java反序列化利用链自动挖掘工具gadgetinspector源码浅析

threedr3am / 2020-01-08 09:34:15 / 浏览数 14488

0x01 前言

我们在使用ysoserial的时候，经常会用它生成序列化的payload，用于攻击具有反序列化功能的endpoint，而这些payload大部分都是比较长的一条执行链，在反序列化期间，由执行程序执行攻击者可控的source，然后通过依赖中存在的执行链，最终触发至slink，从而达到攻击的效果。

这些gadget chain有长有短，大部分可以通过类似Intellij idea这类工具去根据slink，查找调用者，以及各种调用者的实现，一路反向的跟踪，对于一些比较简单比较短的链，通常通过人工查找也能快速的找到，但是对于一些比较长的链，人工查找会耗费巨大的精力和时间，并且不一定能挖掘到gadget chain。

而有段时间，我苦恼于人工查找浪费巨大精力得不偿失时，忽然发现这样一款自动化挖掘gadget chain的工具，通过阅读分析它的源码，它给我带来了非常多的知识以及自动化挖掘的思路，其中就包括类似污点分析，如何去分析方法调用中，参数是否可以影响返回值，从而跟踪数据流动是否可以从source最终流动至slink，并影响至最终的slink点。

gadgetinspector：

https://github.com/JackOfMostTrades/gadgetinspector

个人加了点注释的gadgetinspector

https://github.com/threedr3am/gadgetinspector

slink：

- Runtime.exec()：这种利用最为简单，但是实际生产情况基本不会遇到
- Method.invoke()：这种方式通过反射执行方法，需要方法以及参数可控
- RMI/JRMP：通过反序列化使用RMI或者JRMP链接到我们的exp服务器，通过发送序列化payload至靶机实现
- URL.openStream：这种利用方式需要参数可控，实现SSRF
- Context.lookup：这种利用方式也是需要参数可控，最终通过rmi或ldap的server实现攻击
- ...等等

在分析gadgetinspector源码的时候，大概会在以下几方面去讲解，并核心分析ASM部分，详细讲解如何进行污点分析：

1. GadgetInspector：main方法，程序的入口，做一些配置以及数据的准备工作
2. MethodDiscovery：类、方法数据以及父子类、超类关系数据的搜索
3. PassthroughDiscovery：分析参数能影响到返回值的方法，并收集存储
4. CallGraphDiscovery：记录调用者caller方法和被调用者target方法的参数关联
5. SourceDiscovery：入口方法的搜索，只有具备某种特征的入口才会被标记收集
6. GadgetChainDiscovery：整合以上数据，并通过判断调用链的最末端slink特征，从而判断出可利用的gadget chain

0x02 GadgetInspector：入口代码的分析

程序启动的入口，在该方法中，会做一些数据的准备工作，并一步步调用MethodDiscovery、PassthroughDiscovery、CallGraphDiscovery、SourceDiscovery、GadgetChainDiscovery，最终实现gadget chain的挖掘

参数合法判断：

```
if (args.length == 0) {
    printUsage();
    System.exit(1);
}
```

在程序的入口处，会先判断启动参数是否为空，若是空，则直接退出，因为程序对挖掘的gadget chain会有类型的区分，以及class所在位置的配置

日志、序列化类型配置：

```
//配置log4j用于输出日志
configureLogging();

boolean resume = false;
//挖掘的gadget chain序列化类型，默认java原生序列化
GIConfig config = ConfigRepository.getConfig("jserial");
```

日志配置是便于统一的输出管理，而序列化类型的配置，因为对链的挖掘前，我们需要确定挖掘的是哪种类型的链，它可以是jackson的json序列化，也可以是java原生的序列化等等

序列化配置接口：

```
public interface GIConfig {

    String getName();
    SerializableDecider getSerializableDecider(Map<MethodReference.Handle, MethodReference> methodMap, InheritanceMap inheritanceMap);
    ImplementationFinder getImplementationFinder(Map<MethodReference.Handle, MethodReference> methodMap,
                                                 Map<MethodReference.Handle, Set<MethodReference.Handle>> methodImplMap,
                                                 InheritanceMap inheritanceMap);
    SourceDiscovery getSourceDiscovery();

}
```

既然我们选择了不同的序列化形式，那么，相对来说，它们都会有自身特有的特征，因此我们需要实现jackson特有的SerializableDecider、ImplementationFinder、SourceDiscovery，从而能达到区分，并最终实现gadget chain的挖掘，

例jackson：

- SerializableDecider-JacksonSerializableDecider：

```
public class JacksonSerializableDecider implements SerializableDecider {
    //类是否通过决策的缓存集合
    private final Map<ClassReference.Handle, Boolean> cache = new HashMap<>();
    //类名-方法集合 映射集合
    private final Map<ClassReference.Handle, Set<MethodReference.Handle>> methodsByClassMap;

    public JacksonSerializableDecider(Map<MethodReference.Handle, MethodReference> methodMap) {
        this.methodsByClassMap = new HashMap<>();
        for (MethodReference.Handle method : methodMap.keySet()) {
            Set<MethodReference.Handle> classMethods = methodsByClassMap.get(method.getClassReference());
            if (classMethods == null) {
                classMethods = new HashSet<>();
                methodsByClassMap.put(method.getClassReference(), classMethods);
            }
            classMethods.add(method);
        }
    }

    @Override
    public Boolean apply(ClassReference.Handle handle) {
        Boolean cached = cache.get(handle);
        if (cached != null) {
            return cached;
        }

        Set<MethodReference.Handle> classMethods = methodsByClassMap.get(handle);
        if (classMethods != null) {
            for (MethodReference.Handle method : classMethods) {
                //该类，只要有无参构造方法，就通过决策
                if (method.getName().equals("<init>") && method.getDesc().equals("()V")) {
                    cache.put(handle, Boolean.TRUE);
                    return Boolean.TRUE;
                }
            }
        }

        cache.put(handle, Boolean.FALSE);
        return Boolean.FALSE;
    }
}
```

这一块代码，我们可以主要关心在apply方法中，可以看到，具体细节的意思就是，只要存在无参的构造方法，都表示可以被序列化。因为在java中，若没有显式的实现无参构造函数，而实现了有参构造函数，在这种情况下，该类是不具有无参构造方法的，而jackson对于json的反序列化，都是先通过无参构造方法进行实例化，因此，若无无参构造方法，则表示不能被jackson进行反序列化。所以，该决策类的存在意义，就是标识gadget chian中不可被反序列化的类，不可被反序列化就意味着数据流不可控，gadget chain无效。

- ImplementationFinder-JacksonImplementationFinder

```
public class JacksonImplementationFinder implements ImplementationFinder {

    private final SerializableDecider serializableDecider;

    public JacksonImplementationFinder(SerializableDecider serializableDecider) {
        this.serializableDecider = serializableDecider;
    }

    @Override
    public Set<MethodReference.Handle> getImplementations(MethodReference.Handle target) {
        Set<MethodReference.Handle> allImpls = new HashSet<>();

        // For jackson search, we don't get to specify the class; it uses reflection to instantiate the
        // class itself. So just add the target method if the target class is serializable.
        if (Boolean.TRUE.equals(serializableDecider.apply(target.getClassReference()))) {
            allImpls.add(target);
        }

        return allImpls;
    }
}
```

该实现类核心方法是getImplementations，因为java是一个多态性的语言，只有在运行时，程序才可知接口的具体实现类是哪一个，而gadgetinspector并不是一个运行时的gadget chain挖掘工具，因此，当遇到一些接口方法的调用时，需要通过查找该接口方法的所有实现类，并把它们组成链的一节形成实际调用的链，最后去进行污点分析。而该方法通过调用JacksonSerializableDecider的apply方法进行判断，因为对于接口或者子类的实现，我们是可控的，但是该json是否可被反序列化，需要通过JacksonSerializableDecider判断是否存在无参构造方法。

- SourceDiscovery-JacksonSourceDiscovery

```
public class JacksonSourceDiscovery extends SourceDiscovery {

    @Override
    public void discover(Map<ClassReference.Handle, ClassReference> classMap,
                         Map<MethodReference.Handle, MethodReference> methodMap,
                         InheritanceMap inheritanceMap) {

        final JacksonSerializableDecider serializableDecider = new JacksonSerializableDecider(methodMap);

        for (MethodReference.Handle method : methodMap.keySet()) {
            if (serializableDecider.apply(method.getClassReference())) {
                if (method.getName().equals("<init>") && method.getDesc().equals("()V")) {
                    addDiscoveredSource(new Source(method, 0));
                }
                if (method.getName().startsWith("get") && method.getDesc().startsWith("()")) {
                    addDiscoveredSource(new Source(method, 0));
                }
                if (method.getName().startsWith("set") && method.getDesc().matches("\\(L[^;]*;\\)V")) {
                    addDiscoveredSource(new Source(method, 0));
                }
            }
        }
    }

}
```

该实现类，仅有discover这一个方法，不过，对于gadget chain的挖掘，它可以肯定是最重要的，因为一个gadget chain的执行链，我们必须要有一个可以触发的入口，而JacksonSourceDiscovery的作用就是找出具备这样特征的入口方法，对于jackson反序列化json时，它会执行无参构造方法以及setter、getter方法，若我们在数据字段可控的情况下，并由这些被执行的方法去触发，若存

在gadget chain，那么就能触发source-slink整条链的执行。

```
int argIndex = 0;
while (argIndex < args.length) {
    String arg = args[argIndex];
    if (!arg.startsWith("--")) {
        break;
    }
    if (arg.equals("--resume")) {
        //不删除dat文件
        resume = true;
    } else if (arg.equals("--config")) {
        //--config参数指定序列化类型
        config = ConfigRepository.getConfig(args[++argIndex]);
        if (config == null) {
            throw new IllegalArgumentException("Invalid config name: " + args[argIndex]);
        }
    } else {
        throw new IllegalArgumentException("Unexpected argument: " + arg);
    }

    argIndex += 1;
}
```

此处是对于一些参数的一些解析配置：

```
--resume：不删除dat文件
--config：指定序列化类型
```

```
final ClassLoader classLoader;
//程序参数的最后一部分，即最后一个具有前缀--的参数（例：--resume）后
if (args.length == argIndex+1 && args[argIndex].toLowerCase().endsWith(".war")) {
    //加载war文件
    Path path = Paths.get(args[argIndex]);
    LOGGER.info("Using WAR classpath: " + path);
    //实现为URLClassLoader，加载war包下的WEB-INF/lib和WEB-INF/classes
    classLoader = Util.getWarClassLoader(path);
} else {
    //加载jar文件，java命令后部，可配置多个
    final Path[] jarPaths = new Path[args.length - argIndex];
    for (int i = 0; i < args.length - argIndex; i++) {
        Path path = Paths.get(args[argIndex + i]).toAbsolutePath();
        if (!Files.exists(path)) {
            throw new IllegalArgumentException("Invalid jar path: " + path);
        }
        jarPaths[i] = path;
    }
    LOGGER.info("Using classpath: " + Arrays.toString(jarPaths));
    //实现为URLClassLoader，加载所有指定的jar
    classLoader = Util.getJarClassLoader(jarPaths);
}
//类枚举加载器，具有两个方法
//getRuntimeClasses获取rt.jar的所有class
//getAllClasses获取rt.jar以及classLoader加载的class
final ClassResourceEnumerator classResourceEnumerator = new ClassResourceEnumerator(classLoader);
```

这段代码，解析了程序启动参数最后一个"--参数"后的部分，这部分可以指定一个war包，也能指定多个jar包，并最终放到ClassResourceEnumerator，ClassResourceEnumerator通过guava的ClassPath，对配置加载的war、jar中的所有class进行读取或对

jre的rt.jar中的所有class进行读取

```
//删除所有的dat文件
if (!resume) {
    // Delete all existing dat files
    LOGGER.info("Deleting stale data...");
    for (String datFile : Arrays.asList("classes.dat", "methods.dat", "inheritanceMap.dat",
            "passthrough.dat", "callgraph.dat", "sources.dat", "methodimpl.dat")) {
        final Path path = Paths.get(datFile);
        if (Files.exists(path)) {
            Files.delete(path);
        }
    }
}
```

这段代码，可以看到，如果没有配置--resume参数，那么在程序的每次启动后，都会先删除所有的dat文件

```
//扫描java runtime所有的class（rt.jar）和指定的jar或war中的所有class

// Perform the various discovery steps
if (!Files.exists(Paths.get("classes.dat")) || !Files.exists(Paths.get("methods.dat"))
        || !Files.exists(Paths.get("inheritanceMap.dat"))) {
    LOGGER.info("Running method discovery...");
    MethodDiscovery methodDiscovery = new MethodDiscovery();
    methodDiscovery.discover(classResourceEnumerator);
    //保存了类信息、方法信息、继承实现信息
    methodDiscovery.save();
}

if (!Files.exists(Paths.get("passthrough.dat"))) {
    LOGGER.info("Analyzing methods for passthrough dataflow...");
    PassthroughDiscovery passthroughDiscovery = new PassthroughDiscovery();
    //记录参数在方法调用链中的流动关联（如：A、B、C、D四个方法，调用链为A->B B->C C->D，其中参数随着调用关系从A流向B，在B调用C过
    //该方法主要是追踪上面所说的"B调用C过程中作为入参并随着方法结束返回"，入参和返回值之间的关联
    passthroughDiscovery.discover(classResourceEnumerator, config);
    passthroughDiscovery.save();
}

if (!Files.exists(Paths.get("callgraph.dat"))) {
    LOGGER.info("Analyzing methods in order to build a call graph...");
    CallGraphDiscovery callGraphDiscovery = new CallGraphDiscovery();
    //记录参数在方法调用链中的流动关联（如：A、B、C三个方法，调用链为A->B B->C，其中参数随着调用关系从A流向B，最后流C）
    //该方法主要是追踪上面所说的参数流动，即A->B入参和B->C入参的关系，以确定参数可控
    callGraphDiscovery.discover(classResourceEnumerator, config);
    callGraphDiscovery.save();
}

if (!Files.exists(Paths.get("sources.dat"))) {
    LOGGER.info("Discovering gadget chain source methods...");
    SourceDiscovery sourceDiscovery = config.getSourceDiscovery();
    //查找利用链的入口（例：java原生反序列化的readObject）
    sourceDiscovery.discover();
    sourceDiscovery.save();
}

{
    LOGGER.info("Searching call graph for gadget chains...");
    GadgetChainDiscovery gadgetChainDiscovery = new GadgetChainDiscovery(config);
    //根据上面的数据收集，最终分析利用链
    gadgetChainDiscovery.discover();
}

LOGGER.info("Analysis complete!");
```

最后这部分，就是核心的挖掘逻辑。

0x03 MethodDiscovery

这部分，主要进行了类数据、方法数据以及类继承关系数据的收集

```
if (!Files.exists(Paths.get("classes.dat")) || !Files.exists(Paths.get("methods.dat"))
        || !Files.exists(Paths.get("inheritanceMap.dat"))) {
    LOGGER.info("Running method discovery...");
    MethodDiscovery methodDiscovery = new MethodDiscovery();
    methodDiscovery.discover(classResourceEnumerator);
    //保存了类信息、方法信息、继承实现信息
    methodDiscovery.save();
}
```

从上述代码可以看到，先判断了classes.dat、methods.dat、inheritanceMap.dat三个文件是否存在，若不存在则执行MethodDiscovery的实例化，并依次调用其discover、save方法

```
public void discover(final ClassResourceEnumerator classResourceEnumerator) throws Exception {
    for (ClassResourceEnumerator.ClassResource classResource : classResourceEnumerator.getAllClasses()) {
        try (InputStream in = classResource.getInputStream()) {
            ClassReader cr = new ClassReader(in);
            try {
                //使用asm的ClassVisitor、MethodVisitor，利用观察模式去扫描所有的class和method并记录
                cr.accept(new MethodDiscoveryClassVisitor(), ClassReader.EXPAND_FRAMES);
            } catch (Exception e) {
                LOGGER.error("Exception analyzing: " + classResource.getName(), e);
            }
        }
    }
}
```

MethodDiscovery.discover方法中，通过调用classResourceEnumerator.getAllClasses()获取到rt.jar以及程序参数配置的jar、war中所有的class，然后遍历每一个class，接着通过ASM，对其每个类进行观察者模式的visit

跟进MethodDiscoveryClassVisitor，对于ClassVisitor，ASM对其每个方法的调用顺序是这样的：

visit顺序：

```
void visit(int version, int access, String name, String signature, String superName, String[] interfaces)
visit( 类版本 , 修饰符 , 类名 , 泛型信息 , 继承的父类 , 实现的接口)
```

->

```
void visitSource(String source, String debug)
```

->

```
void visitOuterClass(String owner, String name, String descriptor)
```

->

```
void visitAttribute(Attribute attribute)
```

->

```
AnnotationVisitor visitAnnotation(String descriptor, boolean visible)
visitAnnotation(注解类型，注解是否可以在 JVM 中可见)
```

->

```
void visit*()
```

->

```
void visitEnd()
```

->

```
FieldVisitor visitField(int access, String name, String descriptor, String signature, Object value)
visitField(修饰符，字段名，字段类型，泛型描述，默认值)
```

->

```
MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[] exceptions)
visitMethod(修饰符，方法名，方法签名，泛型信息，抛出的异常)
```

那么，跟进这个调用顺序，我们跟进其实现代码：

```
private class MethodDiscoveryClassVisitor extends ClassVisitor {

    private String name;
    private String superName;
    private String[] interfaces;
    boolean isInterface;
    private List<ClassReference.Member> members;//类的所有字段
    private ClassReference.Handle classHandle;

    private MethodDiscoveryClassVisitor() throws SQLException {
        super(Opcodes.ASM6);
    }

    @Override
    public void visit ( int version, int access, String name, String signature, String superName, String[]interfaces)
    {
        this.name = name;
        this.superName = superName;
        this.interfaces = interfaces;
        this.isInterface = (access & Opcodes.ACC_INTERFACE) != 0;
        this.members = new ArrayList<>();
        this.classHandle = new ClassReference.Handle(name);//类名

        super.visit(version, access, name, signature, superName, interfaces);
    }

    ...

}
```

visit()这个方法，会在类被观察的第一时间执行。可以看到在visit()这个方法执行时，保存了当前观察类的一些信息：

1. this.name：类名
2. this.superName：继承的父类名
3. this.interfaces：实现的接口名
4. this.isInterface：当前类是否接口
5. this.members：类的字段集合
6. this.classHandle：gadgetinspector中对于类名的封装

```
public FieldVisitor visitField(int access, String name, String desc,
                               String signature, Object value) {
    if ((access & Opcodes.ACC_STATIC) == 0) {
        Type type = Type.getType(desc);
        String typeName;
        if (type.getSort() == Type.OBJECT || type.getSort() == Type.ARRAY) {
            typeName = type.getInternalName();
        } else {
            typeName = type.getDescriptor();
        }
        members.add(new ClassReference.Member(name, access, new ClassReference.Handle(typeName)));
    }
    return super.visitField(access, name, desc, signature, value);
}
```

第二步，被观察类若存在多少个field字段，那么visitField()这个方法，就会被调用多少次，每调用一次，就代表一个字段。看实现代码，visitField()方法在被调用时，会通过判断字段的类型去生成typeName类型名称，最后添加到visit()方法中初始化的this.members集合

```
@Override
public MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions) {
    boolean isStatic = (access & Opcodes.ACC_STATIC) != 0;
    //找到一个方法，添加到缓存
    discoveredMethods.add(new MethodReference(
            classHandle,//类名
            name,
            desc,
            isStatic));
    return super.visitMethod(access, name, desc, signature, exceptions);
}
```

而被观察类若存在多少个方法，那么visitMethod()这个方法，就会被调用多少次，每调用一次，就代表一个方法，看上述代码，可以清楚的看到，其对方法进行了收集，并缓存在this.discoveredMethods中

```
@Override
public void visitEnd() {
    ClassReference classReference = new ClassReference(
            name,
            superName,
            interfaces,
            isInterface,
            members.toArray(new ClassReference.Member[members.size()]));//把所有找到的字段封装
    //找到一个方法遍历完成后，添加类到缓存
    discoveredClasses.add(classReference);

    super.visitEnd();
}
```

而在每一个visit*方法被执行后，最后一个执行的方法就是visitEnd()，在这段代码中，把当前的被观察的类信息缓存到了this.discoveredClasses，其中包括前面visitField阶段收集到的所有字段members

至此，MethodDiscovery.discover方法就执行完毕了，而下一步就是MethodDiscovery.save方法的执行

```
public void save() throws IOException {
    //保存和读取使用Factory实现

    //classes.dat数据格式：
    //类名(例：java/lang/String) 父类 接口A,接口B,接口C 是否接口 字段1!字段1access!字段1类型!字段2!字段2access!字段1类型
    DataLoader.saveData(Paths.get("classes.dat"), new ClassReference.Factory(), discoveredClasses);

    //methods.dat数据格式：
    //类名 方法名 方法描述 是否静态方法
    DataLoader.saveData(Paths.get("methods.dat"), new MethodReference.Factory(), discoveredMethods);

    //形成 类名(ClassReference.Handle)->类(ClassReference) 的映射关系
    Map<ClassReference.Handle, ClassReference> classMap = new HashMap<>();
    for (ClassReference clazz : discoveredClasses) {
        classMap.put(clazz.getHandle(), clazz);
    }
    //保存classes.dat和methods.dat的同时，对所有的class进行递归整合，得到集合{class:[subclass]},
    // class为subclass父类、超类或实现的接口类，保存至inheritanceMap.dat
    InheritanceDeriver.derive(classMap).save();
}
```

通过DataLoader.saveData保存了收集到的discoveredClasses类信息以及discoveredMethods方法信息，对于这些信息的存储格式，通过了ClassReference.Factory()、MethodReference.Factory()进行实现

```
public static <T> void saveData(Path filePath, DataFactory<T> factory, Collection<T> values) throws IOException {
    try (BufferedWriter writer = Files.newWriter(filePath.toFile(), StandardCharsets.UTF_8)) {
        for (T value : values) {
            final String[] fields = factory.serialize(value);
            if (fields == null) {
                continue;
            }

            StringBuilder sb = new StringBuilder();
            for (String field : fields) {
                if (field == null) {
                    sb.append("\t");
                } else {
                    sb.append("\t").append(field);
                }
            }
            writer.write(sb.substring(1));
            writer.write("\n");
        }
    }
}
```

saveData方法中会通过调用factory的serialize对数据进行序列化，然后一行一行的输出

```
public static class Factory implements DataFactory<ClassReference> {

    ...

    @Override
    public String[] serialize(ClassReference obj) {
        String interfaces;
        if (obj.interfaces.length > 0) {
            StringBuilder interfacesSb = new StringBuilder();
            for (String iface : obj.interfaces) {
                interfacesSb.append(",").append(iface);
            }
            interfaces = interfacesSb.substring(1);
        } else {
            interfaces = "";
        }

        StringBuilder members = new StringBuilder();
        for (Member member : obj.members) {
            members.append("!").append(member.getName())
                    .append("!").append(Integer.toString(member.getModifiers()))
                    .append("!").append(member.getType().getName());
        }

        return new String[]{
                obj.name,
                obj.superClass,
                interfaces,
                Boolean.toString(obj.isInterface),
                members.length() == 0 ? null : members.substring(1)
        };
    }
}
```

```
public static class Factory implements DataFactory<MethodReference> {

    ...

    @Override
    public String[] serialize(MethodReference obj) {
        return new String[] {
                obj.classReference.getName(),
                obj.name,
                obj.desc,
                Boolean.toString(obj.isStatic),
        };
    }
}
```

对于类信息的存储，最终形成classes.dat文件的数据格式是：

```
类名(例：java/lang/String) 父类 接口A,接口B,接口C 是否接口 字段1!字段1access!字段1类型!字段2!字段2access!字段1类型
```

对于方法信息的存储，最终形成methods.dat文件的数据格式是：

```
类名 方法名 方法描述 是否静态方法
```

在对类、方法信息存储后，会再进一步利用已得到的类信息，进行类继承、实现关系的整合分析：

```
//形成 类名(ClassReference.Handle)->类(ClassReference) 的映射关系
Map<ClassReference.Handle, ClassReference> classMap = new HashMap<>();
for (ClassReference clazz : discoveredClasses) {
    classMap.put(clazz.getHandle(), clazz);
}
//保存classes.dat和methods.dat的同时，对所有的class进行递归整合，得到集合{class:[subclass]},
// class为subclass父类、超类或实现的接口类，保存至inheritanceMap.dat
InheritanceDeriver.derive(classMap).save();
```

核心实现位于InheritanceDeriver.derive方法

```
public static InheritanceMap derive(Map<ClassReference.Handle, ClassReference> classMap) {
    LOGGER.debug("Calculating inheritance for " + (classMap.size()) + " classes...");
    Map<ClassReference.Handle, Set<ClassReference.Handle>> implicitInheritance = new HashMap<>();
    //遍历所有类
    for (ClassReference classReference : classMap.values()) {
        if (implicitInheritance.containsKey(classReference.getHandle())) {
            throw new IllegalStateException("Already derived implicit classes for " + classReference.getName());
        }
        Set<ClassReference.Handle> allParents = new HashSet<>();

        //获取classReference的所有父类、超类、接口类
        getAllParents(classReference, classMap, allParents);
        //添加缓存：类名 -> 所有的父类、超类、接口类
        implicitInheritance.put(classReference.getHandle(), allParents);
    }
    //InheritanceMap翻转集合，转换为{class:[subclass]}
    return new InheritanceMap(implicitInheritance);
}

/**
 * 获取classReference的所有父类、超类、接口类
 *
 * @param classReference
 * @param classMap
 * @param allParents
 */
private static void getAllParents(ClassReference classReference, Map<ClassReference.Handle, ClassReference> classMap, Set<ClassReference
    Set<ClassReference.Handle> parents = new HashSet<>();
    //把当前classReference类的父类添加到parents
    if (classReference.getSuperClass() != null) {
        parents.add(new ClassReference.Handle(classReference.getSuperClass()));
    }
    //把当前classReference类实现的所有接口添加到parents
    for (String iface : classReference.getInterfaces()) {
        parents.add(new ClassReference.Handle(iface));
    }

    for (ClassReference.Handle immediateParent : parents) {
        //从所有类数据集合中，遍历找出classReference的父类、接口
        ClassReference parentClassReference = classMap.get(immediateParent);
        if (parentClassReference == null) {
            LOGGER.debug("No class id for " + immediateParent.getName());
            continue;
        }
        //继续添加到集合中
        allParents.add(parentClassReference.getHandle());
        //继续递归查找，直到把classReference类的所有父类、超类、接口类都添加到allParents
        getAllParents(parentClassReference, classMap, allParents);
    }
}
```

前面类信息的收集保存，其得到的数据：

```
类名(例：java/lang/String) 父类 接口A,接口B,接口C 是否接口 字段1!字段1access!字段1类型!字段2!字段2access!字段1类型
```

通过这些信息，可以清楚的知道每个类继承的父类、实现的接口类，因此，通过遍历每一个类，并且通过递归的方式，从而一路向上查找收集，最终形成了父子、超类间的关系集合：

```
类名 -> 所有的父类、超类、接口类
```

并在实例化InheritanceMap返回时，在其构造方法中，对关系集合进行了逆向的整合，最终形成了：

```
类名 -> 所有的子孙类、实现类
```

构造方法细节：

```
public class InheritanceMap {
    //子-父关系集合
    private final Map<ClassReference.Handle, Set<ClassReference.Handle>> inheritanceMap;
    //父-子关系集合
    private final Map<ClassReference.Handle, Set<ClassReference.Handle>> subClassMap;

    public InheritanceMap(Map<ClassReference.Handle, Set<ClassReference.Handle>> inheritanceMap) {
        this.inheritanceMap = inheritanceMap;
        subClassMap = new HashMap<>();
        for (Map.Entry<ClassReference.Handle, Set<ClassReference.Handle>> entry : inheritanceMap.entrySet()) {
            ClassReference.Handle child = entry.getKey();
            for (ClassReference.Handle parent : entry.getValue()) {
                subClassMap.computeIfAbsent(parent, k -> new HashSet<>()).add(child);
            }
        }
    }

    ...
}
```

最后，对于收集到的继承、实现关系数据，通过调用InheritanceDeriver.save方法，在其内部调用DataLoader.saveData并通过InheritanceMapFactory的序列化方法，对数据进行保存

```
public void save() throws IOException {
    //inheritanceMap.dat数据格式：
    //类名 父类或超类或接口类1 父类或超类或接口类2 父类或超类或接口类3 ...
    DataLoader.saveData(Paths.get("inheritanceMap.dat"), new InheritanceMapFactory(), inheritanceMap.entrySet());
}
```

```
private static class InheritanceMapFactory implements DataFactory<Map.Entry<ClassReference.Handle, Set<ClassReference.Handle>>> {
    ...

    @Override
    public String[] serialize(Map.Entry<ClassReference.Handle, Set<ClassReference.Handle>> obj) {
        final String[] fields = new String[obj.getValue().size()+1];
        fields[0] = obj.getKey().getName();
        int i = 1;
        for (ClassReference.Handle handle : obj.getValue()) {
            fields[i++] = handle.getName();
        }
        return fields;
    }
}
```

最终保存到inheritanceMap.dat文件中的数据格式：

```
类名 父类或超类或接口类1 父类或超类或接口类2 父类或超类或接口类3 ...
```

0x04 方法入参和返回值污点分析-PassthroughDiscovery

在这一小节中，我主要讲解的是PassthroughDiscovery中的代码，该部分也是整个gadgetinspector中比较核心的部分，我在阅读相关代码的时候，通过查看网络上的一些资料、博文，他们对于大体原理的讲解，都分析得比较详细，其中有一篇https://paper.seebug.org/1034/，个人觉得讲得非常不错，其中就有关于逆拓扑结构等部分，在阅读本文章的时候，大家可以同时阅读这篇文章，相互结合着看，会有意向不到的效果，但该文章也有部分细节讲得不够透彻，其中就有ASM实现细节部分，而本篇文章，这一部分章节部分原因是为了弥补它的细节不足处而编写，还有就是主要为了阐述我对gadgetinspector的理解。

在讲这部分代码之前，我想要展示一个代码例子：

```
public void main(String args) throws IOException {
    String cmd = new A().method(args);
    Runtime.getRuntime().exec(cmd);
}

class A {
  public String method(String param) {
    return param;
  }
}
```

从上述代码，我们可以看到类A和方法method，方法method接收到参数后，通过return返回，接着赋值给main方法中的cmd变量，最后Runtime.exec执行命令。

所以，根据上面代码展示，我们只要能控制method这个方法的入参，就能控制其方法的返回值，并控制数据流最终流向Runtime.exec。这其实类似于污点分析，而在PassthroughDiscovery这个类的处理阶段中，最主要就是做这样的一件事，通过不断的分析所有的方法，它们是否会被入参所污染。

还有就是，方法数据流的传递，不仅仅是一层两层，可能在整个gadget chain中，会牵涉到非常之多的方法，那么，对于所有方法数据流的污点分析，其分析顺序将会是成功与否的前提条件。这边继续讲一个例子吧：

```
public void main(String args) throws IOException {
    String cmd = new A().method1(args);
    new B().method2(cmd);
}
class A {
  public String method1(String param) {
    return param;
  }
}
class B {
  public void method2(String param) throws IOException {
    new C().method3(param);
  }
}
class C {
  public void method3(String param) throws IOException {
    Runtime.getRuntime().exec(param);
  }
}
```

上述代码，可以看到source-slink之间的具体流程，经过数据流的污点分析，我们可以得到结果：

```
A$method1-1
B$method2-1
C$method3-1
```

从代码上分析，因为A.method1的入参我们可以控制，并且其返回值间接的也被入参控制，接着赋值给了cmd变量，那么就表示cmd这个变量我们也是可以控制的，接着调用B.method2，cmd变量作为入参，并接着再把其入参作为C.method3的入参，最终走到Runtime.getRuntime().exec(param)，那么，就意味着只要我们控制了A.method1的入参，最终我们可以通过这个数据，最终影响整个source->slink，并最终得到执行exec。

而从上面的代码流程，我们只要搞明白了A类的method1方法、B类的method2方法以及C类的method3方法能被哪个参数污染下去，那么，我们就能确定整个source至slink的污点传递，但是，这里有个问题，在得到B类的method2方法参数的污染结果之前，必须得先把C类的method3方法参数的污染结果得到，而具体怎么做到呢？在gadgetinspector中，通过了DTS，一种逆拓扑顺序的方式，先得到方法执行链的逆序排序的方法集合，然后由此，从最末端进行参数污点分析，倒着回来，也就是，我先确认C类的method3方法参数的污染结果，并存储起来，接着进行分析B类的method2方法的时候，就能根据前面得到的结果，继续分析下去，最后得到B类的method2方法的参数污染结果。

那么，逆拓扑顺序的具体代码实现是如何呢？

我们跟进passthroughDiscovery.discover方法

```
//加载文件记录的所有方法信息
Map<MethodReference.Handle, MethodReference> methodMap = DataLoader.loadMethods();
//加载文件记录的所有类信息
Map<ClassReference.Handle, ClassReference> classMap = DataLoader.loadClasses();
//加载文件记录的所有类继承、实现关联信息
InheritanceMap inheritanceMap = InheritanceMap.load();
```

可以看到前三个操作分别是加载前面MethodDiscovery收集到的类、方法、继承实现的信息

接着，调用discoverMethodCalls方法，整理出所有方法，调用者方法caller和被调用者target方法之间映射的集合

```
//搜索方法间的调用关系，缓存至methodCalls集合，返回 类名->类资源 映射集合
Map<String, ClassResourceEnumerator.ClassResource> classResourceByName = discoverMethodCalls(classResourceEnumerator);
```

通过ASM Visitor的方式，使用MethodCallDiscoveryClassVisitor这个ClassVisitor实现类进行方法调用的收集

```
private Map<String, ClassResourceEnumerator.ClassResource> discoverMethodCalls(final ClassResourceEnumerator classResourceEnumerator
    Map<String, ClassResourceEnumerator.ClassResource> classResourcesByName = new HashMap<>();
    for (ClassResourceEnumerator.ClassResource classResource : classResourceEnumerator.getAllClasses()) {
        try (InputStream in = classResource.getInputStream()) {
            ClassReader cr = new ClassReader(in);
            try {
                MethodCallDiscoveryClassVisitor visitor = new MethodCallDiscoveryClassVisitor(Opcodes.ASM6);
                cr.accept(visitor, ClassReader.EXPAND_FRAMES);
                classResourcesByName.put(visitor.getName(), classResource);
            } catch (Exception e) {
                LOGGER.error("Error analyzing: " + classResource.getName(), e);
            }
        }
    }
    return classResourcesByName;
}
```

MethodCallDiscoveryClassVisitor中的运转流程：

```
private class MethodCallDiscoveryClassVisitor extends ClassVisitor {
    public MethodCallDiscoveryClassVisitor(int api) {
        super(api);
    }

    private String name = null;

    @Override
    public void visit(int version, int access, String name, String signature,
                      String superName, String[] interfaces) {
        super.visit(version, access, name, signature, superName, interfaces);
        if (this.name != null) {
            throw new IllegalStateException("ClassVisitor already visited a class!");
        }
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public MethodVisitor visitMethod(int access, String name, String desc,
                                     String signature, String[] exceptions) {
        MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
        //在visit每个method的时候，创建MethodVisitor对method进行观察
        MethodCallDiscoveryMethodVisitor modelGeneratorMethodVisitor = new MethodCallDiscoveryMethodVisitor(
                api, mv, this.name, name, desc);

        return new JSRInlinerAdapter(modelGeneratorMethodVisitor, access, name, desc, signature, exceptions);
    }

    @Override
    public void visitEnd() {
        super.visitEnd();
    }
}
```

方法的执行顺序是visit->visitMethod->visitEnd，前面也说过了，ASM对于观察者模式的具体表现。

- visit：在这个方法中，把当前观察的类名赋值到了this.name
- visitMethod：在这个方法中，继续进一步的对被观察类的每一个方法细节进行观察

继续进一步对方法的观察实现类是MethodCallDiscoveryMethodVisitor：

```
private class MethodCallDiscoveryMethodVisitor extends MethodVisitor {
    private final Set<MethodReference.Handle> calledMethods;

    /**
     *
     * @param api
     * @param mv
     * @param owner 上一步ClassVisitor在visitMethod时，传入的当前class
     * @param name visit的方法名
     * @param desc visit的方法描述
     */
    public MethodCallDiscoveryMethodVisitor(final int api, final MethodVisitor mv,
                                            final String owner, String name, String desc) {
        super(api, mv);

        //创建calledMethod收集调用到的method，最后形成集合{{sourceClass,sourceMethod}:[{targetClass,targetMethod}]}
        this.calledMethods = new HashSet<>();
        methodCalls.put(new MethodReference.Handle(new ClassReference.Handle(owner), name, desc), calledMethods);
    }

    /**
     * 方法内，每一个方法调用都会执行该方法
     *
     * @param opcode 调用操作码：INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC or INVOKEINTERFACE.
     * @param owner 被调用的类名
     * @param name 被调用的方法
     * @param desc 被调用方法的描述
     * @param itf 被调用的类是否接口
     */
    @Override
    public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
        calledMethods.add(new MethodReference.Handle(new ClassReference.Handle(owner), name, desc));
        super.visitMethodInsn(opcode, owner, name, desc, itf);
    }
}
```

具体的代码，我这里也做了比较详细的注释，在MethodCallDiscoveryMethodVisitor构造方法执行的时候，会对this.calledMethods集合进行初始化，该集合的主要作用是在被观察方法对其他方法进行调用时（会执行visitMethodInsn方法），用于缓存记录被调用的方法，因此，我们可以看到visitMethodInsn方法中，执行了

```
calledMethods.add(new MethodReference.Handle(new ClassReference.Handle(owner), name, desc));
```

并且在构造方法执行的时候，集合calledMethods也会被添加到gadgetinspector.PassthroughDiscovery#methodCalls中，做全局性的收集，因此，最后我们能通过discoverMethodCalls这一个方法，实现对这样一个数据的全量收集：

```
{{sourceClass,sourceMethod}:[{targetClass,targetMethod}]}
收集哪个class的method调用了哪一个class的method关系集合
```

接着，在下一步，通过调用

```
List<MethodReference.Handle> sortedMethods = topologicallySortMethodCalls();
```

完成了对上述收集到的数据：

```
{{sourceClass,sourceMethod}:[{targetClass,targetMethod}]}
```

实现逆拓扑的排序，跟进topologicallySortMethodCalls方法

```
Map<MethodReference.Handle, Set<MethodReference.Handle>> outgoingReferences = new HashMap<>();
for (Map.Entry<MethodReference.Handle, Set<MethodReference.Handle>> entry : methodCalls.entrySet()) {
    MethodReference.Handle method = entry.getKey();
    outgoingReferences.put(method, new HashSet<>(entry.getValue()));
}
```

第一步，对methodCalls的数据进行了封装整理，形成了Map<MethodReference.Handle, Set<MethodReference.Handle>>这样结构的数据

```
// Topological sort methods
LOGGER.debug("Performing topological sort...");
Set<MethodReference.Handle> dfsStack = new HashSet<>();
Set<MethodReference.Handle> visitedNodes = new HashSet<>();
List<MethodReference.Handle> sortedMethods = new ArrayList<>(outgoingReferences.size());
for (MethodReference.Handle root : outgoingReferences.keySet()) {
    //遍历集合中的起始方法，进行递归搜索DFS，通过逆拓扑排序，调用链的最末端排在最前面，
    // 这样才能实现入参、返回值、函数调用链之间的污点影响
    dfsTsort(outgoingReferences, sortedMethods, visitedNodes, dfsStack, root);
}
LOGGER.debug(String.format("Outgoing references %d, sortedMethods %d", outgoingReferences.size(), sortedMethods.size()));
```

```
private static void dfsTsort(Map<MethodReference.Handle, Set<MethodReference.Handle>> outgoingReferences,
                             List<MethodReference.Handle> sortedMethods, Set<MethodReference.Handle> visitedNodes,
                             Set<MethodReference.Handle> stack, MethodReference.Handle node) {

if (stack.contains(node)) {
    return;
}
if (visitedNodes.contains(node)) {
    return;
}
//根据起始方法，取出被调用的方法集
Set<MethodReference.Handle> outgoingRefs = outgoingReferences.get(node);
if (outgoingRefs == null) {
    return;
}

//入栈，以便于递归不造成类似循环引用的死循环整合
stack.add(node);
for (MethodReference.Handle child : outgoingRefs) {
    dfsTsort(outgoingReferences, sortedMethods, visitedNodes, stack, child);
}
stack.remove(node);
visitedNodes.add(node);//记录已被探索过的方法，用于在上层调用遇到重复方法时可以跳过
sortedMethods.add(node);//递归完成的探索，会添加进来
}
```

接着，通过遍历每个方法，并调用dfsTsort实现逆拓扑排序，具体细节示意图，我前面推荐的那篇文章画得非常不错，建议此时去看看

1. dfsStack用于在在逆拓扑时候不会形成环
2. visitedNodes在一条调用链出现重合的时候，不会造成重复的排序
3. sortedMethods最终逆拓扑排序出来的结果集合

最终，实现的效果如下：

```
public void main(String args) throws IOException {
    String cmd = new A().method1(args);
    new B().method2(cmd);
}
class A {
  public String method1(String param) {
    return param;
  }
}
class B {
  public void method2(String param) throws IOException {
    new C().method3(param);
  }
}
class C {
  public void method3(String param) throws IOException {
    Runtime.getRuntime().exec(param);
  }
}
```

调用链main->A.method1,main->B.method2->C.method3

排序后的结果：

```
A.method1
C.method3
B.method2
main
```

通过这样的一个结果，就如我们前面所讲的，就能在污点分析方法参数的时候，根据这个排序后的集合顺序进行分析，从而在最末端开始进行，在上一层也能通过缓存取到下层方法已经过污点分析的结果，继而继续走下去。

这些，便是逆拓扑排序的实现以及意义。

接着，就到重头戏了，我这篇文章最想要描述的ASM怎么进行参数和返回结果之间的污点分析

```
/**
 * classResourceByName：类资源集合
 * classMap：类信息集合
 * inheritanceMap：继承、实现关系集合
 * sortedMethods：方法集合
 * SerializableDecider：决策者
 */
passthroughDataflow = calculatePassthroughDataflow(classResourceByName, classMap, inheritanceMap, sortedMethods,
        config.getSerializableDecider(methodMap, inheritanceMap));
```

跟进calculatePassthroughDataflow这个方法

首先，会初始化一个集合，用于收集污染结果，key对应方法名，value对应可以污染下去的参数索引集合

```
final Map<MethodReference.Handle, Set<Integer>> passthroughDataflow = new HashMap<>();
```

紧接着，遍历被排序过后的方法，并跳过static静态初始化方法，因为静态代码块我们基本上是没办法污染的，其执行的时机在类加载的阶段

```
//遍历所有方法，然后asm观察所属类，经过前面DFS的排序，调用链最末端的方法在最前面
for (MethodReference.Handle method : sortedMethods) {
    //跳过static静态初始化代码
    if (method.getName().equals("<clinit>")) {
        continue;
    }
    ...
}
```

然后根据方法信息，获取到所属的类，接着通过ASM对其进行观察

```
//获取所属类进行观察
ClassResourceEnumerator.ClassResource classResource = classResourceByName.get(method.getClassReference().getName());
try (InputStream inputStream = classResource.getInputStream()) {
    ClassReader cr = new ClassReader(inputStream);
    try {
        PassthroughDataflowClassVisitor cv = new PassthroughDataflowClassVisitor(classMap, inheritanceMap,
                passthroughDataflow, serializableDecider, Opcodes.ASM6, method);
        cr.accept(cv, ClassReader.EXPAND_FRAMES);
        passthroughDataflow.put(method, cv.getReturnTaint());//缓存方法返回值与哪个参数有关系
    } catch (Exception e) {
        LOGGER.error("Exception analyzing " + method.getClassReference().getName(), e);
    }
} catch (IOException e) {
    LOGGER.error("Unable to analyze " + method.getClassReference().getName(), e);
}
```

PassthroughDataflowClassVisitor实现中，重点在于visitMethod方法

```
//不是目标观察的method需要跳过，上一步得到的method都是有调用关系的method才需要数据流分析
if (!name.equals(methodToVisit.getName()) || !desc.equals(methodToVisit.getDesc())) {
    return null;
}
```

因为在上述构造PassthroughDataflowClassVisitor时，最后一个参数传入的便是需要观察的方法，因此，在ASM每观察到一个方法都会执行visitMethod的时候，通过此处重新判断是否我们关心的方法，只有我们关心的方法，最终才通过下一步构建PassthroughDataflowMethodVisitor对其进行方法级别的观察

```
//对method进行观察
MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
passthroughDataflowMethodVisitor = new PassthroughDataflowMethodVisitor(
        classMap, inheritanceMap, this.passthroughDataflow, serializableDecider,
        api, mv, this.name, access, name, desc, signature, exceptions);
```

继续跟进PassthroughDataflowMethodVisitor，可以看到，它继承了TaintTrackingMethodVisitor，并有以下几个方法的实现：

1. visitCode：在进入方法的第一时间，ASM会先调用这个方法
2. visitInsn：在方法体重，每一个字节码操作指令的执行，ASM都会调用这个方法
3. visitFieldInsn：对于字段的调用，ASM都会调用这个方法
4. visitMethodInsn：方法体内，一旦调用了其他方法，都会触发这个方法的调用

在展示这四个方法的具体代码前，我还要说一下其父类中的一个方法：visitVarInsn，这个方法，会在方法体内字节码操作变量时，会被调用

为了实现类似污点分析，去分析参数对方法的污染，其模仿了jvm，实现了两个个集合，分别是本地变量表和操作数栈，通过其，实现具体的污点分析，那么具体是怎么进行的呢？

在分析前，我继续贴一个代码例子：

```
public class Main {

  public String main(String args) throws IOException {
    String cmd = new A().method1(args);
    return new B().method2(cmd);
  }
}
class A {
  public String method1(String param) {
    return param;
  }
}
class B {
  public String method2(String param) {
    return new C().method3(param);
  }
}
class C {
  public String method3(String param) {
    return param;
  }
}
```

在这个例子中，通过逆拓扑排序后得到的列表为：

```
A.method1
C.method3
B.method2
main
```

那么，分析也是根据这个顺序进行

- A.method1：
  第一步，ASM对A.method1进行观察，也就是PassthroughDataflowMethodVisitor进行观察，那么，在其方法被执行开始的时候，会触发PassthroughDataflowMethodVisitor.visitCode方法的调用，在这一步的代码中，我们可以看到，会对方法是否是static方法等进行判断，接着做了一个操作，就是把入参放到了本地变量表中来，为什么要这样做呢？我们可以想象一下，一个方法内部，能用到的数据要不就是本地变量表的数据，要不就是通过字段调用的数据，那么，在分析调用其他方法，或者对返回值是否会被入参污染时的数据流动，都跟它紧密关联，为什么这样说？根据jvm字节码的操作，在调用方法前，肯定需要对相关参数进行入栈，那入栈的数据从哪里来，必然就是本地变量表或者其他字段。那么在形成这样的一个本地变量表之后，就能标识一个方法内部的数据流动，并最终确定污染结果。

```
@Override
public void visitCode() {
    super.visitCode();

    int localIndex = 0;
    int argIndex = 0;
    if ((this.access & Opcodes.ACC_STATIC) == 0) {
        //非静态方法，第一个局部变量应该为对象实例this
        //添加到本地变量表集合
        setLocalTaint(localIndex, argIndex);
        localIndex += 1;
        argIndex += 1;
    }
    for (Type argType : Type.getArgumentTypes(desc)) {
        //判断参数类型，得出变量占用空间大小，然后存储
        setLocalTaint(localIndex, argIndex);
        localIndex += argType.getSize();
        argIndex += 1;
    }
}
protected void setLocalTaint(int index, T ... possibleValues) {
    Set<T> values = new HashSet<T>();
    for (T value : possibleValues) {
        values.add(value);
    }
    savedVariableState.localVars.set(index, values);
}
```

第二步，在入参进入本地变量表之后，会执行return这个代码，并把param这个参数返回，在这个指令执行的时候会触发visitVarInsn方法，那么在进行return操作前，首先，会对其参数param进行入栈，因为param是引用类型，那么操作代码就是Opcodes.ALOAD，可以看到，代码中，从本地变量表获取了变量索引，并放入到操作数栈中来

```
@Override
public void visitVarInsn(int opcode, int var) {
    // Extend local variable state to make sure we include the variable index
    for (int i = savedVariableState.localVars.size(); i <= var; i++) {
        savedVariableState.localVars.add(new HashSet<T>());
    }

    Set<T> saved0;
    switch(opcode) {
        case Opcodes.ILOAD:
        case Opcodes.FLOAD:
            push();
            break;
        case Opcodes.LLOAD:
        case Opcodes.DLOAD:
            push();
            push();
            break;
        case Opcodes.ALOAD:
            push(savedVariableState.localVars.get(var));
            break;
        case Opcodes.ISTORE:
        case Opcodes.FSTORE:
            pop();
            savedVariableState.localVars.set(var, new HashSet<T>());
            break;
        case Opcodes.DSTORE:
        case Opcodes.LSTORE:
            pop();
            pop();
            savedVariableState.localVars.set(var, new HashSet<T>());
            break;
        case Opcodes.ASTORE:
            saved0 = pop();
            savedVariableState.localVars.set(var, saved0);
            break;
        case Opcodes.RET:
            // No effect on stack
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }

    super.visitVarInsn(opcode, var);

    sanityCheck();
}
```

第三步，执行return指令，也就触发visitInsn这个方法，因为返回的是引用类型，那么相应的指令就是Opcodes.ARETURN，可以看到，在这个case中，会从栈顶，获取刚刚入栈（第二步中visitVarInsn从本地变量表获取的参数索引）的参数索引，并存储到returnTaint中，因此，即表示A.method1这个方法的调用，参数索引为1的参数param会污染返回值。

```
@Override
public void visitInsn(int opcode) {
    switch(opcode) {
        case Opcodes.IRETURN://从当前方法返回int
        case Opcodes.FRETURN://从当前方法返回float
        case Opcodes.ARETURN://从当前方法返回对象引用
            returnTaint.addAll(getStackTaint(0));//栈空间从内存高位到低位分配空间
            break;
        case Opcodes.LRETURN://从当前方法返回long
        case Opcodes.DRETURN://从当前方法返回double
            returnTaint.addAll(getStackTaint(1));
            break;
        case Opcodes.RETURN://从当前方法返回void
            break;
        default:
            break;
    }

    super.visitInsn(opcode);
}
```

第四步，经过return之后，该方法的观察也就结束了，那么，回到gadgetinspector.PassthroughDiscovery#calculatePassthroughDataflow中，对于刚刚放到returnTaint污点分析结果，也会在其方法中，缓存到passthroughDataflow

```
ClassReader cr = new ClassReader(inputStream);
try {
    PassthroughDataflowClassVisitor cv = new PassthroughDataflowClassVisitor(classMap, inheritanceMap,
            passthroughDataflow, serializableDecider, Opcodes.ASM6, method);
    cr.accept(cv, ClassReader.EXPAND_FRAMES);
    passthroughDataflow.put(method, cv.getReturnTaint());//缓存方法返回值与哪个参数有关系
} catch (Exception e) {
    LOGGER.error("Exception analyzing " + method.getClassReference().getName(), e);
}
```

- C.method3：该方法和A.method1的污点分析流程是一样的
- B.method2：这个方法和前面连个都不一样，它内部调用了C.method3方法，因此，污点分析时，具体的细节就又不一样了

第一步，在其方法被执行开始的时候，同样会触发PassthroughDataflowMethodVisitor.visitCode方法的调用，在其中，也是做了相应的操作，把入参存到了本地变量表中来

第二步，因为方法内部即将调用C.method3，那么ASM调用visitVarInsn方法，对其参数param进行入栈，因为param是引用类型，那么操作代码就是Opcodes.ALOAD，因此，从第一步保存的本地变量表中获取变量入栈

第三步，方法内部调用了C.method3，那么，ASM就会触发visitMethodInsn方法的执行，在这一步，会先对被调用方法的入参进行处理，并把被调用方法的实例放到argTypes的第一个索引位置，后面依次放置其他参数，接着计算返回值大小。然后，因为方法调用，第二步已经把参数入栈了，而这些参数都是从本地变量表获取的，那么，可以从栈顶取到相关参数，并认为这些参数是可被控制，也就是被当前调用者caller方法污染的，最后，也就是最重点的一步，从passthroughDataflow中获取了被调用方法的参数污染结果，也就是C.method3方法被分析时候，return存储的数据，所以，这里就印证了前面为什么要使用逆拓扑排序，因为如果不这样做的话，C.method3可能在B.method2后被分析，那么，缓存就不可能存在污点分析的结果，那么就没办法对B.method2进行正确的污点分析。接着就是对从缓存取出的污染结果和入参对比，取出相应索引的污点参数，放入到resultTaint中

```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    //获取method参数类型
```

```
    Type[] argTypes = Type.getArgumentTypes(desc);
    if (opcode != Opcodes.INVOKESTATIC) {
        //如果执行的非静态方法，则把数组第一个元素类型设置为该实例对象的类型，类比局部变量表
        Type[] extendedArgTypes = new Type[argTypes.length+1];
        System.arraycopy(argTypes, 0, extendedArgTypes, 1, argTypes.length);
        extendedArgTypes[0] = Type.getObjectType(owner);
        argTypes = extendedArgTypes;
    }
    //获取返回值类型大小
    int retSize = Type.getReturnType(desc).getSize();

    Set<Integer> resultTaint;
    switch (opcode) {
        case Opcodes.INVOKESTATIC://调用静态方法
        case Opcodes.INVOKEVIRTUAL://调用实例方法
        case Opcodes.INVOKESPECIAL://调用超类构造方法，实例初始化方法，私有方法
        case Opcodes.INVOKEINTERFACE://调用接口方法
            //todo 1 构造污染参数集合，方法调用前先把操作数入栈
            final List<Set<Integer>> argTaint = new ArrayList<Set<Integer>>(argTypes.length);
            for (int i = 0; i < argTypes.length; i++) {
                argTaint.add(null);
            }

            int stackIndex = 0;
            for (int i = 0; i < argTypes.length; i++) {
                Type argType = argTypes[i];
                if (argType.getSize() > 0) {
                    //根据参数类型大小，从栈底获取入参，参数入栈是从右到左的
                    argTaint.set(argTypes.length - 1 - i, getStackTaint(stackIndex + argType.getSize() - 1));
                }
                stackIndex += argType.getSize();
            }

            //todo 2 构造方法的调用，意味参数0可以污染返回值
            if (name.equals("<init>")) {
                // Pass result taint through to original taint set; the initialized object is directly tainted by
                // parameters
                resultTaint = argTaint.get(0);
            } else {
                resultTaint = new HashSet<>();
            }

            //todo 3 前面已做逆拓扑，调用链最末端最先被visit，因此，调用到的方法必然已被visit分析过
            Set<Integer> passthrough = passthroughDataflow.get(new MethodReference.Handle(new ClassReference.Handle(owner), name, desc));
            if (passthrough != null) {
                for (Integer passthroughDataflowArg : passthrough) {
                    //判断是否和同一方法体内的其它方法返回值关联，有关联则添加到栈底，等待执行return时保存
                    resultTaint.addAll(argTaint.get(passthroughDataflowArg));
                }
            }
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }

    super.visitMethodInsn(opcode, owner, name, desc, itf);

    if (retSize > 0) {
        getStackTaint(retSize-1).addAll(resultTaint);
    }
}
}
```

第四步，接着执行return，跟前面一样，保存到passthroughDataflow

- main：最后需要分析的是main方法的入参args是否会污染到其返回值

```
public String main(String args) throws IOException {
    String cmd = new A().method1(args);
    return new B().method2(cmd);
}
```

按照上面A.method1、B.method2、C.method3的参数污染分析结果，很明显在观察main方法的时候

第一步，执行visitCode存储入参到本地变量表

第二步，执行visitVarInsn参数入栈

第三步，执行visitMethodInsn调用A.method1，A.method1被污染的返回结果，也就是参数索引会被放在栈顶

第四步，执行visitVarInsn把放在栈顶的污染参数索引，放入到本地变量表

第五步，执行visitVarInsn参数入

第六步，执行visitMethodInsn调用B.method2，被污染的返回结果会被放在栈顶

第七步，执行visitInsn，返回栈顶数据，缓存到passthroughDataflow，也就是main方法的污点分析结果

到此，ASM实现方法入参污染返回值的分析就到此为止了。

接下来，passthroughDiscovery.save方法就被调用

```
public void save() throws IOException {
    if (passthroughDataflow == null) {
        throw new IllegalStateException("Save called before discover()");
    }

    DataLoader.saveData(Paths.get("passthrough.dat"), new PassThroughFactory(), passthroughDataflow.entrySet());
}
```

也是通过DataLoader.saveData把结果一行一行的保存到passthrough.dat文件中，而每行数据的序列化，是通过PassThroughFactory实现

```
public static class PassThroughFactory implements DataFactory<Map.Entry<MethodReference.Handle, Set<Integer>>> {

    ...

    @Override
    public String[] serialize(Map.Entry<MethodReference.Handle, Set<Integer>> entry) {
        if (entry.getValue().size() == 0) {
            return null;
        }

        final String[] fields = new String[4];
        fields[0] = entry.getKey().getClassReference().getName();
        fields[1] = entry.getKey().getName();
        fields[2] = entry.getKey().getDesc();

        StringBuilder sb = new StringBuilder();
        for (Integer arg : entry.getValue()) {
            sb.append(Integer.toString(arg));
            sb.append(",");
        }
        fields[3] = sb.toString();

        return fields;
    }
}
```

最终，这一阶段分析保存下来passthrough.dat文件的数据格式：

```
类名 方法名 方法描述 能污染返回值的参数索引1,能污染返回值的参数索引2,能污染返回值的参数索引3...
```

0x05 方法调用关联-CallGraphDiscovery

在这一阶段，会进行对方法调用关联的分析，也就是方法调用者caller和方法被调用者target直接的参数关联

举个例子描述：

```
public class Main {

  public void main(String args) throws IOException {
    String cmd = new A().method1(args);
  }
}
class A {
  public String method1(String param) {
    return param;
  }
}
```

在经过这个阶段，能得到的数据：

```
调用者类名 调用者方法caller 调用者方法描述 被调用者类名 被调用者方法target 被调用者方法描述 调用者方法参数索引 调用者字段名 被调用者方
Main (Ljava/lang/String;)V main A method1 (Ljava/lang/String;)Ljava/lang/String; 1 1
```

跟回代码，gadgetinspector.CallGraphDiscovery#discover：

加载了前面几个阶段分析处理的数据

```
//加载所有方法信息
Map<MethodReference.Handle, MethodReference> methodMap = DataLoader.loadMethods();
//加载所有类信息
Map<ClassReference.Handle, ClassReference> classMap = DataLoader.loadClasses();
//加载所有父子类、超类、实现类关系
InheritanceMap inheritanceMap = InheritanceMap.load();
//加载所有方法参数和返回值的污染关联
Map<MethodReference.Handle, Set<Integer>> passthroughDataflow = PassthroughDiscovery.load();
```

接着遍历每一个class，并对其使用ASM进行观察

```
SerializableDecider serializableDecider = config.getSerializableDecider(methodMap, inheritanceMap);

for (ClassResourceEnumerator.ClassResource classResource : classResourceEnumerator.getAllClasses()) {
    try (InputStream in = classResource.getInputStream()) {
        ClassReader cr = new ClassReader(in);
        try {
            cr.accept(new ModelGeneratorClassVisitor(classMap, inheritanceMap, passthroughDataflow, serializableDecider, Opcodes.ASM6),
                    ClassReader.EXPAND_FRAMES);
        } catch (Exception e) {
            LOGGER.error("Error analyzing: " + classResource.getName(), e);
        }
    }
}
```

ModelGeneratorClassVisitor的实现没什么重点的逻辑，主要就是对每一个方法都进行了ASM的观察

```
private class ModelGeneratorClassVisitor extends ClassVisitor {

    private final Map<ClassReference.Handle, ClassReference> classMap;
    private final InheritanceMap inheritanceMap;
    private final Map<MethodReference.Handle, Set<Integer>> passthroughDataflow;
    private final SerializableDecider serializableDecider;

    ...

    @Override
    public MethodVisitor visitMethod(int access, String name, String desc,
                                     String signature, String[] exceptions) {
        MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
        ModelGeneratorMethodVisitor modelGeneratorMethodVisitor = new ModelGeneratorMethodVisitor(classMap,
                inheritanceMap, passthroughDataflow, serializableDecider, api, mv, this.name, access, name, desc, signature, exceptions);

        return new JSRInlinerAdapter(modelGeneratorMethodVisitor, access, name, desc, signature, exceptions);
    }

    ...
}
```

ModelGeneratorMethodVisitor的实现，是这一步的重点逻辑所在，因为单单文字描述可能理解不太清楚，我这边继续以一个例子进行讲解：

```
public class Main {

  private String name;

  public void main(String args) throws IOException {
    new A().method1(args, name);
  }
}
class A {
  public String method1(String param, String param2) {
    return param + param2;
  }
}
```

可以看到上述例子中，Main的main方法中，调用了A.main1方法，并且入参是main的参数args以及Main的字段name

ASM的实现流程：

- 在Main.main方法体被观察到的第一时间，ASM会调用ModelGeneratorMethodVisitor.visitCode，在这个方法中，根据参数的数量，一一形成名称arg0、arg1...，然后放入到本地变量表

```
@Override
public void visitCode() {
    super.visitCode();

    int localIndex = 0;
    int argIndex = 0;
    //使用arg前缀来表示方法入参，后续用于判断是否为目标调用方法的入参
    if ((this.access & Opcodes.ACC_STATIC) == 0) {
        setLocalTaint(localIndex, "arg" + argIndex);
        localIndex += 1;
        argIndex += 1;
    }
    for (Type argType : Type.getArgumentTypes(desc)) {
        setLocalTaint(localIndex, "arg" + argIndex);
        localIndex += argType.getSize();
        argIndex += 1;
    }
}
```

- 接着，因为即将要调用A.method1，ASM会调用visitVarInsn，把刚刚放入到本地变量表的arg0入栈

```
@Override
public void visitVarInsn(int opcode, int var) {
    // Extend local variable state to make sure we include the variable index
    for (int i = savedVariableState.localVars.size(); i <= var; i++) {
        savedVariableState.localVars.add(new HashSet<T>());
    }

    Set<T> saved0;
    switch(opcode) {
        case Opcodes.ILOAD:
        case Opcodes.FLOAD:
            push();
            break;
        case Opcodes.LLOAD:
        case Opcodes.DLOAD:
            push();
            push();
            break;
        case Opcodes.ALOAD:
            push(savedVariableState.localVars.get(var));
            break;
        case Opcodes.ISTORE:
        case Opcodes.FSTORE:
            pop();
            savedVariableState.localVars.set(var, new HashSet<T>());
            break;
        case Opcodes.DSTORE:
        case Opcodes.LSTORE:
            pop();
            pop();
            savedVariableState.localVars.set(var, new HashSet<T>());
            break;
        case Opcodes.ASTORE:
            saved0 = pop();
            savedVariableState.localVars.set(var, saved0);
            break;
        case Opcodes.RET:
            // No effect on stack
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }

    super.visitVarInsn(opcode, var);

    sanityCheck();
}
```

- 然后，ASM调用visitVarInsn把当前实例对应的参数入栈，上一步visitCode已经把实例命名为arg0存在本地变量表中，因此入栈的参数名称为arg0，截止调用visitFieldInsn获取字段name，并命名为arg0.name入栈

```
@Override
public void visitFieldInsn(int opcode, String owner, String name, String desc) {

    switch (opcode) {
        case Opcodes.GETSTATIC:
            break;
        case Opcodes.PUTSTATIC:
            break;
        case Opcodes.GETFIELD://入操作栈
            Type type = Type.getType(desc);
            if (type.getSize() == 1) {
                Boolean isTransient = null;

                // If a field type could not possibly be serialized, it's effectively transient
                if (!couldBeSerialized(serializableDecider, inheritanceMap, new ClassReference.Handle(type.getInternalName()))) {
                    isTransient = Boolean.TRUE;
                } else {
                    ClassReference clazz = classMap.get(new ClassReference.Handle(owner));
                    while (clazz != null) {
                        for (ClassReference.Member member : clazz.getMembers()) {
                            if (member.getName().equals(name)) {
                                isTransient = (member.getModifiers() & Opcodes.ACC_TRANSIENT) != 0;
                                break;
                            }
                        }
                        if (isTransient != null) {
                            break;
                        }
                        clazz = classMap.get(new ClassReference.Handle(clazz.getSuperClass()));
                    }
                }

                Set<String> newTaint = new HashSet<>();
                if (!Boolean.TRUE.equals(isTransient)) {
                    for (String s : getStackTaint(0)) {
                        newTaint.add(s + "." + name);
                    }
                }
                super.visitFieldInsn(opcode, owner, name, desc);
                //在调用方法前，都会先入栈，作为参数
                setStackTaint(0, newTaint);
                return;
            }
            break;
        case Opcodes.PUTFIELD:
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }

    super.visitFieldInsn(opcode, owner, name, desc);
}
```

- 最后ASM调用visitMethodInsn，因为Main.main调用了A.method1，在这里个环境，清楚的用代码解释了为什么前面需要把参数命名为arg0、arg1、arg0.name这样，因为需要通过这样的一个字符串名称，和被调用方法的入参进行关联，并最终形成调用者和被调用者直接的参数关联

```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    //获取被调用method的参数和类型，非静态方法需要把实例类型放在第一个元素
    Type[] argTypes = Type.getArgumentTypes(desc);
    if (opcode != Opcodes.INVOKESTATIC) {
        Type[] extendedArgTypes = new Type[argTypes.length+1];
        System.arraycopy(argTypes, 0, extendedArgTypes, 1, argTypes.length);
        extendedArgTypes[0] = Type.getObjectType(owner);
        argTypes = extendedArgTypes;
    }

    switch (opcode) {
        case Opcodes.INVOKESTATIC:
        case Opcodes.INVOKEVIRTUAL:
        case Opcodes.INVOKESPECIAL:
        case Opcodes.INVOKEINTERFACE:
            int stackIndex = 0;
            for (int i = 0; i < argTypes.length; i++) {
                //最右边的参数，就是最后入栈，即在栈顶
                int argIndex = argTypes.length-1-i;
                Type type = argTypes[argIndex];
                //操作数栈出栈，调用方法前，参数都已入栈
                Set<String> taint = getStackTaint(stackIndex);
                if (taint.size() > 0) {
                    for (String argSrc : taint) {
                        //取出出栈的参数，判断是否为当前方法的入参，arg前缀
                        if (!argSrc.substring(0, 3).equals("arg")) {
                            throw new IllegalStateException("Invalid taint arg: " + argSrc);
                        }
                        int dotIndex = argSrc.indexOf('.');
                        int srcArgIndex;
                        String srcArgPath;
                        if (dotIndex == -1) {
                            srcArgIndex = Integer.parseInt(argSrc.substring(3));
                            srcArgPath = null;
                        } else {
                            srcArgIndex = Integer.parseInt(argSrc.substring(3, dotIndex));
                            srcArgPath = argSrc.substring(dotIndex+1);
                        }
                        //记录参数流动关系
                        //argIndex：当前方法参数索引，srcArgIndex：对应上一级方法的参数索引
                        discoveredCalls.add(new GraphCall(
                                new MethodReference.Handle(new ClassReference.Handle(this.owner), this.name, this.desc),
                                new MethodReference.Handle(new ClassReference.Handle(owner), name, desc),
                                srcArgIndex,
                                srcArgPath,
                                argIndex));
                    }
                }

                stackIndex += type.getSize();
            }
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }

    super.visitMethodInsn(opcode, owner, name, desc, itf);
}
}
```

到此，gadgetinspector.CallGraphDiscovery#discover方法就结束了，然后执行gadgetinspector.CallGraphDiscovery#save对调用者-被调用者参数关系数据进行保存到callgraph.dat文件，其中数据的序列化输出格式，由GraphCall.Factory实现

```
public static class Factory implements DataFactory<GraphCall> {

    ...

    @Override
    public String[] serialize(GraphCall obj) {
        return new String[]{
                obj.callerMethod.getClassReference().getName(), obj.callerMethod.getName(), obj.callerMethod.getDesc(),
                obj.targetMethod.getClassReference().getName(), obj.targetMethod.getName(), obj.targetMethod.getDesc(),
                Integer.toString(obj.callerArgIndex),
                obj.callerArgPath,
                Integer.toString(obj.targetArgIndex),
        };
    }
}
```

数据格式：

```
调用者类名 调用者方法caller 调用者方法描述 被调用者类名 被调用者方法target 被调用者方法描述 调用者方法参数索引 调用者字段名 被调用者方
Main (Ljava/lang/String;)V main A method1 (Ljava/lang/String;)Ljava/lang/String; 1 1
```

0x06 利用链入口搜索-SourceDiscovery

在这一个阶段中，会扫描所有的class，把符合，也就是可被反序列化并且可以在反序列化执行的方法，全部查找出来，因为没有这样的入口，就算存在执行链，也没办法通过反序列化的时候进行触发。

因为入口的触发，不同的反序列化方式会存在不同是实现，因此，在gadgetinspector中，存在着多个SourceDiscovery的实现，有jackson的，java原生序列化的等等，我这里主要以jackson的SourceDiscovery实现开始分析。

先看SourceDiscovery抽象类：

```
public abstract class SourceDiscovery {

    private final List<Source> discoveredSources = new ArrayList<>();

    protected final void addDiscoveredSource(Source source) {
        discoveredSources.add(source);
    }

    public void discover() throws IOException {
        Map<ClassReference.Handle, ClassReference> classMap = DataLoader.loadClasses();
        Map<MethodReference.Handle, MethodReference> methodMap = DataLoader.loadMethods();
        InheritanceMap inheritanceMap = InheritanceMap.load();

        discover(classMap, methodMap, inheritanceMap);
    }

    public abstract void discover(Map<ClassReference.Handle, ClassReference> classMap,
                                  Map<MethodReference.Handle, MethodReference> methodMap,
                                  InheritanceMap inheritanceMap);

    public void save() throws IOException {
        DataLoader.saveData(Paths.get("sources.dat"), new Source.Factory(), discoveredSources);
    }
}
```

可以看到，它的discover实现中，加载了所以的类、方法、继承实现关系数据，接着调用抽象方法discover，然后，我们跟进jackson的具体实现中

```
public class JacksonSourceDiscovery extends SourceDiscovery {

    @Override
    public void discover(Map<ClassReference.Handle, ClassReference> classMap,
                         Map<MethodReference.Handle, MethodReference> methodMap,
                         InheritanceMap inheritanceMap) {

        final JacksonSerializableDecider serializableDecider = new JacksonSerializableDecider(methodMap);

        for (MethodReference.Handle method : methodMap.keySet()) {
            if (serializableDecider.apply(method.getClassReference())) {
                if (method.getName().equals("<init>") && method.getDesc().equals("()V")) {
                    addDiscoveredSource(new Source(method, 0));
                }
                if (method.getName().startsWith("get") && method.getDesc().startsWith("()")) {
                    addDiscoveredSource(new Source(method, 0));
                }
                if (method.getName().startsWith("set") && method.getDesc().matches("\\(L[^;]*;\\)V")) {
                    addDiscoveredSource(new Source(method, 0));
                }
            }
        }
    }

}
```

从上述代码可以看出，实现非常之简单，只是判断了方法：

1. 是否无参构造方法
2. 是否getter方法
3. 是否setter方法

为什么对于source会做这样的判断？因为对于jackson的反序列化，在其反序列化时，必须通过无参构造方法反序列化（没有则会反序列化失败），并且会根据一定情况调用其反序列化对象的getter、setter方法

在扫描所有的方法后，具备条件的method都会被添加到gadgetinspector.SourceDiscovery#discoveredSources中，并最后通过gadgetinspector.SourceDiscovery#save保存

```
public void save() throws IOException {
    DataLoader.saveData(Paths.get("sources.dat"), new Source.Factory(), discoveredSources);
}
```

保存数据的序列化实现由Source.Factory实现

```
public static class Factory implements DataFactory<Source> {

    ...

    @Override
    public String[] serialize(Source obj) {
        return new String[]{
                obj.sourceMethod.getClassReference().getName(), obj.sourceMethod.getName(), obj.sourceMethod.getDesc(),
                Integer.toString(obj.taintedArgIndex),
        };
    }
}
```

最终输出到sources.dat文件的数据形式：

```
类名 方法名 方法描述 污染参数索引
```

0x07 最终挖掘阶段-GadgetChainDiscovery

这个阶段，是gadgetinspector自动化挖掘gadget chain的最终阶段，该阶段利用前面获取到的所有数据，从source到slink进行整合分析，最终判断slink，确定是否有效的gadget chain。

分析gadgetinspector.GadgetChainDiscovery#discover代码：

加载所有的方法数据以及继承实现关系数据

```
Map<MethodReference.Handle, MethodReference> methodMap = DataLoader.loadMethods();
        InheritanceMap inheritanceMap = InheritanceMap.load();
```

重写方法的扫描

获取方法的所有实现，这是什么意思呢？因为java的继承特性，对于一个父类，它的方法实现，可以通过子孙类进行重写覆盖，为什么要这样做呢？因为多态特性，实现类只有运行时可确定，因此，需要对其所有重写实现都形成分析链，就能确保在非运行时，做到gadget chain的挖掘

```
Map<MethodReference.Handle, Set<MethodReference.Handle>> methodImplMap = InheritanceDeriver.getAllMethodImplementations(
                inheritanceMap, methodMap);
```

分析InheritanceDeriver.getAllMethodImplementations代码：

1. 获取类->方法集

```
//遍历整合，得到每个类的所有方法实现，形成 类->实现的方法集 的映射
Map<ClassReference.Handle, Set<MethodReference.Handle>> methodsByClass = new HashMap<>();
for (MethodReference.Handle method : methodMap.keySet()) {
    ClassReference.Handle classReference = method.getClassReference();
    if (!methodsByClass.containsKey(classReference)) {
        Set<MethodReference.Handle> methods = new HashSet<>();
        methods.add(method);
        methodsByClass.put(classReference, methods);
    } else {
        methodsByClass.get(classReference).add(method);
    }
}
```

1. 获取父类->子孙类集

```
//遍历继承关系数据，形成 父类->子孙类集 的映射
Map<ClassReference.Handle, Set<ClassReference.Handle>> subClassMap = new HashMap<>();
for (Map.Entry<ClassReference.Handle, Set<ClassReference.Handle>> entry : inheritanceMap.entrySet()) {
    for (ClassReference.Handle parent : entry.getValue()) {
        if (!subClassMap.containsKey(parent)) {
            Set<ClassReference.Handle> subClasses = new HashSet<>();
            subClasses.add(entry.getKey());
            subClassMap.put(parent, subClasses);
        } else {
            subClassMap.get(parent).add(entry.getKey());
        }
    }
}
```

1. 遍历每个方法，并通过查询方法类的子孙类的方法实现，确定重写方法，最后整合成 方法->重写的方法集 的映射集合，静态方法跳过，因为静态方法是不可被重写的

```
Map<MethodReference.Handle, Set<MethodReference.Handle>> methodImplMap = new HashMap<>();
for (MethodReference method : methodMap.values()) {
    // Static methods cannot be overriden
    if (method.isStatic()) {
        continue;
    }

    Set<MethodReference.Handle> overridingMethods = new HashSet<>();
    Set<ClassReference.Handle> subClasses = subClassMap.get(method.getClassReference());
    if (subClasses != null) {
        for (ClassReference.Handle subClass : subClasses) {
            // This class extends ours; see if it has a matching method
            Set<MethodReference.Handle> subClassMethods = methodsByClass.get(subClass);
            if (subClassMethods != null) {
                for (MethodReference.Handle subClassMethod : subClassMethods) {
                    if (subClassMethod.getName().equals(method.getName()) && subClassMethod.getDesc().equals(method.getDesc())) {
                        overridingMethods.add(subClassMethod);
                    }
                }
            }
        }
    }

    if (overridingMethods.size() > 0) {
        methodImplMap.put(method.getHandle(), overridingMethods);
    }
}
```

保存方法重写数据

回到gadgetinspector.GadgetChainDiscovery#discover中，接着，对扫描到的重写方法数据进行保存

```
try (Writer writer = Files.newBufferedWriter(Paths.get("methodimpl.dat"))) {
    for (Map.Entry<MethodReference.Handle, Set<MethodReference.Handle>> entry : methodImplMap.entrySet()) {
        writer.write(entry.getKey().getClassReference().getName());
        writer.write("\t");
        writer.write(entry.getKey().getName());
        writer.write("\t");
        writer.write(entry.getKey().getDesc());
        writer.write("\n");
        for (MethodReference.Handle method : entry.getValue()) {
            writer.write("\t");
            writer.write(method.getClassReference().getName());
            writer.write("\t");
            writer.write(method.getName());
            writer.write("\t");
            writer.write(method.getDesc());
            writer.write("\n");
        }
    }
}
```

保存的数据格式：

```
类名 方法名 方法描述
\t重写方法的类名 方法名 方法描述
\t重写方法的类名 方法名 方法描述
\t重写方法的类名 方法名 方法描述
\t重写方法的类名 方法名 方法描述
类名 方法名 方法描述
\t重写方法的类名 方法名 方法描述
\t重写方法的类名 方法名 方法描述
```

整合方法调用关联数据

在前面阶段中，扫描出来的方法调用参数关联数据，都是独立的，也就是说，例如：

```
public class Main {

  private String name;

  public void main(String args) throws IOException {
    new A().method1(args, name);
    new A().method2(args, name);
  }
}
class A {
  public String method1(String param, String param2) {
    return param + param2;
  }

  public String method2(String param, String param2) {
    return param + param2;
  }
}
```

形成的方法调用参数关联数据：

```
Main (Ljava/lang/String;)V main A method1 (Ljava/lang/String;)Ljava/lang/String; 1 1
Main (Ljava/lang/String;)V main A method2 (Ljava/lang/String;)Ljava/lang/String; 1 1
```

上面形成的数据是分为了两条独立的数据，在统一的分析中，不太利于分析，因此，对其进行了整合，因为对于这两条记录来说，其都是Main.main发起的方法调用

整合代码：

```
Map<MethodReference.Handle, Set<GraphCall>> graphCallMap = new HashMap<>();
for (GraphCall graphCall : DataLoader.loadData(Paths.get("callgraph.dat"), new GraphCall.Factory())) {
    MethodReference.Handle caller = graphCall.getCallerMethod();
    if (!graphCallMap.containsKey(caller)) {
        Set<GraphCall> graphCalls = new HashSet<>();
        graphCalls.add(graphCall);
        graphCallMap.put(caller, graphCalls);
    } else {
        graphCallMap.get(caller).add(graphCall);
    }
}
```

gadget chain的初始化

```
Set<GadgetChainLink> exploredMethods = new HashSet<>();
LinkedList<GadgetChain> methodsToExplore = new LinkedList<>();
for (Source source : DataLoader.loadData(Paths.get("sources.dat"), new Source.Factory())) {
    GadgetChainLink srcLink = new GadgetChainLink(source.getSourceMethod(), source.getTaintedArgIndex());
    if (exploredMethods.contains(srcLink)) {
        continue;
    }
    methodsToExplore.add(new GadgetChain(Arrays.asList(srcLink)));
    exploredMethods.add(srcLink);
}
```

上述代码中，加载了sources.dat文件的数据，这些数据我们前面分析过，都是利用链入口，在被反序列化的时候可被触发执行的方法

```
private static class GadgetChainLink {
    private final MethodReference.Handle method;
    private final int taintedArgIndex;

    private GadgetChainLink(MethodReference.Handle method, int taintedArgIndex) {
        this.method = method;
        this.taintedArgIndex = taintedArgIndex;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        GadgetChainLink that = (GadgetChainLink) o;

        if (taintedArgIndex != that.taintedArgIndex) return false;
        return method != null ? method.equals(that.method) : that.method == null;
    }

    @Override
    public int hashCode() {
        int result = method != null ? method.hashCode() : 0;
        result = 31 * result + taintedArgIndex;
        return result;
    }
}
```

最后形成gadget chain的初始化工作

遍历初始化后的gadget chain集合

gadget chain取出，进行链可利用的判断

```
GadgetChain chain = methodsToExplore.pop();
```

获取链的最后一个方法

```
GadgetChainLink lastLink = chain.links.get(chain.links.size()-1);
```

获取最后一个方法调用到的所有方法

```
Set<GraphCall> methodCalls = graphCallMap.get(lastLink.method);
```

遍历调用到的方法，若方法不能被污染传递，则跳过

```
for (GraphCall graphCall : methodCalls) {
    if (graphCall.getCallerArgIndex() != lastLink.taintedArgIndex) {
        continue;
    }
    ...
}
```

获取被调用方法的所有重写方法

```
Set<MethodReference.Handle> allImpls = implementationFinder.getImplementations(graphCall.getTargetMethod());
```

遍历所有重写方法，并加入链的最后一节，若已存在的链，为了避免死循环，因此会跳过

```
for (MethodReference.Handle methodImpl : allImpls) {
GadgetChainLink newLink = new GadgetChainLink(methodImpl, graphCall.getTargetArgIndex());
if (exploredMethods.contains(newLink)) {
    continue;
}
GadgetChain newChain = new GadgetChain(chain, newLink);
```

判断是否到了slink，若已到，则表示这条链可用，并缓存到discoveredGadgets中，若还没到slink，则把newChain加到集合中，随着下一次循环到的时候，再次分析下一层的调用

```
if (isSink(methodImpl, graphCall.getTargetArgIndex(), inheritanceMap)) {
    discoveredGadgets.add(newChain);
} else {
    methodsToExplore.add(newChain);
    exploredMethods.add(newLink);
}
```

slink的判断：

```
private boolean isSink(MethodReference.Handle method, int argIndex, InheritanceMap inheritanceMap) {
    if (method.getClassReference().getName().equals("java/io/FileInputStream")
            && method.getName().equals("<init>")) {
        return true;
    }
    if (method.getClassReference().getName().equals("java/io/FileOutputStream")
            && method.getName().equals("<init>")) {
        return true;
    }
    if (method.getClassReference().getName().equals("java/nio/file/Files")
        && (method.getName().equals("newInputStream")
            || method.getName().equals("newOutputStream")
            || method.getName().equals("newBufferedReader")
            || method.getName().equals("newBufferedWriter"))) {
        return true;
    }
```

```
if (method.getClassReference().getName().equals("java/lang/Runtime")
        && method.getName().equals("exec")) {
    return true;
}
/*
if (method.getClassReference().getName().equals("java/lang/Class")
        && method.getName().equals("forName")) {
    return true;
}
if (method.getClassReference().getName().equals("java/lang/Class")
        && method.getName().equals("getMethod")) {
    return true;
}
*/
// If we can invoke an arbitrary method, that's probably interesting (though this doesn't assert that we
// can control its arguments). Conversely, if we can control the arguments to an invocation but not what
// method is being invoked, we don't mark that as interesting.
if (method.getClassReference().getName().equals("java/lang/reflect/Method")
        && method.getName().equals("invoke") && argIndex == 0) {
    return true;
}
if (method.getClassReference().getName().equals("java/net/URLClassLoader")
        && method.getName().equals("newInstance")) {
    return true;
}
if (method.getClassReference().getName().equals("java/lang/System")
        && method.getName().equals("exit")) {
    return true;
}
if (method.getClassReference().getName().equals("java/lang/Shutdown")
        && method.getName().equals("exit")) {
    return true;
}
if (method.getClassReference().getName().equals("java/lang/Runtime")
        && method.getName().equals("exit")) {
    return true;
}

if (method.getClassReference().getName().equals("java/nio/file/Files")
        && method.getName().equals("newOutputStream")) {
    return true;
}

if (method.getClassReference().getName().equals("java/lang/ProcessBuilder")
        && method.getName().equals("<init>") && argIndex > 0) {
    return true;
}

if (inheritanceMap.isSubclassOf(method.getClassReference(), new ClassReference.Handle("java/lang/ClassLoader"))
        && method.getName().equals("<init>")) {
    return true;
}

if (method.getClassReference().getName().equals("java/net/URL") && method.getName().equals("openStream")) {
    return true;
}

// Some groovy-specific sinks
if (method.getClassReference().getName().equals("org/codehaus/groovy/runtime/InvokerHelper")
        && method.getName().equals("invokeMethod") && argIndex == 1) {
    return true;
}

if (inheritanceMap.isSubclassOf(method.getClassReference(), new ClassReference.Handle("groovy/lang/MetaClass"))
        && Arrays.asList("invokeMethod", "invokeConstructor", "invokeStaticMethod").contains(method.getName())) {
```

```
            return true;
        }

        return false;
    }
```

至此，整个gadgetinspector的源码浅析就结束，祝大家阅读愉快，新年将至，提前说声新年快乐！

关注 | 3　　点击收藏 | 9

上一篇： 渗透测试实战（一）　　下一篇： mysql jdbc 反序列化漏洞测试

3 条回复

threedr3am

2020-01-08 10:06:10

https://github.com/threedr3am/gadgetinspector
大部分代码都加上了注释，大家可以看这个学习，不过某些注释可能会有点问题

4　回复Ta

orich1

2020-01-09 17:02:40

厉害厉害

0　回复Ta

LFYSec

2020-07-24 23:06:36

大哥太猛了，看了一遍污点分析部分还是晕晕的。。

0　回复Ta

登录 后跟帖