Binary Bomb Lab

Table of contents:

- Introduction
- Phase 1
- Phase 2
- Phase 3
- Phase 4
- Phase 5
- Phase 6

Introduction

- Binary bomb is a program that consists of a sequence of phases, each phase expects to type the correct inputs, then the phase is defused and the bomb proceeds to the next phase, otherwise the bomb explodes by printing “BOOM!!!” and then the program is terminating.
- The bomb is defused when every phase has been defused.
- Our goal is to give the correct inputs and defuse the bomb by skipping the bomb explode message.
- This exercise will be solved by using IDA tool.

When loading the program in IDA, first step we want to go to the main function where the program will start from it, as we see in Fig.1.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_E8= qword ptr -0E8h
arg_0= dword ptr  10h
arg_8= qword ptr  18h

mov     [rsp-8+arg_8], rdx
mov     [rsp-8+arg_0], ecx
push    rbp
push    rdi
sub     rsp, 108h
lea     rbp, [rsp+20h]
mov     rdi, rsp
mov     ecx, 42h ; 'B'
mov     eax, 0CCCCCCCCh
rep stosd
mov     ecx, dword ptr [rsp+110h+arg_8]
lea     rcx, unk_140026003
call    j___CheckForDebuggerJustMyCode
cmp     [rbp+0F0h+arg_0], 1
jnz     short loc_140011A26
```

Fig.1

we notice that nothing is interesting yet, Let's scroll down and check what we will find.

```
loc_140011ACD:
call    j_initialize_bomb
lea     rcx, aWelcomeToMyFie ; "Welcome to my fiendish little bomb. You"...
call    j_printf
lea     rcx, aWhichToBlowYou ; "which to blow yourself up. Have a nice "...
call    j_printf
call    j_read_line
mov     [rbp+0F0h+var_E8], rax
mov     rcx, [rbp+0F0h+var_E8]
call    j_phase_1
call    j_phase_defused
lea     rcx, aPhase1DefusedH ; "Phase 1 defused. How about the next one"...
call    j_printf
call    j_read_line
mov     [rbp+0F0h+var_E8], rax
mov     rcx, [rbp+0F0h+var_E8]
call    j_phase_2
call    j_phase_defused
```

Fig.2

In Fig.2, We have found the function calls for the first phases, Alright that's indicates that our program will start it. Let's move to Phase_1 function and see what is going on.

Phase 1

In Fig.3, we found an interesting function "j_strings_not_equal", Firstly, it takes 2 arguments, first argument is stored in RCX register and second argument is moving the address of "aIAmJustARenega" and stored it in RDX register.

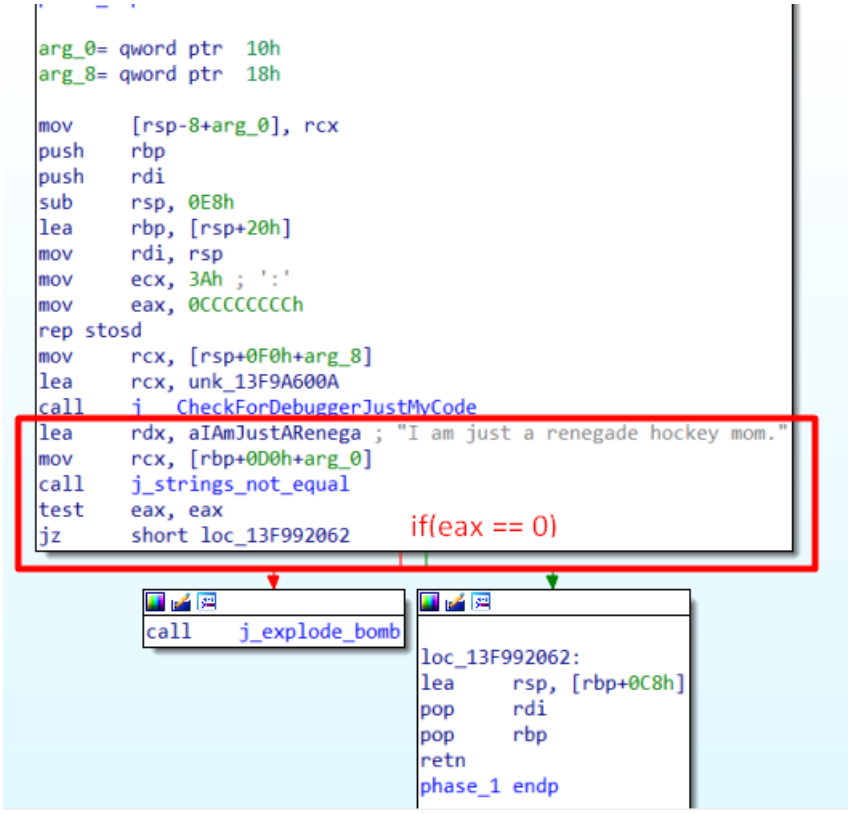


Fig.3

IDA has indicated to what is stored in RDX register. Let's set a breakpoint to phase_1 and run the program and give it any sort of inputs.

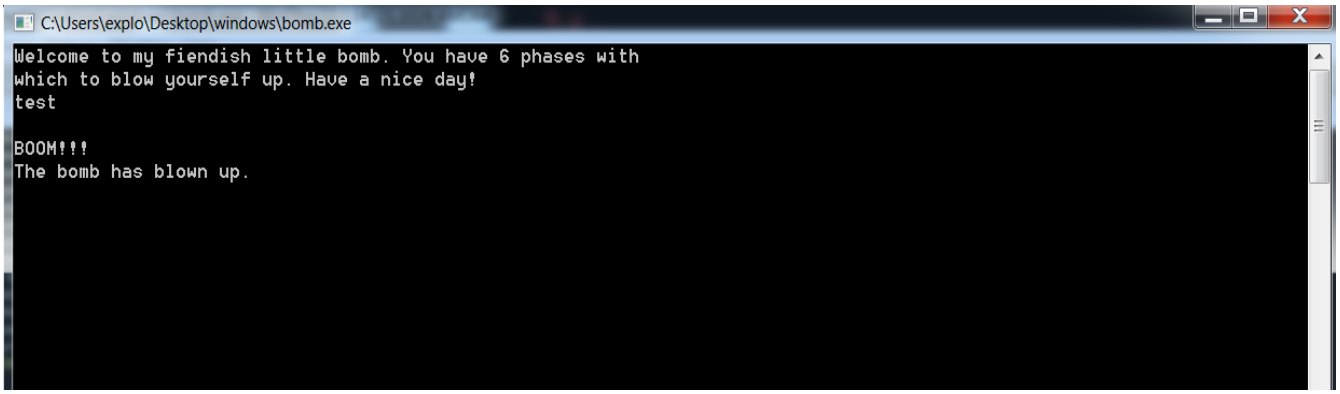


Fig.4

From Fig.4, the bomb is exploded because we give the wrong input, Let's examine it in the debugger.

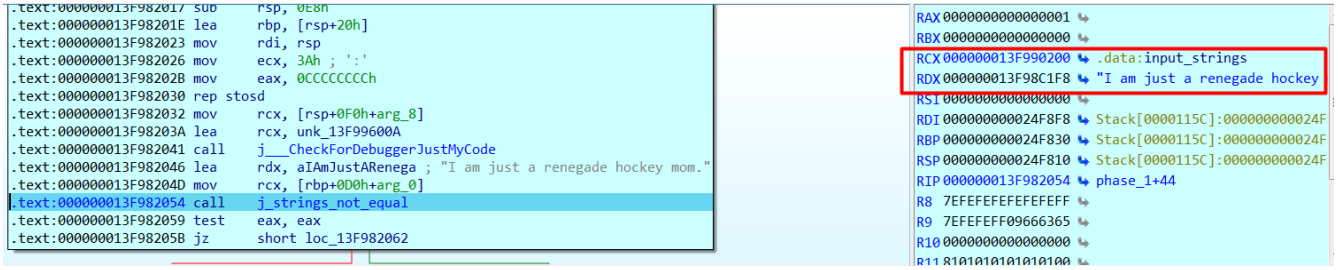
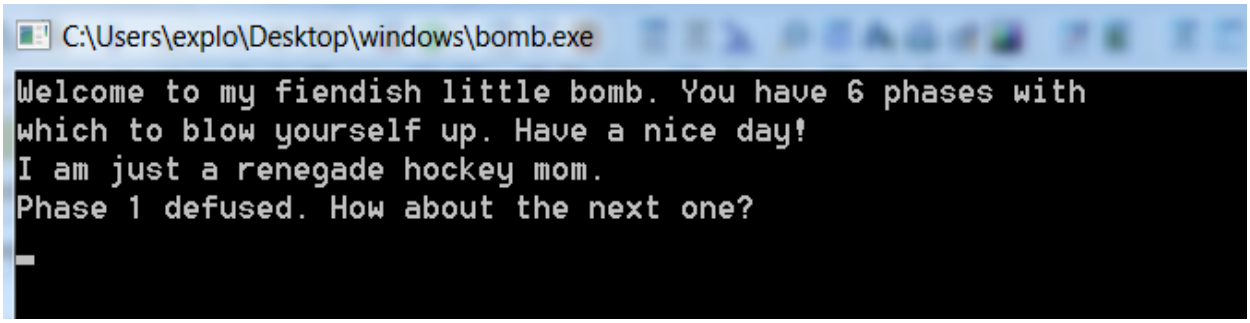


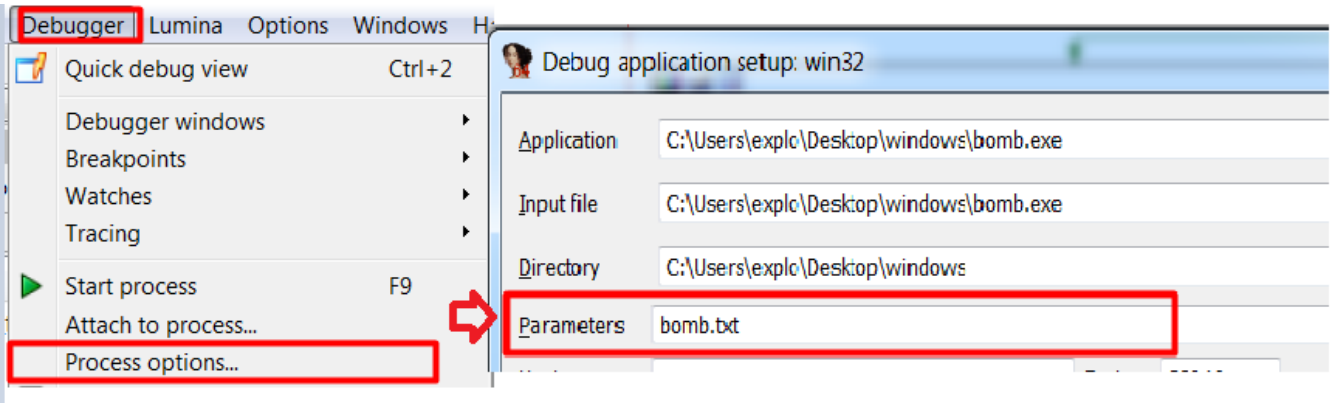
Fig.5

From Fig.5, We notice that our input is stored in RCX register and compared to the string that stored in RDX register, alright so we have indicated what is the job of "j_strings_not_equal" function it checks if the strings are equal to each other or not, It will return non zero value if they are not equal, and will return 0 value if they are equal, and that's what we need, Once the function returns 0, This value will be stored in EAX register and then we find that we have a condition "test eax, eax" this check if the value of EAX will equal 0 or not, If it is 0 then the "Jz" will run and the zero flag will be set.

If the value was not zero, the bomb will be exploded, and now let's try to give to the program the input "I am just a renegade hockey mom."



That's great, As we expected!, we have defused the bomb of phase_1, Let's move to Phase_2. Before moving to phase_2, we forgot to mention something above that this executable file is accepting a file and reading inputs per line so it's like an argument to pass it instead of typing every inputs every time we run the debugger, so we will create a txt file next to the executable file and attach it in IDA like in the following picture



Alright, we will put the first input in bomb.txt file, Move to the next!

Phase 2

In Fig.6, Once we examine the phase_2 function, we will find that it calls another function "j_read_six_numbers" , let's take a look what is inside it.

```
lea     rdx, [rbp+120h+var_F8]
mov     rcx, [rbp+120h+arg_0]
call    j_read_six_numbers
mov     eax, 4
imul    rax, 0
cmp     [rbp+rax+120h+var_F8], 1
```

Fig.6

When we move to it, We will find another call to "j_sscanf", As we see in Fig.7

```
lea     rdx, aDDDDDD    ; "%d %d %d %d %d %d"
mov     rcx, [rbp+0F0h+Buffer] ; Buffer
call    j_sscanf
mov     [rbp+0F0h+var_EC], eax
cmp     [rbp+0F0h+var_EC], 6
jge     short loc_13F06306E

call    j_explode_bomb

loc_13F06306E:
lea     rsp, [rbp+0E8h]
pop     rdi
pop     rbp
retn
read_six_numbers endp
```

Fig.7

this function is reading our inputs, each one is separated by space, also we notice that it takes 6 integer values and storing it in Array,

This function returns the number of fields that we inserted in, and then we find that there is a comparison between 6 and the return value that saved in memory address [rbp+0F0h+var_EC] if it was greater than or equal 6, we will return back to phase_2, otherwise the bomb will be exploded. Alright, So we know now that we are going to give the program 6 integer values

```
C:\Users\explo\Desktop\windows\bomb.exe
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?
9 6 2 3 1 8
```

Let's set a breakpoint on Phase_2 and check the result through debugging it.

```
.text:000000013F1A20E2 call    j_read_six_numbers
.text:000000013F1A20E7 mov     eax, 4
.text:000000013F1A20EC imul    rax, 0
RIP .text:000000013F1A20F0 cmp     [rbp+rax+120h+var_F8], 1
.text:000000013F1A20F5 jz      short loc_13F1A20FC
.text:000000013F1A20F7 call    j_explode_bomb
.text:000000013F1A20FC ; ---------------------------------------
.text:000000013F1A20FC
.text:000000013F1A20FC loc_13F1A20FC:                          ; CO
```

Fig.8

From Fig.8, we notice that the first input must equal 1 to skip the explosion of the bomb so we have an important note now that the first element that stored in the Array is equal to 1. // "arr[0] = 1"

In Fig.9, we will meet a loop that has a counter variable equal to 1 and if it less than 6, it goes to the process of the loop and seems like it does some mathimatical operation.

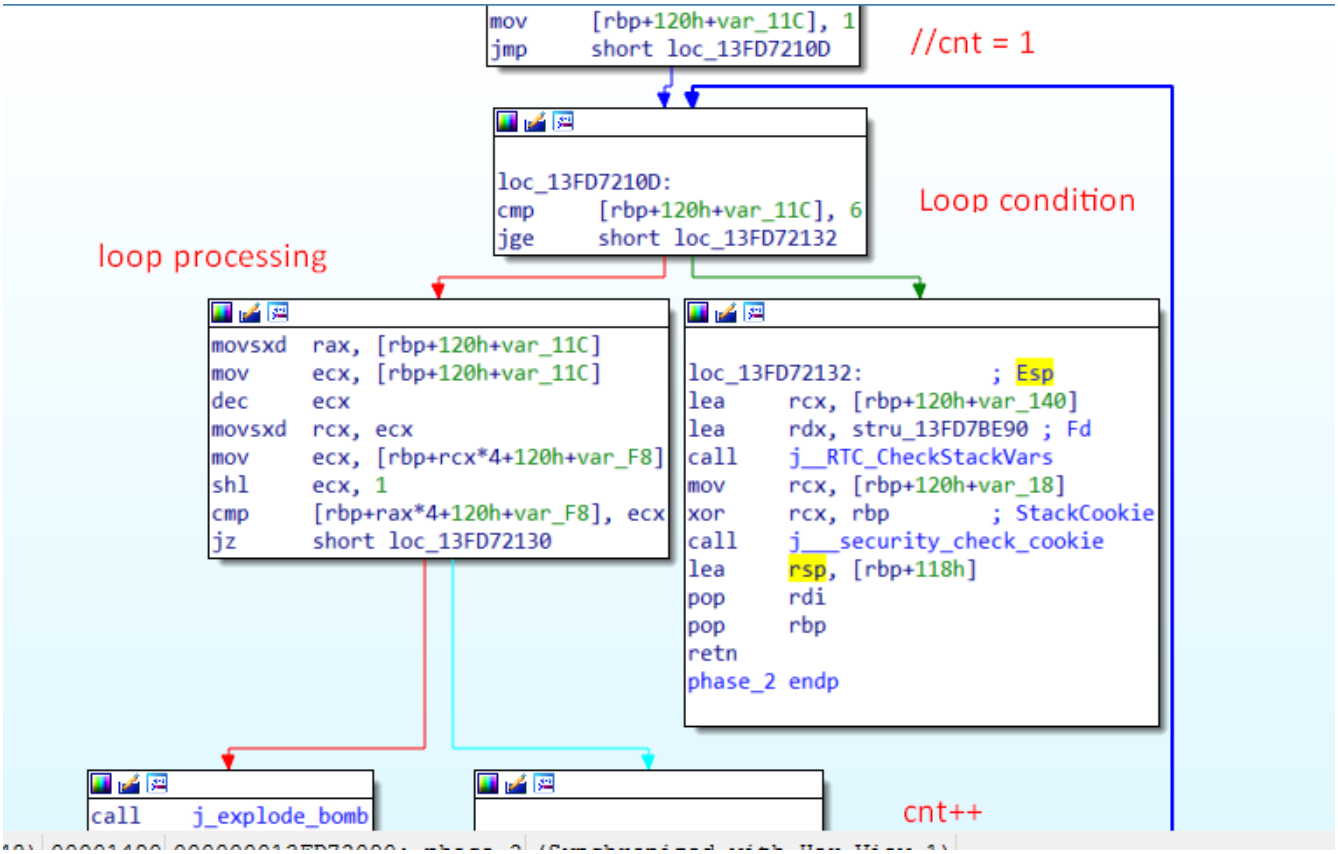


Fig.9

Well, the loop processing is comparing if the current element is equal to the double of the previous element so we are starting from the second element in the array, and as we noted before that our first element is equal to 1, so our program processing will run like what is written in Fig.10.

```
arr[0] = 1;

for(int i=1; i < 6; i++)
{
    if(arr[i] != 2 * arr[i-1])
        explode_bomb();
}
```

Fig.10

So if the first input was 1, then each of the rest of inputs will be doubled by the previous element.

Let’s give a try and insert the inputs and check the result.

```
C:\Users\explo\Desktop\windows\bomb.exe
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?
1 2 4 8 16 32
That's number 2.  Keep going!
```

Seems like we have passed the explosion of the bomb, That’s great!, we have defused the bomb of phase_2!!, Let’s move to the next one.

Phase 3

Let's take a look at the phase_3 function, we will meet "j_sscanf" function again but at this time, it takes 2 integr values as inputs and comparing if the number of inputs are 2 or not as we see in Fig.11

```
mov     [rbp+150h+var_10C], 0
mov     [rbp+150h+var_EC], 0
lea     r9, [rbp+150h+var_12C]
lea     r8, [rbp+150h+var_14C]
lea     rdx, aDD          ; "%d %d"
mov     rcx, [rbp+150h+Buffer] ; Buffer
call    j_sscanf
mov     [rbp+150h+var_EC], eax
cmp     [rbp+150h+var_EC], 2
jge     short loc_13FF9220E

call    j_explode_bomb

loc_13FF9220E:
```

Fig.11

In Fig.12, we will find another comparison between the first input and 7, if it was greater than 7, the bomb will be exploded, otherwise, it will move to switch statements, so we have a note now that the first input will be less than the value 7 to skip the explosion of the bomb.

```
loc_13FF9220E:
mov     eax, [rbp+150h+var_14C]
mov     [rbp+150h+var_1C], eax
cmp     [rbp+150h+var_1C], 7 ; switch 8 cases
ja      short def_13FF92238 ; jumptable 000000013FF92238 default case

2238:           ; jumptable 000000013FF92238 default case
_explode_bomb

movsxd  rax, [rbp+150h+var_1C]
lea     rcx, cs:13FF80000h
mov     eax, ds:(jpt_13FF92238 -
add     rax, rcx
```

Fig.12

IDA has indicated that there is a switch statement includes 8 cases, oh! That's a lot but that doesn't mean that we will examine all, we will discover where the next explosion of the bomb to skip it because it's our main goal. An important note to mention, that the condition of switch is depending on the first input value where it will move to the labeled statements.

Let's scroll down and check what we will find.

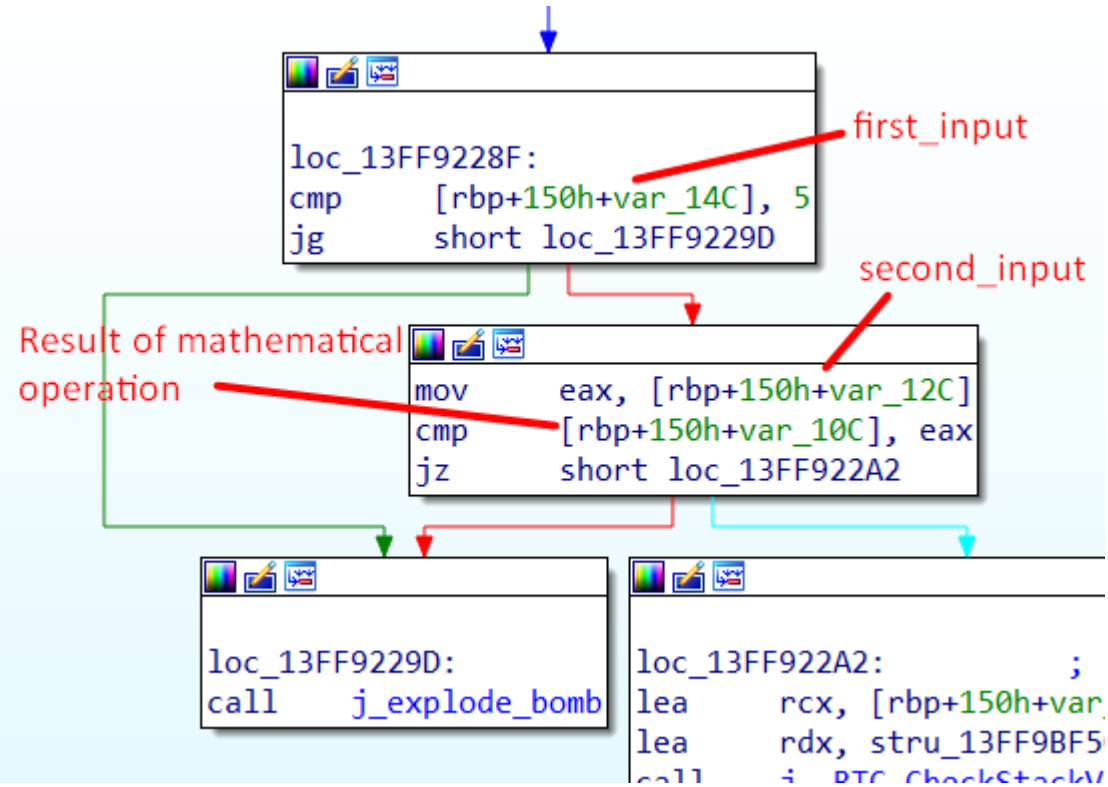
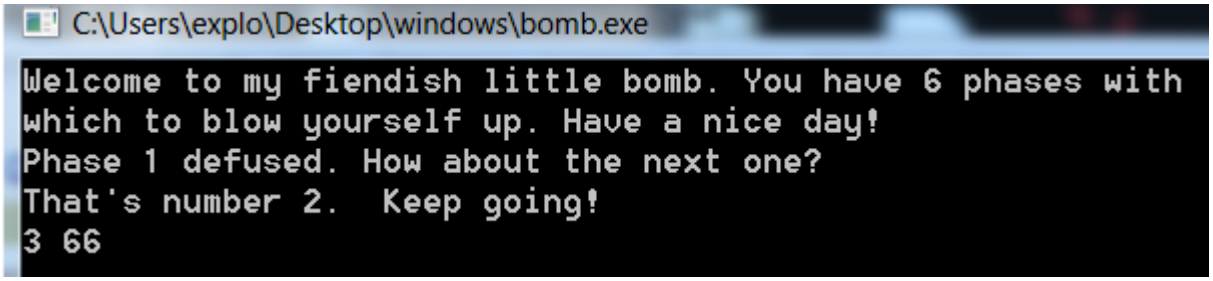


Fig.13

From Fig.13, we noticed that the first input is compared to value 5 and that's another point to know, that the first input will be at least less than 5, so we will give it a random input in a range that at least more than 0 and less than 5.



Let's set a breakpoint at Phase_3 and the comparison between the first input and value 5 and check what is going on.

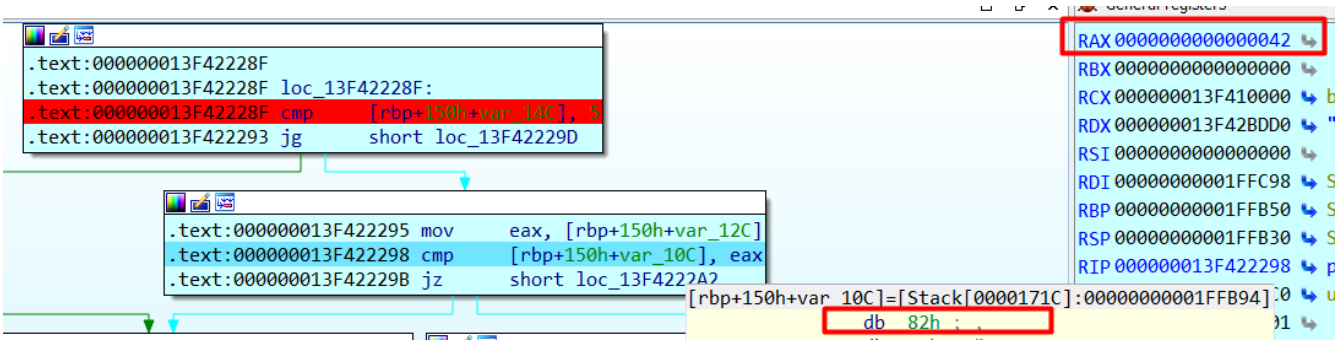
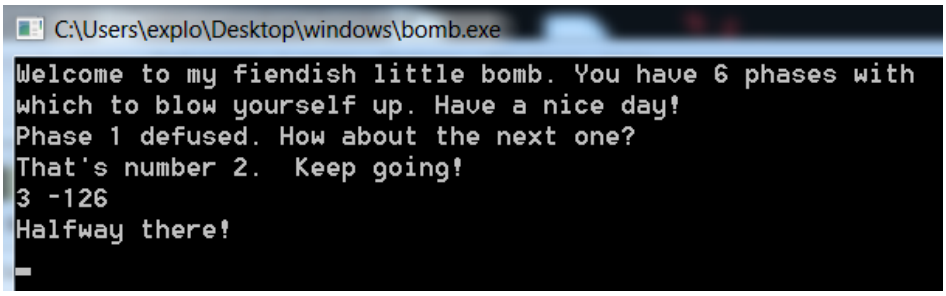


Fig.14

In Fig.14, we have skipped the explosion of the bomb, but unfortunately we don't give the correct second input, the second input depends on the result of the mathematical operation in switch statement (case 3) and this value must equal the second input we give to the program. Well the result of this mathematical operation is -126, It's normal add/sub operations. Let's check if these inputs are correct or not!



Alright, That's great!, we have defused the bomb of phase_3!!, Let's move to the next one.

Phase 4

At the phase_4 function, we will meet "j_sscanf" function again and also it takes 2 integr values as inputs and comparing if the number of inputs are 2 or not, also we will find that there is a check on the first input if it was less than 0 or larger than 14, the bomb will be exploded, As we see in Fig.15

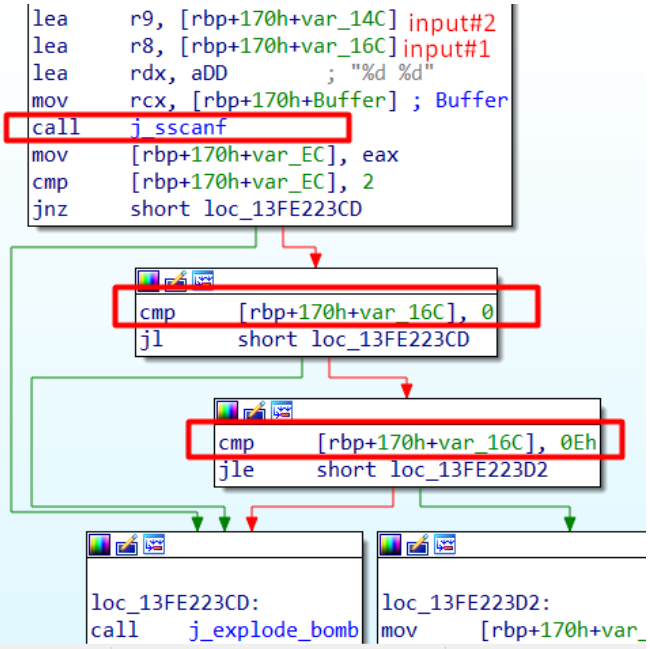


Fig.15

When scrolling down, we will find that we have a variable has integr value 10, then the first input is passed to the "j_func4", well seems like it's an interesting function but let's continue to check where is the next explosion, Alright if we look at the Fig.16 we will notice that the return value from the "j_func4" must equal 10 because there is a comparison between it and var1, then the second input is compared to var1 which indicates to the value 10 to skip the explosion of the bomb.

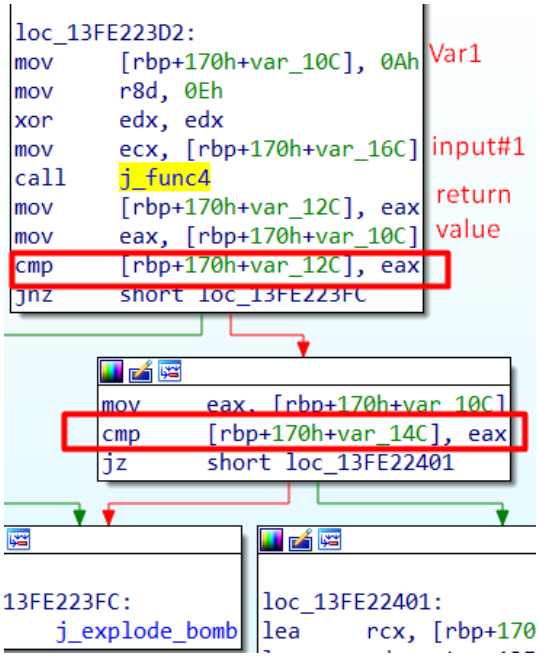


Fig.16

Right now we have two important notes:

1. First input must equal a value larger than 0 and less than 14.
2. Second input must equal the value 10.

Let's check what is happening in "j_func4" to get the value of the first input

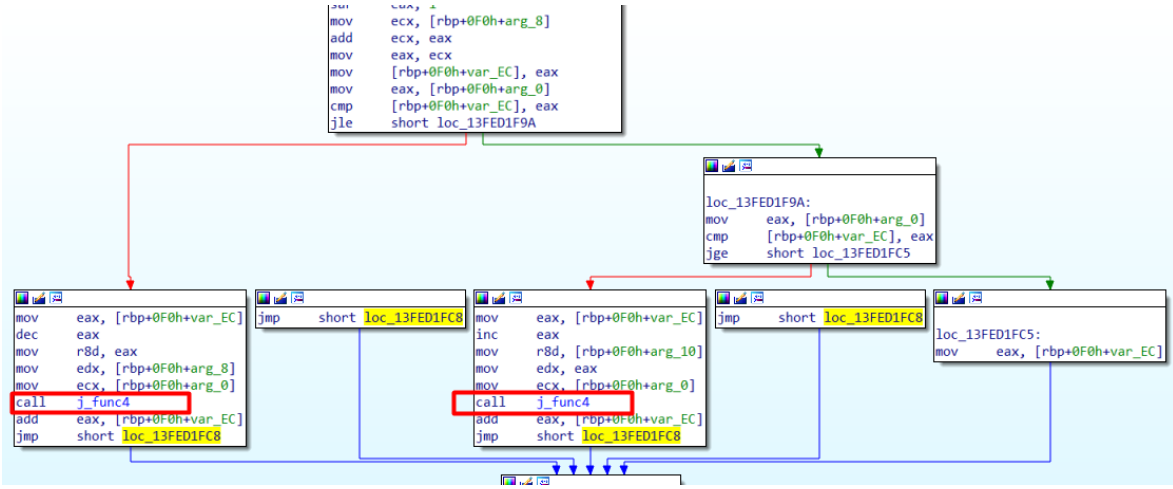


Fig.17

From Fig.17, The function indicates that it does some recursion mathematics operations, So let's give the first input a random number in the range we have mentioned before.

```
C:\Users\explo\Desktop\windows\bomb.exe
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!
6 10
```

Let's put breakpoints to Phase_4 & j_func4 and check the result of RAX register by the debugger.

```
loc_13FA723D2:
mov     [rbp+170h+var_10C], 0Ah
mov     r8d, 0Eh
xor     edx, edx
mov     ecx, [rbp+170h+var_16C]
call    j_func4
mov     [rbp+170h+var_12C], eax
mov     eax, [rbp+170h+var_10C]
```

```
General registers
RAX 0000000000000015
RBX 0000000000000000
RCX 0000000000000006
RDX 0000000000000000
RSI 0000000000000000
RDI 00000000001FF5A8 Stack[
RBP 00000000001FF440 Stack[
RSP 00000000001FF420 Stack[
RIP 000000013FA723E9 phase
```

Fig.18

From Fig.18, we check the result of RAX is 15h, that's not the value we need, Through debugging, I have noticed that if value of the first input was more than 5, the value of RAX is getting increased, and will be decreased if the first input was less than 5. So if we try to give the first input the value 3 we will get the result that needed to defuse this phase.

```
C:\Users\explo\Desktop\windows\bomb.exe
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!
3 10
So you got that one.  Try this one.
```

That's great, we have defused the bomb of phase_4!!, Let's move to the next one.

Phase 5

The start of phase_5 is like phase_4 where it takes 2 integr values as inputs and comparing if the number of inputs are 2 or not, alright, we will notice that the first input has AND operation with 0Fh, simple any value has AND operation with 0Fh will give us the same value, Example: (5 & 0Fh = 5) Also there is two variables initialized by zero as we see in Fig.19

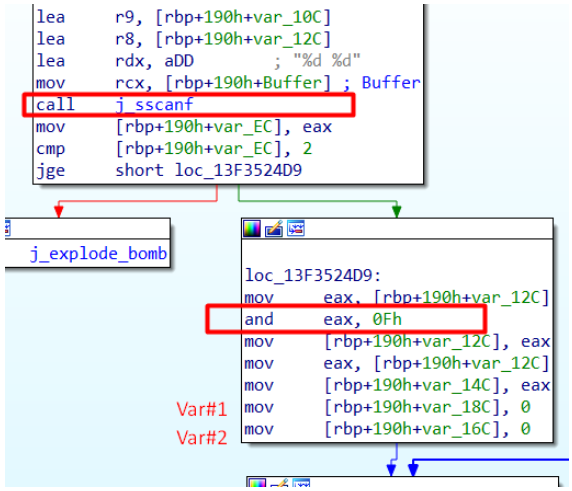


Fig.19

When scrolling down and check the rest of the function, we will notice as found in Fig.20 that there is a loop and its condition between the first input and 0Fh, but to perform the operations inside the loop, the first input shouldn't equal the value 15 or the result will go to explode the bomb, another thing our loop should repeat it self by 15 times why ? when the loop ends, we will meet another comparison where it indicates that the value of Var1 should equal 15 to skip the explosion of the bomb. Alright, We have important notes now:

1. First input value shouldn't equal 15.
2. Var#1 value should equal 15.

Let's explain the operations that is running inside the loop.

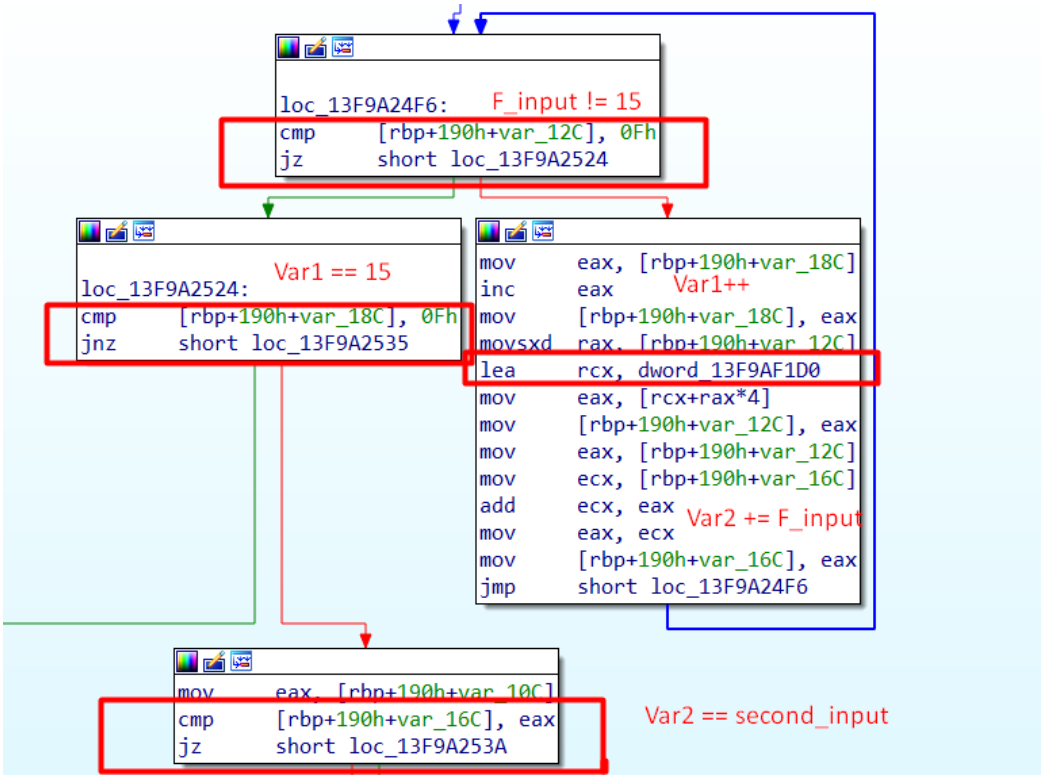


Fig.20

Firstly, it increases the value of Var1 by 1, then RCX register points to an address in the memory, includes an array that have 44 elements.

```
dword_13F35F1D0 dd 0Ah, 2, 0Eh, 7, 8, 0Ch, 0Fh, 0Bh, 0, 4, 1, 0Dh, 3, 9
                                                ; DATA XREF: phase_5+A8↑o
                dd 6, 5, 1Ch dup(0)
```

As we see from Fig.20 that the value of first input will point to an element from this array and pass it again to the variable of the first input and then we will have a sum operation between Var2 and the value of the first input, then the loop will be exit when the value of the first input becomes 15, at this time the value of Var1 should equal 15 too, then we will meet the last comparison that between Var2 where it refers to the sum operation that done inside the loop and the second input that we give to the program, both values should equal each other to skip the explosion of the bomb.

The following picture refers to a simple program to give us the inputs we need to defuse the bomb.

```
#include <iostream>
using namespace std;

int main()
{

int j=0, v1=0, v2=0;
//                0  1   2   3  4  5   6   7   8  9 10  11 12 13 14 15 16 17 18 19 20 21
int arr[44] = { 10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };

for(int i=21; i >= 6; i--)
{

    v1 = i & 15;

    while(v1 != 15)
    {
        j++; // This value must equal 15 to skip the explosion of the bomb!
        v1 = arr[v1];
        v2+=v1;
    }
    if(j == 15)
    {
            cout << "value of first input = " << i << " \n" << "value of second input = " << v2 ;
            break;
    }
}

}
```

Results of the program:

```
value of first input = 21
value of second input = 115
Process returned 0 (0x0)   execution time : 0.076 s
Press any key to continue.
```

Let’s try to give the program these results and check if its right or not.

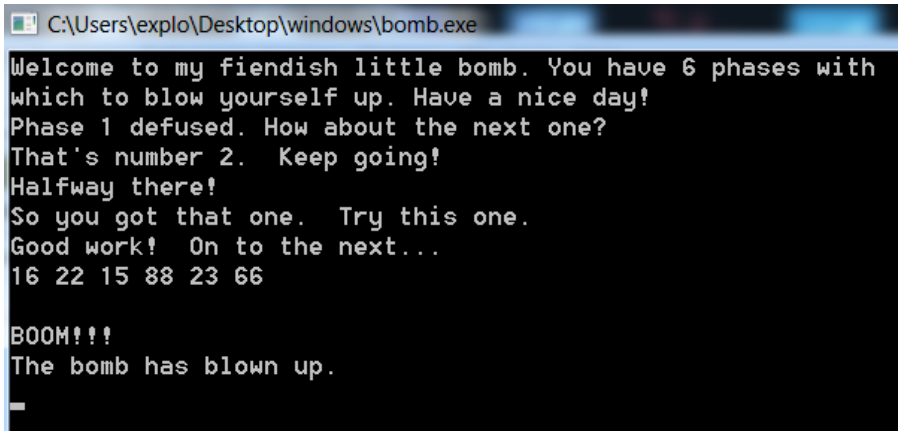
```
C:\Users\explo\Desktop\windows\bomb.exe
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
21 115
Good work!  On to the next...
```

Seems the results are accepted.

That’s great, we have defused the bomb of phase_5!!, Let’s move to the next one.

Phase 6

The start of phase_6 is like previous phases but it takes 6 integr values as inputs and comparing if the number of inputs are 6 or not, the values are stored in Array, So let's test the program and give it 6 random values then check the result.



The explosion has been done, Let's analyse phase_6 and check what is running around it. Alright, This phase have much things that is not interested to us, so like we did before we will focus on where the explode of the bomb is located.

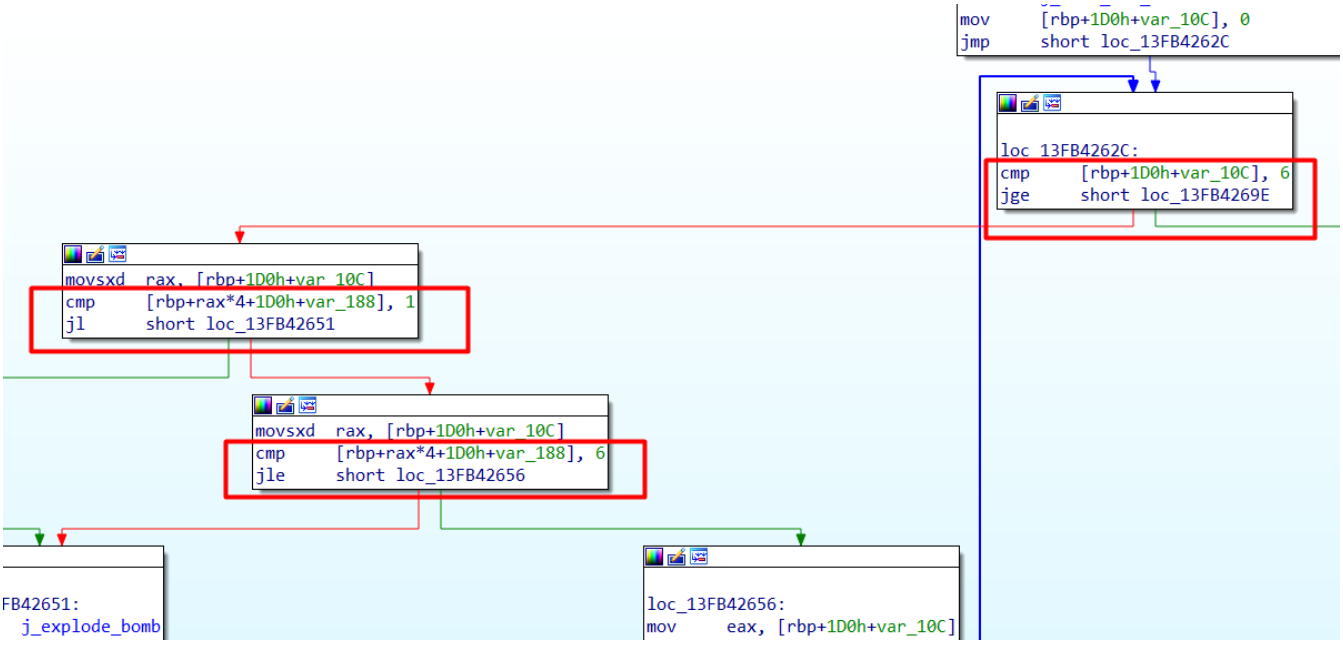


Fig.21

From the above picture, It indicates that it's a loop and has an if conditions where it checks if the values of the array is greater than or equal 1 or less than or equal 6, otherwise it will explode the bomb, So let's give the program inputs from 1 to 6 and check the result.

```
C:\Users\explo\Desktop\windows\bomb.exe
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...
1 3 5 6 6 4

BOOM!!!
The bomb has blown up.
```

The explosion has been done again, Let's scroll down and analyse the rest of this loop.

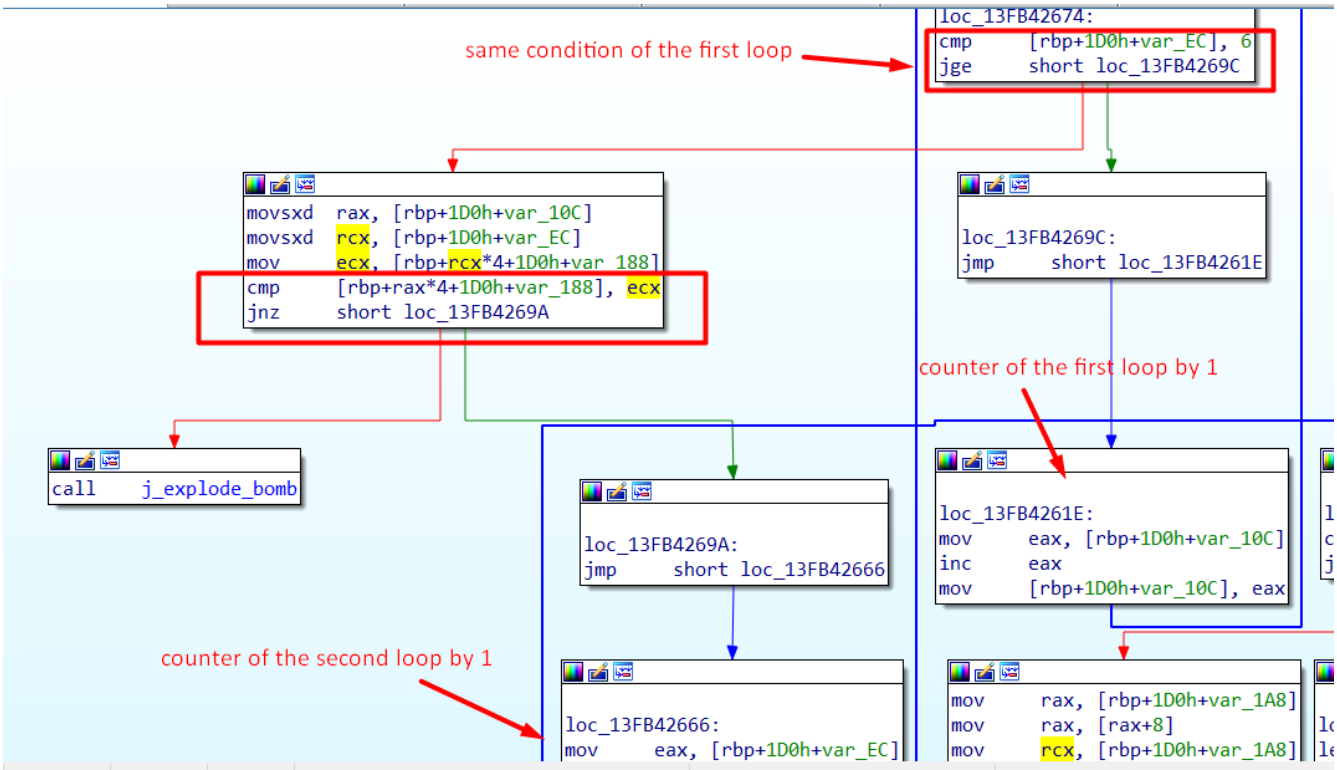
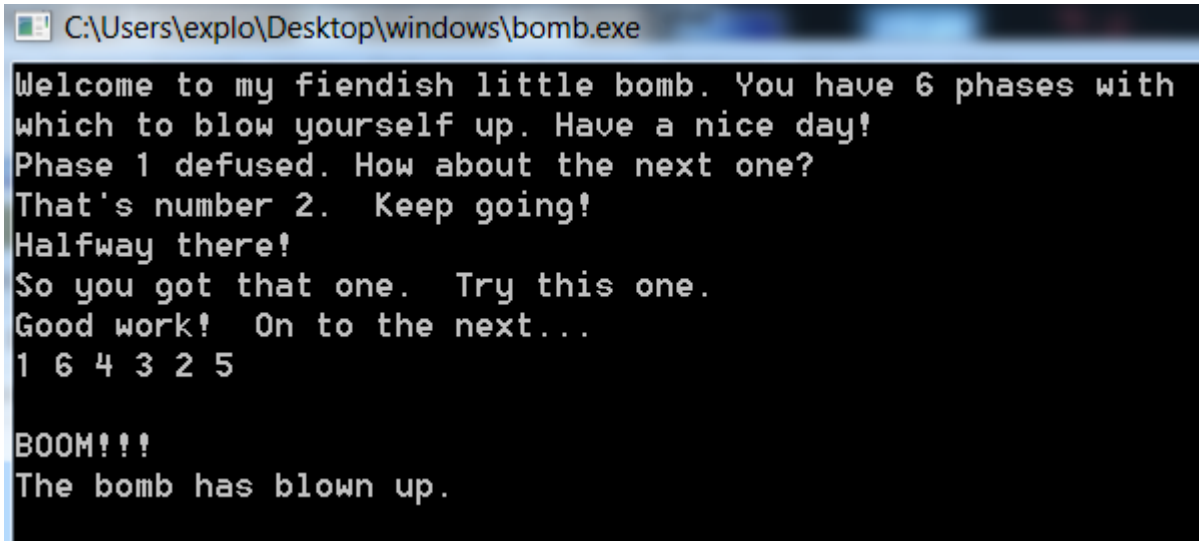


Fig.22

From Fig.22, we will notice that we are going to enter another loop inside the first loop, so its considered as nested loop, well the inner loop checks if the value of the array is repeated once again or not, if it is repeated, the explosion will happen. We have important notes now:

1. Values of the array greater than or equal 1 and less than or equal 6.
2. Values of the array shoundn't be repeated.

Let's give the program inputs from 1 to 6 with no repeated values.



The explosion has been done once again. When we check the next explosion we will find much stuff in the CFG that makes the program very complicated to find what is happening so we will set a breakpoint before the explosion and check what is going on. Oh!, an important point forget to mention when the phase takes 6 inputs values, there was a pointer to an address in the memory called node1, when I moved to it, I noticed that it has two integr values and an offset address to another node, I have checked all nodes and found that they are 6 nodes, each one of them is pointing to the next one. It's like the linked list data structure as we see in Fig.23

```
lea     rax, node1
mov     [rbp+1D0h+var_1C8], rax
lea     rdx, [rbp
mov     rcx, [rbp
call    j_read_si
mov     [rbp+1D0h
jmp     short loc

node1           db  12h                 ; DATA XREF: phase_6+47↑o
                db   2
                db   0
                db   0
                db   1
                db   0
                db   0
                db   0
                dq offset node2
loc_13FF9262C n48  db 0E9h ; é          ; DATA XREF: .data:000000013FF9F130↓o
```

Fig.23

Well the program now is looking for specefic order for these values, let's check what we will find through debugging.

```
mov     eax, [rax]
cmp     [rcx], eax
jge     short loc_13FFB27C3

lea     rdx, stru_13FFBC1C0 ; Fd
call    j__RTC_CheckStackVars
mov     rcx, [rbp+1D0h+var_18]
xor     rcx, rbp        ; StackCookie
call    j___security_check_cookie
lea     rsp, [rbp+1C8h]
pop     rdi
pop     rbp
retn
phase_6 endp

call    j_explode_bomb

RAX 00000000000003A7
RBX 0000000000000000
RCX 000000013FFBF050  .data:node1
RDX 000000013FFBBDD0  "H"
RSI 0000000000000000
RDI 000000000030FC18  Stack[000016F4]:
RBP 000000000030FA50  Stack[000016F4]:
RSP 000000000030FA30  Stack[000016F4]:
RIP 000000013FFB27BA  phase_6+20A
R8  000007FEE9F630C0  ucrtbased.dll:00
R9  0000000000000001
R10 0000000000000068
R11 000000000030F358  Stack[000016F4]:
R12 0000000000000000
R13 0000000000000000
```

Fig.24

From Fig.24, we notice that the content of RCX register is compared to RAX register if it was less than it, then explosion will happen, well the nodes are doing a simple operation in order to arrange to value that is inside it, I will show the content of each node and the operation it does in the following table to clear more what is running around.

Nodes	Assigned_Values	Node_Value1	Node_Value2
Node1	12h	2	1
Node2	C2h	1	2
Node3	15h	2	3
Node4	93h	3	4
Node5	A7h	3	5
Node6	0h	2	6

As we see from the table, each node has a unique value and contain two other values, well the operation that has been done through the debugging in order to obtain an address like what we see in RAX register as found in Fig.24

Result of the address = Node_Value1 * 100h + Assigned_Value

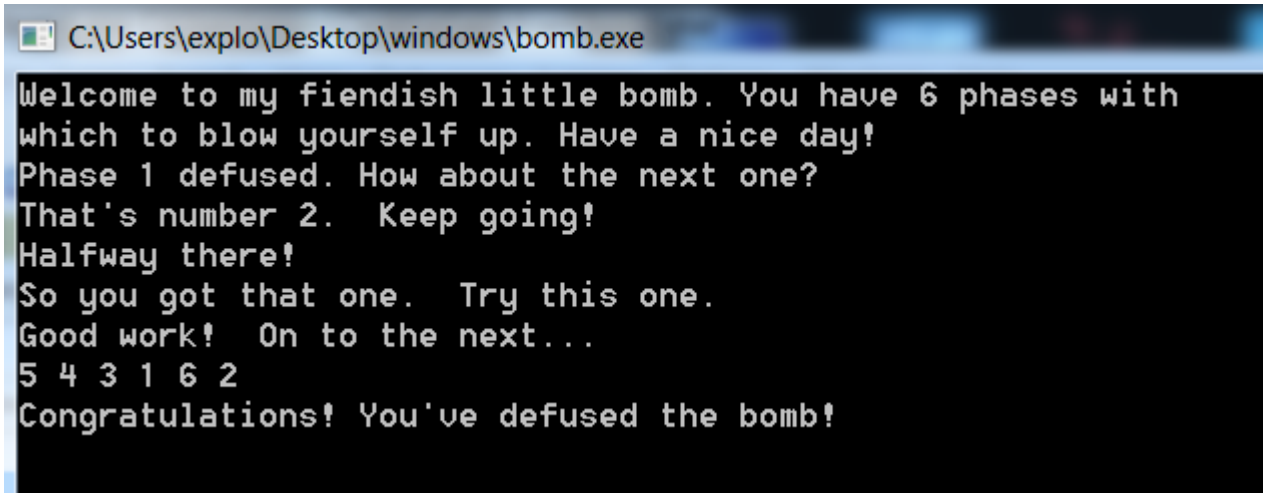
After calculating the operation for each node we will have another table:

Nodes	Address_In_Hex	Address_In_Dec	Node_Values
Node1	212	530	1
Node2	1C2	450	2
Node3	215	533	3
Node4	393	915	4
Node5	3A7	935	5
Node6	200	512	6

Alright, everything is clear now, if we take a look again at Fig.24 to the comparison of RCX and RAX registers, we will notice that the program is arranging the values in descending order that depends on the value of the address so if we arranged each address_in_dec by the node_value that assigned to it in a table we will find the correct inputs to skip the explosion of the bomb.

935	915	533	530	512	450
5	4	3	1	6	2

Let's try to give the program these inputs in this order and check the result.



As we expected!!!, Finally, we have defused the bomb of phase_6 and finished all phases of the program.

Presented by: Abdelrahman Alaa