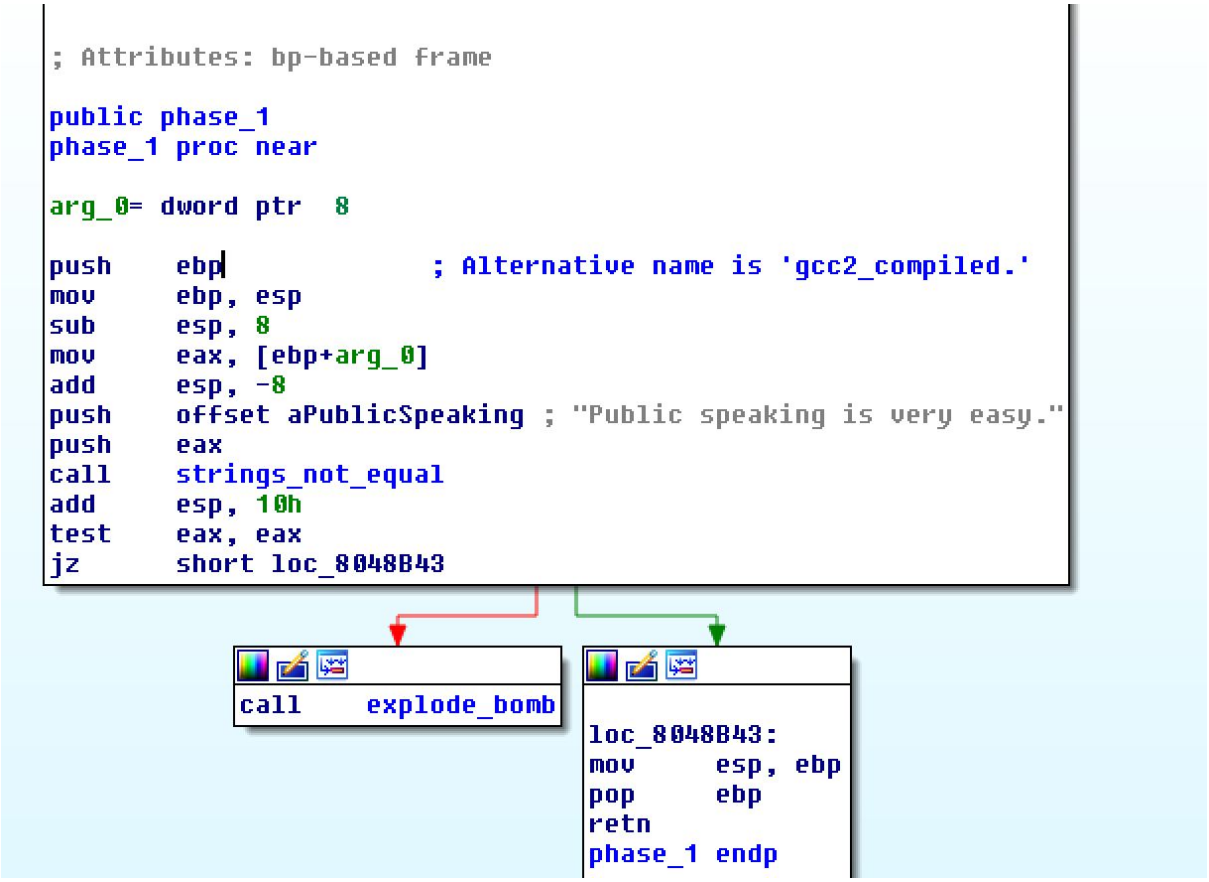
Song Yang
Xin Yang
Zhuohang Li

Report of Homework 4

Phase 1

Phase 1 first takes your input and then stores it in the EAX register. Then push EAX and the string “Public speaking is very easy.” onto a stack and call the string_not_equal function to compare two strings. If they’re equal, you can get to the next phase, otherwise, call explode_bomb.

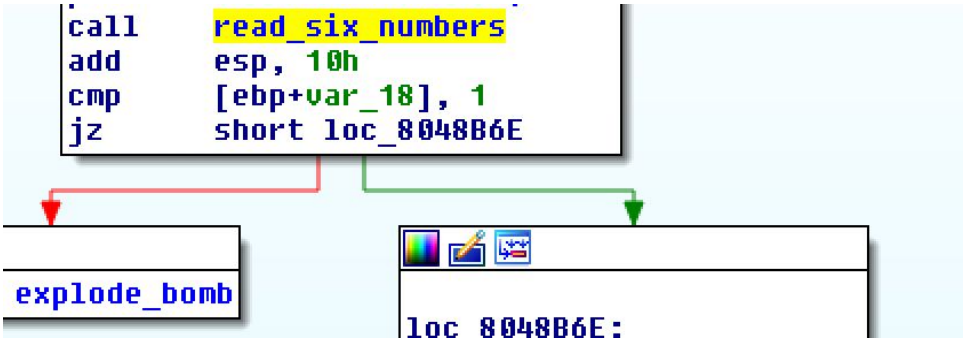


Phase 2

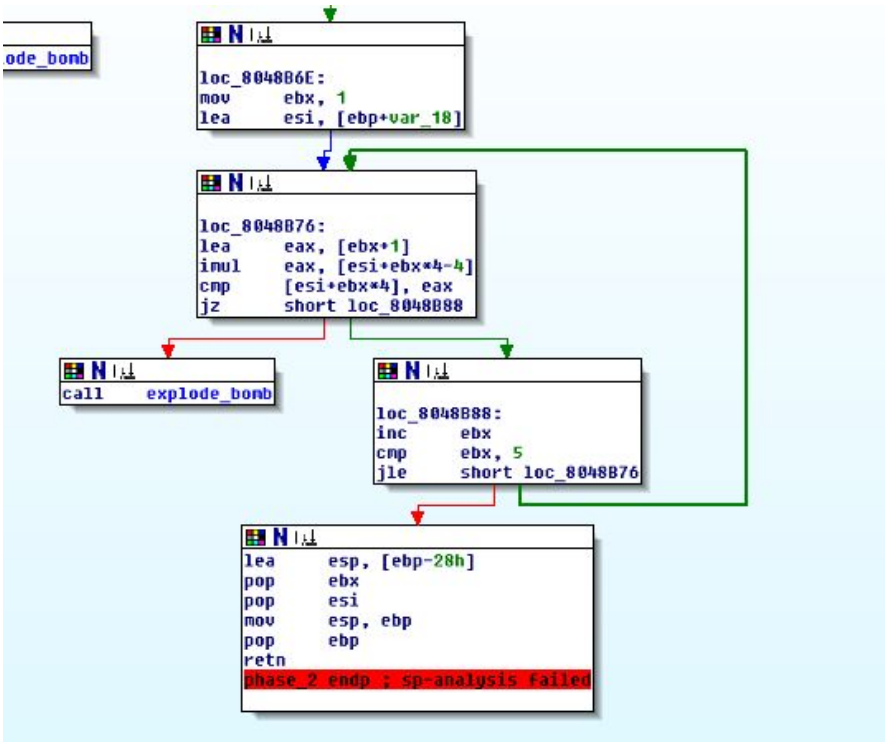
The first function called is read_six_numbers, which reads 6 formatted decimal inputs separated by space.

```
; int __cdecl read_six_numbers(char *s, int)
public read_six_numbers
read_six_numbers proc near

s= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 8
mov     ecx, [ebp+s]
mov     edx, [ebp+arg_4]
lea     eax, [edx+14h]
push    eax
lea     eax, [edx+10h]
push    eax
lea     eax, [edx+0Ch]
push    eax
lea     eax, [edx+8]
push    eax
lea     eax, [edx+4]
push    eax
push    edx
push    offset aDDDDDD  ; "%d %d %d %d %d %d"
push    ecx             ; s
call     sscanf
```

After calling on read_six_numbers, it will check the value of the first input, if the first input is equal to 1 the function will continue to execute, otherwise, the bomb shall explode.



Noticed that there is a loop in phase 2 function.



The EBX, which is the loop counter, is initialized with 1, and increases by 1 until it's larger than 5. For each iteration, the input*(EBX+1) is compared with last input, the code will continue if they're equal. Since the first input must be 1, the input will be a sequence of 1 2 6 24 120 720.

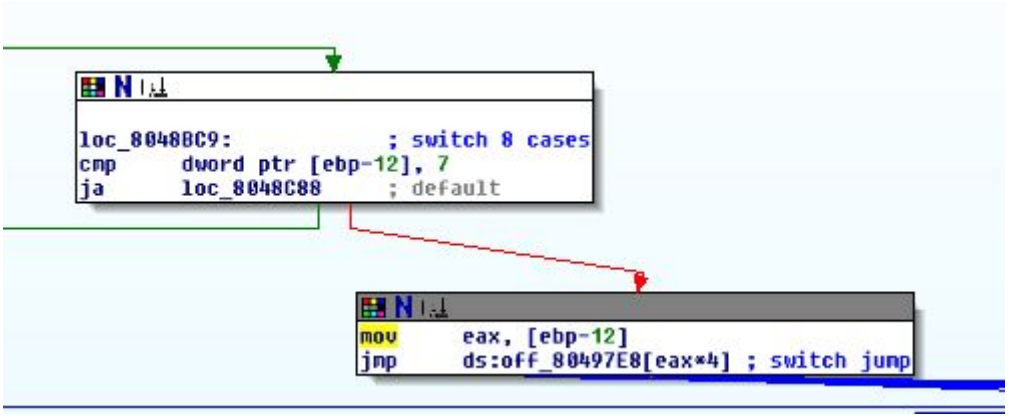
ESI is the head address of the inserted array. So the loop is actually running like the code below.

```
// ESI is an array.

for(ebx = 1;;){
        eax = ebx + 1;
        eax = eax * esi[ebx-1]
        if (esi[ebx] != eax)
                //The bomb exploded
        ebx++;
        if(ebx == 5)
                break
}
```

Phase 3

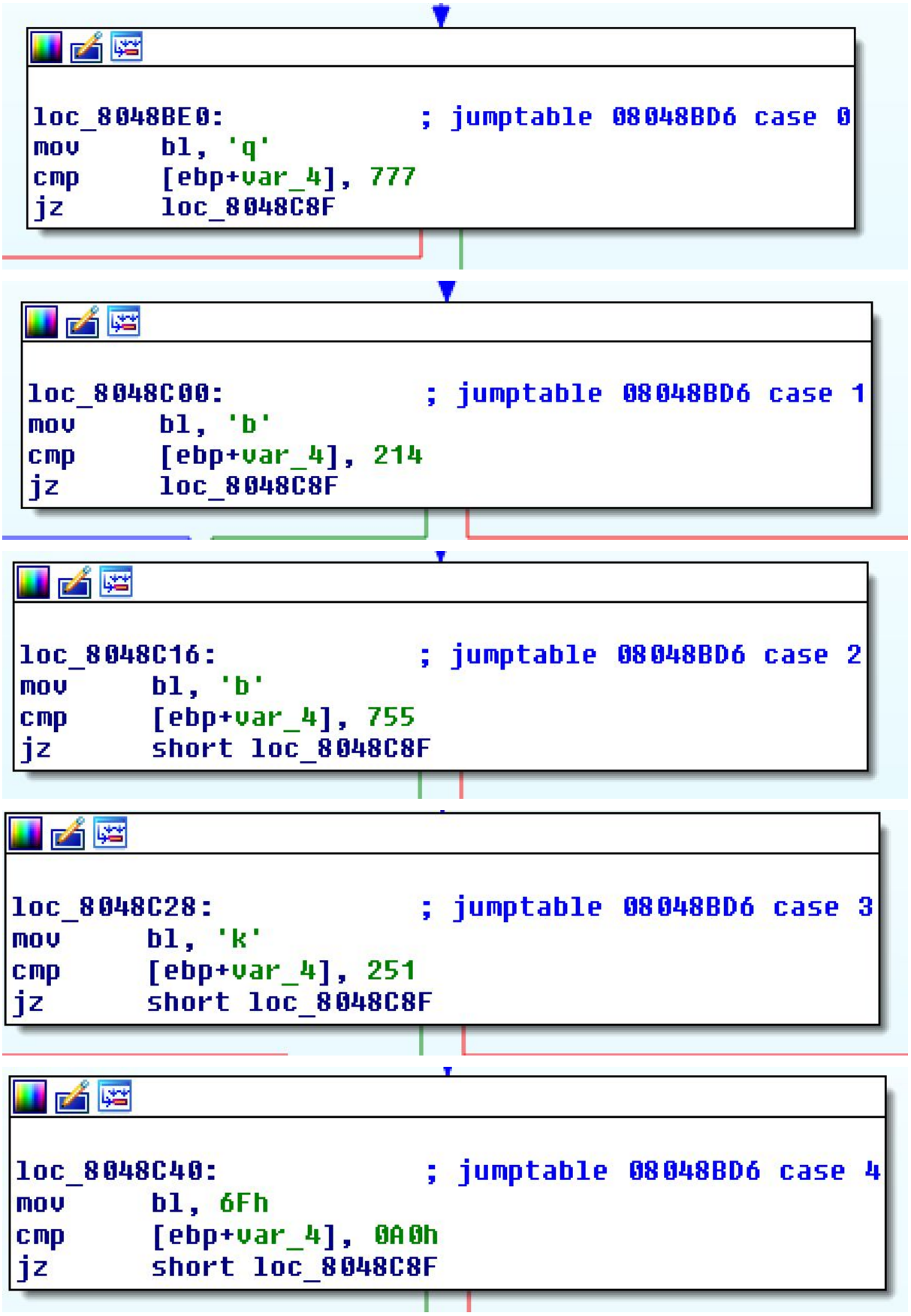
```
; Attributes: bp-based frame

; int __cdecl phase_3(char *s)
public phase_3
phase_3 proc near

var_18= dword ptr -18h
var_C= dword ptr -0Ch
var_5= byte ptr -5
var_4= dword ptr -4
s= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 20
push    ebx
mov     edx, [ebp+8]
add     esp, 0FFFFFFF4h
lea     eax, [ebp-4]
push    eax
lea     eax, [ebp-5]
push    eax
lea     eax, [ebp-12]
push    eax
push    offset aDCD     ; "%d %c %d"
push    edx             ; s
call    _sscanf
add     esp, 20h
cmp     eax, 2
jg      short loc_8048BC9
```

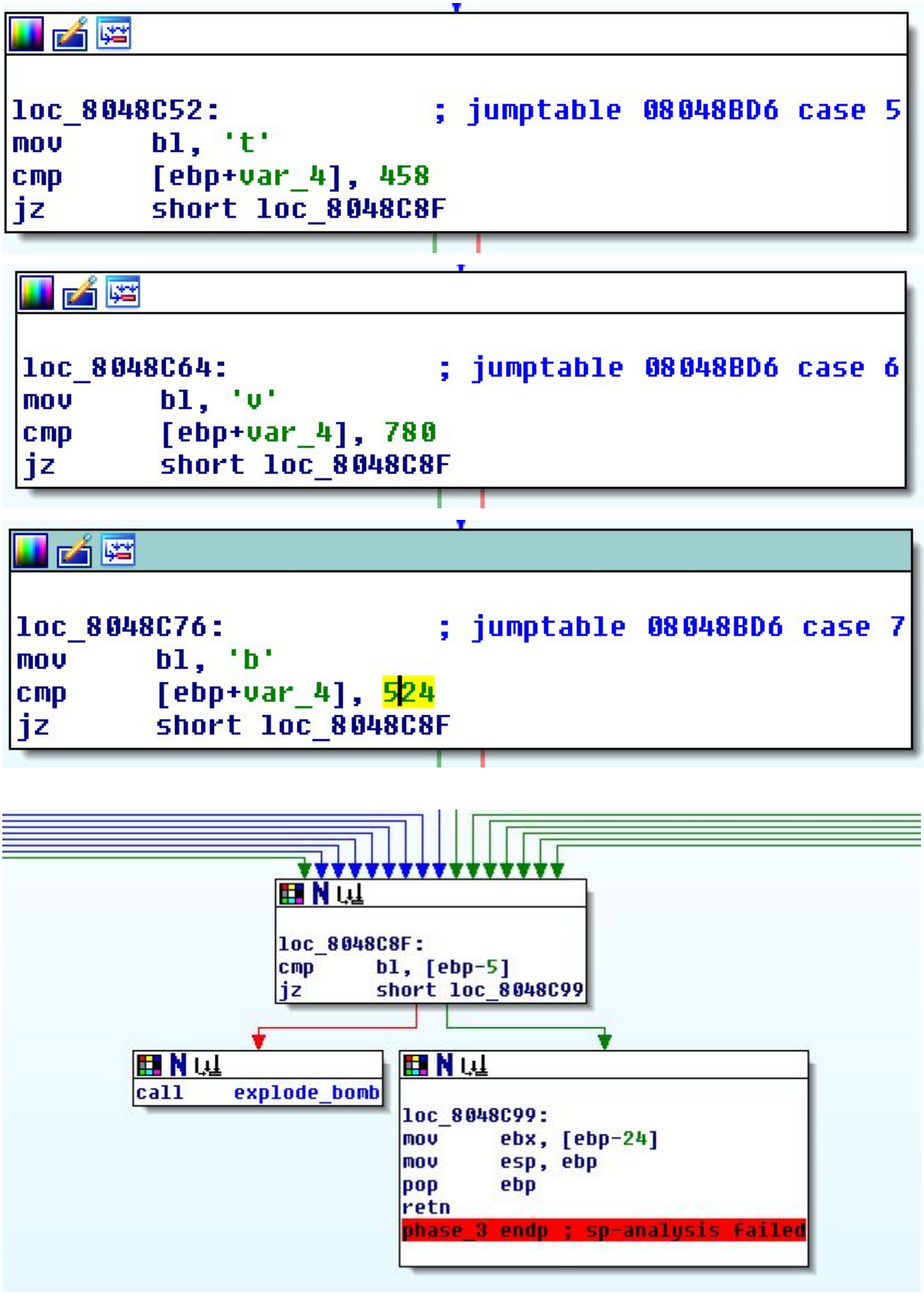
The struct of phase_3 shows that the insertion should follow the sequence of number, character, and number. (ex: 1 a 1). The first challenge is comparing the EAX with 2. EAX stores all the inserted parameters, and the number of parameters should larger than 2. In next block, ebp-1 means the address of the first parameter, and it should smaller than 2.



Then the program goes to a switch jump in 8 cases. All 7 cases, give variable bl a number and compare the last input number with 777, 214, 755, 251, 160, 458, 780, 524. The bomb explodes when the last input number does not match, in 0-7 cases, depending on the first input.

And according to given bl number, variable bl can be 113, 98, 98, 107, 111, 116, 118, 98.


loc_8048BE0: ; jumptable 08048BD6 case 0
mov bl, 'q'
cmp [ebp+var_4], 777
jz loc_8048C8F
loc_8048C00: ; jumptable 08048BD6 case 1
mov bl, 'b'
cmp [ebp+var_4], 214
jz loc_8048C8F
loc_8048C16: ; jumptable 08048BD6 case 2
mov bl, 'b'
cmp [ebp+var_4], 755
jz short loc_8048C8F
loc_8048C28: ; jumptable 08048BD6 case 3
mov bl, 'k'
cmp [ebp+var_4], 251
jz short loc_8048C8F
loc_8048C40: ; jumptable 08048BD6 case 4
mov bl, 6Fh
cmp [ebp+var_4], 0A0h
jz short loc_8048C8F

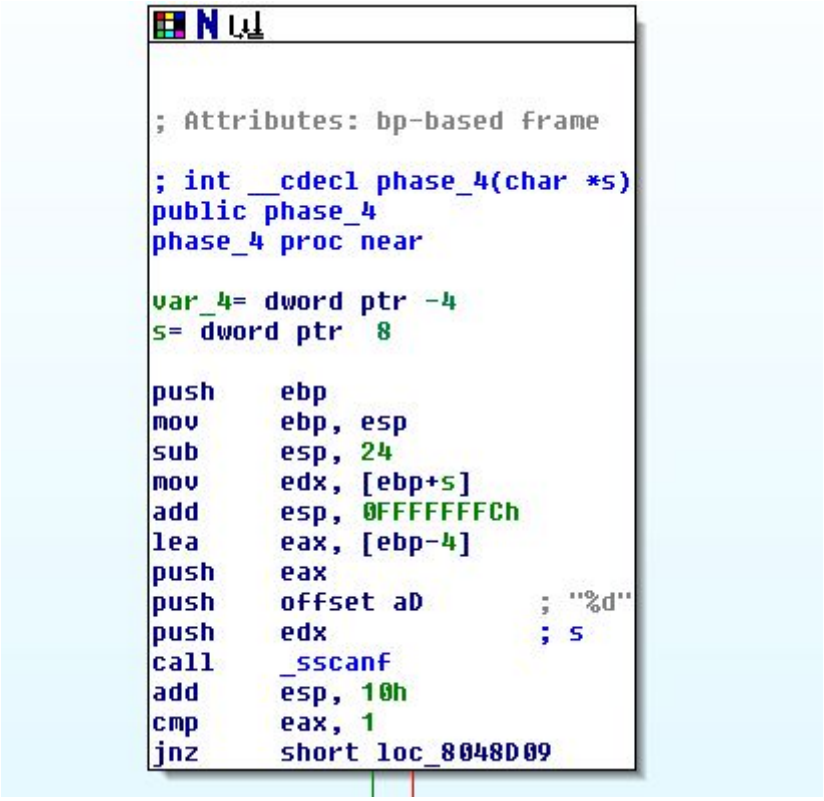


The last block compares the bl with second input which is a character. As a result, the valid input can be following sequences (numbers are in decimal).
0, 113, 777

1, 98, 214
2, 98, 755
3, 107, 251
4, 111, 160
5, 116, 458
6, 118, 780
7, 98, 524

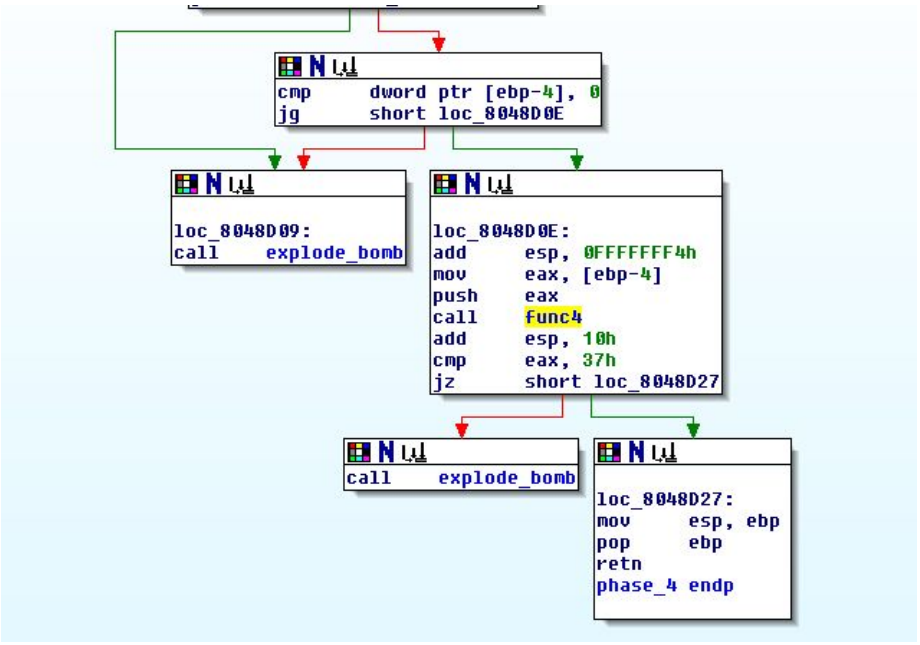
Second input should be a character. After matching with ASCII code, we get:
0 q 777
1 b 214
2 b 755
3 k 251
4 o 160
5 t 458
6 v 780
7 b 524

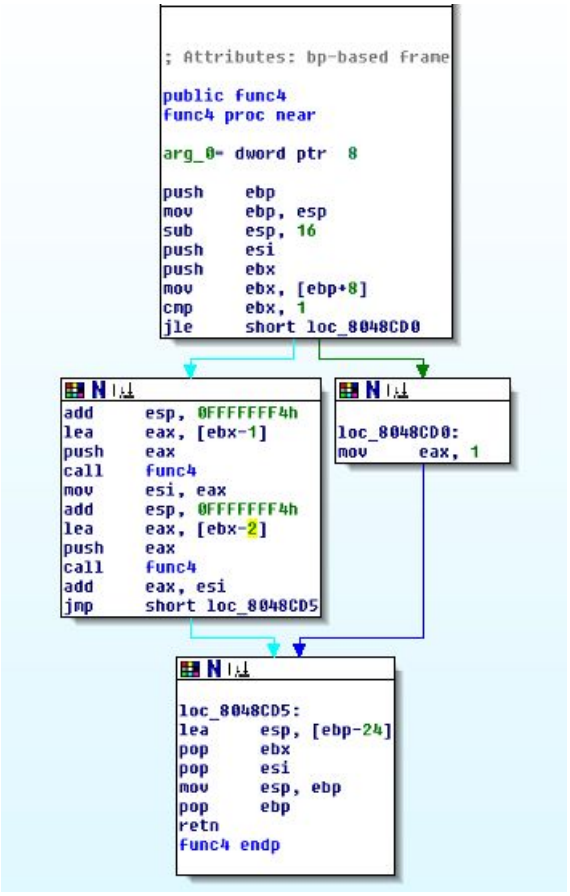
Phase 4



```
; Attributes: bp-based frame

; int __cdecl phase_4(char *s)
public phase_4
phase_4 proc near

var_4= dword ptr -4
s= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 24
mov     edx, [ebp+s]
add     esp, 0FFFFFFFCh
lea     eax, [ebp-4]
push    eax
push    offset aD           ; "%d"
push    edx                 ; s
call    _sscanf
add     esp, 10h
cmp     eax, 1
jnz     short loc_8048D09
```

Guessing that phase 4 require a numeric integer, and the input should also larger than 0. If detected a valid input, pass the input number to func4.

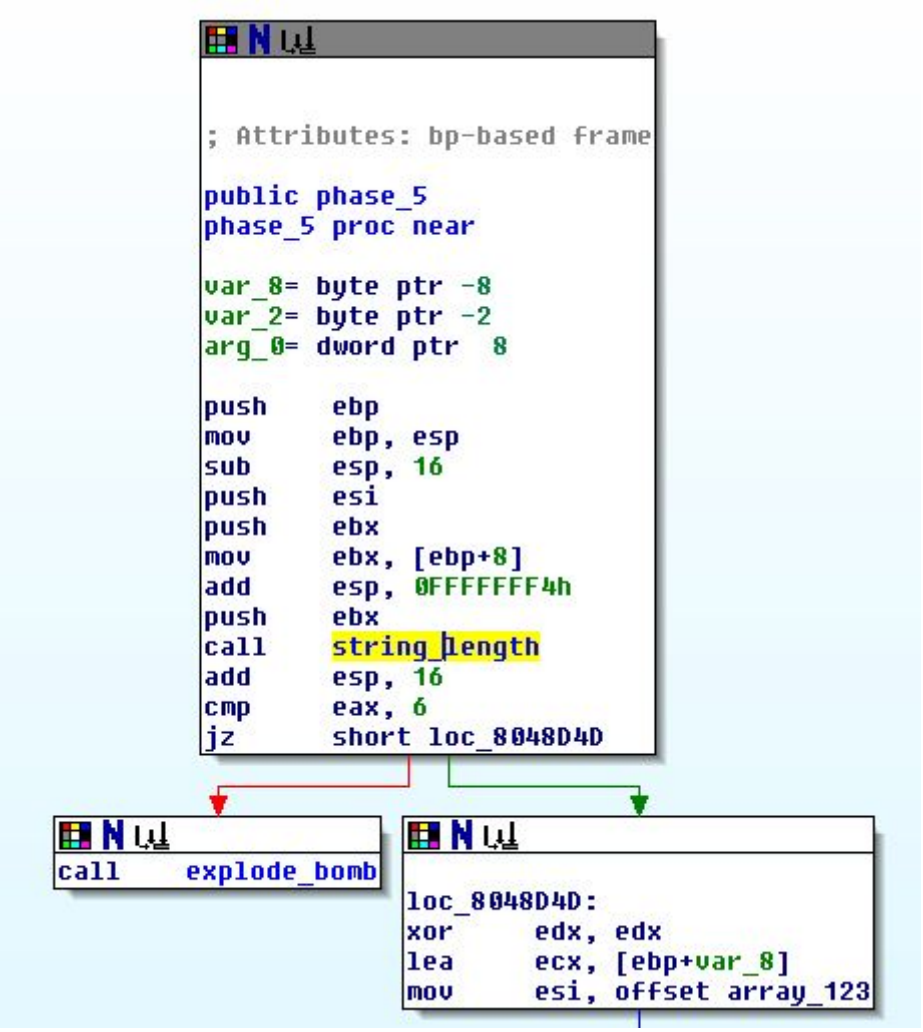




In func4, it is a recursive call. EAX is the return number of func4 with the parameter of EBX-1, and it was moved to ESI after returning. EAX then become the return of func4 with the parameter of EBX-2. Finally add EAX and ESI, finish the recursive call. It is exactly saying that func4(x) = func4(x-1)+func4(x-2), if input smaller than 2, just return.

Going back to phase_4, it compares the return value with 34h, which is 55 in decimal, and it will not call explode_bomb in this situation.We find that it is the Fibonacci numbers! In order to get the result of 55 from func4() where func4(0) and func(1) are 1, the input should be 9.

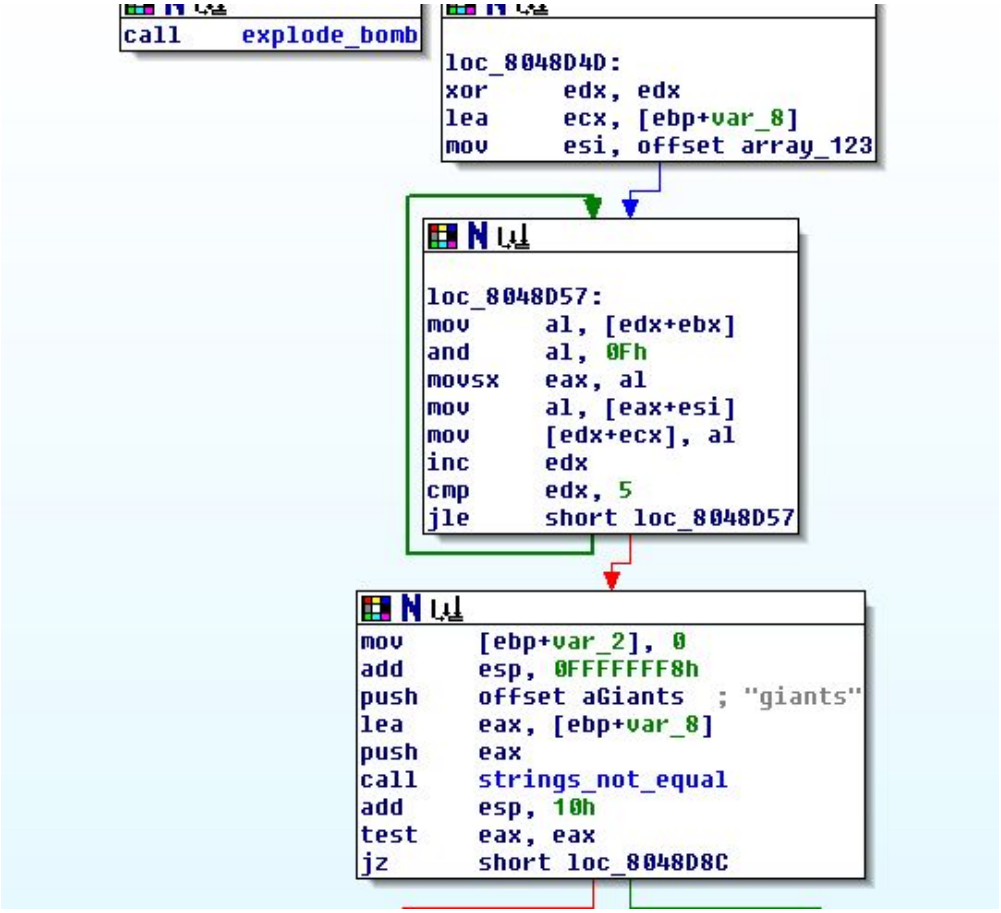
Phase 5



We find that there is no scanf here, so we trace back, find the string is actually in EAX since there is a read_line function before calling phase_5.

```
add     esp, 0FFFFFFF4h
push    offset aSoYouGotThatOn ; "So you got that one.  Try this one.\n"
call    _printf
add     esp, 32
call    read_line
add     esp, 0FFFFFFF4h
push    eax
call    phase_5
call    phase_defused
```

The string_length function will get the length of the input string and store the length in eax, then compare it with 6, so the length of input string should be 6.



EDX is set to be 0 before the loop. It increased itself each time in the loop, which works as a counter which will exit when it's greater than 5. So the loop runs 6 times. In this loop, the start address of input string is stored in EBX. An array named "array_123" is stored in ESI, which stores the mapping relationship of some ASCII. Each round of the loop will do an AND operation with the input character and "0Fh", this operation will get the last four binary digits, i.e., the last digit of hex, then store the results in EAX. Then use the result of AND as an offset to find the corresponding character in "array_123", and store the result in ECX. This process is like a decryption which will decrypt the input string and map it using the keys in "array_123".

```
data:0804AE20
data:0804B220 array_123       db  69h ; i
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
```

Then a “strings_not_equal” function is called to compare the decrypted string and the preset “giants”, and return 0 if two strings are equal, and keep the result in EAX. If EAX is 0, then the “test EAX, EAX” will be 0 and phase 5 is defused, otherwise, the result would be 1 and the bomb would explode.

Since the offset of “giants” in “array_123” is F,0,5, B, D,1, we can figure out that after the “& 0F” operation, which sets the high four digits to 0, and remain the low four digits unchanged, the ASCII input is ending as F05BD1 correspondingly. So the ASCII of input should be any combination of characters looks like “xFh x0h x5h xBh xDh x1h”. An example is “opekmq”, another example is “OpekmQ” since “O” and “o” both ends as “F”, meanwhile “Q” and “q” both ends as “1” in ASCII.

Phase 6

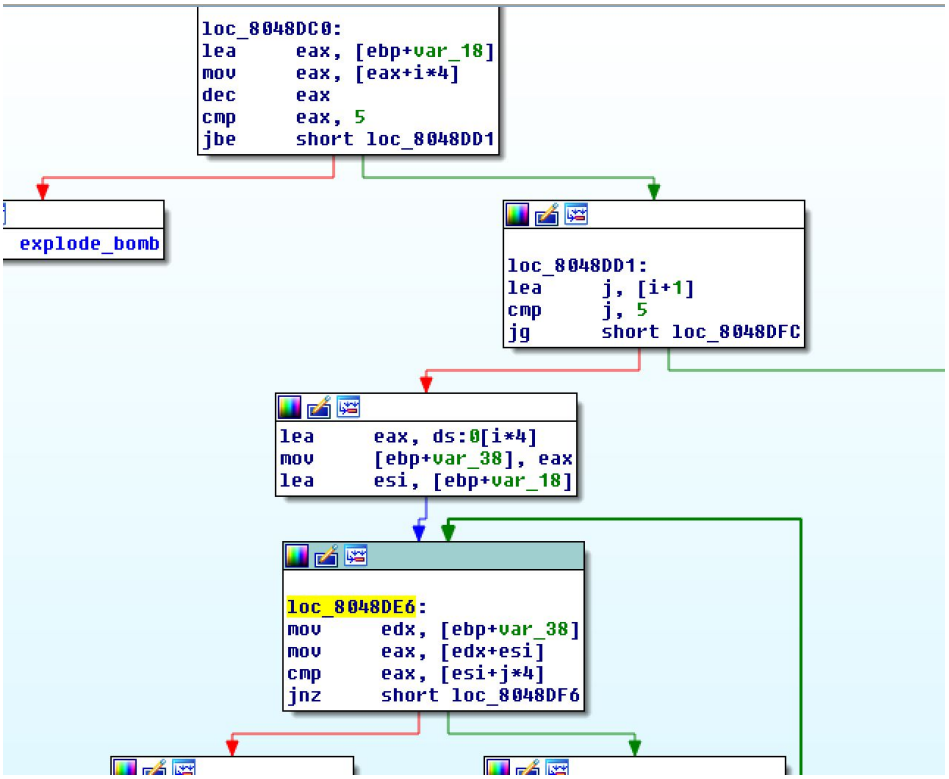
Phase 6 is calling the read_six_numbers function from phase 2. So we can know that it’s still taking 6 integers separated by space as input.

```
push    edx             ; s
call    read_six_numbers
xor     i, i
add     esp, 10h
lea     esi, [esi+0]
```

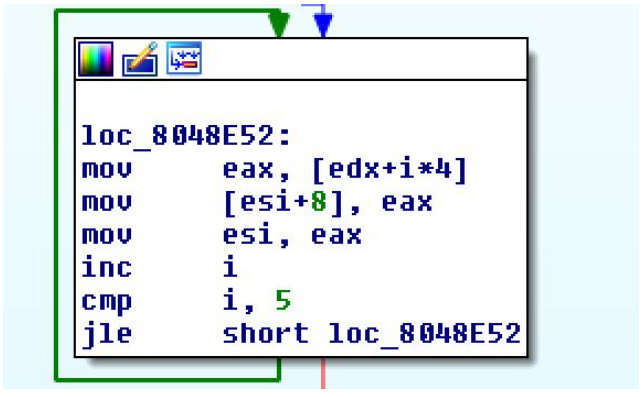
Then we noticed that there is a multi-layer loop structure using 2 loop counters which are marked as ‘i’ and ‘j’ as shown below. Counter ‘i’ is initialized with 0 and increased by 1 until 5 to traverse all 6 inputs.

```
loc_8048DFC:
inc     i
cmp     i, 5
jle     short loc_8048DC0
```

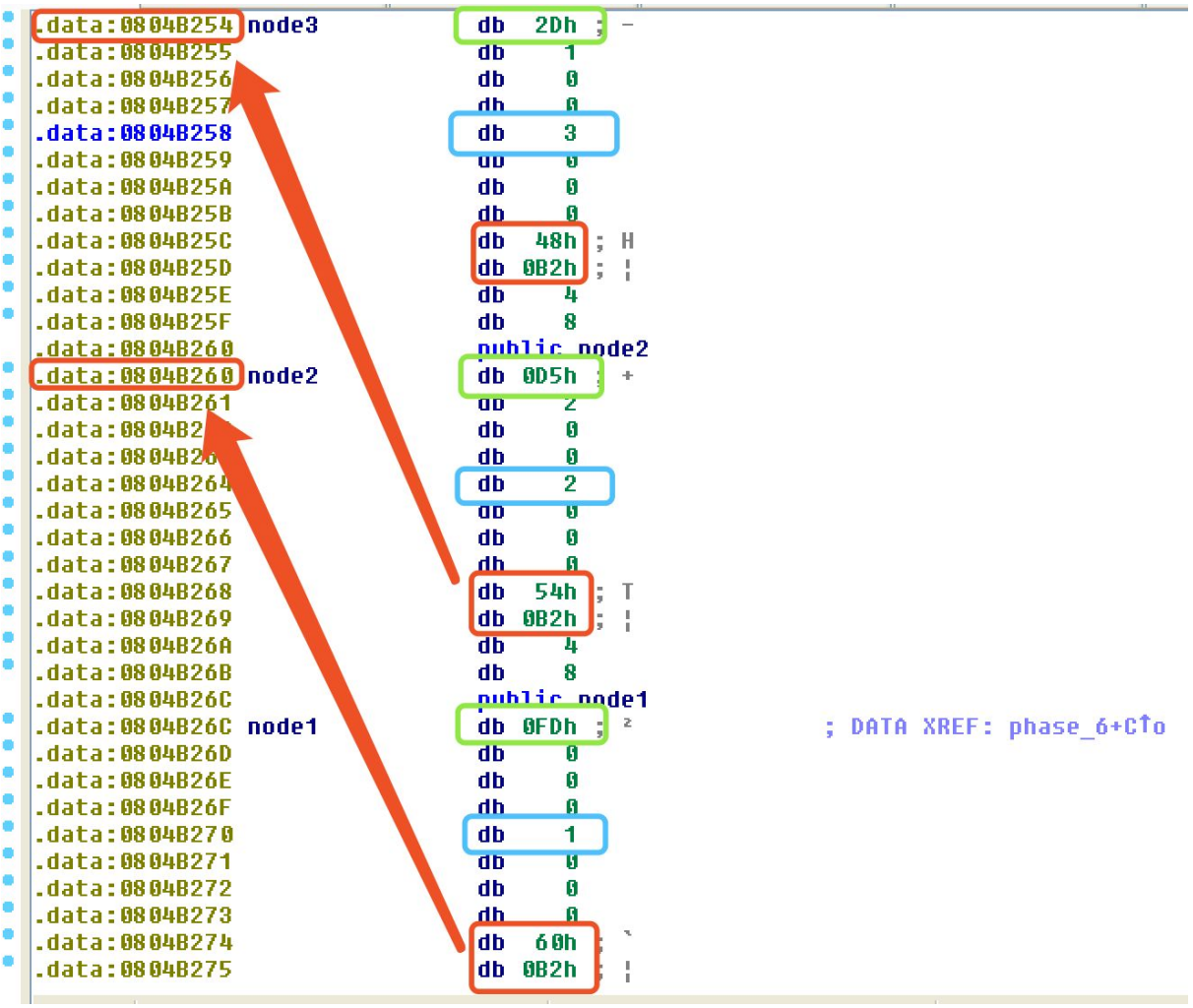
We can see from here that every input integer is decreased by 1 and then compared with 5 and if anyone of them is not less or equal than 5 the bomb will explode which means every input should be equal or greater than 6. The inner loop counter ‘j’ ranges from i+1 to 5 to compare every input integer with other integers. If any two of them is equal, the bomb will also explode.



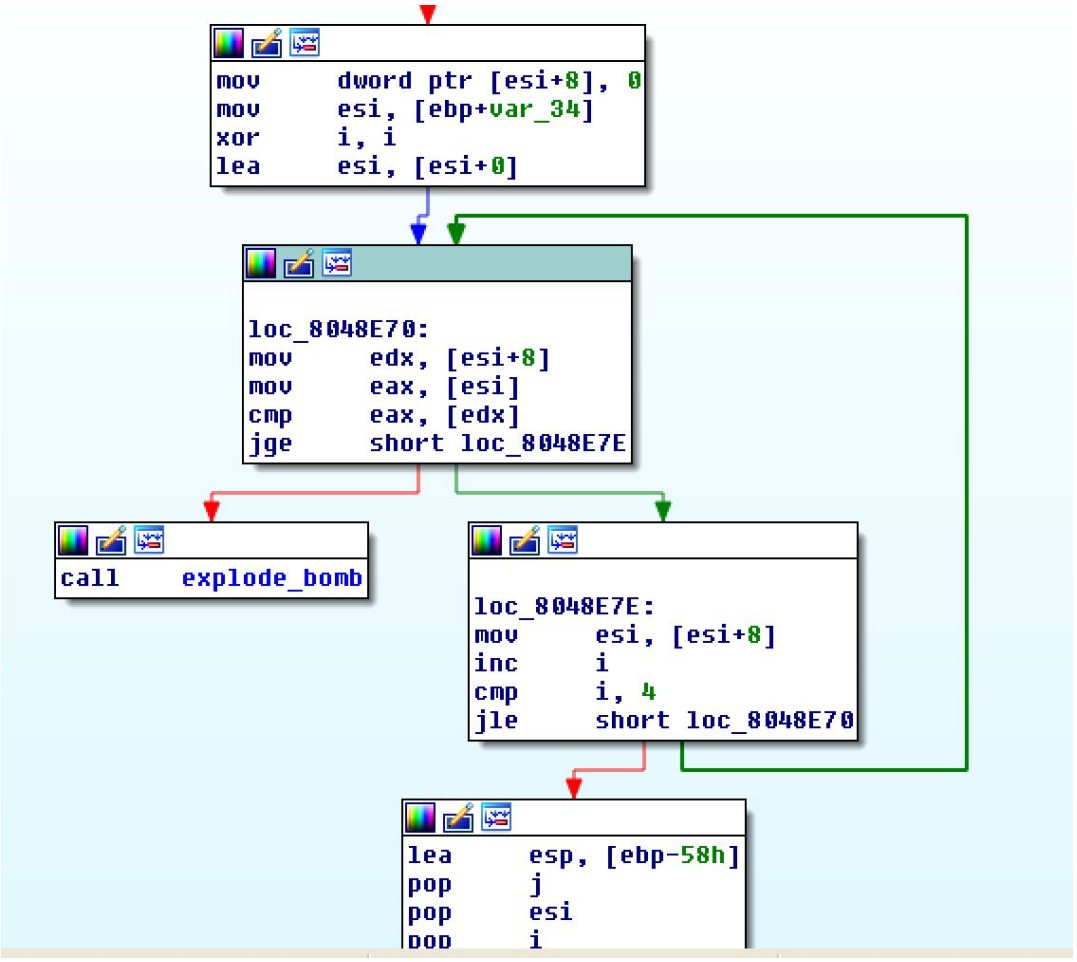
As we are looking into the detailed structure in phase 6, we noticed that there are a lot of operations on the address. For example in the following figure, mov eax, [edx+i*4] is moving the value locates in address[edx+i*4] to eax, where 4 represents 4 bytes. [esi+8] is also an address. So we go back to the beginning of the function call and look into the offset node.



We find that node is a sequence of structures that stores basically 3 pieces of information. One of them is the sequential number (the part marked blue in the screenshot). Another part is the address of the next node (marked red). The last part is a numerical value (marked green) of which the meaning is not yet clear. But a proper guess would be that this structure is actually a linked list, where every node contains the sequential number, a mysterious numerical value and a pointer to the next node.



Back to the analysis in the function phase 6. This time we trace back from the bottom. As shown in the following screenshot, before the function ends, the loop counter “i” is reset to 0 to count for another loop. This time, the value stored in address[esi] is compared with the value stored in address[esi+8]. If [esi] is less than [esi+8], the boom will explode. So we know it is actually doing a compare between something in the current node and same thing in the next node. Obviously it’s not the sequential number nor the pointer. Then it must be the value stored in each node marked green in the screenshot. So what we need to do is to put the nodes in the order that they are sorted by the green value from the largest to the smallest.



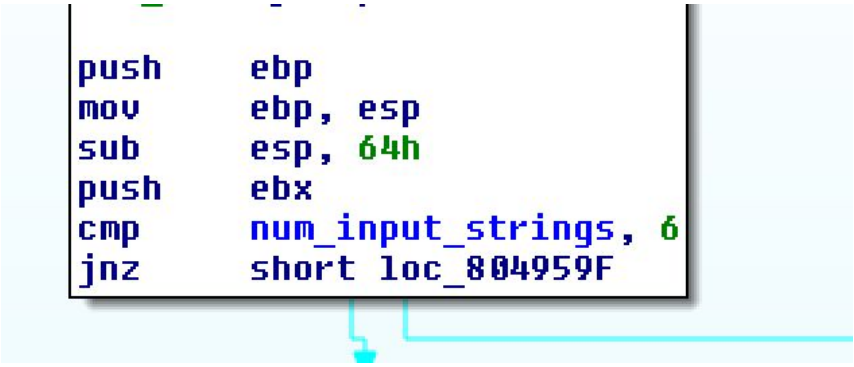
Given value in each node:

node1	0FD
node2	2D5
node3	12D
node4	3E5
node5	0D4
node6	1B0

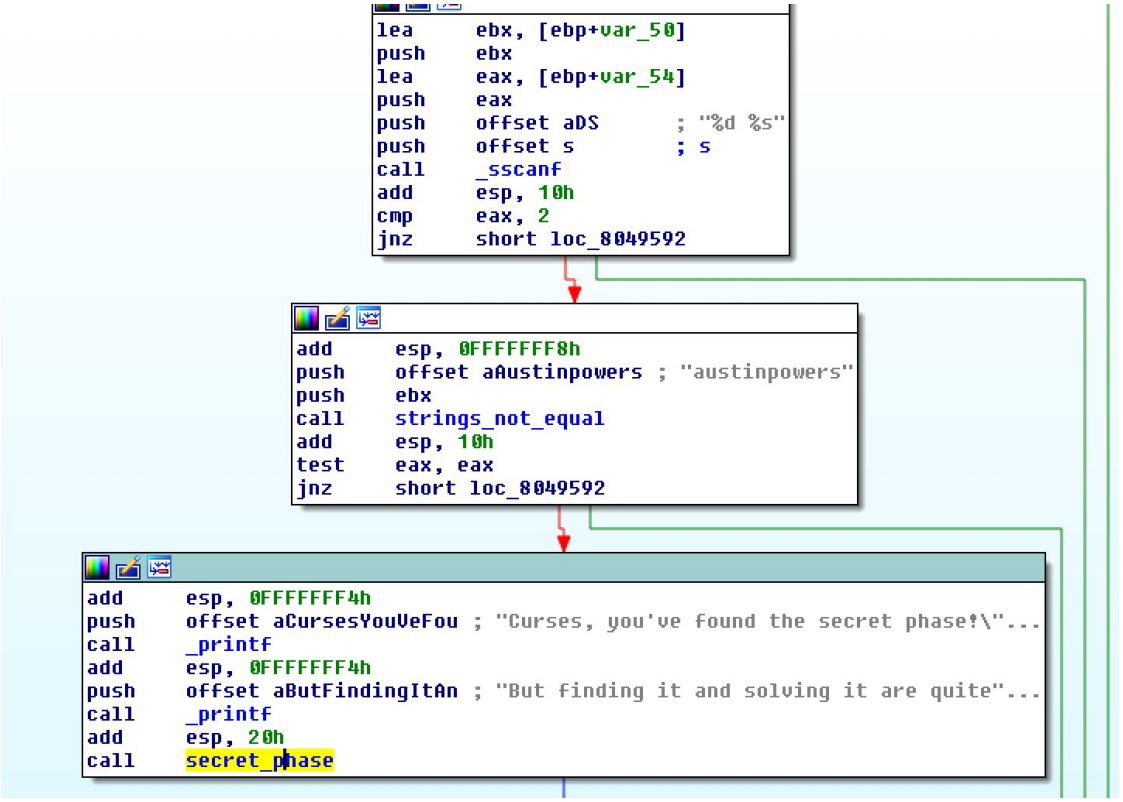
So the order should be 4 2 6 3 1 5. Then we try to input 4 2 6 3 1 5, as expected, the bomb is successfully defused.

Secret Phase

While we were working on the bomb, we found that there is a secret phase trigger in the phase_defused function as shown below. The input in each phase is passed to the phase_defused function and compared with 6. If the input number is 6, the function ends.



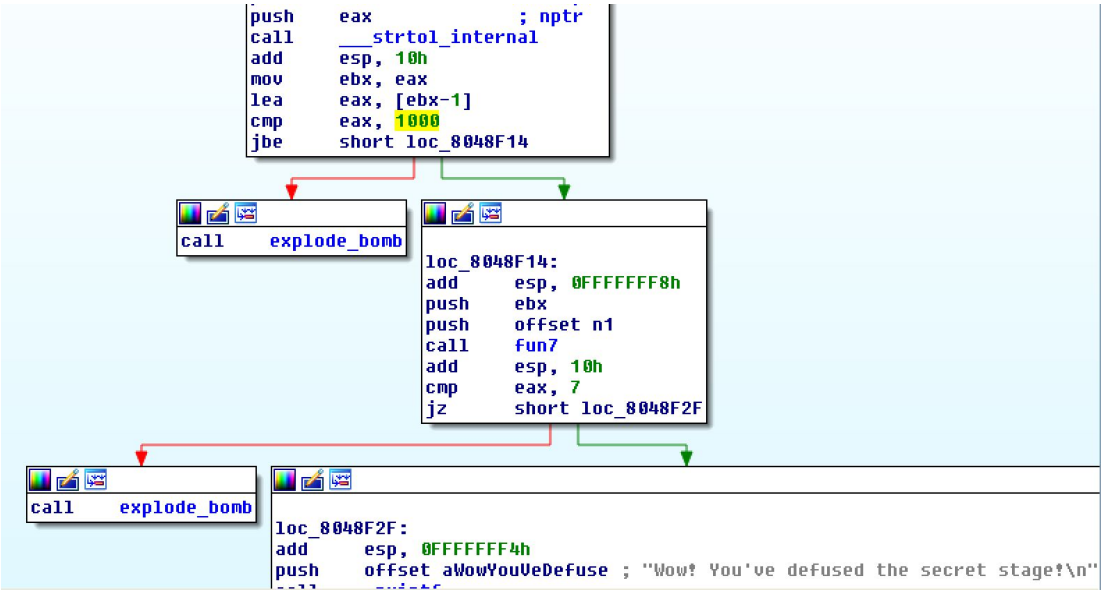
Otherwise, push an input decimal and a string onto the stack, and compare the input string with “austinpowers”. If they are equal, call function secret_phase.



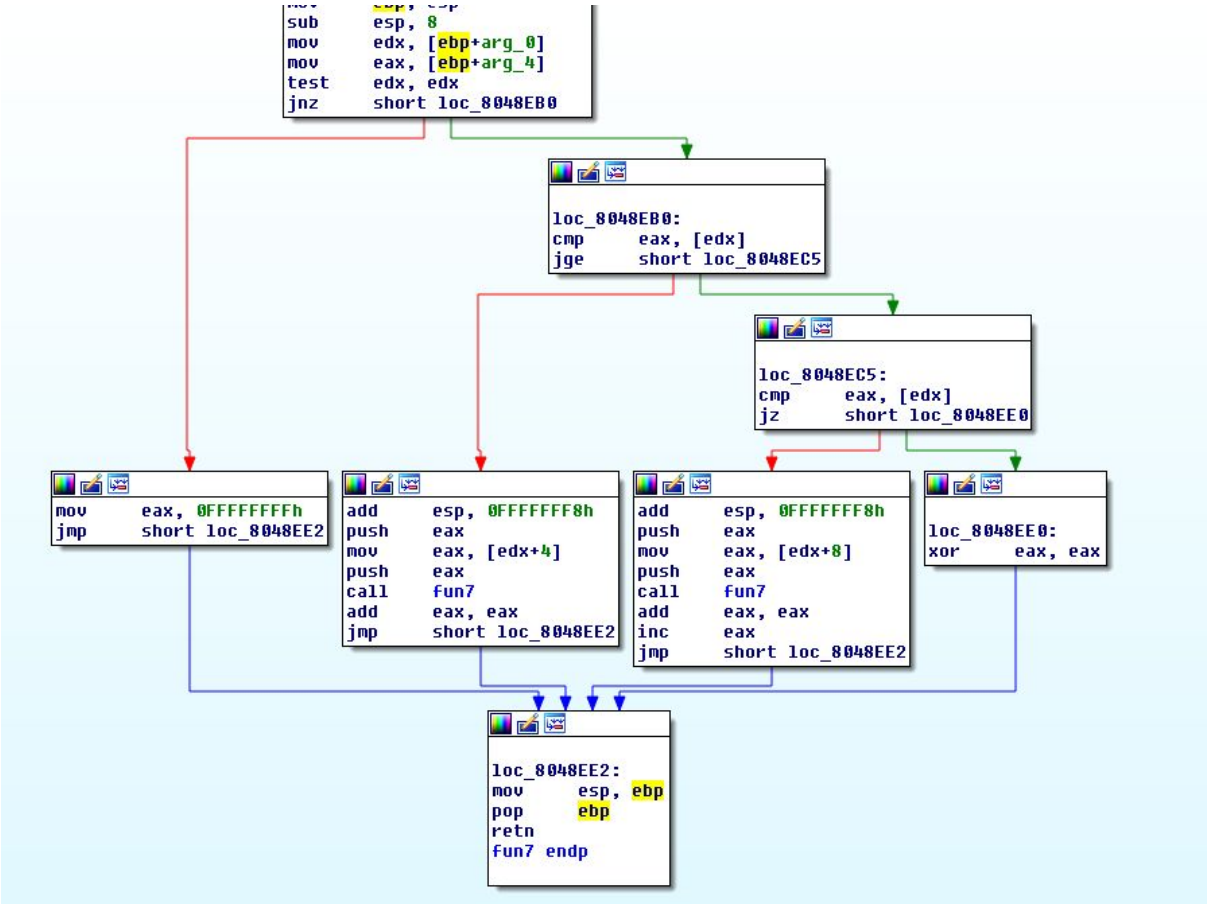
The only phase that takes only a decimal number as input is phase 4. So we tried to type in “austinpowers” also as input after the correct number “9” for phase 4. This time after we finished all 6 phases we entered the secret phase.

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
0 q 777
Halfway there!
9 austinpowers
So you got that one.  Try this one.
opekmq
Good work!  On to the next...
4 2 6 3 1 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

Continue to look into function secret_phase. First, we can learn that the input should be greater than 1000.



In the secret phase, the program will run into fun7() and compare eax with 7, if eax is 7 then this phase is defused, so we go into fun7() to find out how to return 7.



This part will compare the input number in eax with the [edx], since the input is required greater than 1000, and [edx] is offset n1, which is 24h, fun7() will enter a recursive call, and change the edx into [edx + 8], meanwhile the returned eax will be doubled and plus 1. We can calculate that the result 7 comes from ((0 * 2 + 1) * 2 + 1) * 2 + 1. The root eax value should be 0, which could only appear when eax quals [edx]. So overall this program is tracing the changing edx and stop if the result of [edx] is equal to the input number, and the input number is 7, and there are exactly four recursions as we figured out above. We can then trace the offset n1 and find the result. The offset of n1 is 320, thus edx = 320 and [edx] = 24h, but 24h is smaller than 1000(3E8h). Since [edx + 8] = [328] = 8, so the offset of the first sub fun7() is 308.

```
.data:0804B320                  public n1
.data:0804B320 n1               db  24h ; $
.data:0804B321                  db    0
.data:0804B322                  db    0
.data:0804B323                  db    0
.data:0804B324                  db  14h
.data:0804B325                  db 0B3h ; ¦
.data:0804B326                  db    4
.data:0804B327                  db    8
.data:0804B328                  db    8
.data:0804B329                  db 0B3h ; ¦
```

At 0804B308 we find here edx = 308 and [edx] = 32h, still smaller than 3E8h, so we will continue find [edx + 8] = [310] = 2D8h. So the second sub fun7() has an input edx as 2D8h.

```
.data:0804B308 n22          db  32h ; 2
.data:0804B309              db    0
.data:0804B30A              db    0
.data:0804B30B              db    0
.data:0804B30C              db 0F0h ; =
.data:0804B30D              db 0B2h ; ¦
.data:0804B30E              db    4
.data:0804B30F              db    8
.data:0804B310              db 0D8h ; +
.data:0804B311              db 0B2h ; ¦
```

Tracing the offset 0804B2D8, similarly, the result 6Bh is smaller than 3E8h, then we enter the last recursion, here we find [edx] = [2D8 + 8] = [2E0] = 278h. So we go to 0804B278.

```
.data:0804B2D8 n34          db  6Bh ; k
.data:0804B2D9              db    0
.data:0804B2DA              db    0
.data:0804B2DB              db    0
.data:0804B2DC              db 0B4h ; ¦
.data:0804B2DD              db 0B2h ; ¦
.data:0804B2DE              db    4
.data:0804B2DF              db    8
.data:0804B2E0              db  78h ; x
.data:0804B2E1              db 0B2h ; ¦
```

Here we found [0804B278] = 3E9h, which is greater than 3E8h.

```
.data:0804B277              db    8
.data:0804B278              public n48
.data:0804B278 n48          db 0E9h ; T
.data:0804B279              db    3
```

We should end here with the input number to be 3E9h, which is 1001. The input number is 1001.

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
0 q 777
Halfway there!
9 austinpowers
So you got that one.  Try this one.
opekmq
Good work!  On to the next...
4 2 6 3 1 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

By far we have defused all 6 bombs plus a secret one.