计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2019.20180577
56(7): 1470-1487, 2019

基于动态策略学习的关键内存数据访问监控

冯馨玥[1,2] 杨秋松[1] 石 琳[1] 王 青[1,2,3] 李明树[1]
1(中国科学院软件研究所基础软件国家工程研究中心 北京 100190)
2(中国科学院大学 北京 100049)
3(计算机科学国家重点实验室(中国科学院软件研究所) 北京 100190)
(xinyue@nfs.iscas.ac.cn)

Critical Memory Data Access Monitor Based on Dynamic Strategy Learning

Feng Xinyue[1,2], Yang Qiusong[1], Shi Lin[1], Wang Qing[1,2,3], and Li Mingshu[1]
[1](*National Engineering Research Center for Fundamental Software*, *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)
[3](*State Key Laboratory of Computer Science*(*Institute of Software*, *Chinese Academy of Sciences*), *Beijing* 100190)

Abstract VMM-based approaches have been widely adopted to monitor fine-grained memory accessing behavior through intercepting safety-critical memory accessing and critical instructions executing. However, intercepting memory accessing operations lead to significant performance overhead as CPU control travels to VMM frequently. Some existing approaches have been proposed to resolve the performance problem by centralizing safety critical data to given memory regions. However, these approaches need to modify the source code or binary file of the monitored system, and cannot change monitoring strategies during runtime. As a result, the application scenarios are limited. To reduce the performance overhead of monitoring memory access in this paper, we propose an approach, named DynMon, which controls safety-critical data access monitoring dynamically according to system runtime states. It does not dependent on source code and need not to modify binary file of the monitored systems. DynMon obtains dynamic monitor strategies by learning from historical data automatically. With system runtime status and monitor strategies, DynMon decides memory access monitoring region dynamically at runtime. As a result, DynMon can alleviate system performance burden by reducing safety irrelevant region monitoring. The evaluations prove that it can alleviate 22.23% performance cost compared with no dynamic monitor strategy. Besides, the performance overhead will not increase significantly with large numbers of monitored data.

Key words safety critical data; memory access monitor; monitor strategy; sequence pattern mining; event intercept

摘 要 在基于虚拟机监控器(virtual machine monitor, VMM)的系统监控中,通常需要截获关键内存访问事件和关键指令执行从而监控细粒度的内存访问行为.然而利用 VMM 截获内存访问行为使得

收稿日期:2018-08-17;**修回日期**:2019-02-18
基金项目:"核高基"国家科技重大专项基金项目(2014ZX01029101-002);国家自然科学基金项目(61432001);中国科学院战略性先导科技专项(XDA-Y01-01);国家自然科学基金青年科学基金项目(61802374)
This work was supported by the National Science and Technology Major Projects of Hegaoji (2014ZX01029101-002), the National Natural Science Foundation of China (61432001), the Strategic Priority Program of Chinese Academy of Science (XDA-Y01-01), and the National Natural Science Foundation of China for Young Scientists (61802374).

CPU 控制权频繁陷入 VMM 中,导致性能开销巨大.当前已有的研究为了解决该问题,在内核编译阶段修改内核源码或者直接修改内核二进制文件,将安全关键数据重定向到单独的区域以减小陷入 VMM 的频率.然而这些方法必须修改被监控系统本身,并且被监控的区域在系统运行阶段不能修改,很大程度上影响了它们的应用场景,并且不够灵活.为了解决以上问题,提出了一种运行时动态调整需要监控的安全关键内存数据的方法 DynMon,该方法对被监控的系统透明且不需要修改被监控系统.首先,通过对历史数据的收集和分析,自动学习系统运行状态和安全关键数据访问行为间的关系,将其作为安全关键数据监控策略的依据.然后,对系统运行状态实时监控,根据安全关键数据的监控策略,实时动态调整需要监控的内存访问区域,以减小不必要的监控带来的性能开销.实验结果表明:与没有动态监控策略的方法相比,该方法减小了 22.23%的额外性能开销,并且在加大内存监控规模时,并不会过大增加系统的性能开销.

关键词 安全关键数据;内存访问监控;监控策略;序列模式挖掘;事件截获

中图法分类号 TP393

现代操作系统内核空间中包含代码和内核数据,内核攻击可以通过篡改代码和内核中的数据来达到它们的攻击目的.例如通过修改中断描述符表(interrupt descriptor table, IDT)来更改系统调用的函数入口,更改可装载内核模块(loadable kernel module, LKM)链表来隐藏攻击本身的存在,更改 task_struct 结构来隐藏恶意进程等.这些攻击的本质上都是通过重写内核中的代码和数据来达到攻击目的,为了检测内核攻击,对内核空间的内存访问行为监控可以有效地帮助攻击检测.为了使内核行为监控程序不被内核中的攻击篡改,现有的研究[1-2]通常将内核攻击检测和保护程序部署在虚拟机监控器(virtual machine monitor, VMM)中来解决这个问题,如图 1 所示.这是因为 VMM 具有更高的权限,将内核攻击检测程序部署在该层,监控程序本身在更高权限的 VMM 环境中,内核层的程序感知不到它的存在,并且不易被内核中的攻击篡改.将内核攻击检测程序部署在 VMM 层,可以利用硬件虚拟化的技术截获系统事件并且获取系统运行的状态.

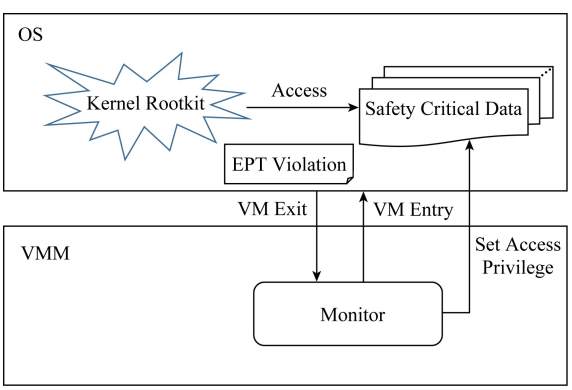

Fig. 1 Memory access monitor with VMM

图 1 基于 VMM 的内存访问监控

在 VMM 中监控内存访问事件的最大瓶颈是系统性能开销问题.我们使用硬件辅助虚拟化技术监控内存访问事件,如图 1 所示,通过设置内存的访问权限,当被监控区域内存访问事件发生时,访问权限不满足,产生可扩展页表(extended page table, EPT) Violation 使 CPU 控制权由客户机陷入到 VMM 中.客户机陷入 VMM 时会触发 VM Exit 指令,每次 VM Exit 将导致 2 000~7 000 周期的 CPU 开销[3],因此频繁陷入 VMM 会导致系统性能开销增大.对于动态区域的内存访问行为的监控,由于动态区域可以被频繁地写,导致频繁陷入 VMM 中截获内存访问行为,性能开销巨大.为了减小内存访问事件监控的开销,已有的研究通常采用的方案是只监控极少量安全关键数据[1].安全关键数据分散地分布在内核地址空间的各个位置,并且和非安全关键数据混合分布在同一页面.由于硬件支持的权限设置的粒度是页面级,而我们需要监控的安全关键数据的粒度是变量级,这就导致了和被监控安全关键数据在同一个页面内,存在大量不需要监控的非安全关键数据的访问行为,也会产生 EPT Violation,频繁陷入 VMM,从而增大系统性能开销,这就是监控粒度的鸿沟问题[4].

为了解决内存访问监控带来的巨大开销问题,研究者们提出了将安全关键数据重定向到 1 个集中的内存区域的方法.HookSafe[4]通过直接分析和更改内核的二进制文件,将被监控安全关键区域的访存指令替换为 1 个固定的跳转指令.当对该区域的内存访问时,跳转到固定的重定向代码段,在 1 个集中的影子区域内完成访问操作后跳转回程序正常运行的位置.Sentry[5]通过修改内核的源代码,在内存加载时将安全敏感数据和非安全敏感数据加载到

不同的页面中,采用插入空指令和更改内存分配函数的方法重新对内存空间进行布局.这些将安全关键数据和非安全关键数据隔离的方法都需要修改被监控系统的源码或者编译好的二进制文件,对于闭源系统不适用,修改难度大,且在一定程度上限制了保护机制的应用场景.且一旦系统被部署后,就不能再修改被监控的安全关键数据的范围,不能在系统运行时灵活更改需要监控的对象.

传统的监控内存访问行为的系统,在监控开始时对页面权限进行设置,之后其监控的粒度和范围不再改变,这种方式的监控灵活性受到了限制.Lu等人[6]提出了一种灵活的页面级的内存访问监控,首先将每个安全关键数据都分配1个独立的客户机虚拟内存页面,在系统运行时根据内存访问情况动态改变内存访问的监控粒度.他们使用的方法是利用硬件EPT支持2级页表映射的特性,通过监控历史数据中哪些客户机物理页面通常一起被访问,将这些客户机物理内存页面映射到同一机器物理页面中.这种方法仅对监控粒度做出了动态的调整,并没有对监控区域本身做出动态调整,也需要对被监控系统源码进行修改.

针对这3个问题:1)关键内存数据访问监控的性能开销大,2)当前解决该性能问题的方法需要修改被监控系统本身,3)被监控的关键内存数据在被监控系统部署前已经确定,在运行期间不能调整,本文提出了一种根据内核运行状态,动态地调整需要被监控的内存区域的方法DynMon.首先,收集系统运行时的事件序列、时间信息和安全关键数据访问行为的相关数据,根据系统的历史数据,利用时序关联规则算法,自动地学习安全关键数据被访问的场景,建立系统运行状态、时间和安全关键数据访问行为之间的关系作为监控策略.根据安全关键数据访问行为的动态监控策略,在VMM层中截获被监控系统的事件,收集这些事件.当在VMM中收集到的事件序列满足动态监控策略中的条件时,在相应监控策略规定的一段时间内,触发对特定的安全关键数据的监控.

本文主要贡献有3个方面:

1) 提出一种根据内核运行状态,动态地调整需要被监控的内存区域的方法.当根据系统当前运行状态可以判断某些安全关键数据不会被访问时,不设置监控安全关键数据所在的页面的访问权限,避免了该页面中非安全关键数据的访问导致频繁陷入VMM,从而减小了额外产生的系统开销.该方法不依赖于内核源码,不需要重新编译内核源码或者修改内核二进制文件,并且对被监控系统透明.

2) 带有时间信息的动态监控策略自动生成,动态监控策略指的是在系统运行过程中,并不是始终对所有安全关键区域的内存访问行为进行监控,而是根据系统状态自动地进行动态调整.

3) 在基于硬件虚拟化技术的开源虚拟化平台Xen上实现了原型系统,可以容易地移植到任何支持硬件虚拟化技术的VMM平台.通过实验评估,验证了DynMon的有效性.使用了动态安全关键数据监控策略的方法比静态监控的方法降低了22.23%的性能开销,并且可以检测到实验中考虑的内核攻击.当被监控的安全关键数据的范围增大时,系统的性能开销依然在可以接受的范围内.

1 相关工作

近年来,为了监控和保护内核中的动态数据,研究者通过对内核区域设置访问权限来达到监控和保护的目的.然而,这些被监控的动态数据被动态加载到内核堆中的,和其他不需要被监控的内核数据共用同一区域,而现代硬件支持的内存权限设置粒度是页面,这导致该页面中所有数据的访问都会陷入到VMM,使得系统性能开销巨大.Wang等人[4]提出的了保护粒度鸿沟问题,即需要被保护的数据是字节级的,而硬件只提供页面粒度的保护.他们提出了HookSafe,通过修改内核的二进制文件,将钩子点的访问进行重定向,把钩子点集中到1个特定的内存区域,对该集中的内存区域进行写保护;Sentry[5]为了解决内存访问监控开销巨大的问题,通过修改内核源码,在内核数据加载时将安全关键数据和非安全关键数据分离在不同的页面,这样可以对安全关键数据和非安全关键数据实施不同的访问控制策略;Li等人[7]提出了一种通过修改编译器的方式,将钩子点索引到1个只读的跳转表中,只有符合内核的控制流图的方式才可以访问该表中被保护的钩子点.这些方法都需要修改被监控系统的源码,并且重新编译,或者直接修改被监控系统的二进制文件,对于真实场景下的被监控系统,通常是不接受系统本身被修改的,并且方法的可移植性差.

另一类解决该问题的方法是使用页面粒度的监控,Lu等人[6]提出了一种灵活的页面粒度的内存访问行为的监控方法,为了解决保护粒度的鸿沟引起的性能问题,首先将被监控的对象分配到独立的

虚拟地址页面中,为了防止每个监控对象都分配1个独立页面造成物理页面的浪费,在内存分配时,将1个页面分为多个虚拟页面,再将多个虚拟页面映射到同一物理页面,并利用EPT技术在运行中动态调整内存访问截获的粒度.该方法依然需要修改被监控系统的内核源码,并且只能在运行过程中动态控制监控粒度,并不能在动态调整监控区域本身.

由于内核中的对象对计算机系统起着至关重要的作用,近年来研究者对内存中数据的访问模式进行研究来更有效地保护系统安全.在入侵检测系统的研究中,大量的工作使用行为模型描述正常的行为和恶意行为.一种常用的方法是使用系统调用序列和系统调用的参数来分别刻画正常行为和恶意行为[8];一些研究者使用基于图的模型,用系统调用来描述系统行为[9];Rhee等人[10]使用系统访问的模式来描述系统行为;Mao等人[11]使用page-rank算法评估内核对象的重要性;Xu等人[12]使用机器学习的方法分析了虚拟内存访问.

入侵检测系统被用来检测系统中潜在的安全问题.研究者们将大量的数据挖掘方法应用到入侵检测领域,关联规则可以用来查找到给定攻击行为中的频繁子集,被用来挖掘恶意行为的模式.文献[13]中提出一种自动的方法学习带有时序的系统状态,并建立一种混合的入侵检测系统;Lee等人[14]提出了一种系统自动化的入侵检测框架,使用数据挖掘算法计算审计数据中的活跃模式,并从这些模式中抽取出可以用来预测入侵的特征;文献[15]提出了一种带权重的序列挖掘的方法用在入侵检测系统中,对于更倾向于恶意篡改行为的属性,增大其在序列挖掘算法中的权重.

2 背景知识介绍

在本节中,我们描述硬件辅助虚拟化(hardware assisted virtualization, HAV)的背景和基于HAV的内存监控的性能问题.

2.1 硬件辅助虚拟化

为了可以在高权限级下监控系统行为,我们使用了硬件虚拟化技术.硬件虚拟化技术可以支持不更改操作系统,并且带来的性能开销更小.目前主流的CPU都支持硬件虚拟化技术,如Intel的VT-x技术[16]、AMD的AMD-V技术,它们的实现原理类似,本文中使用Intel的VT-x技术为例介绍硬件虚拟化技术.

为了支持CPU虚拟化,VT-x提供了2个模式:VMX(virtual machine extension)根模式和VMX非根模式,分别用来支持客户机和VMM,如图2(a)所示.这2种模式可以通过1组指令进行相互切换:VM Exit和VM Entry.当1个高权限的指令在非根模式下执行时,处理器通过VM Exit指令将其切换到根模式下执行.在根模式下,虚拟机监控器处理VM Exit后,通过VM Entry指令重新返回到虚拟机层.在截获内存访问行为时,设置某个页面的访问权限为不可访问,当出现访问该页面的操作时,会触发EPT Violation,从而使得CPU通过VM Exit从客户机陷入到VMM中.在VMM中处理EPT Violation后,通过VM Entry重新将CPU控制权返回给客户机.

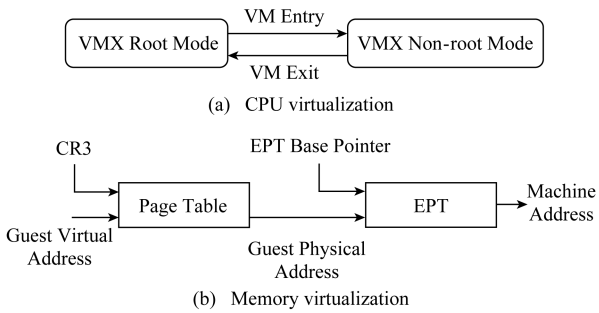

Fig. 2 Hardware assisted virtualization

图2 硬件虚拟化

扩展页表(extended page table, EPT)使用硬件来虚拟内存管理单元(memory management unit, MMU),如图2(b)所示.MMU利用CR3指向的页表,通过查找页表将客户机虚拟地址翻译成客户机物理地址,客户虚拟地址是指客户机操作系统使用的虚拟地址,客户物理地址是客户机的操作系统使用的物理地址.当EPT被使能后,利用EPT页表指针(EPT base pointer, EPTP)指向的扩展页表EPT,通过1组EPT页表查找操作,将客户物理地址翻译成真实的机器地址.即使用了EPT技术后可以支持客户虚拟地址,客户物理地址和真实机器地址之间的转换[17].EPT也可以设置软件访问特定地址内存数据的权限,包括读、写和执行.任何不在权限允许访问内的访问操作都会触发EPT Violation,然后导致VM Exit.

2.2 性能开销问题

使用硬件辅助虚拟化技术时,特权事件会导致CPU控制权从客户机陷入到VMM中.当这些事件触发VM Exit时,将导致将近2 000~7 000周期的

CPU 开销[3],这是硬件虚拟化技术增大系统开销的1个原因.基于 EPT 的内存访问监控会加大此方面的开销,因为硬件设置页面访问权限的最小粒度是页,利用 EPT 设置内存访问权限后,任何在这个页面中的超出访问权限的操作都会触发 VM Exit,导致监控内存访问行为会极大增加 VM 陷入 VMM 的频率.动态内存区域会被频繁读写,页面内大量的内存访问的截获产生 EPT Violation,使得 VM Exit 被频繁触发.文献[4]中,作者在监控钩子点的内存访问行为时,发现 14 个页面中仅仅存在 50 个钩子点,其中最差的情况下,1 个页面中只有 1 个钩子点.即在监控的 1 个动态页面中,只有 4 B 是真正需要捕获的内存访问,其他 4 092 B 的动态数据的访问行为由于 EPT 权限设置也需要触发 VM Exit,导致 VM Exit 触发特别频繁,系统开销巨大.本文的工作试图通过动态设置内存访问行为监控的时机,尽量在真正需要捕获的安全关键数据被访问时对该内存区域进行监控,减少内存访问监控时 VM 陷入到 VMM 的频率,从而减小这部分的性能开销.

3 内存关键数据动态监控方法

本文提出的内存关键数据动态监控方法 DynMon 主要包括事件监控、内存访问监控和动态监控策略,如图 3 所示.其中动态监控策略是依据线下分析历史运行数据得出的,分析的数据包含事件序列、时间影响和安全关键数据的访问行为.利用时序关联规则挖掘出事件序列、时间影响和安全关键数据访问行为间的关系作为动态监控策略.事件监控利用硬件虚拟化技术和内存虚拟化技术在运行时对操作系统层的事件进行截获,并收集这些事件序列.当收集到的事件序列和动态监控策略匹配时,根据动态监控策略在一定的时间范围内,触发内存访问监控特定的安全关键数据的内存访问行为.内存访问监控收到特定的安全关键数据的监控请求后,利用内存虚拟化技术限制安全关键数据所在页面的内存访问权限,当被监控的系统试图访问该数据时,触发 EPT Violation 陷入到 VMM 中,从而捕获该数据的访问行为.这样,就可以减少和安全关键数据无关的动态数据的监控,从而减小系统性能开销.我们将详细介绍动态内存监控系统中的 5 个关键组件.

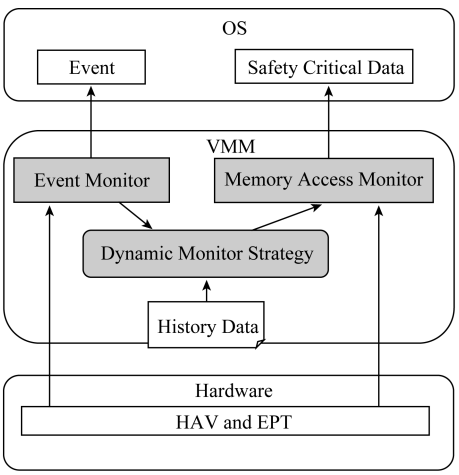

Fig. 3 Dynamic memory access monitor architecture

图 3 动态内存监控框架

3.1 事 件

VMM 可以截获包括硬件相关的和软件相关的事件,硬件相关的事件主要包括寄存器访问、I/O 访问、内存访问等,软件层的事件主要包括中断、异常和系统调用.在本文的工作中,我们主要关注的是寄存器访问事件,内存访问事件和系统调用事件.

定义 1. 寄存器访问事件.寄存器访问事件是对系统中的关键寄存器访问,表示为$\langle r, op\rangle$,其中 r 表示被访问的寄存器,op 表示被访问的类型.包括控制寄存器(control register, CR)、调试寄存器(debug register, DR)、特殊模块寄存器(model special register, MSR)和全局描述符表寄存器(global descriptor table register, GDTR)等.通过截获寄存器访问事件,可以感知系统状态的变化,如 CR3 中记录了运行进程在虚拟地址空间的页目录信息,一旦进程发生了切换,CR3 中的内容也将被改变.通过截获 CR3 的写事件,可以监控进程切换的行为.

定义 2. 内存访问事件.内存访问事件只的是对内存中数据进行读、写、执行的操作,表示为$\langle m, l, op\rangle$,其中 m 表示被监控的关键内存区域的名称,l 表示关键内存区域 m 在内存中的位置,l 的范围表示为 $l\in[start\_address, end\_address]$.这些地址指的是操作系统中的虚拟地址,$op$ 表示访问的类型,如读、写或者执行.攻击程序如果要破坏系统的完整性,不可避免地需要通过内存的写事件,内存访问事件是非常关键的系统事件.

定义 3. 系统调用事件.系统调用提供了用户程序和系统函数之间的关键接口,用户通过系统调用可以使用内核定义的特权操作.本文中将系统调用表示为$\langle syscall_num, arg_1, arg_2, \cdots, arg_n\rangle$,其中 $syscall\_num$ 指系统调用号,arg_i 指第 i 个系统调用的参数.攻击程序通常在 1 个系统调用后破坏内

核的完整性,大量内核攻击通过破坏系统调用来达到它们的攻击目的.

定义 4. 中断和异常.中断提供了操作系统和外设之间的交互,在计算机运行中,出现意外情况需要计算机进行干预,计算机中断当前执行的程序,进行中断处理程序,处理完成后,转回之前暂停的程序继续执行.异常指 CPU 执行指令内部的事件出现错误,转入到异常处理程序来处理异常.表示为〈*intr*,*vec*〉,其中 *intr* 表示中断或者异常,*vec* 表示中断和异常向量号.

3.2 安全关键数据

安全关键数据是指有可能影响到系统安全性的数据,如 *task_struct* 存储了进程相关信息,LKM 链表存储了 Linux 内核模块的关键信息.一些攻击通过修改 *task_struct* 中的成员信息来达到它们的攻击目的,如:修改 *flag* 域以指示隐藏哪些进程,修改 *uid* 域以提升程序权限,通过修改 LKM 链表的头指针来隐藏攻击模块本身的存在.安全关键数据可以分为静态数据和动态数据,对于静态数据要求始终不能被写访问,静态数据正常情况下不会被写访问并且静态数据是集中存储在静态区域中的,实时监控所有静态区域不会造成频繁陷入 VMM 导致性能开销增大.而动态安全关键数据是和动态非安全关键数据通常混合在一起的,即 1 个页面中通常同时存在安全关键动态数据和非安全关键数据.动态数据可以被合法地进行访问,这也增加了检测动态数据是否被非法篡改的难度.

3.3 动态监控策略

DynMon 通过使用动态监控策略,根据系统实时运行状态,对监控的内存区域进行调整,这样如果根据当前系统状态可以判断不存在访问该安全关键区域的行为,就不对相应区域监控,从而减少该区域中的非安全关键数据陷入产生的不必要性能开销.为了得到合理的动态监控策略,首先收集系统运行时的事件序列、时间信息和安全关键数据的访问行为,通过数据分析获取事件序列、时间信息和安全关键数据访问行为间的关系,得出动态监控的策略为:当某个事件序列发生后的某段时间内对相应的安全关键数据的访问行为进行监控.

通过观察,发现安全关键数据的访问通常伴随着一系列的行为,如 LKM 链表的读写是由 Linux 内核模块加载和删除导致的,在系统调用 *init_module* 和 *execve* 后改写 LKM 链表.再例如 *task_struct* 中内容的修改需要经过一系列系统调用 *execve*,*open*,*close* 之后才对 *task_struct* 中的内容进行修改来达到攻击内核的目的.由于系统事件和安全关键数据的访问之间存在这样的关系,通过分析历史数据来得到它们之间的联系,并将其作为动态监控策略.

定义 5. 事件序列.事件序列是多个不同的事件通过不同的关系组合起来的,我们使用模式语言来描述.在不同模式之间存在 4 种关系:基本事件关系、顺序关系、选择关系和重复关系.不同的行为模式间的关系:

1) $pat.reg\_op|mem\_op|system\_call|intr$ 表示基本事件,*reg_op*,*mem_op*,*system_call*,*intr* 分别表示寄存器访问事件、内存访问事件、系统调用事件、中断和异常.

2) sequencing. pat1;pat2 表示 pat2 紧接着 pat1 出现.它表示 2 个事件序列间的顺序关系.

3) alternation. pat1 ‖ pat2 表示 pat1 或者 pat2 同时出现,它表示 2 个事件序列间的选择关系.

4) repetition. pat^* 表示 pat 出现了 0 次或者多次,它表示 1 个事件序列重复出现了多次.

定义 6. 动态监控策略.本文中将动态监控策略定义为:当某个事件序列发生后的某段时间内,对相应的安全关键数据的访问行为进行监控.表示为 $\{s_1,s_2,\cdots,s_n,[t_1,t_2]\}\rightarrow\{security\_data_j,access_j\}$,其中:

1) $s_1,s_2,\cdots,s_n$ 表示定义 5 中定义的事件序列.

2) $[t_1,t_2]$表示在事件序列发生后的一段时间区间.

3) $security\_data_j$ 表示某个安全关键数据.

4) $access_j$ 表示访问行为包括读、写和执行.

即在序列$\{s_1,s_2,\cdots,s_n\}$发生后的经过 t_1 到 t_2 时间内,需要对安全关键数据 $security\_data_j$ 的 $access_j$ 访问行为进行监控.由于在系统运行中时间点 t_1 和 t_2 并不容易评估,我们使用被监控的 VM 陷入 VMM 的次数来表达关键数据的访问行为监控和事件序列$\{s_1,s_2,\cdots,s_n\}$的相对时间关系.$\{s_1,s_2,\cdots,s_n,[n_1,n_2]\}\rightarrow\{security_data_j,access_j\}$,其中$[n_1,n_2]$表示在事件序列发生后 n_1 次 VMM 陷入事件 n_2 次 VMM 陷入事件之间,需要对安全关键数据 $security\_data_j$ 的访问行为进行监控.

为了挖掘出定义 6 所述的事件序列和安全关键数据访问的关系作为安全监控策略,我们需要经过图 4 所示的 4 个步骤.

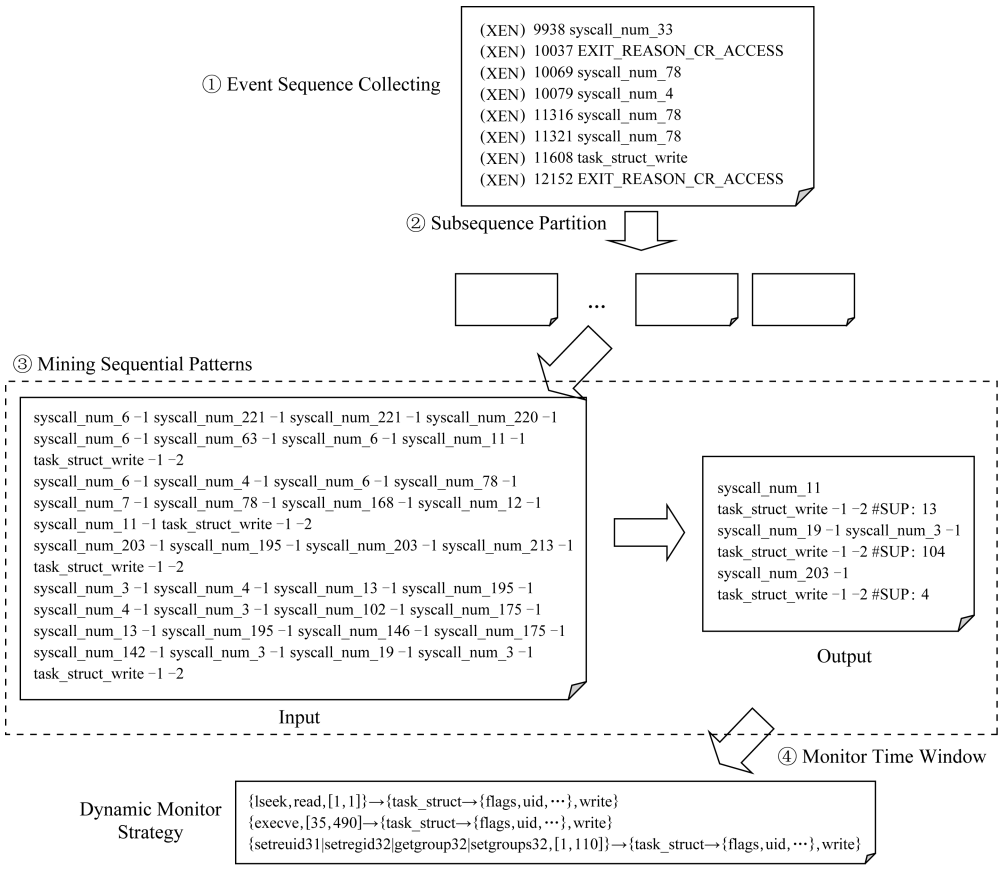

Fig. 4 Safety-critical data access behavior analysis

图 4 安全关键数据访问行为分析

步骤①. 在被监控系统运行过程中,利用硬件虚拟化技术截获事件序列和安全关键数据访问行为的序列.

步骤②. 将完整序列合理的划分为多个子序列.

步骤③. 将子序列的集合转换为时序关联规则可以处理的输入格式,利用带有时序关系的数据挖掘算法得出事件序列和安全关键数据的访问行为的关系.

步骤④. 根据序列关联规则挖掘出的频繁序列结合原始收集的带有时间信息的数据,抽取出在某个事件序列完成后会存在安全关键数据的访问行为的时间范围.根据上述一系列数据分析可以得到在某个序列完成后,在一段时间内会存在某个安全关键数据的访问行为.

3.3.1 事件序列收集

首先我们在系统运行状态下,收集系统运行时的事件,包括控制寄存器的访问、系统调用、中断异常和安全关键数据的访问.2.2 节中已经描述,对动态安全关键数据访问行为的截获使得性能开销急剧增大,在我们的事件收集中同时对多个动态区域的内存访问进行监控会因为性能开销过大导致系统崩溃,我们每次只对 1 个安全关键数据所在页面的访问行为进行监控.在算法 1 中,描述了带有时序信息的事件收集过程,从事件收集开始,使用 1 个计数器来记录陷入 VMM 中的次数,每 1 个收集到的事件的格式为($counter$,$event$),其中 $counter$ 为从开始收集事件到事件 $event$ 发生时陷入 VMM 的次数,在开始监控时将 $counter$ 初始化为 0,每次陷入 VMM 处理时将 $counter$ 进行加 1 处理.

算法 1. 事件序列的收集算法.

输入:计数器 $counter$、运行时系统运行状态;

输出:事件序列$\{(counter_1,event_1),(counter_2,event_2),\cdots,(counter_n,event_n)\}$.

① $counter=0,i=1$;

② while 事件收集 do

③ if (VM Exit)

④ if (检查到事件)

⑤ $event_i=VM\ Exit\ reason$;

⑥ $counter_i=counter$;

⑦ $i++$;

⑧ end if
⑨ $counter$++;
⑩ 处理陷入的原因;
⑪ end if
⑫ 恢复事件监控;
⑬ end while

3.3.2 事件子序列的划分

3.3.1 节中收集到的是系统运行时完整的事件序列,我们需要对其进行分割,分割成事件序列片段,我们以标志性事件作为事件序列的分割点.由于我们需要挖掘的是事件序列和关键数据访问行为之间的关系,关键数据的访问行为就是每个事件序列的结束位置.事件序列开始的位置根据关键数据采用 2 种方式来确定:1)基于标志性事件的方法;2)基于窗口的方法.针对一些特征性明显的关键数据,如 LKM 链表头的写访问在起始位置伴随着 2 个系统调用 $init\_module$ 或者 $delete\_module$,我们使用这些关键事件作为序列的起始位置.对于一些关键数据,并没有一些明显的标志性起始事件如 $task\_struct$ 中安全关键数据的访问,我们使用窗口的方法,将距离结尾固定长度的事件序列截取出来作为一个事件序列.

3.3.3 时序关联规则分析

由于我们抽取的事件序列是带有前后时序关系的,时序关联规则[18]可以用于挖掘这种带有前后时序关系的序列.时序关联规则可以在 1 组带有时序信息的序列中挖掘出子序列模式,当某个模式出现的次数超过给定的阈值时,它是频繁模式.频繁模式出现的次数和所有序列个数的比值称为支持度,给定的阈值称为最小支持度.研究者们已经提出了许多时序关联规则的算法,如 AprioriAll[18],GSP[19]是广度优先的算法来查找频繁时序模式;Spade[20],Spam[21]是通过深度优先的方法查找;FreeSpan[22],PrefixSpan[23]是使用模式增长的方法查找.这些传统的方法会挖掘出大量的子序列模式,为了减少挖掘出的冗余的频繁子模式,基于闭集的序列模式挖掘方法(closed sequence patterns, CSP)[24]被提出,该方法挖掘出的频繁序列模式中,1 个模式不被任何 1 个其他模式所包含.本文做时序关联规则分析时,使用了基于闭集的方法 CloFast[24],该算法对挖掘大规模的长序列更为有效.如图 4 所示,需要先将收集到的数据转化为 CloFast 的输入格式,通过 CloFast 挖掘出时序序列的频繁模式.对于 CloFast,输入包括事件子序列集和最小支持度,在本文中使用 CloFast 算法挖掘频繁序列模式如算法 2 中所描述.最小支持度的选择根据第 4 节中实验得出的数据确定,选择性能开销可以接受,并且可以截获到绝大多数关键数据访问行为的最小支持度作为 CloFast 的最小支持度.

算法 2. 基于 CloFast 算法的安全关键数据.

输入:最小支持度 min_sup、事件子序列集 $\{\{s_{11},s_{21},\cdots,s_{n1},security\_data_1,access_1\},\cdots,\{s_{1m},s_{2m},\cdots,s_{nm},security\_data_m,access_m\}\}$;

输出:序列模式闭集序列模式闭集 $\{s_1,s_2,\cdots,s_n\}\rightarrow\{security\_data_j,access_j\}$.

① 频繁项 FI:识别支持度大于 min_sup 的频繁 1-项集;
② 频繁项闭集 CFI:识别出 FI 中支持度大于 min_sup 的频繁项闭集;
③ for each $cfi\in CFI$
④ $n=createNode(cfi)$;
⑤ $addChildNode(T,root(T),n)$;
⑥ end for
⑦ for each $child\in children(T,root(T))$
⑧ 深度优先搜索频繁序列(T,$child$,min_sup);
⑨ end for
⑩ return 序列模式闭集 $\{s_1,s_2,\cdots,s_n\}\rightarrow\{security\_data_j,access_j\}$.

3.3.4 监控窗口的确定

对于关键数据访问事件监控的时间窗口的确定,我们使用序列中最后 1 个事件到关键数据访问事件的时间距离来确定.窗口的开始时间是频繁序列模式中最后 1 个事件到第 1 个关键数据访问事件间的最小时间间隔,窗口的结束时间是频繁序列模式中最后 1 个事件到该模式中最后 1 个关键数据访问事件的最大时间间隔.由于在虚拟机监控器 VMM 中直接度量时间间隔,时间间隔不易精确度量并受到虚拟机的陷入影响,我们使用一种更易度量的相对时间度量方式,使用陷入 VMM 的次数作为相对时间.由于在图 4 中步骤①事件收集时,收集到的每个事件信息包括事件发生时陷入 VMM 的次数、以这个次数作为事件发生时的时间信息.

3.3.5 安全关键数据访问的动态监控策略

根据 3.3.1 节到 3.3.4 节中安全关键数据访问行为的分析,我们可以得到一种动态监控安全关键数据访问的策略.在特定事件序列出现后的一段时间内,将相应的安全关键数据所在页面设置为不可

访问,当超过监控时间后,取消对该安全关键数据所在页面的访问限制.即在匹配到事件序列$\{s_1,s_2,\cdots,s_n\}$后,陷入 VMM 的次数在$[n_1,n_2]$区间内,对相应的安全关键数据的特定访问行为(如读、写或者执行)进行监控.这样就可以在运行时,根据系统的运行状态动态调整需要监控的安全关键数据,从而减小不必要的内存动态区域访问行为监控造成的巨大性能开销.由于不同类型的安全关键数据具有不同的特征,相应的监控策略也不同.我们对安全关键数据进行分类:分为静态数据和动态数据,动态数据又包含控制流数据,链表类数据,非控制流数据等.对于静态数据,由于它所在的区域内都是静态数据,在没有恶意攻击存在的情况下不存在写行为,所以始终监控该区域的写行为不会导致系统频繁陷入 VMM,几乎不会增加系统性能开销.当出现写行为时,通常可以认为是恶意程序篡改了内核中的静态数据,所以从系统初始化稳定开始监控后,就始终设置静态数据的写行为监控.表示为 $monitor\_init\rightarrow\{static\_data,write\}$,即从监控开始,就对静态区域 $static\_data$ 进行写访问行为的监控.

对于动态数据,我们使用 3.3.1 节到 3.3.4 节中数据分析的结果$\{s_1,s_2,\cdots,s_n,[n_1,n_2]\}\rightarrow\{security_data_j,access_j\}$作为动态内存区域访问行为监控的策略,我们以几个安全关键数据为例,表 1 中列了通过对收集事件序列,时间信息和相应的关键数据访问行为分析,得到的动态监控策略.以 LKM_list_head 为例,当新的内核模块被加载或者被卸载时,会将其 $module$ 结构插入到 LKM 链表的头指针位置(本文中使用 LKM_list_head 描述),根据时序关联规则 *CloFast* 挖掘的结果可以看出存在 2 种频繁序列,即$\{init_module,close,socketcall,LKM_list_head_write\}$和$\{delete_module,LKM_list_head_write\}$.通过对其中系统调用事件的分析可以看出这 2 个序列的语义分别是加载内核模块时需要进行 $init\_module$(加载内核模块的系统调用),$close$(文件关闭的系统调用)和 $socketcall$(套接字相关系统调用总入口),卸载内核模块时,需要 $delete\_module$(卸载模块的系统调用).监控的时间范围是由 3.3.4 节中根据数据分析确定的窗口大小,在该时间范围内对 LKM 链表的头指针进行内存访问监控.

Table 1 Partial Monitor Strategies of Critical Data

表 1 部分关键数据的监控策略

Critical Data	Monitor Strategy
LKM_list_head	$\{init\_module,close,socketcall,[10,100]\}\rightarrow\{LKM\_list\_head,write\}$ $\{delete\_module,[20,186]\}\rightarrow\{LKM\_list\_head,write\}$
$task\_struct$	$\{lseek,read,[1,1]\}\rightarrow\{task\_struct\rightarrow\{flags,uid,gid\},write\}$ $\{execve,[35,490]\}\rightarrow\{task\_struct\rightarrow\{flags,uid,gid\},write\}$ $\{setreuid32\|setregid32\|getgroups32\|setgroups32,[1,110]\}\rightarrow\{task\_struct\rightarrow\{flags,uid,\cdots\},write\}$
$proc\_sys\_ops$	$\{stat64,open,[1,8]\}\rightarrow\{proc\_sys\_ops,write\}$ $\{lstat64,open,[1,8]\}\rightarrow\{proc\_sys\_ops,write\}$ $\{access,access,access,open,[1,9]\}\rightarrow\{proc\_sys\_ops,write\}$
$proc\_dentry\_ops$	$\{stat64,open,[1,8]\}\rightarrow\{proc\_dentry\_ops,write\}$ $\{lstat64,open,[1,8]\}\rightarrow\{proc\_dentry\_ops,write\}$ $\{access,access,access,open,[1,9]\}\rightarrow\{proc\_dentry\_ops,write\}$
$proc\_mnt$	$\{fstat64,read,close,[3,155]\}\rightarrow\{proc\_mnt\_ops,write\}$

3.4 事件监控

事件监控在系统运行时截获系统的事件,并收集事件.VMM 可以截获和软硬件相关的多种事件,在本文原型系统的事件监控,我们主要关注的是寄存器访问事件和系统调用的截获.

对于寄存器相关的操作,本文关注于控制寄存器 CR、调试寄存器 DR,特殊模块寄存器 MSR 和全局描述符表寄存器 GDTR.通过截获寄存器访问事件,可以感知到操作系统状态的变化,如 CR3 中记录了运行进程在虚拟地址空间的页目录信息,一旦进程发生了切换,CR3 中的内容也将被改变.通过截获 CR3 的写事件,可以监控进程切换的行为.寄存器访问行为是 1 种高权限操作,我们利用硬件辅助虚拟化技术截获寄存器访问行为.虚拟机控制块(virtual machine control data structure, VMCS)结构中的客户状态域可以管理客户寄存器的访问权限,当寄存器访问事件发生时,VM Exit 将被触发.因此,我们通过 VMCS 结构中的 *VM Exit reason* 域解析可以截获寄存器访问事件.

中断和异常是操作系统的重要事件,用于处理操作系统和外部事件的交互,内部的错误和陷入.硬件虚拟化技术支持对中断和异常的截获,中断的截

获通过设置 VMCS 域中的虚拟机执行控制域中的“外部中断陷入位”,可以使外部中断发生时陷入到 VMM,DynMon 通过对陷入原因解析可以截获到中断和中断号.对于异常的截获,硬件虚拟化提供了 1 个数据结构 Exception Bitmap,可以通过该结构具体设置对哪些异常进行截获.例如需要截获 *page_fault* 异常,需要将 Exception Bitmap 中的第 14 位设置为 1,当发生 *page_fault* 时陷入 VMM,DynMon 通过解析陷入的原因截获该异常事件.

系统调用提供了用户程序和系统函数之间的关键接口,入侵检测的研究表明许多攻击通过破坏系统调用来达到它们的攻击目的[8].本文在事件监控中截获系统调用和系统调用中的相关参数.系统调用的截获需要根据系统调用的实现不同将系统调用分为 2 类:软中断和快速系统调用.在截获软中断时,将原有的 int80 在中断描述符表中的项存储起来,然后将该位置指向 1 个不可执行的地址.这样,当执行系统调用时,会产生 EPT Violation 陷入 VMM.在 EPT Violation 的处理函数中我们记录系统调用事件,并将 EIP 寄存器的值设置成之前保存的原始 int80 在中断描述符表中的地址,这样就模拟了系统调用的执行.对于快速系统调用,使用硬件指令完成系统调用,现代操作系统通常在 32 b 系统中使用 SYSENTER/SYSEXIT,在 64 b 系统中使用 SYSCALL/SYSRET.对于 SYSENTER 的截获,将 IA32_SYSENTER_EIP MSR 寄存器中的值设置为 1 个不可执行的地址,同软中断的截获,当产生系统调用时会触发 1 个 EPT Violation 陷入 VMM.之后的处理同软中断的处理.记录系统调用事件的详细信息,将 EIP 寄存器的值设置成之前保存的 IA32_SYSENTER_EIP MSR 中的地址以模拟系统调用的执行.对于 SYSCALL 指令的截获,首先将 EFER 寄存器中的 SCE 位(该位表明是否允许 SYSCALL/SYSRET 指令)置为 0.当发生系统调用时,产生 *invalid_op* 异常陷入到虚拟机中,我们在虚拟机中通过陷入信息解析出该异常,通过寄存器的值获取系统调用的相关信息,然后将 SCE 位恢复为 1 使得该条指令可以正常执行.并设置 *TF*(trap flag)位,执行完 SYSCALL 后由于 *trap_debug* 异常重新陷入 VMM.在 *Trap Debug* 处理程序中将 SCE 重新置为 0,继续对 SYSCALL 进行截获.

3.5 内存访问监控

为了截获内存访问行为,首先需要确定监控哪些页面的内存访问事件.传统的方法是在初始化系统监控程序时就已经决定对哪些位置的关键内存访问行为进行监控,在之后的监控过程中需要监控的内存范围不改变.为了减小监控内存访问带来的系统性能开销,本文提出了动态安全关键数据内存访问行为的监控方法.

算法 3. 内存访问行为的动态监控.

输入:动态监控策略$\{s_1,s_2,\cdots,s_n,[n_1,n_2]\}\rightarrow\{security_data_j,access_j\}$.

输出:$set\_mem\_access(security\_data_j,access_j)$.

① 系统运行时收集事件;
② if (事件序列匹配$\{s_1,s_2,\cdots,s_n\}$)
③ $counter=0$;
④ end if
⑤ if (VM Exit)
⑥ $counter=counter+1$;
⑦ if ($counter=n_1$)
⑧ $set\_mem\_access(security\_data_j,access_j)$;
⑨ end if
⑩ if ($counter=n_2$)
⑪ $set\_mem\_access(security\_data_j,access\_rxz)$;
⑫ end if
⑬ end if

算法 4. 关键数据访问的截获.

输入:安全关键数据的地址;

输出:被访问的安全关键数据.

① if (安全关键数据所在页面被访问)
② 触发 EPT Violation;
③ if (VM Exit 地址=安全关键数据地址)
④ 记录该安全关键数据被访问;
⑤ end if
⑥ $eflag\models TF$;
⑦ end if
⑧ 下一条指令执行陷入 VMM;
⑨ $set\_mem\_access(security\_data_j,access_j)$.

算法 3 描述了安全关键数据的动态监控策略,当事件监控模块收集到的事件序列满足动态监控策略中的事件序列$\{s_1,s_2,\cdots,s_n\}$时,在 VMM 中开启计数器 *counter* 的功能.*counter* 初始为 0,*counter* 记录被监控系统陷入 VMM 的次数.当 $counter=n_1$ 时,将相应安全关键数据所在页面的 *access* 权限从 EPT 页表中移除;当 $counter=n_2$ 时,系统取消对该安全关键数据的监控,即在 EPT 页表中重新添加被监控安全关键数据所在页面的访问权限.

算法 4 描述了在算法 3 中 $n_1 \sim n_2$ 之间,对安全关键数据进行监控,一旦有事件试图访问被监控的页面,将触发 EPT Violation,导致 CPU 从 VM 陷入到 VMM 中去处理 EPT Violation.陷入 VMM 后,内存访问监控模块判断产生 EPT Violation 的客户机物理地址是否属于我们需要监控的安全关键数据的客户机物理地址范围,如果属于,则记录相应的被访问的安全关键数据.在 EPT Violation 处理函数中,将页面的读写权限恢复,使得内存访问的指令可以重新执行.最后,我们需要恢复对该页面的访问监控,即在下一条指令执行之前,重新将该页面的读写权限移除.为了实现该功能,在 EPT Violation 处理函数中,设置 TF 位,并且在处理异常的 bitmap 结构中使能 *trap_debug* 位,这样在内存访问指令重新执行后会再陷入 VMM 中.在 *trap_debug* 异常处理函数中,我们重新将该页面的读写权限移除.

通过上述方法,DynMon 实现对安全关键数据进行数据级的实时监控.例如需要监控的关键进程信息 *task_struct* 中 *flag* 域的客户机线性地址为 0xc03d63ac 的 1 B,需要先将该客户机线性地址转换为客户机物理地址,将该页面设置为不可访问.当关键信息 *task_struct* 的 *flag* 域被写时,产生 EPT Violation 并将导致该异常的客户机物理地址作为陷入信息传递给 VMM,DynMon 在 VMM 中解析出产生异常的客户机物理地址,和被监控的客户机物理地址进行对比,相同时认为监控到了该关键数据的写操作,并记录下该访问行为.当该页面中 1 个非安全关键数据 *task_struct* 中的 *used_math* 域被写访问,由于该数据所在的页面被设置了访问权限,陷入 VMM 后,DynMon 判断该数据的客户机物理地址不在监控范围内,不记录该内存访问行为.DynMon 可以根据数据的线性地址或者线性地址的范围,控制被监控的数据.

4 实验与验证

为了验证本文提出方法的有效性,本节首先说明了实验环境和实验中使用的攻击程序.然后从 3 个方面说明 DynMon 的有效性:1)和不采用动态监控策略的方法做性能比对,验证 DynMon 是否可以大幅降低动态区域的安全关键数据监控的性能开销;2)增大监控范围,评估 DynMon 的性能开销是否可以接受;3)当监控到安全关键数据的访问时,触发完整性检查,验证是否可以检测到内核攻击行为.

4.1 实验设置

4.1.1 实验环境设置

本文提出的方法原型系统实现在基于硬件虚拟化的 Xen 平台上.所有的实验都是在 Intel Core i7-4710MQ CPU 上实现的,带有 2.5.0 GHz 的主频和 8 GB 内存.使用的 Xen 平台是 Xen 4.4.0 version,使用的被监控硬件虚拟机 HVM 的操作系统分别是 32 位 Linux 2.6.24 和 64 位 Linux3.11.0.

4.1.2 对比实验设置

在实验中我们使用本文提出的基于动态监控策略的安全关键数据监控的方法和从内存监控开始就始终监控安全关键数据的方法进行对比对比.我们使用 2 组实验名称:

1) StaticMon 表示从监控开始始终对安全关键数据所在的内存页面进行访问监控[1].

2) DynMon 表示本文提出的方法对安全关键数据采用动态监控的策略进行访问监控.由于动态策略的学习使用了时序关联规则算法 CloFast,不同支持度的设置会影响到挖掘出的监控策略,在实验中我们设置了不同的最小支持度,比较最小支持度不同对性能和监控效率的影响,并根据实验选择出合适的最小支持度.DynMon_0.1 表示最小支持度为 0.1 的时序关联规则算法 CloFast 挖掘出的动态监控策略.依次类推,DynMon_0.2,DynMon_0.3,DynMon_0.4 分别表示使用最小支持度为 0.2,0.3,0.4 的 CloFast 挖掘出的动态监控策略.

4.1.3 内核攻击程序设置

表 2 中列举了我们的实验评估中使用的 3 个攻击程序,包括它们攻击的对象、需要检测的被攻击的内核数据和被攻击的内核数据类型.我们选择这 3 个攻击程序是因为它们攻击的对象包含了内核代码、

Table 2 Rootkits Used in Evaluation

表 2 实验评估中使用的攻击程序

Rootkit	Attacked Data	Attacked Data Type
Enyelkm	content in system call function	Static Code
	module→list	Dynamic Data
Adore-ng 0.56	*inode→i_ops*	
	file→f_op	Dynamic Data
	task_struct→{flags, uid, …}	Dynamic Data
	module→list	Dynamic Data
xingyiquan	system call table	Static Data
	module→list	Dynamic Data

内核静态数据和内核动态数据.除此之外,它们的攻击行为和攻击数据是短暂存在的.

Enyelkm 是一个基于 LKM 实现的攻击程序,它可以通过更改系统调用的入口函数来实现隐藏文件、目录和进程的目的.Enyelkm 更改系统调用 getdents 以隐藏目录,更改系统调用 read 从而屏蔽读出文件中的部分内容.除此之外 Enyelkm 还更改了系统调用 kill 来获取根权限.当 Enyelkm 被加载,它使用 LKM 隐藏技术对操作系统隐藏自己的存在.

Adore-ng 是一个基于 LKM 的攻击程序.Adore-ng 攻击的是虚拟文件系统 VFS 中的文件操作的结构从而达到隐藏文件,进程和端口的效果.它也更改 *module→list* 结构隐藏自己.

Xingyiquan 也是一个基于 LKM 的攻击程序,它通过更改系统调用表重定向系统调用,从而可以隐藏进程、文件、目录网络连接和增加后门.

4.2 实验评估

为了评估运行时动态监控内存访问行为对系统开销的影响,我们对比了开启动态策略的内存监控方法和不开启动态策略始终监控内存访问行为方法的性能开销.我们使用 STREAM benchmark[25] 评估性能,STREAM benchmark 是一种常用的开源性能评估工具,通常被用来度量计算机系统内存的带宽.性能评估涵盖 4 组操作:*Copy*,*Scale*,*Add*,*Triad*,我们使用平均开销表示系统的性能开销.*Copy* 函数将数据从一个位置复制到另一个位置.*Scale* 和 *Copy* 相似,只是在写到另一个位置之前将数据乘以 1 个常量.*Add* 函数从内存中一个位置读出数据,和内存中另一个位置读出的数据相加,将结果写到新的内存位置中.*Triad* 函数将 *Scale* 和 *Add* 函数组合起来.我们在被监控的系统中运行 STREAM benchmark.在下面描述的性能评估中,将使用的方法和 Xen 平台上没有部署事件监控的性能做比较.原始没有修改的 Xen 跑出的性能开销为 100%.为了减少性能检测中误差带来的影响,我们对每组实验重复 100 次后取平均值作为性能检测的结果.除了 STREAM benchmark 外,我们还使用另外的 2 个 benchmark 做进一步对比分析.io_zone[26] 是一个文件系统和缓存的性能测试 benchmark 工具,用来测试文件系统的读写性能,包括读、写、再读、再写等操作;ramspeed[27] 是一个缓存和内存的测试工具.

本文的实验评估主要从 4 方面进行:1)对比从开始监控始终对安全关键数据进行监控的方法(StaticMon)和本文提出的动态内存监控方法(DynMon)的性能开销进行对比.由于 StaticMon 不能监控过多的内存动态区域,监控过多动态内存区域会导致性能开销过大而测不到性能开销,在这组对比实验时只监控 1 个安全关键数据.2)为了研究监控安全关键数据大小对 DynMon 的影响,我们使用 1 组实验对比了不同数量的安全关键数据对应的性能开销和内存使用情况.3)为了说明 DynMon 原型系统可以监控系统需要的安全关键数据并不增加过大的性能开销,我们对本文中实现的原型系统中安全关键数据集进行监控的性能开销进行评估.4)我们使用 1 组实验说明本文提出的 DynMon 监控安全关键数据的访问可以帮助检测内核的攻击程序.

4.2.1 DynMon 和 StaticMon 性能对比

由于 StaticMon 监控多个频繁写操作的动态数据页面会导致性能开销过大难以测量,我们在性能开销对比实验中选择 1 个动态安全关键数据区域(进程链表中的最后 1 个进程的 *task_struct*)进行访问行为的监控,仅监控 1 个动态内存页面的访问.图 5~8 中是监控进程列表中最新进程的 *task_struct* 的写访问行为的性能对比,分别对应 32 b 系统和 64 b 系统在不同 benchmark 下的性能对比.可以看出在 32 b 系统和 64 b 系统中,3 个 bench-mark 的结果均显示使用了动态内存监控的方法 DynMon 可以减小监控安全关键数据的系统性能开销,64 b 系统内存监控的开销稍高于 32 b 系统.

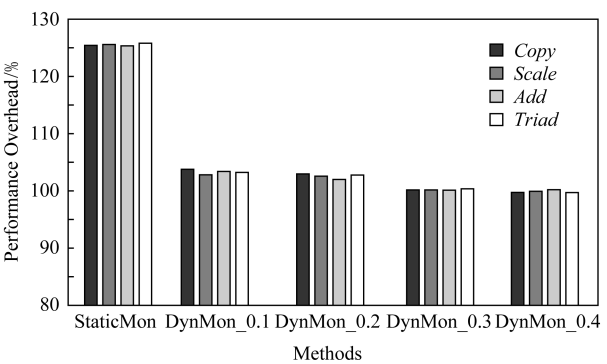

Fig. 5 Performance comparation of 32 b system by STREAM

图 5 32 b 系统性能开销对比图-STREAM

在动态关键数据监控的过程中,系统的额外性能开销主要来源于监控的关键数据所在页面内动态数据被访问导致频繁陷入虚拟机的开销.对比 64 b 系统和 32 b 系统,虽然其虚拟地址大小和寻址空间差距悬殊,但是它们每个页面的大小相同,都使用

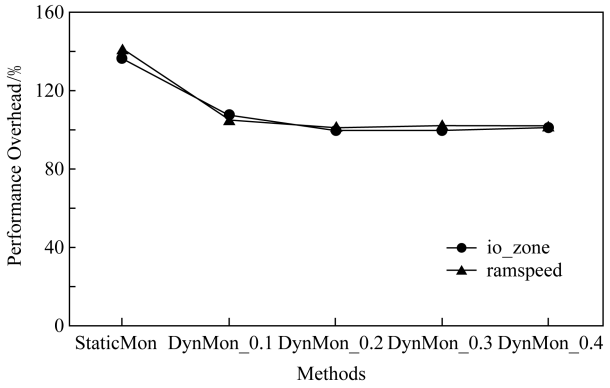

Fig. 6 Performance comparation of 32 b system by io_zone and ramspeed

图 6 32 b 系统性能开销对比图-io_zone-ramspeed

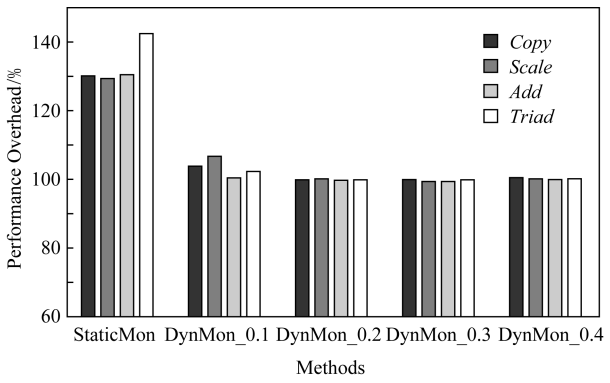

Fig. 7 Performance comparation of 64 b system by STREAM

图 7 64 b 系统性能开销对比图-STREAM

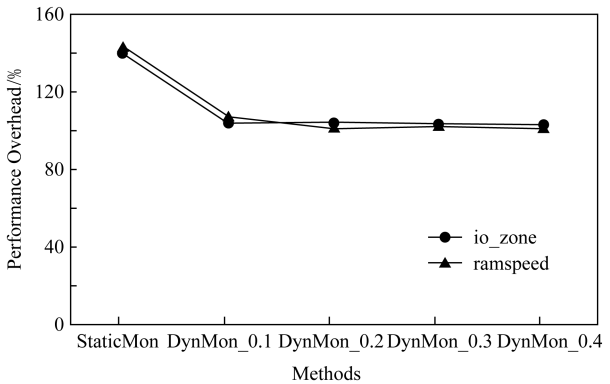

Fig. 8 Performance comparation of 64 b system by io_zone and ramspeed

图 8 64 b 系统性能开销对比图-io_zone-ramspeed

4 KB 大小的页面,对于动态关键内存数据的访问监控,32 b 系统和 64 b 系统的性能开销十分接近.32 b 和 64 b 系统在监控关键数据访问的主要区别在于虚拟地址转换为机器地址的开销,即发生转换检测缓冲区(translation look aside buffer, TLB)缺失时,32 b 系统本身需要 2 级走表,而 64 b 系统需要 4 级走表.虚拟机中由于硬件需要二维走表来转换客户机虚拟地址到机器地址,32 b 系统共需要 12 级走表,而 64 b 系统需要 24 级走表[28],64 b 系统性能开销会稍高于 32 b 系统.根据图 5~8 可以看出无论在 32 b 系统还是 64 b 系统下 DynMon 都可以减小关键内存数据监控的性能开销,下文中为了简洁,我们以 32 b 系统为例分析 DynMon 性能开销下降的原因,不同的监控数据规模对性能开销的影响.

以图 5 中 32 b 系统中在 STREAM benchmark 的测试结果分析,在不增加动态策略的内存访问监控方案 StaticMon 中需要 25.86%的额外性能开销,而本文提出的基于动态监控策略的动态内存监控方法 DynMon 在最小支持度为 0.1 的时序关联规则挖掘得到的监控策略中额外性能开销为 3.63%,对于最小支持度为 0.4 的监控策略下仅需要 0.22%的额外性能开销.动态安全关键区域监控的开销大的原因是由于监控安全关键数据时,安全关键数据相同页面的其他内存访问也会陷入到 VMM 中,导致性能开销增大,表 3 中统计了针对 task_struct 中安全域的监控,每秒陷入 VMM 中的事件由真正的安全关键数据引起的次数和非安全关键数据引起的次数.其中监控页面被写访问的次数为每秒 17 829 次,所有 task_struct 结构体中内容被写访问的次数为每秒 8 865 次,其中真正的安全关键域被写访问的次数为每秒 113 次,如表 3 所示.由安全关键数据访问导致的 VMM 陷入次数仅占该页面数据所有陷入次数的 0.634%.大量和安全关键域无关的动态数据的访问监控导致了频繁陷入 VMM,从而导致系统性能开销增大.本文提出的 DynMon 的主要目的是减小由于安全无关的动态数据被监控引起的 VMM 陷入次数,从而降低不必要的系统性能开销.

Table 3 Statistics of Memory Writing Access

表 3 内存写访问次数统计

Accessed Data	Access Times per Second
Dynamic Data in the Same Page of *task_struct*	17 829
All Members in *task_struct*	8 865
Safety-critical Regions in *task_struct*	113

本文中提出的方法引入了时序关联规则学习监控策略的方法来动态调整需要监控的内存区域,由于不同最小支持度对应不同数量的监控策略,内存

访问行为的监控时机不同.表 4 给出了 StaticMon 和不同最小支持度 DynMon 的 5 组数据对比,包括监控策略数量、陷入 VMM 的频率和漏报率.StaticMon 表示始终对特定的动态区域进行内存访问监控,DynMon_0.1 表示动态监控策略中抽取频繁时序规则时使用的最小支持度为 0.1.监控策略数指的是 3.3 节中描述的方法自动抽取出来的内存区域的动态监控策略的数目,每秒内存访问次数指的是特定动态区域的内存访问截获导致陷入 VMM 的次数,漏报率指的是该安全关键数据访问没有被监控到的次数和实际被访问的次数比.

Table 4 Comparison of StaticMon and DynMon

表 4 StaticMon 和 DynMon 对比

Methods	Number of Monitor Strategies	Number of Memory Access Intercepting per Second	False Negative Rate/%
StaticMon		17 829	0
DynMon_0.1	169	5 600	1.95
DynMon_0.2	37	5 531	2.43
DynMon_0.3	6	1 776	13.38
DynMon_0.4	1	141	52.85

从表 4 的结果可以看出,对于本文提出的 DynMon 方法,最小支持度越低,监控的策略数越多,陷入 VMM 的次数越高,漏报率也越低.这是因为监控的策略数越多,触发动态内存监控的时机也就越多,导致内存访问行为被截获的次数越多,遗漏掉的安全关键数据访问行为越少.结合图 5 可以看出,DynMon 的最小支持度越小,漏报率越低,但是陷入 VMM 的频率越高,系统的性能开销越大.通过表 4 和图 5 的对比,可以看出安全关键数据监控的漏报率和性能开销是呈负相关的关系,而我们希望漏报率和性能开销同时低,需要在漏报率和性能开销之间做出权衡,使得漏报率和性能开销都可以接受.根据该组数据,我们可以得出对该组实验中监控的安全关键数据,基于支持度为 0.1 的时序关联规则挖掘出的动态监控策略比较适合用于安全关键数据的访问监控,在额外性能开销不大(3.63%)的情况下可以截获到绝大多数(漏报率为 1.95%)的安全关键数据访问行为.

综合性能和动态安全关键数据的访问行为的监控效率,我们采用支持度为 0.1 的动态监控策略.当我们采用支持度为 0.1 的时序关联规则挖掘出来的动态监控策略时,和 StaticMon 相比,性能开销减少了 22.23%.对于安全关键数据访问行为的监控漏报率为 1.95%.

4.2.2 安全关键数据数量和性能的开销对比

4.2.1 节为了和 StaticMon 对比,防止 StaticMon 性能开销过大导致系统崩溃,仅开启了最新进程的 *task_struct* 的访问行为的监控.本节中我们以 32 b 系统,最小支持度为 0.1 挖掘出的监控策略,使用 STREAM benchmark 为例,对比不同数量的安全关键数据被监控时的系统性能开销.如图 9 中所示,DynMon_1,DynMon_10,DynMon_50,DynMon_100 分别代表选用 1 个安全关键数据、10 个安全关键数据、50 个安全关键数据和 100 个安全关键数据,在该实验中每个安全关键数据都在不同的页面.根据图 9 的结果可以看出,当被监控的安全关键数据增多时,系统的性能开销也在逐渐变大.

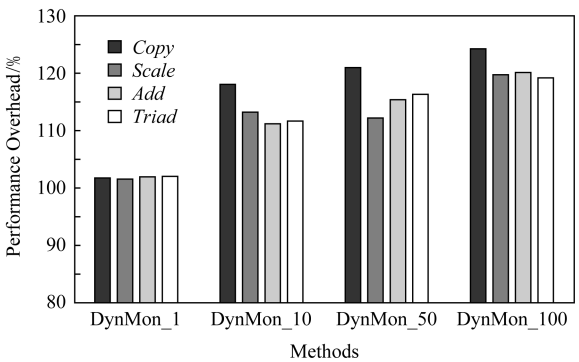

Fig. 9 Performance overhead of DynMon with different data sizes

图 9 不同数量安全关键数据的 DynMon 性能开销图

这是由于这些关键数据所在的页面中的动态数据被访问陷入 VMM 导致的开销,这部分性能开销和被监控的页面数量相关.在本组实验中选用的每个安全关键数据在不用的页面中,即 DynMon_100 这组实验中有分散在 100 个页面的动态数据访问都会陷入 VMM 产生性能开销.当被监控的安全关键数据增多时,DynMon 系统的性能开销也会增加,在监控的安全关键数据涉及到 100 页面时,额外性能开销约为 20%,仍然没有 StaticMon 监控 1 个页面的性能开销大.

本组实验中还比较了不同监控数据集大小,对应的系统内存实际使用大小.如图 10 所示,横坐标表示被监控的安全关键数据的数量,纵坐标表示实际使用的内存大小,单位为 MB.当监控的安全关键数据集大小发生变化时,系统内存的实际使用大小基本没有区别.这是由于本文提出的方法并不会将

安全关键数据重新映射到新的独立页面,不增大对系统内存的使用量.

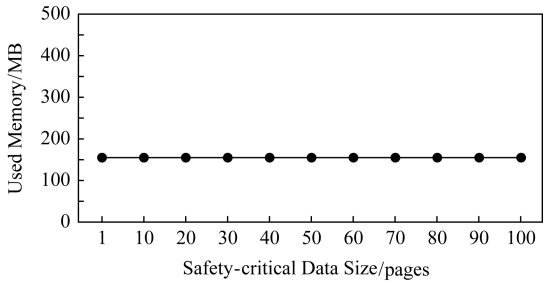

Fig. 10 Used memory of different data sizes

图 10 不同数量安全关键数据的内存实际使用数量

4.2.3 原型系统中 DynMon 的性能开销

在我们的原型系统中,我们使用大家熟知的安全关键数据集合作为监控对象{*av_decision*, *cred*, *dentry*, *file*, *inode*, *kernel_cap_struct*, *policydb*, *posix_acl*, *rlimit*, *socket*, *super_block*, *task_struct*, *module*, *thread_info*, *vfsmount*, *vm_area_struct*},该集合可以根据监控需要进行扩充和调整.我们在 32 b 被监控系统中,对安全关键数据集合所有的安全关键域开启动态监控策略后,使用 STREAM benchmark 测得的性能开销如图 11 所示:

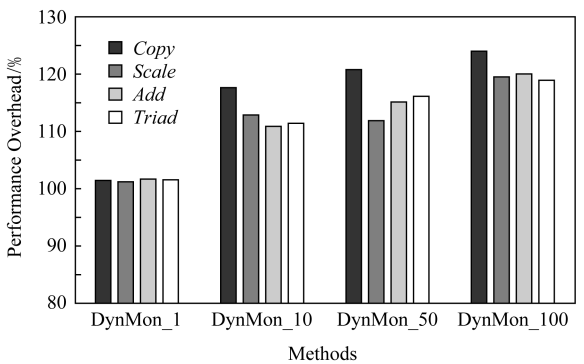

Fig. 11 Performance overhead of DynMon prototype

图 11 DynMon 原型系统性能开销图

本节中对所有进程的 *task_struct* 和其他安全关键数据的访问行为进行监控,监控的内存页面的总数量根据系统运行状态改变.如系统中的进程数量在不断变化,每个进程的 *task_struct* 是分散在内核的动态数据空间的,在本组实验评估性能开销的过程中我们对监控的 *task_struct* 数量进行了统计,在 96～102 的范围内变化,所有被监控的安全关键数据所占的页面个数约为 120 个.在最小支持度为 0.1,0.2,0.3,0.4 的情况下,关键数据监控带来的平均额外性能开销分别为 20.93%,14.11%,7.74%,0.30%.即当需要监控的安全关键数据和页面大量增多时,额外的系统性能开销也相应地增加,但是仍在可以接受的范围内,如果不采用动态监控策略,原型系统中关键数据内存访问监控会因为性能开销过大导致系统崩溃.

4.2.4 攻击的检测效率

DynMon 设计的初衷是因为在内核完整性检测和内存取证等方法[29-30]中,需要通过事件来触发完整性检测,关键内存访问事件是完整性检测中最重要的事件之一,然而由于内存访问监控的开销过大,只能在极少量的关键数据上使用.为了验证本文提出的方法使用到完整性检测中,在内核攻击检测中的有效性,我们分别在 4 种不同的最小支持度下对表 2 中攻击进行检测.根据对表 2 中的攻击程序的被攻击对象分析,可以分为静态代码数据和动态数据两大类,对于静态代码和数据特征都是一致的,即这些不允许被修改;对于动态数据,每种动态数据都有自己的特点,表 5 中列出了 2 种比较典型的常被攻击的 2 种动态数据.对 *task_struct* 中的安全关键数据和 LKM 链表的攻击通常都是短暂攻击,即内核攻击对其篡改后,为了防止被检测程序发现立即将数据状态还原到正常状态以隐藏攻击行为,需要通过实时监控和检测才能检查到该安全数据被恶意攻击.对于对静态代码和数据的攻击,我们的监控策略是始终对静态区域进行写访问监控,可以检测到所有篡改静态代码和数据的攻击.对于动态数据,当截获到安全关键数据的写访问时,触发安全关键数据的完整性检测[29],检测该数据是否被非法的篡改.

Table 5 Rootkits Detection Against 100 Trials of Recurring Attacks

表 5 100 次重复实验检测到的攻击数

Attacked Object	Static Code and Data	*task_struct*	LKM_list
DynMon_0.1	100	100	100
DynMon_0.2	100	99	100
DynMon_0.3	100	98	100
DynMon_0.4	100	9	100

根据表 5 的实验结果我们可以看出,对 *task_struct* 中的安全关键域攻击检测到的概率随着最小支持度的增大而降低,这是因为挖掘监控策略的最小支持度越大,得到的策略越少,结合表 4,内存访问的截获次数越少,越容易漏掉一些安全关键数据访问行为的截获.对于 LKM 链表头指针的篡改,由于其被更改的模式比较固定,都伴随着加载模块和

删除模块的系统调用,对于不同的支持度得到的监控策略差别不大,在本文的实验中,对于不同的最小支持度,都可以检测到所有的 LKM 链表头指针的篡改攻击.因此,对于篡改模式多的安全关键数据如 *task_struct*,当使用序列关联规则挖掘的最小支持度较大时,挖掘出的监控策略不足以覆盖该安全关键数据的,需要选择较小的最小支持度的关联规则挖掘.对于篡改模式比较单一的安全关键数据如 LKM,对最小支持度的要求不高,如实验结果中从 0.1 到 0.4 都可以.

4.2.1 节中的结果已经可以看出,综合性能开销和安全关键数据监控的漏报率,使用最小支持度为 0.1 的序列关联规则挖掘出的动态监控策略最为合适.在本组实验中,结合表 5 的结果可以看出,最小支持度为 0.1 的序列关联规则的动态内存监控方法 DynMon 可以检测到实验中静态数据的攻击和不同类型的动态数据攻击.根据图 11 所示,监控所需要的开销也在可以接受的范围内.

4.3 实验结论

通过 4.2 节实验分析,我们可以看出 StaticMon 监控安全关键数据访问时,大量的安全关键无关数据(99.37%)的内存访问截获产生了巨大的性能开销.为了解决直接监控内存关键数据时的性能问题,本文提出的 DynMon 可以减少不必要的安全无关数据的内存访问截获,使得内存访问监控的性能下降,4.2.1 节实验中 1 个页面的内存访问监控降低了 22.23% 的性能开销.不使用动态监控策略的方法 StaticMon 监控安全关键数据内存访问的性能开销巨大,不能支持本文原型系统中安全关键数据集的内存访问监控,会直接因为性能开销巨大导致系统崩溃.本文提出的 DynMon 可以支持该安全关键数据集的内存访问监控,且额外性能开销在可以接受的范围.因此,DynMon 可以解决监控内存关键数据的访问行为性能开销过大的问题,并且从表 4 中可以看出,可以检测到大多数安全关键数据的访问行为.最后,将本文提出的安全关键数据访问监控的方法应用到内核完整性检测系统中,当检测到安全关键数据访问触发完整性检测,可以检测到实验中提供的所有内核攻击.

5 总 结

本文提出了一种基于动态监控策略的安全关键数据访问的监控方法.通过使用时序关联规则对历史数据进行学习,自动获取安全关键数据的动态监控策略.根据该监控策略和系统运行状态,实时调整需要监控的内存区域,从而解决动态内存区域性能开销大的问题.本文通过在开源虚拟机监控平台 Xen 上搭建原型系统,不需要修改被监控系统的源码和二进制文件,验证了使用动态内存监控的方法可以检测到实验中使用的内核攻击,并通过性能对比实验说明了使用动态内存监控的策略可以减低动态区域内存监控的性能开销.在后续工作中,我们将扩充当前原型系统的安全关键数据集合,并自动学习出更多的监控策略来进一步验证本文提出的方法.考虑利用或者修改硬件特性,在不修改被监控系统的情况下,将安全关键数据进行集中式的存储,以进一步减小动态内存监控的开销.

参 考 文 献

[1] Pham C, Estrada Z, Cao P, et al. Reliability and security monitoring of virtual machines using hardware architectural invariants [C] //Proc of the 44th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks. Piscataway, NJ: IEEE, 2014: 13-24

[2] Li Xun, Huang Hao. Approach of kernel integrity monitoring using hardware virtualization [J]. Computer Science, 2011, 38(12): 68-72 (in Chinese)
(李珣, 黄皓. 一个基于硬件虚拟化的内核完整性监控方法 [J]. 计算机科学, 2011, 38(12): 68-72)

[3] Intel Corporation. Intel Xeon processor E7 V2 family technical overview [OL]. (2014-02-18) [2018-08-08]. https://software.intel.com/en-us/articles/intel-xeon-processor-e7-v2-family-technical-overview

[4] Wang Zhi, Jiang Xuxian, Cui Weidong, et al. Countering kernel rootkits with lightweight hook protection [C] //Proc of the 16th ACM Conf on Computer and Communications Security. New York: ACM, 2009: 545-554

[5] Srivastava A, Giffin J. Efficient protection of kernel data structures via object partitioning [C] //Proc of the 28th Annual Computer Security Applications Conf. New York: ACM, 2012: 429-438

[6] Lu Kai, Zhang Wenzhe, Wang Xiaoping, et al. Flexible page-level memory access monitoring based on virtualization hardware [C] //Proc of the 13th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2017: 201-213

[7] Li Jinku, Wang Zhi, Bletsch T, et al. Comprehensive and efficient protection of kernel control data [J]. IEEE Transactions on Information Forensics and Security, 2011, 6(4): 1404-1417

[8] Maggi F, Matteucci M, Zanero S. Detecting intrusions through system call sequence and argument analysis [J]. IEEE Transactions on Dependable and Secure Computing, 2010, 7(4): 381-395

[9] Kolbitsch C, Milani C P, Kruegel C, et al. Effective and efficient malware detection at the end host [C] //Proc of the 18th Conf on USENIX Security Symp. Berkeley, CA: USENIX Association, 2009: 351-366

[10] Rhee J, Lin Zhiqiang, Xu Dongyan. Characterizing kernel malware behavior with kernel data access patterns [C] //Proc of the 6th ACM Symp on Information, Computer and Communications Security. New York: ACM, 2011: 207-216

[11] Mao Weixuan, Cai Zhongmin, Guan Xiaohong, et al. Centrality metrics of importance in access behaviors and malware detections [C] //Proc of the 30th Annual Computer Security Applications Conf (ACSAC'14). New York: ACM, 2014: 376-385

[12] Xu Zhixing, Ray S, Subramanyan P, et al. Malware detection using machine learning based analysis of virtual memory access patterns [C] //Proc of the 20th Conf on Design, Automation & Test in Europe. Leuven, Belgium: European Design and Automation Association, 2017: 169-174

[13] Pan Shengyi, Morris T, Adhikari U. Developing a hybrid intrusion detection system using data mining for power systems [J]. IEEE Transactions on Smart Grid, 2015, 6(6): 3104-3113

[14] Lee W, Stolfo S. A framework for constructing features and models for intrusion detection systems [J]. ACM Transactions on Information and System Security, 2000, 3(4): 227-261

[15] Srivastava A, Sural S, Majumdar A. Database intrusion detection using weighted sequence mining [J]. Journal of Computers, 2006, 1(4): 8-17

[16] Uhlig R, Neiger G, Rodgers D, et al. Intel virtualization technology [J]. Computer, 2005, 38(5): 48-56

[17] Huang Xiao, Deng Liang, Sun Hao, et al. Secure and efficient kernel monitoring model based on hardware virtualization [J]. Journal of Software, 2016, 27(2): 481-494 (in Chinese)
(黄啸, 邓良, 孙浩, 等. 基于硬件虚拟化的安全高效内核监控模型[J]. 软件学报, 2016, 27(2): 481-494)

[18] Agrawal R, Srikant R. Mining sequential patterns [C] //Proc of the 11th Int Conf on Data Engineering. Piscataway, NJ: IEEE, 1995: 3-14

[19] Srikant R, Agrawal R. Mining sequential patterns: Generalizations and performance improvements [C] //Proc of the 10th Int Conf on Extending Database Technology. Berlin: Springer, 1996: 1-17

[20] Zaki M J. SPADE: An efficient algorithm for mining frequent sequences [J]. Machine Learning 2001, 42(1/2): 31-60

[21] Ayres J, Flannick J, Gehrke J, et al. Sequential pattern mining using a bitmap representation [C] //Proc of the 8th ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining. New York: ACM, 2002: 429-435

[22] Han Jiawei, Pei Jian, Mortazavi-Asl B, et al. FreeSpan: Frequent pattern-projected sequential pattern mining [C] //Proc of the 6th ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining. New York: ACM, 2000: 355-359

[23] Pei Jian, Han Jiawei, Mortazavi-Asl B, et al. Mining sequential patterns by pattern-growth: The prefixspan approach [J]. IEEE Transactions on Knowledge & Data Engineering, 2004, 16(11): 1424-1440

[24] Fumarola F, Lanotte P F, Ceci M, et al. CloFAST: Closed sequential pattern mining using sparse and vertical id-lists [J]. Knowledge and Information Systems, 2016, 48(2): 429-463

[25] McCalpin J D. Stream: Sustainable memory bandwidth and machine balance in current high performance computers [OL]. (2016-07-28)[2018-08-15]. http://www.cs.virginia.edu/stream/

[26] Iozone Project. IOzone filesystem benchmark [OL]. (2016-01-23)[2019-01-01]. http://www.iozone.org/

[27] Cruvolo. RAMspeed/SMP, a cache and memory benchmarking tool [OL]. (2018-07-12)[2019-01-01]. https://github.com/cruvolo/ramspeed-smp

[28] Gandhi J, Basu A, Hill M D, et al. Efficient memory virtualization: Reducing dimensionality of nested page walks [C] //Proc of the 47th Annual IEEE/ACM Int Symp on Microarchitecture (MICRO-47). New York: ACM, 2014: 178-189

[29] Feng Xinyue, Yang Qiusong, Shi Lin, et al. BehaviorKI: Behavior pattern based runtime integrity checking for operating system kernel [C] //Proc of IEEE Int Conf on Software Quality, Reliability and Security. Piscataway, NJ: IEEE, 2018: 13-24

[30] Cui Chaoyuan, Li Yonggang, Wu Yun, et al. A memory forensic method based on hidden event trigger mechanism [J]. Journal of Computer Research and Development, 2018, 55(10): 2278-2290 (in Chinese)
(崔超远, 李勇钢, 乌云, 等. 一种基于隐藏事件触发机制的内存取证方法[J]. 计算机研究与发展, 2018, 55(10): 2278-2290)

Feng Xinyue, born in 1989. PhD. Her main research interests include system security, software engineering and hardware-assistant virtualization.

Yang Qiusong, born in 1977. PhD, professor, PhD supervisor. His main research interests include operating system, software engineering and system security.

Shi Lin, born in 1985. PhD, associate professor. Her main research interests include automated software engineering, requirements engineering, and empirical software engineering.

Wang Qing, born in 1964. PhD, professor, PhD supervisor. Senior member of CCF. Her main research interests include software process, software requirements engineering, empirical software study, etc.

Li Mingshu, born in 1966. PhD, professor, PhD supervisor. Fellow of CCF. His main research interests include operating system, software engineering and distributed system.