Sichuan University

Laboratory Report

Student ID： 2023141520228 Name： 陈祁轩

Major： Computer Science Date: 2024.11.30

|  |  |  |  |
| --- | --- | --- | --- |
| Course Name | Introduction to Systems Software | Credit Hours | 3 |
| Experiment Name | Data lab & Bomb lab | Time | 2024 Fall |
| Purpose of the experiment | 1. Having a deep understanding of bit operations and logic operations. 2. Having a deep knowledge of assembly languages. | | |
| Experimental  Environment | Windows 11; Online Platform: www.educoder.net. | | |

|  |  |
| --- | --- |
| Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results  Experimental Procedures and Results | **Lab1: data lab**  1. **bitAnd** function  (1) Experimental Procedure:  Firstly, we need to understand the requirements of the implementation of the function. We can only use ~ | to implement & operation. Then, we can consider using De Morgan's law to change the & operation into a combined expression of ~|.  (2) Experimental Results:    2. **getByte** function  (1) Experimental Procedure:  Firstly, we need to understand the requirements of the implementation of the function. To obtain the nth byte in the data x, you can consider moving the byte to the position of the 0th byte through the >> operation, and then calculate it by taking the remainder.    (2) Experimental Results:    3. **logicalShift** function  (1) Experimental Procedure:  Firstly, we need to understand the requirements of the implementation of the function. We need to logically shift the int type data x to the right by n bits. Hence, we can use >> operation to implement the function.    (2) Experimental Results:    4. **bitCount** function  (1) Experimental Procedure:  First, we need to know that the goal is to calculate the number of 1s in the binary string of int type data x, and return an int type result. Then, we can use & 1 operation for every bit to implement the algorithm.    (2) Experimental Results:    5. **Bang** function  (1) Experimental Procedure:  We need to use other operators like ~ & ^ | + << >> to implement the ! logic operation. Based on the definition of the ! operation: if x is 0, then !x will be 1; if x is not 0, then !x will be 0, we can get the algorithm below for implementation.    (2) Experimental Results:    6. **Tmin** function  (1) Experimental Procedure:  We need to return the least two's complement integer. According to the definition of the two’s complement number, we know that the MSB should be 1 and all others should be 0. Then, we can get the result using << operator.    (2) Experimental Results:    7. **fitsBits** function  (1) Experimental Procedure:  In this function, we need to know whether an int type data x can be represented as an n-bit two's complement integer. The range of an n-bit two's complement integer is from -2^(n-1) to 2^(n-1) – 1. Then, we can use shift operations to check whether x is in this range.    (2) Experimental Results:    8. **divpwr2** function  (1) Experimental Procedure:  In this function, for 0 < = n < = 30, calculate x/(2n), rounding to zero. For positive numbers, right shift directly is fine, but we need to ensure that for negative numbers, the result is also rounded toward zero. Therefore, we design such an algorithm below:    (2) Experimental Results:    9. **negate** function  (1) Experimental Procedure:  This function needs to return the opposite number of int type x, and we know that the opposite of x is (~x) + 1. Then we get the algorithm below:    (2) Experimental Results:    10、**isPositive** function  (1) Experimental Procedure:  For int type x, if x > 0, 1 is returned, otherwise 0 is returned. Then we can design algorithm based on the sign bit.    (2) Experimental Results:    11、**isLessOrEqual** function  (1) Experimental Procedure:  For int type x and y, if x <= y, 1 is returned, otherwise 0 is returned. Then we can design algorithm based on the sign bit.    (2) Experimental Results:    **1**2. **ilog2** function  (1) Experimental Procedure:  The function returns log2x. The key of this algorithm is to find the highest bit of the integer.    (2) Experimental Results:    13. **float\_neg** function  (1) Experimental Procedure:  We first need to know that the function returns the expression of the floating-point parameter f -the bit equivalent of f. Both the argument and the result are passed as unsigned ints, but they will be interpreted as bit-level representations of single-precision floating-point values. If the parameter is NaN, the parameter is returned. Based on the understanding, we can design our algorithm.    (2) Experimental Results:    14. **float\_i2f** function  (1) Experimental Procedure:  We first need to know the function returns the equivalent value of the expression (floating-point number) x. The result is returned as an unsigned int, but is interpreted as a bit-level representation of a single-precision floating-point value. Based on the understanding, we can design our algorithm.    (2) Experimental Results:    15. **float\_twice** function  (1) Experimental Procedure:  We need to understand the function returns the bit equivalent of the expression 2 \* f of the floating-point parameter f. Both the argument and the result are passed as unsigned ints, but they will be interpreted as bit-level representations. Single-precision floating-point values. When the parameter is NaN, the return parameter. Based on the understanding, we can design our algorithm.    (2) Experimental Results:    **Lab2: bomb lab**  **Phase 1**   1. Experimental procedure:    * + 1. Looking at the bomb.c source logic:     The input is our input, and the phase\_1 function takes this input as a parameter, and if the input is correct, the bomb is defused. Function phase\_1 is the number one bomb.   * + - 1. Looking at the assembly code for bomb.s, the assembly code for main corresponds to the source logic above.     The read\_line function, as the name suggests, is to place the return result in the %rax register (which is commonly used by %rax Save the returned result); Send the %eas content to the %rdi register (the %rdi register is usually used to hold the tune-up with the arguments of the function); Call the phase\_1 function, then call the phase\_defused function.   * + - 1. Look for the definition of the phase\_1 function in the assembly code     You can see that the explode\_bomb function is called in it, and you can see that the phase\_1 function is a bomb; Called before the explosion strings\_not\_equal function and use the test directive to phase and sum the two operands if the result is 0 ZF is used to identify position 1, otherwise it is set to 0; je command checks the ZF flag position, if it is 1, it jumps to the specified position, i.e. skip the explosion; It seems that the strings\_not\_equal function compares the two strings and returns 0 if they are the same, no returns 1, and one of the two parameter addresses is already in %rdi, which is our input, and the other is stored in %ESI, its value is the virtual memory address 0x402400.   * + - 1. Open the executable bomb in gdb-mode in emacs and enter to output this stringx/s 0x402400     (2)Experimental result:    **Phase 2**  (1)Experimental procedure:  The logical structure of the bomb.c source code remains unchanged, and phase\_2 is the second bomb.  The part of the assembly code that calls the phase\_2 function is the same as the No. 1 bomb except for the function name.  Let's take a look at the phase\_2 function definition    In this case, the input to the read\_line function is stored in the %rdi register, and the contents of the %rsi register are saved The %rsp register is the content of the stack pointer register; There are two places in the phase\_2 definition that call the explode\_bomb function, and the instructions for the conditional transfer are    Apparently the call to read\_six\_numbers changes the data in the caller's stack. If you look at some of the definitions of read\_six\_numbers, it even calls the explode\_bomb function.    4. read\_six\_numbers Functional analysis   * The first half will locate the addresses of the parameters passed in by its caller function, from left to right, respectively %rsp %rsp+4%rsp+8%rsp+12%rsp+16%rsp+20 * And then there's the part of the 3rd code example above, where 0x4025c3 holds the string pointer, and its contents are "%d %d %d %d %d %d %d", it can be inferred that the input data will be parsed with this formatted string; * The register %eax immediately places the number 0, which will be checked after the \_\_isoc99\_sscanf@plt is called The value of the memory, if it is parsed less than 6 times, will explode the bomb, so the data string we enter should be It is 6 numbers separated by spaces (in fact, it doesn't matter if there are more than 6, only the first 6 will be taken in); * According to the contents of the assembly code file, the main working logic of the \_\_isoc99\_sscanf@plt function cannot be derived, Because it calls the glibc library function, but it can be inferred that its main job is to scan the input string, and Parse it with the formatted string in the previous one, turning the string digits inside into numbers one by one.   + - 1. phase\_2 Functional analysis * In the second code example above, after the read\_six\_numbers is called, the cmpl command will be used Check the size of the value stored in the bottom of the stack variable, which is generated and stored by read\_six\_numbers; as If this value is equal to 1, it will carry out subsequent logical operation, if it is not equal to, it will explode; So you can know what is entered The string must have the number 1 in it; * Jumping to the 400F30 address, you can see that this is followed by a loop, and the loop condition is the %rbx register is not equal to the value of the %rbp register; First, store the value of %rsp+4 in %rbx, and then increment by 4; Compare the values stored at the %rsp and %rsp+4 virtual addresses, and the latter must be twice as large as the former Check up to %rsp+18 address; If the conditions are not met, the bomb is detonated.   + - 1. Conclusion: From the above, it can be inferred that the bomb needs to be entered: 1 2 4 8 16 32  1. Experimental result:     **Phase 3**   1. Experimental procedure:   In front of the assembly code of phase\_3, the \_\_isoc99\_sscanf@plt function called in bomb No. 1 is called. It will scan the number representation in the input string and assign the number to the parameter;    As can be seen from the CMP and JG parts of the code, there should be more than two digits in the string to be entered this time;  In the second line of the mov instruction, a memory address is passed, and the x/s command knows that its value is "%d %d", so this time The string required for bomb disposal should contain two numbers;  The two digits obtained from the scan are stored in 0x8 (%rsp) 0xc (%rsp);  The first input can be extrapolated from the last CMPL instruction followed by the JA instruction (jumping to bomb explosion). The number cannot be greater than 7.  If the input string is accepted, an indirect jump of the base address address scale factor offset will be performed:    The address to which you are redirected is (%rax \* 8) + 0x402470 the address stored in this address, and %rax is stored in the first digit entered;  If %rax is 0; You will be redirected to the address stored in the 0x402470 address; By x/w yes It turns out that this redirect address is 0x400f7c;  For simplicity's sake, the first number we enter is indeed 0, then we'll jump to:    The MOV command sends the immediate 0xcf into the %eax register, which is converted to 207 in decimal decimal, and then jumps to the virtual address 0x400fbe;    At the jump address, run the CMP command and compare the values stored by %EAX and 0xc (%rsp), and the following values will be is the second number we enter, and this number should be equal to 207, otherwise detonate the bomb;  So the result should be "0 207", of course, there are 7 combinations in theory, so I won't calculate them one by one.   1. Experimental result:     **Phase 4**   1. Experimental procedure: 2. Again, this time the scan function was called, and this time the string that needs to be entered still needs two digits (this time it must must be two, no more and no less), separated by spaces; 3. Compare and jump code in the following paragraph:     It means that the first value obtained by scanning should not be greater than the decimal number 14, otherwise the bomb will detonate;   1. Prepare the parameters (%edx: 14, %esi: 0, %edi: first input number) and call the fun4 function, call Then check the return value of func4 and detonate the bomb if the lowest bit is 1, and there are two key jump structures in func4:     The value in %edx has been converted and stored in %ecx before the first key jump, and the value is 7 if 7 is small So equal to the first input number, it jumps, and the place where the jump is first stored in %eax, and then it is carried out the second jump logic;  Doing the same comparison again, if this time the result is greater than or equal to, the jump is made, and the position of the jump is the end of the function Stage;  If it is both greater than or equal to or less than or equal to, it can be seen that it is required to be equal to, so the first number entered is 7 under the condition that it jumps twice The return value of the function is 0;  The func4 internal logic is a recursive structure, assuming that 7 is less than or equal to the first incoming number, but not by much If it is equal to it, then it will enter recursion, and in order for this recursion to be terminated, then the conditions of two subsequent jumps must be met. When the next level is recursively returned, its caller's operation on %eax will inevitably put the last position of the value in it to 1, integer After func4 is over, %eax will be 1, and subsequent jumps based on this will detonate the bomb, so the func4 function Each time the second jump condition is encountered during the call, the jump must be realized, that is, the first number passed in must be certain is less than or equal to 7;  According to the previous jump condition, if the processing of the median value of %ecx before and after it fails, the first number of inputs can be deduced The word must be one of 0 1 3 7, which is a sequence).   1. If the return value of 0 after calling func4 in phase\_4 is 0, it will not jump to the location of the detonating bomb; correct After comparing whether the second input parameter and 0 are equal, if they are equal, they jump to the end stage of the function, indicating the requirement of the first The two input digits must be 0; 2. Then we get the result is "7 0" (the first number can be any of 0 1 3 7). 3. Experimental result:     **Phase 5**   1. Experimental procedure: 2. Note that phase\_5 explicitly sends the address of the input string stored in %rdi to the %rbx register      1. mov %fs:0x28,%rax %fs This segment register points to the TEB structure (thread structure), table, of the currently active thread The data on the offset address 0x28 on the fs segment is transferred to the %rax register (it has nothing to do with the puzzle solution) 2. First Jump:   Zeroing out the contents of %eax;  Call the string\_length function to calculate the length of the input string;  Compare the return value of string\_length with the immediate number of 6, and jump if it is equal, otherwise detonate the bomb;  So the input string should be 6 characters long.   1. If the string length is eligible, jump to 0x4010d2, set the value of %eax to 0, and then jump to 0x40108b.   Execute the mobzbl instruction here (extending a source operand 1 byte below the value 0 to 32 bits and depositing it at the target register), in this case the first character in the string; This character is then stored in %cl value, %cl is the low 8-bit register of %ecx) to the top of the stack.     1. The characters obtained in step 4 are processed, and the last character (which is actually the lower four-digit binary representation of the ASCII code of the character) is stored %edx's low 8-bit register %dl, all but the bottom 8 bits are 0 in the %edx register, and the following:     Transfer the value stored at the %rdx + 0x4024b0 address to the %edx register;  Transfer the bottom 8 bits of data at the %edx register to the +10 address at the top of the stack,  Add the value stored in %rax by 1;  Comparing the value stored in %rax, if it is not equal to 0, it jumps to 0x40108b, which is actually a new loop of steps 4 and 5;   1. After the loop is complete, pass in the address stored at %rsp + 10 and 0x40245e as parameters strings\_not\_equal, if the return value is 1, it means that it is not equal, and if the bomb is detonated, the return value is 0 By the end of the phase\_5 function, the bomb is removed. 2. x/s 0x40245e to get the string "flyers" that starts at that address.   x/s 0x4024b0 gets the string "maduiersnfotvbylSo you think you" at the address can stop the bomb with ctrl-c, do you?”  The input string should contain 6 numbers, each number corresponds to 1 index of the second string, and the index starts at 0, 6 characters with opposite indexes can be pieced together to form "flyers"   1. It can be extrapolated that the combination of index is 9fe567, and looking up the ASCII table, the final result is "I O N E F G". 2. Experimental result:     **Phase 6**   1. phase\_6 first calls the read\_six\_numbers function, which parses the input string out of 6 numbers, the detailed logic of which was analyzed in Bullet No. 2; 2. Then the value at %rsp is judged, which is the first digit of the input string, which must be unsigned. and must be less than or equal to 6; 3. Then enter a cycle to compare whether the first number and the subsequent number are equal, the second number and the subsequent number ...... in turn, In the event of an equal, the bomb is detonated, and all numbers must be less than or equal to 6 and cannot be 0;      1. This is followed by another cycle that subtracts the entered number from 7 and stores the result at the original address:      1. Continuing into a cycle      * This loop in turn compares the values stored at addresses such as %rsp %rsp+4 obtained from the previous steps with 1, and if they are equal, then store the 0x6032d0 address at %rsp + %rsi\*2 + 0x20 address, the number stored in %rsi The word is used to determine whether to terminate the cycle, and 4 is added sequentially from 0, which is equal to the termination of the cycle at 24; * If it is not equal, then compare it with 2 and add 8 to 0x6032d0, and this is another cycle: if it is not equal, it is incremented, until it is equal, at which point the incremented value of the obtained 0x6032d0 is placed at the corresponding address;  1. It's another cycle     The logic looks complicated, but it can be understood by grasping one key point: the values obtained in step 5 are always in order Place the next value in the address obtained by the previous value +8; In fact, it is creating a linked list.   1. Last loop:     Loops alternately compare the sizes in the linked list elements, with the preceding value being greater than or equal to the following value.   1. According to the previous deduction, it can be seen that the 6 digits in the input string can only be permutations and combinations of 123456; Step 5: Middle-raised and 332 in 0x6032d0, 0x6032d8 address, and 168 in 0x6032e0, This is followed by 924, 691, 477, 443, and the conclusion from step 6 is that the previous element of the linked list must be connected In fact, the big one is to use the data we input as an index to create a new descending linked list according to the original linked list. that The result should be "4 3 2 1 6 5"   (2) Experimental result:    **Hidden Bomb**   1. And that's not all! Although it is said that the results obtained above are entered in turn, the bomb program will prompt the bomb to complete the normal exit, but There is also a hidden bomb secret\_phase in the bomb.s assembly code file, which is defused in the sixth bomb The demolition will only be accepted when it is entered after the secret\_phase function is successfully defused phase\_defused function (possibly):      * As you can see, \_\_isoc99\_sscanf@plt accepts a formatted string and an input string, and that An input string is made up of two numbers and a string, where is this string? Note that it doesn't It is to re-enter a line after the sixth string is entered, but one line from the original six-line string; * Use gdb to set a breakpoint on the executable, and the breakpoint is set at the entrance of the phase\_defused function, Perform the correct answer - input - output, to the sixth string input, inside the phase\_defused function Step to the 0x4015f5 address, and X/s 0x603870 get "7 0"; It can be seen that here is the analysis of the 4th input strings; * To defuse a hidden bomb, the 4th string should be "7 0 DrEvil"  1. secret\_phase Function Logic:      1. fun7 function logic:     The return value of fun7 analyzed in step 2 must be 2, so there is only one possibility for fun7 to be recursively invoked  Enter the fun7 function with an initial return value of 0  Enter 401217 recursion, which doubles the return value when it ends;  Recursion at 40122d, after the recursion ends, the return value will be doubled and increased by 1, that is, %eax set the value to 1;  In this call, %eax is kept at 0, both jump conditions are satisfied, and recursion is not entered.   1. gdb x/2w queries the value at the 0x6030f0 address and the one that will be used by the predicted function call trace The value at the delta address determines that the input number must be 22 for the trajectory in step 3 to be realized.   (2)Experimental result: |
| Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis  Experiment Result Analysis | **Lab1: data lab**  1. **bitAnd** function  (1) Result explanation:  In this function, we implement Bitwise AND without using the & operator.  (2) Result reasoning:  Given that only the bitwise NOT (~) and bitwise OR (|) operations are available, yet the desired operation to be implemented is the bitwise AND (&), one can leverage De Morgan's Laws to transform the & operation into an equivalent expression composed of ~ and |. Specifically, De Morgan's Laws provide the following equivalence for bitwise operations:  **A & B = ~(~A| ~ B)**  This transformation allows us to express the bitwise AND operation using only the available bitwise NOT and OR operations. By applying this law, we can effectively implement the desired functionality within the constraints provided.  2. **getByte** function  (1) Result explanation:  In this function, we can get any specific byte in an integer.  (2) Result reasoning:  To extract the n-th byte from an integer x*x*, one can employ bitwise shift and masking operations. Specifically, the desired byte can be moved to the least significant byte (0th byte) position through a right shift operation, followed by applying a bitwise AND with the mask 256. This process isolates the n-th byte while ensuring that all higher-order bytes are set to zero.  3. **logicalShift** function  (1) Result explanation:  In this function, we can logically left shift an integer by n bits.  (2) Result reasoning:  To implement a logical right shift of an integer x*x* by n bits, one can first perform an arithmetic right shift, followed by a bitwise AND operation with a mask that has the highest n bits set to 0 and all other bits set to 1. This ensures that the highest n bits are cleared to 0, while the remaining bits remain unchanged.  4. **bitCount** function  (1) Result explanation:  In this function, we can count the number of 1s in an integer.  (2) Result reasoning:  To count the number of 1s in the binary representation of an integer x, one can employ a parallel bit-counting algorithm that progressively reduces the problem size by summing adjacent bits. This method divides the 32-bit integer into smaller groups, summing the bits within each group, and then combining the results iteratively until a single sum is obtained.  5. **Bang** function  (1) Result explanation:  In this function, we implement the ! operator without using it.  (2) Result reasoning:  To determine whether an integer x is non-zero, one can leverage the properties of bitwise operations and the two's complement representation. Specifically, by performing a bitwise OR operation between x*x* and its negation −x, the highest bit (sign bit) of the result will be 1 if x is non-zero, and 0 if x is zero. This property can be used to construct a compact and efficient expression to compute the logical NOT of x*.*  6. **Tmin** function  (1) Result explanation:  In this function, we get the minimum two's complement integer.  (2) Result reasoning:  In the two's complement representation, the minimum integer value for a 32-bit signed integer is −2^31, which is represented by the binary pattern where the highest bit (sign bit) is 1 and all other bits are 0. This value can be directly obtained by shifting the integer 1 left by 31 positions.  7. **fitsBits** function  (1) Result explanation:  In this function, we detect whether an integer can be represented as an n-bit two's complement integer.  (2) Result reasoning:  The maximum integer that can be represented by the n-bit binary is the highest bit is 0, the other bits are 1, and the smallest number that can be represented is the highest bit is 1, and the other bits are 0. Therefore, we can shift x to the right by n-1 bits, and then compare the shifted result with the sign bit, if the two are the same, then x can be represented as n is a binary number.  8. **divpwr2** function  (1) Result explanation:  In this function, we calculate x/(2n) and round the result to 0.  (2) Result reasoning:  When performing division operations using bitwise shifts, non-negative numbers are rounded towards zero by default when right-shifted. For negative numbers, however, right-shifting rounds towards negative infinity, which may not be the desired behavior. To ensure that negative numbers are also rounded towards zero, an offset of 2n−1 (half of the divisor) should be added to the number before performing the shift. This adjustment ensures consistent rounding behavior for both positive and negative integers. Mathematically, for a non-negative integer x, the result of dividing by 2n is x≫n, while for a negative integer x, the result is (x+2n−1)≫n, where >> denotes a signed right shift.  9. **negate** function  (1) Result explanation:  In this function, we implement the opposite number of an integer without using – operator.  (2) Result reasoning:  Based on the equation -x = (~x) + 1, we find the implementation easily.  10、**isPositive** function  (1) Result explanation:  In this function, we can justify whether an integer is positive.  (2) Result reasoning:  To determine whether an integer x is positive, we can use bitwise operations to compute the sign bit and the logical NOT of x. According to the table, the **isPositive** function returns 1 if and only if both the sign bit (signal) and the logical NOT of x (!x) are 0. This can be achieved by performing a bitwise OR between the sign bit and the logical NOT of x, and then applying the logical NOT to the result.  11、**isLessOrEqual** function  (1) Result explanation:  In this function, we can compare 2 integers using bit operations.  (2) Result reasoning:  To determine whether x≤y, consider two cases based on the signs of x and y: If x and y have different signs, x≤y is true if and only if x is negative, as any negative number is less than any positive number in a two's complement system. If xand y have the same sign, x≤y can be determined by checking if their difference x−y is non-positive (i.e., less than or equal to zero). This approach ensures that the comparison correctly handles both positive and negative values, leveraging the properties of two's complement arithmetic.  **1**2. **ilog2** function  (1) Result explanation:  In this function, we can calculate log2x, x is an integer.  (2) Result reasoning:  The function is to find the closest number n such that 2nis closest to x, and satisfies 2n <= x. If the value is 1, it means that there is at least one 1 in the upper 16 bits of x, then 16 should be added to the result. If the resulting value is 0, then there is no 1 in the upper 16 bits. Then move the X to the right (result+8) bit, and perform the same two negation operations, if the obtained value is 1, it means that there is at least one 1 in the 8 bits (result+8) and (result+15), then the result should be added with 8; If the resulting value is 0, then there is no 1 in the 8 bits. And so on, continue to move the X to the right (result+4), (result+2), (result+1) bits, and do the same to get the result.  13. **float\_neg** function  (1) Result explanation:  In this function, we get the expression of the floating-point parameter f -the bit equivalent of f.  (2) Result reasoning:  The parameter of the function may be NaN (Not a Number), so it needs to be judged. If the 23rd to 30th bits of the parameter uf are all 1, and the lower 23 bits of uf are not 0, this means that uf is interpreted as a bit-level representation of a single-precision floating-point value as a NaN, so it should be returned directly, otherwise the highest bit (sign bit) of uf should be directly negated to obtain -f.  14. **float\_i2f** function  (1) Result explanation:  In this function, we get the equivalent value of the expression (floating-point number) x.  (2) Result reasoning:  This function mainly investigates the process of converting int-type data to float-type data, so you can judge and execute it step by step. First of all, the x is judged, if it is equal to 0, it can be returned directly, if it is less than 0, it enters the loop, and finds the 1 of the highest bit of x except the sign bit. After finding 1, you can get the value of the left shift, and then you can get the order value, the mantissa tail that will be rounded, and the mantissa that will be obtained. After that, the symbol bit, the order code, and the mantissa are combined or the result is obtained. Of course, the mantissa tail must be judged in the end, if it is greater than 0x80, then the result is added to 1; If it is equal to 0x80, it should be rounded to the couple. Finally, the result can be returned.  15. **float\_twice** function  (1) Result explanation:  In this function, we get the bit equivalent of the expression 2 \* f of the floating-point parameter f.  (2) Result reasoning:  First of all, the uf is judged, if the uf is positive 0 or negative 0, then the uf is directly returned, because they are twice as much as itself. Then, if the order value is equal to 0xff, then uf is infinity or NaN, then uf should be returned directly. If the order value is 0, the uf is a non-normalized value, and the mantissa should be multiplied by 2 first. Then determine whether there is a carry after the mantissa change, if so, the order value should be increased by 1, and the value of the mantissa should be updated; If the order value is not 0, you can simply add 1 to it. Finally, the sign bit, order code, and mantissa are combined to get 2\*uf, and finally the value can be returned.  **Lab2: bomb lab**  **Phase 1**  Result Analysis:  In summary, it can be judged that this function needs to input a string, and the input string is required to be equal to a certain string that has been given in the question. The 0x804a1c4 is an immediate number, which can be judged to be the first address of the given string in the question, and the other argument is the string we input stored in the 0x20 (%esp). So we only need to get the memory string by 0x804a1c4 this address, and that string is the password of the phase\_1.  **Phase 2**  Result Analysis:   * <phase\_1+25> - <phase\_1+32>: Comparing the value of 0x18 (%esp) in memory to 1, i.e. comparing the first parameter to 1, if the two are equal it will not explode, here the first number must be 1 * <phase\_1+45> - <phase\_1+64>: This code is equivalent to a do-while loop, because the previous code limits the first number to 1, and the following code limits the remaining 5 numbers. First get the value of the first number (-0x4(%ebx)), then store 2 times it in the %eax register, then compare the value of (%ebx) in memory (that is, the second number entered) with %eax, only when the two values are equal will not explode, then add %ebx to 4 (that is, the address of the next number), and finally determine if %ebx and %esi are equal (because the address of 6 numbers is from 0x18(%esp) to 0x2c(%esp), So here is equivalent to judging whether there are already 6 numbers in the loop judgment), if they are equal, they will jump out of the loop, otherwise they will enter the loop again.   To sum up, the 6 numbers entered should meet the requirements of the latter number is 2 times the previous number, and because the first number can only be 1, the phase\_2 passcode is: 1 2 4 8 16.  **Phase 3**  Result Analysis:   * <phase\_3+19>: Storing the 0x804a3cb in the 0x4 (%esp) and using the x/s 0x804a3cb command to see "%d %d", so it comes to the conclusion that two integer arguments are to be entered * <phase\_3+34> - <phase\_3+42>: Call the < \_\_isoc99\_sscanf@plt> input function, store the number of input parameters in %eax, and compare %eax with 1, if %eax is greater than 1, it will not explode, that is, the number of input parameters must be greater than 1 * <phase\_3+49> - <phase\_3+54>: Compares the value of 0x18 (%esp) in memory (i.e. the first parameter) to 7, and it does not explode when it is less than or equal to 7 * <phase\_3+67> - <phase\_3+156>: This code is a jump table that jumps to the corresponding statement and performs the corresponding operation according to the address obtained in the previous step, and stores the final result in the %eax register * <phase\_3+168> - <phase\_3+173>: Comparing the first parameter to 5, it does not explode when it is less than or equal to 5, and here it is concluded that the first number must be less than or equal to 5 * <phase\_3+175> - <phase\_3+179>: Compares %eax (the result of the operation in the jump table) with the value of the 0x1c (%esp) in memory, if the two values are equal it will not explode, here the second parameter is required to be equal to the result of the calculation.   To sum up, to input two integer parameters, the first parameter must be less than or equal to 5, and the second parameter is equal to the operation result in the jump table.  **Phase 4**  Result Analysis:  The value in %edx has been converted and stored in %ecx before the first key jump, and the value is 7 if 7 is small So equal to the first input number, it jumps, and the place where the jump is first stored in %eax, and then it is carried out the second jump logic;  Doing the same comparison again, if this time the result is greater than or equal to, the jump is made, and the position of the jump is the end of the function Stage;  If it is both greater than or equal to or less than or equal to, it can be seen that it is required to be equal to, so the first number entered is 7 under the condition that it jumps twice The return value of the function is 0;  The func4 internal logic is a recursive structure, assuming that 7 is less than or equal to the first incoming number, but not by much If it is equal to it, then it will enter recursion, and in order for this recursion to be terminated, then the conditions of two subsequent jumps must be met. When the next level is recursively returned, its caller's operation on %eax will inevitably put the last position of the value in it to 1, integer After func4 is over, %eax will be 1, and subsequent jumps based on this will detonate the bomb, so the func4 function Each time the second jump condition is encountered during the call, the jump must be realized, that is, the first number passed in must be certain is less than or equal to 7;  According to the previous jump condition, if the processing of the median value of %ecx before and after it fails, the first number of inputs can be deduced The word must be one of 0 1 3 7, which is a sequence.  **Phase 5**  Result Analysis:  In the phase\_5 function, the program first transfers the address of the input string from the %rdi register to the %rbx register. It then calls the string\_length function to calculate the length of the input string and checks if this length equals 6. If the length is not 6, the program triggers the bomb by calling the explode\_bomb function. If the length is correct, the program processes each character of the input string in a loop. For each character, it extracts the lower 8 bits, uses this value as an index into a second string located at address 0x4024b0, and stores the corresponding character from this second string into a buffer on the stack. After processing all 6 characters, it compares the resulting string with the string "flyers" located at address 0x40245e. If the strings do not match, the bomb is triggered. The input string should contain 6 numbers, each corresponding to an index in the second string, such that the characters at these indices form the string "flyers". The final solution can be inferred as the combination of indices "9fe567", which translates to the string "IONEFG" based on the ASCII table.  **Phase 6**  Result Analysis:  In phase\_6, the program first calls the read\_six\_numbers function to parse an input string into six numbers. Each number must be unsigned, less than or equal to 6, and non-zero. The program then enters a loop to compare each number with all subsequent numbers in the input. If any two numbers are equal, the bomb is triggered. Additionally, each number must be unique and within the range of 1 to 6.  After ensuring all numbers are distinct and valid, the program performs another cycle where it subtracts each entered number from 7 and stores the result back at the original address. This effectively transforms the input numbers into their complements relative to 7.  Next, the program creates a linked list using the transformed values. It places each value at an address determined by the previous value plus 8 bytes, effectively linking the elements together. The program then iterates through this linked list, comparing each element with the next one. The preceding value in the linked list must be greater than or equal to the following value. If this condition is violated, the bomb is triggered.  Based on the logic, the six digits in the input string can only be permutations of the numbers 1 through 6. The final solution requires creating a descending linked list using the input numbers as indices. The correct input that satisfies all conditions is "4 3 2 1 6 5".  **Hidden Bomb**  Result Analysis:  This hidden phase is triggered only if exactly six input strings have been entered, with the fourth string specifically required to be "7 0 DrEvil". The program validates that the input consists of two numbers and a string matching the format, and then calls the secret\_phase function. Within secret\_phase, it reads a seventh input, converts it to a number using strtol, and ensures this number does not exceed 1001. It then calls the fun7 function, which must return a value of 2 for the bomb to remain defused. Through analysis, it is determined that the input number for fun7 must be 22 to satisfy this condition. Thus, the complete solution involves entering "4 3 2 1 6 5" for the sixth phase, followed by "7 0 DrEvil" as the fourth string, and finally inputting "22" to successfully defuse the hidden secret\_phase. |