Big Bang Theory-master

Sheldon 1

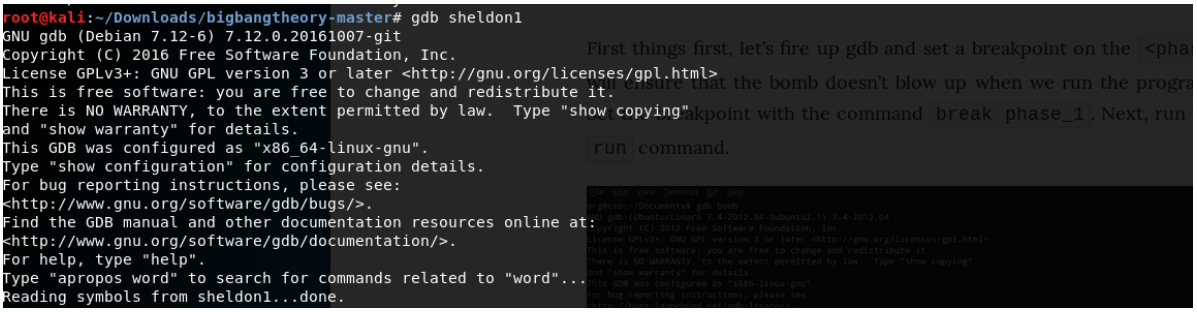
Introduction

First, download big bang theory from git hub and extract it.

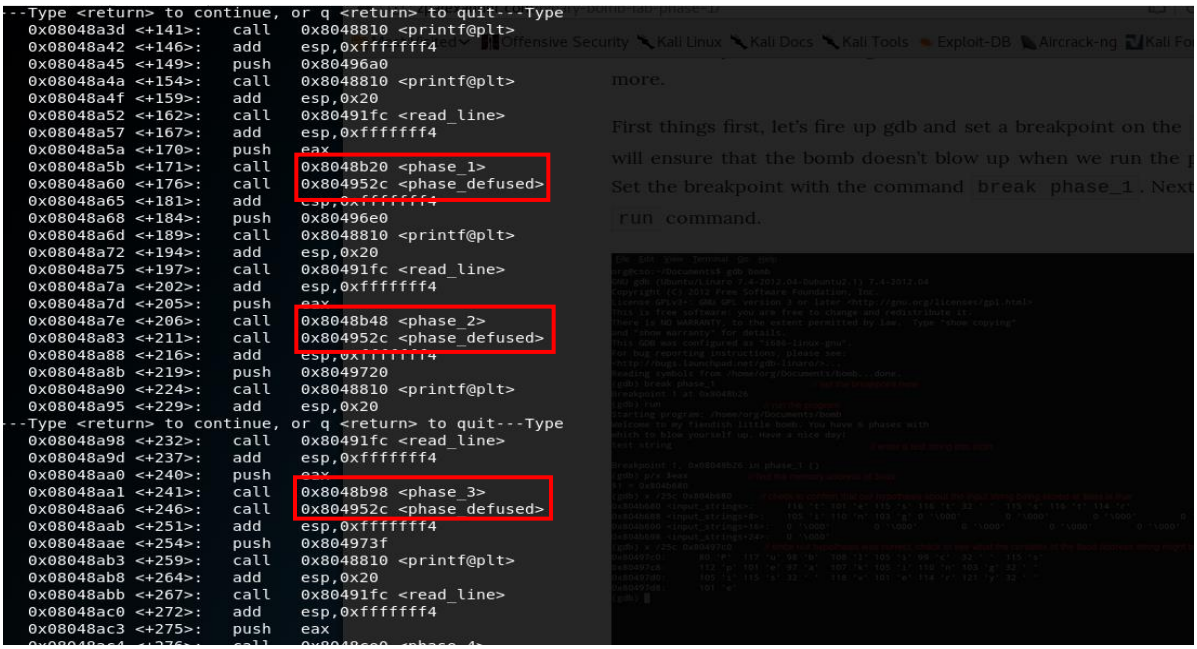
There are 4 files.

- learnord_win.exe
- README.md
- sheldon1
- sheldon2

Then launching the sheldon1 file under **gdb**



Then we can find 6 phases in that assembly code.



Now the challenge starts. You need to find the password to that phases.

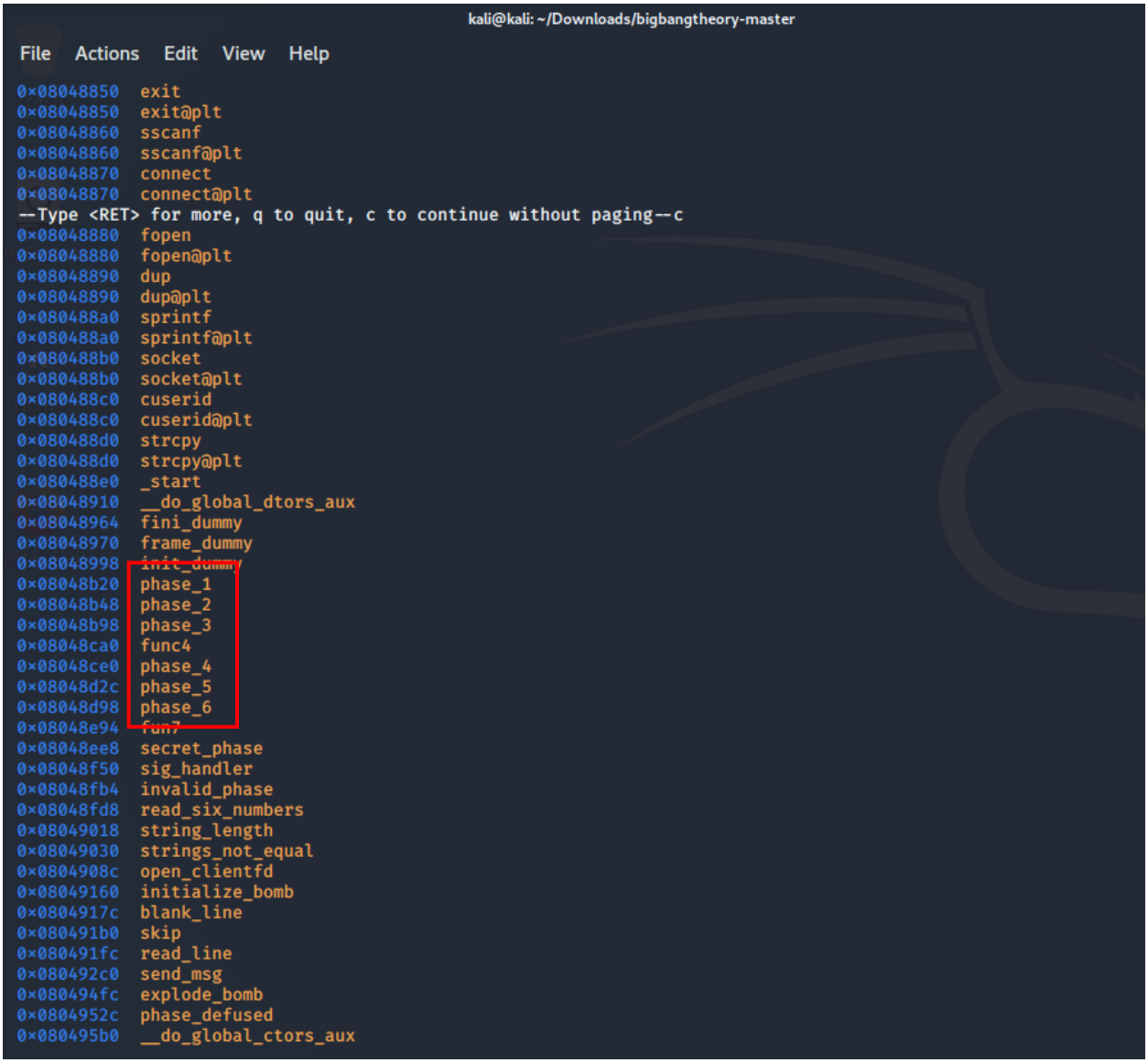
Phase 01

First, run the sheldon1 file using **chmod +x sheldon1** command and **./sheldon1** command. Then open it using **gdb**.

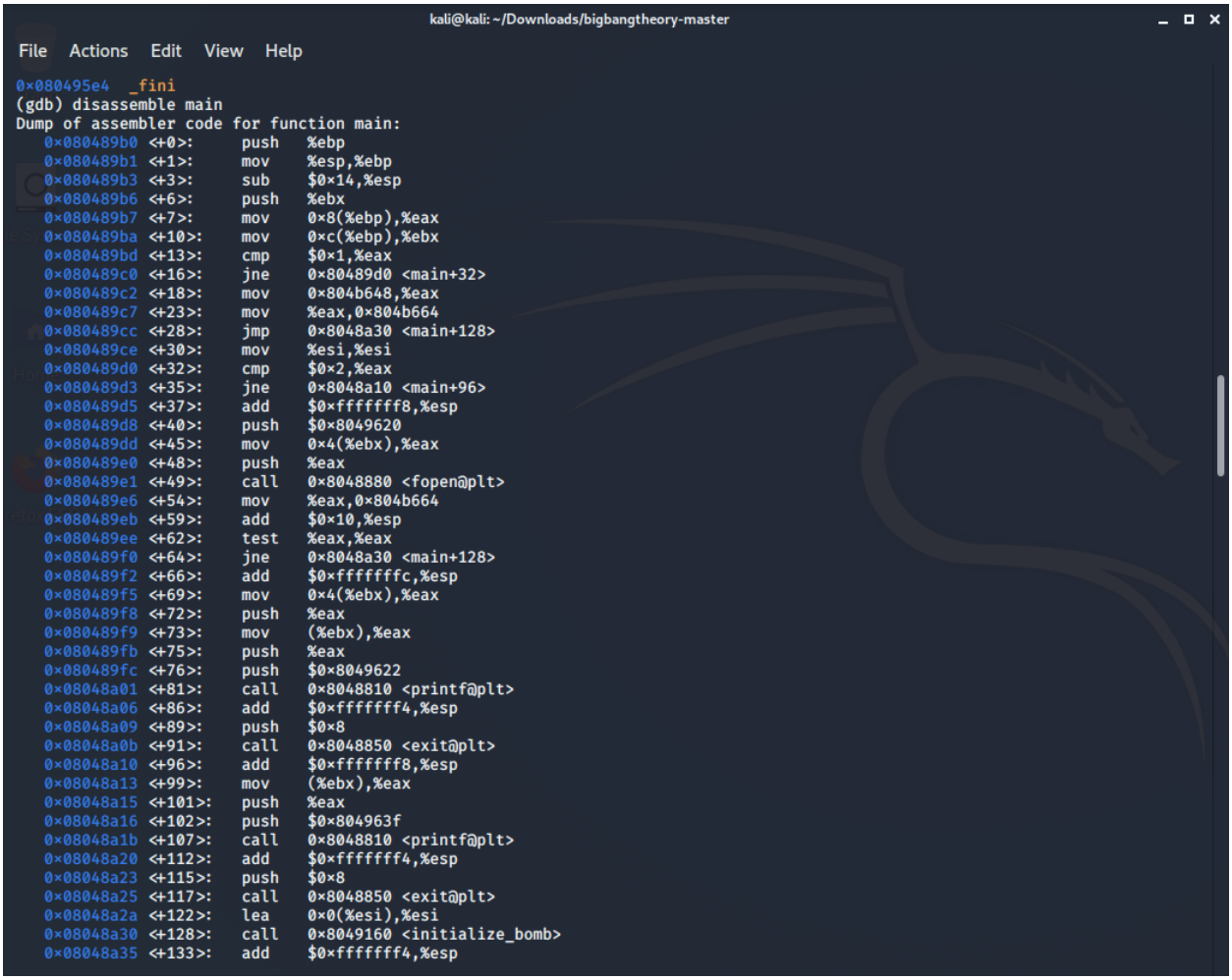
```
kali@kali:~/Downloads/bigbangtheory-master$ chmod +x sheldon1
kali@kali:~/Downloads/bigbangtheory-master$ ./sheldon1
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
^CSo you think you can stop the bomb with ctrl-c, do you?
Well ... OK. :-)
kali@kali:~/Downloads/bigbangtheory-master$ gdb sheldon1
GNU gdb (Debian 8.3.1-1) 8.3.1
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from sheldon1...
```

Look for the functions inside sheldon1 using the command “info functions”. As we can see there are separate functions for each phase there.

```
(gdb) info functions
All defined functions:

File bomb.c:
36:     int main(int, char **);

Non-debugging symbols:
0x080486e0  _init
0x08048720  __register_frame_info@plt
0x08048730  close@plt
0x08048740  fprintf@plt
0x08048750  tmpfile@plt
0x08048760  getenv@plt
0x08048770  signal
0x08048770  signal@plt
0x08048780  fflush
0x08048780  fflush@plt
0x08048790  bcopy
0x08048790  bcopy@plt
0x080487a0  rewind
0x080487a0  rewind@plt
0x080487b0  system
0x080487b0  system@plt
0x080487c0  __deregister_frame_info
0x080487c0  __deregister_frame_info@plt
0x080487d0  fgets
0x080487d0  fgets@plt
0x080487e0  sleep
0x080487e0  sleep@plt
0x080487f0  __strtol_internal
0x080487f0  __strtol_internal@plt
0x08048800  __libc_start_main
0x08048800  __libc_start_main@plt
0x08048810  printf
0x08048810  printf@plt
0x08048820  fclose
0x08048820  fclose@plt
```

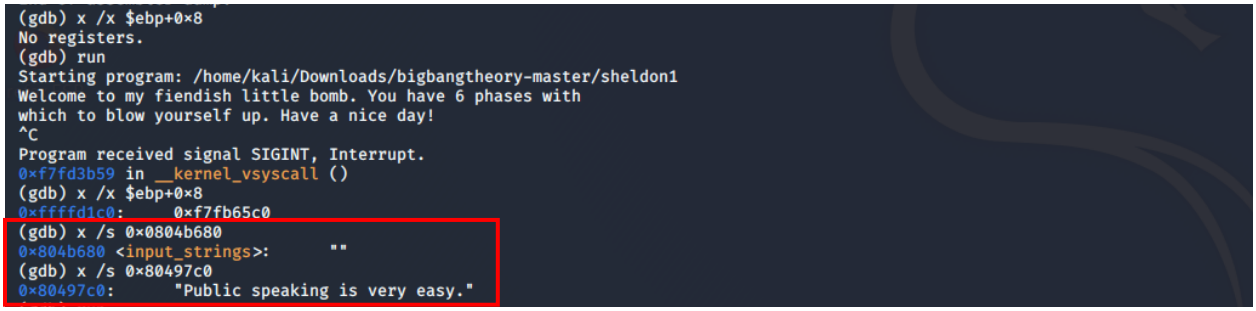


Then view the assembly code of the main function using **disassemble main** command.

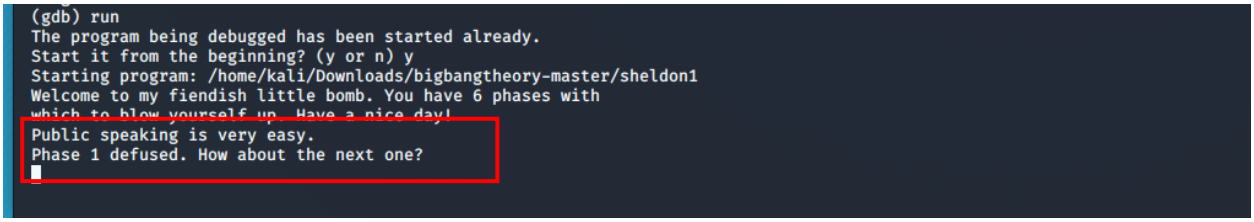


View the assembly code of the phase_1 using **disassemble phase_1** command.

In the first phase, there's a simple string comparison in place. Very straight-forward. The first string being compared is our input. The second string is the password for phase_1.

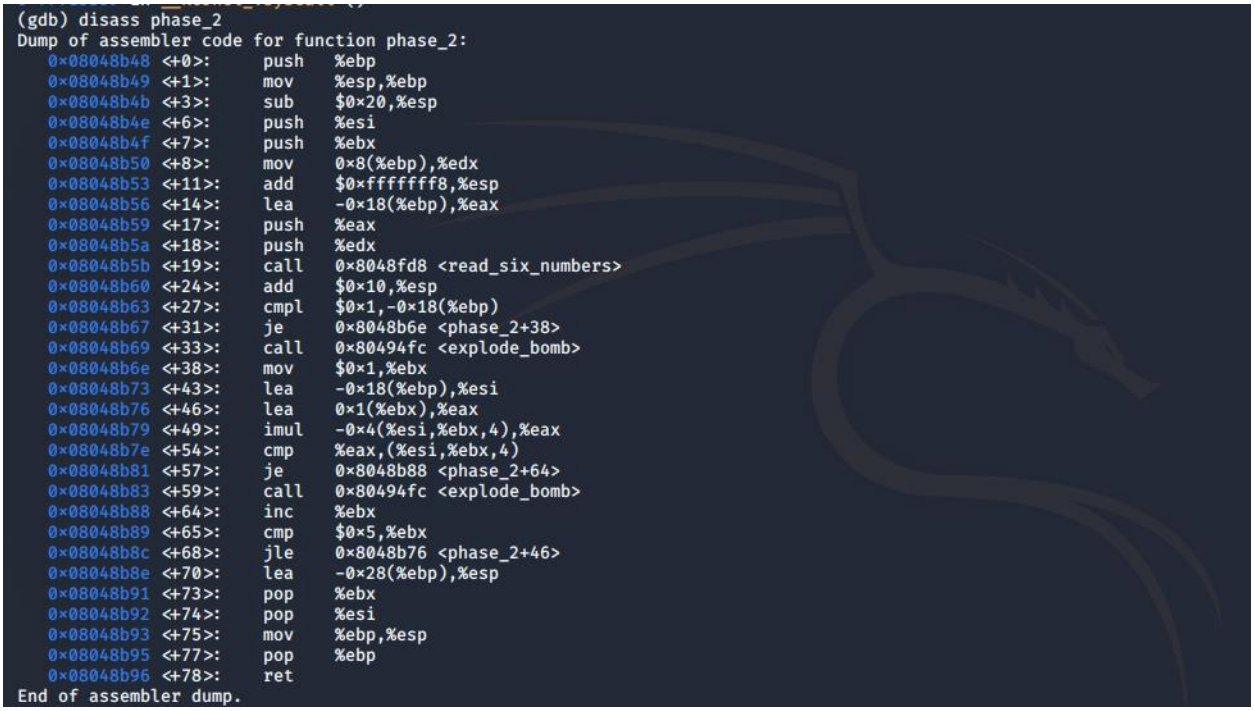


Password for phase 01: **Public speaking is very easy.**



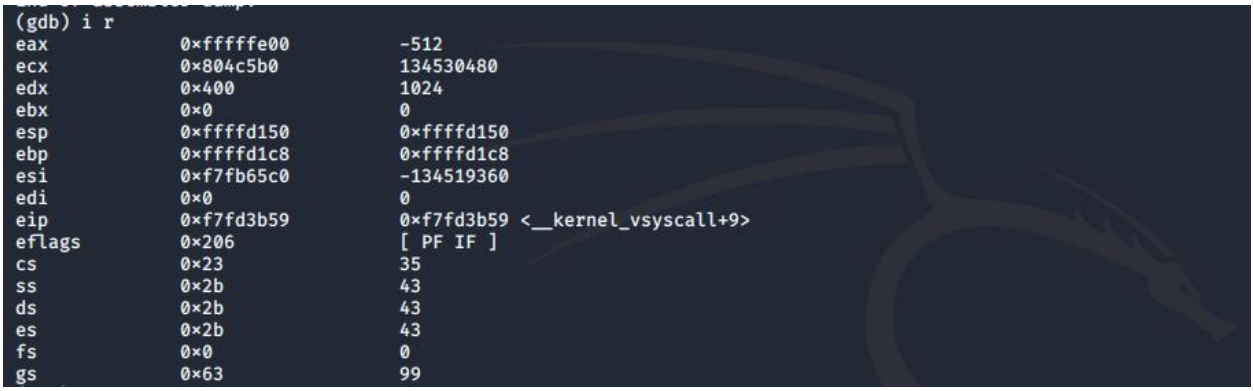
Phase 02

View the assembly code of the phase_2 using **disassemble phase_2** command.



```
(gdb) disass phase_2
Dump of assembler code for function phase_2:
   0x08048b48 <+0>:     push   %ebp
   0x08048b49 <+1>:     mov    %esp,%ebp
   0x08048b4b <+3>:     sub    $0x20,%esp
   0x08048b4e <+6>:     push   %esi
   0x08048b4f <+7>:     push   %ebx
   0x08048b50 <+8>:     mov    0x8(%ebp),%edx
   0x08048b53 <+11>:    add    $0xfffffff8,%esp
   0x08048b56 <+14>:    lea    -0x18(%ebp),%eax
   0x08048b59 <+17>:    push   %eax
   0x08048b5a <+18>:    push   %edx
   0x08048b5b <+19>:    call   0x8048fd8 <read_six_numbers>
   0x08048b60 <+24>:    add    $0x10,%esp
   0x08048b63 <+27>:    cmpl   $0x1,-0x18(%ebp)
   0x08048b67 <+31>:    je     0x8048b6e <phase_2+38>
   0x08048b69 <+33>:    call   0x80494fc <explode_bomb>
   0x08048b6e <+38>:    mov    $0x1,%ebx
   0x08048b73 <+43>:    lea    -0x18(%ebp),%esi
   0x08048b76 <+46>:    lea    0x1(%ebx),%eax
   0x08048b79 <+49>:    imul   -0x4(%esi,%ebx,4),%eax
   0x08048b7e <+54>:    cmp    %eax,(%esi,%ebx,4)
   0x08048b81 <+57>:    je     0x8048b88 <phase_2+64>
   0x08048b83 <+59>:    call   0x80494fc <explode_bomb>
   0x08048b88 <+64>:    inc    %ebx
   0x08048b89 <+65>:    cmp    $0x5,%ebx
   0x08048b8c <+68>:    jle    0x8048b76 <phase_2+46>
   0x08048b8e <+70>:    lea    -0x28(%ebp),%esp
   0x08048b91 <+73>:    pop    %ebx
   0x08048b92 <+74>:    pop    %esi
   0x08048b93 <+75>:    mov    %ebp,%esp
   0x08048b95 <+77>:    pop    %ebp
   0x08048b96 <+78>:    ret
End of assembler dump.
```

Using the **i r** command to see the contents of registers.



```
(gdb) i r
eax            0xfffffe00          -512
ecx            0x804c5b0           134530480
edx            0x400               1024
ebx            0x0                 0
esp            0xffffd150          0xffffd150
ebp            0xffffd1c8          0xffffd1c8
esi            0xf7fb65c0          -134519360
edi            0x0                 0
eip            0xf7fd3b59          0xf7fd3b59 <__kernel_vsyscall+9>
eflags         0x206               [ PF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

In assembly language basics a few things should jump out at you right away. Before anything else happens, a number is getting compared to 1, and the bomb is going off this number isn't also 1. This means that the first integer in our secret phrase is undoubtedly 1.

```
(gdb) i r eax
eax            0x1                 1
```

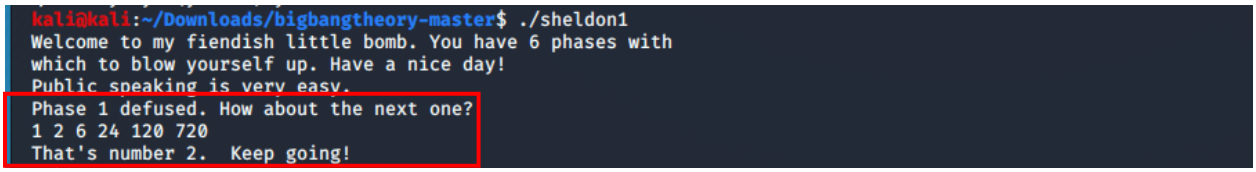
Once we've arrived at the comparison statement, we can use the **i r** command to see the contents of our registers. **eax**, which is the register to which our value is being compared, is equal to 2. Therefore, the second integer in our passphrase should be 2.

The simplest way to solve this level completely is by continuing to step through the code, seeing what **eax** is equal to after each iteration. Next, you'll find 6, then 24, then 120 followed by 720. The assembly code is implementing the following algorithm:

v[0] = 1

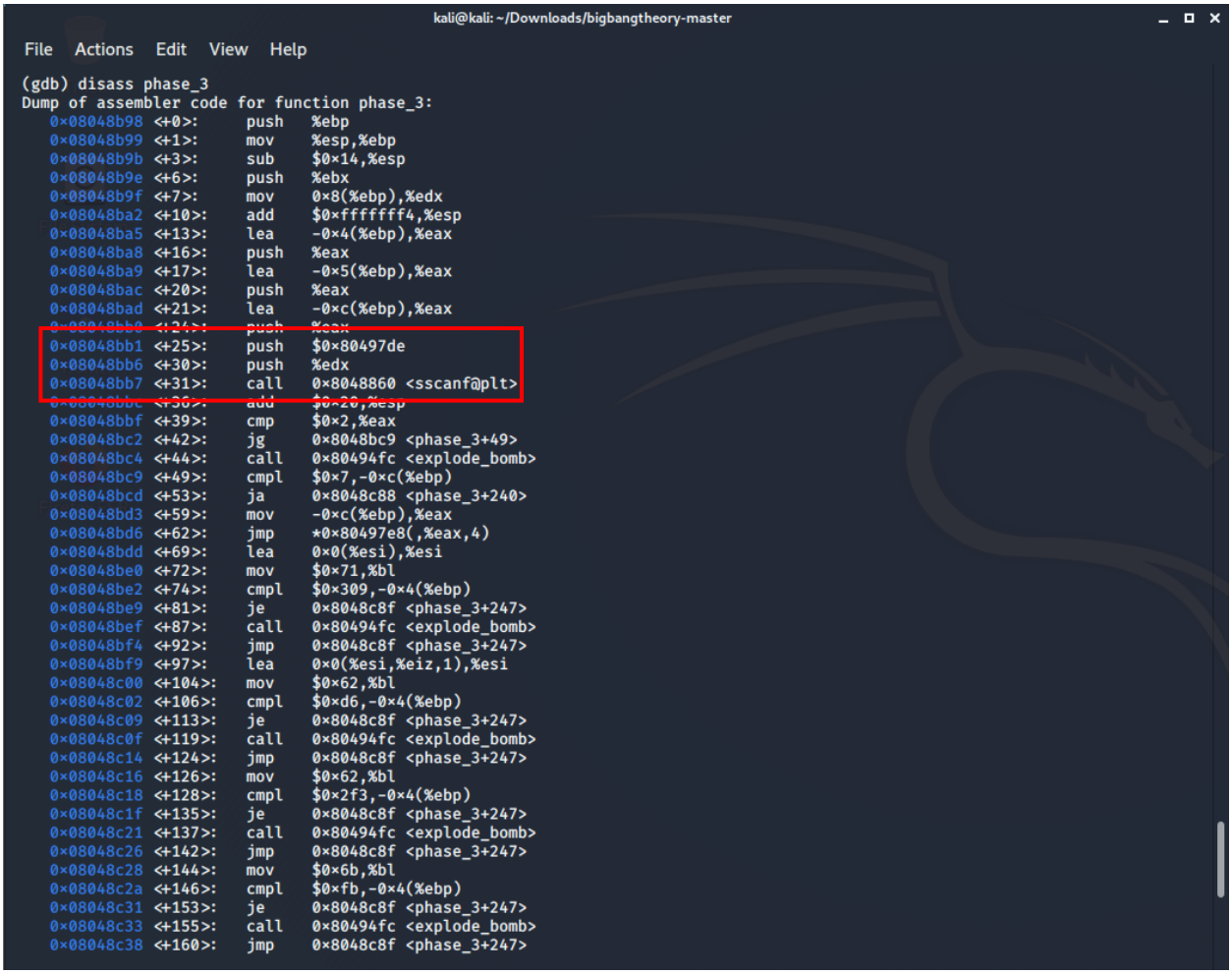
v[i] = (i+i) * v[i-1]

Password for phase 02 : **1 2 6 24 120 720**



Phase 03

View the assembly code of the phase_3 using **disassemble phase_3** command.



This expects as: “an integer”- “a character”- “an integer”

Input the passphrase as “1 a 7”. Results:

The first condition to not explode the bomb: we have to fill all the 3 variables passed to **sscanf**.

0x08048bbf <+39>: cmp eax,0x2
0x08048bc2 <+42>: jg 0x8048bc9 <phase_3+49>
0x08048bc4 <+44>: call 0x80494fc <explode_bomb>

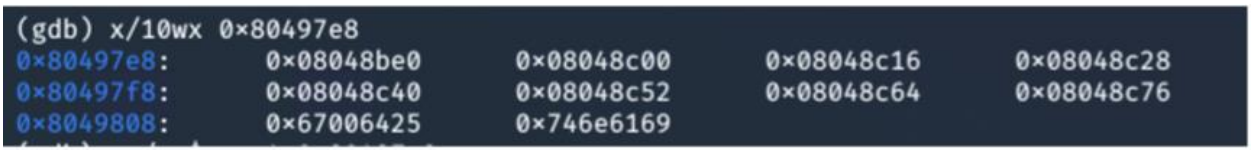
Second condition: the first number must be <=7

0x08048bc9 <+49>: cmp DWORD PTR [ebp-0xc],0x7
0x08048bcd <+53>: ja 0x8048c88 <phase_3+240>

The last part of the function looks like a case structure. We have the following table of addresses:

0x08048bd6 <+62>: jmp DWORD PTR [eax*4+0x80497e8]

In **eax,** we have the first number, which we chose as 7.

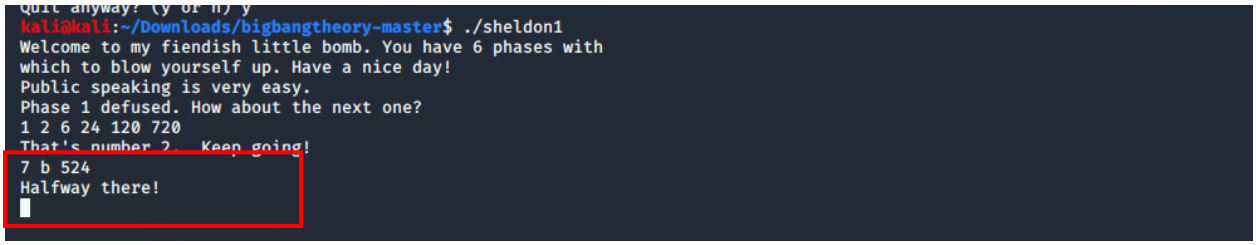


In our case, when the first parameter was 7, we'll jump to 0x08048c76.

(gdb) x /x $eax*4+0x80497e8
0x8049804: 0x08048c76

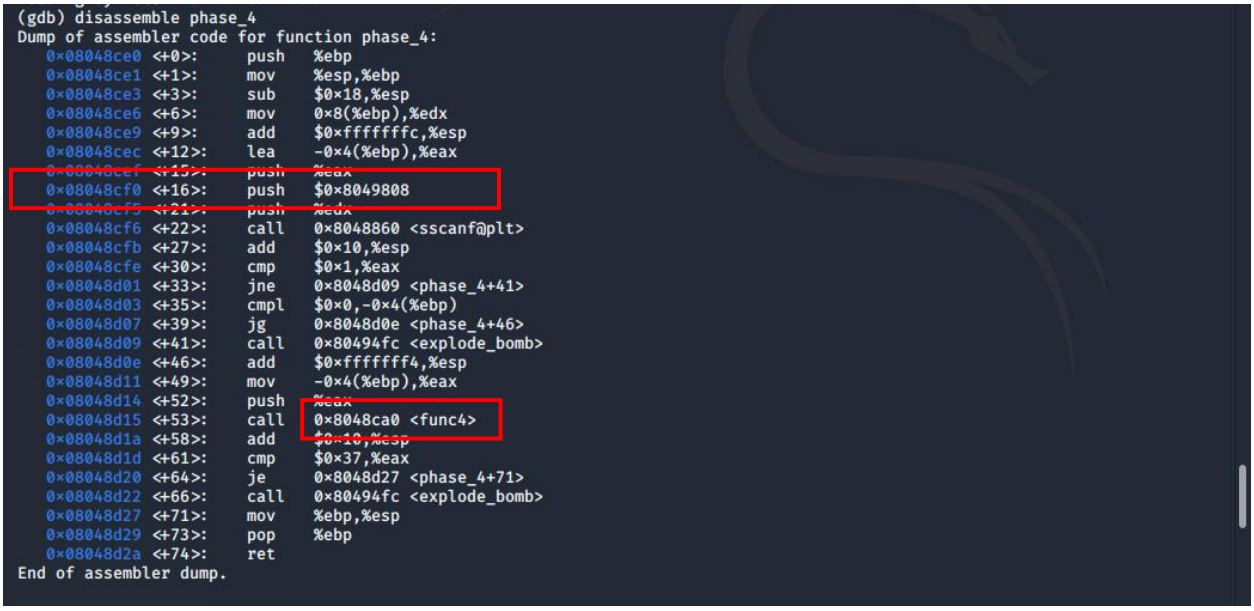
0x08048c76 <+222>: mov bl,0x62 …… ascii letter 'b'
0x08048c78 <+224>: cmp DWORD PTR [ebp-0x4],0x20c ……524 in decimal
0x08048c7f <+231>: je 0x8048c8f <phase_3+247>
0x08048c81 <+233>: call 0x80494fc <explode_bomb>
0x08048c86 <+238>: jmp 0x8048c8f <phase_3+247>

Password for phase 03: **7 b 524**

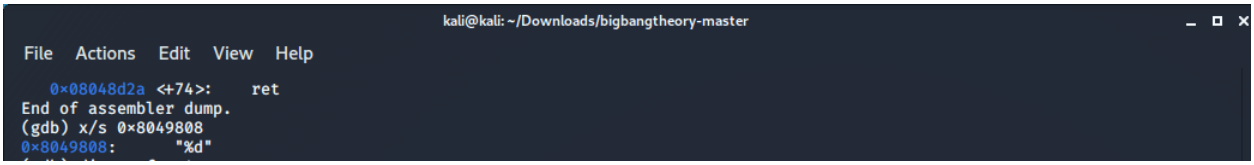


Phase 04

View the assembly code of the phase_4 using **disassemble phase_4** command.



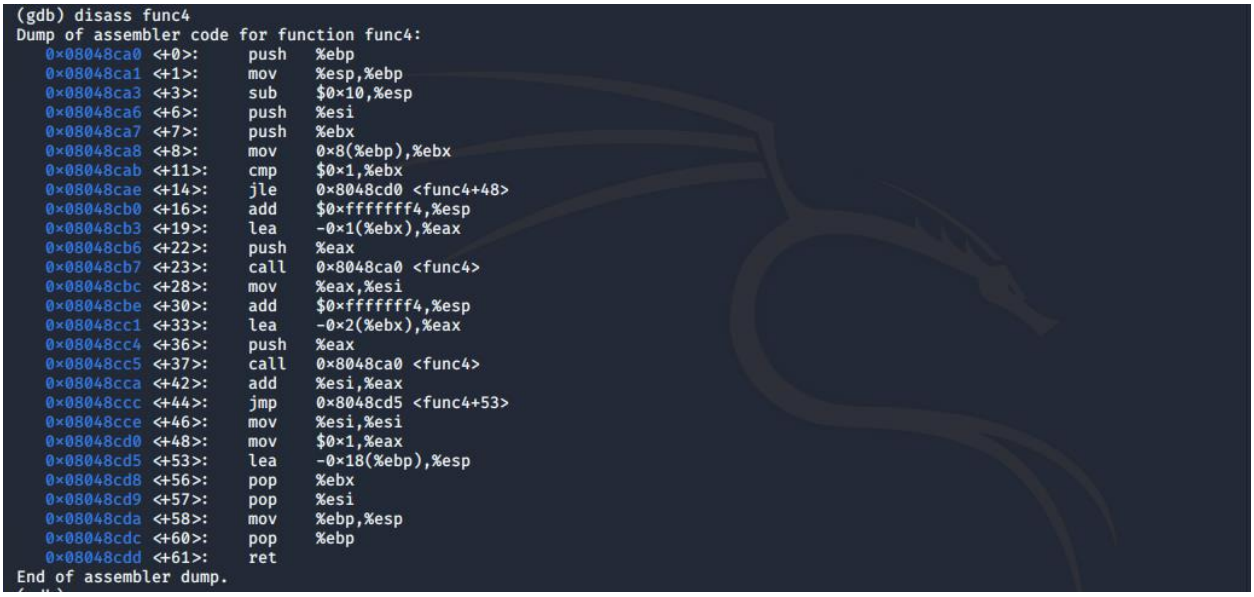
On line phase_4+16, the phase_4 function is pushing a fixed value stored at memory address 0x8049808 onto the stack right before a call to **scanf** is made. As we have learned from the past phases, fixed values are almost always important. Lo and behold, when we dump the contents of the memory address we get “%d”, which tells us that the answer to this phase should be a single integer.



The second important feature of this code occurs phase_4+53. Input value, which is stored in **eax**, is getting input into this other function called func4. Right now func4 is a black box, so we will need to dig into it using **si** to figure out what it is doing to our integer.

The last thing we need to consider is that, after <func4> gets called and returns our potentially altered input to **eax**, **eax** then gets compared to the hex value 37, which in decimal is 55. This means that, whatever our input is, to begin with, it needs to be turned into 55 by func4.

Now dig into func4 to figure out what we need to input to generate 55.



```
(gdb) disass func4
Dump of assembler code for function func4:
   0x08048ca0 <+0>:     push   %ebp
   0x08048ca1 <+1>:     mov    %esp,%ebp
   0x08048ca3 <+3>:     sub    $0x10,%esp
   0x08048ca6 <+6>:     push   %esi
   0x08048ca7 <+7>:     push   %ebx
   0x08048ca8 <+8>:     mov    0x8(%ebp),%ebx
   0x08048cab <+11>:    cmp    $0x1,%ebx
   0x08048cae <+14>:    jle    0x8048cd0 <func4+48>
   0x08048cb0 <+16>:    add    $0xfffffff4,%esp
   0x08048cb3 <+19>:    lea    -0x1(%ebx),%eax
   0x08048cb6 <+22>:    push   %eax
   0x08048cb7 <+23>:    call   0x8048ca0 <func4>
   0x08048cbc <+28>:    mov    %eax,%esi
   0x08048cbe <+30>:    add    $0xfffffff4,%esp
   0x08048cc1 <+33>:    lea    -0x2(%ebx),%eax
   0x08048cc4 <+36>:    push   %eax
   0x08048cc5 <+37>:    call   0x8048ca0 <func4>
   0x08048cca <+42>:    add    %esi,%eax
   0x08048ccc <+44>:    jmp    0x8048cd5 <func4+53>
   0x08048cce <+46>:    mov    %esi,%esi
   0x08048cd0 <+48>:    mov    $0x1,%eax
   0x08048cd5 <+53>:    lea    -0x18(%ebp),%esp
   0x08048cd8 <+56>:    pop    %ebx
   0x08048cd9 <+57>:    pop    %esi
   0x08048cda <+58>:    mov    %ebp,%esp
   0x08048cdc <+60>:    pop    %ebp
   0x08048cdd <+61>:    ret
End of assembler dump.
```

The next condition is that the func4 should return 0x37

0x08048d1d <+61>: cmp eax,0x37
0x08048d20 <+64>: je 0x8048d27 <phase_4+71>

In func4, if argument <=1, the return value is 0x1:

0x08048cab <+11>: cmp ebx,0x1
0x08048cae <+14>: jle 0x8048cd0 <func4+48>

0x08048cd0 <+48>: mov eax,0x1

Func4 :

func4(x):

if x <= 1 :

return 1

else :

y = func4(x-1)

z = func4(x-2)

return y + z

It's the Fibonacci function, implemented recursively. We quickly convert it to python code:

#!/usr/bin/python

```
def func4(x):
    if x <= 1 :
        return 1
    else :
        y = func4(x-1)
        z = func4(x-2)
        return y + z
if __name__ == "__main__":
    print func4(9)
```

We're expecting 0x37, which is 55 decimal, which is Fibonacci(9).

Password for phase 04: **9**