



Veracode “2010 CWE/SANS Top 25 Most Dangerous Software Errors” Detection

Introduction

Veracode SecurityReview® uses a combination of automated static analysis, automated dynamic analysis, and manual penetration testing to assess web applications against the vulnerabilities listed in the [2010 CWE/SANS Top 25 Most Dangerous Software Errors](#). The following is a description of the combined analysis and testing process for detecting this list of the most widespread and critical software errors.

1. [CWE-79](#) Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Tests used: Automated Static, Automated Dynamic, Manual

Static analysis is used for XSS testing. The analyzer looks at the data flow model and finds tainted data that is output back to the web browser through the web applications response. If output encoding is not performed on the data, XSS is detected. Automated static accuracy for XSS is high.

Dynamic analysis detects XSS vulnerabilities using a similar methodology to other injection attacks. To prevent false positives, each HTTP response is evaluated using a Javascript engine to verify code execution.

Manual penetration will attempt to find XSS, particularly in cases where other methods did not detect any, but this is not a major focus of manual testing since the automated methods are so effective.

2. [CWE-89](#) Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Tests used: Automated Static, Automated Dynamic, Manual

Veracode SecurityReview® static analysis creates a detailed data flow model that covers 100% of the application code. For each supported platform, such as Java, .NET, PHP, Cold Fusion, etc, the analyzer has a list of the sources of tainted data and a list of the sinks where tainted data will cause an injection vulnerability. Example sources are OS APIs for network and file reads and example sinks for SQL Injection are database queries.

Bytecode and interpreted languages have data flow graphs that are relatively easy for a static analyzer to model. The 100% coverage of sources and sinks, combined with an accurate data flow modeler, creates a high degree of accuracy for the automated static detection of SQL Injection flaws.

Dynamic analysis detects SQL Injection vulnerabilities by crawling the live web site, injecting attack strings into forms, headers, cookies, etc., and then using various heuristics to determine which requests caused an exploitable condition.

Manual SQL Injection attacks are also attempted in order to add depth in areas where the automated dynamic scanner may not have been able to reach or exploit fully.

3. [CWE-120](#) Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Tests used: Automated Static

Buffer overflows are found through static analysis by mapping all the data flows into a buffer copy function that does not do proper bounds checking. The possible ranges of lengths of source buffers, destination buffers, and the ranges of size values are tracked alongside their data flow. If a condition is found where a source may be larger than the destination and no range validation is detected then the code is reported as having a buffer overflow at that point.

4. [CWE-352](#) Cross-Site Request Forgery (CSRF)

Tests used: Manual

It is not possible for automation to reliably detect CSRF vulnerabilities without generating an unacceptable rate of false positives. It is also difficult to determine in an automated fashion which functionality within the application need to be protected from such attacks in the first place. Manual penetration testing is required to detect CSRF accurately. Testers look for standard CSRF defenses such as the use of unique tokens accompanying requests for sensitive application functions.

5. [CWE-285](#) Improper Access Control (Authorization)

Tests used: Manual

It is not possible for automated testing to understand the authorization model of an application or determine whether the values provided by an end user constitute access to a restricted object or transaction. This design knowledge requires the insight of a manual penetration tester. During the engagement, Veracode will attempt a variety of authorization attacks, including attempts to access resources and data outside of the current user's role and/or community and attempts to discover and access unprotected pages and resources. Veracode's testing also includes evaluations of the application to determine whether the users could access any functionality or resources not normally permitted.

6. [CWE-807](#) Reliance on Untrusted Inputs in a Security Decision

Tests used: Static Automated, Manual

Security decisions are almost always codified within application business logic so the authorization model must be understood in order to determine if a security decision is in fact being made. Manual testing identifies these security decisions and attempts to undermine their integrity by manipulating inputs the application implicitly trusts.

Static analysis can detect if the results of DNS lookups are being used in a security decision.

7. [CWE-22](#) Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Tests used: Automated Static

Path traversal can be thought of as an injection style attack where untrusted input reached a system call for a file operation. The analyzer has a list of the sources of tainted data and a list of the sinks where tainted data will cause an injection vulnerability. Example sources are OS APIs for network and file reads and example sinks for path traversal are all of the system calls that take file and directory names as inputs.

8. [CWE-434](#) Unrestricted Upload of File with Dangerous Type

Tests used: Manual

A manual penetration tester will attempt to upload files containing specific data or an extension that may trigger application server execution of the provided content.

9. [CWE-78](#) Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

Tests used: Automated Static, Automated Dynamic, Manual

Veracode SecurityReview® static analysis creates a detailed data flow model that covers 100% of the application code. For each supported platform, such as Java, .NET, PHP, Cold Fusion, etc, the analyzer has a list of the sources of tainted data and a list of the sinks where tainted data will cause an injection vulnerability. Example sources are OS APIs for network and file reads and example sinks for Command Injection are exec and command APIs.

Bytecode and interpreted languages have data flow graphs that are relatively easy for a static analyzer to model. The 100% coverage of sources and sinks, combined with an accurate data flow modeler, creates a high degree of accuracy for the automated static detection of OS Command Injection flaws.

Dynamic analysis uncovers OS Command Injection vulnerabilities by crawling the live web site, injecting attack strings into forms, headers, cookies, etc. and then using various heuristics to determine which requests caused an exploitable condition.

Manual OS Command Injection attacks are also attempted in order to add depth in areas where the automated dynamic scanner may not have been able to reach or exploit fully.

10. [CWE-311](#) Missing Encryption of Sensitive Data

Tests used: Automated Static

Static analysis has the capability of detecting known crypto API procedures. Sensitive data, such as passwords, are checked to see if they are encrypted before being stored on disk.

11. [CWE-798](#) Use of Hard-coded Credentials

Tests used: Automated Static

Static analysis uses dataflow analysis to determine where the values that enter security related functions are derived from. If credentials such as passwords and keys are static values within the software, the static analyzer can determine that they were hard-coded.

12. [CWE-805](#) Buffer Access with Incorrect Length Value

Tests used: Automated Static

Buffer access with incorrect length value flaws are found through static analysis by mapping all the data flows into a buffer access functions. The possible ranges of lengths of source buffers, destination buffers, and the ranges of length values are tracked alongside their data flow. If a condition is found where an access length may be larger (or smaller than zero) than the buffer length then the code is reported as having a buffer access error at that point.

13. [CWE-98](#) Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')

Tests used: Automated Static

Static analysis uses the data flow graph of the PHP application to inspect include statements within the PHP files for the use of include filenames that have sources controlled by user inputs.

14. [CWE-129](#) Improper Validation of Array Index

Tests used: Automated Static

Improper validation of array index errors are found through static analysis by inspecting all array index values and checking to make sure the potential range of the value is constrained to fall within zero and the possible length of the array.

15. [CWE-754](#) Improper Check for Unusual or Exceptional Conditions

Tests used: Manual

Manual penetration testing attempts to trigger unusual conditions that may have been unanticipated by the application's designers and developers. These conditions are reported when they impact a security decision or might otherwise affect the confidentiality, integrity, or availability of the application or protected data.

16. [CWE-209](#) Information Exposure Through an Error Message

Tests used: Automated Static, Automated Dynamic, Manual

Static analysis can inspect for error messages that are generated through error and exception conditions. The data flow of these messages is followed and if these messages are output to the user then an information exposure error is reported.

Dynamic analysis views the output returned to the user. The keywords that are often in error messages are tested for and, if found, an information exposure error is reported.

While manual testers are performing their testing they will report if they view an error message that contains information that contains sensitive information or is otherwise useful to an attacker in identifying or exploiting a vulnerable condition.

17. [CWE-190](#) Integer Overflow or Wraparound

Tests used: Automated Static

Static analysis knows the types of variables and can track the potential range of the variables. For a particular variable type, operations are tested to make sure that the range of the variable doesn't overflow or underflow the maximum or minimum value it can hold. If a variable is an unsigned integer, all operations, such as shift, add, multiply, etc., that could set the value below zero or above the maximum unsigned integer value are reported as errors.

18. [CWE-131](#) Incorrect Calculation of Buffer Size

Tests used: Automated Static

Static analysis knows the sizes of source and target buffers. Often a target buffer is allocated based on a size calculation equal to the count of objects multiplied by the object size. A common instance of this is the usage of double byte character source buffers and a target buffer calculated as for single byte

characters. If the target buffer size is wrong and it is based on an allocation calculation, an error is reported.

19. [CWE-306](#) Missing Authentication for Critical Function

Tests used: Manual

Automation has a limited ability to understand the authentication state of an application and what the authorization requirements of the software are. Manual penetration testing must be conducted to classify which of the exposed functions would be deemed critical and to determine if sufficient authentication requirements are present and enforced properly.

20. [CWE-494](#) Download of Code Without Integrity Check

Tests used: Manual

Manual penetration testing is used to differentiate between trusted and untrusted external sources of executable code that are retrieved and interpreted by an application component. A security flaw is reported if an untrusted code source is utilized without an appropriate integrity check, such as validating the identity of the remote host or validating the signature of the retrieved code block.

21. [CWE-732](#) Incorrect Permission Assignment for Critical Resource

Tests used: Manual

Automation has a limited ability to understand what the correct permissions for resources should be. Manual penetration testing is conducted to sufficiently verify that this category isn't present in the application. Manual authorization-related attacks attempt to exploit or subvert an application's authentication infrastructure. During the penetration test, Veracode attempts a number of authorization attacks.

22. [CWE-770](#) Allocation of Resources Without Limits or Throttling

Tests used: Manual

Manual penetration testing is conducted to sufficiently verify that this category isn't present in the application. Manual attacks attempt to perform application-layer denial of service attacks which, if successful, demonstrate a lack of internal limits and throttling.

23. [CWE-601](#) URL Redirection to Untrusted Site ('Open Redirect')

Tests used: Automated Static

Automated static testing uses the same mechanisms to detect unvalidated redirects as it uses for injection and XSS attacks. If tainted data is used by the application to construct the target of an HTTP redirect, it is marked as a flaw.

24. [CWE-327](#) Use of a Broken or Risky Cryptographic Algorithm

Tests used: Automated Static

Automated static testing can inspect for common OS level and 3rd party cryptographic APIs. If the API is known to be broken or risky, or if the API is used incorrectly, an error is reported.

25. [CWE-362](#) Race Condition

Tests used: Automated Static

Automated static testing can identify certain types of race conditions, such as signal handler race conditions and time-of-check time-of-use race conditions, by examining the generated data flow model.