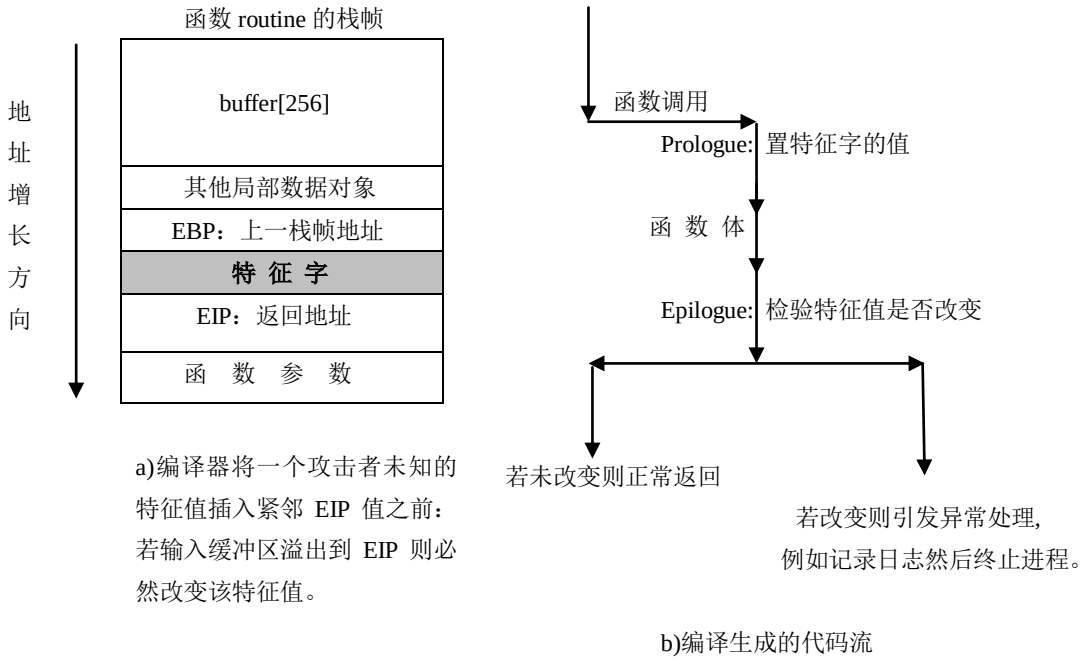
第四章 反入侵技术(I): 基于主机的机制

这一章和下一章阐述反入侵技术。我们首先针对前一章揭示出的具体的入侵机制，讨论几类针对性很强的反入侵技术(4.1-4.2 节)。虽然这类技术对具体入侵机制的反制很有效，但毕竟合法系统难以预测攻击者究竟会以什么机制实施入侵，攻击者也肯定不会总遵循固定的规则行事，因此需要更具有普遍性的解决方案，这就引导到对入侵检测系统的讨论(4.3-4.6 节)。

4.1 栈的一致性保护技术

针对 3.1 节的分析，容易给出一个简单的解决方案：在编译程序的过程中，在每个函数的栈帧中位于“返回地址”项之前插入一个攻击者无法预测、但其值对合法系统本身是已知的特征字，如图 4-1 (a)所示。显然，超界的输入串要能达到改写“返回地址”项的位置，一定会改变特征字的值(注意合法的特征值对攻击者保密)，于是每当函数返回之前只要检查该特征字的值是否改变，就可以判定当前是否发生了栈溢出入侵，如图 4-1 (b)。



a)编译器将一个攻击者未知的特征值插入紧邻 EIP 值之前：若输入缓冲区溢出到 EIP 则必然改变该特征值。

b)编译生成的代码流

图 4-1 *StackGuard* 保护机制

以上过程很容易由编译生成的前导代码和清场代码自动完成(参考第 3 章注释 3)。这一解决方案要求改变编译器，这对编译器开发商和 GNU C++/C 这样的开放源代码编译器是不

难做到的，对程序运行期间增加的额外代价也微不足道，是一个针对性很强的解决方案。这就是著名的 *StackGuard* 栈一致性保护技术，目前许多著名的编译器如 VC++都有实现。

可以将 *StackGuard* 的思想扩展到解决单字节溢出入侵(图 4-2)，方法是使编译器将一个攻击者未知的特征值插入到紧邻“上一栈帧地址”项之前：若输入缓冲区溢出到“上一栈帧地址”项或“返回地址”项则必然改变该特征值。此外，编译器总将输入缓冲区置于最高地址(例如图 4-2 中的 buffer[])，以避免其他局部变量因溢出而被篡改。这就是著名的 *ProPilice* 栈一致性保护技术，也已经在包括 VC++在内的许多编译器采用。

注意要使用 *StackGuard* 或 *ProPilice* 保护已经交付的程序，则只能重新编译。

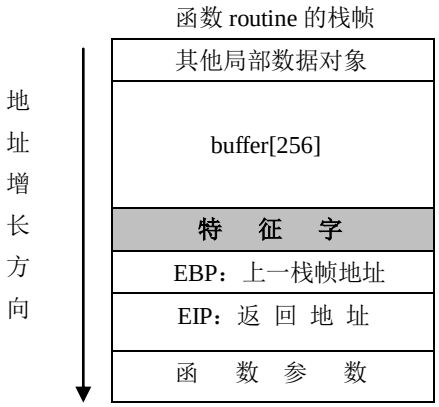


图 4-2 *Propilice* 保护机制

还有其他抗栈溢出措施。我们注意到(回顾操作系统知识)，在正常情况下一个进程的栈是用于暂存函数调用的上下文，例如栈帧这类数据，从来不会被用于保存指令代码，换句话说，正常的程序决不会在栈上执行一段指令，而这恰恰是栈溢出攻击时所发生的情况。实际上，操作系统为进程分配任何页面时(页面是操作系统分配内存空间的基本单元，操作系统总是按页面的整数倍而非实际字节数分配空间)，每个页面是有严格的存取属性的，这些属性包括可读(PG_READ)、可写(PG_WRITE)、可执行(PG_EXEC)，而作为数据页面被分配给栈空间的页面应该具有(但实际上没有被设置)不可执行这一属性(!PG_EXEC)。

因此，一个简单(但需要修改操作系统内核)的解决方案就是使操作系统在为栈空间分配页面时关闭页面的 PG_EXEC 属性，从而禁止从栈空间执行(病毒)程序代码。

一种不必修改操作系统的用户模态机制是在运行时对函数入口进行检查保护，主要保护对象之一就是最常用的标准 C 函数库。已经开发出了 *Linux* 的(用户模态)共享库 *LibSafe*，它针对某些常用的 C 库函数具体检验该函数当前是否有过长的、以至覆盖当前栈帧的“上一栈帧地址”项和“返回地址”项的输入参数。图 4-3 描述了这一保护过程。

LibSafe 针对的典型库函数包括 *memcpy*(), *strcpy*(), *strncpy*(), *strcat*(), *strncat*(), *sprintf*(), *vprintf*(), *vfprintf*(), *gets*(), *realpath*()等。

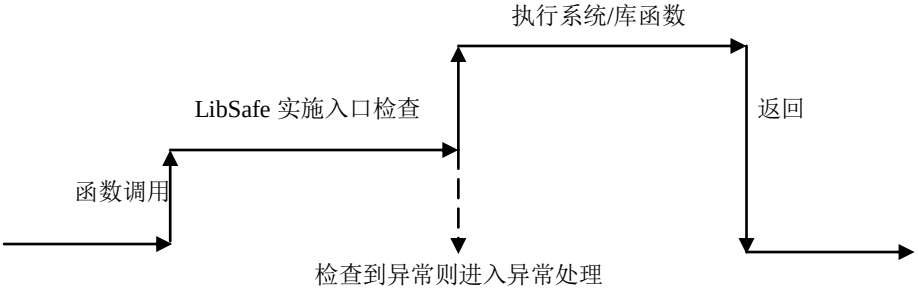


图 4-3 对函数入口进行检查

4.2 代码注入检测技术

经验分析表明大量病毒在入侵时最常调用的 API 仅集中于较少数的一族，例如 Windows 上的系统函数 *GetProcessAddress*(), *GetModuleHandle*(), *LoadLibrary*(), *CreateThread*(), *CreateProcess*(), *listen*(), *send*(), *sendto*(), *connect*(), *CreateFile*(), *CereateAccount*()等。因此，若在这类 API 中添加病毒检测代码，在其被调用时先检测某些入侵特征或条件，例如若检测到当前的调用来自栈空间则表明该调用实际上来自病毒程序，以此判定该系统调用是否应该继续执行。这就是代码注入检测的基本思想。

代码注入思想的具体实现属 Symantec 公司[1]的专利技术，在运行时将反病毒代码勾挂(Hook)到以上那些最常被病毒调用的 API 函数，以此在运行时提供反入侵保护。

4.3 入侵检测系统

从现在起我们转而阐述一类更普遍适用的解决方案，这类方案并不针对某种特定的入侵机制实施防卫，而是识别出最广泛的入侵行为，这就是入侵检测系统(*Intrusion Detection System: IDS*)。

[1] 如果从最广泛的意义上观察和分析，许多事物或现象并不象它们表面上表现的那样单纯的“好”或“坏”，例如令人深恶痛绝的盗窃行为在另一方面却促使防盗器材成为当今很大一项产业，为此，该产业的受益者对盗窃的看法肯定与很多盗窃的受害者完全不同。类似地(也是很不幸的)，计算机病毒和入侵现象也是如此，它在造成巨大损失的同时也促使反病毒和反入侵技术发展成为当代 IT 领域非常活跃的分支产业之一，其每年的营业额与利润令人印象深刻而且还在快速上升之中。Symantec 公司就是这一领域的领袖开发商之一，另一个当之无愧的巨人是 Kaspersky 公司。

IDS 实现为软件，按照其目的划分为两类：第一类是所谓基于主机的 IDS(*Host-based IDS: HIDS*)，这类软件运行于个人计算机、工作站和服务器，所保护的对象仅仅涉及其运行环境中的系统本身，目的在于识别出针对这一系统的、来自外部的入侵行为；第二类是所谓基于网络的 IDS(*Network-based IDS:NIDS*)，这类软件运行于网络设备，例如防火墙、有安全功能的路由器/交换机，也可以是起边界网关作用的普通计算机，但这类 IDS 的目的主要不是保护其运行的系统本身，而是保护网络或网络中的一个区域，这一区域与其它区域之间的所有信息(IP 分组)都要流经该 NIDS 所运行的设备，因此 NIDS 软件有机会通过分析所有这些 IP 分组而检测出是否存在针对所保护网络区域中的对象(包括主机、进程等)的入侵企图。从基本检测方式上讲，HIDS 主要通过分析本机上的各个进程的行为来判定当前是否存在入侵，NIDS 则主要通过分析所观测到的 IP 分组的数据特征来判定当前是否存在入侵。

目前 HIDS 和 NIDS 都有广泛应用，而且不难看出这两类 IDS 具有相互补充的关系，都属当前最重要的反入侵技术之一。本章阐述 HIDS，下一章阐述 NIDS。

这里需要指出的是，单纯从概念上讲，仅仅依靠检测来对抗入侵是不够的，在检测出入侵现象之后还应该自动实施保护或反制行为，检测与保护的智能化结合才构成最完整意义上的反入侵系统，这就是入侵保护系统(*Intrusion Protection System: IPS*)的概念。然而，目前的技术能力还达不到使入侵检测与智能保护两者能合理地集成，实际应用多以自动检测为主，对识别出的入侵事件加以详细记录，而进一步的保护机制可能需要人机交互来解决，而非完全由软件自动、实时地决策。因此，我们在这里也主要以入侵检测为主要内容进行阐述。

实现 IDS 的具体技术途径有多种，有些涉及当代计算机软件甚至硬件领域的前沿，例如数据挖掘、人工智能等。但总的来讲，可以将 IDS(无论 HIDS 还是 NIDS)的检测方法分为两类：第一类是所谓基于误用模型的方法(*misuse-based*)，这类方法首先建立各种入侵行为的模型和特征，IDS 在实际运行期间识别进程的行为(HIDS)或 IP 分组的特征(NIDS)是否符合预先建立的入侵模型，如果符合则判定为入侵事件；第二类是所谓基于反常或基于规范的方法(*anomaly-based/specification-based*)，这类方法首先建立被保护的对象的正确的行为模型和特征(即系统规范)，IDS 在运行期间识别进程的实际行为(HIDS)或 IP 分组的实际特征(NIDS)是否偏离预先建立的规范，如果偏离则判定为入侵事件。换句话说，第一类 IDS 所依据的是关于各类恶意入侵机制的知识，实施检测的关键在于如何判定实际情况与已知的模式是否“符合”或“匹配”，匹配即意味着入侵；而第二类 IDS 所依据的是各类正确行为的知识，实施检测的关键在于如何判定实际情况与预期情况是否“偏离”，偏离即意味着入侵。如何将以上概念定量化并实现为性能合理的检测算法，是两类 IDS 实现的关键。

不难想象，IDS 实施其检测算法的速度是影响其性能的重要因素之一，一个速度明显低于被保护对象典型速度的 IDS 是不能令人满意的。IDS 的实际运行速度又取决于许多因素，包括 IDS 的检测机制、模型的复杂程度、IDS 的软件体系结构设计以及与检测精度的平衡折衷等。这里要强调的是，任何 IDS 都很难达到完美检测：对基于误用模型的 IDS，总可能遇到其未知特征的实际入侵事件，这时该类 IDS 将出现漏报；对基于反常或规范方法的 IDS，总可能遇到未被其规范完全覆盖的实际正常行为，这时该类 IDS 将出现虚警。虽然对特定的检测方法可以在理论上分析其漏报或虚警概率，但这方面内容不在本书讨论，对读者而言，重要的是理解 IDS 的实际运行效果取决于对诸如速度、算法精度、漏报或虚警概率这些因素的恰当的平衡与折衷。

以上描述了关于 IDS 的一些普遍概念，接下来具体描述几个典型的 HIDS 实例，使读者对 IDS 的系统设计与工作机理有更具体的理解。

4.4 HIDS 实例: 基于进程的系统调用模型的 HIDS

这一节描述 *Wagner* 和 *Dean* 最近开发的一个 HIDS，完整、详细的描述见本章后列举的文献。这是一个典型的基于规范方法的 HIDS，具体思路是：进程[2]行为可以由其程序所允许的、运行期间的函数调用序列的形态完全刻画，如果实际调用序列偏离所允许的调用序列就意味着出现异常行为，即入侵。这里的“函数”指操作系统的编程接口，例如用 C 语言定义的 POSIX 或 Win32 系统调用函数。回顾第二章所描述的各种病毒行为不难看出，伴随入侵而来的几乎总是运行恶意的、出于攻击者特殊目的的程序片段，很难想象这些病毒程序片段所表现的行为特征会与原始(诚实)的程序行为几乎一致，更符合现实的情况应该是两者明显地不一致。因此，以上解决问题的思路确实具有现实意义。问题在于：如何有效获取—最好是自动获取—程序所允许的函数调用序列、如何表达这些合法的函数调用序列、如何判定实际的函数调用序列偏离合法的函数调用序列？

4. 4. 1 基于有限状态自动机的检测模型

在展开讨论之前，我们先描述一个关于程序行为的简单模型。虽然这一模型不足以用于现实的入侵检测，但可以用来解释系统的设计思想。把一个(被保护的)程序看做系统调用

[2] 从现在我们遵循操作系统课程中的概念约定，每当指静态性质时，采用“程序”一词，而讨论运行时的(动态)行为时采用“进程”一词，以使概念明确。

函数的集合，如果事先已知该程序所包含的所有系统调用函数(要获取这一信息并不困难，例如通过扫描源程序或程序目标代码中出现的函数名)，譬如一个 Windows 程序的这种合法函数集合中不包含 API setEvent[3]，那么，一旦在实际运行中检测到该进程请求调用函数 setEvent(要做到这一点也很容易，例如截获进程对操作系统的函数调用并识别出函数标识号即可)，这毫无疑问意味着一个非法行为，而引起这一非法调用的主体只能是已经入侵该进程的病毒程序。

这一方法的技术实现虽然简单，但不足以完成有实际意义的入侵检测，原因在于这里用来描述程序行为的模型过于“粗糙”。举一个例子，如果已知一个程序的合法的系统调用集合包含 open、read、write、close 这些 API，实际运行中一次观测到该进程以顺序 open、read、 read 访问同一个文件，另一次观测到以顺序 read、open、read 的顺序访问同一个文件，按照这一模型，两种情况都属于合法事件，但实际上有编程经验的读者很容易看出后一种情况明显反常：一个正常的程序行为不应该在打开文件对象之前就试图从中读取数据。读者也不难看出，之所以会出现这类漏警，是因为以上这个过于简单的程序模型完全没有考虑函数调用顺序这一因素，而调用顺序当然应该是程序行为的重要特征之一，而且由程序本身决定。

如果既考虑被保护程序调用哪些系统函数、又考虑程序对这些函数的调用顺序，一个有用的模型是有限状态机。我们以一个简单的例子来解释这类模型的思想以及如何用于入侵检测。

看下面这个 Unix/Linux 上的 C 程序片段：

```
int x;
…….
f(int y){
      x>y ? getuid() : geteuid();
}
g(void){
      fd = open(“A”, O_RDONLY);
      f(0); /*调用函数 f */
      close(fd);
      f(1);
      exit(0);
}
```

程序中的 getuid、geteuid、open、close、exit 都是系统调用函数(分别用来读取进程当前

[3] 这实际上是一个很有用的 Win32 API, 将一个命名的事件对象置为“有信号”的状态。

的用户标识、有效用户标识、打开文件、关闭文件、终止进程)，f 和 g 是两个应用程序自定义的函数。我们用一张有向图来表达这一程序片段，称为调用关系图(*callgraph*)，其中每个自定义函数用两个节点表示，分别代表该函数的进入状态和离出状态，例如函数 f 对应的两个进入节点和离出节点分别是 Entry(f)和 Exit(f)；对系统函数的调用表达为事件，每经过一次特定的事件(系统函数调用)，程序转移到另一个状态，用一个带唯一命名的圆圈标识。此外，用实线表示程序经系统函数调用所发生的状态转移，而当调用某个自定义函数时，用无事件(空事件)的虚线表示。例如，自定义函数 g 进入后先调用系统函数 open，从状态 Entry(g)转移到 v，然后调用另一个自定义函数 f，表达为一条从状态 v 转移到状态 Entry(f)的虚线。对以上程序片段，这样构造出的完整的调用关系图如图 4-4，注意这张图可以看做是两个自定义函数 f 和 g 的调用关系图联结而成。

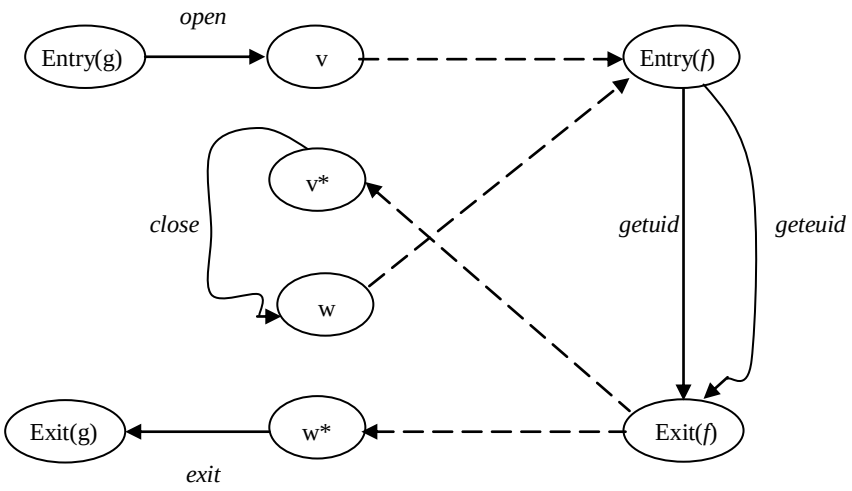


图 4-4 示例程序的调用关系图

把调用关系图中的节点解释为状态、边解释为引发状态转移的事件，由此调用关系图本质上是一个非确定性的、有限状态自动机模型(NDFA)。之所以说该模型是非确定性的，是因为从一个状态出发，即使经同一个事件也可能转移到不同的状态，具体取决于运行时的条件[4]。

从以上描述不难归纳出一个算法，通过分析源程序，遍可以构造出该程序的调用关系图，即调用关系图可以自动生成(类似于编译中的控制流分析技术)。程序在运行期间可能出现的函数调用序列都表现为其调用关系图中的某一条路经，例如若运行时 x 的初值为 0，则进程

[4] 学习过高等离散数学或编译理论的读者请回忆：确定性自动机模型和带空动作的、非确定性自动机模型等价，换句话说,一类机器的行为可以由另一类机器完全模拟。就能够接受的输入字的集合结构来讲，两者都接受所谓正规集合。

中出现的调用序列是 Entry(g)→v→Entry(f)→Exit(f)→v*→w→Entry(f)→Exit(f)→w*。根据调用关系图不难实现一个状态跟踪算法，在运行期间跟踪该进程的状态路径，每当出现不属于该调用关系图所蕴涵的路径，例如检测到转移 Exit(f)→Entry(f) 或 Exit(f)→v*→w→Entry(g)，都意味着发现了一个反常行为，即入侵。

然而这一方法存在一个问题：虽然调用关系图蕴涵了表达程序正常行为的全部转移路径，但并非刚好如此。存在这样一类路径，它们实际上不对应进程的任何可能出现的状态转移。例如图 4-5 中粗线所描绘的路径，就不对应示例程序中任何可能的函数调用序列(为什么？)。这意味着基于调用关系图实施入侵检测将存在漏警的可能：一旦病毒入侵导致进程的行为沿着(例如)图 4-5 中的粗线路径转移，则 IDS 将完全识别不出当前存在入侵！

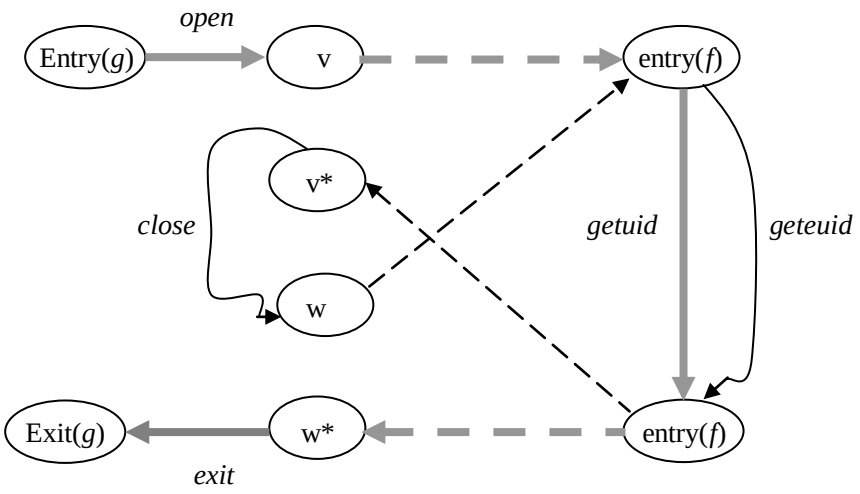


图 4-5 示例程序调用关系图中的例外路径(粗线)

因此，调用关系图所表达的程序行为模型也嫌粗糙。实际上，这是一切有限状态机模型的内在表达能力所决定的。我们知道，程序语言(从而进程的行为)本质上属于由下推自动机模型(PDM)的输出所刻画的范畴，因此进程行为应该由等价的上下文无关文法(CFG)来准确描述。

4.4.2 基于下推自动机的检测模型与实现

沿袭上一小节的基本思想，同时在技术上采用上下文无关文法(等价于下推自动机模型)精确表达进程的行为模型，以此实施对进程反常行为的检测，这就是 *Wagner* IDS 的设计思想。这里具体阐述其实现方法。

仍然以上一节的示例程序片段为例，读者不难验证下面的语法确实精确刻画了该程序所

蕴涵的函数调用序列：

Entry(*f*) ::= getuid() Exit(*f*) | geteuid() Exit(*f*)
Exit(*f*) ::= ε
Entry(*g*) ::= *open*() *v*
v ::= Entry(*f*) *v**
*v** ::= *close*() *w*
w ::= Entry(*f*) *w**
*w** ::= *exit*() Exit(*g*)
Exit(*g*) ::= ε

以上这一模型是以上下文无关语法的产生式的形式来表达的[5]，竖线“|”表示对具有相同的左侧符号的多个产生式的缩写，例如 Entry(*f*) ::= getuid()Exit(*f*) | geteuid()Exit(*f*) 实际上表示两个产生式 Entry(*f*) ::= getuid() Exit(*f*) 和 Entry(*f*) ::= geteuid() Exit(*f*)。ε 表示空字，符号 getuid()、geteuid()、*open*()、*close*()、*exit*()是语法常元(或称终极符)，仅出现在产生式的右面；Entry(*f*)、Exit(*f*)、Entry(*g*)、Exit(*g*)、*v*、*v**、*w*、*w**是语法变元。在这里，每个终极符表示一个系统调用函数，正是这些函数构成进程运行时可直接观测的事件，但终极符略去了函数的参数，因为这些参数本身不作为刻画进程行为所需要的因素()[6]。当一个变元出现在产生式左面(例如产生式 *v** ::= *close*() *w* 中的 *v**)，在该产生式右面出现的表达式(即 *close*() *w*)可以在任何时候整个地替换左面的变元，用这种方式从一个变元出发持续实施对变元的替换直到所得到的字串中仅包含终极符而不含任何变元，例如从 Entry(*g*)出发，首先根据 Entry(*g*) ::= *open*()*v* 用 *open*() *v* 替换左边的 Entry(*g*) 得到 *open*() *v*，然后根据产生式 *v* ::= Entry(*f*) *v** 用 Entry(*f*) *v** 替换 *v* 结果得到 *open*()Entry(*f*)*v**，再用 *v** ::= *close*()*w* 替换 *v** 得到 *open*()Entry(*f*)*close*()*w*、用 Entry(*f*) ::= *getuid*()Exit(*f*)替换 Entry(*f*) 得到 *open*()*getuid*()Exit(*f*)*close*()*w*，如此下去，最终得出一个表示系统调用函数的合法序列 *open*()*getuid*()*close*()*geteuid*()*exit*()。

注意采用这一模型的关键优点在于反方向的命题成立：程序所蕴涵的每一个可能出现的系统函数调用序列都可以如此产生。正是在这一意义上，下推自动机模型精确表达了进程行为。同时我们也注意一个要点：因为略去了系统函数的参数，所得到的是所谓非确定性下推自动机模型，这意味着对同一个变元符号有多个产生式可以用来替换该变元时，随机选择任何一个用以替代。

如何从程序自动获取进程行为的这种规范模型？一个有效方法是从程序的控制流图(*control-flow*)出发进行计算。控制流图是编译器生成的、表达程序中的函数调用关系以及条

[5] 请回顾编译理论中的概念。
[6] 也可以使用这些参数作为表达进程行为的附加因素，这时得到更复杂但更精确的模型。

件分支等因素的一种复杂的数据结构，它刚好适合生成以上这类形式模型，具体细节不再这里深入了。

以上语法模型的目的是为了使 IDS 能在线检测出所观测到的系统函数序列是否合法，也就是直到当前为止所观测到的系统函数调用序列是否被以上语法所蕴涵。就检测机制本身而论，这类似于编译器的工作性质，后者在语法分析阶段的工作之一就是识别当前已读入的字符序列是否属于该编程语言所蕴涵的合法形式。从这一意义上讲，*Wagner* IDS 可以看做一个“在线编译器”，只不过所读入的对象是系统调用函数的序列而非程序语句，所输出的是是否发生入侵，即该序列是否合法，而非程序目标代码。更具体地讲，*Wagner* IDS 的工作过程基于状态栈，栈顶元素 *A* 或者是一个终极符，即某个具体的系统函数，或者是一个变元。IDS 始终监测进程当前调用的系统函数 *a*，如果当前的栈顶元素 *A* 恰是 *a* 本身，则抛弃当前的栈顶元素、继续观测下一个系统调用；如果 *A* 是变元且 *a* 恰是以某个产生式右面替换 *A* 后所得到的某个字符串的第一项，则抛弃 *A* 并且将这个新产生式右面压入当前的栈(该串的最后一项第一个压入、第一项最后压入)、继续观测下一个系统调用；如此循环反复进行下去，如果发生不属于以上两种情况的事件，就意味着观测到一个不属于该进程正常系统函数调用的状态，判定为存在入侵事件。这也正是编译器的自顶向下分析算法(LL-算法)的精神。下面是根据前一段的语法模型所建立的入侵检测算法，为简单起见这里仅给出了对栈的处理部分：

```
while(1) {
    case pop() of /*测试栈顶元素*/
        Entry(f): push(Exit(f)); push(getuid());
                  push(Exit(f); push(geteuid());
        Exit(f): 空动作
        Entry(g): push(v); push(open());
        v: push(v*); push(Entry(f));
        v*: push(w); push(close());
        w: push(w*); push(Entry(f));
        w*; push(Exit(g)); push(exit());
        Exit(g): 空动作
        default: 报告错误;
}
```

最后指出，这类入侵检测算法可以从提取出的语法模型自动生成而不必手工编制。

4.4.3* 一些技术细节

上面概要阐述了 *Wagner* IDS 的工作机理。完整实现 IDS 是一个复杂的软件工程，需要

处理许多技术细节，这些细节往往不为系统所依据的主要原理所覆盖，因此需要特殊的处理。这里针对 *Wagner* IDS 指出一些这样的细节，目的是使读者体会到 IDS 的复杂性，至于其具体解决途径则超过了本教程的程度，赶兴趣的读者可以参考本章后面列举的资料。

这些细节多数源于运行时的进程流程不能完全通过对其源程序的分析来确定，典型地包括对信号的处理、或更一般地说对异步事件的处理；对长跳转的处理；对线程的处理，等方面。

信号(signal)是 Unix/Linux 系统上特有的一种机制，每个信号表达一种特定类型的事件，例如时钟超时(SIGALRM)、(子)进程终止(SIGCLD)、进程停止(SIGSTOP)、地址空间被非法访问(SIGSEGV)等等[7]。信号机制类似于中断：信号可能在进程运行期间的任何时刻出现；每个信号关联一个特定的处理例程(signal routine)，一旦某个信号出现，进程实际上(在操作系统控制下)转入对应的处理例程，处理完成后才转回原来的流程继续运行。在 Windows 上虽然没有信号机制，但与存在 Unix/Linux 系统一样存在许多所谓异步风格的系统函数，以及所谓回调函数，前者被调用后的实际上与调用它的进程并行执行而返回时刻不确定(取决于该任务何时实际完成)，后者的调用时刻完全不确定，取决于特定的事件何时发生，因此它们引发的进程流程都不能由在运行之前由源程序完全确定。。

标准 C 库定义有一对长跳转函数 setjump/longjump，目的是方便异常处理或异步处理，使得进程流程可以从任何位置(通过调用 longjump)转移到任何事先(通过调用 setjump)指定的位置。由于长跳转操作实际上改变进程运行时的栈，因此不能直接由上一节基于栈的检测算法处理。

当进程中存在多个线程时，进程运行时的流程也不能完全由源程序确定，典型的情形就是多个线程之间存在同步，这时各个线程的执行次序取决于运行时的事件发生的条件，本质上属于典型的异步行为。

我们不在此深入阐述解以上决问题的技术。总之，实现高性能的 IDS 软件是非常复杂的工程，需要综合多方面的因素和技术。

4.5* HIDS 实例: 基于虚拟机的 HIDS

IDS 是目前反入侵技术领域很活跃的发展分支,已经提出许多有价值的新思想和新技

[7] Unix/Linux 一般具有近 30 种信号, 这些信号的涵义大多数相同, 但也有一部分存在区别或是某些系统的特有信号, 在开发具体软件时需要仔细研读该系统的编程手册。

术，但作为一本并非专门针对 IDS 的教程，这里无法详细展开阐述其丰富的内容，有兴趣的读者可以参考章末的一些材料。本节要阐述的是关于 IDS 的一个很有意思也非常现实的问题：如何设计和实现可靠的 IDS 软件本身？

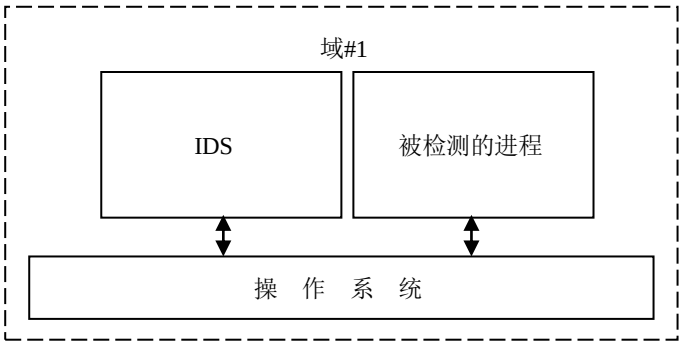
从概念上讲，这一问题一开始就存在：虽然 IDS 的目的在于保护其他软件免遭入侵，但其本身毕竟也是一种软件，所有导致其他软件遭受入侵的潜在缺陷和威胁对 IDS 本身同样存在。不仅如此，从攻击者的立场出发，IDS 必是他们要集中技术刻意破坏的第一个对象，理由不言而喻。因此，如何保护 IDS 本身就成为特别敏感、决定系统“命运”的关键问题。但从技术上讲，以往并没有特别好的解决方法。目前随着攻击技术日益成熟，关于这类问题的解决方案也被日益重视，同时当代软件(协同硬件)技术的发展也使得有条件发展出全新的解决途径，其中 *Garfinkel* 等最近建立的基于虚拟机的技术是一类很有代表性和发展前景的解决方法，本节对此概要阐述。

4.5.1 虚拟机

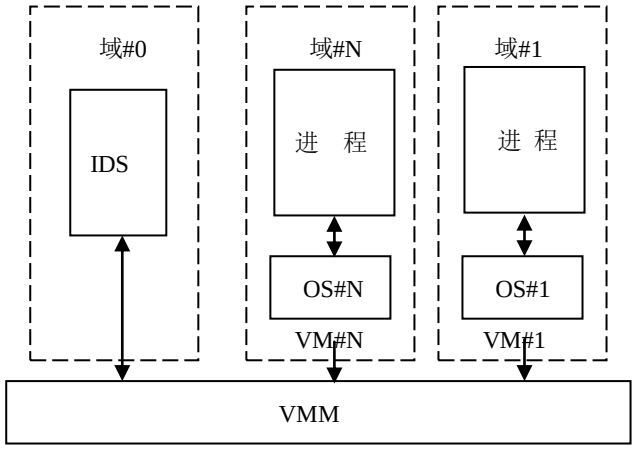
首先阐述虚拟机控制器(Virtual Machine Monitor，简称 VMM)的概念。VMM 实际上是直接位于硬件平台之上的一个“薄”的软件层，所谓“薄”是指相对于完整的操作系统而言，其功能简单(VMM 无需包含文件管理、虚拟内存管理、进程上下文切换等这类非常复杂的操作系统内核功能)，因此代码紧凑，例如当前典型的 VMM 仅约 3 万 C 程序。VMM 之上是完整的普通操作系统，VMM 将真正的硬件系统操作映射到这些操作系统内核所依赖的特定的硬件操作，由此，单一一个硬件平台可以同时运行多个不同的操作系统，例如同时运行 Windows 和 Unix。这些操作系统中的每一个连同其中运行的所有进程称为 VMM 支撑的一个虚拟机，简称 VM。因此，基本图像是一个 VMM 上同时运行多个 VM，VMM 完全隔离各个 VM，每个 VM 就象单独存在和使用硬件系统资源一样，当然这不过是 VMM 在底层对各个 VM 的资源使用(包括 CPU、内存空间、设备及网络上的输入输出处理等)进行协同和调度所造成的逻辑后果。

虚拟机系统的这一图像能够应用于新型 IDS 的关键特征就在于每个 VM 单独构成一个保护域，这些 VM 彼此隔离、互不影响，即使其中一个被完全入侵，入侵者也没有途径能影响到其他的 VM/保护域。既然如此，设想使 IDS 单独运行于一个 VM，即单独占据一个保护域，再通过特殊设计的接口使 IDS 能直接监视 VMM 中的事件(由于 VMM 的特殊功能地位，任何进程行为都不可能绕开 VMM)，由此既实现了对进程运行行为的检测，又使 IDS

本身与任何可能被入侵的进程隔离开，以此实现对 IDS 的保护。图 4-6 直观地表达了这一思想。注意在基于 VM 的 IDS 中，只有最底层的 VMM 是特权进程，其它 VM/保护域中的进程都可以仅仅实现为普通权限的进程，这一点也进一步增强了系统抵抗入侵的能力。



a) 普通 HIDS：所有进程位于一个保护域内(没有隔离)



b)基于 VM 的 IDS: IDS 单独占据一个保护域

图 4-6 基于 VM 的 IDS 与普通 HIDS

爱思考的读者或许会问：在图 4-6 的结构中 VMM 是系统安全的关键因素，既然操作系统和应用程序都可能存在缺陷从而导致入侵，有什么理由认为 VMM 就不会存在同样的问题？答案的关键在于 VMM 的复杂程度：因为其功能相对于完整的操作系统要精简得多，典型的 VMM 源程序仅 2-3 万行 C 语句，对这类小规模的软件，目前的软件工程技术是可以做到几乎没有缺陷或容易在开发阶段就发现几乎所有缺陷，这一点与至少几十万行 C 语句规模的大型软件完全不同。这也体现当代计算机安全技术的一条重要原则：软件越简单，越有把握进行验证。

4.5.2 基于虚拟机的 HIDS

基于 VM 的 IDS 的体系结构如图 4-7 所示，其中 IDS 位于单独一个保护域，被保护的操作系统及其中运行的进程位于另一个独立的保护域(构成 VMM 上的一个完整的 VM)。

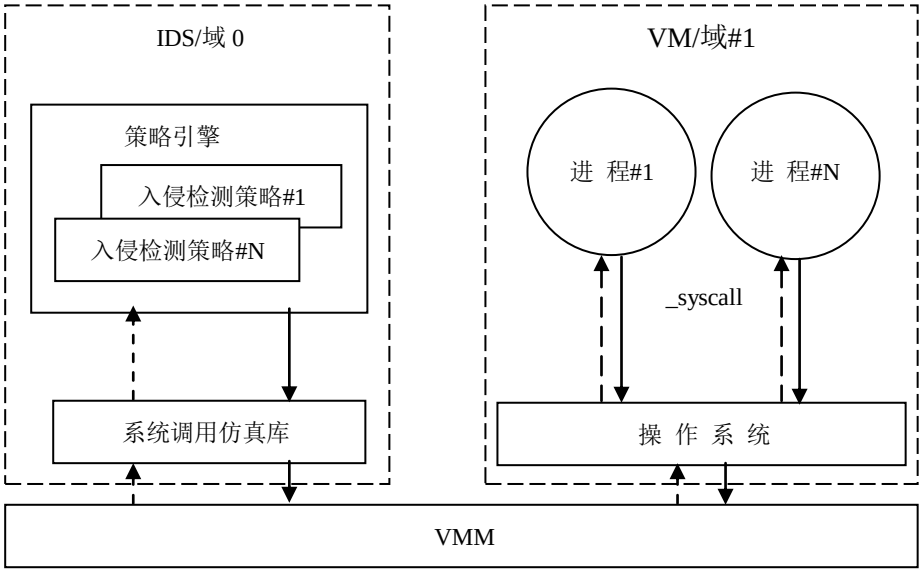


图 4-7 基于 VMM 的 IDS 体系结构

(实线表示请求, 虚线表示响应)

完整的 IDS 由两部分组成：入侵检测策略引擎和系统调用仿真库。系统调用仿真库和被保护域中的操作系统对应，每种特定的操作系统对应 IDS 中一个特定的系统调用仿真库。每当进程请求其所属的 VM 中的操作系统的任何系统调用时，只有那些需要存取硬件或导致多 VM 协同的情形才会通过 VMM 引发 IDS 可直接观测的事件，即使如此这些事件也不能直接反映出进程的系统调用特性，因此 IDS 不能直接根据这些最底层的事件建立进程的规范模型和进行入侵检测。IDS 进行入侵检测所依据的模型最好是象第 4.4 中的调用关系图或函数栈那样的高层模型，因为这类模型表达能力强，同时也相对容易提取，因此对基于 VM 的 IDS 需要一个翻译机制，将发生在保护域 VM 内的系统调用事件直接映射到 IDS 所属的域，使 IDS 在运行期间能够逻辑地“看到”被监视进程的系统调用序列，而非 VMM 上的基本事件。实现这一翻译机制正是系统调用仿真库的最主要功能，它为 IDS 的策略引擎提供一组接口，使 IDS 根据具体检测算法的需要提出各种检测请求，例如请求检测 VM 上当前存在那些进程、这些进程打开的文件状态、特定进程当前的系统调用栈、特定逻辑地址空间的内容等等。系统调用仿真库将这类请求进一步翻译为对 VMM 的访问请求以获取这些请

求的答案。借助于这一接口，现有 HIDS 可以在几乎保持现行检测策略和算法不变的情况下移植到基于 VM 的 IDS。

注意如果 VMM 上存在多个 VM、每个 VM 需要实施不同的入侵检测策略，基于 VM 的 IDS 只要配置相应的系统调用仿真库和策略组件就可以同时对这些 VM 上的进程进行检测，也就是说，基于 VM 的 IDS 具有多策略检测能力。

图 4-7 的 IDS 还有一种特性，就是一旦检测出异常，IDS 可以完全接管该进程或该进程所属的 VM。不难看出，能够做到这一点是因为 IDS 和被保护的进程属于两个独立的域。这一特性对普通的 IDS(图 4-6(a))是不可能做到的。

以上概要解释了基于 VM 的 IDS 的总体设计思想和优越性。这类 IDS 实现起来还需要解决许多细节问题，感兴趣的读者可以阅读本章后面列举的研究论文和著作。

4.6 其他技术

在结束关于 HIDS 的讨论之前，我们再概要列举几类 HIDS 的设计思想，这些思路也向读者展示出 IDS 设计思想的丰富多彩。

有一类 IDS(包括 HIDS 和 NIDS)，特别是早期的 IDS，在运行期间的主要功能实际上是尽可能地详细记录进程的系统调用事件，从功能上看，他们更象日志进程，而由另一个专门的分析程序通过分析这些日志记录来识别是否发生入侵。这是典型的非实时入侵检测系统，但它们的优势是能深入、完整地进行入侵分析，特别是结合应用当代人工智能和数据挖掘技术，因此这类系统在今天仍然在继续发展。

另一类 HIDS 与此相反。首先我们注意，第 4.4-4.5 两节所阐述的 IDS 虽然结构和检测机理很不相同，但都是在运行期间对进程实施入侵检测。对那些不依赖永久性感染的网络病毒，这是进行防卫的唯一途径。然而还有许多病毒，一旦入侵则将自己永久性地隐藏起来，日后继续实施破坏。最常见的方式就是病毒通过改变可执行程序的目标代码来使程序具有恶意行为。对这类入侵，最恰当的保护方式是在程序运行之前进行检测，以确认该程序一旦真正运行时不会导致不良后果。这就涉及程序仿真技术，或称反汇编技术，这一技术完全基于程序的目标代码(2-进制文件)判定该程序的运行时行为，下面是一个这类代码仿真算法的例子，将代码的目标指令(例如 JAVA 的字节码)解释为本机 Intel CPU 指令并在一个仿真环境(例如 VM)中执行：

```
while( ! halt && ! interrupt ) {
    inst = code[PC]; /*取当前指令到 inst, code 是被解释的代码起始地址.*/
    opcode = extract( inst, 31, 6); /*取源指令中的操作码*/
    switch( opcode ) {
        case  LoadWordAndZero:  LoadWordAndZero( inst ); break;
        case  ALU:  ALU( inst ); break;
        case  Branch: Branch( inst ); break;
        ……..
    }
}
/*  指令解释例程  */
LoadWordAndZero( inst ) {
    RT = extract( inst, 25, 5 ); /*  进一步提取指令参数  */
    RA = extract( inst, 20, 5);
    displacement = extract( inst, 15, 16);
    if( RA==0 ) source; else source = regs[RA];
    address = source + displacement; /*计算源指令中的逻辑地址*/
    regs[RT] = (data[address] << 32) >> 32;
    PC = PC + 4; /*指令指针*/
}
ALU ( inst ) {……. }
```

回顾第 4.4 节的 *Wagner* IDS，其检测模型依赖于源程序，这对许多软件并不现实，因此一个自然的想法是，如果能够从程序的目标代码—这是任何用户都能获得的对象—建立进程的行为规范模型，再结合以上仿真算法和已经实现的入侵检测算法，就可以完成对任何软件的入侵检测。因此，目标代码的仿真技术是非常有用的。

然而实现精确的代码仿真也需要解决许多细节问题，其中主要一类(也是从目标代码提取正确的行为规范模型时所遇到的主要问题)是如何正确地实现反汇编。这里的主要障碍包括如何准确区分 2-进制程序文件中的指令和数据、如何正确计算运行期间的地址转换，等。就目前流行的 ELF 或 PE 格式的可执行文件，要做到这一点并非总可能行，需要对这类可执行文件的内部结构做适当修改，这方面的具体细节读者可以参考本章后面列举的 DuVarney 等人的非常有趣的论文。

还有一类 IDS 针对可迁移代码，这类程序在因特网上非常流行，例如 Applet、ActiveX 和 JAVA classfile 这类可迁移代码，同时也是造成入侵的很大一类潜在威胁。这类 IDS 实际上是所谓沙箱(SandBox)，它们本身类似于一个虚拟机，为外来代码建立虚拟执行环境，外来代码在执行过程中只能访问虚拟机中提供的资源对象，虚拟机严格限定其访问该环境之外

的本地或远程资源对象，以此强制其行为符合本地安全策略。图 4-8 是一个沙箱系统的典型结构。

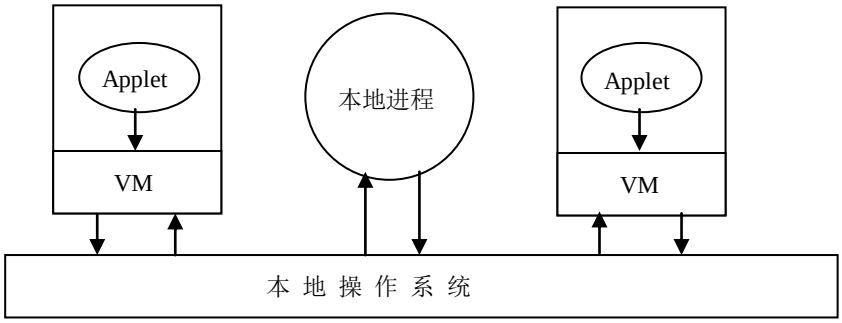


图 4-8 沙箱系统

4.7 小节与进一步学习的指南

IDS 是计算机安全领域目前非常活跃的研究课题之一，已经提出许多有趣的思想和技术，同时 IT 产业界也已经开发出多种 IDS 商品软件(包括 HIDS 和 NIDS)。第 4.4 节描述的 HIDS 及其方法的完整阐述详见 *Wagner* 的博士论文，其中包括检测算法的详细描述和对典型入侵事件的检测性能的详细测试与分析：

D.Wagner *Static Analysis and Computer Security: New Techniques for Software Assurance*, Ph.D Thesis, University of California at Berkerley, 2000. 该系统后来还被进一步改进，参见：

D.Wagner *Mimicry Attacks on Host-based Intrusion Detection Systems*, CCS'02, 2002.

关于基于规范方法的 IDS 的最经典的工作是 Ko 的博士论文，其中对许多问题和概念的阐述至今仍然值得参考：

C.Ko *Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach*, Ph.D thesis, University of California at Davis, 1996.

读者不必对这些博士论文产生畏难情绪。这里列举它们，主要是因为它们属于该领域有重要参考价值的文献，初学者不妨读读这些论文的前言和背景介绍部分，以得到关于该领域的一个概要认识，同时作为教科书内容的补充，这将是很有益处的学习方法。关于其中所运用的有限状态自动机模型(等价于正规语言)和下推自动机模型(等价于上下文无关语言，所有编程语言都属于这一类语言)的详细理论，可以参考任何一本关于编译理论和技术的优秀著作。

基于进程的系统调用模型建立 IDS 的经典论述可以参考 S.Forrest, S.Hofmer,

A.Somayaji *Intrusion Detection using Sequences of System Calls*, Journal of Computer Security, 6(3):151-180, 1998. 基于虚拟机的 HIDS 的奠基性工作可以参考 T.Garfinkel M.Rosenblum *A Virtual Machine Introspection based Architecture for Intrusion Detection*, Proc. Symp. Network and Distributed Computing Security, 2003. 这些论文都写得概念清晰、深入浅出，适合初学者阅读。

虚拟机是一项十分经典的技术，早在 60 年代就在大型机上有应用，但目前又发展成一个活跃的领域，这方面最新的内容丰富的著作是 A.Smith *Virtual Machines*, Prentice-Hall, 2005, 电子工业出版社已出版影印本。

关于修改可执行文件格式以支持从目标代码获取入侵检测模型的工作见 D.DuVarney *SELF: a Transparent Security Extension for ELF Binaries*, Proc. CSS, 2002. 该设计最终通过修改的编译器自动实现。读者也可以从中了解到可执行文件的结构。

本章描述的关于 IDS 的技术内容多取自研究性的系统，其中一些技术已经或将有潜力应用于 IDS 商品软件。关于具体 IDS 产品的技术性描述，可以参考 *Symantec* 公司和 *Kaspersky* 公司等领袖开发商网站上的技术白皮书，其中许多材料特别是案例部分适合作为教科书的补充内容。

习 题

4-1 比较 4.1-4.2 节中的各种解决方案，从实用性角度分析它们各自的优缺点。

4-2 根据 4.4.1 节的描述，请给出从源程序构造调用关系图的完整算法，以及根据调用关系图实现对进程的函数调用的状态跟踪算法。

提示：前者可以参考编译理论中控制流分析技术的最简单情况；后者实际上类似于编译器中的词法分析器的工作原理，两者都容易实现。

4-3 用正规表达式描述调用关系图 4-4 所蕴涵的所有事件序列(即示例程序运行期间所“允许”的函数调用序列)。

4-4 对以下简单的 TCP 服务器程序，1)做出其函数调用关系图 2)写出表达其系统函数调用序列的上下文无关语法的产生式系统。

```
int fd1, fd;
 fd=socket(AF_INET, SOCK_STREAM, 0);
 bind(fd, …);
 listen(fd,…);
 while(1){
```

```
        fd1=accept(fd, ….);
        read(fd1, buff, size);
         /*解析客户请求并响应*/
        ret = request_proc(buff,size,…);
        if(ret>=0){ /*ret<0 表示有错误，例如非法请求*/
            write(fd1,…);
        }
        close(fd1);
    }
```

4-5 选择一个你自己编写的较复杂的程序，重做习题 4-4。