



Veracode PCI Report
Application Security Report for PCI Compliance
for Example Company

September 29, 2011

Application:	Roller
Application Version:	2011.2 - Java
Application Origin:	Internally Developed
Industry:	Other
Business Criticality:	BC4 (High)
Required Analysis:	
Type(s) of Analysis Conducted:	Static and Dynamic
Scope of Static Analysis:	1 of 1 Modules Analyzed
Scope of Dynamic Analysis:	http://www.example.com/roller

Inside This Report

About this Analysis	1
Flaws Related to PCI Compliance	1
Immediate Action Items for Compliance	1
Summary of Flaws by PCI Requirement	3
Detailed PCI Compliance Flaws by Severity	4
Compliance Information	16
Methodology	18

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.

Veracode PCI Report Application Security Report for PCI Compliance for Example Company

Application: Roller
 PCI Compliance on Tests Performed **Did Not Pass***
 Version: 2011.2 - Java
 Business Criticality: BC4 (High)
 Adjusted/Published Rating: DA*/DA
 Date Scanned: May 3, 2011

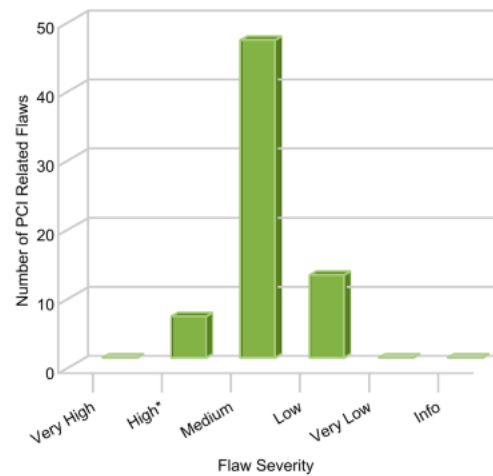
About this Report

Veracode's automated static binary and dynamic analysis techniques detect security flaws in applications. For organizations who need to comply with Payment Card Industry (PCI) regulations, this report provides guidance on fixing security flaws toward achieving compliance for PCI DSS Version 2.0 Sections 6.3.2, 6.5 and 6.6 and PCI PA-DSS Version 2.0 Sections 5.1.4 and 5.2.

Veracode's review of this application is contained in scope to the above referenced sections of the standard. In addition to this analysis, other PCI relevant issues (e.g., security training and incident response process, network security, regular testing, restriction of physical access to cardholder data, etc.) should be reviewed and assessed by a PCI Qualified Security Assessor.

For more information about PCI Regulations and Compliance, please view the Compliance Information section of this report.

Flaws By Severity



Immediate Action Items for Compliance

Veracode recommends the following approaches to achieve PCI Compliance and increase the overall security level of the application.

Fixes Necessary for PCI Compliance

- Fix 6 High flaws, 45 Medium flaws and 12 Low flaws from the Static Analysis to achieve PCI Compliance for Section 6.5. Fixing these flaws will also increase the application Static Analysis Security Quality Score to 62.
- Submit application for follow-up Static Analysis once flaws have been remediated.
- Fix 1 Medium flaw from the Dynamic Analysis to achieve PCI Compliance for Section 6.5. Fixing these flaws will also increase the application Dynamic Analysis Security Quality Score to 99.
- Submit application for follow-up Dynamic Analysis once flaws have been remediated.

Scope of Analysis (Static)

The following modules were included in the application scan:

Module Name	Compiler	Operating Environment
roller-orig.war	JAVAC_5	Java J2SE 6

→ It is important to note that this application may include additional modules which were not included in this analysis. We recommend that you contact the vendor to determine whether all modules have been included.

Scope of Analysis (Dynamic)

These are the parameters that were used to perform the application scan:

Setting	Value
Target URL	http://www.example.com/roller
Restrict to Directory	true
Number of Links Visited	310
Logged In Successfully	N/A
Login User ID	
Scan Began	Feb 8, 2010 9:37:04 AM

→ It is important to note that this application may include additional directories or URLs which were not included in this analysis. We recommend that you contact the vendor to determine whether all relevant URLs have been included.

Policy Control

Policy Name: PCI 2.0

Revision: 1

Policy Status: Did Not Pass

Description

Starting with PCI version 2.0, PCI compliance requires that an application have no flaws that are categorized in security standards like the OWASP Top 10, CWE/SANS Top 25, or the CERT Secure Coding Standard. PCI compliance also requires that flaws marked as "High" severity (in Veracode's scale, High and above) must be fixed and that scans must be performed annually.

Rules

Rule type	Requirement	Findings	Status
Standard	CERT	Flaws found: 28	Did not pass
Standard	OWASP	Flaws found: 0	Passed
Max Severity	High	Flaws found: 6*	Did not pass
Standard	SANS Top 25	Flaws found: 60*	Did not pass

* - Reflects violated rules that have mitigated flaws

Scan Requirements

Scan Type	Frequency	Last performed	Status
Any	Annually	5/3/11	Passed

Remediation

Flaw Severity	Grace Period	Flaws Exceeding	Status
Very High	0 days	0	Passed
High	0 days	6	Did not pass
Medium	0 days	46	Did not pass
Low	0 days	12	Did not pass
Very Low	0 days	0	Passed
Informational	0 days	0	Passed

Type	Grace Period	Exceeding	Status
Min Analysis Score	0 days	0	Passed

Detailed PCI Compliance Related Flaws by Severity

Very High (0 flaws)

No flaws of this type were found

High (6 flaws*)

→ SQL Injection(6 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 Software Flaw Types:

→ Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (CWE ID 89)(6 flaws*)

Description

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

Effort/Complexity of Fix: 3

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 user-supplied input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

Module #	Class #	Module	Location	Fix By	Flaw Id
19	-	roller-orig.war	org/.../ConsistencyCheck.java 213	4/13/11	356
19	-	roller-orig.war	org/.../ConsistencyCheck.java 220	4/13/11	373
19	-	roller-orig.war	org/.../ConsistencyCheck.java 245	4/13/11	387
19	-	roller-orig.war	org/.../ConsistencyCheck.java 259	4/13/11	340
19	-	roller-orig.war	org/.../ConsistencyCheck.java 290	4/13/11	530
19	-	roller-orig.war	org/.../ConsistencyCheck.java 321	4/13/11	411

Medium (46 flaws)

→ Code Quality(1 flaw)

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 Software Flaw Types:

→ 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 user-supplied class name will have already executed.

Effort/Complexity of Fix: 2

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

Module #	Class #	Module	Location	Fix By	Flaw Id
55	-	roller-orig.war	org/.../model/ModelLoader.java 91	4/13/11	326

→ Credentials Management(4 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 Software Flaw Types:

→ Plaintext Storage of a Password (CWE ID 256)(1 flaw)

Description

A method reads and/or stores sensitive information in plaintext, making the data more susceptible to compromise.

Effort/Complexity of Fix: 4

Recommendations

Never store sensitive data in plaintext. Consider using cryptographic hashes as an alternative to plaintext.

Instances found via Static Scan

Module #	Class #	Module	Location	Fix By	Flaw Id
19	-	roller-orig.war	org/.../ConsistencyCheck.java 95	4/13/11	414

→ Use of Hard-coded Password (CWE ID 259)(3 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/Complexity of Fix: 4

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

Module #	Class #	Module	Location	Fix By	Flaw Id
22	-	roller-orig.war	.../CustomUserRegistry.java 94	4/13/11	480
108	-	roller-orig.war	org/.../actions/UserNewAction.java 219	4/13/11	501
108	-	roller-orig.war	org/.../actions/UserNewAction.java 220	4/13/11	440

→ Cross-Site Scripting(25 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 Software Flaw Types:

→ Improper Neutralization of Script in Attributes in a Web Page (CWE ID 83)(1 flaw)

Description

The application does not filter text or other data for potentially malicious HTML content. This enables an attacker to craft arbitrary HTML content. This vulnerability typically requires that an attacker be able to submit JavaScript <script> tags as part of a field that is re-displayed to one or more users. The <script> tag contains instructions that are executed in a user's web browser, not on the web application server. JavaScript functions can be used to write raw HTML, read cookie values, pull JavaScript code from a third-party web server, or send data to a third-party web server.

Effort/Complexity of Fix: 3

Recommendations

Cross-site scripting and HTML injection attacks can be defeated by applying robust input validation filters for all data received from the web browser. 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 entities, then limit access to specific elements or attributes. Use the programming language's built-in routines to remove potentially malicious characters.

Attack Vectors found via Dynamic Scan

URL	Parameter	Exploitability	Flaw Id
http://10.0.4.89:28080/theme		-	301

→ Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS) (CWE ID 80)(24 flaws)

Description

This call contains a cross-site scripting (XSS) flaw. The application populates the HTTP response with user-supplied 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/Complexity of Fix: 3

Recommendations

Properly encode all untrusted data before using it to construct an HTTP response. The encoding method should be chosen based on the specific 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 (e.g. `StringEscapeUtils.escapeHtml()` in J2EE or `HttpUtility.HtmlEncode()` in ASP.NET). In addition, as a best practice, always validate user-supplied input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances found via Static Scan

Module #	Class #	Module	Location	Fix By	Flaw Id
4	-	roller-orig.war	/WEB-INF/.../bannerStatus.jsp 45	4/13/11	386
4	-	roller-orig.war	/WEB-INF/.../bannerStatus.jsp 62	4/13/11	515
4	-	roller-orig.war	/WEB-INF/.../bannerStatus.jsp 83	4/13/11	343
12	-	roller-orig.war	org/.../calendar/CalendarTag.java 186	4/13/11	358
12	-	roller-orig.war	org/.../calendar/CalendarTag.java 191	4/13/11	379
12	-	roller-orig.war	org/.../calendar/CalendarTag.java 244	4/13/11	352
15	-	roller-orig.war	.../CommentAuthenticatorServlet.java 68	4/13/11	324
30	-	roller-orig.war	/WEB-INF/.../FolderForm.jsp 56	4/13/11	353
30	-	roller-orig.war	/WEB-INF/.../FolderForm.jsp 74	4/13/11	488
33	-	roller-orig.war	/WEB-INF/.../jsps/tiles/head.jsp 30	4/13/11	513
87	-	roller-orig.war	org/.../tags/RssBadgeTag.java 56	4/13/11	484
89	-	roller-orig.war	.../ShowEntrySummaryTag.java 70	4/13/11	402
96	-	roller-orig.war	/.../tiles/tiles-mainmenupage.jsp 20	4/13/11	359
97	-	roller-orig.war	/.../tiles/tiles-simplepage.jsp 20	4/13/11	461
98	-	roller-orig.war	/.../tiles/tiles-tabbedpage.jsp 20	4/13/11	327
103	-	roller-orig.war	/WEB-INF/.../UploadFile.jsp 102	4/13/11	383
106	-	roller-orig.war	org/.../ajax/UserDataServlet.java 76	4/13/11	367
106	-	roller-orig.war	org/.../ajax/UserDataServlet.java 78	4/13/11	365
113	-	roller-orig.war	/WEB-INF/.../WeblogEdit.jsp 401	4/13/11	363
113	-	roller-orig.war	/WEB-INF/.../WeblogEdit.jsp 407	4/13/11	509
113	-	roller-orig.war	/WEB-INF/.../WeblogEdit.jsp 408	4/13/11	514
117	-	roller-orig.war	/.../WeblogEntryRemove.jsp 23	4/13/11	474
117	-	roller-orig.war	/.../WeblogEntryRemove.jsp 28	4/13/11	394
117	-	roller-orig.war	/.../WeblogEntryRemove.jsp 29	4/13/11	398

→ Cryptographic Issues(2 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 Software Flaw Types:

→ Insufficient Entropy (CWE ID 331)(2 flaws)

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/Complexity of Fix: 2

Recommendations

If this random number is used where security is a concern, such as generating a session key or session identifier, use a trusted cryptographic random number generator instead. These can be found on the Windows platform in the CryptoAPI or in an open source library such as OpenSSL.

Instances found via Static Scan

Module #	Class #	Module	Location	Fix By	Flaw Id
52	-	roller-orig.war	.../MathCommentAuthenticator.java 46	3/17/10	204
52	-	roller-orig.war	.../MathCommentAuthenticator.java 47	3/17/10	32

→ Directory Traversal(13 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 Software Flaw Types:

→ External Control of File Name or Path (CWE ID 73)(13 flaws)

Description

This call contains a path manipulation flaw. The argument to the function is a filename constructed using user-supplied 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/Complexity of Fix: 2

Recommendations

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.

Instances found via Static Scan

Module #	Class #	Module	Location	Fix By	Flaw Id
24	-	roller-orig.war	org/.../DiskFeedInfoCache.java 53	4/13/11	337
24	-	roller-orig.war	org/.../DiskFeedInfoCache.java 82	4/13/11	413
28	-	roller-orig.war	org/.../FileManagerImpl.java 149	4/13/11	450
28	-	roller-orig.war	org/.../FileManagerImpl.java 171	4/13/11	403
28	-	roller-orig.war	org/.../FileManagerImpl.java 182	4/13/11	535
28	-	roller-orig.war	org/.../FileManagerImpl.java 206	4/13/11	487
28	-	roller-orig.war	org/.../FileManagerImpl.java 212	4/13/11	432
28	-	roller-orig.war	org/.../FileManagerImpl.java 240	4/13/11	521
35	-	roller-orig.war	.../HibernatePlanetManagerImpl.java 408	4/13/11	320
41	-	roller-orig.war	.../ImportEntriesAction.java 86	4/13/11	310
41	-	roller-orig.war	.../ImportEntriesAction.java 173	4/13/11	485
80	-	roller-orig.war	org/.../ResourceServlet.java 102	4/13/11	421
82	-	roller-orig.war	org/.../RollerAtomHandler.java 646	4/13/11	481

→ Time and State(1 flaw)

Description

Time and State flaws are related to unexpected interactions between threads, processes, time, and information. These interactions happen through shared state: semaphores, variables, the filesystem, and basically anything that can store information. Vulnerabilities occur when there is a discrepancy between the programmer's assumption of how a program executes and what happens in reality.

State issues result from improper management or invalid assumptions about system state, such as assuming mutable objects are immutable. Though these conditions are less commonly exploited by attackers, state issues can lead to unpredictable or undefined application behavior.

Recommendations

Limit the interleaving of operations on resources from multiple processes. Use locking mechanisms to protect resources effectively. Follow best practices with respect to mutable objects and internal references. Pay close attention to asynchronous actions in processes and make copious use of sanity checks in systems that may be subject to synchronization errors.

Associated Software Flaw Types:

→ **Insecure Temporary File (CWE ID 377)(1 flaw)**

Description

Creating and using insecure temporary files can leave application and system data vulnerable to attack. In particular, file names created by the tmpnam family of functions can be easily guessed by an attacker. If an attacker can predict the filename and create a malicious collision, he may be able to manipulate the behavior of the application.

Effort/Complexity of Fix: 2

Recommendations

Ensure that unpredictable names are used for temporary files and that files are created in a secure directory with appropriate permissions. Using mkstemp() is a reasonably safe way to create temporary files. It will attempt to create and open a unique file based on a filename template provided by the user, combined with a series of randomly generated characters. Note that mkstemp() is safe if only the descriptor is used and the returned filename is not used in a subsequent function call with extra privileges. Using mkstemp() does not completely eliminate race conditions but does provide better protection than other methods.

Instances found via Static Scan

Module #	Class #	Module	Location	Fix By	Flaw Id
82	-	roller-orig.war	org/.../RollerAtomHandler.java 634	3/17/10	283

Low (12 flaws)

→ **Code Quality(1 flaw)**

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 Software Flaw Types:

→ Improper Resource Shutdown or Release (CWE ID 404)(1 flaw)

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/Complexity of Fix: 2

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

Module #	Class #	Module	Location	Fix By	Flaw Id
19	-	roller-orig.war	org/.../ConsistencyCheck.java 160	4/13/11	527

→ 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 Software Flaw Types:

→ 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 an SSL 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/Complexity of Fix: 1

Recommendations

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

Instances found via Static Scan

Module #	Class #	Module	Location	Fix By	Flaw Id
49	-	roller-orig.war	/.../logout-redirect.jsp 30	4/13/11	314
79	-	roller-orig.war	org/.../core/util/RequestUtil.java 195	4/13/11	449
79	-	roller-orig.war	org/.../core/util/RequestUtil.java 250	4/13/11	395

→ Information Leakage(8 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 Software Flaw Types:

→ **Information Exposure Through an Error Message (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/Complexity of Fix: 1

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

Module #	Class #	Module	Location	Fix By	Flaw Id
2	-	roller-orig.war	org/.../AtomAdminServlet.java 63	3/17/10	189
2	-	roller-orig.war	org/.../AtomAdminServlet.java 87	3/17/10	46
2	-	roller-orig.war	org/.../AtomAdminServlet.java 111	3/17/10	284
2	-	roller-orig.war	org/.../AtomAdminServlet.java 134	3/17/10	71
25	-	roller-orig.war	/roller-ui/tools/dstest.jsp 44	3/17/10	143
31	-	roller-orig.war	/WEB-INF/.../jsps/tiles/footer.jsp 21	4/13/11	477
100	-	roller-orig.war	org/.../TrackbackServlet.java 131	4/13/11	496
100	-	roller-orig.war	org/.../TrackbackServlet.java 220	4/13/11	446

Very Low (0 flaws)

No flaws of this type were found

Info (0 flaws)

No flaws of this type were found

PCI Compliance Information

PCI Overview

The PCI Security Standards Council is an open global forum for the ongoing development, enhancement, storage, dissemination and implementation of security standards for account data protection. The organization was founded by American Express, Discover Financial Services, JCB International, MasterCard Worldwide and Visa Inc. and is chartered with enhancing payment account data security by fostering broad adoption of the PCI Security Standards.

The PCI Data Security Standards (PCI DSS) Version 2.0 represents a common set of industry tools and measurements to help ensure the safe handling of sensitive information which Merchants and Service Providers must meet in order to do business with the major credit card companies. The standard provides an actionable framework for developing a robust account data security process - including preventing, detecting and reacting to security incidents.

PCI DSS Scope

PCI DSS applies to merchants and service providers who accept credit cards as a form of payment or who process, store, or transmit cardholder data.

Veracode & PCI DSS

Veracode provides independent application security testing to assist merchants and service providers in meeting PCI DSS Requirement 6: Develop and Maintain Secure Systems and Applications. Specifically, Veracode's Application Security Report can be used to provide evidence of compliance with the following PCI Requirements:

PCI DSS Requirement Number	Description
6.3.2	Review of custom code prior to release to production or customers in order to identify any potential coding vulnerabilities.
6.5.1. to 6.5.9	<p>Develop applications based on secure coding guidelines. Prevent common coding vulnerabilities in software development processes, to include the following: Injection flaws, particularly SQL injection; buffer overflow; insecure cryptographic storage; insecure communications; improper error handling; all "High" vulnerabilities as identified in PCI-DSS requirement 6.2 (Note: Veracode interprets this as all Very High or High severity flaws); Cross-site scripting; improper access control; and cross-site request forgery.</p> <p>Note: The vulnerabilities listed at 6.5.1 through 6.5.9 were current with industry best practices when this version of PCI DSS was published. However, as industry best practices for vulnerability management are updated (for example, the OWASP Guide, SANS CWE Top 25, CERT Secure Coding, etc.), the current best practices must be used for these requirements.</p>
6.6	<p>For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either of the following methods:</p> <ul style="list-style-type: none"> •Reviewing public-facing web applications via manual or automated application vulnerability security assessment tools or methods, at least annually and after any changes •Installing a web-application firewall in front of public-facing web applications.

PCI PA-DSS Scope

PCI PA-DSS applies to software vendors who develop payment applications that store, process, or transmit cardholder data as part of authorization or settlement. Examples of applicable payment applications include but are not limited to POS software, e-commerce shopping carts, and web-based payment applications.

Veracode & PCI PA-DSS

Veracode provides independent application security testing to assist payment vendors in meeting PCI PA-DSS Requirement 5. The PCI PA-DSS requirement around application security testing are identical to that of the PCI DSS, thus this report can be used to satisfy the requirements of both standards. Specifically, Veracode's PCI Application Security Report can also be used to provide evidence of compliance with PCI PA-DSS because the applications security requirements of VISA PABP are equivalent to PCI as described below:

PCI PA-DSS Requirement Number	Description
5.1.4 (equivalent to PCI-DSS 6.3.2)	Review of payment application code prior to release to customers after any significant change, to identify any potential coding vulnerability.
5.2.1 to 5.2.9 (equivalent to PCI-DSS 6.5.1 to 6.5.10)	<p>Develop applications based on secure coding guidelines. Prevent common coding vulnerabilities in software development processes, to include the following: Injection flaws, particularly SQL injection; buffer overflow; insecure cryptographic storage; insecure communications; improper error handling; all "High" vulnerabilities as identified in PCI-DSS requirement 6.2 (Note: Veracode interprets this as all Very High or High severity flaws); Cross-site scripting; improper access control; and cross-site request forgery.</p> <p>Note: The vulnerabilities listed at 6.5.1 through 6.5.9 were current with industry best practices when this version of PCI DSS was published. However, as industry best practices for vulnerability management are updated (for example, the OWASP Guide, SANS CWE Top 25, CERT Secure Coding, etc.), the current best practices must be used for these requirements.</p>

About Veracode's Methodology

The Veracode platform uses static and dynamic analysis (for web applications) to inspect executables and identify security flaws in your applications. Using both static and dynamic analysis helps reduce false negatives and detect a broader range of security flaws. The static binary analysis engine models the binary executable into an intermediate representation, which is then verified for security flaws using a set of automated security scans. 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 on a five point scale:

Severity	Name	Description
5	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.
4	High	The offending line or lines of code have significant weakness, and the code should be modified immediately to avoid potential attacks.
3	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.
2	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.
1	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.
0	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 0) 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 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: Mitigation Comments (by Example Company)

NOTE: Veracode does not review the mitigation strategy described below and is not responsible for its contents or the accuracy of any statements provided.

High (1 flaw)

→ SQL Injection(1 flaw) *PCI Requirement 5.2.1 / 6.5.1*

Associated Software Flaw Types:

→ Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (CWE ID 89)(1 flaw) *PCI Requirement 5.2.1 / 6.5.1*

Instances found via Static Scan

Flaw Id	Exploitability	Module #	Class #	Module	Location
328	Neutral	19	-	roller-orig.war	org/.../ConsistencyCheck.java 234
↳ <i>Mitigated as Potential False Positive</i> foo					
↳ <i>Mitigated by Design</i> We make an organizational decision to trust the data in the database based on our protection of the database from SQL injection.					
↳ <i>Mitigation Accepted</i> Concur					

Appendix B: Referenced Source Files

Id	Filename	Path
1	AcronymsPlugin.java	org/apache/roller/ui/rendering/plugins/
2	AtomAdminServlet.java	org/apache/roller/webservices/adminapi/
3	BakeWeblogForm.java	org/apache/roller/ui/authoring/struts/formbeans/
4	bannerStatus.jsp	/WEB-INF/jsp/tiles/
5	Blacklist.java	org/apache/roller/util/
6	BlacklistUpdateTask.java	org/apache/roller/ui/core/tasks/
7	BookmarkForm.java	org/apache/roller/ui/authoring/struts/forms/
8	BookmarksAction.java	org/apache/roller/ui/authoring/struts/actions/
9	BookmarksForm.java	org/apache/roller/ui/authoring/struts/formbeans/
10	CacheHttpServletResponseWrapper.java	org/apache/roller/ui/core/util/
11	CacheManager.java	org/apache/roller/util/cache/
12	CalendarTag.java	org/apache/roller/ui/core/tags/calendar/
13	CategoriesAction.java	org/apache/roller/ui/authoring/struts/actions/
14	CategoryDeleteForm.java	org/apache/roller/ui/authoring/struts/formbeans/
15	CommentAuthenticatorServlet.java	org/apache/roller/ui/rendering/servlets/
16	CommentForm.java	org/apache/roller/ui/authoring/struts/forms/
17	CommentManagementForm.java	org/apache/roller/ui/authoring/struts/formbeans/
18	CommentServlet.java	org/apache/roller/ui/rendering/servlets/
19	ConsistencyCheck.java	org/apache/roller/business/utills/
21	CreateWebsiteForm.java	org/apache/roller/ui/authoring/struts/formbeans/
22	CustomUserRegistry.java	org/apache/roller/ui/core/security/
23	DebugFilter.java	org/apache/roller/ui/core/filters/
24	DiskFeedInfoCache.java	org/apache/roller/util/rome/
25	dstest.jsp	/roller-ui/tools/
26	ExpiringLRUCacheImpl.java	org/apache/roller/util/cache/
27	FeedServlet.java	org/apache/roller/ui/rendering/servlets/
28	FileManagerImpl.java	org/apache/roller/business/
29	FolderForm.java	org/apache/roller/ui/authoring/struts/forms/
30	FolderForm.jsp	/WEB-INF/jsp/authoring/
31	footer.jsp	/WEB-INF/jsp/tiles/
32	FuturePostingsInvalidationJob.java	org/apache/roller/util/cache/
33	head.jsp	/WEB-INF/jsp/tiles/
34	HibernatePersistenceStrategy.java	org/apache/roller/business/hibernate/
35	HibernatePlanetManagerImpl.java	org/apache/roller/business/hibernate/
36	HibernateRefererManagerImpl.java	org/apache/roller/business/hibernate/

Id	Filename	Path
	ava	
37	HibernateRollerPlanetManagerImpl.java	org/apache/roller/business/hibernate/
38	HibernateUserManagerImpl.java	org/apache/roller/business/hibernate/
39	HibernateWeblogManagerImpl.java	org/apache/roller/business/hibernate/
40	ImportBookmarksFormAction.java	org/apache/roller/ui/authoring/struts/actions/
41	ImportEntriesAction.java	org/apache/roller/ui/authoring/struts/actions/
42	ImportEntriesForm.java	org/apache/roller/ui/authoring/struts/formbeans/
43	InitFilter.java	org/apache/roller/ui/core/filters/
44	InvitationsForm.java	org/apache/roller/ui/authoring/struts/formbeans/
45	InviteMemberForm.java	org/apache/roller/ui/authoring/struts/formbeans/
46	LinkbackExtractor.java	org/apache/roller/util/
47	LinkTag.java	org/apache/roller/ui/core/tags/
48	login-redirect.jsp	/roller-ui/
49	logout-redirect.jsp	/roller-ui/
50	MailUtil.java	org/apache/roller/util/
51	MaintenanceForm.java	org/apache/roller/ui/authoring/struts/formbeans/
52	MathCommentAuthenticator.java	org/apache/roller/ui/rendering/util/
53	MemberPermissionsForm.java	org/apache/roller/ui/authoring/struts/formbeans/
54	ModDateHeaderUtil.java	org/apache/roller/ui/rendering/util/
55	ModelLoader.java	org/apache/roller/ui/rendering/model/
56	ObjectAuditForm.java	org/apache/roller/ui/authoring/struts/forms/
57	OldCommentsRequest.java	org/apache/roller/ui/rendering/velocity/deprecated/
58	OldFeedRequest.java	org/apache/roller/ui/rendering/velocity/deprecated/
59	OldPageRequest.java	org/apache/roller/ui/rendering/velocity/deprecated/
60	OldRollerConfig.java	org/apache/roller/util/
62	PageServlet.java	org/apache/roller/ui/rendering/servlets/
63	PasswordUtility.java	org/apache/roller/business/utills/
64	PermissionsForm.java	org/apache/roller/ui/authoring/struts/forms/
65	PingSetupForm.java	org/apache/roller/ui/authoring/struts/formbeans/
66	PingTargetForm.java	org/apache/roller/ui/authoring/struts/forms/
67	PlanetConfigForm.java	org/apache/roller/ui/authoring/struts/forms/
68	PlanetGroupForm.java	org/apache/roller/ui/authoring/struts/forms/
69	PlanetRequest.java	org/apache/roller/ui/rendering/util/
70	PlanetSubscriptionForm.java	org/apache/roller/ui/authoring/struts/forms/
71	PreviewServlet.java	org/apache/roller/ui/rendering/servlets/
72	RebuildWebsiteIndexOperation.java	org/apache/roller/business/search/operations/

Id	Filename	Path
73	RedirectServlet.java	org/apache/roller/ui/rendering/velocity/deprecated/
74	RefererForm.java	org/apache/roller/ui/authoring/struts/forms/
75	ReferrerProcessingJob.java	org/apache/roller/business/referrers/
76	ReferrerQueueManagerImpl.java	org/apache/roller/business/referrers/
77	RefreshEntriesTask.java	org/apache/roller/ui/core/tasks/
78	RemoveWebsiteIndexOperation.java	org/apache/roller/business/search/operations/
79	RequestUtil.java	org/apache/roller/ui/core/util/
80	ResourceServlet.java	org/apache/roller/ui/rendering/servlets/
81	RoleForm.java	org/apache/roller/ui/authoring/struts/forms/
82	RollerAtomHandler.java	org/apache/roller/webservices/atomprotocol/
83	RollerConfigForm.java	org/apache/roller/ui/authoring/struts/forms/
84	RollerRequest.java	org/apache/roller/ui/core/
85	RollerRuntimeConfig.java	org/apache/roller/config/
86	RollerXMLRPCServlet.java	org/apache/roller/webservices/xmlrpc/
87	RssBadgeTag.java	org/apache/roller/ui/authoring/tags/
88	SchemeEnforcementFilter.java	org/apache/roller/ui/core/filters/
89	ShowEntrySummaryTag.java	org/apache/roller/ui/authoring/tags/
90	SiteWideCache.java	org/apache/roller/ui/rendering/util/
91	SyncWebsitesTask.java	org/apache/roller/ui/core/tasks/
92	TechnoratiRankingsTask.java	org/apache/roller/ui/core/tasks/
93	ThemeEditorAction.java	org/apache/roller/ui/authoring/struts/actions/
94	ThemeEditorForm.java	org/apache/roller/ui/authoring/struts/formbeans/
95	ThemeManagerImpl.java	org/apache/roller/business/
96	tiles-mainmenupage.jsp	/WEB-INF/jsp/tiles/
97	tiles-simplepage.jsp	/WEB-INF/jsp/tiles/
98	tiles-tabbedpage.jsp	/WEB-INF/jsp/tiles/
99	TrackbackForm.java	org/apache/roller/ui/authoring/struts/formbeans/
100	TrackbackServlet.java	org/apache/roller/ui/rendering/servlets/
101	TurnoverReferersTask.java	org/apache/roller/ui/core/tasks/
102	UpgradeDatabase.java	org/apache/roller/business/utills/
103	UploadFile.jsp	/WEB-INF/jsp/authoring/
104	UploadFileForm.java	org/apache/roller/ui/authoring/struts/formbeans/
105	UploadFileFormAction.java	org/apache/roller/ui/authoring/struts/actions/
106	UserDataServlet.java	org/apache/roller/ui/authoring/ajax/
107	UserForm.java	org/apache/roller/ui/authoring/struts/forms/
108	UserNewAction.java	org/apache/roller/ui/core/struts/actions/
109	Utilities.java	org/apache/roller/util/
110	VelocityRenderder.java	org/apache/roller/ui/rendering/velocity/
111	WeblogCategoryForm.java	org/apache/roller/ui/authoring/struts/forms/

Id	Filename	Path
112	WeblogCommentRequest.java	org/apache/roller/ui/rendering/util/
113	WeblogEdit.jsp	/WEB-INF/jsp/authoring/
114	WeblogEntryForm.java	org/apache/roller/ui/authoring/struts/forms/
115	WeblogEntryFormAction.java	org/apache/roller/ui/authoring/struts/actions/
116	WeblogEntryManagementForm.java	org/apache/roller/ui/authoring/struts/formbeans/
117	WeblogEntryRemove.jsp	/WEB-INF/jsp/authoring/
118	WeblogFeedCache.java	org/apache/roller/ui/rendering/util/
119	WeblogFeedRequest.java	org/apache/roller/ui/rendering/util/
120	WeblogPageCache.java	org/apache/roller/ui/rendering/util/
121	WeblogPageRequest.java	org/apache/roller/ui/rendering/util/
122	WeblogPreviewRequest.java	org/apache/roller/ui/rendering/util/
123	WeblogRequest.java	org/apache/roller/ui/rendering/util/
124	WeblogRequestMapper.java	org/apache/roller/ui/rendering/
126	WeblogTemplate.java	org/apache/roller/pojos/
127	WeblogTemplateForm.java	org/apache/roller/ui/authoring/struts/forms/
128	WeblogTrackbackRequest.java	org/apache/roller/ui/rendering/util/
129	WeblogUpdatePinger.java	org/apache/roller/ui/core/pings/
130	WebsiteData.java	org/apache/roller/pojos/
131	WebsiteDisplayForm.java	org/apache/roller/ui/authoring/struts/forms/
132	WebsiteForm.java	org/apache/roller/ui/authoring/struts/forms/
133	WebsiteFormAction.java	org/apache/roller/ui/authoring/struts/actions/
134	XSLTransform.java	org/apache/roller/util/
135	YourWebsitesForm.java	org/apache/roller/ui/authoring/struts/formbeans/