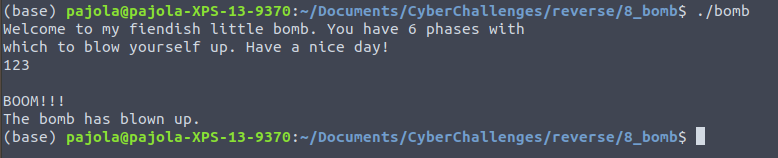
Reverse 3: Bomb

We can start by executing the code.

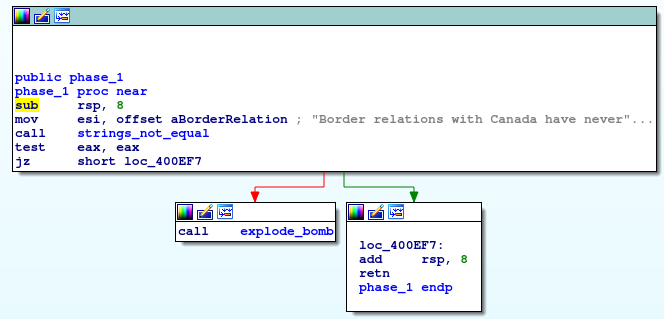


We can understand the following:

* 6 phases must be passed;
* If you do something wrong the bomb will explode.

Let’s hope that IDA can help us. First of all, we see that there are 6 functions, one per each phase, so my idea right now is to study one phase per time (I am supposing that they are unrelated).

## Phase 1

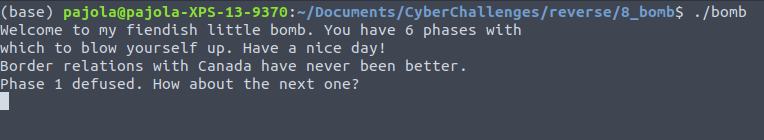


Mmm… “Border relations [...]” seems suspicious. If something goes wrong, the bomb explodes.

The string says:

*“Border relations with Canada have never been better.”*

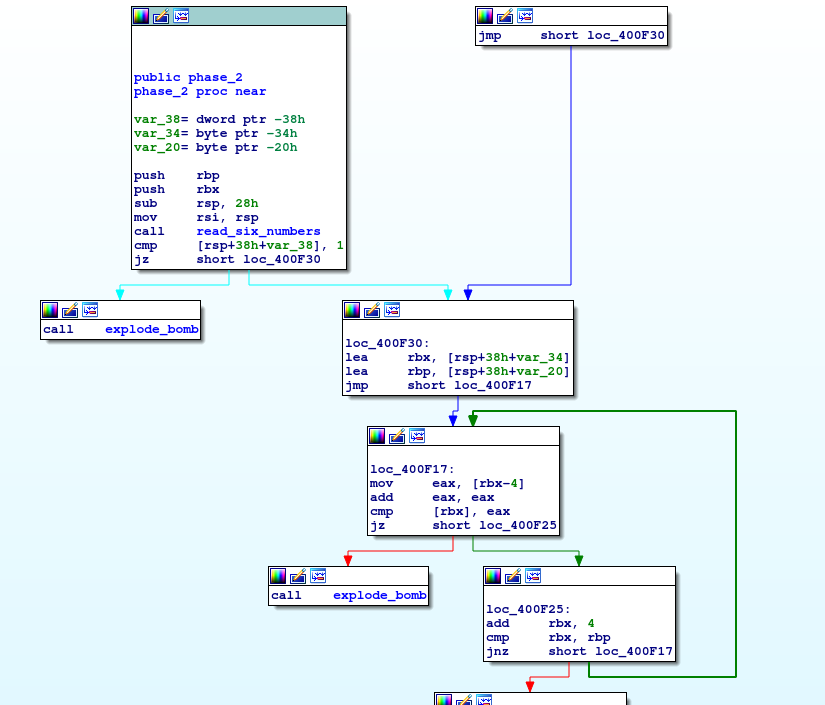
We can try to put this as value.



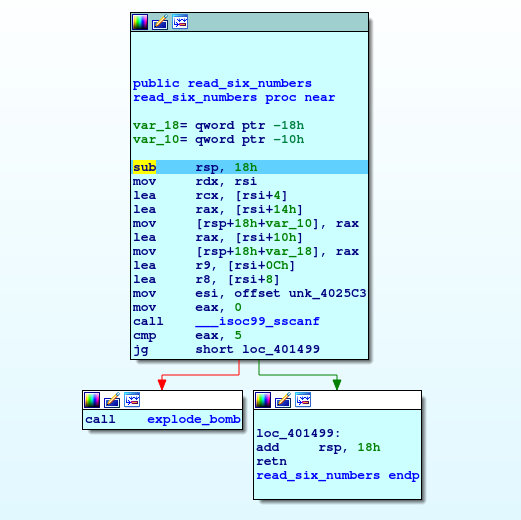
The bomb did not explode (good sign).

## Phase 2

First of all, in the main we have that the program uses read\_line to get the user input, and put it on rdi. Then, it calls phase\_2.



First, it’s subtracting 28 from the stack pointer, which probably means it’s preparing space to store some variables. It movs rsp into rsi, and calls read\_six\_numbers.



Here it’s again subtracting 0x18 from rsp, so probably other variables will be used.

Now follows a series of mov and lea, and finally a call to function sscanf. Here comes the first tricky part, since we have to understand how sscanf works:

int sscanf(const char \*str, const char \*format, ...) reads formatted input from a string.

Parameters

* str − This is the C string that the function processes as its source to retrieve the data.
* format − This is the C string that contains one or more of the following items: Whitespace character, Non-whitespace character and Format specifiers

A format specifier follows this prototype: [=%[\*][width][modifiers]type=], where type is for example s for string and d for decimal integer.

However, there can be **other arguments:**

− This function expects a sequence of pointers as additional arguments, each one pointing to an object of the type specified by their corresponding %-tag within the format string, in the same order.

For each format specifier in the format string that retrieves data, an additional argument should be specified. If you want to store the result of a sscanf operation on a regular variable you should precede its identifier with the reference operator, i.e. an ampersand sign (&), like: int n; sscanf (str,"%d",&n);

We can see that the registers used before the calls are:

* Rdi: that contains the string inserted by the user
* Esi: contains a string loaded from a specific address. Inspecting it, it is %d %d %d %d %d %d
* Rdx, rcx, r8, r9 which are filled with addresses starting from rsi and adding +0x4 every time.
* We notice that rsi+10 and rsi+14 are moved into the stack

Remembering how functions are called in x86, the first 6 parameters are put in rdi,rsi,rdx,rcx,r8,r9 and the others go to the stack. So basically, the function is calling sscanf giving the string in rdi, the format in rsi, and the addresses in memory where to store the values sscanf is reading. And they are starting from rsi, in which in the previous function (phase2) was exactly rsp. So when the function returns, in rsp the function phase2 will have the user input formatted in decimals.

To conclude this explanation, sscanf returns the number of input correctly read based on the passed format (%d %d %d %d %d %d), which is six integers. If they are 5 or less, the bomb will explode.

Going back to phase2, we understood that the program is expecting 6 integer numbers, each one every 4 bytes starting from rsp. The Input then should be something like this:

x1 x2 x3 x4 x5 x6

In the first comparison we read the first number (rsp + 38h + var\_38 (which is -38h) == rsp+0h) and we compare it with 1, if True the bomb doesn’t explode.

1 x2 x3 x4 x5 x6

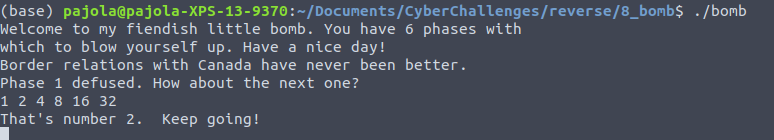
We load the second number (rsp+4d) in the register *rbx* and in *rbp* it’s loading rsp+18h (rsp+24d) . We don’t know yet what’s this, but let’s move on. Then, the block loc\_400F17 basically do the following comparison: (*array[i - 1]\*2) == array[i]* where:

* *mov eax, [rbx - 4]*: eax contains *array[i - 1]* since rbx contains *array[i]*
* *add eax, eax* is equivalent to *array[i - 1]\*2*
* *cmp [rbx], eax* is equivalent to (*array[i - 1]\*2) == array[i]*

And it does it cycling through the six numbers (increasing rbx by 4) until rbx is different from rbp. Since rbx increments of 4 every variable, to read 6 numbers we will have rbx+20d, since it starts from rbx+0, and so at rbx+24d there would be the 7th number, which doesn’t exist because of the previous check. So when 6 correct numbers are compared, the function ends.

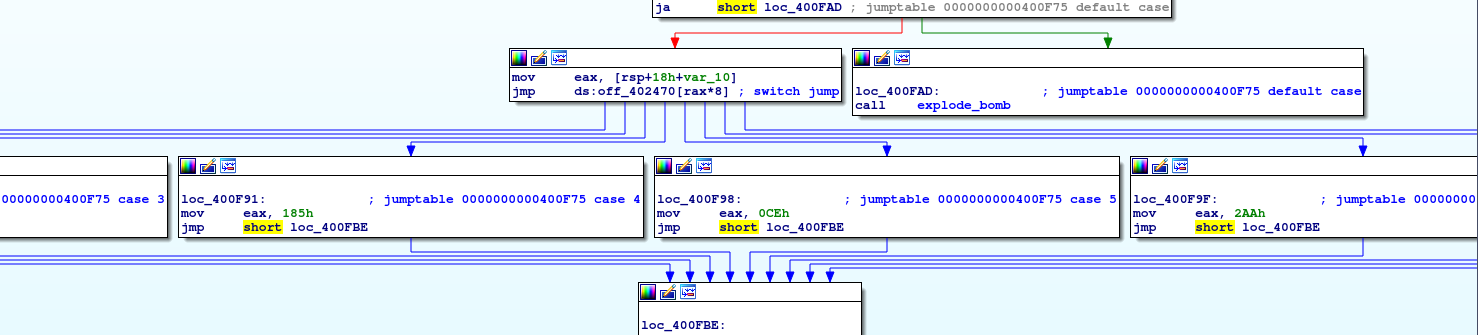
Then, starting from 1, the input should be the following one:

1 2 4 8 16 32

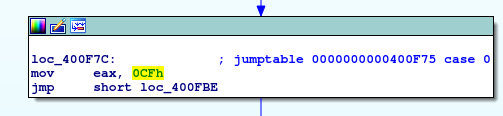


## Phase 3

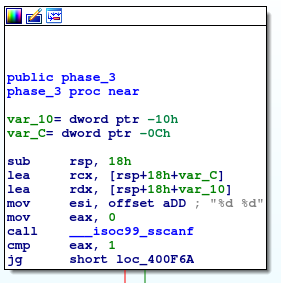
This is a mess. What we should notice is a “switch” instruction.



For example, one case of the switch is the following one:



Ok, first we need to understand the input format:



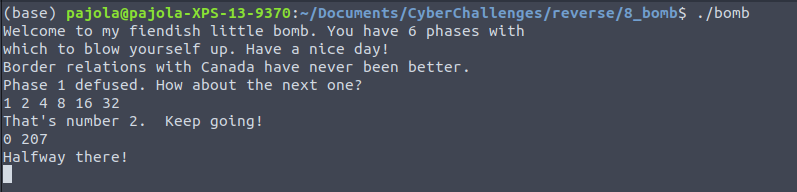
From the comment “%d %d” I guess that we are talking about two integers. In the offset with “18h + var\_10” we store the first number, in “18h+var\_C” the second.

The two values stored in “18h + var\_10” and “18h+var\_C” are used as follows:

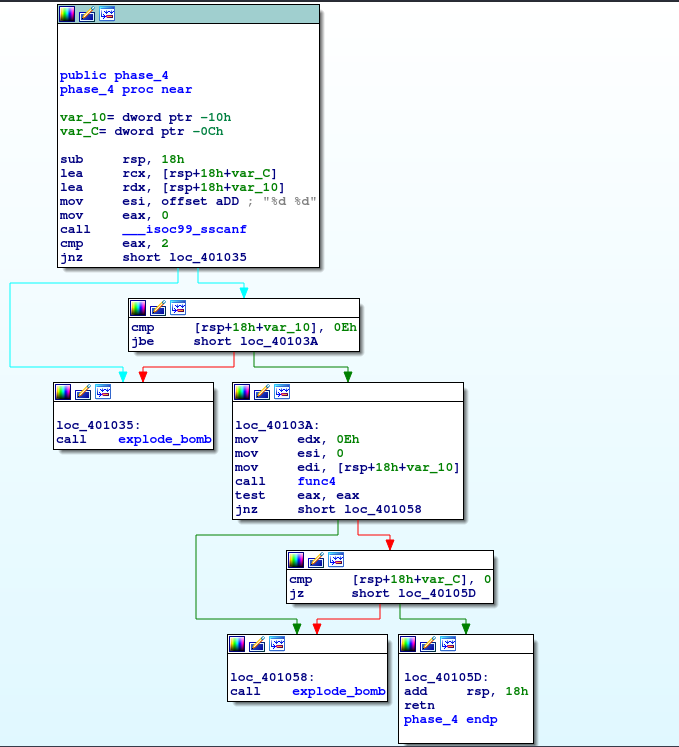
* “18h + var\_10”: this is used to choose the switch case
* “18h+var\_C”: this is compared with a constant value saved in the specific switch case (e.g., 0CFh, 185h, 0CEh)

Thus, if we choose the “0” switch case, we can see that the constant value used for the comparison is 0CFh, which is equivalent to 207. We try with the following input:

0 207



## Phase 4



As we did previously, we can notice that we need to guess two numbers. We can see that:

* [18h + var\_10] must be <= 0Eh (i.e., 15);
* [18h + var\_C] must be 0 (0x401051).

In *loc\_40103A*, *func4* is called, with *var\_10* as parameter. Its return value is tested against zero (in instruction *test eax, eax*). Hence, we would like to have a return value for func4 equal to zero.

We need to understand what the function *func4* does and you can use the python reimplementation that we provided or try to solve it by yourself. What you should notice is that:

* It is recursive;
* If we go on the base case (that returns 0) we avoid all the other calls and thus, we just need to understand how the base case is defined.

Based on this, you can find that the right combination is:

7 0

