

Homework7

这是 [Homework7.pdf](#) 的标准答案。

注：“过程与解析”是 Gemini 给出的解析，并不是答案本身的一部分，仅供参考。

Answer1

Variable	Start Address
d[0]	0x8049600
d[1]	0x8049614
d[0].a	0x8049600
d[0].b[1]	0x8049604
d[0].c	0x8049608
d[0].p.y	0x804960C
d[0].p.z	0x804960C
d[0].d	0x8049610

过程与解析

在 32 位机器上，数据类型大小和对齐要求如下：

- `char`: 1 字节大小，1 字节对齐
- `short`: 2 字节大小，2 字节对齐
- `int`: 4 字节大小，4 字节对齐
- `char*` (指针): 4 字节大小，4 字节对齐
- `union p`: 大小为其最大成员的大小，即 `int z` 的 4 字节。对齐要求为其成员的最严格对齐要求，即 `int z` 的 4 字节。
- `struct data`: 结构体大小必须是其最严格对齐要求 (`char*` 和 `union p` 的 4 字节) 的倍数。

计算 `d[0]` (基地址: `0x8049600`):

1. `d[0].a` (`char`, 1B):
 - 地址: `0x8049600`
 - 偏移: 0
 - 占用: `[0x8049600]`
2. `d[0].b` (`short[2]`, 4B):
 - `short` 需 2 字节对齐。下一个可用地址是 `0x8049601`，不是 2 的倍数。
 - 填充 1 字节 (at `0x8049601`)。
 - `d[0].b[0]` 地址: `0x8049602` (偏移 2)
 - `d[0].b[1]` 地址: `0x8049604` (偏移 4)
 - 占用: `[0x8049602 - 0x8049605]`
3. `d[0].c` (`char*`, 4B):

- `char*` 需 4 字节对齐。下一个可用地址是 0x8049606，不是 4 的倍数。
- 填充 2 字节 (at 0x8049606, 0x8049607)。
- 地址: 0x8049608 (偏移 8)
- 占用: [0x8049608 - 0x804960B]

4. `d[0].p` (union, 4B):

- `union p` 需 4 字节对齐。下一个可用地址是 0x804960C，是 4 的倍数。
- 地址: 0x804960C (偏移 12)
- `d[0].p.x, d[0].p.y, d[0].p.z` 均始于此地址。
- 占用: [0x804960C - 0x804960F]

5. `d[0].d` (char, 1B):

- `char` 需 1 字节对齐。下一个可用地址是 0x8049610。
- 地址: 0x8049610 (偏移 16)
- 占用: [0x8049610]

计算 `struct data` 总大小:

- 当前已占用 17 字节 (偏移 0 到 16)。
- 结构体 `struct data` 的最严对齐要求是 4 字节。
- 总大小必须是 4 的倍数。17 向上取 4 的倍数是 20。
- 在末尾填充 3 字节 (at 0x8049611, 0x8049612, 0x8049613)。
- `sizeof(struct data)` = 20 字节 (0x14)。

计算 `d[1]` 地址:

- `d[1]` 的地址 = `d[0]` 的地址 + `sizeof(struct data)`
- `0x8049600 + 0x14 = 0x8049614`。

Answer2

```
do you want a midterm exam?
yes!
```

过程与解析

1. `printf("%s?\n", char_table);`

- `char_table` 是一个 `char[3][13]` 类型的数组。当作为 `%s` 参数传递给 `printf` 时，它会衰变为指向其第一个元素 (即 `char_table[0]`) 的指针。
- `printf` 会从 `char_table[0]` 的起始地址开始打印字符，直到遇到 `\0`。
- C 语言中，多维数组在内存中是连续存储的。
- `char_table[0]` 包含: "do you want a" (ASCII 码)
- `char_table[1]` 紧随其后，包含: " midterm exam" (ASCII 码)
- `char_table[2]` 紧随其后，包含: `0` (即 `\0`, null 终止符)。
- 因此，`printf` 会连续读取 `char_table[0]` 和 `char_table[1]` 的内容，直到在 `char_table[2][0]` 处遇到 `\0` 才停止。
- 输出: `do you want a midterm exam?` 后面跟一个换行符。

2. `printf("%c%c%c!\n", ...);`

- 第一个参数: `(char)((char **)ans)[6]`

- `ans` 是一个 `char[]` 数组。
- `(char **)ans` 是一个强制类型转换, 将 `ans` 数组的起始地址 (类型为 `char*`) 强行转换为 `char**` (指向 `char` 指针的指针)。
- `((char **)ans)[6]` 是指针运算, 等价于 `*((char **)ans + 6)`。
- 在 32 位机器上, `sizeof(char*) = 4` 字节。
- `(char **)ans + 6` 的地址等于 `ans` 的基地址 $+ 6 * \text{sizeof}(\text{char}^*) = \text{ans} + 24$ 。
- 所以 `((char **)ans)[6]` 表达式的含义是: 从 `ans[24]` (`ans` 数组的第 25 个字节) 开始, 读取 4 个字节, 并将这 4 个字节 (在小端序机器上) 解释为一个 `char*` 指针。
- `ans` 字符串中:
 - `ans[24] = 'y'` (ASCII 0x79)
 - `ans[25] = 'y'` (ASCII 0x79)
 - `ans[26] = 'z'` (ASCII 0x7A)
 - `ans[27] = '\0'` (ASCII 0x00)
- 这 4 个字节 `0x79, 0x79, 0x7A, 0x00` 在小端序下组成的 32 位指针地址是 `0x007A7979`。
- 最外层的 `(char)(...)` 将这个指针 值 `0x007A7979` 强制转换为 `char`。
- 转换时, 只取其最低字节, 即 `0x79`。
- `0x79` 对应的 ASCII 字符是 '`y`'。

- 第二个参数: `(char)((char *)ans)[4]`

- `(char *)ans` 只是 `ans` 数组衰变后的标准 `char*` 指针。
- `((char *)ans)[4]` 等价于 `ans[4]`。
- `ans[4]` 是 '`e`'。

- 第三个参数: `(char)(ans[18])`

- `ans[18]` 是 '`s`'。

- `printf` 组合: 打印 `y e s`, 然后是 ! 和换行符。

- 输出: `yes!`

Answer3

	Offset 1	Offset 2	Offset 3	Offset 4	Total size	Alignment
A	i:0	c:4	j:8	d:16	24	8
B	i:0	c:8	d:9	j:12	16	8
C	w:0	c:8			32	8
D	a:0	p:48			56	8
E	w:0	c:6			10	2

过程与解析

x86-64 (64位) 架构下的对齐规则:

- `char`: 1 字节 (对齐 1)
- `short`: 2 字节 (对齐 2)

- `int`: 4 字节 (对齐 4)
- `long`: 8 字节 (对齐 8)
- `pointer (*)`: 8 字节 (对齐 8)
- 结构体：
 - **Alignment**: 等于其成员中最大的对齐要求。
 - **Total size**: 必须是其 **Alignment** 的整数倍。

A. struct P1 { int i; char c; long j; char d;};

- `i` (`int`): offset 0 (size 4)
- `c` (`char`): offset 4 (size 1)
- (next = 5) `j` (`long`, align 8): 需 3B 填充。offset 8 (size 8)
- (next = 16) `d` (`char`): offset 16 (size 1)
- (end = 17) **Alignment**: 8 (from `long j`)
- **Total size**: 17 向上取 8 的倍数 = 24

B. struct P2 { long i; char c; char d; int j;};

- `i` (`long`): offset 0 (size 8)
- `c` (`char`): offset 8 (size 1)
- `d` (`char`): offset 9 (size 1)
- (next = 10) `j` (`int`, align 4): 需 2B 填充。offset 12 (size 4)
- (end = 16) **Alignment**: 8 (from `long i`)
- **Total size**: 16 是 8 的倍数 = 16

C. struct P3 { short w[3]; char *c[3];}

- `w` (`short[3]`): offset 0 (size 6, align 2)
- (next = 6) `c` (`char*[3]`, align 8): 需 2B 填充。offset 8 (size 3*8=24)
- (end = 32) **Alignment**: 8 (from `char*`)
- **Total size**: 32 是 8 的倍数 = 32

D. struct P4 { struct P1 a[2]; struct P2 *p};

- (P1: size 24, align 8; P2: size 16, align 8)
- `a` (`P1[2]`): offset 0 (size 2*24=48, align 8)
- (next = 48) `p` (`P2*`, align 8): offset 48 (size 8)
- (end = 56) **Alignment**: 8 (from `P1` and `p`)
- **Total size**: 56 是 8 的倍数 = 56

E. struct P5 { short w[3]; char c[3];}

- `w` (`short[3]`): offset 0 (size 6, align 2)
- (next = 6) `c` (`char[3]`): offset 6 (size 3, align 1)
- (end = 9) **Alignment**: 2 (from `short w`)
- **Total size**: 9 向上取 2 的倍数 = 10

Answer4

输入内容 (针对 `username` 的输入):

- **位置 0-19 (共 20 字节):** 任意填充字节 (例如 20 个 'A')。
- **位置 20-23 (共 4 字节):** 目标地址 **0x804013da** (小端序)。
 - 位置 20: **0xda**
 - 位置 21: **0x13**
 - 位置 22: **0x40**
 - 位置 23: **0x08**

过程与解析

这是一个经典的栈缓冲区溢出攻击。`gets()` 函数不检查输入边界，允许我们写入超过 `username` 缓冲区（在 C 代码中声明为 8 字节）的数据，从而覆盖栈上的关键数据，特别是函数的返回地址。

1. 栈帧分析

根据 `login` 函数的汇编代码，我们可以分析其栈帧（Stack Frame）布局。栈在 x86 架构上向低地址增长。

- `pushl %ebp`: 将调用者 (caller) 的基址指针 (Old EBP/SFP) 压栈。
- `movl %esp, %ebp`: 设置 `login` 函数自己的基址指针 (EBP)。
- `subl $40, %esp`: 在栈上分配 40 字节的本地空间。
- `leal -16(%ebp), %eax`: `username` 缓冲区的起始地址被分配在 `EBP - 16`。
- `leal -24(%ebp), %eax`: `password` 缓冲区的起始地址被分配在 `EBP - 24`。

2. 栈布局可视化

我们可以绘制出 `login` 函数栈帧的关键部分（从高地址到低地址）：

内存地址	栈内容	说明
<code>EBP + 8</code>	...	传递给 <code>login</code> 的参数（如果有）
<code>EBP + 4</code>	返回地址 (RA)	<code>login</code> 函数执行 <code>ret</code> 时将跳转的地址
<code>EBP + 0</code>	旧 EBP (SFP)	调用者函数的基址指针
<code>EBP - 8</code>	(8 字节 padding)	本地变量空间的一部分
<code>EBP - 16</code>	<code>username</code> 缓冲区	<code>gets(username)</code> 写入的起始位置
<code>EBP - 24</code>	<code>password</code> 缓冲区	<code>gets(password)</code> 写入的起始位置
...	...	更多本地变量空间 (直到 <code>EBP - 40</code>)

3. 计算偏移量

攻击的目标是覆盖存储在 `EBP + 4` 的 **返回地址 (RA)**。

- 我们的输入 (`payload`) 通过 `gets(username)` 开始写入的地址是 `EBP - 16`。
- 我们需要计算从 `EBP - 16` 到 `EBP + 4` 总共需要多少字节。
- **偏移量 = (目标地址) - (缓冲区起始地址) = (EBP + 4) - (EBP - 16) = 20 字节。**

4. 构造 Payload

- **字节 0-19 (共 20 字节):** 这 20 个字节的“填充数据”（例如 20 个 'A'）将依次填满：
 1. `EBP - 16` 到 `EBP - 9` (8 字节, `username` 区域)

2. EBP - 8 到 EBP - 1 (8 字节, padding 区域)
 3. EBP + 0 到 EBP + 3 (4 字节, 旧 EBP)
- **字节 20-23 (共 4 字节):** 这 4 个字节将精确地覆盖在 EBP + 4 处的返回地址。我们将其设置为 login_ok 的地址 0x804013da。由于 x86 是小端序 (little-endian), 地址在内存中必须反转存储为 DA 13 40 08。

当 login 函数执行 ret 指令时, 它会从 EBP + 4 弹出地址 0x804013da 并跳转执行 login_ok 函数, 攻击成功。