Malware Analysis & Reverse Engineering Midterm Report

Malware Bomb

Xinyu Lyu(xl422)

At first, I found that the malware Bomb is unpacked when I tried to find the Strings in Ubuntu.

```
yingyue@hkx-OptiPlex-5040:~/Desktop$ file ./malware-bomb
./malware-bomb: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynami
cally linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.0.0, not stripped
yingyue@hkx-OptiPlex-5040:~/Desktop$
```

Therefore, I directly open the malware-bomb in Ubuntu terminal. And Game Sta

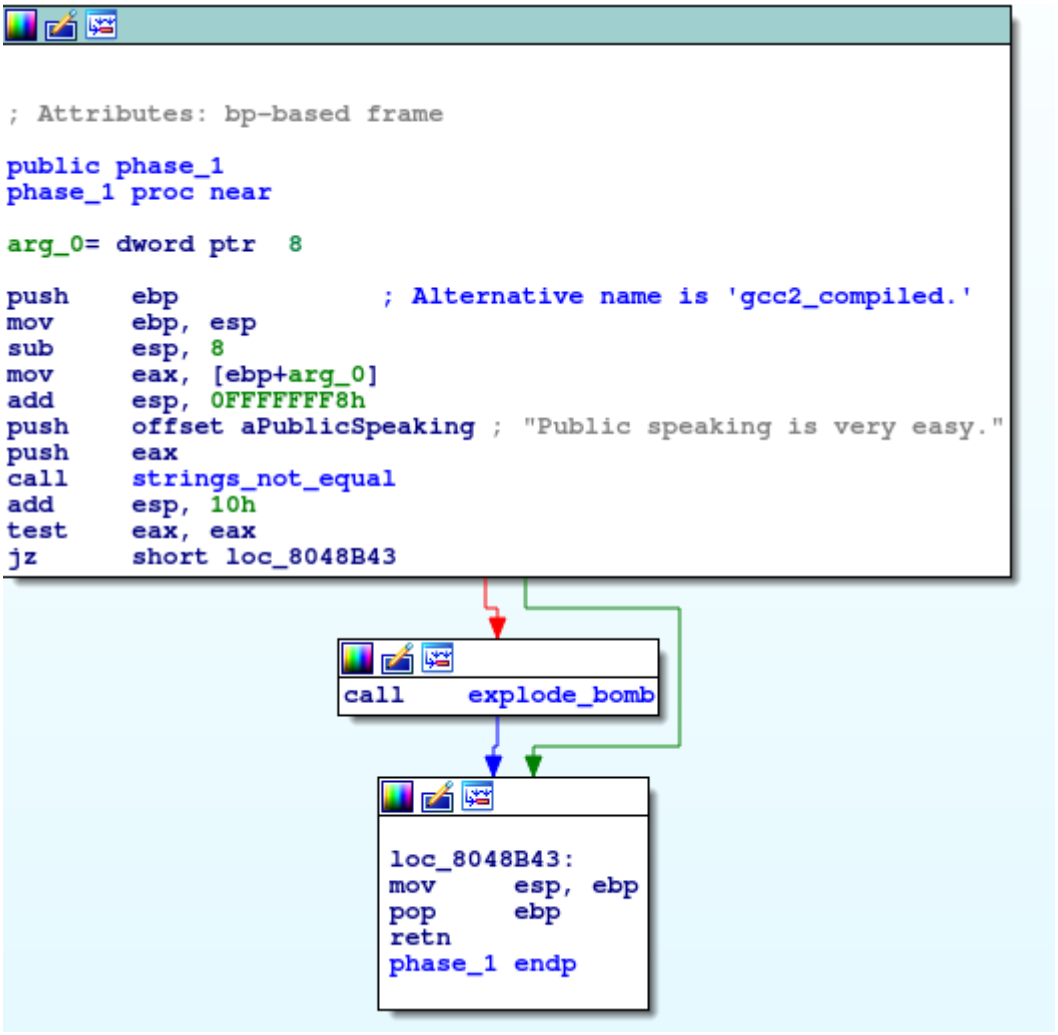
```
yingyue@hkx-OptiPlex-5040:~/Desktop/malware$ ./malware-bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

Next, I load the bomb in IDA Pro, and it took 6 steps (from phase_1 to phase_6)for me to diffuse the Bomb.

```
loc_8048A30:
call    initialize_bomb
add     esp, 0FFFFFFF4h
push    offset aWelcomeToMyFie ; "Welcome to my fiendish little bomb. You"...
call    _printf
add     esp, 0FFFFFFF4h
push    offset aWhichToBlowYou ; "which to blow yourself up. Have a nice "...
call    _printf
add     esp, 20h
call    read_line
add     esp, 0FFFFFFF4h
push    eax
call    phase_1
call    phase_defused
add     esp, 0FFFFFFF4h
push    offset aPhase1DefusedH ; "Phase 1 defused. How about the next one"...
call    _printf
add     esp, 20h
call    read_line
add     esp, 0FFFFFFF4h
push    eax
call    phase_2
call    phase_defused
add     esp, 0FFFFFFF4h
push    offset aThatSNumber2Ke ; "That's number 2.  Keep going!\n"
call    _printf
add     esp, 20h
call    read_line
add     esp, 0FFFFFFF4h
push    eax
call    phase_3
call    phase_defused
add     esp, 0FFFFFFF4h
push    offset aHalfwayThere ; "Halfway there!\n"
call    _printf
add     esp, 20h
call    read_line
add     esp, 0FFFFFFF4h
push    eax
call    phase_4
call    phase_defused
add     esp, 0FFFFFFF4h
push    offset aSoYouGotThatOn ; "So you got that one.  Try this one.\n"
call    _printf
add     esp, 20h
call    read_line
add     esp, 0FFFFFFF4h
push    eax
call    phase_5
call    phase_defused
add     esp, 0FFFFFFF4h
push    offset aGoodWorkOnToTh ; "Good work!  On to the next...\n"
call    _printf
add     esp, 20h
call    read_line
add     esp, 0FFFFFFF4h
push    eax
call    phase_6
```

Phase 1:

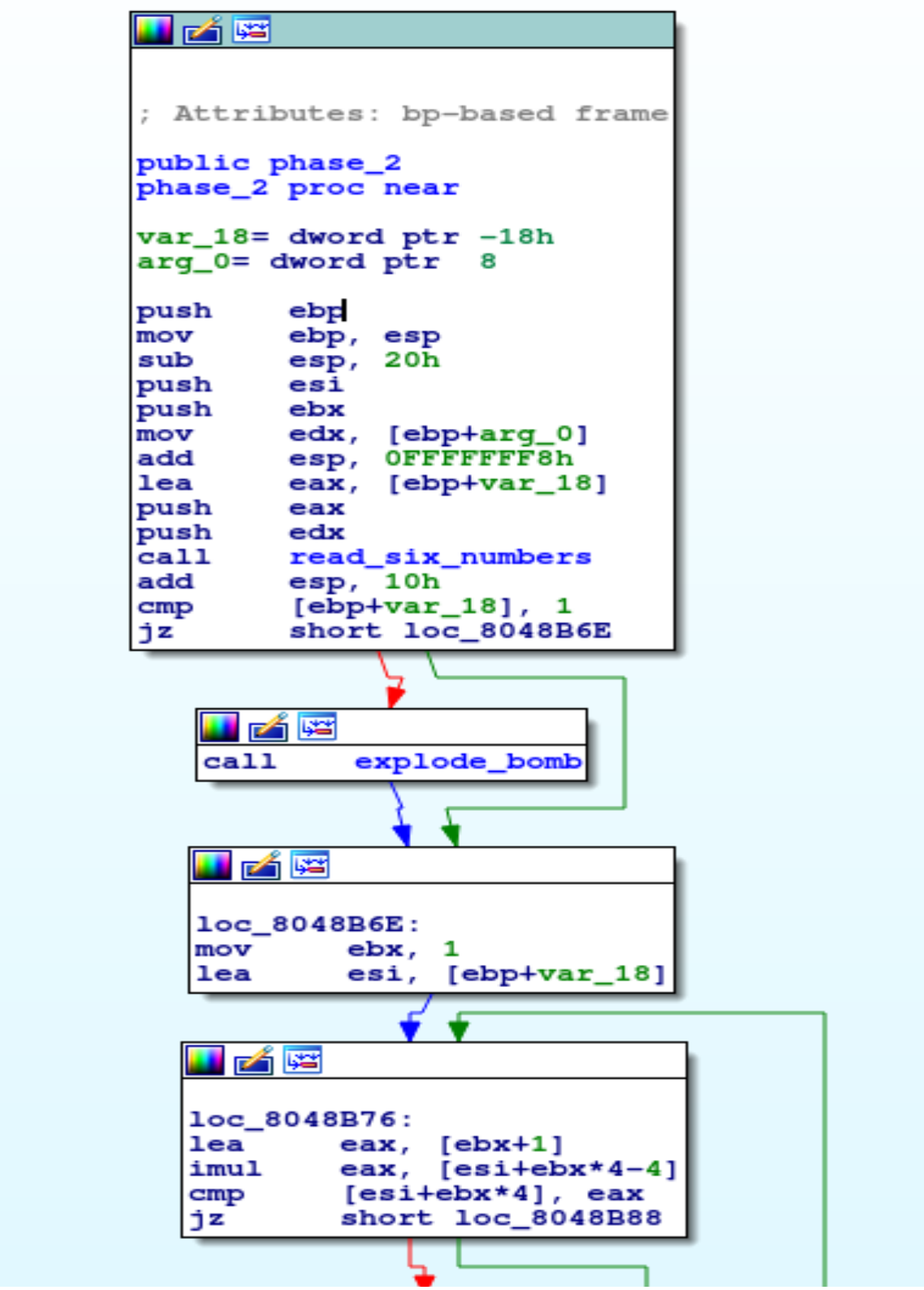
After entering the phase_1, I found a string “Public speaking is very easy.” which seems to be the answer to phase_1. Therefore, I tried it and succeeded.

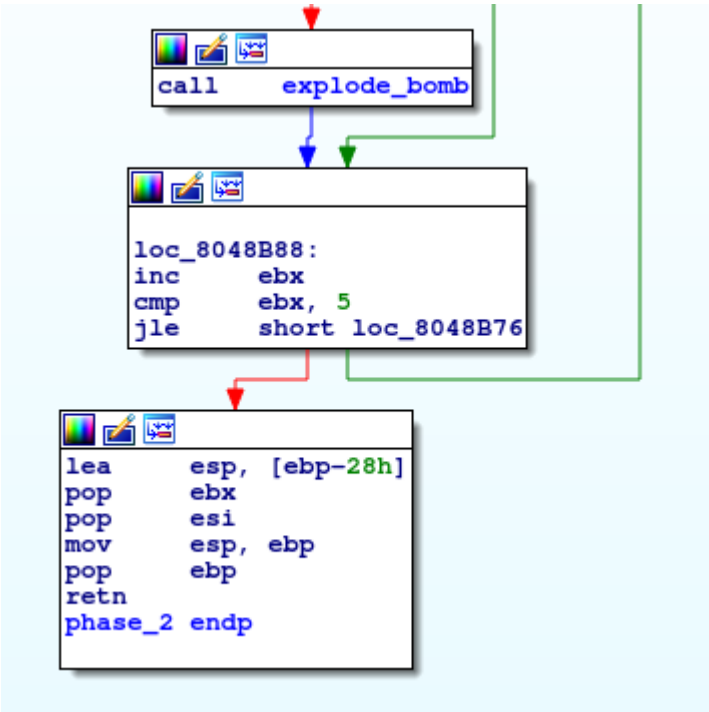



```
yingyue@hkx-OptiPlex-5040:~/Desktop/malware$ ./malw
Welcome to my fiendish little bomb. You have 6 phas
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
```

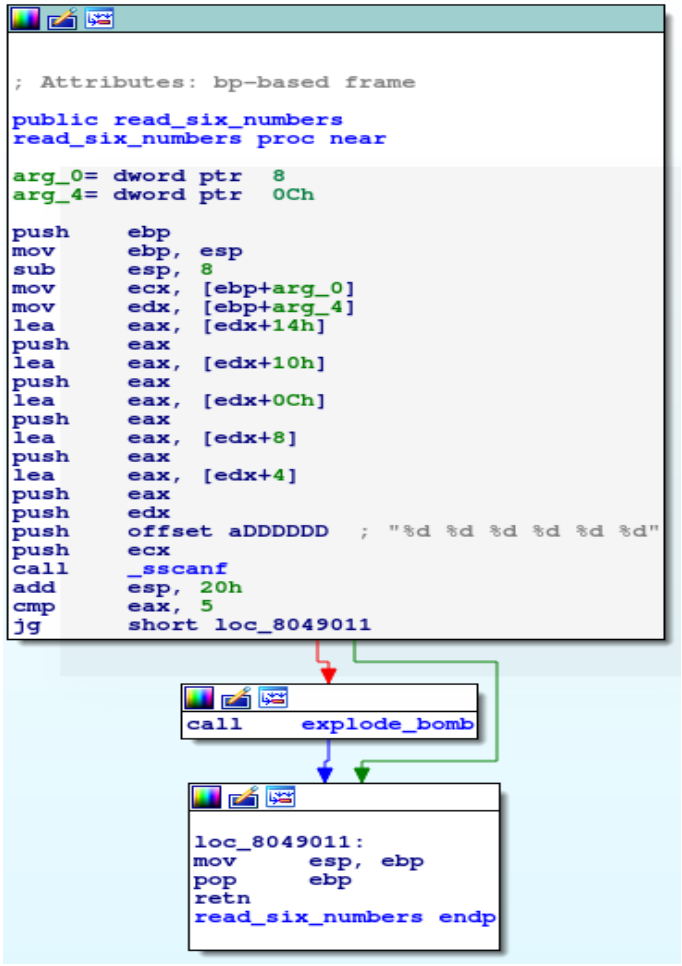
Phase2:

Then, I entered phase_2:





In phase_2, I entered the read_six_numbers function.

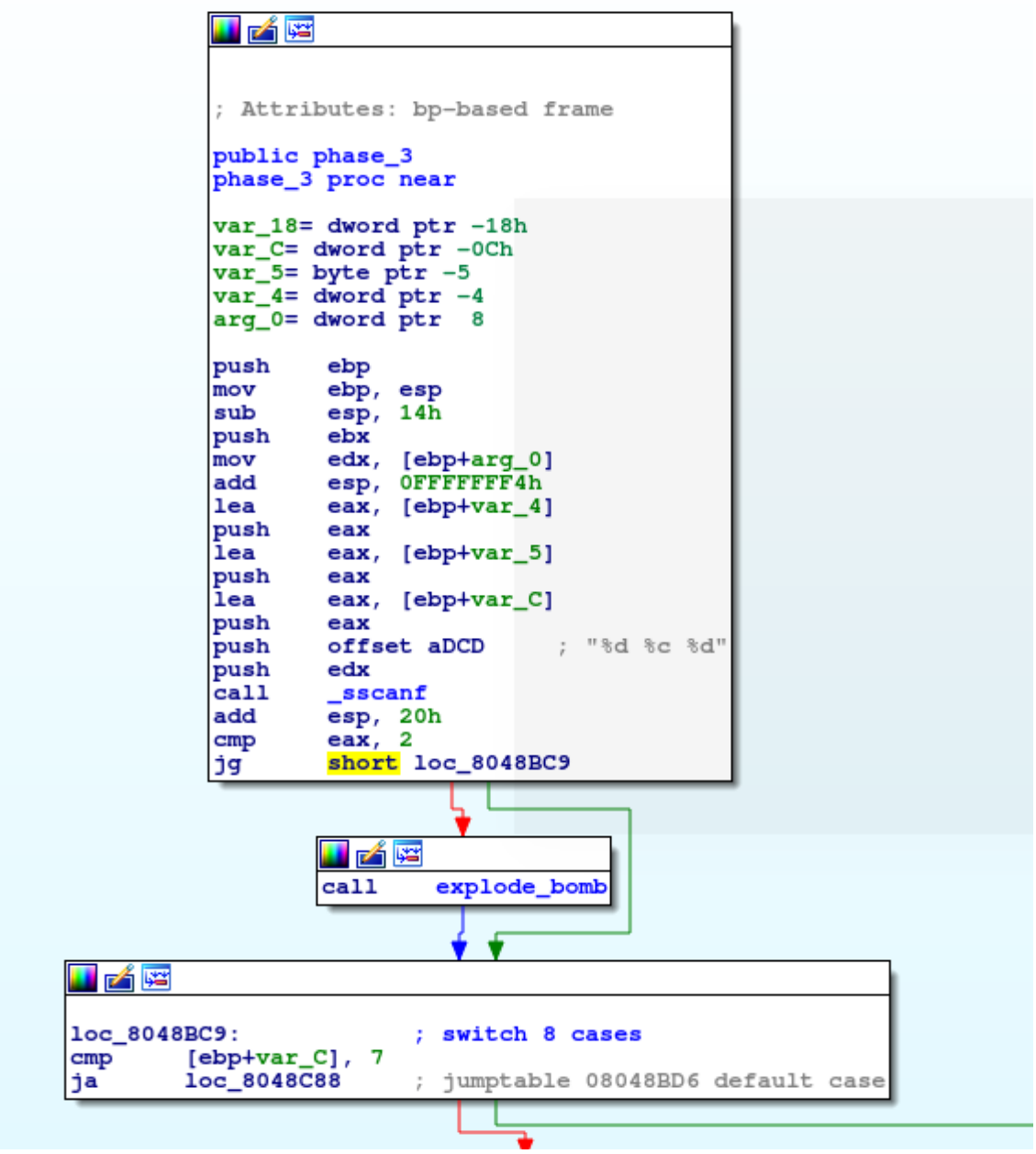


After understanding the whole function flow, I draw the conclusion of the solution to phase_2.
First, it takes an array of 6 numbers as input sequence. Then, it store the numbers in the integer array var_18. Moreover, the first number should be 1 and each input number should be (i+1) i∈[1,5] times larger than the previous input number.
Therefore, I tried a sequence of numbers as input [1,2,6,24,120,720], and it works.

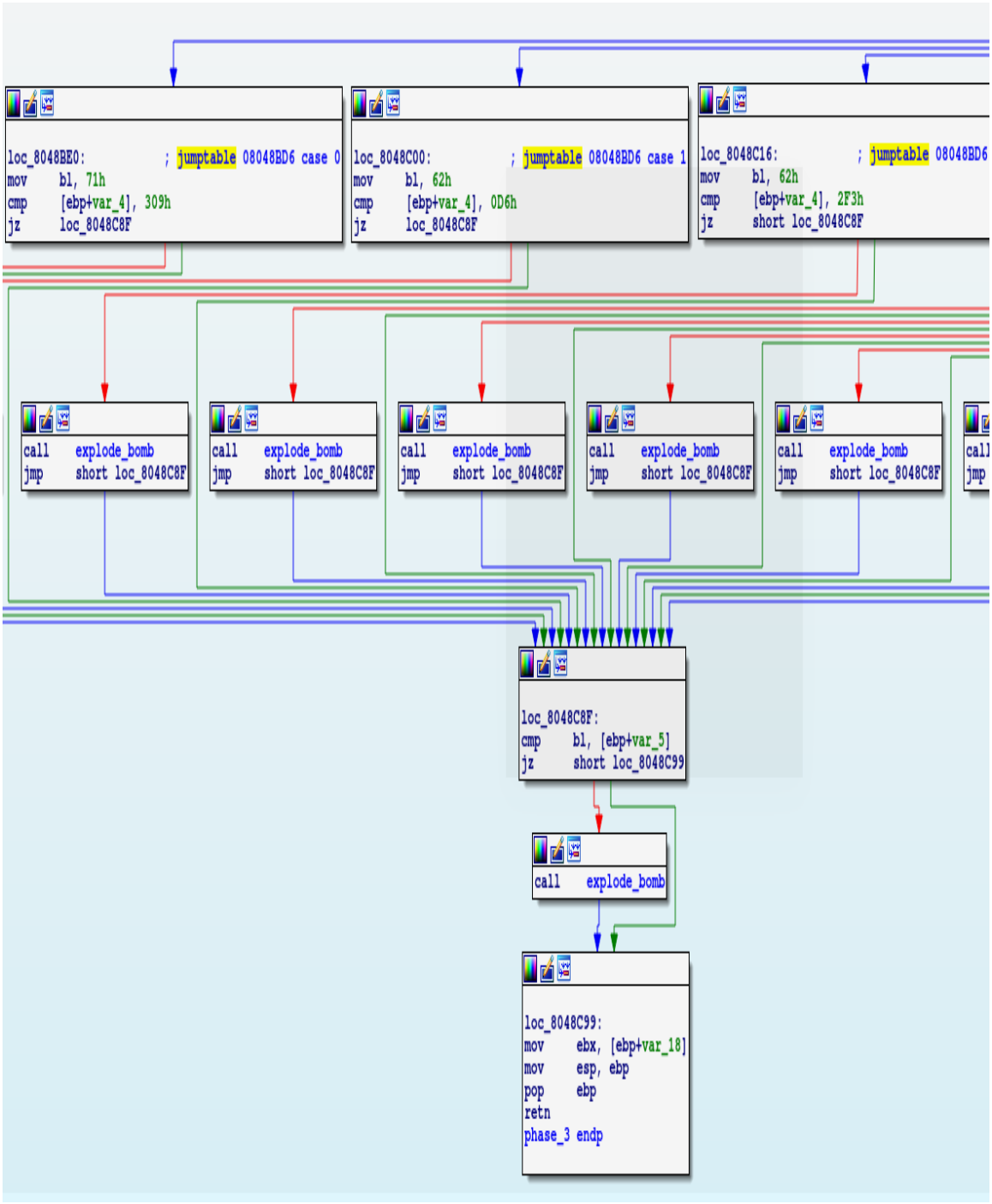
```
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
```

Phase 3:

Then, I entered phase_3 and it looks huge because of 8 cases switch:



When you switch to each case, the case inside determines which character and number should be the valid input pair. For example, for case 0, the valid input should be [q,777].

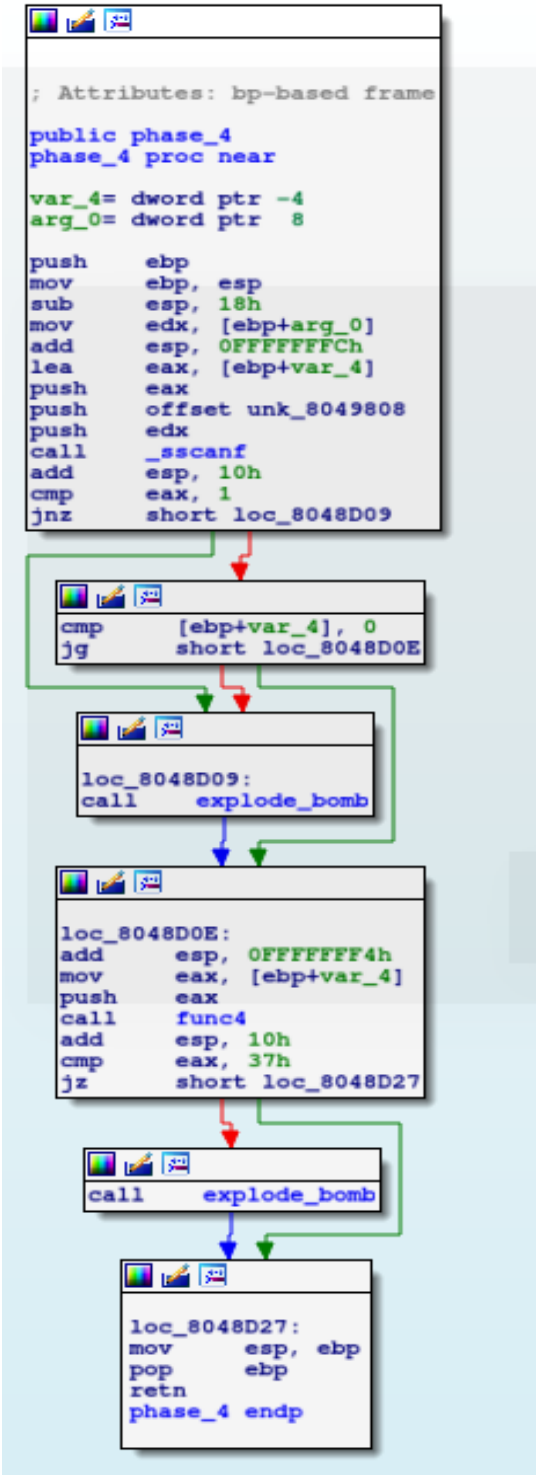


Therefore, I switch to case 0 with ‘q’ as input character and 777 as integer input. Fortunately, it works.

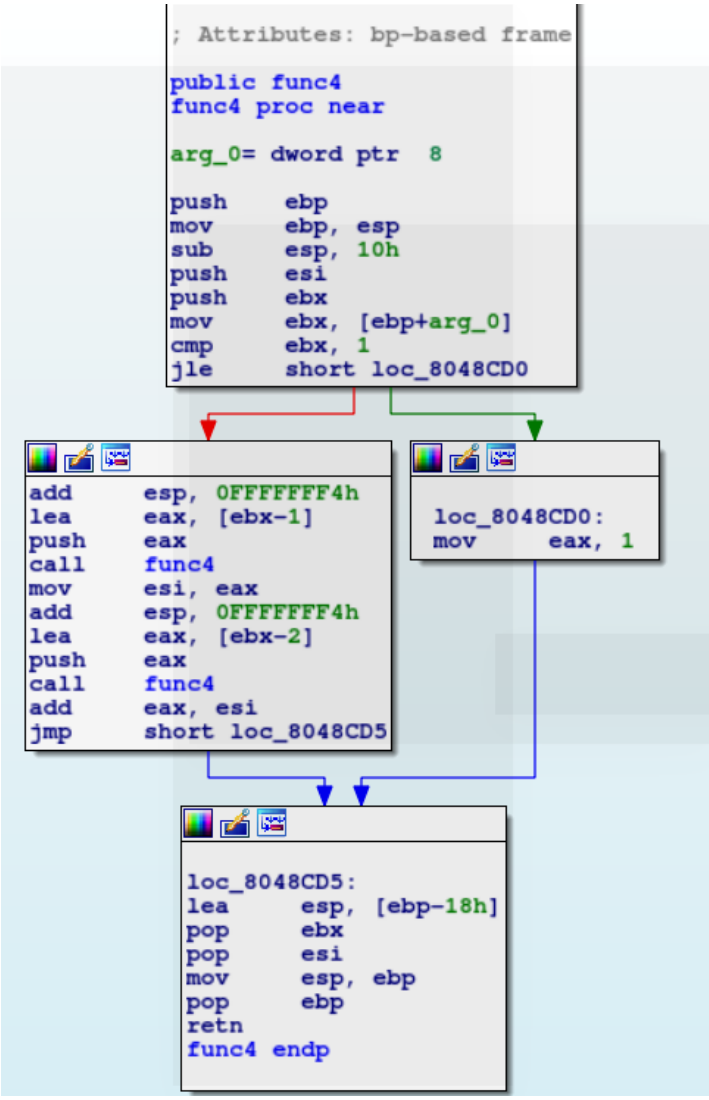
```
That's number 2.  Keep going!
0 q 777
Halfway there!
```

Phase 4:

Then, I entered phase_4:



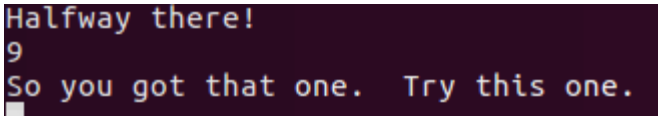
In phase_4, firstly, it takes a integer as input. Then, it passes the input to function 4 as the argument. Then, it make a comparison between the return of function 4 with the 0x37. Next, I look through the function 4 in details.



Obviously, function 4 is a recursive function, for it calls itself a lot. And , it returns 1 and ends the recursive if the argument <=1. After read the codes in details, I found that, actually, it returns a Fibonacci sequence. And I list the [input, output] pairs below.

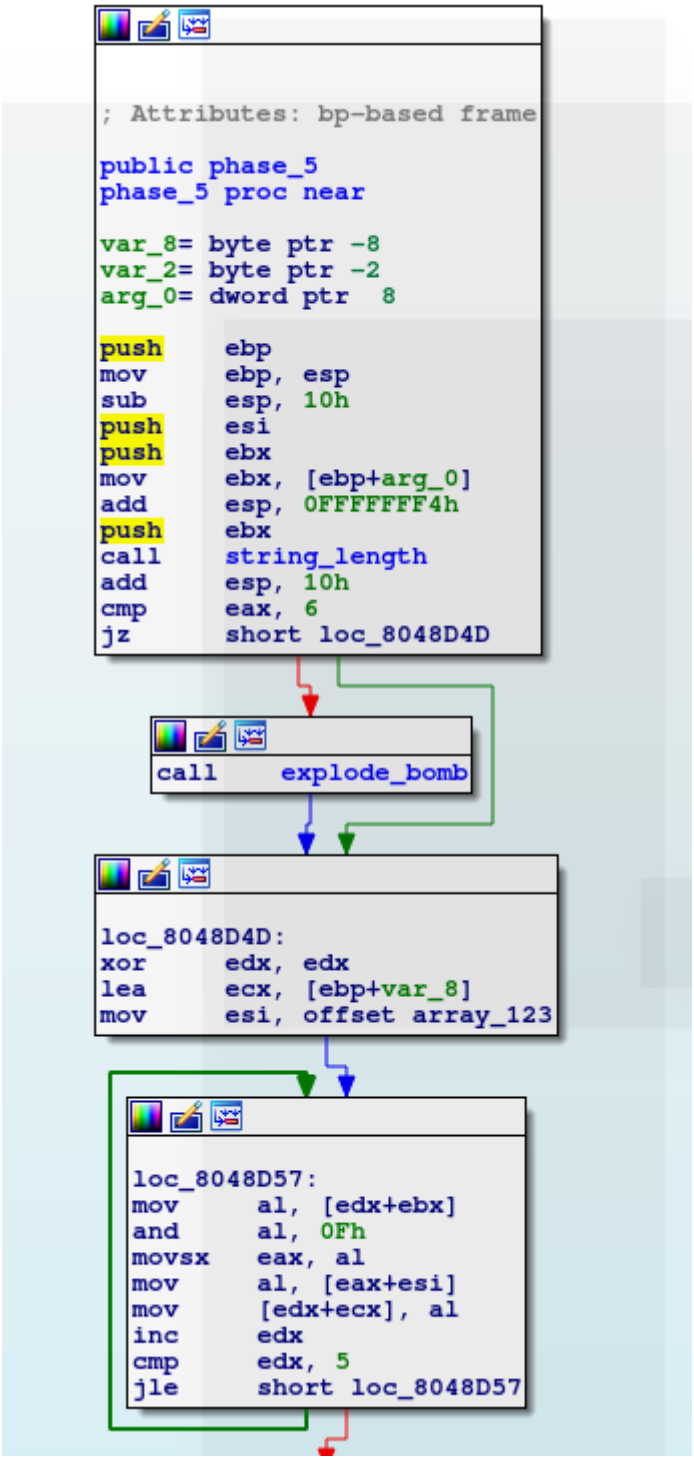
1 1
2 2
3 3
4 5
5 8
6 13
7 21
8 34
9 55

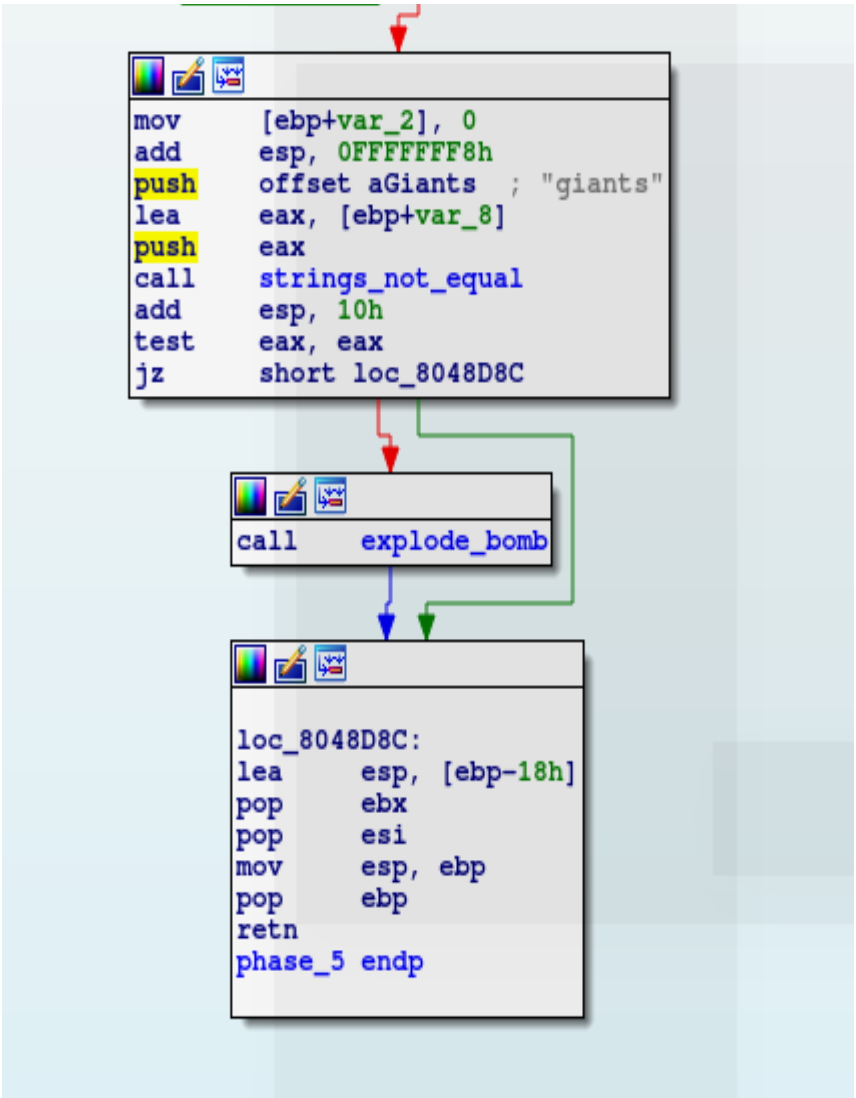
And in phase_4, at last, it makes a comparison between the return of function 4 with the 0x37. ANd 0x37 is exactly 55 in decimal. Therefore, it is easy to get the valid input is 9.



Phase 5:

Then, I entered phase_5:






```
.data:0804B220 array_123       db  69h ; i             ; DATA XREF: phase_5+26↑o
.data:0804B221                 db  73h ; s
.data:0804B222                 db  72h ; r
.data:0804B223                 db  76h ; v
.data:0804B224                 db  65h ; e
.data:0804B225                 db  61h ; a
.data:0804B226                 db  77h ; w
.data:0804B227                 db  68h ; h
.data:0804B228                 db  6Fh ; o
.data:0804B229                 db  62h ; b
.data:0804B22A                 db  70h ; p
.data:0804B22B                 db  6Eh ; n
.data:0804B22C                 db  75h ; u
.data:0804B22D                 db  74h ; t
.data:0804B22E                 db  66h ; f
.data:0804B22F                 db  67h ; g
.data:0804B230                 public node6
```

The phase_5 accepts a string with 6 characters as input. First, it gets the string data from array_123. Then, it iteratively visit all the character input[i] i∈[0,5] of the input. Next, , it finds the position index j of input[i] in array_123. Next, it finds out the printable character X(ascii from 33 to 127), if "X & 0x0f" is equal to j (the position index of input[i] in array_123). Finally, it compares the decoded strings with the "giant".
Actually, I wrote the python codes to find the valid input string.

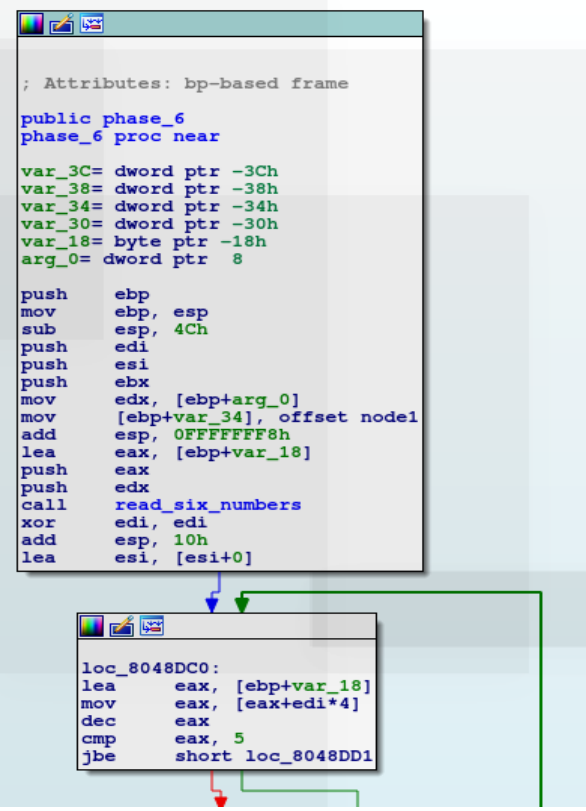
```
array_123 = "isrveawhobpnutfg"
target = "giants"
solution = []
for i in range(len(target)):
      j = array_123.index(target[i])
      for ch in range(33,127):
            if(ch & 0x0f) == j:
                  output.append(chr(ch))
                  break
print (''.join(solution))
```

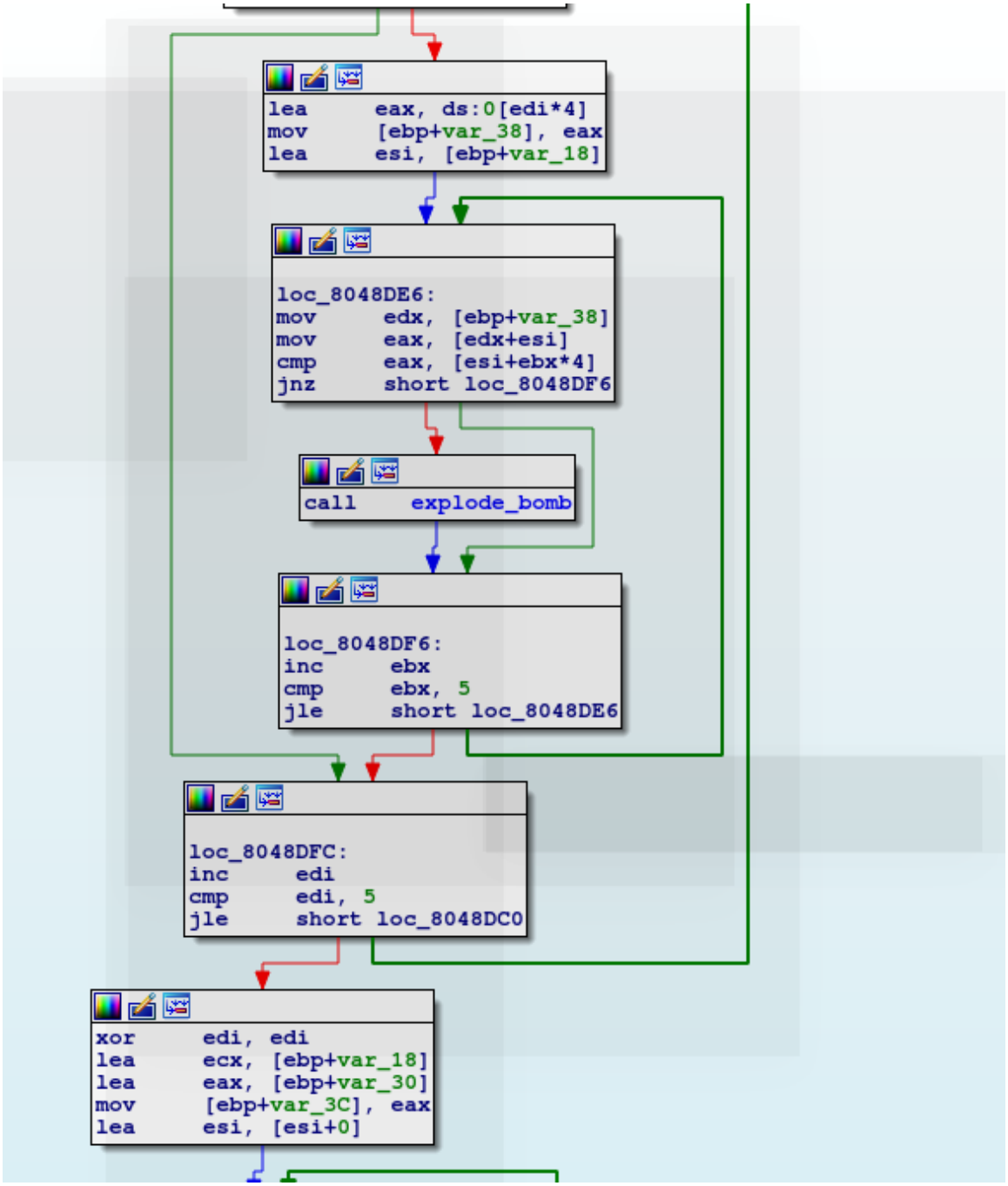
Therefore, after executing the codes, I get the solution to phase_5 is /0%+-!

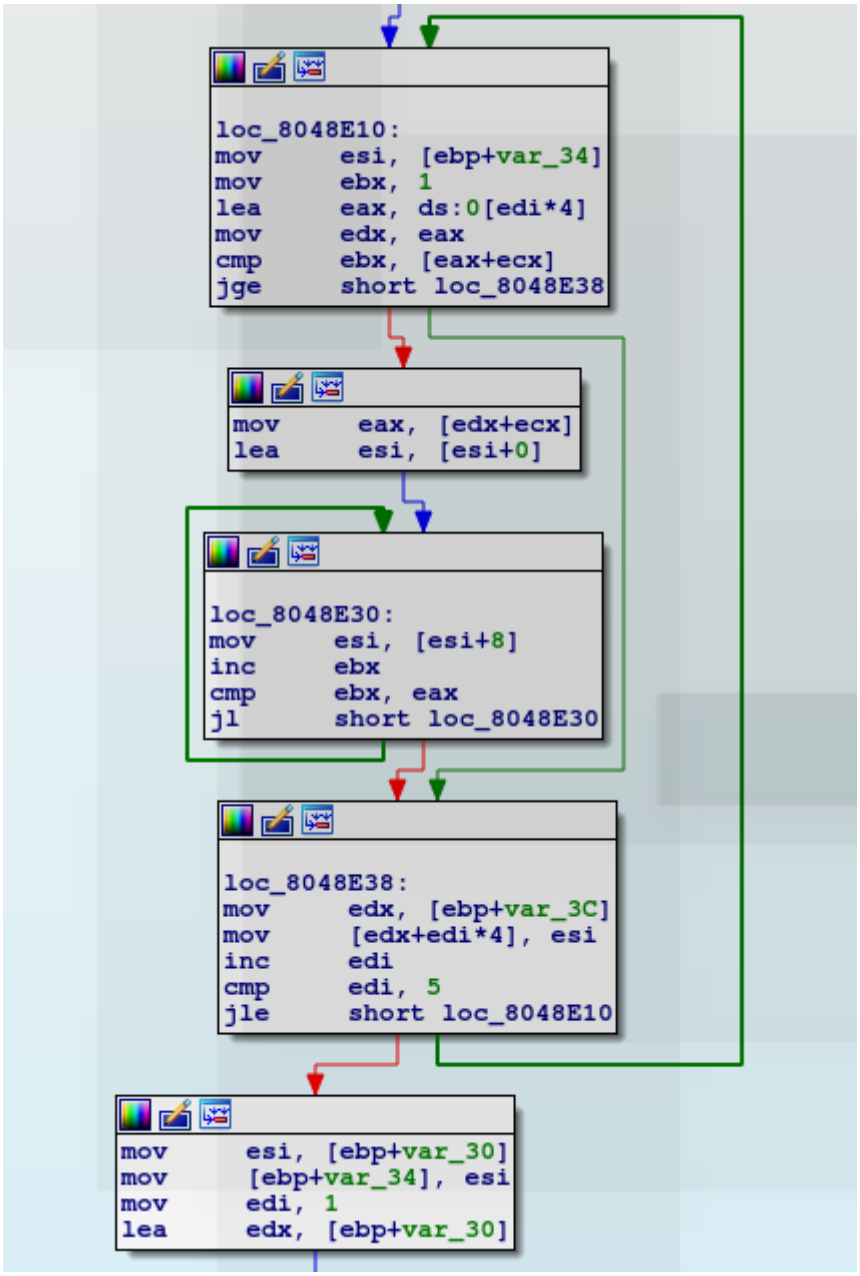
```
So you got that one.  Try this one.
/0%+-!
Good work!  On to the next...
```

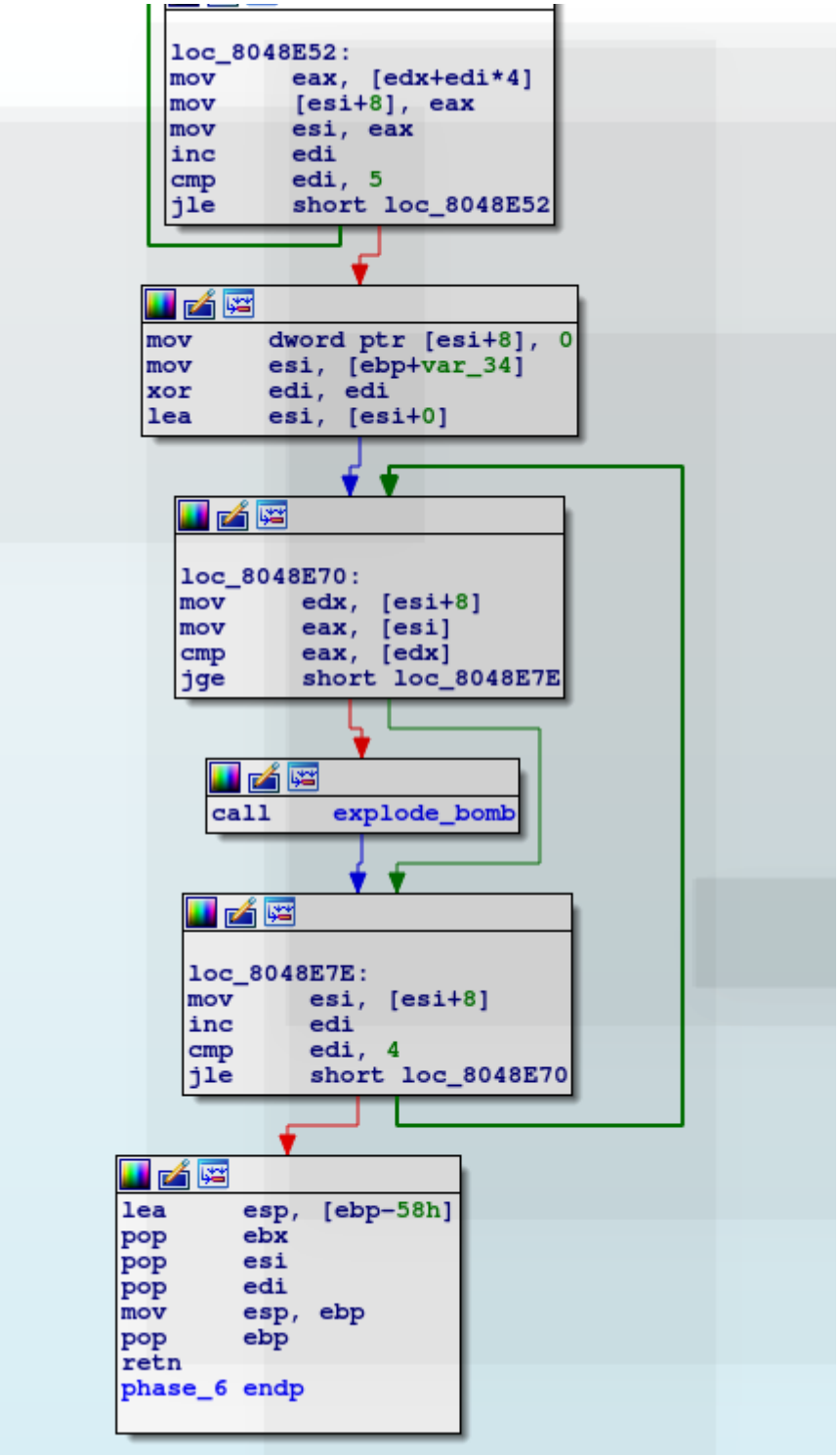
Phase 6:

Finally, i entered the phase_6:




lea eax, ds:0[edi*4]
mov [ebp+var_38], eax
lea esi, [ebp+var_18]
loc_8048DE6:
mov edx, [ebp+var_38]
mov eax, [edx+esi]
cmp eax, [esi+ebx*4]
jnz short loc_8048DF6
call explode_bomb
loc_8048DF6:
inc ebx
cmp ebx, 5
jle short loc_8048DE6
loc_8048DFC:
inc edi
cmp edi, 5
jle short loc_8048DC0
xor edi, edi
lea ecx, [ebp+var_18]
lea eax, [ebp+var_30]
mov [ebp+var_3C], eax
lea esi, [esi+0]


loc_8048E10:
mov esi, [ebp+var_34]
mov ebx, 1
lea eax, ds:0[edi*4]
mov edx, eax
cmp ebx, [eax+ecx]
jge short loc_8048E38
mov eax, [edx+ecx]
lea esi, [esi+0]
loc_8048E30:
mov esi, [esi+8]
inc ebx
cmp ebx, eax
jl short loc_8048E30
loc_8048E38:
mov edx, [ebp+var_3C]
mov [edx+edi*4], esi
inc edi
cmp edi, 5
jle short loc_8048E10
mov esi, [ebp+var_30]
mov [ebp+var_34], esi
mov edi, 1
lea edx, [ebp+var_30]



The phase_6 is a bit hard. It has two loops. And the outer loop visits all the numbers. And if the number is smaller than 1, it get skipped. Then, it adds a node pointing to node1. If the number is bigger than 1, it goes to another loop. In the inner loop, it iterate to get ebx=numbers -1 . Then, it adds the node as the next node in the linkedlist. For Phase 6, I also write codes, but it is too long to put it here.

After running the codes, it helps me to get the solution [4,2,6,3,1,5] to the final phase.