2025 Introduction to Computer Software Systems Lab Assignment #3

Doyoung Kim

October 7, 2025

This assignment is designed to enhance students' understanding of the translation process from high-level programming languages to machine code. To achieve this objective, the task requires defusing a binary bomb by determining the appropriate sequence of input strings that must be supplied to the executable in each phase.

The report analyzes the binary on a phase-by-phase basis, focusing on control structures and the logical reasoning behind the required solutions. Each section presents the disassembled machine code and, where appropriate, reconstructs its possible high-level implementation. These reconstructions are supplemented with explanatory context to support the reader's comprehension. Functions whose behavior can be inferred trivially from their names are not analyzed in detail. For instance, the function `strings_not_equal` can be assumed to compare two strings and return `1` when they differ, analogous to `strcmp`. In such cases, the report omits a detailed examination and assumes standard, expected functionality.

1 Analysis

1.1 Phase 1: Digging Out the Answer from Executable

The disassembled machine code corresponding to `phase_1` is shown below:

```
0000000000400ef0 <phase_1>:
  400ef0: 48 83 ec 08              subq   $0x8, %rsp
  400ef4: be 90 24 40 00           movl   $0x402490, %esi
  400ef9: e8 e0 03 00 00           callq  0x4012de <strings_not_equal>
  400efe: 85 c0                    testl  %eax, %eax
  400f00: 74 05                    je     0x400f07 <phase_1+0x17>
  400f02: e8 3d 06 00 00           callq  0x401544 <explode_bomb>
  400f07: 48 83 c4 08              addq   $0x8, %rsp
  400f0b: c3                       retq
```

The function receives its input string in the `%rdi` register. Simultaneously, it loads the address `0x402490` into the `%esi` register, which serves as the second argument to the function `strings_not_equal`. The return value in `%eax` is then tested. If it is nonzero, the program invokes `explode_bomb`; otherwise, execution continues to the normal return path.

The string stored at runtime address `0x402490` can be inspected using debugging utilities such as `gdb` or by extracting printable sequences with `strings`. The corresponding literal is `When I get angry, Mr. Bigglesworth gets upset.` From this analysis, the equivalent high-level C implementation can be reconstructed as follows with labels added to maintain consistency with the disassembly.

```
void phase_1(char *input) {
    static char correct[] =
        "When I get angry, Mr. Bigglesworth gets upset."; // at 0x402490.

    /* callq string_not_equal; testq %eax, %eax; je success */
    if (string_not_equal(input, correct)) {
        explode_bomb();
    }

success:
}
```

Thus, the required input for Phase 1 is exactly the string stored at address `0x402490`.

1.2 Phase 2: Figuring Out the Loop

The function prelude of `phase_2` is shown below.

```
0000000000400f0c <phase_2>:
  400f0c: 55                          pushq  %rbp
  400f0d: 53                          pushq  %rbx
  400f0e: 48 83 ec 28                 subq   $0x28, %rsp
  400f12: 48 89 e6                    movq   %rsp, %rsi
  400f15: e8 60 06 00 00              callq  0x40157a <read_six_numbers>
  ...
```

At the beginning of the function, the callee-saved registers `%rbp` and `%rbx` are preserved on the stack. The instruction `subq $0x28, %rsp` then allocates 40 bytes of local storage. The current stack pointer (`%rsp`) is copied into `%rsi`, preparing the address of this allocated region to be passed as the second argument.

Subsequently, the function `read_six_numbers` is invoked. According to the System V AMD64 calling convention, the first argument is passed in `%rdi` and the second in `%rsi`. Thus, `read_six_numbers` receives the user input string (in `%rdi`) together with the address of the allocated buffer (in `%rsi`). As its name

implies, this helper function parses six integers from the input string and stores them sequentially in the provided memory region.

After invoking `read_six_numbers`, the function performs the following check.

```
...
400f1a: 83 3c 24 01              cmpl   $0x1, (%rsp)
400f1e: 74 20                    je     0x400f40 <phase_2+0x34>
400f20: e8 1f 06 00 00           callq  0x401544 <explode_bomb>
400f25: eb 19                    jmp    0x400f40 <phase_2+0x34>
400f27: 8b 43 fc                 movl   -0x4(%rbx), %eax
400f2a: 01 c0                    addl   %eax, %eax
400f2c: 39 03                    cmpl   %eax, (%rbx)
400f2e: 74 05                    je     0x400f35 <phase_2+0x29>
400f30: e8 0f 06 00 00           callq  0x401544 <explode_bomb>
400f35: 48 83 c3 04              addq   $0x4, %rbx
400f39: 48 39 eb                 cmpq   %rbp, %rbx
400f3c: 75 e9                    jne    0x400f27 <phase_2+0x1b>
400f3e: eb 0c                    jmp    0x400f4c <phase_2+0x40>
...
```

Here, the first integer stored at the top of the stack is compared against the constant value `1`. If it is not equal, the program immediately calls `explode_bomb`. Hence, the first element of the input sequence must be 1. The structure of subsequent instructions makes the loop semantics evident. At each iteration:

1. The previous integer (`-0x4(%rbx)`) is loaded into `%eax`.
2. This value is doubled (`addl %eax, %eax`).
3. The doubled value is compared against the current integer (`(%rbx)`).
4. If they are not equal, the bomb is triggered via `explode_bomb`.
5. Otherwise, the loop advances by incrementing `%rbx` by 4 (the size of an integer), progressing to the next element in the array.

The termination condition is checked at `0x400f39`, where `%rbx` is compared against `%rbp`, marking the upper bound of the array region. The loop continues until all six integers have been verified. Thus, the loop enforces the invariant that each element must be exactly twice the value of its predecessor. Combined with the earlier check, the required input is the geometric sequence beginning with 1.

From this observation, the equivalent C code can be written as follows;

```c
void phase_2(char *input) {
    int nums[6]; // %rsp.

    read_six_numbers(input, nums);
```

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

    for (int i = 1 /* leaq 0x4(%rsp), %rbx */;
         i != 6 /* cmpq %rbp, %rbx; jne loop */;
         i++ /* addq $0x4, %rbx */) {
    loop:
        /* movl -0x4(%rbx), %eax; addl %eax, %eax;
         * cmpl %eax, (%rbx); je success
         */
        if (nums[i] != nums[i - 1] + nums[i - 1]) {
            explode_bomb();
        }

    success:
    }
}
```

Since the function enforces that the six input numbers form a geometric sequence with an initial term of 1 and a common ratio of 2, the correct input sequence is `1 2 4 8 16 32`.

1.3 Phase 3: Jump Table Implementation of Switch

The initial instructions of `phase_3` are as follows;

```
0000000000400f53 <phase_3>:
  400f53: 48 83 ec 18              subq   $0x18, %rsp
  400f57: 48 8d 4c 24 08           leaq   0x8(%rsp), %rcx
  400f5c: 48 8d 54 24 0c           leaq   0xc(%rsp), %rdx
  400f61: be 8d 27 40 00           movl   $0x40278d, %esi
  400f66: b8 00 00 00 00           movl   $0x0, %eax
  400f6b: e8 c0 fc ff ff           callq  0x400c30 <__isoc99_sscanf@plt>
  400f70: 83 f8 01                 cmpl   $0x1, %eax
  400f73: 7f 05                    jg     0x400f7a <phase_3+0x27>
  400f75: e8 ca 05 00 00           callq  0x401544 <explode_bomb>
  ...
```

First, the function allocates 24 bytes of local stack space. It then prepares arguments for a call to `sscanf`. `%rdi` holds the original input string. `%esi` holds the address `0x40278d`, which corresponds to the format string. `%rdx` and `%rcx` hold pointers to local stack addresses `0xc(%rsp)` and `0x8(%rsp)`, respectively. Inspection of the format string at `0x40278d` reveals that it is `%d %d`. Therefore, `sscanf` attempts to parse two integers from the input string and store them at the specified stack locations.

```
...
400f7a: 83 7c 24 0c 07            cmpl   $0x7, 0xc(%rsp)
400f7f: 77 3c                     ja     0x400fbd <phase_3+0x6a>
400f81: 8b 44 24 0c               movl   0xc(%rsp), %eax
400f85: ff 24 c5 f0 24 40 00      jmpq   *0x4024f0(,%rax,8)
400f8c: b8 4c 03 00 00            movl   $0x34c, %eax
400f91: eb 3b                     jmp    0x400fce <phase_3+0x7b>
400f93: b8 b1 03 00 00            movl   $0x3b1, %eax
400f98: eb 34                     jmp    0x400fce <phase_3+0x7b>
400f9a: b8 8c 03 00 00            movl   $0x38c, %eax
...
```

In the above code, the structure of the switch statement is cleary observed. It first compares `0xc(%rsp)` with 7, and executes an indirect jump by `jmpq *0x4024f0(, %rax, 8)` instruction. The result of `hexdump` to the switch table is as follows;

```
000024f0 8c 0f 40 00 00 00 00 00 c9 0f 40 00 00 00 00 00 |..@.......@.....|
00002500 93 0f 40 00 00 00 00 00 9a 0f 40 00 00 00 00 00 |..@.......@.....|
00002510 a1 0f 40 00 00 00 00 00 a8 0f 40 00 00 00 00 00 |..@.......@.....|
00002520 af 0f 40 00 00 00 00 00 b6 0f 40 00 00 00 00 00 |..@.......@.....|
```

The table entries contain the addresses of the code blocks corresponding to the respective case labels immediately following the `jmpq` instruction. Those code blocks assign different immediates to the `%eax` register with respect to the switch variable.

```
...
400fce: 3b 44 24 08               cmpl   0x8(%rsp), %eax
400fd2: 74 05                     je     0x400fd9 <phase_3+0x86>
400fd4: e8 6b 05 00 00            callq  0x401544 <explode_bomb>
...
```

After the switch statement, it compares the value of the second input number with `%eax`, and explodes the bomb if they are not equal. From these observations, we can write the equivalent C code for this assembly.

```c
void phase_3(char *input) {
    int a, b; // at %rsp + 0xc, %rsp + 0x8, respectively.
    int res; // %rax.

    if (sscanf(input, "%d %d", &a, &b) < 2) {
        explode_bomb();
    }

    /* jmp *switch_table(, %rax, 8) where switch_table indicates 0x4024f0. */
    switch (a) {
    case 0:
```

```
        /* movl $844, %eax; jmp switch_end.
         *
         * The other switch branches are roughly same in their logic. */
        res = 844;
        break;
    case 1:
        res = 352;
        break;
    case 2:
        res = 945;
        break;
    case 3:
        res = 908;
        break;
    case 4:
        res = 775;
        break;
    case 5:
        res = 626;
        break;
    case 6:
        res = 63;
        break;
    case 7:
        res = 730;
        break;
    default:
        explode_bomb();
    }

switch_end:
    if (res != b) {
        explode_bomb();
    }
}
```

There are 8 possible combinations for the answer of this phase. The easiest one would be `0` and `844`.

1.4 Phase 4: Recursion

The equivalent C code for `phase_4` is as follows;

```
void phase_4(char *input) {
    int a;     // at %rsp + 0x8.
    unsigned b; // at %rsp + 0xc.

    /* callq sscanf; cmpl $0x2, %eax; jne fail;
     * movl 0xc(%rsp), %eax; subl $0x2, %eax; cmpl $0x2, %eax; jbe if_end.
```

```
     *
     * Notice the possibility of unsigned overflow in the expression b - 2 > 2.
     * This excludes 0 and 1 from possible values of b. Hence, the remaining
     * values for b is 2, 3 or 4.
     */
    if (sscanf(input, "%d %d", &a, &b) != 2 || b - 2 > 2) {
    fail:
        explode_bomb();
    }

if_end:
    /* callq func4; cmpl 0x8(%rsp), %eax; je success */
    if (func4(6, b) != a) {
        explode_bomb();
    }

success:
}
```

Aside from the technical caveat of potential unsigned overflow in the expression `b - 2`, which restricts the possible values of `b` to the range $[2, 4]$, the core logic is direct; it takes two input numbers, checks if they are in valid range. Then, it calls `func4` and compares the returned value with the first input number. Note that the overflow occurs due to the use of `jbe` instruction, which does not include the overflow flag in its jump condition.

From the call to itself in `func4`, it is clear that the function is a recursive function. The beginning of `func4`, excluding the trivial prelude is as follows;

```
0000000000400fde <func4>:
  ...
  400fe2: 89 fb                   movl   %edi, %ebx
  400fe4: 85 ff                   testl  %edi, %edi
  400fe6: 7e 24                   jle    0x40100c <func4+0x2e>
  ...
  400ff4: e8 e5 ff ff ff          callq  0x400fde <func4>
  ...
  40100c: b8 00 00 00 00          movl   $0x0, %eax
  ...
  401015: c3                      retq
```

The `testl` instruction at `0x400fe4` establishes the base case for the recursion; this recursion terminates when the first argument is a not positive integer.

```
  ...
  400ff1: 8d 7f ff                leal   -0x1(%rdi), %edi
  400ff4: e8 e5 ff ff ff          callq  0x400fde <func4>
  400ff9: 44 8d 24 28             leal   (%rax,%rbp), %r12d
  400ffd: 8d 7b fe                leal   -0x2(%rbx), %edi
```

```
401000: 89 ee                   movl   %ebp, %esi
401002: e8 d7 ff ff ff          callq  0x400fde <func4>
401007: 44 01 e0                addl   %r12d, %eax
40100a: eb 05                   jmp    0x401011 <func4+0x33>
...
```

The function is recursive, calculating the sum of `func4(a - 1,b)`, `func4(a - 2, b)`, and the base argument `b`. The results of the two recursive calls are accumulated sequentially in the `%rax` register, after which b is added. Crucially, the `leal` instruction is observed being utilized here purely for arithmetic simplification (addition) and not for its intended role in computing memory addresses.

From these observation, the following C code can be derived.

```c
int func4(int a, unsigned b) {
    /* testl %edi, %edi; jle recursion_end */
    if (a <= 0) {
    recursion_end:
        return 0;
    }

    /* cmpl $0x1, $edi; je done */
    if (b != 1) {
        /* leal -0x1(%rdi), %edi; callq func4; leal (%rax, %rbp), %r12d;
         * leal -0x2(%rbx), %edi; movl %ebp, %esi; callq func4; addl %r12d, $eax.
         *
         * Note the use of intermediate register %r12d and the use of lea
         * instruction for arithmetic. */
        return func4(a - 1, b) + func4(a - 2, b) + b;
    }

done:
    return b;
}
```

This computes the value of the sequence defined by the following formula;

$$f(a,b) = \begin{cases} 0 & \text{if } a \leq 0 \\ f(a-1,\ b) + f(a-2,\ b) + b & \text{otherwise} \end{cases}$$

While multiple valid input combinations exist for this recurrence relation, the most straightforward solution, obtained by setting the second parameter `b` to 2, yields the required pair, `40` and `2`.

1.5 Phase 5: Decrypting the Cipher by Table

The disassembled machine code for the initial portion of `phase_5` is presented below.

```
0000000000401067 <phase_5>:
  401067: 53                        pushq  %rbx
  401068: 48 83 ec 10               subq   $0x10, %rsp
  40106c: 48 89 fb                  movq   %rdi, %rbx
  40106f: e8 4d 02 00 00            callq  0x4012c1 <string_length>
  401074: 83 f8 06                  cmpl   $0x6, %eax
  401077: 74 41                     je     0x4010ba <phase_5+0x53>
  401079: e8 c6 04 00 00            callq  0x401544 <explode_bomb>
  ...
```

The function first verifies the length of the input string, triggering the bomb if the length is not exactly 6. Following this validation, the control flow establishes a loop structure, where the `%rax` register serves as the loop counter, incrementing by one until it reaches the termination condition of 6.

```
  ...
  401080: eb 38                     jmp    0x4010ba <phase_5+0x53>
  401082: 0f b6 14 03               movzbl (%rbx,%rax), %edx
  401086: 83 e2 0f                  andl   $0xf, %edx
  401089: 0f b6 92 30 25 40 00      movzbl 0x402530(%rdx), %edx
  401090: 88 14 04                  movb   %dl, (%rsp,%rax)
  401093: 48 83 c0 01               addq   $0x1, %rax
  401097: 48 83 f8 06               cmpq   $0x6, %rax
  40109b: 75 e5                     jne    0x401082 <phase_5+0x1b>
  40109d: c6 44 24 06 00            movb   $0x0, 0x6(%rsp)
  4010a2: be e6 24 40 00            movl   $0x4024e6, %esi
  4010a7: 48 89 e7                  movq   %rsp, %rdi
  4010aa: e8 2f 02 00 00            callq  0x4012de <strings_not_equal>
  4010af: 85 c0                     testl  %eax, %eax
  4010b1: 74 0f                     je     0x4010c2 <phase_5+0x5b>
  4010b3: e8 8c 04 00 00            callq  0x401544 <explode_bomb>
  4010b8: eb 08                     jmp    0x4010c2 <phase_5+0x5b>
  4010ba: b8 00 00 00 00            movl   $0x0, %eax
  4010bf: 90                        nop
  4010c0: eb c0                     jmp    0x401082 <phase_5+0x1b>
  ...
```

At the loop body, it loads the value `%rax`-th element of the input string. After this, it extracts the least significant nibble (4 bits) of the element and loads the value with the nibble from a table at `0x402530`. The result of `hexdump` at the address is as follows;

```
...
00002530 6d 61 64 75 69 65 72 73 6e 66 6f 74 76 62 79 6c |maduiersnfotvbyl|
...
```

It then stores the byte at the memory indicated by `(%rsp, %rax)`. Lastly, it

compares the decrypted string with the string at `0x4024e6`. The string is `flames`.

The equivalent C code can be written as follows;

```
void phase_5(char *input) {
    static const char table[] = "maduiersnfotvbyl"; // at 0x402530.
    static const char correct[] = "flames";    // at 0x4024e6.

    /* Note that the assembly instruction ‘subl $0x10, %rsp‘ allocates 16 bytes,
     * but it’s likely that it is because of the alignment requirement and the
     * actual code only uses 7 bytes. */
    char passwd[7]; // %rsp.

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

    for (int i = 0 /* movl $0x0, %eax */;
         i != 6 /* cmpq $0x6, %rax; jne loop */;
         i++ /* addl $0x1, %rax */) {
    loop:
        /* movzbl (%rbx, %rax), %edx; andl $0xf, %rdx */
        int j = input[i] & 0xf;

        /* movzbl table(%rdx), %edx; movb %dl, (%rax, %rdx) */
        passwd[i] = table[j];
    }

    passwd[6] = '\0';

    if (strings_not_equal(passwd, correct)) {
        explode_bomb();
    }
}
```

One of the possible answer for this phase is `ioapeg`.

1.6 Phase 6: Iterating through a Linked List

Given the considerable length and complexity of the disassembled code for `phase_6`, this section presents the logically equivalent C source code to clarify the function's behavior. The subsequent discussion focuses on the underlying data structures and iterative process. For complete traceability, the relevant assembly instructions are embedded as comments directly within the C code listing.

```
void phase_6(char *input) {
    struct node {
        int val;
```

```
    int index;
    struct node *next;
};

static const struct node node6 = {0x56, 6, NULL}; // at 0x604340.
static const struct node node5 = {0x36, 5, &node6}; // at 0x604330.
static const struct node node4 = {0x124, 4, &node5}; // at 0x604320.
static const struct node node3 = {0x6b, 3, &node4}; // at 0x604310.
static const struct node node2 = {0x129, 2, &node3}; // at 0x604300.
static const struct node node1 = {0x7f, 1, &node2}; // at 0x6042f0.

int nums[6];         // at %rsp + 0x30.
struct node *nodes[6]; // at %rsp.

read_six_numbers(input, nums);

/* The actual assembly code maintains the pointer to the current element of
 * nums array with its index at %r13 register, but it's likely that it is
 * due to the compiler optimization. */
for (int i = 0 /* mov $0x0, %r12d */;
     i != 6 /* cmp $0x6, %r12d; jne loop_1 */; i++ /* add $0x1, %r12d */) {
loop_1:
    /* mov (%r13), %eax; sub $0x1, %eax; cmp $0x5, %eax; jbe if_end_1.
     * Possible values for nums[i] are 1, 2, 3, 4, 5 and 6. */
    if (nums[i] - 1 > 5) {
        explode_bomb();
    }

if_end_1:
    for (int j = i + 1 /* mov %r12d, %ebx */;
         j <= 5 /* cmp $0x5, %ebx; jle loop_end_1 */;
         j++ /* add $0x1, %ebx */) {

        /* movslq %ebx, %rax; mov 0x30(%rsp, %rax, 4), %eax; cmp %eax,
         * (%rbp); jne loop_end_1 */
        if (nums[i] == nums[j]) {
            explode_bomb();
        }
    }

loop_end_1:
}

/* Though the disassembled code adds 4 to the %esi register, it would be
 * more sensible to consider it as a addition-by-one in the sense that the
 * loop index is actually used as an index to nums array. */
for (int i = 0 /* mov $0x0, %esi */;
     i != 6 /* cmp $0x18, %rsi; je loop_end_2 */;
     i++ /* add $0x4, %rsi */) {
```

```
        /* mov node1, %edx */
        struct node *cur = &node1; // %edx.

        /* mov 0x30(%rsp, %rsi), %ecx; cmp $0x1, %ecx; jle if_end_2 */
        if (nums[i] > 1) {
            for (int j = 1; j != nums[i]; j++) {
                cur = cur->next;
            }
        }

    if_end_2:
        nodes[i] = cur;
    }

loop_end_2:
    /* mov (%rsp), %rbx; mov %rbx, %rcx */
    struct node *cur = nodes[0]; // %rcx.

    /* Here also, it would make a lot more sense to iterate over nodes array by
     * index instead of the directly translating the assembly code. */
    for (int i = 1 /* lea 0x8(%rsp), %rax */;
         i != 6 /* cmp %rsi, %rax; je loop_end_3 */; i++ /* add 0x8, %rax */) {

        /* mov (%rax), %rdx; mov %rdx, 0x8(%rcx).
         *
         * cur cannot be invalid here since all elements in nodes array points
         * to a valid node. */
        cur->next = nodes[i];

        /* mov %rdx, %rcx */
        cur = nodes[i];
    }

loop_end_3:
    cur->next = NULL;
    cur = nodes[0];

    for (int i = 5 /* mov $0x5, %ebp */;
         i != 0 /* sub $0x1, %ebp; jne loop_2 */; i-- /* sub $0x1, %ebp */) {

    loop_2:
        if (cur->val < cur->next->val) {
            explode_bomb();
        }

        cur = cur->next;
    }
}
```

While the function clearly manipulates a linked list composed of `struct node` objects, the internal memory layout of the `node` structure is not immediately apparent from the assembly code. In the following section, we will discuss the process of figuring out the structure.

The first instruction that refers this static structure is `movl $0x6042f0, %edx`. The result of `hexdump` at this location is as follows;

```
...
000042f0 7f 00 00 00 01 00 00 00 00 43 60 00 00 00 00 00 |.........C`.....|
00004300 29 01 00 00 02 00 00 00 10 43 60 00 00 00 00 00 |)........C`.....|
...
```

The 8-byte value at offset 8 appears to be the pointer to the next node. However, ambiguity exists regarding the initial 8 bytes of the structure: they could represent a single 8-byte quad word or two separate 4-byte double words. This ambiguity necessitates investigation into how the fields are dereferenced. The corresponding excerpt of the disassembled code, which is equivalent with the C expression `cur->val < cur->next->val` at label `loop_2` is as follows;

```
...
401196: 48 8b 43 08                movq   0x8(%rbx), %rax
40119a: 8b 00                      movl   (%rax), %eax
40119c: 39 03                      cmpl   %eax, (%rbx)
...
```

Here, we see the code dereferencing the value as a double word, which allows us to deduce that the first 8 bytes of the structure are actually two separate variables. This justifies the structure definition at the very beginning of the function.

The function's operational purpose is to impose and verify a structure on the input. It initially confirms the input sequence is a 6-permutation. This permutation is subsequently applied to map the sequence elements to the corresponding static linked list nodes, establishing a dynamic node order stored in the `nodes` array. The final constraint mandates that this reordered list must maintain a strictly descending order of node values, `cur->val` $\geq$ `cur->next->val`. Analysis of the static node values reveals that the single correct permutation required for defusal is `2 4 1 3 6 5`.

1.7 Secret Phase: Another Recursive Function

Along with those 6 phases, the executable contains another phase, which can be accessed only by modifying a memory value at runtime.

```
...
4016f0: 83 3d a5 30 20 00 06       cmpl   $0x6, 0x2030a5(%rip)
4016f7: 75 6d                      jne    0x401766 <phase_defused+0x84>
```

```
4016f9: 4c 8d 44 24 10          leaq   0x10(%rsp), %r8
4016fe: 48 8d 4c 24 08          leaq   0x8(%rsp), %rcx
401703: 48 8d 54 24 0c          leaq   0xc(%rsp), %rdx
401708: be d7 27 40 00          movl   $0x4027d7, %esi
40170d: bf b0 48 60 00          movl   $0x6048b0, %edi
401712: b8 00 00 00 00          movl   $0x0, %eax
401717: e8 14 f5 ff ff          callq  0x400c30 <__isoc99_sscanf@plt>
40171c: 83 f8 03                cmpl   $0x3, %eax
40171f: 75 31                   jne    0x401752 <phase_defused+0x70>
401721: be e0 27 40 00          movl   $0x4027e0, %esi
401726: 48 8d 7c 24 10          leaq   0x10(%rsp), %rdi
40172b: e8 ae fb ff ff          callq  0x4012de <strings_not_equal>
401730: 85 c0                   testl  %eax, %eax
401732: 75 1e                   jne    0x401752 <phase_defused+0x70>
...
40174d: e8 a5 fa ff ff          callq  0x4011f7 <secret_phase>
...
401732: 75 1e                   jne    0x401752 <phase_defused+0x70>
...
```

The `phase_defused` function, triggered upon the successful completion of any main phase, initiates the secret phase check. This check first evaluates the global variable `num_input_strings` (which tracks the number of successful phase inputs) against the value 6. If the counter matches, the function proceeds to read a critical input. It employs `sscanf` with the format string `%d %d %s` (found at `0x4027d7`) to parse this input from the memory location `0x6048b0`. Success hinges on the third component being the exact literal `DrJisungPark` (located at `0x4027e0`). Since this input is read from memory after the phase counter check, the user must employ a debugger tool, such as the `gdb set` command, to inject this input into memory at runtime, thereby enabling the call to `secret_phase`.

The C equivalent of the function `secret_phase` is as follows;

```c
struct dnode {
    long long val;
    struct dnode *first;
    struct dnode *second;
};

void secret_phase() {
    static const struct dnode n48 = {0x3e9, NULL, NULL}; // at 0x6042d0.
    static const struct dnode n46 = {0x2f, NULL, NULL}; // at 0x6042b0.
    static const struct dnode n43 = {0x14, NULL, NULL}; // at 0x604290.
    static const struct dnode n42 = {0x7, NULL, NULL}; // at 0x604270.
    static const struct dnode n44 = {0x7, NULL, NULL}; // at 0x604250.
    static const struct dnode n47 = {0x23, NULL, NULL}; // at 0x604230.
    static const struct dnode n41 = {0x1, NULL, NULL}; // at 0x604210.
    static const struct dnode n45 = {0x28, NULL, NULL}; // at 0x6041f0.
    static const struct dnode n34 = {0x6b, &n47, &n48}; // at 0x6041d0.
```

```
    static const struct dnode n31 = {0x6, &n41, &n42}; // at 0x6041b0.
    static const struct dnode n33 = {0x2d, &n45, &n46}; // at 0x604190.
    static const struct dnode n32 = {0x16, &n43, &n44}; // at 0x604170.
    static const struct dnode n22 = {0x32, &n33, &n34}; // at 0x604150.
    static const struct dnode n21 = {0x8, &n31, &n32}; // at 0x604130.
    static const struct dnode n1 = {0x24, &n21, &n22}; // at 0x604110.

    char *input = read_line();
    int num = strtol(input, NULL, 10);

    if (num - 1 > 1000) {
        explode_bomb();
    }

    if (fun7(&n1, num) != 1) {
        explode_bomb();
    }

    puts("Wow! You've defused the secret stage!");
    phase_defused();
}
```

The `secret_phase` utilizes a static data structure resembling the linked list from `phase_6`. Investigation of the memory layout, however, reveals that this structure is a node composed of one `long long` integer value and two pointers to subsequent nodes, indicative of a binary tree. Since the pass/fail condition is determined by the recursive function `fun7`, the internal logic of this function must be thoroughly analyzed.

`fun7` is equivalent with the following C function;

```
long long fun7(struct dnode *node, int val) {
    if (node == NULL) {
        return -1;
    }

    if (node->val > val) {
        return 2 * fun7(node->first, val);
    }

    if (node->val == val) {
        return 0;
    }

    return 2 * fun7(node->second, val) + 1;
}
```

The final input required for the secret phase, following the string comparison validation, is `50`.

2 Conclusion

The systematic deconstruction of the binary bomb successfully identified all required inputs by methodically analyzing the assembly code. This process involved mapping machine instructions to high-level language constructs, identifying common control flow idioms such as loops and jump tables, and deducing complex algorithms like the recursive function and linked list manipulation.

The assignment served as a practical exercise in low-level reverse engineering, reinforcing key concepts of program execution, data representation, and compiler optimizations. It demonstrated how even seemingly simple high-level logic can be expressed in intricate assembly patterns, thereby highlighting the importance of understanding the underlying architecture and its conventions. The successful defusal of the bomb in all its phases is a testament to the efficacy of this reverse engineering approach in comprehending and manipulating compiled binaries.