LandGrey's Blog

Spring Boot Fat Jar 写文件漏洞到稳定 RCE 的探索

3年前 • 原创

0x00: 背景

现在生产环境部署 `spring boot` 项目一般都是将其打包成一个 `FatJar`，即把所有依赖的第三方 jar 也打包进自身的 `app.jar` 中，最后以 `java -jar app.jar` 形式来运行整个项目。

运行时项目的 `classpath` 包括 `app.jar` 中的 `BOOT-INF/classes` 目录和 `BOOT-INF/lib` 目录下的所有 jar，因此无法在其运行的时候往 classpath 中增加文件。并且现阶段 spring boot 项目多以 RESTful API 接口形式向外提供服务，很少会动态解析 jsp 和其他外部模版文件，直接 webshell 文件的情况一般不会出现。

这样就导致了一个不太好解决但又经常有人来问的有趣问题：一个正在运行中的 **spring boot** 项目如果存在本地任意写文件漏洞，怎么升级成 **RCE** 漏洞？

搜索...

分类

代码审计 (9)

其他 (5)

安全开发 (2)

渗透测试 (6)

红队战争 (2)

最新文章

spring-security 三种情况下的认证绕过

Spring Boot Fat Jar 写文件漏洞到稳定 RCE 的探索

目前口口相传和网络上能查到的通用方法基本就是通过写 linux `crontab` 计划任务文件、替换 `so/dll` 系统文件进行劫持等一些操作系统层面的东西来实现 RCE。实际环境下因为网络联通性、文件权限等各种条件限制，都不是特别好用，所以找一个 java 代码层面的利用方法就显得很有必要。

0x01: 类装载与类初始化

现在大家很多时候会把 类装载 **(Class loading)** 和 类初始化 **(Class initialization)** 的概念混淆在一起，并称为类加载，平常讨论时不需要特意的区别对待，但在比较精细化的漏洞利用中，有必要先清晰的理解两者的差别。 

类装载是由 jvm 的不同 ClassLoader，包括 `Bootstrap Classloder`、`Extention ClassLoader`、`App ClassLoader` 和用户自定义的 `Classloder` 完成的。类装载通常是一个 Class 在字节码中引用另一个 Class 时被动触发的，也有通过 **Classloder loadClass** 和 **Class forName** 等方式主动触发的。

在 java 程序运行的命令行中使用 `-XX:+TraceClassLoading`，可以观察到像如下在控制台输出的类装载过程日志：

```
[Opened **/jre/lib/rt.jar]
[Loaded java.lang.Object from **/jre/lib/rt.jar]
[Loaded java.io.Serializable from **/jre/lib/rt.
[Loaded java.lang.Comparable from **/jre/lib/rt.
[Loaded java.lang.CharSequence from **/jre/lib/r
......
```

其中里面的 `Opened` 操作代表打开指定文件，通常表示第一次读取相关字节码到内存；`Loaded` 操作代表将读取的指定类的字节码进行装载。

我仔细跟踪了下 jdk 的类装载过程，发现当经过 `java.lang.Classloder.defineClass` 方法的时

spring actuator restart logging.config rce

Apollo 配置中心未授权获取配置漏洞利用

利用 intercetor 注入 spring 内存 webshell

xxl-job 执行器 RESTful API 未授权访问 RCE

Beetl 模版引擎执行任意 Java 代码的方法

使用 MAT 查找 spring heapdump 中的密码明文

bypass openrasp SpEL RCE 的过程及思考

通过mysql jdbc 反序列化触发的 SpringBoot RCE 新利用方法

标签

SPRING PHP

红队攻防 分享

JAVA PYTHON

备忘 随笔

候，即读取字节码并定义Class类型后，类装载 `Loaded` 操作就算完成了。

类初始化（也可以叫类实例化）一定发生在类装载之后，当调用类中的静态属性或者静态方法时，会被动触发类的初始化；调用 `new` 关键词、`newInstance` 方法、`Class.forName` 等方法时，会主动触发类的初始化。

用一小段简单的代码来形象的区别类装载和类初始化：

- loadClass 类装载

```
classLoader.loadClass(className);
```

- loadClass 类初始化

```
classLoader.loadClass(className).newInstance();
```

- Class forName 类装载

```
Class.forName(className, false, classLoader);
```

- Class forName 类初始化

```
Class.forName(className, true, classLoader);
```

总结一下：

- 类装载并不一定会接着进行类初始化，不会执行类中的任何代码；
- 类初始化意味着类装载已经完成，会执行类中的一些特殊代码；

0x02: 类初始化与写文件漏洞的思考

关于类加载器的双亲委托机制，类初始化的时机等知识可以参考 深入探讨 Java 类加载器、深入理解Java类加载，本文中不做重复描述，人生苦短，减少搬运。

这里讲一下为什么要重点说类初始化：

- 类初始化的时候会执行 static 代码块、static 属性引用的方法等，还可能执行构造器中的代码
- 类装载和类初始化对于应用来说是个很平常的操作
    - 应用第一次启动时会装载很多类
    - 运行过程中也会因为一些代码第一次执行或者第一次异常报错而触发一些类的装载和初始化
    - 有些框架和程序员很喜欢用 `Class.forName(className)` 类似的，可以初始化类的代码去加载类，其中的 `className` 也容易被外部控制

- 如果找到一种方法可以控制程序在指定的写文件漏洞可控的文件范围内，主动触发初始化恶意的类，就有可能让写文件漏洞变成代码执行漏洞

0x03: 写什么文件

前文已经讲了，由于 spring boot 把所有资源打包进一个 jar 文件内，所以我们无法在运行时往其 classpath 等目录内写入任何东西，也就没办法通过写 webshell、替换模版资源文件、替换 class 文件等方式来达到 RCE。

那么自然而然的可以想到：没办法写入文件到应用的 classpath 目录，可以写入文件到更底层的 "系统的 **classpath** 目录"，即 JDK HOME 目录下。

经过一段时间的类加载过程观察、研究和不断的~~测试~~试错，我发现 jvm 为了避免一下加载太多暂时用不到或者以后都用不到的类，不会在一开始运行时把所有的 JDK HOME 目录下自带的 jar 文件全部加载到类中，存在 "懒加载" 行为。主要特征就是相关 jar 文件的 `Opened` 操作没有在一开始就发生，而是调用到相关类时被触发。

这样一来，结合 JDK HOME 目录下的系统 jar 文件、写文件漏洞，就很自然而然的想到，通过增加或替换 JDK HOME 目录下的系统 jar 文件，再主动触发 jar 文件里的类初始化来达到执行任意代码的方法。

JDK 在启动后，不会主动寻找 HOME 目录下新增的 jar 文件去尝试加载，所以也只有替换 JDK HOME 目录下原有的 jar 才可行，而且还得寻找系统启动后没有进行过 `Opened` 操作的系统 jar 文件。

然后又经过长时间的 debug 和不断~~测试~~试错，我发现程序代码中如果没有使用 `Charset.forName("GBK")` 类似的代码，默认就不会加载到 `/jre/lib/charsets.jar` 文件，同时为了兼容下文讲的更多的利用场景，所以我下面举例覆盖的系统 jar 文件都是 `charsets.jar` 。

其他类似没有一开始就被 `Opened` 操作的系统 jar 文件可能也有较好的姿势和效果，留给读者自行去扩展研究了。

另外，写文件一般需要指定写入文件的目录，而 JDK HOME 的目录一般不是固定的，所以可以提前收集好 JDK HOME 字典文件，在信息不透明的时候可以使用目录枚举的方式去批量上传测试。

测试时收集了几个常见 JDK 的目录，希望能有小伙伴一块补充下：

```
/usr/lib/jvm/java-8-oracle/jre/lib/
/usr/lib/jvm/java-1.8-openjdk/jre/lib/
/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/
```

0x04: 可控的主动类初始化

解决 "写什么文件" 的问题只是提供了 RCE 的土壤，想要发芽还得需要解决另一个重要问题：怎么主动触发可以控制类名的类初始化行为。

经过一番思考和调试，我暂时总结了 6 种比较常见的场景来完成可控的主动类初始化，希望也有小伙伴能提些建议和或者补充更多的场景。

下面主要介绍了下第一种场景，其余完整的测试场景 demo 代码托管在 spring-boot-upload-file-lead-to-rce-tricks 项目中，有兴趣的小伙伴可以看看。

零. spring 原生场景

spring-web 组件中的 `org.springframework.web.accept.HeaderContentNegotiationStrategy` 类中有下面一段代码（部分省略）：

```
public List<MediaType> resolveMediaTypes(NativeW
    String[] headerValueArray = request.getHeade
    ...
        try {
            List<MediaType> mediaTypes = MediaTy
            ...
        } catch (InvalidMediaTypeException var5)
            ...
        }
    }
}
```

其作用是对于每一次请求，spring 框架都会尝试解析 `Accept` 头的值，设置相应的字符集编码。

正常请求的一条 `Accept` 头示例可能如下：

```
Accept: text/plain, */*; q=0.01
```

可以利用的地方在 `MediaType.parseMediaTypes(headerValues)` 这行代码。经过一番调用会来到 **spring-core** 组件 `org.springframework.util.MimeTypeUtils` 类的 `parseMimeTypeInternal` 方法中（部分代码省略）：

```
private static MimeType parseMimeTypeInternal(St
    int index = mimeType.indexOf(';');
    ...
```

```
    Map<String, String> parameters = null;
    do {
        int nextIndex = index + 1;
        boolean quoted = false;
        while (nextIndex < mimeType.length()) {
            char ch = mimeType.charAt(nextIndex)
            if (ch == ';') {
                if (!quoted) {
                    break;
                }
            }
            else if (ch == '"') {
                quoted = !quoted;
            }
            nextIndex++;
        }
        String parameter = mimeType.substring(in
        if (parameter.length() > 0) {
            if (parameters == null) {
                parameters = new LinkedHashMap<>
            }
            int eqIndex = parameter.indexOf('=')
            if (eqIndex >= 0) {
                String attribute = parameter.sub
                String value = parameter.substri
                parameters.put(attribute, value)
            }
        }
        index = nextIndex;
    }
    while (index < mimeType.length());

    try {
        return new MimeType(type, subtype, param
    }
    ...
}
```

可以看到，上述代码经过对获取的 `Accept` 头格式的解析，最终用可控的 `type`、`subtype` 和 `parameters` 生成 `org.springframework.util.MimeType` 类型实例：

```
new MimeType(type, subtype, parameters)
```

跟入对应的有参构造函数中：

```
public MimeType(String type, String subtype, @Nu
    ...
```

```
    if (!CollectionUtils.isEmpty(parameters)) {
        Map<String, String> map = new LinkedCase
        parameters.forEach((attribute, value) ->
            checkParameters(attribute, value);
            map.put(attribute, value);
        });
        ...
    }
    ...
}
```

发现会利用

`checkParameters(attribute, value)` 对 `parameters` 进行解析

```
protected void checkParameters(String attribute,
    ...
    if (PARAM_CHARSET.equals(attribute)) {
        value = unquote(value);
        Charset.forName(value);
    }
    ...
}
```

然后就看到 `Charset.forName(value)` 用来主动加载字符集的代码，剩下的部分就和 **一. fastjson 最新版(目前是 1.2.76)默认配置场景** 后面讲的一致了。

在实际的环境中，替换过 `charsets.jar` 后，用如下的数据包就可触发 RCE 了 ^_^:

```
GET / HTTP/1.1
Accept: text/html;charset=GBK
```

一. fastjson 最新版(目前是 1.2.76)默认配置场景

选择替换 `charsets.jar` 文件可以绕过 fastjson 最新版的 autoType 限制。

以 fastjson 1.2.76 举例，

`com.alibaba.fastjson.parser.ParserConfig` 类中有如下代码:

```
if (clazz == null) {
    clazz = this.deserializers.findClass(typeNam
}
```

实际的 `findClass` 方法会枚举 `this.buckets` 中保存的 `IdentityHashMap<Type, ObjectDeserializer>` 类型键值对

```
public Class findClass(String keyString) {
    for(int i = 0; i < this.buckets.length; ++i)
        IdentityHashMap.Entry bucket = this.buck
        if (bucket != null) {
            for(IdentityHashMap.Entry entry = bu
                Object key = bucket.key;
                if (key instanceof Class) {
                    Class clazz = (Class)key;
                    String className = clazz.get
                    if (className.equals(keyStri
                        return clazz;
                    }
                }
            }
        }
    }

    return null;
}
```

而 `java.nio.charset.Charset` 类名正好在白名单中，所以可以直接返回 clazz。并且在 fastjson `com.alibaba.fastjson.serializer.MiscCodec` 类的后续处理中有以下代码

```
else if (clazz == Charset.class) {
    return Charset.forName(strVal);
}
```

发现 Class 类型为 `java.nio.charset.Charset` ，就会使用 `Charset.forName(strVal)` 方法。经过一番调试，发现最终会调用到 `jre/lib/rt.jar!/sun/nio/cs/AbstractCharsetProvider.class` 类的如下代码：

```
Class var4 = Class.forName(this.packagePrefix +
Charset var5 = (Charset)var4.newInstance();
```

其中的 `this.packagePrefix` 值为 `sun.nio.cs.ext` ，即 `charsets.jar` 的包名前缀，这些对于写文件都是可控的，所以我们可以通过如下的 json 包直接加载可控的类：

```
{
    "x":{
        "@type":"java.nio.charset.Charset",
        "val":"GBK"
    }
}
```

这样就完成了写文件漏洞在使用 fastjson 的场景下的 RCE 利用。

二. jackson 开启 enableDefaultTyping 场景

三. jdbc url getConnection 场景

四. 直接 Class forName 场景

五. 直接 loadClass newInstance 场景

0x05: 测试环境

快速搭建 docker 漏洞测试环境及相关代码托管在 spring-boot-upload-file-lead-to-rce-tricks，敬请查看。

希望有小伙伴能提 issue，一起补充下常见的 JDK HOME 目录和更多的漏洞利用场景 ^_^

参考文章：

when-class-loading-initialization

Improving Performance with Caching

Understanding Extension Class Loading

深入探讨 Java 类加载器

深入理解Java类加载

细说Class.forName()底层实现

LeadroyaL/fastjson-blacklist

标签： **spring** 红队攻防 **java**

上篇 spring actuator restart logging.config rce

下篇 spring-security 三种情况下的认证绕过

5 评论

MatthewStymn

2021-04-29 22:27:29.639394 / 回复

Woodworking projects ideas and DIY

killdooks

2021-04-30 18:07:18.979850 / 回复

Como se llamo esto i am from SPAIN

于洋

2021-05-07 02:36:18.846478 / 回复

very np，but, 你列举的jdk的路径都是ubuntu的？centos的jdk路径是包含具体版本号

andGrey

021-05-07 03:35:11.730067 / 回复

@于洋 https://github.com/LandGrey/spring-boot-upload-file-lead-to-rce-tricks 这里面列的 jdk 路径大部分都是网络搜集来的，相当一部分是 docker 环境里的 jdk 路径，不带具体的版本号。你说的存在具体版本号的路径我也注意到了，没有其他任何辅助信息的情况下盲猜确实困难。

darkless

2022-02-25 02:19:34.868969 / 回复

经测试，在linux默认配置下，是不会加载 charsets.jar包的。 默认 LANG=zh_CN.UTF-8，当把 LANG改为zh_CN.GBK时才可以加载 charsets.jar

« 1 »

评论

昵称 *

邮箱 *

提交