# 基于快速语义修复的操作系统隐藏对象检测技术

李勇钢[1,2],崔超远[1],乌 云[3],孙丙宇[1]
(1. 中国科学院合肥物质科学研究院,合肥智能机械研究所智能决策实验室,安徽合肥 230031;2. 中国科学技术大学,安徽合肥 230027;3. 中国科学院合肥物质科学研究院应用技术所,安徽合肥 230088)

**摘 要:** 与传统的入侵检测系统相比,基于虚拟机自省的入侵检测系统的抗干扰性更强.但由于存在语义鸿沟问题,即低层的硬件字节信息与操作系统级语义之间的差异,导致入侵检测系统的通用性和实时性下降.针对此问题,本文提出了Vlhd,一种基于语义鸿沟修复方法的rootkit隐藏对象检测技术.Vlhd将系统分离成离线和在线模块两部分.在线模块用于即时地在虚拟机外部重构虚拟机语义视图;离线模块用于离线地提取操作系统语义知识,并向在线模块提供语义服务.通过对各类Linux操作系统和多种rootkit进行入侵检测试验,发现Vlhd对rootkit的隐藏对象检测效果良好,通用性强.Vlhd的单次扫描时间为34ms,对系统引入了1.1%(扫描周期设置为8s时)的性能开销.

**关键词:** 虚拟化;隐藏检测;rootkit;xen;语义修复
**中图分类号:** TP316 **文献标识码:** A **文章编号:** 0372-2112 (2018)05-1025-07
**电子学报URL:** http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2018.05.001

# The OS Hidden Object Detection Technology Based on Fast Semantic Repair

LI Yong-gang[1,2],CUI Chao-yuan[1],WU Yun[3],SUN Bing-yu[1]
(1. *Lab of Intelligent Decision,Institute of Intelligent Machines,Hefei Institutes of Physical Science,Chinese Academy of Sciences,Hefei,Anhui* 230031,*China*;2. *University of Science and Technology of China,Hefei,Anhui* 230027,*China*;
3. *Institute of Applied Technology,Hefei Institutes of Physical Science,Chinese Academy of Sciences,Hefei,Anhui* 230088,*China*)

**Abstract:** Compared with the traditional intrusion detection system,the intrusion detection system based on virtual machine introspection has stronger anti-jamming ability. However,the difference between the hardware byte information and the high level semantics leads to the decrease of the versatility and real-time. Aiming to solve this problem,this paper proposes Vlhd,a OS hidden object detection technology based on semantic repair method. Vlhd separates the system into two parts:the offline module and the online module. The online module reconstructs semantic view outside the target virtual machine;the offline module is responsible for the extraction of OS semantic knowledge,and providing semantic services to the online module. Through a variety of rootkit intrusion detection test in various types of Linux OS,we find that Vlhd can detect the objects hidden by rootkits. A single scan time of Vlhd is 34ms,and the performance overhead is 1.1% (scan cycle is set to 8s).

**Key words:** virtualization;hidden detection;rootkit;xen;semantic repair

## 1 引言

随着虚拟化市场的不断扩大,虚拟机(VM)已经成为rootkit攻击的新目标[1~3].新生恶意软件也往往和rootkit相结合进行自我隐藏,以规避检测.受商业利益驱使,rootkit开发者不断提升其隐藏性,所采用隐藏的技术也由操作系统指令的简单替换升级为劫持内核控制流、直接内核对象操作等先进技术[4,5],导致恶意软件更加难以被检测到.

近年来,研究人员针对rootkit进行了大量关于安

收稿日期:2016-10-12;修回日期:2017-07-21;责任编辑:蓝红杰
基金项目:国家自然科学基金(No.31371340);国家重点研发计划基金(No.2016YFB0502604)

全检测服务[6]的研究,采用虚拟机自省(VMI)技术提出了多种入侵检测模型[7~9]. 基于 VMI 的检测工具被从 VM 内部移至外部,使其与目标操作系统之间具有了更高的隔离性. 另外,VMI 检测技术能够取得并分析 VM 硬件字节信息,以查看 VM 内部状态. 但由于硬件的字节信息与高层语义之间存在巨大差异,因此,此类方法也面临着一项重大挑战—消除语义鸿沟.

解决语义鸿沟问题,需要对底层的硬件状态信息进行系统级的语义重构. 语义重构涉及到底层硬件状态数据、高层语义信息和语义知识. 其中,语义知识是桥接底层状态数据和高层语义信息的桥梁. 但不同操作系统之间的语义知识存在差异,即使是细微差距,都可能导致检测系统失灵. 语义知识的获取主要采用在线内核语义截取和手动获取两类方法. 前者会严重影响到系统的执行效率;后者则使系统通用性受到限制.

针对以上问题,本文本文提出了 Vlhd,一种基于轻量级语义修复的 rootkit 隐藏对象检测方法. Vlhd 离线地构建完备的语义知识库,在线对语义进行重构,能够检测到目标 VM(TVM)中被 rootkit 隐藏的对象. 通过与与其他检测工具的对比可发现,Vlhd 在检测效果、效率以及通用性上都有较大的优势.

## 2 相关工作

研究者们采用 VMI 技术解决语义鸿沟问题并展开了大量的研究工作. X-TIER[7]采用介入 TVM 的方式获取 VM 状态信息. 首先向 TVM 中注入模块,然后通过该模块读取 TVM 的数据结构以获取 VM 的状态信息,最后以超级调用(Hypercall)的形式将获取的信息实时传递给 VM 管理器(VMM). 与 X-TIER 不同,SYRINGE[8]采用 function-call injection 技术,使其能够在 VM 外部对其内部的函数进行调用. 同时采用 Localized shepherding 技术监控数据获取代码的控制流完整性. X-TIER 与 SYRINGE 的实现过程需 VM 内外频繁交互,影响系统性能.

LibVMI[10]无需介入 TVM 即可获得其内部状态. 用户通过 LibVMI 接口可获取到诸如进程列表、网络端口、可加载模块等 TVM 状态信息. 但 LibVMI 生成语义视图的过程,需要 TVM 内部的特定文件的支持. 如对进程描述符和可加载内核模块的定位,都需要在线手动提取 TVM 的 system. map 文件,导致 LibVMI 与 TVM 之间存在强耦合的特点.

Virtuoso[11]和 VMST[12]从控制逻辑角度着手,解决语义鸿沟问题. Virtuoso 首先进行训练,将 VM 内部程序在 TVM 内运行多次,提取相关指令及指令执行路径;之后动态地提取生成自省代码所需的路径;最后将其融合翻译为可在 VM 外执行语义重构的代码. VMST 针对 virtuoso 训练工作繁琐且自动化程度低的问题,将代码执行与数据访问进行分离后利用内核重定向技术,将处于 TVM 内部的代码与 VM 内存中的数据相结合生成自省程序. Virtuoso 与 VMST 的显著缺陷是性能损耗偏大.

上述方法自动化程度低,生成语义视图的过程大都依赖于专业的操作系统和内核知识,通用性不强. 此外,频繁的在线交互也会影响系统效率. 相比之下,Vlhd 一方面离线地建立了完备的语义知识库,使其对内核版本不再敏感,提高了通用性;另一方面,避免了检测过程中为获取语义信息而进行的在线交互和过多的手工操作,提高了检测效率.

## 3 Vhld 系统总体设计

Vlhd 的整体设计如图 1 所示,主要包括在线和离线模块两部分,它们在逻辑上相互联系,部署上相互独立. 离线模块负责构建语义知识库并向在线模块提供语义服务. 语义知识库需要在系统运行之前就已由高级语义解析模块生成,其中包含各类 OS 语义信息及内核操作函数. 当接收到索引请求时,离线模块会根据请求参数调用相关函数获取语义信息并反馈至在线模块. 完成上述动作后,离线模块进入静默状态,等待下一个索引请求.

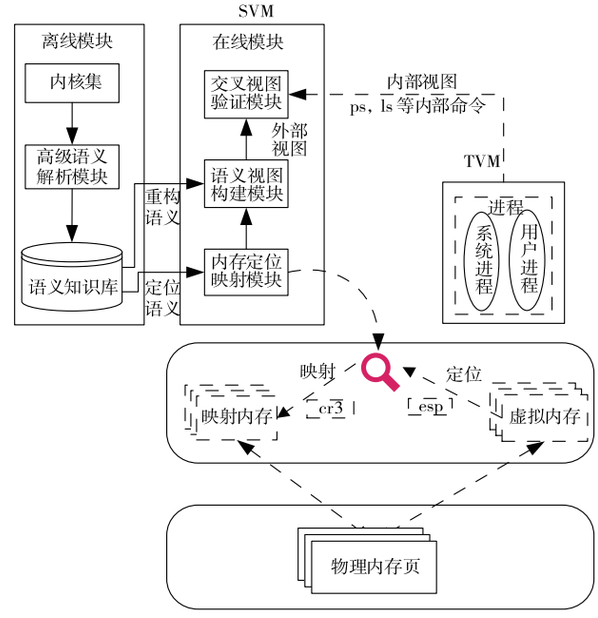


图1 Vlhd总体架构

在线模块部署在安全 VM(SVM)内部,监视客户 VM(GVM)状态. 在线模块采用在线即时处理的方式,通过 hypervisor 获取 GVM 的硬件字节信息,然后解析语义,进行入侵检测. 在线模块主要包括三部分:内存定位映射模块,语意视图构建模块及交叉视图验证模块. 内存定位映射模块定位 GVM 中监视对象(如进程)的虚拟地址,进行可读映射,然后精确读取特定地址中相应字节数的内存

内容;语义视图构建模块将传递过来的二进制形式的内存内容转换为高层语义;交叉视图验证模块对比内外语义视图,判定是否存在被隐藏的对象.

## 4　系统实现

### 4.1　GVM 内部状态

本文以进程监视作为 rootkit 检测方式,因为进程能够描述出 VM 当前正在进行的动作. 在 Linux 中,*task_struct* 描述进程状态,其中包含了进程名、进程号等状态信息. 所有 *task_struct* 组成双向循环链表,通过遍历双向链表获取到整个进程列表.

通过超级调用的方式,Vlhd 可读取指定 CPU 的上下文信息,从中提取出包括 *esp*、*cr3* 在内的各寄存器信息. 每个进程在创建时都会分配一个 8KB 大小的内核栈,其基址可通过屏蔽 *esp* 寄存器后 13 位得到. 以内核栈基址为起点的空间中,存储着数据结构 *thread_info*,它的第一个成员变量指向当前进程的 *task_struct*.

以单个进程作为操作对象,Vlhd 能够解析出 GVM 状态信息,如进程名(*task_struct* → *comm*),进程号(*task_struct* → *pid*)、进程状态(*task_struct* → *state*)以及文件(*task_struct* → *mm* → *mmap* → *vm_file* → *f_path* → *path* → *dentry*)等高层语义信息,也可得到进程代码段(*task_struct* → *mm* → *start_code*, *task_struct* → *mm* → *end_code*),数据段(*task_struct* → *mm* → *start_date*, *task_struct* → *mm* → *end_date*)等二进制内容[13,15].

### 4.2　构建语义知识库

语义知识库由封装函数和存储数据两部分组成,其数据存储设计采用如图 2 所示的树形层级结构,便于语义查询以及库内容扩展. 由“根”到“叶”依次为系统版本(32/64 位)、内核主版本 + 次版本、内核修订版本、内核中用于状态描述的数据结构、数据结构中的成员变量以及元数据. 数据节点是数据结构 *mem_data*,是对内核数据结构中成员变量属性的抽象,包含二类元数据:成员变量相对所属数据结构首地址的偏移量和成员变量的数据类型. 偏移量用于对成员变量在内存中进行精确定位. 数据类型用于两方面,一是确定读取内存的数量,二是在视图重构过程中确定相应数据应恢复为何种类型. 为获取元数据,需要对各内核版本分别进行编译处理. 编译只针对包含特定数据结构的源文件,缩短了单个内核的编译时间. 编译结束之后,目标文件返回各成员变量的数据类型和偏移量,以预定义的数据结构形式(*struct mem_data*)进行存储.

基于自动语义索引决策的需要,Vlhd 制定了 GVM 命名协议,以确定所要索引的内核版本和语义类型. 根据协议内容,GVM 命名应遵循 username – 32/64@ kernel_version 的形式. 例如,64 位 ubuntu12.04,内核 3.2.16 的 GVM,遵循协议应当命名为 xx-64@3.2.16. 在线模块以此命名为请求参数,传递给离线模块. 离线模块再根据请求参数,对语义知识库进行逐层索引,提取出特定的语义信息并将该信息反馈给在线模块.

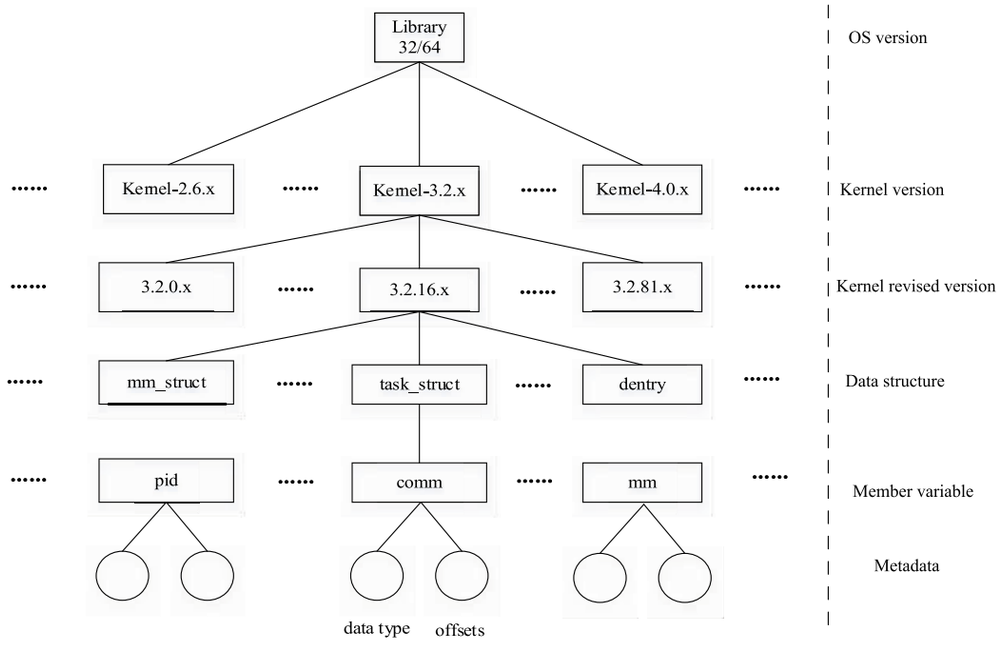


图2　语义知识库拓扑结构

语义库内的数据结构以及部分封装函数如算法1所示. 3~5行是库中数据存储结构. 函数 *GetData* 在内核编译时返回语义重构所需的元数据. 函数 *ReadAddr* 读取指针成员变量所指向的地址. 函数 *GetCurrent _ task _ struct* 读取当前进程的 *task _ struct* 的首地址,其中,17,18行通过超级调用读取TVM中虚拟CPU的上下文信息,提取出 *esp* 寄存器信息;19行屏蔽 *esp* 后13位得到当前进程的内核栈基址,即 *thread _ info* 的首地址;20,21行通过内存映射及读地址操作得到当前进程的 *task _ struct* 的虚拟首地址. 函数 *GetNext _ task _ struct* 获取双向链表中下一个 *task _ struct* 的首地址. 除上述进程相关操作函数之外,语义知识库中还封装了文件、模块以及网络端口等OS对象操作函数.

Vlhd中所有在GVM外部对其内存的操作都需首先进行内存映射,即调用 *MapMem* 之后,才能由Vlhd读取. *MapMem* 的实现,首先需要精确的定位成员变量在虚拟内存中的位置. 内核数据结构首地址加上特定成员变量相对首地址的偏移量(对应语义知识库中元数据的 *Offset* 项)即为该成员变量在VM中的虚拟地址. 然后借助 *cr3* 寄存器将虚拟地址转换为虚拟物理地址. CR3寄存器可以通过 *ctx→cr3* 操作从vcpu上下文信息提取出来. 得到CR3寄存器和虚拟地址之后,通过内存分页机制完成由虚拟地址到虚拟物理地址的转换;紧接着通过影子页表操作完成由虚拟物理地址到机器地址的转换;最终映射到SVM地址空间内,成为Vlhd的可读地址.

**算法1 库中数据结构以及部分封装函数**

```
1.  typedef string MemberType,StructType;
2.  typrdef unsigned long long AddrType;
3.  struct mem _ data{
4.      unsigned int Offset;
5.      string DataType;};
6.  mem _ data GetData (StructType struct _ type,MemberType member){
7.      struct mem _ data *data;
8.      data →Offset = &(((struct struct _ type *)0) →member);
9.      data →DataType = Stringof(struct _ type. member);
10.     return data;}
11.  AddrType ReadAddr(AddrType *src) {
12.     AddrType dest = 0;
13.     memcpy(&dest,src,sizeof(AddrType));
14.     return dest;}
15.  AddrType GetCurrent _ task _ struct() {
16.     vcpu _ ctx *ctx;
17.     ctx = get _ vcpu _ context (virtual _ machine _ name,vcpu);
18.     esp = ctx →user _ regs →esp;
19.     gva _ thread _ info = ~(THREAD _ SIZE -1)&esp;
20.     map _ thread _ info = MapMem(gva _ thread _ info);
21.     gva _ task _ struct = ReadAddr(map _ thread _ info);}
22.  AddrType GetNext _ task _ struct (AddrType task _ struct _ addr) {
23.     struct task _ struct gva _ next _ task _ struct;
24.     struct mem _ data data;
25.     data = GetData(task _ struct,tasks);
26.     MapAddr = MapMem(task _ struct _ addr + (data →Offset));
27.     gva _ next _ task _ struct = ReadAddr(MapAddr) - data →Offset;
28.     return gva _ next _ task _ struct;}
```

## 4.3 语义重构

算法2是语义重构中进程列表的获取过程. $\mathscr{V}$ 代表外部语义视图,$\mathscr{L}$ 代表构建的语义知识库,$\mathscr{P}$ 代表单个进程的语义视图,$\mathscr{T}$ 代表所要重构的语义条目集合. 算法可分为为:视图初始化(2行)、获取当前进程 *task _ struct* 的首地址(3行)、遍历双向链表以获取整个进程列表(4~10行)以及返回外部语义视图(11行)四个步骤. 其中4~7行解析单个进程的全部状态信息,如进程名、号、状态和所属用户等. 函数 *ParseItem* 用于语义重构,首先根据 *task _ struct* 首地址及元数据定位成员变量的虚拟地址,然后进行地址映射,之后根据元数据的数据类型在映射内存中读取特定字节数内容进行语义恢复.

**算法2 获取进程列表**

```
1.  Require external view 𝒱,semantic library 𝓛,process view 𝒫,intention 𝒯
2.  𝒱=𝒫=∅;
3.  gva _ task _ struct = 𝓛. GetCurrent _ task _ struct();
4.  do {
5.      for item ∈𝒯 {
6.          data = 𝓛. GetData(task _ struct,item);
7.          𝒫=𝒫∪ParseItem(gva _ task _ struct,data);}
8.      𝒱=𝒱∪𝒫;
9.      gva _ next _ task _ struct = 𝓛. GetNext _ task _ struct();
10.    } while (gva _ next _ task _ struct != gva _ task _ struct)
11. return 𝒱
```

## 4.4 交叉视图验证

交叉视图验证就是将GVM内部以及外部视图做对比,寻找视图差异. 外部视图由Vlhd生成,内部视图则通过内部指令ls、ps等获得. 为保证内外视图同步生成,防止VM状态不一致性[14]的出现,在线模块在产生动作之前会通过xenstore向GVM发送一个同步信号,GVM收到同步信号之后立刻调用内部命令,生成内部视图.

# 5 实验结果

## 5.1 实验环境

物理机采用Intel 4核i5处理器,主频2.8GHz,4G内存,200G硬盘. 虚拟化平台采用Xen4.1.2,VM实验环境如表1所示.

**表 1　实验环境**

| VM | CPU | 内存 | OS 发行版本 | 发行时间 | 内核版本 |
|---|---|---|---|---|---|
| SVM | 4 | 4GB | Ubuntu12.04 | 2012.04.26 | 3.2.16 |
| GVM11 | 1 | 1GB | Ubuntu10.04 | 2010.04.29 | 2.6.32-21 |
| GVM12 | 1 | 1GB | Ubuntu12.04 | 2012.04.26 | 3.2.0-23 |
| GVM13 | 1 | 1GB | Ubuntu14.04 | 2014.04.17 | 3.13.0-24 |
| GVM21 | 1 | 1GB | Debian7.0.0 | 2013.05.06 | 3.2.0-4 |
| GVM22 | 1 | 1GB | Debian7.3.0 | 2013.12.16 | 3.2.0-4 |
| GVM23 | 1 | 1GB | Debian7.6.0 | 2014.07.14 | 3.2.0-4 |
| GVM31 | 1 | 1GB | CentOS6.3 | 2012.07.09 | 2.6.32-279 |
| GVM32 | 1 | 1GB | CentOS6.4 | 2013.03.09 | 2.6.32-358 |
| GVM33 | 1 | 1GB | CentOS6.5 | 2013.12.01 | 2.6.32-431 |
| GVM41 | 1 | 1GB | Fedora 16 | 2011.11.08 | 3.1.0-7 |
| GVM42 | 1 | 1GB | Fedora 17 | 2012.05.29 | 3.3.4-5 |
| GVM43 | 1 | 1GB | Fedora 18 | 2013.01.15 | 3.6.10-4 |
| GVM44 | 1 | 1GB | Fedora 19 | 2013.07.02 | 3.9.5-301 |

## 5.2　检测效果

实验以 adore-ng 所隐藏的信息为例进行隐藏检测. 它直接修改内核级函数,达到隐藏信息的目的. 图 3、图 4 分别是对 adore-ng 隐藏进程和隐藏文件的检测结果,左右侧窗口分别是 SVM 和 GVM.

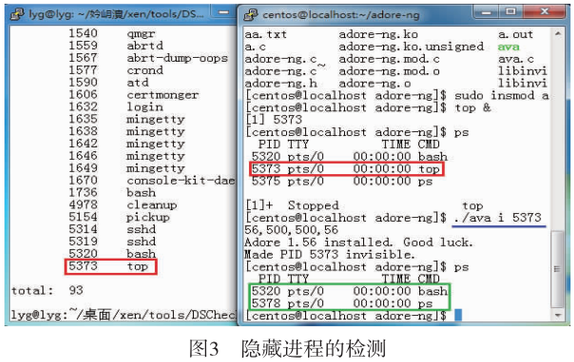

图3　隐藏进程的检测

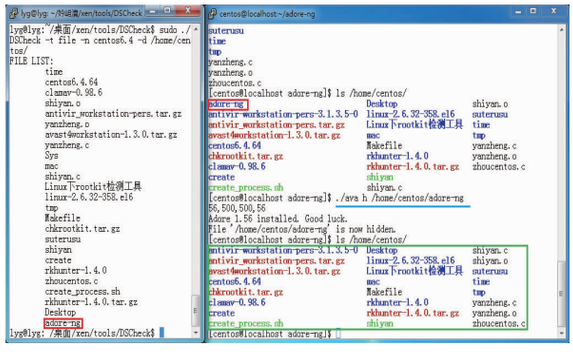

图4　隐藏文件的检测

Adore-ng 加载到 GVM 后,通过用户级应用程序 ava 就可控制隐藏动作. 要将 pid 号为 5373 的 *top* 进程隐藏,只需执行指令./ava *i* 5373;要将/home/centos 目录下的 adore-ng 文件夹隐藏,只需执行./ava *h* /home/centos/adore-ng. 通过检测结果可以发现,对 GVM 被 adore-ng 隐藏的对象都可由 Vlhd 检测到. 此外,Vlhd 还可以获得进程的代码段、数据段、映射库文件和执行路径等有用信息.

本文对多种 rootkit 进行了入侵检测,并与传统检测工具做对比,如表 2 所示. 对比发现,Vlhd 对 rootkit 的检测结果优于传统检测工具. 因为 Vlhd 处于 GVM 外部,避免了被恶意软件绕过的风险. Vlhd 通过在线模块(而非 GVM 的系统函数)提取出内核对象的内存原始数据,然后借助语义知识库解析内存数据,最终得到内核对象的真实语义视图. 传统的检测工具大都是基于主机的,导致与 GVM 处于相同的安全等级,因此二者之间缺乏足够的隔离性. 它们在检测过程中往往需要借助 GVM 的系统函数获取内核对象的语义. 若 GVM 的系统函数被篡改(例如,adore-ng 会修改系统函数 *getdents*())),那么它所得到的语义视图将会是虚假视图,因此容易被绕过.

**表 2　Rootkit 检测结果对比**

| | adore-ng | kbeast v1 | suterusu v1 | diamorphine | f00lkit | maK_it |
|---|---|---|---|---|---|---|
| Vlhd | √ | √ | √ | √ | √ | √ |
| Avast 1.3.0 | – | – | √ | – | – | √ |
| AVG 2013.3118 | √ | √ | – | √ | – | – |
| Avira 7.6.0.10 | √ | – | √ | – | – | √ |
| chkrootkit 0.5 | – | – | √ | – | – | – |
| ClamAV 0.98.3 | – | √ | – | √ | – | √ |
| F-PROT 6.2.39 | √ | – | – | – | – | – |
| rkhunter 1.4.2 | – | √ | – | – | – | – |

## 5.3　检测性能

### 5.3.1　执行速率

表 3 是 Vlhd 与其它传统检测工具的效率对比. 所有检测工具分别对 Ubuntu12.04-64 进行扫描,Vlhd 扫描一次约为 34ms,远小于其他工具. Vlhd 的效率优势源于两方面:一是 Vlhd 只关注扫描对象的物理内存,缩小了扫描范围;二是离线生成完备的语义知识库,当在线模块工作的时候向其提供就绪态的信息集,因此在线模块无需与操作系统和内核进行复杂的交互动作.

Vlhd 的执行效率不仅优于传统的检测工具,与当下应用范围最广的 libVMI 对比仍具有较大优势. 图 5 是 Vlhd 与 libVMI 对不同的操作系统的扫描时间对比. 通过对比可以发现,Vlhd 的扫描速率比 libVMI 快 2 ~ 3 倍. libVMI 在运行之前不仅需要针对特定 VM 加载 *findoffset. ko* 模块以获取语义信息,还需要提取特定 VM 中的 System. map 文件,将其复制到 SVM 的*/boot/*目录下. 其操作复杂,依赖于手动操作,与 GVM 之间存在强耦合性.

除 libVMI 之外,Vlhd 还与其它的外部检测工具进行了执行速度对比. 例如,SYRINGE 执行一次需要 57ms,VMST 仅转储进程列表中的 pid 号就需要 60ms,而 Virtuoso 获取 Linux 中的进程列表需耗时 6394.1ms. 繁杂的运行环节和频繁的在线交互操作是影响 SYRINGE 和 VMST

执行速率的关键因素. Virtuoso 在获取进程列表的过程中需要打开/proc 文件系统,并进行多次读取操作,因此耗时较多. 与上述方法相比,Vlhd 的在线监视过程仅包含内存提取和语义解析两大类操作,这大大简化了它的体系结构和执行环节. 此外,离线构建的语义知识库为 Vlhd 提供了所有语义信息,避险了频繁的在线交互. 所以,Vlhd 的执行速率优于其它技术.

**表 3 与传统检测工具的检测效率对比**

| 扫描工具 | 扫描时间 |
|---|---|
| Vlhd | 34ms |
| Avast 1.3.0 | 317ms |
| AVG 2013.3118 | 497ms |
| Avira 7.6.0.10 | 1s |
| chkrootkit 0.5 | 29.061s |
| ClamAV 0.98.3 | 7.014s |
| F-PROT 6.2.39 | 1s |
| rkhunter 1.4.2 | 24m15s |

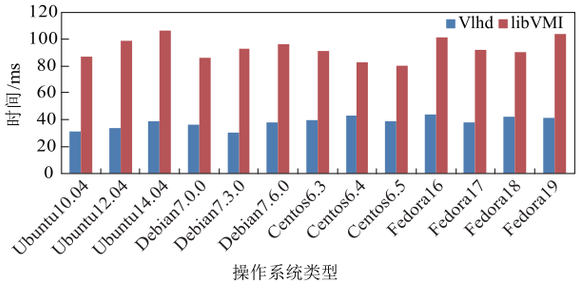


图5 与libVMI的扫描效率对比

图 6 是离线模块离线地建立语义知识库时所需的时间. 对于不同的操作系统,离线模块的运行时间不同,但都不超过 13s. 这是由于离线模块对内核的编译过程,仅针对包含特定数据结构的特定文件进行编译,而非针对整个内核,这大大减少了建库所需的时间.

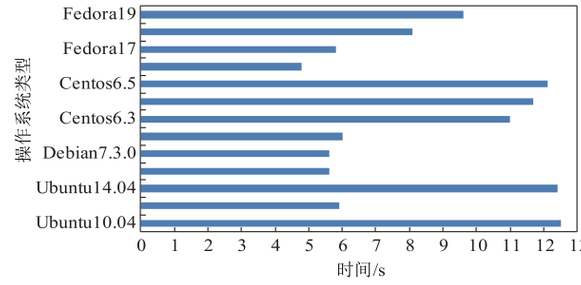


图6 构建语义知识库所需时间

### 5.3.2 通用性

除上述 13 种 OS 之外,我们对当下流行的 Linux 发行版本都进行了测试,测试结果表明 Vlhd 对各类 OS、不同内核版本均有效. 而且扫描过程无需任何人工干预就可完成,通用性较好.

### 5.3.3 性能开销

本实验中,采用 Linux 性能测试工具 nbench 测试 Vlhd 对 SVM 引入的性能开销. 测试过程中,首先测量 SVM 中不运行 Vlhd 时的性能. 然后将 Vlhd 设置为周期性扫描,扫描间隔分别设置为 1s、2s、4s、8s,分别测量扫描过程中 SVM 的性能. 表 4 中每列共有 10 项测试,每项测试表示每秒能够完成的测试个数,进行多次测试取平均值,数值越大表明处理速度越快. 从这 10 项测试中可以发现,Vlhd 的 4 种扫描周期对 SVM 分别引入最大 6.7%、3.4%、2.6% 和 1.1% 的性能开销. 而 SYRINGE 在以上四种扫描周期下分别引入了 8.3%、4.1%、3.5% 和 1.7% 的性能开销. 若换算成单次扫描的计算结果,那么 Vlhd 引入的性能开销仅有 0.03%,远低于 X-TIER 的 4.3%. 由于 Vlhd 建立了离线和在线两种模块,将语义提取从在线交互操作转化成离线操作,从而减少了在线模块对系统引入的性能开销.

**表 4 nbench 测量结果**

| | no Vlhd | 1(s) | 2(s) | 4(s) | 8(s) |
|---|---|---|---|---|---|
| NUMERIC SORT | 1057.6 | 986.7 | 1022.7 | 1037.51 | 1047 |
| STRING SORT | 631.04 | 589.31 | 609.58 | 616.53 | 625.36 |
| BITFIELD | 4.6568e+08 | 4.347e+08 | 4.5e+08 | 4.5357e+08 | 4.6079e+08 |
| FP EMULATION | 232.04 | 216.6 | 224.31 | 226.47 | 230.18 |
| FOURIER | 29649 | 29692 | 28673 | 29026 | 29322 |
| ASSIGNMENT | 35.289 | 32.946 | 34.138 | 34.583 | 34.982 |
| IDEA | 6914.3 | 64659.3 | 6680.6 | 6741.4 | 6856.2 |
| HUFFMAN | 2413.4 | 2251.8 | 2332.8 | 2355.5 | 2386.9 |
| NEURAL NET | 64.025 | 59.793 | 61.848 | 62.36 | 63.767 |
| LU DECOMPOSITION | 1704.7 | 1592.84 | 1653.9 | 1668.9 | 1690.5 |

## 6 结论

本文提出了一种基于语义鸿沟修复方法的 rootkit 隐藏对象检测技术 Vlhd. Vlhd 将系统分为离线和在线模块两部分,在线模块用于在线即时地语义重构;离线模块则首先离线地构建完备的语义知识库,之后向在

线模块提供即求即取的语义信息. 通过实验可以发现,Vlhd 能够检测到多种 rootkit 的入侵行为. 与当其它检测工具的对比结果表明,Vlhd 在检测效果与执行效率两方面都有较大的优势. 其单次扫描时间仅为 34ms,在扫描周期为 1s、2s、4s、8s 时,对 SVM 引入的性能开销分别为 6.7%、3.4%、2.6% 和 1.1%. 我们未来的研究工作重点将是入侵预警和处理,目标是建立起一套包括入侵检测、预警和处理的完整的安全体系.

## 参考文献


[1] Jain B, Baig M B, Zhang D, et al. Sok: Introspections on trust and the semantic gap[A]. IEEE Symposium on Security and Privacy[C]. IEEE, 2014. 605 – 620.

[2] Pearce M, Zeadally S, Hunt R. Virtualization: Issues, security threats, and solutions[J]. ACM Computing Surveys (CSUR), 2013, 45(2): 17.

[3] Laniepce S, Lacoste M, Kassi-Lahlou M, et al. Engineering intrusion prevention services for iaas clouds: The way of the hypervisor[A]. Service Oriented System Engineering (SOSE)[C]. IEEE, 2013. 25 – 36.

[4] Egele M, Scholte T, Kirda E, et al. A survey on automated dynamic malware-analysis techniques and tools[J]. ACM Computing Surveys (CSUR), 2012, 44(2): 6.

[5] Kim S, Park J, Lee K, et al. A brief survey on rootkit techniques in malicious codes[J]. Journal of Internet Services and Information Security, 2012, 3(4): 134 – 147.

[6] 俞能海,郝卓,徐甲甲,等. 云安全研究进展综述[J]. 电子学报, 2013, 41(2): 371 – 381.
YU Neng-hai, Hao Zhou, XU Jia-jia, et al. Review of cloud computing security[J]. Acta Electronica Sinica, 2013, 41(2): 371 – 381. (in Chinese)

[7] Vogl S, Kilic F, Schneider C, et al. X-tier: Kernel module injection[A]. Proceedings of the International Conference on Network and System Security[C]. Berlin, Heidelberg: Springer, 2013. 192 – 205.

[8] Carbone M, Conover M, Montague B, et al. Secure and robust monitoring of virtual machines through guest-assisted introspection[A]. International Workshop on Recent Advances in Intrusion Detection[C]. Berlin, Heidelberg: Springer, 2012. 22 – 41.

[9] 王丽娜,高汉军,刘 炜,等. 利用虚拟机监视器检测及管理隐藏进程[J]. 计算机研究与发展, 2011, 48(8): 1534 – 1541.
Wang L, Gao H, Wei L, et al. Detecting and Managing Hidden Process via Hypervisor[J]. Journal of Computer Research & Development, 2011, 48(8): 1534 – 1541. (in Chinese)

[10] Xiong H, Liu Z, Xu W, et al. Libvmi: a library for bridging the semantic gap between guest OS and VMM[A]. Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on[C]. IEEE, 2012. 549 – 556.

[11] Dolan-Gavitt B, Leek T, Zhivich M, et al. Virtuoso: Narrowing the semantic gap in virtual machine Introspection[J]. Security & Privacy IEEE Symposium on, 2011, 42(12): 297 – 312.

[12] Fu Y, Lin Z. Space Traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection[A]. Proceedings of the 2012 IEEE Symposium on Security and Privacy[C]. IEEE Computer Society, 2012. 586 – 600.

[13] 李勇钢,崔超远,李平. 基于虚拟机自省的客户机进程内容获取[J]. 计算机工程与设计, 2016, 37(6): 1697 – 1701.
Yong-Gang Li, Chao-Yuan Cui, et al. Get the process content of guest OS based on VMI[J]. Computer Engineering & Design, 2016, 37(6): 1697 – 1701. (in Chinese)

[14] Suneja S, Isci C, De Lara E, et al. Exploring VM introspection: techniques and trade-offs[J]. Acm Sigplan Notices, 2015, 50(7): 133 – 146.

[15] 崔超远,乌云,李平,等. 虚拟机自省中一种消除语义鸿沟的方法[J]. 通信学报, 2015, 36(8): 31 – 37.
CUI Chao-yuan, WU Yun, et al. Narrowing the semantic gap in virtual machine introspection[J]. Journal on Communications, 2015, 36(8): 31 – 37. (in Chinese)


## 作者简介

**李勇钢**　男,1988 年生于河北沧州,博士研究生. 研究方向为虚拟化、操作系统和信息安全.

**崔超远(通信作者)**　男,1972 年出生,博士,研究员. 主要研究方向为系统虚拟化、信息通信安全和大数据.
E-mail: cycui@iim.ac.cn

**乌　云**　女,1974 年出生,博士,副研究员. 主要研究方向为人工智能和机器学习.

**孙丙宇**　男,1974 年出生,博士,研究员. 主要研究方向为模式识别、智能装备和智能系统.