一种基于污点追踪的系统审计日志压缩方法

贲永明[1,2], 韩言妮[1], 安 伟[1], 徐 震[1]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 中国科学院大学网络空间安全学院 北京 中国 100049

摘要 近十年来, 高级持续性威胁(APT, advanced persistent threat)越来越引起人们的关注。为了防御和检测 APT 攻击, 学者提出了基于系统审计日志的入侵取证方案。系统审计日志可以详细记录主机上的系统调用过程, 因此非常适用于入侵取证工作。然而, 系统审计日志也有着致命的弊端: 日志庞大冗余。再加上 APT 攻击往往长期潜伏、无孔不入, 企业不得不为每台联网主机长期保存日志, 因此导致巨大的存储计算成本。为了解决这一问题, 本文提出一种模仿二进制动态污点分析的日志压缩方案 T-Tracker。T-Tracker 首先检测日志内部与外部数据发生交互的系统调用, 生成初始污点集合, 然后追踪污点在主机内的扩散过程, 这个过程中只有污点扩散路径上的系统调用能被保留下来, 其余均不保留, 从而达到日志压缩的目的。本研究的测试表明, 该方案可以达到 80%的压缩效果, 即企业将能够存储相当于原来数量五倍的日志数据。同时, T-Tracker 完整保留了受到外部数据影响的日志记录, 因此对于入侵取证而言, 可以等价地替换原始日志, 而不会丢失攻击痕迹。

关键词 高级持续性威胁; 入侵取证; 系统级审计日志; 日志压缩; 污点追踪
中图法分类号 TP309.2 DOI 号 10.19363/J.cnki.cn10-1380/tn.2020.09.03

A System Audit Log Compression Method based on Taint Tracking

BEN Yongming[1,2], HAN Yanni[1], An Wei[1], Xu Zhen[1]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] Schol of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract In the past ten years, considerable attention has been paid to Advanced Persistent Threats(APTs). To defend and detect APT attack, intrusion forensics based on system-level audit log has been proposed. System-level audit log is highly suitable for intrusion forensics because it records the interactions among system entities in details. However, it has a fatal shortcoming due to its massive growth of log size. And the condition becomes worse when we are talking about defense of APTs. Enterprises have to monitor each host in a long period of time to expose stealthy attackers, which causes overwhelming storage costs. To address this issue, this paper proposes an audit log compression algorithm, named T-Tracker, which imitates Dynamic Taint Analysis in binary program. Firstly T-Tracker detects the events that cause the information flow from external data sources and generates the initial taint sets. Then it tracks the diffusion of the taint according to the audit log. By retaining the events on diffusion path only, we can achieve the audit log compression. Our evaluation on different system workloads and attack cases demonstrates that our approach can achieve significant log compression without affecting the accuracy of intrusion forensics.

Key words advanced persistent threat; intrusion forensics; system-level audit log; log compression; taint tracking

1 概述

随着网络攻防对抗的升级, 具备隐蔽性、持续性和针对性的高级持续性威胁(APTs)对各类高等级信息系统造成的威胁日趋严重。例如, 2010 年的“震网”病毒经过多年的准备和潜伏, 成功攻击了位于物理隔离内网中的工业控制系统, 迟滞了伊朗的核计划。2010 年的 Google Aurora(极光)攻击是另一个十分著名的 APT 攻击, Google 的一名雇员由于点击了即时消息中的一条恶意链接, 引发一系列事件, 从而导致这个搜索引擎巨人的网络被渗入数月, 造成各种系统的数据被窃取。2011 年 3 月, EMC 公司下属的 RSA 公司遭受入侵, 部分 SecurID 技术及客户资料被窃取, 结果导致很多使用 SecurID 作为认证凭

通讯作者: 韩言妮, 副研究员, Email: hanyanni@iie.ac.cn。
本课题得到中国移动内容分发网络二期工程扩容部分采购内容管理层(No.Y8V0211105)、青年之星人才计划(No.Y7Z0091105)资助。
收稿日期: 2018-07-26; 修改日期: 2018-09-19; 定稿日期: 2020-08-24

据建立 VPN 网络的公司受到攻击, 重要资料被窃取, 包括洛克希德马丁公司、诺斯罗普公司等美国国防外包商。2011 年 8 月份, McAfee/Symantec 发现并报告了“暗鼠攻击”, 该攻击在长达数年的持续攻击过程中, 渗透并攻击了全球多达 70 个公司和组织的网络, 包括美国政府、联合国、红十字会、武器制造商、能源公司、金融公司等等。2012 年的超级病毒“火焰”则成功获取了中东各国大量的机密信息。可见, APT 攻击已经对各类关键信息基础设施安全造成了巨大威胁, 开展 APT 攻击防御的工作刻不容缓。

由于 APT 攻击具有手段多样、长期潜伏的特点, 使得传统的防护手段在它面前见效甚微, 在这种情况下, 基于系统审计日志的入侵取证技术获得了安全社区的关注。入侵取证技术被广泛应用于还原攻击路径、定位攻击源头和评估攻击影响等方面。系统审计框架会在内核层面拦截系统调用, 并将系统调用信息以事件的形式记录下来, 最终形成一个完整的事件序列。该事件序列包含主机上发生的所有系统调用过程, 因此也包括潜在攻击者的所有操作痕迹, 这种特性使得系统审计日志非常适合入侵取证工作。常见的系统审计框架包括 Linux 系统下的 System Audit Framework 和 Windows 系统下的 Event Tracing Framework。

然而, 该方案在实际应用中还存在着较大的限制。

由于 APT 攻击潜伏时间较长, 攻击行动动辄长达数月, 为了尽可能完整地保留攻击痕迹, 企业需要维持至少半年以上的日志数据。此外, 企业网络的主机数量都比较庞大, 任意一台主机都可能成为潜在的突破口。为了做到全面防范, 企业必须为每一台主机长期地保存日志数据。

另一个挑战来自于系统审计日志本身。系统审计日志会详细记录每一次系统调用过程, 当主机负载较大时, 日志文件会以显著的速度增长。实际测试表明, 单台 Linux 主机在一天时间内产生的日志量可轻易达到 GB 级别。图 1 中所示分别为一台 WEB 服务器和客户端在一周时间内的日志增长情况(测试数据会随主机的负载情况而有所变化)。由图 1 可以看到, WEB 服务器上的日志文件以每天 5.09GB 的速度增长, 即使是客户端主机, 日志增长也达到了 2.12GB/天。当企业网络中的主机数目比较多时, 为每台主机维持长时间的日志存储将是一笔巨大的存储开销。

如何有效的减少日志大小, 同时完整保留攻击痕迹, 成为基于系统审计日志进行入侵取证亟需解决的问题。

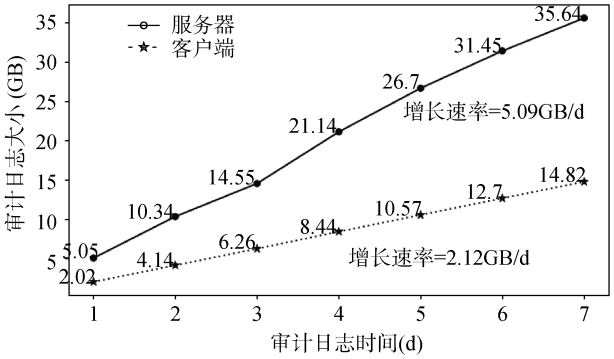

图 1 系统审计日志大小增长图

Figure 1 The growth of system audit log

考虑到 APT 攻击者都是从外部渠道潜入, 渗透到企业内网之后进行横向搜索和扩散, 直到攻陷高价值目标。在这个过程中, 更准确地说, 是在攻击者能够完全控制系统, 并可以关闭或者破坏审计功能之前, 攻击者的所有行动都将被系统审计日志如实地记录下来。与此同时主机内部的正常操作记录和内部信息流动也将被记录在案。对于入侵取证来说, 这些正常记录构成了冗余, 可以通过去除这些内部操作记录来实现系统审计日志的压缩。

为了在压缩过程中区分事件是否由外部数据导致, 本文借鉴动态污点分析的思想, 将由外部数据引起的事件标记为初始污点, 将系统调用过程视为主机内部的信息流动, 设计实现了一套基于污点追踪的日志压缩方案 T-Tracker, T-Tracker 的压缩结果只保留由外部数据引起的事件序列。

真实环境下的测试表明, 在不同的系统负载下, T-Tracker 平均可以达到 80%的压缩效果, 减少五倍的存储空间, 同时完整保留由外部依赖引起的可疑痕迹。

本文主要贡献总结如下:

(1) 结合 APT 攻击从外部渠道渗透的特点, 设计提出一套模仿动态污点分析的日志压缩方案。

(2) 编程实现原型系统 T-Tracker, T-Tracker 读取原始系统审计日志, 输出压缩后的事件序列。基于真实数据的测试表明, 借助污点追踪, 系统可以实现高达 66%的压缩效果, 如果结合进一步的优化, 最终的日志压缩率可以达到 80%, 显著降低了企业的存储压力。

(3) T-Tracker 的输出结果, 完整保留了攻击者的攻击痕迹, 从而支持更为快速准确的入侵检测和攻击取证工作。模拟多个攻击场景的测试表明, T-Tracker 对攻击痕迹的保留完整率均达到 100%。

论文其余组织结构如下: 第二部分介绍系统审

计日志压缩的相关工作; 第三部分介绍系统审计日志在入侵取证方面的应用, 并通过案例分析提出我们的压缩思路; 第四部分详细介绍了动态污点分析的基本原理以及如何改进该方案应用于系统审计日志压缩, 第五部分提出压缩方案原型系统 T-Tracker 的基本流程及其算法实现; 第六部分对 T-Tracker 系统进行实验验证, 第七部分总结本文的工作。

2 相关工作

在系统审计日志的压缩方面, Zhang Xu 等人[1]提出原始日志中存在很多重复记录, 这些记录具有相同的调用主体、客体和调用类型, 而且对主机造成的影响等价, 因此可以安全地进行合并, 据此实现了名为 CPR 以及 PCAR 的日志压缩算法。LogGC[2]提出原始日志中存在很多独立的、不会对主机内其他对象造成影响的“无用”记录, 能够被安全地剔除, 由此设计出一种仿照内存垃圾回收的日志压缩算法。ProTracer[3]提出了一套全新的系统审计框架: 在内核层面, 实现了一套轻量级的内核审计模块, 解决了 Linux audit 框架的性能问题; 同时在用户空间实现一种高效、可以并发处理的日志压缩算法, 在保证不丢失关键证据的前提下, 减少生成的日志大小。

以上方案尽管都致力于实现日志的压缩, 但均存在一定的局限性。ProTracer 重新设计日志采集和压缩框架, 尽管在压缩率方面表现出色, 但由于需要对内核进行改动, 限制了该方案的推广部署。LogGC 和 CPR 两种方案虽然基于现有的审计框架, 但它们仅仅根据某一事件是否影响主机状态, 来决定是否保留该事件, 而不管该事件是由外部数据引起, 还是主机自身的正常操作。考虑到 APT 攻击均来自于外部入侵, 日志记录中只有由外部数据引起的事件才有可能携带攻击痕迹。因此上述这两种方案, 在进一步去除冗余方面均存在改进的空间。

与本文相关的另一主题是动态污点分析 (Dynamic Taint Analysis, DTA)。DTA 是一种有效检测各种蠕虫攻击和自动提取特征码用于 IDS 和 IPS 的一系列解决方案。很多工作都围绕这个主题展开研究[4-7]。 James Newsome 和 Dawn Song 在文献[4]中, 提出了一种快速检测软件漏洞的动态污点分析方案, 命名为 TaintCheck, TaintCheck 将来自可疑来源的数据标记为污点, 动态追踪程序的执行过程和内存污点扩散过程, 从而发现可能的溢出漏洞以及恶意攻击。Argos[5]构造了一个针对蠕虫病毒和人为攻击的检测容器, 通过监测网络数据在程序执行中的恶意使用, 来发现和阻断攻击, 并生成用于入侵检测的特征签名。Edward J. Schwartz 等人[7]则致力于精确地定义动态污点分析算法, 并总结了将该方案用于典型安全场景时需要考虑的因素。

在系统依赖分析方面, 鉴于 APT 攻击的长期潜伏特点, 越来越多的研究聚焦在基于系统审计日志的取证分析和入侵检测[8-15]。BackTracker[9]借助系统审计日志, 通过回溯与可疑对象相关的事件序列来重现入侵过程。另一个工作[11]则是通过对隶属于不同进程的日志条目进行不同的着色, 从而实现对蠕虫病毒的高效检测。Shiqing Ma 等人[13]提出了一个基于 windows 事件追踪框架(ETW)的日志审计技术, 能够实现 windows 平台下的精确入侵检测。

其他工作[16-19]致力更加精确地捕捉因果依赖, 来缓解系统审计日志带来的依赖爆炸问题。BEEP[16]通过把程序处理流程划分为多个独立单元, 来精细化日志审计粒度, 以减少错误的依赖。Trail of Bytes[17]除了监控系统调用外, 还增加了对硬盘和内存监控, 从而更加精细地追踪数据访问过程, 其他研究工作则通过使用文件偏移[18]或者时间戳[19]来更准确地捕捉依赖关系。

在日志压缩方面, 若干工作借助图压缩技术来实现日志压缩, 其中包括结合网络图压缩和字典编码压缩实现的混合日志压缩[20], 还有工作通过合并共有子树来实现压缩目的[21]。本文的工作借鉴了这种思路, 通过将日志转化为依赖图的形式, 用节点表示系统内的进程、文件等对象, 用有向边表示系统对象间的信息流动, 提出了模仿污点追踪的压缩框架。

3 系统审计日志的规范化表达

在这一章, 我们首先介绍如何根据系统审计日志来绘制系统依赖图, 并借助系统依赖图, 对一个典型攻击场景进行案例分析, 呈现攻击过程中的依赖扩散路径和依赖图中的冗余信息。最后, 我们提出一种压缩系统依赖图的思路。

3.1 系统依赖图

系统审计日志记录了主机内部各个进程、文件、内存区域以及外部主机之间的交互关系, 并以事件 *event* 的形式保存下来。从对主机造成影响的角度来看, 一个事件的发生标志着主机内各个对象间的一次信息流动: 在时间段[*Ts, Te*]内, 主机通过执行特定的操作 *OP*, 导致信息从一个系统对象流向了另一个系统对象。除了常见的文件读写操作, 网络通讯操作、进程 fork 和程序执行操作等也会导致主机内部的信息流动。

本文借鉴 Zhang Xu 等人[1]的做法, 将系统审计日志转化为一种规范化的表达形式——*event*, 它可以用属性六元组进行表示: *event*={*pid, sub, obj, Ts, Te, OP*}。其中, *pid* 表示操作的进程号, *sub* 代表操作的主体, *obj* 表示操作的对象, *Ts* 和 *Te* 分别代表事件的开始时间和结束时间, *OP* 代表事件的操作类型。

在系统依赖图中, 一个 *event* 等价地转化为一条依赖边。依赖边的两端分别表示操作主体和操作对象, 边上方的文本表示操作类型及操作发生的时间段, 例如图 2 中的 E_{GH} 表示在时间段[2, 2]内操作主体 *G*(/usr/bin/unrealircd)改写了操作对象 *H*(/etc/unrealircd.tune)。通过将系统审计日志抽象为系统依赖图, 可以直观地了解主机内部发生的信息流动, 并据此回溯攻击源头或者评估攻击影响等等。

3.2 攻击分析示例

系统审计日志压缩的目的在于更好地支持入侵取证工作, 因此本文通过分析一个典型的入侵案例, 来证明系统审计日志中存在的冗余问题。本节中, 我们在模拟环境下重现了利用 vsftp 服务的后门实现入侵的过程, 然后依据系统审计服务记录下的审计日志, 绘制系统依赖图。

vsftp 服务是 Linux 环境下一个使用广泛的 FTP 服务, v 2.3.4 版本的 vsftp 源码被恶意植入后门, 该后门允许一个未知的入侵者进入核心代码。用户登录的时候, 如果在发送的用户名后面加上“:)”(笑脸符号), 该版本的程序会在 6200 端口上打开一个具有 root 权限的监听 shell。当攻击者通过 telnet 等工具登录后, 便可以远程执行任意命令, 例如添加普通用户等。

我们在局域网环境下重现了这个攻击过程, 其中靶机所在的 IP 地址为 192.168.192.3, 上面运行存在漏洞的 vsftp 服务, 攻击者从 IP 地址为 192.168.192.4 的主机发起入侵。在这个过程中, 运行在靶机上的系统审计服务, 会完整地记录整个入侵过程。根据系统审计日志, 我们绘制了该攻击事件的系统依赖图, 如图 2 所示。为了简单示意, 我们只绘制了部分正常事件的依赖图, 但保留完整的攻击过程, 攻击相关的依赖边用红色背景加以区分, 并用攻击发生的相对时间来代替实际时间。

从图 2 中可以看到, 系统审计日志除了记录来自外部主机的攻击过程, 还记录了主机自身的正常操作。例如节点 *K*(postgres 数据库服务)和节点 *H*(unrealircd 服务)相关的依赖边。在图 2 的依赖图中, 只有节点 *B*(vsftp 服务)接受了来自外部主机的数据, 因此攻击过程只可能包含在与节点 *B* 相关的

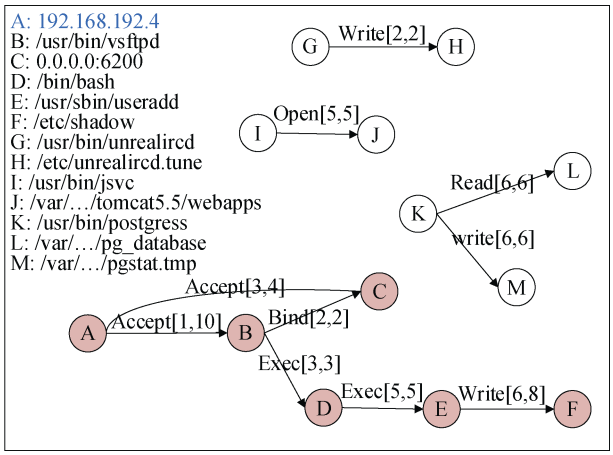

图 2 入侵 vsftp 服务的系统依赖图示例

Figure 2 The system causal graph of attacking on the vsftpd

依赖图中, 其余节点的依赖边均可以被安全移除, 以此达到压缩的目的。假设过了一段时间, 系统管理员发现了这个离奇增加的用户, 可以借助压缩过的系统依赖图, 从节点 *F*(/etc/shadow 文件)开始回溯, 通过还原图 2 中红色节点所示的攻击链, 最终定位攻击源头。

因此, 为了实现上述的日志压缩过程, 首先需要识别日志内部与外部数据来源发生的交互事件, 并追踪所有由这些事件引起的后续操作。此过程类似于二进制程序中的动态污点分析(DTA, Dynamic Taint Analysis)。因此下一章我们首先介绍动态污点分析的基本原理, 并讨论如何借鉴污点分析思路, 实现系统审计日志压缩。

4 动态污点分析理论基础

4.1 基本原理

动态污点分析是检测蠕虫攻击和二进制代码漏洞以及隐私信息泄露的有效方法之一。其流程主要分为三个部分: 污点标记和传播、污点消除和非法操作检测。

污点标记和传播: 主要原理是将来自于网络等不被信任的渠道数据都标记为“被污染”的数据, 由此产生的一系列算术和逻辑操作所新生成的数据, 也会继承源数据的“是否被污染”的属性。这一过程通过动态地更新某一内存地址或者寄存器对应的污点记录结构来实现。

污点消除: 污点数据在传播的过程中可能会经过无害处理模块, 无害处理模块是指污点数据经过该模块处理后, 数据本身不再携带敏感信息或者针对该数据的操作不会再对系统产生危害。换言之, 带

污点标记的数据在经过无害处理模块后, 污点标记可以被安全地移除。

非法操作检测: 一旦检测到被污染的数据作为跳转(jmp 族指令)、调用(call, ret)以及作为数据移动的目的地址, 或者是其他使 EIP 寄存器被填充为被污染数据的操作, 都会被视为非法操作。系统会报警并产生当前的相关内存、寄存器和一段时间内网络数据流的快照。同时服务会立刻终止或者在蜜罐环境中继续捕获进一步的入侵数据。

动态污点分析的主要实现过程是利用虚拟机技术, 对特定的指令进行特殊处理, 比如更新记录相应内存是否被污染的位图或者检查跳转是否安全。由此实现对外部可疑数据的跟踪和非法操作检测。

4.2 审计日志中的污点分析

类似于二进制程序内部的污点扩散过程, 在主机内部, 当污点信息从外部渠道进入主机后, 伴随着不同类型的系统调用, 污点信息会在主机内部对象之间扩散, 最常见的主机对象就是进程和文件。如果污点信息携带攻击意图, 那么它最终会触发一些高危操作, 达到入侵目的, 比如修改主机文件或者回传机密材料。系统审计日志会完整地记录整个污点扩散过程, 同时也会记录主机内部的正常信息流动, 我们可以通过污点追踪, 来区分受污染的和未受污染的信息流动, 从而达到日志压缩的目的。

但是, 系统审计日志中的污点追踪与经典动态污点分析在方式、监控对象和目的上都存在较大的不同。

首先是两者监控的方式不同: 动态污点分析是一个实时监控过程, 在程序运行时进行实时污点更新; 而 T-Tracker 基于日志信息进行污点追踪, 为了提高性能和降低对主机的影响, 本文采用滞后分析的方式, 在每天的 24 点, 主机相对空闲的时候, 读取这一天内产生的系统审计日志, 并进行压缩处理。

另一个显著的不同点是传播污点信息的载体。对某一程序而言, 动态污点分析中传播污点的载体只有一种, 就是数据流的传播, 因此只需维护被污染内存(以及寄存器)的集合即可; 对主机而言, 在 T-Tracker 中传播污点的载体分为两种: 进程和文件。例如, 进程 fork 操作导致污点在进程间的传播, 文件读写操作导致污点在进程和文件中的传播。因此在污点追踪过程中, 需要维护 *tProcs* 和 *tFiles* 两个污点集合, 分别对应受到污染的进程列表和受到污染的文件列表。

同时, 两者的追踪目的也不一样, 因此最终输出的追踪结果各不相同。动态污点分析用来检测恶意代码跳转, 所以只用输出漏洞触发瞬间的内存快照; 而 T-Tracker 的目的在于日志压缩, 并将之用于入侵取证, 所以需要保存污点扩散路径上的所有操作信息。除此之外, 由于 T-Tracker 只负责处理前一天内生成的日志, 污点追踪也仅限于前一天内发生的事件, 而主机内的信息流动却不存在周期与时间限制, 有可能刚好在零点前后发生持续的攻击行动。为了防止因此而导致的传播路径截断, 在 T-Tracker 处理结束后, 还需要把内存中维护的污点集合包括 *tProcs* 和 *tFiles* 写到硬盘上去, 在下一个处理周期, 据此初始化相应的污点集合, 恢复被人为截断的污点传播路径。

5 基于污点分析的日志压缩方案

接下来的这一章, 我们介绍 T-Tracker 系统的总体框架, 并详细讨论算法的实现细节。

5.1 T-Tracker 系统架构设计

T-Tracker 可以以代理的方式部署在每一台被监控主机上, 或者作为一台审计日志服务器部署在局域网中, 收集来自主机的日志记录。T-Tracker 默认以天为处理周期, 在每日零点, 运行压缩算法。首先将原始日志解析为 *event* 序列, 并根据开始时间从小到大对 *event* 进行排序, 随后进行污点追踪分析, 去除没有受到污染的 *event* 记录, 最终输出压缩后的日志。

系统的总体框架图如下图 3 所示。

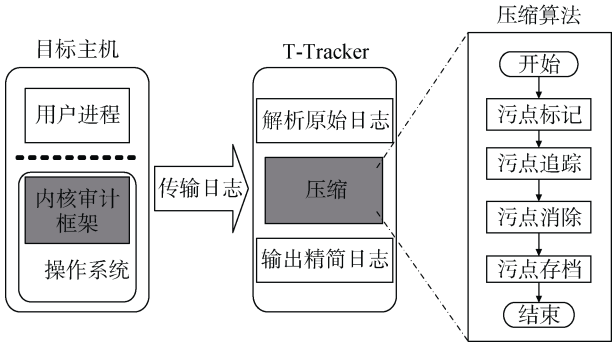

图 3 T-Tracker 框架图

Figure 3 Architecture of T-Tracker

由图可知, T-Tracker 的压缩算法分为四个处理单元: “污点标记”负责标记与外部数据来源发生交互的 *event*; “污点追踪”负责追踪污点扩散过程, 并去除污点扩散路径之外的 *event* 记录; “污点消除”进一步消除那些尽管受到污染但不会对主机安全状态造成影响的污点事件(对这些事件的详细讨论将在 4.3 节进行); “污点存档”负责在 T-Tracker 即将结

束运行时, 将内存中维护的污点信息写入到硬盘中, 当下一个周期的污点追踪开始时, 还原人为截断的扩散路径。

经过上述流程的处理后, T-Tracker 输出被外部数据影响的操作记录和一个维护污点信息的文本文件。

5.2 T-Tracker 处理流程

5.2.1 污点标记

在经典的动态污点分析中, 为了监控程序的运行内存, 常见的做法包括为内存空间的每一个字节建立一一对应的位图来标记其污染状态, 或者建立一个类似页表的结构, 为每一个字节维护一个指向存储该字节的污染信息的结构的指针。

然而上述的这些做法并不适合 T-Tracker。因为 T-Tracker 需要监控主机内部的所有文件与进程, 无法事先确定监控范围, 只能在追踪过程中动态地建立标记结构; 同时, T-Tracker 的目的在于保留污点扩散路径上的操作记录, 因此需要对中间过程进行标记, 而不能只维护被污染结果的集合。

因此本文采用标记 *event*, 同时维护污点集合的方式实现污点标记。具体来讲就是, 给 *event* 增加“*isT*”标志来表示事件是否被污染(*isT* 初始值为 0, 表示未受污染), 并动态地维护 *tProcs* 和 *tFiles* 两个污点集合。此时 *event* 属性增加为 7 个: *event* = *{pid,sub,obj,OP,Ts,Te,isT}*。当一个 *event* 发生从污点源的信息流动时, 我们把它的污点标志 *isT* 置 1, 代表 *event* 的 *obj* 对象受到污染(当受污染的 *event* 的操作类型为读操作时, 情况则不太相同, *isT* 标志置 1 并不代表被读的 *obj* 对象受到污染, 只表示读操作本身由外部主机引起, 可能携带恶意企图, 类似的操作还有文件删除操作)。

污点源分为两种情况: 一种是外部主机, 另一种是外部存储载体, 例如 U 盘等。

定义 1. 初始污点的标记

当 *event* 满足如下条件之一时, 我们把 *event.isT* 标志置 1, 并把 *event.pid* 添加到 *tProcs* 中:

(1) *event.obj* = *InetAddr*, *OP* ∈ socket; 其中 *InetAddr* 表示外部主机的 IP 地址, *OP* ∈ socket 表示操作类型属于 socket 函数簇, 包括 socket 的 creat, accept 和 receive 操作等等。

(2) *event.obj* ∈ *FileInU-disk*, *OP*=read; 表示该 *event* 从外部存储载体中读取了信息, 导致进程受到污染。

5.2.2 污点追踪

主机内部的进程会与主机上的文件、外部主机以及其他进程之间发生各种交互, 并产生各种各样的系统调用。在所有的系统调用之中, 只有部分调用会发生污点信息的扩散, 比如文件读写操作、网络通讯操作、进程 fork 操作和程序执行操作。除此之外, 其他会修改主机文件的操作如删除、重命名等, 也会对污点扩散过程造成影响。我们只对这些会影响主机内部依赖扩散的调用进行讨论。

根据系统调用对污点扩散造成的影响不同, 可以把系统调用分为三种类型:

表 1 三种不同类型的系统调用

Table 1 Three different types of system calls

调用类型	系统调用
Input	file read, socket receive. etc.
Output	file write, socket send, fork, execve. etc.
Dead-end	file deletion, process exit. etc.

第一种类型的系统调用会导致信息从 *obj* 流入 *sub*; 第二种类型的调用会导致信息从 *sub* 流出到 *obj*; 第三种类型的调用则会终止依赖的扩散, 并导致对应污点进程或文件的剔除。关于第三类调用, 我们需要注意, 尽管这些操作会终止依赖的进一步扩散, 但是如果这些操作由外部数据导致, 这些操作本身可能含有恶意企图。比如攻击者恶意删除主机上其他用户的文件, 尽管文件被删除, delete 事件本身仍然需要保留。

因为主机内部的污点扩散是随着时间的推移而进行的, 所以我们将事件按照开始时间从小到大进行排列, 然后依次进行污点检测, 从而可以在污点传递的过程中, 只考虑调用类型的影响, 而不用考虑时间因素。

初始列表中, 我们把所有事件的 *isT* 标记都置为 0, 当检测到发生来自污点源的信息流入时, 我们把该标志置 1。随后, 进行污点的扩散追踪, 当系统调用匹配上一小节列出的三种类型时, 我们采用相应的措施更新污点集合。

如下算法 1 简单地描述了系统审计日志中的污点追踪。算法中的 Input-OP, Output-OP, Dead-end-OP 分别对应上述讨论的三种类型的系统调用。需要注意的是, 当事件的调用类型属于 Output-OP 和 Dead-end-OP 时, 对 *tProcs* 和 *tFiles* 的更新取决于特定的调用类型。比如当某个事件输出污点信息到另一个系统实体时, 它可能是受到污染的进程写出到一个普通文件, 也可能是受到污染的父进程克隆了一个子进程。对应地, 采取的更新操作可能是把新被污染的文件添加到 *tFiles*, 也可能是把继承污点依赖

的子进程添加到 *tProcs*。

算法 1: 系统审计日志中的污点追踪算法

输入： *E* - 原始事件列表(按照 Ts 进行排序, 并且 event.isT=0)

输出： *T* - 受到污染的事件列表

```
BEGIN:
FOR e IN E:
IF e.OP ∈ Input AND e.src ∈ tFiles THEN
e.isT <-- 1
tProcs.add(e.pid)
ELSE IF e.OP ∈ Output AND e.pid ∈ tProcs THEN
e.isT <-- 1
tFiles.add(e.obj) 或者
tProcs.add(e.pid)
ELSE IF e.OP ∈ Dead-end AND e.pid ∈ tProcs THEN
e.isT <-- 1
tProcs.remove(e.pid) 或者
tFiles.remove(e.obj)
ELSE IF e.pid ∈ tProcs THEN
e.isT <-- 1
END IF
END FOR

FOR e IN E:
IF (e.isT == 1) THEN
T.add(e)
END IF
END FOR
```

经过以上的污点追踪处理, 只有受到外部数据影响的事件被保留下来, 剩余的事件均被认为是系统内部的正常信息流动, 而被安全地移除, 以此达到对原始审计日志的压缩。值得注意的是, 此处的压缩, 侧重于对冗余日志事件的剔除, 而不仅仅是传统意义上对日志存储空间的压缩。

5.2.3 污点消除

经过前述模块的处理后, 剩余的记录虽然都受到外部数据的影响, 但仍存在部分不会对主机安全状态造成影响的情况。比如 linux 系统下的 proc 文件夹, 只是用来存储进程的独立信息, 而不会与别的进程发生交互, 当进程退出时, 对应的文件便会被删除。因此, 系统审计日志中与 proc 目录下面的文件交互的记录可以被安全地移除。此外很多程序在运行的时候, 都会把一些无用的输出信息写入到 /dev/null 文件里面。类似于上面这些样例, 我们可以根据操作系统相关的领域知识, 设定文件白名单, 与名单里面的文件交互的日志记录不可能携带攻击意图, 可以被安全地移除。

另一类不会对主机安全状态造成影响的因素来自于与临时文件的交互。

临时文件被定义为仅由同一个进程创建、读写, 并且删除的文件。在临时文件的生命周期中, 除了创建它的进程以外, 与其他主机对象均没有发生信息的交换。主机内的很多服务程序都会频繁的创建并删除临时文件。虽然临时文件也可能受到污染, 但是污染信息并不会继续扩散, 而是随着文件的删除而消亡, 因此围绕临时文件的所有事件均可以安全地被移除。

在下一章的实际测试中, 我们发现在 Taint tracking 处理后的结果中, 结合上述两种领域知识, 可以进一步去除 10.23%的冗余记录。

5.2.4 污点存档

在经典的动态污点分析中, 程序的退出, 就标志着污点扩散过程的结束。但是主机内的信息流动, 只要主机没有关机, 就一直在进行。因此当一个 T-Tracker 的处理周期结束后, 我们需要在硬盘上保存内存中维护的污点集合, 具体的讲就是 *tProcs* 和 *tFiles* 两个集合, 并在 T-Tracker 下一次运行时, 把这些对象也视为污点源。

T-Tracker 经过 Taint tagging、Taint tracking 和 Taint erasing 的处理后, 在结束处理周期之前, 还需要进行污点的存档工作, 我们把这个过程叫做 Taint logging。

6 实验验证

为了评估本文提出的设计的有效性, 我们用 JAVA 语言实现了原型系统 T-Tracker, 并在真实网络环境下, 对系统的效果进行测试。系统效果包括三个方面: 日志压缩比率、对攻击痕迹的保留效果、算法的时间空间性能。

6.1 实验环境设置

为了全面评估 T-Tracker 的压缩效果, 我们选择了局域网环境下的 6 台主机。这 6 台主机分为服务器和客户端两组, 其中 Client 1~4 充当客户端, 用于文档编辑和浏览网页; Server 1~2 充当服务器, 用于对局域网内提供 web 服务、FTP 服务和数据库服务。6 台主机位于同一个局域网内, 并通过出口路由器连到外部互联网。于此同时, 局域网内部还部署有一台模拟的攻击机(Attacker)和一台目标靶机(Target Server)用于之后的模拟攻击实验。如下图 4 所示为实

验环境网络拓扑图。

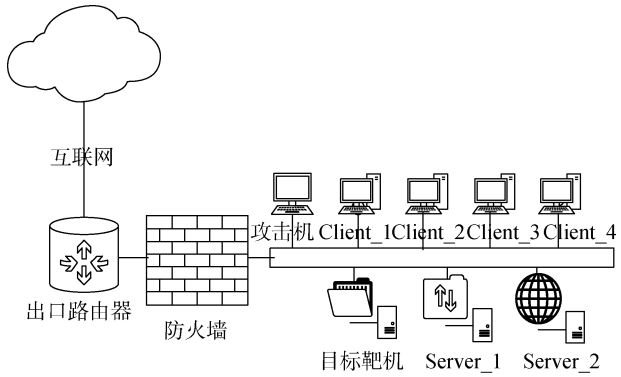

图 4　实验环境网络拓扑图

Figure 4　Network topology of experimental environment

6.2　日志压缩比率

首先，我们在长达一个月的时间内，对 T-Tracker 的压缩性能进行评估。Client 1~4 和 Server 1~2 总共 6 台主机组成了测试平台。为了降低对网络传输的要求(但是牺牲了部分安全性能)，我们采用代理式的部署方式，每台主机都运行相应的日志采集客户端和压缩程序。

表 2 描述了一个月内，6 台主机上采集的原始日志的统计信息。评估期间，我们总共采集了超过 600GB 的原始日志，总共监测到将近 6 千万条的系统调用事件。

表 2　6 台主机在一个月内的原始日志统计

Table 2　Original log statistics of 6 hosts in one month

主机	日志总量 (GB)	事件总数	单日平均日志量	单日平均事件数
Client_1	29.4843	2463026	0.9828	82101
Client_2	64.0355	6289887	2.1345	209663
Client_3	79.1972	7404989	2.6399	246833
Client_4	59.4278	5668423	1.9809	188947
Server_1	262.0415	24772595	8.7347	825753
Server_2	130.6534	11458377	4.3551	381946
Total	624.8397	58057297	20.8279	1935243

与此同时，为了对两组主机的日志构成有一个全面的了解，我们分别对客户端主机和服务器主机的系统负载进行了一次切面分析。简单起见，我们选择具有典型性的 Client_3 和 Server_2 来分别代表客户机和服务器。

我们将主机的系统负载大致分成了五类: a、日常办公，包括文档编辑和代码开发等等; b、系统功能，代表系统自带的软件功能，比如 Linux 系统下的 ls、mv、cp 等常用命令和 Windows 系统下的控制面板、计算器等功能; c、系统服务进程，代表一类在后台运行的特殊进程，用于执行特定的系统任务; d、通信软件，包括电子邮件、浏览器和即时通讯软件等等; e、用户服务端程序，比如 WEB 服务器、数据库程序和 FTP 服务器等。

图 5 所示，为 Client_3 和 Server_2 的负载分布情况。从中可以得知，文档办公和通信软件构成了 Client_3 的主要负载; 而对于 Server_2 来说，日志的大部分由用户服务端程序和系统服务进程的记录构成。

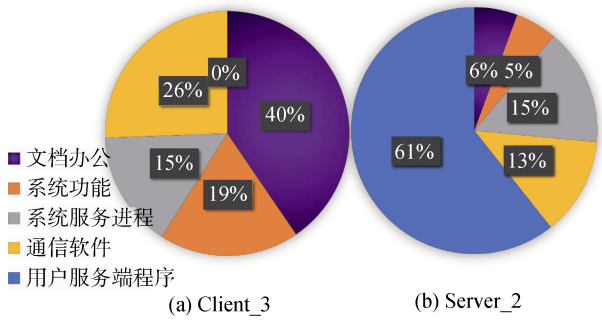

图 5　Client_3 和 Server_2 的负载分布图

Figure 5　Load distribution of Client_3 and Server_2

T-Tracker 在每日 0 点运行压缩算法，输出压缩日志的同时生成关于压缩率指标的报告。经过长达一个月的评估后，我们计算了 T-Tracker 在 6 台主机上的平均压缩率，结果如表 3 所示。其中，用缩写 CR 表示压缩比率(compression ratio)，Track_CR 表示污点追踪贡献的压缩比率，Domain_CR 表示借助领域知识完成的压缩效果，Total_CR 表示总的压缩比率，其值等于 Track_CR 和 Domain_CR 之和。

表 3　一个月内 6 台主机的平均压缩率

Table 3　Average compression ratios for six hosts in one month

主机	Track_CR (%)	Domain_CR (%)	Total_CR (%)
Client_1	41.80	18.35	60.15
Client_2	84.63	2.79	87.42
Client_3	61.16	14.17	75.33
Client_4	83.52	0.71	84.23
Server_1	58.72	0.008	58.73
Server_2	10.21	25.34	35.55
Average	56.67	10.23	66.90

由表 3 的结果可知，全部 6 台主机的平均压缩率达到 66.90%，其中 Taint tracking 贡献了里面的 56.67%，操作系统领域知识进一步提高了 10.23%的

压缩率。结果表明, T-Tracker 可以帮企业减少三分之二的日志存储成本。

同时, 从表 3 可以看出, 服务器主机的压缩率显著低于客户端主机的压缩率, 严重拖累了平均压缩效果。这是因为充当服务器的主机需要不间断地对外提供服务, 网络业务占据日志的主要组成部分。根据 T-Tracker 的压缩原理容易得知, 这些记录都会被保留下来, 因此导致较低的压缩率。最明显的是 Server_2 主机, Taint tracking 模块只实现了 10.21%的压缩率, 这与上述讨论的 Server_2 主机的负载分布结果相一致。然而, 在接下来的讨论中, 我们将通过进一步的改进来提高 T-Tracker 的压缩性能。

6.3 进一步提高压缩率

在第一章中, 我们提到, Zhang Xu 等人[1]提出了一种通过合并冗余重复记录, 来达到压缩目的的 CPR+PCAR 算法。由于 T-Tracker 采用了与之完全不同的压缩思路, 因此两者可以结合起来达到更好的压缩效果。接下来, 我们通过实验数据来验证这种思路。

首先, 我们在 6 台主机上分别单独运行 T-Tracker 和 CPR+RCAR 算法, 统计 6 台主机在一周时间内的单日平均压缩率, 结果如下图 6 柱形图所示。蓝色柱形图表示 T-Tracker 的压缩率, 橘色柱形图表示 CPR+PCAR 的压缩率。

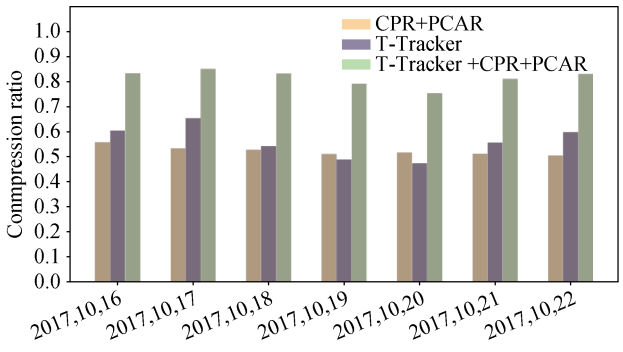

图 6 T-Tracker 与 CPR 的压缩效果比较图

Figure 6 Comparison of T-Tracker and CPR

可以看到, 单独借助污点追踪, T-Tracker 的表现略胜出 CPR+PCAR; 但是 CPR+PCAR 的压缩率保持稳定, 而 T-Tracker 的压缩率波动比较大, 这与上节的讨论相一致。

接下来, 我们在 T-Tracker 压缩的结果之上, 进一步使用 CPR+PCAR 算法进行压缩, 统计一周时间内的单日平均压缩率, 结果如图 5 中绿色柱形图所示。可以看到, 两者结合的平均压缩率稳定在 80%左右。换句话说, 结合 T-Tracker 和 CPR+RCAR 算法, 可以减少企业 5 倍的存储成本。除了降低存储成本外, 更为重要的是, 企业将能够借助精简的系统审计日志, 实现更为快速准确的取证分析。

6.4 攻击痕迹完整性分析

T-Tracker 的压缩结果用于对 APT 攻击的取证分析, 所以输出的结果需要支持后续的依赖分析, 不能丢失攻击证据。

为了评估 T-Tracker 对攻击痕迹的保留效果, 我们在实验环境中模拟了五种典型的攻击场景, 来进行检测。首先选用局域网中的一台服务器充当靶机, 上面运行不同的存在漏洞的服务程序, 并用另一台客户机充当攻击机, 混在正常的网络流量中, 进行远程入侵。其中充当靶机的服务器上面运行系统审计服务, 会完整记录整个攻击过程中的系统调用。

场景一利用 2.3.4 版本的 vsftpd 服务的后门漏洞。该后门允许远程攻击者无需登录即可获得 root 权限的 shell, 并执行任意命令, 如执行 useradd 命令添加用户。详细讨论见 3.2 节攻击案例分析。

场景二利用 Samba 文件共享服务的目录遍历漏洞(CVE-2010-0926[22])进行文件窃取。当把共享文件的权限配置为可写, 同时允许“wide links”时(默认就是“允许”), 将使得远程的认证用户可以越权访问任意主机文件。

场景三针对 unreal_ircd 服务的后门漏洞进行攻击(CVE-2010-2075[23])。unreal_ircd 是一个开源的 IRC 服务器, 用于实现基于网络的群体聊天。攻击者可以利用该后门漏洞获取一个交互 shell, 并在系统上下载恶意软件。

场景四利用分布式编译环境 distcc 2.x 的远程命令执行漏洞(CVE-2004-2687[24]), 该漏洞允许攻击者远程执行任意命令。比如, 充当跳板扫描其他主机。

场景五利用 web 服务器的目录遍历漏洞。当 web 服务器进行了错误的配置, 并对客户端提交的 URL 不设置过滤时, 可能造成恶意构造的 URL 访问服务器文件系统。假设名为“hello”的 web 项目位于目录“/*WEB-PATH*/hello/”下面, 系统账号密码文件为“/etc/shadow”, 如果服务器错误配置的话, 攻击者可以通过构造 url=“http://*SERVER-IP*/../../etc/shadow”来非法访问 shadow 文件。

每一个攻击场景, 我们分别根据原始日志和压缩后的日志生成系统依赖图, 用于之后的入侵取证。假定攻击发生一段时间后, 系统管理员发现了可疑症状, 并从攻击症状开始往后回溯, 以寻找攻击源头。

以场景三为例, 假设当主机上的“unreal_ircd”服务被攻陷后, 攻击者企图下载恶意软件到主机上。为了简单起见, 我们执行“wget www.baidu.com”命令来模仿这一过程。由此导致主机上新增了一个名为“index.html”的文件。过了一段时间后, 系统管理员发现了这个可疑的文件, 并尝试根据系统审计日志回溯攻击源头。如下图 7 和图 8 所示分别为根据原始审计日志和压缩后审计日志绘制的系统依赖图。

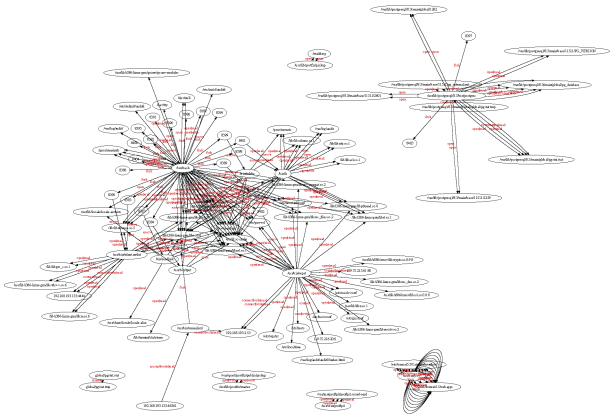

图 7　根据原始日志绘制的系统依赖图

Figure 7　System causal graph based on original log

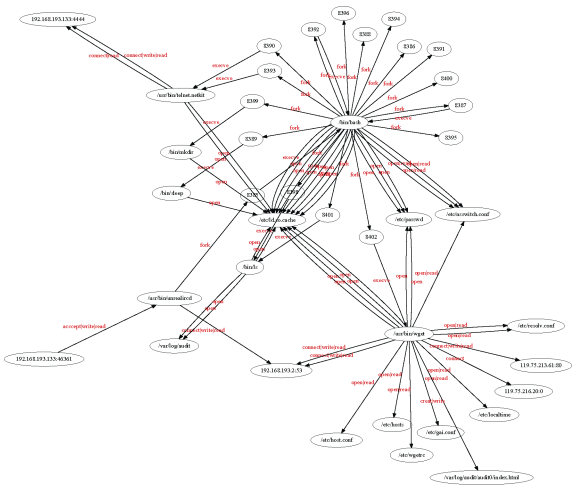

图 8　根据压缩后审计日志绘制的系统依赖图

Figure 8　System causal graph based on compressed log

对比图 7 和图 8 可以发现, 经过 T-Tracker 的压缩后, 系统依赖图变得更加精简。除了未受到外部数据污染的依赖子图被移除外, 剩余的包含可疑事件的依赖子图也变得更加简单明了。不难理解, 被简化的部分分别来自 T-Tracker 的污点追踪算法, 和结合 CPR+PCAR 的进一步改进。

根据精简的系统依赖图, 我们可以快速地进行取证分析, 来回溯攻击源头。图 9 描述了依据简化后的系统依赖图, 进行回溯分析的结果示意(实际上, 正如下面表 5 所证实的, 图 9 的结果与根据原始的系统依赖图进行回溯的结果完全一致)。图 9 完整地描述了攻击者利用“unreal_ircd”程序的漏洞(CVE-2010-2075[23])进行入侵的路径。

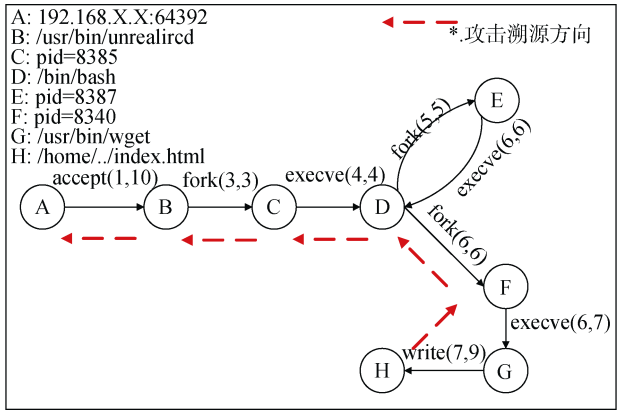

图 9　场景三的攻击路径回溯结果

Figure 9　Attack path backtracking results of scenario 3

为了证实 T-Tracker 是否能完整的保留攻击痕迹, 在所有五个典型攻击场景下, 我们分别统计了原始日志和压缩之后日志中与攻击相关的事件总数, 并进行了对比。因为原始日志记录了攻击过程中的所有系统调用, 从而包含了所有的攻击痕迹, 因此可以参照此基准值, 来评估 T-Tracker 对攻击痕迹的保留完整性。

统计结果如下表 4 所示。其中, 第二列和第三列分别表示原始日志和压缩后日志中的全部事件总数, 第四列和第五列分别表示原始日志和压缩后日志中与攻击相关的事件总数。最后一列表示攻击痕迹保留完整率, 也就是用压缩后日志中与攻击相关事件总数除以原始日志中攻击相关事件总数。

表 4　五种典型攻击场景下的攻击痕迹完整率

Table 4　Attack trace integrity rate in five typical attack scenarios

攻击场景	Tb	Ta	Rb	Ra	Ra/Rb (%)
vsftp (*backdoor*)	253	57	15	15	100
samba (*access rootFS*)	149	44	4	4	100
unreal_ircd (*download*)	155	54	8	8	100
distcc (*remote exec*)	146	55	12	12	100
web (*file stealing*)	98	31	4	4	100

表 4 结果说明, T-Tracker 在全部五个攻击场景下, 都完整地保留了攻击痕迹, 从而为 T-Tracker 应用于

实际入侵取证提供了数据支持。

6.5 性能分析

算法的性能分时间性能和空间性能两个方面。我们选用 6 台主机中的一台服务器主机进行测试。被测主机运行 linux 2.6.32 内核, 硬件环境包括 8G 内存, 并搭载 Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz。

为了测试系统的运行耗时与日志大小的关系, 我们选取大小分别为 100MB, 1GB, 10 GB 的日志文件, 运行 T-Tracker 算法。然后统计系统运行耗时, 结果如表 5 所示。表格第二列表示原始日志解析时间, 第三列表示运行 T-Tracker 压缩算法的时间, 第四列为总的时间消耗。

表 5 算法的时间消耗随日志大小变化统计
Table 5 T-Tracker's run-time overhead under different log size

日志大小	解析时间(ms)	污点追踪时间(ms)	总时间消耗(ms)
100MB	4172	84	4256
1GB	31224	315	31539
10GB	385378	2152	387530

由表 5 可知, 原始日志的解析时间占据了总时间消耗的主要部分, T-Tracker 压缩算法的运行耗时占比非常小; 同时系统总运行耗时与日志大小保持线性增长。同样, 我们用上述方案对 T-Tracker 算法的空间消耗进行了统计。

表 6 T-Tracker 内存消耗随日志大小变化情况
Table 6 T-Tracker's memory consumption under different log size

日志大小	虚拟内存(M)	物理内存(M)
100MB	3195	618
1GB	3195	693
10GB	3195	778

由表 6 可知, 在不同的日志大小情况下, T-Tracker 的虚拟内存消耗稳定在 3GB。当日志大小以 10 倍的速度增长时, 物理内存仅仅以不超过 100MB 的速度缓慢增长。因此表现出了优异的空间性能。即使当原始日志达到 10GB, T-Tracker 的物理内存消耗也不到 1 个 GB。这种空间消耗即使对于普通的主机而言, 都是可以接受的。

7 讨论

本文工作的威胁模型基于如下三个假设——第一、操作系统和系统审计框架构成一个可信计算基。攻击者通过入侵被系统审计框架保护的应用软件和系统资源来达到恶意目的。比如在操作系统上安装恶意软件, 利用正在运行的进程的漏洞, 或者植入软件后门, 等等。同时, 为了实现完整的攻击取证, 我们假设所有攻击都发生在系统审计服务开始运行之后, 也就是说从攻击刚一开始, 所有的操作都被完整地记录在案。关于硬件木马和操作系统脆弱性的防御超出了我们的讨论范围。

第二, 尽管攻击者可能攻陷操作系统以及运行其上的系统审计服务, 我们假设攻击者并没有能力篡改系统审计服务的历史记录。当然, 在攻击者取得系统控制权之后, 所有的审计记录将不再可靠。我们有三种方式来实现这种假设: ①在另一台经过强化的日志服务器上实时备份审计日志。②通过部署安全的日志取证技术[25-26]来确保数据完整性。③通过现有的完整性检测机制[27, 31], 系统可以在日志被恶意篡改时, 发出告警。

最后一个假设是攻击者只使用显式的攻击通道, 而不使用超出本文范围的旁路和侧信道攻击。

8 结论

本文提出了一种新颖的模仿动态污点分析的系统审计日志压缩方案。通过将外部数据导致的事件标记为污点, 实现了日志范围内的污点追踪, 据此剔除攻击无关的日志记录。实验环境下的测试表明, 借助污点追踪, T-Tracker 可以实现 66%的压缩效果; 如果再结合进一步的优化, 可以达到 80%的压缩效果, 显著降低企业的存储计算成本。

T-Tracker 的压缩结果完整保留了由外部数据导致的可疑事件。在五个典型场景下的攻击测试中, T-Tracker 的攻击痕迹保留完整率都达到了 100%。T-Tracker 的输出结果, 包含了可疑数据从进入主机到造成破坏的完整过程, 可以支持入侵取证系统更为快速精确地定位攻击源头, 还原攻击路径。

参考文献

[1] Xu Z, Wu Z Y, Li Z C, et al. High Fidelity Data Reduction for Big Data Security Dependency Analyses[C]. *the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 504-516.

[2] Lee K H, Zhang X Y, Xu D Y. LogGC: Garbage Collecting Audit Log[C]. *the 2013 ACM SIGSAC conference on Computer & communications security*, 2013: 1005-1016.

[3] Ma S., Zhang X., Xu D. ProTracer: Towards Practical Provenance

Tracing by Alternating Between Logging and Tainting[C]. *In NDSS*, 2016: 285-260.

[4] Newsome J., Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[C]. *USENIX Security Symposium*, 2004: 321-336.

[5] Portokalidis, G., Slowinska, A, Bos, H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation[C]. *ACM SIGOPS Operating Systems Review, ACM*, 2006: 15-27.

[6] Clause J, Li W C, Orso A. Dytan: A Generic Dynamic Taint Analysis Framework[C]. *the 2007 international symposium on Software testing and analysis*, 2007: 196-206.

[7] Schwartz, E. J., Avgerinos, T., Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)[C]. *Security and privacy (SP), IEEE symposium*, 2010: 317-331.

[8] Ammann P, Jajodia S, Liu P. Recovery from Malicious Transactions[J]. *IEEE Transactions on Knowledge and Data Engineering*, 2002, 14(5): 1167-1185.

[9] King S T, Chen P M. Backtracking Intrusions[J]. *ACM SIGOPS Operating Systems Review*, 2003, 37(5): 223-236.

[10] King S T, Mao Z M, Lucchetti D G, et al. Enriching Intrusion Alerts through Multi-Host Causality[J]. *Ndss*, 2005, 05: 1-13.

[11] Jiang X., Walters A. Xu, D. Spafford, et al. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach[C]. *Distributed Computing Systems, ICDCS*, 2006: 38-38.

[12] Goel A, Feng W C, Feng W C, et al. Automatic High-performance Reconstruction and Recovery[J]. *Computer Networks*, 2007, 51(5): 1361-1377.

[13] Ma S Q, Lee K H, Kim C H, et al. Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows[C]. *the 31st Annual Computer Security Applications Conference*, 2015: 401-410.

[14] Tariq D., Ali M., Gehani A. Towards Automated Collection of Application-Level Data Provenance[C]. *In TaPP*, 2012: 258-259.

[15] Chow J, Pfaff B, Garfinkel T, et al. Understanding Data Lifetime via Whole System Simulation[C]. *USENIX Security Symposium*, 2004: 321-336.

[16] Lee K. H., Zhang X., Xu D. High Accuracy Attack Provenance via Binary-based Execution Partition[C]. *InNDSS*, 2013: 268-270.

[17] Krishnan S, Snow K Z, Monrose F. Trail of Bytes: Efficient Support for Forensic Analysis[C]. *the 17th ACM conference on Computer and communications security*, 2010: 50-60.

[18] Sitaraman S., Venkatesan S. Forensic analysis of file system intrusions using improved backtracking[C]. *In Information Assurance*, 2005: 154-163.

[19] Goel A, Po K, Farhadi K, et al. The Taser Intrusion Recovery System[J]. *ACM SIGOPS Operating Systems Review*, 2005, 39(5): 163-176.

[20] Xie Y L, Feng D, Tan Z P, et al. A Hybrid Approach for Efficient Provenance Storage[C]. t*he 21st ACM international conference on Information and knowledge management*, 2012: 1752-1756.

[21] Chapman A P, Jagadish H V, Ramanan P. Efficient Provenance Storage[C]. *the 2008 ACM SIGMOD international conference on Management of data*, 2008: 993-1006.

[22] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0926.

[23] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2075.

[24] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2687.

[25] Bates A, Butler K, Haeberlen A, et al. Let SDN be your Eyes: Secure Forensics in Data Center Networks[C]. Proceedings 2014 Workshop on Security of Emerging Networking Technologies, 2014: 236-240.

[26] Zhou W C, Fei Q, Narayan A, et al. Secure Network Provenance[C]. *the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011: 295-310.

[27] Hofmann O S, Dunn A M, Kim S, et al. Ensuring Operating System Kernel Integrity with OSck[C]. *the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011: 279-290.

[28] Lee H, Moon H, Heo I, et al. KI-Mon ARM: A Hardware-Assisted Event-triggered Monitoring Platform for Mutable Kernel Object[J]. *IEEE Transactions on Dependable and Secure Computing*, 2019, 16(2): 287-300.

[29] Moon H, Lee H, Lee J, et al. Vigilare: Toward Snoop-based Kernel Integrity Monitor[C]. *the 2012 ACM conference on Computer and communications security*, 2012: 28-37.

[30] Fraser T., Molina J., Arbaugh, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor[C]. *Conference on Usenix Security Symposium,* 2004: 13.

[31] Seshadri A., Luk M., Qu N., et al. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes[C]. *ACM Sigops Symposium on Operating Systems Principles,* 2007: 335-350.

贲永明 于 2015 年在北方工业大学电子信息工程专业获得学士学位。现在中国科学院信息工程研究所计算机系统结构专业攻读博士学位。研究领域为计算机系统安全。研究兴趣包括: 网络攻防、终端安全。Email: benyongming@iie.ac.cn

韩言妮 于 2010 年在北京航空航天大学计算机软件与理论专业获得博士学位。现任中国科学院信息工程研究所第五研究室副研究员。研究领域为 SDN 网络、网络安全。研究兴趣包括: 安全防御、新型网络架构。Email: hanyanni@iie.ac.cn

安伟 于 2012 年在华东理工大学控制科学与工程专业获得工学博士学位。现任中国科学院信息工程研究所第五研究室助理研究员。研究领域为网络与系统安全。研究兴趣包括: 网络优化、网络安全及防护、5G 移动通信与安全。Email: anwei@iie.ac.cn.

徐震 于 2005 年在中国科学院软件研究所信息安全专业获得工学博士学位。现任中国科学院信息工程研究所第五研究室正高级工程师。研究领域为网络体系结构与安全防护。研究兴趣包括云安全、可信计算。Email: xuzhen@iie.ac.cn