

基于混合分析的 Java 反序列化利用链挖掘方法

武永兴, 陈力波, 姜开达

(上海交通大学网络空间安全学院, 上海 200240)

摘要: Java 反序列化漏洞已经成为当下 Java 应用安全的常见威胁, 其中能够找到反序列化利用链是该类型漏洞能否利用的关键。由于 Java 应用及依赖库的代码空间大和 Java 本身多态的问题, 人工分析 Java 反序列化利用链, 需消耗大量的时间和精力, 且高度依赖分析人员的经验知识。因此, 研究如何高效且准确地自动化挖掘反序列化利用链至关重要。提出了基于混合分析的 Java 反序列化利用链挖掘方法。根据变量声明类型构造调用图, 通过调用图分析筛选可能到达危险函数的反序列化入口函数。将筛选出的入口函数作为混合信息流分析的入口, 开展同时面向指针和污点变量的混合信息流分析, 对隐式创建的对象标记污点, 在传播指针信息的同时传播污点信息, 构建混合信息流图。基于混合信息流图判断外部污点数据传播到危险函数的可达性。根据污点传播路径构造相应的反序列化利用链。混合分析兼顾了调用图分析的速度和混合信息流分析的精度。基于提出的混合分析方法, 实现相应的静态分析工具——GadgetSearch。GadgetSearch 在 Ysoserial、Marshalsec、Jackson 历史 CVE、XStream 历史 CVE4 个数据集上的误报率和漏报率比现有的工具 GadgetInspector 低, 并且发现多条未公开利用链。实验结果证明, 所提方法能够在多个实际 Java 应用中高效且准确地挖掘 Java 反序列化利用链。

关键词: 反序列化漏洞; 指针分析; 污点分析; 混合分析

中图分类号: TP393

文献标志码: A

DOI: 10.11959/j.issn.2096-109x.2022009

Java deserialization gadget chain discovery method based on hybrid analysis

WU Yongxing, CHEN Libo, JIANG Kaida

School of Cyber Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

Abstract: Java deserialization vulnerabilities have become a common threat to Java application security nowadays. Finding out the gadget chain determines whether this type of vulnerability can be exploited. Due to the large code space of Java applications and dependent libraries and the polymorphism of Java itself, manual analysis of Java

收稿日期: 2021-12-06; 修回日期: 2022-01-06

通信作者: 陈力波, bob777@sjtu.edu.cn

基金项目: 国家重点研发计划 (2019QY0703); 上海市科委高新技术领域项目 (20511102002)

Foundation Items: The National Key R&D Program of China (2019QY0703), Science and Technology Commission of Shanghai Municipality Research Program (20511102002)

引用格式: 武永兴, 陈力波, 姜开达. 基于混合分析的 Java 反序列化利用链挖掘方法[J]. 网络与信息安全学报, 2022, 8(2): 160-174.

Citation Format: WU Y X, CHEN L B, JIANG K D. Java deserialization gadget chain discovery method based on hybrid analysis[J]. Chinese Journal of Network and Information Security, 2022, 8(2): 160-174.

deserialization gadget chains consumes a lot of time and effort and it is highly dependent on the experienced knowledge. Therefore, it is crucial to study how to efficiently and accurately automate the discovery of gadget chains. Java deserialization gadget chain discovery method based on hybrid analysis was proposed. Call graph based on the variable declaration type was constructed, and then the deserialization entry functions that may reach the dangerous functions were screened using the call graph analysis. The screened entry functions were used as the entry point of the hybrid information flow analysis. The hybrid information flow analysis was carried out for both pointer and tainted variables. The tainted objects created implicitly were marked. The tainted information and the pointer information were propagated simultaneously to construct the hybrid information flow graph. The reachability of external taint data propagation to the dangerous function was judged based on the hybrid information flow graph. The corresponding deserialization gadget chain was constructed according to the taint propagation path. The hybrid analysis took into account the efficiency of call graph analysis and the accuracy of hybrid information flow analysis. The corresponding static analysis tool, namely GadgetSearch, was implemented based on the proposed hybrid analysis method. In the experimental evaluation, GadgetSearch had lower false positive and lower false negative than the existing tool GadgetInspector on four datasets of Ysoserial, Marshalsec, Jackson historical CVE, and XStream historical CVE. Additionally, GadgetSearch also found multiple undisclosed gadget chains. The experimental results show that the proposed method can efficiently and accurately discover the Java deserialization gadget chain in multiple practical Java applications.

Keywords: deserialization vulnerability, pointer analysis, taint analysis, hybrid analysis

0 引言

2015年, Lawrence和Frohoff首先提出了Java反序列化漏洞。随后, Breenmachine利用Java反序列化漏洞成功攻击WebLogic、WebSphere、JBoss、Jenkins、OpenNMS等Java应用, 并实现了远程代码执行。Fastjson、Jackson、Hessian、XStream等第三方反序列化库被爆出存在反序列化漏洞, 在反序列化攻击者可控的数据时可能会执行攻击者构造的恶意代码。攻击者通常需要通过组合应用中存在的代码片段, 即gadgets, 形成反序列化利用链, 才能挟持反序列化器的执行控制流, 达到执行对应危险操作的目的。因此, 反序列化利用链是Java反序列化漏洞利用的关键, 研究如何自动化挖掘Java反序列化利用链至关重要。

序列化是指将Java对象转换成字节序列、JSON字符串、XML字符串等, 用于存储、网络传输等。反序列化是序列化的逆过程, 将字节序列、JSON字符串、XML字符串等, 转换成Java对象。Java反序列化漏洞常见于RPC调用过程中, 客户端将参数对象序列化成字节码, 通过网络传输给服务端, 服务端将收到的数据反序列化成参数对象, 然后调用相应的处理函数。另一种常见

的场景是对HTTP请求的参数进行反序列化, 将用户传递的JSON/XML格式的数据转换成Java对象。按照反序列化的机制, Java反序列化器可以分为两类, 一类是基于bean, 通过调用setter方法对field进行赋值; 另一类是基于field, 通过反射直接对field进行赋值。常见的基于bean的反序列化器有: Fastjson和Jackson。常见的基于field的反序列化器包括: JND (java naive deserialization)、XStream、Hessian。攻击者根据反序列化器的特点, 在类路径中搜索合适的gadget, 将这些gadget组装成反序列化利用链。通过反序列化利用链可以挟持程序的控制流, 执行危险的代码片段, 达到执行任意代码的目的。由于反序列化是一项正常的功能, 修复时不能简单地关闭反序列化功能。因此, 针对反序列化漏洞, 常见的修复手段是将已知的利用链加到黑名单中。换言之, 如果攻击者能够找到新的利用链, 那么就可以继续利用反序列化漏洞进行攻击。

然而在反序列化漏洞利用链的搜索中面临以下3个方面的困难。

首先, 在反序列化的过程中用户可以控制生成对象的类型和属性, field实际对应的类型可能是声明类型的任意子类, 从而造成所调用的函数

不确定,需要排查所有子类情况。在确定被调函数的时候,可能要排查 N 倍的具体函数(N 为实现方法的数量)。随着调用链长度的增长,要排查的函数数量将呈指数性增长,导致搜索空间爆炸。

其次,Java 多态特性导致调用方法不确定,需要通过指针分析计算其实际运行中对应的类型。但是,指针分析需要消耗大量计算资源。

最后,当前整个 JDK 的代码量已然十分庞大,以 Java (1.8.0_171) 为例,共有 38 466 个类和 32 4595 个方法。庞大的代码数量,给自动化分析带来巨大挑战。

与 Java 反序列化漏洞相关的工具大多是 payload 生成工具,如 Marshalsec 和 Ysoserial,这些工具将已知的利用链封装成 payload,但不具备挖掘未知利用链的能力;同时,有一部分工业界的工具,如 GadgetInspector,提供了自动化搜索 gadget 的能力,但其没有进行指针分析,导致误报率和漏报率都很高。当前学术界大多数的工作是关于如何检测注入类漏洞,对于自动化挖掘 Java 反序列化利用链没有探索过。

综上所述,针对序列化利用链搜索的空间十分大,人工审计难以覆盖的现状,提出一种能够协助安全研究人员自动化挖掘利用链的方法是相当有必要的。针对该问题,本文主要贡献如下。

1) 提出了基于混合分析的 Java 反序列化利用链挖掘方法,设计并实现了在字节码级别进行自动化分析挖掘 Java 反序列化利用链工具——GadgetSearch。

2) 实验表明 GadgetSearch 的误报率和漏报率远小于现有的工具 GadgetInspector,并且利用 GadgetSearch 挖掘出 1 条 Jackson 未公开利用链、4 条 XStream 未公开的 JDK 利用链和 1 条 Hessian 未公开的 JDK 利用链,申请分配了 5 个 CVE (CVE-2021-39141、CVE-2021-39144、CVE-2021-39146、CVE-2021-39153、CVE-2021-43297) 和 1 个 CNVD 编号 (CNVD-2021-44381)。

1 相关工作

指针分析 (pointer analysis)^[1-4]是计算程序变量在运行期间可能指向的对象集合。Java 多态的性质,需要根据变量运行时指向对象的类型解

析被调函数,所以指针分析是其他分析的基础。Bravenboer^[5]等提出了一种声明式指针分析框架 Doop,他们通过在 Datalog 中描述完整的端到端分析,并使用专门针对递归的 Datalog 程序进行优化,推动了声明式指针分析的发展。Antoniadis^[6]等将 Doop 中的 Datalog 推理引擎从 LogicBlox 换成了 Souffle, Souffle 支持通过共享内存并行分析,切换推理引擎之后,Doop 整体提速 4 倍。

污点分析 (taint flow analysis)^[7-11]是信息流分析 (information flow analysis) 的一种。Livshits^[12]等探索使用 Datalog 进行污点分析,但没有将该污点分析与指针分析统一起来。Tripp^[13]等认为污点分析是一个需求驱动问题,不需要进行完整的全程序分析,他们提出了一套名为 Andromeda 的工具,利用数据流方程和污点集进行数据流分析。FlowDroid^[14]是一种 Android 应用污点分析框架,可以进行流敏感 (flow-sensitive) 的污点分析。FlowDroid 对常见的 Android 特性 (如回调等) 进行了建模,并且手工筛选出了 700 多条 Source、Sink 以及转移函数。这些转移函数包含了常见数据结构的操作。但是 FlowDroid 没有将信息流分析和调用图构造统一起来。Pidgin^[15]通过程序依赖图 (program dependence graph) 的方式实现了信息流分析。程序依赖图可以被用来模拟任何形式的数据以及指令之间的控制依赖。DroidInfer^[16]使用约束流图的可达性算法进行污点分析,它依靠 WALA8 在进行主要分析之前产生一个控制流图。

Grech 等提出了 P/Taint^[17],将指针分析和污点分析结合起来,他们通过很小的改动把信息流分析整合到了 Doop 指针分析框架。通过扩充堆对象的定义域来实现这一目标,并增加污点分析的内容。程序分析通常将污点分析和指针分析视为不同的过程,但该项工作表明,同样的算法可以同时计算污点信息和指针信息这两种相互关联但独立的信息。同时,由于反序列化漏洞的污点源不是来自 Source 函数,而是来自隐式创建的对象,无法直接利用 Doop 的信息流分析功能来跟踪反序列化的污点数据流。

Dahse^[18]等介绍了 PHP 反序列化漏洞,他们提出了一个过程间字段敏感和对象敏感的数据流

分析方法,利用该方法自动化挖掘 PHP 反序列化利用链。Shcherbakov^[19]等研究了.NET 中的反序列化漏洞。他们先将.NET 字节码转换成.NET 通用的中间语言;然后实现了一个高效、可扩展的过程间污点分析;最后基于该方法实现了自动化挖掘.NET 利用链工具 SerialDetector。由于他们的方法只针对 PHP 和.NET 这两种特定语言,无法直接应用到 Java 反序列化利用链的挖掘。

JackOfMostTrades 在 BlackHat 上介绍了一款挖掘利用链的工具 GadgetInspector^[20]。GadgetInspector 利用 ASM 库解析字节码,通过模拟操作栈实现污点分析,支持 JND、XStream, Jackson 这 3 种反序列化利用链的挖掘。杜笑宇等^[21]将 GadgetInspector 中的广度优先搜索算法改造成了深度优先算法,找到了一些新的利用链。Threedr3am 对 GadgetInspector 进行了改造,增加了对 Fastjson 和 Hessian 的支持。但是 GadgetInspector 构造调用图时没有进行指针分析,导致调用图不准确,包含大量不存在的调用关系,误报率和漏报率非常高。

2 研究动机

通过分析 Java 反序列化器的实现可知,Java 反序列化漏洞能被利用的原因包含以下 3 点。反序列化的数据用户可控,用户能够控制对象的类型及其属性值;反序列化器在反序列化的时候会自动调用入口函数;类路径(包括 JDK 和第三方库)中存在合适的 gadget,通过构造将 gadget 串成一条链,从自动调用的入口函数开始链式调用到危险函数。

从上述原因中,可以提取出反序列化漏洞利用环节的三要素。污染源:攻击者可控的变量;入口函数:反序列化时自动调用的接口函数;危险函数:能够执行命令的函数。

因此,挖掘反序列化利用链首先需要根据反序列化器的实现,分析出入口函数和污点源;然后从入口函数开始,根据声明类型初始化污点源,跟踪污点传播。同时在追踪的过程中,会遇到新的函数调用,这时通过指针分析,解析其真实的调用,进行过程间污点传播分析,判断污点数据能否传播到危险函数。

下面将根据图 1 所示的反序列化利用链的示例,解释寻找 Java 反序列化利用链的详细过程。

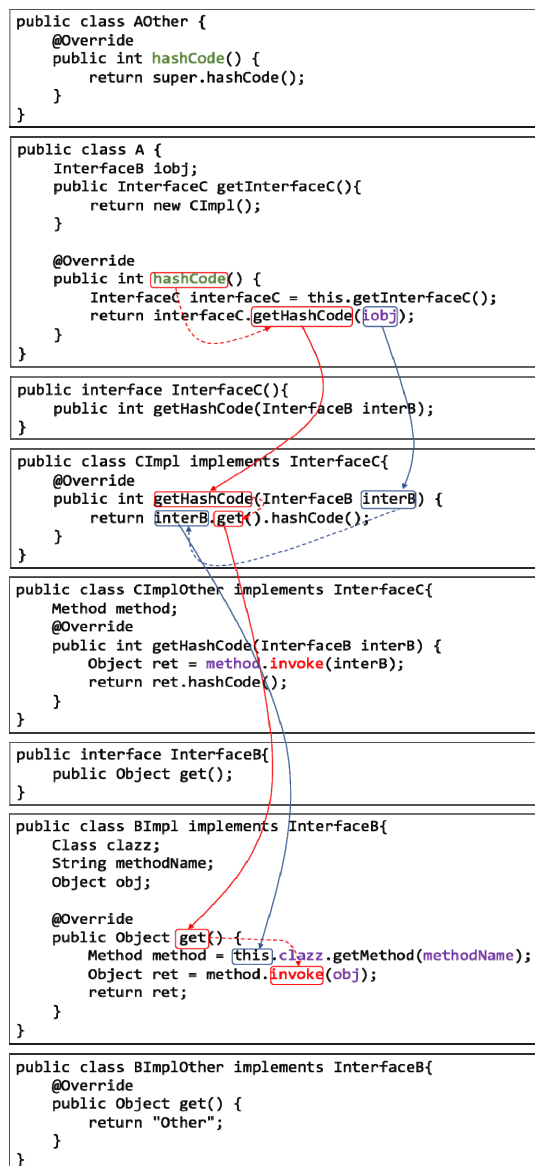


图1 反序列化利用链示例
Figure1 Example of a gadget chain

其中,hashCode 是常见的入口函数,本文从类 A 的 hashCode 函数开始分析调用路径。A#hashCode 先调用 A#getInterfaceC,然后调用 InterfaceC#getHashCode。变量 interfaceC 声明类型是 InterfaceC 接口类,但是由于 Java 多态的性质,根据声明类型 InterfaceC,无法判断出 interfaceC.getHashCode 在运行时调用的是 CImpl#getHashCode 还是 CImplOther# getHash-

Code。因此，为了得到准确的调用图，需要进行指针分析，计算变量 `interfaceC` 在实际运行过程中可能指向的对象类型，再根据指针信息解析实际调用。在图 1 所示的例子中，通过指针分析，可以计算出变量 `interfaceC` 在运行过程中指向 `CImpl` 对象，因此 `interfaceC.hashCode` 对应的实际调用方法是 `CImpl#hashCode`。在 `CImpl#hashCode` 中调用了 `interB.get`，由于变量 `interB` 也是接口类型，所以需要指针分析计算其运行时可能指向的对象类型。变量 `interB` 来自变量 `iobj`，`iobj` 为 `A` 的 `field`。同时由于通过反序列化创建的 `A` 类型对象的 `iobj field` 不像变量 `interfaceC` 有显式的创建对象代码，所以无法通过常规的指针分析计算 `interB` 的指针信息。在反序列化的过程中能够控制类型和属性值，所以可以控制 `iobj` 为 `InterfaceB` 的所有实现类，即图 1 示例中的 `BImpl` 或者 `BImplOther`。因此，`interB.get` 可能会调用 `BImpl#get` 或者 `BImplOther#get`，需要遍历追踪这两个接口，发现 `BImplOther#get` 不会调用危险函数，但这个分支属于不得不排查的无效分支；接下来，在 `BImpl#get` 分支中调用了 `method#invoke`，其中 `clazz` 和 `methodName` 都是 `field`，在反序列化时对应的值是攻击者可控的，因此变量 `method` 也是攻击者可控的，所以该分支最终可以调用 `method.invoke` 危险函数，而且变量 `method` 和 `obj` 都是攻击者可控的。至此，本文找到了一条以 `A#hashCode` 为起点，`Method#invoke` 为终点的反序列化利用链。完整的调用链如图 1 中的红色箭头所示：`A#hashCode`→`CImpl#hashCode`→`BImpl#get`→`Method#invoke`。

过程间的污点传播链为图 1 中蓝色箭头所示：`A#hashCode@iobj`→`CImpl#hashCode@interB`→`BImpl#get@this`。

从上面示例中可以看出，寻找利用链过程的本质是污点分析：分析污点源能否从入口函数传播到危险函数。同时，过程间的污点传播需要准确的函数调用图，但 Java 多态导致不能只根据声明类型确定调用关系。所以需要根据指针分析的结果解析准确的函数调用。反序列化的过程中会隐式创建对象，而且创建的对象可能不唯一，可以是声明类型的所有实现子类，导致传统的指针

分析无法直接应用在这项任务中，需要进一步遍历可能的所有接口。但是反序列化入口函数多，类路径代码量大，反序列化时创建的 `field` 可能是声明类型所有子类对象，这些因素导致直接从所有入口函数进行指针分析，计算量非常大，在有限的时间和受限的内存情况下无法得出指针分析的结果。

另外，如果只通过声明类型构造简单调用图，然后在调用图中寻找从入口函数到危险函数的调用链作为反序列化利用链。这种做法将存在大量的误报：一是因为根据声明类型构造的调用图不准确，调用关系在实际运行的过程中可能不存在；二是即使调用链存在，如果污点源无法传播到危险函数，调用链也将无法利用。

综上所述，虽然通过调用图分析挖掘反序列化利用链的速度快，但调用图不准确导致存在大量的误报；同时为了构造更准确的调用图用于过程间信息流分析，需要进行指针分析。引入指针分析后，准确的调用图让污点传播更准确，但指针分析计算复杂导致效率较低。

因此，为了准确且高效地挖掘反序列化利用链，本文提出了一种基于混合分析的 Java 反序列化利用链挖掘方法。首先，通过调用图分析找出可能到达危险函数的入口函数；然后，利用筛选出的入口函数作为混合信息流分析的入口，构建包含指针信息和污点信息流向的混合信息流图；最后，基于混合信息流图判断从入口函数开始的污点是否能传播到危险函数，从而高效查找实际可行的反序列化利用链。

3 方法设计

3.1 面向危险函数的调用图分析

判断一条利用链是否有效的充分必要条件是：攻击者控制的污点输入能否从利用链的入口函数传播到利用链的敏感危险函数。如果利用链有效，一定存在从入口函数到危险函数的调用路径。因此，有效的反序列化利用链一定在通过调用图分析找到的调用路径集合中，可以通过调用图分析进行筛选，得到可能到达危险函数的入口点集合。

进行调用图分析首先需要构造调用图，调用图中的节点是方法，边为调用关系。Java 中的函

数调用分为 3 类：静态方法调用、特殊方法调用、虚拟方法调用，其中特殊方法调用包括私有方法调用、超级方法调用和构造方法调用。静态方法调用和特殊方法调用，不需要根据指针信息进行解析，能够根据声明类型确定被调用函数。由于 Java 多态的特性，虚拟方法不能根据声明类型确定被调函数。如何解析虚拟方法调用的被调函数是调用图分析面临的主要难题。为了防止调用图不全导致的漏报，本文假设变量在实际运行时，指向声明类型所有的子类。按照这种策略，在图 1 所示的代码中变量 `interfaceC` 可能指向 `CImpl` 或者 `CImplOther`，调用 `interfaceC.hashCode` 被解析成 `CImpl#hashCode` 或者 `CImplOther#hashCode`；同理，调用 `interB.get` 被解析成 `BImpl#get` 或者 `BImplOther#get`。最终调用图如图 2 所示，构造好调用图之后，根据调用图从危险函数开始回溯，寻找能够到达危险函数的入口函数。图 2 中能够到达危险函数的入口函数为 `A#hashCode`，另外一个入口函数 `AOther#hashCode` 无法到达危险函数。在后续污点分析时，将污点的入口函数设置为 `A#hashCode` 即可。入口函数到危险函数的调用路径有两条分别是：`A#hashCode`→`CImpl#hashCode`→`BImpl#get`→`Method#invoke` 和 `A#hashCode`→`CImplOther#hashCode`→`Method#invoke`。第一条是有效的利用链，第二条是调用图不准确导致的误报，即图 2 中红色边表示的调用图中覆盖但程序运行时不存在的调用。

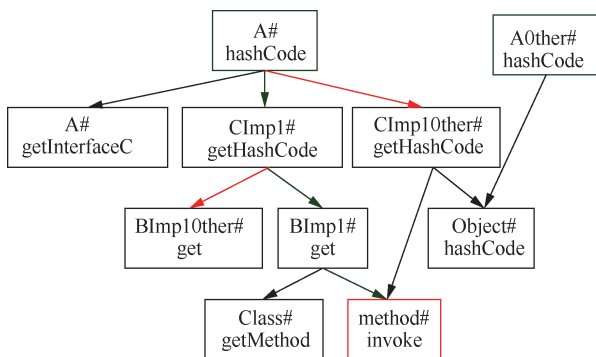


图 2 调用图
Figure 2 Call graph

需要指出的是，在实际的 Java 应用分析中，由于代码量过大，为了能在有限的时间内得到简

单调用图的分析结果，在构造调用图时增加了如下两类约束：一是通过设置参数 `MAXSTEPS`，限制从入口函数出发最大的调用长度，当达到最大的调用长度之后，就不再继续解析新的调用；二是由于 JDK 中的 `java.util.*` 内部调用相对复杂，而且对寻找利用链帮助不大，所以限制以 `java.util` 开头的类的方法，不允许再调用其内部函数。

3.2 面向指针的信息流分析

通过调用图分析，能够找到入口函数到达危险函数的路径集合。但是调用关系在程序实际运行中是否真实存在，还需要指针分析进一步验证。

下面根据图 1 的信息流分析示例介绍面向指针分析的信息流图构造。

(1) 指针信息初始化

传统指针分析中的指针信息初始化操作是 `new` 语句，如图 3 中代码第 9 行 `b3=new B()`，通过 `new` 创建的对象为 `OB9`，赋值操作将对象 `OB9` 的信息传播到变量 `b3`，信息流图中 `OB9` 指向 `b3`。

```

1 public class A{
2     B fldb1;
3     B fldb2;
4     String fldstr;
5
6     public void entry(){
7         B b1 = this.fldb1;
8         B b2 = b1;
9         B b3 = new B();
10        this.fldb2 = b3;
11        B b4 = this.func2(b1);
12        String str1 = this.fldstr;
13        String str2 = str1.substring(1);
14    }
15
16    public B func2(B paramb){
17        B ret = paramb;
18        return ret;
19    }
20 }

```

图 3 信息流分析示例
Figure 3 Example code for information flow analysis

而在反序列化的过程中，程序会隐式创建对象，需要在信息流图中加上隐式对象流向边。隐式创建的对象是声明类型的所有实现子类。图 3 中，反序列化时创建的隐式对象设为 `OA0`，`OA0` 被赋值给 `A#entry@this`，在信息流图中 `OA0` 指向 `A#entry@this`。实际上反序列化时也会隐式对 `field` 创建对象，但静态分析时没有必要对所有 `field` 创建对象，可以在加载 `field` 时再实例化 `field`

对象,这样就可以做到只对用到的 field 才实例化,而且利用该规则可以对 field 对象的 field 进行实例化。在第 7 和 12 行中,加载了 `OA0.fdb1`, `OA0.fldstr`, 此时才创建隐式对象 `OB0`、`OString0`, 并在信息流图中将其指向 `OA0.fdb1`、`OA0.fldstr`。

(2) 过程内信息流图构造

在 Java 中过程内变量之间的信息传播有 Assign、Load、Store 这 3 种语句。

Assign 表示赋值,如图 3 中的第 8 行,变量 `b1` 包含的信息将传播给变量 `b2`, 信息流图中 `b1` 指向 `b2`。

Load 表示获取一个对象的 field 值,如第 7 行,对象 `OA0` 流向变量 `A#entry@this`, 所以 `OA0.fdb1` 包含的信息传播给 `b1`, `OA0.fdb1` 指向 `b1`。

Store 表示对一个对象的 field 进行赋值,第 10 行,对象 `OA0` 流向变量 `A#entry@this`, 所以是 `b3` 包含的信息传播给 `OA0.fdb2`, `b3` 指向 `OA0.fdb2`。

(3) 过程间信息流构造

Call 表示函数调用,涉及过程间变量信息传播。例如图 3 中第 11 行,遇到函数调用时,需要根据变量指针信息解析真实的调用。因为信息流图中对象 `OA0` 流向变量 `A#entry@this`, 因此变量 `A#entry@this` 运行中指向 `OA0` 对象, `OA0` 类型为 `A`, 因此, `this.func2` 实际调用的是 `A#func2`。函数调用涉及 3 种变量传播: 实际参数向形式参数传播, 比如 `b1` 向 `paramb` 传播; 函数内返回值向函数调用返回值传播, 如 `ret` 向 `b4` 传播; 对象向调用方法的 `this` 传播, 如 `A#entry@this` 向 `A#func2@this` 传播。因此, 在控制流图中 `b1` 指向 `paramb`, `ret` 指向 `b4`, `A#entry@this` 指向 `A#func2@this`。

3.3 面向污点的信息流分析

虽然通过指针分析能够得到准确的调用图, 但准确的调用关系还不足以判断一条调用链是否真实有效, 还需要通过污点分析判断外部污点数据能否传播到危险函数。

表面上, 污点分析和指针分析传播不同的信息: 指针分析是在变量之间传递指针指向的对象集合; 污点分析在变量传递内容时根据污点传播策略传递变量对应的污点属性。但是, 污点分析

和指针分析本质上均为信息流分析, 变量之间传播的实际均为对象信息。可以根据指针变量指向的对象是不是污点, 判断当前指针是否需要标记污点属性, 从而将两者合并成同一个过程。例如, 在图 4 中, 对象 `OB0` 在信息流图上能够传播到变量 `b1`, 所以 `b1` 指向对象 `OB0`。如果 `OB0` 含有污点标记, 那么就可以认为指针变量 `b1` 是污点。

基于污点分析等价于传播带污点标记的对象, 本文提出了混合分析——即同时进行指针分析和污点分析。具体地说, 在对象初始化时打上是否为污点的标记, 即可在指针分析的同时进行污点分析。相较于传统的指针分析, 本文根据反序列化应用的场景, 需要针对性地增加规则来完成混合分析, 其主要包含污点源初始化、污点转移规则。

(1) 污点源初始化

污点信息的初始化是将反序列化时隐式创建的对象作为污点源, 打上污点标记。`OA0` 是污点源, 将其标记为图 4 中的红框。

(2) 污点转移规则

污点转移规则包含 3 种。第一种是当加载污点对象的 field 时, 如 `z=x.f`, 如果 `x` 是污点, 那么 `z` 也是污点。因为 `OA0` 是污点源, `OB0` 和 `OString0` 是加载 `OA0` field 时隐式创建的对象, 所以 `OB0` 和 `OString0` 也是污点源。第二种是当调用函数的对象是污点时, 如 `z = x.func()`, 如果 `x` 是污点那么认为 `z` 也是污点。在图 3 中有语句 `str2=str1.substring(1)`, 所以 `str1` 有条指向 `str2` 的污点转移流向的边。第三种是当调用函数的参数是污点时, 如 `z=x.func(y)`, 如果 `y` 是污点, 那么 `z` 也是污点。在实际测试中发现第二种规则会带来误报, 如 `this.getClass().getClassLoader()` 这种情况, 由于 `this` 是污点数据, `getClass()` 返回的也是污点数据, 所以 `getClassLoader()` 返回的也是污点数据, 但实际情况下 `getClassLoader()` 返回值的攻击者并不可控。类似还有 `this.set.Iterator()`, `this.map.keys()` 等的返回值攻击者都不可控。本文通过限制返回值的类型, 阻止这种传播, 减少误报。

因此, 综合利用污点分析和指针分析两方面的信息流传播规则, 得到图 3 对应的混合信息流

图, 如图 4 所示。混合信息流图节点中包括变量节点 (蓝框表示), 隐式对象节点 (红框表示), 显式对象节点 (黑框表示), 对象 field 节点 (绿框表示)。信息流图边包括隐式对象流向、显式对象流向、信息流向、污点转移流向。

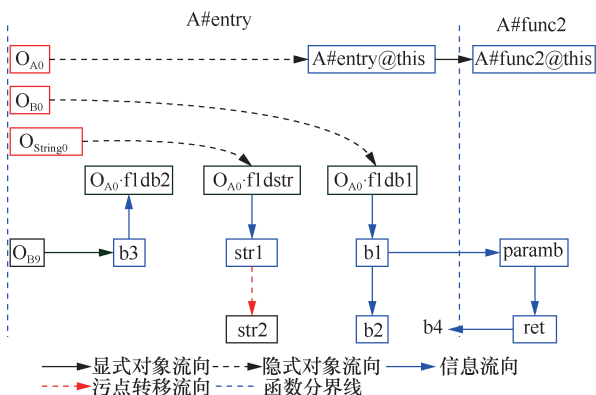


图 4 混合信息流图
Figure 4 Hybrid information flow graph

根据指针分析, 变量 b1、b2、paramb、ret、b4 指向对象 O_{B0}, 变量 str1、str2 指向对象 O_{String0}。由于对象 O_{B0} 和 O_{String0} 都带污点标记所以变量 b1、b2、paramb、ret、b4、str1、str2 都是污点。

综上所述, 本文提出的混合分析方法先通过调用图分析筛选出可能达到危险函数的入口函数; 然后, 再以这些入口函数作为入口进行同时面向指针和污点变量的混合信息流分析, 最后基于信息流分析找出从入口函数开始污点源流向危险函数的路径。

混合分析示意如图 5 所示。其中 A 函数是调用图分析提取的潜在利用链入口函数。将 A 函数

作为混合信息流分析的起点, 通过传播污点标记的对象, 进行指针分析和污点分析。

4 方法实现

基于上述方法, 本文实现了一套自动化挖掘 Java 反序列化利用链的原型系统——Gadget Search, GadgetSearch 支持挖掘 Jackson、Fastjson、XStream、JND、Hessian 5 种常见反序列化器的利用链。

GadgetSearch 首先利用 Soot 解析 Java 字节码, 将字节码转化中间语言 Jimple。然后, 从 Jimple 中解析出类的基本信息, 如类修饰符、父类、field、方法等以及方法中的操作指令 new、Assign、Load、Store、Call, 将这些信息保存成 facts。接着用 Datalog 语法表达第 4 节描述的规则, 用 Souffle 执行 Datalog 文件进行混合分析。分析完成后产生两个主要文件: Nodes.csv 和 Calls.csv。Nodes.csv 记录各个 Method, 有 3 类标签, 分别是 EntryPoint、Method 和 SinkMethod; Calls.csv 记录方法之间的调用关系。最后, 通过 Neo4j-admin 将 Nodes.csv 和 Calls.csv 文件导入 Neo4j 数据库, 并通过 Cypher 语句查询从 EntryPoint 到 SinkMethod 的路径, 对方法调用路径可视化, 方便安全排查调用链是否可利用。

5 实验分析

为了评估本文提出混合分析方法的有效性, 需要设计实验回答下面 3 个问题。

RQ1: GadgetSearch 能否挖掘到未公开的利

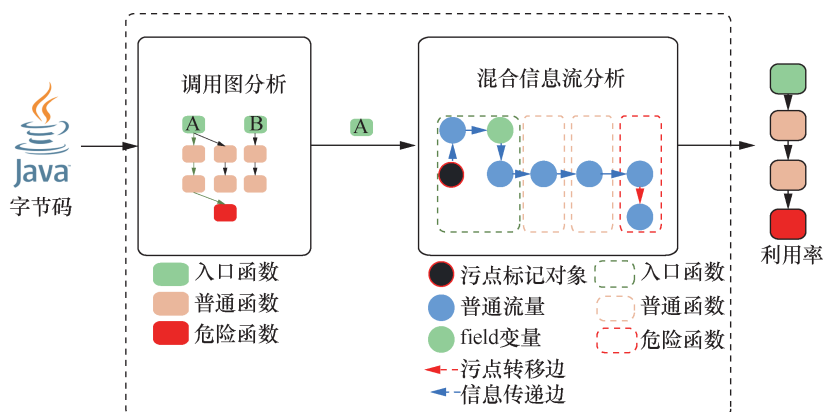


图 5 混合分析示意
Figure 5 Hybrid analysis method diagram

用链?

RQ2: 与现有的反序列化利用链工具相比, GadgetSearch 的性能如何?

RQ3: 如何评估本文提出的混合分析方法的必要性?

测试的实验环境为 Ubuntu16.04, Java 1.8.0_171, 内存 36 GB, 24 核 Intel(R) Xeon(R) CPU E5-2630 处理器。

5.1 针对 RQ1 的实验

为了验证 GadgetSearch 能够挖掘未知利用链, 本文对 JDK 和一些流行的 Jar 包进行了测试。最终发现 5 条未公开利用链, 具体数据如表 1 所示。其中 CVE-2021-43297 是第一条仅依赖 JDK 就能够命令执行的 Hessian 利用链。换句话说, 只要存在 Hessian 反序列化漏洞, 攻击者无须关心类路径包含哪些第三方库, 直接利用这条链即可攻击, 危害十分严重。本文成功利用该链攻击了 Dubbo。

表 1 未公开利用链列表
Table 1 List of CVE/CNVD

CVE/CNVD	反序列化器	依赖	危害
CNVD-2021-44381	Jackson	datanucleus-rdbms-5.1.7.jar	JNDI 注入
CVE-2021-39141	XStream	仅 JDK	代码执行
CVE-2021-39144	XStream	仅 JDK	代码执行
CVE-2021-39146	XStream	仅 JDK	代码执行
CVE-2021-39153	XStream	仅 JDK	代码执行
CVE-2021-43297	Hessian	仅 JDK	代码执行

5.2 针对 RQ2 的实验

挖掘 Java 反序列化的工具有 Jack Of-MostTrades 版本的 GadgetInspector (记为 JGadgetInspector), 以及 Threedr3am 改进后的

GadgetInspector (记为 TGadgetInspector)。JGadgetInspector 支持检测 Jackson、XStream、JND 这 3 种反序列化利用链。GadgetSearch 和 TGadgetInspector 都支持检测 Jackson、Fastjson、XStream、JND、Hessian 这 5 种反序列化利用链。

为了只找出能够命令执行的利用链, 本文去掉了 JGadgetInspector 和 TGadgetInspector 中不能够造成命令执行的危险函数 (如 ssrf, 文件读写等)。TGadgetInspector 检测 Fastjson 利用链时, 其为了检测出新的利用链, 将已知的利用链加入了黑名单, 对黑名单中的链不再检测。为了实验的公平性, 本文将 TGadgetInspector 中 Fastjson 的黑名单置为空。

为了对比 GadgetSearch 和 JGadget Inspector, TGadgetInspector 检测反序列化利用链的误报率、漏报率以及运行效率, 本文在 Marshalsec, Ysoserial、Jackson CVE、XStream CVE 数据集上进行测试。Marshalsec 和 Ysoserial 是两个业界较出名的 Java 反序列化利用链 Payload 生成工具, 包含一些已知的反序列化利用链。Marshalsec 包含 Jackson、XStream、Hessian 3 种反序列化利用链, Ysoserial 只包含 JND 反序列化利用链, 数据集详情如表 2 所示。Jackson 和 XStream 官方会为反序列化利用链申请 CVE, 所以本文从 Jackson 历史 CVE 中筛选出 15 个能够命令执行的 CVE, 从 XStream 历史 CVE 中筛选出 14 个能够命令执行的 CVE 作为另外两组数据集。

Jackson 检测结果如表 3 所示。Marshalsec 数据集中总计有 2 条 Jackson 利用链。JGadgetInspector 挖掘出 8 条链, 其中 0 条有效链, 9 条误报链, 漏报率为 100%, 误报率为 100%;

表 2 Marshalsec 数据集
Table 2 Marshalsec dataset

缩写	Jar	Class	Method	Jackson	Fastjson	XStream	Hessian
cb	commons-beanutils-1.9.2.jar	6 050	66 404	0		1	0
rome	rome-1.0.jar	5 984	65 439	0		1	1
resin	resin-4.0.65.jar	12 850	139 930	0		1	1
xbean	xbean-naming-4.5.jar	3 274	35 098	0		1	1
c3p0	c3p0-0.9.5.2.jar	6 608	73 687	2		0	0
ccfg	commons-configuration-1.10.jar	6 191	67 883	0		1	0

TGadgetInspector 挖掘出 9 条链, 其中 0 条有效链, 9 条误报链, 漏报率为 100%, 误报率为 100%; GadgetSearch 一共挖掘出 18 条利用链, 且均为 18 条有效链, 漏报率为 0%, 误报率 0%。

Fastjson 检测结果如表 4 所示。Marshalsec 数据集中没有相应的 Fastjson 利用链, JGadgetInspector 不支持 Fastjson 利用链的检测。TGadgetInspector 一共检测出 633 条利用链, 其中 0 条有效链, 漏报率 100%, 误报率 100%; GadgetSearch 一共检测出 85 条利用链, 均为有效链, 误报率为 0%, 漏报率为 0%。

XStream 检测结果如表 5 所示。Marshalsec 数据集中包含 5 条 XStream 利用链, JGadgetInspector 检测出 101 条利用链, 其中有 14 条有效利用链, 漏报了 5 条, 误报率为 86%, 漏报率为 100%; TGadgetInspector 一共检测出 269 条利用链, 其中 12 条有效利用链, 漏报了 5 条, 误报率为 95%, 漏报率为 100%; GadgetSearch 一共

检测出 126 条利用链, 且均为有效链, 误报率为 0%, 漏报率为 0%。

Hessian 检测结果如表 6 所示。Marshalsec 数据集中包含 3 条 Hessian 利用链, JGadgetInspector 不支持 Hessian 反序列化利用链的挖掘。TGadgetInspector 一共检测出 1021 条利用链, 其中 6 条有效利用链, 误报率为 99.4%, 没有检测出 Marshalsec 数据集中包含的利用链, 漏报率为 100%; GadgetSearch 一共检测出 36 条利用链, 且均为有效链, 误报率为 0%, 漏报率为 0%。

JND 检测结果如表 7 所示。Ysoserial 数据集中包含 13 条 JND 反序列化利用链。JGadgetInspector 一共检测出 24 条利用链, 分析发现 24 条全是误报, 漏报率为 100%, 误报率 100%。针对 JGadgetInspector 报告的 common-collections 反序列化利用链, 如图 6 所示, 经过分析可认为此条链并不能直接利用。因为最终 Method#invoke 调用时, 第一个参数 input 是

表 3 Jackson 检测结果
Table 3 Results of Jackson

缩写	JGadgetInspector		TGadgetInspector		GadgetSearch	
	时间/s	结果 (有效链数/利用链总数)	时间	结果 (有效链数/利用链总数)	时间/s	结果 (有效链数/利用链总数)
cb	49.13	0/1	60.79	0/1	9.12	0/0
rome	51.13	0/1	59.82	0/1	6.28	0/0
resin	58.68	0/3	75.40	0/4	9.37	0/0
xbean	48.79	0/1	63.53	0/1	4.45	0/0
c3p0	50.03	0/1	62.31	0/1	16.43	18/18
ccfg	52.45	0/1	63.70	0/1	33.07	0/0
总计	310.21	0/8	385.55	0/9	78.72	18/18

表 4 Fastjson 检测结果
Table 4 Results of Fastjson

缩写	JGadgetInspector		TGadgetInspector		GadgetSearch	
	时间/s	结果 (有效链数/利用链总数)	时间/s	结果 (有效链数/利用链总数)	时间/s	结果 (有效链数/利用链总数)
cb			60.79	0/101	7.18	0/0
rome			59.82	0/96	9.04	0/0
resin			75.40	0/96	1085.32	67/67
xbean			63.53	0/101	3.81	0/0
c3p0			62.31	0/97	22.51	10/10
ccfg			63.70	0/142	60.01	8/8
总计			385.55	0/633	1187.87	85/85

表 5 XStream 检测结果
Table 5 Results of XStream

缩写	JGadgetInspector		TGadgetInspector		GadgetSearch	
	时间/s	结果（有效链数/利用链总数）	时间/s	结果（有效链数/利用链总数）	时间/s	结果（有效链数/利用链总数）
cb	51.26	2/12	60.79	2/32	20.95	20/20
rome	52.29	2/12	59.82	2/27	7.71	60/60
resin	60.69	4/40	75.40	2/107	206.91	19/19
xbean	51.12	2/12	63.53	2/30	3.27	0/0
c3p0	52.67	2/12	62.31	2/27	9.12	1/1
ccfg	52.83	2/13	63.70	2/46	8.67	26/26
总计	320.86	14/101	385.55	12/269	256.63	126/126

表 6 Hessian 检测结果
Table 6 Results of Hessian

缩写	JGadgetInspector		TGadgetInspector		GadgetSearch	
	时间/s	结果（有效链数/利用链总数）	时间/s	结果（有效链数/利用链总数）	时间/s	结果（有效链数/利用链总数）
cb			60.79	1/156	120.35	0/0
rome			59.82	1/151	18.79	30/30
resin			75.40	1/249	129.25	5/5
xbean			63.53	1/156	3.78	1/1
c3p0			62.31	1/152	16.43	0/0
ccfg			63.70	1/157	33.07	0/0
总计			385.55	6/1021	321.67	36/36

表 7 JND 检测结果
Table 7 Results of JND

Jar	Class	Method	yso-serial	JGadgetInspector		TGadgetInspector		GadgetSearch	
				时间/s	结果（有效链数/利用链总数）	时间/s	结果（有效链数/利用链总数）	时间/s	结果（有效链数/利用链总数）
bsh-2.0b5.jar	6033	65696	1	49.18	0/2	57.37	0/2	6.59	0/0
c3p0-0.9.5.2.jar	6608	73687	1	51.78	0/2	58.54	0/2	12.66	6/6
click-nodeps-2.3.0.jar	3394	36764	1	49.04	0/4	58.08	0/4	4.33	1/1
commons-beanutils-1.9.2.jar	6050	66404	1	49.36	0/2	57.67	0/2	10.08	1/2
commons-collections-3.1.jar	6290	67873	5	51.46	0/4	57.67	0/4	50.02	20/20
js-1.7R2.jar	6178	68039	2	51.63	0/3	59.59	0/3	10.16	13/13
rome-1.0.jar	5984	65439	1	48.76	0/2	56.48	0/3	7.91	60/60
vaadin-server-7.7.14.jar	7005	74987	1	52.18	0/5	58.94	0/6	32.51	18/18
总计	47542	518889	13	403.39	0/24	464.34	0/26	134.26	118/118

Class 类，无法造成危害，不能达到命令执行的效果，如图 7 所示。TGadgetInspector 一共发现了 26 条利用链，26 条利用链均为误报，漏报率 100%，误报率 100%。GadgetSearch 一共检测出

118 条利用链，均为有效链，针对 bsh-2.0b5.jar 利用链过长超过了 GadgetSearch 设置的最大长度，未被检测出有效利用链，漏报率为 7%，误报率为 0%。

```
com/sun/corba/se/spi/orbutil/proxy/CompositeInvocationHandlerImpl#invoke 0
org/apache/commons/collections/map/LazyMap#get 0
org/apache/commons/collections/functors/InvokerTransformer.#transform 0
java/lang/reflect/Method#invoke 0
```

图 6 JGadgetInspector 检测 common-collections 结果
Figure 6 The results of common-collections detected by JGadgetInspector

```
public Object CompositeInvocationHandlerImpl#invoke( Object proxy, Method
method, Object[] args )
{
    Class cls = method.getDeclaringClass() ;
    InvocationHandler handler =
    (InvocationHandler)classToInvocationHandler.get( cls ) ;

    public Object LazyMap#get(Object key) {
        if (map.containsKey(key) == false) {
            Object value = factory.transform(key);

            public Object InvokerTransformer#transform(Object input) {
                try {
                    Class cls = input.getClass();
                    Method method = cls.getMethod(iMethodName, iParamTypes);
                    return method.invoke(input, iArgs);
```

图 7 common-collections 无效利用链
Figure 7 common-collections Invalid gadget chain

在 Jackson CVE 数据集上 JGadgetInspector 和 TGadgetInspector 均未检测出相应包含关键类的利用链，漏报率 100%，GadgetSearch 则全部检测出，漏报率 0%，检测结果如表 8 所示。

在 XStream CVE 数据集上的检测结果如表 9

所示，JGadgetInspector 和 TGadgetInspector 能检测出 3 个 CVE，GadgetSearch 能够检测出 12 个 CVE。其中 CVE-2021-39141、CVE-2021-39144、CVE-2021-39146、CVE-2021-39153 为上报给 XStream 官方的，此外 CVE-2021-39149 基本等同

表 8 Jackson CVE 检测结果
Table 8 Results of Jackson CVE

CVE	Class	JGadgetInspector	TGadgetInspector	GadgetSearch
CVE-2018-7489	com.mchange.v2.c3p0.ComboPooledDataSource	×	×	√
CVE-2019-17267	net.sf.ehcache.hibernate.EhcacheJtaTransactionManagerLookup	×	×	√
CVE-2019-20330	net.sf.ehcache.transaction.manager.selector.GenericJndiSelector	×	×	√
CVE-2019-20330	net.sf.ehcache.transaction.manager.selector.GlassfishSelector	×	×	√
CVE-2020-14062	com.sun.org.apache.xalan.internal.lib.sql.JNDIConnectionPool	×	×	√
CVE-2020-14195	org.jsecurity.realm.jndi.JndiRealmFactory	×	×	√
CVE-2020-24750	com.pastdev.httpcomponents	×	×	√
CVE-2020-35490	org.apache.commons.dbcp2.datasources.PerUserPoolDataSource	×	×	√
CVE-2020-35491	org.apache.commons.dbcp2.datasources.SharedPoolDataSource	×	×	√
CVE-2020-36179	org.apache.commons.dbcp.cpdsadapter.DriverAdapterCPDS	×	×	√
CVE-2020-36180	org.apache.commons.dbcp2.cpdsadapter.DriverAdapterCPDS	×	×	√
CVE-2020-36181	org.apache.tomcat.dbcp.dbcp.cpdsadapter.DriverAdapterCPDS	×	×	√
CVE-2020-36182	org.apache.tomcat.dbcp.dbcp2.cpdsadapter.DriverAdapterCPDS	×	×	√
CVE-2020-36186	org.apache.tomcat.dbcp.dbcp.datasources.PerUserPoolDataSource	×	×	√
CVE-2020-36187	org.apache.tomcat.dbcp.dbcp.datasources.SharedPoolDataSource	×	×	√
总计	15	0	0	15

注：“√”表示工具检测出相应的 CVE，“×”表示未检测出。

表 9 XStream CVE 检测结果
Table 9 Results of XStream CVE

CVE	class	JGadgetInspector	TGadgetInspector	GadgetSearch
CVE-2013-7285	Java.beans.EventHandler	×	×	√
CVE-2020-26217	Javax.imageio.ImageIO\$ContainsFilter	×	×	√
CVE-2021-21344	com.sun.xml.internal.bind.v2.runtime.reflect.Accessor \$Getter-SetterReflection	×	×	×
CVE-2021-21345	com.sun.xml.internal.bind.v2.runtime.reflect.Accessor \$Getter-SetterReflection	×	×	×
CVE-2021-21346	sun.swing.SwingLazyValue	√	√	√
CVE-2021-21347	com.sun.tools.Javac.processing.JavacProcessingEnvironment\$NameProcessIterator.NameProcessIterator	×	×	√
CVE-2021-21350	com.sun.tools.Javac.processing.JavacProcessingEnvironment\$NameProcessIterator.NameProcessIterator	×	×	√
CVE-2021-21351	com.sun.org.apache.xml.internal.dtm.ref.IncrementalSAXSource_Xerces	×	×	√
CVE-2021-39141	com.sun.xml.internal.ws.client.sei.SEIStub	×	×	√
CVE-2021-39144	sun.tracing.ProviderSkeleton	√	√	√
CVE-2021-39146	Javax.swing.UIDefaults\$ProxyLazyValue	×	×	√
CVE-2021-39149	sun.tracing.ProviderSkeleton	√	√	√
CVE-2021-39153	com.sun.Javafx.fxml.BeanAdapter	×	×	√
CVE-2021-39154	Javax.swing.UIDefaults\$ProxyLazyValue	×	×	√
总计	14	3	3	12

注：“√”表示工具检测出相应的 CVE，“×”表示未检测出。

于本文发现的 CVE-2021-39144, CVE-2021-39154 基本等同于本文发现的 CVE-2021-39146。CVE-2020-21344 和 CVE-2020-21345 利用链过长, 导致 GadgetSearch 无法检测出相应的利用链。

从实验结果可以看出, GadgetSearch 的漏报率和误报率都远低于 JGadgetInspector 和 TGadgetInspector, 主要原因是 JGadgetInspector 和 TGadgetInspector 入口函数和危险函数设置不全, 以及没有进行指针分析且调用图不准确。

GadgetSearch 检测 Jackson、XStream、Hessian、JND 的运行效率优于 JGadgetInspector 和 TGadgetInspector; 检测 Fastjson 的效率低于 JGadgetInspector 和 TGadgetInspector。GadgetSearch 设置的危险函数比 TGadget Inspector 全, 在测试 resin 时 GadgetSearch 找到的从入口函数到危险函数的路径比 TGadgetInspector 多, 导致 GadgetSearch 后续指针分析和污点分析消耗了较多的时间。

5.3 针对 RQ3 的实验

为了验证本文提出的混合分析方法的有效

性, 本文做了 3 组对比实验, 分别利用调用图分析、混合信息流分析以及混合分析, 实验结果如表 10 所示。在入口函数设置为 toString 和危险函数设置为 Method#invoke 的情况下, 对 JDK 中的 XStream 利用链进行检测。实验结果表明, 调用图分析效率最高, 但误报数也最高。混合信息流分析在有限的时间 (24 h) 未能完成分析, 混合分析误报数最低, 同时运行时间在可接受的范围。混合分析的误报主要来自转移函数的黑名单不健全, 污染了一些不该污染的返回值。

表 10 3 种分析对比实验结果
Table 10 Comparison of experimental results

分析	运行时间/s	节点	边	有效链	总共链
简单调用图分析	25.01	4 102	72 729	392	38 680
混合信息流分析	N/A	N/A	N/A	N/A	N/A
混合分析	7 978.44	630	2060	392	398

注: N/A 表示不适用。

5.4 Dubbo Hessian 反序列化 RCE 分析

下面将介绍 GadgetSearch 挖掘到了 Hessian

仅依赖 JDK 的反序列化利用链，并基于此利用链攻击 Dubbo 的全过程。Dubbo 是一个高性能、轻量级的开源服务框架。其默认支持 Hessian 作为其数据传输时序列化和反序列化的方式。Hessian 在反序列化 Map 类型时执行 map#put 时会触发 Object#hashCode，但是 Object#hashCode 调用范围比较小，未能找到可以利用的调用链。为了扩大利用链的搜索范围，寻找除了 Object#hashCode 以外新的入口函数。分析 GadgetSearch 的调用图，发现在执行 HessianInput#readMap 时会触发反序列化之后的 obj 和字符串拼接，触发 obj.toString() 函数，所以本文找到了一个新的入口函数 Object#toString，如图 8 所示。

```
public Object AbstractDeserializer#readMap(AbstractHessianInput in)
{
    Object obj = in.readObject();
    String className = this.getClass().getName();
    if (obj != null) {
        throw this.error(className + ": unexpected object " +
            obj.getClass().getName() +
            " (" + obj + ")");
    }
}
```

图 8 Hessian 新入口函数
Figure 8 New entry point of Hessian

将 Object#toString 添加到入口函数重新运行 GadgetSearch，输出了利用链如图 9 所示。MimeTypeParameterList#toString 中调用了 parameters.get(keys)，parameters 是类型为 Hashtable 的 field，在反序列化时可控，可以设置 parameters 值为 UIDefaults 的实例，当执行 parameters.get(keys) 时，实际是执行 UIDefaults# get，这样就从 MimeTypeParameterList#toString 这个 gadget 跳到了 UIDefaults#get 这个 gadget。UIDefaults#get 中调用了 UIDefaults# getFromHashtable，在 UIDefaults# getFromHashtable 中，通过 super.get(key) 获取了 value 值，由于 UIDefaults 继承了 Hashtable 类，并且 UIDefaults 值是攻击者可控的，所以取出的 value 也是攻击者可控的。如果 value 是 LazyValue 类型，则调用 value.createValue(this)。因为 value 攻击者可控，所以可以设置 value 值为 UIDefaults\$ProxyLazyValue 的实例，调用 value.createValue(this) 时，实际上是调用 UIDefaults\$ProxyLazyValue.createValue(this)，调到了 UIDefaults\$ProxyLazyValue#createValue gadget 上。在 UIDefaults\$ProxyLazyValue#createValue 中变量 className、args、methodName 均为 field，所以可以从类名为

className 中获取方法名为 methodName 的方法，然后通过 MethodUtil.invoke 调用获取的方法，MethodUtil.invoke 就是本文定义的危险函数。通过 4 个 gadget，完成了从入口函数 Object#toString 到危险函数 MethodUtil.invoke 的调用。但是利用到此还未结束，还需要确定要调用哪个类的哪个方法，MethodUtil.invoke(method,obj,args) 第一个参数是要调用的方法，第二个参数是调用的对象，第三个参数是要传的参数。在 UIDefaults\$ProxyLazyValue#createValue 中能控制第一个参数和第三个参数，第二个参数无法设置为对应的对象。所以这里要调用 class 的方法，但调用 class 的方法没有什么危害。此外还可以调用静态函数，因为静态函数的调用，对对象没有要求，即使设置为 null 也可以。所以需要找一个有危害的静态函数调用。从所有 JDK 的静态函数中，找到了两个合适的静态函数，com.sun.org.apache.xml.internal.security.utils.JavaUtils#writeBytesToFilename 和 Java.lang.System#load，JavaUtils#writeBytesToFilename 函数可以对任意文件写任意内容，System#load 可以加载 so 文件，在加载 so 文件时触发代码执行。所以可以将两者组合，利用 JavaUtils#writeBytesToFilename 在临时目录写入一个恶意的 so 文件，然后再用 System#load 加载该恶意的 so 即可触发命令执行。

```
public String MimeTypeParameterList#toString() {
    Enumeration<String> keys = parameters.keys();
    while (keys.hasMoreElements()) {
        String key = keys.nextElement();
        buffer.append(quote(parameters.get(key)));
    }
}

public Object UIDefaults#get(Object key) {
    Object value = getFromHashtable(key);
    return (value != null) ? value : getFromResourceBundle(key, null);
}

private Object UIDefaults#getFromHashtable(final Object key) {
    Object value = super.get(key);
    if (value instanceof LazyValue) {
        try {
            value = ((LazyValue)value).createValue(this);
        }
    }
}

public Object UIDefaults$ProxyLazyValue#createValue(final UIDefaults table) {
    return AccessController.doPrivileged(new PrivilegedAction<Object>() {
        public Object run() {
            try {
                c = Class.forName(className, true, (ClassLoader)cl);
                if (methodName != null) {
                    Class[] types = getClassArray(args);
                    Method m = c.getMethod(methodName, types);
                    return MethodUtil.invoke(m, c, args);
                }
            }
            catch (Exception e) {
                return null;
            }
        }
    }, acc);
}
```

图 9 Hessian 反序列化利用链
Figure 9 Hessian gadget chain

6 结束语

本文提出了一种基于混合分析的反序列化利

用链挖掘方法,并基于该方法实现了一款自动化挖掘 Java 反序列化利用链工具 GadgetSearch,利用该工具发现了一条第三方库中的 Jackson 未公开利用链,4 条 JDK 中的 XStream 未公开利用链,1 条 JDK 中的 Hessian 未公开利用链。同时,在对比实验中 GadgetSearch 的误报率和漏报率远低于现有工具 GadgetInspector。

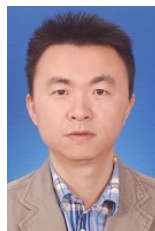
参考文献:

- [1] ANDERSEN L O. Program analysis and specialization for the C programming language[D]. University of Copenhagen, 1994.
- [2] LHOTÁK O, HENDREN L. Scaling Java points-to analysis using Spark[C]//International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 2003: 153-169.
- [3] BERNDL M, LHOTÁK O, QIAN F, et al. Points-to analysis using BDDs[J]. ACM SIGPLAN Notices, 2003, 38(5): 103-114.
- [4] 黄波, 臧斌宇, 韦俊银, 等. 上下文敏感的过程间指针分析[J]. 计算机学报, 2000, 23(5): 477-485.
HUANG B, ZANG B Y, WEI J Y, et al. Context sensitive inter-procedural pointer analysis[J]. Chinese Journal of Computers, 2000, 23(5): 477-485.
- [5] BRAVENBOER M, SMARAGDAKIS Y. Strictly declarative specification of sophisticated points-to analyses[C]//Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. 2009: 243-262.
- [6] ANTONIADIS T, TRIANTAFYLLOU K, SMARAGDAKIS Y. Porting doop to Soufflé: a tale of inter-engine portability for data-log-based analyses[C]//Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis. 2017: 25-30.
- [7] TRIPP O, PISTOIA M, FINK S J, et al. TAJ: effective taint analysis of web applications[J]. ACM Sigplan Notices, 2009, 44(6): 87-97.
- [8] SRIDHARAN M, ARTZI S, PISTOIA M, et al. F4F: taint analysis of framework-based web applications[C]//Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. 2011: 1053-1068.
- [9] HUANG W, DONG Y, MILANOVA A. Type-based taint analysis for Java web applications[C]//International Conference on Fundamental Approaches to Software Engineering. Springer, Berlin, Heidelberg, 2014: 140-154.
- [10] LERCH J, HERMANN B, BODDEN E, et al. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases[C]//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014: 98-108.
- [11] 王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实际应用[J]. 软件学报, 2017, 28(4): 860-882.
WANG L, LI F, LI L, et al. Principle and practice of taint analysis[J]. Journal of Software, 2017, 28(4): 860-882.
- [12] LIVSHITS B. Improving software security with precise static and runtime analysis[D]. Stanford University, 2006.
- [13] TRIPP O, PISTOIA M, COUSOT P, et al. Andromeda: accurate and scalable security analysis of web applications[C]//Fundamental Approaches to Software Engineering. 2013: 210-225.
- [14] ARZT S, RASTHOFER S, FRITZ C, et al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android Apps[J]. Acm Sigplan Notices, 2014, 49(6): 259-269.
- [15] JOHNSON A, WAYE L, MOORE S, et al. Exploring and enforcing security guarantees via program dependence graphs[J]. ACM SIGPLAN Notices, 2015, 50(6): 291-302.
- [16] HUANG W, DONG Y, MILANOVA A, et al. Scalable and precise taint analysis for Android[C]//Proceedings of the 2015 International Symposium on Software Testing and Analysis. 2015: 106-117.
- [17] GRECH N, SMARAGDAKIS Y. P/Taint: unified points to and taint analysis[J]. Proceedings of the ACM on Programming Languages, 2017: 1-28.
- [18] DAHSE J, KREIN N, HOLZ T. Figurer euse attacks in PHP: automated POP chain generation[C]//Proceedings of CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014: 42-53.
- [19] SHCHERBAKOV M, BALLIU M. SerialDetector: principled and practical exploration of object injection vulnerabilities for the web[C]//Proceedings 2021 Network and Distributed System Security Symposium. Reston, VA: Internet Society, 2021.
- [20] HAKEN I. Automated discovery of deserialization gadget chains[R]. 2018.
- [21] 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]. 信息安全学报, 2020, 20(7): 19-29.
DU X Y, YE H, WEN W P. Java deserialization vulnerability gadget chain discovery method based on bytecode search[J]. Netinfo Security, 2020, 20(7): 19-29.

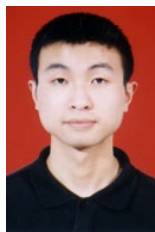
[作者简介]



武永兴(1997-),男,安徽亳州人,上海交通大学硕士生,主要研究方向为自动化代码审计、Web 安全。



陈力波(1985-),男,江西广丰人,上海交通大学工程师,主要研究方向为软件与系统安全、物联网安全、虚拟化安全等。



姜开达(1980-),男,安徽池州人,上海交通大学高级工程师,主要研究方向为网络信息安全、安全漏洞挖掘、网络流量监测。