Lab #2. Reversing Lab

Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

About this Lab

- **In this lab, you have to analyze the assembly code of a program to figure out what it is doing**
  - This is called ***reverse engineering***, or ***reversing*** in short: that's why this assignment is named ***Reversing Lab***
  - Inspired by *Bomb Lab* of the original *CS:APP* course in CMU
- **Good chance to deepen your understanding on x86-64 assembly language**
  - Also, you will learn the basic usage of **gdb** (debugger)
- **You will see similar problems in the midterm exam**
  - So **start working on this lab early**, if you want to practice for the midterm exam

Remind: Cheating Policy

- **Cheating in assignment will give you a serious penalty**
  - Your final grade will be downgraded (e.g., from ***B+*** to **C+**)
- **Scope of cheating in assignment**
  - Copying the code of other people
  - Sharing your solution with others
  - Asking ChatGPT to write your code
  - Discussing with others how to solve the problem
- **Based on my experience from previous years, I know that cheating is quite prevalent in this course**
  - Don’t buy or sell solutions from online communities
  - If suspicious, I can call you to come to my office and ask to solve the problems in front of me

General Information

- **Check the *Assignment* tab of *Cyber Campus***
  - Skeleton code (`Lab2.tgz`) is attached together with this slide
  - Submission will be accepted in the same post, too
- **Deadline: 5/2 Friday 23:59**
  - Late submission deadline: **5/4 Sunday 23:59** **(-20% penalty)**
  - Delay penalty is applied uniformly **(not problem by problem)**
- **Please read the instructions in this slide carefully**
  - This slide is a step-by-step tutorial for the lab
  - It also contains important submission guidelines
    - If you do not follow the guidelines, **you will get penalty**

Skeleton Code Structure

- **Copy `Lab2.tgz` into CSPRO server and decompress it**
  - Recommend to use [cspro**2**.sogang.ac.kr](cspro2.sogang.ac.kr)
  - Don't decompress-and-copy; copy-and-decompress
- **`2-1~2-4`: Each directory contains a problem**
- **`check.py`: Script for self-grading (explained later)**
- **`config`: Used by grading script (you don't have to care)**
- **`helper.py`: Helper library (you don't have to care)**

```
jason@ubuntu:~$ tar -xzf Lab2.tgz
jason@ubuntu:~$ ls Lab2
2-1  2-2  2-3  2-4  check.py  config  helper.py
```

Problem Directory (Example: 2-1)

- **`problem1.c`: Partial source code provided as hint**
  - Complete source code is **NOT** given, as explained before
- **`problem1.bin`: The binary executable compiled from the source file (`problem1.c`)**
  - You have to analyze the assembly code of this file
- **`solve1.py`: Solution script that you have to fill in later**

```
jason@ubuntu:~/Lab2/2-1$ ls -l
total 28
-rwxrwxr-x 1 jason jason 16528 Feb 17 07:16 problem1.bin
-rw-rw-r-- 1 jason jason   174 Feb 17 07:16 problem1.c
-rwxr-xr-x 1 jason jason   511 Feb 17 07:20 solve1.py
```

Tasks to do

- **The program will ask you to enter some input**
  - If the input satisfies certain condition, the program prints out a message for congratulation: **`"You passed the challenge!"`**
  - Otherwise, the program will reject your input
- **Your job is to analyze the assembly code and figure out what kind of input you have to give to the program**

```
jason@ubuntu:~/Lab2/2-1$ ./problem1.bin
Provide your input:
abcde12345
No, that is not the input I want!
jason@ubuntu:~/Lab2/2-1$ ./problem1.bin
Provide your input:

You passed the challenge!
```

TODO: Find out what should come here

Partial Source File: `problem1.c`

- **Only some part of the source code is provided**
  - When you open **`2-1/problem1.c`**, you will see the code below
  - In the remaining (hidden) part of the code, the program will check your input and decide what to print out

```c
#include <stdio.h>

int main(void) {
  char buf[32];
  // ... More variables may exist

  puts("Provide your input:");
  scanf("%31s", buf);
  // ... More code will follow
}
```

GDB Usage: Disassemble Code

- **Command: `disassemble <func>` (or `disas <func>`)**
  - Print the assembly code of `<func>`

```
jason@ubuntu:~/Lab2/2-1$ gdb ./problem1.bin -q
Reading symbols from ./problem1.bin...
(No debugging symbols found in ./problem1.bin)
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401136 <+0>:     sub    $0x28,%rsp
   0x000000000040113a <+4>:     mov    $0x402004,%edi
   0x000000000040113f <+9>:     callq  0x401030 <puts@plt>
   0x0000000000401144 <+14>:    mov    %rsp,%rsi
   0x0000000000401147 <+17>:    mov    $0x402018,%edi
   0x000000000040114c <+22>:    mov    $0x0,%eax
   0x0000000000401151 <+27>:    callq  0x401040 <__isoc99_scanf@plt>
   0x0000000000401156 <+32>:    mov    $0x0,%eax
   0x000000000040115b <+37>:    movslq %eax,%rdx
   0x000000000040115e <+40>:    movzbl 0x404038(%rdx),%edx
```

(You type this)

GDB Usage: Examine Memory

- **Let' examine the argument of the first `puts()`**
  - From the source code, we already know that the first argument is string "**`Provide your input:`**"
  - In assembly, **`0x402004`** is passed as the argument of **`puts()`**
  - Let's confirm if this address really contains the expected string

```
char buf[32];
puts("Provide your input:");
```

Correspond

```
0x401136:sub    $0x28,%rsp
0x40113a:mov    $0x402004,%edi
0x40113f:callq  0x401030 <puts@plt>
```

GDB Usage: Examine Memory

- **Command:** `x/<N><t> <addr>`
  - Print `<N>` chunks of data in `<t>` type, starting from `<addr>`
  - `<N>` can be omitted when it is 1
  - `<t>` can specify various formats
  - Ex) `x/16xb <addr>` : print 16 **b**ytes in hex
  - Ex) `x/10xw <addr>` : print 10 **w**ords (4-byte chunks) in hex
  - Ex) `x/2xg <addr>` : print 2 **g**iant words (8-byte chunks) in hex
  - Ex) `x/s <addr>` : print one **s**tring (until the null character)

```
(gdb) x/s 0x402004
0x402004:       "Provide your input:"
(gdb) x/24xb 0x402004
0x402004:       0x50    0x72    0x6f    0x76    0x69    0x64    0x65    0x20
0x40200c:       0x79    0x6f    0x75    0x72    0x20    0x69    0x6e    0x70
0x402014:       0x75    0x74    0x3a    0x00    0x25    0x33    0x31    0x73
```

* In GDB, word means 4-byte for some historical reason

GDB Usage: Runtime Debugging

- **Sometimes, you may want to observe the program execution to confirm whether your analysis is correct**
- **Command: `b * <addr>`**
  - Set a **b**reakpoint at **`<addr>`**
- **Command: `r`**
  - **R**un the program (will stop when breakpoint is met)
- **Command: `c`**
  - **C**ontinue the execution by resuming from the breakpoint
- **Command: `si` or `ni`**
  - **S**tep through the current **i**nstruction
  - `si` / `ni` are slightly different when the current instruction is **`call`**

GDB Usage: Example (1/3)

- Let's put a breakpoint at `0x40115e` and observe how the value of `%rdx` changes before and after this point
- The `movzbl` instruction will load a byte from address `0x404038+%rdx` and put it in `%edx` (part of `%rdx`)

```
0x0000000000401136 <+0>:     sub    $0x28,%rsp
0x000000000040113a <+4>:     mov    $0x402004,%edi
0x000000000040113f <+9>:     callq  0x401030 <puts@plt>
0x0000000000401144 <+14>:    mov    %rsp,%rsi
0x0000000000401147 <+17>:    mov    $0x402018,%edi
0x000000000040114c <+22>:    mov    $0x0,%eax
0x0000000000401151 <+27>:    callq  0x401040 <__isoc99_scanf@plt>
0x0000000000401156 <+32>:    mov    $0x0,%eax
0x000000000040115b <+37>:    movslq %eax,%rdx
0x000000000040115e <+40>:    movzbl 0x404038(%rdx),%edx
```

(We will set a breakpoint here)

GDB Usage: Example (2/3)

- You can see the use of `b * <addr>` and `r` below
- When you type `r`, the program will start and wait for you to type the input
- Let's assume that you type "`ABCDE`" as input

```
(gdb) b * 0x40115e
Breakpoint 1 at 0x40115e
(gdb) r
Starting program: /home/jason/Lab2/2-1/problem1.bin
Provide your input:
ABCDE

Breakpoint 1, 0x000000000040115e in main ()
```

(Waiting for your input)

(This is your input)

GDB Usage: Example (3/3)

- **Now we *hit* the breakpoint and you have the control**
  - In other words, you can type the GDB commands
- **Command: info reg <register>**
  - Print the current value of <register>
- **You can see that %rdx has changed from 0x0 to 0x43, as you type si command to execute one instruction**

```
Breakpoint 1, 0x000000000040115e in main ()
(gdb) info reg rdx
rdx            0x0                 0
(gdb) si
0x0000000000401165 in main ()
(gdb) info reg rdx
rdx            0x43                67
```

More Hint on GDB Usage

- **One of the problem in this Lab uses `switch` statement**
  - Recall that **`switch`** is often compiled to use a jump table
- **To examine the jump table, you can use `x/_xg <addr>`**
  - Following memory dump shows an example of jump table entries (caution: this dump is **not** obtained from Lab #2 binaries)

```
(gdb) x/4xg 0x402008
0x402008:       0x0000000000401113      0x0000000000401119
0x402018:       0x000000000040111d      0x0000000000401127
```

Address	Jump Table
0x402008	0x401113
0x402010	0x401119
0x402018	0x40111d
0x402020	0x401127
0x402028	...

Solution Code (Script)

- **Once you figure out what kind of input you must give to the program, you must write it down in the code form**
  - Programmers talk with code
- **For `problem1.bin`, you must write down your solution code in `solve1.py` and submit this file**
- **I prepared some useful class and methods in `helper.py`**
  - You don't have to read or understand the **`helper.py`** file
  - You can just use the provided methods to write your code

How to write the solution code

- **You can interact with the target program by using the following class and methods**
  - First, create an object of **`Program`** class
  - **`read_line()`**: reads a single line of program output
  - **`send_line(s)`**: send **`s + "\n"`** as a program input

Example code for solve1.py

```
prog = Program("./problem1.bin")
print(prog.read_line()) # Read the initial message
prog.send_line("ABCDE") # Send your input to the program
print(prog.read_line()) # Read the response of the program
```

(When you run the code above)

```
jason@ubuntu:~/Lab2/2-1$ ./solve1.py
Provide your input:
No, that is not the input I want!
```

Self-Grading

- **Once you think everything is done, run `check.py` to confirm that you solved all the problems**
  - Each character in the result has following meaning:
    `'O'`: correct, `'X'`: wrong, `'T'`: timeout, `'E'`: runtime error
- **Four problems (from `2-1` to `2-4`) in total, 25 point each**
  - You'll get the point for each problem if your solution script works
  - **No partial point if the solution script does not work**

```
jason@ubuntu:~/Lab2$ ./check.py
[*] 2-1: O
[*] 2-2: X
[*] 2-3: X
[*] 2-4: X
```

Report

- **For each problem, explain how you analyzed the assembly code and figured out the program's behavior**
  - You may copy & paste relevant part of the assembly code
  - Clearly describe your reasoning process (not your guess)
  - Don't have to write report for the problem that you couldn't solve
- **The role of report is to prove that you solved the problems by yourself, with a clear understanding**
  - Report will not give you score; it is only used to deduct score if the explanation is incorrect or insufficient
  - Also, I will use your reports to ***spot the cheating*** (code copy)
  - So it doesn't have to be long: no more than 1 page per problem
- **You can use either Korean or English in the report**

Submission Guideline

- **You should submit the following 5 files**
  - Problem 2-1: **`solve1.py`**
  - Problem 2-2: **`solve2.py`**
  - Problem 2-3: **`solve3.py`**
  - Problem 2-4: **`solve4.py`**
  - Report **(don’t forget this)**: **`report.pdf`**
- **Submission format**
  - Upload these files directly to *Cyber Campus* (**do not zip them**)
  - **Do not change the file name** (e.g., adding any prefix or suffix)
  - If your submission format is wrong, you will get **-20% penalty**