CS 341 - Binary Bomb Lab Write-up

Adam Fasulo

April 23, 2025

Phase 1

Annotated Assembly Dump

The following is the annotated assembly code for `phase_1`, obtained using GDB's `disas` command. Annotations indicate the purpose of key instructions identified during the analysis.

```
Dump of assembler code for function phase_1:
   0x0000000000400f2d <+0>:   sub    $0x8,%rsp        # Allocate 8 bytes on the stack frame.
   0x0000000000400f31 <+4>:   mov    $0x4026f0,%esi    # Load immediate value 0x4026f0 (address of target strin
                                                     # %esi will hold the 2nd argument for strings_not_equal.
   0x0000000000400f36 <+8>:   call   0x401473 <strings_not_equal> # Call strings_not_equal function.
                                                     # Compares string in %rdi (user input) with string address
   0x0000000000400f3b <+13>:  test   %eax,%eax        # Test return value from strings_not_equal (in %eax).
                                                     # Sets Zero Flag (ZF) if %eax is 0 (strings are equal).
   0x0000000000400f3d <+15>:  je     0x400f44 <phase_1+23> # Jump if Equal (ZF=1). If strings matched, jump pas
   0x0000000000400f3f <+17>:  call   0x401742 <explode_bomb> # If strings did not match (ZF=0), call explode_bo
   0x0000000000400f44 <+23>:  add    $0x8,%rsp        # Deallocate the 8 bytes from the stack.
   0x0000000000400f48 <+27>:  ret                     # Return from phase_1 function.
End of assembler dump.
```

Procedure

Phase 1 requires a specific string input to be defused. The analysis of the assembly code revealed the following procedure:

1. **Function Analysis:** The `phase_1` function starts by allocating stack space (`sub $0x8, %rsp` at `<+0>`).

2. **Identifying the Comparison:** The core logic involves comparing the user's input string with a target string.

   - At `<+4>`, the instruction `mov $0x4026f0, %esi` loads the immediate value `0x4026f0` into the `%esi` register. This value represents the *memory address* where the secret target string is stored. This address is loaded into `%esi` to serve as the second argument to the comparison function.
   - At `<+8>`, the function `call 0x401473 <strings_not_equal>` is invoked. This function compares two strings: the first string's address is expected in `%rdi` (which holds the user's input string provided by the calling function `main` after calling `read_line`), and the second string's address is in `%esi` (the target string address `0x4026f0`).

3. **Checking the Result:** The `strings_not_equal` function returns 0 in `%eax` if the strings are identical, and a non-zero value otherwise.

   - At `<+13>`, `test %eax, %eax` checks if the return value in `%eax` is zero. It sets the CPU's Zero Flag (ZF) if `%eax` is 0.
   - At `<+15>`, `je 0x400f44` ("Jump if Equal") checks the Zero Flag. If ZF is set (meaning `%eax` was 0 and the strings matched), the program jumps directly to address `0x400f44`, which is the stack cleanup and return sequence.

4. **Handling Incorrect Input:** If the strings did not match, the return value in `%eax` is non-zero, the Zero Flag is not set, and the `je` instruction does not jump. Execution proceeds to the next instruction at `<+17>`.

   - `call 0x401742 <explode_bomb>`: This calls the `explode_bomb` function, indicating the input was incorrect.

5. **Finding the Target String:** The crucial step is to determine the string stored at memory address `0x4026f0`. This can be done safely using GDB:

   - Start GDB: `gdb ./bomb`
   - Set a breakpoint to prevent the bomb from exploding, for example, at the `explode_bomb` function itself: `break explode_bomb`
   - Run the program: `run`
   - When the program stops (either immediately if a breakpoint was set at `main`, or when incorrect input leads to the `explode_bomb` breakpoint), the program's memory is loaded.
   - Examine the string at the target address: `x/s 0x4026f0`
   - This command displays the null-terminated string stored at address `0x4026f0`.

6. **Solution:** The string revealed by the `x/s 0x4026f0` command in GDB is the required input for Phase 1. Providing this exact string when prompted will cause `strings_not_equal` to return 0, the `je` instruction will be taken, and the phase will be defused without calling `explode_bomb`.

7. **Conclusion:** Phase 1 is defused by identifying the address of the expected string from the assembly (`0x4026f0`), using GDB to inspect the contents of that memory address, and providing the retrieved string as input.

Phase 2

Annotated Assembly Dump

The following are the annotated assembly code snippets for `phase_2` and the helper function `read_six_numbers`, obtained using GDB's `disas` command. Annotations indicate the purpose of key instructions identified during the analysis.

phase_2:

```
Dump of assembler code for function phase_2:
   0x0000000000400f49 <+0>:   push   %rbp
   0x0000000000400f4a <+1>:   push   %rbx # will be used as a counter
   0x0000000000400f4b <+2>:   sub    $0x28,%rsp # setup 40 bytes on the stack
   0x0000000000400f4f <+6>:   mov    %fs:0x28,%rax # stack canary
   0x0000000000400f58 <+15>:  mov    %rax,0x18(%rsp)
   0x0000000000400f5d <+20>:  xor    %eax,%eax # 0 out eax
   0x0000000000400f5f <+22>:  mov    %rsp,%rsi # move top of stack into rsi so read_six_numbers knows where to
   0x0000000000400f62 <+25>:  call   0x401778 <read_six_numbers> # rsi is passed as second argument to function
   0x0000000000400f67 <+30>:  cmpl   $0x0,(%rsp)  # compare first argument with 0
   0x0000000000400f6b <+34>:  jns    0x400f72 <phase_2+41>   # jump if not signed SF = 0
   0x0000000000400f6d <+36>:  call   0x401742 <explode_bomb> # first number was < 0
   0x0000000000400f72 <+41>:  mov    %rsp,%rbp    # rsp is the same address as rsi which contains the string of
   0x0000000000400f75 <+44>:  mov    $0x1,%ebx    # set ebx to 1 prolly to set up a loop?
   0x0000000000400f7a <+49>:  mov    %ebx,%eax    # move 1 to eax. eax = index (starts at 1)
   0x0000000000400f7c <+51>:  add    0x0(%rbp),%eax # eax = current number + index
   0x0000000000400f7f <+54>:  cmp    %eax,0x4(%rbp) # compare sum (eax) with next array element (at rbp+4)
   0x0000000000400f82 <+57>:  je     0x400f89 <phase_2+64> # if next_num == current_num + index, jump and conti
   0x0000000000400f84 <+59>:  call   0x401742 <explode_bomb> # if not equal, explode
   0x0000000000400f89 <+64>:  add    $0x1,%ebx # increment counter/index i
   0x0000000000400f8c <+67>:  add    $0x4,%rbp # move base pointer to next element in array (4 bytes forward)
   0x0000000000400f90 <+71>:  cmp    $0x6,%ebx # if ebx = 6 then end loop (checked 5 pairs)
   0x0000000000400f93 <+74>:  jne    0x400f7a <phase_2+49> # go back to top of loop if ebx != 6
   0x0000000000400f95 <+76>:  mov    0x18(%rsp),%rax # stack canary stuff (check)
   0x0000000000400f9a <+81>:  xor    %fs:0x28,%rax # compare with original canary
```

```
   0x0000000000400fa3 <+90>:  je     0x400faa <phase_2+97> # if canaries are unchanged continue and clean up st
   0x0000000000400fa5 <+92>:  call   0x400b90 <__stack_chk_fail@plt> # Canary check failed
   0x0000000000400faa <+97>:  add    $0x28,%rsp # clean up stack allocation
   0x0000000000400fae <+101>: pop    %rbx
   0x0000000000400faf <+102>: pop    %rbp
   0x0000000000400fb0 <+103>: ret
End of assembler dump.
```

read_six_numbers:

```
Dump of assembler code for function read_six_numbers:
   0x0000000000401778 <+0>:   sub    $0x8,%rsp    # setup stack for 8 bytes
   0x000000000040177c <+4>:   mov    %rsi,%rdx    # rsi holds starting address for storing numbers, copy to rdx
   0x000000000040177f <+7>:   lea    0x4(%rsi),%rcx   # loads address for 2nd digit into rcx (4th sscanf arg)
   0x0000000000401783 <+11>:  lea    0x14(%rsi),%rax  # loads address for 6th digit into rax
   0x0000000000401787 <+15>:  push   %rax       # push address for 6th digit onto stack (8th sscanf arg)
=> 0x0000000000401788 <+16>:  lea    0x10(%rsi),%rax  # loads address for 5th digit into rax
   0x000000000040178c <+20>:  push   %rax       # push address for 5th digit onto stack (7th sscanf arg)
   0x000000000040178d <+21>:  lea    0xc(%rsi),%r9    # load address for 4th digit into r9 (6th sscanf arg)
   0x0000000000401791 <+25>:  lea    0x8(%rsi),%r8    # load address for 3rd digit into r8 (5th sscanf arg)
   0x0000000000401795 <+29>:  mov    $0x4029f1,%esi   # format string address for scanf (2nd sscanf arg)
   0x000000000040179a <+34>:  mov    $0x0,%eax        # Zero out eax (required for variadic sscanf call)
   0x000000000040179f <+39>:  call   0x400c40 <__isoc99_sscanf@plt> # Call sscanf (1st arg, input string, alrea
   0x00000000004017a4 <+44>:  add    $0x10,%rsp        # Clean up stack (remove the two pushed addresses)
   0x00000000004017a8 <+48>:  cmp    $0x5,%eax        # Check if sscanf read at least 6 numbers (returns count)
   0x00000000004017ab <+51>:  jg     0x4017b2 <read_six_numbers+58> # If > 5 nums read, continue
   0x00000000004017ad <+53>:  call   0x401742 <explode_bomb> # Explode if fewer than 6 numbers were read
   0x00000000004017b2 <+58>:  add    $0x8,%rsp         # Clean up remaining stack allocation
   0x00000000004017b6 <+62>:  ret                     # Return from function
End of assembler dump.
```

Procedure

Phase 2 requires a specific sequence of six integers as input. The analysis proceeded as follows:

1. **Input Requirements:** The function `phase_2` begins by calling `read_six_numbers` at `0x400f62`. Examining `read_six_numbers` reveals it uses `__isoc99_sscanf` (at `0x40179f`) to parse the input string. Based on the arguments prepared (pointers to consecutive 4-byte locations derived from `%rsi`, which points to the stack) and the check at `0x4017a8` (`cmp $0x5, %eax` followed by `jg`), it's clear the function expects exactly six integers separated by whitespace. The integers are stored as 4-byte values on the stack starting at the address initially held by `%rsp` in `phase_2`. If fewer than six integers are provided, `read_six_numbers` calls `explode_bomb`.

2. **First Number Check:** Back in `phase_2`, the instruction `cmpl $0x0, (%rsp)` at `0x400f67` compares the first integer read (now at the top of the stack) with 0. The next instruction, `jns 0x400f72`, jumps if the number is "not sign" (i.e., non-negative, ≥ 0). If the first number is negative, the jump is not taken, and `explode_bomb` is called at `0x400f6d`. Therefore, the first number in the sequence must be 0 or greater.

3. **Loop Analysis:** A loop is established starting at `0x400f72`.
   - `mov %rsp, %rbp`: The register `%rbp` is set to point to the beginning of the six-number sequence on the stack.
   - `mov $0x1, %ebx`: The register `%ebx` is initialized to 1. It serves as both a loop counter and an index for the pattern check.
   - The loop runs from address `0x400f7a` to `0x400f93`.
   - `cmp $0x6, %ebx` at `0x400f90` checks if the loop counter has reached 6.
   - `jne 0x400f7a` at `0x400f93` jumps back to the start of the loop if `%ebx` is not equal to 6. This means the loop iterates for `%ebx` values $1, 2, 3, 4, 5$, performing five comparisons in total.

4. **Pattern Identification:** The core logic resides within the loop (`0x400f7a` to `0x400f82`):

- `mov %ebx, %eax`: The current index (`%ebx`) is copied to `%eax`.
- `add 0x0(%rbp), %eax`: The integer value at the address pointed to by `%rbp` (the *current* number, N_{i-1}) is added to `%eax`. So, `%eax` now holds $N_{i-1} + index$.
- `cmp %eax, 0x4(%rbp)`: This compares the calculated value in `%eax` with the integer value stored 4 bytes after `%rbp` (the *next* number, N_i).
- `je 0x400f89`: If the values are equal, the check passes, and the program jumps to the loop increment logic.
- `call 0x401742 <explode_bomb>`: If `next_number != current_number + index`, the bomb explodes.

5. **Sequence Derivation:** The required pattern is: $N_0 \geq 0$, and for i from 1 to 5, $N_i = N_{i-1} + i$.

6. **Solution Verification (GDB):** Stepping through the code using GDB, as documented in the conversation, confirmed the required pattern. We tested the sequence starting with 1:

   - $N_0 = 1$ (satisfies $N_0 \geq 0$)
   - $N_1 = N_0 + 1 = 1 + 1 = 2$
   - $N_2 = N_1 + 2 = 2 + 2 = 4$
   - $N_3 = N_2 + 3 = 4 + 3 = 7$
   - $N_4 = N_3 + 4 = 7 + 4 = 11$
   - $N_5 = N_4 + 5 = 11 + 5 = 16$

   The conversation confirmed that inputting `"1 2 4 7 11 16"` successfully passed all checks. The GDB state showed `%ebx` reaching 6, and the final `jne` instruction at `0x400f93` was not taken, indicating the loop completed without triggering the bomb.

7. **Conclusion:** The correct input sequence to defuse Phase 2 is `1 2 4 7 11 16`.

Phase 3

Annotated Assembly Dump

The following is the annotated assembly code for `phase_3`, obtained using GDB's `disas` command. Annotations indicate the purpose of key instructions identified during the analysis.

```
Dump of assembler code for function phase_3:
=> 0x0000000000400fb1 <+0>:   sub    $0x28,%rsp # setup stack with 40 bytes
   0x0000000000400fb5 <+4>:   mov    %fs:0x28,%rax # stack canary setup
   0x0000000000400fbe <+13>:  mov    %rax,0x18(%rsp) # store canary on stack
   0x0000000000400fc3 <+18>:  xor    %eax,%eax # zero out eax = 0000....
   0x0000000000400fc5 <+20>:  lea    0x14(%rsp),%r8 # sets up pointer for 3rd sscanf arg (%d)
   0x0000000000400fca <+25>:  lea    0xf(%rsp),%rcx # sets up pointer for 2nd sscanf arg (%c)
   0x0000000000400fcf <+30>:  lea    0x10(%rsp),%rdx # sets up pointer for 1st sscanf arg (%d)
   0x0000000000400fd4 <+35>:  mov    $0x40274e,%esi # format string address for sscanf ("%d %c %d")
   0x0000000000400fd9 <+40>:  call   0x400c40 <__isoc99_sscanf@plt> # read input according to format string
   0x0000000000400fde <+45>:  cmp    $0x2,%eax # check return of sscanf - did it read >= 3 items?
   0x0000000000400fe1 <+48>:  jg     0x400fe8 <phase_3+55> # if >2 items read, jump past explode
   0x0000000000400fe3 <+50>:  call   0x401742 <explode_bomb> # if <=2 items read, explode
   0x0000000000400fe8 <+55>:  cmpl   $0x7,0x10(%rsp) # compare first input number (at 0x10(%rsp)) with 7
   0x0000000000400fed <+60>:  ja     0x4010ef <phase_3+318> # if first input > 7 (unsigned), jump to explode
   0x0000000000400ff3 <+66>:  mov    0x10(%rsp),%eax # move first input number into %eax (index for jump table)
   0x0000000000400ff7 <+70>:  jmp    *0x402760(,%rax,8) # indirect jump using jump table at 0x402760, indexed b

   # Case 0 handler (jump table entry: 0x0000000000400ffe)
   0x0000000000400ffe <+77>:  mov    $0x6d,%eax # load 0x6d (ASCII 'm') into %eax
   0x0000000000401003 <+82>:  cmpl   $0x31b,0x14(%rsp) # compare third input (number at 0x14(%rsp)) with 0x31b
   0x000000000040100b <+90>:  je     0x4010f9 <phase_3+328> # if equal, jump to final character check
   0x0000000000401011 <+96>:  call   0x401742 <explode_bomb> # if not equal, explode bomb
   0x0000000000401016 <+101>: mov    $0x6d,%eax # (unreachable) reload 'm' into eax
```

```
0x000000000040101b <+106>: jmp    0x4010f9 <phase_3+328> # (unreachable) jump to final check

# Case 1 handler (jump table entry: 0x0000000000401020)
0x0000000000401020 <+111>: mov    $0x75,%eax # load 0x75 (ASCII 'u') into %eax
0x0000000000401025 <+116>: cmpl   $0x325,0x14(%rsp) # compare third input (number) with 0x325 (805)
0x000000000040102d <+124>: je     0x4010f9 <phase_3+328> # if equal, jump to final check
0x0000000000401033 <+130>: call   0x401742 <explode_bomb> # if not equal, explode
0x0000000000401038 <+135>: mov    $0x75,%eax # (unreachable)
0x000000000040103d <+140>: jmp    0x4010f9 <phase_3+328> # (unreachable)

# Case 2 handler (jump table entry: 0x0000000000401042)
0x0000000000401042 <+145>: mov    $0x6e,%eax # load 0x6e (ASCII 'n') into %eax
0x0000000000401047 <+150>: cmpl   $0x176,0x14(%rsp) # compare third input (number) with 0x176 (374)
0x000000000040104f <+158>: je     0x4010f9 <phase_3+328> # if equal, jump to final check
0x0000000000401055 <+164>: call   0x401742 <explode_bomb> # if not equal, explode
0x000000000040105a <+169>: mov    $0x6e,%eax
0x000000000040105f <+174>: jmp    0x4010f9 <phase_3+328>

# Case 3 handler (jump table entry: 0x0000000000401064)
0x0000000000401064 <+179>: mov    $0x63,%eax # load 0x63 (ASCII 'c') into %eax
0x0000000000401069 <+184>: cmpl   $0x397,0x14(%rsp) # compare third input (number) with 0x397 (919)
0x0000000000401071 <+192>: je     0x4010f9 <phase_3+328> # if equal, jump to final check
0x0000000000401077 <+198>: call   0x401742 <explode_bomb> # if not equal, explode
0x000000000040107c <+203>: mov    $0x63,%eax
0x0000000000401081 <+208>: jmp    0x4010f9 <phase_3+328>

# Case 4 handler (jump table entry: 0x0000000000401083)
0x0000000000401083 <+210>: mov    $0x73,%eax # load 0x73 (ASCII 's') into %eax
0x0000000000401088 <+215>: cmpl   $0x99,0x14(%rsp) # compare third input (number) with 0x99 (153)
0x0000000000401090 <+223>: je     0x4010f9 <phase_3+328> # if equal, jump to final check
0x0000000000401092 <+225>: call   0x401742 <explode_bomb> # if not equal, explode
0x0000000000401097 <+230>: mov    $0x73,%eax
0x000000000040109c <+235>: jmp    0x4010f9 <phase_3+328>

# Case 5 handler (jump table entry: 0x000000000040109e)
0x000000000040109e <+237>: mov    $0x73,%eax # load 0x73 (ASCII 's') into %eax
0x00000000004010a3 <+242>: cmpl   $0xd6,0x14(%rsp) # compare third input (number) with 0xd6 (214)
0x00000000004010ab <+250>: je     0x4010f9 <phase_3+328> # if equal, jump to final check
0x00000000004010ad <+252>: call   0x401742 <explode_bomb> # if not equal, explode
0x00000000004010b2 <+257>: mov    $0x73,%eax
0x00000000004010b7 <+262>: jmp    0x4010f9 <phase_3+328>

# Case 6 handler (jump table entry: 0x00000000004010b9)
0x00000000004010b9 <+264>: mov    $0x6c,%eax # load 0x6c (ASCII 'l') into %eax
0x00000000004010be <+269>: cmpl   $0x1d3,0x14(%rsp) # compare third input (number) with 0x1d3 (467)
0x00000000004010c6 <+277>: je     0x4010f9 <phase_3+328> # if equal, jump to final check
0x00000000004010c8 <+279>: call   0x401742 <explode_bomb> # if not equal, explode
0x00000000004010cd <+284>: mov    $0x6c,%eax
0x00000000004010d2 <+289>: jmp    0x4010f9 <phase_3+328>

# Case 7 handler (jump table entry: 0x00000000004010d4)
0x00000000004010d4 <+291>: mov    $0x78,%eax # load 0x78 (ASCII 'x') into %eax
0x00000000004010d9 <+296>: cmpl   $0xdc,0x14(%rsp) # compare third input (number) with 0xdc (220)
0x00000000004010e1 <+304>: je     0x4010f9 <phase_3+328> # if equal, jump to final check
0x00000000004010e3 <+306>: call   0x401742 <explode_bomb> # if not equal, explode
0x00000000004010e8 <+311>: mov    $0x78,%eax
0x00000000004010ed <+316>: jmp    0x4010f9 <phase_3+328>

# Default case handler (input > 7)
0x00000000004010ef <+318>: call   0x401742 <explode_bomb> # explode bomb for invalid first input index

0x00000000004010f4 <+323>: mov    $0x63,%eax # (unreachable from ja) load 'c' into eax
```

```
# Final check for all valid cases
0x00000000004010f9 <+328>: cmp    0xf(%rsp),%al # compare second input char (at 0xf(%rsp)) with expected cha
0x00000000004010fd <+332>: je     0x401104 <phase_3+339> # if second input char matches expected char, jump
0x00000000004010ff <+334>: call   0x401742 <explode_bomb> # otherwise, explode bomb

# Clean up stack and return
0x0000000000401104 <+339>: mov    0x18(%rsp),%rax
0x0000000000401109 <+344>: xor    %fs:0x28,%rax
0x0000000000401112 <+353>: je     0x401119 <phase_3+360>
0x0000000000401114 <+355>: call   0x400b90 <__stack_chk_fail@plt>
0x0000000000401119 <+360>: add    $0x28,%rsp
0x000000000040111d <+364>: ret
```

Procedure

Phase 3 requires a specific input pattern consisting of three parts. Analysis of the assembly code revealed the following procedure:

1. **Input Reading:** The function begins by setting up the stack frame and then calls `__isoc99_sscanf` at address `0x400fd9`. The arguments passed via registers `%rdx`, `%rcx`, and `%r8` point to memory locations on the stack (`0x10(%rsp)`, `0xf(%rsp)`, and `0x14(%rsp)` respectively). The format string, likely `"%d %c %d"` based on the subsequent checks, indicates that `sscanf` attempts to read an integer, a character, and another integer from the input string. These correspond to the first, second, and third inputs required.

2. **Input Count Check:** Immediately after `sscanf`, the instruction `cmp $0x2, %eax` at `0x400fde` checks the return value of `sscanf` (stored in `%eax`). `sscanf` returns the number of items successfully scanned. The subsequent `jg 0x400fe8` instruction means "jump if greater". If `%eax` is greater than 2 (i.e., 3 or more items were successfully scanned), the program continues. Otherwise (if 0, 1, or 2 items were scanned), the jump is not taken, and `explode_bomb` is called at `0x400fe3`. This confirms that exactly three inputs are required.

3. **First Input Range Check:** At `0x400fe8`, the instruction `cmpl $0x7, 0x10(%rsp)` compares the first integer read by `sscanf` (stored at `0x10(%rsp)`) with the immediate value 7. The following instruction `ja 0x4010ef` ("jump if above") will jump to `0x4010ef`, which calls `explode_bomb`, if the first input is greater than 7 (unsigned comparison). This means the first input number must be in the range $[0, 7]$.

4. **Jump Table (Switch Statement):** The core logic uses a jump table to handle different cases based on the first input number.
   - The first input number (from `0x10(%rsp)`) is moved into `%eax` at `0x400ff3`.
   - The instruction `jmp *0x402760(,%rax,8)` at `0x400ff7` performs an indirect jump. It calculates an address by taking the base address `0x402760`, adding the value in `%rax` (the first input, $0-7$) multiplied by 8 (the scale factor, as addresses are 8 bytes). The program then jumps to the 8-byte address stored at this calculated location in memory. This effectively implements a switch statement based on the first input.
   - Inspecting the jump table memory using GDB (`x/8gx 0x402760`) reveals the target addresses for each case:
     - Case 0: `0x0000000000400ffe`
     - Case 1: `0x0000000000401020`
     - Case 2: `0x0000000000401042`
     - Case 3: `0x0000000000401064`
     - Case 4: `0x0000000000401083`
     - Case 5: `0x000000000040109e`
     - Case 6: `0x00000000004010b9`
     - Case 7: `0x00000000004010d4`

5. **Case-Specific Logic:** Each case handler performs two main actions before potentially jumping to the final check:

- It loads a specific immediate byte value into `%eax` (specifically, the lower byte `%al`). This value corresponds to the ASCII code of the expected *second* input character.
- It compares the *third* input number (read by `sscanf` into `0x14(%rsp)`) against a specific immediate value.
- If the comparison is equal (`je 0x4010f9`), it jumps to the final check. Otherwise, it calls `explode_bomb`.

For example, analyzing Case 0 (starting at `0x400ffe`):

- `mov $0x6d, %eax`: Loads $0x6d$ (109 decimal, ASCII 'm') into `%eax`.
- `cmpl $0x31b, 0x14(%rsp)`: Compares the third input number with $0x31b$ (795 decimal).
- `je 0x4010f9`: Jumps if the third input number is 795.

6. **Final Check:** All successful case paths converge at `0x4010f9`.
   - `cmp 0xf(%rsp), %al`: This compares the *second* input read by `sscanf` (the character stored at `0xf(%rsp)`) with the value in `%al` (the expected character loaded by the specific case handler).
   - `je 0x401104`: If the characters match, the jump is taken, bypassing the bomb and proceeding to the function's cleanup and return sequence.
   - `call 0x401742 <explode_bomb>`: If the characters do not match, the bomb explodes.

7. **Solution Derivation:** By examining each case handler (from the jump table addresses), we can determine the required third number and expected second character for each valid first input index (0-7):
   - Case 0 (Index 0): Requires number 795 (`$0x31b`) and character 'm' (`$0x6d`). Input: `0 m 795`
   - Case 1 (Index 1): Requires number 805 (`$0x325`) and character 'u' (`$0x75`). Input: `1 u 805`
   - Case 2 (Index 2): Requires number 374 (`$0x176`) and character 'n' (`$0x6e`). Input: `2 n 374`
   - Case 3 (Index 3): Requires number 919 (`$0x397`) and character 'c' (`$0x63`). Input: `3 c 919`
   - Case 4 (Index 4): Requires number 153 (`$0x99`) and character 's' (`$0x73`). Input: `4 s 153`
   - Case 5 (Index 5): Requires number 214 (`$0xd6`) and character 's' (`$0x73`). Input: `5 s 214`
   - Case 6 (Index 6): Requires number 467 (`$0x1d3`) and character 'l' (`$0x6c`). Input: `6 l 467`
   - Case 7 (Index 7): Requires number 220 (`$0xdc`) and character 'x' (`$0x78`). Input: `7 x 220`

   Providing any one of these input strings (e.g., `"0 m 795"`) will satisfy all checks and defuse Phase 3.

Phase 4

Annotated Assembly Dump

The annotated assembly dumps for `phase_4` and the helper function `func4` obtained via GDB are shown below.

`phase_4` **Assembly:**

```
Dump of assembler code for function phase_4:
   0x0000000000401151 <+0>:   sub    $0x18,%rsp # setup 24 bytes of space on stack
   0x0000000000401155 <+4>:   mov    %fs:0x28,%rax # stack canary
   0x000000000040115e <+13>:  mov    %rax,0x8(%rsp) # store canary on stack
   0x0000000000401163 <+18>:  xor    %eax,%eax # clear eax
   0x0000000000401165 <+20>:  lea    0x4(%rsp),%rcx # sets rcx to addr of second input
   0x000000000040116a <+25>:  mov    %rsp,%rdx # sets rdx to first addr of 1st input
   0x000000000040116d <+28>:  mov    $0x4029fd,%esi # format string for sscanf
   0x0000000000401172 <+33>:  call   0x400c40 <__isoc99_sscanf@plt>
   0x0000000000401177 <+38>:  cmp    $0x2,%eax # checks if 2 values were read
   0x000000000040117a <+41>:  jne    0x401182 <phase_4+49> # explode bomb if input != 2
   0x000000000040117c <+43>:  cmpl   $0xe,(%rsp) # compare if first num <= 14
```

```
   0x0000000000401180 <+47>:  jbe    0x401187 <phase_4+54> # skip explode if first num <= 14
   0x0000000000401182 <+49>:  call   0x401742 <explode_bomb>
   0x0000000000401187 <+54>:  mov    $0xe,%edx # sets third param to 14
   0x000000000040118c <+59>:  mov    $0x0,%esi # sets second param to 0
   0x0000000000401191 <+64>:  mov    (%rsp),%edi # set first param to first input
   0x0000000000401194 <+67>:  call   0x40111e <func4> # call func4(num1, 0, 14)
   0x0000000000401199 <+72>:  cmp    $0x1b,%eax # check if func4 returns 27
   0x000000000040119c <+75>:  jne    0x4011a5 <phase_4+84> # explode if func4 doesnt return 27
   0x000000000040119e <+77>:  cmpl   $0x1b,0x4(%rsp) # check is 2nd input is 27
   0x00000000004011a3 <+82>:  je     0x4011aa <phase_4+89> # skip explode if second input is 27
   0x00000000004011a5 <+84>:  call   0x401742 <explode_bomb> # explode bomb
   # clean up stack
   0x00000000004011aa <+89>:  mov    0x8(%rsp),%rax
   0x00000000004011af <+94>:  xor    %fs:0x28,%rax
   0x00000000004011b8 <+103>: je     0x4011bf <phase_4+110>
   0x00000000004011ba <+105>: call   0x400b90 <__stack_chk_fail@plt>
   0x00000000004011bf <+110>: add    $0x18,%rsp
   0x00000000004011c3 <+114>: ret
End of assembler dump.
```

`func4` **Assembly:**

```
Dump of assembler code for function func4:
   0x000000000040111e <+0>:   push   %rbx # save rbx reg
   0x000000000040111f <+1>:   mov    %edx,%eax # eax = edx (third param, 14)
   0x0000000000401121 <+3>:   sub    %esi,%eax # eax = edx - esi (14-0) = 14
   0x0000000000401123 <+5>:   mov    %eax,%ebx # ebx = eax (14)
   0x0000000000401125 <+7>:   shr    $0x1f,%ebx # shift ebx right by 31 (extract sign bit)
   0x0000000000401128 <+10>:  add    %ebx,%eax # eax = eax + ebx (if positive, 14+0; if negative, eax+sign) ->
   0x000000000040112a <+12>:  sar    %eax # arithmetic shift right eax by 1 (eax = eax / 2) -> (14 / 2 = 7)
   0x000000000040112c <+14>:  lea    (%rax,%rsi,1),%ebx # ebx = rax + rsi (mid + low) -> (7 + 0 = 7)
   0x000000000040112f <+17>:  cmp    %edi,%ebx # compare input (edi) with ebx (midpoint 7)
   0x0000000000401131 <+19>:  jle    0x40113f <func4+33> # if ebx <= input, jump to +33
   # Case: midpoint > input (edi)
   0x0000000000401133 <+21>:  lea    -0x1(%rbx),%edx # edx = rbx -1 (new high = mid - 1)
   0x0000000000401136 <+24>:  call   0x40111e <func4> # recursion. func4(input, low, mid-1)
   0x000000000040113b <+29>:  add    %ebx,%eax # eax = eax + ebx (recursive_result + mid)
   0x000000000040113d <+31>:  jmp    0x40114f <func4+49> # jmp to return
   # Case: midpoint <= input (edi) Jump target from <+19>
   0x000000000040113f <+33>:  mov    %ebx,%eax # Save mid value (ebx) in eax (potential return value)
   0x0000000000401141 <+35>:  cmp    %edi,%ebx # cmp input with ebx (midpoint 7)
   0x0000000000401143 <+37>:  jge    0x40114f <func4+49> # if ebx >= input (i.e., ebx == input), jmp -> return
   # Case: midpoint < input (edi)
   0x0000000000401145 <+39>:  lea    0x1(%rbx),%esi # esi = rbx + 1 (new low = mid + 1)
   0x0000000000401148 <+42>:  call   0x40111e <func4> # func4 (input, mid+1, high)
   0x000000000040114d <+47>:  add    %ebx,%eax # eax = eax + ebx (recursive_result + mid)
   # Return path
   0x000000000040114f <+49>:  pop    %rbx # restore rbx
   0x0000000000401150 <+50>:  ret # return
End of assembler dump.
```

Procedure

1. **Initial Analysis of `phase_4`:** I started by disassembling `phase_4` in GDB. The initial instructions set up the stack (`sub $0x18,%rsp`) and check for a canary (`mov %fs:0x28,%rax`).
2. **Input Reading:** The instruction `call <__isoc99_sscanf@plt>` at `<+33>` indicated that the phase reads input from the user. The arguments prepared before the call (`mov %rsp,%rdx`, `lea 0x4(%rsp),%rcx`, `mov $0x4029fd,%esi`) showed it expects two integers (`"%d %d"`) stored at `%rsp` (let's call this `num1`) and `0x4(%rsp)` (let's call this `num2`).
3. **Input Validation:** The code immediately checks if `sscanf` returned 2 (`cmp $0x2, %eax` at `<+38>`), meaning two integers must be provided. If not, the bomb explodes (`jne 0x401182 <phase_4+49>`).

4. **First Number Check:** The instruction `cmpl $0xe, (%rsp)` at `<+43>` compares the first input number (`num1`) with 14 (0xe). The following `jbe` means the phase proceeds only if `num1 <= 14`. Otherwise, the bomb explodes.

5. **Function Call `func4`:** The phase then calls `func4` at `<+67>`. The arguments are set up just before the call: `mov (%rsp),%edi` (sets `num1` as arg1), `mov $0x0,%esi` (sets 0 as arg2), `mov $0xe,%edx` (sets 14 as arg3). So the call is effectively `func4(num1, 0, 14)`.

6. **`func4` Return Value Check:** After `func4` returns, its result (in `%eax`) is compared with 27 (0x1b) using `cmp $0x1b, %eax` at `<+72>`. If the return value is not 27, the bomb explodes (`jne 0x4011a5 <phase_4+84>`).

7. **Second Number Check:** Finally, the second input number (`num2`, stored at `0x4(%rsp)`) is compared with 27 (0x1b) using `cmpl $0x1b, 0x4(%rsp)` at `<+77>`. If `num2` is not equal to 27, the bomb explodes.

8. **Summary of Conditions:** To pass Phase 4, we need to provide two integers, `num1` and `num2`, such that:

   - `num1 <= 14`
   - `func4(num1, 0, 14)` returns 27
   - `num2 == 27`

9. **Analysis of `func4`:** I disassembled `func4`. It's a recursive function. It calculates a midpoint (`lea (%rax,%rsi,1),%ebx`).

   - If `midpoint == input_val`, it returns the midpoint.
   - If `midpoint > input_val`, it recursively calls `func4(input, low, mid-1)` and returns `midpoint + recursive_result`.
   - If `midpoint < input_val`, it recursively calls `func4(input, mid+1, high)` and returns `midpoint + recursive_result`.

10. **Finding the Correct Input for `func4`:** The goal was to find `num1` (where `0 <= num1 <= 14`) such that `func4(num1, 0, 14)` returns 27. I manually traced the execution for several potential inputs:

    - `func4(7, 0, 14)` → mid=7. Returns 7.
    - `func4(1, 0, 14)` → mid=7. Recurse `func4(1, 0, 6)`. mid=3. Recurse `func4(1, 0, 2)`. mid=1. Return 1. Back: 3+1=4. Back: 7+4=11.
    - `func4(6, 0, 14)` → mid=7. Recurse `func4(6, 0, 6)`. mid=3. Recurse `func4(6, 4, 6)`. mid=5. Recurse `func4(6, 6, 6)`. mid=6. Return 6. Back: 5+6=11. Back: 3+11=14. Back: 7+14=21.
    - `func4(9, 0, 14)` → mid=7. Recurse `func4(9, 8, 14)`. mid=11. Recurse `func4(9, 8, 10)`. mid=9. Return 9. Back: 11+9=20. Back: 7+20=27.

11. **Solution:** The input `num1 = 9` results in `func4(9, 0, 14)` returning 27. Since `9 <= 14` and we also need `num2 = 27`, the final input string is `9 27`.

Phase 5

Annotated Assembly Dump

Here is the assembly code for `phase_5` obtained using GDB's `disas` command.

```
Dump of assembler code for function phase_5:
   0x00000000004011c4 <+0>:   push   rbx
   0x00000000004011c5 <+1>:   sub    rsp,0x10 # create 16 bytes on stack
   0x00000000004011c9 <+5>:   mov    rbx,rdi
   0x00000000004011cc <+8>:   mov    rax,QWORD PTR fs:0x28
   0x00000000004011d5 <+17>:  mov    QWORD PTR [rsp+0x8],rax # stack canary
   0x00000000004011da <+22>:  xor    eax,eax # zero out eax
   0x00000000004011dc <+24>:  call   0x401455 <string_length> # call some string length function
   0x00000000004011e1 <+29>:  cmp    eax,0x6 # string length must = 6
   0x00000000004011e4 <+32>:  je     0x4011eb <phase_5+39> # if str.len = 6 then jump to +39
   0x00000000004011e6 <+34>:  call   0x401742 <explode_bomb> # otherwise blow up bomb
   0x00000000004011eb <+39>:  mov    $0x0,%eax # move zero into eax
   # loop
   0x00000000004011f0 <+44>:  movzbl (%rbx,%rax,1),%edx # load charecter from input string
   0x00000000004011f4 <+48>:  and    $0xf,%edx # bitwise and, get last 4 bits
   0x00000000004011f7 <+51>:  movzbl 0x4027a0(%rdx),%edx # use result as index in table at 0x4027a0
   0x00000000004011fe <+58>:  mov    %dl,(%rsp,%rax,1) # store lookup char on stack
   0x0000000000401201 <+61>:  add    $0x1,%rax # increment loop counter
   0x0000000000401205 <+65>:  cmp    $0x6,%rax # check if we have done all 6 chars
   0x0000000000401209 <+69>:  jne    0x4011f0 <phase_5+44> # if not then jump to top of loop
   0x000000000040120b <+71>:  movb   $0x0,0x6(%rsp) # null terminate string?
   0x0000000000401210 <+76>:  mov    $0x402757,%esi # load addr of target string
   0x0000000000401215 <+81>:  mov    %rsp,%rdi # load addr of constructed string
   0x0000000000401218 <+84>:  call   0x401473 <strings_not_equal> # comapre string
   0x000000000040121d <+89>:  test   %eax,%eax # check func results
   0x000000000040121f <+91>:  je     0x401226 <phase_5+98> # skip explosion if strs are =
   0x0000000000401221 <+93>:  call   0x401742 <explode_bomb> # blow up
   # clean up stack
   0x0000000000401226 <+98>:  mov    0x8(%rsp),%rax
   0x000000000040122b <+103>: xor    %fs:0x28,%rax
   0x0000000000401234 <+112>: je     0x40123b <phase_5+119>
   0x0000000000401236 <+114>: call   0x400b90 <__stack_chk_fail@plt>
   0x000000000040123b <+119>: add    $0x10,%rsp
   0x000000000040123f <+123>: pop    %rbx
   0x0000000000401240 <+124>: ret
End of assembler dump.
```

Procedure for Solving Phase 5

1. **Analyze Initial Checks:** The first significant check (`<+29>` to `<+34>`) compares the length of the input string (obtained via `string_length`) to 6. If the length is not exactly 6, the bomb explodes immediately. Therefore, the input must be a 6-character string.

2. **Understand the Loop:** The code then enters a loop (`<+39>` to `<+69>`) that iterates 6 times (controlled by `rax` going from 0 to 5). Inside the loop: a. It retrieves one character from the input string (`<+44>`). b. It isolates the lower 4 bits of the character's ASCII value using a bitwise AND with \$0xf\$ (`<+48>`). This produces an index value between 0 and 15. c. It uses this index to access an element within a character array (lookup table) located at address `0x4027a0` (`<+51>`). Using GDB's `x/s` command on this address revealed the string starting with `"maduiersnfotvbyl..."`. The relevant first 16 characters are:

```
Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Char:  m a d u i e r s n f o  t  v  b  y  l
```

d. The character retrieved from this lookup table is stored sequentially in a buffer on the stack (`<+58>`).

3. **Analyze the Comparison:** After the loop finishes, a null terminator is added to the buffer on the stack (`<+71>`), forming a 6-character C string. This generated string is then compared (`<+84>`) to a

static string located at address `0x402757` (`<+76>`). GDB's `x/s 0x402757` command showed this target string to be `"oilers"`. If the generated string does not match `"oilers"`, the bomb explodes (`<+93>`).

4. **Reverse the Transformation:** To find the correct input, we must reverse engineer this process. We know the target output string is `"oilers"`. We need to find the indices in the lookup table (`"maduiersnfotvbyl..."`) that correspond to these characters:

- 'o' is at index 10 ($0xa$)
- 'i' is at index 4 ($0x4$)
- 'l' is at index 15 ($0xf$)
- 'e' is at index 5 ($0x5$)
- 'r' is at index 6 ($0x6$)
- 's' is at index 7 ($0x7$)

So, the lower 4 bits of the ASCII values of our six input characters must be, in order: $0xa, 0x4, 0xf, 0x5, 0x6, 0x7$.

5. **Construct the Input String:** We need to find 6 characters whose ASCII values satisfy the condition `ASCII_value & 0xf = required_index`. There are multiple characters that satisfy each condition. We can pick any valid combination. For example:

- Index $0xa$: `'j'` (ASCII $0x6a$), $0x6a \& 0xf = 0xa$
- Index $0x4$: `'4'` (ASCII $0x34$), $0x34 \& 0xf = 0x4$
- Index $0xf$: `'o'` (ASCII $0x6f$), $0x6f \& 0xf = 0xf$
- Index $0x5$: `'e'` (ASCII $0x65$), $0x65 \& 0xf = 0x5$
- Index $0x6$: `'f'` (ASCII $0x66$), $0x66 \& 0xf = 0x6$
- Index $0x7$: `'g'` (ASCII $0x67$), $0x67 \& 0xf = 0x7$

Combining these gives a valid input string: `"j4oefg"`. Other valid inputs exist, such as `"J4OEFG"` as only the lower 4 bits of each character matter for the transformation.

6. **Final Checks:** The phase concludes with a stack canary check (`<+98>` to `<+114>`) and standard function epilogue. Providing the correct 6-character string avoids the `explode_bomb` calls and passes the canary check, thus defusing the phase.

Phase 6

Annotated Assembly Dump

The following is the GDB disassembly dump for phase_6, displayed using the verbatim environment to preserve formatting.

```
; Function phase_6: Expects 6 unique integers (1-6) as input.
; Rearranges pointers to nodes based on input order and checks if node values are descending.

; stack stuff
0x0000000000401241 <+0>:    push   %r13        ; Save register r13
0x0000000000401243 <+2>:    push   %r12        ; Save register r12
0x0000000000401245 <+4>:    push   %rbp        ; Save register rbp
0x0000000000401246 <+5>:    push   %rbx        ; Save register rbx
0x0000000000401247 <+6>:    sub    $0x68,%rsp  ; Allocate 104 bytes on stack
0x000000000040124b <+10>:   mov    %fs:0x28,%rax   ; Get stack canary
0x0000000000401254 <+19>:   mov    %rax,0x58(%rsp) ; Store canary
0x0000000000401259 <+24>:   xor    %eax,%eax      ; Zero out eax

; --- Read Input ---
0x000000000040125b <+26>:   mov    %rsp,%rsi      ; Argument 2 for read_six_numbers: pointer to stack buffer (r
0x000000000040125e <+29>:   call   0x401778 <read_six_numbers> ; Reads 6 integers into [rsp] to [rsp+0x14]

; --- Input Validation Loop 1: Check Range (1-6) and Uniqueness ---
0x0000000000401263 <+34>:   mov    %rsp,%r12      ; r12 points to the start of the input numbers on the stack
0x0000000000401266 <+37>:   mov    $0x0,%r13d     ; r13d = 0 (Outer loop counter: 0 to 5)
; Outer loop starts (checks numbers at index r13d = 0 through 5)
0x000000000040126c <+43>:   mov    %r12,%rbp      ; rbp points to the current number being checked by the outer
0x000000000040126f <+46>:   mov    (%r12),%eax    ; eax = value of input[r13d]
0x0000000000401273 <+50>:   sub    $0x1,%eax      ; eax = input[r13d] - 1
0x0000000000401276 <+53>:   cmp    $0x5,%eax      ; Compare (input[r13d] - 1) with 5
0x0000000000401279 <+56>:   jbe    0x401280       ; Jump if <= 5 (means original number was 1 <= input[r13d] <=
0x000000000040127b <+58>:   call   0x401742 <explode_bomb> ; Explode if number is out of range [1, 6]

; Start inner loop (checks for duplicates against input[r13d])
0x0000000000401280 <+63>:   add    $0x1,%r13d     ; Increment outer loop counter (now represents count 1 to 6)
0x0000000000401284 <+67>:   cmp    $0x6,%r13d     ; processed all 6 numbers in outer loop yet?
0x0000000000401288 <+71>:   je     0x4012c7       ; If yes, all numbers validated (range+unique), jump to node s
0x000000000040128a <+73>:   mov    %r13d,%ebx     ; ebx = Inner loop counter, starting from index r13d (index 1
; Inner loop starts (checks input[r13d] against input[ebx] where ebx > r13d-1)
0x000000000040128d <+76>:   movslq %ebx,%rax      ; rax = 64-bit version of inner loop index ebx
0x0000000000401290 <+79>:   mov    (%rsp,%rax,4),%eax ; eax = value of input[ebx] (number at index ebx)
0x0000000000401293 <+82>:   cmp    %eax,0x0(%rbp) ; Compare input[ebx] with input[r13d-1] (pointed to by rbp)
0x0000000000401296 <+85>:   jne    0x40129d       ; Jump if they are different (not a duplicate)
0x0000000000401298 <+87>:   call   0x401742 <explode_bomb> ; Explode if input[ebx] == input[r13d-1] (duplicate i

; Increment inner loop and continue duplicate check
0x000000000040129d <+92>:   add    $0x1,%ebx      ; Increment inner loop index ebx
0x00000000004012a0 <+95>:   cmp    $0x5,%ebx      ; Have we checked against all subsequent numbers (up to index
0x00000000004012a3 <+98>:   jle    0x40128d       ; If ebx <= 5, loop back to check next inner index
; Inner loop finished for input[r13d-1]
0x00000000004012a5 <+100>:  add    $0x4,%r12      ; Move r12 to point to the next input number (input[r13d])
0x00000000004012a9 <+104>:  jmp    0x40126c       ; Jump back to start of outer loop for the next number

; --- Node Selection Loop (Using input numbers to select nodes) ---
; This section iterates through the validated input numbers
; For each input number N, it selects a node from the preset list
; The traversal code below appears to follow N-1 'next' links from 0x604300 which held me up for awhile
; However, this logic conflicts with the list structure for N=5 and N=6.
; The actual behavior is that input N selects the node identified as "Node N".
0x00000000004012c7 <+134>:  mov    $0x0,%esi      ; esi = 0 (Loop index for input array 0 to 5, used as offset r
; Node selection loop starts
0x00000000004012cc <+139>:  mov    (%rsp,%rsi,1),%ecx ; ecx = N = value of input[esi/4]
```

```
0x00000000004012cf <+142>:  mov    $0x1,%eax      ; eax = 1 (Counter for list traversal)
0x00000000004012d4 <+147>:  mov    $0x604300,%edx ; edx = Pointer to the head of the predefined linked list (No
0x00000000004012d9 <+152>:  cmp    $0x1,%ecx      ; Is the input number N == 1?
0x00000000004012dc <+155>:  jg     0x4012ab       ; If N > 1 jump to the traversal loop
0x00000000004012de <+157>:  jmp    0x4012b6       ; If N == 1 skip traversal and jump directly to store the hea

; List traversal sub-loop (Attempts to find Nth node)
0x00000000004012ab <+106>:  mov    0x8(%rdx),%rdx ; rdx = rdx->next (Move to next node pointer at offset 8)
0x00000000004012af <+110>:  add    $0x1,%eax      ; Increment traversal counter
0x00000000004012b2 <+113>:  cmp    %ecx,%eax      ; Compare N (ecx) with counter (eax)
0x00000000004012b4 <+115>:  jne    0x4012ab       ; If counter != N continue traversing

; Store the selected node pointer
0x00000000004012b6 <+117>:  ; rdx now holds the pointer to the effectively selected node for input N
                           mov    %rdx,0x20(%rsp,%rsi,2) ; Store node pointer rdx into array at rsp+0x20 + esi*
                           ; Array indices: (esi=0)->rsp+0x20, (esi=4)->rsp+0x28, ... (esi=20)->rsp+0x48
0x00000000004012bb <+122>:  add    $0x4,%rsi      ; Increment index offset esi by 4 for next input number
0x00000000004012bf <+126>:  cmp    $0x18,%rsi     ; Compare esi with 24 (0x18)
0x00000000004012c3 <+130>:  jne    0x4012cc       ; If esi != 24 (haven't processed all 6 inputs), loop back
0x00000000004012c5 <+132>:  jmp    0x4012e0       ; Finished selecting nodes, jump to re-linking phase

; --- Re-linking Phase ---
; Takes the 6 node pointers stored in the array at rsp+0x20 to rsp+0x48
; and links them according to the original input order.
0x00000000004012e0 <+159>:  mov    0x20(%rsp),%rbx ; rbx = pointer to first selected node ( input[0])
0x00000000004012e5 <+164>:  lea    0x20(%rsp),%rax ; rax = address of the start of the pointer array (rsp+0x20)
0x00000000004012ea <+169>:  lea    0x48(%rsp),%rsi ; rsi = address of the last pointer in the array (rsp+0x48)
0x00000000004012ef <+174>:  mov    %rbx,%rcx      ; rcx = pointer to current node being linked (starts with nod
; Re-linking loop
0x00000000004012f2 <+177>:  mov    0x8(%rax),%rdx ; rdx = pointer to next node from the array (node from input[
0x00000000004012f6 <+181>:  mov    %rdx,0x8(%rcx) ; Set (current node)->next = pointer to next node
0x00000000004012fa <+185>:  add    $0x8,%rax      ; Move rax to point to the next pointer slot in the array
0x00000000004012fe <+189>:  mov    %rdx,%rcx      ; Update current node for next iteration (rcx = next node)
0x0000000000401301 <+192>:  cmp    %rsi,%rax      ; Have we reached the address of the last pointer slot?
0x0000000000401304 <+195>:  jne    0x4012f2       ; If not, continue linking
; Finished loop, set last node's next pointer
0x0000000000401306 <+197>:  movq   $0x0,0x8(%rdx) ; Set (last node in sequence)->next = NULL

; --- Final Check: Descending Order Verification ---
; Iterates through the newly re-linked list (headed by rbx)
; and checks if the node values (at offset 0) are in descending order.
0x000000000040130e <+205>:  mov    $0x5,%ebp      ; ebp = 5 (Loop counter for 5 comparisons: 0-1, 1-2, ..., 4-5
; Verification loop starts
0x0000000000401313 <+210>:  mov    0x8(%rbx),%rax ; rax = current_node->next (pointer to the next node)
0x0000000000401317 <+214>:  mov    (%rax),%eax    ; eax = value of next node (dereference pointer rax, read fir
0x0000000000401319 <+216>:  cmp    %eax,(%rbx)    ; Compare next_node_value (eax) with current_node_value (at r
0x000000000040131b <+218>:  jge    0x401322       ; Jump if current_node_value >= next_node_value (descending o
0x000000000040131d <+220>:  call   0x401742 <explode_bomb> ; Explode if current_node_value < next_node_value (n

; Move to next node for comparison
0x0000000000401322 <+225>:  mov    0x8(%rbx),%rbx ; rbx = current_node->next (Move to the next node)
0x0000000000401326 <+229>:  sub    $0x1,%ebp      ; Decrement loop counter
0x0000000000401329 <+232>:  jne    0x401313       ; If counter != 0, loop back to compare next pair

; --- Phase Defused: Clean up and Return ---
; Stack Canary Check
0x000000000040132b <+234>:  mov    0x58(%rsp),%rax ; rax = canary value read from stack
0x0000000000401330 <+239>:  xor    %fs:0x28,%rax   ; Compare with original canary value. Result is 0 if they ma
0x0000000000401339 <+248>:  je     0x401340        ; Jump if canary is intact
0x000000000040133b <+250>:  call   0x400b90 <__stack_chk_fail@plt> ; Canary check failed, abort.

; Restore Stack and Registers
0x0000000000401340 <+255>:  add    $0x68,%rsp     ; Deallocate stack frame
```

```
0x0000000000401344 <+259>:  pop    %rbx           ; Restore rbx
0x0000000000401345 <+260>:  pop    %rbp           ; Restore rbp
0x0000000000401346 <+261>:  pop    %r12           ; Restore r12
0x0000000000401348 <+263>:  pop    %r13           ; Restore r13
0x000000000040134a <+265>:  ret                   ; Return from function
```

Procedure Used to Solve Phase 6

Phase 6 involved understanding linked list manipulation and ordering based on node values. Here's the procedure followed:

1. **Initial Analysis:** The phase begins by calling `read_six_numbers` (at `0x40125e`), indicating it expects six integer inputs.

2. **Input Validation:** The code block from `0x40126c` to `0x4012a9` performs two crucial checks on the input numbers stored on the stack:

   - **Range Check:** Each number N is checked (`sub $0x1`, `cmp $0x5`, `jbe`) to ensure $1 \leq N \leq 6$. If any number is outside this range, the bomb explodes.
   - **Uniqueness Check:** A nested loop compares each number (`input[outer]`) against all subsequent numbers (`input[inner]`) using `cmp %eax, 0x0(%rbp)` at `0x401293`. If any two numbers are identical (`jne` doesn't jump), the bomb explodes.
   - *Conclusion 1:* The input must be a permutation of the integers 1, 2, 3, 4, 5, 6.

3. **Linked List Identification:** The code block starting at `0x4012c7` iterates through the six validated input numbers. Inside the loop, the address `0x604300` is loaded into `%edx` (`0x4012d4`), strongly suggesting the start of a predefined linked list.

4. **Node Selection Mechanism:** The code then uses the input number N (in `%ecx`) to select a node from this list. The assembly includes a traversal loop (`0x4012ab` to `0x4012b4`) that *appears* to follow $N-1$ `next` pointers (stored at offset 8) from the head (`0x604300`). The pointer to the selected node (`%rdx`) is stored in a temporary array on the stack (`mov %rdx,0x20(%rsp,%rsi,2)` at `0x4012b6`).

5. **Examining the List Structure (GDB):** Using GDB (`x/24wx 0x604300`), the structure of the linked list was examined:

   ```
   0x604300 <node1>: 0x000003a4 0x00000001 0x00604350 0x00000000 (Val: 932, Next: Node6)
   0x604310 <node2>: 0x000000ba 0x00000002 0x00604330 0x00000000 (Val: 186, Next: Node4)
   0x604320 <node3>: 0x000000ad 0x00000003 0x00604340 0x00000000 (Val: 173, Next: Node5)
   0x604330 <node4>: 0x0000025e 0x00000004 0x00000000 0x00000000 (Val: 606, Next: NULL)
   0x604340 <node5>: 0x000002a2 0x00000005 0x00604300 0x00000000 (Val: 674, Next: Node1)
   0x604350 <node6>: 0x00000210 0x00000006 0x00604310 0x00000000 (Val: 528, Next: Node2)
   ```

   Node values (at offset 0) were identified: N1=932, N2=186, N3=173, N4=606, N5=674, N6=528.

6. **Resolving the Contradiction:** Tracing the apparent traversal logic with the actual list structure revealed that inputs 5 and 6 would lead to errors (selecting NULL or crashing). Since the validation requires using inputs 1-6, a hypothesis was formed: the *intended* behavior is that **input N directly selects the node identified as Node N**, likely using the node's address or the sequence number stored at offset 4.

7. **Re-linking Analysis:** The code from `0x4012e0` to `0x401306` takes the six selected node pointers stored in the stack array and modifies their `next` pointers (`mov %rdx,0x8(%rcx)` at `0x4012f6`). It links them sequentially according to the order of the *original input numbers.* The `next` pointer of the last node in the sequence is set to NULL.

8. **Final Check Identification:** The final loop (`0x40130e` to `0x401329`) iterates through this newly created linked list. It compares the value of the current node (at offset 0, read via `(%rbx)`) with the value of the next node (at offset 0, read via `(%rax)`). The comparison `cmp %eax,(%rbx)` followed by `jge` (`0x40131b`) means the bomb only proceeds if `current_node_value >= next_node_value`.

- *Conclusion 2:* The values of the nodes, when ordered according to the input sequence, must be in **descending order**.

9. **Deriving the Solution:** To satisfy Conclusion 2, the input sequence must correspond to the order of Node N identifiers whose values are sorted descendingly:

   - Sort node values: 932, 674, 606, 528, 186, 173
   - Identify corresponding Node N identifiers (based on the hypothesis from step 6): Node 1, Node 5, Node 4, Node 6, Node 2, Node 3
   - Required input sequence: `1 5 4 6 2 3`

10. **Verification:** This input sequence (`1 5 4 6 2 3`) uses unique numbers between 1 and 6 (satisfying Conclusion 1) and arranges the node values (932, 674, 606, 528, 186, 173) in descending order (satisfying Conclusion 2). This sequence successfully defused Phase 6.