April 1, 2021

# GHSL-2020-021: Bypass input sanitization of EL expressions in Eclipse-EE4J

Alvaro Munoz

## Coordinated Disclosure Timeline

- 02/07/2020: Report sent to Vendor (security@eclipse.org)
- 03/12/2020: Ping them for acknowledgment
- 04/14/2020: Sent report through Bugzilla (https://bugs.eclipse.org/bugs/show_bug.cgi?id=562121)
- 03/26/2021: No response received from Eclipse. Disclosure deadline reached.
- 04/01/2021: Publication as per our disclosure policy

## Summary

A bug in the `ELParserTokenManager` enables invalid EL expressions to be evaluated as if they were valid. For example, the following message will evaluate an invalid EL expression and the interpolated message will be `1+1 = 2`:

```
1+1 = $\#{1+1}
```

(Note: EL expression delimiter is escaped and therefore it should be treated as a literal expression and not be evaluated)

This bug enables attackers to bypass input sanitization (escaping, stripping) controls that developers may have put in place when handling user-controlled data in error messages.

## Product

Eclipse-EE4J Expression Language Reference Implementation

## Tested Version

3.0.3

## Details

### Incorrect EL expression tokenization

EL expressions are used in many places. One of them is in Bean Validation constraint error messages where it can take user-controlled data. As specified in Hibernate Validator documentation:

> Note that the custom message template is passed directly to the Expression Language engine. Thus, you should be very careful when integrating user input in a custom message template as it will be interpreted by the Expression Language engine, which is usually not the behavior you expect and could allow malicious users to leak sensitive data. If you need to integrate user input, you should:
>
> - either escape it by using the Jakarta Bean Validation message interpolation escaping rules;
> - or, even better, pass it as message parameters or expression variables by unwrapping the context to HibernateConstraintValidatorContext.

Several applications attempt to prevent such EL injections by replacing the EL opening delimiter `${` with just `{`. e.g.:

```
    public String replaceElDelimiter(final String value) {
        if (value != null) {
            return value.replaceAll("\\$+\\{", "{");
        }
        return null;
    }
```

This is seemingly a secure way to prevent injection attacks since all occurrences of `${` will be replaced with `{`, and since the regex matches repeating `$` it will also fix more intricate injection attempts that send e.g. `$${` in an attempt to arrive at the `${` delimiter to achieve EL execution.

A way of bypassing this control is trying to use deferred expressions instead (`#{expr}`) since they are not protected by this function. However, the bean validation specs do not allow the use of deferred expressions and Bean Validation implementations such as Apache BVal enforce this restriction by escaping them (`#{expr}` -> `\#{expr}`). For example:

```
// Java Bean Validation does not support EL expressions that look like JSP "deferred" expressions
return expressionFactory.createValueExpression(context,
    EvaluationType.DEFERRED.regex.matcher(message).replaceAll("\\$0"), String.class).getValue(context)
    .toString();
```

This way the underlying EL processor should ignore them and treat them as literal expressions.

However, a bug in the Eclipse EE4J EL implementation allows attackers to bypass this protection with a payload such as `FOO $\#{payload}`. This should be considered a literal expression and not be evaluated, however this is not the case.

The bug seems to be in the parser's grammar. Specifically in:

```
<DEFAULT> TOKEN :
{
  < LITERAL_EXPRESSION:
    ((~["\\", "$", "#"])
      | ("\\" ("\\" | "$" | "#"))
      | ("$" ~["{", "$", "#"])
      | ("#" ~["{", "$", "#"])
    )+
    | "$"
    | "#"
  >
|
  < START_DYNAMIC_EXPRESSION: "${" > {stack.push(DEFAULT);}: IN_EXPRESSION
|
  < START_DEFERRED_EXPRESSION: "#{" > {stack.push(DEFAULT);}: IN_EXPRESSION
}
```

A `$` or `#` followed by a character that is not `{`, `$` or `#` will be treated as a literal expression. The interesting case is where the character following the `$` or `#` chars is a backslash. The parser will consume the backslash as part of the literal expression and will leave the character that follows it unescaped.

#### Impact

This issue may lead to mitigation bypasses that allow for remote code execution in affected applications.

## Credit

This issue was discovered and reported by GHSL team member @pwntester (Alvaro Muñoz).

## Contact

You can contact the GHSL team at `securitylab@github.com`, please include the `GHSL-2020-021` in any communication regarding this issue.