[5-12轻量级隔离 1](#_Toc105353680)

[review 1](#_Toc105353681)

[是否有轻量级的隔离 3](#_Toc105353682)

[CHROOT 3](#_Toc105353683)

[Linux Container 6](#_Toc105353684)

[SFI：软件错误隔离 12](#_Toc105353685)

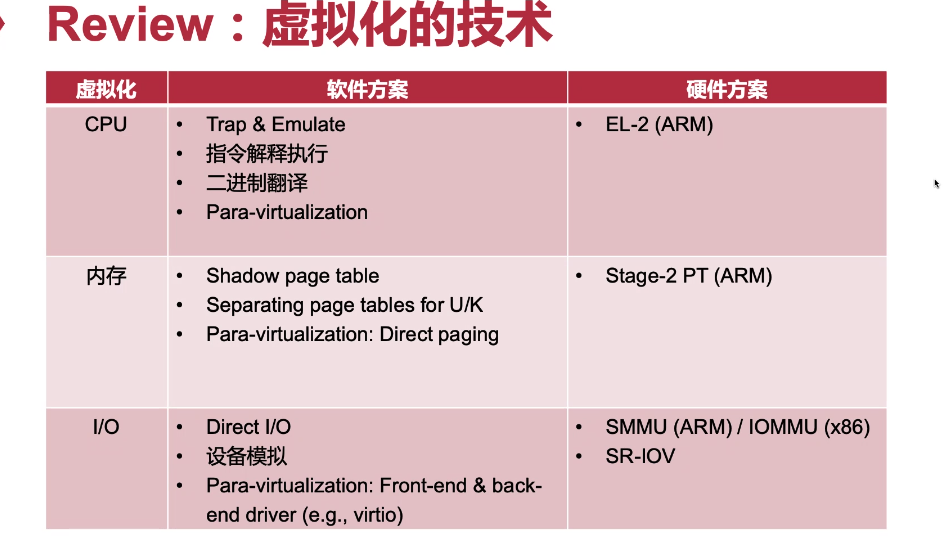
[CFI：控制流完整保护 14](#_Toc105353686)

[XFI : SFI+CFI 16](#_Toc105353687)

# 5-12轻量级隔离

上完虚拟化之后大家肯定感受到这一点，虚拟机之所以很实用，因为它可以把整个系统隔离开，让整个软件栈跑在一个隔离的环境里。

## review



trap and emulate是最重要的，第一点是在什么地方拦截一道，在什么地方设好关卡。比如我们在trap的时候trap特权指令（在特权指令设了关卡），在面向对象编程的时候，类里面暴露字段的时候可以直接暴露（public），但这种设计模式不是特别被推荐，推荐的方式是setter和getter，它又什么好处呢？当你用get和set的时候读取field的时候拦一道。当然你可以写的非常简单。但是你也可以加更多的逻辑，比如检查要set的值。当我们去执行特权指令的时候截获到。最体现在设备模拟里面，截获所有的io操作，比如要控制显示器相关的寄存器，可以截获然后最后画到一个特定的窗口里面去。那么一条对于显示器的这么底层的指令可以转化到canvas这么high level，屏幕上某一个窗口的像素。trap and emulate一下子变得非常软件定义。虽然放在了cpu里面，但它和io，内存等等都是非常相关的。比如shadow pagetable。

（漏2min左右）

实现已有接口叫做虚拟化，加一个新的接口叫做抽象。假设你要linux里构造一个win的运行环境，这是非常复杂的一个事情，win的驱动框架太大了，这就要逆向win的很多东西，然后搬到linux里面去，这就变得非常不可行。

为什么我们可以在四节课就把虚拟化讲清楚，因为它这一刀拦在system ISA这一层，我们只要软件解决这些ISA就可以了。

我们为什么还要讲软件的方法，比如shadow pagetable，因为讲它可以让大家更好理解页表是如何工作的。

direct paging是破坏封装性的，让vm知道自己跑在vm里面。这种暴露本身会导致一定问题，它可以看到HPA，它就知道内存上的布局，这就会导致一些攻击：rowhammer。我们在做这个虚拟化的时候，看上去没有什么关系，比如direct paging，正确性上面没有影响，但是暴露的信息可能会导致侧信道啊，之类的风险。

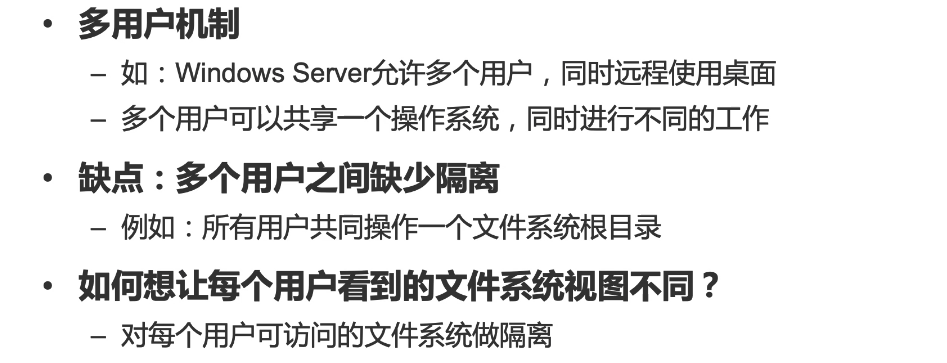
对于整个虚拟化来说，今天硬件方案很成熟，所以基本都是走硬件方案，可以做到允许哪些特权指令trap，哪些可以在内部解决。比如syscall，能不能svc的时候能不能直接跳到vm里面的异常向量表。也可以vm exit出来，做一些事情，再回去。这个跳出来有什么用呢？可以用这个方式记录调了哪些syscall。VMM可以把vm内部调的syscall重新实现，也可以直接bypass给vm的kernel，也可以ban掉一些危险的syscall。所以今天cpu虚拟化已经非常成熟了，两级页表虚拟化内存也非常成熟，但是访存次数一下子会变多。可以通过增加cache（TLB）来解决，但是第一次的cold miss无法避免。还是有人怀念shadow page——只要走一次页表就可以了。vmware内部有人做过量化的研究。硬件也在不断加速，比如tlb更大，用硬件也肯定比软件实现起来简单——实现相同的东西，越简单的东西越会存活下来。对于IO来说，硬件方案也比较成熟，但是大部分用的还不是硬件的方案，比如要改网卡，网卡本身能被多少个虚拟机用也是有限的。最早的SIOV只能虚拟化出7个虚拟网卡。但是如果不插这个网卡，没有siov这个功能怎么办？假设有一台vm从一台主机动态迁移到另一台主机，这两台主机上装的网卡还不一一样，那就没办法。所以IO大家还是用软件的方法，virtIO，虚拟机只要装一个frontend就行。软件的方案也没有数量的限制，只要内存够，虚拟机迁移也会更简单。这些不同技术各自有各自的优缺点。



底下物理硬件，上层都是虚拟机。甚至可以热迁移。fork bomb对于虚拟机来说就是一个死循环，但是它不可能把所有CPU都占完。线程是用来实现vCPU的，无论虚拟机内部无论fork也好，对外它也只有vCPU个线程。虚拟机的隔离优势是非常明显的，它可以很好的适应，但是虚拟化毫无疑问也有问题：性能损失。但现在虚拟机已经做的非常好了，启动可以达到10ms这个级别。

尤其是两层都是linux的话，会造成性能浪费。这样意味着cache就有两层，我的调度有两层，然后导致我的network stack有两层。这种操作就会导致系统的资源被浪费掉。这就是隔离带来的代价。

## 是否有轻量级的隔离

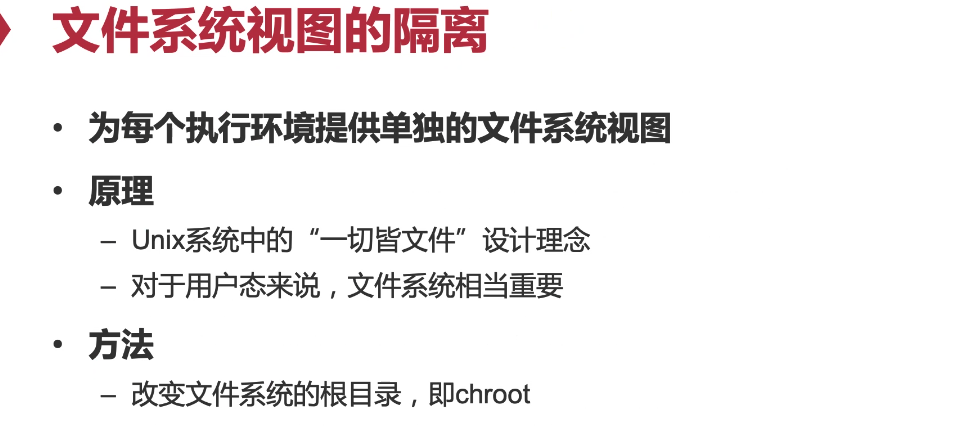


windows server它的缺点就是多个用户之间缺少隔离。

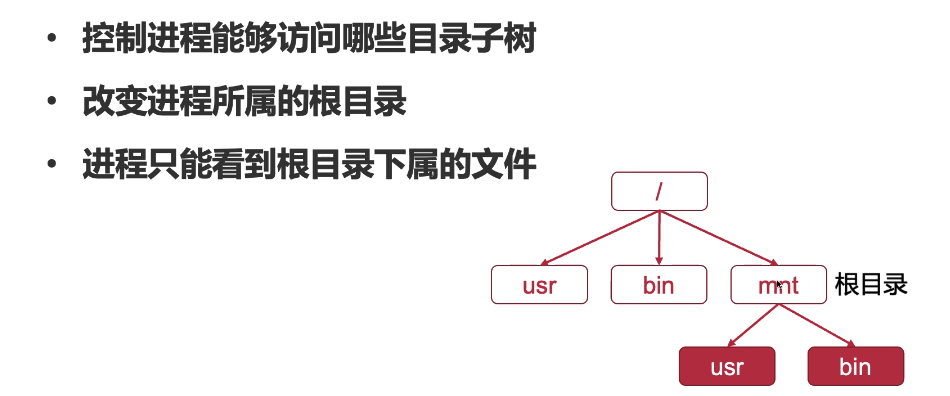
大家看到的C盘都是一样的。

### CHROOT

不能多个用户共享根目录。

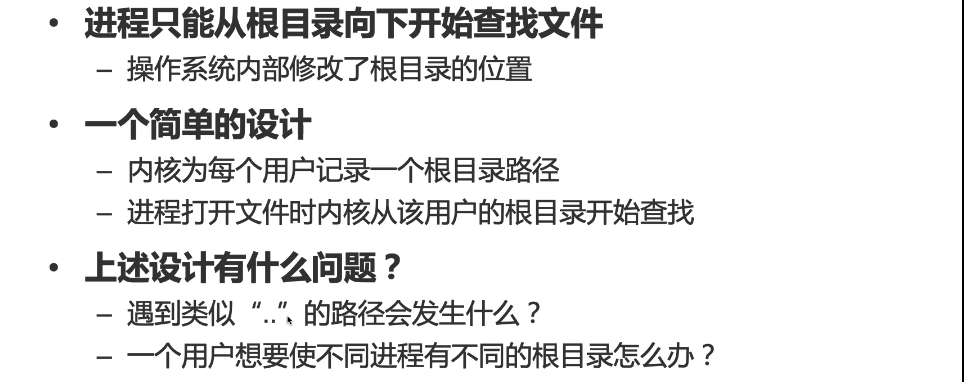


我能改变用户的根目录我就能把用户限制在很小的范围里面。

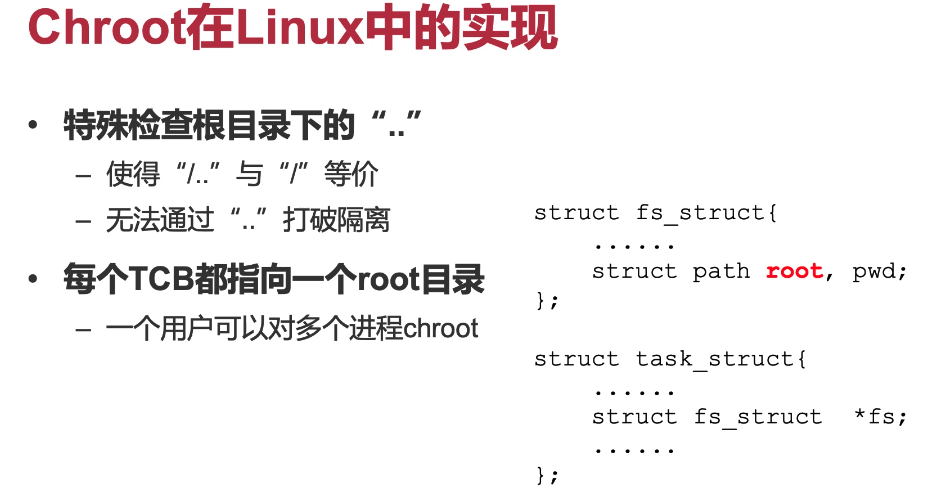


交叉编译的时候就非常有用。

这就是namespace的隔离：



`..`是hard link。

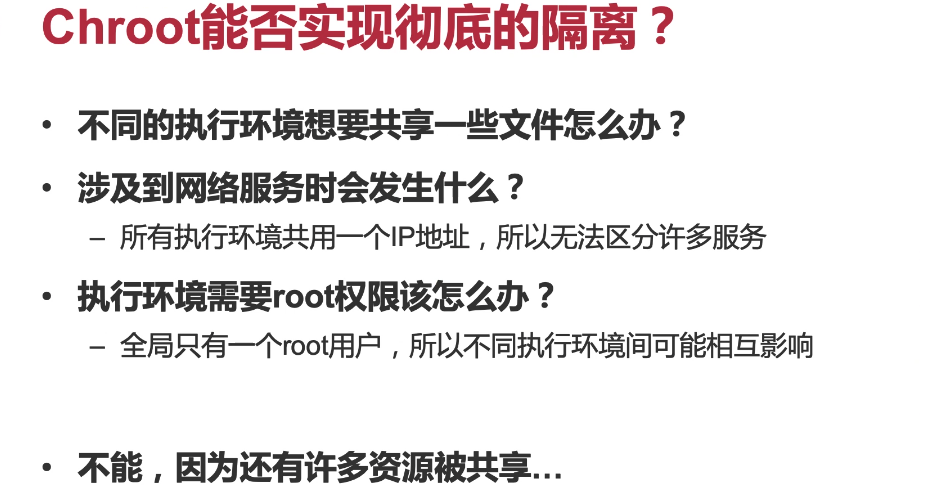




一旦root用户调chroot就跳出来了。为了保证chroot有效，要用set uid消除目标进程的root权限。chroot本质上就是通过namespace。

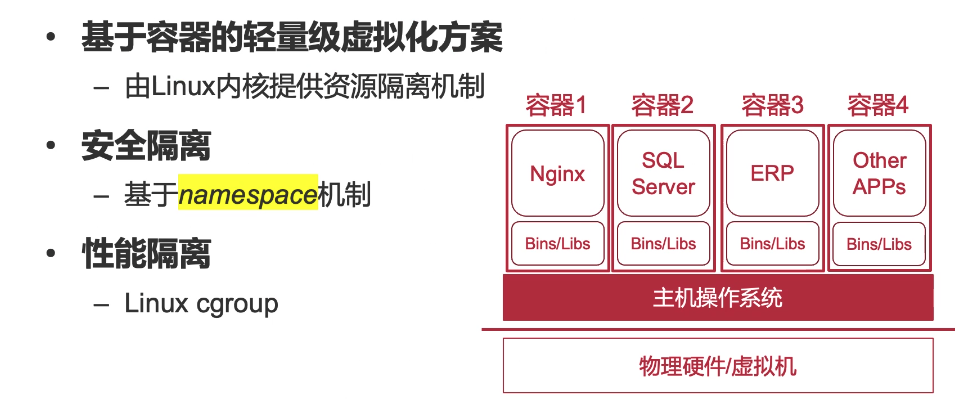


假设可以通过inode访问就可以绕开。



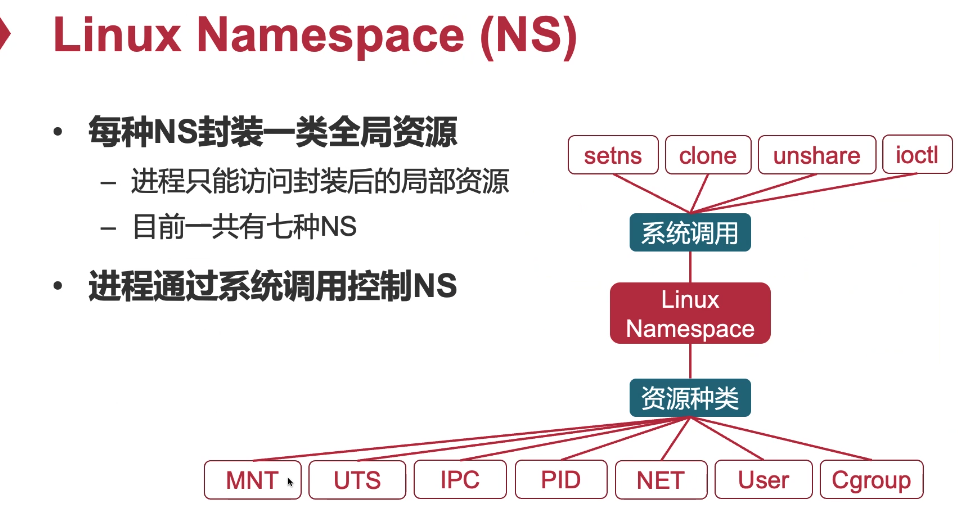
隔离和共享——双刃剑，涉及到网络服务的话怎么办。全局只能由一个root所以我们说chroot接踵而来有更多的问题。针对chroot的很多问题。

### Linux Container



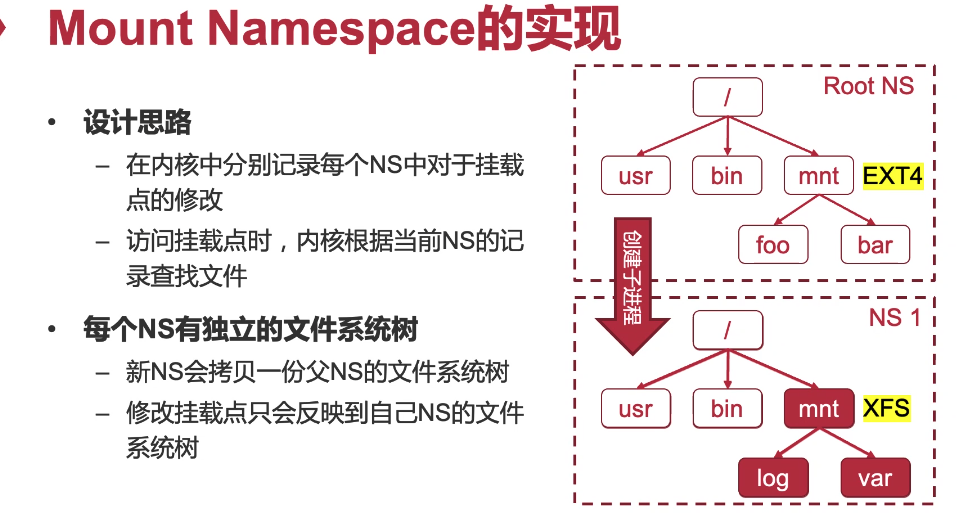
它是由linux内核提供的一个机制。它这个拆分拆的也很痛苦，kernel之间千丝万缕的联系。

先来看namespace：之前chroot改的是文件系统的namespace。



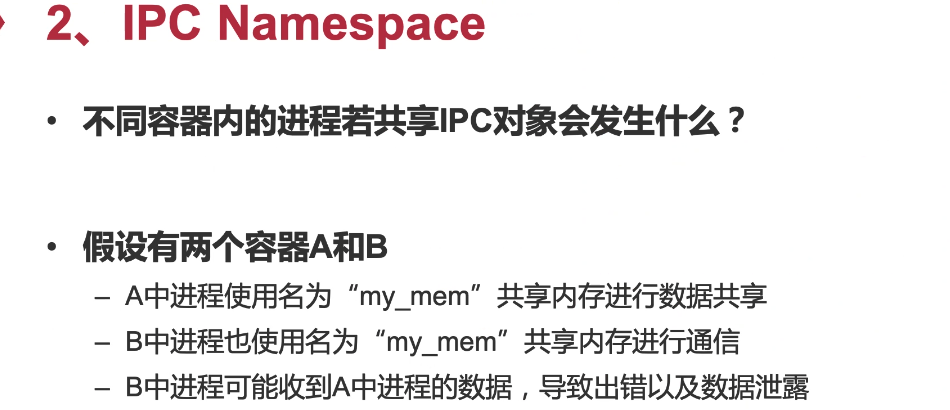
每一个namespace能保证的点是进程只能访问封装后的局部资源。



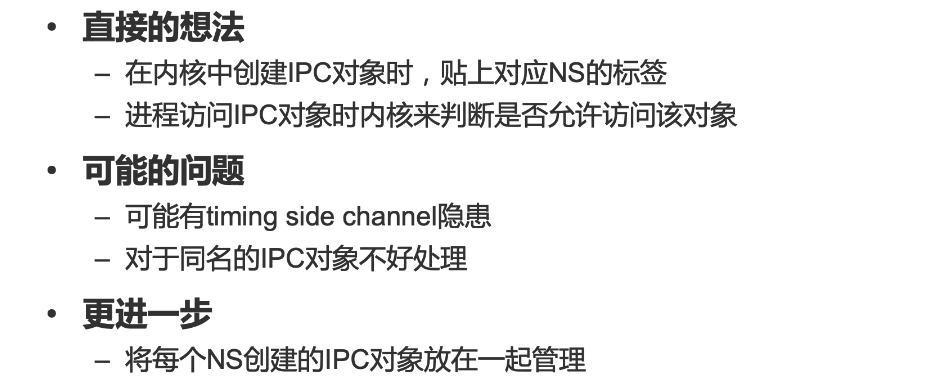


每一个容器都会有自己的mount namespace，它会记录你是怎么挂载，host也看不见。有了这个mount之后整个namespace就多了一份copy。

只能改kernel。

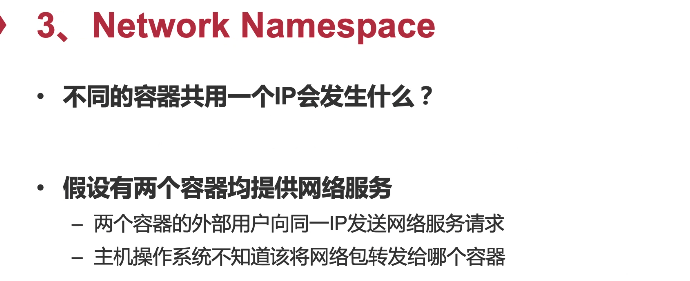


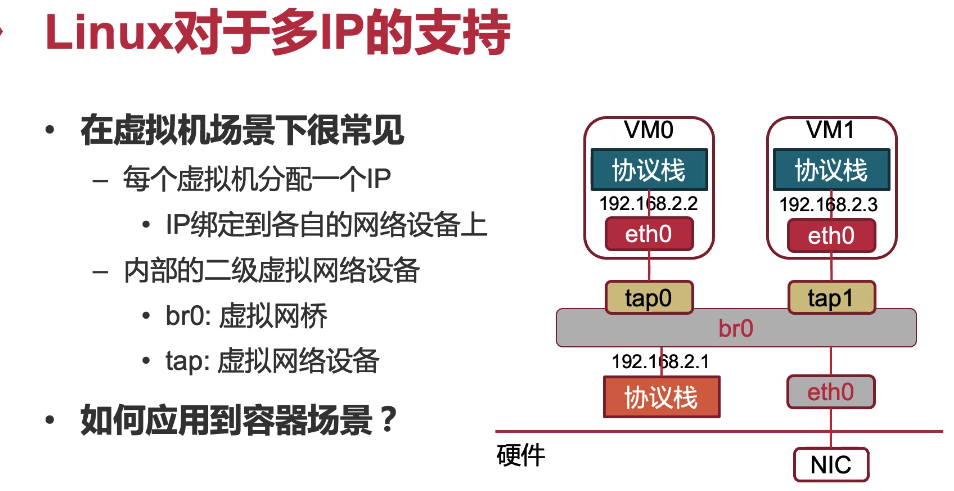
开始的时候内核有一个IPC对象，如果我们要两个进程之间共享内存的话，可以给这个共享内存一个名字。内核来维护这个shared memory。这里就有一个命名空间的问题。这里是全局唯一的。假设容器A有mymemory，容器B也有mymemory，是容器内部有两个进程要共享，不想容器间共享。一个简单办法就是不要用同样的名字。暴露了一个信息——有别人在用mymemory这个IPC的namespace。更好的一个操作是把这个mymemory啊，不同的人有不同的namespace，你是你的我是我的。



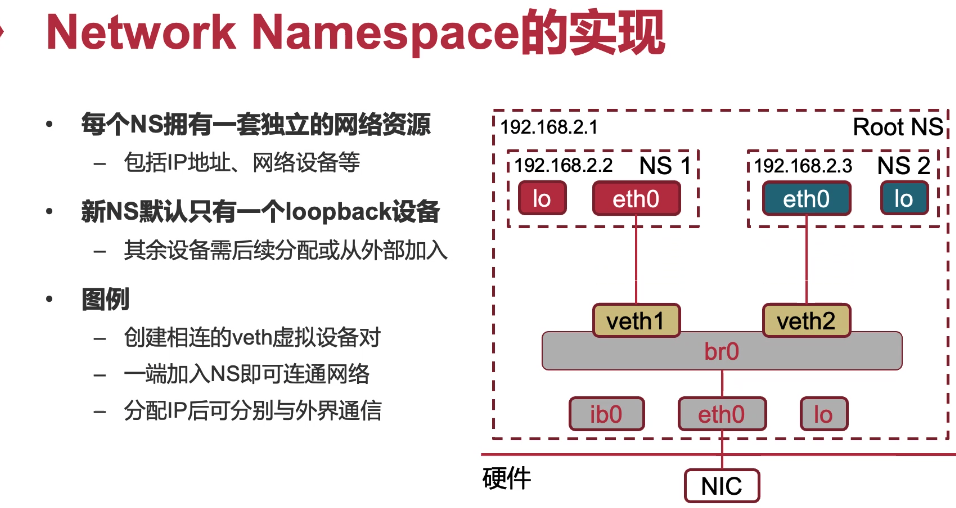


名字本身也是一种资源，所以当在内核里谈资源隔离的时候我们要考虑的就不仅仅是硬件资源的隔离，所以network也要有namespace，不同容器不希望公用一个ip。

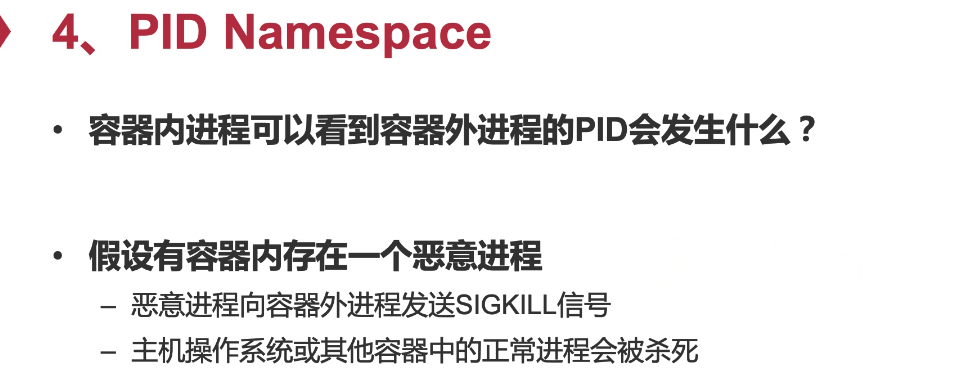




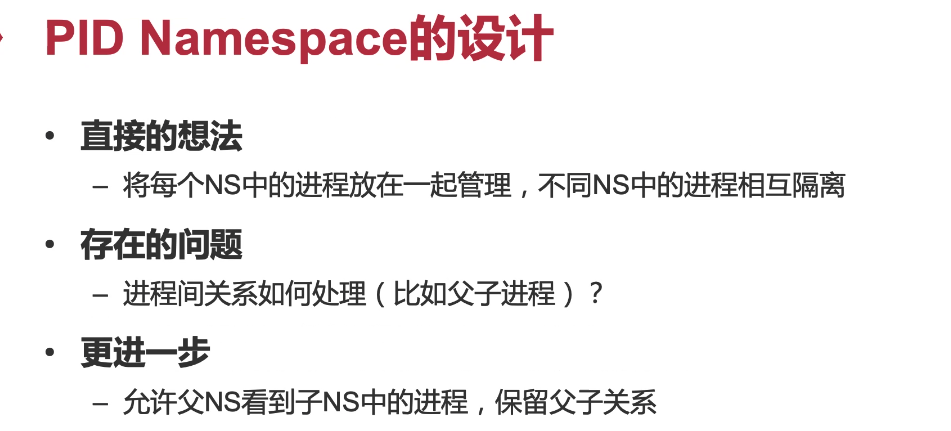
虚拟网桥，虚拟IP，虚拟的网络设备都可以在linux内核里面做虚拟化。



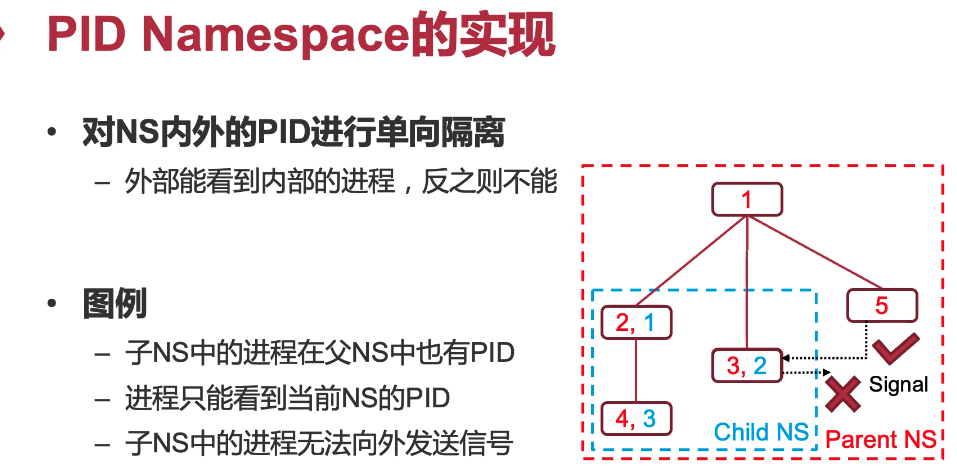
在虚拟的网桥上创建出虚拟的以太网，一端加到容器，一端加到主机的eth0。



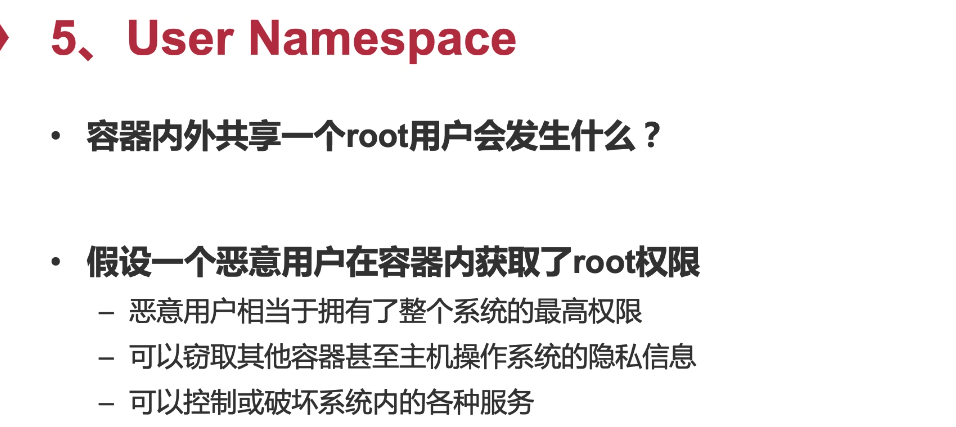
pid全局唯一，一种简单的方法就是在pid前面加一个前缀，比如一号容器就是10000x，二号容器就是20000x，为什么host上还要有一个名字呢？因为在容器里面新建了一个进程，真实的host里也会有一个进程，容器无法防御fork bomb。在容器内部创建了一个thread，host上看确实多了一个thread，因为TCB是由host的kernel来控制的。



我们依然要去处理一些进程间的父子关系。有一个parent NS，也有一个child NS，外面能看到里面的进程，里面看不到外面的进程。

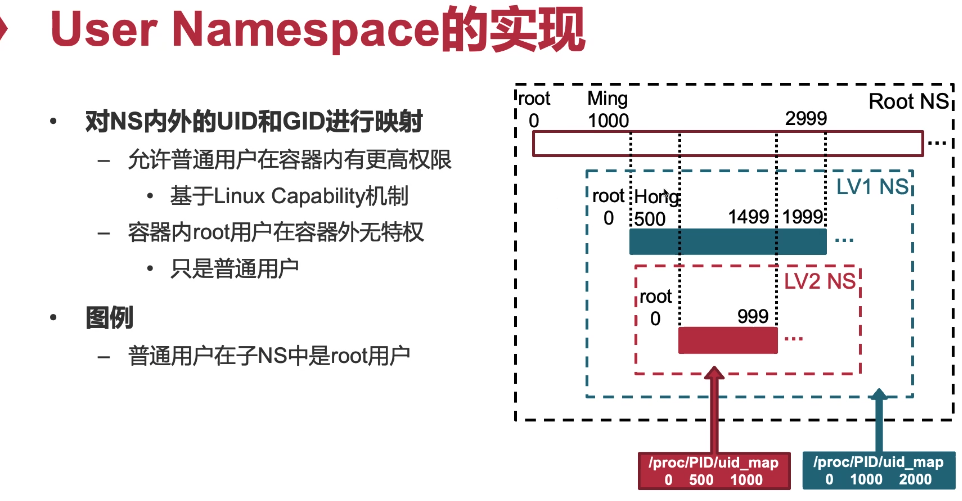


在全局角度依然能看到进程的hierarchy关系。进程不能是孤立的，因为linux里进程需要有人去wait来回收。我们并没有因为namespace的引入打破原来的进程间的关系。



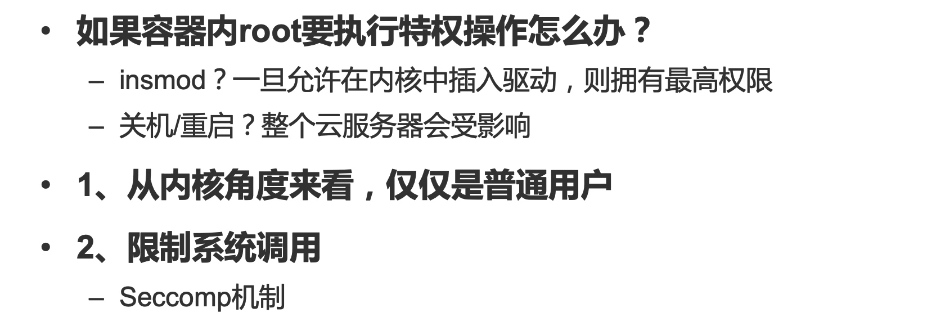
一种方法里面不给你root，但是里面有可能就是要去执行一些需要root才能执行的操作。比如要在容器里面创建一个用户。创建用户前和创建用户后系统发生了什么变化。

（差3min）



通过linux capability来设置。然后呢对于一个容器，容器外面是没有特权的，但是在容器里面是root。

在host上Ming1000，在LV1里Ming是root。任何一个用户在里面有一个user id，在外面也有一个user id。在内核里就可以根据这个不同的id来判断一个user在不同的层次有没有权限。



限制系统调用，seccomp——可以给每个容器配置它的seccomp的表，假设一个容器没有能力调某一个syscall，一旦调就会报错。

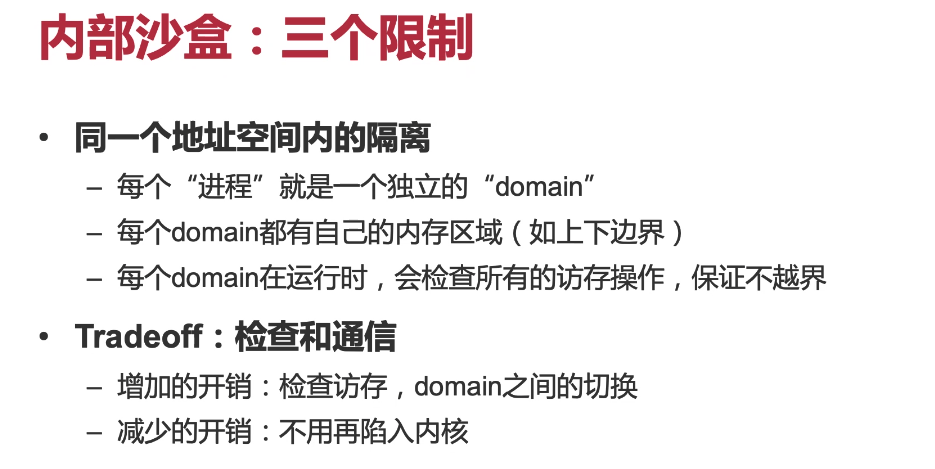


cgroup之后会讲，它主要做资源方面的隔离。cgroup可以用来防御fork bomb。可以限制这个进程或者说这个容器最多能创建多少个pthread。

## SFI：软件错误隔离

之前说的都是用已有的东西来做隔离。一旦想要突破进程的边界那肯定是做不到的。SFI则是另外一种思路：在用户态用软件的方式实现隔离。

同一个address space里只有一个进程，一个进程里能不能做隔离。比如一个write跑飞了，所有能write的地方都会去write。

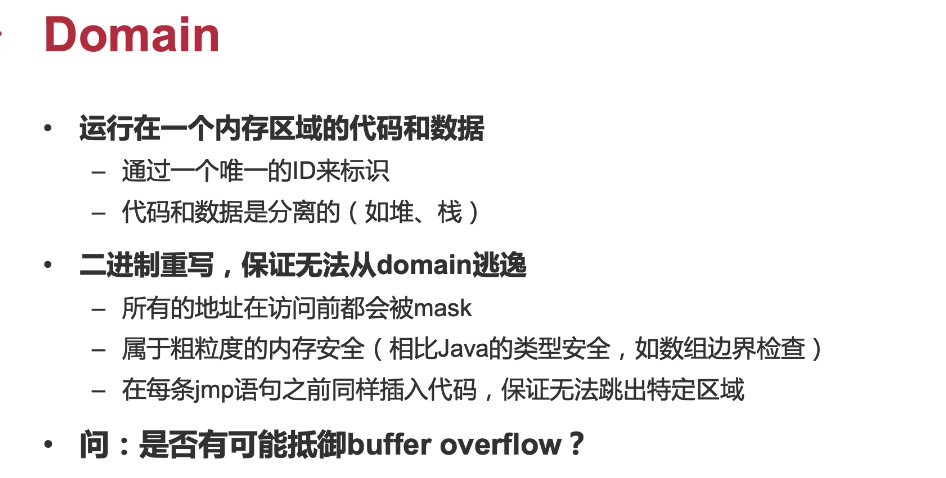


domain确定内存的上界和下界。

这个检查不仅包括load store还要检查jmp。域间通信。网上下一段二进制代码，隔出一个域。

而domain之间的切换也不像过去一样简单了。

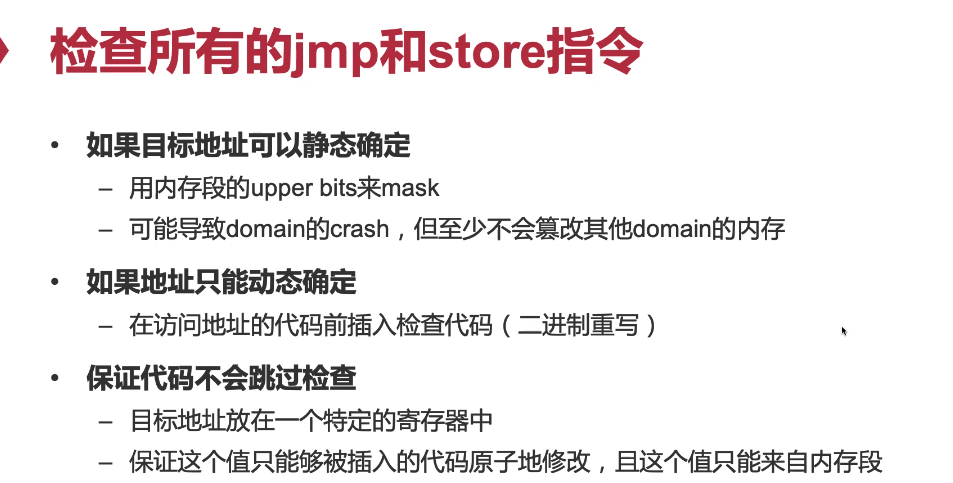
减少的开销：都不用进内核了。



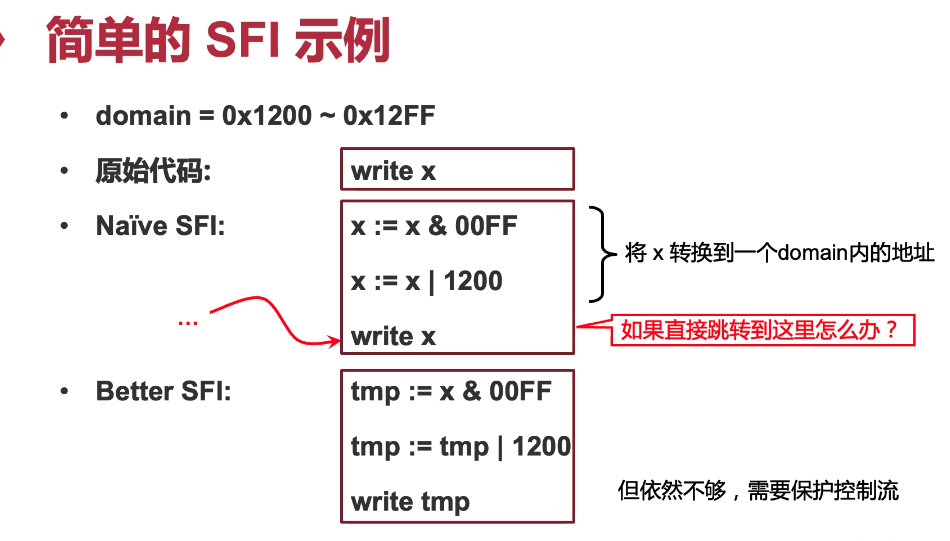
代码+数据，形成一个闭包。

如何限制——二进制重写——加载到这段区域之前，每一条跟指令相关的操作无论访存还是跳转，都加一个mask。使得你所有的访存都在我指定的范围里。这种方式是粗粒度的内存安全。java是类型安全，一个指针没有办法访问到类型外面的。但是SFI来说，0-1000不会越这个界。同样每条jmp之前也会mask防止。大家想覆盖掉函数指针会出什么问题：就算了你覆盖了函数指针，跳转之前会把你前面那些数字清零。还是在这个区域里。

ret怎么防呢？



如何保证检查代码不会被跳过，目标地址放在一个寄存器里。



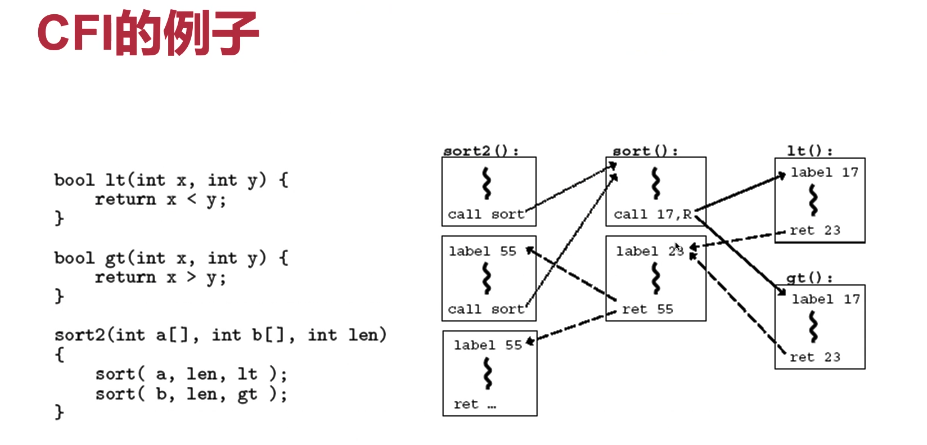
直接跳到write x就会绕过前面的操作。

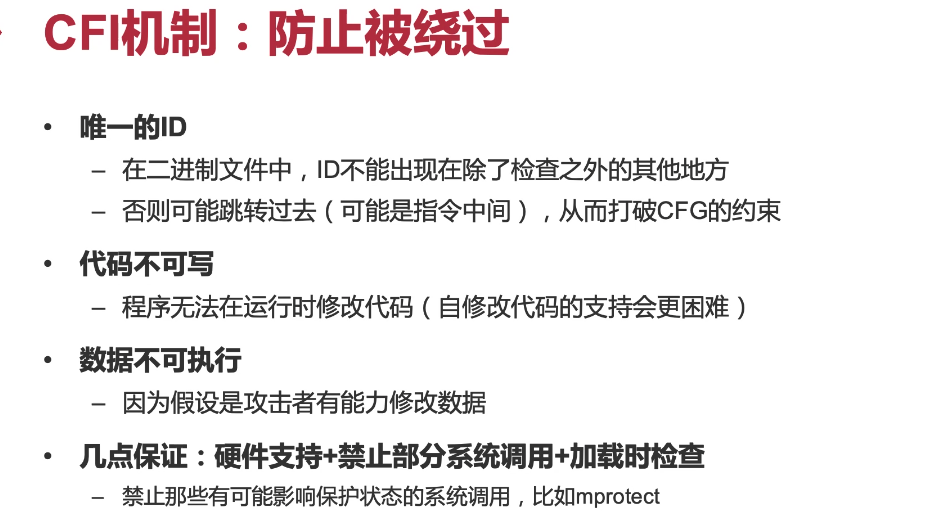
tmp在一个寄存器里，tmp这个值永远不可能超过1200到12ff的。我可以把tmp初始化成1200。之后tmp的修改只会在合法的范围里改。

函数正常的逻辑根本不用这个寄存器，所以这个寄存器不会被修改。就算有人跳过来也没事。

### CFI：控制流完整保护



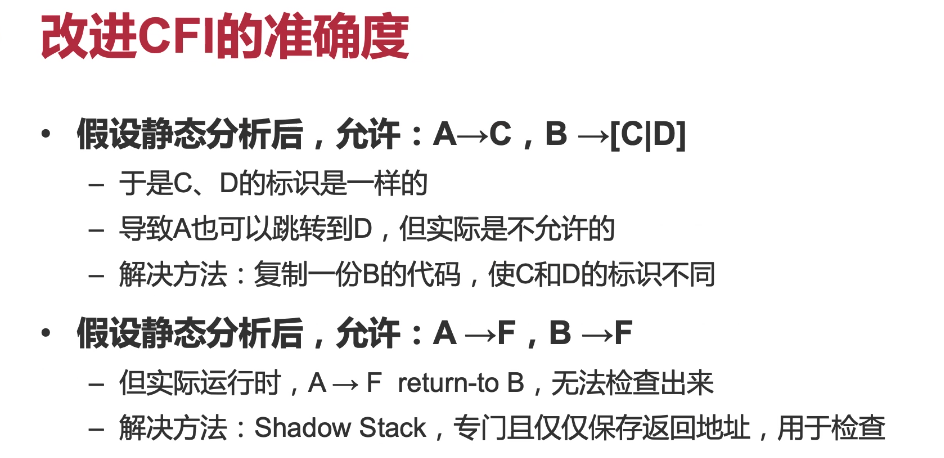




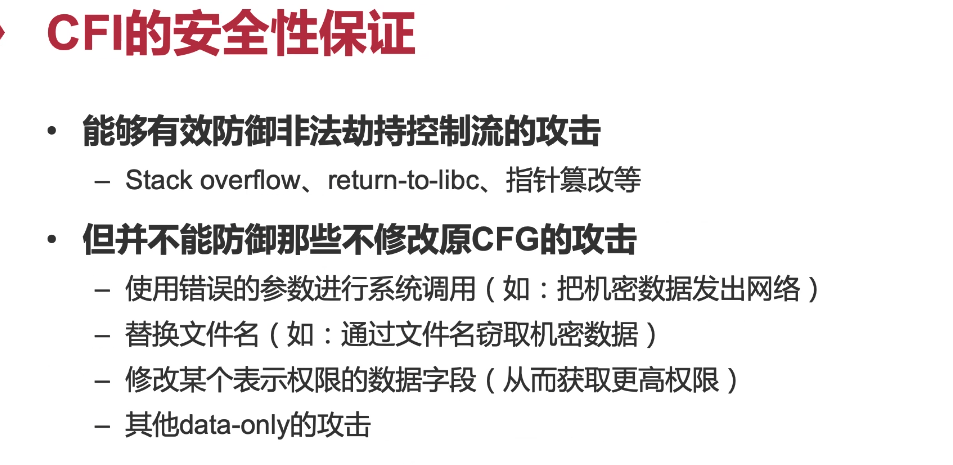
如果代码可以写的话

buffer overflow只在ret的时候才破坏control flow。代码如果能被改那那些二进制插桩都没用了。

加载时检查：要检查是否真正的加了CFI的检查。

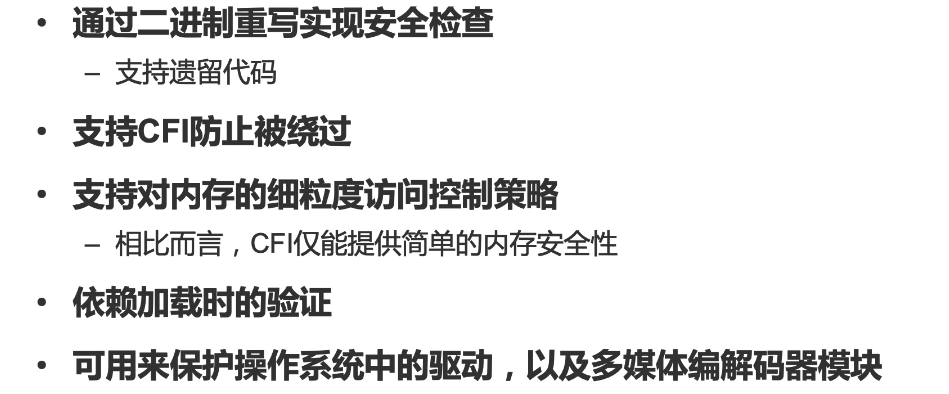


每次return的时候两遍pop出来检查一下就可以了。



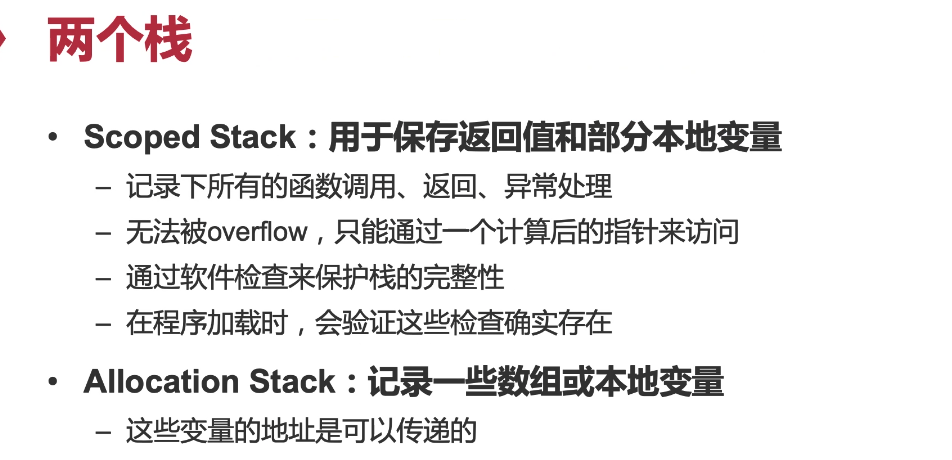
mproct就是一个例子，把mproct的参数破坏掉。

## XFI : SFI+CFI

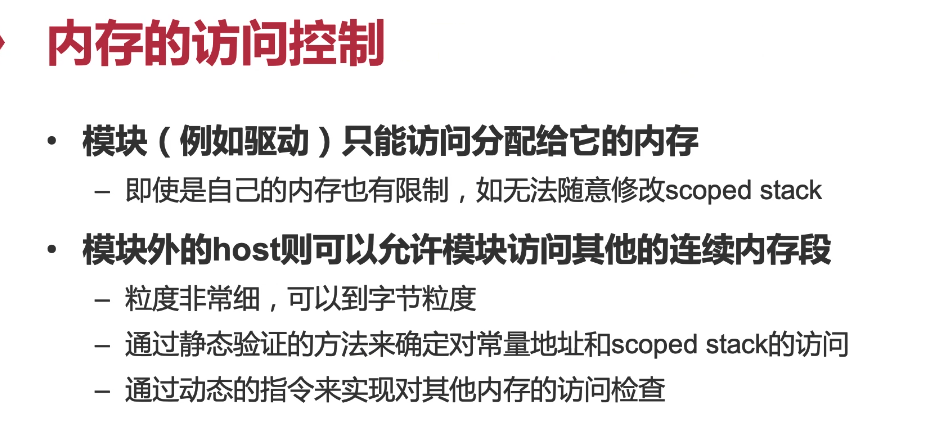


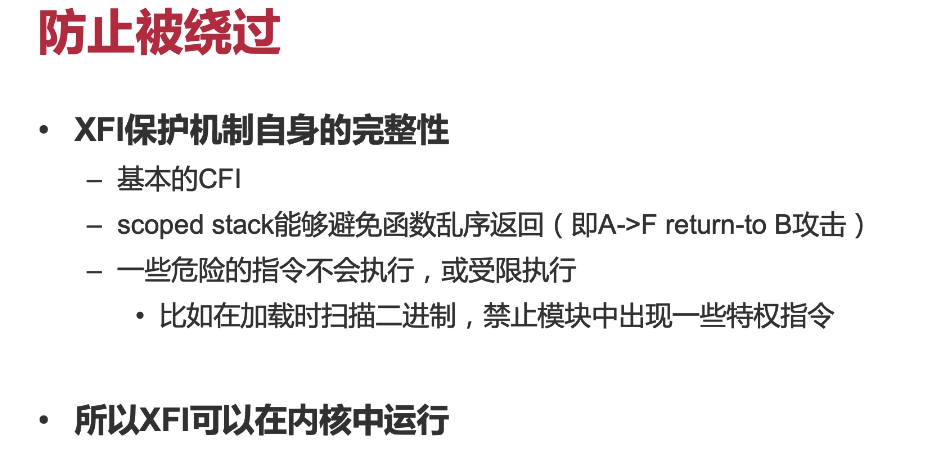
通过XFI把这个驱动和OS的其他部分隔离开。

微软提出了两个栈

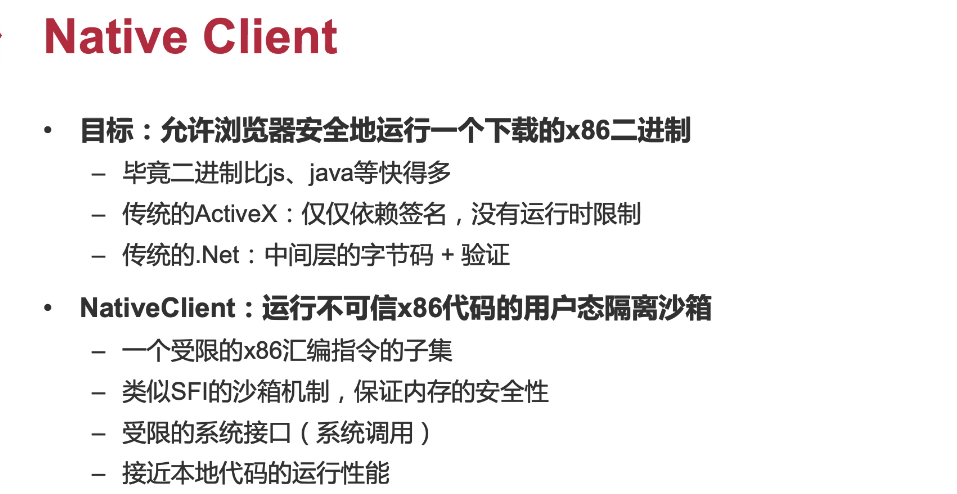


对于scoped stack的访问是通过特殊插入指令来访问的。





防止运行特殊指令：比如关中断，加载的时候扫描一遍二进制，保证不会出现这样的指令。

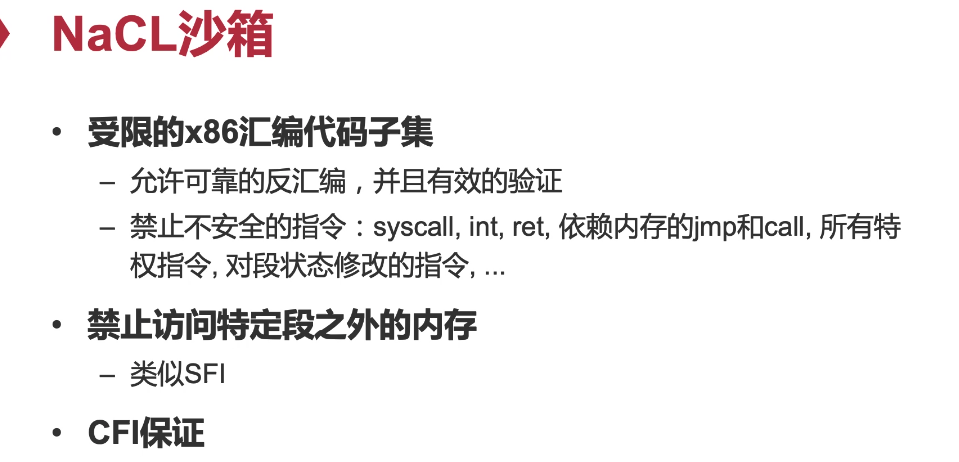


比如flash要用很多CPU，允许跑二进制。这个native跑就很危险。传统的ActiveX安不安全全靠签名。

不能指望所有检查代码的工具能把所有bug都找出来。

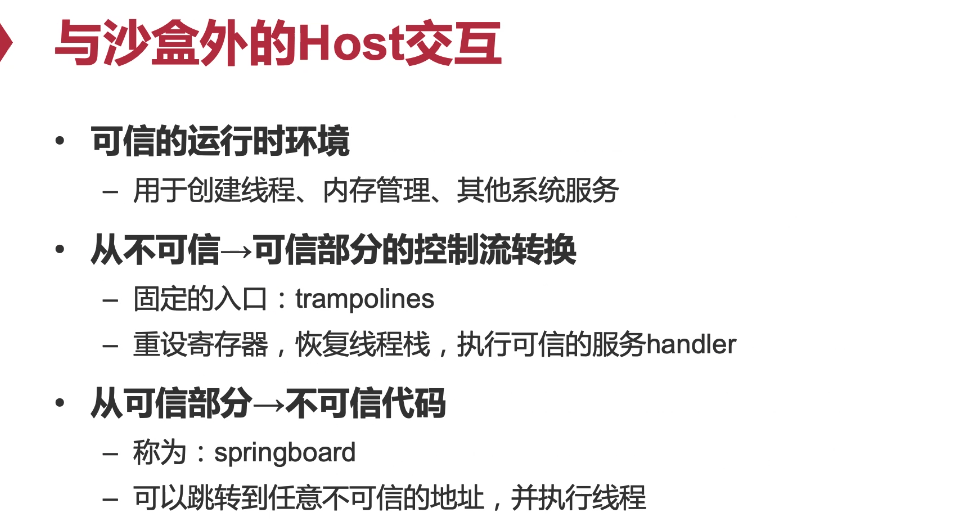
dot Net依然性能比较差。

native client和SFI就很像了，所有的访存指令都会去做一个检查，其次系统调用时受限的。再提供一个受限的X86的汇编的子集。

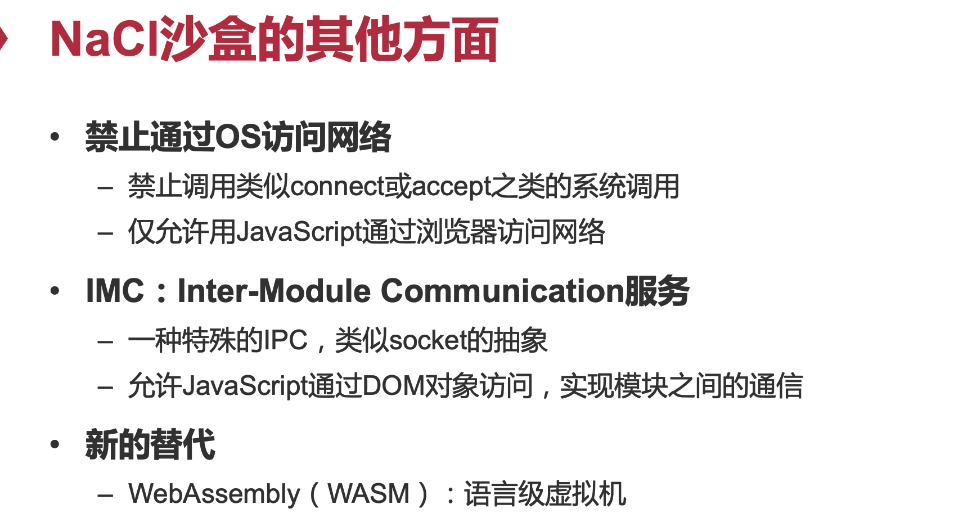




没有ret指令都是通过间接跳转。



springboard可以转到任意不可信地址。



syscall都不支持，当然不支持访问网络，必须回到JavaScript。

Native Client只是用来执行一段二进制代码。

任何代码都可以通过LLVM编译成WASM。