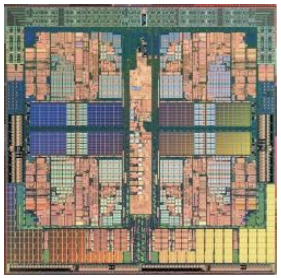


Computer Architecture Lab.

Lab 2
Bomblab

CS230 System Programming

Sep 26, 2016
Sep 29, 2016

Contents

- Overview
- Tools Introduction
  - SCP
  - GDB
- Precautions
  - Running a bomb
  - Skipping defused phases
  - Grading

Overview

- You should defuse ‘a bomb’
- Each bomb has 6 stages + 1 hidden stage
- No pre-provided hint: You can only use a debugger to defuse the bomb!
- The purpose of this assignment
  - To understand assembly
  - To be familiar with gdb (command interface debugger)

Tools Instruction: SCP

- You can upload your bomb by using SCP client
- There are many SCP clients
  - PSCP
  - WinSCP (recommended)

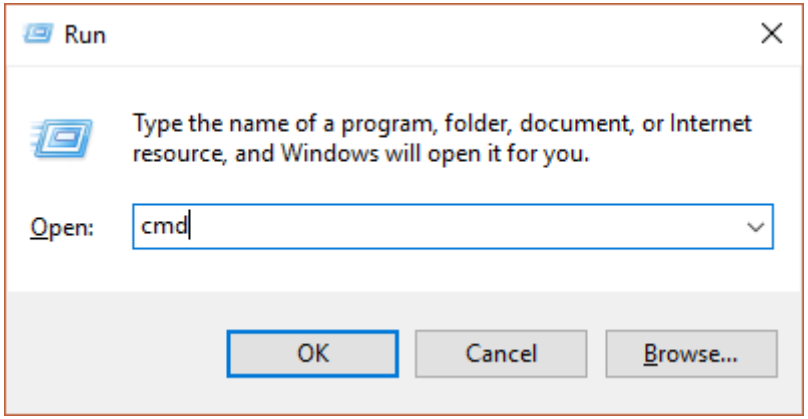
- For Linux or Mac users, you can upload your bomb by using SCP command
  - scp [file name] [user name]@[server ip]:[path]
  - Please check Linux crash course slide

PSCP

- How to use PSCP
  - Download
  - Open command line interface window
  - Change directory where PSCP is located
  - Send a file by using PSCP command
    (pscp [file name] [user name]@[server ip]:[path])

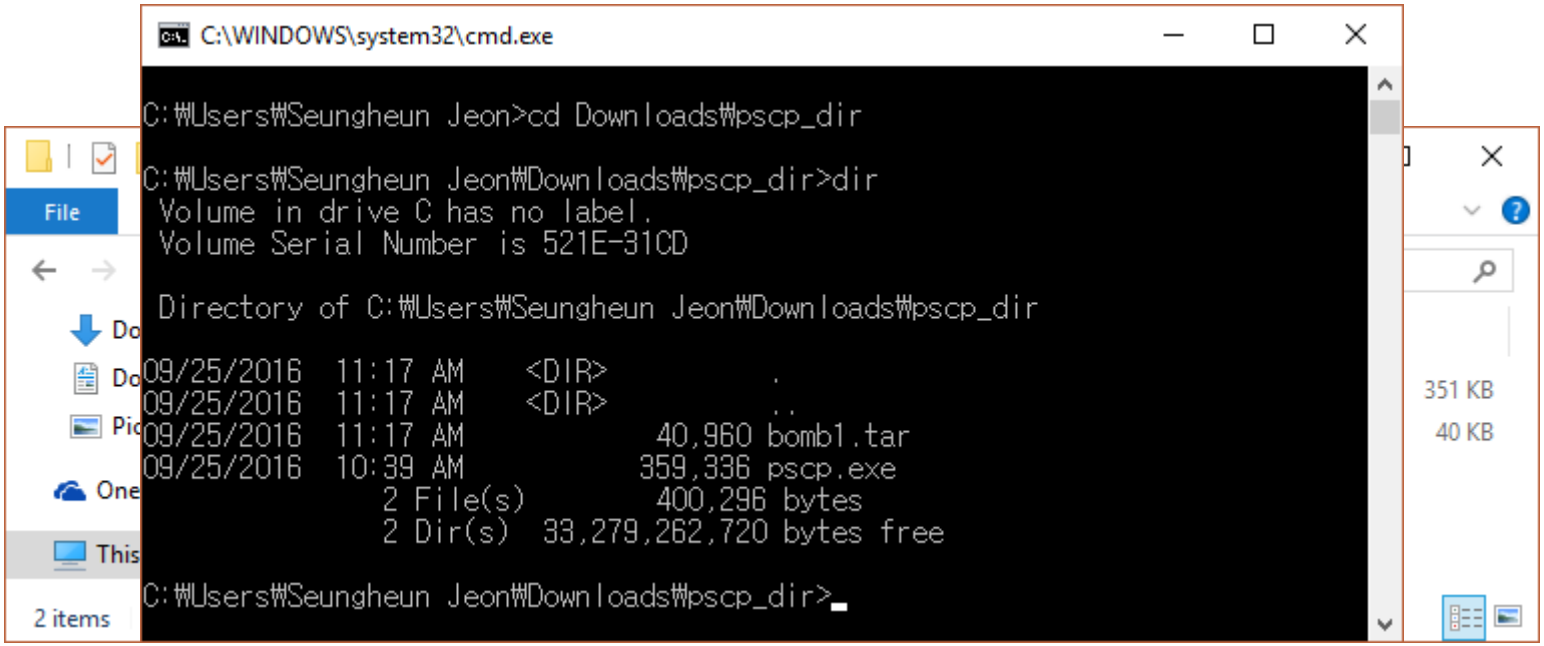
PSCP

- **Exercise**
  - **Download**
  - **Open command line interface window**
    - **Win + r (shortcut to open execution window)**
    - **Type “cmd” then enter**



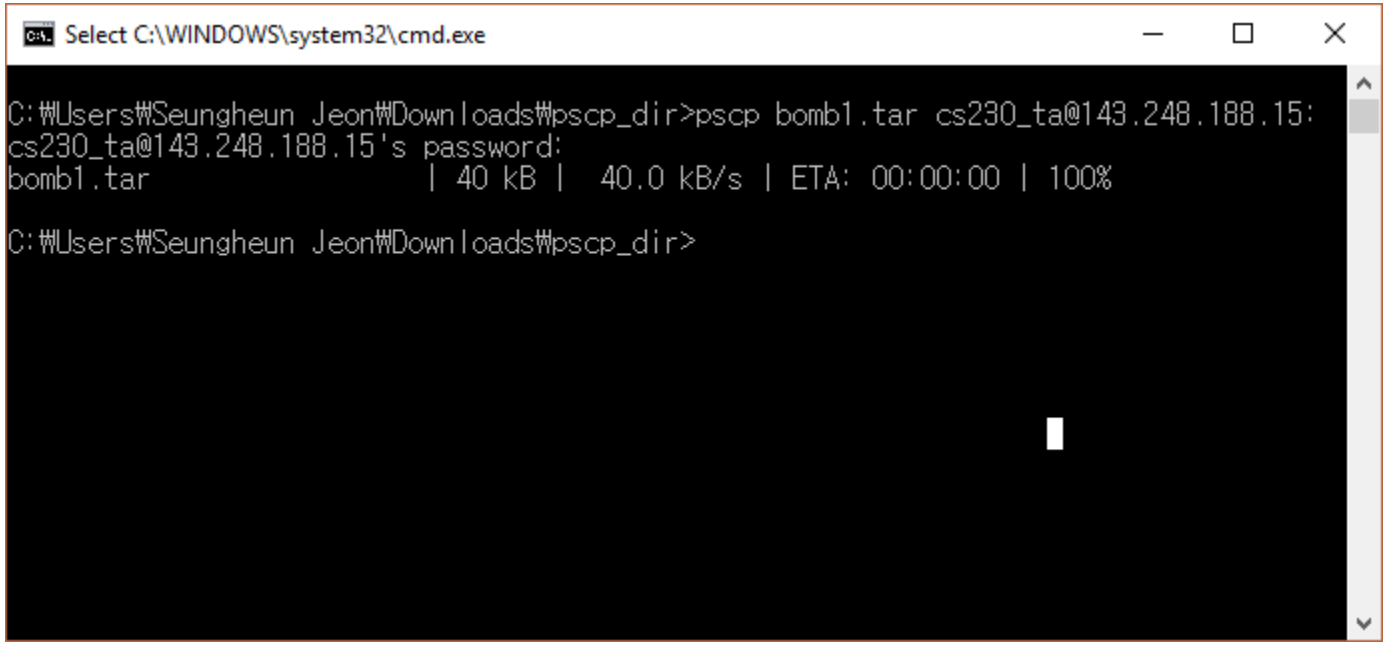
PSCP

- **Exercise**
  - **Change directory where PSCP is located**
    - **In my case, the path is “Downloads₩pscp_dir”**
    - **Type “cd Downloads₩pscp_dir”**
    - **Caution: In Windows machine, you have to use ‘₩’**



PSCP

- **Exercise**
  - **Send a file by using PSCP command**
    - **pscp [file name] [user name]@[server ip]:[path]**
  - **If you want to send “bomb1.tar” to “cs230_ta” whose server IP is “143.248.188.15”**
    - **pscp bomb1.tar cs230_ta@143.248.188.15:**



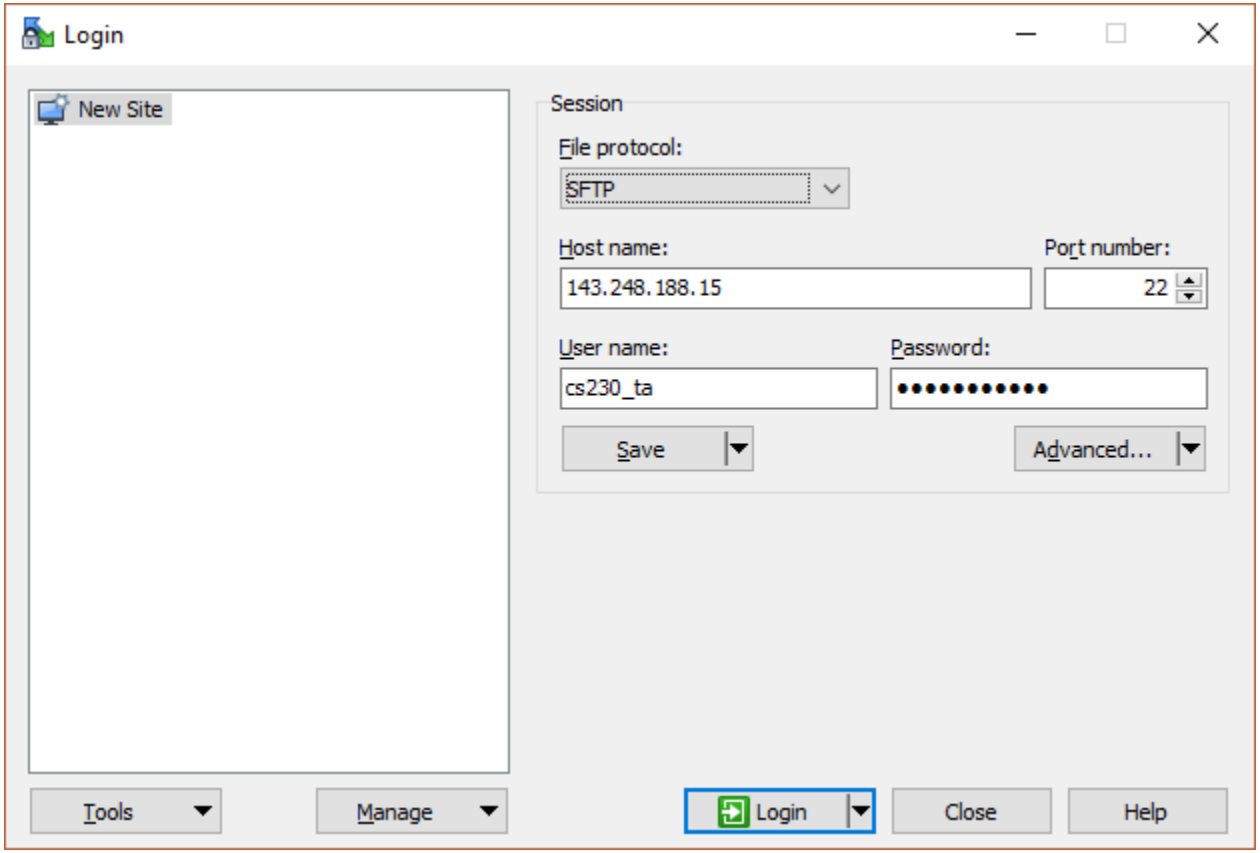
WinSCP

- **WinSCP is much easier than PSCP**

- **How to use WinSCP**
  - **Download it (I recommend you to use portable version)**
  - **Execute it**
  - **Enter your server ip, user name and password**
  - **Click login button**
  - **Transfer your file just by dragging**

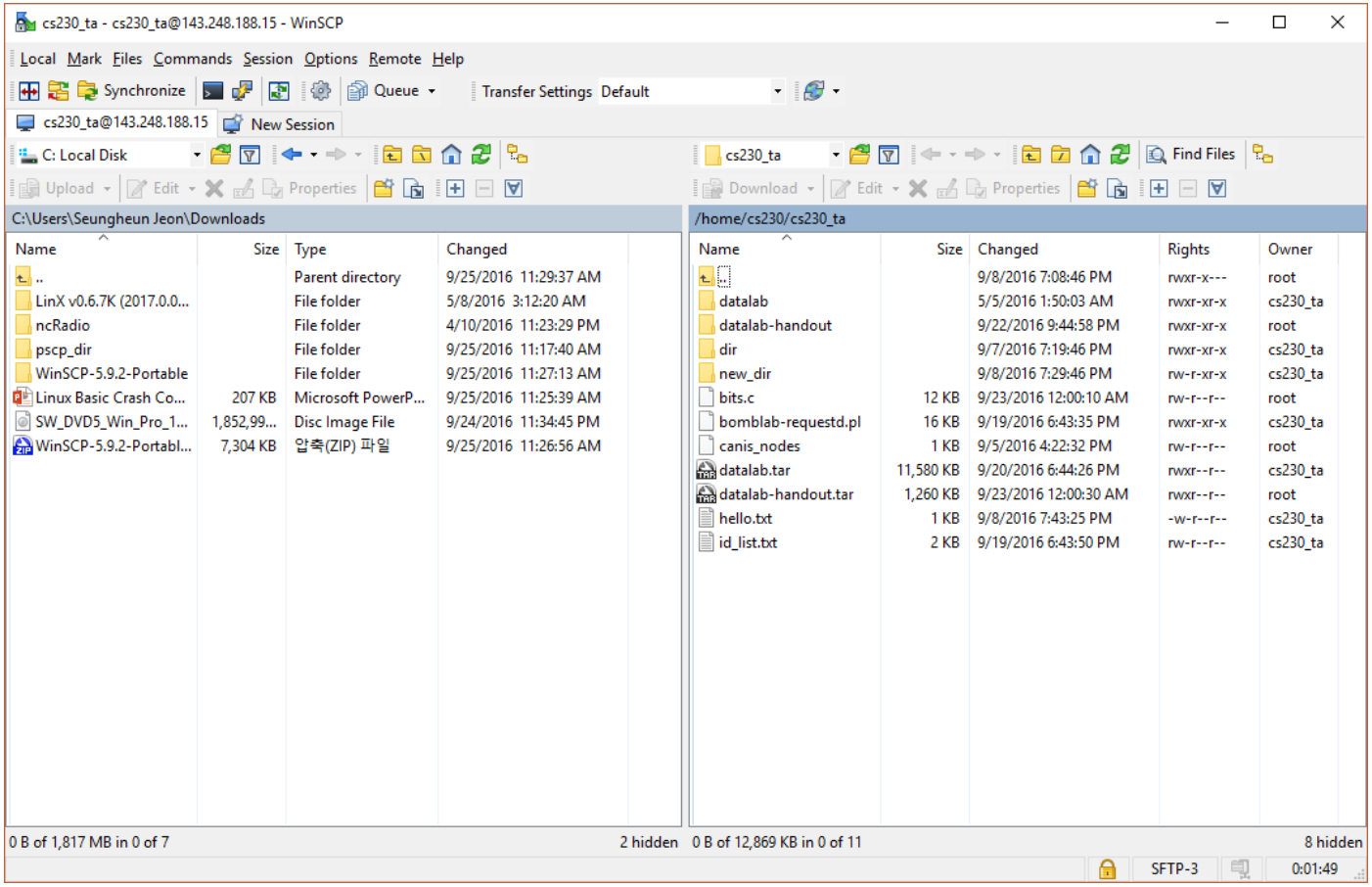
WinSCP

- **Exercise**
  - **Enter your server ip, user name and password**
    - **Use default option for protocol and port number field**



WinSCP

- **Exercise**
  - **Left half display files in Window machine**
  - **Right half display files in your remote Linux machine**



SCP command

- If you are Linux or Mac user, you can use SCP command to transfer files to remote Linux machine

- Exercise
  - Open your terminal
  - Type SCP command
    - scp [file name] [user name]@[server ip]:[path]
  - If you want to send “bomb1.tar” to “cs230_ta” whose server IP is “143.248.188.15”

```
➜  ~ ls
bomb1.tar  CS492      Downloads         mongo    nutcracker-0.4.1  Templates        YCSB-C
CS230      Desktop    examples.desktop  Music    Pictures          thkim_redisRepo
CS402      Documents  gem5-stable       my_work  Public            Videos
➜  ~ scp bomb1.tar cs230_ta@143.248.188.15:~
cs230_ta@143.248.188.15's password:
bomb1.tar                                                     100%  170KB 170.0KB/s   00:00
➜  ~
```

Tools Instruction: objdump

- One kind of GNU binary utilities
- Displays more detail information about object files
- With –d option, it disassembles the object file and **displays the machine instructions**
- e.g.) `$ objdump -d bomb`

```
0000000000400e4d <main>:
  400e4d:   53                      push   %rbx
  400e4e:   83 ff 01                cmp    $0x1,%edi
  400e51:   75 10                   jne    400e63 <main+0x16>
  400e53:   48 8b 05 4e 39 20 00    mov    0x20394e(%rip),%rax        # 6047a8 <stdin@@GLIBC_2.2.5>
  400e5a:   48 89 05 5f 39 20 00    mov    %rax,0x20395f(%rip)        # 6047c0 <infile>
  400e61:   eb 63                   jmp    400ec6 <main+0x79>
  400e63:   48 89 f3                mov    %rsi,%rbx
  400e66:   83 ff 02                cmp    $0x2,%edi
  400e69:   75 3a                   jne    400ea5 <main+0x58>
  400e6b:   48 8b 7e 08             mov    0x8(%rsi),%rdi
  400e6f:   be 64 25 40 00          mov    $0x402564,%esi
  400e74:   e8 57 fe ff ff          callq  400cd0 <fopen@plt>
  400e79:   48 89 05 40 39 20 00    mov    %rax,0x203940(%rip)        # 6047c0 <infile>
  400e80:   48 85 c0                test   %rax,%rax
  400e83:   75 41                   jne    400ec6 <main+0x79>
```

Tools Introduction: GDB

- The GNU Project DeBugger
- You can see what is going on in a program with the debugger
- Line by line data observation, data manipulation, assembly dump, etc

```
cs230_ta@canis08:~$ gdb
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 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".
(gdb)
```

Tools Introduction: GDB

- **Based on Beej's Quick Guide to GDB**

- **After this presentation, You will be able to**
  1. **Compile to use a debugger**
  2. **Start GDB**
  3. **Add breakpoints in a program**
  4. **Continue after breakpoints**
  5. **Examine variables**

1. Compile to Use a Debugger

- `$ gcc -g [source_files] -o [output_name]`
- e.g.) `$ gcc -g test.c -o a.out`
- "`make btest`" you have used is actually same as `gcc -O -Wall -m32 -o btest bits.c btest.c decl.c tests.c`
- You should add **–g** flag to get extra information

```
Reading symbols from a.out...(no debugging symbols
found)...done.
(gdb) list
No symbol table is loaded.  Use the "file" command.
(gdb)
```

GDB without –g flag

```
Reading symbols from a.out...done.
(gdb) list
1       #include <stdio.h>
2
3       int main(){
4           int i=0;
5           int sum=0;
6
7           for(i=0; i<10; i++){
8               sum+=i;
9           }
10
(gdb)
```

GDB with –g flag

2. Start GDB

- `$ gdb [options] [executable-file]`
- e.g.) `$ gdb bomb`
  - bomb is a name of binary file, so you should have a binary file named 'bomb' in the current directory

```
insujang@canis08:~/bomb5$ ls
README  bomb  bomb.c  input  objdumpout
insujang@canis08:~/bomb5$ gdb bomb
```

```
jeffreychang — insujang@canis08: ~ — ssh insujang@cluster.kaist.ac.kr — 79×17
insujang@canis08:~$ gdb a.out
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 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
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Reading symbols from a.out...done.
(gdb)
```

```
Reading symbols
(gdb)
```

You can type any commands in gdb shell

2. Start GDB

- See the source code
- `(gdb) list`
- You should use –g flag when compile it to see

jeffreychang — insujang@canis08: ~ — ssh insujang@cluster.kaist.ac.kr — 79×17

```
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 a.out...done.
(gdb) list
1       #include <stdio.h>
2
3       int main(){
4           int i=0;
5           int sum=0;
6
7           for(i=0; i<10; i++){
8               sum+=i;
9           }
10
(gdb)
```

2. Start GDB

- **Running a program**
- **(gdb) r or run**

jeffreychang — insujang@canis08: ~ — ssh insujang@cluster.kaist.ac.kr — 79×17

```
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 a.out...done.
(gdb) run
Starting program: /home/insujang/a.out
[Inferior 1 (process 3483) exited normally]
(gdb)
```

3. Add Breakpoints in a Program

- **Breakpoints: tell gdb "stop executing when you meet them"**
- **Without any breakpoint, gdb run the program until a program exits normally or an error occurs.**

```
(gdb) run
Starting program: /home/insujang/a.out
dl-debug.c:74: No such file or directory.
dl-debug.c:74: No such file or directory
[Inferior 1 (process 27222) exited normally]
(gdb)
```

- **To analyze variables in the runtime, we should add a breakpoint**

3. Add Breakpoints in a Program

- **Adding a breakpoint**

Command	Meaning
`(gdb) b 5` `(gdb) break 5`	Break at line 5 of the current file
`(gdb) b main` `(gdb) break main`	Break at the beginning of the `main()` function
`(gdb) b hello.c:5` `(gdb) break hello.c:5`	Break at line 5 of `hello.c`

- **List current breakpoints**
  - **(gdb) i(nfo) b(reakpoints)**

```
(gdb) b 2
Breakpoint 1 at 0x400535: file test.c, line 2.
(gdb) i breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400535 in main at test.c:2
```

3. Add Breakpoints in a Program

- **After adding a breakpoint, run will stop at the breakpoint**

```
(gdb) b 4
Breakpoint 1 at 0x4004f1: file test.c, line 4.
(gdb) run
Starting program: /home/insujang/a.out

Breakpoint 1, main () at test.c:4
4           int i=0;
(gdb)
```

Paused at line number 4

- **You can see assembly codes near the breakpoint**
  - `(gdb) disas`

```
Breakpoint 1, main () at test.c:4
4           int i=0;
(gdb) disas
Dump of assembler code for function main:
   0x00000000004004ed <+0>:     push   %rbp
   0x00000000004004ee <+1>:     mov    %rsp,%rbp
=> 0x00000000004004f1 <+4>:     movl   $0x0,-0x8(%rbp)
   0x00000000004004f8 <+11>:    movl   $0x0,-0x4(%rbp)
   0x00000000004004ff <+18>:    movl   $0x0,-0x8(%rbp)
```

3. Add Breakpoints in a Program

- **Enable/disable a breakpoint**
  - **(gdb) disable [number]**
  - **Disable a breakpoint <u>by the number of the breakpoint</u>**

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400535 in main at test.c:2
(gdb) disable 1
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x0000000000400535 in main at test.c:2
```

- **Delete a breakpoint**
  - **(gdb) delete [number] or clear [line number]**

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004004f1 in main at test.c:2
(gdb) delete 1
(gdb) i b
No breakpoints or watchpoints.
(gdb)
```

  - **e.g.) `(gdb) clear 2` = `(gdb) delete 1`**

4. Continue after Breakpoints

- `continue` will execute the program until the end of the program or an error occurs
- Do we need to add breakpoints <u>to every lines</u> to debug?
- `(gdb) n or next [number]`
- `(gdb) s or step [number]`
- Perform [number] of lines and stop again

```
Breakpoint 1, main () at test.c:4
4           int i=0;
(gdb) n 1
5           int sum=0;
(gdb) n
7           for(i=0; i<10; i++){
(gdb)
```

Line 6 is an empty line, hence skipped

4. Continue after Breakpoints

- `next` and `step` perform by source line level
- One source code line can contain multiple assemly instructions, e.g.) function call

```
Breakpoint 1, main () at test.c:10
10          add(4,7);
(gdb) disas
Dump of assembler code for function main:
   0x0000000000400501 <+0>:     push   %rbp
   0x0000000000400502 <+1>:     mov    %rsp,%rbp
   0x0000000000400505 <+4>:     sub    $0x10,%rsp
   0x0000000000400509 <+8>:     movl   $0x4,-0x8(%rbp)
   0x0000000000400510 <+15>:    movl   $0x7,-0x4(%rbp)
=> 0x0000000000400517 <+22>:    mov    $0x7,%esi
   0x000000000040051c <+27>:    mov    $0x4,%edi
   0x0000000000400521 <+32>:    callq  0x4004ed <add>
   0x0000000000400526 <+37>:    mov    $0x0,%eax
   0x000000000040052b <+42>:    leaveq
   0x000000000040052c <+43>:    retq
End of assembler dump.
(gdb)
```

Instructions for calling add function
Place 4 and 7 to registers and call it

4. Continue after Breakpoints

- **Need more fine-grained control: control by assembly instruction level**
- **`(gdb) ni or nexti [number]`**
- **`(gdb) si or stepi [number]`**

```
Breakpoint 1, main () at test.c:10
10          add(4,7);
(gdb) ni 1
0x000000000040051c      10          add(4,7);
(gdb) ni 1
0x0000000000400521      10          add(4,7);
(gdb) disas
Dump of assembler code for function main:
   0x0000000000400501 <+0>:     push   %rbp
   0x0000000000400502 <+1>:     mov    %rsp,%rbp
   0x0000000000400505 <+4>:     sub    $0x10,%rsp
   0x0000000000400509 <+8>:     movl   $0x4,-0x8(%rbp)
   0x0000000000400510 <+15>:    movl   $0x7,-0x4(%rbp)
   0x0000000000400517 <+22>:    mov    $0x7,%esi
   0x000000000040051c <+27>:    mov    $0x4,%edi
=> 0x0000000000400521 <+32>:    callq  0x4004ed <add>
```

Using `ni` instruction: Debugging is done by assembly instruction level

Assembly instructions for calling add function

4. Continue after Breakpoints

- You can perform `next` <u>until specific location</u>
- `(gdb) u(ntil) [line number]`
- e.g) `until 7` will perform execution until **line number 7** has been reached

```
Breakpoint 1, main () at test.c:4
4           int i=0;
(gdb) until 7
main () at test.c:7
7           for(i=0; i<10; i++){
(gdb)
```

- Also use `until` by machine instruction level
- `(gdb) u(ntil) *[memory address]`
- e.g.) `until *0x400521` will perform execution until **instruction at memory 0x400521**

4. Continue after Breakpoints

jeffreychang — insujang@canis08: ~ — ssh insujang@cluster.kaist.ac.kr — 79×32

```
(gdb) disas
Dump of assembler code for function main:
   0x0000000000400501 <+0>:     push   %rbp
   0x0000000000400502 <+1>:     mov    %rsp,%rbp
   0x0000000000400505 <+4>:     sub    $0x10,%rsp
=> 0x0000000000400509 <+8>:     movl   $0x4,-0x8(%rbp)
   0x0000000000400510 <+15>:    movl   $0x7,-0x4(%rbp)
   0x0000000000400517 <+22>:    mov    $0x7,%esi
   0x000000000040051c <+27>:    mov    $0x4,%edi
   0x0000000000400521 <+32>:    callq  0x4004ed <add>
   0x0000000000400526 <+37>:    mov    $0x0,%eax
   0x000000000040052b <+42>:    leaveq
   0x000000000040052c <+43>:    retq
End of assembler dump.
(gdb) until *0x400526
main () at test.c:12
12          return 0;
(gdb) disas
Dump of assembler code for function main:
   0x0000000000400501 <+0>:     push   %rbp
   0x0000000000400502 <+1>:     mov    %rsp,%rbp
   0x0000000000400505 <+4>:     sub    $0x10,%rsp
   0x0000000000400509 <+8>:     movl   $0x4,-0x8(%rbp)
   0x0000000000400510 <+15>:    movl   $0x7,-0x4(%rbp)
   0x0000000000400517 <+22>:    mov    $0x7,%esi
   0x000000000040051c <+27>:    mov    $0x4,%edi
   0x0000000000400521 <+32>:    callq  0x4004ed <add>
=> 0x0000000000400526 <+37>:    mov    $0x0,%eax
   0x000000000040052b <+42>:    leaveq
   0x000000000040052c <+43>:    retq
End of assembler dump.
(gdb)
```

5. Examine Variables

- See what value is stored in a variable
- `(gdb) print [variable name]`
- e.g.) `print i`

```
Breakpoint 1, main () at test.c:4
4           int i=0;
(gdb) u 7
main () at test.c:7
7           for(i=0; i<10; i++){
(gdb) n 2
7           for(i=0; i<10; i++){
(gdb) print i
$2 = 0
(gdb) n 2
7           for(i=0; i<10; i++){
(gdb) print i
$3 = 1
(gdb)
```

5. Examine Variables

- See what value is stored that this address indicates
  - `(gdb) x [address]`
  - `e.g.) x &i`
  - Especially useful to see strings (`char array` in C)

```
(gdb) list
3       int main(){
4           int i=0;
5           int sum=0;
6           char str[10] = "Hello!\n";
7
8           for(i=0; i<10; i++){
9               sum+=i;
10          }
11
12          return 0;
(gdb) x/s str
0x7fffffffe480: "Hello!\n"
(gdb)
```

Use /s to examine char[]

5. Examine Variables

- **Options for x instruction**

instruction	Meaning	Example
x/x	Print the value as hexadecimal	`int value1 = 0xff;` `x/x &value1 -> 0x000000ff`
x/t	Print the value as binary	`x/t &value1 -> 0000…0011111111`
x/b	Print by byte	`x/x &value1 -> 0x000000ff` `x/xb &value1 -> 0xff`
x/w	Print by word	`x/x &value1 = x/xw &value1`
x/s	Print string until ₩0 character	`char str[10] = “Hello!\n”` `x/s str -> “Hello!\n”`
x/[number]	Print number of variables	`x/xb &value1 -> 0xff` `x/3xb &value1 -> 0xff 0x00 0x00`

- **You can combine options such as**

x/4wx 0xbffff2a0

Examine 4 words as hexadecimal from address 0xbffff2a0

5. Examine Variables

- **See which type the variable is**
- **`(gdb) whatis [variable]`**
- **e.g.) `whatis b`**

```
Breakpoint 1, main () at test.c:12
(gdb) print b
$5 = 4
(gdb) whatis b
type = int
(gdb)
```

5. Examine Variables

- **You can also see value of <u>registers</u>**
- **`(gdb) print $<register>`**
  - **e.g.) `print $rax`**

```
Breakpoint 1, main () at test.c:11
(gdb) print $rax
$1 = 140737488348308
(gdb)
```

Example

```
int add (int a, int b){
    return a+b;
}

void main (){
    int i=0;
    int sum=0;
    for(; i<10; i++){
        sum = add(i, sum);
    }
}
```

- How this code is in machine instruction?

Precautions: Running a Bomb

- **Bombs are restricted to be run in your local machine**
  - **Defusing and exploding bombs requires the running machine connected online**
  - **Local machine can be offline after downloading the bomb**
  - **Defusing a bomb is only allowed in the canis machines**

  - `$ gdb bomb`
  - `(gdb) run`

Precautions: Skipping Defused Phases

- You are asked to defuse every phases whenever run a bomb
- Make a file to skip already defused phases (you should remember the answer and save it into a file)
  - Use vim to make a file: e.g.) `vim input_file`
  - Any name is fine
  - One line for each phase

```
jeffreychang — insujang@canis08: ~/bomb5 — ssh insujang@cluster.kaist.ac.k...
1 hello, this string will be fed for phase 1.
2 This will be for phase 2.
```

- Run GDB with the name of file
  - `$ gdb bomb`
  - `(gdb) run input_file`
  - This will automatically put each line to each phase

Precautions: Grading

- This is important: grading policy is different from that in datalab
- Defusing all phases will give you 70 points
- -0.5 point penalty per explosion (the final score will be rounded up)

- If you use only one bomb
  - Your score = (score earned by defusing the bomb) – (score panelized by exploding the bomb)
  - e.g.) You solve 5 phases with 13 explosions: 55 – 6.5 = 48.5 (Rounded up, hence your final score = 49)

Precautions: Grading

- **If you use multiple bombs**
  - Again, we do not recommend you to use multiple bombs
  - The maximum earning score by defusing phases among bombs is applied
  - But, all explosions for all bombs will be penalized

  - Example 1
    - You use two bombs and defused / exploded them respectively
    - Bomb 1: defused 3 phases, exploded 9 times
    - Bomb 2: defused 4 phases, exploded 7 times
    - Final score: $max(30, 40) - 0.5 \times (9 + 7) = 32$
  - Example 2
    - You use two bombs and defused / exploded them respectively
    - Bomb 1: defused 6 phases, exploded 42 times
    - Bomb 2: not used
    - Final score: $max(70, 0) - 0.5 \times (42 + 0) = 59$