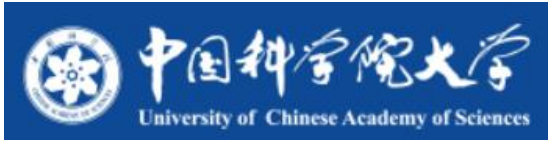
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

操作系统攻防

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



回顾：fd与Capability的类似之处

- **文件描述符 fd 可以看做是一类 Capability**
  - 用户不能伪造 fd，必须通过内核打开文件（回顾 file_table/fd_table）
  - fd 只是一个指向保存在内核中数据结构的"指针"
  - 拥有 fd 就拥有了访问对应文件的权限
  - 一个文件可以对应不同 fd，相应的权限可以不同
- **fd 也可以在不同进程之间传递**
  - 父进程可以传递给子进程（回顾pipe）
  - 非父子进程之间可以通过 sendmsg 传递 fd

回顾： 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
  - 但是对部分场景（如军队）来说，DAC过于灵活
    - 例如，文件拥有者是否真的有权可随意设置文件权限?
- **强制访问控制（MAC: Mandatory Access Control）**
  - 由"系统"增加一些强制的、不可改变的规则
    - 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
  - MAC与DAC可以结合，此时MAC的优先级更高

回顾：Bell-LaPadula 模型

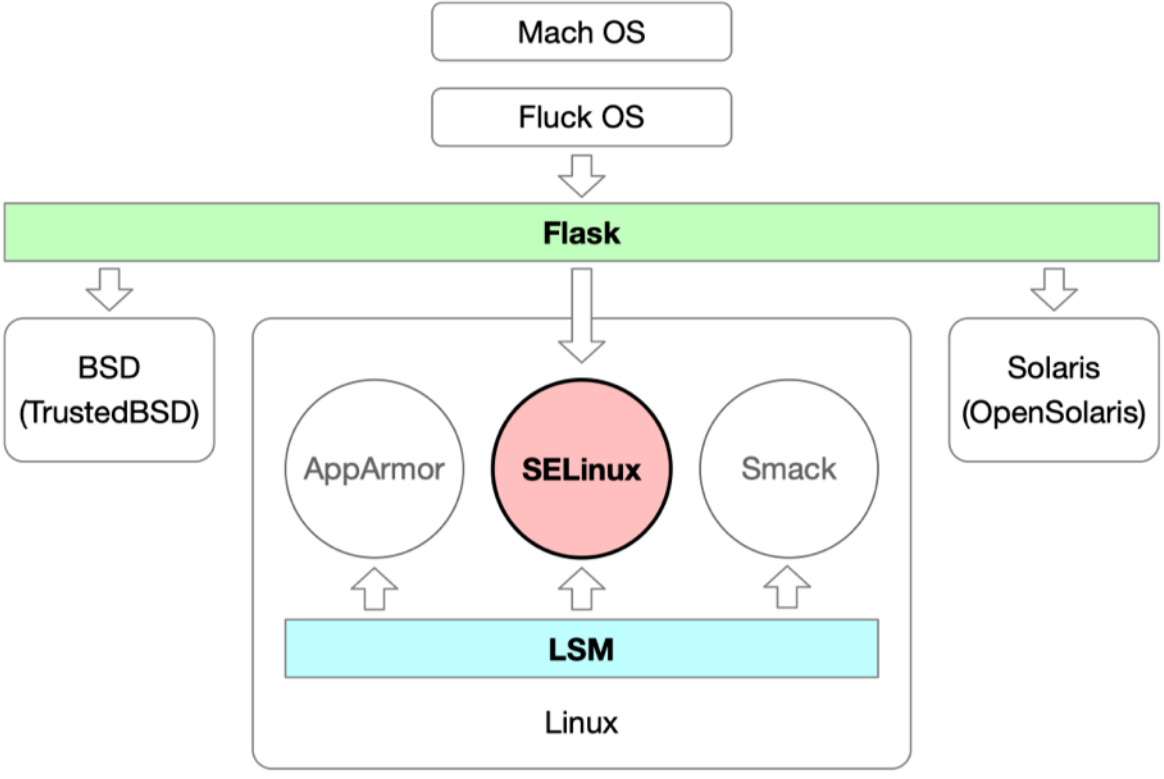
- **BLP属于强制访问控制（MAC）模型**
  - 一个用于访问控制的状态机模型
  - 目的是为了用于政府、军队等具有严格安全等级的场景
- **BLP 规定了两条 MAC 规则和一条 DAC 规则：**
  - 简单安全属性：某个安全级别的主体无法读取更高安全级别的对象
  - * 属性（星属性）:某一安全级别的主体无法写入任何更低安全级别的对象
  - 自主安全属性：使用访问矩阵来规定自主访问控制（DAC）

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

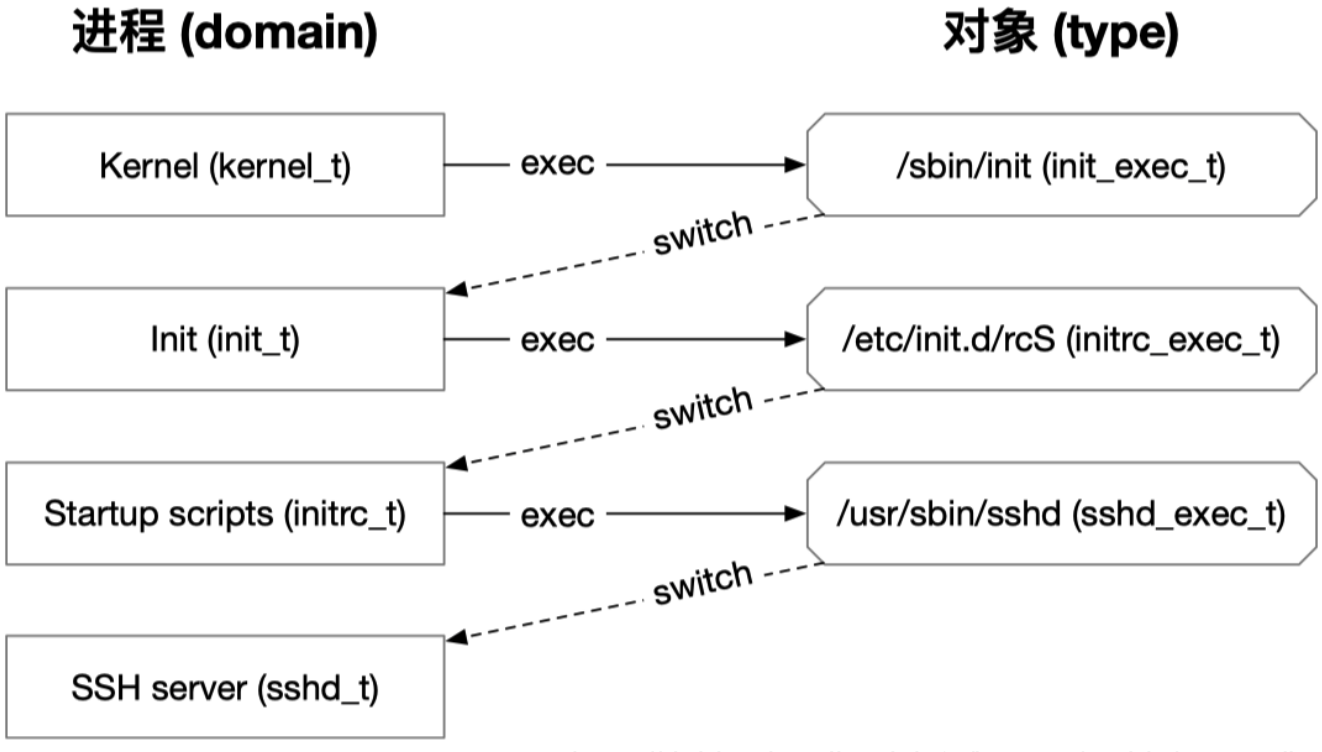
SELinux的访问向量

- **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** >
  - 3W：Who, Which (obj), What (operation)
- **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

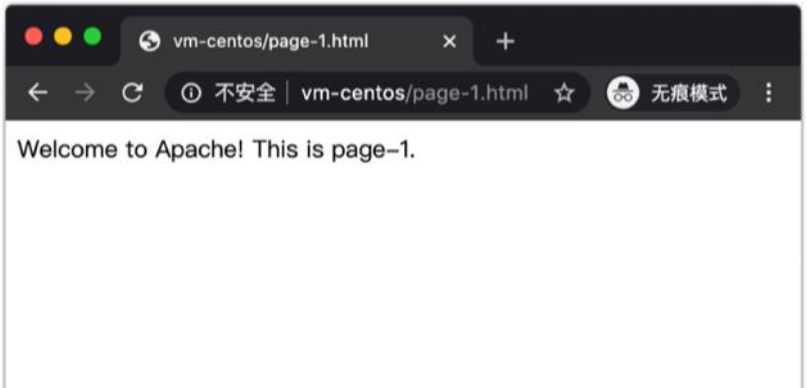


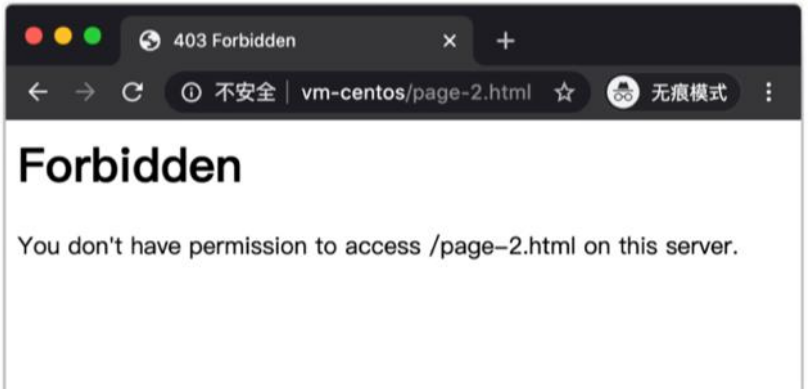
https://debian-handbook.info/browse/stable/sect.selinux.html

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
```

page-2.html 被标记为 admin_home_t，即使被错误放入网页目录，也无法被apache访问



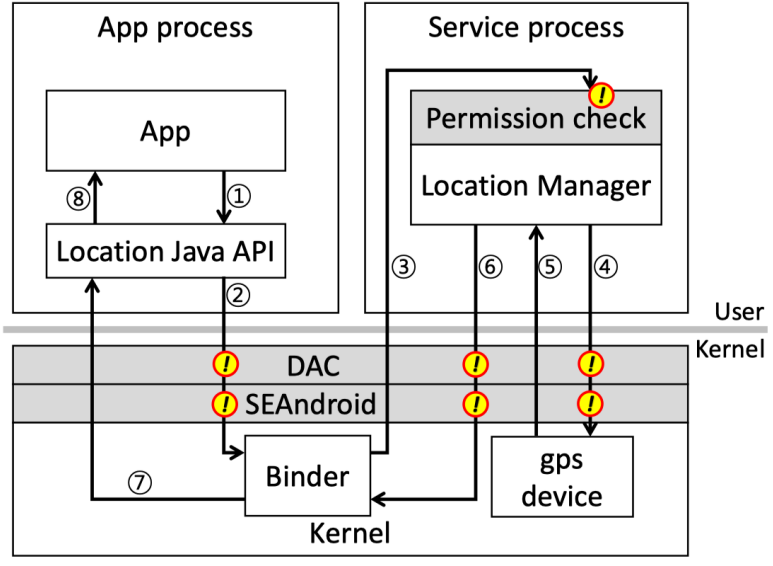


SELinux在实际应用中的问题

- **规则的设置过于复杂**
  - 不同规则之间可能存在冲突，错误的规则影响可用性
- **日志难以被理解**
  - 当发生违反规则的情况，很难解释发生了什么，该如何判定
- **应用程序不支持**
  - 部分规则需要应用程序的配合
- **性能影响**
  - 权限检查不可避免的带来性能的损失

SEAndroid的改进与应用

- **2013年引入Android**
  - SEAndroid是Android开源项目（AOSP）的一部分
  - 并默认包含在所有主流Android设备中
- **用于辅助权限检查**
  - 有助于在Android设备上强制执行应用程序沙盒边界和其他安全策略



[ACSAC'18] An Historical Analysis of the SEAndroid Policy Evolution

操作系统内核漏洞

漏洞分类的三个角度

- **漏洞类型（指攻击所利用的漏洞类型）**
  - 栈/堆缓冲区溢出、整形溢出、空指针/指针计算错误、内存暴露、use-after-free、double-free、未初始化读取、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，等等
- **攻击模块（指攻击所利用漏洞的所在的模块）**
  - 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）**
  - 执行任意代码、内存篡改、窃取数据、拒绝服务、破坏硬件等

例：Linux内核漏洞的类型

对从2010年1日到2011年3日的141个Linux内核CVE的分析

Vulnerability	Mem. corruption	Policy violation	DoS	Info. disclosure	Misc.
Missing pointer check	6	0	1	2	0
Missing permission check	0	15	3	0	1
Buffer overflow	13	1	1	2	0
Integer overflow	12	0	5	3	0
Uninitialized data	0	0	1	28	0
Null dereference	0	0	20	0	0
Divide by zero	0	0	4	0	0
Infinite loop	0	0	3	0	0
Data race / deadlock	1	0	7	0	0
Memory mismanagement	0	0	10	0	0
Miscellaneous	0	0	5	2	1
Total	32	16	60	37	2

[APSys'11] Linux kernel vulnerabilities: State-of-the-art defenses and open problems

例：Linux内核漏洞的分布

对从2010年1日到2011年3日的141个Linux内核CVE的分析

Vulnerability	Total	core	drivers	net	fs	sound
Missing pointer check	8	4	3	1	0	0
Missing permission check	17	3	1	2	11	0
Buffer overflow	15	3	1	5	4	2
Integer overflow	19	4	4	8	2	1
Uninitialized data	29	7	13	5	2	2
Null dereference	20	9	3	7	1	0
Divide by zero	4	2	0	0	1	1
Infinite loop	3	1	1	1	0	0
Data race / deadlock	8	5	1	1	1	0
Memory mismanagement	10	7	1	1	0	1
Miscellaneous	8	2	0	4	2	0
Total	141	47	28	35	24	7

[APSys'11] Linux kernel vulnerabilities: State-of-the-art defenses and open problems

操作系统的漏洞

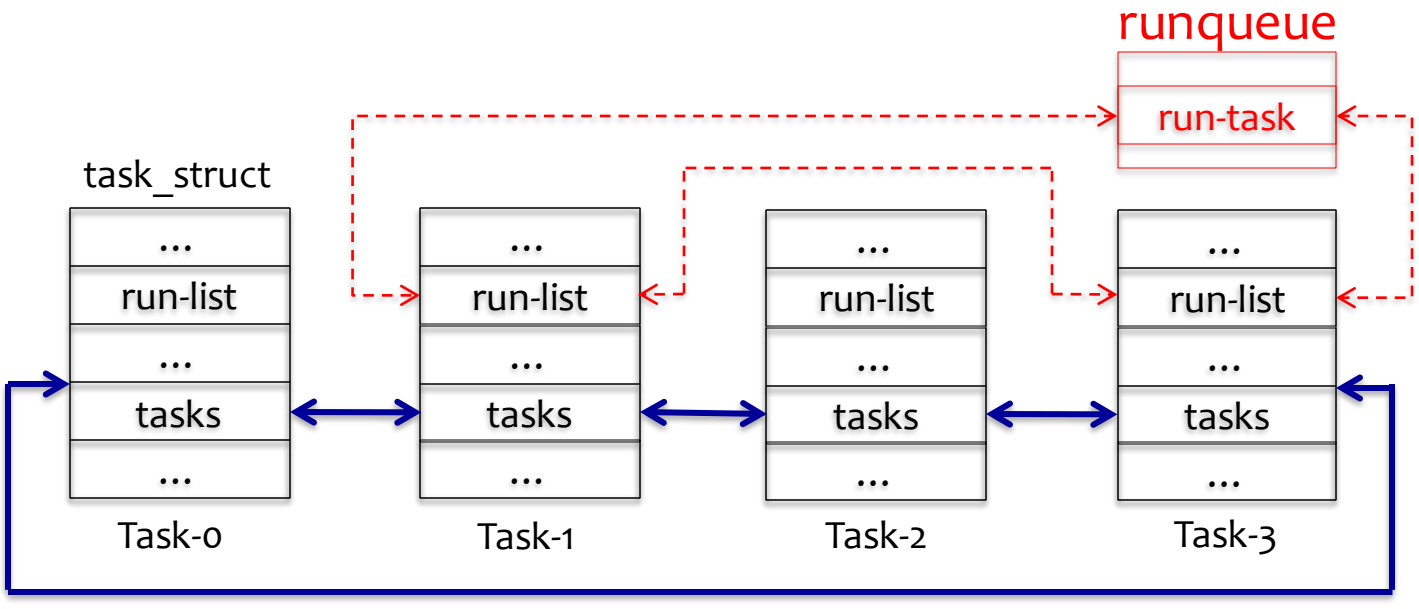
- **操作系统本身也是软件**
  - 同样存在各种漏洞，如缓冲区溢出、未初始化指针、竞争等
- **操作系统与一般应用软件不同**
  - 需要对硬件直接操作，与高级语言的抽象往往不匹配
    - 高级语言的内存安全等特性往往无法使用
  - 硬件语义的加入，为正确性判断带来了更多挑战
    - 例如对栈的操作，会使编译器失去上下文
  - 以数据表示权限等，使数据类攻击具有更强的能力
    - 例如页表的权限位，userid=0

思考问题

- 内核中哪些数据结构非常危险?

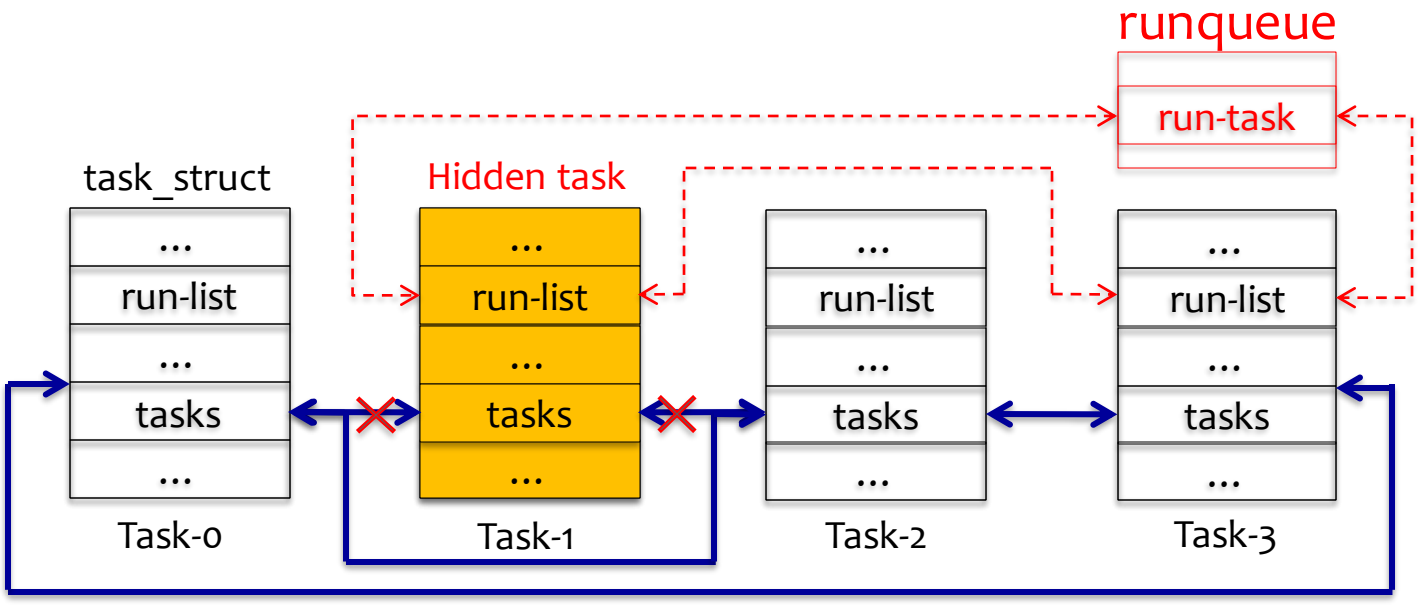
例：木马如何隐藏自己?

- **Task list** used by `*ps aux*` to show all the tasks
- **Runqueue** used by scheduler to run a task



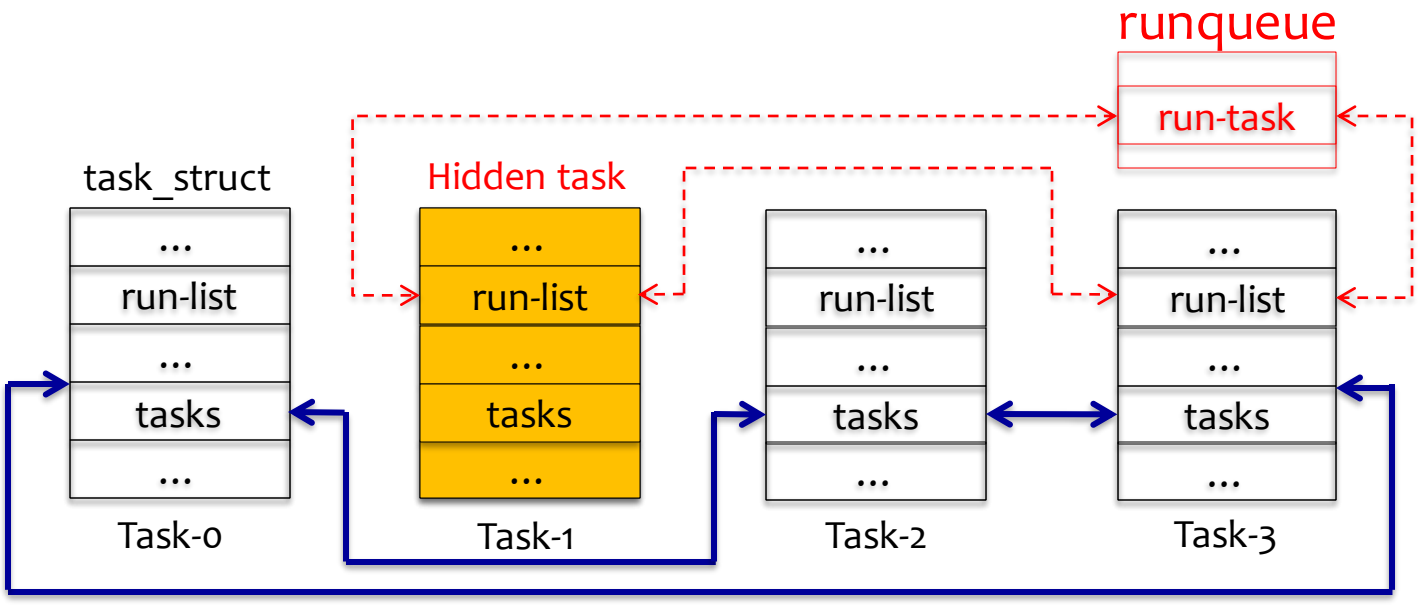
例：木马如何隐藏自己?

→ 被恶意软件使用的技术 *adore-ng*



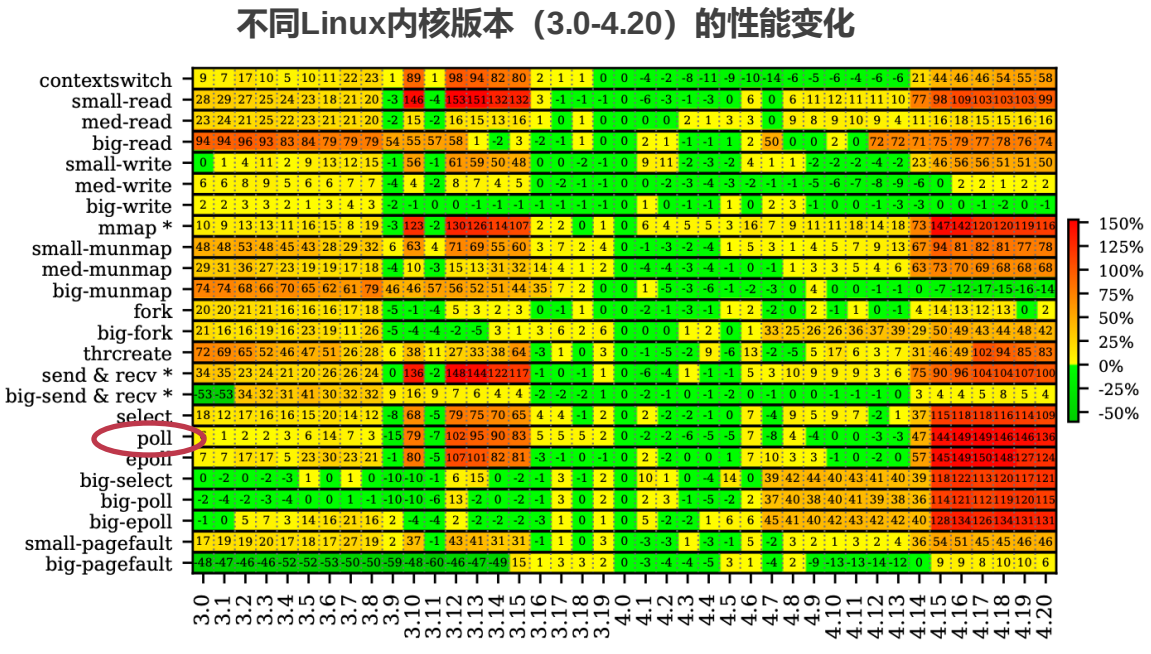
例：木马如何隐藏自己?

→ 被恶意软件使用的技术 *adore-ng*



操作系统内核攻防

操作系统出现性能不升反降的趋势



数据来源：[SOSP'19] An Analysis of Performance Evolution of Linux's Core Operations

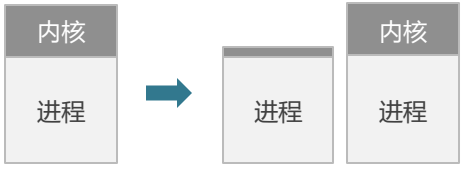
"安全税"导致的性能开销

1. **内核页表隔离（KPTI）**
2. 避免间接跳转预测执行
3. SLAB空闲列表随机化
4. 用户态内存复制安全强化

两年内 poll() 时延上升至146%

例如：内核页表隔离（KPTI）

1. 为防御2018年的熔断漏洞
2. 隔离导致性能下降达30%



Linux社区对"安全vs.性能"的态度转变

LWN.net
News from the source

Content
Weekly Edition
Archives
Search
Kernel
Security
Events calendar
Unread comments

LWN FAQ
Write for us

Edition
Return to the Kernel page

Kernel security: beyond bug fixing

By **Jonathan Corbet**
October 28, 2015

2015 Kernel Summit

As kernel security maintainer James Morris noted in the introduction to a 2015 Kernel Summit session, a lot of progress has been made with regard to kernel security in the last 10–15 years. That said, there are lot of things we could be doing better, and one could make the case that we have fallen behind the state of the art in a number of areas, including self-protection and hardening. On that note, he stepped aside and let Kees Cook give the group the bad news about what needs to be done to improve the kernel's security.

The kernel community has often been hostile to changes that increase security if they decrease usability or performance, or if they make development harder. But this particular talk led to a lot of discussion among the attendees. It would seem that the kernel development community is coming around to the idea that some sacrifices may need to be made to provide the level of security that our users need. The real test will come when the patches start to arrive; if, as Kees suggested, developers manage to avoid reflexively rejecting security patches, things will have started moving in the right direction.

... community is coming around to the idea that some sacrifices may need to be made to provide the level of security that our users need.

"为了安全，一些（性能与可用性的）牺牲是必要的"

Kees Cook

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] = ...;
```

防御方法：增加对溢出的检查代码；利用自动化工具查找并修复

Return-to-user攻击（ret2usr）

- **内核错误地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
  - 分配给应用程序后，直接映射依然存在
  - 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（前提是内核与应用在一个地址空间），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，并接收攻击者的命令
- **内核态的Rootkit**
  - 可以 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 可缓解 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

内核漏洞防御机制（一部分）

- **运行时工具**
  - SFI（Software Fault Isolation）：对内存做访问控制
  - 代码完整性保护：阻止非法代码运行
  - 用户态驱动：降低内核攻击面
  - 未初始化内存跟踪：防止未初始化内存被使用或复制到用户态
- **编译时工具**
  - 代码静态分析工具，通常需要开发者添加annotation

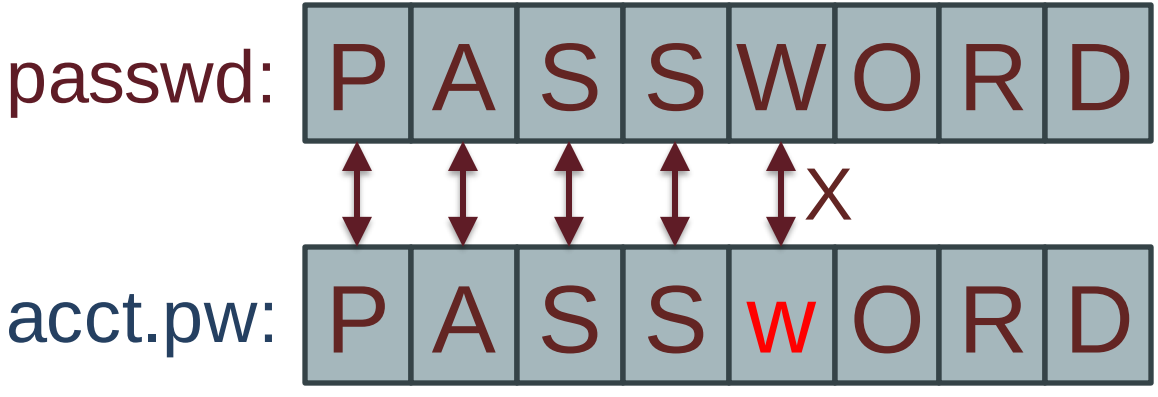
侧信道与隐秘信道

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
  - …

Review: Guessing Password (Tenex)

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



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（信道）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
    - 即被攻击者无意间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，来判断 i 的值
  - 判断方法：func_a 和 func_b 的时延
- **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

- **假设**：攻击进程和目标进程共享一块内存
- **攻击步骤**
  - 1. 攻击进程首先将 cache 清空（如：不断访问其他内存占满cache或直接flush）
  - 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
    - 前提：攻击者知道关键数据的内存地址，以及CPU上内存-cache的映射机制
  - 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）

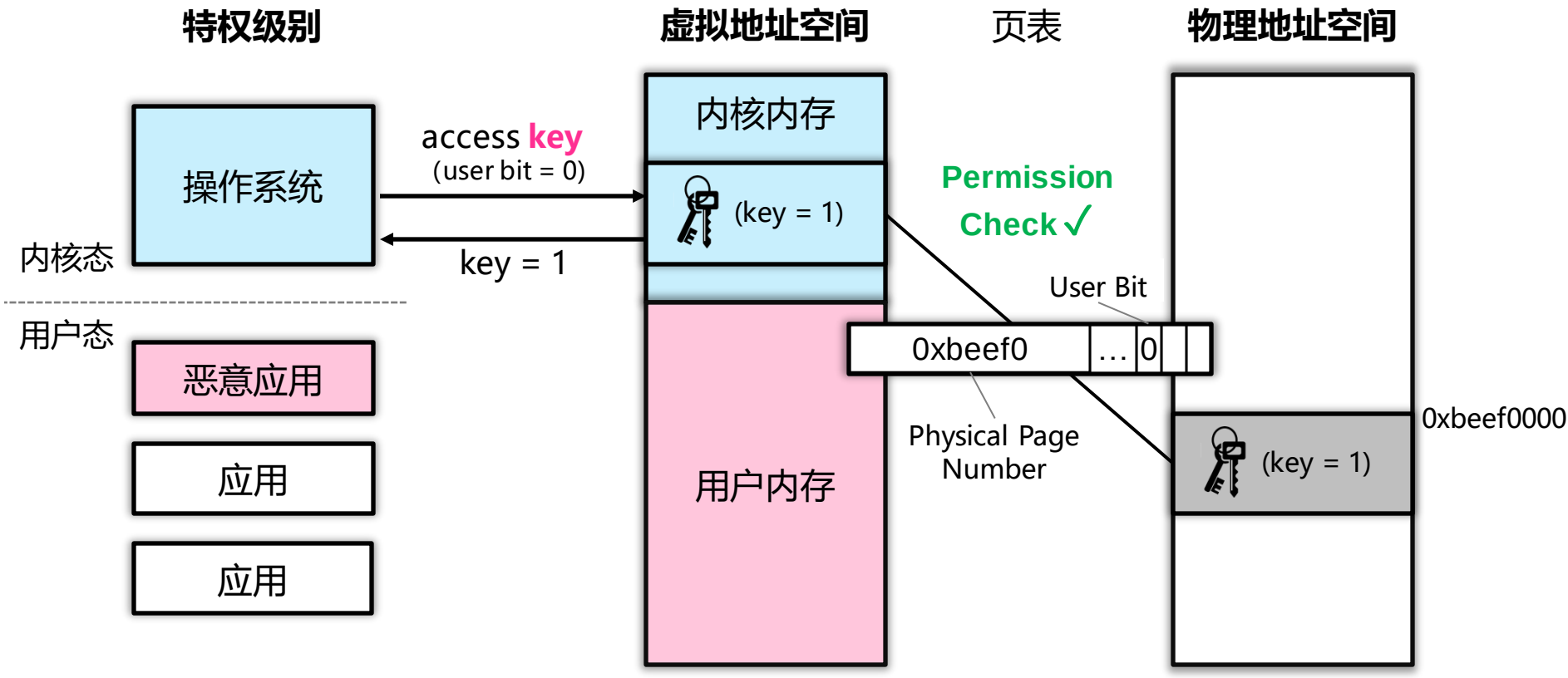
- **ORAM 将访存行为与程序执行过程解耦**
  - 攻击者即使能够观察到所有的访存请求，也无法反推出与程序执行相关的信息
- **最简单的实现：定时、定量、定位的访问方式**
  - 无论实际是否有访存需求，均以**固定周期**，访问**固定位置**，每次访问**固定的大小**
  - 例如， CPU 顺序循环访问所有的有效内存区域，程序按需获得真正想要访问的数据，若还没访问到则等待，若已经访问过了则等待下次循环
  - 类似上海和北京之间的高铁，无论乘客是谁，都按照时刻表运行，哪怕有时候位子没坐满也发车，因此根据高铁的班次并不能反推出谁坐了高铁
- **ORAM 会引入很大的额外负载**
  - 产生大量的无效内存访 问，导致有效访存的吞吐率下降
  - 访存需要等待一定的时刻，导致时延大幅度增加

案例分析：MELTDOWN

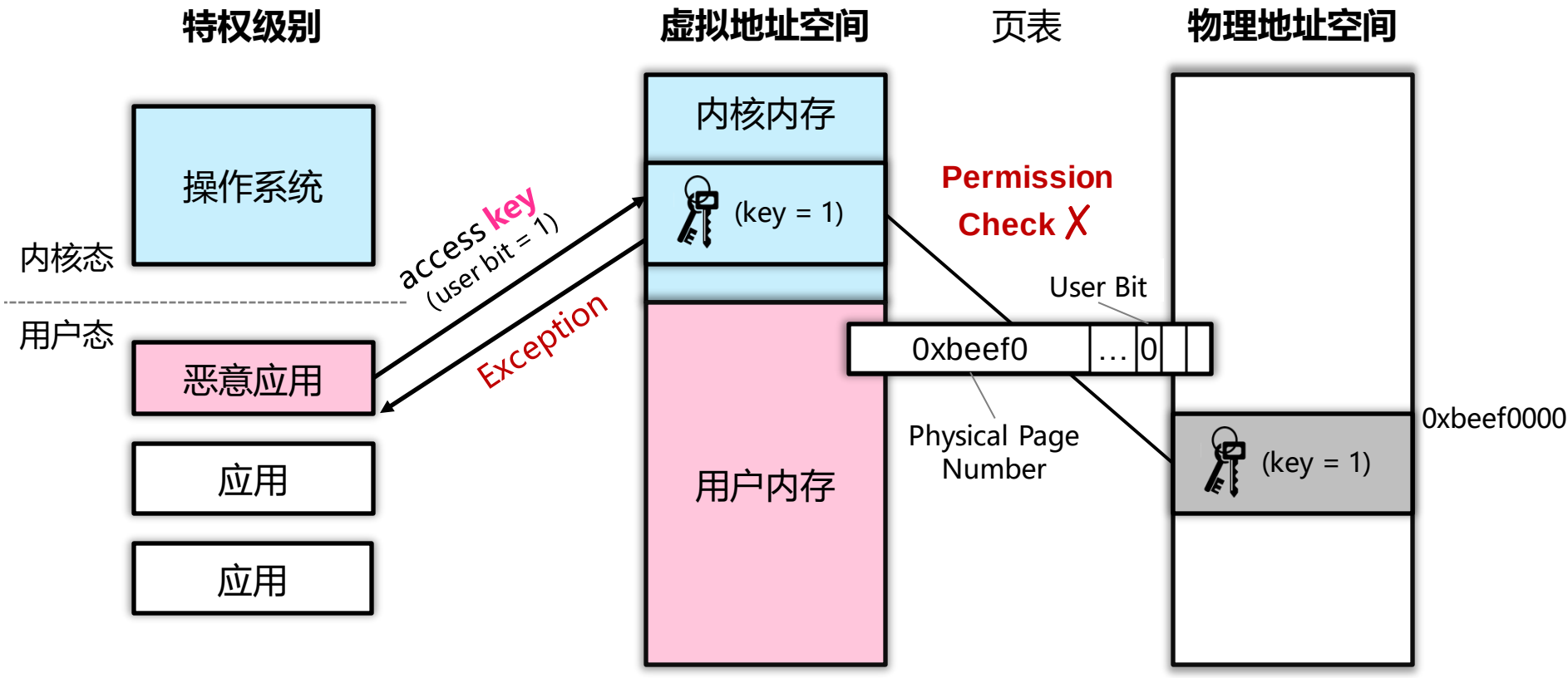
Meltdown漏洞

- **Meltdown（熔断） 漏洞**
  - 和 Spectre（幽灵） 漏洞一同被公布
  - 在 2017 年被发现，在 2018 年被公布
- **效果：允许应用程序读取任意内核内存数据**
  - 利用了 CPU 的投机执行机制
    - 漏洞利用简单、攻击效果显著
  - 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

背景-1：基于页表权限的内核数据保护



背景-1：基于页表权限的内核数据保护



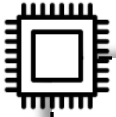
背景-2：CPU投机执行（预测执行）

- **CPU为了性能会进行投机执行**
  - 处理器内部会并发执行多条指令，无依赖关系的邻近指令在处理器内部的执行顺序会被打乱，部分指令会被提前
- **问：若提前执行了错误指令怎么办？**
  - 如前一条指令异常，处理器投机执行了后续本不应被执行的指令
- **答：执行结果丢弃/回滚**
  - 对于不应被执行的指令，处理器会丢弃/回滚其对寄存器、内存等执行状态的修改
- **复杂的CPU能确保所有状态均正确丢弃/回滚吗？不能**

Meltdown漏洞原理

- **起因：迟到的内存访问异常**
  - 在投机执行期间，跨权限的内存访问不会立刻触发异常，而是仍会继续执行后续指令
- **故障：遗漏的缓存状态回滚**
  - 当硬件抛出内存访问异常时，CPU理应回滚被错误执行指令对所有状态的修改
  - 但是，实际上CPU未回滚被错误执行指令对CPU缓存的状态修改
- **结果：应用可任意读取操作系统内存**
  - 利用缓存隐秘信道，窃取非法访问到的内核数据

Meltdown漏洞原理



Instruction

Effect

Kernel

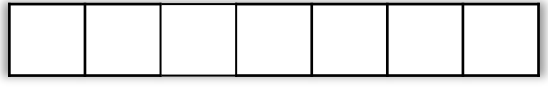
key = 0x01

User

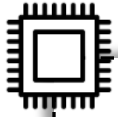
Meltdown

Cache

Memory



Meltdown漏洞原理



非法内核内存访问！

投机执行

	Instruction	Effect
①	Load **key**, %rax	%rax = **1**
②	Load **buf**[%rax], %rbx	Access **buf[1]**

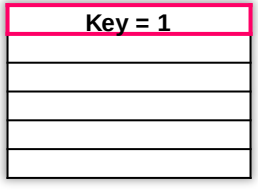
Kernel

key = 0x01

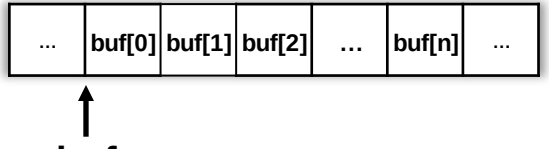
User

Meltdown

Cache

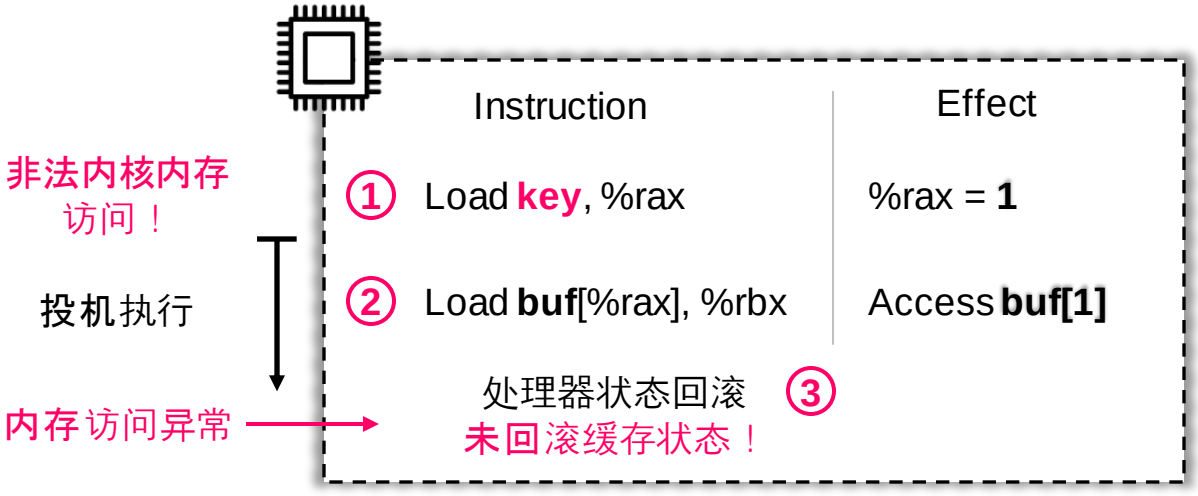


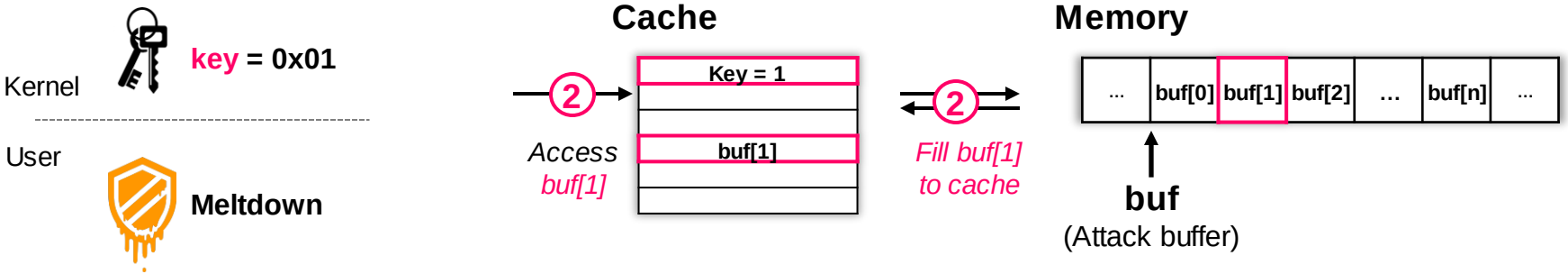
Memory



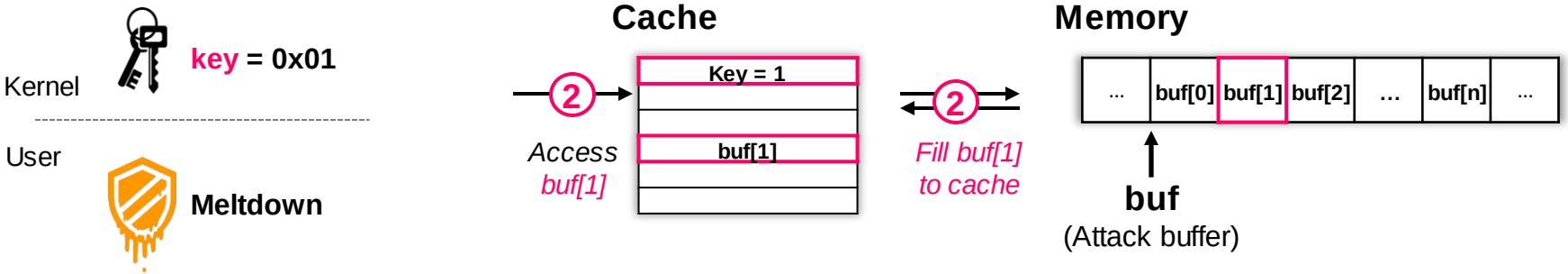
buf
(Attack buffer)

Meltdown漏洞原理



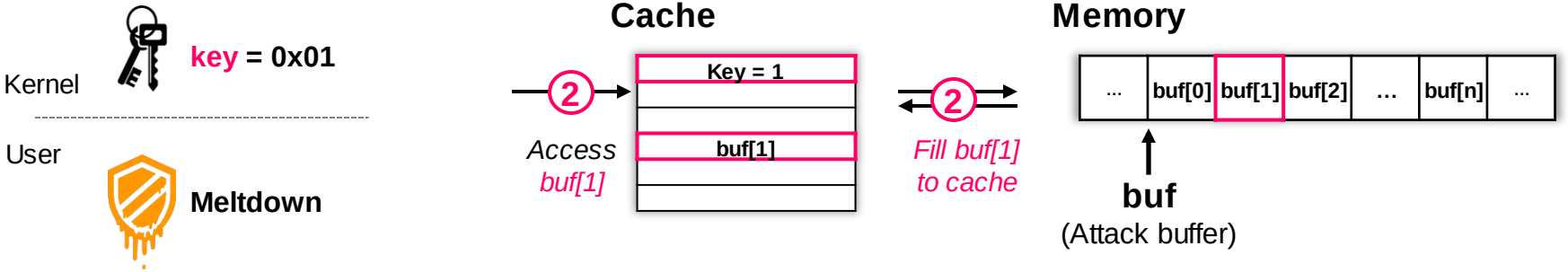


Meltdown漏洞原理



Meltdown漏洞原理

- **通过未回滚缓存状态窃取目标数据**
  - 使用目标数据作为索引访问攻击数组 buf （指令2）
    - 被访问元素会被加载到缓存中
  - 异常发生后，根据数组元素是否被缓存，窃取目标数据
    - 若buf中第 i 个元素在缓存中，则目标数据 key = i



Meltdown漏洞危害

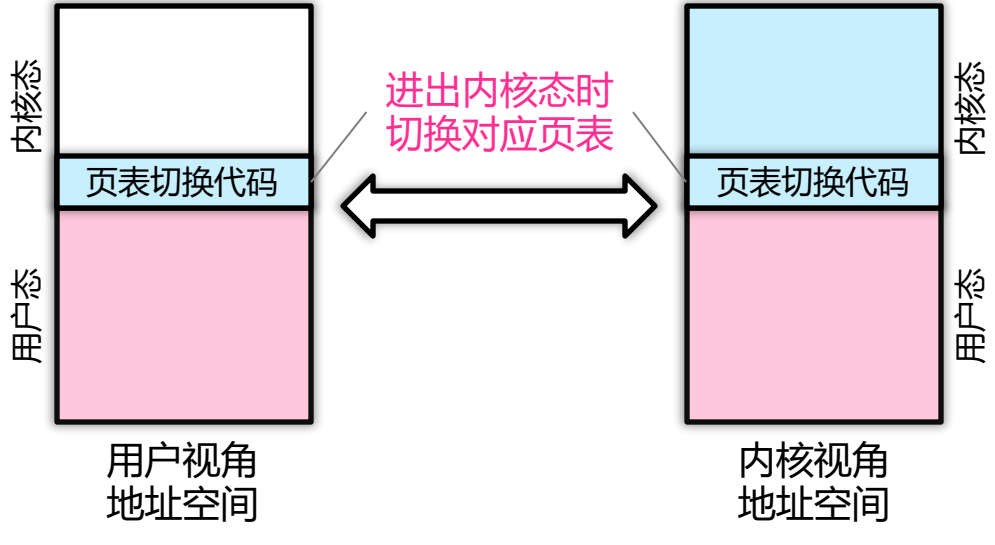
- **允许对内核内存的随意访问**
  - 用户态与内核态共享同一虚拟地址空间，仅通过页表中的权限位进行内核内存的隔离保护
  - 利用Meltdown可随意读取任意内核内存
- **允许对所有内存资源的随意访问**
  - 内核地址空间大多映射了所有物理内存
    - 如Linux中的direct map区域直接映射了整个物理内存区域
  - 利用Meltdown可对direct map区域进行读取，从而访问所有内存数据

Meltdown漏洞的软件防御方法

- **观察：Meltdown仅能够绕过页表中的用户/内核权限位**
  - Meltdown漏洞能够允许攻击者无视“内存权限”限制，访问内核内存数据
  - Meltdown漏洞无法访问“无内存映射”的内存数据
    - 如一个虚拟地址在页表中无映射，Meltdown漏洞则无数据可拿
- **思路：去除内核地址在用户态的映射**
  - 为用户态构建专用页表，将内核地址在用户态的映射去除
    - 攻击者利用Meltdown访问内核地址时，页表无法定位目标物理地址

KPTI：Linux的Meltdown漏洞防御机制

- **KPTI (Kernel Page Table Isolation)**
  - 将用户态与内核态所用页表分离
    - 用户页表：仅映射用户地址空间与部分内核地址空间
    - 内核页表：映射包括用户与内核的全部地址空间（与原有一致）

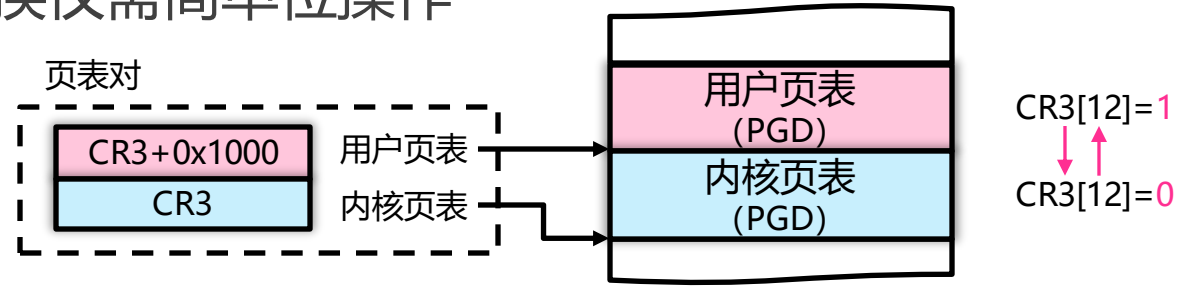


KPTI造成的性能损失

- **页表切换导致显著的性能损失**
  - 切换操作本身的时延增加
    - 切换操作：旧页表保存 -> 新页表加载 -> 修改CR3 -> 刷新TLB
    - 进出内核均需切换页表并刷新TLB，额外指令导致切换时延增长
    - 操作发生在系统关键路径
  - 切换后的运行时性能损失
    - TLB刷新导致TLB未命中率提升，直接影响程序执行性能
  - 部分应用性能降低可达30%

KPTI：性能优化方法

- **优化单一切换操作的时延**
  - 页表切换仅需简单位操作



- **优化非必要的TLB刷新**
  - 用户/内核页表使用不同的ASID，进出内核无需刷新TLB
    - ASID会作为tag进行TLB索引
    - CPU仅匹配与当前ASID一致的TLB项

总结

- **Meltdown漏洞**
  - 综合利用硬件漏洞与侧信道的全新攻击方法
    - 影响几乎所有的主流处理器（包括Intel、AMD和部分ARM）
  - 绕过页表中的内核权限，直接窃取任意内核数据
- **KPTI (Kernel Page Table Isolation)**
  - 通过分离用户态与内核态页表，进行Meltdown漏洞防御
    - Meltdown无法从无映射的虚拟地址窃取数据
  - 利用ASID等硬件特性提升性能
    - 消除内核进出时非必要的TLB刷新操作

课后练习（选做）

- **Meltdown漏洞利用**
  - 查看自己的Linux环境是否启用了KPTI
  - 关闭KPTI，尝试在自己的电脑上复现Meltdown攻击
    - 参考PoC：https://github.com/paboldin/meltdown-exploit
- **了解Spectre（幽灵）漏洞**
  - Spectre漏洞与Meltdown漏洞的原理有何异同？
  - 尝试在自己的电脑上复现Spectre攻击
    - 参考文献：“Spectre Attacks: Exploiting Speculative Execution”
    - 参考PoC：https://github.com/flxwu/spectre-attack-demo