* Phase\_1:
  + So, I created a breakpoint at phase\_1 before running to see what was going on and I see this function being called <strings\_not\_equal>. From there I see that this function is used to tell if two strings are equal using objdump -d to look at the assembly code. I look at the strings of the bomb using <strings bomb> command to see an odd string that is probably used to check this phase1. I test this string, and it works. Phase\_1 defused.

A close up of text on a black background

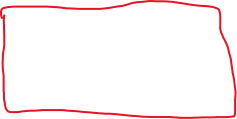
Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screen shot of a computer

Description automatically generated



* Phase\_2:
  + Using <objdump -d bomb> command to see a list of all the function in the file, I see phase\_2 is calling a function called read\_six\_numbers. After inspecting this, I notice this function is used to make sure the input has 6 integers, separated by white space. Now I just need to figure out what these 6 numbers should be. In phase\_2 we see a comparison to 1, so I know the first number is going to be 1. Then I see a comparison between the first number + first number and the next value after 4 bytes. This must be 2, since the first number must be 1. That is a pattern that will continue to loop until the end of the sequence I reached as this is a looping function. So final sequence will be 1,2,4,8,16,32.

A screenshot of a cell phone

Description automatically generated



A screenshot of a cell phone

Description automatically generated

A screenshot of text

Description automatically generated



* Phase\_3
  + Using objdump again we take a look at the code for phase\_3. We see from the comparisons being done that our input is going to start with a number that is greater than 2, otherwise the bomb explodes. After running disas to see next steps, I check out the input memory address, and I find a string that I recognized from phase1 “%d %c %d”. This tells me the input will take the form of a int, character, int separated by whitespace. I go to the next test, and see that the next value scanned neexs to be smaller than 7 bytes, so our char will just be one letter. Moving to the next conditional, we see the value at 0x14(%rsp) need to be equal to the decimal equivlalent of 0x31, which is 795. Because we already have one int, and the other value is a char, we now know the last int is equal to 795. Immediately after this check, there is another conditional with a value at a specific memory address. After checking what is at this memory address I get the hexadecimal value of 0x64, which is the char “d”. Assuming this to be our char, I use this, and we finish the phase with the input “3 d 795”.

A screenshot of a cell phone

Description automatically generated



A screenshot of a cell phone

Description automatically generated



A picture containing drawing, flower

Description automatically generatedA close up of a black background

Description automatically generated

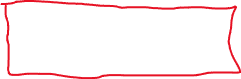


A screenshot of a cell phone

Description automatically generated



* Phase\_4
  + First, I enter a test string into the program. And run <disas> command. I see we are reading in our input and I see this peculiar memory address, so I check it and see “%d %d”. This tells me the input should be 2 integers. I then see a comparison between $eax and 2, which tells me again we should have 2 integers. After moving some stuff around, there is a comparison between the second integer – 1 and two. This tells me the second integer must be at most 3. Then I see two things moved into func4. Both the second integer and the hardcoded integer 6 will be passed to func4. Then there is a comparison between the first integer and the return value from func4, which after inspecting, is 60. This tells me that when integer 2 = 3, integer 1 must = 60. From here I recognize that this pair will defuse the phase and send the string “60 3”.



A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Here we see values of the return from func4 when the second integer is 3. I also confirm the second integer is what is being passed to func4, and that we are comparing to the first integer.

* Phase\_5
  + So, we start by looking at the disas code. The first thing that jumps out to me in the comparison of $0x6 and string length. This tells me that our input will first of all be a string and also needs to be of length 6. I see then that we have a for loop running for the length of 6 from line <+65> and <+69>. After finishing the for loop I see the old function to compare strings. I take a look at the strings in $rsp and the other loaded memory address highlighted in yellow. I see two strings. One random string from my test input and one string I need to end up with. From here its clear the for loop is altering my inputted string. From what I can see there’s an and operation being used on my input for each letter with 0xf. From here I simply mapped out the alphabet to find the correct starting string which is “ioqpew”. This returns “flames” once put through the for loop.

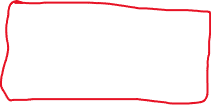
A screenshot of a cell phone

Description automatically generated



* Phase\_6
  + We open up the disas and see our function <read\_six\_numbers>, so we know the input will be 6 integers separated by whitespace. Going through the stepi I then see a loop appear that goes through each int and makes sure they are unique. We also see that each integer must be less than or equal 6 (highlighted), which is checked within the for loop. So, after we have these constraints, the only thing left to figure out is the order of the numbers. After we verify those two conditions, I noticed a peculiar memory address being loaded in, so I took a look (it is highlighted), and it seems to be a node. Because this is also in a loop, we end up with a linked list. After this, I noticed the final condition before the integers can be verified as a solution. They have to be in order from largest to smallest as seen by the comparison using jge on line <+249>. If the numbers are not in the correct order, then the bomb explodes. So, what I did was use the <x $rdx> command to see the value of node1. I then appended bytes to node1 in order to see node2-node6. Once I had the values, I was able to write them down and then rearrange them from largest node value to smallest node value to get the solution “4 1 2 3 6 5”. Bomb defused!

Whooo!



A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

