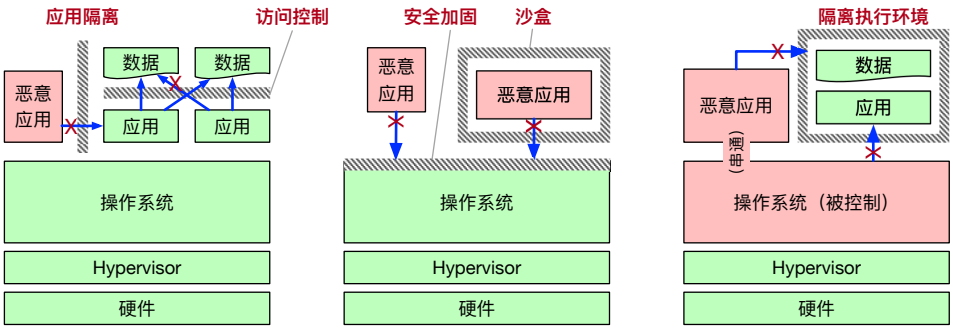
第十六章　操作系统安全

安全，是操作系统所提供的一个必不可少的功能和服务。为什么需要在操作系统层面关注安全？我们从三个角度来看这个问题：首先，从数据的角度来看，计算机系统中有隐私或机密的数据，比如账号密码、信用卡号、地理位置、银行账单、视频照片、电子邮件、设计图纸，等等；这些数据能够被一些应用程序访问，但不能允许任何一个应用程序都能随便访问，因此必须要有一种机制来保证哪些应用有权限访问哪些数据。其次，从应用的角度来看，操作系统之上运行了很多应用程序，用户有可能有意或无意地安装了恶意软件，这些恶意应用可能利用操作系统所提供的一些正常功能，来窃取用户的数据，比如周期性发送用户的地理位置；也可能利用操作系统的安全漏洞，获取更高的权限。操作系统需要在保护自己的同时，检测、识别、限制恶意应用的各种攻击。再次，从操作系统自身的角度来看，由于操作系统通常非常复杂，动辄上千万行代码，因此不可避免的存在各种 Bug。在系统运行的过程中，由于软件的缺陷或硬件的故障（比如内存中某个 bit 发生了反转），存在被攻击者完全控制的可能。需要有一种方法，在操作系统已经沦陷的情况下，依然能够为用户数据提供一定程度的保护。因此，我们在本章中所讨论的操作系统安全，主要包含三个层次，如图 16.1所示：

首先，操作系统需要保证数据等资源只能被有权限的应用访问。相关的机制具体包括两方面：一方面，对于文件数据来说，由于文件系统是全局的[1]，在没有保护的前提下，所有应用都能看到所有文件。因此，需要通过访问控制机制来限制应用对文件的访问。另一方面，对于在内存中的数据来说，由于应用的虚拟内存空间本身是私有的，因此已经能够提供较强的应用间隔离，本章不再详细介绍。注意，由于整个访问控制机制都是有操作系统来提供的，因此这时我们假设操作系统是安全可靠的。

其次，操作系统需要面对病毒、蠕虫、木马等恶意软件，这些恶意软件往往会利用操作系统的隔离机制与访问控制机制的实现漏洞，绕过隔离攻击其他

[1]使用沙盒机制可以将文件系统设置为非全局，详见后文。



(a) 基于操作系统的应用隔离与访问控制

(b) 操作系统对恶意应用的隔离与防御

(c) 操作系统不可信时对应用的保护

图 16.1: 操作系统安全的三个层次

应用甚至攻击操作系统。因此，我们会介绍操作系统中常见的安全漏洞，以及操作系统针对这些漏洞的安全加固方法。在这些方法中，第 十五章提到的**沙盒**（Sandbox）机制是一种常见的用来运行不可信应用的方法，操作系统构造一个受限的运行环境，仅提供基本的功能与资源，以最大化提升恶意软件的攻击难度。

最后，我们假设操作系统已经被攻击者完全控制，并有可能与恶意应用串通，联合起来对其他应用发起攻击。在这种情况下，由操作系统所提供的访问控制机制与沙盒等安全加固方法均已失效，因为攻击者可在具有更高权限的操作系统中对应用发起“降维攻击”，或者干脆关闭这些安全机制。此时，我们能依赖的只有位于更高“维度”，也就是具有更高权限且不依赖操作系统的软硬件组件，包括 hypervisor 和硬件的安全扩展等。其中，基于 hypervisor 的安全隔离是利用虚拟化技术运行多个虚拟机，每个虚拟机均运行自己的客户操作系统，多个虚拟机之间彼此隔离；本章将介绍基于硬件扩展的**可信执行环境**（又称“Enclave”，飞地）如何构造一个不受外界特权软件攻击的应用执行环境。

这三个层次也可以被认为是防御的三个阶段，体现出“可信计算基”、“攻击面”与“防御纵深”这三个概念。**可信计算基**（Trusted Computing Base，TCB），是指“为实现计算机系统安全保护的所有安全保护机制的集合”[2]。这里的机制包括了软件、硬件和固件等所有与计算机直接相关的组成部分。TCB 是一个人为定义的概念，首先确定“安全保护”的目标，然后确定系统中对该目标有影响的组件和机制的集合，即为 TCB；换句话说，TCB 之外的组件即使受到攻击或被攻击者完全控制，也不会对安全目标产生影响。因此，TCB 与

[2]https://en.wikipedia.org/wiki/Trusted_computing_base

两个因素相关：一是安全目标，二是系统架构。同一个系统，不同的安全目标可能对应不同的 TCB；对同一个安全目标，不同系统架构的 TCB 也可以存在区别。TCB 越大，需要信任的组件越多，组件中潜在的 bug 数量就越多，那么可能存在的安全风险就越大；因此，在系统设计的时候，应当尽可能减小 TCB 的大小。

某个组件的攻击面，是指该组件被其他组件攻击的所有方法的集合，包括来自底层更高权限软硬件的攻击、来自其他组件通过该组件接口发起的攻击、以及所有通过控制该组件所依赖的数据而改变组件行为的攻击等。对于操作系统来说，攻击面包括恶意 hypervisor 和硬件，所有的系统调用接口，以及相关的配置文件等；对于应用来说，攻击面包括恶意操作系统、hypervisor 和硬件，对外提供的 IPC，以及所有来自网络和磁盘的数据。

为了提高系统安全性，一种方法是减小系统的 TCB，另一种方法是减小攻击面。然而，操作系统的功能越多，提供的接口越丰富，TCB 和攻击面通常就越大；换句话说，为了降低 TCB 和攻击面，往往不得不减少操作系统的功能，限制对外提供的接口，这会影响到整个系统的可用性、灵活性、兼容性，也常常会导致系统性能的下降。这里的问题在于系统的复杂度：安全性要求系统复杂度越低越好，而功能的增加则不可避免的会导致复杂性增加，这是一个根本性的矛盾。

如何解决这个矛盾？一种方法是，为系统设置多道防线，当一道防线被攻破后，后面的防线依然可以防御，这就是“防御纵深”的概念。我们从时间的维度，将图 16.1视为三个阶段：第一阶段，攻击者尝试利用应用和系统所提供的 API 和接口进行攻击，此时的 TCB 包含了应用和操作系统；第二阶段，攻击者利用了应用的漏洞控制了整个应用，从而可以直接攻击操作系统，攻击面为系统调用，TCB 则为操作系统；第三阶段，攻击者进一步利用漏洞控制了整个操作系统，但我们可以利用可信执行环境，将操作系统从 TCB 中去除，而仅为可信执行环境提供非常有限的功能，且每个功能都可以被隔离环境中的应用程序所验证 [3]。此外，从应用的角度来看，也可以综合利用多种技术，例如将关键的逻辑和数据分拆后放到多个可信执行环境中，以提高应用自身的防御纵深。

[3]例如，应用程序可以通过计算哈希来验证 read() 返回的文件数据是否正确，但并不是所有的系统调用都可以被直观的验证。

16.1 安全目标与威胁模型

本节主要知识点

- ❑ 为什么说“没有绝对的安全”？
- ❑ 安全的目标包括哪几点？
- ❑ 系统设计时，为什么需要设定威胁模型？

对于系统来说，安全的目标与其他目标有一个很大的区别，那就是绝对的安全是无法达到的。对于性能，目标可以是“启动时间小于 1 毫秒”，或者“吞吐率高于每秒处理 100 万请求”，这些目标是可量化、可判读、可对比的；而安全并没有直接的量化方法，这是因为安全是一个**负面目标**（Negative Goal）[7]，并不是“能做到什么”，而是“不能做到什么”。例如，“除了小明之外的任何人都不能访问小明的工资数据”，对这个目标来说，证伪要比证明简单的多：证明需要穷举所有可能的获取小明工资数据的方法，并论证这些方法都无法成功；而证伪只需要例举一个反例就够了。所以，我们很容易比较两个系统哪个更快，但如果想比较两个系统哪个更安全，或某个系统是否比之前更安全，往往很难。

现实中，通常会使用两个近似的指标来衡量安全：代码的数量，以及安全漏洞的数量。第一种方法的假设是：代码的数量一般与 bug 数量成正比，bug 数量与安全漏洞数量成正比。因此，通常认为代码数量越少，系统的安全漏洞越少——这既符合人们的直觉，又符合统计规律。一个常用的指标为每 1,000 行代码的平均缺陷数量，称为**缺陷密度**（Defect Density），例如，Linux 的缺陷密度近年来已经小于 0.5 [4]。然而，直接比较两个模块的缺陷密度也并不一定合适，例如 Linux 内核中，GPU 驱动的缺陷密度仅为 0.19，而 SMACK（Linux 的一个安全模块）高达 1.11；但从绝对数量上看，GPU 驱动的有多达 160 个 bug，而 SMACK 则只有 6 个 bug。哪个模块更安全呢？所以，仅仅通过缺陷密度来比较两个模块的安全程度是不够的，还需要综合考虑代码规模、模块功能、缺陷重要性等多个因素。

第二种方法是以安全漏洞的数量来衡量安全性。CVE（Common Vulnerabilities and Exposures）是一种常见的漏洞编号方式，格式类似 CVE-2020-10757，目前所有的 CVE 由 Mitre 公司负责记录 [5]。中国也有对应的编号方式，例如国家计算机网络应急处理协调中心的 CNCVE 编号，国家信息安全漏洞

[4] https://scan.coverity.com/projects/linux

[5] https://cve.mitre.org/

共享平台的 CNVD 编号，以及中国信息安全测评中心的 CNNVD 编号。对于同一个系统的不同版本来说，CVE 的数量具有一定的比较意义，例如系统在应用某个安全模块之后，能够防御更多的 CVE（同时仅引入较少或没有引入新的 CVE），则可以表明系统变得更安全。然而，不同系统之间很难通过 CVE 的数量直接对比其安全程度。一个系统被发现的 CVE 数量越多，有可能意味着其没有被发现的 CVE 更多；也有可能是因为其本身很开放（例如开源），测试的非常全面，因此并不意味着比那些 CVE 数量少的系统更不安全。例如，Linux 内核的 CVE 数量目前排在第 3 位（2,357 个），而微软的 Windows XP 则排第 28 位（741 个），但大部分人并不会据此认为 Windows XP 就比 Linux 内核更安全。

尽管绝对的安全是无法达到的，我们依然可以为安全设置合适的目标，前提是明确威胁模型。换句话说，在一定的假设和前提下，实现与安全相关的系统属性。具体来说，系统安全有三个属性（目标）：机密性、完整性、可用性，合起来简称 “CIA”。由于数据是系统安全保护的主要对象，本章将以数据保护为例来介绍操作系统安全。从数据的角度来看，上述三个属性分别表示：

- 机密性（Confidentiality）：又称隐私性（Privacy），是指数据不能被未授权的主体窃取（即恶意读操作）；
- 完整性（Integrity）：是指数据不能被未授权的主体篡改（即恶意写操作）；
- 可用性（Availability）：是指数据能够被授权主体正常访问。

系统中的数据不仅包含私钥字符串、word 文档、音视频文件等静态数据，也包含代码，以及运行过程中的各种动态数据，比如堆栈上的变量、CPU 寄存器中的运算结果、I/O 设备内部的配置等。并不是所有数据都需要同时保证 CIA，不同数据的安全目标可以不同。例如，对于需要公开的数据，如网站上的一篇新闻，只需要保证完整性及可用性而不需要保护机密性；对于保存在手机上的邮箱登录密码，机密性相对完整性与可用性来说更加重要——毕竟，如果本地保存的密码被篡改或被删除了，登录时会报错，重新输入一遍就行了（若攻击者的目的就是为了让用户重新输入一遍密码，从而利用键盘记录器等工具窃取密码，这种情况另当别论了）。

威胁模型是指与安全相关的前提和假设，包括系统中哪些组件是可信的（即 TCB），哪些组件不需要是可信的，有哪几类攻击者，攻击者具有哪些能力等。威胁模型的例子包括：“在一个多计算节点的系统中，攻击者能够控制 K 个（而不是所有的）计算节点，以及可以看到整个网络的数据”，“攻击者能够在系统中安装并运行任意的应用，但无法直接控制操作系统内核”。在威胁模

型中，一般也会明确列出哪些攻击是在考虑范围之外的，例如“不考虑基于内存嗅探的硬件攻击”等。明确威胁模型的目的，是为了将安全问题进行简化，从而将需要考虑的攻击类型和数量控制在一个合理的范围。

16.2 访问控制

本节主要知识点

- ❑ 访问控制的威胁模型是什么？
- ❑ 有哪几种认证机制？各自有什么优缺点？
- ❑ 授权的原则有哪些？
- ❑ *DAC* 和 *MAC* 的区别是什么？

访问控制（Access Control）是按照访问主体的身份来限制其访问对象的一种方法。为了实现对不同应用访问不同数据的权限控制，操作系统提出了两个基本的机制：认证和授权。

- 认证（Authentication）：验证某个发起访问请求的主体的身份；
- 授权（Authorization）：授予某个身份一定的权限以访问特定的对象。

引用监视器（Reference Monitor）是实现访问控制的一种方式，具体架构如图 16.2所示。该机制的思路是：将主体与对象隔开，即不允许主体直接访问对象，而是必须通过引用（reference）的方式间接访问。引用监视器位于主体和对象之间，负责将来自主体的访问请求应用到对象。通过增加这层抽象，系统能够保证所有请求必须都经过引用监视器，因此可以在这一层进行访问控制。具体来说，引用监视器首先认证主体的身份，然后根据预设的策略，判断主体是否有权限访问对象，并选择允许或拒绝，同时将所做的选择记录到审计日志中。

引用监视器机制必须保证其**不可被绕过**（Non-bypassable）。设计者必须充分考虑应用访问对象的所有可能路径，并保证所有路径都必须通过引用才能进行。例如，应用必须通过文件描述符来访问文件，而无法直接访问磁盘上的数据或通过 inode 号来访问文件数据。文件系统此时就是引用监视器，文件描述符就是引用。

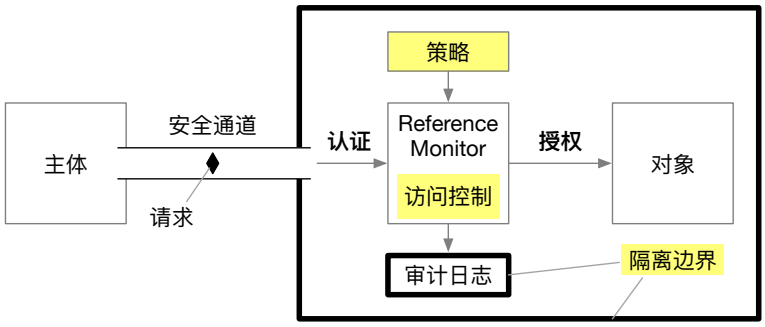


图 16.2: 基于引用监视器的访问控制机制 [1]

16.2.1 认证机制

认证的过程，就是建立发起请求的主体与系统中某个 ID 之间的绑定关系。例如，用户登录操作系统，首先要选择用户名（即 ID），然后输入密码/口令完成登录过程。

认证过程中，判断某一个主体身份的方法主要有三种：

- 你知道什么（Something you know）：例如密码/口令、手势密码、某个问题的答案等；
- 你有什么（Something you have）：例如 USB-key、密码器等实物；
- 你是什么（Something you are）：例如指纹、虹膜、步态、键盘输入习惯等，属于人的一部分。

Linux 一般通过用户名和密码[6]登录，所有用户的密码被加密后保存在*/etc/shadow* 文件中。远程登录时，也可以选择使用公钥登录方式，使用私钥来验证身份。以 ssh 登录为例，用户提前将自己的公钥保存在 ssh 服务器上，当登录时服务器会向客户端发送一段随机字符串，客户端使用私钥加密字符串后发回，服务端用公钥解密验证后则登录成功。在这个例子里，私钥一般属于“有什么”一类（如果有人能够把上百位的私钥背下来，那就属于“知道什么”一类了）。

使用密码登录的问题在于：密码较容易泄露。泄露可能发生在输入密码的时候被其他人看见，也可能是服务器端没有按规范对密码加密，而保存了密码明文的数据库被黑客偷走。使用**双因子认证**（Two-factor Authentication）或**多因子认证**（Multi-factor Authentication），则可以比较有效的解决这个问题。登

[6]此处密码指口令。

录的过程除了输入密码外，还需要额外提供一个或多个证明，例如手机验证码，或者需要插入一个 USB-key 来验证，即要求用户证明自己“有什么”。

使用密码器或手机验证码等实物进行认证的问题在于，容易忘带或丢失。现在越来越多的系统，尤其是手机、平板、笔记本、汽车等设备，开始采用生物特征来进行用户的认证。指纹识别和人脸识别是现在最常用的生物识别技术。除此之外，基于人的输入习惯来进行识别，包括物理键盘的输入模式，如不同键位的输入频率、甚至是习惯的失误；手机的输入模式，如手机倾斜的角度、输入时产生的震动等，这些可通过手机内的传感器获得。基于用户生物特征和行为可以实现连续的用户认证，避免了频繁提示用户输入密码，可以实现更好的用户体验。但由于生物特征是有限的，一旦泄露，更换本身可能会遇到问题（例如十个手指的指纹数据都遭到泄露）。因此，对于生物特征数据的安全保护非常关键，目前工业界一般使用基于硬件的可信执行环境来进行生物数据的采集和比较。

16.2.2 授权机制：以文件系统为例

授权，是判断某个主体是否有权限访问某个对象的过程。授权机制主要考虑三个问题：1）用何种数据结构来表达主体与对象之间的权限关系；2）如何设置和修改这种权限关系；3）如何强制保证这种权限关系。本节将以文件系统为例来介绍授权机制。

文件是数据的抽象方式之一。操作系统强制所有应用都必须通过文件系统访问保存在底层存储设备上的数据，因此在文件系统实现授权机制不会被恶意应用绕过。对于文件系统来说，这里的主体就是进程，每个进程都属于一个用户；对象就是文件，文件主要有三种权限，分别是可读、可写、可执行（简写为r、w和x）。为表示进程与文件的权限关系，最直观的数据结构是权限矩阵，类似表 16.3所示。

	对象-1	**对象-1**	**对象-3**
实体-1	读/写	读/执行	读
实体-2		读/执行	读/写
实体-3	读		读/写

图 16.3: 权限矩阵示例

然而，从实现角度来说，这个表格会非常大：假如系统中有 100 个用户，每种权限用 1 个 bit 来表示，那么每个文件都至少需要 300 个 bit 来表示 100 个用户的 3 种权限。假设这些 bit 都保存在 inode 中，通常 inode 的大小为 128-Byte 或 256-Byte，300 个 bit 相当于一个 inode 的 15% 至 30% 的存储空间大小，这是一个相当大的比例。更重要的是，每当新建一个用户的时候，都必须要更新所有 inode 中的权限 bit，这在性能上也是不能接受的。

因此，为了简化权限矩阵，在 Linux 中，一个文件并不会对每个用户都记录相应的权限，而是利用了用户组的概念，将用户分为三类：文件拥有者、文件拥有组、其他用户。用户组是用户的集合，拥有独立的组 ID（group-id），每个用户都属于一个或多个用户组。每个文件都在 inode 中记录了其拥有者的 user-id 以及拥有组的 group-id，若某个用户不属于这两者，则为其他用户。于是，每个文件只需要用 9 个 bit 即可（3 个权限 x 3 类用户），且用户的新建与删除过程无需更新这些权限位。每个文件的访问权限位形成了一个列表，称为**访问控制列表**（Access Control List，ACL）。文件的拥有者可以设置文件的权限，通过`chmod`命令操作，该命令则会调用`chmod()`系统调用。

常见的文件权限：目录权限通常设置为`755`。其中`7`表示`rwx`，`5`表示`rx`。这里，`x`权限用于进入目录，`r`权限用于读取目录；换句话说，若去掉某个目录`dir`的`x`权限，则`cd dir`会报错；若去掉`r`保留`x`，则可以进入这个目录，但在目录中运行`ls`会出错；没有`w`权限，表示不能在目录中删除或新建文件。注意，删掉一个文件并不需要该文件的`w`权限，而只需要文件所在目录的`w`权限。一个文档文件的权限通常设置为`422`，即没有`x`权限。符号链接文件的权限为`777`，因为真正起作用的是链接所指向文件的权限。

权限检查

系统什么时候去检查用户是否有权限访问某个文件呢？从设计上来说，可以有多种选择。一种方式是在每次操作文件的时候检查，包括打开、读、写文件等；另一种是仅仅在打开文件的时候检查。Linux 的选择是：仅对 open() 进行权限检查，但要求必须先打开文件才能对文件进行其他操作，从而兼顾了安全与性能。为保证文件其他操作必须在 open() 操作成功之后才能进行，操作系统引入了文件描述符（file descriptor，简写为 fd）的概念，要求对文件的读写等操作必须用 fd 作为参数，而为了得到一个文件的 fd，则必须先进行 open() 操作，并传入期望获得的权限（如读写、只读、只写）作为参数。因此，一个文件可以对应多个 fd，即被打开多次，每次打开的权限可以不同；后续对

某个 fd 的读写操作，仅会检查与该 fd 对应的权限，而不会再检查文件的权限。

小思考

文件的权限被修改，对已被打开的文件会立即生效么？

考虑如下情况：在进程 *A* 打开某个文件时，该文件具有可写权限，因此进程 *A* 以可读可写权限打开了文件；然后，文件的权限被拥有者修改为只读，那么之后当进程 *A* 对文件进行写操作时，会成功还是失败呢？根据前一段的描述，进程 *A* 会一直拥有对文件的写权限，直到关闭该文件。若系统希望对文件的权限更新立即生效，则需要在更新权限的同时，遍历所有打开文件的 *fd* 并做相应的处理，例如直接关闭所有权限不匹配的 *fd*，这样进程 *A* 下次进行文件操作时就会出现错误。

最小特权级：SUID 机制

在检查权限时，主要是以当前进程的用户 ID 作为参数进行检查。在一个简化的模型下，每一个进程 ID 都会对应一个用户 ID，在用户 ID 和进程 ID 之间具有一个绑定关系。打开文件时，文件系统首先找到文件 inode，并根据 inode 中记录的权限 bit，判断当前进程的用户 ID 是否有对应的权限。

然而，这样的设计在遇到以下场景时会变得很麻烦：某个可执行文件由于包含了对一些特权资源的访问，需要特权用户才能访问。例如，某个用户希望修改登录的密码，需要用 passwd 这个命令，但这个命令包含了对保存密码文件的操作，该文件需要特权用户才能更改。于是，这个用户必须先让自己成为特权用户（root），执行完命令后再退出特权用户。这个过程违背了最小特权原则：当普通用户提权为 root 时，除了修改密码还可以做许多其他的特权操作，因此存在很大的安全隐患。

为了解决这个问题，保证该用户在提权为 root 后只能运行 passwd 这个文件，文件系统提供了一种新的方法：在文件的 inode 中增加了一个 SUID bit，当这个 bit 被置为 1 时，打开并执行这个文件的进程会被临时提升为特权用户；当执行 passwd 命令结束后，恢复之前用户的身份。这个设计即允许了普通用户的进程能运行 passwd，又限制了进程只能运行 passwd，让进程拥有 root 权限的时间和能力缩到了最小。

SUID 的安全隐患：需要注意的一点是，由于进程在执行具有 SUID 位的文件时获得了短暂的提权，若文件中的某些代码存在安全漏洞，例如在提

权运行的阶段有一个任意代码执行的漏洞，那么攻击者就有可能利用漏洞，让这个进程在提权状态下打开一个 shell，从而获得了一个具有特权的 shell。因此，很多黑客都会遍历系统中 SUID 位被置上的文件，然后去找这些文件的漏洞；历史上也曾出现大量这类的攻击。

16.2.3 自主访问控制（DAC）和强制访问控制（MAC）

自主访问控制（Discretionary Access Control，DAC），是指一个对象的拥有者有权限决定该对象是否可以被其他人访问。例如，文件系统就是一类典型的 DAC，因为文件的拥有者可以设置文件如何被其他用户访问。换句话说，DAC 允许一个普通用户配置其所拥有的对象的访问权限。

强制访问控制（Mandatory Access Control，MAC）则与 DAC 相对，即一个对象的拥有者不能决定该对象的访问权限，什么数据能被谁访问，完全由底层的系统决定。例如，在军用的计算机系统中，如果某个文件设置为机密，那么就算是指挥官也不能把这个文件给没有权限的人看——这个规则是由军法（系统）规定的，任何人，包括通讯员（普通用户）和指挥官（特权用户）也不能违反。

可以简单地为 DAC 和 MAC 做一个对比：DAC 比较灵活，允许用户通过不同的配置自行决定数据的访问权限；但安全保证相对较弱，很有可能某个用户对文件的权限配置错误，导致一些关键文件数据泄露；相反，MAC 的权限等级与限制都是固定的，用户可配置的余地较少，好处是安全性有保证，例如一旦某个文件被设置为机密，且系统规定机密的文件不能发送给有网络权限的进程，那么无论其他配置如何，系统一定能保证该机密文件不会通过网络发送出去。

用“规章和法律”与“DAC 和 MAC”类比：这两种文本都可以用来约束人们的行为。不同的是，规章通常是某个机构的负责人制定，例如学校宿舍某栋楼可以规定晚上 11 点之后不得入内，某个人可以在自己家门上写“风能进，雨能进，国王不能进”以明确进入的权限；这些规章也可以随时被负责人修改。而法律则只有立法者才能修改，并且会约束所有人。

16.2.4 基于角色的访问控制（RBAC）

基于角色的权限访问控制，简称 RBAC（Role-Based Access Control），是一种将用户与角色解耦的访问控制方法。不同于基于用户的权限访问控制，

RBAC 提出了角色的概念，与权限直接相关；用户并不直接拥有权限，而是通过拥有一个或多个角色间接地拥有权限，从而形成“用户-角色-权限”的关系，其中“用户-角色”，以及“角色-权限”，一般都是多对多的关系。在后文介绍的 SELinux 中，会引入角色的概念，映射到 Linux 的系统用户，可基于角色进行权限的分配。

RBAC 相对基于用户的权限访问控制有以下好处：首先，角色是一种粒度更细的抽象，设定角色与权限之间的关系通常比设定用户与权限之间的关系更直观。其次，通过增加一层抽象，可以通过对某个角色的权限更新一次性地更新所有拥有该角色用户的权限，而不必为每个用户单独操作，提高了权限更新的效率。再次，通常来说，角色与权限之间的关系比较稳定，而用户和角色之间的关系变化相对频繁，因此可以解耦系统的设计者和管理者的任务：设计者负责设定权限与角色的关系（机制），管理者只需要配置用户属于哪些角色（策略），从而方便日常的运营和维护。这种最基本的 RBAC，被称为 RBAC0。

RBAC 虽然灵活，却也有可能导致角色爆炸问题——即在系统中存在大量的角色，甚至为每一种权限的组合都设置一个角色。为了管理角色，RBAC 还有不少扩展，包括角色继承、角色约束等。例如，某两个角色“开发实习生”和“测试实习生”都拥有 9 个相同的权限，但各自拥有 1 个不同的权限。为了简化角色与权限之间的管理，可以创建一个新的角色“实习生”，然后将这两个角色作为“实习生”的子类。这样，在角色之间就有了继承关系，这被称为 RBAC1。

同时，角色与角色之间还可以设定不同的约束关系。例如，禁止一个用户同时拥有“测试”与“质量管理”这两种角色。管理员可以设定哪些角色之间存在冲突，即任意时刻都不允许同一个用户同时拥有冲突的角色，这被称为**静态职责分离**（Static Separation of Duty）；也可以允许用户拥有相冲突的角色，但在一次登录过程中，该用户只能选择相冲突的角色中的一个，这被称为**动态职责分离**（Dynamic Separation of Duty）。支持职责分离的系统也被称为 RBAC2。还有一类 RBAC3，是 RBAC0、RBAC1 和 RBAC2 的综合。

最基本的 RBAC 与操作系统中的用户组有相类似的地方：用户组类比角色，每个用户组可以有一个或多个权限，每个用户可以属于一个或多个用户组，且用户可以动态地在不同用户组之间切换。不同的是，通常用户组和用户同时具有权限，相互独立，因此如果要基于用户组来实现 RBAC，则需要完全不为用户分配权限，而仅为用户组配置权限；这种权限控制称为 ACLg（g 表示 group）。

尽管最简化的 RBAC 与 ACLg 是等价的，但两者依然存在一些区别。其

中一个区别在于：ACL 是建立在语义层次较低的对象，而 RBAC 的语义层次更高。例如，ACL 通常是类似“允许读/写某个文件”；而 RBAC 则是类似“允许创建一个用户账号”，或“允许设置网卡 IP”，这些语义相对来说更容易让人理解。

16.2.5 Bell-LaPadula 强制访问控制模型（BLP）

BLP 是一个用于访问控制的状态机模型，设计的目的是为了用于政府、军队等具有严格安全等级的场景，是一种典型的 MAC 设计。BLP 规定了两条 MAC 规则和一条 DAC 规则，一共三个主要的属性：

1. 简单安全属性（Simple Security Property）：某个安全级别的主体无法读取更高安全级别的对象；
2. * 属性（Star Property，星属性）：某一安全级别的主体无法写入任何更低安全级别的对象；
3. 自主安全属性（Discretionary Security Property）：使用访问矩阵来规定自主访问控制（DAC）。

因此，BLP 仅允许数据从低机密程度的对象向高机密程度的对象流动。例如，有如下的文件和用户：

- 文件 A：高机密级别（机密文件）
- 文件 B：低机密级别（普通文件）
- 进程 X：高机密级别（长官创建）
- 进程 Y：低机密级别（士兵创建）

那么，根据上面的两条 MAC 规则，读与写的限制如下：

- 根据简单安全属性，进程 X 可以读取文件 A，也可以读取文件 B，而进程 Y 只能读取文件 B；
- 根据星属性，进程 X 只可以写入文件 A，而进程 Y 可以写入文件 A 或文件 B。

可以做这样的类比：长官只能写入文件 A，而士兵不能读文件 A，所以长官的信息不会向下流入士兵（即不会泄露）；相反，士兵可以写入文件 B，而长官可

以读取文件 B，所以士兵的信息可以向上流入长官（即向上汇报）。因此，这里的策略也可以简称“下读，上写”。

BLP 不允许数据从高机密级别的对象流向低机密级别的对象，但这点在实际应用中带来了很多不便。例如，长官无法向士兵传递命令，因为长官不能“向下写”任何数据。这是一个典型的安全性与可用性产生矛盾的场景。为此，BLP 引入了“受信任主体”的概念：**受信任主体**（Trusted Subject）可以不受星属性的限制，但前提是该主体必须遵守相应的“降密策略”（Declassification Policy）。具体的降密策略因不同场景而不同，需要具体情况具体设置。然而，这不可避免的引入复杂性，并没有一种简单普适的方法来保证数据可靠安全的降密。

16.2.6 能力机制（Capability）

Capability 列表是权限矩阵的实现方法之一。与访问控制列表相比，Capability 列表从主体的角度出发，列出该主体所拥有的能访问的对象及相应的权限。相比宏内核，Capability 对于微内核来说尤为重要。这是因为微内核中许多计算资源由用户态的服务进行操作，Capability 机制可以非常有效地实现对资源的访问控制与权限管理。

以进程主体为例，Capability 列表是每个进程都有的属性，但不能由进程的用户态直接访问，而是保存在内核中，内核对用户态提供对应的 Capability ID 用于操作。应用在访问某个对象或进行某项操作时，需要以该 Capability ID 作为参数，内核根据 ID 找到对应的 Capability，进行权限检查，检查通过才允许相应的操作。

如果对上述的流程觉得熟悉——是的，前文所述的文件系统 fd 机制与此非常类似：Capability ID 就是文件描述符 fd，每个进程的 fd 列表由内核管理，每个 fd 记录了本次打开文件时允许的读写权限，之后所有对文件的操作均以 fd 作为凭证。可以看到，Capability 就像是一个信息量更大的“胖指针”（Fat Pointer），一方面指向了对象，另一方面也记录了访问该对象时允许的权限。

Capability 作为一种通用的访问控制方法，不仅可以用在文件系统中，也可以用于管理其他资源。例如，在 IPC 的场景中，服务器可基于 Capability 机制来限制哪些客户端可以进行 IPC 调用。具体过程的示例如下：

1. 服务端通过系统调用创建一个 Capability，获得相应的 ID；
2. 服务端通过系统调用，将此 Capability ID 传递给某个客户端；
3. 客户端通过 IPC 调用服务端的某个服务函数，以 Capability ID 作为参数；

4. IPC 调用过程中，操作系统根据该 Capability ID 检查该客户端是否有权限调用服务端函数，检查通过则切换至服务端继续运行；

5. 服务端执行函数，并将结果返回给客户端。

操作系统可以支持 Capability 的传递和复制。在上面的例子里，客户端 A 可以把自己的 Capability 传递给客户端 B，这样客户端 A 和 B 就都有权限通过 IPC 调用服务端的函数。操作系统也可以限制 Capability 不能被传递，或能传递但不能被复制，从而支持更灵活的权限管理。

小思考

既然 fd 是一种 Capability，那么两个进程之间是否也可以传递 fd 呢？

可以。进程 *A* 可通过 *sendmsg()* 给进程 *B* 发送一个消息，以 *"SCM_RIGHTS"* 作为参数之一，告诉内核需要传递某个 *fd*；内核会直接在进程 *B* 的 *fd* 列表中新增一项，并返回给 *B* 一个新的 *fd*。新的 *fd* 很可能与进程 *A* 的 *fd* 在数值上不同，但都指向了同一个打开的文件，且权限也是相同的。

Linux 的 Capability 机制

Linux 自身有一套 Capability 机制，用于限制进程的权限操作，具体如表 16.1所示。需要注意的是，Linux 的 Capability 机制与本节所介绍的 Capability 机制有所不同。引入这套机制的初衷，是为了解决 root 用户权限过高的问题：Linux 的 root 用户具有所有的权限，一旦普通用户因为需要执行某个特权操作而切换到 root 用户，则可实际进行所有的特权操作；这破坏了最小特权原则。通过 Linux Capability 机制，可仅分配给 root 用户的某个进程一个或几个必需的权限。结合前文所介绍的 SUID 机制，可以认为 SUID 机制减小了 root 的时间窗口，而 Capability 则减小了 root 的权限窗口。

具体来说，Linux 的 Capability 机制与本节所介绍的 Capability 机制有以下不同：首先，Linux Capability 的语义都是预先由内核定义，而不允许用户进程自定义；其次，Linux Capability 不允许传递，而是在创建进程的时候，与该进程相绑定；再次，Linux Capability 并没有为用户态提供 Capability ID，因此用户态无法通过 Capability ID 索引内核资源进行操作。可以看到，本节所介绍的 Capability 机制更类似于 Linux 中 fd 机制的通用版，而 Linux Capability 则类似一种受限的、静态的、主要由内核控制的 Capability 机制。这点比较容易混淆，请辨析清楚两者之间的关系和异同。

表 16.1: Linux 的 Capability 机制（部分示例）

Capability 名称	具体描述
CAP_AUDIT_CONTROL	允许控制内核审计（启用和禁用审计，设置审计过滤规则，获取审计状态和过滤规则）
CAP_AUDIT_READ	允许读取审计日志（通过 multicast netlink socket）
CAP_AUDIT_WRITE	将记录写入内核审计日志
CAP_BLOCK_SUSPEND	允许使用可阻止系统挂起的特性
CAP_CHOWN	允许任意修改文件的 UID 和 GID
CAP_DAC_OVERRIDE	忽略对文件读、写、执行权限检查（即 DAC 访问限制）
CAP_WAKE_ALARM	允许触发一些能唤醒系统的事件（如 CLOCK_REALTIME_ALARM 计时器等）

16.3 案例分析：SELinux

本节主要知识点

- *SELinux*、*Flask*、*LSM* 三者之间的关系是什么？
- 为什么在有了文件系统的权限控制机制后，还需要 *SELinux*？
- *SELinux* 中的标签有什么作用？在什么时候会进行标签的检查？

访问控制有不同的方式和方法，操作系统需要一种机制，允许系统管理员灵活地配置不同的策略。SELinux（Security-Enhanced Linux）是一个 Linux 内核的安全模块，是 Flask 安全架构在 Linux 上的实现，提供了一套访问控制的框架，以支持不同的安全策略，包括强制类型访问（MAC）。该项目于上世纪 90 年代末由美国 NSA（National Security Agency）发起，2000 年在 GPL 许可证下开放源码，并于 2003 年合入 Linux 内核主线中。

Flask 是一个 OS 的安全架构，能够灵活地提供不同的安全策略。Flask 的全称是 “Flux Advanced Security Kernel”，由 Utah 大学、美国 NSA 和 SCC 公司[7]共同开发。1995 年，该架构开始在 Mach（Utah 版本）上进行开发，一年后转移到了一个名为 Fluke（Utah 开发）的研究性微内核操作系统上，后来移植到许多不同的操作系统框架中，包括 OSKit、BSD、OpenSolaris 和 Linux。

然而，Flask 的架构需要对内核进行大量的修改，尤其要在很多关键操作处进行权限检查，而 Linux 当时的模块框架无法支持 Flask 以模块的方式实现。为此，Linux 社区在 2002 年提出了 LSM（Linux Security Modules）项目，在内核的关键代码区域插入了许多 hook，包括在所有系统调用即将访问关键

[7]SCC（Secure Computing Corporation）是美国的一家安全公司，2008 年被 McAfee 收购。

内核对象（如 inode 或 task control block）之前，这些 hook 会调用模块实现的函数（即 upcall），进行访问控制、安全等检查。在 Linux 上基于 Flask 实现的 LSM 模块，就是 SELinux。除了 SELinux，还有其他基于 LSM 实现的安全模块，例如 AppArmor、Smack、TOMOYO Linux 等。SELinux 与 flack 以及 LSM 的关系如图 16.4所示。

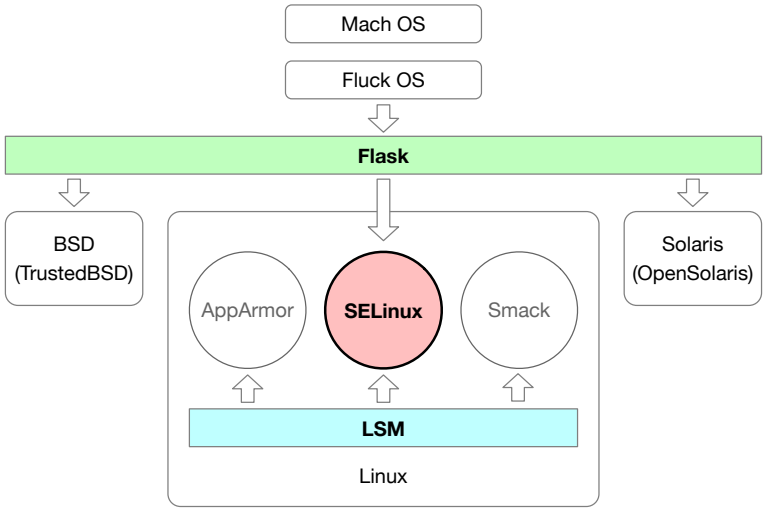


图 16.4: SELinux 是 Flask 安全架构在 Linux 的实现，具体形式是基于 LSM 的一个安全模块

16.3.1 SELinux 的基本抽象

为了能够灵活地制定安全策略，SELinux 提出了一套抽象，主要包括主体、对象、用户、策略与安全上下文。

- 主体（Subject）指访问各种对象的程序；
- 对象（Object）指系统中的各种资源，一般来说就是文件；
- 用户（User）指系统中的用户，需要注意的是，SELinux 的用户与我们前面所提到的 Linux 系统用户并没有关系；
- 策略（Policy）是一组规则（Rule）的集合，Linux 目前提供了多种策略，默认是 “Targeted” 策略，主要对服务进程进行访问控制；另一个是 MLS（Multi-Level Security），实现了 Bell-LaPadula 强制访问控制模型，对系

统中所有的进程进行访问控制；还有一个是 Minimum，类似 Targeted，但出于对资源消耗的考虑而仅应用了一些基础的策略规则，一般用于手机等平台。

- 安全上下文（Security Context）：是主体和对象的标签（Label），用于访问时的权限检查。可以通过“ls -Z”的命令来查看文件对应的安全上下文。

在上述基础上，SELinux 将访问控制抽象为一个问题：“一个 < 主体 > 是否可以在一个 < 对象 > 上做一个 < 操作 >?”。这个问题包含一个三元组，即主体、对象、操作。这些规则保存在专门的服务器中，又称**安全服务器**，一个安全服务器可以通过网络为多个 Linux 主机提供服务。在主体访问某个对象时，SELinux 会询问安全服务器，由安全服务器在规则数据库中查找并检查主体与对象对应的安全上下文（Security Context），从而判断访问是否有权限。为了提高性能，内核会将这些规则缓存在“访问向量缓存”（AVC：Access Vector Cache）中，避免每次都去询问安全服务器。

16.3.2 SELinux 的标签机制：安全上下文

SELinux 本质上是一个标签系统，所有的主体和对象都对应了各自的标签。标签就是安全上下文，其格式为“用户：角色：类型：MLS 层级”。

```
# 标签：user:role:type:category
# 例子：常见的对象安全上下文
user_u:object_r:tmp_t:s0:c0
```

输出记录 16.1: 安全上下文示例

每一项的具体解释如下：

- 用户（User）：SELinux 的用户与传统的 Linux 用户是两套机制，每个 Linux 的用户都会映射到一个 SELinux 的用户，在新建一个 Linux 用户时，可以指定两种用户间的映射关系。
- 角色（Role）：这是 RBAC 安全模型的一个属性，介于域（domain，下文介绍）和 SELinux 用户之间。用户需要认证属于哪个角色，角色需要认证属于哪个域；进程则在不同的隔离域中运行。一般来说，文件的角色通常为 object_r；程序的角色为 system_r；用户的角色，对于 targeted

policy 来说，一般为 system_r，而对 strict policy 来说，包括 sysadm_r、staff_r 和 user_r。一个用户可以有多个角色，但一次只能使用一个角色。

- 类型（Type）：类型是类型强制（Type Enforcement，TE）的一个属性。每个对象都有一个 type，一个进程的类型相对特殊，通常被称为域（domain）。SELinux 的策略规则定义了类型之间如何访问，区分是一个域访问一个类型，还是一个域访问另一个域。

- 安全级别（Sensitivity）：级别是一个可选项，可由组织自定义；一个对象有且只有一个安全级别。安全级别包括 0-15 级，Targeted 策略默认使用 s0。

- 类别（Category）：这是一个可选项，可由组织自定义；共 1024 个分类，一个对象可以有多个类别。Targeted 策略不使用类别。

在安全上下文的各项中，类型是最重要的，SELinux 可以基于类型来制定策略，即何种类型的主体可以访问何种类型的对象。每个进程的安全上下文中的 role 和 domain，为启动该进程的用户的 role 和 domain；其中，权限与 domain 最为相关，而 domain 的转换则由 role 来控制，role 的转换则依赖于用户的 identity。

具体来说，当用户登录到系统后，SELinux 会根据用户对应的 role，分配给用户一个默认的安全上下文。这个安全上下文定义了当前的 domain，因此所有新的子进程均属于同样的 domain。通常来说，一个 role 会对应一个 domain，可通过命令`newrole -r role_r -t domain_t` 来切换 role 和 domain（SELinux 有对应的规则来判断是否允许切换）。对于一些重要的服务进程，SELinux 会将其标记为特定的 domain。例如，/usr/sbin/sshd 的类型为 sshd_exec_t，当程序开始运行时会自动切换到 sshd_t 的 domain。这个自动切换的机制能够保证一个进程仅仅拥有其所需要的权限，而不是启动该进程的用户所有的权限——很多时候这两组权限是不同的。图 16.5给出了一个在 Debian 平台下关键进程 domain 自动切换的例子[8]：

每个文件（包括设备、socket 等对象）的标签称为 type。一个 domain 的权限由一组“允许/不允许访问某个 type”的规则组成。这里可能会有同学产生疑问：进程的 domain 和对象的 type 是什么关系呢？在安全上下文的各项中并没有 domain，只有 type。事实上，domain 就是 type 的一种，是专属于进程的 type。因此，进程的安全上下文中，domain_t 中的“_t”后缀表示 type。具体

[8] https://debian-handbook.info/browse/stable/sect.selinux.html

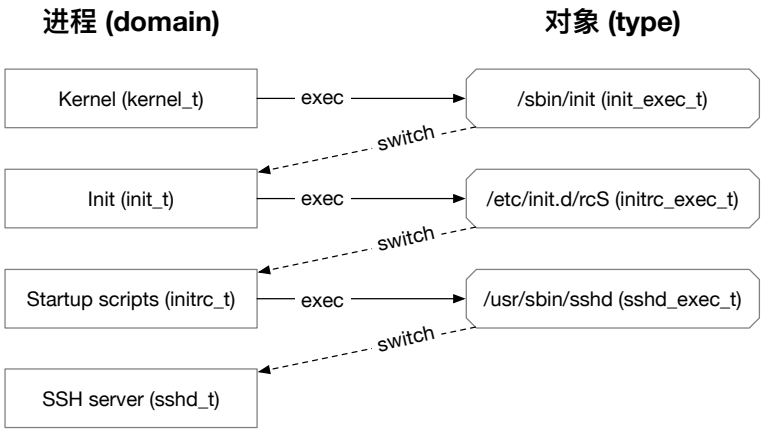


图 16.5: Debian 中关键服务进程 domain 的自动切换

来说，SELinux 会给文件一个 type 标签，给进程一个 domain 标签；domain 能够执行的操作安全策略里定义。

```
# 规则： allow domains types:classes permissions;
# 例子：所有sys_domain的进程都可读写带sys_data_file标签的文件
allow sys_domain sys_data_file:file rw_file_perms;
```

输出记录 16.2: 规则示例

16.3.3 SELinux 实例解析

SELinux 中主要用到三个命令：`semanage`、`chcon`、`restorecon`。

- semanage：查询、修改、增加、删除对象的默认类型；
- chcon：修改对象的安全上下文；
- restorecon：恢复对象的安全上下文为默认的安全上下文。

下面我们通过实际的例子来具体看下如何操作对象的标签，以及对象的标签如何影响不同主体的访问。需要注意，在这里 “标签” 是一个泛指：有时指安全上下文，有时候也可特指安全上下文中的`type`。以下操作基于 CentOS-8.1 进行，不同发行版，统一发行版的不同版本在操作上可能不完全一致。

打开 SELinux

SELinux 有三种模式：Disable、Permissive、Enforcing。

- Disable：关闭，SELinux 不运行；
- Permissive：SELinux 运行但仅监控系统，并不会干预系统运行；
- Enforcing：SELinux 监控并且干预系统，会拒绝不符合规则的访问，是真正起作用的模式。

具体配置一般在/etc/sysconfig/selinux文件中。

```
[root@CentOS-8 ~]# cat /etc/sysconfig/selinux
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled - No SELinux policy is loaded.
SELINUX=enforcing
# SELINUXTYPE= can take one of these three values:
#     targeted - Targeted processes are protected,
#     minimum - Modification of targeted policy. Only selected
    processes are protected.
#     mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

输出记录 16.3: SELinux 的配置文件：/etc/sysconfig/selinux

注意文件中的SELINUX项，将其设置为enforcing后保存，然后重启。如果是第一次开启 SELinux，重启时系统会自动对所有文件加上标签，可能需要一点时间。重启完成后，可通过sestatus查看当前的状态，若为 “enabled” 则表示 SELinux 已经启动成功。

```
[root@CentOS-8 ~]# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   enforcing
Mode from config file:          enforcing
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
```

```
Memory protection checking:     actual (secure)
Max kernel policy version:      31
```

输出记录 16.4: SELinux 的状态

SELinux 的标签操作与访问控制

接下来我们以 apache 服务器为例来看 SELinux 中标签的操作，以及基于标签的访问控制的具体应用[9]。我们先看下面的一组命令：

```
[root@CentOS-8 ~]# cat "Welcome to Apache! This is page-1." > page
    -1.html
[root@CentOS-8 ~]# cat "Welcome to Apache! This is page-2." > page
    -2.html
[root@CentOS-8 ~]# ls -lZ
total 14
... unconfined_u:object_r:admin_home_t:s0  ...  page-1.html
... unconfined_u:object_r:admin_home_t:s0  ...  page-2.html
[root@CentOS-8 ~]# cp page-1.html /var/www/html/
[root@CentOS-8 ~]# mv page-2.html /var/www/html/
[root@CentOS-8 ~]# cd /var/www/html/
[root@CentOS-8 html]# chown apache: page*
[root@CentOS-8 html]# ls -lZ
total 12
... unconfined_u:object_r:httpd_sys_content_t:s0 ... index.html
... unconfined_u:object_r:httpd_sys_content_t:s0 ... page-1.html
... unconfined_u:object_r:admin_home_t:s0        ... page-2.html
```

输出记录 16.5: SELinux 示例：文件标签与权限控制

在上面这段操作中，首先生成了两个简单的页面，`page-1.html`和`page-2.html`，对`page-1.html`我们将其复制到`/var/www/html`目录，也就是 apache 的默认根目录；而对`page-2.html`，我们将其移动到了同样的目录。然后我们通过浏览器打开这两个页面，发现`page-1.html`可以正常显示，而`page-2.html`则报错“访问被禁止”，具体如图 16.6。这是为什么呢？

因为两个页面文件的标签不同。回到前面的代码片段，通过`ls -lZ`来显示文件的安全上下文。在一开始创建两个文件后，它们的标签是一样的，具体如下：

[9]部分参考自：https://debian-handbook.info/browse/stable/sect.selinux.html

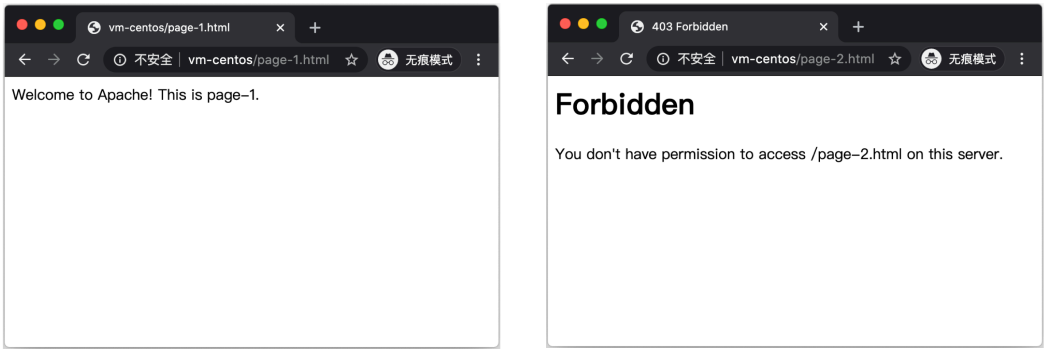


图 16.6: SELinux 应用示例：两个具有不同的 type 的页面浏览效果

- `unconfined_u`，表示用户（user）的类型。可以通过`semanage login -l`来查看所有的用户，如代码 16.6。注意，这里的用户是 SELinux 的用户，与 Linux 的用户存在映射关系。一个对象的用户类型来自操作进程；
- `object_r`，表示角色（role）是对象，这是文件的默认值；
- `admin_home_t`，表示类型（type），一个文件的类型取决于很多因素，包括文件所在的目录；
- `s0`，表示 MLS 的安全级别为 0 级，同样来自于操作进程。

当把两个页面文件分别“复制”和“移动”到网页服务器的根目录后，再次检查它们的标签，发现`page-1.html`的类型变成了`httpd_sys_content_t`，而`page-2.html`的类型没有变化。这是因为，一个文件的安全上下文保存在其 inode 中的扩展属性。“复制”操作会在目标目录新建一个文件（新建一个 inode），其类型由创建时所在的目录决定；而“移动”操作则并没有新建文件的操作（只是在目标文件中加了一个目录项，并没有新建 inode），因此文件的类型不变。

```
[root@CentOS-8 ~]# semanage login -l
Login Name    SELinux User    MLS/MCS Range    Service

__default__   unconfined_u    s0-s0:c0.c1023   *
root          unconfined_u    s0-s0:c0.c1023   *
```

输出记录 16.6: 使用 semanage 命令查看用户映射

那么为什么page-1.html可以正常显示，page-2.html的访问则被禁止呢？因为 CentOS 对 apache 应用了特定的 SELinux 规则：只能显示标签为httpd_sys_content_t的文件。因此，当 apache 进程访问标签为admin_home_t的page-2.html时，SELinux 通过 LSM 在文件访问前的 hook 进行了权限检查，并禁止了访问；同时，在/var/log/audit/audit.log中做了记录。

```
[root@CentOS-8 ~]# netstat -Ztulpen
Active Internet connections (only servers)
... PID/Program name     Security Context
... 1069/sshd            system_u:system_r:sshd_t:s0-s0:c0.c1023
... 5120/httpd           system_u:system_r:httpd_t:s0
... 1069/sshd            system_u:system_r:sshd_t:s0-s0:c0.c1023
... 1061/NetworkManager  system_u:system_r:NetworkManager_t:s0
```

输出记录 16.7: 查看系统服务的安全上下文（有删减）

如何才能让page-2.html能够被网页服务器正常访问呢？可以通过restorecon或chcon这两个命令。其中，chcon可以将文件的类型设置为任意预定义的类型，而restorecon则是将文件设置为 SELinux 默认的安全上下文（等价于在当前目录重新创建该文件得到的安全上下文）。

```
[root@CentOS-8 html]# chcon -t admin_home_t page-1.html
[root@CentOS-8 html]# restorecon page-2.html
[root@CentOS-8 html]# ls -lZ
total 12
... unconfined_u:object_r:httpd_sys_content_t:s0 ... index.html
... unconfined_u:object_r:admin_home_t:s0        ... page-1.html
... unconfined_u:object_r:httpd_sys_content_t:s0 ... page-2.html
```

输出记录 16.8: 通过“restorecon”和“chcon”命令修改文件的安全上下文

SELinux 的策略与规则

在上面的例子中，我们已经体验到了 SELinux 设置的许多规则，包括：

- 在某个目录下创建新文件，会给该文件设置对应的标签（type）；
- 服务进程有自己的标签（domain），决定了其能访问哪些文件；

表 16.2: SELinux 系统上 Apache 服务器的标签

类型	文件	标签
二进制	/usr/sbin/httpd	httpd_exec_t
配置目录	/etc/httpd	httpd_config_t
日志文件目录	/var/log/httpd	httpd_log_t
内容目录	/var/www/html	httpd_sys_content_t
启动脚本	/usr/lib/systemd/system/httpd.service	httpd_unit_file_d
服务进程	/usr/sbin/httpd	httpd_t
网络端口	80/tcp, 443/tcp	httpd_t, http_port_t

- 不同的 Linux 用户会对应到不同的 SELinux 用户，等等。

那么这些规则是如何设置的呢？是否可以新增规则和删除规则呢？SELinux 中的规则很多，而且很复杂，一不小心就有可能导致规则与规则之间产生冲突。为此，编写规则需要非常小心。Linux 将一些已经过实践验证的规则组合在一起，形成策略，管理员可以在不同的策略之间进行选择（CentOS 可通过修改`/etc/selinux/config`文件切换策略）。例如默认的`targeted`策略仅对服务进程做出了限制，而`mls`策略则会有更复杂的访问控制机制。

我们可以通过`semanage`命令来查看部分规则，如代码 16.9所示。可以看到这条规则明确了在`/var/www`目录下新建文件的默认安全上下文，这也是`restorecon`命令所用到的安全上下文。

```
[root@CentOS-8 html]# semanage fcontext -l | grep
    httpd_sys_content_t
...
/var/www(/.*)?  all files  system_u:object_r:httpd_sys_content_t:s0
...
```

输出记录 16.9: 文件的安全上下文的默认规则

与 apache 网页服务器相关的标签如表16.2所示。其中，一个`domain`为`httpd_t`的进程可以访问标记为`httpd_xxx_t`标签的对象。

16.4 操作系统内部安全

本节主要知识点

- ❑ 操作系统有哪些常见的漏洞？这些漏洞有哪些危害？
- ❑ 有哪些常见的针对操作系统的攻击？
- ❑ 如果操作系统变得恶意，会如何攻击应用程序？

由于操作系统拥有系统权限，一旦出现安全漏洞并被攻击者利用，后果比普通应用程序出现漏洞的影响更大。攻击者若获得了操作系统的控制权，则可以绕过包括访问控制在内的所有基于操作系统的安全机制，从而窃取或篡改系统中属于其他应用程序的关键数据，或直接对硬件设备进行非法操作从而造成系统毁坏。操作系统的实现常常包含上千万行代码，因此不可避免地存在安全漏洞。针对操作系统的攻击有很多，可以从三个方面对攻击进行归类：

- **漏洞类型**：指攻击所利用的漏洞类型，包括栈/堆缓冲区溢出错误、整型溢出错误、空指针/指针计算错误、内存暴露错误、use-after-free 错误、格式化字符串错误、竞争条件错误、参数检查错误、认证检查错误等；
- **漏洞组件**：指攻击所利用漏洞的所在的内核模块，包括调度模块、内存管理模块、通信模块、文件系统、设备驱动等；
- **攻击效果**：指攻击的目的或攻击导致的结果，包括提升权限、执行任意代码、内存篡改、窃取数据、拒绝服务、破坏硬件等。

上述分类还可以进一步细化：例如“内存篡改”这一类，还可以进一步细化为“任意内存写任意值”、“固定地址写任意值”、“任意地址写固定值”这些子类。不同类型的攻击所占比例也不同。根据 cvedetails 的统计，在 Linux 所有的 CVE 中拒绝服务攻击占 45%，溢出漏洞占 14% [10]。本章将会介绍常见的操作系统安全漏洞，以及相应的防御方法。

16.4.1 整型溢出漏洞

操作系统内核通常使用较底层的语言来实现，例如 C 语言。使用底层语言能够更方便地直接操作硬件，但也更容易犯错。整型溢出漏洞就是其中的一

[10] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html

种错误。在 32 位的系统中，一个无符号的 int 数在累加到 2^{32} 后，再加 1 就会变成 0；一个有符号的 int 数在累加到 2^{31} 后，再加 1 就会变成 -2^{31}。

```
1 unsigned long count = ... ; // from user space
2 if (count > 1<<30)
3   return -EINVAL;
4 table = vmalloc(sizeof(struct rps_dev_flow_table) +
5         count * sizeof(struct rps_dev_flow));
6 ...
7 for (i = 0;i < count; i++)
8   table->flow[i] = ...;
```

代码片段 16.1: 整型溢出导致错误的边界检查

在代码片段 16.1中，count变量来自用户空间，尽管对其长度进行了检查，但这个检查是不够的：因为sizeof(struct rps_dev_flow)的长度是8，因此如果count的长度正好是 2^{30}，那么vmalloc的参数依然会造成溢出。由于count远远大于 vmalloc 得到的内存大小，因此在后续for循环中便会写入超出table边界的内存区域。

16.4.2 Return-to-user 攻击

在许多平台，内核与应用程序共享同一个页表[11]，内核运行时可以任意访问用户态的虚拟地址空间，这虽然方便了内核与应用之间的交互，例如内核可以通过指针直接访问用户态的任何地址，但这种方便也很容易导致内核错误的执行位于用户态地址空间的代码。这种攻击称为“return-to-user”攻击，简称“ret2usr” [5]。

攻击者如何主动让操作系统内核执行用户态的代码呢？一个最直观的方式是，先在用户态中初始加载一段恶意代码，然后利用内核的某个漏洞，修改内核中的某个函数指针指向这段恶意代码的地址，这样当内核执行这个函数指针的时候就会执行这段恶意代码。除了函数指针，也可以是利用内核的栈溢出漏洞，覆盖栈上的返回地址，修改为恶意代码的地址，使内核在执行 ret 指令时跳转到位于用户态的恶意代码。或者可以利用内核中的空指针错误，让内核跳转到 0 地址执行（通常 0 地址都是映射在用户空间）。注意：虽然应用默认不使用 0 地址，但在 Linux 以及其他一些操作系统中，程序可以主动将任意内存页映射到 0 地址。一个原因是为了更好地保证兼容性：x86 的 vm86 模式总是

[11]若内核采用了 KPTI 机制，则内核与应用程序不共享页表。

从 0 地址开始执行，而且有些模拟环境也有类似的需求；更详细的解释可参见 Linus Torvalds 在邮件列表中的讨论 [12]。代码 16.2是 Linux 中曾经存在的一个 bug：由于没有检查 sendpage 字段，攻击者可主动将其设置为 0，从而触发 ret2usr 攻击。

```
sock = file->private_data;
flags = !(file->f_flags & O_NONBLOCK) ?
        0 : MSG_DONTWAIT ;
if (more)
  flags |= MSG_MORE;
// sendpage could be NULL
return sock->ops->sendpage(sock, page, offset,
                           size , flags );
```

代码片段 16.2: Linux 中的 NULL 函数指针（*net/socket.c*）

为了解决 ret2usr 攻击，第一种方法是仔细检查内核中的每个函数指针，并在执行 ret 指令之前确保返回地址在内核态。这种方法需要对内核所有模块进行检查，很难做到 100% 的覆盖率。第二种方法是在陷入内核时修改页表，将用户态所有的内存都标记为不可执行，这样万一由于 bug 跳转到用户态，CPU 就会触发异常。第二种方法对性能的影响是非常明显的：由于修改页表后必须要刷新 TLB 才能生效，因此修改页表、刷新 TLB，以及后续运行触发 TLB miss 都会导致性能下降；而且，在返回用户态之前必须将页表恢复，并再次刷掉 TLB，这样又会导致用户态执行时出现 TLB miss，因此对性能的影响非常大。第三种方法则是依靠硬件，保证 CPU 处于内核态时不得运行任何映射为用户态可访问的代码，否则触发异常。这种方法对性能没有影响，也不需要修改任何内核中已有的代码。Intel 的 SMEP（Supervisor Mode Execution Prevention）技术就是这么做的。此外，Intel 还提供了 SMAP（Supervisor Mode Access Prevention）技术，防止内核不小心读写任何映射为用户态的数据。ARM 同样有类似 SMAP 和 SMEP 的技术，分别称为 PAN（Privileged Access Never）和 PXN（Privileged Execute-Never） [13]。

然而，SMEP/PXN 并不能完全解决 ret2usr 的攻击。许多操作系统为了更方便地管理物理内存，通常会将一部分或所有的物理内存映射到一段连续的内核态虚拟地址空间，这种映射称为直接映射。这意味着，同一块物理内存在系

[12] https://yarchive.net/comp/linux/address_zero.html

[13] 需要注意的是，在 AArch32 平台上，只能在 LPAE 模式才能开启 PXN（LPAE 即 Large Physical Address Extensions，允许 32 位处理器访问大于 4GB 的物理内存），同时会禁止 MMU domain 功能，因此 Linux 并没有在 AArch32 平台直接开启 PXN。

统中有多个虚拟地址：例如，某个内存页分配给了应用程序，那么内核既可以通过应用程序的虚拟地址访问，也可以通过直接映射的虚拟地址访问。因此，攻击者如果能够推算出位于用户态的恶意代码在内核直接映射区域的虚拟地址，就可以在 ret2usr 攻击中让内核跳转到该地址执行，从而绕过 SMEP 的保护。攻击成功还有一个前提：直接映射区域必须是可执行的——而 3.8.13 以及之前的 Linux 版本恰恰将直接映射区域的权限设置为“可读可写可执行”。这种利用直接映射区域的 ret2usr 攻击被称为“ret2dir”攻击 [4]。

16.4.3 Rootkit：获取内核权限的恶意代码

Rootkit 是指以得到 root 权限为目的的恶意软件。Rootkit 可以运行在用户态，也可以运行在内核态。运行在用户态的 rootkit 可以将自己注入到某个具有 root 权限的进程中，并接收来自攻击者的命令。例如，某个具有 root 权限的进程存在栈缓冲区溢出漏洞，攻击者可通过该漏洞覆盖栈上的返回地址，操纵其执行任意代码，从而将该进程变成一个具有 root 权限的恶意进程。

运行在内核态的 rootkit 形态较为多样，可以是 hook 了某个内核中的关键函数，从而在该函数被调用时触发 hook 运行；也可以是以内核线程的方式运行。常见的一种内核态 rootkit，是修改内核中的系统调用表，用恶意代码来替换掉正常的系统调用；同时，用户态的恶意应用能通过发起系统调用的方式来操纵恶意代码的运行。

Rootkit 在运行时，通常会尽可能隐藏自己的信息。例如，在 Linux 中，列出所有进程的命令是`ps aux`。Rootkit 会试图不让自己出现在上述命令的输出结果中，达到隐藏的目的。*adore-ng* 是一个著名的 Rootkit，它隐藏自己的方式，是将自己的进程信息从内核的`task-list`中删除，却依然保留在内核的`runqueue`中。由于`ps aux`命令只会遍历`task-list`数据结构，因此输出结果中便不会包含 rootkit 的信息。

16.5 可信执行环境（TEE）

本节主要知识点

- ❑ 使用可信执行环境为什么可以不信任操作系统？
- ❑ 可信执行环境目前主要用来保护哪些代码和数据？
- ❑ *ARM*、*Intel* 和 *AMD* 的可信执行环境，各自有哪些特点？

❑ 可信执行环境能否仅依赖软件？

操作系统本身由于过于复杂，不可避免地存在安全漏洞。**可信执行环境**（Trusted Execution Environment，TEE）是计算机系统中一块通过底层软硬件构造的区域，在假设操作系统是恶意的前提下，依然可以保证加载到该区域的数据和代码在执行过程中的完整性和隐私性。TEE 可以由底层硬件直接构造（如 Intel SGX），也可以由底层软硬件协同提供（如基于 Hypervisor 与底层的 TPM 硬件，或基于隔离域中的安全操作系统）。

可信执行环境目前在工业界得到了较广泛的应用，已被部署在手机、服务器、车载设备以及其他形态的物联网设备中，负责保护指纹、证书、金融等关键数据，支撑着包括移动支付、生物特征识别等具有高安全要求的应用。在云平台，亚马逊于 2017 年推出了支持 Intel SGX 技术的云服务器，允许用户在其服务器中使用 Enclave 硬件特性。同年，微软推出了基于 Intel SGX 的 ACC（Azure Confidential Computing）产品[14]。2018 年，Google 也基于 SGX 开发了 Asylo 系统[15]，对 Enclave 进行了一层封装，允许用户利用硬件 Enclave 在其云平台构建可信的数据处理应用。在国内，阿里云也已经推出支持 Intel SGX 的裸金属（baremetal）服务器。2020 年 8 月，Google 推出了机密虚拟机（Confidential VM）的功能[16]，基于 AMD 的安全加密虚拟化技术（Secure Encrypted Virtualization，SEV），将整个虚拟机作为一个 TEE，不必信任属于云平台的 hypervisor，并为用户提供更好的兼容性。在移动平台，TEE 通常基于 ARM TrustZone 技术。目前所有手机的指纹识别都强制要求将对指纹、虹膜等生物特征的采集和识别任务部署在 TrustZone 所提供的可信执行环境中，即使主系统（如 Android 或 iOS）被攻击者完全控制，用户的生物特征数据依然受到保护。在 IoT 平台，TEE 同样在不断发展，目前 RISC-V 平台开源的 TEE 项目包括加州大学伯克利分校的 Keystone Enclave 与上海交通大学的蓬莱 Enclave。

16.5.1 TEE 的两个特性

TEE 具有两个特性：**远程认证**（Remote Attestation）和**隔离执行**（Isolated Execution）。远程认证是指一段代码和数据可以向其他人证明其确实在一个

[14]微软 ACC：https://azure.microsoft.com/en-us/solutions/confidential-compute/

[15]Google Asylo 系统：https://asylo.dev/docs/guides/sgx_release_enclaves.html

[16]Google 机密虚拟机：https://cloud.google.com/compute/confidential-vm/docs/about-cvm

TEE 中，且该 TEE 中没有其他代码和数据。隔离执行是指 TEE 中的一段代码开始执行后，TEE 外的任何软件都不能影响到其执行的过程，也无法窃取 TEE 内部的数据。远程认证保证加载过程的安全性，隔离执行保证运行过程的安全性。这两个过程如果都安全，那么 TEE 中整个应用程序的生命周期都是安全的。

TEE 的一个典型使用场景，是企业将应用程序部署到不可信的云平台，处理输入数据，并得到处理结果。假设代码的二进制并不需要保密，我们仅需要保护代码执行的完整性、数据（包括输入数据与处理结果）的完整性和隐私性。那么，典型的应用 TEE 的方式包括如下步骤：

1. 加载应用，即客户端要求云平台将应用的二进制加载到一个 TEE 中；

2. 远程认证，即客户端生成一个随机数 nonce 发送给云平台，要求云端的 TEE 生成一个证明，于是相关的硬件会计算 TEE 中所加载的所有代码的哈希值，与 nonce 和一个临时秘钥一起组成一个证明，用 TEE 硬件的私钥进行签名；

3. 客户端收到证明后，去第三方机构验证该签名的有效性，若签名有效，则表示代码确实运行在一个合法的 TEE 中；

4. 客户端可用证明中的临时秘钥与 TEE 建立起一个安全的网络信道，并通过该信道安全地与 TEE 交换数据，包括输入数据和结果。

上述过程有一个假设，即远程认证中的硬件签名所使用的私钥是安全可信的——这也是 TEE 最重要的信任根。如果该秘钥发生了泄露，则 TEE 所有的信任都会崩塌：一个攻击者可以轻易构造假的证明用于远程验证，欺骗用户与一个并不运行在 TEE 中的应用进行网络传输，从而窃取用户的所有数据，或者篡改执行结果。由于该秘钥的安全与硬件厂商的信誉（股价）直接挂钩，因此通常硬件厂商都会部署最严格的保护措施。

TEE 的隔离执行特性则依赖于不同系统的实现，在隔离粒度、隔离能力、隔离方式等方面均有不同。其中，隔离粒度通常包括应用级、操作系统级、虚拟机级等；隔离能力包括防御软件攻击和防御物理攻击；隔离方式则包括体系结构隔离、软件隔离和加密隔离。下文将从使用的角度，根据 TEE 的不同隔离粒度，挑选三个典型的 TEE 技术进行介绍。

16.5.2 应用级 TEE：以 Intel SGX 为例

Intel Software Guard eXtension (SGX) 是 Intel 提出对 CPU 架构的一系列扩展[17]，其构造出一个与外界隔离的运行“飞地”（Enclave），为应用程序提供数据和代码执行的完整性与机密性保证，即使当操作系统内核和 Hypervisor 等特权极软件是恶意的情况下，依然能保证运行在 Enclave 中的应用程序安全。

SGX 通过隔离和加密两个技术来保证 Enclave 的机密性与完整性。支持 SGX 的系统在启动时，首先在物理内存中隔离出一部分作为保留区域，称为 PRM（Processor Reserved Memory ）。CPU 只允许 Enclave 访问 PRM，拒绝来自操作系统、Hypervisor 和外部设备 DMA 对 PRM 的访问。PRM 中的大部分内存用于保存 Enclave 相关的数据，称为 EPC（Enclave Page Cache）。另一部分 PRM 则用于保存关于 EPC 的元数据：CPU 会追踪 EPC 中每个页的状态，确保每个页面只会被分配给一个 Enclave，这部分元数据称为 EPCM（Enclave Page Cache Metadata）。

为了保证数据的机密性，SGX 将所有的 EPC 都用 AES 算法加密，从而保证即使攻击者可以通过物理攻击的方式绕过软件直接读取内存，所得到的也只是密文。加密的密钥封在 CPU 中，每次启动随机生成。为了保证 CPU 能正常运行，当数据从 EPC 加载到 CPU 的 Cache 时才进行解密；换句话说，只有 Cache 中的数据才是明文，安全的边界就是 CPU——仅假设 CPU 内部是安全可信的，CPU 的外部均不安全。

对数据的加密采用了基于计数器的加密方式，加密数据时，生成的密文是数据与计数器的混合。每次写入内存时，对应的计数器会 +1，从而保证即使两次写入内存的明文数据是一样的，相应的密文数据也是不同的，将写内存时暴露的信息降至最低。

为了保证数据的完整性，防止数据被 Enclave 之外的代码恶意篡改，SGX 对所有的 EPC 都计算了 hash，并采用 hash tree 的数据结构进一步对 hash 计算 hash，层层计算后得到一个 hash root，最后将该 hash root 保存在 CPU 内部。这种类型的数据结构通常被称为**默克尔哈希树**（Merkle Hash Tree，MHT）[18]。每次写入内存时，均对整个 hash tree 上所有相关的节点进行更新；每次读取内存时，均对所读取数据重新计算 hash，并与 hash tree 中的节点进行比较以确认数据没有被恶意修改。

[17]Intel SGX：https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html

[18]https://en.wikipedia.org/wiki/Merkle_tree

由于每次内存写操作均会更新整个哈希树上的相关节点，因此会对性能产生较大的影响。为了优化，CPU 除了保存 hash root 之外，还会保存哈希树的最上面几层，从而降低写内存的时延。然而，由于 CPU 内部存储有限，限制了 hash tree 的大小，进而限制了 EPC 的大小。SGX 在刚推出时只支持 128MB 的 EPC，之后扩展到 256MB，但依然非常有限。注意这里的内存限制仅仅是物理内存的限制，对应用程序来说是透明的；换句话说，Enclave 的虚拟地址空间是没有限制的，只是其在同一时刻能使用的物理地址最多只有 256MB。若运行在 Enclave 中的应用需要使用大于 256MB 的内存，则会发生换出（swap-out）操作；这里的换页不是从物理内存交换到存储设备，而是从 EPC 交换到不受保护的内存区域。为了保证换页操作本身的安全，CPU 提供了专门的指令，将一个 4KB 的内存页重新加密并计算 hash，将加密后的数据放在非 EPC 的内存中，并在 CPU 内部记录对应的元数据，用于在换入（swap-in）的时候进行相应的完整性检查。

16.5.3 操作系统级 TEE：以 ARM TrustZone 为例

TrustZone 是 ARM 在 2002 年提出的一种隔离机制。与 Intel SGX 仅依赖 CPU 不同，ARM TrustZone 是包含了 CPU、总线结构和系统外围设备在内的一整套安全扩展。

TrustZone 从逻辑上将整个系统分为**安全世界**（Secure World）和**普通世界**（Normal World），计算资源可以被划分到这两个世界。安全世界可以不受限制地访问所有的计算资源，而普通世界不能访问被划分到安全世界的计算资源。具体来说，对于 CPU 部分，系统控制以及系统状态的切换只能在安全世界进行操作；对于内存部分，系统会阻止普通世界访问指定为安全物理内存的区域；对于设备部分，任何一个设备都可被划分到两个世界中的一个。

这两个世界拥有完全独立的软件栈，包括操作系统、Hypervisor 等系统软件也完全独立。两个世界的切换通过 smc（secure monitor call）指令实现。由于在两个世界之间切换需要考虑状态的保存和恢复，ARM 又引入了一个新的模式，“Monitor 模式”，用来实现两个世界的切换。这个 Monitor 模式本质上依然属于安全世界，但其权限比安全世界的 Hypervisor 更高。

由于安全世界的所有软件栈与普通世界完全隔离，因此即使普通世界存在恶意的软件，也无法直接窃取或篡改安全世界中的应用和数据。然而，由于 TrustZone 只用了隔离机制，因此对于来自硬件的攻击（例如通过监听总线直接读取内存数据）并不能直接防御，这是因为安全世界的内存中的数据依然是明文。

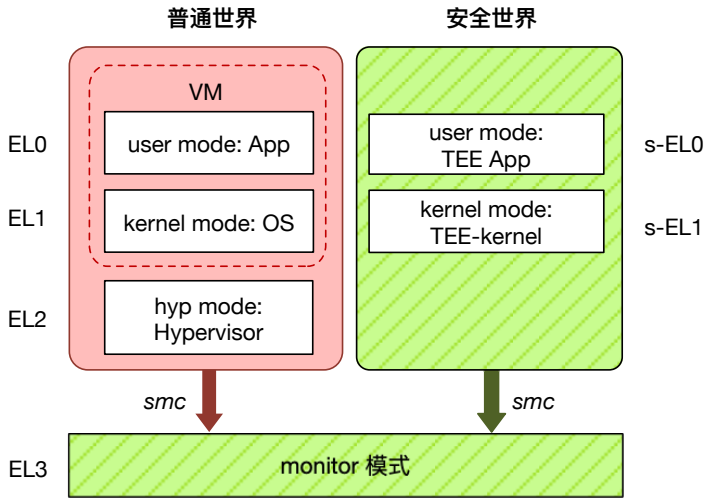


图 16.7: TrustZone 的架构

为了防御上述硬件攻击，一种方法是利用 ARM CPU 内部的存储，即 internal RAM 或 cache。ARM CPU 允许通过软件设定部分 cache 不写回到内存，可以利用这部分 cache 来保存明文数据（类似 SGX 中的 EPC），当 cache 不够的时候，则通过换页机制将数据加密并计算 hash 后放到内存中（类似 SGX 的换页机制）。然而，由于 ARM CPU 内部的 cache 容量非常小，通常只有数百 KB，因此这种方法对性能的影响非常大，可能带来几十倍甚至上百倍的性能开销 [2]。

目前 TrustZone 已经被广泛使用，在手机中，所有与生物特征识别相关的存储和运算，包括指纹识别、人脸识别、虹膜识别等，均由部署在安全世界的软件完成，包括传感器驱动、指纹匹配算法，第三方应用，例如支付宝和微信支付，也可通过在安全环境中部署相应的安全应用来实现指纹快捷支付等功能。

除了用于保护关键数据，系统级 TEE 还可被用于保护操作系统。例如，在 TrustZone 的安全环境中可以部署内存安全检查模块，定期地或在某些关键节点触发对普通世界的内存扫描和检查，类似前文提到的虚拟机安全自省功能。这正是利用了安全世界的权限更高、有能力访问普通世界所有内存的特点。

16.5.4 虚拟机级 TEE：以 AMD SEV 为例

虚拟机级 TEE 的思路是将整个虚拟机作为一个隔离对象，保证外部的 hypervisor 无法直接读写虚拟机内部的任何信息。这种隔离的粒度非常适用于云平台：云厂商提供 hypervisor，云用户上传自己的虚拟机并运行；通过虚拟机级的 TEE，云用户不需要信任云厂商，从而很好的解决了一直以来云用户担心的安全问题。理论上说，在不考虑侧信道攻击的前提下，使用了虚拟机级 TEE 后，用户虚拟机运行在云端的安全性与运行在本地的安全性可以是等价的。

安全加密虚拟化（Secure Encrypted Virtualization，SEV）是 2016 年 AMD 在 Zen 平台推出的技术 [3]。SEV 利用 AMD SME（Secure Memory Encryption）技术，使用不同的秘钥来加密不同虚拟机和 hypervisor 的内存。因此，hypervisor 若直接读取客户虚拟机的内存数据，只会得到密文。SEV 技术在刚推出时，依然需要部分信任 hypervisor；后期进一步推出了 SEV-ES（SEV Encrypted State）和 SEV-SNP（SEV Secure Nested Paging）技术，进一步降低了对 hypervisor 的安全依赖。其中，SEV-ES 增加了对虚拟机寄存器的加密，使得 hypervisor 在处理 VMExit 的时候，无法直接获取虚拟机的寄存器状态。SEV-SNP 则进一步保护虚拟机加密内存的完整性，防止恶意 hypervisor 对虚拟机内存的篡改。

根据 AMD 的文档[19]，具体来说，SEV-SNP 引入了**反向映射表**（Reverse MaP table，RMP）和**页验证**（Page Validation）两项技术。其中，RMP 表用于记录与跟踪每个内存页的拥有者（某个虚拟机或 hypervisor），当某个内存页第一次被访问时会发生 TLB miss，此时 CPU 会根据页表找到该页对应的物理地址，并以该物理地址为索引在 RMP 表中找到对应的 RMP 项，其中记录了该页的拥有者以及所映射的 GPA 地址。然后 CPU 检查拥有者是否与当前 CPU 的运行主体一致，若不一致则报错，若一致则正常填充 TLB 项。页验证技术则用于保证 RMP 表构造过程的安全性。RMP 表的每一项除了记录虚拟机的 GPA 外，还有一个验证位（Validation bit），初始化为 0；此时虚拟机无法访问该页。虚拟机必须显示调用一条新的指令——PVALIDATE——来验证该内存页的映射，从而将验证位置 1；这条指令只有虚拟机内部才能调用。验证的目的是为了保证某个物理内存页应当仅映射到一个 GPA。客户虚拟机可以尽在启动的时候对每个内存页进行验证，并拒绝在运行时验证任何新的内存页，以迫使 hypervisor 无法动态修改虚拟机的内存映射；虚拟机还可以主动

[19]AMD SEV-SNP 文档：https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

维护一个列表，记录其验证过的所有内存页，并在 hypervisor 修改其内存映射时，验证新的内存映射没有违反一一映射的原则。

2020 年，Intel 推出了 TDX（Trusted Domain eXtension）技术[20]，同样支持虚拟机级 TEE，通过体系结构隔离与硬件内存加密的方式，保护虚拟机不受非可信 hypervisor 的威胁。

16.6 侧信道与隐秘信道

本节主要知识点

- ❑ 侧信道与隐秘信道有什么区别？
- ❑ 有哪些常见的侧信道/隐秘信道攻击方法？
- ❑ 有哪些常见的侧信道/隐秘信道防御方法？
- ❑ 为什么说很难完全消除侧信道攻击？

无论是访问控制机制，还是可信执行环境，都对隔离能力有着很强的依赖。在访问控制中，若一个进程能够直接访问另一个进程的内存空间，就可以绕过引用监视器而访问其原本没有权限访问的数据。同样，若一个恶意进程能够窃取可信执行环境中的数据，那么隐私性将不复存在。我们之前所介绍的技术，是在数据直接访问的主路上设置关卡进行检查，而本小节即将介绍的侧信道攻击或隐秘信道攻击，则试图从旁路实现数据的窃取和传递，从而绕开权限检查的一道道关卡。

侧信道攻击（Side-channel Attacks），又称为“边信道攻击”或“旁路攻击”，是一种利用时间、功耗、电磁泄漏、声音等非传统信道推测出系统中关键信息的攻击方式。**隐秘信道攻击**（Covert-channel Attacks），是指原本无法直接通信的两方，通过原本不被用于通信的机制，进行数据传输。

举一个隐秘信道的例子：用户希望运行一个处理自己消费记录的应用 A，统计自己每周花了多少钱。消费记录是非常私密的数据，用户担心这个应用会偷偷把数据上传到攻击者的网站，然后去非法兜售，于是取消了这个应用的网络、磁盘和进程间通信的权限——这是沙盒的一种基本形态——只允许应用最后输出一个结果到屏幕。不出所料，这个应用确实是恶意的，那么它如何才能把用户的隐私数据偷偷的上传呢？我们假设攻击者在用户的系统中还安装了另

[20] https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

一个应用 B，这个应用不访问任何用户的隐私数据，因此用户放心地给了它网络访问的权限。现在的问题变成了：在不能直接通信的前提下，应用 A 如何才能把数据发送给应用 B？这是一个可以开脑洞的问题：

- 如果应用有 A 播放声音的权限，应用 B 有录音的权限，则应用 A 可以把数据编码为音频，通过声音将数据发送给应用 B；
- 如果应用 A 有打开闪光灯的权限，应用 B 有访问摄像头的权限，则应用 A 可以把数据编码为光的闪烁长短与频率传递给应用 B；
- 如果应用 A 有访问震动接口的权限，应用 B 有访问运动传感器的权限，则应用 A 可以把数据编码为震动的频率传递给应用 B；
- 如果应用 B 有访问 CPU 温度的权限，应用 A 可以长时间运行计算密集的代码让 CPU 升温表示 1，降温表示 0，应用 B 根据 CPU 温度来获取数据——这种方法的数据传输率会很低；
- 应用 A 可以故意刷掉操作系统的磁盘数据在内存中的缓存（例如通过一定的磁盘访问模式），应用 B 通过访问磁盘的时间判断磁盘 hit 或 miss（分别表示 1 和 0）来得到应用 A 编码的信息；
- ...

侧信道攻击和隐秘信道攻击的手段有许多相似之处，其不同在于：侧信道攻击中，一方是攻击者，另一方是被攻击者，攻击者窃取被攻击者的数据；而隐秘信道中的两方，则是互相串通的，其目的就是为了将信息从一方传给另一方。在本节中，我们将主要以侧信道攻击为主，介绍常见的攻击与防御方法。

16.6.1 时间信道：Timing Channel

在所有侧信道攻击中，利用**时间信道**（Timing-channel）的攻击最为常见。攻击者利用系统完成不同任务的时间差异来推测任务执行过程中的细节，并根据已知的任务语意信息再推测出相应的关键信息。例如代码16.3所示：

```
checkpw (user, passwd):
  acct = accounts[user]
  for i in range(0, len(acct.pw)):
    if acct.pw[i] != passwd[i]:
      return False
  return True
```

代码片段 16.3: 检查密码的代码片段：攻击者可根据返回时间判断猜对了几位密码

这段代码的功能是用户登录过程中，检查输入的密码与之前保存的密码是否一致。从功能上看，这段代码并没有逻辑错误；然而如果仔细分析，会发现一个问题：代码执行的时间与输入的密码是否正确存在关联性。假设密码的长度是 8 个字符，如果输入的密码是正确的，那么函数将在循环 8 次后返回；如果密码的第一位就错了，那么函数将在循环第一次的时候就返回。于是，攻击者可以不断尝试输入不同的密码，并利用该函数的返回时间来判断密码的第一位是否猜对了——如果第一位猜对了，那么函数返回会稍稍慢一些，因为多运行了一次循环；然后再猜第二位，如果猜对了，函数返回又会慢一些。这样不断尝试，就能把所有的密码位都猜出来。如果密码是由大小写字母和数字组成的，那么每一位密码有 62 种可能，对于 8 位密码，理论上攻击者应当穷举 62^8 次才能遍历所有可能的密码；而采用了时间信道攻击后，则最多只需要尝试 $62 * 8$ 次就能把密码偷到。

有同学可能会问：循环 1 次和循环 2 次的时间差，可能只有几个 cycle，真的能够精确测量出来么？攻击者可以尝试通过缺页异常把这个时间差扩大。具体来说，如果攻击者把密码存在 2 个内存页上，其中，第一个字符在第一个内存页，第 2-8 个字符在第二个内存页，并且故意让第二个内存页处于换出状态（可以思考下如何做到），然后运行上述函数。如果函数迅速返回，说明第一位猜错了；反之，如果第一位猜对了，那么系统在访问第二位时会触发缺页异常，操作系统需要将第二个内存页从磁盘换入内存，这个过程的时间会远远长于不发生缺页异常的情况，因此攻击者可以更准确、更方便的做出判断。

上面这段代码来自 Tenex 操作系统。Tenex 是 1969 年 BBN 公司为了 PDP-10 系统开发的系统，也是早期使用虚拟内存的操作系统；1974 年，BBN 公司的专家 Alan Bell 发现了这段代码存在的漏洞。

16.6.2 缓存侧信道：Cache Side Channel

缓存侧信道是利用时间信息推测程序执行中缓存的行为，进而推测出程序中的关键信息。一个简单的例子如代码16.4所示：

```
1 if (i == 0)
2   func_a();
3 else
4   func_b();
```

代码片段 16.4: 缓存侧信道的代码片段：攻击者可根据哪个函数在缓存中，判断变量 i 的值

假设攻击的程序和被攻击的程序运行在同一个 CPU，攻击者知道`func_a`和`func_b`两个函数在内存中的地址，那么可以推算出这两个函数在 cache 中的位置；假设这两个函数在 cache 中处于不同的 set。在攻击准备阶段，攻击者将自己的函数`mal_func_a`和`mal_func_b`加载到 cache 中，且位置与`func_a`和`func_b`重合，并记录下执行这两个函数的时间。然后，上述函数执行。攻击者等到函数执行完后，再运行`mal_func_a`和`mal_func_b`，并记录时间：如果执行`mal_func_a`的时间变慢了，说明这个函数的代码从 cache 中被 evict 了，导致 cache miss，这说明`func_a`运行了，于是可以推测出变量 i 的值为 0。

我们来简单回顾下 cache 的架构。在现代处理器上，通常由三层 cache，分别是 L1、L2 和 L3。其中，L1 和 L2 是每个 CPU 核私有的，L3 则被同一个 CPU 上的所有核共享，又被称为末级缓存（Last Level Cache，LLC）。L3 是 inclusive 的，即所有在 L1 和 L2 中的数据均会在 L3 中存在。Cache 通常采用 LRU 算法作为替换策略。Cache 一般由硬件直接管理，对软件透明；CPU 也会提供一些与 cache 相关的指令，x86 提供了`clflush`指令（Cache Line Flush）用于刷掉 cache，ARM 处理器也提供了类似的指令，如`DC IVAC`指令。

为了实现基于 cache 的攻击，攻击者需要非常了解 cache 的架构以及 evict 策略。由于应用程序无法直接操控 cache，因此需要通过对内存的操作而间接的实现对 cache 的控制，以完成对 cache 内容的布局。具体来说，有四类方法来控制 cache：Flush+Reload，Flush+Flush，Prime+Probe 和 Evict+Reload。

Flush+Reload

Flush+reload 方法的思路是：假设攻击进程和目标进程共享一块内存，攻击者的目标是想知道目标进程是否访问了这块共享内存中的某个变量。具体步骤如下：

1. 第一步：攻击进程首先将 cache 清空，方法是不断访问其他内存，用其他内存的数据填满 cache，或直接通过 flush 指令将 cache 清空；

2. 第二步：等待目标进程执行;

3. 攻击进程访问共享内存中的某个变量，并记录访问的时间：若时间长，则表示 cache miss，意味着目标进程在第二步中没有访问过该变量；若时间短，则表示 cache hit，意味着目标进程在第二步中访问过该变量。

然后，攻击者可根据该变量是否被访问，结合程序的逻辑继续推断出更多的信息。

到这里可能会产生这样一个问题：攻击进程和目标进程怎么会正好有一块共享内存呢？这是否意味着两个进程必须主动地建立共享内存？而攻击进程又怎么会和目标进程主动协作呢？这里需要回顾操作系统中常见的内存去重机制（Deduplication）：如果两块内存的内容是一样的，那么操作系统就会将其中一块删去，将剩下的内存映射为只读，并通过写时复制的方法来保证任意进程都能够正常写入。因此，攻击者可以通过构造一块和目标进程完全相同的内存，并触发内存去重机制（通常固定一段时间就会触发一次）来实现内存页的共享。那么，下一个问题是：如何才能知道目标进程某一块内存的内容呢？方法有很多，最简单的一种方法是：攻击者先运行一遍目标进程，在某个时间点对内存做一次快照，然后将包含关键数据的内存页记录下来，用于后续的攻击。问题又来了：怎么保证攻击者运行得到的内存页与目标进程运行时的内存页数据完全一致呢？确实有可能不同，尤其是当内存页中数据是动态生成的时候，攻击者很难猜到数据具体的数值。因此，攻击者需要通过各种方法来猜可能的不同，并做出调整。这样的攻击并不一定能成功，即使成功也通常需要许多次的尝试。

Flush+Reload 方法具有以下优点：可以跨 CPU 核，甚至跨多个 CPU（因为 flush 操作会作用于所有的 CPU）；对 non-inclusive cache 也可以攻击；噪音相对低，因为可以只关注某一行 cacheline，而不需要关注整个 cache set。但也有一个缺点，就是攻击准备有些难，需要构造与目标进程完全相同的内存页。一个利用 Flush+Reload 方法的攻击，可参见 CVE-2014-3356。

Flush+Flush

Flush+Flush 方法是基于缓存刷新时间（如`clflush`）来推测数据在缓存中的状态，进而推测出之前程序的执行行为。若数据在缓存中，那么`clflush`的执行时间相比数据不在缓存中要更长。具体攻击步骤如下：

1. 第一步：攻击进程首先将 cache 清空（Flush）；
2. 第二步：等待目标进程执行；
3. 第三步：运行`clflush`再次清空不同的缓存区域，若时间较短说明缓存中无数据，时间较长则说明缓存中有数据，意味着目标进程曾经访问过对应的内存。

Flush+flush 方法具有以下优点：攻击程序在整个攻击过程中只需要清空缓存，而不需要进行实际的访存，因此具有一定的隐蔽性。对攻击者来说的缺点包括：`clflush`对于有数据和无数据的缓存操作时间差异不是非常明显，因此该攻击的精度并不高。

Prime+Probe

有些 CPU 可能没有针对某条 cacheline 的 flush 指令，或者攻击进程无法建立与目标进程的共享内存。因此攻击者需要将数据从 cache 中 evict 出去。由于 cache set 的存在，为了 evict 某一个数据，需要将该数据在内存中对应的整个 cache set 都 evict，这就是 Prime+Probe 的攻击思路。Prime+Probe 攻击的具体步骤如下：

1. 第一步：攻击进程用自己的数据将 cache set 填满（Prime）；
2. 第二步：等待目标进程执行；
3. 第三步：再次访问自己的数据，若时间很短，表示 cache hit，说明目标进程没有将该数据 evict，推导出目标进程没有访问过某个关键数据，反之则说明目标进程访问了某个关键数据。

Prime+Probe 方法的优点包括：不需要共享内存；支持动态和静态分配的内存。对攻击者来说的缺点包括：噪音更多；需要考虑 LLC 的实现细节，如组相连等；Cache 必须是 inclusive；无法很好地支持多 CPU 环境；需要首先定位目标进程使用的 cache set。

Evict+Reload

与 Prime+Probe 方法不同的是，Evict+Reload 先执行目标进程，之后将 cache set 中的数据清出去，之后再次运行目标程序。比较两次程序执行的时间，从而得出关键数据是否被访问到。

具体步骤如下：

1. 第一步：等待目标进程执行，并测量执行时间；
2. 第二步：将关键数据所在的 cache set 都替换成攻击进程的数据；
3. 第三步：再次执行目标进程，并测量执行时间。若时间较短，表示 cache hit，说明攻击进程 evict 的数据没有被目标进程访问，推导出目标进程没有访问过某个关键数据，反之则说明目标进程访问了某个关键数据。

Evict+Reload 与 Prime+Probe 的优点类似，主要是无需依赖 flush 指令。对攻击者来说的缺点包括：无法支持动态分配的内存；需要非常详细的了解 LLC 的 eviction 策略，很多都需要逆向才能得知；Cache 必须是 inclusive；无法很好地支持多 CPU 环境。

总结以上四种攻击：基于缓存的侧信道攻击。除了对 cache 的了解和操控，攻击者还需要知道程序的逻辑，从而发现数据和程序执行的对应关系；并且需要了解程序在内存中的布局，从而可以从 cache 的状态推测出程序的执行；最后，攻击者还必须让自己的攻击代码和被攻击的进程运行在同一个 CPU Socket 上。

16.6.3 侧信道攻击的防御方法

侧信道攻击很难被完全防御住。由上面的介绍，我们可以归纳出侧信道攻击的一个共性：共享。换句话说，攻击者和被攻击者都使用或访问了同样的硬件或资源。当被攻击者在做了某个操作后，对系统整体产生了影响，而这个影响能够被使用同样系统的攻击者发现，那么就构成了一个最简单的侧信道：发现该影响存在和发现该影响不存在（即做了操作和没做操作）可以被编码为 0 和 1。

因此，防御侧信道的方法，在于保证任何一个操作在执行后，不会对整个系统产生其他应用可见的影响。如果希望完全保证这一点，最理想的方式是将攻击者和被攻击者运行在完全隔离的物理主机，使其没有任何共享，包括计算硬件、网络、空间（光、声音甚至温度）。但很多时候彻底的隔离既不现实也无必要，更实际的做法可以对某些常见的侧信道进行有针对性的防御，例如时

间侧信道和缓存侧信道。首先，为了消除时间侧信道，可解除计算过程与运行时间之间的映射关系，即无论如何运行，所消耗的时间是固定的，这种方法称为“常量时间算法”；其次，为了消除缓存侧信道，可解除计算过程与缓存状态之间的映射关系，即无论如何运行，最后缓存的状态是一样的，这种方法称为“不经意随机访问内存”（ORAM，Oblivious RAM）。如此，若攻击者仅通过观察时间或缓存状态，便无法获得任何信息。最后，也可以通过内存分配策略，故意为攻击进程和目标进程分配不同的内存，从而防止攻击进程污染目标进程在 cache 中的数据，这种方法称为“缓存着色技术”。

常量时间（Constant Time）算法

常量时间（Constant-time）算法，顾名思义就是算法的运行时间与输入无关，恒定为一个常量，因此攻击者无法根据算法运行时间获得除该时间之外的任何信息。一种常见的实现方式是利用体系结构的cmov指令（Conditional MOV），该指令会根据条件将两个值中的一个复制到目标寄存器中。例如：

```
// 传统实现方式
if (secret == 0)
  x = a + b;
else
  x = a / b;

// 常量时间实现方式
v1 = a + b;
v2 = a / b;
cond = (secret == 0);
x = cmov(cond, v1, v2);
```

代码片段 16.5: 传统实现方式与常量时间实现方式

假设secret的值为 0 或 1，攻击者能够获取这段代码执行的时间。代码上半部分的传统实现方式根据secret的值执行了不同的运算，其中除法会比加法慢，因此攻击者可通过时间来判断secret的值为 0 或 1。代码下半部分的常量实现则先把除法和加法都做了，分别放在两个临时变量中，然后通过 cmov 将其中一个值赋给 x。代码的执行时间与secret的值无关，但缺点是计算变得更慢了——毕竟原本只要做一个运算，现在需要做两个。

不经意随机访问内存（ORAM）

ORAM 的目的是将访存行为与程序执行过程解耦，使攻击者即使能够观察到所有的访存请求，也无法反推出与程序执行相关的信息。一种最简单的隐藏访存模式的方法，是定时、定量、定位的访问方式，即无论实际是否有访存需求，均以固定的周期，访问固定的位置，每次访问固定大小的数据。例如，CPU 顺序循环访问所有的有效内存区域（预先设定分配），程序按需获得真正想要访问的数据，若还没访问到则等待，若已经访问过了则等待下次循环。在实际系统中，ORAM 会引入很大的额外负载：一方面，会产生大量的无效内存访问，导致有效访存的吞吐率下降；另一方面，访存需要等待一定的时刻，导致时延大幅度增加。如何提高 ORAM 的效率依然是一个开放的问题。

缓存着色技术（Page Coloring）

除了 L1 Cache，其他层的 cache（例如 LLC）以物理地址计算 cache index，并决定与 cache set 的对应关系。在这些 cache 里，因为相邻的物理地址对应的 cache index 不同，它们会分配到不同的 cache set 中，因此减少了访问连续物理内存时的 cache miss。缓存着色技术利用了这样的思想，可以进一步减少进程内存在 cache 中的冲突，并提升程序运行的性能。同时，该技术也能抵御利用“Cache Set Eviction”的 cache 侧信道攻击。具体来说，该技术将分配到同一个 cache set 的物理内存地址都标记成同一种颜色，分配到不同 cache set 的物理内存地址则拥有不同的颜色。在物理内存分配器中，故意为不同进程分配不同颜色的物理内存区域，因此它们的内存将不会在 cache 中产生冲突，从而防御了 Prime+Probe 和 Evict+Time 的攻击。

16.7 案例分析：Meltdown 与 KPTI

本节主要知识点

- *Meltdown/Spectre* 与传统的隐秘信道的区别是什么？
- 这两种攻击的实际危害有多大？
- 通过这个攻击，再次理解性能与安全性的权衡。

Meltdown（熔断）和 Spectre（幽灵）是在 2017 年被发现，在 2018 年被公开的两个攻击。这两个攻击开创了一类新的隐秘信道攻击方式：利用 CPU

的预测执行机制。在当时，几乎所有的主流 CPU 都受到了影响，许多软件厂商不得不紧急打补丁并做出对应的防御措施。

16.7.1 CPU 预测执行机制

现代 CPU 为了获得更高的性能，普遍实现了**预测执行**（Speculative Execution）的技术。例如代码片段 16.6，当x的值为 0 时，对y的赋值并不会发生。然而，若 CPU 在访问x时发生了阻塞（如 cache miss），CPU 并不会等在`if`语句上，而是会先假设该条件成立，并实际去执行对y的赋值。如果最后`if`的条件满足，则y的赋值已经完成，使性能得以提高；若`if`条件不满足，CPU 会抛弃对y的赋值，等价于没有执行过。

```
1 if (x != 0)
2   y = a[10];
```

代码片段 16.6: CPU 预测执行技术

然而，CPU 的设计者忽略了一个问题：数组`a[]`在预测执行的过程中被访问了一次——这有什么问题呢？从软件的角度，读内存对系统状态不产生任何影响；但从 CPU 体系结构的角度，读内存会对 cache 产生影响，换句话说，`a[10]`的值会被加载到 cache 中，进而形成一个隐秘信道。这就是 Meltdown 和 Spectre 攻击所利用的基本原理。

16.7.2 Meltdown 攻击

在预测执行机制的基础上，攻击者进一步发现，若让 CPU 预测执行一条跨权限非法内存访问的指令，那么 CPU 会真的执行，且不会进行权限检查，只要最后`if`条件不满足，CPU 不会报错。因此，攻击者可以构造一段用户态的指令，先去访问位于内核的数据x，然后根据x的值（假设仅为 0 或 1）去访问用户态的内存地址 A 或 B，导致 cache 发生对应的变化。这段代码被 CPU 预测执行，而预测失败所以 CPU 放弃了最后的结果，此时攻击者就可以根据 cache 的变化判断出x的值究竟是 0 还是 1——换句话说，攻击者可以在用户态访问任意的内核数据！

```
1    ; rcx = kernel address, rbx = probe array
2    xor rax, rax
3  retry:
4    mov al, byte [rcx]
5    shl rax, 0xc
6    jz retry
7    mov rbx, qword [rbx + rax]
```

代码片段 16.7: Meltdown 核心代码

代码片段 16.7来自于 Meltdown 的论文 [6]。其中，`rcx`寄存器中是内核数据的地址（即需要窃取的数据），`rbx`寄存器指向一个用户态可以访问的数组，攻击者会通过访问这个数组来改变 cache。首先将`rax`寄存器清零，然后读取内核地址至`al`（`rax`中低位的 8-bit 区域），然后将其左移 12 位，如果不为 0 则以左移后的值作为数组下标访问`rbx`所指向的数组。然后攻击者通过预测执行的方式运行这段代码，并在运行结束后根据 cache 的改变，判断到底访问了数组的哪个字段，即数组下标的值，进而窃取到内核数据的具体值。

Meltdown 是一种隐秘信道攻击，但与传统攻击不同的是，该攻击并不存在一个目标进程，而只需要一个攻击者进程。换句话说，并不需要被攻击的对象（内核）做任何事情，这大大降低了攻击的难度，提高了攻击的速度——把整个内核的内存都 dump 出来，最高可以达到 503KB/秒 [6]。Meltdown 攻击几乎意味着内核与应用的隔离被完全打破了，这对系统安全界造成了一个巨大的冲击。

16.7.3 KPTI：内核页表隔离

如何防御 Meltdown 攻击呢？从根本上说，这是一个 CPU 的漏洞：即使是预测执行，CPU 也不应当允许用户态的代码访问内核的数据；但 CPU 为了更好的性能，将权限检查放到了访存之后。因此，硬件厂商给出了相应的修复方案。

那么，在不改变硬件的前提下，如何通过软件的方式防御该漏洞呢？Linux 给出了 KPTI（Kernel Page Table Isolation）的方法。传统的 Linux 内核与应用共享一个地址空间，也共享一个页表；内核一般位于在地址空间的顶部，应用一般位于底部。因此，Meltdown 攻击可以直接在用户态读取位于内核区域的数据。KPTI 则提出了让内核单独拥有一个页表的设计，将内核区域的地址映射从应用的页表中完全删去，于是用户态的代码即使访问内核区域的数据，也会由于没有对应的地址映射而失败。

相比之前的设计，KPTI 需要在内核与应用切换的时候增加一个操作：切换页表。因此 KPTI 不可避免的会引入较高的额外负载，尤其是对于频繁进行系统调用的程序来说。但为了安全，许多 Linux 的发行版都会默认开启 KPTI。

16.8 思考题

1. Hello World 在运行的时候，需要哪些文件的什么权限?
2. 普通用户可以通过运行`/usr/bin/passwd`这个文件修改自己的密码，用户密码的 hash 值保存在`/etc/shadow`这个文件中，但是这个文件的拥有者是 root 用户，且权限为 000。那么一个普通用户是如何通过`passwd`来修改`/etc/shadow`这个文件的呢?
3. 假设某个系统使用 NFS 管理文件，客户端在打开文件 F 时（T1 时刻），具有可写权限；然后文件 F 在服务器端被设置为只读（T2 时刻)，那么当客户端写入文件 F 时（T3 时刻），是否会报错?
4. 从安全的角度来看，为什么对文件的操作不使用 inode 号作为参数，而要用 fd 作为参数?
5. 在同一台手机上的两个应用，还能通过哪些隐秘信道的方式传输数据? 如果这两个应用分别安装在两台手机呢?
6. 如果允许且只允许你修改一台计算机内存中的任意一个 bit，你该如何获取到最高权限?

参考文献

[1]

[2] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal De Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–189, 2015.

[3] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[4] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, 2014.

[5] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. kguard: lightweight kernel protection against return-to-user attacks. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 459–474, 2012.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[7] Jerome H Saltzer and M Frans Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

操作系统安全：扫码反馈