Fortify Security Report

Jan 29, 2020

pgupta25

Executive Summary

Issues Overview

On Jan 29, 2020, a source code review was performed over the metadatasharing code base. 421 files, 12,014 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 68 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Critical	39
High	29

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: /srv/openmrs_code/org/openmrs/module/metadatasharing
Number of Files: 421
Lines of Code: 12014
Build Label: <No Build Label>

Scan Information

Scan time: 21:16
SCA Engine version: 19.1.0.2241
Machine Name: vclv99-89.hpc.ncsu.edu
Username running scan: pgupta25

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:
File System:
java.io.FileInputStream.FileInputStream
java.io.FileInputStream.FileInputStream
java.util.zip.ZipInputStream.getNextEntry

Private Information:
null.null.null
java.util.Properties.getProperty

Java Properties:
java.util.Properties.load

Stream:
java.io.BufferedInputStream.read

System Information:
null.null.null
java.lang.ClassLoader.getSystemResource
java.lang.ClassLoader.getSystemResourceAsStream

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

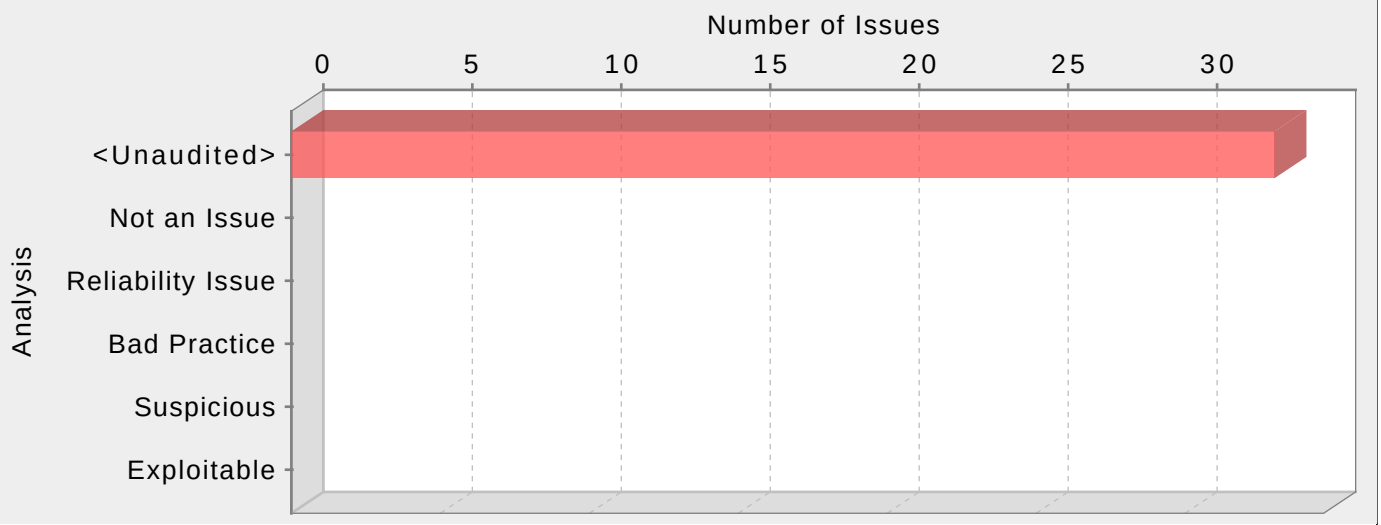
Results Outline

Overall number of results

The scan found 68 issues.

Vulnerability Examples by Category

Category: Cross-Site Scripting: DOM (33 Issues)



Abstract:

The method _fnDrawHead() in jquery.dataTables.min.js sends unvalidated data to a web browser on line 247, which 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 DOM-based XSS, data is read from a URL parameter or other value within the browser and written back into the page with client-side code. 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 data store.

2. The data is included in dynamic content that is sent to a web user without being validated. In the case of DOM Based XSS, malicious content gets executed as part of DOM (Document Object Model) creation, whenever the victim's browser parses the HTML page.

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 executes. 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 JavaScript code segment reads an employee ID, eid, from a URL and displays it to the user.

```
<SCRIPT>
var pos=document.URL.indexOf("eid=")+4;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
```

Example 2: Consider the HTML form:

```
<div id="myDiv">
Employee ID: <input type="text" id="eid"><br>
...
<button>Show results</button>
</div>
<div id="resultsDiv">
...
</div>
```

The following jQuery code segment reads an employee ID from the form, and displays it to the user.

```
$(document).ready(function(){
$("#myDiv").on("click", "button", function(){
var eid = $("#eid").val();
$("resultsDiv").append(eid);
...
});
});
```

These code examples operate correctly if the employee ID, from the text input with ID 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.

Example 3: The following code shows an example of a DOM-based XSS within a React application:

```
let element = JSON.parse(getUntrustedInput());
ReactDOM.render(<App>
{element}
</App>);
```

In Example 3, if an attacker can control the entire JSON object retrieved from getUntrustedInput(), they may be able to make React render element as a component, and therefore can pass an object with dangerouslySetInnerHTML with their own controlled value, a typical cross-site scripting attack.

Initially these might not appear to be much of a vulnerability. After all, why would someone provide input containing malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use email 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 example demonstrates, 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:

- 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 emailed 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.

- 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.

Recommendations:

The solution to XSS is to ensure that validation occurs in the correct places and checks are made 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 and 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, which is why we do not encourage the use of blacklists as a means to prevent XSS. 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 greater than 127 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>:

- Semicolons, parentheses, curly braces, and new line characters should be filtered out 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).

After 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 Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also 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. Nonetheless, it is often still a good idea to validate everything read from the database.

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. Older versions of React are more susceptible to cross-site scripting attacks by controlling an entire component. Newer versions use Symbols to identify a React component, which prevents the exploit, however older browsers that do not have Symbol support (natively, or through polyfills), such as all versions of Internet Explorer, are still vulnerable. Other types of cross-site scripting attacks are valid for all browsers and versions of React.

jquery.dataTables.min.js, line 247 (Cross-Site Scripting: DOM)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method _fnDrawHead() in jquery.dataTables.min.js sends unvalidated data to a web browser on line 247, which can result in the browser executing malicious code.		

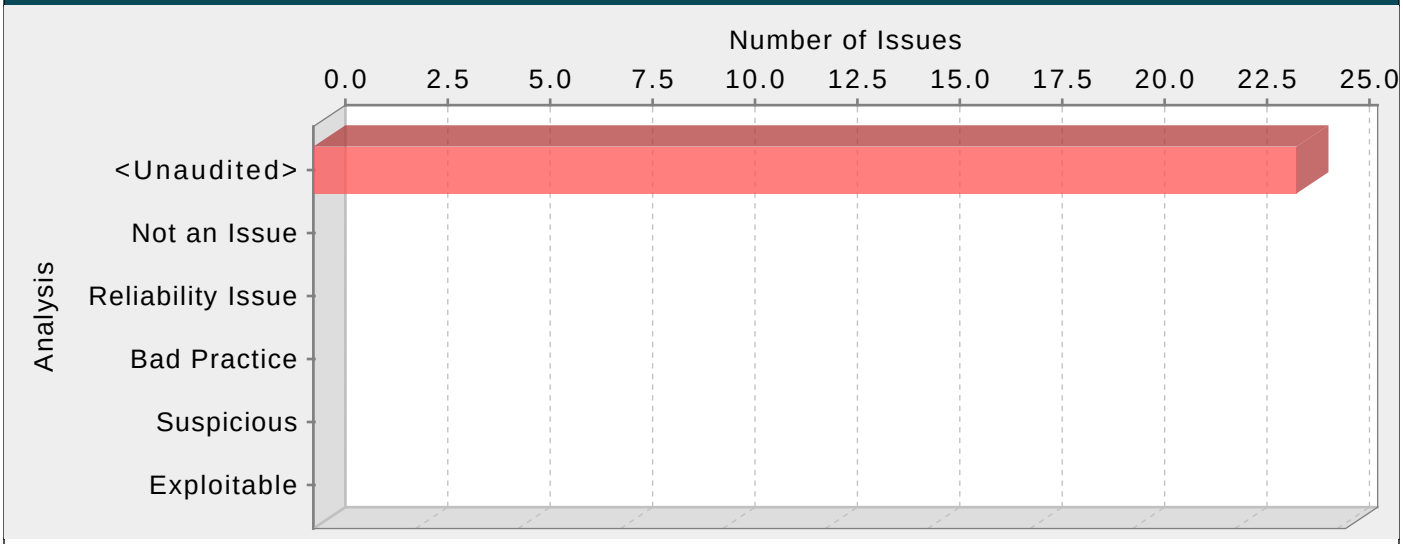
Source: jquery.dataTables.min.js:512 Read document.cookie()

```
510  sName+="_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase();document.cookie
     =sName+"="+encodeURIComponent(sValue)+"; expires="+date.toGMTString()+"; path=/"
511  }function _fnReadCookie(sName){var
     sNameEQ=sName+"_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase()+"=";
512  var sCookieContents=document.cookie.split(";");for(var i=0;i<sCookieContents.length;
513  i++){var c=sCookieContents[i];while(c.charAt(0)==""
     "){c=c.substring(1,c.length)}if(c.indexOf(sNameEQ)===0){return
     decodeURIComponent(c.substring(sNameEQ.length,c.length))
514  }}return null}function _fnGetUniqueThs(nThead){var
     nTrs=nThead.getElementsByTagName("tr");
```

Sink: jquery.dataTables.min.js:247 jQuery()

```
245  }if(oSettings.bJUI){for(i=0,iLen=oSettings.aoColumns.length;i<iLen;i++){oSettings.aoCo
     lumns[i].nTh.insertBefore(document.createElement("span"),oSettings.aoColumns[i].nTh.fi
     rstChild)
246  }}if(oSettings.oFeatures.bSort){for(i=0;i<oSettings.aoColumns.length;i++){if(oSettings
     .aoColumns[i].bSortable!==false){_fnSortAttachListener(oSettings,oSettings.aoColumns[i
     ].nTh,i)
247  }else{$(oSettings.aoColumns[i].nTh).addClass(oSettings.oClasses.sSortableNone)}}$("the
     ad:eq(0)
     th",oSettings.nTable).mousedown(function(e){if(e.shiftKey){this.onselectstart=function
     (){return false
248  };return false}})}var
     nTfoot=oSettings.nTable.getElementsByTagName("tfoot");if(nTfoot.length!==0){iCorrector
     =0;
249  var
     nTfs=nTfoot[0].getElementsByTagName("th");for(i=0,iLen=nTfs.length;i<iLen;i++){oSettin
     gs.aoColumns[i].nTf=nTfs[i-iCorrector];
```

Category: Privacy Violation (24 Issues)



Abstract:

The file jquery.dataTables.min.js mishandles confidential information on line 225, which 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 stores user's plain text password to the local storage.

localStorage.setItem('password', password);

Although many developers treat the local storage as a safe 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 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 email 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.

jquery.dataTables.min.js, line 225 (Privacy Violation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The file jquery.dataTables.min.js mishandles confidential information on line 225, which can compromise user privacy and is often illegal.		

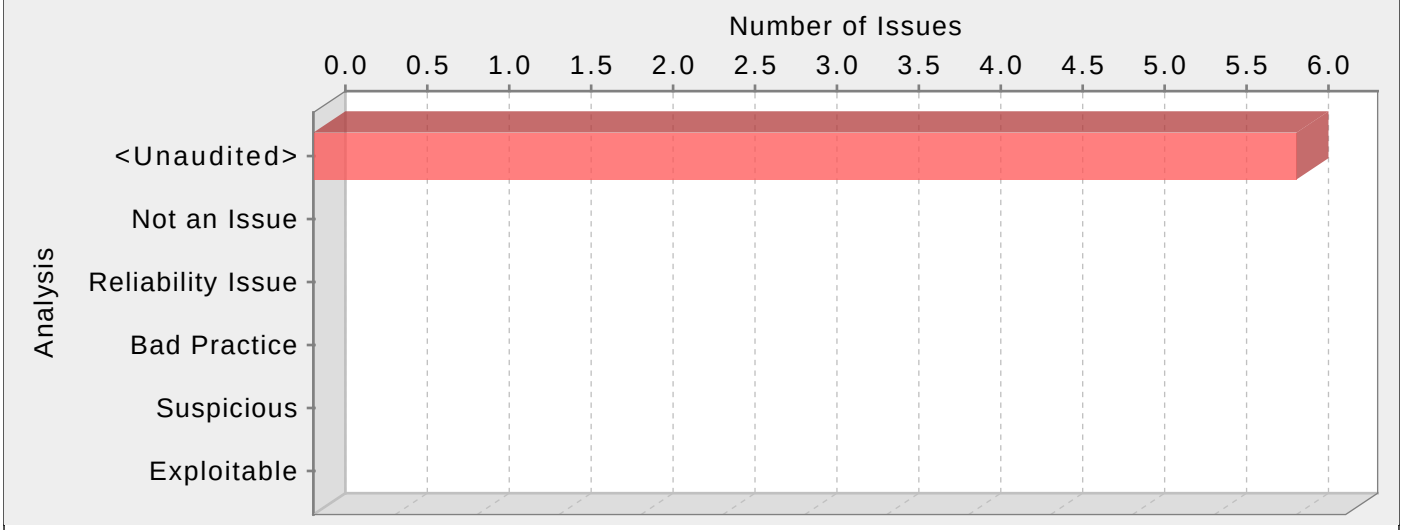
Source: jquery.dataTables.min.js:512 Read document.cookie()

```
510  sName+="_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase();document.cookie
     =sName+"="+encodeURIComponent(sValue)+"; expires="+date.toGMTString()+"; path=/"
511  }function _fnReadCookie(sName){var
     sNameEQ=sName+"_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase()+"=";
512  var sCookieContents=document.cookie.split(";");for(var i=0;i<sCookieContents.length;
513  i++){var c=sCookieContents[i];while(c.charAt(0)==
     "){c=c.substring(1,c.length)}if(c.indexOf(sNameEQ)===0){return
     decodeURIComponent(c.substring(sNameEQ.length,c.length))
514  }}return null}function _fnGetUniqueThs(nThead){var
     nTrs=nThead.getElementsByTagName("tr");
```

Sink: jquery.dataTables.min.js:225 alert()

```
223  jInner++}}}}}nTrs=_fnGetTrNodes(oSettings);nTds=[];for(i=0,iLen=nTrs.length;i<iLen;
224  i++){for(j=0,jLen=nTrs[i].childNodes.length;j<jLen;j++){nTd=nTrs[i].childNodes[j];
225  if(nTd.nodeName=="TD"){nTds.push(nTd)}}}if(nTds.length!=nTrs.length*oSettings.aoColumn
     s.length){alert("DataTables warning: Unexpected number of TD elements. Expected
     "+(nTrs.length*oSettings.aoColumns.length)+" and got "+nTds.length+". DataTables does
     not support rowspan / colspan in the table body, and there must be one cell for each
     row/column combination.")
226  }for(iColumn=0,iColumns=oSettings.aoColumns.length;iColumn<iColumns;iColumn++){if(oSet
     tings.aoColumns[iColumn].sTitle===null){oSettings.aoColumns[iColumn].sTitle=oSettings.
     aoColumns[iColumn].nTh.innerHTML
227  }var bAutoType=oSettings.aoColumns[iColumn]._bAutoType,bRender=typeof
     oSettings.aoColumns[iColumn].fnRender=="function",bClass=oSettings.aoColumns[iColumn].
     sClass!==null,bVisible=oSettings.aoColumns[iColumn].bVisible,nCell,sThisType,sRendered
     ;
```

Category: Dynamic Code Evaluation: Code Injection (6 Issues)



Abstract:

The file jquery.dataTables.min.js interprets unvalidated user input as source code on line 363. 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 may 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, use a level of indirection: 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.

jquery.dataTables.min.js, line 363 (Dynamic Code Evaluation: Code Injection)

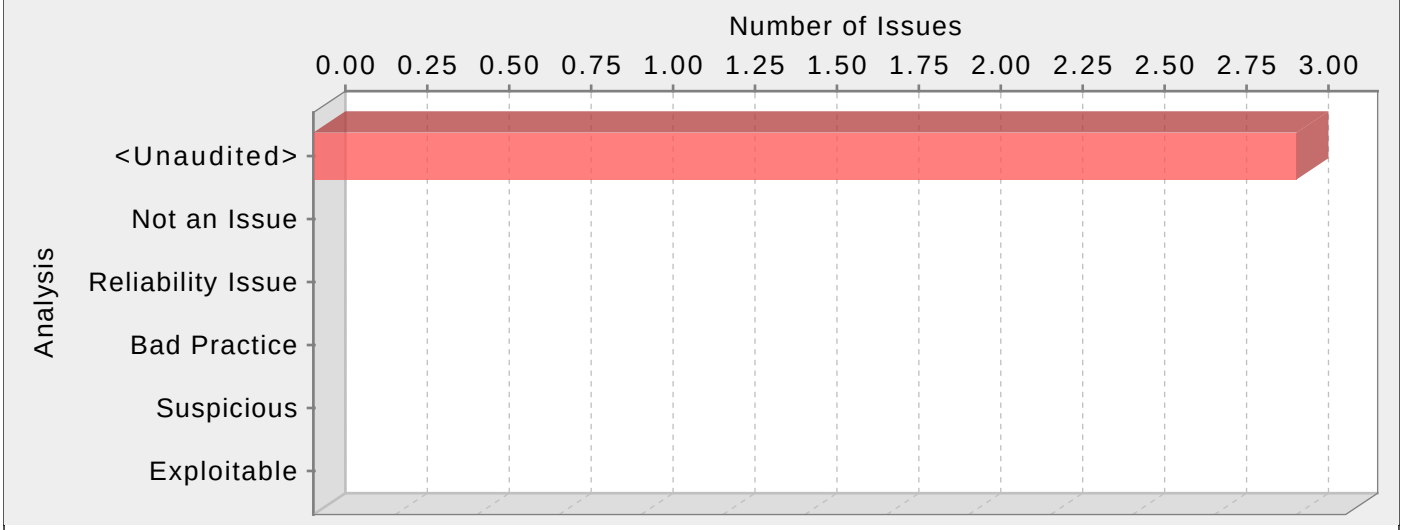
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The file jquery.dataTables.min.js interprets unvalidated user input as source code on line 363. Interpreting user-controlled instructions at run-time can allow attackers to execute malicious code.		

Source: jquery.dataTables.min.js:512 Read document.cookie()

```
510  sName+="_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase();document.cookie
     =sName+"="+encodeURIComponent(sValue)+"; expires="+date.toGMTString()+"; path=/"
511  }function _fnReadCookie(sName){var
     sNameEQ=sName+"_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase()+"=";
512  var sCookieContents=document.cookie.split(";");for(var i=0;i<sCookieContents.length;
513  i++){var c=sCookieContents[i];while(c.charAt(0)==""
     "){c=c.substring(1,c.length)}if(c.indexOf(sNameEQ)===0){return
     decodeURIComponent(c.substring(sNameEQ.length,c.length))
```

514	}}return null}function _fnGetUniqueThs(nThead){var nTrs=nThead.getElementsByTagName("tr");
Sink:	jquery.dataTables.min.js:363 eval()
361	}iDataSort=oSettings.aoColumns[aaSort[aaSort.length-1][0]].iDataSort;iDataType=oSettings.aoColumns[iDataSort].sType;
362	sDynamicSort+="iTest = oSort['"+iDataType+"-"+aaSort[aaSort.length-1][1]+"'](aoData[a]._aData["+iDataSort+"], aoData[b]._aData["+iDataSort+"]);if (iTest===0) return oSort['numeric-"+aaSort[aaSort.length-1][1]+"'](a, b); return iTest;}";
363	eval(sDynamicSort);oSettings.aiDisplayMaster.sort(fnLocalSorting)}else{var aAirSort=[];
364	var iLen=aaSort.length;for(i=0;i<iLen;i++){iDataSort=oSettings.aoColumns[aaSort[i][0]].iDataSort;
365	aAirSort.push([iDataSort,oSettings.aoColumns[iDataSort].sType+"-"+aaSort[i][1]])}oSettings.aiDisplayMaster.sort(function(a,b){var iTest;

Category: Header Manipulation: Cookies (3 Issues)



Abstract:

The method _fnCreateCookie() in jquery.dataTables.min.js includes unvalidated data in an HTTP cookie on line 510. This enables Cookie manipulation attacks and can lead to other HTTP Response header manipulation attacks like: cache-poisoning, cross-site scripting, cross-user defacement, page hijacking or open redirect.

Explanation:

Cookie 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 cookie sent to a web user without being validated.

As with many software security vulnerabilities, cookie 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 cookie.

Cookie Manipulation: When combined with attacks like cross-site request forgery, attackers may change, add to, or even overwrite a legitimate user's cookies.

Being an HTTP Response header, Cookie manipulation attacks can also lead to other types of attacks like:

HTTP Response Splitting:

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.

```
author = form.author.value;
...
document.cookie = "author=" + author + ";expires="+cookieExpiration;
...
```

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 will be able to 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 may 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 may 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.

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

Recommendations:

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

Since Header Manipulation vulnerabilities like cookie manipulation 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.

After 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.

jquery.dataTables.min.js, line 510 (Header Manipulation: Cookies)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method _fnCreateCookie() in jquery.dataTables.min.js includes unvalidated data in an HTTP cookie on line 510. This enables Cookie manipulation attacks and can lead to other HTTP Response header manipulation attacks like: cache-poisoning, cross-site scripting, cross-user defacement, page hijacking or open redirect.		

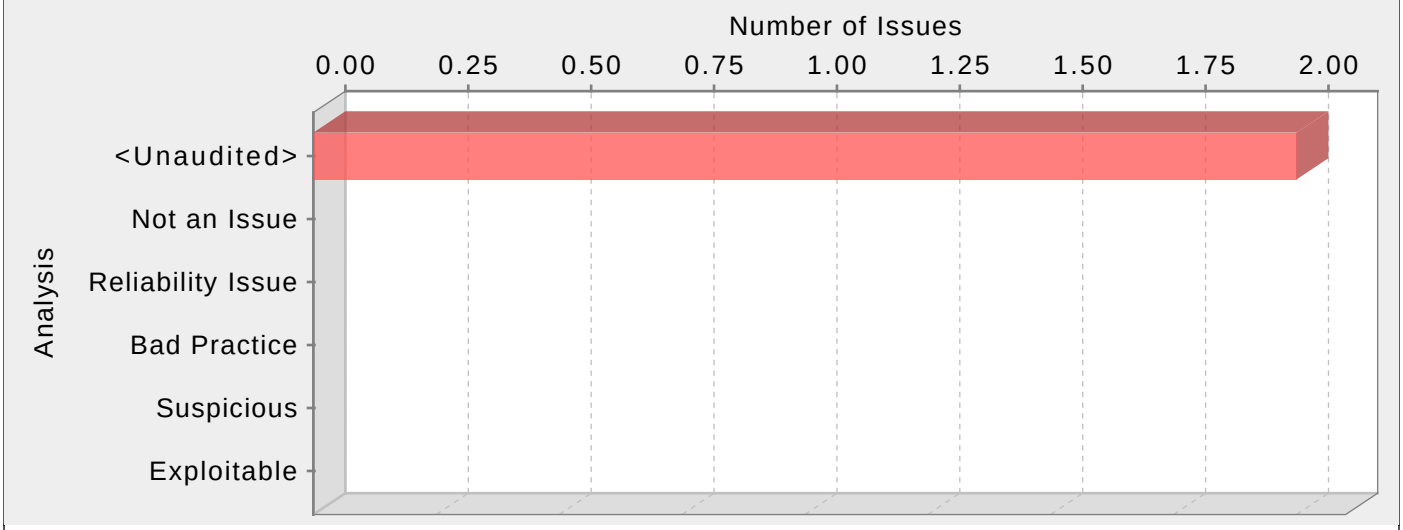
Source: jquery.dataTables.min.js:510 Read window.location()

```
508  i++){oInit.saved_aoColumns[i]={};oInit.saved_aoColumns[i].bVisible=oData.abVisCols[i]
509  }}}}function _fnCreateCookie(sName,sValue,iSecs){var date=new
     Date();date.setTime(date.getTime()+(iSecs*1000));
510  sName+="_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase();document.cookie
     =sName+"="+encodeURIComponent(sValue)+"; expires="+date.toGMTString()+"; path=/"
511  }function _fnReadCookie(sName){var
     sNameEQ=sName+"_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase()+"=";
512  var sCookieContents=document.cookie.split(";");for(var i=0;i<sCookieContents.length;
```

Sink: jquery.dataTables.min.js:510 Assignment to document.cookie()

```
508  i++){oInit.saved_aoColumns[i]={};oInit.saved_aoColumns[i].bVisible=oData.abVisCols[i]
509  }}}}function _fnCreateCookie(sName,sValue,iSecs){var date=new
     Date();date.setTime(date.getTime()+(iSecs*1000));
510  sName+="_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase();document.cookie
     =sName+"="+encodeURIComponent(sValue)+"; expires="+date.toGMTString()+"; path=/"
511  }function _fnReadCookie(sName){var
     sNameEQ=sName+"_"+window.location.pathname.replace(/[\/:]/g,"").toLowerCase()+"=";
512  var sCookieContents=document.cookie.split(";");for(var i=0;i<sCookieContents.length;
```

Category: XML External Entity Injection (2 Issues)



Abstract:

XML parser configured in ConverterEngine.java:75 does not prevent nor limit external entities resolution. This can expose the parser to an XML External Entities attack.

Explanation:

XML External Entities attacks benefit from an XML feature to build documents dynamically at the time of processing. An XML entity allows inclusion of data dynamically from a given resource. External entities allow an XML document to include data from an external URI. Unless configured to do otherwise, external entities force the XML parser to access the resource specified by the URI, e.g., a file on the local machine or on a remote system. This behavior exposes the application to XML External Entity (XXE) attacks, which can be used to perform denial of service of the local system, gain unauthorized access to files on the local machine, scan remote machines, and perform denial of service of remote systems.

The following XML document shows an example of an XXE attack.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

This example could crash the server (on a UNIX system), if the XML parser attempts to substitute the entity with the contents of the /dev/random file.

Recommendations:

The XML unmarshaller should be configured securely so that it does not allow external entities as part of an incoming XML document.

To avoid XXE injection do not use unmarshal methods that process an XML source directly as java.io.File, java.io.Reader or java.io.InputStream. Parse the document with a securely configured parser and use an unmarshal method that takes the secure parser as the XML source as shown in the following example:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.parse(<XML Source>);
Model model = (Model) u.unmarshal(document);
```

Tips:

1. Fortify RTA adds protection against this category.

ConverterEngine.java, line 75 (XML External Entity Injection)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	XML parser configured in ConverterEngine.java:75 does not prevent nor limit external entities resolution. This can expose the parser to an XML External Entities attack.		
Source:	HTTPDownloader.java:88 java.net.URLConnection.getInputStream()		

```
86      throw new ServiceUnavailableException("Error code:" + connection.getResponseCode());
87    }
```

```
88                      reader = new BufferedInputStream(connection.getInputStream(), bufferSize);
89                      int read;
90                      byte[] buffer = new byte[bufferSize];
```

Sink: ConverterEngine.java:75 javax.xml.parsers.DocumentBuilder.parse()

```
73                      dbf.setNamespaceAware(true);
74                      DocumentBuilder db = dbf.newDocumentBuilder();
75                      doc = db.parse(new InputSource(new StringReader(xml)));
76                    }
77                    catch (Exception e) {
```

Issue Count by Category

Issues by Category	
Cross-Site Scripting: DOM	33
Privacy Violation	24
Dynamic Code Evaluation: Code Injection	6
Header Manipulation: Cookies	3
XML External Entity Injection	2

Issue Breakdown by Analysis

Issues by Analysis