VERACODE

Veracode Detailed Report

Application Security Report As of 18 Oct 2024

Prepared for: SEGTIUM S.A (MSP)
Prepared on: October 19, 2024
Application: AWSVeraJavaVxV

Industry: Other

Business Criticality: BC5 (Very High)

Required Analysis: Manual Penetration Test, Static

Type(s) of Analysis Conducted: Static

Scope of Static Scan: 1 of 2 Modules Analyzed

Inside This Report

Executive Summary	1
Summary of Flaws by Severity	1
Action Items	1
Flaw Types by Category	3
Policy Summary	4
Findings & Recommendations	5
Methodology	

While every precaution has been taken in the preparation of this document, Veracode, Inc. assumes no responsibility for errors, omissions, or for damages resulting from the use of the information herein. The Veracode platform uses static and/or dynamic analysis techniques to discover potentially exploitable flaws. Due to the nature of software security testing, the lack of discoverable flaws does not mean the software is 100% secure.

© 2024 Veracode, Inc.

SEGTIUM S.A (MSP) and Veracode Confidential

Veracode Level: VL1
Rated: Oct 18, 2024



Veracode Detailed Report Application Security Report

As of 18 Oct 2024

AWSVeraJavaVxV Business Criticality: Very High

VL5 Published Rating: F

Scans Included in Report

Application:

Target Level:

Static Scan	Dynamic Scan	Manual Penetration Test
2.1 Score: 59 Completed: 10/18/24	Not Included in Report	Not Included in Report

Executive Summary

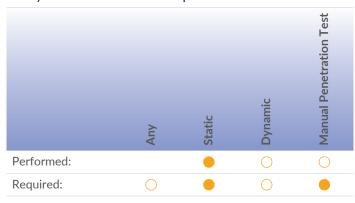
This report contains a summary of the security flaws identified in the application using manual penetration testing, automated static and/or automated dynamic security analysis techniques. This is useful for understanding the overall security quality of an individual application or for comparisons between applications.

Application Business Criticality: BC5 (Very High)

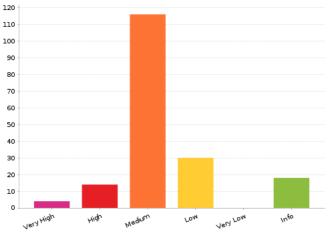
Impacts:Operational Risk (High), Financial Loss (High)

An application's business criticality is determined by business risk factors such as: reputation damage, financial loss, operational risk, sensitive information disclosure, personal safety, and legal violations. The Veracode Level and required assessment techniques are selected based on the policy assigned to the application.

Analyses Performed vs. Required



Summary of Flaws Found by Severity



Action Items:

Veracode recommends the following approaches ranging from the most basic to the strong security measures that a vendor can undertake to increase the overall security level of the application.

Required Analysis

- Your policy requires Manual Penetration Test but it has not been performed. Please submit your application for Manual Penetration Test and remediate the required detected flaws to conform to your assigned policy.
- Your policy requires periodic Static Scan. Your next analysis must be completed by 1/18/25. Please submit your application for Static Scan by the deadline and remediate the required detected flaws to conform to your assigned policy.

Flaws To Fix By Expires Date



A grace period is specified for any flaw that violates the rules contained in your policy. These include CWE, Rollup Category, Issue Severity, Industry Standards as well as any flaws that prevent an application from achieving a minimum Veracode Level and/or score. To maintain policy compliance you must fix these flaws and resubmit your application for scanning before the grace period expires. The detailed flaw listing will badge the flaws that must be fixed and show the fix by date as well.

The grace period has expired [10/18/24] for 134 flaws that were found in your Static Scan.

Flaws To Fix For Minimum Score

- A rule in your policy requires a minimum score of 90. Fix 4 Very High flaws, 14 High flaws and 94 Medium flaws to reach a score of 90.
- Your Static Scan was due on 10/18/24 for follow-up analysis to satisfy the grace period on your minimum score rule and your application is no longer compliant with your policy. Submit application for follow-up Static Scan once flaws have been remediated in order to regain compliance with your policy.

Flaw Severities

→ Medium severity flaws and above must be fixed for policy compliance.

Longer Timeframe (6 - 12 months)

Certify that software engineers have been trained on application security principles and practices.



Scope of Static Scan

The following modules were included in the static scan because the scan submitter selected them as entry points, which are modules that accept external data.

Engine Version: 20240927195913

The following modules were included in the application scan:

Module Name	Compiler	Operating Environment	Engine Version
verademo.war	JAVAC_8	Java J2SE 8	202409271 95913

The following modules were not selected for a full scan. Code paths in these modules that are not called from a scanned module are not included in this report.

Module Name	Compiler	Operating Environment	Engine Version
JS files within verademo.war	JAVASCRIPT_5_1	JavaScript	202409271 95913

Flaw Types by Severity and Category

Static Scan Security Quality Score = 59					
Very High	4				
Command or Argument Injection	4				
High	14				
SQL Injection	14				
Medium	116				
CRLF Injection	33				
Credentials Management	30				
Cross-Site Scripting (XSS)	36				
Cryptographic Issues	9				
Directory Traversal	4				
Encapsulation	1				
Insufficient Input Validation	3				
Low	30				
API Abuse	17				
Cryptographic Issues	3				
Information Leakage	10				
Very Low	0				
Informational	18				
Code Quality	18				
Total	182				



Policy Evaluation

Policy Name: Veracode Recommended Very High

Revision: 1

Policy Status: Did Not Pass

Description: Veracode provides default policies to make it easier for organizations to begin measuring their applications against policies. Veracode Recommended Policies are available for customers as an option when they are ready to move beyond the initial bar set by the Veracode Transitional Policies. The policies are based on the Veracode Level definitions.

Rules

Rule type	Requirement	Findings	Status
Minimum Veracode Level	VL5	VL1	Did not pass
(VL5) Min Analysis Score	90	59	Did not pass
(VL5) Max Severity	Medium	Flaws found: 134	Did not pass

Scan Requirements

Scan Type	Frequency	Last performed	Status
Static	Quarterly	10/18/24	Did not pass
Manual	Annually	Never	Need to complete (before 10/18/25)

Remediation

Flaw Severity	Grace Period	Flaws Exceeding	Status
Very High	0 days	4	Did not pass
High	0 days	14	Did not pass
Medium	0 days	116	Did not pass
Low	0 days	0	Passed
Very Low	0 days	0	Passed
Informational	0 days	0	Passed

Туре	Grace Period	Exceeding	Status
Min Analysis Score	0 days	1	Did not pass



Findings & Recommendations

Detailed Flaws by Severity

Very High (4 flaws)





Command or Argument Injection(4 flaws)

Description

Command or argument injection vulnerabilities occur when data enters an application from an untrusted source and is used to dynamically construct and execute a command. In the case of OS command injection, an attacker may be able to either alter the command executed by the application or append additional commands. In the case of argument injection, the attacker may influence the behavior of the program in other ways, for example, changing the destination of an outbound network request or injecting additional commands into an argument or parameter. The command is typically executed with the privileges of the executing process and gives an attacker a privilege or capability that he would not otherwise have.

Recommendations

Careful handling of all untrusted data is critical in preventing injection attacks. Using one or more of the following techniques provides defense-in-depth and minimizes the likelihood of an vulnerability.

- * If possible, use library calls rather than external processes to recreate the desired functionality.
- * Validate user-supplied input using positive filters (white lists) to ensure that it conforms to the expected format, using centralized data validation routines when possible.
- * Select safe API routines. Some APIs that execute system commands take an array of strings as input rather than a single string, which protects against some forms of command injection by ensuring that a user-supplied argument cannot be interpreted as part of the command.

Associated Flaws by CWE ID:



Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') (CWE ID 78)(4 flaws)

Description

This call contains a command injection flaw. The argument to the function is constructed using untrusted input. If an attacker is allowed to specify all or part of the command, it may be possible to execute commands on the server with the privileges of the executing process. The level of exposure depends on the effectiveness of input validation routines, if any.

Effort to Fix: 3 - Complex implementation error. Fix is approx. 51-500 lines of code. Up to 5 days to fix.

Recommendations

Validate all untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible. When using black lists, be sure that the sanitizing routine performs a sufficient number of iterations to remove all instances of disallowed characters. Most APIs that execute system commands also have a "safe" version of the method that takes an array of strings as input rather than a single string, which protects against some forms of command injection.

Instances found via Static Scan



	Flaw Id	Module #	Class #	Module	Location	Fix By
8	1	15	-	verademo.war	com//ToolsController.java 56	10/18/24
8	2	15	-	verademo.war	com//ToolsController.java 59	10/18/24
8	3	15	-	verademo.war	com//ToolsController.java 91	10/18/24
×	4	15	-	verademo.war	com//ToolsController.java 94	10/18/24

High (14 flaws)





SQL Injection(14 flaws)

Description

SQL injection vulnerabilities occur when data enters an application from an untrusted source and is used to dynamically construct a SQL query. This allows an attacker to manipulate database queries in order to access, modify, or delete arbitrary data. Depending on the platform, database type, and configuration, it may also be possible to execute administrative operations on the database, access the filesystem, or execute arbitrary system commands. SQL injection attacks can also be used to subvert authentication and authorization schemes, which would enable an attacker to gain privileged access to restricted portions of the application.

Recommendations

Several techniques can be used to prevent SQL injection attacks. These techniques complement each other and address security at different points in the application. Using multiple techniques provides defense-in-depth and minimizes the likelihood of a SQL injection vulnerability.

- * Use parameterized prepared statements rather than dynamically constructing SQL queries. This will prevent the database from interpreting the contents of bind variables as part of the query and is the most effective defense against SQL injection.
- * Validate user-supplied input using positive filters (white lists) to ensure that it conforms to the expected format, using centralized data validation routines when possible.
- * Normalize all user-supplied data before applying filters or regular expressions, or submitting the data to a database. This means that all URL-encoded (%xx), HTML-encoded (&#xx;), or other encoding schemes should be reduced to the internal character representation expected by the application. This prevents attackers from using alternate encoding schemes to bypass filters.
- * When using database abstraction libraries such as Hibernate, do not assume that all methods exposed by the API will automatically prevent SQL injection attacks. Most libraries contain methods that pass arbitrary queries to the database in an unsafe manner.

Associated Flaws by CWE ID:

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (CWE ID 89)(14 flaws)

Description

This database query contains a SQL injection flaw. The function call constructs a dynamic SQL query using a variable derived from untrusted input. An attacker could exploit this flaw to execute arbitrary SQL queries against the database.

Effort to Fix: 3 - Complex implementation error. Fix is approx. 51-500 lines of code. Up to 5 days to fix.



Recommendations

Avoid dynamically constructing SQL queries. Instead, use parameterized prepared statements to prevent the database from interpreting the contents of bind variables as part of the query. Always validate untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
B	11	3	-	verademo.war	com//BlabController.java 490	10/18/24
×	17	6	-	verademo.war	com//IgnoreCommand.java 40	10/18/24
×	18	6	-	verademo.war	com//IgnoreCommand.java 47	10/18/24
×	15	7	-	verademo.war	com//ListenCommand.java 40	10/18/24
×	16	7	-	verademo.war	com//ListenCommand.java 47	10/18/24
×	12	12	-	verademo.war	/RemoveAccountCommand.java 40	10/18/24
×	13	12	-	verademo.war	/RemoveAccountCommand.java 47	10/18/24
×	14	12	-	verademo.war	/RemoveAccountCommand.java 51	10/18/24
×	5	17	-	verademo.war	com//UserController.java 166	10/18/24
×	6	17	-	verademo.war	com//UserController.java 251	10/18/24
×	7	17	-	verademo.war	com//UserController.java 316	10/18/24
×	8	17	-	verademo.war	com//UserController.java 384	10/18/24
B	9	17	-	verademo.war	com//UserController.java 495	10/18/24
×	10	17	-	verademo.war	com//UserController.java 506	10/18/24

Medium (116 flaws)





CRLF Injection(33 flaws)

Description

The acronym CRLF stands for "Carriage Return, Line Feed" and refers to the sequence of characters used to denote the end of a line of text. CRLF injection vulnerabilities occur when data enters an application from an untrusted source and is not properly validated before being used. For example, if an attacker is able to inject a CRLF into a log file, he could append falsified log entries, thereby misleading administrators or cover traces of the attack. If an attacker is able to inject CRLFs into an HTTP response header, he can use this ability to carry out other attacks such as cache poisoning. CRLF vulnerabilities primarily affect data integrity.

Recommendations

Apply robust input filtering for all user-supplied data, using centralized data validation routines when possible. Use output filters to sanitize all output derived from user-supplied input, replacing non-alphanumeric characters with their HTML entity equivalents.



Associated Flaws by CWE ID:



Improper Neutralization of CRLF Sequences ('CRLF Injection') (CWE ID 93)(1 flaw)

Description

A function call contains a CRLF Injection flaw. Writing untrusted input to an interface or external application that treats the CRLF (carriage return line feed) sequence as a delimiter to separate lines or records can result in that data being misinterpreted. FTP and SMTP are examples of protocols that treat CRLF as a delimiter when parsing commands.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Sanitize CRLF sequences from untrusted input when the data is being passed to an entity that may incorrectly interpret it.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
×	35	17	-	verademo.war	com//UserController.java 433	10/18/24



Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Request/Response Splitting') (CWE ID 113)(4 flaws)

Description

A function call contains an HTTP response splitting flaw. Writing untrusted input into an HTTP header allows an attacker to manipulate the HTTP response rendered by the browser, leading to cache poisoning and cross-site scripting attacks.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Remove unexpected carriage returns and line feeds from untrusted data used to construct an HTTP response. Always validate untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
8	26	17	-	verademo.war	com//UserController.java 173	10/18/24
8	38	17	-	verademo.war	com//UserController.java 631	10/18/24
*	44	17	-	verademo.war	com//UserController.java 711	10/18/24
*	45	17	-	verademo.war	com//UserController.java 713	10/18/24





Improper Output Neutralization for Logs (CWE ID 117)(28 flaws)

Description

A function call could result in a log forging attack. Writing untrusted data into a log file allows an attacker to forge log entries or inject malicious content into log files. Corrupted log files can be used to cover an attacker's tracks or as a delivery mechanism for an attack on a log viewing or processing utility. For example, if a web administrator uses a browser-based utility to review logs, a cross-site scripting attack might be possible.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Avoid directly embedding user input in log files when possible. Sanitize untrusted data used to construct log entries by using a safe logging mechanism such as the OWASP ESAPI Logger, which will automatically remove unexpected carriage returns and line feeds and can be configured to use HTML entity encoding for non-alphanumeric data. Only write custom blacklisting code when absolutely necessary. Always validate untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
*	91	3	-	verademo.war	com//BlabController.java 486	10/18/24
8	92	3	-	verademo.war	com//BlabController.java 558	10/18/24
8	93	3	-	verademo.war	com//BlabController.java 559	10/18/24
8	100	6	-	verademo.war	com//IgnoreCommand.java 39	10/18/24
8	101	6	-	verademo.war	com//IgnoreCommand.java 46	10/18/24
8	98	7	-	verademo.war	com//ListenCommand.java 39	10/18/24
*	99	7	-	verademo.war	com//ListenCommand.java 46	10/18/24
*	95	12	-	verademo.war	/RemoveAccountCommand.java 39	10/18/24
*	96	12	-	verademo.war	/RemoveAccountCommand.java 46	10/18/24
*	97	12	-	verademo.war	/RemoveAccountCommand.java 50	10/18/24
*	78	13	-	verademo.war	com//ResetController.java 109	10/18/24
*	80	13	-	verademo.war	com//ResetController.java 132	10/18/24
*	83	13	-	verademo.war	com//ResetController.java 159	10/18/24
8	88	13	-	verademo.war	com//ResetController.java 193	10/18/24
8	89	13	-	verademo.war	com//ResetController.java 194	10/18/24
*	25	17	-	verademo.war	com//UserController.java 114	10/18/24
8	27	17	-	verademo.war	com//UserController.java 229	10/18/24



	Flave Id	Module #	Class #	Module	Lagation	Eiv Dv
	Flaw Id	Module #	Class #	Module	Location	Fix By
×	29	17	-	verademo.war	com//UserController.java 237	10/18/24
8	30	17	-	verademo.war	com//UserController.java 249	10/18/24
8	31	17	-	verademo.war	com//UserController.java 255	10/18/24
8	34	17	-	verademo.war	com//UserController.java 385	10/18/24
8	36	17	-	verademo.war	com//UserController.java 493	10/18/24
8	37	17	-	verademo.war	com//UserController.java 504	10/18/24
8	39	17	-	verademo.war	com//UserController.java 658	10/18/24
8	41	17	-	verademo.war	com//UserController.java 694	10/18/24
8	43	17	-	verademo.war	com//UserController.java 708	10/18/24
×	46	17	-	verademo.war	com//UserController.java 803	10/18/24
×	47	17	-	verademo.war	com//UserController.java 859	10/18/24

Credentials Management(30 flaws)

Description

Improper management of credentials, such as usernames and passwords, may compromise system security. In particular, storing passwords in plaintext or hard-coding passwords directly into application code are design issues that cannot be easily remedied. Not only does embedding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If a hard-coded password is compromised in a commercial product, all deployed instances may be vulnerable to attack, putting customers at risk.

One variation on hard-coding plaintext passwords is to hard-code a constant string which is the result of a cryptographic one-way hash. For example, instead of storing the word "secret," the application stores an MD5 hash of the word. This is a common mechanism for obscuring hard-coded passwords from casual viewing but does not significantly reduce risk. However, using cryptographic hashes for data stored outside the application code can be an effective practice.

Recommendations

Avoid storing passwords in easily accessible locations, and never store any type of sensitive data in plaintext. Avoid using hard-coded usernames, passwords, or hash constants whenever possible, particularly in relation to security-critical components. Store passwords out-of-band from the application code. Follow best practices for protecting credentials stored in alternate locations such as configuration or properties files.

Associated Flaws by CWE ID:





Use of Hard-coded Credentials (CWE ID 798)(28 flaws)

Description

The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data.

Effort to Fix: 4 - Simple design error. Requires redesign and up to 5 days to fix.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
*	64	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	65	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	67	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	50	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	51	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	53	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	55	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	56	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	57	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	59	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	58	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	60	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	61	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	62	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	66	13	-	verademo.war	com//ResetController.java 38	10/18/24
*	52	13	-	verademo.war	com//ResetController.java 38	10/18/24
×	54	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	72	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	63	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	68	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	69	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	70	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	71	13	-	verademo.war	com//ResetController.java 38	10/18/24
8	73	13	-	verademo.war	com//ResetController.java 38	10/18/24



	Flaw Id	Module #	Class #	Module	Location	Fix By
×	74	13	-	verademo.war	com//ResetController.java 38	10/18/24
×	75	13	-	verademo.war	com//ResetController.java 38	10/18/24
×	76	13	-	verademo.war	com//ResetController.java 38	10/18/24
×	77	13	-	verademo.war	com//ResetController.java 38	10/18/24

→

Use of Hard-coded Password (CWE ID 259)(2 flaws)

Description

A method uses a hard-coded password that may compromise system security in a way that cannot be easily remedied. The use of a hard-coded password significantly increases the possibility that the account being protected will be compromised. Moreover, the password cannot be changed without patching the software. If a hard-coded password is compromised in a commercial product, all deployed instances may be vulnerable to attack.

Effort to Fix: 4 - Simple design error. Requires redesign and up to 5 days to fix.

Recommendations

Store passwords out-of-band from the application code. Follow best practices for protecting credentials stored in locations such as configuration or properties files.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
B	21	4	-	verademo.war	com//utils/Constants.java 1	10/18/24
8	22	4	-	verademo.war	com//utils/Constants.java 14	10/18/24

\rightarrow

Cross-Site Scripting (XSS)(36 flaws)

Description

Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed occur whenever a web application uses untrusted data in the output it generates without validating or encoding it. XSS vulnerabilities are commonly exploited to steal or manipulate cookies, modify presentation of content, and compromise sensitive information, with new attack vectors being discovered on a regular basis. XSS is also commonly referred to as HTML injection.

XSS vulnerabilities can be either persistent or transient (often referred to as stored and reflected, respectively). In a persistent XSS vulnerability, the injected code is stored by the application, for example within a blog comment or message board. The attack occurs whenever a victim views the page containing the malicious script. In a transient XSS vulnerability, the injected code is included directly in the HTTP request. These attacks are often carried out via malicious URLs sent via email or another website and requires the victim to browse to that link. The consequence of an XSS attack to a victim is the same regardless of whether it is persistent or transient; however, persistent XSS vulnerabilities are likely to affect a greater number of victims due to its delivery mechanism.



Recommendations

Several techniques can be used to prevent XSS attacks. These techniques complement each other and address security at different points in the application. Using multiple techniques provides defense-in-depth and minimizes the likelihood of a XSS vulnerability.

- * Use output filtering to sanitize all output generated from user-supplied input, selecting the appropriate method of encoding based on the use case of the untrusted data. For example, if the data is being written to the body of an HTML page, use HTML entity encoding. However, if the data is being used to construct generated Javascript or if it is consumed by client-side methods that may interpret it as code (a common technique in Web 2.0 applications), additional restrictions may be necessary beyond simple HTML encoding.
- * Validate user-supplied input using positive filters (white lists) to ensure that it conforms to the expected format, using centralized data validation routines when possible.
- * Do not permit users to include HTML content in posts, notes, or other data that will be displayed by the application. If users are permitted to include HTML tags, then carefully limit access to specific elements or attributes, and use strict validation filters to prevent abuse.

Associated Flaws by CWE ID:

→ Im

Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS) (CWE ID 80)(36 flaws)

Description

This call contains a cross-site scripting (XSS) flaw. The application populates the HTTP response with untrusted input, allowing an attacker to embed malicious content, such as Javascript code, which will be executed in the context of the victim's browser. XSS vulnerabilities are commonly exploited to steal or manipulate cookies, modify presentation of content, and compromise confidential information, with new attack vectors being discovered on a regular basis.

Effort to Fix: 3 - Complex implementation error. Fix is approx. 51-500 lines of code. Up to 5 days to fix.

Recommendations

Use contextual escaping on all untrusted data before using it to construct any portion of an HTTP response. The escaping method should be chosen based on the specific use case of the untrusted data, otherwise it may not protect fully against the attack. For example, if the data is being written to the body of an HTML page, use HTML entity escaping; if the data is being written to an attribute, use attribute escaping; etc. When a web framework provides built-in support for automatic XSS escaping, do not disable it. Both the OWASP Java Encoder library for Java and the Microsoft AntiXSS library provide contextual escaping methods. For more details on contextual escaping, see https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html. In addition, as a best practice, always validate untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
×	129	1	-	verademo.war	/WEB-INF/views/blab.jsp 57	10/18/24
×	130	1	-	verademo.war	/WEB-INF/views/blab.jsp 61	10/18/24
×	131	1	-	verademo.war	/WEB-INF/views/blab.jsp 69	10/18/24
×	132	1	-	verademo.war	/WEB-INF/views/blab.jsp 105	10/18/24
×	133	1	-	verademo.war	/WEB-INF/views/blab.jsp 109	10/18/24



Fix By	Location	Class # Module	Module #	Flaw Id	
10/18/24	/WEB-INF/views/blab.jsp 111	- verademo.war	1	134	B
10/18/24	/WEB-INF/views/blabbers.jsp 66	- verademo.war	2	126	Ø
10/18/24	/WEB-INF/views/blabbers.jsp 101	- verademo.war	2	127	Ø
10/18/24	/WEB-INF/views/blabbers.jsp 104	- verademo.war	2	128	B
10/18/24	com//BlabController.java 204	- verademo.war	3	90	B
10/18/24	/WEB-INF/views/feed.jsp 59	- verademo.war	5	120	8
10/18/24	/WEB-INF/views/feed.jsp 70	- verademo.war	5	121	Ø
10/18/24	/WEB-INF/views/feed.jsp 94	- verademo.war	5	122	B
10/18/24	/WEB-INF/views/feed.jsp 97	- verademo.war	5	123	Ø
10/18/24	/WEB-INF/views/feed.jsp 99	- verademo.war	5	124	B
10/18/24	/WEB-INF/views/feed.jsp 142	- verademo.war	5	125	Ø
10/18/24	/WEB-INF/views/login.jsp 58	- verademo.war	8	117	B
10/18/24	/WEB-INF/views/login.jsp 77	- verademo.war	8	118	B
10/18/24	/WEB-INF/views/login.jsp 84	- verademo.war	8	119	8
10/18/24	/WEB-INF/views/profile.jsp 63	- verademo.war	9	109	Ø
10/18/24	/WEB-INF/views/profile.jsp 91	- verademo.war	9	110	Ø
10/18/24	/WEB-INF/views/profile.jsp 102	- verademo.war	9	111	Ø
10/18/24	/WEB-INF/views/profile.jsp 111	- verademo.war	9	112	Ø
10/18/24	/WEB-INF/views/profile.jsp 120	- verademo.war	9	113	B
10/18/24	/WEB-INF/views/profile.jsp 161	- verademo.war	9	114	Ø
10/18/24	/WEB-INF/views/profile.jsp 164	- verademo.war	9	115	Ø
10/18/24	/WEB-INF/views/profile.jsp 201	- verademo.war	9	116	Ø
10/18/24	//views/register-finish.jsp 60	- verademo.war	10	107	Ø
10/18/24	//views/register-finish.jsp 83	- verademo.war	10	108	B
10/18/24	/WEB-INF/views/register.jsp 60	- verademo.war	11	105	B
10/18/24	/WEB-INF/views/register.jsp 87	- verademo.war	11	106	B
10/18/24	/WEB-INF/views/tools.jsp 65	- verademo.war	14	102	B
10/18/24	/WEB-INF/views/tools.jsp 68	- verademo.war	14	103	B
10/18/24	/WEB-INF/views/tools.jsp 78	- verademo.war	14	104	B
10/18/24	com//UserController.java 256	- verademo.war	17	32	Ø



	Flaw Id	Module #	Class #	Module	Location	Fix By
8	33	17	-	verademo.war	com//UserController.java 263	10/18/24

Cryptographic Issues(9 flaws)

Description

Applications commonly use cryptography to implement authentication mechanisms and to ensure the confidentiality and integrity of sensitive data, both in transit and at rest. The proper and accurate implementation of cryptography is extremely critical to its efficacy. Configuration or coding mistakes as well as incorrect assumptions may negate a large degree of the protection it affords, leaving the crypto implementation vulnerable to attack.

Common cryptographic mistakes include, but are not limited to, selecting weak keys or weak cipher modes, unintentionally exposing sensitive cryptographic data, using predictable entropy sources, and mismanaging or hard-coding keys.

Developers often make the dangerous assumption that they can improve security by designing their own cryptographic algorithm; however, one of the basic tenets of cryptography is that any cipher whose effectiveness is reliant on the secrecy of the algorithm is fundamentally flawed.

Recommendations

Select the appropriate type of cryptography for the intended purpose. Avoid proprietary encryption algorithms as they typically rely on "security through obscurity" rather than sound mathematics. Select key sizes appropriate for the data being protected; for high assurance applications, 256-bit symmetric keys and 2048-bit asymmetric keys are sufficient. Follow best practices for key storage, and ensure that plaintext data and key material are not inadvertently exposed.

Associated Flaws by CWE ID:



Description

Standard random number generators do not provide a sufficient amount of entropy when used for security purposes. Attackers can brute force the output of pseudorandom number generators such as rand().

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

If this random number is used where security is a concern, such as generating a session identifier or cryptographic key, use a trusted cryptographic random number generator instead.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
8	79	13	-	verademo.war	com//ResetController.java 128	10/18/24
*	81	13	-	verademo.war	com//ResetController.java 153	10/18/24
×	82	13	-	verademo.war	com//ResetController.java 156	10/18/24
×	84	13	-	verademo.war	com//ResetController.java 179	10/18/24



	Flaw Id	Module #	Class #	Module	Location	Fix By
*	85	13	-	verademo.war	com//ResetController.java 183	10/18/24
*	86	13	-	verademo.war	com//ResetController.java 187	10/18/24
8	87	13	-	verademo.war	com//ResetController.java 191	10/18/24

Use of a Broken or Risky Cryptographic Algorithm (CWE ID 327)(2 flaws)

Description

The use of a broken or risky cryptographic algorithm is an unnecessary risk that may result in the disclosure of sensitive information.

Effort to Fix: 1 - Trivial implementation error. Fix is up to 5 lines of code. One hour or less to fix.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
*	20	16	-	verademo.war	com//verademo/utils/User.java 103	10/18/24
8	49	17	-	verademo.war	com//UserController.java 961	10/18/24

Directory Traversal(4 flaws)

Description

Allowing user input to control paths used in filesystem operations may enable an attacker to access or modify otherwise protected system resources that would normally be inaccessible to end users. In some cases, the user-provided input may be passed directly to the filesystem operation, or it may be concatenated to one or more fixed strings to construct a fully-qualified path.

When an application improperly cleanses special character sequences in user-supplied filenames, a path traversal (or directory traversal) vulnerability may occur. For example, an attacker could specify a filename such as "../../etc/passwd", which resolves to a file outside of the intended directory that the attacker would not normally be authorized to view.

Recommendations

Assume all user-supplied input is malicious. Validate all user-supplied input to ensure that it conforms to the expected format, using centralized data validation routines when possible. When using black lists, be sure that the sanitizing routine performs a sufficient number of iterations to remove all instances of disallowed characters and ensure that the end result is not dangerous.

Associated Flaws by CWE ID:





External Control of File Name or Path (CWE ID 73)(4 flaws)

Description

This call contains a path manipulation flaw. The argument to the function is a filename constructed using untrusted input. If an attacker is allowed to specify all or part of the filename, it may be possible to gain unauthorized access to files on the server, including those outside the webroot, that would be normally be inaccessible to end users. The level of exposure depends on the effectiveness of input validation routines, if any.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Validate all untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible. When using black lists, be sure that the sanitizing routine performs a sufficient number of iterations to remove all instances of disallowed characters.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
8	28	17	-	verademo.war	com//UserController.java 230	10/18/24
8	40	17	-	verademo.war	com//UserController.java 660	10/18/24
8	42	17	-	verademo.war	com//UserController.java 699	10/18/24
8	48	17	-	verademo.war	com//UserController.java 863	10/18/24



Encapsulation(1 flaw)

Description

Encapsulation is about defining strong security boundaries governing data and processes. Within an application, it might mean differentiation between validated and unvalidated data, between public and private members, or between one user's data and another's.

In object-oriented programming, the term encapsulation is used to describe the grouping together of data and functionality within an object and the ability to provide users with a well-defined interface in a way which hides their internal workings. Though there is some overlap with the above definition, the two definitions should not be confused as being interchangeable.

Recommendations

The wide variance of encapsulation issues makes it impractical to generalize how these issues should be addressed, beyond stating that encapsulation boundaries should be well-defined and adhered to. Refer to individual categories for specific recommendations.

Associated Flaws by CWE ID:





Deserialization of Untrusted Data (CWE ID 502)(1 flaw)

Description

The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
8	19	18	-	verademo.war	com//utils/UserFactory.java 44	10/18/24



Insufficient Input Validation(3 flaws)

Description

Weaknesses in this category are related to an absent or incorrect protection mechanism that fails to properly validate input that can affect the control flow or data flow of a program.

Recommendations

Validate input from untrusted sources before it is used. The untrusted data sources may include HTTP requests, file systems, databases, and any external systems that provide data to the application. In the case of HTTP requests, validate all parts of the request, including headers, form fields, cookies, and URL components that are used to transfer information from the browser to the server side application.

Duplicate any client-side checks on the server side. This should be simple to implement in terms of time and difficulty, and will greatly reduce the likelihood of insecure parameter values being used in the application.

Associated Flaws by CWE ID:



URL Redirection to Untrusted Site ('Open Redirect') (CWE ID 601)(2 flaws)

Description

A web application accepts a untrusted input that specifies a link to an external site, and uses that link to generate a redirect. This enables phishing attacks.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Always validate untrusted input to ensure that it conforms to the expected format, using centralized data validation routines when possible. Check the supplied URL against a whitelist of approved URLs or domains before redirecting.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
8	23	17	-	verademo.war	com//UserController.java 82	10/18/24
	24	17	-	verademo.war	com//UserController.java 95	10/18/24







Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection') (CWE ID 470)(1 flaw)

Description

A call uses reflection in an unsafe manner. An attacker can specify the class name to be instantiated, which may create unexpected control flow paths through the application. Depending on how reflection is being used, the attack vector may allow the attacker to bypass security checks or otherwise cause the application to behave in an unexpected manner. Even if the object does not implement the specified interface and a ClassCastException is thrown, the constructor of the untrusted class name will have already executed.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Validate the class name against a combination of white and black lists to ensure that only expected behavior is produced.

Instances found via Static Scan

	Flaw Id	Module #	Class #	Module	Location	Fix By
*	94	3	-	verademo.war	com//BlabController.java 571	10/18/24

Low (30 flaws)



API Abuse(17 flaws)

Description

An API is a contract between a caller and a callee. Incorrect usage of certain API functions can result in exploitable security vulnerabilities.

The most common forms of API abuse are caused by the caller failing to honor its end of this contract. For example, if a program fails to call chdir() after calling chroot(), it violates the contract that specifies how to change the active root directory in a secure fashion. Providing too few arguments to a varargs function such as printf() also violates the API contract and will cause the missing parameters to be populated with unexpected data from the stack.

Another common mishap is when the caller makes false assumptions about the callee's behavior. One example of this is when a caller expects a DNS-related function to return trustworthy information that can be used for authentication purposes. This is a bad assumption because DNS responses can be easily spoofed.

Recommendations

When calling API functions, be sure to fully understand and adhere to the specifications to avoid introducing security vulnerabilities. Do not make assumptions about trustworthiness of the data returned from API calls or use the data in a context that was unintended by that API.



Associated Flaws by CWE ID:

-

J2EE Bad Practices: Direct Management of Connections (CWE ID 245)(17 flaws)

Description

The J2EE application directly manages connections rather than using the container's resource management facilities to obtain connections as specified in the J2EE standard. Every major web application container provides pooled database connection management as part of its resource management framework. Duplicating this functionality in an application is difficult and error prone, which is part of the reason it is forbidden under the J2EE standard.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Request the connection from the container rather than attempting to access it directly.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
151	3	-	verademo.war	com//BlabController.java 69	
152	3	-	verademo.war	com//BlabController.java 182	
153	3	-	verademo.war	com//BlabController.java 232	
154	3	-	verademo.war	com//BlabController.java 307	
155	3	-	verademo.war	com//BlabController.java 400	
156	3	-	verademo.war	com//BlabController.java 483	
157	3	-	verademo.war	com//BlabController.java 568	
149	13	-	verademo.war	com//ResetController.java 101	
150	13	-	verademo.war	com//ResetController.java 269	
136	17	-	verademo.war	com//UserController.java 157	
138	17	-	verademo.war	com//UserController.java 246	
139	17	-	verademo.war	com//UserController.java 312	
140	17	-	verademo.war	com//UserController.java 368	
141	17	-	verademo.war	com//UserController.java 470	
142	17	-	verademo.war	com//UserController.java 573	
145	17	-	verademo.war	com//UserController.java 769	
146	17	-	verademo.war	com//UserController.java 829	

—

Cryptographic Issues(3 flaws)

Description

Applications commonly use cryptography to implement authentication mechanisms and to ensure the confidentiality and integrity of sensitive data, both in transit and at rest. The proper and accurate implementation of cryptography is extremely critical to its efficacy. Configuration or coding mistakes as well as incorrect assumptions may negate a large degree of the protection it affords, leaving the crypto implementation vulnerable to attack.



Common cryptographic mistakes include, but are not limited to, selecting weak keys or weak cipher modes, unintentionally exposing sensitive cryptographic data, using predictable entropy sources, and mismanaging or hard-coding keys.

Developers often make the dangerous assumption that they can improve security by designing their own cryptographic algorithm; however, one of the basic tenets of cryptography is that any cipher whose effectiveness is reliant on the secrecy of the algorithm is fundamentally flawed.

Recommendations

Select the appropriate type of cryptography for the intended purpose. Avoid proprietary encryption algorithms as they typically rely on "security through obscurity" rather than sound mathematics. Select key sizes appropriate for the data being protected; for high assurance applications, 256-bit symmetric keys and 2048-bit asymmetric keys are sufficient. Follow best practices for key storage, and ensure that plaintext data and key material are not inadvertently exposed.

Associated Flaws by CWE ID:



Sensitive Cookie in HTTPS Session Without 'Secure' Attribute (CWE ID 614)(3 flaws)

Description

Setting the Secure attribute on an HTTP cookie instructs the web browser to send it only over a secure channel, such as a TLS connection. Issuing a cookie without the Secure attribute allows the browser to transmit it over unencrypted connections, which are susceptible to eavesdropping. It is particularly important to set the Secure attribute on any cookies containing sensitive data, such as authentication information (e.g. "remember me" style functionality).

Effort to Fix: 1 - Trivial implementation error. Fix is up to 5 lines of code. One hour or less to fix.

Recommendations

Set the Secure attribute for all cookies used by HTTPS sessions.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
137	17	-	verademo.war	com//UserController.java 173	
143	17	-	verademo.war	com//UserController.java 631	
135	18	-	verademo.war	com//utils/UserFactory.java 96	



Information Leakage(10 flaws)

Description

An information leak is the intentional or unintentional disclosure of information that is either regarded as sensitive within the product's own functionality or provides information about the product or its environment that could be useful in an attack. Information leakage issues are commonly overlooked because they cannot be used to directly exploit the application. However, information leaks should be viewed as building blocks that an attacker uses to carry out other, more complicated attacks.

There are many different types of problems that involve information leaks, with severities that can range widely depending on the type of information leaked and the context of the information with respect to the application. Common sources of information leakage include, but are not limited to:



- * Source code disclosure
- * Browsable directories
- * Log files or backup files in web-accessible directories
- * Unfiltered backend error messages
- * Exception stack traces
- * Server version information
- * Transmission of uninitialized memory containing sensitive data

Recommendations

Configure applications and servers to return generic error messages and to suppress stack traces from being displayed to end users. Ensure that errors generated by the application do not provide insight into specific backend issues.

Remove all backup files, binary archives, alternate versions of files, and test files from web-accessible directories of production servers. The only files that should be present in the application's web document root are files required by the application. Ensure that deployment procedures include the removal of these file types by an administrator. Keep web and application servers fully patched to minimize exposure to publicly-disclosed information leakage vulnerabilities.

Associated Flaws by CWE ID:



Generation of Error Message Containing Sensitive Information (CWE ID 209)(8 flaws)

Description

The software generates an error message that includes sensitive information about its environment, users, or associated data. The sensitive information may be valuable information on its own (such as a password), or it may be useful for launching other, more deadly attacks. If an attack fails, an attacker may use error information provided by the server to launch another more focused attack. For example, file locations disclosed by an exception stack trace may be leveraged by an attacker to exploit a path traversal issue elsewhere in the application.

Effort to Fix: 1 - Trivial implementation error. Fix is up to 5 lines of code. One hour or less to fix.

Recommendations

Ensure that only generic error messages are returned to the end user that do not reveal any additional details.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
164	1	-	verademo.war	/WEB-INF/views/blab.jsp 69	
163	2	-	verademo.war	/WEB-INF/views/blabbers.jsp 66	
162	5	-	verademo.war	/WEB-INF/views/feed.jsp 70	
161	8	-	verademo.war	/WEB-INF/views/login.jsp 58	
160	9	-	verademo.war	/WEB-INF/views/profile.jsp 63	
159	10	-	verademo.war	//views/register-finish.jsp 60	
158	11	-	verademo.war	/WEB-INF/views/register.jsp 60	
147	17	-	verademo.war	com//UserController.java 949	





Insertion of Sensitive Information Into Sent Data (CWE ID 201)(2 flaws)

Description

Sensitive information may be exposed as a result of outbound network connections made by the application.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

Ensure that the transfer of sensitive data is intended and that it does not violate application security policy or user expectations.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Loc	ation	Fix By
144	17	-	verademo.war	com	//UserController.java 711	
148	17	-	verademo.war	com	//UserController.java 949	

Very Low (0 flaws)

No flaws of this type were found

Info (18 flaws)



Code Quality(18 flaws)

Description

Code quality issues stem from failure to follow good coding practices and can lead to unpredictable behavior. These may include but are not limited to:

- * Neglecting to remove debug code or dead code
- * $\,$ Improper resource management, such as using a pointer after it has been freed
- * Using the incorrect operator to compare objects
- * Failing to follow an API or framework specification
- * Using a language feature or API in an unintended manner

While code quality flaws are generally less severe than other categories and usually are not directly exploitable, they may serve as indicators that developers are not following practices that increase the reliability and security of an application. For an attacker, code quality issues may provide an opportunity to stress the application in unexpected ways.

Recommendations

The wide variance of code quality issues makes it impractical to generalize how these issues should be addressed. Refer to individual categories for specific recommendations.

Associated Flaws by CWE ID:





Improper Resource Shutdown or Release (CWE ID 404)(18 flaws)

Description

The application fails to release (or incorrectly releases) a system resource before it is made available for re-use. This condition often occurs with resources such as database connections or file handles. Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, it may be possible to launch a denial of service attack by depleting the resource pool.

Effort to Fix: 2 - Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.

Recommendations

When a resource is created or allocated, the developer is responsible for properly releasing the resource as well as accounting for all potential paths of expiration or invalidation. Ensure that all code paths properly release resources.

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
176	3	-	verademo.war	com//BlabController.java 182	
181	6	-	verademo.war	com//IgnoreCommand.java 38	
182	6	-	verademo.war	com//IgnoreCommand.java 40	
179	7	-	verademo.war	com//ListenCommand.java 38	
180	7	-	verademo.war	com//ListenCommand.java 40	
177	12	-	verademo.war	/RemoveAccountCommand.java 38	
178	12	-	verademo.war	/RemoveAccountCommand.java 40	
174	15	-	verademo.war	com//ToolsController.java 65	
175	15	-	verademo.war	com//ToolsController.java 99	
165	17	-	verademo.war	com//UserController.java 166	
166	17	-	verademo.war	com//UserController.java 246	
167	17	-	verademo.war	com//UserController.java 250	
168	17	-	verademo.war	com//UserController.java 251	
169	17	-	verademo.war	com//UserController.java 312	
170	17	-	verademo.war	com//UserController.java 315	
171	17	-	verademo.war	com//UserController.java 316	
172	17	-	verademo.war	com//UserController.java 494	
173	17	-	verademo.war	com//UserController.java 495	



About Veracode's Methodology

The Veracode platform uses static and dynamic analysis (for web applications) to identify software security flaws in your applications. Using both static and dynamic analysis helps reduce false negatives and detect a broader range of security flaws. Veracode static analysis models the application into an intermediate representation, which is then analyzed for security flaws using a set of automated security tests. Dynamic analysis uses an automated penetration testing technique to detect security flaws at runtime. Once the automated process is complete, a security technician verifies the output to ensure the lowest false positive rates in the industry. The end result is an accurate list of security flaws for the classes of automated scans applied to the application.

Veracode Rating System Using Multiple Analysis Techniques

Higher assurance applications require more comprehensive analysis to accurately score their security quality. Because each analysis technique (automated static, automated dynamic, manual penetration testing or manual review) has differing false negative (FN) rates for different types of security flaws, any single analysis technique or even combination of techniques is bound to produce a certain level of false negatives. Some false negatives are acceptable for lower business critical applications, so a less expensive analysis using only one or two analysis techniques is acceptable. At higher business criticality the FN rate should be close to zero, so multiple analysis techniques are recommended.

Application Security Policies

The Veracode platform allows an organization to define and enforce a uniform application security policy across all applications in its portfolio. The elements of an application security policy include the target Veracode Level for the application; types of flaws that should not be in the application (which may be defined by flaw severity, flaw category, CWE, or a common standard including OWASP, CWE/SANS Top 25, or PCI); minimum Veracode security score; required scan types and frequencies; and grace period within which any policy-relevant flaws should be fixed.

Policy constraints

Policies have three main constraints that can be applied: rules, required scans, and remediation grace periods.

Evaluating applications against a policy

When an application is evaluated against a policy, it can receive one of four assessments:

Not assessed The application has not yet had a scan published

Passed The application has passed all the aspects of the policy, including rules, required scans, and grace period. **Did not pass** The application has not completed all required scans; has not achieved the target Veracode Level; or has one or more policy relevant flaws that have exceeded the grace period to fix.

Conditional pass The application has one or more policy relevant flaws that have not yet exceeded the grace period to fix.

Understand Veracode Levels

The Veracode Level (VL) achieved by an application is determined by type of testing performed on the application, and the severity and types of flaws detected. A minimum security score (defined below) is also required for each level.

There are five Veracode Levels denoted as VL1, VL2, VL3, VL4, and VL5. VL1 is the lowest level and is achieved by demonstrating that security testing, automated static or dynamic, is utilized during the SDLC. VL5 is the highest level and is achieved by performing automated and manual testing and removing all significant flaws. The Veracode Levels VL2, VL3, and VL4 form a continuum of increasing software assurance between VL1 and VL5.

For IT staff operating applications, Veracode Levels can be used to set application security policies. For deployment scenarios of different business criticality, differing VLs should be made requirements. For example, the policy for applications that handle credit card transactions, and therefore have PCI compliance requirements, should be VL5. A medium business criticality internal application could have a policy requiring VL3.

Software developers can decide which VL they want to achieve based on the requirements of their customers. Developers of software that is mission critical to most of their customers will want to achieve VL5. Developers of general purpose business software may want



to achieve VL3 or VL4. Once the software has achieved a Veracode Level it can be communicated to customers through a Veracode Report or through the Veracode Directory on the Veracode web site.

Criteria for achieving Veracode Levels

The following table defines the details to achieve each Veracode Level. The criteria for all columns: Flaw Severities Not Allowed, Flaw Categories not Allowed, Testing Required, and Minimum Score.

*Dynamic is only an option for web applications.

Veracode Level	Flaw Severities Not Allowed	Testing Required*	Minimum Score
VL5	V.High, High, Medium	Static AND Manual	90
VL4	V.High, High, Medium	Static	80
VL3	V.High, High	Static	70
VL2	V.High	Static OR Dynamic OR Manual	60
VL1		Static OR Dynamic OR Manual	

When multiple testing techniques are used it is likely that not all testing will be performed on the exact same build. If that is the case the latest test results from a particular technique will be used to calculate the current Veracode Level. After 6 months test results will be deemed out of date and will no longer be used to calculate the current Veracode Level.

Business Criticality

The foundation of the Veracode rating system is the concept that more critical applications require higher security quality scores to be acceptable risks. Less business critical applications can tolerate lower security quality. The business criticality is dictated by the typical deployed environment and the value of data used by the application. Factors that determine business criticality are: reputation damage, financial loss, operational risk, sensitive information disclosure, personal safety, and legal violations.

US. Govt. OMB Memorandum M-04-04; NIST FIPS Pub. 199

Business Criticality Description

Very High	Mission critical for business/safety of life and limb on the line
High	Exploitation causes serious brand damage and financial loss with long term business impact
Medium	Applications connected to the internet that process financial or private customer information
Low	Typically internal applications with non-critical business impact
Very Low	Applications with no material business impact

Business Criticality Definitions

Very High (BC5) This is typically an application where the safety of life or limb is dependent on the system; it is mission critical the application maintain 100% availability for the long term viability of the project or business. Examples are control software for industrial, transportation or medical equipment or critical business systems such as financial trading systems.

High (BC4) This is typically an important multi-user business application reachable from the internet and is critical that the application maintain high availability to accomplish its mission. Exploitation of high criticality applications cause serious brand damage and business/financial loss and could lead to long term business impact.

Medium (BC3) This is typically a multi-user application connected to the internet or any system that processes financial or private customer information. Exploitation of medium criticality applications typically result in material business impact resulting



in some financial loss, brand damage or business liability. An example is a financial services company's internal 401K management system.

Low (BC2) This is typically an internal only application that requires low levels of application security such as authentication to protect access to non-critical business information and prevent IT disruptions. Exploitation of low criticality applications may lead to minor levels of inconvenience, distress or IT disruption. An example internal system is a conference room reservation or business card order system.

Very Low (BC1) Applications that have no material business impact should its confidentiality, data integrity and availability be affected. Code security analysis is not required for applications at this business criticality, and security spending should be directed to other higher criticality applications.

Scoring Methodology

The Veracode scoring system, Security Quality Score, is built on the foundation of two industry standards, the Common Weakness Enumeration (CWE) and Common Vulnerability Scoring System (CVSS). CWE provides the dictionary of security flaws and CVSS provides the foundation for computing severity, based on the potential Confidentiality, Integrity and Availability impact of a flaw if exploited.

The Security Quality Score is a single score from 0 to 100, where 0 is the most insecure application and 100 is an application with no detectable security flaws. The score calculation includes non-linear factors so that, for instance, a single Severity 5 flaw is weighted more heavily than five Severity 1 flaws, and so that each additional flaw at a given severity contributes progressively less to the score.

Veracode assigns a severity level to each flaw type based on three foundational application security requirements — Confidentiality, Integrity and Availability. Each of the severity levels reflects the potential business impact if a security breach occurs across one or more of these security dimensions.

Confidentiality Impact

According to CVSS, this metric measures the impact on confidentiality if a exploit should occur using the vulnerability on the target system. At the weakness level, the scope of the Confidentiality in this model is within an application and is measured at three levels of impact -None, Partial and Complete.

Integrity Impact

This metric measures the potential impact on integrity of the application being analyzed. Integrity refers to the trustworthiness and guaranteed veracity of information within the application. Integrity measures are meant to protect data from unauthorized modification. When the integrity of a system is sound, it is fully proof from unauthorized modification of its contents.

Availability Impact

This metric measures the potential impact on availability if a successful exploit of the vulnerability is carried out on a target application. Availability refers to the accessibility of information resources. Almost exclusive to this domain are denial-of-service vulnerabilities. Attacks that compromise authentication and authorization for application access, application memory, and administrative privileges are examples of impact on the availability of an application.

Security Quality Score Calculation

The overall Security Quality Score is computed by aggregating impact levels of all weaknesses within an application and representing the score on a 100 point scale. This score does not predict vulnerability potential as much as it enumerates the security weaknesses and their impact levels within the application code.

The Raw Score formula puts weights on each flaw based on its impact level. These weights are exponential and determined by empirical analysis by Veracode's application security experts with validation from industry experts. The score is normalized to a scale of 0 to 100, where a score of 100 is an application with 0 detected flaws using the analysis technique for the application's business criticality.

Understand Severity, Exploitability, and Remediation Effort

Severity and exploitability are two different measures of the seriousness of a flaw. Severity is defined in terms of the potential impact to confidentiality, integrity, and availability of the application as defined in the CVSS, and exploitability is defined in terms of the likelihood



or ease with which a flaw can be exploited. A high severity flaw with a high likelihood of being exploited by an attacker is potentially more dangerous than a high severity flaw with a low likelihood of being exploited.

Remediation effort, also called Complexity of Fix, is a measure of the likely effort required to fix a flaw. Together with severity, the remediation effort is used to give Fix First guidance to the developer.

Veracode Flaw Severities

Veracode flaw severities are defined as follows:

Severity	Description
Very High	The offending line or lines of code is a very serious weakness and is an easy target for an attacker. The code should be modified immediately to avoid potential attacks.
High	The offending line or lines of code have significant weakness, and the code should be modified immediately to avoid potential attacks.
Medium	A weakness of average severity. These should be fixed in high assurance software. A fix for this weakness should be considered after fixing the very high and high for medium assurance software.
Low	This is a low priority weakness that will have a small impact on the security of the software. Fixing should be consideration for high assurance software. Medium and low assurance software can ignore these flaws.
Very Low	Minor problems that some high assurance software may want to be aware of. These flaws can be safely ignored in medium and low assurance software.
Informational	Issues that have no impact on the security quality of the application but which may be of interest to the reviewer.

Informational findings

Informational severity findings are items observed in the analysis of the application that have no impact on the security quality of the application but may be interesting to the reviewer for other reasons. These findings may include code quality issues, API usage, and other factors.

Informational severity findings have no impact on the security quality score of the application and are not included in the summary tables of flaws for the application.

Exploitability

Each flaw instance in a static scan may receive an exploitability rating. The rating is an indication of the intrinsic likelihood that the flaw may be exploited by an attacker. Veracode recommends that the exploitability rating be used to prioritize flaw remediation within a particular group of flaws with the same severity and difficulty of fix classification.

The possible exploitability ratings include:

Exploitability	Description
V. Unlikely	Very unlikely to be exploited
Unlikely	Unlikely to be exploited



Exploitability	Description
Neutral	Neither likely nor unlikely to be exploited.
Likely	Likely to be exploited
V. Likely	Very likely to be exploited

Note: All reported flaws found via dynamic scans are assumed to be exploitable, because the dynamic scan actually executes the attack in question and verifies that it is valid.

Effort/Complexity of Fix

Each flaw instance receives an effort/complexity of fix rating based on the classification of the flaw. The effort/complexity of fix rating is given on a scale of 1 to 5, as follows:

Effort/Complexity of Fix	Description	
5	Complex design error. Requires significant redesign.	
4	Simple design error. Requires redesign and up to 5 days to fix.	
3	Complex implementation error. Fix is approx. 51-500 lines of code. Up to 5 days to fix.	
2	Implementation error. Fix is approx. 6-50 lines of code. 1 day to fix.	
1	Trivial implementation error. Fix is up to 5 lines of code. One hour or less to fix.	

Flaw Types by Severity Level

The flaw types by severity level table provides a summary of flaws found in the application by Severity and Category. The table puts the Security Quality Score into context by showing the specific breakout of flaws by severity, used to compute the score as described above. If multiple analysis techniques are used, the table includes a breakout of all flaws by category and severity for each analysis type performed.

Flaws by Severity

The flaws by severity chart shows the distribution of flaws by severity. An application can get a mediocre security rating by having a few high risk flaws or many medium risk flaws.

Flaws in Common Modules

The flaws in common modules listing shows a summary of flaws in shared dependency modules in this application. A shared dependency is a dependency that is used by more than one analyzed module. Each module is listed with the number of executables that consume it as a dependency and a summary of the impact on the application's security score of the flaws found in the dependency.

The score impact represents the amount that the application score would increase if all the flaws in the shared dependency module were fixed. This information can be used to focus remediation efforts on common modules with a higher impact on the application security score.

Only common modules that were uploaded with debug information are included in the Flaws in Common Modules listing.



Action Items

The Action Items section of the report provides guidance on the steps required to bring the application to a state where it passes its assigned policy. These steps may include fixing or mitigating flaws or performing additional scans. The section also includes best practice recommendations to improve the security quality of the application.

Common Weakness Enumeration (CWE)

The Common Weakness Enumeration (CWE) is an industry standard classification of types of software weaknesses, or flaws, that can lead to security problems. CWE is widely used to provide a standard taxonomy of software errors. Every flaw in a Veracode report is classified according to a standard CWE identifier.

More guidance and background about the CWE is available at http://cwe.mitre.org/data/index.html.

About Manual Assessments

The Veracode platform can include the results from a manual assessment (usually a penetration test or code review) as part of a report. These results differ from the results of automated scans in several important ways, including objectives, attack vectors, and common attack patterns.

A manual penetration assessment is conducted to observe the application code in a run-time environment and to simulate real-world attack scenarios. Manual testing is able to identify design flaws, evaluate environmental conditions, compound multiple lower risk flaws into higher risk vulnerabilities, and determine if identified flaws affect the confidentiality, integrity, or availability of the application.

Objectives

The stated objectives of a manual penetration assessment are:

- Perform testing, using proprietary and/or public tools, to determine whether it is possible for an attacker to:
- Circumvent authentication and authorization mechanisms
- · Escalate application user privileges
- Hijack accounts belonging to other users
- Violate access controls placed by the site administrator
- Alter data or data presentation
- · Corrupt application and data integrity, functionality and performance
- Circumvent application business logic
- Circumvent application session management
- · Break or analyze use of cryptography within user accessible components
- Determine possible extent access or impact to the system by attempting to exploit vulnerabilities
- Score vulnerabilities using the Common Vulnerability Scoring System (CVSS)
- Provide tactical recommendations to address security issues of immediate consequence

Provide strategic recommendations to enhance security by leveraging industry best practices

Attack vectors

In order to achieve the stated objectives, the following tests are performed as part of the manual penetration assessment, when applicable to the platforms and technologies in use:

- Cross Site Scripting (XSS)
- SQL Injection
- Command Injection
- Cross Site Request Forgery (CSRF)
- Authentication/Authorization Bypass
- · Session Management testing, e.g. token analysis, session expiration, and logout effectiveness
- · Account Management testing, e.g. password strength, password reset, account lockout, etc.
- Directory Traversal
- Response Splitting
- Stack/Heap Overflows
- Format String Attacks



- Cookie Analysis
- Server Side Includes Injection
- · Remote File Inclusion
- LDAP Injection
- XPATH Injection
- Internationalization attacks
- Denial of Service testing at the application layer only
- · AJAX Endpoint Analysis
- · Web Services Endpoint Analysis
- HTTP Method Analysis
- · SSL Certificate and Cipher Strength Analysis
- Forced Browsing

CAPEC Attack Pattern Classification

The following attack pattern classifications are used to group similar application flaws discovered during manual penetration testing. Attack patterns describe the general methods employed to access and exploit the specific weaknesses that exist within an application. CAPEC (Common Attack Pattern Enumeration and Classification) is an effort led by Cigital, Inc. and is sponsored by the United States Department of Homeland Security's National Cyber Security Division.

Abuse of Functionality

Exploitation of business logic errors or misappropriation of programmatic resources. Application functions are developed to specifications with particular intentions, and these types of attacks serve to undermine those intentions.

Examples:

- · Exploiting password recovery mechanisms
- · Accessing unpublished or test APIs
- · Cache poisoning

Spoofing

Impersonation of entities or trusted resources. A successful attack will present itself to a verifying entity with an acceptable level of authenticity.

Examples:

- Man in the middle attacks
- · Checksum spoofing
- Phishing attacks

Probabilistic Techniques

Using predictive capabilities or exhaustive search techniques in order to derive or manipulate sensitive information. Attacks capitalize on the availability of computing resources or the lack of entropy within targeted components.

Examples:

- Password brute forcing
- Cryptanalysis
- · Manipulation of authentication tokens

Exploitation of Authentication

Circumventing authentication requirements to access protected resources. Design or implementation flaws may allow authentication checks to be ignored, delegated, or bypassed.

Examples:

- · Cross-site request forgery
- · Reuse of session identifiers
- Flawed authentication protocol



Resource Depletion

Affecting the availability of application components or resources through symmetric or asymmetric consumption. Unrestricted access to computationally expensive functions or implementation flaws that affect the stability of the application can be targeted by an attacker in order to cause denial of service conditions.

Examples:

- · Flooding attacks
- Unlimited file upload size
- Memory leaks

Exploitation of Privilege/Trust

Undermining the application's trust model in order to gain access to protected resources or gain additional levels of access as defined by the application. Applications that implicitly extend trust to resources or entities outside of their direct control are susceptible to attack.

Examples:

- Insufficient access control lists
- Circumvention of client side protections
- · Manipulation of role identification information

Injection

Inserting unexpected inputs to manipulate control flow or alter normal business processing. Applications must contain sufficient data validation checks in order to sanitize tainted data and prevent malicious, external control over internal processing.

Examples:

- SQL Injection
- · Cross-site scripting
- XML Injection

Data Structure Attacks

Supplying unexpected or excessive data that results in more data being written to a buffer than it is capable of holding. Successful attacks of this class can result in arbitrary command execution or denial of service conditions.

Examples:

- Buffer overflow
- Integer overflow
- · Format string overflow

Data Leakage Attacks

Recovering information exposed by the application that may itself be confidential or may be useful to an attacker in discovering or exploiting other weaknesses. A successful attack may be conducted passive observation or active interception methods. This attack pattern often manifests itself in the form of applications that expose sensitive information within error messages.

Examples:

- Sniffing clear-text communication protocols
- Stack traces returned to end users
- · Sensitive information in HTML comments

Resource Manipulation

Manipulating application dependencies or accessed resources in order to undermine security controls and gain unauthorized access to protected resources. Applications may use tainted data when constructing paths to local resources or when constructing processing environments.



Examples:

- · Carriage Return Line Feed log file injection
- File retrieval via path manipulation
- User specification of configuration files

Time and State Attacks

Undermining state condition assumptions made by the application or capitalizing on time delays between security checks and performed operations. An application that does not enforce a required processing sequence or does not handle concurrency adequately will be susceptible to these attack patterns.

Examples:

- · Bypassing intermediate form processing steps
- · Time-of-check and time-of-use race conditions
- · Deadlock triggering to cause a denial of service

Terms of Use

Use and distribution of this report are governed by the agreement between Veracode and its customer. In particular, this report and the results in the report cannot be used publicly in connection with Veracode's name without written permission.



Appendix A: Referenced Source Files

Id	Filename	Path
1	blab.jsp	/WEB-INF/views/
2	blabbers.jsp	/WEB-INF/views/
3	BlabController.java	com/veracode/verademo/controller/
4	Constants.java	com/veracode/verademo/utils/
5	feed.jsp	/WEB-INF/views/
6	IgnoreCommand.java	com/veracode/verademo/commands/
7	ListenCommand.java	com/veracode/verademo/commands/
8	login.jsp	/WEB-INF/views/
9	profile.jsp	/WEB-INF/views/
10	register-finish.jsp	/WEB-INF/views/
11	register.jsp	/WEB-INF/views/
12	Remove Account Command. java	com/veracode/verademo/commands/
13	ResetController.java	com/veracode/verademo/controller/
14	tools.jsp	/WEB-INF/views/
15	ToolsController.java	com/veracode/verademo/controller/
16	User.java	com/veracode/verademo/utils/
17	UserController.java	com/veracode/verademo/controller/
18	UserFactory.java	com/veracode/verademo/utils/