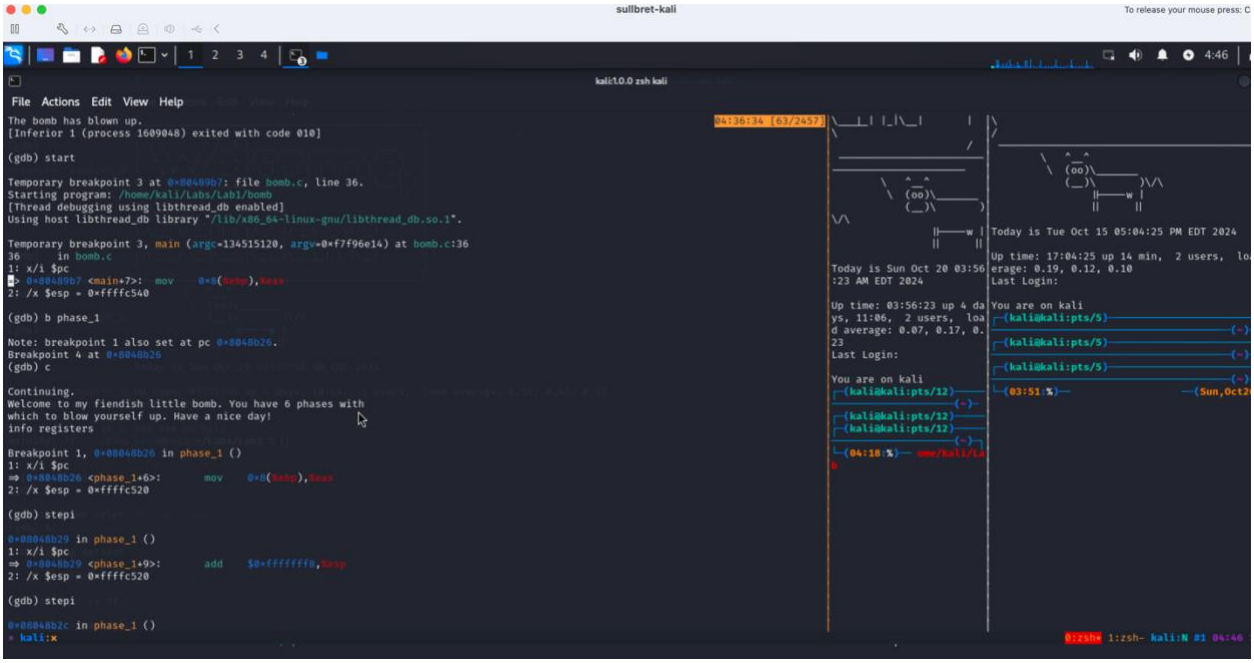
Brett Sullivan

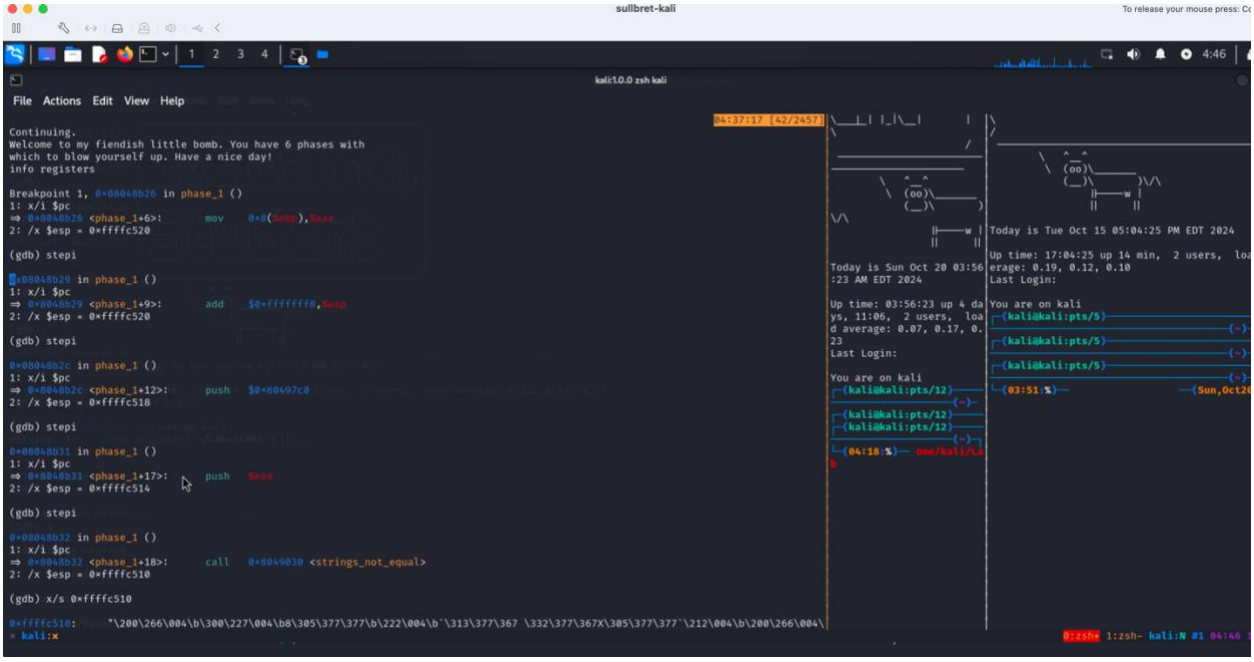
10-20-24

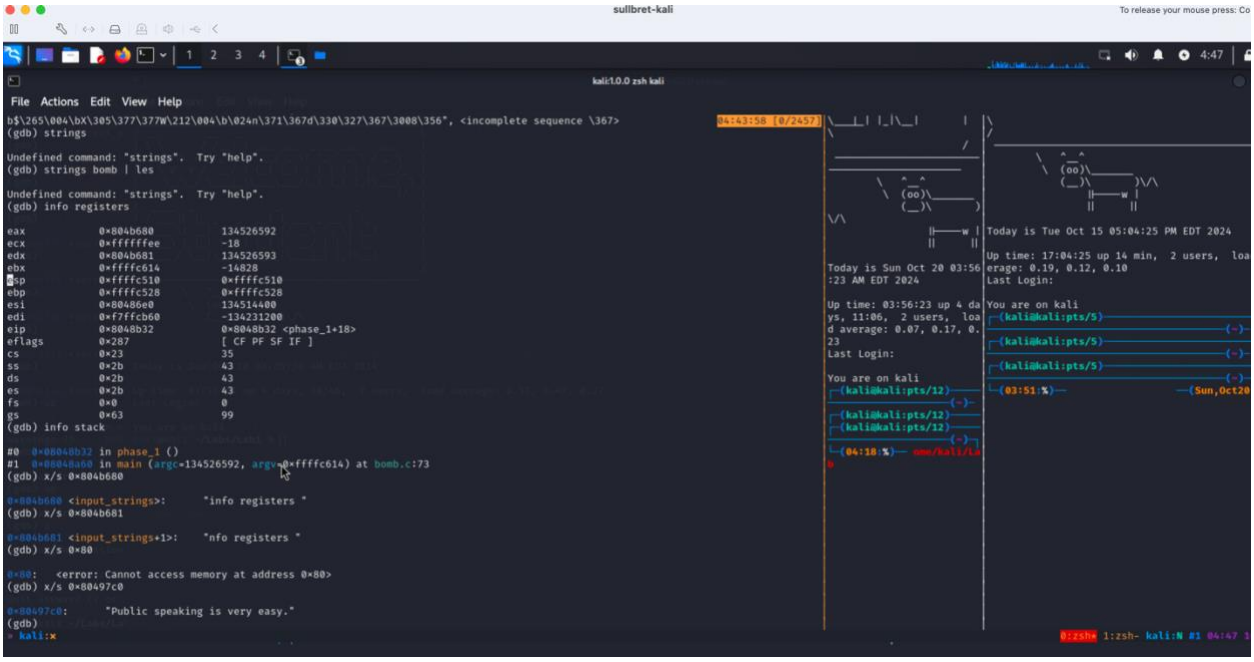
Lab 1 GDB Bomb!

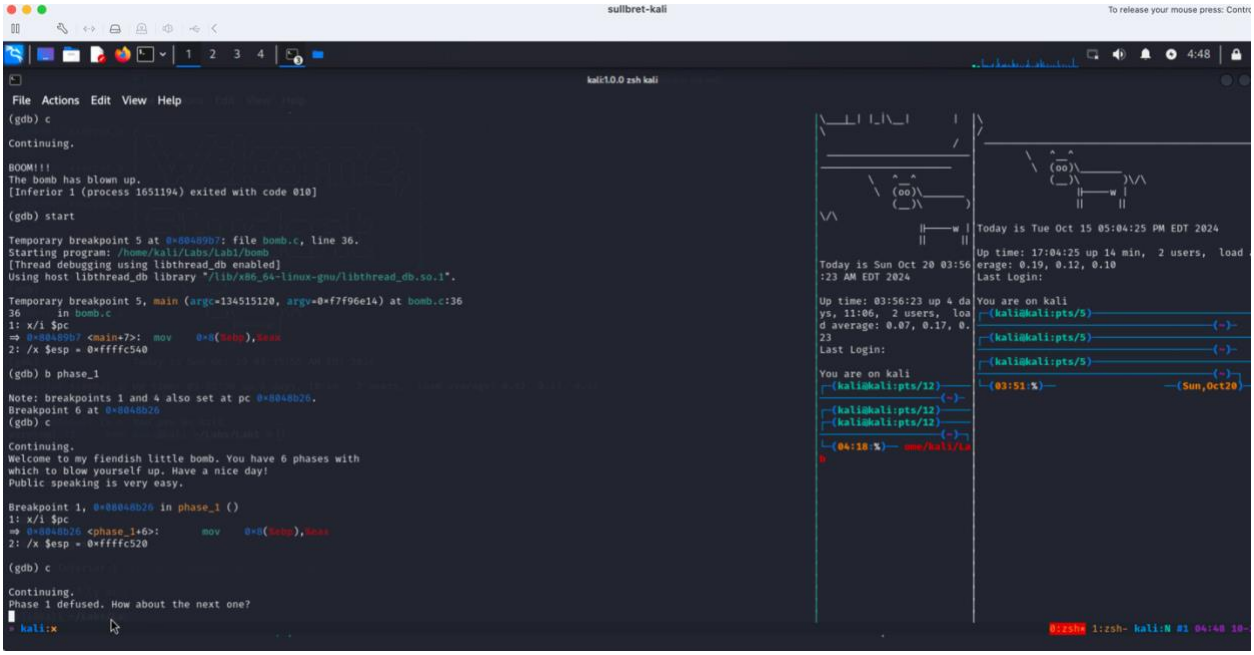
Phase 1

I was first looking in the registers and thinking the string had already been passed into an assembly there, but the string to pass the first phase," Public speaking is very easy", was in fact just being passed/pushed raw onto the stack right before the string comparison function call, which I think is a bit different than how it was shown in the video. Using "disassemble" helped a bit in seeing how the function worked too. For now, I will just depend on analyzing the stack, registers and instructions separately and see how far that gets me. I will paste the screenshots of the steps I took below for this.



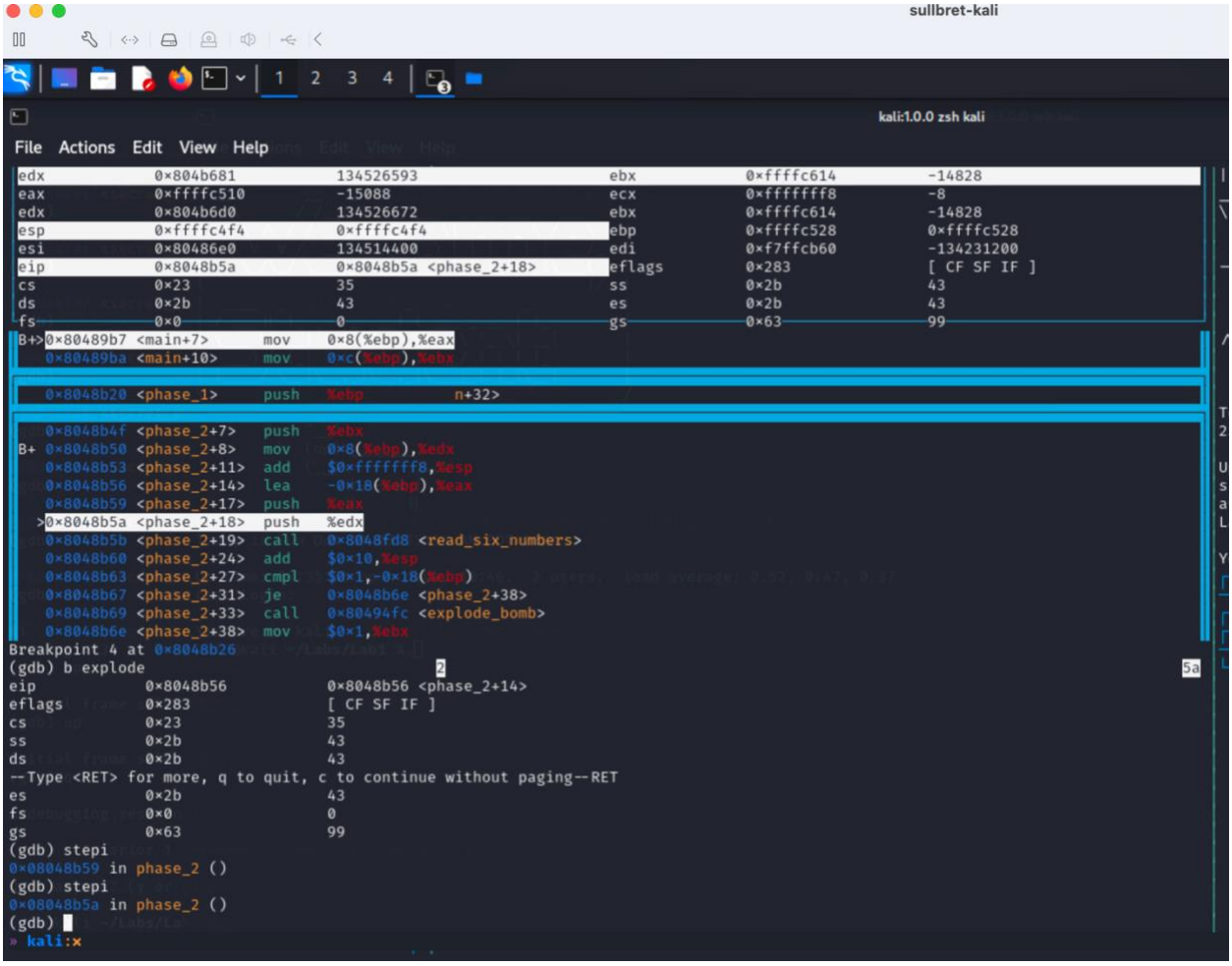

sullbret-kali
kali:1.0.0 zsh kali
File Actions Edit View Help
Continuing.
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
info registers
Breakpoint 1, 0x08048b26 in phase_1 ()
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) stepi
(gdb) x/s 0xffffc510
(gdb) stepi
(gdb) stepi
Today is Sun Oct 20 03:56:23 AM EDT 2024
Up time: 03:56:23 up 4 days, 11:06, 2 users, load average: 0.07, 0.17, 0.23
Last Login:
You are on kali
Today is Tue Oct 15 05:04:25 PM EDT 2024
Up time: 17:04:25 up 14 min, 2 users, load average: 0.19, 0.12, 0.10
Last Login:
You are on kali


sullbret-kali
kali:1.0.0 zsh kali
File Actions Edit View Help
(gdb) strings
Undefined command: "strings". Try "help".
(gdb) strings bomb | les
Undefined command: "strings". Try "help".
(gdb) info registers
(gdb) info stack
(gdb) x/s 0x804b680
(gdb) x/s 0x804b681
(gdb) x/s 0x80
(gdb) x/s 0x80497c0
(gdb)
Today is Sun Oct 20 03:56:23 AM EDT 2024
Up time: 03:56:23 up 4 days, 11:06, 2 users, load average: 0.07, 0.17, 0.23
Last Login:
You are on kali
Today is Tue Oct 15 05:04:25 PM EDT 2024
Up time: 17:04:25 up 14 min, 2 users, load average: 0.19, 0.12, 0.10
Last Login:
You are on kali

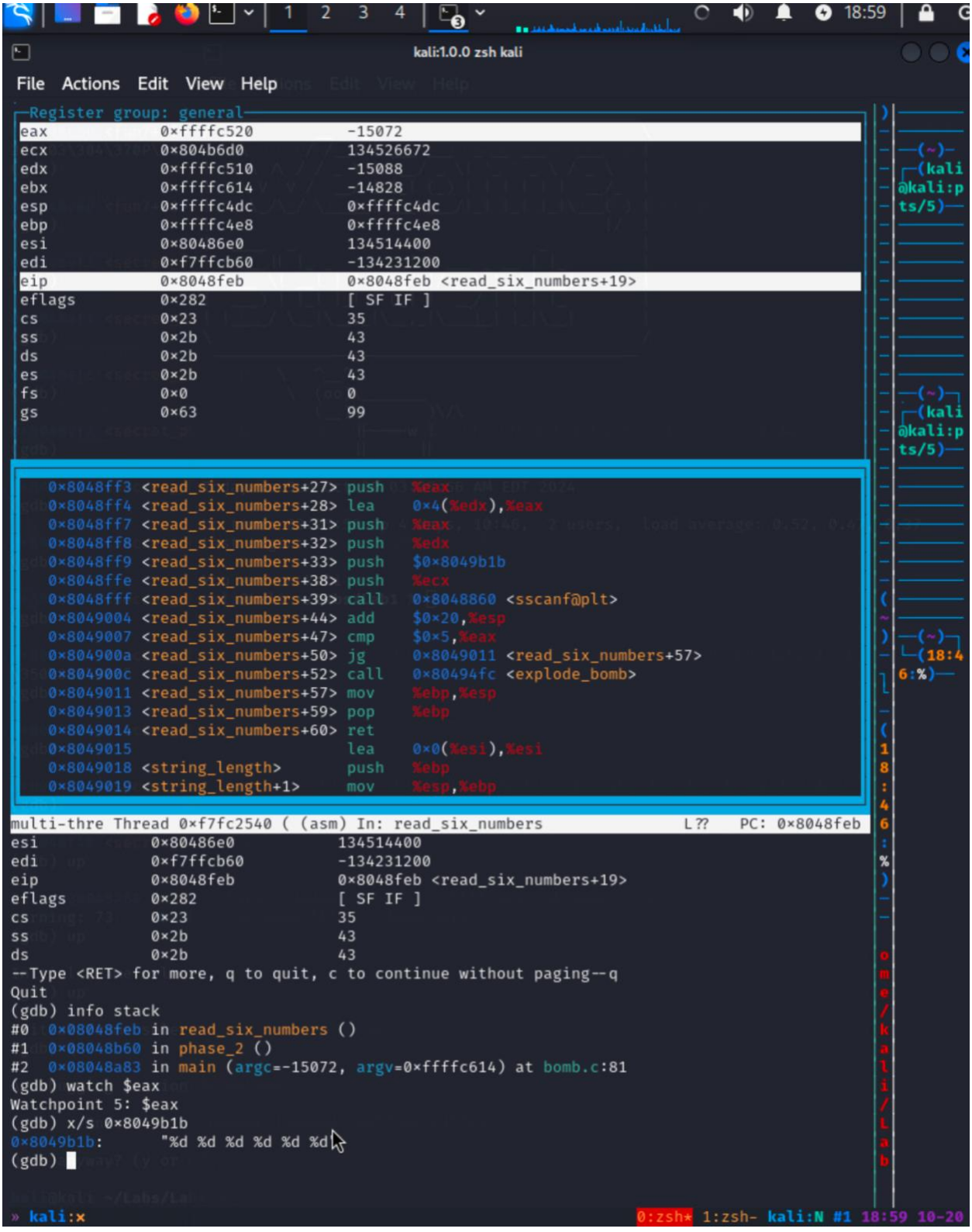


Phase 2:

This is stepping through phase two, right before the ‘read_six_numbers’ is being called.I am now using tui to be able to view the registers and instructions as I step through each command, which helps a little so I don’t have to keep doing individual calls for them each time.

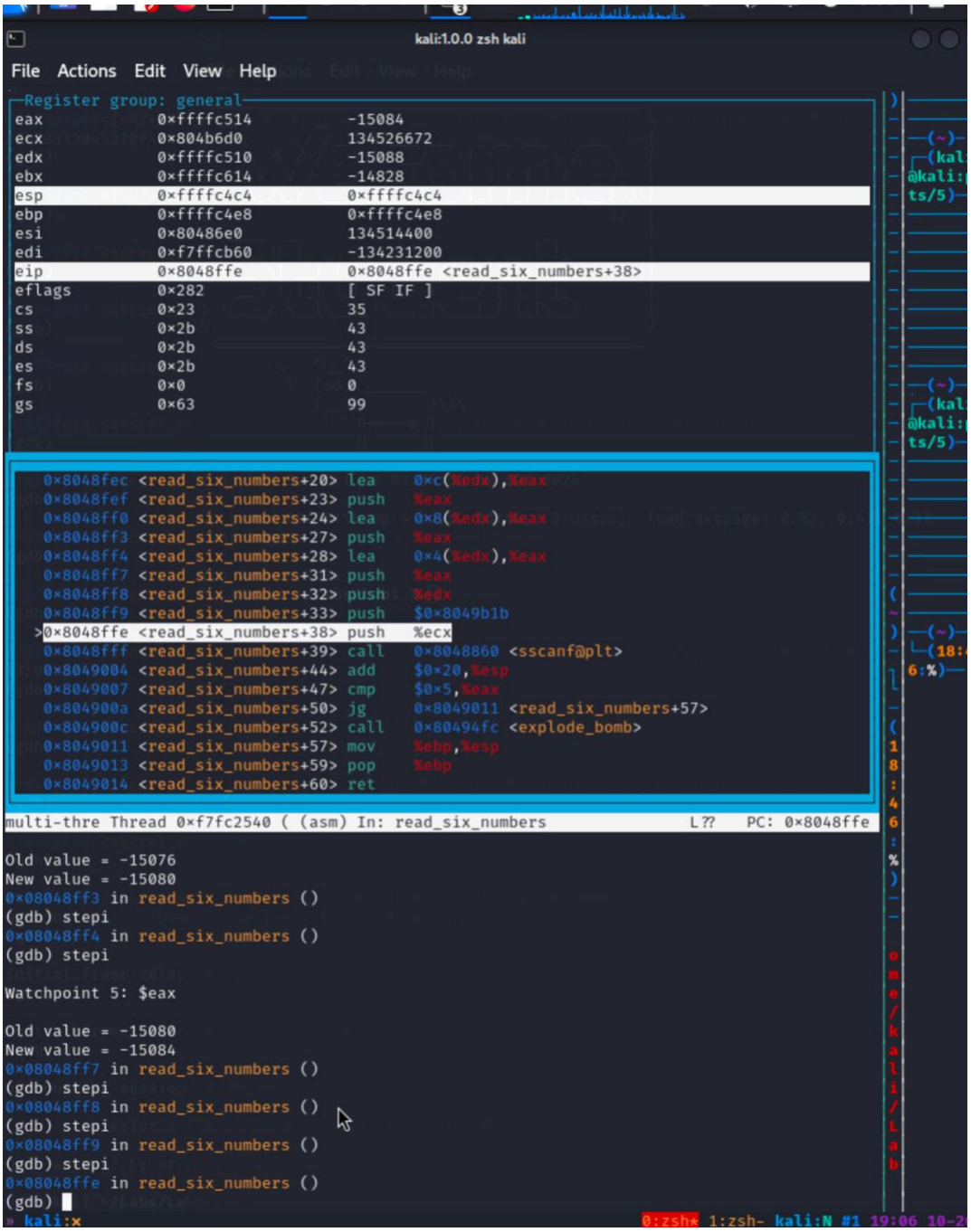


I can see what kind of input is expected by examining the function call string, after doing this I can see that it is expecting 6 integers, each separated by a space. Shown below:

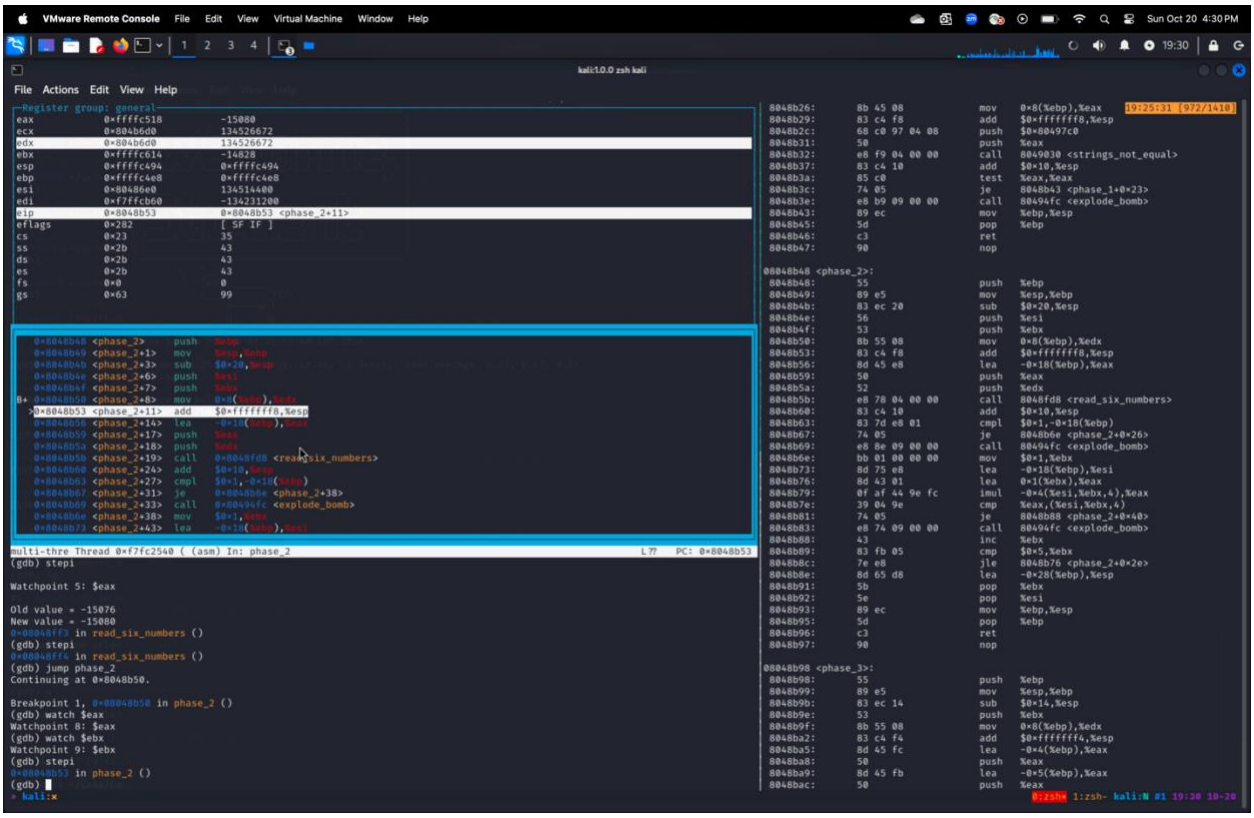


Since it appears the values are likely being moved into eax during the loop inside the function, I set a watch point for both eax and ebx (since it was hinted these are where the

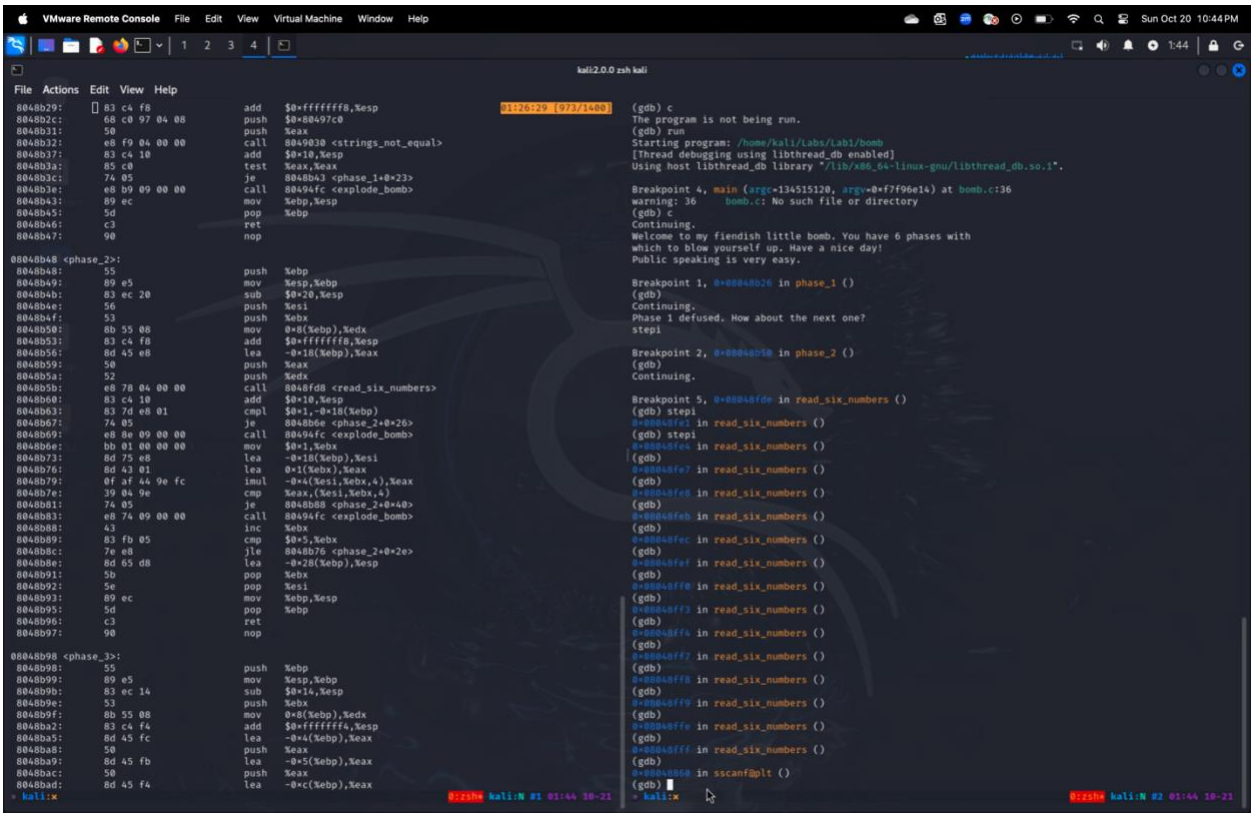
iterated comparison values will be stored.) so I am being informed every time the value in it changes, shown below.



I also did an object jump on the file to have a view of both phase_2 and the read_six_numbers to check and understand the patterns of calls, shown below:

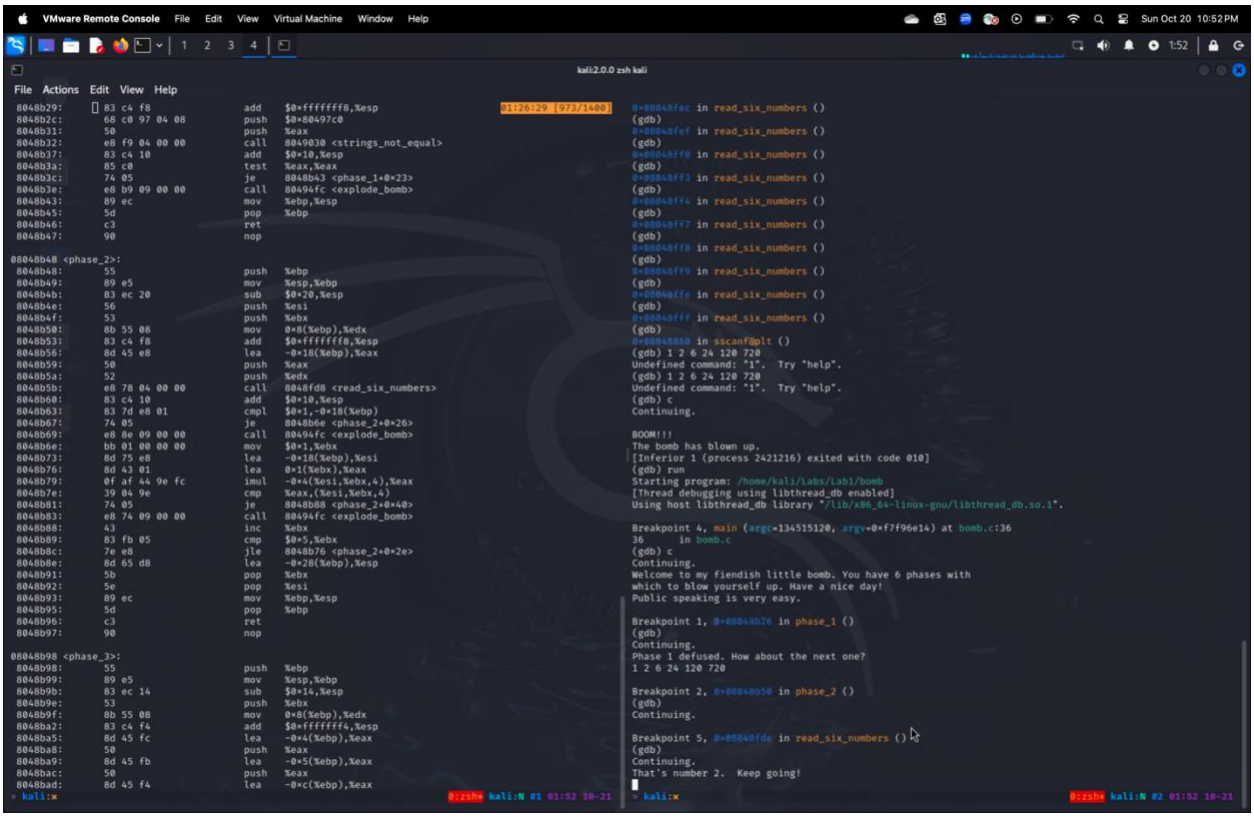


After further analysis, mainly from getting a better understanding of the assembly mainly in phase_2 from the object dump (I spent too long trying to get it from the function itself first), I was able to see where the loop is starting and how it is being incremented each time.



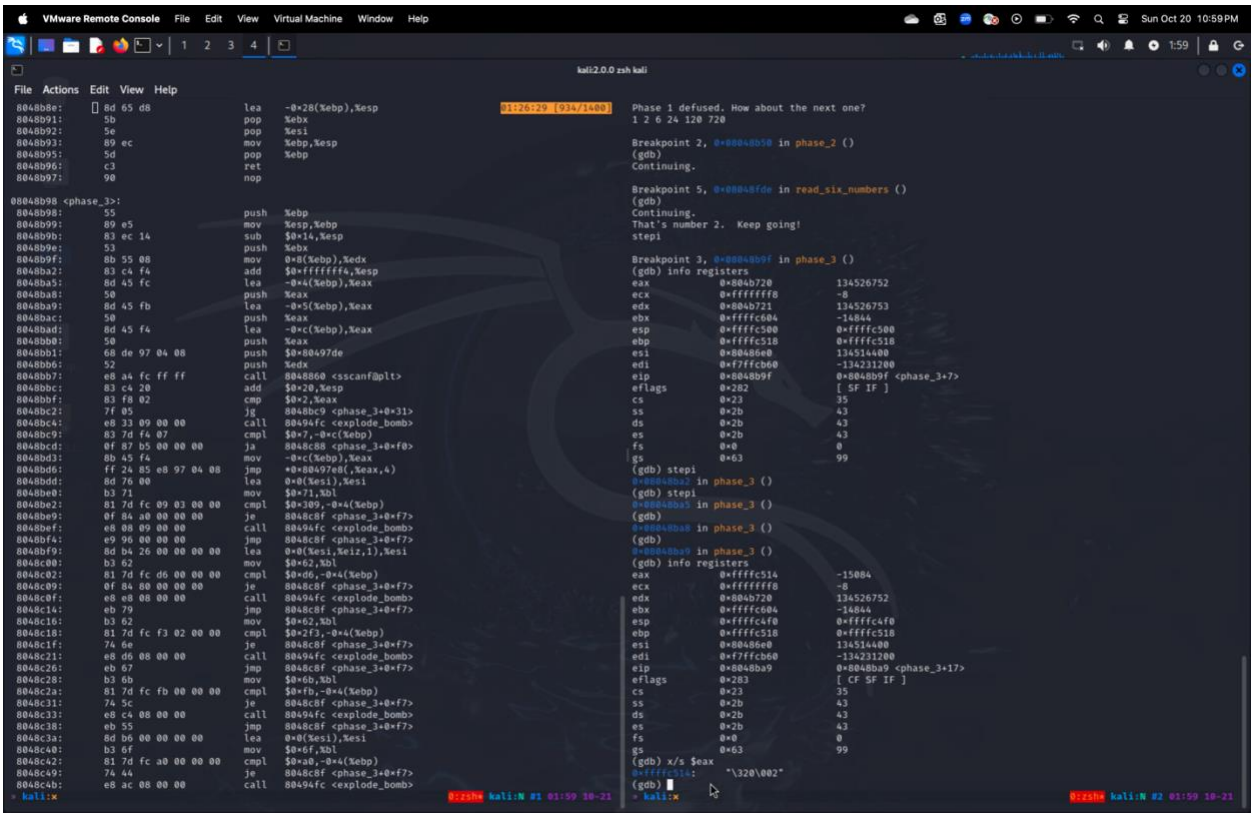
At the address The loop starts by incrementing the loop counter with lea 0x1(%ebx), %eax, storing the result in the %eax register. Then, using imul -0x4(%esi,%ebx,4), %eax, it multiplies this value by a value from memory, which suggests the program is calculating the next number in a sequence based on previous numbers. The next instruction, cmp %eax, (%esi,%ebx,4), compares the result against a value in memory. If the two values match, the program jumps forward with je 8048b88 <phase_2+0x40>, allowing the loop to continue; otherwise, the bomb explodes.

By reverse engineering this loop, I discovered that it implements a factorial-like pattern: v[0] = 1 and v[i] = (i + i) * v[i - 1] for each subsequent value. Using this insight, I determined that the six integers required for phase 2 are “1 2 6 24 120 720”. Entering these values satisfied the conditions of the loop, successfully diffusing the bomb for this phase. The screenshot below shows phase 2 being diffused.

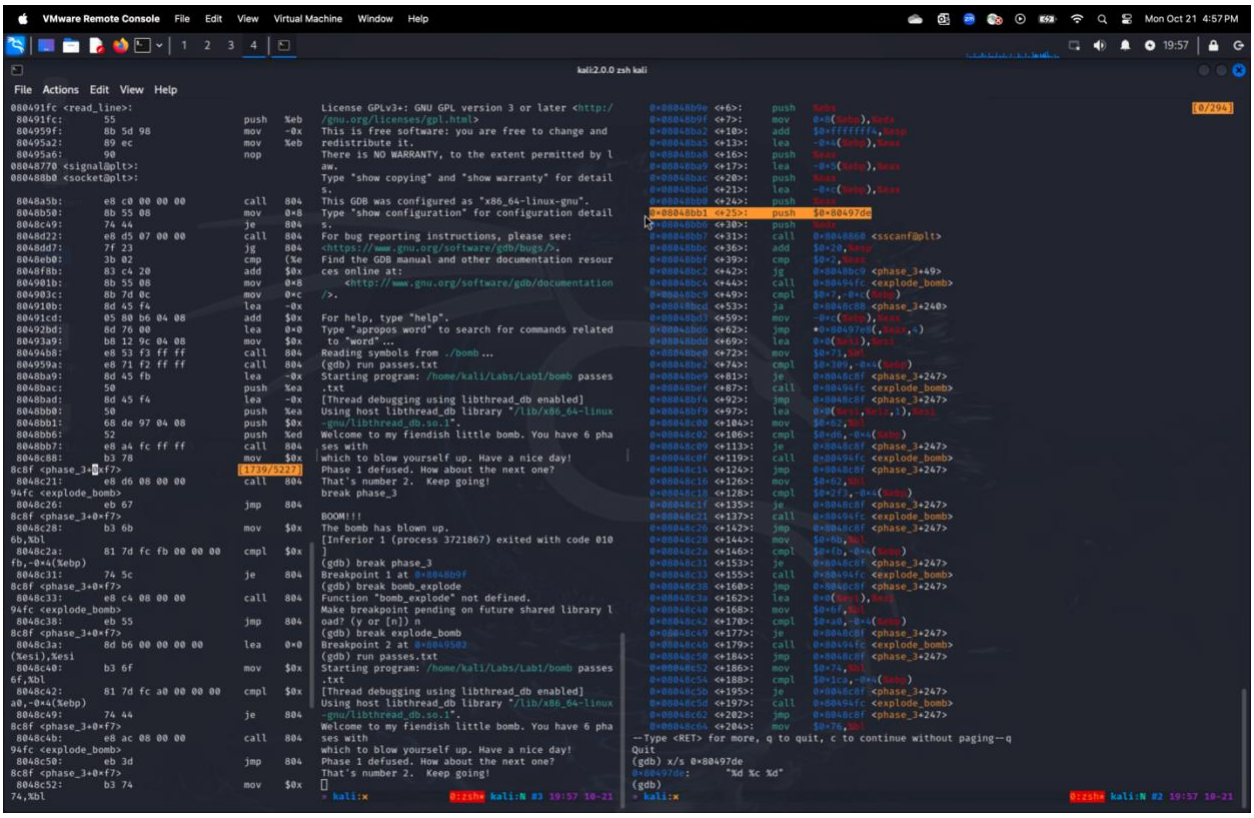


Phase 3:

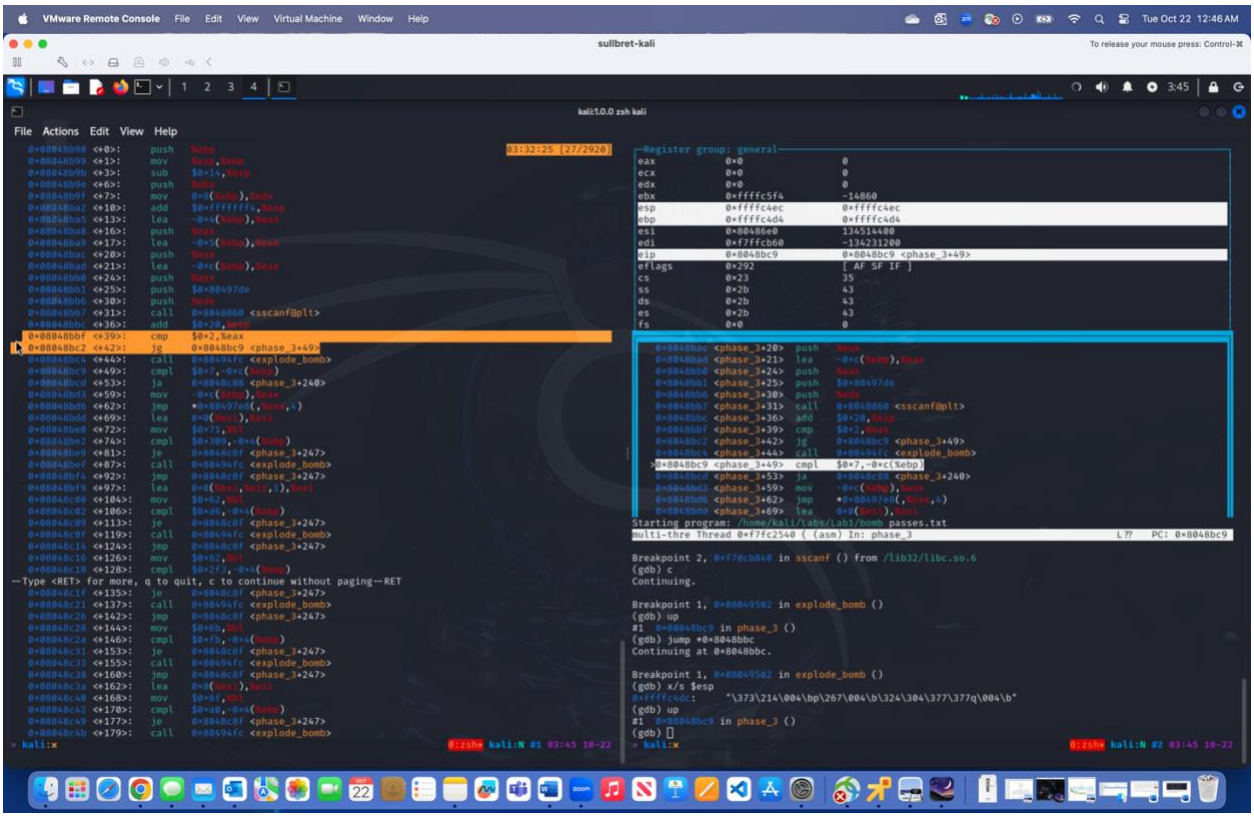
There is quite a lot going on in phase 3 after looking at the object dump shown below:



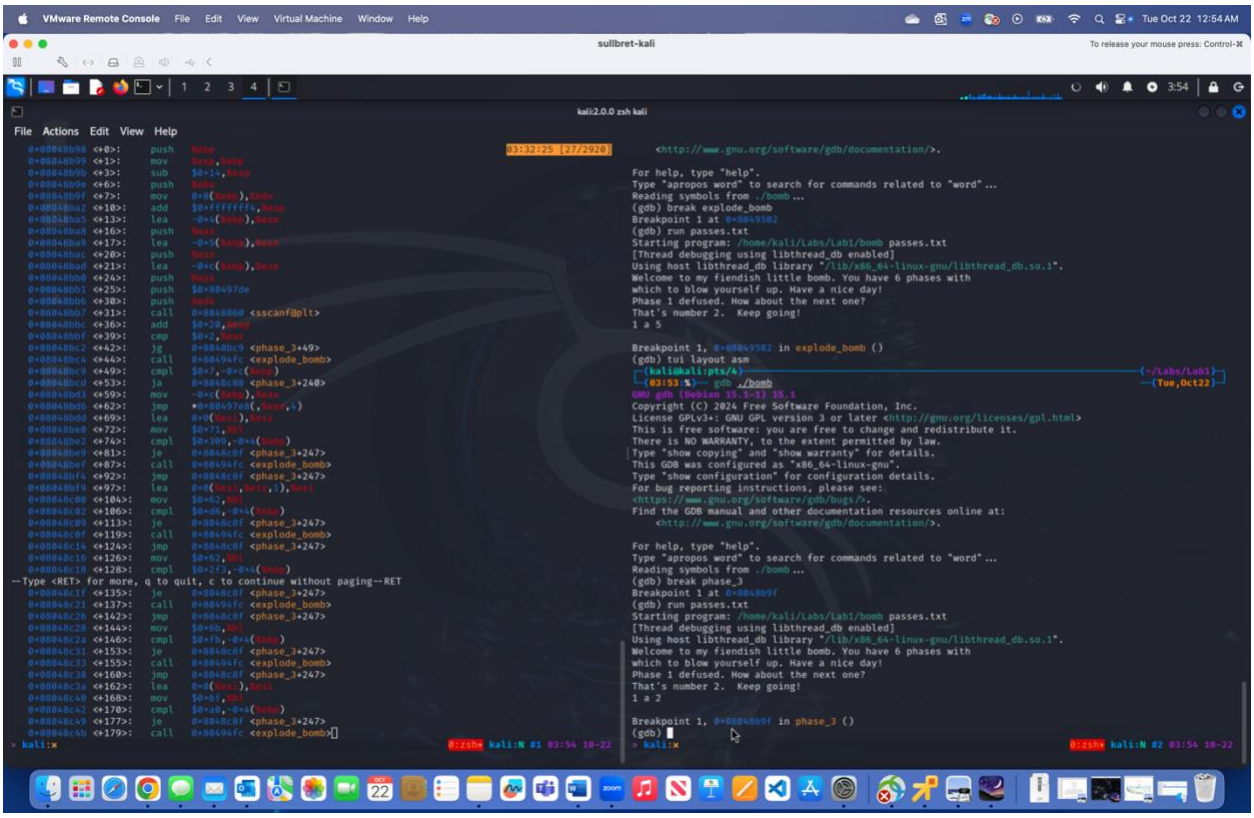
The first thing that stands out (and was hinted at in EdDiscussions) is the “sscanf” happening at 8048bb7. The couple lines prior to this are pushing some values onto the stack beforehand. First, I’ll try to see what input is expected there with an x/s call. By inspecting the highlighted line below after running a disassemble on phase_3, I can see that the format value expected is “%d %c %d”, meaning for phase 3 it will expect an integer then a char then another integer.



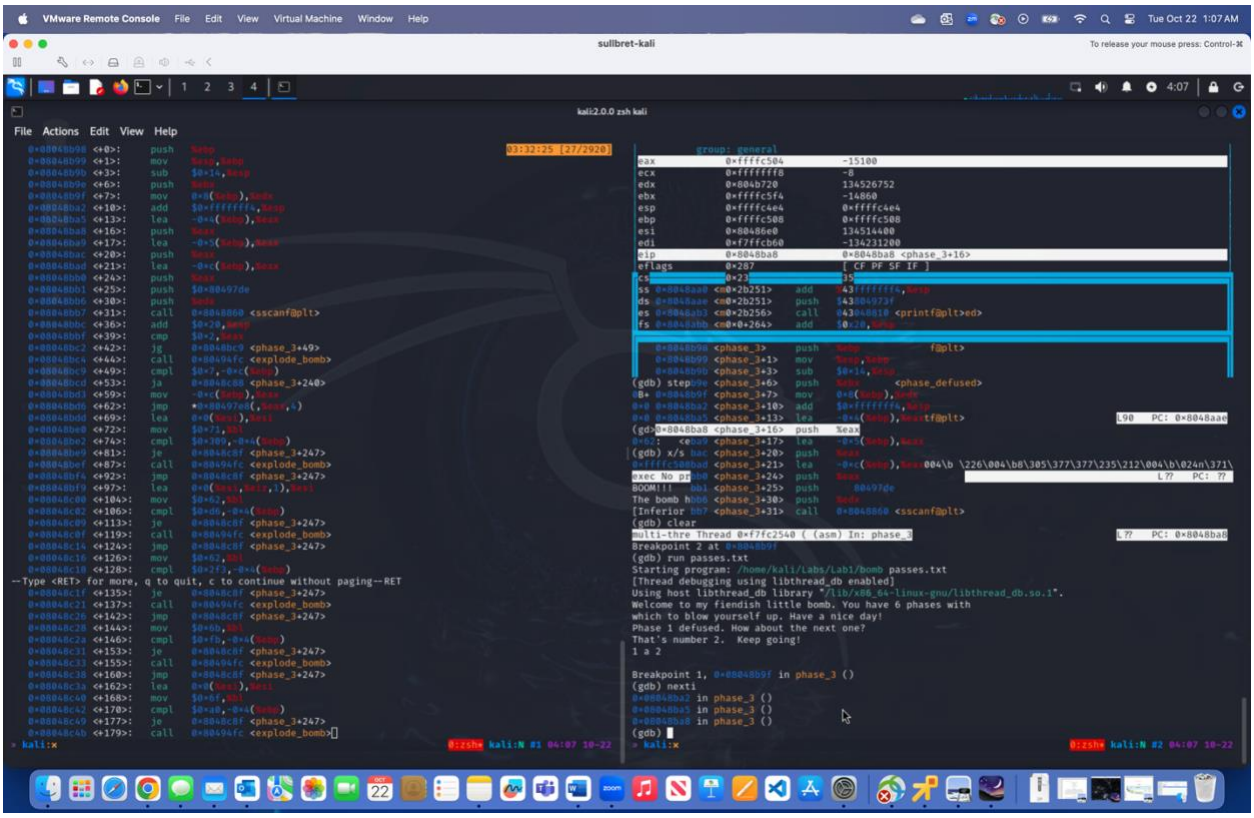
We can now try to watch which compares and jumps are happening after the first ‘sscanf’ and avoid the ones that lead to bomb explosions. We can see the first comparison after ‘sscanf’ is with eax and 0x2, and if the jump is not taken the bomb explodes.



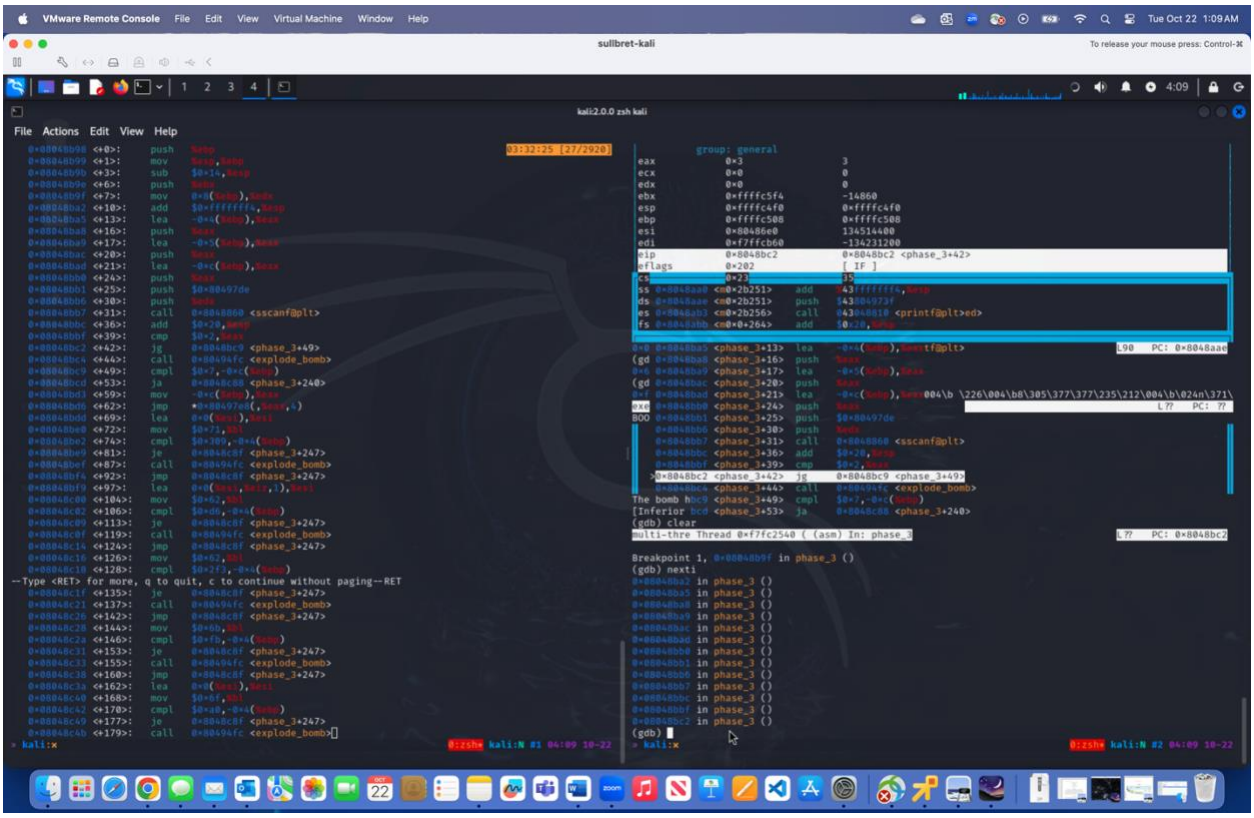
We will not continue stepping through to see where the jumps are happening when we try the phrase ‘1 a 2’ shown below



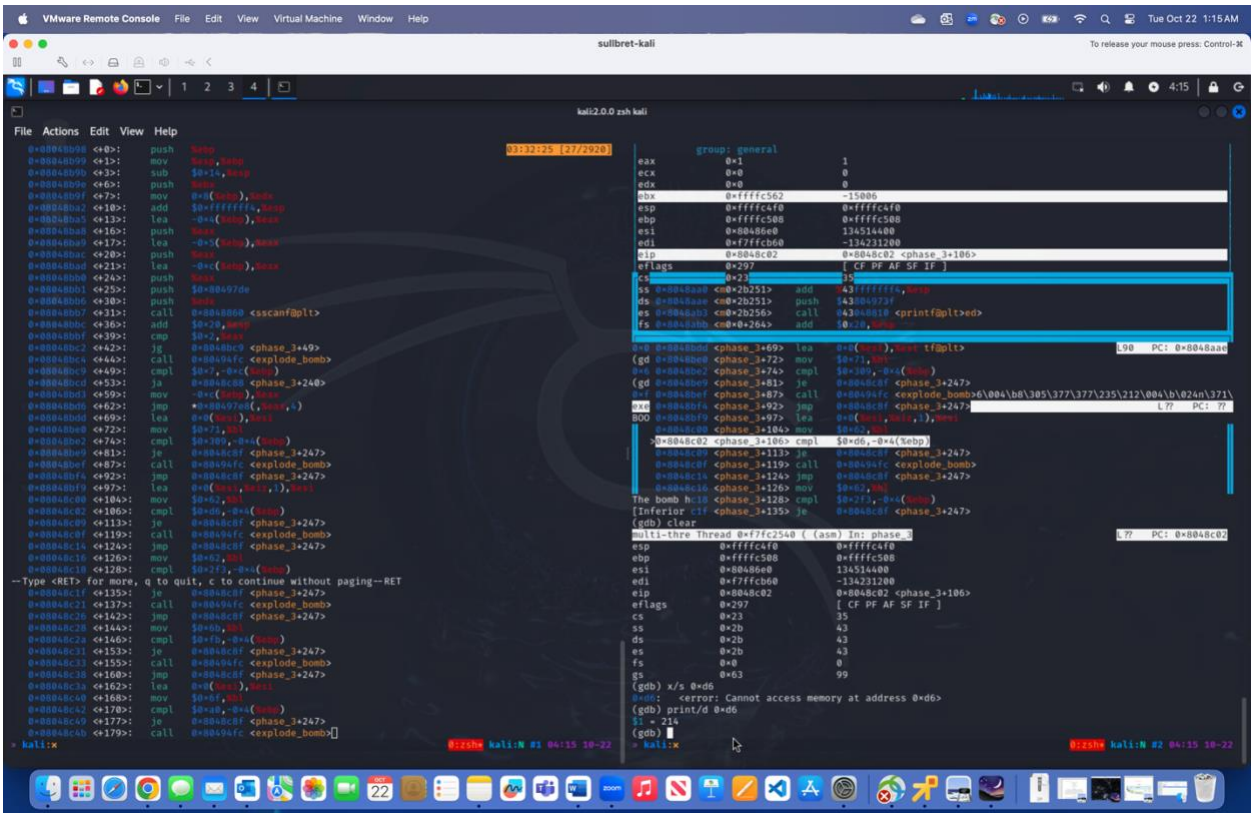
We can see this finally passes the first check which occurs in sscanf, and allows us to continue on, as the bomb hasn't been triggered yet.



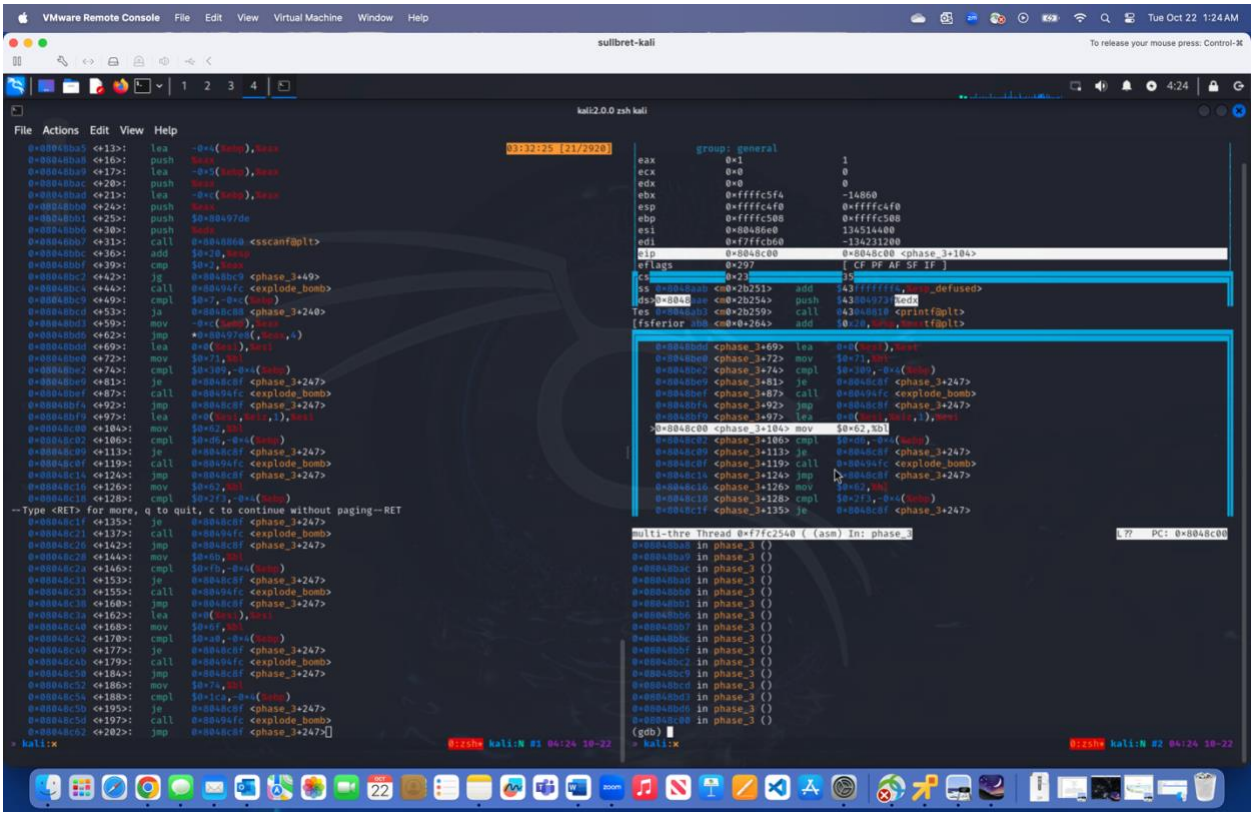
Here it is comparing eax to 2, our stored value in eax is 3, so it does a jump to avoid a bomb explosion.



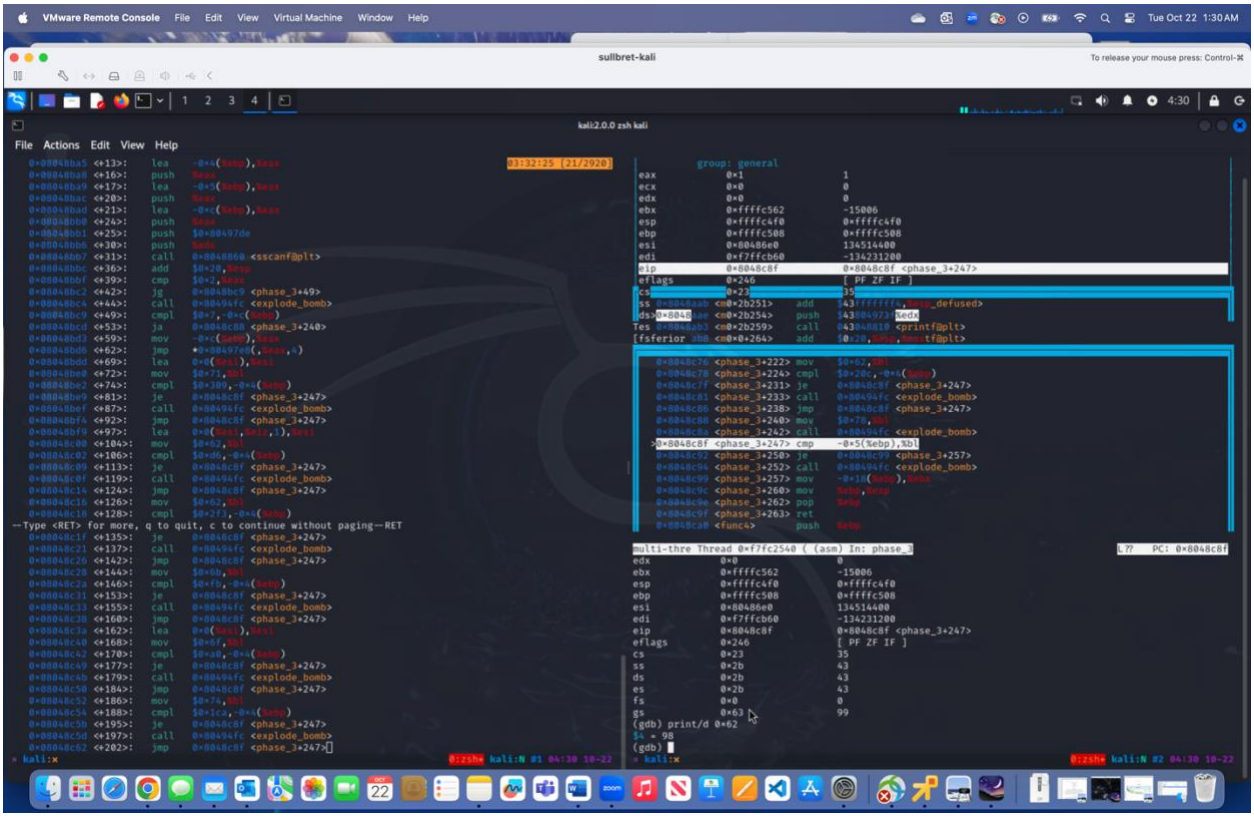
The next thing we notice is a comparison to 0xd6, which when we run a ‘x/s’ on, is the integer 214, which we can assume must be our second integer, as the char is being skipped over for now.

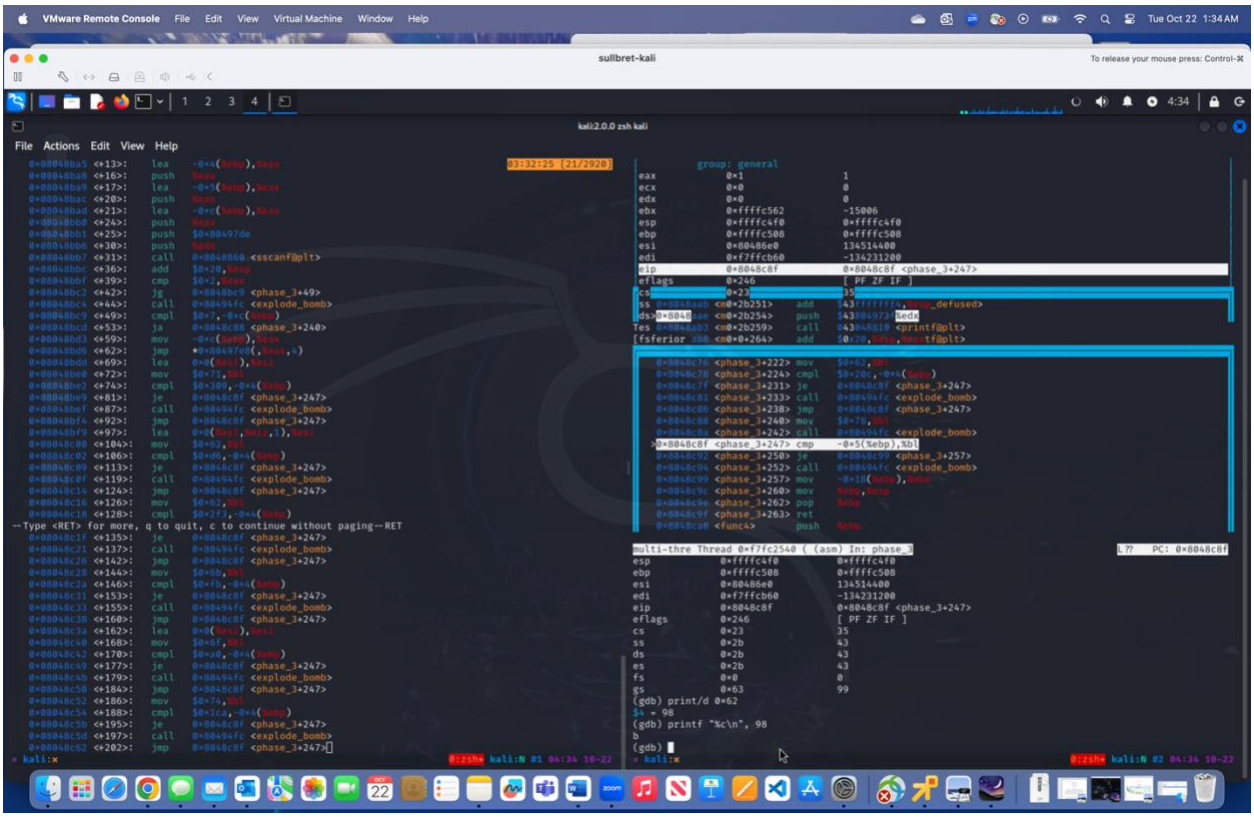


Now that we have both our integers, we just look for the char value as we step through.

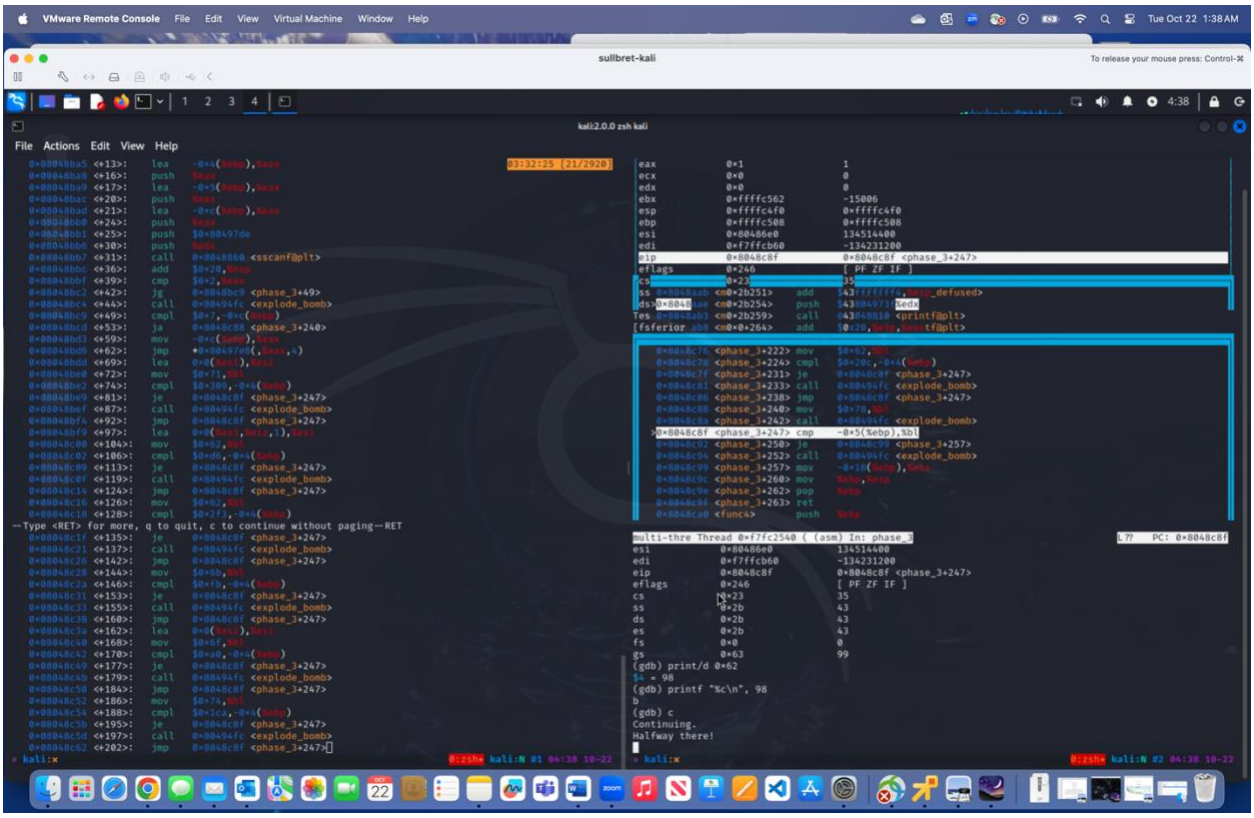


As we keep stepping through, we can see our char value is being moved into ebp and being compared against the $bl value. After the conversion shown in the terminal on the hex value in the $b1 register, we know it must be 98, which I then ran a conversion to ASCii command, which gives us the char ‘b’. Shown below



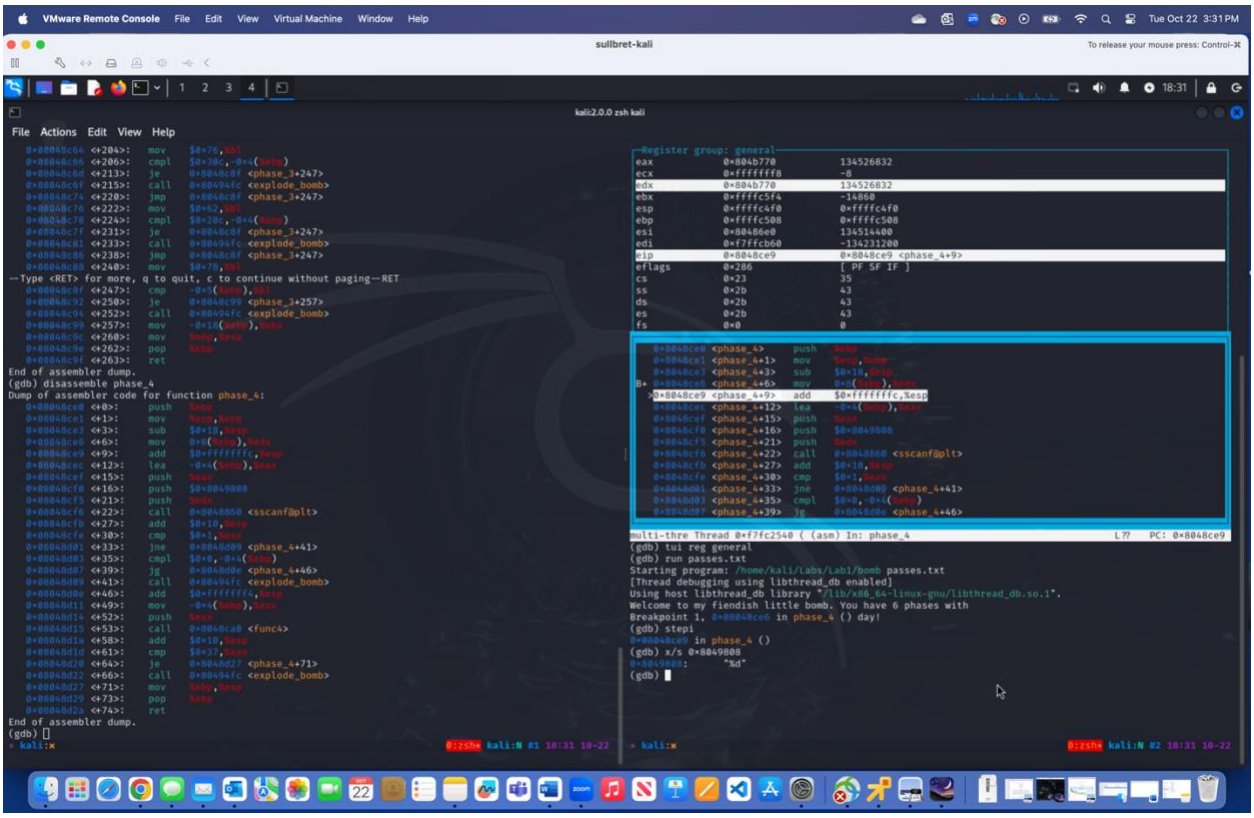


I still wasn't 100% on if 1 was the right first integer (as it seemed to work so far but there was not definitive proof of it), but after plugging in "1 b 214", the phase 3 was passed with no explosion, hooray!

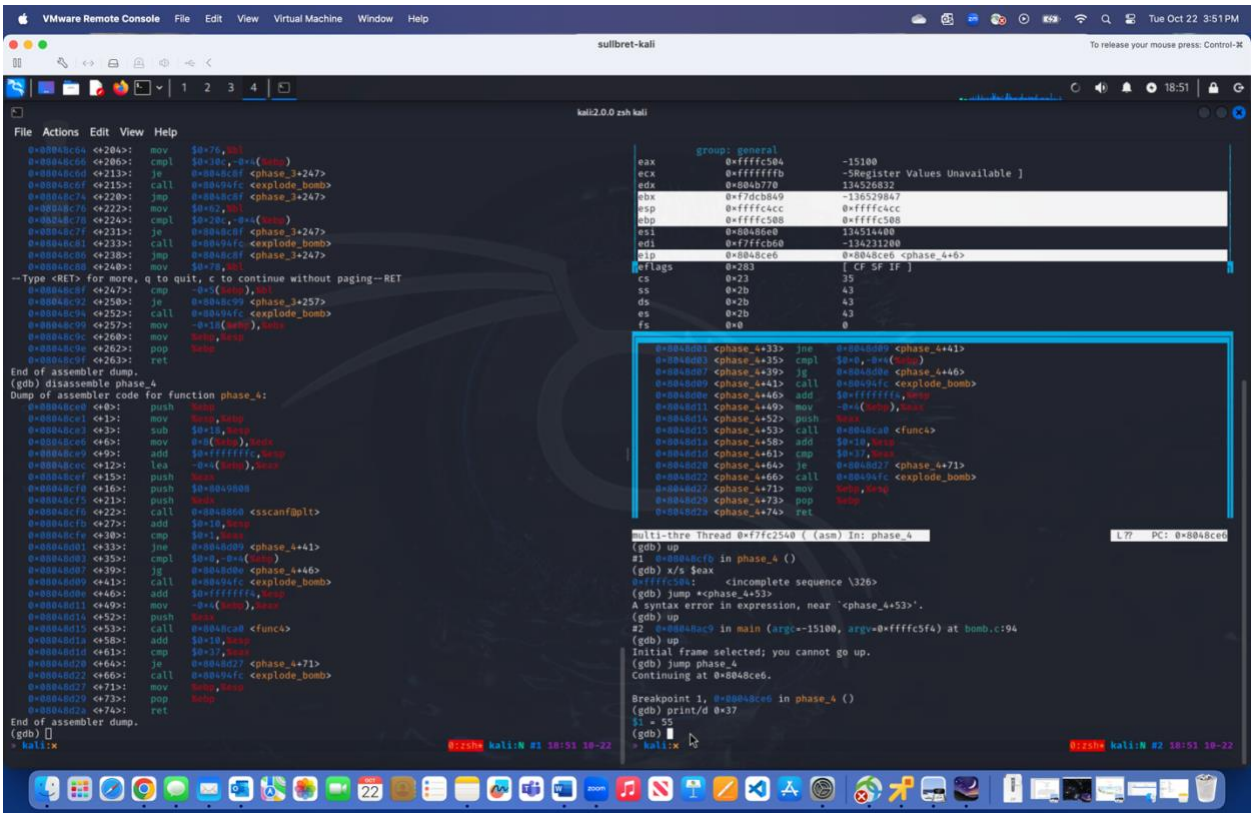


Phase 4:

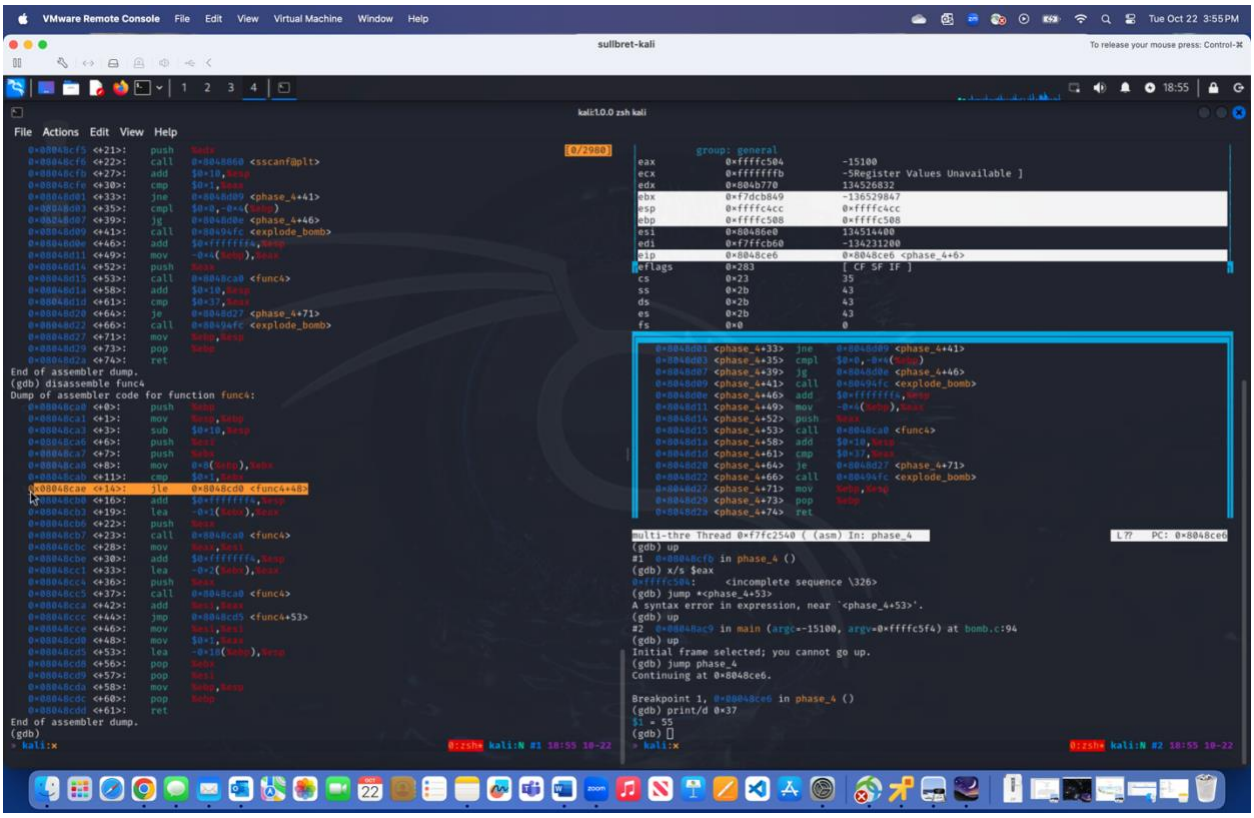
After doing a disassemble on phase_4 and starting to step through instructions, we can see that there is a raw value at ‘<phase_4+16>’ being pushed onto the stack right before the ‘sscanf’ is being called. We run a x/s on this and see our input that is expected is a “%d” value, meaning a single integer is needed to pass this phase.



The next thing noticed is that there is a function being called, “func4” below the sscanf. Here we can see our input is in eax, and is being compared to the hex value 37 (on line <phase4+61>), which after running a check on it is 55 in decimal form. This tells us that our input needs to be 55 after returning from the function call, shown below.

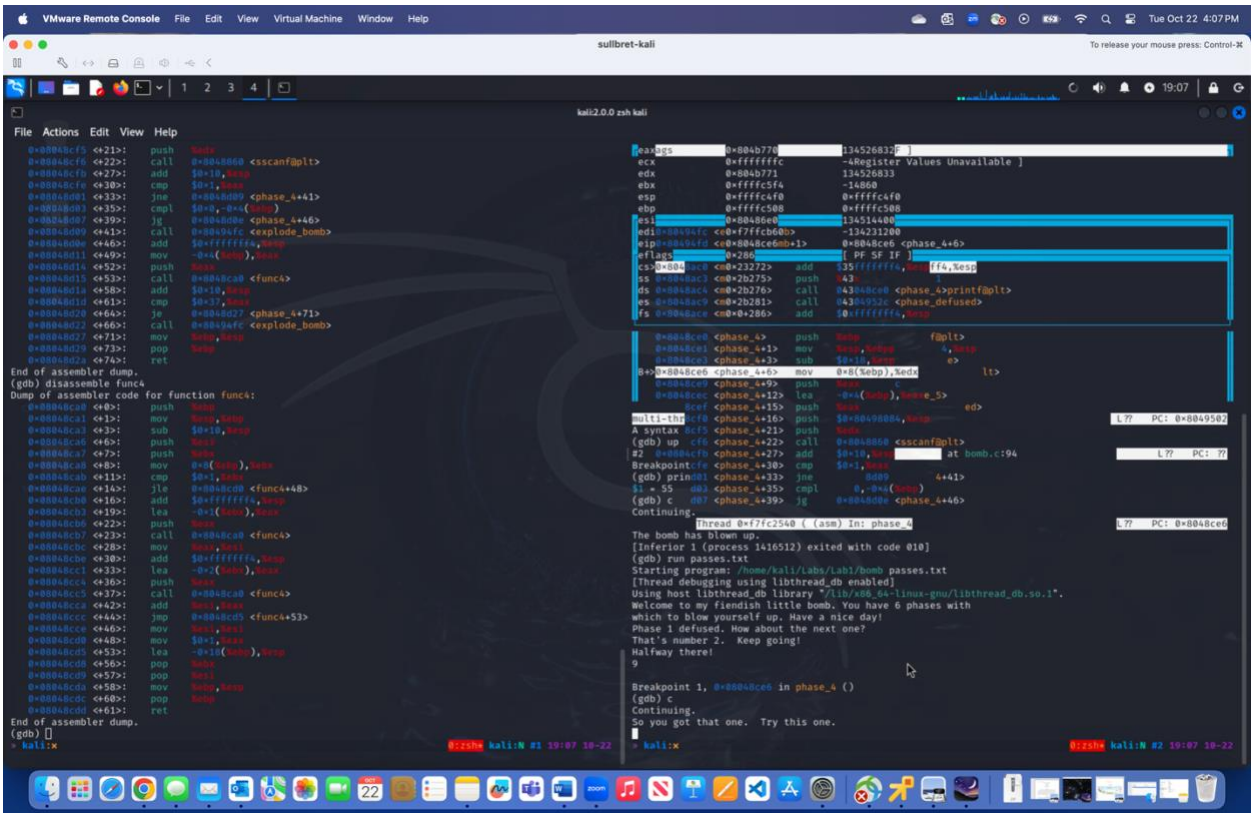


Next we will run a ‘disassemble’ on func4 to see what is taking place with our input and what it needs to be. This is where things become clear. It is checking to see if it is less than or equal to 1, which is the base case for a recursive loop taking place inside the function, so it keeps calling itself and restarting the function until this base case is reached.



There is another recursive call happening, and this is where it becomes clear that an algorithm is in place, which is the Fibonacci sequence. In phase 4, the recursive function <func4> doesn't directly return the Fibonacci number for the input but instead follows a slightly different logic. When the input to func4 is 1 or less, it returns 1. For an input like x = 2, it will return 1 for both x - 1 = 1 and x - 2 = 0.

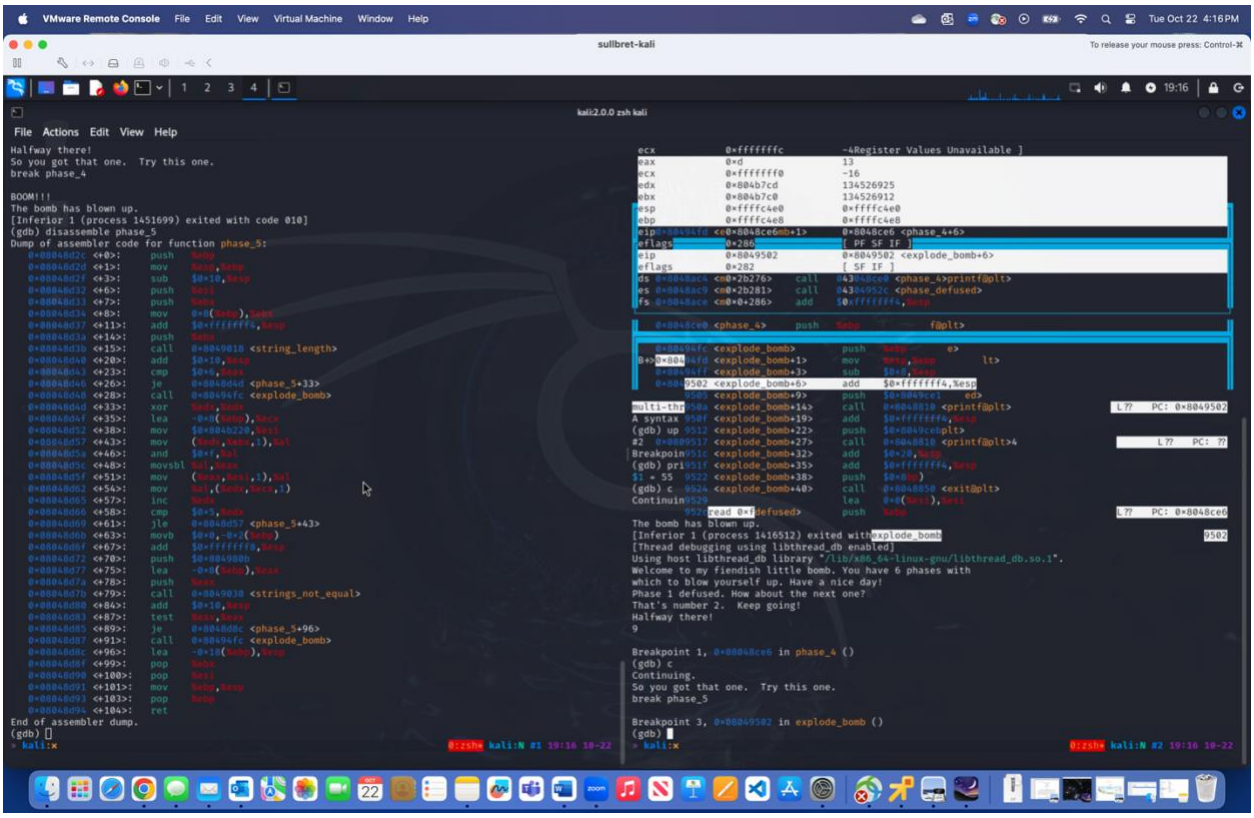
To solve phase 4, the value returned by the function must match hex 37, which equals 55 in decimal. Knowing that func4(0) and func4(1) return 1, and func4(2) returns 2, the input must generate a Fibonacci-like value that sums to 55. By examining the Fibonacci sequence, we see that the Fibonacci number corresponding to 55 is 10. Therefore, subtracting 1 from 10, the input for phase 4 should be 9. We will try this input to see if it is right, shown below.



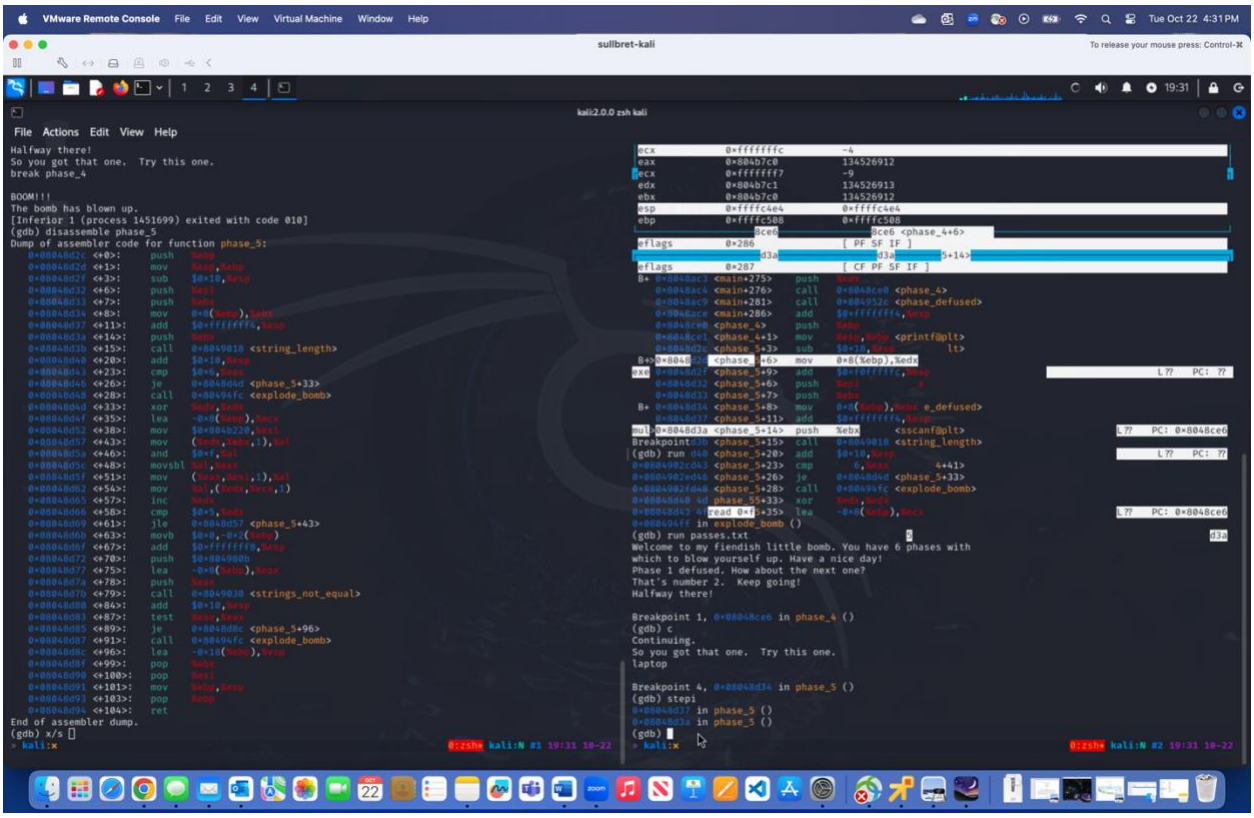
And entering the input integer 9 works! Onto the next phase

Phase 5:

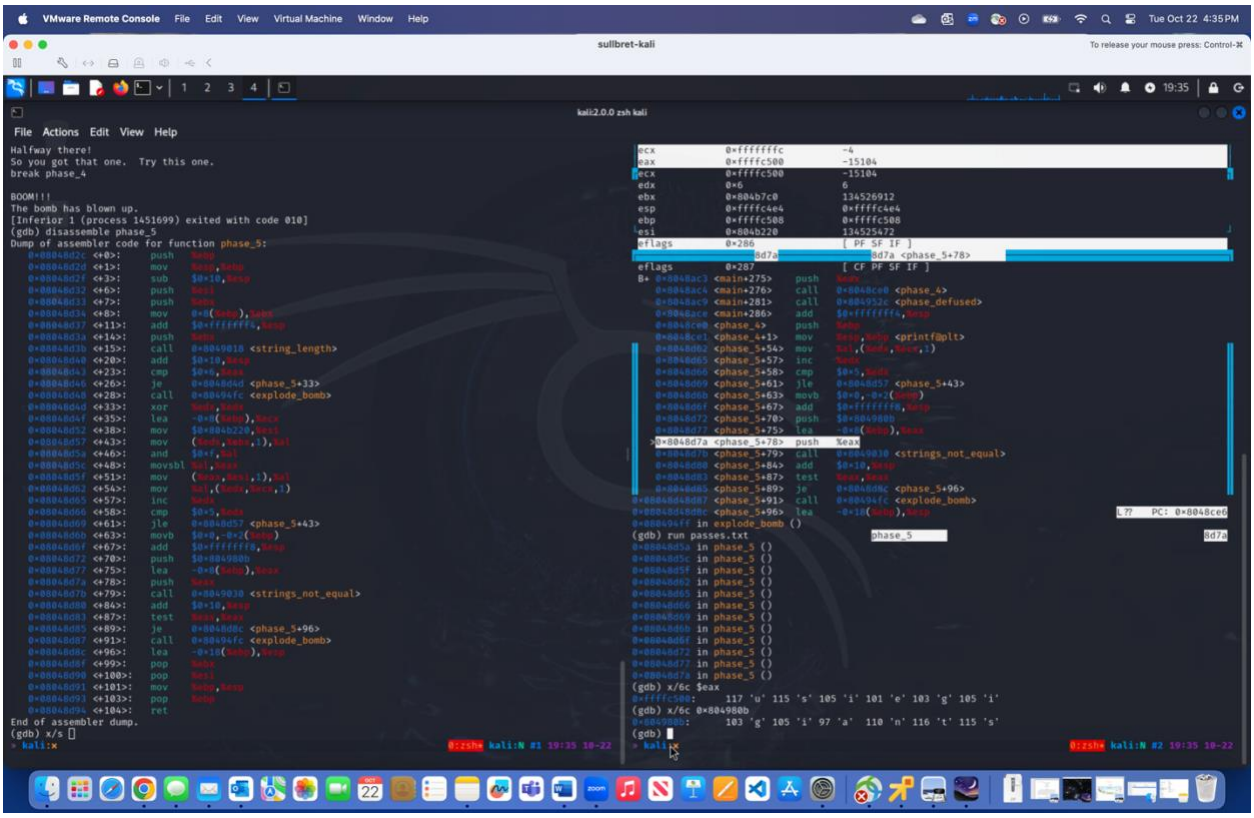
The first things that pop out for phase_5 after a disassemble call are a call to the function “string_length” at “<phase_5+15>” then a comparison of 0x6 and eax, where if they are even, an explode call is jumped on the next line, which is obviously needed to continue.



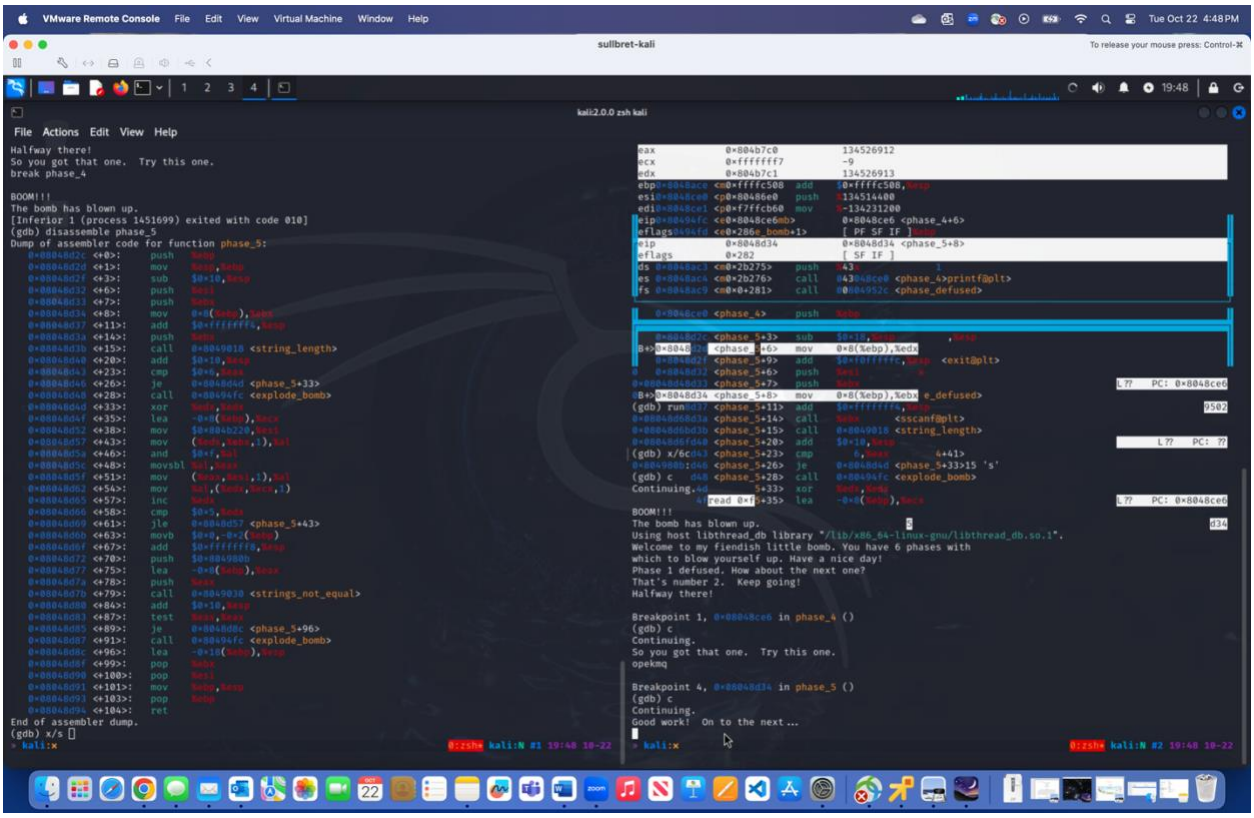
This tells us the string length after that function call must return an equivalent to 6 to continue without blowing up. The next thing that stands out in an “inc” call happening at line <+57> which is incrementing %edx, then comparing to 0x5, then jumping if less than or equal back to line <+43>. We can see this happening 6 times, which would make sense for each character in our string input being iterated through. Now we will run a test string of length 6, “laptop” and step through phase_5.



We were able to step through most of phase_5 with this string, and can see what our string stored in eax register is turned into at the end, compared with what is expected that it is compared against that is pushed on line <phase_5+70>, shown below:



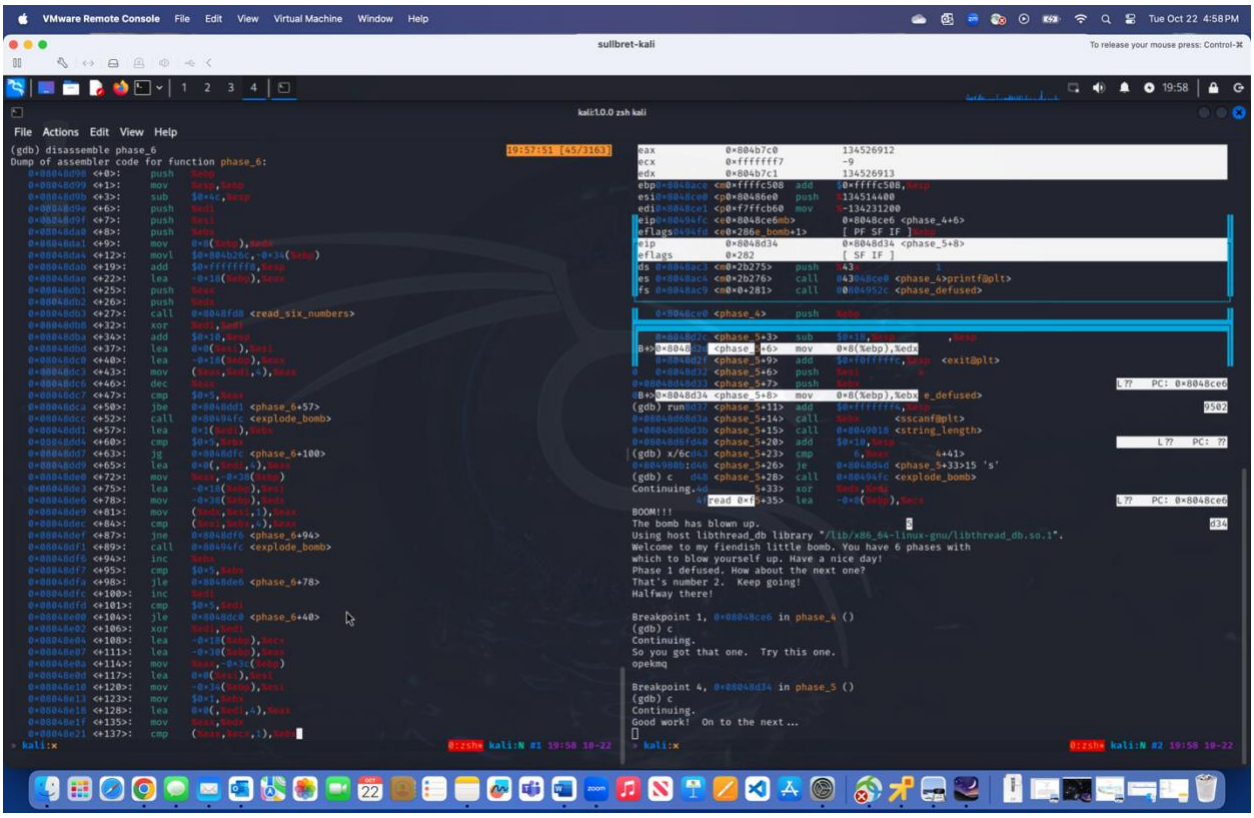
We now know that our string needs to be turned into the string “giants”. The only thing left to figure out is what is happening to each character to achieve this as our final string, as ours became “usiegi”. After much digging through what is happening during the loop to try to avoid a long character mapping through tons of trial and error, I noticed by looking at the register %esi, I see that an array of characters is stored there, with each character having an index. The loop takes each character from the string, applies an AND operation with the value 0xf on line <phase_5+46>, and uses the result to get a character from this array. By understanding this process, I was able to reverse it to decode "giants" without having to map each character individually. It appears the string “opekmq” is one that should work here, so we will test that, shown below:



The string “opekmq” seems to work! Onto the next and final phase.

Phase 6:

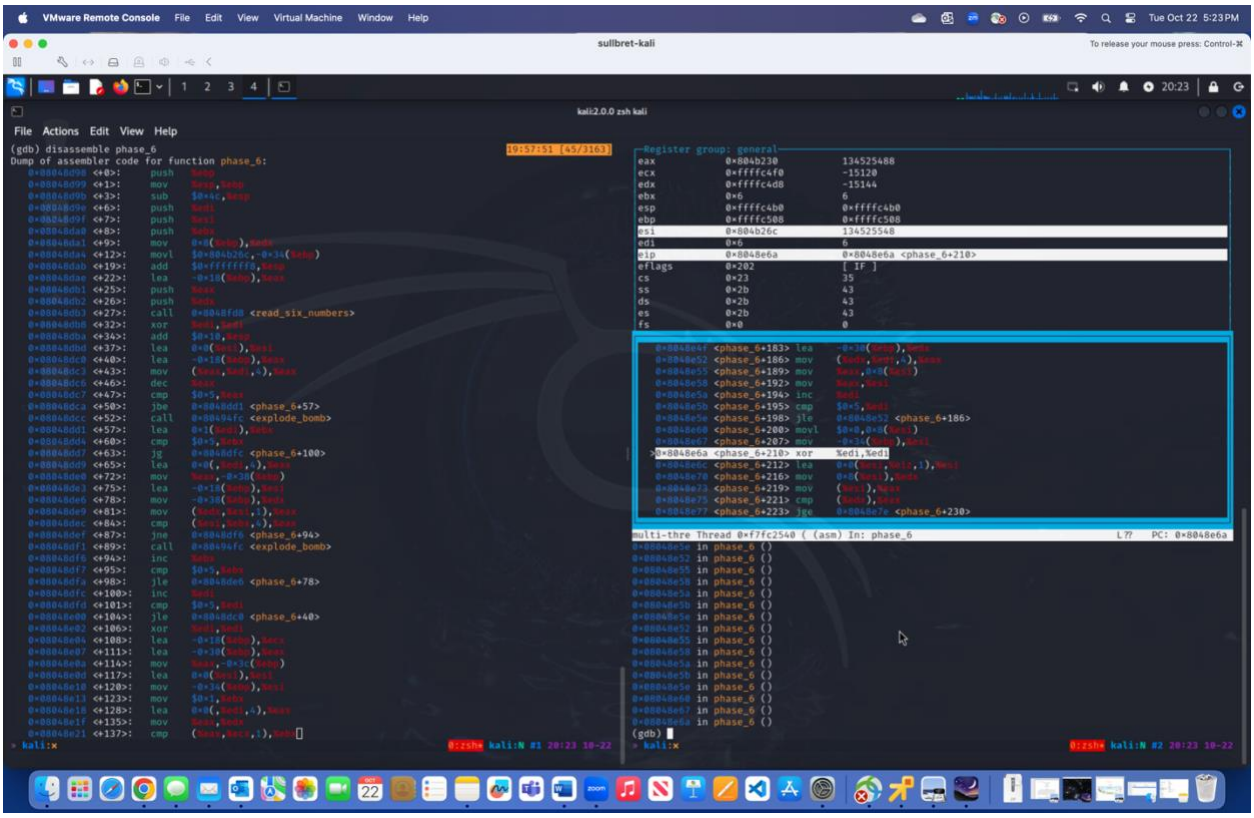
The first things that stick out about phase_6 is that there are a lot of instructions, and the only function being called is one we used in phase_2, which was read_six_numbers. We will assume for now that our input expected will be 6 integers separated by spaces, as before.



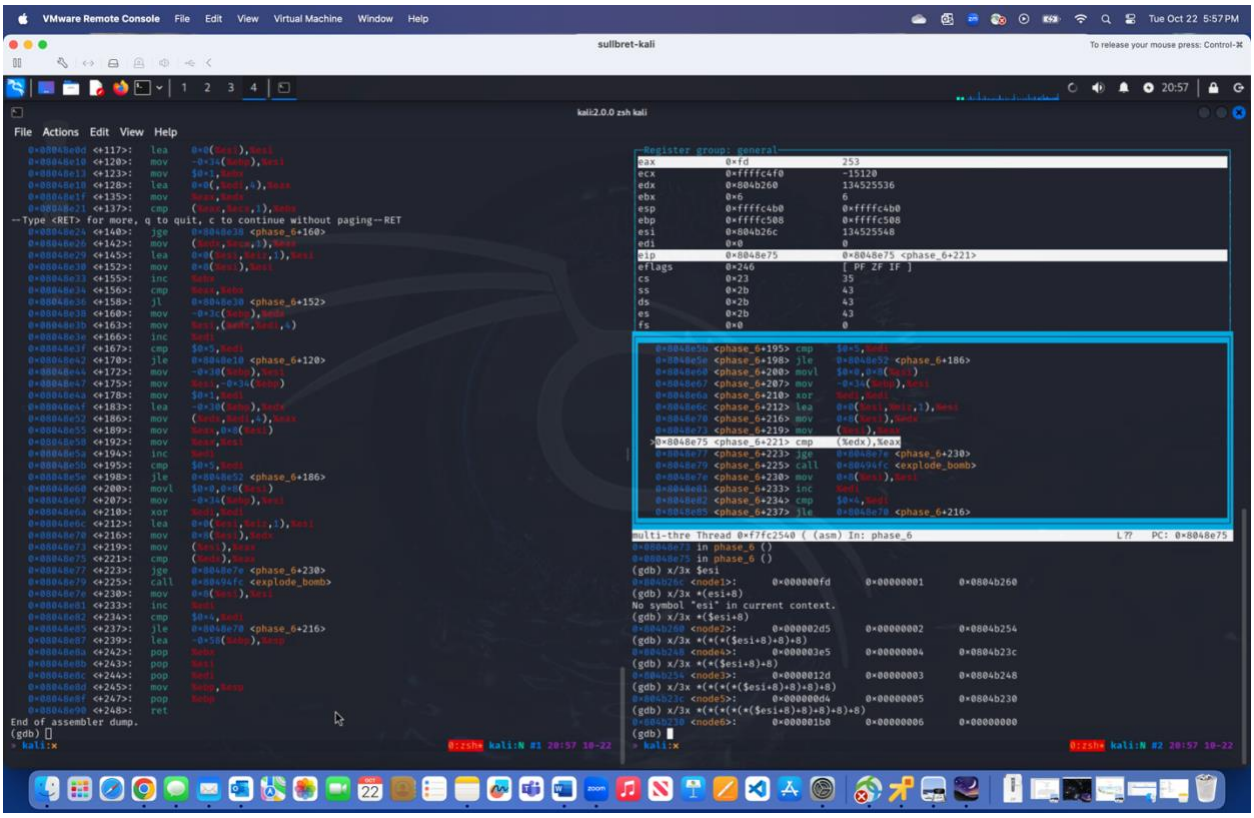
So, out first line of interest is where this is called at <+27>. The next thing that stands out is the comparison on line <+50> where a loop is comparing each int to 5, and if below or even it skips over the bomb explode call. This tells us each int needs to be less than 6, along with what we already know, that our input needing to be 6 integers separated by spaces. The next thing that jumps out is a loop starting at <+68> and finishing at <+98>. Here a comparison is happening between eax, two registers and an int, which turns out is checking to see if any of the inputted numbers are the same as it loops through them.

The last condition for our input is checked within the nested loops between <phase_6+57> and <phase_6+104>. In these loops, the iterators %ebx and %edi are used. What happens here is that each number is compared to every other number, and the bomb will explode at <phase_6+89> if any of the comparisons show that the numbers are the same, so all six integers we input must be unique.

Now we will test the integers “1 2 3 4 5 6” and try to walk through what is happening. After finally stepping through all the loops with that input working so far, the next lines of interest are the few that I am on now, shown below:

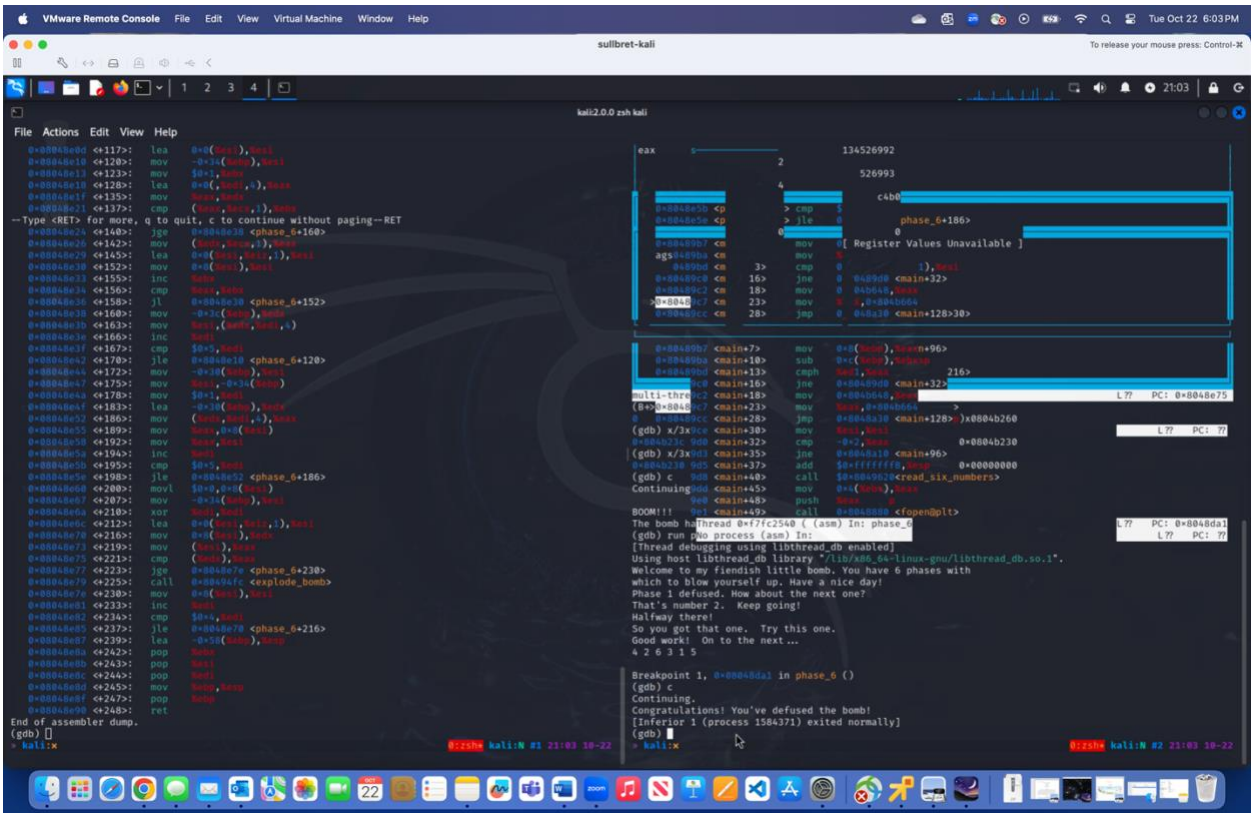


First, the value from %esi + 8 is moved into %edx, and then the value in %esi is moved into %eax. These two values are compared. If the value in %edx (which corresponds to %esi + 0x8) is less than the value in %eax (which is just %esi), the bomb will explode. Upon inspecting the values at %esi and %esi + 0x8, we notice they represent structures called <node1> and <node2>, each having three parts. The second part clearly represents the node number (1 for <node1>, 2 for <node2>, and so on), while the third part looks like a pointer. Analyzing each of these shown below:



The first column of values appears to be random, which suggests we're looking at a familiar data structure: a singly linked list. The first column represents the node's value, the second shows the node's position in the list, and the third is a pointer to the next node.

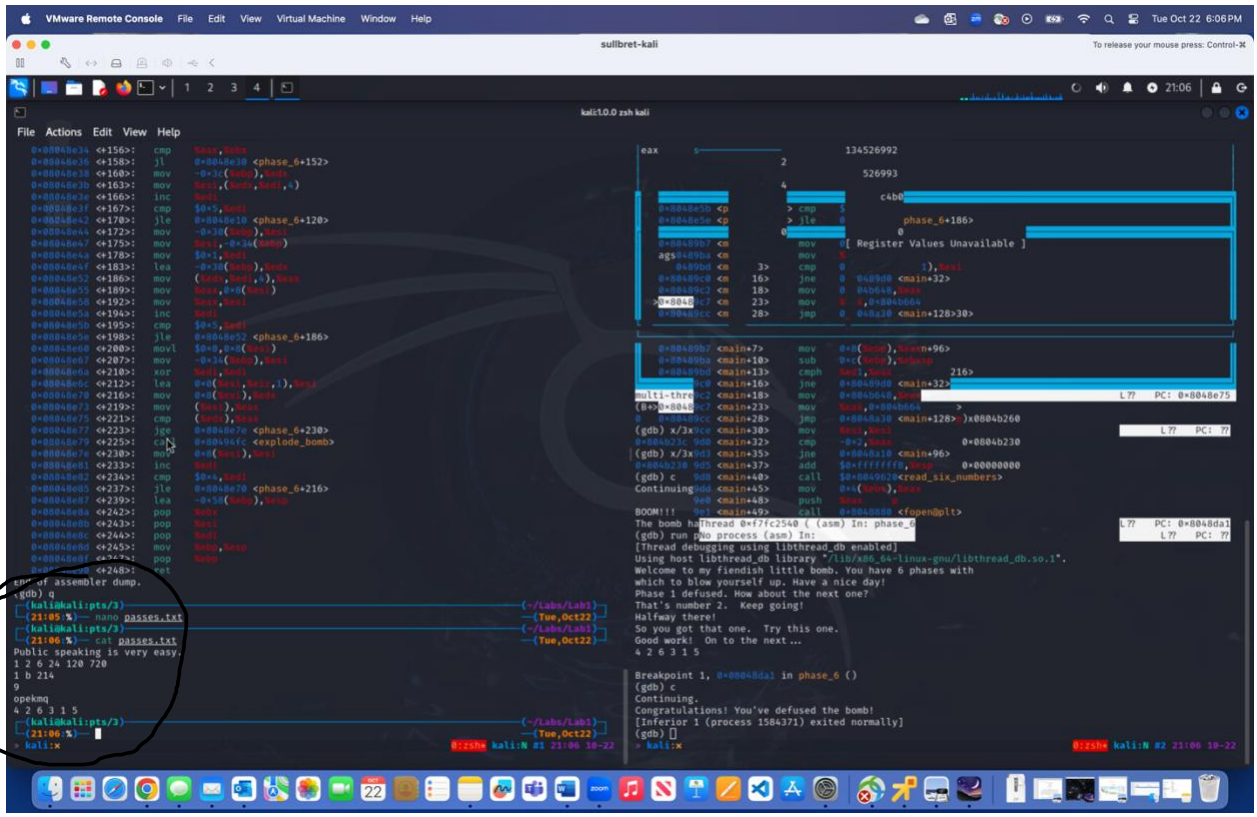
Now that we understand this, when %edx and %eax are compared at <phase_6+221>, the program is checking the value of the current node in the linked list against the next node's value. If the next node's value isn't smaller than the current one, the bomb goes off. Essentially, we need to arrange the nodes from highest to lowest by providing the correct input. Looking at the values we just got printed for each node, we can arrange them accordingly to make sure this works. Doing this, the order of each node from largest to smallest becomes nodes "4 2 6 3 1 5". Let's run this to be sure, shown below.



Voila, the bomb has been diffused.

Here are all the phrases listed, that I have been adding to my passes.txt file each time.

Shown in screenshot below in the bottom left corner:


VMware Remote Console File Edit View Virtual Machine Window Help
Tue Oct 22 6:06 PM
sullbret-kali
To release your mouse press: Control-⌘
kali:1.0.0 zsh kali
File Actions Edit View Help
Register Values Unavailable
End of assembler dump.
(gdb) q
(kali㉿kali:pts/3)
(21:05:%)
nano passes.txt
(Tue,Oct22)
(kali㉿kali:pts/3)
(21:06:%)
cat passes.txt
(Tue,Oct22)
Public speaking is very easy.
1 2 6 24 120 720
1 b 214
9
opekmq
4 2 6 3 1 5
(kali㉿kali:pts/3)
(21:06:%)
(~/Labs/Lab1)
» kali:x
BOOM!!!
The bomb has blown up.
(gdb) run
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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...
4 2 6 3 1 5
Breakpoint 1, 0x08048da1 in phase_6 ()
(gdb) c
Continuing.
Congratulations! You've defused the bomb!
[Inferior 1 (process 1584371) exited normally]
(gdb)
» kali:x