



西安电子科技大学
XIDIAN UNIVERSITY

计算机科学与技术学院
School of Computer Science and Technology
国家示范性软件学院
National Pilot School of Software Engineering

计算机安全导论

第3章 操作系统安全

主讲人：张志为

二〇二四年秋季学期



□ 操作系统概念

- 内核、输入/输出、进程、文件系统、内存管理、虚拟机

□ 进程安全

- 信任、监控、管理、日志

□ 内存和文件系统安全

- 虚拟内存安全、基于密码的认证、访问控制、文件描述符、符号链接、快捷方式

□ 应用程序安全

- 编译和链接、缓冲区溢出攻击、格式化字符串攻击



PART 1

操作系统的概念

PART 2

进程的安全

PART 3

内存与文件系统的安全

PART 4

应用程序的安全

PART 5

练习题



PART 1

操作系统的概念

PART 2

进程的安全

PART 3

内存与文件系统的安全

PART 4

应用程序的安全

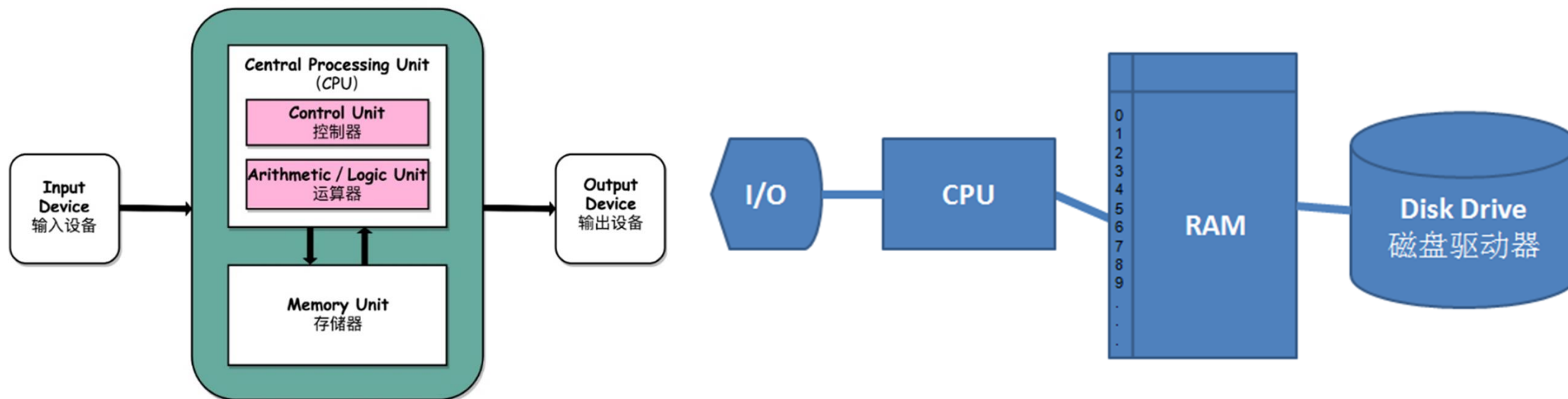
PART 5

练习题

□包括计算机硬件系统和计算机软件系统

□计算机硬件由五大部件组成：运算器、控制器、存储器、输入设备和输出设备

□硬件仅仅建立了计算机的物质基础，如果要计算机硬件发挥作用，必须配置软件系统，其中操作系统是最为重要的





- 提供计算机用户和该计算机硬件之间的接口，管理应用程序访问计算机资源的方式
 - 主要资源
 - 磁盘、CPU、主存储器、输入输出设备和网络接口等
 - 它是用户、应用程序和计算机硬件之间的“粘合剂”
 - 允许应用程序开发人员只编写程序，而无需处理底层细节
 - 多用户
 - 不同用户对计算机资源的需求与权限不同
 - 多任务
 - 需保护运行的应用程序免受其它恶意应用程序的干扰
 - 运行在同一台计算机上的应用程序即使不同时运行，也可能会访问文件系统
- 因此，操作系统应该部署适当的措施，使应用程序不能恶意或错误地破坏其他应用程序所需的资源




- 操作系统的核心组件，处理对底层硬件资源的管理
 - 底层资源
 - 内存、CPU、输入输出设备(键盘、鼠标、显示器)
 - 操作系统分层
 - 不同操作系统内核具体实现细节不同
 - 现在内核的主要分类有四类：宏内核（单内核），微内核，混合内核，外内核




<https://www.kernel.org/>

The Linux Kernel Archives

About Contact us FAQ Releases Signatures Site news



Protocol	Location
HTTP	https://www.kernel.org/pub/
Git	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

Latest Release
5.8.13 

mainline:	5.9-rc8	2020-10-04	[tarball]	[patch]	[inc. patch]	[view diff]	[browse]
stable:	5.8.13	2020-10-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff] [browse] [changelog]
longterm:	5.4.69	2020-10-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff] [browse] [changelog]
longterm:	4.19.149	2020-10-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff] [browse] [changelog]
longterm:	4.14.200	2020-10-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff] [browse] [changelog]
longterm:	4.9.238	2020-10-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff] [browse] [changelog]
longterm:	4.4.238	2020-10-01	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff] [browse] [changelog]
linux-next:	next-20201002	2020-10-02					[browse]



□ 输入输出设备

- 键盘、鼠标、视频显示卡、网卡
- 扫描仪、无线网络接口、USB接口等

□ 设备驱动程序(Device driver)

- 封装了如何与该设备进行交互的细节，完成硬件设备电子信号与操作系统及软件的高级编程语言之间的互相翻译
- 声卡播放音乐：1. 上层程序发送指令到声卡驱动程序；2. 声卡驱动程序将指令翻译成声卡能识别的电子信号；3. 声卡处理信号，播放音乐
- **与安全相关**：声辐射(Acoustic emission)、键盘记录器(Keylogger)

□ 应用程序接口(API)

- 设备驱动程序提供给应用程序的接口，允许程序与设备进行交互，交互细节由操作系统完成



- 应用程序不能直接与底层硬件进行通信，需要将任务委托给内核
 - 系统调用是一种机制，通过这种机制，应用程序可以委托内核来执行与硬件相关的操作
 - 系统调用通常是一组程序的集合，也称作库(library)
- 库函数提供操作接口，允许应用程序使用一系列预定义的API与内核进行通信
 - 例子：文件I/O（打开、关闭、读、写），运行应用程序(exec)等
- 如何实现系统调用？
 - 许多系统采用软中断(software interrupt)
 - 软中断：1. 应用程序发送请求；2. 处理器停止当前的执行流；3. 处理器切换到内核模式，执行中断要求的处理程序
 - 进入内核空间涉及直接与硬件交互，所以操作系统限制应用程序与内核交互方法和手段，以提高安全性和正确性



□进程基本概念

- 进程是正在执行的程序的一个实例
- 所以程序的实际内容存储在永久性存储器上，如磁盘，SSD等
- 程序执行时，一段程序必须加载至随机存取存储器，即内存(RAM)
- 一个程序的多个拷贝可作为不同进程来执行
- 多个进程能够同时运行依赖于对CPU的时间切片(time slicing)

任务管理器

文件(E) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	状态	8% CPU	49% 内存	1% 磁盘	0% 网络
应用 (5)					
> Adobe Acrobat DC (32 位) (3)		0%	102.5 MB	0 MB/秒	0 Mbps
> Google Chrome (10)		0.2%	405.3 MB	0 MB/秒	0 Mbps
> Microsoft PowerPoint (2)		0%	306.1 MB	0 MB/秒	0 Mbps
> Windows 资源管理器 (2)		0.3%	80.0 MB	0 MB/秒	0 Mbps
> 任务管理器 (2)		2.8%	47.7 MB	0 MB/秒	0 Mbps
后台进程 (139)					
AcroTray (32 位)		0%	0.1 MB	0 MB/秒	0 Mbps
Adobe AcroCEF (32 位)		0%	1.3 MB	0 MB/秒	0 Mbps
Adobe AcroCEF (32 位)		0%	1.3 MB	0 MB/秒	0 Mbps
Adobe AcroCEF (32 位)		0%	2.9 MB	0 MB/秒	0 Mbps
> Adobe Genuine Software Int...		0%	0.1 MB	0 MB/秒	0 Mbps
> Adobe Genuine Software Ser...		0%	1.5 MB	0 MB/秒	0 Mbps

↑ 简略信息(D) 结束任务(E)



- ❑ 用户创建一个新进程时，内核将其视为已有进程请求建立一个新进程
- ❑ 已有进程一般为shell程序或者图形用户接口程序
- ❑ 此过程中，已有进程称为父进程，被创建的新进程为子进程
- ❑ 这些进程在操作系统中被组织为一颗有根的树，即进程树
- ❑ 这棵树的根是进程init，在加载和运行内核后开始执行，并创建登录会话和系统任务等子进程

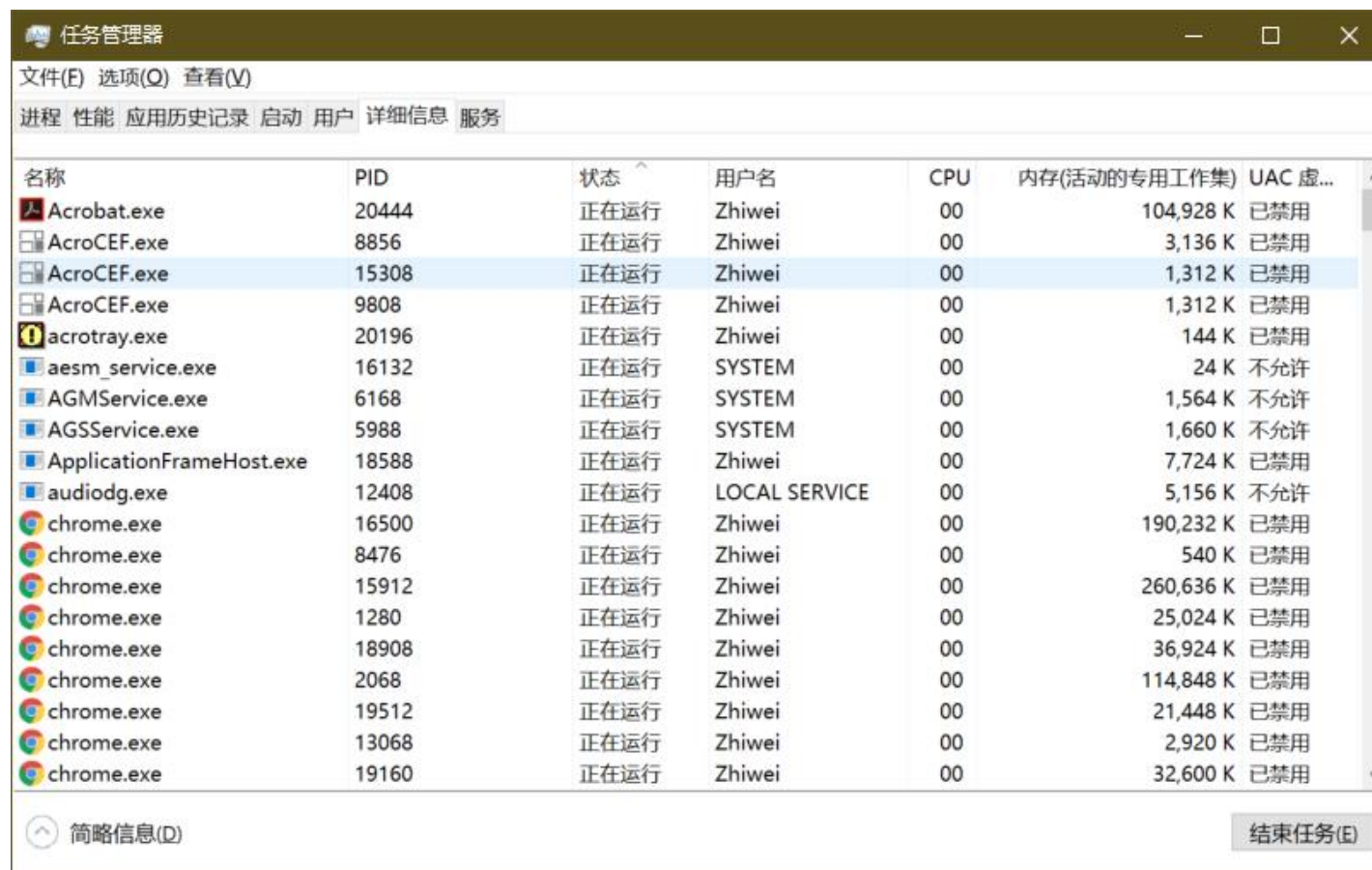
```

1  qingsong@db2a:/tmp$ pstree
2  init--+-atd
3      |-cron
4      |-db2fmc
5      |-db2syscr--+-db2fmp---4*[{db2fmp}]
6          |       |-db2fmp---3*[{db2fmp}]
7          |       |-db2sysc---13*[{db2sysc}]
8          |       |-3*[{db2syscr}]
9          |       |-db2vend
10         |       `--[{db2syscr}]
11     |-dbus-daemon
12     |-dhclient
13     |-dockerd--+-docker-containe---7*[{docker-containe}]
14         |       `--9*[{dockerd}]
15     |-5*[getty]
16     |-login---bash
17     |-master--+-pickup
18         |       `--qmgr
19     |-mdadm
20     |-rpc.idmapd
21     |-rpc.mountd
22     |-rpc.statd
23     |-rpcbind
24     |-rsyslogd---3*[{rsyslogd}]
25     |-sshd--+-sshd---sshd---bash--+-bash---command-not-fou
26         |   |                               |-pstree
27         |   |                               `--sudo
28         |   `--sshd---sshd---bash
29     |-sudo---su---bash---mysqld---27*[{mysqld}]

```



- 进程是正在执行的程序的一个实例一个给定计算机上运行的每个进程都由唯一的非负整数来标识，称为进程ID(PID)
- 给定一个进程的PID，可以关联其CPU运行时间、内存消耗、用户ID、程序名称等



任务管理器

文件(F) 选项(O) 查看(V)

进程 性能 应用历史记录 启动 用户 详细信息 服务

名称	PID	状态	用户名	CPU	内存(活动的专用工作集)	UAC 虚...
Acrobat.exe	20444	正在运行	Zhiwei	00	104,928 K	已禁用
AcroCEF.exe	8856	正在运行	Zhiwei	00	3,136 K	已禁用
AcroCEF.exe	15308	正在运行	Zhiwei	00	1,312 K	已禁用
AcroCEF.exe	9808	正在运行	Zhiwei	00	1,312 K	已禁用
acrotray.exe	20196	正在运行	Zhiwei	00	144 K	已禁用
aesm_service.exe	16132	正在运行	SYSTEM	00	24 K	不允许
AGMSservice.exe	6168	正在运行	SYSTEM	00	1,564 K	不允许
AGSService.exe	5988	正在运行	SYSTEM	00	1,660 K	不允许
ApplicationFrameHost.exe	18588	正在运行	Zhiwei	00	7,724 K	已禁用
audiodg.exe	12408	正在运行	LOCAL SERVICE	00	5,156 K	不允许
chrome.exe	16500	正在运行	Zhiwei	00	190,232 K	已禁用
chrome.exe	8476	正在运行	Zhiwei	00	540 K	已禁用
chrome.exe	15912	正在运行	Zhiwei	00	260,636 K	已禁用
chrome.exe	1280	正在运行	Zhiwei	00	25,024 K	已禁用
chrome.exe	18908	正在运行	Zhiwei	00	36,924 K	已禁用
chrome.exe	2068	正在运行	Zhiwei	00	114,848 K	已禁用
chrome.exe	19512	正在运行	Zhiwei	00	21,448 K	已禁用
chrome.exe	13068	正在运行	Zhiwei	00	2,920 K	已禁用
chrome.exe	19160	正在运行	Zhiwei	00	32,600 K	已禁用

^ 简略信息(D) 结束任务(E)



为了管理共享资源，进程间必须进行通信

- 进程间通信方法1：使用读写文件来传递信息
 - 文件处理需要从磁盘读写文件，与RAM比慢很多
 - 如果两个进程间希望进行私密通信，使用文件传递消息会比较麻烦
- 进程间通信方法2：共享同一块物理内存区，通过共享的RAM来传递消息
 - 只要内核能得当地管理共享内存空间，这种方法就会快速高效
- 进程间通信方法3：管道(pipe)和套接字(socket)
 - 提供一个进程到另一个进程的隧道
- 进程间通信方法4：信号(signal)，进程间发送异步消息
 - 本质上是一个进程发送给另一个进程的通知
 - 例如Linux系统中，命令窗口输入Ctrl+ C可以向进程发送INT信号，终止该进程



远程过程调用(Remote procedure call, RPC)

- Windows系统下类似于信号机制的方法
- 本质上允许一个进程调用另一个进程中的子例程
- 例如，Windows使用内核级的TerminateProcess()来杀死进程，任何进程可以调用它

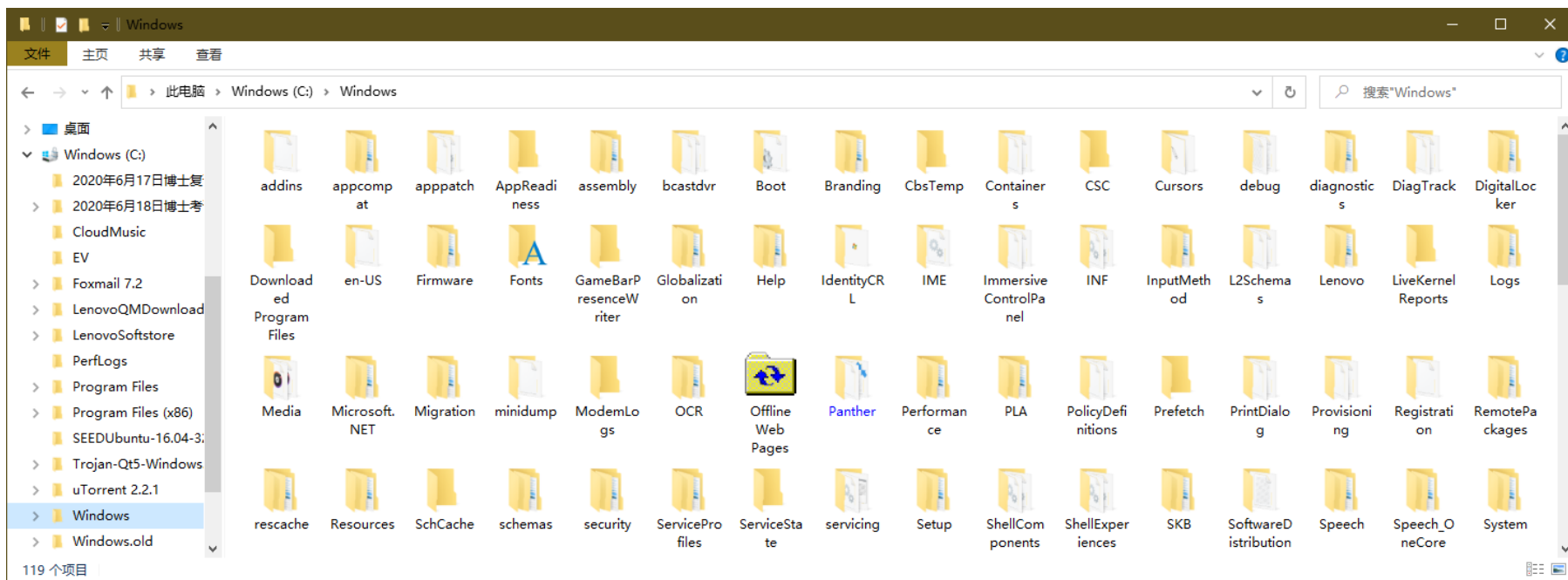
守护进程(Daemon)和服务(Service)

- Linux系统中称后台进程为守护进程(Daemons)
- 守护进程运行权限高于任何用户，并在登录会话结束前一直存在
- 守护进程示例：Web服务器、远程登录、打印服务器进程等
- Windows下与守护进程类似的称为服务(service)



文件系统File system

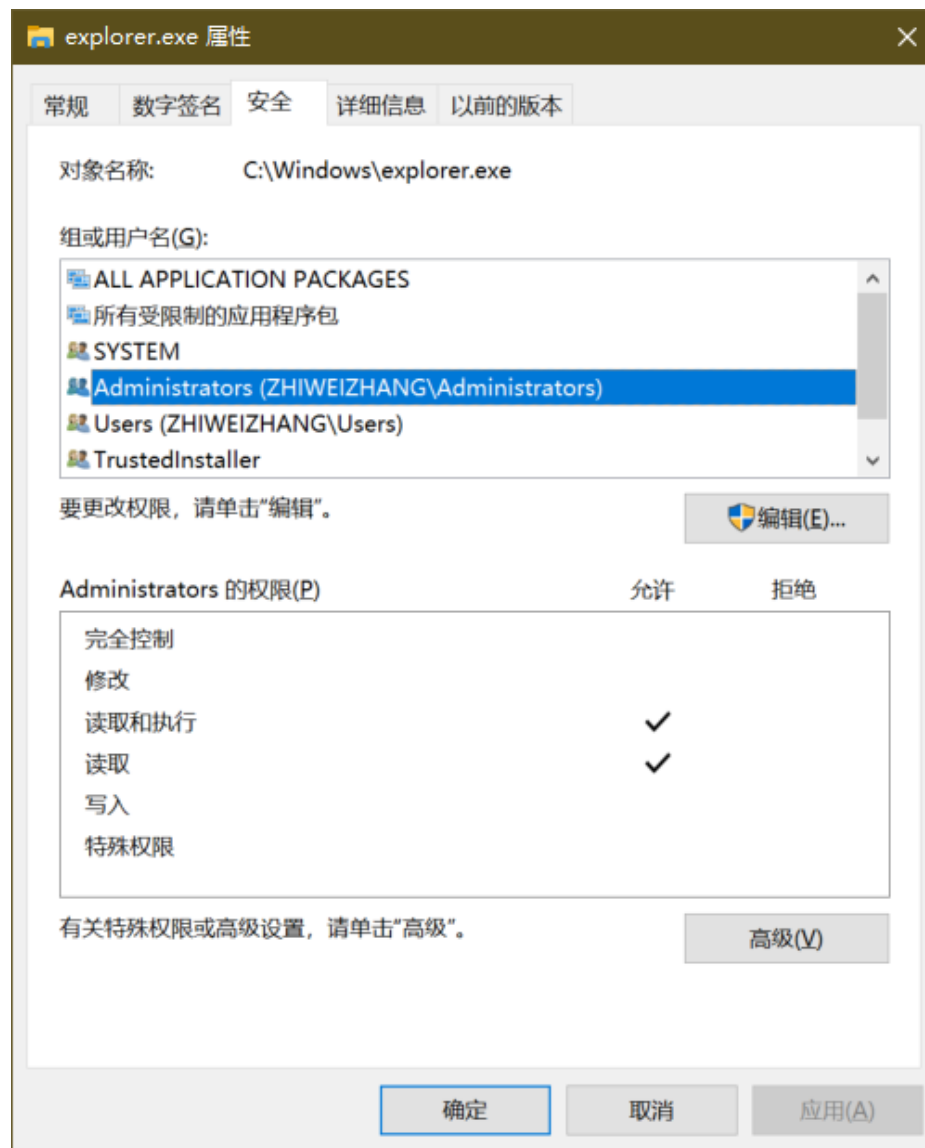
- 文件系统是如何组织计算机的外部、非易失性存储器的一种抽象
- 操作系统通常将文件系统组织成由文件夹构成的层次结构
- 每个文件夹下可以包含文件和子文件夹
- 卷或者驱动器由嵌套的文件夹集合组成，形成树形结构





文件权限File permission

- ❑ 操作系统安全的一个主要问题是如何界定哪些用户可以访问哪些资源，也就是说，哪个用户可以读取文件、写入数据和执行程序
- ❑ 这一概念封装在文件权限之中，它的具体实现取决于操作系统
- ❑ 磁盘上的每类资源，包括数据文件和程序，都有一个权限与之相关联
- ❑ 操作系统会检查文件权限，以确认用户(组)是否可以读、写、执行一个特定的文件
- ❑ 一些类UNIX操作系统的使用文件权限矩阵标示哪些用户可以对哪些文件进行何种操作
- ❑ 共有三类权限，其中每一类都是位的组合。文件的所有者对应着用户的uid，组对应着组id



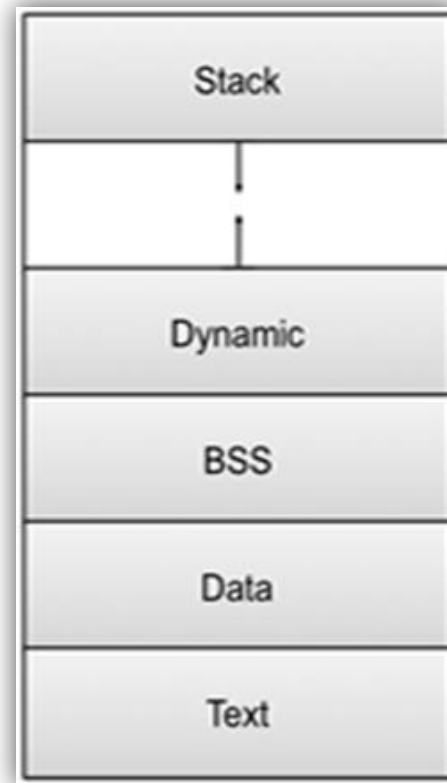


- ❑ 第一个是owner类，它决定了文件创建者的权限
- ❑ 第二个是group类，它决定了相同组中的用户的文件权限
- ❑ 第三个是others类，它决定了既不是所有者的文件，也不是相同组中用户的文件权限
- ❑ 每一类都有一系列位，用于确定具有的权限。
 - ❑ 第一位是read位，其允许用户读取文件
 - ❑ 第二位是write位，其允许用户改变文件的内容
 - ❑ 最后一位execute位，其允许用户运行程序或脚本

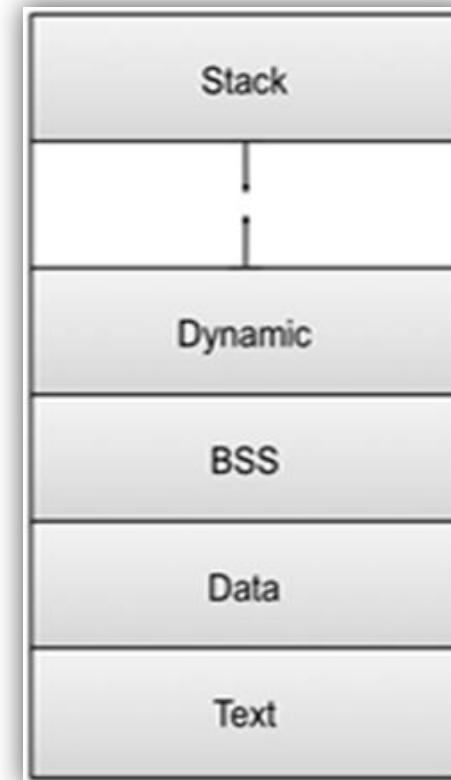
```
rodan:~/java % ls -l
total 24
-rwx rwx rwx 1 goodrich faculty 2496 Jul 27 08:43 Floats.class
-rw- r-- r-- 1 goodrich faculty 2723 Jul 12 2006 Floats.java
-rw----- 1 goodrich faculty 460 Feb 25 2007 Test.java
rodan:~/java %
```



- 内存管理是指如何组织和分配计算机的内存
- 进程执行时，需分配一块内存(地址空间)来存储其程序代码、数据和进程执行过程中需要的存储空间
- 一个进程的地址空间分为五个部分
 - 文本：包含程序的代码
 - 数据：包含已初始化的静态程序变量
 - BSS：符号起始区块，包含未初始化的静态变量
 - 堆：动态段，存储进程执行期间所产生的数据
 - 栈：向下生长的堆栈数据结构，记录函数调用结构和它们的参数



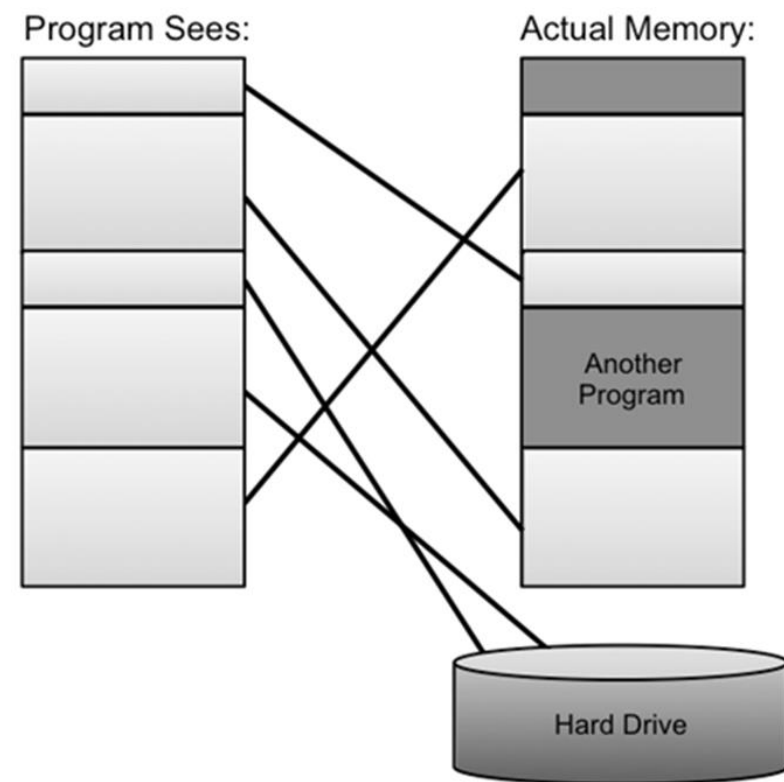
- ❑ 五段内存区中，每一段都有自己的访问权限，由操作系统执行这些权限
- ❑ 文本区通常是只读的，其余区段可以写入
- ❑ 对操作系统安全至关重要的一点：不允许进程访问其它进程的地址空间
- ❑ 加强地址空间边界保护能够避免许多安全问题
- ❑ 操作系统一般将地址空间分为两个区：用户空间和内核空间
- ❑ 内核空间是为操作系统内核功能保留的特殊空间，有着最为严格的访问特权





虚拟内存Virtual memory

- ❑ 如果每个进程的地址空间都是一个连续的内存块，那么对于使用了大型数组等的程序则需要很大块的连续地址空间，这样内存使用效率会很低
- ❑ 如果进程所需的连续地址空间总量超出了计算机的内存总量，则获取连续内存区是不可能的
- ❑ 如何解决上述两个问题？——使用虚拟内存
 - ❑ 每个进程都有一个虚拟地址空间，虚拟内存系统将每个虚拟地址映射为实际的内存地址；当访问虚拟地址时，内存管理单元(MMU)硬件组件会查找虚拟地址映射的实际内存地址；对于进程来说，看到的虚拟地址空间是连续的，但实际内存地址空间是不连续的
 - ❑ 另一个优点是，允许正在执行进程的总地址空间大小大于计算机的实际主存储器大小，主要原因是虚拟内存系统可以使用外部驱动器的一部分来存放执行进程不常使用的内存块
 - ❑ 主要缺点：虚拟内存访问硬盘驱动器要慢于访问RAM。一般情况下访问硬盘比访问主存慢10000倍



- ❑ 为了使大部分要访问的内存块在主存而不是在硬盘中，操作系统用硬盘驱动器来存储当前不需要的内存块
- ❑ 如果地址空间在长时间内都没有被访问，则可能进行页换出，并将页换出写入磁盘。当进程试图访问驻留在换出页中的虚拟地址时，将触发缺页
- ❑ 当出现缺页时，虚拟内存系统中的另一部分：分页管理器在硬盘驱动器上查找所需的内存块，将其读回内存，更新物理地址和虚拟地址之间的映射，并换出其他的未使用的内存块
- ❑ 处理缺页的这种换入换出机制允许操作系统运行的进程所需的总内存要大于可用的RAM存储器的存储大小

1. Process requests virtual address not in memory, causing a page fault. 进程请求不在内存中的虚地址，产生缺页



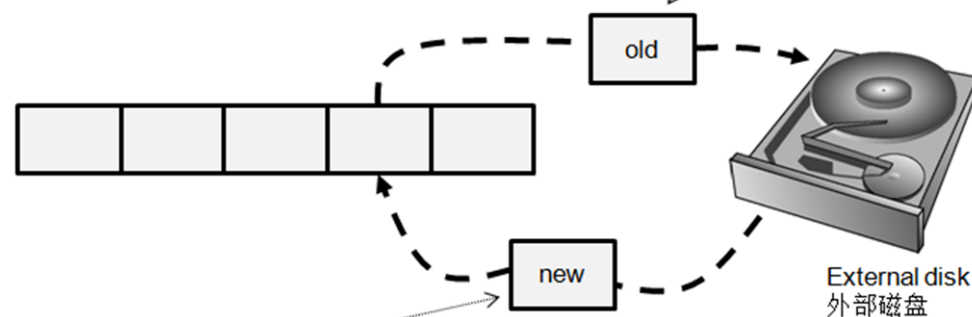
"read 0110101"

"Page fault,
let me fix that."
缺页 等我来修复它



2. Paging supervisor pages out an old block of RAM memory. 分页管理程序换出RAM中的就内存

Blocks in
RAM memory:
RAM内存中的块



3. Paging supervisor locates requested block on the disk and brings it into RAM memory. 分页管理程序在磁盘上定位所请求的块，并将其换入RAM内存



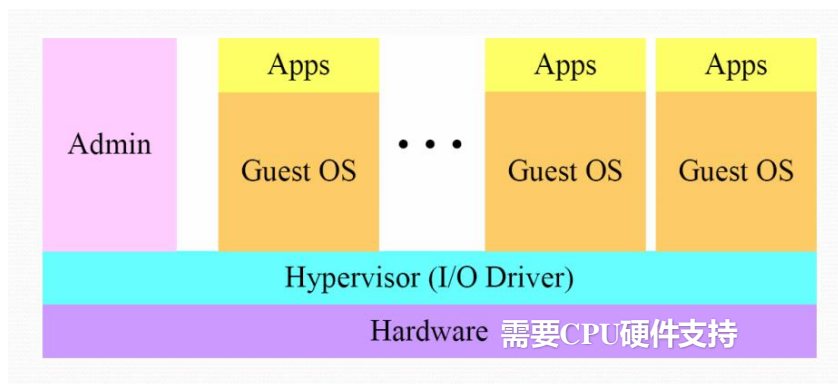
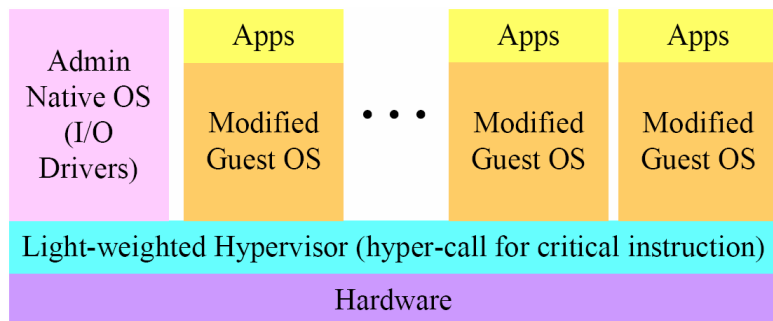
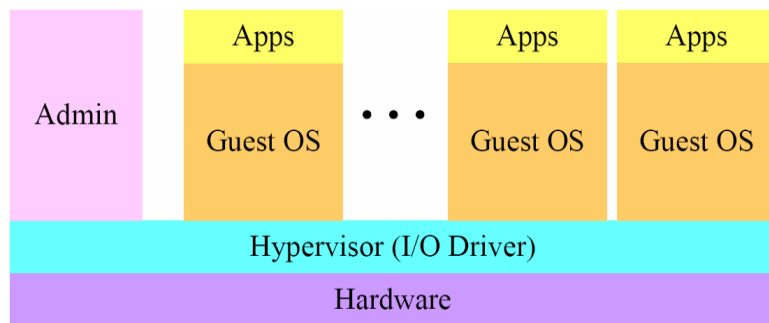
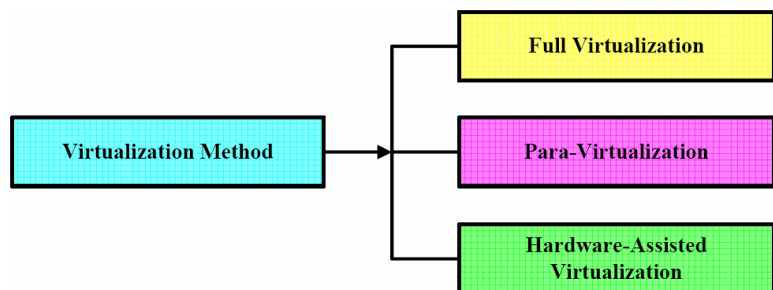
虚拟化的类型

- 全虚拟化
- 半虚拟化
- 硬件辅助虚拟化

	利用二进制翻译的全虚拟化 [❏]	硬件辅助虚拟化 [❏]	半虚拟化 [❏]
实现技术 [❏]	BT 和直接执行 [❏]	遇到特权指令转到 root 模式执行 [❏]	Hypercall [❏]
客户操作系统修改 [❏]	无需修改操作系统，最佳兼容性 [❏]	无需修改操作系统，最佳兼容性 [❏]	客户操作系统需要修改来支持 Hypercall，因此它不运行在物理硬件本身或其他的 hypervisor 上，兼容性差 [❏]
性能 [❏]	差 [❏]	全虚拟化下，CPU 需要在两种模式之间切换，带来性能开销，但是其性能在逐渐逼近半虚拟化 [❏]	好，半虚拟化下 CPU 性能开销几乎为 0，虚机的性能接近于物理机 [❏]
应用产商 [❏]	VMware Workstation/QEMU/Virtual PC [❏]	VMware ESXi/Microsoft Hyper-V/Xen 3.0/KVM [❏]	Xen [❏]

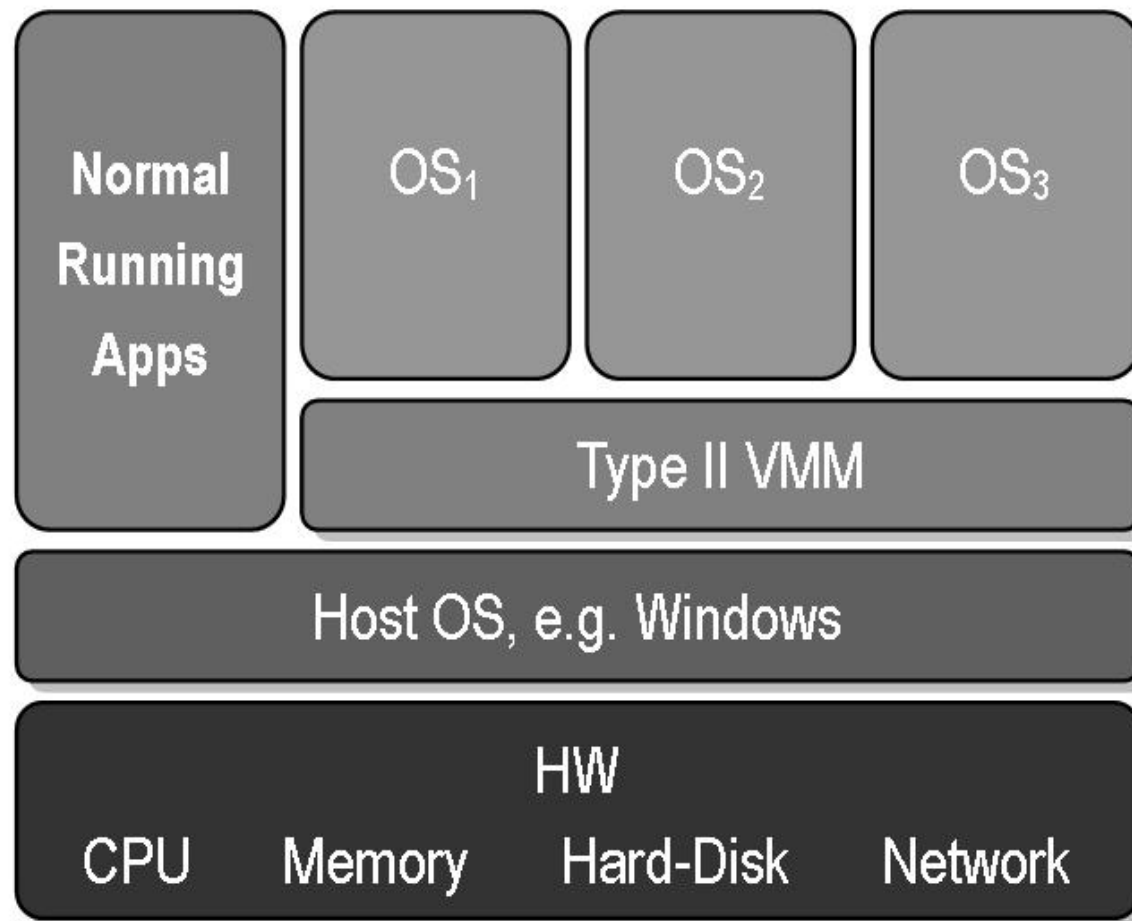
虚拟化的对象

- 硬件虚拟化
 - CPU虚拟化
 - 内存虚拟化
 - 网络虚拟化
 - GPU虚拟化
- 软件虚拟化
 - 指令集虚拟化
 - 操作系统虚拟化
 - 虚拟机
 - 容器
 - 编程语言虚拟化
 - 库函数虚拟化





- ❑ 虚拟机不直接接触底层硬件就允许操作系统的运行
- ❑ 虚拟机软件会创建操作系统交互的模拟环境，软件层提供的这种环境称为管理程序(hypervisor)或者虚拟机管理器(VMM)
- ❑ 在虚拟机中运行的操作系统称为客户机，本地的操作系统称为主机





❑ 虚拟机实现：模拟(emulation)和虚拟化(Virtualization)

- ❑ 模拟：主机操作系统模拟与客户机操作系统交互的虚拟接口。主机系统翻译通过这些接口的通信，并最终传递给硬件
- ❑ 模拟的优点：更多的硬件灵活性，例如可以在一个处理器的计算机上模拟一个虚拟环境支持另一个处理器；缺点：性能下降
- ❑ 虚拟化：不需要模拟的转换过程，性能提升，但损失了硬件灵活性

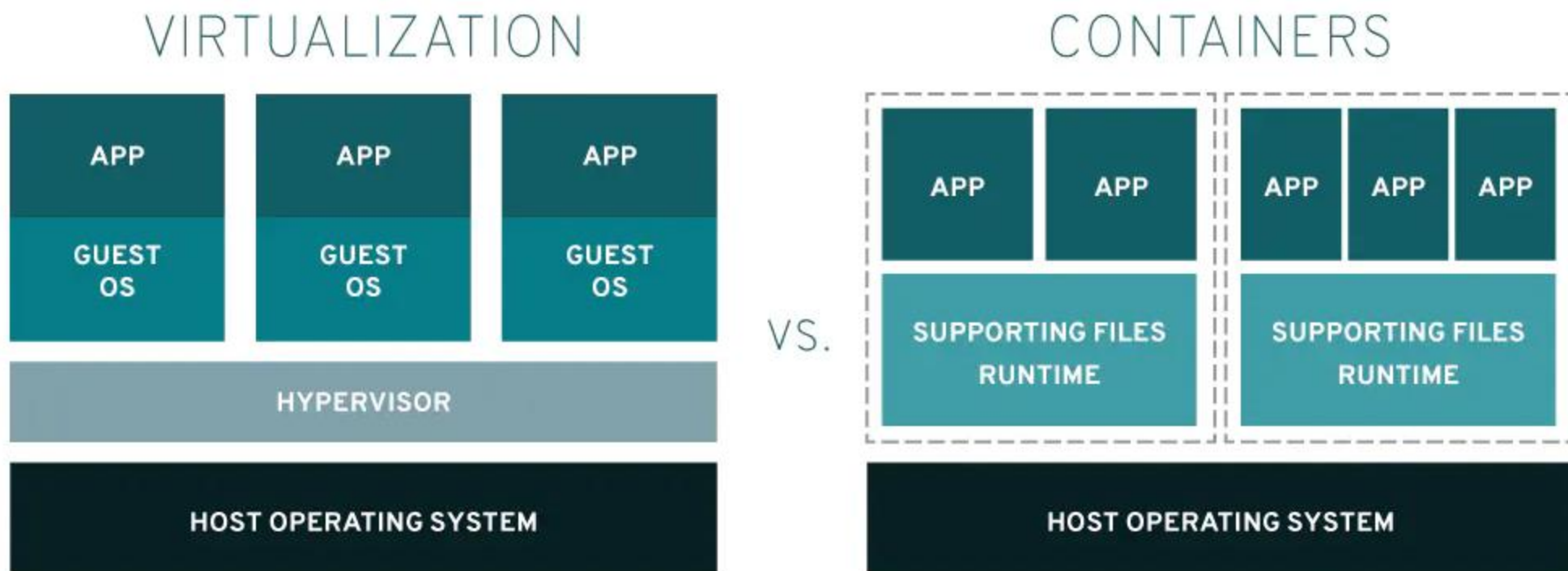
❑ 虚拟机优点

- ❑ 硬件效率：同一台计算机上托管多个操作系统
- ❑ 可移植性：将客户机操作系统保存快照在另一台计算机上运行
- ❑ 安全：沙箱，客户机操作系统被攻击时保护其他系统的正常运行
- ❑ 管理便捷性：定期保存快照，可以快速恢复受感染前的系统



容器的本质就是一组受到资源限制，彼此间相互隔离的进程。总的来说，容器是一种基于操作系统能力的隔离技术，这和基于hypervisor的虚拟化技术复杂度不可同日而语。

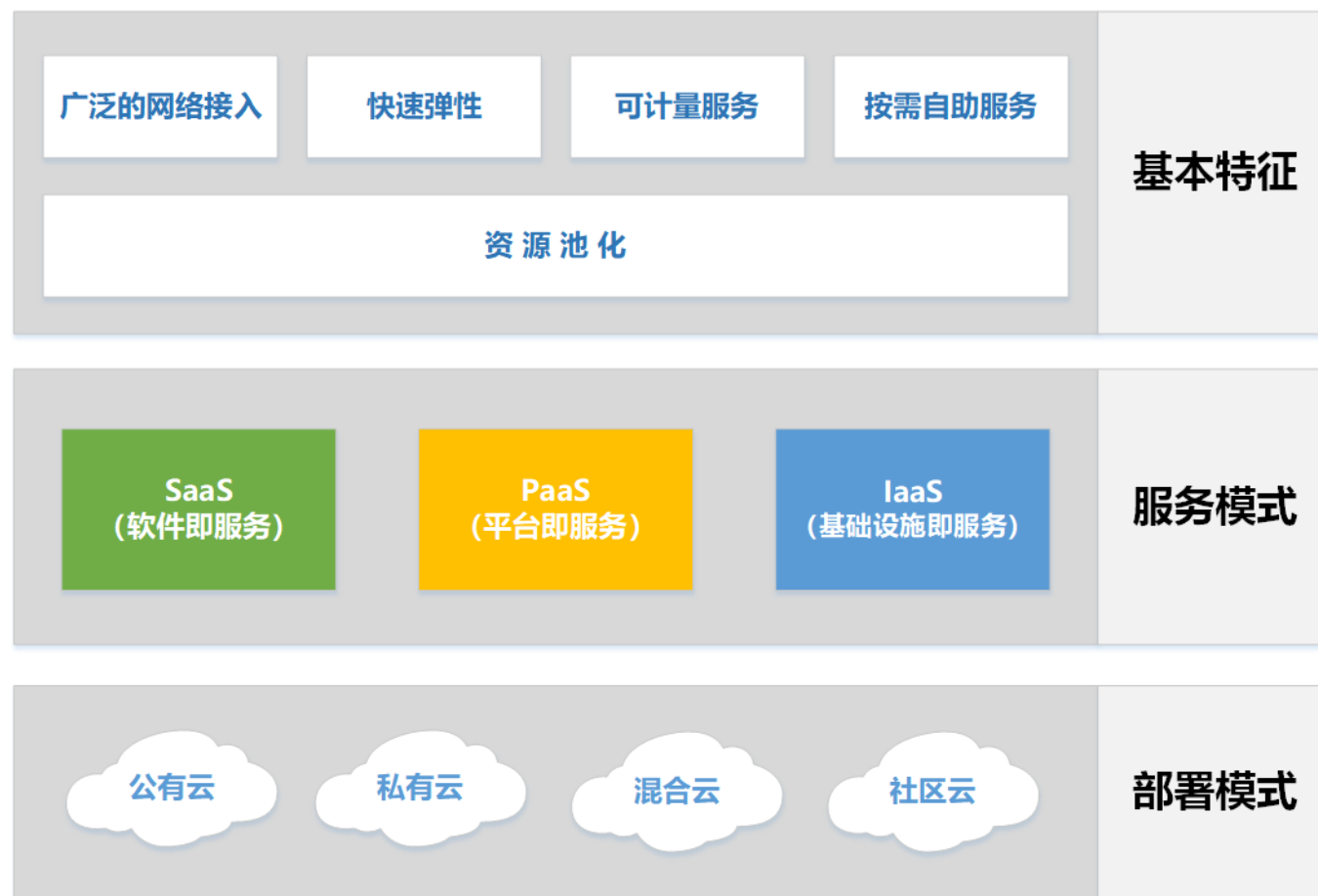
容器没有自己的OS，直接共享宿主机的内核，也没有hypervisor这一层进行资源隔离和限制，所有对于容器进程的限制都是基于操作系统本身的能力来进行的。由此，容器获得了一个很大的优势：轻量化，占用资源小，镜像文件也要比虚拟机小的多。





云计算是一种将可伸缩、弹性、共享的物理和虚拟资源池以按需自服务的方式供应和管理，并提供网络访问的模式

- 随着云计算的快速发展和推广应用，越来越多的组织和企业开始把业务系统和数据迁移到云平台上
- 云平台中也潜藏着诸多安全隐患，云数据泄漏事件层出不穷，影响深远



推荐阅读: Above the Clouds: A Berkeley View of Cloud Computing



PART 1 | 操作系统的概念

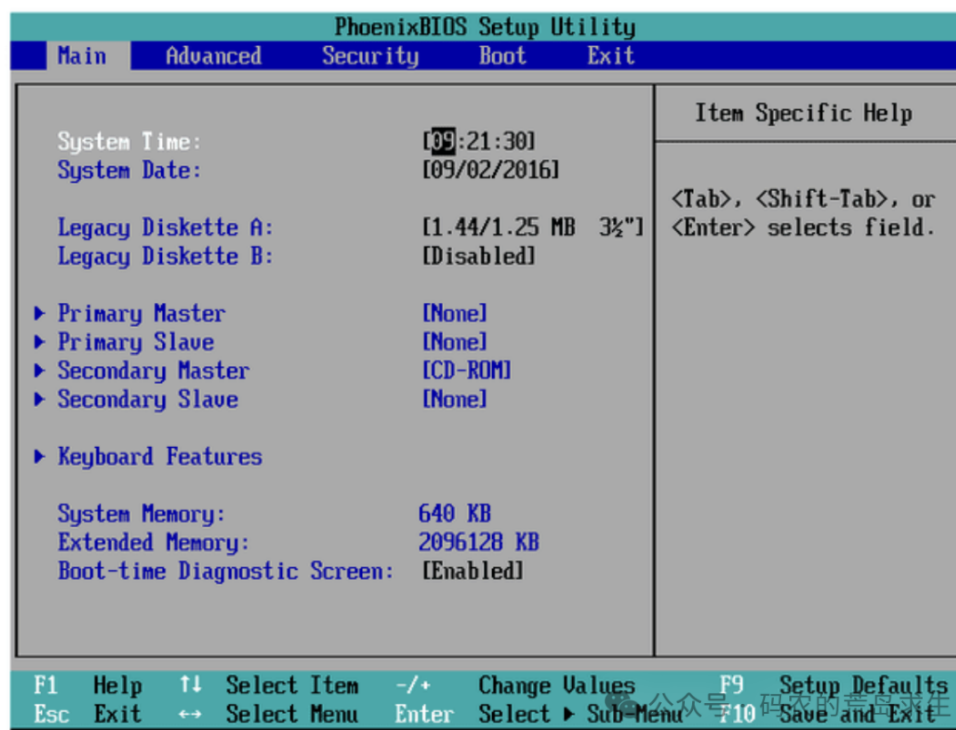
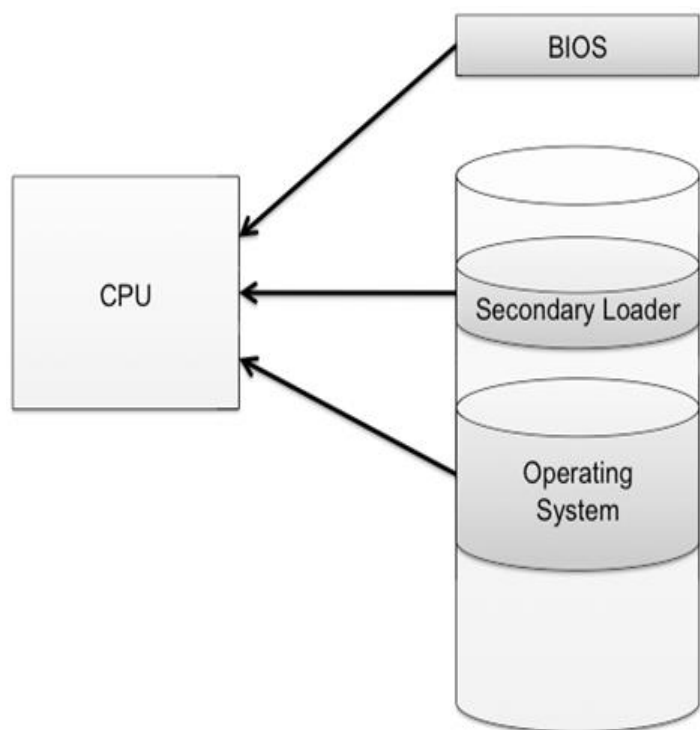
PART 2 | 进程的安全

PART 3 | 内存与文件系统的安全

PART 4 | 应用程序的安全

PART 5 | 练习题

- ❑ 从关机状态到将操作系统加载到内存的操作称为引导
- ❑ 当按下计算机的启动按钮后，它首先执行的存储在基本输入/输出系统(BIOS)固件组件中的代码；BIOS将第二阶段的引导加载程序加载到内存，该程序会将操作系统其余部分加载到内存，然后将控制权交给操作系统
- ❑ 恶意用户可以利用引导过程的一些脆弱点来通过控制计算机的执行；可以通过设置BIOS密码来防止恶意用户引导第二阶段的加载程序



扩展阅读：操作系统是怎么一步步启动起来的？

<https://mp.weixin.qq.com/s/P8aK376Koq4j4E-GTWtUsw>



- 第二阶段引导程序允许用户指定应该使用哪个设备来加载操作系统
- 默认选项是从硬盘驱动器引导；也可能从外部介质如DVD驱动器或者引导U盘引导
- **从安全角度考虑，应该确保操作系统是从值得信任的介质启动**
- BIOS系统会提供一个引导的层次结构，它可以决定引导设备的优先级
- 攻击者可以绕过运行的计算机上的安全机制，从外部介质引导其他的操作系统来访问硬盘
- 防止这类攻击的有效措施：对第二阶段引导程序的启动设置了密码保护，只允许授权用户从外部存储介质引导计算机启动



- ❑ 现代计算机系统都有进入电源关闭状态的功能，这种状态称为休眠
- ❑ 当进入休眠时，操作系统将计算机内存的全部内容都存储到磁盘的休眠文件中；当系统通电后，可以迅速恢复计算机状态
- ❑ 安全隐患：如果没有安全防范措施，攻击者可以利用休眠，入侵计算机
 - ❑ 例如，内存的全部内容都保存在休眠文件中，攻击者可以从休眠文件中获取一些敏感信息。
 - ❑ Windows系统的休眠文件为C:\hiberfil.sys
 - ❑ 攻击者还有可能会修改hiberfil.sys文件，当计算机再次启动时改变在计算机上执行的程序
 - ❑ Windows系统不会删除休眠文件，计算机重启后该文件依然存在
 - ❑ 防御措施：使用硬盘加密来保护休眠文件



- 操作系统安全一个重要方面是：“事态感知” (Situational awareness)
 - 记录哪些进程正在运行、哪些计算机在联网
 - 如果操作系统遇到任何意外或者可疑操作，都会留下重要线索
 - 通过留下的线索，可以解决常见的问题，还能确定出现安全漏洞的原因
 - 例如，反复失败的尝试登录日志可能警告有蛮力攻击，提示系统管理员修改密码以确保系统安全
- 操作系统为了监控和记录提供的主要手段：事件日志、进程监控和查看

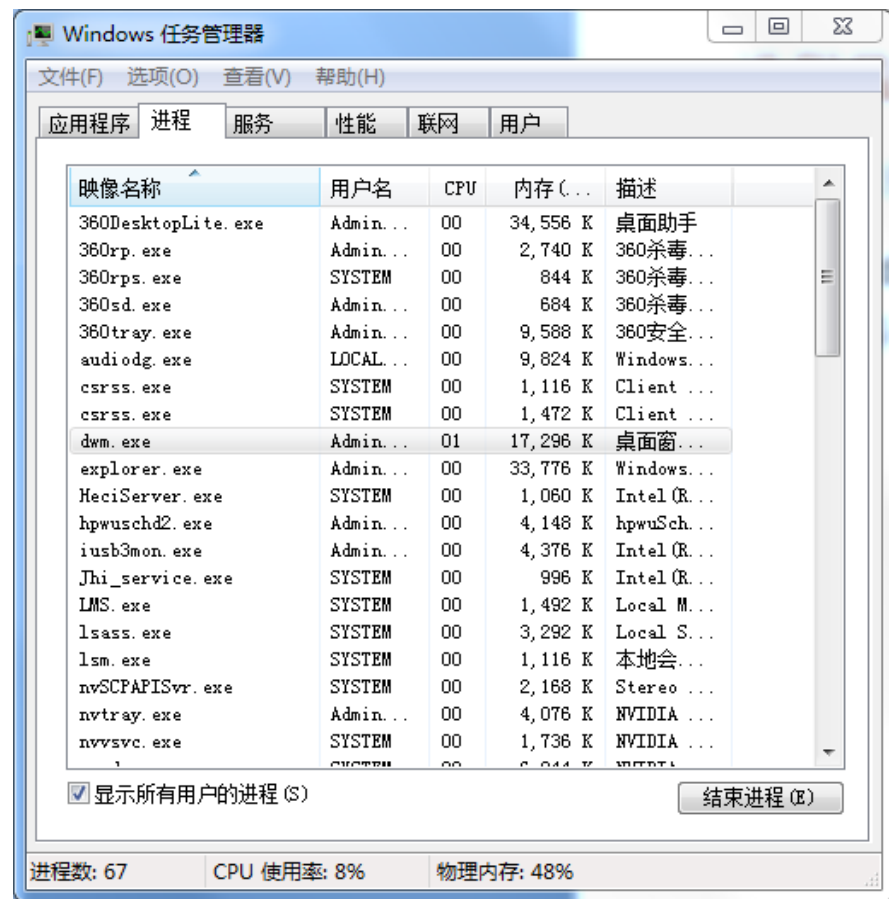


- ❑ 操作系统有内置功能来管理事件日志
- ❑ Windows系统定义了三种日志源：系统、应用程序和安全
- ❑ 系统日志只有操作系统才能写入；应用程序只能写应用程序日志；而安全日志由操作系统提供的本地安全授权子系统(LSASS)服务才能写入
- ❑ Linux系统的具体日志机制跟发布的版本有关。通常情况下，日志文件存储在/var/log中，是具有描述性名称的简单文本文件
- ❑ 如auth.log包含用户身份验证记录；kern.log记录内核操作等。通常情况下只有特殊的syslog守护进程才能写这些日志文件
- ❑ 日志项一般都包含时间戳和对事件的描述

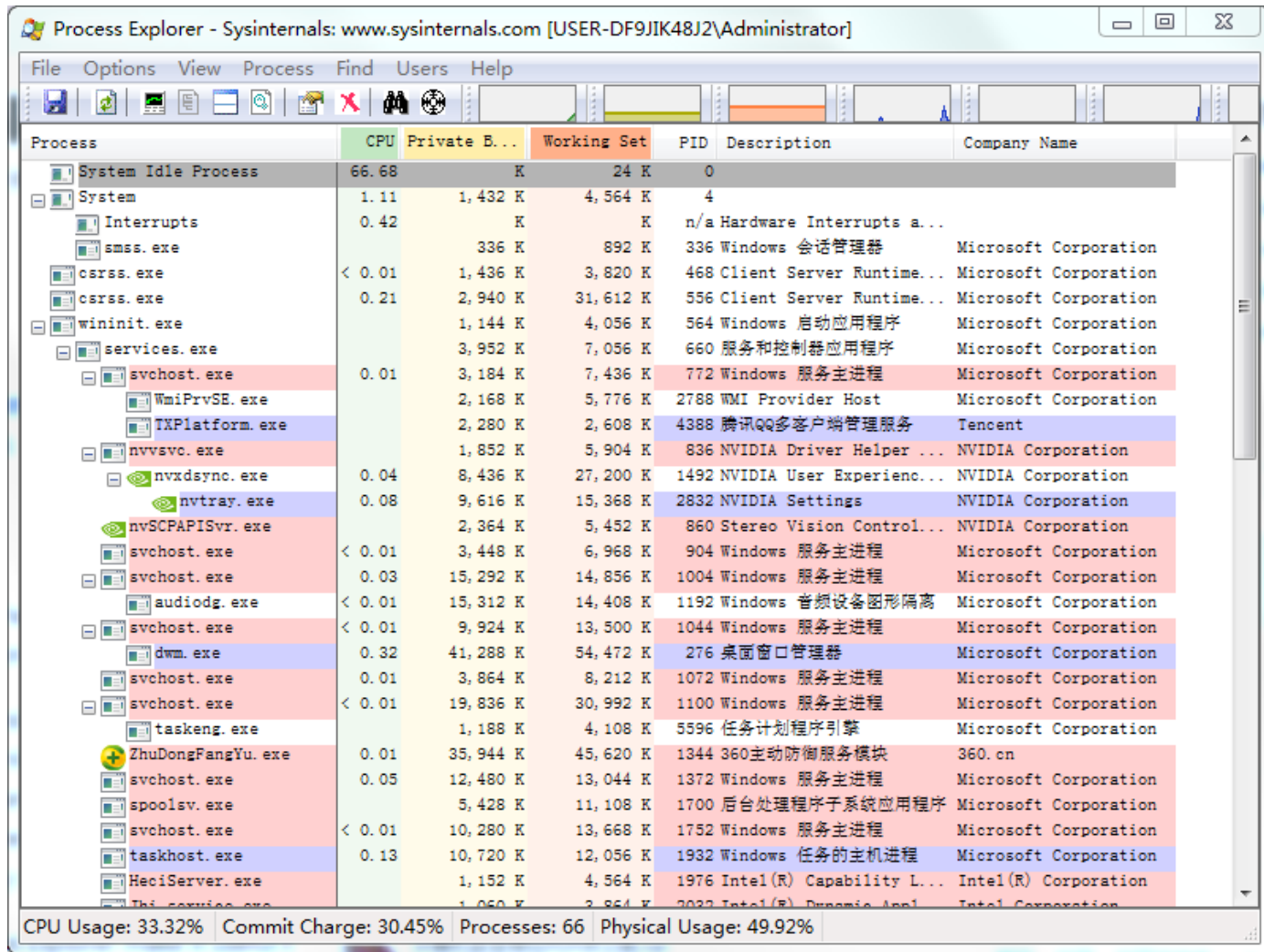


进程监控 Process monitoring

- 用来探查计算机上正在运行的进程
- Windows系统可以使用任务管理器查看
- Linux可以使用ps、top、pstree、kill等命令



- 软件开发人员可以对映像使用数字签名来确保磁盘上存储的应用程序不被攻击者替换





PART 1

操作系统的概念

PART 2

进程的安全

PART 3

内存与文件系统的安全

PART 4

应用程序的安全

PART 5

练习题



❑ 进程查看器计算机的信息都封装在内存和文件系统之中。因此保护计算机安全就首先要保护内存和文件系统

❑ 虚拟内存安全

❑ Windows系统中，虚拟内存页面存储在C:\pagefile.sys中

❑ Linux系统需用户自己设定一个硬盘分区作为交换分区，存储内存页面

❑ 关闭计算机时，硬盘上的虚拟内存页面会被删除

❑ 虚拟内存安全问题

❑ 如果攻击者突然将计算机断电，没有正常关闭计算机，此时虚拟内存页面可能未被删除

❑ 攻击者通过外部介质引导另一个操作系统，能够查看这些文件并重建部分内存，导致信息泄露

❑ 解决办法：对硬盘内容进行加密

□ 如何才能安全地确定合法用户呢？

□ 通过身份验证，常见认证机制：

用户名和密码

□ 密码不能以明文的形式存储， 在密码文件或数据库中以密码 的单向散列值来存储

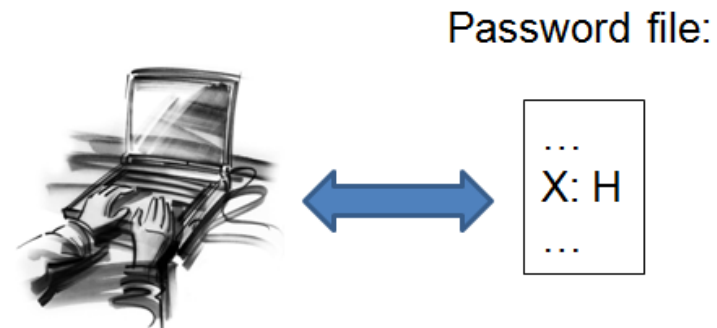
□ 字典攻击对弱密码比较高效，

如何使字典攻击变得困难呢？

- 口令盐(password salt)技术是一种加密技术，使用随即位作为散列函数或加密算法的部分输入，从而增加输出的随机性
- 设U为用户ID，P是对应的口令。当使用盐时，口令文件存储三元组(U,S,h(S||P))，其中S为U的盐，h是加密散列函数

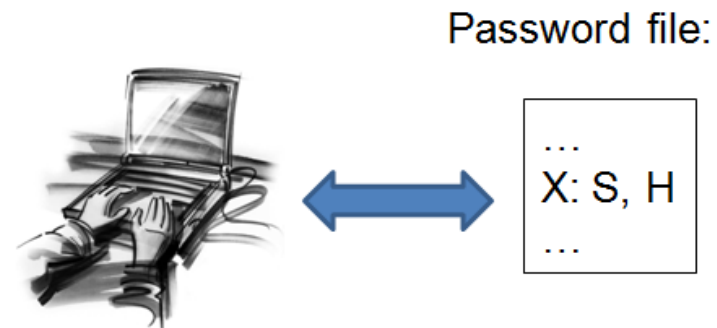
Without salt:

1. User types userid, X, and password, P.
2. System looks up H, the stored hash of X's password.
3. System tests whether $h(P) = H$.



With salt:

1. User types userid, X, and password, P.
2. System looks up S and H, where S is the random salt for userid X and H is stored hash of S and X's password.
3. System tests whether $h(S || P) = H$.





使用盐大大增加了字典攻击的搜索空间

- 假设攻击者无法获取用户使用的盐，那么对于加盐的口令，字典攻击的搜索空间大小为

$$2^B \times D$$

- 其中B是随机盐的位数，D是字典攻击的单词列表大小。
- 例如，对于每个用户，系统使用32位的盐，并且用户在50万单词的字典中挑选口令，那么攻击加盐口令的搜索空间为：

$$2^{32} \times 500\,000 = 2147483648000000$$

- 即使攻击者能够找到每个用户ID相关的盐，通过采用加盐口令，操作系统依然可以限制字典攻击：一次只能攻击一个用户ID，因为攻击者对每个用户得使用不同的盐值



- 系统完成对用户的身份验证后，下一个重要问题是：

操作系统如何确定用户具有哪些操作权限？

- 相关术语

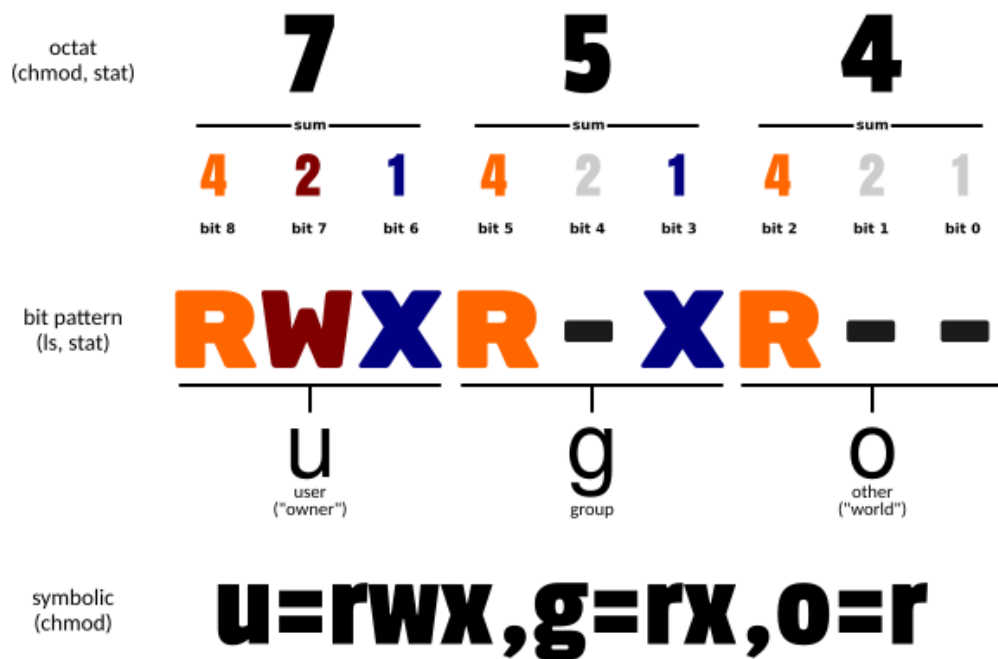
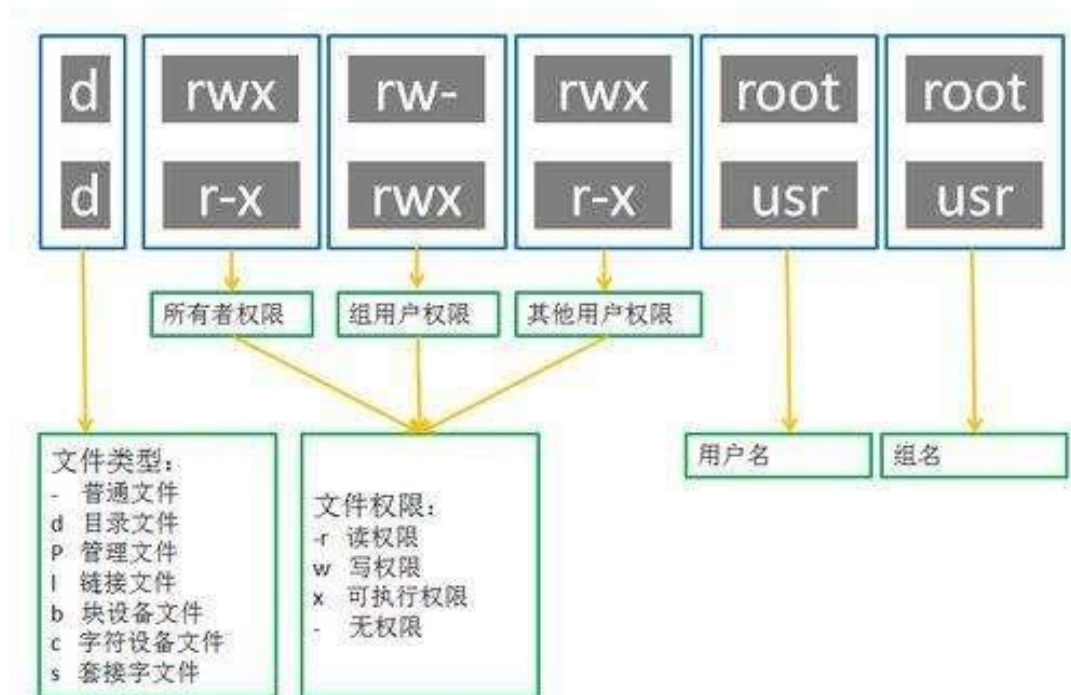
- 主体：用户或组用户。Unix文件系统中每个文件都有一个用户owner和owning group。
- 权限：对文件或者文件夹执行的具体操作。文件权限包括读、写、执行等，文件夹有列出和执行权限
- 访问控制项：三元组(主体，类型，权限)，类型是允许或拒绝。
- 访问控制列表：访问控制项的有序列表
- 操作系统权限方案如何设计：权限如何与系统的文件组织进行交互？权限的继承是层次结构码



- ❑ Linux系统使用文件权限矩阵来确定各个用户对文件的访问权限
- ❑ 所有未明确授予的权限都以为着拒绝
- ❑ 为了访问文件，在文件系统树中的每个祖先文件夹都必须具有执行权限
- ❑ 文件所有者具有自主访问控制能力，可以改变这些文件的权限
- ❑ 扩展的文件权限：使文件只能追加、将文件标记为不可修改。
- ❑ 可以使用chattr命令设置这些属性，lsattr命令查看这些属性
- ❑ 最近，Linux开始支持基于ACL的权限方案。
- ❑ 访问使用getfacl命令查看ACL，使用setfacl命令设置ACL
- ❑ 对owner, group和other主体，每个文件都有基本的ACE；对于命名用户和命名组可创建额外的ACE；
还有掩码ACE,即规定主体允许的最大权限



Linux的权限



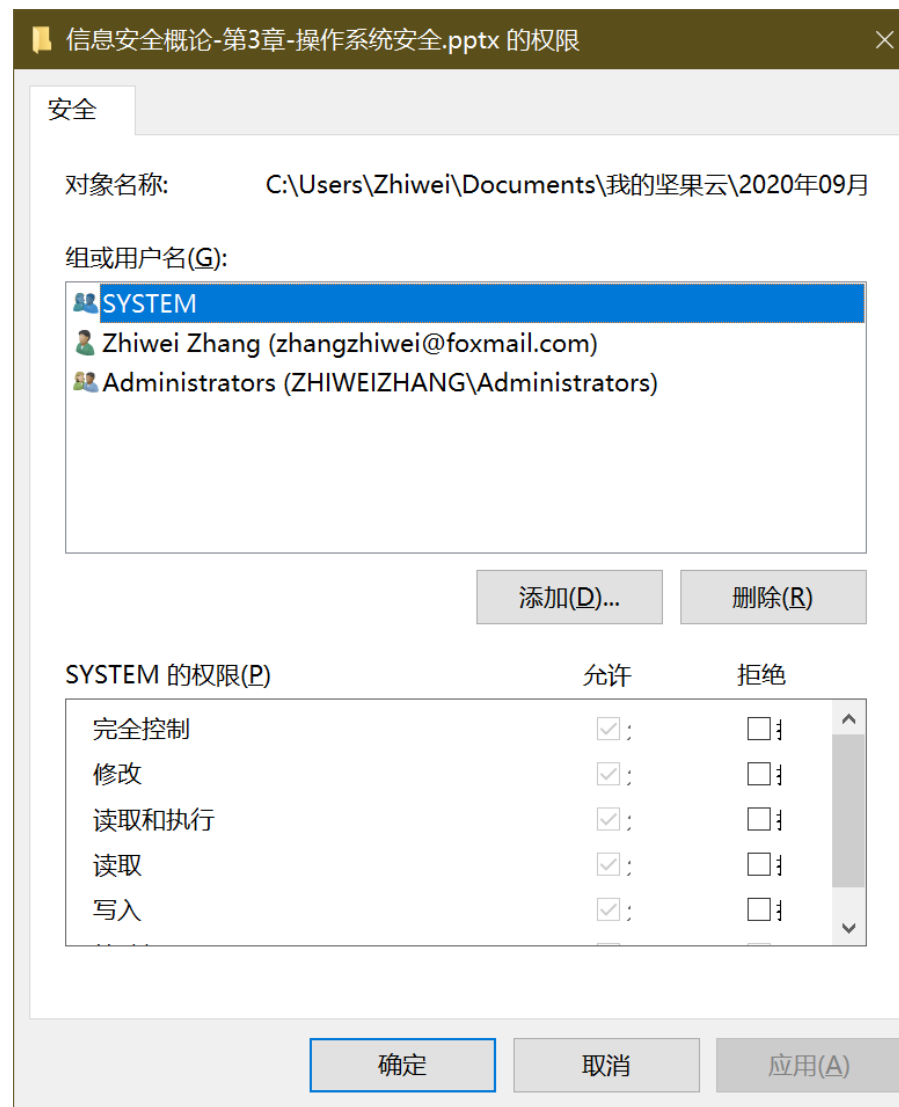
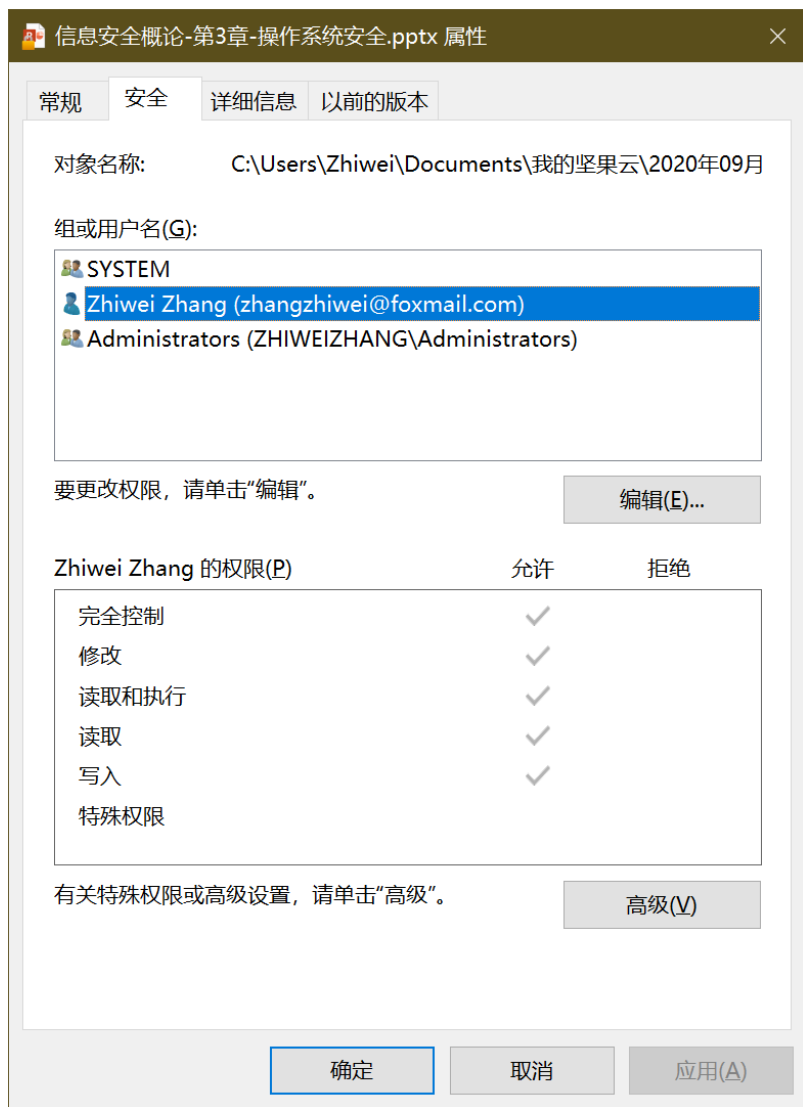
权限项	文件类型	读	写	执行	读	写	执行	读	写	执行
字符表示	(d l c s p)	(r)	(w)	(x)	(r)	(w)	(x)	(r)	(w)	(x)
数字表示		4	2	1	4	2	1	4	2	1
权限分配		文件所有者			文件所属组用户			其他用户		



- ❑ Windows系统使用访问控制列表模型，允许用户创建访问控制规则
- ❑ 规则可以设计为允许或者拒绝类型的，如果不设置，则默认是拒绝
- ❑ 标准权限：读、写、修改、读和执行、完全控制
- ❑ 与读权限相关的高级权限：读取数据、读取属性、读取扩展属性和读取权限
- ❑ 文件夹权限：读权限，可以列出文件夹内容，写权限允许用户创建新文件夹
- ❑ 与Linux不同，Windows系统中管理员可以禁止用户访问文件夹，但允许用户访问该文件夹下的特定文件
- ❑ 继承ACE：文件夹的任何ACE可用于该文件夹内的子文件夹和文件
- ❑ 显式ACE：针对文件或者文件夹专门设置的ACE
- ❑ ACE优先级：拒绝ACE>允许ACE、显示ACE>继承ACE、继承ACE优先级由祖先与对象之间的距离决定，越近优先级越高



Windows文件权限





□ SetUID操作

- 程序由普通用户运行，但期望允许程序改变普通用户无法改变的文件
- 例如：在早期UNIX系统中，用户登录信息存储在/etc/passwd中。普通用户不能编辑这个文件，但允许该用户更改自己的密码是合理的要求
- 如何解决这一矛盾？文件权限矩阵中有一个额外位，称为setUID位
- 如果设置了setUID位，则程序以其所有者的有效用户ID来运行，而不是以正在执行程序进程的ID来运行
- 例如UNIX系统中的passwd，它的所有者是根用户。当用户运行passwd时，程序以根用户的权限来运行，能够修改/etc/passwd
- **setUID机制解决了无特权情况下的访问问题，同时也引发了安全隐患。**
- 攻击者可以强制setUID程序执行任意代码，如缓冲区溢出攻击。
- 权限升级：攻击者可以通过setUID机制来运用程序的所有者权限

```
chmod 4750 文件名  
or  
chmod u+s 文件名
```




- ❑ 存储在文件描述符表中的索引值，用来索引特定文件
- ❑ 当程序需要访问文件时，访问open系统调用，该调用使内核创建一个文件描述符表中的新项并将其返回给程序，该项映射到文件的磁盘位置
- ❑ 程序可以使用文件描述符发送读或者写命令。内核在接到读、写系统调用时，在文件描述符表中查找相关的表项，并在磁盘适当位置执行读、写操作
- ❑ 完成操作后，要用close系统调用删除打开的文件描述符
- ❑ 使用文件描述符进行操作时，内核检查调用进程对文件是否有操作权限，如果没有相关的权限，返回操作失败
- ❑ **文件描述符漏洞**：当进程创建子进程时，子进程会集成父进程打开的所有文件描述符副本。当程序以高权限打开文件描述符，但未关闭，然后又创建了低权限的进程，那么新进程就能够读写相关文件，但子进程本身不具有打开该文件的权限
- ❑ **产生文件描述符漏洞的重要原因之一**：在创建文件描述符项的时刻，操作系统只检查进程是否具有读写权限；在实际读写文件操作时，只根据文件描述符被打开时的权限来确认是否允许请求的操作



- ❑ 对用户而言，能创建系统中其他文件的链接或快捷方式非常有用，因为它不需要将整个文件复制到新的位置
- ❑ Linux系统完成上述功能的是符号链接，使用ln命令来创建符号链接
- ❑ **安全隐患：** 恶意方会利用符号链接诱使应用程序执行不良的操作
 - ❑ 例如，考虑程序打开并读取用户指定文件的情形，假设程序不能读取一个特定文件/home/admin/passwords
 - ❑ 这个程序的不安全版本只检查用户指定的名是不是特定文件。然而，攻击者可以创建一个到该文件的符号链接，并指定符号链接的路径来欺骗这个程序
- ❑ 解决办法：程序检查文件名是否指向符号链接，以确定打开的实际文件名
- ❑ Windows系统类似的称为快捷方式
 - ❑ 两者的区别是：符号链接由操作系统处理，使用透明；而快捷方式是普通文件



PART 1

操作系统的概念

PART 2

进程的安全

PART 3

内存与文件系统的安全

PART 4

应用程序的安全

PART 5

练习题



- ❑ 编译：将源代码转换为处理器能够执行的机器代码的过程
- ❑ 静态链接：程序执行时所需的共享库需要复制到编译程序中。一般比较安全，但重复代码会占用额外空间
- ❑ 动态链接：程序真正运行时，才会加载共享库。加载程序确定待运行程序需要哪些共享库，然后在磁盘上找到这些库，并将它们导入进程的地址空间。
- ❑ Windows中这些外部库称为动态链接库(Dynamic linking library, DLL); Linux中这些外部库称为共享对象(Shared object)
- ❑ 动态链接可以节省硬盘空间，允许开发者将代码模块化。例如为了修复DLL产生的漏洞，只需要改变一个特定DLL
- ❑ 优点是便于调试，缺点是潜在的安全风险，恶意用户能向合法程序注入自己的代码
- ❑ **DLL注入**：通过共享库向程序注入任意代码的过程。



- **缓冲区溢出(Buffer overflow)**

- 缓冲区：进程为程序在内存中分配固定大小的存储空间
- 确保缓冲区安全需要在向缓冲区复制用户数据时进行边界检查
- 缓冲区溢出：如果不进行边界检查，攻击者提供的输入数据可能会超出缓冲区的大小。此时，内存缓冲区之外的数据可能会被覆盖。
- 攻击者利用缓冲区溢出获得进程的控制权，执行任意恶意代码

- **算术溢出(Arithmetic overflow)**

- 整数在内存中使用二进制补码表示。在32位计算机中，0x00000000-0x7fffffff表示正整数，0x80000000-0xffffffff表示正整数负整数
- 溢出：如果某个程序不断加上非常大的整数，导致结果超出0x7fffffff，产生上溢，最后结果变为负数；类似，如果某个程序不断加上负数，最终结果会下溢，变为正数。

• 一个利用算术溢出的脆弱性程序示例

- 假设网络服务记录从启动开始建立的连接数，且只允许前5个用户进行访问

```
int main(int argc, char * argv[])
{
    unsigned int connections = 0;
    // Insert network code here
    // ...
    // ...
    // Does nothing to check overflow conditions
    connections++;
    if(connections < 5)
        grant_access();
    else
        deny_access();
    return 1;
}
```

一个容易出现算术溢出的c程序

```
#include <stdio.h>

int main(int argc, char * argv[])
{
    unsigned int connections = 0;
    // Insert network code here
    // ...
    // ...
    // Prevents overflow conditions
    if(connections < 5)
        connections++;
    if(connections < 5)
        grant_access();
    else
        deny_access();
    return 1;
}
```

防止算术溢出的c程序



- 基于栈的缓冲区溢出(Stack-based buffer overflow)

- 进程地址空间中栈由帧组成，每一帧存储局部变量、调用参数和返回地址
- 如果攻击者提供的输入大于调用函数中缓冲区的大小，会导致溢出。如C库函数strcpy()和gets()，这两个函数不检查输入长度，从而导致栈缓冲区之外的内存空间被覆盖。

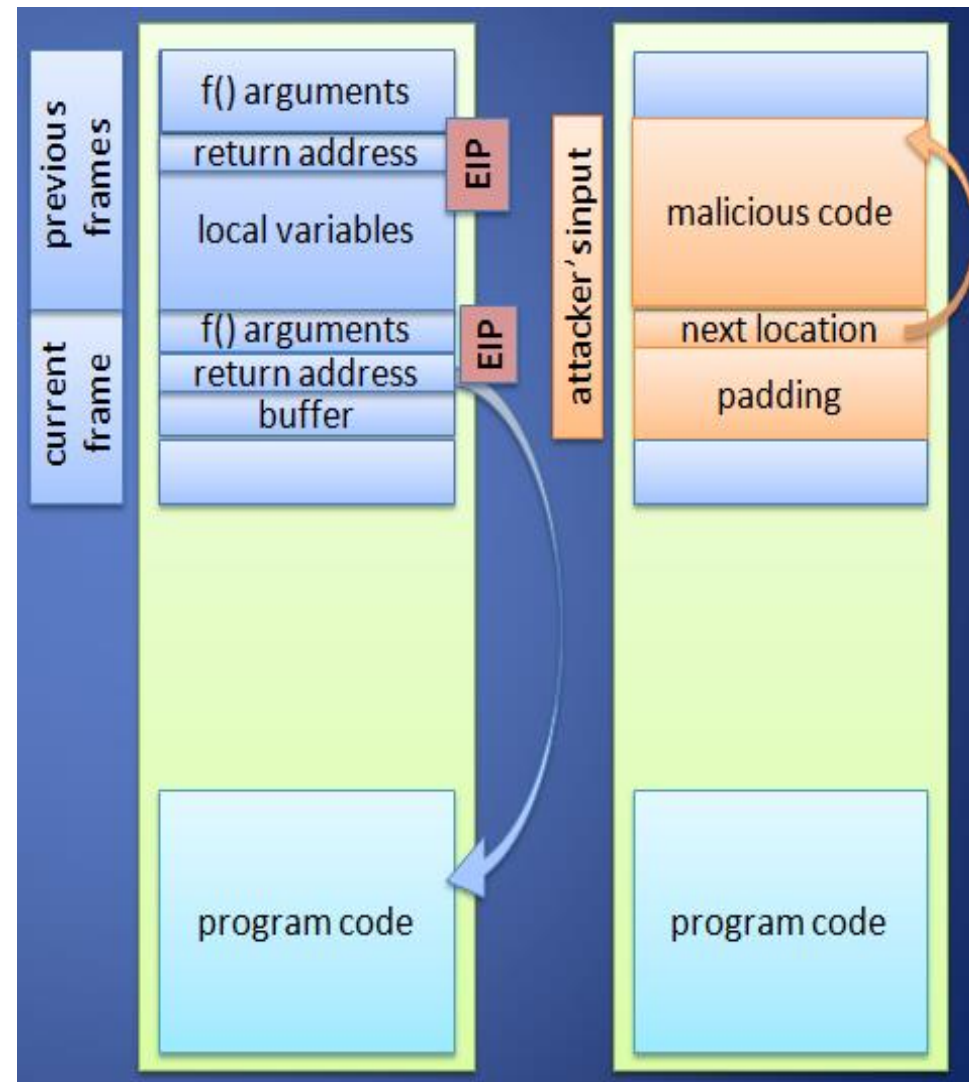
```
#include <stdio.h>

int main(int argc, char * argv[])
{
    // Create a buffer on the stack
    char buf[256];
    // Does not check length of buffer before copying argument
    strcpy(buf,argv[1]);
    // Print the contents of the buffer
    printf("%s\n",buf);
    return 1;
}
```

一个可能出现栈缓冲区溢出的C程序示例

□ 缓冲区溢出攻击

- 缓冲区溢出非常常见，且很危险。当缓冲区是局部变量或帧的参数时，用户的输入可能覆盖返回地址，改变程序的执行
- 攻击者利用栈缓冲区的脆弱性，在栈中注入恶意代码，覆盖当前调用的返回地址，从而将执行权限传递给攻击者的恶意代码
- 在理想的栈溢出攻击中，假定攻击者知道返回地址的确切位置



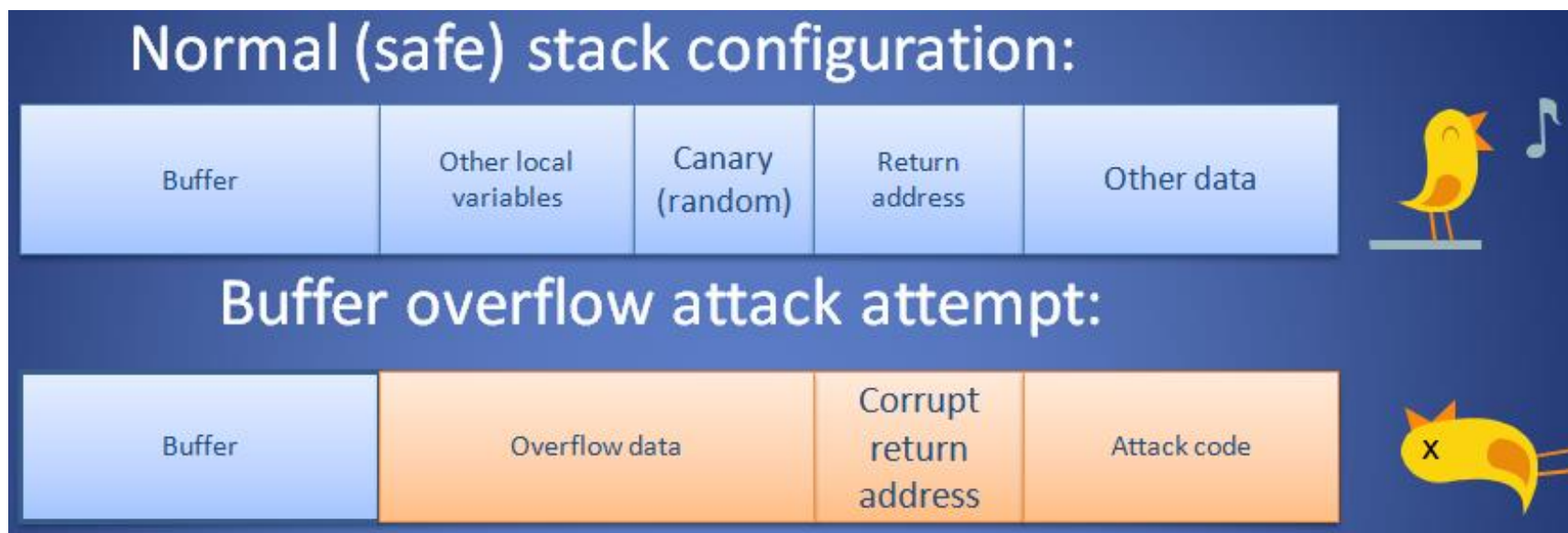


□ 主要措施

- 缓冲区溢出根源在于不安全的编程实践，程序员需确保自己的程序复制的信息比缓冲区小。
C语言容易受缓冲区溢出攻击，其它语言不允许这种行为
- 可以使用更安全的函数来写代码，如用strncpy来代替strcpy函数
- 操作系统也听过了保护机制以防止缓冲区溢出。如当检测到缓冲区溢出时，控制恶意代码的重定向。其具体实现包括：(1)防止攻击者覆盖返回地址；(2)使用金丝雀技术
- 防止攻击者覆盖返回地址：(1) 微软开发了一种编译器的扩展，称为Point-Guard，其主要思想是将所有指针在使用前后都进行异或编码。(2) 将内存的栈空间设置为非执行权限。(3) 地址空间布局随机化(ASLR)，随机地重新安排地址空间的数据，使得攻击者很难预测

□ 金丝雀预警技术

- 金丝雀(canary)的呼吸频率与人类相比高很多。在相同时间里金丝雀呼入的有害气体要比人类的多。另一方面，金丝雀在空气中有害气体浓度还很低时就会有反应，如烦躁不安、大声叫喊等。因此，煤矿等常使用金丝雀来预警
- 重新组织分配程序的堆栈数据，使用一个canary值，并将此值放在缓冲区和控制数据之间。系统定期检查canary值的完整性。如果此值被更改，表明缓冲区溢出，就要防止恶意代码的执行





- 允许程序员动态分配内存的地址空间称为堆(heap)。如果在堆上显示分配了内存(如使用malloc、new函数),但没有释放,会导致**内存泄露**问题,对基于堆的缓冲区溢出攻击类似于基于栈的攻击
- 但基于堆的攻击更加复杂,它需要攻击者深入理解垃圾回收机制和堆是如何实现的,攻击者一般通过修改堆中的数据或者滥用管理堆内存的函数和宏来执行恶意代码

□ 基于堆的缓冲区溢出攻击例子

- 考虑GCC中malloc函数的实现,堆中的内存块使用链表来维护。当一个块被释放时,unlink宏会将与待释放的内存块相邻的两个内存块相互链接
- 如果攻击者向程序提供输入,而程序以不安全的方式将其复制到堆中的内存块中。那么攻击者可以使数据溢出块的边界,并覆盖下一个内存块
- 通过精心设计,攻击者会覆盖下一个内存块的链表指针,并将该内存块标记为空闲。此时,unlink例程可以向内存地址空间中的任意地址写入数据。如果写入的数据是恶意代码的存储位置,会导致程序跳转并执行恶意代码

□ 防御措施

- 与栈溢出防御措施类似,如安全编程、地址空间随机化、设置堆数据不可执行、将存储堆内存的指针数据和存储堆中的实际数据相分离



- ❑ C库函数在打印或者读入数据时，使用格式化字符串，如printf(“%s”, str)。但如果不输入格式化信息，而是直接使用printf(str)，会出现漏洞
- ❑ 攻击者可以精心指定所使用的格式化字符串作为输入，并可以将输入写入内存的任意位置。如果写入的位置正好覆盖了返回地址、函数指针等，则攻击者能够取得控制权，并在程序的上下文中执行任意代码
- ❑ 与格式化字符串相关的C库函数有fprintf, printf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf等
- ❑ 格式化字符串参数包括%%, %p, %d, %c, %u, %x, %s, %n等，其中%x从栈中读取数据，%s读取字符串，%n将当前打印出的字符数目写入参数指定的地址



格式化字符串攻击Format string attack

- 打印内存：打印地址空间栈中的8个数值

```
1 #include <stdio.h>
2 int main(void)
3 { printf("%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x");
4   return 0; }
```

```
root@lh:/usr/local/ex# ./ex1
00000000,08048419,b7774ff4,08048410,00000000,00000000,b75ed4d3,00000001,bfffab84
```

- 修改内存：a的值被修改为7.利用这个漏洞，可以修改地址空间中任意内存的值

```
1 #include <stdio.h>
2 int main(void)
3 { int a; printf("aaaaaaa\n\n",&a);
4   printf("%d\n",a);
5   return 0; }
```

```
root@lh:/usr/local/ex# ./ex2
aaaaaaa
7
```



□ 格式化字符串攻击的具体操作可以参考一下资料

- <http://forum.ouah.org/FormatString.PDF>
- <http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html>
- <https://blog.csdn.net/aqifz/article/details/49704287>
- <https://blog.csdn.net/immcss/article/details/6267849>
-



PART 1

操作系统的概念

PART 2

进程的安全

PART 3

内存与文件系统的安全

PART 4

应用程序的安全

PART 5

练习题



- 使用VMware、VirtualBox等软件，搭建自己的虚拟机系统；结合本章相关内容，并进一步查找溢出攻击、格式化字符串攻击的资料；在虚拟机中复现相关的案例，分析自己成功或失败的原因
- R-3.9 R-3.11 R-3.12 R-3.20 R-3.21
- C-3.4 C-3.8

本章结束

~End~

是什麼讓火焰燃燒，是薪柴之間的空隙，它們靠此呼吸。

What makes a fire burn
is space between the
logs, a breathing space.