

# 如何快速复现/挖掘一个漏洞？

## CodeAuditAssistant高阶技巧

我和 Unam4 最近发布了一个新的 IDEA 代码审计辅助插件 CodeAuditAssistant，测试阶段目前也收到了很多反馈，那么今天就从一个漏洞实战案例出发来详细讲解下插件的使用技巧。

### 第零步：关于调用链

调用链（Call Chain）顾名思义就是方法间的调用所组成的链条，在人工代码审计中我们往往更想快速找到方法被调用的地方进行进一步的确认。

那么这个过程中对于单线程执行的程序（比如我录得视频中的 Log4j）其实检测是相对简单的（只需要处理好抽象/接口即可，必要时处理依赖注入等，这里不展开说），但是对于一些多线程执行才能触发的漏洞（比如定时任务），其实是不能轻易从一个调用链发现整个漏洞的调用的（因为设置定时任务和运行定时任务可能并不是同一个线程），所以本文从多线程漏洞的识别方式出发，带你感受 CodeAuditAssistant 带给你的效率提升。

### 第一步：查找Sink

如果不是很特殊的触发点，你都可以通过 SinkFinder/Sink 查找器进行查找，当然对于你自己的独家Sink，可以用传统ctrlf大法或者使用后续上线的 Sink自定义 功能。

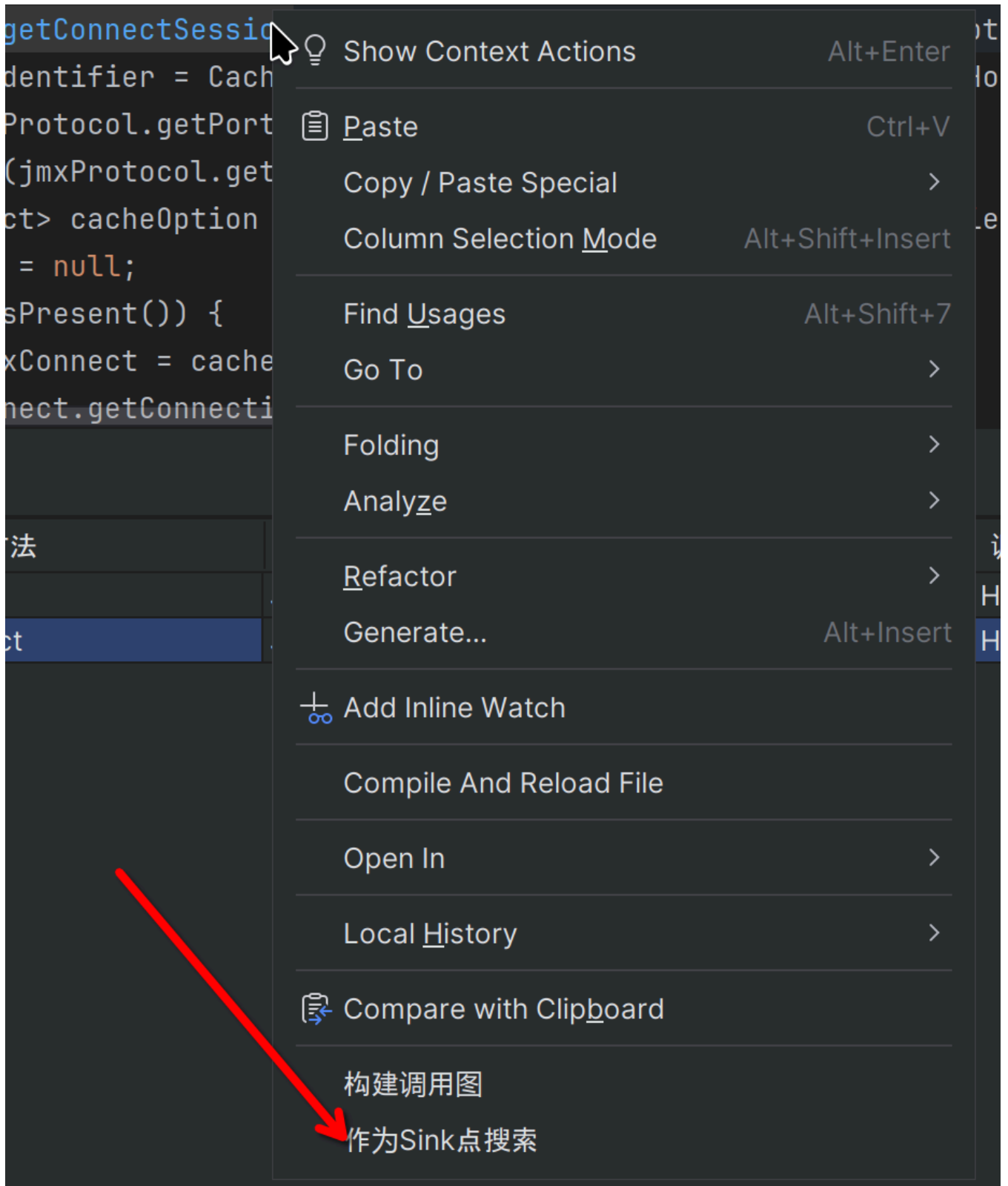
完成调用图构建和 sink 初始化之后，通过Sink查找我们找到了一个危险的 JNDI 方法：

```
    JmxServiceUrl jmxServiceUrl = new JmxServiceUrl(url);
    conn = JMXConnectorFactory.connect(jmxServiceUrl, environment);
    connectionCommonCache.addCache(identifier, new JmxConnect(conn));
    return conn;
}
```

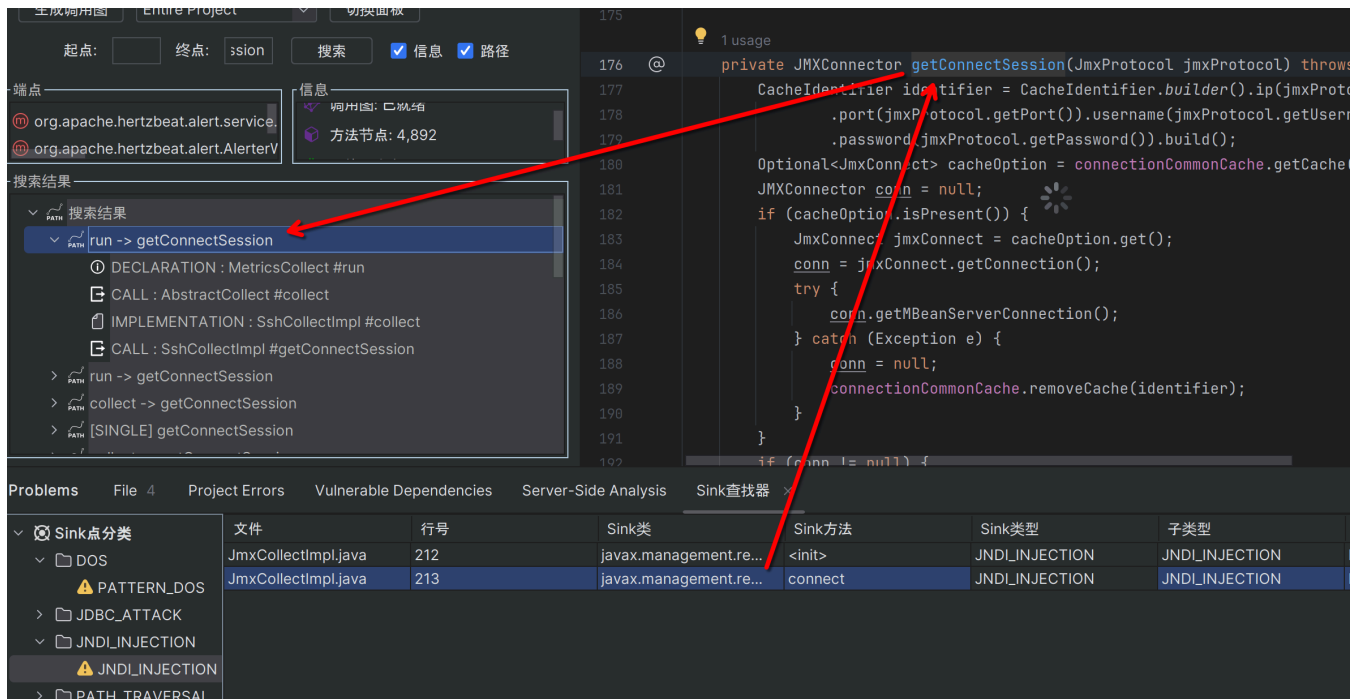
Sink查找器

Sink类	Sink方法	Sink类型	子类型	调用方式
javax.management.re...	<init>	JNDI_INJECTION	JNDI_INJECTION	HAS_CALL
javax.management.re...	connect	JNDI_INJECTION	JNDI_INJECTION	HAS_CALL

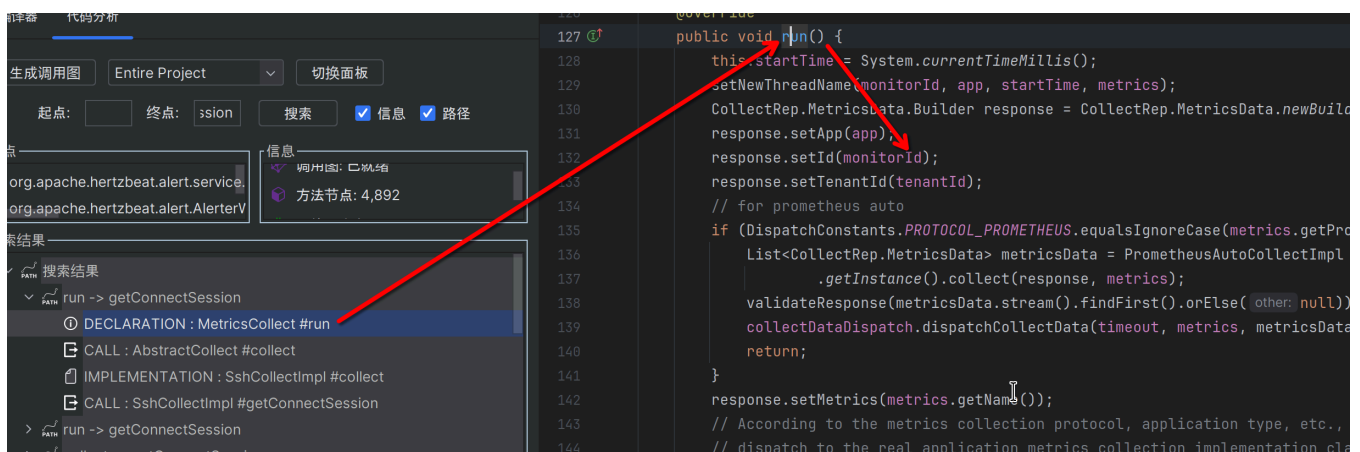
接下来我们右键点击该方法后，选择 作为Sink点搜索：



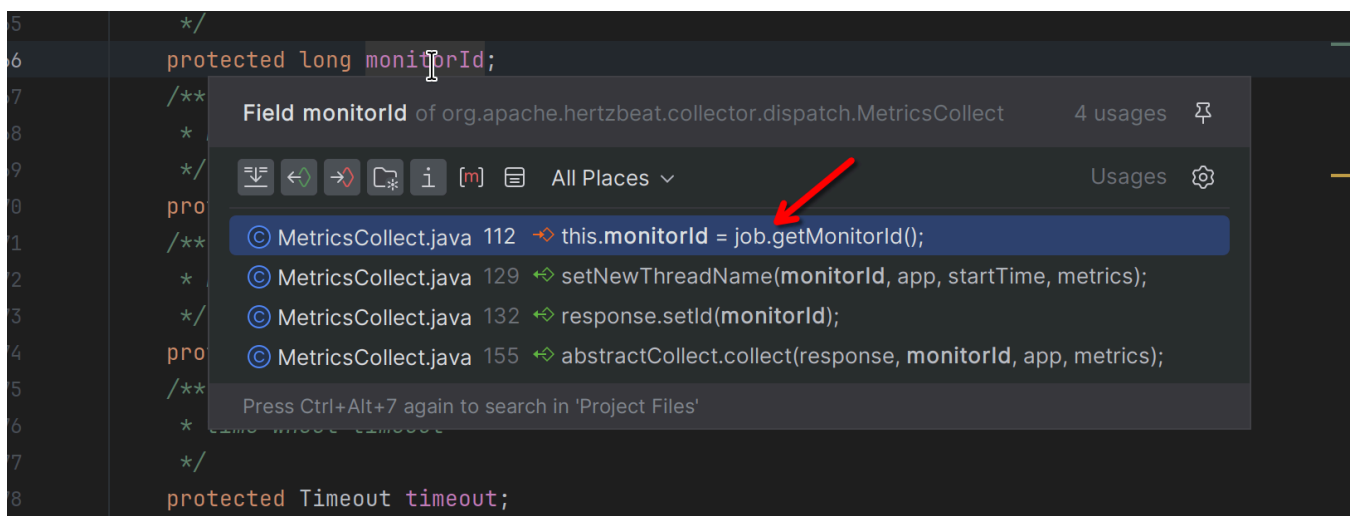
现在你就可以从搜索结果中看到这个方法的调用链了，我们展开调用链进行查看：



跟踪到第一个方法 `run` 中，在这里，调用链的第一部分结束，程序运行时的最后一段结束，接下来我们就要跟踪该类中的字段了，通过查看字段定义，我们继续跟踪到 `monitorId` 字段：



ctrl点击并发现这个字段被赋值的地方：



进入后，我们继续手动跟踪，找到task方法：

```

        this.timeout = timeout;
        this.metrics = metrics;
        this.collectorIdentity = collectorIdentity;
        WheelTimerTask timerJob = (WheelTimerTask) timeout.task();
        Job job = timerJob.getJob();
        this.monitorId = job.getMonitorId();
        this.tenantId = job.getTenantId();
        this.app = job.getApp();
        this.collectDataDispatch = collectDataDispatch;
        this.isCyclic = job.isCyclic();
        this.unitConvertList = unitConvertList;
        // Temporary one-time tasks are executed with high priority
        if (isCyclic) {
            runPriority = (byte) -1;
        } else {

```

接下来找到对 `task` 赋值的地方后，我们再次将这个作为Sink搜索：

起点: 终点: leout 搜索 信息 路径

信息

- 调用图: 已就绪
- 方法节点: 4,892
- 已使用内存: 1,119 MB
- 构建信息: Call Graph Finished

结果

- detectMonitor -> HashedWheelTimeout
  - CALL : MonitorService #detectMonitor
  - IMPLEMENTATION : MonitorServiceImpl #detectMonitor
  - CALL : CollectJobScheduling #collectSyncJobData
  - IMPLEMENTATION : CollectorJobScheduler #collectSyncJobData
  - DECLARATION : CollectJobService #collectSyncJobData
  - DECLARATION : TimerDispatch #addJob
  - DECLARATION : TimerDispatcher #addJob
  - CALL : Timer #newTimeout
  - IMPLEMENTATION : HashedWheelTimer #newTimeout
  - NEW : HashedWheelTimeout #HashedWheelTimeout
- addMonitor -> HashedWheelTimeout
- addAsyncCollectJob -> HashedWheelTimeout

5 usages

```

HashedWheelBucket bucket;

1 usage
HashedWheelTimeout(HashedWheelTimer timer, T
    this.timer = timer;
    this.task = task;
    this.deadline = deadline;
}

no usages
@Override
public Timer timer() { return timer; }

1 usage
@Override
public TimerTask task() {
    return task;
}

6 usages
@Override

```

至此，你找到了这个这个漏洞的起始点 `Service` 层的 `detectMonitor` 方法：

```

no usages
@PostMapping(path = "/detect")
@Operation(summary = "Perform availability detection on this monitoring based on monitoring",
    description = "Perform availability detection on this monitoring based on monitoring")
public ResponseEntity<Message<Void>> detectMonitor(@Valid @RequestBody MonitorDto monitorDto) {
    monitorService.validate(monitorDto, isModify: null);
    monitorService.detectMonitor(monitorDto.getMonitor(), monitorDto.getParams(), monitorDto);
    return ResponseEntity.ok(Message.success(msg: "Detect success."));
}

```

那么我们来实际运行一下查看，发现多线程调用中确实同上所述：

✓ "http-nio-1157-exec-6"...n group "main": RUNNING

← <init>:582, HashedWheelTimer\$HashedWheelTimeout (org.apa  
newTimeout:402, HashedWheelTimer (org.apache.hertzbeat.co  
addJob:85, TimerDispatcher (org.apache.hertzbeat.collector.d  
collectSyncJobData:95, CollectJobService (org.apache.hertzbe  
collectSyncJobData:290, CollectorJobScheduler (org.apache.h  
detectMonitor:162, MonitorServiceImpl (org.apache.hertzbeat.i  
invoke0:-1, NativeMethodAccessorImpl (jdk.internal.reflect)  
invoke:77, NativeMethodAccessorImpl (jdk.internal.reflect)  
invoke:43, DelegatingMethodAccessorImpl (jdk.internal.reflect)  
invoke:569, Method (java.lang.reflect)  
invokeJoinpointUsingReflection:351, AopUtils (org.springframework  
invokeJoinpoint:196, ReflectiveMethodInvocation (org.springfra  
proceed:162, ReflectiveMethodInvocation (org.springframework

