Introduction

In today's environment, companies have little choice but to invest in information technology security and with the trend of attackers targeting an organizations web application environment specifically, securing web applications has become a top priority. In light of this trend, there are a wide variety of solutions and methodologies delivered by an increasing number of vendors and consulting companies to help organizations mitigate the risk associated with vulnerabilities in their applications. While all of these organizations offer solutions, how do you know that you are covering your organization with the best of breed solutions and that you have covered "all the problems"?

This paper provides an overview of application security testing methodologies. Various approaches with their benefits and limitations are presented to provide a decision maker with a framework for making the best decisions for their organization.

Web Application Vulnerability Classes

While there are no clear delineations between application vulnerability types, there are generally three accepted classes of web application vulnerability as defined in *The Art of Software Security Assessment*:

- Design Vulnerabilities – covering more obscure issues such as logic flaws, authorization problems, authentication vulnerabilities, etc.
- Implementation Vulnerabilities – covering issues such as code injection, command execution, information gathering, error handling, etc.
- Operational and Platform Vulnerabilities – covering issues such as information disclosure, OS buffer overflows/missing patches, service configurations, improper error handling, etc.

Design Vulnerabilities

Design vulnerabilities are fundamental mistakes in the software design. These can result from a lack of oversight in the SDLC process or an omission in the SDLC process to account for vulnerabilities. In these cases, the software does exactly what it was designed to do, but that design creates a vulnerability. These often can be the most devastating from a risk perspective and correcting the mistakes can consume precious resources and time.

Consider the following design vulnerability examples:

Corporate Drug Testing Results: This was a web application portal that displayed results of a drug test and allow other users to view the results. Users were able to swap results by altering the associated first/last name.

Online Tax Filing: A woman in Nebraska with a very common last name typed her name into the site to find her tax filing. She was presented with tax filings for many other people with the same last name. This included social security numbers and bank account numbers. The parent company for the tax filing application was shocked by the discovery and called it "a quirk, an individual circumstance". The link was removed from the application.

Local News Station Public Alerting Service: A component of the site was designed to alert local residents to school and business closings, etc. by running a byline across the screen during news broadcasts. The application failed to re-check user supplied content after the initial clearance. Thus, by modifying previously 'approved' content, non-authorized users were able to post and display their own content on the byline.

Search Engine Marketing Portal: The site was designed with a component that rewards marketing credit dollars for deposits. For example, you would be granted $50 of search engine marketing advertising credits for a $30 deposit. If the user’s $30 deposit subsequently failed, they would still be provided $50 of marketing credit.

Implementation Vulnerabilities

An implementation vulnerability is a programming error or fault in an application that prevents it from behaving as intended. These are often manifested as inappropriate uses of an API. For example, allowing values in a call to default. SQL injection is another common fault. This can be manifested as truncated user supplied data to an API. The use of parameterized or prepared statements or stored procedures can mitigate these types of vulnerabilities.

Operational and Platform Vulnerabilities

These involve configuration files for the applications, servers and any business polices around these applications. Phishing or luring attacks can also be considered a part of this type of vulnerability.

Homeland Security sponsors a site cve.mitre.org that provides a list of common vulnerabilities and exposures that catalogs common ‘known’ vulnerabilities that can fall into this category.

Testing Methodologies and Their Effectiveness

Approaches to testing applications can also be classified using general terminology:

- White Box: Generally referring to privileged access to information and the application source-code, testing in this category can include logical threat modeling, manual code review, automated static code analysis, etc.
- Black Box: Generally referring to the analysis of the application without privileged information and targeting the runtime environment, testing in this category can include automated vulnerability scanning, application penetration testing, etc.
- Grey Box: Generally referring to a combination of approaches that includes elements of both black and white box testing.

White Box Methods

White box methods rely more on manual and conceptual capabilities. An expert team must be engaged and analyzes the environment surrounding the applications, the sensitive data that can be exposed, relevant application entry points and focuses on analysis of the applications source code.

Threat Modeling

Threat modeling is used to identify risks within an application and to assess the business impact of those risks. To begin the process, one must first create a conceptual model of the application. This includes documenting the user roles, user scenarios, protected resources, and application entry points. Brainstorming sessions with the development team are used to identify potential threats and mitigating controls.

Threat analysis modeling tools allow the team to build the conceptual model, document the roles etc, and capture the results of the brainstorming sessions in a structured manner.

The benefit of this approach is that it can quickly identify Design vulnerabilities and can be implemented early in the SDLC process, often before the application is even coded. Catching these issues in the initial phases of the design, as opposed to when the application is in production, can save the organization a lot of time and resources. However, the approach will not find Implementation or Operational Issues and can be very time consuming.

Manual Code Review

A manual code review involves a person or team reviewing the source code. Often a top-down comprehensive review is required, which can involve thousands of lines of code. Text editors, grep utilities, or short scripts are also used to find pointers to possible vulnerabilities.

Having expert security resources review all of the source code is a great benefit as it provides much better coverage of the security risks. Detailed remediation tasks can be planned and the candidate point method can quickly identify standard vulnerabilities. A key benefit is that the team is able to chain together bugs to show more invasive issues that could lead to devastating effects. For example, using a stored cross-site scripting bug to steal the administrator's password and then use SQL injection in the admin website to steal all customer information from the database.

The obvious issue with this approach, however, is that the comprehensive approach can be very time consuming. An average developer can review anywhere from 300 to 3,000 lines of code per hour. With average applications being 40,000 to 400,000 lines of code, this effort can add up quickly. In addition, if the complexity of the code is high, it is often difficult for the developers to detect Design issues. Resources from the application development team may be required to answer questions and the process can expose sensitive IP.

Automated Static Analysis Tools

There are a variety of automated tools that review code by operating on the source code or binary files. The tools attempt to find Implementation vulnerabilities by guessing what the program will do when executed. While very thorough, they are simply looking for patterns in the source code that correspond to known vulnerability patterns. Fortify, Ounce Labs, Veracode and HP DevInspect provide these kinds of analyses.

Automated tools have the desired characteristic of speed and they can be regularly integrated into a process - for example the tool can be run on every check-in as a part of standard testing suite. While automation is a desirable characteristic of a solution, automated tools have their limitations and undesirable characteristics. They can be expensive, require a considerable amount of tuning, and may require a powerful server just to run the tool. They cannot find certain classes of implementation vulnerabilities, such as, exception handling routines. Design vulnerabilities are outside their scope and they must be updated by the vendor to include new vulnerability classes.

A key limitation is that they produce few immediately actionable results. For example, consider cross-site scripting where a malicious script is first stored in a database and later displayed. A static analysis tool cannot track the path of the script into the database and, therefore, must treat all database input as potentially malicious. This can cause millions of false results. If it doesn't make this assumption then it will miss second-level cross-site scripting issues. Additionally, the model for these tools is to find a pattern and then offer a standard suggestion for the issues. As such, the tool has no context of the problem and cannot determine if the situation may require a fix that goes beyond the standard solution.

Black Box Methods

Black box methods include automated vulnerability scanning and penetration testing. No information about the applications or infrastructure is supplied to the testing expert – traditionally targeted at seeing the target application from a ”hackers perspective”.

Automated Vulnerability Scanning

Automated vulnerability scanning tools are designed to find implementation vulnerabilities. Application scanning tool examples include: HP Web Inspect, NTO Spider, IBM Appscan, Acunetix, etc. Automated scanning tools for operating system and service vulnerabilities include: McAfee Foundstone Enterprise, ISS, eEye, NeXpose, Qualys, etc.

The tools are very effective at identifying many implementation and operational vulnerabilities. However, there are a high number of false positives, they often overlook design vulnerabilities, they can impact network resources and can potentially disrupt IDS systems.

Penetration Testing

With penetration testing, an application is considered a black box in which data goes in and results are delivered as output. A list of ‘test cases’ is exercised in the application’s native environment. An example test case might include a check for an SQL injection, or testing using a browser plus man-in-the-middle (MITM) proxy.

The benefit of penetration testing is that it tests the actual implementation and can quickly exploit issues with the application. It can find Implementation, Design and Operational vulnerabilities and can ultimately have very little impact on organization resources. However, it can be very time consuming. For example, a web site with 50 pages may require 50 test cases per page equating to 2500 test cases. It can impact production systems (if no test environment is available) and is dependent on the availability of the applications. While it exposes what a hacker would see, it may not find all implementation vulnerabilities.

Grey Box Methods

Grey box methodologies blend the best of both black and white box testing. They target privileged analysis of the runtime environment and presentation tier interface, as well as the source code. Runtime black box testing can quickly identify design vulnerabilities, such as logic flaws and analysis of the code base allows for rapid identification of Implementation vulnerabilities.

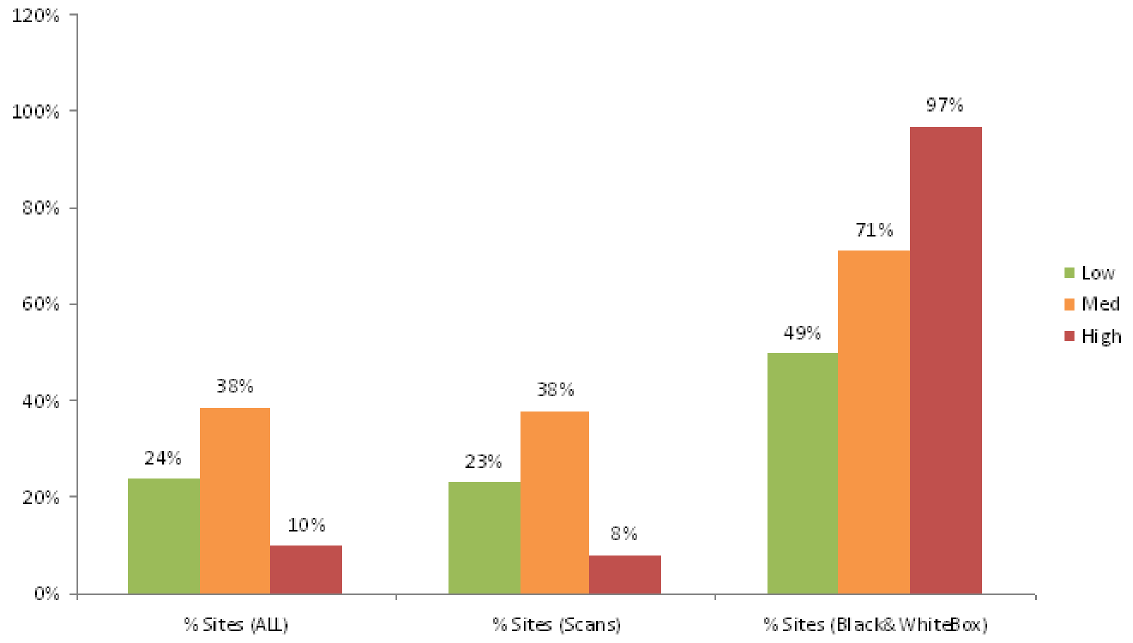
A grey box approach allows for a more comprehensive analysis of a target application. Efficiencies are gained by coming at the problems from both sides. The availability of information allows for a higher accuracy of detection and more granular recommendations. It allows for a more complete understanding of proof of impact.

It, too, can be very time consuming compared to point solution analyses. Time from the product development team may be required and it requires a high level of expertise and capabilities to execute properly. Repeatability can be challenging and costly and it can also require access to sensitive corporate IP by the expert team engaged to perform the analysis.

Statistical Analysis of Testing Methods

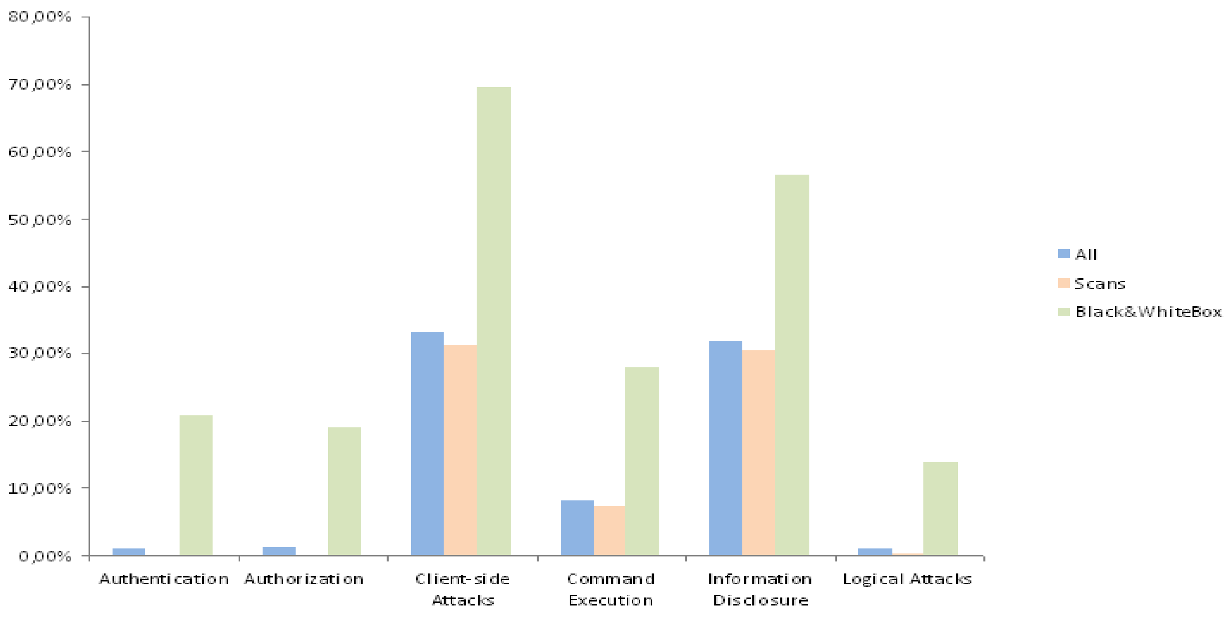
The WASC Statistics Project provided a consolidated analysis of common vulnerabilities across a wide variety of web applications. This analysis provided a view of over 32,000 sites and 70,000 vulnerabilities. Data sources came from automated vulnerability scanning tests and grey box approaches that coupled automated scanning with manual approaches.

The results, Figure 1, show that the probability of detecting high risk vulnerabilities using grey box methods is 12.5 times higher than using only an automated scanning approach. Approximately 7% of the analyzed sites can be compromised automatically (using black box methods alone) and detection of high severity vulnerabilities reaches to 96.85% when using grey box methods.



It should be noted, however, that the automated scanning tests were conducted without customizing the settings for the tool. With customized settings created by an expert, these tools would likely show improved effectiveness. In addition, not every site tested uses interactive elements which are generally a source of critical vulnerabilities.

The specific methodology's ability to identify vulnerabilities within certain classes is shown in Figure 2. It also shows the prevalence of certain vulnerability classes within the application's testing overall. Figure 3 identifies the vulnerabilities by risk level and application.



`

Figure 2: Vulnerabilities by Classes

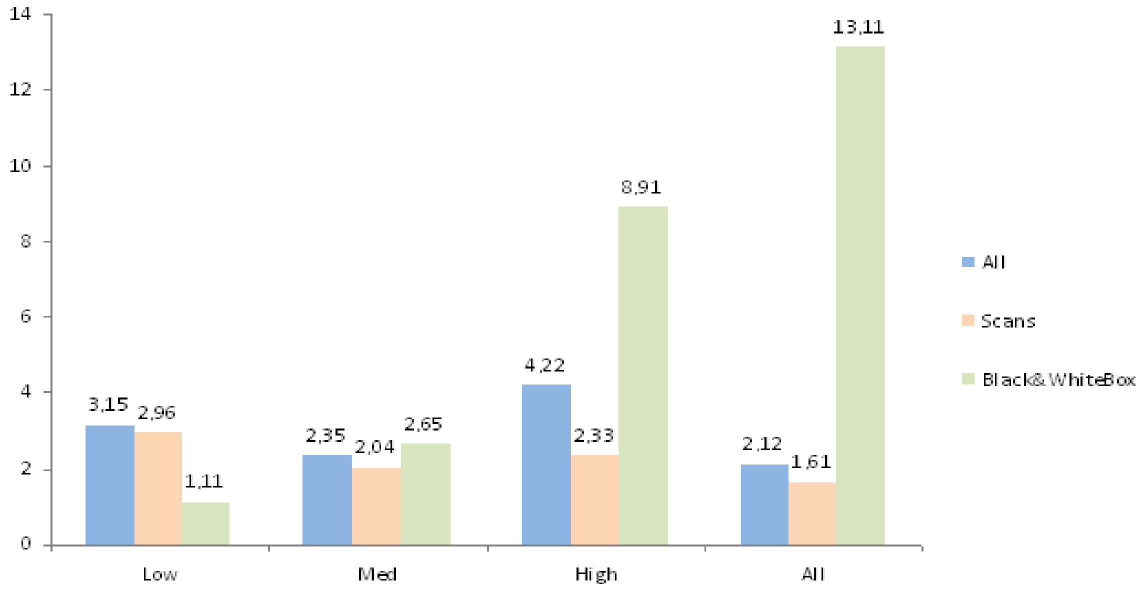


Figure 3: Vulnerabilities by Risk

Integrating Application Assessment into the Organizational Process

Applications are composed of third party developed code, legacy code and current development. For third party and legacy code the only choices are to execute the most comprehensive methodology possible in context of the application risk profile. Ongoing development within the organization, however, allows for controlled management of the software development life cycle. As the life cycle progresses, the costs to remediate only increases, thus integrating the security controls into the development cycle creates the most effective approach to containing vulnerabilities.

If we consider the standard software development life cycle, security concerns have a place at every stage.

Analyze User Requirements: Security requirements should include policies and security standards, security requirements based on risk profile and analyses, and the creation of use cases that relate to the relevant security standards.

Design: Security concerns should consider threat modeling and associated abuse case development.

Coding: Developers should be educated to implement the security controls and execute automated static code analyses.

Document and Test: Testing suites should include automated static code analyses. Manual code reviews and automated vulnerability scanning should be performed, as well as other grey box methods that apply.

Operation and Maintenance: Ongoing security testing should include automated vulnerability testing with current versions of automated tools as well as other grey box methods that apply and are deemed important as a function of the risk profile.

Conclusions

There are no single solutions that can comprehensively identify all application vulnerabilities. Instead, the organization must adopt a blend of the relevant methodologies. Relevancy is based on the application risk profile, criticality, timeframes for addressing the issues, availability of resources and budget.

A summary of vulnerability class coverage by the method type is displayed in Table 1. Table 2 provides an overview of the strengths and weaknesses of all the methods. Table 3 indicates the appropriate position for the method in the SDLC.

Testing Approach	Method	Design	Implementation	Operational and Platform
White Box	Threat Modeling	yes	no	no
White Box	Manual Code Review	yes	yes	no
White Box	Automated Static Analysis	no	partial	no
Black Box	Automated Vulnerability Scanning	no	yes	yes
Black Box	Penetration Testing	partial	partial	yes
Grey Box	A blend of tools	yes	yes	yes

Table 1: Vulnerability Class Coverage by Method

Testing Approach	Method	Strengths	Weaknesses
White Box	Threat Modeling	Can be implemented early in the design	High personnel impact
White Box	Manual Code Review	Better analysis "coverage" Detailed remediation information Some methods can quickly identify LHF issues Able to provide deeper analysis to show impact	Comprehensive approach can be time consuming It is often difficult to detect design issues due to complexity Can require high personnel involvement
White Box	Automated Static Analysis	Tools can be fast to run Can be run whenever desired Thorough for the patterns of the issues they can find Often faster and cheaper than a manual review	Few actionable results Frequently not 100% current Cannot find certain classes of Implementation vulnerabilities Boilerplate suggestions on remediation
Black Box	Automated Vulnerability Scanning	Quickly identifies vulnerabilities	High number of false positives Noisy traffic for IDS systems Can impact resources
Black Box	Penetration Testing	Tests actual implementation Quickly finds issues from an attackers perspective Low personnel impact	Can be slow Testing can impact production Ability to test dependent on availability of systems
Grey Box	A blend of tools	Efficiency Accuracy Comprehensive analysis and identification of vulnerability classes	Cost and duration

Table 2: Method Strengths and Weaknesses

Testing Approach	Method	Position in SDLC
White Box	Threat Modeling	Requirements Analysis and Design
White Box	Manual Code Review	Coding
White Box	Automated Static Analysis	Coding
Black Box	Automated Vulnerability Scanning	Testing and/or post-production deployment
Black Box	Penetration Testing	Testing and/or post-production deployment
Grey Box	A blend of tools	Where appropriate

Table 3: Method Position in the SDLC

Every organization has different concerns and, consequently, will likely take advantage of different solutions. Solution design must consider the possible attack vectors and the solution's ability to find the types of flaws that are specific to the organization's profile. Often, as the process continues, a good analysis will find much more than was originally suspected.