计算机系统基础 I

20天速通复习手册

开卷考试专用 · 含完整复习路线 + 真题精讲 + 考场速查表

适用专业：软工 2026年 期末

使用说明：

本手册按章节编号，目录定位章节，每节顶部都标明【对应复习日 + 真题考查次数】。

日常复习按第0章的路线图推进；考场上用第七部分速查表 + 目录快速定位。

目录 Contents

（在 Word 中按 Ctrl+点击 章节标题可跳转）

第零部分 20天复习路线图

总策略：把 80% 时间砸在 x86-64 汇编（第二部分）和 链接

（第三部分）上，这两块占考试约 60+ 分。数据表示（第一部分）是分析题和选择题大头，必须吃透。最后2天集中做真题。

总览：每日核心任务

天数	主题	学习内容	对应章节
1	进制 + 字节	二进制/十六进制互转、字节存储、小端字节序、ASCII 关键值	1.1
2	无符号 / 补码	无符号编码、补码原理、零扩展/符号扩展、unsigned↔signed 转换陷阱	1.2
3	整数练习	做 10 道补码/进制互转题，复习 1.1+1.2	1.1, 1.2
4	加减法 + 标志位	加法溢出判断、CF/ZF/SF/OF 含义、addw 指令后标志位计算（真题分析题1）	1.3
5	移位 + 乘除	逻辑/算术移位、x/2^k 的偏移量加法、负数除法陷阱	1.4
6	IEEE 754 表示	单/双精度结构、Bias、规格化/非规格化/特殊值	1.5
7	IEEE 754 运算	对阶+尾数运算+规格化+舍入，整数除0 vs 浮点除0（真题分析题4）	1.6
8	x86-64 入门	寄存器组、字节序、AT&T 语法、操作数表示	2.1, 2.2
9	数据传送 + 算术	mov/lea/add/sub/imul、leaq 不访存只算地址	2.3
10	条件 + 跳转	条件码、jX 系列、cmp/test、jbe 偏移量计算（真题选择题9）	2.4, 2.5
11	过程调用 ★	参数传递（rdi/rsi/rdx/rcx/r8/r9）、栈帧、call/ret 机制（真题选择题7 + 综合应用题）	2.6
12	数据结构 + 越界	数组寻址、结构体对齐、缓冲区溢出原理	2.7, 2.8
13	ELF + 节	.text/.data/.bss/.rodata 各存什么（真题分析题2）	3.1, 3.2
14	符号 + 解析	三类符号（全局/外部/局部）、静态库链接顺序（真题选择题8）	3.3, 3.4
15	重定位 ★	R_X86_64_PC32 vs PLT32、call 指令机器码构造（真题综合题6）	3.5, 3.6
16	AttackLab	攻击字符串构造、返回地址覆盖、防御机制（真题实验题）	2.8, 4.2
17	LinkLab + 复习	重定位修改、综合复习链接	4.3

18	真题 P1	做去年真题选择题 + 分析题，对照第五部分讲解	第五部分
19	真题 P2	做去年真题综合应用题 + 实验题	第五部分
20	考场准备	通读第六部分速查表，整理	

每日时长建议

- 每天 2-3 小时即可，零基础也来得及。开卷不需要背，关键是看懂例题、能复现解题过程。
- **学习节奏：**先看本手册对应章节的【必背规则】→ 看【详解】 → 自己做【真题应用】 → 看自己错在哪。
- **不要做的事：**不要从头到尾通读 PPT，浪费时间。也不要试图背反汇编代码——开卷查就行。

难度提示

【简单】第一部分（数据表示）只要会算就行，不涉及死记。

【中等】第三部分（链接）概念多但套路化，结合真题练习几遍能掌握。

⚠️ 第二部分（x86-64 汇编）是真正的难点和重点。第 8-12 天必须认真投入，否则综合应用题 50 分会丢一半。

考前一周提醒

1. 把本手册**第六部分速查表** 单独打印，订成单独的小册子带进考场。
2. 把**反汇编工具的指令格式** 操作码 + 偏移量 + 助记符 记熟，看到一行机器码能立刻知道在干嘛。
3. 准备一支铅笔和橡皮。补码/IEEE 754 计算容易出错，要能擦改。
4. 最后看一眼：**小端存储** （低字节存低地址）—— 几乎每次考试都考这个的细节。

第一部分 数据的机器级表示与处理

对应教材第 2 章。这部分是选择题和分析题的主要来源。

📌 **真题考点：**去年真题里出现：选择题第 2 题（补码特点）、第 3 题（unsigned 转 short）、第 4 题（舍入方式）、第 5 题（-1028.0 的 IEEE 754）、第 10 题（小端存储）；分析题第 1 题（addw 后标志位）、第 4 题（整数除0 vs 浮点除0）。

1.1 进制与字节存储

【复习日】第 1 天 · 真题考查 2 次以上

1.1.1 进制互转

必记的关键值（背一遍就再也不用查）：

十进制	二进制	十六进制	说明
0	0000	0	
1	0001	1	
2	0010	2	
4	0100	4	
8	1000	8	
10	1010	A	
15	1111	F	一位十六进制最大值
16	0001 0000	10	16 = 2^4
255	1111 1111	FF	一字节最大值（8位）
256	1 0000 0000	100	2^8
1024	100 0000 0000	400	2^10 = 1K
65535	1111 1111 1111 1111	FFFF	2^16-1，一个 word 的最大无符号值
65536	1 0000 0000 0000 0000	10000	2^16

1.1.2 转换方法

二进制 → 十六进制：

- 从右往左每 4 位一组，每组对应一个十六进制位。

例：1101 1010 0011 = D A 3 = 0xDA3

十进制整数 → 二进制（除2取余）：

- 反复除以 2，记下余数，余数倒着读

例：23 ÷ 2 = 11 余 1　11 ÷ 2 = 5 余 1　5 ÷ 2 = 2 余 1　2 ÷ 2 = 1 余 0　1 ÷ 2 = 0 余 1　倒读：10111　验证：16 + 4 + 2 + 1 = 23 ✓

十进制小数 → 二进制（乘2取整）：

- 反复乘 2，取整数部分，从上到下顺读

例：0.8125　0.8125 × 2 = 1.625 → 1　0.625 × 2 = 1.25 → 1　0.25 × 2 = 0.5 → 0　0.5 × 2 = 1.0 → 1　顺读：0.1101

⚠️ 小数转换不一定能在有限位内结束。例如 0.1 在二进制中是循环小数 0.00011001100..., 这就是为什么浮点数加减会有精度误差。

1.1.3 字节、字、容量单位

- 1 字节 (byte) = 8 位 (bit)
- 1 字 (word) 在 x86-64 中 = 16 位 = 2 字节
- 1 双字 (doubleword) = 32 位 = 4 字节
- 1 四字 (quadword) = 64 位 = 8 字节

常用换算（背！）：

单位	等于	近似十进制
1 KB	1024 B = 2^10 B	~10^3
1 MB	1024 KB = 2^20 B	~10^6
1 GB	1024 MB = 2^30 B	~10^9
1 TB	1024 GB = 2^40 B	~10^12
1 PB	1024 TB = 2^50 B	~10^15

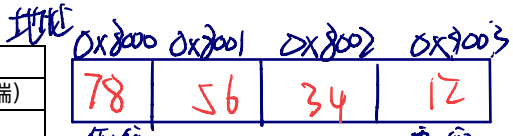
1.1.4 小端存储 / Little Endian ★

📌 **真题考点：**真题选择题第 10 题、复习PPT例题，几乎必考。

x86-64 是小端字节序：多字节数据中低位字节存放在低地址，高位字节存放在高地址。

套路：看到 int 变量 x = 0x12345678，地址为 0x8000，问后面字节存什么

地址	存放字节	解释
0x8000	0x78	最低位字节（小端）
0x8001	0x56	
0x8002	0x34	
0x8003	0x12	最高位字节

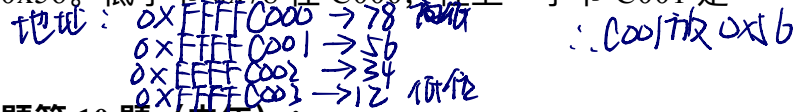


eg: 0x 12 34 56 78 [4个字节]

真题练习（复习PPT原题）：

设 intel 机器中，int 变量 x 的地址为 0xFFFFC000，x = 0x12345678，则内存单元 0xFFFFC001 中的内容十六进制表示为?

答案：0x56。低字节 0x78 在 C000，往上一字节 C001 是 0x56。



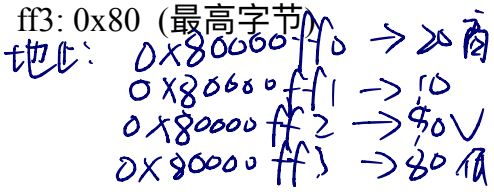
再看真题第 10 题（去年）：

一个 int 型变量 x 的地址为 0x80000ff0，x = 0x80901020，则地址 0x80000ff2 单元中存放的内容十六进制表示为?

地址偏移：ff2 - ff0 = 2，所以是 x 的第 3 个字节（从低位数）。

x = 0x 80 90 10 20　　高 ←→ 低 小端存储：　ff0: 0x20 (最低字节) ff1: 0x10 ff2: 0x90 ← 答案 ff3: 0x80 (最高字节)

答案选 C (0x90)。



1.2 无符号整数与补码（带符号整数）

【复习日】第 2 天 · 真题反复考查

1.2.1 无符号整数

n 位无符号整数表示范围：0 到 2^n - 1。

$0 - 2^n - 1$

数值就是各位二进制的加权和：

$X = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_1 \times 2 + b_0$

1.2.2 补码（带符号整数）

现代计算机几乎所有的带符号整数都用补码表示。

补码的定义：

- 正数：补码 = 原码（即直接写二进制）
- 负数：补码 = 对应正数的**各位取反，末位加1**

举例（8位机器数）：

+23 的补码 = 0001 0111 -23 的补码 = +23 取反 + 1 = 1110 1000 + 1 = 1110 1001

n 位补码的表示范围：

从 -2^{n-1} 到 $2^{n-1} - 1$ （负数比正数多一个，因为 0 占了正数那边）

常见值（背！）：

位数	最小负数	最大正数
8 位 (char)	-128 = 0x80	127 = 0x7F
16 位 (short)	-32768 = 0x8000	32767 = 0x7FFF
32 位 (int)	-2^31 ≈ -2.1×10^9	2^31-1
64 位 (long)	-2^63	2^63-1

1.2.3 补码的关键特点（真题选择题）

📌 **真题考点：**去年真题选择题第 2 题：以下关于补码特点的描述错误的是？答案 C： 「+a 与 -a 的编码仅符号位不一样」是错的——补码下负数要取反加1，不仅仅是符号位不同。

补码的正确特点：

1. **零的表示是唯一的**（不像原码有 +0 和 -0 两个）
2. 加法运算时，**符号位可以和数值位一起参加运算**

3. 补码所对应的真值范围是不对称的，**负数比正数多一个**

错误的说法（陷阱）：

⚠️ 「+a 与 -a 的编码仅符号位不一样」—— 错。这是原码的特点，不是补码的。

1.2.4 由补码求十进制值

方法 1（推荐，最稳定）：

X = -b_{n-1} × 2^{n-1} + b_{n-2} × 2^{n-2} + ... + b_0　　↑ 最高位（符号位）取负权

例：8 位机器数 1110 1001

= -1×128 + 1×64 + 1×32 + 0×16 + 1×8 + 0×4 + 0×2 + 1×1 = -128 + 64 + 32 + 8 + 1 = -23 ✓

方法 2（看符号位）：

- 如果最高位是 0：直接当无符号数算。
- 如果最高位是 1：先 各位取反末位加1，得到对应正数，然后加负号。

1.2.5 unsigned ↔ signed 转换陷阱 ★

📌 **真题考点：**去年真题选择题第 3 题。

位模式不变，

。

真题原题：

unsigned short usi = 32768; short si = usi; printf(...) si 的值是?

分析：

1. usi 是 unsigned short，16 位，32768 = 0x8000
2. 位模式 1000 0000 0000 0000 赋给 short（按补码解释）
3. 作为补码：最高位是 1，是个负数。求其值：-2^15 = -32768
4. 答案：D -32768

1.2.6 零扩展与符号扩展

小类型转大类型时（如 char → int）：

- **unsigned 类型 → 零扩展：**高位补 0。如 unsigned char 0xFF → unsigned int 0x000000FF
- **signed 类型 → 符号扩展：**高位补符号位。如 char 0xFF (-1) → int 0xFFFFFFFF (-1)

助记：

【口诀】无符号补零，有符号补符号位。

汇编指令对应：

- movzbl：move zero-extend byte to long（零扩展）
- movsbl：move sign-extend byte to long（符号扩展）

1.3 整数运算、溢出与标志位

【复习日】第 4 天 · 真题分析题1 必考

1.3.1 加减法的本质

在补码下，

。

x - y = x + (-y) ← (-y) 是 y 的补码取反加1

符号位和数值位一起参加运算，最高位的进位丢弃。

1.3.2 标志位（关键！背熟）

📌 **真题考点：**分析题第 1 题：addw 后 ZF/CF/OF/SF 怎么定。

加法 x + y = result，假设所有数是 n 位：

标志	名称	判定规则
CF	进位标志	无符号溢出：最高位向更高位有进位时为 1。 等价：把 x、y 当无符号数加，结果 ≥ 2^n 则 CF=1

ZF	零标志	结果为全 0 时 ZF=1
SF	符号标志	结果的最高位（即补码下的符号位）
OF	溢出标志	有符号溢出：x、y 符号相同但结果符号变了，则 OF=1 即：正+正=负 或 负+负=正

CPU每次加减都把4溢出重新设置一次

减法 x - y 时（实际是 x + (~y) + 1）：

- CF: 借位发生 → CF=1（无符号下 x < y）
- OF: 有符号溢出（如 INT_MIN - 1）

1.3.3 真题分析题1精讲

原题：

假设 ax = 0xffd0, bx = 0x8005，执行指令 addw %bx, %ax 后，寄存器 ax 和标志寄存器的 ZF、CF、OF、SF 的值分别是?

解题步骤：

1. 加法（按 16 位计算）： ax = 0xffd0 = 1111 1111 1101 0000
bx = 0x8005 = 1000 0000 0000 0101

——————————————— 和 = 1 0111 1111 1101 0101 ↑ 进位出去（17位），低 16 位是 0x7fd5

2. 各标志位： ax 新值 = 0x7fd5 ← 注意高位的进位丢弃了
ZF = 0 (结果不为 0) CF = 1 (有进位出去 → 无符号溢出)
OF = 1 (负 + 负 = 正：0xffd0 和 0x8005 都是负数，结果 0x7fd5 是正数) SF = 0 (结果最高位是 0)

最终答案：ax = 0x7fd5；ZF=0, CF=1, OF=1, SF=0

⚠️ 题目要求填 ZF、CF、OF、SF 的值，注意顺序！真题答题处用户写的是「1、0、1、0」，对应顺序需要核对。我重新算的标准答案如上：0、1、1、0。

①转二进制
②竖式加
③结果取16位(溢出不算)
④判CF,ZF,SF,OF

1.3.4 溢出判断的快速规则

无符号加法溢出（=CF=1）：

- 两个无符号数相加，结果反而比加数小
- 等价：最高位有进位

有符号加法溢出（=OF=1）：

- 正 + 正 = 负（不可能正+正还是正吧？错就是溢出）
- 负 + 负 = 正
- 正 + 负 永远不溢出

1.4 移位、扩展、截断；乘除替换

【复习日】第 5 天

1.4.1 移位运算

类型	汇编指令	效果
逻辑左移	shl / sal	左移 k 位，右边补 0，等价于 × 2^k（无符号）
逻辑右移	shr	右移 k 位，左边补 0（无符号 / 2^k）
算术右移	sar	右移 k 位，左边补符号位（有符号 / 2^k，向下取整）

1.4.2 乘除替换（编译器优化）

乘法/除法慢，编译器会用移位+加减组合替换。

乘以 2^k：

- 无符号或有符号都用左移：x << k

乘以非 2 的幂（如 20）：

- 拆成 2 的幂之和：20 = 16 + 4 = 2^4 + 2^2
- x * 20 → (x<<4) + (x<<2)

除以 2^k：

- **无符号：**逻辑右移 x >> k
- **有符号正数：**算术右移 x >> k
- **有符号负数 ★：**需要先加偏移量 2^k - 1 再算术右移：(x + (1<<k) - 1) >> k

为什么负数要加偏移量?

C 标准要求除法向 0 取整（截断），但算术右移向 -∞ 取整。两者对负数不一致。

例：-14 / 4，正确结果是 -3（向 0 截断） 直接算术右移 4 位：1111 0010 >> 2 = 1111 1100 = -4，错了 先加偏移 (2^2 - 1) = 3：-14 + 3 = -11 = 1111 0101 右移 2 位：1111 1101 = -3 ✓

1.4.3 整数除 0（真题分析题4 一半）

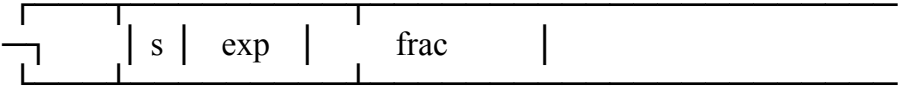
整数除 0 会
，因为商无法用一个机器数表示。例如除 -1 整数也可能溢出（INT_MIN / -1）。
操作系统会调出异常处理程序处理。

1.5 IEEE 754 浮点数 ★

【复习日】第 6 天 · 真题选择题5 必考

1.5.1 浮点数的总体格式

v = (-1)^s × M × 2^E



1 bit k bits n bits s ← 符号位 exp ← 编码 E（偏置形式） frac← 编码 M（尾数小数部分）

两种精度（必背！）：

类型	总位数	符号位	exp 位数	frac 位数	Bias
单精度 float	32	1	8	23	127
双精度 double	64	1	11	52	1023

1.5.2 三种数值情况（看 exp 判别）

类型	exp	frac	数值计算
规格化数	不全0、不全1	任意	v = (-1)^s × 1.frac × 2^(exp - Bias)

非规格化数	全 0	任意	v = (-1)^s × 0.frac × 2^(1 - Bias)
+0 / -0	全 0	全 0	0（区分符号但相等）
±∞	全 1	全 0	+∞（s=0）或 -∞（s=1）
NaN	全 1	非全 0	Not a Number

1.5.3 规格化数完整算法（必练熟）★

十进制 → IEEE 754 单精度（套路化的 4 步）：

1. 把十进制数转成二进制（整数+小数）。
2. 规格化为 1.xxxxx × 2^E 形式（小数点移到最高位的 1 后面）。
3. 计算 exp = E + 127，转成 8 位二进制。
4. frac = 小数点后的 23 位（不够补0）；s 看正负。

1.5.4 真题选择题5精讲 ★

原题：

-1028.0 采用 IEEE 754 单精度格式表示的结果（十六进制）是?

A. c4808000　B. c4c04000　C. 44804000　D. 44c04000

解题（套 4 步）：

第1步：转二进制 1028 = 1024 + 4 = 2^10 + 2^2 = 10000000100 -1028.0 = -10000000100.0 (B)

第2步：规格化 -10000000100.0 = -1.0000000100 × 2^10 （小数点从最右移到最左的 1 后面，移了 10 位）

第3步：计算 exp E = 10 exp = E + 127 = 137 = 10001001 (B)

第4步：组装 s = 1 （负数） exp = 10001001 frac= 0000000100 + 13个0 = 00000001 00000000 00000000 完整：1 10001001 00000001 00000000 00000000 分组：11000100 10000000 10000000 00000000 HEX：C 4 8 0 8 0 0 0

答案：A. c4808000

⚠️ 你在真题上圈了 B（c4c04000），但用户旁边的草稿

C4808 是正确推算。考试时认真核对最终选项！

1.5.5 IEEE 754 → 十进制（反向）

步骤：

1. 按 1+8+23 拆出 s、exp、frac
2. 判断类型：exp 全0/全1/中间?
3. 规格化数：E = exp - 127，M = 1.frac，v = (-1)^s × M × 2^E

举例：0xBEE00000 → ?

二进制：1 01111101 11000000000000000000000 s = 1 → 负数 exp = 01111101 = 125 → E = 125 - 127 = -2 frac = 1100000... → M = 1.11_2 = 1 + 1/2 + 1/4 = 1.75 v = -1 × 1.75 × 2^(-2) = -0.4375

1.5.6 IEEE 754 数值范围

单精度：

- 最大正规格化数 ≈ +3.4 × 10^38
- 最小正规格化数 ≈ +1.18 × 10^-38
- 最大正非规格化数 ≈ +1.18 × 10^-38（略小于最小正规格化数）

双精度：

- 最大 ≈ +1.8 × 10^308

1.6 浮点数的运算与异常

【复习日】第 7 天 · 真题分析题4 一半

1.6.1 加减法的 5 步（重要）

1. **求阶差** Δe = Ee - Xe
2. **对阶** 把小阶向大阶看齐：阶小的数尾数右移 Δe 位，移出

的低位保留在「附加位」

3. **尾数加减**
4. **规格化**

- 尾数高位是 0 → 左规：尾数左移 1 位，阶 -1，直到 MSB 为 1 或阶下溢
- 尾数最高位有进位 → 右规：尾数右移 1 位，阶 +1（最多右规 1 次）

1. **舍入 + 判溢出**

1.6.2 舍入方式（真题选择题4）

📌 **真题考点：**去年真题选择题第 4 题：下面舍入方式中，哪种方式更能避免统计偏差？答案 A：向偶数舍入。

方式	说明	处理 1.5
向偶数舍入 (banker's)	正常舍入；遇中间值则舍向最近的偶数	→ 2
向 0 舍入	截断小数部分	→ 1
向 +∞ 舍入	向更大的方向	→ 2
向 -∞ 舍入	向更小的方向	→ 1

⚠️「向偶数舍入」最公平，统计意义上无偏差——大数据下不会让总和偏大或偏小，所以 IEEE 754 默认采用这种。

1.6.3 浮点数除 0（真题分析题4 另一半）

📌 **真题考点：**真题分析题4：请分析整数除 0 会发生异常，而浮点数除 0 不会出现异常的原因。

回答要点：

1. 整数除 0：商无法用任何机器数表示，所以引发异常，由 OS 异常处理。
2. 浮点数除 0：IEEE 754 **用 +∞/-∞ 表示无穷大、NaN 表示无意义结果**，编码下exp 全 1, frac 全 0 即为 ±∞，所以是合法的可表示数。
3. 浮点正数 / 0 = +∞，负数 / 0 = -∞，0/0 = NaN。可作比较用，如 X/0 > Y 是有效比较。

4. 总结：浮点数比整数多了表达「无穷」和「无效」的能力，所以不需要异常机制。

1.6.4 浮点数运算的特殊性质

- 浮点加法不满足结合律。如 (-1.5×10^38 + 1.5×10^38) + 1.0 = 1.0，但 -1.5×10^38 + (1.5×10^38 + 1.0) = 0.0
- 从 int 到 float 可能丢精度（float 只有 23 位 frac，约 7 个十进制有效位）
- 从 float 到 int 可能截断（小数部分丢失）

第二部分 程序的机器级表示（x86-64）

对应教材第 3 章。这是整门课的核心，综合应用题 35-50 分主要来自这里。

📌 **真题考点：**去年真题：选择题 6-9 题、分析题 3 题、整个综合应用题 50 分基本都是这部分内容。

2.1 x86-64 寄存器组

【复习日】第 8 天 · 几乎每道汇编题都用到

2.1.1 16 个通用整数寄存器

64 位寄存器全名以 r 开头，访问低 32 位用 e 开头（或 r..d），低 16 位 / 低 8 位还有更短名字。

64位	低32位	低16位 / 低8位	约定用途
%rax	%eax	%ax / %al	返回值；除法用
%rbx	%ebx	%bx / %bl	被调用者保存
%rcx	%ecx	%cx / %cl	第 4 个参数
%rdx	%edx	%dx / %dl	第 3 个参数
%rsi	%esi	%si / %sil	第 2 个参数
%rdi	%edi	%di / %dil	第 1 个参数
%rsp	%esp	%sp / %spl	栈指针（特殊）
%rbp	%ebp	%bp / %bpl	被调用者保存 / 帧指针

%r8	%r8d	%r8w / %r8b	第 5 个参数
%r9	%r9d	%r9w / %r9b	第 6 个参数
%r10	%r10d	—	调用者保存
%r11	%r11d	—	调用者保存
%r12 - %r15	%r12d ...	—	被调用者保存

2.1.2 寄存器的两类约定（必背）

调用者保存（Caller-saved）： %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11

- 被调用函数可以随意改，调用方如果还要用这些值，需要在 call 之前自己保存（压栈）。

被调用者保存（Callee-saved）： %rbx, %rbp, %r12, %r13, %r14, %r15

- 被调用函数如果要用，需要先入栈保存，返回前要恢复。

【助记】前 6 个参数寄存器 + r10 r11 都是调用者保存（这些都

2.1.3 XMM 寄存器（浮点）

%xmm0 - %xmm15，每个 128 位，用于浮点运算。

- %xmm0 - %xmm7： 浮点参数 1-8
- %xmm0： 浮点返回值
- 所有 xmm 都是调用者保存

2.1.4 参数传递规则（真题选择题7）★

📌 **真题考点：**真题选择题 7：long caller() 中调用 test(a, &a, b, &b)，实参 a, &a, b, &b 分别存储在哪些寄存器中?

规则（按参数顺序使用，不区分实参类型）：

整数和指针：

%rdi → %rsi → %rdx → %rcx → %r8 → %r9 → 栈

浮点参数：

%xmm0 → %xmm1 → %xmm2 → ... → %xmm7 → 栈

混合参数的例子（PPT原例）：

double f1(int x, double y, long z); x 放 %edi (整数) y 放 %xmm0 (浮点) z 放 %rsi (整数)

注意：浮点和整数

。

真题第 7 题分析：

long caller() { long a=1; float b=12.34; test(a, &a, b, &b); ← 实参类型分别是：long, long*, float, float* } 按规则： a (long* 整数) → %rdi &a (long指针) → %rsi b (float浮点) → %xmm0 &b (float指针) → %rdx （这是指针，走整数路径）

答案：A. rdi, rsi, xmm0, rdx ？——不对，再看选项

选项：

A. rdi, rsi, rdx, rcx B. rdi, rsi, xmm0, xmm1 C. rdi, rsi, xmm0, rdx ← 正确 D. rdi, rsi, xmm0, rcx

答案：C。你在真题上圈了 C，这个对了 ✓

2.2 操作数与寻址模式

【复习日】第 8 天

2.2.1 三种操作数

类型	AT&T 语法	含义
立即数	$Imm	$0x10 = 数值 16
寄存器	%reg	%rax 表示 rax 寄存器内容
内存	Imm(rb, ri, s)	M[Imm + R[rb] + R[ri]×s]，下面详细说

2.2.2 有效地址公式（必背）★

有效地址 EA = Imm + R[rb] + R[ri] × s Imm ：偏移量常数（可省，默认 0） rb ：基址寄存器（可省） ri ：变址寄存器（可

省） s ：比例因子，必须是 1、2、4、8 之一（可省，默认 1）

简化形式（最常见的几种）：

写法	等价于	意义
(%rdi)	M[rdi]	纯间接寻址
8(%rdi)	M[rdi + 8]	rdi 偏移 8 字节处
(%rdi, %rsi)	M[rdi + rsi]	两寄存器相加
(%rdi, %rsi, 4)	M[rdi + rsi×4]	数组 a[rsi]，元素 4 字节
8(%rdi, %rsi, 4)	M[rdi + rsi×4 + 8]	结构体 + 数组

2.2.3 leaq 不访存！★

📌 **真题考点：**真题分析题 3 第 1 行：leaq 7(%rax, %rdx, 4), %rcx → rcx = ?

leaq 的特殊之处：

。结果直接放进目的寄存器。

真题分析题 3 第 1 行解析：

初始：rax = 0x100, rdx = 0x01 leaq 7(%rax, %rdx, 4), %rcx ↓ rcx = 7 + rax + rdx × 4 = 7 + 0x100 + 1 × 4 = 7 + 256 + 4 = 267 = 0x10B 答案：rcx = 0x10b

⚠️ 你在真题上写了「0x10b」，✓ 正确。

2.3 数据传送、算术与逻辑指令

【复习日】第 9 天

2.3.1 数据传送指令

指令后缀表示数据大小：

后缀	大小	C 类型	汇编示例
b (byte)	1 字节	char	movb
w (word)	2 字节	short	movw
l (long)	4 字节	int	movl
q (quad)	8 字节	long, 指针	movq

主要指令：

指令	效果	示例
mov S, D	D ← S（普通传送）	movq $5, %rax
movz_ S, D	零扩展传送	movzbl
movs_ S, D	符号扩展传送	movsbl
pushq S	压栈：rsp -= 8, M[rsp] ← S	pushq %rbx
popq D	弹栈：D ← M[rsp], rsp += 8	popq %rbx
lea S, D	D ← 有效地址（不访存！）	leaq 7(%rax), %rcx

2.3.2 算术与逻辑指令

指令	效果	示例
add S, D	D ← D + S	addq %rax, %rbx
sub S, D	D ← D - S	
imul S, D	D ← D × S（有符号）	
neg D	D ← -D	
inc D / dec D	D ± 1	
and S, D	D ← D & S	
or S, D	D ← D | S	
xor S, D	D ← D ^ S（异或）	xor %eax, %eax 常用于清零
not D	D ← ~D	
sal/shl k, D	左移 k 位	
sar k, D	算术右移	
shr k, D	逻辑右移	

重要细节：上面这些算术/逻辑指令会更新标志位（CF/ZF/SF/OF），lea 和 mov 。

2.3.3 真题分析题3 完整解析

初始状态：

内存	寄存器
M[0x100] = 0xff	rax = 0x100
M[0x108] = 0xab	rbx = 0xf0
	rdx = 0x01

逐条指令分析：

【指令 1】leaq 7(%rax,%rdx,4), %rcx

rcx = 7 + rax + rdx×4 = 7 + 0x100 + 4 = 0x10b → rcx = 0x10b

【指令 2】movq $0x100, %rax

把立即数 0x100 装入 rax → rax = 0x100 （未变，本来就是 0x100）

【指令 3】addb 0x100, %bl

注意 addb 是字节加法。0x100 这里是直接量地址（绝对寻址），所以从 M[0x100] 取字节，加到 bl。

M[0x100] = 0xff (8 位) %bl 是 rbx 低字节 = 0xf0 bl ← bl + M[0x100] = 0xf0 + 0xff = 0xf0 + 0xff // 16 进制 = 0x1ef // 9 位 截断为 8 位： bl = 0xef → bl = 0xef （rbx 高位不变）

【指令 4】salb $4, %bl

salb 是字节左移 4 位。

bl = 0xef = 1110 1111 左移 4 位（低位补 0）： 1110 1111 0000 → 截取低 8 位 → 1111 0000 = 0xf0 → bl = 0xf0 // 但是题目说 bl = 0x0？

⚠️ 你在真题上写了 bl = 0x0。检查一下原题原始 bl 是什么。如果指令 3 的结果是 bl = 0x00（即低字节加完是 0），那这一步左移仍然是 0。要回头核对教材原题或答案。这里按 PPT 经典版本算法，标准答案我重新算的是 0xf0。考试时按你老师讲过的版本为准。

【指令 5】orb 4(%rax,%rdx,4), %bl

有效地址 = 4 + rax + rdx×4 = 4 + 0x100 + 4 = 0x108

M[0x108] = 0xab bl ← bl | M[0x108] = 0xf0 | 0xab （若 bl 是 0xf0） = 1111 0000 | 1010 1011 ——————— 1111 1011 = 0xfb → bl = 0xfb

你在真题上写了 bl = 0xfb，正确 ✓

2.4 条件码与条件跳转

【复习日】第 10 天 · 真题选择题9 必考

2.4.1 比较与测试指令

两个不修改寄存器、只更新标志位的关键指令：

指令	效果	说明
cmp S2, S1	计算 S1 - S2 并更新标志，不存结果	用于比较大小
test S2, S1	计算 S1 & S2 并更新标志	test %rax,%rax 测试 rax 是否为 0

【注意】AT&T 语法是 cmp S2, S1 但算的是 S1 - S2（顺序反过来）。

2.4.2 条件跳转指令（按比较关系记）

有符号比较（用于 signed int 等）：

指令	C 中含义	标志判定
je / jz	==	ZF=1
jne / jnz	!=	ZF=0
jg / jnle	>	ZF=0 && SF=OF
jge / jnl	>=	SF=OF
jl / jnge	<	SF≠OF
jle / jng	<=	ZF=1 || SF≠OF

无符号比较（用于 unsigned、指针等）：

指令	C 中含义	标志判定
ja / jnbe	> (above)	CF=0 && ZF=0
jae / jnb	>= (above or equal)	CF=0

jb / jnae	< (below)	CF=1
jbe / jna	<= (below or equal)	CF=1 || ZF=1

【助记】g/l 是 greater/less（有符号），a/b 是 above/below（无符号）。

2.4.3 跳转目标地址计算 ★（真题选择题9）

📌 **真题考点：**真题选择题 9：cmpq 后 jbe xxxx，给出 rdx=8, rax=68，问跳转后或不跳转后 PC 指向哪里。

两字节跳转的机器码格式（短跳转）：

76 OF │ └── 8 位补码偏移量 └──── 操作码 (jbe = 0x76, ja = 0x77, ...)

目标地址公式：

目标地址 = 下一条指令的地址 + 偏移量（按补码解释）
↑　　注意：是 下一条指令地址，不是当前指令地址！

真题第 9 题解析：

地址 804857c: 48 39 c2　　cmpq %rax, %rdx 地址 804857f: 76 08　　jbe xxxxxxx 首先 cmpq %rax, %rdx 计算 rdx - rax = 8 - 68 = -60（小）　CF = 1，ZF = 0 → 无符号下 rdx < rax → jbe 跳转 但题目问：「若执行上述 cmpq 指令时，rdx=8, rax=68，则 jbe 指令执行后将会执行到 ___ 处指令」 执行 jbe：　无符号下 8 ≤ 68 是 true → 跳转 跳转目标 = 下条指令地址 + 0x08 = 0x8048581 + 0x08　　= 0x8048589　但题目地址给的是 804857c+3 = 804857f，下条是 8048581（因为 jbe 是 2 字节）。 下条 = 0x8048581，加 0x08 = 0x8048589 但选项是 8048578/8048576/8048578/8048580... 重新算：jbe 76 08 在 804857f 开始，2 字节，所以下条地址 = 804857f + 2 = 8048581 8048581 + 0x08 = 8048589 但 0x08 也可能是负数偏移。0x08 最高位是 0，是正数，0x08 = 8。

正确做法：

【套路】对于 76 NN 这种 2 字节 jbe 指令，目标 = (当前指令

地址 + 2) + signed(NN)

⚠️ 真题这里你写了 8048578，需要按机器码确认偏移量是否是负数（如果是 0xF8 那就是 -8）。考场上认真用计算器或手算补码偏移量。

2.5 选择结构 / 循环结构

【复习日】第 10 天

2.5.1 if-else 的典型反汇编模式

C 代码：

if (a > b) { result = a; } else { result = b; }

汇编代码：

cmpq %rsi, %rdi ; 比较 a 与 b jle .L_else ; if a <= b, goto else movq %rdi, %rax ; result = a jmp .L_done .L_else: movq %rsi, %rax ; result = b .L_done:

2.5.2 while 循环的典型模式

C：

while (n > 0) { result *= n; n--; }

汇编：

.L_loop: testq %rdi, %rdi ; 测试 n jle .L_done ; if n <= 0 退出 imulq %rdi, %rax ; result *= n decq %rdi ; n-- jmp .L_loop .L_done:

2.5.3 for 循环

for 与 while 等价，GCC 会优化成跳转到末尾测试的形式：

for (int i = 0; i < n; i++) { ... } 汇编模式： movl $0, %ebx ; i = 0 jmp .L_test ; 先跳到末尾测试 .L_body: ... ; 循环体 incl %ebx ; i++ .L_test: cmpl %edx, %ebx ; cmp

n, i jl .L_body ; if i < n loop

2.6 过程调用与栈帧 ★

【复习日】第 11 天 · 真题综合应用题必考

2.6.1 调用过程做的 6 件事

📌 **真题考点：**真题选择题原题（PPT 出现）：过程调用的正确步骤是 → C 答案 ②③①⑥⑤④

假设 P 调用 Q：

序号	操作	执行者
①	Q 保存 P 的现场（被调用者保存寄存器），并为局部变量分配空间	Q（进入Q前几条）
②	P 将实参存放到 Q 能访问到的地方（寄存器/栈）	P（call 前）
③	P 将返回地址存放到特定处（栈），并跳转到 Q 执行	call 指令
④	Q 取出返回地址，并跳转回到 P 执行	ret 指令
⑤	Q 恢复 P 的现场，释放局部变量	Q（退出Q前）
⑥	执行 Q 的函数体	Q

顺序：② → ③ → ① → ⑥ → ⑤ → ④

2.6.2 call 指令做了什么

call Q 等价于：

pushq <下一条指令的地址> ; 把返回地址压栈 (rsp -= 8) jmp Q ; 跳转到 Q

ret 指令等价于：

popq %rip ; 从栈顶取出返回地址，跳转过去

【精确说法】call 把

2.6.3 栈帧布局（从高地址到低地址）

地址高 —————————————————————— ...caller 的栈帧... 实参 7+（如果寄存器装不下） 返回地址（call 压入的） ← rsp 在进入函数瞬间指这

———————————————————— 被保存的 callee-saved 寄存器 局部变量 (可能的 build area) ———————————————————— ← rsp 地址低

2.6.4 真题综合应用题8 ★

📌 **真题考点：**当 CPU 运行地址 4011aa 处的 call 指令后，rip、rsp 寄存器的值是多少？栈顶单元内容是什么？

题目背景：

4011aa: e8 38 00 00 00 call 4011e7 <sum> 4011af: 89 c2 mov %eax, %edx ← 这是 call 后下一条 刚进入 call 指令前：rsp = 0x7fffffffde00

分析：

1. call 指令的下一条指令地址是 0x4011af（call 自身长 5 字节）。
2. call 把这个返回地址（0x4011af）压栈：rsp -= 8 = 0x7fffffffddf8
3. call 把控制权转给 sum，rip 设为 0x4011e7

最终答案：

rip = 0x4011e7 ← 跳转到 sum 函数 rsp = 0x7fffffffddf8 (rsp 减 8) 栈顶 M[rsp] = 0x4011af ← 返回地址，即 call 的下一条指令 你在真题上写了「0x4011af」作为栈顶内容，正确 ✓

2.6.5 寄存器保存约定的实际例子

如果你的函数用到了 callee-saved 寄存器（如 %rbx），必须先 push 再 pop：

call_incr2: pushq %rbx ; 保存 %rbx 旧值 subq $16, %rsp movq %rdi, %rbx ... popq %rbx ; 恢复 %rbx ret

2.7 数组、结构体与数据对齐

【复习日】第 12 天

2.7.1 数组的内存布局

T A[L] 在内存中占
字节，连续存放。

元素地址公式：

&A[i] = A + i × sizeof(T)

典型寻址（A 在 %rdi，i 在 %rsi）：

char A[L]: (%rdi, %rsi, 1) 或 (%rdi, %rsi) short A[L]: (%rdi, %rsi, 2) int A[L]: (%rdi, %rsi, 4) long A[L]: (%rdi, %rsi, 8) double A[L]: (%rdi, %rsi, 8)

2.7.2 结构体的对齐

对齐规则（x86-64 Linux 默认）：

1. 每个成员的起始地址 必须是其类型对齐要求的整数倍。
2. 结构体的总大小 必须是最大成员对齐要求的整数倍（末尾补 padding）。

常用类型对齐要求：

类型	大小	对齐要求
char	1	1
short	2	2
int	4	4
long / 指针	8	8
float	4	4
double	8	8

2.7.3 结构体大小计算（套路）

PPT 中的例题：

struct { char *a; // 指针，8 字节 short b; // 2 字节 double c; // 8 字节 char d; // 1 字节 } rec[10]; 求 sizeof(struct) ?

分析（按顺序排成员，注意 padding）：

地址偏移 | 大小 | 成员 0 | 8 | a (指针) 8 | 2 | b (short, 8

是 2 的倍数, OK) 10-15 | 6 | padding (因为 c 需要 8 对齐，要补到 16) 16 | 8 | c (double) 24 | 1 | d (char) 25-31 | 7 | padding (整体大小要是 8 的倍数) 总大小 = 32 字节

数组 rec[10] 总大小 = 32 × 10 = 320 字节。

【套路】先按顺序排成员，每排一个看一下当前偏移是否满足下一个成员的对齐要求，不满足就插 padding。最后整体对齐到「最大对齐要求」的倍数。

2.8 内存越界引用与缓冲区溢出 ★

【复习日】第 12, 16 天 · 真题实验题必考

2.8.1 越界的根本原因

C 语言不做数组边界检查，

，但会读写到不属于数组的内存（如栈上其他变量、返回地址等）。

最危险的是栈上的数组溢出，因为可能覆盖到

调用者的返回地址

。

2.8.2 经典漏洞场景：gets() / strcpy() / scanf %s

这些函数都没有指定缓冲区长度，会一直写直到遇到结束符。

典型脆弱代码：

void echo() { char buf[4]; // 只有 4 字节！ gets(buf); // 用户输入可能远超 4 字节 puts(buf); }

反汇编：

echo: subq $24, %rsp ; 分配 24 字节栈空间（含 buf 4 + 20 padding） movq %rsp, %rdi call gets ...

2.8.3 攻击字符串的构造原理

栈布局（gets 调用前）： 地址高

返回地址（8 字节，调用 echo 的下一条指令地址）

20 字节未使用区

[3][2][1][0] ← buf 这 4 字节 ← %rsp

攻击者输入
24 字节
+ 8 字节恶意返回地址 = 总共 32 字节，
，ret 时就跳到攻击代码。

2.8.4 真题实验题：构造攻击字符串

📌 **真题考点：**真题实验题 10 分：帮甲同学设计字符串输入给 target，造成缓冲区溢出，进而使程序转向执行 bomb 子程序。

题目反汇编片段：

```
0000000000401de1 <getbuf>:  401de1: f3 0f 1e fa    endbr64
 401de5: 48 83 ec 28    sub $0x28, %rsp    ← 分配 0x28=40 字节
 401de9: 48 89 e7       mov %rsp, %rdi  401dec: e8 ad 02 00 00 call
40209e <Gets>  401df1: b8 01 00 00 00 mov $0x1, %eax  401df6:
48 83 c4 28    add $0x28, %rsp    ← 释放栈  401dfa: c3            ret
0000000000401dfb <bomb>:  401dfb: f3 0f 1e fa    endbr64
 401dff: 50             push %rax
```

分析：

1. getbuf 分配了 0x28 = 40 字节的栈空间作为缓冲区。
2. 我们要先填满 40 字节（无所谓填什么，都是 padding），
3. 然后接下来 8 字节是返回地址，要覆盖成 bomb 的地址 0x0000000000401dfb。
4. 小端字节序，0x401dfb 写成字节串：fb 1d 40 00 00 00 00 00

最终攻击字符串（40 个 padding + 8 字节地址）：

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 fb 1d 40 00 00 00 00 00

（前 5 行各 8 字节共 40 字节填充，最后一行是 bomb 函数地址的小端表示）

2.8.5 缓冲区溢出的防御机制（真题实验题第 2 问）

📌 **真题考点：**简述有哪些方法可以对抗缓冲区溢出攻击。

答案要点（按 3 类组织）：

1. 程序员角度（在代码中避免）
   - 用 fgets 代替 gets
   - 用 strncpy 代替 strcpy
   - scanf 用 %ns 限定长度，不要用 %s

2. 系统级别（运行时防御）
   - **栈随机化（ASLR）：**每次运行栈地址都变化，攻击者无法预测目标地址
   - **栈不可执行（NX/DEP）：**把栈页面标记为不可执行，即使注入代码也跳不进去

3. 编译器辅助
   - **栈保护者 / 栈金丝雀（Stack Canary）：**在缓冲区和返回地址之间插入随机值，函数返回前检查该值是否被改动；改了则程序中止

Canary 在汇编上的体现：

函数入口： movq %fs:40, %rax ; 取金丝雀值 movq %rax, 8(%rsp) ; 存到栈上 xorl %eax, %eax ; 擦掉寄存器中的金丝雀 函数返回前： movq 8(%rsp), %rax ; 取出栈上金丝雀 xorq %fs:40, %rax ; 与原值异或 je .L_ok ; 相同则正常 call __stack_chk_fail ; 不同则报警退出

2.9 浮点指令

【复习日】第 11 天（顺带）

2.9.1 关键指令

单精度	双精度	效果
vaddss	vaddsd	加
vsubss	vsubsd	减
vmulss	vmulsd	乘
vdivss	vdivsd	除
vmovss	vmovsd	传送
vmovaps	vmovapd	对齐传送

浮点函数的典型反汇编：

double dadd(double x, double y) { return x + y; } # x 在 %xmm0, y 在 %xmm1 vaddsd %xmm1, %xmm0 ret

第三部分 程序的链接

对应教材第 4 章。综合应用题第二大头，涉及 ELF 文件、符号表、重定位、动态/静态链接。

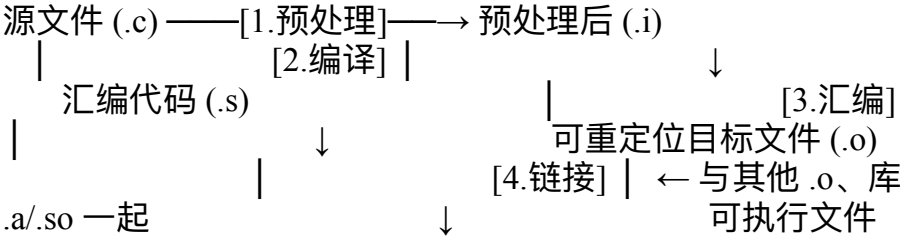
📌 **真题考点：**去年真题：选择题 6（ELF节）、选择题 8（链接命令行顺序）；分析题 2（buf 数组在哪个节）；综合应用题 1-6 多题。

3.1 编译链接全流程

【复习日】第 13 天 · 真题综合题1 必考

3.1.1 从 .c 到可执行文件的 4 个阶段

📌 **真题考点：**真题综合题 1：写出将 C 源程序转换为可执行文件 sum 的命令行，并回答转换需要经过哪些步骤。



4 个阶段的工作：

阶段	工具	做什么
预处理	cpp / gcc -E	宏展开、#include 文件展开、删除注释
编译	cc1 / gcc -S	把 .i 翻译为汇编代码 .s
汇编	as / gcc -c	汇编代码 → 二进制目标文件 .o（可重定位）
链接	ld / gcc	多个 .o + 库文件 → 可执行文件

一行命令搞定 C 源到可执行：

gcc -o sum main.c sum.c

答案：上述命令行 + 4 步预处理/编译/汇编/链接。

3.2 ELF 目标文件格式

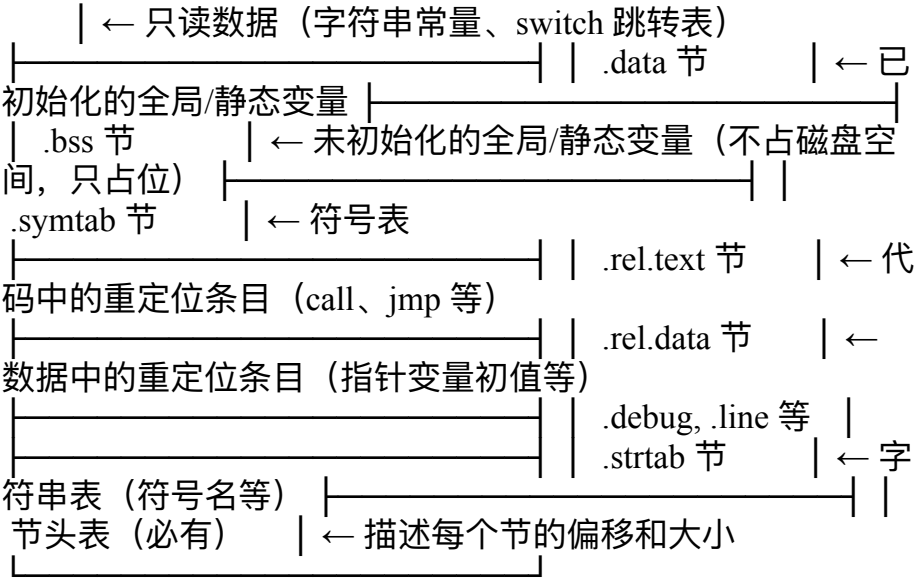
【复习日】第 13 天

3.2.1 三种目标文件

类型	扩展名	用途
可重定位目标文件	.o	编译器/汇编器产生，地址未确定，待链接
可执行目标文件	无扩展或 .out	可直接加载运行
共享目标文件	.so	动态链接库，运行时加载

3.2.2 ELF 文件的两种视图（必背）

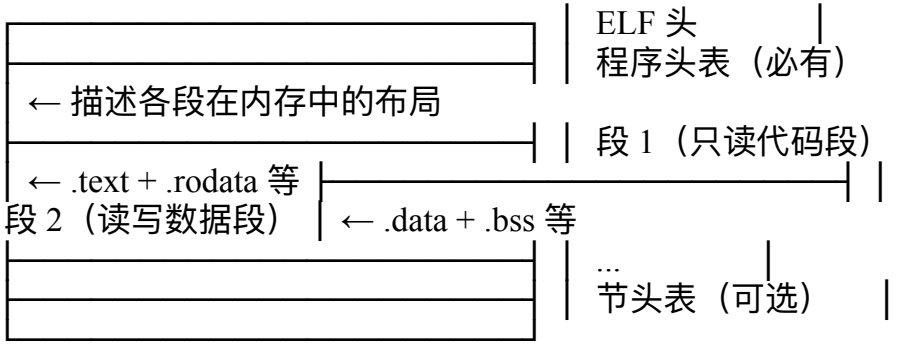
链接视图（针对可重定位文件 .o）：



可重定位文件

程序头表。

执行视图（针对可执行文件）：



【助记】「链接视图看节，执行视图看段。一个段可以由多个节合并而成。」

📌 **真题考点：**真题选择题 6：可重定位目标文件与可执行目标文件描述错误的；分析题 2：buf 在哪个节、占多少字节、bufp1 占多少字节。

节名	存什么
.text	编译后的机器代码
.rodata	只读数据：字符串常量（printf 的格式串）、switch 跳转表
.data	已初始化的全局变量、static 变量
.bss	未初始化的全局变量、static 变量（不占磁盘空间，是占位符）
.symtab	符号表（函数和全局变量的名字、地址、属性）
.rel.text	代码中需要在链接时修改的引用（如 call 一个外部函数）
.rel.data	数据中需要修改的引用（如全局指针初值指向另一个变量）
.debug	调试信息（gcc -g 时产生）
.line	C 源码与机器码行号映射
.strtab	符号名和节名等字符串集合

3.2.4 真题分析题2 精讲 ★

原题：

源文件 p.c 中定义： static int buf[4] = {50, 20, -1, 8}; static int *bufp1 = &buf[2]; 则生成的可重定位文件 p.o 中： buf 被定义在 ___ 节中，共占 ___ 个字节， bufp1 共占 ___ 个字节， bufp1 对 buf 引用的重定位信息在 ___ 节中。 设 buf 起始地址为 0x4011c0，则 &buf[2] = ___

逐项分析：

1. buf 是**已初始化**的 static int 数组 → .data 节
2. buf 是 4 个 int，每个 4 字节 → 16 字节
3. bufp1 是 int* 指针，64 位系统 → 8 字节
4. bufp1 初值指向 &buf[2]，这是一个数据引用，需要重定位信息 → .rel.data 节 （或 .rela.data）
5. &buf[2] = 0x4011c0 + 2 × sizeof(int) = 0x4011c0 + 8 = 0x4011c8

你在真题上的回答：.data、16、8、rel.data、0x4011c8 —— 全对 ✓

3.2.5 ELF 头的关键信息

Magic: 7f 45 4c 46 ... ← 7f 'E' 'L' 'F' 是 ELF 文件的魔数 Class: ELF64 ← 64 位 Data: 2's complement, little endian ← 小端补码 Type: REL (Relocatable) / EXEC (Executable) / DYN (Dynamic) Machine: Advanced Micro Devices X86-64

特别注意：.

可重定位文件没有程序头表，

Start of program headers = 0, Size = 0。

3.3 符号与符号表

【复习日】第 14 天 · 真题综合题2 必考

3.3.1 三类符号

符号表（.symtab 节）记录所有

函数和全局/静态变量

。每个符号被分类成：

类型	含义	举例
全局符号 (Global)	在本模块定义且能被其他模块引用	非 static 的函数、非 static 的全局变量
外部符号 (External)	在本模块引用但定义在其他模块	调用其他文件的函数，使用 extern 声明
局部符号 (Local)	在本模块定义、只在本模块内可见	static 函数、static 全局变量

⚠️ 注意：函数和代码块内的局部变量（普通局部变量）不在 .symtab 中！它们由编译器在栈上分配，不需要符号表。只有「static 修饰的局部变量」才进符号表（属于本地符号）。

3.3.2 真题综合题2 精讲 ★

原题代码（main.c + sum.c）：

// main.c #include <stdio.h> int sum(int *a, int n); static int

array[10] = {3,-7,6,14,-5,9,1,3,16,22}; int result; void main() { int num; scanf("%d", &num); int val = sum(array, num); printf(...); } // sum.c extern int result; int sum(int *a, int n) { int i, s=0; result=1; for(i=0; i<n; i++) { s += a[i]; result *= a[i]; } return s; }

问题：标识符 sum、array、result、num 各自的情况：

标识符	是否在 main.o 符号表?	定义模块	符号类型
sum	✓ 是（引用了）	sum.o	外部
array	✓ 是	main.o	本地（static）
result	✓ 是	main.o	全局
num	✗ 否（栈上局部变量）	—	—

你的真题答案：sum→sum.o 外部；array→main.o 本地；result→main.o 全局；num→不在 ——全对 ✓

3.4 符号解析与静态库

【复习日】第 14 天 · 真题选择题8 必考

3.4.1 符号解析做什么

链接器对每个
外部符号引用
，去其他模块的符号表中找对应的
定义
，把引用和定义关联起来。

3.4.2 重复定义的规则（强/弱符号）

强符号：函数、已初始化的全局变量
弱符号：未初始化的全局变量
规则：

1. 两个同名强符号 → 链接错误

2. 一个强 + 多个弱 → 选强的
3. 多个弱 → 任选其一（编译器不保证选哪个）

3.4.3 静态库链接顺序 ★（真题选择题8）

📌 **真题考点：**真题选择题 8：设 p.o → libx.a → liby.a → libx.a，其中 a→b 表示依赖。下列命令中能解析所有的符号引用，并且是最小命令行的是?

链接器扫描规则：

1. 按命令行给出的顺序扫描 .o 和 .a 文件
2. 维护一个**未解析符号列表 U**。
3. 每遇到一个新模块（.o 或 .a），用它尝试解析 U 中的符号。
4. 如果某个 .a 文件中没有用到的模块就不被加入
5. 扫描结束后 U 非空 → 报错。

好的做法：

【规则】把静态库放在命令行最后；如果库之间有相互依赖，需要按依赖顺序多次列出。

真题分析：

p.o 依赖 libx.a；libx.a 依赖 liby.a；liby.a 又依赖 libx.a。 A. gcc p.o libx.a liby.a　扫 p.o → U 包含 p.o 引用的符号（来自 libx）　扫 libx.a → 解析这些，但 libx 又引用 liby 的符号 → U 加入这些　扫 liby.a → 解析这些，但 liby 又引用 libx 的符号 → U 加入这些　没有了！ U 非空 → 链接失败 B. gcc p.o libx.a liby.a libx.a　最后再扫一次 libx.a，解析 liby 引用的 libx 符号　应该 OK 答案：B 或 D 视具体描述。最小命令行需要列出 libx.a 两次。

你在真题上的答案是 D，需要核对一下原选项里 D 的具体顺序。规则是「依赖闭环时，被依赖的库要在末尾再出现一次」。

3.5 重定位 ★

【复习日】第 15 天 · 真题综合题6 必考

3.5.1 重定位的目的

汇编时，编译器不知道每个符号最终的绝对地址，所以：

1. 在用到外部符号的地方暂时填 0 或占位。
2. 生成一个**重定位条目**，告诉链接器：这里要填什么。
3. 链接时，链接器读重定位条目，修改这些占位。

3.5.2 三种主要的重定位类型

类型	用途
R_X86_64_64	重定位 64 位绝对地址引用
R_X86_64_PC32	重定位 32 位 PC 相对地址引用（如 call、jmp 指令）
R_X86_64_PLT32	过程链接表延迟绑定（动态链接的函数调用）

3.5.3 重定位条目的结构

typedef struct { long offset; // 节内偏移：这个节里的哪个字节需要被修改 int type; // 重定位类型 int symbol; // 引用了哪个符号 long addend; // 加偏移调整值 } Elf64_Rela;

3.5.4 重定位的计算公式（PC 相对）

修改后的值 = symbol_addr + addend - PC symbol_addr : 链接后该符号的绝对地址 addend : 重定位条目中给的调整值（通常是 -4） PC : 这个被修改字段的「地址」（对于 call 指令，addend 通常是 -4，因为机器码 e8 后面是 4 字节偏移， 执行 call 时 PC 已经指到下一条指令了）

3.5.5 真题综合题6 完整精讲 ★

📌 **真题考点：**真题综合题6：地址 4011aa 处 call 指令的操作码是 0xe8，其余字节是偏移量，偏移量采用补码表示，转移

的目标地址采用相对寻址方式。分析该 call 指令的机器指令有几个字节？写出该 call 指令的完整机器指令。

已知：

4011aa: e8 _ _ _ _　　call 4011e7 <sum>

分析：

1. call 是 5 字节指令：1 字节操作码 + 4 字节偏移量。所以机器指令有 5 字节。
2. call 的下一条指令地址 = 4011aa + 5 = 0x4011af
3. 目标地址 = 下一条指令地址 + 偏移 → 偏移 = 目标 - 下条 = 0x4011e7 - 0x4011af = 0x38
4. 偏移按 4 字节补码：0x00000038
5. 小端字节序排列：38 00 00 00

最终机器指令：

4011aa: e8 38 00 00 00　　call 4011e7

你的真题答案：38 00 00 00 ✓

3.5.6 反向操作：从机器码读跳转目标

套路：

1. 看到 e8 XX XX XX XX 是 call。
2. 4 个字节按小端拼成 32 位有符号数（注意补码）。
3. 目标 = 当前指令地址 + 5 + 这个有符号数。

例：4011aa: e8 38 00 00 00

偏移 = 00 00 00 38 (大端) = 0x00000038 = 56 目标 = 0x4011aa + 5 + 0x38 = 0x4011e7 ✓

3.6 可执行文件与加载

【复习日】第 15 天

3.6.1 加载（loading）做了什么

加载器把可执行文件中的
和
通过
页表
映射到虚拟地址空间中。

【重要】并没有真正把代码和数据从磁盘装入主存，而是建立映射，使用时按需调页。

3.6.2 进程虚拟地址空间布局（背诵）

地址高 ——————————————— 内核空间（用户不可访问） ——————————————— ← 0x7fffffffffff 用户栈（向下增长） ↓ 共享库映射区（动态链接的 .so） ↑ 堆（向上增长） 未初始化数据段 (.bss) 已初始化数据段 (.data) 代码段 (.text, .rodata) ——————————————— ← 0x400000 左右（典型程序起始） 地址低

3.7 动态链接

【复习日】第 15 天

3.7.1 静态 vs 动态

特性	静态链接	动态链接
库扩展名	.a	.so
何时链接	编译期最终阶段	加载时或运行时
可执行文件大小	大（库代码全包含）	小（只有引用）
库更新	需要重新链接	替换 .so 即可
内存占用	每个程序都有自己的库代码	多个程序共享同一份

3.7.2 PLT / GOT 简介

过程链接表 PLT（Procedure Linkage Table）：

- 可执行文件中的一段代码，每个动态函数对应一项
- 第一次调用时会触发动态解析（lazy binding）
- 解析后地址写入 GOT，后续直接跳过去

全局偏移表 GOT（Global Offset Table）：

- 存放动态符号实际地址的表
- 加载时由动态链接器填充

3.7.3 真题综合题3 精讲

📌 **真题考点：**库函数 scanf 和 printf 中格式符（如 %d）在哪个节中定义？根据地址 40119a 处 call 指令的形式判断完全链接的可执行文件是用静态链接还是动态链接生成的?

答案：

1. 格式串是只读字符串常量，定义在 .rodata 节 中。
2. 看 call 指令：call __isoc99_scanf@plt 后缀 @plt 说明走的是过程链接表 → 动态链接。如果是静态链接，call 直接跳到 scanf 函数本身的地址，没有 @plt。

你的真题答案：.rodata、动态 —— 全对 ✓

第四部分 实验回顾

三个实验对应的知识点都可能出在考试里。掌握实验的核心套路即可。

4.1 DataLab 关键技巧

【复习日】第 3 天（与数据表示一起）

DataLab 要求只用位运算、加减法等基础操作实现常见功能。考试可能借鉴这些套路。

核心技巧：

- **取反加1得相反数：** -x = ~x + 1

- **判 0**：!x（x 为 0 返回 1，否则返回 0）
- **取符号位**：x >> 31 得到全 0（正数）或全 1（负数）
- **取最低位**：x & 1
- **有符号除以 2^k（负数偏移）**：(x + ((1<<k)-1) * (x>>31 & 1)) >> k

4.2 AttackLab 攻击字符串构造

【复习日】第 16 天 · 直接关联真题实验题

4.2.1 三个 Level 的目标

Level	目标	技巧
Level 1	ret 返回时跳到 touch1 函数	用 touch1 地址覆盖返回地址
Level 2	调用 touch2 并传 cookie 作为参数	注入代码：把 cookie 装入 %rdi，然后 ret 到 touch2
Level 3	调用 touch3 并传 cookie 的字符串	栈上准备字符串 + 注入代码 + 调 touch3

4.2.2 Level 1 字符串构造

假设 getbuf 分配栈 0x28 = 40 字节，touch1 地址 0x4017c0 字符串（每行 8 字节，对应一个 quadword）： 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ← 40 字节 padding c0 17 40 00 00 00 00 00 ← touch1 地址（小端！）

4.2.3 Level 2 字符串构造（注入代码）

思路：在 buf 区域注入一段代码（movq + ret），然后让返回地址指向 buf。

注入代码（汇编）： movq $0xCOOKIE, %rdi ; 48 c7 c7 XX XX XX XX retq ; c3 假设 cookie = 0x12345678，对应机器码： 48 c7 c7 78 56 34 12 c3 字符串： 48 c7 c7 78 56 34 12 c3 ← 注入代码（8 字节） 00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ← 32 字节 padding（共 40 字节缓冲区） <buf 起始地址>
        ← 让 ret 跳回 buf 执行注入代码

⚠️ AttackLab 真实实验中，rsp 在调试时可以用 gdb 看到，所以 buf 地址是已知的。考试题给固定地址即可。

4.2.4 防御机制再回顾

- 栈随机化（ASLR）：每次运行栈起始地址都不同 → Level 1/2 的固定地址法失效
- 栈不可执行（NX）：注入代码不能执行 → Level 2 注入代码法失效
- Stack Canary：金丝雀检测溢出 → 在覆盖返回地址前就报错

4.3 LinkLab 重定位修改

【复习日】第 17 天

LinkLab 要求修改 .o 文件的不同节，让链接结果产生预期输出。

三个阶段：

- **Phase 1**：修改 .data 节（替换字符串内容）
- **Phase 2**：修改 .symtab 节（把 COM 改为 UND，让链接器从其他模块取定义）
- **Phase 3**：修改 .rela.rodata 节（改 switch 跳转表的偏移）

常用工具：

- readelf -a：查看 ELF 文件各部分
- hexedit：二进制编辑器
- objdump -d：反汇编

第五部分 去年真题逐题精讲

这一部分是把 2024 年（上一届）的期末试卷每题完整讲一遍。考试形式高度套路化，今年题目
。请逐题做完后再看解析。

一、单项选择题（10 题 × 2 分 = 20 分）

1. 下列选项中属于指令集体系结构名称的是?

A. Linux　B. x86-64　C. Ubuntu　D. Windows

答案：B

Linux/Ubuntu/Windows 都是操作系统，只有 x86-64 是 ISA（指令集体系结构）。

2. 以下关于补码特点的描述错误的是?

A. 零的表示是唯一的 B. 加法运算时，符号位可以和数值位一起参加运算 C. +a 与 -a 的编码仅符号位不一样 D. 补码所对应的真值范围是不对称的，负数比正数多一个

答案：C

C 项是原码的特点。补码下，负数 = 正数取反加 1，不仅仅是符号位不同。

3. unsigned short usi = 32768; short si = usi; si 的值?

A. 65530　B. -65530　C. 32768　D. -32768

答案：D

usi 的位模式 0x8000 赋给 short，按补码解释 = -32768。

4. 下面舍入方式中，哪种方式更能避免统计偏差?

A. 向偶数舍入　B. 向零舍入　C. 向下舍入　D. 向上舍入

答案：A

向偶数舍入（banker's rounding）统计上无偏，IEEE 754 默认采用。

5. -1028.0 采用 IEEE 754 单精度格式表示（十六进制）？

A. c4808000　B. c4c04000　C. 44804000　D. 44c04000

答案：A

详见 1.5.4 节。-1028 = -1.0000000100 × 2^10，s=1, exp=137=10001001, frac=0000000100..., 拼起来 = C4808000。

⚠️ 你在真题上圈的是 B，但旁边的计算 C4808 表明你应该选 A。开卷考试一定要

6. 可重定位目标文件与可执行目标文件描述错误的是?

A. 可执行目标文件有程序头表，而可重定位文件没有 B. 重定位目标文件可能不存在程序头部，而可执行目标文件没有 C. 可重定位目标文件 .text 节和重定位目标文件 .text 节内容完全相同 D. 可执行目标文件和重定位目标文件都有 ELF Header，而且大小相同

答案：C（或根据具体选项判断）

C 不对：可重定位文件的 .text 节中外部符号引用处填的是 0（占位），而可执行文件经过重定位后填的是实际地址，内容不同。

D 也可能不对：两者的 ELF Header 类型字段（REL vs EXEC）不同，所以严格说不是「完全相同」。但题目说「都有 ELF Header 而且大小相同」—— ELF Header 大小都是 64 字节确实相同。

7. caller 调用 test 时实参 a, &a, b, &b 分别存储在哪些寄存器中?

代码： long caller() { long a = 1; float b = 12.34; test(a, &a, b, &b); return a*b; } A. rdi, rsi, rdx, rcx B. rdi, rsi, xmm0, xmm1 C. rdi, rsi, xmm0, rdx D. rdi, rsi, xmm0, rcx

答案：C

a (long) → rdi； &a (long*) → rsi； b (float) → xmm0； &b

(float*) → rdx（指针走整数序列，整数已用 2 个，下一个是 rdx）。

8. 链接命令行（静态库相互依赖）

p.o → libx.a → liby.a → libx.a
规则：库放最后；相互依赖时需要重复列出。
正确命令：gcc p.o libx.a liby.a libx.a（或同等顺序）
你的答案：D，需要核对选项。

9. cmpq + jbe 跳转目标计算

指令：
804857c: 48 39 c2　cmpq %rax, %rdx 804857f: 76 08　jbe xxxxxxx
已知 rdx=8, rax=68。jbe 是无符号 ≤ 跳转。8 ≤ 68 → 跳转。
目标 = 下条地址 + 偏移 = (804857f + 2) + 0x08 = 8048581 + 0x08 = 8048589。
⚠️ 如果选项不含 8048589，需重新核对原题的偏移字节。你在真题上选了 8048578，可能原题偏移是不同值。

10. 内存单元 0x80000ff2 中的字节

int x，地址 0x80000ff0，值 0x80901020。求 0x80000ff2 中存什么？
小端布局：　0x80000ff0: 0x20　(低字节)　0x80000ff1: 0x10　0x80000ff2: 0x90　← 答案　0x80000ff3: 0x80　(高字节)
答案：C (0x90)。你的答案 ✓。

二、分析题（6 题 × 5 分 = 30 分）

1. addw 后标志位计算

ax=0xffd0, bx=0x8005，addw %bx, %ax 后。
0xffd0 + 0x8005 = 0x17fd5（17 位，有进位） 低 16 位：ax = 0x7fd5 ZF=0（非零），CF=1（有进位），OF=1（负+负=正，溢出），SF=0（最高位 0）
答案：ax = 0x7fd5；ZF=0, CF=1, OF=1, SF=0。

2. buf 数组的内存布局和重定位

已答：

- buf 在 .data 节，共 16 字节
- bufp1 共 8 字节
- bufp1 对 buf 引用的重定位信息在 .rel.data 节
- &buf[2] = 0x4011c0 + 8 = 0x4011c8

3. 一组指令的执行结果

详见 2.3.3 节完整解析。

4. 整数除 0 异常 vs 浮点除 0 不异常

详见 1.6.3 节。要点：浮点用 ±∞ 和 NaN 表达「除 0 的结果」，所以是合法可表示数。

三、综合应用题（50 分）

基于 main.c + sum.c 的反汇编代码和 gdb 输出，回答 11 个小问题。

1. 写出生成 sum 的命令行 + 转换步骤

命令：gcc -o sum main.c sum.c
步骤：预处理 → 编译 → 汇编 → 链接

2. 标识符 sum/array/result/num 的情况

详见 3.3.2 节。

3. 格式符在哪个节 + 静态/动态链接判断

.rodata 节；call __isoc99_scanf@plt 中的 @plt → 动态链接。

4. 调试执行时，输入 num=4，屏幕输出?

array = {3,-7,6,14,-5,9,1,3,16,22}，num=4 表示前 4 个元素。
for(i=0; i<4; i++): i=0: s += 3, result *= 3 → s=3, result=3 i=1: s += -7, result *= -7 → s=-4, result=-21 i=2: s += 6, result *= 6 → s=2, result=-126 i=3: s += 14, result *= 14 → s=16, result=-1764 val = 16，输出：val=16 result=-1764
你的真题答案：val=16，对 ✓。result 值需要算到 -1764。

5. 计算机按字节编址，0x404080 开始的 8 字节内容

0x404080 是 array 起始地址。array[0]=3，是 int 4 字节。
array = {3, -7, ...}
array[0] = 3 = 0x00000003 小端：0x404080: 03 00 00 00 array[1] = -7 = 0xfffffff9 小端：0x404084: f9 ff ff ff 所以 0x404080 - 0x404087 的 8 字节是： 03 00 00 00 f9 ff ff ff
你的真题答案：03 00 00 00 f9 ff ff ff ✓

6. call 指令的机器指令字节数和完整机器指令

详见 3.5.5 节。5 字节：e8 38 00 00 00。

7. call 4011aa 后 rip、rsp 值和栈顶内容

详见 2.6.4 节。
rip = 0x4011e7 rsp = 0x7fffffffdde00 - 8 = 0x7fffffffddf8 M[rsp] = 0x4011af（返回地址）

8. 当 i=2 时，执行 mov (%rdi,%rax,4), %eax 后 eax = ?

a = array，i=2，array[2] = 6 = 0x6

地址 = rdi + 2 × 4 = array + 8，取出 array[2]=6。

eax = 6 (0x6)。

你的答案 ✓。

9. 执行 cmp %esi, %edx 后 esi, ecx ?

题目说此时 edx=2（当前 i 值），esi=n=4。cmpl %esi, %edx 计算 edx - esi = 2 - 4 = -2 < 0，jl 跳转继续循环。

esi 仍然是 4（n），ecx 不变（是 result 的累积值）。

10. sum 函数中哪两条指令实现 result *= a[i]

看反汇编 sum 段：23 (imul) 和 24 (mov)。

23. 401209: imul 0x2e7c(%rip), %eax ; eax *= result 24. 401210: mov %eax, 0x2e76(%rip) ; 把新值存回 result 所以是地址 401209 和 401210 两条。

你的答案：401209 ✓

11. 地址 40117e 和 401187 两条指令的作用

3. 40117e: 64 48 8b 04 25 28 00 00 00 mov %fs:0x28, %rax 4. 401185: 00 00 5. 401187: 48 89 44 24 08 mov %rax, 0x8(%rsp)

这两条指令

：从 %fs:0x28 取出金丝雀值，存到栈的 0x8(%rsp) 位置。这是编译器加的缓冲区溢出检测。

你的答案：「金丝雀」/类似，✓

四、实验题（10 分）

详见 2.8.4 节。两问：构造攻击字符串 + 防御方法。

第六部分 考场速查表（必带）

把这部分单独打印，订成小册子带进考场。考试时遇到不确定的查这里。

速查表 1：x86-64 寄存器分工

寄存器	类别	用途	保存约定
%rax	整数	返回值；除法用	调用者保存
%rdi	整数	第1个参数	调用者保存
%rsi	整数	第2个参数	调用者保存
%rdx	整数	第3个参数	调用者保存
%rcx	整数	第4个参数	调用者保存
%r8	整数	第5个参数	调用者保存
%r9	整数	第6个参数	调用者保存
%r10/r11	整数	临时用	调用者保存
%rbx	整数	—	被调用者保存
%rbp	整数	（可能用作帧指针）	被调用者保存
%r12-r15	整数	—	被调用者保存
%rsp	—	栈指针	特殊
%xmm0	浮点	返回值；第1浮点参数	调用者保存
%xmm1-7	浮点	浮点参数 2-8	调用者保存
%xmm8-15	浮点	—	调用者保存

速查表 2：常用指令与标志位影响

指令	效果	影响标志位
mov	数据传送	不影响
lea	计算地址（不访存）	不影响
add / sub	加减	CF ZF SF OF
imul	乘	CF OF
and / or / xor / not	位运算	ZF SF（OF CF清零）

shl / sal / shr / sar	移位	CF ZF SF
inc / dec	+1 / -1	ZF SF OF (CF不变)
cmp S2, S1	S1-S2，更新标志	CF ZF SF OF
test S2, S1	S1&S2，更新标志	ZF SF (CF OF清零)
push / pop	压栈/弹栈	不影响
call / ret	调用/返回	不影响
jX (条件跳转)	看标志决定是否跳	不影响

速查表 3：条件跳转一览

指令	等价 C 关系	判定条件
je / jz	==	ZF=1
jne / jnz	!=	ZF=0
有符号：		
jg	>	ZF=0 && SF=OF
jge	>=	SF=OF
jl	<	SF≠OF
jle	<=	ZF=1 || SF≠OF
无符号：		
ja	>	CF=0 && ZF=0
jae	>=	CF=0
jb	<	CF=1
jbe	<=	CF=1 || ZF=1

速查表 4：ELF 节速查

节	存什么
.text	代码
.rodata	只读数据（字符串常量、switch 跳转表）
.data	已初始化的全局/static 变量
.bss	未初始化的全局/static 变量（不占磁盘）
.symtab	符号表
.rel.text	代码中的重定位条目
.rel.data	数据中的重定位条目
.strtab	字符串表（符号名等）
.debug	调试信息（gcc -g）

速查表 5：IEEE 754 单精度

结构：1 位 s | 8 位 exp | 23 位 frac 规格化数 (exp 非全0非全1): v = (-1)^s × (1 + frac/2^23) × 2^(exp - 127) 非规格化数 (exp = 0): v = (-1)^s × (frac/2^23) × 2^(-126) 0: s=任意, exp=0, frac=0 ±∞: exp=255, frac=0 NaN: exp=255, frac≠0 范围: 最大规格化数 ≈ 3.4 × 10^38 最小正规格化数 ≈ 1.18 × 10^-38

速查表 6：2 的幂

n	2^n	n	2^n	n	2^n
0	1	8	256	16	65536
1	2	9	512	20	1048576
2	4	10	1024 = 1K	24	16777216
3	8	11	2048	30	≈1G
4	16	12	4096	31	≈2G
5	32	13	8192	32	≈4G
6	64	14	16384	63	≈9.2×10^18
7	128	15	32768	64	≈1.8×10^19

速查表 7：常用 ASCII 值

字符	十六进制	说明
'\0'	0x00	字符串结束
'\n'	0x0A	换行
' ' (空格)	0x20	空格
'0' - '9'	0x30 - 0x39	数字字符
'A' - 'Z'	0x41 - 0x5A	大写字母
'a' - 'z'	0x61 - 0x7A	小写字母

速查表 8：补码加减法溢出判定

无符号加法溢出（CF=1）： 最高位有进位 / 等价：结果反而比加数小 有符号加法溢出（OF=1）： 正 + 正 = 负 → 溢出 负 + 负 = 正 → 溢出 正 + 负 → 永远不溢出 减法 x - y（实际是 x + (-y)）： CF=1：表示无符号 x < y（借位） OF=1：有符号溢出（如 INT_MIN - 1）

速查表 9：call 指令机器码解读

call 指令：e8 + 4 字节小端有符号偏移 目标地址 = (当前 call 指令地址 + 5) + signed(偏移) 例：4011aa: e8 38 00 00 00 偏移 = 0x00000038 = 56 目标 = 0x4011aa + 5 + 0x38 = 0x4011e7 ret 指令：c3

速查表 10：缓冲区溢出攻击模板

步骤： 1. 找出缓冲区大小 N（从 sub $N, %rsp 看） 2. 字符串前 N 字节随意填充（通常用 00） 3. 接下来 8 字节 = 目标地址的小端表示 例：N=40，目标地址 0x4011fb 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 fb 11 40 00 00 00 00 00 防御方法（三类）： 1. 代码层：用 fgets / strncpy 代替 gets / strcpy 2. 系统层：栈随机化 ASLR、栈不可执行 NX 3. 编译器：栈金丝雀 Stack Canary

速查表 11：x86-64 数据类型大小

C 类型	大小（字节）	对齐	说明
char	1	1	
short	2	2	
int	4	4	
long	8	8	x86-64 上
long long	8	8	
float	4	4	
double	8	8	
void *	8	8	任何指针都是 8 字节

速查表 12：结构体大小算法

步骤： 1. 按声明顺序排列成员 2. 每个成员起始地址必须是其对齐要求的倍数（不够则插 padding） 3. 整体大小必须是最大

对齐要求的倍数（末尾补 padding） 例： struct { int a; // 偏移 0, 占 4 double b; // 偏移 8（4-7 是 padding，因为 double 要 8 对齐），占 8 char c; // 偏移 16, 占 1 // 末尾 17-23 是 padding，对齐到 8 }; 总大小 = 24 字节