我与RocketMQ目录穿越CVE-2019-17572

> - 身为一个Javaer，我觉得都听说过RocketMQ，不会mq我只能说你Java学的依托答辩。于是这天我就搜了一下相关cve，只搜索到一个**CVE-2019-17572(目录穿越)**。决定搞一下。

信息搜集

- 有用的只有一篇**issue**。
- 描述：`A directory traversal vulnerability exists in RocketMQ's automatic topic creation`。
- > Some topics need checking filtering logic
  >
  > Test environmental conditions:
  >
  > - RocketMQ4.6.0 The latest version
  > - autoCreateTopicEnable=true default setting

漏洞复现

- 我觉得这是场恶战，所以github上下载了下4.3.2的中文注释版源码。至于如何IDEA调试源码可以搜，网上很详细。
- 接下来使用example里自带的Producer来发一个消息，其中主题包含恶意的`../`

```
Message msg = new Message("../../../../../tmp/1ue" /* Topic */,
  "TagA" /* Tag */,
  ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET) /*
Message body */
);
```

- 但是确失败了，说好的4.6.0以下呢

- 稍做迟疑以后，我跟进了源码，发现应该是后来打的补丁。

```
    public static boolean regularExpressionMatcher(String origin,
Pattern pattern) {
        if (pattern == null) {
            return true;
        }
        Matcher matcher = pattern.matcher(origin);
        return matcher.matches();
    }
```

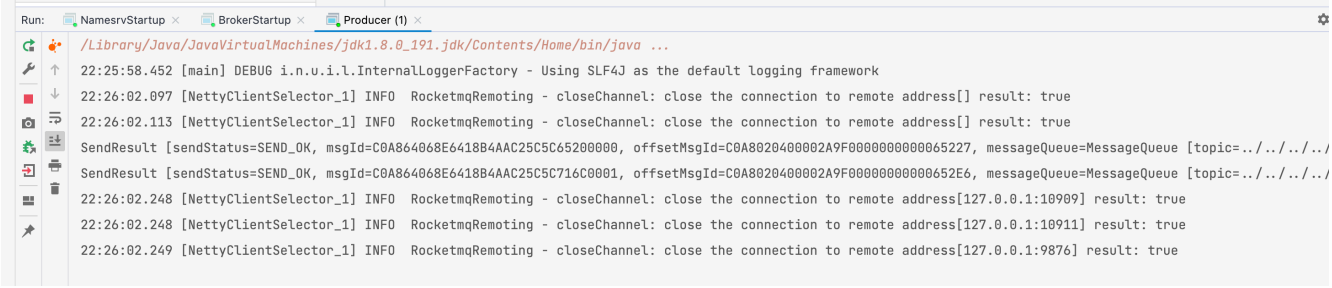
```
  /**
    * 大小写字母数字下划线
    */
  public static final String VALID_PATTERN_STR = "^[%|a-zA-Z0-9_-]+$";

  public static final Pattern PATTERN =
Pattern.compile(VALID_PATTERN_STR);
```

源码中是对topic的内容进行了正常匹配。

这个补丁。。。我觉得先删了它。

```
  public static final Pattern PATTERN = null;
```

再次启动Producer，成功目录穿越。

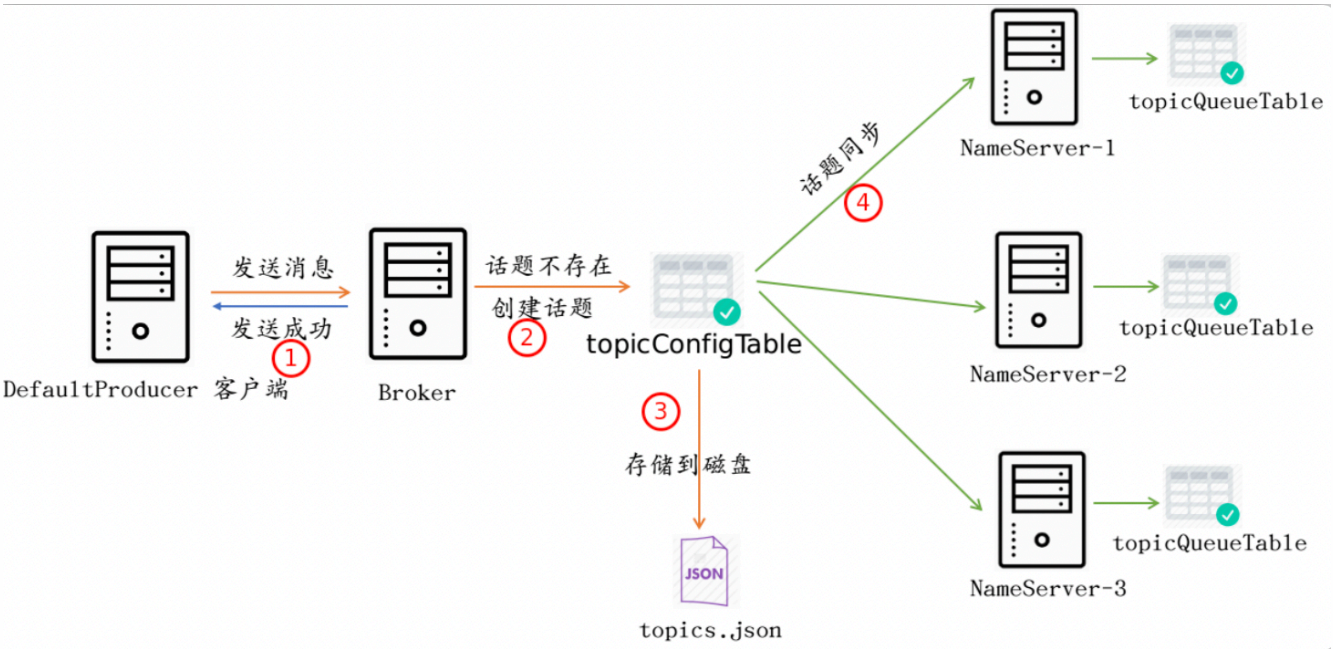



```
(base) zhchen@zhdeMacBook-Pro store-5.0 % ls /tmp
1ue
```

- 但是我发现这个1ue是个文件夹啊，合着目录穿越是任意创建文件夹，这有🐱用（怪不得没师傅分析）。但是抱着学习态度，我决定研究下原理，也为以后中间件审计打基础。

漏洞原理

- 首先明确我们的poc是啥，其实就是一个Producer发送消息，其中消息的主题是我们的evil。下图为rocketmq消息发送流程。



- 阅读相关源码或者学习相关文章后，可以了解到topic的创建是在Broker接受到Producer的消息时触发的。**这里推荐一篇学习文章**。于是把断点下在`NettyRemotingAbstract#processMessageReceived`，因为RocketMQ底层是Netty通信的。这里是接受消息的方法。

```
154         * 处理接收的请求
155         */
156        public void processMessageReceived(ChannelHandlerContext ctx, RemotingCommand m
157            final RemotingCommand cmd = msg;   msg: "RemotingCommand [code=310, language
158            if (cmd != null) {
159                switch (cmd.getType()) {
160                    case REQUEST_COMMAND:
161                        processRequestCommand(ctx, cmd);   ctx: DefaultChannelHandlerCor
162                        break;
```

- 在这里提交了线程池任务，也就是发送消息任务

```
            try {
                //在线程池里提交这个处理
                final RequestTask requestTask = new RequestTask(run, ctx.channel(), cmd);   ctx: Defaul
                pair.getObject2().submit(requestTask);   pair: Pair@3057   requestTask: RequestTask@30
            } catch (RejectedExecutionException e) {
```

- Netty中将会有一个 `SendMessageProcessor` 来处理这个任务，随后会经历 `msgCheck()` 检查消息，然后发现主题不存在进入 `createTopicInSendMessageMethod` 代码来创建Topic。

```
            log.warn("the topic {} not exist, producer: {}", requestHeader.getTopic(), ctx.channel().remoteAddr
            topicConfig = this.brokerController.getTopicConfigManager().createTopicInSendMessageMethod(
                requestHeader.getTopic(),
                requestHeader.getDefaultTopic(),
                RemotingHelper.parseChannelRemoteAddr(ctx.channel()),
                requestHeader.getDefaultTopicQueueNums(), topicSysFlag);
```

- createTopicInSendMessageMethod方法实际是在 `TopicConfigManager` 这个类中，这个类维护了 `topicConfigTable` ，记录了topic信息。
- 看看这个方法

```
/**
     * 在发消息的方法里创建topic
     * @param topic topic
     * @param defaultTopic 默认topic
     * @param remoteAddress 远程地址
     * @param clientDefaultTopicQueueNums 客户端默认队列数
     * @param topicSysFlag topic系统flag
     * @return ;
     */
    public TopicConfig createTopicInSendMessageMethod(final String
topic, final String defaultTopic,
                                                      ...
                                                      this.persist();
                                                      ...
```

```
    }
```

这个 `persist()` 就是网上常说的持久化。跟进

```
    /**
     * 持久化
     */
    public synchronized void persist() {
        String jsonString = this.encode( prettyFormat: true);   jsonString: "{\n\t"dataVersion":{\n\t\t"counte
        if (jsonString != null) {
            String fileName = this.configFilePath();   fileName: "/Users/zhchen/store-4.3.2/config/topics.js
            try {
                MixAll.string2File(jsonString, fileName);   jsonString: "{\n\t"dataVersion":{\n\t\t"counter
stamp":1674055149367\n\t}, \n\t"topicConfigTable":{\n\t\t"SELF_TEST_TOPIC":{\n\t\t\t"order":false, \n\t\t\t"perm":6, \n\t\t\t"readQueueNums":1, \n\t\t\t"
                log.error( "persist file " + fileName + " exception", e);
            }
        }
    }
```

- `string2FileNotSafe()` 看来开发者自己也知道这样不安全呀

```
242
243         //临时文件
244         String tmpFile = fileName + ".tmp";   tmpFile: "/Users/zhchen/store-4.3.2/config/topics.json.tmp"
245         string2FileNotSafe(str, tmpFile);   str: "{\n\t"dataVersion":{\n\t\t"counter":11,\n\t\t"timestamp":167
246         //备份文件
247         String bakFile = fileName + ".bak";   bakFile: "/Users/zhchen/store-4.3.2/config/topics.json.bak"
248         String prevContent = file2String(fileName);   fileName: "/Users/zhchen/store-4.3.2/config/topics.json"
249         if (prevContent != null) {
250             string2FileNotSafe(prevContent, bakFile);   bakFile: "/Users/zhchen/store-4.3.2/config/topics.json.
```

- 回手查看/tmp，嗯？文件夹居然没有创建。
- 在仔细思考，看到确实是文件落地了，但这不对劲啊。对的，没错，落地的是topics.json文件，里面存储了topic的相关信息。
- 那大不了接着往下调试呗。在笔者自己调试了十几遍之后，依旧没有发现触发点。但可以确定的是当 `requestTask` 结束之后，文件夹就出现了！猜测是有其他线程在做，但太奇怪了！

```
101         @Override
102         public void run() {
103             if (!this.stopRun) {   stopRun: false
104                 this.runnable.run();   runnable: NettyRemotingAbstract$
105             }
106         }
```

- 决定还是深入学习一波。原来RocketMQ的存储另有洞天。笔者下面只讲述一些漏洞相关。
- 首先我们明确每个topic名称对应的文件夹是什么？queuId。

- 而 `ConsumeQueue` 类对应的是每个topic和queuId下面的所有文件。Consumequeue类文件的存储路径默认为$HOME/store/consumequeue/{topic}/{queueId}/{fileName}，每个文件由30W条数据组成。

```
public class ConsumeQueue {
    // ...
    // 每个 队列下，会有很多的 File，所以这边是个 队列
    private final MappedFileQueue mappedFileQueue;
    private final String topic;
    private final int queueId;
    // ...
}
```

- 而每个 `ConsumeQueue` 下会有 `MappedFileQueue` ，这就是RocketMQ里面存储的映射机制。但这背后太复杂，暂时不用关心。来看看这个类中的蛛丝马迹。
- 可以看到在存储consumequeue的时候， `queueDir` 实际上是字符的拼接。并直接传给了 `MappedFileQueue` 的构造方法。

```
String queueDir = this.storePath
    + File.separator + topic
    + File.separator + queueId;

this.mappedFileQueue = new MappedFileQueue(queueDir, mappedFileSize, allocateMappedFileService: null);
```

```
public MappedFileQueue(final String storePath, int mappedFileSize,
    AllocateMappedFileService allocateMappedFileService) {
    this.storePath = storePath;
    this.mappedFileSize = mappedFileSize;
    this.allocateMappedFileService = allocateMappedFileService;
}
```

- 而在 `MappedFileQueue` 中的直接使用了**new File()**！

```
public boolean load() {

    File dir = new File(this.storePath);
    。。。
}
```

- 至于何时触发，RocketMQ有一个 `FlushConsumeQueueService` ，每隔 1s 执行1次刷盘动作。也解释了前面为什么requestTask结束，文件夹就出现，说明这个服务线程进行了一次刷盘，所以单调试是很难调出来的。

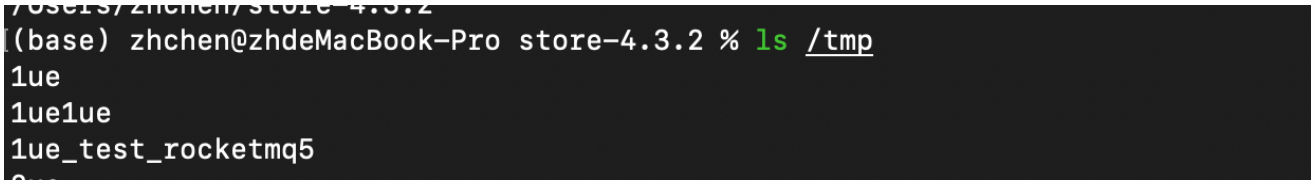
- 所以漏洞原因归根到底还是**new File()**嘛。

漏洞后续

- github那篇issue的对话还挺有意思的，感觉好像是几番周折修复了这个洞，于是我又想看看最新版是怎么修复的。
- 于是我下载了RocketMQ的5.0最新版源码，翻看了一下MappedFileQueue，还是没变。
- 当然直接poc还是不行，发现多了一个`TopicValidator`这个类，好像还是对topic校验的加强，有如下方法。

```
    public static boolean isTopicOrGroupIllegal(String str) {
        int strLen = str.length();
        int len = VALID_CHAR_BIT_MAP.length;
        boolean[] bitMap = VALID_CHAR_BIT_MAP;
        for (int i = 0; i < strLen; i++) {
            char ch = str.charAt(i);
            if (ch >= len || !bitMap[ch]) {
                return true;
            }
        }
        return false;
    }

    // regex: ^[%|a-zA-Z0-9_-]+$
        // %
        VALID_CHAR_BIT_MAP['%'] = true;
        // -
        VALID_CHAR_BIT_MAP['-'] = true;
        // _
        VALID_CHAR_BIT_MAP['_'] = true;
        // |
        VALID_CHAR_BIT_MAP['|'] = true;
        for (int i = 0; i < VALID_CHAR_BIT_MAP.length; i++) {
            if (i >= '0' && i <= '9') {
                // 0-9
                VALID_CHAR_BIT_MAP[i] = true;
            } else if (i >= 'A' && i <= 'Z') {
                // A-Z
                VALID_CHAR_BIT_MAP[i] = true;
            } else if (i >= 'a' && i <= 'z') {
                // a-z
                VALID_CHAR_BIT_MAP[i] = true;
            }
        }
```

- 对每个字符校验，太细了。调皮的我还是注释掉这个方法，想看看有没有其他防御措施。很遗憾，并没有，果然这种项目都是懒得改底层的。
-



总结

- 对于漏洞原理来说，这个洞还是很简单的。但是想找到确实困难。在如今微服务盛行时代，代码越来越复杂，有时候真的很难看出漏洞点。
- 总体下来虽然花了好几天时间，但也算是学习了学多，顺便培养下独立复现漏洞和中间件代码审计的能力hh。