软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2023,34(12):5552−5577 [doi: 10.13328/j.cnki.jos.006763]

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

# 基于双层 BiLSTM 的安装程序 DLL 劫持漏洞挖掘方法*

陈 霄[1,2], 肖 甫[1,2], 沙乐天[1,2], 王 众[3], 底伟鹤[1,2]

[1](南京邮电大学 计算机学院, 江苏 南京 210023)

[2](江苏省无线传感网高技术研究重点实验室, 江苏 南京 210023)

[3](字节跳动 无恒实验室, 浙江 杭州 311100)

通信作者: 肖甫, E-mail: xiaof@njupt.edu.cn

**摘 要**: 动态链接库 (dynamic link library, DLL) 的出现给开发人员提供了极大的便利, 也提高了操作系统与应用程序之间的交互性. 然而, 动态链接库本身存在的安全性隐患不容忽视, 如何有效地挖掘 Windows 平台下安装程序执行过程中出现的 DLL 劫持漏洞是当下保障 Windows 操作系统安全的关键问题之一. 搜集并提取大量安装程序的属性特征, 从安装程序、安装程序调用 DLL 模式、DLL 文件本身 3 个角度出发, 使用双层 BiLSTM (bi-directional long short-term memory) 神经网络进行学习, 抽取出漏洞数据集的多维特征, 挖掘 DLL 劫持未知漏洞. 实验可有效检测 Windows 平台下安装程序的 DLL 劫持漏洞, 共挖掘 10 个未知漏洞并获得 CNVD 漏洞授权, 此外通过和其他漏洞分析工具进行对比进一步验证该方法的有效性和完整性.

**关键词**: 漏洞挖掘; 神经网络; 动态链接库

**中图法分类号**: TP311

中文引用格式: 陈霄, 肖甫, 沙乐天, 王众, 底伟鹤. 基于双层BiLSTM的安装程序DLL劫持漏洞挖掘方法. 软件学报, 2023, 34(12): 5552–5577. http://www.jos.org.cn/1000-9825/6763.htm

英文引用格式: Chen X, Xiao F, Sha LT, Wang Z, Di WH. DLL-hijacking Vulnerability Mining in Installers Based on Double-layer BiLSTM. Ruan Jian Xue Bao/Journal of Software, 2023, 34(12): 5552–5577 (in Chinese). http://www.jos.org.cn/1000-9825/6763.htm

## DLL-hijacking Vulnerability Mining in Installers Based on Double-layer BiLSTM

CHEN Xiao[1,2], XIAO Fu[1,2], SHA Le-Tian[1,2], WANG Zhong[3], DI Wei-He[1,2]

[1](School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China)

[2](Jiangsu High Technology Research Key Laboratory for Wireless Sensor Networks, Nanjing 210023, China)

[3](Wuheng Lab, ByteDance, Hangzhou 311100, China)

**Abstract**: The emergence of the dynamic link library (DLL) provides great convenience for developers, which improves the interaction between the operating system (OS) and applications. However, the potential security problems of DLL cannot be ignored. Determining how to mine DLL-hijacking vulnerabilities during the running of Windows installers is important to ensure the security of Windows OS. In this paper, the attribute features of numerous installers are collected and extracted, and the double-layer bi-directional long short-term memory (BiLSTM) neural network is applied for machine learning from the perspectives of installers, the invocation modes of DLL from installers, and the DLL file itself. The multi-dimensional features of the vulnerability data set are extracted, and unknown DLL-hijacking vulnerabilities are mined. In experiments, DLL-hijacking vulnerabilities can be effectively detected from Windows installers, and 10 unknown vulnerabilities are discovered and assigned CNVD authorizations. In addition, the effectiveness and integrity of this method are further verified by comparison with other vulnerability analyzers.

**Key words**: vulnerability mining; neural network; dynamic link library (DLL)

---

* 基金项目: 国家重点研发计划 (2018YFB0803400); 国家杰出青年科学基金 (62125203); 国家自然科学基金面上项目 (62072253)
收稿时间: 2021-12-05; 修改时间: 2022-03-12, 2022-06-26; 采用时间: 2022-08-19; jos 在线出版时间: 2023-03-02
CNKI 网络首发时间: 2023-03-03

随着个人电脑的快速普及, Windows 操作系统迅速占领市场, 总体份额也在不断提升. 然而, 其自身设计所存在的安全问题也随之不断凸显, 威胁系统和用户的安全. 作为 Windows 系统的核心组件之一, 动态链接库 (dynamic link library, DLL) 用于提高软件开发人员的软件编写水平和软件运行效率, 不仅可以提高内存使用效率, 也可以实现对关键技术的保护, 但也引入了较大的攻击面.

目前针对动态链接库的攻击主要分为 DLL 注入、DLL 劫持、DLL 侧加载等, 以上攻击行为主要利用 DLL 机制存在跨进程加载、优先搜索应用程序目录以及缺少验证等缺陷, 对应用程序和操作系统造成破坏, 严重威胁程序和系统安全. 其中 DLL 注入本来是软件用于向其他程序添加扩展功能、调试或逆向工程的一种合法技术[1], 但是经常被恶意软件开发者利用, 进行恶意攻击行为[2]. DLL 侧加载则是通常所说的“白加黑”技术, 利用一个正常的 EXE 程序 (白程序) 加载一个恶意的 DLL 文件 (黑文件). DLL 劫持是指如果进程尝试加载一个 DLL 时没有指定 DLL 的绝对路径, 则系统会尝试去指定的目录下查找这个 DLL. 当攻击者把恶意的 DLL 文件放在优先于正常 DLL 所在的目录时, 就能够欺骗系统优先加载恶意 DLL, 从而实现“劫持”效果, 导致恶意代码执行. 这就是 DLL 劫持漏洞的成因[3]. DLL 劫持漏洞自 2010 年被微软披露以来, 一直屡禁不止, 国际主流的漏洞收录平台通用漏洞披露 (common vulnerabilities and exposures, CVE) 上, 共收录了 350 个和 DLL 劫持相关的漏洞信息, 近 5 年收录数量占比达到了 45.14%. 与此同时, 国内主流的漏洞收录平台国家信息安全漏洞共享平台 (China national vulnerability database, CNVD) 上, 共收录了 637 条涉及到 DLL 劫持的漏洞信息, 近 5 年收录数量的占比高达 96.86%. 由此可见, DLL 劫持漏洞虽然起源较早, 但依旧无法忽视, 仍然是目前亟需解决的漏洞之一. 这些 DLL 劫持漏洞几乎涉及到所有类型的应用程序, 包括音频、图片、文本等, 相对于 DLL 注入和 DLL 侧加载, DLL 劫持漏洞的数量更多, 危害更大, 影响范围更广.

根据 DLL 劫持产生的方式可将其划分为以下几种, 如图 1 所示.

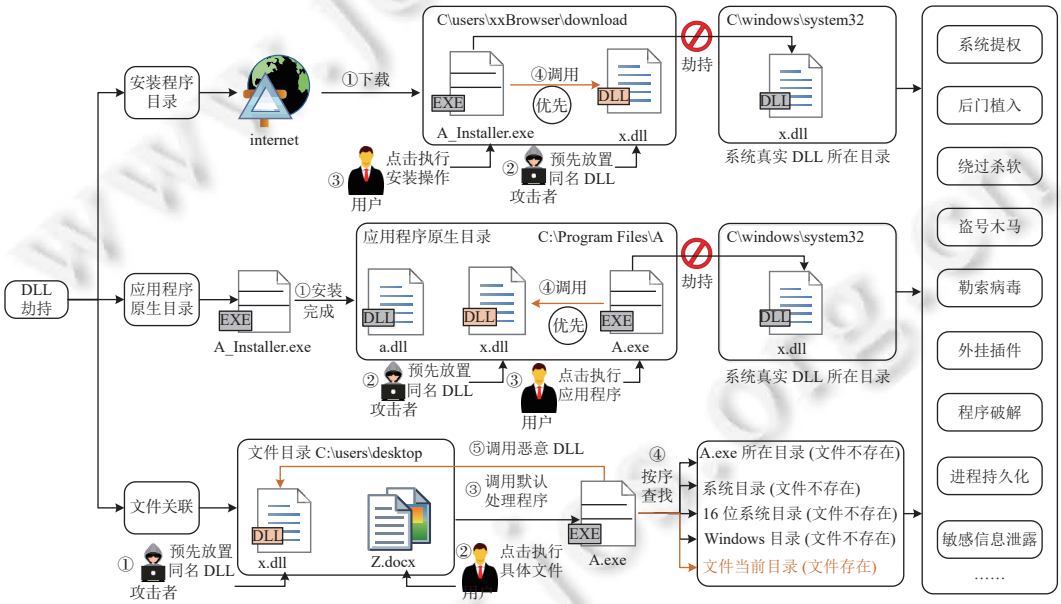



图 1 DLL 劫持 3 种方式

第一, 针对安装程序进行 DLL 劫持攻击. 安装程序即应用程序的安装软件, 用来在主机上执行安装操作. 通过网络下载安装程序时, 用户通常不会修改其所在目录, 攻击者可以在获取到该目录的前提下, 伪造恶意 DLL 文件供安装程序在安装时进行调用, 从而实现恶意 DLL 加载, 攻击系统安全. 第二, 针对应用程序原生目录进行 DLL 劫持攻击. 应用程序安装包在执行完安装操作后, 会在主机上的某个目录下存放所有的文件, 该目录即为原生目录. 原生目录中不仅有该应用程序可执行的文件, 也包括了程序正常执行过程中所需的 DLL 文件. 应用程序在运行时会找寻特定名称的 DLL 文件, 而攻击者可以通过伪造一个恶意的同名 DLL 文件并释放到原生目录, 实现

DLL 劫持攻击, 从而制作程序外挂或者进行免杀. 第三, 针对文件关联进行 DLL 劫持攻击. 操作系统上各种类型文件都有一个与之关联的默认处理软件. 当某个文件被打开时, 操作系统会自动创建一个进程来处理该文件, 进程对应的程序就是该文件类型关联的默认处理程序, 进程的当前目录就是被打开文件所在的目录. 如果进程在处理该文件的过程中, 需要加载一个系统中并不存在的 DLL 文件, 例如执行特定扩展功能的 DLL, 则攻击者可以通过伪造该恶意的 DLL 文件到音视频、文本等文件所在目录, 进程最终会搜索到文件所在目录, 进而加载该恶意 DLL, 形成 DLL 劫持. 3 种 DLL 劫持方式的差异如表 1 所示.

表 1　3 种 DLL 劫持方式对比

| 劫持方式 | 目标对象 | 攻击前提 | 攻击方式 | 限制条件 | 触发难度 |
|---|---|---|---|---|---|
| 安装程序型 | 网络下载的安装程序 | 获取安装程序所在目录进行读写 | 预先放置同名恶意DLL到下载目录中, 执行安装时进行劫持 | 安装程序和DLL在同一目录 | 低, 下载目录通常已知且权限低 |
| 应用程序原生目录型 | 应用程序安装后所在原生目录 | 获取原生目录并进行读写 | 预先放置同名恶意DLL文件到原生目录中, 程序执行时进行劫持 | 通常为系统级目录 | 高, 原生目录未知且权限高 |
| 文件关联型 | 音视频、文本等常见文件 | 提前了解文件默认打开方式; 确保进程目录、系统目录、Windows目录下都不存在该DLL文件; 获取文件所在目录进行读写 | 预先放置同名恶意DLL文件到文件所在目录, 默认处理进程执行时搜索该DLL, 最终执行当前目录下的恶意DLL | 部分系统环境下不存在特定DLL文件 | 高, 文件默认打开方式未知且环境不可控 |

从表 1 可以看出, 第 2 种方式实现过程中限制条件较多, 鉴于大部分程序默认安装到系统级目录下, 无法轻易对其进行操作. 第 3 种方式则需要首先获取文件默认打开方式, 找到处理文件的进程, 加入恶意 DLL 的同时不能被识别, 而且通常需要特定环境的支持. 如 dwmapi.dll 文件在 Windows XP 环境下不存在, 只在 Vista 以上版本存在, 导致无法在 Windows XP 环境下劫持 dwmapi.dll 文件, 因此该种方式实施难度也较大. 第 1 种针对安装程序的 DLL 劫持攻击, 由于安装程序的默认下载目录通常是已知的, 并且权限相对较低, 只需要在安装程序目录下放置恶意 DLL 文件, 因此难于检测. 综上, 第 1 种劫持方式更加简单, 隐蔽性更强, 并且可以在更多场景下实施, 因此是攻击者所利用的主要漏洞类型, 且经常与 Edge、Chrome 的自动下载漏洞相结合, 威胁系统安全.

如图 2 所示, 以 CVE-2019-1367 为例, 该漏洞为 IE 浏览器的远程代码执行漏洞, 攻击者利用该漏洞, 能够实现任意代码执行, 也可以打开浏览器的自动下载功能, 诱导用户下载一个恶意的 DLL 文件. 对于 Edge 浏览器来说, 其默认的下载目录为 C:\Users\<Username>\Downloads, 通过 Edge 下载的文件都会默认保存在这个目录下. 自动下载漏洞下载的恶意 DLL 文件会保存到该目录下, 然后利用浏览器的页面超时跳转功能可以让 Edge 跳转到正常页面来下载一个正常的安装程序. 当用户运行正常的安装程序时, 同目录下恶意的 DLL 文件就会被自动加载. 而恶意的 DLL 文件中已经被攻击者加入了不同类型的攻击代码, 例如后门木马、权限获取等. 用户无法发现恶意 DLL 在后台所进行的操作, 而攻击者可以安装程序、查看、更改或删除数据, 或创建具有完全用户权限的新帐户, 也可以在特定位置植入后门, 同时能够对抗系统重启, 最终将系统中数据, 包括网络流量、隐私数据等都上传到攻击者手中, 攻击者还可以获取目标系统的最高操作权限, 进行其他恶意攻击.

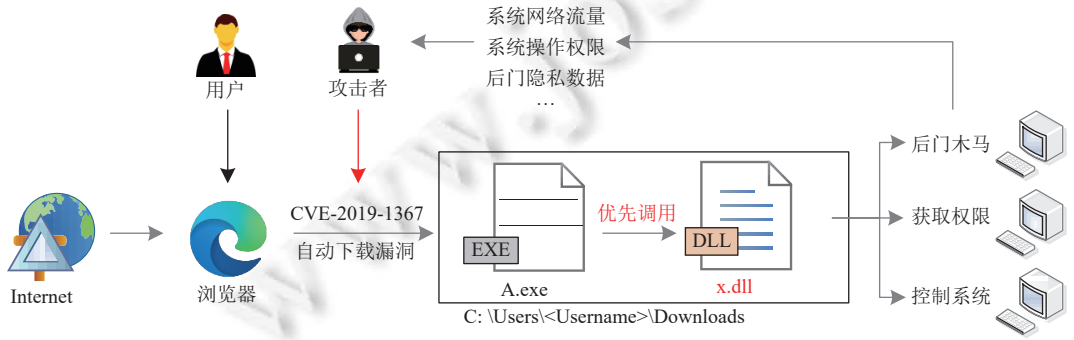



图 2　安装程序 DLL 劫持漏洞攻击示意图

然而, 现有的 DLL 安全研究很少有直接针对 DLL 劫持漏洞的挖掘与分析, 相关工作主要集中在以下两个方面: 一方面是通过代码插桩、API Hook 等技术实现对 DLL 注入、DLL 侧加载等攻击的防御; 另一方面则主要是通过监控 DLL 文件的调用或者验证 DLL 文件的合法性进行恶意软件的检测. 虽然 DLL 劫持漏洞由来已久, 但是相关工作呈现零散化的状况, 真正针对该漏洞进行挖掘分析的研究较少.

工业界相关工具根据 DLL 劫持漏洞的原理, 通过动态分析来获得程序运行时加载的 DLL 文件列表, 进而判断目标 DLL 能否被劫持. 该方法较为直观, 但是使用比较复杂, 并且只考虑 DLL 文件路径这一唯一属性, 漏洞覆盖面较窄, 效率较低且通用性不强. 尤其针对安装程序 DLL 劫持漏洞的挖掘不够深入, 粒度不够细致, 难以实现对大规模安装程序的高效率漏洞挖掘与安全分析.

安装程序 DLL 劫持型漏洞主要涉及到安装程序、调用 DLL 模式和 DLL 文件 3 个方面, 基于 DLL 文件路径只能覆盖一定范围漏洞, 而在其基础上加入安装程序自身的多维属性和调用 DLL 模式的不同维度, 能够使得对漏洞研究对象的描述粒度更加具体, 不仅可以覆盖之前单纯基于 DLL 文件路径属性所挖掘的漏洞, 而且能够利用神经网络来寻找研究对象 3 方面的多维属性与漏洞的深层次映射关系, 并且神经网络可以在多次迭代中自动调整每个属性的具体权重, 实现漏洞挖掘的全面性和高覆盖性. 此外, 存在漏洞的安装程序本身的一些属性也存在共性信息, 单纯考虑 DLL 路径可能会缺失一些漏洞, 而基于已知漏洞可以分析出不同属性的安装程序执行安装所调用的 DLL 文件大体范围与种类, 相对于已有工具, 能够节省大量时间, 效率更高.

因此, 基于上述分析, 目前亟待提出一种针对 Windows 平台下安装程序 DLL 劫持的漏洞挖掘与分析方法, 从安装程序、调用模式以及 DLL 文件的属性角度分析相关漏洞与缺陷, 为安装程序的安全加固及 DLL 劫持漏洞的高效防御提供研究依据.

本文实现了一种基于双层双向长短时记忆网络 (bi-directional long short-term memory, BiLSTM) 的 Windows 平台安装程序 DLL 劫持的漏洞挖掘及验证系统 BiL-DLLMining. 通过对目前已知存在 DLL 劫持漏洞的 EXE 文件和尚未挖掘到漏洞信息的 EXE 文件进行分析, 从静态和动态的角度抽取属性特征, 同时对属性进行归一化, 使得各属性类型一致. 建立双层 BiLSTM 神经网络, 训练所有 EXE 文件的自然属性与漏洞情况的映射关系, 获取存在劫持漏洞的 EXE 文件属性特征集. 训练存在漏洞的 EXE 文件属性与 DLL 调用的映射关系, 获取可劫持 DLL 属性特征集, 最终综合两个特征集, 挖掘出若干未知的 DLL 劫持漏洞, 为 Windows 平台安装程序与动态链接库的安全风险评估提供重要的技术支撑.

本文第 1 节对相关研究工作进行介绍. 第 2 节简单分析一下 DLL 劫持漏洞的原理. 第 3 节给出关于 EXE 文件和 DLL 文件的相关属性集合, 并且对 EXE 文件的保护机制给出判定规则. 第 4 节详细介绍本文提出的基于双层 BiLSTM 的 DLL 劫持漏洞挖掘及验证系统 BiL-DLLMining, 主要包括安装程序漏洞特征判决模型和可劫持对象判决模型两部分. 第 5 节通过实验来分析 BiL-DLLMining 的效能. 第 6 节对工作进行总结, 并对不足之处进行分析和讨论.

## 1 相关研究

本文所提出的 DLL 劫持漏洞挖掘方法, 主要针对 Windows 平台的安装程序和其调用的 DLL 文件进行属性抽取与调用关系映射分析, 所采用的神经网络为双层 BiLSTM 网络. 其中安装程序为 EXE 文件, 即可执行文件 (executable file), 而神经网络不仅在图像分类、人脸识别等领域有所应用, 在入侵检测、系统漏洞挖掘等安全方面也应用广泛. 因此相关研究工作主要包括以下几个方面: 第一, 恶意可执行文件分析与检测; 第二, 动态链接库安全研究; 第三, 基于神经网络的攻击检测与漏洞挖掘.

### 1.1 恶意可执行文件分析与检测

对于恶意可执行文件的分析与研究, 主流的方法可以分为动态/静态分析和机器学习两大类. 具体来看, 动态与静态分析主要从可执行文件的代码属性特征、控制流程图、网络流量交互等方面入手, 建立恶意可执行文件判决规则. 文献 [4] 提出了一种用于二进制可执行文件污点分析的静态分析工具框架, 通过崩溃分析器、污点读取器和代码跟踪器获取二进制可执行文件的污点分析结果, 绘制可执行文件的控制流程图, 从中检测出恶意可执行文件. 与文献 [4] 类似, 文献 [5] 开发了一个静态的恶意软件分析工具 PEFAF. 不同点在于, 文献 [5] 并非从控制流

程图入手, 而是对大量的可执行文件进行特征提取, 使用数据挖掘技术分析文件相关属性值、范围和频率等, 选取出恶意可执行文件的关键特征, 并且定义 34 个检测规则, 在 7000 个恶意文件集中实现了 98.16% 的高检测率. 文献 [6] 则利用 REMnux 操作系统上的各种工具对可执行文件进行测试和分析, 通过检测异常数据、文件完整性元数据、节熵以及可执行文件将执行的功能等方面, 成功检测出恶意可执行文件中存在的压缩加密或木马的情况, 表明静态分析在恶意文件检测方面的重要作用. 文献 [7] 通过对可执行文件进行反编译, 抽取影响文件安全性的特征, 建立行为集合, 并使用 BP 神经网络训练得到各个行为的概率分配函数, 使用加权和形式的合成法则对行为进行合成, 实现对文件恶意性的判定. 此外, 基于语义、子图相似性以及数据特征的多种恶意软件检测方法也被提出, 从不同的角度对恶意文件进行分析与研究[8−10].

使用机器学习相关算法进行恶意可执行文件的检测, 主要利用不同算法对于可执行文件关键特征的高效分析能力, 可以大幅度提高传统检测的效率与准确率. 文献 [11] 从 4 种不同类型的 PE 文件的元数据、导入 DLL 和函数中提取 28 个特征, 选取随机森林算法来检测网络上的恶意可移植可执行文件, 最终实现 98.7% 的检测率, 1.8% 的误检率. 文献 [12] 通过分析可执行文件的二进制代码来收集静态特征, 包括 PE 头信息和可打印字符串等, 同时在沙箱中收集动态特性, 如 API 调用日志等, 最终在对 SVM、朴素贝叶斯、J48 和随机森林 4 种分类器进行比较的基础上选择随机森林算法, 准确率达到了 97.2%. 文献 [13] 则从机器学习中的树结构出发, 采用 6 种不同的基于树的集成机器学习技术, 对 489 个恶意可执行文件和 700 个良性的可执行文件 PE 文件元数据, 计算出 54 个特征, 最终使用 voting ensemble classifier (VEC) 实现了对可执行文件 95% 的分类准确率. 类似地, 文献 [14] 和文献 [15] 分别使用支持向量机 (SVM) 和半监督学习来对恶意可执行文件进行识别与分类, 前者利用 SVM 在二分类问题上的高效性, 后者则利用最近距离法来判别可执行文件的安全属性.

总体而言, 对恶意可执行文件的研究一直围绕着文件自身的属性展开, 通过对关键属性的抽取, 动态、静态分析与机器学习算法相结合, 最终实现对不同平台下恶意可执行文件的高效分析与检测. 因此后续对于恶意可执行文件的研究以及可执行文件本身的缺陷检测, 可以继续从文件自身入手, 将自身属性作为研究的出发点.

### 1.2 动态链接库安全研究

目前有关动态链接库的安全研究主要聚焦在 DLL 注入、DLL 劫持、DLL 侧加载等攻击及防御技术的演进. 文献 [16] 将恶意软件检测方法分为 3 类: 基于静态功能、基于主机行为和基于网络行为. 针对恶意软件使用 DLL 注入技术来规避基于主机行为的恶意软件检测, 作者利用深度学习, 分析网络行为, 收集恶意软件通信数据, 实现对 DLL 注入等恶意行为的检测. 文献 [17] 则从恶意软件内存镜像的角度出发, 在沙箱环境中执行恶意软件样本, 从内存镜像中提取 DLL 文件进行特征分析, 同时加入机器学习的方法来识别恶意 DLL 和合法 DLL 的行为模式, 从而检测恶意注入的 DLL 文件, 但该方法无法检测恶意进程与相关驱动程序. 为了弥补这一缺点, 文献 [18] 对 DLL 文件进行流程分析, 加入了 hidden naive Bayes (HNB) 分类算法, 来解释高语义级别的恶意软件行为, 从而自动识别未知进程中的恶意软件进程. 文献 [19] 对 PE 文件 (portable executable file) 进行了细致研究, 通过构建 PE 文件的 DLL 依赖树展开静态逆向分析, 实现对 DLL 注入行为的检测, 弥补了现有基于签名的恶意软件检测产品效率较低等问题, 但是 DLL 依赖树的结构较为复杂, 该研究成果存在一定改进空间. 文献 [20] 通过分析 DLL 注入 Mozilla Firefox 浏览器引起 bug 的特征, 发现 90.3% 的 DLL 注入会导致崩溃产生, 原因在于第三方软件提供商不使用公共 API 作为软件扩展的手段, 作者建议建立一个公开可访问的验证测试框架来保证 DLL 注入的安全. 鉴于恶意软件在执行攻击行为时, 通常会调用一些特定的 DLL 文件, 如 version.dll、cldapi.dll 等, 文献 [21] 从该角度出发, 分析恶意软件加载频率较高的 DLL 文件, 建立通用 DLL 列表, 使用系统监视器对计算机上加载的 DLL 信息进行实时监视, 从而对比分析出可疑的 DLL 文件以及恶意攻击行为.

具体到 DLL 劫持相关安全研究, 文献 [22] 详细分析了 DLL 加载时可能出现的远程攻击, 包括地毯炸弹攻击、快捷方式与组件攻击以及文件与组件攻击, 并提出采用插桩方式对 DLL 加载时的系统调用信息进行监控, 从而动态发现存在的 DLL 劫持等漏洞. 但由于采用动态分析的原因, 所提方法在代码覆盖率上存在一定限制, 会漏判一些恶意 DLL 加载, 并且难以解决延迟加载的问题. 为了进一步解决这些问题, 文献 [23] 提出一种静态分析不安全 DLL 加载的技术, 通过对加载时调用点上下文进行切片和仿真, 获取所有可能的加载文件, 并对每一种可能性进

行安全分析, 从最终分析结果中寻找 DLL 劫持等安全漏洞. 该方法在多个主流的 Windows 应用程序上进行了测试, 因此可以作为动态检测的有效补充. 文献 [24] 在上述工作的基础上, 根据 DLL 劫持产生的原理, 从 DLL 搜索路径出发, 结合加载时的上下文信息, 精简了所需的监控点, 对内核函数 ZwQueryAttributesFile() 进行 hook 操作, 同时利用栈回溯技术过滤文件类型, 获取 DLL 加载时的路径信息, 从而分析可能出现的劫持漏洞. 文献 [25] 从交叉验证的角度出发, 基于代码签名和同源策略, 在动态组件加载过程中进行调用者和被调用者的交叉验证, 保证 DLL 加载的安全性, 扩展了 Windows 完整性机制, 达到对 DLL 注入、DLL 劫持等恶意行为防御的目的. 但该方法对操作系统的加载机制提出了更高要求, 具体实现上存在一定难度. 文献 [26] 则根据系统所加载组件的特性、植入组件与加载组件之间的相关性以及用户意图与组件加载操作之间的联系, 设计了 3 种组件加载安全策略, 同时构造一个系统 DLL 模块列表对将加载的 DLL 文件进行验证, 通过一个真实的 DLL 劫持漏洞来验证了安全策略的有效性. 文献 [27] 使用操作系统的回调机制来对加载的 DLL 文件进行监控和验证, 当操作系统发生 DLL 加载动作时, 会调用事先注册的系统回调函数, 回调函数将加载的 DLL 信息提交给 DLL 认证模块, 认证模块对加载的 DLL 文件进行认证, 根据认证结果做出相应决策. 通过对 5 款常用应用软件进行 DLL 劫持测试, 有效验证防御系统的检测效果, 但是该系统要求事先存储安全 DLL 文件的哈希值, 因此需要一定的工作量和存储代价.

综上所述, 目前对于 DLL 劫持漏洞的研究较少, 尚无针对该漏洞本身的挖掘与分析方法. DLL 劫持漏洞与 DLL 注入以及 DLL 侧加载相比, 存在着较大差异性, 针对 DLL 注入或者 DLL 侧加载等方面的研究不具备通用性, 无法准确移植到 DLL 劫持漏洞的挖掘上. 目前需要对 DLL 劫持漏洞投入更多研究, 保障系统的安全和稳定.

### 1.3 基于神经网络的攻击检测与漏洞挖掘

近些年来, 随着深度学习的快速发展, 越来越多的领域开始尝试使用深度学习的方法来解决分类及识别问题. 作为神经网络的一部分, 循环神经网络 (recurrent neural network, RNN) 与长短时记忆网络 (long short-term memory, LSTM) 能够处理任意输入序列长度, 建模多时间尺度的序列关系, 十分适合于学习和抽取序列数据中的特征和模式, 在自然语言处理、语音图像等多个领域均有非常广泛的应用.

目前使用 RNN、LSTM 等神经网络进行攻击检测和漏洞挖掘, 已经成为安全人员研究的重点和热点, 其中利用神经网络来提高网络入侵检测系统的检测率尤为突出. 文献 [28] 使用梯度提升决策树提取入侵检测数据集类别特征, 并进行重要性排序, 再利用随机森林对最优特征子集进行特征转换, 最终再使用梯度提升决策树进行训练和预测. 该模型在 UNSW-NB15 数据集上达到了 83.78% 的检测率, 误报率则为 1.8%. 文献 [29] 引入了循环神经网络来发现攻击中的复杂模式, 利用 RNN 生成新的攻击突变体, 利用长短时记忆网络生成多个多态蠕虫, 将现有的网络入侵检测系统 (NIDS) 检测率提高了 16.67%. 文献 [30] 利用卷积神经网络 (convolutional neural networks, CNN) 和长短时记忆网络的混合网络来提取网络流量数据的时空特征, 在 CICIDS2017 数据集上进行了实验, 总体准确率和 $F1$ 得分上均优于其他机器学习模型.

神经网络不仅仅被应用在提高入侵检测准确率方面, 还在其他方面发挥着重要作用. 文献 [31] 细致梳理了基于机器学习的软件漏洞挖掘的现有研究工作, 并从代码表征的形式角度出发, 对现有研究工作进行对比分析, 说明基于机器学习神经网络等方法进行漏洞挖掘研究, 已经成为安全领域的研究热点, 并且已经取得了一系列的成果. 在文献 [31] 的基础上, 文献 [32] 将程序源代码转换为代码属性图并进行切片, 选取神经网络中的双向 LSTM 和注意力机制进行训练, 最终在多个漏洞数据集上都取得了 77% 以上的 $F1$ 分数. 文献 [33] 同样使用 LSTM, 基于恶意工具加载 Windows 内置 DLL 的顺序具有一定特征这一规律, 提出了一种利用 LSTM 分析 DLL 信息的恶意工具检测方法, 基于 LSTM 对序列顺序的高效学习能力, 对 DLL 及其在各个进程中加载的顺序进行分析, 能够以 90% 以上的检测率检测出 4 种常见的恶意工具.

基于神经网络进行软件攻击检测和漏洞挖掘, 一直都是安全研究的热点, 但是目前利用神经网络进行动态链接库漏洞挖掘的研究匮乏, 需要更好地利用神经网络的特性, 发挥神经网络在动态链接库漏洞挖掘上的作用.

本文基于上述研究现状, 抽取 EXE 文件和 DLL 文件的属性特征, 生成属性特征序列, 利用 LSTM 神经网络在处理序列数据方面的优势, 构建双层 BiLSTM 神经网络, 对训练集进行学习, 并用测试集进行验证. 通过不断调整和优化相关参数, 挖掘 EXE 属性和可被劫持的 DLL 文件属性之间深层映射关系, 从而实现对安装程序漏洞特征

的判定, 进而通过神经网络训练可被劫持动态链接库对象模型, 并对发现的漏洞进行验证, 最终构建漏洞挖掘与验证的原型系统, 实现对 DLL 劫持漏洞的分析与挖掘, 保障计算机系统的稳固和安全.

## 2 漏洞原理

在 Windows 系统中, 如果进程加载某个 DLL 时发现路径未知, 则会按照系统的默认搜索顺序到对应的目录下进行 DLL 文件搜索, 一旦在某个目录下通过 DLL 名匹配到该 DLL 文件, 则会直接加载该 DLL, 并且停止对后续目录的查找操作. 这种搜索并加载 DLL 的方式在一定程度上简化了应用程序对 DLL 的调用, 但也带来了很多风险. 如图 3 所示, 在 Windows XP sp2 之前, Windows 查找 DLL 的目录及对应顺序为: 进程对应的应用程序所在目录、当前目录、系统目录、16 位系统目录、Windows 目录以及 PATH 环境变量中的各个目录.

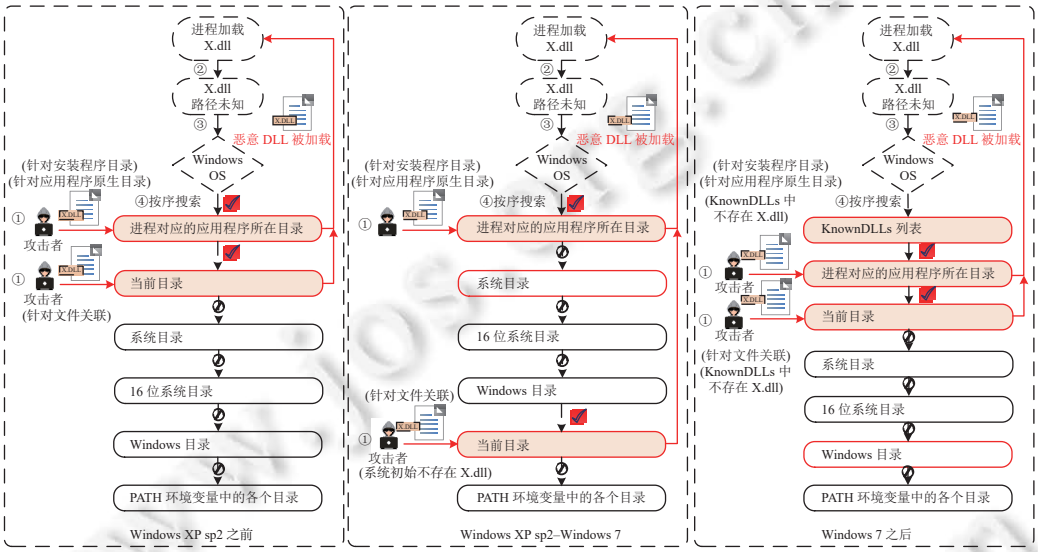



图 3 DLL 劫持漏洞原理图

由图 1 中关于 DLL 劫持的 3 种方式可知, 当攻击者把恶意的同名 DLL 文件放置在进程对应的应用程序所在目录或当前目录时, 操作系统优先搜索到该同名恶意 DLL, 直接返还给进程, 而不会再去后续的系统目录下查找, 导致劫持发生.

为了缓解这一问题, 从 Windows XP sp2 开始, SafeDllSearchMode 机制被提出. 由图 3 可以看出, 开启 SafeDllSearchMode 后, 搜索顺序中的当前目录被从之前的第 2 步调整至第 5 步, 在一定程序上抵御了针对文件关联的 DLL 劫持攻击, 系统 DLL 文件可以更优先被系统调用. 但是当所需 DLL 文件初始时并不存在于系统中时, 例如需要联网下载, 攻击者依然可以构造该 DLL 来进行文件关联攻击. 从 Windows 7 开始, 系统采用了新的 KnownDLLs 机制. 只要是该项中的 DLL 文件则禁止从应用程序自身所在目录下调用, 只能从系统目录下调用. 该机制的加入使得系统 DLL 能够在 KnownDLLs 中优先被系统调用. 然而未开启 SafeDllSearchMode 机制时, 攻击者可以对不存在于 KnownDLLs 中的 DLL 进行劫持攻击, 因此动态链接库的安全仍然需要更多研究.

本文研究的是 Windows 10 系统下针对安装程序的 DLL 劫持漏洞挖掘, 即安装程序在执行正常安装过程中调用 DLL 文件而出现的劫持行为. 用户通过网络下载的 EXE 安装程序会存放在某个特定的文件夹中, 而该文件夹的权限是普通用户级别的, 因此攻击者可以获取到该文件夹的路径, 伪造一些不在 KnownDLLs 中的恶意 DLL 文件, 并且加入大量恶意代码或者病毒木马程序. 当用户按照正常的操作流程, 打开该文件夹并点击 EXE 安装程序时, 安装进程会通过操作系统来寻找安装所需的 DLL 文件, 而当前目录下的恶意同名 DLL 文件会被匹配到, 并直接返还给安装进程, 导致恶意代码被加载, 给系统造成一定的安全风险.

微软在 2010 年 8 月 23 日发布的 2269637 号安全公告, 首次提到多个版本的操作系统存在 DLL 劫持的缺陷,

可能导致远程攻击, 随后 DLL 劫持漏洞开始大规模传播, 不断威胁着计算机系统的安全. 由于 DLL 文件在 Windows 操作系统中的基础性作用, 无论是安装一个程序, 还是执行一个文件, 都会涉及到对 DLL 的调用操作, 因此如何保证 DLL 文件的安全性是保障系统安全的一个关键点.

## 3 相关定义

为了更加准确地描述安装程序加载动态链接库文件的具体过程与操作, 本文分别对安装程序 EXE、动态链接库 DLL 文件和 EXE 调用 DLL 的模式 3 方面进行研究. EXE 文件是应用程序执行安装操作的主体, 需要首先进行研究. 通过对 Windows 平台上大量安装程序的分析发现, 对 EXE 进行属性划分, 可以高精度地描述某一个 EXE 的特征, 而具体属性可以分为功能类别、保护机制等. 通过多个维度的属性判决, 进而综合各属性取值, 可以精确描述 EXE 的信息. DLL 文件作为应用程序执行安装操作的客体, 也需要通过类似的属性划分方式进行定量描述. 为了更加细粒度的分析 DLL 文件情况, 本方法对 DLL 劫持中涉及到的 DLL 文件进行归纳整合, 避免了对系统中所有 DLL 文件进行大规模分析带来的低效率问题, 提高了实际安全研究的效率. 针对 DLL 劫持中涉及到的某个 DLL 文件, 通过叠加生成类别、目录属性、地址级别等属性的取值, 可以达到最大限度的特征描述, 方便对具有特定特征的某一个或某一类 DLL 文件进行定位. 此外, 安装操作是 EXE 和 DLL 之间动态的调用过程, 因此不能完全将两部分孤立开来. 第 3 部分即是对 EXE 调用 DLL 的模式进行属性划分, 主要考虑 EXE 在执行安装操作时所处系统级别和加载 DLL 文件的方式. 通过综合这两个方面的信息, 可以更加准确地描述调用的过程.

具体定义内容与属性取值如下.

### 3.1 EXE 文件属性

本文定义 EXE 的属性集合为 EXE.Attribute (EXE.A), 包括: 功能类别, 保护机制, 动态特征, PE 内容, 漏洞属性. 以下给出各项的具体内容.

(1) 功能类别 (function category, EXE.fc). 根据 EXE 安装程序具备的主要功能, 具体可分为以下几种: ① 输入法类 (inputting type, it); ② 下载类 (download type, dt); ③ 浏览器类 (browser type, bt); ④ 音乐类 (music type, mt); ⑤ 办公类 (office type, ot); ⑥ 游戏类 (game type, gt); ⑦ 视频类 (video type, vt); ⑧ 社交类 (social type, st); ⑨ 解压刻录类 (decompression type, det). 不同类别的应用程序由于所执行的功能不同, 安装程序在安装阶段所调用的 DLL 文件也存在着差别, 例如音乐类的安装程序会调用 Linkinfo.dll、Cscapi.dll 等, 而浏览器类安装程序则会调用 WindowsCodecs.dll、Msls31.dll 等, 因此该属性需要首先考虑.

(2) 保护机制 (protection mechanism, EXE.pm). 根据 EXE 安装程序自身所带的安全保护机制, 具体可分为以下几种: ① 加壳 Shell; ② 序列号保护 serial number ptotection; ③ 完整性检验 integrity check; ④ 时间限制 time limit; ⑤ 功能限制 function limit; ⑥ 信息隐藏 information hiding; ⑦ 加密算法 encryption algorithm; ⑧ 花指令 junk code; ⑨ 反调试 anti-debug; ⑩ 反跟踪 anti-trace; ⑪ 自修改代码 (self modifying code, SMC) 技术; ⑫ 多态变形 Polymorphic and metamotphic. 保护机制会限制对于 EXE 文件的静态和动态分析, 因此需要优先进行研究.

(3) 动态特征 (dynamic feature, EXE.df). 通过本地或在线的沙箱环境对 EXE 安装程序进行动态分析, 提取动态特征信息, 主要包括以下几类信息: ① 文件操作 (file operation, fo), 包括文件查找、文件读写、文件建立、文件删除等; ② 注册表操作 (registry operation, ro), 包括注册表条目查找、注册表写入删除等; ③ 网络流量 (network flow, nf), 主要包括安装过程中读取、发送与接收网络请求等; ④ 系统交互 (system interaction, si), 主要包括读取系统信息、读取用户输入等.

(4) PE 内容 (pe content, EXE.pc). 根据安装程序在静态环境下 DLL 文件路径, 具体可分为以下几种: ① 全部为绝对路径情况 (all-absolute-paths, aap), 主要指各 DLL 都为绝对路径; ② 存在非绝对路径情况 (non-absolute-paths, nap), 主要指存在一些 DLL, 其路径没有给出绝对地址.

(5) 漏洞属性 (vulnerability attribute, EXE.va). 基于目前已知存在 DLL 劫持漏洞的 EXE 文件信息, 对 EXE 数据集进行划分, 具体包括: ① 已知存在漏洞 (known vulnerability, kv), 主要包括 CVE、CNVD 等平台上已知漏洞号;

② 尚未发现漏洞 (no vulnerability found, nvf), 指实验前尚未在互联网上存在 DLL 劫持漏洞信息.

3.1.1 保护机制判定规则

软件保护机制是软件开发者为了保护软件不被静态逆向分析和动态调试的有效措施, 具体包括加壳、序列号保护、花指令、信息隐藏、反调试等. 由于网络下载的 EXE 安装程序种类繁多, 功能各异, EXE 安装程序本身可能存在各种保护机制, 在抽取动态特征、PE 内容等属性时会对研究产生屏障, 因此首先需要对保护机制进行判别, 再根据保护机制的不同类别进行针对性去除操作, 从而保证后续属性的抽取能顺利进行, 保证 EXE 属性集的准确性与完备性.

保护机制的分析判定与去除需要考虑每一种保护机制的不同类别, 每一种类别所采用方法的合理性影响着最终分析与研究的效率, 因此要对各种类型的保护机制进行更细致化的分类. 具体如表 2 所示.

表 2 保护机制具体分类

| 保护机制 | 具体分类 | 简记 | 保护机制 | 具体分类 | 简记 |
|---|---|---|---|---|---|
| 加壳<br>Shell | 加密壳<br>protectors | prs | 完整性检验<br>integrity check | 磁盘完整性校验<br>disk integrity check | dic |
| | 压缩壳<br>packers | pac | | 内存完整性校验<br>memory integrity check | mic |
| | 虚拟机保护<br>virtual machine protect | vmp | | 校验和<br>checksum | chs |
| 序列号保护<br>serial number ptotection | 单变量简单加密<br>single variable && simple encryption | svse | 时间限制<br>time limit | 限制单次时长<br>limit single-use time | lsut |
| | 多变量简单加密<br>multiple variables && simple encryption | mvse | | 限制总时长<br>limit total-use time | ltut |
| | 单变量复杂加密<br>single variable && complex encryption | svce | 功能限制<br>function limit | 正式版与试用版<br>official and trial versions | oatv |
| | 多变量复杂加密<br>multiple variables && complex encryption | mvce | | 注册机制<br>registration | reg |
| 加密算法<br>encryption algorithm | 单向散列加密<br>one-way hash encryption algorithms | ohea | 信息隐藏<br>information hiding | 利用文件冗余空间<br>using the redundant space | utrs |
| | 对称加密<br>symmetric encryption algorithms | sea | | 利用结构与运行特点<br>using the structure and operation characteristics | usoc |
| | 非对称加密<br>asymmetric encryption algorithms | aea | 花指令<br>junk code | 可执行花指令<br>executable junk code | ejc |
| | 其他加密算法<br>other encryption algorithms | oea | | 不可执行花指令<br>non-executable junk code | nejc |
| 反调试<br>anti-debug | 调试器<br>debugger | deb | 反跟踪<br>anti-trace | 结构化异常处理<br>structured exception handler | seh |
| | 调试标志位<br>debugger flag | def | | 调试器断点<br>debug register breakpoint | drb |
| | 调试时长<br>debugger time | det | | 不可执行花指令<br>non-executable junk code | nejc |
| 代码自修改<br>SMC | 多层嵌套加密<br>multilevel nested encryption | mne | 多态变形<br>polymorphic and metamotphic | 静态多态性<br>static polymorphic | sp |
| | 复杂加密函数<br>complex encryption function | cef | | 动态多态性<br>dynamic polymorphic | dp |
| | 与反跟踪结合<br>combined with anti-trace | cwat | | | |

对于表 2 中每一种保护机制及其具体分类, 通过静态分析、动态分析、相关工具等方式, 进行针对性的去除, 具体过程如图 4 所示.

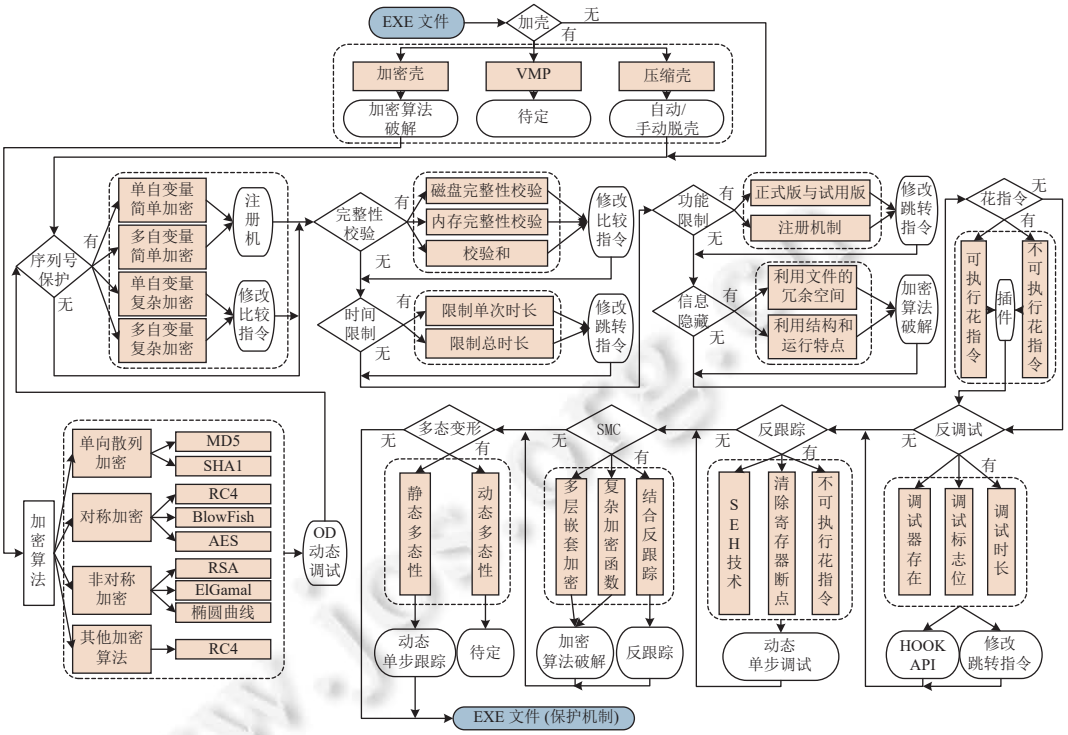



图 4 安装程序保护机制判定与去除流程

由图 4 可以看出, 通过加壳工具与算法的检测, 判定 EXE 文件是否存在加密壳、压缩壳或者 VMP, 加密壳可以通过破解加密算法进行脱壳, 压缩壳也可以通过工具或手动脱壳, 而 VMP 由于较为复杂, 去除难度大, 脱壳成功率较低, 因此将存在此类加壳的 EXE 文件放入待定集合, 不作为目前所研究的内容. 对于加密壳和压缩壳, 脱壳之后继续进行序列号保护检测, 分别使用注册机或修改比较指令来处理序列号保护机制, 接着在完整性校验、时间限制、功能限制的处理部分, 通过修改相关指令也可以较为高效地进行去除. 使用花指令去除插件完成对花指令的处理后, 进入动态保护机制检测部分, 分别检测反调试、反跟踪、SMC 以及多态变形技术, 相关分类以及去除方式如图 4 所示. 其中, 针对动态多态性, 由于该保护机制去除难度大, 并且在实际研究中出现频率较低, 因此此处也是将存在该保护机制的 EXE 文件放入待定集合. 按照图 4 所示对已知存在漏洞和漏洞属性未知两种类型的 EXE 文件进行保护机制判定与去除, 最终可以得到完备的 EXE 文件保护机制属性集.

### 3.1.2 保护机制判定算法

为了更加细化 EXE 安装程序的保护机制判定及去除, 并且将去除的过程与属性生成联系起来, 增加算法的描述, 对每个保护机制的具体分析过程进行代码处理, 将判定的结果用具体的数值标记, 最终综合生成目标 EXE 安装程序的唯一保护机制属性值.

下面针对保护机制的判定与去除, 给出具体的算法描述, 如算法 1.

**算法 1**. 保护机制的判定与去除.

Input: EXE(fc) //输入: 带有功能类别的 EXE 文件;

Output: EXE(fc, pm) //输出: 经过保护机制检测算法后增加保护属性的 EXE 文件.

```
1. if Shell_Det(EXE(fc))!=0 //是否存在加壳机制
2.      EXE.pm1=1; //存在则置该标志位为 1
3.      if Shell_Det(EXE(fc))==prs || pac, crack encryption algorithm || use unpackers; //加密壳或压缩壳
4.      if Shell_Det(EXE(fc))==vmp, put it as pending, then return EXE((fc, pm)); //VMP 则放入待定集合
5. endif
6. if Ser_Num_Prot_Det(EXE(fc))!=0 //是否存在序列号保护机制
7.      EXE.pm2=1; //存在则置该标志位为 1
8.      if Ser_Num_Prot_Det(EXE(fc))==svse || mvse, use KeyGen; //简单加密
9.      if Ser_Num_Prot_Det(EXE(fc))==svce || mvce, modify comparison instruction; //复杂加密
10. endif
11. if Int_Che_Det(EXE(fc))!=0, modify comparison instruction, EXE.pm3=1; //是否存在完整性校验
12. if Time_Limit_Det(EXE(fc))!=0, modify jumping instruction, EXE.pm4=1; //是否存在时间限制
13. if Func_Limit_Det(EXE(fc))!=0, modify jumping instruction, EXE.pm5=1; //是否存在功能限制
14. if Info_Hiding_Det(EXE(fc))!=0, crack encryption algorithm, EXE.pm6=1; //是否存在信息隐藏
15. if Encry_Algo_Det(EXE(fc))!=0, dynamic debugging, EXE.pm7=1; //是否存在加密算法
16. if Junk_Code_Det(EXE(fc))!=0, dynamic debugging, EXE.pm8=1; //是否存在花指令
17. if Anti-Deb_Det(EXE(fc))!=0, api hooking or modify jumping instruction, EXE.pm9=1; //是否有反调试
18. if Anti-Trace_Det(EXE(fc))!=0, dynamic debugging, EXE.pm10=1; //是否存在反跟踪
19. if SMC_Det(EXE(fc))!=0, EXE.pm11=1 //是否存在 SMC, 存在则置该标志位为 1
20.      if SMC_Det(EXE(fc))==mne || cef, crack encryption algorithm; //多层嵌套或复杂加密
21.      if SMC_Det(EXE(fc))==cwat, dynamic debugging; //涉及反跟踪技术
22. endif
23. if Poly_Meta_Det(EXE(fc))!=0, EXE.pm12=1 //是否存在多态变形, 存在则置该标志位为 1
24.      if Poly_Meta_Det(EXE(fc))==sp, dymatic debugging; //静态多态性
25.      if Poly_Meta_Det(EXE(fc))==dp, put it as pending; //动态多态性, 则放入待定集合
26. endif
27. EXE.pm = (EXE.pm12, EXE.pm11, ..., EXE.pm2, EXE.pm1);//所有保护机制的标志位集合
28. return EXE(fc, pm) //确定最终的保护机制属性值
```

算法初始对于每个保护机制设定标志位 pm1 至 pm12, 所有标志位初始全部赋值为 0, 最终返回的 EXE.pm 为各个标志位的集合. 为了更好地对后续的网络进行训练, 此处按照静态保护机制与动态保护机制的分类方法, 先对所有保护机制进行分类: 静态保护机制 (加壳 Shell、序列号保护 serial number ptotection、完整性检验 integrity check、时间限制 time limit、功能限制 function limit、信息隐藏 information hiding 和花指令 junk code); 动态保护机制 (反跟踪 anti-trace、反调试 anti-debug、SMC 技术、多态变形 polymorphic and metamotphic 和加密算法 encryption algorithm), 接着用保护机制的复杂程度来定义每个 EXE 的保护机制属性, 即从 Exe.pm12 开始, 直到 Exe.pm1, 首个不为 0 的属性即为该 EXE 文件最后的保护机制属性. 如果不存在任何保护机制, 则该属性置 0. 通过该方法, 可以最大程度反映 EXE 安装程序自身的保护机制复杂程度, 方便后期对程序进行分析研究.

算法的复杂性方面, 通过具体实例分析可以得出, 算法执行的时间复杂度与 EXE 安装程序自身的保护机制层数 $n$ 有关. 具体研究不同保护机制所采取的方法, 对于加壳、序列号保护、完整性校验、时间限制、功能限制、信息隐藏等模块, 采用的去保护机制较为简单, 并且通常存在完善的工具可以使用, 若存在 $n$ 个保护机制, 则这些模块的时间复杂度为 O($n$), 而对于花指令、加密算法、反调试、反跟踪、SMC 以及多态变形等模块, 由于保护机制的去除难度大, 而且对应的去保护工具需要结合人工处理, 并且通常存在某个保护机制嵌套另一个保护机制的

情况出现, 因此对于这几个模块, 若保护机制的层数为 $n$, 则时间复杂度约为 $O(n^2)$. 最终本原型系统所使用的保护机制判定与去除算法, 时间复杂度为 $O(n^2)$. 而空间复杂度则与算法中设置的标志位 pm1 至 pm12 有关, 此外涉及到的具体判定函数中所用空间则根据保护机制的不同而不同, 因此空间复杂度为常数阶, 即 O(1). 总体而言, 本算法主要是采用伪代码的形式对保护机制判定与去除流程进行简要示意, 具体的时间复杂度、空间复杂度等, 需要根据不同安装程序所带有的保护机制类别来进行判定, 而通过图 4 可以看出, 所列举的保护机制中, 除去待定的 VMP 和动态多态性, 其他保护机制都有较为成熟的去保护措施, 包括破解算法、动态调试等, 因此算法整体的复杂性是可接受的范围, 时间复杂度和空间复杂度都是合理可控的.

### 3.2 DLL 文件属性

本文定义 DLL 的属性集合为 DLL.Attribute (DLL.A), 包括生成类别, 地址级别, 目录属性, 调用依赖. 以下给出各项的具体内容.

(1) 生成类别 (generation category, DLL.gc). DLL 文件根据生成方式具体可以分为 4 种: ① 非 MFC 的 DLL (non-mfc, nm); ② 静态链接到 MFC 的 DLL (static link to mfc, sltm); ③ 动态链接到 MFC 的 DLL (dynamic link to mfc, dltm); ④ 扩展 DLL (extension dll, exd). 使用不同生成方式构造的 DLL 文件, 可能存在被调用方式上的不同.

(2) 目录属性 (directory attribute, DLL.da). 根据 DLL 文件原始所在地址, 具体可分为如下 3 种: ① \windows\system32 (ws32), 操作系统的系统文件夹, 用来存放 64 位程序文件; ② \windows\syswow64 (ws64), 64 位 Windows 用来存放 32 位 Windows 系统文件的地方; ③ \windows (wi), 该目录下存在少量 DLL 文件. 本属性主要考虑被劫持的 DLL 文件原始位置, 用于后期对文件进行定位.

(3) 地址级别 (address attribute, DLL.aa). 根据劫持漏洞发生时目标 DLL 文件所在地址属性, 具体包括如下几种可能: ① 在系统核心目录 KnownDLLs (knd) 中; ② 正在加载该 DLL 的进程的可执行文件目录 (exe directory, ed); ③ 系统目录 (system directory, sd); ④ 正在加载该 DLL 的进程的当前目录 (current directory, cd); ⑤ PATH 环境变量中列出的目录 (path directory, pd). 发生 DLL 劫持时, DLL 文件处于不同位置, 对应着不同的劫持方式, 因此需要考虑地址级别.

(4) 调用依赖 (call dependency, DLL.cd). 根据 DLL 文件在被调用时是否会依赖其他 DLL, 具体可以分为: ① 依赖其他 DLL 文件 (dependent on, do); ② 不依赖其他 DLL 文件 (non-dependent on, ndo). DLL 文件在调用时是否依赖其他 DLL, 将会影响该 DLL 被调用到进程空间的流程.

### 3.3 EXE 调用 DLL 的模式属性

本文定义 EXE 调用 DLL 的模式属性集合为 EXE_DLL.Attribute (EXE_DLL.A), 包括: 运行级别, 加载方式. 以下给出各项的具体内容.

(1) 运行级别 (operation authority, EXE_DLL.oa). 根据 EXE 安装程序在执行安装操作时所处系统级别, 具体可以分为以下几种: ① 管理员级别 (administrator level, al); ② 普通用户级别 (ordinary user level, oul). 当安装程序需要管理员权限来执行安装操作时, 需要调用的 DLL 文件可能不等同于普通用户级别下的情况, 因此在调用模式属性集里需要考虑该属性.

(2) 加载方式 (loading mode, EXE_DLL.lm). 根据 EXE 安装程序在加载 DLL 文件到进程空间时的方式, 具体可以分为以下几种: ① 静态加载 (static loading, stl), 也称隐式调用, 指在运行程序之前由操作系统的加载器将 DLL 和 EXE 一起加载到内存里; ② 动态加载 (dynamic loading, dyl), 也称显式调用, 指应用程序运行过程中程序自己完成对 DLL 的加载; ③ 延迟加载 (delayed loading, del), 指使用特定编译链接技术使得相关 DLL 不在 EXE 一开始运行时就加载进来, 而是等到正式调用该 DLL 的时候, 才会进行加载, 使得程序在正式调用 DLL 之前, 都可以正常执行. EXE 加载 DLL 的 3 种不同方式, 决定了 DLL 被调用的模式, 也会对最终漏洞的挖掘产生影响.

## 4 基于双层 BiLSTM 的 DLL 劫持漏洞挖掘方法

本文利用循环神经网络中的双层 BiLSTM 实现了 DLL 劫持漏洞挖掘及验证的原型系统 BiL-DLLMining, 其

中选用 BiLSTM 神经网络, 主要原因在于该网络是一种采用前向 LSTM 和后向 LSTM 结合的多层反馈神经网络, 在对大量 EXE 文件的多维度属性所构成的数据集进行训练的过程中, 能根据正向传播与反向传播的数值进行权值和阈值的动态调整, 对不同属性赋予不同权重, 同时挖掘属性之间关联性, 直到输出结果与预期结果的误差达到要求, 从而形成高准确率的安装程序漏洞特征判决模型. 此外, 利用该神经网络还可以通过对安装程序与动态链接库的全属性集进行神经网络训练, 分类得出安装程序属性集与动态链接库属性集之间的深层映射关系, 从而构建可劫持对象判决模型. 而采用双层的原因在于, 双层网络的判决准确率相较于单层的更高. 综合安装程序漏洞特征判决模型与可劫持对象判决模型, 即可实现对未知安装程序的 DLL 劫持漏洞分析与挖掘.

系统的整体架构图如图 5 所示.

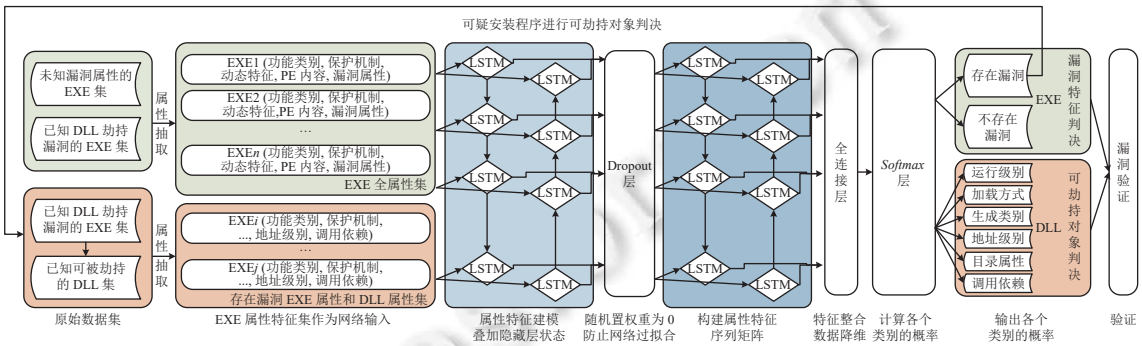



图 5 BiL-DLLMining 原型系统架构图

由图 5 可以看出, BiL-DLLMining 原型系统主要包括安装程序漏洞特征判决模型与可劫持对象判决模型, 安装程序漏洞特征判决模型所选用的数据集为未知漏洞属性的 EXE 文件与已知 DLL 劫持漏洞的 EXE 文件, 预期输出为 EXE 文件是否具有 DLL 劫持漏洞, 而可劫持对象判决模型则将已知存在 DLL 劫持的 EXE 文件与被劫持的 DLL 文件对应起来, 作为模型的输入数据, 预期输出为可劫持的 DLL 文件多维度特征. 系统最终对挖掘到的漏洞进行逐一验证, 不仅可以保证系统挖掘的准确性, 也保障了 Windows 操作系统与动态链接库的安全.

### 4.1 安装程序漏洞特征判决模型

为了达到对未知漏洞属性的安装程序进行漏洞判定的目的, 需要通过神经网络对安装程序的静态特征与动态特征进行训练, 学习出具备漏洞属性的安装程序特征集, 进而利用该特征集挖掘新的安装程序 DLL 劫持漏洞. 首先, 在对大量安装程序进行静态分析与动态分析的基础上, 加入目前已知存在 DLL 劫持漏洞的安装程序, 对各个安装程序进行多维度属性抽取, 共同构建安装程序样本集, 作为双层 BiLSTM 网络训练和测试的原始数据集. 随后按照神经网络中标准的 7:3 比例[34]划分训练集与测试集. 接着通过双层 BiLSTM 网络的训练得到漏洞特征判决模型, 最后挖掘存在漏洞的 EXE 安装程序. 具体过程如下.

(1) 输入数据规范化. 由于原始数据集中包含了大量 EXE 的属性信息, 而这些属性在最初抽取时是以文本形式存在数据库中, 长度不一, 字符类型也不便于直接放入模型进行训练, 因此需要先对属性进行规范化操作, 从而生成属性向量集, 使得模型的输入是规范一致的. 具体规范化时需要考虑各个属性的实际取值范围以及统计性规律, 第 3 节中详细给出了各属性的实际取值范围, 相关统计性规律则在第 5 节的实验部分进行给出, 此处直接展示规范化的结果. 安装程序 EXE 各属性的规范化结果为: 功能类别为 $(-4,4)$ 上的均匀分布, 即 $\text{EXE.fc} \sim U(-4,4)$; 保护机制为 $(-6,6)$ 上的标准正态分布, 即 $\text{EXE.pm} \sim N(0,1)$ && $\text{EXE.pm} \in [-6,6]$; 动态特征为 $(-2,2)$ 上的标准正态分布, 即 $\text{EXE.df} \sim N(0,1)$ && $\text{EXE.df} \in [-2,2]$; PE 内容和漏洞属性分别按 $-1/1$ 标注法进行统一化, 如表 3 所示.

(2) 计算遗忘门 $f_t$、输入门 $i_t$ 和输出门 $o_t$ 的各自取值. LSTM 模型主要由 $t$ 时刻的输入 $X_t$、记忆神经单元 $C_t$、临时记忆神经单元 $\tilde{C}_t$、隐藏层状态的输出 $h_t$、遗忘门 $f_t$、输入门 $i_t$ 和输出门 $o_t$ 组成. 对于 $t$ 时刻的遗忘门 $f_t$、输入门 $i_t$ 和输出门 $o_t$, 计算公式如下:

$$f_t = \sigma \cdot W_f x_t + \sigma \cdot W_f h_{t-1} + \sigma \cdot b_f \tag{1}$$

$$i_t = \sigma \cdot W_i x_t + \sigma \cdot W_i h_{t-1} + \sigma \cdot b_i \tag{2}$$

$$o_t = \sigma \cdot W_o x_t + \sigma \cdot W_o h_{t-1} + \sigma \cdot b_o \tag{3}$$

其中, $W_f$、$W_i$ 和 $W_o$ 分别代表遗忘门 $f_t$、输入门 $i_t$ 和输出门 $o_t$ 的权重, $h_{t-1}$ 代表 $t-1$ 时刻隐藏层状态的输出. $b_f$、$b_i$ 和 $b_o$ 分别为遗忘门 $f_t$、输入门 $i_t$ 和输出门 $o_t$ 的偏置项. $\sigma$ 为 Sigmoid 激活函数, 即:

$$\sigma(t) = \frac{1}{1+\mathrm{e}^{-t}} \tag{4}$$

$\sigma$ 的作用是将实数映射到 (0, 1) 上, 并且使得 90% 以上的值都接近 0 或者 1.

表 3　对 PE 内容和漏洞属性进行标注

| 属性 | 取值 | 说明 |
|---|---|---|
| EXE.pc | 1 | 存在非绝对路径 |
| | −1 | 全部为绝对路径 |
| EXE.va | 1 | 已知存在漏洞 |
| | −1 | 尚未发现漏洞 |

(3) 计算临时记忆神经单元 $\tilde{C}_t$ 和当前时刻记忆神经单元 $C_t$.

$$\tilde{C}_t = \tanh \cdot W_C x_t + \tanh \cdot W_C h_{t-1} + \tanh \cdot b_C \tag{5}$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{6}$$

其中, $W_C$ 为权重矩阵, $b_C$ 为对应的偏置项, tanh 为不同于 Sigmoid 的另一种激活函数, 即:

$$\tanh(t) = \frac{\mathrm{e}^{t}-\mathrm{e}^{-t}}{\mathrm{e}^{t}+\mathrm{e}^{-t}} \tag{7}$$

tanh 为 Sigmoid 函数向下平移和收缩后的表示.

(4) 计算当前时刻隐藏层状态的输出 $h_t$:

$$h_t = o_t \cdot \tanh(C_t) \tag{8}$$

(5) 获取隐藏层状态序列 $H$. $H = \{h_0, h_1, ..., h_{n-1}\}$.

(6) 构建双向 LSTM, 计算输入安装程序的属性特征表示 $H_t$. BiLSTM 由正向和反向两个 LSTM 网络组成, 每个网络会从相反两个方向对输入的属性特征进行处理, 最终对正反两部分的隐藏层状态表示 $\overrightarrow{H_t}$ 和 $\overleftarrow{H_t}$ 进行叠加操作, 得到双向 LSTM 的输出:

$$H_t = \overrightarrow{H_t} \oplus \overleftarrow{H_t} \tag{9}$$

(7) 加入 dropout 层, 设定每一个神经网络层进行 dropout 的概率, 在每次训练过程中随机将上一个 LSTM 输出的部分元素的权重 $r_j^j$ 置为 0, 使得输入不改变输出. 这样也可以在一定程度上缩减参数量, 防止网络过拟合.

计算 dropout 层的输出 $D_t$:

$$D_t = H_t * R_t \tag{10}$$

其中, 权重 $r_j^j$ 符合伯努利分布, 取值为 1 的概率为 $p$, 即 $r_j^j \sim Bernoulli(p)$. $R_t$ 为权重 $r_j^j$ 的集合. *为矩阵乘法.

(8) 加入第 2 个 BiLSTM 层, 构建关于学习到安装程序属性特征的序列矩阵, 具体 LSTM 的网络训练过程此处不再赘述. 接着进入一个全连接层, 对高维数据进行降维处理的同时, 进一步提取和整合关键特征信息, 得到全连接层的安装程序输出表示 $FC_t$.

(9) 最后进入 *Softmax* 层, 接收经全连接层降维处理后的特征向量, 计算并输出各个类别的概率:

$$p(\text{EXE.va}) = Softmax(FC_t) \tag{11}$$

其中, *Softmax* 的计算公式如下:

$$Softmax(x_i) = \frac{\mathrm{e}^{x_i}}{\sum_{j=1}^{n}\mathrm{e}^{x_j}},\ 0 \leqslant i \leqslant n-1 \tag{12}$$

通过 *Softmax* 层, 可以得出EXE.va为 1 和−1 的概率, 即可知安装程序存在 DLL 劫持漏洞的概率.

(10) 漏洞特征判决. 根据训练结果不断调整参数, 包括 LSTM 层各个门结构的权重、隐藏层的个数、dropout 层的概率等, 再进行逐步优化, 当测试集给出的训练精度满足预先要求时, 结束算法, 基于双层 BiLSTM 循环神经网络的安装程序漏洞特征判决模型至此构建完成. 对于未知的安装程序, 进行属性抽取之后, 将数据放入判决模型中, 根据模型输出的结果进行有无漏洞的判决. 下一步将判定结果为存在漏洞的安装程序放入可劫持对象判决模型中, 目的是通过该模型找到安装程序可被劫持的具体 DLL 文件, 通过后续的验证对漏洞判决模型的结果给出反馈, 同时得到真正存在 DLL 劫持漏洞的安装程序和对应的 DLL 文件.

### 4.2 可劫持对象判决模型

对于漏洞判决模型中输出为存在漏洞的安装程序, 需要进一步找到其可被劫持的 DLL 文件, 寻找方法是通过预先抽取的 DLL 文件属性, 对特定的 DLL 文件进行定位, 对满足要求的 DLL 文件逐个验证. 所涉及到的属性具体包括安装程序调用 DLL 的权限、加载方式、DLL 文件生成类别、地址属性、目录属性、调用依赖等. 可劫持对象判决模型的构建过程与漏洞判决模型具体操作存在一些差异, 主要体现在: 从安装程序全数据集中随机选取 70% 的数据, 作为新的数据集. 原因在于此时不再需要标签为−1 的数据, 只对已知存在漏洞的安装程序数据进行训练. 数据集的删减可以大幅度地降低训练的难度, 提高训练效率. 此外的一个不同点在于, 双层 BiLSTM 神经网络的输出不再是上一个模型中的−1/1, 而是 DLL 文件的各项属性值, 即将 DLL 文件的各项属性值作为安装程序的预测输出, 目的是通过双层 BiLSTM 神经网络训练出安装程序各项属性与被劫持 DLL 文件各项属性之间的高精度映射关系, 从而通过对安装程序的属性分析, 找到相对应被劫持概率高的 DLL 文件, 进行后期的 DLL 劫持验证. 具体过程如下.

(1) DLL 文件属性提取. 对目前已知存在 DLL 劫持的 EXE 文件, 首先在实验环境下复现 DLL 劫持情况, 并通过静态和动态分析相结合的方式, 提取 DLL 文件的各项关键属性, 包括: 安装程序调用 DLL 的权限、加载方式、生成类别、目录属性、地址级别和调用依赖. 各项属性的集合可以有效地反映 DLL 文件的基本信息, 同时可以精确定位到指定范围内的 DLL 文件. 最终得到目前已知存在 DLL 劫持漏洞的 DLL 文件各项属性信息, 包括 EXE_DLL.Attribute (EXE_DLL.A) 和 DLL.Attribute (DLL.A), 即双层 BiLSTM 的预期输出属性集为 *Out*(DLL) = EXE_DLL.A+DLL.A.

(2) 预期输出规范化. 从原始 EXE 特征数据集中抽取出具备 DLL 劫持漏洞的数据, 命名为 EXEVul, 作为可劫持动态链接库对象判决模型的输入, 而预期输出则为所对应的 DLL 文件相关属性. 对预期输出的各个属性进行规范化操作. 具体方式如表 4 所示.

表 4 预期输出规范化

| 属性 | 取值 | 说明 | 属性 | 取值 | 说明 |
|---|---|---|---|---|---|
| EXE_DLL.oa | −1 | 管理员级别 | DLL.cd | −1 | 不依赖其他DLL |
| | 1 | 用户级别 | | 1 | 依赖其他DLL |
| EXE_DLL.lm | −1 | 静态加载 | DLL.da | −1 | \windows\system32 |
| | 0 | 动态加载 | | 0 | \windows\syswow64 |
| | 1 | 延迟加载 | | 1 | \windows |
| DLL.gc | −1 | 非MFC的DLL | DLL.aa | −2 | KnownDLLs |
| | 0 | 静态链接到MFC的DLL | | −1 | 可执行文件的目录 |
| | 1 | 动态链接到MFC的DLL | | 0 | 系统目录 |
| | 2 | 扩展DLL | | 1 | 进程当前目录 |
| | | | | 2 | PATH变量目录 |

(3) 模型构建. 将安装程序漏洞特征判决模型的输入和预期输出进行相应修改, 赋予各个门初始权重, 进行多次迭代训练, 使得最终 *Softmax* 层输出结果近似等于预期情况. 至此, 可劫持动态链接库对象判决模型构

建完成.

(4) 漏洞验证. 将安装程序漏洞特征判决模型给出的 EXE 文件放入模型中, 根据输出结果, 对满足各属性要求的 DLL 文件进行抽取, 并利用相关工具验证. 最终给出确切的 DLL 劫持漏洞名单. 具体验证过程主要是使用 AheadLib 工具构造同名 DLL 文件, 并在 dllmain 函数中加入 MessageBox 函数进行弹窗操作, 将该 DLL 文件放置到 EXE 文件所在的安装程序目录下, 如果 EXE 安装程序调用了该 DLL 文件, 则会执行 dllmain 函数中的弹窗操作, 进行弹窗显示, 进而证明该 EXE 文件存在 DLL 劫持漏洞, 可被该 DLL 劫持.

## 5 实 验

本节首先给出实验环境与评价指标, 接着介绍安装程序漏洞特征判决模型及其分类结果, 包括属性特征的规范化、判决模型过程、测试输出结果. 随后介绍可劫持动态链接库对象判决模型的运行过程与结果, 同时对给出的安装程序与 DLL 文件进行对应验证. 最后, 通过与其他类似工具的对比分析来说明 BiL-DLLMining 原型系统的高性能.

### 5.1 实验环境与评价指标

实验数据集. 实验主要搜集目前已知的安装程序 DLL 劫持漏洞以及对应的 DLL 文件信息, 包括但不限于 CVE、CNVD 等平台, 数据获取方法为: 搜集 CVE、CNVD、漏洞银行、漏洞盒子等漏洞平台上从 2010 年 DLL 劫持漏洞爆发至今 (2020 年 6 月) 所有的 DLL 劫持漏洞所涉及的安装程序, 主要包括漏洞号、具体的安装程序名称、安装程序版本、对应可劫持的 DLL 名称、操作系统版本等, 通过数据比对去除重复信息, 最终综合得到已知存在 DLL 劫持漏洞的安装程序 378 个, 对应可劫持的 DLL 文件 315 个, DLL 文件数量少于安装程序数量的原因在于部分安装程序可劫持的 DLL 文件相同. 同时对主流应用程序提供平台上的安装程序进行搜集, 主要包括腾讯软件管家、360 软件管家、2345 软件管家等, 时间限定在近 3 年, 获取多类型的安装程序信息, 并综合各个应用程序的官网, 去除重复数据后, 最终得到不同种类、不同版本的安装程序共 1 954 个, 默认这些安装程序的 DLL 劫持漏洞属性为未知.

经过上述步骤, 最终搜集到已知存在 DLL 劫持漏洞的安装程序 378 个, 构成数据集 EXEVul, 漏洞属性未知的安装程序 1 954 个, 构成数据集 EXEUn, 而 EXEVul 和 EXEUn 汇总后构成 EXE 全数据集, 命名为 EXEFul, 共有 EXE 安装程序 2 332 个. DLL 全数据集命名为 DLLFul, 共有 DLL 文件 315 个. 其中, 安装程序漏洞特征判决模型的训练数据集为 70% 的 EXEFul, 即 1 632 个安装程序 (EXEVul 中随机选取 265 个, EXEUn 中随机选取 1 367 个), 测试数据集为剩下的 30% 的 EXEFul, 即 700 个安装程序 (EXEVul 中剩下 113 个, EXEUn 中剩下 587 个). 同理, 对于可劫持对象判决模型, 训练数据集为 70% 的 EXEVul, 即 265 个安装程序以及对应的可劫持 DLL 文件 219 个 (265 个 EXE 程序为 EXEVul 中随机选取), 而测试数据集为 EXEVul 剩下的 113 个已知存在漏洞的安装程序以及对应的可劫持 DLL 文件 102 个, 其中由于存在不同 EXE 可被相同 DLL 劫持的情况, 导致训练集和测试集中 DLL 文件的总和并不等于 DLLFul 的元素总量.

对安装程序和 DLL 文件进行属性特征分析与提取时, 涉及到工具主要包括 PEiD、Pestudio、IDA、OllyDbg、Depends、WinHex 等, 涵盖了主流的 PE 文件静态与动态分析工具. 通过前述工具对 EXE 安装程序和 DLL 文件进行多方位属性提取, 能够多维度表征 EXE 和 DLL 的深层次特征, 保证系统可以高效地处理输入的数据集, 最终在多次迭代和训练之后, 给出 EXE 漏洞特征集和可劫持 DLL 文件属性集, 经过实际的验证之后成功挖掘到 10 个未知的 DLL 劫持漏洞, 证实了 BiL-DLLMining 原型系统的可靠性与高效性, 为后续系统的进一步优化与应用打下坚实的基础.

具体的实验环境参数如下所示.

1) 硬件平台

处理器: Inter(R) Core(TM) i5-8300H CPU@2.30 GHz

内存: 16 GB

显卡: NVIDIA GeForce GTX1050Ti

2) 软件平台

操作系统: Windows 10 家庭版 (64 位)

系统版本: 1909

开发环境: VS 2019

实验所用评价指标为针对二分类结果常见的评估方法. 首先做如表 5 所示定义.

表 5　评价指标所用参数对照表

| 参数 | 具体含义 |
|---|---|
| *FP* | 本身不存在漏洞但被模型判定为存在漏洞的安装程序数量 |
| *FN* | 本身存在漏洞但被模型判定为不存在漏洞的安装程序数量 |
| *TP* | 本身存在漏洞并且被模型判定为存在漏洞的安装程序数量 |
| *TN* | 本身不存在漏洞并且被模型判定为不存在漏洞的安装程序数量 |

接着对表 5 中 4 个数值进行算术运算, 构成以下几个评估指标:

$$FPR=\frac{FP}{FP+TN},\ FNR=\frac{FN}{FN+TP},\ TPR=\frac{TP}{TP+FN},$$

$$TNR=\frac{TN}{TN+FP},\ P=\frac{TP}{TP+FP},\ F1=2\times\frac{P\times TPR}{P+TPR}.$$

分析各个指标的具体含义可知, 理想的模型输出需要满足误报率 (false positive rate, *FPR*) 和漏报率 (false negative rate, *FNR*) 尽量小, 而查全率 (true positive rate, *TPR*)、特异度 (true negative rate, *TNR*)、精确率 (precision, *P*) 和 *F*1 值 (*F*1-measure, *F*1) 尽量大. 后续对于两个模型所给的结果, 都通过计算出上述 6 个指标的数值, 分析评估模型训练的优劣, 从而更精确地表征原型系统 BiL-DLLMining 的 DLL 劫持漏洞挖掘能力.

### 5.2　漏洞特征判决模型运行与结果分析

● 属性特征的规范化. 通过对 EXE 的静态分析与动态分析, 发现属性的取值呈现一定的规律性. 在功能类别部分, 样本集中音乐、视频、下载等常见类别呈现近似均匀分布, 因此对该属性在进行规范化时可以采用简单的区间随机分配方式. 对于保护机制, 通过统计发现, 83.72% 的 EXE 安装程序并没有采用复杂的保护机制, 例如花指令、反调试、SMC、多态变形等, 而是只采用简单保护机制或者并没有保护机制存在, 整体呈现近似正态分布的特征, 因此对该属性的规范化操作可以采用指定区间上标准正态分布的方法, 从而保证对该属性的精确描述. 同理, 动态特征部分, EXE 安装程序在安装过程中, 91.47% 的程序都会进行文件相关操作, 例如文件查找、文件创建、文件读写等, 同时也会在注册表中进行程序注册等操作, 而网络流量和系统交互则相对较少一些. 因此, 总体来看, 动态特征项也可以采用标准正态分布的方法进行规范化. PE 内容主要分为文件中 DLL 文件名全部为绝对路径和存在非绝对路径两种情况, 因此直接采用−1/1 标注即可, 在训练后期根据网络训练效果进行进一步归一化等操作, 例如加入 mapminmax 等归一化函数. 此外, 使用−1/1 标签法对每个程序目前可被劫持情况进行标记, 用作网络的期望输出值. 具体规范化操作已在第 4.1 节给出, 此处不再赘述.

● 模型判决过程分析. 针对收集到的大量数据集进行属性提取、输入, 通过 BiLSTM 网络进行建模, 在每次训练过程中用 dropout 层进行随机元素权重置 0, 接着通过全连接层进行特征整合和数据降维, 最后通过 *Softmax* 层进行计算输出, 得出模型识别的各类别概率信息. 实际训练为 160 轮, 迭代次数为 3 200 次, 每轮的迭代次数为 20 次. 整体模型训练过程如图 6 所示. 由图 6 可知, 训练准确度最初较低, 随着迭代次数不断增加, 整体准确度开始逐步爬升, 最终稳定在 89.76%. 在训练初始阶段的损失率最高, 达到了 69.83%, 而不断迭代使得损失率呈现明显下降, 并很快下降到 21.05%, 稳定在小范围内, 直到训练结束. 整体而言, 双层 BiLSTM 的训练呈现较好态势, 训练准确度由低至高, 逐步提升, 训练损失率由高至低, 逐渐降低. 通过前期对于安装程序属性的准确划分与规范化, 以及网络中相关参数的动态调整, 整个网络训练情况良好.

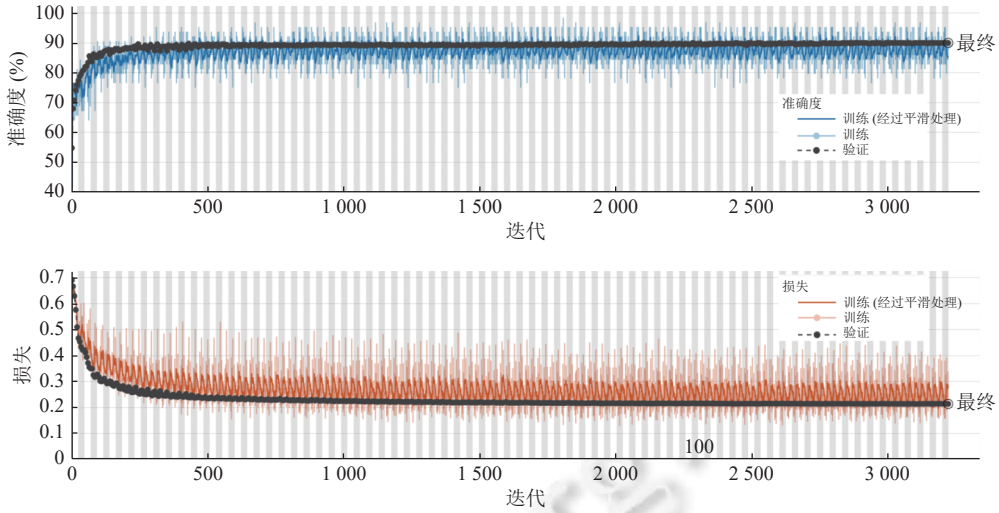



图 6 双层 BiLSTM 网络训练准确度与损失图

● 模型判决结果分析. 双层 BiLSTM 网络在最终的 *Softmax* 层会输出安装程序属于不同类别的概率. 为了更清晰地反映判决结果, 本文采用混淆矩阵的形式进行输出. 同时将原始混淆矩阵由预测数量与真实数量的对比改为百分比的形式. 模型会给出多次判决的结果, 此处选取其中 3 次的结果作为对比, 具体结果如图 7 所示.

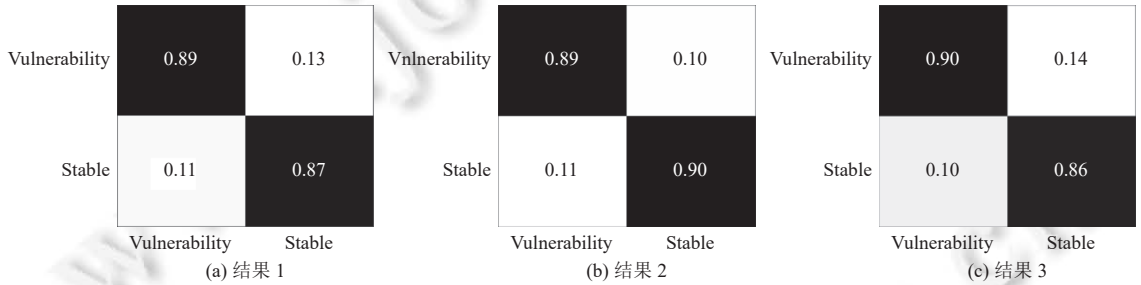



图 7 漏洞特征判决模型输出结果

由图 7 可以看出, 模型 3 次输出的结果相差不大, 证明在经过多层多次迭代与数据降维等操作后, 模型可以很好地对 EXE 文件的漏洞属性进行判决, 高精度地判断出 EXE 文件是否存在 DLL 劫持漏洞. 实验最终选取多次输出中漏洞属性判决精度最高的一次, 即图 7(c), 该结果下模型给出的概率为:

$$\begin{cases} p(Softmax(\text{EXE}) = 1 \mid \text{EXE.va} = 1) = 90\% \\ p(Softmax(\text{EXE}) = -1 \mid \text{EXE.va} = 1) = 10\% \\ p(Softmax(\text{EXE}) = -1 \mid \text{EXE.va} = -1) = 86\% \\ p(Softmax(\text{EXE}) = 1 \mid \text{EXE.va} = -1) = 14\% \end{cases}$$

对应指标可知 $FPR = 14\%$, $FNR = 10\%$, $TPR = 90\%$, $TNR = 86\%$, 查询具体输出可知 $P = 81.08\%$, $F1 = 85.31\%$. 通过对所有输出结果的分析可以发现, 双层 BiLSTM 网络的判决准确率维持在 80% 以上, 满足预期设定要求.

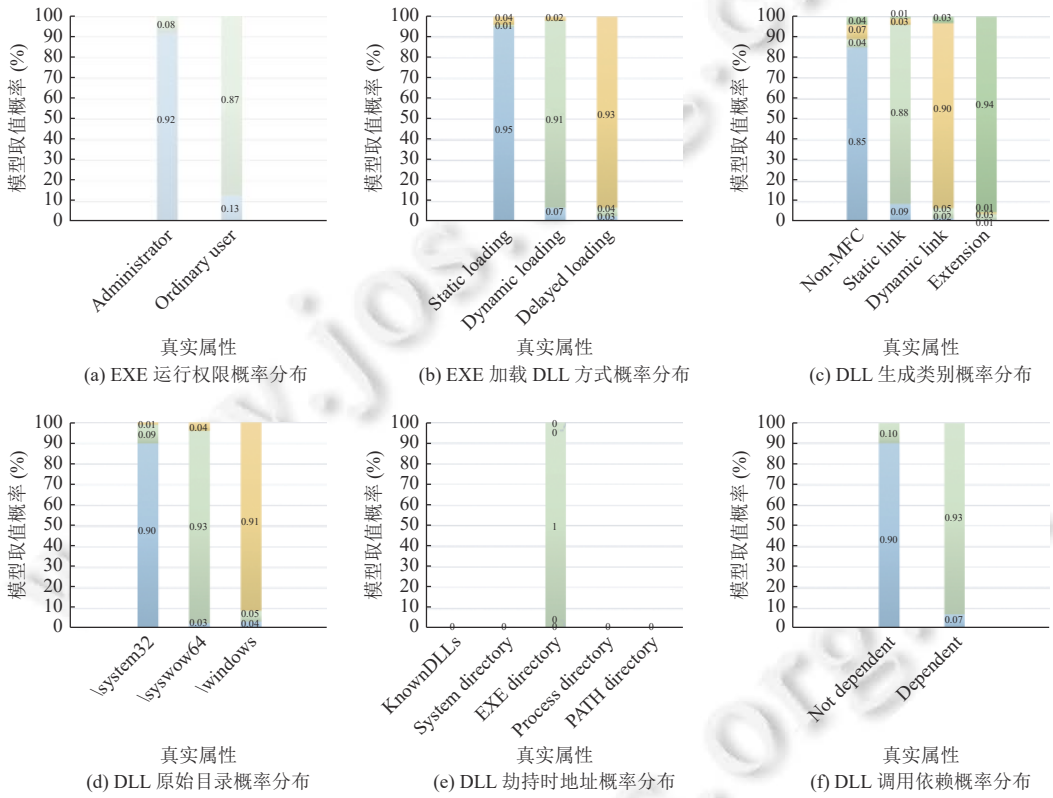
● 可疑安装程序选取. 通过上述步骤, 漏洞特征判决模型即建模完毕. 训练集和测试集最终得到的判决准确率也满足模型的要求. 在同步进行可劫持对象判决模型构建的同时, 从 EXEFul 和金山毒霸软件管家、160 软件管家等其他平台上, 选取 200 个安装程序, 其中包括已知存在 DLL 劫持漏洞的安装程序和新的未知漏洞属性的安装程序, 进行属性抽取与规范化操作后, 放入训练好的漏洞特征判决模型中, 最终有 16 个 EXE 安装程序的 EXE.va 被判定为 1, 即大概率存在 DLL 劫持漏洞. 在可劫持对象判决模型构建完成后, 这些 EXE 安装程序将放入模型中, 由

该模型给出对应可被劫持的 DLL 信息, 并进行后续的验证操作.

### 5.3 可劫持对象判决模型运行与结果分析

● 属性特征的规范化. 通过对 DLL 文件进行多角度分析, 得出各个属性的统计性数据. 在分析 EXE 调用 DLL 情况时, 普通用户级别下静态加载是主要的方式. 而 DLL 文件本身的各个属性里, 静态链接到 MFC 的 DLL 文件占主体, 同时大部分可劫持的 DLL 文件原始目录都处于\windows\syswow64 下, 被劫持时处于可执行文件目录下, 并且不依赖其他 DLL 的情况占到 85% 以上. 根据这些统计性数据, 对各个属性进行规范化操作, 具体方法已在第 4.2 节列出, 此处不再赘述.

● 模型判决过程分析. 将 EXEVul 划分为训练集和测试集, 使用修改后的双层 BiLSTM 网络进行训练, 最终通过 *Softmax* 层得到各个属性取值的概率分布. 概率分布采取百分比柱状图的表达形式. 具体情况如图 8 所示.




(a) EXE 运行权限概率分布 (b) EXE 加载 DLL 方式概率分布 (c) DLL 生成类别概率分布

(d) DLL 原始目录概率分布 (e) DLL 劫持时地址概率分布 (f) DLL 调用依赖概率分布

图 8 可劫持对象判决模型输出类别概率

从图 8 可以看出, 模型对于 EXE 运行权限、EXE 加载 DLL 方式、DLL 生成类别、DLL 原始目录属性、DLL 劫持时地址级别以及 DLL 调用依赖进行判决时, 准确率都在 85% 以上. 其中, 对于 DLL 劫持时地址级别, 由于本文所研究的是安装程序的 DLL 劫持漏洞, 因此搜集以及复现的都是 EXE 在安装阶段出现的漏洞, 所劫持的 DLL 文件全都处于可执行文件 EXE 所在目录下, 该属性值都为可执行文件的目录, 模型所给判决图自然只有这一个值. 然而由于 DLL 劫持不限于安装程序这一种, 因此对这一属性进行尽可能全面的划分, 有利于后期对其他几种 DLL 劫持类型进行更进一步的研究.

将混淆矩阵中的准确率数值转换为表格形式, 具体如表 6 所示. 由表 6 可以看出, 对于 DLL 文件涉及到的几个属性, 模型给出的分类平均准确率都达到了 85% 以上, 而整体属性的平均准确率也都为 90% 以上. 因此可劫持动态链接库对象判决模型的实际效果达到预期要求, 可以高精度地给出可疑安装程序的可被劫持 DLL 文件.

表 6 可劫持对象判决模型输出准确率

| 属性 | 分类 | 分类平均准确率 (%) | 属性平均准确率 (%) |
|---|---|---|---|
| 运行级别 (operation authority) | Administrator level<br>Ordinary user levl | 92<br>87 | 89.50 |
| 加载方式 (loading mode) | Static loading<br>Dynamic loading<br>Delayed loading | 95<br>91<br>93 | 93.00 |
| 生成类别 (generation category) | Non-MFC<br>Static link<br>Dynamic link<br>Extension | 85<br>88<br>90<br>94 | 89.25 |
| 目录属性 (directory attribute) | \windows\system32<br>\windows\syswow64<br>\windows | 90<br>93<br>91 | 91.33 |
| 地址级别 (address attribute) | EXE directory | 100 | 100 |
| 调用依赖 (call dependency) | Not dependent on<br>Dependent on | 90<br>93 | 91.50 |

● 可劫持对象判决分析. 对该模型给出的可疑 EXE 安装程序与对应可劫持 DLL 文件的属性进行统计分析, 具体统计性结果如图 9 所示. DLL 文件的生成类别、目录属性、地址级别和调用依赖 4 个属性用来定位某一范围内的 DLL 文件, 而运行级别和加载方式两个属性用于最终验证阶段对 EXE 的环境进行准确配置.

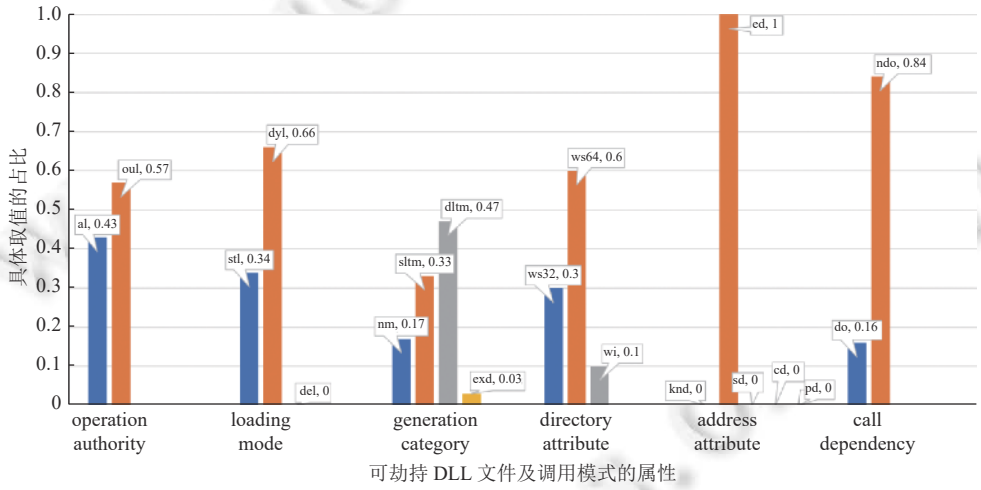



图 9 可劫持对象判决结果统计性分析

从图 9 可以看出, DLL 文件生成类别方面, 静态链接到 MFC 和动态链接到 MFC 占据了 80%, 而原始所在位置主要为\windows\system32 和\windows\syswow64, 此外有 84% 的 DLL 文件在被调用时是不依赖其他 DLL 的. 在 EXE 调用 DLL 的模式方面, 普通用户级别达到了 57%, 动态加载达到了 66%. 通过模型可以很清晰地给出各个属性取值的概率, 选取概率最大的各个属性进行拼接, 即可对应到特定某一个或某几个 DLL 文件, 从而便于使用其他方法进一步验证漏洞是否真实存在.

● 漏洞验证. 基于安装程序漏洞特征判决模型所给出的 16 个可疑 EXE 文件, 经过可劫持对象判决模型的分析后, 给出对应每个可疑 EXE 文件的 DLL, 使用 AheadLib 等相关工具生成同名的 DLL 文件, 在文件中加入 MessageBox 函数进行弹窗操作, 将 DLL 文件对应放置在 EXE 文件所在目录下, 按照 EXE 调用 DLL 文件的运行权限, 逐一进行安装操作, 进行漏洞验证. 验证情况如表 7 所示. 从表 7 可以看出, 安装程序漏洞特征判决模型所给

出的 16 个 EXE 文件中, 14 个经验证的确存在 DLL 劫持漏洞, 其中 4 个为已知漏洞, 10 个为未知漏洞, 未知漏洞都已进行申报并获批 CNVD 编号.

表 7 漏洞验证

| EXE | 中文名称 | 功能类别 | 运行环境 | 存在漏洞 | 劫持所用DLL | CNVD编号 |
|---|---|---|---|---|---|---|
| WPS.exe | WPS办公 | 办公 | Win10家庭版 | True | shfolder.dll | 未知, 已获批, CNVD-2020-02235 |
| Potplayer.exe | Potplayer播放器 | 视频 | Win10家庭版 | False | — | — |
| PCQQ.exe | QQ客户端 | 社交 | Win10家庭版 | True | version.dll | 未知, 已获批, CNVD-2020-26364 |
| Sogou_Pinyin.exe | 搜狗拼音输入法 | 输入法 | Win10家庭版 | True | version.dll | 未知, 已获批, CNVD-2020-18637 |
| Chrome.exe | 谷歌浏览器 | 浏览器 | Win10家庭版 | True | linkinfo.dll | 已知, CNVD-2020-23843 |
| BaiduPinyin.exe | 百度拼音输入法 | 输入法 | Win10家庭版 | True | shfolder.dll | 未知, 已获批, CNVD-2020-18738 |
| XunLei.exe | 迅雷 | 下载 | Win10家庭版 | True | version.dll | 未知, 已获批, CNVD-2020-26371 |
| KuaiZip.exe | 快压 | 解压刻录 | Win10家庭版 | True | bcrypt.dll | 未知, 已获批, CNVD-2020-26363 |
| 360se.exe | 360浏览器 | 浏览器 | Win10家庭版 | False | — | — |
| KSbrowser.exe | 猎豹浏览器 | 浏览器 | Win10家庭版 | True | winsta.dll | 未知, 已获批, CNVD-2020-18662 |
| Sogou_explorer.exe | 搜狗浏览器 | 浏览器 | Win10家庭版 | True | iertutil.dll | 未知, 已获批, CNVD-2020-18671 |
| GuaGuaPlayer.exe | K歌伴侣客户端 | 音乐 | Win10家庭版 | True | version.dll | 已知, CNVD-2020-00790 |
| BaiduNetdisk.exe | 百度网盘 | 下载 | Win10家庭版 | True | dpapi.dll | 未知, 已获批, CNVD-2020-26360 |
| ysyy.exe | 央视影音客户端 | 视频 | Win10家庭版 | True | cldapi.dll | 已知, CNVD-2020-03921 |
| QQBrowser.exe | QQ浏览器 | 浏览器 | Win10家庭版 | True | usp10.dll | 未知, 已获批, CNVD-2020-26361 |
| yy.exe | YY语音 | 社交 | Win10家庭版 | True | usp10.dll | 已知, CNVD-2020-12841 |

以 CNVD-2020-18637 为例, BiL-DLLMining 原型系统挖掘的完整流程如下: 首先抽取目标 EXE 安装程序的属性特征, 即搜狗拼音输入法安装程序的功能类别、保护机制、动态特征与 PE 内容, 其中功能类别为输入法, 保护机制为无, 动态特征为文件操作和注册表操作, PE 内容为存在非绝对路径情况; 接着将上述属性规范化后, 将其输入到原型系统 BiL-DLLMining 中, 漏洞特征判决模型对属性进行多维度分析训练, 最终给出漏洞属性判定, 结果为存在 DLL 漏洞概率为 93.25%, 因此该安装程序被判定为存在 DLL 劫持漏洞; 随后将上一步中获取的属性信息输入到可劫持对象判决模型中, 通过对已有 DLL 劫持漏洞中 EXE 文件和 DLL 文件的映射关系解析, 得到目标 EXE 文件, 即搜狗拼音输入法安装程序可劫持的 DLL 文件, 结果共输出 3 个, 分别为 version.dll、usp10.dll 和 shfolder.dll. 最后使用 AheadLib 工具分别对 3 个 DLL 文件进行逐一验证, 验证方法主要是通过构造同名 DLL 文件放置到对应的安装程序目录下, 并在 dllmain 函数中加入 MessageBox 函数进行弹窗操作. 如果可以成功劫持某个 DLL 文件, 则安装程序会调用伪造的 DLL, 并且执行 dllmain 函数中的弹窗操作, 进行弹窗显示. 实验最终发现伪造的 version.dll 成功被劫持, 安装程序直接在当前目录下调用了该伪造的 DLL 文件, 并进行弹窗, 继续修改伪造的 DLL 文件中相关函数, 可以实现更多的操作, 例如在安装过程中打开其他进程、获取主机的敏感信息等.

对于原型系统的实时性分析, 目前主流的系统实时性评估工具有 Benchmark 和 profile 等, 其中 Benchmark 采用压力测试挖掘整个系统的性能状况, 而 profile 工具最大限度地呈现系统的运行状态和性能指标, 方便用户诊断性能问题和进行调优. 本文采用上述两种工具来对 BiL-DLLMining 进行实时性分析. 使用 Benchmark 生成 50 个具体的安装程序 EXE 文件, 其中属性集为工具按照已有规则自动生成, 采用在线处理的方式测试 BiL-DLLMining 原型系统对于测试用例的执行时间, 最终某一测试用例的输出结果为 (50, 17947362948 ns/op), 表示该测试用例执行了 50 次, 平均每次执行时间约为 17.947 s (17 947 362 948 ns), 共花费约 897.368 s. 通过改变测试用例的属性, 即安装程序 EXE 文件的多种属性后, 重复实验 20 次, 最终获得的平均单个测试用例执行时间为 16.276 s. 从时间角度考虑, 原型系统的表现满足预期要求. 在实验过程中同时采用 profile 进行运行状态的分析, 单次单个测试用例中具体的函数执行情况如表 8所示, 表中只选取占比较高的函数.

表 8 profile 分析结果

| Time percentage (%) | Cumulative time (s) | Self time (s) | Calls | Self/call (s) | Total/call (s) | Name |
|---|---|---|---|---|---|---|
| 40.79 | 7.324 | 7.324 | 2 | 3.662 | 4.104 | DoubleBiLSTM |
| 23.24 | 11.497 | 4.173 | 5 | 0.835 | 0.925 | Data_split |
| 22.79 | 15.593 | 4.096 | 8 | 0.512 | 0.617 | Attrib_weight |

从表 8 可以看出, BiL-DLLMining 原型系统对于测试用例执行漏洞分析与挖掘过程中, 耗时最多的函数为 DoubleBiLSTM, 即系统的核心函数处理函数, 时间占比达到 40.79%, 其他关于数据划分、权重赋值等函数也消耗了一定的时间量. 综合 Benchmark 和 profile 的结果来看, 原型系统对于安装程序 EXE 文件的处理, 实时性表现良好, 核心函数运行效果较高.

### 5.4 对比分析

目前学术界和工业界对于 Windows 系统安全都有较多的研究, 但对于动态链接库劫持漏洞的研究则较少. 传统对于 DLL 劫持的检测方法, 主要是采用 process explorer 等工具分析进程加载的所有 DLL 文件, 通过与注册表项中的 DLL 文件对比, 同时加上加载路径的判定, 最终选取可疑的 DLL 文件进行逐一验证, 从而尝试查找存在劫持漏洞的程序. 这类方法思路简单, 但是实用性不强, 对于新的程序需要重复操作, 不具备普适性, 并且人工操作繁琐, 难以实现漏洞的高效挖掘与分析. 为了增加对漏洞的快速挖掘, 工业界出现了一些 DLL 劫持漏洞挖掘的自动化工具, 尝试从不同的角度进行漏洞的自动化挖掘. 本文选取目前主流的几种自动化挖掘工具进行对比分析, 最终给出各维度的分析结果, 表明本文所提出模型的高效性与完备性.

本文选取的 3 种自动化挖掘工具分别为 Rattler[35]、ChkDllHijack[36]和 DLLHSC[37], 3 款工具分别通过不同的方式实现对 DLL 劫持漏洞的挖掘.Rattler 作为一款开源的自动化漏洞挖掘工具, 主要通过枚举进程调用的 DLL 列表, 解析出 DLL 文件, 再将测试 DLL 分别重命名为列表中的 DLL 名称, 最后重启程序来观察是否有测试进程被劫持, 从来判断目标应用程序是否存在 DLL 劫持漏洞. ChkDllHijack 是一款批量验证 DLL 劫持漏洞的工具, 提供了 UI 界面便于操作. 将目标应用程序和对应加载的 DLL 文件放入对应文本框即可进行检查. DLLHSC 全称为 DLL Hijack Scanner, 即 DLL 劫持扫描工具, 提供了 3 种操作模式: 轻量级模式, 解析内存中可执行镜像的导入表, 接着用特定 DLL 替换导入表中引用的任何 DLL, 通过消息弹窗来寻找可疑 DLL 文件; 列表模块模式, 使用可执行镜像创建进程, 接着枚举加载到此进程的地址空间中的模块, 对不属于 KnownDlls 的模块进行逐一验证; 运行时模式, 通过 Microsoft Detours 对 LoadLibrary 和 LoadLibraryEx API 进行 hook 操作, 并报告在运行时加载的模块, 最后选取可疑 DLL 文件进行验证.

为了更加细致且针对性地对 3 款工具进行分析, 本文从 EXEFul 中随机选取 500 个安装程序, EXEFul 为上文提到的安装程序集, 包括 CVE、CNVD、漏洞银行、漏洞盒子等漏洞平台上从 2010 年 DLL 劫持漏洞爆发至今(2020 年 6 月) 所有的 DLL 劫持漏洞涉及的安装程序, 加上腾讯软件管家、360 软件管家、2345 软件管家等平台上获取到的安装程序. 500 个安装程序为随机选取, 包含 150 个已知存在漏洞的 EXE 文件和 350 个漏洞属性未知的 EXE 文件, 研究 3 款工具与本文所提模型在使用方式、分析粒度、挖掘形式、输出结果等特征以及准确率等方面的差异. 具体结果如表 9 和表 10 所示.

由表 9 可以看出, Rattler 使用命令行的方式, 对于应用程序分析较为粗粒度, 对于所加载的 DLL 文件需要逐一验证, 输出的结果也只是 DLL 路径和 DLL 名. 而同样使用命令行方式的 DLLHSC, 则需要按照要求进行编译, 并且 3 种模式使用起来较为复杂. ChkDllHijack 使用了更为简易的 UI 界面, 但是需要和其他工具进行配合, 在其他工具先给出应用程序所加载的 DLL 文件的基础上, 才能进行验证操作. 本文实现的系统 BiL-DLLMining 在对应用程序和 DLL 文件进行细粒度分析的基础上, 利用神经网络对属性特征进行多维抽取与分类, 挖掘各个属性之间隐藏的关联性, 最终输出结果也是应用程序和 DLL 的多维度属性, 便于后续高效精确地对目标应用程序和 DLL 进行验证. 此外, 几款工具对于漏洞的检测是最为重要的考察指标, 因此依旧采取第 5.1 节所述的几种评价指标, 例如 *FPR*、*FNR* 等, 通过多次实验分析几款工具在检测安装程序 DLL 劫持漏洞方面的准确率.

表 9 特征对比结果

| | Rattler | ChkDllHijack | DLLHSC | BiL-DLLMining |
|---|---|---|---|---|
| 使用方式 | 命令行 | UI界面 | 命令行 | 模型接口 |
| 分析粒度 | 粗粒度 | 粗粒度 | 细粒度 | 细粒度 |
| 挖掘形式 | 枚举DLL,<br>逐一验证 | 其他工具预先分析出所有<br>DLL文件, 逐个验证 | 3种不同模式, 编译以及<br>使用较复杂 | 应用程序多维属性集合, 特征抽取与分<br>类, 漏洞模型进行判决 |
| 输出结果 | DLL路径<br>DLL名 | DLL路径<br>DLL名 | DLL路径<br>DLL名 | 应用程序运行级别<br>DLL文件加载方式<br>DLL文件生成类别<br>DLL文件原始地址<br>DLL文件劫持目录<br>DLL文件调用依赖 |

表 10 准确率对比结果 (%)

| 指标 | Rattler | ChkDllHijack | DLLHSC | 本文方法 |
|---|---|---|---|---|
| *FPR* | 20.66 | 19.33 | 16.00 | **12.66** |
| *FNR* | 22.76 | 19.51 | 17.89 | **13.82** |
| *TPR* | 77.24 | 80.49 | 82.11 | **86.18** |
| *TNR* | 79.34 | 80.67 | 84.00 | **87.34** |
| *P* | 75.40 | 77.34 | 81.45 | **84.80** |
| *F*1 | 76.33 | 82.54 | 81.78 | **85.48** |

如表 10 所示, Rattler 的 *F*1 分数只能达到 76.33%, ChkDllHijack 和 DLLHSC 在前者的基础上提高了准确率, *F*1 分数分别达到了 82.54% 和 81.78%, 但是依旧存在可提升空间. 本文实现的系统 BiL-DLLMining 由于对应用程序、DLL 和应用程序调用 DLL 这 3 方面进行属性抽取, 更加细粒度地分析漏洞应用程序的特征以及可被劫持的 DLL 文件特征, 从而能更加准确地实现对 DLL 劫持进行分析与挖掘, 最终不仅对已知漏洞进行了有效验证, 还挖掘出多个未知漏洞, 实现了真正意义上的 DLL 劫持漏洞挖掘, 准确率也有很大幅度的提升.

## 6 结 论

动态链接库作为 Windows 操作系统底层的核心组件, 为上层众多的应用程序服务, 不仅方便开发者进行程序开发, 也提高了系统的整体运行效率. DLL 劫持漏洞的出现将 DLL 的安全性推到了人们的面前, 促使安全人员对其展开深入研究, 从而保障 DLL 文件以及整个系统的安全. 本文基于目前已知的 DLL 劫持漏洞, 搜集并抽取了大量安装程序的属性特征, 从安装程序自身、安装程序调用动态链接库模式、动态链接库自身 3 个角度出发, 使用双层 BiLSTM 神经网络进行特征学习, 分类出具备 DLL 劫持的漏洞特征, 在验证了已知漏洞的同时也挖掘到一些未知漏洞. 同时通过与目前主流的 DLL 劫持漏洞分析与挖掘工具对比, 表明了本文基于 BiLSTM 神经网络的漏洞挖掘及验证系统 BiL-DLLMining 的高效与完备性. 当然目前的系统还存在一些不足之处, 比如在对安装程序进行保护机制判定部分, 对于虚拟机保护以及多态变形等复杂机制的解决还存在可以改进的地方, 整个模型的数据集也需要继续扩大, 准确率也有继续提升的空间. 未来会针对这些问题进行更加深入的研究.

**References:**

[1] Berdajs J, Bosnić Z. Extending applications using an advanced approach to DLL injection and API hooking. Software: Practice and Experience, 2010, 40(7): 567–584. [doi: 10.1002/spe.973]

[2] Fu JM, Yang Z, Luo CK, Huang JW. An anti-cheat method of game based on windows kernel events. Journal of Electronics & Information Technology, 2020, 42(9): 2117–2125 (in Chinese with English abstract). [doi: 10.11999/JEIT190695]

[3] Peng D, Zhou AM, Fang Y, Liu L. A method for detecting Dynamic Link Library preload vulnerability. Information and Electronic Engineering, 2011, 9(4): 515–518 (in Chinese with English abstract). [doi: 10.3969/j.issn.1672-2892.2011.04.026]

[4] Choi YH, Min JW, Park MW, Eom JH, Chung TM. A framework of static analyzer for taint analysis of binary executable file. In: Park JJ, Barolli L, Xhafa F, eds. Information Technology Convergence. Dordrecht: Springer, 2013. 185−192. [doi: 10.1007/978-94-007-6996-0_20]

[5] Yousaf MS, Durad MH, Ismail M. Implementation of portable executable file analysis framework (PEFAF). In: Proc. of the 16th Int'l Bhurban Conf. on Applied Sciences and Technology (IBCAST). Islamabad: IEEE, 2019. 671−675. [doi: 10.1109/IBCAST.2019.8667202]

[6] Salman M, Husna D, Viani N. Static analysis method on portable executable files for REMNUX based malware identification. In: Proc. of the 10th IEEE Int'l Conf. on Awareness Science and Technology (iCAST). Morioka: IEEE, 2019. 1−6. [doi: 10.1109/ICAwST.2019.8923331]

[7] Zhang YC, Pang JM, Zhao RC. Evidential reasoning method for decision of program maliciousness. Ruan Jian Xue Bao/Journal of Software, 2012, 23(12): 3149−3160 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4221.htm [doi: 10.3724/SP.J.1001.2012.04221]

[8] Wang R, Feng DG, Yang Y, Su PR. Semantics-based malware behavior signature extraction and detection method. Ruan Jian Xue Bao/Journal of Software, 2012, 23(2): 378−393 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3953.htm [doi: 10.3724/SP.J.1001.2012.03953]

[9] Wang J, Wang CQ. Malware detection method based on subgraph similarity. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3436−3447 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5863.htm [doi: 10.13328/j.cnki.jos.005863]

[10] Chen ZF, Li QB, Zhang P, Ding WB. Data characteristics-based kernel malware detection. Ruan Jian Xue Bao/Journal of Software, 2016, 27(12): 3172−3191 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4927.htm [doi: 10.13328/j.cnki.jos.004927]

[11] Vyas R, Luo X, McFarland N, Justice C. Investigation of malicious portable executable file detection on the network using supervised learning techniques. In: Proc. of the 2017 IFIP/IEEE Symp. on Integrated Network and Service Management (IM). Lisbon: IEEE, 2017. 941−946. [doi: 10.23919/INM.2017.7987416]

[12] Awan S, Saqib NA. Detection of malicious executables using static and dynamic features of portable executable (PE) file. In: Wang GJ, Ray I, Calero JMA, eds. Security, Privacy and Anonymity in Computation, Communication and Storage. Cham: Springer, 2016. 48−58. [doi: 10.1007/978-3-319-49145-5_6]

[13] Atluri V. Malware classification of portable executables using tree-based ensemble machine learning. In: Proc. of the 2019 Southeast Conf. Huntsville: IEEE, 2019. 1−6. [doi: 10.1109/SoutheastCon42311.2019.9020524]

[14] Bazzi A, Onozato Y. IDS for detecting malicious non-executable files using dynamic analysis. In: Proc. of the 15th Asia-Pacific Network Operations and Management Symp. (APNOMS). Hiroshima: IEEE, 2013. 1−3.

[15] Hubballi N, Dogra H. Detecting packed executable file: Supervised or anomaly detection method. In: Proc. of the 11th Int'l Conf. on Availability, Reliability and Security (ARES). Salzburg: IEEE, 2016. 638−643. [doi: 10.1109/ARES.2016.18]

[16] Shibahara T, Yagi T, Akiyama M, Chiba D, Yada T. Efficient dynamic malware analysis based on network behavior using deep learning. In: Proc. of the 2016 IEEE Global Communications Conf. (GLOBECOM). Washington: IEEE, 2016. 1−7. [doi: 10.1109/GLOCOM.2016.7841778]

[17] Glendowne D, Miller C, McGrew W, Dampier D. Characteristics of malicious DLLs in Windows memory. In: Proc. of the 2015 IFIP Int'l Conf. on Advances in Digital Forensics XI. Orlando: Springer, 2015. 149−161. [doi: 10.1007/978-3-319-24123-4_9]

[18] Duan YH, Fu X, Luo B, Wang ZQ, Shi J, Du XJ. Detective: Automatically identify and analyze malware processes in forensic scenarios via DLLs. In: Proc. of the 2015 IEEE Int'l Conf. on Communications (ICC). London: IEEE, 2015. 5691−5696. [doi: 10.1109/ICC.2015.7249229]

[19] Narouei M, Ahmadi M, Giacinto G, Takabi H, Sami A. DLLMiner: Structural mining for malware detection. Security and Communication Networks, 2015, 8(18): 3311−3322. [doi: 10.1002/sec.1255]

[20] An L, Castelluccio M, Khomh F. An empirical study of DLL injection bugs in the Firefox ecosystem. Empirical Software Engineering, 2019, 24(4): 1799−1822. [doi: 10.1007/s10664-018-9677-7]

[21] Matsuda W, Fujimoto M, Mitsunaga T. Real-time detection system against malicious tools by monitoring DLL on client computers. In: Proc. of the 2019 IEEE Conf. on Application, Information and Network Security (AINS). Pulau Pinang: IEEE, 2019. 36−41. [doi: 10.1109/AINS47559.2019.8968697]

[22] Kwon T, Su ZD. Automatic detection of unsafe dynamic component loadings. IEEE Trans. on Software Engineering, 2012, 38(2): 293−313. [doi: 10.1109/TSE.2011.108]

[23] Kwon T, Su Z. Static detection of unsafe component loadings. In: Proc. of the 21st Int'l Conf. on Compiler Construction. Tallinn: Springer, 2012. 122−143. [doi: 10.1007/978-3-642-28652-0_7]

[24] Fu JM, Peng BC, Du H. Dynamic detection of component loading vulnerability. Journal of Tsinghua University (Science & Technology), 2012, 52(10): 1356–1363, 1369 (in Chinese with English abstract). [doi: 10.16511/j.cnki.qhdxxb.2012.10.007]

[25] Min B, Varadharajan V. Secure dynamic software loading and execution using cross component verification. In: Proc. of the 45th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Rio de Janeiro: IEEE, 2015. 113–124. [doi: 10.1109/DSN.2015.17]

[26] Ding S, Fu JM, Peng BC. ModuleGuard: A gatekeeper for dynamic module loading against malware. Wuhan University Journal of Natural Sciences, 2013, 18(6): 489–498. [doi: 10.1007/s11859-013-0962-4]

[27] Liu FY, Xie W. Research on DLL loading vulnerability defense technology based on signature verification. Netinfo Security, 2017(11): 62–66 (in Chinese with English abstract). [doi: 10.3969/j.issn.1671-1122.2017.11.010]

[28] Zhou JY, He PF, Qiu RF, Chen G, Wu WG. Research on intrusion detection based on random forest and gradient boosting tree. Ruan Jian Xue Bao/Journal of Software, 2021, 32(10): 3254−3265 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6062.htm [doi: 10.13328/j.cnki.jos.006062]

[29] Sohi SM, Seifert JP, Ganji F. RNNIDS: Enhancing network intrusion detection systems through deep learning. Computers & Security, 2021, 102: 102151. [doi: 10.1016/J.COSE.2020.102151]

[30] Sun PF, Liu PJ, Li Q, Liu CX, Lu XL, Hao RC, Chen JP. DL-IDS: Extracting features using CNN-LSTM hybrid network for intrusion detection system. Security and Communication Networks, 2020, 2020: 8890306. [doi: 10.1155/2020/8890306]

[31] Li Y, Huang CL, Wang ZF, Yuan L, Wang XC. Survey of software vulnerability mining methods based on machine learning. Ruan Jian Xue Bao/Journal of Software, 2020, 31(7): 2040−2061 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6055.htm [doi: 10.13328/j.cnki.jos.006055]

[32] Duan X, Wu JZ, Luo TY, Yang MT, Wu YJ. Vulnerability mining method based on code property graph and attention BiLSTM. Ruan Jian Xue Bao/Journal of Software, 2020, 31(11): 3404−3420 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6061.htm [doi: 10.13328/j.cnki.jos.006061]

[33] Matsuda W, Fujimoto M, Mitsunaga T. Detection of malicious tools by monitoring DLL using deep learning. Journal of Information Processing, 2020, 28: 1052–1064. [doi: 10.2197/ipsjjip.28.1052]

[34] Radwan AM. Machine learning techniques to detect maliciousness of portable executable files. In: Proc. of the 2019 Int'l Conf. on Promising Electronic Technologies (ICPET). Gaza: IEEE, 2019. 86–90. [doi: 10.1109/ICPET.2019.00023]

[35] Roy CL. Rattler. 2017. https://github.com/sensepost/rattler/

[36] Anhkgg. ChkDllHijack. 2021. https://github.com/anhkgg/anhkgg-tools

[37] Context Information Security. DLLHSC. 2020. https://github.com/ctxis/DLLHSC

附中文参考文献:

[2] 傅建明, 杨铮, 罗陈可, 黄坚伟. 一种基于内核事件的Windows系统游戏反外挂方法. 电子与信息学报, 2020, 42(9): 2117–2125. [doi: 10.11999/JEIT190695]

[3] 彭帝, 周安民, 方勇, 刘亮. 动态链接库预加载漏洞检测方法. 信息与电子工程, 2011, 9(4): 515–518. [doi: 10.3969/j.issn.1672-2892.2011.04.026]

[7] 张一弛, 庞建民, 赵荣彩. 基于证据推理的程序恶意性判定方法. 软件学报, 2012, 23(12): 3149–3160. http://www.jos.org.cn/1000-9825/4221.htm [doi: 10.3724/SP.J.1001.2012.04221]

[8] 王蕊, 冯登国, 杨轶, 苏璞睿. 基于语义的恶意代码行为特征提取及检测方法. 软件学报, 2012, 23(2): 378–393. http://www.jos.org.cn/1000-9825/3953.htm [doi: 10.3724/SP.J.1001.2012.03953]

[9] 汪洁, 王长青. 子图相似性的恶意程序检测方法. 软件学报, 2020, 31(11): 3436−3447. http://www.jos.org.cn/1000-9825/5863.htm [doi: 10.13328/j.cnki.jos.005863]

[10] 陈志锋, 李清宝, 张平, 丁文博. 基于数据特征的内核恶意软件检测. 软件学报, 2016, 27(12): 3172−3191. http://www.jos.org.cn/1000-9825/4927.htm [doi: 10.13328/j.cnki.jos.004927]

[24] 傅建明, 彭碧琛, 杜浩. 一种组件加载漏洞的动态检测. 清华大学学报(自然科学版), 2012, 52(10): 1356–1363, 1369. [doi: 10.16511/j.cnki.qhdxxb.2012.10.007]

[27] 刘峰宇, 解炜. 基于签名认证的DLL加载漏洞防御技术研究. 信息网络安全, 2017(11): 62–66. [doi: 10.3969/j.issn.1671-1122.2017.11.010]

[28] 周杰英, 贺鹏飞, 邱荣发, 陈国, 吴维刚. 融合随机森林和梯度提升树的入侵检测研究. 软件学报, 2021, 32(10): 3254–3265. http://www.jos.org.cn/1000-9825/6062.htm [doi: 10.13328/j.cnki.jos.006062]

[31] 李韵, 黄辰林, 王中锋, 袁露, 王晓川. 基于机器学习的软件漏洞挖掘方法综述. 软件学报, 2020, 31(7): 2040−2061. http://www.jos.org.

cn/1000-9825/6055.htm [doi: 10.13328/j.cnki.jos.006055]
[32] 段旭, 吴敬征, 罗天悦, 杨牧天, 武延军. 基于代码属性图及注意力双向LSTM的漏洞挖掘方法. 软件学报, 2020, 31(11): 3404−3420. http://www.jos.org.cn/1000-9825/6061.htm [doi: 10.13328/j.cnki.jos.006061]

**陈霄**(1995－), 男, 博士生, 主要研究领域为系统安全, 虚拟化安全, 网络安全.

**王众**(1999－), 男, 本科, 主要研究领域为软件安全, 物联网安全.

**肖甫**(1980－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为传感网, 物联网.

**底伟鹤**(1998－), 男, 本科生, 主要研究领域为软件安全, 物联网安全.

**沙乐天**(1985－), 男, 博士, 副教授, 主要研究领域为软件安全, 网络安全, 物联网安全.