-as-attacker." The test scenarios themselves should be based on misuse and abuse cases, and should incorporate known attack patterns as well as anomalous interactions that seek to invalidate assumptions made by and about the software and its environment.

Software Security Testing vs. Security Requirements Testing

Software security testing is not the same as testing the correctness and adequacy of security functions implemented by software, which are most often verified through requirements-based testing. While such tests are important, they reveal only a small piece of the picture needed to verify the security of the software.

> ***Useful Security Testing Techniques:***
>
> » Risk analysis,
> » Code reviews,
> » Automated static analysis,
> » Source & binary code fault injection,
> » Binary code analysis,
> » Vulnerability scanning,
> » Fuzz testing, and
> » Penetration testing.

Unfortunately, no amount of requirements-based testing can fully demonstrate that software is free from vulnerabilities. Nor is requirements-based testing the best approach to determining how software will behave under anomalous and hostile conditions. This is because even the most robustly-specified security requirements are unlikely to address all possible conditions that the software may be forced to operate in the real world. First, at least some of the assumptions under which the requirements were originally specified are very likely to be obsolete by the time the software is ready for testing. This is due, in part, to the changing nature of the threats to the software, and the new attack strategies and assistive technologies that have emerged with the potential to target its vulnerabilities. These factors change often, and at a speed that is much quicker than any specification can match. Second, if the software contains acquired components, the versions actually included in the implemented system may be different than those imagined when the software was architected. The new versions may contain more, fewer, and/or different vulnerabilities than those that shaped the assumptions under which the software was developed. For all these reasons, requirements-based testing should always be augmented with software security testing.

The Case for Software Security Testing

Today's society is heavily dependent on software. Many business activities and critical functions such as national defense, banking, healthcare, telecommunications, aviation and control of hazardous materials depend upon the correct, predictable operation of software. This extreme reliance on software makes it a high-value target for those who wish to exploit or sabotage such activities and functions, whether for financial gain, political or military advantage, to satisfy ideological imperatives, or out of simple malice.

The presence of flaws and defects makes software a target for attackers. Software flaws and defects can cause software to behave incorrectly and unpredictably, even when it is used purely as its designers intended. Moreover, a number of software flaws and defects can be intentionally exploited by attackers to subvert the way the software operates—making it untrustworthy—or to sabotage the software's ability to operate—making it undependable.

Software threats can occur:

» **During development***:* A developer may corrupt the software intentionally or unintentionally.

» **During deployment***:* Those responsible for distributing the software may fail to tamperproof it.

» **During operation***:* Any software system that runs on a network-connected platform will have its vulnerabilities exposed during its operation.

» **During sustainment***:* Discovered vulnerabilities in released software are not issued patches or updates in a timely manner to correct those vulnerabilities.

The question is ***when***, not if, a particular software system will be attacked. The following statistics highlight the severity of this problem:

» 80% of organizations will experience an application security incident by 2010 (Gartner),

» 90% of web sites are vulnerable to attacks (Watchfire), and

» 75% of attacks are directed at the application layer (Gartner).

There are multiple costs associated with insecure software that are not easily quantifiable. A security breach can lead to loss of reputation, loss of customer trust, fines, recovery costs, and liability exposure for the victimized organization due to insecure software. But the more damaging issue will be to regain the confidence of the organization's customers. David Rice, former NSA cryptographer and author of ***Geekonomics: The Real Cost of Insecure Software***, approximates that the total economic cost of security flaws is around US $180 billion a year, as reported on Forbes.com. While the economics of software assurance can be extrapolated from the aggregate amount of fines levied on organizations that have experienced a breach due to insecure software, it still does not provide a complete view of the cost of insecure software. The real cost, not as neatly quantifiable, is the extent of reputational damage and loss of customer trust.

Software security testing, code reviews and risk analysis are some of the most effective methods for identifying software weaknesses and vulnerabilities.

What to Test

It can be helpful to think of software development as a combination of people, processes, and technology. If these are the factors that "create" software, then it is logical that these are the factors that must be tested. Today most people generally test the technology or the software functionality itself.

An effective testing program should have components that test **People** – to ensure that there are adequate education and awareness; **Processes** – to ensure that there are adequate policies and standards and that people know how to follow these policies; and **Technology** – to ensure that the processes have been effective in their implementation. Unless a holistic approach is adopted, testing just the technical implementation of software will not uncover management or operational vulnerabilities that could be present. By testing the people, processes and technology, an organization can catch issues that would later manifest themselves into defects in the technology, thus eradicating bugs early and identifying the root causes of defects. Likewise, testing only some of the technical issues that can be present in a system will result in an incomplete and inaccurate security posture assessment.

Test Planning

Understand the Subject - One of the first steps in developing a security test plan should be to require accurate documentation of the software. The architecture, data-flow diagrams, use cases, and more should be written in formal documents and made available for review. The technical specification and software documents should include information that lists not only the desired use cases, but also any specifically disallowed use cases.

If only the compiled executable and vendor-provided documentation are available, the tester will need to infer much of the information he/she needs by extrapolating it from observations of how the executing software behaves under as many conditions and inputs as can be exercised during testing.

A key goal of the test planner, then, will be to define the combination of tests, test scenarios, and test data that will reveal sufficient information about the software to enable the tester to make reliable judgments about how secure it will be once deployed.

The test plan should include:

» Security test cases and scenarios based on the misuse/abuse cases,
» Test data (both meaningful and fuzzed),
» Identification of the test tools and integrated test environment or "ecosystem,"
» Pass/fail criteria for each test, and
» Test report template. This should enable capture and analysis of test results and of actions for addressing failed tests.

The misuse and abuse cases and attack/threat models developed early in the SDLC also provides basis for developing appropriately comprehensive and revealing security test scenarios, test cases, and test oracles. As noted earlier, attack patterns can provide the basis for much of the malicious input incorporated into those cases and models.

The test environment should duplicate as closely as possible the anticipated execution environment in which the software will be deployed. The test environment should be kept entirely separate from the development environment. All test data, testing tools, integrated test environment, *etc.,* as well as the test plan itself and all test results (both raw results and test reports) should be maintained under strict configuration management control to prevent tampering or corruption.

A single test method cannot identify all the different type of software vulnerabilities so it is crucial to obtain the right tool set and test technique. A combination of different test techniques and tool sets with complementing characteristics provides the best chance to obtain the greatest depth and breadth of coverage.

On-line Resources

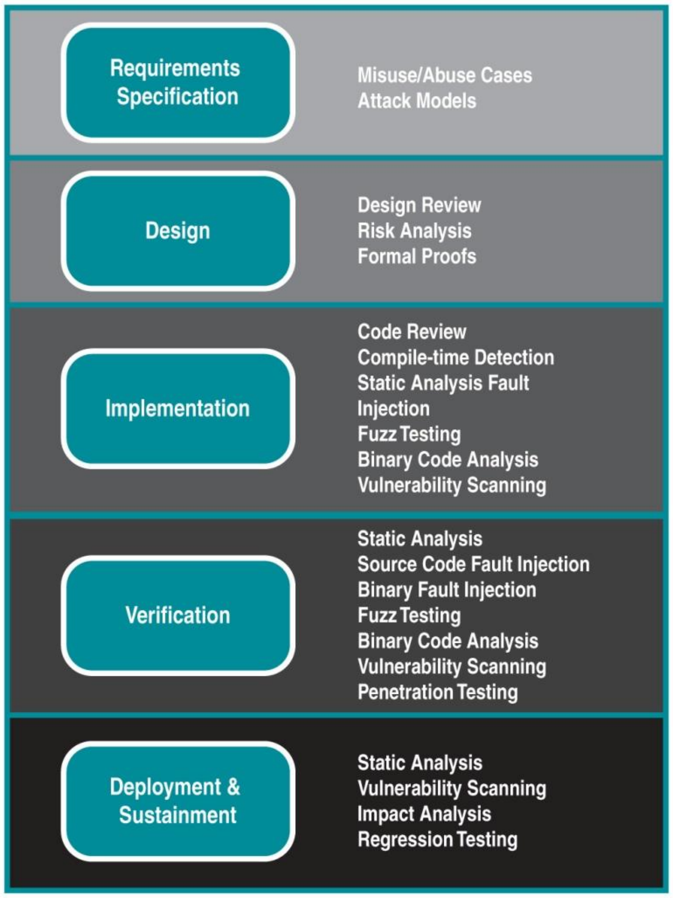
- » NIST SAMATE Tool Survey at http://samate.nist.gov/index.php/Main_Page.html.
- » Metasploit exploit generation tool and library at http://www.metasploit.com/.

Test Timing

Historically, problems found early in the SDLC are significantly easier and less costly to correct than problems discovered post-implementation or, worse, post-deployment. A thorough regiment of security reviews and tests should begin as early in the SDLC as is practicable, and should continue iteratively until the operational software is "retired."

Figure 1 shows a suggested distribution of different security test techniques throughout various life cycle phases.

Figure 1 – Test Techniques through the SDLC



Risk Analysis

Risk analysis should be conducted during the design phase of development to carefully review security requirements and to identify security risks. It is an efficient way to identify and address these risks during the design phase.

For security concerns, threat modeling is a systematic process that is used to identify threats and vulnerabilities in software. Threat modeling has become a popular technique to help system designers think about the security threats that their system might face. Therefore, threat modeling can be seen as risk assessment for software development. In fact, it enables the designer to develop mitigation strategies for potential vulnerabilities and helps them focus their limited resources and attention on the parts of the system most at risk. It is recommended that all applications have a threat model developed and documented. Threat models should be created as early as possible in the SDLC, and should be revisited as the application evolves and development progresses. To develop a threat model, implement a simple approach that follows the NIST 800-30 [11] standard for risk assessment. This approach involves:

- » **Decomposing the application** – understand, through a process of manual inspection, how the application works, its assets, functionality, and connectivity.
- » **Defining and classifying the assets** – classify the assets into tangible and intangible assets and rank them according to business importance.
- » **Exploring potential vulnerabilities** - whether technical, operational, or management.
- » **Exploring potential threats** – develop a realistic view of potential attack vectors from an attacker's perspective, by using threat scenarios or attack trees.
- » **Creating mitigation strategies** – develop mitigating controls for each of the threats deemed to be realistic. The output from a threat model itself can vary but is typically a collection of lists and diagrams. The OWASP Code Review Guide http://www.owasp.org/index.php/Application_Threat_Modeling outlines a methodology that can be used as a reference for the testing for potential security flaws.

Some people within the SwA community recommend threat modeling training before developing any models. Others such as Microsoft provide a free threat modeling tool available for downloading at http://msdn.microsoft.com/en-us/security/dd206731.aspx that is not specifically designed for security experts. It can be used by developers with limited threat modeling experience by providing guidance on creating and analyzing threat models.

On-line Resources

- » The Microsoft Security Development Lifecycle (SDL): Training and Resources at http://msdn.microsoft.com/en-us/security/cc448120.aspx.
- » "Threat Modeling: Diving into the Deep End", Jeffrey A. Ingalsbe, et al, at https://buildsecurityin.us-cert.gov/daisy/bsi/resources/articles/932-BSI.html, *IEEE Software*, January/February 2008.

Code Review

Source code review (also known as static analysis) is the process of manually checking source code for security weaknesses. Many serious security vulnerabilities cannot be detected with any other form of analysis or testing. Most security experts agree that there is no substitute for actually looking at code for detecting subtle vulnerabilities. Unlike testing third party closed software such as operating systems, when testing applications (especially if they have been developed in-house) the source code should be made available for testing purposes. Many unintentional but significant security problems are also extremely difficult to discover with other forms of analysis or testing, such as penetration testing, making source code analysis the technique of choice for technical testing.

Code Review Advantages:

» Completeness and effectiveness, and
» Accuracy.

Code Review Disadvantages:
» Not practical for large code bases,
» Requires highly skilled reviewers,
» Labor intensive, and
» Difficult to detect runtime errors.

With the source code, a tester can accurately determine what is happening (or is supposed to be happening) and remove the guess work of black box testing. Examples of issues that are particularly conducive to being found through source code reviews include concurrency problems, flawed business logic, access control problems, and cryptographic weaknesses as well as backdoors, Trojans, Easter eggs, time bombs, logic bombs, and other forms of malicious code. These issues often manifest themselves as the most harmful vulnerabilities in web sites. Source code analysis can also be extremely efficient to find implementation issues such as sections of the code where input validation was not performed or where fail open control procedures may be present. Operational procedures need to be reviewed as well, since the source code being deployed might not be the same as the one being analyzed.

Code review is highly labor intensive, but can, when reviewers with appropriate levels of experience perform the review, produce the most complete, accurate results early in the review process, before the reviewer fatigues. It is common for the reviewer to begin by very meticulously checking every line of code, then to gradually skip larger and larger portions of code, so that by the end of the review, the inconsistent and decreasing amount of "code coverage" is inadequate to determine the true nature of the software. It is important to note, that as the size of the code-base increases it becomes less feasible to perform a complete manual review. In such cases, it may be beneficial to perform a semi-automated review, with manual reviews being performed on critical subsets of the code base.

Code reviews are also useful for detecting indicators of the presence of malicious code. For example, if the code is written in C, the reviewer might seek out comments that indicate exploit features, and/or portions of code that are complex and hard-to-follow, or that contain embedded assembler code (*e.g.,* the *_asm_* feature, strings of hexadecimal or octal characters/values).

Other things to look for during code review include:

» Presence of developer backdoors and implicit and explicit debug commands. Implicit debug commands are seemingly innocuous elements added to the source code to make it easier for the developer to alter the software's state while testing; if these commands are left in the software's comment lines they may be exploitable after the software has been compiled and deployed.

» Unused calls that don't accomplish anything during system execution, such as calls invoking environment-level or middleware-level processes or library routines that are not expected to be present in the installed target environment.

Automated Static Analysis

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

Static Analysis Advantages:

» Faster than Code Review,
» Complete, consistent coverage,
» Detection of "low hanging fruit" type flaws,
» Efficient for large code bases, and
» Results easier to decipher for less skilled security reviewers.

Static Analysis Disadvantages:

» Large number of false positives,
» False negatives,
» Inability to detect subtle errors,
» Results review is labor intensive, and
» Inability to detect runtime errors.

Static analysis testing should be performed as early and as often in the life cycle as possible. The most effective tests are performed on granularly small code units—individual modules or functional-process units—which can be corrected relatively easily and quickly before they are added into the larger code base. Iteration of reviews and tests ensures that flaws within smaller units will be dealt with before the whole system code review, which can then focus on the "seams" between code units, which represent the relationships among and interfaces between components.

Static analysis tools are effective at detecting language rules violations such as buffer overflows, incorrect use of libraries, type checking and other flaws. Static analysis tools can scan very large code bases in a relative short time when compared to other techniques. The reviewer's job is limited to running the tool and interpreting its results. Static analysis tools are not sophisticated enough to detect anomalies that a human reviewer would notice. Static analysis tools can provide additional benefits, by allowing developers to run scans as they are developing—addressing potential security vulnerabilities early in the process. Similarly, the level of expertise required for an automated review is less than that required for a manual review. In many cases, the tool will provide detailed information about the vulnerability found, including suggestions for mitigation.

Static analysis tools have several limitations including:

» Limited number of path to analyze since a full exploration of all possible paths through the program could be very resource intensive.
» Third party code – If part of a source code is not available, such as library code, OS, etc., the tool has to make assumptions about how the missing code operates.
» Inability to detect unexpected flaws - Flaw categories must be predefined.
» Inability to detect architectural errors.
» Inability to detect system administration or user mistakes.
» Inability to find vulnerabilities introduced or exacerbated by the execution environment.
» Inability to support novice code reviewers, i.e., those without profound understanding of the security implications of software coding constructs.

One of the primary concerns with static analysis tools is assessing their accuracy and generation of false positives. In most situations, the accuracy of any analysis tool revolves around its false negative (*i.e.,* vulnerabilities overlooked by the tool) and false positive (*i.e.,* false alarms produced by the tool) rates. Some tools that employ aggressive algorithms generate a large amount of false positives that require manual verification by the reviewer that can be time consuming. Most of the algorithms or rules used by these tools can be tailored to reduce the false positive rate at the expense of increasing the false negative rate or vice-versa. When relying on these tools to generate evidence to support assurance case, the reviewer must provide strong evidence that the tool was used correctly and in a way that ensured that the number of false negatives and false positives were minimized, and the maximum number of *true* results was discovered.

There are several metrics to assess the performance of the tools that also helps in understanding their limitations. **Recall** is measured as the number of flaws detected by the tool divided by all the flaws present in the code. **Precision** is the ratio of true positives to all the flaws reported. A number of resources are available to aid in this regard. The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project has defined the draft NIST SP 500-268, *Source Code Security Analysis Tool Functional Specification*, which identifies a set of requirements against which source code analysis tools can be measured. Additionally, SAMATE has released the draft NIST SP 500-270, *Source Code Security Analysis Tool Test Plan*, which provides insight into using the SAMATE Reference Dataset (SRD), which is a compilation of insecure code examples, against which users can run multiple source code analysis tools and compare their results to get a better understanding of their comparative strengths and weaknesses.

On-line Resources

» "“The Use and Limitations of Static Analysis Tools to Improve Software Quality”, Paul Anderson, Grammatech, Inc.at http://www.stsc.hill.af.mil/crosstalk/2008/06/0806Anderson.html.

  “Source Code Analysis Tools – Overview”, Michael, et al, Cigital, Inc., at al, at https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/code/263-BSI.html.

» The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project at http://samate.nist.gov/index.php/Main_Page.html.

Source Code Fault Injection

Source code fault injection is a testing technique originated by the software safety community. It is used to induce stress in the software, create interoperability problems among components, simulate faults in the execution environment, and thereby reveal safety-threatening faults that are not made apparent by traditional testing techniques. Security fault injection extends standard fault injection by adding error injection, thus enabling testers to analyze the security of the behaviors and state changes that result in the software when it is exposed to various perturbations of its environment data. These data perturbations are intended to simulate the types of faults that would result during unintentional user errors as well as intentional attacks on the software *via* its environment, as well as attacks on the environment itself. A data perturbation is simply the alteration of the data the execution environment passes to the software, or that one software component passes to another. Fault injection can reveal both the effects of security faults on individual component behaviors, and the behavior of the system as a whole. Also analyzed during source code fault injection are the ways in which faults propagate through the source code.

Fault propagation analysis involves two source code fault injection techniques: *extended propagation analysis* and *interface propagation analysis*. The objective of both techniques is to trace how state changes resulting from a given fault propagate through the source code tree. To prepare for *fault propagation analysis*, the tester must generate a fault tree from the program's source code. To perform an extended propagation analysis, the tester injects faults into the fault tree, then traces how each injected fault propagates through the tree. From this, one can extrapolate outward to anticipate the overall impact a particular fault may have on the behavior of the software as a whole.

In *interface propagation analysis*, the focus is shifted from perturbing the source code of the module or component itself to perturbing the states that propagate *via* the interfaces between the module/component and other application-level and environment-level components. As with source code fault injection, in interface propagation analysis anomalies are injected into the data feeds between components, enabling the tester to view how the resulting faults propagate and to discovery whether any new anomalies result. In addition, interface propagation analysis enables the tester to determine how a failure of one component may affect neighboring components, which is a particularly important determination to make for components that either provide protections to or rely on protections from others.

Source code fault injection is particularly useful in detecting incorrect use of pointers and arrays, use of dangerous calls, and race conditions. Like all source code analyses, source code fault injection is most effective when used iteratively throughout the code implementation process. When new threats (attack types and intrusion techniques) are discovered, the source code can be re-instrumented with faults representative of those new threat types.

On-line Resources

» "Second-order Code Injection: Advanced Code Injection Techniques and Testing Procedures". Ollmann, Gunter. Undated whitepaper posted on his Weblog at http://www.technicalinfo.net/papers/SecondOrderCodeInjection.html.

Binary Fault Injection

Binary fault injection is most useful when performed as an adjunct to security penetration testing to enable the tester to obtain a more complete picture of how the software responds to attacks.

Software programs interact with their execution environment though operating system calls, remote procedure calls, application programmatic interfaces, man machine interfaces, *etc.* Binary fault injection involves monitoring the fault injected software's execution at runtime. For example, by monitoring system call traces, the tester can decipher the names of system calls (which reveal the types of resources being accessed by the calling software); the parameters to each call (which reveal the names of the resources, usually in the first parameter, and how the resources are being used); and the call's return code/value (which reveals success or failure of the access attempt).

In binary fault injection, faults are injected into the environment resources that surround the program. Environmental faults in particular are useful to simulate because they are most likely to reflect real world attack scenarios. However, injected faults should not be limited to those simulating real world attacks. As with penetration testing, the fault injection scenarios exercised should be designed to give the tester as complete as possible an understanding of the security of the behaviors, states, and security properties of the software system under all possible operating conditions.

Environment perturbation via fault injection provides the tester with several benefits:

» The ability to simulate environment anomalies without understanding how those anomalies occur in the real world. This enables fault injection by testers who do not have a deep knowledge of the environment whose faults are being simulated. Fault injection can emulate environment anomalies without needing reference to how those anomalies occur in the real world.

» The tester can decide which environment faults to emulate at which times, thus avoiding the problem that arises when doing a full environment emulation in which the environment state, when the software interacts with it, may not be what is expected, or may not have the expected effect on the software's behavior.

» Unlike penetration tests, fault injection tests can be automated with relative ease.

Binary fault injection tools include source and binary fault injectors and brute force testers. The main challenges in fault injection testing are determining the most meaningful number and combination of faults to be injected and, as with all tests, interpreting the results. Also, just because a fault injection does not cause the software to behave in a non-secure manner, or fail into a non-secure state, the tester cannot interpret this to mean that the software will be similarly "well-behaved" when exposed to the more complex inputs it typically receives during operational execution. For this reason, security tests that are performed with the software deployed in its actual target environment (*e.g.,* penetration tests and vulnerability scans) are critical for providing a complete picture of how the software will behave under such conditions.

On-line Resources

» "Testing Fault Injection in Local Applications". Wysopal, Chris, *et al.* SecurityFocus, 23 January 2007 (excerpt from *The Art of Software Security Testing) at* http://www.securityfocus.com/infocus/1886.

» "Vulnerability Testing of Software Systems Using Fault Injection". Du, Wenliang and Aditya P. Mathur. Technical Report COAST TR98-02, 6 April 1998 at https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/98-02.pdf.

» "On the Use of Fault Injection to Discover Security Vulnerabilities in Applications." Sivaramakrishnan, Hariharan. University of Maryland master of science thesis, May 2006 at https://drum.umd.edu/dspace/bitstream/1903/3566/1/umi-umd-3404.pdf.

Fuzz Testing

Fuzz testing inputs random invalid data (usually produced by modifying valid input) to the software under test *via* its environment or *via* another software component. The term *fuzzing* is derived from the fuzz utility (ftp://grilled.cs.wisc.edu/fuzz), which is a random character generator for testing applications by injecting random data at their interfaces [Miller 90]. In this narrow sense, fuzzing means injecting noise at program interfaces. Fuzz testing is implemented by a program or script that submits a combination of inputs to the software to reveal how that software responds. The idea is to look for interesting program behavior that results from noise injection and may indicate the presence of a vulnerability or other software fault.

Fuzzers are generally specific to a particular type of input, such as HTTP input, and are written to test a specific program; they are not easily reusable. Their value is their specificity, because they can often reveal security vulnerabilities that generic testing tools such as vulnerability scanners and fault injectors cannot.

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

For many interfaces, the idea of simply injecting random bits works poorly. For example, presenting a web interface with the randomly generated URL "Ax@#1ZWtB." Since this URL is invalid, it will be rejected more or less immediately, perhaps by a parsing algorithm relatively near to the interface. Fuzzing with random URLs would test that parser extensively, but since random strings are rarely valid URLs, this approach would rarely test anything else about the application. The parser acts as a sort of artificial layer of protection that prevents random strings from reaching other interesting parts of the software.

Completely random fuzzing is a comparatively ineffective way to uncover problems in an application. Fuzzing technology (along with the definition of fuzzing) has evolved to include more intelligent techniques. For example, fuzzing tools are aware of commonly used Internet protocols, so that testers can selectively choose which parts of the data will be fuzzed. These tools also generally let testers specify the format of test data, which is useful for applications that do not use one of the standard protocols. These features overcome the limitation of fuzzing. In addition, fuzzing tools often let the tester systematically explore the input space. The tester is able to specify a range of input values instead of having to rely on randomly generated noise.

Resources

- *Fuzzing: Brute Force Vulnerability Discovery.* Sutton, Michael, Adam Greene, and Pedram Amini. Indianapolis, Indiana: Addison-Wesley Professional, 2007.
- "An empirical study of the reliability of UNIX utilities." [Miller 90] Miller, Barton P.; Fredriksen, Lars; & So, Bryan. *Communications of the ACM 33*, 12 (December 1990): 32-44.

  Black Box Security Testing Tools. BuildSecurityIn. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box/261-BSI.html.

Binary Code Analysis

Binary code analysis uses tools that support reverse engineering and analysis of binary executables such as decompilers, disassemblers, and binary code scanners, reflecting the varying degrees of reverse engineering that can be performed on binaries.

The least intrusive technique is binary scanning. Binary scanners, analyze machine code to model a language-neutral representation of the program's behaviors, control and data flows, call trees, and external function calls. Such a model may then be traversed by an automated vulnerability scanner in order to locate vulnerabilities caused by common coding errors and simple back doors. A source code emitter can use the model to generate a human-readable source code representation of the program's behavior, enabling manual code review for design level security weaknesses and subtle back doors that cannot be found by automated scanners.

The next least intrusive technique is disassembly, in which binary code is reverse engineered to intermediate assembly language. The disadvantage of disassembly is that the resulting assembler code can only be meaningfully analyzed by an expert who both thoroughly understands that particular assembler language and who is skilled in detecting security-relevant constructs within assembler code.

The most intrusive reverse engineering technique is decompilation, in which the binary code is reverse engineered all the way back to source code, which can then be subjected to the same security code review techniques and other white box tests as original source code. Note, however, that decompilation is technically problematical: the quality of the source code generated through decompilation is often very poor. Such code is rarely as navigable or comprehensible as the original source code, and may not accurately reflect the original source code. This is particularly true when the binary has been obfuscated or an optimizing compiler has been used to produce the binary. Such measures, in fact, may make it impossible to generate meaningful source code. In any case, the analysis of decompiled source code will always be significantly more difficult and time consuming than review of original source code. For this reason, decompilation for security analysis only makes sense for the most critical of high consequence components.

Reverse engineering may also be legally prohibited. Not only do the vast majority of software vendors' license agreements prohibit reverse engineering to source code and assembler) form, software vendors have repeatedly cited the Digital Millennium Copyright Act (DMCA) of 1999 to reinforce such prohibitions, even though the DMCA explicitly exempts reverse engineering as well as "encryption research" (which involves intentional breaking of encryption applied to the software being reverse engineered) from its prohibitions against copy protection circumvention.

Resources

» *Reversing: Secrets of Reverse Engineering.* Eliam, Eldad. Indianapolis, Indiana: Wiley Publishing, 2005.

» "Chapter 3: Reverse Engineering and Program Understanding" McGraw, Gary and Greg Hoglund. *Exploiting Software: How to Break Code at* http://www.amazon.com/Exploiting-Software-Break-Addison-Wesley-Security/dp/0201786958/ref=pd_sim_b_title_3.

» "Putting trust in software code". Wysopal,Chris. *USENIX ;login:,* Volume 29, Number 6, December 2004 at http://www.usenix.org/publications/login/2004-12/pdfs/code.pdf.

Vulnerability Scanning

Automated vulnerability scanning is supported for application level software, as well as for Web servers, database management systems, and some operating systems. Application vulnerability scanners can be useful for software security testing. These tools scan the executing application software for input and output of known patterns that are associated with known vulnerabilities. These vulnerability patterns, or "signatures", are comparable to the signatures searched for by virus scanners, or the "dangerous coding constructs" searched for by automated source code scanner, making the vulnerability scanner, in essence, an automated pattern-matching tool.

While they can find simple patterns associated with vulnerabilities, automated vulnerability scanners are unable to pinpoint risks associated with aggregations of vulnerabilities, or to identify vulnerabilities that result from unpredictable combinations of input and output patterns.

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

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

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

Because automated vulnerability scanners are signature-based, as with virus scanners, they need to be frequently updated with new signatures from their vendor. Two important evaluation criteria for selecting a vulnerability scanner are: (1) how extensive the tool's signature database is, and (2) how often the supplier issues new signatures.

Vulnerability scanners are most effective when used:

» During the security assessment of binary components;

» Before penetration testing, in order to locate straightforward common vulnerabilities, and thereby eliminate the need to run penetration test scenarios that check for such vulnerabilities.

In the software's target environment, vulnerabilities in software are often masked by environmental protections such as network- and application-level firewalls. Moreover, environment conditions may create novel vulnerabilities that cannot be found by a signature-based tool, but must be sought using a combination of other black box tests, most especially penetration testing of the deployed software in its actual target environment.

In addition to application vulnerability scanners, there are scanners for operating systems, databases, web servers, and networks. Most of the vulnerabilities checked by these other scanners focus on configuration deficiencies and information security vulnerabilities (*e.g.,* data disclosed that should not be). However, these scanners do often look for conditions such as buffer overflows, race conditions, privilege escalations, *etc.,* at the environment level that can have an impact on the security of the software executing in that environment. Therefore, it is a good idea to run execution environment scanners with an eye towards observing problems at the environment's interfaces with the hosted application, in order to get an "inside out" view of how vulnerabilities in the environment around the application may be exploited to "get at" the application itself.

On-line Resources

» *Black Box Testing tools resources.* BuildSecurityIn at https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box.html.

» "Automating Web application security testing". Anantharaju, Srinath. Google Online Security Blog, 16 July 2007 at http://googleonlinesecurity.blogspot.com/2007/07/automating-web-application-security.html.

» "Analyzing the Effectiveness and Coverage of Web Application Security Scanners". Suto, Larry. October 2007 at http://www.stratdat.com/webscan.pdf.

» "The Best Web Application Vulnerability Scanner in the World". Grossman, Jeremiah. On his Weblog, 23 October 2007 at http://jeremiahgrossman.blogspot.com/2007/10/best-web-application-vulnerability.html.

Penetration Testing

Penetration testing, also known as ethical hacking, is a common technique for testing network security. While penetration testing has proven to be effective in network security, the technique does not naturally translate to applications. Penetration testing is, for the purposes of this guide, the "art" of testing a running application in its "live" execution environment to find security vulnerabilities. Penetration testing observes whether the system resists attacks successfully, and how it behaves when it cannot resist an attack. Penetration testers also attempt to exploit vulnerabilities that they have detected and ones that were detected in previous reviews.

Types of penetration testing include black-box, white box and grey box. In black-box penetration testing, the testers are given no knowledge of the application. White-box penetration is the opposite of black-box in that complete information about the application may be given to the testers. Grey-box penetration testing, the most commonly used, is where the tester is given the same privileges as a normal user to simulate a malicious insider.

Penetration testing should focus on those aspects of system behavior, interaction, and vulnerability that cannot be observed through other tests performed outside of the live operational environment. Penetration testers should subject the system to sophisticated multi-pattern attacks designed to trigger complex series of behaviors across system components, including non-contiguous components. These are the types of behaviors that cannot be forced and observed by any other testing technique. Penetration testing should also attempt to find security problems that are likely to originate in the software's architecture and design (versus coding flaws that manifest as vulnerabilities), as it is this type of vulnerability that tends to be overlooked by other testing techniques.

The penetration test plan should include “worst case” scenarios that reproduce threat vectors (attacks, intrusions) that are considered highly damaging, such as insider threat scenarios. The test plan should capture:

» The security policy the system is supposed to respect or enforce,
» Anticipated threats to the system, and
» The sequences of likely attacks that are expected to target the system.

The Rules of Engagement (ROE), is an essential piece of the penetration test plan. The ROE is the agreement between the testers, the application’s administrators, the system owners (management of the target application or system) and any stakeholders deemed key. The ROE describes the limitations, scope, and controls to be placed around the testing activities. Establishing an ROE ensures that the signatories are aware and agree on what will take place during the penetration test. The ROE protects the testers from liability provided they stay within the confines of the agreement. The ROE is a living document, but all signatories must approve any amendments.

Testers usually start by gathering information about the target through Internet searches, newspapers, third party sources and spidering (automated browsing and downloading website pages). After that, the tester will often try querying for known vulnerable scripts or components, and all points of user input to test for flaws like SQL injection, cross-site scripting, cross-site request forgery (CSRF), command execution, etc. The testers will usually use an application scanner or a web application assessment proxy to perform these techniques. A combination of fuzzing and injection of strings known to cause error conditions may also be used. In addition to these technical penetration testing techniques, the tester may also try to hack the application by using social engineering techniques. These include dumpster diving, and phishing.

Many developers use application penetration testing as their primary security testing technique. While it certainly has its place in a testing program, application penetration testing should not be considered the primary or only testing technique. Penetration testing can lead to a false sense of security. Just because an application passes penetration testing does not mean that it is free of vulnerabilities. Conversely, if an application fails penetration, there is a strong indication that there are serious problems that should be mitigated.

Resources

» Penetration Testing resources. BuildSecurityIn. 20 May 2009 at https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/penetration.html.
» “Professional Pen Testing for Web Applications.” Andreu, Andres. Indianapolis, Indiana: Wiley Publishing, 2006.
» “Application Penetration Testing”. Thompson, Herbert H. IEEE Security and Privacy, Volume 3 Number 1, January/February 2005, pages 66-69.
» Build Security In: Penetration Testing Tools resources. Accessed 27 January 2010 at https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/penetration.html
» *Licensed Penetration Tester Methodology*. EC-Council (http://www.eccouncil.org/).
» “Penetration Testing: Not Just a Hack Job”. Decker, Matthew J. Apr. 2004. Accessed 1 February 2010 at http://www.agilerm.net/pdfs/ISSA-Journal-Article-June2004.pdf.

Interpreting and Using Test Results

As soon as each test is completed, test results and the exact version of the software artifact tested should be checked in to the configuration management system.

The results of the requirements- and software security testing regimen should provide adequate information for the software's developers to:

» Identify missing security requirements that need to be evaluated to determine the risk posed to the software if those requirements are not added to the specification and the software is not modified to implement them. These could include requirements for constraints, countermeasures, or protections to eliminate or mitigate weaknesses, vulnerabilities, and unsafe behaviors/state changes; and

» Determine the amount and nature of reengineering (or refactoring) needed to satisfy the new requirements.

Note that just because software successfully passes all of its security tests, this does not mean that novel attack patterns and anomalies will never arise in deployment to compromise the software. For this reason, iterative testing throughout the software's life time is imperative to ensure that its security posture does not degrade over time.

In terms of providing meaningful security measures for risk management, the aggregation of test results should provide an indication of whether the software is able to resist, withstand, and/or recover from attacks to the extent that risk can be said to be mitigated to an acceptable level. This is the best that can be achieved given the current state of the art of software security test technique and tools.

Questions to Ask Software Developers

Managers in development and procurement organizations should ask questions to determine if the team responsible for testing the software uses practices that harness secure testing techniques. These questions should highlight the major security testing techniques for assuring secure applications. These questions are intended to raise awareness of the content of this Guide; they are not a complete set of questions. A more comprehensive set of questions can be found in the Pocket Guides for "Software Assurance in Acquisition and Contract Language" and "Software Supply Chain Risk Management & Due-Diligence."

» Of the software that are being delivered, were some compiler warning flags disabled? If so, who made the risk decision for disabling it?
» Does the testing program have components that test People, Processes, and Technologies? This will help uncover management or operational vulnerabilities.
» Are attack patterns being used to provide the basis for testing of malicious inputs?
» Are accurate documentation (e.g. misuse cases, threat models, attack trees, data-flow diagrams, etc.) of the software being provided to the testers?
» Are security reviews and testing techniques performed in the Software Development Life Cycle (SDLC) as early as possible and as practicable?
» Was the source code available for source code review and testing purposes?
» Is static analysis testing performed as early and as often in the life cycle as possible?
» Are security fault injection used to check for safety-threatening faults?
» Are the software's interfaces tested using Fuzz testing?
» Of the tools being used, what are the rates of false-positives and false-negatives?
» Is the system scanned for vulnerabilities, using vulnerability scanners? Are the vulnerability scanners' signatures frequently updated?
» Is Penetration Testing performed on the software system? Is it black box, white box, or grey box penetration testing?

Conclusion

This pocket guide compiles effective software security testing techniques and offers guidance on when they should be employed during the SDLC. The material in this pocket guide can also be used as a modified template for implementing software security testing and as a measure of how an enterprise's methods stack up against the industry's known best practices.

The Software Assurance Pocket Guide Series is developed in collaboration with the SwA Forum and Working Groups and provides summary material in a more consumable format. The series provides informative material for SwA initiatives that seek to reduce software vulnerabilities, minimize exploitation, and address ways to improve the routine development, acquisition and deployment of trustworthy software products. Together, these activities will enable more secure and reliable software that supports mission requirements across enterprises and the critical infrastructure.

For additional information or contribution to future material and/or enhancements of this pocket guide, please consider joining any of the SwA Working Groups and/or send comments to Software.Assurance@dhs.gov. SwA Forums are open to all participants and free of charge. Please visit https://buildsecurityin.us-cert.gov for further information.

No Warranty

Reprints

For more information, including recommendations for modification of SwA pocket guides, please contact Software.Assurance@dhs.gov or visit the Software Assurance Community Resources and Information Clearinghouse: https://buildsecurityin.us-cert.gov/swa to download this document either format (4"x8" or 8.5"x11").

Software Assurance (SwA) Pocket Guide Series

SwA is primarily focused on software security and mitigating risks attributable to software; better enabling resilience in operations. SwA Pocket Guides are provided; with some yet to be published. All are offered as informative resources; not comprehensive in coverage. All are intended as resources for 'getting started' with various aspects of software assurance. The planned coverage of topics in the SwA Pocket Guide Series is listed:

SwA in Acquisition & Outsourcing

I. Software Assurance in Acquisition and Contract Language
II. Software Supply Chain Risk Management & Due-Diligence

SwA in Development

I. Integrating Security into the Software Development Life Cycle
II. Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses
III. Software Security Testing
IV. Requirements Analysis for Secure Software
V. Architecture & Design Considerations for Secure Software
VI. Secure Coding
VII. Security Considerations for Technologies, Methodologies & Languages

SwA Life Cycle Support

I. SwA in Education, Training & Certification
II. Secure Software Distribution, Deployment, & Operations
III. Code Transparency & Software Labels
IV. Assurance Case Management
V. Assurance Process Improvement & Benchmarking
VI. Secure Software Environment & Assurance Ecosystem
VII. Penetration Testing throughout the Life Cycle

SwA Measurement & Information Needs

I. Making Software Security Measurable
II. Practical Measurement Framework for SwA & InfoSec
III. SwA Business Case

SwA Pocket Guides and related documents are freely available for download via the DHS NCSD Software Assurance Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa.