doi :10.3969/j.issn.1671-1122.2020.07.003

基于字节码搜索的 Java 反序列化漏洞调用链挖掘方法

杜笑宇，叶何，文伟平

（北京大学软件与微电子学院，北京 100080）

摘　要：反序列化漏洞是近年来应用安全研究的热点之一。随着 Java 类库的功能不断更新和扩展，反序列化漏洞的潜在范围更加广泛。手工进行反序列化漏洞挖掘需要大量的时间和人力对调用链进行筛查和构造。文章介绍了 Java 反序列化漏洞的原理、常见场景和反序列化漏洞调用链的构造方法，并结合常见的漏洞挖掘方法，提出一种调用链挖掘方法，同时将该方法实现为调用链挖掘工具 Zero Gadget。文章方法采用污点分析与符号执行技术生成从反序列化漏洞入口点到危险函数的调用树，利用深度优先搜索算法搜索调用树并生成相关调用链。文章选取常见的 Java 基础库进行调用链挖掘效果的测试。实验结果表明，文章方法可成功挖掘潜在调用链并具有较高的准确率，对于反序列化漏洞调用链的自动化挖掘有着积极的意义。

关键词：反序列化漏洞；调用链；Java 漏洞挖掘

中图分类号：TP309　**文献标志码：**A　**文章编号：**1671-1122（2020）07-0019-11

中文引用格式：杜笑宇，叶何，文伟平 . 基于字节码搜索的 Java 反序列化漏洞调用链挖掘方法 [J]. 信息网络安全，2020，20（7）：19-29.

英文引用格式：DU Xiaoyu, YE He, WEN Weiping. Java Deserialization Vulnerability Gadget Chain Discovery Method Based on Bytecode Search[J]. Netinfo Security, 2020, 20(7): 19-29.

Java Deserialization Vulnerability Gadget Chain Discovery Method Based on Bytecode Search

DU Xiaoyu, YE He, WEN Weiping

(*School of Software and Microelectronics*, *Peking University*, *Beijing* 100080, *China*)

Abstract: Deserialization vulnerability is one of the hotspots of application security research in recent years. As the functions of Java class library are constantly updated and expanded, the potential scope of deserialization vulnerability is more extensive. Discovering deserialization vulnerability through manpower requires a lot of time to screen and construct the gadget chain. This paper introduces the Java deserialization vulnerability principle, common scenarios and deserialization vulnerability gadget chain construction method, and combining with the common vulnerability discovery methods, proposes a method to discover gadget chain, which

收稿日期：2020-5-2

基金项目：国家自然科学基金 [61872011]

作者简介：杜笑宇（1996—），女，北京，硕士研究生，主要研究方向为网络安全、Web 攻击与漏洞挖掘；叶何（1998—），男，安徽，硕士研究生，主要研究方向为网络安全、系统安全、云计算；文伟平（1977—），男，湖南，教授，博士，主要研究方向为系统与网络安全、大数据与云安全、智能计算安全。

通信作者：文伟平 weipingwen@ss.pku.edu.cn

is implemented as a gadget chain discovering tool Zero Gadget. The method uses the stain analysis and symbol execution technologies to generate the gadget tree from the deserialization vulnerability entry point to the dangerous function, and uses the depth-first search algorithm to search the gadget tree and generate the relevant gadget chain. This paper selects common Java basic libraries to test the effect of gadget chain discovery. The experimental results show that this method can successfully discover the potential gadget chains and have a high accuracy rate, which has positive significance for automatic discovery of deserialization vulnerability gadget chain.

Key words: deserialization vulnerability; gadget chain; Java vulnerability discovery

0 引言

Java序列化机制是把对象转换为字节序列的过程，可以把内存中的对象保存成文件或其他易于存储和传输的形式。反序列化作为序列化的逆过程，将字节序列转换为对象。二者相结合广泛用于应用程序的开发。随着互联网应用的发展，针对各类应用的攻击呈上升趋势，Java应用程序也暴露出许多漏洞，其中就包含Java反序列化漏洞。

反序列化漏洞在2006年就已经有人提出[1]。2015年，LAWRENCE[2]在Apache的Commons Collections基础库中找到了Java反序列化攻击的执行链，并首次公开了漏洞利用的方法。Commons Collections基础库作为Apache开源项目的重要组件，被广泛用于各种Java应用的开发，包括WebLogic、WebSphere、JBoss和Jenkins等产品。由于该组件覆盖范围广且危险程度较高，使得Java反序列化漏洞得到广泛关注，更多的Java类库被验证存在反序列化漏洞。2017年，MUNOZ[3]对Java和.NET中大量JSON序列化库进行了全面分析，提出许多基于.NET的库（如FastJSON、JavascriptSerializer）和基于Java的库（如Jackson、Genson和FlexSON等）的组件都可能存在反序列化漏洞的问题。反序列化漏洞的挖掘需要寻找相关调用链，如何自动化调用链的挖掘成为越来越多Web安全研究者关注的问题。

1 相关工作

1.1 常见的漏洞挖掘方法

应用程序的漏洞挖掘方法主要分为静态分析和动态分析两种[4]。静态分析主要是对应用程序进行静态扫描，通过词法分析和语法分析生成抽象语法树（Abstract Syntax Tree，AST），进而生成控制流图（Control Flow Graph，CFG）和数据流图（Data Flow Graph，DFG）等，以检验应用程序是否满足安全性要求。动态分析则是结合插桩技术进行动态污点分析，实时检测污点数据的传播。二者各有利弊，近些年静态分析技术向模拟执行技术（如符号执行、抽象解释等）发展，发现了更多传统意义上动态分析才能发现的问题，更好地弥补了静态分析的劣势。

1）符号执行

符号执行是KING[5]于1976年提出的一种程序分析技术，用于在逻辑上推理程序的正确性。符号执行以符号值作为输入来执行程序，并维持一个符号状态σ和路径约束π。假设程序的功能是根据用户的输入来显示相应结果，如代码1所示，通过符号执行技术解析代码1则得到如图1所示的符号执行树。其中，A、B、C等代表不同的符号执行阶段；符号状态σ表示一个变量到符号表达式的映射[6]，最开始存储了用户输入的变量a的映射，后面存储了程序中的x、y等变量的映射。路径约束π初始化时设为“true”，遇到条件语句会进行更新。

代码1 符号执行示例

```
public class Test {
    interface Choice {
        public void show(int num);
    }
    public static class Wrong implements Choice{
        @Override
        public void show(int num){
            System.out.println("Result is "+num);
        }
    }
    public static class Right implements Choice{
```

```
    @Override
    public void show(int num){
      try{
        Runtime.getRuntime().exec("calc");
      }catch (IOException e){
          e.printStackTrace();
      }
      System.out.println("You get the flag"+num);
    }
  }
  public static void main(String[] args) throws IOException {
    Scanner input=new Scanner(System.in);
    int a=input.nextInt();
    int x=0,y=0;
    if(a!=0){
      y=x+a;
      if(a==2){
        Choice rightchoice = new Right();
        rightchoice.show(y);
      }
      else {
        x=2*y;
        Choice wrongchoice = new Wrong();
        wrongchoice.show(x);
      }
    }
    else{
      System.out.println("You did nothing");
    }
  }
}
```

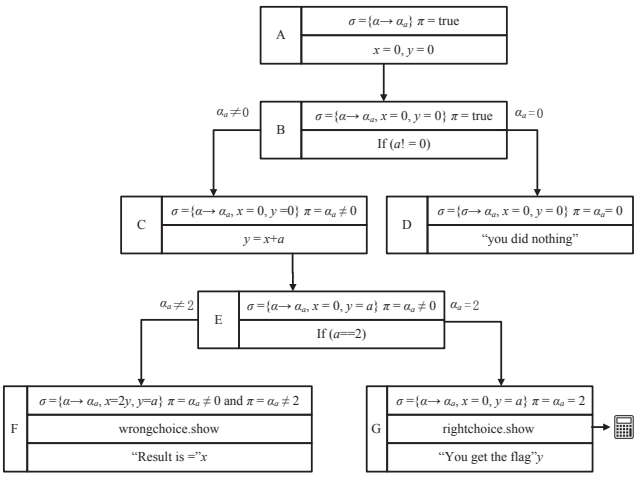

图 1 符号执行树

假设图 1 的目标是确定在何种条件下，程序的执行结果为弹出计算器，这也是反序列化漏洞攻击中最简单的测试目标。通过图 1 可以得出，在 α_a=2 的条件下，该假设成立，即用户在传入数值时将 a 赋值为 2 时，假设成立。这是符号执行树进行逻辑判断较为简单的例子。如果表达式较为复杂，在路径选择时可使用求解器对路径约束表达式进行求解，较常用的有 SMT（Satisfiability Module Theories，可满足性模理论）求解器。

除了上述静态符号执行，许多应用场景还会采用动态符号执行技术。以图 1 为例，假设以“a=1”作为输入，那么具体执行路径即为图 1 中的“A→B→C→E→F”，发现未成功弹出计算器。因此，把最后一个路径约束 π 取反，即把 α_a≠2 取反，进而得到新的输入“a=2”和新的路径“A→B→C→E→G”[7]。动态符号执行中有一个关键技术——路径选择，即在收集了路径约束后，采用一定路径约束策略对其中的约束进行取反。该策略主要有随机路径选择和覆盖率优化两种。前者保证了路径选择的公平性，但可能存在路径爆炸的问题；后者通过对路径中的节点添加权重值来实现，但可能存在路径遗漏。还有一种动态符号执行模式为选择性符号执行，一般用于只运行程序特定部分时，此方法在运行程序特定部分时采用多路径模式，而执行其他代码时采用单路径模式。

2）污点分析

污点分析技术通过标记系统中的敏感函数并对其数据在程序中的传播进行追踪来检测系统的安全性，广泛用于程序的安全性分析。XIE[8] 等人使用敏感数据流分析及中间过程处理等方法对污点数据流向进行标记。ANDROMEDA 工具[9] 采用别名分析降低静态污点分析的开销，利用对象敏感别名分析方法解决分析对象的访问路径问题。CHEN[10] 等人提出一种基于控制依赖的动态污点分析方法来追踪敏感数据流。

污点分析可以抽象成一个三元组<scources，sinks，sanitizers>。其中，source 为污染源，即不受信任的数据；sink 为污点汇聚点，代表可能造成隐私泄露或权限控

制等问题的敏感操作；sanitizer为无害化处理，一般指阻碍数据传播或移除危险的数据处理操作。污点分析过程也可以分为3个阶段：污染源识别、污点传播分析和无害化处理[11]。

以代码1为例，变量*a*由用户传入程序且没有进行任何无害化处理，被定义为污染源。该污点数据通过*y=x+a*传递给变量*y*，最终进入Right类中的函数show()，触发了命令执行。采用污点分析技术可以从源头进行识别并跟踪整条传播链，对于反序列化漏洞调用链的挖掘有着重要意义。

污点分析可分为静态污点分析和动态污点分析。静态污点分析是在不改变且不运行源代码的前提下，采用控制流图和数据流图等静态分析方法，分析程序中的污点数据能否传播到污点汇聚点。动态污点分析是在程序运行过程中，实时监控程序中的污点数据的传播。对污点数据打标签并保存在存储单元，这其中主要采用了插桩技术。插桩技术是在保证目标程序原有逻辑完整的情况下，对源程序或二进制代码的特定位置植入探针，从而获取程序运行信息、执行轨迹、状态数据等信息的一项技术。Java中的插桩主要通过Java中的基础包java.lang.instrument以及字节码操作工具（ASM、Javasist或Byte Buddy等）实现。污点分析技术的应用可能存在过污染或欠污染的问题。其中，过污染指在污染分析过程中将与污染源没有依赖关系的数据变量标记为污点变量，从而产生误报；欠污染则是未能将污点变量进行标记，产生了漏报。

1.2 Java 漏洞挖掘研究

许多学者从事Web应用漏洞的研究。JOVANOVIC[12]等人通过静态源代码分析中的数据流分析来发现Web程序中的脆弱点；LIVSHITS[13]等人提出一种基于路径扫描的静态分析方法，识别从输入点到不安全的使用点的传播路径中是否存在对污染源的验证来进行漏洞挖掘；ALHUZALI[14]等人提出动静结合的Web漏洞挖掘工具NAVEX，用静态分析来定位关键敏感函数并记录，对关键变量进行动态回溯来定位漏洞；BALZAROTTI[15]等人开发出一款名为Sanner的工具，结合静态和动态分析来验证Web应用中的无害化处理过程是否存在缺陷，进而挖掘相关漏洞；FELMETSGER[16]等人采用动态污点分析跟踪常规操作，并采用符号执行识别可能违反规范的程序路径，以寻找Web应用中的逻辑漏洞。然而上述这些Web应用漏洞挖掘方法在反序列化漏洞挖掘方面不是很出色。

很多反序列化研究都针对反序列化的防御和攻击。2015年，在Apache Commons Collections中找到了Java反序列化漏洞攻击的调用链[2]；2016年CARETTONI[17]提出了如何应对Java反序列化的方法，并开发了相关工具SerialKiller。还有研究人员针对Java反序列化漏洞集成常见的一些调用链开发了soserial、marshalsec、Java Deserialization Scanner、NCC Group Burp Plugin等反序列化漏洞攻击工具。这些攻击工具无法自动挖掘反序列化漏洞的潜在调用链，但是对于漏洞调用链的挖掘有一定的借鉴意义。有研究人员根据攻击工具的思路进行挖掘工具的开发，但是对于大量第三方库中可能存在的其他调用链，无法进行识别。

2018年，HAKEN[18]提出一种反序列化漏洞调用链的自动挖掘工具Gadget Inspector。该工具基于字节码开发，对编译完成的Java文件进行检查，生成从反序列化入口到危险函数（即污点分析中从污染源到污点汇聚点）的调用链，是反序列化漏洞挖掘领域较为出色的工具。但是Gadget Inspector也有许多不足之处，如调用链搜索结果不够完整、存在一定漏报和误报等。

2 Java 反序列化漏洞

2.1 漏洞原理

Java序列化由JDK内置的ObjectOutputStream类的writeObject方法实现，相应的Java反序列化由ObjectInputStream类的readObject方法实现，是一种原生过程，即JDK自带的功能。Java序列化后的数据格式有显著特征，它以一个双字节的魔法数字0xACED开始，后面跟着一个双字节的版本号，如0x0005。在4字节头部之后是一个或多个内容元素，每个内容元素的第一个字节在0x70~0x7E

之间，以表示内容元素的类型，如 0x70 代表 NULL、0x74 代表 STRING 等。序列化与反序列化过程如代码 2 所示。

代码 2 Java 序列化与反序列化示例

```
public class Serialize {
    public static void main(String args[])throws Exception{
        DemoObject d1=new DemoObject();
        d1.name="hello world!";
        FileOutputStream fos=new FileOutputStream("object");
        ObjectOutputStream os=new ObjectOutputStream(fos);
        os.writeObject(d1);
        os.close();
        FileInputStream fis=new FileInputStream("object");
        ObjectInputStream ois=new ObjectInputStream(fis);
        DemoObject d2=(DemoObject)ois.readObject();
        System.out.print(d2);
        ois.close();
    }
}
class DemoObject implements Serializable {
    public String name;
    private void readObject(java.io.ObjectInputStream in) throws
IOException,ClassNotFoundException{
        in.defaultReadObject();
        Runtime.getRuntime().exec("calc");
    }
}
```

想要成功对类进行序列化，需要满足两个条件：一是该类必须实现 java.io.Serializable 接口或 java.io.Externalizable 接口；二是类中的所有字段是可序列化的，如果字段标记为 transient，则不可被序列化。另外，反序列化操作会自动调用 ObjectInputStream 类的 readObject 方法，如果该方法被重写，则优先调用重写后的方法。

代码 2 创建了名为“DemoObject”的类并实现了 Serializable 接口，同时重写了 readObject 方法。主程序首先通过 ObjectOutputStream 类的 writeObject 方法将代码 2 序列化，并写入名为“object”的文件，然后通过 ObjectInputStream 类的 readObject 方法读取 object 文件中的序列化数据进行反序列化。因为会优先调用 readObject 方法的重写方法，如果攻击者在其中写入了恶意代码，那么该恶意代码会得到执行[19]。以代码 2 为例，恶意代码会使程序弹出计算器。反序列化漏洞之所以较为严重，是因为 readObject 之类的魔法方法会在反序列化时自动执行，攻击者会构造恶意输入而产生非预期行为，其结果往往会使攻击者获取到服务器的操作权限。

2.2 常见场景

除了 Java 反序列化的原生过程可能存在漏洞，反序列化漏洞的常见场景还包括以下 3 种：解析序列化数据的第三方库、应用序列化机制的协议以及反序列化类名校验不够完善的机制。对于第 1 种场景，首先要知道哪些格式的数据可以被序列化。可被序列化的输入数据格式除了能实现指定接口的 Java 对象，还包括 JSON、XML 和 Yaml 等格式。用于解析 JSON 的主流第三方库包括 Jackson、Fastjson、Genson、FlexSON 等，用于解析 XML 的主流第三方库包括 XStream、XMLDecoder 等。这些解析库会被用于中间件或服务的开发中，可能存在反序列化漏洞。

常见的应用序列化机制的协议包括 RMI（Remote Method Invocation）、JMX（Java Management Extension）、JMS（Java Messaging System）、Spring Service Invokers、AMF（Action Message Format）、Weblogic T3、LDAP 和 CORBA 等。以 RMI 协议为例，它使用反序列化机制来传输 Remote 对象，如果该对象是恶意的，服务端进行反序列化时就会触发反序列化漏洞，此时如果服务端存在 Apache Commons Collections 这种可以构造调用链的库，就会导致远程命令执行。

近年来被爆出存在反序列化漏洞的部分解析库或组件如表 1 所示。

由于 Java 反序列化漏洞存在较为广泛且危害较高，近些年针对反序列化出现了很多防御方案。Java 反序列化的原生过程会自动调用 readObject 方法，该方法又会调用 resolveClass 方法来反序列化类名，因此很多防御手段是在 resolveClass 方法中加入对被反序列化的类进行校验。例如，检验类是否存在于程序的黑名单中，如果解析得到的类名存在于黑名单中，就抛出异常，同时程序终止后续反序列化处理流程[20]。但是如果黑名单不够完整，即能够找到不在黑名单中的可造成恶意

表 1 部分解析库 / 组件及其相关漏洞

库名称	类别	相关漏洞
Jackson	JSON	CVE-2017-7525、CVE-2017-15095、CVE-2019-12384
SnakeYAML	YAML	CVE-2016-9606、CVE-2017-3159、CVE-2016-8744
BlazeDS	AMF4	CVE-2017-3066、CVE-2017-5641
Red5 IO AMF	AMF	CVE-2017-5878
XMLDecoder	XML	CVE-2017-3506、CVE-2017-10352、CVE-2019-2725
Hessian	binary/XML	CVE-2019-9212、CVE-2017-12633
XStream	XML/various	CVE-2017-7947、CVE-2017-9805、CVE-2019-10173
Weblogic	Middleware	CVE-2015-4852、CVE-2016-0638、CVE-2016-3510
JBoss	Middleware	CVE-2017-7504 、CVE-2017-12149
ActiveMQ	Middleware	CVE-2015-5254
Jenkins	Application	CVE-2015-8103、CVE-2016-0792、CVE-2016-9299
Apache Log4j	Components	CVE-2017-5645、CVE-2017-17571
……	……	……

结果的类进行反序列化，就会造成反序列化漏洞，即第3种反序列化漏洞的常见场景。

无论是反序列化的原生过程还是上述3种反序列化漏洞的常见场景，都可以具体到某个版本，如Commons-Fileupload 1.3.1、Commons-Collections 3.1、Commons-Logging 1.2等。它们通常拥有一些像ObjectInputStream.readObject一样的方法，如ObjectInputStream.readUnshared、XMlDecoder.readObject、Yaml.load、XStream.fromXML、ObjectMapper.readValue、JSON.parseObject等，在反序列化过程中被自动调用。这些方法也被称为反序列化漏洞的入口点。如果从入口点通过构造攻击语句可以最终运行到危险函数，就会形成一条反序列化漏洞调用链。

2.3 反序列化漏洞调用链

对Apache Commons Collections基础库中的反序列化漏洞的利用实现了远程命令执行攻击，从而使该漏洞得到极大关注。该漏洞利用的方法主要依赖于Java的反射机制，并利用Apache Commons Collections基础库构造了形如Runtime.getRuntime().exec("calc")的命令执行语句。

在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性。这种动态获取信息以及动态调用对象的方法的功能称为Java语言的反射机制[21]。利用反射机制，可以利用Class.forName方法或通过调用某个对象的getClass方法来获取该对象。此外，利用反射机制还可以获取某个对象的相关方法，主要有getDeclaredMethods、getMethods、getMethod等方法。得到类中的方法后就可以利用函数invoke()进行方法调用。有了反射机制，也即有了动态调用对象的方法，为构造反序列化漏洞的调用链提供了条件。

Java漏洞利用中常用Runtime类的exec方法执行系统调用，而Java为了实现单例模式，将Runtime类的构造函数使用关键词private进行修饰，因此无法直接通过关键词new进行实例创建。这时就需要用到反射机制，先通过getRuntime方法来获取Runtime对象，再通过该对象获取相应方法并利用函数invoke()进行方法调用。

Apache Commons Collections基础库实现了一个名为“Transformer”的接口类，该接口有许多实现类，利用其中的3个类，即ConstantTransformer、InvokerTransformer和ChainedTransformer，可以完成攻击语句Runtime.getRuntime().exec("calc")的构造。把Runtime对象作为参数传入ConstantTransformer中，最终返回该对象；把待调用的方法名和相应参数传入InvokerTransformer中，把ConstantTransformer和InvokerTransformer组装成一个数组全部放进ChainedTransformer中。在执行transform方法时，将前一个元素的返回结果作为下一个元素的参数，即可以完成整条调用链的执行。最后只需要考虑如何触发transform方法。

Apache Commons Collections基础库的反序列化漏洞完整调用链如图2所示。

从入口点（如ObjectInputStream.readObject）到危险函数（如Runtime.exec）的路径构造就是反序列化漏洞调用链的构造方法。如果可以构造出从入口点到危险函数的调用链，那么就会造成严重的危害。

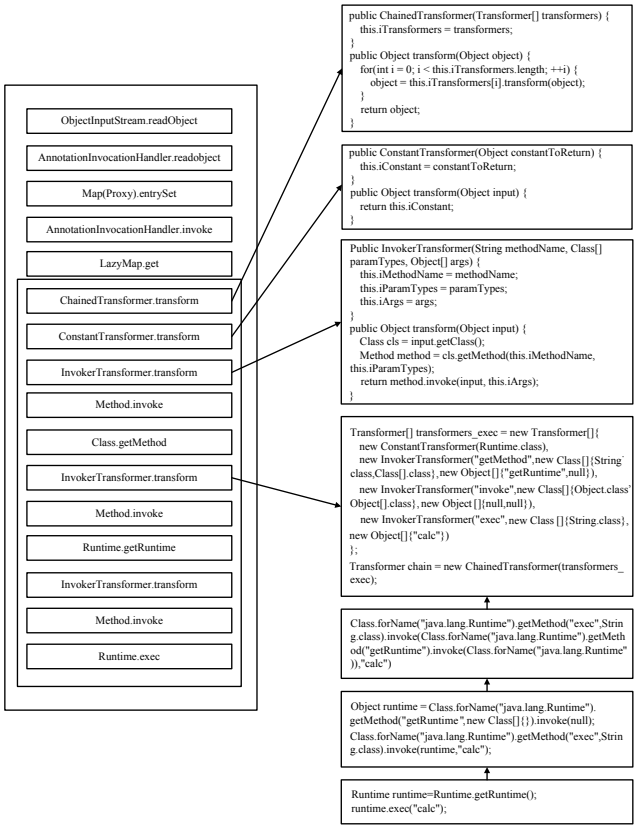

图 2 Apache Commons Collections 调用链

3 调用链挖掘方法设计

本文所开发的基于字节码搜索的Java反序列化漏洞调用链挖掘工具命名为Zero Gadget，该工具对调用链的挖掘主要分为3大步骤：静态扫描与分析、端点搜索与可达性分析以及调用链搜索。

1）静态扫描与分析

静态扫描与分析过程如图3所示。首先通过字节码搜索，对程序进行扫描以获取全部可用的类、方法、父与子关系等信息；然后采用静态分析中的数据流、控制流和函数调用图分析方法，分别获取方法参数与返回值、方法参数与条件选择、方法参数与子方法参数等的关系。在静态分析过程中引入污点分析，通过标记并追踪JVM中的数据，判断上述关系是否存在。

如果方法参数与返回值无关，表明返回结果不可控，调用链可能无法继续构造，那么后续步骤生成调用链时，该路径可能到达了尽头。Java对象多态性的

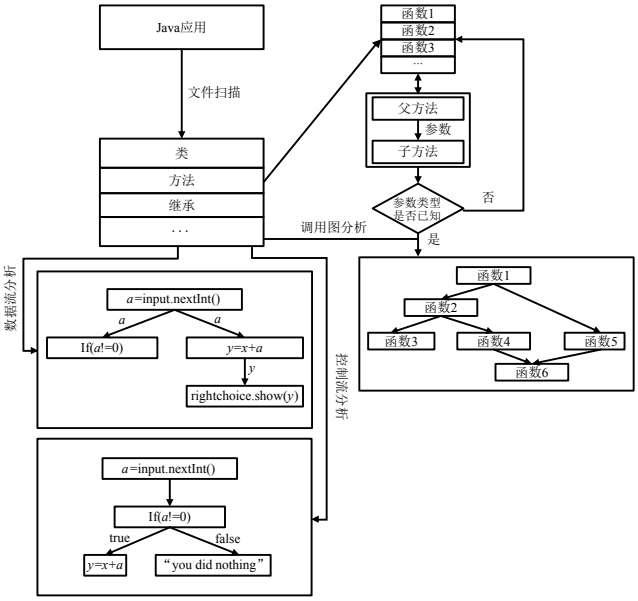

图 3 静态扫描与分析

特点导致调用接口方法时无法确定其具体实现，只有在运行时才能确定具体调用哪个子类的方法，此时污点追踪就会遇到问题，因此要记录方法参数与条件选择的关系。判断方法参数与条件选择的关系依赖于参数类型是否已知，如果参数类型未知，则对该接口所有可能的实现方法进行扫描并记录，依次判断是否有污点传递的可能。函数调用图分析过程中还要判断方法参数是否影响其子方法参数，如果子方法参数不受其影响，调用链也可能就此终结。

数据流分析、控制流分析和函数调用图分析都有其特点，数据流分析体现了方法参数与方法返回值的关系，控制流分析中每个方法的每个分支都是可达的，函数调用关系图表达了方法参数与调用的子方法参数的关系。

2）端点搜索与可达性分析

端点搜索与可达性分析过程如图4所示。想要确定一条路径，就要知道起点和终点。反序列化漏洞调用链的起点就是一些原生反序列化过程的入口点，如Object.readObject、Object.readResolve、Object.finalize 等，或是一些实现了上述入口点且自动调用了其他方法的可序列化JDK类，如HashMap类中的Object.hashCode、

Object.equals等方法，PriorityQueue类中的Comparator.compare、Comparable.CompareTo等方法。除了上述这些原生的反序列化入口点，反序列化漏洞的常见场景也可以作为反序列化的入口点。反序列化漏洞调用链的终点就是危险函数，一般包含Runtime.exec、Method.invoke等。Runtime.exec方法最为简单直接，即直接在目标环境中执行命令；Method.invoke方法需要适当选择方法和参数。此外，也可以通过应用序列化机制的协议，如RMI、JNDI、JRMP等，通过引用远程对象，间接实现任意代码执行的效果。

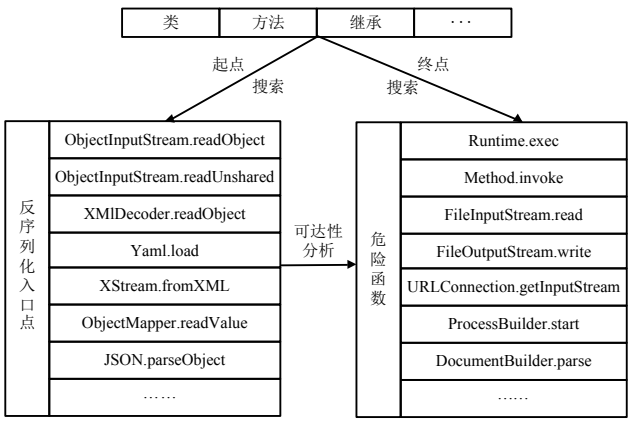

图 4 端点搜索与可达性分析

从静态扫描与分析的结果中查找常见反序列化入口点，并判断方法所属的类是否可以被反序列化；同时查找常见的危险函数来确定调用链的尾部端点。此外，为了提升挖掘的命中率，本文将较为隐蔽的入口点，如HashSet、Hashtable等类，也收录到入口点搜索集合中。

从静态分析的角度讲，首尾端点能否成功连接是一个可达性分析问题，分析从反序列化入口点到危险函数是否存在如图1中“A→B→C→E→G”这样的路径。采用静态污点分析跟踪的方式，根据步骤1）中各类分析手段的分析结果生成调用树，结合符号执行技术进行路径选择。

3）调用链搜索

调用链搜索过程如图5所示。由步骤2）得到的每个反序列化入口点开始，搜索可达性分析中生成的调用树。一般树的搜索主要有两种算法，一种是深度优先搜索（DFS），另一种是广度优先搜索（BFS）。两种算法各有利弊，且路径搜索过程中有一些问题需要考虑。例如，有些入口点可以到达危险函数，但是危险函数中执行命令的参数无法控制，即危险函数无法被实际利用，那么该调用链不是有效利用链。面对此问题就需要污点分析技术来跟踪最终的危险函数是否可用。

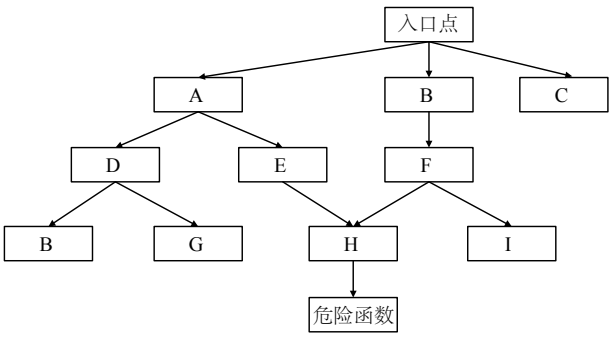

图 5 调用链搜索

如果采用如Gadget Inspector工具那样的广度优先算法，则从反序列化入口点开始，将入口点加入堆栈，创建相应的存储空间记录已经访问过的节点和被选中的方法。依次读入入口点的子方法A、A的子方法D、D的子方法B，直到B没有子方法停止。将B从堆栈中取出加入存储空间，然后回溯到上层D，有子方法G加入堆栈。回溯过程中遇到没有子方法，就弹出堆栈元素，加入相应的存储空间，有子方法就加入堆栈，直至最后堆栈为空。广度优先算法防止了遍历方法的调用过程变成死循环，但是这样的搜索也存在问题。图5所示的搜索中，如果先走通了“入口点→A→E→H→危险函数”这条调用链，那么当再遍历“入口点→B→F→H”时，由于节点H之前已经被访问过，所以直接被跳过，因此可能导致调用链漏报的现象。例如，假设方法H中有一定的条件限制，而“A→H”的过程恰好不符合该条件，那么“入口点→A→E→H→危险函数”并不是一条有效的调用链，即实际上“A→H”的调用链是无法触发的，而“入口点→B→F→H→危险函数”却因为节点H已被访问未被记录，那么就会造成漏报。

采用深度优先搜索算法可以较好地解决此类漏报

问题。在本文挖掘工具Zero Gadget的设计中，调用树的搜索方法采用的就是深度优先搜索算法。为防止出现无限循环，算法中设置了最大深度，并增设记录来探寻从分支节点到最大深度的链条。算法以入口点、方法间调用关系、危险函数等为输入，调用链为输出。从每个入口点开始，根据方法间调用关系，对下一个调用方法进行深度优先遍历。如果达到最大深度，该条链路结束；否则，继续遍历。如果遇到危险函数，就将此链路加入调用链集合。重复这些步骤，直到所有的入口点被遍历完毕。

4 实验及分析

本文选用一台使用2.5GHz主频英特尔4核i7处理器、内存为16GB的主机进行实验，系统为64位Mac OS 10.15.4。选取了4个常见的存在反序列化漏洞的Java基础库：Commons-collections-3.2.1、Commons-Collections-4.0、Weblogic Coherence和Groovy-2.3.9，将工具Zero Gadget和Gadget Inspector对于这4种常见的Java库的反序列化漏洞调用链的挖掘情况进行对比，并且对已挖掘的调用链进行人工分析，检查调用链是否为反序列化漏洞有效的利用方式。

1）命中率对比

对两个工具挖掘到的有效调用链数量在挖掘到的总调用链数中的占比即命中率进行计算，如表2所示。通过表2可以分析得出，Zero Gadget对于反序列化漏洞调用链的挖掘命中率约为38%，效果明显优于Gadget Inspector。

表 2 反序列化漏洞调用链命中率对比

库名 / 工具（平均命中率）	Commons-collections-3.2.1	Commons-Collections 4.0	Weblogic Coherence	Groovy-2.3.9
Gadget Inspector(38%)	25%	0%	0%	0%
Zero Gadget(6.67%)	57%	50%	20%	20%

下面给出调用链的具体分析。首先以Commons-collections-3.2.1为例，Gadget Inspector找到4条调用链，如表3所示，其中关键点表示4条调用链所经过的不同关键节点。

表 3 Gadget Inspector 对于 Commons–collections–3.2.1 的搜索结果

序号	入口点	关键点
1	org/apache/log4j/spi/LoggingEvent.readObject	org/apache/log4j/spi/LoggingEvent.readLevel
2	org/apache/log4j/pattern/LogEvent.readObject	org/apache/log4j/pattern/LogEvent.readLevel
3	java/awt/Component.readObject	org/apache/commons/collections/map/DefaultedMap.get
4	com/sun/corba/se/spi/orbutil/proxy/CompositeInvocationHandlerImpl.invoke	org/apache/commons/collections/map/DefaultedMap.get

对Gadget Inspector发现的这4条调用链以人工筛选的方式进行检查，发现这4条调用链并非都是有效调用链。而Zero Gadget在Gadget Inspector的基础上额外发现了如表4所示的3条调用链。

表 4 Zero Gadget 对于 Commons–collections–3.2.1 发现的额外调用链

序号	入口点	关键点
1	sun/reflect/annotation/AnnotationInvocationHandler.readObject	org/apache/commons/collections/map/LazyMap.get
2	java/util/HashSet.readObject	org/apache/commons/collections/map/LazyMap.get
3	java/util/Hashtable.readObject	org/apache/commons/collections/functors/InstantiateTransformer

对Zero Gadget发现的这额外3条调用链通过人工方式进行检查，发现它们是公开的调用链，证实了这3条调用链的有效性。

表5给出了Gadget Inspector和Zero Gadget对于Commons-Collections-4.0、Weblogic Coherence和Groovy-2.3.9的部分挖掘结果。由表5可知，Zero Gadget相较Gadget Inspector在Commons-Collections 4.0中额外发现了PriorityQueue和TreeBag两条可利用调用链，而它们共同发现的两条调用链均不可利用。

以PriorityQueue为例进行分析。PriorityQueue的调用流程为ObjectInputStream.readObject ⇒ PriorityQueue.readObject ⇒ TransformingComparator.compare ⇒ InvokerTransformer.transform ⇒ Method.invoke ⇒Runtime.exec。在对PriorityQueue进行反序列化操作时，将会依次调用java.util.PriorityQueue的heapify、siftDown、siftDownUsingComparator等方法。在调用路径执行到org.apache.commons.

表 5 两个工具对于 Commons–Collections–4.0、Weblogic Coherence 和 Groovy–2.3.9 的部分挖掘结果

库名	Gadget Inspector	Zero Gadget
Commons-Collections 4.0	(1)org/apache/log4j/spi/LoggingEvent.readObject (2)org/apache/log4j/pattern/LogEvent.readObject	(1)java/util/PriorityQueue.readObject (2)org/apache/log4j/spi/LoggingEvent.readObject (3)org/apache/log4j/pattern/LogEvent.readObject (4)org/apache/commons/collections4/bag/TreeBag.readObject
Weblogic Coherence	(1)com/tangosol/coherence/component/util/Daemon.finalize (2)org/apache/log4j/spi/LoggingEvent.readObject (3)java/security/cert/CertificateRevokedException.readObject (4)org/apache/log4j/pattern/LogEvent.readObject (5)com/tangosol/coherence/component/util/Daemon.finalize	(1)javax/management/BadAttributeValueExpException.readObject (2)com/tangosol/coherence/Component (3)com/tangosol/coherence/component/application/console/Coherence$CacheItem.readObject (4)org/apache/log4j/pattern/LogEvent.readObject (5)org/apache/log4j/spi/LoggingEvent.readObject
Groovy-2.3.9	(1)org/apache/log4j/pattern/LogEvent.readObject (2)org/apache/log4j/spi/LoggingEvent.readObject (3)org/codehaus/groovy/transform/tailrec/TailRecursiveASTTransformation$_replaceRecursiveReturnsInsideClosures_closure9.doCall (4)org/codehaus/groovy/runtime/MethodClosure.doCall	(1)java/util/PriorityQueue.readObject (2)org/apache/log4j/pattern/LogEvent.readObject (3)org/apache/log4j/spi/LoggingEvent.readObject (4)org/codehaus/groovy/runtime/MethodClosure.doCall (5)org/codehaus/groovy/transform/tailrec/TailRecursiveASTTransformation$_replaceRecursiveReturnsInsideClosures_closure9.doCall

collections4.comparators.TransformingComparator 类的 compare 方法时，会调用org.apache.commons.collections4.Transformer 类的子类对象，从而有机会执行到 org.apache.commons. collections4. functors.InvokerTransformer 类的 transform 方法进行反射调用，最终导致任意类反射调用执行。

Zero Gadget在Weblogic Coherence中发现了BadAttributeValueExpException 所在的调用链，该调用链是CVE-2020-2555 的攻击方式；在groovy-2.3.9 中发现了PriorityQueue 调用链。

此外，值得注意的是，在测试Commons-Collections 类库时，除了发现了 2.3 节中提到的可利用的 InvokerTransformer 方法，还发现了可替代方法InstantiateTransformer，经过人工检查发现该函数同样可以加以利用。对实验结果进一步研究发现，Gadget Inspector 没有发现 Instantiate Transformer 这种利用方式，原因除了该工具在搜索算法方面的不足之外，还有就是没有考虑反射调用在构造器中实现命令执行攻击的情况。

综上可知，Zero Gadget 比 Gadget Inspector 具有更好的挖掘能力。

2）运行时间对比

Zero Gadget 与 Gadget Inspector 的运行时间对比如图 6 所示。由图 6 可知，Zero Gadget 对于不同 Java 基础库的测试时间较为稳定，平均约为 37 s，具有较高的扫描与挖掘效率。

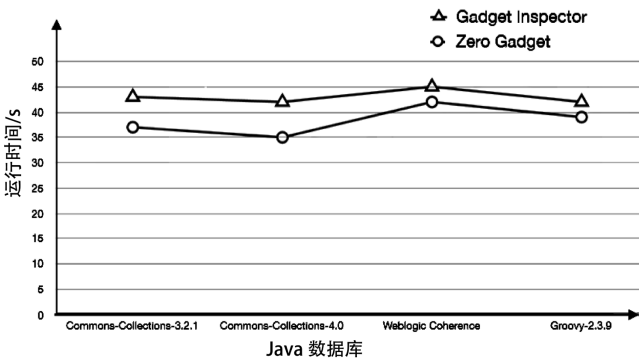

图 6 Zero Gadget 与 Gadget Inspector 运行时间对比

5 结束语

Java 反序列化漏洞是近些年出现较为频繁的高危漏洞。本文采用静态分析方法，并结合符号执行和污点分析等技术，提出一种基于字节码搜索的 Java 反序列化漏洞调用链挖掘工具 Zero Gadget。实验表明，Zero Gadget 相较于 Gadget Inspector 具有更准确的调用链挖掘结果，对于 Java 反序列化漏洞调用链挖掘研究具有积极的意义。但 Zero Gadget 在测试中也存在误报，下一步工作将对函数中的参数增加可控性判断，并在最终的危险函数阶段对可控性判断进一步优化。（责编 马珂）

参考文献：
[1] SCHOENEFELD M. Pentesting J2EE[EB/OL]. https://www.researchgate.net/publication/267774292_Pentesting_J2EE, 2020-4-29.
[2] LAWRENCE G. Marshalling Pickles[EB/OL]. http://frohoff.github.io/appseccali-marshalling-pickles/, 2020-4-29.
[3] MUNOZ A. Friday the 13th: JSON Attacks[EB/OL]. https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf, 2020-4-29.
[4] BALZAROTTI D, COVA M, FELMETSGER V, et al. Saner:

Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications[C]//IEEE. 2008 IEEE Symposium on Security and Privacy (sp 2008), May 18–22, 2008, Oakland, CA, USA. NJ: IEEE, 2008: 387–401.

[5] KING J C. Symbolic Execution and Program Testing[J]. Communications of the ACM, 1976, 19(7): 385–394.

[6] SONG Xuejiao. Research and Application of Symbolic Execution in Software Security[D]. Chengdu: Xihua University, 2018.
宋雪勰 . 符号执行在软件安全领域中的研究与应用 [D]. 成都：西华大学，2018.

[7] GODEFROID P, KLARLUND N, SEN K. DART: Directed Automated Random Testing[J]. ACM SIGPLAN Notices, 2005, 40(6): 213–223.

[8] XIE Yichen, AIKEN A. Static Detection of Security Vulnerabilities in Scripting Languages[C]//USENIX. The 15th Conference on USENIX Security Symposium, July 31–August 4, 2006, Vancouver, B. C. , Canada. Berkeley: USENIX Association, 2006: 179–192.

[9] TRIPP O, PISTOIA M, COUSOT P, et al. Andromeda: Accurate and Scalable Security Analysis of Web Applications[M]//Springer. Fundamental Approaches to Software Engineering. Heidelberg: Springer, Berlin, Heidelberg, 2013: 210–225.

[10] CHEN Zhe, WANG Xiaojuan, ZHANG Xinxin. Dynamic Taint Analysis with Control Flow Graph for Vulnerability Analysis[C]//IEEE. 2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control, October 21–23, 2011, Beijing, China. NJ: IEEE, 2011: 228–231.

[11] WANG Lei, LI Feng, LI Lian, et al. Principle and Practice of Taint Analysis[J]. Journal of Software, 2017, 28(4): 860–882.
王蕾，李丰，李炼，等 . 污点分析技术的原理和实践应用 [J]. 软件学报，2017，28（4）：860–882.

[12] JOVANOVIC N, KRUEGEL C, KIRDA E. Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications[J]. Journal of Computer Security, 2010, 18(5): 861–907.

[13] LIVSHITS B, LAM M S. Finding Security Vulnerabilities in Java Applications with Static Analysis[EB/OL]. https://www.researchgate.net/publication/228633652_Finding_security_vulnerabilities_in_Java_applications_with_static_analysis, 2020–4–29.

[14] ALHUZALI A, GJOMEMO R, ESHETE B, et al. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications[C]//USENIX. The 27th USENIX Conference on Security Symposium, August 15–17, 2018, Baltimore, MD, USA. Berkeley: USENIX Association, 2018: 377–392.

[15] BALZAROTTI D, COVA M, FELMETSGER V, et al. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications[C]//IEEE. 2008 IEEE Symposium on Security and Privacy (sp 2008), May 18–22, 2008, Oakland, CA, USA. NJ: IEEE, 2008: 387–401.

[16] FELMETSGER V, CAVEDON L, KRUEGEL C, et al. Toward Automated Detection of Logic Vulnerabilities in Web Applications[EB/OL]. https://www.usenix.org/legacy/events/sec10/tech/full_papers/Felmetsger.pdf, 2020–4–29.

[17] CARETTONI L. Defending against Java Deserialization Vulnerabilities[EB/OL]. https://www.ikkisoft.com/stuff/Defending_against_Java_Deserialization_Vulnerabilities. pdf, 2020–4–29.

[18] HAKEN I. Automated Discovery of Deserialization Gadget Chains[EB/OL]. https://i.blackhat.com/us–18/Thu–August–9/us–18–Haken–Automated–Discovery–of–Deserialization–Gadget–Chains–wp.pdf, 2020–4–29.

[19] GUO Rui. Research of Java Deserialization Vulnerability[J]. Information Security and Technology, 2016, 7(3): 27–30.
郭瑞 . Java 反序列化漏洞研究 [J]. 信息安全与技术，2016，7（3）：27–30.

[20] ERNST P. Look–ahead Java Deserialization[EB/OL]. https://www.ibm.com/developerworks/library/se–lookahead/, 2013–1–15.

[21] ORACLE. The Java Tutorials[EB/OL]. https://docs.oracle.com/javase/tutorial/reflect/index. html, 2020–4–29.