# Java MVC 1.0 Vulnerability Research

By Juan C. Moreno

Prepared for Micro Focus Fortify

Abstract

The following research seeks to identify and explore a series of critical vulnerability classes that applications leveraging the Java MVC 1.0 (Ozark) framework may be exposed to. This body of work will focus mainly on SSTI and EL injection vulnerabilities, which can expose applications to Critical RCE exploits.

## Java MVC 1.0 Vulnerability Research

Java MVC 1.0 is commonly known as Ozark, however in more recent years it became known as Krazo (due to trademark issues). This framework is an action-oriented implementation of the popular architectural design pattern Model-View-Controller (MVC), and a direct alternative to the traditional component-oriented JSF MVC implementation.

### MVC

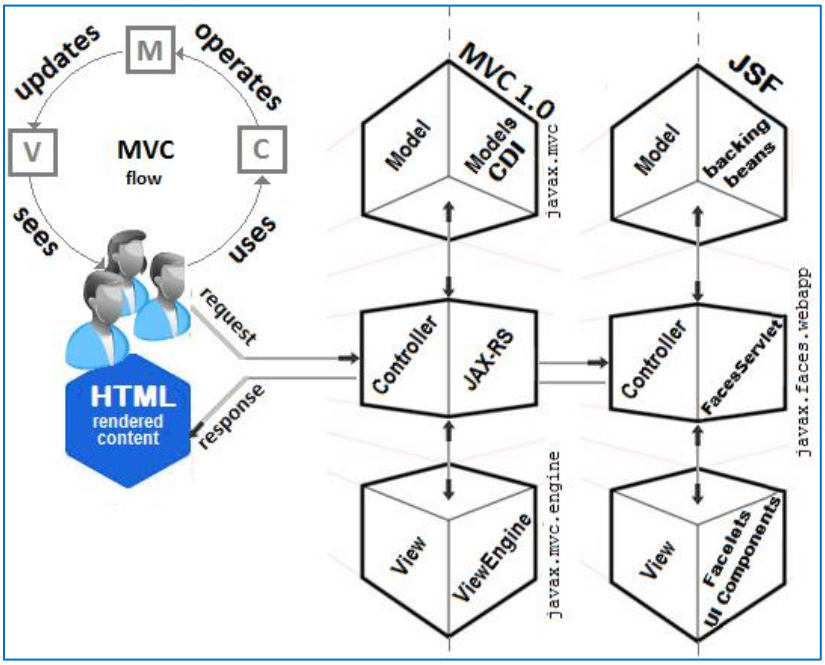
MVC or Model-View-Controller is an architectural design pattern that seeks to separate applications in 3 main components: a model, a view and a controller. The model is the data layer of an application, the view handles the presentation layer and the controller is the business logic layer. The loose coupling between these layers allows for greater flexibility and independence. A concrete example on Ozark is a capability of leveraging various view engines interchangeably "without" causing unintended side effects upon the other layers.

### Java MVC 1.0 vs JSF MVC

**JSF** is an early implementation of the MVC design pattern within the Java ecosystem. This framework is component-oriented, and places a great emphasis on the concept of inversion IoC; whereas the framework itself exerts a greater amount of control over the execution flow of the application. JSF applications place a heavy abstraction from the Request and Response of the underlying HTTP protocol, and client-side components such as HTML/CSS/JS.

In contrast **Java MVC 1.0** (from hereon will be referred to as **Ozark**) is action-oriented. Meaning the controller dispatches to an action based on the information from the request. This implementation provides low level access to the request and response, as well as the client-side components HTML/CSS/JS.

Figure 1: MVC Architecture – Ozark  vs JSF




JSF can be considered analogous to traditional ASP.NET WebForms in the sense that the application imposes a heavy abstraction on the request and response, and UI components. Ozark is similar to ASP.NET MVC.

Table1: Similarities of JSF and ASP.NET WebForms

| Commonalities | JSF | ASP.NET WebForms |
|---|---|---|
| HTTP Request and Response | Initial Request and Postback. No GET or POST. | |
| HTML | Reusable UI components instead of low level HTML (e.g. `h:inputText, h:panelGrid`) | Reusable UI components instead of low level HTML (e.g. `asp:TextBox, asp:gridview`) |
| CSS | Provides styles and themes by default | Provides specialized attributes for handling CSS |
| JS | Provide special components for things such as AJAX<br><br>(e.g. `<f:ajax execute="@form" render="@form" />`) | Provide special components for things such as AJAX<br>(e.g. `<asp:UpdatePanel ID="UpdatePanel1" runat="server">`) |
| Drivers | Page Centric | |

Table 2: Similarities of Ozark and ASP.NET MVC

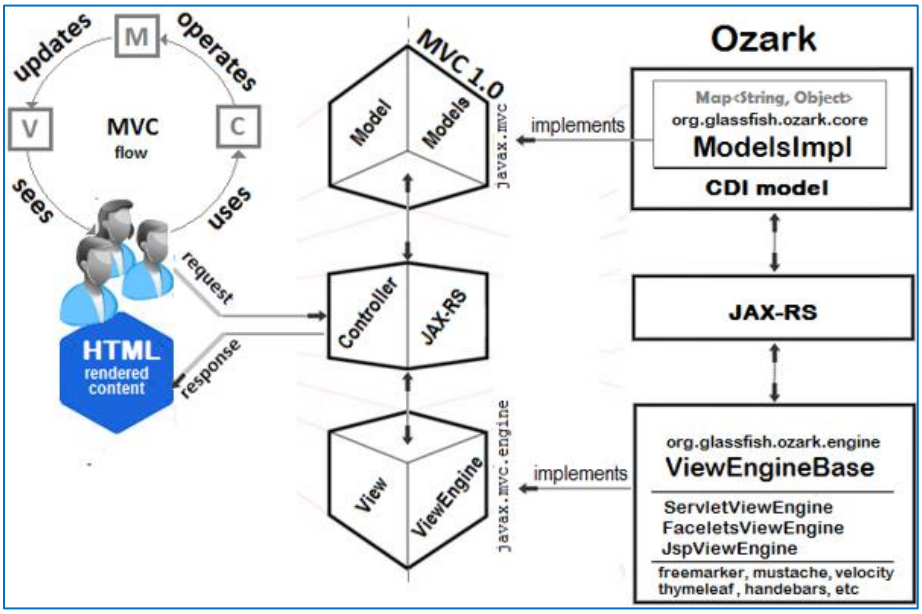
| Commonalities | Ozark | ASP.NET WebForms |
|---|---|---|
| HTTP Request and Response | Manages GET, POST and other HTTP Verbs directly | |
| HTML/CSS/JS | Low level usage of standard HTML, CSS and JavaScript is common-place<br><br>Favored for usage of popular Front-End frameworks (e.g. jQuery, Bootstrap, Angular) | |
| Drivers | | Request Centric |

## Ozark Vulnerability Research

### Ozark Architecture

At the design level Ozark's architecture of obviously driven by the MVC design pattern that was previously highlighted, however it's the implementation specifics that separates Ozark from JSF.

Figure 2: Ozark Architecture




At its core Ozark leverages various standard java features to support it's MVC architecture.

Table 3: Ozark Supporting Technologies

| Layer | Technologies |
|---|---|
| Controller | JAX-RS – This is used for request processing, as this specification was already inherently action-oriented. |
| Model | CDI – Stands for Contexts and Dependency Injection, which is leveraged at the model layer, and it's also a form of IoC that was still retained on the Ozark specification. |
| View | ViewEngineBase – Provides an abstraction layer to support various presentation later technologies and templating engines. Examples of these are: |

| Layer | Technologies |
|---|---|
| | • JSP - JspViewEngine<br>• Facelets – FaceletsViewEngine (it provides cross-compatibility with JSF)<br>• Freemarker – FreemarkerViewEngine<br>• Velocity – VelocityViewEngine<br>• Thymeleaf – ThymeleafViewEngine<br>• Handlebars – HandlebarsViewEngine<br>• Mustache – MustacheViewEngine, etc. |
| Cross-Layer Technologies | Bean Validation JSR 380 – Leveraged for the validation layer. This layer interacts with both the Model and the Controller.<br><br>Expression Language – It's the glue that ties together the view with the model (and it will be focused on heavily on this research).<br><br>Ozark supports various EL technologies depending on the particular ViewEngine that it being leveraged. |

**Vulnerability Classes in Ozark**.

This research will be focused on SSTI (Server-Side Template Injection) and EL Injection (Expression Language Injection) vulnerabilities.

SSTI and EL injection vulnerabilities are quite related in the sense that template engines leverage various types of EL to access and process beans at the presentation layer. An adversary trying to exploit any of these would generally be injecting some form of EL (amongst other things).

One attack vector that is unique to the concept of SSTI is the fact that applications often times seek to leverage template engines in order to provide templating capabilities for its users. Adversaries that can define or modify templates may be able to inject and persist arbitrary code; including but not limited to EL, for example template specific directives.

Injection of arbitrary code via SSTI will generally include EL, however EL does not necessarily need to be processed by a template engine.

**View Engines Evaluated**

This research reviewed multiple view engines with various degrees of success in the time invested. The main body of this research was focused on attack vectors that may be injected and evaluated at runtime, rather than focusing on attack vectors, which require "Template Editor" access.

The main view engines covered will be: Freemarker, Velocity, and Facelets. Some of these will highlight view level evaluation and some will highlight evaluation at the Controller and validation tiers.

***Apache Freemarker***

This is a template engine owned and maintained by the Apache Software Foundation.

*Realtime Injection*

Freemarker performs double evaluation on user input that is processed by the "interpret" directive, and subsequently processed as a user-defined directives (denoted by <@... >).

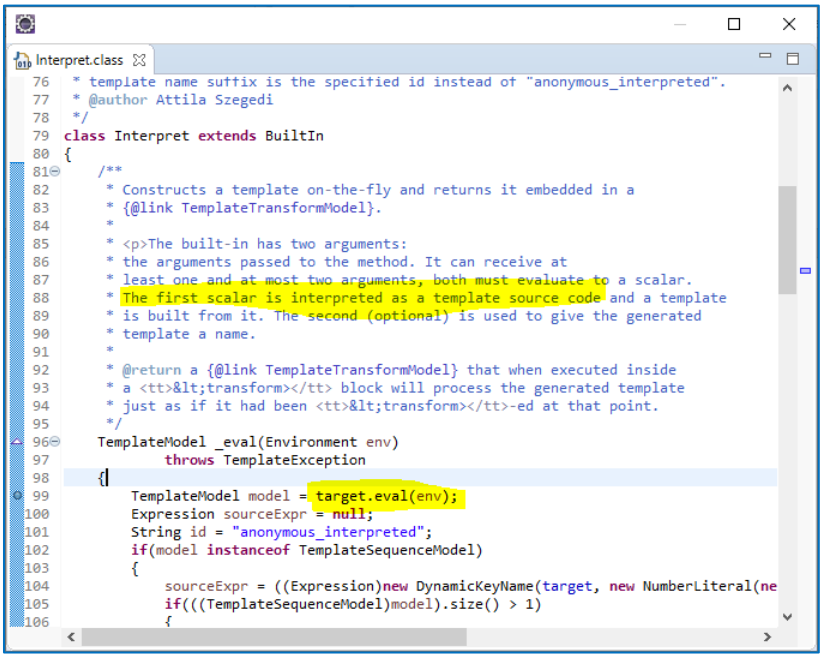
Table 4: Freemarker Payloads:

| Attacker Input | Template Evaluation |
|---|---|
| Any of the following:<br>➢ ${"freemarker.template.utility.Execute"?new()("calc")}<br>➢ #{"freemarker.template.utility.Execute"?new()("calc")} | Any of the following:<br>➢ <#assign tmpVar= inputVar?interpret><@ tmpVar /><br>➢ <@inputVar?interpret /><br>Note: This is the short-hand version of the first example |

**Causes:**

1. *<#assign tmpVar= inputVar?interpret>* - Interpret treats the argument passed as "Template Source Code" (see Figure 3).

Figure 3: Interpret Definition




2. <@tmpVar /> - Treats tmpVar as a user defined directive or macro, and it evaluates it accordingly.

**Effect:**

1. The attacker payload is executed and the Calc is executed.

**Mitigation:**

1. Avoid leveraging the Interpret directive on input from untrusted sources.
2. If the use of the Interpret directive on an untrusted source is required, thoroughly sanitize the input before passing it to the View.
3. Maintain the Freemarker library up to date to ensure that the application in question has the most current sandbox.
4. Apply egress filters on the application server to prevent loading of remote artifacts from untrusted sources.

**Detection:**

1. Perform a data-flow analysis.
2. Detect all dynamic *sources* that cross the trust boundary (e.g. client-side input, persistence layer).
3. Evaluate if said *sources* have been processed by an input sanitization routine.
4. Flag all of the aforementioned *sources* that flow onto a *sink* that's processed by the ?interpret directive.
    a. All *sinks* that may have been processed by a sanitization routine should be flagged at least as a medium unless the sanitization routine can be ascertained to be robust for the applicable context.
    *b.* *Sinks* that have not been identified as being processed a sanitization routine should flagged as High.

***Apache Velocity***

This is a template engine owned and maintained by the Apache Software Foundation.

*Realtime Injection*

Velocity performs double evaluation on user input that is assigned to a variable via the "#set" directive and it's subsequently processed by the "#evaluate" directive.

Table 5: Velocity Payloads:

| Attacker Input | Template Evaluation |
|---|---|
| Any of the following:<br>➢ #set ($run=7*7) $run<br><br>**Note:** Although double evaluation was achieved, more time needs to be spent experimenting with the various sandbox bypasses in order to provide a working RCE payload.<br><br>This being said various valid sandbox bypasses have been showcased on Room for Escape: | #set - Any of the following:<br>➢ #set($myVar1 = $text)<br>➢ #set($myVar1 = "$text")<br>#evaluate – Any of the following:<br>➢ #evaluate($myVar1)<br>➢ #evaluate("$myVar1")<br><br>**Note:** Velocity treats single-quoted texts as as literal, hence these are not interpolated. Double-quoted variables are passed as |

| Attacker Input | Template Evaluation |
| --- | --- |
| Scribbling Outside the Lines of Template Security | velocity references and may be interpolated. |

**Causes:**

1. #set($myVar1 = $text) – Assigns the input onto a static variable.
    a. If the user input is sent directly to the #evaluate, the input is only evaluated once. Without double evaluation attacker defined expressions cannot be executed.
2. #evaluate("$myVar1") – Evaluates the value as a VTL string (Velocity Template Language) (see Figure 4 and 5).

Figure 4: Evaluate Definition

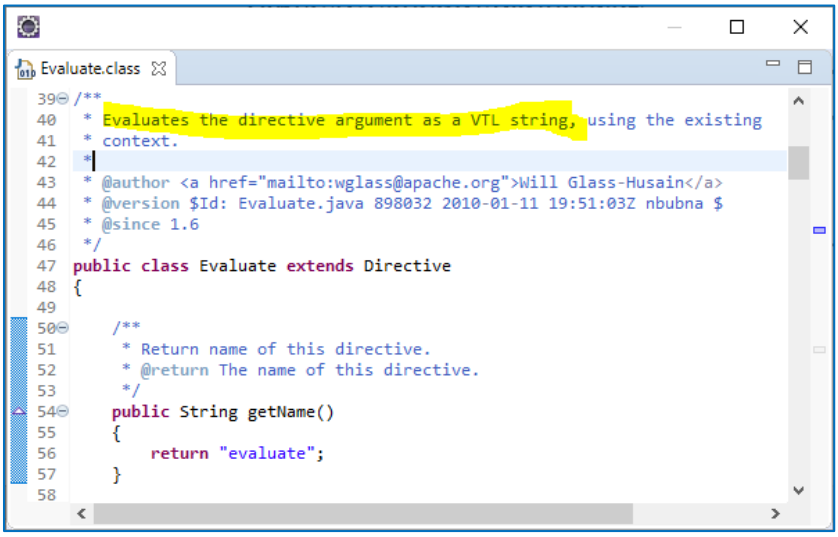
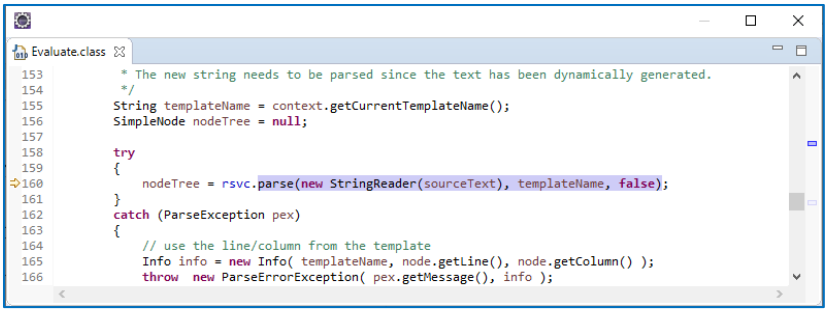



Figure 5: Parsing of dynamically generated string




**Effect:**

1. The attacker payload is executed and **#set ($run=7*7) $run** turns into a **49.**

**Mitigation:**

1. Always use singe-quotes on literal values, and values that do not require interpolation.
2. Avoid using the #evaluate directive on input from untrusted sources.
3. If the use of the #evaluate directive on an untrusted source is required, thoroughly sanitize the input before passing it to the View.
    a. If user input is being processed by various directives and subsequently evaluated, ensure that the value is not interpolated up to the moment it needs to be evaluated.

Table 6 – Safer and unsafe usage of single and double quotes:

| Safer Usage | Unsafe Usage |
|---|---|
| ➢ On the controller: model.put("param1", "#set($run=7*7) $run");<br>➢ #set( $name = '$param1' ) // Result: $name → $param1<br>➢ #evaluate("My name is $name") // Result → My name is #set($run=7*7) $run | ➢ On controller: model.put("param1", "#set($run=7*7) $run");<br>➢ #set( $name = "$param1" ) // Result: $name → #set($run=7*7) $run<br>➢ #evaluate("My name is $name") // Result → My name is 49 |

4. Maintain the Velocity library up to date to ensure that the application in question has the most current sandboxing technology.

5. Apply egress filters on the application server to prevent loading of remote artifacts from untrusted sources.

**Detection:**

5. Perform a data-flow analysis.
6. Detect all dynamic *sources* that cross the trust boundary (e.g. client-side input, persistence layer).
7. Evaluate if said *sources* have been processed by an input sanitization routine.
8. Flag all of the aforementioned *sources* that flow onto *sinks* that are processed either directly or indirectly by the #evaluate directive.
    a. All *sinks* that may have been processed by a sanitization routine should be flagged at least as a Medium unless the sanitization routine can be ascertained to be robust for the applicable context.
    *b.* *Sinks* that have not been identified as being processed a sanitization routine should flagged as High.

    **Note:** More robust risk calculations can be implemented by accounting for all of the following factors (worst case scenario):

    1. Un-sanitized input,
    2. Processed by a chain of 2 or more directives,
    3. No quotes or wrapped by double quotes more than once in the chain, and
    4. Finally, processed by the #evaluate directive.

***Facelets***

This is the view engine for JSF.

*Realtime Injection*

Facelets leverages the javax.el.ELProcessor engine. All user input that either directly or indirectly flows into a ValueExpression will be subject to evaluation once the getValue() method is invoked.

The following illustrates a series of references from Faces to the javax.el.ValueExpression.getValue() method.

Figure 6: References to ValueExpression evaluation

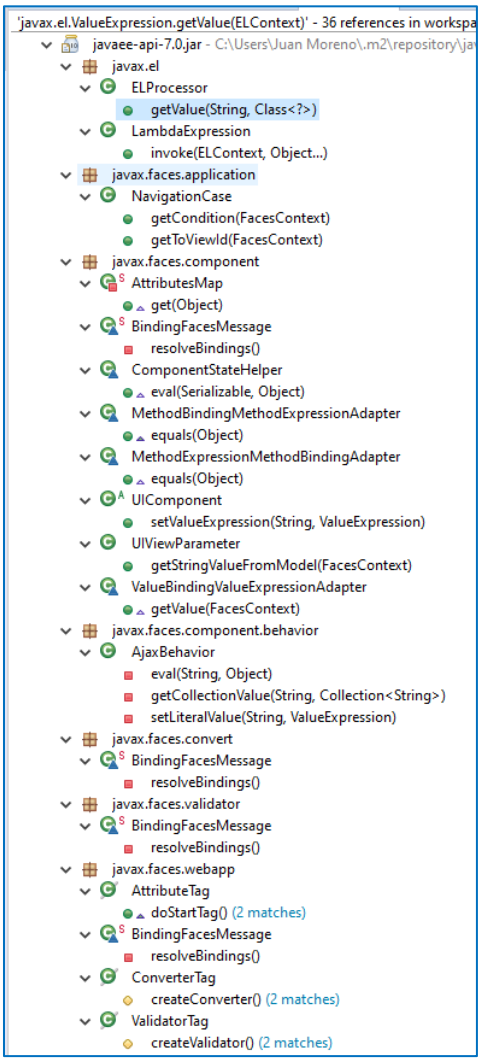


Figure 7: Sample vulnerable code

```
@GET
@Context
public String sayHello(@QueryParam("name") String name, @Context HttpServletRequest req, @Context H
    User myUser = new User();
    myUser.setName(name);
    this.models.put("text", "Hello " + myUser.getName());
    this.models.put("user", myUser);

    FacesContext context = FacesUtil.getFacesContext(req, resp); //FacesContext.getCurrentInstance(
    ExpressionFactory expressionFactory = context.getApplication().getExpressionFactory();
    ELContext elContext = context.getELContext();
    ValueExpression vex = expressionFactory.createValueExpression(elContext, name, String.class);
    String result = (String) vex.getValue(elContext);
```

**Causes:**

1. All input that flows directly in a getValue() method for a ValueExpression is subject to direct EL evaluation.

**Effect:**

1. The attacker's payload may be executed within the confines of what's available on the classpath.

**Mitigation:**

1. Never purposefully evaluate user input by a ValueExpression.
2. Maintain Java and Javax API version up to date, to comply with security updates and minor releases.
3. Avoid using EOL versions of the Java SDK.
4. Apply egress filters on the application server to prevent loading of remote artifacts from untrusted sources.

**Detection:**

1. Perform a data-flow analysis.
2. Detect all dynamic *sources* that cross the trust boundary (e.g. client-side input, persistence layer).
3. Evaluate if said *sources* have been processed by an input sanitization routine.

4. Flag all of the aforementioned *sources* that flow onto a *sink* that's processed by a javax.el.ValueExpression.
    a. All *sinks* that may have been processed by a sanitization routine should be flagged at least as a medium unless the sanitization routine can be ascertained to be robust for the applicable context.
    *b.* *Sinks* that have not been identified as being processed a sanitization routine should flagged as High.

***Bean Validation***

Ozark leverages Bean Validation for its data validation layer. Recent research on Bean Validation (JSR 380) highlighted the that custom constraint validators can build error messages with templates. Injection of untrusted data into said template will be processed as EL.

*Realtime Injection*

User input that's been explicitly added to a template prior to compilation, only requires a single evaluation in order to get processed as EL. The previous examples discussed for various template engines require double evaluation of user input since the first evaluation will merely pull the data from the bean. A secondary evaluation is needed in order get EL execution.

The example highlighted below by Alvaro's research (see *Bean Stalking*) demonstrates that strings passed to the buildConstraintViolationWithTemplate() method will be treated as a template.

Table 7: Bean Validation Payload:

| Attacker Input | Result |
|---|---|
| Any of the following:<br>➢ ${"".getClass().forName("java.lang.Runtime").getMethods()[6].invoke("".getClass().forName("java.lang.Runtime")).exec("calc.exe")} | ➢ Spawns a Calc |

Figure 8: Spawn Calc via Bean Validation vuln

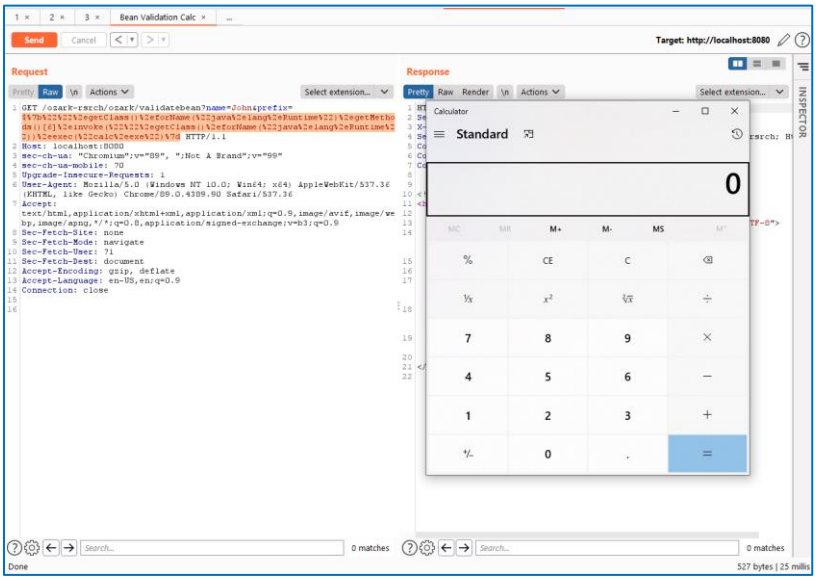



Figure 9: SSTI in constraint violation template (from Bean Stalking)

```
public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

    @Override
    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    @Override
    public boolean isValid(String object, ConstraintValidatorContext constraintContext) {
        if ( object == null ) {
            return true;
        }

        boolean isValid;
        String message = object;
        if ( caseMode == CaseMode.UPPER ) {
            isValid = object.equals( object.toUpperCase() );
            message = message + " should be in upper case."
        }
        else {
            isValid = object.equals( object.toLowerCase() );
            message = message + " should be in lower case."
        }

        if ( !isValid ) {
            constraintContext.disableDefaultConstraintViolation();
            constraintContext.buildConstraintViolationWithTemplate(message)
            .addConstraintViolation();
        }
    }
}
```

**Causes:**

1. All input that flows directly into the buildConstraintViolationWithTemplate() method will be compiled as a part of a new template, and hence it's subject to SSTI and EL processing.

**Effect:**

1. The attacker will be able to inject EL expressions as well as template directives if applicable, which would prove to be a power attack vector.

**Mitigation: (Echoes Alvaro's recommendations)**

1. Never include user input on constraint violation templates.
2. If user input is absolutely necessary:
    a. Disable EL interpolation, and only use parameter interpolation.

Figure 9: Disabling expression interpolation and enabling parameter interpolation only (from Bean Stalking).

```
Validator validator = Validation.byDefaultProvider()
   .configure()
   .messageInterpolator( new ParameterMessageInterpolator() )
   .buildValidatorFactory()
   .getValidator();
```

    b. Apply a robust sanitization on the input prior to injection.
3. Maintain all applicable framework libraries up to date to comply with security updates and minor releases.
4. Apply egress filters on the application server to prevent loading of remote artifacts from untrusted sources.

**Detection:**

1. Perform a data-flow analysis.

2. Detect all dynamic *sources* that cross the trust boundary (e.g. client-side input, persistence layer).
3. Evaluate if said *sources* have been processed by an input sanitization routine.
4. Flag all of the aforementioned *sources* that flow onto a *sink* that's processed by the ConstraintValidatorContext.buildConstraintViolationWithTemplate(String) method.
    a. All *sinks* that may have been processed by a sanitization routine should be flagged at least as a medium unless the sanitization routine can be ascertained to be robust for the applicable context.
    b. *Sinks* that have not been identified as being processed a sanitization routine should flagged as High.

**Other Vectors Including Persistent Injections and Sandbox Bypasses**

Refer to the research conducted by Alvaro Muñoz and Oleksandr Mirosh (Room for Escape: Scribbling Outside the Lines of Template Security).

## Conclusion

MVC is a very common design pattern on web applications, and with good reason, since it involves the decoupling of 3 very basic tiers (model, view and controller). Template engines form a key yet powerful component for the view (i.e. presentation layer).

Applications built on Ozark will commonly leverage these powerful presentation layer components. The unbridled capacity of templating engines may be subject to misuse by skilled attackers, hence great care must be exercised when using these.

SSTI and EL injection are common attack vectors that plague these components. All input that either directly or indirectly originates from an untrusted source, should be thoroughly sanitized. Evaluation of such input should be avoided at all costs.

References


1. Leonard Anghel, *"Introduction to the New MVC 1.0 (Ozark RI)"* (https://www.developer.com/open-source/introduction-to-the-new-mvc-1-0-ozark-ri/)
2. Ed Burns, *"Why Another MVC?"* (https://www.oracle.com/technical-resources/articles/java/mvc.html)
3. Eclipse Foundation, *"Eclipse Krazo"* (https://projects.eclipse.org/proposals/eclipse-krazo)
4. Eugen (Baeldung), *"An Introduction to CDI (Contexts and Dependency Injection) in Java"* (https://www.baeldung.com/java-ee-cdi)
5. Apache Foundation, *"Velocity Template Language (VTL): An Introduction"* (http://people.apache.org/~henning/velocity/html/ch02s02.html)
6. Java Community Process, "JSR 371: Model-View-Controller (MVC 1.0) Specification" (https://jcp.org/en/jsr/detail?id=371)
7. Oleksandr Mirosh (oleksandr.mirosh@microfocus.com) & Alvaro Muñoz (pwntester@github.com), *"Room for Escape: Scribbling Outside the Lines of Template Security"* (https://i.blackhat.com/USA-20/Wednesday/us-20-Munoz-Room-For-Escape-Scribbling-Outside-The-Lines-Of-Template-Security-wp.pdf)
8. Alvaro Muñoz (pwntester@github.com), *"Bean Stalking: Growing Java beans into RCE"* (https://securitylab.github.com/research/bean-validation-RCE/)
9. Daniel Dias,"Eclipse Krazo - Supported Engines Template" (https://medium.com/danieldiasjava/eclipse-krazo-template-engines-suportadas-ea1431ca8f4b)

10. Antón García Dosil *"TESTING VELOCITY SERVER-SIDE TEMPLATE INJECTION"* (https://antgarsil.github.io/posts/velocity/)