第 4 章　操作系统安全

正如我们所想的那样，很多用户使用计算机进行计算，而不是用自己的大脑计算。

早期，我们将安全定义为阻止对手对系统重要属性的破坏。“ 系统 ”主要由“ 计算 ”过程组成。为了破坏计算，对手必须能够通过某种方法进入系统，如通过某种输入或者一些其他的计算过程。计算机使用操作系统来执行计算指令和控制。因此，操作系统很容易成为敌我双方开展安全攻击和防御的场所。

本章将探讨操作系统安全。

- 4.1 节介绍操作系统的背景知识。
- 4.2 节介绍操作系统安全的基本概念。
- 4.3 节介绍现代操作系统行为的复杂性问题。
- 4.4 节讨论操作系统复杂性所带来的安全问题。
- 4.5 节讨论操作系统安全的发展方向。

4.1　操作系统的背景

本章首先简单介绍一下操作系统的背景知识。

4.1.1　计算机体系结构

不论相信与否，过去人们在使用计算机时需要用到烙铁和绕接工具，并且需要知道 16 进制转换器的基本原理。在过去的几十年中，计算技术取得了很大的进步，所开发出的计算机系统已经在抽象层次上符合用户对计算机的直觉观念。而操作系统在抽象层次上扮演了十分重要的角色。但是，我们需要对操作系统的各个层次有充分了解，才能够深入地认识它的价值。因此，我们将首先探讨操作系统最底层的部分：硬件。

计算机采用二进制方式操作数据。在最底层，所有的操作和数据都使用二进制数字表示，二进制数字的每位都是一根金属线，该位的值用金属线上电压的高低来表示。CPU 和内存使用总线传输数据(参考图 4-1)。物理内存是一个字节数组，每个物理地址表示某一字

节的位置。我们可以将物理内存的每个字节看成一个八位长的寄存器：8 个触发器(对应静态 RAM)或 8 个电容器(对应动态 RAM)。并用简单布尔门开发解码器来寻址，该解码器能够根据一个 AND 门和 N 个转换器识别一个 N 位的地址。因此，内存就是一个由单字节寄存器构成的数组，每个寄存器都有一个解码电路，以便于读写该字节。当从内存中读取某一字节时，CPU 将地址放在地址总线上，然后在控制总线上发出一个读信号，内存将发送该字节。当向内存中写入某一字节时，CPU 将地址放在地址总线上，将该字节的值放在数据总线上，然后在控制信号上发出写信号，内存载入该字节。

现代系统中的程序，从基础层面看，是由一系列的机器指令组成的。这些指令从字面上看是驻留在内存中由操作码和数据组成的二进制值。CPU 有一个内置的程序计数器寄存器，用于存储下一条指令的地址。CPU 将该地址放在地址总线上，取出指令，然后使用内部逻辑电路解码指令，分析指令的功能(如将寄存器 B 的内容移动到由操作码后面 4 个字节表示的内存地址中)并执行该指令。CPU 的逻辑电路在该条指令执行完成后，将程序计数器的内容更新为下一条要执行的指令地址。

图 4-1 在物理计算机体系结构的基本模型中，CPU 通过总线来读写内存

上述过程就是计算过程。实际上，所有用户点击鼠标的操作、在文字处理器中输入文字或者运行大型应用的过程都可以归结为上述基本过程。

正如前面所述，我们所介绍的是一个简化的模型。实际上，由于使用了各种优化措施来提高性能，现代计算机中的这一过程更为复杂。例如，CPU 现在通常采用流水线的方法执行指令，在上一指令完成之前就读入并执行下一指令，并且采用很多数据和指令缓存措施。但是，所有这些也仅是优化，本节我们所描述的是基本思想。

4.1.2 操作系统的功用

在非常低的层次上处理计算是一件令人烦恼的工作，尤其是在允许各种复杂任务同时执行的现代计算环境中。操作系统的出现解决了此问题，它能够更加方便高效地创建和协同所有这些任务。从专业的角度来看，操作系统可以在以下三个方面带来方便：

(1) 从低级工作中解脱出来。操作系统可以避免程序处理同样低层次任务的繁琐细节问题，还可以避免程序被移植到其他机器上时导致的低级指令兼容性问题。例如，文件的读写对很多程序来说都是非常普遍的任务，然而，所有读文件某一部分的操作都包括以下几个步骤：定位文件在磁盘中的位置，定位读取部分在文件中的位置，如何读取该部分。如果能够提供一个可以处理上述步骤的公共服务，那么问题将简单得多。

(2) 多道程序设计。Andrew Birrell 曾经指出，“ 人类非常擅长同时处理两三件事情，如果计算机不能处理这么多的事情，似乎有点不妥 ” [Bir89，p.2]。操作系统可以使一台计算

机很容易地同时处理很多事情。操作系统能够实现程序间的切换，从而提高了方便性和效率。如果程序在等待某一磁盘数据传输的完成，那么 CPU 可以执行另外一个程序，如果 CPU 能够快速地在两个程序之间切换，那么用户就可以在某一窗口运行长时计算的同时，在另外一个窗口中输入文字。

(3) 进程隔离。当计算机同时执行多个任务时，操作系统能够避免它们相互之间的非预期影响。由于本书主要关注安全，因此我们首先想到的是避免恶意实体的破坏。但是，隔离在普通情况下也是非常有用的，例如，程序员可以不用再关心其他程序破坏自己的内存，或者引起程序异常。

4.1.3　基本元素

操作系统通过一些基本元素，在硬件支持的基础上来达到目标。

用户模式和内核模式　现代 CPU 通常运行在两种模式下：

(1) 内核模式，也称为特权模式，在 Intel x86 系列中，称为核心层(Ring 0)。

(2) 用户模式，也称为非特权模式，或者用户层(Ring 3)。

如果 CPU 处于特权模式，那么硬件将允许执行一些仅在特权模式下许可的特殊指令和操作。一般看来，操作系统应当运行在特权模式下，或者称为内核模式下：其他应用应当运行在普通模式，或者用户模式下。然而，事实与此有所不同，我们将在后面讨论。

显然，要使特权模式所提供的保护真正有效，那么普通指令就不能自由修改 CPU 的模式。在标准的模型中，将 CPU 模式从用户模式转到内核模式的唯一方法是触发一个特殊的硬件自陷，如

- 中断：一些外部硬件引发的，如 I/O 或者时钟
- 异常：如除数为零，访问非法的或者不属于该进程的内存
- 显式地执行自陷指令

与上述行为的处理过程基本相同：CPU 挂起用户程序，将 CPU 模式改变为内核模式，查表(如中断向量表)以定位处理过程，然后开始运行由表定义的操作系统代码。

内存管理　如果要避免用户进程相互影响，那么最好的方法是避免它们互相从对方的存储区域读写数据，同时还应当避免读写操作系统的内存区域，其中也包括操作系统代码。系统达到该目的方法是在 CPU 和系统其他部分之间的地址总线上嵌入 MMU(Memory Management Unit，内存管理单元)(参见图 4-2)。CPU 发射合适的地址，MMU 单元负责将逻辑地址转换成物理地址，操作系统使用特殊指令来建立和控制这种转换关系。

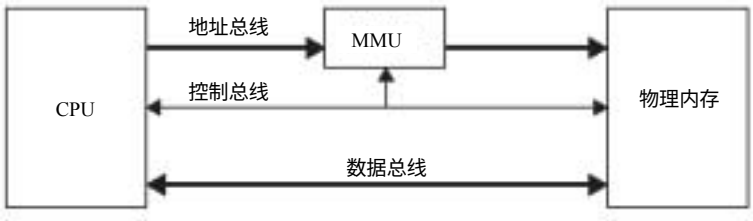

图 4-2　在现代系统中，MMU 将 CPU 发射的虚拟地址转换成实际的物理地址。这种控制和改变转换关系的能力使操作系统能够为每个进程分配自己的地址空间，并避免用户代码访问不该访问的数据

内存管理的概念还带来了其他便利，从运行在 CPU 上的程序的角度来看，内存被当成一大片地址连续的物理存储空间来使用。但是，MMU 通常将内存分为很多页(page)或者帧(frame)。页由很多逻辑地址空间组成，在物理内存中不必按序显示。实际上，有些内存页可能甚至根本不存在物理内存中。

MMU 的转换机制也可以使我们能够标记某些内存区域仅是可读的(对用户代码而言)，如果用户级的 CPU 试图发出一条写入该地址的指令，那么 MMU 将拒绝转换地址，同时将引发异常，从而进入内核级。

与此类似，MMU 可以避免用户代码读取不该访问的内存。内核创建仅由它自己才能查看的内存区域，并使用该片内存来存储用户进程维护数据、I/O 缓冲等。然而，如何完成这一过程依赖于硬件体系结构和操作系统的设计选择。图 4-3 表示了一个经典的示例过程。每个用户进程有自己的地址空间，为保证所有的进程地址空间中内核私有的内存地址相同，需要建立内存管理单元表。当用户进程切换到内核模式时，不需要改变内存管理单元表，并且，内核级代码可以很容易地查看和改变用户数据。其次，内核私有内存到用户地址空间的这种匹配关系在不同的操作系统中也不同。另外，这只是一种非常经典的方法，但不是必需的，在某些系统中，如 SPARC 系统上的 Solaris，内核有自己的地址空间，但如果内核要访问用户内存。需要额外的措施。由于能够完全在用户进程之间切换并且能够转换其可查看的地址空间，这使得进程可以被看做是由操作系统控制和管理的基本元素。

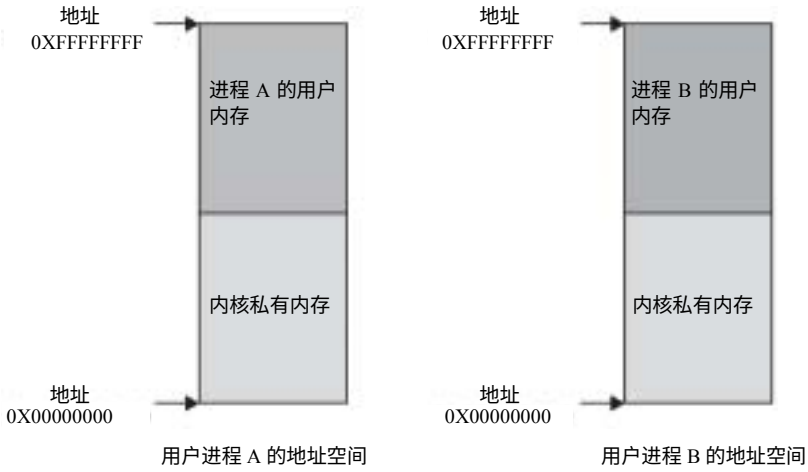

图 4-3 在典型情形中，每个用户进程都有自己的地址空间，但是内核私有内存映射到进程地址空间的同一部分(必须指出，该处简化了内核栈表示)

很多现代系统也允许操作系统将内存页迁移到后备存储设备上。通常为硬盘，尽管出现了更新的技术。由于大部分进程并不是总使用所有的内存，该技术可以极大地提高性能，不过这依赖于我们评价性能的方法。

系统调用 当用户代码需要请求操作系统提供的服务时，通常采用系统调用的方法来完成这一过程：用户代码将在约定的地方(如 CPU 寄存器)存储一个值，该值能够代表它所请求的服务。如果服务需要参数，则用户代码采用同样的方法将其存放在约定的地址。然后用户代码发出一条自陷指令。与其他类型的中断相同，系统做出响应，保存用户进程的

状态，记录程序计数器、其他硬件寄存器、进程栈以及其他进程数据。硬件切换到内核模式，执行系统调用处理程序，该函数通常需要查找服务表以便于找到所请求服务的处理子过程入口。上述整个过程称为上下文切换。一旦所请求的服务完成，就恢复用户进程的状态，控制重新交给进程。

然而，程序员可能经常在没有意识到是在自陷指令执行的情况下使用服务，例如，使用 fork()或者 exec()。原因可能是程序员在请求这样一个服务时，通常使用的是库函数(预编译了的用户代码，可以链接到以后编译的程序中)，库函数负责调用自陷指令(如图 4-4 所示)。这种方法使程序员很难判定它们在手册中查看和调用的子过程是库函数？系统调用？还是两者都不是。

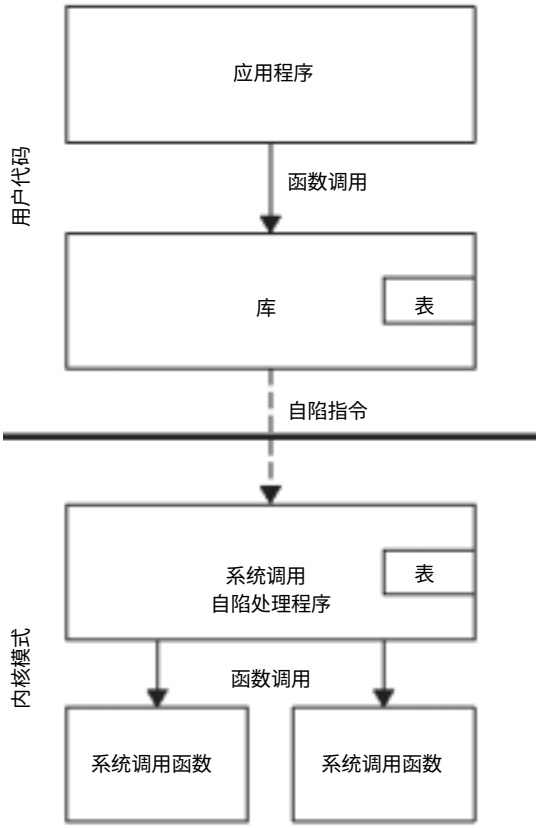

图 4-4　为实现与操作系统的交互，用户空间的应用级代码通常将函数调用预编译到库中，用户空间中的进程交互也采用这种方法。然后这些库函数发出自陷指令来实现系统调用，自陷指令将导致 CPU 切换到内核模式，用户代码被挂起，进入系统调用处理程序，由该函数负责调用相应的内核函数。通常情况下，库和系统调用处理程序使用函数表指针判定处理程序的入口

簿记　为了管理所有进程，操作系统必须做很多簿记工作，如跟踪正在运行的进程、阻塞的进程等等。用户代码可以执行一些系统调用来查看某些簿记。有些操作系统提供了另外一些查看簿记的方法。例如，Linux 文件系统的/proc 目录下有一批伪文件，每个进程有自己的子目录，每一个都包含该进程的簿记和资源使用信息。

用户交互 用户通常采用两种方法与计算机系统进行交互：通过命令行 shell 和通过 GUI(Graphical User Interface，图形用户界面)，这两种方法都不是操作系统的必要部分。例如，本身通常就是一个用户进程，获取用户输入的命令，然后执行相应的操作；有时候 shell 直接执行命令；有时，shell 到预定义的位置查找特定名称的程序。在发现该程序之后，shell 使用操作系统调用创建一个新的进程来运行该程序。GUI 的过程与上述过程类似，但可能复杂了一些。

通常，人们讨论操作系统均是指与它进行交互的方式，需要注意的是界面并不是内核自身的必要构成部分，它是 shell 或者 GUI 包的构成部分。

4.2 操作系统安全的基本概念和原理

4.2.1 进程隔离和内存保护

如 4.1 节所述，操作系统的一个主要功能是提供进程隔离。我们也观察到，该功能除了安全，在其他方面也是有益的。但是，如果讨论安全性，进程隔离则是操作系统在安全性方面的主要贡献。

进程隔离在有些方面的思想非常简单直接，如通过内存管理禁止进程读写其他进程的内存来实现隔离。但是，即使是这么简单的思想在实现上也有一定难度。例如，如果用户进程 A 能够重写内核内存的某些部分，那么进程 A 就可以重写操作系统的这些部分，而操作系统禁止进程 A 访问进程 B 的数据。在另外一些情况下，此问题更为复杂。例如，使系统拥有物理内存两倍大的虚拟内存，这样做的结果是同一物理字节在物理地址空间有两个地址。

在实际应用中，我们可能不希望完全地隔离进程，希望它们有一定的交互，这种考虑使进程隔离问题更为复杂。我们希望进程能够通过一些标准机制来进行交互，如通过流水线(pipeline)、消息传递、或者其他形式的进程间通信(IPC)；希望进程间能够共享内存；希望多个进程能够读写某些同样的文件。上述考虑都与实现进程间的完全隔离相违背，使进程隔离问题变得更为复杂。在什么样的情况下允许进程 A 将部分进程 B 地址空间映射到自己的地址空间？如果内存管理以页为粒度划分内存，那么当进程 A 请求的内存小于一页但操作系统共享内存等于一页时，系统该怎么做？另外一个例子，进程 A 发送消息给进程 B，但进程 B 在收到消息前退出，这种情况该如何处理？我们知道，如果新的进程 C 与旧的进程 B 具有同样的标识，那么进程 C 将收到该消息，正如公寓中新的居住者将收到旧居住者的信件一样。在开发系统的过程中，开发者可能不会想到此问题，但在使用时，此问题就会暴露出来，这时在重新发现和删除未发送的消息说起来很容易，但是做起来则比较困难。所以，十全十美是很困难的。

正如 2.3 节所讨论的那样，使用其他方法可以破坏进程隔离。如内存帧或者内核堆这样的对象重用可能会泄露信息。进程可能通过一些可观察的变化(如能判定内存或者缓冲中的共享代码可能是什么内容)来推导其他操作相关的信息。

最后，不要忽略现代操作系统代码量巨大的问题，操作系统有很多潜在内部配置和外部输入场景假设，理论上来说，输入场景是无穷的。操作系统的测试难度要远大于某个程

序的测试，这类程序在用户提供输入后，处理，然后终止。即使是定义操作系统的正确状态也是非常困难的！一架波音 747 喷气式飞机有六百万个部件，但是 Windows XP 则有四千万行代码(有人据此认为膝上型电脑的操作系统的复杂度要超过六架喷气式飞机)。所以，即使设计准则是合理的，仍然可能会出现非常多的 bug。

再次强调：十全十美是困难的。

4.2.2　用户

1.2 节讨论了使用访问控制矩阵来建模和管理系统安全的问题，即谁可以对谁做什么？为了管理访问，操作系统首先需要知道“ 谁 ”这个概念。

根据传统观点，“ 谁 ”通常指的是用户。系统为用户维护账号或者身份，简单地说，每个账号属于一个真实的用户，该用户使用密码或者其他身份认证技术进行认证。(9.2 节将深入讨论这一问题)。“ 用户=人类 ”的观念快速推广到一组人(如一个班级的人)，尤其是非人类的实体，如客户或者游戏[1]。

很显然，有些用户必然拥有超出其他用户的权限。有些系统采用创建特殊账号的方法来实现这一点，该账号(通常称为根、超级用户、管理员)拥有超过其他账号和系统的最大权限。其他一些系统则采用了不同的方法来标记拥有管理权限的特殊用户。

总处于最大权限级别是非常危险的，因为在该级别的错误或者攻击可能会带来很大的破坏。例如，当与一个坏的邻居进行现金交易时，我们只需要带足刚好够的现金去进行交易即可，计算机安全人员称这一准则为最小权限准则(它也是 Saltzer 和 Schroeder 准则之一)。因此，系统可能会在操作时，强制拥有特殊权限的用户仅能执行普通权限的操作。然后在需要的时候再赋予其全部权限，全部权限的获取通常是在一个弹出窗口中输入特定管理员密码或者调用 sudo 命令来实现。

计算机系统在使用时可能会与上述用户的要求有所不同。单用户膝上型电脑可能配置为启动时就进入用户账号，而不需要登录；普通终端用户在个人电脑上执行操作时经常需要管理员权限，导致所有的用户在默认状态下都是管理员。Web 服务器软件经常配置为以最大权限运行，这样任何破坏了系统的攻击者都可以获取最大权限。

4.2.3　文件系统访问控制

谈及“ 谁对谁做了什么 ”这一问题时，也需要讨论用户执行行为的对象。在典型的现代系统中，我们通常称该对象为文件系统。现代系统中的数据和程序都表现为文件。有些程序文件是系统功能的关键部分，如执行 shell 关键命令的程序；有些数据文件是系统配置的关键部分，如/etc/passwd。伪文件，如显示在/proc 和/dev/kmem 上的文件，也非常重要。因为伪文件使进程间的交互可以通过标准文件系统接口来实现。

典型的文件操作权限是读、写和执行，对于目录而言，执行意味着进入子目录。需要说明的是上述权限仅是典型的权限，不是全部权限。系统也可能定义更为详细的权限，一个很好的例子是 UNIX 系列操作系统的 setuid 权限。它允许用户以程序拥有者的权限来运行该程序，但该权限也仅限于运行该程序。该权限在文件更新时非常有用，任何人都可以

[1] 我们中的一员的第一次编程实践就是攻破社区大学大型分时游戏系统上的游戏账号。

更新该文件，但是必须以一种非常谨慎的方式更新文件。例如，假设一个文件记录了游戏玩家的最高分，如果任何人都可以随时写这个文件，那么没有人会信任该文件所报告的分数。企业系统的管理员通常使用类似 setuid 权限的技术来允许特定用户以根用户的权限执行特定任务。管理员可以使用程序或者 shell 脚本来完成这一任务，使其拥有者为 root，但是赋予其 setuid 权限，用户可以在需要的时候拥有 root 权限。但是，这种技术可能有负面作用，攻击者可以通过欺骗该程序来以 root 权限执行他自己的代码。

1.2 节描述了访问控制矩阵问题，并讨论了两种方法：基于列的方法为每个客体存储一个访问控制列表(ACL)；基于行的方法则存储每个主体访问能力(capability)。现代 Windows 系统均使用 ACL 方法。系统中的每个客体，如目录、文件、网络共享等等，都有一个相应的访问控制列表。ACL 是访问控制表项(access control entry，ACE)的列表，该列表包含：用户或者组；操作，如读或者写；权限，如允许或者禁止。当 Alice 试图操作某个客体时，如打开某个文件，内核检查该客体的 ACL，以判定该操作是否允许。如果禁止 Alice 或者她所属的组访问该文件，那么该检查将立即完成，Alice 被禁止访问。否则，系统将检查 ACL 中的所有访问控制入口，以判定 Alice 所拥有的权限集合。如果她所尝试的操作在其权限集合内，那么就允许该操作，否则拒绝该操作。

相反，UNIX 系列的系统，如 Linux 和 OSX(基于 FreeBSD)，则使用混合访问控制方法。每个文件都有一个拥有者和一个组。文件的拥有者可以改变其所属的组；组由很多用户组成；一个用户可以属于多个组。每个文件拥有三个权限集合：拥有者权限集合；所属组权限集合；任何其他人的权限集合。每个集合都是下面集合的子集

{ “ read ”, “ write ”, “ execute ” }

当用户 A 请求操作该文件时，操作系统将顺序检查这些集合。

- 如果用户 A 是文件的拥有者，那么由拥有者权限集来判定其是否可以执行该操作。
- 否则，如果用户 A 是文件所属组的成员，那么由组权限集合判定该问题。
- 否则，根据任何其他人的权限判定该问题。

UNIX 系统所采用的方法与 ACL 类似，每个对象都有子集的权限列表。但是，此方法也有些类似权限能力，组成员身份可以赋予某个主体特定的访问权限。需要指出的是，在这两种方案中，管理员或根账户拥有所有这些权限的检查权。也就是说，超级用户拥有完全控制系统中所有对象的能力。

上述方法仅是经典的做法，不是所有系统都需要采用的方法，也不是适用于所有应用场景的正确方法。例如，回忆 2.3 节所描述的术语，我们可能不满足于任意访问控制策略，它使用户能够自由决定他所拥有的对象的访问控制权限；我们可能更希望系统实施强制访问控制，以达到更高的安全性，如多级安全 MLS。另外一个例子，传统 UNIX 文件系统的权限[2]表达能力非常差，我们可能更希望系统采用其他模型，如 RBAC 或者长城模型(参考 2.2.3 节)。

[2] 需要指出的是，一个我们认识的 UNIX 发烧友坚持认为：标准的权限加上 setuid 能够处理所有有用的场景。

4.2.4　引用监视器

计算机科学经常鼓吹将策略与机制分离。在本书中，我们不是很赞同将这两者分离的观点。操作系统设计可能很好地定义了主体、客体、访问控制，但是，如果在系统开发完成后，不能很好地检查访问许可，那么可能就不能成功地实现预期目标。如 2.3 节和 3.4 节所讨论的那样，系统执行完备性检查，检查每个动作是否许可是非常必要的。操作系统可以将所有检查在一个重要模块中实现，该模块称为引用监视器，它是美国国家安全局安全增强 Linux(SELinux)的主要贡献之一，SELinux 在 Linux 中引入了引用监视器的形式化可嵌入框架[LS01，NSAa，WCS+02]。

4.2.5　可信计算基础(TCB)

从某种意义上来说，操作系统中的所有保护措施都是在转移和减少信任问题。例如，相信个人计算机大部分都可以正确地、无恶意地运行是非常危险的。系统是一个代码和用户组成的庞大集合，谁也不知道如果某个部件产生错误将会怎样。然而，我们可以使系统尽可能的减小该集合，也许我们只需要相信操作系统和 CPU/ MMU 能够正确运行，就不必去假定用户进程能够正确地和无恶意地执行操作。最好的做法是，使我们必须相信的事物集合尽可能小并处于控制之中。

正如 2.3 节所讨论的那样，这一概念称为可信计算基础(trusted computing base，TCB)：我们必须相信的事物集合。但是，如果我们相信可信计算基础，那么我们就没必要相信其他事物。如前面例子所示，计算机的操作系统至少在直觉上看起来是可信计算基础的自然组成部分。实际上，可信计算基础包含的范围可能更大。例如，操作系统允许任意设备驱动器运行在内核模式下，可信计算基础在工程上也难以满足信任的要求。

4.3　真实操作系统：几乎实现了所有功能

真实操作系统远超我们所描述的简单抽象模型。操作系统曾经仅由内核、驱动、命令解释器和文件系统组成。现代操作系统则包含了大量各种各样的程序、服务，以及为用户提供便利的工具。很多操作系统都提供了函数库，不只提供满足程序员系统调用以及方便字符串复制的过程，还包括远程过程调用、密码学等更多功能。有些提供商提供的应用捆绑，如在操作系统上捆绑 Web 浏览器，甚至违反了美国政府的相关规定。由于操作系统面向的用户群体非常广泛，包括普通用户、应用开发者和硬件制造商，它们已经成为了“瑞士军刀”式的多功能软件。信任也随之而消失。

4.3.1　操作系统的访问

操作系统为应用程序开发者提供了用以完成某些任务的服务，而用户级应用没有执行这类任务的权限，或者不能执行。这类服务包括进程间通信机制(IPC)和内核级数据结构访问方法如信号量、内核级线程和共享内存。由于应用程序有时要超过操作系统限定的边界，如内存保护。因此，操作系统需要提供某种机制以协助这种操作。

如 4.1 节所述，用户进程与操作系统之间的通信通常采用系统调用的方法，应用可用的系统调用集合构成了操作系统的接口。该接口在不同的操作系统中也不同，尽管人们已经就该接口的标准化工作付出了大量的努力。

POSIX 在 20 世纪 80 年代中期，IEEE 定义了一套 UNIX 操作系统的接口，并将该系列标准称为可移植操作系统接口(POSIX)[IEE04]。那时，甚至今天，存在很多版本的 UNIX 操作系统，在某一版本上编写的程序在另外一个版本上的系统不见得能够运行，主要原因是不同操作系统提供的 API 不相同。POSIX 的基本思想就是使在某一操作系统上编写的代码可以在另一系统上运行。

POSIX 标准有多个部分，不同操作系统可以自由选择不同的部分来实现。即使是 Windows NT 内核也仅实现了 POSIX 标准的一部分。Linux(在该上下文中是指 Linux 内核和 GNU 工具)也有自己的标准，提供跨 Linux 版本的兼容性，尽管 Linux 并不严格遵循 POSIX 兼容标准，但 Linux 标准的很多部分都与 POSIX 保持一致。

Win32 应用程序接口 Windows 操作系统在应用程序和系统调用之间提供了一个抽象层，即 Wind32 API。Wind32 一词源自支持 Intel 的 IA32 体系结构的 Windows 操作系统(为了将此与 4.1 节中的背景概念相关联，我们特指明 Win32 API 为库)。

该抽象层允许 Microsoft 随意地改变系统，而应用程序开发者则不需要改变。当然，当 Win32 API 自身改变时，应用开发者就必须改变。但 Microsoft 通常会保持某些接口的一致性以避免引起标准应用程序的更新。

尽管 Win32 API 的概念与 POSIX API 的概念相似，但它们在语义上是不同的。尽管它们都有 C 语言风格的函数语义，但是，Win32 API 需要使用大量的特定数据结构和技术。一个例子是允许同一 API 被调用很多次，每次调用的参数也不同，这样做可以便于应用程序获取需要分配的目标缓冲区大小信息。相关文档可以在 Microsoft 网站上获取[Mic06]。

4.3.2 远程过程调用支持

操作系统经常绑定的另外一个功能就是远程过程调用(RPC)。远程过程调用开发于二十世纪七十年代晚期和八十年代早期，它允许其他机器像调用本机的一个标准过程那样调用一台机器上的代码[Nel81]。在底层，RPC 库截取过程调用，封装该调用，排列[3]参数，然后将请求发送到目标机器。在目标机器上，远程过程调用库解装该调用及其参数，将其发送到预定目标。当过程处理完成之后，目标机器上的 RPC 库封装返回结果，将其发送到调用者机器。

理论上，程序员可以在每次需要从本机调用其他机器上的过程时，编程实现整个过程。但实际上，整个过程比较繁琐，并会隐含许多 bug，而这些 bug 并不是程序员主要目标的构成部分。如果由程序员负责调用该过程，编写该过程，由编译构造环境负责其余部分，那么事情将会简单得多。这就是 RPC 库所具有的功能。

[3] 当调用一个局部过程时，该过程希望其参数能够以特定方式存储在内存中。为了通过网络发送参数，我们需要遍历所有的局部排列，然后将其转换成线性字节序列，该过程称为排列（marshaling）。在另一端，我们对其进行复原。

RPC 的功能并不是操作系统的必要组成部分，但是远程 RPC 库通常与操作系统绑定。由于网络协议栈通常由操作系统负责监控，因此，将 RPC 服务放在这里是合理的。我们将在后面讨论，这种方法同时也为攻击者提供了机会。

DCOM　今天主要使用两种远程过程调用技术：UNIX 系统系列的 SunRPC 和 Windows 系统系列的 MSRPC(后者基于 DCE RPC)[Ope97，Sun88]。除了 MSRPC，Microsoft 系统也支持称为组件对象模型(Component Object Model，COM)的远程过程调用技术。COM 的基本思想是允许开发者使用自己熟悉的语言编写软件组件，发布组件，然后由其他开发者使用该组件，其他开发者所使用的编程语言可能与组件的编程语言不同。

COM 允许来自不同团体的开发者共享他们的组件，一旦该组件安装在他们的机器上就可以使用。Microsoft 后来又开发了分布式组件对象模型(DCOM)，允许开发者使用远程主机上的 COM 组件。分布式组件对象模型基于 MSRPC 开发，允许访问远程主机所提供的服务，同时其安全性也受到了研究者的关注。

4.3.3　密码学支持

最近几年，操作系统开始通过系统调用和系统级 API 来提供密码学服务。密码学应用程序接口(Cryptographic API，CAPI)是 Microsoft Win32 API 的一个子集，它为开发者提供了一套密码学处理程序和证书管理函数。Windows 操作系统甚至内嵌存储了证书和私钥，使用称为 Windows 注册表[4]的特殊存储系统来存储数据(第 10 章将详细探讨证书和私钥)。操作系统的所有这些功能为应用开发者带来了极大的便利。但是，对安全研究人员来说，则需要考虑新功能所带来的新的安全问题(第 8 章将探讨 CAPI 可能带来安全问题[MSZ03，MSZ05])。

在操作系统中整合密码学操作的部分原因是满足 Microsoft 浏览器 Internet Explorer(IE)加密的需要，由于需要支持安全套接层(Secure Sockets Layer，SSL)，浏览器是最早需要密码学支持的。没有操作系统的支持，浏览器需要自己实现密码学子系统。例如，Mozilla 浏览器有较为完善的密码学子系统以支持 SSL 和 S/MIME(后者用于加密 MIME 编码的消息，如电子邮件附件)。

Linux 系统则较晚才采用了系统范围内的密码学支持。到作者编写本书为止，最新内核版本即成为第一个支持密码学应用程序接口(CAPI)的版本。为了便于使用，Linux 内核的密码学应用程序接口(CAPI)不仅为内核代码如网络协议栈、其他模块等提供密码学服务，也为内核外的应用程序代码提供密码学服务。尽管 Linux 密码学支持不提供证书存储，但是提供了实现文件系统安全和完整性的必要工具。

4.3.4　内核扩展

到目前为止，我们已经讨论了一些绑定到现代操作系统的公共服务，这些服务通常由提供商封装到操作系统中，或者是由维护操作系统的开发者团体提供。除了这些服务，操作系统还提供了允许任何人扩展操作系统的 API。该技术最常用的用途是允许第三方开发

[4] 注册表是一个存储系统，很多 Windows 应用、操作系统自身都使用注册表来存储配置信息，注册表与*nix 系统中的/etc 目录大致上起着相同的作用。

者开发外围硬件设备驱动，如打印机、网卡、声卡等。由于驱动可以访问内核级的数据结构，并能够以内核权限执行。因此，这类 API 也为攻击者提供了一种攻击操作系统的途径(4.4.3 节将讨论这类攻击)。

Windows 操作系统为实现设备驱动提供了一套良定义的 API。当用户安装设备驱动时，它首先注册到硬件抽象层(Hardware Abstraction Layers，HAL)，该层支持操作系统直接使用该设备而不需要知道该设备的细节，毕竟细节问题是驱动程序所要做的。

Linux 系统的设备驱动要么直接编译到内核，要么以动态可装载内核模块的方式实现。像 Windows 设备驱动一样，Linux 内核模块实现了一套良定义的接口。但与 Windows 系统不同的是，Linux 内核模块可以动态地从内核装载和卸载，而不需要重启。对于这两种系统来说，适当的特权用户通常可以编写和装载内核模块，从而可以有效地访问内核执行线程及其数据结构[5]。

4.4 针对操作系统的攻击

作为计算机系统的中枢程序，操作系统担当着很多角色。它负责执行用户程序，并避免它们相互影响。另外，它为系统中的其他程序提供主机服务。最后，它存储和保护存储在文件系统中的信息。

从攻击者的角度来看，操作系统成为攻击目标的理由很多。首先，如果计算机中保存有攻击者所需要的信息，如公共信息或者军事数据，那么操作系统就是最后的防线。第二，操作系统也可能成为攻击者攻击其他系统的跳板。例如，攻击者在攻破目标主机的操作系统后，就可以通过该主机向目标网络发送大量的格式错误或者恶意的网络数据。第三，操作系统提供了隐藏远程攻击者攻击踪迹的方法。例如 Carlo 可能首先攻破 Bob 的机器，然后从 Bob 的机器上攻击 Alice 的机器，如果 Carlo 做得好，那么调查人员就会相信 Bob 是攻击者。本节将讨论针对操作系统的攻击模式。

4.4.1 通用攻击策略

不论攻击者的动机是什么，其目标通常是获取拥有者(own)或者根用户(root)的权限。也就是说，攻击者能够完全控制程序的安装和运行，能够访问文件系统中所有的文件，能够修改用户账号。有时，如果幸运的话，攻击者可能仅需要利用一个隐患就可以达到该目标(假设是操作系统的隐患，如 DCOM 漏洞)。有些时候，攻击者需要付出更多的努力才能达到目标。比如仅能够获取低级访问权限用户账号的攻击者必需提升自己的权限等级才能够达到目标，获得目标机的拥有者权限。

需要提醒的是，权限等级有其自然的顺序：“低权限级别”权限小，“高权限级别”权限大。计算系统通常将权限级别或者优先级编码为一个数字，数字也有自己的顺序：≤和≥。不幸的是，这些顺序的对应关系有两种，究竟是低权限级别对应低数值还是高数值？需要注意的是，在真实系统中，这两种方法都在用。

[5] 将 Windows 配置为不加载任何非 Microsoft 签署的驱动是可能的(至少从理论上是可能的)，但是默认行为是仅发出一个警告消息给用户。

如我们所讨论的那样，现代操作系统提供了很多服务，有些服务甚至运行在操作系统的最高权限级别。很多服务，如 DCOM，都要使用网络，这样就给远程攻击者提供了一种访问目标机操作系统的方法。有一类攻击就是远程攻击者利用这类服务，发送自己的程序到目标操作系统并执行造成的，这种攻击模式称为远程代码执行(remote code execution)攻击。

另外一种攻击方法是攻击者在目标机上安装按键记录程序，攻击者利用该程序捕获所有系统合法用户的按键操作。采用这种方法，攻击者可以发现密码，甚至可能提升自己的权限等级。有时候，按键记录器(keylogger)仅是一个大型攻击工具集合的一个组成部分，这类大型攻击工具集合称为 rootkit。4.4.3 节将详细地讨论这类攻击工具。

攻击者可以利用的另外一些方法是如果发现远程代码执行漏洞，在目标系统上安装一个程序，利用该程序发起针对另一站点的拒绝服务攻击(denial-of-service，DoS)[6]，例如，可能仅是简单的发起大量看起来合法的请求，耗尽站点的资源。这类程序称为 bot 程序，它是 robot 的缩写，是指实现恶意控制功能的程序代码，它们是由攻击者远程控制的，有时候可能需要通过一个 Internet 中继聊天(Internet Relay Chat，IRC)服务器。攻击者有时可能已经攻破了大量的机器，并在上面植入了 bot 程序。植入 bot 程序的计算机一般称为僵尸计算机，存在大量僵尸计算机的网络通常称为僵尸网络(botnet)。要达到控制目标网络的目的，攻击者需要做很多工作，需要编写运行在每个目标操作系统上的 bot 程序，或者在黑市上购买 bot 程序。僵尸网络(botnet)可以用来实施大规模分布式拒绝服务攻击(Distributed DoS，DDoS)，从而关闭 Web 站点。

攻击者为什么要攻击？我们在调查攻击时，发现：找到破坏系统安全的方法和产生利用该方法进行攻击的动机之间还有差距，即攻击者即使发现了某个隐患，也不一定会利用该隐患来发起攻击。这种差距使得管理人员认为企业不需要防御某些攻击，因为“从没有人利用该隐患实施攻击。”这种短视通常只是缺乏想象。例如，在僵尸网络中，攻击者可以利用 DDoS 的威胁来勒索依赖 Internet 的公司(这也引发了一些有趣的社会问题：拥有缺陷系统的团体需要采取措施来阻止攻击，但这类团体往往不是最后受伤害的团体)。

另一类攻击是运行在低权限级别的攻击者可以获取高级权限(如前所述)。在这种情形下，攻击者首先获取一个低级的账号，然后利用隐患来提升自己的权限级别。过去很多系统为来宾账号设置默认访问权限，这类账号通常没有密码，或者密码安全性弱，很容易遭受权限提升攻击。

远程代码执行漏洞的最后用途是传播蠕虫或者病毒。攻击者首先发现一个不需要用户交互(例如，不需要目标系统访问某个 URL)就可触发的远程代码执行漏洞，然后编写自传播的程序扫描和利用该漏洞。由于这个原因，该类漏洞有时称为蠕虫型漏洞(6.4 节将讨论蠕虫、病毒和其他恶意软件)。

4.4.2　通用攻击技术

本书全程都会讨论系统攻击技术。本节将介绍一些这方面的主题，并给出一些针对操作系统的攻击的细节内容。

---

[6] 人们很可能将 DoS 攻击与古老的 DoS 操作系统关联起来，但我们不是这个意思。

一个非常古老的骗术就是欺骗其他人泄露证书，这类策略通常称为社会工程学。典型的社会工程学攻击是攻击者像合法系统用户那样呼叫帮助台，然后编造谎言说他丢失了密码，希望帮助台能够提供一个新的密码。为了提高攻击的有效性，攻击者经常会根据人的自然心理来增强欺骗性，比如漂亮年轻的女人可能很容易欺骗男性技术工人；扮演一个愤怒没有耐性的高级经理的攻击者比较容易骗过一个地位较低的 IT 人员。

拥有更多资源的攻击者能够依靠垃圾搜寻(dumpster dive)来从大量的垃圾信息中搜集有用的信息。尽管很多机构对相关人员进行了大量的培训以阻止该类型的攻击，但很多攻击者仍然将人视作操作系统安全中最为薄弱的环节，第 18 章将深入讨论这一点。必须承认的是，这种方法非常普通，没什么新意，但仍然有效。有兴趣了解更多社会工程方面知识的读者可以查阅 Kevin Mitnick 著的 *The Art of Deception* 一书[MS02]。

攻击者经常使用的另外一种技术是反向工程(reverse engineering)，通过对软件实施反向工程来发现其中的隐患。软件可能是操作系统，或者是操作系统提供的服务，如 DCOM(事实上，由于害怕攻击者在获取源码后能够轻易地发现隐患，提供商经常保留源码的所有权，这一点至少可以避免潜在攻击者轻易地克服反向工程的障碍，后面将详细讨论)。一旦发现某个隐患，攻击者就可以编写一个程序，利用该隐患，获取目标主机的远程特权级访问权限，在目标主机上运行任意命令，或者直接在目标主机上运行攻击者的代码(第 6 章将讨论该过程)。

最后一种攻击者经常使用的技术利用与操作系统紧耦合的应用程序来获取操作系统访问权限。IE 浏览器是这类攻击的典型代表。由于浏览器的很多工作需要内核级权限，并且通过 Web 页面从 Web 服务器获取信息，攻击者可以利用 Web 页面巧妙地欺骗用户以让他们在自己的机器上安装攻击者的代码。第 18 章将探讨此类型的攻击和类似的攻击，如网络钓鱼攻击。

4.4.3 按键记录器和 Rootkit

前面已简单地讨论过该问题，一种很常用的攻击方法就是在目标主机上安装软件。按键记录器软件就是该类型的恶意软件，通常称为间谍软件。间谍软件可被攻击者用来远程监控目标机。

能够发现该类攻击软件存在的系统用户或者管理员可以采取一些正确的措施来避免该类软件的破坏行为，如移除软件，或者至少需要隔离受感染的机器。因此，攻击者可能要付出大量的努力来避免其软件被检测和移除。Rootkit 就是该类攻击软件，根用户可能也无法检测到它的存在。正如 4.1.3 节所讨论的那样，操作系统的功能之一就是跟踪进程，并通过系统调用或者 shell 命令向用户报告跟踪记录。因此，Rootkit 需要付出很大努力才能够使自己不显示在标准报告工具和文件系统中。例如，UNIX 系统使用 ps 命令来显示当前正在运行的进程。因此，要想隐藏自己的进程需要重写 ps 代码，以使 ps 不报告自己的存在。另外一些 Rootkit 直接修改内核的数据结构来达到同样的效果，例如，很多内核级 Windows Rootkit 将自己从进程表中移除。

Rootkit 是一套攻击工具集合，攻击者在成功获取目标机的根权限后，在目标机上安装 Rootkit 以达到隐藏自身踪迹的目的。由于整本书都会讨论 Rootkit，因此，本节我们仅简单地介绍一些基础知识，有兴趣的读者可以在本书参考文献的[HB05，Hog6]中获取更

为详细的信息。

Windows Rootkit　Windows 有两种类型的 Rootkit：一类运行在用户空间，一类运行在内核空间。运行在用户空间的 Rootkit 虽然危险，但是很容易检测。这类 Rootkit 通常将自己注入到某些报告系统状态的程序中(如用来浏览文件系统的 Windows Explorer)。Rootkit 利用动态链接库(Dynamic Link Library，DLL)注入技术，可以强制 Windows 装载程序将 Rootkit 代码载入到目标进程中，该处目标进程就是指 Windows Explorer。另外一种方法是利用 Win32 API 调用函数 CreateRemoteThread 在目标进程中创建执行线程(代码由攻击者选择)。然后，攻击者就可以查看和修改目标进程的地址空间。这种方法虽然可以用来攻击，但也是一种合法的使用方法。因此，在 Windows 系统编程文献[Pie00，Ric99]中有详细的描述。

一旦安装到目标进程，Rootkit 代码所要做的第一件事就是通过修改目标进程的导入地址表(Import Address Table，IAT)来重定向它所感兴趣的 Win32 API 调用。IAT 是一个指针表，装载程序使用该指针来查找 API 调用的地址。重定向目标的 IAT，使其指向 Rootkit 规定的地址，Rootkit 就可以截取它需要的任何 API 调用。用户级 Rootkit 通过这种方法来达到隐蔽自己。例如，当需要显示某个目录的内容时，Windows Explorer 会调用某个 API，如果已经安装了某个 Rootkit(如 Vanquish[xsh06])，那么就可以截取该调用，避免其报告 Rootkit 自身信息。它的作用就是避免使用 Windows Explorer 浏览文件系统的用户看到 Vanquish Rootkit。该技术称为 API 钩子(API hooking)，使用该技术的 Rootkit 有时称为钩子(hooker)。

以驱动程序方式安装的 Windows Rootkit 更加难以检测，这种方式的 Rootkit 能够访问一些内核数据结构。与用户空间 Rootkit 重定向目标进程的 IAT 不同，内核 Rootkit 将自己钩在系统服务描述符表(System Service Descriptor Table，SSDT)上，该表也称为系统调用表。Rootkit 利用该技术截取任何它感兴趣的机器进程的系统调用。另外，内核级 Rootkit 可以使用如直接内核对象操作(Direct Kernel Object Manipulation，DKOM)技术修改内核的内部状态。Rootkit 可以使用该方法修改数据结构，如进程表，来有效地隐藏自己。

检测 Rootkit 比想象中的更为艰苦。很多反病毒软件能够检测一些非常普通的 Rootkit，但是难以检测更为先进的 Rootkit。当前已经出现了很多检测 Rootkit 的专用工具，2004 年，Microsoft 研究院发布了一种称为 Ghostbuster 的检测工具，该工具能够检测一些 Rootkit[WBV+05]。其主要工作原理是，首先在 API 级扫描感染的系统，然后再在原始磁盘级扫描系统，最后进行对比，如果 API 级扫描没有报告磁盘级扫描的数据，那么系统中就可能存在 Rootkit。2005 年，Bryce Cogswell 和 Mark Russinovich 发布了一个称为 Rootkit Revealer 的工具，它利用上述类似的技术来检测 Rootkit。

Linux Rootkit　Linux 并不是不受 Rootkit 的影响，只是在方法上可能有所不同。旧版本 Linux(以及 UNIX)Rootkit 的工作原理是替换负责向系统管理员或者根用户报告系统状态的系统代码：旧版本 Rootkit 的通常做法是先关闭系统日志监控进程(负责记录系统变化的组件)，然后，将系统中的一些关键代码替换为自己的代码，这类关键代码通常包括 netstat、ps、top 等[7]。Rootkit 通过替换这些程序来隐藏自己使用的网络连接和进程。但是，管理员

[7] 好奇的读者可以使用 man 来查看这些程序的详细描述。

通过使用或者编写未被 Rootkit 修改的程序可以很容易地检测到 Rootkit。

更为复杂的 Linux Rootkit 与 Windows 上的 Rootkit 工作原理很相似：钩在系统调用上，然后直接修改内核数据结构。由于 Linux 内核允许用户和攻击者通过模块扩展内核，因此，很多内核 Rootkit 以模块的形式实现。一旦安装，Rootkit 将重定向部分系统调用表指向自己的代码，以截取机器上其他进程的系统调用。

目前有很多检测 Windows Rootkit 的方法，虽然没有一种是完善的。一种方法是通过文件系统完整性检查工具(如 Tripwire)来监控磁盘的变化，这类工具检查文件的内容是否已经遭受非法修改(我们曾使用该技术发现了 Windows 和 Linux 系统上的 Rootkit)。尽管 Rootkit 在系统程序级的隐藏非常成功，但是要实现在磁盘上的隐藏要困难得多，虽然也不是不可能。另外一些方法则试图检测内核模块是否修改了系统调用表[KRV04]，该方法的基本思想认为只有 Rootkit 才会修改系统调用表。

4.5 选择何种操作系统

在我们的职业生涯中，总是会面临选择使用何种操作系统的问题。这些问题可能是在新的计算机上选择安装何种操作系统；在客户站点上推荐部署什么；在什么样的平台上开发新的应用。对很多人来说，选择使用什么操作系统仅仅是一个偏好或者心理上的问题，或者是由其他人决定。本节将讨论在面临这种选择时，要考虑的与安全相关的因素。

4.5.1 Windows 和 Linux

在对比 Windows 和 Linux 时，一个非常古老但仍然在争论的问题就是：谁更安全？几乎每一年，都有团体会号召一些“公正的”第三方调查和判定哪个操作系统更安全，这些第三方通常只比较两者发布在 BugTraq 邮件列表[Sec06]上的漏洞数量、蠕虫和病毒的数量、安全补丁的数量等等。有趣的是，发起调查的一方几乎总是胜出的一方，失败的一方也总是质疑谁做的调查，并宣称该团体并没有做到公正无私，还经常在他们的调查方法上挑毛病。

真实的情况是两种操作系统都有安全问题，并且，如果一直有人使用它们，那么他们将很可能会不断的发现安全问题。正如第 18 章所述，安全性和易用性之间需要折衷考虑。如果这些操作系统被一定数量的人购买使用，那么它们就必须保持一定的易用性。但是，如果它们易于使用，那么在安全性上就相对差点。在安全作为最高优先级考虑的环境中，这些操作系统或其他操作系统在特定安全上的配置可能更有必要(稍后将讨论其他操作系统)。

假设两种操作系统都有共同的安全问题，那么一个很明显的问题是：为什么不使用 Mac 呢？有人可能由于 Mac 没有发布像 Windows 和 Linux 那么多的负面安全问题，就认为 Mac 必然不存在同样的安全问题。也有人认为 Mac 的市场份额不够，不足以引起攻击者的注意，发现一个 Windows 系统隐患所获取的回报要高于发现一个 Mac 系统隐患的回报，因为攻击者可以感染更多的系统。谁是正确的？我们不知道！

但我们知道的是，已经有人开始调查 Mac 的安全问题，随着 Mac OSX[Win06]推广范围的不断扩大，最近已经出现了一些针对该系统的蠕虫。有人认为 Mac 系统存在大量的隐患，因为安全研究人员并没有像关注 Windows 系统那样关注它[Kot06]。不论如何，操作系统的使用范围越广，它受到安全团体的关注就越多。

Mac OSX 也有超过 Windows 的地方，它对在内核模式中运行的程序有更多的规定，它基于有着悠久历史的 BSD 和 Mach 微内核技术。Mac OSX 内核的核心是开源的。更多有关 OSX 工程思想方面的信息可以在 Amit Singh 的书中查阅[Sin06b]。

4.5.2　其他操作系统

1. OpenBSD

OpenBSD 操作系统是一个考虑了安全性的具有 BSD 风格的 UNIX 系统。OpenBSD 官方网站自豪地宣布：“在 10 多年里，默认安装仅发现了两个远程漏洞！”[Ope06]。OpenBSD 是第一批整合了密码学支持的 UNIX 系列系统之一，从为开发者提供支持的密码学服务和一些如密码存储机制的内部系统中可以看出这点，OpenBSD 使用 Blowfish 算法加密码。

OpenBSD 试图做到在默认配置下是安全的，管理员不需要执行冗长的任务列表检查就可以将机器连接到网络。该项目仍然关注安全方面的问题，并且已经开发了一些可自由使用的安全工具。

2. SELinux

几年前，美国国家安全局发布了一个考虑很多安全因素的 Linux。安全增强 Linux(Security Enhanced Linux，SELinux)[NSAa]采用基于角色的强制访问控制，保护用户免受其他用户(包括根用户)的影响。SELinux 的核心是一套大规模且复杂的策略声明，用来为主体分配角色，为客体分配类型，并解释了特定角色的主体如何与特定类型的客体交互。

我们可以根据该策略声明来创建软件隔间(software compartment)，使得应用程序在软件隔间中运行，不能访问其他进程的内存或者数据。根据该策略声明还可以限制隔间之间的交互。SELinux 可以从美国国家安全局的项目网站下载，它已经是某些版本 RedHat Linux 的一个组成部分。

3. OpenSolaris

Solaris 是来自 Sun Microsystems 的 UNIX 系列操作系统之一，它开发了一个评估机制，利用一些非常有趣的检查工具来评估可靠性(尤其是为高端服务器)和可观察性。最新版本的 Solaris 还开发了进程权限管理和容器[PT04，TC04]。Sun 的销售宣传鼓吹说“Solaris 10 操作系统，是由 Sun 公司投资超过 5 亿开发出的系统，是地球上最为先进的操作系统”，参见[Suna]。

尤为有趣的是，Sun 最近发布了一个 Solaris 开源版本[Sunb]，尽管 Sun 工程师有最终解释权，但是用户可以阅读代码，修改并重新编译代码，并提供反馈。这种做法使各种研究团体有机会观察一个长期演化且高度工程化的系统的内部结构。

4. 开源问题

我们经常需要讨论软件开源和其安全性问题，尤其是操作系统开源所带来的安全问题。我们的回答可能使任何一方的支持者都失望。

首先，需要弄清楚什么是“开源”，我们采用以下方法分类：

(1) 公开源码的软件，这类软件可以检查。

(2) 满足(1)的条件，此外，用户/消费者也能够修改和重新编译该软件，并且在其公司部署修改后的软件是合法的。

(3) 满足(1)和(2)的条件，此外，源码本身是由许多自愿者组成的团体编写的。

有人认为，发布源码会降低软件的安全性，因为攻击者能够很容易地发现隐患。我们强烈反对这种观点。从历史上看，“通过隐藏内部结构来保证安全性”的效果不是很好。并且，来自 Diebold 投票机的最新事件表明，“保密源码”只为不良软件工程提供了借口。端用户重新编译和重新部署软件的能力能够使他们在需要的时候补上隐患，而不必等待开发商的相关补丁，但这对软件开发企业的软件工程和测试技巧提出了更高的要求。

来自大型机构的软件则更为复杂[Ray99]。曾经有一个经理反对在产品中使用自由软件，因为她担心没有人会为修补那些软件的 bug 而负责。国家安全人员担心大型匿名机构中的恶意编程人员可能在软件中插入恶意代码。例如，几年前，一个开源的操作系统中曾发现以下语句：

```
if ((options == (_WCLONE|_WALL))&&(current->uid = 0))
    retval = -EINVAL;
```

一个字符上的疏忽

```
current->uid ==0
```

对此，一个看似无害的错误检查能够通过将调用者的权限设为 root(0)，而形成提升权限攻击。

就我们的亲身调查来看，小型高效的小组所开发的软件要好于大型效率较低的团队。

4.6 本章小结

操作系统在现代计算系统中有着十分重要的地位，它为计算和信息提供保护。由于这一原因，它也很容易遭受攻击。

许多桌面操作系统并不能很好的处理攻击，操作系统对开发者和用户的易用性越高，保证其安全性就越难。由于不断增加新的功能，操作系统的复杂度也不断提升。希望您已了解：在开发系统时保证所有细节都是正确的非常困难。大量的部件相互交互，现代操作系统的复杂度已经到了一个非常高的程度。而复杂性通常是安全性的敌人。

4.7　思考和实践

(1) 阅读某个真实操作系统的源码。Linux 操作系统的源码可以从 www.kernel.org 上自由下载，或者可以查看 Linux 的简化版：Minix，能够从 www.minix3.org 上下载。为了能够对复杂性有一个正确的认识，尝试完成一些需要你修改代码的任务，如实现一个系统调用。

(2) 比较和对比将某个应用，如 Windows Explorer，放到用户空间所带来的安全上的好处。它是可信计算基础(TCB)的组成部分吗？它的某些部分需要放在 TCB 中吗？哪些应用或者服务应该运行在内核空间？能否提出一个需求列表，以满足将某个应用放到内核空间的需要？

(3) 调查你的朋友和同事。他们使用的操作系统是什么？如果是 Windows，他们日常使用时，用的是管理员身份还是普通用户身份？