

Java安全漫谈 - 13.Java中动态加载字节码的那些方法

这是[代码审计知识星球](#)中Java安全的第十三篇文章。

经过前面几篇反序列化文章的讲解，相信大家对反序列化漏洞有了一点初步的认识，可能会有人自己翻翻ysoserial的源码。这时候你可能会看到很多次 `Gadgets.createTemplatesImpl(command)`，另外你也许曾在fastjson等漏洞的利用中看到过 `TemplatesImpl` 这个类，它究竟是什么，为何出镜率这么高呢？

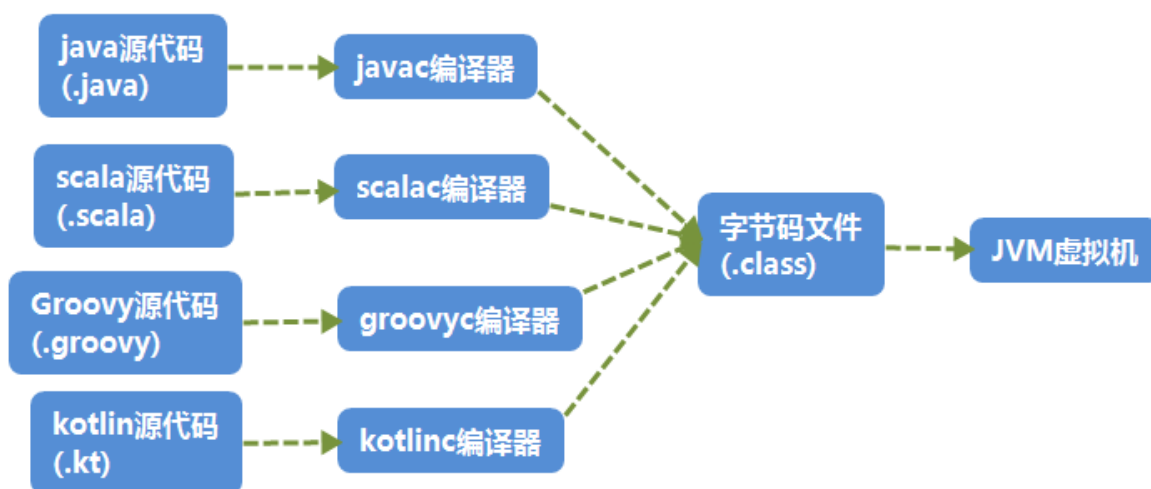
本文暂时先把反序列化漏洞搁一搁，我们来认识下Java中支持动态加载“字节码”的那些方法。

什么是Java的“字节码”

严格来说，Java字节码（ByteCode）其实仅仅指的是Java虚拟机执行使用的一类指令，通常被存储在.class文件中。

众所周知，不同平台、不同CPU的计算机指令有差异，但因为Java是一门跨平台的编译型语言，所以这些差异对于上层开发者来说是透明的，上层开发者只需要将自己的代码编译一次，即可运行在不同平台的JVM虚拟机中。

甚至，开发者可以用类似Scala、Kotlin这样的语言编写代码，只要你的编译器能够将代码编译成.class文件，都可以在JVM虚拟机中运行：



但是，本文中所说的“字节码”，可以理解的更广义一些——**所有能够恢复成一个类并在JVM虚拟机里加载的字节序列，都在我们的探讨范围内**。所以，如果你阅读到后面，发现我讲的不是狭义的“Java字节码”，请不要有疑虑。

利用URLClassLoader加载远程class文件

Java的ClassLoader来用来加载字节码文件最基础的方法，我们曾在本系列最开头反射相关部分讲过：

`ClassLoader` 是什么呢？它就是一个“加载器”，告诉Java虚拟机如何加载这个类。Java默认的 `ClassLoader` 就是根据类名来加载类，这个类名是类完整路径，如 `java.lang.Runtime`。

`ClassLoader`的概念的确不是一语概之的，所以我本文也不做深入分析，本文要说到的是这个 `ClassLoader`：`URLClassLoader`。

`URLClassLoader` 实际上是我们平时默认使用的 `AppClassLoader` 的父类，所以，我们解释 `URLClassLoader` 的工作过程实际上就是在解释默认的Java类加载器的工作流程。

正常情况下，Java会根据配置项 `sun.boot.class.path` 和 `java.class.path` 中列举到的基础路径（这些路径是经过处理后的 `java.net.URL` 类）来寻找.class文件来加载，而这个基础路径有分为三种情况：

- URL未以斜杠 / 结尾，则认为是一个JAR文件，使用 `JarLoader` 来寻找类，即为在Jar包中寻找.class文件
- URL以斜杠 / 结尾，且协议名是 `file`，则使用 `FileLoader` 来寻找类，即为在本地文件系统中寻找.class文件
- URL以斜杠 / 结尾，且协议名不是 `file`，则使用最基础的 `Loader` 来寻找类

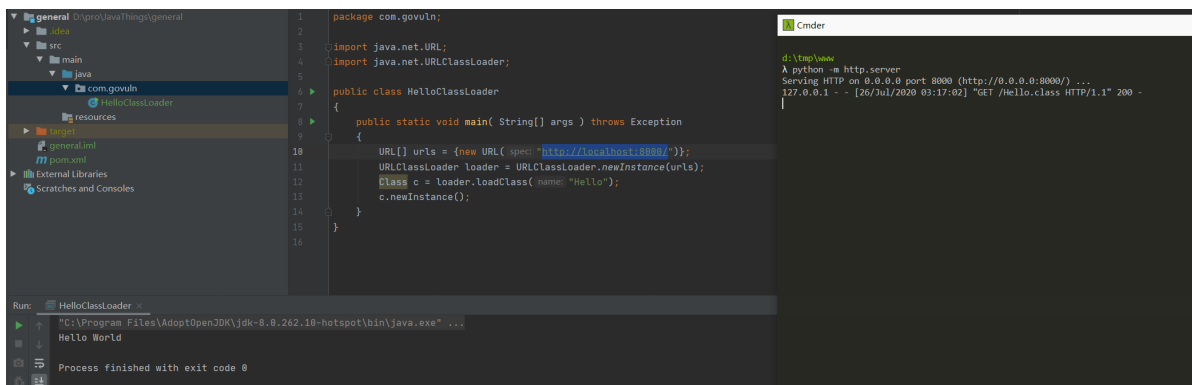
我们正常开发的时候通常遇到的是前两者，那什么时候才会出现使用 `Loader` 寻找类的情况呢？当然是非 `file` 协议的情况下，最常见的就是 `http` 协议。

这里其实会涉及到一个问题：“Java的URL究竟支持哪些协议”，但这并不是本文的重点，以后我们肯定会在SSRF相关的章节中说到，所以这里就不深入研究了。

我们可以使用HTTP协议来测试一下，看Java是否能从远程HTTP服务器上加载.class文件：

```
1 package com.govu1n;
2
3 import java.net.URL;
4 import java.net.URLClassLoader;
5
6 public class HelloClassLoader
7 {
8     public static void main( String[] args ) throws Exception
9     {
10         URL[] urls = {new URL("http://localhost:8000/")};
11         URLClassLoader loader = URLClassLoader.newInstance(urls);
12         Class c = loader.loadClass("Hello");
13         c.newInstance();
14     }
15 }
```

我们编译一个简单的HelloWorld程序，放在 `http://localhost:8000/Hello.class`：

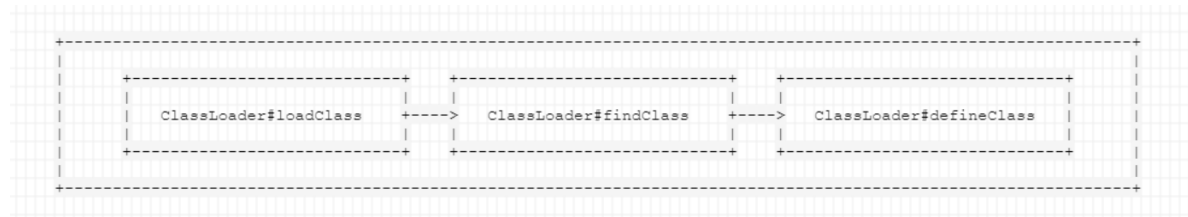


成功请求到我们的 `/Hello.class` 文件，并执行了文件里的字节码，输出了"Hello World"。

所以，作为攻击者，如果我们能够控制目标Java ClassLoader的基础路径为一个http服务器，则可以利用远程加载的方式执行任意代码了。

利用 `ClassLoader#defineClass` 直接加载字节码

上一节中我们认识到了如何利用URLClassLoader加载远程class文件，也就是字节码。其实，不管是加载远程class文件，还是本地的class或jar文件，Java都经历的是下面这三个方法调用：



其中：

- `loadClass` 的作用是从已加载的类缓存、父加载器等位置寻找类（这里实际上是双亲委派机制），在前面没有找到的情况下，执行 `findClass`
- `findClass` 的作用是根据基础URL指定的方式来加载类的字节码，就像上一节中说到的，可能会在本地文件系统、jar包或远程http服务器上读取字节码，然后交给 `defineClass`
- `defineClass` 的作用是处理前面传入的字节码，将其处理成真正的Java类

所以可见，真正核心的部分其实是 `defineClass`，他决定了如何将一段字节流转变成一个Java类，Java默认的 `ClassLoader#defineClass` 是一个native方法，逻辑在JVM的C语言代码中。

我们可以编写一个简单的代码，来演示如何让系统的 `defineClass` 来直接加载字节码：

```
1 package com.govu1n;
2
3 import java.lang.reflect.Method;
4 import java.util.Base64;
5
6 public class HelloDefineClass {
7     public static void main(String[] args) throws Exception {
8         Method defineClass =
9         ClassLoader.class.getDeclaredMethod("defineClass", String.class,
10         byte[].class, int.class, int.class);
11         defineClass.setAccessible(true);
12
13         byte[] code =
14         Base64.getDecoder().decode("yv66vgAADQAGWoABgANCQA0AA8IABAKABEAegCAEwCAFAEA
15         Bjxpbm10PgEAAygpVgEABENVZGUBAA9Maw51TnVtYmVYVGFibGUBAApTb3VyY2VGawx1AQAKSGVs
16         bG8uamF2YQwABWAIBWAVDAAWABcBAAtIZWxsbyBxb3JzZACAGAWAGQAaAQAFSGVsbg8BABBqYXZh
17         L2xhbmcvT2JqZW50AQQAamF2YS9sYW5nL1N5c3R1bQEAA291dAEAFUXqYXZhL21vL1Byaw50U3Ry
18         ZWftOWEAE2phdmEvaw8vUHJpbmRTdHJlYW0BAAdwcm1udGxuAQAVKExqYXZhL2xhbmcvU3Ryaw5n
19         Oy1WACEABQAGAAAAAABAAEABWIAAEACQAAC0AAGABAAAADSq3AAGyAAISA7YABLEAAAAABAAoA
20         AAAOAMAAAAACAAQABAAMAAUAAQALAAAAgAM");
21
22         Class hello =
23         (Class)defineClass.invoke(ClassLoader.getSystemClassLoader(), "Hello", code,
24         0, code.length);
25         hello.newInstance();
26     }
27 }
```

注意一点，在 `defineClass` 被调用的时候，类对象是不会被初始化的，只有这个对象显式地调用其构造函数，初始化代码才能被执行。而且，即使我们将初始化代码放在类的static块中（在本系列文章第一篇中进行过说明），在 `defineClass` 时也无法被直接调用到。所以，如果我们要使用 `defineClass` 在目标机器上执行任意代码，需要想办法调用构造函数。

执行上述example，输出了Hello World：

```
1 package com.govuln;
2
3 import java.lang.reflect.Method;
4 import java.util.Base64;
5
6 public class HelloDefineClass {
7     public static void main(String[] args) throws Exception {
8         Method defineClass = ClassLoader.class.getDeclaredMethod("defineClass", String.class, byte[].class, int.class, int.class);
9         defineClass.setAccessible(true);
10
11         byte[] code = Base64.getDecoder().decode(
12             "src: \"yv66vgAAADQAGwoABqANCQA0AABIAKABAEgcAEwcAFAEABixpbm10PgEAAygpVgEABENVZGUBAA9MaW51TnVtYmVvV6FibGUBAApTb3VyY2V6aWx1AQAKSGVsG8uamF2YQwABwAIBwAVDAAV\n";
13         Class hello = (Class)defineClass.invoke(ClassLoader.getSystemClassLoader(), "Hello", code, 0, code.length);
14         hello.newInstance();
15     }
16 }
17
18 Run: HelloDefineClass
19 "C:\Program Files\AdoptOpenJDK\jdk-8.0.262.10-hotspot\bin\java.exe" ...
20 Hello World
21
22 Process finished with exit code 0
```

这里，因为系统的 `ClassLoader#defineClass` 是一个保护属性，所以我们无法直接在外访问，不得不使用反射的形式来调用。

在实际场景中，因为 `defineClass` 方法作用域是不开放的，所以攻击者很少能直接利用到它，但它却是我们常用的一个攻击链 `TemplatesImpl` 的基石。

利用 `TemplatesImpl` 加载字节码

虽然大部分上层开发者不会直接使用到 `defineClass` 方法，但是Java底层还是有一些类用到了它（否则他也没存在的价值了对吧），这就是 `TemplatesImpl`。

`com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl` 这个类中定义了一个内部类 `TransletClassLoader`：

```
1 static final class TransletClassLoader extends ClassLoader {
2     private final Map<String, Class> _loadedExternalExtensionFunctions;
3
4     TransletClassLoader(ClassLoader parent) {
5         super(parent);
6         _loadedExternalExtensionFunctions = null;
7     }
8
9     TransletClassLoader(ClassLoader parent, Map<String, Class> mapEF) {
10         super(parent);
11         _loadedExternalExtensionFunctions = mapEF;
12     }
13
14     public Class<?> loadClass(String name) throws ClassNotFoundException {
15         Class<?> ret = null;
16         // The _loadedExternalExtensionFunctions will be empty when the
17         // SecurityManager is not set and the FSP is turned off
18         if (_loadedExternalExtensionFunctions != null) {
19             ret = _loadedExternalExtensionFunctions.get(name);
20         }
21         if (ret == null) {
22             ret = super.loadClass(name);
23         }
24         return ret;
25     }
26
27     /**
28      * Access to final protected superclass member from outer class.
29      */
30 }
```

```
30     Class defineClass(final byte[] b) {
31         return defineClass(null, b, 0, b.length);
32     }
33 }
```

这个类里重写了 `defineClass` 方法，并且这里没有显式地声明其定义域。Java 中默认情况下，如果一个方法没有显式声明作用域，其作用域为 `default`。所以也就是说这里的 `defineClass` 由其父类的 `protected` 类型变成了一个 `default` 类型的方法，可以被类外部调用。

我们从 `TransletClassLoader#defineClass()` 向前追溯一下调用链:

```
1 TemplatesImpl#getOutputProperties() -> TemplatesImpl#newTransformer() ->
  TemplatesImpl#getTransletInstance() -> TemplatesImpl#defineTransletClasses()
  -> TransletClassLoader#defineClass()
```

追到最前面两个方法 `TemplatesImpl#getOutputProperties()`、
`TemplatesImpl#newTransformer()`，这两者的作用域是public，可以被外部调用。我们尝试用
`newTransformer()` 构造一个简单的POC：

```

1 public static void main(String[] args) throws Exception {
2     // source: bytecodes/HelloTemplateImpl.java
3     byte[] code =
Base64.getDecoder().decode("yv66vgAAADQAIQoABgASCQATABQIABUKABYAFwcAGACAGQEA
CXRYyW5zzm9ybQEacihMY29tL3N1bi9vcmcvYXBhy2hlL3hhbGFuL2ludGvybmFsL3hz bHRjL0RPR
TTbtTGNvbS9zdW4vb3JnL2FwYWNoZS94bWVaw50ZXJuYXwwvc2VyawFsaXplci9TZXJpYXpxpemFO
aw9uSGFuZGxlclcspVgEABENVZGUBAA9Maw51TnvTymvYVGfIBGUBAapFeGnlCHRpb25zBWAaaQCm
KEXjb20vc3VuL29yzy9hcGFjaGUveGFsYW4vaw50ZXJuYXwwveHNsdGMvRE9NO0xjb20vc3VuL29y
zy9hcGFjaGUveG1sL2ludGvybmFsL2R0bS9EVE1BeGlzSXRLcmF0b3I7TGNVbs9zdW4vb3JnL2Fw
YWN0ZS94bWVaw50ZXJuYXwwvc2VyawFsaXplci9TZXJpYXpxpemFOaw9uSGFuZGxlclcspVgEABjxp
bm10PgEAAygpvGEAlNvdXjjZUZpbGUBABdIZWxsblRlbXBsYXRlc0ltcGwuamF2YQwADgAPBWAb
DAACAB0BABNIZWxsbyBUZWlwbgF0ZXNjbXBsBWAeDAAfACABABJIZWxsblRlbXBsYXRlc0ltcGWB
AEBjb20vc3VuL29yzy9hcGFjaGUveGFsYW4vaw50ZXJuYXwwveHNsdGMvcnVudGltZS9BYnn0cmFj
dFRyYW5zbGV0AQAsY29tL3N1bi9vcmcvYXBhy2hlL3hhbGFuL2ludGvybmFsL3hz bHRjL1RyYW5z
bGV0RXhjZXB0aw9uAQAAqamF2YS9syW5nL1NsC3RlbQEAA291dAEAFuxqYXZhL2l vL1Byaw50U3Ry
ZWftOWEAE2phdmEvaw8vUHJpbnRTdHJlYW0BAAdwcm1udGxuAQAVKExqYXZhL2xhbmcvU3Ryaw5n
OylWACEABQAGAAAAAADAAEABWIAAAIACQAAAABKAAAADAAAAABEAAAABAAoAAAAGAAEEAAAAIAASA
AAAAEAEADAABAACADQACAakAAAAAZAAAAABAAAAAGxAAAAAQAKAAAABGABAAAAACgALAAAABAABAwa
AQAOAA8AAQAJAAAAAQACAAEAAAAANKrCAabIAAhIDtgAESQAAAAEACgAAAA4AAwAAAAOABAAOAawa
DWABABAAAAACABE=");
4     TemplatesImpl obj = new TemplatesImpl();
5     setFieldValue(obj, "_bytecodes", new byte[][] {code});
6     setFieldValue(obj, "_name", "HelloTemplatesImpl");
7     setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
8
9     obj.newTransformer();
10 }

```

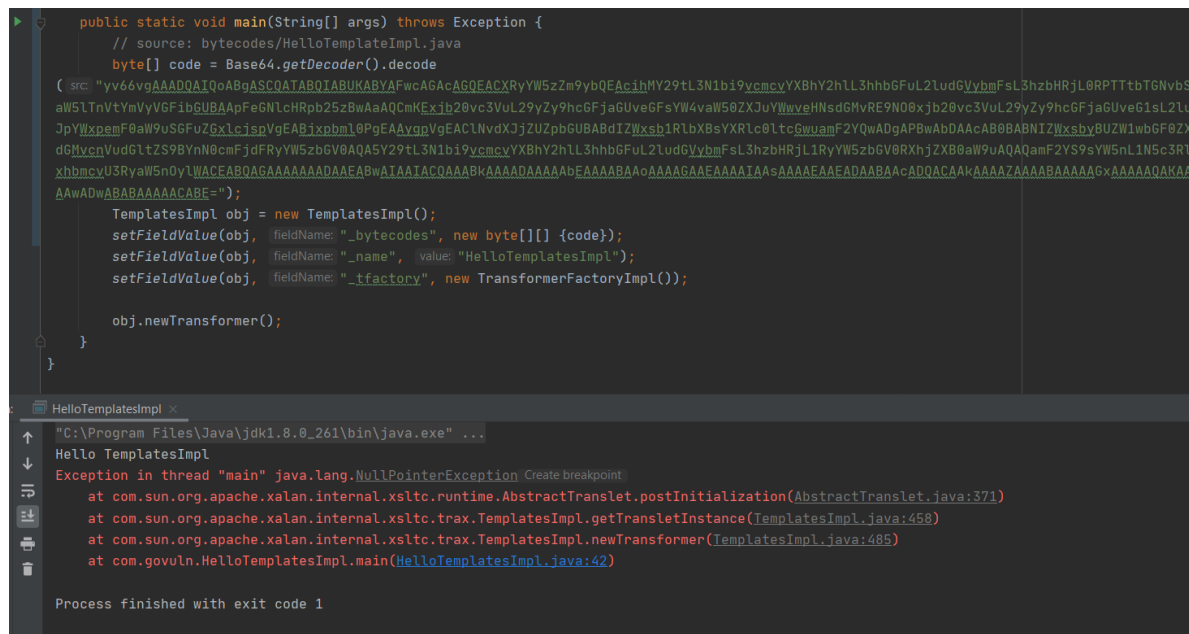
其中，`setFieldValue` 方法用来设置私有属性，可见，这里我设置了三个属性：`_bytecodes`、`_name` 和 `_tfactory`。`_bytecodes` 是由字节码组成的数组；`_name` 可以是任意字符串，只要不为null即可；`_tfactory` 需要是一个 `TransformerFactoryImpl` 对象，因为 `TemplatesImpl#defineTransletClasses()` 方法里有调用到 `_tfactory.getExternalExtensionsMap()`，如果是null会出错。

另外，值得注意的是，`TemplatesImpl` 中对加载的字节码是有一定要求的：这个字节码对应的类必须是 `com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet` 的子类。

所以，我们需要构造一个特殊的类：

```
1 import com.sun.org.apache.xalan.internal.xsltc.DOM;
2 import com.sun.org.apache.xalan.internal.xsltc.TransletException;
3 import com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet;
4 import com.sun.org.apache.xml.internal.dtm.DTMAxisIterator;
5 import com.sun.org.apache.xml.internal.serializer.SerializationHandler;
6
7 public class HelloTemplatesImpl extends AbstractTranslet {
8     public void transform(DOM document, SerializationHandler[] handlers)
9     throws TransletException {}
10
11     public void transform(DOM document, DTMAxisIterator iterator,
12     SerializationHandler handler) throws TransletException {}
13
14     public HelloTemplatesImpl() {
15         super();
16         System.out.println("Hello TemplatesImpl");
17     }
18 }
```

它继承了 `AbstractTranslet` 类，并在构造函数里插入Hello的输出。将其编译成字节码，即可被 `TemplatesImpl` 执行了：



在多个Java反序列化利用链，以及fastjson、jackson的漏洞中，都曾出现过 `TemplatesImpl` 的身影，这个系列后文中仍然会再次见到它的身影。

利用BCEL ClassLoader加载字节码

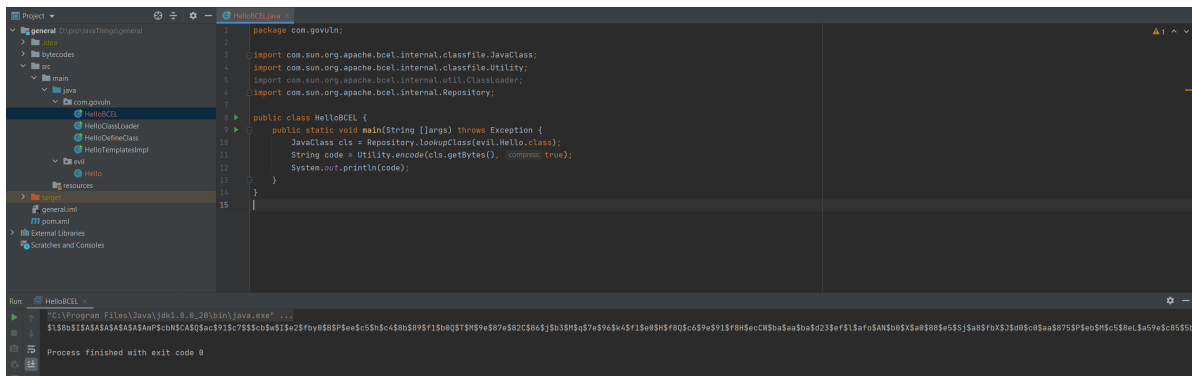
在本文第一节中，所有能够恢复成一个类并在JVM虚拟机里加载的字节序列，都在我们的探讨范围内。所以，bcel字节码也必然在我们的讨论范围内，且占据着比较重要的地位。

BCEL的全名应该是Apache Commons BCEL，属于Apache Commons项目下的一个子项目，但其因为被Apache Xalan所使用，而Apache Xalan又是Java内部对于JAXP的实现，所以BCEL也被包含在了JDK的原生库中。

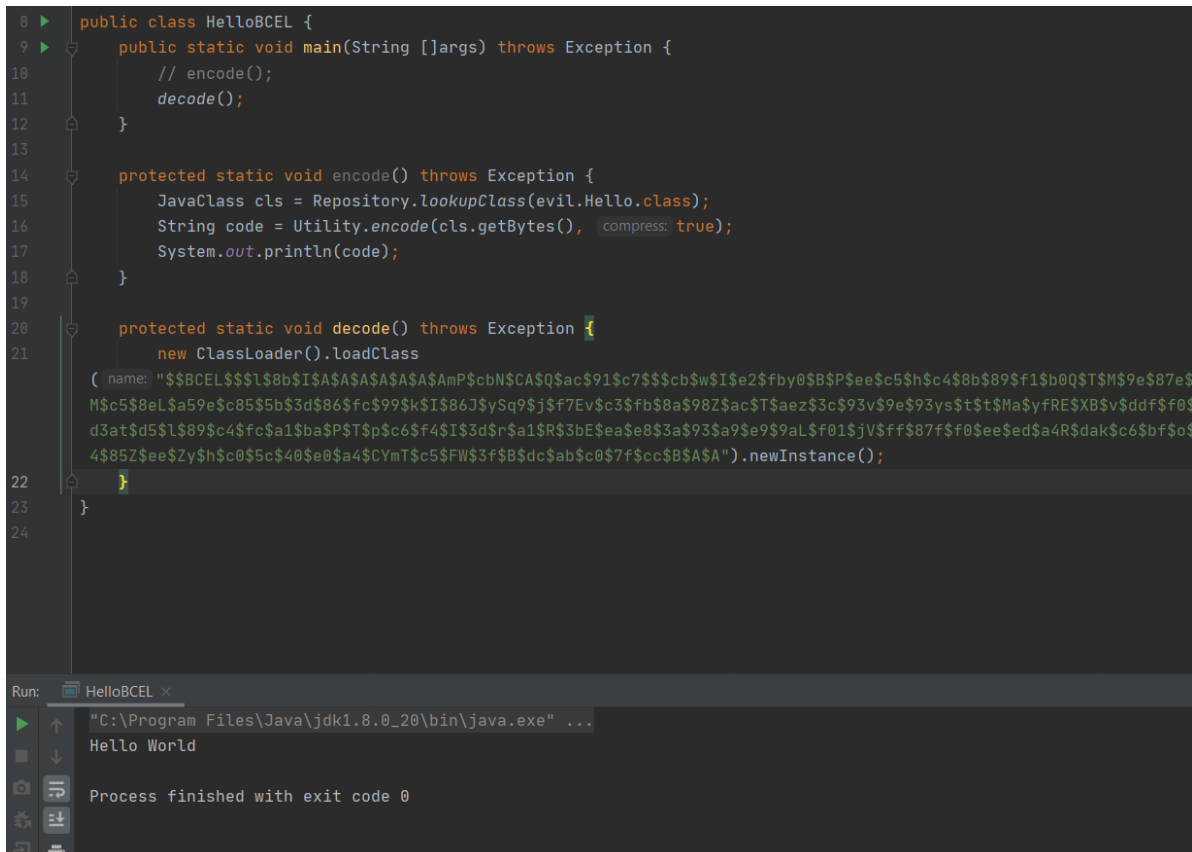
关于BCEL的详细介绍，请阅读我写的另一篇文章《[BCEL ClassLoader去哪了](#)》，建议阅读完这篇文章再来阅读本文。

我们可以通过BCEL提供的两个类 `Repository` 和 `Utility` 来利用：`Repository` 用于将一个Java Class先转换成原生字节码，当然这里也可以直接使用javac命令来编译java文件生成字节码；`Utility` 用于将原生的字节码转换成BCEL格式的字节码：

```
1 package com.govu1n;
2
3 import com.sun.org.apache.bcel.internal.classfile.JavaClass;
4 import com.sun.org.apache.bcel.internal.classfile.Utility;
5 import com.sun.org.apache.bcel.internal.Repository;
6
7 public class HelloBCEL {
8     public static void main(String []args) throws Exception {
9         JavaClass cls = Repository.lookupClass(evil.Hello.class);
10        String code = Utility.encode(cls.getBytes(), true);
11        System.out.println(code);
12    }
13 }
```



而BCEL ClassLoader用于加载这串特殊的“字节码”，并可以执行其中的代码：



BCEL ClassLoader在Fastjson等漏洞的利用链构造时都有被用到，其实这个类和前面的TemplatesImpl都出自于同一个第三方库，Apache Xalan。但是由于各种原因（详见前面所说的《[BCEL ClassLoader去哪了](#)》），在Java 8u251的更新中，这个ClassLoader被移除了，所以之后只能且用且珍惜了。

总结

本篇总结了Java漏洞利用中常见的几种加载“字节码”的方式，当然应该没有覆盖全，大家如果有一些新的方法，也可以在评论里发出来，之后可以补充补充。