

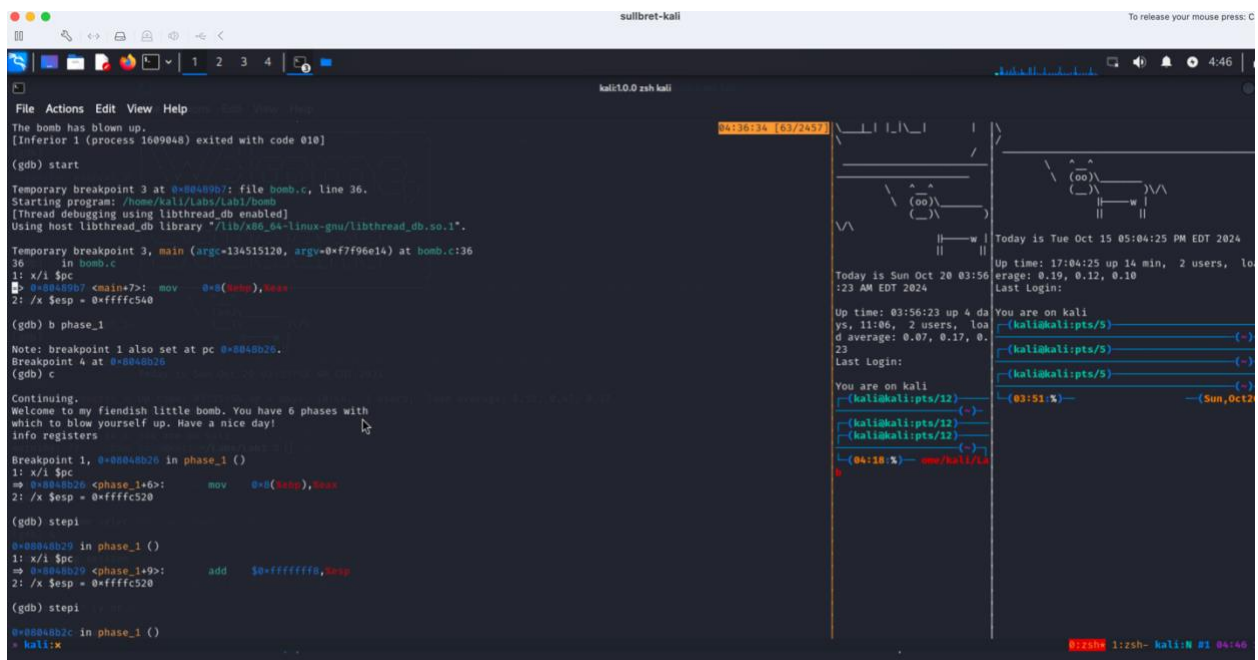
Brett Sullivan

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.



```
File Actions Edit View Help
The bomb has blown up.
[Inferior 1 (process 1009040) exited with code 010]

(gdb) start

Temporary breakpoint 3 at 0x00400b20: file bomb.c, line 36.
Starting program: /home/kali/Labs/Lab1/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 3, main (argc=134515120, argv=0xf7f96e14) at bomb.c:36
36   in bomb.c
1: x/i $pc
=> 0x00400b20 <main+7>: mov     0x0(%eax),%eax
2: /x $esp = 0xffffc540

(gdb) b phase_1

Note: breakpoint 1 also set at pc 0x00400b20.
Breakpoint 4 at 0x00400b20
(gdb) c

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, 0x00400b20 in phase_1 ()
1: x/i $pc
=> 0x00400b20 <phase_1+0>: mov     0x0(%eax),%eax
2: /x $esp = 0xffffc520

(gdb) stepi

0x00400b22 in phase_1 ()
1: x/i $pc
=> 0x00400b22 <phase_1+2>: add     $0xfffffff0,%esp
2: /x $esp = 0xffffc520

(gdb) stepi

0x00400b2c in phase_1 ()
= kali:x
```

```
sullbret-kali
kali10.0 zsh kali

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, 0x0040b26 in phase_1 ()
1: x/1 $pc
=> 0x0040b26 <phase_1+6>: mov     0x0(%eax),%eax
2: /x $esp = 0xffffc520

(gdb) stepi

0x0040b29 in phase_1 ()
1: x/1 $pc
=> 0x0040b29 <phase_1+9>: add     $0+0xfffff8,%eax
2: /x $esp = 0xffffc520

(gdb) stepi

0x0040b2c in phase_1 ()
1: x/1 $pc
=> 0x0040b2c <phase_1+12>: push   $0+0x00407c0
2: /x $esp = 0xffffc510

(gdb) stepi

0x0040b31 in phase_1 ()
1: x/1 $pc
=> 0x0040b31 <phase_1+17>: push   %eax
2: /x $esp = 0xffffc514

(gdb) stepi

0x0040b32 in phase_1 ()
1: x/1 $pc
=> 0x0040b32 <phase_1+18>: call   0x0040b30 <strings_not_equal>
2: /x $esp = 0xffffc510

(gdb) x/s 0xffffc510
0xffffc510: "\200\266\004\b\300\227\004\b\305\377\377\b\222\004\b\313\377\367\332\377\367\305\377\377\212\004\b\200\266\004\b"
= kali:x

04:37:17 (42/2457)
```

```
sullbret-kali
kali10.0 zsh kali

b5\265\004\b\305\377\377\212\004\b\024n\371\367d\330\327\367\300\356", <incomplete sequence \367>
(gdb) strings

Undefined command: "strings". Try 'help'.
(gdb) strings bomb | less

Undefined command: "strings". Try 'help'.
(gdb) info registers
eax             0x004b600      134526592
ecx             0xfffffee     -18
edx             0x004b601      134526593
ebx             0xffffc14     -14028
ebp             0xffffc510     0xffffc510
ebp             0xffffc528     0xffffc528
esi             0x004b6e0      134514400
edi             0xf7fcb60     -134231200
eip             0x0040b32     0x0040b32 <phase_1+18>
eflags          0x207         [ CF PF SF IF ]
cs              0x23         35
ss              0x2b         43
ds              0x2b         43
es              0x2b         43
fs              0x0          0
gs              0x63         99
(gdb) info stack
#0  0x0040b32 in phase_1 ()
#1  0x0040a80 in main (argc=134526592, argv=0xffffc014) at bomb.c:173
(gdb) x/s 0x004b600
0x004b600 <input_strings>: "info registers "
(gdb) x/s 0x004b601
0x004b601 <input_strings+1>: "nfo registers "
(gdb) x/s 0x0
0x00: <error: Cannot access memory at address 0x00>
(gdb) x/s 0x00497c0
0x00497c0: "Public speaking is very easy."
(gdb)
= kali:x

04:43:58 (0/2457)
```

```
File Actions Edit View Help
(gdb) c
Continuing.
BOOM!!!
The bomb has blown up.
[Inferior 1 (process 1651194) exited with code 010]
(gdb) start
Temporary breakpoint 5 at 0x004090d7: file bomb.c, line 36.
Starting program: /home/kali/Labs/Lab1/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 5, main (argc=134515120, argv=0xf7f96e14) at bomb.c:36
36      in bomb.c
1: x/i $pc
=> 0x004090d7 <main+7>: mov     0x0(%ebp),%eax
2: /x $esp = 0xffffc540

(gdb) b phase_1
Note: breakpoints 1 and 4 also set at pc 0x00408026.
Breakpoint 6 at 0x00408026
(gdb) c
Continuing.
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.

Breakpoint 1, 0x00408026 in phase_1 ()
1: x/i $pc
=> 0x00408026 <phase_1+6>: mov     0x0(%ebp),%eax
2: /x $esp = 0xffffc520

(gdb) c
Continuing.
Phase 1 defused. How about the next one?
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.

```
sullbret-kali
File Actions Edit View Help
edx 0x804b681 134526593 ebx 0xffffc614 -14828
eax 0xffffc510 -15088 ecx 0xffffffff8 -8
edx 0x804b6d0 134526672 ebx 0xffffc614 -14828
esp 0xffffc4f4 0xffffc4f4 ebp 0xffffc528 0xffffc528
esi 0x80486e0 134514400 edi 0xf7fcb60 -134231200
eip 0x8048b5a 0x8048b5a <phase_2+18> eflags 0x283 [ CF SF IF ]
cs 0x23 35 ss 0x2b 43
ds 0x2b 43 es 0x2b 43
fs 0x0 0 gs 0x63 99
B> 0x80489b7 <main+7> mov 0x8(%ebp),%eax
0x80489ba <main+10> mov 0xc(%ebp),%ebx
0x8048b20 <phase_1> push %ebp n+32>
0x8048b4f <phase_2+7> push %ebx
B> 0x8048b50 <phase_2+8> mov 0x8(%ebp),%edx
0x8048b53 <phase_2+11> add $0xffffffff8,%esp
0x8048b56 <phase_2+14> lea -0x18(%ebp),%eax
0x8048b59 <phase_2+17> push %eax
> 0x8048b5a <phase_2+18> push %edx
0x8048b5b <phase_2+19> call 0x8048fd8 <read_six_numbers>
0x8048b60 <phase_2+24> add $0x10,%esp
0x8048b63 <phase_2+27> cmpl $0x1,-0x18(%ebp)
0x8048b67 <phase_2+31> je 0x8048b6e <phase_2+38>
0x8048b69 <phase_2+33> call 0x80494fc <explode_bomb>
0x8048b6e <phase_2+38> mov $0x1,%ebx
Breakpoint 4 at 0x8048b26
(gdb) b explode
(gdb) b explode
eip 0x8048b56 0x8048b56 <phase_2+14>
eflags 0x283 [ CF SF IF ]
cs 0x23 35
ss 0x2b 43
ds 0x2b 43
es 0x2b 43
fs 0x0 0
gs 0x63 99
--Type <RET> for more, q to quit, c to continue without paging--RET
(gdb) stepi
0x8048b59 in phase_2 ()
(gdb) stepi
0x8048b5a in phase_2 ()
(gdb)
» kali:~
```

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:

```

kali1.0.0 zsh kali
File Actions Edit View Help
Register group: general
eax 0xffffc520 -15072
ecx 0x804b6d0 134526672
edx 0xffffc510 -15088
ebx 0xffffc614 -14828
esp 0xffffc4dc 0xffffc4dc
ebp 0xffffc4e8 0xffffc4e8
esi 0x80486e0 134514400
edi 0xf7ffc6b0 -134231200
eip 0x8048feb 0x8048feb <read_six_numbers+19>
eflags 0x282 [ SF IF ]
cs 0x23 35
ss 0x2b 43
ds 0x2b 43
es 0x2b 43
fs 0x0 0
gs 0x63 99

0x8048ff3 <read_six_numbers+27> push    %eax
0x8048ff4 <read_six_numbers+28> lea     0x4(%edx),%eax
0x8048ff7 <read_six_numbers+31> push    %eax
0x8048ff8 <read_six_numbers+32> push    %edx
0x8048ff9 <read_six_numbers+33> push    $0x8049b1b
0x8048ffe <read_six_numbers+38> push    %ecx
0x8048fff <read_six_numbers+39> call    0x8048860 <sscanf@plt>
0x8049004 <read_six_numbers+44> add     $0x20,%esp
0x8049007 <read_six_numbers+47> cmp     $0x5,%eax
0x804900a <read_six_numbers+50> jg      0x8049011 <read_six_numbers+57>
0x804900c <read_six_numbers+52> call    0x80494fc <explode_bomb>
0x8049011 <read_six_numbers+57> mov     %ebp,%esp
0x8049013 <read_six_numbers+59> pop     %ebp
0x8049014 <read_six_numbers+60> ret
0x8049015 <read_six_numbers+61> lea     0x0(%esi),%esi
0x8049018 <string_length> push    %ebp
0x8049019 <string_length+1> mov     %esp,%ebp

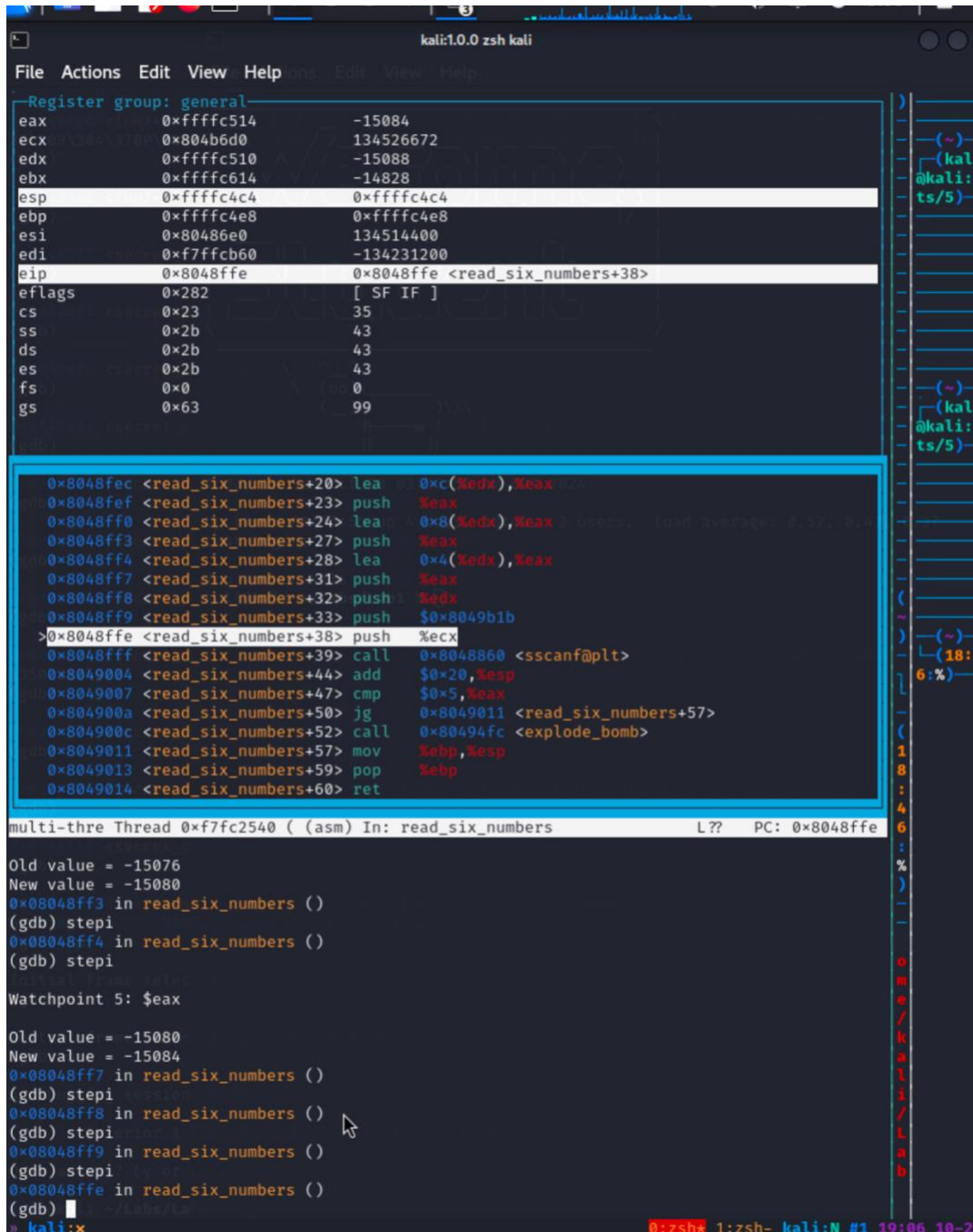
multi-thre Thread 0xf7fc2540 ( asm) In: read_six_numbers L?? PC: 0x8048feb
esi 0x80486e0 134514400
edi 0xf7ffc6b0 -134231200
eip 0x8048feb 0x8048feb <read_six_numbers+19>
eflags 0x282 [ SF IF ]
cs 0x23 35
ss 0x2b 43
ds 0x2b 43
--Type <RET> for more, q to quit, c to continue without paging--q
Quit? n
(gdb) info stack
#0  0x08048feb in read_six_numbers ()
#1  0x08048b60 in phase_2 ()
#2  0x08048a83 in main (argc=-15072, argv=0xffffc614) at bomb.c:81
(gdb) watch $eax
Watchpoint 5: $eax
(gdb) x/s 0x8049b1b
0x8049b1b: "%d %d %d %d %d %d"
(gdb)

kali:~/.Lab
» kali:
0:zsh* 1:zsh- kali:N #1 18:59 10-20

```

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.

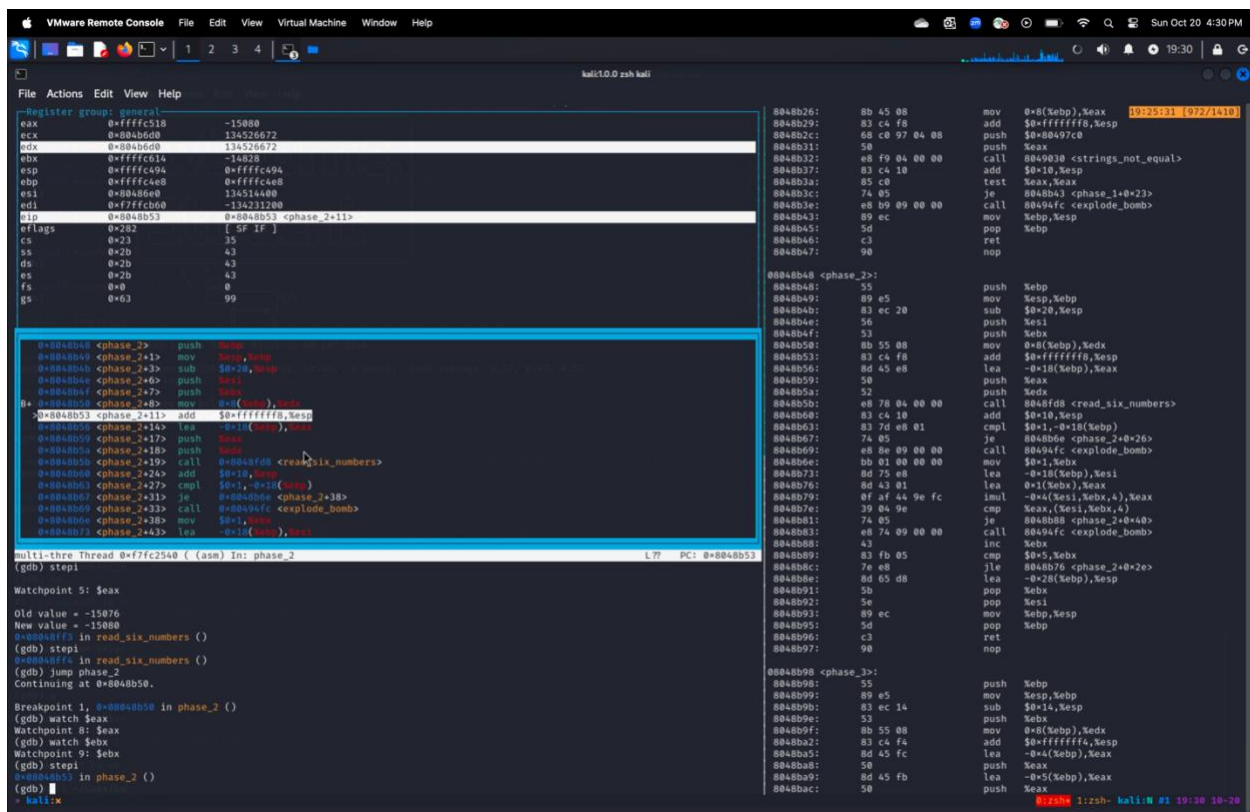


```
kali:1.0.0 zsh kali
File Actions Edit View Help
Register group: general
eax      0xffffc514      -15084
ecx      0x804b6d0      134526672
edx      0xffffc510      -15088
ebx      0xffffc614      -14828
esp      0xffffc4c4      0xffffc4c4
ebp      0xffffc4e8      0xffffc4e8
esi      0x80486e0      134514400
edi      0xf7fcb60      -134231200
eip      0x8048ffe      0x8048ffe <read_six_numbers+38>
eflags   0x282          [ SF IF ]
cs       0x23          35
ss       0x2b          43
ds       0x2b          43
es       0x2b          43
fs       0x0           0
gs       0x63          99

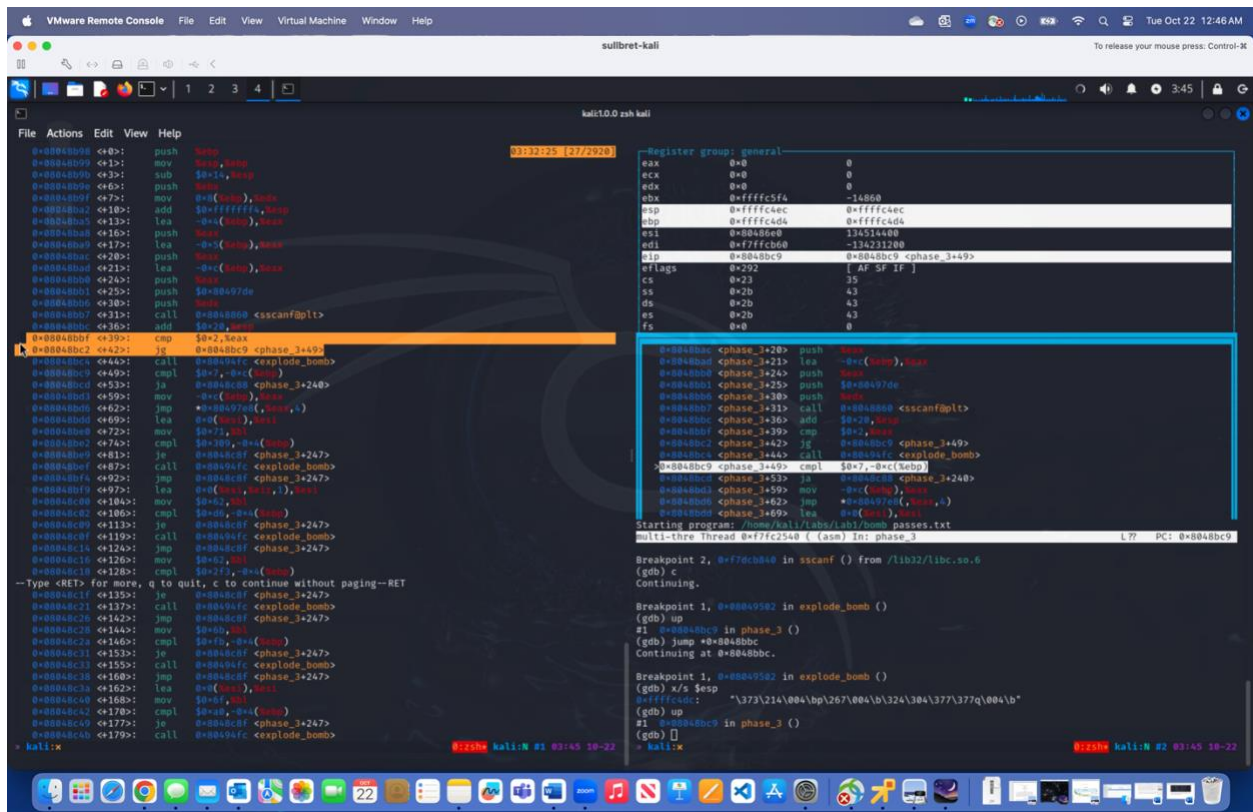
0x8048fec <read_six_numbers+20> lea    0xc(%edx),%eax
0x8048fef <read_six_numbers+23> push  %eax
0x8048ff0 <read_six_numbers+24> lea    0x8(%edx),%eax
0x8048ff3 <read_six_numbers+27> push  %eax
0x8048ff4 <read_six_numbers+28> lea    0x4(%edx),%eax
0x8048ff7 <read_six_numbers+31> push  %eax
0x8048ff8 <read_six_numbers+32> push  %edx
0x8048ff9 <read_six_numbers+33> push  $0x8049b1b
0x8048ffe <read_six_numbers+38> push  %ecx
0x8048fff <read_six_numbers+39> call  0x8048860 <sscanf@plt>
0x8049004 <read_six_numbers+44> add    $0x20,%esp
0x8049007 <read_six_numbers+47> cmp    $0x5,%eax
0x804900a <read_six_numbers+50> jg     0x8049011 <read_six_numbers+57>
0x804900c <read_six_numbers+52> call  0x80494fc <explode_bomb>
0x8049011 <read_six_numbers+57> mov    %ebp,%esp
0x8049013 <read_six_numbers+59> pop    %ebp
0x8049014 <read_six_numbers+60> ret

multi-thre Thread 0xf7fc2540 ( (asm) In: read_six_numbers L?? PC: 0x8048ffe
Old value = -15076
New value = -15080
0x08048ff3 in read_six_numbers ()
(gdb) stepi
0x08048ff4 in read_six_numbers ()
(gdb) stepi
Watchpoint 5: $eax
Old value = -15080
New value = -15084
0x08048ff7 in read_six_numbers ()
(gdb) stepi
0x08048ff8 in read_six_numbers ()
(gdb) stepi
0x08048ff9 in read_six_numbers ()
(gdb) stepi
0x08048ffe in read_six_numbers ()
(gdb)
kali:~
```

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.

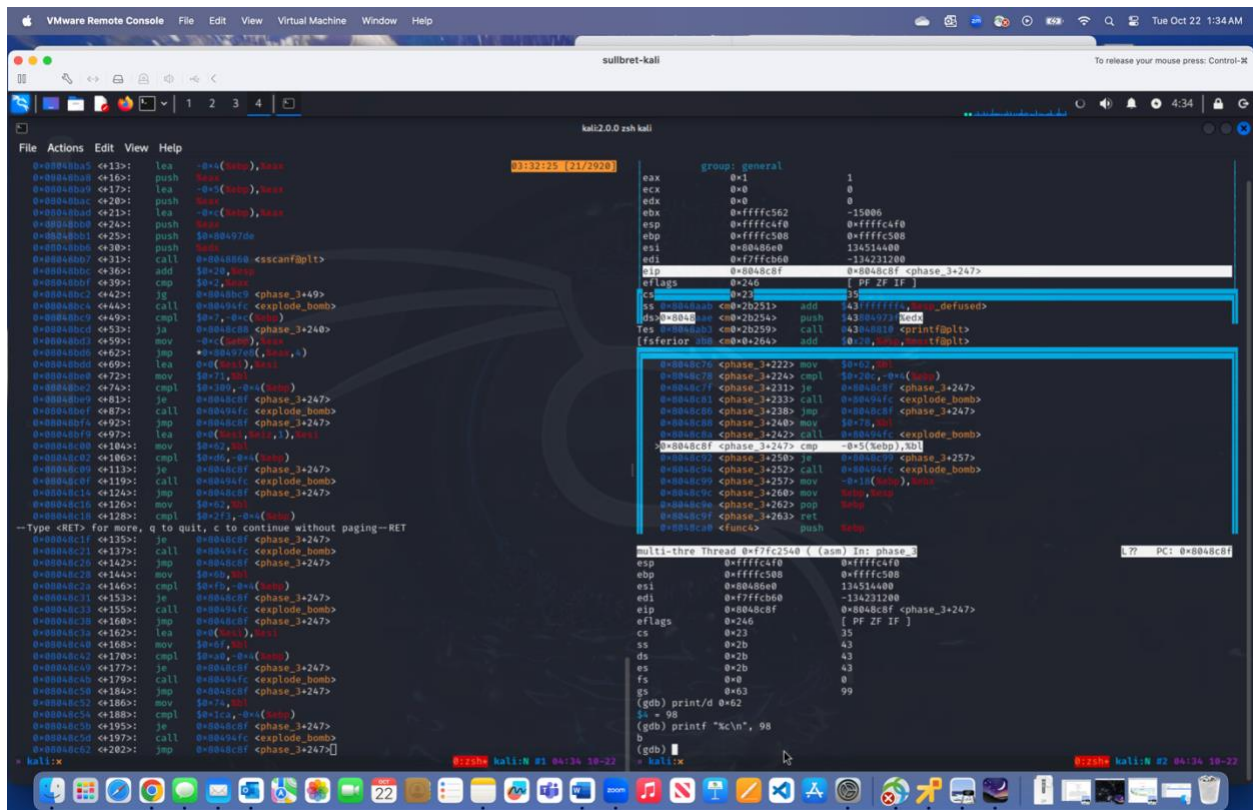


We will not continue stepping through to see where the jumps are happening when we try the phrase '1 a 2' shown below


```
File Actions Edit View Help
0x00400908 <+0>: push $heap
0x00400909 <+1>: mov $eax, %eax
0x0040090b <+3>: sub $0x10, %eax
0x0040090e <+6>: push $eax
0x0040090f <+7>: mov 0x0(%eax), %eax
0x00400910 <+8>: add $0xffffffff, %eax
0x00400911 <+9>: lea -0x1(%eax), %eax
0x00400912 <+10>: push %eax
0x00400913 <+11>: lea -0x1(%eax), %eax
0x00400914 <+12>: push %eax
0x00400915 <+13>: lea -0x1(%eax), %eax
0x00400916 <+14>: push %eax
0x00400917 <+15>: lea -0x1(%eax), %eax
0x00400918 <+16>: je 0x00400918
0x00400919 <+17>: call 0x00400919
0x0040091a <+18>: add $0x28, %eax
0x0040091b <+19>: cmp $0x7, %eax
0x0040091c <+20>: jg 0x0040091c
0x0040091d <+21>: call 0x0040091d
0x0040091e <+22>: call 0x0040091e
0x0040091f <+23>: cmp $0x1, 0x0(%eax)
0x00400920 <+24>: ja 0x00400920
0x00400921 <+25>: mov -0x1(%eax), %eax
0x00400922 <+26>: jmp 0x00400922
0x00400923 <+27>: lea 0x0(%eax), %eax
0x00400924 <+28>: mov $0x7, %eax
0x00400925 <+29>: cmpl $0x105, 0x0(%eax)
0x00400926 <+30>: je 0x00400926
0x00400927 <+31>: call 0x00400927
0x00400928 <+32>: call 0x00400928
0x00400929 <+33>: jmp 0x00400929
0x0040092a <+34>: lea 0x0(%eax), %eax
0x0040092b <+35>: mov $0x2, %eax
0x0040092c <+36>: cmpl $0x46, 0x0(%eax)
0x0040092d <+37>: je 0x0040092d
0x0040092e <+38>: call 0x0040092e
0x0040092f <+39>: jmp 0x0040092f
0x00400930 <+40>: mov $0x2, %eax
0x00400931 <+41>: cmpl $0x46, 0x0(%eax)
0x00400932 <+42>: je 0x00400932
0x00400933 <+43>: call 0x00400933
0x00400934 <+44>: jmp 0x00400934
0x00400935 <+45>: mov $0x0, %eax
0x00400936 <+46>: cmpl $0x7b, 0x0(%eax)
0x00400937 <+47>: je 0x00400937
0x00400938 <+48>: call 0x00400938
0x00400939 <+49>: jmp 0x00400939
0x0040093a <+50>: lea 0x0(%eax), %eax
0x0040093b <+51>: mov $0x0f, %eax
0x0040093c <+52>: cmpl $0x0a, 0x0(%eax)
0x0040093d <+53>: je 0x0040093d
0x0040093e <+54>: call 0x0040093e
0x0040093f <+55>: jmp 0x0040093f
0x00400940 <+56>: call 0x00400940
0x00400941 <+57>: jmp 0x00400941
0x00400942 <+58>: call 0x00400942
0x00400943 <+59>: jmp 0x00400943
0x00400944 <+60>: call 0x00400944
0x00400945 <+61>: jmp 0x00400945
0x00400946 <+62>: call 0x00400946
0x00400947 <+63>: jmp 0x00400947
0x00400948 <+64>: call 0x00400948
0x00400949 <+65>: jmp 0x00400949
0x0040094a <+66>: call 0x0040094a
0x0040094b <+67>: jmp 0x0040094b
0x0040094c <+68>: call 0x0040094c
0x0040094d <+69>: jmp 0x0040094d
0x0040094e <+70>: call 0x0040094e
0x0040094f <+71>: jmp 0x0040094f
0x00400950 <+72>: call 0x00400950
0x00400951 <+73>: jmp 0x00400951
0x00400952 <+74>: call 0x00400952
0x00400953 <+75>: jmp 0x00400953
0x00400954 <+76>: call 0x00400954
0x00400955 <+77>: jmp 0x00400955
0x00400956 <+78>: call 0x00400956
0x00400957 <+79>: jmp 0x00400957
0x00400958 <+80>: call 0x00400958
0x00400959 <+81>: jmp 0x00400959
0x0040095a <+82>: call 0x0040095a
0x0040095b <+83>: jmp 0x0040095b
0x0040095c <+84>: call 0x0040095c
0x0040095d <+85>: jmp 0x0040095d
0x0040095e <+86>: call 0x0040095e
0x0040095f <+87>: jmp 0x0040095f
0x00400960 <+88>: call 0x00400960
0x00400961 <+89>: jmp 0x00400961
0x00400962 <+90>: call 0x00400962
0x00400963 <+91>: jmp 0x00400963
0x00400964 <+92>: call 0x00400964
0x00400965 <+93>: jmp 0x00400965
0x00400966 <+94>: call 0x00400966
0x00400967 <+95>: jmp 0x00400967
0x00400968 <+96>: call 0x00400968
0x00400969 <+97>: jmp 0x00400969
0x0040096a <+98>: call 0x0040096a
0x0040096b <+99>: jmp 0x0040096b
0x0040096c <+100>: call 0x0040096c
0x0040096d <+101>: jmp 0x0040096d
0x0040096e <+102>: call 0x0040096e
0x0040096f <+103>: jmp 0x0040096f
0x00400970 <+104>: call 0x00400970
0x00400971 <+105>: jmp 0x00400971
0x00400972 <+106>: call 0x00400972
0x00400973 <+107>: jmp 0x00400973
0x00400974 <+108>: call 0x00400974
0x00400975 <+109>: jmp 0x00400975
0x00400976 <+110>: call 0x00400976
0x00400977 <+111>: jmp 0x00400977
0x00400978 <+112>: call 0x00400978
0x00400979 <+113>: jmp 0x00400979
0x0040097a <+114>: call 0x0040097a
0x0040097b <+115>: jmp 0x0040097b
0x0040097c <+116>: call 0x0040097c
0x0040097d <+117>: jmp 0x0040097d
0x0040097e <+118>: call 0x0040097e
0x0040097f <+119>: jmp 0x0040097f
0x00400980 <+120>: call 0x00400980
0x00400981 <+121>: jmp 0x00400981
0x00400982 <+122>: call 0x00400982
0x00400983 <+123>: jmp 0x00400983
0x00400984 <+124>: call 0x00400984
0x00400985 <+125>: jmp 0x00400985
0x00400986 <+126>: call 0x00400986
0x00400987 <+127>: jmp 0x00400987
0x00400988 <+128>: call 0x00400988
0x00400989 <+129>: jmp 0x00400989
0x0040098a <+130>: call 0x0040098a
0x0040098b <+131>: jmp 0x0040098b
0x0040098c <+132>: call 0x0040098c
0x0040098d <+133>: jmp 0x0040098d
0x0040098e <+134>: call 0x0040098e
0x0040098f <+135>: jmp 0x0040098f
0x00400990 <+136>: call 0x00400990
0x00400991 <+137>: jmp 0x00400991
0x00400992 <+138>: call 0x00400992
0x00400993 <+139>: jmp 0x00400993
0x00400994 <+140>: call 0x00400994
0x00400995 <+141>: jmp 0x00400995
0x00400996 <+142>: call 0x00400996
0x00400997 <+143>: jmp 0x00400997
0x00400998 <+144>: call 0x00400998
0x00400999 <+145>: jmp 0x00400999
0x0040099a <+146>: call 0x0040099a
0x0040099b <+147>: jmp 0x0040099b
0x0040099c <+148>: call 0x0040099c
0x0040099d <+149>: jmp 0x0040099d
0x0040099e <+150>: call 0x0040099e
0x0040099f <+151>: jmp 0x0040099f
0x004009a0 <+152>: call 0x004009a0
0x004009a1 <+153>: jmp 0x004009a1
0x004009a2 <+154>: call 0x004009a2
0x004009a3 <+155>: jmp 0x004009a3
0x004009a4 <+156>: call 0x004009a4
0x004009a5 <+157>: jmp 0x004009a5
0x004009a6 <+158>: call 0x004009a6
0x004009a7 <+159>: jmp 0x004009a7
0x004009a8 <+160>: call 0x004009a8
0x004009a9 <+161>: jmp 0x004009a9
0x004009aa <+162>: call 0x004009aa
0x004009ab <+163>: jmp 0x004009ab
0x004009ac <+164>: call 0x004009ac
0x004009ad <+165>: jmp 0x004009ad
0x004009ae <+166>: call 0x004009ae
0x004009af <+167>: jmp 0x004009af
0x004009b0 <+168>: call 0x004009b0
0x004009b1 <+169>: jmp 0x004009b1
0x004009b2 <+170>: call 0x004009b2
0x004009b3 <+171>: jmp 0x004009b3
0x004009b4 <+172>: call 0x004009b4
0x004009b5 <+173>: jmp 0x004009b5
0x004009b6 <+174>: call 0x004009b6
0x004009b7 <+175>: jmp 0x004009b7
0x004009b8 <+176>: call 0x004009b8
0x004009b9 <+177>: jmp 0x004009b9
0x004009ba <+178>: call 0x004009ba
0x004009bb <+179>: jmp 0x004009bb
0x004009bc <+180>: call 0x004009bc
0x004009bd <+181>: jmp 0x004009bd
0x004009be <+182>: call 0x004009be
0x004009bf <+183>: jmp 0x004009bf
0x004009c0 <+184>: call 0x004009c0
0x004009c1 <+185>: jmp 0x004009c1
0x004009c2 <+186>: call 0x004009c2
0x004009c3 <+187>: jmp 0x004009c3
0x004009c4 <+188>: call 0x004009c4
0x004009c5 <+189>: jmp 0x004009c5
0x004009c6 <+190>: call 0x004009c6
0x004009c7 <+191>: jmp 0x004009c7
0x004009c8 <+192>: call 0x004009c8
0x004009c9 <+193>: jmp 0x004009c9
0x004009ca <+194>: call 0x004009ca
0x004009cb <+195>: jmp 0x004009cb
0x004009cc <+196>: call 0x004009cc
0x004009cd <+197>: jmp 0x004009cd
0x004009ce <+198>: call 0x004009ce
0x004009cf <+199>: jmp 0x004009cf
0x004009d0 <+200>: call 0x004009d0
0x004009d1 <+201>: jmp 0x004009d1
0x004009d2 <+202>: call 0x004009d2
0x004009d3 <+203>: jmp 0x004009d3
0x004009d4 <+204>: call 0x004009d4
0x004009d5 <+205>: jmp 0x004009d5
0x004009d6 <+206>: call 0x004009d6
0x004009d7 <+207>: jmp 0x004009d7
0x004009d8 <+208>: call 0x004009d8
0x004009d9 <+209>: jmp 0x004009d9
0x004009da <+210>: call 0x004009da
0x004009db <+211>: jmp 0x004009db
0x004009dc <+212>: call 0x004009dc
0x004009dd <+213>: jmp 0x004009dd
0x004009de <+214>: call 0x004009de
0x004009df <+215>: jmp 0x004009df
0x004009e0 <+216>: call 0x004009e0
0x004009e1 <+217>: jmp 0x004009e1
0x004009e2 <+218>: call 0x004009e2
0x004009e3 <+219>: jmp 0x004009e3
0x004009e4 <+220>: call 0x004009e4
0x004009e5 <+221>: jmp 0x004009e5
0x004009e6 <+222>: call 0x004009e6
0x004009e7 <+223>: jmp 0x004009e7
0x004009e8 <+224>: call 0x004009e8
0x004009e9 <+225>: jmp 0x004009e9
0x004009ea <+226>: call 0x004009ea
0x004009eb <+227>: jmp 0x004009eb
0x004009ec <+228>: call 0x004009ec
0x004009ed <+229>: jmp 0x004009ed
0x004009ee <+230>: call 0x004009ee
0x004009ef <+231>: jmp 0x004009ef
0x004009f0 <+232>: call 0x004009f0
0x004009f1 <+233>: jmp 0x004009f1
0x004009f2 <+234>: call 0x004009f2
0x004009f3 <+235>: jmp 0x004009f3
0x004009f4 <+236>: call 0x004009f4
0x004009f5 <+237>: jmp 0x004009f5
0x004009f6 <+238>: call 0x004009f6
0x004009f7 <+239>: jmp 0x004009f7
0x004009f8 <+240>: call 0x004009f8
0x004009f9 <+241>: jmp 0x004009f9
0x004009fa <+242>: call 0x004009fa
0x004009fb <+243>: jmp 0x004009fb
0x004009fc <+244>: call 0x004009fc
0x004009fd <+245>: jmp 0x004009fd
0x004009fe <+246>: call 0x004009fe
0x004009ff <+247>: jmp 0x004009ff
0x00400a00 <+248>: call 0x00400a00
0x00400a01 <+249>: jmp 0x00400a01
0x00400a02 <+250>: call 0x00400a02
0x00400a03 <+251>: jmp 0x00400a03
0x00400a04 <+252>: call 0x00400a04
0x00400a05 <+253>: jmp 0x00400a05
0x00400a06 <+254>: call 0x00400a06
0x00400a07 <+255>: jmp 0x00400a07
0x00400a08 <+256>: call 0x00400a08
0x00400a09 <+257>: jmp 0x00400a09
0x00400a0a <+258>: call 0x00400a0a
0x00400a0b <+259>: jmp 0x00400a0b
0x00400a0c <+260>: call 0x00400a0c
0x00400a0d <+261>: jmp 0x00400a0d
0x00400a0e <+262>: call 0x00400a0e
0x00400a0f <+263>: jmp 0x00400a0f
0x00400a10 <+264>: call 0x00400a10
0x00400a11 <+265>: jmp 0x00400a11
0x00400a12 <+266>: call 0x00400a12
0x00400a13 <+267>: jmp 0x00400a13
0x00400a14 <+268>: call 0x00400a14
0x00400a15 <+269>: jmp 0x00400a15
0x00400a16 <+270>: call 0x00400a16
0x00400a17 <+271>: jmp 0x00400a17
0x00400a18 <+272>: call 0x00400a18
0x00400a19 <+273>: jmp 0x00400a19
0x00400a1a <+274>: call 0x00400a1a
0x00400a1b <+275>: jmp 0x00400a1b
0x00400a1c <+276>: call 0x00400a1c
0x00400a1d <+277>: jmp 0x00400a1d
0x00400a1e <+278>: call 0x00400a1e
0x00400a1f <+279>: jmp 0x00400a1f
0x00400a20 <+280>: call 0x00400a20
0x00400a21 <+281>: jmp 0x00400a21
0x00400a22 <+282>: call 0x00400a22
0x00400a23 <+283>: jmp 0x00400a23
0x00400a24 <+284>: call 0x00400a24
0x00400a25 <+285>: jmp 0x00400a25
0x00400a26 <+286>: call 0x00400a26
0x00400a27 <+287>: jmp 0x00400a27
0x00400a28 <+288>: call 0x00400a28
0x00400a29 <+289>: jmp 0x00400a29
0x00400a2a <+290>: call 0x00400a2a
0x00400a2b <+291>: jmp 0x00400a2b
0x00400a2c <+292>: call 0x00400a2c
0x00400a2d <+293>: jmp 0x00400a2d
0x00400a2e <+294>: call 0x00400a2e
0x00400a2f <+295>: jmp 0x00400a2f
0x00400a30 <+296>: call 0x00400a30
0x00400a31 <+297>: jmp 0x00400a31
0x00400a32 <+298>: call 0x00400a32
0x00400a33 <+299>: jmp 0x00400a33
0x00400a34 <+300>: call 0x00400a34
0x00400a35 <+301>: jmp 0x00400a35
0x00400a36 <+302>: call 0x00400a36
0x00400a37 <+303>: jmp 0x00400a37
0x00400a38 <+304>: call 0x00400a38
0x00400a39 <+305>: jmp 0x00400a39
0x00400a3a <+306>: call 0x00400a3a
0x00400a3b <+307>: jmp 0x00400a3b
0x00400a3c <+308>: call 0x00400a3c
0x00400a3d <+309>: jmp 0x00400a3d
0x00400a3e <+310>: call 0x00400a3e
0x00400a3f <+311>: jmp 0x00400a3f
0x00400a40 <+312>: call 0x00400a40
0x00400a41 <+313>: jmp 0x00400a41
0x00400a42 <+314>: call 0x00400a42
0x00400a43 <+315>: jmp 0x00400a43
0x00400a44 <+316>: call 0x00400a44
0x00400a45 <+317>: jmp 0x00400a45
0x00400a46 <+318>: call 0x00400a46
0x00400a47 <+319>: jmp 0x00400a47
0x00400a48 <+320>: call 0x00400a48
0x00400a49 <+321>: jmp 0x00400a49
0x00400a4a <+322>: call 0x00400a4a
0x00400a4b <+323>: jmp 0x00400a4b
0x00400a4c <+324>: call 0x00400a4c
0x00400a4d <+325>: jmp 0x00400a4d
0x00400a4e <+326>: call 0x00400a4e
0x00400a4f <+327>: jmp 0x00400a4f
0x00400a50 <+328>: call 0x00400a50
0x00400a51 <+329>: jmp 0x00400a51
0x00400a52 <+330>: call 0x00400a52
0x00400a53 <+331>: jmp 0x00400a53
0x00400a54 <+332>: call 0x00400a54
0x00400a55 <+333>: jmp 0x00400a55
0x00400a56 <+334>: call 0x00400a56
0x00400a57 <+335>: jmp 0x00400a57
0x00400a58 <+336>: call 0x00400a58
0x00400a59 <+337>: jmp 0x00400a59
0x00400a5a <+338>: call 0x00400a5a
0x00400a5b <+339>: jmp 0x00400a5b
0x00400a5c <+340>: call 0x00400a5c
0x00400a5d <+341>: jmp 0x00400a5d
0x00400a5e <+342>: call 0x00400a5e
0x00400a5f <+343>: jmp 0x00400a5f
0x00400a60 <+344>: call 0x00400a60
0x00400a61 <+345>: jmp 0x00400a61
0x00400a62 <+346>: call 0x00400a62
0x00400a63 <+347>: jmp 0x00400a63
0x00400a64 <+348>: call 0x00400a64
0x00400a65 <+349>: jmp 0x00400a65
0x00400a66 <+350>: call 0x00400a66
0x00400a67 <+351>: jmp 0x00400a67
0x00400a68 <+352>: call 0x00400a68
0x00400a69 <+353>: jmp 0x00400a69
0x00400a6a <+354>: call 0x00400a6a
0x00400a6b <+355>: jmp 0x00400a6b
0x00400a6c <+356>: call 0x00400a6c
0x00400a6d <+357>: jmp 0x00400a6d
0x00400a6e <+358>: call 0x00400a6e
0x00400a6f <+359>: jmp 0x00400a6f
0x00400a70 <+360>: call 0x00400a70
0x00400a71 <+361>: jmp 0x00400a71
0x00400a72 <+362>: call 0x00400a72
0x00400a73 <+363>: jmp 0x00400a73
0x00400a74 <+364>: call 0x00400a74
0x00400a75 <+365>: jmp 0x00400a75
0x00400a76 <+366>: call 0x00400a76
0x00400a77 <+367>: jmp 0x00400a77
0x00400a78 <+368>: call 0x00400a78
0x00400a79 <+369>: jmp 0x00400a79
0x00400a7a <+370>: call 0x00400a7a
0x00400a7b <+371>: jmp 0x00400a7b
0x00400a7c <+372>: call 0x00400a7c
0x00400a7d <+373>: jmp 0x00400a7d
0x00400a7e <+374>: call 0x00400a7e
0x00400a7f <+375>: jmp 0x00400a7f
0x00400a80 <+376>: call 0x00400a80
0x00400a81 <+377>: jmp 0x00400a81
0x00400a82 <+378>: call 0x00400a82
0x00400a83 <+379>: jmp 0x00400a83
0x00400a84 <+380>: call 0x00400a84
0x0040
```




[illegible]



```
File Actions Edit View Help
0x00000000 <13>: lea -0x(0x0),%eax
0x00000001 <16>: push %eax
0x00000002 <17>: lea -0x(0x0),%eax
0x00000003 <20>: push %eax
0x00000004 <21>: lea -0x(0x0),%eax
0x00000005 <24>: push %eax
0x00000006 <25>: push 0x00000000
0x00000007 <30>: push 0x00000000
0x00000008 <31>: call 0x00000000 <sscanf@plt>
0x00000009 <36>: add $0x20,%eax
0x0000000A <39>: cmp $0x1,%eax
0x0000000B <42>: je 0x0000000C <phase_3+49>
0x0000000C <44>: call 0x00000000 <explode_bomb>
0x0000000D <49>: cmpl $0x7,%eax
0x0000000E <52>: ja 0x0000000F <phase_3+240>
0x0000000F <55>: mov 0x(0x0),%eax
0x00000010 <62>: jmp 0x00000000 <0x0,0x0,>
0x00000011 <69>: lea 0x0(0x0),%eax
0x00000012 <72>: mov $0x1,%eax
0x00000013 <74>: cmpl $0x100,%eax
0x00000014 <81>: je 0x0000000F <phase_3+247>
0x00000015 <87>: call 0x00000000 <explode_bomb>
0x00000016 <92>: jmp 0x0000000F <phase_3+247>
0x00000017 <97>: lea 0x0(0x0,%eax,%eax),%eax
0x00000018 <104>: mov $0x2,%eax
0x00000019 <108>: cmpl $0x0,%eax
0x0000001A <113>: je 0x0000000F <phase_3+247>
0x0000001B <119>: call 0x00000000 <explode_bomb>
0x0000001C <124>: jmp 0x0000000F <phase_3+247>
0x0000001D <126>: mov $0x2,%eax
0x0000001E <128>: cmpl $0x1,%eax
--Type <RET> for more, q to quit, c to continue without paging--RET
0x0000001F <135>: je 0x0000000F <phase_3+247>
0x00000020 <137>: call 0x00000000 <explode_bomb>
0x00000021 <142>: jmp 0x0000000F <phase_3+247>
0x00000022 <144>: mov $0x0,%eax
0x00000023 <146>: cmpl $0x10,%eax
0x00000024 <153>: je 0x0000000F <phase_3+247>
0x00000025 <155>: call 0x00000000 <explode_bomb>
0x00000026 <160>: jmp 0x0000000F <phase_3+247>
0x00000027 <162>: lea 0x0(0x0),%eax
0x00000028 <168>: mov $0x1,%eax
0x00000029 <170>: cmpl $0x0,%eax
0x0000002A <177>: je 0x0000000F <phase_3+247>
0x0000002B <179>: call 0x00000000 <explode_bomb>
0x0000002C <184>: jmp 0x0000000F <phase_3+247>
0x0000002D <186>: mov $0x7,%eax
0x0000002E <188>: cmpl $0x10,%eax
0x0000002F <195>: je 0x0000000F <phase_3+247>
0x00000030 <197>: call 0x00000000 <explode_bomb>
0x00000031 <202>: jmp 0x0000000F <phase_3+247>
group: general
eax 0x1
ecx 0x0
edx 0x0
ebx 0xffffc562
esp 0xffffc4f0
ebp 0xffffc508
esi 0x00000000
edi 0xffffc000
eip 0x00000000 <phase_3+247>
eflags 0x246
cs 0x23
ss 0x23
ds 0x2b
es 0x2b
fs 0x0
gs 0x0
(gdb) print/d $0x2
98
(gdb) printf "%c\n", 98
b
(gdb)
= kali:~
```

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!

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.

```
File Actions Edit View Help
--Type RET> for more, q to quit, c to continue without paging--RET
(gdb) disassemble phase_4
Dump of assembler code for function phase_4:
0x00400c00: push    %eax
0x00400c01: mov     %eax, %ebx
0x00400c02: sub     $0x10, %ebx
0x00400c03: mov     0x0(%eax), %ecx
0x00400c04: add     $0xffffffff, %ecx
0x00400c05: lea     -0x4(%ebx), %eax
0x00400c06: push    %eax
0x00400c07: push    $0x00400800
0x00400c08: push    %eax
0x00400c09: call    0x00400800 <scanf@plt>
0x00400c0a: add     $0x10, %ebx
0x00400c0b: cmpl    %eax, %ecx
0x00400c0c: jne     0x00400d00 <phase_4+41>
0x00400c0d: cmpl    $0x0, 0x4(%ebx)
0x00400c0e: jg      0x00400d00 <phase_4+46>
0x00400c0f: call    0x00400800 <explode_bomb>
0x00400c10: add     $0xffffffff, %eax
0x00400c11: mov     -0x4(%ebx), %eax
0x00400c12: push    %eax
0x00400c13: call    0x00400800 <func4>
0x00400c14: add     $0x10, %ebx
0x00400c15: cmpl    $0x17, %ebx
0x00400c16: je      0x00400d00 <phase_4+71>
0x00400c17: call    0x00400800 <explode_bomb>
0x00400c18: mov     %eax, %eax
0x00400c19: pop     %eax
0x00400c1a: ret

End of assembler dump.
(gdb)

Multi-thread Thread 0x7fc2540 (asm) in: phase_4
L?? PC: 0x00400c09
(gdb) info reg general
Starting program: /home/kali/labs/lab1/bomb/passes.txt
[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
Breakpoint 1, 0x00400c0e in phase_4 () day!
(gdb) step
0x00400c0e in phase_4 ()
(gdb) x/s 0x00400800
0x00400800: "sd"
(gdb)
```

The next thing noticed is that there is a function being called, “func4” below the scanf.

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.

```

File Actions Edit View Help
0x00400000: <204> mov $0x78,%eax
0x00400001: <206> cmpl $0x18,%eax(0x00)
0x00400002: <213> je 0x0040000f <phase_3+247>
0x00400003: <215> call 0x004000fc <explode_bomb>
0x00400004: <220> jmp 0x0040000f <phase_3+247>
0x00400005: <222> mov $0x0,%eax
0x00400006: <224> cmpl $0x18,%eax(0x00)
0x00400007: <231> je 0x0040000f <phase_3+247>
0x00400008: <233> call 0x004000fc <explode_bomb>
0x00400009: <238> jmp 0x0040000f <phase_3+247>
0x0040000a: <240> mov $0x78,%eax
--Type <RET> for more, q to quit, c to continue without paging--RET
0x0040000b: <247> cmpl $0x18,%eax(0x00)
0x0040000c: <250> je 0x00400009 <phase_3+257>
0x0040000d: <252> call 0x004000fc <explode_bomb>
0x0040000e: <257> mov -0x1(%eax),%eax
0x0040000f: <260> mov %eax,%eax
0x00400010: <262> pop %eax
0x00400011: <263> ret
End of assembler dump.
(gdb) disassemble phase_4
Dump of assembler code for function phase_4:
0x00400000: push %eax
0x00400001: mov %eax,%eax
0x00400002: sub $0x18,%eax
0x00400003: mov 0x0(%eax),%eax
0x00400004: add $0xffffffff,%eax
0x00400005: lea -0x1(%eax),%eax
0x00400006: push %eax
0x00400007: push $0x00000000
0x00400008: push %eax
0x00400009: call 0x00400000 <scan@plt>
0x0040000a: add $0x18,%eax
0x0040000b: cmpl $0x18,%eax
0x0040000c: jne 0x00400000 <phase_4+1>
0x0040000d: cmpl $0x0,%eax(0x00)
0x0040000e: jg 0x00400000 <phase_4+46>
0x0040000f: call 0x004000fc <explode_bomb>
0x00400010: add $0xffffffff,%eax
0x00400011: mov -0x1(%eax),%eax
0x00400012: push %eax
0x00400013: call 0x00400000 <func4>
0x00400014: add $0x18,%eax
0x00400015: add $0x18,%eax
0x00400016: cmpl $0x17,%eax
0x00400017: je 0x00400027 <phase_4+71>
0x00400018: call 0x004000fc <explode_bomb>
0x00400019: mov %eax,%eax
0x0040001a: pop %eax
0x0040001b: ret
End of assembler dump.
(gdb)
= kali:

```

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.

The screenshot shows a VMWare Remote Console window titled 'sullibret-kali'. The main terminal window displays assembly code for a function named 'func4'. The code includes instructions like 'push', 'call', 'add', 'cmp', 'jne', 'jg', 'call', 'add', 'mov', 'sub', 'push', 'pop', 'ret', and 'jmp'. A red box highlights the instruction 'jle 0x0040c0e0 <func4+0x15>'. To the right, a debugger window shows the 'Registers' tab with values for 'eax', 'ecx', 'edx', 'esp', 'ebp', 'esi', 'edi', 'eip', 'eflags', 'cs', 'ss', 'ds', 'es', and 'fs'. Below the registers, the 'Disassembly' tab shows the assembly code for the function 'func4' starting at address 0x0040c0e0. The code includes instructions like 'push', 'mov', 'sub', 'push', 'pop', 'mov', 'cmp', 'jne', 'call', 'add', 'mov', 'sub', 'push', 'pop', 'ret', and 'jmp'. A red box highlights the instruction 'jle 0x0040c0e0 <func4+0x15>'. At the bottom, a status bar shows 'kali:~' and 'kali:~'.

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.


```
File Actions Edit View Help
0x0040c0f0 <+21>: push $eax
0x0040c0f0 <+22>: call 0x00408060 <sscanf@plt>
0x0040c0f0 <+27>: add $0x10,%eax
0x0040c0f0 <+30>: cmp $0x1,%eax
0x0040c0f0 <+32>: jne 0x00408060 <phase_4+41>
0x0040c0f0 <+35>: cmpl $0x0,%eax
0x0040c0f0 <+39>: jg 0x00408060 <phase_4+46>
0x0040c0f0 <+41>: call 0x00408060 <explode_bomb>
0x0040c0f0 <+44>: add $0xffffffff,%eax
0x0040c0f0 <+49>: mov -0x4(%eax),%eax
0x0040c0f0 <+52>: push $eax
0x0040c0f0 <+53>: call 0x0040c0e6 <func4>
0x0040c0f0 <+58>: add $0x10,%eax
0x0040c0f0 <+61>: cmp $0x1,%eax
0x0040c0f0 <+64>: je 0x00408060 <phase_4+71>
0x0040c0f0 <+66>: call 0x00408060 <explode_bomb>
0x0040c0f0 <+71>: mov %eax,%eax
0x0040c0f0 <+73>: pop %eax
0x0040c0f0 <+74>: ret
End of assembler dump.
(gdb) disassemble func4
Dump of assembler code for function func4:
0x0040c0e6 <+0>: push $eax
0x0040c0e6 <+1>: mov %eax,%eax
0x0040c0e6 <+3>: sub $0x10,%eax
0x0040c0e6 <+6>: push $eax
0x0040c0e6 <+7>: push $eax
0x0040c0e6 <+8>: mov 0x0(%eax),%eax
0x0040c0e6 <+11>: cmp $0x1,%eax
0x0040c0e6 <+14>: jle 0x0040c0e6 <func4+48>
0x0040c0e6 <+16>: add $0xffffffff,%eax
0x0040c0e6 <+19>: lea -0x1(%eax),%eax
0x0040c0e6 <+22>: push $eax
0x0040c0e6 <+23>: call 0x0040c0e6 <func4>
0x0040c0e6 <+28>: mov %eax,%eax
0x0040c0e6 <+30>: add $0xffffffff,%eax
0x0040c0e6 <+33>: lea -0x2(%eax),%eax
0x0040c0e6 <+36>: push $eax
0x0040c0e6 <+37>: call 0x0040c0e6 <func4>
0x0040c0e6 <+42>: add %eax,%eax
0x0040c0e6 <+44>: jmp 0x0040c0d0 <func4+53>
0x0040c0e6 <+46>: mov %eax,%eax
0x0040c0e6 <+48>: mov $0x1,%eax
0x0040c0e6 <+53>: lea -0x10(%eax),%eax
0x0040c0e6 <+56>: pop %eax
0x0040c0e6 <+57>: pop %eax
0x0040c0e6 <+58>: mov %eax,%eax
0x0040c0e6 <+60>: pop %eax
0x0040c0e6 <+61>: ret
End of assembler dump.
(gdb)
= kali:x
```

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.


```
File Actions Edit View Help
Halfway there!
So you got that one. Try this one.
break phase_4

BOOM!!!
The bomb has blown up.
[Inferior 1 (process 1451699) exited with code 010]
(gdb) disassemble phase_5
Dump of assembler code for function phase_5:
0x0040c020 <+0>: push    %eax
0x0040c020 <+1>: mov     %eax, %eax
0x0040c020 <+3>: sub     $0x10, %eax
0x0040c020 <+6>: push    %eax
0x0040c020 <+7>: push    %eax
0x0040c020 <+8>: mov     0x0(%eax), %eax
0x0040c020 <+11>: add     $0xffffffff, %eax
0x0040c020 <+14>: push    %eax
0x0040c020 <+15>: call    0x0040c010 <string_length>
0x0040c020 <+20>: add     $0x10, %eax
0x0040c020 <+23>: cmp     $0x4, %eax
0x0040c020 <+26>: je      0x0040c04d <phase_5+33>
0x0040c020 <+28>: call    0x0040c0fc <explode_bomb>
0x0040c020 <+32>: xor     %eax, %eax
0x0040c020 <+35>: lea     -0x4(%eax), %eax
0x0040c020 <+38>: mov     $0x00000000, %eax
0x0040c020 <+43>: mov     (%eax, %eax, 1), %eax
0x0040c020 <+44>: and     $0x5, %eax
0x0040c020 <+48>: movsbl  %eax, %eax
0x0040c020 <+51>: mov     (%eax, %eax, 1), %eax
0x0040c020 <+54>: mov     %eax, (%eax, %eax, 1)
0x0040c020 <+57>: jnc     %eax, %eax
0x0040c020 <+58>: cmp     $0x5, %eax
0x0040c020 <+61>: jle     0x0040c057 <phase_5+43>
0x0040c020 <+62>: movb    %eax, %eax
0x0040c020 <+67>: add     $0xffffffff, %eax
0x0040c020 <+70>: push    $0x00000000
0x0040c020 <+75>: lea     -0x4(%eax), %eax
0x0040c020 <+78>: push    %eax
0x0040c020 <+79>: call    0x0040c030 <strings_not_equal>
0x0040c020 <+84>: add     $0x10, %eax
0x0040c020 <+87>: test    %eax, %eax
0x0040c020 <+89>: je      0x0040c08c <phase_5+96>
0x0040c020 <+91>: call    0x0040c0fc <explode_bomb>
0x0040c020 <+96>: lea     -0x10(%eax), %eax
0x0040c020 <+99>: pop     %eax
0x0040c020 <+100>: pop     %eax
0x0040c020 <+101>: mov     %eax, %eax
0x0040c020 <+103>: pop     %eax
0x0040c020 <+104>: ret
End of assembler dump.
(gdb)
= kali:x
```

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.

```
VMware Remote Console  File  Edit  View  Virtual Machine  Window  Help
sulibret-kali
Tue Oct 22 4:31 PM
To release your mouse press: Control-X

kali2.0.0 zsh kali
File Actions Edit View Help
Halfway there!
So you got that one. Try this one.
break phase_4
BOOM!!!
The bomb has blown up.
[Inferior 1 (process 1451699) exited with code 010]
(gdb) disassemble phase_5
Dump of assembler code for function phase_5:
0x00400200 <+0>: push %eax
0x00400201 <+1>: mov %eax,%eax
0x00400202 <+2>: sub $0x10,%eax
0x00400203 <+3>: push %eax
0x00400204 <+4>: push %eax
0x00400205 <+5>: push %eax
0x00400206 <+6>: mov 0x0(%eax),%eax
0x00400207 <+7>: add $0xffffffff,%eax
0x00400208 <+8>: push %eax
0x00400209 <+9>: call 0x00400100 <string_length>
0x0040020a <+10>: add $0x10,%eax
0x0040020b <+11>: cmp $0x4,%eax
0x0040020c <+12>: je 0x0040020d <phase_5+33>
0x0040020d <+13>: call 0x004001fc <explode_bomb>
0x0040020e <+14>: jmp $0x0
0x0040020f <+15>: lea -0x4(%eax),%ecx
0x00400210 <+16>: mov $0x00400200,%ecx
0x00400211 <+17>: mov (%eax,%ecx,1),%eax
0x00400212 <+18>: xor %eax,%eax
0x00400213 <+19>: mov $0x0040020d,%ecx
0x00400214 <+20>: movsl %eax,%eax
0x00400215 <+21>: mov (%eax,%eax,1),%eax
0x00400216 <+22>: mov $0x0040020d,%ecx
0x00400217 <+23>: mov (%eax,%ecx,1),%eax
0x00400218 <+24>: xor %eax,%eax
0x00400219 <+25>: cmp $0x5,%eax
0x0040021a <+26>: jle 0x0040021b <phase_5+43>
0x0040021b <+27>: movb $0x0,%eax
0x0040021c <+28>: mov $0x0,%eax
0x0040021d <+29>: add $0xffffffff,%eax
0x0040021e <+30>: push $0x00400200
0x0040021f <+31>: lea -0x4(%eax),%ecx
0x00400220 <+32>: call 0x00400100 <string_not_equal>
0x00400221 <+33>: add $0x10,%eax
0x00400222 <+34>: test %eax,%eax
0x00400223 <+35>: je 0x00400224 <phase_5+96>
0x00400224 <+36>: call 0x004001fc <explode_bomb>
0x00400225 <+37>: lea -0x10(%eax),%ecx
0x00400226 <+38>: pop %eax
0x00400227 <+39>: pop %eax
0x00400228 <+40>: mov %eax,%eax
0x00400229 <+41>: pop %eax
0x0040022a <+42>: mov %eax,%eax
0x0040022b <+43>: pop %eax
0x0040022c <+44>: ret
End of assembler dump.
(gdb) x/s
= kali:x

0x00400200 <+0>: push %eax
0x00400201 <+1>: mov %eax,%eax
0x00400202 <+2>: sub $0x10,%eax
0x00400203 <+3>: push %eax
0x00400204 <+4>: push %eax
0x00400205 <+5>: push %eax
0x00400206 <+6>: mov 0x0(%eax),%eax
0x00400207 <+7>: add $0xffffffff,%eax
0x00400208 <+8>: push %eax
0x00400209 <+9>: call 0x00400100 <string_length>
0x0040020a <+10>: add $0x10,%eax
0x0040020b <+11>: cmp $0x4,%eax
0x0040020c <+12>: je 0x0040020d <phase_5+33>
0x0040020d <+13>: call 0x004001fc <explode_bomb>
0x0040020e <+14>: jmp $0x0
0x0040020f <+15>: lea -0x4(%eax),%ecx
0x00400210 <+16>: mov $0x00400200,%ecx
0x00400211 <+17>: mov (%eax,%ecx,1),%eax
0x00400212 <+18>: xor %eax,%eax
0x00400213 <+19>: mov $0x0040020d,%ecx
0x00400214 <+20>: movsl %eax,%eax
0x00400215 <+21>: mov (%eax,%eax,1),%eax
0x00400216 <+22>: mov $0x0040020d,%ecx
0x00400217 <+23>: mov (%eax,%ecx,1),%eax
0x00400218 <+24>: xor %eax,%eax
0x00400219 <+25>: cmp $0x5,%eax
0x0040021a <+26>: jle 0x0040021b <phase_5+43>
0x0040021b <+27>: movb $0x0,%eax
0x0040021c <+28>: mov $0x0,%eax
0x0040021d <+29>: add $0xffffffff,%eax
0x0040021e <+30>: push $0x00400200
0x0040021f <+31>: lea -0x4(%eax),%ecx
0x00400220 <+32>: call 0x00400100 <string_not_equal>
0x00400221 <+33>: add $0x10,%eax
0x00400222 <+34>: test %eax,%eax
0x00400223 <+35>: je 0x00400224 <phase_5+96>
0x00400224 <+36>: call 0x004001fc <explode_bomb>
0x00400225 <+37>: lea -0x10(%eax),%ecx
0x00400226 <+38>: pop %eax
0x00400227 <+39>: pop %eax
0x00400228 <+40>: mov %eax,%eax
0x00400229 <+41>: pop %eax
0x0040022a <+42>: mov %eax,%eax
0x0040022b <+43>: pop %eax
0x0040022c <+44>: ret

Breakpoint 1, 0x00400200 in phase_4 ()
(gdb) c
Continuing.
So you got that one. Try this one.
laptop

Breakpoint 4, 0x0040021a in phase_5 ()
(gdb) stepi
0x0040021b in phase_5 ()
0x0040021c in phase_5 ()
(gdb)
= kali:x
```

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:

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, our 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 needs 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.

[illegible]

Shown in screenshot below in the bottom left corner:

