

🔗 b079a654c1 ▾

...

Knowage-Server / knowageutils / src / main / java / it / eng / spagobi / utilities / filters / XSSRequestWrapper.java / <> Jump to ▾



n3ils Fixed a typo.

🕒 History

👤 3 contributors



453 lines (353 sloc) | 16.8 KB

...

```

1  /*
2   * Knowage, Open Source Business Intelligence suite
3   * Copyright (C) 2016 Engineering Ingegneria Informatica S.p.A.
4   *
5   * Knowage is free software: you can redistribute it and/or modify
6   * it under the terms of the GNU Affero General Public License as published by
7   * the Free Software Foundation, either version 3 of the License, or
8   * (at your option) any later version.
9   *
10  * Knowage is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU Affero General Public License for more details.
14  *
15  * You should have received a copy of the GNU Affero General Public License
16  * along with this program. If not, see <http://www.gnu.org/licenses/>.
17  */
18  package it.eng.spagobi.utilities.filters;
19
20  import java.io.IOException;
21  import java.net.MalformedURLException;
22  import java.net.URL;
23  import java.util.Iterator;
24  import java.util.List;
25  import java.util.regex.Matcher;
26
27  import java.util.regex.Pattern;
```

```
28 import javax.servlet.ServletInputStream;
29 import javax.servlet.http.HttpServletRequest;
30 import javax.servlet.http.HttpServletRequestWrapper;
31
32 import org.apache.commons.validator.UrlValidator;
33 import org.apache.log4j.Logger;
34
35 import it.eng.spagobi.utilities.whitelist.WhiteList;
36
37 public class XSSRequestWrapper extends HttpServletRequestWrapper {
38
39     private static transient Logger logger = Logger.getLogger(XSSRequestWrapper.class);
40     private static WhiteList whitelist = WhiteList.INSTANCE;
41
42     public XSSRequestWrapper(HttpServletRequest servletRequest) {
43         super(servletRequest);
44     }
45
46     @Override
47     public String[] getParameterValues(String parameter) {
48         String[] values = super.getParameterValues(parameter);
49
50         if (values == null) {
51             return null;
52         }
53
54         int count = values.length;
55         String[] encodedValues = new String[count];
56         for (int i = 0; i < count; i++) {
57             encodedValues[i] = stripXSS(values[i]);
58         }
59
60         return encodedValues;
61     }
62
63     @Override
64     public String getParameter(String parameter) {
65         String value = super.getParameter(parameter);
66
67         return stripXSS(value);
68     }
69
70     @Override
71     public String getHeader(String name) {
72         String value = super.getHeader(name);
73         return stripXSS(value);
74     }
75
76     @Override
```

```

77     public ServletInputStream getInputStream() throws IOException {
78
79         return super.getInputStream();
80     }
81
82     public static String stripXSS(String value) {
83         logger.debug("IN");
84         String initialValue = value;
85
86         if (value != null) {
87             // NOTE: It's highly recommended to use the ESAPI library and uncomment th
88             // avoid encoded attacks.
89             // value = ESAPI.encoder().canonicalize(value);
90
91             // Avoid null characters
92             value = value.replaceAll("", "");
93
94             // Avoid anything between script tags
95             Pattern scriptPattern = Pattern.compile("<script>(.*?)</script>", Pattern.
96             value = scriptPattern.matcher(value).replaceAll("");
97
98             scriptPattern = Pattern.compile("&script&;(.*?)&;/script&;", Patt
99             value = scriptPattern.matcher(value).replaceAll("");
100
101             // Avoid anything in a src='...' type of expression
102             // Pattern.compile("src[\\r\\n]*=[\\r\\n]*\\\"(.*?)\\\"\"", Pattern.CASE_INSENSI
103             // value = scriptPattern.matcher(value).replaceAll("");
104             //
105             // scriptPattern = Pattern.compile("src[\\r\\n]*=[\\r\\n]*\\\"(.*?)\\\"\"", Patt
106             // value = scriptPattern.matcher(value).replaceAll("");
107
108             value = checkImgTags(value);
109             value = checkIframeTags(value);
110             value = checkAnchorTags(value);
111             value = checkVideoTags(value);
112             value = checkCSS(value);
113
114             // Remove any lonesome </script> tag
115             scriptPattern = Pattern.compile("</script>", Pattern.CASE_INSENSITIVE);
116             value = scriptPattern.matcher(value).replaceAll("");
117
118             scriptPattern = Pattern.compile("&;/script&;", Pattern.CASE_INSENSITIV
119             value = scriptPattern.matcher(value).replaceAll("");
120
121             // Remove any lonesome <script ...> tag
122             scriptPattern = Pattern.compile("<script(.*?)>", Pattern.CASE_INSENSITIVE
123             value = scriptPattern.matcher(value).replaceAll("");
124
125             scriptPattern = Pattern.compile("&script(.*?)&;", Pattern.CASE_INSENS

```

```

126         value = scriptPattern.matcher(value).replaceAll("");
127
128         // Avoid eval(...) expressions
129         scriptPattern = Pattern.compile("eval\\((.*?)\\)", Pattern.CASE_INSENSITIVE);
130         value = scriptPattern.matcher(value).replaceAll("");
131
132         // Avoid expression(...) expressions
133         scriptPattern = Pattern.compile("expression\\((.*?)\\)", Pattern.CASE_INSENSITIVE);
134         value = scriptPattern.matcher(value).replaceAll("");
135
136         // Avoid javascript:... expressions
137         scriptPattern = Pattern.compile("javascript:", Pattern.CASE_INSENSITIVE);
138         value = scriptPattern.matcher(value).replaceAll("");
139
140         // Avoid vbscript:... expressions
141         scriptPattern = Pattern.compile("vbscript:", Pattern.CASE_INSENSITIVE);
142         value = scriptPattern.matcher(value).replaceAll("");
143
144         // Avoid onload= expressions
145         scriptPattern = Pattern.compile("onload(.*)=", Pattern.CASE_INSENSITIVE);
146         value = scriptPattern.matcher(value).replaceAll("");
147
148         // Avoid onClick= expressions
149         scriptPattern = Pattern.compile("onClick(.*)=", Pattern.CASE_INSENSITIVE);
150         value = scriptPattern.matcher(value).replaceAll("");
151
152         // Avoid anything between form tags
153         Pattern formPattern = Pattern.compile("<form(.*)></form>", Pattern.CASE_INSENSITIVE);
154         value = formPattern.matcher(value).replaceAll("");
155
156         // Avoid anything between a tags
157         // Pattern aPattern = Pattern.compile("<a(.*)></a>", Pattern.CASE_INSENSITIVE);
158         // value = aPattern.matcher(value).replaceAll("");
159
160         // aPattern = Pattern.compile("<a(.*)></a>", Pattern.CASE_INSENSITIVE);
161         // value = aPattern.matcher(value).replaceAll("");
162
163         Pattern aPattern = Pattern.compile("&lt;a(.*)&lt;/a>", Pattern.CASE_INSENSITIVE);
164         value = aPattern.matcher(value).replaceAll("");
165
166         // Avoid anything between button tags
167         Pattern buttonPattern = Pattern.compile("<button(.*)></button>", Pattern.CASE_INSENSITIVE);
168         value = buttonPattern.matcher(value).replaceAll("");
169
170         buttonPattern = Pattern.compile("<button(.*)></button>", Pattern.CASE_INSENSITIVE);
171         value = buttonPattern.matcher(value).replaceAll("");
172
173         buttonPattern = Pattern.compile("&lt;button(.*)&lt;/button&gt;", Pattern.CASE_INSENSITIVE);
174         value = buttonPattern.matcher(value).replaceAll("");

```

```

175
176         // Example value ="<object data=\"javascript:alert('XSS')\"></object>"
177         // Avoid anything between script tags
178         Pattern objectPattern = Pattern.compile("<object(?:.*?)></object>", Pattern.C
179         value = objectPattern.matcher(value).replaceAll("");
180
181         objectPattern = Pattern.compile("&lt;object(?:.*?)&lt;/object&gt;", Pattern.
182         value = objectPattern.matcher(value).replaceAll("");
183
184         // Remove any lonesome </object> tag
185         objectPattern = Pattern.compile("</object>", Pattern.CASE_INSENSITIVE);
186         value = objectPattern.matcher(value).replaceAll("");
187
188         objectPattern = Pattern.compile("&lt;/object&gt;", Pattern.CASE_INSENSITIV
189         value = objectPattern.matcher(value).replaceAll("");
190
191         // Remove any lonesome <object ...> tag
192         objectPattern = Pattern.compile("<object(?:.*?)>", Pattern.CASE_INSENSITIVE
193         value = objectPattern.matcher(value).replaceAll("");
194
195         objectPattern = Pattern.compile("&lt;object(?:.*?)&gt;", Pattern.CASE_INSEN
196         value = objectPattern.matcher(value).replaceAll("");
197
198         if (!value.equalsIgnoreCase(initialValue)) {
199             logger.warn("Message: detected a web attack through injection");
200         }
201
202     }
203
204     logger.debug("OUT");
205     return value;
206 }
207
208 private static String checkImgTags(String value) {
209     logger.debug("IN");
210     Pattern maliciousImgPattern = Pattern.compile("&lt;img(?:.*?)&gt;", Pattern.CASE_INS
211     value = maliciousImgPattern.matcher(value).replaceAll("");
212
213     Pattern scriptPattern = Pattern.compile("<img[^>]+(src\\s*=\\s*['\"]([^\"]+)['\"]|
214         Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
215     Pattern dataPattern = Pattern.compile("data:image\\/(gif|jpeg|pjpeg|png|svg|\\+xml|
216         Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
217     Matcher scriptMatcher = scriptPattern.matcher(value);
218
219     while (scriptMatcher.find()) {
220         String img = scriptMatcher.group();
221         String link = scriptMatcher.group(2);
222
223         Matcher dataMatcher = dataPattern.matcher(link);

```

```

224
225         if (!dataMatcher.find()) {
226             try {
227                 URL url = new URL(link);
228                 String baseUrl = url.getProtocol() + "://" + url.getHost()
229
230                 if (!whitelist.getExternalServices().contains(baseUrl)) {
231                     logger.warn("Provided image's src is: " + url + ".
232                         value = value.replace(img, "");
233                 }
234
235             } catch (MalformedURLException e) {
236                 logger.debug("URL [" + link + "] is malformed. Trying to s
237                 if (isValidRelativeURL(link) && isTrustedRelativePath(link
238                     logger.debug("URL " + link + " is recognized to be
239                 } else {
240                     logger.error("Malformed URL [" + link + "]", e);
241                     value = value.replace(img, "");
242                 }
243             }
244
245         }
246
247     }
248
249     logger.debug("OUT");
250     return value;
251 }
252
253 private static String checkIframeTags(String value) {
254     logger.debug("IN");
255     Pattern maliciousTagPattern = Pattern.compile("<iframe(?:.*?)*>", Pat
256     value = maliciousTagPattern.matcher(value).replaceAll("");
257
258     Pattern scriptPattern = Pattern.compile("<iframe[^>]+(src\\s*=\\s*['\"]([^\"]+)'
259         Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
260     Matcher scriptMatcher = scriptPattern.matcher(value);
261
262     while (scriptMatcher.find()) {
263         String iframe = scriptMatcher.group();
264         String link = scriptMatcher.group(2);
265
266         try {
267             URL url = new URL(link);
268             String baseUrl = url.getProtocol() + "://" + url.getHost();
269
270             if (!whitelist.getExternalServices().contains(baseUrl)) {
271                 logger.warn("Provided iframe's src is: " + url + ". Iframe
272                 value = value.replace(iframe, "");

```

```

273         }
274
275     } catch (MalformedURLException e) {
276         logger.debug("URL [" + link + "] is malformed. Trying to see if it
277         if (isValidRelativeURL(link) && isTrustedRelativePath(link)) {
278             logger.debug("URL " + link + " is recognized to be a valid
279         } else {
280             logger.error("Malformed URL [" + link + "]", e);
281             value = value.replace(iframe, "");
282         }
283     }
284
285 }
286
287 logger.debug("OUT");
288 return value;
289 }
290
291 private static String checkAnchorTags(String value) {
292     logger.debug("IN");
293     Pattern aPattern = Pattern.compile("<a([>]+)>(.*?)</a>", Pattern.CASE_INSENSITIVE
294     Pattern hrefPattern = Pattern.compile("\\s*href\\s*=\\s*['\"]([^\"]+)['\"]", Patt
295
296     Matcher aTagMatcher = aPattern.matcher(value);
297
298     while (aTagMatcher.find()) {
299         String aTag = aTagMatcher.group();
300         String href = aTagMatcher.group(1);
301
302         // In <a> tag find href attribute
303         Matcher hrefMatcher = hrefPattern.matcher(href);
304
305         while (hrefMatcher.find()) {
306             String link = hrefMatcher.group(1);
307
308             try {
309                 URL url = new URL(link);
310                 String baseUrl = url.getProtocol() + "://" + url.getHost()
311
312                 if (!whitelist.getExternalServices().contains(baseUrl)) {
313                     logger.warn("Provided anchor's href is: " + url +
314                     value = value.replace(aTag, "");
315                 }
316
317             } catch (MalformedURLException e) {
318                 logger.debug("URL [" + link + "] is malformed. Trying to s
319                 if (isValidRelativeURL(link) && isTrustedRelativePath(link
320                     logger.debug("URL " + link + " is recognized to be
321             } else {

```

```

322         logger.error("Malformed URL [" + link + "]", e);
323         value = value.replace(aTag, "");
324     }
325 }
326 }
327
328 }
329
330     logger.debug("OUT");
331     return value;
332 }
333
334 private static String checkVideoTags(String value) {
335     logger.debug("IN");
336     Pattern maliciousPattern = Pattern.compile("<video(?:.*?)video>", Pattern.CASE_INSENSITIVE);
337     value = maliciousPattern.matcher(value).replaceAll("");
338
339     Pattern scriptPattern = Pattern.compile("<video(?:.*?)>", Pattern.CASE_INSENSITIVE);
340     Pattern srcAttributePattern = Pattern.compile("\\s*src\\s*=\\s*['\"]([^\"]+)['\"]", Pattern.CASE_INSENSITIVE);
341     Matcher matcher = scriptPattern.matcher(value);
342
343     while (matcher.find()) {
344         String video = matcher.group();
345         String betweenVideoTags = matcher.group(1);
346
347         Matcher srcMatcher = srcAttributePattern.matcher(betweenVideoTags);
348
349         while (srcMatcher.find()) {
350             String link = srcMatcher.group(1);
351
352             try {
353                 URL url = new URL(link);
354                 String baseUrl = url.getProtocol() + "://" + url.getHost();
355
356                 if (!whitelist.getExternalServices().contains(baseUrl)) {
357                     logger.warn("Provided anchor's href is: " + url + " is not in the whitelist");
358                     value = value.replace(video, "");
359                 }
360
361             } catch (MalformedURLException e) {
362                 logger.debug("URL [" + link + "] is malformed. Trying to sanitize it");
363                 if (isValidRelativeURL(link) && isTrustedRelativePath(link)) {
364                     logger.debug("URL " + link + " is recognized to be a trusted relative path");
365                 } else {
366                     logger.error("Malformed or untrusted URL [" + link + "]", e);
367                     value = value.replace(video, "");
368                 }
369             }
370         }
371     }

```



```

371
372     }
373
374     logger.debug("OUT");
375     return value;
376 }
377
378 private static String checkCSS(String value) {
379     logger.debug("IN");
380     Pattern cssUrlPattern = Pattern.compile("url\\s*\\(['\"]?([^\"]\\\\\\\\)+['\"]?\\\\)", P
381     Pattern cssUrlDataPattern = Pattern.compile("data:image\\\\/(gif|jpeg|png|svg\\
382         Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
383     Pattern domElementID = Pattern.compile("(#[a-zA-Z0-9\\\\_\\\\-]+)", Pattern.CASE_INSEN
384     String domId = "";
385
386     Matcher urlMatcher = cssUrlPattern.matcher(value);
387
388     while (urlMatcher.find()) {
389         String cssUrl = urlMatcher.group();
390         String link = urlMatcher.group(1);
391
392         Matcher dataMatcher = cssUrlDataPattern.matcher(link);
393         Matcher domIdMatcher = domElementID.matcher(link);
394
395         if (domIdMatcher.find()) {
396             domId = domIdMatcher.group();
397             if (domId.length() > 50) {
398                 logger.warn("Provided url attribute with Id is: " + domId
399                 value = value.replace(cssUrl, "");
400             }
401         }
402
403         if (!dataMatcher.find()) {
404             try {
405                 URL url = new URL(link);
406                 String baseUrl = url.getProtocol() + "://" + url.getHost()
407
408                 if (!whitelist.getExternalServices().contains(baseUrl)) {
409                     logger.warn("Provided CSS url attribute is: " + ur
410                     value = value.replace(cssUrl, "");
411                 }
412             } catch (MalformedURLException e) {
413                 logger.debug("URL [" + link + "] is malformed. Trying to s
414                 if (isValidRelativeURL(link) && isTrustedRelativePath(link
415                     logger.debug("URL " + link + " is recognized to be
416                 } else if (link.equals(domId)) {
417                     return value;
418                 } else {
419                     logger.error("Malformed or untrusted URL [" + link

```

```
420         value = value.replace(cssUrl, "");
421     }
422 }
423 }
424
425 }
426
427     logger.debug("OUT");
428     return value;
429 }
430
431 private static boolean isValidRelativeURL(String url) {
432     String absoluteUrl = "http://mynonexistingserver.something.smt:99999" + url;
433     UrlValidator urlValidator = new UrlValidator();
434     if (urlValidator.isValid(absoluteUrl)) {
435         return true;
436     } else {
437         return false;
438     }
439 }
440
441 private static boolean isTrustedRelativePath(String url) {
442     List<String> relativePaths = whitelist.getRelativePaths();
443     Iterator<String> it = relativePaths.iterator();
444
445     while (it.hasNext()) {
446         if (url.startsWith(it.next())) {
447             return true;
448         }
449     }
450     return false;
451 }
452
453 }
```