[Review 1](#_Toc105005634)

[DAC和MAC 3](#_Toc105005635)

[SELinux 4](#_Toc105005636)

[安全上下文 6](#_Toc105005637)

[操作系统的漏洞 7](#_Toc105005638)

[系统安全标准化 8](#_Toc105005639)

[操作系统内核攻防 9](#_Toc105005640)

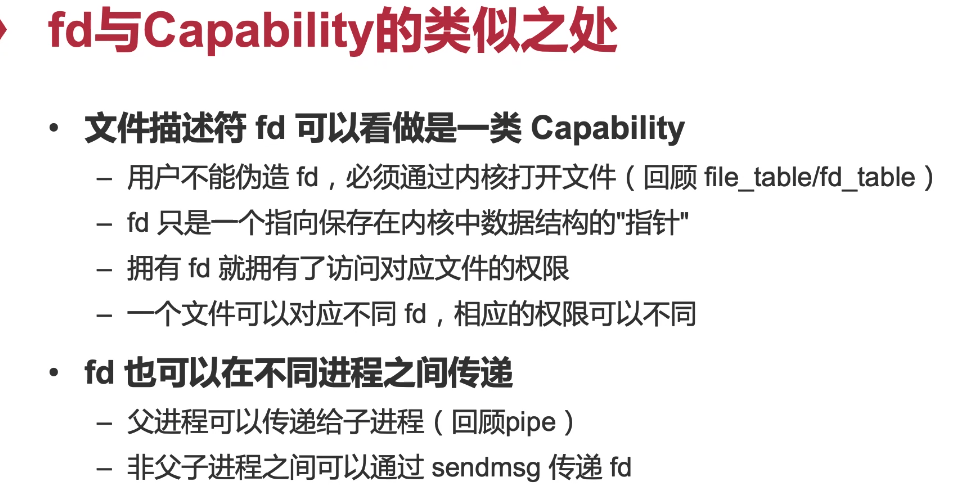
[侧信道和隐秘信道 11](#_Toc105005641)

## Review

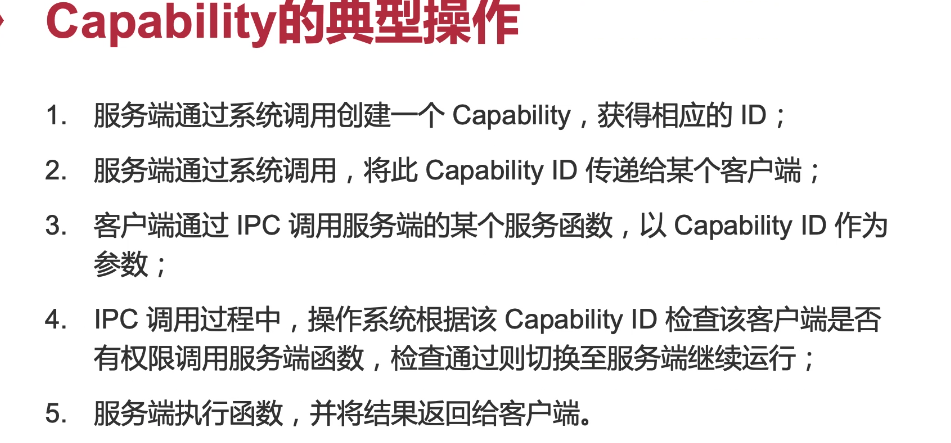


posix没有windows那么细粒度，无法单独为一个用户建一个权限。我们在copy数据的时候，从windows到linux，文件数据可以copy，文件元数据如何copy。windows里有很多attribute，linux里只有9个bit。这一点其实比较复杂的事情。一种方法只copy数据，元数据新建。但是如果是这样的话呢要面临access control不一致这样一个事情。

capability是一个不同与access control的方法：发钥匙，通行证。通行证相当于把你要做的事情写在上面，一张通行证的功能是有限的。和权限绑定。对于FS来说，其实比较简单：读权限写权限。一个进程访问一个文件，如果有capability，就可以去做读或者写。fd其实就是capability方式，可以生成。



最典型一个问题：可传递性。



fd不能够直接传递，它是一个号，fd table index。所以必须要通过特殊的方式传递。linux里面send msg可以传递打开的fd。

Q：这个不是破坏了安全性吗？

A：既然能通过send msg能传递fd，那么就可以传任何数据，这意味着我愿意为别人提供服务。被我提供服务的人也可以让我作为一个代理去访问这个文件。被服务的进程间接的已经拥有了访问这个文件的权限。但既然是我主动把fd给它，在我同意的情况下，另外一个进程有这个权限，其实并没有违反管理的边界。

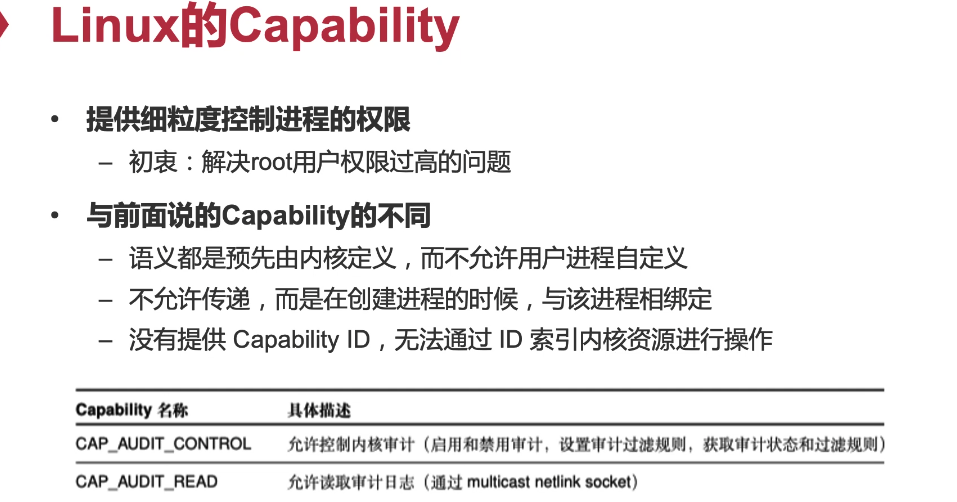
我作为代理和我把权限传给另一个进程是等价的。

在chcore中如何传递fd：一定让os在对方那的fd table中建立一项。对方进程也要做open操作，不是真的做open因为它不一定有权限，os可以不做权限检查。并不会改变这个fd本身的权限——我打开是RO，传过去之后还是RO。但是fd index不一定要相同。

有了这个fd之后其实就比较好的去实现权限的访问控制。

不用去新建group，新建group很麻烦，传递capability更加的灵活。比如不想让对方访问，可以回收。os也可以增加policy：给了别人我就没了或者给了他人我这还有。

linux自己有capability和我们说的capability不是一回事。



细粒度用户权限控制。当它以root身份运行的时候，给它限制一堆开关。不能做什么不能做什么。

比如AUDIT\_CONTROL：是否允许审计。

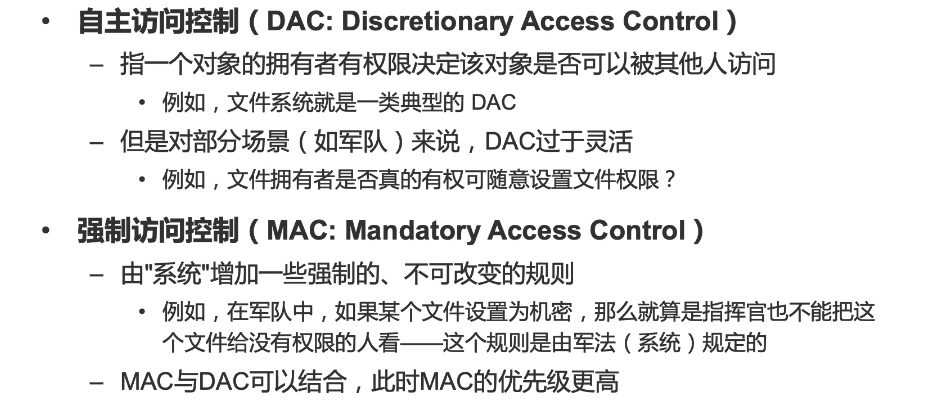
如果被关掉这个权限的root用户想要去做这个事情，就不让你做。

但它不能传递，这是和我们所说的capability很大的一个区别。

chcore进程就是一堆capability的集合。你有这个capability就能访问这个资源。进程是资源分配单位，线程是调度单位。所以一堆capability堆在一起，就形成了进程。

可以把自己的capability送给别人。

### DAC和MAC



linux下chown，chmod（chmod 777所有权限打开）但是真的可以这样吗？真的有权限吗？我是文件的owner当然可以改变权限。

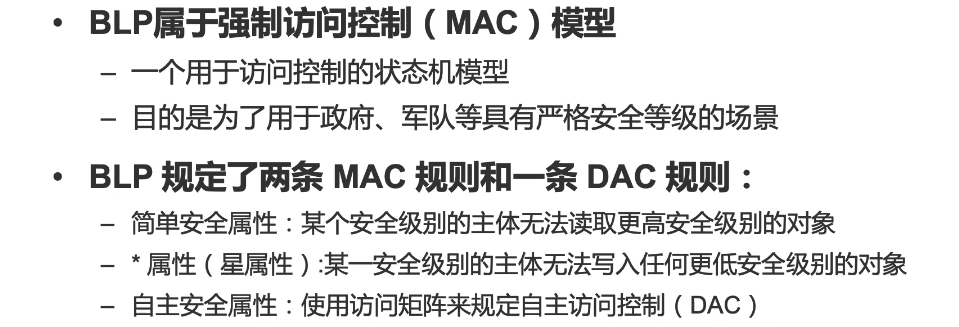
DAC：owner有能力设置文件的权限。

一些场景DAC就不够了：军队中有密集：绝密文件：只有很少的一部分人能看，但是看得人并没有权利把它给别人。所以DAC的问题就是过于灵活。一个将军使用电脑的时候不小心放到一个公共目录里，就狠危险。

MAC：所有用户的基础之上冒出一个system，定了一套规则任何人都不能改变：只要用我这个系统任何人都不能改变。

只要不破坏这个MAC设立的规则剩下的可以DAC，冲突了以MAC为准。

#### Bell-LaPadula



star：四级士兵不能够写入到三级的文件里。

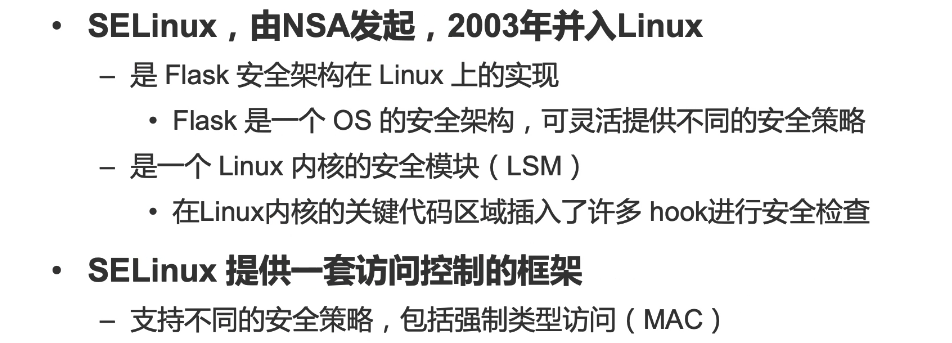
我写的东西只有五级的人能看见。我写了五级的文件我也是看不到的。

> 之前用ftp上交作业的时候upload only，防止大家下载别人作业抄。

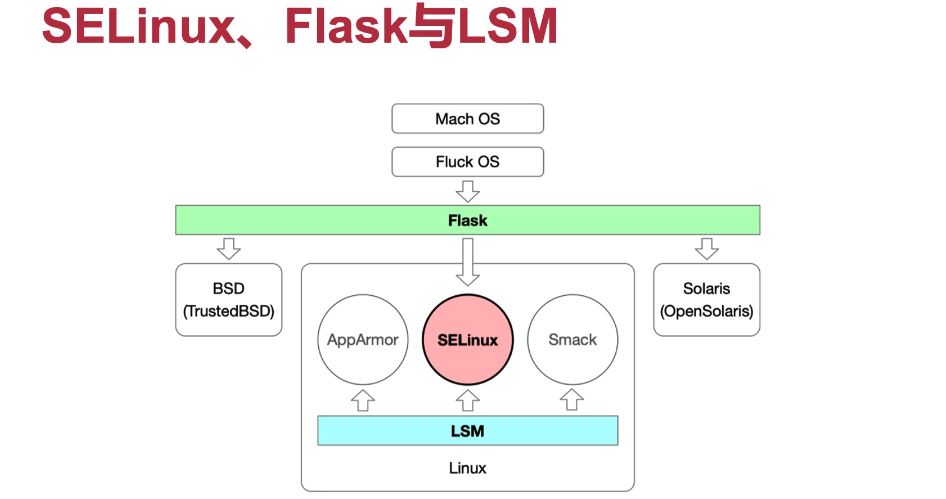
第三条：除了前面两者之外就可以用DAC。比如都是三级的就是可以设置owner才能访问。或者777之类的。

（设成777之后二级的人也看不了，因为违反了第一条MAC

## SELinux



Linux里面已经有了SElinux的能力，只不过平时不enable。

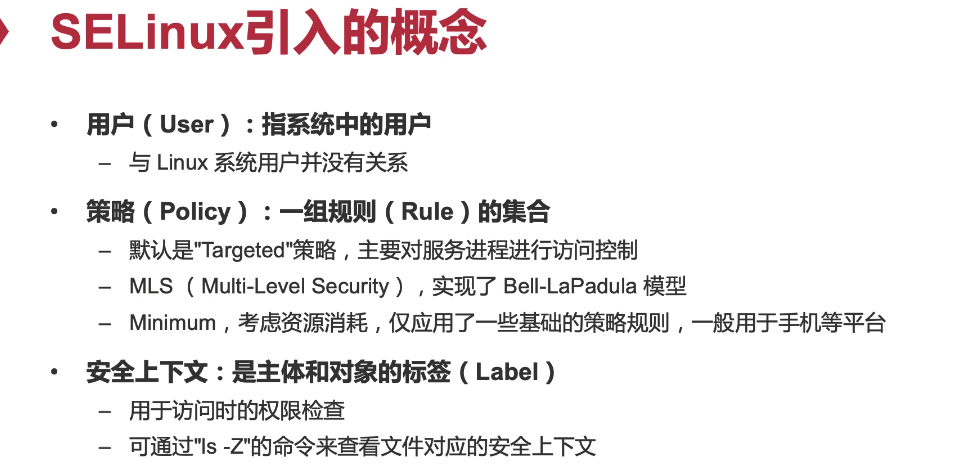


这框架与OS解耦。为了支持Flask这套东西，Linux提出来LSM（内核安全模块）—— OS发展逐渐插件化（insert mod）插进来。security提供了一套新的接口，埋在内核很多地方的一些check。

e.g.：你要打开文件open里加上MAC，应该在哪里加if else。

应该要在open里加一个hook，马上要打开这个文件的时候检查一下有没有一个函数指针要在这里运行一下。并且根据函数返回结果做一些事情。LSM就是kernel里要插很多这种hook。开发者要去实现很多hook。然后作为一个模块load到kernel里面来。还可以加时间判断，GPS判断等等等等。可以让整个安全的policy变得非常灵活。kernel埋的hook更多了，hook还能做什么呢？

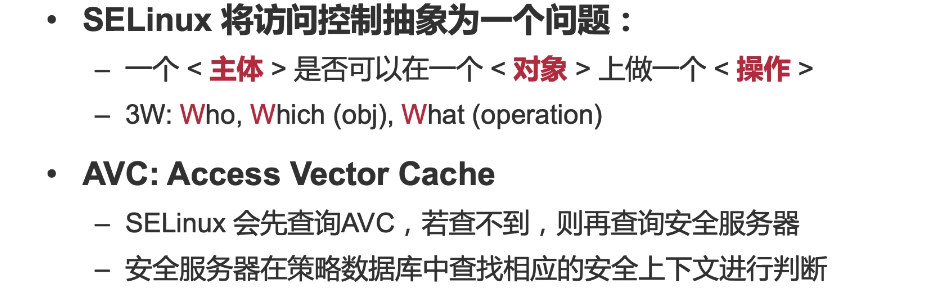
> 内核结构一直在变化，让机制和策略分离。但这绝对不是随意运行。你必须要对hook非常了解才能去执行一些功能。



允许用户写很多的policy。同样需要提供基本的概念：用户（User）

有标签的用户：policy：一组rule的集合。今天想用一组policy，把这个模块load进来，明天想用另一种，load另一个。

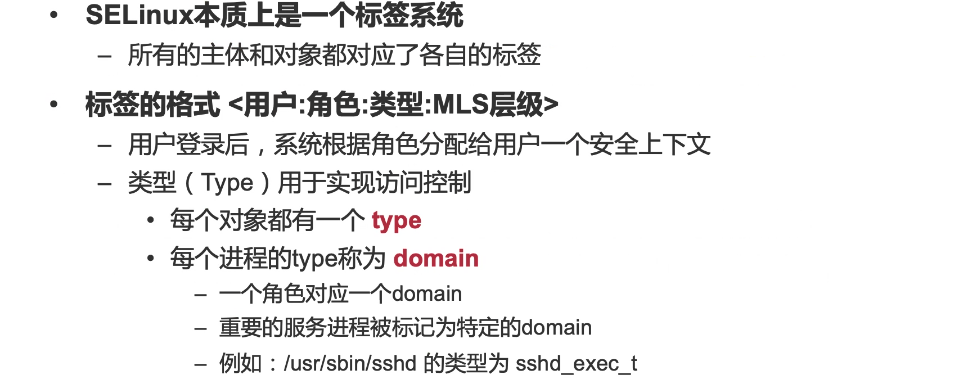
安全上下文：label

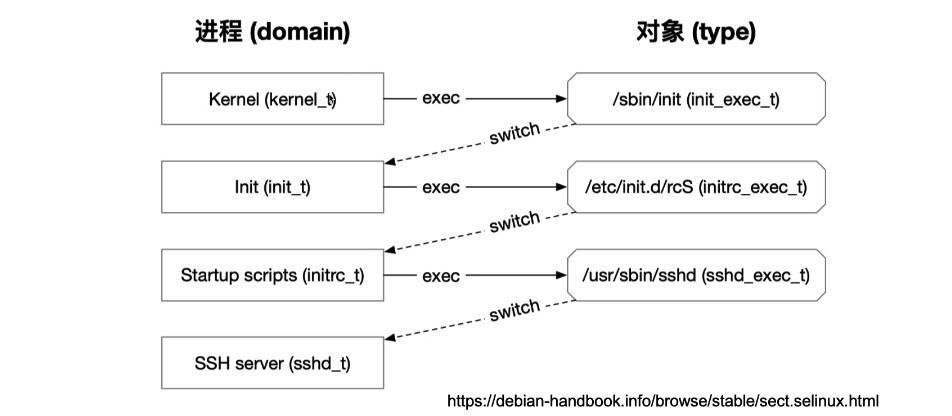


你有这样一条policy之后，当你想做一个事情的时候，就会先去找这样一个policy。这就是非常general的一个实现。

有专门一个服务器保存这样的规则（一台服务器可以服务100个机器）。当碰到一条hook的时候去服务器拿，本地可以做cache。

### 安全上下文





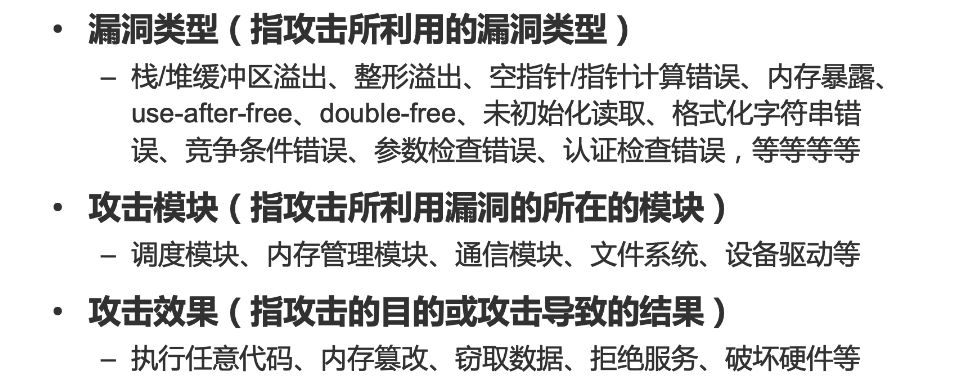
如果一个sshd\_t进程被攻击了，变成了sh了，它的标签也没有改变。



应该只能放在admin\_home\_t目录下。

cp规则：在www/html下创建文件的时候默认会打上http\_sys\_content\_t的标签。mv的话标签会被直接拷贝。

## 操作系统的漏洞



stack/heap overflow

整形溢出：two complement

空指针引用：整个OS可能就panic了。之所以有指针计算错误是因为OS大都是用C写的。C中指针计算还是个undefined behavior。

double free：可能你free掉了分配给另一个人的space。

free的时候并不会清0。

race condition

认证检查错误

攻击模块最常见的就是内存。

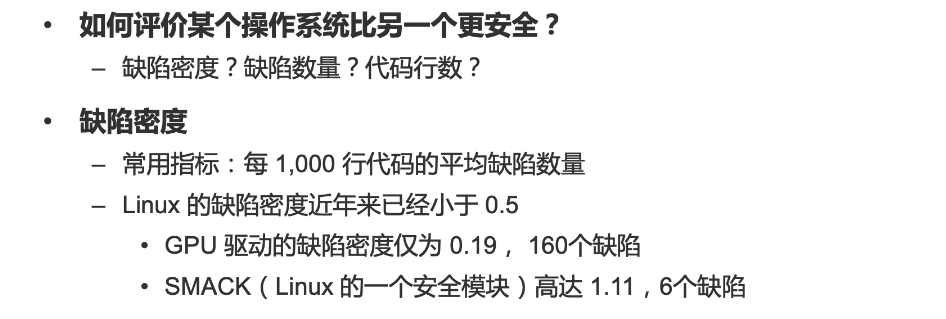
设备驱动占了所有问题中的50%以上。

DoS：crash

窃取数据就很危险

执行恶意代码肯定不能接受

### 系统安全标准化



千行代码bug数

GPU驱动和SMACK哪个更安全



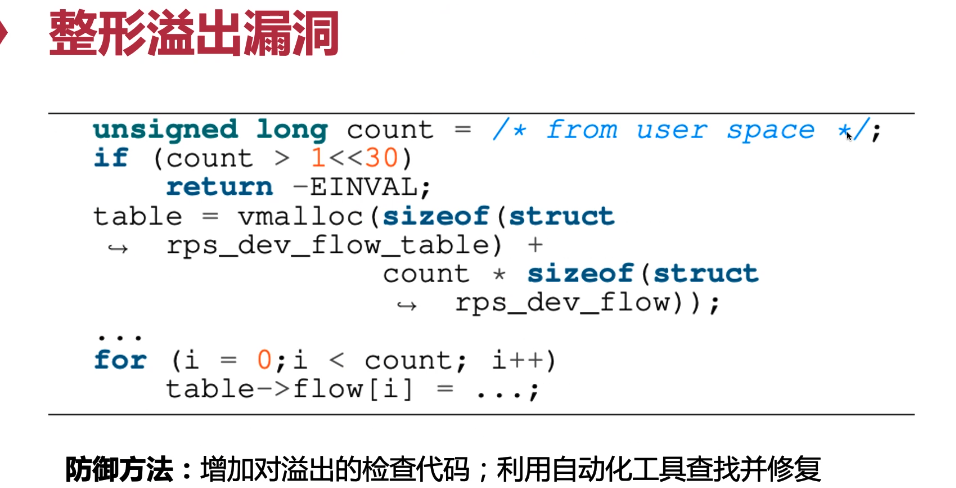
之所以linux CVE2000多个，是因为它开源，谁都可以去测。

windows不开源。

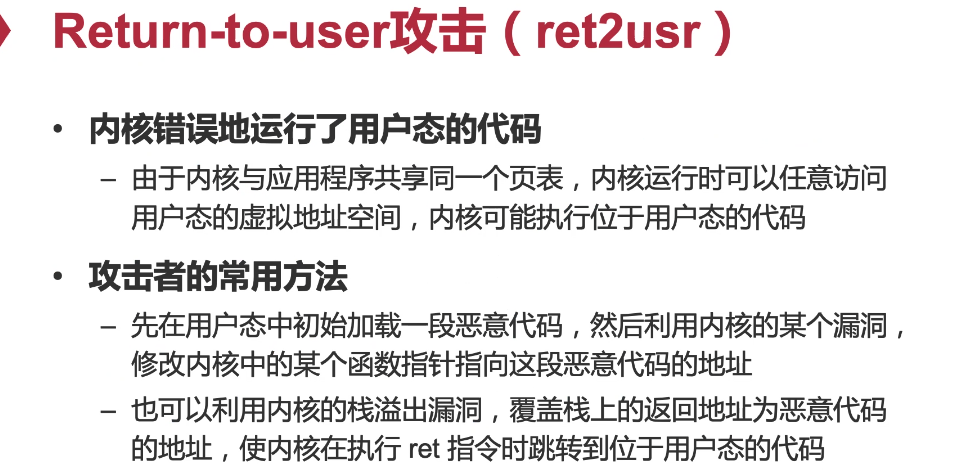
这些因素考虑进去也不能用缺陷数量来判断。

找到缺陷之后会往CVE提交，拿到一个CVE编号。

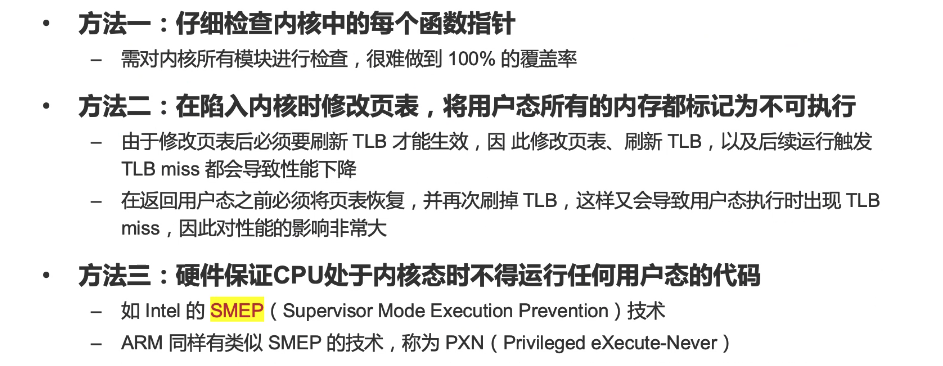
### 操作系统内核攻防



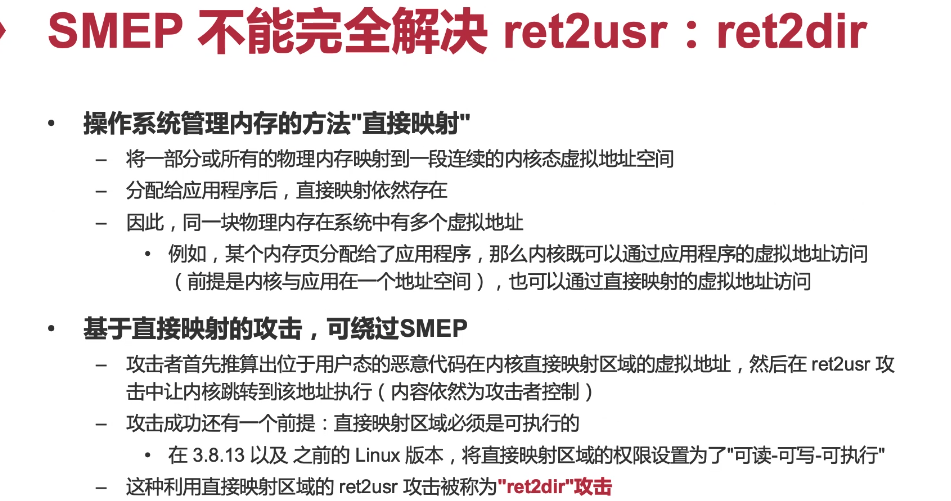
整形溢出至今都是kernel中很严重的漏洞。



防御方法：



改页表要flush TLB，下次跑用户态应用就会大量TLB miss，很伤。

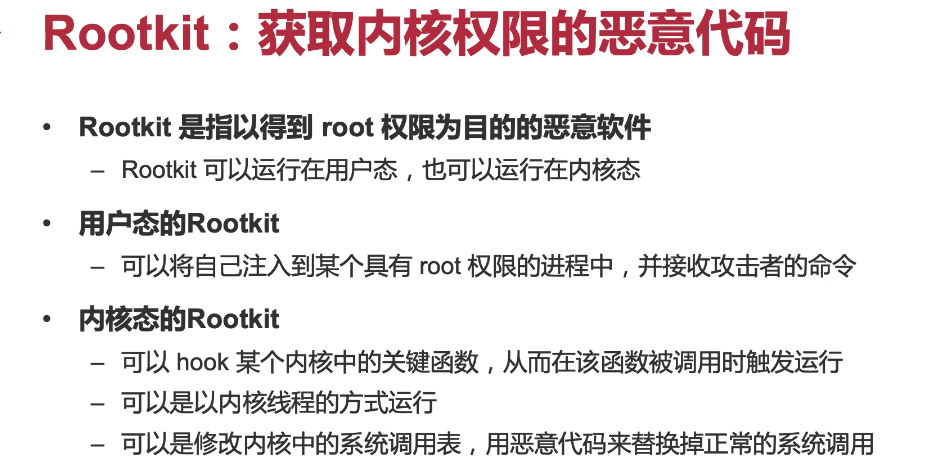


buddy system它就是这么做的，直接把整块memory映射过来，自己管理。当我用户态应用程序调malloc的时候是额外的一块页表，它可以不映射到任何内存（on demand paging）

用的时候触发page fault，向OS要一块page（从buddy system里抽），但是kernel映射并没有取消，kernel还是能direct mapping的方式访问这块。同时这块地址又被映射到了用户态，第一部分是用户页表映射的，第二是kernel内核direct mapping映射的地址。

假设用户态恶意程序知道你有direct mapping。它知道了这两个虚拟地址，它就可以让用户ret到这块区域，这块区域并没有违反SMEP的规则（并没有跳到user level）。

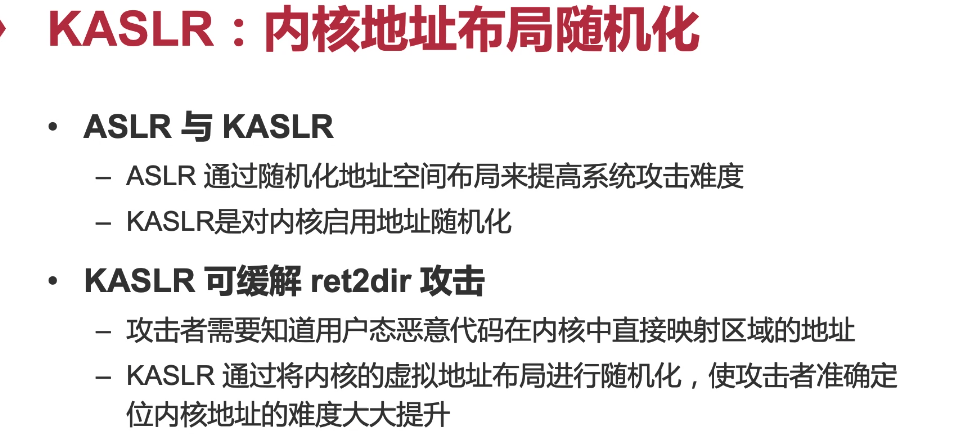
在direct mapping把这些页都标记为不可执行。



所谓的进程注入就是这个目的。这是用户态的一类，用root的高权限。（安全的人不喜欢root，rootless kernel）

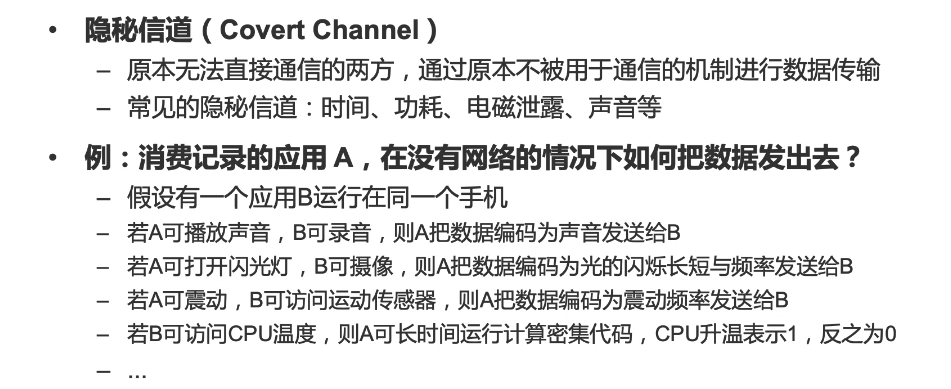
hook比如把write syscall给hook了，在内核里就是一个指针，sycall table。

攻击者通过一些方法，把这个syscall table给overflow掉，然后指向自己写的代码。man in the middle。然后内核态的rootkit可以通过改变函数指针，通过hook的方式去调。改变hook的方式是最常见的。启动内核线程，但是容易暴露自己。但是hook函数指针，然后再用户态主动触发这个syscall，是很难被发现的。



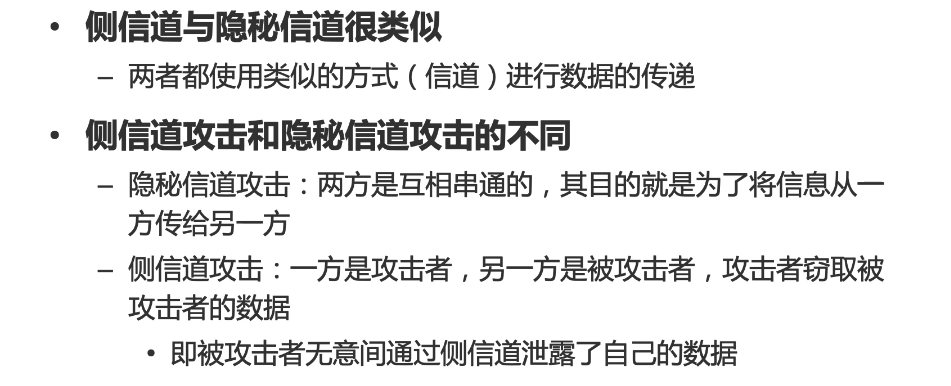
很多攻击都需要知道一些指针。如果我们能够把地址空间随机化——内核虚拟地址布局进行随机化。运行起来之后无法随机。现在有很少一部分是在运行的时候做随机（性能overhead比较大）有可能要停顿一段时间。overhead比较高。

### 侧信道和隐秘信道

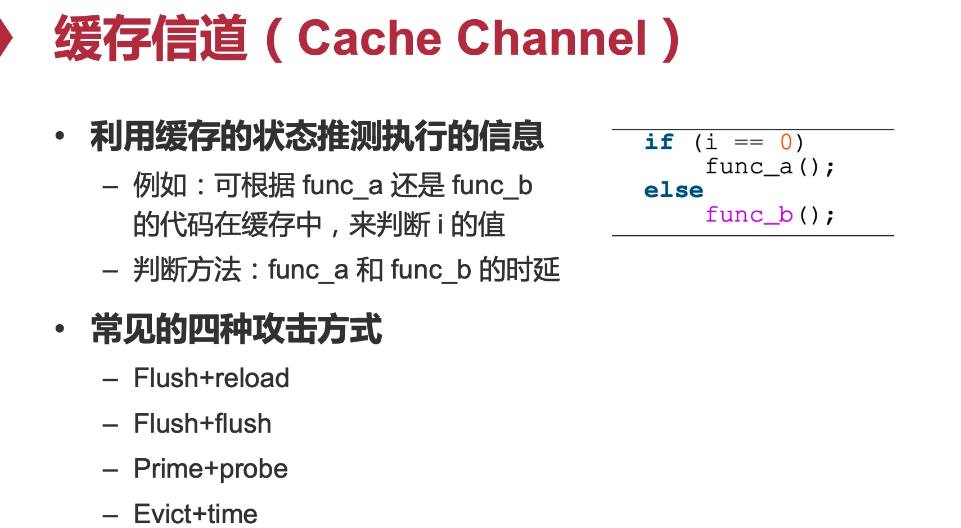


把内存变成WIFI，通过对内存的访问，形成电磁波，这个辐射，假设有一台精密的仪器，能获得这个电磁信号的变化。

侧信道和隐秘信道其实非常类似，隐秘信道攻击，两边是串通的。



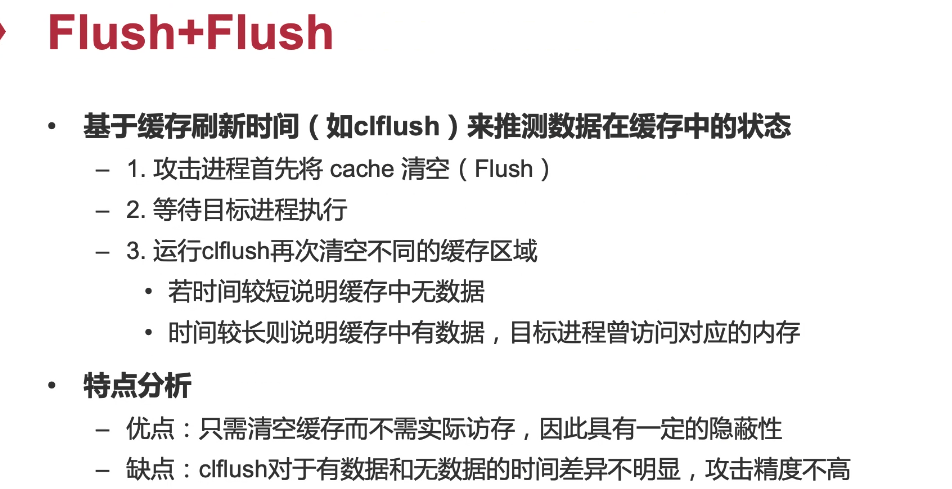
侧信道一方是攻击者，另一方式被攻击者。



我要去猜i是否为0。

先把这四行代码运行完。再去运行func\_a和b。



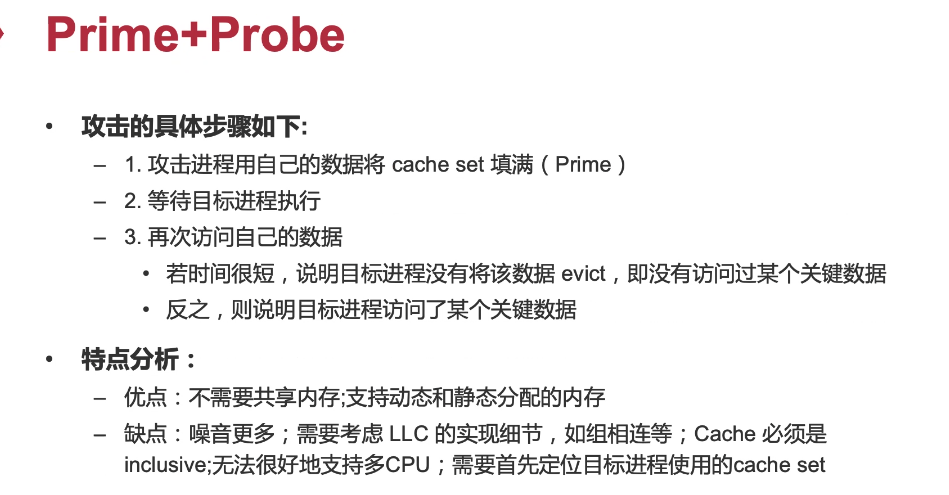




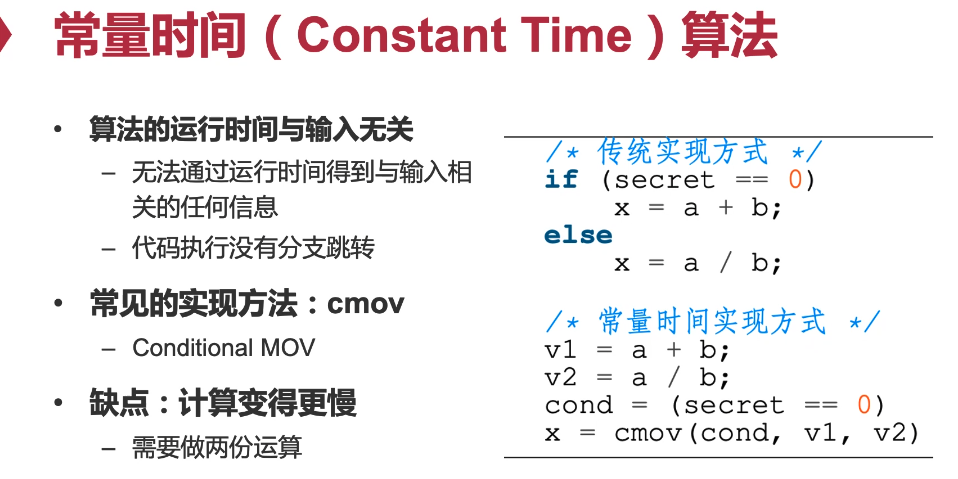
inclusive：在level2 cache一定在level3 cache。

第一次cache miss肯定在level3

exclusive：进了上一层这一层就可以删掉了。



不需要共享内存，但是得知道内存怎么映射到cache。cache肯定共享（有规律）。通过这些evict的信息知道目标进程访问了哪些数据。



加法一个cycle，除法一个cycle肯定搞不定。

把两者拉到一样慢就好了。cmov：condition move