**“ 计算机系统基础 ”模拟试卷**

1. **计算机工作基本原理填空**

1、在对某程序进行调试时，看到如下一段信息 ：

004116DA 83 7D FC 00 cmpl ………………

004116DE 75 08  jne 004116E8

004116E0 8B 45 FC mov ………………

004116E3 89 45 F8 mov ………………

004116E6 EB 07 jmp 004116EF

004116E8 C7 45 F8 01 00 00 00 movl ………………

004116EF E8 2D FB FF FF callq func (00411221) ; func为函数名

004116F4 C7 45 F4 21 12 41 00 movl $0x411221, -0x0c(%ebp)

004116FB FF 55 F4 callq -0x0c(%ebp)

………… ………… …………

① 观察指令在内存中的存放形式(每个空对应一个16进制字节数据) ：

004116DA \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

004116E2 ………………………………

② 设当前 eip 为 0x004116DE，在取出eip指向的指令并进行译码后，eip = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_；

③ 在0x004116DE 处的指令为： 75 08 jne 004116E8，直观上是在zf =\_\_\_\_\_\_时，会将0x004116E8 →eip , 计算出该地址值的方法 ；当转移条件不成立时，则该指令执行完成， eip （会、不会）改变。

④ 004116EF 处的指令中有0xFFFFFB2D，计算出该值的方法是 ；

⑤ 004116FB 处的子程序调用指令，调用的子程序的入口地址是 ，执行该处语句时，CPU会将 0x 压入堆栈中。

⑥ 执行子程序中的RET指令时，CPU会 →eip。

2、设一个C语言程序中定义有全局数组 int array[5]; array的起始地址（即&array[0]）为 0x420100，每个元素占4个字节，现要将数组的第2个元素(即array[2])置为20。按指定的寻址方式写出实现该功能的机器指令段（汇编语句段）。

注：完成该功能的C语句有：“array[2]=20; \*(array +2) =20; int i=2; array[i]=20; int \*p=array; p[i] = 20;”等等。不同的C语句写法编译后，生成的机器指令不同，会出现多种寻址方式访问同一单元。

① 使用直接寻址方式，机器指令中含有操作数的偏移地址 ：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

② 使用寄存器间接寻址方式，先将操作数的地址送ebx，再访存：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ； 给 ebx赋值

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

③ 使用变址寻址方式，先将元素的下标（即2）送eax，再访存：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_； 给 eax赋值

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

④ 使用基址加变址寻址方式，先将数组的起始地址送eax，第2个元素在数组中的偏移字节数送ebx，再访存：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ； 给 eax赋值

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ； 给 ebx赋值

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

⑤ 对于全局变量，其空间分配在\_\_\_\_\_\_\_\_\_\_\_\_\_。而对于非静态的局部变量，其空间分配在\_\_\_\_\_\_\_\_\_\_\_\_\_，对应的地址表达方式一般为 disp(%ebp) 、disp(%esp) 等。

1. **数据存储及C语句转换填空**

在Linux环境下，对一个C语言程序进行编译、链接、调试运行，程序片段如下。

int fadd(int a, int b)

{ int temp;

temp = a + b;

return temp;

}

void main( )

{

int x = 0x1234;

int y = -32;

int result = 0;

char msg[6] = "abc12"; // '1'的ASCII是 0x31, 'a'的ASCII是 0x61

result = fadd(x, y);

result = \*(int \*)(msg+1);

}

调试时，设变量 x 的地址（即&x）为 0xffffd508； y 的地址(即&y) 为 0xffffd50c，result的地址为0xffffd510，数组 msg 的起始地址为 0xffffb516。

1. 执行到“result = fadd(x,y)；”时，以字节为单位观察内存内容（用16进制数的形式填空，最左边是内存地址）。

0xffffd508 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_

0xffffd510 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ XX XX \_\_\_\_\_ \_\_\_\_\_

0xffffd518 \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ \_\_\_\_\_ XX XX XX XX

1. 数据传送指令解读

“int x = 0x1234; ”对应的机器指令为： movl $0x1234, -0x20(%ebp)，执行该语句时， ebp= 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

“int result = 0; ”对应的反汇编指令为 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

执行 “result = \*(int \*)(msg+1);”后， result中的值为 0x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 。

1. 函数调用语句解读

语句“result = fadd(x, y);”对应的反汇编代码（最左边的是机器指令的地址）如下。

|  |  |
| --- | --- |
|  | 0xffffd4ec |
|  | 0xffffd4f0 |
|  | 0xffffd4f4 |
|  | 0xffffd4f8 |
|  | 0xffffd4fc |
| XXXXXXXX | 0xffffd500 |

0x56556210 <+72>: push -0xlc(%ebp)

0x56556213 <+75>: push -0x20(%ebp)

0x56556216 <+78>: call 0x5655619d <fadd>

0x5655621b <+83>: add $0x8, %esp

0x565562le <+86>: mov %eax, -0x18(%ebp)

设执行“result = fadd(x, y);”之前，esp的值为0xffffd500。

① 在表格的适当位置填写刚进入函数fadd内部时，堆栈中存放的相关数据（6分）；

② 刚进入函数fadd内部时，esp = 0x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_；

③ 执行 “ add $0x8, %esp ” 之后，esp = 0x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_。

4. 函数fadd的指令解读

函数体对应的反汇编代码有：

0x5655619d <+0>: push %ebp

0x5655619e <+1>: mov %esp, %ebp

0x565561a0 <+3>: sub $0x10, %esp

0x565561a3 <+6>: mov 0x8(%ebp), %edx

0x565561a6 <+9>: mov 0xc(%ebp), %eax

0x565561a9 <+12>: add %edx, %eax

0x565561ab <+14>: mov %eax, -0x4(%ebp)

0x565561ae <+17>: mov -0x4(%ebp), %eax

0x565561b1 <+20>: ……………….

① 函数参数 a的地址（即&a）是 0x\_\_\_\_\_\_\_\_\_\_\_\_\_。

② 局部变量 temp的地址（即&temp）是 0x\_\_\_\_\_\_\_\_\_\_\_\_\_。

③ 执行 “add %edx, %eax” 后， CF=\_\_\_\_, SF=\_\_\_\_\_, ZF=\_\_\_\_\_, OF= \_\_\_\_\_\_\_。

④ 在函数的结束处，有程序段

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ret

执行后可返回到调用函数的断点处。

1. **程序阅读与理解**

**1、阅读下面的程序，回答问题 。**

void func ()

{ unsigned short us = 65535;

unsigned int ui;

short s = -1;

int i;

int x = 0, y = 0; ; ①

ui = us;

i = s; ; ②

if (ui > 0) x = 1;

if (i > 0) y = 1; ; ③

ui = ui >> 1; ; >>1 右移一个二进制位

i = i >> 1; ; ④

ui = ~ui; ; 按位取反

i = -i; ; ⑤

}

**⑴ 执行完** ① 处语句后，

us = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_ s = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_（2个字节的16进制数）

**⑵ 执行完** ② 处语句后，

ui = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ i = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_（4个字节的16进制数）

ui=us; 对应的机器指令： \_\_\_\_\_\_\_ us, %eax 、 mov %eax, ui

i=s; 对应的机器指令： \_\_\_\_\_\_\_ s, %eax 、 mov %eax, i

**⑶ 执行完 ③** 处语句后，

x = \_\_\_\_\_\_\_\_\_\_\_\_\_\_ y = \_\_\_\_\_\_\_\_\_\_\_\_\_\_

if (ui>0) x=1; 对应的机器指令： cmp $0, ui、 \_\_\_\_\_\_\_ lp1、 mov $1, x、 lp1:…

if (i >0) y=1; 对应的机器指令： cmp $0, i、 \_\_\_\_\_\_\_ lp2、 mov $1, y、 lp2:…

**⑷ 执行完 ④** 处语句后，

ui = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_ i = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_ （4个字节的16进制数）

ui = ui >> 1; 对应的机器指令： \_\_\_\_\_\_\_ $1, ui

i = i >> 1; 对应的机器指令： \_\_\_\_\_\_\_ $1, i

**⑸**  **执行完**  ⑤ 处语句后，

ui = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_ i = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_ （4个字节的16进制数）

ui = ~ui; 对应的机器指令： \_\_\_\_\_\_\_ ui

i = -i; 对应的机器指令： \_\_\_\_\_\_\_ i

**2、调试一个C程序时，在函数的反汇编中看到源程序语句与对应的反汇编语句如下。根据观察到的信息填空。**

int a[2][5];

int i = 1;

movl $1, -0x2c(%ebp)

i 的地址为 0x0019feb8， ebp = 0x \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

int j = 3;

movl $3, -0x30(%ebp) &j = 0x\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

a[i][j] = 10;

imul $0x14, -0x2c(%ebp), %eax

执行后，eax = \_\_\_\_\_\_\_\_\_\_\_\_\_\_

该值的含义是\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lea -0x28(%ebp, %eax, 1), %ecx

ecx中存放的值的含义是\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

mov -0x30(%ebp), %edx

movl $0xa, (%ecx, %edx, 4)

指令中的 edx 乘4的原因是\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

a[0] 的地址表达形式是 \_\_\_\_\_\_\_\_\_\_\_\_;

global[i] = 18; global 为一个全局整型数组

mov -0x2c(%ebp), %eax

movl $0x12, 0x417120(, %eax, 4)

global 数组的起始地址是\_\_\_\_\_\_\_\_\_¬¬¬\_\_\_,

int\* p = &global[1];

mov $4, %eax

shl $0, %eax

add $0x417120, %eax

mov %eax, -0x34(%ebp)

变量p 中的值是 0x\_\_\_\_\_\_\_\_\_¬¬¬\_\_\_\_\_\_

\*p = 19;

mov -0x34(%ebp), %eax

movl $0x13, (%eax)

第二条汇编语句访存的寻址方式是\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3、 以下是结构test的声明。假设在32位Windows平台上编译（采用缺省的自然对齐方式），问结构成员d和v的偏移量是多少？ 结构体所占存储空间是多少字节？ 如何调整成员的先后顺序使得结构所占存储空间最小？

struct{

char c;

int i;

double d;

short s;

double \*v;

long l;

} test;

**4、阅读下面的程序，回答问题 。**

.section .data

array: .long 10, -20, 30, -40, 50

length = (. -array)/4 # length 为array中元数的个数，= 5

format: .ascii "%d\n\0"

.section .text

.global \_start

\_start:

mov $0, %eax

mov $length, %ecx

lea array, %edi # ①

lp\_1:

cmpl $0, (%edi)

jl lp\_2 # ②

inc %eax

lp\_2:

add $4, %edi

sub $1, %ecx # ③

jne lp\_1

push %eax

push $format

call printf

mov $1, %eax # 程序正常退出

mov $0, %ebx

int $0x80

1. 上述程序的功能是什么？运行后，屏幕上显示的是什么？
2. 若标号 lp\_1 写到 **①**处语句前，程序运行的结果是什么？为什么？
3. 若将 **②** 处的语句改为 “jb lp\_2”,程序运行的结果是什么？
4. 若漏写了 ③ 处的语句 ，程序运行会出现什么现象？为什么？

5、已知**function\_test**的C语言代码框架如下，根据对应的汇编代码填写C代码中缺失部分。

**int** **function\_test( unsigned x) // unsigned int**

**{**

**int result=0;**

**int i；**

**for (** **① ;** **② ;** **③ ) {**

**④ ；**

**⑤ ；**

**}**

**return result;**

**}**

上述函数过程体对应的汇编代码如下：

1 movl 8(%ebp), %ebx

2 movl $0, %eax

3 movl $0, %ecx

4 .L12:

5 leal (%eax,%eax), %edx

6 movl %ebx, %eax

7 andl $3, %eax

8 orl %edx, %eax

9 shrl %ebx // shr $1, %ebx

10 addl $1, %ecx

11 cmpl $64, %ecx

12 jne .L12

……

ret

**参见教材 P138 -P139 例 3-13的分析**

1. **程序优化**
2. 举例说明编写C程序或者编译器优化时，利用CPU特性的做法（包括优化前的方法，优化后的方法）。

① 提高CPU中cache的命中率

② 提高CPU中指令流水线的利用率

③ 使用CPU中处理速度更快的指令

④ 使用CPU中单指令多数据流指令或串操作指令

⑤ 使用多核CPU中多线程处理能力

2、 在编写程序（也包括编译器）可以做哪些 与CPU无关的优化工作？（给出5种不同类型的示例）。

以下优化不依赖特定CPU架构，适用于通用编译器和程序开发：

**(1) 常量传播（Constant Propagation）**

原理 ：在编译时计算表达式中已知常量的值，直接替换结果。

示例 ：

int a = 10;

int b = a \* 2; // 优化为 int b = 20;

**(2) 死代码消除（Dead Code Elimination）**

原理 ：移除永远不会执行的代码（如不可达分支或未使用的变量）。

示例 ：

if (false) {

printf("This will never run"); // 直接删除

}

**(3) 循环不变量外提（Loop Invariant Code Motion）**

原理 ：将循环内不变的计算移到循环外，减少重复计算。

示例 ：

for (int i = 0; i < n; i++) {

int x = y \* z; // y和z在循环内不变，外提到循环外

arr[i] = x;

}

**(4) 函数内联（Function Inlining）**

原理 ：将小函数调用替换为函数体本身，减少调用开销。

示例 ：

inline int square(int x) { return x \* x; }

int a = square(5); // 优化为 int a = 25;

**(5) 公共子表达式消除（Common Subexpression Elimination, CSE）**

原理 ：识别并重用重复计算的表达式。

示例 ：

int a = x \* y + z;

int b = x \* y + w; // x\*y 只计算一次

3、为了提高程序运行的安全性，编译器有多种编译开关（选项），增强发现程序漏洞的能力。试举出二种编译开关的例子，并说明增强安全性的实现原理。【堆栈帧检查、未初始化变量使用检查】

**(1) 堆栈帧检查（Stack Frame Protection）**

编译选项 ：GCC/Clang: -fstack-protector 或 -fstack-protector-all

原理 ：

在函数栈帧中插入 金丝雀值（Canary） ，在函数返回前检查该值是否被修改。

若检测到溢出（如缓冲区溢出覆盖了返回地址），立即终止程序。

效果 ：防止栈溢出攻击（如ROP攻击）。

**(2) 未初始化变量检查（Uninitialized Variable Check）**

编译选项 ：

GCC/Clang: -Wuninitialized （警告）或 -O2 （自动优化时报错）

MSVC: /RTC1

原理 ：

静态分析或运行时检查变量是否未初始化即被使用。

对局部变量（尤其是栈上的变量）进行标记，访问时验证其初始化状态。

示例 ：

int x;

printf("%d", x); // 编译时报错或运行时终止

(3) 其他常见安全选项

**地址随机化（ASLR） ：**

原理：随机化代码和数据的内存地址，增加攻击者预测难度。

**格式化字符串检查 ：**

原理：禁止直接使用用户输入作为 printf 的格式化字符串。

1. **链接和异常控制流问答**
2. **设一个函数中有语句 int temp=global; 其中global是一个初值为35的int 类型全局变量。编译器对global的定义(int global=35;)和temp=global编译时，分别会在可重定位目标文件中的哪些节生成哪些信息？**

对于 int global=35; 在 数据节(.data) 节存放 35（0x00000023）; 在 字符串节(.strtab)，存放字符串 “global”；在符号表节（.symtab），存放有关符号 global的信息，包括定义该符号的节号、在相应节（.data）节的地址、数据长度（4个字节）、属性信息等。

对于 int temp=global; 在代码节(.text)中， 生成相应的语句，其中 global 的地址（偏移量）用占位符 0x00000000。在代码节的重定位节(.rel.text)，要记录 代码节的相应位置（global占位符的起始地址）要被什么符号（符号表的哪一项）、用什么定位方式（地址相对程序计算器PC的32位偏移）所代替，还包括计算地址时的附加项等。

1. **可重定位目标文件中有哪些节？各节中主要有什么信息？什么是符号解析？链接的过程是什么？**

**1. 可重定位目标文件中的节（Sections）及各节信息**

可重定位目标文件（如 .o 文件）通常包含以下核心节（以 ELF 格式为例）：

|  |  |  |
| --- | --- | --- |
| **节名** | **作用** | **存储内容** |
| .text | 代码节 | 程序的机器指令（函数实现、逻辑代码）。 |
| .data | 已初始化数据节 | 已初始化的全局变量和静态变量（如 int global = 42;）。 |
| .bss | 未初始化数据节 | 未初始化的全局变量和静态变量（如 int buffer[100];），仅记录大小，不占空间。 |
| .rodata | 只读数据节 | 常量数据（如字符串字面量 "Hello"、const 变量）。 |
| .symtab | 符号表 | 所有符号（函数、全局变量）的名称、类型、大小、所在节等信息。 |
| .strtab | 字符串表 | 符号名称字符串（如 "main"、"global"）。 |
| .rel.text | 代码重定位表 | .text 节中需要重定位的指令（如函数调用、全局变量引用）。 |
| .rel.data | 数据重定位表 | .data 节中需要重定位的数据（如全局指针 int\* p = &global;）。 |
| .debug | 调试信息（可选） | 调试符号（如行号、变量类型），需 -g 选项生成。 |
| .line | 行号信息（可选） | 源代码行号与机器指令的映射。 |
| .comment | 注释信息（可选） | 编译器版本信息（如 GCC: (Ubuntu 11.3.0)）。 |

**2. 符号解析（Symbol Resolution）**

定义

符号解析是链接器的核心任务之一，目的是将每个符号（如函数名、变量名）的引用与其定义关联起来。

**符号类型：**

全局符号：定义在当前文件（如 int global;）或外部文件（extern int global;）。

局部符号：静态函数/变量（如 static int x;），仅当前文件可见。

外部符号：未定义的引用（如 printf）。

**解析规则**

强符号 vs 弱符号：

强符号：函数名或已初始化的全局变量（如 int x = 5;）。

弱符号：未初始化的全局变量（如 int x;）。

规则：强符号优先，重复强符号会报错；弱符号可被覆盖。

**解析过程：**

链接器遍历所有目标文件的 .symtab，为每个符号找到唯一的定义。

若找不到定义，报 undefined reference 错误（如未链接 -lm 时调用 sqrt）。

**3. 链接的过程（Linking）**

链接器将多个可重定位目标文件（.o）合并为可执行文件（a.out），主要步骤如下：

（1）符号解析（Symbol Resolution）

将所有目标文件的符号表（.symtab）合并，解决符号引用。

示例：

main.o 调用 foo()，链接器在 lib.o 中找到 foo 的定义。

（2）重定位（Relocation）

地址分配：为每个节（.text、.data 等）分配运行时内存地址。

修正引用：根据重定位表（.rel.text、.rel.data）修改代码和数据中的占位符。

绝对地址修正：如全局变量 global 的地址。

相对地址修正：如 call foo 指令的偏移量。

（3）合并节（Section Merging）

将所有输入文件的 .text 节合并到输出文件的 .text 节，其他节同理。

示例：

main.o 和 lib.o 的 .text 合并为可执行文件的 .text。

（4）生成可执行文件

写入ELF头、程序头（描述内存布局）、节头表等元信息。

最终文件：包含可直接加载到内存执行的代码和数据。

1. **什么是中断和异常？两者有何差别？什么是中断描述符表？中断和异常的响应过程是什么？**

中断和异常：一个进程在执行过程中，正常的逻辑控制流被特殊的事件所打断，CPU转到处理这些事件的内核程序去执行，从而引发一个异常控制流。

中断一般是指CPU之外的事件发生，如按键盘，鼠标等，是与当前正在执行的指令无关的异步事件。异常是CPU正在执行的指令引发的事件，如除0、访问数据的地址超出程序空间范围等等，是与正在执行的指令相关的同步事件。异常分为故障、陷阱和终止。

中断描述表（中断矢量表）：中断服务程序和异常处理程序的入口地址构成的一个表，每个表项占 8个字节，存放入口偏移地址及相应代码段的描述信息；这些表项是按照中断和异常的编号（即中断和异常的类型号）顺序排列的。

中断和异常的响应过程：CPU的控制逻辑确定检测到的中断和异常类型号，从中断描述表取出对应的表项，计算相应的处理程序的入口地址，保存当前的CS、EIP、EFLAGS等信息，然后转到处理程序的入口地址对应的位置去执行。在中断处理程序中有IRET指令，执行该指令从堆栈中恢复CS、EIP、EFLAGS等，从而回到被中断的位置继续执行。对于三类异常：故障、陷阱和终止的处理策略有所差异。故障表示能够修复，引起故障的指令会被再次执行；陷阱则是执行引发陷阱的下一条指令；而终止就要结束程序的运行。

1. 在一个程序的运行过程中（如正在执行一个二维数组求累加和），用户按了键盘上的某个键，计算机系统会做出哪些响应（即一系列的处理过程）？中断分哪几类？请举例说明各类中断在何种情况下产生。

# **六、 已知以下关于Lab3 Bang阶段的信息，请完成填空，注意涉及数值全部使用16进制。**

08048e6d <test>:

8048e6d: 55 push %ebp

8048e6e: 89 e5 mov %esp,%ebp

8048e70: 53 push %ebx

8048e71: 83 ec 24 sub $0x24,%esp

8048e74: e8 6e ff ff ff call 8048de7 <uniqueval>

8048e79: 89 45 f4 mov %eax,-0xc(%ebp)

8048e7c: e8 6b 03 00 00 call 80491ec <getbuf>

8048e81: 89 c3 mov %eax,%ebx

8048e83: e8 5f ff ff ff call 8048de7 <uniqueval>

......

080491ec <getbuf>:

80491ec: 55 push %ebp

80491ed: 89 e5 mov %esp,%ebp

80491ef: 83 ec 38 sub $0x38,%esp

80491f2: 8d 45 d8 lea -0x28(%ebp),%eax

80491f5: 89 04 24 mov %eax,(%esp)

80491f8: e8 55 fb ff ff call 8048d52 <Gets>

80491fd: b8 01 00 00 00 mov $0x1,%eax

8049202: c9 leave

8049203: c3 ret

08048d05 <bang>:

8048d05: 55 push %ebp

8048d06: 89 e5 mov %esp,%ebp

8048d08: 83 ec 18 sub $0x18,%esp

8048d0b: a1 18 c2 04 08 mov 0x804c218,%eax

8048d10: 3b 05 20 c2 04 08 cmp 0x804c220,%eax

8048d16: 75 1e jne 8048d36 <bang+0x31>

8048d18: 89 44 24 04 mov %eax,0x4(%esp)

8048d1c: c7 04 24 e4 a2 04 08 movl $0x804a2e4,(%esp)

8048d23: e8 a8 fb ff ff call 80488d0 <printf@plt>

……

Breakpoint 2, 0x080491f2 in getbuf ()

(gdb) info r

eax 0x6f50c1c5 1867563461

ecx 0xf7fbd068 -134492056

edx 0xf7fbd3cc -134491188

ebx 0x0 0

esp 0x55683458 0x55683458 <\_reserved+1037400>

ebp 0x55683490 0x55683490 <\_reserved+1037456>

esi 0x55686018 1432903704

edi 0x1 1

eip 0x80491f2 0x80491f2 <getbuf+6>

……

(gdb) x 0x804c218

0x804c218 <global\_value>: 0x00000000

acd@ubuntu:~/Lab3$ ./makecookie U201414XXX

0x250d3ee8

int global\_value = 0;

void bang(int val)

{

if (global\_value == cookie) {

printf("Aha Bang!: You set global\_value to 0x%x.\n", global\_value);

validate(2);

} else

printf("Oh Misfire: global\_value = 0x%x\n", global\_value);

exit(0);

}

1)调用getbuf后的返回地址是: ⑴ **0x08048d05**

2)调用getbuf时，存放getbuf返回地址的单元是: ⑵

3)调用getbuf时，执行到断点2时，getbuf的栈帧栈顶地址是： ⑶

4)getbuf中缓冲区buf的起始地址是： ⑷

5)攻击字符串的长度为: ⑸ 字节

6)Cookie值 0x250d3ee8所在的存储单元地址是： ⑹

7)以下是所设计的一个带有注释的攻击字符串文件的内容，请填空：

c3 c3 c3 c3

b8 ⑺ /\* mov ⑻ ,%eax \*/

a3 18 c2 04 08 /\* mov %eax, ⑼ \*/

68 05 8d 04 08 /\* push ⑽ \*/

c3 /\* ret \*/

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

14)（11） （12） （13） 55

15)上述攻击字符串可使bang输出： （14）

**七、假设一个c语言程序有两个模块:** **main.c和swap.c, 对它们单独编译，分别生成可重定位目标文件main.o和swap.o。 main.c和swap.c代码如下：**

int buf[2] = {1, -4};

extern void swap( );

int sum=0;

int main( )

{

swap( );

return 0;

}

int ave( ) {

int a;

static int count=2;

sum=buf[0]+buf[1];

a=sum/count;

return a;

}

main.c

extern int buf[];

extern int ave();

int \*bufp0 = &buf[0];

static int \*bufp1;

int sum;

void swap( ) {

int temp;

bufp1 = &buf[1];

temp = \*bufp0;

\*bufp0 = \*bufp1;

\*bufp1 = temp;

ave( )；

}

swap.c

请指出main.o和swap.o中所有强符号和弱符号分别有哪些？

请指出swap.o的符号表中分别有哪些符号？

请指出swap.o的符号表中各符号分别是什么类型的符号（全局符号, 本地符号或外部符号）？请指出swap.o的符号表中各符号分别出现在swap.o中的哪个节（.text, .data或.bss）?

**（2）假定swap.o的反汇编代码如下：**

00000000 <swap>:

0: 55 push %ebp

1: 89 e5 mov %esp,%ebp

3: 83 ec 18 sub $0x18,%esp

6: c7 05 00 00 00 00 04 movl $0x4,0x0 # 重定位位置①

d: 00 00 00

8: R\_386\_32 .bss

c: R\_386\_32 buf

10: a1 00 00 00 00 mov 0x0,%eax # 重定位位置②

11: R\_386\_32 bufp0

15: 8b 00 mov (%eax),%eax

17: 89 45 f4 mov %eax,-0xc(%ebp)

1a: a1 00 00 00 00 mov 0x0,%eax # 重定位位置③

1b: R\_386\_32 bufp0

1f: 8b 15 00 00 00 00 mov 0x0,%edx

21: R\_386\_32 .bss # 重定位位置④

25: 8b 12 mov (%edx),%edx

27: 89 10 mov %edx,(%eax)

29: a1 00 00 00 00 mov 0x0,%eax # 重定位位置⑤

2a: R\_386\_32 .bss

2e: 8b 55 f4 mov -0xc(%ebp),%edx

31: 89 10 mov %edx,(%eax)

33: e8 fc ff ff ff call 34 <swap+0x34> # 重定位位置⑥

34: ⑥

38: 90 nop

39: c9 leave

3a: c3 ret

**假定可执行目标文件里swap函数代码的起始地址是0x00808100，swap函数代码紧接在ave函数代码的后面，且ave函数代码占0x2e个字节。对swap.o中的重定位位置⑥进行重定位。回答：此处待重定位的符号是什么？ 重定位类型是什么？ 重定位前的值是什么？ 该值的含义是什么? 重定位后的值是什么（需要给出计算过程）？ 重定位后最终指向的虚拟地址是多少?**

|  |  |
| --- | --- |
| **问题** | **答案** |
| 待重定位的符号 | ave |
| 重定位类型 | R\_386\_PC32（PC 相对偏移） |
| 重定位前的值 | 0xfffffffc（即 -4） |
| 该值的含义 | 未重定位时无效的占位符偏移量（跳转到 call 自身）。 |
| 重定位后的值 | 0xffffffc6（计算：ave地址 - PC = -0x3a） |
| 最终指向的虚拟地址 | 0x008080d1（ave 的入口地址） |