**深圳大学实验报告**

**课程名称： 计算机系统(2)**

**实验项目名称： 逆向工程实验**

**学院： 计算机与软件学院**

**专业： 计算机科学与技术**

**指导教师： 马晨琳**

**报告人： 李文俊 学号： 2023150001 班级： 高性能班**

**实验时间： 2025年4月16日～ 2025年4月30日**

**实验报告提交时间： 2025年4月 29 日**

**教务处制**

|  |
| --- |
| **一、实验目的：**   1. 理解程序（控制语句、函数、返回值、堆栈结构）是如何运行的 2. 掌握GDB调试工具和objdump反汇编工具 |
| **二、实验环境：**   1. 计算机（Intel CPU） 2. Linux64位操作系统 3. GDB调试工具 4. objdump反汇编工具 |
| **三、实验内容**  本实验设计为一个黑客拆解二进制炸弹的游戏。我们仅给黑客（同学）提供一个二进制可执行文件bomb和主函数所在的源程序bomb.c，不提供每个关卡的源代码。程序运行中有6个关卡（6个phase），每个关卡需要用户输入正确的字符串或数字才能通关，否则会引爆炸弹（打印出一条错误信息，并导致评分下降）！  要求同学运用**GDB调试工具和objdump反汇编工具**，通过分析汇编代码**，**找到在每个phase程序段中，引导程序跳转到“explode\_bomb”程序段的地方，并分析其成功跳转的条件，以此为突破口寻找应该在命令行输入何种字符串来通关。  本实验要求解决Phase\_1、Phase\_2、Phase\_3、Phase\_4、Phase\_5、Phase\_6。通过截图把结果写在实验报告上。 |
| **四、实验步骤和结果**  1.反汇编可执行程序bomb\_64.dat  **1.1输入反汇编命令查看汇编代码**（保存在bomb.txt文件中）:  ***$ objdump -d bomb\_64.dat > 1.txt***    **图 1**  **具体汇编代码如下：**    **图 2**  **1.2 输入objdump -t bomb\_64.dat | less 可查看可执行文件的符号表**    **图 3**  **输出 /bomb 查看和bomb相关的符号**  **bomb.c**  **initializebombsolve**  **explodebomb**  **bombid**  **initialize\_bomb**  2. Phase\_1  2.1 观察main函数，找到phase\_1相关信息    **图 4**  主函数通过调用 <read\_line> 将返回值赋值给%rdi, phase\_1地址为400e70  **查看phase\_1具体汇编代码:**    **图 5**  **Q: 第一个参数%rdi是什么呢？**  2.2 对可执行文件进行gdb调试  **gdb bomb\_64.dat**    **图 6**  **break phase\_1 :** 对phase\_1设置breakpoint，**run直接跳转**  程序会提示你输入炸弹秘密，第一次先输入测试“666”    **图 7**  **可以通过disas 查看汇编代码,通过info registers 查看寄存器的值，stepi逐步测试**    **图 8**  **当程序运行到跳转函数时，查看寄存器的值，得知%rdi=0x602f40,%rsi=0x401af8**  **通过 p (char \*) 0x… 输出得到 %rdi表示的就是输出的密文 “666”，%rsi输出的是一个字符串**    **图 9**  保存到文件中，避免输入产生的错误    **图 10**  2.3 分析 <string\_not\_equal>具体实现    **图 11**  **程序先比较两个字符串长度是否相等，然后判空，最后逐个字符判断两个字符串是否相等,如果不相等返回1.**    **图 12**  **因此，可以得出结论：第一关的密码就是%rsi存储的字符串**  ***Science isn't about why, it's about why not?***  结果：    **图 13**  3.Phase\_2  **节约时间的小技巧**：在同一个目录下创建一个ans.txt文件，按行存储密码，这样，只需要 **r ans.txt**，就能够自动输出密码，不用每次都重复输入。  3.1 设置断点 phase\_2  b phase\_2 并且disas查看phase\_2代码    **图 14**  程序先分配栈帧，然后调用 read\_six\_numbers来读取6个数字  3.2 设置断点 b read\_six\_numbers    **图 15**  看到函数分配0x18栈空间  发现函数分配了一个特殊值$0x401eb2给%esi，输出看一下具体是什么    **图 16**  正好对应了函数需要读六个numbers，%d说明数据类型为int，对应4字节，总数就是0x18  因此，这个函数的作用是读取六个int，目的寄存器对应x86中规定的四个寄存器 %rdx %rcx %r8 %r9  3.3分析read\_six\_numbers函数  read\_six\_numbers 函数的本质是通过 **sscanf** 将输入字符串解析为六个整数，并将它们连续存储到调用者 phase\_2 的栈空间，在进入 read\_six\_numbers 前：**%rsi 已被设置为 phase\_2 的栈顶地址 0x7fffffffde20。**  **sscanf 需要为每个整数提供存储地址：**  第一个参数（%rdi）：输入字符串地址（由 read\_six\_numbers 的调用者传入）。  第二个参数（%rsi）：格式字符串地址 0x401eb2（格式为 "%d %d %d %d %d %d"）  后续参数（%rdx, %rcx, %r8, %r9, 栈...）：六个整数的存储地址  这是比较反常识的一点，如果不理解sscanf，可能就搞不懂%rdi,%rsi其实已经作为参数了。  画出栈的分配对理解函数参数读取尤为重要  **地址 存储的值 对应输入元素**  **0x7fffffffde20 %rdx 第一个参数**  **0x7fffffffde24 %rcx 第二个参数**  **0x7fffffffde28 %r8 第三个参数**  **0x7fffffffde2c %r9 第四个参数**  **0x7fffffffde30 栈顶 第五个参数**  **0x7fffffffde34 栈顶+8 第六个参数**  3.4 分析循环逻辑  rbp 初始指向 rsp（第一个元素），r13 指向 rsp+0xc（第四个元素）。  循环三次，每次比较当前元素（rbp 指向）与后三个位置的元素（如 rbp+0xc）是否相等：  第 1 次：元素 1 vs 元素 4  第 2 次：元素 2 vs 元素 5  第 3 次：元素 3 vs 元素 6  同时累加前三个元素值到 r12d。    **图 17**  3.5 结果  元素相等：必须满足 元素 i == 元素 i+3（i=1,2,3）。  非零和：前三个元素之和 r12d != 0  e.g. 1 1 1 1 1 1 就是正确的    **图 18**  4.Phase\_3  4.1 分析主函数  首先还是找到一个特殊的数字0x401ebe赋值给了%esi，输出看一下    **图 19**  得到函数需要输入两个整数，然后调用sscanf得到两个整数，所以**第一次判断**和phase\_2读取六个整数的逻辑相同，通过%rax判断输入的数量，因此可以输入两个整数用作测试    **图 20**  继续分析遇到第二个判断条件：**cmpl $0x7,0xc(%rsp)，**%rdx对应第一个参数，**因此要保证第一个值<=7**  接着将%rdx存到%rax中，函数出现了一个和之前学的switch结构非常常用的一个跳转语句  **jmp \*0x401b60(,%rax,8) 使用一个 间接内存寻址 jmp \*基地址(偏移寄存器, 比例因子)**  **Dst = \*(0x401b60 + %rax \* 8) ，因此，需要知道%rax从0-7对应跳转的不同地址**  GDB查看指定内存地址的内容——指令x：p/x \*(address)    **图 21**  **其他同理，得到跳转地址后，%eax被赋予特定值，最后和第二个参数比较，相等则成功拆弹。**   |  |  | | --- | --- | | **%rdx** | **%eax** | | **0** | **535** | | **1** | **926** | | **2** | **214** | | **3** | **339** | | **4** | **119** | | **5** | **352** | | **6** | **919** | | **7** | **412** |   4.2 结果  输入6 919    **图 22**  5.Phase\_4  5.1 分析主函数  经典奇怪数字$0x401ec1赋值给%esi，输出看一下    **图 23**  这一次只需要输出一个整数，接着经典%rax判断输入个数大于1,然后和0比较，需要大于零，这些和phase\_3一模一样。  把输入传给%rdi，调用fuc4  5.2 分析fun4  分析func4得到函数在进行递归调用，一般递归调用都要先保存传入的参数，这里保存到%rbx    **图 24**  写成C语言形式：经典的斐波那切数列    **图 25**  5.3 回到主函数  主函数需要结果返回值要等于0x37（十进制55），所以x应该是9（1 2 3 5 8 13 21 34 55）  5.4 结果  输入9，得到：    **图 26**  6.Phase\_5  6.1 分析主函数  经典奇怪数字$0x401ebe赋值给%esi，已经能推断出是两个输入，输出看一下具体类型    **图 27**  输入1（x） -> %rdx 输入2 （y） ->%rcx  **第二次判断**：x = x & 0xf , 低四位等于0xff 引爆 —— **x的低四位必须不是全1**  6.2 分析循环    **图 28**  即经过&操作后的%eax在0~14，经过分析，**数据结构是链表**，需要顺序遍历链表12个元素，最后到15，输出内存的连续15个元素，按首尾连在一起得到：  **5 -> 12 -> 3 -> 7 -> 11 -> 13 -> 9 -> 4 -> 8 -> 0 -> 10 -> 1 -> 2 -> 14 -> 6 -> 15(End)**    **图 29**  6.3 结果  **11+13+9+4+8+0+10+1+2+14+6+15= 93**  开始输入的是 11 93 ，发现不对，掉进了一个经典的循环结构的坑里：**最后15确实参与累加，但是计数器到12的元素是6，因此计数器为1的值是7，每次累加的是计数器的后一个元素**  **输入 7 93**    **图 30**  7.Phase\_6  7.1 分析主函数  **call 400b80 <strtol@plt> ：strtol@plt核心作用是将 字符串转换为Long类型数值(8bytes)**  **参数传递：**  **%rdi 存储待转换的字符串地址**（输入的 "123"）。  **%rsi**可能设置为 NULL 或一个用于接收终止位置的指针。  **%rdx**设置基数（如 10 表示十进制，16 表示十六进制）。  **返回值**：转换后的长整型数值存储在 %rax 中。  **所以函数在调用之前要设置%edx=10,%rsi=0**  mov %eax,0x20168e(%rip)存储在node<0>位置，**这里要注意虽然strtol得到的是long类型，但是movl指令把它截断成了int类型。**  推测它是一个链表, $0x602780 根据提示可能对应链表的首地址，输出看一下是怎么存储的  可以使用指令 p \*address@n ，连续输出n个字节的值    **图 31**    **图 32**  GDB默认是按4字节为单位显示，最后的0表明小端序存储  7.2 分析func6  func6实现了**对一个单链表进行降序排序，采用类似于插入排序的算法**  **初始状态：**%rdi 初始指向链表头节点地址 0x602780  %r8 存储头节点的下一个节点地址（mov 0x8(%rdi), %r8）  **头节点的 next 指针被置空**（movq $0x0, 0x8(%rdi)），为后续排序做准备  ​**主循环逻辑：**  ​**1. 遍历未排序节点**：每次处理一个节点（由%r8指向），将其插入到已排序链表的正确位置。  ​**2. 比较节点值：**  若当前已排序链表节点的值（%rcx指向）大于待插入节点（%r8指向），则继续向后查找插入位置。  若遇到值小于或等于待插入节点的已排序节点，或到达链表末尾，则将待插入节点插入到当前位置之后。  ​**3.插入操作：**  ​**调整指针：**将待插入节点（%r8）的next指向当前节点的下一个节点，并将当前节点的next指向待插入节点，完成插入。  ​**迭代处理：**更新%r8为下一个待处理的节点，重复上述过程，直到所有节点处理完毕。输出执行完func6的结果验证分析  7.3 结果  **主函数连续三次寻址 M[M[M[8+%rax]+8]+8] ，即访问第四个元素，意味着输入的元素要处于第四大，即 600 <= x <673**  **输入600**    **图 33**  8.Secret\_Phase  8.1 进入secret\_phase  在每次拆出炸弹，都会调用<phase\_defused>，所以在这里分析要怎么进入secret\_phase    **图 34**  输出一些特殊数字，得到需要输入值为 一个整数和一个字符串，%rdi是输入字符串地址，字符串对应<strings\_not\_equal>中给定的字符串**austinpowers**    **图 35**    **图 36**  最开始因为只有在phase\_6后才会进入到这个判断中，所以在6的答案里加入字符串，发现不行，要在phase\_4也加上字符串，phase\_6不需要加。    **图 37**  8.2 分析Secret\_phase  **关键判断：**  **1.输入<=1001**  **2.传入一个地址给%rdi，作为func7的第一个参数，输出具体值类似链表**    **图 38**  **3. fuc7的返回值必须等于3**  8.3 分析func7  函数内部采用了递归，开始时地址判空，每次递归都会读取 Node \*next,然后通过查看内存分布，可以推断出这应该是一个类似二叉树的数据结构，根据完整内存地址画出整棵树    **图 39**    **图 40**    **图 41**  化为C语言：    **图 42**  8.4 问题是怎么让返回值等于3呢？  **如果x不在这些数据中返回-1，所以输入一定是节点的某一个值**  如果x < p->value ，需要得到 func7返回3/2 ，那必不可能，第二个返回0也不行  **所以第一层只能大于value，需要第二层返回1，同理第二层返回1，必须第三层返回0，所以x的值为连续两次走右子树**    **图 43**  8.5 Answer  输入107    **图 44** |
| **三、实验总结与体会：**  这个实验就像一场用汇编语言写的“密室逃脱”——每一关都埋着逻辑炸弹，必须拆解代码才能逃生。虽然一开始对着满屏的mov、jmp、cmp指令头皮发麻，但越玩越上瘾，简直是**用硬核方式学编程**  **1. 逆向工程：当一次“代码侦探”​** 实验把C语言写的炸弹程序编译成汇编，逼着我们像侦探一样，从call、jne这些指令的“脚印”里，倒推出程序原本的逻辑。比如看到test和je组合，就猜到是“如果条件满足就爆炸”；遇到add和mov来回操作，可能是在算数或者倒腾数据。​**课本上死记硬背的指令突然活了**，每条代码都变成破案线索，拆弹成功时简直感觉自己强的可怕。  **2. GDB工具：开了“上帝视角”​** 以前觉得GDB是程序员用的高级玩意儿，这次直接真香！用p命令偷看寄存器的值，break在关键代码设路障，stepi像显微镜一样观察程序怎么一步步跑——**仿佛给代码装上了监控摄像头**。最爽的是Phase 5，眼看着链表被拆解成索引，从内存里“拼图”一样捞出正确密码，妥妥的黑客既视感！  **3. 汇编实战：把知识“焊”进脑子里** 课本第三章的指令（比如cmp比较、jmp跳转、lea算地址）全用上了！以前觉得“学这玩意儿干啥”，这次直接被教做人。比如Phase 4的递归函数，光看C代码可能10分钟就懂，但对着汇编得硬生生从call和ret的嵌套里脑补出递归逻辑——**像把数学公式拆成加减乘除再做一遍**，虽然痛苦，但打通任督二脉后看啥代码都不怕了。  **4. 系统思维：程序竟是“精细的机械表”​** 实验最震撼的是看到**程序如何用内存和寄存器搭积木**。比如Phase 2的循环验证，栈内存像快递柜一样整齐排着6个数字；Phase 6的链表操作，每个节点占多少字节、指针怎么跳转，全得自己算清楚。​**原来高级语言里轻轻一句**if-else**，底层竟是几百条指令的精密配合**，计算机系统的美感拉满！  这场实验就像给大脑装了个“汇编编译器”——从前觉得玄乎的指针、内存、函数调用，现在看代码都能脑补出它们在CPU里蹦迪的样子。虽然被炸弹虐到熬夜秃头，但看到最终通关的Congratulations!时，直接理解了什么叫做**​“痛苦是暂时的，牛皮是一辈子的”​**​ |

|  |
| --- |
| **指导教师批阅意见：**  **成绩评定：**  指导教师签字：马晨琳  2025年5月 日 |
| 备注： |