2023-2024学年春季学期

计算机体系结构安全
Computer Architecture Security

授课团队：史岗，陈李维

计算机体系结构安全

Computer Architecture Security

[第4次课] 操作系统对安全的影响

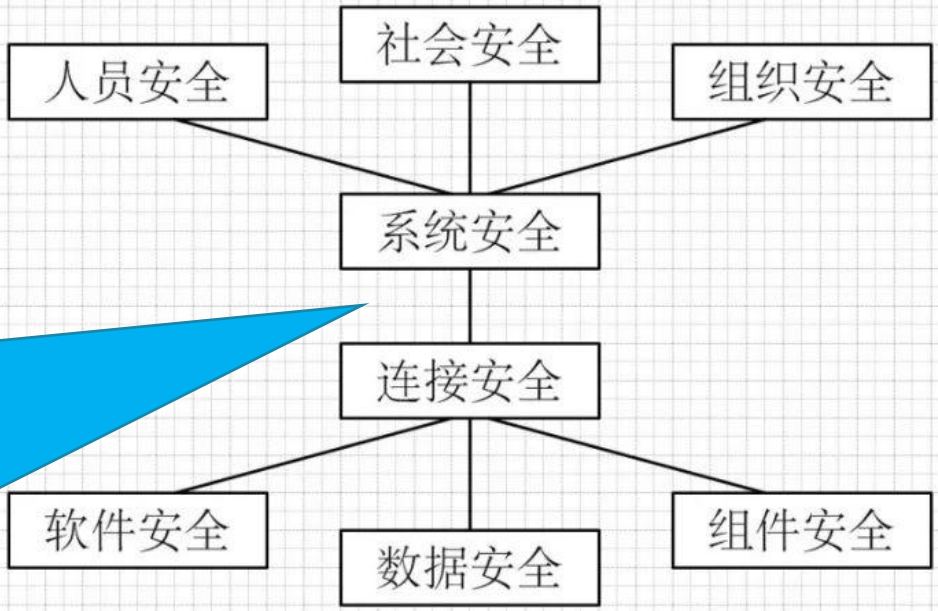
授课教师：史岗

授课时间：2024.3.18

网络空间安全学科知识体系 (CSEC2017)

- ACM SIGCSE 2018国际会议上正式发布
- 迄今为止国际上最具广泛代表性和权威性的网络空间安全学科知识体系

着眼于由组件通过连接而构成的系统的安全问题，强调不能仅从组件集合的视角看问题，还必须从系统整体的视角看问题，关键知识包括整体方法论、安全策略、身份认证、访问控制、系统监测、系统恢复、系统测试、文档支持。



内容概要

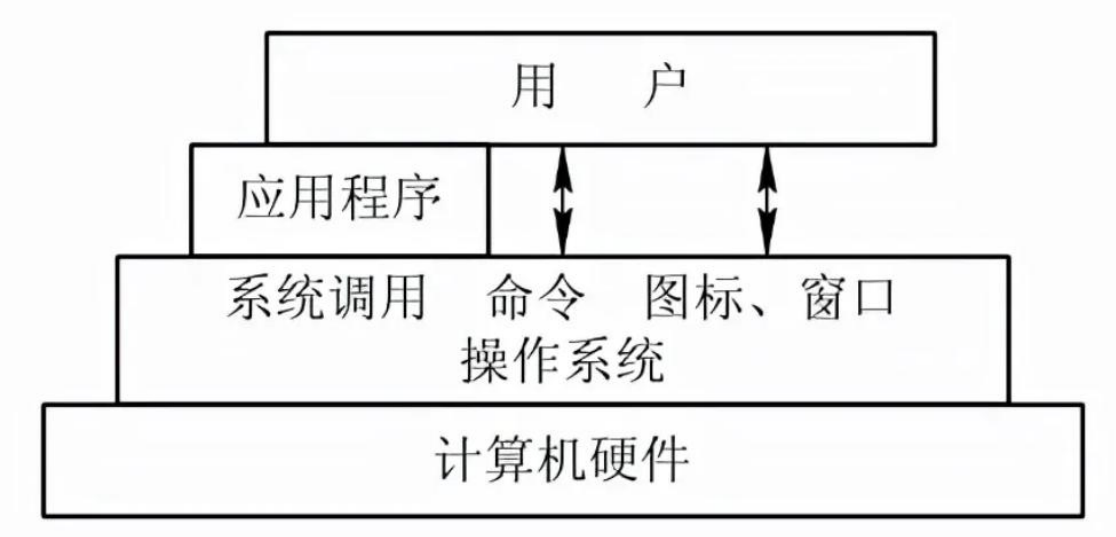
- 操作系统的背景知识
- 操作系统自身脆弱性
- 操作系统对安全支持
- 总结

内容概要

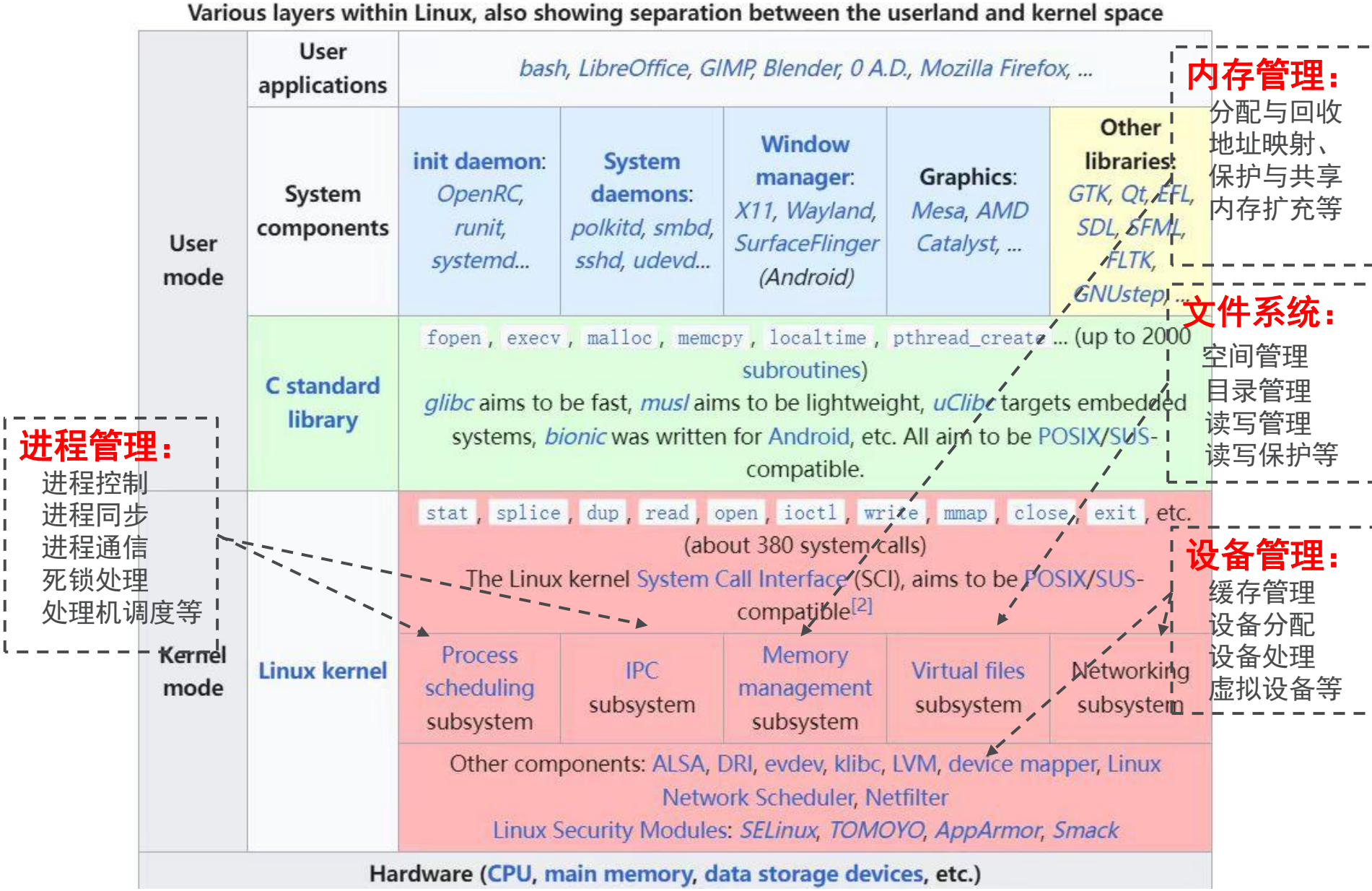
- 操作系统的背景知识
- 操作系统自身脆弱性
- 操作系统对安全支持
- 总结

- 操作系统(Operating System)：控制和管理整个计算机系统的硬件和软件资源，并提供用户和其他软件开发接口和运行环境，是计算机系统中最重要的系统软件。

- 操作系统的主要目的是使应用程序能够有效地使用计算机硬件资源，而不需要知道具体的硬件细节。



linux操作系统组成



内容概要

- 操作系统的背景知识
- **操作系统自身脆弱性**
- 操作系统对安全支持
- 总结

内容概要

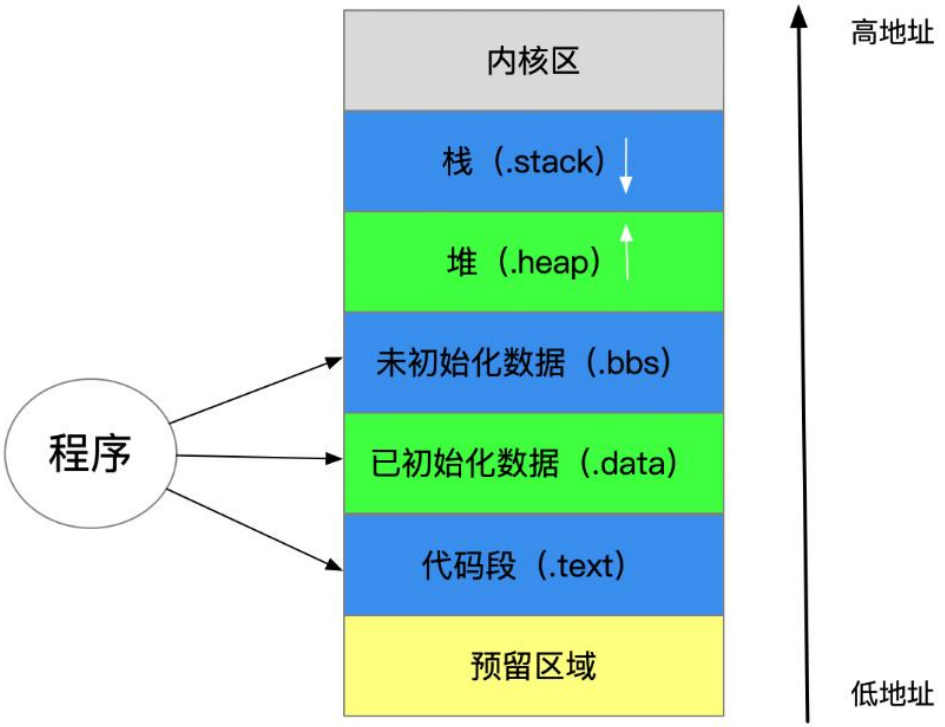
- 操作系统的背景知识
- 操作系统自身脆弱性
  - 存储管理
  - 隐蔽信道
  - 设备管理
- 操作系统对安全支持
- 总结

内容概要

- 操作系统的背景知识
- 操作系统自身脆弱性
  - 存储管理
  - 隐蔽信道
  - 设备管理
- 操作系统对安全支持
- 总结

操作系统自身脆弱性

- 存储（内存）管理
  - 操作系统的内存管理主要负责内存的分配与回收。
  - 最主要的目的就是提高内存利用率，所谓的提高内存利用率，就是尽可能的让物理内存都用起来；按进程调度的规则，运行尽可能多的进程。
- 内存结构



内存管理自身脆弱性

2023 年被利用最多的 10 大 CWE 漏洞排名

排名	CWE	分类	得分	KEV 数量	平均 CVSS
1	CWE-416:释放后使用	CWE-1399:内存安全	73.99	44	8.54
2	CWE-122:堆缓冲区溢出	CWE-1399:内存安全	56.56	32	8.79
3	CWE-787:跨界内存写	CWE-1399:内存安全	51.96	34	8.19
4	CWE-20:输入验证不恰当	CWE-1406:不正确的输入验证	51.38	33	8.27
5	CWE-78:OS命令中使用的特殊元素转义处理不恰当(OS命令注入)	CWE-1409:注入问题	49.44	25	9.36
6	CWE-502:不可信数据的反序列化	CWE-1415:资源控制	29.00	16	9.06
7	CWE-918:服务端请求伪造(SSRF)	CWE-1396:访问控制	27.33	16	8.72
8	CWE-843:使用不兼容类型访问资源(类型混淆)	CWE-1416:资源生命周期管理	26.24	16	8.61
9	CWE-22:对路径名的限制不恰当(路径遍历)	CWE-1404:文件处理	19.90	14	8.09
10	CWE-306:关键功能的认证机制缺失	CWE-1396:访问控制	12.98	8	8.86

- 释放后使用
  - 内核中**内存管理回收机制**：应用程序释放内存时，如果内存块大小没有达到一个值，并不马上会将内存块释放回内存，而是将内存块标记为空闲状态。当达到一定值时，才将空闲内存释放。
  - "use after free"漏洞是指在释放了内存后，再次使用这块内存的情况。这种情况可能会导致程序崩溃，或者被恶意利用来执行任意代码。
  - 一个简单示例：
    - 此代码申请了ptr指针并分配内容，然后释放，但没有将此内存回收，此指针仍有内容，再次使用这个指针仍可以读取其内存的数据。

```
01. #include <stdlib.h>
02. #include <string.h>
03. #include <stdio.h>
04.
05. int main() {
06.     char *ptr = malloc(100);
07.     strcpy(ptr, "Hello, world!");
08.
09.     free(ptr);
10.
11.     // Use after free!
12.     printf("%s\n", ptr);
13.
14.     return 0;
15. }
```

- 漏洞利用
  - 此漏洞利用程序定义了一个结构体，里面包含一个函数指针。

```
01. typedef struct {
02.     void (*callback)();
03. } StructWithCallback;
04.
05. void goodFunction() {
06.     printf("This is the good function.\n");
07. }
08.
09. void badFunction() {
10.     printf("This is the bad function. Arbitr
11. }
12.
13. int main() {
14.     StructWithCallback *ptr = malloc(sizeof(StructWithCallback));
15.     ptr->callback = goodFunction;
16.
17.     free(ptr);
18.
19.     /
20.     s
21.     e
22.
23.     /
24.     p
25.
26.     return 0;
27. }
```

```
cwh@cwh-virtual-machine:~/Desktop$ gcc uaf.c -o uaf
cwh@cwh-virtual-machine:~/Desktop$ ./uaf
This is the bad function. Arbitrary code execution!
```

分配一块内存，ptr指向此内存，此时ptr结构体中的函数指针指向good函数。释放ptr之

小

向

区

的

函数指针的值就改变了ptr指针指向的值。

- 堆缓冲区溢出
  - 堆溢出是指程序向某个堆块中写入的字节数超过了堆块本身可使用的字节数，因而导致了数据溢出，并覆盖到相邻地址的下一个堆块。
  - 堆溢出漏洞轻则可以使得程序崩溃，重则可以使得攻击者控制程序执行流程。
  - 堆溢出是一种特定的缓冲区溢出（还有栈溢出， bss 段溢出等）。但是其与栈溢出所不同的是，堆上并不存在返回地址等可以让攻击者直接控制执行流程的数据，因此我们一般无法直接通过堆溢出来控制 EIP 。

堆缓冲区溢出

```
01. #include <stdlib.h>
02. #include <stdio.h>
03. #include <unistd.h>
04. struct AAA
05. {
06.  char buf[0x20];
07.  void (*func)(char *);
08. };
09. void out(char *buf)
10. {
11.  puts(buf);
12. }
13. void vuln()
14. {
15.  struct AAA *a = malloc(sizeof(struct AAA));
16.  a->func = out;
17.  read(0, a->buf, 0x30);
18.  a->func(a->buf);
19. }
20. void main()
21. {
22.  vuln();
23. }
```

在此代码中可以看到明显的堆溢出，结构体AAA中buf的大小为32字节，却读入了48字节的字符，过长的字符直接覆盖了结构体中的函数指针，进而调用该函数指针时实现了对程序控制流的劫持。

- 跨界内存写
  - out-of-bound write，在预期大小缓冲区的末尾之后或开始之前写入数据，可能导致数据损坏、崩溃或代码执行。
  - 一个最简单的例子：

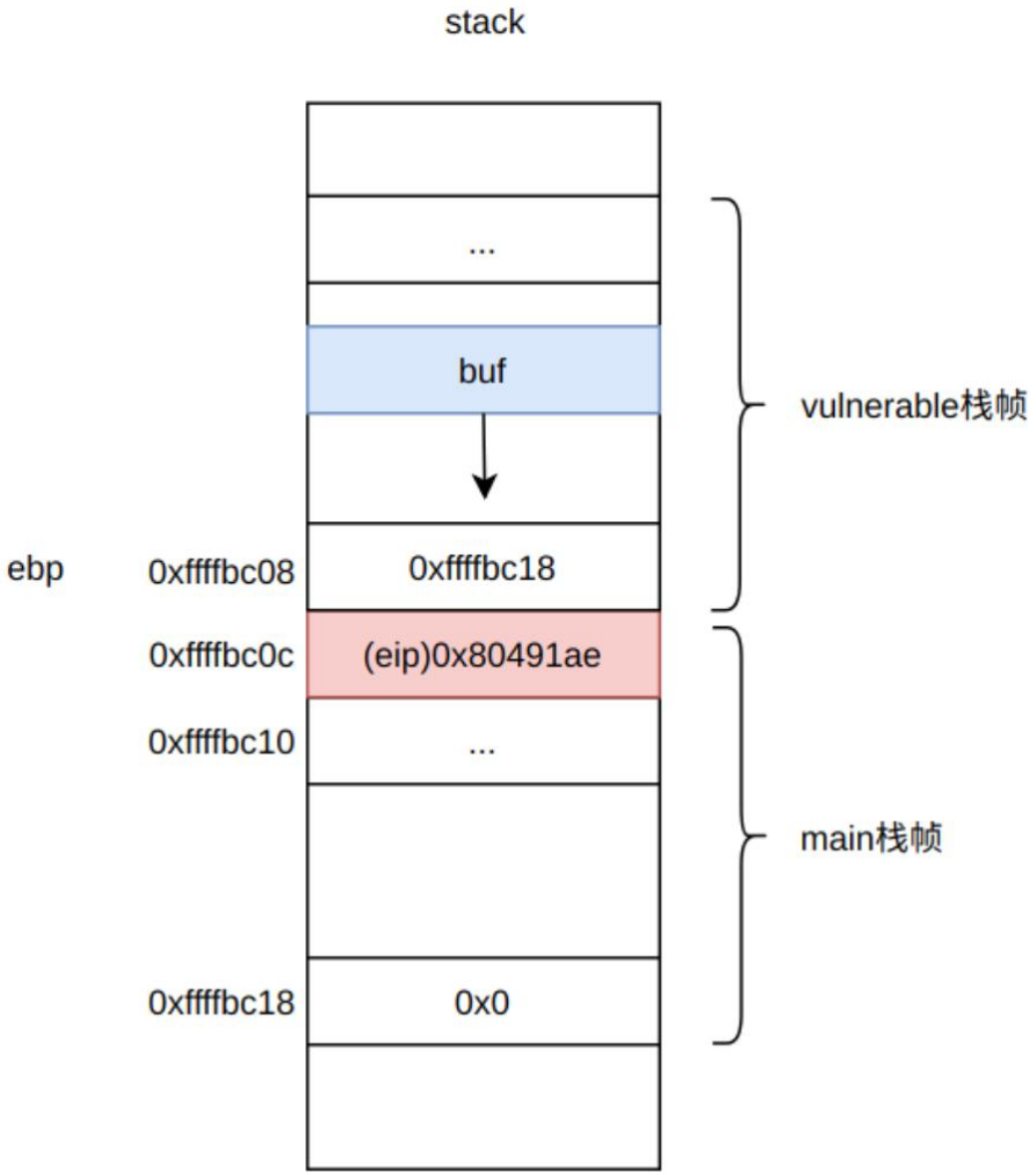
```
01. int id_sequence[3];
02.
03. /* Populate the id array. */
04.
05. id_sequence[0] = 123;
06. id_sequence[1] = 234;
07. id_sequence[2] = 345;
08. id_sequence[3] = 456;
```

- 跨界内存写
  - 栈溢出是跨界内存写的一种情况。
  - 当程序向栈上的缓冲区写入数据时，如果写入的数据超过了缓冲区的大小，就会发生栈缓冲区溢出。
  - 这种溢出会覆盖栈上的其他数据，包括其他变量的值，返回地址，以及可能的函数指针等。
  - 可能导致任意代码执行或改变程序的执行流。

跨界内存写

栈溢出改变程序流

在main函数调用vulnerable函数时，会在栈中压入main函数此时的程序执行的地址eip。vulnerable函数中定义了一个buf缓冲区，此缓冲区为临时变量放在栈中。若缓冲区发生溢出，因为他们在物理上是相邻的，当发生溢出时，可能覆盖eip（也就是当vulnerable函数执行完成之后，下一条要执行的指令的地址），导致程序控制流的改变。



内容概要

- 操作系统的背景知识
- 操作系统自身脆弱性
  - 存储管理
  - 隐蔽信道
  - 设备管理
- 操作系统对安全支持
- 总结

- 隐蔽信道
  - 允许进程以违反系统安全策略的方式传递信息的信道，这里的安全策略指强制安全策略

- 操作系统隐蔽信道种类
  - **隐蔽存储通道：一个进程直接或者间接写入一个存储位置，而另一个进程直接或间接读这个存储位置。一般来说，隐蔽存储通道涉及到不同安全级主体可以共享的某种数量有限的资源（比如硬盘）。判定标准为**
    - 发送方和接收方进程能访问共享资源的同一属性。
    - 发送方进程能改变该共享资源的状态。
    - 接收方进程能察觉该共享资源的状态变化。
    - 发送方与接收方进程之间有同步机制。
  - **隐蔽定时通道：信道的发送信号方式是一个进程调节自己对系统资源（比如CPU）的使用，从而影响另外一个进程观察到的真实系统响应时间。实例：CPU调度信道。判定标准为：**
    - 发送方和接收方进程有权存取共享资源的同一属性。
    - 发送方和接收方进程必须有权存取一个时间参照，比如实时时钟。
    - 发送方进程能够调整接收方进程侦查共享属性的变化所用的响应时间。
    - 存在某种机制，使发送方和接收方进程能启动隐蔽通信并正确给事件排序

隐蔽通道举例

- **最经典的一个隐通道是磁盘移臂隐通道，这个隐通道是1977年在KVM/370系统（虚拟机监视器）中发现的。**

假定系统中有H和L两个进程，H的安全级高于L的安全级。H中隐藏有特洛依木马试图泄露信息给窥探者L，但是按照强制访问控制的策略，L无法访问到H的信息，因此H不敢直接传送信息给L，并且也担心审计的监督，于是H和L寻求利用系统的安全漏洞，采用表面上合法的手段泄露信息。

L开辟一个属于自己的文件目录，H合法地具有了读这个目录的权力，假定磁盘上从51磁道到第59磁道属于这个目录，H和L按以下约定进行操作：

1、L请求读磁道55(此请求完成后，释放CPU)

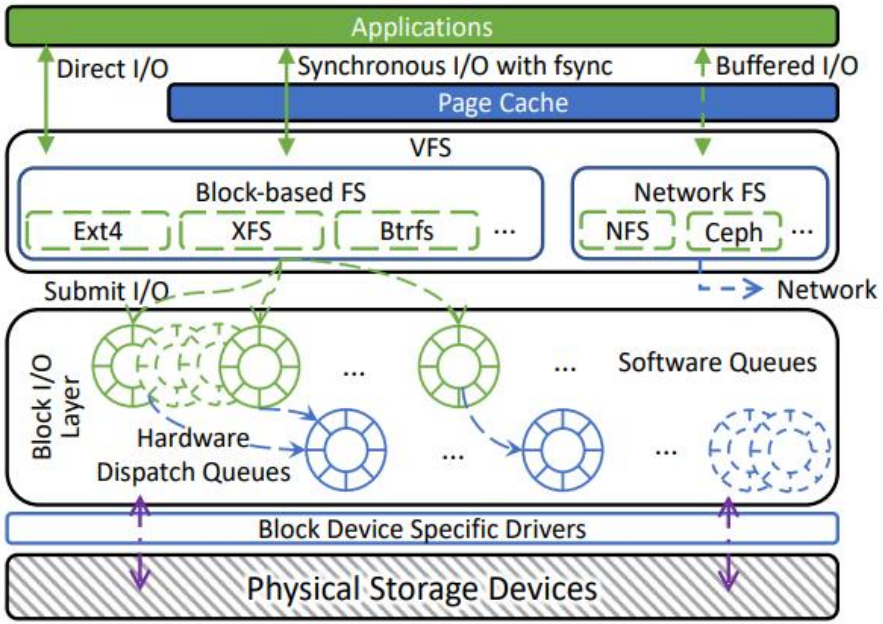
2、H请求读磁道53(送“0”)或读磁道57(送“1”)(请求完成后，立即释放CPU)

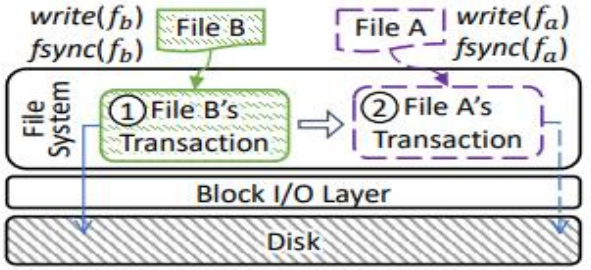
3、L同时请求读磁道51和磁道59，并观察这两个请求完成的先后次序，以确定H发出的信息，若访问磁道51先完成，则L确认收到“0”，反之，L确认收到“1”

重复以上操作，就可以在H和L之间传送连续的比特流。（当然还要采取一定的同步措施）

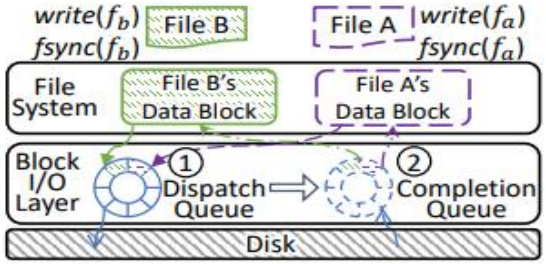
隐蔽通道举例

- “Sync+Sync: A Covert Channel Built on fsync with Storage”
  - **fsync系统调用：以文件描述符为输入参数，确保一直到写磁盘操作结束才返回。**
  - **预备知识：如果进程同时调用fsync，尽管它们不共享任何数据，但程序的响应时间会显著延长。时间延长的原因是由于共享软件结构（例如Ext4的日志）和硬件资源（例如磁盘的I/O调度队列），并发的fsync调用在存储层次的多个级别上发生争用。**





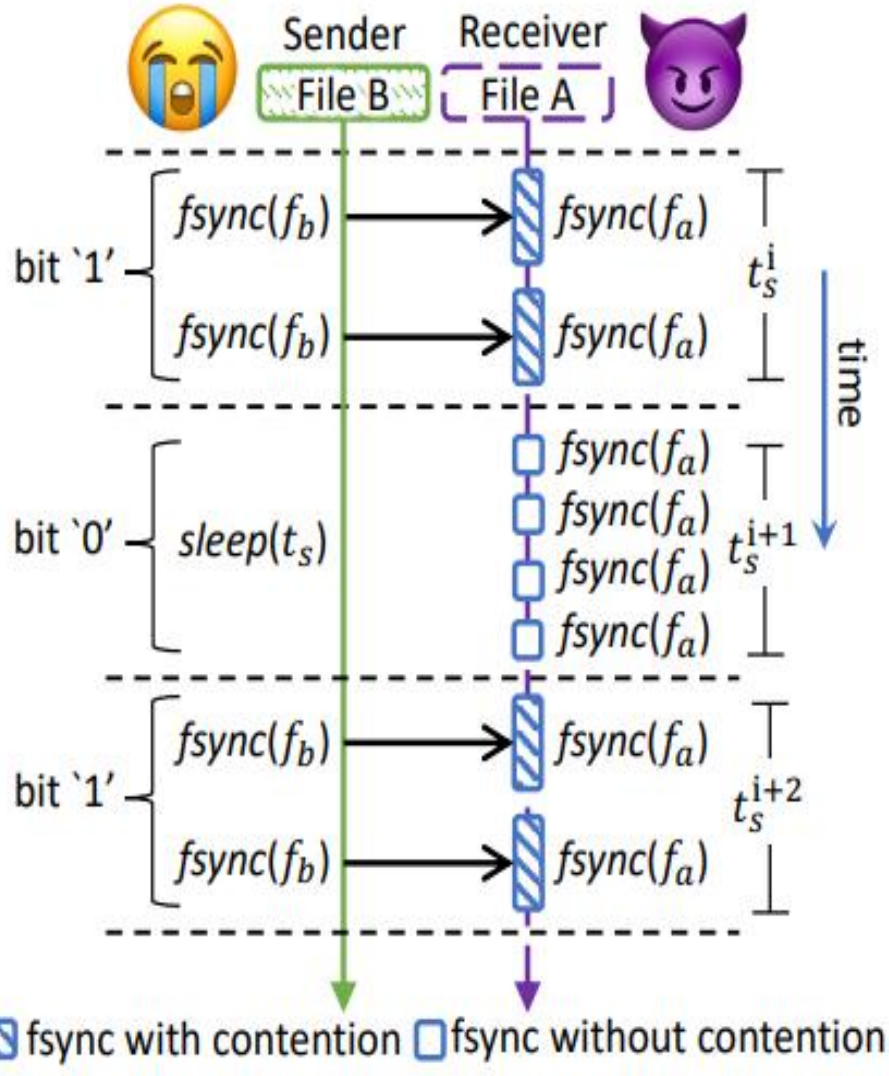
(c) Contention in File System.



(d) Contention in Block I/O Layer.

隐蔽通道举例

- Sync+Sync隐蔽信道
  - **发送方：通过在文件上调用fsync来传递位（bit）或不传递。**
  - **接受方：通过持续地fsync同步另一个文件，并测量fsync的延迟来决定接收到的位的值。**
  - **攻击效果：**
    - 错误率：0.4%
    - 传输带宽：20000b/s



内容概要

- 操作系统的背景知识
- 操作系统自身脆弱性
  - 存储管理
  - 隐蔽信道
  - 设备管理
- 操作系统对安全支持
- 总结

- 设备管理
  - 指计算机系统中除了CPU和内存以外的所有输入、输出设备的管理。
  - 除了进行实际I/O操作的设备外，也包括：设备控制器、DMA控制器、中断控制器、通道。

- 操作系统设备管理目的
  - 向用户提供使用外部设备的方便、统一的接口，按照用户的要求和设备的类型，控制设备工作，完成用户的输入输出请求。（设备的独立性/无关性）
    - 方便：用户能独立于具体设备的复杂物理特性而方便地使用设备
    - 统一：对不同设备尽量能统一操作方式

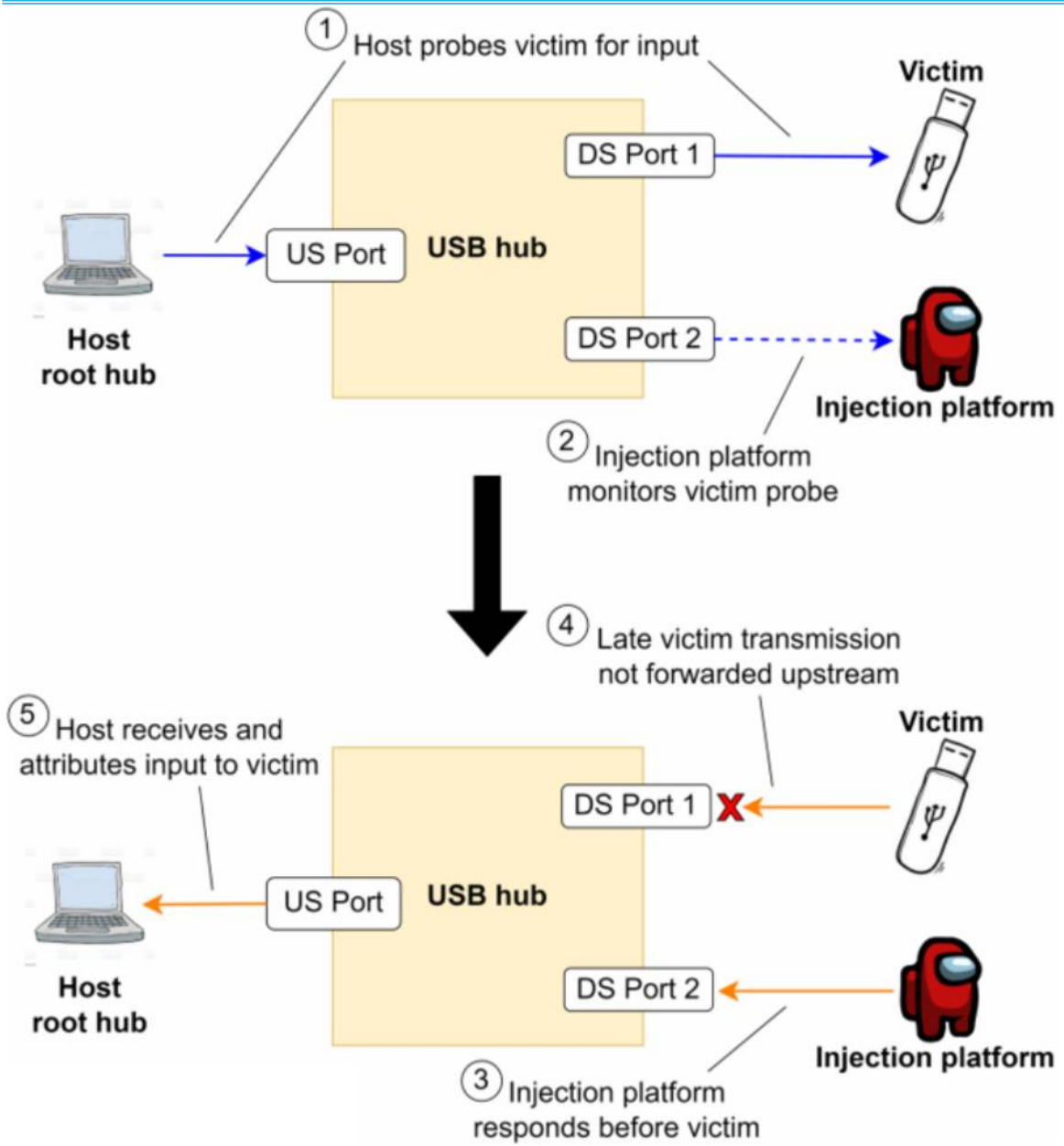
- 操作系统设备管理目的
  - 充分利用中断技术、多通道技术和缓冲技术，提高CPU与设备、设备与设备间的并行工作能力，充分利用设备资源，提高外部设备的使用效率。
  - 设备管理就是要保证在多道程序环境下，当多个进程竞争使用设备时，按照一定的策略分配和管理设备，以使系统能有条不紊地工作。

- I/O控制方式
  - 轮询方式：CPU 需要等待设备就绪，且参与数据传送。
  - 中断方式：CPU 无需等待设备就绪，但响应中断后参与数据传送。
  - DMA方式：CPU 在数据传送开始和结束时参与，与主存进行数据交换时不参与。

- 设备管理
  - 外设多样性，管理复杂，被攻击者利用的概率更高
    - 恶意USB设备：攻击者可以创建一个看起来像普通USB存储设备的恶意USB设备，但是当它被插入计算机时，它会自动运行恶意软件。这种攻击有时被称为"BadUSB"。
    - DMA攻击：一些高速外设，例如Thunderbolt设备，可以直接访问计算机的内存（DMA）。如果攻击者可以控制这样的设备，他们就可以读取或修改内存中的数据，绕过操作系统的保护。
    - 蓝牙设备攻击：KNOB(Key Negotiation of Bluetooth) Attack，该漏洞的编号为CVE-2019-9506，由于是蓝牙核心协议中的设计漏洞，因此影响了大量的蓝牙设备，比如Broadcom、CYW、Apple、Snapdragon等蓝牙芯片。

- 对USB通信的路径外注入攻击（Off-Path Injection Attack）
  - 一个恶意设备，可以放置在目标设备和主机之间的通信路径之外。该恶意设备能向通信路径注入数据，从而伪造数据的输入来源，欺骗主机系统。
- 威胁模型
  - 在注入攻击的威胁模型中，存在以下情景：至少有两个USB设备通过一个USB集线器连接到一个公共主机。其中一个设备是恶意攻击平台，由攻击者控制。另一个设备是目标设备，攻击者希望冒充的设备，不受攻击者的影响。
  - 根据USB系统的树形拓扑结构，攻击平台和目标设备虽然是独立的，但在物理上共享与主机的通信路径。系统可能连接了其他USB设备，这些设备被视为旁观者，不受攻击者影响，并在不受干扰的USB通信下执行其功能。
  - 其他系统组件都是可信的、且设备提供了授权策略。

自身脆弱性及分析



1、主机首先广播一个探针，请求目标设备的输入。

2、攻击平台观察主机的探针。

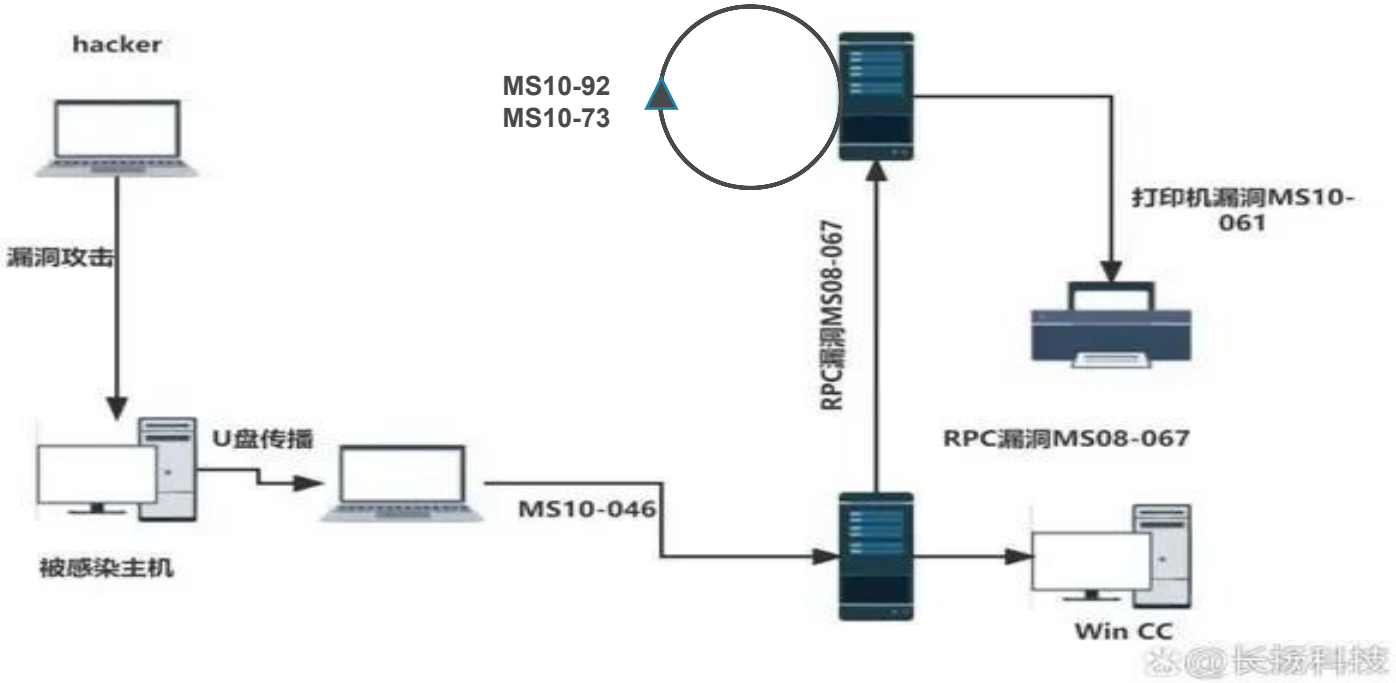
3、使用与预期目标设备响应格式匹配的上游数据传输进行响应。

4、如果攻击平台能够在目标设备之前做出响应，集线器会接受注入的传输并将其向上游转发，同时忽略目标设备的真实响应。

5、由于USB数据和握手响应不携带地址信息，因此当响应到达主机时，主机无法根据接收到的数据来区分其来源，而是将响应识别为最近一次探测到的设备。因此，在USB集线器向上游转发注入响应的情况下，来自攻击平台的响应会被识别为目标设备。

研究由澳大利亚国防部的国防科技集团完成。参考链接：
https://www.usenix.org/conference/usenixsecurity23/presentation/dumitru

- 例子：震网APT攻击
  - 漏洞利用、安装植入：
    - 快捷方式文件解析漏洞（MS10-046）
    - RPC远程执行溢出漏洞（MS08-067 ）
    - 打印机后台程序服务漏洞（MS10-061）
    - 任务计划程序权限提升漏洞（MS10-092）
    - 内核模式驱动程序权限提升漏洞（MS10-073）



内容概要

- 操作系统的背景知识
- 操作系统自身脆弱性
- **操作系统对安全支持**
- 总结

内容概要

- 存储保护
- 安全隔离
- 最小特权
- 身份认证
- 访问控制
- 安全审计

存储保护

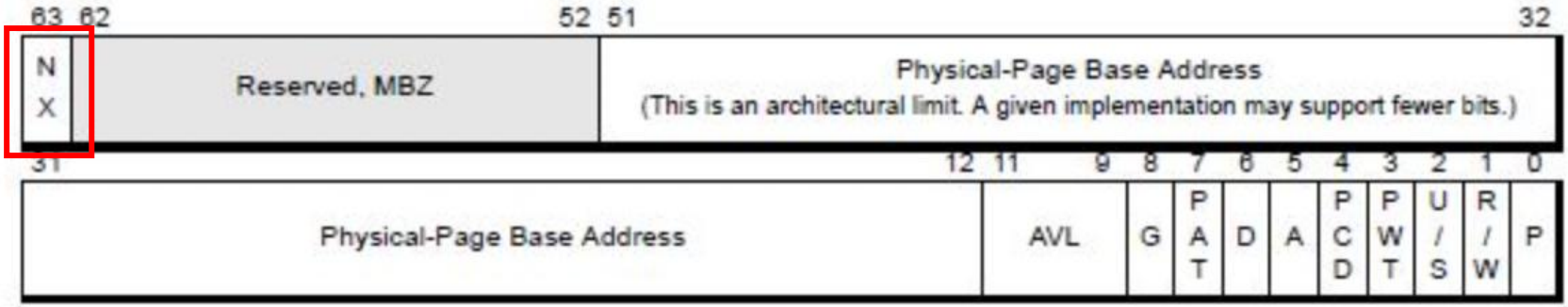
主要的存储保护机制有：

- **数据执行保护**（DEP）
- **地址空间随机化**（ASLR）
- **内核地址空间随机化**（KASLR）等。

数据执行保护（DEP）

DEP是一种安全机制，用于防止在**非执行内存**区域（如堆和栈）执行代码。这主要是为了阻止某些类型的攻击，特别是恶意代码注入攻击。

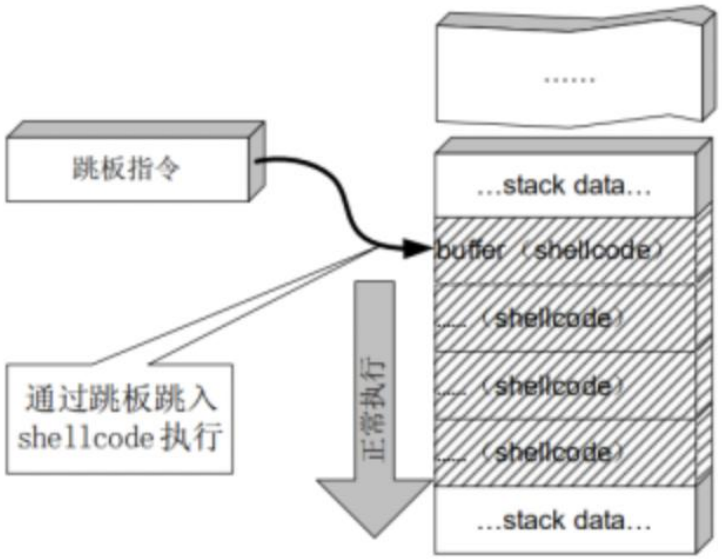
它需要 **CPU 支持**，AMD称为NX位、Intel称为XD位。**操作系统**通过设置**页表项**的**最高位NX/XD**属性标记，来指明是否可以从该内存中执行代码。若NX/XD为**0**表示页面**允许执行**，为**1**表示**不允许执行**。



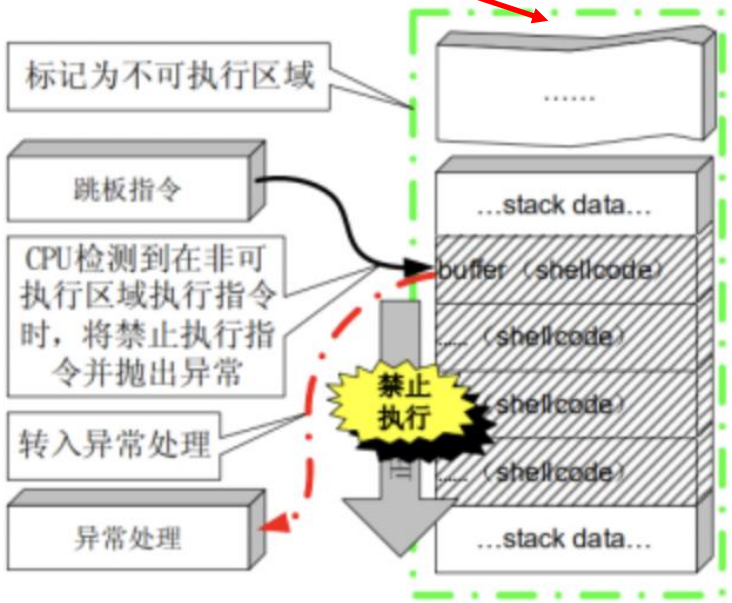
数据执行保护（DEP）

启用DEP前，此数据页的NX/XD位为0，可正常执行shellcode

启用DEP后，此数据页的NX/XD位为1，执行shellcode时CPU 就会抛出异常



经典溢出流程



启用DEP后流程

地址空间布局随机化（ASLR）

ASLR是一种针对缓冲区溢出的安全保护技术，在程序加载时，通过对**堆**、**栈**、**共享库**映射等线性区布局的**随机化**，增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置。

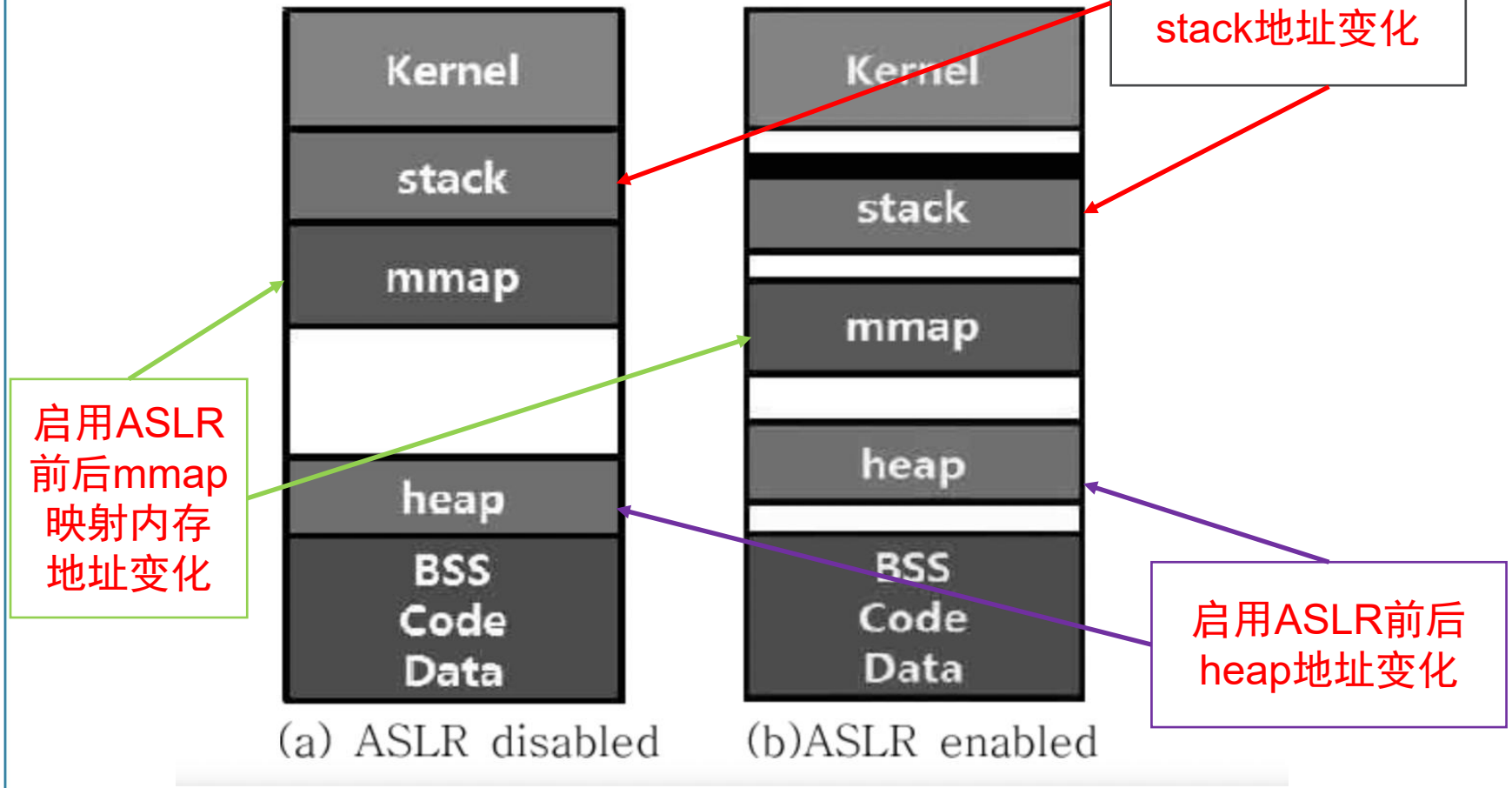
ASLR**不**会对**text段**、**data段**、**bss段**进行**随机化**

Linux下的ASLR有三种级别：

- 0：关闭ASLR，没有随机化
- 1：普通ASLR，mmap基地址、栈基地址、.so加载基地址都将被随机化
- 2：增强ASLR，增加了堆随机化　下面以级别2为例

地址空间布局随机化（ASLR）

ASLR主要随机化从用户空间地址空间顶部到堆栈的距离，以及从堆栈保留空间底部到第一个mmap的距离。



地址空间布局随机化（ASLR）

```c
#include <stdio.h>

void func();

int uninitialGlobalVar;
int globalVar = 1;

int main(void)
{
    int localVar = 1;

    printf("Address of func() is %p, in text setment\n", func);
    printf("Address of uninitialGlobalVar is %p, in bss segment\n", &uninitialGlobalVar);
    printf("Address of globalVar is %p, in data segment\n", &globalVar);
    printf("Address of localVar is %p, in stack\n", &localVar);

    return 0;
}

void func()
{
    ;
}
```

实验程序addr.c,打印函数所在代码段、变量所在bss段、数据段以及栈地址

地址空间布局随机化（ASLR）

```
root@iZzv2fbt7b40akZ:~# sudo echo 0 > /proc/sys/kernel/randomize_va_space
root@iZzv2fbt7b40akZ:~# ./addr
Address of func() is 0x4005f0,in text setment
Address of uninitialGlobalVal is 0x601040,in bss segment
Address of globalVal is 0x601038,in data segment
Address of localVal is 0x7fffffffe524,in stack
```

开启ASLR前，第一次运行

```
root@iZzv2fbt7b40akZ:~# ./addr
Address of func() is 0x4005f0,in text setment
Address of uninitialGlobalVal is 0x601040,in bss segment
Address of globalVal is 0x601038,in data segment
Address of localVal is 0x7fffffffe524,in stack
```

开启ASLR前，第二次运行

地址空间布局随机化（ASLR）

```
root@iZzv2fbt7b40akZ:~# sudo echo 2 > /proc/sys/kernel/randomize_va_space
root@iZzv2fbt7b40akZ:~# ./addr
Address of func() is 0x4005f0,in text setment
Address of uninitialGlobalVal is 0x601040,in bss segment
Address of globalVal is 0x601038,in data segment
Address of localVal is 0x7ffd0ef4d6e4,in stack
```

开启ASLR后，第一次运行

```
root@iZzv2fbt7b40akZ:~# ./addr
Address of func() is 0x4005f0,in text setment
Address of uninitialGlobalVal is 0x601040,in bss segment
Address of globalVal is 0x601038,in data segment
Address of localVal is 0x7ffe3bb849c4,in stack
```

开启ASLR后，第二次运行

内核堆栈随机化（KASLR）

内核地址随机化用于增强操作系统的安全性，它的基本原理是通过在每次系统启动时**随机化内核代码、数据和其他关键结构在内存中的位置**，从而使得攻击者难以利用内核漏洞进行精确的攻击。

KASLR专门针对**操作系统内核的内存布局**进行随机化，而ASLR是针对**用户空间应用程序**和其加载的**库**的内存布局进行随机化。

内核堆栈随机化（KASLR）

正常情况下，kernel image会按照vmlinux链接脚本中的链接地址去映射虚拟地址，如果开启kaslr，则会重新再映射一次，映射到链接地址 + offset的新地址上去。如果offset每次开机都随机生成，那么每次开机后，kernel image最后映射的虚拟地址都不一样，这就是内核地址随机化。

例如，在Android中，bootloader中会随机生成一个随机数seed，通过fdt传给kernel，kernel根据这个seed生成地址偏移offset，然后去做重映射。

内容概要

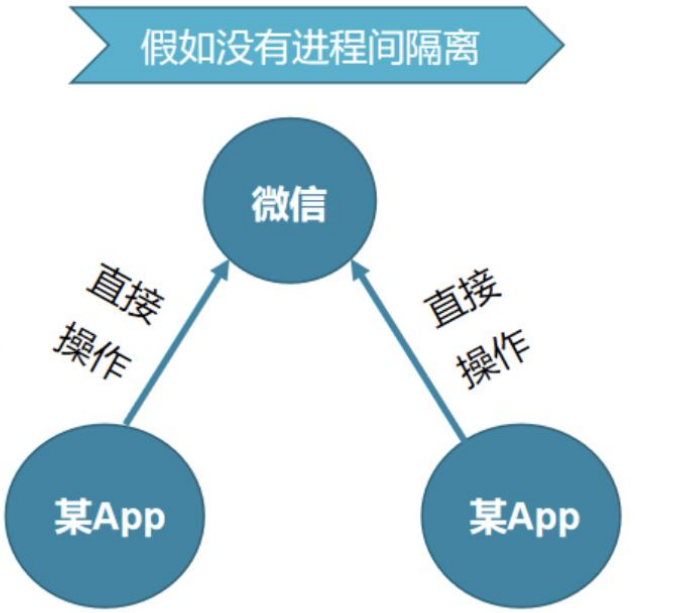
- 存储保护
- **安全隔离**
- 最小特权
- 身份认证
- 访问控制
- 安全审计

安全隔离

- 隔离是一种将计算机系统中的不同部分隔离开来的机制，以提高系统的安全性和可靠性，包括对不同的进程、应用程序、服务或者是整个子系统的隔离。
- 隔离的主要目的是确保一个部分的故障或安全威胁不会影响到其他部分，从而限制恶意行为的影响范围，提升系统的整体安全性。
- 常见的隔离技术有进程隔离、沙箱技术等

进程隔离

进程隔离是为保护操作系统中进程互不干扰而设计的一组不同硬件和软件的技术。这个技术是为了避免进程A写入进程B的情况发生。

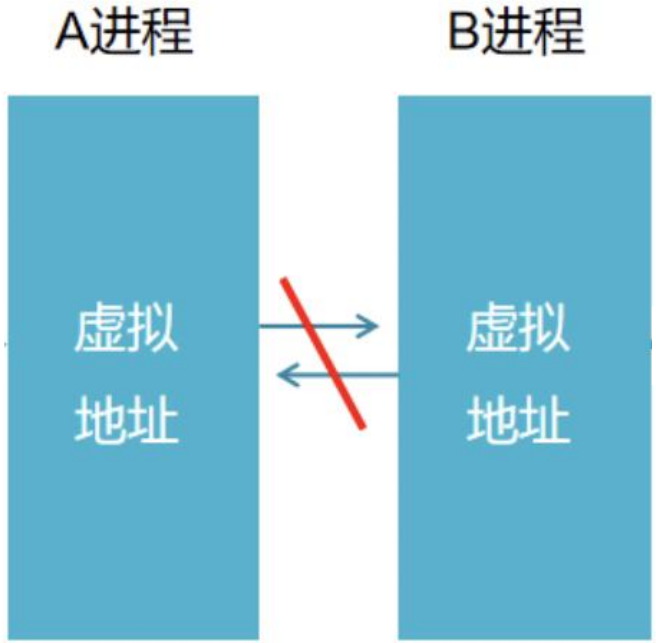


大家可以想象，如果没有进程间隔离，

那么是一种多么可怕的场景

进程隔离

进程间的隔离采用了虚拟地址空间。进程A的虚拟地址和进程B的虚拟地址不同，这样就能防止进程A将数据信息写入进程B。



沙箱技术

- 沙箱（Sandbox）是一种安全机制，为运行中的程序提供隔离环境。
- 核心是创建一个对程序操作进行限制的执行环境，在沙箱中运行不受信任和未知目的的程序，可以避免对系统可能造成的破坏

沙箱技术

沙箱**主要**通过**重定向**实现安全隔离环境。

重定向，让沙箱内软件操作的文件、注册表等路径重定向到其他位置（**沙箱指定位置**），这样可疑软件程序本来想访问或执行的系统资源就不会被访问或执行，保证资源的安全性

举个例子，如果沙箱中的程序要删除c:\boot.ini

- 沙箱hook 原始ZwDeleteFile函数，函数名是HOOK_ZwDeleteFile。
- 在HOOK_ZwDeleteFile中，将路径c:\boot.ini加上一个前缀c:\sandbox\boot.ini，转到沙箱内文件路径。
- c:\sandbox\boot.ini不存在，会先把c:\boot.ini拷贝到沙箱内。
- 然后调用原始ZwDeleteFile，删除c:\sandbox\boot.ini。

内容概要

- 存储保护
- 安全隔离
- **最小特权**
- 身份认证
- 访问控制
- 安全审计

最小特权原则

一方面给予主体“必不可少”的特权，保证所有的主体都能在所赋予的特权之下完成所需要完成的任务或操作;

另一方面，它只给予主体“必不可少”的特权，限制每个主体所能进行的操作。

依据最小特权可为操作系统设立三种管理角色（三员分立）

- 系统管理员

主要负责系统的资源和运行进行配置、控制和管理

- 审计管理员

主要负责对审计记录进行分析，并根据分析结果进行处理

- 安全管理员

主要对系统中的安全策略进行配置

最小特权

最小特权管理基于操作系统对**权限进行更细粒度的控制**，实现按需授权。

以linux为例，从linux2.2系统开始，引入了 capabilities机制。

原理：将root关联的特权细分为不同的功能组，Capabilites 作为线程的属性存在，每个功能组都可以独立启用和禁用。其本质就是将内核调用分门别类，具有相似功能的内核调用被分到同一组中。

权限检查的过程：在**执行特权操作**时，**检查线程是否具有该特权操作所对应的** capabilities，并以此为依据，决定是否可以执行特权操作

最小特权管理

Linux capabilities 分为**进程 capabilities** 和**文件 capabilities**。对于进程来说，capabilities 是**细分到线程**的，即每个线程可以有自己的capabilities。

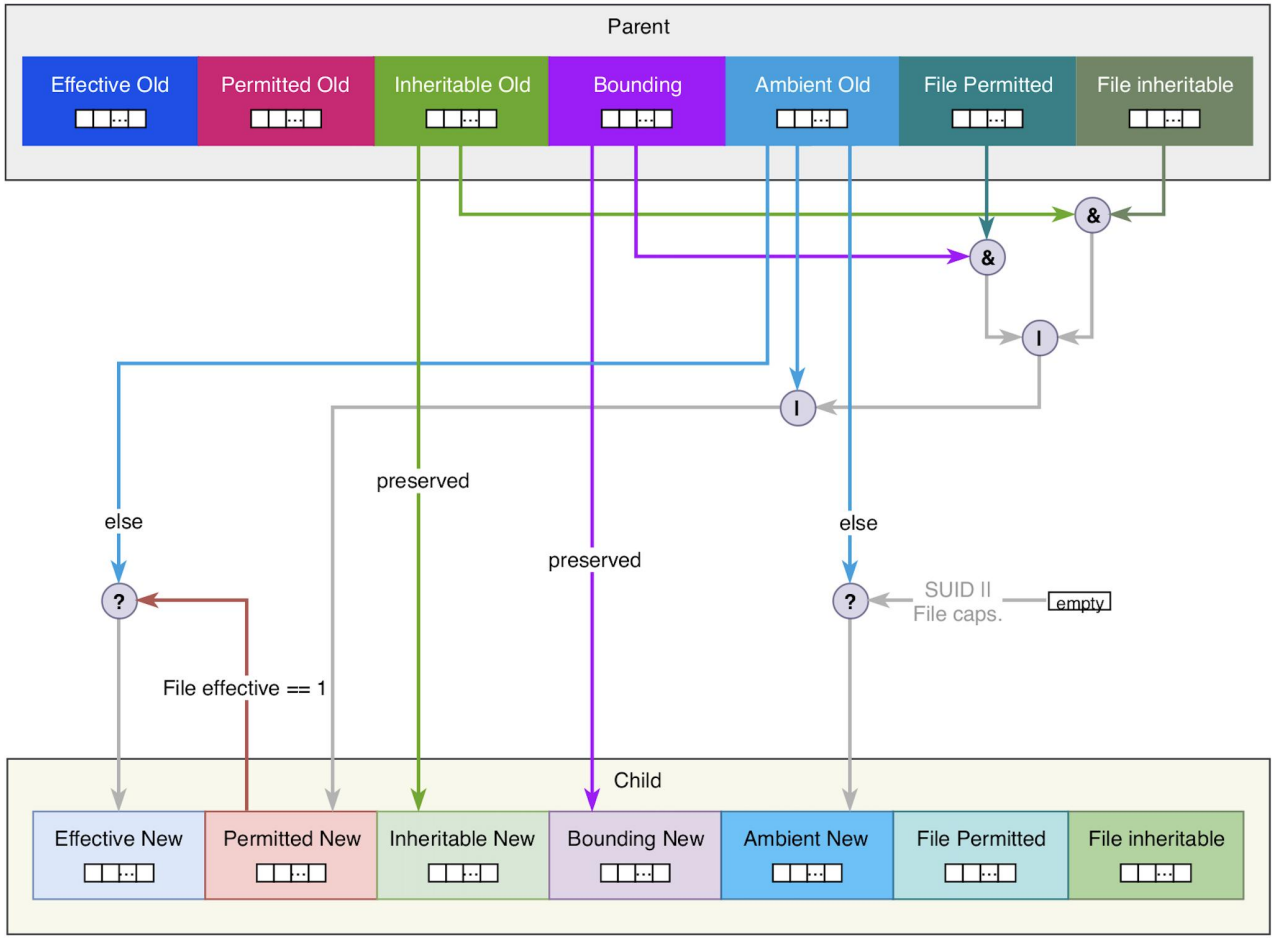
每一个线程，具有 5 个 capabilities 集合，分别为**Permitted** 、**Effective**、**Inheritable**、**Bounding**、**Ambient** 。

```
root@pan:~# cat /proc/self/status | grep Cap
CapInh: 0000000000000000
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
```

最小特权

当父线程通过fork（） 创建子线程时，子线程会完全复制父进程的 capabilities 信息。

当父线程通过execve（）创建子线程时，则会通过一定规则将capabilities传递给子线程。



内容概要

- 存储保护
- 安全隔离
- 最小特权
- **身份认证**
- 访问控制
- 安全审计

认证

- 身份认证对用户身份进行识别和验证，是防止恶意用户进入系统的一个重要环节。

- 识别

识别访问者身份，并可区分不同用户。

- 验证

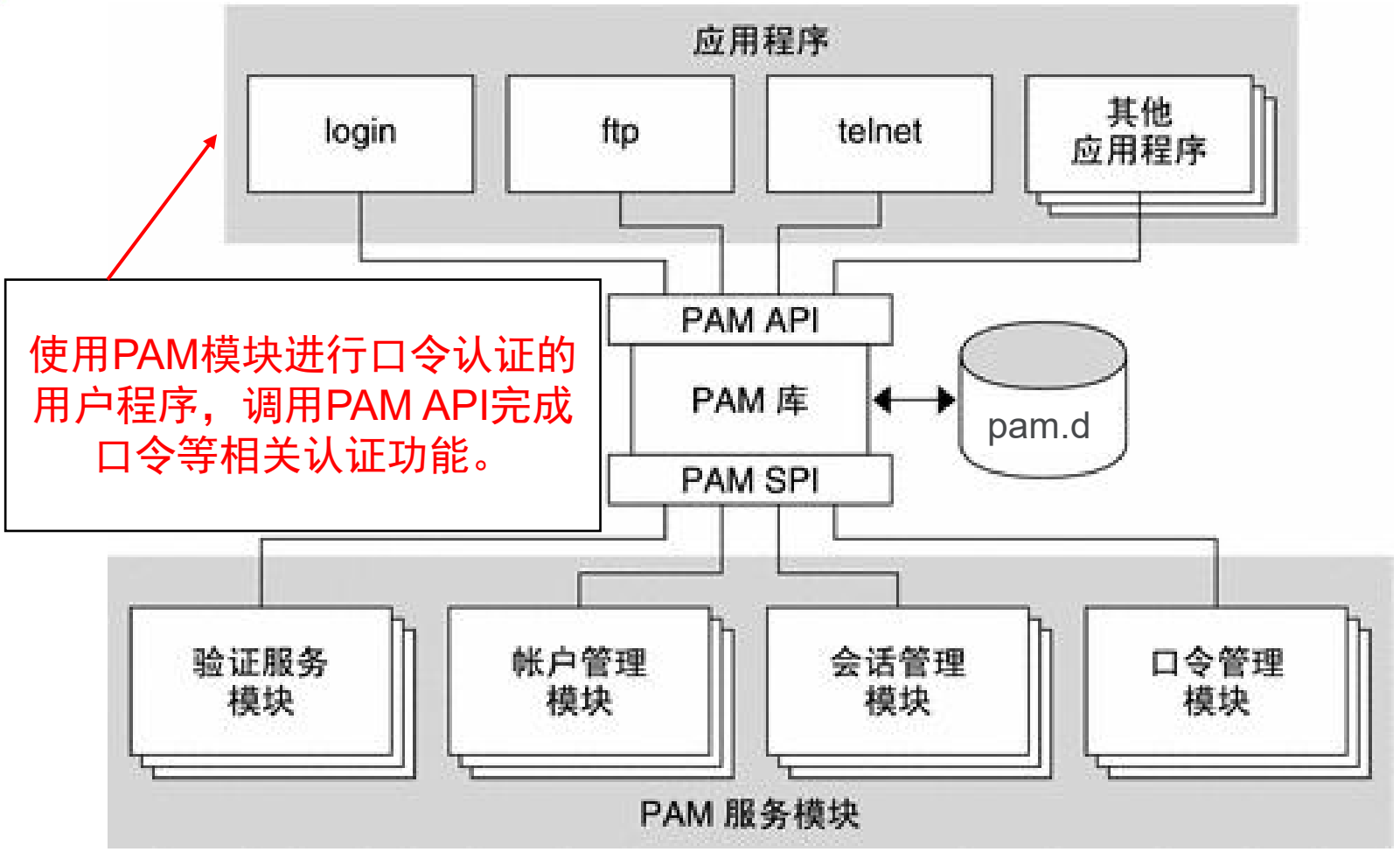
对访问者生成的身份进行确认。

认证

PAM（Pluggable Authentication Modules）即可插拔验证模块，它是一个非常完善的身份验证机制。

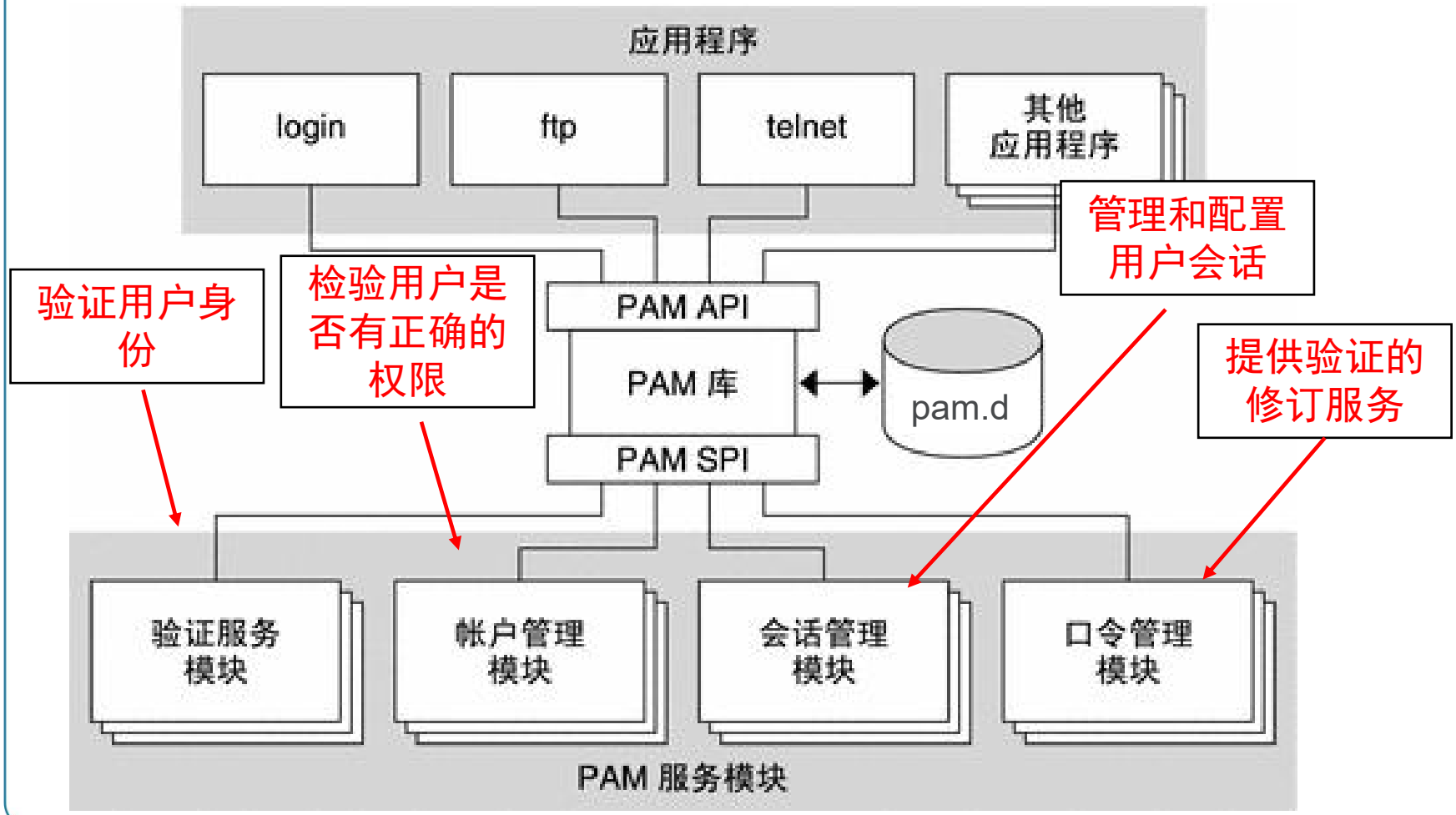
它采用模块化设计和插件功能，可以轻易的在应用程序中插入新的鉴别模块或者替换原来的组件。

PAM 的易用性较强，它对上层屏蔽了鉴别的具体细节，还实现了多鉴别机制的集成问题。

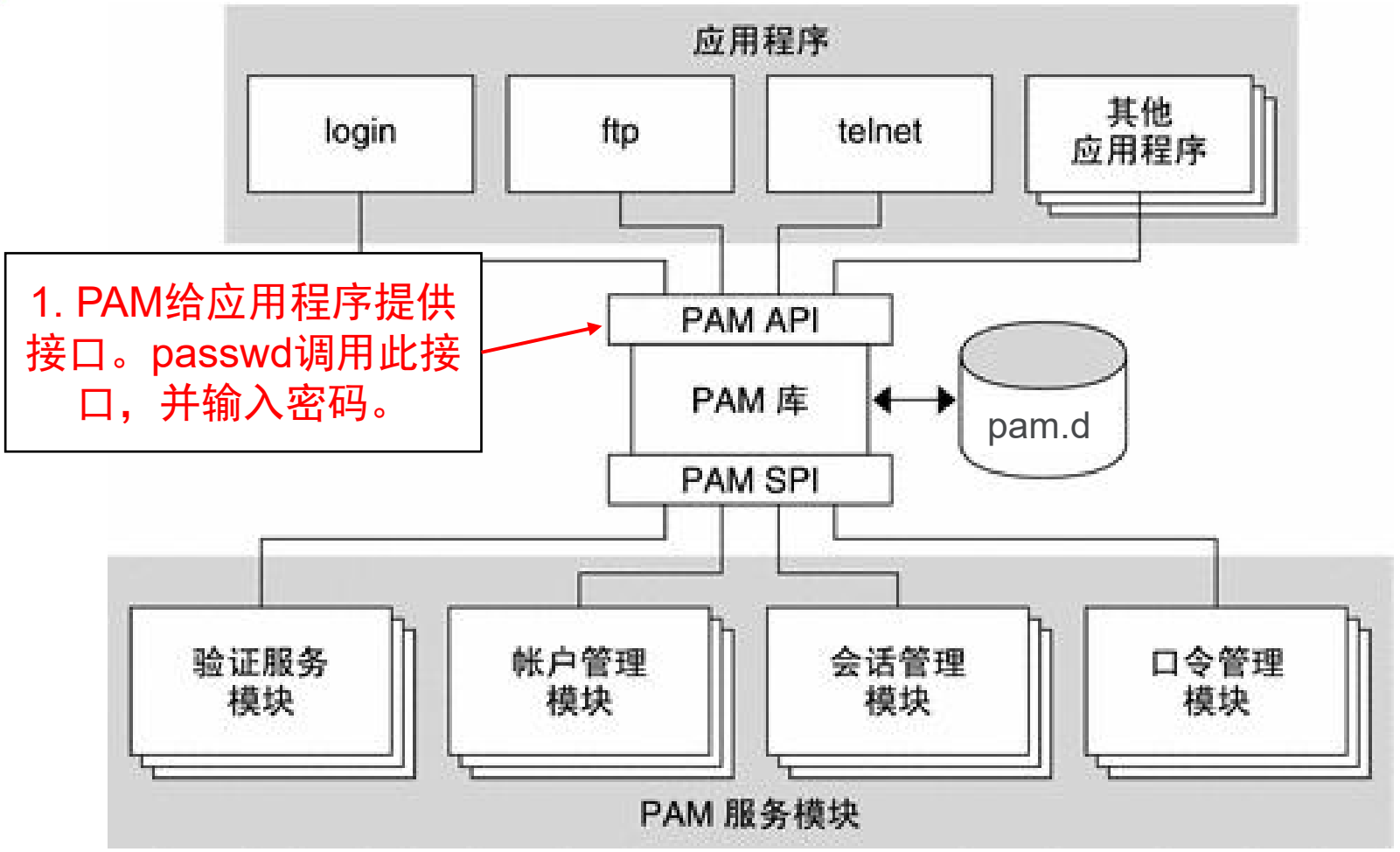

应用程序
login
ftp
telnet
其他
应用程序
使用PAM模块进行口令认证的用户程序，调用PAM API完成口令等相关认证功能。
PAM API
PAM 库
PAM SPI
pam.d
验证服务
模块
帐户管理
模块
会话管理
模块
口令管理
模块
PAM 服务模块

PAM系统架构

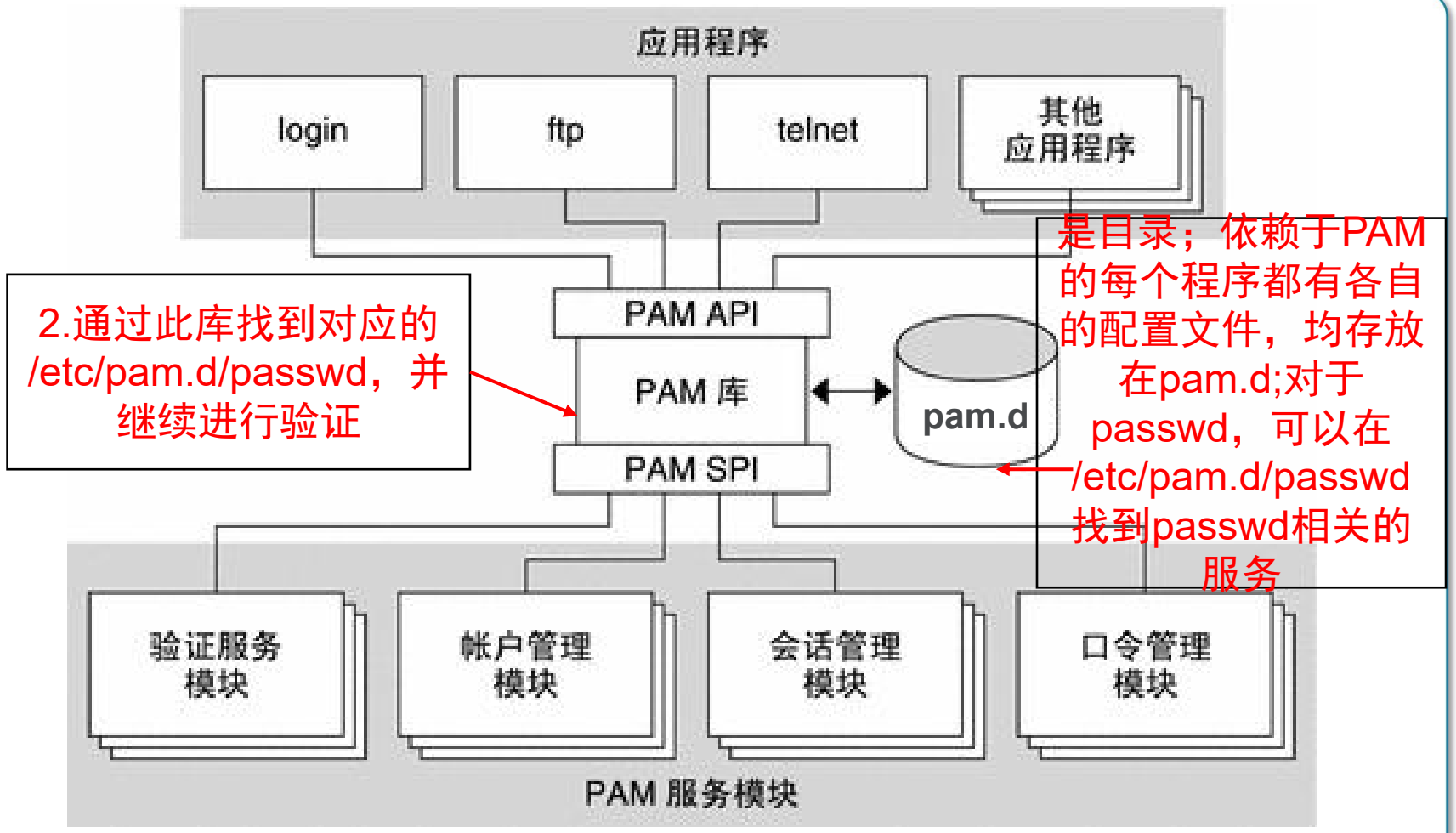
验证类别（type）


应用程序
login
ftp
telnet
其他
应用程序
管理和配置
用户会话
验证用户身
份
检验用户是
否有正确的
权限
PAM API
PAM 库
PAM SPI
pam.d
提供验证的
修订服务
验证服务
模块
帐户管理
模块
会话管理
模块
口令管理
模块
PAM 服务模块

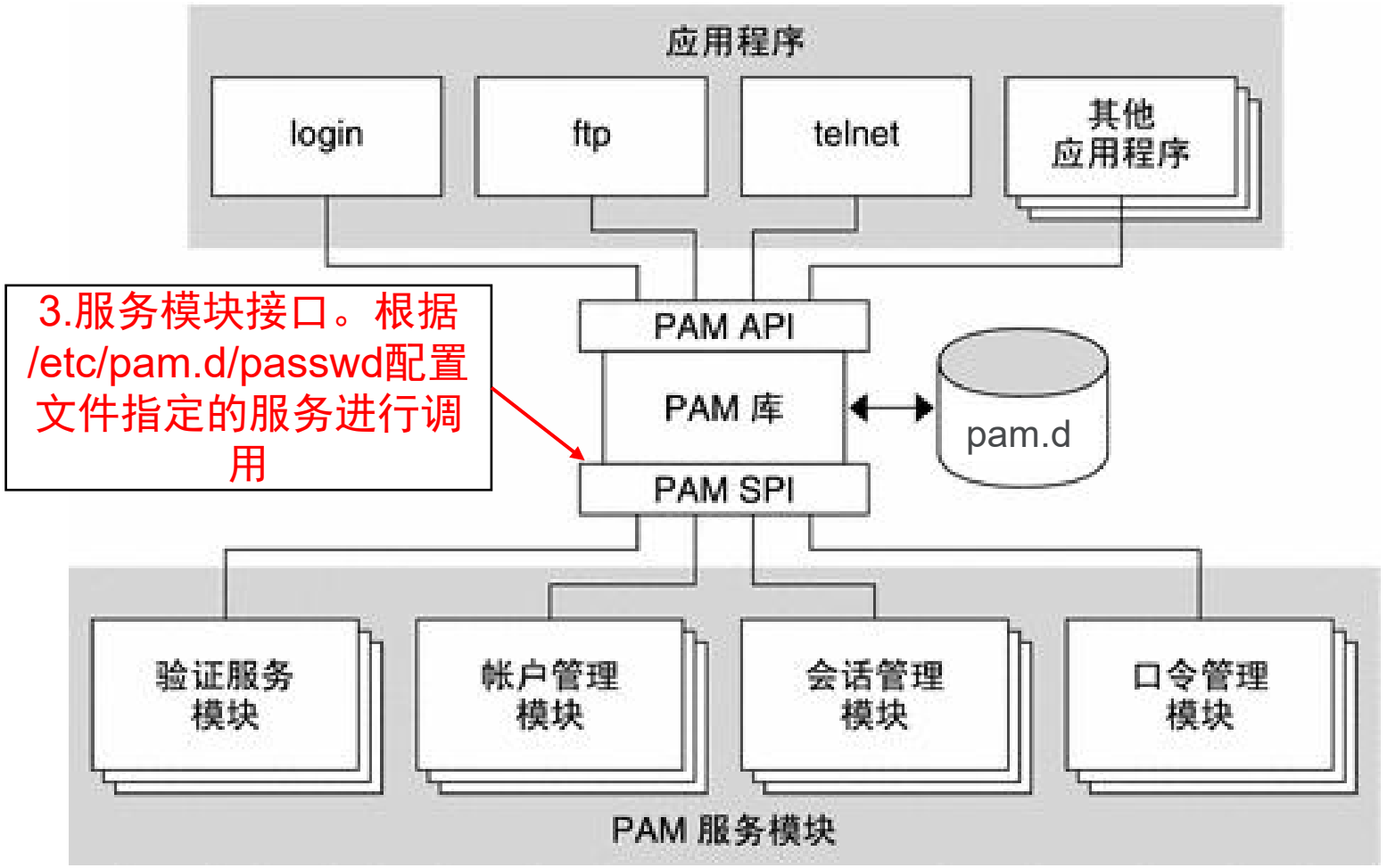
下面以程序passwd为例来进行过程分析：



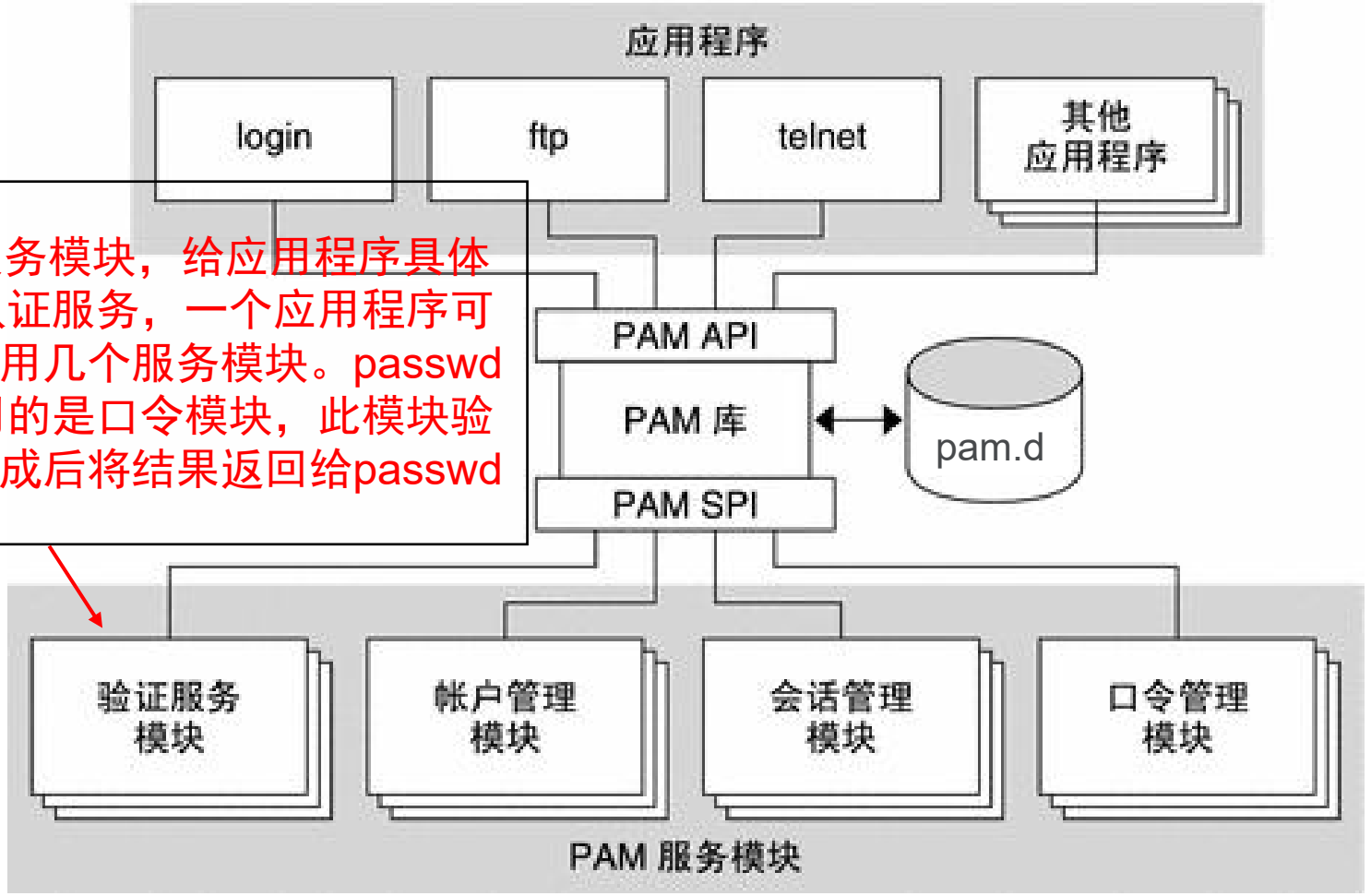
PAM系统架构


应用程序
login
ftp
telnet
其他
应用程序
2.通过此库找到对应的/etc/pam.d/passwd，并继续进行验证
PAM API
PAM 库
PAM SPI
pam.d
是目录；依赖于PAM的每个程序都有各自的配置文件，均存放在pam.d;对于passwd，可以在/etc/pam.d/passwd找到passwd相关的服务
验证服务
模块
帐户管理
模块
会话管理
模块
口令管理
模块
PAM 服务模块

PAM系统架构


应用程序
login
ftp
telnet
其他
应用程序
3.服务模块接口。根据/etc/pam.d/passwd配置文件指定的服务进行调用
PAM API
PAM 库
PAM SPI
pam.d
验证服务
模块
帐户管理
模块
会话管理
模块
口令管理
模块
PAM 服务模块

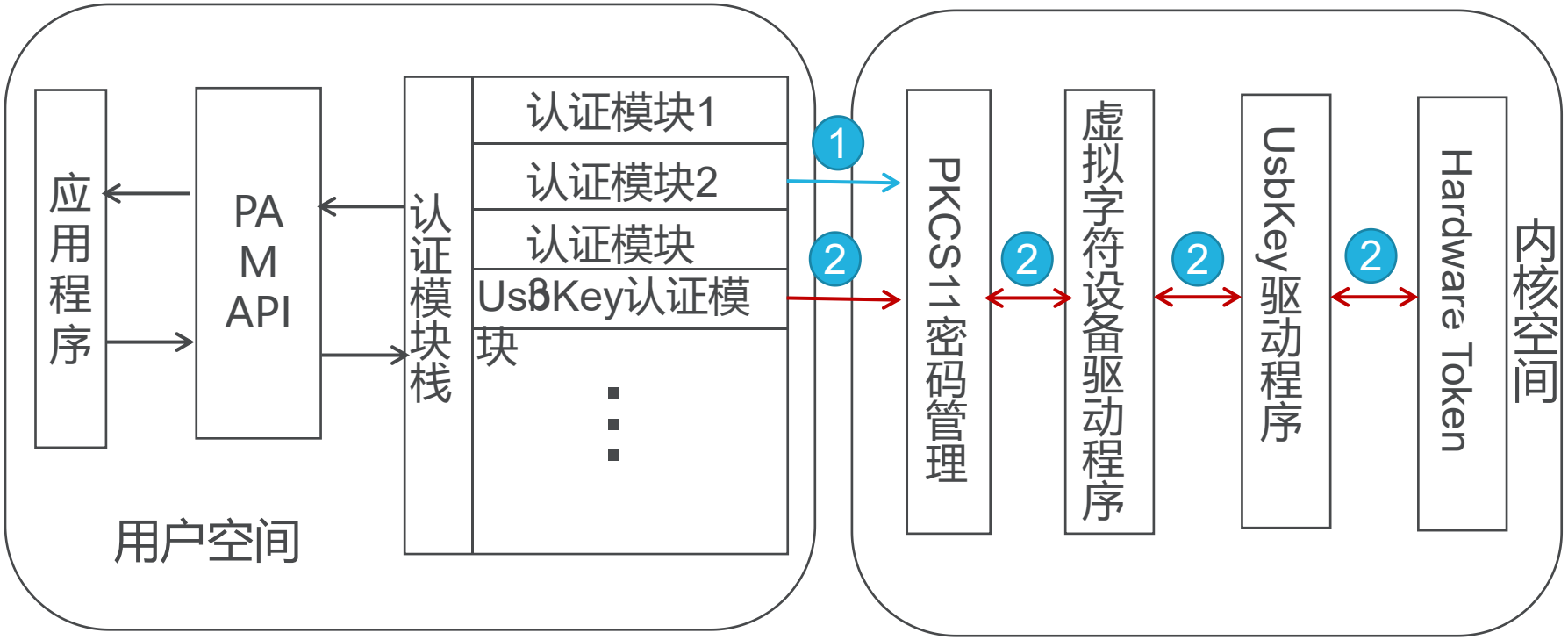
PAM系统架构



4.服务模块，给应用程序具体的认证服务，一个应用程序可以使用几个服务模块。passwd调用的是口令模块，此模块验证完成后将结果返回给passwd

PAM系统架构

Pam交互过程



（1）PKCS#11是公钥加密标准。

（2）字符设备驱动程序负责将设备的操作与内核进行协调，应用程序需要对设备进行读写时，它会通过系统调用将请求传递给字符设备驱动程序，然后由驱动程序将请求传递给实际的设备。

内容概要

- 存储保护
- 安全隔离
- 最小特权
- 身份认证
- **访问控制**
- 安全审计

访问控制

- 访问控制用来提供**授权**
- **用户**在通过**身份鉴别**后，还需通过**授权**，才能**访问资源**或进行操作
- 访问控制三要素
  - 主体
  - 客体
  - 控制策略
- 访问控制主要类型
  - 自主访问控制（DAC）
  - 强制访问控制（MAC）

自主访问控制（DAC）

- 自主访问控制（DAC）

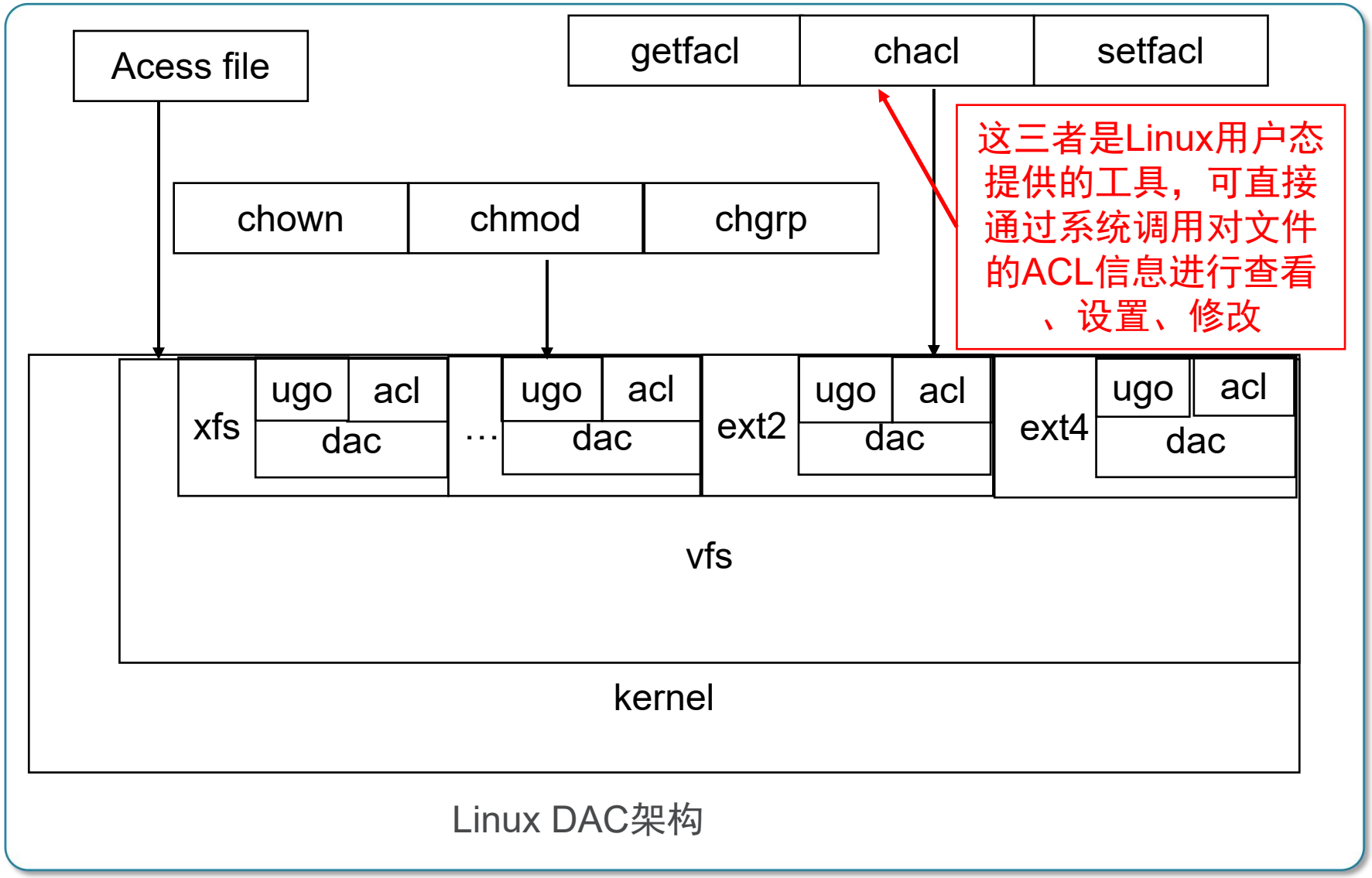
基于对主体和主体所属的主体组的识别来限制对客体的访问

- 自主的含义

一个拥有特定权限的主体(subject)有能力将其权限传递给其它主体

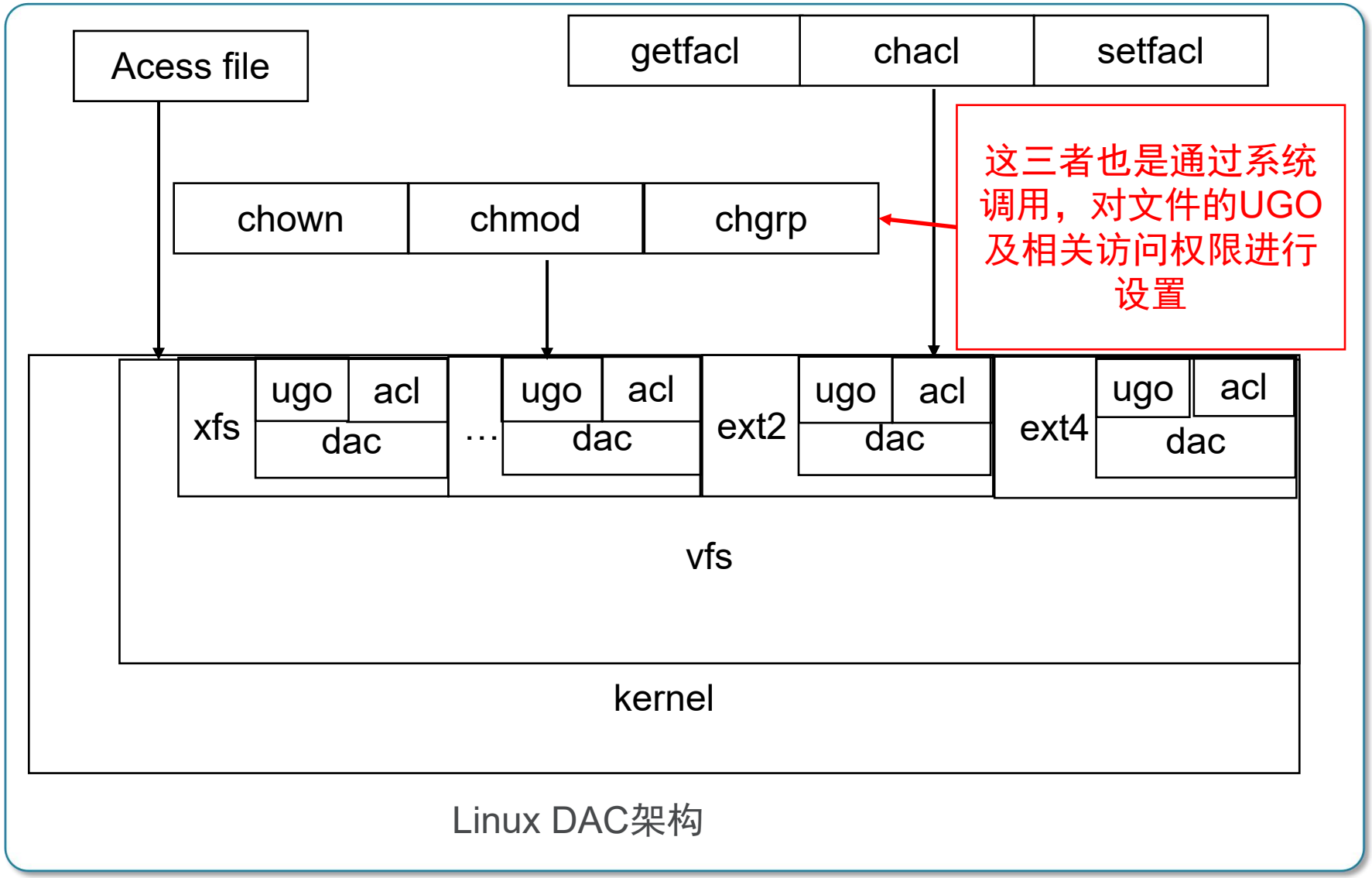
- DAC包含两部分UGO（user、group、other）和ACL（访问控制表）,**ACL是UGO的扩充**。
  - UGO对一个文件只能设置一个所属用户一个组和其他人，相关信息存储在文件的inode中
  - ACL对一个文件可以设置多个所属用户和多个组，相关信息存储在文件的扩展属性中

自主访问控制



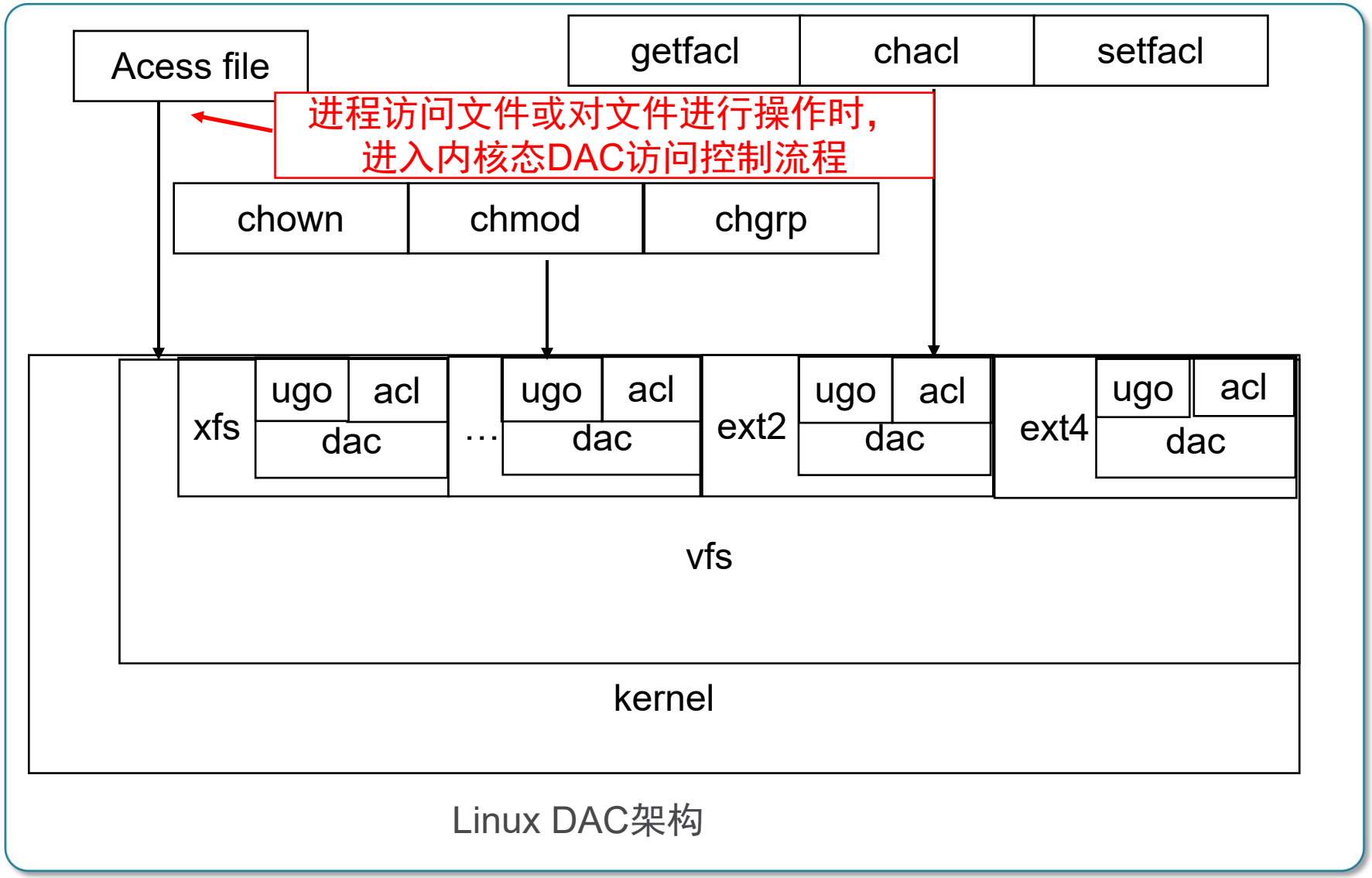
Linux DAC架构

自主访问控制



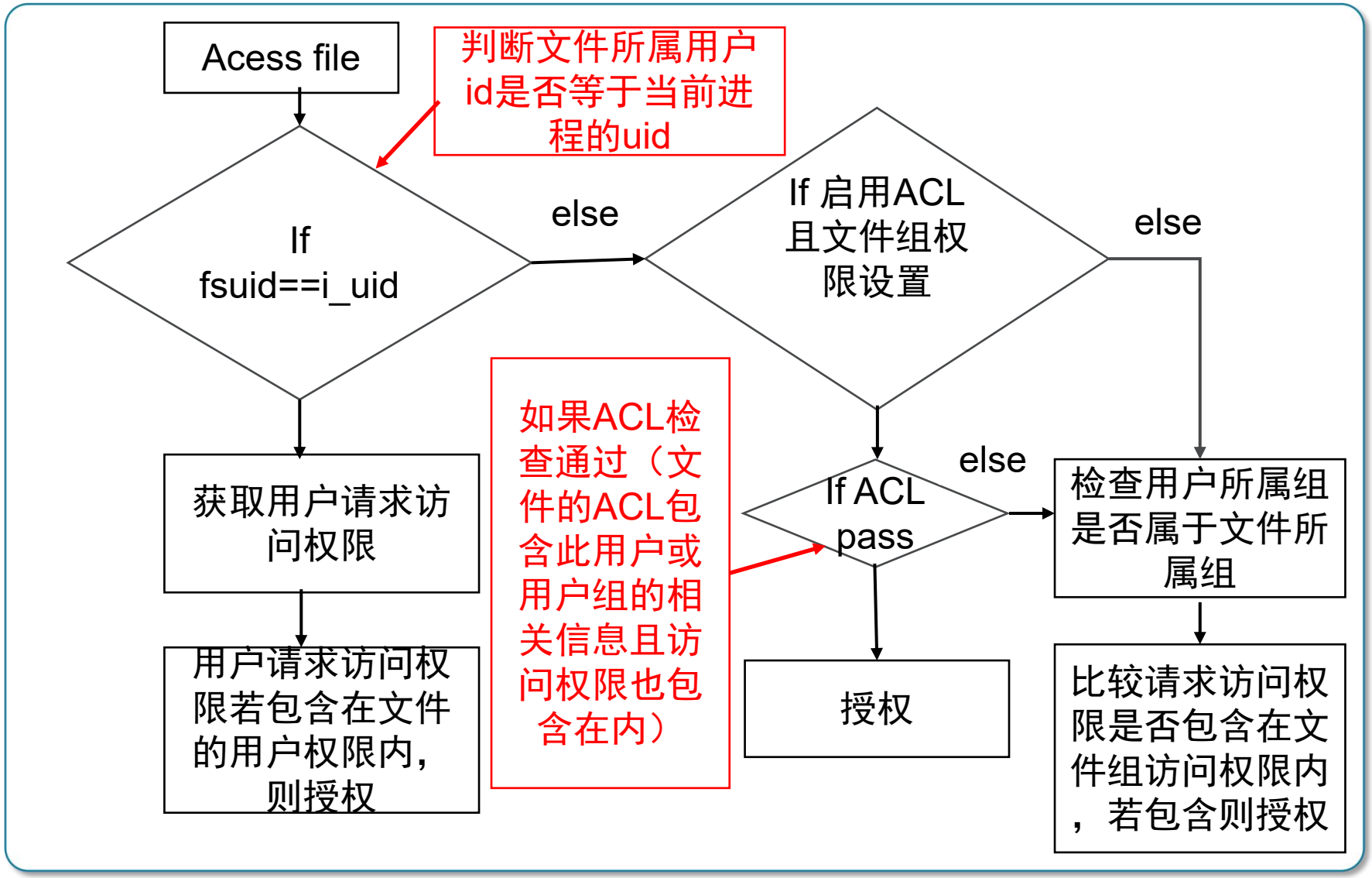
Linux DAC架构

自主访问控制



Linux DAC架构

自主访问控制


Acess file
判断文件所属用户id是否等于当前进程的uid
If fsuid==i_uid
else
If 启用ACL且文件组权限设置
else
获取用户请求访问权限
用户请求访问权限若包含在文件的用户权限内，则授权
如果ACL检查通过（文件的ACL包含此用户或用户组的相关信息且访问权限也包含在内）
If ACL pass
else
授权
检查用户所属组是否属于文件所属组
比较请求访问权限是否包含在文件组访问权限内，若包含则授权

自主访问控制

- 在大型系统中主、客体的数量巨大，使用DAC将使系统开销大到难以支付的程度
- **由此引入强制访问控制**

强制访问控制（MAC）

- 强制访问控制方法

每个主体和每个客体都有既定安全属性，是否能执行特定的操作取决于二者之间的关系 。

- 实现方式
  - 由特定用户(管理员)实现授权管理;
  - 通常指TCSEC的多级安全策略:安全属性用二元组表示，记作 (密集，类别集合)，密集表示机密程度，类别集合表示部门或组织的集合。
- 弱点

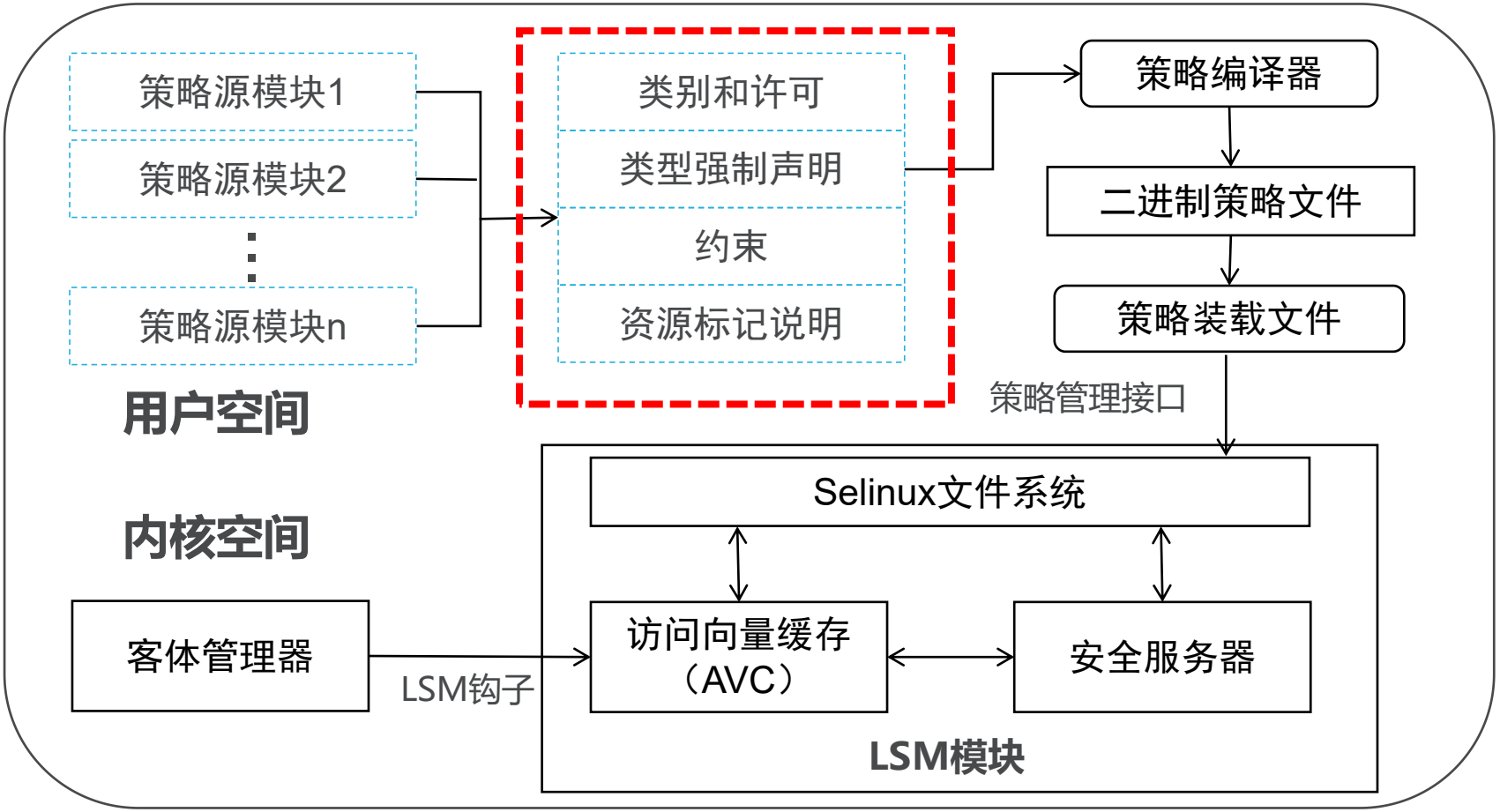
应用的领域比较窄，使用不灵活，一般只用于军方等具有明显等级观念的行业或领域，完整性方面控制不够

强制访问控制（MAC）

- 两个重要的安全特性 (公理)
  - Simple Security Condition:**主体读客体**，当且仅当用户的安全等级必须**大于或等于**该客体的安全级，并且该用户必须具有包含该客体所有访问类别的类别集合
  - *-Property(Star Property): **一个主体/用户要写一个客体**，当且仅当用户的安全等级**不大于**该客体安全等级， 并且该客体包含该用户的所有类别 。
- 安全特性

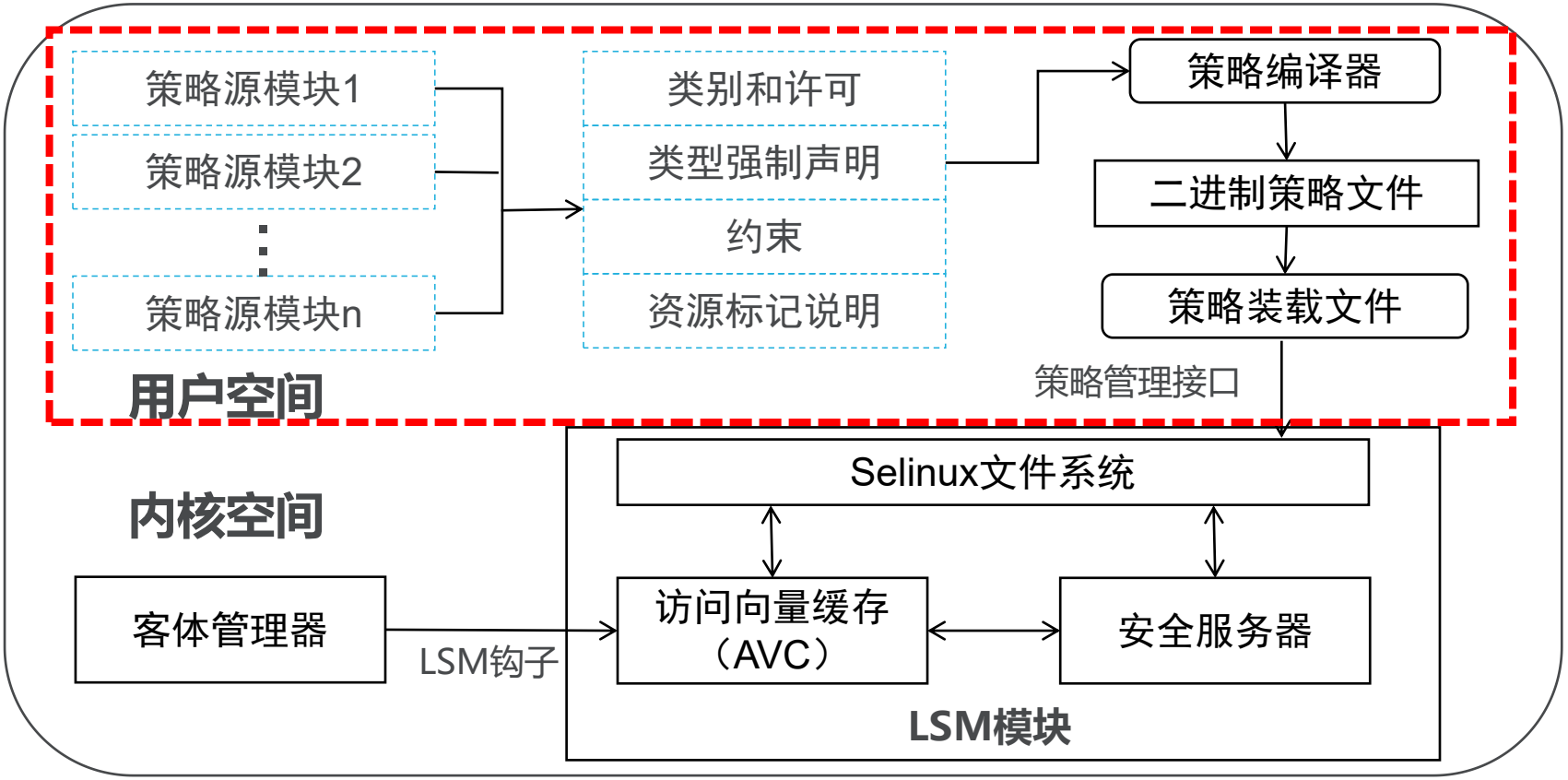
保证了信息的单向流动，即信息只能向高安全属性的方向流动

Selinux架构



（1）类别许可：安全服务器的客体类别。
（2）类型强制声明：包括所有的类型声明和所有的TE（类型强制规则）。
（3）约束：TE规则许可范围之外的规则，为TE规则提供必要的限制。
（4）资源标记说明：对所有客体都必须添加的一个“安全上下文”标记，是SELinux实施访问控制的前提。

Selinux架构

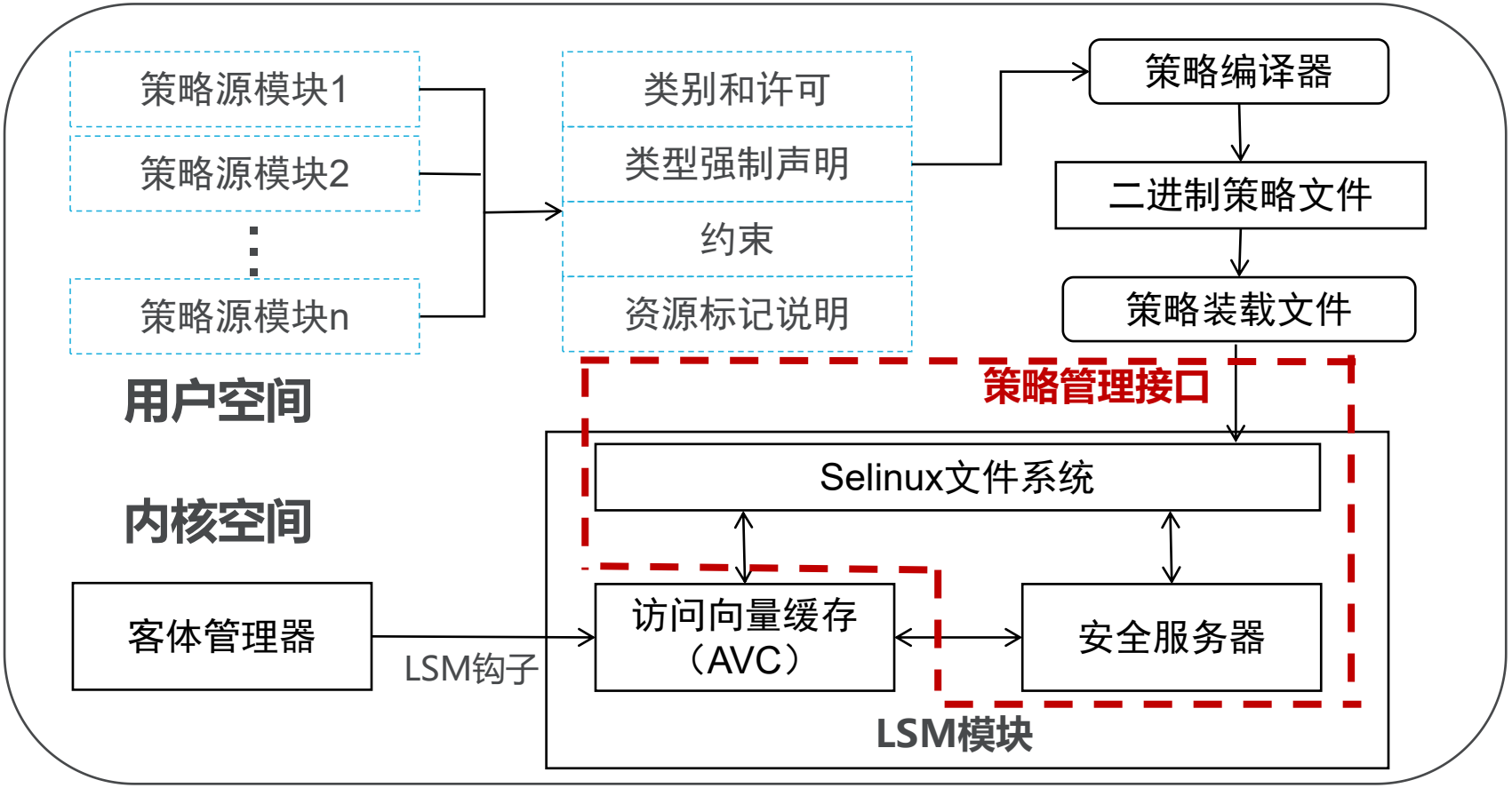


（1）通过源模块法生成一个个策略模块，策略模块聚合形成一个大的策略源文件policy.conf；

（2）策略源文件policy.conf通过策略编译器checkpolicy，生成可被内核读取的二进制文件policy.xx；

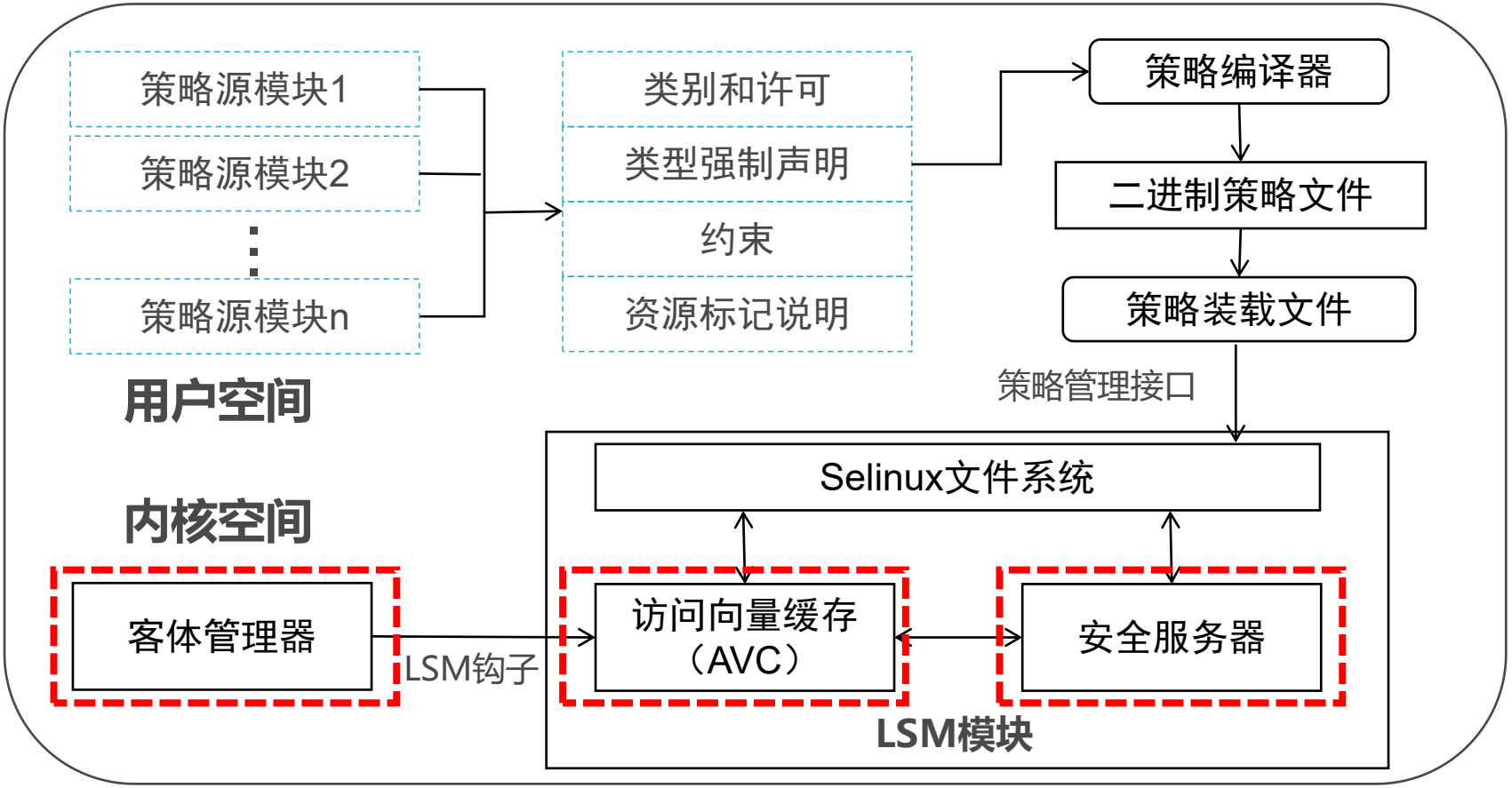
（3）policy.xx通过策略装载函数security_load_policy载入内核空间并实施访问控制。

Selinux架构



策略通过策略管理接口载入SELinux LSM模块的安全服务器中，从而决定访问控制。

Selinux架构



（1）安全服务器负责策略决定，用来验证某个主体是否可以真正的访问某个客体；
（2）客体管理器负责按照安全服务器的策略决定强制执行它管理的资源集，可以知道主体需要访问哪些资源，并且触发验证机制；
（3）访问向量缓存 (AVC) 用来记录以往的访问验证情况，提升了访问确认的速度，并为LSM钩子和内核客体管理器提供了SELinux接口。

内容概要

- 存储保护
- 安全隔离
- 最小特权
- 身份认证
- 访问控制
- **安全审计**

安全审计

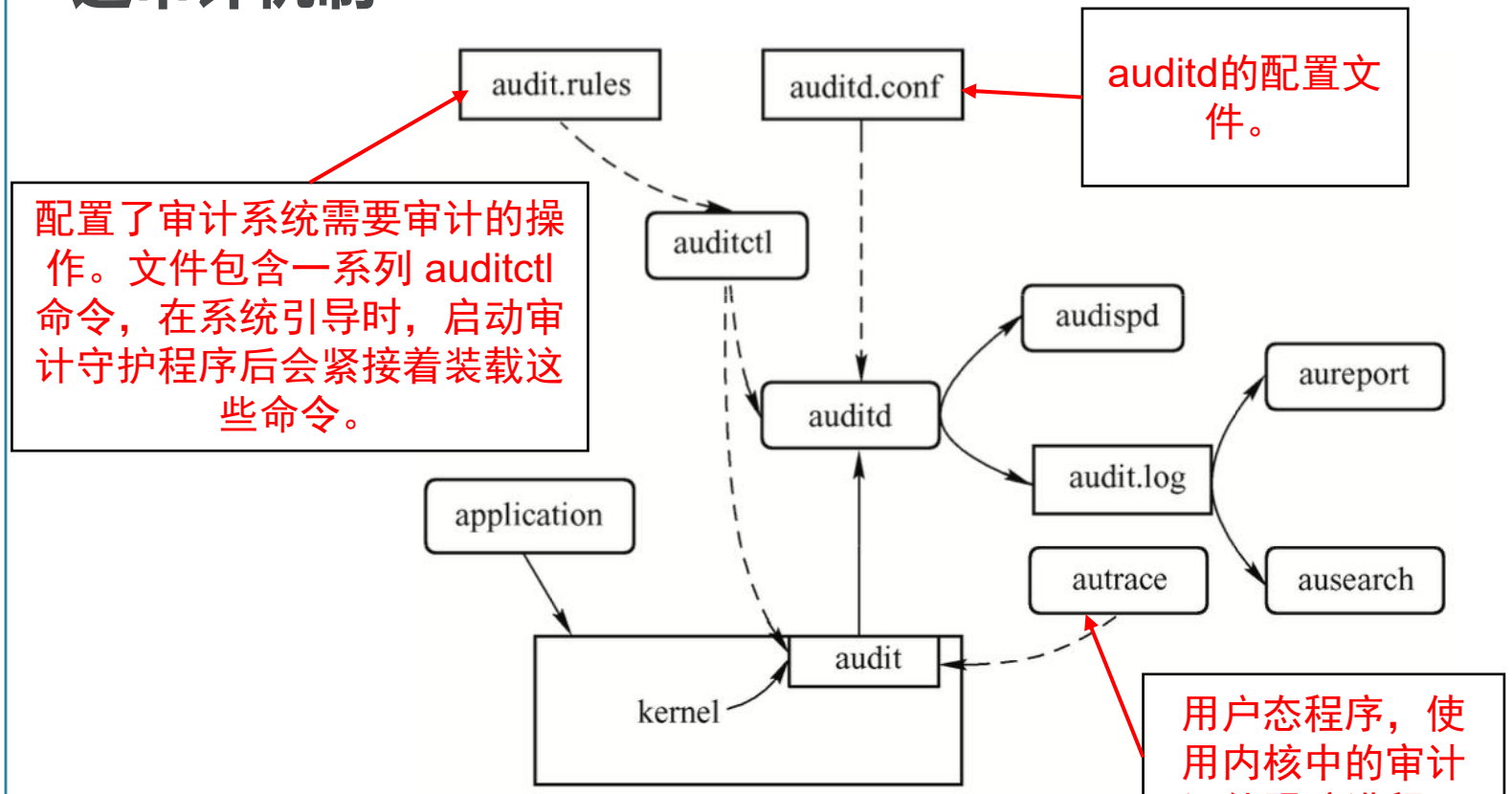
- 安全审计是由审计管理员对系统中有关安全的活动按照一定规则进行检查和审核。
- 主要目的：监视和检测非法用户对计算机系统的入侵。
- 审计(audit)机制要点
  - 选择最主要的事件加以审计，不必设置太多的审计事件，以免过多影响系统性。
  - 在系统调用的总入口处设置审计点，审计系统调用，也就审计了所有使用内核服务的事件。

安全审计

通过使用审计，系统可以跟踪许多事件类型来监视和审计系统。包括：

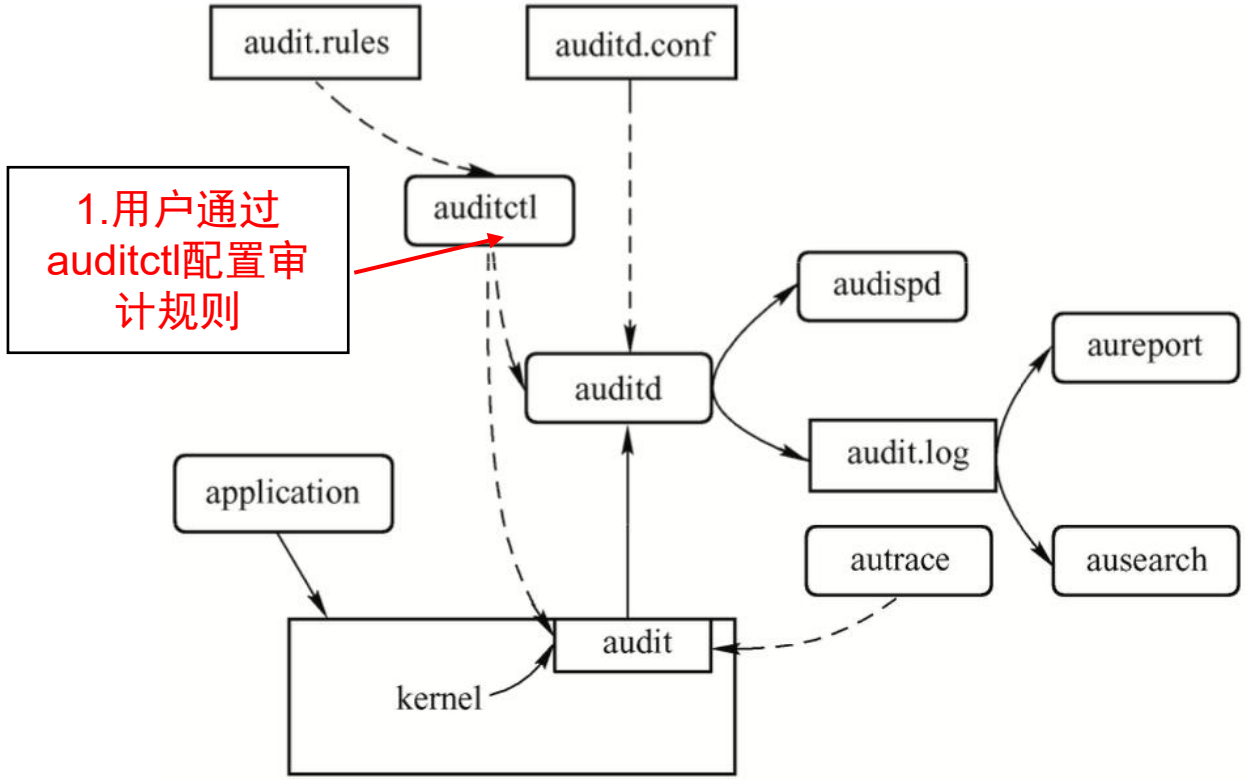
- 审核文件访问和修改
- 查看是谁更改了特定的文件
- 检测未经授权的更改
- 监控系统调用和功能
- 检测异常，如崩溃进程
- 记录单个用户使用的命令

不同操作系统审计机制大同小异，以linux系统为例详细讲述审计机制



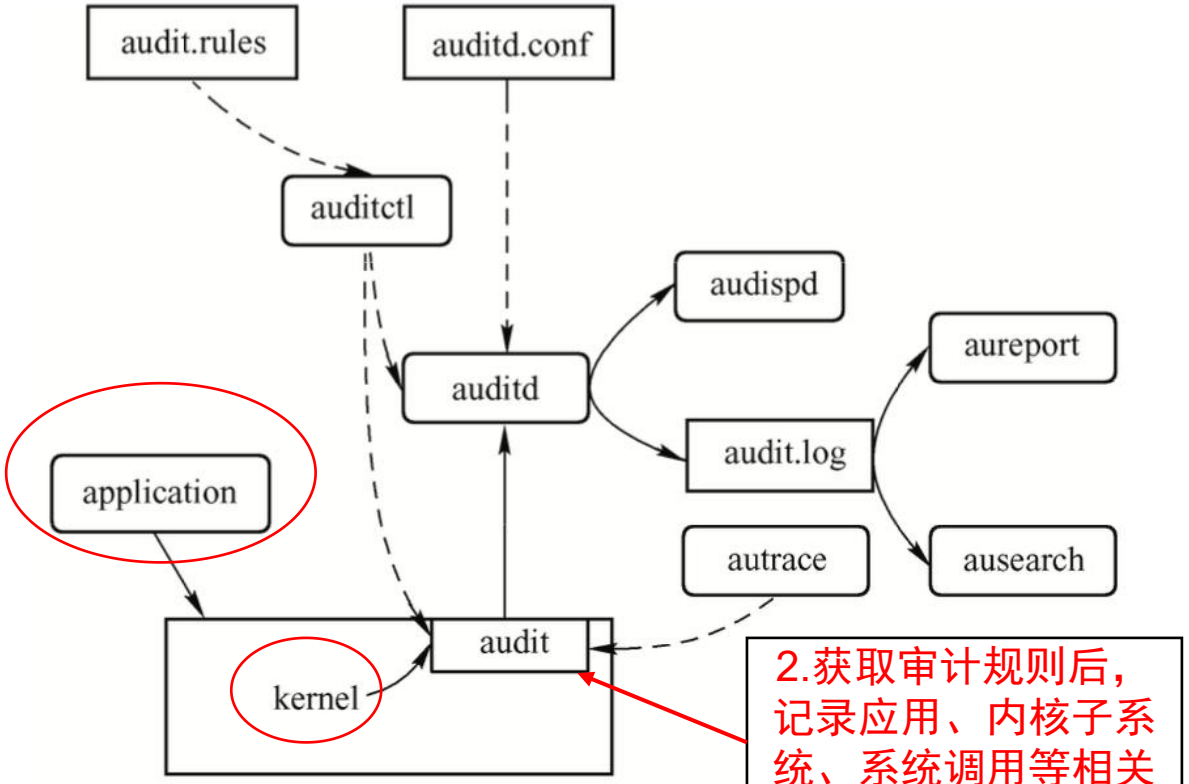
Linux audit架构

不同操作系统审计机制大同小异，以linux系统为例详细讲述审计机制



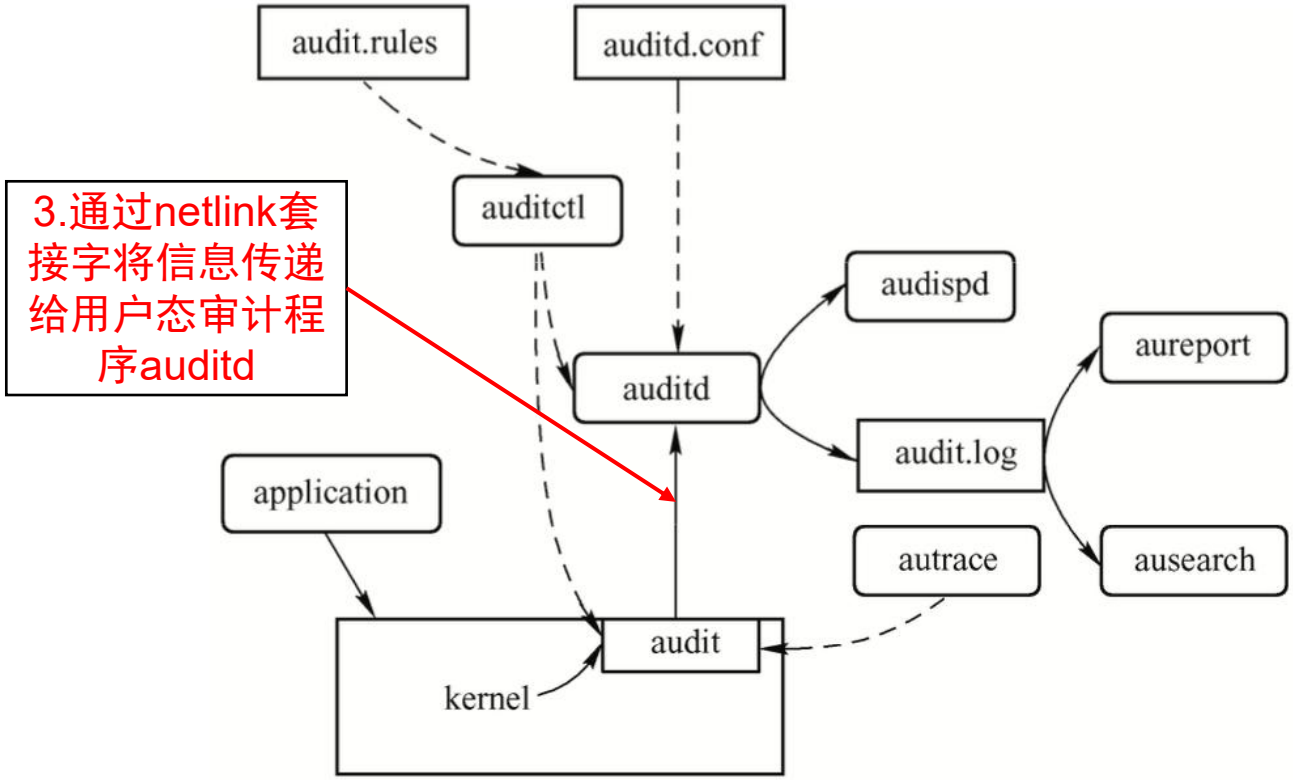
实线代表数据流，虚线代表组件关之间的控制关系

不同操作系统审计机制大同小异，以linux系统为例详细讲述审计机制



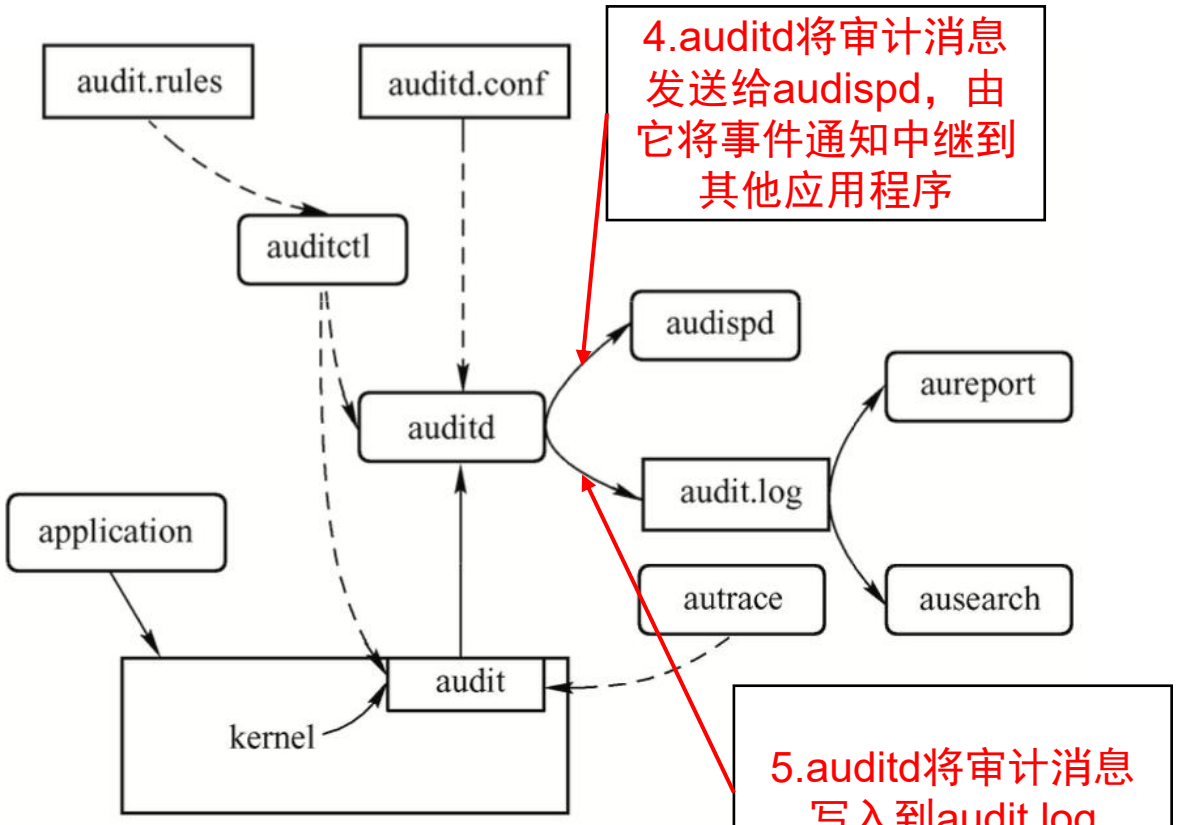
Linux audit架构

不同操作系统审计机制大同小异，以linux系统为例详细讲述审计机制



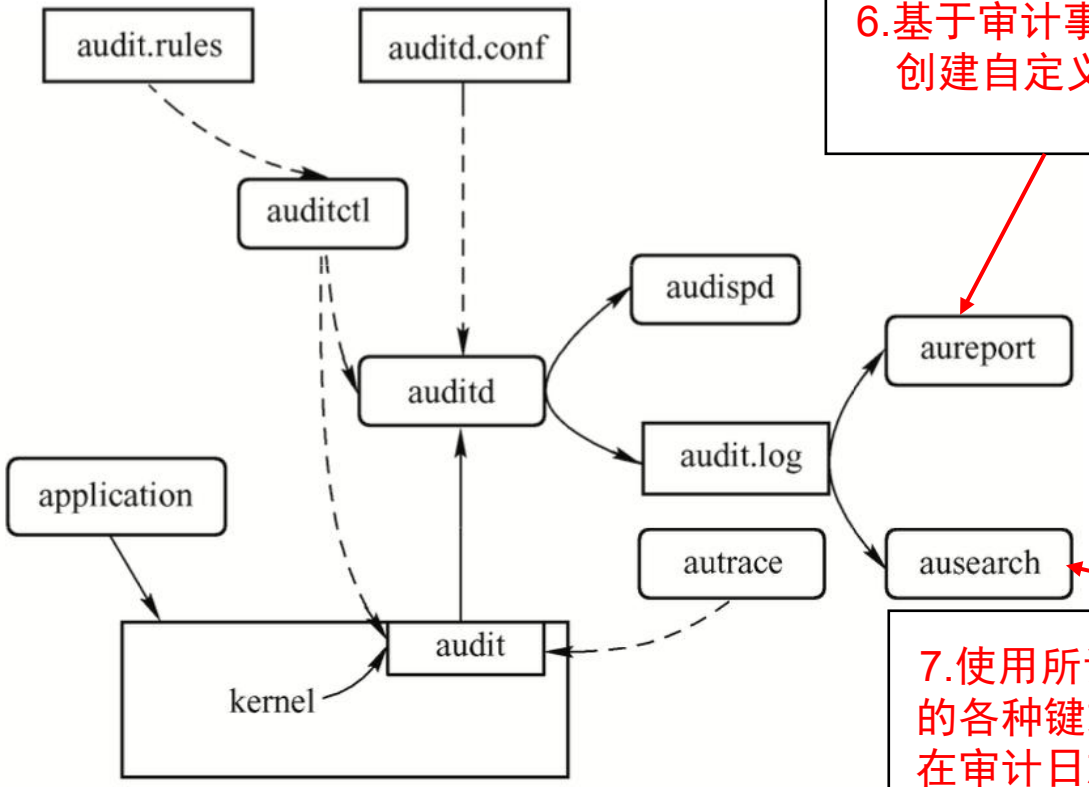
Linux audit架构

不同操作系统审计机制大同小异，以linux系统为例详细讲述审计机制



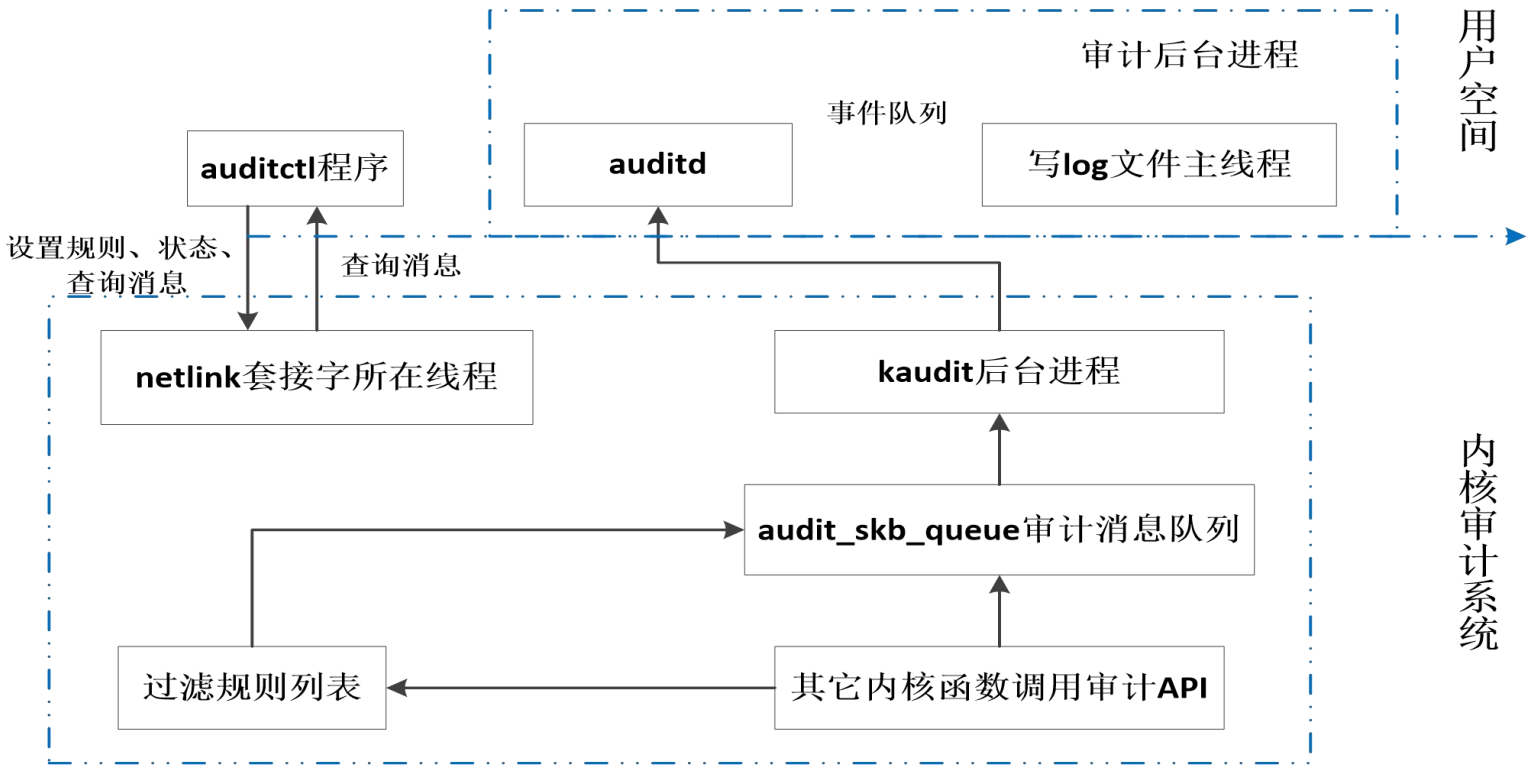
Linux audit架构

不同操作系统审计机制大同小异，以linux系统为例详细讲述审计机制



Linux audit架构

Linux审计系统架构



1）auditctl通过netlink机制与内核审计系统的socket线程进行双向通信；
2）内核其他线程的审计信息由内核审计API先经过规则链表的过滤，然后写入netlink套接字缓冲区队列中；
3）内核线程 kauditd通过 netlink机制将审计消息定向发送给用户空间的审计后台 auditd线程，auditd线程再通过事件队列将审计消息传给审计后台的写log文件线程，由写 log文件线程将审计消息写入 log文件。

内容概要

- 操作系统的背景知识
- 操作系统自身脆弱性
- 操作系统对安全支持
- 总结

内容概要

- 介绍了操作系统安全的背景知识
- 对操作系统自身的脆弱性进行了分析
- 对操作系统中的安全机制进行了分析