软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(6):2628−2640 [doi: 10.13328/j.cnki.jos.006852]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 Java 语言生态的软件供应链安全分析技术*

毛天宇[1,3], 王星宇[2,3], 常 瑞[1,4], 申文博[1], 任 奎[1]

[1](浙江大学 网络空间安全学院, 浙江 杭州 310058)

[2](浙江大学 软件学院, 浙江 宁波 310013)

[3](浙江大学 杭州国际科创中心, 浙江 杭州 311200)

[4](浙江省区块链与网络空间治理重点实验室(浙江大学), 浙江 杭州 310027)

通信作者: 常瑞, E-mail: crix1021@zju.edu.cn

摘 要: 随着开源软件技术的不断发展, 为提高开发效率并降低人力成本, 组件化开发模式逐渐得到行业的认可, 开发人员可以利用相关工具便捷地使用第三方组件, 也可将自己开发的组件贡献给开发社区, 从而形成了软件供应链. 然而, 这种开发模式必然会导致高危漏洞随组件之间的依赖链条扩散到其他组件或项目, 从而造成漏洞影响的扩大化. 例如 2021 年底披露的 Log4j2 漏洞, 通过软件供应链对 Java 生态安全造成了巨大影响. 当前, 针对 Java 语言软件供应链安全的分析与研究大多是对组件或项目进行抽样调研, 这忽略了组件或项目对整个开源生态的影响, 无法精准衡量其对生态所产生的影响. 为此, 针对 Java 语言生态软件供应链安全分析技术展开研究, 首次给出了软件供应链安全领域的组件依赖关系和影响力等重要指标的形式化定义, 并据此提出了基于索引文件的增量式组件配置收集和基于 POM 语义的多核并行依赖解析, 设计实现了 Java 开源生态组件依赖关系提取与解析框架, 收集并提取超过 880 万个组件版本和 6 500 万条依赖关系. 在此基础上, 以受到漏洞影响的日志库 Log4j2 为例, 全面评估其对生态的影响以及修复比例. 结果表明: 该漏洞影响了生态 15.12%的组件(71 082 个)以及 16.87%的组件版本(1 488 971 个), 同时, 仅有 29.13%的组件在最新版本中进行了修复.

关键词: 软件供应链; 组件依赖关系; 漏洞传播影响力; Log4j2

中图法分类号: TP311

中文引用格式: 毛天宇, 王星宇, 常瑞, 申文博, 任奎. 面向 Java 语言生态的软件供应链安全分析技术. 软件学报, 2023, 34(6): 2628–2640. http://www.jos.org.cn/1000-9825/6852.htm

英文引用格式: Mao TY, Wang XY, Chang R, Shen WB, Ren K. Software Supply Chain Analysis Techniques for Java Ecosystem. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2628–2640 (in Chinese). http://www.jos.org.cn/1000-9825/6852.htm

Software Supply Chain Analysis Techniques for Java Ecosystem

MAO Tian-Yu[1,3], WANG Xing-Yu[2,3], CHANG Rui[1,4], SHEN Wen-Bo[1], REN Kui[1]

[1](School of Cyber Science and Technology, Zhejiang University, Hangzhou 310058, China)

[2](College of Software Technology, Zhejiang University, Ningbo 310013, China)

[3](ZJU-Hangzhou Global Scientific and Technological Innovation Center, Zhejiang University, Hangzhou 311200, China)

[4](Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province (Zhejiang University), Hangzhou 310027, China)

Abstract: With the prosperity of open-source software, almost all software companies use these reusable components as basic build blocks to build their software products, thus forming the software supply chain. The software supply chain improves development

* 基金项目: 国家重点研发计划(2022YFE0113200); 浙江省重点研发计划(2022C01165)

本文由“软件可信性与供应链安全前沿进展”专题特约编辑向剑文教授、郑征教授、申文博研究员、常瑞副教授、田聪教授推荐.

毛天宇和王星宇对本文工作具有相同的贡献, 被视作共同一作.

收稿时间: 2022-09-05; 修改时间: 2022-10-10, 2022-12-14; 采用时间: 2022-12-28; jos 在线出版时间: 2023-01-13

efficiency and reduces labor costs for software companies. However, it may also introduce new security problems. In particular, if one software component has high-risk vulnerabilities, the software supply chain inevitably spreads these vulnerabilities to all its dependencies, thus amplifying these vulnerabilities' impact. For example, through the software supply chain, the Log4j2 vulnerability causes a catastrophic security issue for the whole Java ecosystem. Unfortunately, current research studies on Java software supply chain mainly focus on a single component or a group of components and miss the impact study on the ecosystem scale. Therefore, this paper presents the essential software supply analysis techniques to study the component and vulnerability impact on the Java ecosystem. More specifically, the formal definition of component dependencies is first given in the software supply chain. Next, new techniques are proposed and an analysis tool is built to analyze all component dependencies in the Java ecosystem, including over 8.8 million component versions and 65 million dependencies. Finally, Log4j2, a logging library affected by the vulnerability, is used as an example to evaluate its impact on the whole Java ecosystem. The results show that the vulnerability affects 15.12% of the ecological components (71 082) and 16.87% of the component versions (1 488 971), and the vulnerability-fix rate is only 29.13%.

Key words: software supply chain; component dependencies; vulnerability propagation impact; Log4j2

伴随开源软件生态的发展, 基于第三方可复用组件的二次开发模式逐渐成为一种趋势, 从而形成了软件供应链. 根据 Synopsys 发布的 2022 年开源安全和风险分析报告[1], 通过对不同行业领域的 2 409 个代码库进行审计后发现, 97%的代码库都包涵第三方开源代码, 其中, 计算机硬件和半导体、网络安全、能源与清洁科技以及物联网这 4 个行业软件对第三方开源软件的使用率达到了 100%. Java 生态作为最为活跃的开源生态之一, 其最大的第三方组件仓库 Maven-Central[2], 目前已经托管接近 50 万个 Java 第三方组件, 共计超过 880 万个不同版本. 根据 SONATYPE 的官方统计[3], 2021 年度, Java 生态开源组件的下载量达到 4 570 亿次. 如此大规模地使用第三方组件, 对 Java 软件生态系统构成了新的安全威胁, 例如 2021 年底爆出的 Log4j2 漏洞, 对 Java 生态安全造成了巨大影响.

目前已有的研究工作[4−27]大多是小规模分析抽样项目的依赖关系以及是否受到漏洞影响, 但是由于这些工作所采用的动态分析检测方法(例如, 实际安装一个项目来获取依赖关系)以及效率等因素, 往往无法有效扩展至生态层面, 也缺乏高效、精准的解析. 因此, 对依赖关系以及漏洞传播影响力的大规模分析和生态层面的呈现, 仍具有较大的挑战性.

为了弥补上述缺陷, 主要面临的技术挑战包括:

1) 完整性. Java 生态系统目前拥有超过 880 万个不同版本的第三方组件, 很难全面分析, 目前已有工作大多通过抽样, 只分析了小规模的组件;
2) 准确性. Java 生态的组件依赖关系更加复杂, 存在其特有的依赖配置方式, 导致目前一些基于静态分析组件依赖配置文件的工作准确性无法得到保证;
3) 特征分析. 目前, 对于漏洞传播给 Java 生态造成的影响只停留在数量上的分析, 无法反映漏洞在依赖链上传播的特征.

为了解决上述挑战, 本文首先选取了 Maven-Central 作为研究对象, 全面分析了 Maven-Central 上已发布组件的依赖配置文件, 实现了一个组件依赖信息自动化提取与解析框架, 该框架可以快速地更新以及解析 Maven-Central 上所发布组件的依赖关系; 其次, 为了指导漏洞组件的分析, 提出了组件影响力等相关重要指标的形式化定义; 最终, 基于大规模的依赖关系, 对漏洞日志库 Log4j2 给 Java 生态造成的影响进行了全面的分析, 给出了影响力的层级图以及传播路径上的关键节点用于指导漏洞修复, 并揭示了当前该漏洞的整体修复情况. 综上所述, 本文的主要贡献如下:

- 重要指标的形式化定义: 首次对组件依赖关系和漏洞组件影响力等软件供应链安全领域相关重要指标给出形式化定义, 为漏洞传播及影响力分析提供理论依据;
- 高效、精准的依赖解析: 构建了自动化依赖信息提取与解析框架, 实现了高效、精准的依赖解析器, 对超过 47 万个第三方组件的所有版本(共计超过 880 万个版本)进行依赖关系提取, 得到 6 500 万条直接依赖关系. 最终, 在超过 1 万个流行组件的最新版本数据集上进行验证, 准确度达到了 94.64%;
- 漏洞影响力的全面分析: 针对漏洞日志库 Log4j2, 分析了其对生态造成的影响以及修复情况, 发现该

漏洞影响了生态中 15.12%的组件(71 082 个)以及 16.87%的组件版本(1 488 971 个), 且生态整体修复情况不容乐观, 修复比例仅为 29.13%.

本文第 1 节介绍研究工作的动机和相关概念. 第 2 节对软件供应链领域重要指标进行定义. 第 3 节阐述组件依赖关系提取与解析框架的设计思路. 第 4 节以解析得到的依赖关系为基础, 对受漏洞影响的 Log4j2 组件进行影响力以及修复情况分析. 第 5 节介绍 Java 语言生态的有关研究工作. 最后, 第 6 节总结本文的工作.

1 研究动机与相关概念

本节首先基于 Log4j2 日志库爆出的高危漏洞事件以及该漏洞对生态的巨大影响力阐述了本文的研究动机; 然后, 为方便介绍后续的研究工作, 在第 1.2 节详细介绍了相关概念.

1.1 研究动机

2021 年 11 月 24 日, 美国国家漏洞数据库(NVD)披露了代号为 CVE-2021-44228[28]的远程代码执行高危漏洞, 其 CVSS 评分高达 10 分. 包含该漏洞的 Log4j2 作为一个 Java 日志框架, 被各大厂商应用程序和云服务广泛使用, 因此, 该漏洞无异于一枚深水炸弹. 该漏洞不仅影响了直接使用该组件的项目, 还影响到许多其他流行的依赖该组件的众多 Java 组件和开发框架. 然而, 阿里云安全团队作为第一个发现该漏洞的组织, 未能意识到该漏洞的严重性, 没有及时向电信主管部门报告, 致使其被工信部暂停作为网络安全威胁信息共享平台合作单位 6 个月[29,30]. 该漏洞组件所属组织在漏洞被披露之后的 1 周时间发布了修复版本, 但并未完全修复相关漏洞, 依然存在被攻击的风险. 最终, 经过几轮修复迭代才得以完全修复. 由此可见, 对漏洞的修复往往也需要一段较长的时间. 如今, 距离该漏洞完全修复已经过去 9 个月, 但由于其生态以及依赖关系的复杂性, 含有漏洞的该组件版本依然被大量下载[31]. 由此可见, 理解漏洞的影响范围、传播特征以及修复情况变得至关重要, 基于软件依赖关系的漏洞分析亦成为当前的研究热点.

1.2 相关概念

由于本文研究的主要对象为发布至 Maven-Central 的组件, 因此本节将主要介绍 Maven 相关概念.

在 Java 开源生态环境中, Maven[32]是使用得最广泛的包管理工具, 开发者通过 Maven 提供的接口, 将组件发布至 Maven-Central, 通过组件的三要素(GroupId,ArtifactId,Version)来确定某个特定版本的组件, 其中, GroupId 一般为组件开发者所属的组织, ArtifactId 为组件的名称, Version 代表该组件的发布版本号. 组件的基础信息被记录在 POM (ProjectObjectModel)[33]文件中. 与本研究相关的配置信息见表 1.

表 1 POM 文件属性简介

属性	说明
Basic	组件基础信息, 包括 GroupId、ArtifactId、Version 等基本属性
Parent	父节点信息
Dependencies	依赖信息
DependencyManagement	依赖配置管理信息
Properties	组件全局配置信息

大多数组件都依赖于其他组件来构建, 因此, 开发者需要在 POM 文件中进行依赖声明等配置. POM 中与依赖相关的声明主要分成两类.

1) 显式声明: 即直接定义在组件 POM 文件中的 Dependencies 属性下, 可以直接解析获取;

2) 继承声明: 即组件通过定义 Parent 字段, 从父节点组件继承依赖配置信息.

开发者还可以通过 DependencyManagement 以及 Properties 属性对依赖信息进行管理.

2 软件供应链形式化指标定义

为了系统地研究软件供应链中的组件间关系特征, 本节给出本文研究工作所使用指标的形式化定义. 其中, 第 2.1 节给出基础定义, 介绍 Java 生态中的基础单元, 即直接依赖和间接依赖关系; 在第 2.1 节的基础上,

第 2.2 节定义直接影响力和传播影响力, 用于揭示组件直接影响其他组件的能力和组件通过依赖链对生态造成的影响.

2.1 基础定义

令 G 为 Maven 生态中的组件版本以及组件间的依赖关系构成的有向无环图, 则 $G=(V,R)$, 其中, V 代表生态中的所有组件版本, $R\subseteq V\times V$ 代表组件版本 $v\in V$ 及其依赖的有向边的集合, (v_i,v_j)代表 v_i 直接依赖于 v_j. 图 1 展示了组件版本之间的依赖关系, 其中每个节点代表具体的组件版本, 组件版本间的关系使用有向箭头表示. 例如: $A_{1.0}$ 代表组件 A 的 1.0 版本, $A_{1.0}$ 指向 $D_{1.0}$ 代表 $A_{1.0}$ 依赖于 $D_{1.0}$.

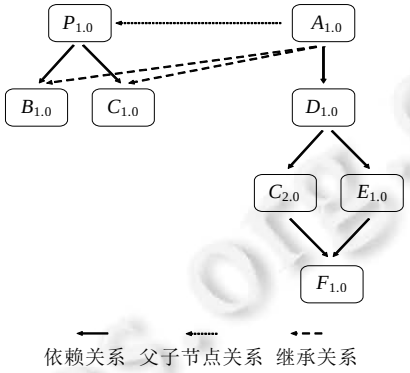

图 1 组件版本依赖关系示例图

定义 2.1(直接依赖). 直接依赖是指在编译或运行等场景下, 其需要直接使用的组件版本, 需要在其配置文件中直接声明或者从父组件继承得到. 令 $Dep_d(v)$指代 v 的直接依赖, $Dep_d(v)=\{v_i|(v,v_i)\in R\}$. 对照图 1, 有:

$$Dep_d(A_{1.0})=\{B_{1.0},C_{1.0},D_{1.0}\}.$$

定义 2.2(间接依赖). 对于所有通过自身直接依赖而间接引入的组件版本, 称其为间接依赖, 如图 1 所示, 这些间接引入的组件版本按照层级分布, 用 $Dep^n(v)$代表 v 的第 n 级依赖, 则 $Dep^n(v)=\{Dep_d(e)|e\in Dep^{n-1}(v)\}$, 其中, $n\geqslant 2$ 且 $Dep^1(v)=Dep_d(v)$, 则 v 的间接依赖为其所有层级大于 1 的依赖集合: $Dep_t(v)=\bigcup_{i=2}^{n}Dep^i(v)$. 如图 1 中, $Dep_t(A_{1.0})=\{C_{2.0},E_{1.0},F_{1.0}\}$.

2.2 组件影响力

组件是指三要素中 GroupId 与 ArtifactId 相同的组件版本集合, 记为 A, 即: $A=\{v_1,v_2,\ldots,v_n\}$, 其中, $\varphi(v_i)=\varphi(v_j)$, $i,j=1\ldots n$. φ代表将组件版本映射为其所属组件.

定义 2.3(组件直接影响力). 组件直接影响力由直接依赖于该组件各版本所属组件构成, 记为 $W_d(A)$, 则 $W_d(A)=W_1(A)=\{\varphi(e)\mid e\in\bigcup_{v\in A}Dep_d(v)\}$. 以图 1 为例, $A_{1.0}$ 以及 $D_{1.0}$ 分别直接依赖于 $C_{1.0}$ 和 $C_{2.0}$, 则组件 C 直接影响组件 A 以及组件 D, 记为 $W_d(C)=\{A,D\}$.

定义 2.4(组件层级影响力). 以定义 2.3 为基础, 定义组件的层级影响力为其被依赖层级为 n 的组件集合, 记为 $W_n(A)$, 则 $W_n(A)=\{\varphi(e)\mid e\in\bigcup_{v\in A}Dep^n(v)\}$. 以图 1 中的组件 $F_{1.0}$ 为例, 组件 $D_{1.0}$ 相对它的依赖路径长度为 2, 则组件 F 的第 2 层级影响力为 $W_2(F)=\{D\}$. 同理, $W_3(F)=\{A\}$.

定义 2.5(组件间接影响力). 以定义 2.4 为基础, 定义组件间接影响力为其被依赖层级大于 1 的组件集合, 记为 $W_t(A)$, 则 $W_t(A)=\bigcup_{i=2}^{n}W_i(A)$. 以图 1 中的组件 $F_{1.0}$ 为例, $W_t(F)=\{D,A\}$.

定义 2.6(组件传播影响力). 组件传播影响力是指组件被依赖路径上能够影响到的所有组件, 即 $W(A)=W_d(A)\cup W_t(A)$. 以图 1 中的组件 $F_{1.0}$ 为例, $W(F)=\{C,E,D,A\}$.

3 组件依赖关系提取与解析框架

为了支持基于依赖关系的漏洞组件在供应链中的传播影响力分析，本文设计并构建了组件依赖关系提取与解析框架. 本节主要介绍数据获取以及依赖关系提取的技术选型、技术实现以及流程.

3.1 框架总览

如图 2 所示: 本文设计并实现了一整套自动化的依赖关系提取框架，对组件配置信息增量式更新、爬取以及提取依赖关系信息. 这里，主要的挑战在于: (1) 如何大规模地收集 Maven 生态中的组件配置信息; (2) 如何实现高效、准确的依赖关系提取. 为了解决上述问题，本文基于 Maven-Central 索引文件以及配置文件的语义，开发了组件信息爬虫以及依赖解析器.

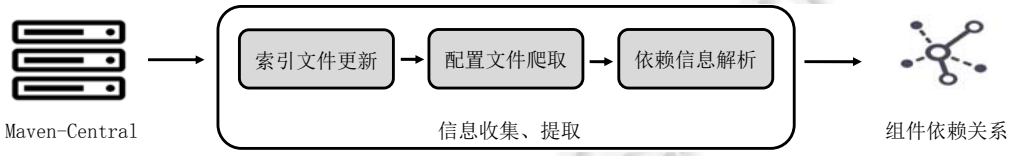

图 2 组件依赖信息自动化提取框架

3.2 基于索引文件的增量式组件配置收集

为了从 Maven-Central 获取组件的配置文件，本文选用爬虫技术自动化地获取组件的配置文件，下面对配置信息收集模块的实现给出详细介绍.

首先对 Maven-Central 的索引系统[34]进行分析. 该索引系统每周会对当前仓库中的组件建立索引文件，索引文件有两种类型: 全量以及增量. 其中，全量索引文件包含了当前所有组件的信息，增量索引文件仅包含自上一个更新周期以来所新增的组件信息. 在首次收集时，系统使用全量索引文件，获取当前时间 Maven-Central 上所有组件的三要素，并记录更新时间节点. 在之后对 Maven-Central 变更的持续追踪中，通过更新时间节点获取本次更新周期的增量索引文件，以加速系统的更新速率. Maven-Central 中，组件配置文件(POM 文件)的请求路径是由组件的三要素来确定的，因此只需获取组件的三要素，并按照一定规则就可以生成组件配置文件的请求路径.

然后，确定爬虫的主要功能. 爬虫按照功能被划分为两个部分: 第 1 部分是根据本地的更新节点信息判断是使用全量索引文件还是增量索引文件来生成任务队列; 第 2 部分为信息爬取，任务队列中的每个任务参数为拟爬取组件的三要素，通过三要素生成该组件配置文件的请求路径，并通过该路径下载组件配置文件.

为满足上述功能，本文设计并实现了针对 Maven-Central 定制化的爬虫模块，其具体工作流程如图 3 所示.

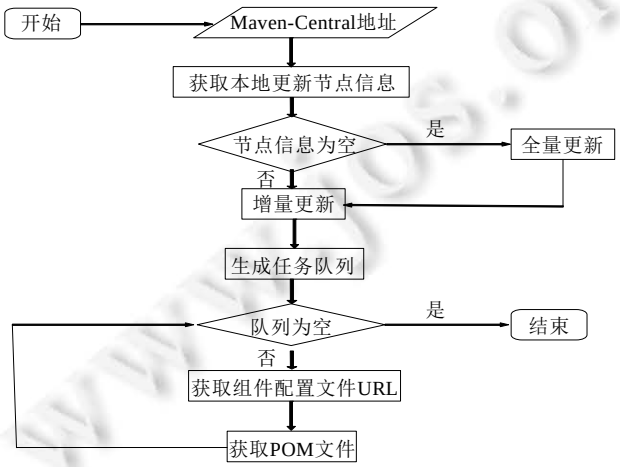

图 3 Maven 组件配置信息爬虫工作流程图

3.3 基于POM语义的多核并行依赖解析

依赖解析过程中, 主要需要解决两个问题: 准确度和解析效率. 下面对依赖信息解析模块的实现给出详细介绍. 为了确保准确率, 本文研究了 POM 文件中各个属性段的语义以及官方包管理工具的解析过程, 主要分两步: 继承和插值. 继承主要处理从父组件继承引入的配置信息; 插值主要用于解析配置文件中的变量或者一些未显式定义的值, 从而获得最终属性值.

根据以上思路, 设计并实现了基于 POM 语义的依赖解析器. 对于继承, 本文设计了递归的继承算法, 通过 POM 中的 Parent 属性, 优先对父节点进行处理. 在目标组件版本获得了父节点的依赖信息之后, 将其与自身的依赖信息进行合并, 在合并过程中如果发生冲突, 则以目标组件版本的声明为准. 如图 1 所示: 在解析组件 $A_{1.0}$ 的直接依赖信息时, 首先解析其父节点 $P_{1.0}$ 的依赖信息集合$\{B_{1.0},C_{1.0},D_{1.0}\}$, 将其与 $A_{1.0}$ 自身声明的依赖信息集合$\{D_{1.0}\}$合并, 得到其最终的依赖信息集合$\{B_{1.0},C_{1.0},D_{1.0}\}$. 为了准确地解析依赖关系, 还需考虑若干其他继承的配置信息, 具体配置信息见表 2.

表 2 继承属性

继承属性
GroupId
Version
Dependencies
DependencyManagement
Properties

为了给予配置文件一定的灵活性, 因此在插值处理阶段, 需要对依赖信息中所有变量或者未显式定义的值进行分析, 得到最终的属性值. 具体而言, 对于例如${PropertyName}的定义, 需要以 PropertyName 为关键字, 在配置文件的 properties 段中查找其最终的属性值. 同时, 对于依赖定义中没有给出的版本信息, 需要根据该依赖的 GroupId 以及 ArtifactId 在 dependencyManagement 段中进行搜索, 以确定具体的依赖组件版本. 若在插值阶段出现依赖信息无法被解析的情况, 则表示该组件的配置文件是不合法的. 拥有不合法配置文件的组件也无法成功编译, 可被视为脏数据, 因此可直接结束该配置文件的解析过程.

为了提高依赖信息解析的效率, 从以下几个方面对解析过程进行了优化.

1) 由于已经通过爬虫技术获取了所有组件的配置文件, 因此可以规避解析时下载所带来的网络 I/O 开销;
2) 为了避免同一个组件配置信息被多次解析(比如多个子组件共享同一个父组件), 系统对每个解析完成的组件配置信息进行了缓存. 每次解析时, 先查询该组件是否已经完成解析: 若完成, 则直接返回; 否则, 进行解析;
3) 由于每个组件的解析是独立的, 因此系统采用多处理器并行的解析模式来加速过程.

3.4 评 测

3.4.1 效 率

效率的测试分为两个部分: 组件收集和依赖解析. 测试平台为: Ubuntu20.04, 512 GB 内存, 20 核 Intel(R) Xeon(R) Gold 5218R CPU@2.10 GHz. 对于组件收集, 仅当第 1 次收集时, 系统需要对当前 Maven-Central 所有的组件信息进行全量更新, 耗时较多, 大部分的耗时在于网络请求. 但是对于之后的增量式更新, 数据规模较小, 可以在 1 小时内完成. 因此, 该部分的效率是满足要求的. 在解析效率方面, 对 880 万个组件版本配置文件在测试平台上进行依赖解析, 耗时小于 2 小时.

3.4.2 准确性

为了验证依赖解析器的准确性, 本文选择 Libraries.io[35]作为数据源. Libraries.io 收集了 32 个不同开源生态的组件信息, 根据组件的相关信息进行评分排序, 并提供了相关 API 供研究人员获取数据. 首先, 按照组件的仓库评分、星标数以及被依赖数这 3 个维度分别选取了 Java 生态排名前 5 000 的组件; 再经过去重, 共得到 12 973 个组件; 最后, 选取每个组件的当前最新版本作为测试数据集. 为了获得测试数据集真实的依赖关

系数据, 我们以 Maven 官方提供的插件工具 Maven Effective-Pom[36]的解析结果为基准. Maven Effective-Pom 插件通过解析 POM 文件中的继承关系和依赖关系, 下载相关依赖配置文件, 最终生成完备的配置信息, 我们将该配置信息中的依赖信息提取出来作为真实值(依赖的 GroupId、ArtifactId、Version). 与此同时, 本文实现的依赖解析器也对以上 11 856 个组件进行了解析, 我们对每个组件的依赖信息进行逐条比对. 表 3 中的测试结果显示, 94.64%的组件依赖信息解析完全一致. 主要有两个原因会造成分析不一致的情况: (1) Maven Effective-Pom 在解析时存在一些默认的解析环境变量, 而本文的依赖解析器无法获取; (2) 部分组件的依赖并未发布到 Maven-Central, 依赖解析系统因此缺失了相关信息.

表 3 依赖解析准确性测试数据集以及结果

数据集	准确率(%)
11 856 个组件版本	94.64

4 漏洞影响力分析

本节基于第 2 节中介绍的指标定义和第 3 节中提取的组件依赖关系, 以日志库 Log4j2 的核心组件 log4j-core 为典型案例进行了关键漏洞组件影响力的实例分析, 并统计关键组件各版本在不同时期被直接依赖的数量, 动态分析了 log4j-core 在漏洞爆出后的修复态势. 以期回答如下问题.

问题 1: log4j-core 对生态产生了多大影响?

问题 2: log4j-core 漏洞目前的修复情况如何?

4.1 log4j-core漏洞在Java生态中的影响力

为了回答问题 1, 下面从 3 个角度来分析 log4j-core 漏洞对 Java 生态的影响力, 包括受影响组件版本的数量、层次分布规律和传播链上的关键组件.

组件 log4j-core 共包含共 53 个组件版本, 时间跨度为 10 年, 版本跨度从 log4j-core@2.0-alpha1 到 log4j-core@2.17.2, 根据 Apache Log4j Security Vulnerabilities[37], 其中有 49 个版本受到 4 个 log4j-core 漏洞的影响, 具体情况见表 4. 后文将综合受这些漏洞影响的 log4j-core 版本, 以研究分析该组件所爆漏洞对 Java 生态的影响.

表 4 log4j-core 组件受漏洞影响的版本范围

CVE 版本号	CVSS 评分	发布日期	受影响版本
CVE-2021-44228	10	2021/12/10	2.0-beta9 至 2.14.1
CVE-2021-45046	9	2021/12/14	2.0-beta9 至 2.15.0, 除去 2.12.2
CVE-2021-45105	5.9	2021/12/18	2.0-alpha1 至 2.16.0, 除去 2.12.3
CVE-2021-44832	6.6	2021/12/28	2.0-beta7 至 2.17.0, 除去 2.3.2 和 2.12.4

(1) 受 log4j-core 漏洞影响的组件版本数量

根据定义 2.1 和定义 2.2 对组件版本之间依赖关系的定义, 以及定义 2.6 对组件传播影响力的定义, 通过解析得到的 6 500 万条依赖关系, 统计了所有依赖于 log4j-core 的组件版本数量及其传递影响力 $W(\text{Log4}j)$, 其结果见表 5.

表 5 漏洞组件 log4j-core 的生态影响力

受影响组件数	受影响组件版本数
7 1082 (15.12%)	1 488 971 (16.87%)

共有 1 488 971 个组件版本直接/间接依赖于 log4j-core 组件, $W(\text{Log4}j)$包含了 71 082 个组件. 其影响了生态中 16.87%的组件版本和 15.12%的组件. 可见, 该组件对整个生态有着不可小觑的影响力.

(2) 受影响组件在传播链上的分布规律

除从宏观角度计量 log4j-core 的影响力外, 本节还根据定义 2.3–定义 2.5 对组件层级影响力的定义, 从依赖链的角度按照传播距离进行分层, 统计了共计 14 层的依赖关系, 按照组件依赖的不同层级统计绘制了图 4.

数据表明: 下游依赖组件大多分布于前 5 层, 相对而言, 漏洞更易被触发.

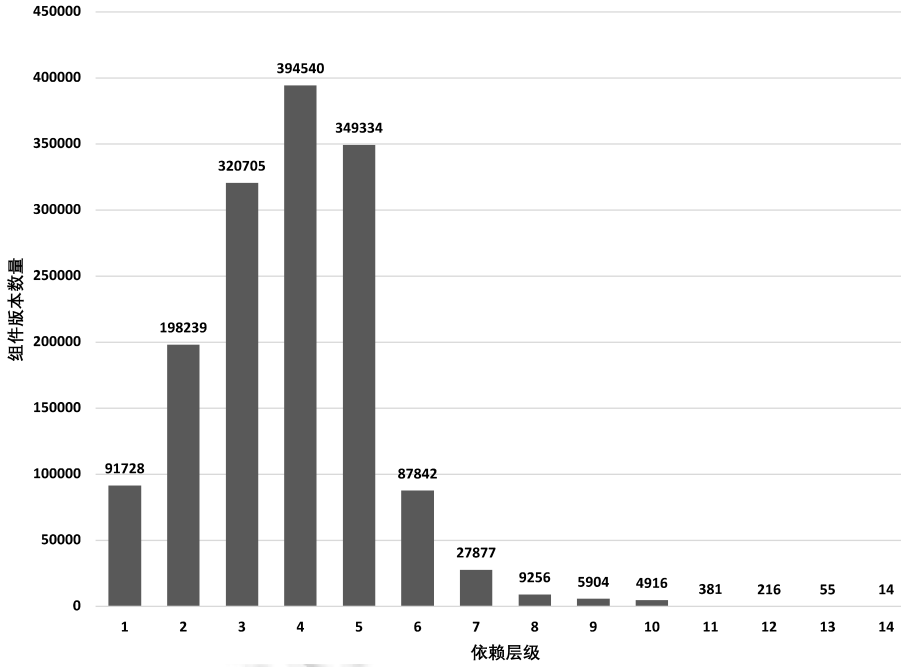

图 4 log4j-core 组件的层级依赖数量

通过图 4 所示的组件版本数量分布, 可以发现以下两个特性.

1) 绝大多数的下游依赖组件主要分布在前 5 层之内, 从第 6 层开始锐减, 14 层之后的依赖数量相较于前 5 层的数量可以忽略不计. 这一规律说明, 大部分受影响组件对 log4j-core 的引用层次较浅, 依赖路径较短, 因此受到该漏洞影响的可能性更高;

2) 对于该漏洞组件的影响力, 其间接影响力的占比更大. 相比于直接依赖, 间接依赖关系更为隐蔽, 因此, 这一特点也是造成该漏洞难以检测和修复的根本原因.

(3) 传播链上的关键组件

本节根据定义 2.6, 对依赖于 log4j-core 组件 14 层内的所有组件进行了传播影响力分析, 根据各组件的影响力大小排序, 选取排名前 10 的组件, 制作了表 6, 展示了在漏洞传播过程中影响范围较广的 10 个关键组件.

表 6 log4j-core 在传播链中的前 10 个关键组件

排名	GroupId: ArtifactId	传播影响力
1	io.netty: netty-common	87 403
2	io.netty: netty-buffer	87 205
3	io.netty: netty-transport	84 887
4	io.netty: netty-codec	84 720
5	io.netty: netty-handler	84 482
6	io.netty: netty-codec-http	79 488
7	io.netty: netty-codec-socks	72 929
8	io.netty: netty-resolver	72 036
9	io.netty: netty-codec-http2	63 612
10	io.netty: netty-codec-haproxy	62 631

从表 6 可以看出, 依赖于 log4j-core 的影响力 top3 组件是 io.netty: netty-common、io.netty: netty-buffer、io.netty: netty-transport. 这些组件存在进一步扩大漏洞传播影响力的可能性. 因此, 关键组件的修复状态对依赖链下游组件的安全也有着不可忽视的潜在影响.

4.2 漏洞修复分析

目前, 完全修复版本为 2.3.2、2.12.4、2.17.1 和 2.17.2. 为了研究 log4j-core 漏洞的修复情况, 将直接依赖于 log4j-core 的组件按照时间进行划分, 选取每个时间点这些组件的最新版本: 若最新版本依赖于 log4j-core 的 4 个修复版本, 则认为该组件在该时间点已经得到修复; 否则, 该组件依然受到漏洞的影响. 最终, 本文逐日统计了从 2021 年 12 月 28 日(log4j-core@2.17.1 发布时间)至 2022 年 5 月 5 日, 所有直接依赖于 log4j-core 各个版本的最新组件数量的变化情况. 然后, 根据漏洞是否被修复对各版本分别加以汇总, 抽样选择了 14 个时间点绘制了直接依赖组件版本修复比例对比图, 如图 5 所示. 其中, 黑色为依赖于受漏洞影响的组件占比, 灰色则是依赖于修复版本的组件比例.

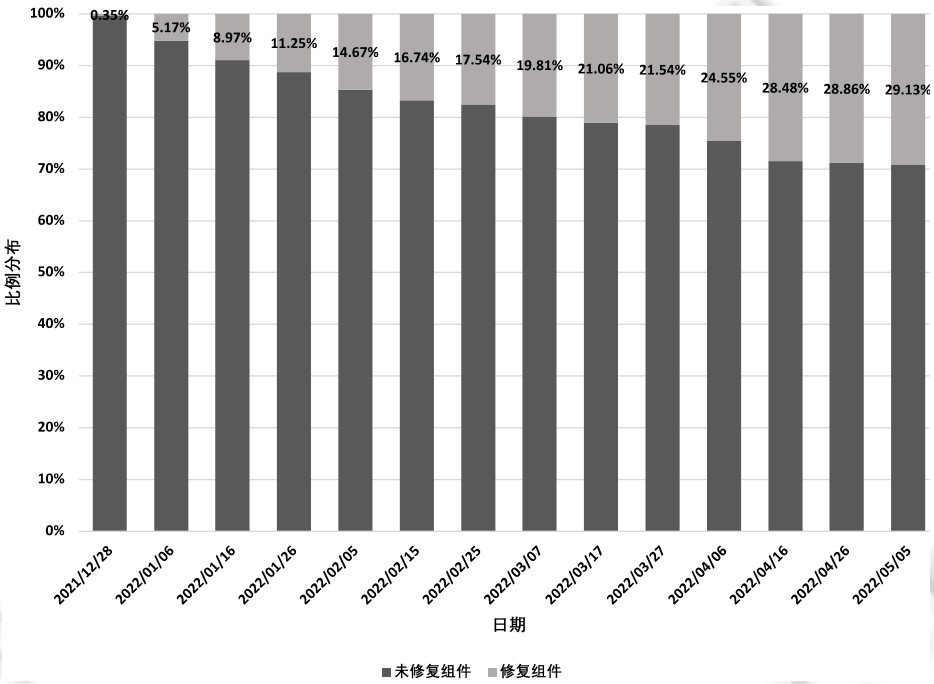

图 5 log4j-core 直接依赖组件版本修复比例对比图

通过对比图 5 中不同时期修复版本的依赖比例可以发现: 随着时间的推移, 黑色所标识的受漏洞影响组件占比不断缩小, 虽然修复组件的占比(浅色部分)在不断增加, 但占比依然较小, 绝大部分组件依然持续受到漏洞影响. 截止 2022 年 5 月 5 日, log4j-core 直接依赖组件的修复比例仅为 29.13%. 说明虽然软件供应链安全日益被开发者所关注, 并在漏洞爆出后的短时间内发布了修复版本, 但是由于依赖关系的复杂性, 漏洞依然会大量且长久地存在于生态中.

5 相关工作

5.1 生态依赖关系分析

He 等人[4]对 Java 生态中的依赖组件迁移现象进行了研究, 他们将项目更改依赖组件的现象称作库迁移. 作者共分析了 GitHub 中 19 652 个 Java 项目, 总结出了 14 个常见的迁移原因. Wang 等人[5]研究了 Java 生态系统中第三方库的使用和更新问题, 他们选取了 GitHub 中 806 个开源项目和 13 565 个第三方库. 首先, 作者以基于 API 调用的粒度对第三方库的使用情况进行了量化分析, 发现绝大多数项目都使用了过时的库; 之后, 他们对项目的 Commit 信息进行了语义分析, 发现有一半的项目所使用的半数库从未更新过. Soto-Valero 等

人[6,7]在 2021 年研究了 Java 生态中的臃肿依赖项, 他们将依赖列表中不会用到的第三方组件称作臃肿依赖项. 作者首先利用组件的配置信息获取其依赖列表, 再利用 ASM 工具对类文件的字节码进行细粒度分析以判断是否被调用, 最后将臃肿项从依赖列表中移除. 他们实现了一个名为 DepClean 的工具, 用于自动分析和删除 Java 项目中的臃肿依赖项. Bavota 等人[8]对 Apache 社区中的 147 个 Java 项目进行了深入研究, 分析了项目之间依赖关系的变化规律和原因. 他们发现, 项目大小、代码行修改数量和 Bug 的修复数量对依赖关系均存在不同程度的影响. 相似地, Kula 等人[9]针对从 GitHub 收集的 4 600 个 Java 项目研究了开发人员更新依赖项的行为, 他们发现: 开发人员不仅严重依赖于第三方库, 并且很少升级依赖项, 偏好依赖于流行的旧版本. 可以看出: 之前的工作多集中于对有限的项目进行分析, 而本文的研究是基于海量开源组件对 Java 生态的整体性分析.

此外, Benelallam 等人[10]根据 2018 年 9 月 6 日的 Maven Central 数据, 以每个组件版本的三元组“GroupId: ArtifactId:Version”为唯一标识, 构建了一个图数据库, 并将其开源. 他们首先通过 Maven 的官方索引获取每个组件版本的标识, 再根据标识获取其配置文件, 最后根据配置文件中依赖信息在 Neo4j 数据库中构建了 Maven 的依赖网络.

除 Java 以外, 还有研究人员对其他语言生态也进行了系统的分析. Decan 等人[11−13]对多个生态系统进行了比较研究, 其中, 文献[13]使用 Libraries.io 数据集对 7 个不同规模的语言生态进行了演变规律分析, 作者提出了 3 个通用指标来定量地对比这些语言生态在数量增长、组件复用和依赖特征上的差异. 他们最后发现: 依赖网络的规模和组件的数量都会随着时间的推移而增长, 而少数组件的更新频率远高于大部分组件. Decan 等人[14]研究了 4 个不同生态系统(Cargo、npm、Packagist 和 Rubygems)中的组件是否遵守语义化版本控制规范的问题, 发现生态中的多数组件会遵守该规范. 类似地, Kikas 等人[15]研究了 3 个生态系统(Cargo、npm 和 RubyGems)的依赖网络, 他们发现: 个别关键组件的修改, 可能会对生态系统中不少于 30%的组件产生影响. 还有一些针对语言生态的研究工作从其他切入点展开, 如: Stringer 等人[16]研究了语言生态中的更新滞后现象, Abate[17]等人研究了不同组件管理器的依赖解析方案.

相比于我们的研究, 目前已有关于 Java 生态系统分析的工作大多只关注了有限的研究对象, 更多的是对整个生态系统的一般性统计. 相反, 我们对 Java 生态系统中的开源软件包进行了大规模的研究, 并基于依赖关系对关键组件的影响力进行了分析.

5.2 漏洞传播分析

DüsingJohannes 等人[18]结合 Snyk 漏洞库对 NuGet、NPM Registry 和 Maven-Central 库中漏洞的直接和传递依赖关系进行了分析. 他们借助 Neo4j 构建了生态依赖图, 并将漏洞信息作为实体对象添加到数据库中, 最后, 利用数据库的查询语法分析了修复漏洞的比例、补丁相对漏洞的发布延迟和依旧依赖漏洞版本的组件分布. 类似地, 陈晨[19]结合 CVE 漏洞库和 GitHub 中的工程信息在 Neo4j 数据库中构建了 Maven-Central 的漏洞检测知识图谱, 包含了漏洞、GitHub 信息、组件版本这 3 种实体类型, 可以根据输入的组件版本名称或上传的工程压缩包来检测目标是否包含漏洞. 但是由于采用的是 Maven 官方接口, 无法拓展到超大规模的数据集上进行分析. Pashchenko 等人[20]提出了一种通过分析依赖关系来高精度地检测漏洞传播的方法, 并对 Maven 中 25 767 个组件进行了分析. 他们发现, 80%的漏洞依赖项可以按照升级直接依赖项的方式修复. 作者的检测方法相较于传统方法减少了 27%的误报数量. 除通过依赖关系分析漏洞传播的研究工作以外, Ponta 等人[21]提出了一种无需元数据文件就能检测开源软件漏洞的技术, 该技术能够以动态和静态分析结合的方式检测目标程序中是否(直接/间接)引用了存在漏洞的组件. Prana 等人[22]在 GitHub 上选取了 450 个 Java、Python 和 Ruby 编写的软件项目, 利用 Veracode 软件组成分析工具分析了这些项目所使用的开源库组件所包含的漏洞. 他们发现: 最普遍的漏洞类型是拒绝服务和信息泄露, 并且通常需要 3–5 个月才能修复漏洞. 与本文研究相比, 现有的漏洞传播分析多关注有限的研究对象, 更多地关注漏洞的一般统计数据. 相反, 本文对 Java 开源组件生态系统的所有组件进行了大规模研究, 并且揭示了漏洞是如何通过依赖链对生态造成巨大影响的.

除与 Java 生态有关的漏洞分析工作以外, Lauinger 等人[23]研究了 JavaScript 语言生态中的组件使用情况,

发现所检查的 13.3 万个网站中有 37%的网站使用了包含已知漏洞的第三方组件. Zimmermann 等人[24]研究了 npm 生态中的组件依赖关系, 发现单个组件包可能会对整个生态中的大部分组件产生影响, 并且极少数的维护者账户可被用于向大量组件中注入恶意代码. Liu 等人[25]针对 npm 生态提出了一种基于知识图谱的精准依赖关系解析方法, 将组件的依赖关系解析为依赖树, 研究了漏洞的传播和演化规律. Ohm 等人[26]提出了一种针对供应链中通过恶意组件进行攻击的动态分析检测方法, 通过监测组件的构建过程来分析是否引入了恶意组件. Gkortzis 等人[27]研究了 1 244 个开源项目, 发现较高数量的依赖和漏洞之间存在着很强的相关性, 代码重用具有两面性, 既不能用于解决漏洞, 也不是导致漏洞数量增加的罪魁祸首.

总体而言, 当前工作多是利用开源的组件数据或者利用动态方法来检测对象组件是否受漏洞影响, 存在数据过时或效率较低等问题. 本文的工作基于自建的依赖解析器, 首次针对 Java 生态组件进行了大规模的依赖关系提取, 在依赖关系之上分析漏洞传播的规律, 并给出了传播路径的关键节点用于指导漏洞修复与检测.

6 结束语

本文针对 Java 语言生态软件供应链安全进行了研究与分析. 为了准确分析及量化漏洞组件在生态中的影响力, 本文首次对组件依赖关系和漏洞组件影响力等重要指标提出了形式化定义, 用于指导漏洞组件生态影响力以及修复情况的分析. 为了支持基于依赖关系的漏洞组件分析, 本文设计并实现了依赖关系提取解析框架, 对 880 万个第三方组件的依赖关系进行了提取, 得到 6 500 万条直接依赖关系, 准确度达到 94.64%. 最终, 根据上述定义的重要指标以及全生态组件依赖关系数据, 以 Log4j2 漏洞为例, 对该漏洞的生态影响力以及修复情况进行了分析. 结果表明, 15.12%的组件受到该漏洞的影响. 为了研究漏洞传播链上组件的分布规律, 对受影响组件按照层级划分, 发现多数位于 5 层以内, 漏洞触发可能性较大; 同时, 对传播链上的关键组件进行了定位. 在漏洞修复方面, 虽然受影响组件的修复比例在逐步扩大, 但整体的修复仅为 29.13%, 修复情况不容乐观. 未来将对因间接依赖关系被漏洞影响组件的修复方式进行深入研究, 以探索出更高效快捷的修复方式. 此外, 未来还将把本文工作扩展到 Rust、Go 等其他主流语言的开源生态, 并对其结果进行横向对比分析.

References:

[1] Rep-ossra-2022. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf

[2] Maven-Central. https://repo1.maven.org/maven2/

[3] State of the software supply chain report. 2021. https://www.sonatype.com/resources/white-paper-2021-state-of-the-software-supply-chain-report-2021

[4] He H, He R, Gu H, Zhou M. A large-scale empirical study on Java library migrations: Prevalence, trends, and rationales. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. 2021. 478−490. [doi: 10.1145/3468264.3468571]

[5] Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Wu Y, Liu Y. An empirical study of usages, updates and risks of third-party libraries in Java projects. In: Proc. of the 2020 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). 2020. 35−45. [doi: 10.1109/ICSME46990.2020.00014]

[6] Soto-Valero C, Harrand N, Monperrus M, Baudry B. A comprehensive study of bloated dependencies in the Maven ecosystem. Empirical Software Engineering, 2021, 26(3): 1−44. [doi: 10.1007/s10664-020-09914-8]

[7] Soto-Valero C, Durieux T, Baudry B. A longitudinal analysis of bloated Java dependencies. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. 2021. 1021−1031. [doi: 10.1145/3468264.3468589]

[8] Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S. How the apache community upgrades dependencies: An evolutionary study. Empirical Software Engineering, 2015, 20(5): 1275−1317. [doi: 10.1007/s10664-014-9325-9]

[9] Kula RG, German DM, Ouni A, Ishio T, Inoue K. Do developers update their library dependencies. Empirical Software Engineering, 2018, 23(1): 384−417. [doi: 10.1007/s10664-017-9521-5]

[10] Benelallam A, Harrand N, Soto-Valero C, Baudry B, Barais O. The Maven dependency graph: A temporal graph-based representation of Maven Central. In: Proc. of the 16th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). 2019. 344−348. [doi: 10.1109/MSR.2019.00060]

[11] Decan A, Mens T, Claes M. On the topology of package dependency networks: A comparison of three programming language ecosystems. In: Proc. of the 10th European Conf. on Software Architecture Workshops. 2016. 1−4. [doi: 10.1145/2993412.3003382]

[12] Decan A, Mens T, Claes M. An empirical comparison of dependency issues in OSS packaging ecosystems. In: Proc. of the 24th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). 2017. 2−12. [doi: 10.1109/SANER.2017.7884604]

[13] Decan A, Mens T, Grosjean P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering, 2019, 24(1): 381−416. [doi: 10.1007/s10664-017-9589-y]

[14] Decan A, Mens T. What do package dependencies tell us about semantic versioning. IEEE Trans. on Software Engineering, 2019, 47(6): 1226−40. [doi: 10.1109/TSE.2019.2918315]

[15] Kikas R, Gousios G, Dumas M, Pfahl D. Structure and evolution of package dependency networks. In: Proc. of the 14th IEEE/ACM Int'l Conf. on Mining Software Repositories (MSR). 2017. 102−112. [doi: 10.1109/MSR.2017.55]

[16] Stringer J, Tahir A, Blincoe K, Dietrich J. Technical lag of dependencies in major package managers. In: Proc. of the 27th Asia-Pacific Software Engineering Conf. (APSEC). 2020. 228−237. [doi: 10.1109/APSEC51365.2020.00031]

[17] Abate P, Di Cosmo R, Gousios G, Zacchiroli S. Dependency solving is still hard, but we are getting better at it. In: Proc. of the 27th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). 2020. 547−551. [doi: 10.1109/SANER48275.2020.9054837]

[18] Düsing J, Hermann B. Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. In: Proc. of the Digital Threats: Research and Practice. 2021. [doi: 10.1145/3472811]

[19] Chen C. Design and implementation of vulnerability detection system based on knowledge graph [Ph.D. Thesis]. Beijing: Beijing University of Posts and Telecommunications, 2021 (in Chinese with English abstract). [doi: 10.26969/d.cnki.gbydu.2021.001167]

[20] Pashchenko I, Plate H, Ponta SE, Sabetta A, Massacci F. Vuln4real: A methodology for counting actually vulnerable dependencies. IEEE Trans. on Software Engineering, 2020, 1. [doi: 10.1109/TSE.2020.3025443]

[21] Ponta SE, Plate H, Sabetta A. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). 2018. 449−460. [doi: 10.1109/ICSME.2018.00054]

[22] Prana GA, Sharma A, Shar LK, Foo D, Santosa AE, Sharma A, Lo D. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. Empirical Software Engineering, 2021, 26(4): 1−34. [doi: 10.1007/s10664-021-09959-3]

[23] Lauinger T, Chaabane A, Arshad S, *et al*. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the Web. In: Proc. of the Network and Distributed System Security Symp. 2017. [doi: 10.48550/arXiv.1811.00918]

[24] Zimmermann M, Staicu CA, Tenny C, Pradel M. Small world with high risks: A study of security threats in the NPM ecosystem. In: Proc. of the 28th USENIX Security Symp. (USENIX Security 2019). 2019. 995−1010. [doi: 10.48550/arXiv.1902.09217]

[25] Liu C, Chen S, Fan L, Chen B, Liu Y, Peng X. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE 2022). New York: Association for Computing Machinery, 2022. 672−684. [doi: 10.48550/arXiv.2201.03981]

[26] Ohm M, Sykosch A, Meier M. Towards detection of software supply chain attacks by forensic artifacts. In: Proc. of the 15th Int'l Conf. on Availability, Reliability and Security. 2020. 1−6. [doi: 10.1145/3407023.3409183]

[27] Gkortzis A, Feitosa D, Spinellis D. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. Journal of Systems and Software, 2021, 172: 110653. [doi: 10.1016/j.jss.2020.110653]

[28] NVD CVE-2021-44228. 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-44228

[29] Information note on the Apache Log4j2 vulnerability in the open source community (in Chinese). https://mp.weixin.qq.com/s/dWublxXRE2NHae7SRXUuiA

[30] Cybersecurity risk alert on Apache Log4j2 component critical security vulnerability (in Chinese). https://wap.miit.gov.cn/jgsj/waj/gzdt/art/2021/art_d0cd32999d9941209ba9358a2e62638c.html

[31] Log4j Download. https://www.sonatype.com/resources/log4j-vulnerability-resource-center

[32] Maven. https://maven.apache.org/

[33] Introduction to the POM. https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#:~:text=Available%20Variables-,What%20is%20a%20POM%3F,default%20values%20for%20most%20projects

[34] Maven Central index. https://maven.apache.org/repository/central-index.html

[35] Libraries.io. https://libraries.io/

[36] Maven effective pom. https://maven.apache.org/plugins/maven-help-plugin/effective-pom-mojo.html

[37] Apache Log4j security vulnerabilities. https://logging.apache.org/log4j/2.x/security.html

附中文参考文献:

[19] 陈晨. 基于知识图谱的漏洞检测系统的设计与实现 [硕士学位论文]. 北京: 北京邮电大学, 2021. [doi: 10.26969/d.cnki.gbydu.2021.001167]

[29] 关于开源社区 Apache log4j2 漏洞情况的说明. https://mp.weixin.qq.com/s/dWublxXRE2NHae7SRXUuiA

[30] 关于阿帕奇 Log4j2 组件重大安全漏洞的网络安全风险提示. https://wap.miit.gov.cn/jgsj/waj/gzdt/art/2021/art_d0cd32999d9941209ba9358a2e62638c.html

毛天宇(1992－), 男, 硕士生, 主要研究领域为软件供应链安全.

申文博(1989－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为软件供应链安全, 操作系统安全, 云原生系统安全.

王星宇(1993－), 男, 硕士生, 主要研究领域为软件供应链安全.

任奎(1978－), 男, 博士, 教授, 博士生导师, CCF 会士, ACM 会士, IEEE 会士, 主要研究领域为数据安全, 人工智能安全, 物联网安全, 隐私保护.

常瑞(1981－), 女, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为系统安全, 软件供应链安全, 程序分析, 形式化验证.