VERACODE

Veracode Detailed Report

Application Security Report
As of 27 Feb 2024

Prepared for:	BANCO DAVIVIENDA S.A.
Prepared on:	27 de febrero de 2024
Application:	CreditoBajoMontoJava

Industry:	Financial Services
Business Criticality:	BC5 (Very High)
Required Analysis:	Any
Type(s) of Analysis Conducted:	Static
Scope of Static Scan:	2 of 2 Modules Analyzed

Inside This Report

Veracode Detailed Report
Application Security Report
As of 27 Feb 2024

Veracode Level: VL3
Rated: 27-feb-2024

Application:	CreditoBajoMontoJava	Business Criticality:	Very High
Target Level:	None	Published Rating:	B

Scans Included in Report

Static Scan	Dynamic Scan	Manual Penetration Test
rv29v1.8.6 Score: 87 Completed: 27/02/24	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

	Any	Static	Dynamic	Manual Penetration Test
Performed:		●	○	○
Required:	●	○	○	○

Summary of Flaws Found by Severity
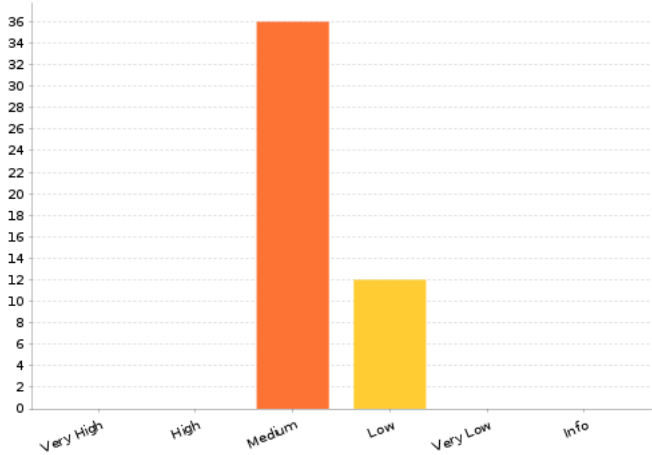
Action Items:

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.

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 [27/02/24] for 36 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 10 Medium flaws to reach a score of 90.

→ Your Static Scan was due on 27/02/24 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: 20240218234503

The following modules were included in the application scan:

Module Name	Compiler	Operating Environment	Engine Version
CreditoBajoMonto_v1.8.6.ear	JAVAC_7	Java J2SE 7	202402182 34503
JS files within CreditoBajoMontoWeb.war	JAVASCRIPT_5_1	JavaScript	202402182 34503

Flaw Types by Severity and Category

	Static Scan Security Quality Score = 87		
Very High	0		
High	0		
Medium	36		
CRLF Injection	34		
Cross-Site Scripting (XSS)	1		
Insufficient Input Validation	1		
Low	12		
Code Quality	9		
Information Leakage	3		
Very Low	0		
Informational	0		
Total	48		

Policy Evaluation

Policy Name: Davi_Veracode Recommended Very High + SCA

Revision: 2

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
Min Analysis Score	90	87	Did not pass
Max Severity	Medium	Flaws found: 36	Did not pass

Software Composition Analysis Rules

Rule type	Requirement	Findings	Status
Disallow Vulnerabilities by Severity	Medium and Above Not Allowed	3 Components	Did not pass
Disallow Component Blocklist	Prevent an application from passing policy if blocklisted components are detected	0 Blocklisted	Passed

Scan Requirements

Scan Type	Frequency	Last performed	Status
Any	Once	27/02/24	Passed

Remediation

Flaw Severity	Grace Period	Flaws Exceeding	Status
Very High	0 days	0	Passed
High	0 days	0	Passed
Medium	0 days	36	Did not pass
Low	0 days	0	Passed
Very Low	0 days	0	Passed
Informational	0 days	0	Passed

Type	Grace Period	Exceeding	Status
Min Analysis Score	0 days	1	Did not pass
Disallow Component Blocklist	0 days	0	Passed

Unsupported Frameworks

This report may have incomplete results based on the following unsupported frameworks identified during the static scan:

* Google GSON

The lack of support for all frameworks in use by this application and/or its supporting libraries may prevent the static discovery of some flaws in the application, however, it does not invalidate the flaws that were found.

Findings & Recommendations

Detailed Flaws by Severity

Very High (0 flaws)

No flaws of this type were found

High (0 flaws)

No flaws of this type were found

Medium (36 flaws)   ⊗ **Fix Required by Policy**

→ ### CRLF Injection(34 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 in HTTP Headers ('HTTP Request/Response Splitting') (CWE ID 113)(1 flaw)

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
⊗	23	3	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB	com/.../filter/CreditoFilter.java 122	27/02/24

Flaw Id	Module #	Class #	Module	Location	Fix By
			ajoMontoWeb.war		

→ **Improper Output Neutralization for Logs (CWE ID 117)(33 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
✖	31	1	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../CatalogsResource.java 54	27/02/24
✖	32	1	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../CatalogsResource.java 54	27/02/24
✖	33	1	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../CatalogsResource.java 56	27/02/24
✖	34	1	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../CatalogsResource.java 57	27/02/24
✖	35	2	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../filter/CORSFilter.java 45	27/02/24
✖	36	2	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../filter/CORSFilter.java 58	27/02/24
✖	20	3	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../filter/CreditoFilter.java 82	27/02/24
✖	21	3	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../filter/CreditoFilter.java 112	27/02/24
✖	24	3	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../filter/CreditoFilter.java 188	27/02/24

	Flaw Id	Module #	Class #	Module	Location	Fix By
❌	25	3	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../filter/CreditoFilter.java 202	27/02/24
❌	26	3	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../filter/CreditoFilter.java 275	27/02/24
❌	27	3	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../filter/CreditoFilter.java 281	27/02/24
❌	19	5	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../servlet/InicioServlet.java 76	27/02/24
❌	30	8	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../SimulatorResource.java 93	27/02/24
❌	4	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 77	27/02/24
❌	5	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 94	27/02/24
❌	6	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 113	27/02/24
❌	7	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 131	27/02/24
❌	8	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 150	27/02/24
❌	9	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 166	27/02/24
❌	10	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 183	27/02/24
❌	11	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 200	27/02/24
❌	12	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 234	27/02/24
❌	13	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 251	27/02/24
❌	14	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoB	com/.../util/SingletonCache.java 268	27/02/24

	Flaw Id	Module #	Class #	Module	Location	Fix By
				ajoMontoComunes.jar		
❌	15	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 285	27/02/24
❌	16	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 302	27/02/24
❌	17	9	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../util/SingletonCache.java 320	27/02/24
❌	1	10	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../credito/util/Util.java 497	27/02/24
❌	2	10	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../credito/util/Util.java 504	27/02/24
❌	3	10	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../credito/util/Util.java 900	27/02/24
❌	28	11	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../resources/WCMResource.java 57	27/02/24
❌	29	11	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoApi.war	com/.../resources/WCMResource.java 83	27/02/24

Eliminar línea

➡️ **Cross-Site Scripting (XSS)(1 flaw)**

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:

→ Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS) (CWE ID 80)(1 flaw)

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
❌	18	5	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../servlet/InicioServlet.java 67	27/02/24

→ **Insufficient Input Validation(1 flaw)**

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)(1 flaw)**

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
❌	22	3	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../filter/CreditoFilter.java 122	27/02/24

Low (12 flaws)

→ **Code Quality(9 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:

→ Use of Wrong Operator in String Comparison (CWE ID 597)(9 flaws)

Description
Using '==' to compare two strings for equality or '!=' for inequality actually compares the object references rather than their values. It is unlikely that this reflects the intended application logic.

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

Recommendations
Use the equals() method to compare strings, not the '==' or '!=' operator

Instances found via Static Scan

Flaw Id	Module #	Class #	Module	Location	Fix By
43	4	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoApi.war	.../EvaluationResource.java 108	
44	4	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoApi.war	.../EvaluationResource.java 116	
45	4	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoApi.war	.../EvaluationResource.java 129	
46	6	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoComunes.j ar	.../ObjectFactoryEvaluadorlightPCO.java 202	
47	6	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoComunes.j ar	.../ObjectFactoryEvaluadorlightPCO.java 205	
48	6	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoComunes.j ar	.../ObjectFactoryEvaluadorlightPCO.java 208	
40	7	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoApi.war	com/.../SignatureResource.java 139	
41	7	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB ajoMontoApi.war	com/.../SignatureResource.java 147	
42	7	-	CreditoBajoMonto_ v1.8.6.ear/CreditoB	com/.../SignatureResource.java 160	

Flaw Id	Module #	Class #	Module	Location	Fix By
				ajoMontoApi.war	

→ **Information Leakage(3 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:

→ **Insertion of Sensitive Information Into Sent Data (CWE ID 201)(3 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	Location	Fix By
39	5	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoWeb.war	com/.../servlet/InicioServlet.java 67	
37	12	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../credito/util/WcmUtil.java 146	
38	12	-	CreditoBajoMonto_v1.8.6.ear/CreditoBajoMontoComunes.jar	com/.../credito/util/WcmUtil.java 199	

Very Low (0 flaws)

No flaws of this type were found

Info (0 flaws)

No flaws of this type were found

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

Tel.+1.339.674.2500 Fax.+1.339.674.2502 URL:http://www.veracode.com

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	CatalogsResource.java	com/davivienda/web/api/resources/
2	CORSFilter.java	com/davivienda/web/api/filter/
3	CreditoFilter.java	com/davivienda/web/credito/filter/
4	EvaluationResource.java	com/davivienda/web/api/resources/
5	InicioServlet.java	com/davivienda/web/credito/servlet/
6	ObjectFactoryEvaluadorlightPCO.java	com/davivienda/credito/wscliente/evaluadorlightPCO/
7	SignatureResource.java	com/davivienda/web/api/resources/
8	SimulatorResource.java	com/davivienda/web/api/resources/
9	SingletonCache.java	com/davivienda/web/credito/util/
10	Util.java	com/davivienda/web/credito/util/
11	WCMResource.java	com/davivienda/web/api/resources/
12	WcmUtil.java	com/davivienda/web/credito/util/