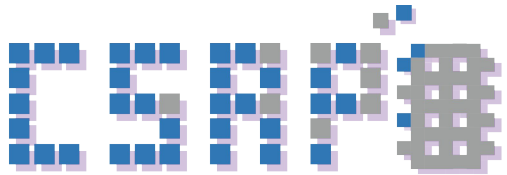
Computer Architecture Lab Session

4. Bomb Lab

2020/05/06

comparch@csap.snu.ac.kr



Computer Systems and Platforms Laboratory
School of Computer Science and Engineering
Seoul National University

Overview

- Due: Wednesday, May 20, 11:00
- In this lab, you will find out what this bomb does by reading assembly code and defuse it by entering secret codes.
- Read the README carefully. It contains the full instructions of the lab. This is an overview with some tips.
- Unlike previous lab, you should get the bomb from url, not from the git.
- However you should fork the repository, make it private and update your report.

Downloading the Bomb

- Visit
  - https://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

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

Problem specification

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:

```
jiyeon@gentoo ~/02.Bomblab/bombs $ tar xvf bomb1.tar
bomb1/
bomb1/bomb
bomb1/bomb.c
bomb1/README
jiyeon@gentoo ~/02.Bomblab/bombs $ cd bomb1
jiyeon@gentoo ~/02.Bomblab/bombs/bomb1 $ cat README
This is bomb 1.

It belongs to 2020-12345 (test)
jiyeon@gentoo ~/02.Bomblab/bombs/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**
- 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**.

```
69      printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
70      printf("which to blow yourself up. Have a nice day!\n");
71
72      /* Hmm...  Six phases must be more secure than one phase! */
73      input = read_line();             /* Get input                   */
74      phase_1(input);                  /* Run the phase               */
75      phase_defused();                 /* Drat!  They figured it out!
76                                        * Let me know how they did it. */
77      printf("Phase 1 defused. How about the next one?\n");
78
```

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":

```
jiyeon@gentoo ~/example $ ./bomb
Welcome to my fiendish little bomb. You have 1 phases with
which to blow yourself up. Have a nice day!
test

BOOM!!!
The bomb has blown up.
jiyeon@gentoo ~/example $
```

- 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.
  - 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

```
jiyeon@gentoo ~/example $ objdump -t bomb > bomb.symbols
jiyeon@gentoo ~/example $ objdump -d bomb > bomb.disas
jiyeon@gentoo ~/example $
```

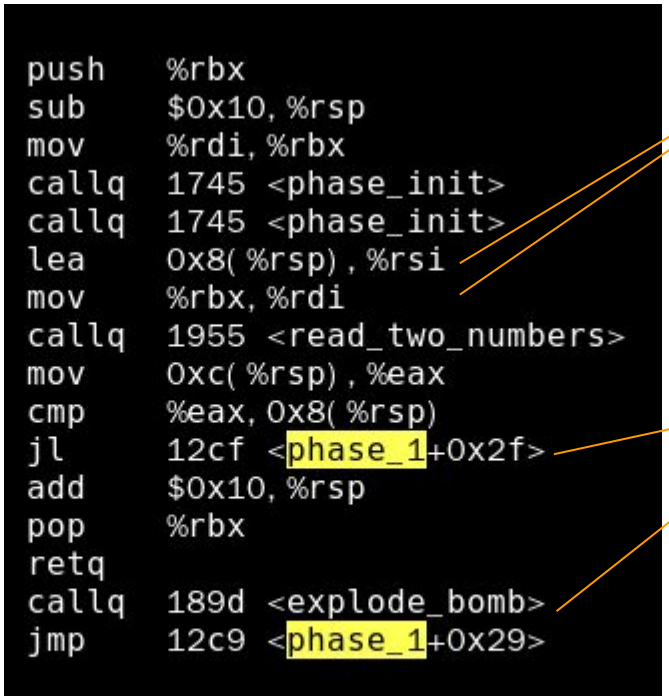
Inspecting the code of phase_1

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

```
199 00000000000012a0 <phase_1>:
200     12a0:  53                      push   %rbx
201     12a1:  48 83 ec 10             sub    $0x10,%rsp
202     12a5:  48 89 fb                mov    %rdi,%rbx
203     12a8:  e8 98 04 00 00          callq  1745 <phase_init>
204     12ad:  e8 93 04 00 00          callq  1745 <phase_init>
205     12b2:  48 8d 74 24 08          lea    0x8(%rsp),%rsi
206     12b7:  48 89 df                mov    %rbx,%rdi
207     12ba:  e8 96 06 00 00          callq  1955 <read_two_numbers>
208     12bf:  8b 44 24 0c             mov    0xc(%rsp),%eax
209     12c3:  39 44 24 08             cmp    %eax,0x8(%rsp)
210     12c7:  7c 06                   jl     12cf <phase_1+0x2f>
211     12c9:  48 83 c4 10             add    $0x10,%rsp
212     12cd:  5b                      pop    %rbx
213     12ce:  c3                      retq
214     12cf:  e8 c9 05 00 00          callq  189d <explode_bomb>
215     12d4:  eb f3                   jmp    12c9 <phase_1+0x29>
216
217 00000000000012d6 <phase_2>:
 173: 46 [ 18%]                                          ~/example/bomb.disas
/phase_1
```

Inspecting the code of phase_1

- From the code we can see that:



- **phase_1** calls a function called **read_two_numbers** with two arguments

- then, depending on the result of **read_two_numbers**, either calls **explode_bomb** or returns.

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 string
- now gdb stops at the entry of **phase_1** (disassemble with disas)

```
jiyeon@gentoo ~/example $ gdb bomb
GNU gdb (Gentoo 8.3 vanilla) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-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...
(gdb) break phase_1
Breakpoint 1 at 0x12a0
(gdb) run
Starting program: /home/jiyeon/example/bomb
Welcome to my fiendish little bomb. You have 1 phases with
which to blow yourself up. Have a nice day!
hello

Breakpoint 1, 0x000055555555552a0 in phase_1 ()
(gdb)
```

Stepping through the Code

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

```
(gdb) run
Starting program: /home/jiyeon/example/bomb
Welcome to my fiendish little bomb. You have 1 phases with
which to blow yourself up. Have a nice day!
hello

Breakpoint 1, 0x000055555555552a0 in phase_1 ()
(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 24878) 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 function **read_two_numbers** seems reasonable.
  - breakpoints to addresses are setby entering **break *<address>**
  - continue execution to the nextbreakpoint 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

```
Breakpoint 1, 0x00005555555552a0 in phase_1 ()
(gdb) stepi
0x00005555555552a1 in phase_1 ()
(gdb) si
0x00005555555552a5 in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
   0x00005555555552a0 <+0>:     push   %rbx
   0x00005555555552a1 <+1>:     sub    $0x10,%rsp
=> 0x00005555555552a5 <+5>:     mov    %rdi,%rbx
   0x00005555555552a8 <+8>:     callq  0x555555555745 <phase_init>
   0x00005555555552ad <+13>:    callq  0x555555555745 <phase_init>
   0x00005555555552b2 <+18>:    lea    0x8(%rsp),%rsi
   0x00005555555552b7 <+23>:    mov    %rbx,%rdi
   0x00005555555552ba <+26>:    callq  0x555555555955 <read_two_numbers>
   0x00005555555552bf <+31>:    mov    0xc(%rsp),%eax
   0x00005555555552c3 <+35>:    cmp    %eax,0x8(%rsp)
   0x00005555555552c7 <+39>:    jl     0x5555555552cf <phase_1+47>
   0x00005555555552c9 <+41>:    add    $0x10,%rsp
   0x00005555555552cd <+45>:    pop    %rbx
   0x00005555555552ce <+46>:    retq
   0x00005555555552cf <+47>:    callq  0x55555555589d <explode_bomb>
   0x00005555555552d4 <+52>:    jmp    0x5555555552c9 <phase_1+41>
End of assembler dump.
(gdb) break *0x555555555955
Breakpoint 2 at 0x555555555955
(gdb) break read_two_numbers
Note: breakpoint 2 also set at pc 0x555555555955.
Breakpoint 3 at 0x555555555955
(gdb)
```

Inspecting Registers and Memory

- After executing s at the call to **read_two_numbers**, enter disas again to see where we currently are.
- as expected, the debugger stopped at the first instruction of **read_two_numbers**
- from the name we guess that the function probably read two numbers. The code confirms this assumption: it first calls the **sscanf** with using address from rsi as first argument(%rdx). If the return value(number of scanned values) is less or equal than 1, it call explode_bomb.

```
0x0000555555555955 <+0>:     sub    $0x8,%rsp
0x0000555555555959 <+4>:     mov    %rsi,%rdx
0x000055555555595c <+7>:     lea    0x4(%rsi),%rcx
0x0000555555555960 <+11>:    lea    0x8f1(%rip),%rsi        # 0x555555556258
0x0000555555555967 <+18>:    mov    $0x0,%eax
0x000055555555596c <+23>:    callq  0x555555555090 <__isoc99_sscanf@plt>
0x0000555555555971 <+28>:    cmp    $0x1,%eax
0x0000555555555974 <+31>:    jle    0x55555555597b <read_two_numbers+38>
0x0000555555555976 <+33>:    add    $0x8,%rsp
0x000055555555597a <+37>:    retq
0x000055555555597b <+38>:    nopl   0x0(%rax,%rax,1)
0x0000555555555980 <+43>:    callq  0x55555555589d <explode_bomb>
```

Inspecting Registers and Memory

- Use **p/x $<reg>** to print the contents of a register in hexadecimal form
  - Also you can specify with **info registers [i r]** command
  - enter help print (or help p) to see what options the print command offers
- Scanned values are stored in memory. You can see it by using gdb commands: x [addr]
  - with options, use x/d, x/s ...
  - x/s $reg

```
Welcome to my fiendish little bomb. You have 1 phases with
which to blow yourself up. Have a nice day!
1 2

Breakpoint 1, 0x00005555555552a0 in phase_1 ()
(gdb) s
Single stepping until exit from function phase_1,
which has no line number information.

Breakpoint 2, 0x0000555555555955 in read_two_numbers ()
(gdb) disas
Dump of assembler code for function read_two_numbers:
=> 0x0000555555555955 <+0>:     sub    $0x8,%rsp
   0x0000555555555959 <+4>:     mov    %rsi,%rdx
   0x000055555555595c <+7>:     lea    0x4(%rsi),%rcx
   0x0000555555555960 <+11>:    lea    0x8f1(%rip),%rsi        # 0x555555556258
   0x0000555555555967 <+18>:    mov    $0x0,%eax
   0x000055555555596c <+23>:    callq  0x555555555090 <__isoc99_sscanf@plt>
   0x0000555555555971 <+28>:    cmp    $0x1,%eax
   0x0000555555555974 <+31>:    jle    0x55555555597b <read_two_numbers+38>
   0x0000555555555976 <+33>:    add    $0x8,%rsp
   0x000055555555597a <+37>:    retq
   0x000055555555597b <+38>:    nopl   0x0(%rax,%rax,1)
   0x0000555555555980 <+43>:    callq  0x55555555589d <explode_bomb>
End of assembler dump.
(gdb) si
0x0000555555555959 in read_two_numbers ()
(gdb) si
0x000055555555595c in read_two_numbers ()
(gdb) si
0x0000555555555960 in read_two_numbers ()
(gdb) si
0x0000555555555967 in read_two_numbers ()
(gdb) i r rdx
rdx            0x7fffffffdae8      140737488345832
(gdb) x/s 0x7fffffffdae8
0x7fffffffdae8: "\223YUUUU"
(gdb) i r rcx
rcx            0x7fffffffdaec      140737488345836
(gdb) x/d 0x7fffffffdaec
0x7fffffffdaec: 85
(gdb) i r rsi
rsi            0x555555556258      93824992240216
(gdb) x/s 0x555555556258
0x555555556258: "%d %d"
```

Inspecting Registers and Memory

```
(gdb) break read_two_numbers+28
Function "read_two_numbers+28" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) break *0x555555555971
Breakpoint 3 at 0x555555555971
(gdb) s
Single stepping until exit from function read_two_numbers,
which has no line number information.

Breakpoint 3, 0x0000555555555971 in read_two_numbers ()
(gdb) disas
Dump of assembler code for function read_two_numbers:
   0x0000555555555955 <+0>:     sub    $0x8,%rsp
   0x0000555555555959 <+4>:     mov    %rsi,%rdx
   0x000055555555595c <+7>:     lea    0x4(%rsi),%rcx
   0x0000555555555960 <+11>:    lea    0x8f1(%rip),%rsi        # 0x555555556258
   0x0000555555555967 <+18>:    mov    $0x0,%eax
   0x000055555555596c <+23>:    callq  0x555555555090 <__isoc99_sscanf@plt>
=> 0x0000555555555971 <+28>:    cmp    $0x1,%eax
   0x0000555555555974 <+31>:    jle    0x55555555597b <read_two_numbers+38>
   0x0000555555555976 <+33>:    add    $0x8,%rsp
   0x000055555555597a <+37>:    retq
   0x000055555555597b <+38>:    nopl   0x0(%rax,%rax,1)
   0x0000555555555980 <+43>:    callq  0x55555555589d <explode_bomb>
End of assembler dump.
(gdb) x/d 0x7fffffffdae8
0x7fffffffdae8: 1
(gdb) x/d 0x7fffffffdaec
0x7fffffffdaec: 2
```

Inspecting Registers and Memory

- Now we can know what the codes are doing.
- We should type two numbers a b, and a should be equal or greater than b.

```
(gdb) disas
Dump of assembler code for function phase_1:
   0x000055555555552a0 <+0>:     push   %rbx
   0x000055555555552a1 <+1>:     sub    $0x10,%rsp
   0x000055555555552a5 <+5>:     mov    %rdi,%rbx
   0x000055555555552a8 <+8>:     callq  0x555555555745 <phase_init>
   0x000055555555552ad <+13>:    callq  0x555555555745 <phase_init>
   0x000055555555552b2 <+18>:    lea    0x8(%rsp),%rsi
   0x000055555555552b7 <+23>:    mov    %rbx,%rdi
   0x000055555555552ba <+26>:    callq  0x555555555955 <read_two_numbers>
   0x000055555555552bf <+31>:    mov    0xc(%rsp),%eax
=> 0x000055555555552c3 <+35>:    cmp    %eax,0x8(%rsp)
   0x000055555555552c7 <+39>:    jl     0x5555555552cf <phase_1+47>
   0x000055555555552c9 <+41>:    add    $0x10,%rsp
   0x000055555555552cd <+45>:    pop    %rbx
   0x000055555555552ce <+46>:    retq
   0x000055555555552cf <+47>:    callq  0x55555555589d <explode_bomb>
   0x000055555555552d4 <+52>:    jmp    0x5555555552c9 <phase_1+41>
End of assembler dump.
(gdb) i r eax
eax            0x2                 2
(gdb) x/d $rsp + 8
0x7fffffffdae8: 1
```

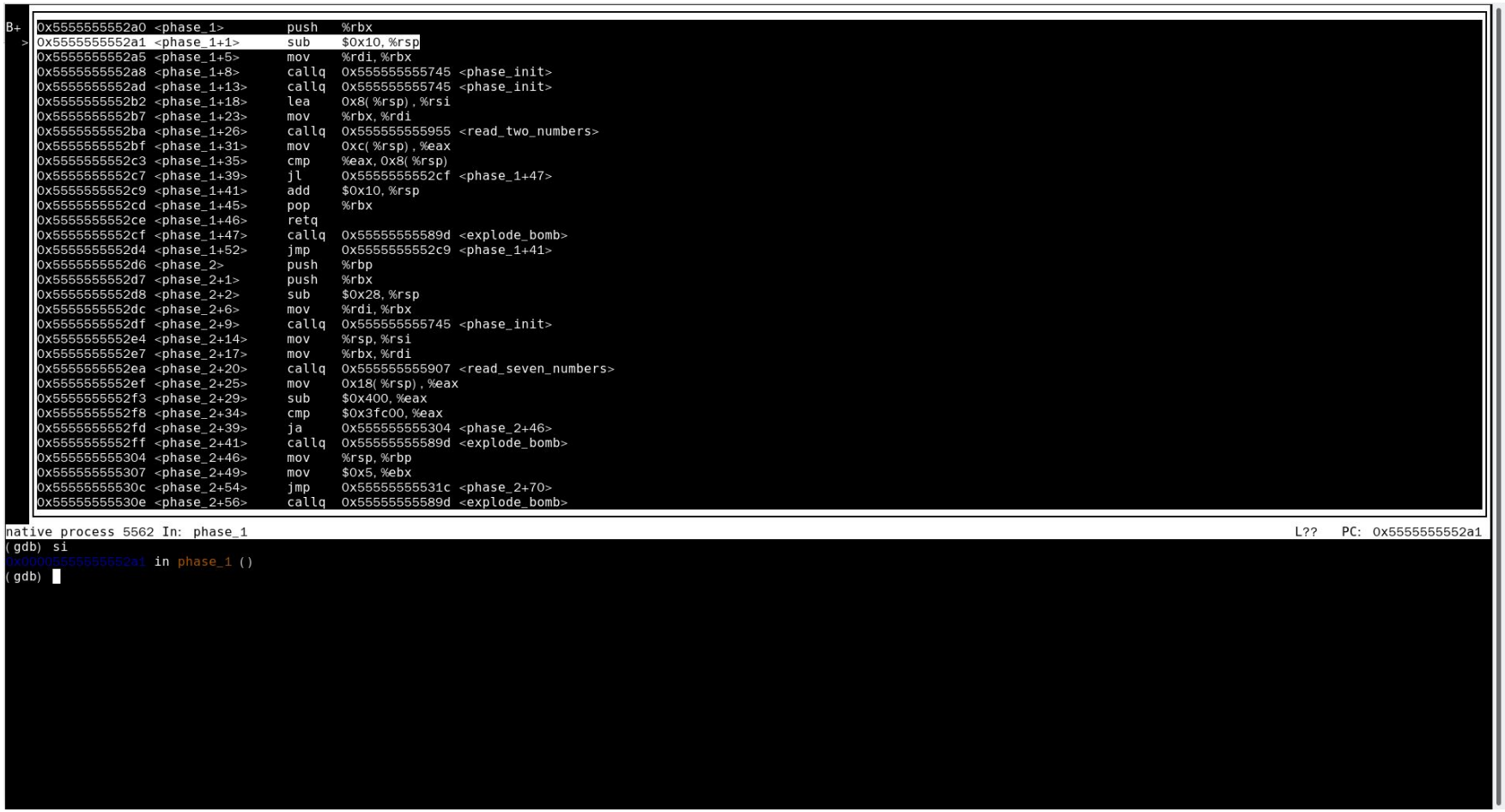
Inspecting Registers and Memory

- 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.
  - You can now defuse the phase 1!
- Quit program by typing **quit**

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/jiyeon/example/bomb
Welcome to my fiendish little bomb. You have 1 phases with
which to blow yourself up. Have a nice day!
3 1

Breakpoint 1, 0x0000555555555a0 in phase_1 ()
(gdb) s
Single stepping until exit from function phase_1,
which has no line number information.
main (argc=<optimized out>, argv=<optimized out>) at bomb.c:75
75          phase_defused();                    /* Drat!  They figured it out!
(gdb) quit
A debugging session is active.

        Inferior 1 [process 31447] will be killed.

Quit anyway? (y or n) y
jiyeon@gentoo ~/example $ ./bomb
Welcome to my fiendish little bomb. You have 1 phases with
which to blow yourself up. Have a nice day!
3 1
Phase 1 defused. How about the next one?
jiyeon@gentoo ~/example $
```

GDB Layout

- By typing layout asm, you can see program running with assembly codes.



Report

- Check report template on git.

Evaluation

- Total 60 points
  - Defusing bomb - 40 points, Report - 20 points
- For each explodes you will lose 0.5 points.
  - Make sure that you set breakpoints before running the bomb!
- You can check your real time score at https://csap.snu.ac.kr/comparch/bomblab/scoreboard

Good Luck!

For questions contact

comparch@csap.snu.ac.kr