操作系统安全

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

https://ipads.se.sjtu.edu.cn

版权声明

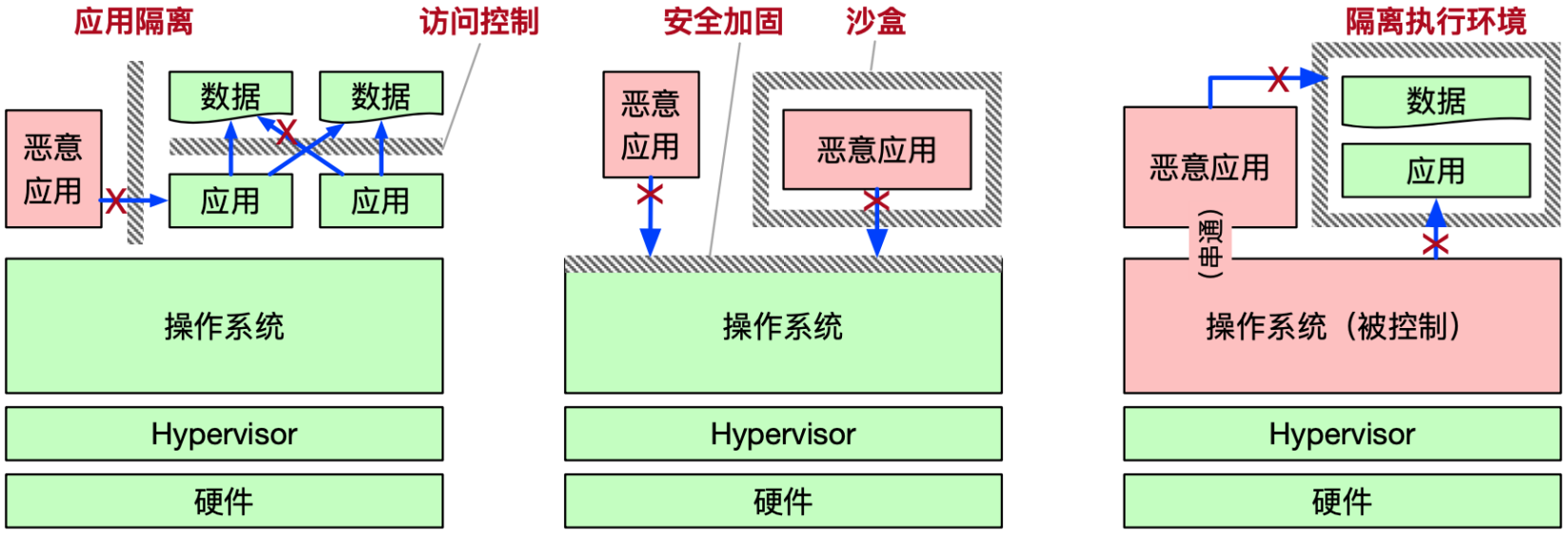
- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

操作系统的安全服务

安全是操作系统的重要功能和服务

- **系统中有许多需要保护的数据**
  - 如账号密码、信用卡号、地理位置、照片视频等
  - 操作系统需要允许这些数据被合法访问，但不允许被非法访问
- **系统中可能存在许多恶意应用**
  - 操作系统需要与这些恶意应用作斗争，保护自己，限制对方
- **操作系统不可避免的存在漏洞**
  - 操作系统需要考虑自己被完全攻破的情况下依然提供一定的保护

操作系统安全的三个层次



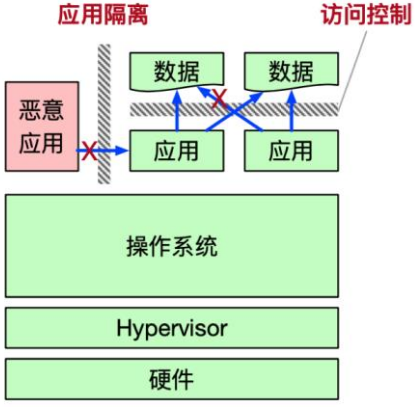
(a) 基于操作系统的应用隔离与访问控制

(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

层次一：基于OS的应用隔离与访问控制

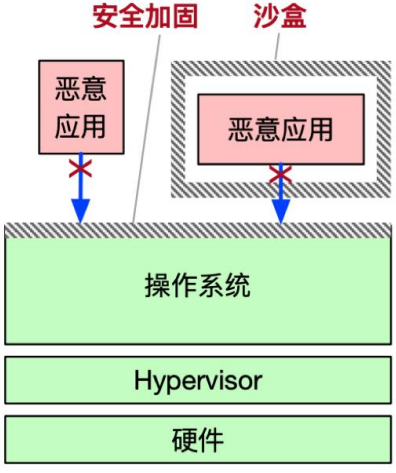
- **威胁模型**
  - 操作系统是可信的，能够正常执行且不受攻击
  - 应用程序可能是恶意的，会窃取其他应用数据
  - 应用程序可能存在bug，导致访问其他应用数据



- **应用隔离**
  - 内存数据隔离：依赖进程间不同虚拟地址空间的隔离
  - 文件系统隔离：文件系统是全局的，需限制哪些应用不能访问哪些文件
    - 操作系统提供对文件系统的**访问控制**机制

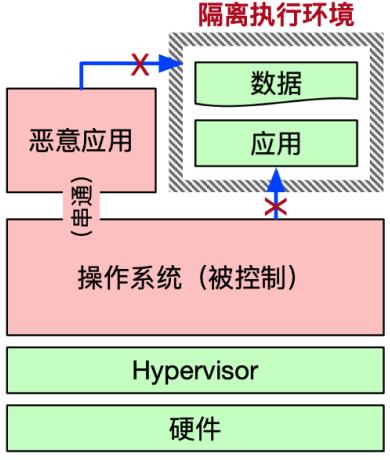
层次二：OS对恶意应用的隔离与防御

- **威胁模型**
  - 操作系统存在bug和安全漏洞
  - 操作系统的运行过程依然可信
  - 恶意应用利用操作系统漏洞攻击，获取更高权限或直接窃取其他应用的数据
- **操作系统防御**
  - 防御常见的操作系统bug/漏洞
  - 沙盒机制限制应用的运行



层次三：OS不可信时对应用的保护

- **威胁模型**
  - 操作系统不可信，有可能被攻击者完全控制
  - 恶意应用可能与操作系统串通发起攻击
- **基于更底层的应用保护**
  - 基于Hypervisor的保护：可信基更小
  - 基于硬件Enclave的保护：硬件通常更可信



操作系统安全的三个概念

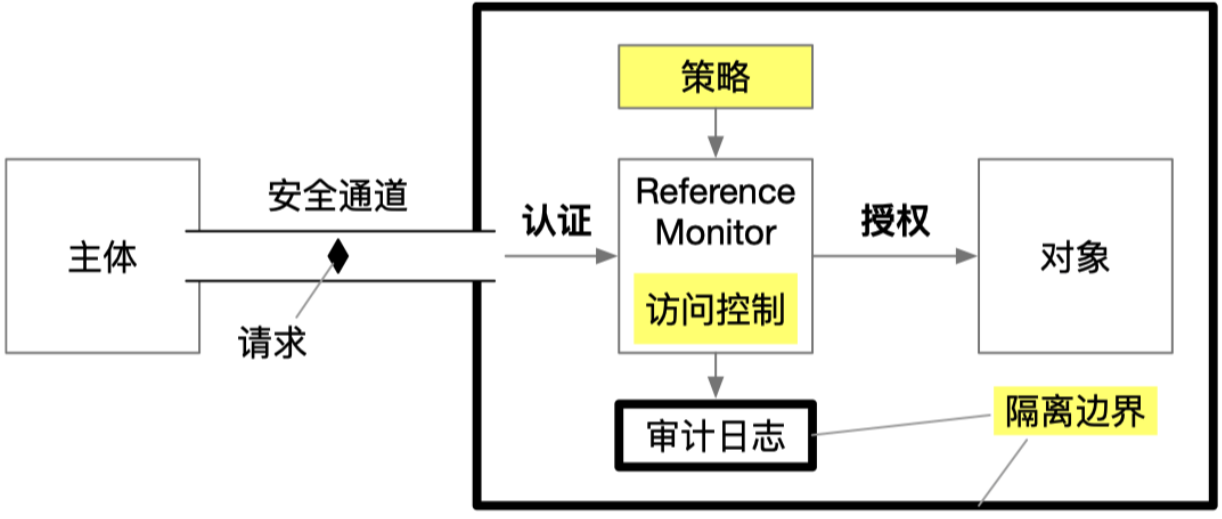
- **可信计算基（Trusted Computing Base）**
  - 为实现计算机系统安全保护的所有安全保护机制的集合
  - 包括软件、硬件和固件（硬件上的软件）
- **攻击面（Attacking Surface）**
  - 一个组件被其他组件攻击的所有方法的集合
  - 可能来自上层、同层和底层
- **防御纵深（Defense in-depth）**
  - 为系统设置多道防线，为防御增加冗余，以进一步提高攻击难度

访问控制（ACCESS CONTROL）

访问控制与引用监视器

- **访问控制**
  - 按照访问实体的身份来限制其访问对象的一种机制
  - 为了实现对不同应用访问不同数据的权限控制
  - 包含"认证"和"授权"两个重要步骤
- **引用监视器（Reference Monitor）**
  - 是实现访问控制的一种方式
  - 主体必须通过引用（reference）的方式间接访问对象
  - Reference monitor 位于主体和对象之间，进行检查

引用监视器（Reference Monitor）机制



Reference Monitor 负责两件事：

1. **认证（Authentication）**：确定发起请求实体的**身份**
2. **授权（Authorization）**：确定实体确实拥有访问资源的**权限**

认证机制

- **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
- **有什么 （Something you have）**
  - 例如 USB-key、密码器等实物
- **是什么 （Something you are）**
  - 如指纹、虹膜、步态、键盘输入习惯等属于人的一部分

问：比特币私钥算哪种？

授权机制

	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

- **权限矩阵**
  - 对象与实体的关系
- **矩阵有多大?**
  - 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%
  - 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

授权机制：以文件系统为例

- **使用用户组的概念，将用户分为三类**
  - 文件拥有者、文件拥有组、其他用户
  - 每个文件只需要用9个bit即可：3种权限（读-写-执行） x3 类用户
- **何时检查用户权限?**
  - 每次操作文件时检查，包括打开、读、写等
    - 可保证检查的完备性，但对性能影响较大
  - 仅在每次打开文件检查，性能开销较小
    - 引入fd，作为其他操作的参数

最小特权级：SUID 机制

- **问题：passwd如何工作?**
  - 用户有权限使用 passwd 修改自己的密码
  - 用户的密码保存在 /etc/ 目录下的文件，用户无权访问
- **解决方法：运行 passwd 时使用 root 身份**
  - 如何保证用户提权为root后只能运行passwd?
  - 在passwd的inode中增加一个sticky位，使得仅在执行该文件时才会被提权，从而将进程提权的时间和能力降至最小

基于角色的访问控制（RBAC）

- **RBAC：将用户与角色解耦的访问控制方法**
  - 提出了角色的概念，与权限直接相关
  - 用户通过拥有一个或多个角色间接地拥有权限
  - "用户-角色"，以及"角色-权限"，一般都是多对多的关系
- **RBAC的优势**
  - 设定角色与权限之间的关系比设定用户与权限之间的关系更直观
  - 可一次性地更新所有拥有该角色用户的权限，提高了权限更新的效率
  - 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
    - 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）

DAC与MAC

- **自主访问控制（DAC: Discretionary Access Control）**
  - 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
- **强制访问控制（MAC: Mandatory Access Control）**
  - 什么数据能被谁访问，完全由底层的系统决定
  - 例如，在军队中，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

Bell-LaPadula 模型

- **BLP属于强制访问控制（MAC）模型**
  - 一个用于访问控制的状态机模型
  - 目的是为了用于政府、军队等具有严格安全等级的场景
- **BLP 规定了两条 MAC 规则和一条 DAC 规则：**
  - 简单安全属性：某个安全级别的主体无法读取更高安全级别的对象
  - * 属性（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
  - 自主安全属性：使用访问矩阵来规定 自主访问控制（DAC）

能力机制（Capability）

- **Capability 列表是权限矩阵的实现方法之一**
  - 从实体角度出发，列出该实体所拥有的能访问的对象及相应的权限
  - 文件系统中，fd就是一种capability
  - Linux还定义了许多其他的capability

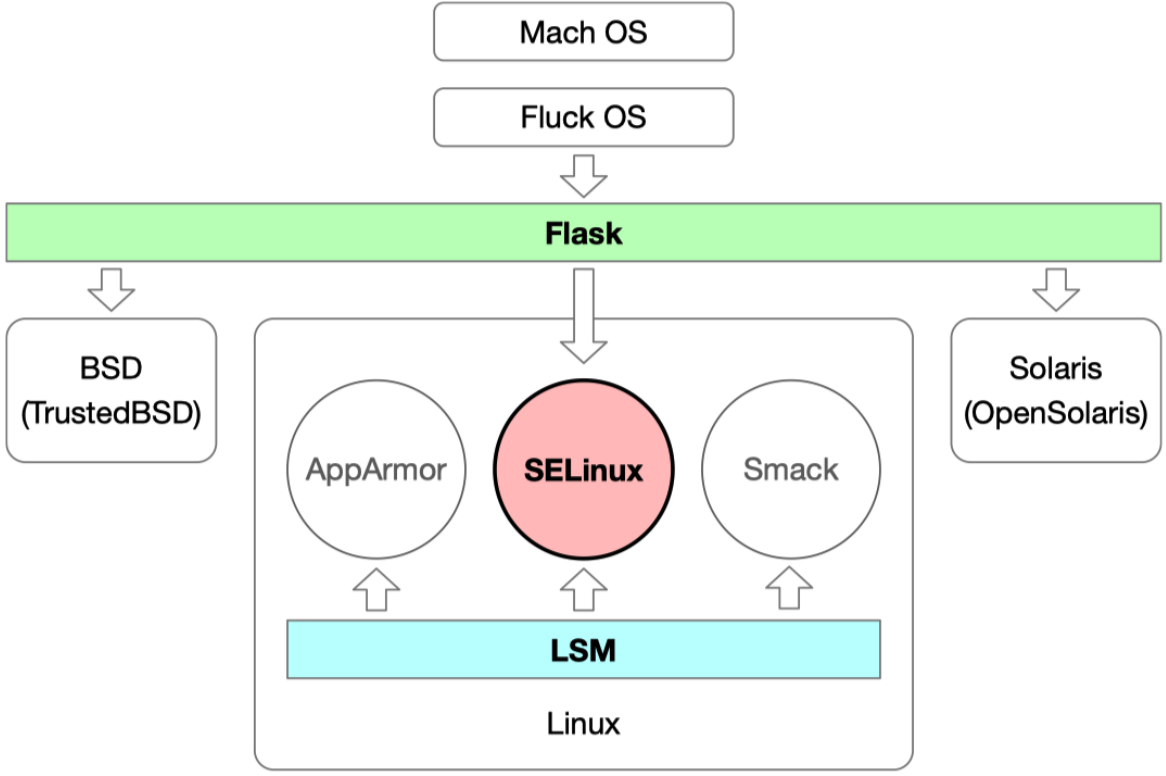
Capability 名称	具体描述
CAP_AUDIT_CONTROL	允许控制内核审计（启用和禁用审计，设置审计过滤规则，获取审计状态和过滤规则）
CAP_AUDIT_READ	允许读取审计日志（通过 multicast netlink socket）
CAP_AUDIT_WRITE	将记录写入内核审计日志
CAP_BLOCK_SUSPEND	允许使用可阻止系统挂起的特性
CAP_CHOWN	允许任意修改文件的 UID 和 GID

案例：SELINUX

SELinux的历史

- **SELinux，由NSA发起，2003年并入Linux**
  - 是 Flask 安全架构在 Linux 上的实现
    - Flask 是一个 OS 的安全架构，可灵活提供不同的安全策略
  - 是一个 Linux 内核的安全模块（LSM）
    - 在Linux内核的关键代码区域插入了许多 hook进行安全检查
- **SELinux 提供一套访问控制的框架**
  - 支持不同的安全策略，包括强制类型访问（MAC）

SELinux、Flask与LSM



SELinux引入的概念

- **用户（User）：指系统中的用户**
  - 与 Linux 系统用户并没有关系
- **策略（Policy）：一组规则（Rule）的集合**
  - 默认是"Targeted"策略，主要对服务进程进行访问控制
  - MLS （Multi-Level Security），实现了 Bell-LaPadula 模型
  - Minimum，考虑资源消耗，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：是主体和对象的标签（Label）**
  - 用于访问时的权限检查
  - 可通过"ls -Z"的命令来查看文件对应的安全上下文

SELinux的访问向量

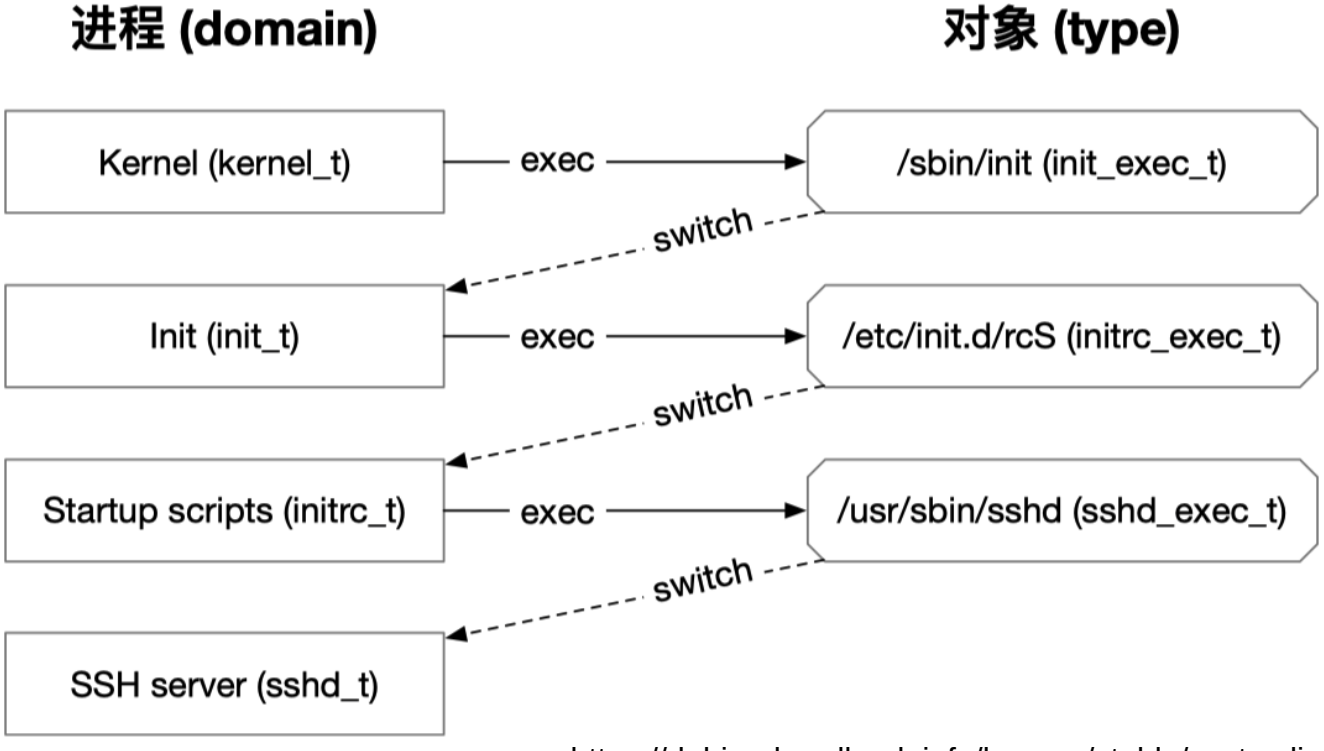
- **SELinux 将访问控制抽象为一个问题：**
  - 一个 < **主体** > 是否可以在一个 < **对象** > 上做一个 < **操作** >

- **AVC: Access Vector Cache**
  - SELinux 会先查询AVC，若查不到，则再查询安全服务器
  - 安全服务器在策略数据库中查找相应的安全上下文进行判断

SELinux的安全上下文

- **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
- **标签的格式：用户:角色:类型:MLS层级**
  - 用户登录后，系统根据角色分配给用户一个安全上下文
  - 类型（Type）用于实现访问控制
    - 每个对象都有一个 **type**
    - 每个进程的type称为 **domain**
      - 一个角色对应一个domain
      - 重要的服务进程被标记为特定的domain：
      - 例如：/usr/sbin/sshd 的类型为 sshd_exec_t

进程的domain与对象的type

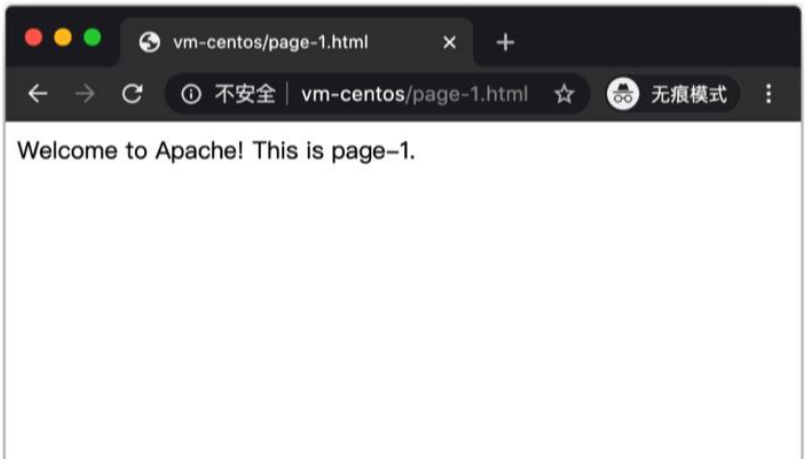


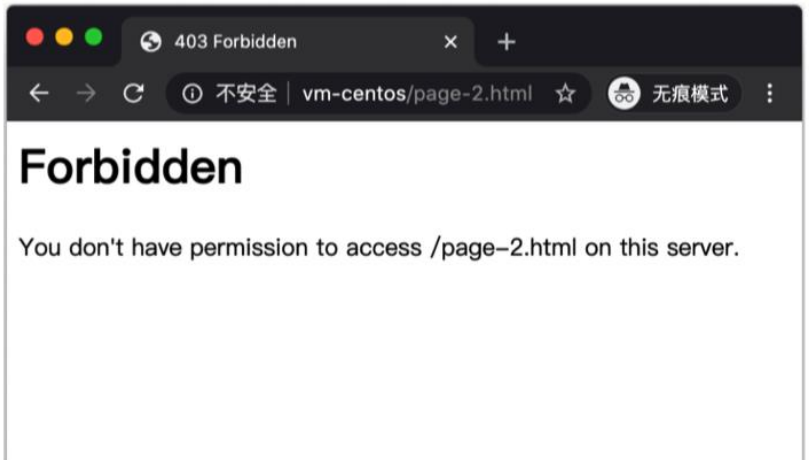
https://debian-handbook.info/browse/stable/sect.selinux.html

实例

```
[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
```





SELinux的策略与规则

- **SELinux 中的规则很多，而且很复杂**
  - 很可能导致规则与规则之间产生冲突，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，形成策略方便选择
    - 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
  - 可通过 semanage 命令来查看规则

```
[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
...
```

操作系统内部安全

操作系统漏洞分类的三个角度

- **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、整形溢出错误、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
- **攻击模块**
  - 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

整形溢出漏洞

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

Return-to-user攻击（ret2usr）

- **内核错误的运行了用户态的代码**
  - 由于内核与应用程序共享同一个页表，内核运行时可以任意访问用户态的虚拟地址空间，内核可能执行位于用户态的代码
- **攻击者的常用方法**
  - 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
  - 也可以利用内核的栈溢出漏洞，覆盖栈上的返回地址为恶意代码的地址，使内核在执行 ret 指令时跳转到位于用户态的代码

ret2usr攻击的防御方法

- **方法一：仔细检查内核中的每个函数指针**
  - 需对内核所有模块进行检查，很难做到 100% 的覆盖率
- **方法二：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**
  - 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
  - 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
- **方法三：硬件保证CPU处于内核态时不得运行任何用户态的代码**
  - 如 Intel 的 SMEP（Supervisor Mode Execution Prevention）技术
  - ARM 同样有类似 SMEP 的技术，称为 PXN（Privileged eXecute-Never）

SMEP 不能完全解决 ret2usr：ret2dir

- **操作系统管理内存的方法"直接映射"**
  - 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
  - 因此，同一块物理内存在系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
- **基于直接映射的攻击，可绕过SMEP**
  - 攻击者首先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，然后在 ret2usr 攻击中让内核跳转到该地址执行（内容依然为攻击者控制）
  - 攻击成功还有一个前提：直接映射区域必须是可执行的
    - 在 3.8.13 以及 之前的 Linux 版本，将直接映射区域的权限设置为了"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**

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
- **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
- **内核态的Rootkit**
  - 可以是 hook 某个内核中的关键函数，从而在该函数被调用时触发运行
  - 可以是以内核线程的方式运行
  - 可以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

KASLR：内核地址布局随机化

- **ASLR 与 KASLR**
  - ASLR 通过随机化地址空间布局来提高系统攻击难度
  - KASLR是对内核启用地址随机化
- **KASLR 防御 ret2dir 攻击**
  - 攻击者需要知道用户态恶意代码在内核中直接映射区域的地址
  - KASLR 通过将内核的虚拟地址布局进行随机化，使攻击者准确定位内核地址的难度大大提升

案例：IOS的系统安全

文件加密保护机制

- **每个文件都加密，且秘钥均不相同**
  - 创建文件时，系统会新建一个 256-bit 的秘钥用于该文件的加密
  - 硬件的 AES 引擎会在写 入文件时，使用该秘钥对文件进行 AES CBC 模式加密，再写入到闪存中
  - 加密过程中的初始化向量（IV，Initialization Vector）使用了当前块在文件中的偏移，然后用文件秘钥的 SHA-1 值进行加密得到

进程沙盒机制

- **所有第三方 App 都运行在一个沙盒中**
  - 被限制访问其他应用程序存储的文件或对设备进行修改
  - 若第三方 App 需要访问其任何沙盒外部的数据， 只能显式地通过 iOS 提供的 API 来访问
  - 因此 iOS 可以在 API设置各种检查来进一步增强安全性
- **第三方 App 都以mobile的非特权用户身份运行**
  - 整个 OS 的分区以只读方式加载，所有的数据内存区域均使用 ARM 的 XN 机制（Execute Never）标记为不可执行
  - 只有很少部分的内存页被标记为可写可执行（例如 Safari，用于 JavaScript 的 JIT），内核会仔细检查 App 是否有权限做这样的映射

侧信道与隐秘信道

什么是隐秘信道?

- **隐秘信道（Covert Channel）**
  - 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
  - 常见的隐秘信道：时间、功耗、电磁泄露、声音等
- **例：消费记录的应用 A，在没有网络的情况下如何把数据发出去?**
  - 假设有一个应用B运行在同一个手机
  - 若A可播放声音，B可录音，则A把数据编码为声音发送给B
  - 若A可打开闪光灯，B可摄像，则A把数据编码为光的闪烁长短与频率发送给B
  - 若A可震动，B可访问运动传感器，则A把数据编码为震动频率发送给B
  - 若B可访问CPU温度，则A可长时间运行计算密集代码，CPU升温表示1，反之为0
  - …

侧信道与隐秘信道的关系

- **侧信道与隐秘信道很类似**
  - 两者都使用类似的方式进行数据的传递
- **侧信道攻击和隐秘信道攻击的不同**
  - 隐秘信道攻击：两方是互相串通的，其目的就是为了将信息从一方传给另一方
  - 侧信道攻击：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
    - 即被攻击者无意通过侧信道泄露了自己的数据

缓存信道（Cache Channel）

- **利用缓存的状态推测执行的信息**
  - 例如：可根据 func_a 还是 func_b 的代码在缓存中，判断 i 的值
- **常见的四种攻击方式**
  - Flush+reload
  - Flush+flush
  - Prime+probe
  - Evict+time

```
if (i == 0)
    func_a();
else
    func_b();
```

Flush+Reload

- **攻击步骤**
  - 1. 攻击进程首先将 cache 清空
    - 可通过不断访问其他内存来占满 cache，或直接通过 flush 将 cache 清空
  - 2. 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，意味着目标进程没有访问过该变量
    - 若时间短，则表示 cache hit，意味着目标进程访问过该变量
- **特点分析**
  - 优点：可以跨CPU核，甚至跨多个CPU；噪音低
  - 缺点：攻击准备难度高，需构造与目标进程完全相同的内存页

Flush+Flush

- **基于缓存刷新时间（如clflush）来推测数据在缓存中的状态**
  - 1. 攻击进程首先将 cache 清空（Flush）
  - 2. 等待目标进程执行
  - 3. 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，目标进程曾访问对应的内存
- **特点分析**
  - 优点：只需清空缓存而不需实际访存，因此具有一定的隐蔽性
  - 缺点：clflush对于有数据和无数据的时间差异不明显，攻击精度不高

Evict+Reload

- **场景：CPU没有 clflush 指令**
  - 1. 将关键数据所在的 cache set 都替换成攻击进程的数据
  - 2. 等待目标进程执行
  - 3. 访问 cache set 中的某个数据
    - 若时间很短，说明目标进程没有将该数据 evict，即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
- **特点分析**
  - 优点：无需依赖 flush 指令
  - 缺点：无法支持动态分配的内存；需要了解 LLC 的 eviction 策略；Cache 必须是 inclusive；无法很好地支持多 CPU

Prime+Probe

- **攻击的具体步骤如下:**
  - 1. 攻击进程用自己的数据将 cache set 填满（Prime）
  - 2. 等待目标进程执行
  - 3. 再次访问自己的数据
    - 若时间很短，说明目标进程没有将该数据 evict，即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
- **特点分析:**
  - 优点：不需要共享内存;支持动态和静态分配的内存
  - 缺点：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；需要首先定位目标进程使用的cache set

侧信道攻击的防御

- **侧信道攻击很难被完全防御住，根本原因在于共享**
  - 当被攻击者在做了某个操作后，对系统整体产生了影响
  - 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：发现影响和没发现影响（即做了操作和没做操作）可以被编码为 0 和 1
- **防御侧信道的根本方法：不共享**
  - 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，甚至空间（光、温度、声音）
  - 更实际的方法是针对常见攻击进行防御

常量时间（Constant Time）算法

- **算法的运行时间与输入无关**
  - 无法通过运行时间得到与输入相关的任何信息
  - 代码执行没有分支跳转
- **常见的实现方法：cmov**
  - Conditional MOV
- **缺点：计算变得更慢**
  - 需要做两份运算

```
/* 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)
```

不经意随机访问内存（ORAM）

- **ORAM 将访存行为与程序执行过程解耦**
  - 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
- **最简单的实现：定时、定量、定位的访问方式**
  - 无论实际是否有访存需求，均以**固定周期**，访问**固定位置**，每次访问**固定的大小**
  - 例如， CPU 顺序循环访问所有的有效内存区域，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
  - 类似上海和北京之间的高铁，无论乘客是谁，都按照时刻表运行，哪怕有时候位子没坐满也发车，因此根据高铁的班次并不能反推出谁坐了高铁
- **ORAM 会引入很大的额外负载**
  - 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

案例：MELTDOWN与KPTI

Meltdown与Spectre攻击

- **Meltdown（熔断）和 Spectre（幽灵）**
  - 在 2017 年被发现，在 2018 年被公开
- **开创了一类新的侧信道攻击方式**
  - 利用了 CPU 的预测执行机制
  - 几乎所有的主流 CPU 都受到了影响
    - 包括 Intel、 AMD和部分 ARM 处理器
  - 许多软件厂商不得不紧急打补丁并做出对应的防御措施

CPU预测执行机制

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

- **CPU为了性能会进行预测执行**
  - 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
- **然而，上述过程忽略了一个问题：**
  - 数组 a[] 在预测执行的过程中被访问了一次——会有什么问题?

Meltdown攻击原理

- **CPU的漏洞**
  - 若让 CPU 预测执行一条跨权限非法内存访问的指令，CPU 不会进行权限检查；若最后预测条件不满足，CPU不会报错
  - 攻击者可通过预测执行的方式，非法访问内核数据，然后根据内核数据改变cache状态，最后观测cache状态反推出内核数据
- **问：属于侧信道还是隐秘信道?**
- **答：属于隐秘信道（Covert Channel)**
  - 攻击的两侧一边是构造cache状态，另一边是读取cache状态，两边都是攻击者控制；中间的墙，就是用户态与内核态的隔离

Meltdown攻击代码

```
; rcx = kernel address, rbx = probe array
xor rax, rax
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

来自 Meltdown 的论文

- **攻击能力很强**
  - 把整个内核的内存都 dump 出来，最高可以达到 503KB/秒

KPTI：内核页表隔离

- **如何防御Meltdown漏洞?**
  - 根本原因是硬件缺陷：预测执行时没有进行权限检查
  - 硬件修改不易，通过软件来弥补
- **在命名空间层，通过页表隔离**
  - 使内核不再与应用共享虚拟地址空间，内核单独使用一个页表
  - 导致内核与应用状态切换时，需要新增页表的切换
    - 引入额外开销，包括TLB的刷新等
    - 部分应用性能降低可达30%

下次课内容

- **硬件辅助的操作系统安全**