软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(2):405−420 [doi: 10.3724/SP.J.1001.2013.04265]

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

基于 VMM 的操作系统隐藏对象关联检测技术*

李 博, 沃天宇, 胡春明, 李建欣, 王 颖, 怀进鹏

(北京航空航天大学 计算机科学与技术系,北京 100191)

通讯作者: 李博, E-mail: libo@act.buaa.edu.cn, http://scse.buaa.edu.cn

摘 要: 恶意软件通过隐藏自身行为来逃避安全监控程序的检测.当前的安全监控程序通常位于操作系统内部,难以有效检测恶意软件,特别是内核级恶意软件的隐藏行为.针对现有方法中存在的不足,提出了基于虚拟机监控器(virtual machine monitor,简称 VMM)的操作系统隐藏对象关联检测方法,并设计和实现了相应的检测系统 vDetector.采用隐式和显式相结合的方式建立操作系统对象的多个视图,通过对比多视图间的差异性来识别隐藏对象,支持对进程、文件及网络连接这 3 种隐藏对象的检测,并基于操作系统语义建立隐藏对象间的关联关系以识别完整攻击路径.在 KVM 虚拟化平台上实现了 vDetector 的系统原型,并通过实验评测 vDetector 的有效性和性能.结果表明,vDetector 能够有效检测出客户操作系统(guest OS)中的隐藏对象,且性能开销在合理范围内.

关键词: 虚拟化;虚拟机监控器;隐藏对象;多视图;关联检测

中图法分类号: TP316 **文献标识码**: A

中文引用格式: 李博,沃天宇,胡春明,李建欣,王颖,怀进鹏.基于 VMM 的操作系统隐藏对象关联检测技术.软件学报,2013,24(2):405−420. http://www.jos.org.cn/1000-9825/4265.htm

英文引用格式: Li B, Wo TY, Hu CM, Li JX, Wang Y, Huai JP. Hidden OS objects correlated detection technology based on VMM. Ruanjian Xuebao/Journal of Software, 2013,24(2):405−420 (in Chinese). http://www.jos.org.cn/1000-9825/4265.htm

Hidden OS Objects Correlated Detection Technology Based on VMM

LI Bo, WO Tian-Yu, HU Chun-Ming, LI Jian-Xin, WANG Ying, HUAI Jin-Peng

(School of Computer Science and Engineering, BeiHang University, Beijing 100191, China)

Corresponding author: NAME N, E-mail: libo@act.buaa.edu.cn, http://scse.buaa.edu.cn

Abstract: To evade the detection of security monitoring systems, malware often hides its behavior. Current monitoring systems usually reside in the operating system (OS). Thus, it is hard to detect the existence of malware, especially the kernel rootkits. In this paper, a hidden OS objects detection and correlation approach based on VMM (virtual machine monitor) is proposed, and the corresponding detection system, vDetector, is designed and implemented. Both implicit and explicit information are used to create multiple views of OS objects, and a multi-view comparison mechanism are designed to identify three kinds of hidden OS objects: process, file and connections. The relations among hidden objects are established based on OS semantic information to trace the complete attack path. vDetector is implemented based on KVM virtualization platform and the effectiveness and performance overhead of vDetector are evaluated by comprehensive experiments. The results show that vDetector can successfully detect the existence of hidden OS objects with reasonable performance overhead.

Key words: virtualization; VMM; hidden object; multi-view; correlated detection

当前,恶意软件呈现出隐蔽性和伪装性等特点,给安全软件的检测带来了极大的困难.恶意软件通常驻留在操作系统内部,为了消除自己的攻击痕迹并达到持续控制被入侵主机的目的,通常都在被入侵主机中预留后门,如建立监听进程和网络连接等.为了隐藏自身的恶意行为,恶意软件通过隐藏进程、网络连接以及文件等方式

* 基金项目: 国家自然科学基金(61202424, 60903149, 91018008); 国家重点基础研究发展计划(973)(2011CB302600)
收稿时间: 2012-02-27; 定稿时间: 2012-04-26

来逃避安全软件的检测.当前,隐藏的手段多种多样,且呈现出底层化趋势.例如:通过直接篡改用户态监控程序(如 ps,netstat 等),可以自动过滤用户枚举进程和网络连接的操作结果;通过重定向应用程序的调用路径(如重定向动态链接库以及系统调用表调用路径等),也可以达到隐藏操作系统对象等目的;特别是以 Rootkit 为代表的内核级恶意程序,它驻留在内核内存,能够直接操纵内核数据结构,具有对操作系统内核的控制权,可以通过篡改内核代码等方式,绕过甚至破坏操作系统的安全检测机制.

针对恶意软件的隐藏行为,学术界和工业界开展了广泛研究,并研发了相应的工具和系统.当前的操作系统隐藏对象检测机制按照实现和部署方式可分为应用层检测、操作系统级检测、硬件辅助检测和虚拟化层检测这 4 种类型.其中,应用层和操作系统级检测[1−5]仍然是使用最广泛的检测方法.然而,首先,由于这两种方法都位于操作系统内部,很容易被 Rootkit 发现;其次,由于 Rootkit(特别是内核级 Rootkit)具有高级别系统权限,因此一旦发现检测工具的存在,就很容易通过篡改、绕过、禁用等手段破坏检测工具的有效性和完整性;最后,出于检测需要,这些工具需要获得操作系统内部对象的状态及信息,因而与操作系统平台紧密绑定(例如,Rootkit Revealer[1]仅适用于 Windows 平台),无法跨平台兼容多种操作系统.为了弥补应用层和操作系统级检测工具存在的安全缺陷,出现了基于辅助硬件的检测方法和工具[6].在此方式下,检测程序固化于硬件,与被检测对象隔离开,具有防篡改性和不可旁路性.但由于需要额外的硬件支持且不可避免地引入性能开销,并未得到广泛应用.

由于虚拟机技术具有隔离性、洞察性和可干涉性等优势,近年来被用于隐藏检测技术中.其基本思路是:将检测程序置于 VMM 层,与被检测软件隔离开;同时,利用 VMM 的底层监控优势来监控上层软件的隐藏行为.其优势是具有良好的透明性和不可旁路性.然而,现有基于 VMM 的隐藏对象检测技术由于缺乏充分的上层语义信息,导致检测结果不精确,同时,现有技术仅关注进程检测,未考虑对操作系统内其他对象(如网络连接、文件等)隐藏行为的检测,特别是缺乏对隐藏对象及行为间的关联分析.例如,进程通过入侵或伪装以普通进程的身份运行,同时隐藏自身的网络连接和文件,仅依赖进程隐藏检测方法无法识别.另一方面,当发现隐藏网络连接和文件时,由于缺乏对恶意行为主体(如进程)的关联,导致难以识别完整攻击路径,无法发现攻击源头.

针对上述问题,本文提出了基于 VMM 的操作系统隐藏对象关联检测方法 vDetector,其主要贡献如下:

(1) 采用多视图对比检测机制,通过比较操作系统用户态视图、内核态视图以及虚拟机监控器视图之间的差异来识别操作系统隐藏对象,支持对进程、网络连接和文件这 3 种操作系统内部对象的隐藏行为检测.

(2) 基于虚拟机自省技术,在 VMM 层建立客户操作系统内进程、网络连接和文件三者之间的关联关系,并识别其交互特征,从而构建完整攻击视图.

(3) 设计并实现了显式和隐式相结合的操作系统对象监控机制,基于 VMM 层获取的隐式信息生成可信视图,并结合操作系统内部语义来获取对象详细信息,通过比较显式和隐式视图之间的差异,识别内核恶意行为.

本文第 1 节介绍背景及相关工作.第 2 节概述 vDetector 系统.第 3 节介绍关键技术.第 4 节给出 vDetector 系统实现.第 5 节对系统进行测试评估.第 6 节总结全文.

1 背景及相关工作

当前,对操作系统内隐藏对象和隐藏行为的检测一直是系统安全的一项重要研究课题.在工业界,Windows Sysinternals 公司推出的 Rootkit 检测工具——Rootkit Revealer[1]通过对比最高级别系统扫描和最低级别系统扫描的结果来发现隐藏恶意程序.其他具有类似功能的检测工具还包括 F-secure 公司的 Blacklight[2],Klister[3]以及微软的恶意软件清除专用工具 WMSR[4]等.在学术界,GhostBuster[5]通过比较 high-level 与 low-level 获得的系统视图识别隐藏的操作系统对象.以上工具和系统均部署于操作系统内部,易被 Rootkits 所发现,存在被绕过和篡改的安全风险.针对上述问题,Copilot[6]采用了硬件方式,基于 PCI Card 来实现,通过在 PCI Card 上运行监控程序,周期性地对内存数据结构进行扫描,从而获取真实的操作系统对象列表.然而,由于采用周期性的检测办法,存在被恶意程序逃避和攻击的“时间窗口”,且 Copilot 仅支持对具有持久状态的内存状态进行快照,导致检测的

对象范围受到限制(检测中存在漏过非持久状态隐藏对象的风险).此外,由于采用硬件实现,还导致了额外的硬件开销.

虚拟化技术的出现给安全技术及系统带来了新的机遇,学术界和工业界针对基于虚拟化技术构建的安全系统及产品开展了广泛研究[7−13].VM Introspection[7]是其中一项具有代表性的技术,主要解决了虚拟机监控器及内部操作系统间的语义鸿沟(semantic gap)问题,其技术手段是利用操作系统内核数据结构特征来为 VMM 层提供语义信息,目前已被众多基于虚拟化技术构建的安全系统及工具所采用.Ether[8]是一种基于 VMM 的通用监控平台,通过设置陷入的方式,对虚拟机的 CPU、内存以及系统调用等进行检测.Secvisor[9]通过内存虚拟化技术保证只有已验证的代码才能在内核态执行,从而能够抵御代码注入对内核完整性的攻击.Xu 等人[10]提出了基于虚拟机监控器来检测并阻止违背内核完整性的行为.Antfarm[11]则通过 CR3 寄存器等硬件数据来区分进程的相关操作(如进程创建、上下文切换和结束等),可用于识别和监控进程行为,缺点是无法获得详细而完整的进程语义信息.Lares[12]基于 Xen 实现,主要面向虚拟机内存和磁盘 I/O 状态的主动监控,其相应功能也可以扩展到网络流量、CPU 等.

针对传统隐藏检测技术中存在的安全问题,一些研究工作[14−17]提出了基于 VMM 的隐藏进程检测技术,通过对比在 VMM 层获得的可信视图与虚拟机内获得的非可信视图间的差异来发现隐藏进程.其中,VMWatcher[14]采用 VM Introspection 技术,通过分析内核数据结构来获得可信视图;Lycosid[15]在 Antfarm 基础上进行改进,从 VMM 层获取 VM 的进程“模糊”信息,并从中区分出隐藏进程.两种方式各具优缺点: VMWatcher 的优点是可以获得操作系统内部语义,有利于获得详细进程信息,缺点是其检测机制存在被内核级恶意程序逃逸的可能;Lycosid 由于使用隐式信息,不依赖于操作系统内部数据结构,因而比 VMWatcher 更安全,然而由于获得的进程信息过于底层,缺乏上层语义,导致检测范围受限,且只能用于检测隐藏进程.SIM[16]则采用混合(hybrid)方法,检测模块部署于 VM 内部,并为其设置单独的地址空间,一旦这些地址空间被访问,就会陷入到 VMM 层对其进行保护;然而,其检测模块代码无法依赖于任何操作系统内核代码,从而导致其语义信息不完整,且它只能检测对象的动态行为,例如进程的创建、系统调用执行等,检测粒度受限.VMware vSphere Hypervisor 产品中的 VMsafe[17]作为虚拟化环境的第三方安全解决方案体系,除了能够获悉虚拟机资源使用情况,监控系统状态之外,还为第三方工具提供了安全接口,可用于检测病毒、Rootkit 和恶意软件.但目前,VMsafe 仅限于 vSphere 环境中使用,且属于非开源的商用系统.本文在文献[18]的工作基础上提出了多视图的隐藏对象检测技术,支持对隐藏进程、文件和网络连接的检测,实现了多类型对象间的关联检测.与文献[18]的工作相比,能够提供更细粒度的检测功能,同时有助于识别完整的攻击路径.

2 vDetector 系统概述

vDetector 在 VMM 层实现,检测对象为操作系统级的恶意程序及其隐藏行为.本节将给出基于多视图对比的隐藏对象检测模型,并基于该模型给出 vDetector 的设计.

2.1 基于多视图对比的隐藏对象检测模型

首先,定义系统中的基本实体:

- P_M={p|p 为具有隐藏功能的恶意程序};
- M={m|m 为操作系统对象的监控程序};
- O={o|o 为操作系统对象};
- O_M={o|o 为对监控程序 M 可见的操作系统对象};
- O_H={o|o 为隐藏的操作系统对象},根据以上定义有:$O_H=O-O_M$.

vDetector 主要针对操作系统中由 Rootkit 等恶意程序发起的隐藏攻击,接下来,通过威胁模型(threat model)对其进行刻画.

(一) 威胁模型

首先需要指出的是,vDetector 所针对的攻击为操作系统内部攻击,本文假设恶意程序仅能威胁虚拟机内部

代码及数据的安全性,而不能跳出虚拟机对 VMM 造成破坏;攻击者的目的是隐藏某些对象使其对监控程序不可见,例如隐藏后门进程和网络连接,从而躲避监控程序的检测;攻击者拥有对虚拟机操作系统的完全控制权,可通过内核方式实现隐藏,同时具有反监控功能,例如,可以绕过甚至破坏位于操作系统内部的监控程序.

定义 1. 对象监控器 m 定义为从 I 到 R 上的函数 $m:I\to R$.其中,I 代表操作系统中用于表示对象的信息和数据,如内核数据结构等;R 表示监控器 m 可见的对象集合.

定义 2(隐藏攻击 *H*). 对于监控器 m,如果$\exists P_M$且满足以下两个条件,则称 P_M 对监控器 m 发起了隐藏攻击:

(1) 当 P_M 未运行时,$\forall o\in O$,有 $o\in m(I)$;

(2) 当 P_M 运行时,$\exists o\in O$,但 $o\notin m(I)$.

隐藏攻击 H 又分为两种:

(1) $H_1:m\to m'$;

(2) $H_2:I\to I'$,

其中,H_1 通过篡改监控器 m 内部逻辑的方式实现对象隐藏,例如,通过篡改 linux 进程查看程序 *ps* 可以过滤进程信息,从而达到隐藏进程的目的;H_2 通过篡改监控器输入的方式实现对象隐藏,例如,对于一个用户态监控器 m,需要通过系统调用获取操作系统对象信息,即将系统调用返回值作为其输入 I,而内核态恶意程序可以篡改内核态系统调用例程及其返回值,从而破坏监控器 m 的输入 I.

(二) 检测模型

针对以上的威胁模型,我们设计了基于视图对比的隐藏对象检测模型,通过对比在不同层次所观测到的对象视图间的差异来发现隐藏攻击行为.首先给出视图(view)的定义.

定义 3. 视图 V 定义为 $V=vb(m,O)$,其中,vb 为视图生成函数,m 表示监控器,O 表示被监控对象集合.视图 V 代表监控器可见的对象集合,同一对象集合在不同的观察者面前产生的视图间可能存在差异.视图分为可信视图 V_t 和非可信视图 V_u.V_t 能如实反映对象的当前状态,而 V_u 则不能.例如,被黑客篡改的进程列表就是 V_u 的一种.

定义 4. 基于多视图对比的隐藏对象检测模型 MVC(multi-view-comparison-based hidden object detection model)定义为六元组 $MVC=\langle O,M,V_t,V_u,VB,CMP\rangle$:

- $O=\{o|o$ 为被检测的操作系统对象$\}$,$O=O_M+O_H$.
- $M=\{\langle L_M,L_O\rangle|\langle L_M,L_O\rangle$表示某一监控器,其中,$L_M$ 表示监控器位于的安全级别,L_O 表示监控对象位于的安全级别$\}$.
- $V_t=\{v|v$ 为可信视图$\}$.
- $V_u=\{v|v$ 为非可信视图$\}$,$V_t\cap V_u=\varnothing$且 $V_t\cup V_u=V$.
- $VB=\{vb|vb$ 为视图生成函数$\}$,$vb:O\to V$.
- $CMP=\{cmp|cmp$ 为视图对比函数$\}$,$cmp:(v,v')\to O_H$.

在本模型中,vb 作为视图生成函数,在监控器 M 中实现,用于生成相应的对象视图 V.其中,可信视图 V_t 具有完整性和真实性.

2.2 vDetector设计

基于以上模型,给出 vDetector 的设计(如图 1 所示):

- $O=O_p\cup O_f\cup O_c$,O_p,O_f,O_c 分别代表 *Process*,*File*,*Connection* 这 3 种对象集合.
- $M=\{\langle user,user\rangle,\langle vmm,system\_call\rangle,\langle vmm,kernel\rangle,\langle vmm,hardware\rangle\}$,$M=M_{int}\cup M_{ext}$,$M_{int}$ 表示位于 Guest OS 内部的监控器,对 P_m 可见;M_{ext} 表示位于 Guest OS 外部的监控器,对 P_m 不可见.
- $V_t=\{v_{hardware}\}$,$v_{hardware}$ 为 M_{ext} 生成的虚拟硬件视图.
- $V_u=\{v_{user},v_{system\_call},v_{kernel}\}$.
- $VB=\{vb_{user},vb_{system\_call},vb_{kernel},vb_{hardware}\}$.其中,$vb_{user}$ 实现于 Guest OS 的用户态,通过调用 API 获得对象信息;Vb_{system_call},vb_{kernel} 和 $vb_{hardware}$ 三者均实现于 VMM 层,其中,vb_{system_call} 通过在 VMM 层截获系统调用来生成对象视图;vb_{kernel} 基于 VM Introspection 机制实现,并依赖于 Guest OS 内部语义;$vb_{hardware}$ 在虚拟

设备层截获 *Process*,*File*,*Connection* 三者的底层活动信息,其实现不依赖于 Guest OS 内部语义.

- 视图的安全等级与视图生成函数(*vb*)相关.V_{user} 从 Guest OS 用户态获得,可能被恶意程序篡改,因此为非可信视图;$v_{hardware}$ 的生成不依赖于 Guest OS 内部语义,且视图生成函数位于 VMM 层,满足完整性和真实性条件,因此为可信视图.v_{system_call} 和 v_{kernel} 也在 VMM 层生成,由于 vDetector 基于硬件虚拟化技术实现,VMM 位于 Ring-1,其安全级别高于用户态(Ring 3),因此,v_{system_call},v_{kernel} 比 v_{user} 更可信.但由于其仍然存在被恶意程序旁路的可能,因此也是非可信视图.对于以上 4 种视图,通过相邻视图间的两两对比(如图 1 所示)共可识别 3 种类型的隐藏攻击:用户态攻击(user-level attack)、系统调用攻击(system-call attack)以及内核态攻击(kernel-level attack).VDetector 的视图对比函数 CMP 采用 diff 方式,首先对比相邻视图中对象列表的长度,如果不相等,则说明存在隐藏对象,接着逐一比较两个对象列表中的元素,从而识别出隐藏对象.

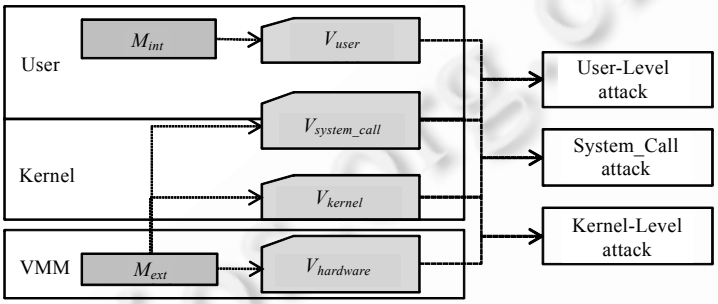


Fig.1 The design of vDetector

图 1 vDetector 设计

vDetector 的系统架构如图 2 所示.首先,位于 VMM 和 Host OS 中的进程监控器(process monitor,简称 PM)、文件监控器(file monitor,简称 FM)和网络连接监控器(connection monitor,简称 CM)三者分别通过 VMM 获得进程、文件和网络连接的显式和隐式信息,缓存于 Host OS 中,并定期更新.其中,进程监控器(PM)采取主动监控方式,实时截获进程切换事件,并根据 CR3 寄存器值判断是否有新进程创建,同时,通过进程结构体(task struct)获取详细进程信息.视图引擎(view engine,简称 VE)定期向 Guest OS 中的代理(agent)发起请求,以获取虚拟机内部不可信视图,并与 PM,FM 和 CM 中维护的可信视图进行比较,所识别出的隐藏对象信息将传递给关联引擎(correlation engine,简称 CE)进行关联分析.CE 在收到 VE 传递的隐藏对象信息后,会遍历 Guest OS 中与进程、文件和网络连接相关的内核数据结构,获取 Guest OS 的内部语义信息,并据此建立三者间的关联关系.

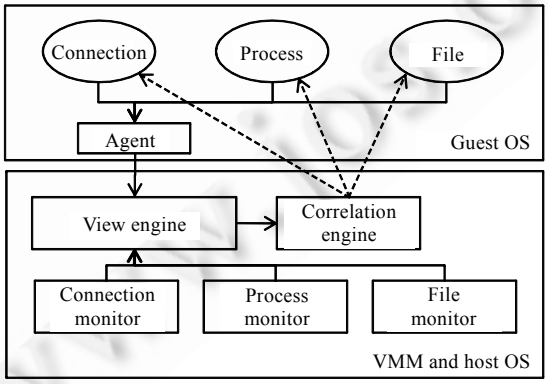


Fig.2 System architecture of vDetector

图 2 vDetector 系统架构

3 vDetector 关键技术

3.1 显式和隐式相结合的VMM层视图建立机制

显式信息(explicit information)是指在 VMM 层基于操作系统内部语义而获取的操作系统对象的状态信息.例如,通过遍历 Linux 内核中的 task list 获得进程列表.隐式信息(implicit information)是指不依赖于操作系统内部语义,而直接从 VMM 中获取的底层监控信息.例如,通过检测到新出现的 CR3 寄存器值可以得知新进程的创建,通过截获并解析虚拟机网卡报文可识别虚拟机内的网络连接等.vDetector 结合了显式和隐式两者的优点:通过隐式信息获取被监控对象的完整集合;同时,显式地获取被监控对象的操作系统内部语义,从而保证了监控信息的真实性和完备性.

进程视图.首先由隐式信息生成进程可信视图.我们利用 Antfarm[11]中的方法,在 VMM 层通过监控 CR3 寄存器值的改变来识别虚拟机中的进程切换事件,并以 CR3 作为进程的标识,如果发现新的 CR3 值,则将其作为进程标识加入进程列表,并同步识别进程退出事件以维护进程列表的最新状态.然而,由于 CR3 值为硬件信息,不包含任何操作系统语义,无法获得进程详细信息.针对这一问题,我们通过虚拟机自省机制,以显式方式获得进程详细信息.进程信息存储于内核数据结构 task_struct 中,而内核在初始化进程时在为每个进程在内核中分配了独立存储空间,其中存放了进程内核态堆栈和进程 thread_info 结构体,分别位于两个不同内存页内,thread_info 结构体中存放了指向当前进程 task_struct 指针.当 VMM 截获进程切换事件时,可以通过 ESP 寄存器值获得进程内核态堆栈的首地址,通过地址计算可以得到 thread_info 首地址,而 thread_info 结构体的前 4 个字节即存放了 task_struct 指针,通过该指针找到 task_struct 结构体,并根据偏移量获得相应进程的具体信息(如进程名、进程号等).

网络连接视图.首先,在 VMM 层通过遍历内核数据结构方式显式地获得网络连接视图.Linux 内核将所有 socket 使用时的端口通过哈希表来管理.该哈希表存放在全局变量 inet_hashinfo 中.vDetector 根据内核符号表导出的内核数据结构偏移地址获得的 inet_hashinfo 的首地址,从中获得 hlist_head,以此为起点遍历所有 hlist_node,获得 socket 列表及相关信息(如端口号等).由于获取网络连接的操作是定期执行的,因此存在被攻击时间窗口,且考虑到内核数据存在被篡改的可能,vDetector 在 VMM 的网络设备模型中插入钩子(hook)函数,截获虚拟机发送和接收的网络报文,解析报文获得相应的端口号.如果发现当前列表中未出现的端口,则说明检测到新的网络活动.这种基于网络设备模型的信息获取方式无须已知操作系统内部语义,因此为隐式信息,且与显式方式相比更安全,具有防篡改、无法旁路等优势.vDetector 主要针对 TCP 和 UDP 两种类型的网络连接.值得注意的是,一般观点认为 TCP 是面向连接的,而 UDP 则是无连接的.本文中的 UDP 连接则是指对 UDP 通信及其状态的保存和跟踪.

文件视图.在虚拟机启动前解析镜像磁盘文件,获得 superblock 起始地址.当虚拟机运行时,从 superblock 出发,首先定位根 inode,并以根 inode 为起点,遍历文件系统目录树获得文件列表.文件系统全局视图的建立由于涉及整个文件系统目录树,因此较为耗时.在具体实现中,每次仅获取局部视图,即每个检测周期内仅遍历指定目录子树,通过多次遍历建立完整文件系统视图.文件视图的建立需要依赖虚拟机内文件系统的相关信息,因此为显式方式.

3.2 基于多视图对比的隐藏对象检测机制

多视图对比检测技术[5]是通过在不同层次对操作系统对象进行观察而得出不同的视图,并通过比较各视图间的差异来发现隐藏行为.在系统软件栈各层次中,底层软件拥有对上层软件的控制权.例如,操作系统内核负责用户态进程调度分配,具有对进程的绝对控制权.因此,软件位于软件栈中的层次越低则越安全,且获得的视图可信度越高.多视图对比检测技术正是利用这一基本前提,当某操作系统对象在高可信度视图中可见而在低可信度视图中不可见时,则认为该对象被隐藏.

vDetector 采用视图对比机制来识别隐藏操作系统对象.首先,vDetector 在 VMM 层分别通过显式和隐式方式获得对象视图,并将两者合成为可信视图;接着,在虚拟机操作系统内,通过代理程序(agent)搜集进程、文件和

网络连接三者的监控信息,并生成非可信视图;最后,通过对比可信与非可信视图间的差异来识别隐藏的操作系统对象.

对于进程,存在以下几种视图:

(1) 虚拟机内部用户视图(VM user view):在虚拟机内部通过用户态 API 获取.

(2) 系统调用视图(system call view):在内核层和 VMM 层都可以截获.

(3) 内核数据视图(kernel data view):在内核和 VMM 层都可以获取,但在 VMM 层获取更安全,方法是遍历 task_list 数据结构.

(4) 虚拟硬件视图(virtual hardware view):在 VMM 层获得,通过监控 CR3 寄存器值的变化以识别进程.

vDetector 在 VMM 层获得视图(3)和视图(4),并合成两者以构建可信视图;在虚拟机用户态获取非可信视图,并对比可信和非可信视图以发现隐藏进程.值得注意的是,由于存在针对 task_list 的内核级攻击的可能(例如,通过移除 task_list 中节点的方式来隐藏进程),视图(3)和视图(4)可能存在不一致的情况.为此,vDetector 在将视图(3)和视图(4)合成可信视图前,首先检测两者的差异,如果视图(3)中的进程数少于视图(4),则说明存在没有被 task_list 索引到的进程,进而表明 task_list 的数据结构被篡改.

对于网络连接,存在以下 3 种视图:

(1) 虚拟机内部用户视图(VM user view):通过用户态 API 获取.

(2) 内核数据视图:在内核和 VMM 层都可以获取,但在 VMM 层获取更安全,方法是通过 inet_hashinfo 遍历 socket 链表.

(3) 虚拟硬件视图:在 VMM 层虚拟网卡设备模型驱动中截获网络报文.这种方式获得的信息最可靠,且无法被旁路.

vDetector 在构造可信视图时以视图(2)为主,同时辅以视图(3),一方面利用视图(2)中包含的丰富语义信息,另一方面基于视图(3)以保证可信视图的完整性.vDetector 通过用户态视图构造非可信视图,同时与可信视图对比,以发现隐藏网络连接.

文件不同于进程和网络连接.首先,操作系统中的文件数量远大于进程和网络连接;其次,文件是静态存在的,而进程和网络连接是运行时概念,这也意味着文件信息的获取不能仅依靠内核运行时数据结构,还需要依赖位于磁盘的静态信息(如 inode 等).对于文件而言,在单个检测周期内获取整个文件系统视图过于耗时.vDetector 将文件系统目录树划分成多个子树,每次仅获取局部视图.同时,每个检测周期内,同步获取 VMM 可信视图和 VM 内部非可信视图,并通过对比来发现隐藏文件.vDetector 无法保证在任一时刻能够识别所有隐藏文件,但通过同步获取可信视图和非可信视图,保证了检测结果的局部准确性.

3.3 基于操作系统语义的隐藏对象关联技术

在识别出进程、文件和网络连接这 3 种隐藏对象后,vDetector 通过建立三者间的关联关系来构建完整的隐藏视图.关联关系的建立依赖于操作系统内部语义.以 Linux 为例,进程访问文件并使用网络连接;而进程是可执行文件的运行态实体,即可执行文件以进程方式在操作系统中运行;网络连接通过 socket 与文件发生联系,因为在 Linux 中,socket 可以看作是一种特殊类型的文件,其结构体中包含了指向文件对象的指针.

由文件定位进程,如图 3 所示,当通过视图对比发现隐藏文件并获得文件路径后,通过虚拟机内核映射表 System.map 中的 init_task 定位 tast_list 首地址,遍历进程列表.对于每个进程 entry,遍历其打开文件数组 Fd_array,读取数组中每个文件对象对应的 dentry 对象,并根据 dentry 中当前目录名称和父目录指针递归向上查找,直到根目录,从而获得该文件的完整路径.如果该文件路径与隐藏文件路径相匹配,则说明隐藏文件被当前进程所使用.需要注意的是,不是每个隐藏文件都一定能够关联到某个进程,因为此时文件可能未被进程使用.对于未被进程使用的隐藏文件,将采用基于 VMM 的文件监控方法来进行监控:一旦发现被访问,则识别出可疑行为,立即重启关联过程以发现相关进程;如果仍未发现使用该隐藏文件的进程,则表明该文件正在被内核所访问,而通常内核并不直接访问文件,这也间接说明了内核攻击行为的发生.

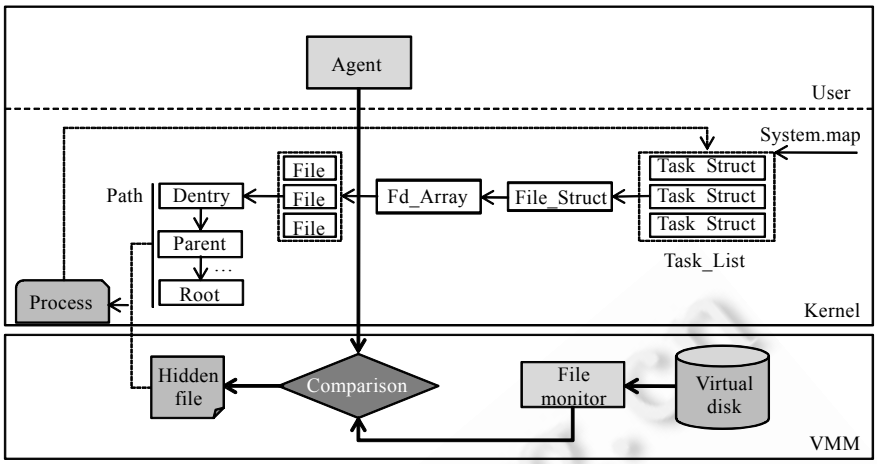


Fig.3 Correlation of file and process

图 3 文件与进程的关联

由网络连接定位进程.网络连接通常表示成五元组:*Connection*=⟨*Protocol*,*Src_IP*,*Src_port*,*Dst_IP*,*Dst_Port*⟩.其中,*Src_IP* 表示源 IP,*Dst_IP* 表示目的 IP,*Src_Port* 和 *Dst_Port* 分别表示源端口和目的端口.网络连接的两端为 Socket,用三元组表示:*Socket*=⟨*Protocol*,*IP*,*Port*⟩.需要注意的是,我们将正处于监听阶段尚未与远端建立通信的 Socket 也看作网络连接.网络连接与进程间的关联主要通过 Socket,而 Socket 即是一种特殊类型的文件,进程访问 Socket 的方式与文件访问相同.具体关联过程如下:

- 位于 VMM 的 CM 模块获取可信视图,并与 Agent 建立的非可信视图进行比较,识别出隐藏网络连接,并提取端口号;
- 以 Inet_hashinfo 为起始索引,遍历 hash_bucket 列表,定位到绑定隐藏端口号的 socket;
- 对于进程列表中每一个进程 task_struct,遍历该进程打开的所有文件对象,将文件对象指针与 socket 的 file 指针进行比较,如果相等,则说明该 socket 被该进程所使用.

由进程定位文件和网络连接.当发现隐藏进程时,根据 task_struct 的 files_struct 指针获取其使用的文件列表,遍历文件列表,并根据文件类型来识别普通文件和网络连接.需要注意的是,task_list 数据结构可能被攻击,vDetector 通过 CR3 识别进程,并通过内核栈指针定位 task_struct 数据结构.

4 vDetector 系统的研制

我们在 KVM[19]虚拟机监控器上实现了 vDetector 的系统原型,并将 vDetector 集成到了 iVIC 虚拟计算平台[20]中.KVM 采用了 Intel VT 硬件辅助虚拟化技术[21],虚拟机在正常情况下以 Linux 进程形式运行于客户模式(guest mode).在该模式下,客户操作系统代码将直接运行在物理 CPU 上;当执行特权指令时则陷入 VMM,此时,CPU 切换为监管模式(VMM mode).KVM 虚拟机监控器由 KVM 驱动(KVM driver,以内核模块形式加载到 Linux 内核)和 Qemu 进程两部分组成,KVM 驱动负责 CPU 和内存虚拟化以及中断异常的捕获预处理,Qemu 进程则主要负责虚拟机 I/O 相关设备的模拟,两者相互配合,为虚拟机提供完整的运行环境.vDetector 基于 KVM 实现(如图 4 所示).其中,进程监控器(process monitor)实现于 KVM 内核驱动中,并利用 KVM 驱动中内置的 CR3 寄存器截获功能来捕获客户操作系统中的进程切换事件,获得进程列表,并利用影子页表提供的虚拟地址到物理地址的转换功能,在 KVM 驱动中直接读取虚拟机内核数据结构获得进程详细信息,并将以上信息合成为进程可信视图,通过 Linux ioctl 通道传递给位于 Qemu 的视图引擎(view engine).连接监控器(connection monitor)实现于用户态 Qemu 进程中,借助 Qemu 中的 KVM_read_guest 方法可以直接读取客户操作系统内存,从而获得位于虚拟机内核的网络连接信息;同时,Qemu 模拟了虚拟机网卡 I/O,因此可以截获并解析网络报文.两种信息

也构成了网络连接的可信视图.文件监控器(file monitor)也位于 Qemu 进程,通过直接读取并遍历位于磁盘镜像的文件系统目录树,获得文件可信视图.视图引擎(view engine)在获得以上 3 个可信视图后,从位于虚拟机用户态的代理(agent)获得非可信视图,通过视图对比发现隐藏对象,并交由关联引擎(correlation engine)进行关联分析.关联引擎通过分析客户操作系统内核数据结构识别出进程、文件和网络连接三者的关联关系,并把隐藏对象及其关联关系存储于数据库 vDetector DB 中.最后,用户通过 iVIC Portal 远程查询宿主机的 vDetector 数据库来查看和监控相应虚拟机的隐藏对象及恶意行为.

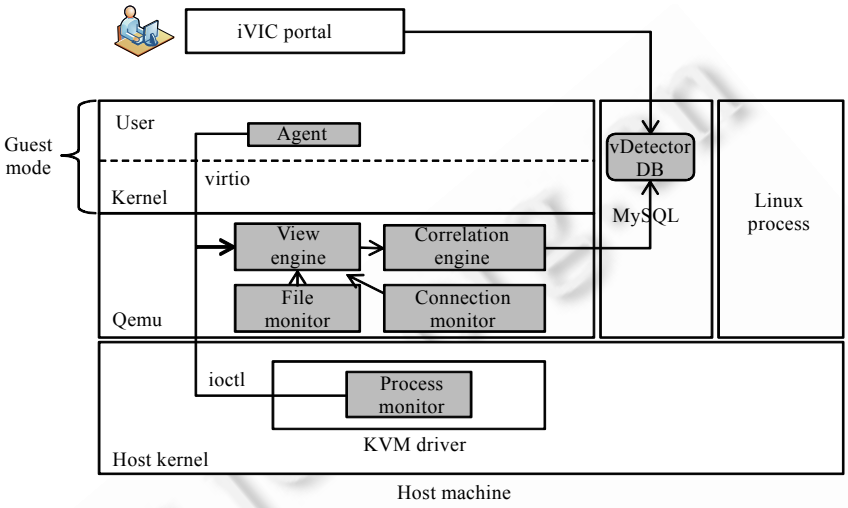


Fig.4 Implementation of vDetector

图 4 vDetector 实现

5 实验及结果分析

我们从功能性和性能两个方面来对 vDetector 进行评测.功能性测试用于验证 vDetector 有效性,即能否有效识别出 Guest OS 中存在的隐藏对象,并分析隐藏对象间的关联关系.性能测试用于验证 vDetector 给原有系统所带来的性能开销.实验环境如下:

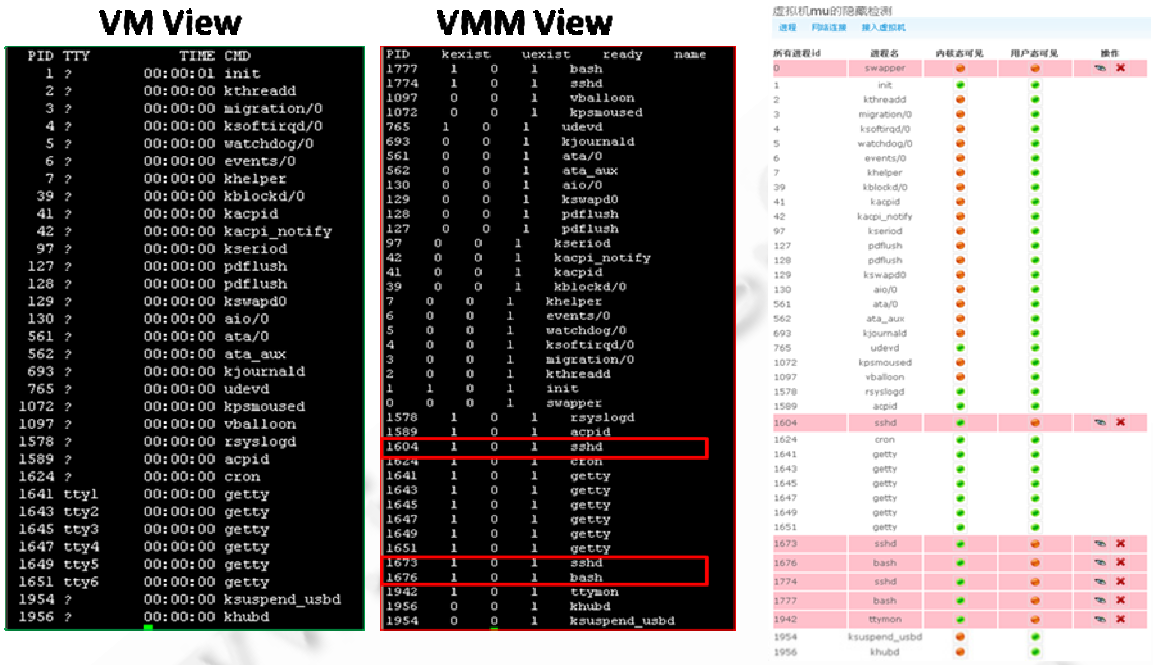
宿主机为 Intel(R) Core(TM)2 @ 2.26GHz 的双核 CPU,3072KB Cache,内存大小 2G,硬盘容量 320G,操作系统为 Linux Debian Lenny,内核版本 2.6.26-2-686.虚拟机为单核 CPU,与宿主机主频相同,内存大小 256M,硬盘容量 10G.在功能测试中,我们采用包括 Linux 2.6 和 Linux 2.4 的多个操作系统版本来验证 vDetector 有效性.在性能测试中,我们选用与宿主机相同的操作系统来测试 vDetector 的性能开销.虚拟机监控器采用 Linux 下的主流虚拟化软件 KVM,其中,内核驱动版本为 kmod-88,Qemu 版本为 0.12.5.

5.1 功能测试

本节首先通过 Adore-ng 和 ddrk 两款 Linux Rookit 来验证 vDetector 的检测能力.首先,在虚拟机内启动后台服务进程 sshd,该进程被作为后门程序以接受攻击者的网络连接和命令,因此需要被隐藏;其次,加载 Adore-ng,并使用其内置的客户端软件 ava 来隐藏 sshd 进程及相应文件;接下来,通过 ddrk 来隐藏 sshd 的网络连接;最后,通过 ps,ls,netstat 等 Linux 系统工具确认相应的进程、文件和网络连接已不可见.

图 5 展示了 vDetector 获得的多个系统视图(view)及对比检测结果.其中,VM View 显示了在虚拟机内部通过执行 ps-A 命令所获得的进程列表,而 VMM View 则给出了从 VMM 层获得虚拟机 OS 内部的进程列表.由图 5(a)和图 5(b)可知,1 604,1 673 以及 1 676 号等进程在 VMM View 中可见,在 VM View 中则不可见.这也说明这些进程在 VM 中被隐藏.我们通过对比两个视图间的差异即可发现隐藏进程的存在.图 5(c)进一步给出了 VMM

合成视图、VMM 系统调用视图和 VM 用户视图之间的比较结果.图 5(c)中的前两列“进程 id”和“进程名”给出了 VMM 合成视图中的进程信息,该视图是可信的;第 3 列给出了 VMM 系统调用视图,即在 VMM 层通过截获系统调用的方式所识别的进程信息;第 4 列给出了 VM 内部所观测到的进程信息,其中,小圆点代表可见,大圆点代表不可见.由图 5(c)可知,通过多视图之间的对比,进一步区分了隐藏攻击级别,识别出了用户态和内核态的进程隐藏行为.



(a) VM 内部视图 (b) VMM 视图 (c) 多视图比较结果

Fig.5 Hidden process detection based on multi-view

图 5 基于多视图的隐藏进程识别

图 6 给出了 vDetector 对隐藏网络连接的检测结果.通过在虚拟机内部执行 netstat 命令,所获得的虚拟机内部用户视图(VM user view)中未发现任何网络连接,而虚拟机监控器视图(VMM view)则显示了两个网络连接,这也说明这两个网络连接已被隐藏.图 6 中的对比结果也表明,vDetector 成功发现了隐藏的网络连接(用大圆点标记).同时,vDetector 还成功地检测到目的端口为 41 662 和 41 794 的两个网络连接分别所属的两个进程(1 676 号和 1 774 号),从而有效识别出隐藏连接的攻击来源(进程),同时验证了 vDetector 的隐藏对象关联功能的有效性.

接下来,我们通过多个典型 Rookit 实例来验证 vDetector 的有效性及适用范围.其中涉及到多种采用不同隐藏技术的 rootkit,例如,lrk5 是一种经典的用户态 rootkit,而 Adore-ng 和 Mood-nt 同属内核态隐藏类别,但采用的隐藏技术却不尽相同(Adore-ng 通过 LKM 方式实现,Mood-nt 则以通过/dev/kmem 篡改内核数据的方式实现).表 1 给出了 vDetector 对多款 rookkit 的检测结果,其中,“√”代表成功检测到隐藏对象,“×”代表未成功检测到,“—”代表当前 Rookit 不支持对该类型对象的隐藏.由表 1 可知,vDetector 能够成功地检测不同类型且采用不同技术的 rootkit 的隐藏行为,且支持 Linux 操作系统中的进程、网络连接和文件这 3 种对象.

为了进一步验证 vDetector 的有效性和先进性,我们还将 vDetector 与当前最新的几款 rootkit 检测工具进行了对比,结果见表 2.其中,Rootkit Revealer 是操作系统内部的一款检测工具,其实现与 OS 相关,无法跨平台;vDetector 与 Rootkit Revealer 相比,在 VMM 层实现,对上层 OS 透明,因此能够支持多种类型的 Guest OS;从

支持的隐藏对象检测类别上看,vDetector 优于 Antfarm 和 Lycosid,不仅支持进程,而且支持对隐藏连接和文件的检测;同时,与其他工具相比,vDetector 支持多级别检测,能够进一步区分内核态和用户态的隐藏行为,为用户提供更细粒度的检测结果;最后,vDetector 支持建立多种隐藏对象间的关联关系,有助于发现完整攻击路径、识别攻击源头.

Fig.6 Correlated detection of hidden connection and process

图 6 隐藏连接及进程关联检测

Table 1 Detection results of vDetector on various rootkits

表 1 vDetector 对多种 rootkit 的检测结果

	内核版本	隐藏级别	隐藏对象类别		
			进程	连接	文件
Lrk5	2.4, 2.6	用户态	√	√	√
Adore-ng	2.4, 2.6	内核态	√	√	√
Ddrk	2.6	用户态,内核态	√	√	√
enyelkm-1.2	2.6	内核态	√	√	—
suckit 2.0	2.6	内核态	√	—	—
Mood-nt	2.4, 2.6	内核态	√	√	—

Table 2 Comparison of vDetector and other detection tools

表 2 vDetector 与其他检测工具的对比

	技术类型	关联支持	多级别检测	隐藏对象检测类别		
				进程	连接	文件
Revealer	OS	√	—	√	√	√
Copilot	硬件,显式	—	—	√	—	—
Antfarm	VMM,隐式	—	—	√	—	—
VMwatcher	VMM,显式	—	—	√	—	√
Lycosid	VMM,隐式	—	—	√	—	—
vDetector	VMM,显式、隐式	√	√	√	√	√

5.2 性能测试

我们从检测效率和运行时开销两方面来测试 vDetector 的性能.检测效率是指执行检测任务所花费的时间;

运行时开销是指在虚拟机运行过程中,由于 vDetector 的引入而造成的性能损失,例如截获进程切换、监控网络流量等操作会对虚拟机性能造成影响.测试中,所有实验都被重复执行 10 次,实验结果是 10 次实验的平均值.

(一) 检测效率

本文采用检测时间 T_D 作为性能指标来评价 vDetector 的检测效率.检测时间是指从检测开始到获得结果所经历的时间.一般而言,检测时间与检测周期成正比,检测周期越短则准确率越高,而过长的检测周期将给攻击者留下足够的时间窗口以逃逸检测.

对于 vDetector 而言,检测时间 T_D 主要由以下 3 个部分组成:

(1) 视图生成(view building)时间 T_{VB}:获取隐藏对象信息并建立视图所需的时间.vDetector 支持多种视图的生成,如虚拟机内部视图和虚拟监控器视图等.

(2) 视图获取(view acquisition)时间 T_{VA}:视图引擎(view engine)从各监控器(monitor)获取和汇总视图信息所需的时间,具体体现在监控器组件与视图引擎间的通信时间上.

(3) 视图比较(view comparison)时间 T_{VC}:比较视图间的差异以发现隐藏操作系统对象所需的时间,同时还包括建立进程、文件和连接视图间关联关系的耗时.

综上,vDetector 的检测时间 T_D 可以表示为

$$T_D=T_{VB}+T_{VA}+T_{VC} \tag{1}$$

其中,由于 vDetector 支持生成多种层次的对象视图,因此 T_{VB} 可以进一步分为虚拟机内部视图(VM view)生成时间 T_{VB_vm} 和虚拟机监控器视图(VMM view)生成时间 T_{VB_vmm}.由于 VM View 和 VMM View 的生成操作可以并发执行,因此有

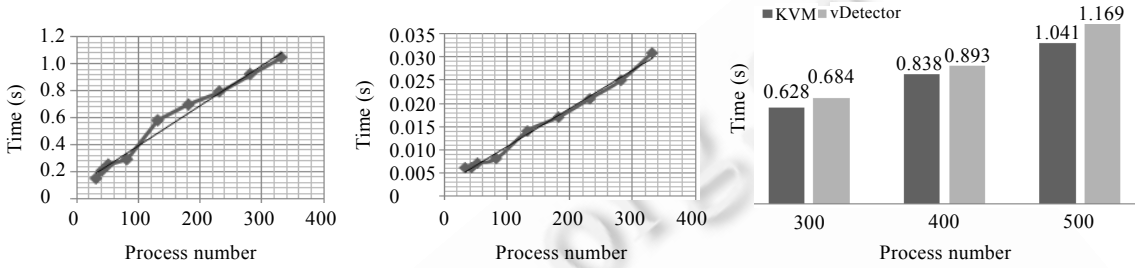
$$T_{VB}\leqslant T_{VB\_vm}+T_{VB\_vmm} \tag{2}$$

值得注意的是,VMM View 的生成方式又分为主动截获和被动检测两种.对于主动截获方式,虚拟机一经运行,视图即被创建,且在虚拟机运行过程中被持续更新.例如,在虚拟机运行过程中,将动态地截获进程发起的系统调用,并实时更新系统调用视图.对于这种方式,在单个检测周期内仅访问视图信息,而并不会执行建立视图操作,因此其视图生成时间并不计算在 T_{VB_vmm} 内.对于被动检测方式,在检测周期内由外部程序触发视图的生成操作,因此生成时间将被计算在 T_{VB_vmm} 内.

视图生成是 vDetector 检测过程中的第 1 步,图 7 给出了生成 3 种不同进程视图的时间开销.图 7(a)给出了 VMM 视图生成时间随进程数变化的曲线.当进程数为 300 时,VMM 视图生成时间约为 1ms,且 T_{VB_vmm} 与进程数成线性正比关系.这一结果符合预期,因为 VMM 视图生成是通过在 VMM 层线性遍历位于 VM 内核中的进程循环链表所得到.由图 7(b)可知,VM 内部视图生成时间 T_{VB_vm} 与进程数也成线性正比,但当进程数为 300 时,VM 内部视图生成时间约为 27ms,远高于 VMM 视图生成时间.这是由于 VMM 视图生成操作仅涉及通过 VMM 对 VM 内核空间的直接内存访问,而生成 VM 内部视图为用户态操作,需要在用户态向内核态发起系统调用请求,特别是 vDetector 还截获了系统调用,从而导致了额外的时间开销.接下来,我们通过对比标准 KVM 和 vDetector 下的进程创建时间来测试基于主动截获方式的 VMM 视图生成过程中所引发的性能开销.图 7(c)给出了在虚拟机中分别创建 300,400,500 个进程的耗时情况.通过计算可知,vDetector 创建 300,400,500 个进程所花费的时间与标准 KVM 相比分别增加了 8.9%,6.5%和 12.2%,这些额外开销主要是由生成 VMM 视图时主动截获系统调用和进程切换所导致.

检测总时间对于确定 vDetector 的检测周期具有重要的参考意义,而分析 vDetector 各个环节的时间消耗则有助于定位性能瓶颈,指导性能优化.为此,我们分别针对进程、连接和文件,测试了 vDetector 的平均单次检测时间,并给出了各分解环节占总时间的百分比,如图 8 所示.其中,图 8(a)给出了进程检测的时间结果,检测规模为 100 个进程;图 8(b)给出了连接检测的时间结果,检测规模为 100 个活动连接;图 8(c)给出了针对 Debian Lenny 初始安装文件系统的文件检测时间结果,检测规模为 26 832 个目录和常规文件.本次实验中,各环节均顺序执行,不存在并发.需要注意的是,图 8(a)和图 8(b)中未给出视图比较时间 T_{VC},这是由于进程和连接的视图比较时间非常小,与其他部分的时间花费相比可以忽略不计.如图 8(a)所示,进程平均单次检测时间为 8.37ms,其中,虚

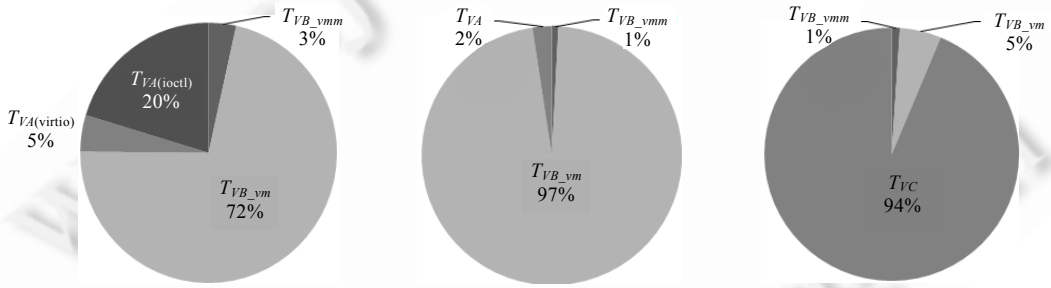
拟机内部视图的生成时间 T_{VB_vm} 占单次检测总时间的 72%,显著高于其他环节.而对于网络连接而言,虚拟机内部视图的生成时间 T_{VB_vm} 与进程相比,在检测总时间中占比例更高,达到了 97%.这是由于需要在虚拟机操作系统用户态通过读取/proc 文件系统获得连接信息,涉及系统调用、文件系统访问等多步操作,因此开销较大.对于文件检测(如图 8(c)所示),首先,其与进程和连接检测相比更为费时,这是由于生成文件视图的过程需要遍历文件目录树;其次,视图比较更为耗时,本例中,视图比较时间 T_{VC} 达到了检测总时间的 94%.这是由于文件视图比较操作需要对比 VMM 视图和 VM 视图中的文件目录树间的差异,其核心机制是字符串比较操作,而字符串比较操作本身较为耗时.此外,由于采用递归算法来比较不同视图中文件目录树的差异,时间复杂度为 $O(n^2)$,n 代表文件数目.当文件数目升高时,性能下降明显.



(a) T_{VB_vmm} (b) T_{VB_vm} (c) 主动截获开销

Fig.7 Process view creation time

图 7 进程的视图生成时间



(a) 进程(总耗时 8.37ms) (b) 连接(总耗时 101.2ms) (c) 文件(总耗时 3.281s)

Fig.8 Breakdown of vDetector detection time for three objects

图 8 vDetector 针对 3 种对象的检测时间分解

(二) 运行时开销

vDetector 在虚拟机运行中将执行跟踪进程切换、截获系统调用以及分析网络 I/O 等操作,因此会为上层应用程序带来运行时开销.为了全面地测试 vDetector 的性能表现,我们使用多组应用基准测试程序来测试 vDetector 的性能,并通过与标准 KVM 进行比较来评测 vDetector 的运行时开销.

首先,采用 dbench 磁盘基准测试程序来测试 vDetector 对磁盘 I/O 吞吐率的影响.Dbench 可以指定同时访问磁盘的进程数来模拟应用程序对磁盘的并发访问.如图 9 所示,当进程并发数为 2,4,6 时,dbench 的性能损失较小,这是由于此时进程数目较少,相应的进程切换和系统调用次数也较少,因此 vDetector 的处理开销也较小;而随着并发进程数的增大,性能损失百分比呈上升趋势,当进程并发数为 12 时,性能损失率接近 9%,其原因是 vDetector 需要频繁截获进程切换和系统调用,从而引发较大的性能开销.

值得注意的是,无论是 KVM 还是 vDetector,随着并发进程数的增大,dbench 磁盘 I/O 吞吐率均呈不断下降

趋势.这也说明了在虚拟机场景下,多进程并发磁盘访问将降低虚拟机的磁盘 I/O 吞吐率.当并发度增大到 14,16 时,磁盘 I/O 吞吐率进一步下降,并发所导致的性能损失逐渐增大,而 vDetector 的性能损失与并发度为 12 时相比则有所降低.

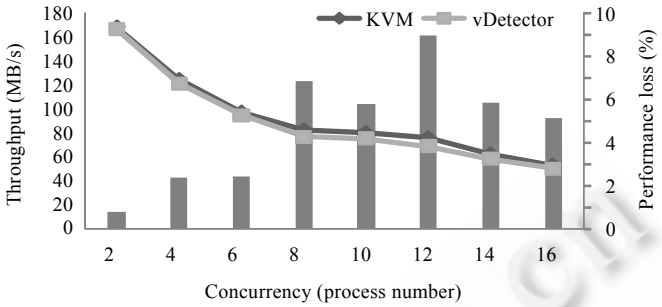


Fig.9 Disk I/O performance evaluation (dbench)

图 9 磁盘 I/O 测试(dbench)

Tbench 是一种网络应用基准测试程序,用来测试多并发场景下的网络 I/O 效率.如图 10 所示,从吞吐率上看,无论是 KVM 还是 vDetector,随着并发度从 2 增加到 6,网络 I/O 吞吐率都有所增长,此后基本保持平稳,最后阶段(当并发度为 16 时)则略微下降.vDetector 所导致的性能损失率也是先升后降,但变化幅度较小,介于 2%~6%之间,在平均性能损失率 4.8%之间上下波动.

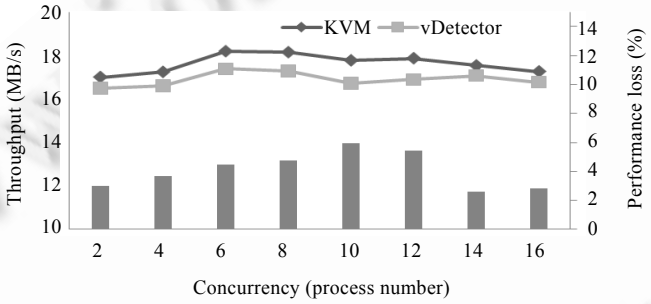


Fig.10 Network I/O performance evaluation (tbench)

图 10 网络 I/O 测试(tbench)

最后,通过文件压缩等常用应用程序来测试 vDetector 对应用执行时间带来的影响.总体来看(如图 11 所示),vDetector 在运行时间上的的平均开销为 3%左右,最坏情况下为 5%(内核编译 Kernel Build).

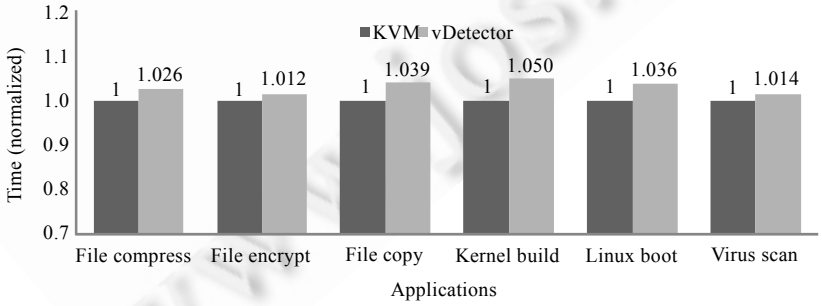


Fig.11 Runtime overhead of vDetector

图 11 vDetector 运行时间开销

VDetector 对文件加密(file encrypt)和病毒扫描(virus scan)两种应用造成的性能影响低于 1.5%,这是因为两种应用均为计算密集型应用,由于 vDetector 基于硬件虚拟化技术实现,计算指令可直接运行于物理 CPU,因此性能开销较低.文件拷贝(file copy)是 I/O 密集型作业,其中包含大量的读写操作,由于 vDetector 截获了系统调用,因此导致了一定的性能开销(3.9%).Linux 启动(Linux boot)过程中将运行多个应用程序并执行系统调用,由于 vDetector 截获了进程切换和系统调用操作,每次截获各包含一次 vm_exit 和 vm_enter 操作,而 vm_exit 和 vm_enter 操作涉及到 CPU 状态的切换,CPU 切换对应用程序的性能影响明显,切换频率越高,则性能开销越大.内核编译(kernel build)是一种综合型应用,不仅需要 CPU 时间,而且还涉及大量的磁盘访问及内存修改,与本实验中的其他应用相比,系统调用和进程切换操作更为频繁,因此,vDetector 引入的性能开销较大,达到了 5%.

6 结 论

本文针对网络安全领域中的恶意程序及其隐藏行为,提出了一种基于 VMM 的多视图关联检测方法 vDetector.vDetector 基于硬件虚拟化技术实现,在 VMM 层对虚拟机内部的进程、文件以及网络活动进行截获并分析,并在不同层次上建立操作系统对象的多维视图,通过多视图间对比来识别隐藏对象,并判断恶意程序的隐藏级别;利用操作系统语义实现了基于 VMM 的隐藏对象关联机制,能够识别隐藏攻击的路径和源头.我们在 KVM 虚拟化平台上实现了 vDetector 原型系统.该系统无须修改客户操作系统内核,且为获得操作系统对象信息提供了良好的接口,具有良好的兼容性和安全性.同时,我们对 vDetector 的有效性和性能进行了评测,结果表明, vDetector 在有效识别操作系统隐藏对象的同时,引入的性能开销不超过 10%.

References:

[1] RootkitRevealer. 2013. http://www.sysinternals.com

[2] F-Secure blacklight. 2013. http://www.f-secure.com/blacklight

[3] Klister—Windows kernel level rootkit detector. 2011. http://www.securiteam.com/tools

[4] Microsoft Windows malicious software removal tool. 2013. http://www.microsoft.com/zh-cn/security/pc-security/malware-removal.aspx

[5] Wang YM, Beck D, Vo B, Roussev R, Verbowski C. Detecting stealth software with strider ghostbuster. In: Proc. of the Int'l Conf. on Dependable Systems and Networks. Washington: IEEE CS Press, 2005. 368−377. [doi: 10.1109/DSN.2005.39]

[6] Petroni NL, Fraser T, Molina J, Arbaugh WA. Copilot—A coprocessor-based kernel runtime integrity monitor. In: Proc. of the 13th USENIX Security Symp. Berkeley: Usenix, 2004. 179−194.

[7] Garnkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. In: Proc. of the 2003 Network and Distributed System Security Symp. Washington: ISOC, 2003. 191−200.

[8] Dinaburg A, Royal P, Sharif M, Lee W. Ether: Malware analysis via hardware virtualization extensions. In: Proc. of the 15th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2008. 51−62. [doi: 10.1145/1455770.1455779]

[9] Arvind S, Mark L, Ning Q, Adrian P. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proc. of the 21th ACM SIGOPS Symp. on Operating Systems Principles. Washington: ACM Press, 2007. 335−350. [doi: 10.1145/1294261.1294294]

[10] Xu M, Jiang XX, Sandhu R, Zhang XW. Towards a VMM-based usage control framework for OS kernel integrity protection. In: Proc. of the 2007 ACM Symp. on Access Control Models and Technology. New York: ACM Press, 2007. 71−80. [doi: 10.1145/1266840.1266852]

[11] Jones ST, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Antfarm: Tracking processes in a virtual machine environment. In: Proc. of the USENIX Annual Technical Conf. Berkeley: Usenix, 2006. 1−14.

[12] Payne B, Carbone M, Sharif M, Lee W. Lares: An architecture for secure active monitoring using virtualization. In: Proc. of the IEEE Symp. on Security and Privacy. Washington: IEEE CS Press, 2008. 233−247. [doi: 10.1109/SP.2008.24]

[13] Li JX, Li B, Wo TY, Hu CM, Huai JP, Liu L, Lam KP. CyberGuarder: A virtualization security assurance architecture for green cloud computing. Future Generation Computer Systems, 2011,28(2):379−390. [doi: 10.1016/j.future.2011.04.012]

[14] Jiang X, Wang X, Xu D. Stealthy malware detection through VMM-based “out-of-the-box” semantic view reconstruction. In: Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2007. 128−138. [doi: 10.1145/1315245.1315262]

[15] Jones ST, Arpaci-Dusseau AC, Arpaci-Dusseau RH. VMM-Based hidden process detection and identification using Lycosid. In: Proc. of the 4th ACM SIGPLAN/SIGOPS Int’l Conf. on Virtual Execution Environments. New York: ACM Press, 2008. 91−100. [doi: 10.1145/1346256.1346269]

[16] Sharif MI, Lee W, Cui W, Lanzi A. Secure In-VM monitoring using hardware virtualization. In: Proc. of the 16th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2009. 477−487. [doi: 10.1145/1653662.1653720]

[17] VMware. VMsafe. 2010. http://www.vmware.com/technology/security/vmsafe.html

[18] Wang Y, Hu CM, Li B. A VMM-based platform to detect hidden process by multi-view comparison. In: Proc. of the 14th IEEE High Assurance Systems Engineering Symp. Washington: IEEE CS Press, 2011. 307−312. [doi: 10.1109/HASE.2011.41]

[19] Kernel based virtual machine. 2009. http://www.linux-kvm.org/page/Main_Page

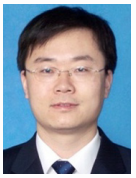
[20] Huai JP, Li Q, Hu CM. Research and design on hypervisor based virtual computing environment. Ruanjian Xuebao/Journal of Software, 2007,18(8):2016−2026 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/2016.htm [doi: 10.1360/jos182016]

[21] Intel virtualization technology. 2008. http://www.intel.com/technology/virtualization

附中文参考文献:

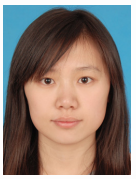
[20] 怀进鹏,李沁,胡春明.基于虚拟机的虚拟计算环境研究与设计.软件学报,2007,18(8):2016−2026. http://www.jos.org.cn/1000-9825/18/2016.htm [doi: 10.1360/jos182016]

李博(1980－),男,辽宁锦州人,博士,讲师,主要研究领域为虚拟化,网络安全,云计算.
E-mail: libo@act.buaa.edu.cn

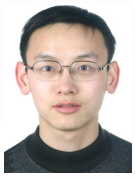


李建欣(1979－),男,博士,副教授,CCF 会员,主要研究领域为信息安全,云计算.
E-mail: lijx@act.buaa.edu.cn

沃天宇(1978－),男,博士,讲师,CCF 会员,主要研究领域为分布式系统,云计算.
E-mail: woty@act.buaa.edu.cn



王颖(1988－),女,硕士,主要研究领域为信息安全.
E-mail: wangying09@act.buaa.edu.cn



胡春明(1977－),男,博士,副教授,CCF 会员,主要研究领域为分布式系统,云计算.
E-mail: hucm@act.buaa.edu.cn

怀进鹏(1962－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为计算机软件与理论,网络计算,信息安全.
E-mail: huaijp@buaa.edu.cn