



Veracode Detailed Report
Application Security Report
for Example Company

July 28, 2011

Application:	WebGoat
Application Version:	5.0 - Java
Application Origin:	Open Source
Industry:	Other
Business Criticality:	BC5 (Very High)
Required Analysis:	Static, Dynamic and Manual
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/webgoat

Inside This Report

Executive Summary	1
Summary of Flaws by Severity	1
Action Items	1
Flaw Types by Category	3
Policy Summary	5
Detailed List of Flaws by Severity	6
Methodology	27

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 Detailed Report Application Security Report for Example Company

Veracode Level: VL1

Rated: Jul 28, 2011

Application Assessed: **WebGoat**

Business Criticality: **Very High**
Target Level: **VL4**

Application Version: **5.0 - Java**
Published Rating: **DD**

Executive Summary

This report contains a summary of the security flaws identified in the application using automated static, automated dynamic and/or manual 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)

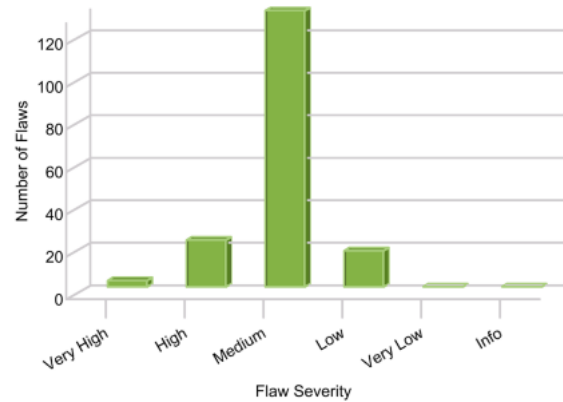
Impacts:Operational Risk (High), Financial Loss (High)

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
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.

Required Analysis

- Your policy requires periodic Static Analysis. Your next analysis must be completed by 10/28/11. Please submit your application for Static Analysis by the deadline and remediate the required detected flaws to conform to your assigned policy.

Flaws To Fix By Expires Date

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

- The grace period has expired [7/28/11] for 143 flaws that were found in your Static Analysis.
- The grace period has expired [7/28/11] for 12 flaws that were found in your Dynamic Analysis.

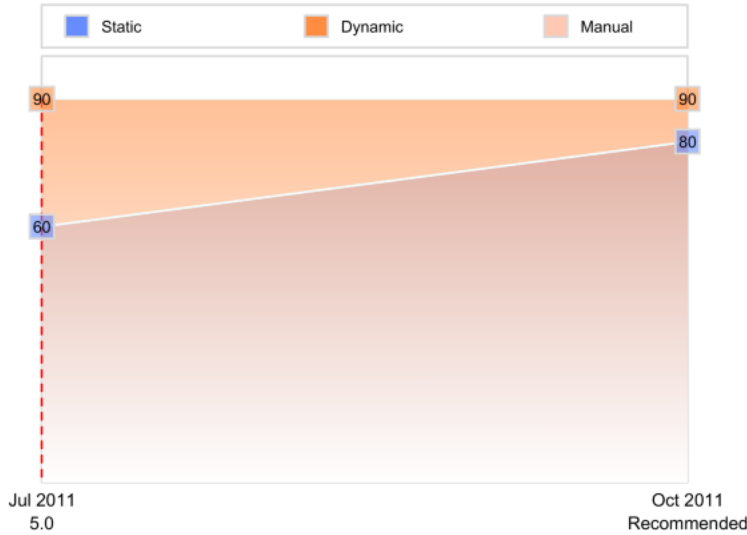
Flaws To Fix For Minimum Score

- Your current policy requires a minimum score. In order to achieve the score, you must fix all of the flaws that violate your current policy plus additional flaws. You must fix 2 Very High flaws, 21 High flaws and 50 Medium flaws to increase the application Static Analysis Security Quality Score to 80.
- Your Static Analysis was due on 7/28/11 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 Analysis once flaws have been remediated in order to regain compliance with your policy.

Longer Timeframe (6 - 12 months)

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

Application Ratings Trend



- In Jul 2011, Veracode analyzed, scored and rated WebGoat 5.0 - Java product. The security quality scores for the two scan types were as follows: Static Scan 60; and Dynamic Scan 90.
- The Action Items outline what Example Company can do to achieve compliance by Oct 2011.

Scope of Analysis (Static)

The following modules were included in the application scan:

Module Name	Compiler	Operating Environment
WebGoat-5.0-with-jsp.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/webgoat
Restrict to Directory	true
Number of Links Visited	644
Logged In Successfully	N/A
Login User ID	
Scan Began	Feb 8, 2010 9:38:01 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.

Flaw Types by Severity and Category

	Static Analysis Security Quality Score = 60	Dynamic Analysis Security Quality Score = 90	
Very High	2	1	
OS Command Injection	2	1	
High	21	1	
SQL Injection	21	1	
	120	10	

	Static Analysis Security Quality Score = 60	Dynamic Analysis Security Quality Score = 90	
Medium			
CRLF Injection	6		
Code Quality	4		
Credentials Management	2		
Cross-Site Scripting	102	10	
Cryptographic Issues	1		
Directory Traversal	3		
Encapsulation	1		
Race Conditions	1		
Low	17	0	
API Abuse	5		
Code Quality	4		
Cryptographic Issues	5		
Information Leakage	3		
Very Low	0	0	
Informational	0	0	
Total	160	12	

Policy Control

Policy Name: Veracode Recommended High

Revision: 1

Policy Status: Did Not Pass

Description

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

Rules

Rule type	Requirement	Findings	Status
Minimum Veracode Level	VL4	VL1	Did not pass
(VL4) Min Analysis Score	80	60	Did not pass
(VL4) Max Severity	Medium	Flaws found: 155	Did not pass

Scan Requirements

Scan Type	Frequency	Last performed	Status
Static	Quarterly	7/28/11	Passed

Remediation

Flaw Severity	Grace Period	Flaws Exceeding	Status
Very High	0 days	3	Did not pass
High	0 days	22	Did not pass
Medium	0 days	130	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

Detailed Flaws by Severity

Very High (3 flaws)

 **Fix Required by Policy**

→ OS Command Injection(3 flaws)

 **Fix Required by Policy**

Description

OS command injection vulnerabilities occur when data enters an application from an untrusted source and is used to dynamically construct and execute a system command. This allows an attacker to either alter the command executed by the application or append additional commands. The command is typically executed with the privileges of the executing process and gives an attacker a privilege or capability that he would not otherwise have.

Recommendations

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

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

Associated Software Flaw Types:

→ Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') (CWE ID 78)(3 flaws)

Description

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

Effort/Complexity of Fix: 3

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. Most APIs that execute system commands also have a "safe" version of the method that takes an array of strings as input rather than a single string, which protects against some forms of command injection.

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
14	-	WebGoat-5.0-with-jsp.war	org/.../webgoat/util/Exec.java 103	7/28/11	149
14	-	WebGoat-5.0-with-jsp.war	org/.../webgoat/util/Exec.java 292	7/28/11	143

Attack Vectors

URL	Parameter	Exploitability	Flaw Id
http://10.0.4.89:8080/WebGoat/attack	File	-	167

High (22 flaws)

 **Fix Required by Policy**

→ SQL Injection(22 flaws)

 **Fix Required by Policy**

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)(22 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

Module #	Class #	Module	Location	Fix By	Flaw Id
1	-	WebGoat-5.0-with-jsp.war	org/.../lessons/BackDoors.java 106	7/28/11	139
1	-	WebGoat-5.0-with-jsp.war	org/.../lessons/BackDoors.java 113	7/28/11	42
2	-	WebGoat-5.0-with-jsp.war	org/.../BlindSqlInjection.java 122	7/28/11	108
3	-	WebGoat-5.0-with-jsp.war	org/.../Challenge2Screen.java 220	7/28/11	11
9	-	WebGoat-5.0-with-jsp.war	org/.../lessons/DOS_Login.java 114	7/28/11	105
9	-	WebGoat-5.0-with-jsp.war	org/.../lessons/DOS_Login.java 134	7/28/11	73
22	-	WebGoat-5.0-with-jsp.war	org/.../Login.java 148	7/28/11	128
23	-	WebGoat-5.0-with-jsp.war	org/.../SQLInjection/Login.java 149	7/28/11	109
23	-	WebGoat-5.0-with-jsp.war	org/.../SQLInjection/Login.java 191	7/28/11	54
34	-	WebGoat-5.0-with-jsp.war	.../SqlNumericInjection.java 130	7/28/11	117
35	-	WebGoat-5.0-with-jsp.war	.../SqlStringInjection.java 112	7/28/11	48
36	-	WebGoat-5.0-with-jsp.war	.../ThreadSafetyProblem.java 103	7/28/11	94
38	-	WebGoat-5.0-with-jsp.war	org/.../UpdateProfile.java 176	7/28/11	23
37	-	WebGoat-5.0-with-jsp.war	org/.../UpdateProfile.java 248	7/28/11	154
38	-	WebGoat-5.0-with-jsp.war	org/.../UpdateProfile.java 295	7/28/11	112
37	-	WebGoat-5.0-with-jsp.war	org/.../UpdateProfile.java 340	7/28/11	106
39	-	WebGoat-5.0-with-jsp.war	org/.../UpdateProfile_i.java 64	7/28/11	65
40	-	WebGoat-5.0-with-jsp.war	org/.../admin/ViewDatabase.java 89	7/28/11	142
41	-	WebGoat-5.0-with-jsp.war	org/.../ViewProfile.java 118	7/28/11	79
41	-	WebGoat-5.0-with-jsp.war	org/.../ViewProfile.java 178	7/28/11	46
49	-	WebGoat-5.0-with-jsp.war	org/.../WsSqlInjection.java 240	7/28/11	153

Attack Vectors

URL	Parameter	Exploitability	Flaw Id
http://10.0.4.89:8080/WebGoat/attack	id	-	166

Medium (130 flaws)

 **Fix Required by Policy**

→ CRLF Injection(6 flaws)

 **Fix Required by Policy**

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 Software Flaw Types:

→ Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting') (CWE ID 113)(5 flaws)

Description

A function call contains an HTTP response splitting flaw. Writing unsanitized user-supplied 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/Complexity of Fix: 2

Recommendations

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

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
5	-	WebGoat-5.0-with-jsp.war	com/.../config_jsp.java 43	7/28/11	74
16	-	WebGoat-5.0-with-jsp.war	org/.../lessons/HttpOnly.java 198	7/28/11	157
16	-	WebGoat-5.0-with-jsp.war	org/.../lessons/HttpOnly.java 212	7/28/11	87
29	-	WebGoat-5.0-with-jsp.war	com/.../General/redirect_jsp.java 43	7/28/11	21
48	-	WebGoat-5.0-with-	org/.../session/WebSession.java 335	7/28/11	131

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

→ **Improper Output Neutralization for Logs (CWE ID 117)(1 flaw)**

Description

A function call could result in a log forging attack. Writing unsanitized user-supplied 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/Complexity of Fix: 2

Recommendations

Avoid directly embedding user input in log files when possible. Sanitize user-supplied data used to construct log entries by removing unexpected carriage returns and line feeds and using HTML entities to encode non-alphanumeric data. Always validate user-supplied input to ensure that it conforms to the expected format, using centralized data validation routines when possible.

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
15	-	WebGoat-5.0-with-jsp.war	org/.../webgoat/HammerHead.java 306	7/28/11	27

→ **Code Quality(4 flaws)**

 **Fix Required by Policy**

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:

→ Leftover Debug Code (CWE ID 489)(4 flaws)

Description

A method may be leftover debug code that creates an unintended entry point in a web application. Although this is an acceptable practice during product development, classes that are part of a production J2EE application should not define a main() method. Whether this method can be remotely invoked depends on the configuration of the J2EE container and the application itself.

Effort/Complexity of Fix: 2

Recommendations

Remove debug code prior to deploying the application. Eliminate unnecessary entry points in deployed web applications to reduce the attack surface.

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
6	-	WebGoat-5.0-with-jsp.war	org/.../session/CreateDB.java 53	7/28/11	99
13	-	WebGoat-5.0-with-jsp.war	org/.../lessons/Encoding.java 744	7/28/11	158
14	-	WebGoat-5.0-with-jsp.war	org/.../webgoat/util/Exec.java 505	7/28/11	22
47	-	WebGoat-5.0-with-jsp.war	org/.../WebgoatProperties.java 114	7/28/11	9

→ Credentials Management(2 flaws)

 **Fix Required by Policy**

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

Module #	Class #	Module	Location	Fix By	Flaw Id
7	-	WebGoat-5.0-with-jsp.war	org/.../DatabaseUtilities.java 97	7/28/11	50

→ Use of Hard-coded Password (CWE ID 259)(1 flaw)

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

Module #	Class #	Module	Location	Fix By	Flaw Id
6	-	WebGoat-5.0-with-jsp.war	org/.../session/CreateDB.java 69	7/28/11	84

→ Cross-Site Scripting(112 flaws)

 **Fix Required by Policy**

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)(10 flaws)

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

URL	Parameter	Exploitability	Flaw Id
http://10.0.4.89:8080/WebGoat/attack		-	168
http://10.0.4.89:8080/WebGoat/css/menu.css		-	169
http://10.0.4.89:8080/WebGoat/images/logos/owasp.jpg		-	170
http://10.0.4.89:8080/WebGoat/javascript/javascript.js		-	171
http://10.0.4.89:8080/WebGoat/lesson_solutions/CommandInjection		-	172

URL	Parameter	Exploitability	Flaw Id
_files/image005.png			
http://10.0.4.89:8080/WebGoat/lesson_solutions/formate.css		-	175
http://10.0.4.89:8080/WebGoat/lesson_solutions/HttpSplitting_files/filelist.xml		-	173
http://10.0.4.89:8080/WebGoat/lesson_solutions/UncheckedEmail_files/image001.png		-	174
http://10.0.4.89:8080/WebGoat/lessons/General/redirect.jsp		-	176
http://10.0.4.89:8080/WebGoat/lessons/RoleBasedAccessControl/images/orgChart.jpg		-	177

→ **Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS) (CWE ID 80)(102 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

Module #	Class #	Module	Location	Fix By	Flaw Id
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 52	7/28/11	119
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 52	7/28/11	129
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 52	7/28/11	59
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 58	7/28/11	96
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 58	7/28/11	58
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 58	7/28/11	144
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 62	7/28/11	33
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 62	7/28/11	67

Module #	Class #	Module	Location	Fix By	Flaw Id
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 62	7/28/11	159
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 66	7/28/11	122
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 66	7/28/11	102
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 66	7/28/11	64
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 70	7/28/11	88
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 70	7/28/11	91
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 70	7/28/11	152
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 74	7/28/11	32
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 74	7/28/11	49
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 74	7/28/11	116
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 78	7/28/11	75
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 78	7/28/11	93
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 78	7/28/11	69
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 82	7/28/11	4
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 82	7/28/11	25
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 82	7/28/11	160
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 90	7/28/11	125
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 90	7/28/11	95
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 90	7/28/11	104
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 98	7/28/11	163
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 98	7/28/11	126
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 98	7/28/11	24
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 102	7/28/11	165
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 106	7/28/11	132
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 113	7/28/11	164
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 113	7/28/11	34

Module #	Class #	Module	Location	Fix By	Flaw Id
		jsp.war			
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 119	7/28/11	12
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 119	7/28/11	68
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 121	7/28/11	30
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 123	7/28/11	107
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 123	7/28/11	110
10	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 133	7/28/11	124
12	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 133	7/28/11	8
11	-	WebGoat-5.0-with-jsp.war	com/.../EditProfile_jsp.java 133	7/28/11	29
17	-	WebGoat-5.0-with-jsp.war	org/.../lessons/HttpSplitting.java 112	7/28/11	60
19	-	WebGoat-5.0-with-jsp.war	com/.../ListStaff_jsp.java 52	7/28/11	92
20	-	WebGoat-5.0-with-jsp.war	com/.../ListStaff_jsp.java 52	7/28/11	137
21	-	WebGoat-5.0-with-jsp.war	com/.../ListStaff_jsp.java 52	7/28/11	150
21	-	WebGoat-5.0-with-jsp.war	com/.../ListStaff_jsp.java 68	7/28/11	97
20	-	WebGoat-5.0-with-jsp.war	com/.../ListStaff_jsp.java 68	7/28/11	147
19	-	WebGoat-5.0-with-jsp.war	com/.../ListStaff_jsp.java 68	7/28/11	114
24	-	WebGoat-5.0-with-jsp.war	com/.../Login_jsp.java 71	7/28/11	7
25	-	WebGoat-5.0-with-jsp.war	com/.../Login_jsp.java 71	7/28/11	141
26	-	WebGoat-5.0-with-jsp.war	com/.../Login_jsp.java 71	7/28/11	111
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 170	7/28/11	70
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 172	7/28/11	66
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 174	7/28/11	133
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 178	7/28/11	83
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 180	7/28/11	89
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 194	7/28/11	20
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 212	7/28/11	47

Module #	Class #	Module	Location	Fix By	Flaw Id
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 223	7/28/11	5
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 231	7/28/11	6
30	-	WebGoat-5.0-with-jsp.war	com/.../SearchStaff_jsp.java 52	7/28/11	39
31	-	WebGoat-5.0-with-jsp.war	com/.../SearchStaff_jsp.java 52	7/28/11	136
32	-	WebGoat-5.0-with-jsp.war	com/.../SearchStaff_jsp.java 52	7/28/11	15
33	-	WebGoat-5.0-with-jsp.war	.../SilentTransactions.java 94	7/28/11	41
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 51	7/28/11	81
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 52	7/28/11	121
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 52	7/28/11	55
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 53	7/28/11	101
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 54	7/28/11	130
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 54	7/28/11	145
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 55	7/28/11	118
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 56	7/28/11	36
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 56	7/28/11	26
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 57	7/28/11	113
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 58	7/28/11	78
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 58	7/28/11	134
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 59	7/28/11	61
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 60	7/28/11	82
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 60	7/28/11	115
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 61	7/28/11	56
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 62	7/28/11	40
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 62	7/28/11	85
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 63	7/28/11	151

Module #	Class #	Module	Location	Fix By	Flaw Id
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 64	7/28/11	127
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 64	7/28/11	10
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 65	7/28/11	71
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 66	7/28/11	120
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 66	7/28/11	123
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 69	7/28/11	2
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 70	7/28/11	37
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 70	7/28/11	45
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 73	7/28/11	103
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 74	7/28/11	162
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 74	7/28/11	14
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 76	7/28/11	57
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 77	7/28/11	31
43	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 78	7/28/11	35
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 78	7/28/11	16
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 79	7/28/11	76
44	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 80	7/28/11	51
42	-	WebGoat-5.0-with-jsp.war	com/.../ViewProfile_jsp.java 143	7/28/11	135

→ **Cryptographic Issues(1 flaw)**

 **Fix Required by Policy**

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)(1 flaw)

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

Module #	Class #	Module	Location	Fix By	Flaw Id
46	-	WebGoat-5.0-with-jsp.war	org/.../lessons/WeakSessionID.java 77	7/28/11	156

→ Directory Traversal(3 flaws)

 **Fix Required by Policy**

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)(3 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

Module #	Class #	Module	Location	Fix By	Flaw Id
4	-	WebGoat-5.0-with-jsp.war	org/.../CommandInjection.java 171	7/28/11	52
4	-	WebGoat-5.0-with-jsp.war	org/.../CommandInjection.java 180	7/28/11	28
28	-	WebGoat-5.0-with-jsp.war	.../PathBasedAccessControl.java 136	7/28/11	17

→ Encapsulation(1 flaw)

 **Fix Required by Policy**

Description

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

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

Recommendations

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

Associated Software Flaw Types:

→ Trust Boundary Violation (CWE ID 501)(1 flaw)

Description

A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted. This application mixes trusted and untrusted data in the same data structure. By doing so, it becomes easier for programmers to mistakenly trust unvalidated data. Without well-established and maintained trust boundaries, programmers will inevitably lose track of which pieces of data have been validated and which have not. This confusion will eventually allow some data to be used without first being validated. A common manifestation of this flaw is in J2EE application, when a Session object is used to store untrusted data from the HTTP request.

Effort/Complexity of Fix: 2

Recommendations

Avoid storing untrusted data alongside trusted data in the same data structure. Establish and maintain trust boundaries to avoid losing track of which pieces of data have been validated and which have not.

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
8	-	WebGoat-5.0-with-jsp.war	.../DefaultLessonAction.java 89	7/28/11	161

→ Race Conditions(1 flaw)

 **Fix Required by Policy**

Description

Race conditions are the most well-known timing flaw and are related to improper handling of a resource that can be accessed by multiple processes. Race conditions exploit the small window of time between when a security control is applied and when the service is used. Vulnerabilities occur when there is a discrepancy between the programmer's assumption of how a program executes and what happens in reality.

Attackers often exploit race conditions to gain unauthorized access to system resources. A common scenario is a time-of-check time-of-use race condition, which takes advantage of a timing gap between the time the authorization check is performed and the time the resource is actually used.

Recommendations

Review the code carefully for any pair of operations that might fail if arbitrary code is executed between them. Often, developers assume that a series of actions represent an atomic operation when in fact they do not. These non-atomic conditions commonly involve the filesystem and/or temporary files.

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:

→ **Race Condition within a Thread (CWE ID 366)(1 flaw)**

Description

If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.

Effort/Complexity of Fix: 2

Recommendations

Implement some form of synchronization or locking mechanism around code which alters or reads persistent data to ensure that the operation is thread-safe.

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
15	-	WebGoat-5.0-with-jsp.war	org/.../webgoat/HammerHead.java 135	7/28/11	72

Low (17 flaws)

→ **API Abuse(5 flaws)**

Description

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

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

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

Recommendations

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

Associated Software Flaw Types:

→ **J2EE Bad Practices: Direct Management of Connections (CWE ID 245)(5 flaws)**

Description

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

Effort/Complexity of Fix: 2

Recommendations

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

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
6	-	WebGoat-5.0-with-jsp.war	org/.../session/CreateDB.java 69		146
7	-	WebGoat-5.0-with-jsp.war	org/.../DatabaseUtilities.java 68		63
7	-	WebGoat-5.0-with-jsp.war	org/.../DatabaseUtilities.java 78		38
7	-	WebGoat-5.0-with-jsp.war	org/.../DatabaseUtilities.java 97		18
7	-	WebGoat-5.0-with-jsp.war	org/.../DatabaseUtilities.java 105		1

→ Code Quality(4 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 Software Flaw Types:

→ Use of Wrong Operator in String Comparison (CWE ID 597)(4 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/Complexity of Fix: 1

Recommendations

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

Instances

Module #	Class #	Module	Location	Fix By	Flaw Id
45	-	WebGoat-5.0-with-jsp.war	.../WeakAuthenticationCookie.java 142		90
50	-	WebGoat-5.0-with-jsp.war	org/.../lessons/XMLInjection.java 280		62
50	-	WebGoat-5.0-with-jsp.war	org/.../lessons/XMLInjection.java 282		148
50	-	WebGoat-5.0-with-jsp.war	org/.../lessons/XMLInjection.java 292		44

→ Cryptographic Issues(5 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)(5 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

Module #	Class #	Module	Location	Fix By	Flaw Id
3	-	WebGoat-5.0-with-jsp.war	org/.../Challenge2Screen.java 172		98
3	-	WebGoat-5.0-with-jsp.war	org/.../Challenge2Screen.java 193		53
45	-	WebGoat-5.0-with-jsp.war	.../WeakAuthenticationCookie.java 146		43
46	-	WebGoat-5.0-with-jsp.war	org/.../lessons/WeakSessionID.java 209		138
48	-	WebGoat-5.0-with-jsp.war	org/.../session/WebSession.java 335		86

→ 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 Software Flaw Types:

→ Information Exposure Through an Error Message (CWE ID 209)(3 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

Module #	Class #	Module	Location	Fix By	Flaw Id
18	-	WebGoat-5.0-with-jsp.war	org/.../lessons/JSONInjection.java 90		3
27	-	WebGoat-5.0-with-jsp.war	com/.../main_jsp.java 227		140
50	-	WebGoat-5.0-with-jsp.war	org/.../lessons/XMLInjection.java 119		100

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 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: Referenced Source Files

Id	Filename	Path
1	BackDoors.java	org/owasp/webgoat/lessons/
2	BlindSqlInjection.java	org/owasp/webgoat/lessons/
3	Challenge2Screen.java	org/owasp/webgoat/lessons/
4	CommandInjection.java	org/owasp/webgoat/lessons/
5	config_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/ConfManagement/
6	CreateDB.java	org/owasp/webgoat/session/
7	DatabaseUtilities.java	org/owasp/webgoat/session/
8	DefaultLessonAction.java	org/owasp/webgoat/lessons/
9	DOS_Login.java	org/owasp/webgoat/lessons/
10	EditProfile_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/SQLInjection/
11	EditProfile_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/RoleBasedAccessControl/
12	EditProfile_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/CrossSiteScripting/
13	Encoding.java	org/owasp/webgoat/lessons/
14	Exec.java	org/owasp/webgoat/util/
15	HammerHead.java	org/owasp/webgoat/
16	HttpOnly.java	org/owasp/webgoat/lessons/
17	HttpSplitting.java	org/owasp/webgoat/lessons/
18	JSONInjection.java	org/owasp/webgoat/lessons/
19	ListStaff_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/RoleBasedAccessControl/
20	ListStaff_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/SQLInjection/
21	ListStaff_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/CrossSiteScripting/
22	Login.java	org/owasp/webgoat/lessons/RoleBasedAccessControl/
23	Login.java	org/owasp/webgoat/lessons/SQLInjection/
24	Login_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/SQLInjection/
25	Login_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/RoleBasedAccessControl/
26	Login_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/CrossSiteScripting/
27	main_jsp.java	com/veracode/compiledjsp/xWebGoat50war/
28	PathBasedAccessControl.java	org/owasp/webgoat/lessons/
29	redirect_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/General/
30	SearchStaff_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/RoleBasedAccessControl/
31	SearchStaff_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/SQLInjection/
32	SearchStaff_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/CrossSiteScripting/
33	SilentTransactions.java	org/owasp/webgoat/lessons/
34	SqlNumericInjection.java	org/owasp/webgoat/lessons/
35	SqlStringInjection.java	org/owasp/webgoat/lessons/
36	ThreadSafetyProblem.java	org/owasp/webgoat/lessons/
37	UpdateProfile.java	org/owasp/webgoat/lessons/CrossSiteScripting/
38	UpdateProfile.java	org/owasp/webgoat/lessons/RoleBasedAccessControl/

Id	Filename	Path
39	UpdateProfile_i.java	org/owasp/webgoat/lessons/instructor/RoleBasedAccessControl/
40	ViewDatabase.java	org/owasp/webgoat/lessons/admin/
41	ViewProfile.java	org/owasp/webgoat/lessons/SQLInjection/
42	ViewProfile_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/CrossSiteScripting/
43	ViewProfile_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/RoleBasedAccessControl/
44	ViewProfile_jsp.java	com/veracode/compiledjsp/xWebGoat50war/lessons/SQLInjection/
45	WeakAuthenticationCookie.java	org/owasp/webgoat/lessons/
46	WeakSessionID.java	org/owasp/webgoat/lessons/
47	WebgoatProperties.java	org/owasp/webgoat/session/
48	WebSession.java	org/owasp/webgoat/session/
49	WsSqlInjection.java	org/owasp/webgoat/lessons/
50	XMLInjection.java	org/owasp/webgoat/lessons/