Black Box Security Testing Tools

C. C. Michael, Cigital, Inc. [vita[1]]
Will Radosevich, Cigital, Inc. [vita[2]]

2005-12-28; Updated 2009-07-27 by Ken van Wyk [vita[3]]

L3 / L, M[4]

This document is about black box testing tools. We use this term to refer to tools that take a black box view of the system under test; they do not rely on the availability of software source code or architecture, and in general try to explore the software's behavior from the outside.

Introduction

This document focuses on black box testing technologies that are unique to software security testing. To go beyond that boundary would entail a full discussion of test automation and automated test management support, which is far beyond the intended scope of the document. These other technologies are touched upon, however, in the Evaluation Criteria[17] section.

This document discusses one particular aspect of black box security testing, namely, the use of automated tools during the test process. But any such discussion should begin with a caveat: security testing relies on human expertise to an even greater extent than ordinary testing, so full automation of the test process is even less achievable than in a traditional testing environment. Although there are tools that automate certain types of tests, organizations using such tools should not be lulled into a false sense of security, since such tools 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.

1. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/251-BSI.html (Michael, C. C.)
2. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/252-BSI.html (Radosevich, Will)
6. #dsy261-BSI_intro
7. #dsy261-BSI_buscase
8. #dsy261-BSI_BWG
9. #dsy261-BSI_types
10. #dsy261-BSI_techs
11. #dsy261-BSI_Evaluation-Criteria
12. #dsy261-BSI_SDLC
13. #dsy261-BSI_study
14. #dsy261-BSI_glossary
17. #dsy261-BSI_Evaluation-Criteria

Scope and Intended Audience

This document is meant for security analysts and aims to provide an overview of the capabilities of black box testing tools. The focus is on categorizing important capabilities of these tools and providing some guidance for evaluating them.

Business Case

Black box testing is generally used when the tester has limited knowledge of the system under test or when access to source code is not available. Within the security test arena, black box testing is normally associated with activities that occur during the pre-deployment test phase (system test) or on a periodic basis after the system has been deployed.

Black box security tests are conducted to identify and resolve potential security vulnerabilities before deployment or to periodically identify and resolve security issues within deployed systems. They can also be used as a “badness-ometer” [McGraw 04] to give an organization some idea of how bad the security of their system is. From a business perspective, organizations conduct black box security tests to conform to regulatory requirements, protect confidential and proprietary information, and protect the organization’s brand and reputation.

Businesses have a legitimate reason to be concerned about potential security vulnerabilities within their systems. In 2003, the CERT Coordination Center received 137,529 reports of security incidents [CERT 06].[23] This was a staggering 67.5% increase in the number of reported incidents from the previous year. A great number of these incidents were due to the widespread use of automated attack tools that have simplified security scans and attacks and allowed them to rapidly be employed against Internet-connected computers and applications.

While the number of reported security incidents continues to rise, the CSI/FBI noted that the total monetary loss reported by 144 companies in 2008 was significant at $41,560,992 [Richardson 08]. In addition, CSI/FBI noted that the average financial loss of reporting organizations subjected to theft of proprietary information was $241,000, and those reporting losses due to financial fraud was $463,100.

These figures describe significant financial losses that are the direct result of security incidents. Although security testing on its own is not a suitable substitute for using security best practices throughout the SDLC**,** black box test tools can help an organization begin to understand and address potential security issues within their systems. These tools allow testers to efficiently and in an automated manner conduct security scans for both known and unknown security vulnerabilities that may adversely impact an organization’s business. Armed with the results of the black box test effort, organizations can better understand and address the risks posed to their business.

Application Security Test Tools. The CSI/FBI *2008 Computer Crime and Security Survey* continues to show steady increases in web application incidents, with 11% of survey respondents advising that they had experienced misuse of their web applications during the past year.

Fortunately, a significant number of black box test tools focus on application security related issues. These tools concentrate on security related issues including but not limited to

- input checking and validation
- SQL insertion attacks
- injection flaws
- session management issues
- cross-site scripting attacks
- buffer overflow vulnerabilities
- directory traversal attacks

23. The CERT Coordination Center no longer collects or reports annual incident data; 2003 was the last year the data was reported.

The tools are developed and distributed by a collection of open source communities and for-profit businesses such as the Open Web Application Security Project (OWASP), Cenzic, HP (formerly SPI Dynamics), NT Objectives, IBM (formerly Sanctum), and others. As the trend for web application security incidents increases, the need for these tools to test Internet-enabled applications increases as well.

Pre-Deployment: During the Development Life Cycle. According to NIST, the relative cost of repairing software defects increases the longer it takes to identify the software bug [NIST 02]. For example, NIST estimates that it can cost twenty times more to fix a coding problem that is discovered after the product has been released than it would have cost if discovered during the system test phase, when black box test tools are normally used.

This figure should not be surprising considering the personnel and processes required to address a security issue after deployment. Help desk personnel are required to take trouble calls from the customer; support engineers are required to confirm and diagnose the problem; developers are needed to implement code fixes; QA personnel are called to perform system regression tests; and managers oversee the entire process. There are additional expenses to consider, such as those associated with patch distribution and the maintenance of multiple concurrently deployed versions. Additionally, a serious post-deployment software vulnerability may result in potential business issues, such as damage to brand or company reputation, and potential legal liability issues.

Accordingly, black box security test tools can be used during the system test phase to identify and address these issues, reduce system development costs, and reduce business risks associated with company reputation and liability.

Post-Deployment: An Evolving Challenge. Unfortunately, the specific instance of security vulnerabilities is constantly changing. For example, during the first quarter of 2009, each month NIST catalogued approximately 568 new security vulnerabilities [NIST 09]. By comparison, CERT cataloged 7,236 vulnerabilities in 2007. These figures highlight the challenges all organizations face when using software in the conduct of their business.

Many of these failures are a direct result of improper security designs or implementation errors that are introduced during development. Organizations that purchase software do not have control over these issues. Fortunately, many black box security test tools are periodically updated to test for these newly discovered vulnerabilities, allowing businesses to conduct periodic security tests of their systems.

Benefits and Limitations of Black Box Testing. As previously discussed, black box tests are generally conducted when the tester has limited knowledge of the system under test or when access to source code is not available. On its own, black box testing is *not* a suitable alternative for security activities throughout the software development life cycle. These activities include the development of security-based requirements, risk assessments, security-based architectures, white box security tests, and code reviews. However, when used to complement these activities or to test third-party applications or security-specific subsystems, black box test activities can provide a development staff crucial and significant insight regarding the system’s design and implementation.

Black box tests can help development and security personnel

- identify implementation errors that were not discovered during code reviews, unit tests, or security white box tests
- discover potential security issues resulting from boundary conditions that were difficult to identify and understand during the design and implementation phases
- uncover security issues resulting from incorrect product builds (e.g., old or missing modules/files)
- detect security issues that arise as a result of interaction with underlying environment (e.g., improper configuration files, unhardened OS and applications)

Accordingly, black box security test efforts complement the critical security activities throughout the SDLC. The tools help developers and security personnel verify that the system security components are operating properly and also identify potential security vulnerabilities resulting from implementation errors. Additionally, black box security tests can help security practitioners test third-party components that may be considered for integration into the overall system and for which source code is not available. These tests may help the development staff uncover potential security vulnerabilities and make intelligent decisions about the use of certain products within their overall system.

Although these tests should not be considered a substitute for techniques that help developers build security into the product during the design and implementation stages, without these tests, developers may overlook implementation issues not discovered in earlier phases. Despite the best efforts of the development staff, mistakes do occur—coding errors, incorrect components in the latest software build, unexpected interaction with the deployed environment, and boundary conditions, to name a few. Black box security tests provide a method to validate the security of the system before it is deployed.

Black box testing tools provide various types of automated support for testers. They help testers work more efficiently by automating whatever tasks can be automated, and they also help testers avoid making mistakes in a number of tasks where careful bookkeeping is needed. Their main roles include

- test automation: providing automated support for the actual process of executing tests, especially tests that have already been run in the past but are being repeated
- test scaffolding: providing the infrastructure needed in order to test efficiently
- test management: various measurements and scheduling and tracking activities that are needed for efficient testing even though they are not directly involved in the execution of test cases

Black Box, White Box, and Gray Box Testing

In this document, we use the term “black box testing” to mean test methods that are not based directly on a program’s architecture source code. The term connotes a situation in which either the tester does not have access to the source code or the details of the source code are irrelevant to the properties being tested. This means that black box testing focuses on the externally visible behavior of the software. For example, it may be based on requirements, protocol specifications, APIs, or even attempted attacks.

In some formulations, a black box tester has access only to the application’s user interface, either for entering data or observing program behavior. We will not adopt this viewpoint here, since one of the main points of black box security testing is to get some idea of what an *attacker* could do to an application. In many situations, attackers are not constrained to interact with a program through the UI (though many do).

Black box testing is different from white box testing, which is testing based on knowledge of the source code. In fact, white box tests are generally *derived from* source code artifacts in some way or another. For example, the tests might target specific constructs found in the source code or try to achieve a certain level of code coverage.

Between black box and white box testing lies gray box testing, which uses *limited* knowledge of the program internals. In principle this could mean that the tester knows about some parts of the source code and not others, but in practice it usually just means that the tester has access to design artifacts more detailed than specifications or requirements. For example, the tests may be based on an architecture diagram or a state-based model of the program’s behavior.

Types of Test Tools

Black box test activities almost universally involve the use of tools to help testers identify potential security vulnerabilities within a system. Among the existing available toolsets, there are subsets of tools that focus on specific areas, including network security, database security, security subsystems, and web application security.

Network security based test tools focus on identifying security vulnerabilities on externally accessible network-connected devices such as firewalls, servers, and routers. Network security tools generally begin by using a port scanner to identify all active devices connected to the network, services operating on the hosts, and applications running on each identified service.

Some network scanning tools also perform vulnerability scanning functions. That is, they identify specific security vulnerabilities associated with the scanned host 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 OS and application related vulnerabilities resulting from poor implementation or configuration. Each vulnerability provides a potential opportunity for an attacker to gain unauthorized access to the system or its resources. These tools have historically been associated with penetration testing, which is not covered in this document.

Database security test tools center on identifying vulnerabilities in a systems database. These can be the result of incorrect configuration of the database security parameters or improper implementation of the business logic used to access the database (e.g., SQL insertion attacks). These vulnerabilities may result in the disclosure or modification of sensitive data in the database. Database scanning tools are generally wrapped in network security or web application security scanning tools and will not be specifically discussed in this document.

Security subsystem tools identify security vulnerabilities in specific subsystems. Whereas the previously discussed tools are used after the system has been developed, these tools are used during the implementation cycle to test whether security-critical subsystems have been designed and implemented properly. As an example, these tools test for correct operation of random number generators (e.g., NIST Statistical Test Suite[37] for random number generation), cryptographic processors, and other security-critical components.

Web application security tools highlight security issues within applications accessed via the Internet. Unlike network security tools, application security tools generally focus on identifying vulnerabilities and abnormal behavior within applications available over ports 80 (HTTP) and 443 (HTTPS). These ports are traditionally allowed through a firewall to support web servers. Note that many of these tools can also test Web Services based application technologies over the same ports.

Through the years, IT managers and security professionals have learned to secure the network perimeter by installing and maintaining firewalls and other security appliances, securely configuring host machines and their applications, and enforcing a software patch management solution to address software vulnerabilities. At the same time, attackers and security professionals alike have identified a new class of security vulnerabilities within web based applications. This new class of security vulnerabilities cannot be controlled by the firewall and must be addressed with proper application design and implementation. The Gartner Group reports that up to 75% of all web-based attacks are now conducted through the open web application ports 80 and 443. As a result, web-based applications must be designed in a manner that does not permit an attacker to take advantage of an application’s vulnerability. These vulnerabilities can result from a number of issues, including

- improper input validation
- parameter injection and overflow
- SQL injection attacks
- cross-site scripting vulnerabilities
- cross-site request forgeries
- directory traversal attacks
- buffer overflows
- inappropriate trust (i.e. client side)
- poor session management
- improper authorization and access control mechanisms

Application security test tools can be used to help identify potential security vulnerabilities within commercial and proprietary based web applications. The tools are frequently used in both the pre-deployment and post-deployment test cycles. A development staff can use application security tools to test their web-based applications prior to deployment. Pre-deployment testing allows the development staff to

37. http://csrc.nist.gov/rng/rng2.html

investigate and resolve noted vulnerabilities and abnormal or interesting test results. The test tools can also be used post-deployment by the developer or the developer's customer to periodically test and monitor the deployed system.

Some of these tools provide rather sophisticated functionality, including capabilities to develop and enforce organization security policies, the ability to create custom rules, the automated scheduling of application security tests, and comprehensive vulnerability databases that attempt to address zero-day attacks.

These tools are created and offered by both open source communities and commercial companies. As previously stated, a number of black box testing tools provide tests that focus on several of these test areas. It is important to note that these tools are different than the plethora of source code scanning and binary/bytecode scanning tools. Although the use of source code and binary/bytecode scanning tools is considered an important element of a robust security engineering process, these tools are not considered black box testing tools and will not be discussed in this section.

Commercial Tools

The following is a sample of commercially available application security black box test tools. The list is intended to familiarize the reader with various tools on the market and to encourage the reader to conduct independent review of application security tool capabilities.

Cenzic Hailstorm[41]

IBM (formerly Internet Security Systems) Internet Security Scanner[42]

NT Objectives NTOSpider[43]

IBM (formerly Watchfire and Santum) Appscan[44]

Security Innovation[45] (Holodeck)

HP (formerly SPI Dynamics) WebInspect[46], DevInspect[47]

Open Source/Freeware

The following is a brief sample list of open source and freeware application security scanning and testing tools.

Nikto[50]

Odysseus[51]

OWASP WebScarab[52]

Paros Proxy[53]

SPIKE[54]

41. http://www.cenzic.com/products/software/overview/
42. http://www.iss.net/products_services/enterprise_protection/vulnerability_assessment/scanner_internet.php
43. http://www.ntobjectives.com/products/ntospider.php
44. http://www-01.ibm.com/software/awdtools/appscan/
45. http://www.sisecure.com/company/ourtechnology/index.shtml
46. http://welcome.hp.com/country/us/en/prodserv/software.html
47. http://welcome.hp.com/country/us/en/prodserv/software.html
50. http://www.cirt.net/code/nikto.shtml
51. http://www.bindshell.net/tools/odysseus
52. http://www.owasp.org/software/webscarab.html
53. http://www.parosproxy.org/index.shtml
54. http://www.immunitysec.com/resources-freesoftware.shtml

Tool Selection

Selecting a black box test tool can be a challenging task due to the wide array of available commercial vendors and open source projects in this area. There are a number of high-level considerations that you should contemplate before selecting a tool that is useful for your specific application and organization:

- test coverage and completeness
- accuracy or “false-positive” rate
- capacity and “freshness” of vulnerability database
- ability to create custom tests
- ease of use
- reporting Capabilities
- cost

A list of criteria that one may consider before selecting a black box test tool is included in Section Evaluation Criteria[65].

Technologies for Black Box Security Testing

Not surprisingly, black box testing for security has a different technological focus than traditional black box testing. [Fink 04] defines positive requirements as those requirements that state what a software system should do, while negative requirements state what it should not do. Although security testing deals with positive requirements as well as negative ones, the emphasis is on negative requirements. In contrast, traditional software testing focuses on positive requirements. This difference in emphasis is reflected in the test tools that support black box test activities.

The technology incorporated in such tools can be classified as follows, according to its functionality:

- **fuzzing:** the injection of random or systematically-generated data at various interfaces, with various levels of human intervention to specify the format of the data
- **syntax testing:** generating a wide range of legal and illegal input values, usually with some knowledge of the protocols and data formats used by the software
- **exploratory testing**: testing without specific expectation about test outcomes, and generally without a precise test plan
- **data analysis:** testing the data created by an application, especially in the context of cryptography
- **test scaffolding:** providing testers with support tools they need in order to carry out their own black box tests. For example, if the tester wants to inject a certain error code when an application tries to open a pipe, support technology is needed to actually carry out this test.
- **monitoring program behavior:** When a large number of tests are automatically applied, it is useful to also have automatic techniques for monitoring how the program responds. This saves testers from having to check for anomalous behavior manually. Of course, a human is better at seeing anomalous behavior, but the anomalies that signal the presence of a security vulnerability are often quite obvious.

In this section, we do not discuss test automation technology, which is a standard technology used to automate the execution of tests once they have been defined. It is technology for traditional testing, and this fact makes it too broad of a subject to cover within the intended scope of this document. However, any extensive treatment of software testing also covers test automation, and the reader may consult standard references on software testing such as [Beizer 95], [Black 02], and [Kaner 93].

Fuzzing

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. For example, one might intercept system calls

65. #dsy261-BSI_Evaluation-Criteria

made by the application while reading a file and make it appear as though the file contained random bytes. 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.

Since the idea was originally introduced, the informal definition of fuzzing has expanded considerably, and it can also encompass domain testing, syntax testing, exploratory testing, and fault injection. This has the unfortunate consequence that when one author denigrates fuzzing (as in [McGraw 04]) while another extols it (as in [Faust 04]), the two authors might not be talking about the same technology[85]. The current section is partly meant to emphasize that in this document, "fuzzing" is used in the narrow sense implied by [Miller 90].

Fuzzing, according to the first, narrower definition, 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, imagine 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.

For this and other reasons, 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. Microsoft refers to this as "smart fuzzing," [Howard 2006] in which the fuzzing tools (and test staff) have significant knowledge of the fuzz target. 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 discussed in the previous paragraph. In addition, fuzzing tools often let the tester systematically explore the input space; for example, the tester might be able to specify a range of input values instead of having to rely on randomly generated noise. As a result, there is a considerable overlap between fuzzing and syntax testing, which is the topic of the next section.

Syntax Testing

Syntax testing [Beizer 90] refers to testing that is based on the syntactic specification of an application's input values. The idea is to determine what happens when inputs deviate from this syntax. For example, the application might be tested with inputs that contain garbage, misplaced or missing elements, illegal delimiters, and so on. In security testing, one might present a web-based application with an HTTP query containing metacharacters or JavaScript, which in many cases should be filtered out and not interpreted. Another obvious syntax test is to check for buffer overflows by using long input strings.

Syntax testing helps the tester confirm that input values are being checked correctly, which is important when developing secure software. On the other hand, syntactically correct inputs are also necessary for getting at the interesting parts of the application under test, as opposed to having the test inputs rejected right away, like the random-character URL in the section on fuzzing.

85. This issue is not unique to fuzzing. When acquiring a testing tool (or any other technology), it is therefore wise to evaluate it according what it actually does, and not according to the names it gives to its capabilities.

Typically, it is possible to automate the task of getting inputs into the right form (or into almost the right form, as the case may be). This lets the tester focus on the work of creating test cases instead of entering them in the right format.

However, the degree of automation varies. It is common for testers to write customized drivers for syntax testing, which is necessary when the inputs have to be in an application-specific format. Test tools can provide support for this by letting the tester supply a syntax specification and automating the creation of a test harness based on that syntax. On the other hand, the tool may also come with a prepackaged awareness of some common input formats.

In security test tools, there is a certain emphasis on prepackaged formats because many applications communicate across the network using standard protocols and data formats. It makes sense for a security test tool to be aware of widely used protocols, including HTTP, FTP, SMTP, SQL, LDAP, and SOAP, in addition to supporting XML and simplifying the creation of malicious JavaScript for testing purposes. This allows the tool to generate test input that *almost* makes sense but contains random values in selected sections. Creating useful syntax tests can be a complex task because the information presented at the application interface might be mixed, perhaps containing SQL or JavaScript embedded in an HTTP query.

Many attacks are injection attacks, where a datastream that is in one format according to the specification actually contains data in another format. Specifically, most data formats allow for user-supplied data in certain locations, such as SQL queries in a URI. The embedded data may be interpreted by the application, leading to vulnerabilities when an attacker customizes that data. Such vulnerabilities may or may not be visible in the design; a classic example of where they are not visible is when a reused code module for interpreting HTML also executes JavaScript.

One important variant of syntax testing is the detection of cross-site scripting vulnerabilities. Here, the actual interpreter is in the client application and not the server, but the server is responsible for not allowing itself to be used as a conduit for such attacks. Specifically, the server has to strip JavaScript content from user-supplied data that will be echoed to clients, and the same goes for other data that might lead to undesired behavior in a client application. Testing for cross-site scripting vulnerabilities (see [Hoglund 02]) amounts to ensuring that dangerous content really is being stripped before data is sent to a client application, and this, too, involves specially formatted data.

Automated support for syntax testing may or may not provide a good return on investment. Good security testing requires a certain level of expertise, and a security tester will probably be *able* to write the necessary support tools manually. Custom data formats make it necessary to write some customized test harnesses in any event. It may also be cost effective to write in-house test harnesses for *standard* protocols, since those harnesses can be reused later on, just as third-party test harnesses can. Although in-house test drivers do not usually come into the world with the same capabilities as a third-party test application, they tend to evolve over time. Therefore, the amount of effort that goes into the development and maintenance of in-house test drivers diminishes over time for commonly used data formats. In spite of these factors, third-party tools often can have usability advantages, especially compared to in-house tools being used by someone who did not develop them originally.

Exploratory Testing and Fault Injection

In security testing, it may sometimes be useful to perform tests without having specific expectations about the test outcome. The idea is that the tester will spot anomalies—perhaps subtle ones—that eventually lead to the discovery of software problems or at least refocus some of the remaining test effort. This contrasts with most other testing activities because usually the test plan contains information about what kind of outcomes to look for. Exploratory testing is discussed in depth in [Whittaker 02] and [Whittaker 03].

There is no technical reason why a test plan cannot map out these tests in advance, but in practice many testers find it useful to let the outcome of one test guide the selection of the next test. In a sense, the tester is exploring the software's behavior patterns. This makes sense because a subtle anomaly may create the need to collect further information about what caused it. In a black box test setting, getting more information implies doing more tests. This leads to the concept of exploratory testing.

Most test technologies that support exploratory testing can also be used for other test activities, but some techniques are associated more closely with exploratory testing than with other types of testing. For example, fuzzing (in the narrow sense of the word described earlier) falls into this category, because usually testers don't have any exact idea of what to expect. More generally, certain test techniques make it hard to say exactly what anomalous behavior might occur even though there is interest in seeing how an application will respond. Some of the other techniques that fall into this category are:

Security stress testing, which creates extreme environmental conditions such as those associated with resource exhaustion or hardware failures. During traditional stress testing, the idea is to make sure that the application can continue to provide a certain quality of service under extreme conditions. In contrast, during security testing it may be a foregone conclusion that the application will provide poor service—perhaps good performance under stress is not a requirement—and the tester might be looking for other anomalies. For example, extreme conditions might trigger an error-handling routine, but error handlers are notorious for being under-tested and vulnerable. As a second example, slow program execution due to resource exhaustion might make race conditions easier to exploit, along with other quality and security related issues such as concurrency. Problems like race conditions and concurrency issues can be problematic to find and to test for without being able to produce heavy load situations like might be found in production environments. In many cases, an attacker might be able to create whatever extreme conditions are needed for an attack to succeed. Thus, stress testing should be considered an important part of the security testing process, particularly in heavily multithreaded computing environments.

Fault injection, which directly modifies the application's internal state [Voas 97]. Fault injection is often associated with white box testing, since it references the program's internal state, but in practice certain types of test modify external data so close to the program's inner workings that they can also be regarded as fault injection. For example, the tester might intercept calls to the operating system and interfere with the data being passed there. Interfering in communication between executable components might also be regarded as a black box technique.

Fault injection can clearly be used for stress testing, but it can also be used to help a tester create conditions with relative ease that an attacker might create with greater effort. For example, if the tester interferes with interprocess communication, it might approximate a situation where one of the communicating processes has been subverted by an attacker. Likewise, intercepting calls to the operating system can be used to simulate the effects of an attacker getting control of external resources, since system calls are used to access those resources. It is not always clear *how* an attacker might exploit problems found using fault injection, but it can still be useful to know that those problems are there. Of course, some of the resulting tests might also be unfair—for example, an attacker intercepting system calls could manipulate all of the application's memory —and in the end the tester has to ensure that the test results are meaningful in an environment where the operating system protects the application from such attacks.

Data Analysis Capabilities

By data analysis, we mean the process of trying to understand a program's internals by examining the data it generates. This might be followed by an attempt to go beyond mere observation and influence the program's behavior as well. One of the concerns of black box security testing is to try performing this type of analysis in order to determine whether an attacker could do the same thing.

Two particularly salient issues are

- Stateless protocols use external mechanisms to keep track of the state of a transaction (HTTP uses cookies, for example). It is not always desirable to expose this state information to a potential attacker, but data analysis can be used to deduce state information at the black box level.
- It is sometimes necessary to use random numbers to generate cryptographically secure keys or hashes on the fly. If an attacker can collect outputs from a weak random number source and analyze those outputs sufficiently well to predict future random bits, even a strong cryptographic algorithm can be compromised.

A related issue that will not be discussed at great length is that random numbers are used in computerized casino gaming, and an attacker who can predict these numbers—even partially—may be able to cheat.

In each of these cases, the security issue is the ability to generate random numbers that prevent the attacker from seeing patterns or predict future values. As a rule, this issue should be addressed in the design phase—using weak random number generation is a design flaw—but testing still plays its usual roles. For example, it can be used to probe whether the design is implemented correctly or to examine third-party components whose source code is unavailable. In one case, a municipality needed secure random numbers to secure a specific aspect of law-enforcement-related communications, but problems were encountered in obtaining the necessary source code from a third-party vendor. Black box testing was used to achieve a minimal level of due diligence in the question of whether the random numbers were unpredictable.

Cookie analysis deserves its own discussion. It consists of deducing how a web application uses cookies in order to examine the application's inner workings, or even to hijack sessions by predicting other users' cookie values. In a well-designed system this should not lead to an immediate compromise—after all, truly sensitive information should be protected by SSL or a similar mechanism—but cookie analysis can provide the toehold that an attacker needs in order to launch a more damaging attack. Of course, not all software systems are well designed, and some are vulnerable to direct compromises using cookie analysis, or even simple replay attacks involving cookies.

These issues lead to the idea of randomness testing, which is within the scope of black box testing.

Some black box testing tools provide simple statistical tests and visualization tools to support cookie analysis. Furthermore, the analysis and detection of cryptographically weak random-number schemes is not purely the domain of software security, and this works to the advantage of the black box tester because it makes more technology available for that task. For example, weak random number generation is often used to generate events in software-based simulations, an application in which speed is more important than security. This creates a need to know exactly what the weaknesses of the random number generator are so that they do not bias the simulation.

There are some standard software packages for evaluating randomness empirically: the NIST battery[115], the Diehard battery[116], and ent[117].

As a final note, testers should be aware that even if a random number source passes these test batteries, this does not imply that the source is cryptographically secure. As in many other areas, testing can only demonstrate the presence of problems, not their absence.

Monitoring Program Behavior

Monitoring program behavior is an important part of any testing process because there must be a way to determine the test outcome. This is often referred to as *observability*. Usually it means examining the behavior of the program under test and asking whether this observed behavior is symptomatic of a vulnerability in the software. This examination can be harder in security testing than it is in traditional testing, because the tester is not necessarily comparing actual program behavior to expectations derived from specifications. Rather, the tester is often looking for unspecified symptoms that indicate the presence of unsuspected vulnerabilities. Nonetheless, there are cases in which the unusual behavior sought by a security tester can be specified cleanly enough to test for it automatically.

For example, if a web application is being tested for cross-site scripting vulnerabilities, an attacker's ability to make the application echo externally supplied JavaScript is enough to indicate a possible problem. Likewise, a series of tests meant to detect potential buffer overflows may just require the application to be monitored for crashes.

115. http://csrc.nist.gov/rng/
116. http://en.wikipedia.org/wiki/Diehard_tests
117. http://www.fourmilab.ch/random/

There are many test automation tools with the ability to monitor program outputs and behavior. In selecting a black box testing tool, it may be useful to consider whether a tool either provides its own monitoring capabilities or integrates with other existing test automation frameworks.

Another aspect of behavior monitoring is that for security testing, one may have to observe black box behavior that is not normally visible to the user. This can include an application's communication with network ports or its use of memory, for example. This functionality is discussed in the next section, which deals with test support tools. A fault injection tool may also support this type of monitoring because the underlying technologies are similar.

A final issue that also applies to traditional testing is that automation is quite useful for spotting anomalous test outcomes. This is especially true during high-volume test activities like fuzzing. In security testing, a great deal of reliance is placed on the tester's ability to see subtle anomalies, but the anomalies are not *always* too subtle for automated detection. Thus, some test tools automate monitoring by letting the tester specify in advance what constitutes anomalous behavior.

Test Scaffolding

By *test scaffolding* we mean tools that support the tester's activities, as opposed to actually generating data. This primarily includes test management technology, but test management is in the domain of traditional test automation and we do not cover it here. Instead, we focus on technology for observing and/or influencing application behavior in ways that would not normally be possible for an ordinary user or tester.

Technologies for observing program behavior are quite common, since they are needed for numerous other purposes as well, such as debugging and performance monitoring. Of course, their utility in test automation depends somewhat on how easily they can be integrated with other test tools, especially those for monitoring program behavior. Thus debuggers, which are usually interactive, can provide testers with valuable information but might be a bottleneck during automated testing. On the other hand, text-based tools can have their outputs postprocessed even if they are not explicitly supported by a testing tool, while some graphical tools might allow a tester to observe anomalies even with a rapid-fire series of automated tests.

There are some testing tools, notably the Holodeck system [Whittaker 02,Whittaker 03], that already include test scaffolding of this kind.

Evaluation Criteria

The following is a list of evaluation criteria that may be considered when selecting a black box security testing tool. Many of the criteria listed here are from Appendix B of [Dustin 01][129]. Readers are encouraged to consult this original document as well, since it gives an expanded list of evaluation criteria and also provides evaluation results for several major test tool suites (albeit not security-specific test tools). Not all of the criteria listed below may be relevant to all test organizations or all test projects. In addition to the criteria listed here, organizations may also want to consider support for the specific black box security testing technologies described previously in this document.

1. Ease of Use
    1. Intuitive and easy to use for users new to automated testing tools
    2. Easy to install; tool may not be used if difficult to install
    3. Tasks can be accomplished quickly, assuming basic user proficiency
    4. Easy to maintain automated tests, with a central repository that enables users to separate GUI object definitions from the script
    5. Can vary how designs and documents are viewed (zooming, multipage diagrams easily supported, multiple concurrent views); basic windowing
2. Tool Customization

129. http://www.uml.org.cn/Test/12/Automated%20Testing%20Tool%20Evaluation%20Matrix.pdf

    1. Fully customizable toolbars to reflect any commonly used tool capabilities
    2. Tool customizable: fields added, deleted
    3. Fully customized editor with formats and colors for better readability
    4. Tool support for required test procedure naming convention
3. Breadth of Testing
    1. Can be used with non-Microsoft platforms (UNIX, Linux, FreeBSD, Mac)
    2. Tests for common website vulnerabilities
    3. Evaluates the test environment as well as the software
    4. Supports standard web protocols for fuzzing and domain testing.
4. Test Coverage and Completeness
    1. Coverage refers to the ability of the tools to test for all (known) categories of vulnerabilities relevant to the product that has been developed. It is important to obtain a sense of the percentage and nature of potential vulnerabilities the tools tests for. For example, if evaluating a web-based system, the organization will want to determine whether the test tool identifies issues that may result from improper input validation, SQL insertion attacks, cross-site scripting attacks, or improper session management.
5. Accuracy/False-Positive Rate
    1. Is there a large number of false positives? False positives will result in more analysis work for the tester, who will be required to manually evaluate the results of the test tool.
    2. Is there a large number of unidentified vulnerabilities?
6. Test Language Features
    1. Allows add-ins and extensions compatible with third-party controls
    2. Does not involve additional cost for add-ins and extensions
    3. Has a test editor/debugger feature
    4. Test scripting language flexible yet robust; allows for modular script development
    5. Scripting language not too complex
    6. Scripting language allows for variable declaration and use and for parameter to be passed between functions
    7. A test script compiler or an interpreter used?
    8. Allows for interfacing and testing of external .dll and .exe files
    9. Published APIs: Language Interface Capabilities
    10. Tool is not intrusive: source code of application does not need to be expanded by inserting additional statements or dlls for the application to be compatible with the tool
    11. Allows for data-driven testing
    12. Allows for automatic data generation
    13. Allows for adding timers for timing transaction start and end
    14. Allows for adding comments during recording
    15. Allows for automatic or specified synchronization between client and server
    16. Allows for object data extraction and verification
    17. Allows for database verification
    18. Allows for text (alphanumeric) verification
    19. Allows for wrappers (shells) whereby multiple procedures can be linked and called from one procedure
    20. Allows for automatic data retrieval from any data source—RDBMS, legacy system, spreadsheet—for data-driven testing
    21. Allows for use of common spreadsheet for data-driven testing

    22. Ease of maintaining scripts when application changes

7. Test Management
    1. Supports test execution management
    2. Support for industry standards in testing processes (e.g., SEI/CMM, ATLM, ISO)
    3. Interoperability with tools being used to automate traditional testing
    4. Application requirements management support integrated with the test management tool
    5. Requirements management capability supports the trace of requirements to test plans to provide requirement coverage metrics
    6. Test plans can be imported automatically into test management repository from standard text files
    7. Can be customized to organization’s test process
    8. Supports planning, managing, and analyzing testing efforts; can reference test plans, matrices, product specifications, in order to create traceability
    9. Supports manual testing
    10. Supports the migration from manual to automated scripts
    11. Can track the traceability of tests to test requirements
    12. Has built-in test requirements modules
    13. Can check for duplicate defects before logging newly found defects
    14. Allows for measuring test progress
    15. Allows for various reporting activities
    16. Allows for tracking of manual and automated test cases
    17. Has interface to software architecture/modeling tool
    18. Is integrated with unit testing tools
    19. Has interface to test management tool
    20. Has interface to requirements management tool
    21. Has interface to defect tracking tool
    22. Has interface to configuration management tool
    23. Provides summary-level reporting
    24. Includes error filtering and review features
    25. Enables metric collection and metric analysis visualization
8. Interoperability
    1. Major test automation suites provide functionality that is useful in any large-scale testing process. For smaller, more specialized tools, interoperability with other test tool suites may be considered as an evaluation criterion.
9. Load and Stress Test Features
    1. All users can be queued to execute a specified action at the same time
    2. Automatic generation of summary load testing analysis reports
    3. Ability to change recording of different protocols in the middle of load-recording session
    4. Actions in a script can be iterated any specified number of times without programming or rerecording of the script
    5. Different modem connection speeds and browser types can be applied to a script without any rerecording
    6. Load runs and groups of users within load runs can be scheduled to execute at different times
    7. Automatic load scenario generation based on load testing goals: hits/second, number of concurrent users before specified performance degradation, and so on
    8. Cookies and session IDs automatically correlated during recording and playback for dynamically changing web environments

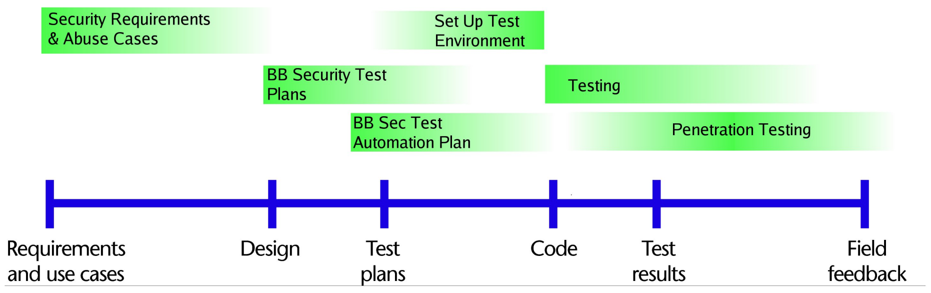
    9. Allows for variable access methods and ability to mix access methods in a single scenario: modem simulation or various line speed simulation
    10. Ability to have data-driven scripts that can use a stored pool of data
    11. Allows for throttle control for dynamic load generation
    12. Allows for automatic service-level violation (boundary value) checks
    13. Allows for variable recording levels (network, web, API, and so on)
    14. Allows for transaction breakdown/drill-down capabilities for integrity verification at the per client, per session, and per instance level for virtual users
    15. Allows for web application server integration
    16. Supports workload, resource, and/or performance modeling
    17. Can run tests on various hardware and software configurations
    18. Support headless virtual user testing feature
    19. Requires low overhead for virtual user feature (web, database, other?)
    20. Scales to how many virtual users?
    21. Simulated IP addresses for virtual users
    22. Thread-based virtual user simulation
    23. Process-based virtual user simulation
    24. Centralized load test controller
    25. Allows for reusing scripts from functional test suite
    26. Support for WAP protocol testing against WAP Gateway or web server
    27. Compatible with SSL recording
    28. Compatible with which network interaction technologies? (e.g., streaming media, COM, EJB, RMI, CORBA, Siebel, Oracle, SAP)
    29. Compatible with which platforms? (e.g., Linux, UNIX, NT, XWindows, Windows CE, Win3.1, Win95, Win98, Win2000, WinME)
10. Monitor Test Features
    1. Monitors various tiers: web server, database server, and app server separately
    2. Supports monitoring for which server frameworks? (e.g., ColdFusion, Broadvision, BEA WebLogic, Silverstream, ATG Dynamo, Apache, IBM Websphere, Oracle RDBMS, MS SQL Server, Real Media Server, IIS, Netscape Web Server
    3. Supports monitoring of which platforms? (e.g., Linux, NT, UNIX, XWindows, Windows CE, Win3.1, Win95/98, Win2000)
    4. Monitors network segments
    5. Supports resource monitoring
    6. Synchronization ability in order to determine locking, deadlock conditions, and concurrency control problems
    7. Ability to detect when events have completed in a reliable fashion
    8. Ability to provide client-to-server response times
    9. Ability to provide graphical results and export them to common formats
11. Consulting Requirements
    1. Maturity of vendor
    2. Market share of vendor
12. Vendor Qualifications
    1. Financial stability of vendor
    2. Length of time in business
    3. Technological maturity

13. Vendor Support
    1. Software patches provided, if deemed necessary
    2. Upgrades provided on a regular basis
    3. Upgrades backward compatible: scripts from previous version can be reused with later version
    4. Training available
    5. Help feature available; tool well documented
    6. Tech support reputation throughout industry
    7. No consulting needed?
    8. Availability of and access to tool user groups
14. Product Pricing
    1. Price consistent within estimated price range
    2. Price consistent with comparable vendor products
    3. ROI compared to current in-house technology
    4. ROI compared to in-house development of needed technology

Use Throughout the Software Development Life Cycle

Security testing (and security analysis per se) is often regarded as something that takes place at the end of the software development life cycle. However, far greater success can be achieved by integrating security testing throughout the life cycle. As with any kind of defect, software vulnerabilities are easier and cheaper to address if they are found earlier.

Figure 1. Black box security testing in the software development life cycle. Note that black box test planning can often begin in the design phase due to its comparative independence from source code.



Black box Security Testing and the Requirements/Design Stages

At the current time, potential vulnerabilities arising in the requirements and design phases cannot be detected with automated tools; human expertise is needed here. Nonetheless, some aspects of test automation should be considered during this phase.

Test planning usually begins in the requirements phase of the SDLC (see the module on risk-based and functional security testing). The test plan should include a *test automation plan* as well. This plan describes which tests will be automated and how. The “how” can be an important issue, because in many cases testing does not involve a single, specialized tool but rather a set of general-purpose tools originally intended for other purposes. The functionality that cannot be obtained in this way will have to be obtained from third parties or built internally, and it is good to know as soon as possible what extra capabilities will be acquired.

Of course, test automation planning also includes the decision of *what* testing to automate and what to do manually. Having a clear idea of the test requirements makes it easier to make this decision, since the necessary technology can be identified and priced (perhaps using some of the evaluation criteria listed in this document). Note that many automation requirements can be shared by security testing and traditional testing; indeed many are supplied only by traditional test automation tools, so interoperability needs to be considered. In the case of security testing, where the testers themselves often have quite a bit of wide-ranging expertise, it may be advisable to consult the testers when determining which test activities can be automated in-house (and at what cost), and to determine whether interoperability can be achieved (possibly without the use of explicit APIs).

When estimating the utility of building or acquiring test automation tools, it should of course be kept in mind that some tools might be able to be reused in the future. This is especially true for black box security testing: the fact that it is black box testing makes it less project-dependent because it does not refer to specific code artifacts, while the fact that it is security testing leads to a plethora of test conditions that will have to be recreated in future test projects as testers try to anticipate what an attacker would typically try out.

In many development projects, testing proceeds as a series of test stages, where one module is in the process of being tested while others are still being developed. In such cases, the *test environment* cannot wait until development is finished, but has to be available when the first module is ready for testing. This is another reason to begin collecting the necessary tools as soon as possible (e.g., to know during the design stage what the necessary tools will be).

The requirements phase is also the time when abuse cases are collected. Together with attack patterns, these can be used to start designing black box tests.

Black box Security Testing in the Test/Coding Phase

Typically, the coding and testing phase for a software product consists of a series of test stages. Distinct test stages arise because different modules are ready for testing at different times during the life cycle, and also because software modules may be repaired or otherwise modified after testing, so that retesting is needed.

Unit testing refers to the process of testing individual segments of code that are too small to execute independently. The exact definition of unit testing is somewhat variable, but usually it refers to testing individual functions, methods, or classes. Since unit testing generally involves software artifacts that cannot be executed by themselves, it usually requires test drivers. The responsibility for unit testing often falls on the shoulders of those who are most capable of writing the test drivers, namely the developers themselves. Unit testing is not black box testing, but certain black box tools may be useful to help with monitoring software behavior and creating error conditions. If the developers are charged with setting up this support software on their own, the process may be chaotic and may not get done, with the result that unit testing neglects security issues. It is preferable for the testing organization to perform this aspect of setting up the test environment, since it was also the testing organization that outlined the security requirements and decided what test tools should be acquired.

QA acceptance testing, also known by a number of other names such as smoke testing, is the process of ensuring that the software is ready to enter the quality assurance process. For example, a module is not actually ready for testing if it fails to compile for the test environment or immediately exhausts memory and crashes.

From the security standpoint, a software module might fail QA acceptance testing for a number of obvious reasons, such as the failure to implement a security requirement that is supposed to be tested. But this test phase is also a good time to test for stupid implementation mistakes. First and foremost, a naïve implementation of some security requirement might be blatantly ineffective and not even attempt to deal with some of the issues that are supposed to be tested. Secondly, QA acceptance testing generally contains an element of ad hoc testing, which also makes it a good time to test for dumb mistakes that might not have been foreseen in the test plan. Finally, problems that are easy to test for should be tested early on (e.g., in the QA acceptance test phase), since it is better to find problems sooner.

For these reasons, black box security testing may play an important role of QA acceptance testing. Fully automated attack simulations and highly automated fuzzing tests are appropriate here, and testers might also use domain testing to pursue intuitions. Like other test phases, QA acceptance testing is likely to require secondary test support tools.

System-level and integration testing are meant to evaluate how the components of the application work together and how the application works in conjunction with other applications. Many types of vulnerabilities may not be apparent until the system under test reaches this stage. For example, suppose the system under test is a web application that creates SQL queries based on user data. For such systems there is a risk of SQL injection vulnerabilities, but it may be that the SQL interface was stubbed out when the user interface was tested, while the SQL interface was tested using a test driver. In principle, one could test for SQL injection vulnerabilities by comparing the input to the user interface with the data that it sends to the SQL stub, but prepackaged, black box test tools might not be able to understand this data or even be able to see it. Furthermore, there is a host of other potential security issues that might not become apparent before the system test or integration test stage even with customized test automation.

At the same time, thorough white box testing [White Box Testing[152]] becomes quite difficult at this stage, to say nothing of static analysis, because the complete system may contain many third-party components, interacting layers implemented in different languages, and communication between different subsystems running on heterogeneous hardware. Thus, black box testing becomes an increasingly important part of the overall test process.

Regression testing is meant to test for recurrences of old bugs. It is common to use regression testing to ensure that bugs have been fixed and that they do not resurface in later versions. By definition, regression testing involves re-executing old tests, so its success depends primarily on how well those tests have been recorded and/or automated. This applies whether it is traditional tests or security tests that are being automated, but if a separate security-testing tool is in use, that tool might have to supply its own infrastructure for recording and automation of past tests. One caveat is that when developers do not fully understand what they are implementing (this seems to happen more often than usual with security, cryptography, and random-number technology), they may write kludges that treat the previous test inputs as special cases. If such a situation is suspected, it may not be appropriate to use fully automated capture-replay-style tests during the regression test phase.

Security as a Cross-Cutting Concern

The above discussion attempted to map out various correlations between security testing and the overall software development cycle. However, this does not mean that security testing should be forced into the same framework used for traditional testing. Instead, security testing should be treated as a cross-cutting concern, even though the entry criteria for certain security test activities might be the same as the entry criteria for traditional test activities.

Case Study

Although it is strongly recommended that an organization does not rely exclusively on black box testing to build security into a system, black box testing, when coupled with other security activities performed throughout the SDLC, can be very effective in validating design assumptions, discovering vulnerabilities associated with the application environment, and identifying implementation issues that may lead to security vulnerabilities.

For example, an organization had assembled a large software development team to build a high-profile Internet-based gaming system. The gaming system was planned to augment an existing, government-sponsored, paper-based gaming system. Understanding the broad and potentially significant security implications relating to the system, the development organization made every effort to design security into the product. A security architect was involved with the development effort from initial requirement generation through system delivery. Security activities were conducted throughout the SDLC to ensure that security was built into the system. These included the following:

- Security-based requirements were developed.
- Security-based risk assessments to identify areas of greatest risk to the business and the technology platform were completed.
- Findings from the risk assessments were addressed in the security architecture and implementation.

152. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/white-box.html (White Box Testing)

- Security-based design and architecture reviews were conducted.
- Security training was provided to developers.
- Code reviews were conducted on security-critical components.

Despite these efforts, an issue associated with the input validation component was identified during system-level security testing. Although input validation was engineered into the overall design and the component had been previously approved in both design and code reviews, there was an issue. The source of the problem was later identified to be associated with the build process. An incorrectly functioning and previously rejected input validation component had made its way into the final build. Had it not been for the final system-level security test activity, the system would have been deployed with the faulty input validation mechanism.

Glossary

acceptance testing	Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component. [IEEE 90]
ad hoc testing	Testing carried out using no recognized test case design technique. [BS-7925]
authentication	The process of confirming the correctness of the claimed identity. [SANS 09]
black box testing	Testing that is based on an analysis of the specification of the component without reference to its internal workings. [BS-7925]
buffer overflow	A buffer overflow occurs when a program or process tries to store more data in a data storage area than it was intended to hold. Since buffers are created to contain a finite amount of data, the extra information —which has to go somewhere—can overflow into the runtime stack, which contains control information such as function return addresses and error handlers.
buffer overflow attack	See **stack smashing**.
bug	See **fault**.
capture/replay tool	A test tool that records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time. [BS-7925]
compatibility testing	Testing whether the system is compatible with other systems with which it should communicate. [BS-7925]
component	A minimal software item for which a separate specification is available. [BS-7925]
conformance testing	The process of testing that an implementation conforms to the specification on which it is based. [BS-7925]
cookie	Data exchanged between an HTTP server and a browser (a client of the server) to store state information on the client side and retrieve it later for

	server use. An HTTP server, when sending data to a client, may send along a cookie, which the client retains after the HTTP connection closes. A server can use this mechanism to maintain persistent client-side state information for HTTP-based applications, retrieving the state information in later connections. [SANS 09]
correctness	The degree to which software conforms to its specification. [BS-7925]
cryptographic attack	A technique for successfully undermining an encryption scheme.
cryptography	Cryptography garbles a message in such a way that anyone who intercepts the message cannot understand it. [SANS 09]
domain	The set from which values are selected. [BS-7925]
domain testing	Testing with test cases based on the specification of input values accepted by a software component. [Beizer 90]
dynamic analysis	The process of evaluating a system or component based on its behavior during execution. [IEEE 90]
encryption	Cryptographic transformation of data (called “plaintext”) into a form (called “cipher text”) that conceals the data’s original meaning to prevent it from being known or used. [SANS 09]
failure	The inability of a system or component to perform its required functions within specified performance requirements. [IEEE 90]
fault	A manifestation of an error in software. A fault, if encountered, may cause a failure. [RTCA 92]
Hypertext Transfer Protocol (HTTP)	The protocol in the Internet Protocol (IP) family used to transport hypertext documents across an internet. [SANS 09]
integration testing	Testing performed to expose faults in the interfaces and in the interaction between integrated components. [BS-7925]
interface testing	Integration testing in which the interfaces between system components are tested. [BS-7925]
isolation testing	Component testing of individual components in isolation from surrounding components, with surrounding components being simulated by stubs. [BS-7925]
National Institute of Standards and Technology (NIST)	A unit of the U.S. Commerce Department. Formerly known as the National Bureau of Standards, NIST promotes and maintains measurement standards. It also has active programs for encouraging and helping industry and science to develop and use these standards. [SANS 09]

negative requirements	Requirements that state what software should not do.
operational testing	Testing conducted to evaluate a system or component in its operational environment. [IEEE 90]
port	A port is nothing more than an integer that uniquely identifies an endpoint of a communication stream. Only one process per machine can listen on the same port number. [SANS 09]
precondition	Environmental and state conditions that must be fulfilled before the component can be executed with a particular input value.
protocol	A formal specification for communicating; the special set of rules that end points in a telecommunication connection use when they communicate. Protocols exist at several levels in a telecommunication connection. [SANS 09]
pseudorandom	Appearing to be random, when actually generated according to a predictable algorithm or drawn from a prearranged sequence.
race condition	A race condition exploits the small window of time between a security control being applied and the service being used. [SANS 09]
regression testing	Retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made. [BS-7925]
requirement	A capability that must be met or possessed by the system/software (requirements may be functional or non-functional). [BS-7925]
requirements-based testing	Designing tests based on objectives derived from requirements for the software component (e.g., tests that exercise specific functions or probe the non-functional constraints such as performance or security). [BS-7925]
reverse engineering	Acquiring sensitive data by disassembling and analyzing the design of a system component [SANS 09]; acquiring knowledge of a binary program’s algorithms or data structures.
risk assessment	The process by which risks are identified and the impact of those risks is determined. [SANS 09]
security policy	A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources. [SANS 09]
server	A system entity that provides a service in response to requests from other system entities called clients. [SANS 09]

session	A virtual connection between two hosts by which network traffic is passed. [SANS 09]
socket	The socket tells a host’s IP stack where to plug in a data stream so that it connects to the right application. [SANS 09]
software	Computer programs (which are stored in and executed by computer hardware) and associated data (which also is stored in the hardware) that may be dynamically written or modified during execution. [SANS 09]
specification	A description, in any suitable form, of requirements. [BS-7925]
specification testing	An approach to testing wherein the testing is restricted to verifying that the system/software meets the specification. [BS-7925]
SQL Injection	SQL injection is a type of input validation attack specific to database-driven applications where SQL code is inserted into application queries to manipulate the database. [SANS 09]
stack smashing	The technique of using a buffer overflow to trick a computer into executing arbitrary code. [SANS 09]
state transition	A transition between two allowable states of a system or component. [BS-7925]
state transition testing	A test case design technique in which test cases are designed to execute state transitions. [BS-7925]
static analysis	Analysis of a program carried out without executing the program. [BS-7925]
static analyzer	A tool that carries out static analysis. [BS-7925]
stress testing	Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements. [IEEE 90]
stub	A skeletal or special-purpose implementation of a software module used to develop or test a component that calls or is otherwise dependent on it. [IEEE 90].
syntax testing	A test case design technique for a component or system in which test case design is based on the syntax of the input. [BS-7925]
system testing	The process of testing an integrated system to verify that it meets specified requirements. [Hetzel 88]
test automation	The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.
test case	A set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program

	path or to verify compliance with a specific requirement. [IEEE 90]
test suite	A collection of one or more test cases for the software under test. [BS-7925]
test driver	A program or test tool used to execute software against a test suite. [BS-7925]
test environment	A description of the hardware and software environment in which tests will be run and any other software with which the software under test interacts when under test, including stubs and test drivers. [BS-7925]
test plan	A record of the test planning process detailing the degree of tester independence, the test environment, the test case design techniques and test measurement techniques to be used, and the rationale for their choice. [BS-7925]
vulnerability	A defect or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. [SANS 09]
web server	A software process that runs on a host computer connected to the Internet to respond to HTTP requests for documents from client web browsers.

References

[Beizer 90]	Beizer, Boris. *Software Testing Techniques*, Chapter 10. New York, NY: van Nostrand Reinhold, 1990 (ISBN 0-442-20672-0).
[Beizer 95]	Beizer, Boris. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. New York, NY: John Wiley & Sons, 1995.
[Binder 99]	Binder, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Addison-Wesley Object Technology Series). Boston, MA: Addison-Wesley Professional**,** 1999.
[Black 02]	Black, Rex. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*, 2nd ed. New York, NY: John Wiley & Sons, 2002.
[BS 7925]	British Computer Society. *Glossary of terms used in software testing* (BS 7925-1).
[Capers 94]	Jones, Capers. Assessment and Control of Software Risks. Englewood Cliffs, NJ: Yourdon Press, 1994.
[CERT 06]	CERT Coordination Center. *CERT/CC Statistics 1988-2006*[168]. 2006.

[DeVale 99]	DeVale, J.; Koopman, P.; & Guttendorf, D. "The Ballista Software Robustness Testing Service[169]," 33-42. 16th International Conference on Testing Computer Software. Washington, D.C., June 14-18, 1999.
[Du 98]	Du, W. & Mathur, A. P. *Vulnerability Testing of Software System Using Fault Injection* (COAST technical report). West Lafayette, IN: Purdue University, 1998.
[Du 00]	Du, W. & Mathur, A. P. "Testing for Software Vulnerability Using Environment Perturbation," 603-612. *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop On Dependability Versus Malicious Faults*. New York, NY, June 25-28, 2000. Los Alamitos, CA: IEEE Computer Society Press, 2000.
[Dustin 99]	Dustin, E.; Rashka, J.; & Paul, J. *Automated Software Testing*. Boston, MA: Addison Wesley Professional, 1999.
[Dustin 01]	Dustin, Elfriede; Rashka; Jeff; McDiarmid, Douglas; & Nielson, Jakob. *Quality Web Systems: Performance, Security, and Usability*. Boston, MA: Addison Wesley Professional, 2001.
[Faust 04]	Faust, S. "Web Application Testing with SPI Fuzzer." SPI Dynamics Whitepaper, 2004.
[Fewster 99]	Fewster, Mark & Graham, Doroty. *Software Test Automation*. Boston, MA: Addison-Wesley Professional**,** 1999.
[Fink 97]	Fink, G. & Bishop, M. "Property-Based Testing: A New Approach to Testing for Assurance." *ACM SIGSOFT Software Engineering Notes 22*, 4 (July 1997): 74-80.
[Friedman 95]	Friedman, Michael A. & Voas, Jeffrey M. *Software Assessment: Reliability, Safety, Testability*. Wiley InterScience, 1995.
[Ghosh 98]	Ghosh, Anup K.; O'Connor, Tom; & McGraw, Gary. "An Automated Approach for Identifying Potential Vulnerabilities in Software," 104-114. *Proceedings of the 1998 IEEE Symposium on Security and Privacy*. Oakland, California, May 3-6, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998.
[Graff 03]	Graff, Mark G. & Van Wyk, Kenneth R. *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596002424).
[Grance 02]	Grance, T.; Myers, M.; & Stevens, M. *Guide to Selecting Information Technology Security*

	Products[170] (NIST Special Publication 800-36), 2002.
[Grance 04]	Grance, T.; Myers, M.; & Stevens, M. *Security Considerations in the Information System Development Life Cycle*[171] (NIST Special Publication 800-64), 2004.
[Guttman 95]	Guttman, Barbara; Roback, Edward. *An Introduction to Computer Security*[172]. Gaithersburg, MD: U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1995.
[Hetzel 88]	Hetzel, William C. *The Complete Guide to Software Testing*, 2nd ed. Wellesley, MA: QED Information Sciences, 1988.
[Hoglund 04]	Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley Professional, 2004.
[Howard 06]	Howard, Michael & Lipner, Steve. *The Security Development Lifecycle*. Redmond, WA: Microsoft Press, 2006, ISBN 0735622142.
[Hsueh 97]	Hsueh, Mei-Chen; Tsai, Timothy K.; & Lyer, Ravishankar K. “Fault Injection Techniques and Tools.” *Computer 30*, 4 (April 1997): 75-82.
[Hunt 99]	Hunt, G. & Brubacher, D. “Detours: Binary Interception of Win32 Functions[174].” *USENIX Technical Program - Windows NT Symposium 99*. Seattle, Washington, July 12-15, 1999.
[IEEE 90]	IEEE. *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990). Los Alamitos, CA: IEEE Computer Society Press, 1990.
[Kaksonen 02]	Kaksonen, R. “A Functional Method for Assessing Protocol Implementation Security[175].” Technical Research Centre of Finland, VTT Publications 48.
[Kaner 99]	Kaner, Cem; Falk, Jack; & Nguyen, Hung Quoc. *Testing Computer Software*, 2nd ed. New York, NY: John Wiley & Sons, 1999.
[Koziol 04]	Koziol, J.; Litchfield, David; Aitel, Dave; Anley, Chris; Eren, Sinan "noir"; Mehta, Neel; & Hassell, Riley. *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes*. New York, NY: John Wiley & Sons, 2004.
[Marick 94]	Marick, Brian. *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*. Upper Saddle River, NJ: Prentice Hall PTR, 1994.

[McGraw 04a]	McGraw, Gary & Potter, Bruce. "Software Security Testing." *IEEE Security and Privacy 2*, 5 (Sept.-Oct. 2004): 81-85.
[McGraw 04b]	McGraw, Gary. "Application Security Testing Tools: Worth the Money?[176]" *Network Magazine*, November 1, 2004.
[Miller 90]	Miller, Barton P.; Fredriksen, Lars; & So, Bryan. "An empirical study of the reliability of UNIX utilities." *Communications of the ACM 33*, 12 (December 1990): 32-44.
[Miller 95]	Miller, B.; Koski, D.; Lee, C.; Maganty, V.; Murthy, R.; Natarajan, A.; & Steidl, J. *Fuzz Revisited: A Re-Examination of the Reliability of Unix Utilities and Services*. Technical report, Computer Sciences Department, University of Wisconsin, 1995.
[NIST 02]	National Institute of Standards and Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing*[177] (Planning Report 02-3). Gaithersburg, MD: National Institute of Standards and Technology, 2002.
[NIST 09]	National Institute of Standards and Technology. National Vulnerability Database[178]. 2009.
[Ricca 01]	Ricca, F. & Tonella, P. "Analysis and Testing of Web Applications," 25–34. *Proceedings of the 23rd IEEE International Conference on Software Engineering*. Toronto, Ontario, Canada, May 2001. Los Alamitos, CA: IEEE Computer Society Press, 2001.
[Richardson 08]	Richardson, Robert. *2008 CSI/FBI Computer Crime and Security Survey*[179]. San Francisco, CA: Computer Security Institute, 2008.
[RTCA 92]	RTCA, Inc. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. Issued in the U.S. by RTCA, Inc. (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B), December 1992.
[Rukhin 01]	Rukhin, Andrew; Soto, Juan; Nechvatal, James; Smid, Miles; Barker, Elaine; Leigh, Stefan; Levenson, Mark; Vangel, Mark; Banks, David; Heckert, Alan; Dray, James; & Vo, San. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*[180]. NIST Special Publication 800-22, 2001.
[SANS 09]	The SANS Institute. *SANS Glossary of Terms Used in Security and Intrusion Detection*[181], 2009.

[SPI 02]	SPI Dynamics. "SQL Injection: Are Your Web Applications Vulnerable?" (white paper). Atlanta, GA: SPI Dynamics, 2002.
[SPI 03]	SPI Dynamics. "Web Application Security Assessment" (white paper). Atlanta, GA: SPI Dynamics, 2003.
[Viega 01]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley Professional, 2001 (ISBN 020172152X).
[Viega 03]	Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596003943).
[Voas 95]	Voas, Jeffrey M. & Miller, Keith W. "Examining Fault-Tolerance Using Unlikely Inputs: Turning the Test Distribution Up-Side Down," 3-11. *Proceedings of the Tenth Annual Conference on Computer Assurance*. Gaithersburg, Maryland, June 25-29, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.
[Voas 98]	Voas, Jeffrey M. & McGraw, Gary. *Software Fault Injection: Inoculating Programs Against Errors*, 47-48. New York, NY: John Wiley & Sons, 1998.
[Wack 03]	Wack, J.; Tracey, M.; & Souppaya, M. *Guideline on Network Security Testing*[182]. NIST Special Publication 800-42, 2003.
[Whalen]	Whalen, Sean; Bishop, Matt; & Engle, Sophie. "Protocol Vulnerability Analysis[183] (draft)."
[Whittaker 02]	Whittaker, J. A. *How to Break Software*. Reading MA: Addison Wesley, 2002.
[Whittaker 03]	Whittaker, J. A. & Thompson, H. H. *How to Break Software Security*. Reading MA: Addison Wesley, 2003.
[Wysopal 03]	Wysopal, Chris; Nelson, Lucas; Zovi, Dino Dai; & Dustin, Elfriede. *The Art of Software Security Testing*. Upper Saddle River, NJ: Pearson Education, Inc.

Cigital, Inc. Copyright

1. mailto:copyright@cigital.com