Phase 2

```
0x5555555555cf <phase_2+4>      push   %rbx
0x5555555555d0 <phase_2+5>      sub    $0x20,%rsp      <Prepare for call? second argument is a pointer to stack?
0x5555555555d4 <phase_2+9>      mov    %rsp,%rsi
0x5555555555d7 <phase_2+12>     call   0x555555555ac9 <read_six_numbers>
0x5555555555dc <phase_2+17>     cmpl   $0x0,(%rsp)                    seems to save the six numbers in the stack
0x5555555555e0 <phase_2+21>     js     0x5555555555e9 <phase_2+30>
0x5555555555e2 <phase_2+23>     mov    $0x1,%ebx
0x5555555555e7 <phase_2+28>     jmp    0x5555555555f8 <phase_2+45>
0x5555555555e9 <phase_2+30>     call   0x555555555a9d <explode_bomb>
0x5555555555ee <phase_2+35>     jmp    0x5555555555e2 <phase_2+23>  <Pointless instruction?
0x5555555555f0 <phase_2+37>     call   0x555555555a9d <explode_bomb>harder to defuse. */
0x5555555555f5 <phase_2+42>     add    $0x1,%ebx
0x5555555555f8 <phase_2+45>     cmp    $0x5,%ebx
0x5555555555fb <phase_2+48>     jg     0x555555555613 <phase_2+72>              th\n");
0x5555555555fd <phase_2+50>     movslq %ebx,%rax
0x555555555600 <phase_2+53>     lea    -0x1(%rbx),%edx
0x555555555603 <phase_2+56>     movslq %edx,%rdx          e than one phase! */
0x555555555606 <phase_2+59>     mov    (%rsp,%rdx,4),%edx
0x555555555609 <phase_2+62>     lea    (%rbx,%rdx,2),%edx
0x55555555560c <phase_2+65>     cmp    %edx,(%rsp,%rax,4)
0x55555555560f <phase_2+68>     je     0x5555555555f5 <phase_2+42>
0x555555555611 <phase_2+70>     jmp    0x5555555555f0 <phase_2+37>
0x555555555613 <phase_2+72>     add    $0x20,%rsp
0x555555555617 <phase_2+76>     pop    %rbx
0x555555555618 <phase_2+77>     ret
```

1 Preserve %rbx for caller function
2 %rsp ← %rsp-0x20 decrement stack pointer by 0x20
3 %rsi ← %rsp copy stack ptr to %rsi
4 Call read_six_numbers function
5 Set condition codes according to (%rsp) - 0
18 if (%rsp) - 0 is negative, jump to explode
19 %ebx ← 0x1, set rest of bits to zero
20 Direct jump

Interpretation:
Allocating memory for user entered numbers
Setting second argument for read_six_numbers function
read_six_numbers parses input string and saves each number to a different spot in the stack
Check if first # entered is 0.
If not zero, explode.
Set %ebx to 1
Jump to loop section

Stack Contents:

%rsp	
+0	first # entered
+4	2nd # entered
+8	3rd # entered
+0xc	4th # entered
+0x10	5th # entered
+0x14	6th # entered

Loop section:
%ebx seems to be a counter that starts at 1.
From 2nd iteration onwards, increment counter by 1 at beginning of loop

21 Set cc for %ebx - 0x5
22 if %ebx > 0x5, jump to end loop
23 %rax ← %ebx, change upper bit to sign bit
24 %ebx ← %rbx - 1
25 %rdx ← %ebx, change upper bit to sign bit
26 %ebx ← M[%rsp + 4*%rdx]
27 %edx ← %rbx + 2*%rdx
28 Set CC for M[%rsp + 4*%rax] - %edx
29 If zero condition true, loop back to 4c
30 Jump to bomb explode
31 Increment stack ptr by 0x20
32 Replenish %rbx for caller function

If %ebx > 5, jump to end loop
Save counter in %rax
Save counter - 1 in %edx
Save M[%rsp + (counter - 1)*4] in %rdx
Save counter + 2*%rdx in %rdx
If %edx is equal to *(%rsp + 4*counter) repeat loop.
Else explode bomb
Deallocate memory

Counter	Pointer offset	Number entered
1	+0	First
2	+4	second
3	+8	third
4	+12	fourth
5	+16	fifth
6	+20	sixth

Hand calculation of correct values

First number must be 0
Subsequent numbers must satisfy the expression
Counter + 2 * prevNumber = number

Second number:
$1 + 2 * 0 = number \Rightarrow number = 1$

Third number:
$2 + 2 * 1 = number \Rightarrow number = 4$

Fourth number:
$3 + 2 * 4 = number \Rightarrow number = 11$

Fifth number:
$4 + 2 * 11 = number \Rightarrow number = 26$

sixth number:
$5 + 2 * 26 = number \Rightarrow number = 57$

Solved!
User must enter "0 1 4 11 26 57"

Phase 3

```
Dump of assembler code for function phase_3:
   0x0000555555555619 <+0>:     endbr64
   0x000055555555561d <+4>:     sub    $0x18,%rsp
   0x0000555555555621 <+8>:     lea    0x8(%rsp),%rcx
   0x0000555555555626 <+13>:    lea    0xc(%rsp),%rdx
   0x000055555555562b <+18>:    lea    0x1d1d(%rip),%rsi        # 0x555555557348
   0x0000555555555632 <+25>:    mov    $0x0,%eax
   0x0000555555555637 <+30>:    call   0x5555555552c0 <__isoc99_sscanf@plt>
   0x000055555555563c <+35>:    cmp    $0x1,%eax
   0x000055555555563f <+38>:    jle    0x555555555665d <phase_3+68>
   0x0000555555555641 <+40>:    mov    0xc(%rsp),%eax
   0x0000555555555645 <+44>:    cmp    $0x7,%eax
   0x0000555555555648 <+47>:    ja     0x555555555687 <phase_3+110>
   0x000055555555564a <+49>:    mov    %eax,%eax
   0x000055555555564c <+51>:    lea    0x1b8d(%rip),%rdx        # 0x5555555571e0
   0x0000555555555653 <+58>:    movslq (%rdx,%rax,4),%rax
   0x0000555555555657 <+62>:    add    %rdx,%rax
   0x000055555555565a <+65>:    notrack jmp *%rax
   0x000055555555565d <+68>:    call   0x555555555a9d <explode_bomb>
   0x0000555555555662 <+73>:    jmp    0x555555555641 <phase_3+40>
   0x0000555555555664 <+75>:    mov    $0x299,%eax
   0x0000555555555669 <+80>:    jmp    0x555555555698 <phase_3+127>
   0x000055555555566b <+82>:    mov    $0x8d,%eax
   0x0000555555555670 <phase_3+87>:     jmp    0x555555555698 <phase_3+127>
   0x0000555555555672 <phase_3+89>:     mov    $0xfffffe26,%eax
   0x0000555555555677 <phase_3+94>:     jmp    0x555555555698 <phase_3+127>
   0x0000555555555679 <phase_3+96>:     mov    $0x61,%eax
   0x000055555555567e <phase_3+101>:    jmp    0x555555555698 <phase_3+127>
   0x0000555555555680 <phase_3+103>:    mov    $0x2e2,%eax
   0x0000555555555685 <phase_3+108>:    jmp    0x555555555698 <phase_3+127>
   0x0000555555555687 <phase_3+110>:    call   0x555555555a9d <explode_bomb>
   0x000055555555568c <phase_3+115>:    mov    $0x0,%eax
   0x0000555555555691 <phase_3+120>:    jmp    0x555555555698 <phase_3+127>
   0x0000555555555693 <phase_3+122>:    mov    $0xfffffd1e,%eax
   0x0000555555555698 <phase_3+127>:    cmpl   $0x4,0xc(%rsp)
   0x000055555555569d <phase_3+132>:    jg     0x5555555556a5 <phase_3+140>
   0x000055555555569f <phase_3+134>:    cmp    %eax,0x8(%rsp)
   0x00005555555556a3 <phase_3+138>:    je     0x5555555556aa <phase_3+145>
   0x00005555555556a5 <phase_3+140>:    call   0x555555555a9d <explode_bomb>
   0x00005555555556aa <phase_3+145>:    add    $0x18,%rsp
   0x00005555555556ae <phase_3+149>:    ret
```

input string in %rdi

1 decrement stack ptr by 0x18
2 %rcx ← %rsp + 0x8
3 %rdx ← %rsp + 0xC
4 %rsi ← %rip + 0x1d1d
5 %eax ← 0x0
6 Load something to stack?
7 Set CC for %eax − 0x1 expression
8 Jump if %eax ≤ 0x1, explode
9 %eax ← M[%rsp + 0xc]
10 Set CC for %eax − 0x7 expression
11 Jump if %eax > 0x7
12 %eax ← %eax, (unsigned) set upper 32 bits to 0?
13 %rdx ← %rip + 0x1b8d
14 move 32 bits, %rax ← M[%rdx + 4 * %rax]
15 %rax ← %rax + %rdx
16 Jump to (%rax), notrack???

Let this address be A (2nd number)
Let this address be B (1st number)
instruction ptr
Contents of B in %eax
resulting address
instruction ptr
(unsigned)

Interpretation

Allocating memory
Sets fourth argument to an address
Sets third argument to an address
Sets second argument to an address
Call this function with string and three address parameters
It seems that function returns amount of numbers

If user entered 1 or less numbers, explode
Copy contents in address B to %eax (Containing first # entered)

If contents of B > 7, explode First # must be less than or equal to 7

Redefine B to new address (takes next instruction address plus 0x1b8d)
Store contents of new address + 4 * 1st number entered in %rax
Increment contents by previous address calculated
(this whole process sets jump target based on the first # entered)
Seems to be an overflow scam which ends up with a jump
forget inside the function code

Lands when 1st # is 0
Lands when 1st # is 2
Lands when 1st # is 3
Lands when 1st # is 4
Lands when 1st # is 6

found jump locations for each number using debugger

Depending on where the jump lands, the second number must be set to prevent explosion

Lands when 1st # is 5 or 1 or 7

2nd # 6 65
2nd # 141
2nd # 4294966822
2nd # 97
2nd # 738
2nd # 0
2nd # 4294966558
First # must be ≤ 5
2nd # must match switch options address

Solved! Must enter "0 665"
or "1 4294966558"
or "2 141
or "3 4294966822"
or "4 97"

Phase 4

```
Dump of assembler code for function phase_4:
   0x00005555555556cd <+0>:     endbr64
   0x00005555555556d1 <+4>:     sub    $0x18,%rsp
   0x00005555555556d5 <+8>:     lea    0x8(%rsp),%rcx
   0x00005555555556da <+13>:    lea    0xc(%rsp),%rdx
   0x00005555555556df <+18>:    lea    0x1c69(%rip),%rsi        # 0x555555557349
   0x00005555555556e6 <+25>:    mov    $0x0,%eax
   0x00005555555556eb <+30>:    call   0x5555555552c0 <__isoc99_sscanf@plt>
   0x00005555555556f0 <+35>:    cmp    $0x2,%eax
   0x00005555555556f3 <+38>:    jne    0x5555555556fc <phase_4+47>
   0x00005555555556f5 <+40>:    cmpl   $0x9,0x8(%rsp)
   0x00005555555556fa <+45>:    jle    0x5555555557701 <phase_4+52>
   0x00005555555556fc <+47>:    call   0x555555555a9d <explode_bomb>
   0x0000555555555701 <+52>:    mov    0x8(%rsp),%esi
   0x0000555555555705 <+56>:    mov    0xc(%rsp),%edi
   0x0000555555555709 <+60>:    call   0x5555555556af <func4>
   0x000055555555570e <+65>:    cmp    $0xa0,%eax
   0x0000555555555713 <+70>:    jne    0x555555555571a <phase_4+77>
   0x0000555555555715 <+72>:    add    $0x18,%rsp
   0x0000555555555719 <+76>:    ret
   0x000055555555571a <+77>:    call   0x555555555a9d <explode_bomb>
   0x000055555555571f <+82>:    jmp    0x555555555715 <phase_4+72>
```

Never gets executed...

1. %rsp ← %rsp - 18
2. %rcx ← %rsp + 8
3. %rdx ← %rsp + 0xc
4. %rsi ← %rip + 0x1c69
5. %eax ← 0
6. Save input numbers to stack, returns amount of numbers inputted
7. Set CC for %eax - 0x2
8. If zero flag not set, explode bomb
9. Set CC for M[%rsp + 0x8] - 0x9
10. If M[%rsp + 0x8] < 10, jump next instruction
11. %esi ← M[%rsp + 0x8]
12. %esi ← M[%rsp + 0xc]
13. Call func4
14. Set CC for %eax - 0xa0
15. If %eax is not equal to 0xa0 jump to explode
16. %rsp ← %rsp + 18

Interpretation

Second number in %rsp + 0x8
First number in %rsp + 0xc
Allocate memory

> Found with debugger

Parse string, save each entered number in the stack
If the amount of numbers entered isn't 2, explode

If second number entered is greater than 9, explode

Save first # entered in %esi
Save second # entered in %edi

> func4 parameters

If value returned from func4 is not 0xa0, explode

Deallocate memory

```
0x5555555556af <func4>          endbr64
0x5555555556b3 <func4+4>        test   %edi,%edi
0x5555555556b5 <func4+6>        jle    0x5555555556ca <func4+27>
0x5555555556b7 <func4+8>        sub    $0x8,%rsp
0x5555555556bb <func4+12>       add    %esi,%esi
0x5555555556bd <func4+14>       sub    $0x1,%edi
0x5555555556c0 <func4+17>       call   0x5555555556af <func4>
0x5555555556c5 <func4+22>       add    $0x8,%rsp
0x5555555556c9 <func4+26>       ret
0x5555555556ca <func4+27>       mov    %esi,%eax
0x5555555556cc <func4+29>       ret
```

1. Set CC for %edi & %edi
2. If %edi is negative or 0 jump to 27
3. %rsp ← %rsp - 8
4. %esi ← %esi * 2
5. %edi ← %edi - 1
6. Recursive call
7. %rsp ← %rsp + 8
8. Deallocate memory
9. %eax ← %esi

If first parameter is negative or zero, jump to 27 } Base case

Allocate memory

Double the second parameter's value

Decrement first parameter's value

Recursion occurs

Deallocate memory

Base case return

In the func4 function, the first parameter serves as a counter and determines if the recursive case occurs. Every time the recursive case occurs, the second parameter value is doubled, the counter decrements, and the function calls itself with the new counter and second parameter values.

Solution:

First number entered can be any number.
First number represents the amount of times the second number will be doubled.
Second number must be less than 10.
Final value for second number must be 0xa0 = 160.

Thus,

$$secondnumber * 2^{firstnumber} = 160$$

Results table:

First number value

Second number value	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	1	2	4	8	16	32	64	128	256
2	2	4	8	16	32	64	128	256	512
3	3	6	12	24	48	96	192	384	768
4	4	8	16	32	64	128	256	512	1024
5	5	10	20	40	80	160	320	640	1280
6	6	12	24	48	96	192	384	768	1536
7	7	14	28	56	112	224	448	896	1792
8	8	16	32	64	128	256	512	1024	2048
9	9	18	36	72	144	288	576	1152	2304

Only one possible combination yields the correct result
first number = 5, second number = 5

Phase 5

```
Dump of assembler code for function phase_5:
   0x0000555555555721 <+0>:     endbr64
   0x0000555555555725 <+4>:     push   %rbx
   0x0000555555555726 <+5>:     mov    %rdi,%rbx
   0x0000555555555729 <+8>:     call   0x55555555598a <string_length>
   0x000055555555572e <+13>:    cmp    $0x6,%eax
   0x0000555555555731 <+16>:    jne    0x55555555575b <phase_5+58>
   0x0000555555555733 <+18>:    mov    $0x0,%ecx
   0x0000555555555738 <+23>:    mov    $0x0,%eax
   0x000055555555573d <+28>:    cmp    $0x5,%eax
   0x0000555555555740 <+31>:    jg     0x555555555762 <phase_5+65>
   0x0000555555555742 <+33>:    movslq %eax,%rdx
   0x0000555555555745 <+36>:    movzbl (%rbx,%rdx,1),%edx
   0x0000555555555749 <+40>:    and    $0xf,%edx
   0x000055555555574c <+43>:    lea    0x1aad(%rip),%rsi        # 0x555555557200 <array.3468>
   0x0000555555555753 <+50>:    add    (%rsi,%rdx,4),%ecx
   0x0000555555555756 <+53>:    add    $0x1,%eax
   0x0000555555555759 <+56>:    jmp    0x55555555573d <phase_5+28>
   0x000055555555575b <+58>:    call   0x555555555a9d <explode_bomb>
   0x0000555555555760 <+63>:    jmp    0x555555555733 <phase_5+18>
   0x0000555555555762 <+65>:    cmp    $0x3c,%ecx
   0x0000555555555765 <+68>:    jne    0x555555555769 <phase_5+72>
   0x0000555555555767 <+70>:    pop    %rbx
   0x555555555769 <phase_5+72>     call   0x555555555a9d <explode_bomb>
   0x55555555576e <phase_5+77>     jmp    0x555555555767 <phase_5+70>
```

never executed?

never executed?

Interpretation

1. Preserve %rbx for caller func
2. %rbx ← %rdi
3. Call string_length function
4. Set CC for %eax - 0x6 expression
5. If %eax is not 6, jump to explode
6. %ecx ← 0x0
7. %eax ← 0x0
8. Set CC for %eax - 0x5 expression
9. If %eax > 0x5, jump to 65
10. %rdx ← %eax, sign extend
11. %edx ← M[%rbx + %rdx] set upper to 0
12. %edx ← %edx & 0xf
13. %rsi ← M[%rip + 0x1aad]
14. %ecx ← M[%rsi + 4*%rdx]
15. %eax ← %eax + 1
16. Jump to 28
17. Set CC for %eax - 0x3c
18. If zero flag not set, jump to explode
19. Restore rbx for caller function

Save input string in callee saved register
address to user input string in %rbx
Counts amount of characters in input string. If not 6, explode.

Set %ecx to zero.

Set %eax to zero.

Loop:

Loop will iterate as long as %ecx < 6
Line 15 increments %eax at the end of the loop.
%eax is a counter.

Accesses input string like an array of char sets %edx to the char located in input string[counter]

Bit wise AND between the current character and 0xf, save in %edx.

Calculate address of array, save to %rsi

Access array with %rdx serving as index
Store value of array[index] in %ecx, which is serving as an accumulator

Increment counter

Restart Loop

If accumulator is not equal to 0x3c after the loop finishes, explode

Solution

To find the correct characters for each loop iteration that will let accumulator have correct end value, we must use debugger to find values of array elements

accumulator final value must be 0x3c
Note that 0x0a * 6 = 0x3c
And 0x0a is located in index number 8 of the array (because 4*8 = 32 = 0x20)

Thus we must find an ASCII character whose integer value yields 8 as a result of the bitwise operation (character & 0xf)
Note that 0xf = 0b1111 & 0b1000 = 0b1000 = 8
Thus we must find a character with 1000 in the last four least significant bits.

The character '(' has a binary value of 0b00101000, which satisfies this requirement.

Array values:

Byte offset	Value
+0x4	0x09
+0x8	0x0b
+0xc	0x01
+0x10	0x07
+0x14	0x02
+0x18	0x0f
+0x1c	0x06
+0x20	0x0a
+0x24	0x0c
+0x28	0x0e
+0x2c	0x08
+0x30	0x04
+0x34	0x0d
+0x38	0x10
+0x3c	0x05
+0x40	Out of array

Solved! User must enter "((((((" Or any character that has 1000 as the last four significant bits.
Other combinations are possible as long as the accumulator finishes with the correct value.

Phase 6

```
Dump of assembler code for function phase_6:
   0x0000555555555770 <+0>:     endbr64
   0x0000555555555774 <+4>:     push   %r12
   0x0000555555555776 <+6>:     push   %rbp
   0x0000555555555777 <+7>:     push   %rbx
   0x0000555555555778 <+8>:     sub    $0x50,%rsp
   0x000055555555577c <+12>:    lea    0x30(%rsp),%rsi
   0x0000555555555781 <+17>:    call   0x555555555ac9 <read_six_numbers>
   0x0000555555555786 <+22>:    mov    $0x0,%ebp
   0x000055555555578b <+27>:    jmp    0x5555555557b6 <phase_6+70>
   0x000055555555578d <+29>:    call   0x555555555a9d <explode_bomb>
   0x0000555555555792 <+34>:    jmp    0x5555555557ca <phase_6+90>
   0x0000555555555794 <+36>:    add    $0x1,%ebx
   0x0000555555555797 <+39>:    cmp    $0x5,%ebx
   0x000055555555579a <+42>:    jg     0x5555555557b3 <phase_6+67>
   0x000055555555579c <+44>:    movslq %ebp,%rax
   0x000055555555579f <+47>:    movslq %ebx,%rdx
   0x00005555555557a2 <+50>:    mov    0x30(%rsp,%rdx,4),%edi
   0x00005555555557a6 <+54>:    cmp    %edi,0x30(%rsp,%rax,4)
   0x00005555555557aa <+58>:    jne    0x555555555794 <phase_6+36>
   0x00005555555557ac <+60>:    call   0x555555555a9d <explode_bomb>
   0x00005555555557b1 <+65>:    jmp    0x555555555794 <phase_6+36>
   0x00005555555557b3 <+67>:    mov    %r12d,%ebp
   0x00005555555557b6 <+70>:    cmp    $0x5,%ebp
   0x00005555555557b9 <+73>:    jg     0x5555555557d3 <phase_6+99>
   0x00005555555557bb <+75>:    movslq %ebp,%rax
   0x00005555555557be <+78>:    mov    0x30(%rsp,%rax,4),%eax
   0x00005555555557c2 <+82>:    sub    $0x1,%eax
   0x00005555555557c5 <+85>:    cmp    $0x5,%eax
   0x00005555555557c8 <+88>:    ja     0x55555555578d <phase_6+29>
   0x00005555555557ca <+90>:    lea    0x1(%rbp),%r12d
   0x00005555555557ce <+94>:    mov    %r12d,%ebx
   0x00005555555557d1 <+97>:    jmp    0x555555555797 <phase_6+39>
   0x00005555555557d3 <+99>:    mov    $0x0,%esi
   0x00005555555557d8 <+104>:   jmp    0x5555555557e1 <phase_6+113>
   0x00005555555557da <+106>:   mov    %rdx,(%rsp,%rcx,8)
   0x00005555555557de <+110>:   add    $0x1,%esi
   0x00005555555557e1 <+113>:   cmp    $0x5,%esi
   0x00005555555557e4 <+116>:   jg     0x555555555804 <phase_6+148>
   0x00005555555557e6 <+118>:   mov    $0x1,%eax
   0x00005555555557eb <+123>:   lea    0x3a1e(%rip),%rdx        # 0x555555559210 <node1>
   0x00005555555557f2 <+130>:   movslq %esi,%rcx
   0x00005555555557f5 <+133>:   cmp    %eax,0x30(%rsp,%rcx,4)
   0x00005555555557f9 <+137>:   jle    0x5555555557da <phase_6+106>
   0x00005555555557fb <+139>:   mov    0x8(%rdx),%rdx
   0x00005555555557ff <+143>:   add    $0x1,%eax
   0x0000555555555802 <+146>:   jmp    0x5555555557f2 <phase_6+130>
   0x0000555555555804 <+148>:   mov    (%rsp),%rbx
   0x0000555555555808 <+152>:   mov    %rbx,%rcx
   0x000055555555580b <+155>:   mov    $0x1,%eax
   0x0000555555555810 <+160>:   jmp    0x555555555823 <phase_6+179>
   0x0000555555555812 <+162>:   movslq %eax,%rdx
   0x0000555555555815 <+165>:   mov    (%rsp,%rdx,8),%rdx
   0x0000555555555819 <+169>:   mov    %rdx,0x8(%rcx)
   0x000055555555581d <+173>:   add    $0x1,%eax
   0x0000555555555820 <+176>:   mov    %rdx,%rcx
   0x0000555555555823 <+179>:   cmp    $0x5,%eax
   0x0000555555555826 <+182>:   jle    0x555555555812 <phase_6+162>
   0x0000555555555828 <+184>:   movq   $0x0,0x8(%rcx)
   0x0000555555555830 <+192>:   mov    $0x0,%ebp
   0x0000555555555835 <+197>:   jmp    0x55555555583e <phase_6+206>
   0x0000555555555837 <+199>:   mov    0x8(%rbx),%rbx
```

Interpretation

1. Save contents of %r12 to stack
2. Save contents of %rbp to stack
3. Save contents of %rbx to stack
→ Preserve register content for caller function
4. %rsp ← %rsp - 0x50 → Allocate memory
5. %rsi ← %rsp + 0x30
6. Call read_six_numbers function → Parse string, store entered numbers in stack
7. %ebp ← 0
8. Jump to line 20
→ Set %ebp to 0. Counter

25. %ebx ← %ebx + 1 → Increment counter2 at each iteration (counter2)
18. Set CC for %ebx - 5
19. If %ebx > 5, jump to 67 — Potential split
→ Loop repeats as long as %ebx < 6. Counter2 starts at counter + 1
20. %rax ← %ebp, sign extend
21. %rdx ← %ebx, sign extend
→ Copy counter to %rax. Copy counter2 to %rdx
22. %edi ← M[%rsp + 4 * %rdx + 30]
23. Set CC for M[%rsp + 4 * %rax + 30] - %edi
24. If both are not equal, jump to 36, else explode
→ Checks if M[%rsp + 4 * counter + 48] is equal to M[%rsp + 4 * counter2 + 48]. If not equal, repeat loop. Must never be equal

20. %ebp ← %r12d → Increment counter
9. Set CC for %ebp - 0x5
10. If %ebp > 5, jump to 99 — Potential split
→ As long as counter hasn't reached 6, loop repeats
11. %rax ← %ebp, sign extend
12. %eax ← M[%rsp + 4 * %rax + 30]
→ %ebp determines which memory address is accessed, value stored in %eax
13. %eax ← %eax - 1
14. Set CC for %eax - 5
15. If %eax > 5, jump to explode
→ Any value read from memory must be between 1 – 6
16. %r12d ← %rbp + 1
17. %ebx ← %r12d
→ %ebx will be counter + 1
18. Jump to 39

11. %esi ← 0
12. Jump to 113
→ Set %esi to 0. Counter3
20. M[%rsp + 8 * %rcx] ← %rdx → Save current node address in M[%rsp + 8 * Counter3?] (%rcx)
21. %esi ← %esi + 1 → Increment Counter3
13. Set CC for %esi - 5
14. If %esi > 5, jump to 148 — Potential split
→ Repeat loop if counter3 < 6
→ Five node addresses will copy to stack array starting at %rsp
15. %eax ← 1 → Set %eax to 1. Counter4
16. %rdx ← %rip + 0x3a1e → Save address of first node in %rdx
17. %rcx ← %esi → Copy Counter3 to %rcx
18. Set CC for M[%rsp + 4 * rcx + 30] - %eax
19. If M[%rsp + 4 * rcx + 30] <= %eax jump to 106
→ If M[%rsp + 4 * Counter3 + 48] <= counter4, jump. Number entered position = counter3
20. %rdx ← M[%rdx + 8] → Copy address of next node to %rdx
21. %eax ← %eax + 1 → Increment counter4
22. Jump to 130 → Loop back
→ As long as number entered > counter4, change to next node

→ Node addresses will be copied to stack. Order of addresses will depend on numbers entered.

15. %rbx ← M[%rsp]
16. %rcx ← %rbx
→ Copy contents of M[%rsp] to %rbx and then %rcx
17. %eax ← 1 → Set %eax to 1. Counter5
18. Jump to 179
→ Next section starts with registers %rbx and %rcx set to the address of the first node copied to %rsp. Exact node depends on the order the user entered numbers in
21. %rdx ← %eax → Copy Counter5 to %rdx
22. %rdx ← M[%rsp + 8 * %rdx] → Copy contents of M[%rsp + 8 * Counter5] to %rdx
23. M[%rcx + 8] ← %rdx → Copy read value to M[%rcx + 8]
24. %eax ← %eax + 1 → Increment Counter5
25. %rcx ← %rdx → Change address in %rcx to address in %rdx
19. Set CC for %eax - 5
20. If %eax <= 5, jump to 162 — Potential split
→ Loop if Counter5 < 6
21. M[%rcx + 8] ← 0 → Set value in M[%rcx + 8] to 0 → Set value of last node pointer to null
22. %ebp ← 0 → Set counter to 0
23. Jump to 206
30. %rbx ← M[%rbx + 8] → Copy value in M[%rbx + 8] to %rbx → Copy address of next node to %rbx

Using debugger, I was able to identify the location where each of the digits I entered was stored after read_six_numbers

number	Location
First	%rsp + 48
Second	%rsp + 52
Third	%rsp + 56
Fourth	%rsp + 60
Fifth	%rsp + 64
Sixth	%rsp + 68

After this loop finishes, counter increments and the process begins again

During each iteration, the contents of M[%rsp + 4 * counter + 48] are compared with the contents of 5 - counter memory locations.

Counter	Number entered
0	First
1	Second
2	Third
3	Fourth
4	Fifth
5	Sixth

Each number entered is compared to all the numbers (up to the fifth) entered after it. The numbers must never be the same or the bomb explodes

First number compared with second, third, fourth, fifth, sixth
Second number compared with third, fourth, fifth, sixth
Third number compared with fourth, fifth, sixth
Fourth number compared with fifth, sixth
Fifth number compared with sixth

All numbers must be between 1 – 6

Node Info
They are 16 bytes in size
First 8 bytes can store long number
Next 8 bytes store address of next node
M[address] = number (int or long)
M[address + 8] = address of next node
Sixth node has null address at +8

Numbers in nodes:

	long	int	char
Node 1 →	0x00000001	000000	4C
Node 2 →	0x00000002	000000	64
Node 3 →	0x00000003	000002	6d
Node 4 →	0x00000004	000000	f9
Node 5 →	0x00000005	000002	6c
Node 6 →	0x00000006	000003	e6

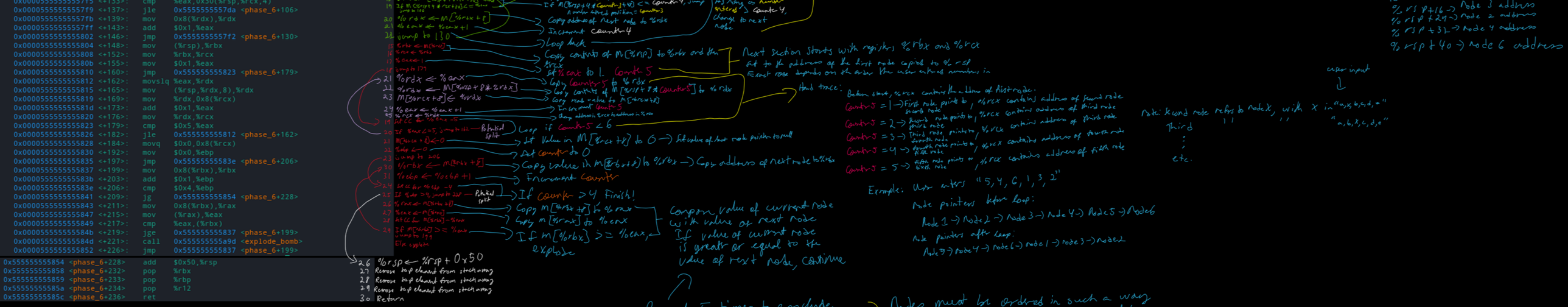
Example: If user enters "5, 1, 3, 2, 4, 6"
Node addresses will be stored as follows:
%rsp → Node 5 address
%rsp + 8 → Node 1 address
%rsp + 16 → Node 3 address
%rsp + 24 → Node 2 address
%rsp + 32 → Node 4 address
%rsp + 40 → Node 6 address

Hand trace:
Before start, %rcx contains the address of first node.
Counter5 = 1 → First node points to second node, %rcx contains address of second node
Counter5 = 2 → Second node points to third node, %rcx contains address of third node
Counter5 = 3 → Third node points to fourth node, %rcx contains address of fourth node
Counter5 = 4 → Fourth node points to fifth node, %rcx contains address of fifth node
Counter5 = 5 → Fifth node points to sixth node, %rcx contains

Note: Second node refers to node X, with X in "a, X, b, c, d, e" (user input)
Third " " "a, b, X, c, d, e"
⋮
etc.

```
0x00005555555557f5 <+133>:   cmp    %eax,0x50(%rsp,%rcx,4)
0x00005555555557f9 <+137>:   jle    0x5555555557da <phase_6+106>
0x00005555555557fb <+139>:   mov    0x8(%rdx),%rdx
0x00005555555557ff <+143>:   add    $0x1,%eax
0x0000555555555802 <+146>:   jmp    0x55555555557f2 <phase_6+130>
0x0000555555555804 <+148>:   mov    (%rsp),%rbx
0x0000555555555808 <+152>:   mov    %rbx,%rcx
0x000055555555580b <+155>:   mov    $0x1,%eax
0x0000555555555810 <+160>:   jmp    0x555555555823 <phase_6+179>
0x0000555555555812 <+162>:   movslq %eax,%rdx
0x0000555555555815 <+165>:   mov    (%rsp,%rdx,8),%rdx
0x0000555555555819 <+169>:   mov    %rdx,0x8(%rcx)
0x000055555555581d <+173>:   add    $0x1,%eax
0x0000555555555820 <+176>:   mov    %rdx,%rcx
0x0000555555555823 <+179>:   cmp    $0x5,%eax
0x0000555555555826 <+182>:   jle    0x555555555812 <phase_6+162>
0x0000555555555828 <+184>:   movq   $0x0,0x8(%rcx)
0x0000555555555830 <+192>:   mov    $0x0,%ebp
0x0000555555555835 <+197>:   jmp    0x55555555583e <phase_6+206>
0x0000555555555837 <+199>:   mov    0x8(%rbx),%rbx
0x000055555555583b <+203>:   add    $0x1,%ebp
0x000055555555583e <+206>:   cmp    $0x4,%ebp
0x0000555555555841 <+209>:   jg     0x555555555854 <phase_6+228>
0x0000555555555843 <+211>:   mov    0x8(%rbx),%rax
0x0000555555555847 <+215>:   mov    (%rax),%eax
0x0000555555555849 <+217>:   cmp    %eax,(%rbx)
0x000055555555584b <+219>:   jge    0x555555555837 <phase_6+199>
0x000055555555584d <+221>:   call   0x55555555a9d <explode_bomb>
0x0000555555555852 <+226>:   jmp    0x555555555837 <phase_6+199>

0x555555555854 <phase_6+228>:  add    $0x50,%rsp
0x555555555858 <phase_6+232>:  pop    %rbx
0x555555555859 <phase_6+233>:  pop    %rbp
0x55555555585a <phase_6+234>:  pop    %r12
0x55555555585c <phase_6+236>:  ret
```



Solution explanation:

User must input the numbers in a way that orders them in descending order during the final comparisons.

Each number the user inputs represents the node of the some number. The order of the nodes during the final comparison will depend on the order in which the user enters the numbers.

Example:

If User enters "6 2 1 3 5 4"

Nodes will be compared as follows:

node 6 >= node 2 >= node 1 >= node 3 >= node 5 >= node 4

If the comparison fails, bomb explodes.
If the comparison is successful, phase solved.

Solution:

Numbers in nodes:

	long	int	char
node 1 →	0x000000001	000000	4C
node 2 →	0x000000002	000000	64
node 3 →	0x000000003	000002	6d
node 4 →	0x000000004	000000	f9
node 5 →	0x000000005	000002	cc
node 6 →	0x000001006	000003	e6

Note that the compare instruction is cmp %eax, (%rdx), which compares only 32 bits integer, not 64 bit integer

In order of decreasing values,
node 6, node 3, node 5, node 4, node 2, node 1

Thus, user must enter "6 3 5 4 2 1"