软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(11):2716−2728 [doi: 10.3724/SP.J.1001.2011.03923]

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

基于对象的软件行为模型*

傅建明[1,2,3+], 陶 芬[1,2], 王 丹[1,2], 张焕国[1,2,3]

[1](空间信息安全与可信计算教育部重点实验室(武汉大学),湖北 武汉 430072)

[2](武汉大学 计算机学院,湖北 武汉 430072)

[3](软件工程国家重点实验室(武汉大学),湖北 武汉 430072)

Software Behavior Model Based on System Objects

FU Jian-Ming[1,2,3+], TAO Fen[1,2], WANG Dan[1,2], ZHANG Huan-Guo[1,2,3]

[1](Key Laboratory of Aerospace Information Security and Trusted Computing of the Ministry of Education (Wuhan University), Wuhan 430072, China)

[2](School of Computer, Wuhan University, Wuhan 430072, China)

[3](State Key Laboratory of Software Engineering (Wuhan University), Wuhan 430072, China)

+ Corresponding author: E-mail: jmfu@whu.edu.cn

Fu JM, Tao F, Wang D, Zhang HG. Software behavior model based on system objects. *Journal of Software*, 2011,22(11):2716−2728. http://www.jos.org.cn/1000-9825/3923.htm

Abstract: On the basis of traditional FSA (finite state automaton), system objects can be resolved from the parameters of system call, and a Software Behavior model based on system object (SBO) is presented. This model defines the software state as all states of system objects, which are owned by the software, and then each state in the model has been assigned semantic information. Therefore, SBO can solve a problem of irrelevant semantics between different traces using the semantic information, and it can detect data semantic attacks, which directly or indirectly modifies system call parameters. Finally, a software anomaly intrusion detection prototype system based on SBO (SBOIDS) is implemented. The experimental and analysis results show that SBO can effectively detect data semantic attack and control the flow-based and mimicry attacks.

Key words: intrusion detection; software behavior; finite state automaton; system object; system call

摘 要: 以传统有限自动机(finite state automata,简称 FSA)为基础,从系统调用参数中解析出系统对象,提出了一种基于系统对象的软件行为模型(model of software behavior based on system objects,简称 SBO).该模型的行为状态由软件所关联的所有系统对象表示,从而赋予状态的语义信息,解决了不同行为迹中 PC(program counter)值的语义不相关问题;同时,该模型可以对抗系统调用参数的直接和间接修改,从而可以检测基于数据语义的攻击.最后,实现了基于 SBO 的软件异常检测原型工具(intrusion detection prototype system based on SBO,简称 SBOIDS),其实验和分析结果表明,该模型可以有效地检测基于控制流的攻击、模仿攻击以及针对数据语义的攻击,并给出了该工具的性能开销.

关键词: 入侵检测;软件行为;有限状态自动机;系统对象;系统调用

* 基金项目: 国家自然科学基金(90718005); 国家高技术研究发展计划(863)(2007AA01Z411)
收稿时间: 2010-02-03; 修改时间: 2010-06-09; 定稿时间: 2010-07-28

中图法分类号: TP311 文献标识码: A

软件(或程序)的行为是指软件运行的表现形态和状态演变的过程,可以从底层的二进制指令到高层的程序语句、函数、系统调用等不同层次刻画软件行为.软件的行为模型就是指根据某一层次的行为信息而构建的行为状态序列以及状态变迁,可以表征软件的正常行为特征,并用于软件行为的异常检测[1].软件行为模型会直接影响到异常检测系统的误报率/漏报率和实时检测的性能,一直是主机异常入侵检测领域的研究热点.同时,因为系统调用是程序访问系统资源的接口,作为操作系统对应用程序提供的访问接口,系统调用序列在一定程度上能够反映程序的行为特征,因此,很多学者从系统调用这一层次来研究软件的行为特征.

一般把软件行为建模方法分为静态建模和动态建模.静态建模[2−4]直接分析程序的源代码或二进制代码,提取出程序的系统调用等相关行为信息并建立行为模型.该方法在理论上可以提取程序的所有可能的执行路径,但实际上,静态行为获取技术往往受到一些因素的限制[5],如:(1) 版权保护使得获取程序的源代码比较困难;(2) 程序的行为往往依赖于操作系统版本、环境设置、用户输入等因素;(3) 具有“自修改”特性的程序逐渐增多;(4) 很多软件使用了抗逆向分析技术,如代码混淆、加密和变形、多态等.动态建模是指采用动态训练的方式,通过对程序进行大量良性执行,监控并记录在不同执行情况下的系统调用等行为信息并生成行为模型[5−12].动态建模可以获取程序的实际执行信息,可以克服静态建模中限制因素(1)、因素(3)、因素(4),但仍然受制于因素(2),即模型的精确性依赖于训练时的外部输入、执行环境以及训练数据的覆盖率,无法捕获到程序所有可能的执行路径.从实际应用来看,动态建模方法比静态建模要简单、实用[1],因此本文仅限于研究动态建模.

目前,动态建模可以捕获系统调用,但对参数的处理和状态设置存在两个限制.一个限制是模型对参数的直接或者间接修改敏感,如修改文件对象的句柄值(称为直接修改,即调用参数修改),或修改文件对象句柄指向的文件(称为间接修改,类似于指针对象修改).程序运行的交错时序、并发运行产生的竞争条件(race condition)等将会导致:(a) 相同的对象句柄值实际代表着不同的系统对象;(b) 不同的对象句柄值实际上代表着同一系统对象.另一个限制是许多 FSA 模型[8]采用 PC 值刻画状态机的状态.PC 值可以刻画同一个系统调用的多次出现和不同系统调用在程序的位置,但 PC 值仅仅是一个位置信息,没有确切的语义信息,而且无法处理不同行为迹中状态的合并.这两个限制增加了动态建模的困难.

本文从系统对象出发,试图突破这两个限制.首先从系统调用参数中解析出系统资源对象,从而克服参数修改的敏感;然后采用程序所关联的系统对象作为模型的状态,从而赋予状态实际的语义,即系统资源的子集;最后提出了一种基于系统对象的行为模型,该模型可以方便地处理不同行为迹的状态合并,也适合离线建模.

本文第 1 节给出基于对象的软件行为模型的定义.第 2 节给出基于 SBO(model of software behavior based on system objects)模型的软件异常检测方法,并分析其检测能力.第 3 节描述基于 SBO 模型的软件异常检测原型工具(intrusion detection prototype system based on SBO,简称 SBOIDS).第 4 节是实验结果及评估.第 5 节是国内外相关研究介绍.第 6 节是全文的总结.

1 SBO 模型的建立

常用的操作系统(如 Windows)体系结构分为用户态和核心态.一般的应用软件位于用户态,需要通过系统调用访问系统资源.而 Windows 操作系统提供的系统资源大部分都是以对象的形式存在的,因为对象是对系统资源的封装[13].应用程序只有通过调用系统服务函数来访问这些对象,才能与操作系统交互并实现其功能.因此,软件对系统资源/对象的访问可以从本质上刻画软件的行为.本文以程序在运行过程中所访问的资源对象为切入点,构建软件的行为特征.

1.1 模型定义

在定义 SBO 模型之前,先给出如下基本元素的定义.

定义 1(ObjectType). 表示系统对象类型集合.Windows 操作系统的文档[13]定义了 15 种可以由 Windows

API 访问的对象类型:符号链接(symbolic link)对象、进程(process)对象、线程(thread)对象、作业(job)对象、内存区(section)对象、文件(file)对象、访问令牌(access token)对象、事件(event)对象、信号量(semaphore)对象、互斥体(mutex)对象、定时器(timer)对象、I/O 完成(IoCompletion)对象、注册表键(key)对象、窗口站(windows station)对象、桌面(desktop)对象.

定义 2(Ostate). 表示对象状态.对象状态在程序运行过程中是不断变迁的,从对象被创建或打开到对象被关闭或删除.不同类型的对象其对象状态各不相同,例如,Process 对象具有“running”状态,而 File 对象具有“Read”状态.对象状态的变化是由对象操作引起的,因此采用对象操作信息来描述对象的状态信息,状态信息由“状态名|状态描述”表示.其中,状态名和当前对象操作的操作名是一一对应的.比如对于读取文件操作,其引起的文件对象状态的状态名定义为“Read”.“状态描述”和对象操作的具体操作信息是一一对应的,比如,操作信息“Read|ByteOffset_Length”表示从 ByteOffset 的偏移位置读 Length 个字节.

定义 3(Object). 表示系统对象集合.$object \in Object$,为三元组($name,type,ostatelist$),表示对象类型的实例,是某一个具体的系统资源,其中,$name$ 表示对象名称,唯一标识一个对象.$type \in ObjectType$,表示对象类型.$ostatelist=\langle ostate_1,ostate_2,ostate_3,\ldots,ostate_n\rangle$,表示对象在程序执行过程中的状态变迁序列.

Windows 操作系统用对象句柄来访问对象,而且对象句柄仅在当前进程虚拟地址空间中有效.因此,可以用对象名称来唯一标识一个对象.如果获得的对象名称为空,则按照预定的命名规则为对象命名[**].

定义 4(ObjectOperation). 表示对象操作集合.$oop \in ObjectOperation$,为一个四元组($pid,opid,name,objectlist$).其中,pid 为对象操作所属进程的标识符,用于区分不同的进程操作;$opid$ 为对象操作的编号,用于唯一标识每个对象操作.$opid$ 由 16 位二进制整数表示,其中,高两位为标志位,用来表示对象操作的类型;低 14 位用来对操作进行编号.最高位标志位 $change$=1 表示该操作能改变对象状态(例如 CreateFile,WriteFile 等),否则不能(例如,ReadFile,QueryInformationFile 等).次高位标志位 $close$=1 表示该操作为“关闭类”操作,即该操作将结束对象的生命周期,将对象从程序的当前对象列表中删除(例如 Close,DeleteFile,DeleteKey 等);$name$ 为对象操作的名称;$objectlist=\{object_1,object_2,object_3,\ldots,object_m\}$为操作的系统对象集合.

定义 5(oop sequence(oops)). $\langle oop_1,oop_2,oop_3,\ldots,oop_n\rangle$,$oop_i \in ObjectOperation$,$1 \leqslant i \leqslant n$,表示程序在执行过程中的对象操作序列.

定义 6(Edge). 表示转变边的集合.$edge \in Edge$,为四元组($opid,name,objectlist,tostateno$).在 SBO 模型中,$edge$ 表示一个状态转换,状态转换只能由对象操作触发,即 $Edge$ 是由 $ObjectOperation$ 映射而来,与对象操作一一对应.其中,$opid,name,objectlist$ 的含义与 $ObjectOperation$ 中的完全一致,$toStateno$ 为该边所转向的状态的状态编号 $stateno$.当对象操作不改变程序状态时($change$=0),则对应的转换边的转向状态即为当前状态自身.

定义 7(Path). 表示转换路径集合.转换路径 $path=\langle edge_1,edge_2,edge_3,\ldots,edge_n\rangle$,定义为程序的一次执行轨迹中每个状态的所有转换边的有序集合.程序的一次执行轨迹对应着状态的一条转换路径,该路径包含状态的所有转换边(转向状态自身的转换边以及转向其他状态的转换边),路径中转换边的顺序就是该执行轨迹中各个边的执行顺序.程序在不同的行为迹中执行流程可能并不相同,从而导致状态的转换路径也不相同.第 2.2 节给出了详细的实例分析.

定义 8(State). 表示程序状态集合.$state \in State$,为三元组{$stateno,objectlist,pathlist$},表示程序运行状态.其中:$stateno$ 为状态编号,唯一标识 SBO 模型中一个独立的状态;$objectlist=\{object_1,object_2,object_3,\ldots,object_n\}$.其中,$n$ 表示当前程序所拥有的对象的个数,$object_i$($1 \leqslant i \leqslant n$)表示第 i 个对象;$pathlist=\{path_1,path_2,path_3,\ldots,path_m\}$,表示该状态所拥有的所有转换路径集合.其中,$path_j$($1 \leqslant j \leqslant m$)表示第 j 个转换路径.

在此基础上,SBO 模型的定义如下:

定义 9(SBO 模型). 为四元组(S,T,S_0,S_F),其中,

** Section 对象和 Thread 对象的对象名称往往为空.为了唯一标识该类对象,设定命名规则为:“对象所属于的进程名”+“对象类型”+“对象编号”,如“Test.exe-Section-0001”表示 Test.exe 进程的第 1 个 Section 对象.

- $S=\{s|s\in State,s$ 为程序运行时的状态$\}$,描述程序运行状态的集合;
- $S_0\subseteq S$,表示初始状态的集合;
- $S_F\subseteq S$,表示结束状态的集合;
- $T=\{e|e\in Edge,$为状态转换$\}$,表示状态转换的集合.

1.2 构建算法

采用动态训练的方式建立 SBO 模型.首先,捕获程序在执行过程中的系统调用;然后,分析系统调用参数关联的系统资源/对象,将系统调用轨迹转化为对象操作轨迹,从而将不同的系统调用通过操作同一系统对象而关联起来;最后,根据对象操作轨迹建立有限状态自动机模型.

对象流分析就是指获取每个系统调用所操作的具体系统对象以及对象的状态描述,生成对应的对象操作.操作系统中的系统调用有很多,如 Windows XP 的 SSDT(系统服务描述表)有 284 个系统服务函数,但通过分析发现,只有 219 个函数会触发系统资源访问.对目前主流杀毒软件关注的系统对象类型以及相应的系统调用函数进行了统计分析,发现杀毒软件关注的对象类型集中在 *Process*,*Thread*,*Section*,*Key*,*File* 等 5 类对象,因为这些对象是应用程序经常访问的,也关联了应用程序的基本功能.因此,本文仅关注这 5 类最关键的系统对象.

对象操作是通过对系统调用进行对象解析获得的.例如,对于系统调用函数 *NtWriteFile*(*FileHandle*,*Event*,*ApcRoutine*,*ApcContext*,*IoStatusBlock*,*Buffer*,*Length*,*ByteOffset*,*Key*),首先通过对 *FileHandle* 进行解析获得文件名称,然后对 *Buffer*,*Length*,*ByteOffset* 参数解析获取得到缓冲区地址、写入的长度以及写入的偏移地址,最后生成的对象操作为"*pid*,*opid*,*WiteFile*,{*filename*,*File*,*Write*|*Buffer_Length_ByteOffset*}",其中,*filename* 为文件名称,"*File*"表示对象类型为文件类型.为了便于状态查询和快速建模,根据对象操作是否会改变对象状态,把对象操作分为改变对象状态和不改变对象状态两类.

定义 10(**CloseOp**). "关闭类"操作,可以关闭或删除某一对象的操作.当操作的次高位标志位 *close*=1 时,表示操作为 *CloseOp* 类操作.

在关注的对象操作中,有 5 个 *CloseOp* 操作:*TerminateProcess*(终止进程对象的执行)、*TerminateThread*(终止线程对象的执行)、*DeleteFile*(内核态的删除文件对象操作)、*DeleteKey*(删除注册表键对象)、*Close*(可以关闭 *Process*,*Thread*,*File*,*Key*,*Section* 等 5 类对象).*Close* 操作是最常用的"关闭类"操作,该操作将关闭其参数句柄所表示的对象.关闭类操作也可以删除对象,比如,关闭 *Process* 和 *Thread* 对象表示该对象从当前程序状态中删除,而 *File* 对象和 *Key* 对象则可以直接删除.表 1 给出了 *Process* 对象的对象操作.同理,可以给出其他 4 种类型的对象操作表.

根据对象流分析获取了程序执行过程中的对象操作轨迹之后,按照 BSBO 算法训练生成 SBO 模型,具体算法介绍如下:

BSBO 算法. Algorithm of building SBO model (BSBO). /**建立 SBO 模型**/

Input: oop sequence: $\{oops_1,oops_2,oops_3,\ldots,oops_n\}$. /*操作序列集合,$oops_i$ 表示第 i 个操作序列*/

Output: SBO. /*基于对象的软件行为模型*/

Procedure:

(1) *Init*(S_0) /*构造 SBO 模型的初始状态 S_0,S_0 包含当前进程对象和当前主线程对象*/

(2) $SBO\rightarrow AddState(S_0)$; i=1; j=1; /*将 S_0 添加进 SBO 中,初始化完成之后 SBO 模型中仅有 S_0 状态;*/

(3) While ($i\leqslant n$) /*多次训练,i 表示当前单次训练的操作序列的编号,n 为待训练操作序列总数*/

(4) { /*对每个操作序列进行单次训练*/

4.1) $CurrentState=SBO\rightarrow S_0$; /*将当前状态 *CurrentState* 初始化为上次训练生成的 SBO 模型的 S_0 状态*/

/*训练过程:依次处理 $oops_i$ 中的每个对象操作*/

4.2) while($j\leqslant oops_i\rightarrow oopSum$) /*$j$ 表示操作序列中的操作编号,$oops_i\rightarrow oopSum$ 表示 $oops_i$ 中的 oop 个数*/

4.3) {

4.3.1) *theObjectOperation*=$oops_i \rightarrow oop_j$; /*取 $oops_i$ 的第 *j* 个对象操作*/

4.3.2) *CreateNextState*(*CurrentState*,*theObjectOperation*,*NextState*); /*根据当前状态和操作产生下一个状态,注意该操作类型是否为关闭类*/

4.3.3) If (!*SBO*→*FindState*(*NextState*)) /*查找指定状态是否存在于当前 SBO 中*/

4.3.4) *SBO*→*AddState*(*NextState*); /*如果不在,将产生新状态添加到 SBO 中*/

4.3.5) *theEdge*=*new Edge*(*theObjectOperation*); /*根据操作生成对应的边*/

4.3.6) *theEdge*→*tostateno*=*NextState*→*Stateno*; /*设当前边指向下一个状态*/

4.3.7) if (!*CurrentState*→*FindEdge*(*theEdge*)) /*查找状态的 *pathlist* 中是否存在该边*/

4.3.8) *CurrentState*→*AddEdge*(*theEdge*); /*如果不存在,添加 *edge* 到状态的 *pathlist* 中*/

4.3.9) *CurrentState*=*NextState*; /*设置 *CurrentState* 为 *theEdge* 指向的下一个状态*/

4.3.10) *j*++; /*操作编号数加 1*/

4.4) }

4.5) *i*++; *j*=1;

(5) }

Table 1 Object operations of *Process* object

表 1 *Process* 对象的对象操作

Object type	Object operation	Operation type (change|close)	Ostate	Description
Process	*CreateProcess*	1|0	*Create*	Create *Process*
	OpenProcess	1|0	*Open*	Open *Process*
	ResumeProcess	1|0	*Resume*	Resume *Process*
	SuspendProcess	1|0	*Suspend*	Suspend *Process*
	ReadVirtualMemory	0|0	*ReadVirtualMemory*| *BaseAddress_* *BufferLength*	*BaseAddress*—The base address of the virtual memory to read. *BufferLength*—The number of bytes to read.
	WriteVirtualMemory	1|0	*WriteVirtualMemory*| *BaseAddress_* *BufferLength*	*BaseAddress*—The base address of the virtual memory to write. *BufferLength*—The number of bytes to write.
	LockVirtualMemory	1|0	*lockVirtualMemory*| *BaseAddress_* *LockLength_* *LockType*	*BaseAddress*—The base address of the virtual memory to be locked. *LockLength*—The number of bytes to be locked. *LockType*—The type of locking to be performed.
	UnlockVirtualMemory	1|0	*UnlockVirtualMemory*| *BaseAddress_* *LockLength_* *LockType*	*BaseAddress*—The base address of the virtual memory to be unlocked. *LockLength*—The number of bytes to be unlocked. *LockType*—The type of unlocking to be performed.
	AllocateVirtualMemory	1|0	*AllocateVirtualMemory*| *BaseAddress_* *AllocationSize_* *Protect*	*BaseAddress*—The base address of the allocated virtual memory. *AllocationSize*—The number of bytes to allocate. *Protect*—Specifies the protection for the pages.
	FreeVirtualMemory	1|0	*FreeVirtualMemory*| *BaseAddress_* *FreeSize*	*BaseAddress*—The base address of the virtual memory to be freed. *FreeSize*—The number of bytes to free.
	ProtectVirtualMemory	1|0	*ProtectVirtualMemory*| *BaseAddres_* *ProtectSize_* *NewProtec_* *OldProtect*	*BaseAddress*—The base address of the virtual memory to be protected. *ProtectSize*—The number of bytes to protected. *NewProtect*—The new access protection. *OldProtect*—The previous access protection.
	TerminateProcess	1|1	*Terminate*	Terminate *Process*

BSBO 算法的基本过程是:首先初始化 SBO 模型,构造 SBO 模型的初始状态 S_0(S_0 包含当前进程对象和当前主线程对象),添加 S_0 到 SBO 模型中,见步骤(1),初始化后的 SBO 模型仅有 1 个状态 S_0;然后,从对象操作序列集合中取一个对象操作序列作为当前待训练的操作序列.对每个操作序列的训练过程是(单次训练):首先,初始化当前状态 *CurrentState*(当前状态为当前正在处理的状态,其值随训练过程变化)为前一次训练产生的 SBO 模型的初始状态 S_0;然后,依次取当前操作序列的一个对象操作,将对象操作转换成对应的转换边 *edge*,根据当前状态和边生成下一个状态 *NextState*,见步骤(4.6.2);然后判断 *NextState* 是否已经存在于 SBO 模型中,见步骤(4.6.3),如果不存在,则将其添加到 SBO 模型的状态集合中,若存在,则将 *NextState* 更新为 SBO 模型中已经存在的 *NextState* 状态;然后,设置 *edge* 指向 *NextState*,并判断 *edge* 是否已经存在于 *CurrentState* 的转换路径集合 *pathlist* 中,见步骤(4.6.6),如果不存在,则将 *edge* 添加到 *CurrentState* 的 *pathlist* 中;最后,更新 *CurrentState* 为 *NextState*.然后取下一个对象操作,依次循环,直至处理完当前操作序列的所有对象操作.多次训练的过程,即是依次取出对象操作序列集合中的每个操作序列,按照上述单次训练过程对每个操作序列进行训练,直至生成最后的 SBO 模型.

本文给出了用 BSBO 算法构建 SBO 模型的一个实例(例 1).

例 1:*Test.c*.

```
void main(int argc,char* argv[])
{
    OFSTRUCT of; DWORD writesize, readsize;
    char fname1[]="c:\\1.txt"; char fname2[]="c:\\2.txt";
    char buf[100]={0}; BOOL rt; HANDLE pfile1, pfile2;
    pfile1=(HANDLE)OpenFile(fname1,& of,OF_READWRITE);
    if (pfile1)
    {
        rt=ReadFile(pfile1,buf,sizeof(buf ),& readsize,NULL);
        if (rt)
        {   pfile2=(HANDLE)OpenFile(fname2,& of,OF_READWRITE);
            if (pfile2)
            {
                rt=WriteFile(pfile2,buf,strlen(buf ),& writesize,NULL);
                CloseHandle(pfile2);
            }
        }
    CloseHandle(pfile1);
}}
```

图 1 是对例 1 编译执行后的可执行程序利用算法 BSBO 构建的 SBO 模型,其初始状态为 S_0,终止状态为 S_F.表 2 给出了转换边的详细说明,表 3 给出了状态的详细说明.

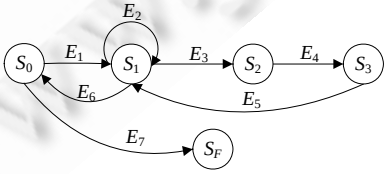

Fig.1 Example 1 of SBO

图 1 SBO 模型示例 1

Table 2 Edges of Fig.1

表 2 图 1 的边

Edge	Description of edge
E_1	1824,32884,*OpenFile*,(*C*:\1.*txt*,*File*,*Open*)
E_2	1824,183,*ReadFile*,(*C*:\1.*txt*,*File*,)
E_3	1824,32884,*OpenFile*,(*C*:\2.*txt*,*File*,*Open*)
E_4	1824,33042,*WriteFile*,(*C*:\2.*txt*,*File*,*WriteFile_end*_100*B*)
E_5	1824,49177,*Close*,1,(*C*:\2.*txt*,*File*,*Close*)
E_6	1824,49177,*Close*,1,(*C*:\1.*txt*,*File*,*Close*)
E_7	1824,33025,*TerminateProcess*,(*C*:*Test.exe*,*Process*,*Terminate*)

Table 3 States of Fig.1

表 3 图 1 的状态

State	Description of state
S_0	*objectlist*: *C*:*Test.exe*,*Process*,*Running*; *pathlist*: ⟨E_1,E_7⟩, ⟨E_7⟩
S_1	*objectlist*: *C*:*Test.exe*,*Process*,*Running*; *C*:\1.*txt*,*File*,*Open*; *pathlist*: ⟨E_2,E_3,E_6⟩, ⟨E_2,E_6⟩, ⟨E_6⟩
S_2	*objectlist*: *C*:*Test.exe*,*Process*,*Running*; *C*:\1.*txt*,*File*,*Open*; *C*:\2.*txt*,*File*,*Open*; *pathlist*: ⟨E_4⟩
S_3	*objectlist*: *C*:*Test.exe*,*Process*,*Running*; *C*:\1.*txt*,*File*,*Open*; *C*:\2.*txt*,*File*,*Open*|*WriteFile_end*_100*B*; *pathlist*: ⟨E_5⟩
S_F	*objectlist*: *NULL* *pathlist*: *NULL*

2 基于 SBO 的软件异常检测

从软件良性执行的多个行为迹中构建 SBO 模型之后,就可以利用该模型检测软件异常.检测系统在启动被检测程序的同时加载该程序的 SBO 模型,将当前状态设为 SBO 模型中的初始状态,然后监控运行过程中产生的每个对象操作.一旦监控到程序调用一个对象操作,则可以根据其 SBO 模型判断该对象操作是否异常.

DASBO 算法. Algorithm of Detection Abnormity based on SBO(DASBO) /*基于 SBO 的异常检测算法*/

Input: SBO; /*SBO 模型*/

oop sequence: {oop_1,oop_2,oop_3,…,oop_n}. /*待检测操作序列*/

Output: *True*/*False*. /**True* 表示存在异常,*False* 表示正常*/

Procedure:

(1) *CurrentState=SBO*→S_0; *i*=1; /*设当前状态为输入的 SBO 模型的初始状态*/

(2) While (*i*≤*n*) /**i* 为对象操作编号*/

(3) {

3.1) *theObjectOperation*=oop_i; /*取第 *i* 个对象操作*/

3.2) *theEdge=new Edge*(*theObjectOperation*); /*根据操作生成对应的边*/

3.3) if (!(*CurrentState*→*FindEdge*(*theEdge*))) /*如果该边不存在于当前状态的转换路径集合中*/

3.4) return *True*; /*表示发生异常*/

3.5) else /*该边存在于当前状态的转换路径集合中*/

3.6) {

3.6.1) *ulStateno*=(*CurrentState*→*FindEdge*(*theEdge*))→*tostateno*; /*找到该边转向的状态号*/

3.6.2) *CurrentState=SBO*→*FindStatebyno*(*ulStateno*); /*通过状态号找到状态,设 *CurrentState* 为转向后的状态*/

3.7) }

3.8) *i*++; /*对象操作编号增 1*/

(4) }

(5) return False; /*表示正常*/

CurrentState 最初设为 SBO 的初始状态,然后,随着 oop 的触发不断更新.其中,检测的关键函数是 *FindEdge* 函数,该函数将判断当前对象操作对应的 *theEdge* 是否属于当前状态的合法转换边,如果不属于则检测到异常.本文定义以下 3 种异常类型:

a) 不可能路径异常:表示当前操作将导致程序状态进入不可能转换路径,即当前操作与待比较边的 *opid* 不同.

b) 操作对象异常:表示当前操作属于当前状态的合法路径,即当前操作与待比较边的 *opid* 相同,但操作的对象不同.

c) 操作对象状态异常:表示当前操作属于当前状态的合法路径,并且操作的是同一对象,但对象的状态链表 *ostatelist* 不同.比如,一个操作从文件结尾处写入 500 字节内容,另一个操作是从该文件对象的开始处写入 500 字节内容.

针对文献[11]中的攻击方法,本文逐一分析 SBO 模型对这些攻击的检测能力.

2.1 代码注入攻击

代码注入攻击[11]是指攻击者本地或者远程向进程空间注入一段可执行的二进制代码,然后通过某种手段修改进程的正常控制流程,使注入代码获得控制权.如果攻击者为了完成某一功能调用系统调用访问某些系统资源,这些系统资源即为本文中定义的系统对象,则采用 SBO 模型可以检测到这些额外的或者修改的系统对象访问.而且注入的 *shellcode* 不同,其触发的异常也不同.如果注入的攻击代码包含有对象操作,并且对象操作的 *opid* 与当前程序状态的当前转换边的 *opid* 不同,将会触发 a)类异常;如果对象操作的 *opid* 与当前转换边的 *opid* 相同,但操作的对象不同,则会触发 b)类异常;如果对象操作的 *opid* 以及操作的对象都相同,但对象状态不同,则会触发 c)类异常.如果注入的攻击代码不含有对象操作,则不会触发异常.

2.2 不可能路径攻击

不可能路径[8]是指模型中存在而进程执行中不可能出现的路径.传统的 FSA 模型[8]依赖于程序 PC 值来刻画程序状态,其不可能路径是指函数返回到不同的另一个同名调用点.虽然 SBO 模型并不依赖于 PC 值建模,但 SBO 仍然属于有限自动机模型,故仍然需要考虑不可能路径情况.本模型通过引入状态的转换路径来检测不可能路径,对每个状态而言并不直接将其所有转换边表示成一个单一的边集合,而是通过不同的转换路径来存储同一条路径上的有序边集合.例如,图 1 中的 SBO 模型的 S_1 状态具有 3 条转换边 E_2,E_3,E_6,这 3 条转换边将组成 3 个不同的转换路径$\langle E_2,E_3,E_6\rangle$,$\langle E_2,E_6\rangle$,$\langle E_6\rangle$.这 3 条转换路径都是合法的,当程序处于 S_1 状态时,可以转入 3 条路径之中的任何一条,但每条路径内的转换边之间是有序的,如$\langle E_2,E_6\rangle$这条路径要求程序必须先执行 E_2(*ReadFile* 操作),然后才能执行 E_6(*Close* 操作).通过在训练时将每个状态的所有合法转换路径都存储在该状态的 *pathlist* 表中,如果当前操作不满足任何一条转换路径,就会被检测为异常,从而有效抵抗不可能路径攻击.例如,从图 1 中的 SBO 模型中可以构造出一条路径$\langle E_1,E_3,E_4,E_5,E_6,E_7\rangle$.该路径中的每条边都属于 SBO 的边集合,但在 S_1 状态下不存在$\langle E_3\rangle$这一转换路径,则该路径将被检测为异常.总之,这类攻击将触发 a)类异常.

2.3 模仿攻击

模仿攻击[14]是指攻击者利用合法的系统调用序列,但是可以通过巧妙地修改调用参数以达到攻击的目的.文献[14]中给出了一个模仿攻击的实例:

fd=*open*(/*tmp*/*file.tmp*,…);…*read*(*fd*,…);…*write*(*fd*,…);…*close*(*fd*);

攻击者可能修改 *open* 调用的参数并且不改变序列顺序获得 *passwd* 文件的内容:

fd=*open*(/*etc*/*passwd*,…);…*read*(*fd*,…);…*write*(1,…);…*close*(*fd*);

对于传统的 FSA 模型以及没有系统调用参数分析的模型而言,这些模型忽略了调用参数,因此不能检测模仿攻击.SBO 模型采用数据流分析技术,可以检测该攻击.例如,上述实例中两个序列中 *open* 操作的对象不同,必然会触发 b)类异常.

2.4 针对数据语义的攻击

对于传统的数据流异常检测而言,其关注的是数据值本身,而没有关注数据值所代表的语义.

例如,*NtWriteFile* 的第 1 个参数为 *FileHandle*.在传统数据流分析中,可以获得 *FileHandle* 的 32 位数据值,但可能没有获知该 *FileHandle* 具体指向的文件对象.

对象句柄的攻击可能使得“相同的对象句柄值实际代表着不同的系统对象”和“不同的对象句柄值可能确代表着同一系统对象”,一般将这类攻击称为针对数据语义的攻击.对于这类攻击,因为数据的值没有发生任何改变,传统的数据流分析可能无法检测到攻击.SBO 模型可以有效检测出这类攻击,将触发 b)类异常.

对于此类攻击,采用与第 2.3 节相似的攻击实例,假设原始的系统调用序列如下:

fhandle=OpenFile(“*c*:\1.*txt*”,…); *ReadFile*(*fhandle*,…); *WriteFile*(*fhandle*,…); *CloseHandle*(*fhandle*);

针对该序列建立的 SBO 模型如图 2 所示.

编写攻击程序截获程序对 *NtWriteFile* 函数的调用,劫持控制权到新的 *NtWriteFile* 函数地址.这样,每当应用程序调用 *NtWriteFile* 函数时都会执行攻击者编写的新的 *NtWriteFile*.该攻击可以将原始 *NtWriteFile* 函数的 *FileHandle* 参数所代表的文件对象(“*c*:\1.*txt*”文件)修改为新的文件对象(“*c*:\ 2.*txt*”文件),从而 *FileHandle* 指向了新的文件.对于应用程序而言,看似正常地执行了写文件操作,其程序执行流程没有任何改变.但是攻击者通过改变了参数所指向的对象,导致程序并没有写“*c*:\1.*txt*”文件,而是写了另一个文件“*c*:\2.*txt*”.基于控制流和传统数据流的检测方法都无法检测出这种攻击,而 SBO 模型可以有效地捕捉到这种数据语义的改变,将会触发 b)类异常,具体如图 2 和图 3 所示.

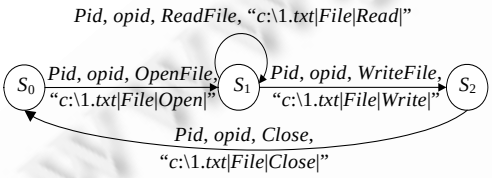

Fig.2 SBO of a mimicry attack example

图 2 模仿攻击实例的 SBO 模型

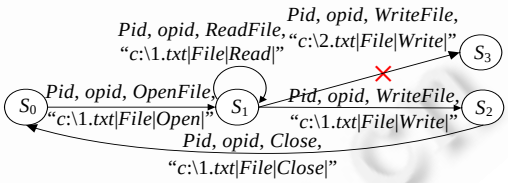

Fig.3 Detection of data semantic attack

图 3 数据语义攻击的检测

3 基于 SBO 的软件异常检测原型工具

在 Windows 平台下设计了基于 SBO 模型的软件异常检测原型系统(简称 SBOIDS).图 4 是 SBOIDS 系统框架图.该软件异常检测系统分为 4 个主要的模块:监控模块、训练模块、检测模块以及预警模块.

监控模块负责跟踪记录被检测程序在执行过程中的对象操作轨迹,可以根据用户的选择决定继续跟踪还是终止执行.在训练模式下,监控模块主要负责跟踪记录下被监控程序的对象操作轨迹,并将对象操作轨迹信息写入对象操作日志文件中,该日志文件用于训练模块进行训练生成 SBO 模型.在检测模式下,监控模块不再将从内核态获取到的对象操作信息写入日志文件,而是直接将实时的对象操作输入到检测模块进行检测.

用户态的用户监控子模块负责启动内核跟踪子模块以及根据用户选择将相应的控制命令传送给内核跟踪模块.内核跟踪子模块以内核驱动的方式运行,负责跟踪记录程序在执行过程中的对象操作信息.该系统采用 SSDT Hook 技术实现内核态的跟踪.

训练模块根据对象操作日志文件训练生成表示程序正常执行的 SBO 模型.训练模块和监控模块一起完成对程序正常行为的建模工作,通常需要对程序进行多次执行,记录程序每次执行的对象操作轨迹,从而提高 SBO 模型的精度.

检测模块完成在线检测的功能,并把检测结果传递给预警模块,最后由预警模块反馈给监控模块.

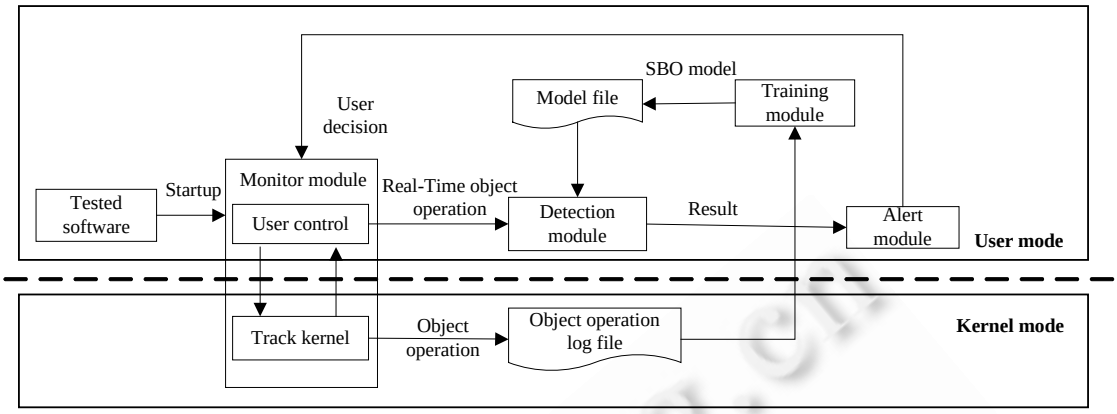

Fig.4 Framework of SBOIDS system

图 4 SBOIDS 系统框架

4 实验结果与评估

本文在 Windows XP 平台下实现了 SBOIDS 系统,并分析 SBO 模型的检测能力以及性能开销.实验的硬件环境为 CPU:Genuine Intel(R),内存:1G,硬盘:150G.软件环境为 Microsoft Windows XP Professional,内核版本为 5.1.2600.

4.1 检测能力

为了评估 SBO 模型的检测能力,选取了被广泛使用的 AdobeReader 8.0 软件作为测试对象.针对该软件的攻击样本,其检测结果见表 4.

表 4 所示实验结果表明,SBO 模型能够有效地检测基于栈溢出、堆溢出的代码注入类型攻击.因为在攻击样本中,攻击者调用了一些额外的系统调用来对系统资源进行访问.这些额外的系统对象访问并不属于当前程序状态的合法对象访问,将会触发 a)类异常.采用 SBO 模型可以准确地检测此类攻击,其误报率和漏报率都是 0.

对于本文第 2 节中提到的不可能路径攻击、模仿攻击以及针对数据语义的攻击,这 3 类攻击在理论上是存在的,但构造起来十分复杂,并且需要软件具有相应可利用的软件漏洞.我们目前还没有这 3 类攻击样本.

Table 4 Results of detection attacks using DASBO

表 4 使用 DASBO 检测攻击结果

Vulnerability	Type of attack	Description	Result
CVE-2008-2992	Stack-Based buffer overflow attack	Javascript printf stack overflow	√
CVE-2008-2549	Denial of service (application crash)	Crash and possible code execution	√
CVE-2009-0927	Stack-Based buffer overflow attack	*getIcon*() stack overflow	√
CVE-2009-0928	Heap-Based buffer overflow attack	JBIG2 encoded stream heap overflow	√
CVE-2009-0658	Heap-Based buffer overflow attack	Crafted JBIG2 streams heap overflow	√

4.2 性能开销

为了测试实时检测的性能开销,本文选取了 Windows 平台下 3 款常用的文档阅读软件:AdobeReader(7.0), FoxitReader(1.3),Microsoft Office Word(2003)作为测试对象.本文通过将软件在正常运行和在检测系统监控下运行的空间、时间开销进行对比分析,评估该系统的性能开销.

本文的检测系统实时运行时间开销主要来自于监控模块和检测模块.为了有效地评估时间开销,本文将检测系统的时间开销分为 3 个部分:对象操作截获开销、模型训练开销、异常检测开销,对这 3 部分开销进行分

别评估.其中,由于记录的对象信息详细,导致模型训练耗费时间较长,例如,对于 AdobeReader,训练 100 个样本将花费约 40 分钟时间.为此,本文采用离线训练,在检测实时性能时对模型训练开销不作考虑.测试结果见表 5.

Table 5 Performance overhead of SBOIDS

表 5 SBOIDS 的性能开销

Target program	Space overhead of normal (KB)	Space overhead with SBOIDS (KB)	Time overhead of normal (ms)	Time overhead with SBOIDS (ms)	Time overhead of object operation interception (ms)	Time overhead of anomaly detection (ms)
AdobeReader	15 836	35 808	8 000	14 250	5 000	1 250
WORD	11 236	79 288	2 500	8 600	2 800	3 300
FoxitReader	5 808	24 500	4 000	6 160	1 500	660

表 5 中,对于不同的应用程序,由检测系统所导致的空间开销并不相同.这主要是因为在检测时需要导入应用程序的 SBO 模型,而不同的应用程序的 SBO 模型的规模各异.由于 SBO 模型会对程序在执行过程中所有的对象操作详细信息进行建模,导致模型规模较大,这也是导致空间开销较大的主要原因.对于时间开销,从表 5 中可以看出,被检测程序在检测系统下监控运行的时间约是原来正常运行时间的 1.5 倍~3 倍左右.时间开销的主要原因在于对象操作的截获,其约占整个时间开销的 30%~40%.由对象操作截获引起的开销取决于使用的截获机制.与用户层截获机制相比,内核态截获机制引起的时间开销更低.本文采用的是内核态截获对象操作机制,在一定程度上降低了总体时间开销.

5 相关研究工作

典型的动态模型有 N-gram 模型[6]、V-gram 模型[7]、FSA 模型[8]、Vt-path 模型[9]和 Execution Graph(执行图)模型[5]等.N-gram 模型和 V-gram 模型仅对系统调用序列建模,没有涉及调用参数.FSA 模型[6]是一种以 PC 值为状态、以系统调用为变迁的 FSA 行为模型,解决序列分割、*-gram(包括 N-gram 和 V-gram)模型空间复杂性高等问题,但存在不可能路径.为此,Feng[7]随后提出了 Vt-Path 模型.该模型利用程序执行时的返回地址、PC 值、系统调用号,剔除不可能路径.执行图[10]是利用系统调用号、函数调用地址以及系统调用的返回地址建立执行图模型.

混合模型是指结合了静态分析和动态分析技术建立的行为模型,如 Dyck 模型[10]和 HFA 模型[11]等.Dyck 模型首先静态分析二进制代码,然后采用动态的“压制”技术来删除多余的 null calls 操作.Giffin[15]在 Dyck 模型基础上考虑环境变量,使得检测攻击的能力更强.HFA 模型将静态分析中不可能获得间接函数调用和跳转的目的地址这一无法解决的问题推迟,在运行过程中利用对函数局部自动机的动态链接和运行时修正解决这一问题,从而可以达到更高的精确度和效率.

动态模型是沿着两个分支发展的.一类是基于系统调用的控制流模型,包括*-gram 模型、FSA 模型、Vt-Path 模型和 Execution 模型等,其行为信息包含系统调用号、系统调用名称,部分模型考虑了系统调用的返回值,没有考虑系统调用的参数.这类模型能够检测出改变程序控制流的入侵,例如,代码注入类型攻击[11].Wanger 等人[14]提出了针对控制流模型的模仿攻击,从而直接暴露了控制流模型的缺陷.另一类是基于系统调用的数据流模型,其行为信息另外还包括系统调用参数和返回值,刻画各个系统调用的参数之间、参数与返回值之间的数据关联关系.Kruegel[16]获取系统调用的参数中字符串的长度和字符分布规律.Sufatrio[17]研究首先根据用户指定的分类规则将单个系统调用的参数进行分类,然后对每类参数建立相应的取值规则.Tandon[18]研究通过学习机制建立系统调用参数的值集规则.Bhatkar[19]提出了基于数据流的异常检测,并在系统调用参数及其返回值之间建立了一元和二元的行为规则.数据流模型基本上是以控制流为基础,把数据关联关系映射到控制流的边上,从而更加准确刻画软件行为.

文献[12]首先从控制流获取系统调用模式,然后根据数据流在模式内和模式间挖掘系统调用参数和返回值中的行为关联规则.该方法可以解决 PC 值随行为迹变化的问题.本文在已有研究的基础上[12]提出了基于系统对象的软件行为模型(SBO).该模型的行为状态由软件所关联的所有系统对象表示,从而赋予状态机中状态的

语义信息.区别于已有的模型,该模型不是以 PC 值作为状态,而是以系统对象作为行为状态.

6 结束语

本文从系统调用的参数值挖掘出系统调用对系统资源的影响(包括创建、打开、修改、删除、关闭等),从而刻画程序行为的语义.SBO 模型从系统调用参数中解析出系统资源对象,从而克服参数修改的敏感.因此,SBO 模型不仅可以检测模仿攻击,而且还可以检测基于数据语义的攻击,使得模型的检测攻击能力更强.与其他模型相比,该模型以系统资源的状态为中心,而不是以程序执行的 PC 值为中心.因此,该模型需要获取更详细的程序运行时信息,导致建模时间过多.由此带来的好处是模型的精度提高,可以检测数据流的攻击.

SBO 模型是一种动态模型,其完备性依赖于训练时的外部输入、执行环境以及训练数据的充足性,动态模型往往无法捕获到程序所有可能的执行路径,模型的泛化是将来的研究工作.另外,如何把非系统对象的软件行为融入 SBO 模型也是下一步的研究内容.

References:

[1] Tao F, Yin ZY, Fu JM. Software behavior model based on system calls. Computer Science, 2010,37(4):151−157 (in Chinese with English abstract).

[2] Wagner D, Dean D. Intrusion detection via static analysis. In: Proc. of the IEEE Symp. on Security and Privacy. Oakland: IEEE Press, 2001. 156−168.

[3] Su PR, Yang Y. Intrusion detection model based on executable static analysis. Chinese Journal of Computers, 2006,29(9): 1572−1578 (in Chinese with English abstract).

[4] Peng GJ, Pan XC, Fu JM, Zhang HG. Static extracting method of software intended behavior based on API functions invoking. Wuhan University Journal of Natural Sciences, 2008,13(5):615−620. [doi: 10.1007/s11859-008-0521-6]

[5] Gao DB, Reiter MK, Song D. Gray-Box extraction of execution graphs for anomaly detection. In: Proc. of the ACM Conf. on Computer and Communications Security. Washington, 2004. 318−329. [doi: 10.1145/1030083.1030126]

[6] Hofmeyr SA, Forrest S, Somayaji A. Intrusion detection using sequences of system calls. Journal of Computer Security, 1998,6(3): 151−180.

[7] Wepsi A, Dacier M, Debar H. Intrusion detection using variable-length audit trail patterns. In: Proc. of the 3rd Int'l Workshop on Recent Advances in Intrusion Detection. London, 2000. 110−129.

[8] Sekar R, Bendre M, Dhurjati D, Bollineni P. A fast automaton-based method for detecting anomalous program behaviors. In: Proc. of the IEEE Symp. on Security and Privacy. Oakland: IEEE Press, 2001. 144−155. [doi: 10.1109/SECPRI.2001.924295]

[9] Feng HH, Kolesnikov OM, Fogla P, Lee W, Gong WB. Anomaly detection using call stack information. In: Proc. of the 2003 IEEE Symp. on Security and Privacy. Oakland: IEEE Press, 2003. 62−75. [doi: 10.1109/SECPRI.2003.1199328]

[10] Giffin JT, Jha S, Miller BP. Efficient context-sensitive intrusion detection. In: Proc. of the 11th Network and Distributed Systems Security Symp. San Diego, 2004.

[11] Li W, Dai YX, Lian YF, Feng PH. Context sensitive host-based IDS using hybrid automaton. Journal of Software, 2009,20(1): 138−151 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3165.htm [doi: 10.3724/SP.J.1001.2009.03165]

[12] Li P, Park H, Gao DB, Fu JM. Bridging the gap between data-flow and control-flow analysis for anomaly detection. In: Proc. of the 2008 Annual Computer Security Applications Conf. Las Vegas, 2008. 392−401. [doi: 10.1109/ACSAC.2008.17]

[13] Russinovich ME, Solomon DA, Wrote; Pan AM, Trans. Microsoft Windows Internals (4th ed.). Beijing: Publishing House of Electronics Industry, 2007 (in Chinese).

[14] Wagner D, Soto P. Mimicry attacks on host based intrusion detection systems. In: Proc. of the 9th ACM Conf. on Computer and Communications Security. Washington, 2002. 255−264. [doi: 10.1145/586110.586145]

[15] Giffin JT, Dagon D, Jha S, Lee W, Miller BP. Environment-Sensitive intrusion detection. In: Proc. of the 8th Int'l Symp. on Recent Advances in Intrusion Detection. Seattle, 2005.

[16] Kruegel C, Mutz D, Valeur F, Vigna G. On the detection of anomalous system call arguments. In: Proc. of the ESORICS. Gjovik, 2003. 101−118. [doi: 10.1007/978-3-540-39650-5_19]

[17] Sufatrio, Yap RHC. Improving host-based ids with argument abstraction to prevent mimicry attacks. In: Proc. of the 8th Int'l Symp. on Recent Advances in Intrusion Detection. Seattle, 2005. [doi: 10.1007/11663812_8]

[18] Tandon G, Chan P. Learning rules from system calls arguments and sequences for anomaly detection. In: Proc. of the ICDM Workshop on DataMining for Computer Security. Melbourne, 2003. 20−29.

[19] Bhatkar S, Chaturvedi A, Sekar R. Dataflow anomaly detection. In: Proc. of the 2006 IEEE Symp. on Security and Privacy. Oakland, 2006. 48−62. [doi: 10.1109/SP.2006.12]

附中文参考文献:

[1] 陶芬,尹芷仪,傅建明.基于系统调用的软件行为模型.计算机科学,2010,37(4):151−157.

[3] 苏璞睿,杨轶.基于可执行文件静态分析的入侵检测模型.计算机学报,2006,29(9):1572−1578.

[11] 李闻,戴英侠,连一峰,冯萍慧,基于混杂模型的上下文相关主机入侵检测系统.软件学报,2009,20(1):138−151. http://www.jos.org.cn/1000-9825/3165.htm [doi: 10.3724/SP.J.1001.2009.03165]

[13] Russinovich ME, Solomon DA,著;潘爱民,译.深入解析 Windows 操作系统(第 4 版).北京:电子工业出版社,2007.

傅建明(1969－),男,湖南宁乡人,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件行为,恶意代码,网络安全.

陶芬(1986－),女,硕士,主要研究领域为软件安全,网络安全.

王丹(1989－),女,主要研究领域为软件安全,网络安全.

张焕国(1945－),男,教授,博士生导师,CCF 高级会员,主要研究领域为密码学,可信计算,信息安全.