软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(12):3172−3191 [doi: 10.13328/j.cnki.jos.004927]

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

基于数据特征的内核恶意软件检测*

陈志锋[1,2], 李清宝[1,2], 张 平[1,2], 丁文博[1,2]

[1](解放军信息工程大学,河南 郑州 450001)

[2](数学工程与先进计算国家重点实验室,河南 郑州 450001)

通讯作者: 陈志锋, E-mail: xiaohouzi06@163.com

摘 要: 内核恶意软件对操作系统的安全造成了严重威胁.现有的内核恶意软件检测方法主要从代码角度出发,无法检测代码复用、代码混淆攻击,且少量检测数据篡改攻击的方法因不变量特征有限导致检测能力受限.针对这些问题,提出了一种基于数据特征的内核恶意软件检测方法,通过分析内核运行过程中内核数据对象的访问过程,构建了内核数据对象访问模型;然后,基于该模型讨论了构建数据特征的过程,采用动态监控和静态分析相结合的方法识别内核数据对象,利用EPT监控内存访问操作构建数据特征;最后讨论了基于数据特征的内核恶意软件检测算法.在此基础上,实现了内核恶意软件检测原型系统 MDS-DCB,并通过实验评测 MDS-DCB 的有效性和性能.实验结果表明:MDS-DCB 能够有效检测内核恶意软件,且性能开销在可接受的范围内.

关键词: 内核恶意软件;数据特征;内核数据对象;恶意软件检测

中图法分类号: TP316

中文引用格式: 陈志锋,李清宝,张平,丁文博.基于数据特征的内核恶意软件检测.软件学报,2016,27(12):3172−3191. http://www.jos.org.cn/1000-9825/4927.htm

英文引用格式: Chen ZF, Li QB, Zhang P, Ding WB. Data characteristics-based kernel malware detection. Ruan Jian Xue Bao/ Journal of Software, 2016,27(12):3172−3191 (in Chinese). http://www.jos.org.cn/1000-9825/4927.htm

Data Characteristics-Based Kernel Malware Detection

CHEN Zhi-Feng[1,2], LI Qing-Bao[1,2], ZHANG Ping[1,2], DING Wen-Bo[1,2]

[1](PLA Information Engineering University, Zhengzhou 450001, China)

[2](State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

Abstract: Kernel malwares are serious threat to the security of operating system. Existing kernel malware detection methods are mainly code view-based, which cannot detect the code reuse and code obfuscation attacks; and a small number of available detection methods for data attacks have limit detection capability due to the limited data invariants. To solve these problems, a kernel malware detection technique based on data characteristics is proposed. First, a kernel data object access model is built by analyzing the kernel object access process during the kernel running. Then, data characteristics building process is discussed based on the model. The process uses dynamic monitoring and static analysis methods to identify the kernel data objects, and employs EPT to monitor the memory access operations to build data characteristics. Finally, the kernel malware detection algorithm based on data characteristics is realized. With this groundwork, a kernel malware detection prototype system MDS-DCB is designed and implemented based on Bitvisor, and the effectiveness and performance overhead of MDS-DCB are evaluated by comprehensive experiments. The results show that MDS-DCB can effectively detect kernel malwares, and the performance penalty induced by MDS-DCB is acceptable.

Key words: kernel malware; data characteristic; kernel data object; malware detection

* 基金项目: “核高基”国家科技重大专项(2013JH00103); 国家高技术研究发展计划(863)(2009AA01Z434)

Foundation item: National Science and Technology Major Project of China (2013JH00103); National High-Tech R&D Program of China (863) (863) (2009AA01Z434)

收稿时间: 2015-01-11; 修改时间: 2015-09-10; 采用时间: 2015-10-09; jos 在线出版时间: 2015-11-27

CNKI 网络优先出版: 2015-11-26 16:06:12, http://www.cnki.net/kcms/detail/11.2560.TP.20151126.1606.004.html

快速发展的计算机和网络给人们带来便利的同时,也带来了很多安全问题.特别是近年来恶意软件数量的日益增长和安全威胁的提升,计算机面临的安全威胁越来越严重.内核(注:如无特别说明,本文的内核均指 Linux 内核)恶意软件是针对内核进行攻击的恶意程序,对计算机系统造成的危害更底层、更彻底,攻击具有隐蔽性、持久性等特点.研究内核恶意软件检测和内核防护技术已成为国内外的研究热点.

现有的内核恶意软件检测方法主要分为基于代码完整性特征和行为特征的检测方法.基于代码完整性特征的检测方法[1,2],通过对内核关键函数、代码等内容进行检测,若发现它们与预期不一致,则发现攻击.为了绕过基于代码完整性的检测方法,攻击者使用了代码复用攻击技术,如 ROP[3,4],JOP[5,6]等,在不改变内核代码完整性的情况下实现恶意攻击.基于行为特征的检测方法[7−10]通过分析指令序列或者系统调用序列特征,判定恶意软件是否存在.为了规避这类方法的检测,攻击者引入了代码混淆技术[11,12],使得基于行为特征检测的方法失效.因此,基于代码完整性和行为特征的检测方法在一定的程度上受到了巨大的挑战.攻击者和检测者围绕着恶意软件的代码属性展开了激烈的竞争.

一般而言,程序是由代码和数据构成的,内核也是如此.随着通过篡改内核代码实现攻击的难度逐步提高,攻击者已将攻击对象扩展到内核数据对象[13−15].而上述的内核恶意软件检测方法主要是考虑了内核代码相关的信息,如代码段、控制流图等,无法检测针对数据篡改的攻击,特别是利用代码复用攻击技术实现的数据篡改.为此,研究人员提出了针对内核数据攻击的检测方法[13,14,16,17].从内核数据的作用来看,内核数据可分为控制数据和非控制数据.在检测篡改内核控制数据的恶意软件上,研究人员对函数指针、返回地址、系统调用表、中断向量表等进行保护和检测[13,16],有效确保了控制数据的完整性.在检测篡改非控制数据的恶意软件方面,文献[14,17]提出了基于数据结构不变量的检测方法,该方法能够有效检测包括篡改非控制数据的 rootkits.但是内核中并不是所有的非控制数据都具备不变量的特征,如果攻击者篡改了这类数据,那么该方法无效.因此,现有针对内核数据对象篡改的检测存在片面性,只能够检测一部分篡改数据的恶意软件,不具备普适性;并且它们无法检测代码复用攻击.LiveDM[18]是一款内存分析系统,能够自动地将内核内存地址转换为数据类型.该系统能够识别内核运行过程中的任意内核对象,可被应用于恶意软件分析和内核调试,但是该系统对所有的内核对象进行分析带来较大的性能开销.

针对上述问题,本文提出了一种基于数据特征的内核恶意软件检测方法.该方法首先通过分析内核数据生命周期内的访问行为构建了内核数据对象访问模型,适用于任意内核软件的数据访问行为描述,通用性好.然后讨论了基于该模型构建数据特征检测内核恶意软件的可行性和存在的难点及其解决方法:第一,提出动态监控和静态分析相结合的内核对象映射方法解决动态内核对象的监控和识别;第二,根据恶意软件攻击对象确定监控对象,提高监控效率和检测效率;第三,基于扩展页表(extended page table,简称 EPT)监控内核对象访问操作,构建内核恶意软件数据特征;第四,提出基于数据特征的内核恶意软件检测算法.在此基础上,设计实现了原型系统 MDS-DCB(malware detection system based on data characteristics).最后,通过实验验证了检测方法的有效性,同时,该方法的运行时性能开销可接受.

本文第 1 节介绍相关工作.第 2 节详细介绍内核数据对象的性质,并基于该性质构建内核数据对象访问模型.第 3 节详细讨论基于数据访问模型的内核恶意软件检测方法的基本思路和过程.第 4 节说明原型系统的基本架构.第 5 节给出实验结果并对本文的方法进行性能分析.最后总结全文.

1 相关工作

本文的主要研究内容是内核恶意软件的数据特征构建及检测方法.为了实现数据特征构建及检测,首先要对内核数据访问行为进行分析;在此基础上,研究内核恶意软件数据特征提取及描述方法,并基于提取及描述的特征,实现基于内核恶意软件数据特征的检测.因此,本文从恶意软件特征描述和内核恶意软件检测这两个方面介绍相关研究工作.

恶意软件的特征描述方法是影响检测能力的重要因素.传统的恶意代码特征大多使用代码特征和序列特征描述法.代码特征主要是指静态二进制特征,其本质上是一段包含汇编指令等信息的二进制字符串.该特征对

于已知的恶意软件具有较好的效果,对于未知的恶意软件无能为力.序列特征描述法主要采用系统调用或者指令序列描述特征.如,Kruege 等人[7]将特征描述为指令序列,Wang 等人[10]用系统调用数目描述特征.序列描述法针对代码或行为的先后次序,易受代码混淆手段的干扰.常用的还有控制流程图(control flow graph,简称CFG)描述法[19],它以代码的执行流程描述特征,但因局限于代码执行顺序,易受顺序无关操作调换等混淆方法的干扰,适合于合法软件的完整性检测.

恶意代码检测方法可分为基于启发式的检测和基于特征的检测两大类[20].

基于启发式的检测方法根据预先设定的规则判断内核恶意软件存在的可能性,如,根据恶意代码的隐藏性特点,XView[21]使用动态交叉视图方法动态分析底层监控器监控的进程信息,与取自系统上层的进程视图进行比较来识别隐藏的进程;又如,vDetector[22]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检测进程、网络连接和文件这 3 种操作系统内部对象的隐藏行为.这类方法的优势在于可检测新恶意代码样本;但其规则的生成依赖于分析人员的经验,在应用中易引发误报及漏报.

基于特征的检测方法根据从恶意代码中提取的特征进行检测,与基于启发式的检测方法相比,具有效率高、误报率低等优点,是目前的主流方法.基于特征的检测方法可分为基于代码完整性特征的检测、基于行为特征的检测、基于语义特征的检测和基于数据结构特征的检测等.

基于代码完整性特征的检测方法[1,2]能够有效检测破坏内核代码完整性的恶意软件,可以有效防止未认证的恶意代码在内核中执行,但是该方法无法检测利用代码复用攻击技术的恶意软件.

基于行为特征的检测方法从恶意软件代码行为出发提取恶意代码控制流信息(如指令序列和系统调用图等).Musavi 等人[8]通过静态分析方法将内核恶意软件的行为分为 5 类,构建了 50 个恶意软件行为特征.PoKeR[9]是一个内核 rootkit 分析软件,通过追踪 rootkit 的指令和 rootkit 攻击下的内存操作,实现 rootkit 行为分析.但是 PoKeR 只对内核 rootkit 操作的内核对象进行分析,无法检测利用合法代码操作内存对象的 rootkit. NumChecker[10]利用 HPCs(hardware performance counters)检测操作系统调用执行是否与预期一致,判断是否存在篡改系统控制流的恶意软件.该类方法的不足在于易受代码混淆技术的干扰,导致检测结果不准确.

基于语义特征的检测从语义角度抽象特征,如,Kruegel 等人[7]通过静态分析内核恶意软件构建非法的内存访问行为模型,基于非法内存访问模型使用符号执行构造特征来进行检测,可阻止恶意模块加载到内存中.文献[23]提出了检测违背内核动态数据语义完整性的方法来检测内核是否遭受篡改.文献[24]提出了一种基于信道的数据交互协同模型,然后,基于该模型识别不同恶意软件的行为语义.这类方法在一定程度上可抵御代码混淆技术,但是语义规则生成依赖分析人员的经验.

基于数据结构特征的检测通过分析恶意软件内存空间内使用的数据结构或者数据结构不变量属性判定恶意软件的存在性.Laika[25]通过分析数据结构使用情况检测恶意软件,但它只适用于拥有独立内存空间的用户层恶意软件的检测,不适用于内核恶意软件的检测,因为内核恶意软件和内核共用内存空间,而且内核恶意软件使用内核中的代码或数据而很少使用自己的数据,使得分析内核恶意软件的数据结构特征很难.文献[14,17]通过静态分析确定内核中数据结构存在的不变量关系,然后在运行过程中扫描内核内存检测是否违背数据结构不变量关系判定恶意软件的存在性.但并不是所有的内核数据都具备不变量关系,一旦攻击者改变了这类数据,这种方法无法检测出此类攻击.

2 内核数据对象访问模型

2.1 内核数据对象访问过程分析

内核数据对象包括内核静态数据对象和内核动态数据对象.内核静态数据对象是指在内核编译时就确定了分配地址的数据或数据结构,在内核运行过程中常驻内存,简称为静态数据.内核动态数据对象是指在内核运行过程中动态分配和释放的数据或数据结构,简称为动态数据.通过观察发现:内核数据对象在它们的生命周期内具备一定的使用规则,其操作顺序均是申请内存空间→内存操作→释放内存空间.但是由于静态数据在编译期就确定了内存地址,在系统开启过程中根据预先约定好的位置将数据值存放到内存中,故静态数据的申请内

存空间动作视为空;动态数据则需要申请内存空间,内存分配函数根据动态数据提出的申请从内存中分配相应的内存.然后,这些数据的内存值在生命周期过程中不断地被读取或者改写.最后,它们在使用结束后被销毁,静态数据对象一般在关机时结束生命周期,动态数据对象则是在内存释放函数被调用时结束生命周期.

图 1 给出了动态数据的生命周期,包括创建-访问(读写)-释放.代码 C_k 调用内存分配函数 *kmalloc* 申请了 d_1 到 d_5 大小的内存空间,该内存区域的属性被设置为可读写.代码 C_r 从地址 d_2 处读取数据值,代码 C_w 将值写到地址 d_1 处.在数据对象的生命周期结束时,代码 C_f 调用释放函数 *kfree* 释放这块内存区域.

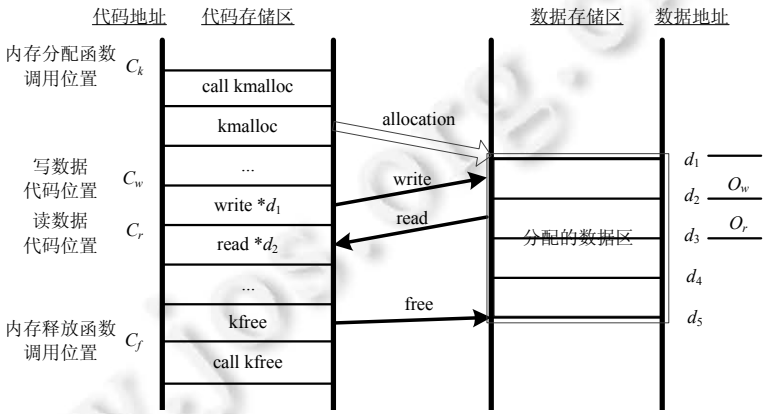

Fig.1 Lifetime of a dynamic data object

图 1 动态分配的数据对象的生命周期

下面我们简要分析与生命周期内数据访问相关的要素.

1) 内存分配和释放

由于静态数据采用的是预分配机制,故内存分配和释放事件只与动态数据相关.当动态数据申请内存空间时,内存分配事件调用内核内存分配函数将内存空间分配给动态数据,内核动态对象生命周期于此刻开始.我们称调用内存分配函数的代码位置为分配位置.当内核动态对象不再需要使用时,内存释放事件调用内存释放函数解除动态对象占用的内存空间.至此,动态对象的生命周期结束.类似地,我们称调用内存释放函数的代码位置为释放位置.从图 1 中我们知道,内存分配和释放事件分别与 allocation 和 free 函数对应.

2) 内存访问

在不考虑 DMA 操作的情况下,数据的读写操作是与 CPU 的访存操作相对应的.在访存过程中,CPU 从内存中取相应的数据进行计算,计算之后,将计算值写回内存中.因此,在内核运行期间,访问内核数据的代码集合描绘了内核数据的使用情况和使用过程.我们称读取内存值的代码位置为读位置.类似地,写内存的代码位置称为写位置.

3) 数据对象标识

由于内存中存在着大量的静态数据和动态数据,有效区分这些数据才能保证数据特征的有效性.为此,本文引入数据对象标识用于区分不同的数据对象.对于静态数据,它们的数据类型和地址(地址范围)是静态确定的,通过符号表中可获得这些信息.通过赋予它们一个特定的编号,我们可以唯一地确定它们的数据类型和地址范围,一般选择地址作为唯一编号.因此,我们称这个唯一的编号是静态数据对象的标识.而对于动态数据对象,由于大量的动态对象在内核运行过程中动态变化,并且内存分配函数在分配完内存空间之后一般返回的是分配内存空间地址而未提供动态数据的类型信息,所以动态数据对象标识是一个极为复杂的工作.

对于同一内核不同的运行时机,同一个动态数据可能分配到不同的内存中,同一个内存位置不同时候的数据时动态变化的,并且在内存中存在着同一种数据类型的动态数据.针对该问题,本文使用分配函数调用位置作为动态数据对象的分类标识.因为一个分配函数调用位置能够确定一个动态数据并被唯一地转化为动态数据

的数据类型,这样就能够区分不同时机不同类型的动态数据.图 2 给出了调用位置和动态数据类型之间的相互关系,在头文件 *define.h* 中定义了数据结构 *M*,在两个文件中分配了两个该数据结构对应的数据 *m*1 和 *m*2.尽管两个数据的数据类型是一样的,但是它们的使用过程不一样,相互独立.因此,本文将分配调用位置及其数据类型作为动态数据的唯一对象标识,可以有效区分内存中的动态数据.

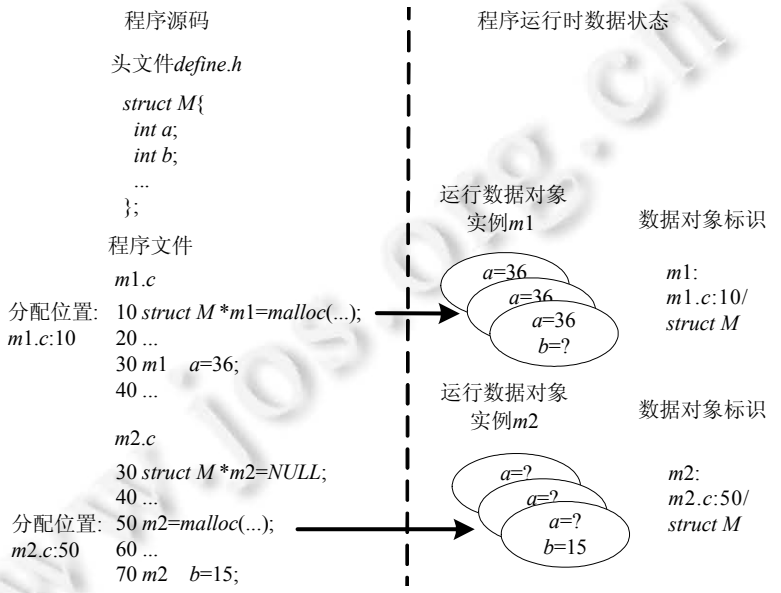

Fig.2 Relation of allocation sites and data identifications

图 2 分配位置与数据标识的关系

4) 数据字段

数据可能以数据结构形式、数组形式或者单一数据形式存在.对于数据结构形式的数据,一般包含有多个数据字段,不同的字段定义不同,偏移值不同;对于数组形式的数据,其包含多项同一类型的数据,每项的偏移值不同;对于单一数据形式的数据,其偏移值恒为 0.图 2 所示的数据结构 *M* 包含字段 *a*,*b* 等,其中字段 *a* 的偏移值为 0,字段 *b* 的偏移值则为 4,其他字段依次类推.攻击者一般对数据结构中的某一字段或者多个字段进行篡改,那么在监控器监视到数据篡改时,偏移值能够快速有效地确定数据结构的数据类型并获得数据字段的相关信息.譬如,攻击者对数据 *m*2 的字段 *b* 进行篡改,那么由 *m*2 的数据对象标识可以得到其数据类型为 *M*,再根据字段偏移值便可知道字段 *b* 的具体语义和取值情况.

2.2 内核数据对象访问模型

根据第 2.1 节的分析,我们构建了内核数据对象访问模型(kernel data object access model,简称 KDAM).该模型描述了内核数据在生命周期内的使用情况.首先给出一次内核数据访问的基本定义.

定义 1. SKDAP(a single kernel data access pattern,内核数据访问模式)是一个 5 元式(*c*,*o*,*t*,*i*,*of*),其中,

- *c*:访问内核数据的代码;
- *o*:访问内核数据的操作,包括读或写,*o*=1 表示写操作,*o*=0 表示读操作;
- *t*:数据对象类型,包括静态数据或动态数据,*t*=1 表示动态数据,*t*=0 表示静态数据;
- *i*:数据对象标识,当 *t*=0 时,*i* 是基于编译阶段的符号信息赋予的编号;当 *t*=1 时,*i* 是分配该数据的分配位置和数据类型;
- *of*:被访问的数据对象字段的偏移值.

根据定义 1,SKDAP 给出了访问一个内核数据的过程.操作系统内核中包含有成百上千种数据类型,这些数据类型又定义了成千上万个内核数据.因此,所有的 SKDAP 集合描述了内核一次运行过程的数据访问情况,详见定义 2.

定义 2. 内核一次运行实例的数据访问可由序列 $D_m=\{SKDAP_i,SKDAP_{i+1},\ldots,SKDAP_j\}(j>i,i\geqslant 0)$定义,其中,$m$表示内核的某一次运行.

由定义 2 可知:D_m给出了内核从开机到关机这一过程的所有内核数据访问模式,描述了内核的某一次运行过程中所有内核数据在生命周期内的使用情况.故,可以认为 D_m是 KDAM 模型的具体表现.

3 基于内核数据对象访问模型的内核恶意代码检测方法

3.1 方法概述及问题的提出

图 3 给出了内核对象访问过程.其中,圆角矩阵是一个动态内核对象数据类型 *task_struct*,由代码 C_1 为其分配内存空间.该图左虚框给出了不含恶意软件的内核(良性内核)运行过程中,不同代码访问动态内核对象的过程.代码 C_2 将 *task_struct* 结构插入到以 *tasks* 字段链接的链表中,对 *task_struct* 的 *tasks* 字段进行了修改;代码 C_3 则是给新创建的进程分配 *pid*;代码 C_4 读取某个进程的 *pid* 号.因此,这一个过程中发生了若干次内核对象访问,根据定义 2,它们的 SKDAP 分别为

$$(C_2,1,1,C_1/task\_struct,328),(C_3,1,1,C_1/task\_struct,444),(C_4,0,1,C_1/task\_struct,444).$$

上述 SKDAP 是在内核不包含恶意软件的情况下构建的.如果在这一过程中有恶意程序运行,那么内核数据的访问过程将会发生变化.如图 3 右侧虚框所示,两款恶意程序对内核数据进行了篡改.C_5和 C_6修改了该对象的 *tasks* 字段,C_5表示 rootkit sk 1.3b 的代码,C_6为内核中的合法代码,它被 rootkit adore 0.38 利用实现恶意攻击,基于代码完整性的检测方法无法发现此类攻击.因此,扩展后的 SKDAP 为

$$(C_5,1,1,C_1/task\_struct,328),(C_6,1,1,C_1/task\_struct,328).$$

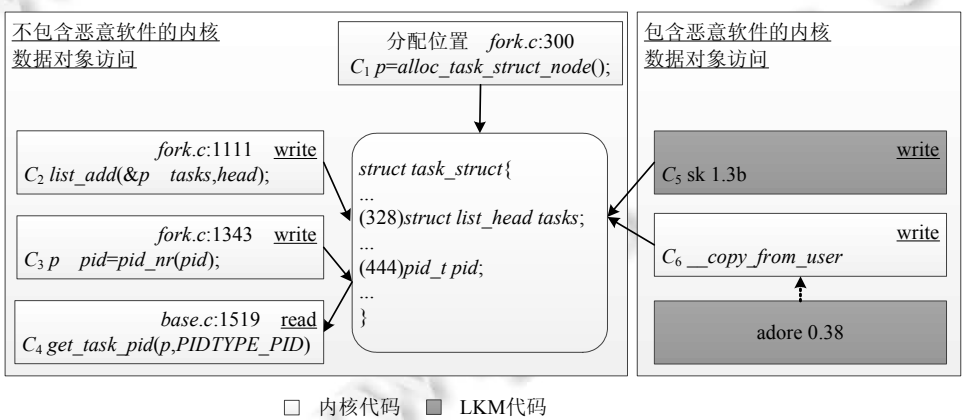

Fig.3 An access process of a kernel data object

图 3 某一个内核数据对象的访问过程

从该例子可以看出,恶意软件的引入使得内核数据访问过程发生变化.因此,可通过分析良性内核和含恶意软件的内核的数据访问模式差异,构建恶意软件数据特征,并基于该特征便可发现内核恶意软件的存在性.

根据上述分析,基于该模型实现恶意软件检测的核心是构建 SKDAP 和数据特征.实现这一核心功能需要解决的问题包括:

(1) 内核动态对象分配行为的捕获;

(2) 内核动态对象的类型识别;

(3) 内核对象访问行为和访问主体的捕获;

(4) SKDAP 的分析和数据特征的构建.

由于 SKDAP 中的动态数据对象标识是分配代码位置和数据类型,在运行过程中无法直接得到该内容,故问题(1)和问题(2)采用动态监控和静态分析相结合的内核对象映射方法解决.问题(3)主要是 SKDAP 中访问代码 *C* 和访问行为 *O* 的确定问题,解决方法是动态监控关键对象的内存区域的访问操作,并通过静态分析获取访问代码 *C*.问题(4)是如何从 SKDA 集合中分析得到恶意软件的数据特征,同时如何减少误判和漏判问题.针对问题(4),采用多次运行采集和集合论思想构建恶意软件的数据特征.

3.2 内核对象识别

对于静态内核对象,通过系统符号表可以确定其对应的数据类型、名称、地址等;而对于动态内核对象,无法直接获取其内存位置和数据类型等信息.为此,提出了一种面向动态数据分配监控的内存映射方法,在内核运行过程中监控内核动态数据分配事件,逆向推导动态内核数据的类型,并且在内存数据释放事件发生时更新内核对象映射关系.

面向动态数据分配监控的内存映射方法通过捕获内核对象分配和释放事件生成一个内核对象的同步映射.尽管监控内核对象分配事件能够捕获到内核对象的创建和销毁行为,但是由于内核内存分配函数未提供获取分配对象数据类型的方法.对于该问题,一种较为直接的方法是通过静态分析重写内核代码,使得在内存分配函数被调用时能够将分配对象的数据类型传递给监控器.这种方法需要修改并重新编译内核,不适用于已发布的操作系统.因此,本文提出了一种基于上下文信息的动态数据类型推导方法.该方法根据监控的内存分配函数信息确定分配调用位置,并通过静态分析将调用位置转换为数据类型,不需要修改操作系统内核,适用性广.其基本过程如下:

(1) 分析内核内存分配机制,确定需要被监控的内核内存分配函数;

(2) 监控内核内存分配事件,获取分配的内存地址范围和调用内核分配函数的代码位置.代码位置即上文提到的分配函数调用位置,它是运行时分配的动态对象的唯一标识;

(3) 结合分配函数调用位置和源码静态分析确定被分配的动态内核数据的类型.

3.2.1 确定被监控的内核内存分配函数

在内核源码中,有很多封装函数负责内核内存管理,其中一些被定义为宏函数或者内联函数,其他的封装函数被定义为普通函数.由于宏函数和内联函数在编译预处理阶段由预处理器将其替换为核心内存分配函数,故捕获它们调用位置的方式与核心函数一致;而对于普通封装函数,它们的调用位置属于封装函数代码.

通过对内核源码的分析,除了核心分配函数,再捕获内核封装函数的内存分配事件,就能够覆盖内核运行过程中的大多数内存分配事件.

因此,根据分配函数的功能,本文将需要监控的内核内存分配函数分为 4 类:(1) 页分配释放函数 *__get_free_pages*/*__free_pages*,以页为单位分配内存;(2) *kmem_cache_alloc*/*kmem_cache_free* 函数,适用于反复分配释放同一大小内存块;(3) *kmalloc*/*kfree* 函数,用于分配范围在 16 字节~128KB 大小以内的小内存区域,并且此函数分配的内存在线性地址和物理地址上都是连续的;(4) *vmalloc*/*vfree* 函数,*vmalloc* 分配的内存只是在线性地址上是连续的,它不保证分配的内存在物理上也连续;*vmalloc* 的主要目的是用于非连续物理内存分配.具体说明见表 1.

Table 1 Analysis of kernel allocation functions

表 1 内核分配函数分析

函数代表	分配原理	最大内存	其他
__get_free_pages	直接对页框进行操作	4MB	适用于分配较大量的连续物理内存
kmem_cache_alloc	基于 slab 机制实现	128KB	适合需要频繁申请释放相同大小内存块时使用
kmalloc	基于 kmem_cache_alloc 实现	128KB	最常见的分配方式,需要小于页框大小的内存时可以使用
vmalloc	建立非连续物理内存到虚拟地址的映射		物理不连续,适合需要大内存,但是对地址连续性没有要求的场合

3.2.2 运行时内核对象映射生成

在内核的运行过程中,监控器通过在分配释放函数首地址和返回处插入 VMCALL 指令(0f 01 c1)截获所有的内核内存分配、释放和返回事件,具体见文献[26]提出的资源函数截获方法.这一过程需要确定 3 件事:(1) 分配函数调用位置;(2) 被分配或释放的动态对象的地址;(3) 分配对象的大小.确定分配函数调用位置是为了确定分配对象的数据类型;确定被分配和释放的对象的地址和大小是为了确定监控的内存空间.

1. 确定分配函数调用位置

前文已指出,分配函数调用位置是指被调用的分配函数在源码中的位置(源码文件中的行号).实现这一功能的步骤如下:

① 获取分配函数调用的返回地址,即,下一条要执行的指令的地址;

② 通过调试符号获取该地址对应的源码位置.

根据程序指令流的执行过程,返回地址是call指令后下一条指令的地址.根据返回地址可以得到call指令的地址,如下所示:

```
__kmalloc:
ffffffff8115f710      push %rbp
ffffffff8115f711      mov %rsp,%rbp
ffffffff8115f714      push %r15
ffffffff8115f716      push %r14
…
ffffffff8115f7ce      retq
…
ffffffff8115fce0      mov %rsi,-0x40(%rbp)
ffffffff8115fce4      mov $0x80d0,%esi
ffffffff8115fce9      mov %rdx,-0x38(%rbp)
ffffffff8115fce0      shl $0x4,%rdi
ffffffff8115fcf1      callq ffffffff8115f710 〈__kmalloc〉
ffffffff8115fcf6      test %rax,%rax
…
```

执行 call__kmalloc 指令时,将下一条指令 test 的地址 ffffffff8115fcf6 存入堆栈中.当__kmalloc 执行完,调用retq,将地址赋给指令寄存器 RIP.因此,通过 RIP 的值可以推导得到 callq 指令的地址,即,调用__kmalloc 的指令地址.

在调试符号中,记录了每一行源代码对应的地址.由于一行代码可能对应多条汇编指令,所以分析过程中还需要考虑该代码行的起始地址与分析的汇编指令间的偏移量,偏移以字节为单位.通过地址与偏移量得到该行代码的首条汇编指令地址,然后以该地址查询调试符号获取该行代码的位置,即,该代码在文件中的具体行号.最后,将计算得到的调用位置存储在内核对象映射中,通过类型推导算法(详见第 3.2.3 节)确定分配对象的类型,具体过程如图 4 所示.

2. 确定分配对象的地址和大小

被分配或释放对象的地址和大小可以通过参数和返回值推导得到.对于分配函数,对象大小通常是其参数之一,内存地址则是作为返回值;对于释放函数,对象内存地址作为函数参数.函数参数是通过栈或者寄存器传递的,这些内容可以通过监控内存分配或释放函数调用的入口位置来捕获.然后,根据函数调用归约确定对应的参数值.返回值的确定相对复杂,我们需要确定返回值的存储位置和存储时机.64 位的整数以及 64 位的指针是通过 RAX 寄存器传递的,本文需要捕获的所有值都是这种类型.当分配函数返回到调用者时,在 RAX 寄存器中的返回值是有效的.为了在正确的时间捕获返回值,我们在分配函数返回处插入 VMCALL 指令.此时,从客户机

陷入VMM中,通过读取VMM保存的状态中的RAX寄存器的值即为返回值.若返回值是非指针型数据,则RAX中的值直接为具体值;否则返回值为地址,该地址中存储的数据为具体值.对于分配函数,返回值一般为地址,地址为分配的内存空间的首地址.

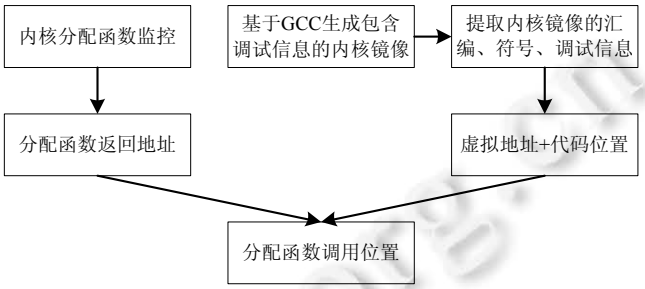

Fig.4 Analysis of allocation function locations

图 4 分配函数位置分析

3.2.3 推导动态数据的类型

我们在第3.2.2节中确定了分配函数的位置,该分配函数负责内存空间分配,但是其未给出分配对象的数据类型,故本节讨论如何基于分配位置推导分配对象的数据类型.图 5 为分配函数给进程结构体分配内存空间的示例程序.首先,动态数据对象 *p* 的分配函数调用位置 *C* 被映射到源代码 *fork.c*:1586 中,该代码将分配对象的内存地址赋给赋值语句 *A* 左边的指针变量.由于该变量的类型可以表示分配的内存类型,故确定该变量的类型即确定了动态数据对象的类型.因此接着通过遍历指针变量的声明 *D* 和类型的定义 *T* 可得到动态数据 *p* 的类型为 *task_struct*.为了获取 *C*,*A*,*D*,*T* 等的代码元素内容,我们参照了文献[27]使用的编译器插桩分析法.

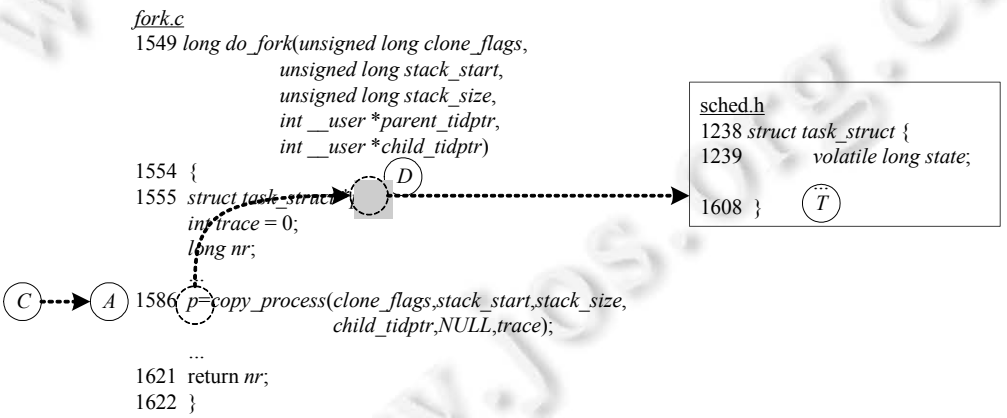

Fig.5 Analysis process of data type determination through the static analysis

图 5 静态分析确定数据类型过程示意图

图5是动态数据类型分析过程的一种类型.文献[18]给出了数据类型推导过程的抽象表示.本文通过对内核源代码中分配函数被调用的代码进行分析可知推导内核数据数据类型的过程与文献[18]一致,分为 3 种情况,如图 6 所示(注:*C* 为调用位置,*A* 为赋值语句,*D* 为变量声明,*T* 为类型定义,*R* 为返回语句,*F* 为函数声明).

图6(b)的推导过程与图6(a)类似,只是将变量的声明和赋值放在同一行代码中实现.图6(c)与前两种相差较大,它没有使用变量存储分配对象的内存地址,而是直接将内存地址返回给分配函数.

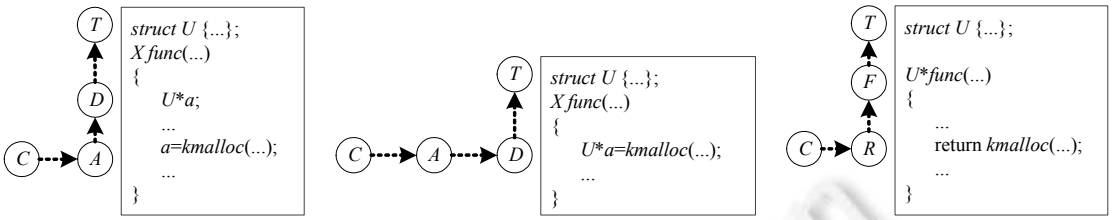

Fig.6 Abstract representation of data type inference[18]

图 6 数据类型推导过程抽象表示[18]

数据类型推导算法流程如下:

(1) 根据调用位置 *C* 获得对应位置的源码语句 *S*;

(2) 判断语句 *S* 的类型:若是赋值语句,转步骤(3);否则,步骤转(6);

(3) 解析赋值语句:若语句左边为单一变量,则转步骤(4);若语句左边包含变量声明,则转步骤(5);

(4) 向上搜索变量的声明语句,根据声明获取变量的数据类型定义;

(5) 根据变量声明获取变量的数据类型定义;

(6) 向上查看函数声明获取函数返回类型确定数据类型.

3.3 内核对象访问监控

监控内存分配函数获取分配对象的内存地址范围,然后在内核运行过程中监控针对已分配内存的内核对象的读写操作,并与内核对象映射内容相结合,即可构建 SKDAP.

由于内核中的对象种类繁多,对所有的对象都进行访问监控带来的开销太大.为了在保证方法有效性的同时确保方法的实用性,本文对需要监控的内核对象类型进行了裁剪和归类.裁剪依据:只保留攻击者常利用的对象以及对内核安全功能影响较大的对象.最终根据对象类型确定了 5 类被监控对象.

1. 进程类对象,包括 *task_struct*,*sighand_struct*,*thread_struct*,*threadinfo*,*signal_struct*,*sched_entity*,*pid*,*pid_namespace* 等;
2. 文件类对象,包括 *file*,*files_struct*,*fs_struct*,*fdtable*,*dentry*,*super_block*,*inode*,*vfsmount*,*proc_inode*,*buffer_head*,*elf*32_*hdr* 等;
3. 网络类对象,包括 *sock*,*socket*,*inet_bind_bucket*,*sock_alloc*,*sock_info*,*socket_alloc*,*dst_entry*,*dn_fib_node* 等;
4. 内存类对象,包括 *mm_struct*,pgd_*t*,*pte_t*,*vm_area_struct*,*vmap_area*,*vm_region*,*vm_fault*,*anon_vma_chain* 等;
5. 模块类对象,包括 *module*,*module_kobject*,*load_info* 等.

在监控到内核分配函数被调用时,根据内核对象映射得到被分配的对象类型,首先判断该对象是否属于以上 5 类被监控对象.

- 若是,对于被监控对象内存区域所跨越的每个页,在 EPT 页表中清除其 PTE(page table entry),使得以后对这些页的访问会产生 EPT violation.由于修改了 GPA 到 HPA 的映射关系,所以还要清除 EPT TLB,使得原来的映射失效;
- 否则,不做额外处理.

每当对这些被监控内存空间发生访问请求时,就会陷入到 VMM 中,VMM 收集访问信息,包括访问对象、访问类型、访问者等.

写访问请求处理过程如图 7 所示,读访问请求处理过程类似.

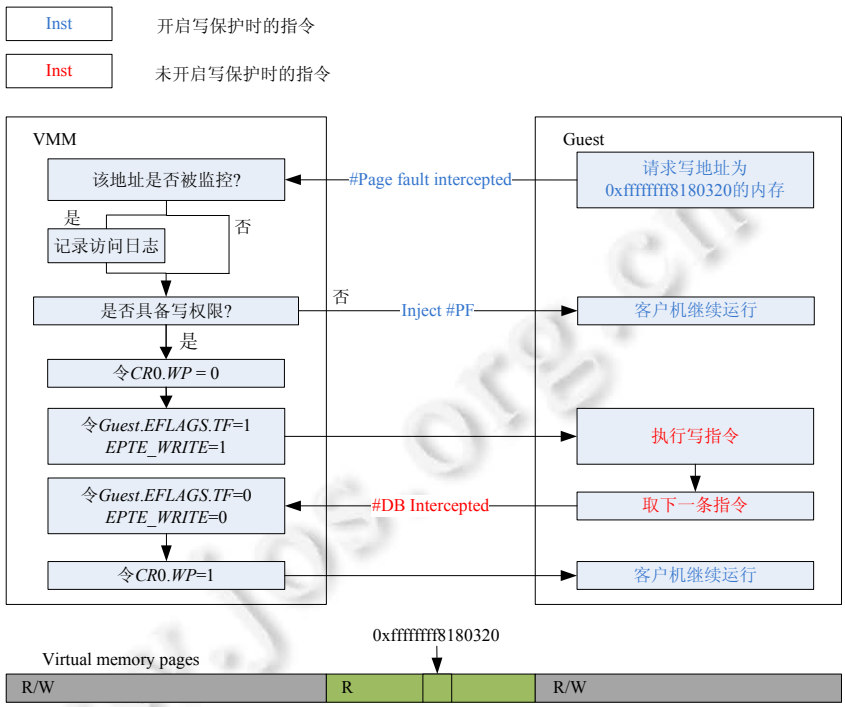

Fig.7 Dealing process of write operation based on EPT

图 7 基于 EPT 的内存写操作处理过程

3.4 内核恶意软件特征构建

由于内核数据在内核运行过程中动态变化,构建良好内核数据特征的难度等同于构建内核执行路径的全集.因此,本文选择了构建恶意软件的数据特征作为恶意软件检测的基准,极大地降低了特征构建的难度.

为了有效、可靠地构建内核恶意软件在动态执行过程中的数据特征,本文采用了多次采集去重法.假设内核恶意软件 M 在第 i 次运行中对应的 KDAM 是 $D_{M,i}$,不含恶意软件的内核在第 j 次运行中对应的 KDAM 为 $D_{B,j}$,那么对 n 次恶意软件运行和 m 次良性内核运行收集的 KDMA 应用集合操作,得到内核恶意软件 M 的数据特征为

$$S_M = \bigcap_{i\in[1,n]} D_{M,i} - \bigcup_{j\in[1,m]} D_{B,j} \tag{1}$$

其中,S_M 表示在 n 次内核恶意软件运行中均出现、但在 m 次良好内核运行中从未出现的 SKDAP 集合.

内核恶意软件一般以可加载内核模块或者驱动形式存在,它们每一次加载到内存中的位置均不一致,并且未知的恶意软件一般无法直接得到其源码,无法获得模型中的访问内存的代码位置.此外,一旦攻击者修改了有源码的恶意软件,那么已分析的特征将失效.例如,我们根据特征构造方法已经生成了某一个恶意软件的特征,但是攻击者修改了恶意代码,增加一行或者加一些不影响恶意软件功能的代码,那么应用特征构造方法获取修改过的恶意软件将与前者构造的不一致,因此便无法识别出该恶意软件.

针对该问题,本文没有使用恶意软件自身的特定标识,而是将恶意软件访问内核对象的代码位置用一个统一的标识标记,设为 η.由于内核支持可扩展,除了恶意模块可使用 LKM 外,合法的内核模块也是使用该机制加载到内存中,那么合法内核模块的数据特征也使用了标识 η,而它们是合法的数据访问,故需要将它们从数据特征中剔除.

如果恶意软件运行过程中没有使用自身的代码执行恶意功能,而是使用内核自身的代码实现,如图 4 所示的 adore rootkit,那么该 SKDAP 就不会出现在 D_B 中,但是需要将其作为恶意软件数据特征的组成之一,才能与公

式(1)定义保持吻合.

下面我们以 adore 0.38 为例,说明如何按照公式(1)构造恶意软件的数据特征.根据公式(1),假设 m=10,n=4,在相同环境下收集 10 次良性内核运行过程对应的 KDMA,分别为 $D_{B,1},\ldots,D_{B,9},D_{B,10}$,同样环境下运行 4 次 adore 0.38,每次加载该 rootkit 后收集对应的 KDAM,分别为 $D_{adore0.38,1},D_{adore0.38,2},D_{adore0.38,3},D_{adore0.38,4}$.然后,按照公式(1)进行运算,最终 adore 0.38 的数据特征为

(η,1,0,0xffffffff8180320,{624,1736,…,8}),(η,1,1,*fork.c*:300/*task_struct*,{692,668,…,720}),…,

(*uaccess_x*64.*h*:55,1,1,*slab_def.h*:130/*linux_dirent*64,720),….

对应的具体数据详见表 3 和表 4.实验中的数据为更好展示每一项 SKDAP,将同一个访问对象的不同偏移值信息分别列出.

3.5 恶意特征匹配

判断内核恶意软件 M 在内核运行过程中存在的可能性,是根据恶意软件 M 的特征 S_M 是否属于运行过程中构建的数据访问行为集合 D 的性质决定的.

由于 SKDAP 中的元素 *of* 有两种表示形式:一种是只有一个偏移值,另一种是一些偏移值的集合,所以不能够纯粹的比较 S_M 和 D 中各元素.本文给出了基于特征匹配的恶意软件检测算法 JudgeMalware,该算法首先构建 S_M 和 D 的交集 I,然后再对比 I 和 S_M 的差异性判断恶意软件的存在性.算法见表 2.

Table 2 Kernel malware detection algorithm

表 2 内核恶意软件检测算法

```
function JudgeMalware(S_M,D)
  I←CheckCharacteristic(S_M,D)
  if number of I equal to number of S_M then
    For each e in I do
      if e∈S_M then
        continue
      else
        report no Malware
      end if
    report exist Malware
  end if
end function
function CheckCharacteristic(S_M,D)
  I←∅;
  for each e in S_M do
    for each e′ in D do
      if CompareElements(e,e′)=1 then
        I←I∪{e}
      end if
    end for
  end for
  return I
end function
```

```
function CompareElements(e,e′)
  if e.c≠e′.c∨e.o≠e′.o∨e.t≠e′.t∨e.i≠e′.i then
    return 0
  end if
  if e.of is an offset then
    if e′.of is an offset then
      if e.of=e′.of then
        return 1
      end if
    else
      if e.of∈e′.of then
        return 1
      end if
    end if
  else
    if e.of is a range of offset then
      if e.of⊂e′.of then
        return 1
      end if
    end if
  end if
end function
```

根据算法可知:判断 S_M 中的元素和 D 中元素是否相同,是由 *CheckCharacteristic* 函数和 *CompareElements* 函数完成的.判断过程分为两步:首先是各元素的字段 *c*,*o*,*t*,*i* 完全一致,然后在一致的基础上根据偏移的类型分类比较,若元素的偏移字段满足相等或者属于关系,那么就认为这两个元素相同.最后,*JudgeMaleware* 函数通过比较 S_M 和 *CheckCharacteristic* 函数的结果是否一致,判定恶意软件的存在性.

算法中,分析 SKDAP 的第 5 个字段偏移值 *of* 需要分情况对待,这里我们仍以分析 adore 0.38 为例来说明特征匹配过程.上面已给出 adore 的数据特征,现在假设收集到 adore 某一次加载时的数据特征的某一项(η,1,0,0xffffffff8180320,{624,1736}),那么匹配过程为:

对 adore 数据特征中的第 1 项(η,1,0,0xffffffff8180320,{624,1736,…,8}),首先,对于前 4 项分别与给定的特征项对应的字段进行对比,对比后可知每一项均相等,那么对比第 5 项.

由于该项偏移值存在两种情况,故首先判断是否以集合形式出现的:若不是,直接比较值的大小;否则,按照集合关系进行比较.对于该例,偏移值为集合{624,1736},那么判断{624,1736}⊂{624,1736,…,8}是否为真;可以看出集合满足关系,说明该项特征与已知特征项匹配.对于剩余的已收集的特征项,按照上述方法进行匹配.若最终匹配的特征项与已知数据特征完全匹配,那么可判定其属于恶意软件.

4 系统设计

我们在开源 VMM Bitvisor[28]的基础上,设计并实现了原型系统 MDS-DCB.原型系统包括内存监控模块、动态内核对象分配监控模块、静态分析模块、特征构建模块等,其整体架构如图 8 所示.

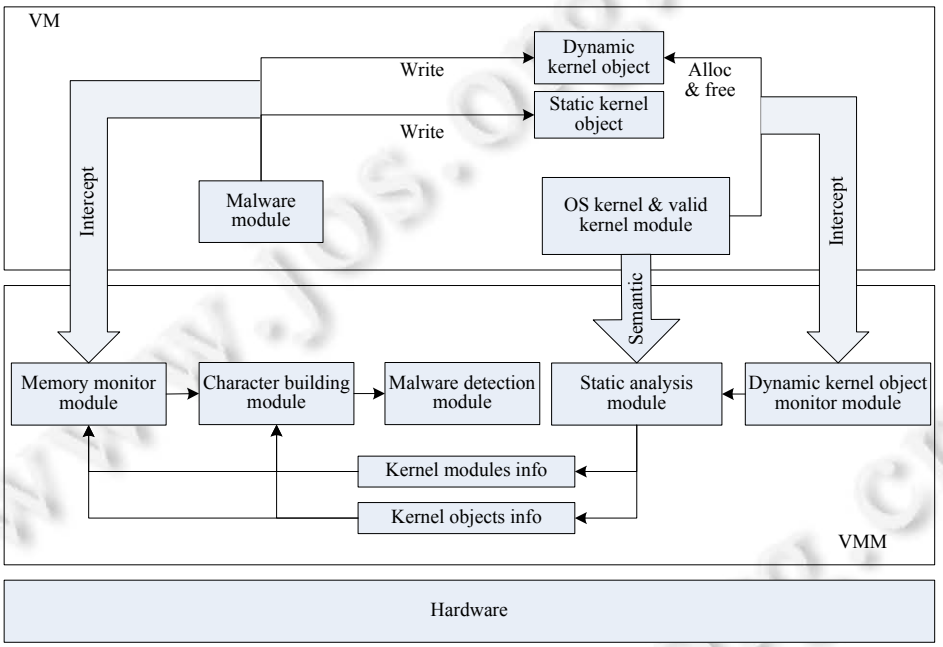

Fig.8 Architecture of MDS-DCB

图 8 系统架构

- 内存监视模块(memory monitor module)监视恶意软件和操作系统对特定内存区域数据的访问,在第 3.3 节已详细阐述了监视原理.由于读访问不会对内核带来实质性的伤害,同时出于效率考虑,本文只监控内核数据写操作;
- 动态内核对象分配监控模块(dynamic kernel object monitor module)监视内核对象分配与释放函数的调用.该模块在内存分配函数的首地址和返回处以及释放函数的首地址处插入 VMCALL 指令,客户机在执行内存分配、返回或者释放时会陷入到 VMM 中.然后,我们在 VMM 中处理 VMCALL 指令产生的 VM Exit 事件,通过 VMCALL 陷入时的地址判断陷入类型:分配函数陷入、释放函数陷入和函数返回陷入.如果是分配函数陷入,则需要分析参数信息;如果是函数返回陷入,则需要分析函数返回结果.这两种陷入分析的内容用于构建内核对象映射关系.如果是释放函数陷入,则需要分析参数,更新内核对象映射关系;
- 静态分析模块(static analysis module)提供操作系统到 VMM 层的语义转换,实现对数据类型的推导,并提供上述两个模块所需要的操作系统信息相关的接口.提供的接口包括:查找内核模块及获取内核模块信息的接口;读取内核对象分配及释放函数参数的接口.获取内核模块信息是通过对模块加载系统调用参数解析实现的.读取内核对象分配及释放函数参数时参照 stdcall 标准:从右向左压栈;

- 特征构建模块(character building module)主要根据内存监控模块和操作系统相关子模块提供的信息建立内核数据行为访问模板,然后,根据公式(1)构建内核恶意软件数据特征.此外,该模块在非特征构建时收集内核运行过程的数据访问行为;
- 恶意软件检测模块(malware detection module)根据恶意软件检测算法对内核运行过程中是否出现恶意软件特征进行匹配,以发现恶意软件.

5 实验及结果分析

本节从数据特征提取、有效性和性能 3 个方面对 MDS-DCB 进行评测.数据特征提取用于验证 MDS-DCB 能否获取恶意软件的数据特征,有效性测试用于验证 MDS-DCB 能否实现恶意软件的检测,性能测试用于验证 MDS-DCB 的效率和开销.实验环境如下:主机 CPU 为 Intel(R) Core(TM) i5-750@2.67GHz,内存大小 4GB. Bitvisor 版本为 1.3.0,Guest OS 采用的是 3.2.43-x86_64 内核的 Ubuntu 12.04.

5.1 特征提取测试

在进行恶意软件特征构建测试前,我们首先验证 MDS-DCB 能够实现内核动态数据对象的识别和监控.如表 3 所示,我们给出了客户机操作系统从开机开始 10min 运行时间内被捕获的部分内核动态数据对象相关信息.表 3 中,第 1 栏、第 2 栏分别是主类型和分类型,第 3 栏、第 4 栏给出了分配对象调用的分配函数位置和对象声明位置,第 5 栏给出了静态分析动态数据对象类型的推导类型,最后一栏给出了捕获的动态数据个数.从表 3 中我们可知,本文的监控方法和分析方法能够有效地监控对象的分配并识别分配对象的数据类型.

Table 3 Information of kernel objects identification

表 3 内核对象识别信息

类别	数据类型	分配代码位置	动态数据声明/函数声明位置	类型推导方法	动态数据对象个数
进程管理类	*task_struct*	*fork.c*:300	*fork.c*:294	(a)	380
	sighand_struct	*fork.c*:991	*fork.c*:985	(a)	160
		exec.c:983	*exec.c*:978	(a)	1
	signal_struct	*fork.c*:1038	fork.c:1033	(a)	165
内存管理类	*mm_struct*	*fork.c*:581	*fork.c*:579	(a)	8 230
		fork.c:816	*fork.c*:810	(a)	43
	pgd_t	*fork.c*:487	*fork.c*:487	(b)	142
	vm_area_struct	*mmap.c*:1457	*mmap.c*:1401	(a)	16 520
		mmap.c:2575	*mmap.c*:2511	(a)	457
	anon_vma_chain	*rmap.c*:116	*rmap.c*:114	(c)	1 569
文件系统类	*file*	*file_table.c*:124	*file_table.c*:110	(a)	7 680
	fs_struct	*fs_struct.c*:112	*fs_struct.c*:112	(b)	142
	files_struct	*file.c*:306	*file.c*:300	(a)	142
	dentry	*dcache.c*:1255	*dcache.c*:1252	(a)	15 289
	inode	*inode.c*:209	*inode.c*:204	(a)	15 004
	fasync_struct	*fcntl.c*:593	*fcntl.c*:591	(c)	348
网络管理类	*sock*	*sock.c*:1230	*sock.c*:1225	(a)	79
	socket	*socket.c*:1136	*socket.c*:1130	(a)	150
	socket_alloc	*socket.c*:246	*socket.c*:243	(a)	150
	inet_bind_bucket	*inet_hashtables.c*:36	*inet_hashtables.c*:36	(b)	29
模块类	*module*	*module.c*:2402	*module.c*:2402	(b)	56
	load_info	*module.c*:2513	*module.c*:2513	(a)	56
	module_kobject	*params.c*:737	*params.c*:729	(a)	56

然后,我们给出了几款典型内核恶意软件数据特征的分析过程,分析结果见表 4.为了使得 rootkit 能够在 3.2.43 版本内核下运行,我们对这些 rootkit 进行了改造,但不影响 rootkit 的具体功能.

首先,表 4 统计了 4 次同一款 rootkit 攻击样例下系统运行过程中数据类型个数和发生写访问的次数,第 1 列表示 rootkit 名称,第 3 列给出了不同情况下的静态数据类型及其写位置和写次数,第 4 列给出了动态数据类个数及其写位置和写次数.

根据表 4 及 rootkit 的内核数据访问过程,adore 0.38 的数据特征为(η,1,0,0xffffffff8180320,624),(η,1,0,

0xffffffff8180320,1736),…,(η,1,0,0xffffffff8180320,8),(η,1,1,*fork.c*:300/*task_struct*,692),(η,1,1,*fork.c*:300/*task_struct*,668),…,(η,1,1,*fork.c*:300/*task_struct*,720),…,(*uaccess_x*64.*h*:55,1,1,*slab_def.h*:130/*linux_dirent*64,720),…;sk 1.3b 的数据特征为(η,1,0,0xffffffff8180320,448),(η,1,0,0xffffffff8180320,456),…,(η,1,0,0xffffffff8180320,24),…,(η,1,1,*fork.c*:300/*task_struct*,692),(η,1,1,*fork.c*:300/*task_struct*,20),…;wipemod 的数据特征为(η,1,1,*module.c*:2402/*module*,8).因此,MDS-DCB 能够有效分析恶意软件的数据访问过程并获取恶意软件的数据特征.

Table 4 Analysis of data characters of rootkits

表 4 rootkits 数据特征分析

rootkit	D/S	静态数据类			动态数据类		
		数据类型	写位置	写次数	数据类型	写位置	写次数
adore 0.38	$D_{adore0.38,1}$	30 450	8 568	105 912	565	6 689	74 358
	$D_{adore0.38,2}$	30 450	8 657	109 247	565	6 780	74 985
	$D_{adore0.38,3}$	30 450	8 631	99 879	565	6 693	75 134
	$D_{adore0.38,4}$	30 450	8 515	101 562	565	6 784	74 657
	$\cap D_{adore0.38}$	30 450	7 624	95 489	493	6 012	71 038
	$\cup D_{begin}$	30 450	21 457	173 056	607	10 577	98 718
	$S_{adore0.38}$	1	1	7	2	2	14
sk 1.3b	$D_{suckit,1}$	30 450	8 600	105 912	565	6 749	74 358
	$D_{suckit,2}$	30 450	8 553	109 247	565	6 750	74 985
	$D_{suckit,3}$	30 450	8 531	99 879	565	6 723	75 134
	$D_{suckit,4}$	30 450	8 611	101 562	565	6 684	74 657
	$\cap D_{suckit}$	30 450	7 699	95 489	493	5 912	71 038
	$\cup D_{begin}$	30 450	21 457	173 056	607	10 577	98 718
	S_{suckit}	1 192	1	9	5	8	12
wipemod	$D_{wipemod,1}$	30 450	8 552	105 912	565	6 694	74 358
	$D_{wipemod,2}$	30 450	8 597	109 247	565	6 678	74 985
	$D_{wipemod,3}$	30 450	8 644	99 879	565	6 757	75 134
	$D_{wipemod,4}$	30 450	8 613	101 562	565	6 724	74 657
	$\cap D_{wipemod}$	30 450	7 724	95 489	493	6 078	71 038
	$\cup D_{begin}$	30 450	21 457	173 056	607	10 577	98 718
	$S_{wipemod}$	0	0	0	1	1	1

5.2 有效性测试

为了进一步验证所提取的恶意软件数据特征的效果,我们在实验中选取了一些较有代表性的恶意代码进行了测试,特别是选择了基于 JOP 技术的 JOP_hideprocess[6].测试结果见表 5.

Table 5 Results of kernel malware detection

表 5 内核恶意软件检测测试结果

rookit	特征值	检测有效性
enyelkm v1.2	(η,1,1,*module.c*:2402/*module*,8),(η,1,0,0xffffffff8158a280,0),…,(η,1,0,0xffffffff81dd4000,640),(η,1,1,*fork.c*:300/*task_struct*,1120),…	√
override	(η,1,0,0xffffffff8180320,816),(η,1,0,0xffffffff8180320,856),...,(η,1,1,*fork.c*:300/*task_struct*,624)	√
aodre-ng-0.56	(η,1,0,0xffffffff811d4b60,0),(η,1,0,0xffffffff811dae80,0),(η,1,0,0xffffffff811d4ab0,0),(η,1,1,*file_table.c*:124/*file*,56),…	√
kbeast v1.0	(η,1,1,*module.c*:2402/*module*,8),(η,1,0,0xffffffff8180320,0),(η,1,0,0xffffffff8180320,8),…,(η,1,0,0xffffffff8158a280,0),(η,1,1,*fork.c*:300/*task_struct*,1120),…	√
rkit v1.0	(η,1,0,0xffffffff8180320,840),(η,1,1,*fork.c*:300/*task_struct*,1116),(η,1,1,*fork.c*:300/*task_struct*,1120)	√
synapsys	(η,1,0,0xffffffff8180320,16),(η,1,0,0xffffffff8180320,816),…,(η,1,0,0xffffffff8180320,1424),…,(η,1,1,*fork.c*:300/*task_struct*,1120),…,(η,1,1,*fork.c*:300/*task_struct*,20),…	√
CaRoGNa	(η,1,0,0xffffffff8180320,624),(η,1,0,0xffffffff8180320,496),…,(η,1,1,*file_table.c*:124/*file*,132),(η,1,1,*file_table.c*:124/*file*,136),(η,1,1,*fork.c*:300/*task_struct*,20),…	√
JOP_hideprocess	(*list.h*:104,1,1,*fork.c*:300/*task_struct*,1120)	√

由表 5 可以看出,本文提出的方法正确提取了各款内核恶意软件的数据特征并成功检测出它们.目前是在植入单一恶意软件的情况下提取恶意软件的数据特征,当同时植入多款恶意软件时,那么运行过程中将捕获到

多款恶意软件的数据特征,由于已提取的特征是多款恶意软件运行过程中数据特征的子集,根据检测算法 JudgeMalware 可知已提取的特征可检测出恶意软件.

为了检验本文方法的误报率,我们使用了一些常用应用程序进行对比检测,其中,LTP(linux test project)测试工具集是目前较为流行的 Linux 基本功能测试集,它包含了多个子功能测试模块,例如系统调用、系统命令、内存分配、磁盘读写、文件系统、网络、数学运算测试等,对 Linux 内核进行各项长时间(24 小时)的测试,可较充分地覆盖内核代码.同时,在运行这些测试程序时给予不同的输入数据或者配置,使其能够运行更多的功能.根据提取的恶意软件数据特征对其进行检测,通过特征匹配判断是否会造成误报,结果见表 6.

Table 6 Results of false positive test

表 6 误报测试结果

benchmark	命令	静态数据类			动态数据类			误报
		数据类型	写位置	写次数	数据类型	写位置	写次数	
内核编译	make	30 450	14 568	165 912	565	6 989	79 358	×
解压缩 bzip2	tar –zxf linux-source.tar.bz2	30 450	8 657	109 247	565	6 780	74 985	×
压缩 gzip	tar –zcf linux-source-dir	30 450	8 631	99 879	565	6 693	75 134	×
unixbench 5.1.2	./Run	30 450	16 515	191 531	563	8 484	104 373	×
Linux 启动	linux boot	30 450	8 515	101 562	564	6 784	74 657	×
LTP	./runalltesh.sh	30 450	31 785	265 714	1128	15 879	210 367	×
apache eberver	./apachectl start	30 450	15 253	180 267	570	7 127	112 589	×
mysql	Service mysql start	30 450	16 308	188 674	565	6 954	83 521	×
thttpd	thttpd thttpd.conf	30 450	17 469	210 679	583	8 739	142 175	×

由表 6 可知,所有的 benchmark 均没有误报产生.这也进一步验证了提取的数据特征是合理的、有效的.

为了比较测试结果,我们使用现有的内核恶意软件检测工具与本文的原型系统进行对比分析,对比结果见表 7.

Table 7 Results of comparison test

表 7 对比测试结果

rootkit	攻击类型	secVisor	Gibraltar	NumChecker	MDS-DCB (ours)
adore 0.38	代码植入、控制数据篡改	√	√	√	√
sk 1.3b	代码植入、控制数据篡改	×	×	√	√
enyelkm v1.2	代码植入和篡改	×	×	√	√
override	代码植入、控制数据篡改	√	√	√	√
wipemod	数据篡改	√	√	×	√
kbeast v1.0	代码植入、控制数据篡改	√	√	√	√
synapsys	代码植入、控制数据篡改	√	√	√	√
CaRoGNa	代码植入、控制数据篡改	√	√	√	√
JOP_hideprocess	代码复用、非控制数据篡改	×	√	×	√
JOP_Synapsys	代码复用、控制数据篡改	×	√	√	√
ptrace_detach_hook[15]	非控制数据篡改	×	×	×	√

根据表 7 可以看出:

- secVisor 无法检测基于 JOP,ROP 实现的 rootkit 和 ptrace_detach_hook,这是因为 JOP_hideprocess,JOP_Synapsys 和 ptrace_detach_hook 绕过了代码完整性的缘故;
- Gibraltar 无法检测 sk 1.3b,enyelkm 和 ptrace_detach_hook,这是因为这 3 款 rootkit 未违背 Gibraltar 提出的数据不变量属性;其能够检测基于代码复用攻击的 JOP_hideprocess 和 JOP_Synapsys,是因为这两款 rootkit 违背了 $run\_list \subseteq all\_tasks$ 和系统调用表表项的值被破坏;
- NumChecker 无法检测 wipemod,JOP_hideprocess 和 ptrace_detach_hook,这是因为 NumChecker 是通过检测系统调用执行变化情况,而这 3 款 rootkit 并未修改任何系统调用;
- MDS-DCB 则检测出了所有的 rootkit,这也表明了本文提出的方法较现有的内核恶意软件检测工具具有更强的检测能力,能够检测代码植入、代码篡改、代码复用、控制数据篡改和非控制数据篡改等恶

意攻击.

5.3 性能测试

5.3.1 微基准测试

我们采用 lmbench 作为本次实验的微基准测试集.由于 MDS-DCB 基于内存分配/释放函数监控和内存读写监控分析技术实现,为了充分测试 MDS-DCB 的性能开销,我们从 lmbench 选择了与内存相关的系统调用、缺页处理和内存读写等作为基准测试指标.通过分别测试 MDS-DCB、Bitvisor、裸机三者下的性能开销,并通过对比来分析 MDS-DCB 所引入的性能开销,如图 9 所示.

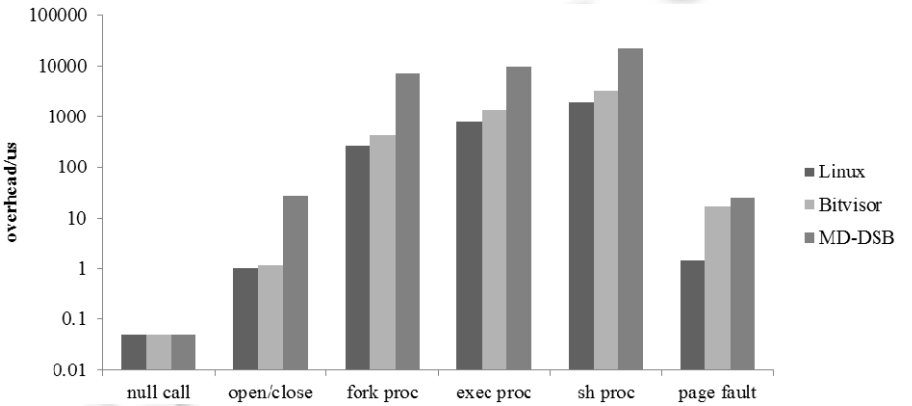

Fig.9 Microbenchmark results of MDS-DCB

图 9 MDS-DCB 微基准测试结果

其中,由于不同程序测试的性能开销数据差距较大,故我们对测试数据(纵坐标)进行对数化(基数为 10)处理.根据图 9 的数据可知:由于 MDS-DCB 未对 Null Call 操作进行监控,直接由处理器处理,故未带来任何开销;而 open/close 操作是先打开文档后关闭文档,这一过程中创建了 file 内核对象,MDS-DCB 对 file 结构的创建和使用进行了监控,Linux,Bitvisor 未进行监控,故 MDS-DCB 较 Linux 和 Bitvisor 带来了额外开销;同时,MDS-DCB 在 fork proc,exec proc 和 sh proc 这 3 项测试指标下的性能远大于其他指标,并且开销比 Bitvisor 大很多,这是因为这 3 项指标的测试过程涉及到进程类对象、文件类对象、内存类对象的创建、使用等操作,MDS-DCB 对这些内核对象进行了监控.prot fault 指的是保护异常带来的开销,MDS-DCB 和 Bitvisor 一样只是捕获该操作,捕获之后将该操作的处理返还给操作系统内核,故开销较小;page fault 指的是缺页异常带来的开销,由于 MDS-DCB 设置了监控的内核数据所在页的写属性,当产生写关键数据操作时,触发缺页异常,由 VMM 处理该异常,处理之后再返回内核,所以 page fault 所需要的开销较大,且大于 prot fault 的开销.

5.3.2 应用程序基准测试

为了进一步评测 MDS-DCB 的性能,我们选择了与误报测试一样的应用程序基准测试集测试 MDS-DCB 的性能开销.测试结果如图 10 所示.

由于内核编译和 unixbench 是综合性应用,既需要 CPU 时间,也需要大量的 I/O 操作,测试过程中大量使用系统资源如文件系统、管道和进程等,这些行为调用了内核服务如系统调用和缺页处理而触发内核的内存活动,而 MDS-DCB 监控内核对象的创建、读写等行为,故带来了较大的性能开销.对于解压缩和压缩程序,分别引入了 15%和 11%的性能开销,它们属于计算密集型应用,相对内核编译和 unixbench 开销较小.Linux 启动可全面测试 MDS-DCB 的性能,因为系统启动过程中囊括了 MDS-DCB 的所有监控和操作的内容,该项测试引入的开销为 45%.综上测试,MDS-DCB 的性能开销在一个可以接受的范围内.

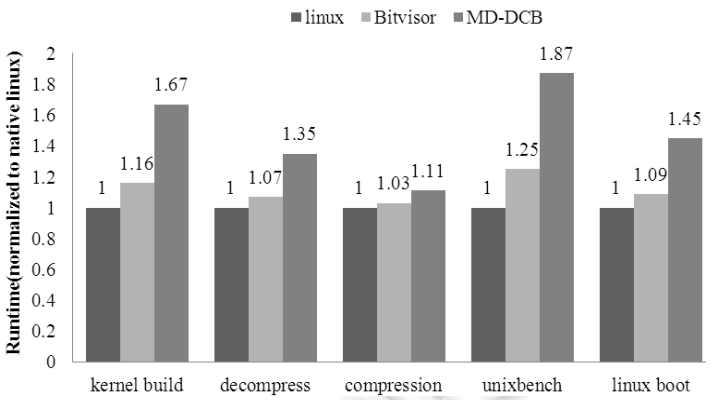

Fig.10 Application benchmark performance test results

图 10 应用基准程序性能测试结果

5.4 方法存在的不足

实验过程中选择的恶意代码测试用例都是目前公开的,由于特征库中存储有他们所有的特征值,所以只要特征值是合理有效的,那么均可以被检测出来.对于未知的内核恶意软件,若采用上述提出的特征匹配算法,那么将可能会产生漏报,因为在未知恶意软件运行过程中提取的数据特征集与已知特征无法匹配成功.另外,对于恶意软件修改自己行为迷惑检测软件的情况,本文的方法也有局限性.修改主要包括两种情况,如图 11 所示.

- 第 1 种情况,如图 11(a)所示,攻击者在不改变原恶意软件功能的基础上增加了一些冗余功能,那么假设新的恶意软件对应的数据特征为 S',过程中采集的数据访问行为集为 D,根据匹配算法,首先求解 D 与 S_M 的交集.从图 11 中可知,该运算结果为 S_M.所以,这种情况下仍可以检测出恶意软件;
- 第 2 种情况,如图 11(b)所示,攻击者修改了原恶意软件的部分功能并增加一些冗余功能,那么同样假设新的恶意软件对应的数据特征为 S',过程中采集的数据访问行为集为 D,根据匹配算法,首先求解 D 与 S_M 的交集.从图中可知,该运算结果为 $S'-R$.而$(S'-R)\neq S_M$,根据算法可知,无法判断是否存在恶意软件,从而造成匹配失效,导致漏报.对于以上提到的不足,我们需要进一步研究不同恶意软件数据特征之间的关系,降低漏报率,提升检测准确率.

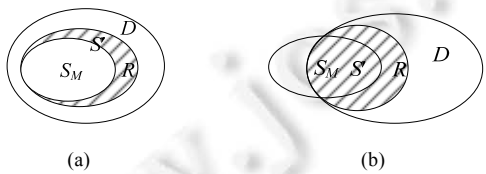

Fig.11 Data characteristic relation between malware and modified malware

图 11 恶意软件修改自身行为后的数据特征关系

6 结 论

本文提出了一种基于数据特征的内核恶意软件检测方法,并设计实现了原型系统 MDS-DCB.该方法基于内核对象访问模型给出了检测方法的基本思路;在内核运行过程中监控内核动态数据的分配事件和内核数据的访问构建恶意软件数据特征;并基于该数据特征检测内核恶意软件.MDS-DCB 在 VMM 层实现,无需修改操作系统内核,具有良好的兼容性和安全性.通过对原型系统的有效性和性能进行测试,测试结果表明:MDS-DCB 能够有效检测内核恶意软件,并且引入的性能开销在一个可接受的范围内.目前,未对分析得到的恶意软件特征

进行处理,特征匹配是通过对比每条 SKDAP 实现的,这将影响检测效率,下一步将研究基于 SKDAP 的特征值计算问题.同时,不同的内核恶意软件之间存在相同或者类似的数据特征,故,特征的相似性问题也是下一步需要研究的内容.

致谢 在此,向对本文研究工作提供基金支持的单位和评阅本文的审稿专家表示衷心的感谢,向为本文研究工作提供基础和研究平台的前辈致敬.

References:

[1] Seshadri A, Luk M, Qu N, Perrig A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proc. of the 21st ACM SIGOPS Symp. on Operating Systems Principles. New York: ACM Press, 2007. 335−350. [doi: 10.1145/1294261.1294294]

[2] Riley R, Jiang X, Xu D. Guest-Transparent prevention of kernel Rootkits with VMM-based memory shadowing. In: Proc. of the 11th Int'l Symp. on Recent Advances in Intrusion Detection. Cambridge: Berlin, Heidelberg: Springer-Verlag, 2008. 1−20. [doi: 10.1007/ 978-3-540-87403-4_1]

[3] Hund R, Holz T, Freiling FC. Return-Oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proc. of the 18th Conf. on USENIX Security Symp. Berkeley: Usenix, 2009. 383−398.

[4] Carlini N, Wagner D. Rop is still dangerous: Breaking modern defenses. In: Proc. of the 23rd Conf. on USENIX Security Symp. Berkeley: Usenix, 2014. 385−399.

[5] Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-Oriented programming: A new class of code-reuse attack. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security. New York: ACM Press, 2011. 30−40. [doi: 10.1145/1966913.1966919]

[6] Chen P. Research on the attack and defense techniques of code Reuse [Ph.D. Thesis]. Nanjing: Nanjing University, 2012 (in Chinese with English abstract).

[7] Kruegel C, Robertson W, Vigna G. Detecting kernel-level rootkits through binary analysis. In: Proc. of the 20th IEEE Annual Conf. on Computer Security Applications. Washington: IEEE CS Press, 2004. 91−100. [doi: 10.1109/CSAC.2004.19]

[8] Musavi SA, Kharrazi M. Back to static analysis for kernel-level rootkit detection. IEEE Trans. on Information Forensics and Security, 2014,9(9):1465−1476. [doi: 10.1109/TIFS.2014.2337256]

[9] Riley R, Jiang X, Xu D. Multi-Aspect profiling of kernel rootkit behavior. In: Proc. of the 4th ACM European Conf. on Computer Systems. New York: ACM Press, 2009. 47−60. [doi: 10.1145/1519065.1519072]

[10] Wang X, Karri R. NumChecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In: Proc. of the 50th Annual Design Automation Conf. New York: ACM Press, 2013. 79−86. [doi: 10.1145/2463209.2488831]

[11] Sharif MI, Lanzi A, Giffin JT, Lee W. Impeding malware analysis using conditional code obfuscation. In: Proc. of the 16th Annual Network and Distributed System Symp. Washington: Internet Society, 2008. 65−88.

[12] Fan W, Lei X, An J. Obfuscated malicious code detection with path condition analysis. Journal of Networks, 2014,9(5):1208−1214. [doi: 10.4304/jnw.9.5.1208-1214]

[13] Li J, Wang Z, Bletsch T, Srinivasan D, Grace M, Jiang XX. Comprehensive and efficient protection of kernel control data. IEEE Trans. on Information Forensics and Security, 2011,6(4):1404−1417. [doi: 10.1109/TIFS.2011.2159712]

[14] Baliga A, Ganapathy V, Iftode L. Detecting kernel-level rootkits using data structure invariants. IEEE Trans. on Dependable and Secure Computing, 2011,8(5):670−684. [doi: 10.1109/TDSC.2010.38]

[15] Vogl S, Gawlik R, Garmany B, Kittel T, Pfoh J, Eckert C, Holz T. Dynamic hooks: Hiding control flow changes within non-control data. In: Proc. of the 23rd USENIX Security Symp. Berkeley: Usenix, 2014. 813−828.

[16] Donghai T, Xuanya L, Changzhen H, Huaizhi Y. OPKH: A lightweight online approach to protecting kernel hooks in kernel modules. China Communications, 2013,10(11):15−23. [doi: 10.1109/CC.2013.6674206]

[17] Zhu F. Integrity-Based kernel malware detection [Ph.D. Thesis]. Florida International University, 2014.

[18] Rhee J, Xu D. LiveDM: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Technical Report 2010-02. Purdue University at West Lafayette, 2010. 1−20.

[19] Bergeron J, Debbabi M, Desharnais J, Erhioui MM, Lavoie Y, Tawbi N. Static detection of malicious code in executable programs. In: Proc. of the Symp. on Requirements Engineering for Information Security. 2001. 184−189.

[20] Yin H, Song D, Egele M, kruegel C, Kirda E. Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2007. 116−127. [doi: 10.1145/1315245. 1315261]

[21] Xu L, Su Z. Dynamic detection of process-hiding kernel rootkit. Technical Report, CSE-2009-24, University of California at Davis, 2009. 1−12.

[22] Li B, Wo TY, Hu CM, Li JX, Wang Y, Huai JP. Hidden OS objects correlated detection technology based on VMM. Ruan Jian Xue Bao/Journal of Software, 2013,24(2):405−420 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4265.htm [doi: 10.3724/SP.J.1001.2013.04265]

[23] Petroni N, Fraser T, Walters A, Arbaugh WA. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: Proc. of the 15th Conf. on USENIX Security Symp. Berkeley: Usenix, 2006. 289−304.

[24] Ren P, Wang X, Wu C, Zhao B, Sun H. A semantic-based malware detection system design based on channels. information and communication technology. Cambridge: Berlin, Heidelberg: Springer-Verlag, 2014. 653−662. [doi: 10.1007/978-3-642-55032-4_67]

[25] Cozzie A, Stratton F, Xue H, King ST. Digging for data structures. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. Berkeley: Usenix, 2008. 255−266.

[26] Wang XL, Wang ZL, Sun YF, Liu Y, Zhang BB, Luo YW. Detecting memory leak via VMM. Chinese Journal of Computers, 2010,33(3):463−472 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2010.00463]

[27] Rhee J, Riley R, Lin Z, Jiang X, Xu D. Data-Centric OS kernel malware characterization. IEEE Trans. on Information Forensics and Security, 2014,9(1):72−87. [doi: 10.1109/TIFS.2013.2291964]

[28] Shinagawa T, Eiraku H, Tanimoto K, hasegawa S, Hirano M, Kourai K, Oyama Y, kawai E, Kono K, Chiba S, Shinjo Y, Kato K. Bitvisor: A thin hypervisor for enforcing I/O device security. In: Proc. of the 2009 ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. New York: ACM Press, 2009. 121−130. [doi: 10.1145/ 1508293.1508311]

附中文参考文献:

[6] 陈平.代码复用攻击与防御技术研究[博士学位论文].南京:南京大学,2012.

[22] 李博,沃天宇,胡春明,等.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 [doi: 10.3724/SP.J.1001.2013.04265]

[26] 汪小林,王振林,孙逸峰,等.利用虚拟化平台进行内存泄露探测.计算机学报,2010,33(3):463−472. [doi: 10.3724/SP.J.1016.2010.00463]

陈志锋(1986－),男,福建漳州人,博士生,主要研究领域为信息安全,可信计算.

张平(1969－),女,博士,副教授,CCF 专业会员,主要研究领域为并行识别,并行编译,信息安全.

李清宝(1967－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,可信计算.

丁文博(1985－),男,讲师,CCF 专业会员,主要研究领域为信息安全.