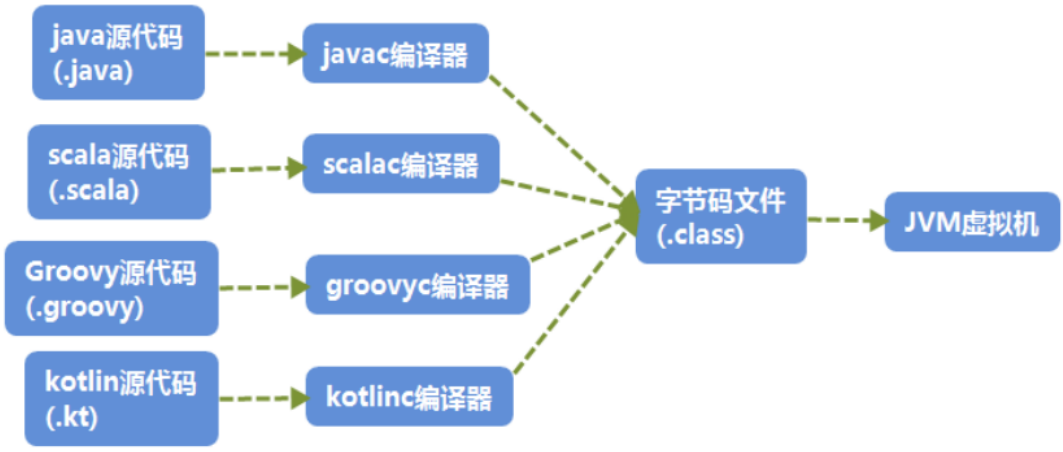
Java中动态加载字节码

什么是Java的字节码

- Java字节码（ByteCode）是Java虚拟机执行使用的一类指令，通常被存储在`.class`文件中
- 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
- 开发者可以用类似`Scala`、`Kotlin`这样的语言编写代码，只要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



- 字节码：所有能够恢复成一个类并在JVM虚拟机里加载的字节序列

利用URLClassLoader加载远程class文件

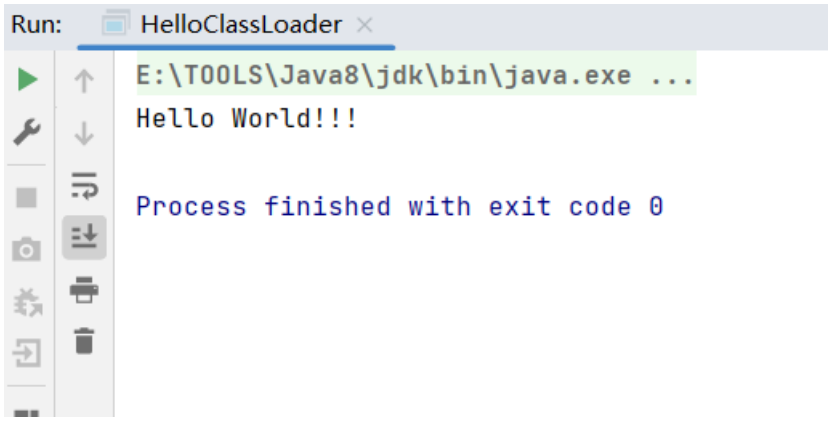
- Java的`ClassLoader`是用来加载字节码文件最基础的方法
- `ClassLoader`就是一个加载器，告诉Java虚拟机如何加载这个类，Java默认的`ClassLoader`就是根据类名来加载类，这个类名是类完整路径，如`java.lang.Runtime`
- `URLClassLoader`上是平时默认使用的`AppClassLoader`的父类，解释`URLClassLoader`的工作过程实际上就是在解释默认的Java类加载器的工作流程
- 正常情况下，Java会根据配置项`sun.boot.class.path`和`java.class.path`中列举到的基础路径（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
- 正常开发的时候通常遇到的是前两者，在非`file`协议的情况下会出现使用Loader寻找类的情况，最常见的就是http协议

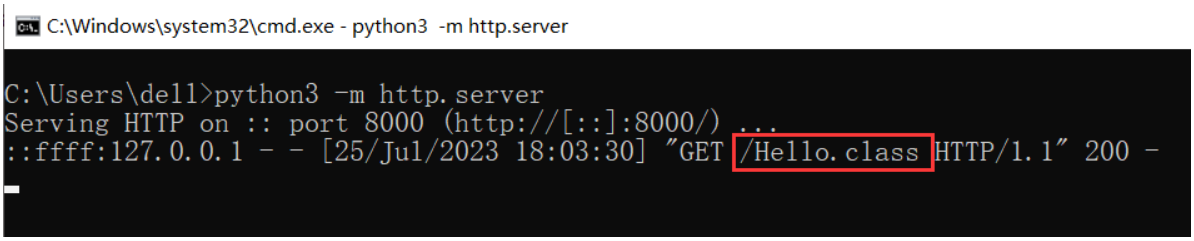
- 使用http协议测试一下Java能否从远程http服务器上加载`.class`文件：

```
package ByteCode;

import java.net.URL;
import java.net.URLClassLoader;
public class HelloClassLoader
{
      public static void main( String[] args ) throws Exception
     {
            URL[] urls = {new URL("http://localhost:8000/")};
            URLClassLoader loader = URLClassLoader.newInstance(urls);
            Class c = loader.loadClass("Hello");
            c.newInstance();
     }
}
```

- 成功请求到`Hello.class`文件，并执行了文件里的字节码，输出了`Hello World`





- 作为攻击者，如果能够控制目标`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

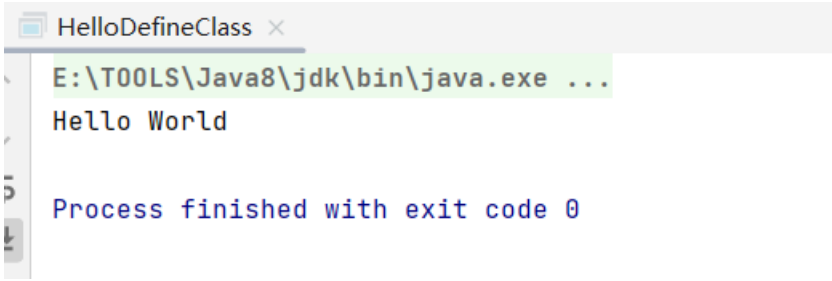
利用ClassLoader#defineClass直接加载字间码

- 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：
    - `ClassLoader#loadClass -> ClassLoader#findClass -> ClassLoader#defineClass`
    - `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指定的方式加载类的字节码，可能会在本地文件系统、jar包、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，将其转化为真正的Java类
- defineClass直接加载字节码例子：

```
package ByteCode;

import java.lang.reflect.Method;
import java.util.Base64;
public class HelloDefineClass {
      public static void main(String[] args) throws Exception {
            Method defineClass = ClassLoader.class.getDeclaredMethod("defineClass",
String.class, byte[].class, int.class, int.class);
            defineClass.setAccessible(true);
            byte[] code =
Base64.getDecoder().decode("yv66vgAAADQAGwoABgANCQAOAA8IABAKABEAEgcAEwcAFAEABjxpbmlOPgEAA
ygpVgEABENvZGUBAA9MaW51TnVtYmVyVGFibGUBAApTb3VyY2VGaWxlAQAKSGVsbG8uamF2YQwABwAIBwAVDAAWAB
cBAAtIZWxsbyBXb3JsZAcAGAwAGQAaAQAFSGVsbG8BABBqYXZhL2xhbmcvT2JqZWNOAQAQamF2YS9sYW5nL1N5c3R
lbQEAA291dAEAFUxqYXZhL2lvL1ByaW50U3RyZWFtOwEAE2phdmEvaW8vUHJpbnRTdHJ1YWOBAAdwcmludGxuAQAV
KExqYXZhL2xhbmcvU3RyaW5nOylWACEABQAGAAAAAAABAAEABwAIAAEACQAAACOAAgABAAAADSq3AAGyAAISA7YAB
LEAAAABAAoAAAAOAAMAAAACAAQABAAMAAUAAQALAAAAAgAM");
            Class hello = (Class)defineClass.invoke(ClassLoader.getSystemClassLoader(),
"Hello", code, 0, code.length);
            hello.newInstance();
      }
}
```



- 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
- 即使我们将初始化代码放在类的`static`块中，在`defineClass`时也无法被直接调用到，所以，如果要使用`defineClass`在目标机器上执行任意代码，需要想办法调用构造函数
- 因为系统的`ClassLoader#defineClass`是一个保护属性，所以无法直接在外部访问，应该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，比如：`TemplatesImpl`
- `com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl`这个类中定义了一个内部类`TransletClassLoader`：

```
static final class TransletClassLoader extends ClassLoader {
    private final Map<String,Class> _loadedExternalExtensionFunctions;

     TransletClassLoader(ClassLoader parent) {
         super(parent);
        _loadedExternalExtensionFunctions = null;
    }

    TransletClassLoader(ClassLoader parent,Map<String, Class> mapEF) {
        super(parent);
        _loadedExternalExtensionFunctions = mapEF;
    }

    public Class<?> loadClass(String name) throws ClassNotFoundException {
        Class<?> ret = null;
        // The _loadedExternalExtensionFunctions will be empty when the
        // SecurityManager is not set and the FSP is turned off
        if (_loadedExternalExtensionFunctions != null) {
            ret = _loadedExternalExtensionFunctions.get(name);
        }
        if (ret == null) {
            ret = super.loadClass(name);
        }
        return ret;
     }

    Access to final protected superclass member from outer class.
    Class defineClass(final byte[] b) { return defineClass( name: null, b,  off: 0, b.length); }
}
```

- 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
- 从`TransletClassLoader#defineClass()`向前追溯一下调用链：

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

```java
    public synchronized Properties getOutputProperties() {
        try {
            return newTransformer().getOutputProperties();
        }
        catch (TransformerConfigurationException e) {
            return null;
        }
    }

public synchronized Transformer newTransformer()
    throws TransformerConfigurationException
{
    TransformerImpl transformer;

    transformer = new TransformerImpl(getTransletInstance(), _outputProperties,
        _indentNumber, _tfactory);

    if (_uriResolver != null) {
        transformer.setURIResolver(_uriResolver);
    }

    if (_tfactory.getFeature(XMLConstants.FEATURE_SECURE_PROCESSING)) {
        transformer.setSecureProcessing(true);
    }
    return transformer;
}

private Translet getTransletInstance()
    throws TransformerConfigurationException {
    try {
        if (_name == null) return null;

        if (_class == null) defineTransletClasses();

        // The translet needs to keep a reference to all its auxiliary
        // class to prevent the GC from collecting them
        AbstractTranslet translet = (AbstractTranslet) _class[_transletIndex].newInstance();
        translet.postInitialization();
        translet.setTemplates(this);
        translet.setOverrideDefaultParser(_overrideDefaultParser);
        translet.setAllowedProtocols(_accessExternalStylesheet);
        if (_auxClasses != null) {
            translet.setAuxiliaryClasses(_auxClasses);
        }

        return translet;
    }
    catch (InstantiationException e) {
        ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
        throw new TransformerConfigurationException(err.toString());
    }
    catch (IllegalAccessException e) {
        ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
        throw new TransformerConfigurationException(err.toString());
    }
}
```

```
private void defineTransletClasses()
    throws TransformerConfigurationException {

    if (_bytecodes == null) {
        ErrorMsg err = new ErrorMsg(ErrorMsg.NO_TRANSLET_CLASS_ERR);
        throw new TransformerConfigurationException(err.toString());
    }

    TransletClassLoader loader = (TransletClassLoader)
        AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                return new TransletClassLoader(ObjectFactory.findClassLoader(),_tfactory.getEx
            }
        });

    try {
        final int classCount = _bytecodes.length;
        _class = new Class[classCount];

        if (classCount > 1) {
            _auxClasses = new HashMap<>();
        }

        for (int i = 0; i < classCount; i++) {
            _class[i] = loader.defineClass(_bytecodes[i]);
            final Class superClass = _class[i].getSuperclass();
```

- 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，尝试用`newTransformer()`构造一个简单的POC：

```
package ByteCode;

import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
import com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;
import java.lang.reflect.Field;
import java.util.Base64;

public class TestTemplatesImpl {
      public static void main(String[] args) throws Exception{
```

```
            byte[] code =
Base64.getDecoder().decode("yv66vgAAADQAIQoABgASCQATABQIABUKABYAFwcAGAcAGQEACXRyYW5zZm9yb
QEAcihMY29tL3N1bi9vcmcvYXBhY2hlL3hhbGFuL2ludGVybmFsL3hzbHRjLORPTTtbTGNvbS9zdW4vb3JnL2FwYW
NoZS94bWwvaW50ZXJuYWwvc2VyaWFsaXplci9TZXJpYWxpemFOaW9uSGFuZGxlcjspVgEABENvZGUBAA9MaW51TnV
tYmVyVGFibGUBAApFeGNlcHRpb25zBwAaAQCmKExjb2Ovc3VuL29yZy9hcGFjaGUveGFsYW4vaW50ZXJuYWwveHNs
dGMvRE9NOOxjb2Ovc3VuL29yZy9hcGFjaGUveG1sL2ludGVybmFsL2ROObS9EVE1BeGlzSXRlcmFOb3I7TGNvbS9zd
W4vb3JnL2FwYWNoZS94bWwvaW50ZXJuYWwvc2VyaWFsaXplci9TZXJpYWxpemFOaW9uSGFuZGxlcjspVgEABjxpbm
lOPgEAAygpVgEAClNvdXJjZUZpbGUBABdIZWxsb1RlbXBsYXRlc0ltcGwuamF2YQwADgAPBwAbDAAcABOBABNIZWx
sbyBUZW1wbGFOZXNJbXBsBwAeDAAfACABABJIZWxsb1RlbXBsYXRlc0ltcGwBAEBjb2Ovc3VuL29yZy9hcGFjaGUv
eGFsYW4vaW50ZXJuYWwveHNsdGMvcnVudGltZS9BYnNOcmFjdFRyYW5zbGVOAQA5Y29tL3N1bi9vcmcvYXBhY2hlL
3hhbGFuL2ludGVybmFsL3hzbHRjL1RyYW5zbGVORXhjZXBOaW9uAQAQamF2YS9sYW5nL1N5c3R1bQEAA291dAEAFU
xqYXZhL2lvL1ByaW50U3RyZWFtOwEAE2phdmEvaW8vUHJpbnRTdHJ1YWOBAAdwcmludGxuAQAVKExqYXZhL2xhbmc
vU3RyaW5nOylWACEABQAGAAAAAAADAAEABwAIAAIACQAAABkAAAADAAAAAbEAAAABAAoAAAAGAAEAAAAIAAsAAAAE
AAEADAABAAcADQACAAkAAAAZAAAABAAAAAGxAAAAAQAKAAAABgABAAAACgALAAAABAABAAwAAQAOAA8AAQAJAAAAL
QACAAEAAAANKrcAAbIAAhIDtgAEsQAAAAEACgAAAA4AAwAAAAOABAAOAAwADwABABAAAAAACABE=");
            TemplatesImpl obj = new TemplatesImpl();
            setFieldValue(obj, "_bytecodes", new byte[][] {code});
            setFieldValue(obj, "_name", "HelloTemplatesImpl");
            setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
            obj.newTransformer();
    }
     public static void setFieldValue(Object obj, String fieldName, Object value) throws
Exception {
            Field field = obj.getClass().getDeclaredField(fieldName);
            field.setAccessible(true);
            field.set(obj, value);
    }
}
```

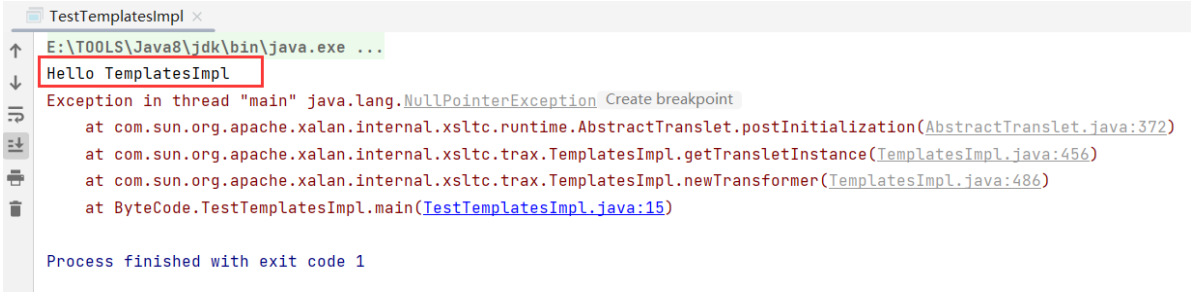
- 其中，`setFieldValue`方法用来设置私有属性，这里设置了三个属性：`_bytecodes`、`_name`和`_tfactory`
- `_bytecodes`是由字节码组成的数组；`_name`可以是任意字符串，只要不为null即可；`_tfactory`需要是一个`TransformerFactoryImpl`对象，因为`TemplatesImpl#defineTransletClasses()`方法里有调用到`_tfactory.getExternalExtensionsMap()`，如果是null会出错
- `TemplatesImpl`中加载的字节码对应的类必须是`com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet`的子类
- 所以，需要构造一个特殊的类：

```
package ByteCode;

import com.sun.org.apache.xalan.internal.xsltc.DOM;
import com.sun.org.apache.xalan.internal.xsltc.TransletException;
```

```
import com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet;
import com.sun.org.apache.xml.internal.dtm.DTMAxisIterator;
import com.sun.org.apache.xml.internal.serializer.SerializationHandler;
public class HelloTemplatesImpl extends AbstractTranslet {
      public void transform(DOM document, SerializationHandler[] handlers)
                  throws TransletException {}
      public void transform(DOM document, DTMAxisIterator iterator,
                                    SerializationHandler handler) throws
TransletException {}
      public HelloTemplatesImpl() {
            super();
            System.out.println("Hello TemplatesImpl");
      }
}
```

- 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

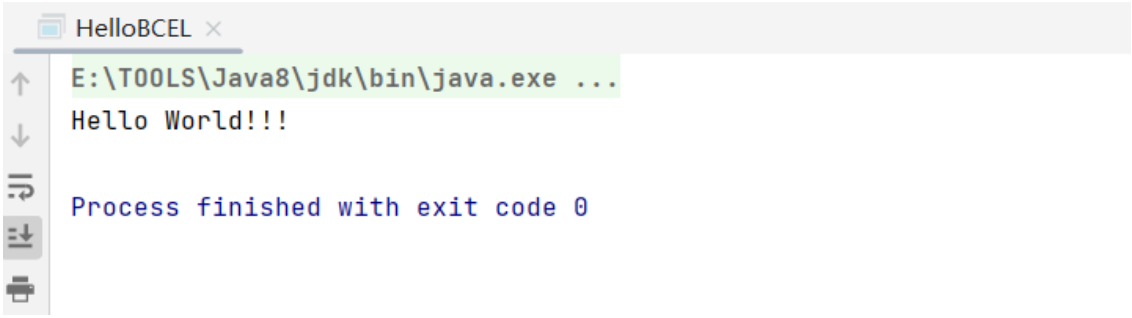
利用BCEL ClassLoader加载字节码

- `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
- 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
- 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

```
package ByteCode;

import com.sun.org.apache.bcel.internal.classfile.JavaClass;
import com.sun.org.apache.bcel.internal.classfile.Utility;
import com.sun.org.apache.bcel.internal.Repository;
```

```
import com.sun.org.apache.bcel.internal.util.ClassLoader;
public class HelloBCEL {
    public static void main(String []args) throws Exception {
        //encode();
        decode();
    }
    protected static void encode() throws Exception{
        JavaClass cls = Repository.lookupClass(Hello.class);
        String code = Utility.encode(cls.getBytes(), true);
        System.out.println(code);
    }
    protected static void decode() throws Exception{
        new
ClassLoader().loadClass("$$BCEL$$"+"$1$8b$I$A$A$A$A$A$AmP$cbN$c30$Q$i$b7i$d3$86$f4AKy$8
3$c8$ad$e5$40$$$dc$8a$40$C$Jq$88$A$a9$a8$9c$9d$d4$w$ae$9c$Y$a5$OR$3f$L$O$mq$eO$D$f8$u$c4$
sp$aO$S$5ei$ed$99$d9$j$af$fd$f9$f5$fe$B$eO$Y$fb$O$aah$d7$b1$8aN$N$5d$Hk$e8$d9X$b7$b1$c1P$
3d$91$894$a7$M$e5$fe$60$cc$60$5d$e8$89$60h$F2$R$d7Y$i$8a$f4$8e$87$8a$98N$aO$p$ae$c6$3c$95
9$fe$r$z$f3$m$e7$M$ed$eO$7caD$de$e9_$J$a5$f4$90$c1$Z$e9$y$8d$c4$a5$cc$cb$9c$82$3d$9a$f1$t
$ee$c2F$cd$c6$a6$8b$zl34$L$e5$e0$5e$a7j$e2y$9e$8d$j$X$bb$d8$pa$d9$91$ee$c8$bb$7d$c5$93$a9
$7f$T$ceDd$96$a8$d1bnDL$8f$d0$Z$J$bd$aOP$a4$f6oS$99$98$91I$F$8fi$a8$ee$3f4$83$fd$98$p$95P
_$3f$f8ci$88$9e$O$Hcx$a8$d0$f7$e5$8bQ$d0$fc$94$eb$84$ceP$a2$T$d08$7c$D$7bA$a9S$7e$85$f5LD
$J$O$e5$s$ca$94$z$8a$K$5c2X$n$e4$fe4$d0$de$u$ec$9aEu$eb$h$bbV$c77$a9$B$A$A").newInstance(
);
    }
}
```



- `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`，但是在Java 8u251的更新中，这个`ClassLoader`被移除了