CSE251: System Programming

Assignment 2 - bomblab

Student name: Nguyen Minh Duc
Student ID: 20202026

This paper contains solutions to the assignment 2 of System Programming course (Spring 2022) at UNIST, which is to defuse the binary bomb.

Contents

Introduction

To begin with, let's first look at the source code **bomb.c**

```
input = read_line();             /* Get input                   */
phase_1(input);                  /* Run the phase               */
phase_defused();                 /* Drat!  They figured it out!
          * Let me know how they did it. */
printf("Phase 1 defused. How about the next one?\n");

/* The second phase is harder.  No one will ever figure out
  * how to defuse this... */
input = read_line();
phase_2(input);
phase_defused();
printf("That's number 2.  Keep going!\n");

/* I guess this is too easy so far.  Some more complex code will
  * confuse people. */
input = read_line();
phase_3(input);
phase_defused();
printf("Halfway there!\n");

/* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
input = read_line();
phase_4(input);
phase_defused();
printf("So you got that one.  Try this one.\n");

/* Round and 'round in memory we go, where we stop, the bomb blows! */
input = read_line();
phase_5(input);
phase_defused();
printf("Good work!  On to the next...\n");

/* This phase will never be used, since no one will get past the
  * earlier ones.  But just in case, make this one extra hard. */
input = read_line();
phase_6(input);
phase_defused();
```

We can see that the bomb has six phases, and each phase requires us to input a certain string to defuse. There are 6 functions correspoding to 6 phases:

```
phase_1, phase_2, phase_3, phase_4, phase_5, phase_6
```

1 Phase 1

This is the first phase of the binary bomb. The bomb will greet you with the following lines

```
[edusc03-052@cheetah022 bomb52]$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

Entering a random string "123" will result in an explosion.

```
[edusc03-052@cheetah022 bomb52]$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
123

BOOM!!!
The bomb has blown up.
[edusc03-052@cheetah022 bomb52]$
```

In order to obtain the correct string, we have to use the **gdb** tool. Now we know that the function that handle the first phase is `phase_1`, we can set a breakpoint at that function. Starting the bomb with the command `r` and inputting the random string "'123"' again, we obtain

```
[edusc03-052@cheetah022 bomb52]$ gdb bomb
(gdb) b phase_1
Breakpoint 1 at 0x400e63
(gdb) r
Starting program: /home/edusc03/edusc03-052/bomb52/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
123

Breakpoint 1, 0x0000000000400e63 in phase_1 ()
(gdb)
```

Disassembling the function, we have

```
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000400e63 <+0>:     sub    $0x8,%rsp
   0x0000000000400e67 <+4>:     mov    $0x402330,%esi
   0x0000000000400e6c <+9>:     callq  0x401354 <strings_not_equal>
   0x0000000000400e71 <+14>:    test   %eax,%eax
   0x0000000000400e73 <+16>:    jne    0x400e7a <phase_1+23>
```

```
    0x0000000000400e75 <+18>:    add    $0x8,%rsp
    0x0000000000400e79 <+22>:    retq
    0x0000000000400e7a <+23>:    callq  0x401451 <explode_bomb>
    0x0000000000400e7f <+28>:    jmp    0x400e75 <phase_1+18>
End of assembler dump.
```

As the name suggests, the intruction `callq  0x401354 <strings_not_equal>` might compare two strings to see whether they are not equal. The previous instruction `mov $0x402330,%esi` also suggests one of the arguments will be in the address `$0x402330`. By examining the address with the command **x/s**, we obtain the following string

```
(gdb) x/s 0x402330
0x402330:       "Verbosity leads to unclear, inarticulate things."
```

which has a high chance to be the desired string. The following lines conducts an if-else statement, if the function returns true, the bomb will detonate, otherwise, the phase will be defused. Let's investigate the `strings_not_equal` function

```
(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
  0x0000000000401354 <+0>:     push   %r12
  0x0000000000401356 <+2>:     push   %rbp
  0x0000000000401357 <+3>:     push   %rbx
  0x0000000000401358 <+4>:     mov    %rdi,%rbx
  0x000000000040135b <+7>:     mov    %rsi,%rbp
  0x000000000040135e <+10>:    callq  0x401337 <string_length>
  0x0000000000401363 <+15>:    mov    %eax,%r12d
  0x0000000000401366 <+18>:    mov    %rbp,%rdi
  0x0000000000401369 <+21>:    callq  0x401337 <string_length>
  0x000000000040136e <+26>:    mov    $0x1,%edx
  0x0000000000401373 <+31>:    cmp    %eax,%r12d
  0x0000000000401376 <+34>:    je     0x40137f <strings_not_equal+43>
---Type <return> to continue, or q <return> to quit---q
```

It is clear that the function is comparing the two strings reside in `%rdi` and `%rsi` registers. By a direct examination, we obtain the two strings

```
(gdb) x/s $rsi
0x402330:       "Verbosity leads to unclear, inarticulate things."
(gdb) x/s $rdi
0x6037a0 <input_strings>:       "123"
```

of which one of them is our inputted string. We can conclude that for any string we entered, the bomb compares it with the string "Verbosity leads to unclear, inarticulate things."
Reset the bomb and enter the newly found string, we have defused the first phase

```
[edusc03-052@cheetah022 bomb52]$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Verbosity leads to unclear, inarticulate things.
Phase 1 defused. How about the next one?
```

2 Phase 2

This is the second phase of the binary bomb. The bomb will greet you with the following lines

```
[edusc03-052@cheetah022 bomb52]$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Verbosity leads to unclear, inarticulate things.
Phase 1 defused. How about the next one?
```

let's first open the **gdb** tool up, examine and put a breakpoint at the function `phase_2`

```
(gdb) r
Starting program: /home/edusc03/edusc03-052/bomb52/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Verbosity leads to unclear, inarticulate things.
Phase 1 defused. How about the next one?
123

Breakpoint 1, 0x0000000000400e81 in phase_2 ()
(gdb) disas
Dump of assembler code for function phase_2:
=> 0x0000000000400e81 <+0>:     push   %rbx
   0x0000000000400e82 <+1>:     sub    $0x20,%rsp
   0x0000000000400e86 <+5>:     mov    %rsp,%rsi
   0x0000000000400e89 <+8>:     callq  0x401473 <read_six_numbers>
   0x0000000000400e8e <+13>:    cmpl   $0x0,(%rsp)
   0x0000000000400e92 <+17>:    js     0x400e9b <phase_2+26>
   0x0000000000400e94 <+19>:    mov    $0x1,%ebx
   0x0000000000400e99 <+24>:    jmp    0x400eac <phase_2+43>
   0x0000000000400e9b <+26>:    callq  0x401451 <explode_bomb>
   0x0000000000400ea0 <+31>:    jmp    0x400e94 <phase_2+19>
   0x0000000000400ea2 <+33>:    add    $0x1,%rbx
   0x0000000000400ea6 <+37>:    cmp    $0x6,%rbx
   0x0000000000400eaa <+41>:    je     0x400ebe <phase_2+61>
   0x0000000000400eac <+43>:    mov    %ebx,%eax
   0x0000000000400eae <+45>:    add    -0x4(%rsp,%rbx,4),%eax
   0x0000000000400eb2 <+49>:    cmp    %eax,(%rsp,%rbx,4)
   0x0000000000400eb5 <+52>:    je     0x400ea2 <phase_2+33>
   0x0000000000400eb7 <+54>:    callq  0x401451 <explode_bomb>
   0x0000000000400ebc <+59>:    jmp    0x400ea2 <phase_2+33>
   0x0000000000400ebe <+61>:    add    $0x20,%rsp
   0x0000000000400ec2 <+65>:    pop    %rbx
   0x0000000000400ec3 <+66>:    retq
End of assembler dump.
```

Again, the function tries to call another function, this time is `read_six_numbers`. Let's examine it

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x0000000000401473 <+0>:     sub    $0x8,%rsp
   0x0000000000401477 <+4>:     mov    %rsi,%rdx
   0x000000000040147a <+7>:     lea    0x4(%rsi),%rcx
   0x000000000040147e <+11>:    lea    0x14(%rsi),%rax
   0x0000000000401482 <+15>:    push   %rax
   0x0000000000401483 <+16>:    lea    0x10(%rsi),%rax
   0x0000000000401487 <+20>:    push   %rax
   0x0000000000401488 <+21>:    lea    0xc(%rsi),%r9
   0x000000000040148c <+25>:    lea    0x8(%rsi),%r8
   0x0000000000401490 <+29>:    mov    $0x402523,%esi
   0x0000000000401495 <+34>:    mov    $0x0,%eax
   0x000000000040149a <+39>:    callq  0x400bc0 <__isoc99_sscanf@plt>
   0x000000000040149f <+44>:    add    $0x10,%rsp
   0x00000000004014a3 <+48>:    cmp    $0x5,%eax
   0x00000000004014a6 <+51>:    jle    0x4014ad <read_six_numbers+58>
   0x00000000004014a8 <+53>:    add    $0x8,%rsp
   0x00000000004014ac <+57>:    retq
   0x00000000004014ad <+58>:    callq  0x401451 <explode_bomb>
End of assembler dump.
```

We can see that the function tries to invoke `__isoc99_sscanf` function. The first few lines of the function load data into registers and push some of them on the stack. The name of those registers suggests that they contain arguments to be passed into `sscanf`. As we know, it takes in at least two arguments, a string s and an C-formatted pattern p, and returns the number of matches in s according to the pattern p. Note that the values in the address `0x402523` is moved into the `%esi` register, which is usually an argument. Inspect that memory, we obtain

```
(gdb) x/s 0x402523
0x402523:       "%d %d %d %d %d %d"
```

which suggests that the desired string should contain six 32-bit integers separated by spaces. The last few lines construct an if-else statement, which is roughly translated into C as

```
if(eax > 5){ // eax = return value of __isoc99_sscanf
  return;
}
explode_bomb();
```

We will pass this stage if eax > 5, i.e, the number of integers in our string is at least 6. If we continue to run our debugger, the bomb will explode since our first guess only contains one

integer 123.
Reset the bomb and type in "1 2 3 4 5 6", we now have passed the `read_six_numbers` function. The remainning lines of codes seem comlicated, let's explore them part by part. Firstly, let's inspect what's residing in the `%rsp` register

```
(gdb) x $rsp
0x7fffffffe210: 0x00000001
```

If we reset the bomb again and enter "0 1 2 3 4 5", we obtain another value of `rsp` in this stage

```
(gdb) x $rsp
0x7fffffffe210: 0x00000000
```

which is the same as the first integer we have just inputted. To confirm our hypothesis that `rsp` contains the six inputted integers, we type in the command `x/6d` in gdb

```
(gdb) x/6d $rsp
0x7fffffffe210: 1       2       3       4
0x7fffffffe220: 5       6
```

Now we know that `rsp` holds our input, let's carry on with the following lines

```
0x0000000000400e8e <+13>:    cmpl   $0x0,(%rsp)
0x0000000000400e92 <+17>:    js     0x400e9b <phase_2+26>
```

This checks if the first number is less than 0. If it is, then the bomb will detonate, otherwise, it will continue with the following instructions

```
0x0000000000400e94 <+19>:    mov    $0x1,%ebx
0x0000000000400e99 <+24>:    jmp    0x400eac <phase_2+43>
0x0000000000400e9b <+26>:    callq  0x401451 <explode_bomb>
0x0000000000400ea0 <+31>:    jmp    0x400e94 <phase_2+19>
0x0000000000400ea2 <+33>:    add    $0x1,%rbx
0x0000000000400ea6 <+37>:    cmp    $0x6,%rbx
0x0000000000400eaa <+41>:    je     0x400ebe <phase_2+61>
0x0000000000400eac <+43>:    mov    %ebx,%eax
0x0000000000400eae <+45>:    add    -0x4(%rsp,%rbx,4),%eax
0x0000000000400eb2 <+49>:    cmp    %eax,(%rsp,%rbx,4)
0x0000000000400eb5 <+52>:    je     0x400ea2 <phase_2+33>
0x0000000000400eb7 <+54>:    callq  0x401451 <explode_bomb>
0x0000000000400ebc <+59>:    jmp    0x400ea2 <phase_2+33>
```

This part of the code is the most complicated. It has a lot of backward jumps, so it might be a loop. Further investigation confirms our hypothesis. First, `ebx` is initialized to 1 and the value is moved to `eax`. Then, the bomb executes `add -0x4(%rsp,%rbx,4),%eax`, which can be translated into C code as: `eax` += `rsp`[`rbx` − 1], assuming that everything is of type `int`. Next, the bomb will compare `eax` with `rsp`[`rbx`], if they are equal, we continue with the line 33, otherwise, the bomb will explode. Lastly, the bomb will increment `rbx` and check if it is equal to 6. If it is, the loop will terminate and phase 2 is defused, otherwise, the loop will continue. In summary, the assembly code can be roughly translated into the following C code

```
// suppose that a[6] is the array that contains the input
int i = 1;
do{
  if(i + a[i - 1] != a[i]) explode_bomb();
  i += 1;
} while(i != 6);
```

This gives us a recurrence formula for the desired series of integers

$$a_n = \begin{cases} c & \text{if } n = 0, \\ a_{n-1} + n & \text{otherwise,} \end{cases}$$

for any arbitary integer $c \geq 0$. Solving this recurrence formula, we obtain the result

$$a_n = c + \frac{n(n+1)}{2}.$$

This suggests that there's more than one solution to this phase. For simplicity, let's choose $c = 0$, reset the bomb and input the string "0 1 3 6 10 15", which will defuse the second phase of the bomb.

```
[edusc03-052@cheetah022 bomb52]$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Verbosity leads to unclear, inarticulate things.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2.  Keep going!
```

3 Phase 3

This is the third phase of the binary bomb. The bomb will greet you with the following lines

```
[edusc03-052@cheetah022 bomb52]$ ./bomb  input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
1 2 3 4 5 6
```

Let's entered a dummy string into the bomb, i.e. "1 2 3 4 5 6".
As usual, let's investigate the phase by **gdb**. Since the assembly code is too long, I will break it down into different sections.

```
0x0000000000400ec4 <+0>:     sub    $0x18,%rsp
0x0000000000400ec8 <+4>:     lea    0x8(%rsp),%r8
0x0000000000400ecd <+9>:     lea    0x7(%rsp),%rcx
0x0000000000400ed2 <+14>:    lea    0xc(%rsp),%rdx
0x0000000000400ed7 <+19>:    mov    $0x40238e,%esi
0x0000000000400edc <+24>:    mov    $0x0,%eax
0x0000000000400ee1 <+29>:    callq  0x400bc0 <__isoc99_sscanf@plt>
0x0000000000400ee6 <+34>:    cmp    $0x2,%eax
0x0000000000400ee9 <+37>:    jle    0x400f01 <phase_3+61>
```

As we can see, this function also invoke `sscanf`, let's investigate what is the pattern `sscanf` takes as argument

```
(gdb) x/s 0x40238e
0x40238e:       "%d %c %d"
```

This implies that our input should have `int`, `char`, and `int` respectively. Since our dummy input satisfies the constraint, we pass this stage.
The next two lines also form an if-else statement. If `0xc(%rsp)`$>$ 7, the bomb will explode, otherwise, it will move the data into `%eax` and continue into the next stage.

```
0x0000000000400eeb <+39>:    cmpl   $0x7,0xc(%rsp)
0x0000000000400ef0 <+44>:    ja     0x400ff9 <phase_3+309>
0x0000000000400ef6 <+50>:    mov    0xc(%rsp),%eax
0x0000000000400efa <+54>:    jmpq   *0x4023a0(,%rax,8)
```

Let's explore what's inside `0xc(%rsp)` and `0x4023a0`

```
(gdb) x $rsp + 0xc
0x7ffffffe22c: 0x00000001 # our first integer
```

```
(gdb) x 0x4023a0
0x4023a0:       0x0000000000400f08
```

The last instruction jumps to the $*(\texttt{0x4023a0} + 8\times$ `%rax`$)$ line, where `%rax` is our first integer. This kind of behavior suggests a switch statement in C. Since `%rax` must be less than 7, there are only eight cases needed to be handled, i.e. from 0 to 7.

```
switch(/*our first integer*/){
  case 0: // do something
  case 1: // do something
  case 2: // do something
  case 3: // do something
  case 4: // do something
  case 5: // do something
  case 6: // do something
  case 7: // do something
  default: explode_bomb();
}
```

In our dummy input, our first number is 1, so we jump to the line `0x0000000000400f2a <+102>`

```
0x0000000000400f2a <+102>:   mov    $0x72,%eax
0x0000000000400f2f <+107>:   cmpl   $0x10f,0x8(%rsp)
0x0000000000400f37 <+115>:   je     0x401003 <phase_3+319>
0x0000000000400f3d <+121>:   callq  0x401451 <explode_bomb>
```

Let's first see what's inside `0x8(%rsp)`

```
(gdb) print *(int*)(0x8 + $rsp)
$17 = 3
```

which is our third integer. Now, let's translate the above assembly code into C code

```
case 1:
  eax = 0x72;
  if(0x10f != *(rsp + 0x8)) explode_bomb();
  break;
```

This means that our third number must be equal to `0x10f`, i.e. 271, which is not the case here. Let's reset the bomb and modify the input string

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

```
Verbosity leads to unclear, inarticulate things.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2.  Keep going!
1 a 271
```

We have now passed the switch statement, and left with just a few instructions

```
0x0000000000401003 <+319>:   cmp    %al,0x7(%rsp)
0x0000000000401007 <+323>:   je     0x40100e <phase_3+330>
0x0000000000401009 <+325>:   callq  0x401451 <explode_bomb>
0x000000000040100e <+330>:   add    $0x18,%rsp
0x0000000000401012 <+334>:   retq
```

It's comparing two values at `%al` and `0x7(%rsp)`. Let's see what's in `0x7(%rsp)`

```
(gdb) x (int*)(0x7 + $rsp)
0x7fffffffe227: 97 'a'
```

which is our second input character. As for the register `%al`

```
(gdb) print (char)$al
$21 = 114 'r'
```

So now, we know that the second character must be '`r`'. Resetting the bomb and entering the string "1 r 271", we have now defused the third phase.

```
[edusc03-052@cheetah022 bomb52]$ ./bomb input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
1 r 271
Halfway there!
```

4 Phase 4

This is the forth phase of the binary bomb. The bomb will greet you with the following lines

```
[edusc03-052@cheetah022 bomb52]$ ./bomb input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
```

Let's entered a dummy string into the bomb, i.e. "1 2 3". Now, we obtain the following assembly code

```
0x0000000000401047 <+0>:     sub    $0x18,%rsp
0x000000000040104b <+4>:     lea    0x8(%rsp),%rcx
0x0000000000401050 <+9>:     lea    0xc(%rsp),%rdx
0x0000000000401055 <+14>:    mov    $0x40252f,%esi
0x000000000040105a <+19>:    mov    $0x0,%eax
0x000000000040105f <+24>:    callq  0x400bc0 <__isoc99_sscanf@plt>
0x0000000000401064 <+29>:    cmp    $0x2,%eax
0x0000000000401067 <+32>:    jne    0x401070 <phase_4+41>
0x0000000000401069 <+34>:    cmpl   $0xe,0xc(%rsp)
0x000000000040106e <+39>:    jbe    0x401075 <phase_4+46>
0x0000000000401070 <+41>:    callq  0x401451 <explode_bomb>
0x0000000000401075 <+46>:    mov    $0xe,%edx
0x000000000040107a <+51>:    mov    $0x0,%esi
0x000000000040107f <+56>:    mov    0xc(%rsp),%edi
0x0000000000401083 <+60>:    callq  0x401013 <func4>
0x0000000000401088 <+65>:    cmp    $0x12,%eax
0x000000000040108b <+68>:    jne    0x401094 <phase_4+77>
0x000000000040108d <+70>:    cmpl   $0x12,0x8(%rsp)
0x0000000000401092 <+75>:    je     0x401099 <phase_4+82>
0x0000000000401094 <+77>:    callq  0x401451 <explode_bomb>
0x0000000000401099 <+82>:    add    $0x18,%rsp
0x000000000040109d <+86>:    retq
```

Again, the function tries to invoke `sscanf`, let's see what's the pattern

```
(gdb) x/s 0x40252f
0x40252f:        "%d %d"
```

which implies our input should contain two 32-bit integer. The two lines

```
0x0000000000401064 <+29>:    cmp    $0x2,%eax
0x0000000000401067 <+32>:    jne    0x401070 <phase_4+41>
```

say that if we do not meet the criteria, the bomb will explode. Luckily, our dummy input statisfies the first constraint. Next, there're these two lines

```
0x0000000000401069 <+34>:    cmpl   $0xe,0xc(%rsp)
0x000000000040106e <+39>:    jbe    0x401075 <phase_4+46>
0x0000000000401070 <+41>:    callq  0x401451 <explode_bomb>
```

comparing `0xc(%rsp)` with `0xe`, i.e. 14. Let's investigate `0xc(%rsp)`

```
(gdb) x/d (0xc + $rsp)
0x7fffffffe22c: 1
```

which is our first integer. So if our first integer is bigger than 14, the bomb will also detonate. Next, let's examine the following lines

```
0x0000000000401075 <+46>:    mov    $0xe,%edx
0x000000000040107a <+51>:    mov    $0x0,%esi
0x000000000040107f <+56>:    mov    0xc(%rsp),%edi
0x0000000000401083 <+60>:    callq  0x401013 <func4>
```

It calls another function `func4` that takes in 3 arguments, `0xe`, `0x0`, and our first number. Dive in `func4`

```
0x0000000000401013 <+0>:     push   %rbx
0x0000000000401014 <+1>:     mov    %edx,%eax
0x0000000000401016 <+3>:     sub    %esi,%eax
0x0000000000401018 <+5>:     mov    %eax,%ebx
0x000000000040101a <+7>:     shr    $0x1f,%ebx
0x000000000040101d <+10>:    add    %eax,%ebx
0x000000000040101f <+12>:    sar    %ebx
0x0000000000401021 <+14>:    add    %esi,%ebx
0x0000000000401023 <+16>:    cmp    %edi,%ebx
0x0000000000401025 <+18>:    jg     0x40102f <func4+28>
0x0000000000401027 <+20>:    cmp    %edi,%ebx
0x0000000000401029 <+22>:    jl     0x40103b <func4+40>
0x000000000040102b <+24>:    mov    %ebx,%eax
0x000000000040102d <+26>:    pop    %rbx
0x000000000040102e <+27>:    retq
0x000000000040102f <+28>:    lea    -0x1(%rbx),%edx
0x0000000000401032 <+31>:    callq  0x401013 <func4>
0x0000000000401037 <+36>:    add    %eax,%ebx
0x0000000000401039 <+38>:    jmp    0x40102b <func4+24>
0x000000000040103b <+40>:    lea    0x1(%rbx),%esi
0x000000000040103e <+43>:    callq  0x401013 <func4>
0x0000000000401043 <+48>:    add    %eax,%ebx
0x0000000000401045 <+50>:    jmp    0x40102b <func4+24>
```

We can see that it calls itself at the line 31 and 43, so it is a recursive function. Note that it takes 3 arguments, let's call them x, y, z, which are `%edx`, `%esi`, and `%edi` respectively. Firstly, let's try to understand these first lines of codes

```
0x0000000000401014 <+1>:    mov    %edx,%eax
0x0000000000401016 <+3>:    sub    %esi,%eax
0x0000000000401018 <+5>:    mov    %eax,%ebx
0x000000000040101a <+7>:    shr    $0x1f,%ebx
0x000000000040101d <+10>:   add    %eax,%ebx
0x000000000040101f <+12>:   sar    %ebx
0x0000000000401021 <+14>:   add    %esi,%ebx
```

which can be translated line by line as

```
eax = edx;
eax -= esi;
ebx = eax;
ebx = (unsigned)ebx >> 0x1f;
ebx += eax;
ebx >>= 1;
ebx += esi;
```

let's instroduce new variables and subtitute x, y, z in the correct registers

```
int a = x - y;
int b = a;
b = (unsigned)b >> 0x1f;
b += a;
b >>= 1;
b += y;
```

The next instructions construct an if-else statement

```
0x0000000000401023 <+16>:   cmp    %edi,%ebx
0x0000000000401025 <+18>:   jg     0x40102f <func4+28>
0x0000000000401027 <+20>:   cmp    %edi,%ebx
0x0000000000401029 <+22>:   jl     0x40103b <func4+40>
0x000000000040102b <+24>:   mov    %ebx,%eax
0x000000000040102d <+26>:   pop    %rbx
0x000000000040102e <+27>:   retq
```

Let's translate into C:

```
if(b > z){
  // func4+28
```

```
} else if(b < z){
  // func4+40
}
a = b;
return a;
```

Let's explore the line 28 of the function

```
0x000000000040102f <+28>:    lea    -0x1(%rbx),%edx
0x0000000000401032 <+31>:    callq  0x401013 <func4>
0x0000000000401037 <+36>:    add    %eax,%ebx
0x0000000000401039 <+38>:    jmp    0x40102b <func4+24>
```

Translating into C, we obtain

```
x = b - 1;
a = func4(x, y, z);
b += a;
```

As for the line 40:

```
0x000000000040103b <+40>:    lea    0x1(%rbx),%esi
0x000000000040103e <+43>:    callq  0x401013 <func4>
0x0000000000401043 <+48>:    add    %eax,%ebx
0x0000000000401045 <+50>:    jmp    0x40102b <func4+24>
```

Translating into C, we obtain

```
y = b + 1;
a = func4(x, y, z);
b += a;
```

Now, we have completely reverse engineered `func4`. The C code is as follow

```
int func4(int x, int y, int z){
  int a = x - y;
  int b = a;
  b = (unsigned)b >> 31;
  b += a;
  b >>= 1;
  b += y;
  if(b - z > 0){
      x = b - 1;
```

```
        a = func4(x, y, z);
        b += a;
    } else if(b - z < 0){
        y = b + 1;
        a = func4(x, y, z);
        b += a;
    }
    a = b;
    return a;
}
```

Now, we have known what `func4` is doing, let's continue with the assembly code of `phase_4`

```
0x0000000000401088 <+65>:    cmp    $0x12,%eax
0x000000000040108b <+68>:    jne    0x401094 <phase_4+77>
0x000000000040108d <+70>:    cmpl   $0x12,0x8(%rsp)
0x0000000000401092 <+75>:    je     0x401099 <phase_4+82>
0x0000000000401094 <+77>:    callq  0x401451 <explode_bomb>
0x0000000000401099 <+82>:    add    $0x18,%rsp
0x000000000040109d <+86>:    retq
```

where `%eax` is the returned value of `func4`. If it is not `0x12`, i.e. 18, then the bomb will explode, otherwise, it continue to compare `0x8(%rsp)` with 18. Let's see what's inside

```
(gdb) x/d (0x8+$rsp)
0x7fffffffe228: 2
```

which is our second integer. So if it is not the same as 18, the bomb will detonate, otherwise, we successfully defuse the bomb. The only problem left is that what should the first number, let's call it z, be in order for the output of `func4` to be 18. Fortunately, we know the range of the input, which is $z \leq 14$, and we also know the C code of `func4`. The task is now trivial as we only need to bruteforce through 15 different values of z, i.e. $\{0, 1, 2, \ldots, 14\}$ with a simple C code

```
#include <stdio.h>
int func4(int x, int y, int z){
    int a = x - y;
    int b = a;
    b = (unsigned)b >> 31;
    b += a; b >>= 1; b += y;
    if(b > z){
        a = func4(b - 1, y, z);
        b += a;
    } else if(b < z){
        a = func4(x, b + 1, z);
```

```
        b += a;
    }
    a = b;
    return a;
}
int main() {
    for(int i = 0; i <= 14; i++)
        printf("i = %d -> %d\n", i, func4(14, 0, i));
    return 0;
}
```

The output is

```
i = 0 -> 11
i = 1 -> 11
i = 2 -> 13
i = 3 -> 10
i = 4 -> 19
i = 5 -> 15
i = 6 -> 21
i = 7 -> 7
i = 8 -> 35
i = 9 -> 27
i = 10 -> 37
i = 11 -> 18
i = 12 -> 43
i = 13 -> 31
i = 14 -> 45
```

From this, we know that our first number should be 11. Resetting the bomb and enter “`11 18`”, this phase will be defused successfully.

```
[edusc03-052@cheetah022 bomb52]$ ./bomb input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
11 18
So you got that one.  Try this one.
```

5 Phase 5

This is the fifth phase of the binary bomb. The bomb will greet you with the following lines

```
[edusc03-052@cheetah022 bomb52]$ ./bomb input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
```

As usual, let's enter a dummy input "1 2" and inspect the phase's assembly code

```
0x000000000040109e <+0>:     sub    $0x18,%rsp
0x00000000004010a2 <+4>:     lea    0x8(%rsp),%rcx
0x00000000004010a7 <+9>:     lea    0xc(%rsp),%rdx
0x00000000004010ac <+14>:    mov    $0x40252f,%esi
0x00000000004010b1 <+19>:    mov    $0x0,%eax
0x00000000004010b6 <+24>:    callq  0x400bc0 <__isoc99_sscanf@plt>
0x00000000004010bb <+29>:    cmp    $0x1,%eax
0x00000000004010be <+32>:    jle    0x40110a <phase_5+108>
0x00000000004010c0 <+34>:    mov    0xc(%rsp),%eax
0x00000000004010c4 <+38>:    and    $0xf,%eax
0x00000000004010c7 <+41>:    mov    %eax,0xc(%rsp)
0x00000000004010cb <+45>:    cmp    $0xf,%eax
0x00000000004010ce <+48>:    je     0x401100 <phase_5+98>
0x00000000004010d0 <+50>:    mov    $0x0,%ecx
0x00000000004010d5 <+55>:    mov    $0x0,%edx
0x00000000004010da <+60>:    add    $0x1,%edx
0x00000000004010dd <+63>:    cltq
0x00000000004010df <+65>:    mov    0x4023e0(,%rax,4),%eax
0x00000000004010e6 <+72>:    add    %eax,%ecx
0x00000000004010e8 <+74>:    cmp    $0xf,%eax
0x00000000004010eb <+77>:    jne    0x4010da <phase_5+60>
0x00000000004010ed <+79>:    movl   $0xf,0xc(%rsp)
0x00000000004010f5 <+87>:    cmp    $0xf,%edx
0x00000000004010f8 <+90>:    jne    0x401100 <phase_5+98>
0x00000000004010fa <+92>:    cmp    %ecx,0x8(%rsp)
0x00000000004010fe <+96>:    je     0x401105 <phase_5+103>
0x0000000000401100 <+98>:    callq  0x401451 <explode_bomb>
0x0000000000401105 <+103>:   add    $0x18,%rsp
0x0000000000401109 <+107>:   retq
0x000000000040110a <+108>:   callq  0x401451 <explode_bomb>
0x000000000040110f <+113>:   jmp    0x4010c0 <phase_5+34>
```

This function also calls `sscanf`, let's see what's the pattern

```
(gdb) x/s 0x40252f
0x40252f:        "%d %d"
```

So our dummy input statisfies the first constraint. Next, let's inspect the following lines

```
0x00000000004010c0 <+34>:    mov    0xc(%rsp),%eax
0x00000000004010c4 <+38>:    and    $0xf,%eax
0x00000000004010c7 <+41>:    mov    %eax,0xc(%rsp)
0x00000000004010cb <+45>:    cmp    $0xf,%eax
0x00000000004010ce <+48>:    je     0x401100 <phase_5+98>
```

It moves something into the `%eax` register, and conduct an `AND` operation with `0xf`, i.e. 15, or equivalently in base 2: `1111`. This effectively is a modulo operation mod $2^4 = 16$. Then, it move the data back into `0xc(%rsp)`. The last two lines form an if statement, if the register is the same as 15, the bomb will explode, otherwise, we are good to go to the next state of the bomb. Let's first check what's inside `0xc(%rsp)`

```
(gdb) x/d (0xc + $rsp)
0x7fffffffe22c: 1
```

which is our first number. Thus, our dummy input statisfies this second constraint. Let's continue with the next lines of assembly codes

```
0x00000000004010d0 <+50>:    mov    $0x0,%ecx
0x00000000004010d5 <+55>:    mov    $0x0,%edx
0x00000000004010da <+60>:    add    $0x1,%edx
0x00000000004010dd <+63>:    cltq
0x00000000004010df <+65>:    mov    0x4023e0(,%rax,4),%eax
0x00000000004010e6 <+72>:    add    %eax,%ecx
0x00000000004010e8 <+74>:    cmp    $0xf,%eax
0x00000000004010eb <+77>:    jne    0x4010da <phase_5+60>
0x00000000004010ed <+79>:    movl   $0xf,0xc(%rsp)
0x00000000004010f5 <+87>:    cmp    $0xf,%edx
0x00000000004010f8 <+90>:    jne    0x401100 <phase_5+98>
```

This portion of the phase has a backward jump, so it might be a loop. Initially, there's two register `%ecx` and `%edx` both are assigned to 0. Then, the loop starts with the incrementation of `%edx`. The bomb move a value into the `%eax` register, let's investigate it.

```
(gdb) x 0x4023e0 + 4*$rax
0x4023e4 <array.3237+4>:        2
```

What's an interesting name, `array.3237`. We know that `%rax` contains our first number, and `(,%rax,4)` is an offset from the "array", so it might be the index of the "array". Let's confirm our guess by examining the address `0x4023e0`

```
(gdb) x/100d 0x4023e0
0x4023e0 <array.3237>:    10 0 0 0 2  0 0 0
0x4023e8 <array.3237+8>:  14 0 0 0 7  0 0 0
0x4023f0 <array.3237+16>: 8  0 0 0 12 0 0 0
0x4023f8 <array.3237+24>: 15 0 0 0 11 0 0 0
0x402400 <array.3237+32>: 0  0 0 0 4  0 0 0
0x402408 <array.3237+40>: 1  0 0 0 13 0 0 0
0x402410 <array.3237+48>: 3  0 0 0 9  0 0 0
0x402418 <array.3237+56>: 6  0 0 0 5  0 0 0
0x402420: 83  111 32  121 111 117 32  116
0x402428: 104 105 110 107 32  121 111 117
0x402430: 32  99  97  110 32  115 116 111
0x402438: 112 32  116 104 101 32  98  111
0x402440: 109 98  32  119
```

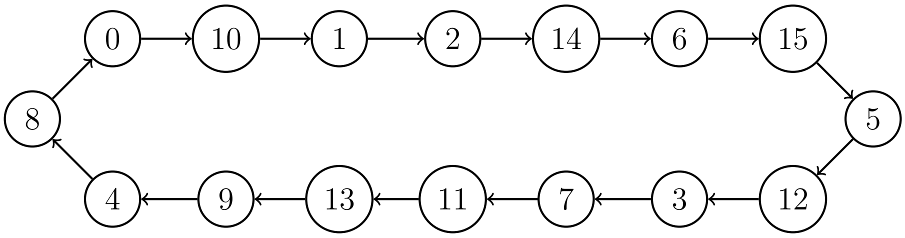
From this data, we can conclude that there's an array of size 16 and it's elements are

```c
int array[] = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5};
```

The next line is pretty simple, it adds `%eax` into `%ecx`, compares whether `%eax` is 15, if it is, terminate the loop, otherwise, continue the loop. Out side the loop, it assigns `0xc(%rsp)` to 15 and compares `%edx` with 15. If they are not equal, the bomb will detonate, otherwise nothing happens. All of this reasoning can be translated into C code as

```c
// Let x be our first number
int cnt, sum;
int array[] = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5};
cnt = 0; sum = 0;
do{
  int p = array[x];
  sum += p;
  x = p;
} while(x != 15);
x = 15;
if(cnt != 15) bomb_explode();
```

This behavior is a traversal through out the array using their values as the next index to visit. The below is the graph of this structure



The code tells us that the traversal terminates when it reaches the node 15. If all of the nodes are not visited, the bomb will explode, so we need to find a starting point such that it will visit every nodes. Fortunately, this is trivial, as we just need to look at the graph, which is the node 5. So our first number should be 5. Reset the bomb and type in "5 6" will pass this loop. Let's focus on the last lines

```
0x00000000004010fa <+92>:     cmp    %ecx,0x8(%rsp)
0x00000000004010fe <+96>:     je     0x401105 <phase_5+103>
0x0000000000401100 <+98>:     callq  0x401451 <explode_bomb>
0x0000000000401105 <+103>:    add    $0x18,%rsp
0x0000000000401109 <+107>:    retq
```

From the code above, we know that `%ecx` is the sum of the nodes that we passed through, let's inspect `0x8(%rsp)`, and `%ecx`

```
(gdb) x 0x8+$rsp
0x7fffffffe228: 6 # our second input
(gdb) print $ecx
$1 = 115
```

Therefore, we obtain the correct input for this phase is "5 115". Resetting the bomb and type in "5 115", we have successfully defused the bomb.

```
[edusc03-052@cheetah022 bomb52]$ ./bomb input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
5 115
Good work!  On to the next...
```

6 Phase 6

This is the last phase of the binary bomb. The bomb will greet you with the following lines

```
[edusc03-052@cheetah022 bomb52]$ ./bomb input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
```

Again, let's enter a dummy input "1 2" into the bomb, and investigate the assembly code. Since the assembly code is too long, I will break it down into different sections.

```
0x0000000000401111 <+0>:     push   %r14
0x0000000000401113 <+2>:     push   %r13
0x0000000000401115 <+4>:     push   %r12
0x0000000000401117 <+6>:     push   %rbp
0x0000000000401118 <+7>:     push   %rbx
0x0000000000401119 <+8>:     sub    $0x50,%rsp
0x000000000040111d <+12>:    lea    0x30(%rsp),%rsi
0x0000000000401122 <+17>:    callq  0x401473 <read_six_numbers>
0x0000000000401127 <+22>:    lea    0x30(%rsp),%r12
0x000000000040112c <+27>:    mov    %r12,%r13
0x000000000040112f <+30>:    mov    $0x0,%r14d
0x0000000000401135 <+36>:    jmp    0x40115d <phase_6+76>
0x0000000000401137 <+38>:    callq  0x401451 <explode_bomb>
```

We meet the function `read_six_numbers` again, which will detonate the bomb if we entered less than 6 integers. Let's reset the bomb and type "1 2 3 4 5 6" into the bomb, which will get us pass this first stage of the phase. Next, the bomb load `0x30(%rsp)` into the register `r12`, let's see what's inside

```
(gdb) x (0x30+$rsp)
0x7fffffffe1f0: 0x00000001
```

which is our first integer. Then, the bomb will jump to line 76. Let's go to it

```
0x000000000040115d <+76>:    mov    %r13,%rbp
0x0000000000401160 <+79>:    mov    0x0(%r13),%eax
0x0000000000401164 <+83>:    sub    $0x1,%eax
0x0000000000401167 <+86>:    cmp    $0x5,%eax
0x000000000040116a <+89>:    ja     0x401137 <phase_6+38>
```

```
0x000000000040116c <+91>:    add    $0x1,%r14d
0x0000000000401170 <+95>:    cmp    $0x6,%r14d
0x0000000000401174 <+99>:    je     0x40117b <phase_6+106>
0x0000000000401176 <+101>:   mov    %r14d,%ebx
0x0000000000401179 <+104>:   jmp    0x401146 <phase_6+53>
```

Note that there's two backward jumps, so it might be a loop. After the first two lines, `eax` is holding our first integer, then, it got decremented and compared with 5. if it is more than 5, we jump to the line 38, which is `explode_bomb`, otherwise, we continue with the next lines. Initially, `r14d` is 0, now it is incremented and compared with 6, if they are equal, the loop will terminate, otherwise, the bomb assigns the value to `ebx` and continues the loop.

```
0x0000000000401146 <+53>:    movslq %ebx,%rax
0x0000000000401149 <+56>:    mov    0x30(%rsp,%rax,4),%eax
0x000000000040114d <+60>:    cmp    %eax,0x0(%rbp)
0x0000000000401150 <+63>:    jne    0x40113e <phase_6+45>
0x0000000000401152 <+65>:    callq  0x401451 <explode_bomb>
0x0000000000401157 <+70>:    jmp    0x40113e <phase_6+45>
0x0000000000401159 <+72>:    add    $0x4,%r13
```

Let's inspect what's residing in `0x30(%rsp,%rax,4)` and `%rbp`

```
(gdb) x/d $rbp
0x7fffffffe1f0: 1
(gdb) x/d $rsp+$rax*4+0x30
0x7fffffffe1f4: 2
```

which is our first and the next number. If they are equal, the bomb will explode, otherwise we go to the line 45

```
0x000000000040113e <+45>:    add    $0x1,%ebx
0x0000000000401141 <+48>:    cmp    $0x5,%ebx
0x0000000000401144 <+51>:    jg     0x401159 <phase_6+72>
```

Here, `ebx` is incremented and compared with 5, if it's greater than 5, we go to the line 72, otherwise, continue with the line 53 like above, which continues the loop.
This behavior suggest a nested loop, which can be roughly translated into C as

```
// Let's suppose that a[6] = {our 6 numbers}
int i = 0, j = 0;
while(i < 6){
  if(a[i] > 6) explode_bomb();
  j = i;
  while(j < 6){
```

```
        if(a[i] == a[j]) explode_bomb();
        j++;
      }
      i++;
    }
```

In conclusion, this part is for checking whether all the numbers in the input is less than or equal to 6 and all of them are pairwise distinct, i.e. a permutation from 1 to 6. Since our dummy input satisfies this constraint, we can pass this stage.

```
0x000000000040117b <+106>:   lea    0x18(%r12),%rcx
0x0000000000401180 <+111>:   mov    $0x7,%edx
0x0000000000401185 <+116>:   mov    %edx,%eax
0x0000000000401187 <+118>:   sub    (%r12),%eax
0x000000000040118b <+122>:   mov    %eax,(%r12)
0x000000000040118f <+126>:   add    $0x4,%r12
0x0000000000401193 <+130>:   cmp    %r12,%rcx
0x0000000000401196 <+133>:   jne    0x401185 <phase_6+116>
0x0000000000401198 <+135>:   mov    $0x0,%esi
0x000000000040119d <+140>:   jmp    0x4011b8 <phase_6+167>
```

This also has a backward jump, so it has a high chance to be Note that `%r12` is the address of our first number, so `%rcx` is the address that is after our last number.

```
(gdb) x/8d $r12
0x7fffffffe1f0: 1       2       3       4
0x7fffffffe200: 5       6       6306096 0
```

Next, `edx` will be equal to `0x7`, and then be assigned to `eax`. Then, `eax` will be subtracted by `r12`, which is our first number, and it will be moved back to `r12`. Finally. `r12` is incremented by 4, to our second integer. In conlusion, our 6 numbers are modified by the following function

$$f(x) = 7 - x,$$

where x is our input numbers. After this, our input will become $\{6, 5, 4, 3, 2, 1\}$. Now, let's continue with our bomb.

```
0x00000000004011b8 <+167>:   mov    0x30(%rsp,%rsi,4),%ecx
0x00000000004011bc <+171>:   mov    $0x1,%eax
0x00000000004011c1 <+176>:   mov    $0x6032f0,%edx
0x00000000004011c6 <+181>:   cmp    $0x1,%ecx
0x00000000004011c9 <+184>:   jg     0x40119f <phase_6+142>
0x00000000004011cb <+186>:   jmp    0x4011aa <phase_6+153>
```

Let's see what's inside the address `0x30(%rsp,%rsi,4)`

```
(gdb) x/8d (0x30+$rsp + $rsi * 4)
0x7fffffffe1f0: 6       5       4       3
0x7fffffffe200: 2       1       6306096 0
```

which is our input. This lines of codes can be translated to C as follow

```
// Let a[6] = {our input numbers}
ecx = a[0];
eax = 1;
edx = *0x6032f0;
if(ecx > 1){
  // Line 142
} else{
  // Line 153
}
```

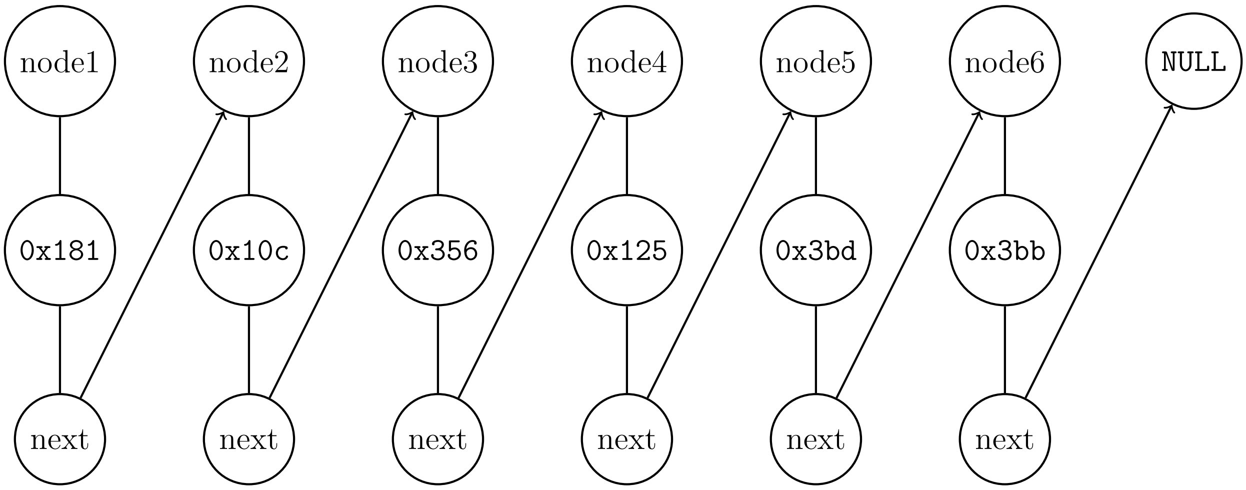
Let's inspect `0x6032f0`

```
(gdb) x 0x6032f0
0x6032f0 <node1>:       385
```

What an interesting name, `node1`. This suggest that it could be some data structures that related to nodes such as linked list, binary tree. Let's examine the address around it

```
(gdb) x/30x 0x6032f0
0x6032f0 <node1>: 0x00000181 0x00000001 0x00603300 0x00000000
0x603300 <node2>: 0x0000010c 0x00000002 0x00603310 0x00000000
0x603310 <node3>: 0x00000356 0x00000003 0x00603320 0x00000000
0x603320 <node4>: 0x00000125 0x00000004 0x00603330 0x00000000
0x603330 <node5>: 0x000003bd 0x00000005 0x00603340 0x00000000
0x603340 <node6>: 0x000003bb 0x00000006 0x00000000 0x00000000
0x603350 <bomb_id>: 0x00000034 0x00000000 0x00000000 0x00000000
0x603360 <host_table>: 0x00402589 0x00000000
```

From this, we know that there are 6 nodes, and each of them as 4 attributes. However, the last attribute is all 0, so it does not serve any practical purpose. Hence, there's only 3 active attributes. The second column is ranging from 1 to 6, so it might be the node's id. The third one looks like address values. Further investigation shows that it actually the address of the next node. The below is the diagram of the nodes



This suggests the follow definition for a linked list data structure

```
struct node{
  int val, id;
  struct node *link;
};
```

Let's go back to our code, note that there's two backward jumps again, so it must be another loop.

```
if(ecx > 1){
  // Line 142
}
// Line 153
```

Let's go to the line 142

```
0x000000000040119f <+142>:    mov    0x8(%rdx),%rdx
0x00000000004011a3 <+146>:    add    $0x1,%eax
0x00000000004011a6 <+149>:    cmp    %ecx,%eax
0x00000000004011a8 <+151>:    jne    0x40119f <phase_6+142>
```

Let's see that's inside `rdx`

```
(gdb) x/3 $rdx
0x6032f0 <node1>: 0x00000181 0x00000001 0x00603300
```

which is the first node in our linked list, and `0x8(%rdx)` is the address of the next node, to which we move. Then, `eax` is incremented, and compared with `ecx`, if it is not equal, loop back to the line 142, if they are equal continue to line 153. These lines are finding the node in the position specified in our inputs.

```
0x00000000004011aa <+153>:   mov    %rdx,(%rsp,%rsi,8)
0x00000000004011ae <+157>:   add    $0x1,%rsi
0x00000000004011b2 <+161>:   cmp    $0x6,%rsi
0x00000000004011b6 <+165>:   je     0x4011cd <phase_6+188>
```

First, it moves the value of our current node into the address `(%rsp,%rsi,8)`, and then increment `rsi`, then continues the loop until `rsi` = 6. After the loop is terminated, we go into the line 188.

```
0x00000000004011cd <+188>:   mov    (%rsp),%rbx
0x00000000004011d1 <+192>:   mov    0x8(%rsp),%rax
0x00000000004011d6 <+197>:   mov    %rax,0x8(%rbx)
0x00000000004011da <+201>:   mov    0x10(%rsp),%rdx
0x00000000004011df <+206>:   mov    %rdx,0x8(%rax)
0x00000000004011e3 <+210>:   mov    0x18(%rsp),%rax
0x00000000004011e8 <+215>:   mov    %rax,0x8(%rdx)
0x00000000004011ec <+219>:   mov    0x20(%rsp),%rdx
0x00000000004011f1 <+224>:   mov    %rdx,0x8(%rax)
0x00000000004011f5 <+228>:   mov    0x28(%rsp),%rax
0x00000000004011fa <+233>:   mov    %rax,0x8(%rdx)
0x00000000004011fe <+237>:   movq   $0x0,0x8(%rax)
0x0000000000401206 <+245>:   mov    $0x5,%ebp
0x000000000040120b <+250>:   jmp    0x401216 <phase_6+261>
```

Note that `rsp` here is holding our values of the linked list. Let's inspect them, using the following custom function in `gdb`.

```
define pa
  set var $i = 1
  while $i <= 6
    printf "a[%d] = %x\n", $i, **(int*)($arg0 + 0x8*($i-1))
    set var $i = $i + 1
  end
end
```

Using this custom command with an argument of `rsp`, we obtain the following

```
(gdb) pa $rsp
a[1] = 0x3bb
a[2] = 0x3bd
a[3] = 0x125
a[4] = 0x356
a[5] = 0x10c
a[6] = 0x181
```

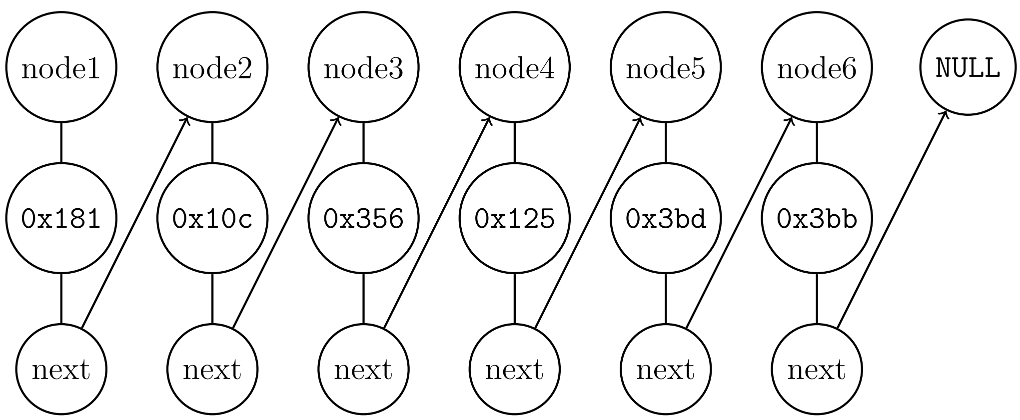
The value now is shuffled according to our modified input, i.e. "6 5 4 3 2 1". Meaning, the first value is the value of `node6`, the second value is the value of `node5`, and so on. These lines just move all the data from `rsp` to `rbx`. Now, let's see the last few lines

```
0x000000000040120b <+250>:   jmp    0x401216 <phase_6+261>
0x000000000040120d <+252>:   mov    0x8(%rbx),%rbx
0x0000000000401211 <+256>:   sub    $0x1,%ebp
0x0000000000401214 <+259>:   je     0x401227 <phase_6+278>
0x0000000000401216 <+261>:   mov    0x8(%rbx),%rax
0x000000000040121a <+265>:   mov    (%rax),%eax
0x000000000040121c <+267>:   cmp    %eax,(%rbx)
0x000000000040121e <+269>:   jge    0x40120d <phase_6+252>
0x0000000000401220 <+271>:   callq  0x401451 <explode_bomb>
0x0000000000401225 <+276>:   jmp    0x40120d <phase_6+252>
```

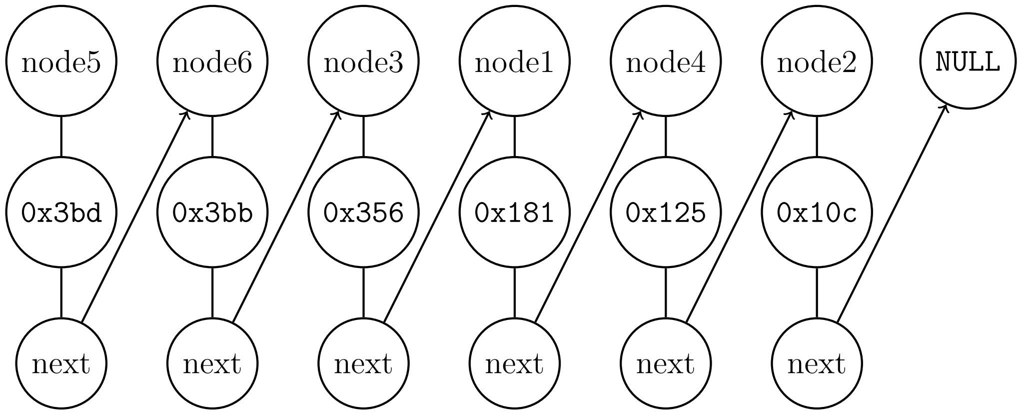
Firstly, we start at the line 261. Here, `eax` is holding our second value, i.e. `a[2]`. Then, the bomb compares `a[1]` and `a[2]`, if $a[1] \geq a[2]$ then we continue with our loop, otherwise, the bomb will explode. In the next iterations, the bomb also compares the current value with the next value, i.e. it is checking whether $a[i] \geq a[i+1]$, if there exists an i that violates the condition, the bomb will detonate, otherwise, it jumps peacefully to the line 278, which is the end of our function.

```
0x0000000000401227 <+278>:   add    $0x50,%rsp
0x000000000040122b <+282>:   pop    %rbx
0x000000000040122c <+283>:   pop    %rbp
0x000000000040122d <+284>:   pop    %r12
0x000000000040122f <+286>:   pop    %r13
0x0000000000401231 <+288>:   pop    %r14
0x0000000000401233 <+290>:   retq
```

In conclusion, the last few lines are checking if our values from the array "a" is nonincreasing. If it is, then we defuse the bomb, otherwise, the bomb will detonate. In our case of dummy input, the bomb will detonate after it detects $a[1] < a[2]$, which violates the condition. Therefore, we have to find a permutation of $\{1, 2, 3, 4, 5, 6\}$ so that our final "array" is nonincreasing. Recall that the original values of the nodes are



After sorting, the nodes will be in the following order



Thus, the correct permutation should be $\{5, 6, 3, 1, 4, 2\}$.However, note that before shuffling the linked list, our input is modified by the function

$$f(x) = 7 - x$$

Therefore, the correct input should be "`2 1 4 6 3 5`". Resetting the bomb and typing in the correct string, we have successfully defused the entire bomb!

```
[edusc03-052@cheetah022 bomb52]$ ./bomb input.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
2 1 4 6 3 5
Congratulations! You've defused the bomb!
```