**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： 数计**

**指导教师： 罗胜**

**报告人： 詹耿羽 学号：2023193026 班级： 数计**

**实验时间： 2025年4月18日至5月15日**

**实验报告提交时间： 2025年4月27日**

**教务处制**

|  |
| --- |
| **一、 实验目标与要求：**   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（Ubuntu 17） 3. GDB调试工具 4. objdump反汇编工具 |
| **三、实验方法与步骤：**  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\_64和主函数所在的源程序bomb\_64.c，不提供每个关卡的源代码。程序运行中有6个关卡（6个阶段），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**，**找到在每个阶段程序段中，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。  本实验需解决阶段\_1(**15分**)、阶段\_2(**15分**)、阶段\_3(**15分**)、阶段\_4(**15分**)、阶段\_5(**15分**)、阶段\_6(**10分**)。通过**截图+文字**的形式把实验过程写在实验报告上，最后并撰写**实验结论与心得(15分**)。 |
| **四、实验过程及内容：**  输入反汇编命令，将汇编代码保存在1.txt文件中  ***$ objdump -d bomb > 1.txt***    图 1 反汇编  然后找到main函数，发现它调用了从阶段1到阶段6这六个函数。下面对这六个函数进行分析求解。   1. **阶段1**     图 2 阶段1代码  1）分析汇编代码    图 3 分析阶段1代码  通过分析，猜测string\_not\_equal是一个比较两个字符串是否相等的函数，若相等则返回0，若不相等则返回1。其中，由于%esi传参寄存器存入了一个地址，这可能是用于比较的字符串的起始地址，用于与我们输入的字符串进行比较。  2）利用gdb调试工具来调试阶段1  ***$ gdb bomb***  3）在0x400e74处设置断点，运行并打印0x401af8地址的值    图 4 调试阶段1代码（1）  可以看到，0x401af8存有字符串“Science isn't about why, it's about why not?”。   1. 在0x400e7e处设置断点，根据输入的“test”，查看返回值%eax。     图 5 调试阶段1代码（2）  可以看到，%eax的返回值为1，同时继续运行程序，则会导致炸弹爆炸。   1. 重新运行程序，输入“Science isn't about why, it's about why not?”，查看返回值%eax。     图 6 输入阶段1答案  可以看到，%eax的返回值为0，同时继续运行程序，则会成功跳过炸弹的引爆。  综上，阶段1函数检测输入的字符串是否与“Science isn't about why, it's about why not?”相等，若相等则跳过炸弹的引爆。   1. **阶段2**     图 7 read\_six\_numbers代码    图 8 阶段2代码   1. 可以看到，在进入阶段2函数后，首先让栈开辟出空间，然后调用read\_six\_numbers函数，推测通过read\_six\_numbers函数通过读入六个数后，将六个数存入栈中。     图 9 分析read\_six\_numbers代码  在read\_six\_numbers函数内0x401767处设置断点，运行程序至断点处，查看传入\_\_isoc99\_sscanf@plt的参数0x401eb2所对应的字符串。    图 10 调试read\_six\_numbers代码  可以看到，传入的参数为"%d %d %d %d %d %d"，即指定\_\_isoc99\_sscanf@plt函数读入六个int型整数。  在阶段2函数0x400eac处设置断点，运行程序，输入“1 2 3 4 5 6”，查看调用read\_six\_numbers函数后，栈的存储情况。    图 11 调试阶段2代码（1）  可以看到，读入的六个数按顺序，从%rsp指向的地址开始，以每4字节一个数的方式存储在栈中。   1. 分析汇编代码     图 12 分析阶段2代码  可以发现，该函数使用%rbp作为迭代指针，%rbp+4\*3为迭代终点，即对前三个输入的数进行迭代，一共迭代三次，而每次都将M[%rbp] 和 M[%rbp+12] 处的整数进行比较，如果相等则跳过炸弹的引爆；接着检查M[%rbp]及其之前迭代过的所有数的累加和是否为0，若不为0，则跳过炸弹的引爆。每次迭代结束后，%rbp=%rbp+4，即栈指针后移，指向下一个数。   1. 分别在对阶段2的三次测试中输入“1 1 1 2 2 2”，“0 0 0 0 0 0”，“5 5 5 5 5 5”。     图 13 调试阶段2代码（2）    图 14 调试阶段2代码（3）    图 15 调试阶段2代码（3）  可以发现，前两次测试都会导致炸弹引爆，最后一次测试才跳过炸弹的引爆。  综上，阶段2函数要求输入的六个int型整数，若第i个数和第i+3个数（i <= 3）相等且第1个数、第1个数+第2个数、第1个数+第2个数+第3个数均不为0则跳过炸弹的引爆。   1. **阶段3**     图 16 阶段3代码   1. 分析阶段3的输入     图 17 分析阶段3代码  可以看到，在进入阶段3函数后，栈开辟出新空间，且把两个栈地址传入输入函数，调用完输入函数，其返回值与1比较，若不大于1则会导致炸弹引爆。  在0x400f07处设置断点，运行程序至断点处，检查另一个传入输入函数的字符串参数，检查结果为“%d %d”，确定阶段3函数要求输入两个int型整数。    图 18 调试阶段3代码（1）  在0x400f16处设置断点，运行程序，输入“1 2”，检查调用完输入函数后，作为参数的两个栈地址内存储的值，能够明确输入的第一个整数存放在M[%rsp+0xc]中，第二个整数存放在M[%rsp+0x8]中。    图 19 调试阶段3代码（2）   1. 分析汇编代码     图 20 分析阶段3代码  可以看到，阶段3要求第一个输入的数不能大于7，然后跳转到目的地址M[0x401b60 + 第一个数\*8]后，将%eax赋对应值，并跳转到0x400f74判断是否与第二个输入的数相等，如果不等则爆炸。因此，可以确定程序为switch型程序，现在只需找出switch对应跳转的8个地址，再对应该地址对%eax的赋值，即可确定输入的第一个数对应的第二个数，使得程序跳过炸弹的引爆。    图 21 调试阶段3代码（3）  根据上面的输出结果，可以找到跳转地址对应%eax的赋值。   |  |  | | --- | --- | | 地址 | %eax的赋值 | | 0x0000000000400f32 | 535 | | 0x0000000000400f6f | 926 | | 0x0000000000400f39 | 214 | | 0x0000000000400f40 | 339 | | 0x0000000000400f47 | 119 | | 0x0000000000400f4e | 352 | | 0x0000000000400f55 | 919 | | 0x0000000000400f5c | 412 |   表 1 跳转地址对应赋值   1. 根据上表可以得到使炸弹不爆炸的输入的第一个数和第二个数的对应关系。  |  |  | | --- | --- | | 输入的第一个数 | 输入的第二个数 | | 0 | 535 | | 1 | 926 | | 2 | 214 | | 3 | 339 | | 4 | 119 | | 5 | 352 | | 6 | 919 | | 7 | 412 |   表 2 对应关系    分别在对阶段3的三次测试中输入“0 535”，“1 926”，“2 214”。    图 22 调试阶段3代码（4）    图 23 调试阶段3代码（5）    图 24 调试阶段3代码（6）  可以看到，三次测试均跳过炸弹的引爆，说明以上分析正确。  综上，阶段3为switch型程序，输入的第一个数是决定switch跳转位置的下标，输入的第二个数要与switch跳转位置对应的值相等，才能跳过炸弹的引爆。   1. **阶段4**     图 25 阶段4代码   1. 分析阶段4的输入     图 26 分析阶段4代码（1）  在阶段4函数内0x400fca处设置断点，运行程序到断点处，查看传入输入函数的字符串参数，检查结果为“%d”，确定阶段4函数要求输入一个int型整数。    图 27 调试阶段4代码（1）  在0x400fd9处设置断点，运行程序，输入“8”，检查调用完输入函数后，作为参数的栈地址内存储的值，能够明确输入的整数存放在M[%rsp+0xc]中。    图 28 调试阶段4代码（2）   1. 分析汇编代码   在阶段4函数中，可以看到，输入的数必须大于0，且传入func4函数后，返回值为55则跳过炸弹的引爆。    图 29 分析阶段4代码（2）  在func4函数中，可以看到，传入的参数与1对比，小于等于1则跳转到0x400fb2，退出函数并返回1。否则将传入的参数减一后调用func4函数，再将返回值存放在%ebp里，然后将参数减二后再次递归调用func4函数，将返回值与上一次递归得到的返回值%ebp相加存在%eax中，相当于返回值%rax = func4(%rdi-1) + func4(%rdi-2)，最终退出函数。    图 30 分析阶段4代码（3）  因此可以确认，func4是一个递归函数，根据传入的参数值，返回斐波那契数列中对应下标的值。根据斐波那契数列 1 1 2 3 5 8 13 21 34 55 …… 可以知道，当在阶段4调用func4函数传入的参数值为9时，func4函数返回值才为55。  3）输入9验证分析。    图 31 调试阶段4代码（3）  可以看到，输入的数为9，则可以跳过炸弹的引爆。  综上，func4是一个求斐波那契数列的函数，在阶段4中，根据输入值和func4求出斐波那契数列中对应下标的值，与55比较，若相等，则跳过炸弹的引爆。根据斐波那契数列，我们可以知道，输入的值必须为9。   1. **阶段5**     图 32 阶段5代码   1. 分析阶段5的输入     图 33 分析阶段5代码（1）  在阶段5函数内0x401010处设置断点，运行程序到断点处，查看传入输入函数的字符串参数，检查结果为“%d %d”，确定阶段5函数要求输入两个int型整数。    图 34 调试阶段5代码（1）  在0x40101f处设置断点，运行程序，输入“88 99”，检查调用完输入函数后，作为参数的栈地址内存储的值，能够明确输入的整数分别存放在M[%rsp+0xc]和M[%rsp+0x8]中。    图 35 调试阶段5代码（2）   1. 分析汇编代码   根据代码可以知道，首先函数对第一个输入数取16余数，如果第一个输入数取余后等于15，将直接引爆炸弹。    图 36 分析阶段5代码（2）  接着函数进入一个do-while循环过程，循环至%eax=15才停止循环。每轮循环中，%edx记录下循环的次数，%eax根据计算出的地址被重新赋值，%ecx则为每轮循环%eax的累加和。    图 37 分析阶段5代码（3）  打印后发现，从地址0x401ba0开始连续存有16个int型整数，结合每轮循环中%eax=\*(0x401ba0 + %eax\*4)的赋值，可以确认，每轮循环%eax会根据当前的值，被赋值为以下16个数中的其中一个值，而且还能发现，%eax的赋值一定按着是5->12->3->7->11->13->9->4->8->0->10->1->2->14->6->15这个顺序进行赋值的，即只要%eax的初始值不为15，则根据循环赋值%eax的值一定能变为15。    图 38 调试阶段5代码（3）  根据下面的代码，可以知道，如果循环次数不为12，则炸弹会引爆，所以%eax的初始值，即输入的第一个数的初始值一定要为7，才能保证循环12次后正好退出循环。    图 39 分析阶段5代码（4）  同时，如果输入的第二个数不等于这12轮中%eax的累加和，也会引爆炸弹。根据下表，可以知道，当结束12轮循环后，%ecx的值为93。所以输入的第二个数为93。   |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | 轮次 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | %eax | 7 | 11 | 13 | 9 | 4 | 8 | 0 | 10 | 1 | 2 | 14 | 6 | 15 | | %ecx | 0 | 11 | 24 | 33 | 37 | 45 | 45 | 55 | 56 | 58 | 72 | 78 | 93 |   表 3 每个轮次%eax和%ecx的变化   1. 输入“7 93”，检验以上分析是否正确。     图 40 调试阶段5代码（4）  可以看到，输入为“7 93”，则可以跳过炸弹的引爆。  综上，阶段5有连续的地址跳转操作，要求输入的第一个数满足跳转的次数限制，第二个数满足跳转过程中累加和的限制。   1. **阶段6**     图 41 fun6代码    图 42 阶段6代码   1. 分析阶段6的输入   通过代码可以看到调用了strtol@plt函数，这个应该是用于读入输入的。    图 43 分析阶段6代码（1）  在0x4010f2处设下断点，输入“999”，运行程序至断点处，可以发现，strtol@plt函数的返回值正好是输入的数，且把输入的数赋值到地址为0x20168e + 0x4010f2 = 0x602780、名为node0的内存空间处。    图 44 调式阶段6代码（1）  由上面给出的地址0x602780查看该内存位置存储的内容，可以发现，从0x602780开始存有一个长度为10的链表，一个节点占16个字节空间，其中第一个4字节存节点的值，第二个4字节存节点的下标，最后两个四字节存有节点指向下一个节点的地址。此时，刚刚我们输入的数作为头节点node0的值，链表连接按地址从小到大的顺序。    图 45 调式阶段6代码（2）  在0x4010fc处设置断点，查看经过func6函数后链表的变化情况。能够发现，链表的连接顺序发生了变化，链表连接顺序为999->826->782->673->600->529->488->374->370->286,不难发现，此时链表是按节点值从大到小的顺序相连的，说明func6对链表进行了降序排序。    图 46 调式阶段6代码（3）    打印func6返回的值%rax，可以发现返回的是排序后的链表头，即节点值最大的节点。    图 47 调式阶段6代码（4）  在分析阶段6调用函数后的代码，可以知道，阶段6对链表排序后，检查第四个节点值与输入值是否相等，相等则不引爆炸弹。其实等价于输入的值赋值给节点node0后，node0是否为降序排序后链表中的第四个节点，若是则跳过炸弹的引爆。    图 48 分析阶段6代码（2）  3）分析func6的功能  进入func6函数首先是对寄存器进行赋值，其中%rax = %rcx = node0的地址，%r8 = node0指向的下一节点的地址。若%r8赋值后不为0，则不退出func6函数。    图 49 分析fun6代码（1）    接下来的一段代码翻译成高级语言是一段while循环代码。\*(%rcx + 0x8)应该是遍历链表的操作，即%rdx指向下一个节点，遍历结束的条件是%rdx指向节点的值小于等于%rsi。    图 50 分析fun6代码（2）   |  | | --- | | 以上汇编代码翻译成高级语言：  %rdx = \*(%rcx + 0x8)  while(%rdx != 0 && \*rdx > %rsi){  %rcx = %rdx  %rdx = \*(%rcx + 0x8)  }  %rdi = %rcx |   接下来的一段代码翻译成高级语言是一段if-else结构代码。    图 51 分析fun6代码（3）   |  | | --- | | if(%rdx == %rdi)  %rax = %r8  else  \*(%rdi + 0x8) = %r8 |   最后一段代码则是给出了退出func6函数的条件，同时如果未能退出func6，则会根据寄存器的情况重新跳转会之前的命令，重复操作。    图 52 分析fun6代码（4）   |  | | --- | | 综上，能够得到func6完整的代码：   1. %r8 = \*(0x8 + %rdi) 2. \*(0x8 + %rdi) = 0 3. %rax = %rdi 4. %rcx = %rdi 5. if(%r8 == 0) 6. %rax = %rdi 7. return 8. while(1){ 9. %rdx = %rcx 10. if(%rcx == 0){ 11. if(%rdx == %rdi) 12. %rax = %r8 13. else 14. \*(%rdi + 0x8) = %r8 15. %rcx = \*(0x8 + %r8) 16. \*(0x8 + %r8) = %rdx 17. if(rcx == 0) 18. return 19. %r8 = %rcx 20. %rcx = %rax 21. %rdi = %rax 22. } 23. else{ 24. %rsi = \*%r8 25. if(\*%rcx > %rsi) 26. %rdx = \*(%rcx + 0x8) 27. while(%rdx != 0 && \*rdx > %rsi){ 28. %rcx = %rdx 29. %rdx = \*(%rcx + 0x8) 30. } 31. %rdi = %rcx 33. if(%rdx == %rdi) 34. %rax = %r8 35. else 36. \*(%rdi + 0x8) = %r8 37. %rcx = \*(0x8 + %r8) 38. \*(0x8 + %r8) = %rdx 39. if(rcx == 0) 40. return 41. %r8 = %rcx 42. %rcx = %rax 43. %rdi = %rax 44. } 45. else{ 46. if(%rdx == %rdi) 47. %rax = %r8 48. else 49. \*(%rdi + 0x8) = %r8 50. %rcx = \*(0x8 + %r8) 51. \*(0x8 + %r8) = %rdx 52. if(rcx == 0) 53. return 54. %r8 = %rcx 55. %rcx = %rax 56. %rdi = %rax 57. } 58. } |  1. 利用gdb调试   显然，光靠翻译代码并不能明了func6的整个工作过程。为了更清楚地了解整个过程，我们可以抓住几个关键的寄存器来查看它们过程中的。同时，由于该函数返回的%rax是链表的头节点，我们可以考虑以%rax为头节点，打印出整个链表在过程中的变化情况。  为此，我们可以写下gdb代码用作调试。   |  | | --- | | 1. define test 2. // 打印各关键寄存器的变化情况 3. printf "rdi = 0x%lx\n", $rdi 4. printf "rcx = 0x%lx\n", $rcx 5. printf "rdx = 0x%lx\n", $rdx 6. printf "rsi = %d\n", $rsi 7. printf "r8  = 0x%lx\n", $r8 8. printf "rax = 0x%lx\n", $rax 9. printf "\n" 11. // 打印10个节点下一指向节点的变化情况 12. set $node = 0x602780 13. set $i = 0 14. while $i < 10 15. printf "Node %d: add = %lx, value = %d, next = 0x%lx\n", $i, $node, \*($node), \*($node+0x8) 16. set $node = $node + 0x10 17. set $i = $i + 1 18. end 19. printf "\n" 21. // 打印以%rax为链表头，整个链表的变化情况 22. set $point = $rax 23. while $point != 0 24. printf "Node%d:%d -> ", \*($point + 0x4), \*($point) 25. set $point = \*($point + 0x8) 26. end 27. printf "\n" 28. end |   我们在0x401081处设下断点，查看刚进入func6的初始情况。并在0x4010b8即能退出func6的地方设下断点，查看每轮循环后的情况。  下面输入“100”，导入gdb调试文件，开始调试代码。  刚进入func6，node0为节点头，此时链表只有node0一个节点。    图 53 调试fun6代码（1）  经过5轮循环调试后可以发现，每一轮循环链表都会加入一个新的节点，并且一直都是按节点值从大到小的顺序维护链表，这相当于插入排序，把每轮新增的节点插入到链表合适的位置。并且我们也能够确认每个寄存器的作用，%rax存链表头节点，%r8存当前遍历的节点的地址，%rsi存当前遍历节点的值，%rcx存下一个要遍历的节点的地址，%rdx、%rdi存临时节点地址，遍历的节点顺序为从node1到node9，由于node9指向下一节点的地址为0，因此遍历完node9后就退出func6。    图 54 调试fun6代码（2）    图 55 调试fun6代码（3）      图 56 调试fun6代码（4）    图 57 调试fun6代码（5）    图 58 调试fun6代码（6）   1. 了解过func6的工作过程后，下面根据上面的分析做出本题的解答。由于要使得node0在排序后位于链表的第四个位置，才能跳过炸弹的引爆。再看node1到node9的降序排序的顺序为826->782->673->600->529->488->374->370->286，因此相当于插入的节点node0必须在值为673的节点和值为600的节点之间，即node0取值必须为[600,673]，输入的数也必须为[600,673]这个范围内。   输入“650”，检验以上分析的正确性，成功通关。    图 59 调式阶段6代码（5）  综上，阶段6是对一个链表进行降序排序，不同的输入会使得链表最终的排序情况不同，因此要求输入的值在一个限定的范围内。   1. **隐藏关**     图 60 fun7代码    图 61 secret\_阶段代码   1. 确认隐藏关入口   在代码中搜索secret\_阶段，查看隐藏关的入口。可以查询到，隐藏关在阶段\_defused中。    图 62 阶段\_defused代码  在0x401789设下断点，运行程序，可以发现每关通过后都会进入这个到这个阶段\_defused函数中，并检查\*(0x2014f0 + %rip)，即\*(0x2014f0 + 0x401790)是否等于6，等于6则进入下面过程，否则退出函数。进过检查可以知道，\*(0x2014f0 + 0x401790)等于当前关卡数，所以当通过阶段6后，才有机会进入到隐藏关。    图 63 调试阶段\_defused代码（1）    图 64 调试阶段\_defused代码（2）  分析接下来的代码，可以发现接下来会调用一个输入函数，输入个数不为2则会退出阶段\_defused函数，否则则继续进入隐藏关的过程。    图 65 分析阶段\_defused代码（1）  在0x4017b0处设下断点，查看两个传入输入函数的参数。不难发现，输入内容为“%d %s”，即一个整数和一个字符串。    图 66 调试阶段\_defused代码（3）  0x603030存着字符“9”，则在一个名为input\_strings的空间内，猜测input\_strings为之前所有输入缓存的地方，由0x603030-240得到input\_strings的起始地址，打印input\_strings的内容，验证了input\_strings为之前所有输入缓存地方的想法。并且可以确认0x603030为阶段4输入内容的地址。    图 67 调试阶段\_defused代码（4）    图 68 调试阶段\_defused代码（5）    图 69 调试阶段\_defused代码（6）    图 70 调试阶段\_defused代码（7）    图 71 调试阶段\_defused代码（8）  结合以上分析，此处代码很可能需要从阶段4输入缓存中读入一个整数和一个字符串，其中整数为阶段4的答案，字符串要符合进入隐藏关的要求，才能进入隐藏关。  接下来便是调用string\_not\_equal函数对输入的字符串进行检验，若字符串与预定的字符串相等，则进入隐藏关，其中传输给string\_not\_equal函数的两个参数为0x10+%rsp和0x401eca。    图 72 分析阶段\_defused代码（2）  在0x4017ba设下断点，在阶段4输入“9 test”，运行程序至断点处，检查0x10+%rsp和0x401eca的内容。可以看到，0x10+%rsp为阶段4中输入字符串的首地址，x401eca为字符串“austinpowers”的首地址。因此可以确认，只有当阶段4中输入字符串的字符串为“austinpowers”，才能进入隐藏关。    图 73 调试阶段\_defused代码（9）  在阶段4输入“9 austinpowers”，检验以上分析。    图 74 调试阶段\_defused代码（10）  成功进入隐藏关。   1. 分析secret\_阶段代码   进入代码，首先是一段用于输入的代码。    图 75 分析secret\_阶段代码（1）    在0x401172设下断点，输入“123”，运行程序至断点处，查看%eax的内容。可以发现，输入的数存储在%eax中。    图 76 调试secret\_阶段代码（1）  根据下面的代码分析可知，输入的数要小于等于1001，才能跳过炸弹的引爆。    图 77 分析secret\_阶段代码（2）  接下来便是传入输入的数和另外一个参数，调用fun7函数，检查返回值是否为3，若不为3则会导致炸弹引爆，若为3则成功通关。    图 78 分析secret\_阶段代码（3）  在0x401185设下断点，查看另一个参数地址在内存中的值。可以看到，0x6025a0地址处被命名为n1，且存的值为36。    图 79 调试secret\_阶段代码（4）   1. 分析fun7代码   首先根据上面的地址0x6025a0打印出这段空间中存有的内容。可以看到，从0x6025a0开始一共有15个节点，每个节点占用32个字节空间，其中每个节点第一个4字节存有一个值，前7个节点第3和第5个4字节存着其他节点的地址，这可能表示与其他节点相连关系。    图 80 调试fun7代码（1）    图 81 调试fun7代码（2）  根据上面每个节点的值和连接情况，可以得到以下二叉树。因此可以明确，fun7传入的第二个参数是该二叉树的根节点，fun7必然是在该二叉树上进行操作。    图 82 二叉树  分析fun7代码，不难看出，fun7为递归函数。    图 83 分析fun7代码  将以上汇编代码翻译成c语言，不难看出，fun7实际便是根据输入的值按照不同的路径向下递归二叉树，不同路径递归二叉树的返回值是不一样的。   |  | | --- | | 1. int func7(int\* node){ 2. if(node == nullptr) return -1; 3. int edx = node -> val; 4. if(edx > esi){ 5. return 2\*func7(node -> ls); 6. } 7. else if(edx == esi){ 8. return 0; 9. } 10. else{ 11. int tmp = func(node -> rs); 12. return 1 + 2\*tmp; 13. } 14. } |   由于在secret\_阶段中返回值为3才能通关，因此需要寻找能使fun7返回值为3的数。  因此，写一个模拟递归过程的代码，通过循环尝试0到1001之间的值（根据以上分析，由于二叉树中最小节点值为1，因此输入所有小于1的均会返回相同的结果，输入值最小到0即可；而最大值1001则是遵循之前的限制），来找到满足答案的数。   1. #include<iostream> 2. using namespace std; 3. int tree[20] = {0,36,8,50,6,22,45,107,1,7,20,35,40,47,99,1001}; 4. int func7(int esi, int node){ 5. if(node > 15) return -1; 6. int edx = tree[node]; 7. if(edx > esi){ 8. return 2\*func7(esi, 2\*node); 9. } 10. else if(edx == esi){ 11. return 0; 12. } 13. else{ 14. return 1 + 2\*func7(esi, 2\*node + 1); 15. } 16. } 17. signed main() { 18. ios::sync\_with\_stdio(0); 19. cin.tie(0); 20. cout.tie(0); 22. for(int i = 0; i <= 1001; i++){ 23. int ans = func7(i, 1); 24. if(ans == 3) cout << i << endl; 25. } 27. return 0;   图 84 破解secret\_阶段代码  运行程序，最终能得到99和107可以满足fun7的返回值为3。    图 85 secret\_阶段答案  4）输入“99”或“107”，验证以上分析的正确性。    图 86 调试secret\_阶段代码（5）    图 87 调试secret\_阶段代码（6）  成功通关。  综上，secret\_阶段对一个二叉树进行递归，要求递归后的返回值满足题目要求，而输入的值会影响这个返回值的大小。 |
| 1. **实验结论：**   由下图可以看到，全部关卡均通关。    图 88 通关截图 |
| 1. **心得体会：**   总结：   1. 为了深入分析代码的作用，必须灵活运用gdb来监控程序运行时寄存器和相关内存的变化。仅凭阅读汇编代码，很难准确地推断出代码的功能。 2. 掌握汇编语言是顺利通过所有关卡的关键。 3. 耐心和兴趣是至关重要的。在这个实验中，我花了整整四天时间才通关，尤其是第六关和隐藏关卡，破解起来异常艰难，耗时极多。如果缺乏耐心，可能在第五关就已放弃。然而，正是兴趣支撑我坚持到最后。这种类型的通关实验极大地激发了我的挑战欲望。尽管困难重重，但每通过一关或解开一个难题，那种成就感便油然而生，激励我继续前进。因此，我衷心感谢这次实验，它让我收获颇丰。 |

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  2025年 4月27 日 |
| 备注： |