素十八

你救赎的人 终将成为你的光

洞态 IAST 试用

2021-04-01 / 24 min read

P.S. I LOVE MY COFFEE

0x01 前情提要

北京时间11点42分，正准备划 20 分钟水去吃午饭，园长突然跟我说过火线的 IAST 突然开源了，原名 `灵芝IAST` 更名为 `洞态IAST`，真是OMG。

IAST 也是同样使用 agent 技术。同样一种技术，在不同人的手里用法也不同，同一种思路在不同人手里的实现方式也可能存在差异。那么既然他开源了，那就来看一看学习一下他的思路和实现，也顺便测试一下产品，取其精华。

能有这样的产品开源供广大安全从业者测试和学习真的是一件非常好的事，真的是要撒花庆祝，感谢老板。由于明确项目定位是 IAST 产品，因此主要关注的是检测能力的实现和 sink/source/hook 点的选取。

项目地址：https://github.com/HXSecurity/DongTai-agent-java

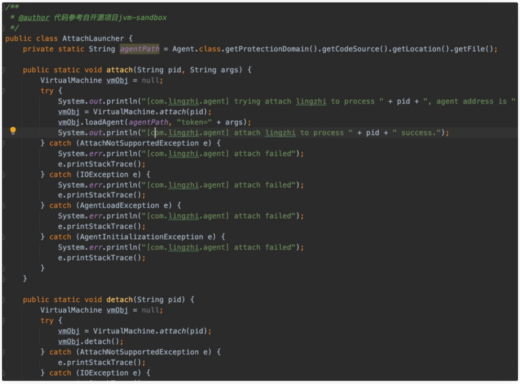
文档地址：https://github.com/HXSecurity/DongTaiDoc

0x02 项目结构

首先看一下整个的代码逻辑，项目总共分为 3 个模块，分别是 `iast-agent`、`iast-core`、`iast-inject`。

iast-agent

入口类是 `com.secnium.iast.agent.Agent`，与任何一家使用 java agent 技术的产品一样，洞态也是使用了 Sun JVM Attach API 将 agent 附加到指定的 Java 进程上。



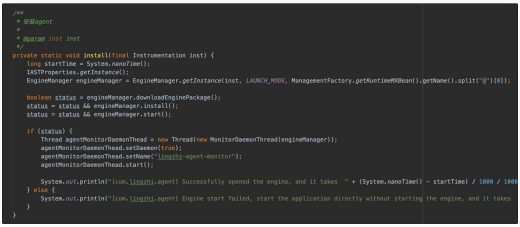
```java
/**
 * @author 代码参考自开源项目jvm-sandbox
 */
public class AttachLauncher {
    private static String agentPath = Agent.class.getProtectionDomain().getCodeSource().getLocation().getFile();

    public static void attach(String pid, String args) {
        VirtualMachine vmObj = null;
        try {
            System.out.println("[com.lingzhi.agent] trying attach lingzhi to process " + pid + ", agent address is "
            vmObj = VirtualMachine.attach(pid);
            vmObj.loadAgent(agentPath, "token=" + args);
            System.out.println("[com.lingzhi.agent] attach lingzhi to process " + pid + " success.");
        } catch (AttachNotSupportedException e) {
            System.err.println("[com.lingzhi.agent] attach failed");
            e.printStackTrace();
        } catch (IOException e) {
            System.err.println("[com.lingzhi.agent] attach failed");
            e.printStackTrace();
        } catch (AgentLoadException e) {
            System.err.println("[com.lingzhi.agent] attach failed");
            e.printStackTrace();
        } catch (AgentInitializationException e) {
            System.err.println("[com.lingzhi.agent] attach failed");
            e.printStackTrace();
        }
    }

    public static void detach(String pid) {
        VirtualMachine vmObj = null;
        try {
            vmObj = VirtualMachine.attach(pid);
            vmObj.detach();
        } catch (AttachNotSupportedException e) {
            e.printStackTrace();
        } catch (IOException e) {
```

`com.secnium.iast.agent.IASTProperties` 是 agent 的单例配置类，从 `src/main/resources/iast.properties` 中读取配置。

`com.secnium.iast.agent.IASTClassLoader` 是 agent 自定义的 ClassLoader，继承自 `URLClassLoader`，这个类的代码参考自 jvm-sandbox，其中需要注意的是，在卸载 agent 时需要关闭 ClassLoader，如果不能通过 ClassLoader 的 `close()` 方法进行关闭，则需要寻找已经打开的 jar 文件并释放文件句柄。

`com.secnium.iast.agent.UpdateUtils` 是由客户端主动向云端发送请求，用来检查版本更新和发送更新状态，其中静态方法 `sendRequest()` 可以向外发送 Http 请求。也可以看到发送请求使用了 ua：`SecniumIast Java Agent`，这部分其实可以做为一个特征。

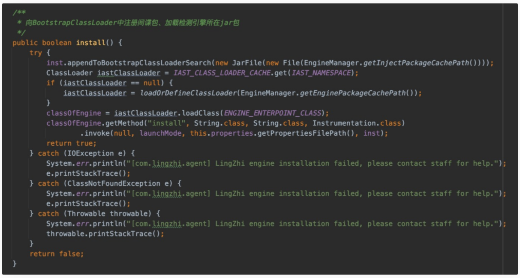
`com.secnium.iast.agent.AgentLauncher` 是在应用程序指定了 javaagent 参数情况下的入口类，这个类中的 `premain` 和 `agentmain` 方法均调用了共同的 `install()` 方法安装 agent。这个方法则是调用 `com.secnium.iast.agent.manager.EngineManager` 对整个流程进行管理。



```java
/**
 * 安装agent
 *
 * @param inst inst
 */
private static void install(final Instrumentation inst) {
    long startTime = System.nanoTime();
    IASTProperties.getInstance();
    EngineManager engineManager = EngineManager.getInstance(inst, LAUNCH_MODE, ManagementFactory.getRuntimeMXBean().getName().split("@")[0]);

    boolean status = engineManager.downloadEnginePackage();
    status = status && engineManager.install();
    status = status && engineManager.start();

    if (status) {
        Thread agentMonitorDaemonThead = new Thread(new MonitorDaemonThread(engineManager));
        agentMonitorDaemonThead.setDaemon(true);
        agentMonitorDaemonThead.setName("lingzhi-agent-monitor");
        agentMonitorDaemonThead.start();

        System.out.println("[com.lingzhi.agent] Successfully opened the engine, and it takes  " + (System.nanoTime() - startTime) / 1000 / 1000
    } else {
        System.out.println("[com.lingzhi.agent] Engine start failed, start the application directly without starting the engine, and it takes
    }
}
```

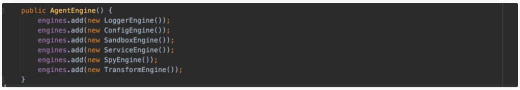
接下来我们看一下 `EngineManager` 这个类，这个类是 IAST 引擎管理器，并且使用单例对象，首先执行的是 `updateEnginePackage()` ，更新 IAST 引擎需要的 jar 包，从云端进行下载。

然后调用 `install()` 方法，首先将 iast-inject.jar 注册到 BootstrapClassLoader 中，然后使用自定义的 `IASTClassLoader` 加载检测引擎 iast-core.jar ，并反射调用里面的 `com.secnium.iast.core.AgentEngine` 的 `install()` 进行检测逻辑的初始化和加载动作。



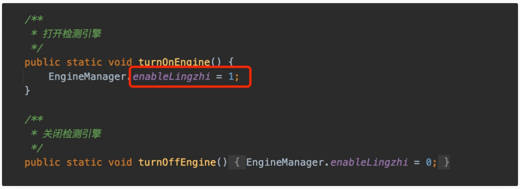
```java
/**
 * 向BootstrapClassLoader中注册间谍包、加载检测引擎所在jar包
 */
public boolean install() {
    try {
        inst.appendToBootstrapClassLoaderSearch(new JarFile(new File(EngineManager.getInjectPackageCachePath())));
        ClassLoader iastClassLoader = IAST_CLASS_LOADER_CACHE.get(IAST_NAMESPACE);
        if (iastClassLoader == null) {
            iastClassLoader = loadOrDefineClassLoader(EngineManager.getEnginePackageCachePath());
        }
        classOfEngine = iastClassLoader.loadClass(ENGINE_ENTERPOINT_CLASS);
        classOfEngine.getMethod("install", String.class, String.class, Instrumentation.class)
                .invoke(null, launchMode, this.properties.getPropertiesFilePath(), inst);
        return true;
    } catch (IOException e) {
        System.err.println("[com.lingzhi.agent] LingZhi engine installation failed, please contact staff for help.");
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        System.err.println("[com.lingzhi.agent] LingZhi engine installation failed, please contact staff for help.");
        e.printStackTrace();
    } catch (Throwable throwable) {
        System.err.println("[com.lingzhi.agent] LingZhi engine installation failed, please contact staff for help.");
        throwable.printStackTrace();
    }
    return false;
}
```

进一步加载检测引擎中的多个引擎：



```java
    public AgentEngine() {
        engines.add(new LoggerEngine());
        engines.add(new ConfigEngine());
        engines.add(new SandboxEngine());
        engines.add(new ServiceEngine());
        engines.add(new SpyEngine());
        engines.add(new TransformEngine());
    }
```

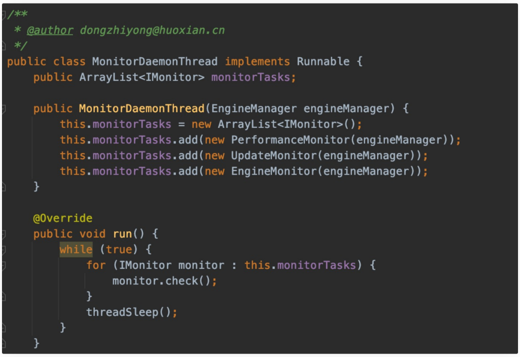
然后反射调用 `com.secnium.iast.core.AgentEngine` 的 `start()` 方法，更新在检测引擎中的一个全局标识位。



```
    /**
     * 打开检测引擎
     */
    public static void turnOnEngine() {
        EngineManager.enableLingzhi = 1;
    }

    /**
     * 关闭检测引擎
     */
    public static void turnOffEngine() { EngineManager.enableLingzhi = 0; }
```

检测引擎启动之后，agent 端还会启动几个守护线程：

- PerformanceMonitor：负责监控jvm性能状态，如果达到停止阈值（80%），则停止检测引擎；如果达到卸载阈值（90%），则卸载引擎。
- UpdateMonitor：监控云端，判断检测引擎是否需要更新。
- EngineMonitor：监控配置文件 `engine.status` 配置项的更改。



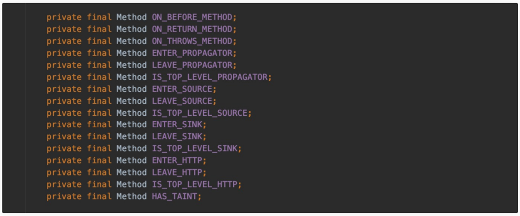
```
/**
 * @author dongzhiyong@huoxian.cn
 */
public class MonitorDaemonThread implements Runnable {
    public ArrayList<IMonitor> monitorTasks;

    public MonitorDaemonThread(EngineManager engineManager) {
        this.monitorTasks = new ArrayList<IMonitor>();
        this.monitorTasks.add(new PerformanceMonitor(engineManager));
        this.monitorTasks.add(new UpdateMonitor(engineManager));
        this.monitorTasks.add(new EngineMonitor(engineManager));
    }

    @Override
    public void run() {
        while (true) {
            for (IMonitor monitor : this.monitorTasks) {
                monitor.check();
            }
            threadSleep();
        }
    }
```

这个模块就是起到一个 agent 入口的作用，无需多言。

iast-inject

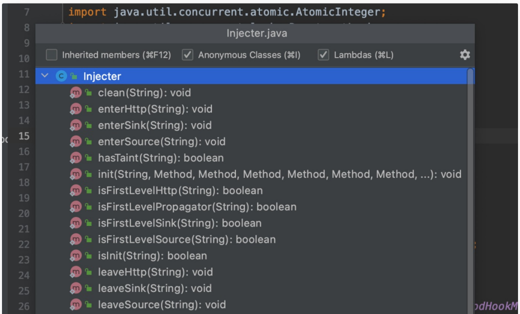
这个模块只有一个类 `java.lang.iast.inject.Injecter`，难道不应该是 Injector 吗，害。

这个类定义了一些回调方法钩子：



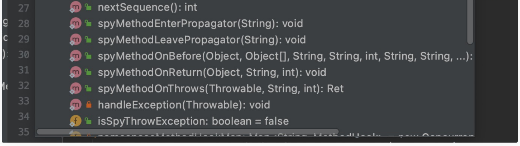
```
    private final Method ON_BEFORE_METHOD;
    private final Method ON_RETURN_METHOD;
    private final Method ON_THROWS_METHOD;
    private final Method ENTER_PROPAGATOR;
    private final Method LEAVE_PROPAGATOR;
    private final Method IS_TOP_LEVEL_PROPAGATOR;
    private final Method ENTER_SOURCE;
    private final Method LEAVE_SOURCE;
    private final Method IS_TOP_LEVEL_SOURCE;
    private final Method ENTER_SINK;
    private final Method LEAVE_SINK;
    private final Method IS_TOP_LEVEL_SINK;
    private final Method ENTER_HTTP;
    private final Method LEAVE_HTTP;
    private final Method IS_TOP_LEVEL_HTTP;
    private final Method HAS_TAINT;
```

并围绕这些钩子定义了若干方法，这些放在将会使用 ASM 时插入指定类的类字节码中。



```
import java.util.concurrent.atomic.AtomicInteger;
```

Injecter
- clean(String): void
- enterHttp(String): void
- enterSink(String): void
- enterSource(String): void
- hasTaint(String): boolean
- init(String, Method, Method, Method, Method, Method, Method, ...): void
- isFirstLevelHttp(String): boolean
- isFirstLevelPropagator(String): boolean
- isFirstLevelSink(String): boolean
- isFirstLevelSource(String): boolean
- isInit(String): boolean
- leaveHttp(String): void
- leaveSink(String): void
- leaveSource(String): void



```
nextSequence(): int
spyMethodEnterPropagator(String): void
spyMethodLeavePropagator(String): void
spyMethodOnBefore(Object, Object[], String, String, int, String, String, ...):
spyMethodOnReturn(Object, String, int): void
spyMethodOnThrows(Throwable, String, int): Ret
handleException(Throwable): void
isSpyThrowException: boolean = false
```

这个类实际上是定义中间处理逻辑以及定义供 ASM 调用的方法。代码太长了，不想细看。

iast-core

这个模块如其名字，是整个 agent 的核心，我们先来看一下整个项目的目录结构：

- **engines**：支撑整个模块调度的几个引擎，包括：ConfigEngine（通过云端返回结果构建IAST的模型规则），LoggerEngine（日志记录引擎），SandboxEngine（创建 EngineManager 实例），ServiceFactory（服务引擎，使用定时任务实现心跳、报告发送以及 http 请求重放机制），SpyEngine（初始化 SpyUtils），TransformEngine（创建自定义 ClassFileTransformer，使用 retransform 方法处理类字节码）
- **enhance**：如其名——增强，包括了自定义 ClassVisitor、自定义 AdviceAdapter、sink/source 点的选取、对一些框架和中间件的适配、SCA 检查功能的实现等等。
- **handler**：针对漏洞的检测逻辑、策略模型的建立、污点图的构造、对整个流程的处理。
- **middlewarerecognition**：用来检测当前的中间件
- **replay**：HTTP 请求重放
- **report**：一些报告和日志的实现
- **threadlocalpool**：一些需要在线程之间传递的类
- **util**：工具类
- **其他**：一些负责管理和调用的类

另外，在这个项目的 **resources** 中，还有一些基础的 txt 以及 xml 的配置，这部分等调用到此的时候再说。

首先来继续之前的调用流程，iast-agent 通过反射调用 `com.secnium.iast.core.AgentEngine` 的 `install()` 方法，调用各个引擎的 `start()` 方法。此处关注其中几个引擎：

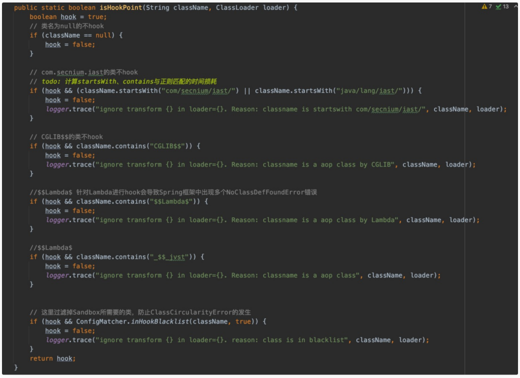
ConfigEngine 通过请求 `/api/v1/profiles`，并解析其中的结果，最后创建了一个 `IASTHookRuleModel` 实例，这个实例就是这个 agent 的处理模型，其中保存了很多处理中用到的规则。

```
private IASTHookRuleModel(HashSet<String> sources,
                          HashMap<String, IASTSinkModel> sinks,
                          HashMap<String, IASTPropagatorModel> propagators,
                          HashMap<String, List<String>> filters,
                          HashMap<String, String> interfaces,
                          HashMap<String, String> classs,
                          HashMap<Pattern, String> prefixHooks,
                          HashSet<String> hookClassnames,
                          HashSet<String> hookSuperClassnames,
                          HashSet<String> prefixHookClassName,
                          HashMap<String, Integer> hooksValue) {
    this.sources = sources;
    this.sinks = sinks;
    this.propagators = propagators;
    this.filters = filters;
    this.interfaces = interfaces;
    this.classs = classs;
    this.hooks.putAll(interfaces);
    this.hooks.putAll(classs);
    this.prefixHooks = prefixHooks;
    this.prefixHookClassName = prefixHookClassName;
    this.hookClassnames = hookClassnames;
    this.hookSuperClassnames = hookSuperClassnames;
    this.hooksValue = hooksValue;
}
```

TransformEngine使用 Instrumentation 接口，进行字节码的转换，调用了 `com.secnium.iast.core.enhance.IASTClassFileTransformer#retransform` 方法。

```
/**
 * 对字节码进行transformer
 *
 * @param inst instrument接口
 */
public static void init(Instrumentation inst) {
    IASTClassFileTransformer iastClassFileTransformer = new IASTClassFileTransformer(inst);
    inst.addTransformer(iastClassFileTransformer, true);
    iastClassFileTransformer.retransform();
}
```

这个方法使用了 `IASTClassHookPointMatcher#findForRetransform()` 使用 Instrumentation 对象的 `getAllLoadedClasses()` 获取所有已经被加载的类，并通过 `com.secnium.iast.core.util.matcher.ConfigMatcher#isHookPoint()` 方法进行筛选判断，返回了一个需要修改的类的 List 。



```java
public static boolean isHookPoint(String className, ClassLoader loader) {
    boolean hook = true;
    // 类名为null的不hook
    if (className == null) {
        hook = false;
    }

    // com.secnium.iast的类不hook
    // todo: 计算startsWith、contains与正则匹配的时间损耗
    if (hook && (className.startsWith("com/secnium/iast/") || className.startsWith("java/lang/iast/"))) {
        hook = false;
        logger.trace("ignore transform {} in loader={}. Reason: classname is startswith com/secnium/iast/", className, loader);
    }

    // CGLIB$$的类不hook
    if (hook && className.contains("CGLIB$$")) {
        hook = false;
        logger.trace("ignore transform {} in loader={}. Reason: classname is a aop class by CGLIB", className, loader);
    }

    //$$Lambda$ 针对Lambda进行hook会导致Spring框架中出现多个NoClassDefFoundError错误
    if (hook && className.contains("$$Lambda$")) {
        hook = false;
        logger.trace("ignore transform {} in loader={}. Reason: classname is a aop class by Lambda", className, loader);
    }

    //$$Lambda$
    if (hook && className.contains("_$$_jvst")) {
        hook = false;
        logger.trace("ignore transform {} in loader={}. Reason: classname is a aop class", className, loader);
    }


    // 这里过滤掉Sandbox所需要的类，防止ClassCircularityError的发生
    if (hook && ConfigMatcher.inHookBlacklist(className, true)) {
        hook = false;
        logger.trace("ignore transform {} in loader={}. reason: class is in blacklist", className, loader);
    }
    return hook;
}
```

通过以上代码可知，agent 对以下的类没有 hook：

- 类名 `com/secnium/iast` 开头的类
- 类名 `java/lang/iast/` 开头的类
- 类名包含 `CGLIB$$` 的类
- 类名包含 `$$Lambda$` 的类
- 类型包含 `_$$_jvst` 的类
- 在资源文件 `com.secnium.iast.resources/blacklist.txt` 写入的类，其中包括了：`EDU/oswego/cs/dl/util/concurrent/*`、`net/sf/ehcache/*`、`net/bytebuddy/*`、`com/secnium/*`、`apple/*`、`com/octo/captcha/*`、`Routes$`、`aQute/service/*`、`ch/qos/logback/*`、`bsh/*`、`antlr/debug/*`、`com/bea/common/*`等。

在 blacklist.txt 中写了多达 7 万多行的类名和前后缀，根据其注释，这是为了过滤掉 Sandbox 所需要的类，防止 ClassCircularityError 的发生。

随后调用 `retransformClasses()` 会让类重新加载，从而使得注册的类修改器能够重新修改类的字节码，这要就会调用之前通过 `addTransformer()` 注册的 IASTClassFileTransformer 中重写的 `transform()` 方法。

方法里首先调用 `com.secnium.iast.core.enhance.IASTClassAncestorQuery#scanCodeSource` 通过获取 jar 包中的 manifest 信息并将其发送回云端，这部分是 SCA 功能的实现。

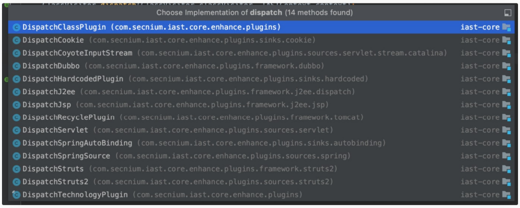
然后二次调用了 `ConfigMatcher.isHookPoint()` 判断 hook 类，感觉这个判断写重了，没必要。

在 `IASTClassAncestorQuery` 里缓存了 CodeSource/ClassLoader/ClassName/SuperName/Interfaces。

创建 ClassWriter，依然是使用 `COMPUTE_FRAMES` 自动计算帧的大小，并且重写了 `getCommonSuperClass()` 方法，在计算两个类共同的父类时指定ClassLoader。

创建 IASTContext 上下文，初始化 PluginRegister，这个类中包含了一个全局常量 PLUGINS，里面保存了很多的处理插件，这些类都实现了 DispatchPlugin 接口，这个接口包含两个方法：

- `dispatch()`：分发不同的 classVisitor 处理对应的类
- `isMatch()`：判断是否命中当前插件



DispatchClassPlugin (com.secnium.iast.core.enhance.plugins) iast-core
DispatchCookie (com.secnium.iast.core.enhance.plugins.sinks.cookie) iast-core
DispatchCoyoteInputStream (com.secnium.iast.core.enhance.plugins.sources.servlet.stream.catalina) iast-core
DispatchDubbo (com.secnium.iast.core.enhance.plugins.framework.dubbo) iast-core
DispatchHardcodedPlugin (com.secnium.iast.core.enhance.plugins.sinks.hardcoded) iast-core
DispatchJ2ee (com.secnium.iast.core.enhance.plugins.framework.j2ee.dispatch) iast-core
DispatchJsp (com.secnium.iast.core.enhance.plugins.framework.j2ee.jsp) iast-core
DispatchRecyclePlugin (com.secnium.iast.core.enhance.plugins.framework.tomcat) iast-core
DispatchServlet (com.secnium.iast.core.enhance.plugins.sources.servlet) iast-core
DispatchSpringAutoBinding (com.secnium.iast.core.enhance.plugins.sinks.autobinding) iast-core
DispatchSpringSource (com.secnium.iast.core.enhance.plugins.sources.spring) iast-core
DispatchStruts (com.secnium.iast.core.enhance.plugins.framework.struts2) iast-core
DispatchStruts2 (com.secnium.iast.core.enhance.plugins.sources.struts2) iast-core
DispatchTechnologyPlugin (com.secnium.iast.core.enhance.plugins) iast-core

在上图的类中是 DispatchPlugin 的实现类，其中包含了 agent 中的一些 sink/source/hook 点，在这些类的 `dispatch()` 方法中，会创建继承至 AbstractClassVisitor 的各个ClassVisitor，在 ClassVisitor 中又通过重写 `visitMethod()` ，注册继承至 AbstractAdviceAdapter 的实现类，这些类重写父类的 `before()/after()` ，实际上是 AdviceAdapter 的 `onMethodEnter()/onMethodExit()` 实现了字节码的插入。具体的字节码插入部分是 ASM 的 API，无需多言。

这里可以看到，洞态为每种不同的 hook点/sink点/source点订制了不同的 ClassVisitor 和 MethodVisitor，也就是说写入的字节码不一致，那到底写入了什么呢？通过看 ASM 的 API 比较难以阅读，还是在字节码写入后把 class dump 出来看比较方便。

SpyEngine 通过调用 `java.lang.iast.inject.Injecter#init` 方法将

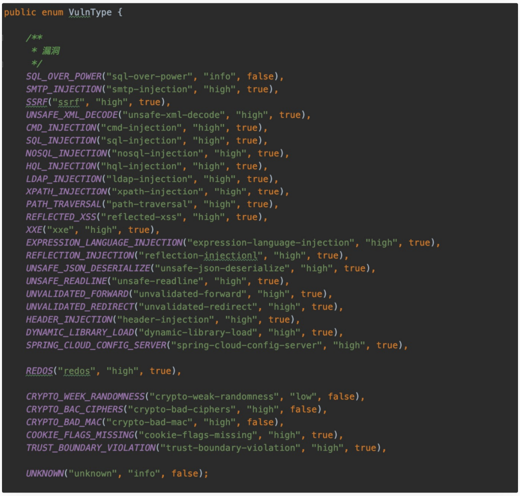
`com.secnium.iast.core.handler.EventListenerHandlers` 中定义的全部处理方法存入了 `namespaceMethodHookMap` 中供全局调用。而在 EventListenerHandlers 中定义的这个方法，实际上又是由 Injecter 通过反射调用。

而后续系统的全部功能，都是由 EventListenerHandlers 中定义的这些方法处理和调度的，这里不再进行一一分析。

0x03 功能实现探究

支持漏洞

洞态 IAST 支持的漏洞类型位于 `com.secnium.iast.core.handler.vulscan.VulnType`，如下图：

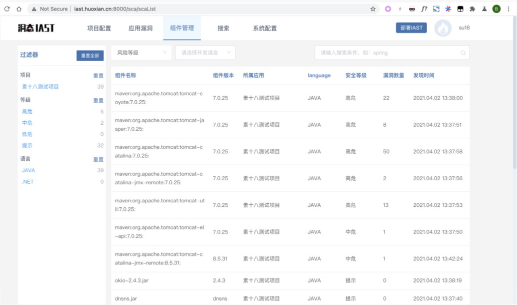


```java
public enum VulnType {

    /**
     * 漏洞
     */
    SQL_OVER_POWER("sql-over-power", "info", false),
    SMTP_INJECTION("smtp-injection", "high", true),
    SSRF("ssrf", "high", true),
    UNSAFE_XML_DECODE("unsafe-xml-decode", "high", true),
    CMD_INJECTION("cmd-injection", "high", true),
    SQL_INJECTION("sql-injection", "high", true),
    NOSQL_INJECTION("nosql-injection", "high", true),
    HQL_INJECTION("hql-injection", "high", true),
    LDAP_INJECTION("ldap-injection", "high", true),
    XPATH_INJECTION("xpath-injection", "high", true),
    PATH_TRAVERSAL("path-traversal", "high", true),
    REFLECTED_XSS("reflected-xss", "high", true),
    XXE("xxe", "high", true),
    EXPRESSION_LANGUAGE_INJECTION("expression-language-injection", "high", true),
    REFLECTION_INJECTION("reflection-injectionl", "high", true),
    UNSAFE_JSON_DESERIALIZE("unsafe-json-deserialize", "high", true),
    UNSAFE_READLINE("unsafe-readline", "high", true),
    UNVALIDATED_FORWARD("unvalidated-forward", "high", true),
    UNVALIDATED_REDIRECT("unvalidated-redirect", "high", true),
    HEADER_INJECTION("header-injection", "high", true),
    DYNAMIC_LIBRARY_LOAD("dynamic-library-load", "high", true),
    SPRING_CLOUD_CONFIG_SERVER("spring-cloud-config-server", "high", true),

    REDOS("redos", "high", true),

    CRYPTO_WEEK_RANDOMNESS("crypto-weak-randomness", "low", false),
    CRYPTO_BAC_CIPHERS("crypto-bad-ciphers", "high", false),
    CRYPTO_BAD_MAC("crypto-bad-mac", "high", false),
    COOKIE_FLAGS_MISSING("cookie-flags-missing", "high", true),
    TRUST_BOUNDARY_VIOLATION("trust-boundary-violation", "high", true),

    UNKNOWN("unknown", "info", false);
```

对应配置文件中的 model.xml，通过反向查找调用就可以查看相关的处理逻辑，各位看官请自行评测，本文不会对每种漏洞的实现进行一一介绍。

SCA 实现

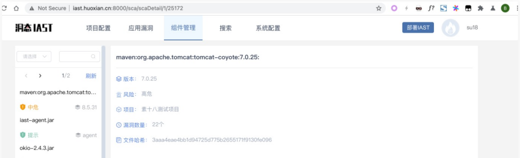
一个优秀的 IAST 一定有 SCA 一类的功能，简单的实现都是通过收集客户段组件信息，发送到云端通过匹配 CPE，并链接到对应的 CVE/CWE/CNNVD 等，并进行展示，先看一下洞态的云端效果，在组件管理：



项目 重置
素十八测试项目 39
等级 重置
高危 5
中危 2
低危 0
提示 32
语言 重置
JAVA 39
.NET 0

组件名称	组件版本	所属应用	language	安全等级	漏洞数量	发现时间
maven:org.apache.tomcat:tomcat-coyote:7.0.25:	7.0.25	素十八测试项目	JAVA	高危	22	2021.04.02 13:38:00
maven:org.apache.tomcat:tomcat-jasper:7.0.25:	7.0.25	素十八测试项目	JAVA	高危	8	2021.04.02 13:37:51
maven:org.apache.tomcat:tomcat-catalina:7.0.25:	7.0.25	素十八测试项目	JAVA	高危	50	2021.04.02 13:37:58
maven:org.apache.tomcat:tomcat-catalina-jmx-remote:7.0.25:	7.0.25	素十八测试项目	JAVA	高危	2	2021.04.02 13:37:56
maven:org.apache.tomcat:tomcat-util:7.0.25:	7.0.25	素十八测试项目	JAVA	高危	13	2021.04.02 13:37:53
maven:org.apache.tomcat:tomcat-el-api:7.0.25:	7.0.25	素十八测试项目	JAVA	中危	1	2021.04.02 13:37:50
maven:org.apache.tomcat:tomcat-catalina-jmx-remote:8.5.31:	8.5.31	素十八测试项目	JAVA	中危	1	2021.04.02 13:42:24
okio-2.4.3.jar	2.4.3	素十八测试项目	JAVA	提示	0	2021.04.02 13:38:19
dnsns.jar	dnsns	素十八测试项目	JAVA	提示	0	2021.04.02 13:37:40

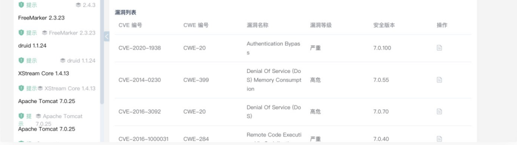
看看右边的数量和左边的数量完全对应不上，难道是我对这些数字的理解有问题？点击进入条目，有该组件对应的一些信息的展示



maven:org.apache.tomcat:to...
中危 8.5.31
iast-agent.jar
提示 agent
okio-2.4.3.jar

maven:org.apache.tomcat:tomcat-coyote:7.0.25:

版本： 7.0.25
风险： 高危
项目： 素十八测试项目
漏洞数量： 22个
文件哈希： 3aaa4eae4bb1d94725d775b2655171f9130fe096

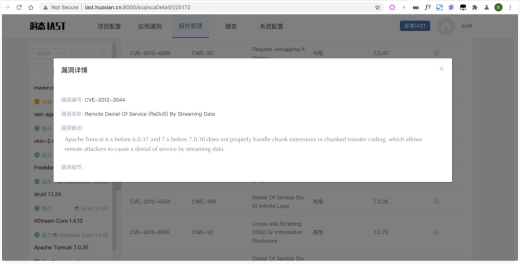


提示 2.4.3
FreeMarker 2.3.23
提示 FreeMarker 2.3.23
druid 1.1.24
提示 druid 1.1.24
XStream Core 1.4.13
提示 XStream Core 1.4.13
Apache Tomcat 7.0.25
提示 Apache Tomcat 7.0.25
Apache Tomcat 7.0.25

漏洞列表

CVE 编号	CWE 编号	漏洞名称	漏洞等级	安全版本	操作
CVE-2020-1938	CWE-20	Authentication Bypass	严重	7.0.100	
CVE-2014-0230	CWE-399	Denial Of Service (DoS) Memory Consumption	高危	7.0.55	
CVE-2016-3092	CWE-20	Denial Of Service (DoS)	高危	7.0.70	
CVE-2016-1000031	CWE-284	Remote Code Executi	严重	7.0.40	

再点击就有对应 CVE 的一些漏洞信息描述的信息。



Not Secure | iast.huoxian.cn:8000/sca/scaDetail/1/25172

漏洞详情

漏洞编号: CVE-2012-3544

漏洞名称: Remote Denial Of Service (ReDoS) By Streaming Data

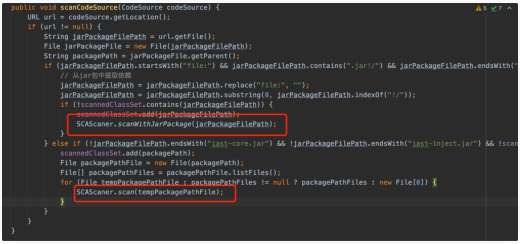
漏洞概述:

Apache Tomcat 6.x before 6.0.37 and 7.x before 7.0.30 does not properly handle chunk extensions in chunked transfer coding, which allows remote attackers to cause a denial of service by streaming data.

漏洞细节:

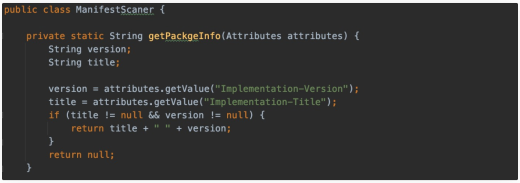
CVE-2013-4286	CWE-20	Request-smuggling Attacks	中危	7.0.47
CVE-2012-4534	CWE-399	Denial Of Service (DoS) Infinite Loop	低危	7.0.28
CVE-2016-6816	CWE-20	Cross-site Scripting (XSS) Or Information Disclosure	高危	7.0.73

那么这个功能是如何实现的呢？在前面提到 SCA 是由 `com.secnium.iast.core.enhance.IASTClassAncestorQuery#scanCodeSource` 所实现，这个方法有两个出口，据我判断，应该是不会走到下面那个 `scan()` 方法。



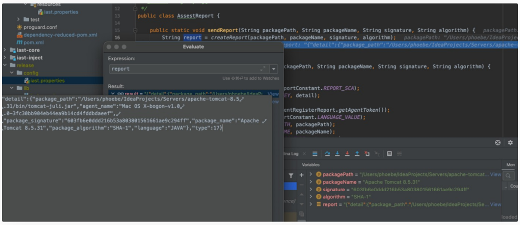
```java
public void scanCodeSource(CodeSource codeSource) {
    URL url = codeSource.getLocation();
    if (url != null) {
        String jarPackageFilePath = url.getFile();
        File jarPackageFile = new File(jarPackageFilePath);
        String packagePath = jarPackageFile.getParent();
        if (jarPackageFilePath.startsWith("file:") && jarPackageFilePath.contains(".jar!/") && jarPackageFilePath.endsWith("
            // 从jar包中提取依赖
            jarPackageFilePath = jarPackageFilePath.replace("file:", "");
            jarPackageFilePath = jarPackageFilePath.substring(0, jarPackageFilePath.indexOf("!/"));
            if (!scannedClassSet.contains(jarPackageFilePath)) {
                scannedClassSet.add(jarPackageFilePath);
                SCAScaner.scanWithJarPackage(jarPackageFilePath);
            }
        } else if (!jarPackageFilePath.endsWith("iast-core.jar") && !jarPackageFilePath.endsWith("iast-inject.jar") && !scan
            scannedClassSet.add(packagePath);
            File packagePathFile = new File(packagePath);
            File[] packagePathFiles = packagePathFile.listFiles();
            for (File tempPackagePathFile : packagePathFiles != null ? packagePathFiles : new File[0]) {
                SCAScaner.scan(tempPackagePathFile);
            }
        }
    }
}
```

`com.secnium.iast.core.enhance.sca.ManifestScaner#parseJarManifest` 调用 `getPackgeInfo` 获取 `Attributes` 中的 Implementation-Version 和 Implementation-Title



```java
public class ManifestScaner {

    private static String getPackgeInfo(Attributes attributes) {
        String version;
        String title;

        version = attributes.getValue("Implementation-Version");
        title = attributes.getValue("Implementation-Title");
        if (title != null && version != null) {
            return title + " " + version;
        }
        return null;
    }
```

最后拼接出了一些对应的信息发送给云端。



```java
 */
public class AssestReport {

    public static void sendReport(String packagePath, String packageName, String signature, String algorithm) {
        String report = createReport(packagePath, packageName, signature, algorithm);
```

Expression: report

```
"detail":{"package_path":"/Users/phoebe/IdeaProjects/Servers/apache-tomcat-8.5
.31/bin/tomcat-juli.jar","agent_name":"Mac OS X-bogon-v1.0
.0-3fc30bb904eb44ea9b14cd4fddbdaeef",
"package_signature":"603fb6e0ddd216b53a803801561661ae9c294ff","package_name":"Apache
Tomcat 8.5.31","package_algorithm":"SHA-1","language":"JAVA"},"type":17}
```

```
packagePath = "/Users/phoebe/IdeaProjects/Servers/apache-tomcat... View
packageName = "Apache Tomcat 8.5.31"
signature = "603fb6e0ddd216b53a803801561661ae9c294ff"
algorithm = "SHA-1"
report = "{"detail":{"package_path":"/Users/phoebe/IdeaProjects/Se... View
```

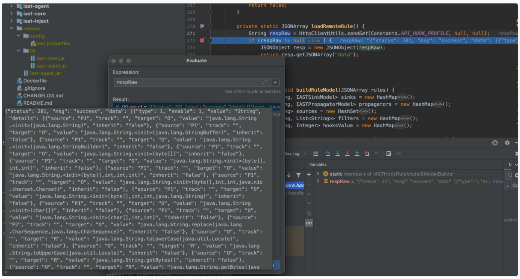
云端接收到这些信息处理入库，并对接自己 CPE/CVE 漏洞信息库进行分析和展示。

这部分实际上是非常简单的实现，没有复杂的检查逻辑，这个功能 OWASP 有开源的，建议参考：https://github.com/jeremylong/DependencyCheck

sink/source/propagator/http

构建一个 IAST，重要的就是整个模型的构建，前面分析过，模型的构建是通过 `buildRemote()`

方法获取远端的配置。



```java
        return false;
    }

    private static JSONArray loadRemoteRule() {
        String respRaw = HttpClientUtils.sendGet(Constants.API_HOOK_PROFILE, null, null);   respRaw
        if (respRaw != null = true ) {   respRaw: "{"status": 201, "msg": "success", "data": [{"type"
            JSONObject resp = new JSONObject(respRaw);
            return resp.getJSONArray("data");
```

```java
oid buildRuleModel(JSONArray rules) {
ng, IASTSinkModel> sinks = new HashMap<~>();
ng, IASTPropagatorModel> propagators = new HashMap<~>();
ng> sources = new HashSet<~>();
ng, List<String>> filters = new HashMap<~>();
ng, Integer> hooksValue = new HashMap<~>();
```

```
respRaw = "{"status": 201, "msg": "success", "data": [{"type": 1, "er... View
```

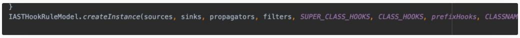
```json
{"status": 201, "msg": "success", "data": [{"type": 1, "enable": 1, "value": "String",
 "details": [{"source": "P1", "track": "", "target": "0", "value": "java.lang.String
 .<init>(java.lang.String)", "inherit": "false"}, {"source": "P1", "track": "",
 "target": "0", "value": "java.lang.String.<init>(java.lang.StringBuffer)", "inherit":
 "false"}, {"source": "P1", "track": "", "target": "0", "value": "java.lang.String
 .<init>(java.lang.StringBuilder)", "inherit": "false"}, {"source": "P1", "track": "",
 "target": "0", "value": "java.lang.String.<init>(byte[])", "inherit": "false"},
 {"source": "P1", "track": "", "target": "0", "value": "java.lang.String.<init>(byte[],
 int,int)", "inherit": "false"}, {"source": "P1", "track": "", "target": "0", "value":
 "java.lang.String.<init>(byte[],int,int,int)", "inherit": "false"}, {"source": "P1",
 "track": "", "target": "0", "value": "java.lang.String.<init>(byte[],int,int,java.nio
 .charset.Charset)", "inherit": "false"}, {"source": "P1", "track": "", "target": "0",
 "value": "java.lang.String.<init>(byte[],int,int,java.lang.String)", "inherit":
 "false"}, {"source": "P1", "track": "", "target": "0", "value": "java.lang.String
 .<init>(char[])", "inherit": "false"}, {"source": "P1", "track": "", "target": "0",
 "value": "java.lang.String.<init>(char[],int,int)", "inherit": "false"}, {"source":
 "P2", "track": "", "target": "0", "value": "java.lang.String.replace(java.lang
 .CharSequence,java.lang.CharSequence)", "inherit": "false"}, {"source": "0", "track":
 "", "target": "R", "value": "java.lang.String.toLowerCase(java.util.Locale)",
 "inherit": "false"}, {"source": "0", "track": "", "target": "R", "value": "java.lang
 .String.toUpperCase(java.util.Locale)", "inherit": "false"}, {"source": "0", "track":
 "", "target": "R", "value": "java.lang.String.getBytes()", "inherit": "false"},
 {"source": "0", "track": "", "target": "R", "value": "java.lang.String.getBytes(java
```

由于这个 json 太长了，我没细看，在这里就不展示了，这个配置在本地也有一个 model.xml，以 xml 格式储存了这些信息，这些信息在处理后会被转为 `IASTHookRuleModel` 对象。

在这个配置中我们发现了一些标记，他们都代表什么呢？

- type：这个 hook 点的类型，一个 hook 点会被分类为：1. 传播节点 2. source 点 3. filter 点 4. sink 点
- value：方法类型
- details：inherit 是否继承，value 方法签名，target 目标参数位置，source 源参数位置，track 标记是否要追踪

其中 sink 点要标记污点所在的参数位置，传播节点要标记源位置和目标位置。分类处理完这些配置文件后，将会将所有信息保存到 `IASTHookRuleModel` 中的一些变量中。



```java
}
IASTHookRuleModel.createInstance(sources, sinks, propagators, filters, SUPER_CLASS_HOOKS, CLASS_HOOKS, prefixHooks, CLASSNAM
```

在之前的分析中就提到过，对于每一种不同的 hook 点，插入的字节码是不一样的。

sink 点插入：

```java
        try {
            Injecter.enterSink("LingZhi");
            if (Injecter.isFirstLevelSink("LingZhi")) {
                Injecter.spyMethodOnBefore(对应sink点参数);
            }

            // 原始逻辑。。。

            Injecter.leaveSink("LingZhi");
            return 返回结果;
        } catch (Exception e) {
            throw e;
        }
```

传播节点：

```java
        try {
            Injecter.spyMethodEnterPropagator("LingZhi");

            // 原始逻辑。。。

            if (Injecter.isFirstLevelPropagator("LingZhi")) {
                Injecter.spyMethodOnBefore(对应传播节点参数);
            }
            Injecter.spyMethodLeavePropagator("LingZhi");
            return 返回结果;
        } catch (Exception e) {
            throw e;
        }
```

source节点：

```java
        try {
            Injecter.enterSource("LingZhi");

            // 原始逻辑。。。

            if (Injecter.isFirstLevelSource("LingZhi")) {
                Injecter.spyMethodOnBefore(对应source参数}
            Injecter.leaveSource("LingZhi");
            return 返回结果;
        } catch (Exception e) {
            throw e;
        }
```

http节点：

```java
        try {
            Injecter.enterHttp("LingZhi");
            if (Injecter.isFirstLevelHttp("LingZhi")) {
                Injecter.spyMethodOnBefore(对应参数);
            }
            // 原始逻辑。。。

            Injecter.leaveHttp("LingZhi");
        } catch (Exception e) {
            throw e;
        }
```

这其中的逻辑，总结起来是这样的，系统中定义了了一个 `com.secnium.iast.core.handler.controller.TrackerHelper`，用来作为一个全局的计数器？（追踪器），当进入一个节点时，对应的成员变量会自增 1，而退出时，会 -1，并且判断当前节点是否为第一层级节点，如果不是，将不会走入后续 `spyMethodOnBefore()` 方法。这个类中还定义个一个 trackCounts，目前还没有用上。

在 `spyMethodOnBefore()` 方法中，将会 Hook 点类型的不同分别调用 `HttpImpl.solveHttp`、`PropagatorImpl.solvePropagator`、`SourceImpl.solveSource`、`SinkImpl.solveSink` 进行不同点的处理。

http 节点处理

创建一个自定义的 `com.secnium.iast.core.util.http.HttpRequest` 对象，储存相关内容和 httpServletRequest 引用对象，静态文件不处理，目前定义的静态文件后缀是 `.js,.css,.htm,.html,.jpg,.png,.gif,.woff,.woff2,.ico,.maps,.xml`，看到了 htm/html 没有处理，这种情况下伪静态的网站可能会漏检查，根据正则查看 url 中是否含有 “login” 字样，并设置其是否为登陆 URL，将一些信息初始化和缓存到 EngineManager 中。这类节点主要负责标记和预处理的。

sink 节点处理

使用程序启动时加载的 IASTHookRuleModel，在其中获取方法签名对应的 sink 方法对象，这里返回的是一个 IASTSinkModel 对象，调用 `com.secnium.iast.core.handler.vulscan.ScannerFactory#preScan` 进行数据预处理，预处理主要是包括对 `unvalidated-redirect` 和 `sql-over-power` 两种漏洞类型的处理。

预处理之后，调用 `com.secnium.iast.core.handler.vulscan.ScannerFactory#scan`，又分别进行动静态的扫描：

- 静态扫描包括：`crypto-weak-randomness`、`crypto-bad-mac`、`crypto-bad-ciphers`、`cookie-flags-missing` 四种漏洞类型的支持。
- OverPower 扫描：目前没有具体实现。
- 动态扫描：判断 sink 方法的污点来源是否命中污点池，将当前调用的污点事件存入 `EngineManager.TRACK_MAP` 中。

source 节点处理

将当前污点来源事件存入 `EngineManager.TRACK_MAP` 中，将污点来源的返回结果放入 `EngineManager.TAINT_POOL` 污点池中。

propagator 节点处理

处理传播节点的逻辑是最复杂，这里还是简单描述：

- 如果污点池为空，证明还没有经过 source 点，则不处理传播节点，否则还是去 IASTHookRuleModel 里找对应的 IASTPropagatorModel 对象。
- 如果找到了对应的对象，则调用 `auxiliaryPropagator()` 对象，根据传播节点的配置将结果写入 `event.outValue` 。并将传播节点时间写入 `EngineManager.TRACK_MAP` 中。
- 如果没找到对应的对象，则调用 `TrackUtils.smartEventMatchAndSetTaint()` 判断，判断太长了没看，最后还是将传播节点写入 `EngineManager.TRACK_MAP` 中。

看完了这些节点的处理方式，我们简单串一下逻辑：

1. 一次请求到达了应用程序，首先进入 http 节点处理逻辑，进行标记和预处理。
2. 请求进入到 source 点，将 event 放入 `EngineManager.TRACK_MAP` 中，将 source 的结果放入了 `EngineManager.TAINT_POOL` 污点池中。
3. 请求再进入 propagator 节点时，根据配置判断传播节点的参数是否存在于污点池中，如果是，则将传播节点 event 放入 `EngineManager.TRACK_MAP` 中。
4. 应用程序走到最后的 sink 点时，根据 sink 点的配置，判断 sink 点的参数是不是在 `TAINT_POOL` 中，如果是，则将 sink 点写入 `EngineManager.TRACK_MAP` 中。
5. 随着程序的多次调用，程序还会再次进入多次传播节点，这些节点也会被放入 `EngineManager.TRACK_MAP` 中。
6. 在应用程序执行完，回到 http 节点，最后执行到 `leaveHttp` 时，会调用 GraphBuilder 构造污点调用图并发送至云端。

越权检测

在看代码的时候，多次看到 `over power` 一类的字样，想来想去终于想明白了，这可能是越权的意思。IAST 能检测越权？有点意思，那我们来看看他是如何实现的。

之前提到过 `com.secnium.iast.core.handler.vulscan.ScannerFactory#preScan`，在这个位置命中 sink 点后，有两个处理逻辑：

1. 如果命中了 `unvalidated-redirect` 的 sink 点，并且是方法签名是 setHeader/addHeader 等，就将其理解为可能是登陆成功后的跳转操作，找到其中 Set-Cookie 的值：

- 如果判断当前为登录逻辑，则保存到 AuthInfoCache中，
- 否则将更新原有缓存的 cookie 信息，并且向云端发送报告

2. 将会创建 `IJdbc` 的实例， 调用 `LoginLogicRecognize.handleLoginLogicRecognize()` 方法处理登陆逻辑识别：

- 通过 `isLoginSqlQuery()` 使用正则匹配 sql 语句，看是否有登录字样，识别是否为登录连接
- 调用 `AuthInfoManager.handleAddCookieAction()` 方法将“登陆相关的sql查询语句”、“cookie”信息、sink 点的 ClassName发送至云端，并把 cookie 缓存到 AuthInfoCache 中。

后续在之前也看到了，对应的 `OverPowerScanner` 的 scan 方法没有具体实现，那这时候我们可以猜测一下，作者主要是想通过 cookie 和登陆的 sql 语句进行关联。通过检查 sql 语句是否与污点池有关、检查 sink 点的参数是否与污点池有关来判断是否有越权。这些数据都被发送到了云端，那对于云端来说，如何区分不同权限的用户？如何判断这个请求是否应该匹配到用

户？

以目前程序里的处理逻辑，还做不到鉴权的功能，在云端中也没有展示这个漏洞类型，等待进一步的更新。

其他

其他漏洞的检测逻辑没什么要说的，主要是 hook 点的选取。

0x04 测试

在看完代码逻辑和简单试用后，我们正式的测试一下这个产品。
首先编译一下 agent，我这边的环境是：

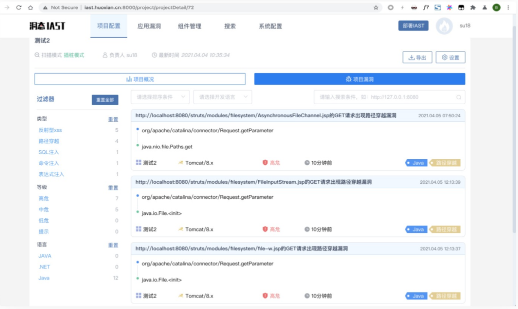
- mvn 3.2.5
- JDK 1.8.0_131
- Tomcat 8.5.31/7.0.25
- 测试代码：自写靶场，由于我的靶场是为了测试 RASP 产品而写，当时为了测试自家 LingXe 以及 OpenRasp 、云锁、安全狗等等产品包含的 RASP 功能，里面隐藏了很多可以用来插桩和绕过的点，因此用来测试 IAST 正好，园长发了个修改版在 https://github.com/tongasdp/tongasdp-test，有兴趣的小伙伴也可以用来自测。

通过调试发现自己编译 core 和 inject 没用，他的 agent 无论如何还是会从官网上自己下载这两个 jar 包并放到 temp 目录下，不知道是故意的还是写出来的 bug，因为如果从官网上下载，也只是下载 agent，在运行时动态下载 core 和 inject，应该是最开始打算试用，没打算开源？？由于我的目的是学习调试，所以我使用了自己编译的关闭了 proguard 混淆的版本，并修改了判断逻辑，使应用程序不去云端判断，直接加载本地 jar。

为了更直观的看到 agent 对类字节码的更改，需要在配置文件中更改 `iast.dump.class.enable` 和 `iast.dump.class.path` 相关参数。

功能型测试

通过上一章功能实现的探究，我们已经关注到了几个安全检测功能的实现，那具体的检测结果怎么样呢？



测试2

扫描模式 插桩模式　负责人 su18　最新时间 2021.04.04 10:35:34

过滤器

类型 重置
反射型xss 5
路径穿越 4
SQL注入 1
命令注入 1
表达式注入 1

等级 重置
高危 7
中危 5
低危 0
提示 0

语言 重置
JAVA 0
.NET 0
Java 12

http://localhost:8080/struts/modules/filesystem/AsynchronousFileChannel.jsp的GET请求出现路径穿越漏洞 2021.04.05 07:50:24
- org/apache/catalina/connector/Request.getParameter
- java.nio.file.Paths.get

测试2　Tomcat/8.x　高危　10分钟前　Java　路径穿越

http://localhost:8080/struts/modules/filesystem/FileInputStream.jsp的GET请求出现路径穿越漏洞 2021.04.05 12:13:39
- org/apache/catalina/connector/Request.getParameter
- java.io.File.<init>

测试2　Tomcat/8.x　高危　10分钟前　Java　路径穿越

http://localhost:8080/struts/modules/filesystem/file-w.jsp的GET请求出现路径穿越漏洞 2021.04.05 12:13:37
- org/apache/catalina/connector/Request.getParameter
- java.io.File.<init>

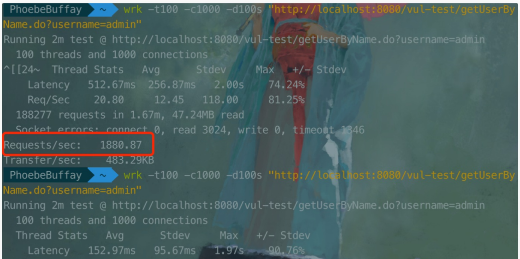
测试2　Tomcat/8.x　高危　10分钟前　Java　路径穿越

我在自己的靶场里触发了绝大部分的漏洞类型，但很遗憾的是，由于云端的搜索引擎问题，以及 sink 点选取问题，我没能在云端看到太多的检出漏洞。云端的漏洞展示是有问题的，看不到前一天的漏洞内容，不知道是 django 的问题还是什么，建议修复一下。

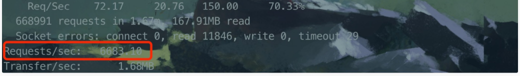
也建议作者出一个官方漏洞靶场，能对应到所有洞态支持的漏洞类型，也容易理解和说明。

性能测试

以下是使用 wrk 进行的压测：



```
 PhoebeBuffay  ~  wrk -t100 -c1000 -d100s "http://localhost:8080/vul-test/getUserBy
Name.do?username=admin"
Running 2m test @ http://localhost:8080/vul-test/getUserByName.do?username=admin
  100 threads and 1000 connections
^[[24~  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   512.67ms  256.87ms   2.00s    74.24%
    Req/Sec    20.80     12.45   118.00     81.25%
  188277 requests in 1.67m, 47.24MB read
  Socket errors: connect 0, read 3024, write 0, timeout 1346
Requests/sec:   1880.87
Transfer/sec:    483.29KB
 PhoebeBuffay  ~  wrk -t100 -c1000 -d100s "http://localhost:8080/vul-test/getUserBy
Name.do?username=admin"
Running 2m test @ http://localhost:8080/vul-test/getUserByName.do?username=admin
  100 threads and 1000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   152.97ms   95.67ms   1.97s    90.76%
```



```
    Req/Sec    72.17     20.76   150.00     70.33%
  668991 requests in 1.67m, 167.91MB read
  Socket errors: connect 0, read 11846, write 0, timeout 29
Requests/sec:   6683.10
Transfer/sec:      1.68MB
```

可以看到洞态给应用程序性能带来的影响特别大，当然 IAST 通常都在测试环境下使用，所以可能并不是特别关注性能。

0x05 评价

出来在之前分析过程中的一些我将从两个方面对目前版本洞态 IAST 进行评价，首先是使用中的一些想法：

1. SCA 漏洞组件管理没有整理和去重，在我测试的过程中多次重启项目导致同一个条目在云端能看到很多次。
2. 页面上表格查询和相关排序用起来真的难受，建议招一个设计。
3. 搜索功能面对小白非常不友好，可以说如果对 IAST 不了解的情况完全用不了这个管理后台。
4. 支持的漏洞数量还是少了点，并且没有对这种漏洞的描述、解决建议什么的信息。
5. 既然是 java agent 技术的产品，那应该能够给到用户完整的调用链、一些关键调用点信息如代码行数等等，而不是只给到一个 http 请求。

作为一名安全研究员，同时也是 RASP 产品的参与者，我提出几点想法：

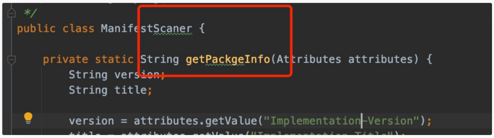
1. 所有的 hook 点，全部是写死的规则文件，无论是本地的 xml 也好，还是远端的 json 也好，都将 hook 点的类，描述符，相关信息完全写死，我相信这些规则是通过某些手段生成出来的，但是一旦在 hook 点选取上没有选择使用动态手段，那就失去了和 0 day 打交道的能力。
2. hook 点（这里指 sink ）的选取还是层次太浅、规律性较差。
3. 洞态 IAST 检测了当前环境使用的中间件，并发送给云端，目前除了信息收集还没看到有什么样的具体用途，但是通过hook点来看，对于http请求的点还是使用了适配各个中间件的方式， OpenRASP 也是采用这种方式，这种方式在功能上没有什么问题，但是同样地还是失去了动态性，不优雅，也不能做到通用。
4. 应用在实现上使用了太多的字符串比较处理，以及正则，这将对原应用性能带来极大的影响。
5. 在调试过程中，包括污点图的处理，总是被大量的无效信息占用了过多的时间，比如 StringBuilder 一类的传播节点，这其实不是漏洞调用的关键节点，个人认为没有必要处理他。
6. agent 每次收到请求，只要不是静态的，都要向云端发送报告，丧心病狂吧。
7. 没有对 ClassLoader 进行相关处理，无论是前两年的冰蝎，还是各种反序列化 的利用 gadget ，包括我自己的 su18.jsp ，都少不了使用 ClassLoader 加载恶意类的请求，这应该目前 Java 安全关注点比较高的地方，还是建议给 IAST 一个挖 gadget 的可能。
8. 攻击者目前常用的类似内存马、动态注册 Filter 一类的、以及像一些反序列化恶意类找 response 对象回显的，其实都可以试着搞一搞。

就这样吧，也不想说太多了。

0x06 吐槽

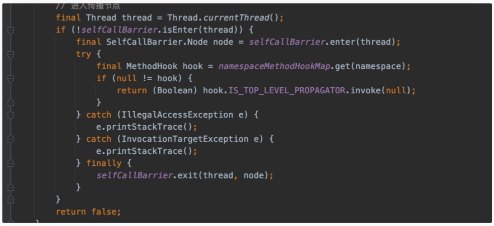
通过个人角度，主要觉得这个项目有以下槽点：

- 错别拼写真的不少



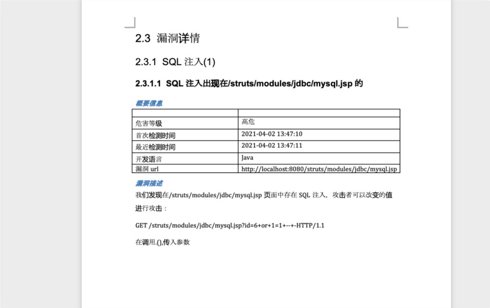
```java
 */
public class ManifestScaner {

    private static String getPackgeInfo(Attributes attributes) {
        String version;
        String title;

        version = attributes.getValue("Implementation-Version");
```

- 代码重复也不少



```java
        // 进入传播节点
        final Thread thread = Thread.currentThread();
        if (!selfCallBarrier.isEnter(thread)) {
            final SelfCallBarrier.Node node = selfCallBarrier.enter(thread);
            try {
                final MethodHook hook = namespaceMethodHookMap.get(namespace);
                if (null != hook) {
                    return (Boolean) hook.IS_TOP_LEVEL_PROPAGATOR.invoke(null);
                }
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (InvocationTargetException e) {
                e.printStackTrace();
            } finally {
                selfCallBarrier.exit(thread, node);
            }
        }
        return false;
```

- 报告导出的 word 字体到底是咋回事



2.3 漏洞详情

2.3.1 SQL 注入(1)

2.3.1.1 SQL 注入出现在/struts/modules/jdbc/mysql.jsp 的

概要信息

危害等级	高危
首次检测时间	2021-04-02 13:47:10
最近检测时间	2021-04-02 13:47:11
开发语言	Java
漏洞 url	http://localhost:8080/struts/modules/jdbc/mysql.jsp

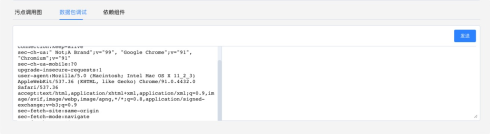
漏洞描述

我们发现在/struts/modules/jdbc/mysql.jsp 页面中存在 SQL 注入，攻击者可以改变的值进行攻击：

GET /struts/modules/jdbc/mysql.jsp?id=6+or+1=1+--+-HTTP/1.1

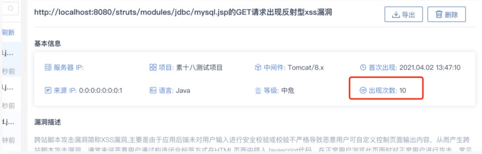
在调用.(),传入参数

- 这个功能我点了，不好使，我没看代码有没有相关实现，但这功能贼危险，建议还是不要搞了



```
sec-ch-ua:" Not;A Brand";v="99", "Google Chrome";v="91",
"Chromium";v="91"
sec-ch-ua-mobile:?0
upgrade-insecure-requests:1
user-agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 11_2_3)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4432.0
Safari/537.36
accept:text/html,application/xhtml+xml,application/xml;q=0.9,im
age/avif,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.9
sec-fetch-site:same-origin
sec-fetch-mode:navigate
```

- 应用漏洞的展示页面中不会出现重复漏洞，会进行合并，但是可怕的是我触发一次SQL 注入漏洞，页面上的出现次数就涨 5 次，这应该是同一条请求调用链上触发多个 sink 点导致的?



http://localhost:8080/struts/modules/jdbc/mysql.jsp的GET请求出现反射型xss漏洞

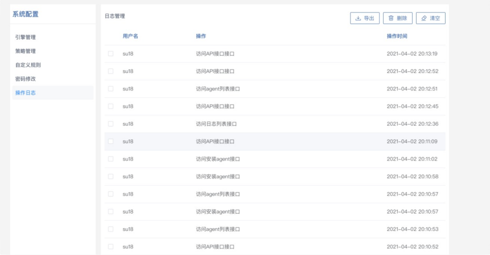
基本信息

服务器 IP:
项目: 素十八测试项目
中间件: Tomcat/8.x
首次出现: 2021.04.02 13:47:10
来源 IP: 0:0:0:0:0:0:0:1
语言: Java
等级: 中危
出现次数: 10

漏洞描述

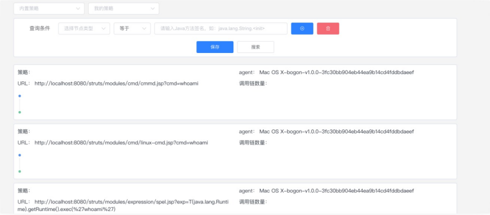
跨站脚本攻击漏洞简称XSS漏洞,主要是由于应用后端未对用户输入进行安全校验或校验不严格导致恶意用户可自定义控制页面输出内容，从而产生跨

- 这个日志记录提供清空和删除功能不说，详细度也完全不够，绝对过不了等保。。。



	用户名	操作	操作时间
☐	su18	访问API接口接口	2021-04-02 20:13:19
☐	su18	访问API接口接口	2021-04-02 20:12:52
☐	su18	访问agent列表接口	2021-04-02 20:12:51
☐	su18	访问API接口接口	2021-04-02 20:12:45
☐	su18	访问日志列表接口	2021-04-02 20:12:36
☐	su18	访问API接口接口	2021-04-02 20:11:09
☐	su18	访问安装agent接口	2021-04-02 20:11:02
☐	su18	访问安装agent接口	2021-04-02 20:10:58
☐	su18	访问agent列表接口	2021-04-02 20:10:57
☐	su18	访问安装agent接口	2021-04-02 20:10:57
☐	su18	访问agent列表接口	2021-04-02 20:10:53
☐	su18	访问API接口接口	2021-04-02 20:10:52

- 搜索显示完全有问题

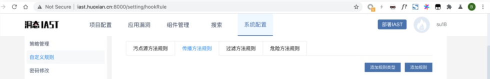


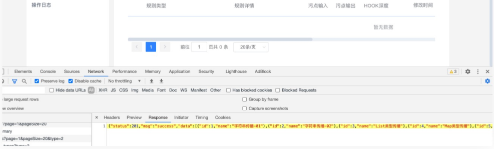
策略：
URL： http://localhost:8080/struts/modules/cmd/cmmd.jsp?cmd=whoami
agent： Mac OS X-bogon-v1.0.0-3fc30bb904eb44ea9b14cd4fddbdaeef
调用链数量：

策略：
URL： http://localhost:8080/struts/modules/cmd/linux-cmd.jsp?cmd=whoami
agent： Mac OS X-bogon-v1.0.0-3fc30bb904eb44ea9b14cd4fddbdaeef
调用链数量：

策略：
URL： http://localhost:8080/struts/modules/expression/spel.jsp?exp=T(java.lang.Runtime).getRuntime().exec(%27whoami%27)
agent： Mac OS X-bogon-v1.0.0-3fc30bb904eb44ea9b14cd4fddbdaeef
调用链数量：

- 后端明明回数据了，为啥前面不展示呢

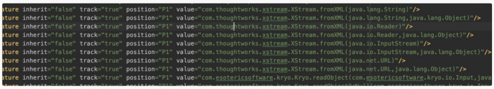




规则类型	规则详情	污点输入	污点输出	HOOK深度	修改时间
暂无数据					

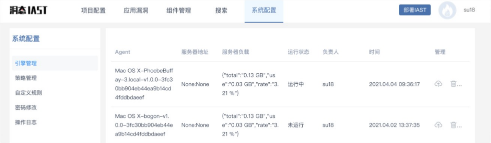
```
{"status":201,"msg":"success","data":[{"id":1,"name":"字符串传播-01"},{"id":2,"name":"字符串传播-02"},{"id":3,"name":"List类型传播"},{"id":4,"name":"Map类型传播"},{"id":5,
```

- 一个方法的多个重载方法，没必要都写进去，因为有自调用啊



```
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.lang.String)"/>
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.lang.String,java.lang.Object)"/>
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.io.Reader)"/>
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.io.Reader,java.lang.Object)"/>
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.io.InputStream)"/>
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.io.InputStream,java.lang.Object)"/>
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.net.URL)"/>
ature inherit="false" track="true" position="P1" value="com.thoughtworks.xstream.XStream.fromXML(java.net.URL,java.lang.Object)"/>
ature inherit="false" track="true" position="P1" value="com.esotericsoftware.kryo.Kryo.readObject(com.esotericsoftware.kryo.io.Input,java
```

- 区分应用是怎么区分的呢?



Agent	服务器地址	服务器负载	运行状态	负责人	时间	管理
Mac OS X-PhoebeBuffay-3.local-v1.0.0-3fc30bb904eb44ea9b14cd4fddbdaeef	None:None	{"total":"0.13 GB","use":"0.03 GB","rate":"3.21 %"}	运行中	su18	2021.04.04 09:36:17	...
Mac OS X-bogon-v1.0.0-3fc30bb904eb44ea9b14cd4fddbdaeef	None:None	{"total":"0.13 GB","use":"0.03 GB","rate":"3.21 %"}	未运行	su18	2021.04.02 13:37:35	...

0x07 总结

对于一款使用 Java agent 技术开发的工具/产品，重中之重就是 hook 点的选取，以及处理各项逻辑的具体实现，因为需要将代码运行到服务器端，不影响原有功能、不影响原有性能是首要考虑的目标。

这是作为框架考虑的，其次是针对各个漏洞点的检测逻辑，这部分是需要对漏洞理解的足够深入，也就是需要安全研究人员的介入，目前洞态有一部漏洞的检测是写死在代码里的，一部分漏洞的检测是依靠配置文件的设置的。还没有处理成大家习惯的框架-插件的模式，所以想在此基础上二开还是需要花费较多时间理解代码。

期待更新。

下一篇

fastjson：我一路向北，离开有你的季节

2 条评论　　su18

说点什么

ⓘ 支持 Markdown 语法　　预览　　评论

su18 发表于 大约 1 个月前

> @exexute
> 师傅分析的很好，也提出了很多的**问题(有很多的误解)**（代码风格、代码抽象程度不够、逻辑写的不够简单导致难以理解、对IAST检测原理的误解等）。看完师傅的分析，决定**推出**一系列洞态IAST**技术的科普**，后续将放在**火线Zone**中，供大家学习、讨论、交流，帮助那些了解相关技术的开发/研究人员**更快**的**理解**洞态IAST，更便捷的**参与**到IAST的**社区建设**。

太好了，期待共同建设和持续更新。

exexute 发表于 大约 1 个月前　　1

师傅分析的很好，也提出了很多的**问题(有很多的误解)**（代码风格、代码抽象程度不够、逻辑写的不够简单导致难以理解、对IAST检测原理的误解等）。看完师傅的分析，决定**推出**一系列洞态IAST**技术的科普**，后续将放在**火线Zone**中，供大家学习、讨论、交流，帮助那些了解相关技术的开发/研究人员**更快**的**理解**洞态IAST，更便捷的**参与**到IAST的**社区建设**。