18-600: Recitation #3

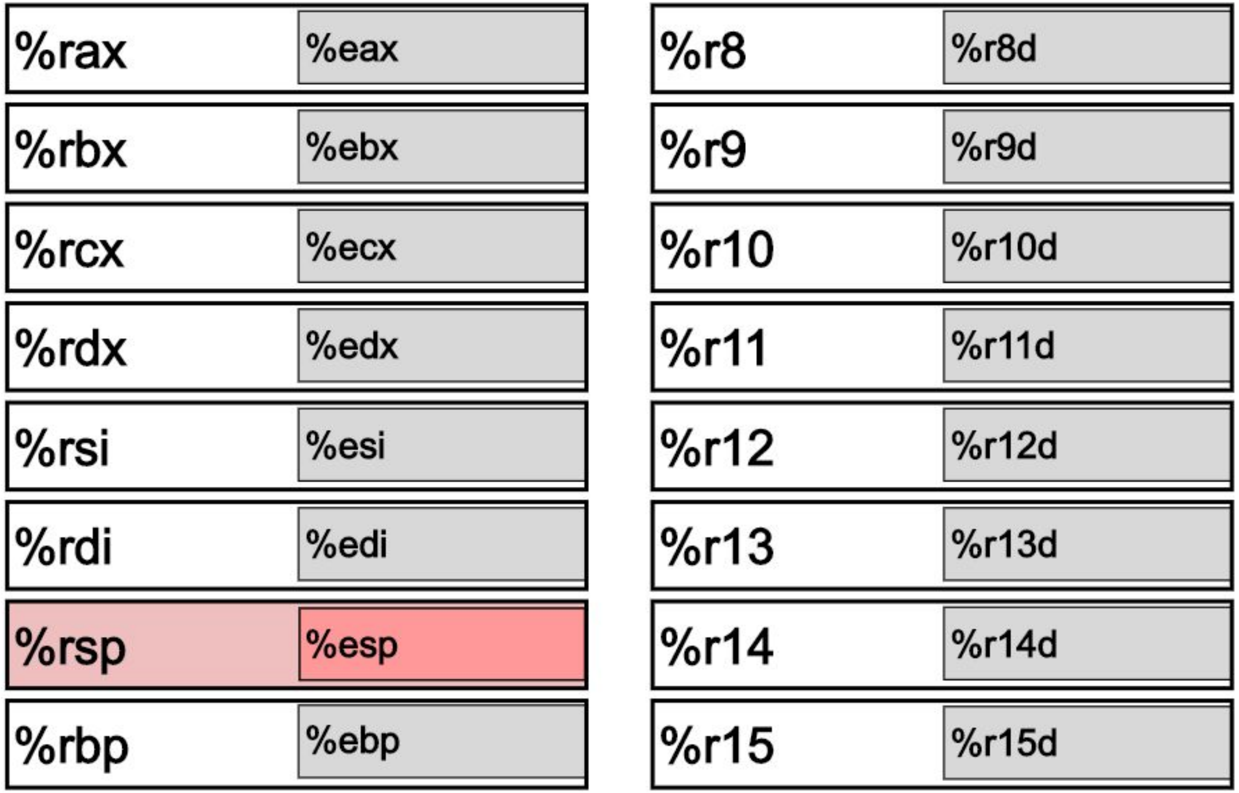
Bomb Lab & GDB Overview

September 12th, 2017

Today

- X86-64 Overview
- Bomb Lab Introduction
- GDB Tutorial

x86-64 Integer Registers



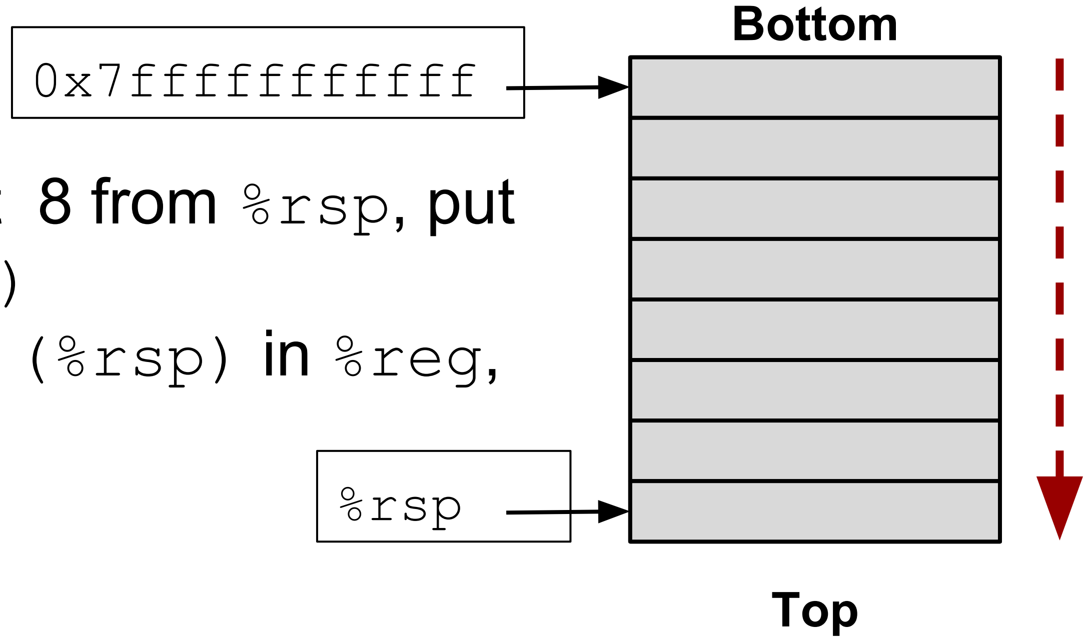
- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

x86-64: Register Conventions

- Arguments passed in registers:
  `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- Return value: `%rax`
- Callee-saved: `%rbx, %r12, %r13, %r14, %rbp, %rsp`
- Caller-saved: `%rdi, %rsi, %rdx, %rcx, %r8, %r9, %rax, %r10, %r11`
- Stack pointer: `%rsp`
- Instruction pointer: `%rip`

x86-64: The Stack

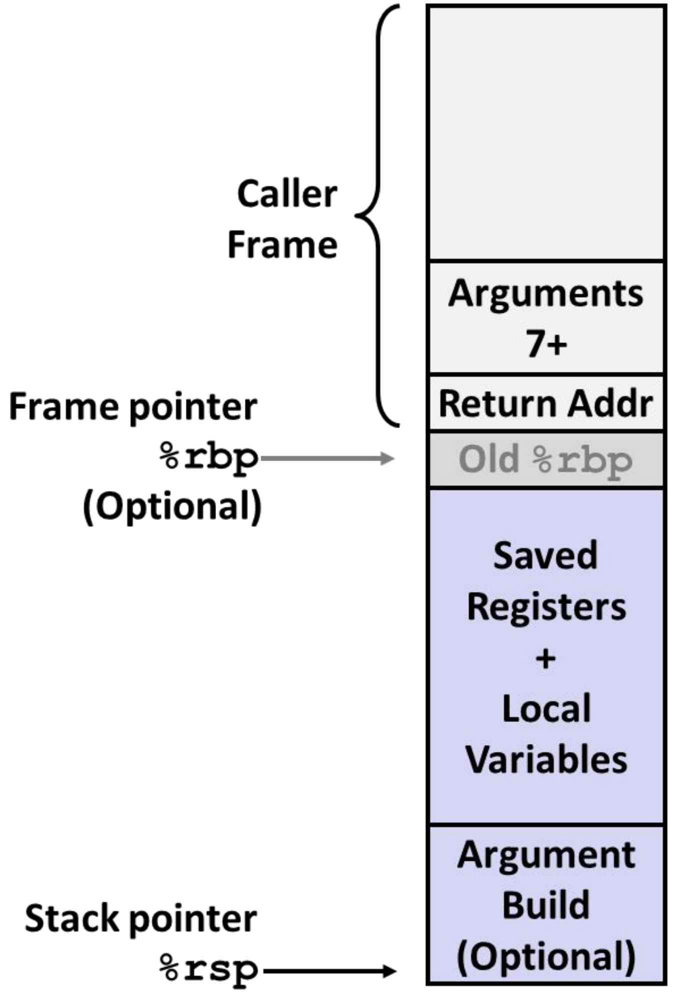
- Grows **downward** towards **lower** memory addresses
- `%rsp` points to **top** of stack



- `push %reg`: subtract 8 from `%rsp`, put val in `%reg` at `(%rsp)`
- `pop %reg`: put val at `(%rsp)` in `%reg`, add 8 to `%rsp`

x86-64: Stack Frames

- Every function call has its own **stack frame**.
- Think of a frame as a workspace for each call.
  - Local variables
  - Callee & Caller-saved registers
  - Optional arguments for a function call


Caller Frame
Arguments 7+
Return Addr
Frame pointer %rbp (Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build (Optional)
Stack pointer %rsp

x86-64: Function Call Setup

Caller:

- Allocates stack frame large enough for saved registers, optional arguments (when no. of arguments > 6)
- Save any caller-saved registers in frame
- Save any optional arguments (in **reverse order**) in frame
- `call foo`: push `%rip` to stack, jump to label `foo`

Callee:

- Push any callee-saved registers (sometimes local variables), decrease `%rsp` to make room for new frame

x86-64: Function Call Return

Callee:

- Increase `%rsp`, pop any callee-saved registers (in **reverse order**), execute `ret: pop %rip`

Question?

Do all functions require a stack frame?

Bomb Lab/GDB Overview

What are the different programming errors ?

What are the different programming errors ?

- **Compile time errors: Occur at the time of compilation**
  - Syntax errors: Rules of the programming language are violated
    - int a, b:
  - Semantic errors: Program statements are not meaningful to the compiler
    - b+c = a;

What are the different programming errors ?

- **Compile time errors: Occur at the time of compilation**
  - Syntax errors: Rules of the programming language are violated
    - int a, b:
  - Semantic errors: Program statements are not meaningful to the compiler
    - b+c = a
- **Runtime errors: Occur during the execution of the program**
  - Illegal operations:
    - Null pointer dereference
    - Illegal memory reference
    - Divide by zero
    - Out of memory
    - Opening non existent files

What are the different programming errors ?

- **Compile time errors: Occur at the time of compilation**
  - Syntax errors: Rules of the programming language are violated
    - int a, b:
  - Semantic errors: Program statements are not meaningful to the compiler
    - b+c = a
- **Runtime errors: Occur during the execution of the program**
  - Illegal operations:
    - Null pointer dereference
    - Illegal memory reference
    - Divide by zero
    - Out of memory
    - Opening non existent files
- **Logical errors: Occur due to unexpected output**
  - Incorrect assumptions about behavior of
    - programming language. Eg: implicit casting in c
    - variables. Eg: volatile vs auto vs static variables
    - functions: user defined, libraries. Eg: use of unsafe strcpy(), strcat() functions
  - Errors in arithmetic operations. Eg: overflow, truncation
  - Not protecting critical sections (more on this in later lectures)
  - Or merely incorrect logic

Debugging

“ There’s one wolf in Alaska, how do you find it? “

What is Debugging ?

What is Debugging ?

- Identifying the problem

What is Debugging ?

- Identifying the problem
- Isolating the source of the problem

What is Debugging ?

- Identifying the problem
- Isolating the source of the problem
- Fixing the problem

What is Debugging ?

Debuggers help here!

- Identifying the problem
- Isolating the source of the problem
- Fixing the problem

Commonly used Debugging Methods

Commonly used Debugging Methods

- Using "printf" in different parts of the program
- Test programs each time more complexity is added
- Have checkers to ensure guarantees at entry and exit of each function. You will do this in malloc lab
- Test incrementally: Use simple to more complex tests
- Use software tools
  - gdb: Program debugger
  - valgrind: Memory debugger
  - objdump -d: Disassembles object file

What is a debugger ?

- **Program that allows you to see what a program is doing while it executes**
- **Program that also allows you to observe program state when it crashed**
- **A good debugger must allow:**
  - Start and stop programs arbitrarily
  - Controlled stepping through a program
  - Enable examining code and data
  - Maintain history of a program run and print useful information about it
  - GDB is a great example of a good debugger!

GDB: Program debugger

- GNU debugger - GDB is the standard debugger for Unix like operating systems
- It is used to debug programs written in Ada, C, C++, Java, Objective-C, Pascal
- GDB can help you in finding memory leakage related bugs but not a tool to detect memory leakages

GDB Commands

Controlling Execution: step, next, break, run

Getting Info: print, info locals, up/down, list, backtrace

Getting started with using GDB

1. Compiling the program: You have to tell your compiler to compile your code with symbolic debugging information included. Here's how to do it with gcc, with the -g switch:

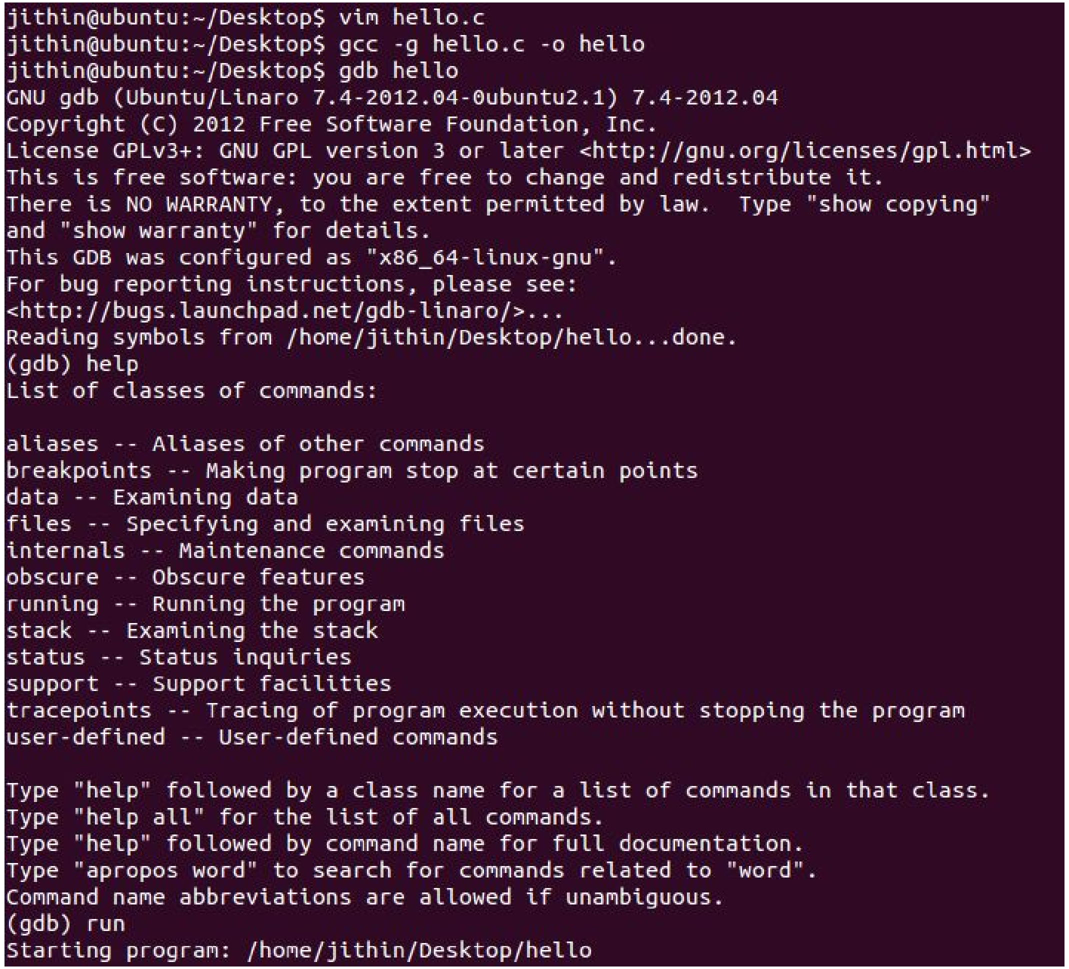
   gcc -g hello.c -o hello

2. Don't use compiler optimizations (-O, -O2…….)

3. Run gdb on the executable

   gdb hello

4. Type 'help' to see how to use gdb



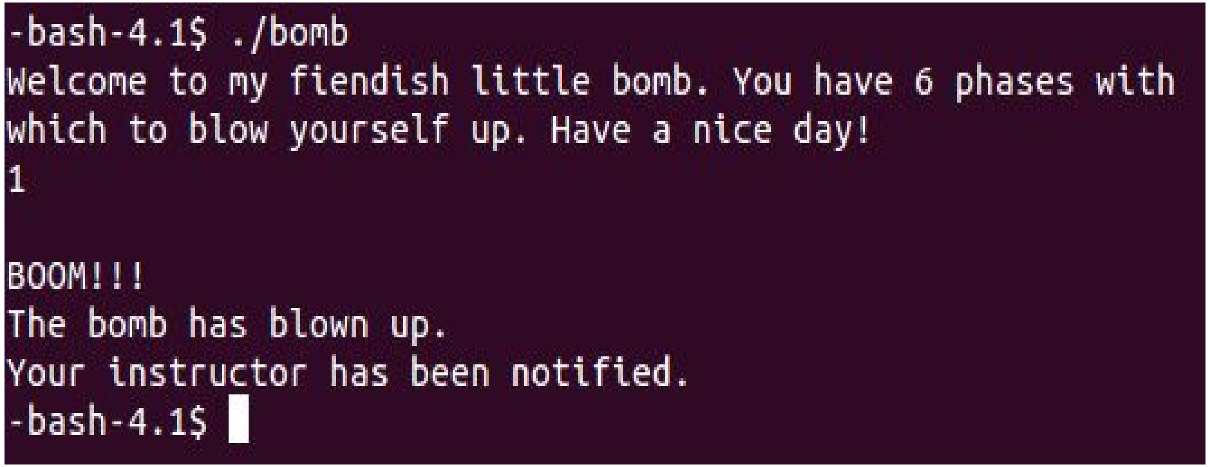
```
jithin@ubuntu:~/Desktop$ vim hello.c
jithin@ubuntu:~/Desktop$ gcc -g hello.c -o hello
jithin@ubuntu:~/Desktop$ gdb hello
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/jithin/Desktop/hello...done.
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) run
Starting program: /home/jithin/Desktop/hello
```

Example Program: The binary bomb !

- The nefarious Dr. Evil has planted a slew of “binary bombs” on our 64-bit shark machines.
- A binary bomb is a program that consists of a sequence of phases. Each phase expects you to type a particular string on stdin.
- If you type the correct string, then the phase is defused and the bomb proceeds to the next phase.
- Otherwise, the bomb explodes by printing "BOOM!!!" and then terminating. The bomb is defused when every phase has been defused.
- Our mission is to defuse the bomb.
- Remember that we do not have the source code of the bomb. But we do know that each phase is a function with prefix ‘phase_’ and appended with the phase number
- Our simple bomb has six phases, we will diffuse one in this class :)

Phase 1

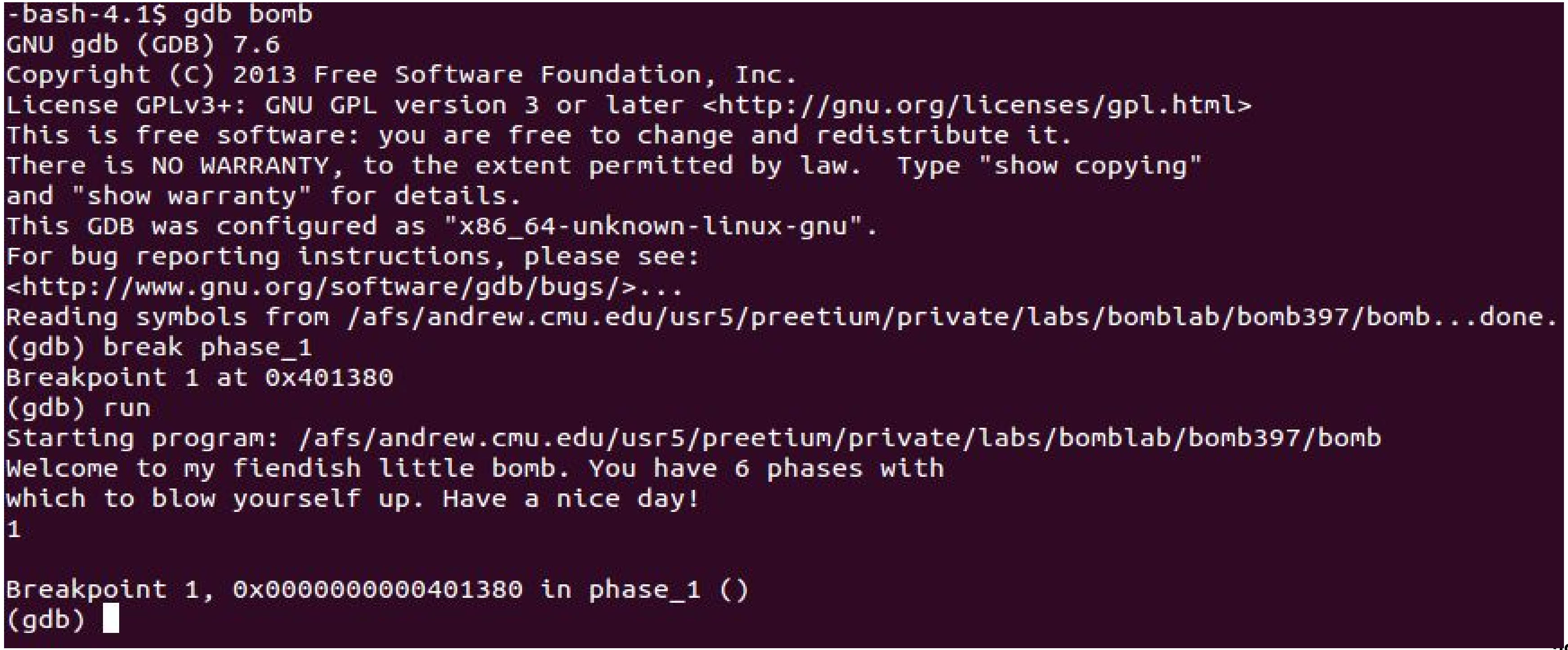
Oops!



```
-bash-4.1$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
1

BOOM!!!
The bomb has blown up.
Your instructor has been notified.
-bash-4.1$ 
```

GDB to the rescue!

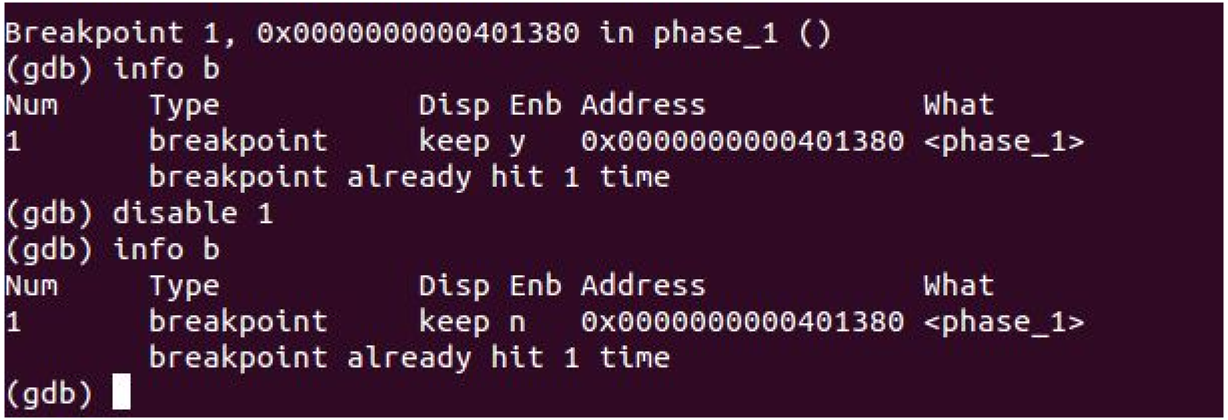
- **We know that the function is called phase_1 (see bomb.c). Let's 'break' at that.**



```
-bash-4.1$ gdb bomb
GNU gdb (GDB) 7.6
Copyright (C) 2013 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-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /afs/andrew.cmu.edu/usr5/preetium/private/labs/bomblab/bomb397/bomb...done.
(gdb) break phase_1
Breakpoint 1 at 0x401380
(gdb) run
Starting program: /afs/andrew.cmu.edu/usr5/preetium/private/labs/bomblab/bomb397/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
1

Breakpoint 1, 0x0000000000401380 in phase_1 ()
(gdb)
```

GDB: Breakpoints

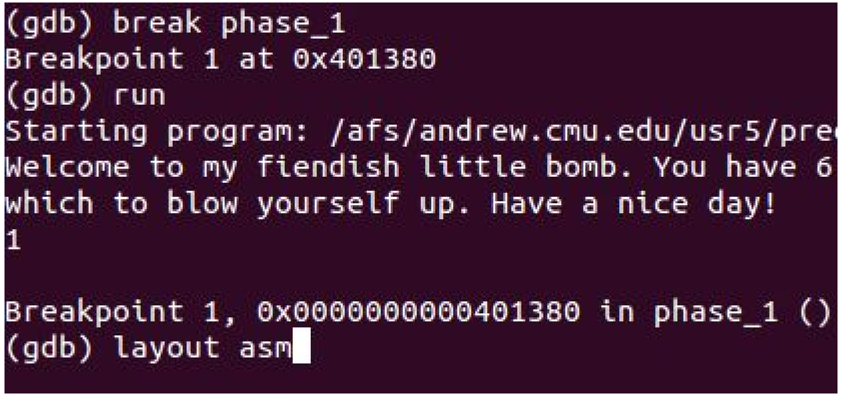
- **Breakpoints are set for specific lines in the code**
- **Running programs always stop at a breakpoint and hand you control**
- **Breakpoints can be set in any of the following ways:**
  - break main - break at the beginning of main()
  - break 50 - break at the 50th line in the executable
  - break hello.c:50 - break at the 50th line in hello.c
- **You can list the current break points and enable/disable break points**



```
Breakpoint 1, 0x0000000000401380 in phase_1 ()
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000401380 <phase_1>
        breakpoint already hit 1 time
(gdb) disable 1
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x0000000000401380 <phase_1>
        breakpoint already hit 1 time
(gdb) 
```

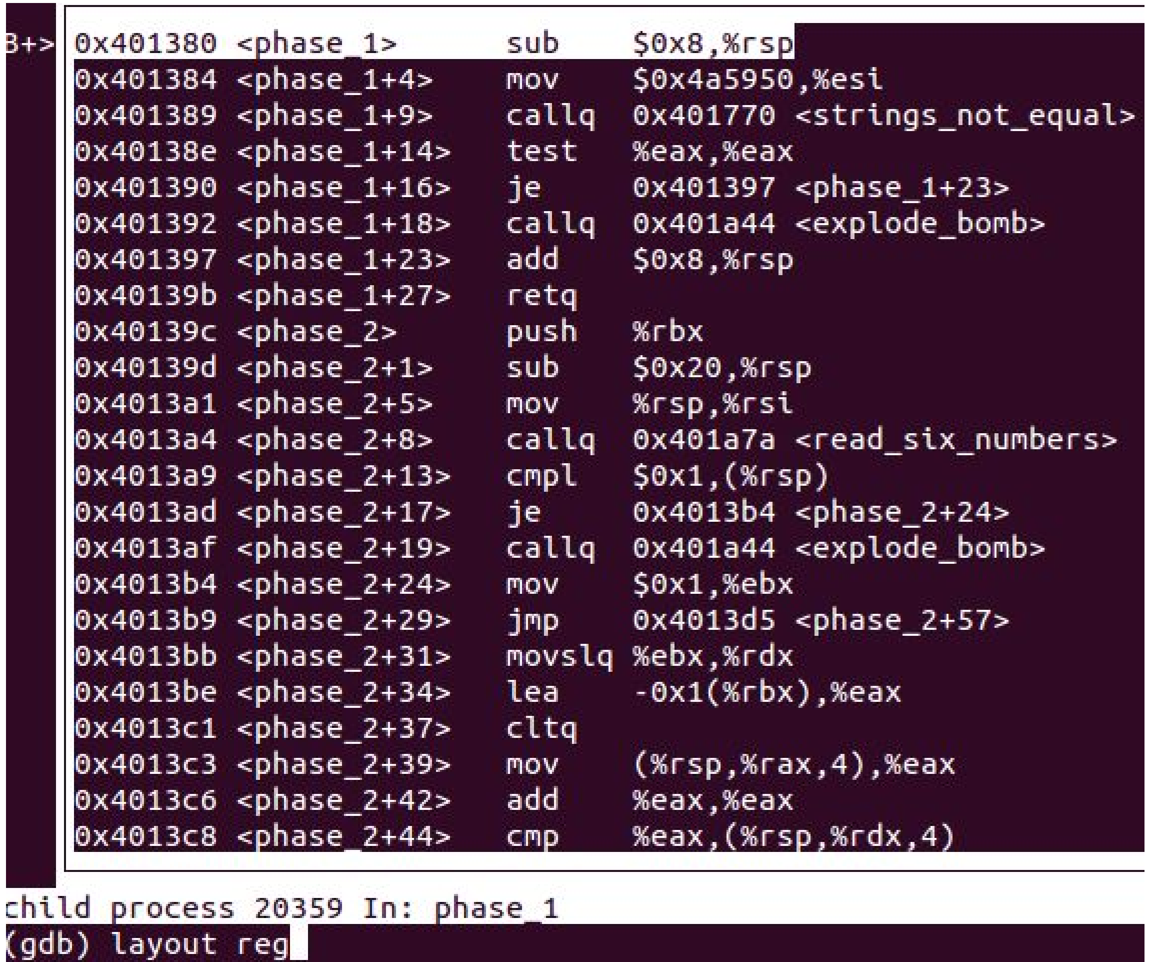
GDB: Layouts

- **‘layout’ command specifies which windows you see**
  - layout asm: Standard layout, assembly window on top, command window on the bottom
  - layout src: Same as previous, but source code window on top (**NOT AVAILABLE FOR THIS LAB**)
  - layout reg: Opens the register window on top of either source or assembly, whichever was opened last
  - layout prev/next: Navigate between layouts
- **‘layout’ command is useful when you want to parallely observe your code**



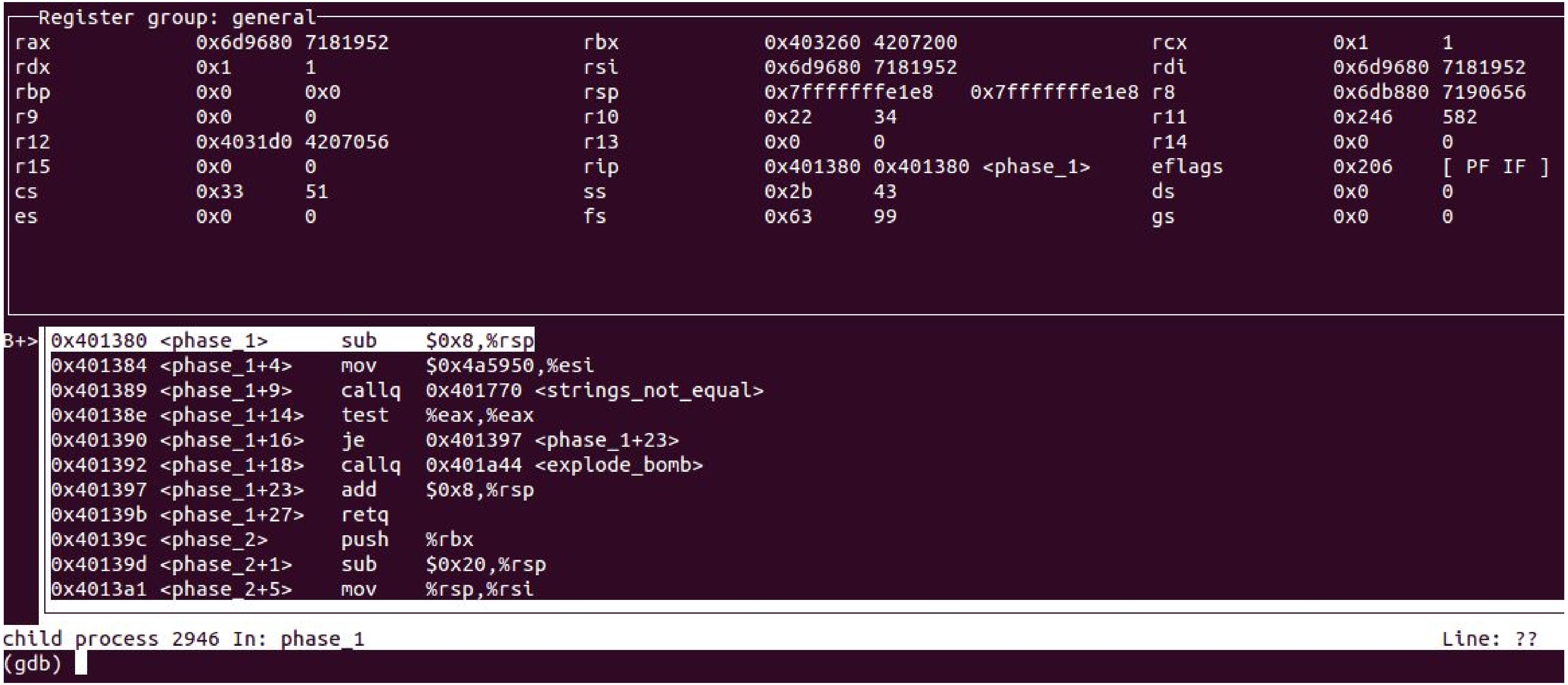
```
(gdb) break phase_1
Breakpoint 1 at 0x401380
(gdb) run
Starting program: /afs/andrew.cmu.edu/usr5/pre
Welcome to my fiendish little bomb. You have 6
which to blow yourself up. Have a nice day!
1

Breakpoint 1, 0x0000000000401380 in phase_1 ()
(gdb) layout asm
```

Result of ‘layout asm’



```
B+> 0x401380 <phase_1>       sub    $0x8,%rsp
    0x401384 <phase_1+4>     mov    $0x4a5950,%esi
    0x401389 <phase_1+9>     callq  0x401770 <strings_not_equal>
    0x40138e <phase_1+14>    test   %eax,%eax
    0x401390 <phase_1+16>    je     0x401397 <phase_1+23>
    0x401392 <phase_1+18>    callq  0x401a44 <explode_bomb>
    0x401397 <phase_1+23>    add    $0x8,%rsp
    0x40139b <phase_1+27>    retq
    0x40139c <phase_2>       push   %rbx
    0x40139d <phase_2+1>     sub    $0x20,%rsp
    0x4013a1 <phase_2+5>     mov    %rsp,%rsi
    0x4013a4 <phase_2+8>     callq  0x401a7a <read_six_numbers>
    0x4013a9 <phase_2+13>    cmpl   $0x1,(%rsp)
    0x4013ad <phase_2+17>    je     0x4013b4 <phase_2+24>
    0x4013af <phase_2+19>    callq  0x401a44 <explode_bomb>
    0x4013b4 <phase_2+24>    mov    $0x1,%ebx
    0x4013b9 <phase_2+29>    jmp    0x4013d5 <phase_2+57>
    0x4013bb <phase_2+31>    movslq %ebx,%rdx
    0x4013be <phase_2+34>    lea    -0x1(%rbx),%eax
    0x4013c1 <phase_2+37>    cltq
    0x4013c3 <phase_2+39>    mov    (%rsp,%rax,4),%eax
    0x4013c6 <phase_2+42>    add    %eax,%eax
    0x4013c8 <phase_2+44>    cmp    %eax,(%rsp,%rdx,4)

child process 20359 In: phase_1
(gdb) layout reg
```

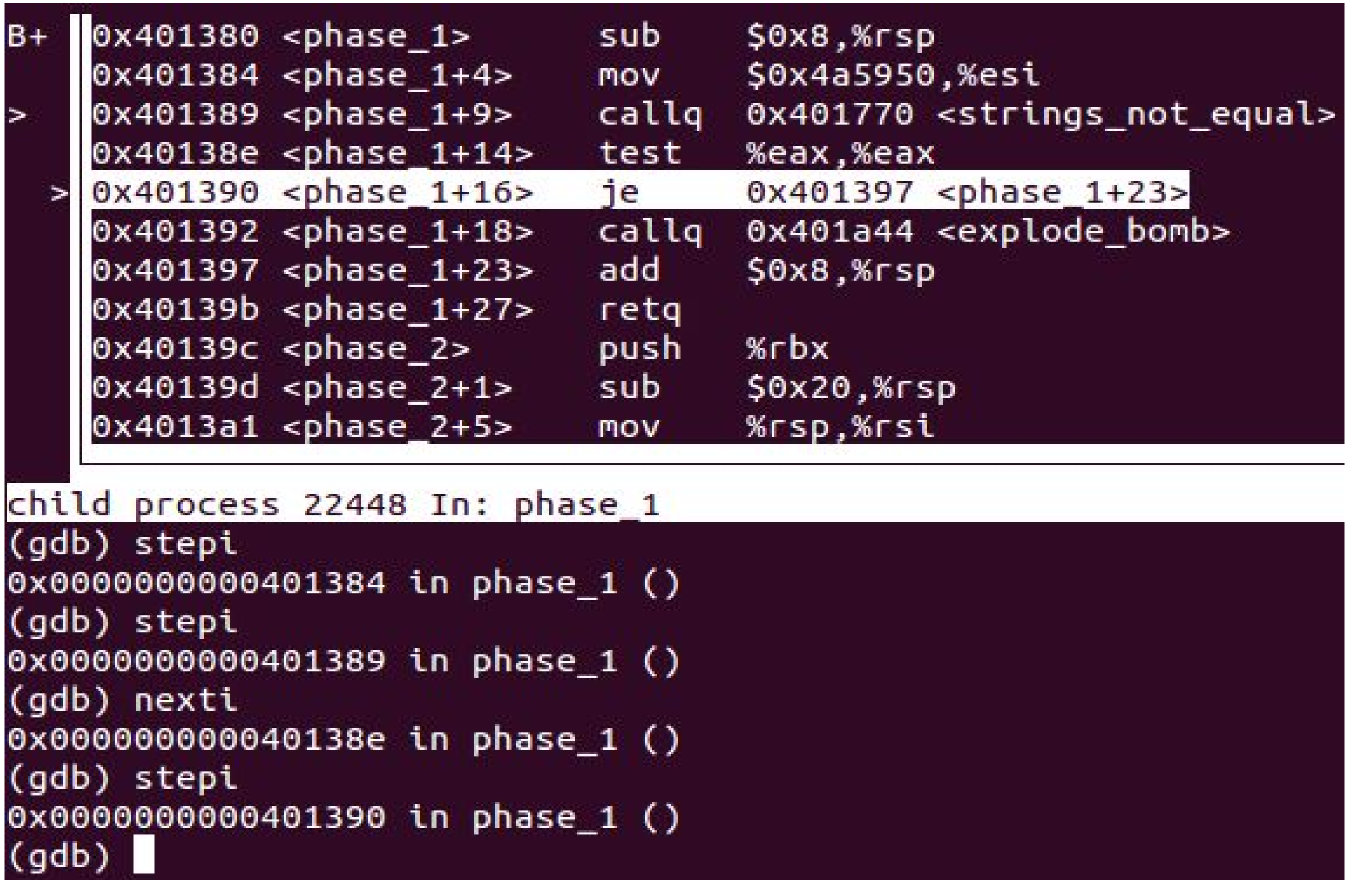
Result of ‘layout reg’



```
┌─Register group: general──────────────────────────────────────────────────────────────────────────┐
│rax            0x6d9680 7181952          rbx            0x403260 4207200          rcx            0x1      1
│rdx            0x1      1                rsi            0x6d9680 7181952          rdi            0x6d9680 7181952
│rbp            0x0      0x0              rsp            0x7fffffffe1e8   0x7fffffffe1e8 r8       0x6db880 7190656
│r9             0x0      0                r10            0x22     34               r11            0x246    582
│r12            0x4031d0 4207056          r13            0x0      0                r14            0x0      0
│r15            0x0      0                rip            0x401380 0x401380 <phase_1>    eflags    0x206    [ PF IF ]
│cs             0x33     51               ss             0x2b     43               ds             0x0      0
│es             0x0      0                fs             0x63     99               gs             0x0      0

B+>│0x401380 <phase_1>      sub    $0x8,%rsp
   │0x401384 <phase_1+4>    mov    $0x4a5950,%esi
   │0x401389 <phase_1+9>    callq  0x401770 <strings_not_equal>
   │0x40138e <phase_1+14>   test   %eax,%eax
   │0x401390 <phase_1+16>   je     0x401397 <phase_1+23>
   │0x401392 <phase_1+18>   callq  0x401a44 <explode_bomb>
   │0x401397 <phase_1+23>   add    $0x8,%rsp
   │0x40139b <phase_1+27>   retq
   │0x40139c <phase_2>      push   %rbx
   │0x40139d <phase_2+1>    sub    $0x20,%rsp
   │0x4013a1 <phase_2+5>    mov    %rsp,%rsi
child process 2946 In: phase_1                                                              Line: ??
(gdb)
```

Stepping around

- **Stepping through source code**
  - gcc -g hello.c: Compiles with line number information (Can also step through assembly)
  - step: Moves to the next line in the current program: steps ‘into’ function calls
  - step n: Move n lines from the current position: ‘n’ includes lines from inside function calls
  - next: Moves to the next line in the current program: steps ‘over’ function calls
  - next n: Move n lines from the current position: ‘n’ excludes lines having function calls
- **Stepping through assembly code (RECOMMENDED)**
  - gcc hello.c: Compiles ‘without’ line number information (Cannot step through source code)
  - stepi: Moves to the next assembly level instruction: steps ‘into’ function calls
  - stepi n: Execute next n instructions: includes instructions from inside function calls
  - nexti: Moves to the next assembly level instruction: steps ‘over’ function calls
  - nexti n: Execute next n instructions: steps over ‘call’ instructions



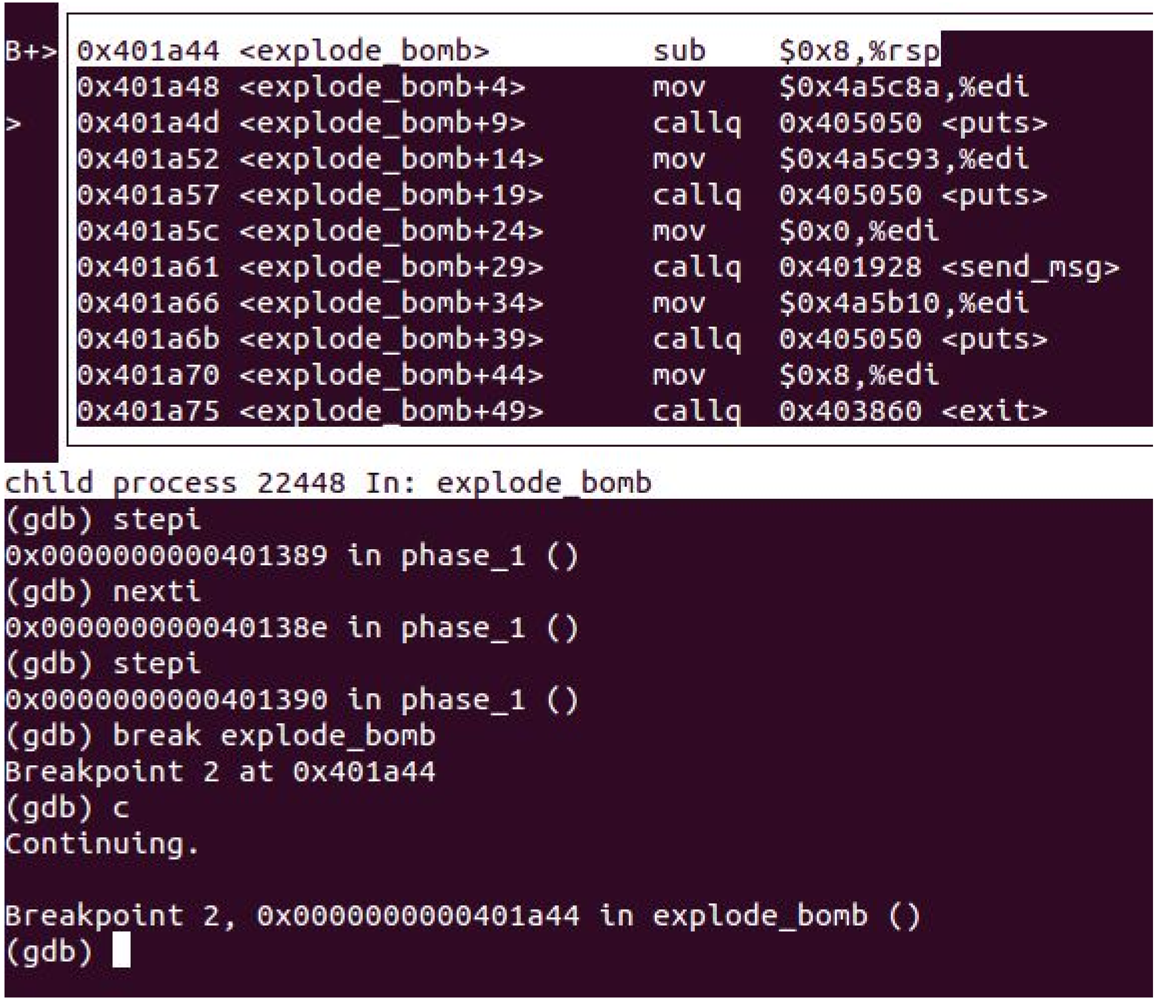
```
B+ │0x401380 <phase_1>       sub    $0x8,%rsp
   │0x401384 <phase_1+4>     mov    $0x4a5950,%esi
>  │0x401389 <phase_1+9>     callq  0x401770 <strings_not_equal>
   │0x40138e <phase_1+14>    test   %eax,%eax
  >│0x401390 <phase_1+16>    je     0x401397 <phase_1+23>
   │0x401392 <phase_1+18>    callq  0x401a44 <explode_bomb>
   │0x401397 <phase_1+23>    add    $0x8,%rsp
   │0x40139b <phase_1+27>    retq
   │0x40139c <phase_2>       push   %rbx
   │0x40139d <phase_2+1>     sub    $0x20,%rsp
   │0x4013a1 <phase_2+5>     mov    %rsp,%rsi

child process 22448 In: phase_1
(gdb) stepi
0x0000000000401384 in phase_1 ()
(gdb) stepi
0x0000000000401389 in phase_1 ()
(gdb) nexti
0x000000000040138e in phase_1 ()
(gdb) stepi
0x0000000000401390 in phase_1 ()
(gdb) 
```

Continuing execution after break

- **If you are tired of single stepping line after line, type 'c' to continue running**
- **But wait! The bomb may explode! Clearly, we should avoid entering explode_bomb()**
- **Insert a breakpoint at explode_bomb() and then type 'c'**
  - Breakpoint hit: Wrong Input,
  - Breakpoint miss: Correct Input
- **We avoid exploding bomb even with the wrong input**

Continuing execution after break

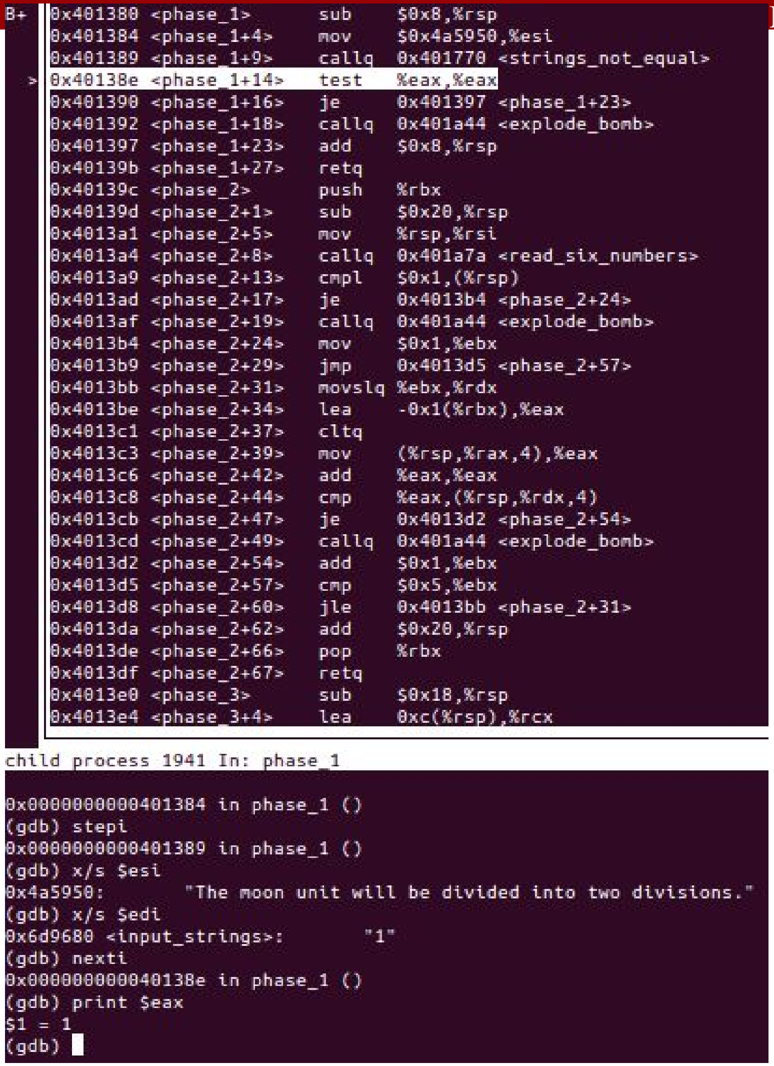
- So, we did hit the explode_bomb() break point!
- Our input ‘1’ was wrong :(
- What is the right input ?



```
B+> 0x401a44 <explode_bomb>         sub    $0x8,%rsp
    0x401a48 <explode_bomb+4>       mov    $0x4a5c8a,%edi
>   0x401a4d <explode_bomb+9>       callq  0x405050 <puts>
    0x401a52 <explode_bomb+14>      mov    $0x4a5c93,%edi
    0x401a57 <explode_bomb+19>      callq  0x405050 <puts>
    0x401a5c <explode_bomb+24>      mov    $0x0,%edi
    0x401a61 <explode_bomb+29>      callq  0x401928 <send_msg>
    0x401a66 <explode_bomb+34>      mov    $0x4a5b10,%edi
    0x401a6b <explode_bomb+39>      callq  0x405050 <puts>
    0x401a70 <explode_bomb+44>      mov    $0x8,%edi
    0x401a75 <explode_bomb+49>      callq  0x403860 <exit>

child process 22448 In: explode_bomb
(gdb) stepi
0x0000000000401389 in phase_1 ()
(gdb) nexti
0x000000000040138e in phase_1 ()
(gdb) stepi
0x0000000000401390 in phase_1 ()
(gdb) break explode_bomb
Breakpoint 2 at 0x401a44
(gdb) c
Continuing.

Breakpoint 2, 0x0000000000401a44 in explode_bomb ()
(gdb) 
```

Examining variables

- Critical function: strings_not_equal()
- Critical values: Arguments and return values of strings_not_equal()
- Examine the values of both these registers
- Remember that our input was “1”

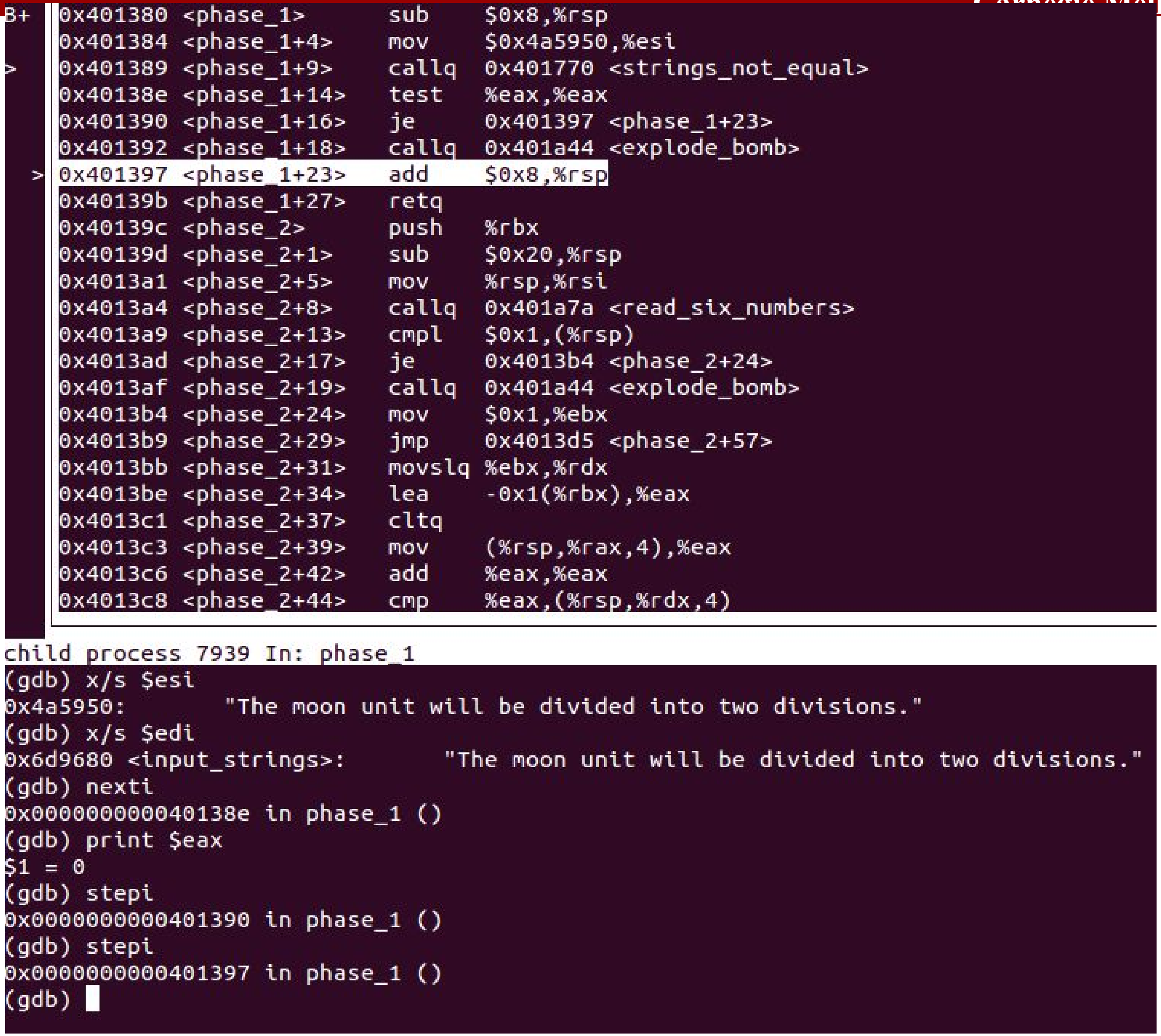


```
B+  0x401380 <phase_1>       sub    $0x8,%rsp
    0x401384 <phase_1+4>     mov    $0x4a5950,%esi
    0x401389 <phase_1+9>     callq  0x401770 <strings_not_equal>
  > 0x40138e <phase_1+14>    test   %eax,%eax
    0x401390 <phase_1+16>    je     0x401397 <phase_1+23>
    0x401392 <phase_1+18>    callq  0x401a44 <explode_bomb>
    0x401397 <phase_1+23>    add    $0x8,%rsp
    0x40139b <phase_1+27>    retq
    0x40139c <phase_2>       push   %rbx
    0x40139d <phase_2+1>     sub    $0x20,%rsp
    0x4013a1 <phase_2+5>     mov    %rsp,%rsi
    0x4013a4 <phase_2+8>     callq  0x401a7a <read_six_numbers>
    0x4013a9 <phase_2+13>    cmpl   $0x1,(%rsp)
    0x4013ad <phase_2+17>    je     0x4013b4 <phase_2+24>
    0x4013af <phase_2+19>    callq  0x401a44 <explode_bomb>
    0x4013b4 <phase_2+24>    mov    $0x1,%ebx
    0x4013b9 <phase_2+29>    jmp    0x4013d5 <phase_2+57>
    0x4013bb <phase_2+31>    movslq %ebx,%rdx
    0x4013be <phase_2+34>    lea    -0x1(%rbx),%eax
    0x4013c1 <phase_2+37>    cltq
    0x4013c3 <phase_2+39>    mov    (%rsp,%rax,4),%eax
    0x4013c6 <phase_2+42>    add    %eax,%eax
    0x4013c8 <phase_2+44>    cmp    %eax,(%rsp,%rdx,4)
    0x4013cb <phase_2+47>    je     0x4013d2 <phase_2+54>
    0x4013cd <phase_2+49>    callq  0x401a44 <explode_bomb>
    0x4013d2 <phase_2+54>    add    $0x1,%ebx
    0x4013d5 <phase_2+57>    cmp    $0x5,%ebx
    0x4013d8 <phase_2+60>    jle    0x4013bb <phase_2+31>
    0x4013da <phase_2+62>    add    $0x20,%rsp
    0x4013de <phase_2+66>    pop    %rbx
    0x4013df <phase_2+67>    retq
    0x4013e0 <phase_3>       sub    $0x18,%rsp
    0x4013e4 <phase_3+4>     lea    0xc(%rsp),%rcx

child process 1941 In: phase_1

0x0000000000401384 in phase_1 ()
(gdb) stepi
0x0000000000401389 in phase_1 ()
(gdb) x/s $esi
0x4a5950:        "The moon unit will be divided into two divisions."
(gdb) x/s $edi
0x6d9680 <input_strings>:        "1"
(gdb) nexti
0x000000000040138e in phase_1 ()
(gdb) print $eax
$1 = 1
(gdb)
```

So, what should our input be ?

The moon unit will be divided into two divisions.

Time to test….



```
B+  0x401380 <phase_1>       sub    $0x8,%rsp
    0x401384 <phase_1+4>     mov    $0x4a5950,%esi
>   0x401389 <phase_1+9>     callq  0x401770 <strings_not_equal>
    0x40138e <phase_1+14>    test   %eax,%eax
    0x401390 <phase_1+16>    je     0x401397 <phase_1+23>
    0x401392 <phase_1+18>    callq  0x401a44 <explode_bomb>
  > 0x401397 <phase_1+23>    add    $0x8,%rsp
    0x40139b <phase_1+27>    retq
    0x40139c <phase_2>       push   %rbx
    0x40139d <phase_2+1>     sub    $0x20,%rsp
    0x4013a1 <phase_2+5>     mov    %rsp,%rsi
    0x4013a4 <phase_2+8>     callq  0x401a7a <read_six_numbers>
    0x4013a9 <phase_2+13>    cmpl   $0x1,(%rsp)
    0x4013ad <phase_2+17>    je     0x4013b4 <phase_2+24>
    0x4013af <phase_2+19>    callq  0x401a44 <explode_bomb>
    0x4013b4 <phase_2+24>    mov    $0x1,%ebx
    0x4013b9 <phase_2+29>    jmp    0x4013d5 <phase_2+57>
    0x4013bb <phase_2+31>    movslq %ebx,%rdx
    0x4013be <phase_2+34>    lea    -0x1(%rbx),%eax
    0x4013c1 <phase_2+37>    cltq
    0x4013c3 <phase_2+39>    mov    (%rsp,%rax,4),%eax
    0x4013c6 <phase_2+42>    add    %eax,%eax
    0x4013c8 <phase_2+44>    cmp    %eax,(%rsp,%rdx,4)

child process 7939 In: phase_1
(gdb) x/s $esi
0x4a5950:       "The moon unit will be divided into two divisions."
(gdb) x/s $edi
0x6d9680 <input_strings>:       "The moon unit will be divided into two divisions."
(gdb) nexti
0x000000000040138e in phase_1 ()
(gdb) print $eax
$1 = 0
(gdb) stepi
0x0000000000401390 in phase_1 ()
(gdb) stepi
0x0000000000401397 in phase_1 ()
(gdb) 
```

Yay, bomb defused !

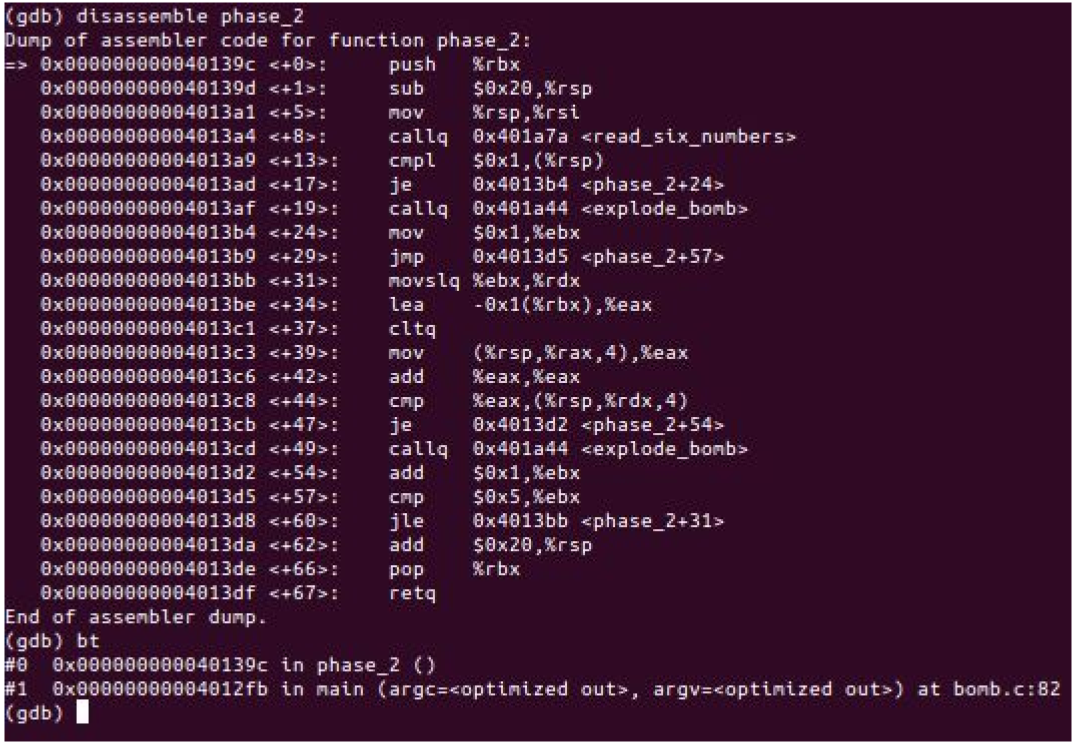
Examining and Modifying Variables

- **print *expression/variable:* Print value of variable/expression**
- **watch expression/variable: Break each time the expression/variable is written**
- **set variable expression: Eg: set variable x=20**
- **Examining registers**

print /d $rax:	Print contents of %rax in decimal
print /x $rax:	Print contents of %rax in hex
print /t $rax:	Print contents of %rax in binary
print *(int *) 0xbffff890:	Print integer at address 0xbffff890
print *(int *) ($rsp+8):	Print integer at address %rsp + 8
print (char *) 0xbfff890:	Examine a string stored at 0xbffff890
x/w 0xbffff890:	Examine (4-byte) word starting at address 0xbffff890
x/2w $rsp:	Examine 2 (4-byte) word starting at address in $rsp
x/s $rsp:	Examine a string stored at the address stored in $rsp

Examining code

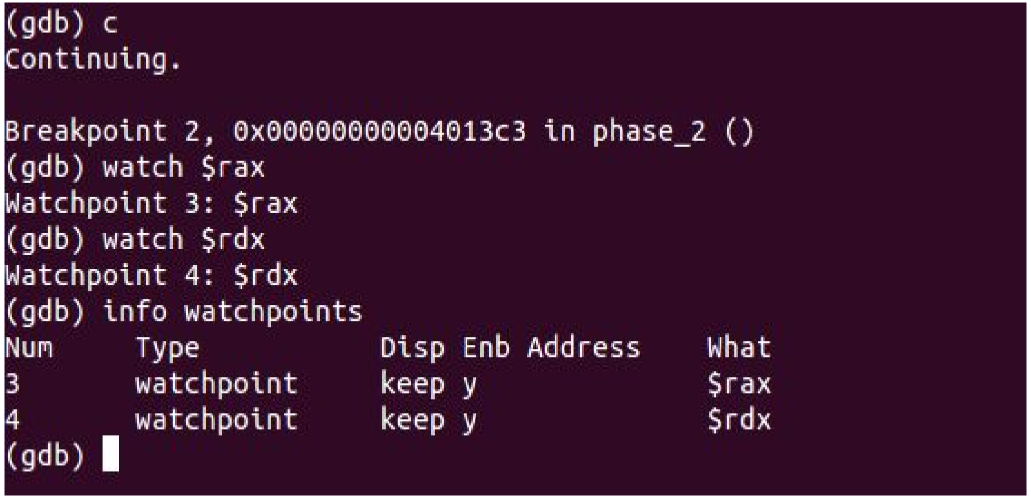
- disas: Disassemble current function
- disas sum: Disassemble function sum
- disas 0x80483b7: Disassemble function around 0x80483b7
- disas 0x80483b7 0x80483c7: Disassemble code within specified address range
- backtrace: print the current stack



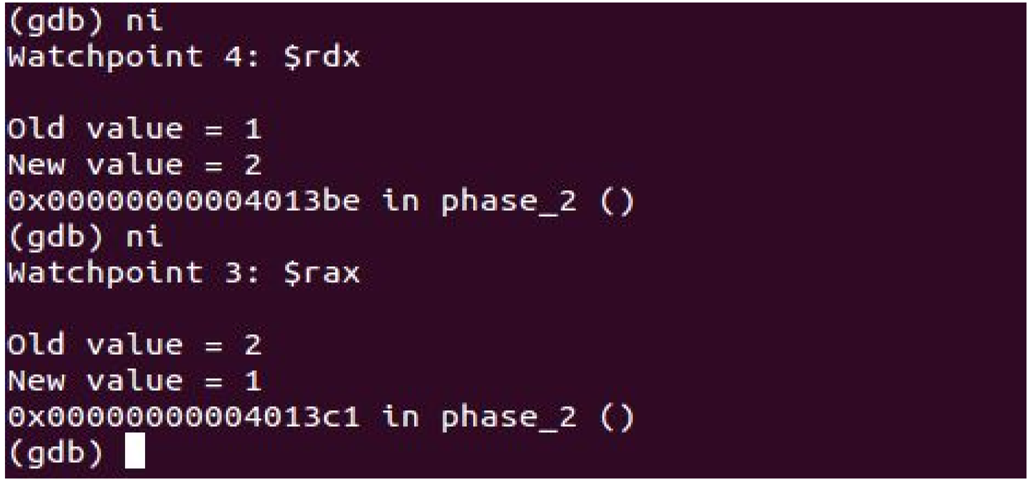
```
(gdb) disassemble phase_2
Dump of assembler code for function phase_2:
=> 0x000000000040139c <+0>:     push   %rbx
   0x000000000040139d <+1>:     sub    $0x20,%rsp
   0x00000000004013a1 <+5>:     mov    %rsp,%rsi
   0x00000000004013a4 <+8>:     callq  0x401a7a <read_six_numbers>
   0x00000000004013a9 <+13>:    cmpl   $0x1,(%rsp)
   0x00000000004013ad <+17>:    je     0x4013b4 <phase_2+24>
   0x00000000004013af <+19>:    callq  0x401a44 <explode_bomb>
   0x00000000004013b4 <+24>:    mov    $0x1,%ebx
   0x00000000004013b9 <+29>:    jmp    0x4013d5 <phase_2+57>
   0x00000000004013bb <+31>:    movslq %ebx,%rdx
   0x00000000004013be <+34>:    lea    -0x1(%rbx),%eax
   0x00000000004013c1 <+37>:    cltq
   0x00000000004013c3 <+39>:    mov    (%rsp,%rax,4),%eax
   0x00000000004013c6 <+42>:    add    %eax,%eax
   0x00000000004013c8 <+44>:    cmp    %eax,(%rsp,%rdx,4)
   0x00000000004013cb <+47>:    je     0x4013d2 <phase_2+54>
   0x00000000004013cd <+49>:    callq  0x401a44 <explode_bomb>
   0x00000000004013d2 <+54>:    add    $0x1,%ebx
   0x00000000004013d5 <+57>:    cmp    $0x5,%ebx
   0x00000000004013d8 <+60>:    jle    0x4013bb <phase_2+31>
   0x00000000004013da <+62>:    add    $0x20,%rsp
   0x00000000004013de <+66>:    pop    %rbx
   0x00000000004013df <+67>:    retq
End of assembler dump.
(gdb) bt
#0  0x000000000040139c in phase_2 ()
#1  0x00000000004012fb in main (argc=<optimized out>, argv=<optimized out>) at bomb.c:82
(gdb)
```

Inserting Watchpoints

- Watchpoints are special breakpoints
- They trigger when an expression changes
- Useful for watching specific registers, especially in loops. Avoids having to print out values each time



```
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004013c3 in phase_2 ()
(gdb) watch $rax
Watchpoint 3: $rax
(gdb) watch $rdx
Watchpoint 4: $rdx
(gdb) info watchpoints
Num     Type           Disp Enb Address    What
3       watchpoint     keep y              $rax
4       watchpoint     keep y              $rdx
(gdb)
```



```
(gdb) ni
Watchpoint 4: $rdx

Old value = 1
New value = 2
0x00000000004013be in phase_2 ()
(gdb) ni
Watchpoint 3: $rax

Old value = 2
New value = 1
0x00000000004013c1 in phase_2 ()
(gdb)
```

More useful GDB constructs

- Examine contents in memory using expressions: print *(int *) ($rsp + 4*$rdx)
- Examine multiple words on stack: x/6w $rsp
- break at certain addresses (useful to examine only the interesting parts of the code): break *0xabcd

Resources

- http://csapp.cs.cmu.edu/2e/docs/gdbnotes-x86-64.pdf
- https://beej.us/guide/bggdb/
- http://www.delorie.com/gnu/docs/gdb/gdb_toc.html
- How debuggers work: https://blog.0x972.info/?d=2014/11/13/10/40/50-how-does-a-debugger-work