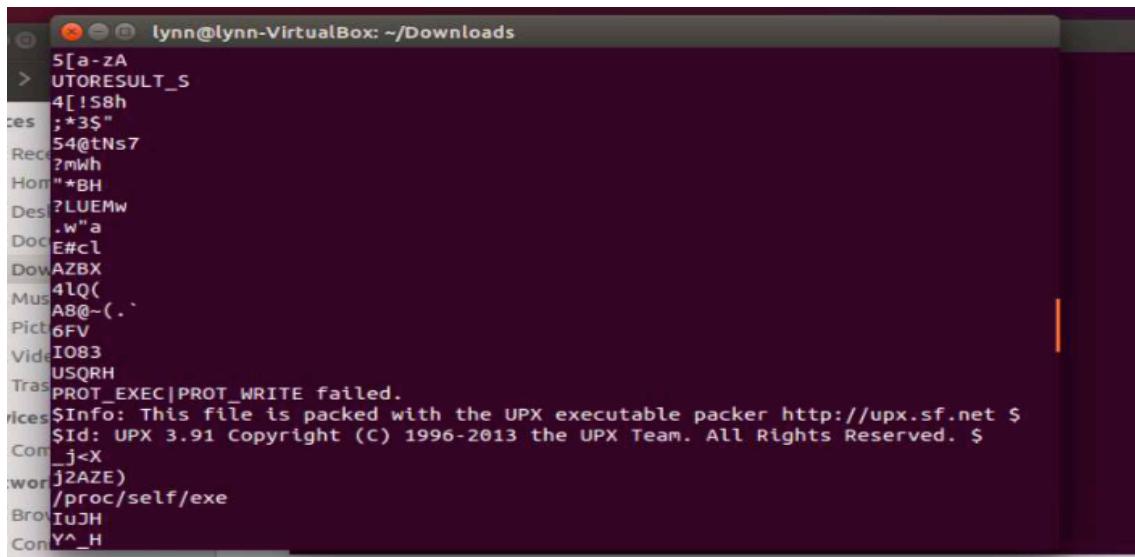


Malware Analysis & Reverse Engineering Midterm Report

Malware Bomb

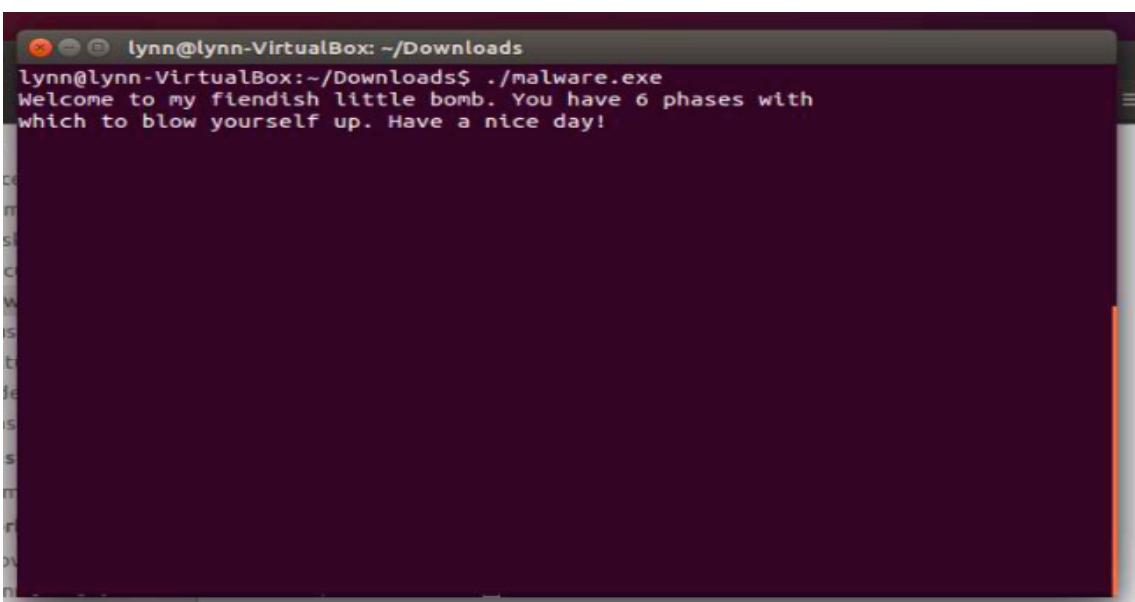
Xinyu Lyu(xl422)

At first, I found that the malware Bomb is packed with UPX when I tried to find the Strings in Ubuntu. Therefore, I unpack the Bomb with UPX unpacker. Next, I load the bomb in IDA Pro, and it took 6 steps for me to diffuse the Bomb.



The terminal window shows the output of the strings command on the malware executable. It lists various file types and names, followed by a warning about UPX packing and its copyright information.

```
lynn@lynn-VirtualBox: ~/Downloads
5[a-zA
> UTORESULT_S
4[!S8h
ces ;*3$"
54@tNs7
Rece ?mWh
Hom *BH
Des ?LUEMw
.w" a
Doc E#cl
Dow AZBX
Mus 4LQ(
Pic 6FV
Vid IO83
USQRH
Tras PROT_EXEC|PROT_WRITE failed.
rives $Info: This file is packed with the UPX executable packer http://upx.sf.net $
$Id: UPX 3.91 Copyright (c) 1996-2013 the UPX Team. All Rights Reserved. $
Com _j<X
wor j2AZE)
Bro /proc/self/exe
Con Y^_H
```

The terminal window shows the execution of the malware executable. It displays a welcome message indicating there are 6 phases and a prompt for the user to blow themselves up.

```
lynn@lynn-VirtualBox: ~/Downloads
lynn@lynn-VirtualBox:~/Downloads$ ./malware.exe
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

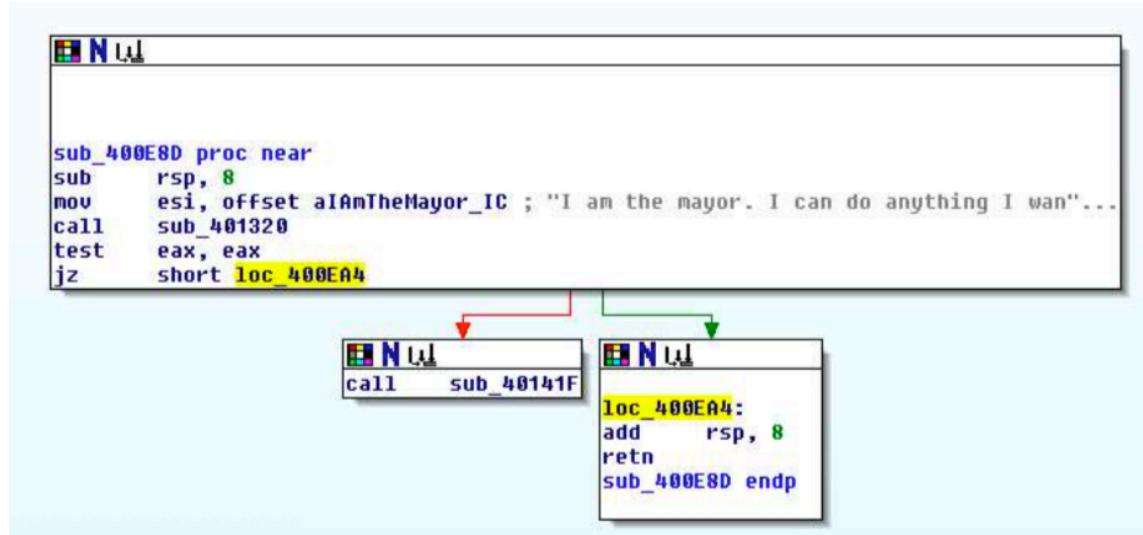
After looking through the Strings in Bomb with IDA Pro, I am pretty sure that the sub_401480 is for getting input and checking if it is generally valid. In other words, the sub_401480 is the entrance to the secret world inside malware Bomb, because it appears a lot of times.

```
call    sub_401387
mov    edi, offset aWelcomeToMyFie ; "Welcome to my fiendish little bomb. You"...
call    _puts
mov    edi, offset aWhichToBlowYou ; "which to blow yourself up. Have a nice ...."
call    _puts
call    sub_401480
mov    rdi, rax
call    sub_400E8D
call    sub_4015A6
mov    edi, offset aPhase1Defused_ ; "Phase 1 defused. How about the next one"...
call    _puts
call    sub_401480
mov    rdi, rax
call    sub_400EA9
call    sub_4015A6
mov    edi, offset aThatSNumber2_K ; "That's number 2. Keep going!"
call    _puts
call    sub_401480
mov    rdi, rax
call    sub_400F11
call    sub_4015A6
mov    edi, offset aHalfwayThere ; "Halfway there!"
call    _puts
call    sub_401480
mov    rdi, rax
call    sub_40101C
call    sub_4015A6
mov    edi, offset aSoYouGotThatOn ; "So you got that one. Try this one."
call    _puts
call    sub_401480
mov    rdi, rax
call    sub_401089
call    sub_4015A6
mov    edi, offset aGoodWorkOnToTh ; "Good work! On to the next..."
call    _puts
call    sub_401480
mov    rdi, rax
call    sub_4010CA
call    sub_4015A6
mov    eax, 0
```

Phase 1:

When checking the functions before the string ‘Phase 1 defused. How about the next one...’, I find that there is a test between Strings.

Therefore, I am sure that the answer to Phase 1 is ‘I am the mayor. I can do anything I want’.



Phase 2:

When checking the sub_400EA9, the string unk_4025A3 indicates that the input should be 4 integers. And among the parent calls in the left, the exa,5 indicates that the input should be larger than 5.

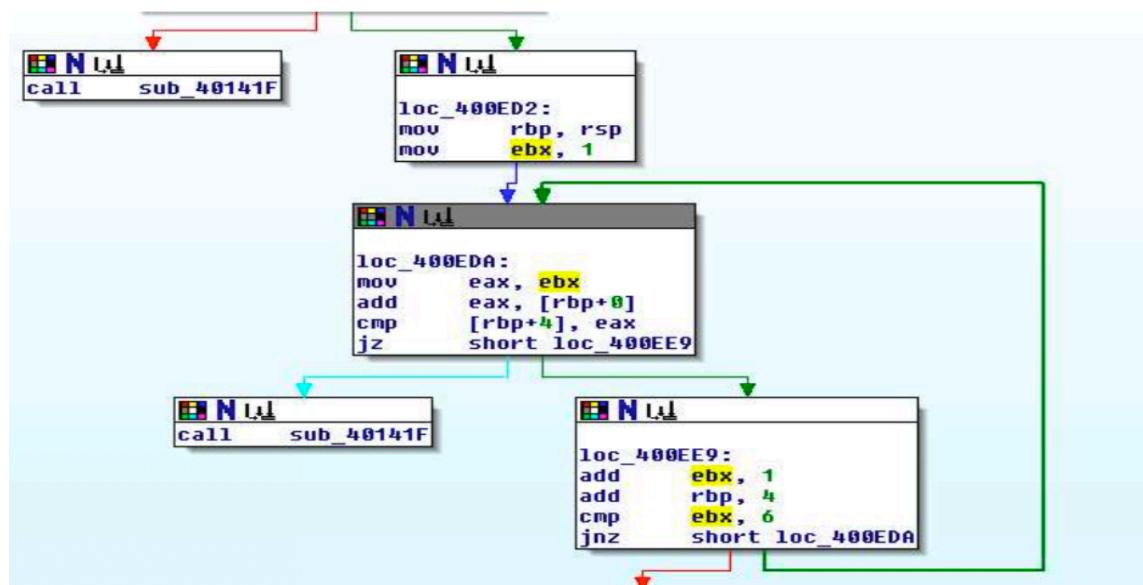
The screenshot shows two windows from the debugger. The left window displays the assembly code for `sub_401441`:

```
sub_401441 proc near
sub    rsp, 8
mov    rdx, rsi
lea    rcx, [rsi+4]
lea    rax, [rsi+14h]
push   rax
lea    rax, [rsi+10h]
push   rax
lea    r9, [rsi+8Ch]
lea    r8, [rsi+8]
mov    esi, offset unk_4025A3
mov    eax, 0
call   __isoc99_sscanf
add    rsp, 10h
cmp    eax, 5
jg    short loc_40147B
```

The right window shows the memory dump for the string `unk_4025A3`:

Address	Value	Content
43	25h	;
44	64h	;
45	20h	
46	25h	;
47	64h	;
48	20h	
49	25h	;
4A	64h	;
4B	20h	
4C	25h	;
4D	64h	;
4E	20h	

With the loop shown below, the rbp seems to be the input array. And the exa just comes from the ebx, which recursively self-increases by 1 in each loop. Therefore, the loop shows that the next number should be the last number plus the index of the last number. According to the above analysis, there are two reasonable answers for phase 2, which should be [1,2,4,7,11,16] or [2,3,5,8,12,17] depending on the first number.



Phase 3:

The screenshot shows the assembly code for a function named `sub_400F11`. The code is as follows:

```
sub_400F11 proc near

var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= qword ptr -10h

; FUNCTION CHUNK AT 0000000000400FB6 SIZE 0000002B BYTES

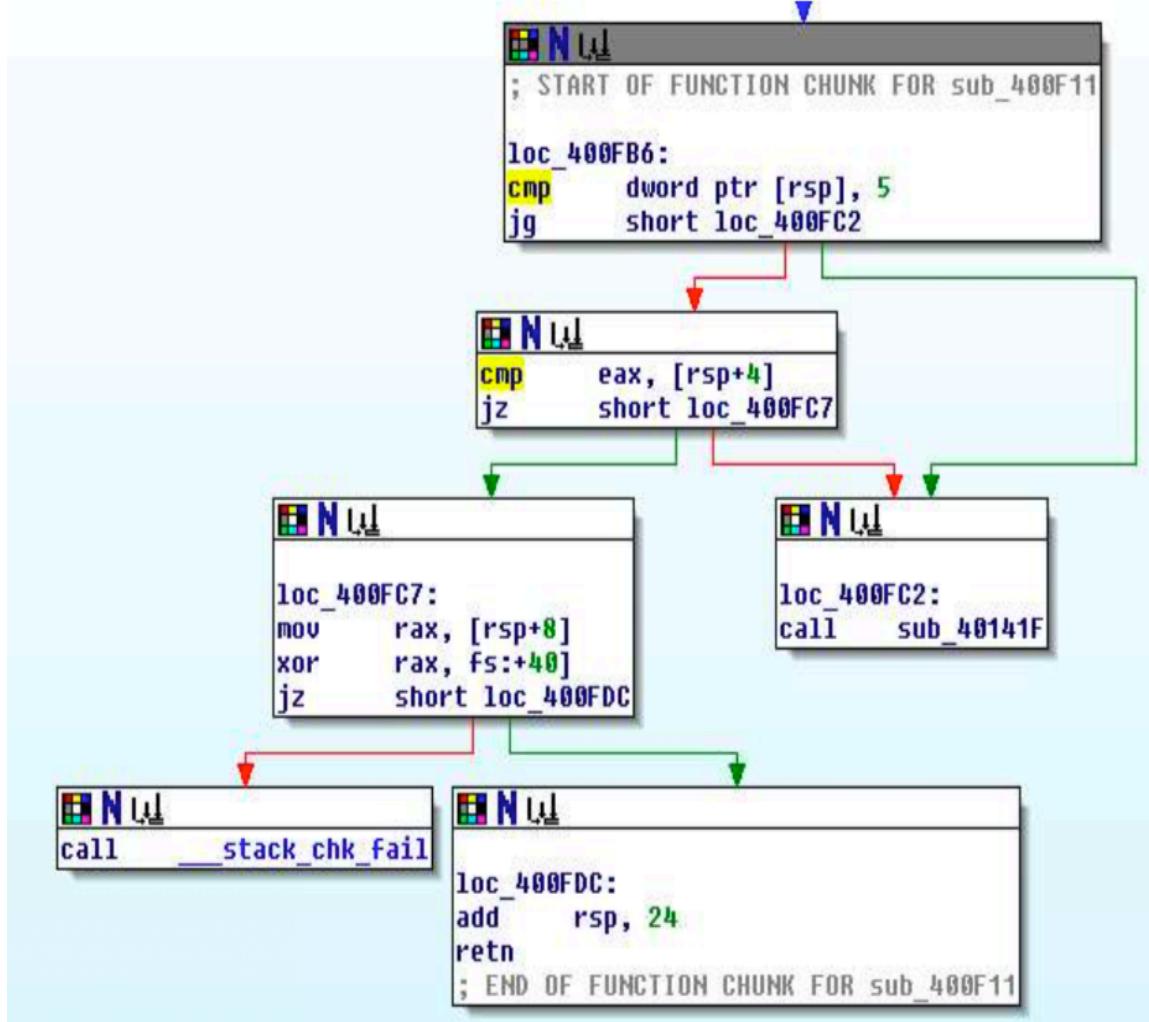
sub    rsp, 18h
mov    rax, fs:28h
mov    [rsp+18h+var_10], rax
xor    eax, eax
lea    rcx, [rsp+18h+var_14]
mov    rdx, rsp
mov    esi, offset aDD ; "%d %d"
call   __isoc99_sscanf
cmp    eax, 1
jg    short loc_400F41
```

Below the main function, two call sites are shown:

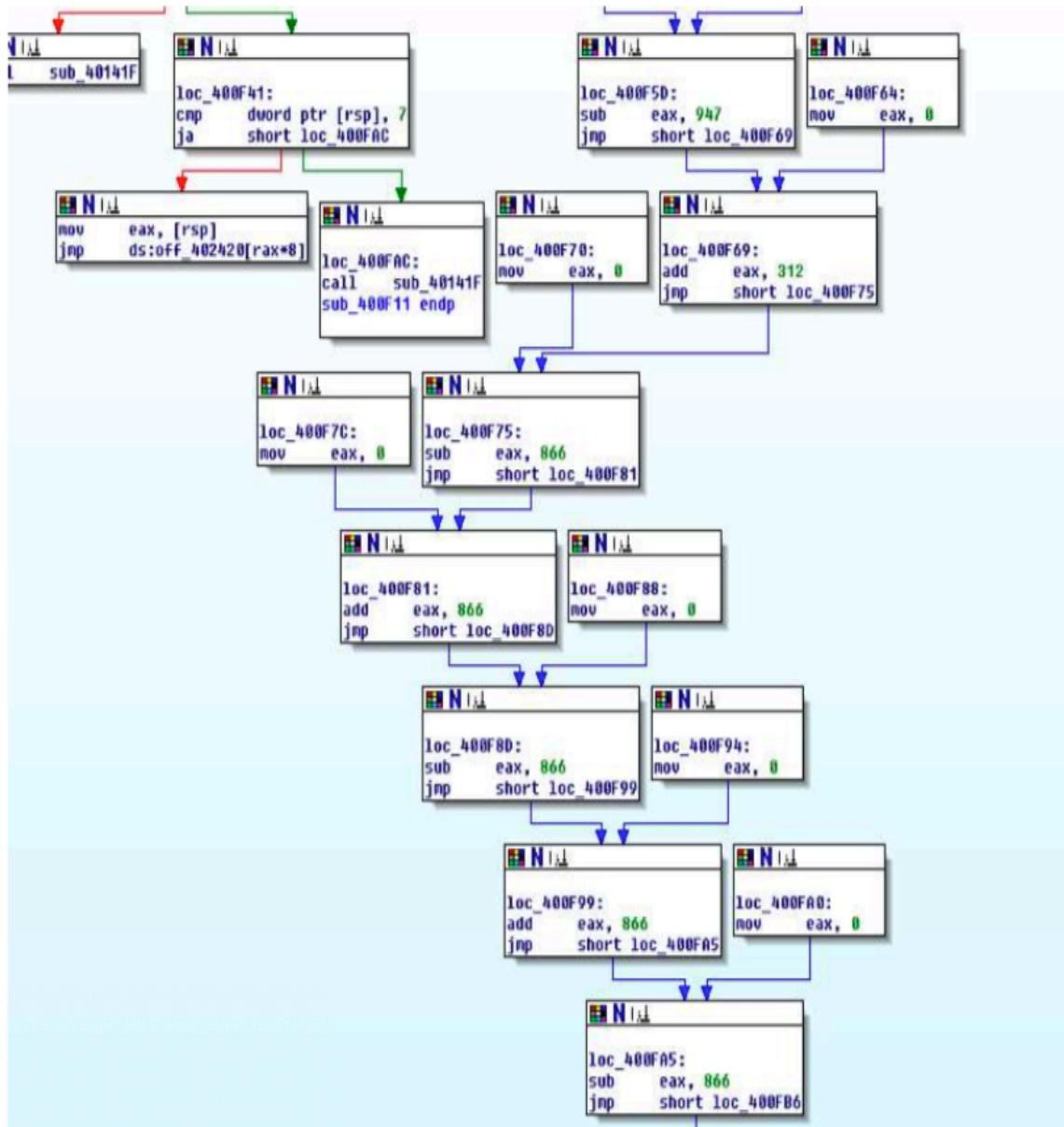
- A call to `sub_40141F`.
- A call to `loc_400F41` which contains the instruction `cmp [rsp+18h+var_18], 7`.

Two cyan arrows point from the `jg` label in the main function to the `call sub_40141F` and `loc_400F41` instructions.

The input should be at least 2 integers. And based on the compare operation `[cmp, eax, 1]`, I finds that the input should be larger than 1. Moreover, according to the compare operation `[cmp, [rsp+18h+var_18], 7]`, the first input should be larger than 7, which will call the explode function `sub_40141F`. Then, there are about 7 situations with the first input, in which they will jumps to 7 different functions.



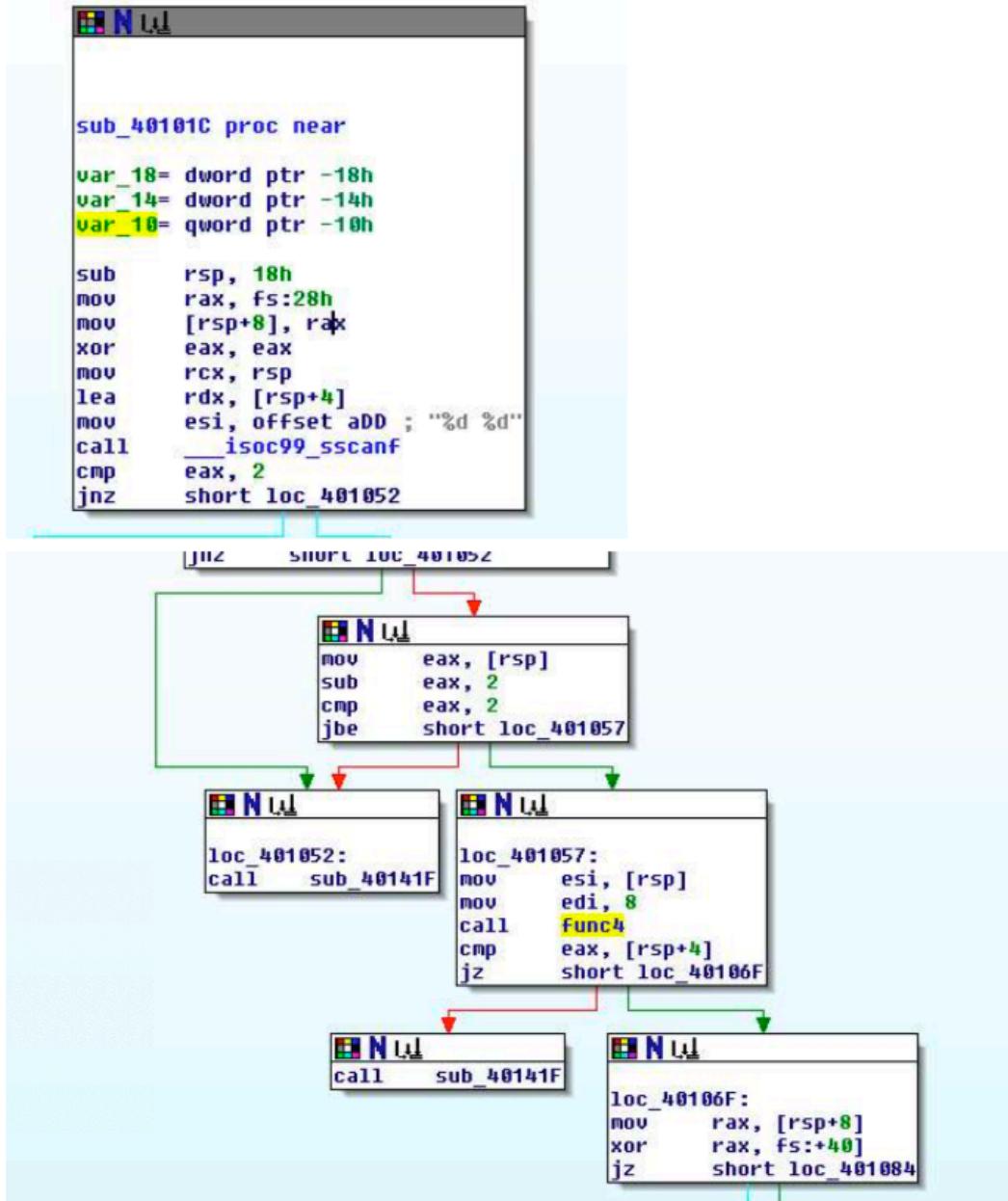
In the next comparison function [cmp, dword ptr [rsp], 5], the first input should be less or smaller than 5. And the next input is [rsp+4]. When exa= input2, it is the safe situation. Then the exa comes from different choice of the first input. Below show the logic bits of operations.



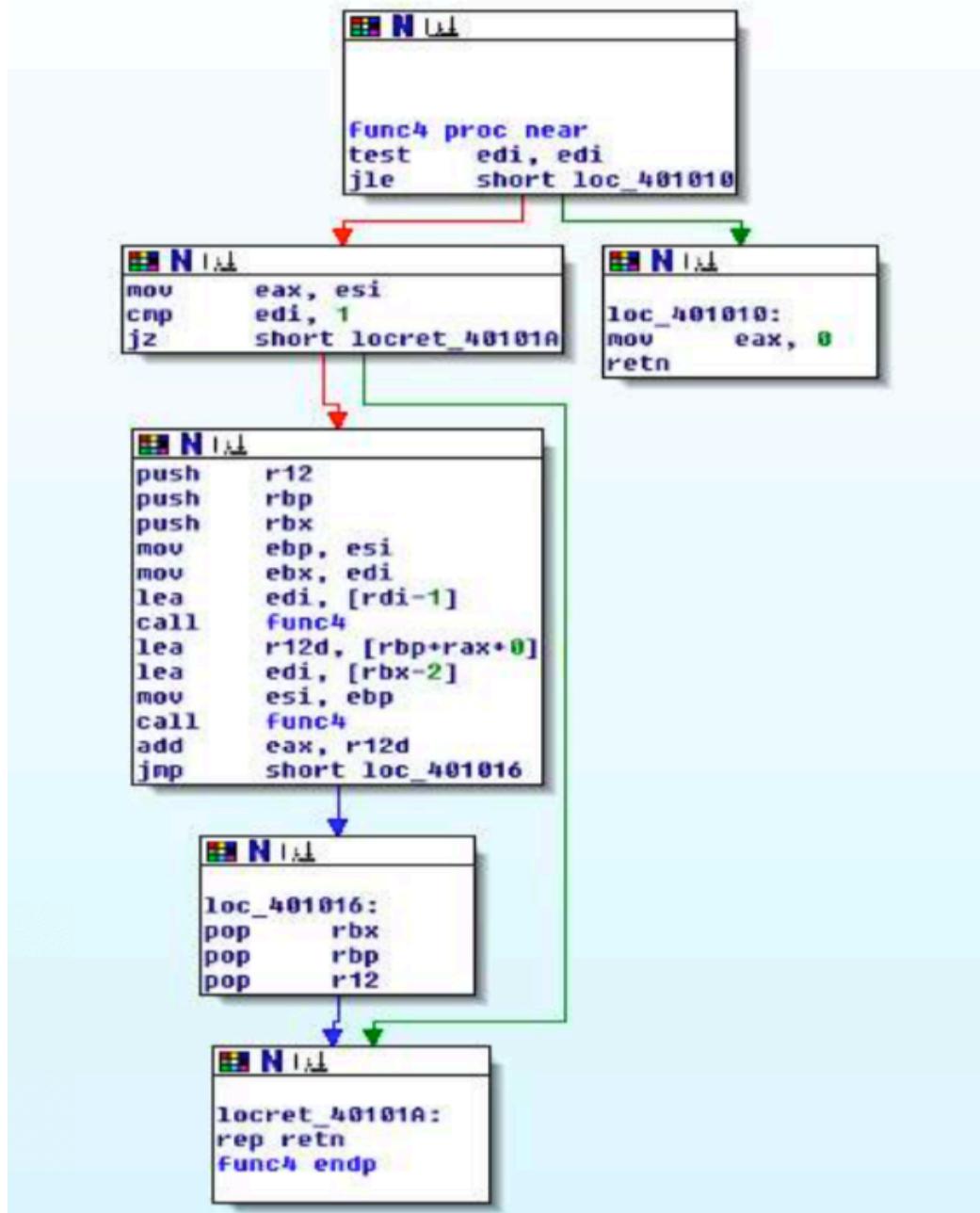
Therefore, totally, there are 6 combinations of inputs to pass the Phase 3 shown below.

1. 0-556
2. 1-1501
3. 2-554
4. 3-866
5. 0
6. 5-866

Phase 4:



With two integers as input, I follow the function calls next. Then, with the operation [sub, exa, 2], the second input gets minus by 2. Then, based on the operation next [cmp, exa, 2], it indicates that with the second minus by 2, it still should be smaller than 2. In other words, the valid input itself should be smaller than 4.

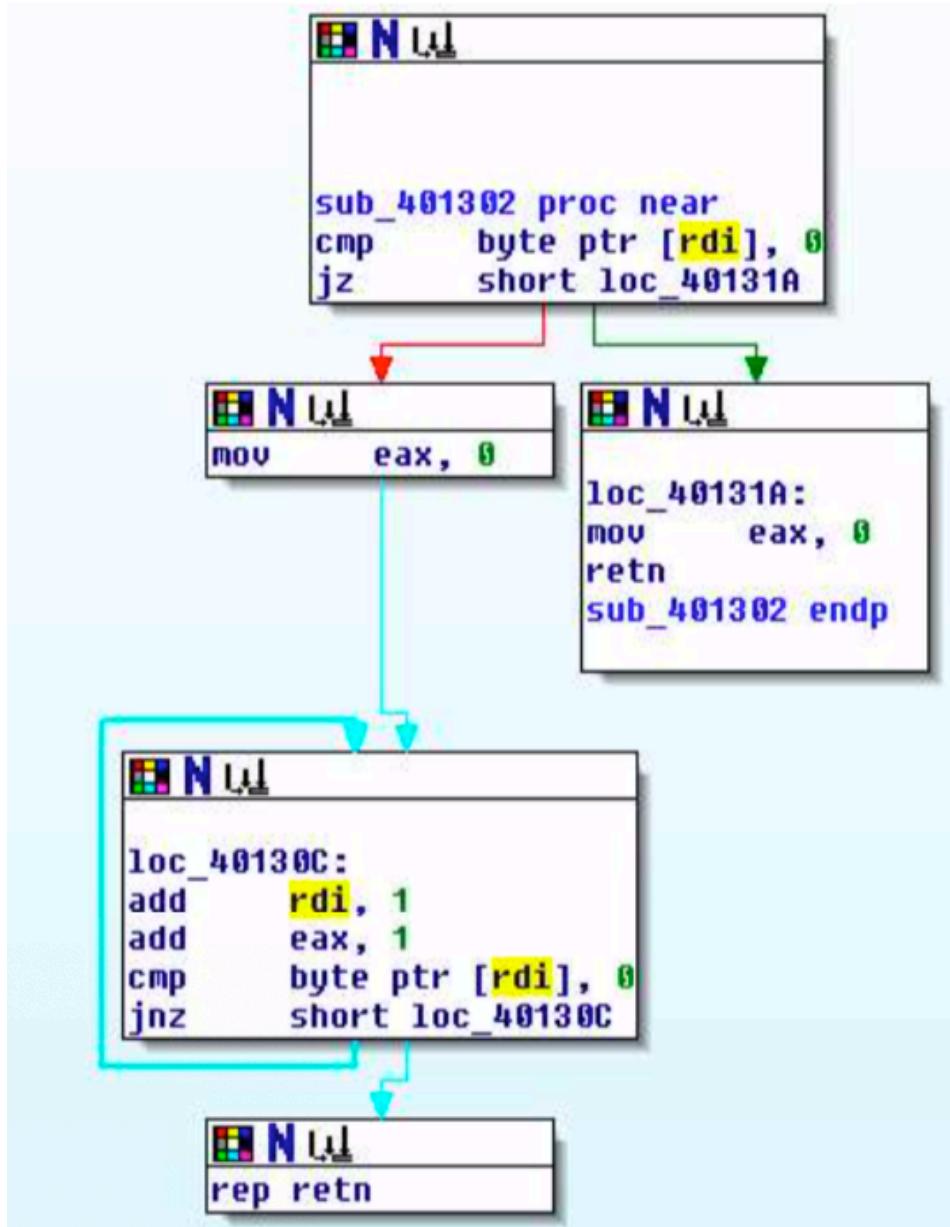


Basically, there is a recursive in function 4. And second input `edi` is initialized as 8. In the recursive, the second input gets added 54 times. If the input is 1 or 0, the function 4 just returns the same input as output. Otherwise, it will return the input number add next recursive with the input minus one and next recursive with input minus 2.

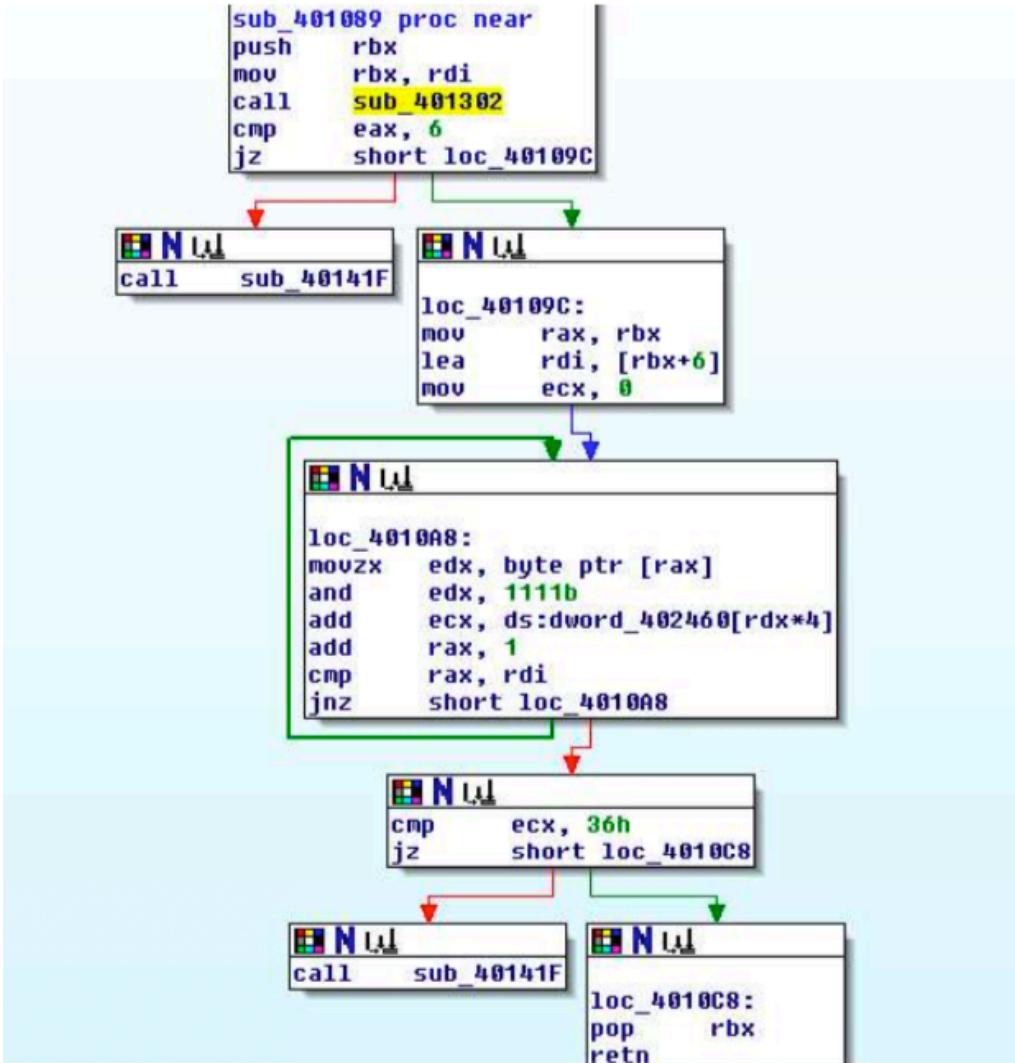
After the recursion session, matching with the outputs, the valid inputs show below

1. 216 4
2. 162 3
3. 108 2

Phase 5:



For Phase 5, I get the function call of `sub_401302` compared to the exa with 6. In `sub_401302`, there is a loop going through the input array `rdi`. Therefore, I am sure that the function call is used to check the length of the input array.



In loc_4010A8, I go through the loop. Based on the [add, ecx, ds:dword_402460[rdx*4]] operation, the input string is a sequence of index as ds:dword_402460[rdx*4] with the ras as the first character and rdi as the last one. Therefore, I got the answer to the Phase 5 is [123456].

Phase 6:

The function called sub_401441 with 6 numbers' input.

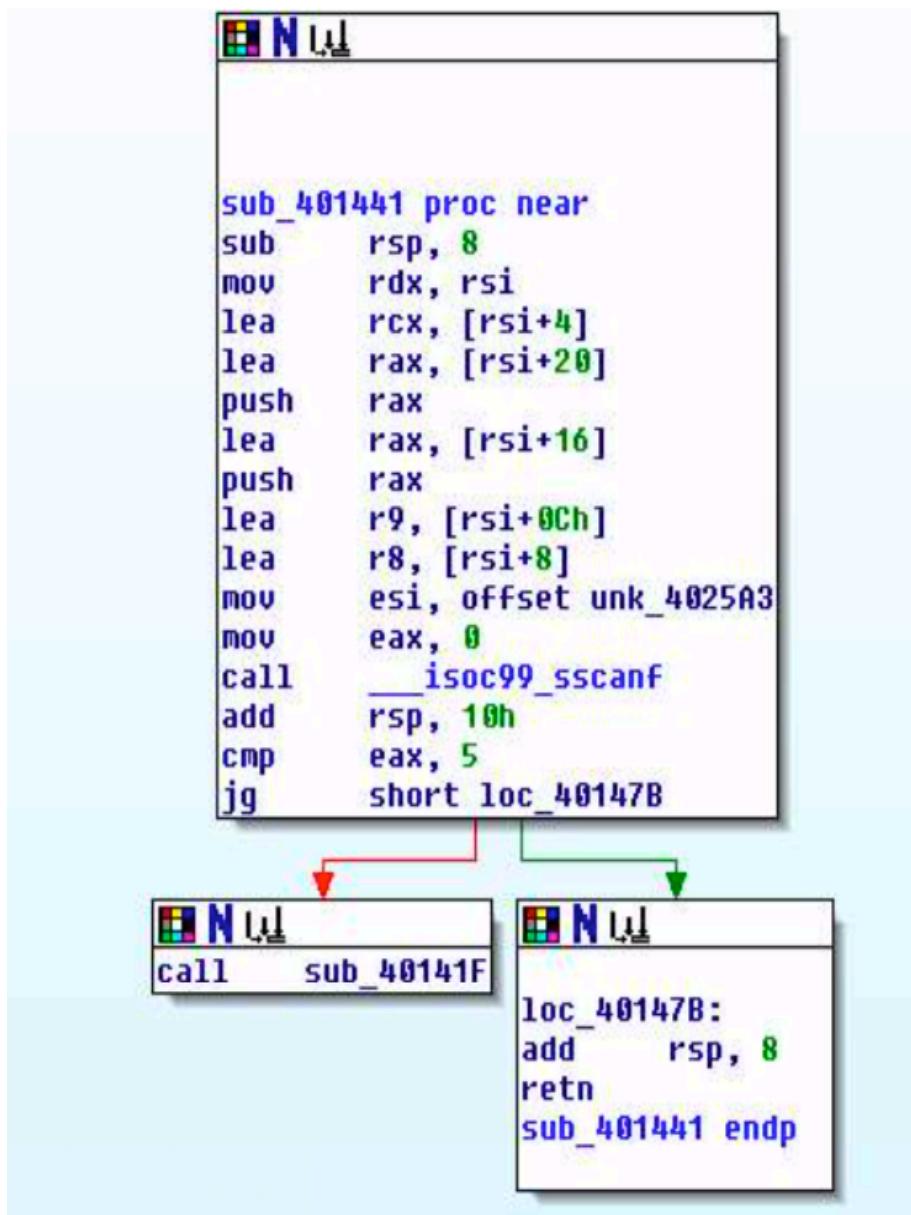
```
NUL
sub_4010CA proc near

var_88= dword ptr -88h
var_70= byte ptr -70h
var_68= qword ptr -68h
var_40= byte ptr -40h
var_30= qword ptr -30h

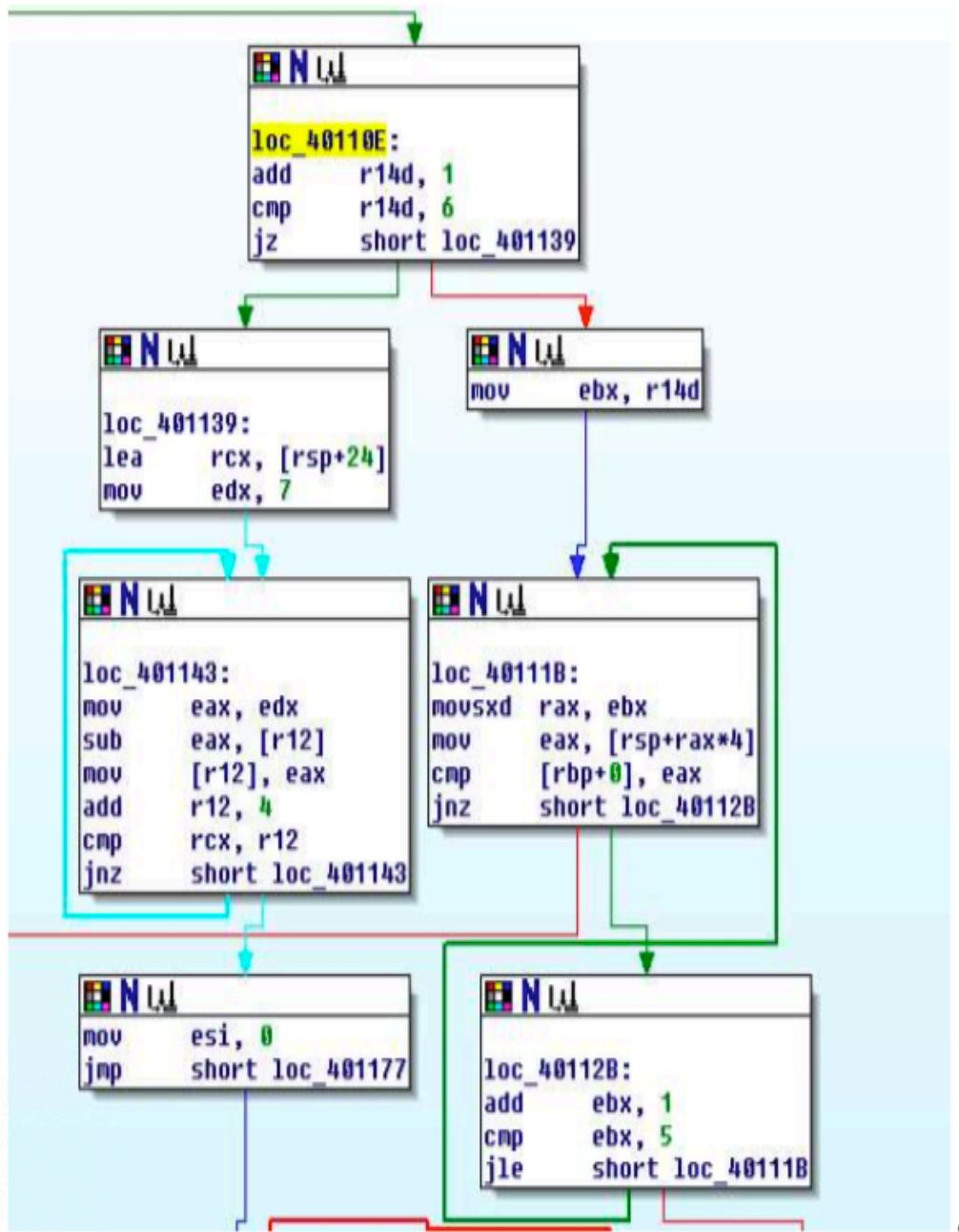
push    r14
push    r13
push    r12
push    rbp
push    rbx
sub     rsp, 96
mov     rax, fs:[+40]
mov     [rsp+88], rax
xor    eax, eax
mov     rsi, rsp
call    sub_401441
mov     r12, rsp
mov     r13, rsp
mov     r14d, 0
```



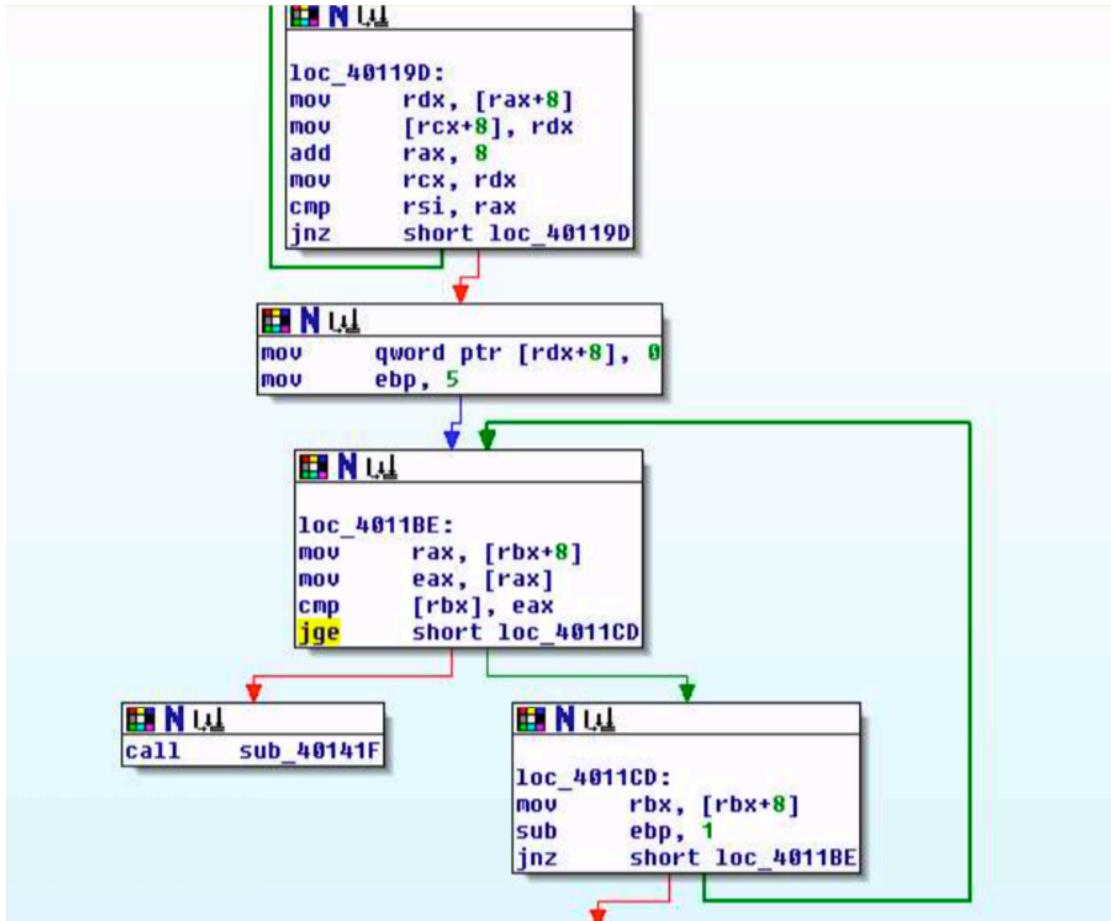
```
NUL
loc_4010FA:
mov     rbp, r13
mov     eax, [r13+0]
sub     eax, 1
cmp     eax, 5
jbe    short loc_40110E
```



With the operation [cmp, eax, 5], the program has a loop that requires each number should be equal or smaller than 6.



Next, in the loc_40110E, the left side indicates that the code subtracts our input number by 7. Then, it gets the new number to replace the old number.



The loop here requires the new sequence in ascending order and the rbx is the first point of all input value. The edp is the controller of the loop. When edp reaches 0, the program would get finished. Backtracking, the answer of Phase 6 is [1,5,3,6,2,4].