第 5 章 操作系统安全

本章讨论与操作系统相关的安全问题，大体包括程序带来的安全问题、操作系统对用户的保护服务和安全操作系统的设计。操作系统对于整个计算系统的安全性起着基础作用。一旦操作系统的防御被突破，整个计算系统的安全就会受到根本的威胁。

5.1 程序带来的安全问题

攻防是一对矛盾，有时不妨从攻击者的角度来看安全问题。下面将讨论如何通过程序利用计算系统的弱点进行攻击的方法。(以下“程序”一词暂指操作系统之外的用户程序和系统公用程序。)

虽然操作系统也是程序，但有其特殊性。首先，操作系统为所有运行于其上的程序提供多方面的保护。其次，操作系统的规模之大和复杂性之高使其成为较难攻击的目标(由于同样的原因，要使操作系统的安全性得以全面、妥善的实现，也很不易)。据此特殊性，将操作系统的安全问题放在后面再讨论为宜。

程序可带来两类麻烦：它们可以成为用户非法窃取或更改数据的手段；它们还可以借以利用计算系统服务的缺陷，使用户非法获取系统访问权或禁止合法用户的访问。

不幸的是，程序缺陷的范围比控制利用这些缺陷构成的威胁的技术措施的选择余地大得多。其原因有二：

1)程序控制应用于单个程序员的水平，某些明显的缺陷可以检测出来，但专业程序员仍能成功地隐藏一些缺陷；

2)虽然一直要求操作系统担当起保护计算系统的职责，但是为了良好的服务和共享，即使有高级别保护要求的操作系统，也只能做到排除主要的违反安全的隐患。

这里的文章内容不打算被可能的计算机罪犯当作“如何作弊”的技术手册，因此许多解释会显得有点含糊，但足以指明弱点之所在。

5.1.1 信息访问问题

程序根据数据运行，其本身说不上是什么安全威胁。但由于计算机数据通常以人不易懂的形式存在，程序经常被未经许可的用户当作访问数据的工具。我们从几类可能被用来对数据进行未经许可的访问的程序着手研究。这种访问可涉及窃取、更改和伪造。

1. 陷门

陷门是一个模块的未记入文档的秘密入口。陷门是在程序开发期间的某个时候插入的，也许是为了测试这个模块；也许是为了将来更改或增强模块而提供的“钩子”；也许是为了将来发生错误时以便于访问。除了这些合法的用处，陷门可允许程序员在程序运行时访问这个程序。

由于计算系统结构复杂，程序员通常以模块方式开发测试系统；先测试系统的每个小部件，再将这些部件组装成一个个逻辑簇，并分别测试每个逻辑簇。

每个部件一开始测试时是没有准备输入和处理输出的外围程序的。为了测试一个个单独的模块，可能需要编写“驱动”程序和“存根”程序，这是一些用来对被测试程序注入数据和收集结果的简单程序。当测试继续进行时，这些“驱动”程序和“存根”程序就被废弃了，因为它们已被它们所模拟的实际程序取代了。

在测试期间，也许会发现模块中的缺陷。当缺陷的根源不明显时，就在有疑问的模块中插入调试代码，以显示这些模块的中间计算结果或执行额外的计算以检查这些模块的正确性。

插入指令是一种公认的测试做法。如果在测试完之后，额外的指令仍然留在被测试的程序中，就可能产生问题。它们作为未归档的控制序列，可能会产生副作用或被用作陷门。这是产生陷门的一种原因。

不良的错误检查是产生陷门的另一种根源。在某些设计不良的系统中，不允许的输入可能未查出而被接受。例如：一个程序的目的是搜寻三个期望的序列中的一个，如果实际写成在 CASE 语句中只测试这三种可能性，而缺少了在一个也找不到的情况下判为出错的程序段落，则故障就会穿过这个程序。

. 这种缺陷的另一个普通例子可以在硬件处理器的设计中发现，即并非所有的 0～1 操作码值都有匹配的机器指令。那些无定义的操作码有时执行奇怪的机器指令，这种在测试处理器设计时出现的反常现象，往往是由于处理器设计者的疏忽造成的。这是先前描述的软件缺陷的硬件对应物。

陷门并非总是坏事，它们在发现安全缺陷方面是非常有用的。审计程序有时需要借助生产程序中的陷门往系统中插入虚设的但可识别的业务，以便追踪这些业务通过系统的流向。

程序员通常在程序开发过程中适时地去掉陷门。但是，程序员也可能因为下列原因使陷门存留于程序中：

1)忘了去掉；

2)故意留在程序中以便有助于别的测试；

3)故意留在程序中以便有助于维护已完成的程序；

4)故意留在程序中以便在它成为认可的产品程序后，有一种访问此程序的隐蔽手段。

第1)种情况是非故意的安全疏忽；第2)和第3)两种情况是对系统安全的严重威胁；而第4)种情况则显然是预谋攻击的第一步。过错不在陷门，陷门是程序测试、改正和维护的十分有用的技术。过错出在程序开发环境上。只有在未受注意并且在程序易受攻击却无人采取行动来防止或控制陷门的使用的情况下，陷门才成为使系统易受伤害的弱点。

陷门正因为使系统在运行之中易被更改而成为弱点。陷门可被原来的程序员利用，也可被意外或通过穷举搜索发现陷门的任何人利用。

2. 特洛伊木马

在希腊神话中，特洛伊木马是一件载有不速之客的礼物。计算系统也可能被特洛伊木马访问。在计算机中，特洛伊木马除了它声明的功能外，还执行隐藏的功能。

举例来说，假定一个程序员想修改别的用户的文件的保护级别。这个程序员编写了一个程序，表面上产生合乎规格的列表文件，并将该程序提供给计算系统的管理员，作为所有用户可用的公用程序。但是，此程序员却不提这个公用程序还会改变这些文件的保护级别。

由于是原程序员之外的一个用户调用这个公用程序，此程序也许将在这个用户的保护级别上执行。因而，这个公用程序可以访问这个用户的文件，它也许有权改变别的用户访问这些文件的权力。更复杂的特洛伊木马更改访问权，通知窃收者可访问一个特定的文件，暂停，然后将访问权限设回原来的值。如果窃收者很快地完成所期望的窃收，则特洛伊木马可掩盖它的踪迹，因而用户也不知道已发生过的访问。

程序员可以采用直接提供目标代码的方法，偷偷地改变已被接受的公用程序，甚至采用加密等方法将特洛伊木马隐藏起来。

3. 意式香肠攻击

意式香肠攻击是喻义用零碎肉做成的意大利式香肠而得名。在这类攻击中，每次计算只搜刮不引起注意的一丁点零头钱，次数多了，便积沙成塔。程序员可将搜刮来的零头转入选定的帐户。用户对小利的忽略，对复杂计算细节的无知，以及管理、审计不严都是造成意式香肠攻击存在的原因。程序规模庞大、结构复杂，对程序员作弊提供了很好的掩护。

4. 泄漏信息的程序

可以编写一种程序，把信息传送给本不应该知道这些信息的人，这种反常的信道称为隐蔽信道。

在有敏感数据的环境中，当程序正式运行后，程序员不应当访问这些程序运算的数据。例如：银行的程序员不应访问储户的帐户名或存支余额。又如，程序员可以从了解

股票出入意向而获益。在许多情况下，程序员可能企图开发隐蔽信道，秘密传送某些数据。

程序员总能找到秘密传送数据的手段。例如：在表面上完全没有问题的报表的特定位置上增减一个空格，就可携带 1 比特信息。在大量正常数据中携带少量秘密信息的办法实际上很难被检测出来。

5.1.2 服务问题

除了访问数据方面的问题，程序也可以用来影响计算系统的服务。可以编制能阻塞系统使其它计算不能正常进行的程序，合法用户被排斥而不能进入系统，这种安全故障称为“拒绝服务”。

1. 贪婪程序

某些计算系统中驻留着插空执行的“后台”任务，即优先级很低的计算。例如：某研究用计算机上有一个计算极长位数的 e 或 π 值的程序，并不急于得到计算结果，只是在机器空闲时才被调用。如果被有意或错误地改变控制，这个后台任务可占据前台的位置，从而阻塞所有其它的计算。

2. 循环

贪婪程序的一个简单例子是无限循环。大多数的多道程序计算系统有时钟限制，以防止一个用户的程序占用的时间过长。操作系统通常设定一个时间上限，比如 24 小时。

时钟所计时间通常是 CPU 占用时间，因为等待 I/O 的时间不好预测，I/O 时间取决于其它程序引起的系统装入，通常 I/O 时间是不受检查的。在某些系统中，一旦一个进程申请一项 I/O 服务，它就等待 I/O 行动的完成，并停止其 CPU 时钟。I/O 行动是异步进行的，由 I/O 处理器控制和执行。编写一个无限循环的 I/O 程序不很困难。这种思想还能够扩展到许多启动部分或全部 I/O 设备的程序。一旦这程序全部开始执行，整个系统就被有效地占据了。

3. 计算机病毒

计算机病毒是贪婪程序的逻辑延伸；在后面第 7 章还将专门讨论它。而且，有时直接用病毒这个词，主要是在不引起混乱的前提下力求简单起见。“病毒”是能够更改其它程序而使其“感染”的程序。一个程序被计算机病毒程序感染，也就是被改造而包含计算机病毒程序的一个复制件，它又能去感染别的程序。感染以几何速率扩散，最终将占领整个计算系统。在最初的计算机病毒试验中，有的计算机病毒可在 5～30 分钟内植入并占领整个计算系统。

并非所有病毒都是不良的。例如：一种病毒可寻找未被感染的程序，将它们压缩以便占据较少的空间，并插入一个当它们执行时解压缩的程序复制件。这种计算机病毒可有效地减少存储程序的存储空间，有时可达 50%。但是这种压缩是应计算机病毒的要求进行的，而程序的拥有者却一无所知。

计算机病毒可以植入共享的系统公用程序，从而访问电子函件、系统消息公告和系

统用户表等公共数据。因为许多用户访问这些公用程序，植入的计算机病毒能够很快扩散。

开发一个计算机病毒程序的时间可以短得惊人。

在某个案例中，一个 200 行 FORTRAN 代码和 50 行命令文件的病毒还不到 24 小时就制造出来了，而作案者对他所攻击的计算机还没有什么经验。除了编制速度快之外，保密也使得发现计算机病毒为时太晚。一个计算机病毒可编写得能掩盖住它扩散的大部分踪迹。

由于计算机病毒可以相当小，它的代码可以隐藏在其它更大更复杂的程序中。200 行程序的计算机病毒可以分成 100 组，每组两行加一个转移指令，从而很容易隐藏在编译程序、数据库管理程序、文件管理程序或其它一些大的公用程序中。当然，可以通过一个确定两个程序是否等价的过程来帮助发现计算机病毒。但是，由于等价问题的复杂性，理论结果非常令人失望。一般而言，这个问题是不可判定的，这就意味着要确定一个被感染的编译程序与一个未被感染的编译程序是否产生同样的结果是极其困难的。因而，不大可能开发出一个能够区别受感染模块和未受感染模块的筛选程序。

检测出某些已知计算机病毒是可能的，也就是说，如果知道一个特定的计算机病毒能够感染一个计算系统，就有可能检测出那个计算机病毒。然而，当发现计算机病毒后，还有从系统中清除计算机病毒的任务。从运行的系统中去掉计算机病毒，要求能够测试出来并且消除计算机病毒的速度快于计算机病毒扩散的速度。

计算机病毒扩散的速度依赖于共享和传递性。计算机病毒必须能够使用共享的信息并使其它程序共享它的信息。因而，限制信息的共享能够限制计算机病毒的感染。在下一节将讨论区隔的概念，通过区隔可将数据分隔成互不联系的部分。在信息被分隔的系统中，计算机病毒的作用被限制在一个区隔中。

类似地，信息从一个进程流向另一个进程。如果有一个信息从 A 流向 B，并从 B 流向 C，就意味着这则信息从 A 流向 C。通过限制使用被传递的信息的数量，就可以限制计算机病毒扩散的范围。

4. 蠕虫

庶克(Shoch)和哈普(Hupp)的蠕虫程序是计算机病毒在网络上的推广。蠕虫利用计算系统的网络管理机制，识别网络中空闲的计算机，并将蠕虫程序传递给它。一旦被激活，这个蠕虫又企图发现另一个空闲的机器，然后将它自身的一段传送过去。与计算机病毒一样，蠕虫几乎可以嵌入任何有意义的计算机程序中。

某些蠕虫程序有合法用途。例如：运行系统广告牌或闹钟。蠕虫还可以应用于由一系列计算机构成的网络上的并行计算，在这些情况下，蠕虫完成特定的任务，然后退出。但是，一般的蠕虫也可以无休止地运行下去，从而导致系统拒绝别的用户的访问。

庶克(Shoch)和哈普(Hupp)列出了几个蠕虫程序的例子，包括在 ARPA net 网上运行的例子。网络控制程序本身就是蠕虫的例子，因为 ARPA net 网使用分布控制来管理这个具有数千个用户的网络的资源和共享。所有关于蠕虫的实例都是有关正当使用网络设施的。在这些例子中，都是启用多主机来完成超出单主机支持能力的计算。

由于蠕虫的行为可能变得不友好，一些有远见的设计者加上了在必要时杀死蠕虫的

手段。当一个蠕虫变异了(由于一些未知硬件和软件的连入)，并复制出可能使它所达系统崩溃的副本时，便使用这一手段。但并非所有的蠕虫开发者都有这种远见。如果不预备制约蠕虫的措施，一旦有事，求助的手段也许只有在网络中为每个机器找一个干净、无蠕虫副本的完整系统。在蠕虫释放前保存的系统备份足以达到此目的。但如果蠕虫一直在活动，也不知道什么时候发生了变异，则所有的网络用户都有可能遭受严重损失。

5.1.3 防备程序攻击的程序开发控制

如上所述，程序员有许多破坏系统以谋取私利的办法。下面将讨论在软件开发过程中，在设计、编码和测试各阶段预防潜在威胁的控制方法。

1. 编程任务的描述

过去，在程序设计的原始生产模式中，程序员接受一项要实现的任务的描述，独自完成该任务的编程，然后将程序交回。支持程序员单独工作的论点如下：

(1)编程是一项个体劳动，要求独立思维。将其思想与别人交流太费时间，甚至无益。

(2)程序是程序员才能的创造性表达，是很个性化的。指望两个不同的程序员共同完成一个项目是没有道理的。

(3)程序员们基本上是些喜欢独自工作的人。破坏他们所喜欢的工作方式，对程序或程序员会产生消极的影响。

(4)程序设计是一项只能由程序员理解的艺术，管理人员无能力理解程序(或者说管理人员不情愿被迫去努力理解程序)。

这些论点没有一条是成立的。阻碍程序设计由单个人完成的基本情况是程序的规模太大。一个好程序员一年的期望产出是 2 000 行代码(由于编程语言、任务的复杂性及环境的差别，或许能乘一个 2～3 的因子)。某些好的程序员在完成复杂的任务时平均每天只能编两三行代码! 以这种产出水平，单个人对数十万甚至上百万行代码的现行较大系统是不能胜任的。

软件工程是指大规模程序设计的问题，也就是说，为巨系统编写代码。软件工程的基本原则是工作划分、代码复用、标准的事先规定的软件工具的使用和行动的组织。

2. 对等检查

当一个系统很大，需要若干人同时编程时，所有人都必须有精心制定的文件，说明程序的各个部分都是做什么的，每个部分如何与其它部分接口。由于制定文件在某种意义上是按照每个程序员的个人解释去做的工作，及早找出理解不一致的地方是很重要的，找出程序员的逻辑缺陷也是很重要的。

软件工程的基本思想是：正确的程序是全体程序员的共同责任。因此，每个成员都要分担对等的设计检查和对等的代码检查的责任。当一个设计人员或编程人员完成了一段特定的工作之后，要请几个同事“过”一遍。原开发者按规定的章程出示材料，听取别人的评论、提问和建议。这些问题是为分辨误解和错误而设计的。

这种程序设计方式称为“无偏”程序设计。人们必须认识到产品属于整个集体，而

不是属于生产它的个人。这种检查不是为了责备出错的个人，而是为了找出错误使产品的质量更好。只有产品没问题，整个小组才算成功了。因而，小组的全体成员对产品的正确性有既定的利害关系。

因为所有的检查者本身都是设计者或程序员，他们了解编程技术；他们能够区别错误和正确的但可能并没有用过的程序；他们知道程序中的什么事情值得怀疑，什么事情不对头，什么事情有不明显的副作用。

严格的设计或代码检查能够找出陷门、特洛伊木马、意式香肠攻击、蠕虫、计算机病毒和其它的程序缺陷。狡猾的程序员能把这当中某些缺陷隐藏起来，但当称职的同行们检查代码的时候，发现的机会还是更大一些；当检查是在 30～60 行模块的级别上进行时更是如此。管理部门应当在整个程序开发期间，采用规定的程序检查作为保证所生产的程序的安全性的手段。

3. 模块化、封装和信息掩盖

软件工程的原则告诫人们，将程序编写成小的、自足的、被称为模块的单元。模块化不仅在程序开发方面有优点，在安全性方面也有优点。模块可以被隔离，免受与之有相互作用的模块的负面影响。这种隔离是采用了称为封装的设计原则而实现的。信息掩盖是模块化的另一个好处。由于信息掩盖其它模块知道一个模块完成一定的任务，却不知道它是如何完成的。下面阐述这三项原则及其在计算机安全中的作用。

模块化是将任务划分成子任务的过程。每个模块完成分离的、独立的部分任务。程序单元的大小应当控制在完成其功能所需要的限度之内。将程序编写成一系列小的、自足的模块有几个优点：

(1)可维护性　如果一个功能由一个单独的模块实现，则此模块可在必要时由改进的模块代替。当设计要求、硬件或环境改变后，就可能需要新的模块。有时这种替换只是一个增强模块，它更小、更快、更正确或其它方面有所改进。这个模块与程序其它部分的接口很少而且描述得很好，因而替换的影响是一目了然的。

(2)可理解性　由许多小模块组成的程序比大的非结构化的程序更容易理解。

(3)可复用性　为某个目的开发的模块通常还能重复使用于其它场合。重复使用现有的正确的程序模块可减少编程和测试的困难。

(4)可更正性　如果每个模块仅完成一个任务，则可很快找到出错的原因。

(5)可测试性　输入、输出及功能定义良好的单个模块可对其进行详尽的测试，而不用考虑对其它模块的影响。当然，预期的功能和输出除外。

从安全观点看，每个模块能够作为独立的单元来理解，并保证它对其它模块只有受限制的作用是重要的。恰当的模块结构可产生与其它模块具有最小相互作用的模块。

模块结构产生一种独立的形式，其中每个模块都作为一个独立的对象来完成其功能。设计良好的模块与程序中的其它子程序只有很少的耦合作用；而其它子程序不受与其它模块的不明干扰，这种特征称为封装。在这种状态下，一个模块工作起来基本就像被盾牌围起一样，防止了外部对模块的非正常访问。

由于封装，模块只能通过一定的定义和良好的接口相互作用。一个模块只能从规定的入口进入，一个模块也尽量只与最少的其它模块相互作用。

封装并不意味着完全的隔离。模块需要一定的输入并必须与其它模块交换信息。然而，这种共享是有详细的文件记录的。因此，一个模块只能被系统中的其它模块以已知的方式影响，共享被限制到最小化，从而只使用可能的最少的接口。对接口的限制，减少了可能构造的隐蔽信道的数量。

模块化设计使得一个模块对其它模块只有有限的影响，反之亦然。进一层说，一个模块可看作一种形式的黑盒子，具有确定的良好定义的输入、输出及功能。其它的模块和其它的设计者不需要知道该模块是如何完成其功能的，只要肯定这个模块以正确的方式完成它的功能就够了。

4. 独立测试

测试的目的是要确保程序的正确性，而不是因错误责备谁。测试的直接功效是找出错误。当什么错误都没发现时，说明被测试程序是正确的，或者虽有错误但未测出。

程序员与自己开发的程序关系太密切对测试不利。因此，通常采用独立的测试小组。当程序设计完成后，测试小组就着手开发测试数据。构造测试数据时并不参考程序本身的源代码。由测试小组设计的测试实例检查程序是否完成设计的功能，而不一定就是程序员所解释的设计的要求。测试小组和程序员对设计的解释可能不同，但应找出有歧义之处加以解决，而不要将想法不同的程序员编出的模块组装起来。

从安全的角度出发，独立测试的效果是十分理想的。因为，企图在程序隐藏些什么的程序员不参与测试工作，这就增加了暴露隐患的可能性。

5. 配置管理

配置管理是有利于安全的软件工程的另一个方面。实施配置管理时，人或系统控制并记录程序和文件的所有更改。一个称为更改控制部的专家组评审所有提出的更改的合理性和正确性。

配置管理的目标是要保证对所有系统部件，包括软件、设计文件、说明文件、控制文件等有用版本的使用和可获性。配置管理简而言之就是强化组织和簿记。

实际上，每个程序员都“丢失”过程序的某个版本。程序员改了一回又一回，直至放弃一个方案。这时，原始版本已被删除了或与其它版本混淆了，要不然就是掺杂了无关的更改。

如果一个程序有几个并行的版本，情况就更糟。例如：假定一个公司有一个计算机程序要销售，在每一个版本开发和销售后，用户报告有一些小错误，公司作了修改。为了不让错误扩散，公司更新了待售版本，改正了任何已知的错误。同时，公司正致力于开发此产品的增强版本，并最终将投入市场。在这种情况下，这个产品至少有三个不同版本：原始版本、错误已更正的版本和增强版本。进一步的错误修改或增强还会导致更多的版本。

如果程序较大而必须由若干程序员编制的模块构成，当某个人更改了一个模块后，别的程序员必须知道，因为这个模块可能影响别的模块。编程人员不能任意修改程序，即使这些修改是为了更正已知的错误也不行。通常，程序员应保存更正后程序的拷贝，等待更新周期的到来。在这一段时间中，所有的程序员将把他们程序的新版本组装起来重新测试整个系统。因此，程序员都有静态版本和工作版本。随着系统开发的进展，就

有在不同阶段测试或与其它模块组装的不同静态版本。

以上阐明了配置管理的三个具体目的，即：

(1)避免无意丢失(删除)某个程序的某个版本。

(2)管理一个程序几个版本的并行开发。

(3)提供控制组装成一个系统的模块共享的设施。

这些目的可通过管理源程序、目标代码和文件的系统方法达到。这个系统方法应有仔细的记录，使人知道每个版本的拷贝存在哪里，一个版本与其它版本有些什么不同的特征。公司通常指定一个或多个配置管理专家来执行这项任务。

一般一个程序员在某个时候“冻结”一个模块，形成一个静态版本，并将其控制权交给配置管理部门。从这时起，这个版本的所有修改都由配置管理部门监督进行。配置管理部门要审查所有修改请求的正确性以及对其它模块的潜在影响。

采用配置管理在安全上有两点益处：一是预防非故意的威胁；二是预防恶意威胁。

为了对修改进行易于理解和易于审计的控制，配置管理部门通常只在源程序级别上接受对程序的修改。当要产生一个新的版本时，配置管理部门建立一个暂时用于编译的源程序。以这种方式，便有了修改的精确记录，每一处修改是何时由谁进行的。

6. 程序正确性证明

安全专家希望能够确认对于一个给定的程序和一个特定的计算结果，该程序除了正确地进行计算之外，再也没有做任何其它的事。遗憾的是，计算机科学理论的结果表明，对于每一个可能的程序来说，安全专家不能够确认这一点。给定任意两个程序，不存在一般的判定过程来确认这两个程序是否等价，这是“停机问题”的一个结果。这个结果说明，当一个任意的程序处理一个任意的输入时，没有一般的技术来确定这个程序是否将会停止。

尽管有这个令人失望的一般结果，一个称为程序验证的过程可在形式上证明某个特定程序的正确性。程序验证要涉及对输入作出初始断言。每个程序语句都翻译成一个逻辑，此语句说明了它对程序逻辑流的贡献。最后，将程序的结束语句与程序希望达到的结果联系起来；然后，通过应用一个逻辑分析器，有可能证明初始假设通过程序语句的蕴含式产生最后的结论。用这种方式可证明程序是否达到它的目的。

程序正确性证明有几个方面的障碍：

(1)程序正确性证明取决于将程序语句译成逻辑蕴含式的程序员或逻辑学家。正像编程会出现错误一样，这种翻译也会出错。

(2)从初始断言和语句的蕴含式推出正确性的证明是很复杂的。因此产生证明的逻辑机器运行缓慢。当程序的规模增大时，逻辑机的速度还会降低。因而正确性证明对于大的程序是不适当的。

(3)当前程序验证的状况跟不上程序的生产。正确性证明还一直没有成功地应用于大型的生产系统。

(4)程序验证系统正在不断地改进。较大的程序的验证耗费时间比以前少了。随着程序验证技术不断成熟，它将成为保证程序安全的更重要的控制手段。

5.1.4 操作系统对程序使用的控制

上述对程序员的程序开发控制可应用于某些软件生产环境中的大型开发项目。但是，并非每个程序都是以这种方式开发的；而且，一个计算机用户也不能总是保证系统以所有的其他用户都遵循适当的程序开发标准。因而，更普通的软件安全标准由操作系统来实施。

在下两节中，将较详细地考察操作系统，以确定它们能为用户提供什么样的安全特征。下面概述操作系统能够提供的保护类型，这些保护能防止本书前面所述的程序缺陷。

1. 可信软件

可信软件指被认为是安全的代码，即功能正确，能完成设计所规定的功能而不做别的事情；还有正确地实现完成这些功能的程序。操作系统可以是可信软件的一部分。相信开发者正确地设计了程序的模块；相信程序员只用了必要的语句；更重要的是，可以相信操作系统正确地控制了在此操作系统中运行的模块的访问。例如：操作系统如期望的那样，限制了某些用户对某些文件的访问。

基于严格的分析和测试，可信软件有下列特征：

(1)功能的正确性　程序只完成要它做的事，并且正确运行。

(2)完整性的实现　即使出现来自未授权的用户的指令或错误的指令，它也能维持所接触数据的正确性。

(3)有限制的特权　允许程序访问安全数据，但访问的数量被限制到最少，并且访问权限和数据都不被传递给不可信的其它程序或不可信的主调用程序。

(4)适当的安全级别　程序已经通过检查，并根据数据的种类和使用环境给定以适当的可信等级。

从根本上来说，可信软件是一般用户访问敏感数据的安全方法。可信软件用来为用户完成敏感的操作，而不允许用户直接访问敏感数据。

作为一个简单的例子，考虑一张按散列编码排序的表，记录的位置是非常重要的。因为，一个位置错误的记录可能意味着不仅这个记录，而且其它记录不能被检索到。为了不让任意访问这张表，可编写一个正确插入和删除记录的模块，使得所有对该表的访问都通过这个模块来进行。此时，该模块就成为一个可信的接口，访问这张表都必须调用该模块。

2. 相互怀疑

程序并非总是可信的。即使有操作系统实施访问限制，要有效地限制一个未测试过的程序的访问特权也许是不可能或不现实的。相互怀疑原则用来描述两个程序之间的关系。相互怀疑的程序运行起来就好像系统中的其它程序都有缺陷一样。一个调用程序不相信它所调用的子程序，而一个被调用的子程序也不相信调用它的程序。每个程序都对它的接口数据进行保护，使得其它程序只能有限地访问。

3. 限定

限定是指操作系统用于一个值得怀疑的程序的技术。一个被限定的程序在它所能够

访问的资源上受到严格的限制。

限定原则类似于军事上的数据按密级分类，每个数据项都被标上密级，如“机密”、“绝密”等。不可信的个人或程序只能访问与自己的密级相应的数据项。

4. 信息分格

对信息的分格类似于限定。在一个分格的系统中，所有的数据和程序都分为不相交的组，一个程序只能访问同格中的程序和数据。如果一个程序是错的或是有恶意的，它只能影响同一格中的数据(或程序)。

分格在限制计算机病毒扩散方面是有用的。计算机病毒只能在一个格中扩散，不能跑到格外面去。

5. 访问日志

访问日志是一张表，它记录了谁访问了哪个计算机目标，什么时候访问的及所用的时间是多少。它广泛地应用于文件和程序。与其说它是一种保护手段，毋宁说它是一种用于事后检查发生了什么的跟踪手段。

访问日志所记录的可能是：联机或脱机，对文件和目录的访问或企图进行的访问，程序的执行以及其它设备的使用。

故障也在访问日志中记录下来。就安全性而言，记录下特定用户列出容许访问的目录，还不如记录下防止了同一个人企图列出受保护目录来得重要。一次联机失败可能是一次键入错误的结果，而短时间内同一设备上的故障记录可能就是入侵者企图入侵系统的证据。

对审计日志中出现的非寻常事件应仔细调查。例如：一个新的程序可在一专用的受控环境中测试。在测试完毕后，应当浏览一下所有被访问文件的审计日志以确定是否存在不希望有的文件访问，这样可能发现这个新程序中的一个特洛伊木马。

5.1.5 行政管理控制

并非所有的控制都是由计算系统自动执行的。以下将讨论能够在人员管理过程应用的控制。

1. 程序开发标准

较好的计算部门不允许程序员随便什么时间、随便怎么编程。除了正确性外，还有与其程序的兼容性及可维护性要考虑。下面是对软件开发进行行政管理控制的典型例子。

(1)设计标准　包括专用设计工具、语言或方法的使用。

(2)风格标准　包括文件、语言和编码风格(一页中代码的布局，变量名的选择，使用易识别的程序结构)标准。

(3)程序设计标准　包括强制性的程序员对等检查，以及对正确性和标准一致性进行定期的代码审计。

(4)测试标准　如使用程序验证技术、独立测试，以及对测试结果存档以备今后查询。

(5)配置管理标准　控制对已定型或已完成程序单元的访问和更改。

这类标准的制定是通过建立一个公用的框架，使得每个程序员能够帮助或接替别人，从而改善所有程序员的状况。标准还有助于程序的维护，因为维护小组可以在组织良好的源程序中找到需要的信息。

2. 实施程序开发标准

标准必须有效地实施。这想法听起来很平常，但有时却不被管理人员所认识。当一个项目落后于计划或关键人员离开项目组时，通常的反应是强调完成这个项目，而不是强调遵循已建立的标准。

承诺要遵循软件开发标准的公司常常进行安全审计。在安全审计中，一个独立的安全评价小组以不声张的方式检查每一个项目。该小组检查设计、文件和代码，以检验标准是否被遵守。如果知道程序要按常规检查，程序员就不大可能在模块中放入可疑的代码。

3. 职责分隔

银行通常把一项任务分成两个或多个部分，交由分开的雇员来完成。在需要别人合作才能作弊的情况下，这些雇员较少打坏主意。在程序设计中，可以采用这种经验。模块化编程和设计就迫使程序员企图编制非法的程序时，不得不冒合谋的风险。独立的测试小组使模块得到更严格的测试。所有这些分隔的形式提高了程序的安全性。

4. 雇佣特征

一个计算公司在雇佣它的雇员前，通常要进行背景调查。一个公司不想雇佣一个不了解的人。当一个雇员还没有在长期的服务中增强他的可信度之前，公司还将仔细限制其访问权限。

5. 雇员调查

在雇佣一个雇员后，为了安全上的原因，公司通常要令其遵守例外的行为标准。例如：一个银行可能要求它的雇员将银行帐户开在这个银行中，从而审计员能够严密地监视这些帐户是否有解释不清的存款增加或减少。如果财会程序是由一个有赌博习惯的雇员编写的，而他的帐户中存款余额通常又很低，则这个程序将受到仔细的检查。另外，一个银行也可能禁止它的雇员在这个银行开户，这样，一个程序员想抽出资金放入一个帐户而为自己所用就更难一些。当然，在以上两例中，雇员都可以用假名开户，但这样就加大了案发的风险，从而起到阻遏作用。

5.1.6　程序控制小结

本节讨论了两类通常的程序缺陷：泄露或更改数据的程序，以及影响计算机服务的程序。对这些事件有三类基本的控制，即程序员控制限制了编程中的行动，使程序员要想产生一个有恶意的程序更加困难，这些困难还可有效地防止程序员的无意错误；操作系统通过对计算系统中目标的访问，而提供某种程度的控制；最后，行政控制限制了人们所能采取的行动的类型。

仅仅考虑这些控制的保守方面，即它们禁止的行动，是错误的。所有这些控制的积

极方面比起它们禁止的特征来说，更为重要，也更为通用。来自软件工程的程序控制的原来目的，就是要改进软件生产的质量；操作系统限制访问，是要促进程序间信息的安全共享；而行政控制和标准，改善了所生产代码的可用性和可维护性。对于所有这些控制来说，安全特征只是其次要方面。

程序控制是限制一个用户影响另一个用户的更一般问题的一部分。下一节将讨论操作系统在控制用户间交互中的作用。

5.2 操作系统对用户的保护服务

操作系统支持多道程序设计，即支持不止一个用户对系统的同时使用。所以，在操作系统中研究出一些办法，可以防止用户之间有意无意地相互干扰计算。操作系统安全性提供的功能包括内存保护、文件保护、对目标的一般性访问的控制及用户认证。

5.2.1 受保护的目标和保护方法

先回顾一下操作系统保护的历史，从回顾中来说明操作系统可以保护什么，以及有什么保护方法。

1. 操作系统的历史

本来在计算中不存在操作系统，用户通过开关输入其二进制形式的程序。每个用户排他性地独用计算系统，即为各用户编排好使用机器的时间段。用户装好自己的支持程序库——汇编程序、编译程序、共享的子程序，用完机器之后，清除所有的敏感数据。

最初的操作系统只不过是公用程序，被称为“执行程序”，用以帮助单个的程序员和帮助新用户顺利地启用机器。为了重新定位、易于访问编译程序和汇编程序，并为了自动从库中取出子程序装入机器，早期的“执行程序”提供链接程序和装入程序。这种执行程序处理那些为了支持程序员而必须做的冗长乏味的事情。这些程序的主要功能是在执行期间支持单一程序员。

随着多道程序的发展，操作系统的作用开始变化。调度、共享和并行使用的概念进入了操作系统的设计。“执行程序”只是消极地等待用户的请求才提供服务，被称为监控器的多道编程的操作系统却积极地监视所有的计算活动，并恰当地把系统资源分配给多个用户使用，防止用户之间的相互干扰。

2. 受保护的目标

当操作系统要负责控制用户共享的系统资源时，它应保护下列目标：

- 存储器
- 可共享的I/O设备，如磁盘
- 可串行地被重复使用的I/O设备
- 可共享的程序
- 可共享的数据

3. 操作系统的安全措施

(1)操作系统的安全措施可以是下述几方面

1)物理上的隔离。各进程使用不同的物理目标，比如用不同的打印机打印安全级别不同的数据。

2)时间上的隔离。具有不同安全要求的进程在不同的时间被执行。

3)逻辑上的隔离。用户感到不存在其它的进程，因为操作系统不允许程序越界访问。

4)密码技术上的隔离。进行以外部进程不能理解的方式隐蔽其数据和计算活动。

可以将上述 1) ~ 4)条隔离措施组合起来使用。

上述隔离大致按照实现复杂性递增，安全性递减的顺序列出。前两种隔离虽很严格，但会导致资源利用率差。因此，希望不同安全需求的进程能并行执行。

(2)操作系统可以提供几种级别的保护

1)无保护。这种系统适合于敏感程序在分开的时间内运行。

2)隔离。当操作系统提供隔离时，并行的进程彼此不感到地方的存在。每个进程都有自己的地址空间、文件和其它目标，完全避开其它进程的目标。

3)共享一切或无共享。在这种保护方式下，一个目标的拥有者说明该目标是“公开的”还是“私有的”。公开的目标所有用户都可利用，私有的目标只有拥有者自己可利用。

4)有访问限制的共享。利用限制访问进行保护，操作系统检查每次可能的访问是否被允许，对指定的用户和指定的目标实行访问控制。操作系统在用户和目标之间起警卫作用，保证只有授权访问才能进行。

5)按权能共享。作为有访问限制共享的推广，这种保护建立目标的动态共享权。共享的程度可取决于拥有者或主体或计算内容或目标本身。

6)目标的有限使用。这种保护不是限制对目标的访问，而是限制目标在被访问后的使用。例如：允许一个用户看某个敏感文件，但不允许打印出来。更说明问题的例子是允许对某数据库的数据进行统计运算并得到结果，但不允许查单个值(例如：允许得到某级别的平均薪水，不允许查某个人的薪水)。

上述操作系统对共享的支持是按实现难度与保护的精细程度二者递增的顺序排列的。一个给定的操作系统可能会对不同的目标、用户和情况，提供不同级别的保护。

访问控制的粒度也是值得关注的。对于数据，访问控制可以在比特级、字节级、元素或字级、字段级、文件级或者卷级进行。访问控制的粒度越大，访问控制越容易实现。对于用户只需要访问其一部分的大目标，必须允许用户访问整体(例如：允许访问一个文件中的单个记录，就必须允许访问整个文件)。

5.2.2 存储器保护和寻址

多道编程的最明显的问题是如何防止一个程序影响其它程序的存储器。幸好，可以把保护方法制造到硬件机制中，这种机制提供有效的存储器使用，这样就在没有额外开

销的情况下从根本上提供了可靠的保护。

1. 围栏

为了防止有故障的用户程序损坏操作系统的驻留部分，在单用户系统中曾引入了围栏这种最简单的存储器保护方法。顾名思义，围栏就是把用户限制在边界的一边的方法。

围栏的一种实现方式是预定一个内存地址，使操作系统在一边，而用户在另一边。遗憾的是，这种实现办法过于死板，缺乏灵活性。

围栏的另一种实现办法使用了硬件寄存器，常称围栏寄存器，它含有操作系统的末地址。每当用户程序为修改数据而产生一个地址时，便自动将该地址与围栏地址比较。如果大于围栏地址(即在用户区域)，该指令便被执行；如果小于围栏地址(即在操作系统区域)，便显示出错。

围栏寄储器只在一个方向上进行保护，可以保护操作系统不被单用户破坏，但不能防止一个用户对另一个用户的破坏；类似地，用户也不能识别某个程序区是否为不可侵犯的(如程序本身的代码或只读数据区)。

2. 重定位

如果可以假定操作系统具有固定规模，程序员就可以假定程序从某个固定不变的地址开始写它们的代码，这使得容易确定程序中任一目标的地址。但是，如果一个新版本的操作系统大于或小于原有的规模，上述做法就行不通了。

重定位是一个过程，它先假定一个程序是从零地址开始写，再按照程序在内存中定位的实际地址改写程序中所有的地址。在许多实例中，这仅仅涉及对程序中的每个地址增加一个“重定位因子”常数。重定位因子是分配给程序的内存起始地址。

在这种情况下，围栏寄存器可以方便地用来提供重要的外加便利。围栏寄存器可以作为一种硬件重定位装置。围栏寄存器的内容被加到每个程序地址上。这样既重新定了位，又保证了不可能访问低于围栏地址的位置。对于少数需要合法访问操作系统的区域的情况时，可加入特别指令。

3. 基址/边界寄存器

上述围栏寄存器可以重定位的额外便利，在多用户环境中更显重要。用户一多，谁也无法事先知道一个程序将装入什么地方执行。重定位寄存器通过提供一个基址或起始地址解决了这一问题。程序中的所有地址都根据基址修正了。一般称可变的围栏寄存器为“基址寄存器”。

围栏寄存器的困难在于它们只提供一个下界(一个起始地址)，而不提供上界。为了克服这一困难，常常增加第二个寄存器。这第二个寄存器称为边界寄存器，是一种上界地址限制，就像基址或围栏寄存器作为低界地址限制那样。每个程序地址被强制高于基址，因为基址寄存器的内容被加到每个程序地址上，每个地址还要检查以确保它低于边界地址。用这种方法，程序的地址被利落地限制在基址和边界寄存器之间的空间中。

这种技术保护一个程序的内存免于被另一用户修改。当程序的执行从一个用户变为另一个用户时，操作系统必须改变基址和边界寄存器的内容以反映当前用户真实的地址空间。这种改变是操作系统把控制从一个用户转向另一个用户时必须完成的一般性处置

(称为上下文转接)的一部分。

利用一对基址/边界寄存器，当前用户被完好地保护，免受外部用户的损坏；更正确地说，使任一用户免受其他用户错误的干扰。但是，用户地址空间内的错误地址仍然可能影响该用户的程序，一种可能的错误是未定义的变量产生的地址引用在该用户程序的可执行指令区，因而把数据存到指令上，损坏了自己的程序。一种解决办法是再增加一对基址/边界寄存器，一对用于程序的指令(代码)，另一对用于数据空间。虽然使用两对寄存器不能免除一切错误，却切实防止了把指令和数据混淆的错误，并带来了将一个用户的程序分隔成可各自重新定位的更为重要的优点。

可以设想使用更多对定位寄存器的好处，但也不难看出额外的开销。实际的设计是使用两对寄存器。

4. 有标记的结构

基址/边界寄存器建立的是“全部或全不”共享的情况。一种替代办法是“标记结构”，在这种结构中，内存的每个字设有额外的比特标明对该字的访问权。这些限制访问的比特只能被有特权的(操作系统)指令设置。每次一条指令访问该字时都要测试这些比特。

这一保护技术已用于一些系统上。

5. 分段

分段是一项使用了多年的保护技术。分段是把一个程序分成一些分离的片段。每个片段是一个逻辑单位，表示其所含的代码及数据的一种关系。每个片段有唯一的名字。用序对〈名字，位移〉对段内一条代码或数据进行寻址，其中的名字就是段名，位移是该代码或数据相对于所在段的起始址的位移量。

逻辑上，编程员把一个程序想象为一些段的集合。段可以分别重定位，允许一段放在内存中任一可以获得的位置。

操作系统必须维持段名和其真内存位置的一张表。当一个程序产生形如〈名字，位移〉的一个地址时，必须在段目录中查找“名字”确定其实际的内存地址，操作系统将“位移”加到该地址上，找出该代码或数据的实际内存地址。为了提高效率，通常对每一个执行的进程都有一张操作系统段地址表。希望共享访问某一段的两个进程可在其段表中有相同的段名和地址。

(1)用户程序并不知道实际使用的真实内存地址。这种对地址的隐蔽对操作系统有三个优点

1)操作系统可以把任一段移到内存的任一位置。既然所有的地址引用都由操作系统利用一张段地址表进行翻译，当段移动时，操作系统就需要更新该表中相对应的地址。

2)如果一个段当前不使用，可以将其从主内存中移出另存。

3)每一个地址引用都通过操作系统传递，从而为了保护也就有机会检查每一次地址引用。

因为有第3)个特点，在其表中没有某个段名的进程就被拒绝对该段的访问。操作系统以此对内存保护提供了一种有效手段。

有可能对任一用户的任一段分配不同的保护类。分段过程由硬件与软件的组合来处

理。

(2)关于保护、分段的优点

1)为了保护而检查每一次地址引用。

2)可以为许多不同的数据项分配不同的保护级。

3)两个或多个用户可以共享对同一个段的访问，但他们具有不同的访问权。

4)用户不可能产生一个地址或访问未经允许的段。

由于位移量可能超过段的长度，这是一个安全隐患，具体实现时需设法解决。

(3)有效实现分段还有两个困难：

1)段名在指令中不便于编码，在名字表中操作系统查表可能很慢。为了克服这一困难，常在编译时将段名转换为数，编译器还要附加一张与实际段名相匹配的链接表。当两个程序要共享一段时，段名对应的数必须相同，实现起来会增加复杂性，降低效率。

2)因为段有各种长度，过了一段时间之后，未用的破碎空间难以使用。一种解决办法是重排内存，但重排内存和相应表的更新都要耗费时间。

6. 分页

分页是与分段有相同与不同之处的办法。像分段一样，每个地址是一个序对〈页号，位移〉决定的。内存被分为一些规模相同的单位，称为“页帧”(为了实现方便，页帧的规模通常选 512 ~ 4 096 字节的某个 2 的方幂)。程序被分为等规模的片断，称为“页”。这样做的优点是不会有未用空间支离破碎的问题；缺点是每页不大可能再像段一样有内在的逻辑上的含义，从而无法按内涵分别保护。

7. 将分页与分段相结合

因为分页与分段有各自的优点，目前已有将此二者结合使用的实用系统。先分段，再分页；这就对每个地址增加了一层翻译。由于增设了相应的硬件，改进了实现的效率。

5.2.3 对一般目标的访问保护

内存保护是更一般的目标保护问题的特殊情况。由于多道编程的发展，共享的目标的数量和种类都增加了。下面是某些要保护的目标：

- 内存
- 辅助存储设备上的文件或数据集
- 内存中执行的程序
- 文件的目录
- 硬件设备
- 数据结构，如堆栈
- 操作系统的表格
- 指令，尤其是特权指令
- 口令和用户认证机制
- 保护机制本身