二进制安全学习笔记

发布 *1.0.0*

lyle

2020 年 11 月 05 日

目录:

CHAPTER 1

基础知识

1.1 计算机发展简史

1.2 布尔代数

1.3 大小端与数据表示

1.3.1 数字

有补码编码 (two's complement) 与反码编码 (one's complement)

CHAPTER 2

计算机体系结构

2.1 计算机系统构成

2.1.1 处理器（Processor）

简介

处理器是计算机的核心部件，由超大规模集成电路芯片构成，通常简称 CPU。

一方面，它完成各种基本运算，包括加、减、乘、除等算术运算和逻辑判断、比较、位移等逻辑运算。另一方面，CPU 还控制计算机各部件协调地工作，也就是说，整个计算机工作的顺序、信息输入、输出都是由 CPU 统一控制。

RISC 与 CISC

精简指令集计算机（Reduced Instruction Set Computer，RISC）和复杂指令集计算机（Complex Instruction Set Computer ）是当前 CPU 的两种架构，它们的区别在于不同的 CPU 设计理念和方法。

早期的 CPU 全部是 CISC 架构，它的设计目的是要用最少的机器语言指令来完成所需的计算任务。比如对于乘法运算，只需要一条指令 `mul addra, addrb` 就可以完成，将地址中的数据读入寄存器，相乘和将结果写回内存的操作全部依赖于 CPU 中设计的逻辑来实现。这种架构会增加 CPU 结构的复杂性和对 CPU 工艺的要求，但对于编译器的开发十分有利。

RISC 架构要求软件来指定各个操作步骤。上面的例子如果要在 RISC 架构上实现，指令可能为 `mov eax, addra; mov ebx, addrb; mul eax, ebx; str addra, a` 这种架构可以降低 CPU 的复杂性以及允许在同样的工艺水平下生产出功能更强大的 CPU，但对于编译器的设计有更高的要求。

2.1.2 内存储器（Memory）

内存储器简称内存。内存分为两部分，有一部分用于永久存放特殊专用数据，只能从中取出数据，不能向内写入数据，称为只读存储器（简称 ROM）。其余的部分由计算机执行程序时使用，既可存入数据又可取出数据，称为随机存储器（简称 RAM）。

2.1.3 外存储器（Auxiliary Memory）

内部存储器容量是有限的，而且当计算机断电后，里面的内容会消失。因此需要容量更大，且能永久保存数据的存储器，这就是外存储器，也称为辅助存储器。BIOS 固件就是一种特殊的外存贮器。

2.1.4 输入设备

从计算机外部获取信息的设备称为输入设备。

常见的输入设备有键盘、鼠标、话筒、扫描仪等。

2.1.5 输出设备

输出设备将计算机中的数据以文字、图形、声音等方式显示出来。

常见的输出设备有：显示器、打印机等。

2.2 CPU

2.2.1 简介

CPU 是执行指令的核心。一个 CPU 的执行周期是从内存中提取一条指令、解码并决定它的类型和操作数，执行，然后再提取、解码执行后续的指令，一直重复这个循环。

每个 CPU 都有一组可以执行的特定指令集，一般情况下不能执行其他指令集的指令。由于访问内存获取执行或数据要比执行指令花费的时间长，因此所有的 CPU 内部都会包含一些寄存器来保存关键变量和临时结果。

大多数 CPU 都具有几个特殊的寄存器。其中之一是程序计数器 (program counter)，程序计数器会指示下一条需要从内存提取指令的地址。提取指令后，程序计数器将更新为下一条需要提取的地址。

另一个寄存器是堆栈指针 (stack pointer)，它指向内存中当前栈的顶端。堆栈指针会包含输入过程中的有关参数、局部变量以及没有保存在寄存器中的临时变量。

特殊寄存器

特殊用途寄存器，顾名思义是仅为一项任务而设计的寄存器。例如，cs/ds/gs 和其他段寄存器属于特殊目的寄存器，因为它们的存在是为了保存段号。eax，ecx 等是一般用途的寄存器，可以无限制地使用。

通用目的寄存器比如有：eax、ecx、edx、ebx、esi、edi、ebp、esp

特殊目的寄存器比如有：cs、ds、ss、es、fs、gs、eip、flag

2.2.2 CPU 结构

冯 · 诺依曼结构

冯 · 诺依曼结构又称作普林斯顿体系结构（Princetionarchitecture）。

1945 年，冯 · 诺依曼首先提出了“存储程序”的概念和二进制原理，后来，人们把利用这种概念和原理设计的电子计算机系统统称为“冯 · 诺依曼型结构”计算机。冯 · 诺依曼结构的处理器使用同一个存储器，经由同一个总线传输。

冯 · 诺依曼结构处理器具有以下几个特点：

- 有一个存储器
- 有一个控制器
- 有一个运算器，用于完成算术运算和逻辑运算
- 有输入和输出设备，用于进行人机通信

冯 · 诺依曼的主要贡献就是提出并实现了“存储程序”的概念。由于指令和数据都是二进制码，指令和操作数的地址又密切相关，因此，当初选择这种结构是自然的。但是，这种指令和数据共享同一总线的结构，使得信息流的传输成为限制计算机性能的瓶颈，影响了数据处理速度的提高。

在典型情况下，完成一条指令需要 3 个步骤，即：取指令、指令译码和执行指令。从指令流的定时关系也可看出冯 · 诺依曼结构与哈佛结构处理方式的差别。举一个最简单的对存储器进行读写操作的指令，指令 1 至指令 3 均为存、取数指令，对冯 · 诺依曼结构处理器，由于取指令和存取数据要从同一个存储空间存取，经由同一总线传输，因而它们无法重叠执行，只有一个完成后再进行下一个。

哈佛结构

哈佛结构是一种将程序指令存储和数据存储分开的存储器结构，它的主要特点是将程序和数据存储在不同的存储空间中，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问，目的是为了减轻程序运行时的访存瓶颈。

例如最常见的卷积运算中，一条指令同时取两个操作数，在流水线处理时，同时还有一个取指操作，如果程序和数据通过一条总线访问，取指和取数必会产生冲突，而这对大运算量的循环的执行效率是很不利的。

哈佛结构能基本上解决取指和取数的冲突问题。

目前使用哈佛结构的中央处理器和微控制器有很多，除了 Microchip 公司的 PIC 系列芯片，还有摩托罗拉公司的 MC68 系列、Zilog 公司的 Z8 系列、ATMEL 公司的 AVR 系列和 ARM 公司的 ARM9、ARM10 和 ARM11。

改进型哈佛结构

改进型哈佛结构虽然也使用两个不同的存储器：程序存储器和数据存储器，但它把两个存储器的地址总线合并了，数据总线也进行了合并，即原来的哈佛结构需要 4 条不同的总线，改进后需要两条总线。

改进型哈佛结构其结构特点为：使用两个独立的存储器模块，分别存储指令和数据，每个存储模块都不允许指令和数据并存，以便实现并行处理。

具有一条独立的地址总线和一条独立的数据总线，利用公用地址总线访问两个存储模块（程序存储模块和数据存储模块），公用数据总线则被用来完成程序存储模块或数据存储模块与 CPU 之间的数据传输。

两条总线由程序存储器和数据存储器分时共用。

2.2.3 指令集架构

简介

指令集架构（英语：Instruction Set Architecture，缩写为 ISA），又称指令集或指令集体系，是计算机体系结构中与程序设计有关的部分，包含了基本数据类型、指令集、寄存器、寻址模式、存储体系、中断、异常处理以及外部 I/O。指令集架构包含一系列的 opcode 即操作码（机器语言），以及由特定处理器执行的基本命令。

指令集体系与微架构（一套用于执行指令集的微处理器设计方法）不同。使用不同微架构的计算机可以共享一种指令集。例如，Intel 的 Pentium 和 AMD 的 AMD Athlon，两者几乎采用相同版本的 x86 指令集体系，但是两者在内部设计上有本质的区别。

分类

指令集分为复杂指令集（Complex Instruction Set Computing，CISC）和精简指令集（Reduced Instruction Set Computing，RISC）。

复杂指令集计算机包含许多应用程序中很少使用的特定指令，由此产生的缺陷是指令长度不固定。精简指令集计算机通过只执行在程序中经常使用的指令来简化处理器的结构，而特殊操作则以子程序的方式实现，它们的特殊使用通过处理器额外的执行时间来弥补。

2.2.4 流水线

流水线（pipeline）技术是指在 CPU 执行时多条指令同时进行操作的一种准并行处理实现技术。

主要是使用不同的电路功能单元组成一条指令处理流水线，然后将一条指令分为多步来执行，这样可以在一个时钟周期内完成一条指令。

一般一条指令可以分为取指、译指、执行、写回等步骤。

流水线优化

- 分支预测
- 指令冒险
- 乱序发射
- 乱序执行

权限

ring0 到 ring3，ring0 为最高权限，ring3 最低，一般 ring0 为内核权限，ring3 为用户权限，很少用 ring1 和 ring2。

2.2.5 Intel

PT

Intel PT 是 Intel 的一个扩展功能, 它利用硬件以很小的开销来记录程序执行数据，这些数据内容包括: 时间,、程序流信息 (e.g. 分支目标, 分支是否执行)。

2.2.6 发展历史

1971 年，Intel 推出了世界上第一款微处理器 4004，它是一个包含了 2300 个晶体管的 4 位 CPU。随后英特尔又推出了 8008，由于运算性能很差，其市场反应十分不理想。1974 年，8008 发展成 8080，成为第二代微处理器。

1978 年，Intel 推出了具有 16 位数据通道、内存寻址能力为 1MB、最大运行速度 8MHz 的 8086，同时还生产出与之配合的数学协处理器 8087，这两种芯片使用相互兼容的指令集，这种指令集之后称之为 x86 指令集。随后，Intel 又推出了 80186 和 80188。

1979 年，Intel 公司推出了 8088 芯片，它是第一块成功用于个人电脑的 CPU。1981 年 8088 芯片首次用于 IBM PC 机中，开创了全新的微机时代。

1982 年，Intel 推出 80286 芯片，虽然它仍旧是 16 位结构，但在 CPU 的内部集成了 13.4 万个晶体管，时钟频率由最初的 6MHz 逐步提高到 20MHz。其内部和外部数据总线皆为 16 位，地址总线 24 位，可寻址 16MB 内存。

1985 年，Intel 推出 80386 芯片，这是 x86 系列中的第一种 32 位 CPU，而且制造工艺也有了很大的进步。80386 内部内含 27.5 万个晶体管，时钟频率从 12.5MHz 发展到 33MHz。80386 的内部和外部数据总线都是 32 位，地址总线也是 32 位，可寻址高达 4GB 内存，可以使用 Windows 操作系统。

1989 年，Intel 推出 80486 芯片，它的特殊意义在于这块芯片首次突破了 100 万个晶体管的界限，集成了 120 万个晶体管。80486 将 80386 和数学协处理器 80387 以及一个 8KB 的高速缓存集成在一个芯片内，并且在 80X86 系列中首次采用了 RISC 技术。它还采用了突发总线（Burst）方式，大大提高了与内存的数据交换速度。

随后，AMD、Cyrix 等陆续推出了 80486 的兼容 CPU。在之后，Intel 没有将 486 的后一代产品称为 586，而是使用了注册商标 Pentium。

2.2.7 参考链接

官方文档

- List of Intel microprocessors
- List of ARM microarchitectures
- List of AMD microprocessors
- Processor Tracing

Blog

- 对 ARM9 哈佛结构的认识

2.3 BIOS

2.3.1 简介

BIOS 这个字眼是在 1975 年第一次由 CP/M 操作系统中出现，是个人电脑启动时加载的第一个软件。

2.3.2 功能

- **开机自检 (Power On Self Test，POST)**
  - 计算机开机时系统将控制权交给 BIOS，BIOS 针对 CPU 各项寄存器、标志位等先检查 CPU 是否工作正常，接下来会检查 8254 定时器、8259A 可编程中断控制器、8237DMA 控制器等的状态，测试其是否正常工作
- **系统初始化**
  - 针对 CPU Cache、DRAM、南北桥芯片组、显卡、PCI 设备控制器、IDE 设备控制器、网卡等的寄存器作初始化操作, 填充相应寄存器的值, 设定成可支持的默认工作模式, 并检测是否能够正常工作
- **提供常驻内存的运行时服务 (Runtime services)**
  - 这些服务程序常驻在某段系统内存中, 操作系统和应用程序能够通过中断方式调用这些服务代码, 典型的如 Int 10h、Int13h、Int 15h 等
- **系统设置**
  - BIOS 提供文本或图形界面的设置程序 (通常称为 BIOS Setup) 供用户在进入操作系统前对系统的一些参数值进行设置，如 BOS 密码、光盘/硬盘/软盘引导顺序、系统时间等
- 引导操作系统

2.3.3 传统的 BIOS 缺陷

- 传统固件 BIOS 仍然运行在 16 位实模式下，这使得 Pentium 4 的 CPU 在 BIOS 运行阶段只相当于一块高速的 8086 中央处理器。不能充分的发挥中央处理器硬件新技术上的优势
- 传统固件 BIOS 封闭的开发方式严重阻碍了固件技术发展。传统固件 BIOS 缺乏统一和开放的架构，不同固件 BIOS 厂商其 BIOS 产品结构设计千差万别
- 传统固件 BIOS 的设计，一开始就缺乏安全方面的考虑

2.4 内存

2.4.1 RAM

主存通常被称做 RAM(Random Access Memory)，由于 1950 年代和 1960 年代的计算机使用微小的可磁化铁氧体磁芯作为主存储器，因此有时将其称为核心存储器。所有不能再高速缓存中得到满足的内存访问请求都会转往主存中。

2.5 硬盘

2.6 总线

2.7 设备

2.7.1 I/O 设备

I/O 设备是用于和计算机进行通信的外部硬件。输入/输出设备能够向计算机发送数据并从计算机接收数据。

I/O 设备通常由机械组件 (mechanical component) 和电子组件 (electronic component) 构成，其中电子组件被称为设备控制器 (device controller) 或者适配器 (adapter)。I/O 设备通常采用可插入 PCI (Periheral Component Interconnect) 扩展插槽的主板上的芯片或印刷电路卡的形式。

编址方式

- **统一编址**
  - 将 I/O 设备当做存储器地址的一部分
- **独立编址**
  - 将 I/O 设备与存储器地址分开，使用专门的 I/O 指令访问

2.7.2 设备控制器

设备控制器是处理 CPU 传入和传出信号的系统。设备通过插头和插座连接到计算机，并且插座连接到设备控制器。设备控制器从连接的设备处接收数据，并将其存储在控制器内部的一些特殊目的寄存器 (special purpose registers) 也就是本地缓冲区中。

2.8 I/O

2.8.1 I/O 接口

接口是两个系统或两个部件之间的交接部分，可以是两种硬设备之间的连接电路，也可以是两个软件之间的共同逻辑边界。

I/O 接口是主机与 I/O 设备间设置的硬件电路及其相应的软件控制，用于实现设备的选择、数据缓冲、通过接口传送控制命令等功能。

I/O 端口指接口电路中的一些寄存器这些寄存器用于存放数据、控制命令、状态信息等。CPU 对 I/O 接口（或 I/O 设备）的信息读写，实际上都是对端口的操作。

I/O 接口类型

- **按数据传送方式**
  - 并行接口 / 串行接口
- **按功能选择分类**
  - 可编程 / 不可编程接口
- **按通用性分类**
  - 通用接口 / 专用接口
- **按数据传送的控制方式分类**
  - 程序型接口：用于低速设备，采用程序中断方式
  - DMA 接口：用于连接高速 I/O 设备

2.8.2 内存映射 I/O

每个设备控制器都会有几个寄存器用来和 CPU 进行通信。通过写入这些寄存器，操作系统可以命令设备发送数据，接收数据、开启或者关闭设备等。通过从这些寄存器中读取信息，操作系统能够知道设备的状态，是否准备接受一个新命令等。

为了控制寄存器，许多设备都会有数据缓冲区 (data buffer)，来供系统进行读写。例如，在屏幕上显示一个像素的常规方法是使用一个视频 RAM，这一 RAM 基本上只是一个数据缓冲区，用来供程序和操作系统写入数据。

CPU 与设备寄存器和设备数据缓冲区进行通信有三种方式。第一种方法是，每个控制寄存器都被分配一个 I/O 端口号，端口号是一个 8 位或 16 位的整数。操作系统可以使用类似 `IN REG,PORT` / `OUT PORT,REG` 的指令访问。

第二种方式是将所有控制寄存器映射到内存空间中。

第三种方式是一种混合方式，这种方式具有与内存映射 I/O 的数据缓冲区，而控制寄存器则具有单独的 I/O 端口。

在这种方式下，当 CPU 想要读入一个字的时候，将需要的地址放到总线地址线上，然后在总线的一条控制线上调用一个 READ 信号。还有第二条信号线来表明需要的是 I/O 空间还是内存空间。如果是内存空间，内存将响应请求。如果是 I/O 空间，那么 I/O 设备将响应请求。如果只有内存空间，那么每个内存模块和每个 I/O 设备都会将地址线和它所服务的地址范围进行比较。如果地址落在这一范围之内，它就会响应请求。

2.8.3 直接内存访问

无论一个 CPU 是否具有内存映射 I/O，都需要寻址设备控制器以便与交换数据。CPU 可以从 I/O 控制器每次请求一个字节的数据，但是这么做会浪费 CPU 时间，所以经常会用到一种称为直接内存访问 (Direct Memory Access) 的方案。

直接内存访问是一种完全由硬件执行 I/O 交换的工作方式，主存与 I/O 设备间高速交换批量数据，传送速度快。硬件 DMA 控制器从 CPU 完全接管对总线的控制，数据交换不经过 CPU，直接在主存和 I/O 设备之间进行。

2.8.4 参考链接

- I/O 设备

2.9 内核

2.9.1 宏内核

宏内核（Monolithic Kernel）是操作系统核心架构的一种，此架构的特性是整个核心程序都是以核心空间（Kernel Space）的身份及监管者模式（Supervisor Mode）来运行。

宏内核通常是以单一静态二进制文件的方式被存储在磁盘，或是高速缓存上，在开机之后被加载存储器中的核心空间，并开始运作。

宏内核的优点是效率高、反应快，但是耦合度高、灵活性差。

2.9.2 微内核

微内核只有最基本的操作系统功能。非基本的服务和应用都在微内核之外构造。

微内核有着扩展性强、灵活、可移植、支持分布式系统等优势。

CHAPTER 3

嵌入式设备

3.1 固件解包

3.1.1 固件

在电子系统和计算中，固件是一种特定的计算机软件类，它为设备的特定硬件提供低级控制。固件可以为设备的更复杂的软件提供标准化的操作环境，或者对于较不复杂的设备，充当设备的完整操作系统，执行所有的控制、监视和数据操作功能。包含固件的设备的典型例子是嵌入式系统、消费设备、计算机、计算机外设等。几乎所有电子设备都包含固件。

计算机固件是计算机系统中不可缺少的底层基础系统。这些固件往往是以软件的形式固化存储在硬件芯片中。计算机主板上有最重要、最核心的计算机固件，通过称为 BIOS(Basic Input / Output System，基本输入输出系统)，计算机加电时，中央处理器 (CPU) 取得并执行的第一条指令，就储存在固件中。

3.1.2 获取固件

- 直接从开发团队、制造商/供应商或用户获取
- 使用制造商提供的项目从头编译
- 从供应商的网站获取
- 从论坛、博客，或官方评论中获取
- 设备更新进行中间人（MITM）获取
- 通过 UART、JTAG、PICit 等直接从硬件中提取
- 嗅探“硬件组件中的串行通信”中的更新服务器请求
- 通过移动应用程序中的硬编码接口
- 将固件从引导加载程序（如：U-boot ）转储到闪存或通过 tftp 的网络转储
- 从主板卸下闪存芯片（如：SPI ）或 MCU，以进行离线分析和数据提取

3.1.3 文件系统

常见的文件系统类型有 squashfs、ubifs、romfs、rootfs、jffs2、yaffs2、cramfs、initramfs 等。

3.1.4 常见模式

无加密，使用 binwalk 等工具解包即可。

加密但是在运行后会解密，提取 nvram 可获得固件。

若之前版本存在无加密固件，可通过劫持升级过程获得新版本固件。

3.1.5 参考链接

- OWASP Firmware Security Testing Methodology

3.2 HAL

3.2.1 硬件抽象层

硬件抽象层是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化。它隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，可在多种平台上进行移植。从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，使得软硬件测试工作的并行进行成为可能。

3.3 硬件

3.3.1 NVRAM

NVRAM（Non-Volatile Random Access Memory）是非易失性随机访问存储器，指断电后仍能保持数据的一种 RAM。

3.3.2 UART

UART(Universal Asynchronous Receiver/Transmitter)，译作通用异步收发传输器，用于将要传输的资料在串行通信与并行通信之间加以转换。作为把并行输入信号转成串行输出信号的芯片，UART 通常被集成于其他通讯接口的连结上。

3.4 4G

3.4.1 参考链接

- IMP4GT Attacks

3.5 BLE

3.5.1 简介

蓝牙低功耗（Bluetooth Low Energy，BLE），是对经典蓝牙 BR/EDR 技术的补充，多用于小数据率、离散传输的应用。虽然 BLE 和经典蓝牙都是蓝牙标准，但是 BLE 并不兼容经典蓝牙 BR/EDR。相比于经典蓝牙，它具有覆盖范围更广，安全性更高，功耗低等特点。

3.5.2 架构

物理层

蓝牙物理层（RF 层），包括 BR/EDR、LE 以及 AMP 三种，主要负责在物理 channel 上收发蓝牙 packet。对 BR/EDR 和 LE RF 来说，还会接收来自 Baseband 的控制命令来控制 RF 频率的选择和 timing。而 AMP PHY，则是使用 802.11（WIFI）的规范。

基带

Link Controller 和 Basebandresource management 组成了蓝牙的基带（baseband）。Link Controller 负责链路控制，主要是根据当前物理 channel 的参数、逻辑 channel 的参数、逻辑 transport 的参数将数据 payload 组装成 bluetoothpacket。另外，通过 LinkControl Protocol（对 LE 来说是 LL LayerProtocol），可以实现流控、ack、重传等机制。Baseband resourcemanagement，主要用于管理 RF 资源。

Link Manager

Link Manager 主要负责创建、修改、释放蓝牙逻辑连接（Logical Link），同时也负责维护蓝牙设备之间物理连接（Physical Link）的参数。它的功能主要是通过 Link Management Protocol（LMP，for BR/EDR）和 Link Layer Protocol（LL，for LE）完成。

Device Manager

Device Manager 主要负责控制蓝牙设备的通用行为（蓝牙数据传输除外的行为），包括：搜索附近的蓝牙设备，连接到其他的蓝牙设备。

HCI

蓝牙系统分为 Bluetooth Controller 和 Bluetooth Host 两个大的模块，它们之间通过 HCI（Host Controller Interface）接口以 HCI 协议进行通信。

L2CAP

L2CAP 位于 Bluetooth Host 中，包括两个子模块：Channel Manager 主要负责创建、管理、释放 L2CAP channel。L2CAP Resource Manager 负责统一管理、调度 L2CAP channel 上传递的 PDU（Packet Data Unit），以确保那些高 QoS 的 packet 可以获得对物理信道的控制权。

SMP

SMP（Security Manager Protocol）是一个点对点的协议，基于专用的 L2CAP channel，用于生成加密（encryption）和识别（identity）用的密匙（keys）。

SDP

SDP（Service Discover Protocol）也是一个点对点的协议，基于专用的 L2CAP channel，用于发现其它蓝牙设备能提供哪些 profile 以及这些 profile 有何特性。在了解清楚了其他蓝牙设备的 profile 以及特性之后，本蓝牙设备可以发起对自己感兴趣的蓝牙 profile 的连接动作。

AMP Manager

基于 L2CAP channel，和对端的 AMP manager 交互，用于发现对方是否具备 AMP 功能，以及收集用于建立 AMP 物理链路的信息。

GAP

GAP（Generic Access Profile）是一个基础的蓝牙 profile，用于提供蓝牙设备的通用访问功能，包括设备发现、连接、鉴权、服务发现等等。

GAP 是所有其它应用模型的基础，它定义了在 Bluetooth 设备间建立基带链路的通用方法。还定义了一些通用的操作，这些操作可供引用 GAP 的应用模型以及实施多个应用模型的设备使用。GAP 确保了两个蓝牙设备（不管制造商和应用程序）可以通过 Bluetooth 技术交换信息，以发现彼此支持的应用程序。

3.5.3 工作流程

- 蓝牙启动
- 扫描设备
- 设备配对
- 数据传输

设备配对

- **生成初始密钥**
  - 初始密钥 Kinit 长度为 128bit, 由 E22 算法生成
  - 首先提出通信的设备为主设备 (Master)，用 A 表示
  - 被动通信的设备为从设备 (Slave)，用 B 表示
  - **E22 算法的输入**
    * 从设备的物理地址 (BD_ADDR)
    * PIN 码及其长度
    * 128 bit 的随机数 (IN_RAND)，由 A 产生，并明文传输给 B
- **生成链路密钥**
  - A 产生 128 位的随机数 LK_RANDA，B 产生随机数 LK_RANDB
  - Kinit 与 LK_RANDA，发送给 B
  - Kinit 与 LK_RANDB，发送给 A
  - 用 E21 加密 LK_RANDA、LK_RANDB、BD_ADDRA、BD_ADDRB，结果异或得到 Kab
- **双方认证**
  - 使用挑战-应答
  - A 为应答，B 为请求
  - A 产生 128 bit 的随机数 AU_RANDA，明文传输至 B
  - A、B 使用 AU_RANDA、Kab、BD_ANDRB 加密运算生成 32 位的 SRESA 和 SRESB
  - B 将 SRESB 传给 A，A 比较 SRESA 和 SRESB，相同则通过

3.5.4 协议栈

Physical Layer

蓝牙使用 2.4GHz 频道，自适应跳频

Link Layer

- 控制设备的射频状态
- Standby
- Advertising
- Scanning
- Initiating
- Connection

HCI

- 主机控制接口层
- Host 和 Controller 的通信协议
- 软硬件接口

Generic Access Profile (GAP)

- 控制设备连接和广播
- **role**
  - Broadcaster：设备发送 Advertising Events
  - Observer: 设备在接受 Advertising Events
  - Peripheral：设备接受 Link Layer 连接
  - Central: 设备发起 Link Layer 连接
- **通信模式**
  - Broadcast Mode and Observation Procedure: 单向无连接通信
  - Discovery modes and procedure: 设备发现
  - Connection modes and procedure: 设备连接
  - Bonding modes and procedure: 设备配对
- Logical Link Control and Adaptation Protocol (L2CAP Protocol)
- **Security Manager (SM)**
  - 配对
  - 认证
  - 加密
- Attribute Protocol (ATT)
- **Generic Attribute Profile (GATT)**
  - 接受和处理主从设备的指令信息，并打包成合适的 profile
  - **Services**
    * 将数据分为独立逻辑项，包含一个或多个 Characteristic
    * 每个 Service 有一个 UUID 标识
  - **Characteristic**
    * 最小的逻辑数据单元

3.5.5 攻击

MITM

B 修改自身为 A 的地址，以 A 的身份和 C 通信 B 修改自身为 C 的地址，以 C 的身份和 A 通信

离线 PIN 码

暴力攻击 PIN 码

中继攻击

以中继的方式放大蓝牙信号

DoS

发起大量的鉴权/文件传送等请求，使设备不能正常工作

3.5.6 分析工具

嗅探工具

- **cc2540 / cc2541**
  - 只能监听一个信道
  - 低价替补方案
- **MRF51822**
  - 数据包捕获不稳定
- Frontline BPA 600
- Ellsys BEX400
- **Ubertooth**
  - 支持有限
- **HackRF**
  - 针对蓝牙，使用复杂

扫描器

- bleah
- bluelog
- btCrawler
- Kismet
- BLE Sniffer
- Blue Scanner

攻击

- BtleJuice
- Blueranger
- Bluebugging
- Peripheral hijacking

模拟器

- hackmelock

手机端

- Bluez
- LightBlue
- nRF Connect
- Ramble
- hackmelock

3.5.7 安全问题

基础

- 配置管理
- 身份鉴别
- 认证授权
- 会话管理
- 输入验证
- 错误处理

业务安全

- 账户管理
- 云端密码下发
- 远程开门
- 开锁记录

网络传输

- 明文传输
- 加密漏洞
- 数据包逆向
- OTA 固件拦截

蓝牙通信

- 协议分析
- 明文传输
- 信号重放
- 信号拦截
- 操作指令

3.5.8 参考链接

- BLE 安全现状与挑战

3.6 可信启动

3.6.1 简介

安全启动机制一般包括多种机制，主要是：

- 用于在启动目标镜像之前对其进行验证。
- 在更新前验证更新固件的完整性，完全擦除旧固件以及防止攻击者恢复为已知状态的回滚防止机制。
- 用于安全维护和证明设备内部信任链的机制，

3.6.2 参考链接

- Secure Firmware Development Best Practices

3.7 路由设备

3.7.1 常见厂商

- ASUS
- D-Link
- Linksys
- Mercury

- Netgear
- TP-Link
- Tenda

3.7.2 常见协议

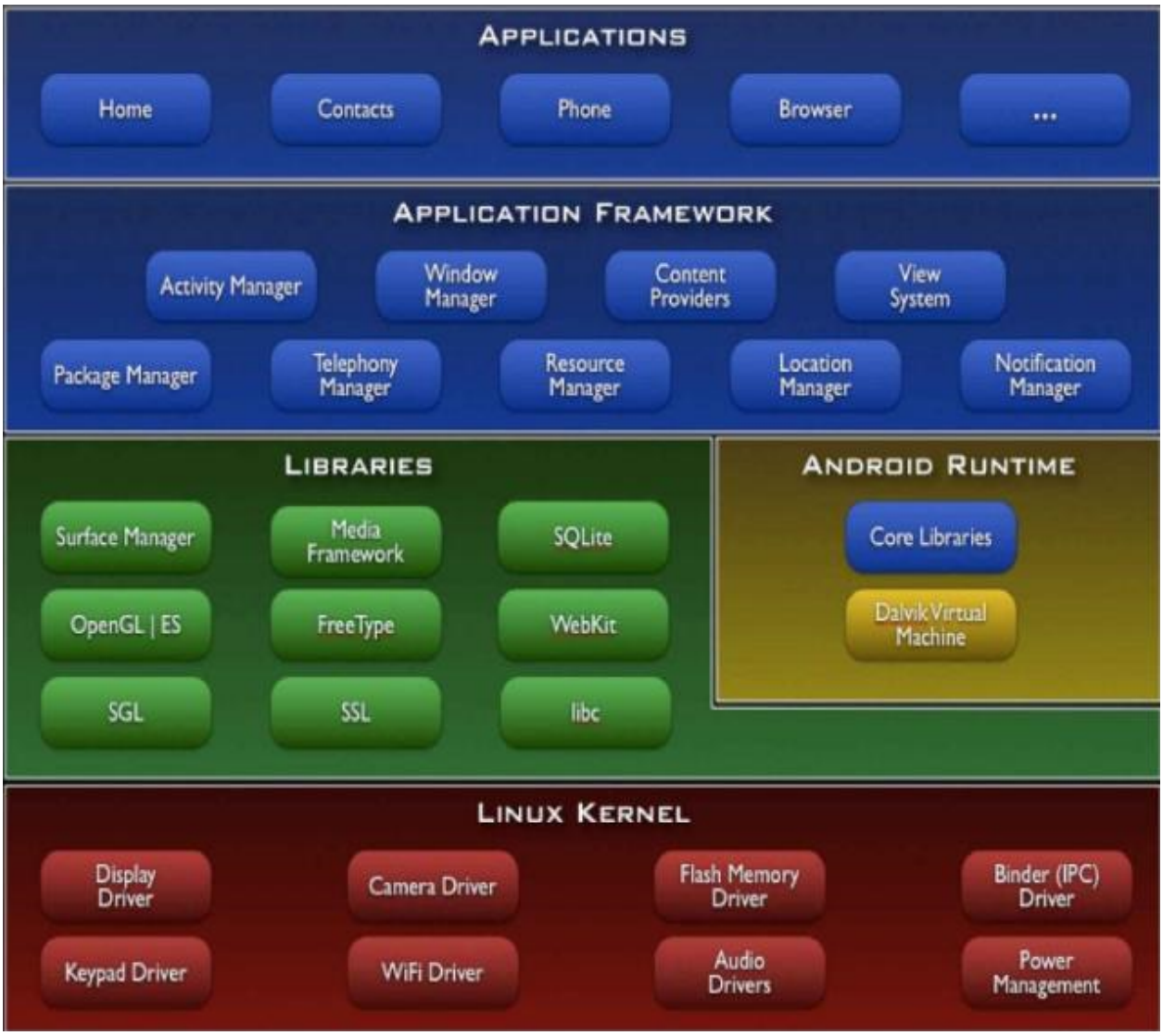
- ARP
- DHCP
- FTP
- TFTP
- HTTP
- SSH
- Telnet
- DNS
- ICMP

3.8 Android

3.8.1 简介

Android 是一种基于 Linux 的自由开源操作系统，主要用于移动设备，如智能手机和平板电脑，由 Google 公司和开放手机联盟领导及开发。

3.8.2 系统框架



3.8.3 安全机制

进程沙箱隔离

Android 应用程序在安装时被赋予独特的用户标识（UID），并永久保持。应用程序及其运行的 Dalvik 虚拟机运行在独立的 Linux 进程空间，与其他程序完全隔离。

应用程序签名

基于共享内存的 Binder 实现，提供轻量级的 RPC，通过 AIDL 定义接口和数据类型，确保不会溢出。

内存管理机制

基于 Linux 的低内存管理机制 OOM (Out Of Memory Killer)，设计实现了地图的 LMK (Low Memory Killer) 机制。清理低级别进程的内存空间。

引入 Ashmen，清理不再使用的共享内存区域。

其它

- 权限声明机制
- 访问控制机制
- 进程通信机制

3.8.4 组件

Activity

Activity(用户界面) 是应用程序与用户进行人机交互的可视化用户界面，包含很多与用户交互的构件，如按钮、文本输入框等。一个 Android 应用程序可以有多个 Activity。

Activity 共有四种状态

- Active：激活或运行，这时 Activity 在屏幕前台
- Pause：暂停状态，这时 Activity 失去焦点，但对用户可见
- Stop：停止状态，这时 Activity 被其他 Activity 覆盖而完全变暗
- Inactive：终止状态，这时 Activity 被系统清理出内存

Service

Service(后台进程) 是 Android 系统运行在后台的服务，不提供用户界面;

Content Provider

Content Provider(内容提供器) 是一种 SQL-数据库，用于为应用程序提供数据，同时为应用程序中的数据共享提供支持;

Broadcast Receivers

Broadcast Receivers(广播接收器) 是接收广播消息的邮箱。

3.8.5 Dalvik

基于寄存器的虚拟机，完成对象生命周期、堆栈、线程的管理、安全和异常的管理以及垃圾回收等。每一个 Android 都运行在一个单独的 Dalvik 虚拟实例上。

3.8.6 APK 结构

- res 目录：资源文件
- META_INF：数字签名信息
- lib 目录：动态链接库
- assets 目录：原始格式的文件
- resource.arsc 文件：编译后的二进制资源文件
- classes.dex：Dalvik 可执行二进制文件
- AndroidManifest.xml

3.8.7 数据存储

- Applications `/data/app/<package-name>/`
- Shared Preferences Files `/data/app/<package-name>/shared_pref`
- SQLite Database `/data/app/<package-name>/databases`
- Internal Storage `/data/app/<package-name>/files`

3.8.8 恶意软件分类

按行为分

- **恶意软件安装 (malware installation)**
  - 重打包 (repackaging)
  - 更新攻击 (update attack)
  - 诱惑下载 (drive-by download)
  - 其他
- 恶意软件运行 (activation)
- **恶意载荷 (malicious payloads)**
  - 提权攻击 (privilege escalation)
  - 远程控制 (remote control)
  - 付费 (financial charge)

    - 信息收集 (information collection)
- 权限使用 (permission uses)

按类别分

- 木马类
- 病毒类
- 后门类
- 僵尸类
- 间谍软件类
- 恐吓软件类
- 勒索软件类
- 广告软件类
- 跟踪软件类

3.8.9 参考链接

- Developer Android

3.9 iOS

3.9.1 系统结构

所有 iOS 设备中，系统与硬件都高度集成，从系统启动、系统更新、应用的安装、应用的运行时等多个方面来保全系统的安全，具体包括以下几个限制。

所有 iOS 设备在处理器内都集成有一段名为 Boot Room 的代码，此代码被烧制到处理其内的一块存储上，并且只读，可以认为是完全可信的。系统启动时，Boot Room 通过苹果的 Apple Root CA Public 证书对 Low-Level BootLoader 进行验证，如果通过验证，Low-Level BootLoader 将运行 iBoot，较 Low-Level Bootloader 高层次的 Bootloader，如果这一步也通过，那么 iBoot 将运行 iOS 的内核, XNU，系统开始运行。以上这几个步骤任一步骤无法通过，都将导致系统无法启动，这样，处理期内烧制的 Boot Room 保证了 iOS 系统只能在 Apple 自家设备上运行，而这些设备也将无法运行 iOS 之外的系统。

iOS 设备的系统升级之后是不允许降级的。这样做的好处是系统的安全等级只会越来越高，二不会出现由于系统降级，已修复安全风险又暴露出来的问题。iOS 系统在升级过程需要联网进行验证，系统升级之前，设备会将 LLB、iBoot、内核、镜像，外加一个随机的不可重复的值发送到苹果的服务器进行验证，服务器端对所有这些进行验证，如果通过验证，将会返回一个通过的结果，结果加入了与设备唯一相关的 ECID。这样做的好处是此值是无法重用的，只能对应与一台设备，且只能使用一次。通过这种机制，保证了系统升级过程都是符合苹果要求的。

所有运行在 iOS 上的代码都是需要签名的。苹果自带应用已经打上了苹果的签名，而第三方应用，则需要开发者账号进行签名，而开发者账号都是通过苹果官方实名审核的账号，从开发者源头上控制了程序的安全性，也就是说，系统内所有运行的程序都是可信的，且知道来源的。

运行与 iOS 系统的第三方软件都是运行与 sandbox 之内，每个第三方程序都有自己的独占的路径，其只能访问独占路境内的内容，其他程序的文件一般情况下无法访问，如果要访问，只能通过苹果官方 API，而不能自行操作文件。连个应用之间无法共享文件，如要互相通信，只能通过 URL Schema 或 shared key chain。另

外，每个应用都有其运行权限，不同权限可进行的操作是不同的，将应用的权限限制在其需要的范围内，而不赋予额外的权限。

3.9.2 数据的加密与保护

强制加密

而 iOS 内所有用户数据都是强制加密的，加密功能不能关闭。所以，苹果的 AES 加解密引擎都是硬件级的，位于存储与系统之间的 DMA 内，这样提供了较高的效率与性能。加密解密使用的 KEY 主要来自 unique ID(UID) 以及 Group ID(gid)，UID 与唯一设备相关，GID 与某种特定型号的 CPU 相关，一台设备的 UID 及 GID 全部被烧制到芯片内部，除了 AES 加密引擎，没有其他方法直接读取，能看到的只有使用 UID 及 GID 加密后的数据。这样，不同设备的加密结果是不同的，同一套密文只能在加密的机器上进行解密。除了 GID 及 UID，其他加密使用的 KEY 全部来自系统自带的随机数生成器，具体使用的算法为 Yarrow。

数据保护

iOS 提供了名为 File Data Protection 的数据保护方法。所有文件在加密时使用的 key 都是不同的，这些 key 被称作的 prefile key，存储于 metafile 内。prefile 的访问需要进行解密的 key，这些 key 包括：

1. File System Key: 系统安装时生成的一个随机的 key
2. Class Key，这个 key 与 UID 相关，如果用户设置了锁屏密码，那么此 Class Key 将的来源也包括锁屏密码。

只有有了这两个 key，一个文件的 prefilekey 才能被读取出来，此加密的文件才能被解密，也就是说，当锁屏之后，或存储位于不同的设备之上，数据是无法读取的。

File System Key 还有一个重要作用，远程删除数据时，其实不用真正的删除磁盘上的数据，只要删除此 key，那么所有文件的 prefile key 将不能访问，也就是所有文件将无法读取。

锁屏密码

锁屏密码为了防止暴力破解，增加了三个限制：

1. 与 uid 绑定，只能在该密码生成的设备上进行尝试，
2. 两次尝试的间隔被强制设成 80ms，机器暴力破解的时间将大大加长
3. 增加选项，如果连续输错次数超过 10 次，可以选择删除设备内数据

Keychain

应用的小量极敏感数据，例如密码，最好存储与 KeyChain 内，而不是应用自己管理。

3.9.3 网络安全

除了本地数据的保护，苹果还对数据的传输提供了多种多样的保护机制。苹果提供了可靠的、可信以及加密的连接。因为 iOS 平台限制了同时监听的端口的数量，并将那些不必要的网络组建，例如 telnet，shell 以及 web server 等，所以不需要防火墙的保护。

3.9.4 设备权限控制

针对企业用户，iOS 系统提供了多样的安全策略，管理原可以根据需求对设备的安全特性进行多样化的设置，包括密码策略，数据保护策略，应用使用策略，远程数据删除等功能，给企业级用户提供了高安全性以及极大的灵活性。

3.10 安全性

3.10.1 常见安全问题

- 硬编码凭证
- 硬编码的 API 端点和后端服务器详细信息

CHAPTER 4

汇编基础

4.1 指令集架构

4.1.1 简介

指令集体系结构 (Instruction set architecture, ISA) 是计算机的抽象模型。它也称为体系结构或计算机体系结构。

4.1.2 分类

ISA 可以通过多种方式分类，常见的分类方式是根据架构的复杂性分为复杂指令集计算机（CISC，complex instruction set computer）和精简指令集计算机（RISC，reduced instruction set computer）。CISC 具有许多专门的指令，其中某些指令可能很少在实际程序中使用。RISC 通过仅有效地执行程序中经常使用的指令来简化处理器。

4.1.3 指令类型

数据处理和内存操作

- 给寄存器设置值
- 将数据从内存拷贝到寄存器
- 将数据从寄存器拷贝到内存
- 从硬件设备中存/取数据

算术和逻辑操作

- 加/减/乘/除并设置状态寄存器
- 位运算
- 比较寄存器中的值
- 浮点运算

控制流操作

- 有条件跳转
- 无条件跳转
- 调用

协处理器

- 从协处理器存/取数据
- 协处理器操作

复杂指令集指令

- 一次性操作多个寄存器 (e.g. 存/取到栈上)
- 移动大块内存 (e.g. DMA 操作)
- 复杂的算术操作 (e.g. 开平方)
- SIMD 操作
- ...

4.2 区别

汇编常用的有 AT&T 和 Intel 两种格式，其区别有下面这些。

4.2.1 前缀

在 AT&T 汇编格式中，寄存器名要加上 `%` 作为前缀，如 `pushl %eax`；而在 Intel 汇编格式中，寄存器名不需要加前缀，如：`push eax`

4.2.2 立即操作数

在 AT&T 汇编格式中，用 `$` 前缀表示一个立即操作数，如 `pushl $1`；而在 Intel 汇编格式中，立即数的表示不用带任何前缀，如 `push 1`

4.2.3 操作数顺序

AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 Intel 汇编格式中，目标操作数在源操作数的左边，如 `addl $1, %eax`；而在 AT&T 汇编格式中，目标操作数在源操作数的右边，如 `add eax, 1`

4.2.4 字长

在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀 `b`、`w`、`l` 分别表示操作数为字节（byte，8bit）、字（word，16bit）和长字（long，32bit），如 `movb val, %al`；而在 Intel 汇编格式中，操作数的字长是用 `byte ptr` 和 `word ptr` 等前缀来表示的，如 `mov al, byte ptr val`

4.2.5 绝对转移和调用指令

在 AT&T 汇编格式中，绝对转移和调用指令（jump/call）的操作数前要加上 `*` 作为前缀，而在 Intel 格式中则不需要。

4.2.6 远程转移指令和远程子调用指令

远程转移指令和远程子调用指令的操作码，在 AT&T 汇编格式中为 `ljump` 和 `lcall`，如 `ljump $section, $offset` `jmp far section:offset`。

而在 Intel 汇编格式中则为 `jmp far` 和 `call far`，即 `lcall $section, $offset` 和 `call far section:offset`。

与之相应的远程返回指令则 AT&T 格式为 `lret $stack_adjust`，Intel 格式为 `ret far stack_adjust`。

4.2.7 寻址

AT&T 汇编格式中，内存操作数的寻址方式是 `section:disp(base, index, scale)`。Intel 汇编格式中，内存操作数的寻址方式为：`section:[base + index*scale + disp]`。

4.3 x86

4.3.1 寄存器

- **ESP**
  - Extended Stack Pointer 栈指针寄存器
  - 指向最上面帧的栈顶
- **EBP**
  - Extended Base Pointer 基址指针寄存器

  - 指向最上面帧的栈底
- **EIP**
  - Extended Instruction Pointer 指令寄存器
  - 指向当前的指令地址
- **EFLAGS**
  - 保存当前执行状态的所有 Flag
- **CR0-CR7**
  - 控制寄存器
  - 保存系统的控制位
- **IDTR**
  - Interrupt Descriptor Table Register
  - 中断寄存器，记录当前中断处理函数的地址
- **IDTR**
  - Global Descriptor Table Register
  - 记录段表

4.3.2 参考链接

- Intel 80386 Reference Programmer's Manual

4.4 x64

4.4.1 常用寄存器

- **RAX**
  - 易失的
  - 返回值寄存器
- **RCX**
  - 易失的
  - 第一个整型参数
- **RDX**
  - 易失的
  - 第二个整型参数
  - 系统调用的第三个参数
- **R8**
  - 易失的
  - 第三个整型参数

- **R9**
  - 易失的
  - 第四个整型参数
- **RDI**
  - 非易失的
  - 必须由被调用方保留
  - 系统调用的第一个参数
- **RSI**
  - 非易失的
  - 必须由被调用方保留
  - 系统调用的第二个参数
- **RBX**
  - 非易失的
  - 必须由被调用方保留
- **RBP**
  - 非易失的
  - 可用作帧指针；必须由被调用方保留
- **RSP**
  - 非易失的
  - 堆栈指针
- **R8**
  - 系统调用的第五个参数
- **R9**
  - 系统调用的第六个参数
- **R10-R11**
  - 易失的
  - 必须根据需要由调用方保留
  - 在 syscall/sysret 指令中使用
  - R10 为系统调用的第四个参数
- **R12-R15**
  - 非易失的
  - 必须由被调用方保留

4.4.2 系统调用号

需要注意的是，x86 和 x64 架构下系统调用号并不相同。系统调用可在 `/usr/include/asm/unistd.h` 文件中查看。

4.5 ARM

4.5.1 基础

出于低功耗、封装限制等原因，以前的一些 ARM 处理器没有独立的硬件浮点运算单元，需要软件来实现浮点运算。随着技术发展，现在的高端 ARM 处理器基本都具备了硬件执行浮点操作的能力。

因此新旧两种架构之间的差异就产生了两种不同的嵌入式应用程序二进制接口 (EABI) 软浮点与矢量浮点 (VFP)。其中软浮点 (soft float) 和硬浮点 (hard float) 之间有向前兼容却没有向后兼容的能力。

在 ARM 体系架构内核中，在没有 fpu 内核的情况下，是不能使用 armel 和 armhf 的。在有 fpu 的情况下，就可以通过 gcc 的选项 `-mfloat-abi` 来指定使用哪种，有如下三种值：

- soft：不用 fpu 计算，即使有 fpu 浮点运算单元也不用。
- armel：(arm eabi little endian) 也即 softfp，用 fpu 计算，但是传参数用普通寄存器传，这样中断的时候，只需要保存普通寄存器，中断负荷小，但是参数需要转换成浮点的再计算。
- armhf：(arm hard float) 也即 hard，用 fpu 计算，传参数用 fpu 中的浮点寄存器传，省去了转换性能最好，但是中断负荷高。
- arm64：64 位的 arm 默认就是 hard float 的，因此不需要 hf 的后缀。

使用 softfp 模式，会存在不必要的浮点到整数、整数到浮点的转换。而使用 hard 模式，在每次浮点相关函数调用时，平均能节省 20 个 CPU 周期。对 ARM 这样每个周期都很重要的体系结构来说，这样的提升无疑是巨大的。

在完全不改变源码和配置的情况下，在一些应用程序上，虽然 armhf 比 armel 硬件要求高一点，但是 armhf 能得到 20-25% 的性能提升。对一些严重依赖于浮点运算的程序，更是可以达到 300% 的性能提升。

4.5.2 工作模式

Arm 处理器有 7 种基本工作模式：

- **User**
  - 非特权模式，大部分任务执行在这种模式
- **FIQ**
  - 当一个高优先级 (fast) 中断产生时将会进入这种模式
- **IRQ**
  - 当一个低优先级 (normal) 中断产生时将会进入这种模式
- **Supervisor**
  - 当复位或软中断指令执行时将会进入这种模式
- **Abort**
  - 当存取异常时将会进入这种模式
- **Undef**

  - 当执行未定义指令时会进入这种模式

- **System**
  - 使用和 User 模式相同寄存器集的特权模式

其中除 User(用户模式) 是 Normal(普通模式) 外，其他 6 种都是 Privilege(特权模式)。

4.5.3 工作状态

ARM 体系的 CPU 有 ARM 和 THumb 两种工作状态，ARM 状态执行字对齐的 32 位 ARM 指令。THumb 执行半字对齐的 16 位指令。

4.5.4 Semihosting

Semihosting 技术将应用程序中的 IO 请求通过一定的通道传送到主机 (host)，由主机上的资源响应应用程序的 IO 请求，而不是像在主机上执行本地应用程序一样，由应用程序所在的计算机响应应用程序 IO 请求，也就是将目标板的输入/输出请求从应用程序代码传递到远程运行调试器的主机的一种机制。

4.6 MIPS

4.6.1 简介

MIPS 的意思是“无内部互锁流水级的微处理器”(Microprocessor without interlocked piped stages)，其机制是尽量利用软件办法避免流水线中的数据相关问题。它最早是在 80 年代初期由斯坦福大学 Hennessy 教授领导的研究小组研制出来的。MIPS 公司的 R 系列就是在此基础上开发的 RISC 工业产品的微处理器。这些系列产品为很多计算机公司采用构成各种工作站和计算机系统。

4.6.2 指令特点

- 所有指令都是 32 位长
- **指令操作必须符合流水线**
  - MIPS 指令一次只能修改一个寄存器的值
- 3 操作数指令
- 32 个寄存器
- 没有条件标志位

4.6.3 数据类型

- 所有 MIPS 指令都是 32 位
- 单个字符用单引号，例如：'b'
- 字符串用双引号，例如："A string"

4.6.4 寄存器

- MIPS 下一共有 32 个通用寄存器
- 在汇编中，寄存器标志由 `$` 符开头
- **寄存器表示可以有两种方式**
  - 直接使用该寄存器对应的编号，例如：从 `$0` 到 `$31`
  - **使用对应的寄存器名称**
    * 例如：`$t1`, `$sp`
- 栈的走向是从高地址到低地址

寄存器编号及其用途

寄存器编号	寄存器名	寄存器用途
0	zero	永远返回零
1	$at	汇编保留寄存器
2-3	$v0 - $v1	存储表达式或者是函数的返回值
4-7	$a0 - $a3	存储子程序的前 4 个参数，在子程序调用过程中释放
8-15	$t0 - $t7	临时变量
16-23	$s0 - $s7	静态变量
24-25	$t8 - $t9	临时变量
26-27	$k0 - $k1	中断函数返回值
28	$gp	指向静态数据块的中间地址
29	$sp	栈顶指针
30	$fp	帧指针
31	$ra	返回地址

4.6.5 读写操作

- **load word**
  - `lw register_destination, RAM_source`
- **store word**
  - `sw register_source, RAM_destination`
- **store byte**
  - `sb register_source, RAM_destination`
- **load immediate**
  - `li register_destination, value`

4.6.6 寻址

- **直接寻址**
  - `la $t0, var1`
- **间接寻址**
  - `lw $t2, ($t0)`
- **偏移**
  - `lw $t2, 4($t0)`

4.6.7 算术指令集

- **sub**
  - `sub $t2,$t3,$t4`
  - `$t2 = $t3 - $t4`
- **addi**
  - `addi $t2,$t3, 5`
  - `$t2 = $t3 + 5;`
  - add immediate
- **addu**
  - `addu $t1,$t6,$t7`
  - `# $t1 = $t6 + $t7;`
  - add as unsigned integers
- **subd**
  - `subu $t1,$t6,$t7`
  - `$t1 = $t6 + $t7;`
  - subtract as unsigned integers
- **mult**
  - `mult $t3,$t4`
  - multiply 32-bit quantities in $t3 and $t4
  - store 64-bit result in special registers Lo and Hi
  - `(Hi,Lo) = $t3 * $t4`
- **div**
  - `div $t5,$t6`
  - `Lo = $t5 / $t6`
  - `Hi = $t5 mod $t6`
  - `mfhi $t0`
  - move quantity in special register Hi to $t0

  - `mflo $t1`
  - move quantity in special register Lo to $t1

4.6.8 控制流

- **`b target`**
  - unconditional branch to program label target
- **`beq $t0,$t1,target`**
  - branch to target if $t0 = $t1
- **`blt $t0,$t1,target`**
  - branch to target if $t0 < $t1
- **`ble $t0,$t1,target`**
  - branch to target if $t0 <= $t1
- **`bgt $t0,$t1,target`**
  - branch to target if $t0 > $t1
- **`bge $t0,$t1,target`**
  - branch to target if $t0 >= $t1
- **`bne $t0,$t1,target`**
  - branch to target if $t0 <> $t1
- **`j target`**
  - unconditional jump to program label target
- **`jr $t3`**
  - jump to address contained in $t3
- **`jal sub_label`**
  - copy program counter (return address) to register $ra (return address register)

4.7 PowerPC

CHAPTER 5

编译原理

5.1 基础知识

5.1.1 编译器

简单来说，编译程序是一种翻译程序，它将高级语言所写的源程序翻译成等价的机器语言或汇编语言的目标程序。

解释程序也是一种翻译程序，它将源程序作为输入并执行之，解释程序与编译程序的主要区别是解释程序的执行过程中不生成目标程序。

编译的过程大致可分为词法分析、语法分析、语义分析、中间代码生成、代码优化、代码生成等几步。

词法分析阶段的任务是对构成源程序的字符串从左到右进行扫描和分解，根据语言的词法规则，识别出一个一个具有独立意义的 token。

词法规则是单词符号的形成规则, 它规定了哪样的字符串构成一个单词符号。

语法分析的任务是在词法分析的基础上, 根据语言的语法规则从单词符号串中识别出各种语法单位 (如表达式、说明、语句等) , 并进行语法检查, 即检查各种语法单位在语法结构上的正确性。

语言的语法规则规定了如何从单词符号形成语法单位，语法规则是语法单位的形成规则。

语义分析的任务是首先对每种语法单位进行静态的语义审查，然后分析其含义，并用另一种语言形式 (比源语言更接近于目标语言的一种中间代码或直接用目标语言) 来描述这种语义。

代码优化的任务是对前阶段产生的中间代码进行等价变换或改造，以期获得更为高效，（省时间和空间）的目标代码。优化主要包括局部优化和循环优化等。

5.1.2 相关书籍

编译原理领域最有名的三本书昵称为龙、虎、鲸。

龙书原名 `Compilers: Principles, Techniques, and Tools` ，由 Alfred V.Aho 等人编写。主要讨论了编译器设计的主题，包括词法分析、语法分析、语法制导分析、类型检查、运行环境、中间代码生成、代码生成、代码优化等问题。

虎书原名 `Modern Compiler Implenentation in C` 由 Andrew W.Apple 等人编写。虎书与龙书内容相似，但是增加了数据流分析、循环优化、内存管理等内容。

鲸书原名 `Advanced Compiler Design and Implementation` 由 Steven S.Muchnick 编写，和前两本书相比，鲸书涉及到了更多的比较高级的编译器实现。

5.1.3 编译阶段

编译第一阶段为预处理阶段，涉及以下操作

- 执行预编译指令 #define 等
- 处理条件预编译指令 #ifdef 等
- 删除注释
- 添加行号和文件名标识
- 保留 #pragma 预编译指令
- `gcc -E hello.c -o hello.i`

编译阶段，对预处理完的文件进行词法分析、语法分析、语义分析以及优化后生成相应的汇编代码。将文本文件 hello.i 翻译成文本文件.s，包含一个汇编语言程序。可使用 gcc 的 `-S` 参数，例如 `gcc -S hello.i -o hello.s` 。

汇编阶段，汇编器将.s 文件翻译成机器语言指令，使用 gcc 的 `-c` 参数，例如 `gcc -c hello.s -o hello.o` 。

链接阶段，编译器以动态链接或静态链接的方式生成二进制文件。

5.2 词法分析

词法分析的任务是对字符串表示的源程序从左到右地进行扫描和分解，根据语言的词法规则识别出一个一个具有独立意义的单词符号。

其中语言的单词符号是指语言中具有独立意义的最小语法单位。词法分析程序所输出的单词符号通常表示成如下的二元式：< 单词种别，单词自身的值 > 。

5.3 语法分析

5.3.1 简介

语法分析以词法分析生成的单词符号序列作为输入，根据语法规则识别出语法成分。

5.3.2 分析方法

自上而下

自上而下的分析法从文法开始符号开始，正向推导。

自下而上

自下而上的分析法从给定的输入串开始，逐步归约。

5.3.3 语法

上下文无关语法

上下文无关语法（context-free grammar）有四个模块

- 一组终结符（terminal symbols），有时候又叫 token
- 一组非终结符（nonterminal）syntactic variables.
- 一组产生式（productions）
- 一个非终结符作为起始符号

二义性文法

ambiguous grammars

5.3.4 错误处理

错误发生时，有一些可选的方法，最简单一种处理方式是遇到第一个错误就退出，但是当错误很多的时候，最好能够容忍，并报出之后的错误。

Panic-Mode Recovery

当遇到错误的时候，就开始丢弃遇到的符号，直到遇到一个终结符，例如分号等。

Phrase-Level Recovery

这个方式是遇到错误的时候，替换一个能让当前 parse 执行下去的符号。

Error Productions

把错误的产生式加到语法解析里面。

Global Corrections

还有一种做法是找到一个最近的能正常 parse 的源程序然后往下，但是这种做法太过消耗时间和空间资源，实际情况下使用的情况较少。

5.4 语义分析

5.5 中间代码生成

5.6 代码优化

5.6.1 简介

代码优化对程序实施各种等价变化，使得变换后程序能生成高效率的目标代码。好的编译器需要生成目标代码占用的存储空间少，且运行时间短。另外，编译器的编译时间、资源消耗也是需要考虑的指标。

5.6.2 优化方式

代码优化可以分为与机器有关的优化和与机器无关的优化。

机器无关的优化主要有下面几种方式：

- **局部优化**
  - 合并已知量
  - 删除公共子表达式（删除多余的运算）
  - 删除无用赋值
- **循环优化**
  - 代码外提
  - 删除归纳变量
  - 强度削弱
- **全局优化**
  - 在整个程序范围内进行优化，需要进行数据流分析，代价很高

5.6.3 优化原则

在当前，优化越来越重要，一个是因为现代处理器越来越复杂，有更多的机制来提供优化的机会。

另一个原因是多核并行越来越普遍，编译器需要适应，但是在硬件条件多变的情况下，很难知道一个解是不是最优解，那么就遵循以下的原则：

- 优化必须保证正确
- 优化要保证大部分程序都能提升表现
- 优化带来的编译时间消耗需要是可以接受的
- 工程量要能接受

5.7 代码生成

5.8 LLVM

5.8.1 简介

LLVM(Low Level Virtual Machine) 项目是模块化和可重用的编译器和工具链技术的集合。虽然它的名称中 Virtual Machine，但是 LLVM 与传统的虚拟机无关。

LLVM 是 Illinois 大学发起的一个开源项目，和 JVM 以及.net Runtime 这样的虚拟机不同，LLVM 提供了一套中立的中间代码和编译基础设施，并围绕这些设施提供了一套全新的编译策略（使得优化能够在编译、连接、运行环境执行过程中，以及安装之后以有效的方式进行）和其他一些非常有意思的功能。

LLVM 可以提供完整的编译器系统的中间层，从编译器获取中间表示（IR）代码并发出优化的 IR。然后可以将此新的 IR 转换并链接到目标平台的依赖于机器的汇编语言代码。LLVM 可以接受来自 GNU 编译器集合（GCC）工具链的 IR，允许它与为该项目编写的大量现存编译器一起使用。

LLVM 还可以在运行时在编译时或链接时甚至二进制机器代码生成可重定位的机器代码。

LLVM 支持独立于语言的指令集和类型系统。每个指令都是静态单一分配形式（SSA），这意味着每个变量（称为类型化寄存器）被分配一次然后被冻结。这有助于简化变量之间依赖关系的分析。LLVM 允许代码按照传统的 GCC 系统进行静态编译，也可以通过与 Java 类似的即时编译从 IR 到机器代码进行后期编译。类型系统由基本类型（如整数或浮点数）和五个派生类型组成：指针，数组，向量，结构和函数。可以通过在 LLVM 中组合这些基本类型来表示具体语言的类型构造。例如，C++ 中的类可以通过结构，函数和函数指针数组的混合来表示。

LLVM JIT 编译器可以在运行时从程序中优化不需要的静态分支，因此在程序具有许多选项的情况下对部分评估非常有用，其中大部分可以在特定环境中轻松确定。

OpenGL 堆栈中的图形代码可以保留在中间表示中，然后在目标机器上运行时进行编译。在具有高端图形处理单元（GPU）的系统上，由此产生的代码仍然相当薄弱，将指令传递给 GPU 并进行最小的更改。在具有低端 GPU 的系统上，LLVM 将编译在本地中央处理单元（CPU）上运行的可选过程，以模拟 GPU 无法在内部运行的指令。LLVM 在使用英特尔 GMA 芯片组的低端机器上提高了性能。在 Gallium3D LLVM 管道下开发了一个类似的系统，并将其并入 GNOME 外壳，以便在没有正确加载的 3D 硬件驱动程序的情况下运行它。

5.8.2 组件

前端

LLVM 最初是为了替代 GCC 堆栈中的现有代码生成器，并且许多 GCC 前端已经修改以使用它。LLVM 目前支持使用各种前端编译 Ada，C，C++，D，Delphi，Fortran，Haskell，Objective-C 和 Swift。

对 LLVM 的广泛兴趣导致了为各种语言开发新前端的一些努力。受到最多关注的是 Clang，一个支持 C，C++ 和 Objective-C 的新编译器。主要由苹果公司支持，Clang 的目标是在 GCC 系统中更换 C / Objective-C 编译器，该系统可以更容易地与集成开发环境（IDE）集成，并且对多线程有更广泛的支持。自 2008 年第 3.8 版起，支持 OpenMP 指令已被包括在 Clang 中。

Utrecht Haskell 编译器可以为 LLVM 生成代码。虽然生成器处于开发的早期阶段，但在许多情况下，它比 C 代码生成器更有效。Glasgow Haskell Compiler (GHC) 具有工作的 LLVM 后端，可以通过 GHC 或 C 代码生成紧跟编译，实现编译代码相对于本地代码编译 30％的加速。

许多其他组件处于不同的开发阶段，包括但不限于 Rust 编译器，Java 字节码前端，通用中间语言（CIL）前端，Ruby 1.9 的 MacRuby 实现，标准 ML 的各种前端，和一个新的图形着色寄存器分配器。

IR

LLVM 的核心是 IR，一种类似于汇编的低级编程语言。IR 是一种 RISC 指令集，它可以摘录目标的细节。例如，调用约定通过带有明确参数的调用和 ret 指令进行抽象。而且，IR 代替一组固定的寄存器，它使用了一个无限大小的形式为%0，%1 等等。LLVM 支持三种同构（即功能上等同的）形式的 IR：一种人类可读的汇编格式，适用于前端的 C++ 对象格式，以及用于序列化的密码比特码格式。一个简单的 Hello World 程序中的汇编格式如下：

```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

后端

在版本 3.4 中，LLVM 支持许多指令集，包括 ARM，Qualcomm Hexagon，MIPS，Nvidia 并行线程执行（PTX;在 LLVM 文档中称为 NVPTX），PowerPC，AMD TeraScale，AMD Graphics Core Next（GCN），SPARC，z/Architecture（在 LLVM 文档中称为 SystemZ），x86，x86-64 和 XCore。某些功能在某些平台上不可用。大多数功能都适用于 x86，x86-64，z/Architecture，ARM 和 PowerPC。

LLVM 机器代码（MC）子项目是用于在文本形式和机器代码之间翻译机器指令的 LLVM 框架。以前，LLVM 依赖于系统汇编程序，或者是由工具链提供的汇编程序，将程序集翻译成机器代码。LLVM MC 的集成汇编器支持大多数 LLVM 目标，包括 x86，x86-64，ARM 和 ARM64。对于某些目标，包括各种 MIPS 指令集，集成程序集支持可用，但仍处于 beta 阶段。

5.8.3 Ref

- llvm org
- wiki

5.9 JIT

5.10 相关术语

5.10.1 符号与字母

- **字母表**
  - 元素的非空有穷集合
- **符号**
  - 字母表中的元素
- **符号串**
  - 符号的有穷序列

5.10.2 文法的形式定义

- **规则**
  - 也称产生式，是符号和符号串的有序对
- **符号分为终结符，非终结符**
  - 非终结符，出现在规则左部，能派生出符号（串）的符号
  - 终结符，语言的基本符号，不可再分
- **文法**
  - 规则的非空有穷集合，通常表示为四元组的形式。包含非终结符的集合、终结符的集合、文法规则的集合、非终结符号

CHAPTER 6

操作系统

6.1 概述

6.1.1 什么是操作系统

通常把操作系统定义为用以控制和管理计算机系统资源，方便用户使用的程序和数据结构的集合。

在所有的系统软件中，操作系统是一种首要的、最基本、最重要的系统程序，也是最庞大、最复杂的系统软件。

从系统的视角看计算机是系统的控制中心，是系统的大脑。CPU 是计算机硬件的核心，是计算机系统的心脏；操作系统则是计算机软件的核心，是计算机系统的大脑，从而操作系统是整个系统的控制中心，是计算机或智能控制管理系统中首要的、最重要的、最复杂的系统软件。

操作系统的作用类似于城市交通的决策、指挥、控制和调度中心，它组织和管理整个计算机系统的硬件和软件资源，在用户和程序之间分配系统资源，使之协调一致地、高效地完成各种复杂的任务。

从用户的视角看，用户几乎不可能使用裸机。如果在裸机之上覆盖一层 I/O 设备管理软件，就能使用户较方便地使用外部设备；如果在其上再覆盖一层文件管理软件，用户就很容易存取系统文件和用户文件；每覆盖一层新的软件，就构造了一台功能更强的虚拟机器。通过 OS，计算机能提供种类更多，质量更高的服务。

从软件的观点看，操作系统是直接与硬件相邻的第一层软件，它是由大量极其复杂的系统程序和众多的数据结构集成的。在计算机中的所有软件中，它起到了核心和控制的作用，其他软件的运行都要依赖它的支持。操作系统是在系统中永久运行的超级程序。

特征

- **并发（Concurrence）**
  - 在某一时间间隔内计算机系统内存在着多个程序活动（与并行的区别）。
- **共享（Sharing）**
  - 多个用户或程序共享系统的硬、软件资源。
  - 互斥共享和同时共享
- **虚拟**
  - 在原先的物理机器上覆盖了一至多层系统软件，将其改造成一台功能更强大而且易于使用的扩展机或虚拟机
  - 提供了高级的抽象服务。
- **不确定性**
  - 使用同样一个数据集的同一个程序在同样的计算机环境下运行，每次执行的顺序和所需的时间都不相同。

主要功能

- **处理机管理**
  - 进程调度、进程控制、进程同步与互斥、进程通信、死锁的检测与处理
- **存储管理**
  - 对要运行的作业分配内存空间
  - 作业运行结束时收回其所占用的内存空间
  - 保护每个作业的内存空间和系统内存空间
- **设备管理**
  - 设备的分配和回收，设备的控制和信息传输即设备驱动
  - 缓冲管理
  - 虚设备管理
- **文件管理**
  - 文件存储空间的分配和回收等
- 用户接口

6.1.2 内核分类

宏内核

宏内核，又称单核心，是操作系统核心架构的一种，此架构的特性是整个核心程序都是以核心空间（KernelSpace）的身份及监管者模式（Supervisor Mode）来运行。

传统 UNIX、BSD、Linux、MS-DOS、Windows 9x 都是宏内核。

宏内核效率高、反应快，但是耦合度高、灵活性差。

微内核

微内核的基本原理是只有最基本的操作系统功能才能放在内核中，非基本的服务和应用程序在微内核之上构造，并在用户态下运行。尽管什么应该在微内核中、什么应该在微内核外，不同的设计有不同的分界线，但是共同的特点是许多传统上属于操作系统一部分的功能现在都是在外部子系统现代操作系统，包括设备驱动程序、文件系统、虚存管理程序、窗口系统和安全服务。他们可以与内核交互，也可以相互交互。

第一代微内核主要是 Mach，第二代微内核是 QNX、L4，第三代微内核是 Windows NT、Mac OS X。

微内核主要有一下优势

- **一致接口**
  - 微内核设计为进程发出的请求提供一致接口，进程不需要区分是内核级服务还是用户级服务，因为所有服务都是通过消息传递提供的
- **可扩展性**
  - 微内核促进了可扩展性，允许增加新的服务以及在同一个功能区域中提供多个服务。因此，用户可以从各种服务现代操作系统中选取最适合的一种。当增加一个新功能时，只需要修改或添补选中的服务。新服务程序或修改过的服务程序的影响被限制在系统的一个子集中，而且修改不会导致需要构造一个新内核
- **灵活性**
  - 不仅可以在操作系统中增加新功能，还可以删减现有的功能，产生一个更小、更有效的实现
- **可移植性**
  - 在微内核结构中，所有或者至少大部分处理器的专用代码都在微内核中。因此，当把系统移植到一个新处理器上时只需要很少的变化，且易于进行逻辑上的归类。
- **可靠性**
  - 小的微内核可以被严格地测试，他使用少量的应用程序编程接口（API），这就为内核外部的操作系统服务产生高质量的代码提供了机会。系统程序员只掌握有限数量的 API 和有限数量的交互方式，因此不易影响到其他系统部件
- 微内核结构有助于提供分布式系统支持，包括分布式操作系统控制的集群
- 微内核结构也适用于面向对象操作系统环境

6.1.3 文件系统

文件

磁盘通常会提供最基础的读写操作，但是往往是以块为单位的。为了更好的提供数据检索、权限校验、判断块是否被占用等功能，提出了文件系统做为存储的抽象。

文件 (Files) 是由进程创建的逻辑信息单元。一个磁盘会包含几千甚至几百万个文件，每个文件是独立于其他文件的。

文件由操作系统进行管理，有关文件的构造、命名、访问、使用、保护、实现和管理方式都是操作系统设计的主要内容。从总体上看，操作系统中处理文件的部分称为文件系统 (file system)。

文件访问

早期的操作系统只有一种访问方式：序列访问 (sequential access)。在这些系统中，进程可以按照顺序读取所有的字节或文件中的记录，但是不能跳过并乱序执行它们。顺序访问文件是可以返回到起点的，需要时可以多次读取该文件。当存储介质是磁带而不是磁盘时，顺序访问文件很方便。

在使用磁盘来存储文件时，可以不按照顺序读取文件中的字节或者记录，或者按照关键字而不是位置来访问记录。这种能够以任意次序进行读取的称为随机访问文件 (random access file)。

6.1.4 参考链接

- 一文带你彻底理解文件系统

6.2 Boot

6.2.1 BIOS

BIOS (Basic Input/Output System) 是固化到计算机内主板上一个 ROM 芯片上的程序，它保存着计算机最重要的基本输入输出的程序、开机后自检程序和系统自启动程序，提供了一种载入操作系统的方式。

BIOS 的主要功能有开机自检、系统初始化、提供运行时服务、系统设置、引导操作系统等。

在电源启动后做自检，如果硬件有问题，则发出蜂鸣，没问题则根据设定好的启动顺序把控制权转交给下一阶段的启动程序。

转交给下阶段启动程序后，计算机会读取设备的第一个扇区，如果扇区的最后两个字节是 `0x55AA` 则表明设备可以用于启动，否则将控制权转移给下一个设备。

扇区共 512 字节，其中第 1-446 字节记录调用操作系统的机器码，第 447-510 字节记录分区表（Partition table），将硬盘分为多个分区，第 511-512 字节为主引导记录签名（0x55AA）。

6.2.2 UEFI

UEFI，全称是统一的可扩展固件接口 (Unified Extensible Firmware Interface)，是计算机操作系统和平台固件之间的接口规范，提供了灵活的配置。

在 UEFI 之后，BIOS 也被称作 Legacy。和之前的 BIOS 相比，UEFI 最大的区别在于编码几乎都是用 C 语言完成，只使用少量的汇编。运行使用 32 位的保护模式，不再使用 16 位的实模式。

BIOS 标准将 bootlloader 存放在主引导记录（MBR）中，容量有限，而 UEFI 中引入了新的系统分区来存储 bootloader 以及驱动等数据。

另外，UEFI 通过固件签名验证提供了安全的启动加载，拒绝不合法的 UEFI 执行程序和驱动程序。

6.2.3 GRUB

GNU GRUB（GRand Unified Bootloader）是一个来自 GNU 项目的多操作系统启动程序。GRUB 是多启动规范的实现，它允许用户可以在计算机内同时拥有多个操作系统，并在计算机启动时选择希望运行的操作系统。GRUB 可用于选择操作系统分区上的不同内核，也可用于向这些内核传递启动参数。

6.2.4 参考链接

- UEFI
- 计算机是如何启动的

6.3 Linux

6.3.1 简介

Linux 内核在 1991 年首次被 Linus 发布，采用 GPL(GNU General Public License) 许可证开源。

Linux 内核采用单内核结构，也借鉴类微内核的优点：模块、抢占式内核、支持内核线程以及动态装载内核模块。

Linux 的版本以 `kernel_version.major_revision.minor_revision` 的格式表示，其中偶数 `major_revision` 代表稳定版，奇数代表开发版。

Linux 的所有历史版本都可以在 这里 找到。

启动

接通电源后，计算机加载 BIOS 或 UEFI，进行相应的启动程序。

而后 BIOS/UEFI 加载 MBR，MBR 调用 GRUB，而后调用 Linux 内核，调用内核文件，之后执行 `init` 作为所有系统进程的起点。`init` 读取配置文件 `/etc/inittab` ，并根据 `inittab` 执行相应程序。

其中常见的内核文件有：

- **vmlinux**
  - 编译出来的最原始的内核文件，未压缩。
- **zImage**
  - vmlinux 经过 gzip 压缩后的文件。
  - zImage 解压缩内核到低端内存 (第一个 640K)
- **bzImage**
  - bz 表示 `big zImage`
  - bzImage 解压缩内核到高端内存 (1M 以上)
- **uImage**
  - U-boot 专用的映像文件，在 zImage 之前加上一个长度为 0x40 的 tag
- **vmlinuz**
  - bzImage/zImage 文件的拷贝或指向 bzImage/zImage 的链接。

- **initrd**
  - `initial ramdisk` 的简写
  - 一般被用来临时的引导硬件到实际内核 vmlinuz 能够接管并继续引导的状态

然后根据不同的运行级别执行 `/etc/rcX.d/` 目录下的文件，例如运行级别为 5 时，执行 `/etc/rc5.d/` 下的文件，接着执行 `/etc/rc.d/rc.local`，最后设置好 tty，为用户登录做准备。

内核功能划分

- 进程管理：进程管理负责创建和销毁进程，不同进程之间的通信也有内核处理。
- 内存管理：内存是计算机的主要资源之一。
- 文件系统：类 Unix 系统的对象几乎都可以当作文件处理。
- 设备控制：几乎所有系统操作都会映射到物理设备上，需要通过相应的代码完成，这段代码被称为驱动程序。
- 网络功能：操作系统通过在应用程序和网络接口之间传递数据包，并根据网络活动控制程序执行。

6.3.2 源代码

目录结构

通常来说，Linux 的目录结构如下:

- **/bin**
  - 必要的一些二进制文件
- **/boot**
  - boot loader 的静态文件
- **/etc**
  - 各种 config 文件
  - /etc/fstab 开机自动挂载的配置文件
  - /etc/mtab 当前的分区挂载情况
  - /etc/passwd 用户文件
  - /etc/shadow 密码文件
- **/usr**
  - 共享的一些只读文件
  - **/usr/local**
    - 本地文件
  - **/usr/share**
    - 所有架构的静态共享文件
  - **/usr/share/man**
    - 手册文件
    - man1 user program
    - man2 system calls
    - man3 lib functions
    - man4 special file
    - man5 file formats

        * man6 games
        * man7 misc
        * man8 system admin
    - **/usr/bin**
        * 常用的用户命令
    - **/usr/include**
        * C 程序标准库
    - **/usr/lib**
        * 程序 obj / bin / lib 文件
    - **/usr/sbin**
        * 非必须文件
- **/var**
    - 各种变量数据文件
    - **/var/cache**
        * 应用 cache
    - /var/lib
    - **/var/yp**
        * NIS 服务
    - **/var/lock**
        * 共享文件的锁
    - **/var/opt**
        * 安装包的数据
    - **/var/run**
        * 系统启动后的数据
    - /var/tmp
    - **/var/spool**
        * 待处理数据
    - **/var/log**
        * 日志
- **/sbin**
    - 系统程序
- **/tmp**
    - 临时文件，重启后删除
- **/dev**
    - /dev/hd[a-t]：IDE 设备
    - /dev/sd[a-z]：SCSI 设备
    - /dev/fd[0-7]：标准软驱
    - /dev/md[0-31]：软 raid 设备
    - /dev/loop[0-7]：本地回环设备
    - /dev/ram[0-15]：内存

  - /dev/null：无限数据接收设备, 相当于黑洞
  - /dev/zero：无限零资源
  - /dev/tty[0-63]：虚拟终端
  - /dev/ttyS[0-3]：串口
  - /dev/lp[0-3]：并口
  - /dev/console：控制台
  - /dev/fb[0-31]：framebuffer
  - /dev/random：随机数设备
  - /dev/urandom：随机数设备
  - /dev/cdrom => /dev/hdc
  - /dev/modem => /dev/ttyS[0-9]
  - /dev/pilot => /dev/ttyS[0-9]
- **/proc**
  - 伪文件系统，它只存在内存当中，而不占用外存空间
  - 以文件系统的方式为访问系统内核数据的操作提供接口
  - 可以通过 proc 得到系统的信息，并可以改变内核的某些参数
  - **/proc/cmdline**
    * 内核命令的启动行
  - **/proc/cpuinfo**
    * 系统 CPU 的多种信息
  - **/proc/crypto**
    * 系统上已安装的内核使用的密码算法及每个算法的详细信息列表
  - **/proc/devices**
    * 字符设备和块设备的主设备号
  - **/proc/dma**
    * DMA 通道
  - **/proc/filesystems**
    * 可供使用的文件系统类型
  - **/proc/interrupts**
    * 保留的中断
  - **/proc/iomem**
    * 物理设备在系统内存中的映射信息
  - **/proc/ioports**
    * 设备驱动程序登记的 I/O 端口范围
  - **/proc/kcore**
    * 以 core 格式保存的系统物理内存

- **/proc/kmsg**
  - 内核消息
- **/proc/ksyms**
  - 内核符号
- **/proc/loadavg**
  - 负载信息
- **/proc/locks**
  - 文件的加锁信息
- **/proc/meminfo**
  - 内存状态信息
- **/proc/misc**
  - 通过 `misc_register` 登记的设备驱动信息
- **/proc/modules**
  - 可加载内核模块的信息
- **/proc/mounts**
  - 以 `/etc/mtab` 文件的格式给出当前系统所安装的文件系统信息
- /proc/mtd
- **/proc/partitions**
  - 块设备每个分区的主设备号 (major) 和次设备号 (minor) 等信息
  - 同时包括每个分区所包含的块 (block) 数目
- **/proc/pci**
  - PCI 设备的信息
- **/proc/stat**
  - CPU 利用率，磁盘，内存页，内存对换，全部中断，接触开关
- **/proc/uptime**
  - 从上次系统自举以来的秒数
- **/proc/version**
  - 正在运行的内核版本
- **/proc/net**
  - 联网代码的行为
- **/proc/scsi**
  - SCSI 设备的文件
- **/proc/sys**
  - 系统信息

- **/home** – 用户文件
- **/lib** – 库和内核模块
- **/mnt** – 挂载目录
- **/opt** – 应用程序
- **/root** – root 用户

6.3.3 文件系统

基础知识

简介

Linux 以一组通用对象的角度看待所有文件系统。这些对象是超级块（superblock）、inode、dentry 和文件。超级块在每个文件系统的根上，描述和维护文件系统的状态。文件系统中管理的每个对象（文件或目录）在 Linux 中表示为一个 inode。inode 包含管理文件系统中的对象所需的所有元数据（包括可以在对象上执行的操作）。另一组结构称为 dentry，它们用来实现名称和 inode 之间的映射，有一个目录缓存用来保存最近使用的 dentry。dentry 还维护目录和文件之间的关系，从而支持在文件系统中移动。最后，VFS 文件表示一个打开的文件（保存打开的文件的状态，比如写偏移量等等）。

超级块

超级块结构表示一个文件系统。它包含管理文件系统所需的信息，包括文件系统名称（比如 ext2）、文件系统的大小和状态、块设备的引用和元数据信息（比如空闲列表等等）。超级块通常存储在存储媒体上，但是如果超级块不存在，也可以实时创建。

block

存储数据。

inode

inode 也称索引节点，表示文件系统中的一个对象，它具有惟一标识符。各个文件系统提供将文件名映射为惟一 inode 标识符和 inode 引用的方法。可用通过 *ls -id* 查看。

Overlayfs

简介

OverlayFS 是一种堆叠文件系统，它依赖并建立在其它的文件系统之上，并不直接参与磁盘空间结构的划分，仅仅将原来底层文件系统中不同的目录进行“合并”，然后向用户呈现。

它于 2014 年被合并到 Linux 内核的 3.18 版本。其 4.0 版本带来了必要的改进，例如 Docker 中所需的 overlay2 存储驱动程序。

机制

OverlayFS 的主要机制涉及当两个文件系统提供同一名称的目录时目录访问的合并，分为 upper 层（读写层）和 lower（只读层）。OverlayFS 下的文件并不是 upper 层或者 lower 层文件的拷贝，而是记录了文件的层次信息。

在 OverlayFS 中，会为每个文件分配一个 ovl_entry 变量，通过文件 dentry 的 d_fsdata 字段指向这个分配的 ovl_entry 变量。

UnionFS

简介

联合文件系统 (Union File System) 在 2004 年由纽约州立大学石溪分校开发，它可以把多个目录内容联合挂载到同一个目录下，而目录的物理位置是分开的。

JFFS

简介

JFFS 意为 “Journaling Flash File System”，该文件系统是瑞典 Axis 通信公司开发的一种基于 Flash 存储器的日志文件系统。

该公司于 1999 年在 GNU/Linux 上发行了第一版 JFFS 文件系统，后来经过 Redhat 公司的发展，发行了第二个版本的 JFFS2。JFFS2 是一个日志结构 (log-structured) 的文件系统，将文件系统的数据和原数据以节点的形式存储在闪存上。

主要用于 NOR 型闪存，基于 MTD 驱动层，特点是：可读写的、支持数据压缩的、基于哈希表的日志型文件系统，并提供了崩溃/掉电安全保护，提供 “写平衡” 支持等。缺点主要是当文件系统已满或接近满时，因为垃圾收集的关系而使 JFFS2 的运行速度大大放慢。

JFFS2 的缺点包括：挂载时间过长；对芯片存储块读写不均衡；扩展性较差等。JFFS2 不适合用于 NAND 闪存主要是因为 NAND 闪存的容量一般较大，这样导致 JFFS2 为维护日志节点所占用的内存空间迅速增大，另外，JFFS2 文件系统在挂载时需要扫描整个 FLASH 的内容，以找出所有的日志节点，建立文件结构，对于大容量的 NAND 闪存会耗费大量时间。

YAFFS

简介

YAFFS/YAFFS2 是专为嵌入式系统使用 NAND 型闪存而设计的一种日志型文件系统。与 JFFS2 相比，它减少了一些功能 (例如不支持数据压缩)，所以速度更快，挂载时间很短，对内存的占用较小。另外，它还是跨平台的文件系统。

YAFFS/YAFFS2 自带 NAND 芯片的驱动，并且为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 与 VFS，直接对文件系统操作。当然，YAFFS 也可与 MTD 驱动程序配合使用。这方便了其跨平台移植。

YAFFS 与 YAFFS2 的主要区别在于，前者仅支持小页 (512 Bytes)NAND 闪存，后者则可支持大页 (2KB) NAND 闪存。同时，YAFFS2 在内存空间占用、垃圾回收速度、读/写速度等方面均有大幅提升。

6.3.4 设备

设备分类

字符设备

Linux 字符设备是一种按字节来访问的设备，字符驱动则负责驱动字符设备，这样的驱动通常实现 open、close、read 和 write 系统调用。例如：串口、Led、按键等。

块设备

块设备是一种具有一定结构的随机存取设备，对这种设备的读写是按块进行的，他使用缓冲区来存放暂时的数据，待条件成熟后，从缓存一次性写入设备或者从设备一次性读到缓冲区。可以随机访问，块设备的访问位置必须能够在介质的不同区间前后移动。

网络接口

通常是个硬件设备，但也可能是个软设备，例如回环 (loopback) 接口。

驱动

错误码

- 1-34 错误码在 `asm-generic/errno-base.h` 中定义
- 35-133 错误码在 `asm-generic/errno.h` 中定义

Linux 字符设备驱动 file_operations

file_operations 在 `/include/linux/fs.h` 中被定义，其结构如下：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iopoll)(struct kiocb *kiocb, bool spin);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    __poll_t (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
```

(下页继续)

(续上页)

```
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,␣
↪unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
↪ unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,␣
↪unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
            loff_t, size_t, unsigned int);
    loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                   struct file *file_out, loff_t pos_out,
                   loff_t len, unsigned int remap_flags);
    int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

owner

owner 不是一个操作，它是一个指向拥有这个结构的模块的指针，这个成员用来阻止模块还在被使用时被重复卸载。在大部分场景下它被简单初始化为 THIS_MODULE, 一个在 <linux/module.h> 中定义的宏。

llseek

llseek 指针参数 filp 为进行读取信息的目标文件结构体指针；参数 p 为文件定位的目标偏移量；参数 orig 为对文件定位的起始地址，这个值可以为文件开头 (SEEK_SET,0)，当前位置 (SEEK_CUR,1)，文件末尾 (SEEK_END,2)。

llseek 方法用作改变文件中的当前读/写位置, 并且新位置作为 (正的) 返回值。

loff_t 参数是一个"long offset"，并且就算在 32 位平台上也至少 64 位宽。错误由一个负返回值指示；如果这个函数指针是 NULL，seek 调用会以潜在地无法预知的方式修改 file 结构中的位置计数器。

read

指针参数 filp 为进行读取信息的目标文件，指针参数 buffer 为对应放置信息的缓冲区（即用户空间内存地址），参数 size 为要读取的信息长度，参数 p 为读的位置相对于文件开头的偏移，在读取信息后，这个指针一般都会移动，移动的值为要读取信息的长度值。

这个函数用来从设备中获取数据。在这个位置的一个空指针导致 read 系统调用以 -EINVAL("Invalid argument") 失败。一个非负返回值代表了成功读取的字节数 (返回值是一个"signed size" 类型, 常常是目标平台本地的整数类型)。

aio_read

aio_read 函数的第一、三个参数和本结构体中的 read() 函数的第一、三个参数是不同的，异步读写的第三个参数直接传递值，而同步读写的第三个参数传递的是指针，因为 AIO 从来不需要改变文件的位置。异步读写的第一个参数为指向 kiocb 结构体的指针，而同步读写的第一参数为指向 file 结构体的指针，每一个 I/O 请求都对应一个 kiocb 结构体。

write

参数 filp 为目标文件结构体指针，buffer 为要写入文件的信息缓冲区，count 为要写入信息的长度，ppos 为当前的偏移位置，这个值通常是用来判断写文件是否越界。函数的返回值代表成功写的字节数。

aio_write

初始化设备上的一个异步写，参数类型同 aio_read() 函数。

readdir

对于设备文件这个成员应当为 NULL; 它用来读取目录, 并且仅对文件系统有意义。

poll

这是一个设备驱动中的轮询函数，第一个参数为 file 结构指针，第二个为轮询表指针。

这个函数返回设备资源的可获取状态，即 POLLIN、POLLOUT、POLLPRI、POLLERR、POLLNVAL 等宏的位“或”结果。每个宏都表明设备的一种状态，如 POLLIN（定义为 0x0001）意味着设备可以无阻塞的读，POLLOUT（定义为 0x0004）意味着设备可以无阻塞的写。

poll 方法是 poll、epoll 和 select 3 个系统调用的后端，都用作查询对一个或多个文件描述符的读或写是否会阻塞。

poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的，并且提供给内核信息用来使调用进程睡眠直到 I/O 变为可能。如果一个驱动的 poll 方法为 NULL, 设备假定为不阻塞地可读可写。

这里通常将设备看作一个文件进行相关的操作，而轮询操作的取值直接关系到设备的响应情况，可以是阻塞操作结果，同时也可以是非阻塞操作结果。

ioctl

inode 和 filp 指针是对应应用程序传递的文件描述符 fd 的值, 和传递给 open 方法的相同参数。cmd 参数从用户那里不改变地传下来, 并且可选的参数 arg 参数以一个 unsigned long 的形式传递，不管它是否由用户给定为一个整数或一个指针。如果调用程序不传递第 3 个参数，被驱动操作收到的 arg 值是无定义的。

因为类型检查在这个额外参数上被关闭，编译器不能警告你如果一个无效的参数被传递给 ioctl，并且任何关联的错误将难以查找。

ioctl 系统调用提供了发出设备特定命令的方法。另外, 几个 ioctl 命令被内核识别而不必引用 fops 表。如果设备不提供 ioctl 方法, 对于任何未事先定义的请求，系统调用返回一个错误。

mmap

mmap 用来请求将设备内存映射到进程的地址空间。

open

inode 为文件节点, 这个节点只有一个，无论用户打开多少个文件，都只是对应着一个 inode 结构；但是 filp 只要打开一个文件，就对应着一个 file 结构体，file 结构体通常用来追踪文件在运行时的状态信息

flush

flush 操作在进程关闭它的设备文件描述符的拷贝时调用，它执行并且等待设备的任何未完成的操作。

release

当最后一个打开设备的用户进程执行 close() 系统调用的时候，内核将调用驱动程序 release() 函数。release 函数的主要任务是清理未结束的输入输出操作，释放资源，用户自定义排他标志的复位等。在文件结构被释放时引用这个操作。如同 open，release 可以为 NULL。

synch

刷新待处理的数据, 允许进程把所有的脏缓冲区刷新到磁盘。

aio_fsync

aio_fsync 是 fsync 方法的异步版本。所谓的 fsync 方法是一个系统调用函数。系统调用 fsync 把文件所指定的文件的所有脏缓冲区写到磁盘中（如果需要，还包括存有索引节点的缓冲区）。相应的服务例程获得文件对象的地址，并随后调用 fsync 方法。通常这个方法以调用函数 __writeback_single_inode() 结束，这个函数把与被选中的索引节点相关的脏页和索引节点本身都写回磁盘。

fasync

fasync 函数是系统支持异步通知的设备驱动。

lock

lock 方法用来实现文件加锁。

readv / writev

readv / writev 实现发散/汇聚读和写操作。应用程序偶尔需要做一个包含多个内存区的单个读或写操作，这些系统调用允许它们这样做而不必对数据进行额外拷贝。如果参数中的函数指针为 NULL，read 和 write 方法被调用。

sendfile

sendfile 实现 sendfile 系统调用的读, 使用最少的拷贝从一个文件描述符搬移数据到另一个。

sendpage

sendpage 是 sendfile 的另一半; 它由内核调用来发送数据，一次一页，到对应的文件。

get_unmapped_area

这个方法的目的是在进程的地址空间找一个合适的位置来映射在底层设备上的内存段中。这个任务通常由内存管理代码进行，这个方法存在为了使驱动能强制特殊设备可能有的任何的对齐请求。大部分驱动可以置这个方法为 NULL。

check_flags

check_flags 用于模块检查传递给 fnctl(F_SETFL...) 调用的标志。

dir_notify

dir_notify 在应用程序使用 fcntl 来请求目录改变通知时调用。只对文件系统有用，驱动不需要实现 dir_notify。

MTD

简介

内存技术设备 (Memory Technology Device, MTD) 是 Linux 的存储设备中的一个子系统。设计此系统的目的是，对于内存类的设备提供一个抽象层的接口。

MTD 设备通常可分为四层，依次为：设备节点、MTD 设备层、MTD 原始设备层和硬件驱动层。

设备节点通过 mknod 在/dev 子目录下建立 MTD 块设备节点（主设备号为 31）和 MTD 字符设备节点（主设备号为 90）。通过访问此设备节点即可访问 MTD 字符设备和块设备。MTD 设备层定义出 MTD 的块设备（主设备号 31）和字符设备（设备号 90）。原始设备层定义了大量的关于 MTD 的数据和操作函数。硬件驱动层负责对硬件的读、写和擦除操作。

源码

Linux 中 MTD 的所有源码位于 `/drivers/mtd` 子目录下。

其中目录如下:

- drivers/mtd/chips: CFI/jedec 接口通用驱动
- drivers/mtd/nand: nand 通用驱动和部分底层驱动程序
- drivers/mtd/maps: nor flash 映射关系相关函数
- drivers/mtd/devices: nor flash 底层驱动

主要文件为:

- mtdcore.c: MTD 原始设备接口相关实现
- mtdpart.c: MTD 分区接口相关实现
- mtdchar.c: MTD 字符设备接口相关实现，设备号 31
- mtdblock.c: MTD 块设备接口相关实现，设备号 90

dev

/dev/mtdN 是 Linux 中的 MTD 架构中，系统自己实现的 mtd 分区所对应的字符设备，其里面添加了一些 ioctl，支持很多命令，如 MEMGETINFO，MEMERASE 等。mtdN 系列字符设备的主设备号为 90，次设备号为 0、2、4、6。

`/dev/mtdN` 和 `/dev/mtdblockN` 是同一个 MTD 设备的同一个分区，mtdblockN 块设备的主设备号为 31，次设备号为 0、1、2、3。

其他

设备号

Linux 系统中一个设备号由主设备号和次设备号构成，Linux 内核用主设备号来定位对应的设备驱动程序（即是主设备找驱动），而次设备号用来标识它同个驱动所管理的若干的设备（次设备号找设备）。设备号可以通过 `/proc/devices` 查看。

TTY/PTS

在计算机早期，人们将一种被称作 teletype 的设备连到计算机上，作为计算机的终端，使得人们可以操作计算机。计算机为了支持这些 teletype，于是设计了名为 TTY 的子系统。而后随着计算机技术的不断发展，teletype 这些设备逐渐消失，但是内核 TTY 驱动这一架构没有发生变化，和系统中的进程进行 I/O 交互时还是需要通过 TTY 设备，于是出现了各种终端模拟软件。tty 理解成一个管道（pipe），将输入输出连接起来。

对用户空间的程序来说，TTY 与 PTS 是透明的，但是从内核的视角看，pts 的另一端连接 ptmx，tty 的另一端连接内核的终端模拟器，ptmx 和终端模拟器都只是负责维护会话和转发数据包。ptmx 的另一端连接用户空间的应用程序，如 sshd、tmux 等，而内核终端模拟器的另一端连接的是具体的硬件，如键盘和显示器。

6.3.5 内存管理

地址定位方式

- **固定定位方式**
  - 汇编和机器代码一一对应
  - 容易覆盖操作系统
  - 只支持单道程序
- **静态重定位方式**
  - 源程序经编译和连接后生产的目标代码的地址是以 0 为起始地址的相对地址
  - 当需要执行时，由装入程序运行重定位程序模块，根据作业在本次分配到的内存起始地址将可执行代码装如指定内存地址，并修改有关地址部分的值
  - 修改的方式是对每一个逻辑地址的值加上内存区首地址
  - 需要整块的地址
- **动态重定位**
  - 在装入内存的时，不修改逻辑地址，访问逻辑地址时，动态的讲逻辑地址变成物理地址
  - 在执行指令时，若设计到逻辑地址，则将该地址送入虚地址寄存器 VR，再将 BR 和 VR 相加送入地址寄存器 MR

内存管理

内存分配

- 首次适应法
- 循环首次适应法
- 最佳适应算法
- 最差适应算法
- 多重分区

内存回收

在运行时需要回收一些很少使用的内存页面来保证系统持续有内存使用。页面回收的方式有页回写、页交换和页丢弃三种方式。

页回写指一个很少使用的页的后备存储器是一个块设备（例如文件映射），则可以将内存直接同步到块设备，腾出的页面可以被重用。

页交换指页面没有后备存储器，则可以交换到特定 swap 分区，再次被访问时再交换回内存。

页丢弃指页面的后备存储器是一个文件，但文件内容在内存不能被修改（例如可执行文件），那么在当前不需要的情况下可直接丢弃。

回收时间一般在内存紧缺时回收，或者周期性回收，或者由用户触发回收。

内存扩充

覆盖

在单用户系统中，为了能在较小的内存空间中运行大作业，可采用“覆盖”技术。覆盖技术就是将一个大程序按程序的逻辑结构划分成若干个程序（或数据）段，并将不会同时执行、从而就不必同时装入内存的程序段分在一组内，该组称为覆盖段。这个覆盖段可分配到同一个称为覆盖区的存储区域。

交换技术

覆盖技术用于一个作业的内部，交换技术用于不同的作业。早期在单一连续分配的存储区管理系统中，采用交换技术也可以实现多道程序设计。任一时刻，主存中只保留一个完整的用户作业。当该作业的时间片用完或因等待某一事件而不能继续运行时，系统就挑选下一个作业进入主存运行。

分页存储

页式存储管理的基本思想是把作业的虚地址空间划分成若干长度相等的页（page），也称虚页，每一个作业的虚页都从 0 开始编号。主存也划分成若干与虚页长度相等的页架（frame），也称页框或实页，主存的页架也从 0 开始编号。程序装入时，每一个虚页装到主存中的一个页架中，这些页架可以是不连续的。

页淘汰

- 最优淘汰算法（OPT）
- 先进先出淘汰算法（FIFO）
- 最近最少使用淘汰算法（LRU）
- 最近未使用淘汰算法（NUR）

虚拟地址转换

地址转换负责将虚拟地址转换成物理地址，有多个应用场景。

应用在虚拟内存中，可以给应用程序一种独占整个计算机内存的假象，可以使用超过实际物理大小的内存，应用程序之间互不干扰。

用于进程隔离，可以构建沙盒技术，避免操作系统内核和应用程序受到病毒或者恶意代码的攻击。

进程间通信时，地址转换可以将不同进程空间的地址映射到同一段物理内存，从而实现进程间通信。

加载动态链接库时，可以在多个程序实例之间进行共享。

6.3.6 ELF

加载

从编译/链接和运行的角度看，应用程序和库程序的链接有两种方式。一种是固定的、静态的链接，在编译时将要用到的库函数的代码从代码库中抽取出来，链接进应用软件中。另一种是动态链接，在编译阶段并不完成跟库函数的链接，到程序时才把链接库的映像装入用户空间并加以定位。

其中 ELF 的载入在 Linux 内核中完成，动态链接的实现在用户空间中由 `ld-linux.so` 来完成，解释器的启动由内核负责。

execve

在用户层面，shell 进行会调用 `fork()` 系统调用创建一个新进程，新进程调用 `execve()` 系统调用执行指定的 ELF 文件。

在内核中，`execve` 系统调用的相应入口是 `sys_execve` ，在执行 `sys_execve` 之后，内核会调用 `do_execve` / `search_binary_handle` / `load_elf_binary` 等函数来完成加载。`do_execve` 相关代码如下：

```
int do_execve(struct filename *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
{
    return do_execve_common(filename, argv, envp);
}

static int do_execve_common(struct filename *filename,
                struct user_arg_ptr argv,
                struct user_arg_ptr envp)
{
    // 选择最小负载的 CPU，以执行新程序
    sched_exec();

    // 填充 linux_binprm 结构体
    retval = prepare_binprm(bprm);

    // 拷贝文件名、命令行参数、环境变量
    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    retval = copy_strings(bprm->envc, envp, bprm);
    retval = copy_strings(bprm->argc, argv, bprm);

    // 调用 search_binary_handler 扫描 formats 链表，根据不同的文本格式，选择不同的 load 函数
    retval = exec_binprm(bprm);
}
```

其中结构体 `linux_binprm` 部分定义是：

```
struct linux_binprm {
    char buf[BINPRM_BUF_SIZE];
#ifdef CONFIG_MMU
    struct vm_area_struct *vma;
    unsigned long vma_pages;
#else
    ...
    unsigned interp_flags;
```

(下页继续)

(续上页)

```
    unsigned interp_data;
    unsigned long loader, exec;
};
```

注册机制

Linux 支持不同格式的可执行程序，这些程序用 linux_binfmt 来描述，其定义在 include/linux/binfmts.h 中。

```
/*
 * This structure defines the functions that are used to load the binary formats that
↪linux accepts.
 */
struct linux_binfmt {
    struct list_head lh;
    struct module *module;
    int (*load_binary)(struct linux_binprm *);
    int (*load_shlib)(struct file *);
    int (*core_dump)(struct coredump_params *cprm);
    unsigned long min_coredump; /* minimal dump size */
} __randomize_layout;
```

所有的 linux_binfmt 对象都处于一个链表中，第一个元素的地址存放在 formats 变量中，可以通过调用 register_binfmt() 和 unregister_binfmt() 函数在链表中插入和删除元素。在系统启动期间，为每个编译进内核的可执行格式都执行 register_binfmt() 函数。

当执行程序的时候，内核打开目标映像文件，并从目标文件的头部读入若干字节，并调用 search_binary_handler 遍历所有注册的 linux_binfmt 对象，对其调用 load_binary 方法来尝试加载，直到加载成功为止。

search_binary_handler 的部分代码如下：

```
int search_binary_handler(struct linux_binprm *bprm)
{
    // 遍历 formats 链表
    list_for_each_entry(fmt, &formats, lh) {
        if (!try_module_get(fmt->module))
            continue;
        read_unlock(&binfmt_lock);
        bprm->recursion_depth++;

        // 应用每种格式的 load_binary 方法
        retval = fmt->load_binary(bprm);
        read_lock(&binfmt_lock);
        put_binfmt(fmt);
        bprm->recursion_depth--;
        // ...
    }
    return retval;
}
```

Load ELF

在 ELF 文件格式中，处理函数是 `load_elf_binary` 函数，流程如下：

- **填充并且检查目标程序 ELF 头部**
  - 是否 `\x7fELF` 开头
  - 映像的类型是否为 `ET_EXEC`
- **`load_elf_phdrs` 加载目标程序的程序头表**
  - 执行程序必须至少有一个段
  - 所有段大小之和不能超过 64k
- **如果需要动态链接，则寻找和处理解释器段**
  - “解释器”段的类型为 PT_INTERP，可通过 `readelf -l` 查看
  - “解释器”段实际上是一个字符串，即解释器的文件位置
  - 通常为 `/lib/ld-linux.so.2` / `/lib64/ld-linux-x86-64.so.2`
- 检查并读取解释器的程序表头
- 装入目标程序的段 segment
- **填写程序的入口地址**
  - **如果需要装入解释器**
    - 通过 `load_elf_interp` 装入映像
    - 设置用户空间的入口地址为 `load_elf_interp()` 的返回值
    - `load_elf_interp()` 的返回值为解释器映像的入口地址
  - **如果不需要装入解释器**
    - 入口地址设置为目标映像本身的入口地址
- **`create_elf_tables` 填写目标文件的参数环境变量等必要信息**
  - 准备 `argc envc` 等变量
- `start_thread` 宏修改 eip / esp，准备进入新的程序入口

`load_elf_binary` 的部分代码如下：

```
static int load_elf_binary(struct linux_binprm *bprm)
{
    ...

    /* Now we do a little grungy work by mmapping the ELF image into
       the correct location in memory. */
    for(i = 0, elf_ppnt = elf_phdata;
        i < loc->elf_ex.e_phnum; i++, elf_ppnt++) {
        int elf_prot = 0, elf_flags, elf_fixed = MAP_FIXED_NOREPLACE;
        unsigned long k, vaddr;
        unsigned long total_size = 0;

        if (elf_ppnt->p_type != PT_LOAD)
            continue;
         ...
}
```

6.3.7 进程

概念

进程是程序处于一个执行环境中在一个数据集上的运行过程，它是系统进行资源分配和调度的一个可并发执行的独立单位。

状态

进程一般存在 7 种基础状态：D-不可中断睡眠、R-可执行 (TASK_RUNNING)、S-可中断睡眠、T-暂停态、t-跟踪态 (TASK_TRACED)、X-死亡态、Z-僵尸态。

不可中断睡眠态。位于这种状态的进程处于睡眠中，并且不允许被其他进程或中断) 打断。因此这种状态的进程，是无法使用 kill 杀死的。这种状态一般由 I/O 等待 (比如磁盘 I/O、网络 I/O、外设 I/O 等) 引起，出现时间非常短暂。

可执行态。这种状态的进程都位于 CPU 的可执行队列中，正在运行或者正在等待运行。

可中断睡眠态。这种状态的进程虽然也处于睡眠中，但是是允许被中断的。这种进程一般在等待某事件的发生，而被挂起。系统中大部分进程都处于 S 态。SLEEP 态进程不会占用任何 CPU 资源。

暂停与跟踪态。这种两种状态的进程都处于运行停止的状态，暂停态进程会释放所有占用资源。

僵尸态的进程实际上已经结束了，但是父进程还没有回收它的资源 (比如进程的描述符、PID 等)。僵尸态进程会释放除进程入口之外的所有资源。

死亡态。进程的真正结束态，这种状态一般在正常系统中无法被捕获到。

进程创建

Linux 有几种进程创建方式，一种是通过 fork() 系统调用创建进程，一种是在用户级通过 pthread 库中的 pthread_create() 创建线程，一种在内核级通过 kthread_create() 创建。

作业调度算法

- 先来先服务 (FCFS) 有利于长作业而不利于短作业。
- 短作业优先 (SJF) 较短的作业平均等待时间，较大的系统吞吐率。
- 响应比高优先 (HRN) 求等待时间与执行时间两者时间之比。相对等待时间长优先。
- 优先级调度作业的紧急程度、资源要求、类别等。

组成

- 进程控制块 (PCB)
- 进程执行的程序 (code)
- 进程执行时所用的数据
- 进程执行时使用的工作区

进程调度

- 先来先服务 (FIFO) 调度算法
- 时间片轮转法
- 优先级调度算法
- 多级反馈队列调度算法
- 实时系统调度策略

进程间通信

- 消息通信
- 共享存储区
- 管道通信

6.3.8 网络

网桥

网桥 (Bridge) 也称桥接器，是连接两个局域网的存储转发设备，用它可以完成具有相同或相似体系结构网络系统的连接。

Linux 网络协议栈已经支持了网桥的功能，但需要进行相关的配置才可以进行正常的转发。而要配置 Linux 网桥功能，需要配置工具 bridge-utils，生成网桥配置的工具有 brctl、ip 等。

网桥可以理解为实现在内核中的二层交换机。

配置方法

```
ip link add br0 type bridge
ip link set br0 up
ip link set br0 down
```

tun/tap

tun/tap 设备支持很多的类 UNIX 平台。tun 表示虚拟的是点对点设备，tap 表示虚拟的是以太网设备。tap 位于网络 OSI 模型的二层，tun 位于网络的三层。

veth-pair

Virtual Ethernet Pair 简称 veth pair，是一对的虚拟设备接口，是一个成对的端口。所有从一端进入的数据包都将从另一端出来。

veth 可以通过 `ip link add eth1-br1 type veth peer name phy-br1` 命令创建。

6.3.9 Socket

Unix Socket

Unix Socket 可以使同一台操作系统上的两个或多个进程进行数据通信。Unix Socket 既可以使用字节流，又可以使用数据队列。

常见结构

sockaddr 是通用的套接字地址是 linux 网络通信的地址结构体的一种，此数据结构用做 bind、connect、recvfrom、sendto 等函数的参数，指明地址信息。

```
struct sockaddr {
    sa_family_t sa_family;  /* address family, AF_xxx    */
    char        sa_data[14];    /* 14 bytes of protocol address */
};
```

sockaddr_in 是 internet 环境下套接字的地址形式，和 sockaddr 长度一样，都是 16 个字节。二者是并列结构，指向 sockaddr_in 结构的指针也可以指向 sockaddr。

```
struct sockaddr_in {
  __kernel_sa_family_t  sin_family; /* Address family       */
  __be16        sin_port;   /* Port number          */
  struct in_addr    sin_addr;   /* Internet address     */

  /* Pad to size of `struct sockaddr'. */
  unsigned char     __pad[__SOCK_SIZE__ - sizeof(short int) -
            sizeof(unsigned short int) - sizeof(struct in_addr)];
};
```

sockaddr_un 是 unix 环境下套接字的地址形式，这种方式也称为本地套接字。

```
struct sockaddr_un {
    __kernel_sa_family_t sun_family; /* AF_UNIX */
    char sun_path[UNIX_PATH_MAX];   /* pathname */
};
```

常用函数

socket 调用常用的函数有：socket/bind/listen/connect/accept/read/write/close 等，其中部分函数的详细定义与分析见 syscall 部分。

listen

listen 函数的原型为 `int listen(int sockfd, int backlog);` 。其中 sockfd 即 socket 描述符，backlog: 为相应 socket 可以排队的最大连接个数，表示排队连接队列的长度。

服务器在调用 socket()、bind() 之后就会调用 listen() 来监听这个 socket。

connect

connect 函数的原型为 `int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);` 。其中 sockfd 即 socket 描述符，addr 是 socket 地址，addrlen 为地址长度。

read / write

常见的用于读写的函数有：

```
int read(int socket, char *buffer, size_t len);
int write(int socket, char *buffer, size_t len);

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct␣
↪sockaddr *dest_addr, socklen_t addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_
↪addr, socklen_t *addrlen);

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

close

close 的函数原型为 `int close(int fd);` ，用于关闭 socket 描述字。

6.3.10 安全机制

Cgroups

四个组件

Cgroups 包含 Cgroup、Subsystem、hierarchy、task 四个组件。

Cgroup

Cgroup 是对进程分组管理的一种机制，一个 Cgroup 包含一组进程，并可以在上面添加添加 Linux Subsystem 的各种参数配置，将一组进程和一组 Subsystem 的系统参数关联起来。

Subsystem

Subsystem 是一个资源调度控制器不同版本的 Kernel 所支持的有所偏差，可以通过 `cat /proc/cgroups` 查看

- **blkio**
  - 对块设备 (比如硬盘) 的 IO 进行访问限制
- **cpu**
  - 设置进程的 CPU 调度的策略，比如 CPU 时间片的分配
- **cpuacct**
  - 统计/生成 cgroup 中的任务占用 CPU 资源报告
- **cpuset**
  - 在多核机器上分配给任务 (task) 独立的 CPU 和内存节点 (内存仅使用于 NUMA 架构)
- **devices**
  - 控制 cgroup 中对设备的访问
- **freezer**
  - 挂起 (suspend) / 恢复 (resume) cgroup 中的进程
- **memory**
  - 用于控制 cgroup 中进程的占用以及生成内存占用报告
- **net_cls**
  - 使用等级识别符（classid）标记网络数据包，这让 Linux 流量控制器 tc (traffic controller) 可以识别来自特定 cgroup 的包并做限流或监控
- **net_prio**
  - 设置 cgroup 中进程产生的网络流量的优先级
- **hugetlb**
  - 限制使用的内存页数量
- **pids**
  - 限制任务的数量
- **net_cls**
  - 可以使不同 cgroups 下面的进程使用不同的 namespace。
  - 每个 subsystem 会关联到定义的 cgroup 上，并对这个 cgoup 中的进程做相应的限制和控制。

hierarchy

hierarchy 树形结构的 CGroup 层级，每个子 CGroup 节点会继承父 CGroup 节点的子系统配置，每个 Hierarchy 在初始化时会有默认的 CGroup(Root CGroup)。

Task

Task (任务) 在 cgroups 中，任务就是系统的一个进程。

四个组件的关系

- 系统在创建新的 hierarchy 之后，该系统的所有任务都会加入这个 hierarchy 的 cgroup，称之为 root cgroup，此 cgroup 在创建 hierarchy 自动创建，后面在该 hierarchy 中创建都是 cgroup 根节点的子节点
- 一个 subsystem 只能附加到一个 hierarchy 上面
- 一个 hierarchy 可以附加多个 subsystem
- 一个 task 可以是多个 cgroup 的成员，但这些 cgroup 必须在不同的 hierarchy
- 一个进程 fork 出子进程时，该子进程默认自动成为父进程所在的 cgroup 的成员，也可以根据情况将其移动到到不同的 cgroup 中

namespace

简介

namespace 是 Linux 内核用来隔离内核资源的方式。通过 namespace 可以让一些进程只能看到与自己相关的一部分资源，而另外一些进程也只能看到与它们自己相关的资源，这两拨进程根本就感觉不到对方的存在。具体的实现方式是把一个或多个进程的相关资源指定在同一个 namespace 中。

Linux 在很早的版本中就实现了部分的 namespace，比如内核 2.4 就实现了 mount namespace。大多数的 namespace 支持是在内核 2.6 中完成的，比如 IPC、Network、PID、和 UTS。还有个别的 namespace 比较特殊，比如 User，从内核 2.6 就开始实现了，但在内核 3.8 中才宣布完成。同时，随着 Linux 自身的发展以及容器技术持续发展带来的需求，也会有新的 namespace 被支持，比如在内核 4.6 中就添加了 Cgroup namespace。

漏洞利用缓解措施

kptr_restrict

在 Linux 内核漏洞利用中常常使用 `commit_creds` 和 `prepare_kernel_cred` 来完成提权，其地址可以从/proc/kallsyms 中读取。

因此 Linux 启用了 kptr_restrict，其值和对应的功能如下：

- 0: root 和普通用户都可以读取
- 1: root 用户有权限读取, 普通用户没有权限
- 2: 内核将符号地址打印为全 0, root 和普通用户都没有权限

该值可以通过 `sysctl kernel.kptr_restrict` 查看和修改

SMEP

SMEP（Supervisor Mode Execution Protection）是一种减缓内核利用的 cpu 策略，禁止内核态到用户态内存页的代码执行，每一页都有 smep 标识来标明是否允许 ring0 的代码执行。

Capabilities

简介

Capabilities 机制是在 Linux 2.2 之后引入的，用于将之前与超级用户 root(UID=0) 关联的特权细分为不同的功能组，Capabilites 作为线程的属性存在，每个功能组都可以独立启用和禁用。其本质上就是将内核调用分门别类，具有相似功能的内核调用被分到同一组中。

这样一来，权限检查的过程就变成了：在执行特权操作时，如果线程的有效身份不是 root，就去检查其是否具有该特权操作所对应的 capabilities，并以此为依据，决定是否可以执行特权操作。

Capabilities 可以在进程执行时赋予，也可以直接从父进程继承。

列表

- **CAP_AUDIT_CONTROL**
  - 启用和禁用内核审计
  - 改变审计过滤规则
  - 检索审计状态和过滤规则
- **CAP_AUDIT_READ**
  - 允许通过 multicast netlink 套接字读取审计日志
- **CAP_AUDIT_WRITE**
  - 将记录写入内核审计日志
- **CAP_BLOCK_SUSPEND**
  - 使用可以阻止系统挂起的特性
- **CAP_CHOWN**
  - 修改文件所有者的权限
- **CAP_DAC_OVERRIDE**
  - 忽略文件的 DAC 访问限制
- **CAP_DAC_READ_SEARCH**
  - 忽略文件读及目录搜索的 DAC 访问限制
- **CAP_FOWNER**
  - 忽略文件属主 ID 必须和进程用户 ID 相匹配的限制
- **CAP_FSETID**
  - 允许设置文件的 setuid 位
- **CAP_IPC_LOCK**
  - 允许锁定共享内存片段

- **CAP_IPC_OWNER**
  - 忽略 IPC 所有权检查
- **CAP_KILL**
  - 允许对不属于自己的进程发送信号
- **CAP_LEASE**
  - 允许修改文件锁的 FL_LEASE 标志
- **CAP_LINUX_IMMUTABLE**
  - 允许修改文件的 IMMUTABLE 和 APPEND 属性标志
- **CAP_MAC_ADMIN**
  - 允许 MAC 配置或状态更改
- **CAP_MAC_OVERRIDE**
  - 忽略文件的 DAC 访问限制
- **CAP_MKNOD**
  - 允许使用 mknod() 系统调用
- **CAP_NET_ADMIN**
  - 允许执行网络管理任务
- **CAP_NET_BIND_SERVICE**
  - 允许绑定到小于 1024 的端口
- **CAP_NET_BROADCAST**
  - 允许网络广播和多播访问
- **CAP_NET_RAW**
  - 允许使用原始套接字
- **CAP_SETGID**
  - 允许改变进程的 GID
- **CAP_SETFCAP**
  - 允许为文件设置任意的 capabilities
- CAP_SETPCAP
- **CAP_SETUID**
  - 允许改变进程的 UID
- **CAP_SYS_ADMIN**
  - 允许执行系统管理任务，如加载或卸载文件系统、设置磁盘配额等
- **CAP_SYS_BOOT**
  - 允许重新启动系统
- **CAP_SYS_CHROOT**
  - 允许使用 chroot() 系统调用
- **CAP_SYS_MODULE**

  - 允许插入和删除内核模块
- **CAP_SYS_NICE**
  - 允许提升优先级及设置其他进程的优先级
- **CAP_SYS_PACCT**
  - 允许执行进程的 BSD 式审计
- **CAP_SYS_PTRACE**
  - 允许跟踪任何进程
- **CAP_SYS_RAWIO**
  - 允许直接访问 /devport、/dev/mem、/dev/kmem 及原始块设备
- **CAP_SYS_RESOURCE**
  - 忽略资源限制
- **CAP_SYS_TIME**
  - 允许改变系统时钟
- **CAP_SYS_TTY_CONFIG**
  - 允许配置 TTY 设备
- **CAP_SYSLOG**
  - 允许使用 syslog() 系统调用
- **CAP_WAKE_ALARM**
  - 允许触发一些能唤醒系统的东西 (比如 CLOCK_BOOTTIME_ALARM 计时器)

线程的 capabilities

每一个线程都有 5 个 capabilities 集合，每一个集合使用 64 位掩码来表示。这 5 个 capabilities 集合分别是：

- Permitted
- Effective
- Inheritable
- Bounding
- Ambient

每个集合中都包含零个或多个 capabilities。这 5 个集合的具体含义如下：

Permitted

定义了线程能够使用的 capabilities 的上限。

Effective

内核检查 Effective 集合以判断线程是否可以进行特权操作时。

Inheritable

当执行 exec() 系统调用时，能够被新的可执行文件继承的 capabilities。

Bounding

Bounding 集合是 Inheritable 集合的超集，如果某个 capability 不在 Bounding 集合中，即使它在 Permitted 集合中，该线程也不能将该 capability 添加到它的 Inheritable 集合中。

Bounding 集合的 capabilities 在执行 fork() 系统调用时会传递给子进程的 Bounding 集合，并且在执行 execve 系统调用后保持不变。

当线程运行时，不能向 Bounding 集合中添加 capabilities。

一旦某个 capability 被从 Bounding 集合中删除，便不能再添加回来。

将某个 capability 从 Bounding 集合中删除后，如果之前 Inherited 集合包含该 capability，将继续保留。但如果后续从 Inheritable 集合中删除了该 capability，便不能再添加回来。

Ambient

Linux 4.3 内核新增的 capabilities 集合，用于弥补 Inheritable 的不足。Ambient 具有如下特性：

- Permitted 和 Inheritable 未设置的 capabilities，Ambient 也不能设置。
- 当 Permitted 和 Inheritable 关闭某权限 (比如 CAP_SYS_BOOT) 后，Ambient 也随之关闭对应权限。确保进程降低权限后子进程也会降低权限。
- 非特权用户如果在 Permitted 集合中有一个 capability，那么可以添加到 Ambient 集合中，这样它的子进程可以获取这个 capability。

6.3.11 syscall

基础

简介

syscall 是内核提供用户空间程序与内核空间进行交互的一套标准接口，这些接口让用户态程序能受限访问硬件设备，比如申请系统资源、操作设备读写、创建新进程等。用户空间发起请求，内核空间负责执行，这些接口便是用户空间和内核空间共同识别的桥梁。

Syscall 是通过中断方式实现的，用户程序通过软中断让程序陷入内核态，执行相应的操作。Linux 内核中，每个 Syscall 都有唯一的系统调用号对应。

基于 Syscall 机制，内核驻留在受保护的地址空间，用户空间程序无法直接执行内核代码，也无法访问内核数据，通过系统调用

系统调用列表可在 `linux/arch/sh/include/uapi/asm/unistd_64.h` 中找到，以 arm 为例，其他 Syscall 相关的源码包括：

- arch/arm/include/Uapi/asm/unistd.h
- arch/arm/kernel/calls.S
- arch/arm/kernel/entry-armv.S
- arch/arm/kernel/entry-common.S
- include/linux/syscalls.h
- include/uapi/asm-generic/unistd.h
- kernel/signal.c

Syscall 流程

x86/64

在 x86/64 架构上，`arch/x86/syscalls/syscall_64.tbl` 映射了系统调用号到函数定义的关系。同目录的 syscalltbl.sh 脚本，根据 syscall_64.tbl 生成一个头文件 arch/x86/include/generated/asm/syscalls_64.h，并用 __SYSCALL_COMMON 将系统调用号和系统函数地址做映射，最终保存到 sys_call_table 数组。

进程用户态执行 syscall 指令后进入到内核入口函数 system_call()，此时寄存器状态：

- rax -> 系统调用号
- rdi -> 参数 1
- rsi -> 参数 2
- rdx -> 参数 3
- r10 -> 参数 4
- r8 -> 参数 5
- r9 -> 参数 6

system_call() 的地址在内核初始化时被 syscall_init() 写入进 MSR_LSTAR 寄存器，该寄存器的作用是保存 syscall 指令的处理函数地址。

ARM

在 ARM 架构上，通过 swi 中断来实现系统调用，实现从用户态切换到内核态，发送软中断 swi 时，从中断向量表中查看跳转代码，其中异常向量表定义在文件 `arch/arm/kernel/entry-armv.S` 。当执行系统调用时会根据系统调用号从系统调用表中来查看目标函数的入口地址，在 calls.S 文件中声明了入口地址信息。

open / openat

当传给函数的路径名是绝对路径时，open 与 openat 无区别，此时 openat 自动忽略第一个参数 fd。

当传给函数的是相对路径时，如果 openat() 函数的第一个参数 fd 是常量 AT_FDCWD 时，则其后的第二个参数路径名是以当前工作目录为基址的；否则以 fd 指定的目录文件描述符为基址。

```
int open(const char *path, int oflag, mode_t mode);
int openat(int fd, const char *path, int oflag, mode_t mode);
```

read

read 的函数原型为 `ssize_t read(int fd, void *buf, size_t nbytes);` 。

read() 返回值为读到的字节数，如果已经到文件末尾，返回 0，若错误发生，返回-1。

write

write 的函数原型为 `ssize_t write(int fd, void *buf, size_t nbytes);` 。

write() 返回值通常与参数 nbytes 相同，否则可能出错。出错存在两种情况，返回-1 时表示写操作失败，返回值小于 nbytes 时，只写入了部分数据。

ioctl

ioctl 是设备驱动程序中设备控制接口函数，一个字符设备驱动通常会实现设备打开、关闭、读、写等功能，在一些需要细分的情境下，如果需要扩展新的功能，通常以增设 ioctl 命令的方式实现。

```
#include <sys/ioctl.h>
int ioctl(int fd, int cmd, ...) ;
```

其中 fd 是文件描述符，cmd 是交互协议，设备驱动将根据 cmd 执行对应操作，最后是可变参数，依赖 cmd 指定长度以及类型。ioctl 函数执行成功时返回 0，失败则返回 -1 并设置全局变量 errorno 值。

fcntl

fcntl 用于根据文件描述词来操作文件的特性，函数原型如下：

```
#include <fcntl.h>;
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

fcntl 函数有 5 种功能：

- 复制一个现有的描述符 `cmd=F_DUPFD`
- 获得/设置文件描述符标记 `cmd=F_GETFD 或 F_SETFD`
- 获得/设置文件状态标记 `cmd=F_GETFL 或 F_SETFL`
- 获得/设置异步 I/O 所有权 `cmd=F_GETOWN 或 F_SETOWN`
- 获得/设置记录锁 `cmd=F_GETLK,F_SETLK 或 F_SETLKW`

cmd 选项有

- **F_DUPFD 返回一个如下的描述符:**
  - 最小的大于或等于 arg 的一个可用的描述符
  - 与原始操作符一样的某对象的引用
  - 如果对象是文件的话, 返回一个新的描述符，这个描述符与 arg 共享相同的偏移量 (offset)
  - 相同的访问模式 (读、写或读/写)
  - 相同的文件状态标志 (如: 两个文件描述符共享相同的状态标志)
  - 与新的文件描述符结合在一起的 close-on-exec 标志被设置成交叉式访问 execve 系统调用

- **F_GETFD**
  - 取得与文件描述符 fd 联合 close-on-exec 标志，类似 FD_CLOEXEC
  - 如果返回值和 FD_CLOEXEC 进行与运算结果是 0 的话，文件保持交叉式访问 exec()
  - 否则如果通过 exec 运行的话，文件将被关闭 (arg 被忽略)
- **F_SETFD**
  - 设置 close-on-exec。该 flag 以参数 arg 的 FD_CLOEXEC 位决定
- **F_GETFL**
  - 取得 fd 的文件状态标志，如同下面的描述一样 (arg 被忽略)
- **F_SETFL**
  - 设置给 arg 描述符状态标志，可以更改的几个标志是：O_APPEND、O_NONBLOCK、O_SYNC 和 O_ASYNC
- **F_GETOWN**
  - 取得当前正在接收 SIGIO 或者 SIGURG 信号的进程 id 或进程组 id，进程组 id 返回成负值 (arg 被忽略)
- **F_SETOWN**
  - 设置将接收 SIGIO 和 SIGURG 信号的进程 id 或进程组 id，进程组 id 通过提供负值的 arg 来说明，否则 arg 将被认为是进程 id

F_GETFL 和 F_SETFL 的标志如下面的描述:

- **O_NONBLOCK**
  - 非阻塞 I/O
  - 如果 read 调用没有可读取的数据或者如果 write 操作将阻塞，read 或 write 调用返回-1 和 EAGAIN 错误
- **O_APPEND**
  - 强制每次写 (write) 操作都添加在文件大的末尾，相当于 open 的 O_APPEND 标志
- **O_DIRECT**
  - 最小化或去掉 reading 和 writing 的缓存影响，系统将企图避免缓存你的读或写的数据
  - 如果不能够避免缓存，那么它将最小化已经被缓存了的数据造成的影响. 如果这个标志用的不够好，将大大的降低性能
- **O_ASYNC**
  - 当 I/O 可用的时候，允许 SIGIO 信号发送到进程组，例如当有数据可以读的时候

fcntl 的返回值与命令有关。如果出错，所有命令都返回-1，如果成功则返回某个其他值。下列三个命令有特定返回值：F_DUPFD、F_GETFD、F_GETFL 以及 F_GETOWN。第一个返回新的文件描述符，第二个返回相应标志，最后一个返回一个正的进程 ID 或负的进程组 ID。

mmap

内存映射函数 mmap 负责把文件内容映射到进程的虚拟内存空间，通过对这段内存的读取和修改，来实现对文件的读取和修改，而不需要再调用 read/write 等操作。

```
void* mmap(void * addr, size_t len, int prot, int flags, int fd, off_t offset)
```

其中 addr 指定映射的起始地址, 通常设为 NULL, 由内核指定。length 表示映射到内存区域的大小。

prot 映射区的保护方式，可以是：

- PROT_EXEC: 映射区可被执行
- PROT_READ: 映射区可被读取
- PROT_WRITE: 映射区可被写入
- PROT_NONE: 映射区域不能存取

flags 表示映射区的特性，可以是:

- MAP_FIXED: 如果参数 `start` 所指的地址无法成功建立映射时，则放弃映射，不对地址做修正，通常不鼓励用此 flag。
- MAP_SHARED: 对映射区域的写入数据会复制回文件内，而且允许其他映射该文件的进程共享。
- MAP_PRIVATE: 对映射区域的写入操作会产生一个映射文件的复制，即私有的写入时复制 (copy-on-write) 对此区域作的任何修改都不会写回原来的文件内容。
- MAP_ANONYMOUS: 建立匿名映射。此时会忽略参数 fd，不涉及文件，而且映射区域无法和其他进程共享。
- MAP_DENYWRITE: 只允许对映射区域的写入操作，其他对文件直接写入的操作将会被拒绝。
- MAP_LOCKED: 将映射区域锁定住，这表示该区域不会被置换（swap）。

fd 是由 open 返回的文件描述符, 代表要映射的文件，如果使用匿名内存映射时，fd 设为-1。当系统不支持匿名内存映射时，则可以使用 fopen 打开/dev/zero 文件。

offset 表示以文件开始处的偏移量，必须是分页大小的整数倍，通常为 0，表示从文件头开始映射。

若映射成功则返回映射区的内存起始地址，否则返回 MAP_FAILED(-1)，错误原因存于 errno 中。

具体的错误代码如下:

- EBADF: 参数 fd 不是有效的文件描述词
- EACCES: 存取权限有误。如果是 MAP_PRIVATE 情况下文件必须可读，使用 MAP_SHARED 则要有 PROT_WRITE 以及该文件要能写入
- EINVAL: 参数 start、length 或 offset 有一个不合法
- EAGAIN: 文件被锁住，或是有太多内存被锁住
- ENOMEM: 内存不足

解除映射可使用 `munmap` 函数，原型为

```
int munmap(void *start, size_t length);
```

ptrace

ptrace 函数原型为 `long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);`，其中 `enum __ptrace_request request` 指示了 ptrace 要执行的命令，`pid_t pid` 指示 ptrace 要跟踪的进程，`void *addr` 指示要监控的内存地址，`void *data` 存放读取出的或者要写入的数据。

函数执行成功时返回 0，错误时返回-1，同时设置 errno。

其中 request 的选项包括：

- **PTRACE_TRACEME**
  - 表示本进程将被其父进程跟踪
- **PTRACE_PEEKTEXT**
  - 从内存地址中读取一个 WORD，内存地址为 addr
- **PTRACE_PEEKDATA**
  - 由于 Linux 不区分文本与数据段的地址空间，PTRACE_PEEKDATA 与 PTRACE_PEEKTEXT 无区别
- **PTRACE_PEEKUSER**
  - 从 USER 区域中读取一个 WORD，内存地址为 addr，值将作为结果返回，其中偏移地址 addr 通常是是字对齐的，data 将被忽略
- **PTRACE_PEEKTEXT**
  - 向内存地址中写入一个 WORD，内存地址为 addr
- PTRACE_PEEKDATA
- **PTRACE_PEEKUSER**
  - 向 USER 区域中写入一个 WORD，内存地址为 addr
- **PTRACE_CONT**
  - 继续运行之前停止的子进程
  - 如果 data 是非空的，它被解释为要发送给 Tracee 的信号
- **PTRACE_KILL**
  - 杀掉子进程
- **PTRACE_SINGLESTEP**
  - 设置单步执行标志
  - 子进程在每次机器指令后都被暂停
- **PTRACE_GETREGS**
  - 读取寄存器
- **PTRACE_GETFPREGS**
  - 读取浮点寄存器
- **PTRACE_SETREGS**
  - 设置寄存器
- **PTRACE_SETFPREGS**

  - 设置浮点寄存器
- **PTRACE_ATTACH**
  - attach 到一个指定的进程，使其成为当前进程跟踪的子进程
  - 子进程的行为等同于它进行了一次 PTRACE_TRACEME 操作
- PTRACE_DETACH
- **PTRACE_SYSCALL**
  - 进程在每次系统调用之后暂停
- **PT_DENY_ATTACH**
  - 反调试
- **PTRACE_GETSIGINFO**
  - 获取导致子进程停止执行的信号信息，并将其存放在父进程内由 data 指向的位置
  - addr 参数将被忽略
- **PTRACE_SETSIGINFO**
  - 将父进程内由 data 指向的数据作为 siginfo_t 结构体拷贝到子进程
  - addr 参数将被忽略
- **PTRACE_SETOPTIONS**
  - 将父进程内由 data 指向的值设定为 ptrace 选项
  - data 作为位掩码来解释
- **PTRACE_GETEVENTMSG**
  - 获取刚发生的 ptrace 事件消息，并存放在 data 指向的位置
  - addr 参数将被忽略
- **PTRACE_SYSEMU**
  - 用于用户模式的程序仿真子进程的所有系统调用
- **PTRACE_SYSEMU_SINGLESTEP**
  - 用于用户模式的程序仿真子进程的所有系统调用

其中 PTRACE_TRACEME 和 PTRACE_ATTACH 的区别为：PTRACE_TRACEME 是子进程主动申请被 TRACE。而 PTRACE_ATTACH 是父进程自己要 attach 到子进程，相当于子进程是被动的 trace。

WORD 的长度在 64 位程序中是 64 位，32 位程序中是 32 位。ptrace 系统调用在核心对应的处理函数为 sys_ptrace。

在 trace 中 Tracer 指追踪进程，Tracee 指被追踪、被观察的进程。

PTRACE_SETOPTIONS 的取值可能为：

- **PTRACE_O_EXITKILL**
  - 当跟踪进程退出时，向所有被跟踪进程发送 SIGKILL 信号将其退出
- **PTRACE_O_TRACECLONE**
  - 被跟踪进程在下一次调用 clone() 时将其停止，并自动跟踪新产生的进程
  - 新的进程在开始时收到 SIGSTOP 信号
  - 其新产生的进程的 pid 可以通过 PTRACE_GETEVENTMSG 获取

- **PTRACE_O_TRACEEXEC**
  - 被跟踪进程在下一次调用 exec() 函数时使其停止
- **PTRACE_O_TRACEEXIT**
  - 被跟踪进程在退出是停止其执行
  - 被跟踪进程的退出状态可通过 PTRACE_GETEVENTMSG 获得
- **PTRACE_O_TRACEFORK**
  - 被跟踪进程在下次调用 fork() 时停止执行，并自动跟踪新产生的进程
  - 新的进程在开始时收到 SIGSTOP 信号
  - 其新产生的进程的 pid 可以通过 PTRACE_GETEVENTMSG 获取
- **PTRACE_O_TRACEVFORK**
  - 被跟踪进程在下次调用 vfork() 时停止执行，并自动跟踪新产生的进程
  - 新的进程在开始时收到 SIGSTOP 信号
  - 其新产生的进程的 pid 可以通过 PTRACE_GETEVENTMSG 获取

execve

exec 系列系统调用的原型如下：

```
int execl(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

其中 path 参数表示要启动程序的完整路径，file 表示程序文件名，argv 表示执行参数，一般 argv 第一个参数为要执行命令名，不是带路径且 argv 必须以 NULL 结束，envp 表示环境变量。

上述 exec 系列函数底层都是通过 execve 系统调用实现，`int execve(const char *filename, char *const argv[], char *const envp[]);` 。

execve() 系统调用的作用是运行另外一个指定的程序。它会把新程序加载到当前进程的内存空间内，当前的进程会被丢弃，它的堆、栈和所有的段数据都会被新进程相应的部分代替，然后会从新程序的初始化代码和 main 函数开始运行。同时，进程的 ID 将保持不变。

fork

简介

fork 系统调用将创建一个与父进程几乎一样的新进程，复制了父亲进程的资源，包括内存的内容 task_struct 内容

在 Linux 的早期系统中，当发出 fork() 系统调用时，内核原样复制父进程的整个地址空间并把复制的那一份分配给子进程。这种完整复制需要如下操作：

- 为子进程的页表分配页帧
- 为子进程的页分配页帧
- 初始化子进程的页表

- 把父进程的页复制到子进程相应的页中

现代的 Linux 内核采用一种更为有效的方法，称之为写时复制。父进程和子进程共享页帧而不是复制页帧。然而，只要页帧被共享，它们就不能被修改，即页帧被保护。无论父进程还是子进程何时试图写一个共享的页帧，就产生一个异常，这时内核就把这个页复制到一个新的页帧中并标记为可写。原来的页帧仍然是写保护的：当其他进程试图写入时，内核检查写进程是否是这个页帧的唯一属主，如果是，就把这个页帧标记为对这个进程是可写的。

fork 失败时返回值为负数，成功时子进程返回 0，父进程返回子进程的 pid。

源码分析

fork 相关代码在以下的位置中：

- kernel/include/linux/sched.h
- kernel/include/linux/kthread.h
- kernel/arch/arm/include/asm/thread_info.h
- kernel/kernel/fork.c
- kernel/kernel/exit.c
- kernel/kernel/sched/core.c

fork 的调用流为 `fork->sys_fork->do_fork->copy_process` 。即用户空间调用 fork() 方法，经过 syscall 陷入内核空间，内核根据系统调用号找到相应的 `sys_fork` 系统调用。`sys_fork` 调用 `do_fork` 。`do_fork()` 会进行一些 check 过程，之后便是进入核心方法 `copy_process` 。

其中 fork/vfork/pthread_create 等几个系统调用都最终调用了 `do_fork`，pthread_create 调用的 flags 参数为 `CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND` ，fork 调用的 flags 参数为 `SIGCHLD` ，vfork 调用的 flags 参数为 `CLONE_VFORK | CLONE_VM | SIGCHLD` 。关于 flags 参数的意义在 clone 调用的文档中可以看到。

```
SYSCALL_DEFINE0(fork)
{
  return do_fork(SIGCHLD, 0, 0, NULL, NULL);
}
```

其中 `do_fork` 的实现为

```
long do_fork(unsigned long clone_flags,
        unsigned long stack_start,
        unsigned long stack_size,
        int __user *parent_tidptr,
        int __user *child_tidptr)
{
  return _do_fork(clone_flags, stack_start, stack_size,
      parent_tidptr, child_tidptr, 0);
}
```

其中 clone_flags 为 clone 方法传递过程的 flags，标记子进程从父进程中需要继承的资源清单。

stack_start 为子进程用户态的堆栈地址，fork() 过程该值为 0，clone() 过程赋予有效值。

stack_size 为不必要的参数，默认设置为 0。

parent_tidptr 为用户态下父进程的 tid 地址。

child_tidptr 为用户态下子进程的 tid 地址。

_do_fork 主要流程为：

- 执行 copy_process，复制进程描述符，分配 pid
- 如果由 vfork 调用，则执行相应的初始化过程
- 执行 wake_up_new_task，唤醒子进程，分配 CPU 时间片
- 如果由 vfork 调用，则父进程等待子进程执行 exec 函数来替换地址空间

copy_process 的主要流程为：

- 执行 dup_task_struct()，拷贝当前进程 task_struct
- 检查进程数是否超过系统所允许的上限 (默认 32678)
- 执行 sched_fork()，设置调度器相关信息，设置 task 进程状态为 TASK_RUNNING，并分配 CPU 资源
- 执行 copy_xxx()，拷贝进程的 files/fs/mm/io/sighand/signal 等信息
- 执行 copy_thread_tls()，拷贝子进程的内核栈信息
- 执行 alloc_pid()，为新进程分配新 pid

缺点

不是线程安全的，fork 创建的子进程只有一个线程（调用线程的副本），当一个线程在 fork 时，如果另一个线程此时进行内存分配并持有堆锁，任何在子进程中分配内存的尝试（从而获得相同的锁）都将立即发生死锁。

fork 对性能有较大的消耗。

vfork

简介

vfork 创建的子进程与父进程共享数据段，而且由 vfork() 创建的子进程将先于父进程运行。

- 由 vfork 创造出来的子进程还会导致父进程挂起，除非子进程 exit 或者 execve 才会唤起父进程
- 由 vfok 创建出来的子进程共享了父进程的所有内存，包括栈地址，直至子进程使用 execve 启动新的应用程序为止
- 由 vfork 创建出来的子进程不应该使用 return 返回调用者，或者使用 exit() 退出，但是可以使用 _exit() 函数来退出

源码分析

```
SYSCALL_DEFINE0(vfork)
{
  return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
      0, NULL, NULL, 0);
}
```

与 fork 区别

- fork 子进程拷贝父进程的数据段、代码段，vfork 子进程与父进程共享数据段
- fork 父子进程的执行次序不确定，vfork 保证子进程先运行

clone

简介

clone 的函数原型为 `int clone(int (fn)(void ), void *child_stack, int flags, void *arg);` 。

其中 fn 是函数指针，child_stack 是为子进程分配系统堆栈空间，flags 标志用来描述需要从父进程继承哪些资源，arg 是传给子进程的参数。

其中 flags 取值如下：

- **CLONE_PARENT**
  - 创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了“兄弟”而不是“父子”
- **CLONE_FS**
  - 子进程与父进程共享相同的文件系统，包括 root、当前目录、umask
- **CLONE_FILES**
  - 子进程与父进程共享相同的文件描述符（file descriptor）表
- **CLONE_NEWNS**
  - 在新的 namespace 启动子进程
  - namespace 描述了进程的文件 hierarchy
- **CLONE_SIGHAND**
  - 子进程与父进程共享相同的信号处理（signal handler）表
- **CLONE_PTRACE**
  - 若父进程被 trace，子进程也被 trace
- **CLONE_VFORK**
  - 父进程被挂起，直至子进程释放虚拟内存资源
- **CLONE_VM**
  - 子进程与父进程运行于相同的内存空间
- **CLONE_PID**
  - 子进程在创建时 PID 与父进程一致
- **CLONE_THREAD**
  - Linux 2.4 中增加以支持 POSIX 线程标准
  - 子进程与父进程共享相同的线程群

源码分析

```
#ifdef CONFIG_CLONE_BACKWARDS
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
     int __user *, parent_tidptr,
     unsigned long, tls,
     int __user *, child_tidptr)
#elif defined(CONFIG_CLONE_BACKWARDS2)
SYSCALL_DEFINE5(clone, unsigned long, newsp, unsigned long, clone_flags,
     int __user *, parent_tidptr,
     int __user *, child_tidptr,
     unsigned long, tls)
#elif defined(CONFIG_CLONE_BACKWARDS3)
SYSCALL_DEFINE6(clone, unsigned long, clone_flags, unsigned long, newsp,
    int, stack_size,
    int __user *, parent_tidptr,
    int __user *, child_tidptr,
    unsigned long, tls)
#else
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
     int __user *, parent_tidptr,
     int __user *, child_tidptr,
     unsigned long, tls)
#endif
{
  return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr, tls);
}
#endif
```

socket

socket 函数的原型为 `int socket(int domain, int type, int protocol);` 。

其中 domain 即协议域，又称为协议族 (family)。常用的协议族有 AF_INET、AF_INET6、AF_LOCAL(或称 AF_UNIX，Unix 域 socket)、AF_ROUTE 等等。协议族决定了 socket 的地址类型，在通信中必须采用对应的地址，如 AF_INET 决定了要用 IPv4 地址 (32 位的) 与端口号 (16 位的) 的组合、AF_UNIX 要使用一个绝对路径名作为地址。

支持的所有协议定义在 `/include/linux/socket.h` 中，具体内容如下：

```
/* Supported address families. */
#define AF_UNSPEC   0
#define AF_UNIX     1   /* Unix domain sockets          */
#define AF_LOCAL    1   /* POSIX name for AF_UNIX       */
#define AF_INET     2   /* Internet IP Protocol         */
#define AF_AX25     3   /* Amateur Radio AX.25          */
#define AF_IPX      4   /* Novell IPX                   */
#define AF_APPLETALK    5   /* AppleTalk DDP            */
#define AF_NETROM   6   /* Amateur Radio NET/ROM        */
#define AF_BRIDGE   7   /* Multiprotocol bridge         */
#define AF_ATMPVC   8   /* ATM PVCs                     */
#define AF_X25      9   /* Reserved for X.25 project    */
#define AF_INET6    10  /* IP version 6                 */
#define AF_ROSE     11  /* Amateur Radio X.25 PLP       */
#define AF_DECnet   12  /* Reserved for DECnet project  */
#define AF_NETBEUI  13  /* Reserved for 802.2LLC project*/
```

(下页继续)

(续上页)

```
#define AF_SECURITY 14  /* Security callback pseudo AF */
#define AF_KEY      15      /* PF_KEY key management API */
#define AF_NETLINK  16
#define AF_ROUTE    AF_NETLINK /* Alias to emulate 4.4BSD */
#define AF_PACKET   17  /* Packet family        */
#define AF_ASH      18  /* Ash              */
#define AF_ECONET   19  /* Acorn Econet         */
#define AF_ATMSVC   20  /* ATM SVCs         */
#define AF_RDS      21  /* RDS sockets          */
#define AF_SNA      22  /* Linux SNA Project (nutters!) */
#define AF_IRDA     23  /* IRDA sockets         */
#define AF_PPPOX    24  /* PPPoX sockets        */
#define AF_WANPIPE  25  /* Wanpipe API Sockets */
#define AF_LLC      26  /* Linux LLC            */
#define AF_IB       27  /* Native InfiniBand address    */
#define AF_MPLS     28  /* MPLS */
#define AF_CAN      29  /* Controller Area Network      */
#define AF_TIPC     30  /* TIPC sockets         */
#define AF_BLUETOOTH    31  /* Bluetooth sockets        */
#define AF_IUCV     32  /* IUCV sockets         */
#define AF_RXRPC    33  /* RxRPC sockets        */
#define AF_ISDN     34  /* mISDN sockets        */
#define AF_PHONET   35  /* Phonet sockets       */
#define AF_IEEE802154   36  /* IEEE802154 sockets       */
#define AF_CAIF     37  /* CAIF sockets         */
#define AF_ALG      38  /* Algorithm sockets        */
#define AF_NFC      39  /* NFC sockets          */
#define AF_VSOCK    40  /* vSockets         */
#define AF_KCM      41  /* Kernel Connection Multiplexor*/
#define AF_QIPCRTR  42  /* Qualcomm IPC Router          */
#define AF_SMC      43  /* smc sockets: reserve number for
                 * PF_SMC protocol family that
                 * reuses AF_INET address family
                 */
#define AF_XDP      44  /* XDP sockets          */

#define AF_MAX      45  /* For now.. */
```

type 指定 socket 类型，常用的 socket 类型为:

```
enum sock_type {
    SOCK_STREAM = 1, // 字节流套接字
    SOCK_DGRAM  = 2, // 数据报文套接字
    SOCK_RAW    = 3, // 原始套接字
    SOCK_RDM    = 4,
    SOCK_SEQPACKET  = 5, // 有序分组套接字
    SOCK_DCCP   = 6,
    SOCK_PACKET = 10,
};
```

protocol 指协议，常用的协议类型为:

```
/* Standard well-defined IP protocols.  */
enum {
  IPPROTO_IP = 0,       /* Dummy protocol for TCP       */
#define IPPROTO_IP      IPPROTO_IP
```

(下页继续)

(续上页)

```
  IPPROTO_ICMP = 1,      /* Internet Control Message Protocol    */
#define IPPROTO_ICMP      IPPROTO_ICMP
  IPPROTO_IGMP = 2,      /* Internet Group Management Protocol   */
#define IPPROTO_IGMP      IPPROTO_IGMP
  IPPROTO_IPIP = 4,      /* IPIP tunnels (older KA9Q tunnels use 94) */
#define IPPROTO_IPIP      IPPROTO_IPIP
  IPPROTO_TCP = 6,       /* Transmission Control Protocol    */
#define IPPROTO_TCP     IPPROTO_TCP
  IPPROTO_EGP = 8,       /* Exterior Gateway Protocol        */
#define IPPROTO_EGP     IPPROTO_EGP
  IPPROTO_PUP = 12,      /* PUP protocol             */
#define IPPROTO_PUP     IPPROTO_PUP
  IPPROTO_UDP = 17,      /* User Datagram Protocol       */
#define IPPROTO_UDP     IPPROTO_UDP
  IPPROTO_IDP = 22,      /* XNS IDP protocol         */
#define IPPROTO_IDP     IPPROTO_IDP
  IPPROTO_TP = 29,       /* SO Transport Protocol Class 4    */
#define IPPROTO_TP      IPPROTO_TP
  IPPROTO_DCCP = 33,        /* Datagram Congestion Control Protocol */
#define IPPROTO_DCCP      IPPROTO_DCCP
  IPPROTO_IPV6 = 41,        /* IPv6-in-IPv4 tunnelling      */
#define IPPROTO_IPV6      IPPROTO_IPV6
  IPPROTO_RSVP = 46,        /* RSVP Protocol            */
#define IPPROTO_RSVP      IPPROTO_RSVP
  IPPROTO_GRE = 47,      /* Cisco GRE tunnels (rfc 1701,1702)    */
#define IPPROTO_GRE     IPPROTO_GRE
  IPPROTO_ESP = 50,      /* Encapsulation Security Payload protocol */
#define IPPROTO_ESP     IPPROTO_ESP
  IPPROTO_AH = 51,       /* Authentication Header protocol   */
#define IPPROTO_AH      IPPROTO_AH
  IPPROTO_MTP = 92,      /* Multicast Transport Protocol     */
#define IPPROTO_MTP     IPPROTO_MTP
  IPPROTO_BEETPH = 94,      /* IP option pseudo header for BEET */
#define IPPROTO_BEETPH      IPPROTO_BEETPH
  IPPROTO_ENCAP = 98,       /* Encapsulation Header         */
#define IPPROTO_ENCAP       IPPROTO_ENCAP
  IPPROTO_PIM = 103,        /* Protocol Independent Multicast   */
#define IPPROTO_PIM     IPPROTO_PIM
  IPPROTO_COMP = 108,       /* Compression Header Protocol      */
#define IPPROTO_COMP        IPPROTO_COMP
  IPPROTO_SCTP = 132,       /* Stream Control Transport Protocol    */
#define IPPROTO_SCTP        IPPROTO_SCTP
  IPPROTO_UDPLITE = 136,    /* UDP-Lite (RFC 3828)          */
#define IPPROTO_UDPLITE     IPPROTO_UDPLITE
  IPPROTO_MPLS = 137,       /* MPLS in IP (RFC 4023)        */
#define IPPROTO_MPLS        IPPROTO_MPLS
  IPPROTO_ETHERNET = 143,   /* Ethernet-within-IPv6 Encapsulation   */
#define IPPROTO_ETHERNET    IPPROTO_ETHERNET
  IPPROTO_RAW = 255,        /* Raw IP packets           */
#define IPPROTO_RAW     IPPROTO_RAW
  IPPROTO_MPTCP = 262,      /* Multipath TCP connection     */
#define IPPROTO_MPTCP       IPPROTO_MPTCP
  IPPROTO_MAX
};
```

bind

bind 函数原型为 `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);` 其中 sockfd 为套接字的文件描述符，addr 指向存放地址信息的结构体的首地址，addrlen 存放地址信息的结构体的大小。

内核实现

bind 的内核实现如下：

```
SYSCALL_DEFINE3(bind, int, fd, struct sockaddr __user *, umyaddr, int, addrlen)
{
    struct socket *sock;
    struct sockaddr_storage address;
    int err, fput_needed;
    // 根据 fd 获取相应的 socket 结构体
    sock = sockfd_lookup_light(fd, &err, &fput_needed);
    if (sock) {
        err = move_addr_to_kernel(umyaddr, addrlen, &address);
        if (err >= 0) {
            err = security_socket_bind(sock, (struct sockaddr *)&address,
                                addrlen);//安全相关，暂不关注
            if (!err)
                err = sock->ops->bind(sock, (struct sockaddr *)
                                &address, addrlen);
        }
        fput_light(sock->file, fput_needed);
    }
    return err;
}
```

其中 sock->ops->bind 指向的是 inet_bind：

```
int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
{
    struct sockaddr_in *addr = (struct sockaddr_in *)uaddr;
    struct sock *sk = sock->sk;
    struct inet_sock *inet = inet_sk(sk);
    struct net *net = sock_net(sk);
    unsigned short snum;
    int chk_addr_ret;
    int err;

    if (sk->sk_prot->bind) {
        err = sk->sk_prot->bind(sk, uaddr, addr_len);
        goto out;
    }
    err = -EINVAL;
    if (addr_len < sizeof(struct sockaddr_in))
        goto out;

    if (addr->sin_family != AF_INET) {
        /* Compatibility games : accept AF_UNSPEC (mapped to AF_INET)
         * only if s_addr is INADDR_ANY.
         */
        err = -EAFNOSUPPORT;
```

(下页继续)

(续上页)

```
        if (addr->sin_family != AF_UNSPEC ||
            addr->sin_addr.s_addr != htonl(INADDR_ANY))
            goto out;
    }

    chk_addr_ret = inet_addr_type(net, addr->sin_addr.s_addr);

    err = -EADDRNOTAVAIL;
    if (!net->ipv4_sysctl_ip_nonlocal_bind &&
        !(inet->freebind || inet->transparent) &&
        addr->sin_addr.s_addr != htonl(INADDR_ANY) &&
        chk_addr_ret != RTN_LOCAL &&
        chk_addr_ret != RTN_MULTICAST &&
        chk_addr_ret != RTN_BROADCAST)
        goto out;

    snum = ntohs(addr->sin_port);
    err = -EACCES;

    // PROT_SOCK 的值为 1024, 非 root 用户不能使用小于 1024 的端口号
    if (snum && snum < PROT_SOCK &&
        !ns_capable(net->user_ns, CAP_NET_BIND_SERVICE))
        goto out;

    lock_sock(sk);

    /* Check these errors (active socket, double bind). */
    err = -EINVAL;

    if (sk->sk_state != TCP_CLOSE || inet->inet_num)
        goto out_release_sock;

    inet->inet_rcv_saddr = inet->inet_saddr = addr->sin_addr.s_addr;
    if (chk_addr_ret == RTN_MULTICAST || chk_addr_ret == RTN_BROADCAST)
        inet->inet_saddr = 0;  /* Use device */

    if ((snum || !inet->bind_address_no_port) &&
        sk->sk_prot->get_port(sk, snum)) {
        // 绑定失败, 绑定的地址已经有人在使用
        inet->inet_saddr = inet->inet_rcv_saddr = 0;
        err = -EADDRINUSE;
        goto out_release_sock;
    }

    if (inet->inet_rcv_saddr)
        sk->sk_userlocks |= SOCK_BINDADDR_LOCK;
    if (snum)
        sk->sk_userlocks |= SOCK_BINDPORT_LOCK;
    inet->inet_sport = htons(inet->inet_num);
    inet->inet_daddr = 0;
    inet->inet_dport = 0;
    sk_dst_reset(sk);
    err = 0;
out_release_sock:
    release_sock(sk);
out:
```

(下页继续)

(续上页)

```
    return err;
}
```

accept

accept 函数原型为 `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);` 其中 sockfd 为套接字的文件描述符，addr 指向存放地址信息的结构体的首地址，addrlen 存放地址信息的结构体的大小。

accept 在 socket()、bind()、listen() 之后调用，用于监听 TCP 请求。如果 accpet 成功，那么其返回值是由内核自动生成的一个的描述字，代表与返回客户的 TCP 连接。

具体来说，accept 的执行流程为：

- 创建一个 socket 结构体
- 获取一个未使用的文件描述符 fd
- 创建一个 file 结构体，并和 socket 关联
- 从全连接队列中获取客户端发来的请求
- 根据请求获取之前新建的 sock 结构体返回
- 将请求中的 sock 结构体和开始分配的 socket 结构体关联
- 将文件描述符 fd 和文件结构体 file 关联，并返回 fd 供用户使用

内核实现

accept 的内核实现如下：

```
SYSCALL_DEFINE3(accept, int, fd, struct sockaddr __user *, upeer_sockaddr,
        int __user *, upeer_addrlen)
{
    return sys_accept4(fd, upeer_sockaddr, upeer_addrlen, 0);
}
```

最终调用的是 SYSCALL_DEFINE4：

```
SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *, upeer_sockaddr,
        int __user *, upeer_addrlen, int, flags)
{
    struct socket *sock, *newsock;
    struct file *newfile;
    int err, len, newfd, fput_needed;
    struct sockaddr_storage address;

    if (flags & ~(SOCK_CLOEXEC | SOCK_NONBLOCK))
        return -EINVAL;

    if (SOCK_NONBLOCK != O_NONBLOCK && (flags & SOCK_NONBLOCK))
        flags = (flags & ~SOCK_NONBLOCK) | O_NONBLOCK;

    // 根据 fd 获取对应的 socket 结构
    sock = sockfd_lookup_light(fd, &err, &fput_needed);
```

(下页继续)

(续上页)

```
    if (!sock)
        goto out;

    err = -ENFILE;
    // 分配一个新的 socket 结构体
    newsock = sock_alloc();
    if (!newsock)
        goto out_put;

    newsock->type = sock->type;
    newsock->ops = sock->ops;

    __module_get(newsock->ops->owner);

    // 获取一个未使用的文件描述符, 这个描述符也就是 accept() 返回的 fd
    newfd = get_unused_fd_flags(flags);
    if (unlikely(newfd < 0)) {
        err = newfd;
        sock_release(newsock);
        goto out_put;
    }
    // 创建一个 file 结构体, 同时将这个 file 结构体和刚刚创建的 socket 关联
    // file->private_data 指向 socket
    newfile = sock_alloc_file(newsock, flags, sock->sk->sk_prot_creator->name);
    if (IS_ERR(newfile)) {
        err = PTR_ERR(newfile);
        put_unused_fd(newfd);
        sock_release(newsock);
        goto out_put;
    }

    err = security_socket_accept(sock, newsock);
    if (err)
        goto out_fd;

    // 调用 inet_accept() 执行主处理操作
    err = sock->ops->accept(sock, newsock, sock->file->f_flags);
    if (err < 0)
        goto out_fd;
    // 如果要获取对端连接信息, 那么拷贝对应信息到用户空间
    if (upeer_sockaddr) {
        // 调用 inet_getname() 获取对端信息
        if (newsock->ops->getname(newsock, (struct sockaddr *)&address,
                      &len, 2) < 0) {
            err = -ECONNABORTED;
            goto out_fd;
        }
        err = move_addr_to_user(&address,
                    len, upeer_sockaddr, upeer_addrlen);
        if (err < 0)
            goto out_fd;
    }

    /* File flags are not inherited via accept() unlike another OSes. */
    // 将文件描述符 fd 和文件结构体 file 关联到一起
    fd_install(newfd, newfile);
```

(下页继续)

(续上页)

```
    err = newfd;

out_put:
    fput_light(sock->file, fput_needed);
out:
    return err;
out_fd:
    fput(newfile);
    put_unused_fd(newfd);
    goto out_put;
}
```

inet_accept 函数调用传输层的 accept 操作，并且返回新的连接控制块，新的连接控制块需要与新的 socket 进行关联，accept 完成，将新 socket 的状态设置为已连接状态；

```
/*
 *    Accept a pending connection. The TCP layer now gives BSD semantics.
 */

int inet_accept(struct socket *sock, struct socket *newsock, int flags,
        bool kern)
{
    struct sock *sk1 = sock->sk;
    int err = -EINVAL;

    /* 执行传输层的 accept 操作 */
    struct sock *sk2 = sk1->sk_prot->accept(sk1, flags, &err, kern);

    if (!sk2)
        goto do_err;

    lock_sock(sk2);

    /* rps 处理 */
    sock_rps_record_flow(sk2);
    WARN_ON(!((1 << sk2->sk_state) &
          (TCPF_ESTABLISHED | TCPF_SYN_RECV |
          TCPF_CLOSE_WAIT | TCPF_CLOSE)));

    /* 控制块连接到新的 socket */
    sock_graft(sk2, newsock);

    /* 设置新 socket 的状态为连接 */
    newsock->state = SS_CONNECTED;
    err = 0;
    release_sock(sk2);
do_err:
    return err;
}
EXPORT_SYMBOL(inet_accept);
```

select

select 是 IO 多种复用的一种实现，它将需要监控的 fd 分为读，写，异常三类，其返回时是读、写、异常事件发生或者超时。

`select` 系统调用的原型如下：

```
int select(
    int __nfds,
    fd_set *__restrict __readfds,
    fd_set *__restrict __writefds,
    fd_set *__restrict __exceptfds,
    struct timeval *__restrict __timeout
);
```

wait / waitpid

wait

wait 的函数原型为 `pid_t wait (int * status);` ，相关的头文件是 `#include<sys/types.h>` / `#include<sys/wait.h>` 。

进程调用 wait 后，就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回。如果没有找到这样的子进程，wait 就会一直阻塞直到出现一个这样的子进程为止。

子进程的结束状态值会由参数 status 返回，如果不在意结束状态值，则参数 status 可以设成 NULL。

如果执行成功则返回子进程识别码 (PID)，如果有错误发生则返回-1。失败原因存于 errno 中。

waitpid

waitpid 的函数原型为 `pid_t waitpid(pid_t pid,int * status,int options);` ，相关的头文件是 `#include<sys/types.h>` / `#include<sys/wait.h>` 。

waitpid 会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 waitpid 时子进程已经结束，则 waitpid 会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，如果不在意结束状态值，则参数 status 可以设成 NULL。

参数 pid 为欲等待的子进程识别码，其他数值意义如下：

- **pid < -1**
  - 等待进程组识别码为 pid 绝对值的任何子进程
- **pid = -1**
  - 等待任何子进程，相当于 wait
- **pid = 0**
  - 等待进程组识别码与目前进程相同的任何子进程
- **pid > 0**
  - 等待任何子进程识别码为 pid 的子进程

参数 option 可以为 0 或下面的 OR 组合：

- **WNOHANG**

  - 如果没有任何已经结束的子进程则马上返回，不予以等待。
- **WUNTRACED**
  - 如果子进程进入暂停执行情况则马上返回，但结束状态不予以理会

子进程的结束状态返回后存于 status，可用下面几个宏判别结束情况：

- **WIFEXITED(status)**
  - 如果子进程正常结束则为非 0 值
- **WEXITSTATUS(status)**
  - 取得子进程 exit() 返回的结束代码，一般会先用 WIFEXITED 来判断是否正常结束才能使用此宏
- **WIFSIGNALED(status)**
  - 如果子进程是因为信号而结束则此宏值为真
- **WTERMSIG(status)**
  - 取得子进程因信号而中止的信号代码，一般会先用 WIFSIGNALED 来判断后才使用此宏
- **WIFSTOPPED(status)**
  - 如果子进程处于暂停执行情况则此宏值为真。一般只有使用 WUNTRACED 时才会有此情况
- **WSTOPSIG(status)**
  - 取得引发子进程暂停的信号代码，一般会先用 WIFSTOPPED 来判断后才使用此宏

如果执行成功则返回子进程识别码 (PID)，如果有错误发生则返回-1。失败原因存于 errno 中。

6.3.12 参考链接

文档与资源

- The Linux Kernel Archives
- The Linux Kernel documentation
- linux insides A book-in-progress about the linux kernel and its insides
- Embedded Linux Wiki
- namespaces Linux manual page

调试

- Linux 内核调试
- 利用 KGDB 调试内核驱动模块

漏洞利用

- Linux kernel pwn notes
- Linux kernel ROP
- SMEP bypass

文件系统

- Linux 文件系统剖析
- Docker: About storage drivers
- 嵌入式文件系统简介 (一) ——Linux MTD 设备文件系统

OverlayFS

- OverlayFS Wiki
- 内核 OverlayFS 文件层次信息
- 深入理解 overlayfs（一）
- 深入理解 overlayfs（二）：使用与原理分析
- Overlayfs And Containers

jffs

- JFFS2 文件系统及新特性介绍

网络

- Universal TUN/TAP device driver
- Tun/Tap interface tutorial

内核

- Linux Kernel Teaching
- slub 算法
- linux mmap 详解

安全机制

- Linux Crgoups 资源限制
- Linux Capabilities 入门

blog

- 容器简史：从 1979 至今的日子
- Linux TTY/PTS 概述
- The TTY demystified
- linux 中 ELF 加载过程分析
- ELF 文件的加载过程
- Linux Namespace 简介

6.4 Mac OS

6.4.1 参考链接

- The Darwin Kernel
- Kernel Architecture Overview
- iOS 冰与火之歌
- macOS notes

6.5 Windows

6.5.1 版本历史

Windows Version	Version	Release Date
Windows NT 3 .1	3.1	1993.7
Windows NT 3 .5	3.5	1994.9
Windows NT 3 .51	3.51	1995.5
Windows NT 4 .0	4.0	1996.7
Windows 2000	5.0	1999.12
Windows XP	5.1	2001.8
Windows Server 2003	5.2	2003.3
Windows Vista	6.0	2007.1
Windows Server 2008	6.0	2008.3
Windows 7	6.1	2009.10
Windows Server 2008 R2	6.1	2009.10
Windows 8	6.2	2012.10
Windows 10	10.0	2015.7
Windows Server 2016	10.0	2016.9
Windows Server 2019	10.0	2019.10

Windows NT

Windows NT 是 Microsoft 发行的一系列操作系统，其第一个版本于 1993 年 7 月发布。它是一个独立于处理器的多处理和多用户操作系统。

Windows NT 的第一个版本是 Windows NT 3.1，是为工作站和服务器计算机生成的。它旨在补充基于 MS-DOS(包括 Windows 1.0 到 Windows 3.1x) 的 Windows 的消费者版本。

渐渐地，Windows NT 系列已经扩展到微软针对所有个人计算机的通用操作系统产品线，不再使用 Windows 9x 系列。

NT 是第一个纯粹的 32 位 Windows 版本，而面向消费者的对应产品 Windows 3.1x 和 Windows 9x 则是 16 位/32 位的混合版本。它是一种多架构操作系统。最初，它支持多种指令集架构，包括 IA-32，MIPS 和 DEC Alpha; 稍后添加了对 PowerPC，Itanium，x64 和 ARM 的支持。

最新版本支持 x86(更具体地说是 IA-32 和 x64) 和 ARM。Windows NT 系列的主要功能包括 Windows Shell，Windows API，Native API，Active Directory，组策略，硬件抽象层，NTFS，BitLocker，Windows 应用商店，Windows Update 和 Hyper-V。

6.5.2 Windows 内核

运行模式

Windows 支持 Intel x86/x64 的两种处理器模式内核模式 (Kernel mode，Ring 0) 和用户模式 (User mode，Ring 3)。

内核模式下的代码可以访问所有的内存空间；可以直接操纵硬件。用户模式下的代码无权访问系统空间的内存页面；无法直接操纵硬件。用户模式向内核模式的切换是受控制的。

基本模块

Windows 操作系统内核模式下的基本模块包含 Windows 执行体、Windows 内核体、设备驱动程序、窗口和图形系统、硬件抽象层。

其中 Windows 执行体是 Windows 内核体的上层接口，包含了基本的操作系统服务，如进程与线程管理、内存管理、I/O 管理、网络连接、进程间通信以及安全服务。

Windows 内核体实现与硬件体系结构支持的代码，实现底层的操作系统功能，如线程调度、中断和异常分发处理、多处理器同步等。

设备驱动程序包括硬件设备驱动程序、文件系统与网络设备驱动程序。

窗口和图形系统即 Win32k.sys(Windows 子系统的内核模式部分)，实现了图形用户界面 (GUI) 函数，包括窗口的处理、绘制等。

硬件抽象层即 Hal.dll 文件，用于屏蔽 Windows 内核、驱动程序与平台硬件差异性的底层代码。

重要的系统进程

- **Idle**
  - 每个 CPU 一个 idle 线程做空闲 CPU 时间统计
- **System**
  - 内核模式的系统进程
- **Wininit.exe**
  - Session 0 初始化
- **Smss.exe**
  - 会话 (Session) 管理器，系统启动时第一个运行的进程。
- **Csrss.exe**
  - Windows 子系统进程，即客户端/服务器运行进程。
- **Winlogon.exe**
  - 处理交互式登录。
- **Services.exe**
  - 服务控制管理器，负责启动和、停止、暂停、恢复服务。
- **Svchost.exe**
  - 共享进程服务的宿主进程。
- **Lsass.exe**
  - 本地安全授权子系统，负责本地系统安全策略、用户认证，以及发送安全审计信息到事件日志。

6.5.3 服务

简介

服务 (Services) 程序是后台运行的进程，常用来执行特定的任务，不需要和用户进行交互。比如自动更新服务、后台智能传输服务、事件日志服务等。

服务程序受 Service Control Manager(SCM，即 services.exe 进程) 所控制，其服务程序的配置数据位于 `HKLM\System\CurrentControlSet\Services` 。

服务通常由三个部分组成：服务应用、服务控制程序 (service control program, SCP)、服务控制管理器 (service control manager, SCM)。

Windows 提供了内置的 SCP，可以启动、停止、继续运行程序。用户也可以自定义 SCP 程序来玩橙更细粒度的控制。服务程序是普通的 Windows 可执行程序，只是会有一些附加模块和 SCM 通信。

6.5.4 进程与线程

进程

进程 (Process) 是一个应用程序运行的实例，包含以下一些基本组件：私有虚拟地址、可执行体程序、被操作系统分配的一份资源句柄 (Handles) 列表、访问控制令牌 (Token)、进程标识号一个或多个线程。

访问控制令牌是用以唯一的标识所有者及其所属组以及和该进程相关联的特权 (Privilege) 信息。

进程创建步骤分为七步：

- 转换并校验参数和标记
- 打开可执行映像文件 (.exe)
- 创建 Windows 执行体进程对象
- 创建初始线程 (栈，上下文和执行体线程对象等)
- 通知 Windows 子系统初始化了一个进程
- 开始运行初始线程。(除非进程创建的时候被挂起)
- 在新进程和线程的上下文空间中，完成地址空间的初始化 (比如加载必需的 DLL 文件) 并开始执行程序

线程

线程 (Thread) 是 CPU 调度执行的基本单元，其包含 CPU 状态、两个栈、线程本地存储 (TLS)、线程标识号和访问控制令牌 (Access Token)。

线程的两个栈分别用于内核模式和用户模式。线程本地存储 (TLS) 包含一个私有存储空间，用来保存子系统、运行时库以及 DLL 文件等。

访问控制令牌 (Access Token)，用以唯一的标识所有者及其所属组以及和该线程相关联的特权 (Privilege) 信息

进程创建的主要过程：

- 转换并校验参数和标记
- 打开可执行映像文件 (.exe)
- 创建 Windows 执行体进程对象
- 创建初始线程 (栈，上下文和执行体线程对象等)
- 通知 Windows 子系统初始化了一个进程
- 开始运行初始线程 (除非进程创建的时候被挂起)
- 在新进程和线程的上下文空间中，完成地址空间的初始化 (比如加载必需的 DLL 文件) 并开始执行程序

数据结构

- 每个 Windows 进程都是由一个执行体进程 (EPROCESS) 块来表示，包含与进程有关的属性及其他的相关数据结构 (一个或者多个线程)。线程由执行体线程 (ETHREAD) 块来表示。
- EPROCESS 块和相关的数据结构位于系统地址空间中，不过 PEB (进程环境块) 和 TEB(线程环境块) 则位于进程地址空间中

6.5.5 内存管理

内存管理器

- 每个进程都有属于它的巨大的、连续的私有内存地址空间 (即虚拟地址)。
- **内存管理器的功能：**
  - 将虚拟的私有内存翻译/映射到真正存放数据的物理内存，可以隔离不同进程之间的数据读写。
  - 当物理内存耗尽时，将内存换页 (page) 到磁盘上去；然后当需要时再换页回来。

虚拟内存 (Virtural Memory)

- 每个进程都有它自己的虚拟地址空间。
- 虚拟内存提供了和物理内存不相关的逻辑视图。
- **换页 (Paging) 是和磁盘传输内存内容的过程**。
  - 虚拟内存的大小可以超过可用的物理内存。

6.5.6 IO 与驱动

管理器

- 输入输出管理器 (I/O Manager)
- 即插即用管理器 (Plug and Play Manager)
- 电源管理器 (Power Manager)

输入输出管理器 (I/O Manager) 接收应用程序的请求后，创建相应的 IRP (I/O request packets，输入输出请求数据包) 并传送至驱动程序进行处理：

- 根据 IRP 的请求，直接操作具体硬件设备，然后完成此 IRP，并返回
- 将此 IRP 的请求，转发到更底层的驱动程序中去，并等待底层驱动的返回
- 接收到 IRP 请求后，不是急于完成，而是分配新的 IRP 发送到其他驱动程序中去，并等待返回

6.5.7 文件系统

系统支持

Windows 支持以下几种文件系统格式：

- **CDFS**
  - 适用于 CD 的只读文件系统
- **UDF**
  - 适用于 DVD 的只读文件系统
- **FAT12, FAT16, FAT32**
  - File Allocation Table，文件分配表文件系统
- **NTFS**
  - New Technology File System，Windows NT 以及之后 Windows 的标准文件系统

FAT

FAT(File Allocation Table) 是“文件分配表”的意思。顾名思义，就是用来记录文件所在位置的表格，

FAT-16

FAT16 使用了 16 位的空间来表示每个扇区 (Sector) 配置文件的状态，故称之为 FAT16。FAT-16 磁盘分区最大只能到 2GB。

FAT-32

FAT16 使用了 32 位的空间来表示每个扇区 (Sector) 配置文件的状态，最大支持 32GB 分区，单个文件最大支持 4GB。

exFAT

exFAT(Extended File Allocation Table File System) 又称扩展 FAT 或扩展文件分配表。

NTFS

NTFS(New Technology File System) 是 Windows NT 内核的系列操作系统支持的、一个特别为网络和磁盘配额、文件加密等管理安全特性设计的磁盘格式，提供长文件名、数据保护和恢复，能通过目录和文件许可实现安全性，并支持跨越分区。

NTFS 支持功能

- 访问控制 (Access control)
- 磁盘配额 (Disk quotas)
- 加密文件系统 (Encrypting File System, EFS)
- 支持多重数据流 (Multiple data streams)

- 支持硬链接和联结点 (Hard links and junction points)
- 基于 Unicode 的命名方式

权限管理

文件夹的标准权限：读取、写入、列目录、读取和执行、修改、完全控制。

文件的标准权限：读取、写入、读取和执行、修改、完全控制。

多重权限规则：

- 权限是累加的
- 拒绝权限优先于其他权限
- 设置给文件的权限优先于给文件夹的权限

共享文件夹权限

- 比 NTFS 权限简单，优先级比 NTFS 权限低
- 和 NTFS 权限可以进行组合，结果取决于哪个权限更严格
- 共享允许的权限：读取、更改、完全操作

共享文件夹权限的特点

- 共享文件夹权限只适用于文件夹、而不适用于单独的文件，也不能对该文件夹中的子文件夹和文件设置共享权限。
- 共享文件夹权限并不对直接登录到计算机上的用户起作用，只适用于通过网络连接该文件夹的用户。
- 在 FAT32 文件系统上，共享文件夹权限是保证网络资源被安全访问的唯一方法。
- 默认的共享文件夹权限是指派给 Everyone 组的读取权限。

ReFS

ReFS(Resilient File System，弹性文件系统) 是在 Windows Server 2012 中新引入的一个文件系统。只能应用于存储数据，不能引导系统。ReFS 是与 NTFS 大部分兼容的，其主要目的是为了保持较高的稳定性，可以自动验证数据是否损坏，并尽力恢复数据。

6.5.8 文件加密与安全

EFS 文件加密系统

EFS(Encrypting File System, 加密文件系统)

- **公私钥加密体系**
  - 加密时，系统使用当前用户的公钥将数据加密并保存到硬盘上；解密时，系统使用该用户对应的私钥将数据解密出来。
- **EFS 的优势**

  - 与操作系统结合紧密，对用户透明。
- **支持 EFS 的操作系统**
  - Windows 2000
  - Windows XP 专业版
  - Windows Server 2003
  - Windows Vista/7/8 商业版、企业版、旗舰版
  - Windows Server 2008

EFS 的数据加密过程

- 文件被复制到临时文件。若复制过程中发生错误，则利用此文件进行恢复。
- 文件被一个随机产生的 Key 加密，这个 Key 叫作文件加密密钥 (FEK)，这个文件使用 DESX 或者 3DES 加密算法进行加密。
- 数据加密区域 (DDF，Data decryption field) 产生，这个区域包含了使用 RSA 算法加密的 FEK 和用户的公钥。
- 数据恢复区域 (DRF，Data recovery field) 产生，这个区域的目的是为了在用户解密文件的中可能解密文件不可用 (丢失 Key、离开公司等)。这些用户叫做恢复代理，恢复代理在加密数据恢复策略 (EDRP) 中定义，它是一个域的安全策略。如果一个域的 EDRP 没有设置，本地 EDRP 被使用。DRF 包含使用 RSA 算法加密的 FEK 和恢复代理的公钥。如果在 EDRP 列表中有多个恢复代理，FEK 须用每个恢复代理的公钥进行加密，也就是会为每个恢复代理创建一个 DRF。
- 包含加密数据、DDF 及所有 DRF 的加密文件被写入磁盘，文件大小一般保持不变。
- 在第 1 步中创建的临时文件被删除。

Bitlocker 磁盘加密

Bitlocker 是 Windows Vista 开始引入的一项数据保护新功能，可以解决计算机设备的物理丢失导致的数据失窃或恶意攻击泄漏。Bitlocker 通过将 Windows 的安装分区进行整卷加密，防止被攻击者通过启动其他操作系统来获取文件的“脱机攻击”。

Bitlocker 的工作模式

Bitlocker 的工作模式有 TPM(Trusted Platform Module) 模式和 U 盘模式。

TPM 模式要求计算机必须带有 TPM 芯片，这种芯片要通过硬件提供，一般只出现在安全性要求较高的商用计算机或工作站上。

U 盘模式要求计算机上有 USB 接口且 BIOS 支持在开机的时候访问 USB 设备，用于解密系统盘的密钥文件保存在 U 盘上，每次重启系统的时候必须在计算机上连接 U 盘。

文件数据的备份和还原

数据备份类型

- **常规备份 (完全备份，Full Backup)**
  - 对整个系统进行备份，包括操作系统和应用程序生成的数据。
  - 优点：当发生数据丢失的灾难时，只要用一盘磁带 (即灾难发生前一天的备份磁带)，就可以恢复全部的数据。
  - 缺点：数据量大，占用备份的存储设备较多，备份时间较长。
- **增量备份 (Incremental Backup)**
  - 每次备份的数据只是上一次备份后增加和修改过的数据。
  - 优点：没有重复的备份数据，节省存储空间，又缩短了备份时间。
  - 缺点：发生灾难时恢复数据比较麻烦，需要环环相套的所有备份。
- **差异备份 (Differential Backup)**
  - 每次备份的数据是相对于上一次全备份之后增加和修改过的数据。
  - 优点：备份所需时间短，节省磁带空间，灾难恢复也很方便，因为只需两盘磁带，即系统全备份与发生灾难前一天的备份。

删除与还原

Windows 删除文件只是简单的将硬盘上 MFT(主文件分配表) 中有关该文件的位置信息删除，而文件内容实际上还保存在硬盘相应的簇中。如果一个文件被删除了，但保存该文件的簇还没有被覆盖新数据的话，就可以将该文件重新恢复。如果需要彻底删除文件，不仅需要删除硬盘上 MFT 中的记录，还需要删除硬盘簇中实际保存的文件。

6.5.9 身份认证

SID

- Windows 使用 SID 来唯一表示安全主体，包括用户和组。Windows NT 6.x 中，系统服务也有 SID 标识。
- **查看账户的 SID**
  - whoami / user
  - PsGetSid
  - user2sid
- 从 Windows Vista/Server 2008(NT 6.0) 开始，每个服务程序都有自己的 SID，而账户的名称则为 “NT SERVICE< 服务名称 >”
- **查看服务 SID 及其账户名称**
  - sc.exe showsid <service name>
  - psgetsid <sid>

相关程序

Winlogon

- **可信任的进程，负责管理与安全有关的用户交互**
  - 协调登录过程，在登录时启动用户的第一个进程，处理注销过程，以管理其他各种与安全有关的操作，包括在登录时输入口令、更改口令、锁住/解锁工作站等。
  - 创建可用的桌面。
  - 向操作系统注册一个安全维护序列 (SAS, Secure Attention Sequence)，默认为 Ctrl+Alt+Delete。
  - 维护工作站状态。
  - 实现超时处理。
- 向 GINA 发送事件通知消息，提供可供 GINA 调用的各种接口函数。
- 保证其操作对其他进程不可见，从而防止登录密码等信息被截获。

GINA 动态链接库

- **提供了 Winlogon 用户标识和验证用户的输出函数**。
  - WlxActivateUserShell、WlxDisplaySASNotice、WlxInitialize、WlxLoggedOnSAS、WlxLoggedOutSAS、WlxLogoff、WlxNegotiate、WlxScreenSaverNotify、WlxShutdown、WlxStartApplication、WlxWkstaLockedSAS
- **微软提供的 GINA 是 MSGINA.dll，但允许被用户替换来自行定制系统的用户识别和身份验证**。
  - [HKEY_LOCAL_MACHINESOFTWAREMicrosoftWindowsNTCurrentVersionWinlogon] ”GinaDLL”=”ginadll.dll”

身份验证程序包

- **身份验证程序包的任务**
  - 验证用户；
  - 为用户新建 LSA 登录会话；
  - 返回绑定到用户安全令牌中的 SID。
- **身份验证程序包位于 DLL 动态链接库中**
  - 在系统启动期间被 LSA 所链接，接受输入的登录证书，通过验证程序决定是否允许用户登录。
- **Windows 安装的身份验证程序包**
  - 独立 (工作组) 环境：MSV1_0，%SystemRoot%System32Msv1_0.dll
  - 域环境：Kerberos，%SystemRoot%System32Kerberos.dll

LogonUI

LogonUI，Logon user interface，登录用户接口

- 为了保护 Winlogon 进程不崩溃，LogonUI 由 Winlogon 按需启动，真正加载 Credential provider，为用户提供验证身份的图形化界面。
- LogonUI 是用户模式下的进程，即%SystemRoot%System32LogonUI.exe

CP

CP，Credential provider，凭证提供者

- Credential provider 是运行在 LogonUI 进程内的 COM 对象，位于一些 DLL 文件中 (触发 SAS 热键后由 Winlogon 按需启动)，用来获取用户的用户名、密码、智能卡 PIN 码或者生物数据 (比如指纹、视网膜信息)
- 标准的 CP 是%SystemRoot%System32authui.dll 和%SystemRoot%System32SmartcardCredentialProvider.dll.

6.5.10 访问控制

访问控制的目的

- 限制安全主体 (用户、进程、服务等) 对客体 (文件目录、注册表键等) 的访问权限，从而使系统在安全合法范围内使用。
- **客体包括：**
  - 文件、目录
  - 注册表键
  - 活动目录对象
  - 内核对象 (事件、信号量、互斥体)
  - 服务
  - 进程、线程
  - 防火墙端口
  - 窗口工作站和桌面

Windows 访问控制的本质

安全主体的访问令牌 <- 比较 -> 客体的安全描述 - 用户成功登录时，系统的本地安全授权 (LSA) 为其创建访问令牌。- 用户启动程序创建进程时，线程获取该令牌的拷贝。- 用户程序请求访问客体时，向系统提交该令牌。- 系统使用该令牌与客体的安全描述进行比较来执行访问检查和控制。

安全访问令牌

- 用户帐号 SID(User account SID)
- 用户所属组的 SID(Group 1-n SID)
- 受限 SID (Restricted SID 1-n)
- **会话 ID (Session ID)**
  - 所属会话的 ID
- **特权列表 (Privilege1-n)**
  - 与该令牌关联的权限和特权列表
- **源 (Token Source)**
  - 创建此令牌的实体，比如会话管理器、网络文件服务器，或者 RPC 服务器等。
- **完整性级别 (Integrity Level)**
  - Windows Vista 开始引入，用以限制读取/写入对象的范围。
- **默认自主访问控制列表 (Default DACL)**
  - 用以创建对象时为其设置的初始 DACL
- **令牌类型 (Type)**
  - 主令牌 (primary) 还是模拟 (impersonation) 令牌。
- 模拟级别 (Impersonation Level)
- **令牌标记 (Flags) 和强制策略 (Mandatory Policy)**
  - 决定 UAC 和 UIPI 机制的行为表现。
- 默认主安全组 (Default primary group)

令牌类型

主令牌 (Primary Token)

- 每个进程都有一个主令牌来描述与该进程相关的用户帐号的安全上下文。模拟令牌 (Impersonation Token)
- 服务进程在自己的帐号下运行，使用自己的主令牌，但当服务接受一个用户的访问请求时，它创建一个线程来完成这项工作并将用户的访问令牌与工作线程相关联。用户的访问令牌是一个模拟令牌，用来标识用户、用户的组和特权。
- 当线程代表用户请求访问资源 (比如文件、打印机、数据库等) 时，在访问检查过程中使用该信息。在模拟结束后，线程重新使用主令牌并返回到服务自己的安全上下文里操作。
- 服务器只能在发起模拟请求的线程内模拟用户

模拟级别

- **为了防止滥用模拟机制，Windows 不允许服务器在没有得到用户同意的情况下执行模拟。用户进程在连接到服务器的时**
  - SecurityAnonymous 级别：最为限制，服务器不能模仿或者识别出用户
  - SecurityIdentification 级别：允许服务器得到用户的 SID 和特权，但不能模拟该用户
  - SecurityImpersonation 级别：默认级别，允许服务器在本地系统上识别和模拟该用户
  - SecurityDelegation：最为随意，允许服务器在本地系统或远程系统上模拟该用户

受限制的令牌 (Restricted Token)

- 出于安全原因，应用程序可创建一个受限的令牌并将它分配给子进程或模拟线程来为它们创建受限的安全上下文。
- **受限令牌在主令牌或模仿令牌的基础上创建的，来源于令牌的拷贝，但可能进行如下修改：**
  - 从该令牌的特权列表中删除一些特权
  - 该令牌中的 SID 被标记为仅仅拒绝 (Deny-only)
  - 该令牌中的 SID 被标记为受限制的 (Restricted)
- 在进行对象访问检查的时候，系统要执行两次访问检查：一次使用令牌的 “Enable SID” 和 “Deny SID”，另一个使用 Restricted SID 的列表。只有当这两个访问检查都允许请求的访问权限时才能允许访问。
- **受限令牌的应用被过滤的管理员令牌 (UAC 权限提升之前)**
  - 完整性级别被设置为中等
  - 除了 Change Notify、Shutdown、Undock、Increase Working Set、Time Zone 以外的其他特权都被去掉了
  - 给管理员以及类管理员的 SID 标记 “Deny-only”，可以消除以下安全缺陷：比如某个文件拒绝管理员组用户的访问，但允许其他包含这个管理员用户的组的访问，这将会导致这个用户最终能够访问该文件

账户权限和特权

- 进程在运行过程中执行的许多操作是无法通过对象访问保护来授权控制的，因为这些操作并没有与一个特定的对象打交道。Windows 使用特权 (Privilege) 和账户权限 (Account Right)，以使得管理员可以控制哪些账户能够执行与安全相关的操作。
- 特权是指一个账户执行某个与系统相关的操作的权限，比如，关闭计算机或者改变系统的时间。
- 账户权限则把 “执行某一特定登录类型 (比如本地登录或交互式登录到一台计算机上) 的能力” 授予它所针对的账户，或拒绝分配给该账户。

账户权限 (Account Rights)

- 账户权限并不是由安全引用监视器 (SRM) 强制实施的，也不存储在令牌中。
- 当一个用户企图登录到系统中时，作为对登录请求的响应，LSA 从 LSA 数据库中获取到已赋予该用户的账户权限，对登录类型 (交互式登录、网络登录、批作业方式登录、服务方式登录、终端服务登录等) 进行检查，如果该用户的账户没有“允许此种登录类型”的权限，或者具有“拒绝此种登录类型”的权限，则 LSA 拒绝该用户的登录请求

特权 (Privilege)

- 与用户权限不同，不同的特权是由不同的组件来定义的，并且也是由这些组件来强制使用的。比如调试特权是由进程管理器来检查的，它使得一个进程在利用 Windows API 函数 OpenProcess 来打开另一个进程的句柄时可以绕过安全检查。
- 与账户权限不同，特权是可以被允许和禁止的。想让一个特权检查能够成功地通过，该特权必须出现在当前特定的令牌中，而且它必须是允许 (Enabled) 的。

超级特权

- **调试程序 (SeDebugPrivilege)**
  - 具有此特权的用户可以打开系统中的任何一个进程，而不必考虑该进程上的安全描述符。例如，用户可以实现这样一个程序，它打开 LSASS 进程，将可执行代码拷贝到它的地址空间中，然后通过 CreateRemoteThread 注入一个线程，让它在一个更加有特权的安全环境下执行这些注入的代码。
- **接管所有权 (SeTakeOwnershipPrivilege)**
  - 特权持有者能够接管任何一个被保护对象的所有权。做法是，将他自己的 SID 写到该对象的安全描述符的所有者域中。由于所有者总是被授予“读取和修改该安全描述符的 DACL”许可，所以，具有此特权的进程可以修改此 DACL，以授予他自己对于该对象的完全访问权。
- **恢复文件和目录 (SeRestorePrivilege)**
  - 具有此特权的用户能够用自己的文件来替代系统中的任何文件。
- **加载和卸载设备驱动程序 (SeLoadDriverPrivilege)**
  - 一个恶意用户可以使用这种特权将一个设备驱动程序加载到系统中。设备驱动程序被认为是操作系统的可信任部分，操作系统会在 SYSTEM 账户凭证下运行设备驱动程序中的代码。
- **创建一个令牌对象 (SeCreateTokenPrivilege)**
  - 可以生成一些代表任意用户账户的令牌，而且其中的用户账户可以有任意的组成员关系和特权。
- **作为操作系统的一部分来执行 (SeTcbPrivilege)**
  - 具有此特权的恶意用户可以建立一个可信的 Lsass 连接，然后可以使用他的用户名和口令来创建一个新的登录会话 (通过 LsaLogonUser 这个创建新登录会话的函数)，而新的登录会话在其令牌中包含多个特权组或用户的 SID

完整性级别

强制完整性控制 (MIC ， Mandatory Integrity Control)

- Windows Vista 之后开始引入，将进程分为不同的完整性级别，从而保证低级别的进程无法影响高级别的进程。
- 进程的完整性级别由令牌中的 SID 声明。
- 系统首先进行完整性级别的检查，然后再进行 DACL 的检查。

安全描述 (Security Descriptor，SD)

- 版本号 (Revision Number)
- 标志 (Flags/Control)
- 所有者的 SID(Owner SID)
- 所有者主要组的 SID(Group SID)，仅被 POSIX 使用
- **DACL(Discretionary Access Control List)**
  - 自主访问控制列表：规定谁可以用什么方式访问对象。
- **SACL(System Access Control List)**
  - 系统访问控制列表：规定哪些用户的哪些操作应该被记录到安全审计日志中。

访问控制项 (ACE)

- ACE 大小 (Size)
- ACE 标志 (Flags)
- **ACE 类型 (Type)**
  - DACL 包含 ACE 的类型：访问允许、访问拒绝、允许的-对象、拒绝的-对象等。
  - SACL 包含 ACE 的类型：系统审计、系统审计-对象
- 用户 SID

DACL 访问权限的确定

确定对于对象期望的访问是否允许 (AccessCheck 函数)

- 如果该对象没有 DACL(即一个空 DACL)，则该对象没有设置保护，所以安全系统授予调用者期望的访问权限。
- 如果调用者具有“接管-所有权 (take-ownership)”的特权，则安全系统授予“写-所有者 (write-owner)”访问权，然后检查 DACL。
- 如果调用者是该对象的所有者，则授予“读-控制 (read-control)”和“写 DACL”访问权。
- **从前往后检查 DACL 中的每一个 ACE。对于每个 ACE，如果下面的条件之一被满足，则它被处理：**
  - “访问-拒绝”的 ACE，且 ACE 中的 SID 是一个可允许的 SID，或者是一个仅仅拒绝的 SID；
  - “访问-允许”的 ACE，且 ACE 中的 SID 不是仅仅拒绝类型的可允许 SID；

  - 在第二遍检查受限制的 SID 过程中，ACE 的 SID 与一个受限制的 SID 合；
  - ACE 没有标记为仅仅继承；
- 如果是一个“访问-允许”的 ACE，则只要调用者请求了，就将该 ACE 的访问掩码中的权限授予给它；如果调用者请求的所有访问权限都已经被授予了，则访问检查成功。如果是一个“访问-拒绝”的 ACE，只要调用者请求的任何一个访问权限属于拒绝访问的权限，则调用者对该对象的访问请求被拒绝。
- 如果已经到达 DACL 的末尾，而且调用者请求的有些访问权限仍然未被授予，则访问被拒绝。
- 如果所有的访问权限都被授予了，但是调用者的访问令牌至少有一个受限制的 SID，则系统重新扫描该 DACL 的 ACE，以寻找这样的 ACE：其访问掩码与用户所请求的访问权限相匹配，并且该 ACE 的 SID 与调用者的任何一个受限制的 SID 相匹配。只有对 DACL 的这两遍扫描都授予用户所请求的权限，才允许该用户访问此对象

用户权限控制

RunAs 服务 (辅助登录服务)

- 使得管理员平时使用标准的用户账户登录，然后在必要的时候调用具有更高权限的管理员控制台来执行管理任务。
- 在管理工具 (Administrative Tool) 应用组件上右击，然后从弹出的菜单中选择“运行为...”。
- 在 CMD 命令提示符中输入“runas”，并按回车键

用户账户控制 (UAC)

UAC，User Account Control

- 在 Windows Vista 之后的系统中，需要管理员权限的操作标有盾牌图标，表示需要经过系统 UAC 的提升确认才能进行进一步的操作。
- UAC 其实就是一种特殊的“缩减特权”模式。

用户界面权限隔离 (UIPI)

UIPI, User Interface Privilege Isolation

- UIPI 是 UAC 机制的一部分，目的在于防止窗口消息攻击。
- **通过结合完整性级别控制，UIPI 具体所做的保护包括：**
  - 低级别进程无法对高级别的窗口句柄做验证
  - 低级别进程无法向高级别进程的窗口发送消息 (SendMessage 或 PostMessage 等)
  - 低级别进程无法把线程注入到高级别进程
  - 低级别进程无法对高级别进程进行消息或日志挂钩
  - 低级别进程无法把 DLL 注入到高级别进程
  - 低级别的进程不可以读取高级别的内存地址空间

6.5.11 网络机制

Windows 的网络结构和组件

- **网络应用程序与服务进程**
  - 对应 OSI 应用层，通常使用各类网络 API DLL 来实现网络交互与通信功能。
- **网络 API DLL 及 TDI 客户端**
  - 对应 OSI 会话层与表示层，为应用程序提供了独立于具体协议的网络交互实现方式，包括 Windows 套接字 (Winsock)、远程过程调用 (RPC)、Web 访问 API、命名管道和邮件槽以及其他网络 API 接口。
  - TDI 客户端作为内核模式驱动程序，是网络 API 接口内核部分的具体实现：将网络 API 的请求转换成 IRP，通过 TDI 标准格式化后，发送给下层的协议驱动 (TDI 传输器)。

TDI

TDI(Transport Driver Interface) 传输驱动程序接口，也被称之为 TDI 传输器、NDIS 协议驱动程序、协议驱动程序等，对应 OSI 的网络层和传输层，实现了 TCP/IP、NetBEUI、IPX 等协议栈，接受上层 TDI 客户端的 IRP 请求，并调用 NDIS 库中提供的网卡驱动程序功能进行网络传输。

TDI 传输器通过透明地进行如分片与重组、排序确认与重传等一系列消息操作，为上层网络应用提供了便捷的支持。Windows Vista 之后，不再使用 TDI，而是 Windows filter platform(WFP) 和 Winsock kernel(WSK) 了。

WSK，Winsock 内核：提供内核模式下的网络通信，使用类似于用户态 Winsock 的编程语法，但也提供 IRP 和事件回调函数的异步 I/O 操作等特性。WSK 还在 Windows 下一代 TCP/IP 网络协议栈中默认支持 IPv6 功能。

WFP，Windows 过滤平台：一套 API 函数和系统服务，提供创建网络过滤应用程序的能力，允许应用程序追踪、过滤甚至修改网络数据包

NDIS

NDIS(Network Driver Interface Specification) 库及小端口 (miniport) 驱动程序位于 OSI 的链路层，为各种不同的网卡驱动程序和 TDI 传输层之间构建一个封装接口。

NDIS 库 (ndis.sys) 对于上层为网络程序模块屏蔽了不同的网卡硬件类型，NDIS miniport 驱动对于下层，为网卡制造商开发设备驱动程序提供了屏蔽 Windows 内核环境细节信息的编程接口。

NDIS 包含各种网卡硬件的设备驱动程序，处于 OSI 模型的物理层。

6.5.12 注册表

简介

注册表 (Registry) 是包含操作系统和其它软件的所有设置和配置相关数据的目录。

注册表的逻辑结构类似于磁盘上的文件系统。注册表包含了键 (key) 和值 (value)。其中键类似文件系统中的目录，而值就像文件，键可以包含子键 (subkey) 和值。值中存储着配置数据，数据有多种类型。最顶层的键称为根键 (root key)。

涉及到注册表的有：

1. 在启动时，boot loader 从注册表中读取数据。

2. 内核启动时，内核驱动从注册表中读取配置。
3. 在登陆时，读取网络驱动设置、墙纸、菜单、启动项等。
4. 在应用启动时，读取各种配置文件。

数据类型

- **REG_NONE**
  - 没有值
- **REG_SZ**
  - 固定长度 Unicode
- **REG_EXPAND_SZ**
  - 可变长度 Unicode
- **REG_BINARY**
  - 任意长度二进制数据
- **REG_DWORD**
  - 32 bit 数字
- **REG_DWORD_BIG_ENDIAN**
  - 32 bit 数字，大端
- **REG_LINK**
  - Unicode 符号链接，指向另一个注册表项
- **REG_MULTI_SZ**
  - Unicode 字符串
- **REG_RESOURCE_LIST**
  - 硬件资源描述
- **REG_FULL_RESOURCE_DESCRIPTOR**
  - 硬件资源描述
- **REG_RESOURCE_REQUIREMENTS_LIST**
  - 资源需求
- **REG_QWORD**
  - 64 bit 数字

root key

- **HKEY_CURRENT_USER：和当前登陆用户有关的数据**
  - AppEvents：事件关联
  - Console：控制台设定
  - Control Panel：控制面板
  - Environment：环境变量
  - EUDC：用户定义字符
  - Identities：Windows Mail 账户信息
  - Keyboard Layout：键盘布局
  - Network：网络驱动信息
  - Printers：打印机
  - Software：软件
  - Volatile Environment：可变的环境变量
- HKEY_USERS：所有用户有关的数据
- **HKEY_CLASSES_ROOT**
  - 文件后缀关联
  - COM(Component Object Model) 注册
  - UAC
- **HKEY_LOCAL_MACHINE：系统相关的数据**
  - HKLMBCD00000000
  - HKLMCOMPONENTS
  - HKLMHARDWARE
  - HKLMSAM
  - HKLMSECURITY
  - HKLMSOFTWARE
  - HKLMSYSTEM
- HKEY_PERFORMANCE_DATA：性能相关的数据
- HKEY_CURRENT_CONFIG：当前配置文件

6.5.13 WMI

简介

Windows Management Instrumentation (WMI) 是 Web-Based Enterprise Management (WBEM) 的一个实现，从 Windows NT 4.0 开始出现在所有的 Windows 操作系统中，用于提供操作界面和对象模式以便访问有关操作系统、设备、应用程序和服务的管理信息。

WMI 由 4 个部分组成，分别为：management applications, WMI infrastructure, providers, and managed objects。

工具

wmic

wmic 是一个与 WMI 进行交互的命令行工具，拥有大量的 WMI 对象的方便记忆的默认别名。

wbemtest

wbemtest 是一个带有图形界面的 WMI 诊断工具。它能够枚举对象实例、执行查询、注册事件、修改 WMI 对象和类，并且可以在本地或远程去调用方法。

winrm

winrm 是 WS-Management 协议的 Microsoft 实现，该协议为使用 Web 服务的本地计算机和远程计算机之间的通信提供了一种安全的方式。

wmic

wmic 是一个简单的 Linux 命令行工具，用于执行 WMI 查询。

6.5.14 安全机制

发展历史

- Windows 9x 内核几乎没有任何安全性机制
- Windows NT 内核引入了身份认证、访问控制和安全审计等安全控制机制
- Windows 2000 大部分的新增内容都集中在了安全方面，以活动目录为核心，包括公钥基础结构、组策略对象、Kerberos、智能卡支持、IPSec、加密文件系统、安全配置工具集等安全功能
- Windows XP SP2 引入了一些安全特性，包括 Windows 防火墙、Windows 安全中心、DEP 机制等
- Windows Vista 引入了众多新的安全机制，包括 UAC(用户帐户控制)、UIPI(用户界面权限隔离)、ASLR、服务会话隔离、文件/注册表虚拟化、高级安全防火墙、IE 低权限保护模式、内核 PatchGuard、代码签名、Bitlocker 等机制
- Windows 7 基本延续了 Windows Vista 的安全特性，只是做了些微调和安全性改进
- Windows 8 尤其在内核和硬件的安全性防护层面作了不少细节上的改变或增强
- Windows 10 在系统内核、应用组件、应用程序层面均有安全改进，同时也提供了一些新的安全功能

Windows 8 引入的重要安全性机制

- **ASLR 随机熵提高**
  - 在 Windows 8 中，特别是在 64 位下可以获得更高的随机熵。
- **空值引用保护 (NULL dereference protection)**
  - 空值引用 (NULL dereference) 是由于声明变量时将变量初始化为 Null，之后引用该对象却未进行空值判断造成的。如果攻击者能够故意触发空指针间接引用，攻击者就有可能利用引发的异常绕过安全逻辑，或致使应用程序泄漏调试信息，最终进行本地漏洞利用。
  - Windows 8 之前，内核允许用户进程映射 NULL 页，而 Windows 8 则禁止对前 64k 的空间进行映射。
- 内核缓冲区完整性检查
- **禁止零页内存分配**
  - Windows 操作系统中，零页内存供 16 位虚拟机 NTVDM 使用，以确保 16 位代码正常运行。之前系统中存在一些内核漏洞，通过 ZwAllocateVirtualMemory 等系统调用可以在进程中分配出零页内存，触发未初始化对象指针/数据指针引用漏洞或辅助漏洞进行攻击。
  - Window 8 中，禁止进程申请低地址内存 (0x0~0x10000)，同时，16 位虚拟机默认禁用，需要管理员权限才能开启。Windows 8 在所有可能的内存分配位置检查零页分配。
- **禁止 Win32k 系统调用**
  - Win32k.sys 是 Windows 内核漏洞高发对象，调用不受进程权限限制。
- 不可执行的非分页池
- **UEFI 启动技术**
  - Windows 8 拥有新的保护机制来对抗 Rootkit 和 Bootkit。所有版本的 Windows 8 都将包括统一可扩展固件接口 (UEFI) 安全开机功能 (Secure Boot)，该功能取代了标准 BIOS 来作为电脑的固件接口。
  - UEFI 安全启动协议是实现跨平台和固件安全的基础，与体系结构无关。在执行固件映像之前，安全启动基于公钥基础结构 (PKI) 流程来验证固件映像，帮助降低遭受启动加载程序攻击的风险。
- **Intel Secure Key 技术**
  - Intel 在 2012 年 4 月正式发布的 Intel 第三代 Core 处理器中加入了 Intel Secure Key 技术 (代号公牛山)。该技术提供对硬件实现的底层数字化随机数生成器 (DRNG) 的支持；提供基于硬件的高性能、高质量的熵和随机数生成器；引入新的指令 RDRAND，Windows 8 系统内核开始使用该指令产生随机数，如重要的 Security Cookie/ASLR 的生成过程

Windows 10 引入的重要安全性机制

- **基于硬件虚拟化的安全隔离**
  - Windows10 引入了 CredentialGuard 和 DeviceGuard 安全功能，运用硬件虚拟化技术，实现安全隔离。这两项功能主要存在于 Windows10 企业版中。
  - CredentialGuard 使用硬件虚拟化功能 (VSM) 将证书/令牌的存储和管理和真实操作系统隔离，使得恶意程序即使拥有系统内核权限，也无法获取用户的证书，避免攻击者使用例如 Mimikatz 之类的工具来实现企业网络的进一步渗透。
  - DeviceGuard 则可以允许企业管理和锁定设备，禁止设备上安装未受信的软件。
- **多因子安全认证的支持**

  - 支持移动设备、生物识别、PIN 码等多种方式的多因子认证保护，取代传统的 Windows 密码。

- **Edge 浏览器的安全改进**
  - 屏蔽了传统的 ActiveXBHOToolbars 的扩展以及一些过时的组件如 VBScript，减少了尤其是第三方控件引入的攻击面。
  - 启用了全 64 位进程和增强保护模式沙箱保护 (IE11 默认仅使用 32 位进程和保护模式沙箱)。
  - 渲染引擎核心也增强了针对过去较多影响 IE 浏览器的一些漏洞的防护或缓解能力。
- **内核模式字体引擎的安全改进**
  - 将内核模式字体引擎部分分离并放入隔离的用户模式环境中运行 (即 UMFD 机制，User Mode Font Driver，用户模式字体驱动)，有效防止了通过字体漏洞直接入侵 Windows 内核的攻击，同时也增加了可以通过组策略禁止或审计非系统字体加载的功能。
- **CFG(控制流保护/执行流保护) 内存保护机制**
  - Control Flow Guard(CFG) 技术是 Win 10 中开始默认启用的一种抵御内存泄露攻击的新机制 (后来被 Win 8.1 Update 3 补丁加入)，是为了弥补此前不完美的保护机制，例如地址空间布局随机化 (ASLR) 导致了堆喷射的发展，而数据执行保护 (DEP) 造成了返回导向编程 (ROP) 技术的发展。
  - 它是一种编译器和操作系统相结合的防护手段，目的在于防止不可信的间接调用。漏洞攻击过程中，常见的利用手法是通过溢出覆盖或者直接篡改某个寄存器的值，篡改间接调用的地址，进而控制了程序的执行流程。CFG 通过在编译和链接期间，记录下所有的间接调用信息，并把他们记录在最终的可执行文件中，并且在所有的间接调用之前插入额外的校验，当间接调用的地址被篡改时，会触发一个异常，操作系统介入处理

AMSI

AMSI(Anti-Malware Scan Interface，反恶意软件扫描接口) 是从 Windows 10 开始引入的一种机制。程序和服务可以将“数据”发送到安装在系统上的反恶意软件服务 (例如 Windows Defender) 上。

AMSI 基于 hook 来实现检测，例如，AMSI 会 hook WSH(Windows Scripting Host) 及 PowerShell 来分析正在执行的代码内容。

在 Windows 中，使用了 AMSI 的所有组件如下：

- 用户账户控制 (UAC)
- PowerShell
- Windows Script Host (wscript.exe / cscript.exe)
- JavaScript / VBScript
- Office VBA 宏

绕过

- hook amsi.dll
- `[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)`
- 混淆代码隐藏关键字
- 关键字字符串变形

内存破坏漏洞防护机制

DEP

DEP(Data Executive Protection，数据执行保护) 从 Windows XP SP2 开始引入，缺省仅为基本的 Windows 程序和服务启用 DEP。

DEP 的基本原理是将数据所在的内存页标记为不可执行，当程序产生溢出，恶意代码试图在数据段执行指令时，CPU 会产生异常而不去执行指令。

实现 DEP 机制需要 CPU 的支持。为此 AMD 公司推出了 EVP(enhanced virus protection) 技术，Intel 推出了 EDB (execute disable bit) 技术，这些技术在原理上均是在内存的页面表 (Page Table) 中加入一个特殊的标识位 (NX/XD) 来标识是否允许在该页上执行指令。

DEP 有四种可选参数：

–Optin：对于大多数用户版本的操作系统来说，默认仅将 DEP 保护是仅仅为一些基本的 Windows 程序和服务启用。该模式可被应用程序动态关闭–Optout：系统为所有在所选列表外的程序和服务启用 DEP，这种模式下，DEP 仍可被应用程序关闭。该模式多用于服务器版本的操作系统，如 Windows Server 2003/2008 –AlwaysOn：对所有的进程启用 DEP 的保护，不存在排除列表。该模式下，DEP 不可以被关闭，这是一种仅仅在 64 位操作系统上才能实现的工作模式，这在最大限度上保证了所有程序都能够抵御常见的数据溢出攻击–AlwaysOff：对所有的进程都禁用了 DEP，DEP 也不能被应用程序动态开启，这该模式一般只有在特殊场合才会使用

ASLR

ASLR(Address Space Layout Randomization，内存地址空间布局随机化) 在加载程序到内存空间时随机化各个模块的起始加载地址，防止攻击者定位攻击指令代码的位置。

ASLR 需要操作系统及应用程序的双重支持才能发挥作用，支持 ASLR 的程序在 PE 头中会设置 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识表明其支持 ASLR。

ASLR 主要影响的部分模块随机化、堆栈随机化和 PEB/TEB 随机化。模块随机化指系统将 PE 文件映射到内存时，对其加载基地址进行随机化处理，基地址在系统启动时确定，系统重启后会变化。堆栈随机化指每次程序加载后，其内存空间中堆、栈的基址都会发生变化。于是内存中的变量所在的地址也会发生变化。

ASLR 在 Windows Vista/7 引入实现，但机制尚不完善，攻击者还能在一定范围内进行漏洞利用。比如使用堆喷射、利用没有随机化的系统或软件的 EXE/DLL 等方式。

在安装了 KB2639308 更新的 Windows 7 以及 Windows 8 之后的版本，ASLR 被强制开启。

其他安全机制

EPM

EPM(Enhanced Protection Mode，增强保护模式) 也叫做“沙盒模式”(Sandbox Mode)，本质上是隔离进程和降低权限。该机制 Windows 8 的 IE 10 开始引入，从 Windows 8.1 开始默认启用。

PatchGuard

PatchGuard 是在 64 位版本的 Windows 操作系统中提供的新功能，用于保护操作系统的核心结构，防止他们被其他程序修改。

其对系统服务描述符表 SSDT(System Service Descriptor Table)、全局描述符表 GDT(Global Descriptor Table) 和中断描述符表 IDT(Interrupt Descriptor Table)、系统映像 System images(ntoskrnl.exe, ndis.sys, hal.dll) 等进行保护。

PatchGuard 处在系统任务的一个较高层面上，通过每隔一定时间进行一些固定的检查来确定这些系统关键内容是否更改。这些检查主要通过将核心内容与缓存中已保存的已知正确的备份进行对比，检测间隔大约为 5-10 分钟左右的某一随机选择时间。

PatchGuard 的缺点在于缺乏本地硬件水平的支持，只能通过轮询的形式，而不是采用事件驱动或硬件驱动的形式。

Code Signing

Code Signing(代码签名) 检查机制需要加载到系统内核中运行的驱动程序必须有数字签名以保证其代码的完整性，否则系统内核就不加载该驱动程序。

代码完整性检测被加载到内核中的驱动程序或系统文件是否已经被签名，或正在运行系统管理员账户权限的系统文件是否已被恶意软件篡改。在基于 x64 版本的操作系统下，内核模式的驱动程序必须进行数字签名后才能被加载。

AppLocker

AppLocker 是 Windows 7 系统和 Windows Server 2008 R2 中新增的一项安全功能，用以代替以前的 “软件限制策略” (Software Restriction Policies)，用于管理 Windows 运行的应用和文件，提供了各种不同程度的定制来方便用户/安全组制定规则。

管理员可以非常方便地进行配置，以实现用户可在计算机上可运行哪些程序、安装哪些文件、运行哪些脚本，可以控制以下这些类型的应用程序文件：

- 可执行程序 (.exe、.com)
- 动态链接库 (.dll、.ocx)
- 微软软件安装程序 (.msi、.msp)
- Windows Powershell 脚本程序 (.ps1)
- 批处理 (.bat、.cmd)
- Visual Basic 脚本 (.vbs)
- JavaScript(.js)

EMET

EMET(Enhanced Mitigation Experience Toolkit，增强缓解体验工具包) 在 2010 年 10 月 9 日推出，使得用户即使在未安装补丁的情况下也可以免受攻击。2014 年发布了最新的 5.1 版。

工具包括数据执行保护 (DEP)、结构化异常处理覆盖保护 (SEHOP)、随机地址空间分配 (ASLR)、空页 (Null Page) 保护等技术。最新版的还包括 ASR (Attack Surface Reduction)、EAF+ (Export Address Table Filtering Plus)、Deep Hook 等机制。

WFP / WRP

Windows 在 5.x 版本引入了 WFP(Windows 文件保护), 6.x 版本引入了 WRP(Windows 资源保护)。防止 Windows 重要的系统文件被恶意篡改、替换或删除。

WFP 的隐藏存储目录为 `%systemroot%/system32/dllcache` ，WRP 的隐藏存储目录为 `%systemroot%/winsxs/backup` 。

SRM

安全参考监视器 (Security Reference Monitor, SRM) 是 Windows 执行体 ntoskrnl.exe(内核态) 的一个组件，负责执行对象的访问控制、管理特权 (用户权限) 以及生成所有安全审计信息。

- **访问控制和特权管理**
  - 根据 LSA 配置安全访问控制策略，联合对象管理器 (Object Manager) 负责所有安全主体访问 Windows 资源对象的授权访问控制。
- **安全审计**
  - 根据 LSA 配置的安全审计策略，对访问过程中关注的事件进行记录，并由事件日志服务生成系统审计日志。

LSASS

本地安全授权子系统 (Local Security Authority Subsystem, LSASS) 包括用户态程序 lsass 和策略数据库，其中 lsass 程序的功能为：

- **用户身份验证和权限管理**
  - 负责交互式身份验证
  - 生成安全访问令牌
  - 分配用户特权
  - 确定用户权限
- **安全策略管理**
  - 管理本地安全策略
  - 管理审核策略
- **对象管理**
  - 建立可信任域列表
  - 内存的配额管理

LSASS 策略数据库 LSASS policy database 包含本地系统安全策略设置的数据库，数据库存储在注册表 HKLMSecurity 子键下面，包含：

- 哪些域是可信任的，从而可以认证用户的登录请求
- 谁允许访问系统，以及如何访问 (交互式登录、网络登录，或者服务登录)
- 分配给谁哪些特权
- 执行哪一种安全审计
- 域登录在本地缓存的信息
- Windows 服务的用户-账户登录信息

安全审计

- 对象管理器 (Object Manager) 可能会生成一些审计事件，并以此作为一次访问检查的结果；此外，有些可供用户应用程序使用的 Windows 函数也可以直接生成审计事件。
- 内核模式的代码总是允许生成审计事件。
- **有两个特权与审计有关：**
  - 一个进程必须有 SeSecurityPrivilege 特权才能管理安全事件日志 (Event Log)，以及查找或设置一个对象的 SACL。
  - 一个进程要调用审计系统服务，必须有 SeAuditPrivilege 特权才能成功地生成一条审计纪录

审核策略的类型

- **审核策略更改**
  - 确定是否对更改策略的每个事件进行审核
- **审核登录事件**
  - 确定是否审核每一个登录或注销计算机的用户实例
- **审核账户登录事件**
  - 确定是否审核在这台计算机用于验证账户时，用户登录到其它计算机或者从其它计算机注销的每个实例
- **审核账户管理**
  - 确定是否对计算机上的每个账户管理事件进行审核。包括创建、修改或删除用户账户或组；重命名、禁用或启用用户账户；设置或修改密码等
  - 确定是否对用户访问指定了自身 SACL(系统访问控制列表) 的对象 (如文件、文件夹、注册表项和打印机等) 的事件进行审核
- **审核目录服务访问**
  - 确定是否对用户访问指定了自身 SACL 的 Active Directory(活动目录) 对象的事件进行审核
- **审核过程追踪**
  - 确定是否审核事件的详细跟踪信息，如程序激活、进程退出、句柄复制和间接对象访问等
- **审核特权使用**
  - 确定是否对用户行使用户权限的每个实例进行审核
- **审核系统事件**

  - 确定在用户重新启动或关闭其计算机时，或者在影响系统安全或安全日志的事件发生时，是否进行审核

6.5.15 安全策略

本地安全策略

账户策略

账户策略仅涉及和用户账户的凭据相关的设置。通过设置账户策略、可以让所有本地账户更加安全，同时要破解账户密码所需的时间更长，所需技术更高。

- **账户策略的种类**
  - 密码策略
  - 账户锁定策略

密码策略

- **密码必须符合复杂性要求**
  - 建议设置为" 启用"
- **密码长度最小值**
  - 建议设置为“14”
- **密码最短使用期限**
  - 建议设置为“5 天”
- **密码最长使用期限**
  - 建议设置为“30 天”
- **强制密码历史**
  - 建议设置为“10 个”
- **用可还原的加密来存储密码**
  - 建议设置为“禁用”

账户锁定策略

- **账户锁定阈值**
  - 建议设置为“3 次”
- **账户锁定时间**
  - 建议设置为“30 分钟”，可根据实际需要更改
- **复位账户锁定计数器**
  - 建议设置为“30 分钟”，可根据实际需要更改

本地策略

本地策略中包含的全部是和账户无关的安全设置。通过设置本地策略，可以让 Windows 实现更严格的安全性，或者实现其他和安全有关的功能

本地策略的种类分为审核策略、用户权限分配、安全选项。

审核策略

- 审核策略更改
- 审核登录事件
- 审核对象访问
- 审核进程跟踪
- 审核目录服务访问
- 审核特权使用
- 审核系统事件
- 审核账户登录事件
- 审核账户管理

组策略

通过组策略 (Group Policy) 可以设置整个组织的集中化策略或分散式策略。

组策略对象

- **组策略对象 (Group Policy Object, GPO)**
  - 组策略对象实际上就是组策略设置的集合。
- **组策略对象包含：**
  - **针对计算机的组策略设置**
    * 应用在操作系统初始化和周期性更新循环过程中
    * 指定操作系统行为、桌面行为、安全性设置、计算机启动和关机指令、计算机赋予的应用程序选项以及应用程序设置
  - **针对用户的组策略设置**
    * 应用在用户登录计算机和周期性更新循环的过程中
    * 指定操作系统行为、桌面行为、安全性设置、赋予和公布的应用程序选项、应用程序设置、文件夹的重定向选项以及用户登录和注销指令。

组策略对象与活动目录

组策略对象与活动目录的联合使用实现了策略的集中与分散管理，适应从小到大的各种网络规模。

- 组策略对象能够链接到站点、域和组织单元，影响其中的用户和计算机。
- 应用组策略对象的顺序和级别决定了用户和计算机实际所采用的组策略设置。

组策略对象的管理单元

- **根节点**
  - **< 组策略对象名称 > [< 域名称 >] 策略**
    - 如：Default Domain Policy [SjtuInfosec.net] 策略
  - “本地计算机”策略
- **第二层节点**
  - 计算机配置
  - 用户配置
- **第三层节点**
  - 软件设置
  - Windows 设置
  - 管理模板

6.5.16 恶意软件

行为检测绕过

- 使用 Use Native API 调用
- 使用间接 API 调用
- 复制 API 函数代码
- 直接跳转制 API 函数
- 直接调用 System Call

DLL 注入

API Call

- CreateToolhelp32Snapshot
- Process32First
- Process32Next
- OpenProcess
- VirtualAllocEx
- CreateRemoteThread

6.5.17 术语与缩写

SSDT

SSDT(System Services Descriptor Table) 是系统服务描述符表，这个表把 Ring3 的 Win32 API 和 Ring0 的内核 API 联系起来。

SSDT 并不仅仅只包含一个庞大的地址索引表，它还包含着一些其它有用的信息，诸如地址索引的基地址、服务函数个数等。

通过修改此表的函数地址可以对常用 Windows 函数及 API 进行 Hook，从而实现对一些关心的系统动作进行过滤、监控的目的。

一些 HIPS、防毒软件、系统监控、注册表监控软件往往会采用此接口来实现自己的监控模块。

IDT

IDT(Interrupt Descriptor Table) 是中断描述符表，是操作系统用于处理中断的。

GDT

GDT，即全局描述表 (GDT Global Descriptor Table)。

LDT

LDT(Local Descriptor Table)，即局部描述符表。

常见缩写

- Alpc: Advanced Local Inter-Process Communication
- Cc: Common Cache
- Cm: Configuration manager
- Dbgk: Debugging Framework for User-Mode
- Em: Errata Manager
- Etw: Event Tracing for Windows
- Ex: Executive support routines
- FsRtl: File system driver run-time library
- Hvl: Hypervisor Library Io I/O manager
- Kd: Kernel Debugger
- Ke: Kernel
- Lsa: Local Security Authority
- Mm: Memory manager
- Nt: NT system services (most of which are exported as Windows functions) Ob Object manager
- Pf: Prefetcher
- Po: Power manager

- Pp: PnP manager
- Ps: Process support
- Rtl: Run-time library
- Se: Security
- Sm: Store Manager
- Tm: Transaction Manager
- Vf: Verifier
- Wdi: Windows Diagnostic Infrastructure Whea Windows Hardware Error Architecture Wmi Windows Management Instrumentation
- Zw: Mirror entry point for system services

6.5.18 参考链接

文档

- Windows Management Instrumentation
- Windows NT Wiki
- AppLocker
- Antimalware Scan Interface (AMSI)

漏洞利用

- windows kernel exploit tutorial
- HEVD
- HolicPOC

漏洞

- 进程注入

6.6 Android

6.6.1 设备识别码

IMEI

国际移动设备识别码（IMEI： International Mobile EquipmentIdentity），即通常所说的手机序列号、手机“串号”，用于在移动电话网络中识别每一部独立的手机等移动通信设备，相当于移动电话的身份证，IMEI 号共有 15~17 位数字。国际移动设备识别码一般贴于机身背面与外包装上，同时也存在于手机存储器中。

MEID

移动设备识别码 (MEID：Mobile Equipment Identifier) 是支持 CDMA 手机等移动通信设备唯一的识别码。

Android ID

Android_ID 是运行安卓系统的手机等设备随机生成一串 64 位的号码，每一个 Android_ID 都是独一无二。Android8.0 及更高版本中，Android_ID 的值通过签名密钥等确定范围。在低于 Android8.0 的版本中，Android_ID 在用户第一次启动设备时随机生成。在恢复出厂设置、或重装系统后，Android_ID 会被重置。获取 Android_ID 不需要开启任何系统权限，可直接通过运行相应代码获取。

设备序列号

设备序列号（SN：SerialNumber）通常用于验证产品的出厂生产的正规性和合法性，也称作机器码、认证码、注册码，通常可在系统信息中的序列号一栏看到，不可更改。App 通常可以使用命令、调用 READ_PHONE_STATE 权限相关函数读取 SN 值，但是因操作系统、手机型号版本等不同存在差异。

IMSI

国际移动用户识别码（IMSI：International Mobile SubscriberIdentification Number）是区别移动用户的标志，储存在 SIM 卡中，在全网和全球范围内唯一。

ICCID

手机 SIM 卡序列号（ICCID：ICC identity）是集成电路卡标识，是唯一标识 SIM 卡的序列号。ICCID 存储在 SIM 卡中，完整的 ICCID 为 19 或 20 个字符。

6.6.2 参考链接

- 手机设备识别码类型分析

6.7 iOS

6.7.1 设备识别码

IDFV

应用开发商标识符（IDFV - Identifier For Vendor）是苹果 iOS 系统给 App 开发商生成的唯一性标识符，同一个开发商开发的不同的 App 获取的 IDFV 值相同，常用于自开发 App 及跨 App 追踪用户行为。

CHAPTER 7

虚拟化

7.1 基础

7.1.1 简介

发展历史

- 1959 年虚拟化概念提出 Christopher Strachey 在国际信息处理大会上发表了虚拟化相关的文章
- 1960 年早期计算机虚拟化 IBM 研究出了 IBM 7044 计算机让用户可以在一台大型机上运行多个系统
- 1999 年现代计算机虚拟化实现 VMware 解决了 x86 虚拟化的问题，推出了虚拟化软件
- 2003 年开源虚拟化技术出现 Xen 是剑桥的一个开源项目，是版虚拟化技术的代表
- 2005 年硬件辅助虚拟化技术出现 Intel 和 ADM 推出了支持虚拟化技术的处理器和芯片组，实现了硬件辅助虚拟化技术
- 2006 年虚拟化技术在云计算中应用

定义

虚拟化是表示计算机资源的抽象方法，通虚拟化可以用于访问抽象前资源一致的方法访问抽象的资源。这种资源的抽象方法并不受实现、地理位置底层资源的物理配置的限制。

意义

- 效率：将原本一台服务器的资源分配给了多台虚拟化的服务器，有效地利用了闲置资源。
- 隔离：虚拟环境中运行的应用程序之间的隔离性优于在传统的非虚拟化系统中运行的应用程序，这对于系统的安全性和可用性都有所提升。
- 可靠：虚拟服务器是独立与硬件进行工作的，通过改进灾难恢复解决方案提高了业务的连续性。当一台服务器出现故障时可在最短时间内恢复且不影响整个集群的运作，在整个数据中心实现高可用性。
- 成本：降低了部署成本，只需要更少的服务器就可实现需要更多服务器才能做到的事情，也间接降低了安全等其他方面的成本。
- 兼容：虚拟化技术改进了桌面管理的方式，可部署多套不同的系统，将因兼容性造成问题的可能性降到最低。
- 便于管理：提高了服务器/管理员比率，一个管理员可以轻松地管理比以前更多的服务器而不会造成更大的负担。

7.1.2 原理

- **非特权指令**
  - 可以在任何级别下执行
- **特权指令**
  - 操作和管理系统资源，只能在最高权限级 ring0 上运行
  - 其他权限运行触发异常被捕获
- **敏感指令**
  - 操作特权资源的指令，读写敏感的寄存器或内存，访问内存系统或 IO 指令。
  - 部分为特权指令，17 条非特权指令
  - 所有的特权指令都是敏感指令。但不是所有的敏感指令都是特权指令
  - x86 架构下部分敏感指令可在低权限级别运行，不触发异常，不能被捕获

CPU 虚拟化

- **指令翻译**
  - 二进制代码动态翻译
  - 在执行时动态地重写虚拟机的执行代码
  - 在需要 VMM 监控和模拟的位置插入陷入
  - 不需要修改 guest OS
  - 动态翻译带来一定的性能开销
- **指令翻译模拟器**
  - 虚拟化技术的实质是一样的：将底层资源进行分区，并向上提供特定的和多样化的执行环境
  - 纯软件方法模拟和实际运行程序不同的指令集去执行，这种方式构造的虚拟机一般称为模拟器
  - 模拟器可将 guest OS 发出的所有指令翻译成本地指令集

  - 指令集虚拟化系统的代表包括 Bochs 和 QEMU 等

- **Bochs**
  - x86 PC 模拟器可以运行在大多主流平台上，包括 x86 PowerPC Alpha Sun 和 MIPS
  - 翻译每一条指令
  - 从加电到重新启动模拟 x86 CPU，为所有标准 PC 外围设备装配设备模型，支持无修改软件执行 (包括操作系统)
  - 模拟需要额外开销
- **QEMU**
  - 使用动态翻译器，Kqemu 使用指令翻译缓冲区快速处理指令翻译
  - 支持两种操作模式：User Mode 和 System Mode
  - 在 User Mode 中，可以将 CPU 上编译的 Linux 进程装载到另一个 CPU 中，或跨编译进行交叉调试
  - 在 System Mode 中，可以模拟完整系统，包括处理器和外围设备
  - QEMU 支持多种处理器架构的模拟，包括 x86、ARM、PowerPC 和 Sparc

计算机虚拟化

- **VMM(Virtual Machine Monitor) 虚拟机监控层：运行于服务器硬件和虚拟机之间的中间软件层，又称为 hypervisor**
  - 管理所有的系统资源
  - 管理虚拟机，提供虚拟机硬件模拟
  - 协调虚拟机使用服务器硬件
- 宿主机操作系统：host OS
- 客户机操作系统：guest OS
- **计算机虚拟化**
  - CPU/内存/IO 虚拟化

7.1.3 虚拟化方式分类

- **半虚拟化 (Para-Virtualization)**
  - 客户操作系统通过修改能和物理硬件进行互动，效率较高，代表是 Xen
  - 实现方式：修改操作系统
  - 和全虚拟化相比，架构更精简，而且在整体速度上有一定的优势。
  - 缺点是需要对 Guest OS 进行修改，所以在用户体验上不太好，对非开放的操作系统则无法提供支持
- **完全虚拟化 (Full Virtualization)**
  - 客户操作系统不用修改，硬件由 VMM 模拟，指令也由 VMM 进行模拟并返回给客户操作系统
  - **实现方式**

    * 软件实现：要经过模拟处理，性能较低，对系统负荷较大
    * 硬件辅助：通过修改虚拟机和 VMM 运行的 ring 级别，提高性能

半虚拟化

- **CPU 半虚拟化**
  - 在半虚拟化实现中，认为与其千方百计去捕获敏感指令，不如直接不用这些指令。于是对 GuestOS 进行了一些修改，替换掉这些指令，转而调用 VMM 提供的特殊 API(hypercall) 来进行模拟。当 Guest OS 需要执行敏感操作时，直接通过 hypercall 调用 VMM，避免了捕获的开销。
  - 不需要 hypervisor 捕获特权指令而耗费一定的资源进行翻译操作，从而获得额外的性能和高扩展性，使其性能非常接近物理机
  - 这种方式需要修改操作系统内核，将不能虚拟化的指令替换为 hypercall，hypercall 直接与虚拟层通信，虚拟层提供内核操作的关键接口，如内存管理、中断处理和时间管理等。
- **内存半虚拟化**
  - 在 VMM 的帮助下，使 guest OS 能够利用物理 MMU 一次完成由虚拟机地址到机器地址的三层转换技术
  - guest OS 的客户页表中的地址不再是客户物理地址，而是机器地址
  - 为了保护各个虚拟机内存空间相互独立，VMM 在对页表进行地址替换前，会对页表中的每一个页表项进行检查，以确保只映射了属于该虚拟机的机器页面，而且不得包含对页表页面的可写映射
- **I/O 设备半虚拟化**
  - 在半虚拟化下，修改 Guest OS 内核，将原生设备驱动从 Guest OS 中移出，放在一个经过 VMM 授权的设备虚拟机中，其余虚拟机中 Guest OS 的 I/O 请求都交由设备虚拟机处理。

全虚拟化

- 在虚拟化技术的早期，计算机没有在硬件层次上对虚拟化技术提供支持，因此虚拟化技术主要基于软件实现。
- 完全虚拟化方法在虚拟服务器和底层硬件之间建立一个抽象层，捕捉和处理那些对虚拟化敏感的特权指令，为指令访问硬件控制器和外设充当中介，使客户操作系统无需修改就能在虚拟服务器上运行，就像运行在真实的物理环境下一样。
- **主要实现技术**
  - 优先级压缩
  - 二进制代码翻译
- **CPU 全虚拟化**
  - 客户操作系统运行在 Ring 1 级，VMM 运行在 Ring 0 级，VMM 提供给操作系统各种虚拟资源 (虚拟 BIOS、虚拟设备和虚拟内存管理等)。对于不能虚拟化的特权指令，通过二进制转换方式转换为同等效果的指令序列运行，而用户级指令可直接运行
- **内存全虚拟化**
  - **影子页表**
    * 虚拟地址 <=> 物理地址

        * 虚拟机虚拟地址 <=> 虚拟机物理地址 <=> 真实的机器地址
    - 影子页表对 guest OS 完全透明
    - **维护影子页表的时间开销和空间开销很大**
        * 缺页补全
        * 每个虚拟机到需要一套影子页表
- **I/O 设备全虚拟化**
    - 不修改 guest OS
    - VMM 处理设备的方式根据 VMM 的位置不同
- **全虚拟化**
    - **代表产品**
        * VMware vSphere 和 Hyper-V
        * 开源 KVM
    - **优点**
        * Guest OS 无需修改
    - **缺点**
        * 开销
- **全虚拟化-硬件辅助**
    - **指令虚拟化**
        * 用户指令直接在硬件上执行
        * 部分特权指令直接执行
        * 部分特权指令在 VMM 上执行
    - **存储器的虚拟化**
        * NPT
        * 影子页表
    - **I/O 设备**
        * 指令模拟
        * Intel - VT
- **硬件辅助虚拟化**
    - **代表产品**
        * Virtual Box / KVM
    - **优点**
        * 引入硬件技术，使虚拟化技术更接近物理机的速度
    - **缺点**
        * 硬件实现不够优化，还有提高空间

7.1.4 内存虚拟化

地址类型

- Guest 虚拟地址 (Guest Virtual Address, GVA)
- Guest 物理地址 (Guest Physical Address, GPA)
- Host 虚拟地址 (Host Virtual Address, HVA)
- Host 物理地址 (Host Physical Address, HPA)

7.1.5 虚拟化技术应用

内核漏洞检测

- **User Space**
  - 部署一些分析和测试的应用程序
- **Kernel Space**
  - 主要工作包括设置监视的内存区域、与超级管理程序通信、拦截特定的函数
- **Hypervisor**
  - 主要负责进行具体的监测检查工作

完整性保护

完整性是指能够保障被传输、接收或存储的信息是完整的和未被篡改的，是信息安全的重要属性之一。关于完整性保护的研究主要集中在文件系统完整性和内核代码完整性方面。

入侵检测

现有的入侵检测系统给系统管理员带来了两难的选择：如果将入侵检测系统部署在主机上，则它可以清晰的观察到主机的系统状态，但是容易遭到恶意攻击或者被屏蔽；如果将其部署在网络上，则它可以更好的抵御攻击，但是对主机内部的状态一无所知，因此可能让攻击者逃脱。

基于虚拟化的入侵检测系统，可以做到既能够观察到被监控系统的内部状态，又能与被监控系统隔离。VMM 能够直接观察到被监控系统的内部状态，可以通过直接访问其内存来重构出 GuestOS 的内核数据结构，通过单独运行的入侵检测系统来进行检测。这种在虚拟机外部监控虚拟机内部运行状态的方法称为虚拟机自省 (virtual machine introspection, VMI)。

恶意代码检测与分析

- **原理**
  - 恶意软件具有静态的特征码，其感染破坏的流程中存在行为特征
  - 恶意软件入侵系统后会进行隐藏系统进程、添加注册表启动项等操作
- **现有解决方案**
  - 基于多引擎的特征码和行为特征检测
  - 基于入侵检测

7.1.6 虚拟机漏洞

- 虚拟机逃逸
- 拒绝服务攻击

7.2 虚拟化技术

7.2.1 中断虚拟化

中断从设备发送到 CPU 需要经过中断控制器，现代 x86 架构采用的中断控制器被称为 APIC(Advanced Programmable Interrupt Controller)。APIC 是伴随多核处理器产生的，所有的核共用一个 I/O APIC，用于统一接收来自外部 I/O 设备的中断，而后根据软件的设定，格式化出一条包含该中断所有信息的 Interrupt Message，发送给对应的 CPU。

7.3 Virtio

7.3.1 简介

virtio 是一种 I/O 半虚拟化解决方案，是一套通用 I/O 设备虚拟化的程序，是对半虚拟化 Hypervisor 中的一组通用 I/O 设备的抽象。提供了一套上层应用与各 Hypervisor 虚拟化设备（KVM，Xen，VMware 等）之间的通信框架和编程接口，减少跨平台所带来的兼容性问题，大大提高驱动程序开发效率。

7.3.2 架构

virtio 可以分为四层，包括前端 guest 中各种驱动程序模块，后端 Hypervisor 上的处理程序模块，中间用于前后端通信的 virtio 层和 virtio-ring 层，virtio 这一层实现的是虚拟队列接口，算是前后端通信的桥梁，而 virtio-ring 则是该桥梁的具体实现，它实现了两个环形缓冲区，分别用于保存前端驱动程序和后端处理程序执行的信息。

7.3.3 设备类型

设备类型与 ID

设备 ID	设备名称
0	reserved (invalid)
1	network card
2	block device
3	console
4	entropy source
5	memory ballooning (traditional)
6	ioMemory
7	rpmsg
8	SCSI host
9	9P transport
10	mac80211 wlan
11	rproc serial
12	virtio CAIF
13	memory balloon
16	GPU device
17	Timer/Clock device
18	Input device

网络设备

virtio 网络设备是虚拟以太网卡，是 virtio 支持的最复杂的设备。

块设备

virtio 块设备是简单的虚拟块设备（即磁盘）。读取和写入请求（以及其他请求）默认被放置在队列中，并由设备提供服务（可能是无序的）。

Console 设备

virtio 控制台设备是用于数据输入和输出的简单设备。一台设备可以有一个或多个端口。每个端口都有一对输入和输出虚拟队列。此外，设备具有一对控制 IO 虚拟状态。控制信息用于在设备和驱动程序之间传递有关在连接的任一侧打开和关闭端口的信息，从设备指示特定端口是否为控制台端口，添加新端口，端口热插拔/ 拔出等，并从驱动程序指示是否已成功添加端口或设备，端口打开/关闭等。对于数据 IO，在接收队列中放置一个或多个空缓冲区，用于输入数据和输出字符。放置在传输队列中。

熵设备

熵设备提供了高度的随机性。

Balloon 设备

通常来说，要改变客户机占用的宿主机内存，是要先关闭客户机，修改启动时的内存配置，然后重启客户机才能实现。而内存的 ballooning（气球）技术可以在客户机运行时动态地调整它所占用的宿主机内存资源，而不需要关闭客户机。

SCSI Host 设备

virtio SCSI Host 设备将一个或多个虚拟逻辑单元（例如磁盘）组合在一起，并允许使用 SCSI 协议与其通信。

7.3.4 参考链接

官方文档

- virtio: Towards a De-Facto Standard For Virtual I/O Devices
- Virtual I/O Device (VIRTIO) Version 1.0
- libvirt

Blog

- Virtio 简介
- Virtio：针对 Linux 的 I/O 虚拟化框架
- 半虚拟化 I/O 框架 virtio

7.4 VMWare

7.4.1 简介

VMWare 是基于宿主模式的桌面虚拟软件，包括 VMX 驱动、ring0 的 VMM、ring3 的用户态程序。

VMware vSphere 则是一个虚拟化方案，其核心是 ESXi，ESXi 独立安装在裸机上的操作系统（注意它不基于任何 OS，它本身就是 OS），通过它物理机的硬件资源被虚拟化为虚拟资源，之后再通过 vCenter 就能将安装了 ESXi 操作系统的物理机的资源进行整合，化为一个总的资源池，在这个资源池里面为各个部门划分不同大小的资源池方便其使用。

7.4.2 安全策略

vSphere

ESXi 通过以下功能提供附加 VMKernel 保护

- 内存强化：将 ESXi 内核、用户模式应用程序及可执行组件 (如驱动程序和库) 位于无法预测的随机内存地址中。在将该功能与微处理器提供的不可执行的内存保护结合使用。
- 内核模块完整性：数字签名确保由 VMKernel 加载的模块、驱动程序及应用程序的完整性和真实性。
- 可信的平台模块：vSphere 使用 Intel 可信的平台模块/受信任的执行技术（TPM/TXT），根据硬件信任根提供管理程序映像的远程证明。

虚拟机安全策略

- **虚拟机常规保护**
  - 包括安装防毒软件、配置客户机操作系统的日志记录级别等
- **禁用虚拟机中不必要的功能**
  - 移出不必要的硬件设别、禁用操作系统中未使用的服务、限制公开复制到剪贴板中的敏感数据、限制用户在虚拟机中运行命令、限制客户机操作系统写入主机内存等
- **限制信息性消息从虚拟机流向 VMX 文件**
  - 限制信息性消息从虚拟机流向 VMX 文件，从而避免填充数据存储和造成拒绝服务。

7.5 KVM

7.5.1 简介

KVM (Kernel-Based Virtual Machine) 是基于内核的虚拟机，是 Linux 内核的一个可加载模块，通过调用 Linux 本身内核功能，实现对 CPU 的底层虚拟化和内存的虚拟化，使 Linux 内核成为虚拟化层。

KVM 必须依赖 CPU 提供的硬件虚拟化技术，以 Intel、AMD 为代表的硬件平台中很多虚拟化相关的硬件特性。

在普通的 Linux 系统中，进程一般有两种执行模式：内核模式和用户模式，在 KVM 环境中，增加了第三种模式：客户模式。KVM 本身并不提供任何虚拟仿真功能，它仅仅是简单的提供的一个接口，接口位于 `/dev/kvm`，用户空间通过 ioctl 使用此接口实现相关功能。

- 用户模式 (User Mode)：主要处理 I/O 的模拟和管理，由 QEMU 实现。
- 内核模式 (Kernel Mode)：主要处理特别需要高性能和安全相关的指令，如处理客户模式到内核模式的转换，处理客户模式下的 I/O 指令或其他特权指令引起的退出 (VM-Exit)，处理影子内存管理 (shadow MMU)。
- 客户模式 (Guest Mode)：主要执行 guest OS 中的大部分指令，I/O 和一些特权指令除外。

历史

KVM 虚拟机最初是由一个名为 Qumranet 的以色列创业公司作为他们的 VDI 产品的虚拟机开发的。

KVM 的开发人员并没有选择从底层开发 Hypervisor，而是选择了基于 Linux Kernel，通过加载新的模块从而使 Linux Kernel 本身变成一个 Hypervisor。

2006 年 8 月，在先后完成了基本功能、动态迁移以及主要的性能优化之后，Qumranet 正式对外宣布了 KVM 的诞生并推向 Linux 内核社区。同年 10 月，KVM 模块的源代码被正式接纳进入 Linux Kernel，成为内核源代码的一部分。之后 2007 年 2 月发布的 Linux 2.6.20 是第一个带有 KVM 模块的 Linux 内核正式发布版本。

2008 年，Qumranet 被 Red Hat 收购。

7.5.2 KVM 工具集

libvirt

libvirt 是操作和管理 KVM 虚机的虚拟化 API，使用 C 语言编写，可以由不同语言调用。可以操作包括 KVM、vmware、XEN、Hyper-v、LXC 等 Hypervisor。

7.5.3 参考链接

- Kernelgo KVM 学习笔记

7.6 Xen

7.6.1 历史

20 世纪 90 年代，伦敦剑桥大学的 Ian Pratt 和 Keir Fraser 在一个叫做 Xenoserver 的研究项目中，开发了 Xen 虚拟机。在这个时候还不具备对虚拟化技术的硬件支持，内核必须针对 Xen 做出特殊的修改才可以运行。

2002 年 Xen 正式被开源。在先后推出了 1.0 和 2.0 版本之后，Xen 开始被诸如 Redhat、Novell 和 Sun 的 Linux 发行版集成，作为其中的虚拟化解决方案。

2005 年发布的 Xen 3.0，开始正式支持 Intel 的 VT 技术和 IA64 架构，之后 Xen 虚拟机可以运行完全没有修改的操作系统。

7.6.2 Xen Hypervisor

Xen Hypervisor 是直接运行在硬件与所有操作系统之间的基本软件层。它负责为运行在硬件设备上的不同种类的虚拟机进行 CPU 调度和内存分配。Xen Hypervisor 对虚拟机来说不单单是硬件的抽象接口，同时也控制虚拟机的执行，让他们之间共享通用的处理环境。

Xen Hypervisor 不负责处理诸如网络、外部存储设备、视频或其他通用的 I/O 处理。

Xen Hypervisor 的管理接口可通过 Libxenctrl 库调用，来实施管理功能。

7.7 QEMU

7.7.1 背景

QEMU 是一款开源的仿真执行工具，可以模拟运行多种 CPU 架构的程序或系统。在执行时，QEMU 会先以基本块（Basic Block）为单位，将目标指令经由 TCG 这一层翻译成宿主机的代码，得到 TB(Translation Block)，最终在主机上执行。

和虚拟化不同的是，仿真基本都是软件实现，宿主机和虚拟机可以是不同的架构。

7.7.2 架构

QEMU 支持两种操作模式：用户模式仿真和系统模式仿真。用户模式仿真允许一种架构的 CPU 构建的程序在另一种架构的 CPU 上执行，动态翻译/转换相应的指令/系统调用等。如果在相同架构的系统上进行用户模式仿真，可以实现近似本地的性能。

系统模式仿真允许对整个系统进行仿真，包括处理器和配套的外围设备。

QEMU 的优势在于其快速、可移植的动态翻译程序。动态翻译程序允许在运行时将用于目标（Guest）CPU 的指令转换为用于主机 CPU，从而实现仿真。

QEMU 实现动态翻译的方法是，首先将目标指令转换为微操作。这些微操作是一些编译成对象的 C 代码。然后构建核心翻译程序。它将目标指令映射到微操作以进行动态翻译。这不仅可产生高效率，而且还可以移植。

QEMU 的动态翻译程序还缓存了翻译后的代码块，使翻译程序的内存开销最小化。当初次使用目标代码块时，翻译该块并将其存储为翻译后的代码块。QEMU 将最近使用的翻译后的代码块缓存在一个 16 MB 的块中。QEMU 甚至可以通过在缓存中将翻译后的代码块变为无效来支持代码的自我修改。

7.7.3 事件处理机制

简介

QEMU 是事件处理设计架构，所有的事件都在事件循环中被处理。实现基础就是 Glib 事件循环。Glib 的核心是 poll 机制，通过 poll 检查用户注册的事件源，并执行对应的回调。

主事件循环

QEMU 的主事件循环是 `main_loop_wait()`，它执行以下任务：

- 等待文件描述符变得可读或可写
- 运行到期的计时器，计时器使用 `qemu_mod_timer()` 函数添加
- 运行 bottom-halves

状态机

Glib 对一个事件源的处理分为 4 个阶段：初始化、准备、poll 和调度。每个阶段都提供了接口供用户注册处理函数，分别为 prepare、query、check、dispatch，函数在事件后被调用。

7.7.4 iothread

简介

在早期的 qemu 版本，只存在一个主线程，同时负载客户虚拟机的指令执行和运行事件循两个任务。

线程执行客户机指令时，通过异常产生和信号量机制收走 qemu 线程控制权。接着通过执行非阻塞的 select 进行一次循环的迭代，之后就返回客户机指令的执行，并不停重复以上过程直到 QEMU 关闭。

这样的架构被称为 non-iothread 架构，这种架构存在着诸多问题，例如不能利用宿主机的多核能力、在运行 SMP 客户机的情况下会表现不佳、无法同时异步执行多个事件处理等。

之后 QEMU 在新版本中使用了新的架构，为每一个 vCPU 分配一个 QEMU 线程，以及一个专用的事件处理循环线程，这个模型被称为 iothread。

在 iothread 架构中，各个 vCPU 线程可以并行的执行客户机指令，进而提供真正的 SMP 支持；iothread 则负责运行事件处理循环。通过使用了一个全局的 mutex 互斥锁来维持线程同步。大多数时间里，vCPU 在运行客户机指令，iothread 则阻塞在 select 中。

这种方式使得 IO 处理能够完全脱离主线程，运行在多个不同的线程中，充分利用现代多核处理器的能力。

7.7.5 外围设备

将 QEMU 作为 PC 系统仿真器使用可提供各种外围设备。需要的标准外围设备包括硬件 Video Graphics Array (VGA) 仿真器、PS/2 鼠标和键盘、电子集成驱动器（Integrated Drive Electronics）硬盘和 CD-ROM 接口，以及软盘仿真。另外，QEMU 包括对 NE2000 Peripheral Controller Interconnect (PCI) 网络适配器、串行端口、大量的声卡和 PCI Universal Host Controller Interface (UHCI) Universal Serial Bus (USB) 控制器（带虚拟 USB 集线器）的仿真。Processor symmetric multiprocessing (SMP) 支持也得到了对 255 个 CPU 的支持。

除了仿真标准 PC 或 ISA PC（不带 PCI 总线）外，QEMU 还可以仿真其他非 PC 硬件，如 ARM Versatile 基线板（使用 926E）和 Malta million instructions per second (MIPS) 板。对于各种其他平台，包括 Power Macintosh G3 (Blue & White) 和 Sun-4u 平台，都能正常工作。

7.7.6 启动流程

KVM 交互

- 获取到 KVM 句柄
- 创建虚拟机，获取到虚拟机句柄
- 为虚拟机映射内存，还有设备/信号处理的初始化
- 创建 vCPU，并为 vCPU 分配内存空间
- 创建 vCPU 个数的线程并运行虚拟机
- 线程进入循环，并捕获虚拟机退出原因，做相应的处理

TCG 模式

- **启动**
  - main
  - cpu_init
  - qemu_init_vcpu
  - qemu_tcg_init_vcpu
  - qemu_tcg_cpu_thread_fn
- **主函数**
  - **cpu_exec 处理中断异常，找到代码翻译块，执行**
    - **tb_find 在 Hash 表中查找，如果找不到则调用 tb_gen_code 创建一个 TB**
      - **tb_gen_code 分配一个新的 TB**

        gen_intermediate_code

        tcg_gen_code 将 TCG 代码转换成主机代码。

        * **cpu_loop_exec_tb**
            - cpu_tb_exec 执行 TB 主机代码
            - tcg_qemu_tb_exec

用户态

用户态的大部分代码都在 linux-user 目录下

- **linux-user/main.c main**
    - **linux-user/linuxload.c loader_exec**
        * **linux-user/elfload.c load_elf_binary**
            - load_elf_image
            - load_elf_interp
    - linux-user/<arch>/cpu_loop.c cpu_loop
    - accel/tcg/cpu-exec.c cpu_exec

7.7.7 QMP

简介

QEMU 对外提供了一个 socket 接口，称为 qemu monitor，通过该接口，可以对虚拟机实例的整个生命周期进行管理，主要有如下功能

- 状态查看、变更
- 设备查看、变更
- 性能查看、限制
- 在线迁移
- 数据备份
- 访问内部操作系统

7.7.8 源码结构

主要文件

- **accel/tcg**
    - **cpu-exec.c**
        * 寻找下一个二进制翻译代码块
        * 如果没有找到就请求得到下一个代码块，并且操作生成的代码块
    - **translate-all.c**
        * cpu_gen_code 函数，初始化真正代码生成
- **hw**
    - 硬件目录

- **主目录**
  - **vl.c**
    - main 函数
    - main_loop 函数，条件判断
    - 最主要的模拟循环，虚拟机环境初始化，和 CPU 的执行
  - **cpu-exec.c**
    - cpu_exec 函数，主要的执行循环
    - 寻找下一个二进制翻译代码块
    - 如果没有找到就请求得到下一个代码块，并且操作生成的代码块
  - **cpus.c**
    - qemu_main_loop_start 函数，分时运行 CPU 核
  - **exec-all.c**
    - **`struct TranslationBlock`**
      - TB（二进制翻译代码块）结构体
  - exec.c
- **target**
  - 不同架构的对应目录
  - 将客户 CPU 架构的 TBs 转化成 TCG 中间代码
  - TCG 前的前端
  - **target/<arch>/translate.c**
    - 将 guest 代码翻译成不同架构的 TCG 操作码
  - **target/<arch>/cpu.h**
    - CPU 状态结构体
- **tcg**
  - **tcg/tcg.c**
    - 主要的 TCG 代码
  - **tcg/<arch>/tcg-target.c**
    - 将 TCG 代码转化生成主机代码
  - TCG 前的后端

7.7.9 安全

相关 CVE

- CVE-2019-6778 tcp_emu 堆溢出
- CVE-2015-7504 pcnet 网卡堆溢出
- CVE-2015-5165 信息泄露漏洞

7.7.10 其他

ioctl

在 QEMU-KVM 中，用户空间的 QEMU 是通过 ioctl 与内核空间的 KVM 模块进行通讯的。

QOM

QOM (QEMU Object Module) 是用 C 语言实现的一种面向对象的编程模型，是设备模拟的基础，QEMU 中所有的设备，包括 CPU、内存、PCIe、外设都是基于 QOM 来实现的。

TCG

TCG 是 Tiny Code Generator 的简称，在 QEMU 中，它将虚拟出来的系统的指令转化成真正硬件支持的指令中的从中间代码到硬件支持的机器代码。

7.7.11 常用选项

用户模式

- `-L` 动态链接库前缀
- `-cpu model` 设置 CPU 模型
- `-E var=value` 设置环境变量
- `-g port` gdb 调试端口

系统模式

- `-net` 设置网络

7.7.12 参考链接

官方材料

- qemu
- qemu wiki
- qemu source code on github
- QEMU Binaries for Windows and QEMU Documentation
- QEMU Doc
- QEMU, a Fast and Portable Dynamic Translator

Blog

- 使用 QEMU 进行系统仿真
- QEMU(1) - QOM
- QEMU Internals
- QEMU Emulator User Documentation

漏洞分析

- QEMU 信息泄露漏洞 CVE-2015-5165 分析及利用
- qemu-pwn-cve-2019-6778 堆溢出漏洞分析
- qemu-pwn-cve-2015-7504 堆溢出漏洞分析
- VM escape - QEMU Case Study
- Debugee in QEMU

漏洞利用

- qemu-vm-escape exploit for CVE-2019-6778

7.8 Unicorn

7.8.1 背景

Unicorn 基于 QEMU，它提取了 QEMU 中与 CPU 模拟相关的核心代码，并在外层进行了包装，提供了多种语言的 API 接口。

7.8.2 参考链接

- unicorn at github
- unicorn

7.9 Intel 虚拟化技术

7.9.1 CPU 虚拟化技术 VT-X

Intel 增加了虚拟机扩展指令集 VMX(Virtual Machine Extensions)，该指令集包含了十条左右新增指令来支持与虚拟机向相关的操作，简称 Intel VT-x 技术。

VT-x 引入了根模式和非根模式两种操作模式，统称为 VMX 操作模式。

根模式是 VMM 所处的模式。所有指令都可运行，行为与正常 IA32 一样。兼容所有原有软件。非根模式是客户机所处模式。所有敏感指令都会被重定义，使得他们不经虚拟化就直接运行或者通过陷入再模拟的方式处理。

VT-x 的 VMCS 能更好的支持虚拟化，VMCS 保存在本地内存中的数据结构，包括

- vCPU 标识信息：标识 vCPU 属性
- 虚拟寄存器信息：上下文环境保存
- vCPU 状态细信息
- 额外寄存器/部件信息
- 其他信息：优化字段等

每个 VMCS 对应一个 vCPU，CPU 每发生 VM-Exit 和 VM-Entry 时会自动查询和更新 VMCS。VMM 也可以通过指令来配置 VMCS 而影响 vCPU。

7.9.2 扩展页表

EPT(Extended Page Table) 扩展页表用于搭配 VT-x 使用。

EPT 页表存放在 VMM 内核空间，由 VMM 维护，当一个逻辑 CPU 处于非根模式下运行客户机代码时，使用的地址是客户机虚拟地址，而访问这个虚拟地址时，同样会发生地址的转换，这里的转换还没有设计到 VMM 层，和正常的系统一样，这里依然是采用 CR3 作为基址，利用客户机页表进行地址转换，只是到这里虽然已经转换成物理地址，但是由于是客户机物理地址，不等同于宿主机的物理地址，所以并不能直接访问，需要借助于第二次的转换，也就是 EPT 的转换。

不管是 32 位客户机还是 64 位客户机，统一按照 64 位物理地址来寻址。EPT 页表是 4 级页表，页表的大小仍然是一个页即 4KB，但是一个表项是 8 个字节，所以一张表只能容纳 512 个表项，需要 9 位来定位具体的表项。

7.9.3 CPU 虚拟化技术 VT-d

Intel VT-d 技术通过在北桥 MCH 引入 DMA 重映射硬件，以提供设备重映射和设备直接分配的功能。在启动 VT-d 的平台上，设备所有的 DMA 传输都会被 DMA 重映射硬件截获。根据设备对应的 IO 页表，硬件可以对 DMA 中的地址进行转换，使设备只能访问到规定的内存。

7.10 其他

7.10.1 Wine

Wine (Wine Is Not an Emulator) 是一个能够在多种 POSIX-compliant 操作系统 (诸如 Linux，macOS 及 BSD 等) 上运行 Windows 应用的兼容层。Wine 不是像虚拟机或者模拟器一样模仿内部的 Windows 逻辑，而是將 Windows API 调用翻译成为动态的 POSIX 调用，免除了性能和其他一些行为的内存占用，让你能够干净地集合 Windows 应用到你的桌面。

7.10.2 参考链接

- Wine

CHAPTER 8

逆向工程

8.1 ELF

8.1.1 概述

ELF(Executable and Linking Format) 是一种对象文件的格式，用于定义不同类型的对象文件 (Object files) 中存放的对象、以及存放对象的格式。最初 ELF 由 UNIX 系统实验室（USL）开发和发布，作为应用程序二进制接口（ABI）的一部分。

ELF 标准旨在通过为开发人员提供一套简化的软件开发流程二进制接口定义，可以在多种环境下运行。

8.1.2 对象文件

对象文件 (Object files) 有如下三个种类

可重定位的对象文件 (Relocatable file)

这是由汇编器汇编生成的.o 文件。后面的链接器 (link editor) 拿一个或一些 Relocatable object files 作为输入，经链接处理后，生成一个可执行的对象文件 (Executable file) 或者一个可被共享的对象文件 (Shared object file)。

可执行的对象文件 (Executable file)

在 Linux 系统里面，存在两种可执行的东西。除了 Executable file，另外一种就是可执行的脚本 (如 shell 脚本)。注意这些脚本不是 Executable file，它们只是文本文件，但是执行这些脚本所用的解释器就是 Executable file，比如 bash shell 程序。

可被共享的对象文件 (Shared object file)

这些就是所谓的动态库文件，也即.so 文件。如果拿前面的静态库来生成可执行程序，那每个生成的可执行程序中都会有一份库代码的拷贝。如果在磁盘中存储这些可执行程序，那就会占用额外的磁盘空间；另外如果拿它们放到 Linux 系统上一起运行，也会浪费掉宝贵的物理内存。如果将静态库换成动态库，那么这些问题都不会出现。

8.1.3 ELF 格式

ELF 格式有两种

8.1.4 段

.bss

bss 段保存未初始化的全局变量和局部静态变量

.data and .data1

data 段保存已初始化的全局变量和局部静态变量

.rodata

存放只读数据

.text

程序代码段

.comment

存放编译器版本信息等

.debug

存放调试信息

.dynamic

该段中保存了动态链接器所需要的基本信息，是一个结构数组，可以看做动态链接下 ELF 文件的文件头。存储了动态链接会用到的各个表的位置等信息。

.dynstr

该段是.dynsym 段的辅助段

.dynsym

该段与 `.symtab` 段类似，但只保存了与动态链接相关的符号

.got

程序全局入口表

.plt

动态链接的跳转表

.hash

在动态链接下，需要在程序运行时查找符号，为了加快符号查找过程，增加了辅助的符号哈希表

.init

程序初始化代码段

.fini

程序终结代码段

.interp

该段里保存的是一个字符串，这个字符串就是可执行文件所需要的动态链接器的位置

8.1.5 内存布局

```
-----------------------------------
|        Kernel Space             |    1GB
-----------------------------------
|                                 |    Random Stack offset
-----------------------------------
|       Stack (growth down)       |    RLIMIT_STACK
-----------------------------------
|                                 |    Random mmap offset
-----------------------------------
|       Memory Map Segement       |
|  (including dynamic libraries)  |
|      e.g. /lib/libc.so          |
|         (growth down)           |
-----------------------------------
|                                 |
-----------------------------------
|        Heap (growth up)         |
-----------------------------------
|                                 |  Random brk offset
-----------------------------------
|          BSS Segement           |
| Uninitalized static variables,  |
|       filled with zeros.        |
|  e.g. static char *userName;    |
```

(下页继续)

(续上页)

```
---------------------------------
|          Data Segement         |
|  Static variables initialized  |
|    by the programmer.          |
---------------------------------
|          Text Segement         |
---------------------------------
```

8.1.6 PLT 与 GOT

在编译和链接阶段，链接器无法知知道进程运行起来之后外部函数的加载地址。因此程序编译时会采用两种表进行辅助，一个为程序链接表（PLT，Procedure Link Table），一个为全局偏移表（GOT, Global Offset Table）。

8.2 PE

8.2.1 简介

PE（全称 Portable Executable）格式，是微软 Win32 环境可移植可执行文件 (如 exe、dll、vxd、sys、vdm 等) 的标准文件格式。

8.2.2 Table

每个 PE 文件在文件最开始的地方都有一个 import 和 export table。import table 包含所有从 dll 中读取的函数，有一个.reloc 段来保存 dll 在内存中信息。

和 linux 不同，pe 会优先加载同一个文件夹内的 dll 而不是去其他地方找。在开发者看来这省去了修改 PATH 的烦恼，避免了 dll-hell，但是在攻击者看来这为一些绕过提供了方便。

8.2.3 RVAs

PE-COFF 的一个比较重要的概念是 Relative Virtual Address(RVAs) RVAs 是用来减少一些 PE Loader 需要做的工作的，简单的说，每个 DLL 都被加载到内存中的某个位置，可以把 RVA 加到一个基址上去找想要的东西。

8.2.4 Heaps

在 dll 加载的时候，会调用一些初始函数，这些函数会用 HeapCreate() 来设置其自身的栈，然后用一个全局变量的指针来指向它。

大多数 dll 在内存中也有一个.data 段来记录全局变量，因为有这么多的 heap，堆溢出攻击会有点麻烦。

在 linux 中只会有一个栈溢出，但是在 win 中可能有几个栈同时溢出，这就增加了分析的难度

每个进程都有一个默认的堆，可以用 GetDefaultHeap() 来找到它，尽管这个 heap 并不一定是溢出的那一个。

8.2.5 Thread

windows 没有 fork()，而是用 CreateProcess，这个函数会起一个新的有自己内存空间的进程，子进程会继承父进程所有可继承的属性。

8.2.6 DCOM

Distributed Common Object Model(DCOM)。windows 有一个特点的是分发程序一般都是用 binary，使得几乎所有程序都支持 COM。

COM 可以用 COM 支持的任何一种语言完成，而且可以在其中进行无缝转换要深入的了解 DOM，需要了解 IDL（Interface Description Language）文件。

8.3 DLL

8.3.1 简介

dll 是被映射到 exe 进程的地址空间中去，换句话说，它可以被认为寄生在 exe 之上，自己并不独立存在。而对于需要独立执行的 dll，Windows 使用 svchost.exe、dllhost.exe 和 rundll32.exe 来执行这些 dll。

svchost.exe

官方的解释是 `svchost.exe is a generic host process name for services that run from dynamic-link libraries.`

dllhost.exe

它的存在是为了容纳 COM 组件。

rundll32.exe

rundll 是 Windows 系统自带的一个直接执行 DLL 中导出函数的小工具

8.3.2 ref

https://zhuanlan.zhihu.com/p/30000572

8.4 Mach-O

8.4.1 参考链接

- Overview of the Mach-O Executable Format
- Mach Object Files

CHAPTER 9

漏洞利用基础

9.1 Shellcode

9.1.1 技巧

在写 shellcode 的时候，可能会出现 x00 的情况，这就需要一定的技巧一个方法是直接替换掉含 null 的指令。另外一种方法是用 xor inc 等运算把 0 凑出来。

9.1.2 常见 Shellcode 功能

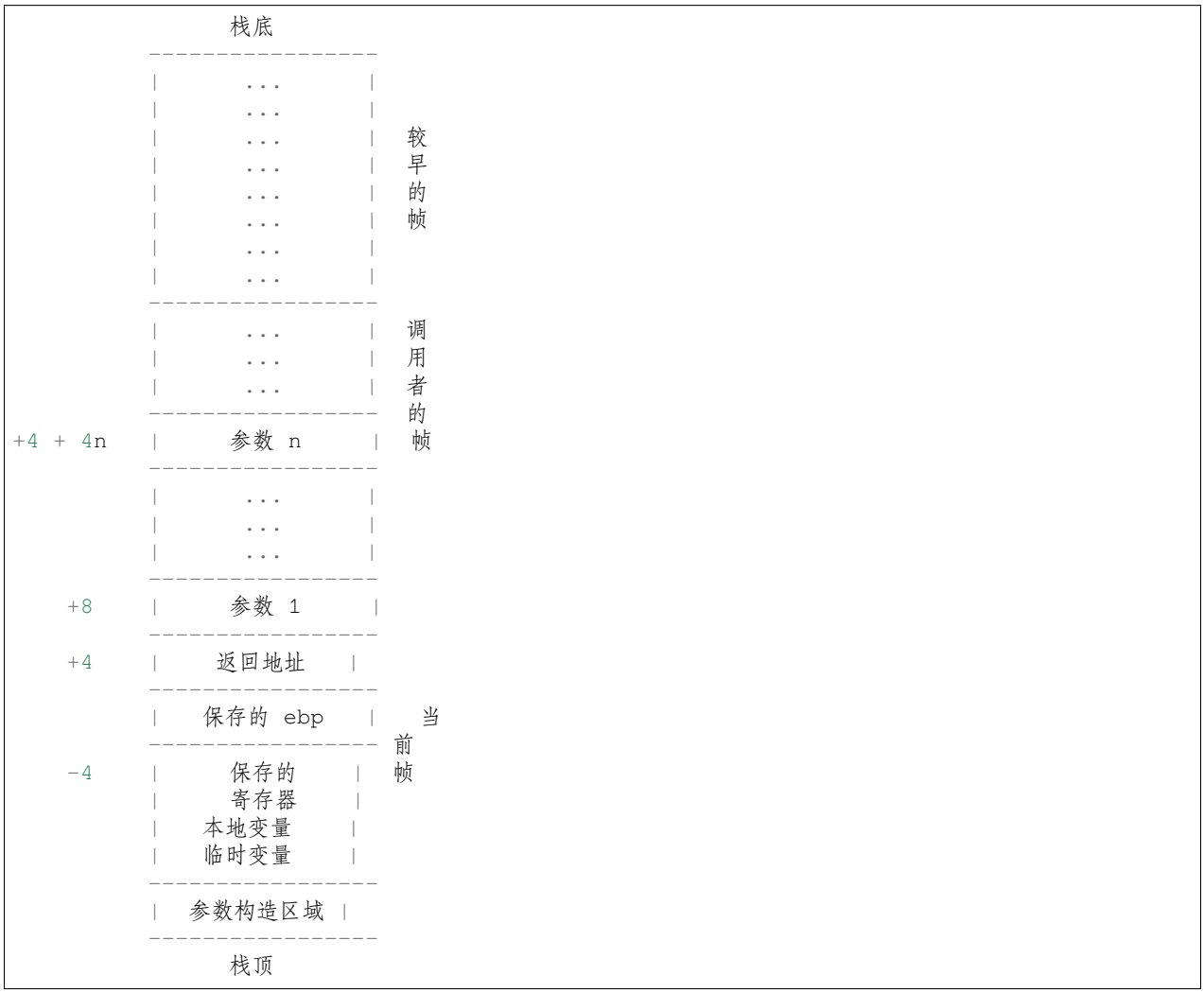
- **Unix**
  - execve /bin/sh
  - port-binding
  - reverse shell
  - setuid
  - breaking chroot
- **Windows**
  - WinExec
  - Reverse shell using CreateProcess cmd.exe

9.2 Stack

9.2.1 函数调用

- 参数入栈，将参数从右到左压入系统栈
- 返回地址入栈，将当前代码区调用指令的下一条指令压入栈中，供函数返回时执行
- 代码区跳转，处理区从当前代码区跳转到被调用函数的入口
- **栈帧调整**
  - 保存当前栈帧，EBP 入栈
  - 切换到新栈帧，ESP 装入 EBP
  - 给新栈帧分配空间，ESP 减去所需的空间，抬高栈顶

9.2.2 栈结构



9.3 Heap

9.3.1 简介

目前各平台有 dlmalloc、ptmalloc2、jemalloc、tcmalloc、libumem 等堆内存管理机制。

ptmalloc2 来自于 dlmalloc 的分支。此后，添加线程支持并于 2006 年发布。正式发布后，patmalloc2 集成到 glibc 源码中。随着源码集成，代码修改便直接在 glibc malloc 源码里进行。因此 ptmalloc2 与 glibc 之间的 malloc 实现有很多不同。

在早期的 Linux 里，dlmalloc 被用做默认的内存分配器。但之后因为 ptmalloc2 添加了线程支持，ptmalloc2 成为了 Linux 默认内存分配器。线程支持可帮助提升内存分配器以及应用程序的性能。

在 dlmalloc 里，当两个线程同时调用 malloc 时，只有一个线程能进入到临界段，因为这里的空闲列表数据结构是所有可用线程共用的。因此内存分配器要在多线程应用里耗费时间，从而导致性能降低。

然而在 ptmalloc2 里，当两个线程同时调用 malloc 时，会立即分配内存。因为每个线程维护一个单独的堆分段，因此空闲列表数据结构正在维护的这些堆也是独立的。这种维护独立堆以及每一个线程享有空闲列表数据结构的行为被称为 Per Thread Arena。

在下文中，主要对 glibc 使用的 ptmalloc2 进行介绍。

9.3.2 Chunk

在内存中，堆（低地址到高地址，属性 RW）有两种分配方式（与 malloc 申请 chunk 做区分）：

- mmap: 当申请的 size 大于 128kb 时，由 mmap 分配
- brk: 当申请的 size 小于 128kb 时，由 brk 分配，第一次分配 132KB（main arena），第二次在 brk 下分配，不够则执行系统调用，向系统申请

在内存中进行堆的管理时，系统基本是以 chunk 作为基本单位，chunk 的结构在源码中有定义

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

chunk 的状态可以分为三种：allocated chunk、free chunk、top chunk

9.3.3 Bins

Bin 在内存中用来管理 free chunk，bin 为带有头结点（链表头部不是 chunk）的链表数组，根据特点，将 bin 分为四种，分别为：fastbin、unsortedbin、smallbin、largebin。

- **fastbin**
  - 根据 chunk 大小维护多个单向链表
  - sizeof(chunk) < 64(bytes)
  - 下一 chunk（内存中）的 free 标志位不取消，显示其仍在使用
  - 后进先出（类似栈），先 free 的先被 malloc
  - 拥有维护固定大小 chunk 的 10 个链表
- **unsortedbin**
  - 双向循环链表
  - 不排序
  - 暂时存储 free 后的 chunk，一段时间后将 chunk 放入对应的 bin 中
  - 只有一个链表
- **smallbin**
  - 双向循环链表
  - sizeof(chunk) < 512 (bytes)
  - 先进先出（类似队列）
  - 16,24,...,64,72,...,508 bytes(62 个链表)
- **largebin**
  - 双向循环链表
  - sizeof(chunk) >= 512 (bytes)
  - free chunk 中多两个指针分别指向前后的 large chunk
  - **63 个链表**
    - 0-31(512+64*i)
    - 32-48(2496+512*i)
    - ...
  - 链表中 chunk 大小不固定，先大后小

9.3.4 Malloc 逻辑

当接收到申请内存的请求时，malloc 的处理逻辑如下

- **申请长度位于 fastbin 时**
  - 根据大小获得 fastbin 的 index
  - **根据 index 获取 fastbin 中链表的头指针**
    - 如果头指针为 NULL，转去 smallbin
  - 将头指针的下一个 chunk 地址作为链表头指针

  - 分配的 chunk 保持 inuse 状态，避免被合并
  - 返回除去 chunk_header 的地址
- **长度位于 smallbin 时**
  - 根据大小获得 smallbin 的 index
  - 根据 index 获取 smallbin 中双向循环链表的头指针
  - **将链表最后一个 chunk 赋值给 victim**
    * 若 victim 为表头，此时链表为空，不从 smallbin 中分配
    * 若 victim 为 0，此时链表未初始化，将 fastbin 中的 chunk 合并
    * 其他情况取出 victim, 设置 inuse
  - 检查 victim 是否为 main_arena, 设置标志位
  - 返回除去 chunk_header 的地址
- **长度位于 largebin 时**
  - 根据大小获得 largebin 的 index
  - 将 fastbin 中 chunk 合并，加入到 unsortbin 中

进一步处理

- **unsortedbin**
  - 反向遍历 unsortedbin, 检查 2*size_t<chunk_size< 内存总分配量
  - **unsortedbin 的特殊分配**
    * 如果前一步 smallbin 分配未完成
    * 并且 unsortedbin 中只有一个 chunk
    * 并且该 chunk 为 last remainder chunk
    * 并且该 chunk 大小 >（所需大小 + 最小分配大小）
    * 则切分一块分配
  - 如果请求大小正好等于当前遍历 chunk 的大小，则直接分配
  - 继续遍历，将合适大小的 chunk 加入到 smallbin 中，向前插入作为链表的第一个 chunk。(smallbin 中每个链表中 chunk 大小相同)
  - 将合适大小的 chunk 加入到 largebin 中，插入到合适的位置（largebin 中每个链表 chunk 由大到小排列）
- **largebin**
  - **反向遍历 largebin，由下到上查找，找到合适大小后切分** 切分后大小 < 最小分配大小，返回整个 chunk，会略大于申请大小切分后大小 > 最小分配大小，加入 unsortedbin。
  - 未找到，index+1，继续寻找

如果这之后还未找到合适的 chunk，就会使用 top chunk 进行分配，还是没有的话，如果在多线程环境中，fastbin 可能会有新的 chunk，再次执行合并，并向 unsortedbin 中重复上面的步骤，之后还是没有的话，就只能向系统申请。

以上为 malloc 分配的经过

malloc 检查

- 从 fastbin 中取出 chunk 后，检查 size 是否属于 fastbin
- 从 smallbin 中除去 chunk 后，检查 victim->bk->fd == victim
- 从 unsortbin 取 chunk 时，要检查 2*size_t<chunk_size< 内存总分配量
- 从 largebin 取 chunk 时，切分后的 chunk 要加入 unsortedbin, 需要检查 unsortedbin 的第一个 chunk 的 bk 是否指向 unsortedbin

9.3.5 free 机制

- 使用 chunksize(p) 宏获取 p 的 size
- **安全检查**
  - chunk 的指针地址不能溢出
  - chunk 的大小 >= MINSIZE(最小分配大小)，并且检查地址是否对齐
- **大小为 fastbin 时**
  - 检查下一个 chunk 的 size：2*size_t<chunk_size< 内存总分配量
  - double free 检查：检查当前 free 的 chunk 是否与 fastbin 中的第一个 chunk 相同，相同则报错

9.3.6 其他情况

- **检查下一个 chunk 的 size**
  - 2*size_t<chunk_size< 内存总分配量
  - 如果当前 chunk 为 sbrk() 分配，那么它相邻的下一块 chunk 超过了分配区的地址，会报错
- **double free 检查**
  - 检查当前 free 的 chunk 是否为 top chunk，是则报错
  - 根据下一块的 inuse 标识检查当前 free 的 chunk 是否已被 free
- **unlink 合并**
  - 检查前后 chunk 是否 free，然后向后（top chunk 方向）合并，并改变对应的 inuse 标志位
  - **unlink 检查**
    * I. 当前 chunk 的 size 是否等于下一 chunk 的 prev_size
    * II.P->bk->fd == P && P->bk->fd == P
  - 如果合并后 chunk_size > 64bytes, 则调用函数合并 fastbin 中的 chunk 到 unsortedbin 中
  - 将合并后的 chunk 加入 unsortedbin
- **unsortedbin 检查**
  - 检查 unsortedbin 的第一个 chunk 的 bk 是否指向 unsortedbin

9.3.7 References

- 深入理解
- malloc homepage
- Memory Allocator
- Dance in Heap 1
- Linux 堆管理分析
- Understanding glibc malloc
- syscalls-used-by-malloc

CHAPTER 10

栈相关漏洞

10.1 调用机制

10.1.1 调用惯例

- **cdecl**
  - 函数调用方出栈
  - 传参从右至左入栈
  - 名称修饰使用下划线 + 函数名
- **stdcall**
  - 函数本身出栈
  - 传参从右至左入栈
  - 名称修饰下划线 + 函数名 +@+ 函数字节数
- **fastcall**
  - 函数本身出栈
  - 传参头两个 DWORD 类型或者占更少字节的参数放入寄存器，其他的从右到左压栈
  - 名称修饰 @+ 函数名 +@+ 参数的字节数
- **pascal**
  - 函数本身出栈
  - 传参从左至右入栈
  - 名称修饰复杂
- **naked call**
  - 在特殊场合使用，不保护寄存器

- **thiscall**
  - C++ 的特殊调用管理，称为 thiscall，用于类成员函数的调用，特点随编译器不同而不同
  - VC 的 thiscall this 指针存放于 ecx，参数从右至左压栈
  - gcc 的 thiscall 和 cdecl 一样，只是将 this 看做函数的第一个参数

10.2 ROP

Return Orientated Programming

10.2.1 参考链接

- ROP

10.3 栈溢出

TODO

CHAPTER 11

堆相关漏洞

11.1 Use After Free

11.1.1 概述

基本概念

UAF 的产生涉及到两个事件，产生和使用悬垂指针。

当所指向的对象被释放或者收回，但是对该指针没有作任何的修改，以至于该指针仍旧指向已经回收的内存地址，此情况下该指针便称悬垂指针（也叫迷途指针，Dangling pointer）。

某些编程语言允许未初始化的指针的存在，而这类指针即为野指针（Wild pointer）。

成因

在许多编程语言中（比如 C），显示地从内存中删除一个对象或者返回时通过销毁栈帧，并不会改变相关的指针的值。该指针仍旧指向内存中相同的位置，即使引用已经被删除，现在可能已经挪作他用。

举例来说

```
char* a = malloc(512);
char* b = malloc(256);
char* c;
strcpy(a, "A here");
free(a);
c = malloc(500);
strcpy(c, "C here");
```

因为 glibc 使用首次适应（FirstFit）来选取合适的 chunk，当 a 被 free 后，调用相近大小的 malloc，则会分配 a 的地址。此时没有设置 a 的地址为 `null` ，则可以通过 a 来访问和修改 c 指向的空间的内容。

危害

可以概括为三方面：

- 破坏完整性：例如对已经释放的内存进行的操作可能破坏正常的数据
- 破坏可用性：例如内存块释放后恰好发生了内存块合并，一些进程使用非法数据作为内存块就会崩溃
- 破坏访问控制：利用 UAF 读取私密信息

11.2 Off By One

一个字节溢出被称为 off-by-one，曾经的一段时间里，off-by-one 被认为是不可以利用的，但是后来研究发现在堆上哪怕只有一个字节的溢出也会导致任意代码的执行。同时堆的 off-by-one 利用也出现在国内外的各类 CTF 竞赛中。

11.2.1 参考链接

- Linux 下堆漏洞利用 off by one

11.3 堆溢出

11.3.1 第一部分

首先简要介绍一下堆 chunk 的结构在 malloc.c 中找到关于堆 chunk 结构的代码

```
struct malloc_chunk {

  INTERNAL_SIZE_T prev_size;
  /* Size of previous chunk (if free).  */

  INTERNAL_SIZE_T size;
  /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;
  /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize;
  /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

这指明了一个 heap chunk 是如下的结构

```
+-----------+---------+------+------+-------------+
|           |         |      |      |             |
|           |         |      |      |             |
| prev_size |size&Flag|  fd  |  bk  |             |
|           |         |      |      |             |
|           |         |      |      |             |
+-----------+---------+------+------+-------------+
```

如果本 chunk 前面的 chunk 是空闲的，那么第一部分 prev_size 会记录前面一个 chunk 的大小，第二部分是本 chunk 的 size, 因为它的大小需要 8 字节对齐，所以 size 的低三位一定会空闲出来，这时候这三个位置就用作三个 Flag(最低位: 指示前一个 chunk 是否正在使用; 倒数第二位: 指示这个 chunk 是否是通过 mmap 方式产生的; 倒数第三位: 这个 chunk 是否属于一个线程的 arena)。之后的 FD 和 BK 部分在此 chunk 是空闲状态时会发挥作用。FD 指向下一个空闲的 chunk，BK 指向前一个空闲的 chunk，由此串联成为一个空闲 chunk 的双向链表。如果不是空闲的。那么从 fd 开始，就是用户数据了。(详细信息请参考 glibc 的 malloc.c 部分，在此不再多做解释。)

引用一位外国博主的漏洞示例程序解释

```
/*
 Heap overflow vulnerable program.
 */
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
    char * first, * second;

    /*[1]*/
    first = malloc( 666 );
    /*[2]*/
    second = malloc( 12 );
    /*[3]*/
    if(argc!=1)
        strcpy( first, argv[1] );
    /*[4]*/
    free( first );
    /*[5]*/
    free( second );
    /*[6]*/
    return( 0 );
}
```

这个程序在 [3] 处有很明显的堆溢出漏洞，argv[1] 中的内容若过长则会越界覆盖到 second 部分。

程序堆结构如下

```
+---------------------+   <--first chunk ptr
|     prev_size       |
+---------------------+
|     size=0x201      |
+---------------------+   <--first
|                     |
|     allocated       |
|      chunk          |
+---------------------+   <--second chunk ptr
|    prev_size        |
+---------------------+
|    size=0x11        |
+---------------------+   <--second
|     Allocated       |
|       chunk         |
+---------------------+   <-- top
|     prev_size       |
+---------------------+
|    size=0x205d1     |
```

(下页继续)

(续上页)

```
+---------------------+
|                     |
|                     |
|                     |
|         TOP         |
|                     |
|       CHUNK         |
|                     |
+---------------------+
```

此时利用方法为通过溢出构造，使得 second chunk

```
prev_size= 任意值
size=-4(因为最低位的 flag 没有设置，所以 prev_size 可以任意)
fd=free@got-12
bk=shellcode 地址
```

在我们的 payload 将指定位置的数值改好后。下面介绍在 [4][5] 行代码执行时发生的详细情况。

第四行执行 free(first) 发生如下操作

1). 检查是否可以向后合并

首先需要检查 previous chunk 是否是空闲的（通过当前 chunk size 部分中的 flag 最低位去判断），当然在这个例子中，前一个 chunk 是正在使用的，不满足向后合并的条件。

2). 检查是否可以向前合并

在这里需要检查 next chunk 是否是空闲的 (通过下下个 chunk 的 flag 的最低位去判断)，在找下下个 chunk(这里的下、包括下下都是相对于 chunk first 而言的) 的过程中，首先当前 chunk+ 当前 size 可以引导到下个 chunk，然后从下个 chunk 的开头加上下个 chunk 的 size 就可以引导到下下个 chunk。但是我们已经把下个 chunk 的 size 覆盖为了-4，那么它会认为下个 chunk 从 prev_size 开始就是下下个 chunk 了，既然已经找到了下下个 chunk，那就就要去看看 size 的最低位以确定下个 chunk 是否在使用，当然这个 size 是-4，所以它指示下个 chunk 是空闲的。

在这个时候，就要发生向前合并了。即 first chunk 会和 first chunk 的下个 chunk(即 second chunk) 发生合并。在此时会触发 unlink(second) 宏，想将 second 从它所在的 bin list 中解引用。

但是执行 unlink 宏之后，再调用 free 其实就是调用 shellcode，这时就可以执行任意命令了。

11.3.2 第二部分

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);
    else {
        FD->bk = BK;
        BK->fd = FD;
        if (!in_smallbin_range (P->size)
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {
            if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)
                || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))
              malloc_printerr (check_action,
                               "corrupted double-linked list (not small)",
```

(下页继续)

(续上页)

```
                                P, AV);
            if (FD->fd_nextsize == NULL) {
                if (P->fd_nextsize == P)
                  FD->fd_nextsize = FD->bk_nextsize = FD;
                else {
                    FD->fd_nextsize = P->fd_nextsize;
                    FD->bk_nextsize = P->bk_nextsize;
                    P->fd_nextsize->bk_nextsize = FD;
                    P->bk_nextsize->fd_nextsize = FD;
                  }
              } else {
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;
              }
          }
      }
}
```

11.3.3 参考链接

- 堆溢出的 unlink 利用方法
- Double Free 浅析

CHAPTER 12

其他漏洞

12.1 格式化字符串

12.1.1 概述

类 printf 函数的最大的特点就是，在函数定义的时候无法知道函数实参的数目和类型。对于这种情况，可以使用省略号指定参数表。

带有省略号的函数定义中，参数表分为两部分，前半部分是确定个数、确定类型的参数，第二部分就是省略号，代表数目和类型都不确定的参数表，省略号参数表中参数的个数和参数的类型是事先的约定计算出来的，每个实参的地址（指针）是根据确定参数表中最后一个实参的地址算出来的。

这里涉及到函数调用时的栈操作。函数栈的栈底是高地址，栈顶是低地址。在函数调用时函数实参是从最后一个参数（最右边的参数）到第一个参数（最左边的参数）依次被压入栈顶方向。也就是说函数调用时，函数实参的地址是相连的，并且从左到右地址是依次增加的。

12.1.2 攻击原理

因为类 printf 函数中省略号参数表中参数的个数和类型都是由类 printf 函数中的那个格式化字符串来决定的，所以攻击者可以利用编程者的疏忽或漏洞，巧妙构造格式化字符串，达到攻击目的。

举一个简单的例子，如果想输出一个字符串，可以使用 `printf("%s", str)` 或者 `printf(str)` 。而第二种写法是有漏洞的，如果攻击者输入 `%d %x` 等格式化字符，那么一个变量的参数值就从堆栈中取出。

12.1.3 常用字符

- **%c**
  - 输出字符，配上 %n 可用于向指定地址写数据
- **%d** 输出十进制整数，配上 %n 可用于向指定地址写数据
- **%x**
  - 输出 16 进制数据
  - %i$x 表示要泄漏偏移 i 处 4 字节长的 16 进制数据
  - %i$lx 表示要泄漏偏移 i 处 8 字节长的 16 进制数据，32bit 和 64bit 环境下一样
- **%p**
  - 输出 16 进制数据，与 %x 基本一样，只是附加了前缀 0x，在 32bit 下输出 4 字节，在 64bit 下输出 8 字节，可通过输出字节的长度来判断目标环境是 32bit 还是 64bit
- **%s**
  - 输出的内容是字符串，即将偏移处指针指向的字符串输出，如 %i$s 表示输出偏移 i 处地址所指向的字符串，在 32bit 和 64bit 环境下一样，可用于读取 GOT 表等信息
- **%n**
  - 将 %n 之前 printf 已经打印的字符个数赋值给偏移处指针所指向的地址位置
  - %100x%10$n 表示将 0x64 写入偏移 10 处保存的指针所指向的地址（4 字节）
  - %$hn 表示写入的地址空间为 2 字节
  - %$hhn 表示写入的地址空间为 1 字节
  - %$lln 表示写入的地址空间为 8 字节，在 32bit 和 64bit 环境下一样
  - 有时，直接写 4 字节会导致程序崩溃或等候时间过长，可以通过 %$hn 或 %$hhn 来适时调整

12.1.4 sinks

- printf
- vprintf
- cprintf
- dprintf
- fprintf
- asprintf
- snprintf
- vdprintf
- vfprintf
- vsprintf
- vasprintf
- vsnprintf

12.2 Type Confusion

12.2.1 简介

类型混淆的问题发生在程序使用一种类型分配或初始化诸如指针，对象或变量之类的资源，但在之后使用与原始类型不兼容的类型访问该资源的情况。

当程序使用不兼容类型访问资源时，可能会触发逻辑错误，因为资源没有预期的属性。在没有内存安全性的语言中，例如 C 和 C++，类型混淆可能导致越界内存访问。

虽然在 C 中使用许多不同的嵌入对象类型解析数据时，这种漏洞经常与 `union` 相关联，但它可以存在于可以以多种方式解释相同变量或内存位置的任何应用程序中。

这个漏洞并不是 C 和 C++ 独有的。PHP 应用程序中的一些漏洞可以通过在期望标量时提供数组参数来触发，反之亦然。像 Perl 这样的语言，当变量被执行自动转换时，也可能包含这些问题。

12.2.2 示例

```
#define NAME_TYPE 1
#define ID_TYPE 2

struct MessageBuffer
{
    int msgType;
    union {
        char *name;
        int nameID;
    };
};


int main (int argc, char **argv) {
    struct MessageBuffer buf;
    char *defaultMessage = "Hello World";

    buf.msgType = NAME_TYPE;
    buf.name = defaultMessage;
    printf("Pointer of buf.name is %p\n", buf.name);
    /* This particular value for nameID is used to make the code architecture-
↪independent. If coming from untrusted input, it could be any value. */
    buf.nameID = (int)(defaultMessage + 1);
    printf("Pointer of buf.name is now %p\n", buf.name);
    if (buf.msgType == NAME_TYPE) {
        printf("Message: %s\n", buf.name);
    }
    else {
        printf("Message: Use ID %d\n", buf.nameID);
    }
}
```

CHAPTER 13

恶意软件

13.1 Windows

13.1.1 常见 API

- CreateProcessA() / CreateProcessW()
- CreateProcessAsUserA() / CreateProcessAsUserW()
- CreateProcessInternalA() / CreateProcessInternalW()
- CreateProcessWithLogonW() CreateProcessWithTokenW()
- LoadLibraryA() / LoadLibraryW()
- LoadLibraryExA() / LoadLibraryExW()
- LoadModule()
- LoadPackagedLibrary()
- WinExec()
- ShellExecuteA() / ShellExecuteW()
- ShellExecuteExA() / ShellExecuteExW()

CHAPTER 14

防御策略

14.1 ASLR

14.1.1 简介

地址空间布局随机化（Address Space Layout Randomization, ASLR）是一种针对缓冲区溢出的安全保护技术。其通过对堆、栈、共享库等地址的随机化，防止攻击者直接定位攻击代码位置。达到阻止缓冲区溢出的目的。

14.1.2 爆破

ASLR 只在 exec 时生效，fork 时并不会改变，因此在一些情况下可以爆破。在 32 位的程序中，可随机化的范围为 8bit，在 64 位的程序中，可随机化的范围为 22bit。

14.1.3 关闭

在 Linux 下可以通过 `echo 0 > /proc/sys/kernel/randomize_va_space` 关闭。

14.2 Canary

14.2.1 简介

在发生栈溢出攻击时，通常会修改函数栈，那么通过在缓冲区和控制信息（如 EBP 等）间插入一个 `canary` 。这样，当缓冲区溢出发生时，在返回地址被覆盖之前 `canary` 会首先被覆盖。通过检查 `canary` 的值是否被修改，就可以判断是否发生了溢出。

14.2.2 特点

- 在 fork 时，canary 并不会改变，可以利用这种方式来爆破其值

14.3 CFI

14.3.1 简介

攻击者常常溢出覆盖或者直接篡改寄存器 EIP 的值，篡改间接调用的地址，进而控制了程序的执行流程。

CFI（控制流完整性）是一种漏洞利用的缓解措施。CFI 用于降低攻击的可能性。最简单的理解 CFI 的概念，在运行时强制要求程序按编译时希望的想法去执行。

一个程序的控制流可以用一张图来呈现，这张图被称为控制流图（CFG，control flow graph）。CFG 是一个每个节点都是程序基本块的有向图，图中的每一条有向边都是可能的控制流路径。CFI 则用来保证 CFG 在运行时和编译时相同。

14.3.2 粒度

较细粒度的 CFI 通过一些方法严格控制每一个间接跳转指令的转移目标。对于由间接跳转指令或者间接分支跳转指令引起的前向控制流，CFI 根据 CFG 在每一个目标地址前插入标记，在控制流转移前插入检查标记的指令，如果检查成功，则控制流转移合法。

细粒度的 CFI 存在性能开销较大的问题，因此有人提出了粗粒度的 CFI，基于攻击特征做检查。

14.3.3 CFI On Window

执行流保护（CFG，Control Flow Guard）是微软从 Windows 8.1 update 3 及 Windows 10 技术预览版开始，默认启用的一项缓解技术。该技术是对 CGI 的一个实现。

这项技术通过在间接跳转前插入校验代码，检查目标地址的有效性，进而可以阻止执行流跳转到预期之外的地点，最终及时并有效的进行异常处理，避免引发相关的安全问题。

14.3.4 CFG 原理

在编译启用了 CFG 的模块时，编译器会分析出该模块中所有间接函数调用可达的目标地址，并将这一信息保存在 Guard CF Function Table 中。

同时，编译器还会在所有间接函数调用之前插入一段校验代码，以确保调用的目标地址是预期中的地址

操作系统在创建支持 CFG 的进程时，将 CFG Bitmap 映射到其地址空间中，并将其基址保存在 ntdll!LdrSystemDllInitBlock 中。

CFG Bitmap 是记录了所有有效的间接函数调用目标地址的位图，出于效率方面的考虑，平均每 1 位对应 8 个地址（偶数位对应 1 个 0x10 对齐的地址，奇数位对应剩下的 15 个非 0x10 对齐的地址）。

提取目标地址对应位的过程如下：

- 取目标地址的高 24 位作为索引 i
- 将 CFG-Bitmap 当作 32 位整数的数组，用索引 i 取出一个 32 位整数 bits
- 取目标地址的第 4 至 8 位作为偏移量 n
- 如果目标地址不是 0x10 对齐的，则设置 n 的最低位

- 取 32 位整数 bits 的第 n 位即为目标地址的对应位

操作系统在加载支持 CFG 的模块时，根据其 Guard CF Function Table 来更新 CFG Bitmap 中该模块所对应的位。同时，将函数指针 _guard_check_icall_fptr 初始化为指向 ntdll!LdrpValidateUserCallTarget。

ntdll!LdrpValidateUserCallTarget 从 CFG Bitmap 中取出目标地址所对应的位，根据该位是否设置来判断目标地址是否有效。若目标地址有效，则该函数返回进而执行间接函数调用；否则，该函数将抛出异常而终止当前进程。

14.3.5 参考链接

- Let's talk about CFI: clang edition
- 分析及防护 Win10 执行流保护绕过问题
- Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM

14.4 NX

14.4.1 简介

即使在今天，栈溢出也是一个很常见的安全问题。而其最常见的攻击方式则是覆盖返回地址，并跳转至 shellcode。

DEP 通常会去掉堆栈的可执行权限（Non-Executable Stack）并设置代码段为不可写来使得这种 payload 无法生效。

在 DEP 提出之后，也出现了相当多的绕过方式，几乎所有的绕过方式都基于一个事实：即使堆栈被标记为不可执行，但是其他地方仍然可以执行代码，而且，基于覆写返回地址的攻击几乎一直有效。

最开始的绕过方法是 return-into-libc，而后则扩展到 ret2plt, ret2strpy, ret2gets, ret2syscall, ret2data, ret2text, ret2code, ret2dl-resolve 等方法。

- ret2data: 很常见的一种方式，把 shellcode 放到 data 段中，然后跳至 data 段执行
- ret2libc: 栈溢出攻击对栈有完全的控制权，即可以完全覆写返回地址和参数
- ret2strcpy：这个方式也是基于 ret2libc，但是几乎可以执行任意代码。在这里，返回到 strcpy，src 指向栈上的 shellcode，dst 指向选择好的可写可执行的内存地址
- ret2gets：类似 strcpy

14.5 沙箱机制

14.5.1 命名空间

Linux 中支持 namespace（命名空间）机制，这是一种轻量级的虚拟化形式。命名空间建立系统的不同视图，对于每一个命名空间看来，都像是一台独立的 Linux 机器。目前 Linux 支持六种命名空间，分别为：

- mnt (mount points, file systems)
- pid (processes)
- net (network stack)
- ipc (system V IPC)

- uts (host name)
- user (UIDs)

14.5.2 chroot

chroot 是较老的沙箱实现方式，其内部的机制和文件系统的命名空间类似。

14.5.3 参考链接

- Many approaches to sandboxing in Linux

14.6 安全编程

14.6.1 内存安全

- 初始化所有变量
- 保持数组和缓冲区的边界检查
- 检查整数的上溢和下溢
- 使用安全的函数
- 永远不给内存可执行权限

CHAPTER 15

工具与资源

15.1 工具列表

15.1.1 逆向工具

二进制

- capstone
- dnspy .NET debugger and assembly editor
- binary ninja
- HAL The Hardware Analyzer
- LIEF Library to Instrument Executable Formats

PE 工具

- EXEInfoPE
- DetectIt Easy
- StudyPE

API

- binaryninja api Public API, examples, documentation and issues for Binary Ninja

Bytecode

- bytecode viewer A Java 8+ Jar & Android APK Reverse Engineering Suite (Decompiler, Editor, Debugger & More)
- pycdc C++ python bytecode disassembler and decompiler

IDA

文档与资料

- IDA Tutorials
- IDA SDK
- idapython cheatsheet
- awesome ida

辅助工具

- HexRaysPyTools Find code patterns within the Hexrays AST

插件

- IDArling
- abyss IDAPython Plugin for Postprocessing of Hexrays Decompiler Output
- Sark IDA Plugins & IDAPython Scripting Library
- IDA minsc is a plugin for IDA Pro that assists a user with scripting the IDAPython plugin that is bundled with the disassembler
- lucid An Interactive Hex-Rays Microcode Explorer
- grap grap: define and match graph patterns within binaries

Golang 插件

- golang loader assist
- IDAGolangHelper Set of IDA Pro scripts for parsing GoLang types information stored in compiled binary
- go parser Yet Another Golang binary parser for IDAPro

Ghidra

文档与资料

- Ghidra
- Ghidra API Overview
- Ghidra Online Courses
- Awesome Ghidra A curated list of awesome Ghidra materials

插件

- Ghidra Cpp Class Analyzer
- GhidraSnippets Python snippets for Ghidra's Program and Decompiler APIs

Radare2

- radare2
- Radare2 Book

Diff

- diaphora
- polypyus

Patch

- e9patch A powerful static binary rewriting tool

文件分析

- oletools python tools to analyze MS OLE2 files and MS Office documents

加壳

- UPX the Ultimate Packer for eXecutables

15.1.2 动态分析

动态插桩

- DynamoRIO Dynamic Instrumentation Tool Platform
- pintools Pintool example and PoC for dynamic binary analysis
- frida
- QBDI A Dynamic Binary Instrumentation framework based on LLVM
- TinyInst A lightweight dynamic instrumentation library

符号执行

- Z3
- manticore Symbolic execution tool

gdb 插件

- peda
- pwndbg Exploit Development and Reverse Engineering with GDB Made Easy
- GEF GDB Enhanced Features for exploit devs & reversers

调试工具

- bcc
- openresty systemtap toolkit
- dtrace
- uftrace
- qira QEMU Interactive Runtime Analyse

模拟执行

- qemu
- unicorn
- OpenEmu
- panda Platform for Architecture-Neutral Dynamic Analysis
- avatar2

进程分析

- Process Explorer
- PeDoll Application behavior monitor based on inline hook
- libunwind

hook

- plthook Hook function calls by replacing PLT(Procedure Linkage Table) entries.
- funchook Hook function calls by inserting jump instructions at runtime

污点分析

- Triton
- bap Binary Analysis BinaryAnalysisPlatform

15.1.3 编译

编译工具

- llvm
- cmake
- ninja
- PeachPy x86-64 assembler embedded in Python

交叉编译

- buildroot Buildroot, making embedded Linux easy
- musl cross A small suite of scripts and patches for building musl libc cross compilers

15.1.4 漏洞利用

利用工具

- pwntools
- ROPgadget

Anti-AV

- GhostShell
- DefenderCheck Identifies the bytes that Microsoft Defender flags on

ShellCode

- GhostShell Malware indetectable, with AV bypass techniques, anti-disassembly, etc
- donut Generates x86, x64, or AMD64+x86 position-independent shellcode that loads .NET Assemblies, PE files, and other Windows payloads from memory and runs them with parameters

15.1.5 持久化

rootkit

- KasperskyHook Hook system calls on Windows by using Kaspersky's hypervisor
- MasterHide

免杀

- BypassAntiVirus 远控免杀系列文章及配套工具

无文件马

- fireELF Fileless Linux Malware Framework

后门

- The Backdoor Factory Patch PE, ELF, Mach-O binaries with shellcode

15.1.6 嵌入式设备

IoT

- awesome iot
- IoTSecurity101
- awesome iot hacks
- HomePWN

仿真

- firmadyne Platform for emulation and dynamic analysis of Linux-based firmware
- frankenstein Broadcom and Cypress firmware emulation for fuzzing and further full-stack debugging

漏洞分析工具

- FACT The Firmware Analysis and Comparison Tool (FACT)
- firmwalker
- Firmware Analysis Toolkit
- fwanalyzer
- Firmware Slap Discovering vulnerabilities in firmware through concolic analysis and function clustering.

BLE

- btlejack Bluetooth Low Energy Swiss-army knife

UEFI

- UEFITool UEFI firmware image viewer and editor

15.1.7 Android

分析工具

- classyshark Analyze any Android/Java based app or game
- jadx Dex to Java decompiler
- jd-gui Java Decompiler GUI
- dex2jar Tools to work with android .dex and java .class files
- Apktool tool for reverse engineering Android apk files
- JNI Helper Find JNI function signatures in APK and apply to reverse tools

漏洞

- qark Tool to look for several security related Android application vulnerabilities

改机工具

- Xposed
- Magisk

15.1.8 模糊测试

AFL 系列

- afl

内核

- kAFL Code for the USENIX 2017 paper: kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels
- syzkaller is an unsupervised coverage-guided kernel fuzzer
- VMI Kernel Fuzzer for Xen Project

PT-Fuzz

- libxdc The fastest Intel-PT decoder for fuzzing

语料

- fuzzdata

Android

- FANS Fuzzing Android Native System Services

ASAN 系列

- QASan a custom QEMU 3.1.1 that detects memory errors in the guest using AddressSanitizer

框架

- onefuzz A self-hosted Fuzzing-As-A-Service platform

网络协议

- boofuzz A fork and successor of the Sulley Fuzzing Framework
- AFLNet A Greybox Fuzzer for Network Protocols

文件结构

- FormatFuzzer is a framework for high-efficiency, high-quality generation and parsing of binary inputs

变异器

- radamsa Radamsa is a test case generator for robustness testing

论文列表

- fuzzing related paper

汇总

- Google Fuzzing Forum Tutorials, examples, discussions, research proposals, and other resources related to fuzzing

15.1.9 Demo

Hypervisor

- Bitdefender Napoca Hypervisor

15.1.10 资源

汇编

- pcasm PC Assembly Language book

论文集

- paper collection Academic papers related to fuzzing, binary analysis, and exploit dev
- Some Papers About Fuzzing

15.1.11 其他

静态工具

- static GDB and GDBServer
- statically compiled tools like Nmap and Socat
- static binaries

Sandbox

- any.run
- Noriben Portable, Simple, Malware Analysis Sandbox
- Cuckoo
- redmimicry
- LiSa Sandbox for automated Linux malware analysis

沙箱检测

- wsb detect enables you to detect if you are running in Windows Sandbox

Malware Sample

- malware samples
- theZoo

实验环境

- ABD Course materials for Advanced Binary Deobfuscation by NTT Secure Platform Laboratories

系统交互

- PythonForWindows is a base of code aimed to make interaction with Windows

15.2 书单

15.2.1 操作系统

综合

- CSAPP
- 程序员的自我修养：链接、装载与库
- 现代操作系统
- 自己动手写操作系统

- 《编码：隐匿在计算机软硬件背后的语言》【美】Charles Petzold
- 《深入理解计算机系统》【美】Randal E.Bryant
- 性能之巅

Linux

- Unix 环境高级编程
- Linux 内核完全注释
- Linux 内核情景分析
- Linux 内核源代码情景分析
- Linux 系统编程第 2 版
- Linux 设备驱动程序
- 《Linux 内核设计与实现》【美】Robert Love
- 《深入理解 Linux 内核》【美】DanielP.Bovet
- 深入理解 Linux 网络内幕
- 深入理解 Linux 虚拟内存管理

Windows

- 《深入理解 Windows 操作系统》【美】Russinovich,M.E.；Solomon,D.A.
- Windows 程序设计
- COM 技术内幕
- COM 原理与应用
- Windows 核心编程
- 深入解析 windows 操作系统 (Windows Internals)

Mac OS

- 深入理解 Mac OS X & iOS 操作系统【美】Jonathan Levin

Android

- Android 内核剖析
- Android 软件安全与逆向分析
- 《Android Dalvik 虚拟机结构及机制剖析（第 1、2 卷)》吴艳霞；张国印
- 《Android Internals::Power User's View》【美】Jonathan Levin
- 《Android 系统源代码情景分析》罗升阳
- 深入理解 Android
- 深入理解 Android 内核设计思想

- 第一行代码

15.2.2 嵌入式

- 计算机组成与设计硬件软件接口 https://book.douban.com/subject/2110638/
- ARM 嵌入式系统开发 https://book.douban.com/subject/1435663/
- MIPS 体系结构透视 https://book.douban.com/subject/3099520/

15.2.3 编译原理

- Compilers Principles Techniques and Tools
- Modern Compiler Implenentation in C
- Advanced Compiler Design and Implementation

15.2.4 浏览器

- 白帽子讲浏览器安全
- Browser Hack Handbook
- The Browser Hacker's Handbook
- How browser works

15.2.5 IoT

- 揭密家用路由器 0day 漏洞挖掘技术

15.2.6 体系架构

- MIPS 体系架构透视

15.2.7 虚拟化

- 系统虚拟化——原理与实现

15.2.8 漏洞利用

- 0day 安全软件漏洞分析技术
- The Shellcode's Handbook
- 堆栈攻击：八层网络安全防御

15.2.9 垃圾回收

- 垃圾回收的算法与实现

15.2.10 逆向工程

- 恶意代码分析实战
- C++ 反汇编与逆向技术分析
- 《编译与反编译技术实战》庞建民
- 《加密与解密》段钢
- 《恶意软件分析诀窍与工具箱——对抗“流氓”软件的技术与利器》【美】Michael Hale Ligh；Steven Adair
- 《C++ 反汇编与逆向分析技术揭秘》钱林松；赵海旭
- 《IDA 权威指南》【美】Chris Eagle
- 《逆向工程权威指南》【乌克兰】Dennis Yurichev，多平台入门大全
- 《Android 软件安全与逆向分析》丰生强
- 《macOS 软件安全与逆向分析》丰生强
- 《iOS 应用逆向工程（第 2 版）》沙梓社；吴航

15.2.11 算法

- Introductionto Algorithms (算法导论)

15.2.12 程序优化

- 代码大全
- 改善既有代码的设计

15.2.13 安全开发

- 《天书夜读：从汇编语言到 Windows 内核编程》谭文；邵坚磊
- 《Rootkit：系统灰色地带的潜伏者》【美】Bill Blunden
- 《Rootkits——Windows 内核的安全防护》【美】Gerg Hoglund；James Butler
- 《BSD ROOTKIT 设计——内核黑客指引书》【美】Joseph Kong
- 《寒江独钓：Windows 内核安全编程》谭文；杨潇；邵坚磊

15.2.14 综合

- 程序员的自我修养——链接、装载与库
- 计算机程序的构造和解释
- 计算机程序设计的艺术
- 《GEB ——一条永恒的金带》【美】道格拉斯 · 霍夫施塔特

15.3 文档资料

15.3.1 逆向工程

- reverse engineering reference manual

15.3.2 硬件

- The Hackers Hardware Toolkit

15.3.3 Windows

- Windows SDK Data Windows API listing in JSON format

15.3.4 Linux

- Linux insides

15.3.5 漏洞利用

- Awesome ARM Exploitation

15.3.6 可信执行

- TEE reversing

15.3.7 Fuzz

- Awesome AFL

15.4 逆向工具

15.4.1 IDA

快捷键

- `;` 添加注释
- `n` 定义或修改名称
- `g` 跳转到任意地方观察代码
- `Esc` 回到跳转前位置

15.4.2 objdump

- `-a` 反汇编所有头信息
- `-d` 反汇编所有包含指令的段
- `-D` 反汇编所有段
- `-s` 将所有段的内容以十六进制的格式打印
- `-S` 将代码段反汇编的同时，将反汇编代码和源代码交替显示，该选项需要调试信息
- `-t` 打印符号表
- `-C` 将 C++ 符号名逆向解析。
- `-l` 反汇编代码中插入源代码的文件名和行号。
- **`-j section`**
  - 仅反汇编指定的 section
  - j 可以有多个-j 参数来选择多个 section。

15.4.3 readelf

- `-h` 打印 ELF 的 header
- `-S` 打印 ELF 的 Section
- `-s` 打印 ELF 的符号表段

15.5 调试器

15.5.1 gdb

gdb

- **BreakPoint 断点**
  - `break [PROBE_MODIFIER] [LOCATION] [thread THREADNUM] [if CONDITION]`

  - **设置断点**
    - `break *0x080483d0` 在内存 0x080483d0 处下断点
    - `break <func name>` 在函数处下断点
    - `break *<func name> + 4` 在函数偏移处下断点
  - **相关操作**
    - `c` 从断点处继续运行
    - `d[elete] 1` 删除断点 1
    - `d 1 2` 删除断点 1 2
    - `d` 删除所有断点
    - `dis[able] 1` Disable breakpoint 1
    - `en[able] 1` Enable breakpoint 1
- `c[ontinue]` 运行至下一个断点或程序结束
- `command` 设置断点的命令
- `disas <func name>` 反汇编函数
- `fin[ish]` 跳出当前函数 / 循环
- **i info**
  - `i b` breakpoints info
  - `i f` frame info
  - `i s` stack info
- **ignore <break_list> count**
  - break_list 所指定的断点号将被忽略 count 次
- `n` / `next` 单步运行
- `p <func name>` 找函数地址
- `r <param>` 从头开始运行
- `s` / `step` 单步运行遇到调用时深入
- `file` 加载文件
- **x/nfu address 打印内存**
  - `n` 表示要显示的内存单元的个数
  - **f 表示显示方式**
    - `x` 按十六进制格式显示变量。
    - `d` 按十进制格式显示变量。
    - `u` 按十进制格式显示无符号整型。
    - `o` 按八进制格式显示变量。
    - `t` 按二进制格式显示变量。
    - `a` 按十六进制格式显示变量。
    - `i` 指令地址格式

        * `c` 按字符格式显示变量。
        * `f` 按浮点数格式显示变量。
    - **`u` 表示一个地址单元的长度**
        * `b` 表示单字节
        * `h` 表示双字节
        * `w` 表示四字节
        * `g` 表示八字节
    - 地址可以是内存地址或者是寄存器

gdb-peda

- `aslr` show aslr setting
- `checksec` Check for various security options of binary
- `dumpargs` Display arguments passed to a function when stopped at a call instruction
- `dumprop` Dump all ROP gadgets in specific memory range
- `elfheader` Get headers information from debugged ELF file
- `elfsymbol` Get non-debugging symbol information from an ELF file
- `lookup` Search for all addresses/references to addresses which belong to a memory range
- `patch` Patch memory start at an address with string/hexstring/int
- **`pattern` Generate, search, or write a cyclic pattern to memory**
    - `pattern_create 200` 生成一段长度为 200 的字符串
    - `pattern_offset 0x223333` 在 0x223333 的位置根据匹配找溢出的字节数
- `procinfo` Display various info from /proc/pid/
- `pshow` Show various PEDA options and other settings
- `pset` Set various PEDA options and other settings
- `readelf` Get headers information from an ELF file
- `ropgadget` Get common ROP gadgets of binary or library
- `ropsearch` Search for ROP gadgets in memory
- `searchmem|find` Search for a pattern in memory; support regex search
- `shellcode` Generate or download common shellcodes.
- `skeleton` Generate python exploit code template
- `vmmap` Get virtual mapping address ranges of section(s) in debugged process
- `xormem` XOR a memory region with a key

15.5.2 Windbg

命令行启动参数

- `g` 忽视第一个 debug 断点
- `p <pid>` attach 到 pid 为 <pid> 的程序

调试命令

- **单步调试**
  - `F11` 单步，遇到函数跟进 (Step Into)
  - `F10` 单步，遇到函数跳过 (Step Over)
  - `Shift + F11` 跳出当前函数 (Step Out)
- **执行到指定位置**
  - `g [地址 | 函数名]` 执行到指定位置
  - `gh [地址 | 函数名]` 执行到指定位置，如遇到异常则中断
  - `gn [地址 | 函数名]` 执行到指定位置，如遇到异常则忽略
- **断点**
  - `Ctrl+Break` 中断当前运行
  - `bl` 列出已设置断点
  - `be [断点 ID]` 激活断点
  - `bd [断点 ID]` 禁用断点
  - `bc [断点 ID]` 清除断点
  - `bp [地址 | 函数名]` 设置断点
- **数据展示**
  - `d [地址]` 显示内存数据
  - `db [地址]` 按字节显示内存数据
  - `dd [地址]` 按双字节显示内存数据
- **数据编辑**
  - `e [地址] [数据]` 修改任意内存地址的值
- **栈的显示**
  - `k [x]` 打印调用栈，x 为需要回溯的栈帧数
  - `kb [x]` 打印调用栈，并额外显示 3 个传递给函数的参数
- **模块显示**
  - `lm` 列出已经读入的所有模块
  - `lmvm [name]` 查看模块的详细信息
  - `x` 查找符号的二进制地址
- **反汇编功能**

  - u 反汇编当前执行的后几条指令并显示
  - u ［起始地址］从指定的地址开始反汇编
  - u ［起始地址］［终止地址］反汇编指定区间的汇编代码
- 重载 *.reload*
- 帮助 *.help*
- 清屏 *.cls*
- 开启 DML 帮助 *.prefer_dml 1*
- **设置符号地址**
  - *.sympath <path>*
  - *.sympath+ srv*<path>*http://msdl.microsoft.com/download/symbols*
- **进程信息**
  - *.tlist*
  - *!process* 显示调试器当前运行进程信息
  - *!process 0* 显示进程列表
  - *!process <pid>* 显示进程信息
  - *!peb <address>* 查看 Process Enviroment Block
- **线程信息**
  - ~ 显示线程信息
  - ~ *<thread id>* 切换
  - *!teb <address>* 查看 TEB 信息
  - *.ttime* 查看线程时间
- **异常信息**
  - *.exr*
  - *.exr -1* 约等于 *.lastevent*
  - *.bugcheck* 显示当前 bug check 的详细信息，用于调试 crash
  - *!analyze* 分析最近的异常事件
- **错误信息**
  - *!error <eid>* 获取错误码为 eid 的 Win32 错误信息
  - *!gle* get last error

15.6 编译工具

15.6.1 gcc

常规选项

- `-c` 只执行预处理、编译、汇编，不执行链接过程
- `-E` 输出预处理文件
- `-g` 在可执行程序中包含标准调试信息
- `-I <dir>` 在头文件的搜索路径列表中添加 dir 目录
- `-L <dir>` 在库文件的搜索路径列表中添加 dir 目录
- `-o <filename>` 设置生成的文件名
- `-Ox` 设置优化的等级
- `-s` 去掉符号
- `-S` 输出汇编文件
- `-v` 打印出编译器内部编译各过程的命令行信息和编译器的版本

安全相关选项

- `-fno-exceptions` 不使用异常
- `-fno-stack-protector` 不开启 canary 栈溢出检测
- `-z execstack` 开启栈可执行关闭 NX

体系结构相关选项

- `-m32` 生成 32bit 程序需要 gcc-multilib (x86 机器上编译不用加)
- `-mcpu=type` 针对不同的 CPU 使用相应的 CPU 指令，type 可为 i386/i486/pentium/i686 等
- `-mieee-fp` 使用 IEEE 标准进行浮点数的比较
- `-mno-ieee-fp` 不使用 IEEE 标准进行浮点数的比较
- `-mrtd` 强行将函数参数个数固定的函数用 `ret NUM` 返回，节省调用函数的一条指令
- `-mshort` 把 int 类型作为 16 位处理，相当于 short int
- `-msoft-float` 输出包含浮点库调用的目标代码

报警相关选项

- `-ansi` 支持符合 ANSI 标准的 C 程序
- `-pedantic` 允许发出 ANSI C 标准所列的全部警告信息
- `-pedantic-error` 允许发出 ANSI C 标准所列的全部错误信息
- `-w` 关闭所有告警
- `-Wall` 允许发出 Gcc 提供的所有有用的报警信息
- `-Werror` 把所有的告警信息转化为错误信息，并在告警发生时终止编译过程

15.6.2 clang

15.6.3 CMake

FLAGS

- `CC` 编译器
- `CXX` C++ 编译器
- `LD` 链接器
- `CFLAGS` C 编译器的选项
- `CXXFLAGS` C++ 编译器的选项
- `LDFLAGS` 链接器的选项
- `LIBS` 链接器要链接的库

15.7 系统工具

15.7.1 ulimit

参数

- `-a` 列出所有当前资源极限
- `-c` 设置 core 文件的最大值. 单位:blocks
- `-d` 设置一个进程的数据段的最大值. 单位:kbytes
- `-f` Shell 创建文件的文件大小的最大值，单位：blocks
- `-h` 指定设置某个给定资源的硬极限。如果用户拥有 root 用户权限，可以增大硬极限。任何用户均可减少硬极限
- `-l` 可以锁住的物理内存的最大值
- `-m` 可以使用的常驻内存的最大值，单位：kbytes
- `-n` 每个进程可以同时打开的最大文件数
- `-p` 设置管道的最大值，单位为 block，1block=512bytes
- `-s` 指定堆栈的最大值：单位：kbytes

- -S 指定为给定的资源设置软极限。软极限可增大到硬极限的值。如果 -H 和 -S 标志均未指定，极限适用于以上二者
- -t 指定每个进程所使用的秒数，单位：seconds
- -u 可以运行的最大并发进程数
- -v Shell 可使用的最大的虚拟内存，单位：kbytes

其中设置 unlimited 为取消内存限制

CHAPTER 16

其他

16.1 垃圾回收

16.1.1 简介

垃圾回收 (Garbage Collection, GC) 指程序不用的内存空间。

满足 找到内存空间里的垃圾 + 回收垃圾，让程序员能再次利用两项功能的程序就是 GC。由 John McCarthy 在 1960 年首次发布。

16.1.2 常用概念

mutator

用于生成对象和更新指针。

堆

用于动态存放对象的内存空间，当 mutator 申请存放对象时，所需的内存空间就会从堆分配给 mutator。

活动对象/非活动对象

活动对象指能通过 mutator 引用的对象，非活动对象指不能通过程序引用的对象。

分配

分配（allocation）在内存空间中分配对象。当没有可分配空间时，GC 有两种选择：销毁所有的计算结果，输出错误信息，扩大堆，分配可用空间。

16.1.3 评价标准

评价 GC 算法的性能，采用吞吐量、最大暂停时间、堆使用效率、访问的局部性 4 个标准。

其中吞吐量指单位时间内的处理能力。最大暂停时间指因执行 GC 而暂停执行 mutator 的最长时间。

16.1.4 GC 算法

标记 - 清除算法（Mark Sweep GC）

分为标记阶段和清除阶段两个阶段。标记阶段为对象打上标签。collector 会遍历整个堆，回收没有打上标签的对象

分配策略：

- First-fit 分配找到的第一个大于等于 size 的分块
- Best-fit 返回大于等于 size 的最小分块
- Worst-fit 找最大的分块

优点在于实现简单，与保守式 GC 算法兼容。

缺点在于碎片化，分配速度慢，与写时复制技术不兼容。

引用计数算法

引入计数器的概念，表示对象的人气指数。

优点在于可即刻回收垃圾，最大暂停时间短，没有必要沿指针查找。

缺点在于计数器值增减处理繁重，计数器需要占用很多位，实现烦琐复杂，循环引用无法回收。

16.2 沙箱

16.2.1 虚拟环境检测

检测进程名

- **Vmware**
  - Vmtoolsd
  - Vmwaretrat

  - Vmwareuser
  - Vmacthlp
- **VirtualBox**
  - vboxservice
  - vboxtray

检测注册表

- HKLMSOFTWAREVmware IncVmware Tools
- HKEY_CLASSES_ROOTApplicationsVMwareHostOpen.exe
- HKEY_LOCAL_MACHINESOFTWAREOracleVirtualBox Guest Additions

硬盘文件检测

- **VMware**
  - C:windowsSystem32DriversVmmouse.sys
  - C:windowsSystem32Driversvmtray.dll
  - C:windowsSystem32DriversVMToolsHook.dll
  - C:windowsSystem32Driversvmmousever.dll
  - C:windowsSystem32Driversvmhgfs.dll
  - C:windowsSystem32DriversvmGuestLib.dll
- **VirtualBox**
  - C:windowsSystem32DriversVBoxMouse.sys
  - C:windowsSystem32DriversVBoxGuest.sys
  - C:windowsSystem32DriversVBoxSF.sys
  - C:windowsSystem32DriversVBoxVideo.sys
  - C:windowsSystem32vboxdisp.dll
  - C:windowsSystem32vboxhook.dll
  - C:windowsSystem32vboxoglerrorspu.dll
  - C:windowsSystem32vboxoglpassthroughspu.dll
  - C:windowsSystem32vboxservice.exe
  - C:windowsSystem32vboxtray.exe
  - C:windowsSystem32VBoxControl.exe

运行服务检测

- VMTools
- Vmrawdsk
- Vmusbmouse
- Vmvss
- Vmscsi
- Vmxnet
- vmx_svga
- Vmware Tools

mac 地址前缀

- 00:05:69 (Vmware)
- 00:0C:29 (Vmware)
- 00:1C:14 (Vmware)
- 00:50:56 (Vmware)
- 08:00:27 (VirtualBox)

硬件

- CPU 核心数
- CPU 温度
- CPUID 指令
- MAC 地址
- 内存大小
- 磁盘大小
- 注册表和文件路径
- 主板序列号 / 主机型号

文件系统

- TEMP 目录下的文件数量
- 最近打开的文件

虚拟指令

- 特定指令

16.2.2 沙箱环境检测

- **延迟运行**
  - 自动化沙箱运行时间较短，可延迟一段时间后再运行
- **检测开机时间**
  - 许多沙箱检测完毕后会重置系统，开机时间很短
- **检测配置**
  - 沙箱环境的配置大多不高
- 检测进程 / 服务
- 代码运行时间

16.2.3 参考链接

- 反虚拟机和沙箱检测的一些小技巧
- CheckVM Sandbox

16.3 常见术语

16.3.1 二进制

- 最低有效位 (Least Significant Bit, LSB)
- 最高有效位 (Most Significant Bit, MSB)

可执行文件

- 动态链接库 (Dynamic Linked Library, DLL)

16.3.2 编译

- 即时编译器 (Just In Time compiler, JIT)
- 中间表示 (Intermediate Representation, IR)
- 控制流图 (Control Flow Graph, CFG)
- 抽象语法树 (Abstract Syntax Tree, AST)
- 静态单赋值 (Static Single-Assignment, SSA)

16.3.3 硬件设备

存储设备

- 只读存储器 (Read Only Memory, ROM)
- 随机存取存储器 (Random Access Memory, RAM)
- 静态随机存取存储器 (Static Random Access Memory, SRAM)
- 动态随机存取存储器 (Dynamic Random Access Memory, DRAM)
- 非易失内存芯片 (Non-Volatile Memory)
- 带电可擦可编程只读存储器 (Electrically Erasable Programmable Read-only Memory, EEPROM)
- 非易失性随机访问存储器 (Non-Volatile Random Access Memory, NVRAM)
- 内存技术设备 (Memory Technology Device, MTD)

处理设备

- CPU (Central Process Unit)
- GPU (Graphic Process Unit)
- 算术逻辑单元 (Arithmetic and Logic Unit, ALU)

硬件接口

- UART (Universal Asynchronous Receiver/Transmitter)
- JTAG (Joint Test Action Group)
- 外设部件互连标准 (Peripheral Component Interconnect, PCI)

16.3.4 数据交互

- MMIO (Memory-Mapped Input/Output)
- PMIO (Port-Mapped Input/Output)
- DMA (direct memory access)
- 应用二进制接口 (Application Binary Interface, ABI)

16.3.5 指令集架构

- ISA (Industrial Standard Architecture)
- 指令集体系结构 (Instruction Set Architecture)
- IRQ (Interrupt Request)
- 复杂指令集运算 (Complex Instruction Set Computing, CISC)
- 精简指令集运算 (Reduced Instruction Set Computing, RISC)
- 显式并行指令集运算 (Explicitly Parallel Instruction Computing, EPIC)

16.3.6 程序分析

- S2E (Symbolic Sombolic Exection)

16.3.7 内存与地址

- Guest 虚拟地址 (Guest Virtual Address, GVA)
- Guest 物理地址 (Guest Physical Address, GPA)
- Host 虚拟地址 (Host Virtual Address, HVA)
- Host 物理地址 (Host Physical Address, HPA)
- 内存管理单元 (Memory Management Unit, MMU)
- 内存管理单元 (Paged Memory Management Unit, PMMU)

16.3.8 安全机制

- WDAC (Windows Defender Application Control)

16.3.9 机制

- 写时复制 (Copy On Write, COW)

CHAPTER 17

目录

- genindex
- modindex
- search