bomblab 报告

姓名：汪逸凡

学号：2024201633

总分	phase_1	phase_2	phase_3	phase_4	phase_5	phase_6	secret_phase
7	1	1	1	1	1	1	1

Scoreboard 截图：

2024201633	0	1	1	0	0	1	1	0	0	0	0	0	0	4	7

解题报告

本次 Bomblab 实验旨在通过逆向工程手段，拆解一个二进制炸弹程序。实验过程中，我主要使用了 `objdump` 进行反汇编，以及 `GDB` 调试器进行动态调试。通过分析汇编代码中的控制流、栈帧结构和寄存器数据流，推导出了每一关的拆解字符串。

以下是对每一关的详细分析与解题过程。

phase_1: 字符串比较与内存寻址

```
// 答案
Step through the gate into Utopia, sink into a world of Melodia.
```

解题思路与深度分析：

本关主要考察对函数调用规范及内存寻址的理解。

1. 静态分析：

   首先使用 objdump -d bomb > bomb.asm 获取反汇编代码。观察 phase_1 函数，发现其核心逻辑非常简洁：

   ```
   1439: lea    0x1d40(%rip), %rsi    ; 加载立即数地址到 %rsi
   1440: call   1c86 <strings_not_equal> ; 调用比较函数
   1445: test   %eax, %eax            ; 检查返回值
   1447: jne    144e                  ; 若不相等则跳转至爆炸
   ```

   根据 System V AMD64 ABI 调用约定，函数调用的第一个参数存放于 `%rdi`，第二个参数存放于 `%rsi`。`phase_1` 接收用户的输入作为第一个参数，而指令 `lea 0x1d40(%rip), %rsi` 则将程序内部的一个静态地址加载到了 `%rsi` 中，作为 `strings_not_equal` 的第二个参数（即预期字符串）。

2. 动态调试：

   为了获取该地址指向的具体内容，我使用 GDB 进行动态调试。

   - `break phase_1`：在函数入口设置断点。
   - `run`：运行程序。
   - `stepi`：单步执行至 `call` 指令前。
   - `x/s $rsi`：以字符串格式查看 `%rsi` 寄存器指向的内存地址。

GDB 输出结果为 `"Step through the gate into Utopia, sink into a world of Melodia."`。这正是程序预期的输入字符串。

phase_2: 循环展开与静态数组访问

```
// 答案
839419 744513 720747 602320
```

解题思路与深度分析:

本关考察了对循环结构的识别以及数组指针的操作。

1. 输入分析:

   函数开头调用了 read_six_numbers。尽管函数名暗示读取 6 个数，但查看该函数内部实现，发现格式化字符串仅包含 4 个 %d，且栈上分配的空间也仅用于存储 4 个整数。因此确定本关需要输入 4 个整数。

2. 核心逻辑分析:

   汇编代码中出现了一个典型的循环结构，使用 %rax 作为循环变量（索引 i），从 0 递增至 3。循环体内部频繁访问了两个全局变量（通过 RIP 相对寻址），不妨称之为 matA 和 matB。

   ```
   14d8: mov    (%rdi,%rax,4), %edx  ; 从 matA 读取数据: val_A = matA[i]
   14db: imul   (%rsi,%rax,8), %edx  ; 从 matB 读取数据并相乘: val = val_A * matB[i]
   14df: add    %edx, %ecx           ; 累加结果: sum += val
   ```

   这段代码的逻辑是对两个数组进行加权求和运算。

3. 逆向策略:

   由于手动提取 matA 和 matB 的数据并进行计算较为繁琐且容易出错，我采用了动态调试策略来获取“标准答案”。

   循环结束后，程序有一条比较指令:

   ```
   151c: cmp    %eax, (%rsp,%rbx,1)  ; 比较计算出的 sum (%eax) 与用户输入的数值
   ```

   这里的 `%eax` 寄存器中存放的即为程序计算出的期望值。我在地址 `0x151c` 处设置断点，随意输入 `1 2 3 4` 运行程序。当程序在断点处暂停时，通过 `info registers eax` 查看 `%eax` 的值。由于外层还有一个循环依次检查这 4 个输入，我通过多次 `continue` 操作，依次获取了 4 个期望的数值，即为本关答案。

phase_3: Switch 跳转表 (Jump Table)

```
// 答案
5 -458
```

解题思路与深度分析:

本关展示了编译器如何优化多分支 `switch` 语句，即使用跳转表（Jump Table）。

1. 识别跳转表:

   在读取两个整数输入后，代码中出现了间接跳转指令:

```
1592: movslq (%rdx,%rax,4), %rax  ; 从表中读取偏移量
1596: add    %rdx, %rax           ; 计算目标绝对地址
1599: jmp    *%rax                ; 间接跳转
```

这种 `jmp *%rax` 结构是 `switch` 语句的典型特征。在此之前的边界检查 `cmpl $0x7, (%rsp)` 说明第一个输入（switch 的 case 值）必须是 **0 到 7** 之间的整数。

2. 路径追踪：

由于 switch 的不同分支最终都会汇聚到同一个检查点，理论上任意合法的 case 值都能推导出对应的答案。我选择了数字 5 作为第一个输入。

在 GDB 中跟踪程序执行流，发现当输入为 5 时，程序跳转到了对应的处理代码块：

```
1614: mov    $0x0, %eax           ; 初始化累加器
1619: jmp    15c0                 ; 跳转到公共计算逻辑
...
15c0: sub    $0x1ca, %eax         ; eax = 0 - 458 = -458
15c5: add    $0x1ca, %eax         ; eax = -458 + 458 = 0
15ca: sub    $0x1ca, %eax         ; eax = 0 - 458 = -458
```

经过一系列加减运算，寄存器 `%eax` 的最终值为 `-458`。

3. 最终校验：

代码最后将计算结果 %eax 与用户输入的第二个整数进行比较。因此，当第一个输入为 5 时，第二个输入必须为 -458。

phase_4: 递归函数与栈帧分析

```
// 答案
31 CB
```

解题思路与深度分析：

本关考察了递归函数的调用机制及栈帧的使用。

1. 递归逻辑还原：

phase_4 函数调用了一个名为 func4_1 的子函数。分析 func4_1 的汇编代码：

```
163e: cmp    $0x1, %edi
1641: je     1658                 ; 基准条件：若 n <= 1，直接返回
...
164a: call   1633 <func4_1>       ; 递归调用：func4_1(n-1)
164f: lea    0x1(%rax,%rax,1), %eax ; 计算返回值：result = 2 * result + 1
```

这是一个经典的递归函数，其数学表达为：

```
f(1) = 1
f(n) = 2 * f(n-1) + 1
```

主函数传入的参数为 5。推导如下：

f(1)=1 -> f(2)=3 -> f(3)=7 -> f(4)=15 -> f(5)=31。

因此，第一个整数答案为 31。

2. 字符串模拟：

除了整数检查，本关还通过 sscanf 读取了一个字符串，并调用了 func4_2 和 strings_not_equal。通过打印信息 "Ancient monks..." 推测这部分逻辑与汉诺塔问题有关，旨在计算第 15 步移动的源柱和目标柱。

为了避免繁琐的手动推导，我继续使用 GDB 动态调试。在 strings_not_equal 调用前查看 %rsi 寄存器（存放预期字符串），发现其值为 "CB"。

phase_5: 数组索引寻址与位运算

```
// 答案
-2 21
```

解题思路与深度分析：

本关利用内存数组作为查找表，考察了指针跳跃逻辑及位运算。

1. 输入预处理与检查：

入口处有一条极易被忽视的指令：

```
17e1: js     17e8                 ; 若结果为负 (SF=1) 则跳转
17e3: call   explode_bomb         ; 否则爆炸
```

这意味着第一个输入必须是**负数**。随后，程序通过 `and $0xf, %eax` 取输入的低 4 位作为初始索引。

2. 指针跳跃循环：

核心逻辑是一个 do-while 循环，利用当前值作为下一次访问的索引：

```
180c: mov    (%rsi,%rax,4), %eax  ; next_index = array[current_index]
```

循环终止条件是取出的值为 `15`。此外，程序还检查了循环次数（步数）必须**恰好为 2**。

3. 逆向推导：

利用 GDB 的 x/16wd 命令导出数组内容，进行逆向追踪：

- 目标值：`15`（位于索引 6）。
- 前一步：寻找值为 6 的元素，发现位于索引 14。
- 路径确定：`Index 14 -> Val 6 -> Val 15`。

4. **答案构造**：

- 第一个数需满足两个条件：是负数，且低 4 位为 14。计算得 `-2`（二进制补码末尾为 `...1110`，即 14）。
- 第二个数需等于路径上所有值的和：`6 + 15 = 21`。

phase_6: 链表结构与指针操作

```
// 答案
5 6 1 2 3 4 unlock
```

解题思路与深度分析:

本关是常规关卡中最复杂的一个，涉及对 C 语言结构体（链表节点）的汇编级操作。

1. 数据结构识别:

   汇编代码中频繁出现 0x8(%rbx) 这种偏移量操作，且用于存储地址。结合上下文（指针移动），推断这是访问链表节点的 next 指针（在 64 位系统中指针占 8 字节）。

2. 重组逻辑:

   程序读取 6 个数字（要求为 1-6 的全排列），通过复杂的指针操作，按照输入数字的顺序重新连接了内存中的链表节点。这是一个物理上的重组过程。

3. 排序检查:

   最后一段循环遍历重组后的链表，检查节点值的顺序:

   ```
   193e: cmp    %eax, (%rbx)          ; 比较当前节点值与下一节点值
   1940: jge    192f                  ; 若 Current >= Next 则继续
   ```

   指令 `jge` (Jump if Greater or Equal) 表明要求链表节点的值按**降序**排列。

4. 解题步骤:

   使用 GDB x/24wd 命令查看链表所有节点的数值，得到:

   Node1(485), Node2(386), Node3(228), Node4(165), Node5(637), Node6(624)。

   将这些值按降序排列：637 > 624 > 485 > 386 > 228 > 165。

   对应的原始节点索引顺序即为答案：5 6 1 2 3 4。

5. Secret Phase 触发:

   在分析 phase_defused 函数时，发现它在检查完 6 个炸弹后，还会读取紧跟在 Phase 6 输入之后的一段内存，并与字符串 "unlock" 进行比对。因此，在 Phase 6 的答案后追加 unlock 即可触发隐藏关。

Secret Phase: 深度优先搜索 (DFS) 与回溯

```
// 答案
ccaac
```

解题思路与深度分析:

这是一个基于网格的迷宫寻路问题（Knight's Tour 变种），考察了 DFS 算法的汇编实现。

1. 栈上造表:

   func7 函数开头有大量的 movl 指令将立即数存入栈中。分析发现这是在动态构建一个查找表，定义了字符 a-h 对应的 8 种 (dx, dy) 移动向量（类似于国际象棋马的走法）。

2. 地图还原:

   汇编中引用了 row0 等符号。通过 GDB x/40wd 导出这些内存数据，可以将其还原为一个二维 0/1 矩阵，其中

1 代表障碍物。

3. 双重碰撞检测（难点）：

   代码中有两处 cmpb $0x1 的检查，这是极其隐蔽的陷阱：

   - 第一处检查**目标落脚点**是否有障碍。
   - 第二处检查**移动路径中间点**（即“马腿”位置）是否有障碍。

   ```
   1b6d: cmpb   $0x1, (%rdx,%rax,1)  ; 检查中间点
   ```

   我最初尝试的路径 `bbcch` 失败，正是因为忽略了这一细节，导致在某一步虽然落点安全，但绊到了马腿。

4. 解题：

   我在纸上绘制了网格和障碍物，寻找从起点 (0,0) 到终点 (4,7) 的路径。通过手动模拟，确保每一步的落点和“马腿”都不碰到障碍物 1。最终得出的合法路径编码为 ccaac。

反馈/收获/感悟/总结

本次 Bomb Lab 是一次极具挑战性的系统级编程实践，极大地加深了我对计算机底层机制的理解。

1. **汇编语言的具象化**：通过逆向分析，我不再将汇编视为枯燥的代码，而是能从中看到高级语言的影子——循环、递归、结构体指针、跳转表等。我学会了如何从汇编指令推断出原始 C 代码的逻辑结构。
2. **调试技能的飞跃**：我熟练掌握了 GDB 的高级用法，如查看动态内存基址、监视寄存器变化、检查栈帧内容等。特别是学会了如何在运行时动态计算内存地址（RIP 相对寻址），这在解决 Secret Phase 时起到了决定性作用。
3. **严谨思维的培养**：在 Secret Phase 中，仅仅理解大体算法是不够的，必须精确到每一条 `cmp` 指令的含义（如“绊马腿”检查）。这种对细节的极致关注，是系统编程的核心素养。

参考的重要资料

1. **《深入理解计算机系统 (CSAPP)》** - 第三章：程序的机器级表示（特别是关于过程调用和数据结构的章节）。
2. **GDB Documentation** - 查阅了关于内存检查格式 (`x`) 和寄存器查看的指令。
3. **x86-64 Instruction Set Reference** - 用于查询特定汇编指令（如 `lea`, `test`, `movzbl`, `cltq` ）的具体行为和标志位影响。