4190.308 Computer Architecture

Bomb Lab Hints

Bomb lab

- Objective of the lab: diffuse a binary bomb by analysing its source and disassembled code. This way, you will be able to find out the correct diffusal inputs.

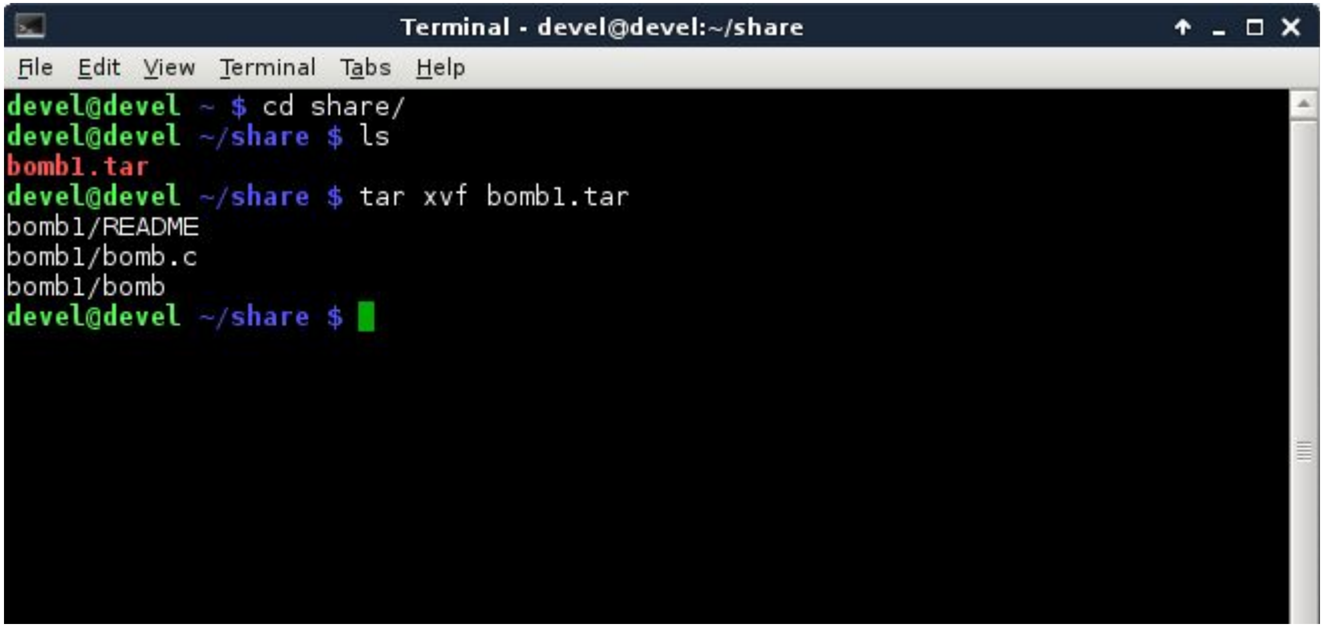
- In this slides, you will:
    - Learn how to read assembly code
    - Learn how to use the tools necessary to deal with assembly code
        - gdb
        - objdump

Getting Started

- Environment
  - we recommend to use the Gentoo virtual machine provide on eTL
    - all tools required to solve the lab are pre-installed in the VM
    - get it from:
    - http://etl.snu.ac.kr/mod/ubboard/article.php?id=806856&bwid=1654804
  - the bomb is compiled for x86_64 and should thus run on (almost) any sufficiently recent Linux installation
  - the bomb does not do any harm to your computer (only to your score)

Downloading the Bomb

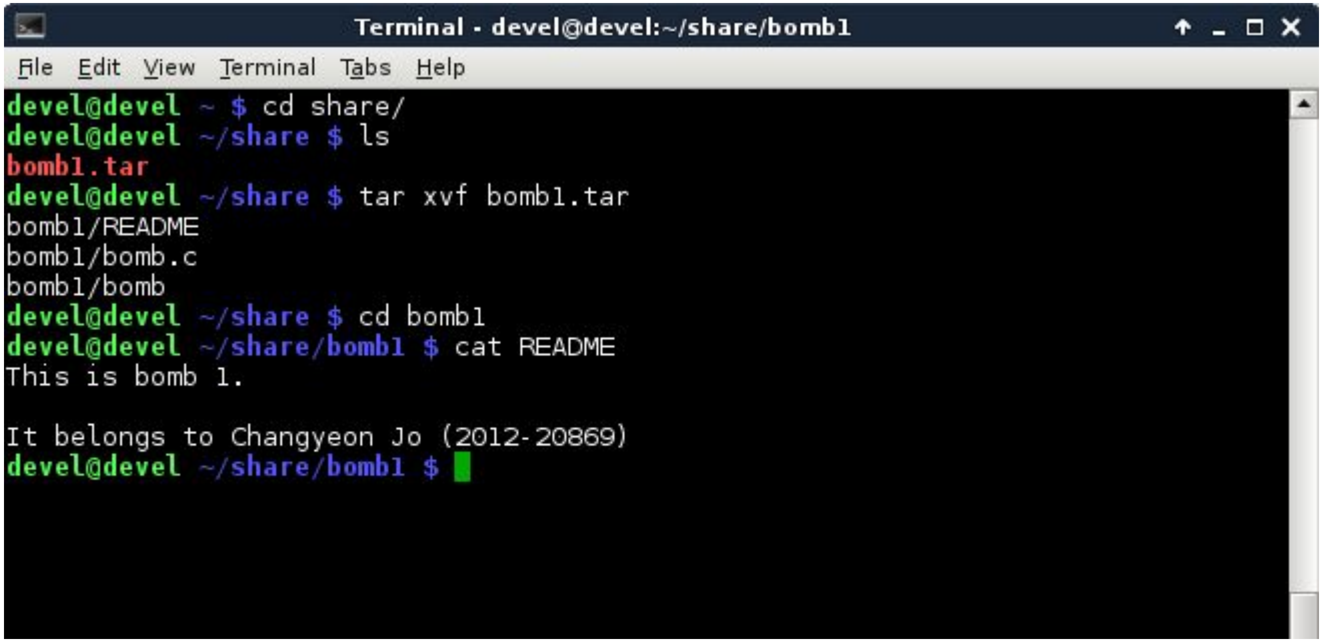
- Visit
  - http://csap.snu.ac.kr/comparch/bomblab/
- Fill in your name and student number to download your personalized bomb
- Save the bomb file to a directory of your choice, then extract the tar archive:


Terminal - devel@devel:~/share
File Edit View Terminal Tabs Help

```
devel@devel ~ $ cd share/
devel@devel ~/share $ ls
bomb1.tar
devel@devel ~/share $ tar xvf bomb1.tar
bomb1/README
bomb1/bomb.c
bomb1/bomb
devel@devel ~/share $
```

Downloading the Bomb

- Bombs are custom-built, i.e., each student gets a different bomb
- The folder contains a README file with the information you entered



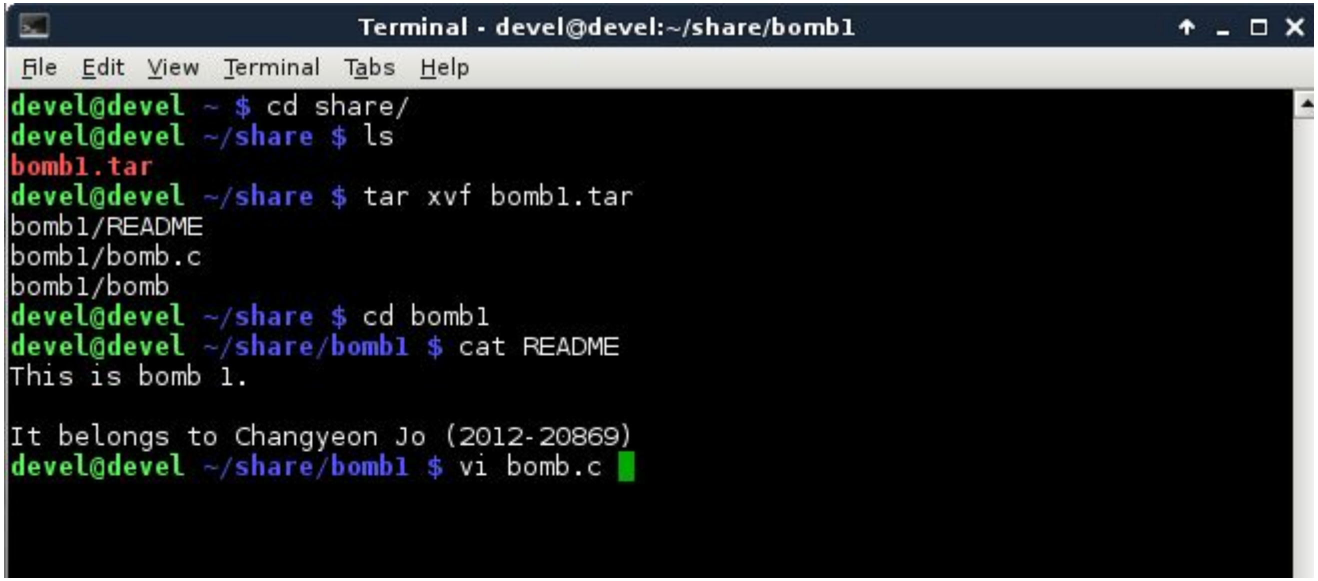
Terminal - devel@devel:~/share/bomb1
File Edit View Terminal Tabs Help

```
devel@devel ~ $ cd share/
devel@devel ~/share $ ls
bomb1.tar
devel@devel ~/share $ tar xvf bomb1.tar
bomb1/README
bomb1/bomb.c
bomb1/bomb
devel@devel ~/share $ cd bomb1
devel@devel ~/share/bomb1 $ cat README
This is bomb 1.

It belongs to Changyeon Jo (2012-20869)
devel@devel ~/share/bomb1 $
```

Inspecting the Bomb's Source Code

- The source code for the main bomb file is provided. From this file, you can get important information on how the bomb runs.

- Open a terminal, cd into the bomb directory, and open the bomb.
  The example below uses the vi editor; if you are not comfortable with vi you can use any other editor:

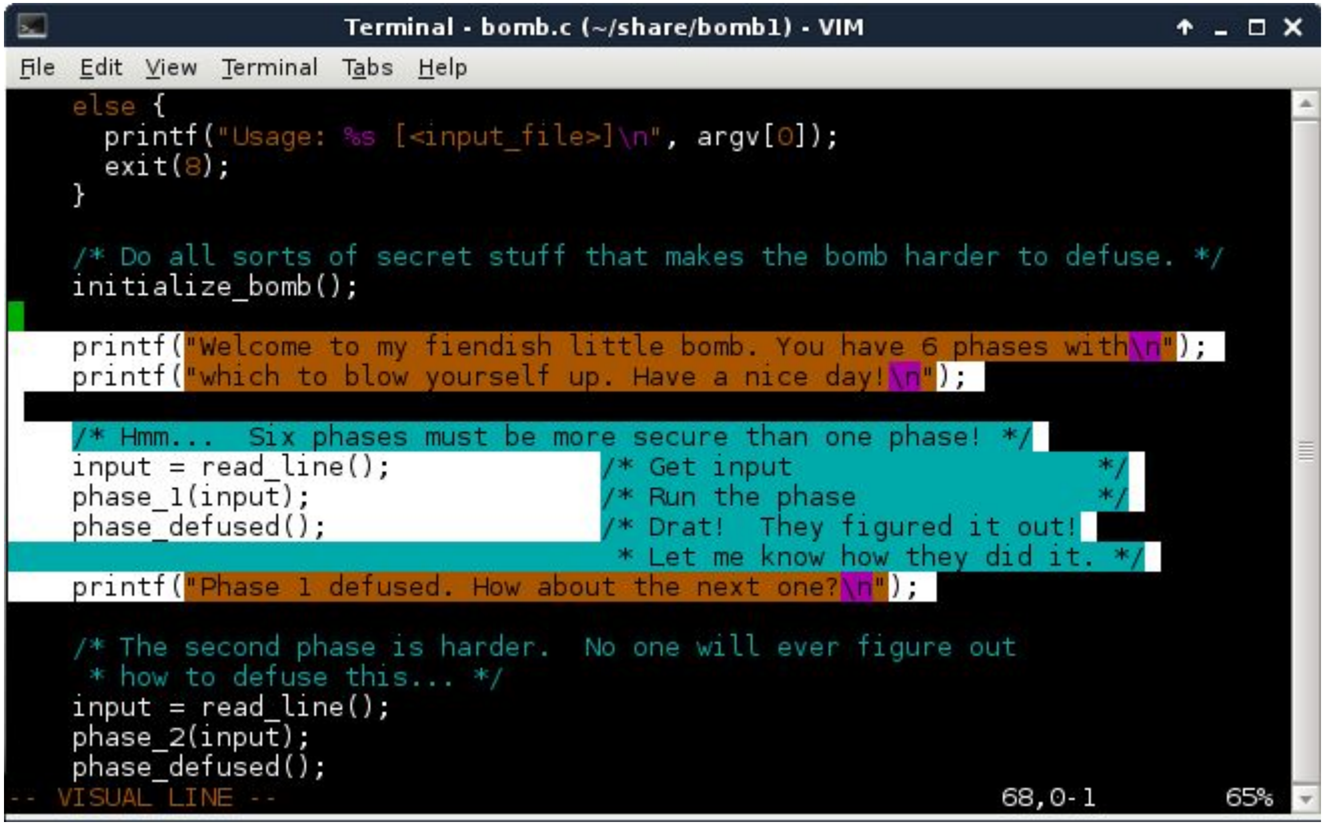


Terminal - devel@devel:~/share/bomb1
File Edit View Terminal Tabs Help

```
devel@devel ~ $ cd share/
devel@devel ~/share $ ls
bomb1.tar
devel@devel ~/share $ tar xvf bomb1.tar
bomb1/README
bomb1/bomb.c
bomb1/bomb
devel@devel ~/share $ cd bomb1
devel@devel ~/share/bomb1 $ cat README
This is bomb 1.

It belongs to Changyeon Jo (2012-20869)
devel@devel ~/share/bomb1 $ vi bomb.c
```

Inspecting the Bomb’s Source Code

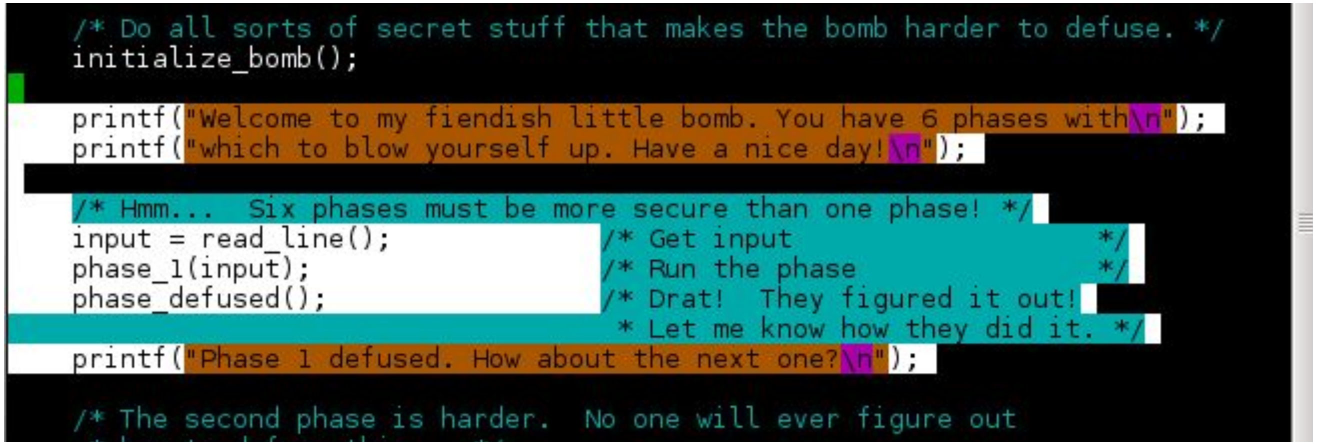
- In the main() function, find the code that reads and checks the input for each phase. In the example below, the code for phase_1 is highlighted



Terminal - bomb.c (~/share/bomb1) - VIM
File Edit View Terminal Tabs Help

```
    else {
      printf("Usage: %s [<input_file>]\n", argv[0]);
      exit(8);
    }

    /* Do all sorts of secret stuff that makes the bomb harder to defuse. */
    initialize_bomb();

    printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
    printf("which to blow yourself up. Have a nice day!\n");

    /* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                                      * Let me know how they did it. */
    printf("Phase 1 defused. How about the next one?\n");

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();
-- VISUAL LINE --                                              68,0-1        65%
```

Inspecting the Bomb’s Source Code



```
    /* Do all sorts of secret stuff that makes the bomb harder to defuse. */
    initialize_bomb();

    printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
    printf("which to blow yourself up. Have a nice day!\n");

    /* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                                      * Let me know how they did it. */
    printf("Phase 1 defused. How about the next one?\n");

    /* The second phase is harder.  No one will ever figure out
```

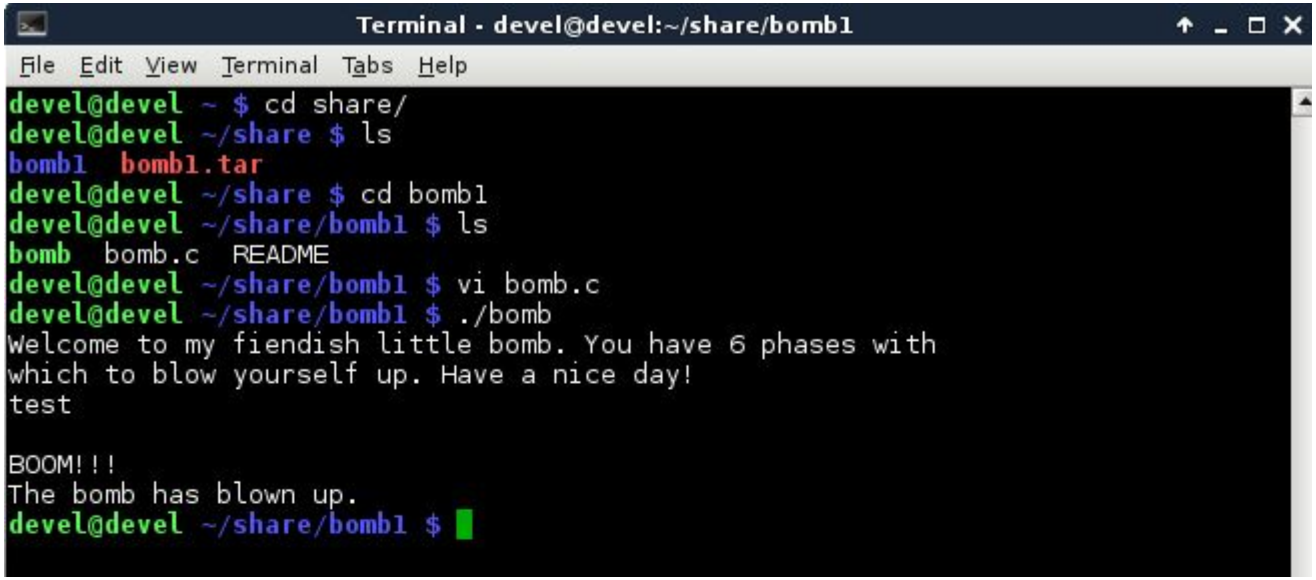
- We see that the input string is stored in variable input which is then used as an argument for the function phase_1().

- We conclude that it might be a good idea to have a closer look at the function phase_1().

Running the Bomb

- First, let’s see what happened when we run the bomb. Maybe we can guess the input string.

Let’s try “test”:



Terminal - devel@devel:~/share/bomb1
File Edit View Terminal Tabs Help

```
devel@devel ~ $ cd share/
devel@devel ~/share $ ls
bomb1  bomb1.tar
devel@devel ~/share $ cd bomb1
devel@devel ~/share/bomb1 $ ls
bomb  bomb.c  README
devel@devel ~/share/bomb1 $ vi bomb.c
devel@devel ~/share/bomb1 $ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
test

BOOM!!!
The bomb has blown up.
devel@devel ~/share/bomb1 $
```

- Hmmm…this is not going to work (you might want to avoid trying that)

Disassembling the Bomb using objdump

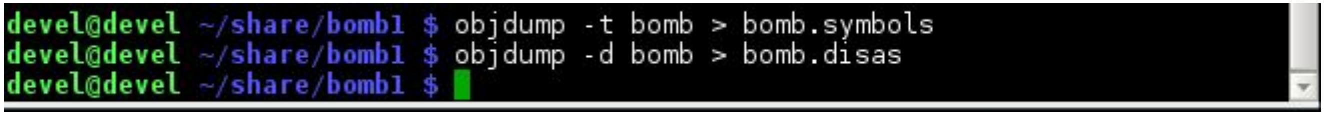
- **objdump** can display the bomb's symbol table (contains names of functions, variables, and other symbols) and also disassemble the code of the bomb.

- The output is rather long, so let's dump it to two files
  - save the symbol table by executing

    objdump –t bomb > bomb.symbols

  - disassemble the bomb's code and save it to bomb.disas by executing

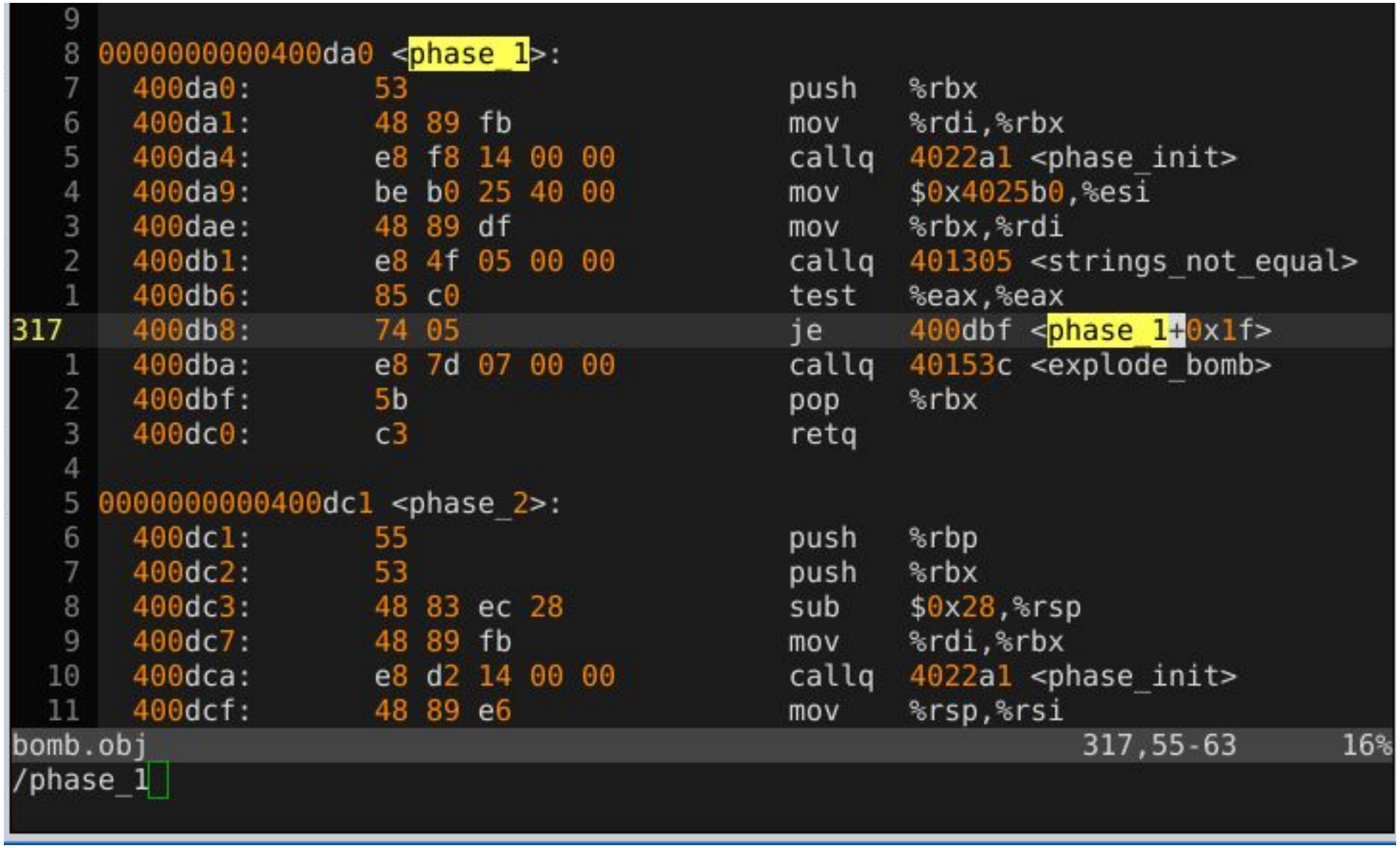
    objdump –d bomb > bomb.disas



```
devel@devel ~/share/bomb1 $ objdump -t bomb > bomb.symbols
devel@devel ~/share/bomb1 $ objdump -d bomb > bomb.disas
devel@devel ~/share/bomb1 $ █
```

Inspecting the code of phase_1()

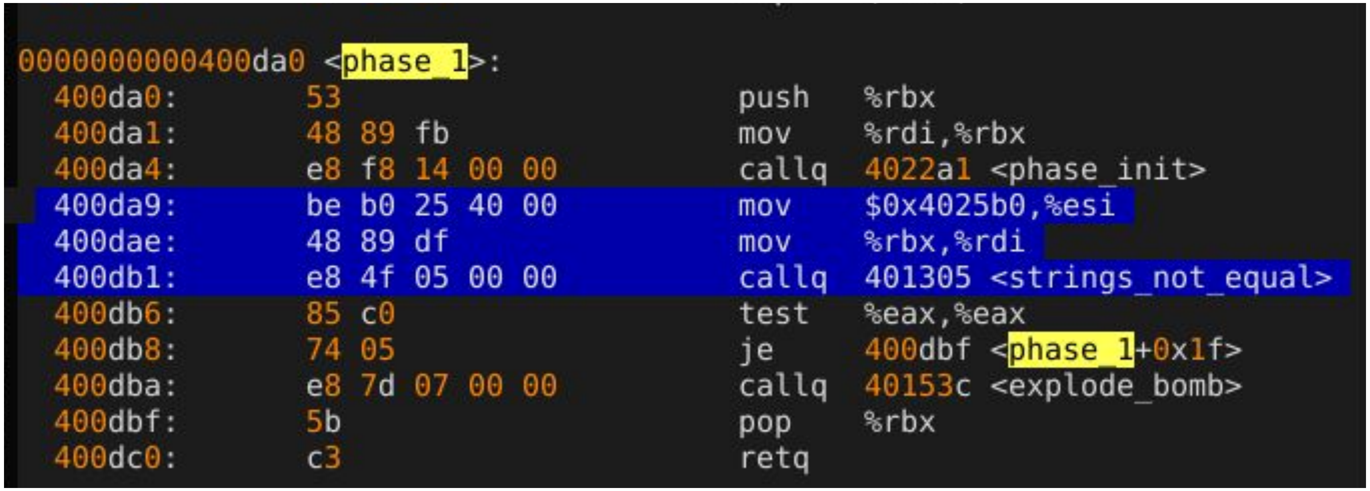
- Open the disassembled code in a text editor and locate **phase_1()**



```

0000000000400da0 <phase_1>:
  400da0:       53                      push   %rbx
  400da1:       48 89 fb                mov    %rdi,%rbx
  400da4:       e8 f8 14 00 00          callq  4022a1 <phase_init>
  400da9:       be b0 25 40 00          mov    $0x4025b0,%esi
  400dae:       48 89 df                mov    %rbx,%rdi
  400db1:       e8 4f 05 00 00          callq  401305 <strings_not_equal>
  400db6:       85 c0                   test   %eax,%eax
  400db8:       74 05                   je     400dbf <phase_1+0x1f>
  400dba:       e8 7d 07 00 00          callq  40153c <explode_bomb>
  400dbf:       5b                      pop    %rbx
  400dc0:       c3                      retq

0000000000400dc1 <phase_2>:
  400dc1:       55                      push   %rbp
  400dc2:       53                      push   %rbx
  400dc3:       48 83 ec 28             sub    $0x28,%rsp
  400dc7:       48 89 fb                mov    %rdi,%rbx
  400dca:       e8 d2 14 00 00          callq  4022a1 <phase_init>
  400dcf:       48 89 e6                mov    %rsp,%rsi
bomb.obj                                                  317,55-63      16%
/phase_1
```

Inspecting the code of phase_1()

- From the code we can see that:
- **phase_1** calls a function called **strings_not_equal()** with two arguments (it pushes two values on the stack)
- then, depending on the result of **strings_not_equal()** in register %eax either calls **explode_bomb()** or returns.

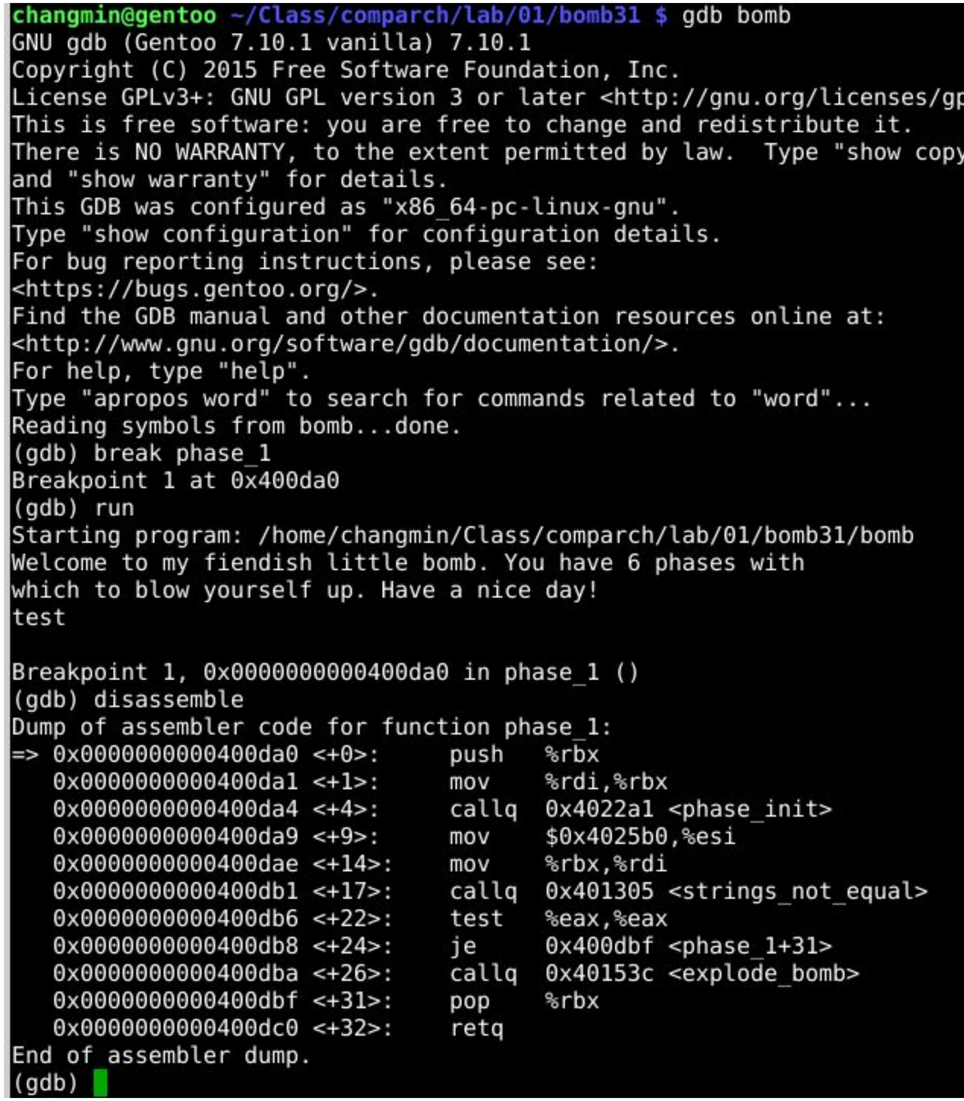


```
0000000000400da0 <phase_1>:
  400da0:       53                      push   %rbx
  400da1:       48 89 fb                mov    %rdi,%rbx
  400da4:       e8 f8 14 00 00          callq  4022a1 <phase_init>
  400da9:       be b0 25 40 00          mov    $0x4025b0,%esi
  400dae:       48 89 df                mov    %rbx,%rdi
  400db1:       e8 4f 05 00 00          callq  401305 <strings_not_equal>
  400db6:       85 c0                   test   %eax,%eax
  400db8:       74 05                   je     400dbf <phase_1+0x1f>
  400dba:       e8 7d 07 00 00          callq  40153c <explode_bomb>
  400dbf:       5b                      pop    %rbx
  400dc0:       c3                      retq
```

Debugging the Bomb in gdb

- With this knowledge we now run the bomb in the GNU debugger go back to the terminal and execute **gdb bomb**
  - set a breakpoint at phase_1 by using **break** as shown right

- entering **break phase_1**
  - run the bomb by entering **run**
  - enter the first string and hit enter
  - now **gdb** stops at the entry of phase_1 (disassemble with **disas**)

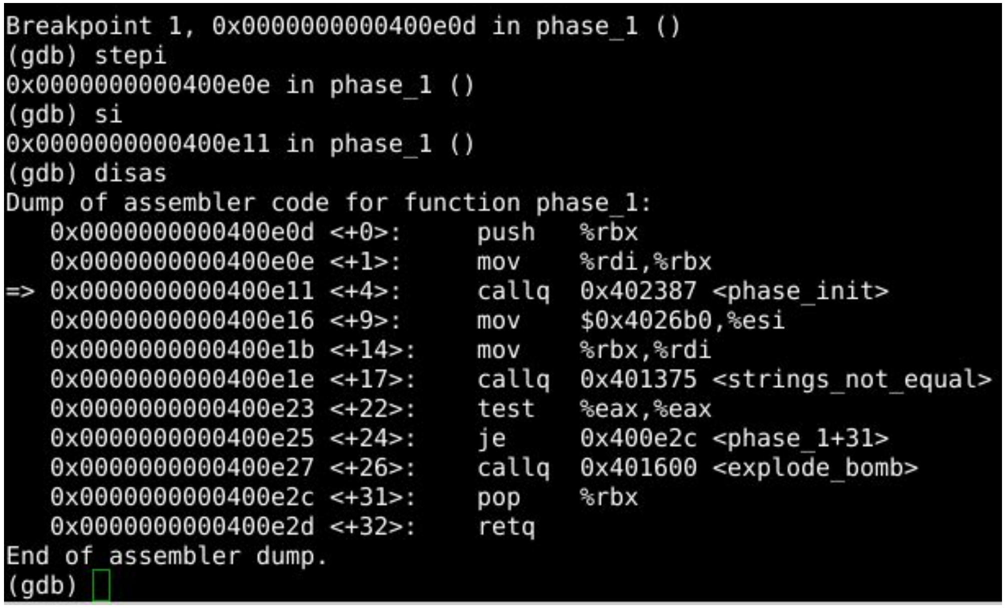


```
changmin@gentoo ~/Class/comparch/lab/01/bomb31 $ gdb bomb
GNU gdb (Gentoo 7.10.1 vanilla) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gp
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copy
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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 bomb...done.
(gdb) break phase_1
Breakpoint 1 at 0x400da0
(gdb) run
Starting program: /home/changmin/Class/comparch/lab/01/bomb31/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
test

Breakpoint 1, 0x0000000000400da0 in phase_1 ()
(gdb) disassemble
Dump of assembler code for function phase_1:
=> 0x0000000000400da0 <+0>:     push   %rbx
   0x0000000000400da1 <+1>:     mov    %rdi,%rbx
   0x0000000000400da4 <+4>:     callq  0x4022a1 <phase_init>
   0x0000000000400da9 <+9>:     mov    $0x4025b0,%esi
   0x0000000000400dae <+14>:    mov    %rbx,%rdi
   0x0000000000400db1 <+17>:    callq  0x401305 <strings_not_equal>
   0x0000000000400db6 <+22>:    test   %eax,%eax
   0x0000000000400db8 <+24>:    je     0x400dbf <phase_1+31>
   0x0000000000400dba <+26>:    callq  0x40153c <explode_bomb>
   0x0000000000400dbf <+31>:    pop    %rbx
   0x0000000000400dc0 <+32>:    retq
End of assembler dump.
(gdb) 
```

Stepping through the Code

- We can set more breakpoints and continue execution until the next breakpoint is reached. Looking at the code, a breakpoint at address **0x400e1e call 0x401375 <strings_not_equal>** seems reasonable.
  - breakpoints to addresses are set by entering **break *<address>**
  - continue execution to the next breakpoint with **cont** (or simply **c**)

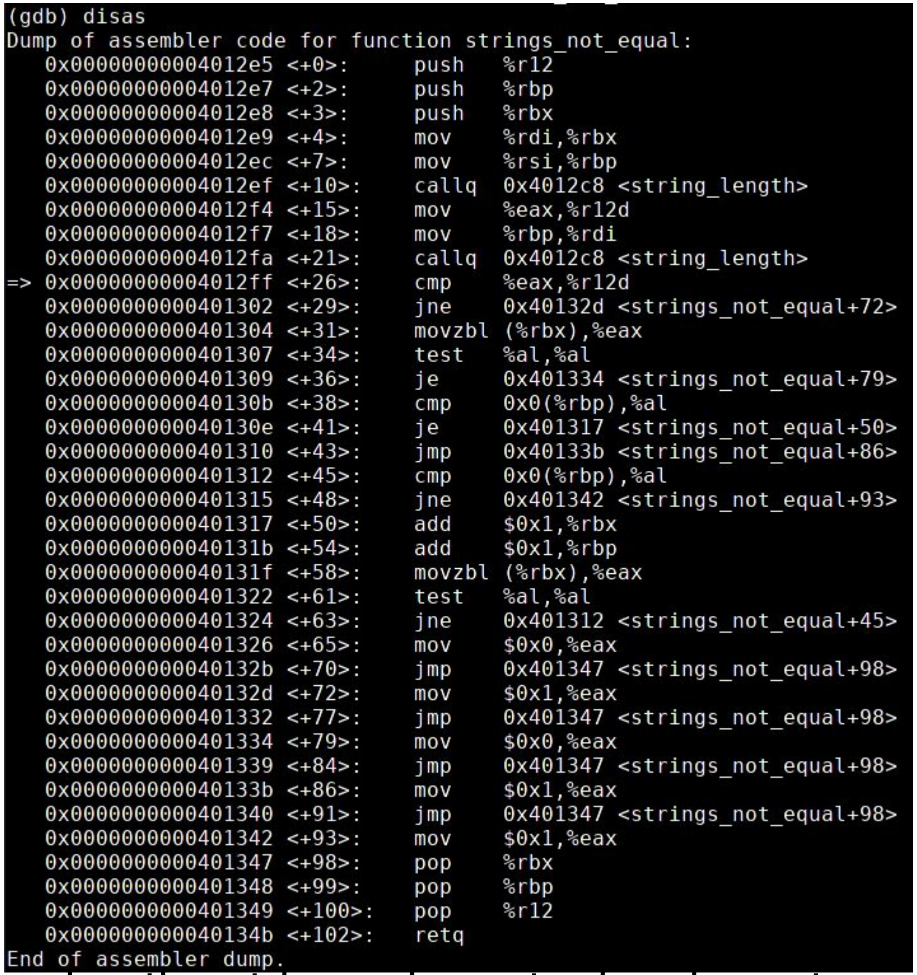
- Now, single-step instruction-by-instruction through the code by executing **stepi**
  - **step**: step through the program line-by-line (C code-wise)
  - **stepi**: step through the program one (machine) instruction exactly



```
Breakpoint 1, 0x0000000000400e0d in phase_1 ()
(gdb) stepi
0x0000000000400e0e in phase_1 ()
(gdb) si
0x0000000000400e11 in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
   0x0000000000400e0d <+0>:     push   %rbx
   0x0000000000400e0e <+1>:     mov    %rdi,%rbx
=> 0x0000000000400e11 <+4>:     callq  0x402387 <phase_init>
   0x0000000000400e16 <+9>:     mov    $0x4026b0,%esi
   0x0000000000400e1b <+14>:    mov    %rbx,%rdi
   0x0000000000400e1e <+17>:    callq  0x401375 <strings_not_equal>
   0x0000000000400e23 <+22>:    test   %eax,%eax
   0x0000000000400e25 <+24>:    je     0x400e2c <phase_1+31>
   0x0000000000400e27 <+26>:    callq  0x401600 <explode_bomb>
   0x0000000000400e2c <+31>:    pop    %rbx
   0x0000000000400e2d <+32>:    retq
End of assembler dump.
(gdb) 
```

Inspecting Registers and Memory

- After executing **stepi** at the call to **strings_not_equal,** enter **disas** again to see where we currently are

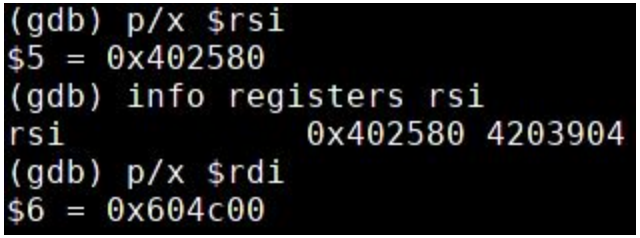


```
(gdb) disas
Dump of assembler code for function strings_not_equal:
   0x00000000004012e5 <+0>:     push   %r12
   0x00000000004012e7 <+2>:     push   %rbp
   0x00000000004012e8 <+3>:     push   %rbx
   0x00000000004012e9 <+4>:     mov    %rdi,%rbx
   0x00000000004012ec <+7>:     mov    %rsi,%rbp
   0x00000000004012ef <+10>:    callq  0x4012c8 <string_length>
   0x00000000004012f4 <+15>:    mov    %eax,%r12d
   0x00000000004012f7 <+18>:    mov    %rbp,%rdi
   0x00000000004012fa <+21>:    callq  0x4012c8 <string_length>
=> 0x00000000004012ff <+26>:    cmp    %eax,%r12d
   0x0000000000401302 <+29>:    jne    0x40132d <strings_not_equal+72>
   0x0000000000401304 <+31>:    movzbl (%rbx),%eax
   0x0000000000401307 <+34>:    test   %al,%al
   0x0000000000401309 <+36>:    je     0x401334 <strings_not_equal+79>
   0x000000000040130b <+38>:    cmp    0x0(%rbp),%al
   0x000000000040130e <+41>:    je     0x401317 <strings_not_equal+50>
   0x0000000000401310 <+43>:    jmp    0x40133b <strings_not_equal+86>
   0x0000000000401312 <+45>:    cmp    0x0(%rbp),%al
   0x0000000000401315 <+48>:    jne    0x401342 <strings_not_equal+93>
   0x0000000000401317 <+50>:    add    $0x1,%rbx
   0x000000000040131b <+54>:    add    $0x1,%rbp
   0x000000000040131f <+58>:    movzbl (%rbx),%eax
   0x0000000000401322 <+61>:    test   %al,%al
   0x0000000000401324 <+63>:    jne    0x401312 <strings_not_equal+45>
   0x0000000000401326 <+65>:    mov    $0x0,%eax
   0x000000000040132b <+70>:    jmp    0x401347 <strings_not_equal+98>
   0x000000000040132d <+72>:    mov    $0x1,%eax
   0x0000000000401332 <+77>:    jmp    0x401347 <strings_not_equal+98>
   0x0000000000401334 <+79>:    mov    $0x0,%eax
   0x0000000000401339 <+84>:    jmp    0x401347 <strings_not_equal+98>
   0x000000000040133b <+86>:    mov    $0x1,%eax
   0x0000000000401340 <+91>:    jmp    0x401347 <strings_not_equal+98>
   0x0000000000401342 <+93>:    mov    $0x1,%eax
   0x0000000000401347 <+98>:    pop    %rbx
   0x0000000000401348 <+99>:    pop    %rbp
   0x0000000000401349 <+100>:   pop    %r12
   0x000000000040134b <+102>:   retq
End of assembler dump.
```

- the debugger stopped at the first instruction of **strings_not_equal**
- we see that the function uses two arguments registers %rdi and %rsi
- from the name we guess that the function probably compares two strings. The code confirms this assumption:
  - it first calls the **string_length** function on both strings and compares their length
  - if they are not equal, it sets the result to false and exits
  - if they are equal, it starts comparing the strings character by character until the characters differ or the end of the string is reached

Inspecting Register and Memory

- Now we now the strings are stored in the two mentioned registers. Let's first print the contents of the two registers
- Use **p/x $<reg>** to print the contents of a register in hexadecimal form (replace x by a for address, d for decimal, s for string)
    - You can also inspect the register with the **info registers [reg]** command

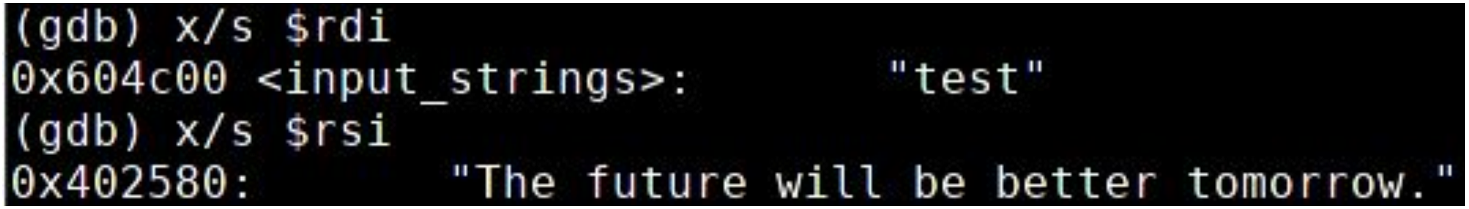


```
(gdb) p/x $rsi
$5 = 0x402580
(gdb) info registers rsi
rsi            0x402580 4203904
(gdb) p/x $rdi
$6 = 0x604c00
```

    - enter **help print (or help p)** to see what options the print command offers

Inspecting Register and Memory

- We assume that both registers contain addresses of strings. Let’s print the contents of the memory at those addresses
  - Use **x/s <address>** to dump memory contents at address interpreted as a string (again, use help **x** to get help on the different options of this function)



```
(gdb) x/s $rdi
0x604c00 <input_strings>:       "test"
(gdb) x/s $rsi
0x402580:       "The future will be better tomorrow."
```

- Indeed, we see the input string (**“test”)** as well as another string **(“The future will be better tomorrow.”)**
- Could this be the passphrase for phase 1?

- **Hint**: to restart the program, you don’t have to exit gdb, simply type “run” This has the additional benefit that all breakpoints are still set.

Now, it’s your turn!

- This walk-through showed you how to use the various debugging tools to defuse phase 1. Go on and attack the other phases, one by one.
- Scoreboard: check your score at http://csap.snu.ac.kr/comparch/bomblab/scoreboard

Good Luck!

Submit your report

- Login to https://git.csap.snu.ac.kr/
- Access https://git.csap.snu.ac.kr/comparchTA/comparch directly or through Projects/”Explore Projects”/All and look for comparchTA/CompArch
- Click the `Fork` button and select your own namespace. This will create a copy of the project to your own repository.
- In the Settings/Permissions/”Project Visibility” select `Private`
- Once this is done, copy the URL of your newly forked project (or click the `Clone` button and copy the “Clone with HTTPS” URL).
- Open a terminal, go where you want to save the project and execute:

  ```
  git clone your_URL
  ```
- You now have a version of the repository on your computer. Once you are done with your report, place it in the appropriate location and execute:

  ```
  git add path_to_report
  ```
- You can check if your were successful with `git status`
- To upload the local changes to your online repository proceed as follows:

  `git commit -m “commit message”` (bomblab report upload for example)

  ```
  git push
  ```