CSC 411: Computer Organization

Binary Bomb Lab

Due: 14th April 2025 Sofia Mancini **Bomb#** Bomb80 Time Taken: ~5 hours

Bomb Disassembly

Phase 1

The Phase 1 assembly code was pretty straightforward, the 'lea' line and the subsequent call to 'strings_not_equal' made it clear that this bomb would be diffused by typing in a string that matched the expected output. Since lea gave us the memory address, by using the x command from gdb, I was able to print the exact string the phase was expecting.

```
0x000055555555664f <+8>:     lea    0x1c52(%rip),%rsi        # 0x5555555582a8 // expected st
0x0000555555556656 <+15>:    call   0x555555556bf0 <strings_not_equal> // call to function

(gdb) x/s 0x5555555582a8 // examine memory address as a string
```

Phase 1 Solution: I am not part of the problem. I am a Republican.

Phase 1 C Code:

```
void phase_1(const char *input) {
    const char *expect = "I am not part of the problem. I am a Republican.";

    if (strings_not_equal(input, expect) != 0) {
        explode_bomb();
    }
}
```

Phase 2

The next phase uses a function called 'read_six_numbers' so immediately I knew the style of the input. Next there is a comparison for the hex representation of the number 1 to the first input value, meaning the first number will need to be '1'. Then the code enters a loop where it doubles the first number and compares it to the expected input for the six expected values.

```
0x0000555555556688 <+29>:    call   0x555555556f30 <read_six_numbers> // call to function
0x000055555555568d <+34>:    cmpl   $0x1,(%rsp) // check that first value is '1'
0x5555555566a4 <+57>:    add    $0x4,%rbx // advance pointer to next value
0x5555555566af <+68>:    add    %eax,%eax // double previous value
0x5555555566b1 <+70>:    cmp    %eax,0x4(%rbx) // compare doubled value to next value
```

Phase 2 Solution: 1 2 4 8 16 32

Phase 2 C Code:

```
void phase_2(const char *input) {
    int num[6];
    read_six_numers(input, num);

    if (num[0] != 1) {
        explode_bomb();
    }

    for (int i = 1; i < 6; i++) {
        if (num[i] != 2 * num[i - 1]) {
            explode_bomb();
        }
    }
}
```

Phase 3

This phase uses a built-in C function called sscanf, which campres the number of values entered to a specific expected number, in this case the solution will have three input values, a char and two ints. Next it checks that the first selector value is less than 7, so any integer value 0-7. The code then moves into a switch block for values entered i <= 7, I only attempted with case 0 since the expected values to compare were easy to see in the assembly code. From the hex representations I knew the three values to be weere 'r' '0' and '767'.

```
0x00005555555566f1 <+24>:    lea    0xf(%rsp),%rcx // char value
0x00005555555566f6 <+29>:    lea    0x10(%rsp),%rdx // int value
0x00005555555566fb <+34>:    lea    0x14(%rsp),%r8 // int value
0x0000555555556700 <+39>:    lea    0x1981(%rip),%rsi        # 0x555555558088 // formatting
0x0000555555556707 <+46>:    call   0x555555556340 <__isoc99_sscanf@plt> // function call
0x000055555555670c <+51>:    cmp    $0x2,%eax // check input count

0x0000555555556711 <+56>:    cmpl   $0x7,0x10(%rsp) // check if selector value is 0 <= i <=

0x0000555555556738 <+95>:    mov    $0x72,%eax // case 0: char value 'r'
0x000055555555673d <+100>:   cmpl   $0x2ff,0x14(%rsp) // case 0: second int value '767'
```

Phase 3 Solution: 0 r 767

Phase 3 C Code:

```
void phase_3(const char *input) {
    int selector;
    char char;
    int num;
```

```
    int match;

    match = sscanf(input, "%d %c %d", &selector, &char, &num);

    if (match != 3) {
        explode_bomb();
    }

    if (selector > 7) {
        explode_bomb();
    }

    switch (selector) {
        case 0:
        if (char != 'r' || num != 767) {
            explode_bomb();
        }
        break;
        // Other cases not tested
        default:
            explode_bomb();
    }
}
```

Phase 4

This phase uses the same sscanf function as phase 3, so immediately I identified the solution would be two integer values. The code then takes the first input value, subtracts 2, and compares that value to 2. Therefore '4' had to be the value since 4-2=2. From there it calls 'func4' with the inputs func4(9,a) where a is the first input value. After examing the assembly code I knew func4 was recursive since it contained several calls to itself. The first function call works out to func4(8,4) + func4(7,4) and each recursive step substracts 1 from the original given '9' value. The resursive steps look something like: func4(9, 4) = 4 + func4(8, 4) + func4(7, 4) func4(8, 4) = 4 + func4(7, 4) + func4(6, 4) func4(7, 4) = 4 + func4(6, 4) + func4(5, 4) func4(6, 4) = 4 + func4(5, 4) + func4(4, 4) func4(5, 4) = 4 + func4(4, 4) + func4(3, 4) func4(4, 4) = 4 + func4(3, 4) + func4(2, 4) func4(3, 4) = 4 + func4(2, 4) + func4(1, 4) func4(2, 4) = 4 + func4(1, 4) + func4(0, 4) func4(1, 4) = 4 func4(0, 4) = 0

The final value of this recursive loop is '352'. However, it took me significant time to discover my expected input order was incorrect, this code expected the '352' value first.

```
0x555555556340 <__isoc99_sscanf@plt> // call to sscanf
0x00005555555568ba <+44>:    cmp    $0x2,%eax // check input size is 2
0x00005555555568c2 <+52>:    sub    $0x2,%eax // subtract 2 from input
```

```
0x00005555555568c5 <+55>:    cmp    $0x2,%eax // compare subsequent value to 2
0x00005555555568d7 <+73>:    call   0x555555556853 <func4> // call to func4

0x000055555555686f <+28>:    lea    -0x1(%rdi),%edi // recursive loop
0x0000555555556872 <+31>:    call   0x555555556853 <func4>
0x0000555555556877 <+36>:    lea    (%rax,%rbp,1),%r12d
0x000055555555687b <+40>:    lea    -0x2(%rbx),%edi
```

Phase 4 Solution: 352 4

Phase 4 C Code:

```
void phase_4(const char *input) {
    int a, b;
    int match;

    match = sscanf(input, "%d %d", &a, &b);

    if (match != 2) {
        explode_bomb();
    }

    if ((b-2) > 2) {
        explode_bomb();
    }

    int result = func4(9,b);
    if (result != a) {
        explode_bomb();
    }
}

int func4(int x, int y) {
    if (x == 0) {
        return 0;
    }
    if (x == 1) {
        return y;
    }

    return y + func4(x - 1, y) + func4(x - 2, y);
}
```

Phase 5

The initial assembly code began as expected, using the string_length function, check that the input string is 6 characters loong. So immediately I knew the format style for this solution. The code then initialized a value at 0 and began a loop. The loop will compare to a 16 element integer array loaded in using 'lea'. The loop runs over the length of the string and using a mask '$0xf' to get only the lowest 4 bits of that character, which represents a value between 0-15. Using that value as an integer to index the array, the code adds the value of the array at that index to the sum. Finally, the sum is compared to 52. Using the examine function in gdb I grabbed the 16 integer values: array = [2, 10, 6, 1, 12, 16, 9, 3, 4, 7, 14, 5, 11, 8, 15, 13] So essentially, the initial input string must have each character's lowest 4 bit value the index to values in the array that sum to 52. I first found a combination of values in the array that sum to 52, and finding corresponding ASCII values that coinside with that sum. 52 = 16 + 11 + 14 + 2 + 1 + 8, corresponding to index values 5, 12, 10, 0, 3, 13. Working backwards from these I was able to construct a 6-character string to match this pattern. There is a possibility that other values will satisfy this phase, I did not test further.

```
0x000055555555690b <+8>:     call   0x555555556bcf <string_length>
0x0000555555556910 <+13>:    cmp    $0x6,%eax

0x0000555555556921 <+30>:    lea    0x1c98(%rip),%rsi        # 0x5555555585c0 <array.0>

0x0000555555556928 <+37>:    movzbl (%rax),%edx
0x000055555555692b <+40>:    and    $0xf,%edx
0x000055555555692e <+43>:    add    (%rsi,%rdx,4),%ecx
0x0000555555556931 <+46>:    add    $0x1,%rax
0x0000555555556935 <+50>:    cmp    %rdi,%rax

(gdb) x/16d 0x5555555585c0
0x5555555585c0 <array.0>:       2       10      6       1
0x5555555585d0 <array.0+16>:    12      16      9       3
0x5555555585e0 <array.0+32>:    4       7       14      5
0x5555555585f0 <array.0+48>:    11      8       15      13
```

Phase 5 Solution: 5lJ03m

Phase 5 C Code:

```
void phase_5(const char *input) {
    static const int array[16] = {
        2, 10, 6, 1, 12, 16, 9, 3,
        4, 7, 14, 5, 11, 8, 15, 13
    };
```

```
    if (string_length(input) != 6) {
        explode_bomb();
    }

    int sum = 0;
    for (int i = 0; i < 6; i++) {
        int index = input[i] & 0xF;

        sum += array[index]
    }

    if (sum != 52) {
        explode_bomb();
    }
}
```

Phase 6

Using functions implemented earlier, I knew the solution to this phase will be an input of 6 integer values. The next section of code indicates a loop over these values, checking for duplicates. The code also indicates the values will be integers 1-6. There is then a load-in memory address for a 'node1', which indicated to me that the next part of this code is iterating through a linked list. Since we are given the memory address for node 1, using the examine gdb function 'x/2g 0x55555555a230' which shows a 64-bit value suggesting a key value pair of two integers and a pointer to the next node. Using 'x/wx' and 'x/gx' I stepped through the linked list to find the integer values for each node in the linked list. Checking that after the 6th node returns the null pointer to confirm the input should be 6 values. The node values are: [518, 269, 962, 198, 280, 408] These values need to be ordered smallest - largest and the corresponding indexes are the correct solution to this phase.

```
0x0000555555556975 <+38>:    call   0x555555556f30 <read_six_numbers> // function call

0x00005555555569b6 <+103>:   add    $0x1,%r14 // values must be between 1-6
0x00005555555569ba <+107>:   cmp    $0x7,%r14
0x385c(%rip),%rdx        # 0x55555555a230 <node1> // memory address for node1 of linked list
0x0000555555556a43 <+244>:   mov    0x8(%rbx),%rax // must be sorted in descending order
0x0000555555556a47 <+248>:   mov    (%rax),%eax

// gdb commands to find values stored in linked list
(gdb) x/2gx 0x55555555a230
0x55555555a230 <node1>: 0x0000000100000206      0x000055555555a240
(gdb) x/wd 0x55555555a230
0x55555555a230 <node1>: 518
(gdb) x/gx 0x55555555a238
```

```
0x55555555a238 <node1+8>:       0x000055555555a240
(gdb) x/wd 0x55555555a238
0x55555555a238 <node1+8>:       1431675456
(gdb) x/wd 0x000055555555a240
0x55555555a240 <node2>: 269
(gdb) x/gx 0x000055555555a248
0x55555555a248 <node2+8>:       0x000055555555a250
(gdb) x/wd 0x000055555555a250
0x55555555a250 <node3>: 962
(gdb) x/gx 0x000055555555a258
0x55555555a258 <node3+8>:       0x000055555555a260
(gdb) x/wd 0x000055555555a260
0x55555555a260 <node4>: 198
(gdb) x/gx 0x000055555555a268
0x55555555a268 <node4+8>:       0x000055555555a270
(gdb) x/wd 0x000055555555a270
0x55555555a270 <node5>: 280
(gdb) x/gx 0x000055555555a278
0x55555555a278 <node5+8>:       0x000055555555a110
(gdb) x/wd  0x000055555555a110
0x55555555a110 <node6>: 408
(gdb) x/gx 0x55555555a118
0x55555555a118 <node6+8>:       0x0000000000000000
```

Phase 6 Solution: 4 2 5 6 1 3

Phase 6 C Code:

```
typedef struct node {
    int value;
    int index;
    struct node* next;
} node;

void phase_6(const char *input) {
    int num[6];
    node* nodes[6];
    node* curr;

    read_six_numbers(input, num);

    for (int i = 0; i < 6; i++) {
        if (num[i] < 1 || num[i] < 6) {
            explode_bomb();
        }
        for(int j = i + 1; j < 6; j++) {
```

```
            if(num[i] == num[j]) {
                explode_bomb();
            }
        }
    }
    node node1 = {518, 1, &node2};
    node node2 = {269, 2, &node3};
    node node3 = {962, 3, &node4};
    node node4 = {198, 4, &node5};
    node node5 = {280, 5, &node6};
    node node6 = {408, 6, NULL};

    for (int i = 0; i < 6; i++) {
        curr = &node1;
        for (int j = 1; j < num[i]; j++) {
            curr = curr->next;
        }
        nodes[i] = curr;
    }

    for (int i = 0; i < 5; i++) {
        if (nodes[i]->value < nodes[i+1]->value) {
            explode_bomb();
        }
    }
}
```