深入分析GadgetInspector核心代码

4ra1n / 2021-10-18 15:56:26 / 浏览数 7368

笔者是某大学本科在读，初涉安全的萌新，才疏学浅，如果文章有错误之处还请大佬指出！

最近本来在做审计相关的事情，遇到`GadgetInspector`比较感兴趣，于是做了些深入的分析

目录

1 前言

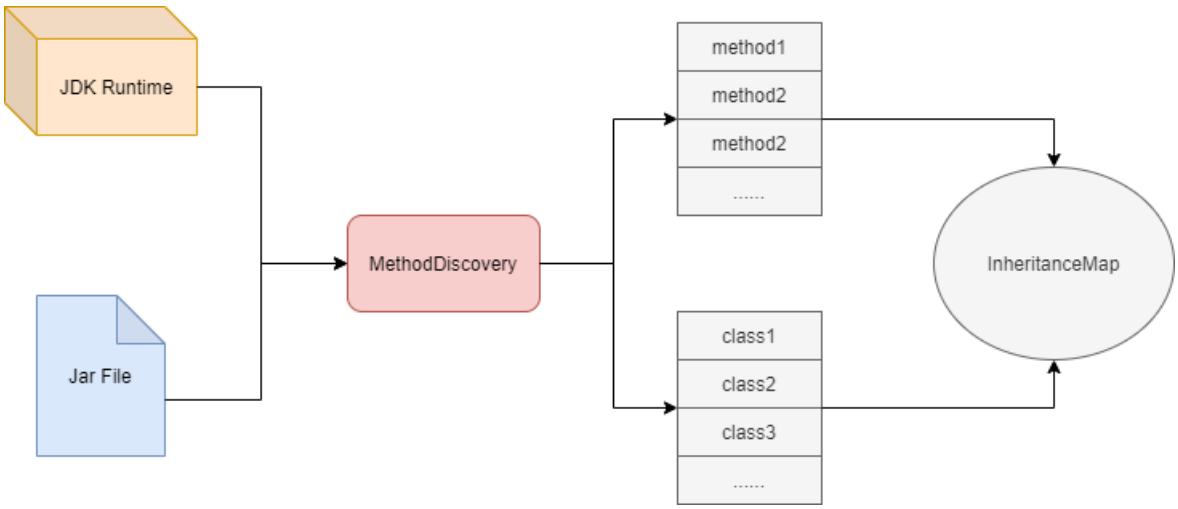
1.1 简介

GadgetInspector是**Black Hat 2018**提出的一个Java反序列化利用链自动挖掘工具，核心技术的Java ASM，结合字节码的静态分析。根据输入JAR包和JDK已有类进行分析，最终得到利用链

本文的核心是：**深入分析数据流模块（PassthroughDataflow）的每一句ASM代码**，进而把握**最底层**的原理

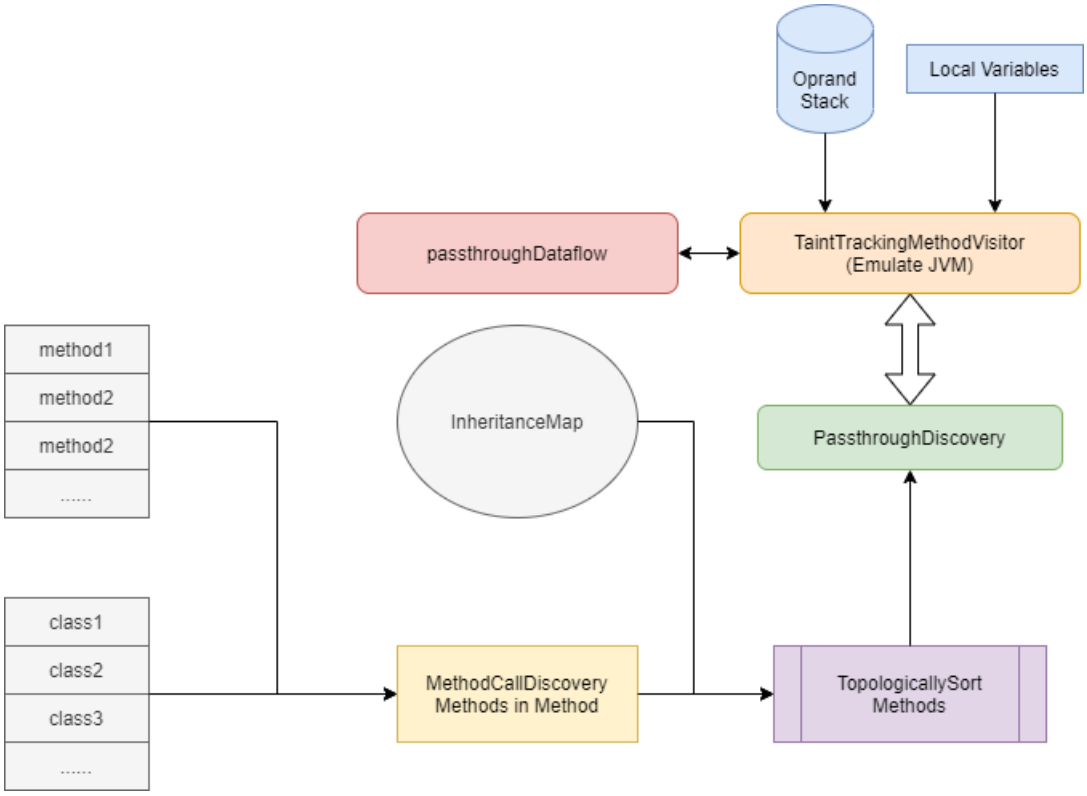
1.2 整体流程

整个流程第一步是根据 `JDK` 和输入的 `Jar` 得到所有的字节码，然后通过 `MethodDiscovery` 分析，参考第2，3章。获取所有的方法信息，类信息和继承信息。继承关系 `InheritanceMap` 指某个类的父类和实现的接口都有哪些

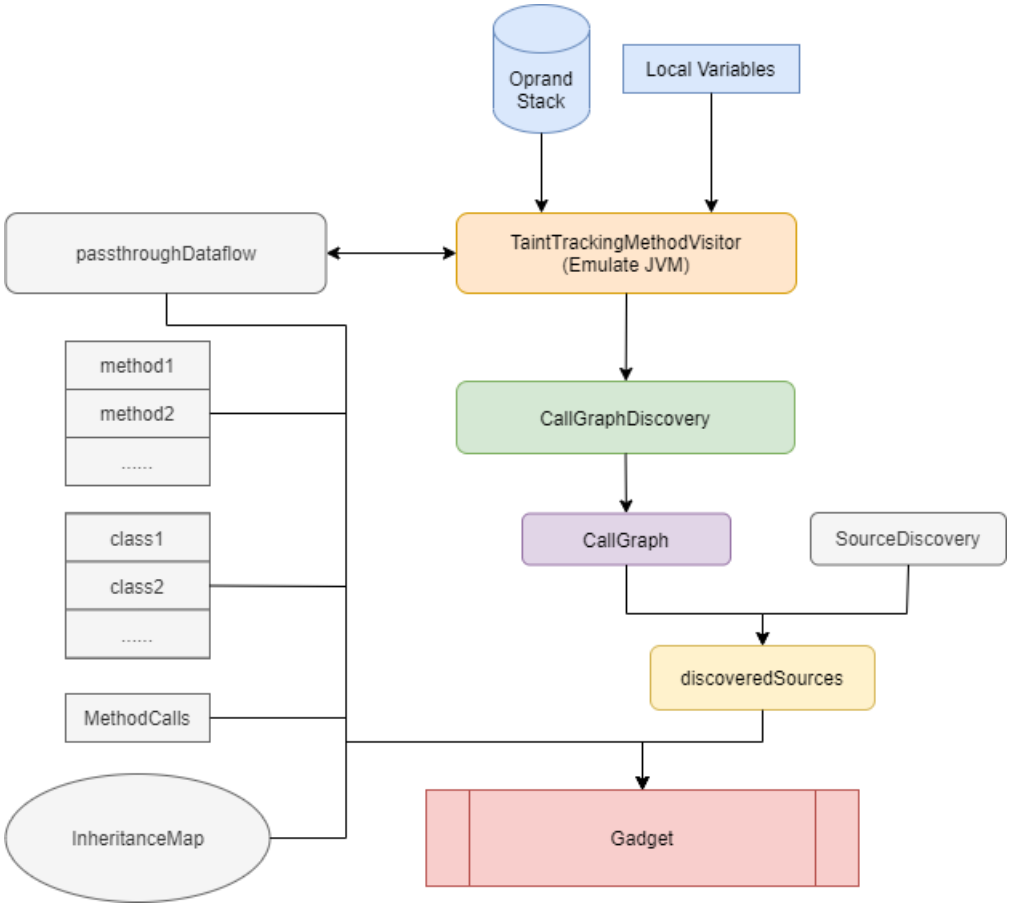


第二步是本文的核心，数据流分析确定：方法的参数和返回值之间的关系。利用第一步获得信息得到方法中的方法调用，结合 `InheritanceMap` 的继承关系，将所有方法进行拓扑逆排序（参考8.7）实现最先调用的方法在最前端。然后利用 `PassthroughDiscovery` 得到每个方法的参数和返回值之间的关系，也就是返回值能够被哪些参数污染

而 `PassthroughDiscovery` 的底层是 `TaintTrakingMethodVisitor` ，这个类是该项目的核心，参考第5节，他模拟了 `JVM Stack Frame` 中的 `Operand Stack` 和 `Local Variables Array` 让代码"**动**"起来，进而根据方法调用流程拿到具体的结果 `passthroughDataflow` ，参考第6，7和8节。这个结果从一开始是最底层的调用，所以他的第一步结果可以被第二步分析使用



后续利用上文模拟的机制，生成调用图（ `CallGraph` ）后结合漏洞触发入口（ `readObject` 等）得到 `discoveredSources` ，主要保存了方法入口和污染参数信息。在最后一步和之前所有信息合并



1.3 加载

主要是区分了 `SpringBoot Fat Jar` ， `Jar Lib` ， `War` 三种方式：

- SpringBoot Fat Jar：一个大List，加入解压后的 `BOOT-INF/classes` 路径，并加入 `BOOT-INF/lib` 下所有Jar Lib的路径，最终构造一个 `URLClassLoader` 用于获取所有字节码文件，JVM停止后自动删除解压路径
- Jar Lib：直接将所有输入的Jar Lib加入 `JarClassLoader` （该类继承自 `URLClassLoader` ）
- War：一个大List，加入解压后的 `WEB-INF/classes` 路径，并加入 `WEB-INF/lib` 下所有Jar Lib的路径，最终构造一个 `URLClassLoader` 用于获取所有字节码文件，JVM停止后自动删除解压路径

加载字节码的核心方法来自 `guava` 库的 `ClassPath.from(classLoader).getAllClasses()`

最终获得的都是 `ClassLoader` 对象，然后统一获得所有字节码文件

```
for (ClassPath.ClassInfo classInfo : ClassPath.from(classLoader).getAllClasses()) {
    result.add(new ClassLoaderClassResource(classLoader, classInfo.getResourceName()));
}
```

加载JDK的rj.jar代码如下，利用String类拿到rt.jar的路径，构造 `URLClassLoader` 然后加载

```
// java.lang.String类在rt.jar中，JDK不只是rt.jar，个别类并不属于rt.jar
URL stringClassUrl = Object.class.getResource("String.class");
URLConnection connection = stringClassUrl.openConnection();
Collection<ClassResource> result = new ArrayList<>();
if (connection instanceof JarURLConnection) {
    URL runtimeUrl = ((JarURLConnection) connection).getJarFileURL();
    URLClassLoader classLoader = new URLClassLoader(new URL[]{runtimeUrl});
    // 类似的操作
    for (ClassPath.ClassInfo classInfo : ClassPath.from(classLoader).getAllClasses()) {
        result.add(new ClassLoaderClassResource(classLoader, classInfo.getResourceName()));
    }
}
```

经过测试，`ClassPath.from`方式拿到Jar的class文件是包含rt.jar的，大概在三万多。如果jar数量多，会出现**大量的重复**，造成不小的性能问题，是否存在一种方式可以直接拿到rt.jar和输入jar的所有class文件的`inputStream`并且不重复？（已实现，后续开源）

当然，也可能是笔者本地调试的问题，由于一些特殊原因导致出现大量的重复，这点不是文章的重点，顺便提到而已

1.4 基础

基础主要是ASM技术的一些基础，需要大致明白ASM如何解析字节码

这里给出`ClassVisit`和`MethodVisit`的顺序，以便于后续的理解

ClassVisit：大体来看`visit->visitAnno->visitField或visitMethod->visitEnd`

```
visit
[visitSource][visitModule][visitNestHost][visitPermittedSubclass][visitOuterClass]
(
 visitAnnotation |
 visitTypeAnnotation |
 visitAttribute
)*
(
 visitNestMember |
 visitInnerClass |
 visitRecordComponent |
 visitField |
 visitMethod
)*
visitEnd
```

MethodVisit：大体来看`visitParam->visitAnno->visitCode->visitFrame或visitXxxInsn->visitMax->visitEnd`

```
(visitParameter)*
[visitAnnotationDefault]
(visitAnnotation | visitAnnotableParameterCount | visitParameterAnnotation | visitTypeAnnotation | visitAttribute)*
[
    visitCode
    (
        visitFrame |
        visitXxxInsn |
        visitLabel |
        visitInsnAnnotation |
        visitTryCatchBlock |
        visitTryCatchAnnotation |
        visitLocalVariable |
        visitLocalVariableAnnotation |
        visitLineNumber
    )*
    visitMaxs
]
visitEnd
```

GadgetInspector模拟了**JVM Stack**中**Frame**的**Operand Stack**和**Local Variables Array**，这一步是基础将在5.1中介绍

1.5 杂项

一些细节，比如`ClassReference.Handle`重写`equal`

可以看到判断两个类名对象`Handle`是否相等是根据字符串`name`做的，因此`hashcode`只需要根据`name`做

```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Handle handle = (Handle) o;
    return name != null ? name.equals(handle.name) : handle.name == null;
}

@Override
public int hashCode() {
    return name != null ? name.hashCode() : 0;
}
```

处理jar包内的class文件，比较巧妙。可以看到创建了一个临时目录，比如windows在`C:\\User\\AppData\\Local\\Temp`中。添加一个`shutdownHook`会在JVM退出的时候调用，在这里面删除这个临时目录。而临时目录中保存的是jar中的所有class文件，用于创建输入流，然后交给`ClassReader`做分析

```java
final Path tmpDir = Files.createTempDirectory("exploded-jar");
// Delete the temp directory at shutdown
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    try {
        deleteDirectory(tmpDir);
    } catch (IOException e) {
        LOGGER.error("Error cleaning up temp directory " + tmpDir.toString(), e);
    }
}));
```

还有一些小问题，这里就不继续写了，回到重点

2 MethodDiscoveryClassVisitor

继承自ASM的ClassVisitor，主要作用是对所有字节码中的类进行观察，下文将根据**ASM定义的visit顺序**进行分析

2.1 visit

```
// 观察到某一个类的时候会先调用visit方法
public void visit ( int version, int access, String name, String signature, String superName, String[]interfaces){
    // 给一些全局变量赋值
    this.name = name;
    this.superName = superName;
    this.interfaces = interfaces;
    // 接下来分析
    this.isInterface = (access & Opcodes.ACC_INTERFACE) != 0;
    this.members = new ArrayList<>();
    // 类名
    this.classHandle = new ClassReference.Handle(name);
    // 注解
    annotations = new HashSet<>();
    // 完成自己的逻辑后需要调用super.visit继续
    // 类似中间人攻击，不阻断正常流程，且可以在中间做事情
    super.visit(version, access, name, signature, superName, interfaces);
}
```

注意到其中的`(access & Opcodes.ACC_INTERFACE) != 0`为什么这样可以判断是否为接口，因为`Opcode`中定义如下，发现每一种标识恰好二进制某一位为1，如果按位与，只要不包含该表示，那么得出结果一定是0

```
// 0000 0000 0000 0001
int ACC_PUBLIC = 0x0001; // class, field, method
// 0000 0000 0000 0010
int ACC_PRIVATE = 0x0002; // class, field, method
// 0000 0000 0000 0100
int ACC_PROTECTED = 0x0004; // class, field, method
// 0000 0010 0000 0000
int ACC_INTERFACE = 0x0200; // class
```

2.2 visitAnnotation

注解在整个流程中没有什么实际的意义

```
// 调用完visit后会到达visitAnnotation
public AnnotationVisitor visitAnnotation(String descriptor, boolean visible) {
    annotations.add(descriptor);
    // 不阻断
    return super.visitAnnotation(descriptor, visible);
}
```

2.3 visitField

```
// visitField和visitMethod调用优先级一样
// 对类属性进行观察
public FieldVisitor visitField(int access, String name, String desc,
                               String signature, Object value) {
    // 该属性非STATIC
    if ((access & Opcodes.ACC_STATIC) == 0) {
        Type type = Type.getType(desc);
        String typeName;
        if (type.getSort() == Type.OBJECT || type.getSort() == Type.ARRAY) {
            // 属性类型非引用调用getInternalName得到名称
            typeName = type.getInternalName();
        } else {
            // 普通类型直接得到类型
            typeName = type.getDescriptor();
        }
        // 给全局变量赋值
        members.add(new ClassReference.Member(name, access, new ClassReference.Handle(typeName)));
    }
    // 传递
    return super.visitField(access, name, desc, signature, value);
}
```

2.4 visitMethod

```
// 对类里的方法进行观察
public MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions) {
    // 是否是STATIC
    boolean isStatic = (access & Opcodes.ACC_STATIC) != 0;
    //找到一个方法，添加到缓存
    discoveredMethods.add(new MethodReference(
        classHandle,
        name,
        desc,
        isStatic));
    // 传递
    return super.visitMethod(access, name, desc, signature, exceptions);
}
```

2.5 visitEnd

```
// 最后调用的一定是visitEnd方法
public void visitEnd() {
    ClassReference classReference = new ClassReference(
        name,
        superName,
        interfaces,
        isInterface,
        //把所有找到的字段（属性）封装
        members.toArray(new ClassReference.Member[members.size()]),
        annotations);
    //添加类到缓存
    discoveredClasses.add(classReference);

    super.visitEnd();
}
```

2.6 作用

得到所有类和方法信息后，进行分析获取进一步的信息，并保存供后续步骤操作

3 MethodCallDiscoveryClassVisitor

3.1 visit

```
@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!");
    }
    // 记录当前visit的类
    this.name = name;
}
```

3.2 visitMethod

在Java7以前的版本会用到jsr指令，本质原因是为了程序的兼容性，兼容Jar包和JDK一些老类

```
public MethodVisitor visitMethod(int access, String name, String desc,
                                 String signature, String[] exceptions) {
    // 先进行正常的方法观察
    MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
    // 跟入4
    MethodCallDiscoveryMethodVisitor modelGeneratorMethodVisitor = new MethodCallDiscoveryMethodVisitor(
            api, mv, this.name, name, desc);
    // 等价于return new MethodCallDiscoveryMethodVisitor(...);
    return new JSRInlinerAdapter(modelGeneratorMethodVisitor, access, name, desc, signature, exceptions);
}
```

3.3 作用

与`MethodCallDiscoveryMethodVisitor`一起记录所有方法内的所有方法调用

```
public String Test(){
    A a = new A();
    a.test1();
    // static
    B.test2();
}
```

例如这里的`test1`和`test2`就是方法内的方法调用

4 MethodCallDiscoveryMethodVisitor

4.1 构造

```
// 内部类构造方法
public MethodCallDiscoveryMethodVisitor(final int api, final MethodVisitor mv,
                                        final String owner, String name, String desc) {
    super(api, mv);
    // 记录当前方法中的所有方法调用
    this.calledMethods = new HashSet<>();
    // 将当前visit的method添加到全局变量
    methodCalls.put(new MethodReference.Handle(new ClassReference.Handle(owner), name, desc), calledMethods);
}
```

4.2 visitMethodInsn

方法中的方法相关指令

```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    // visit的方法中如果存在方法调用都加入全局变量（无论static,interface还是普通调用）
    calledMethods.add(new MethodReference.Handle(new ClassReference.Handle(owner), name, desc));
    super.visitMethodInsn(opcode, owner, name, desc, itf);
}
```

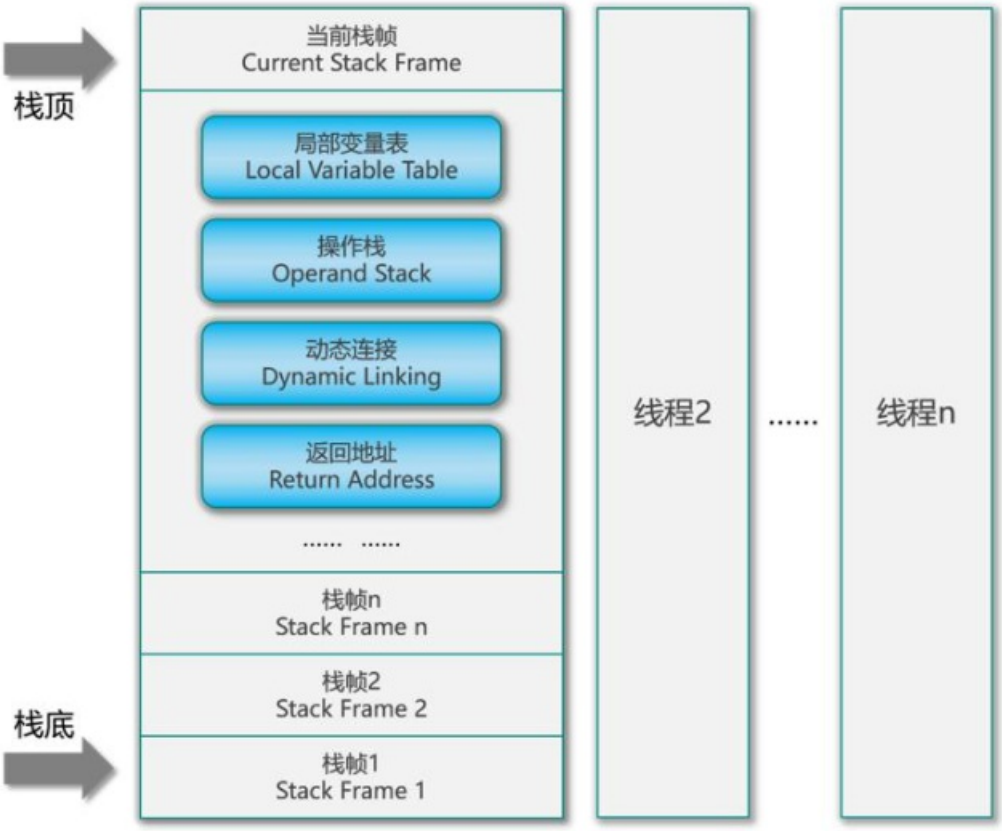
4.3 作用

与 `MethodCallDiscoveryClassVisitor` 一起记录方法内的方法调用，参考3.3

5 TaintTrackingMethodVisitor

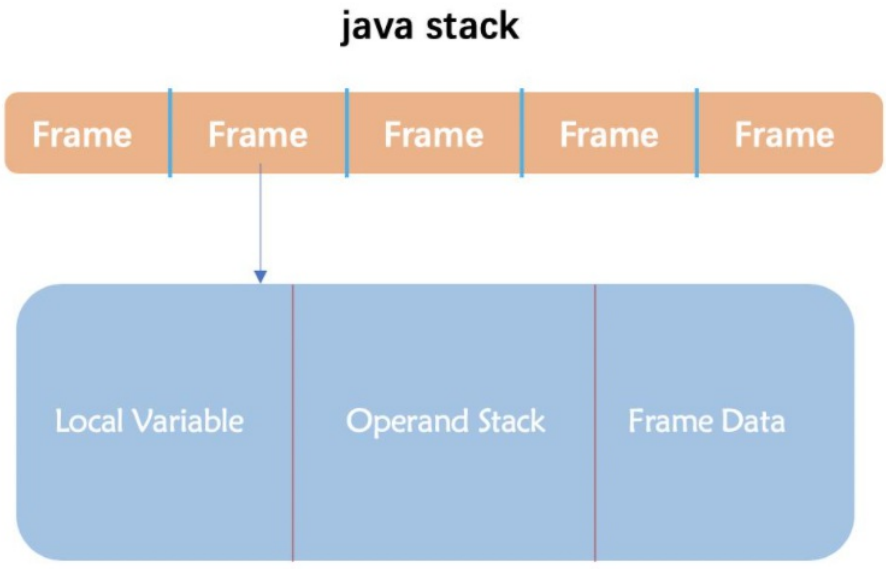
5.1 JVM Frame

分析该类离不开JVM的原理。JVM Stack在每个线程被创建时被创建，用来存放一组栈帧（Frame）



每次方法调用均会创建一个对应的Frame，方法执行完毕或者异常终止，Frame被销毁

而每个Frame的结构如下，主要由本地变量数组（local variables）和操作栈（operand stack）组成

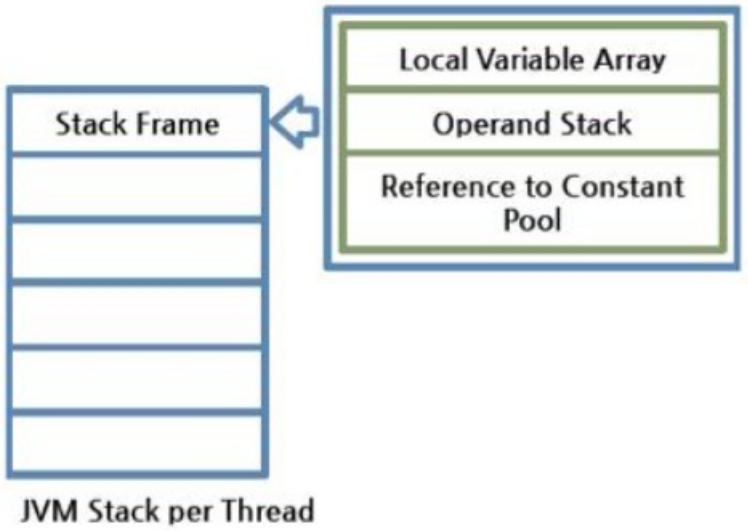


局部变量表所需的容量大小是在编译期确定下来的，表中的变量只在当前方法调用中有效

JVM把操作数栈作为它的**工作区**——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。比如，iadd指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中

例如：方法调用会从当前的Stack里弹出参数，而弹出的参数就到了新的局部变量表里，执行完返回的时候就得把返回值PUSH回

Stack。比如5.4中的visitCode做的事就是将参数放到局部变量表



之所以介绍JVM Frame，是因为代码模拟了比较完善的Operand Stack和Local Varialbles交互，例如方法调用会从Stack中弹出参数，方法返回值会压入栈中。根据这样的规则，进而执行数据流的分析

5.2 SavedVariableState

在5.1中介绍 `stack` 和 `local variables` 因为在 `TaintTrackingMethodVisitor` 中自行实现了这样的结构

注意到这里保存的Set集合，实际上代码中要么是空Set和Null做占位，要么保存的是实际有意义的值，也就是污染点

污染点的含义是参数索引，进而分析影响返回值的参数是什么。那为什么要用Set不会数组或List呢？因为Set自带去重，分析代码中会往Stack中设置多次污染信息（见后文分析）

```
private static class SavedVariableState<T> {
    // [local variables]
    List<Set<T>> localVars;
    // [operand stack]
    List<Set<T>> stackVars;
    // 新建构造
    public SavedVariableState() {
        localVars = new ArrayList<>();
        stackVars = new ArrayList<>();
    }
    // 复制构造
    public SavedVariableState(SavedVariableState<T> copy) {
        this.localVars = new ArrayList<>(copy.localVars.size());
        this.stackVars = new ArrayList<>(copy.stackVars.size());
        for (Set<T> original : copy.localVars) {
            this.localVars.add(new HashSet<>(original));
        }
        for (Set<T> original : copy.stackVars) {
            this.stackVars.add(new HashSet<>(original));
        }
    }
    // 根据传入值合并
    public void combine(SavedVariableState<T> copy) {
        for (int i = 0; i < copy.localVars.size(); i++) {
            while (i >= this.localVars.size()) {
                this.localVars.add(new HashSet<T>());
            }
            this.localVars.get(i).addAll(copy.localVars.get(i));
        }
        for (int i = 0; i < copy.stackVars.size(); i++) {
            while (i >= this.stackVars.size()) {
                this.stackVars.add(new HashSet<T>());
            }
            this.stackVars.get(i).addAll(copy.stackVars.get(i));
        }
    }
}
```

5.3 构造

有一些变量将在后文分析

```
// 根据已有类信息分析生成的[子类->[祖先类,祖先的子类...父类]]
private final InheritanceMap inheritanceMap;
// 暂不分析
private final Map<MethodReference.Handle, Set<Integer>> passthroughDataflow;
// 一个工具类，暂不分析
private final AnalyzerAdapter analyzerAdapter;
// public/private...
private final int access;
// 方法名
private final String name;
// void(int a) -> (I)V
private final String desc;
// 泛型，这里没用
private final String signature;
// 异常，这里没用
private final String[] exceptions;

public TaintTrackingMethodVisitor(...) {
    super(api, new AnalyzerAdapter(owner, access, name, desc, mv));
    // 一系列赋值
    this.inheritanceMap = inheritanceMap;
    ......
}
// stack和local variables
private SavedVariableState<T> savedVariableState = new SavedVariableState<T>();
// 处理goto问题，暂不分析
private Map<Label, SavedVariableState<T>> gotoStates = new HashMap<Label, SavedVariableState<T>>();
// 处理异常问题，暂不分析
private Set<Label> exceptionHandlerLabels = new HashSet<Label>();
```

5.4 visitCode

最先调用`visitCode`，在进入方法体的时候

这是数据流动的起始位置，注意到根据实际情况在局部变量表里设置参数，这是模拟JVM真实的情况，以便于后续数据流分析

```
// 首先执行的方法
public void visitCode() {
    super.visitCode();
    // 清空stack和local variables
    savedVariableState.localVars.clear();
    savedVariableState.stackVars.clear();
    if ((this.access & Opcodes.ACC_STATIC) == 0) {
        // 如果方法非static那么local variables[0]=this
        // 这里没有给出实际的值，而是直接占了一位
        savedVariableState.localVars.add(new HashSet<T>());
    }
    // 方法参数类型
    for (Type argType : Type.getArgumentTypes(desc)) {
        // size只会是1或2
        for (int i = 0; i < argType.getSize(); i++) {
            // 根据size将所有参数都占位
            savedVariableState.localVars.add(new HashSet<T>());
        }
    }
}
```

5.5 push & pop & get

模拟Stack的push和pop操作

```
// 模拟stack的push
private void push(T ... possibleValues) {
    Set<T> vars = new HashSet<>();
    for (T s : possibleValues) {
        vars.add(s);
    }
    savedVariableState.stackVars.add(vars);
}
// 模拟stack的push，直接设置Set
private void push(Set<T> possibleValues) {
    savedVariableState.stackVars.add(possibleValues);
}
// 模拟stack的pop
private Set<T> pop() {
    // 注意stack后入先出，所以是最后一个
    return savedVariableState.stackVars.remove(savedVariableState.stackVars.size()-1);
}
private Set<T> get(int stackIndex) {
    // 假设stack大小为10，传入index是3，那么get的索引为6，是第7个
    return savedVariableState.stackVars.get(savedVariableState.stackVars.size()-1-stackIndex);
}
```

5.6 visitFrame

`visitFrame`在`visitCode`后调用

主要作用是根据ASM给出“正确”的Frame计算方法同步当前模拟的Stack和局部变量表，确保不出现问题

第一步判断的`F_NEW`原因可以参考ASM源码： Must be Opcodes.F_NEW for expanded frames

几个参数的意义参考ASM文档：

- type：the type of this stack map frame
- nLocal：the number of local variables in the visited frame
- local：the local variable types in this frame.This array must not be modified
- nStack：the number of operand stack elements in the visited frame
- stack：the operand stack types in this frame.This array must not be modified

```
public void visitFrame(int type, int nLocal, Object[] local, int nStack, Object[] stack) {
    if (type != Opcodes.F_NEW) {
        throw new IllegalStateException("Compressed frame encountered; class reader should use accept() with
EXPANDED_FRAMES option.");
    }
    int stackSize = 0;
    for (int i = 0; i < nStack; i++) {
        Object typ = stack[i];
        int objectSize = 1;
        // Long和double的大小为2
        if (typ.equals(Opcodes.LONG) || typ.equals(Opcodes.DOUBLE)) {
            objectSize = 2;
        }
        for (int j = savedVariableState.stackVars.size(); j < stackSize+objectSize; j++) {
            // 根据size在模拟stack中占位
            savedVariableState.stackVars.add(new HashSet<T>());
        }
        // 统计总stack大小
        stackSize += objectSize;
    }
    int localSize = 0;
    for (int i = 0; i < nLocal; i++) {
        Object typ = local[i];
        int objectSize = 1;
        // 类似
        if (typ.equals(Opcodes.LONG) || typ.equals(Opcodes.DOUBLE)) {
            objectSize = 2;
        }
        // 类似，占位
        for (int j = savedVariableState.localVars.size(); j < localSize+objectSize; j++) {
            savedVariableState.localVars.add(new HashSet<T>());
        }
        // 统计
        localSize += objectSize;
    }
    // 根据统计出的真实size进行缩容，达到一致
    for (int i = savedVariableState.stackVars.size() - stackSize; i > 0; i--) {
        savedVariableState.stackVars.remove(savedVariableState.stackVars.size()-1);
    }
    // 根据统计出的真实size进行缩容，达到一致
    for (int i = savedVariableState.localVars.size() - localSize; i > 0; i--) {
        savedVariableState.localVars.remove(savedVariableState.localVars.size()-1);
    }
    // 传递
    super.visitFrame(type, nLocal, local, nStack, stack);
    // 验证
    sanityCheck();
}

private void sanityCheck() {
    // 利用analyzerAdapter计算和验证stack的size
    if (analyzerAdapter.stack != null && savedVariableState.stackVars.size() != analyzerAdapter.stack.size()) {
        throw new IllegalStateException("Bad stack size.");
    }
}
```

5.7 visitXxxInsn

根据 `opcode` 和操作数进行push和pop操作，模拟了 `JVM Frame` 中的 `OperandStack` 操作

5.7.1 visitInsn

访问0操作数指令

```
public void visitInsn(int opcode) {
    // 这些是临时变量，用于复制等指令
    Set<T> saved0, saved1, saved2, saved3;
    // 对模拟stack的size进行验证
    sanityCheck();
    switch(opcode) {
        // NOP跳过
        case Opcodes.NOP:
            break;
        // push null
        case Opcodes.ACONST_NULL:
        // push int
        case Opcodes.ICONST_M1:
        case Opcodes.ICONST_0:
        case Opcodes.ICONST_1:
        case Opcodes.ICONST_2:
        case Opcodes.ICONST_3:
        case Opcodes.ICONST_4:
        case Opcodes.ICONST_5:
        case Opcodes.FCONST_0:
        case Opcodes.FCONST_1:
        case Opcodes.FCONST_2:
            // 模拟进行一次push
            push();
            break;
        // long和double的size为2
        case Opcodes.LCONST_0:
        case Opcodes.LCONST_1:
        case Opcodes.DCONST_0:
        case Opcodes.DCONST_1:
            // size为2需要两次push
            push();
            push();
            break;
        // 这里是push各种类型的数组的索引对应的值
        // array,index -> value
        case Opcodes.IALOAD:
        case Opcodes.FALOAD:
        case Opcodes.AALOAD:
        case Opcodes.BALOAD:
        case Opcodes.CALOAD:
        case Opcodes.SALOAD:
            // 弹出数组引用和弹出index
            pop();
            pop();
            // push进去value
            push();
            break;
        // long和double的size为2
        case Opcodes.LALOAD:
        case Opcodes.DALOAD:
            // 弹出数组引用和弹出index
            pop();
            pop();
            // 两次push因为size为2
            push();
            push();
            break;
        // 弹出各种数组以及index和value
        case Opcodes.IASTORE:
        case Opcodes.FASTORE:
        case Opcodes.AASTORE:
```

```
        case Opcodes.BASTORE:
        case Opcodes.CASTORE:
        case Opcodes.SASTORE:
            // value
            pop();
            // index
            pop();
            // array
            pop();
            break;
        // 多pop一次因为size为2
        case Opcodes.LASTORE:
        case Opcodes.DASTORE:
            pop();
            pop();
            pop();
            pop();
            break;
        // 显而易见
        case Opcodes.POP:
            pop();
            break;
        // 显而易见
        case Opcodes.POP2:
            pop();
            pop();
            break;
        // 复制栈顶元素
        case Opcodes.DUP:
            push(get(0));
            break;
        // 复制栈顶插入到第2位
        case Opcodes.DUP_X1:
            saved0 = pop();
            saved1 = pop();
            push(saved0);
            push(saved1);
            push(saved0);
            break;
        // 复制栈顶插入到第3位
        case Opcodes.DUP_X2:
            saved0 = pop(); // a
            saved1 = pop(); // b
            saved2 = pop(); // c
            push(saved0); // a
            push(saved2); // c
            push(saved1); // b
            push(saved0); // a
            break;
        // 复制栈顶两个
        case Opcodes.DUP2:
            push(get(1));
            push(get(1));
            break;
        // 复制两个并向下插入
        // ..., value3 , value2 , value1 →
        // ..., value2 , value1 , value3 , value2 , value1
        case Opcodes.DUP2_X1:
            saved0 = pop();// value1
            saved1 = pop();// value2
            saved2 = pop();// value3
            push(saved1);// value2
            push(saved0);// value1
            push(saved2);// value3
            push(saved1);// value2
            push(saved0);// value1
```

```
            push(saved0);// value1
            break;
        // 复制两个再向下两个插入
        // ..., value4 , value3 , value2 , value1 →
        // ..., value2 , value1 , value4 , value3 , value2 , value1
        case Opcodes.DUP2_X2:
            saved0 = pop();
            saved1 = pop();
            saved2 = pop();
            saved3 = pop();
            push(saved1);
            push(saved0);
            push(saved3);
            push(saved2);
            push(saved1);
            push(saved0);
            break;
        // 交换栈顶和第二个元素
        case Opcodes.SWAP:
            saved0 = pop();
            saved1 = pop();
            push(saved0);
            push(saved1);
            break;
        // 取栈顶两元素做完数学操作后push结果
        case Opcodes.IADD:
        case Opcodes.FADD:
        case Opcodes.ISUB:
        case Opcodes.FSUB:
        case Opcodes.IMUL:
        case Opcodes.FMUL:
        case Opcodes.IDIV:
        case Opcodes.FDIV:
        case Opcodes.IREM:
        case Opcodes.FREM:
            pop();
            pop();
            push();
            break;
        // long和size都乘2
        case Opcodes.LADD:
        case Opcodes.DADD:
        case Opcodes.LSUB:
        case Opcodes.DSUB:
        case Opcodes.LMUL:
        case Opcodes.DMUL:
        case Opcodes.LDIV:
        case Opcodes.DDIV:
        case Opcodes.LREM:
        case Opcodes.DREM:
            pop();
            pop();
            pop();
            pop();
            push();
            push();
            break;
        // 取出栈顶元素判断是否为int
        // 结果push回去
        case Opcodes.INEG:
        case Opcodes.FNEG:
            pop();
            push();
            break;
        // long和double都乘2
        case Opcodes.LNEG:
```

```
        case Opcodes.DNEG:
            pop();
            pop();
            push();
            push();
            break;
        // 取栈顶两个进行位运算
        // 结果push回去
        case Opcodes.ISHL:
        case Opcodes.ISHR:
        case Opcodes.IUSHR:
            pop();
            pop();
            push();
            break;
        // long和double的size为2
        // 操作数是int为1所以是3次pop
        case Opcodes.LSHL:
        case Opcodes.LSHR:
        case Opcodes.LUSHR:
            pop();
            pop();
            pop();
            push();
            push();
            break;
        // 取栈顶两个进行位运算
        // 结果push回去
        case Opcodes.IAND:
        case Opcodes.IOR:
        case Opcodes.IXOR:
            pop();
            pop();
            push();
            break;
        // long和double的size为2
        // long和Long自己操作，所以是4次pop
        case Opcodes.LAND:
        case Opcodes.LOR:
        case Opcodes.LXOR:
            pop();
            pop();
            pop();
            pop();
            push();
            push();
            break;
        // 类型转换结果push回去
        case Opcodes.I2B:
        case Opcodes.I2C:
        case Opcodes.I2S:
        case Opcodes.I2F:
            pop();
            push();
            break;
        // 转long要2次push
        case Opcodes.I2L:
        case Opcodes.I2D:
            pop();
            push();
            push();
            break;
        // long转要2次pop
        case Opcodes.L2I:
        case Opcodes.L2F:
```

```
            pop();
            pop();
            push();
            break;
        // long转double各2次
        case Opcodes.D2L:
        case Opcodes.L2D:
            pop();
            pop();
            push();
            push();
            break;
        // float转int
        case Opcodes.F2I:
            pop();
            push();
            break;
        // long是2次
        case Opcodes.F2L:
        case Opcodes.F2D:
            pop();
            push();
            push();
            break;
        // double是2次
        case Opcodes.D2I:
        case Opcodes.D2F:
            pop();
            pop();
            push();
            break;
        // 比较栈顶两个long，结果push
        case Opcodes.LCMP:
            pop();
            pop();
            pop();
            pop();
            push();
            break;
        // 比较栈顶两个float，结果push
        case Opcodes.FCMPL:
        case Opcodes.FCMPG:
            pop();
            pop();
            push();
            break;
        // 比较栈顶两个double，结果push
        case Opcodes.DCMPL:
        case Opcodes.DCMPG:
            pop();
            pop();
            pop();
            pop();
            push();
            break;
        // return弹出一个
        case Opcodes.IRETURN:
        case Opcodes.FRETURN:
        case Opcodes.ARETURN:
            pop();
            break;
        // size为2
        case Opcodes.LRETURN:
        case Opcodes.DRETURN:
            pop();
            pop();
```

```
            break;
        // void没操作
        case Opcodes.RETURN:
            break;
        // 算数组长度
        case Opcodes.ARRAYLENGTH:
            pop();
            push();
            break;
        // 抛出异常类似return
        case Opcodes.ATHROW:
            pop();
            break;
        // 监视对象，弹出1个即可
        case Opcodes.MONITORENTER:
        case Opcodes.MONITOREXIT:
            pop();
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 传递
    super.visitInsn(opcode);
    // stack校验
    sanityCheck();
}
```

5.7.2 visitIntInsn

单int操作数指令

```
// 类似上文
public void visitIntInsn(int opcode, int operand) {
    switch(opcode) {
        // push一个值
        case Opcodes.BIPUSH:
        case Opcodes.SIPUSH:
            push();
            break;
        // 弹出一个size创建数组push回去引用
        case Opcodes.NEWARRAY:
            pop();
            push();
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 传递
    super.visitIntInsn(opcode, operand);
    // stack校验
    sanityCheck();
}
```

5.7.3 visitVarInsn

加载或存储局部变量值的指令

能够进行数据流动正式因为这一步从局部变量表获得或设置数据，而局部变量表数据是从5.4中获得，形成一个完整的流程

模拟JVM进行PUSH/POP操作

```java
@Override
public void visitVarInsn(int opcode, int var) {
    // 同步到本地模拟local variables
    for (int i = savedVariableState.localVars.size(); i <= var; i++) {
        savedVariableState.localVars.add(new HashSet<T>());
    }
    // 临时变量
    Set<T> saved0;
    switch(opcode) {
        // push int/float
        case Opcodes.ILOAD:
        case Opcodes.FLOAD:
            push();
            break;
        // push long/double
        case Opcodes.LLOAD:
        case Opcodes.DLOAD:
            push();
            push();
            break;
        // push object
        case Opcodes.ALOAD:
            // 从局部变量表里push一个object
            push(savedVariableState.localVars.get(var));
            break;
        // pop int/float -> 加入局部变量表
        case Opcodes.ISTORE:
        case Opcodes.FSTORE:
            pop();
            savedVariableState.localVars.set(var, new HashSet<T>());
            break;
        // size为2
        case Opcodes.DSTORE:
        case Opcodes.LSTORE:
            pop();
            pop();
            savedVariableState.localVars.set(var, new HashSet<T>());
            break;
        // pop object -> 加入局部变量表
        case Opcodes.ASTORE:
            saved0 = pop();
            savedVariableState.localVars.set(var, saved0);
            break;
        // JSR相关，对stack和局部变量表没用影响
        case Opcodes.RET:
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 传递
    super.visitVarInsn(opcode, var);
    // stack校验
    sanityCheck();
}
```

5.7.4 visitTypeInsn

以类的内部名称作为参数的指令

```
@Override
public void visitTypeInsn(int opcode, String type) {
    switch(opcode) {
        // push object
        case Opcodes.NEW:
            push();
            break;
        // 弹出个size后push数组
        case Opcodes.ANEWARRAY:
            pop();
            push();
            break;
        // 检查类型，stack不变化
        case Opcodes.CHECKCAST:
            break;
        // 判断类是否一致
        // pop类引用，push结果
        case Opcodes.INSTANCEOF:
            pop();
            push();
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 传递
    super.visitTypeInsn(opcode, type);
    // stack校验
    sanityCheck();
}
```

5.7.5 visitFieldInsn

加载或存储对象字段值的指令

程序在这一步并没有过多的操作，只是简单的POP和PUSH

```java
@Override
public void visitFieldInsn(int opcode, String owner, String name, String desc) {
    // size
    int typeSize = Type.getType(desc).getSize();
    switch (opcode) {
        // 获得static属性push进去
        case Opcodes.GETSTATIC:
            for (int i = 0; i < typeSize; i++) {
                push();
            }
            break;
        // 设置static属性pop出
        case Opcodes.PUTSTATIC:
            for (int i = 0; i < typeSize; i++) {
                pop();
            }
            break;
        // pop出对象
        // 从对象里获取的值push进去
        case Opcodes.GETFIELD:
            // ref
            pop();
            for (int i = 0; i < typeSize; i++) {
                // value
                push();
            }
            break;
        // pop出对象和值
        // 设置到对象里面（不影响stack）
        case Opcodes.PUTFIELD:
            for (int i = 0; i < typeSize; i++) {
                // value
                pop();
            }
            // ref
            pop();
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 传递
    super.visitFieldInsn(opcode, owner, name, desc);
    // stack校验
    sanityCheck();
}
```

5.7.6 visitMethodInsn

方法调用，比较核心的方法

根据方法调用需要的参数，在Stack中POP，这是对真实情况的模拟

如果是构造方法，那么`argTaint`第0位的this添加到污染

如果是`void ObjectInputStream.defaultReadObject()`不传参，这时候对象本身this就是污染，给当前局部变量表第0位设置污染（这种情况下这一步拿不到污染，在后续的数据流中得到污染）

如果目前的方法恰好匹配到白名单（很可能存在漏洞）那么白名单函数的参数位置设置到污染（其实白名单就是简化了分析，固定出了哪些类的哪些函数是存在漏洞的，它的第几个参数是可被污染的，如果匹配到白名单，直接设置该参数即可）

根据已有的`passthroughDataflow`得到与返回值有关的参数索引Set，加入污染（这一步是外部生成的，也就是Visit其他类的生成

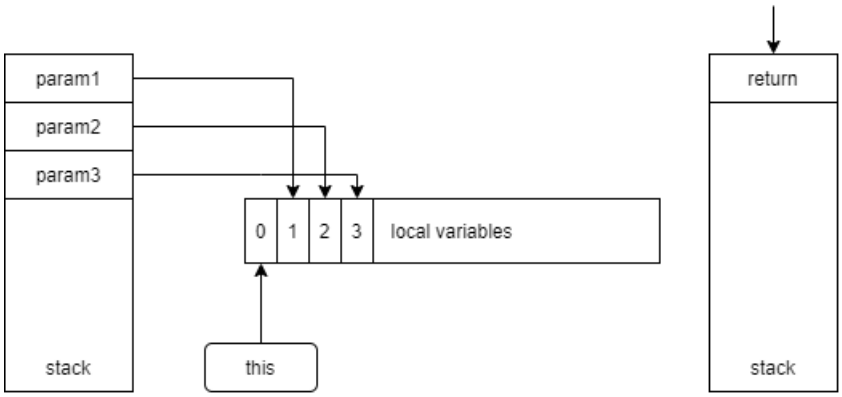
的，根据已有信息进行污点设置。参考8.7中的分析，调用链最末端的最优先被分析，因此调用到的方法必然已被visit分析过。由于Set的特性，并不会冲突，只是一次补充的效果）

如果当前类是集合类子类，认为集合中所有元素都是污染，这里不得到结果，只是设置污染，然后继续分析；如果返回对象或数组，认为返回是污染，这个结果是要并入PUSH的污染中的

最后把污染结果入栈，这模拟的就是执行完方法的PUSH返回值（代码第二次的PUSH是为了补位）

进一步的分析参考7.5

给出一个非STATIC方法调用的图




```
@Override
public void visitMethodInsn(int opcode, String owner, String name, String desc, boolean itf) {
    // 方法信息，这里Handle是重新
    final MethodReference.Handle methodHandle = new MethodReference.Handle(
            new ClassReference.Handle(owner), name, desc);
    // 所有参数类型
    Type[] argTypes = Type.getArgumentTypes(desc);
    // 非静态调用
    if (opcode != Opcodes.INVOKESTATIC) {
        // 如果执行的非静态方法，则本地变量[0]=this
        // 这里获得的参数类型argTypes中不存在this，需要手动加
        Type[] extendedArgTypes = new Type[argTypes.length+1];
        System.arraycopy(argTypes, 0, extendedArgTypes, 1, argTypes.length);
        // 把this的type加到argTypes[0]
        extendedArgTypes[0] = Type.getObjectType(owner);
        argTypes = extendedArgTypes;
    }
    // 返回类型和size
    final Type returnType = Type.getReturnType(desc);
    final int retSize = returnType.getSize();
    switch (opcode) {
        // 四种方法调用
        case Opcodes.INVOKESTATIC:
        case Opcodes.INVOKEVIRTUAL:
        case Opcodes.INVOKESPECIAL:
        case Opcodes.INVOKEINTERFACE:
            // 构造污染参数集合，方法调用前先把操作数入栈
            final List<Set<T>> argTaint = new ArrayList<Set<T>>(argTypes.length);
            for (int i = 0; i < argTypes.length; i++) {
                // 占位
                argTaint.add(null);
            }

            for (int i = 0; i < argTypes.length; i++) {
                Type argType = argTypes[i];
                // 方法调用需要消耗掉这些参数，全部pop了
```

```
                // 方法调用需要消耗掉这些参数，全部pop了
                if (argType.getSize() > 0) {
                    for (int j = 0; j < argType.getSize() - 1; j++) {
                        pop();
                    }
                    // 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
                    // 整体过程参考图片
                    argTaint.set(argTypes.length - 1 - i, pop());
                }
            }
            Set<T> resultTaint;
            // 构造
            if (name.equals("<init>")) {
                // 如果被调用的方法是构造方法，则直接通过this污染
                // 之前已经设置第0位是this
                resultTaint = argTaint.get(0);
            } else {
                resultTaint = new HashSet<>();
            }

            // void ObjectInputStream.defaultReadObject()
            if (owner.equals("java/io/ObjectInputStream") && name.equals("defaultReadObject") && desc.equals("()V"))
{
                // this加入到局部变量表，污染与参数无关
                savedVariableState.localVars.get(0).addAll(argTaint.get(0));
            }

            // 这是一个很大的白名单，在下文给出
            for (Object[] passthrough : PASSTHROUGH_DATAFLOW) {
                // 恰好匹配到item
                if (passthrough[0].equals(owner) && passthrough[1].equals(name) && passthrough[2].equals(desc)) {
                    for (int i = 3; i < passthrough.length; i++) {
                        // 保存信息
                        resultTaint.addAll(argTaint.get((Integer)passthrough[i]));
                    }
                }
            }

            // 方法返回值与哪个参数有关系（见7，8）
            if (passthroughDataflow != null) {
                // 与哪个参数有关
                Set<Integer> passthroughArgs = passthroughDataflow.get(methodHandle);
                if (passthroughArgs != null) {
                    for (int arg : passthroughArgs) {
                        // 污点是第几个参数
                        resultTaint.addAll(argTaint.get(arg));
                    }
                }
            }
            // 如果对象实现java.util.Collection或java.util.Map
            // 则假定任何接受对象的方法都污染了collection
            // 假设任何返回对象的方法都返回集合的污点
            if (opcode != Opcodes.INVOKESTATIC && argTypes[0].getSort() == Type.OBJECT) {
                // 获取被调用函数的所有基类
                Set<ClassReference.Handle> parents = inheritanceMap.getSuperClasses(new
ClassReference.Handle(argTypes[0].getClassName().replace('.', '/')));
                // 如果基类中存在collection或map
                if (parents != null && (parents.contains(new ClassReference.Handle("java/util/Collection")) ||
                        parents.contains(new ClassReference.Handle("java/util/Map")))) {
                    // 如果该类为集合类，则存储的所有元素都是污染
                    for (int i = 1; i < argTaint.size(); i++) {
                        argTaint.get(0).addAll(argTaint.get(i));
                    }
                    // 如果返回是对象或数组，认为this是污染
                    if (returnType.getSort() == Type.OBJECT || returnType.getSort() == Type.ARRAY) {
                        resultTaint.addAll(argTaint.get(0));
```

```
                    }
                }
            }
            // 如果有返回
            if (retSize > 0) {
                // 污染结果入栈
                push(resultTaint);
                // return的push从1开始，少1位
                for (int i = 1; i < retSize; i++) {
                    push();
                }
            }
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 传递
    super.visitMethodInsn(opcode, owner, name, desc, itf);
    // stack校验
    sanityCheck();
}
```

白名单

```
private static final Object[][] PASSTHROUGH_DATAFLOW = new Object[][] {
     { "java/lang/Object", "toString", "()Ljava/lang/String;", 0 },
     { "java/io/ObjectInputStream", "readObject", "()Ljava/lang/Object;", 0},
     { "java/io/ObjectInputStream", "readFields", "()Ljava/io/ObjectInputStream$GetField;", 0},
     ......
 }
```

5.7.7 visitInvokeDynamicInsn

动态调用方法

```
@Override
public void visitInvokeDynamicInsn(String name, String desc, Handle bsm, Object... bsmArgs) {
    // 参数总size
    int argsSize = 0;
    for (Type type : Type.getArgumentTypes(desc)) {
        argsSize += type.getSize();
    }
    // 返回size
    int retSize = Type.getReturnType(desc).getSize();

    // 方法调用需要pop参数
    for (int i = 0; i < argsSize; i++) {
        pop();
    }
    // 返回值需要push进去
    for (int i = 0; i < retSize; i++) {
        push();
    }
    // 传递
    super.visitInvokeDynamicInsn(name, desc, bsm, bsmArgs);
    // stack校验
    sanityCheck();
}
```

5.7.8 visitJumpInsn

跳到其他操作的操作

jump后应该处理Stack和局部变量表的问题

```
@Override
public void visitJumpInsn(int opcode, Label label) {
    switch (opcode) {
        // 取出栈顶元素判断
        case Opcodes.IFEQ:
        case Opcodes.IFNE:
        case Opcodes.IFLT:
        case Opcodes.IFGE:
        case Opcodes.IFGT:
        case Opcodes.IFLE:
        case Opcodes.IFNULL:
        case Opcodes.IFNONNULL:
            pop();
            break;
        // 取出栈顶两个元素判断
        case Opcodes.IF_ICMPEQ:
        case Opcodes.IF_ICMPNE:
        case Opcodes.IF_ICMPLT:
        case Opcodes.IF_ICMPGE:
        case Opcodes.IF_ICMPGT:
        case Opcodes.IF_ICMPLE:
        case Opcodes.IF_ACMPEQ:
        case Opcodes.IF_ACMPNE:
            pop();
            pop();
            break;
        // goto
        case Opcodes.GOTO:
            break;
        // 跳转子程序
        // push地址
        case Opcodes.JSR:
            push();
            super.visitJumpInsn(opcode, label);
            return;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 合并goto后的stack和Local variables
    mergeGotoState(label, savedVariableState);
    // 传递
    super.visitJumpInsn(opcode, label);
    // stack校验
    sanityCheck();
}

private void mergeGotoState(Label label, SavedVariableState savedVariableState) {
    if (gotoStates.containsKey(label)) {
        // goto需要合并stack和Local variables
        SavedVariableState combinedState = new SavedVariableState(gotoStates.get(label));
        combinedState.combine(savedVariableState);
        // 出现过的label直接合并
        gotoStates.put(label, combinedState);
    } else {
        gotoStates.put(label, new SavedVariableState(savedVariableState));
    }
}
```

5.7.9 visitLabel

标签指定了紧接着它将被访问的指令

```
@Override
public void visitLabel(Label label) {
    // 如果跳转Label已被初始化过
    if (gotoStates.containsKey(label)) {
        // 从已被初始化过的地方拿到stack和局部变量表信息
        savedVariableState = new SavedVariableState(gotoStates.get(label));
    }
    // 如果是异常Label，类似return需要把异常push进去
    if (exceptionHandlerLabels.contains(label)) {
        // 只是占位
        push(new HashSet<T>());
    }
    // 传递
    super.visitLabel(label);
    // stack校验
    sanityCheck();
}
```

5.7.10 visitLdcInsn

常量池操作

```
@Override
public void visitLdcInsn(Object cst) {
    // push 常量池
    // size为2
    if (cst instanceof Long || cst instanceof Double) {
        push();
        push();
    } else {
        // size为1
        push();
    }
    // 传递
    super.visitLdcInsn(cst);
    // stack校验
    sanityCheck();
}
```

5.7.11 visitTableSwitchInsn

通过索引访问跳转表并跳转

```
@Override
public void visitTableSwitchInsn(int min, int max, Label dflt, Label... labels) {
    // pop index
    pop();
    // 根据跳转Label合并状态
    mergeGotoState(dflt, savedVariableState);
    for (Label label : labels) {
        mergeGotoState(label, savedVariableState);
    }
    // 传递
    super.visitTableSwitchInsn(min, max, dflt, labels);
    // stack校验
    sanityCheck();
}
```

5.7.12 visitLookupSwitchInsn

通过键匹配和跳转访问跳转表

任何跳转都需要处理Stack和局部变量表

```
@Override
public void visitLookupSwitchInsn(Label dflt, int[] keys, Label[] labels) {
    // pop key
    pop();

    // 根据跳转Label合并状态
    mergeGotoState(dflt, savedVariableState);
    for (Label label : labels) {
        mergeGotoState(label, savedVariableState);
    }
    // 传递
    super.visitLookupSwitchInsn(dflt, keys, labels);
    // stack校验
    sanityCheck();
}
```

5.7.13 visitMultiANewArrayInsn

创建新的多维数组

```
@Override
public void visitMultiANewArrayInsn(String desc, int dims) {
    // 每个维度有个size
    for (int i = 0; i < dims; i++) {
        pop();
    }
    // 创建完把引用push回去
    push();
    // 传递
    super.visitMultiANewArrayInsn(desc, dims);
    // stack校验
    sanityCheck();
}
```

5.7.14 visitOthers

这部分基本没有业务逻辑，也没有POP/PUSH操作

```java
@Override
public AnnotationVisitor visitInsnAnnotation(int typeRef, TypePath typePath, String desc, boolean visible) {
    return super.visitInsnAnnotation(typeRef, typePath, desc, visible);
}

@Override
public void visitTryCatchBlock(Label start, Label end, Label handler, String type) {
    // 异常Label保存
    exceptionHandlerLabels.add(handler);
    super.visitTryCatchBlock(start, end, handler, type);
}

@Override
public AnnotationVisitor visitTryCatchAnnotation(int typeRef, TypePath typePath, String desc, boolean visible) {
    return super.visitTryCatchAnnotation(typeRef, typePath, desc, visible);
}

@Override
public void visitMaxs(int maxStack, int maxLocals) {
    super.visitMaxs(maxStack, maxLocals);
}

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

5.8 xxxTaint

一些出入栈和局部变量表的操作

这些方法并不影响Stack的POP/PUSH，只是往已有的位置设置污染信息

```
protected Set<T> getStackTaint(int index) {
    // 出栈，注意是栈结构，index=0为栈顶
    return savedVariableState.stackVars.get(savedVariableState.stackVars.size()-1-index);
}
protected void setStackTaint(int index, T ... possibleValues) {
    Set<T> values = new HashSet<T>();
    for (T value : possibleValues) {
        values.add(value);
    }
    // 入栈，index=0为栈顶
    savedVariableState.stackVars.set(savedVariableState.stackVars.size()-1-index, values);
}
protected void setStackTaint(int index, Collection<T> possibleValues) {
    // 同上
    Set<T> values = new HashSet<T>();
    values.addAll(possibleValues);
    savedVariableState.stackVars.set(savedVariableState.stackVars.size()-1-index, values);
}

protected Set<T> getLocalTaint(int index) {
    // 局部变量表直接操作
    return savedVariableState.localVars.get(index);
}
protected void setLocalTaint(int index, T ... possibleValues) {
    // 局部变量表直接操作
    Set<T> values = new HashSet<T>();
    for (T value : possibleValues) {
        values.add(value);
    }
    savedVariableState.localVars.set(index, values);
}
protected void setLocalTaint(int index, Collection<T> possibleValues) {
    // 局部变量表直接操作
    Set<T> values = new HashSet<T>();
    values.addAll(possibleValues);
    savedVariableState.localVars.set(index, values);
}
```

5.9 couldBeSerialized

是否能被序列化，决策者其实就是一个方法，用来判断什么情况下可以被序列化，什么情况下存在漏洞

```
protected static final boolean couldBeSerialized(SerializableDecider serializableDecider, InheritanceMap
inheritanceMap, ClassReference.Handle clazz) {
        // 决策者后续分析
        if (Boolean.TRUE.equals(serializableDecider.apply(clazz))) {
            return true;
        }
        // 获取所有子类
        Set<ClassReference.Handle> subClasses = inheritanceMap.getSubClasses(clazz);
        if (subClasses != null) {
            // 遍历所有子类是否存在可被序列化的class
            for (ClassReference.Handle subClass : subClasses) {
                // 使用决策者判断是否可被序列化
                if (Boolean.TRUE.equals(serializableDecider.apply(subClass))) {
                    return true;
                }
            }
    }
    return false;
}
```

比如Jackson的决策者

```
@Override
public Boolean apply(ClassReference.Handle handle) {
    if (isNoGadgetClass(handle)) {
        return false;
    }
    // 类是否通过决策的缓存集合
    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;
}

private boolean isNoGadgetClass(ClassReference.Handle clazz) {
    // 黑名单匹配
    if (JacksonSourceDiscovery.skipList.contains(clazz.getName())) {
        return true;
    }
    return false;
}
```

5.10 作用

该类模拟了JVM中的operand Stack和Local Varaibles，个人理解相当于**把完全静态的代码做成了半动态**，结合业务逻辑代码，实现数据流动分析。POP和PUSH的都是空Set，如果分析中认为存在污点，那么就把对应Set位置设为污染

6 PassthroughDataflowClassVisitor

该类不是重点，第7条`PassthroughDataflowMethodVisitor`应重点关注

6.1 构造

```
// [类名->类信息]
Map<ClassReference.Handle, ClassReference> classMap;
// 要观察的方法
private final MethodReference.Handle methodToVisit;
// [子类->[父类,父类的父类...]]或相反
private final InheritanceMap inheritanceMap;
// 方法返回值与哪个参数有关系
private final Map<MethodReference.Handle, Set<Integer>> passthroughDataflow;
// 决策者
private final SerializableDecider serializableDecider;
// 当前visit的类名
private String name;
// 一个MethodVisitor
private PassthroughDataflowMethodVisitor passthroughDataflowMethodVisitor;

public PassthroughDataflowClassVisitor(...) {
    super(api);
    // 赋值
    this.classMap = classMap;
    this.inheritanceMap = inheritanceMap;
    this.methodToVisit = methodToVisit;
    this.passthroughDataflow = passthroughDataflow;
    this.serializableDecider = serializableDecider;
}
```

6.2 visit

```
public void visit(int version, int access, String name, String signature,
                  String superName, String[] interfaces) {
    super.visit(version, access, name, signature, superName, interfaces);
    this.name = name;
    // 当前类不是要观察方法所属的类则跳过
    if (!this.name.equals(methodToVisit.getClassReference().getName())) {
        throw new IllegalStateException("Expecting to visit " + methodToVisit.getClassReference().getName() + " but
instead got " + this.name);
    }
}
```

6.3 visitMethod

```
public MethodVisitor visitMethod(int access, String name, String desc,
                                 String signature, String[] exceptions) {
    // 当前visit的方法不是目标方法需要跳过
    if (!name.equals(methodToVisit.getName()) || !desc.equals(methodToVisit.getDesc())) {
        return null;
    }
    // method visitor不能重复
    if (passthroughDataflowMethodVisitor != null) {
        throw new IllegalStateException("Constructing passthroughDataflowMethodVisitor twice!");
    }
    // 对当前方法进行观察
    MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
    // 跟入7
    passthroughDataflowMethodVisitor = new PassthroughDataflowMethodVisitor(
            classMap, inheritanceMap, this.passthroughDataflow, serializableDecider,
            api, mv, this.name, access, name, desc, signature, exceptions);
    // 参考3.2，出于兼容性的考虑
    return new JSRInlinerAdapter(passthroughDataflowMethodVisitor, access, name, desc, signature, exceptions);
}
```

6.4 getReturnTaint

```
// 一个非visit方法，获得返回污点，返回值与哪些参数有关
public Set<Integer> getReturnTaint() {
    if (passthroughDataflowMethodVisitor == null) {
        throw new IllegalStateException("Never constructed the passthroughDataflowmethodVisitor!");
    }
    return passthroughDataflowMethodVisitor.returnTaint;
}
```

6.5 作用

与7结合分析数据流之间的污染

7 PassthroughDataflowMethodVisitor

继承自`TaintTrackingMethodVisitor`

7.1 构造

```
// 父类TaintTrackingMethodVisitor在7分析
private static class PassthroughDataflowMethodVisitor extends TaintTrackingMethodVisitor<Integer> {
    // [类名->类信息]
    private final Map<ClassReference.Handle, ClassReference> classMap;
    // [子类->[父类,父类的父类...]]或相反
    private final InheritanceMap inheritanceMap;
    // [方法名->[1,2,3]] 方法返回值与哪个参数有关系
    private final Map<MethodReference.Handle, Set<Integer>> passthroughDataflow;
    // 后文分析，决策者
    private final SerializableDecider serializableDecider;
    // 当前方法access（public/private...）
    private final int access;
    // 当前方法desc（void(int a) -> (I)V）
    private final String desc;
    // 被污染的返回
    private final Set<Integer> returnTaint;

    public PassthroughDataflowMethodVisitor(...);
        // 赋值
        this.classMap = classMap;
        this.inheritanceMap = inheritanceMap;
        this.passthroughDataflow = passthroughDataflow;
        this.serializableDecider = serializableDeciderMap;
        this.access = access;
        this.desc = desc;
        returnTaint = new HashSet<>();
    }
```

7.2 visitCode

在进入方法体的时候调用

父类先清空stack和局部变量表，重新设置局部变量表为正确的值

然后交给子类给该方法每个参数位置设置污染到局部变量表

```
// visit流程中最先调用的
public void visitCode() {
    super.visitCode();
    // local variables数组
    int localIndex = 0;
    // 参数数量
    int argIndex = 0;
    // 判断逻辑参考2.1
    if ((this.access & Opcodes.ACC_STATIC) == 0) {
        // 非静态情况，本地变量[0] = 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
        // 例如Long的大小是2，int大小为1
        setLocalTaint(localIndex, argIndex);
        localIndex += argType.getSize();
        // 参数数量
        argIndex += 1;
    }
}
```

7.3 visitInsn

无操作数的操作，注意子类和父类的顺序不可乱

子类：

如果操作是return，将返回值加入到返回污点中

父类：

进行POP/PUSH等正常操作

```
public void visitInsn(int opcode) {
    switch(opcode) {
        // 从当前方法返回int
        case Opcodes.IRETURN:
        // 从当前方法返回float
        case Opcodes.FRETURN:
        // 从当前方法返回对象引用
        case Opcodes.ARETURN:
            // 父类方法，返回污点里保存栈顶元素
            returnTaint.addAll(getStackTaint(0));
            break;
        // 从当前方法返回long
        case Opcodes.LRETURN:
        // 从当前方法返回double
        case Opcodes.DRETURN:
            // 父类方法，返回污点里保存栈顶元素（size为2）
            returnTaint.addAll(getStackTaint(1));
            break;
        // 从当前方法返回void不处理
        case Opcodes.RETURN:
            break;
        default:
            break;
    }
    // 交给父类
    super.visitInsn(opcode);
}
```

7.4 visitFieldInsn

字段（属性）相关的操作

子类：

如果opcode是**GETFIELD**，这个操作需要从Stack里**POP出一个对象再把得到的值PUSH进去**

如果这个字段类型包括基类都可以被反序列化，那么目前栈顶的这个对象就是污染

可以看到，先暂存了这个污染对象

父类：

模拟JVM的POP/PUSH操作，这时候栈顶就是PUSH进去的值

根据暂存的污染对象，把目前栈顶的值设置为污染

这样做的目的是能够让污染传递下去，从右边到左边

```
public void visitFieldInsn(int opcode, String owner, String name, String desc) {
    switch (opcode) {
        // 如果是获得STATIC变量的值跳过
        case Opcodes.GETSTATIC:
            break;
        // 如果是设置STATIC变量的值跳过
        case Opcodes.PUTSTATIC:
            break;
        // 如果是获得普通字段的值
        case Opcodes.GETFIELD:
            // 获取字段类型
```

```
            Type type = Type.getType(desc);
            // 非long和double时size为1
            if (type.getSize() == 1) {
                // 可能为引用类型
                // 是否不可序列化
                Boolean isTransient = null;
                // 父类方法，判断调用的字段类型是否可序列化
                if (!couldBeSerialized(serializableDecider, inheritanceMap, new
ClassReference.Handle(type.getInternalName()))) {
                    isTransient = Boolean.TRUE;
                } else {
                    // 若调用的字段可被序列化，则取当前类实例的所有字段，找出调用的字段，判断是否被标识了transient
                    // 找到当前的类信息
                    ClassReference clazz = classMap.get(new ClassReference.Handle(owner));
                    while (clazz != null) {
                        // 遍历字段
                        for (ClassReference.Member member : clazz.getMembers()) {
                            if (member.getName().equals(name)) {
                                // 如果字段是TRANSIENT的不可被反序列化则跳过
                                isTransient = (member.getModifiers() & Opcodes.ACC_TRANSIENT) != 0;
                                break;
                            }
                        }
                        if (isTransient != null) {
                            break;
                        }
                        // 向上父类遍历查找可被序列化字段
                        clazz = classMap.get(new ClassReference.Handle(clazz.getSuperClass()));
                    }
                }
                // 污点列表
                Set<Integer> taint;
                if (!Boolean.TRUE.equals(isTransient)) {
                    // 可被序列化
                    taint = getStackTaint(0);
                } else {
                    // 不可被序列化
                    taint = new HashSet<>();
                }
                // 父类处理
                super.visitFieldInsn(opcode, owner, name, desc);
                // 设置栈顶是污点
                setStackTaint(0, taint);
                return;
            }
            break;
        // 如果是设置普通字段的值，跳过
        case Opcodes.PUTFIELD:
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    super.visitFieldInsn(opcode, owner, name, desc);
}
```

7.5 visitMethodInsn

方法的调用需要从Stack里面取参数，最后把返回值压入Stack，参考5.7.6中的图片

关于`passthroughDataflow`，已经进行DFS排序，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（参考三梦师傅）

子类做的事：

得到模拟Stack中应该获取的参数，设置到`argTaint`

如果是构造方法，那么`argTaint`第0位this就是污染

根据已有的`passthroughDataflow`得到与返回值有关的参数索引Set，加入污染

父类做的事：

参考5.7.6，根据方法调用需要的参数，在Stack中POP

如果是构造方法，那么`argTaint`第0位this就是污染

如果是`void ObjectInputStream.defaultReadObject()`不传参，这时候对象本身this就是污染，给局部变量表第0位设置污染

如果目前的方法恰好匹配到白名单（很可能存在漏洞）那么白名单函数的参数位置设置到污染

根据已有的`passthroughDataflow`得到与返回值有关的参数索引Set，加入污染

如果当前类是集合类子类，认为集合中所有元素都是污染；如果返回对象或数组，认为返回也是污染

最后把污染结果入栈，这模拟的就是执行完方法的PUSH返回值

子类继续做：

这时候子类取到Stack顶的RETURN值，在父类的污染中再加入子类得到的污染

注意：重复添加了很多污染，会不会重复？不会，因为污染是参数的位置int值组成的Set，Set特性是不会重复

```
// 如果方法中有方法相关的操作
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);
    // 静态调用
    if (opcode != Opcodes.INVOKESTATIC) {
        // 如果执行的非静态方法，则本地变量[0]=this
        // 这里获得的参数类型argTypes中不存在this，需要手动加
        Type[] extendedArgTypes = new Type[argTypes.length+1];
        System.arraycopy(argTypes, 0, extendedArgTypes, 1, argTypes.length);
        // 把this的type加到0，后面的往后推
        extendedArgTypes[0] = Type.getObjectType(owner);
        argTypes = extendedArgTypes;
    }
    // 获取返回值类型大小
    int retSize = Type.getReturnType(desc).getSize();
    // 返回值污点
    Set<Integer> resultTaint;
    switch (opcode) {
        // 任何一种方法调用
        case Opcodes.INVOKESTATIC:
        case Opcodes.INVOKEVIRTUAL:
        case Opcodes.INVOKESPECIAL:
        case Opcodes.INVOKEINTERFACE:
            // 构造污染参数集合
            final List<Set<Integer>> argTaint = new ArrayList<Set<Integer>>(argTypes.length);
            for (int i = 0; i < argTypes.length; i++) {
                // 占位
                argTaint.add(null);
            }
            // 由于方法调用需要弹出栈中的参数
```

```
            // 由于方法调用需要弹出栈中的参数
            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));
                }
                // stack深度根据size增加
                stackIndex += argType.getSize();
            }

            // 构造方法的调用
            if (name.equals("<init>")) {
                // 参数this就是污点
                resultTaint = argTaint.get(0);
            } else {
                resultTaint = new HashSet<>();
            }

            // 方法返回值与哪个参数有关系，可能是空
            Set<Integer> passthrough = passthroughDataflow.get(new MethodReference.Handle(new
ClassReference.Handle(owner), name, desc));
            if (passthrough != null) {
                for (Integer passthroughDataflowArg : passthrough) {
                    // 加入污点
                    resultTaint.addAll(argTaint.get(passthroughDataflowArg));
                }
            }
            break;
        default:
            throw new IllegalStateException("Unsupported opcode: " + opcode);
    }
    // 传递
    super.visitMethodInsn(opcode, owner, name, desc, itf);
    // 存在返回，设置返回为污点
    if (retSize > 0) {
        getStackTaint(retSize-1).addAll(resultTaint);
    }
}
```

关于这个`passthroughDataFlow`是个全局变量，是一个缓存，visit一个类就缓存一次

```
PassthroughDataflowClassVisitor cv = new PassthroughDataflowClassVisitor(classMap, inheritanceMap,
        passthroughDataflow, serializableDecider, Opcodes.ASM6, method);
cr.accept(cv, ClassReader.EXPAND_FRAMES);
passthroughDataflow.put(method, cv.getReturnTaint());
```

7.6 作用

与6结合分析数据流之间的污染，由`5 TaintTrackingMethodVisitor`作为驱动，模拟JVM执行代码

8 PassthroughDiscovery

业务逻辑代码，同样是重点

8.1 属性

```
// 方法调用关系
private final Map<MethodReference.Handle, Set<MethodReference.Handle>> methodCalls = new HashMap<>();
// passthroughDataflow
private Map<MethodReference.Handle, Set<Integer>> passthroughDataflow;
```

8.2 discover

这里需要注意一个流程：先DFS排序，然后再进行`PassthroughDataflowMethodVisitor`的数据流污染分析，分析见8.7

```
// 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();
// 搜索方法间的调用关系，缓存至methodCalls集合，返回类名->类资源，见8.3
Map<String, ClassResourceEnumerator.ClassResource> classResourceByName =
discoverMethodCalls(classResourceEnumerator);
// 对方法调用关系进行字典排序，见8.4
List<MethodReference.Handle> sortedMethods = topologicallySortMethodCalls();
// 得到passthroughDataflow，见8.6
passthroughDataflow = calculatePassthroughDataflow(classResourceByName, classMap, inheritanceMap,
sortedMethods,config.getSerializableDecider(methodMap, inheritanceMap));
```

8.3 discoverMethodCalls

主要是结合3，4做方法内的方法调用收集，没有什么难度

```
private Map<String, ClassResourceEnumerator.ClassResource> discoverMethodCalls(final ClassResourceEnumerator
classResourceEnumerator) throws IOException {
    // className -> classResource
    Map<String, ClassResourceEnumerator.ClassResource> classResourcesByName = new HashMap<>();
    // all classes
    for (ClassResourceEnumerator.ClassResource classResource : classResourceEnumerator.getAllClasses()) {
        // 读入字节码
        try (InputStream in = classResource.getInputStream()) {
            // ASM解析
            ClassReader cr = new ClassReader(in);
            try {
                // 参考3和4，记录了方法调用
                MethodCallDiscoveryClassVisitor visitor = new MethodCallDiscoveryClassVisitor(Opcodes.ASM6);
                cr.accept(visitor, ClassReader.EXPAND_FRAMES);
                // 保存
                classResourcesByName.put(visitor.getName(), classResource);
            } catch (Exception e) {
                LOGGER.error("Error analyzing: " + classResource.getName(), e);
            }
        }
    }
    return classResourcesByName;
}
```

8.4 topologicallySortMethodCalls

核心代码

```
private List<MethodReference.Handle> topologicallySortMethodCalls() {
    Map<MethodReference.Handle, Set<MethodReference.Handle>> outgoingReferences = new HashMap<>();
    // copy
    for (Map.Entry<MethodReference.Handle, Set<MethodReference.Handle>> entry : methodCalls.entrySet()) {
        MethodReference.Handle method = entry.getKey();
        outgoingReferences.put(method, new HashSet<>(entry.getValue()));
    }
    LOGGER.debug("Performing topological sort...");
    // 深度优先搜索，利用stack回溯
    Set<MethodReference.Handle> dfsStack = new HashSet<>();
    // 已被visit的节点
    Set<MethodReference.Handle> visitedNodes = new HashSet<>();
    // 排序结果
    List<MethodReference.Handle> sortedMethods = new ArrayList<>(outgoingReferences.size());
    for (MethodReference.Handle root : outgoingReferences.keySet()) {
        // 见8.5
        dfsTsort(outgoingReferences, sortedMethods, visitedNodes, dfsStack, root);
    }
    LOGGER.debug(String.format("Outgoing references %d, sortedMethods %d", outgoingReferences.size(),
sortedMethods.size()));
    return sortedMethods;
}
```

8.5 dfsTsort

遍历集合中的起始方法，进行递归深度优先搜索DFS，实现逆拓扑排序。最终结果是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（参考三梦师傅）

stack保证了在进行逆拓扑排序时不会形成环，visitedNodes避免了重复排序（参考Longofo师傅）

深入分析参考8.7

```
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) {
        // 递归
        dfsTsort(outgoingReferences, sortedMethods, visitedNodes, stack, child);
    }
    stack.remove(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
    visitedNodes.add(node);
    // 递归完成的探索，会添加进来
    sortedMethods.add(node);
}
```

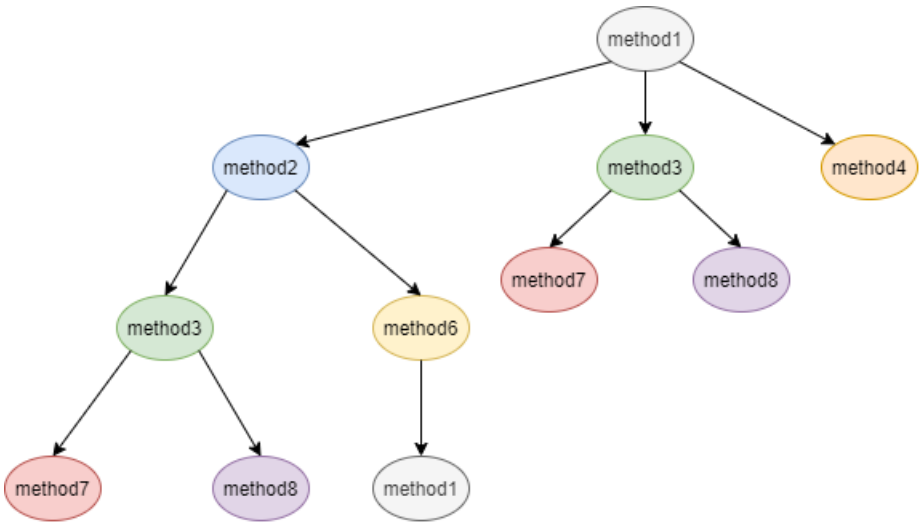
8.6 calculatePassthroughDataflow

```
private static Map<MethodReference.Handle, Set<Integer>> calculatePassthroughDataflow(Map<String,
ClassResourceEnumerator.ClassResource> classResourceByName,Map<ClassReference.Handle, ClassReference>
classMap,InheritanceMap inheritanceMap,List<MethodReference.Handle> sortedMethods,SerializableDecider
serializableDecider) throws IOException {
    final Map<MethodReference.Handle, Set<Integer>> passthroughDataflow = new HashMap<>();
    for (MethodReference.Handle method : sortedMethods) {
        // 跳过static静态代码块
        if (method.getName().equals("<clinit>")) {
            continue;
        }
        // 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 {
                // 参考6和7
                PassthroughDataflowClassVisitor cv = new PassthroughDataflowClassVisitor(classMap,
inheritanceMap,passthroughDataflow, serializableDecider, Opcodes.ASM6, method);
                cr.accept(cv, ClassReader.EXPAND_FRAMES);
                // 缓存方法返回值与哪个参数有关系
                passthroughDataflow.put(method, cv.getReturnTaint());
            } catch (Exception e) {
                LOGGER.error("Exception analyzing " + method.getClassReference().getName(), e);
            }
        } catch (IOException e) {
            LOGGER.error("Unable to analyze " + method.getClassReference().getName(), e);
        }
    }
    return passthroughDataflow;
}
```

8.7 分析

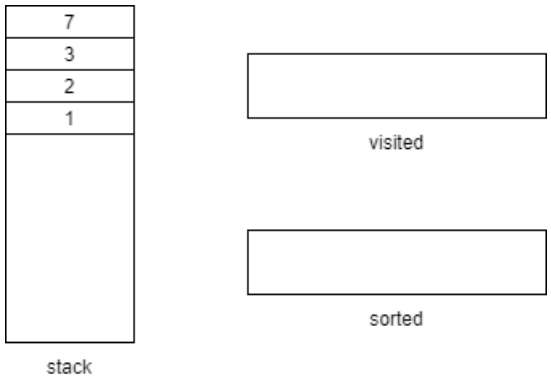
关于逆拓扑排序，参考Longofo师傅的文章，对图片做了一些优化和精简

这是一个方法调用关系：

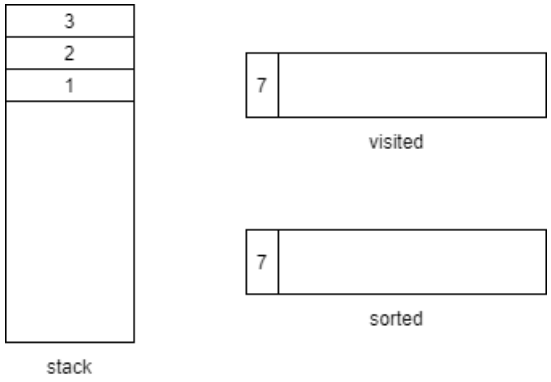


在排序中的stack和visited和sorted过程如下：

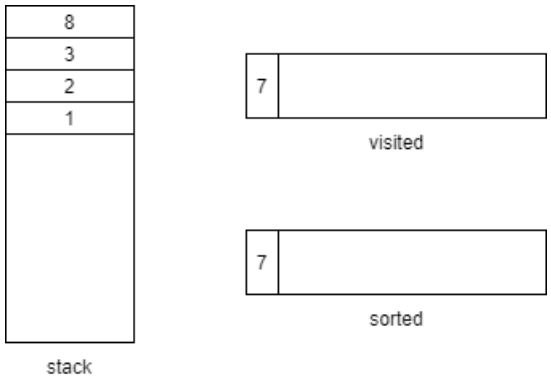
只要有子方法，就一个个地入栈



到达method7发现没有子方法，那么弹出并加入visited和sorted

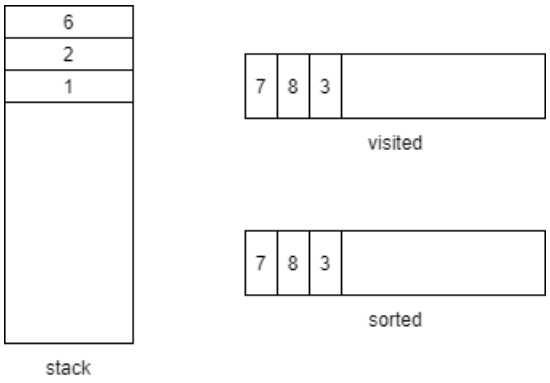


回溯上一层，method3还有一个method8子方法，压栈

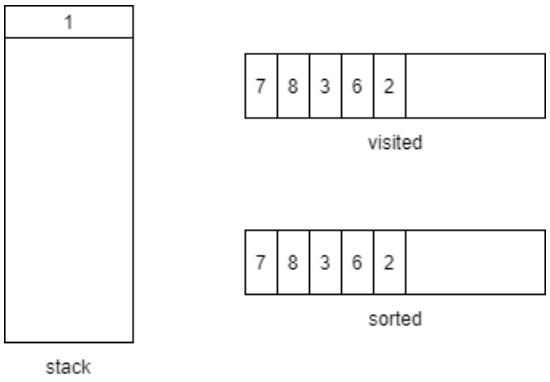


method8没有子方法，回溯上一层method3也没有，都弹出并进入右侧

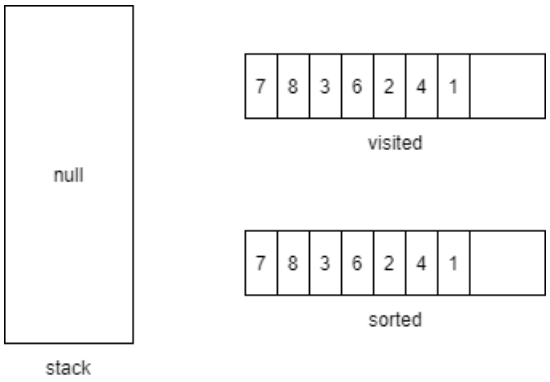
到达method6，有子方法，压栈，找到method6下的method1，压栈，注意这里是Set结构不重复，所以压了等于没压



回溯后method6和method2都没有子方法了，弹出并进入右边



往后执行遇到method1的两个子方法method3和method4，由于method3已在visited，直接return，把method4压栈。然后method4没有子方法弹栈，最后剩下的method1也没有子方法，弹栈



最终得到的排序结果就是7836241，达到了最末端在最开始的效果

9 参考

原版代码：https://github.com/JackOfMostTrades/gadgetinspector

三梦师傅代码：https://github.com/threedr3am/gadgetinspector

Oracle JVM Doc：https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

三梦师傅的文章：https://xz.aliyun.com/t/7058

Longofo师傅的文章：https://paper.seebug.org/1034/

关注 | 2　　点击收藏 | 0

上一篇： Tomcat 内存马（二）Filter型

下一篇： Weblogic漏洞学习：T3反序列化

0 条回复

动动手指，沙发就是你的了！

登录 后跟帖