4190.308 Computer Architecture

Bomb Lab Hints

Bomb lab

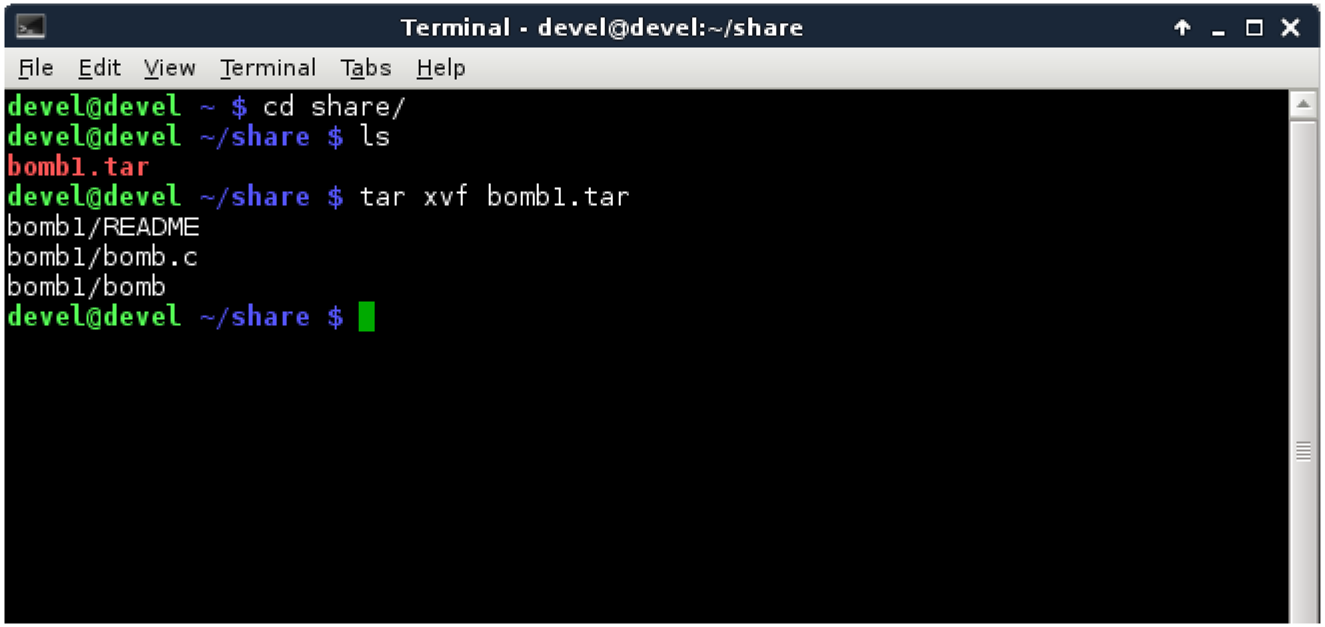
- Goal
  - Learn how to read assembly code
  - Learn how to use the tools necessary to deal with assembly code
    - gdb
    - objdump
    - strings

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=363927&ls=15&bwid=828239
  - the bomb is complied for IA32 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)
    - you might need to install additional software to run the lab

Downloading the Bomb

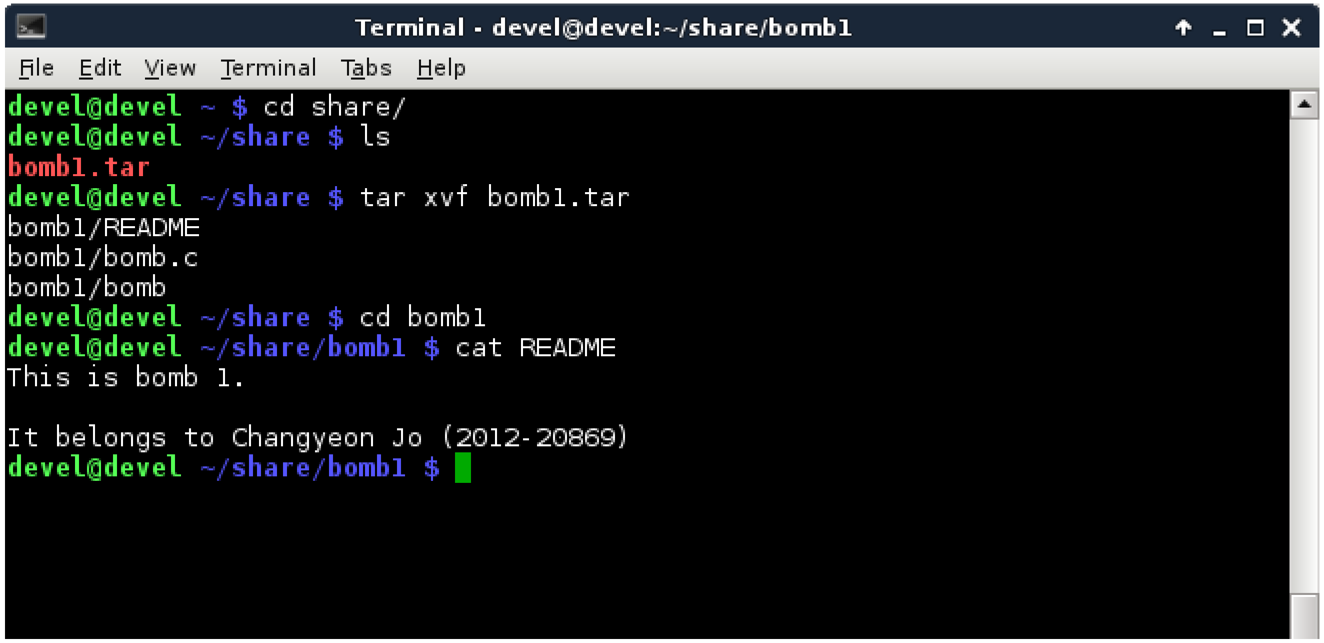
- Visit
  - http://csap.snu.ac.kr:65421/
- and fill in your name and student number to download your personalized bomb
- Save the bomb file to a directory of your choice, then extract the tar archive:


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

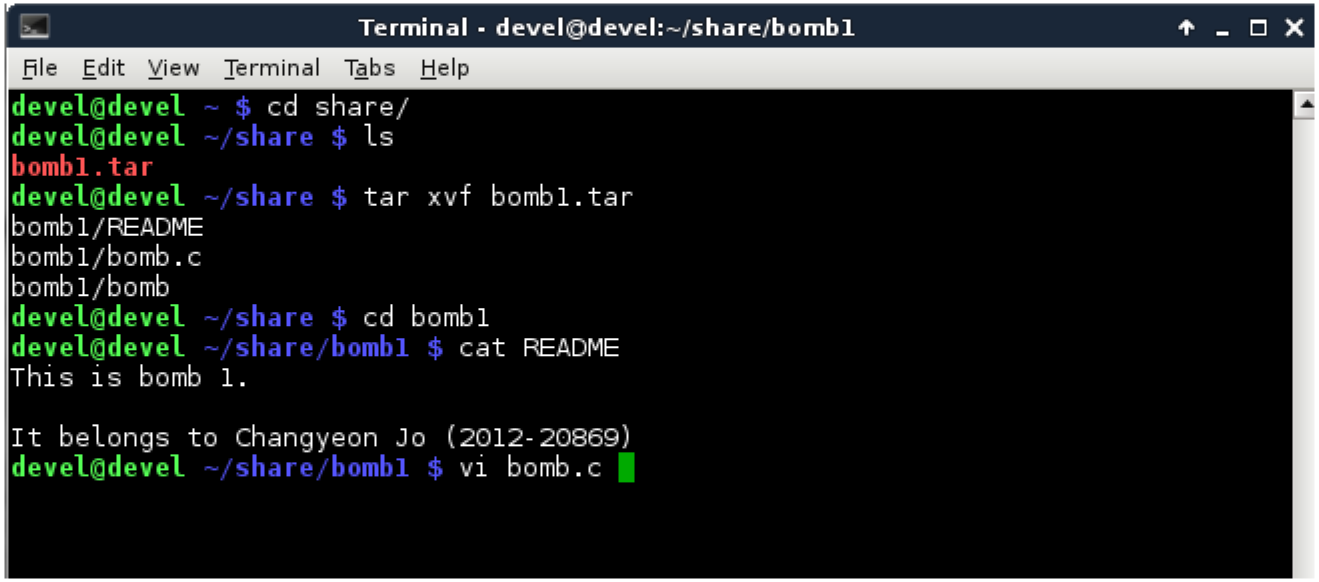

Terminal - devel@devel:~/share/bomb1
File Edit View Terminal Tabs Help

```
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.

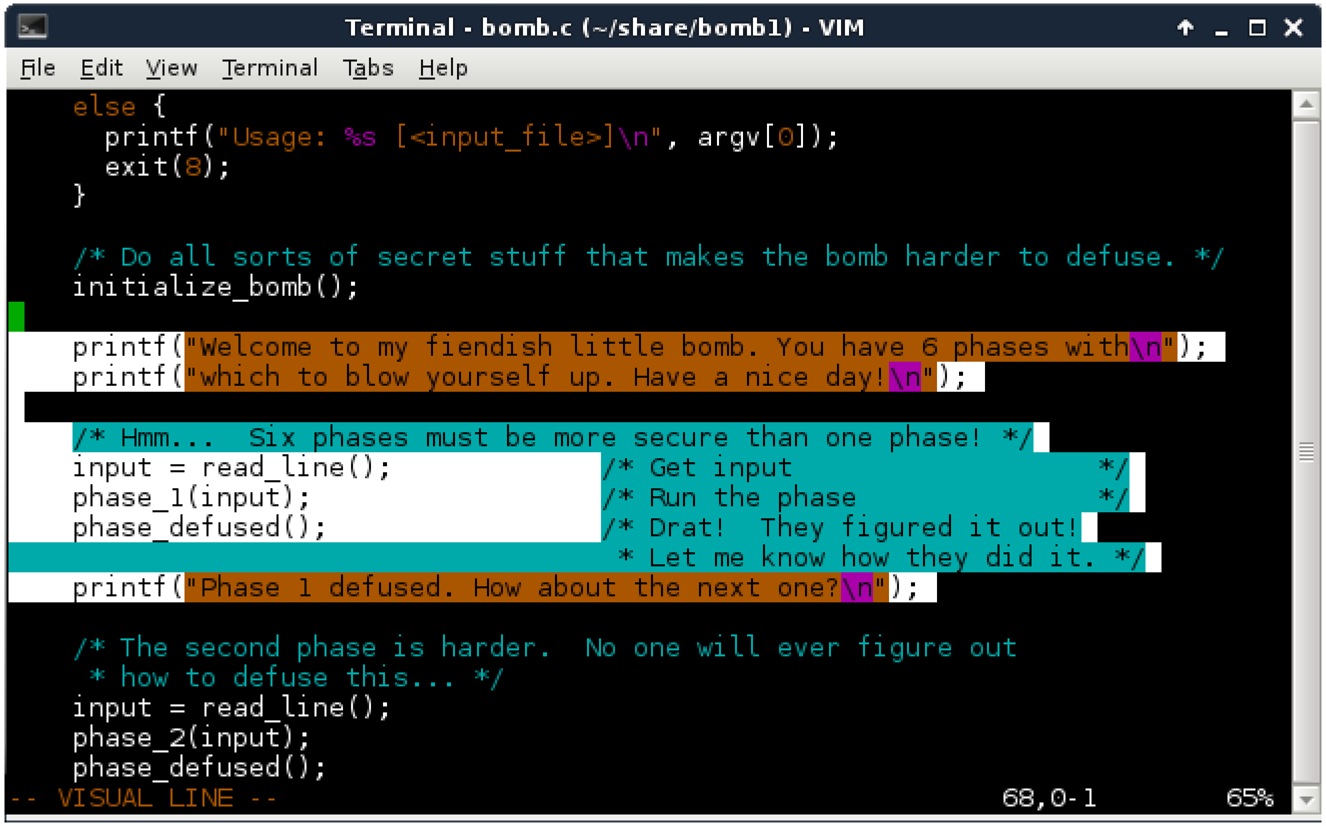
- Open a terminal, cd into the bomb directory, and open the bomb.
  The example below uses the vi editor; if you are not comfortable with vi you can use any other editor:


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
```

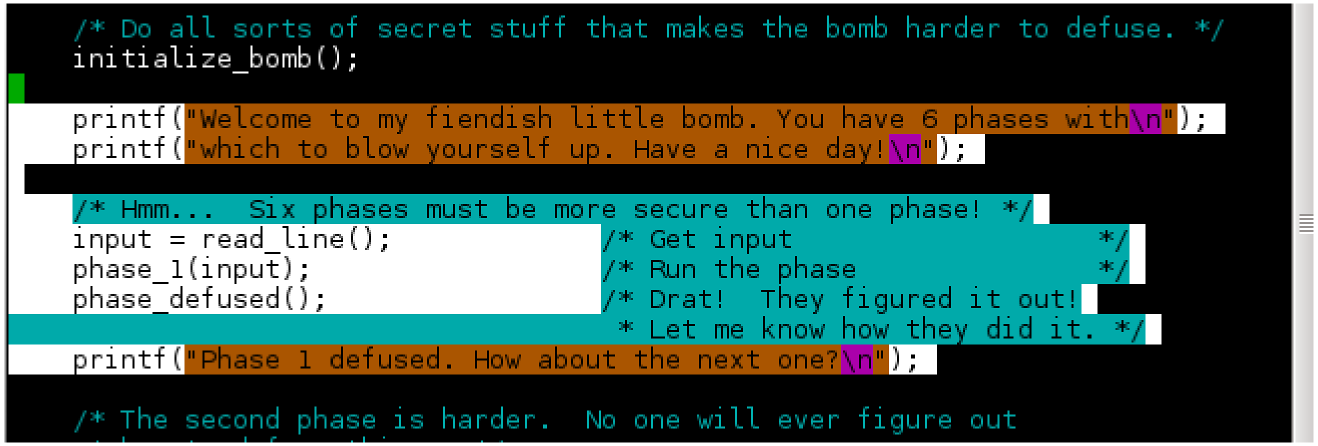
Inspecting the Bomb's Source Code

- 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



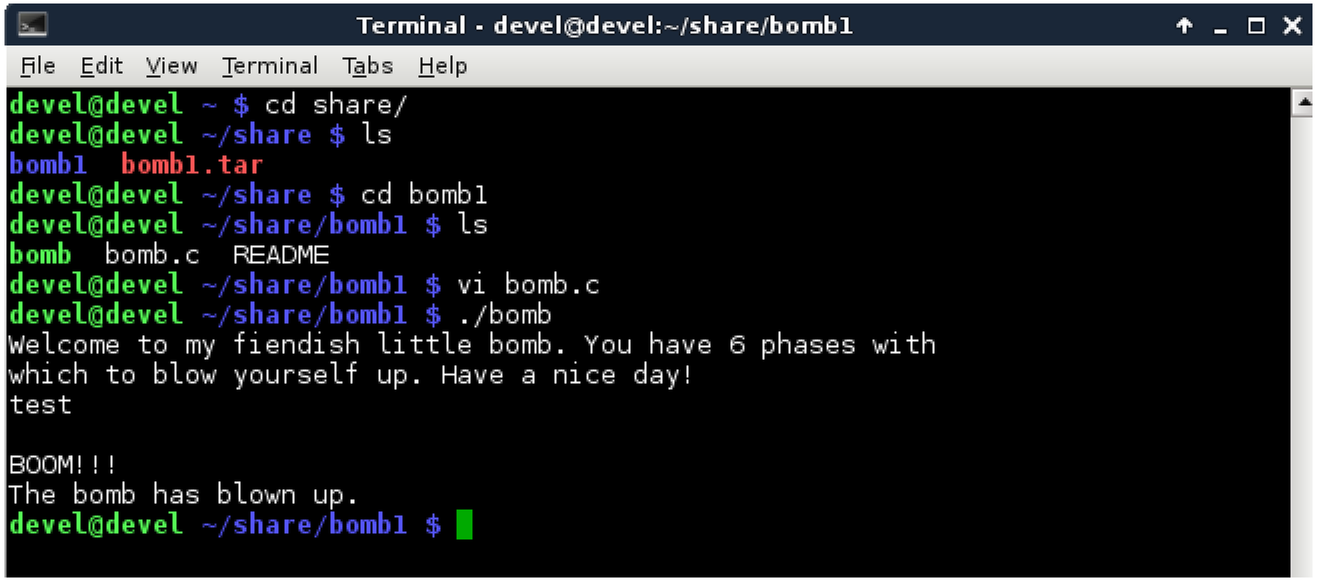
```
    /* Do all sorts of secret stuff that makes the bomb harder to defuse. */
    initialize_bomb();

    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().
- Hint: quit vi by entering ":q" + <Enter>. If that doesn't work, hit <Esc> a couple of times and try entering ":q" + <Enter> again.

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":

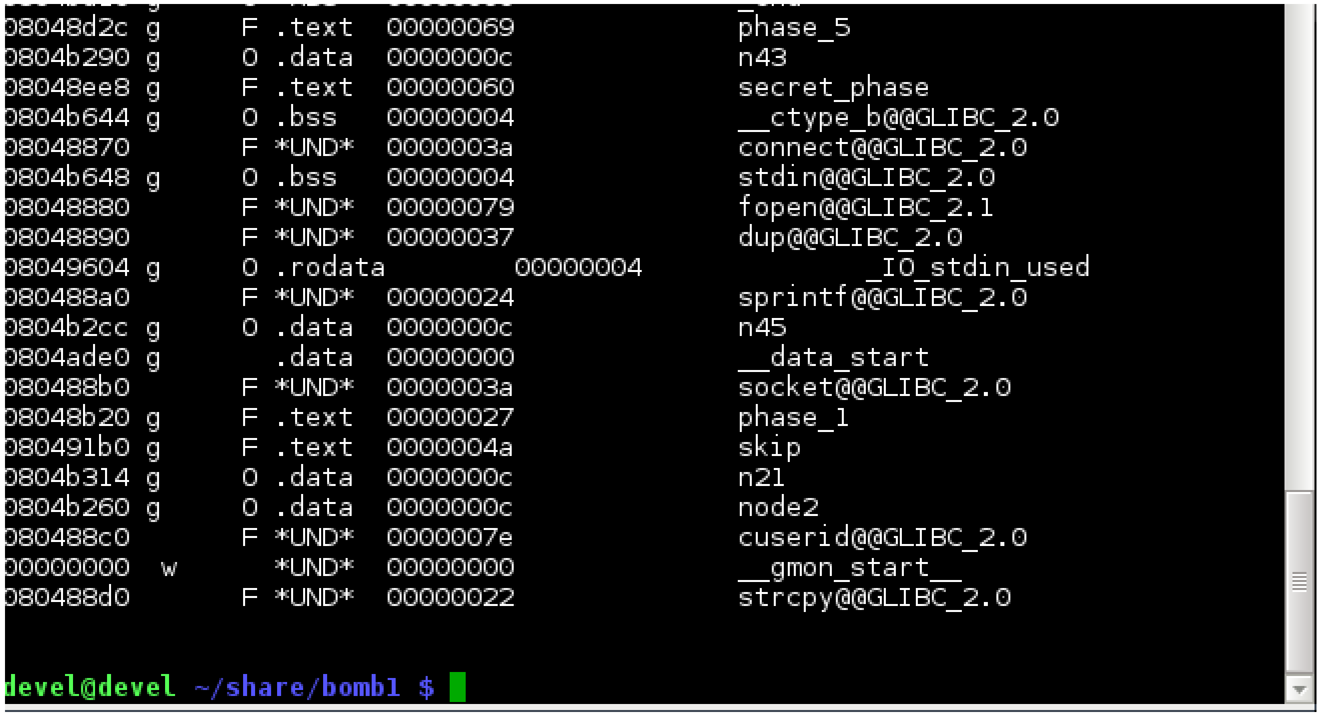

Terminal - devel@devel:~/share/bomb1
File Edit View Terminal Tabs Help

```
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

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.
  - print the symbol table with objdump –t bomb



```
08048d2c g     F .text  00000069              phase_5
0804b290 g     O .data  0000000c              n43
08048ee8 g     F .text  00000060              secret_phase
0804b644 g     O .bss   00000004              __ctype_b@@GLIBC_2.0
08048870       F *UND*  0000003a              connect@@GLIBC_2.0
0804b648 g     O .bss   00000004              stdin@@GLIBC_2.0
08048880       F *UND*  00000079              fopen@@GLIBC_2.1
08048890       F *UND*  00000037              dup@@GLIBC_2.0
08049604 g     O .rodata        00000004              _IO_stdin_used
080488a0       F *UND*  00000024              sprintf@@GLIBC_2.0
0804b2cc g     O .data  0000000c              n45
0804ade0 g       .data  00000000              __data_start
080488b0       F *UND*  0000003a              socket@@GLIBC_2.0
08048b20 g     F .text  00000027              phase_1
080491b0 g     F .text  0000004a              skip
0804b314 g     O .data  0000000c              n21
0804b260 g     O .data  0000000c              node2
080488c0       F *UND*  0000007e              cuserid@@GLIBC_2.0
00000000  w      *UND*  00000000              __gmon_start__
080488d0       F *UND*  00000022              strcpy@@GLIBC_2.0

devel@devel ~/share/bomb1 $
```

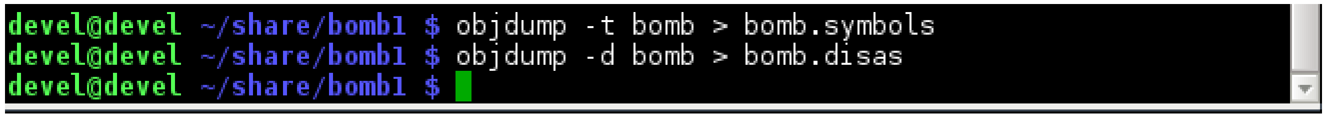
Disassembling the Bomb using objdump

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

    objdump –t bomb > bomb.sysbols

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

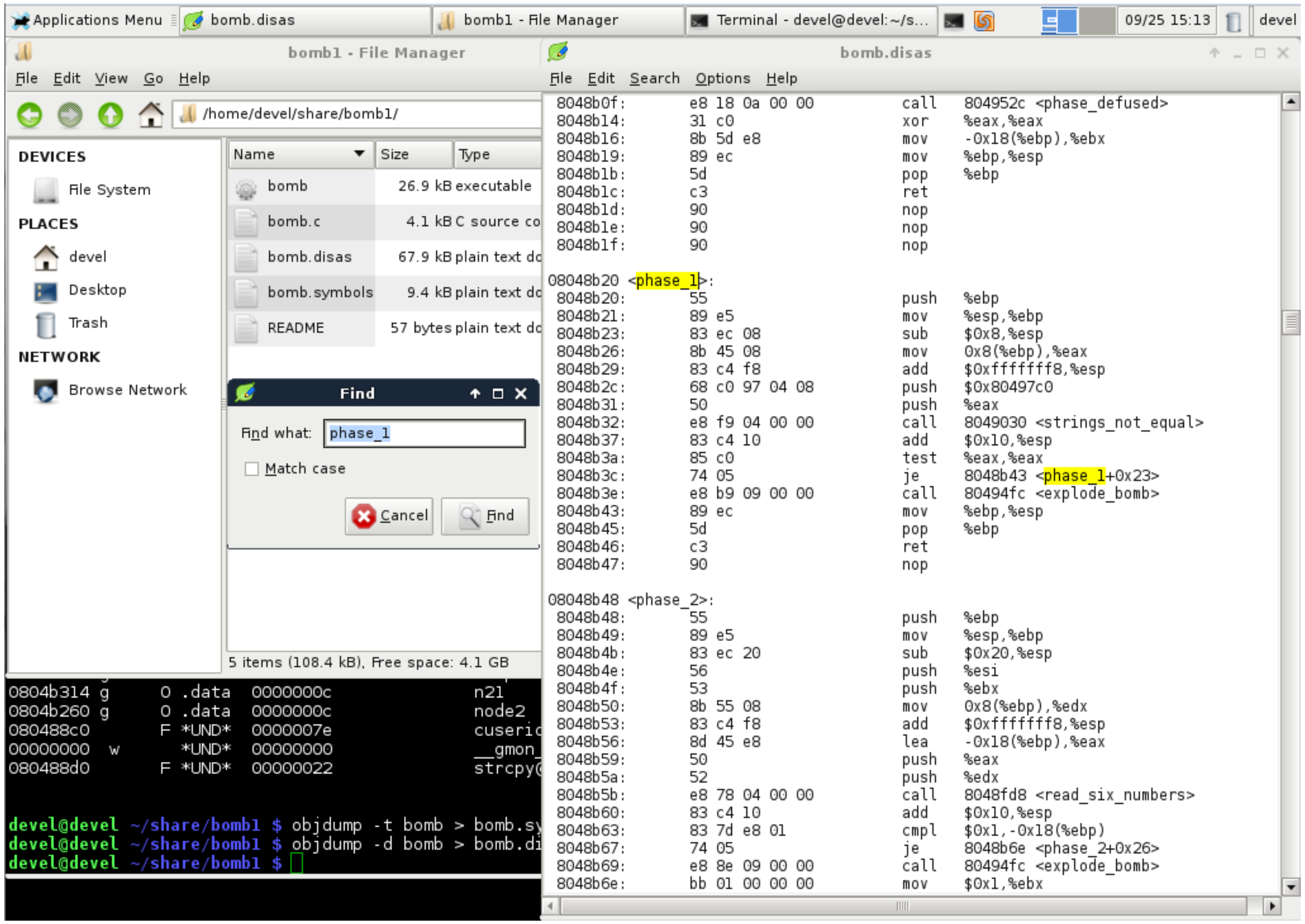
    objdump –d bomb > bomb.disas



```
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()**



/home/devel/share/bomb1/

```
Name            Size      Type
bomb            26.9 kB   executable
bomb.c          4.1 kB    C source co
bomb.disas      67.9 kB   plain text do
bomb.symbols    9.4 kB    plain text do
README          57 bytes  plain text do
```

5 items (108.4 kB), Free space: 4.1 GB

Find what: phase_1

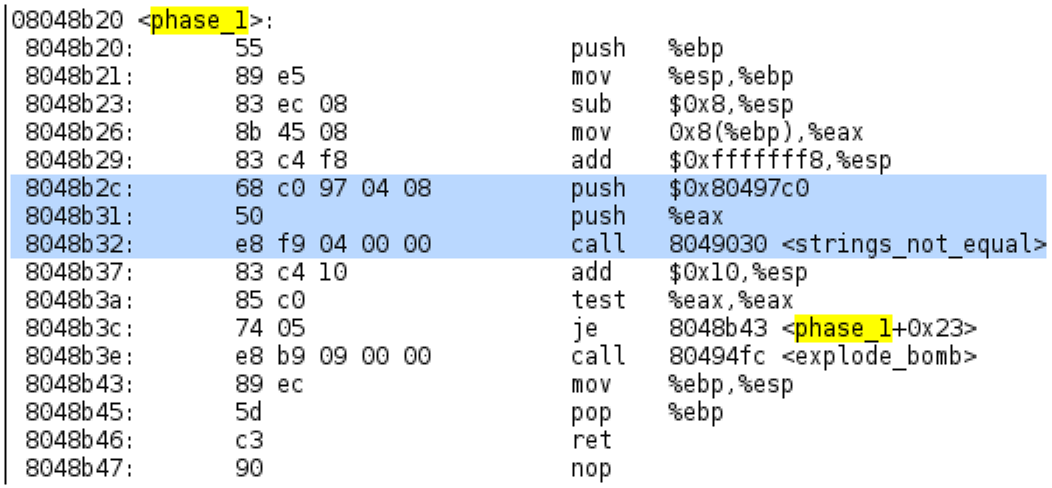
☐ Match case

```
0804b314 g     O .data  0000000c              n21
0804b260 g     O .data  0000000c              node2
080488c0       F *UND*  0000007e              cuseri
00000000  w      *UND*  00000000              __gmon
080488d0       F *UND*  00000022              strcpy(

devel@devel ~/share/bomb1 $ objdump -t bomb > bomb.sy
devel@devel ~/share/bomb1 $ objdump -d bomb > bomb.di
devel@devel ~/share/bomb1 $ 
```

```
 8048b0f:       e8 18 0a 00 00          call   804952c <phase_defused>
 8048b14:       31 c0                   xor    %eax,%eax
 8048b16:       8b 5d e8                mov    -0x18(%ebp),%ebx
 8048b19:       89 ec                   mov    %ebp,%esp
 8048b1b:       5d                      pop    %ebp
 8048b1c:       c3                      ret
 8048b1d:       90                      nop
 8048b1e:       90                      nop
 8048b1f:       90                      nop

08048b20 <phase_1>:
 8048b20:       55                      push   %ebp
 8048b21:       89 e5                   mov    %esp,%ebp
 8048b23:       83 ec 08                sub    $0x8,%esp
 8048b26:       8b 45 08                mov    0x8(%ebp),%eax
 8048b29:       83 c4 f8                add    $0xfffffff8,%esp
 8048b2c:       68 c0 97 04 08          push   $0x80497c0
 8048b31:       50                      push   %eax
 8048b32:       e8 f9 04 00 00          call   8049030 <strings_not_equal>
 8048b37:       83 c4 10                add    $0x10,%esp
 8048b3a:       85 c0                   test   %eax,%eax
 8048b3c:       74 05                   je     8048b43 <phase_1+0x23>
 8048b3e:       e8 b9 09 00 00          call   80494fc <explode_bomb>
 8048b43:       89 ec                   mov    %ebp,%esp
 8048b45:       5d                      pop    %ebp
 8048b46:       c3                      ret
 8048b47:       90                      nop

08048b48 <phase_2>:
 8048b48:       55                      push   %ebp
 8048b49:       89 e5                   mov    %esp,%ebp
 8048b4b:       83 ec 20                sub    $0x20,%esp
 8048b4e:       56                      push   %esi
 8048b4f:       53                      push   %ebx
 8048b50:       8b 55 08                mov    0x8(%ebp),%edx
 8048b53:       83 c4 f8                add    $0xfffffff8,%esp
 8048b56:       8d 45 e8                lea    -0x18(%ebp),%eax
 8048b59:       50                      push   %eax
 8048b5a:       52                      push   %edx
 8048b5b:       e8 78 04 00 00          call   8048fd8 <read_six_numbers>
 8048b60:       83 c4 10                add    $0x10,%esp
 8048b63:       83 7d e8 01             cmpl   $0x1,-0x18(%ebp)
 8048b67:       74 05                   je     8048b6e <phase_2+0x26>
 8048b69:       e8 8e 09 00 00          call   80494fc <explode_bomb>
 8048b6e:       bb 01 00 00 00          mov    $0x1,%ebx
```

Inspecting the code of phase_1()

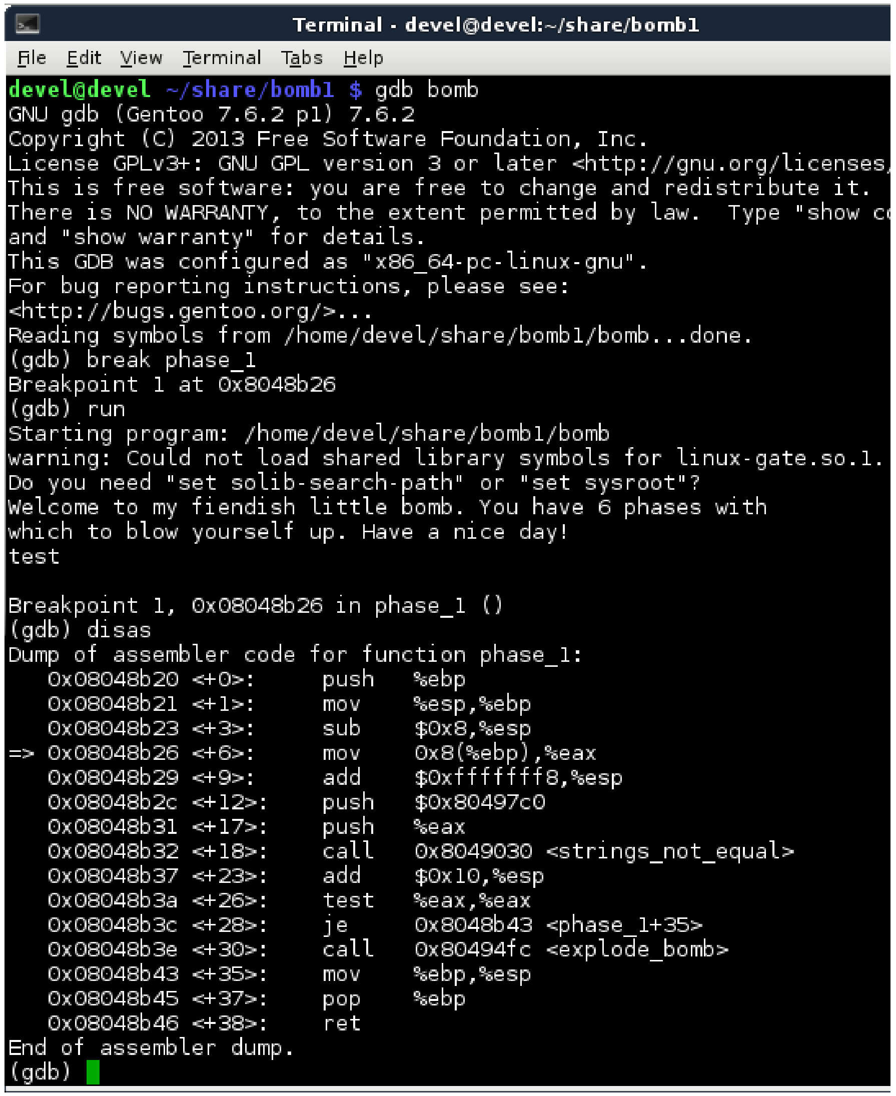
- From the code we can see that:
- **phase_1** calls a function called **strings_not_equal()** with two arguments (it pushes two values on the stack)
- then, depending on the result of **strings_not_equal()** in register %eax either calls **explode_bomb()** or returns.



```
08048b20 <phase_1>:
 8048b20:       55                      push   %ebp
 8048b21:       89 e5                   mov    %esp,%ebp
 8048b23:       83 ec 08                sub    $0x8,%esp
 8048b26:       8b 45 08                mov    0x8(%ebp),%eax
 8048b29:       83 c4 f8                add    $0xfffffff8,%esp
 8048b2c:       68 c0 97 04 08          push   $0x80497c0
 8048b31:       50                      push   %eax
 8048b32:       e8 f9 04 00 00          call   8049030 <strings_not_equal>
 8048b37:       83 c4 10                add    $0x10,%esp
 8048b3a:       85 c0                   test   %eax,%eax
 8048b3c:       74 05                   je     8048b43 <phase_1+0x23>
 8048b3e:       e8 b9 09 00 00          call   80494fc <explode_bomb>
 8048b43:       89 ec                   mov    %ebp,%esp
 8048b45:       5d                      pop    %ebp
 8048b46:       c3                      ret
 8048b47:       90                      nop
```

Debugging the Bomb in gdb

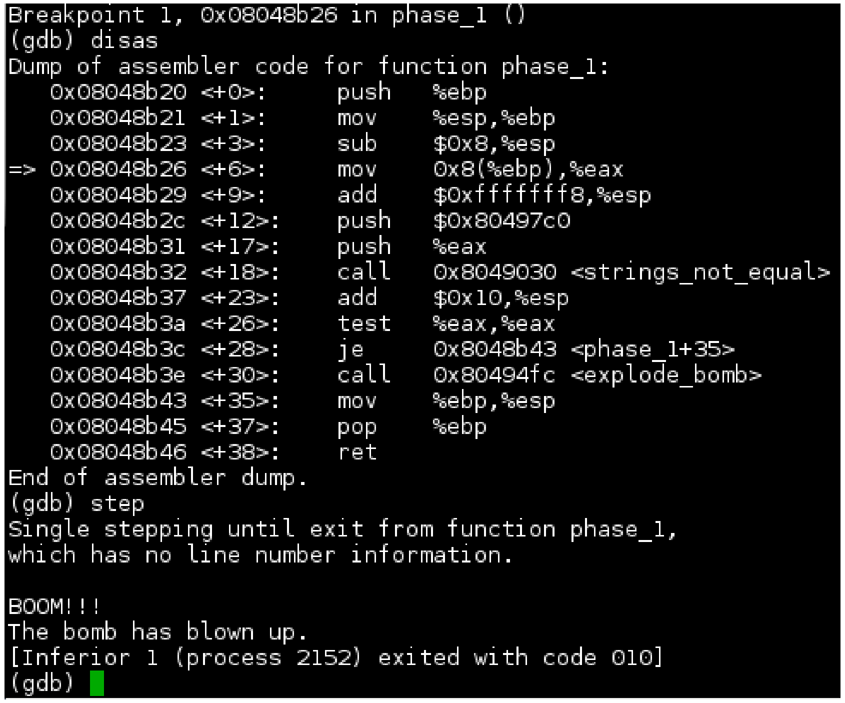
- With this knowledge we now run the
- bomb in the GNU debugger
  - go back to the terminal and
- execute **gdb bomb**
  - set a breakpoint at phase_1 by
- entering **break phase_1**
  - run the bomb by entering **run**
  - enter the first string and hit enter
  - now **gdb** stops at the entry of phase_1 (disassemble with disas)


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

```
devel@devel ~/share/bomb1 $ gdb bomb
GNU gdb (Gentoo 7.6.2 p1) 7.6.2
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses,
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show co
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.gentoo.org/>...
Reading symbols from /home/devel/share/bomb1/bomb...done.
(gdb) break phase_1
Breakpoint 1 at 0x8048b26
(gdb) run
Starting program: /home/devel/share/bomb1/bomb
warning: Could not load shared library symbols for linux-gate.so.1.
Do you need "set solib-search-path" or "set sysroot"?
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
test

Breakpoint 1, 0x08048b26 in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
   0x08048b20 <+0>:     push   %ebp
   0x08048b21 <+1>:     mov    %esp,%ebp
   0x08048b23 <+3>:     sub    $0x8,%esp
=> 0x08048b26 <+6>:     mov    0x8(%ebp),%eax
   0x08048b29 <+9>:     add    $0xfffffff8,%esp
   0x08048b2c <+12>:    push   $0x80497c0
   0x08048b31 <+17>:    push   %eax
   0x08048b32 <+18>:    call   0x8049030 <strings_not_equal>
   0x08048b37 <+23>:    add    $0x10,%esp
   0x08048b3a <+26>:    test   %eax,%eax
   0x08048b3c <+28>:    je     0x8048b43 <phase_1+35>
   0x08048b3e <+30>:    call   0x80494fc <explode_bomb>
   0x08048b43 <+35>:    mov    %ebp,%esp
   0x08048b45 <+37>:    pop    %ebp
   0x08048b46 <+38>:    ret
End of assembler dump.
(gdb)
```

Stepping through the Code

- The command **step** executes the C code line-by-line

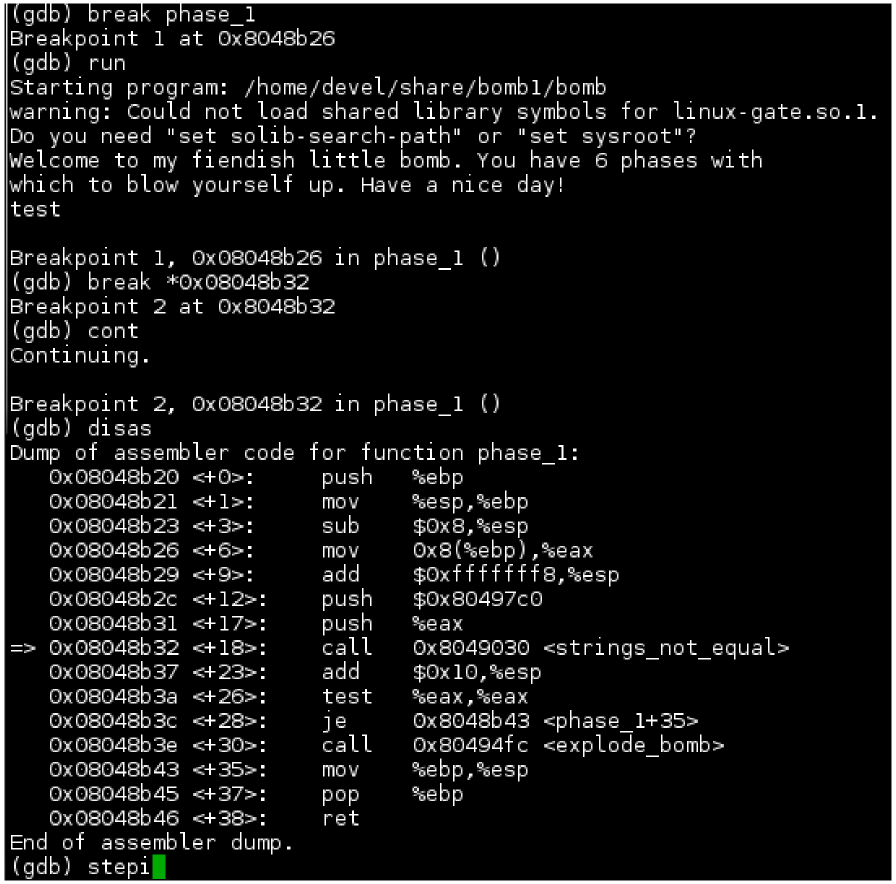


```
Breakpoint 1, 0x08048b26 in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
   0x08048b20 <+0>:     push   %ebp
   0x08048b21 <+1>:     mov    %esp,%ebp
   0x08048b23 <+3>:     sub    $0x8,%esp
=> 0x08048b26 <+6>:     mov    0x8(%ebp),%eax
   0x08048b29 <+9>:     add    $0xfffffff8,%esp
   0x08048b2c <+12>:    push   $0x80497c0
   0x08048b31 <+17>:    push   %eax
   0x08048b32 <+18>:    call   0x8049030 <strings_not_equal>
   0x08048b37 <+23>:    add    $0x10,%esp
   0x08048b3a <+26>:    test   %eax,%eax
   0x08048b3c <+28>:    je     0x8048b43 <phase_1+35>
   0x08048b3e <+30>:    call   0x80494fc <explode_bomb>
   0x08048b43 <+35>:    mov    %ebp,%esp
   0x08048b45 <+37>:    pop    %ebp
   0x08048b46 <+38>:    ret
End of assembler dump.
(gdb) step
Single stepping until exit from function phase_1,
which has no line number information.

BOOM!!!
The bomb has blown up.
[Inferior 1 (process 2152) exited with code 010]
(gdb)
```

- the C code for phase_1 is not available, so gdb executed the function phase_1 until the end
  - not really what we wanted…

Stepping through the Code

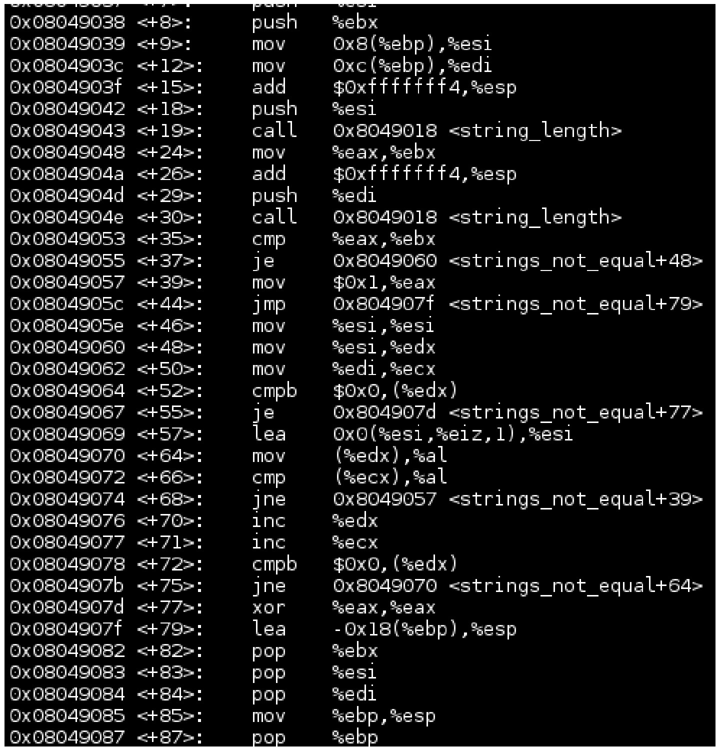
- We can set more breakpoints and continue execution until the next breakpoint is reached. Looking at the code, a breakpoint at address **0x08048b32 call 0x8049030 <strings_not_equal>** seems reasonable.
  - breakpoints to addresses are set by entering **break *<address>**
  - continue execution to the 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
  - **stepi**: step through the program one (machine) instruction exactly



```
(gdb) break phase_1
Breakpoint 1 at 0x8048b26
(gdb) run
Starting program: /home/devel/share/bomb1/bomb
warning: Could not load shared library symbols for linux-gate.so.1.
Do you need "set solib-search-path" or "set sysroot"?
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
test

Breakpoint 1, 0x08048b26 in phase_1 ()
(gdb) break *0x08048b32
Breakpoint 2 at 0x8048b32
(gdb) cont
Continuing.

Breakpoint 2, 0x08048b32 in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
   0x08048b20 <+0>:     push   %ebp
   0x08048b21 <+1>:     mov    %esp,%ebp
   0x08048b23 <+3>:     sub    $0x8,%esp
   0x08048b26 <+6>:     mov    0x8(%ebp),%eax
   0x08048b29 <+9>:     add    $0xfffffff8,%esp
   0x08048b2c <+12>:    push   $0x80497c0
   0x08048b31 <+17>:    push   %eax
=> 0x08048b32 <+18>:    call   0x8049030 <strings_not_equal>
   0x08048b37 <+23>:    add    $0x10,%esp
   0x08048b3a <+26>:    test   %eax,%eax
   0x08048b3c <+28>:    je     0x8048b43 <phase_1+35>
   0x08048b3e <+30>:    call   0x80494fc <explode_bomb>
   0x08048b43 <+35>:    mov    %ebp,%esp
   0x08048b45 <+37>:    pop    %ebp
   0x08048b46 <+38>:    ret
End of assembler dump.
(gdb) stepi
```

Inspecting Registers and Memory

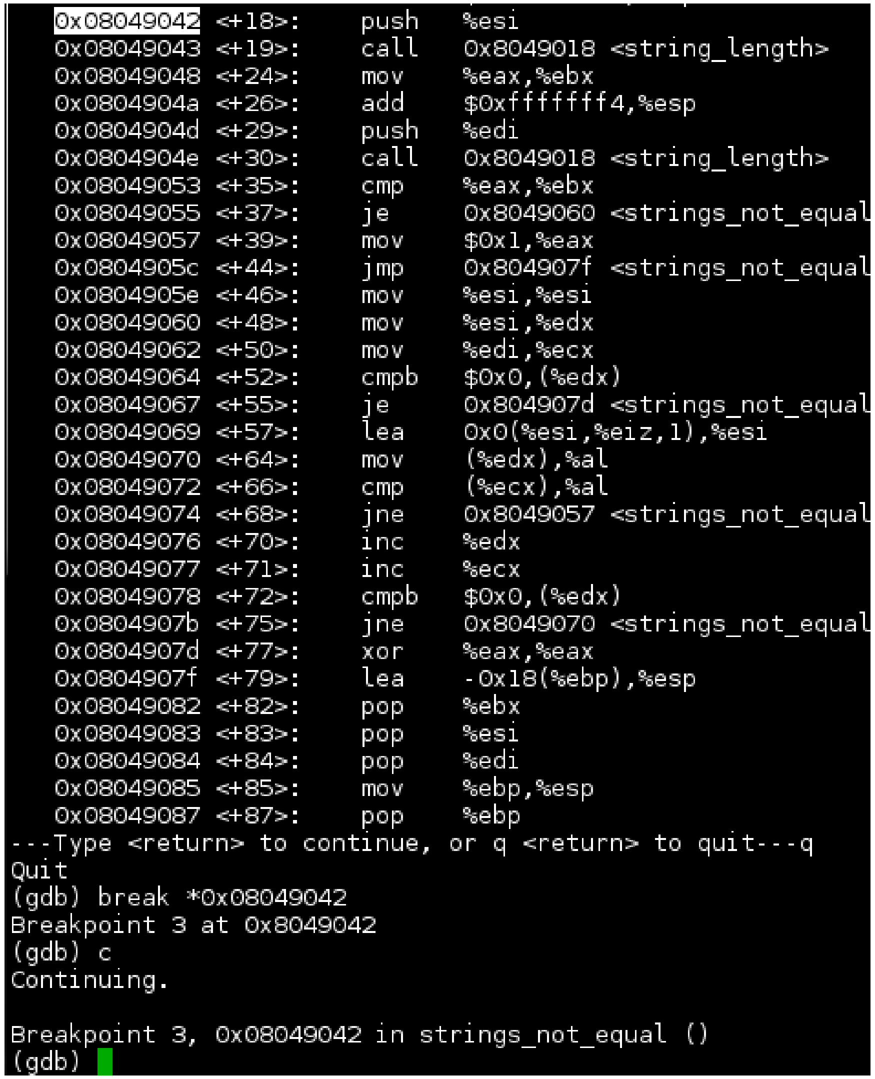
- After executing **stepi** at the call to **strings_not_equal,** enter **disas** again to see where we currently are
  - as expected, the debugger stopped at the first instruction of **strings_not_equal**
  - looking at the code, we see that the function loads the two arguments from the stack into registers %esi and %edi
  - 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 **(0x8049043, 0x804904e)** and then compares their length (**0x8049053**). If they are not equal, it sets the result to false and exits(**0x804905c**). If they are equal, it starts comparing the strings character by character (**0x8049072)** until the characters differ (**0x8049074**) or the end of the string is reached (**0x8049078**).



```
0x08049038 <+8>:    push   %ebx
0x08049039 <+9>:    mov    0x8(%ebp),%esi
0x0804903c <+12>:   mov    0xc(%ebp),%edi
0x0804903f <+15>:   add    $0xfffffff4,%esp
0x08049042 <+18>:   push   %esi
0x08049043 <+19>:   call   0x8049018 <string_length>
0x08049048 <+24>:   mov    %eax,%ebx
0x0804904a <+26>:   add    $0xfffffff4,%esp
0x0804904d <+29>:   push   %edi
0x0804904e <+30>:   call   0x8049018 <string_length>
0x08049053 <+35>:   cmp    %eax,%ebx
0x08049055 <+37>:   je     0x8049060 <strings_not_equal+48>
0x08049057 <+39>:   mov    $0x1,%eax
0x0804905c <+44>:   jmp    0x804907f <strings_not_equal+79>
0x0804905e <+46>:   mov    %esi,%esi
0x08049060 <+48>:   mov    %esi,%edx
0x08049062 <+50>:   mov    %edi,%ecx
0x08049064 <+52>:   cmpb   $0x0,(%edx)
0x08049067 <+55>:   je     0x804907d <strings_not_equal+77>
0x08049069 <+57>:   lea    0x0(%esi,%eiz,1),%esi
0x08049070 <+64>:   mov    (%edx),%al
0x08049072 <+66>:   cmp    (%ecx),%al
0x08049074 <+68>:   jne    0x8049057 <strings_not_equal+39>
0x08049076 <+70>:   inc    %edx
0x08049077 <+71>:   inc    %ecx
0x08049078 <+72>:   cmpb   $0x0,(%edx)
0x0804907b <+75>:   jne    0x8049070 <strings_not_equal+64>
0x0804907d <+77>:   xor    %eax,%eax
0x0804907f <+79>:   lea    -0x18(%ebp),%esp
0x08049082 <+82>:   pop    %ebx
0x08049083 <+83>:   pop    %esi
0x08049084 <+84>:   pop    %edi
0x08049085 <+85>:   mov    %ebp,%esp
0x08049087 <+87>:   pop    %ebp
```

Inspecting Register and Memory

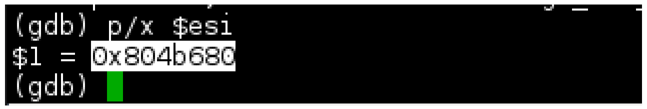
- With this knowledge, we now want to inspect those two strings. The arguments to the function are loaded into registers by the two **mov** instructions at **0x8049039**. We thus want to stop after they have been executed. You can either use **stepi** to reach that location or set another breakpoint at the instruction following the two **movs (0x8049042)** and then continue.



```
   0x08049042 <+18>:    push   %esi
   0x08049043 <+19>:    call   0x8049018 <string_length>
   0x08049048 <+24>:    mov    %eax,%ebx
   0x0804904a <+26>:    add    $0xfffffff4,%esp
   0x0804904d <+29>:    push   %edi
   0x0804904e <+30>:    call   0x8049018 <string_length>
   0x08049053 <+35>:    cmp    %eax,%ebx
   0x08049055 <+37>:    je     0x8049060 <strings_not_equal
   0x08049057 <+39>:    mov    $0x1,%eax
   0x0804905c <+44>:    jmp    0x804907f <strings_not_equal
   0x0804905e <+46>:    mov    %esi,%esi
   0x08049060 <+48>:    mov    %esi,%edx
   0x08049062 <+50>:    mov    %edi,%ecx
   0x08049064 <+52>:    cmpb   $0x0,(%edx)
   0x08049067 <+55>:    je     0x804907d <strings_not_equal
   0x08049069 <+57>:    lea    0x0(%esi,%eiz,1),%esi
   0x08049070 <+64>:    mov    (%edx),%al
   0x08049072 <+66>:    cmp    (%ecx),%al
   0x08049074 <+68>:    jne    0x8049057 <strings_not_equal
   0x08049076 <+70>:    inc    %edx
   0x08049077 <+71>:    inc    %ecx
   0x08049078 <+72>:    cmpb   $0x0,(%edx)
   0x0804907b <+75>:    jne    0x8049070 <strings_not_equal
   0x0804907d <+77>:    xor    %eax,%eax
   0x0804907f <+79>:    lea    -0x18(%ebp),%esp
   0x08049082 <+82>:    pop    %ebx
   0x08049083 <+83>:    pop    %esi
   0x08049084 <+84>:    pop    %edi
   0x08049085 <+85>:    mov    %ebp,%esp
   0x08049087 <+87>:    pop    %ebp
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) break *0x08049042
Breakpoint 3 at 0x8049042
(gdb) c
Continuing.

Breakpoint 3, 0x08049042 in strings_not_equal ()
(gdb)
```

Inspecting Register and Memory

- Once we are there, let’s first print the contents of the two registers
  - Use **p/x $<reg>** to print the contents of a register in hexadecimal form

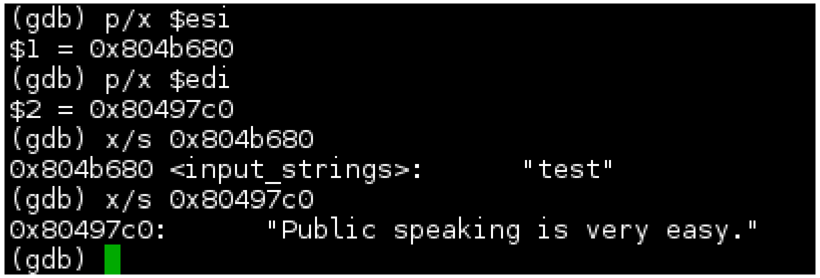


```
(gdb) p/x $esi
$1 = 0x804b680
(gdb)
```

  - 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** do get help on the different options to this function)

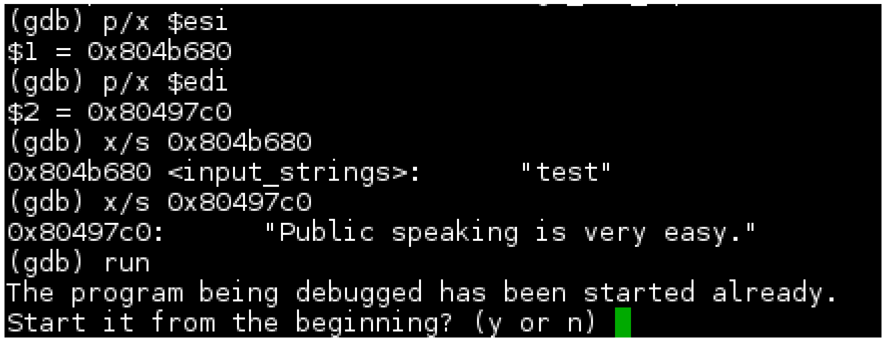


```
(gdb) p/x $esi
$1 = 0x804b680
(gdb) p/x $edi
$2 = 0x80497c0
(gdb) x/s 0x804b680
0x804b680 <input_strings>:      "test"
(gdb) x/s 0x80497c0
0x80497c0:      "Public speaking is very easy."
(gdb) 
```

- Indeed, we see the input string (**"test")** as well as another string **("Verbosity leads to unclear, inarticulate things.")**
- Could this be the passphrase for phase 1?

Restarting the Program from the Beginning

- Let’s check if the second string is indeed the correct string 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.

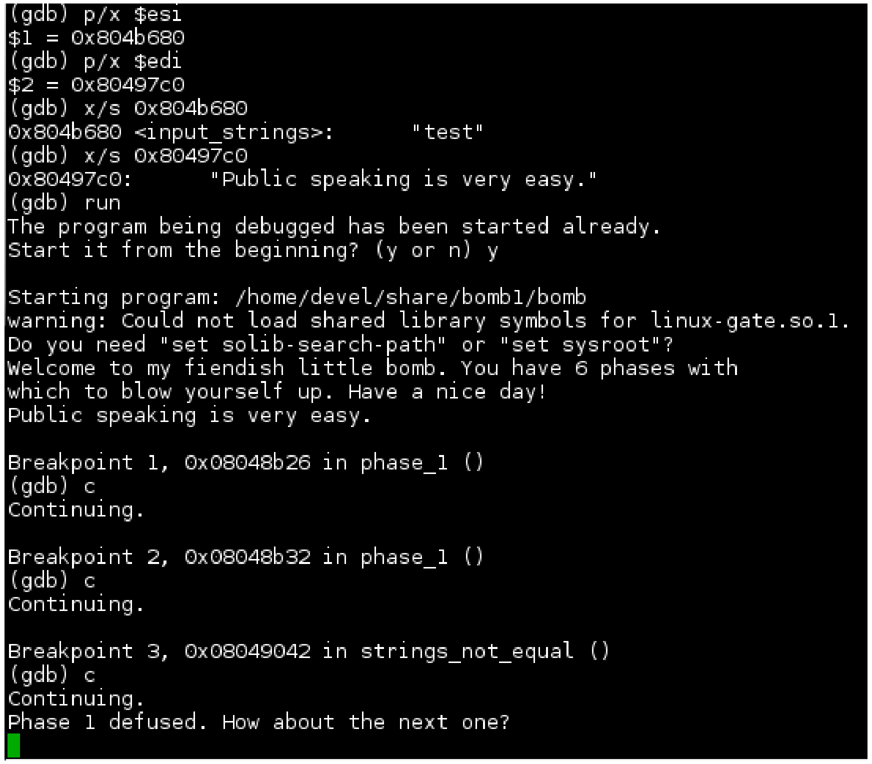


```
(gdb) p/x $esi
$1 = 0x804b680
(gdb) p/x $edi
$2 = 0x80497c0
(gdb) x/s 0x804b680
0x804b680 <input_strings>:      "test"
(gdb) x/s 0x80497c0
0x80497c0:      "Public speaking is very easy."
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

- Confirm with “y”

Restarting the Program from the Beginning

- The program restarts and asks for the passphrase again. Copy-paste (mark with the mouse, then middle-click) and hit enter.
- The program stops at all breakpoints, we are impatient and want to continue
- Indeed, we have defused the first stage and the bomb asks us for the second passphrase!



```
(gdb) p/x $esi
$1 = 0x804b680
(gdb) p/x $edi
$2 = 0x80497c0
(gdb) x/s 0x804b680
0x804b680 <input_strings>:      "test"
(gdb) x/s 0x80497c0
0x80497c0:      "Public speaking is very easy."
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/devel/share/bomb1/bomb
warning: Could not load shared library symbols for linux-gate.so.1.
Do you need "set solib-search-path" or "set sysroot"?
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, 0x08048b26 in phase_1 ()
(gdb) c
Continuing.

Breakpoint 2, 0x08048b32 in phase_1 ()
(gdb) c
Continuing.

Breakpoint 3, 0x08049042 in strings_not_equal ()
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
```

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:65421/scoreboard

Good Luck!