[CSED211] Introduction to Computer Software Systems

Lab 4: Attack Lab

Hyeongmin Oh

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

CAOS
COMPUTER ARCHITECTURE &
OPERATING SYSTEMS LABORATORY

2024.10.17

Today's Agenda

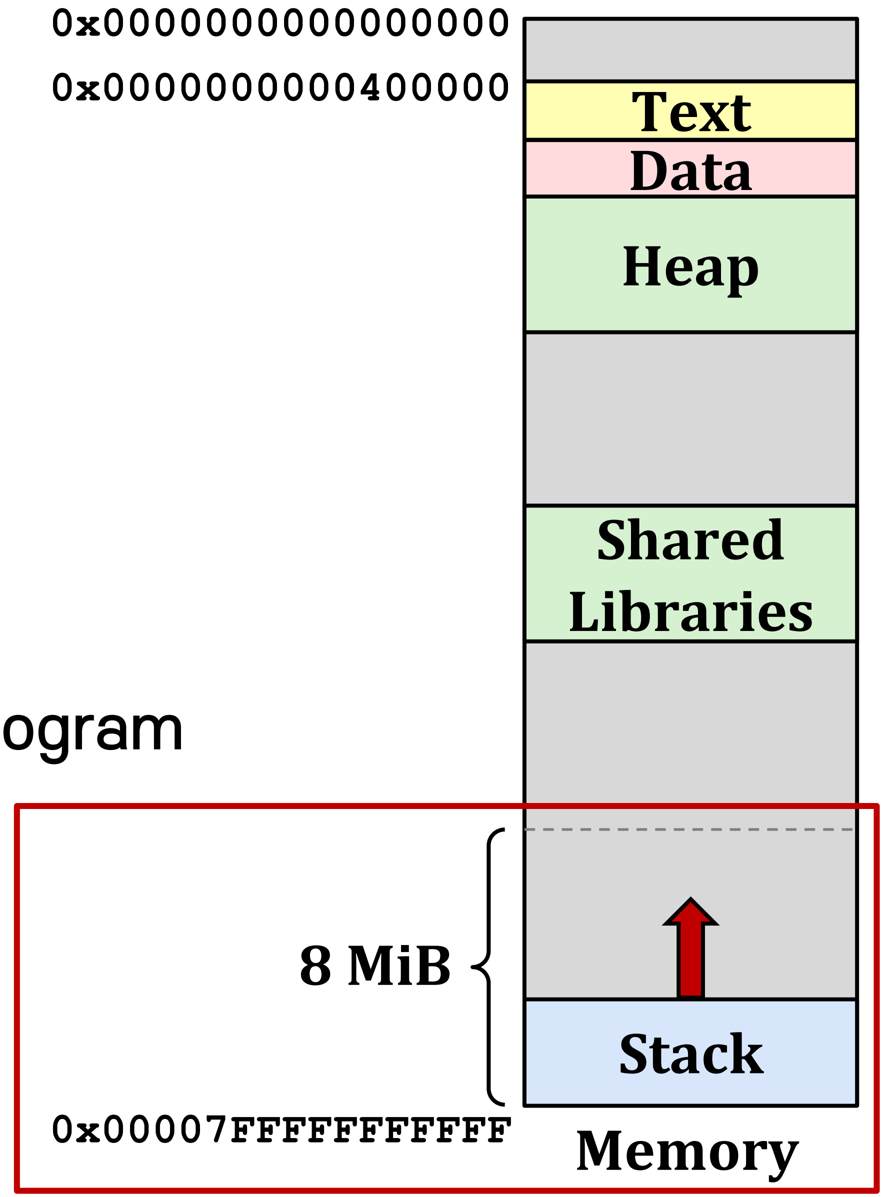
- Background
- Buffer Overflow
    - Vulnerability
    - Protection
- Attack Lab
- Quiz

x86-64 Linux Memory Layout

- Each program has its own address space

- Text and shared libraries
  - Executable machine instructions
  - Read-only

- Stack
  - Stores information about active subroutines of a program



Procedure Control Flow

- Use stack to support procedure call and return

- Procedure call: **`call label`**
    - Push the return address on the stack
        - Return address: The address of the next instruction right after call
    - Jump to **`label`**

- Procedure return: **`ret`**
    - Pop the address from stack
    - Jump to the address

Procedure Data Flow

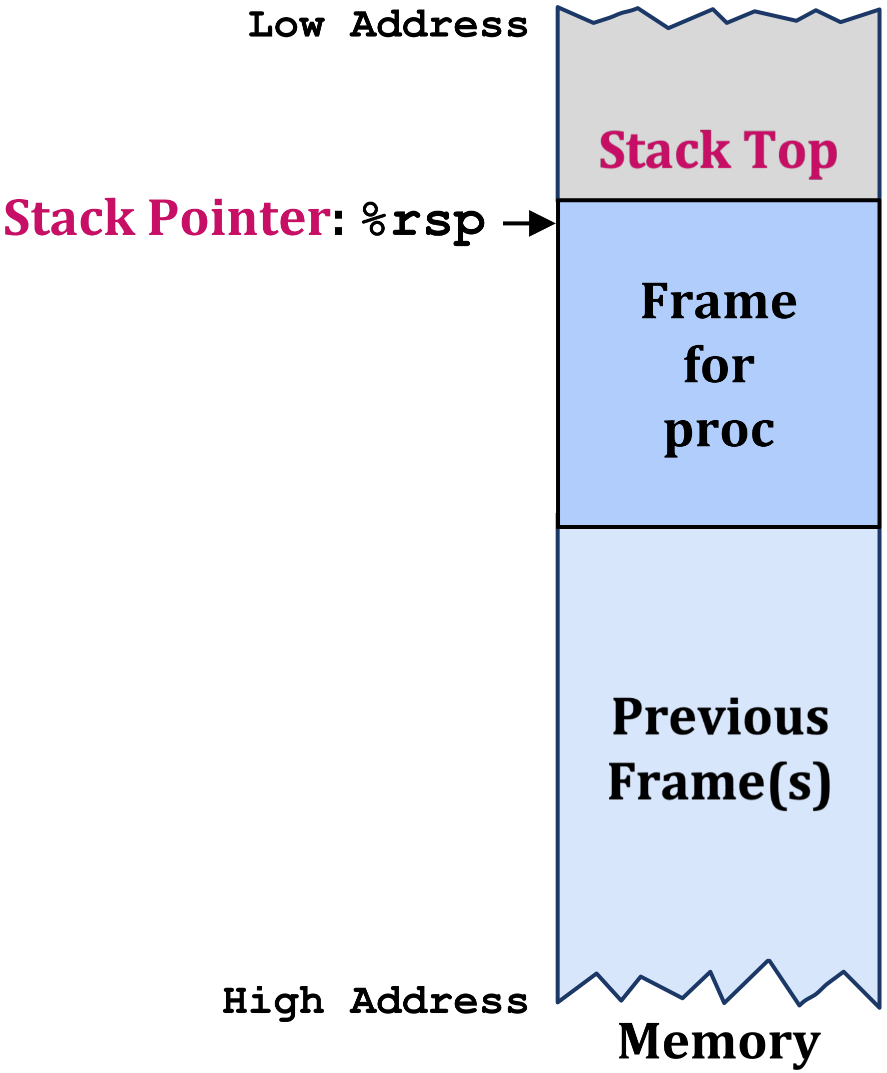
- Passing arguments
  - First six arguments: Registers (`%rdi` → `%rsi` → `%rdx` → `%rcx` → `%r8` → `%r9`)
  - From seventh argument: Stack

- Passing the return value: `%rax` register

Stack Frame

- Dedicated stack area for each procedure

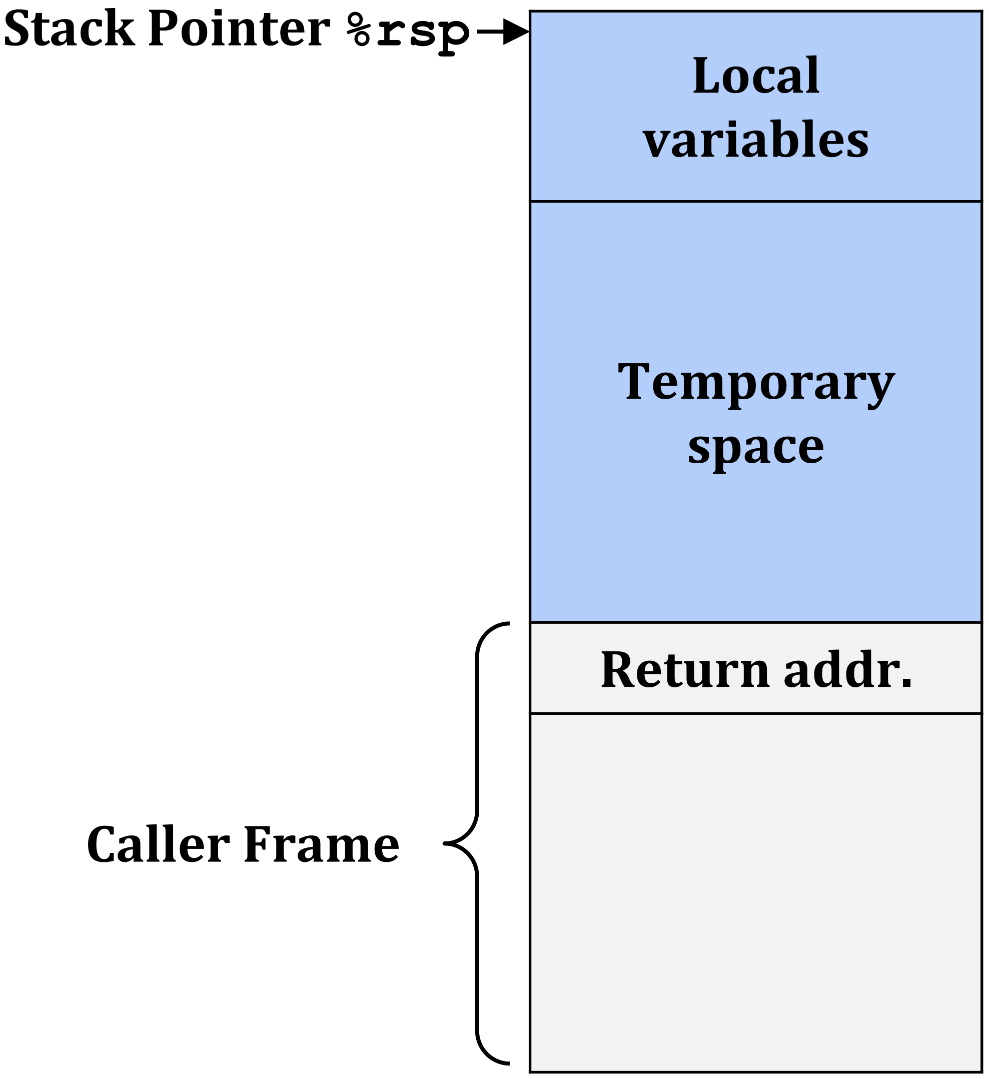
- Management
  - Space allocated when entering the procedure
    - decrease `%rsp`
  - Deallocated when return
    - increase `%rsp`



x86-64/Linux Stack Frame

- Current (callee) stack frame
  - Local variables
  - Temporary space

- Caller stack frame
  - Return address
    - Pushed by `call` instruction

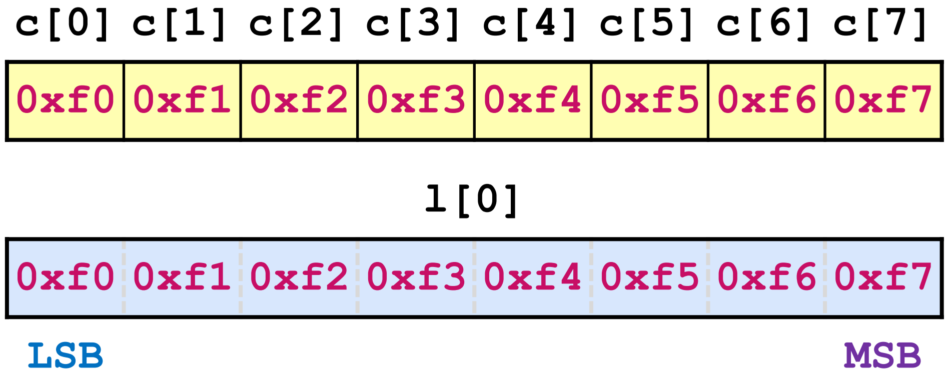


Byte Ordering

- Intel x86 use little endian
    - The least significant byte has the lowest address

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Long 0 == [0x%lx]\n", dw.l[0]);
```

```
Long 0 == [0xf7f6f5f4f3f2f1f0]
```



c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
0xf0	0xf1	0xf2	0xf3	0xf4	0xf5	0xf6	0xf7

l[0]

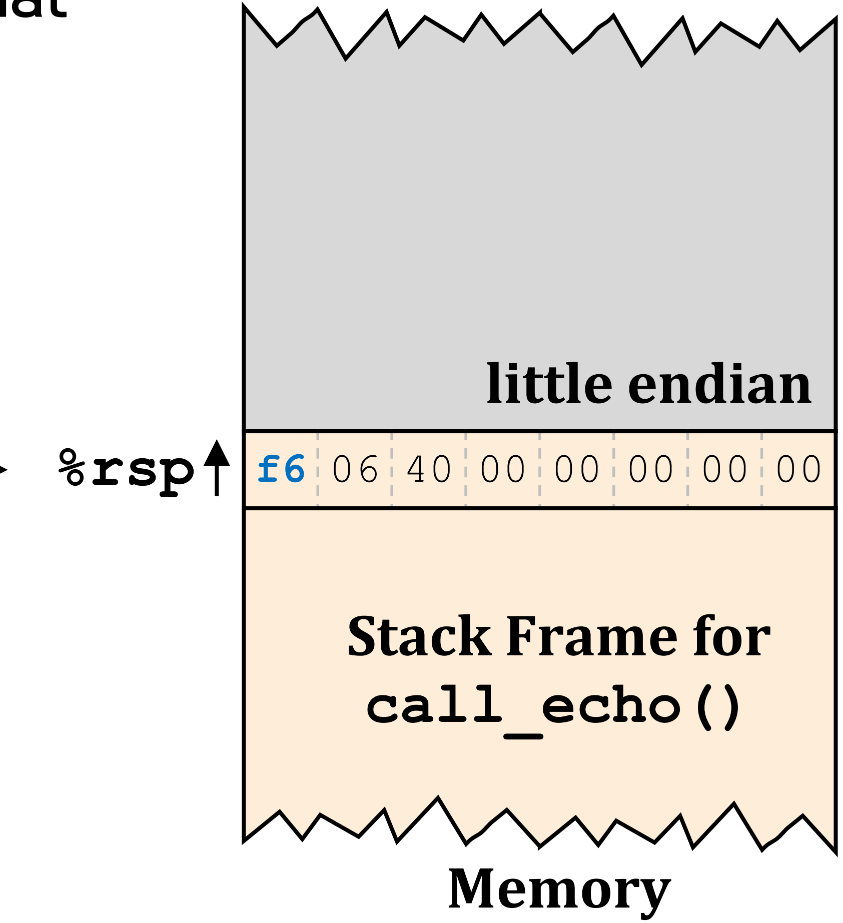
0xf0	0xf1	0xf2	0xf3	0xf4	0xf5	0xf6	0xf7

LSB MSB

Background: Summary

- Push return address to stack and jump to **`echo`**
  - return address is stored in little endian format

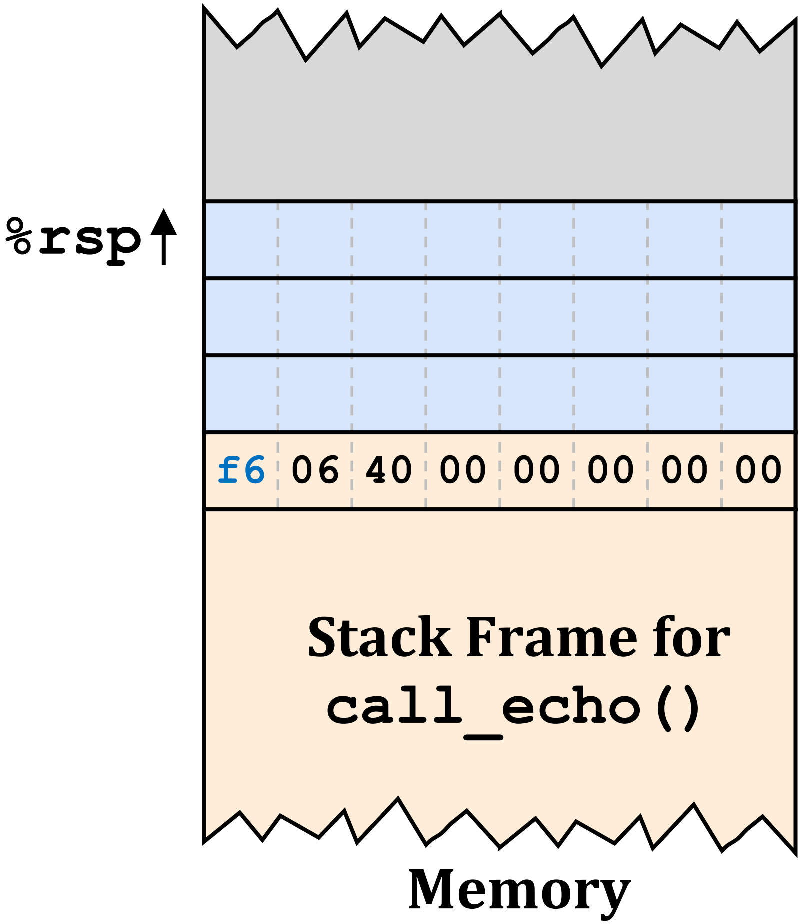
```
00000000004006cf <echo>:
 4006cf:       sub    $24,%rsp
 …
 4006e3:       add    $24,%rsp
 4006e7:       retq

00000000004006e8 <call_echo>:
 4006e8:       sub    $8,%rsp
 …
 4006f1:       callq  4006cf <echo>
 4006f6:       add    $8,%rsp
 4006fa:       retq
```



Background: Summary

- At the beginning of procedure, stack frame is allocated

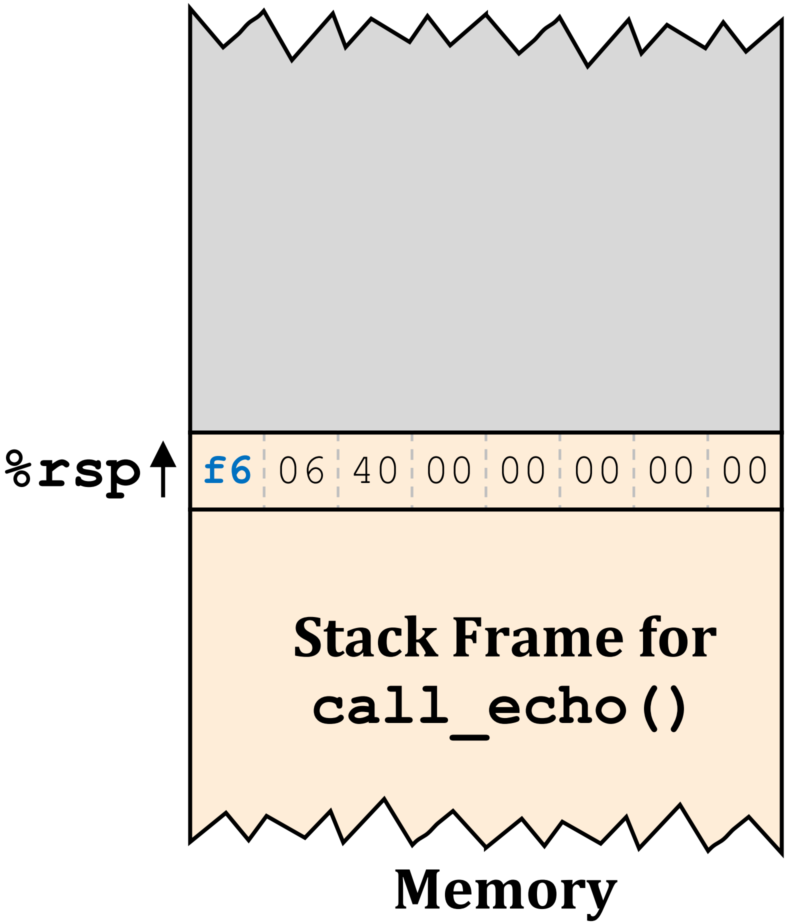
```
00000000004006cf <echo>:
 4006cf:        sub    $24,%rsp
 …
 4006e3:        add    $24,%rsp
 4006e7:        retq

00000000004006e8 <call_echo>:
 4006e8:        sub    $8,%rsp
 …
 4006f1:        callq  4006cf <echo>
 4006f6:        add    $8,%rsp
 4006fa:        retq
```



Background: Summary

- At the end of procedure, stack frame is shrinked

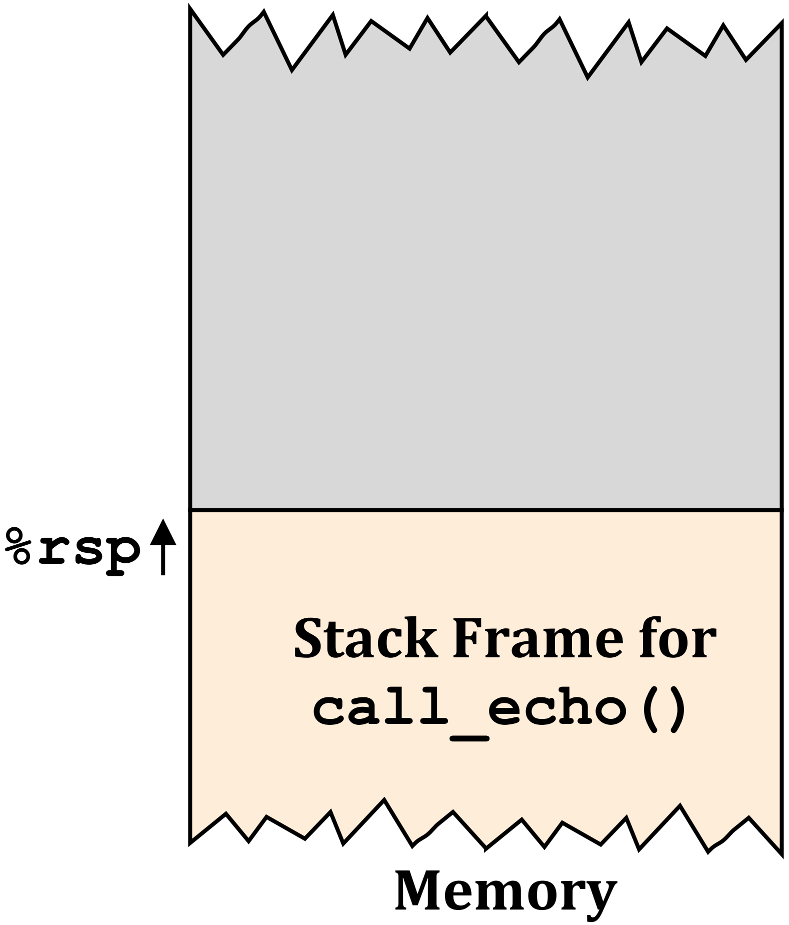
```
00000000004006cf <echo>:
 4006cf:       sub    $24,%rsp
 …
 4006e3:       add    $24,%rsp
 4006e7:       retq

00000000004006e8 <call_echo>:
 4006e8:       sub    $8,%rsp
 …
 4006f1:       callq  4006cf <echo>
 4006f6:       add    $8,%rsp
 4006fa:       retq
```



Background: Summary

- Pop return address from stack and jump to return address

```
00000000004006cf <echo>:
 4006cf:       sub    $24,%rsp
 …
 4006e3:       add    $24,%rsp
 4006e7:       retq

00000000004006e8 <call_echo>:
 4006e8:       sub    $8,%rsp
 …
 4006f1:       callq  4006cf <echo>
 4006f6:       add    $8,%rsp
 4006fa:       retq
```

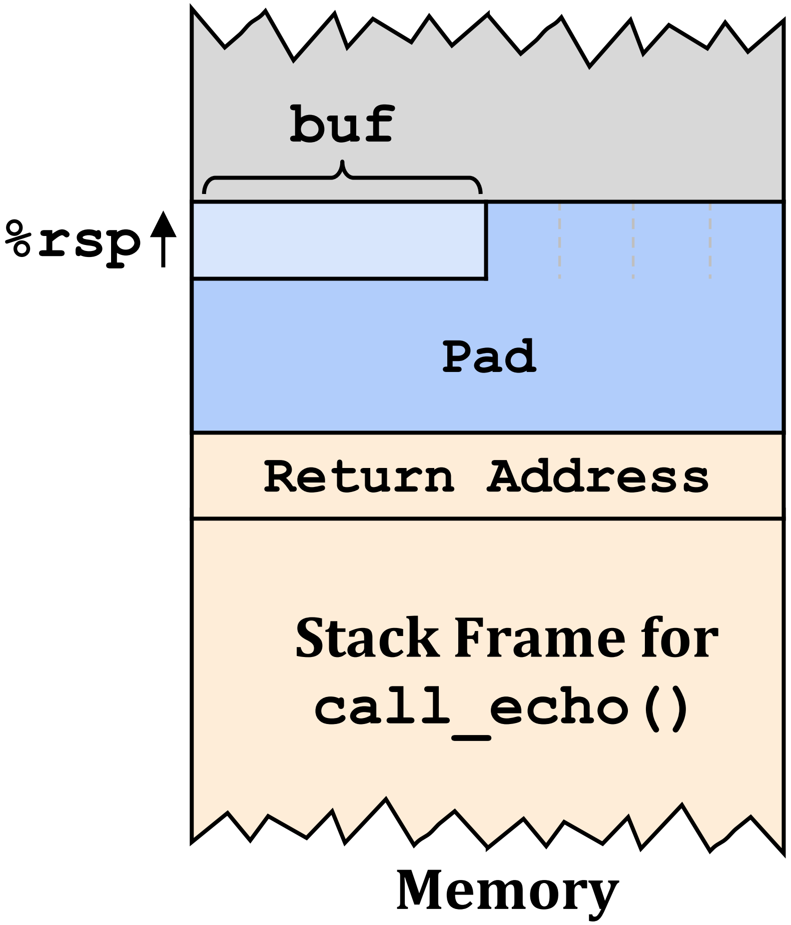


Today’s Agenda

- Background
- Buffer Overflow
  - Vulnerability
  - Protection
- Attack Lab
- Quiz

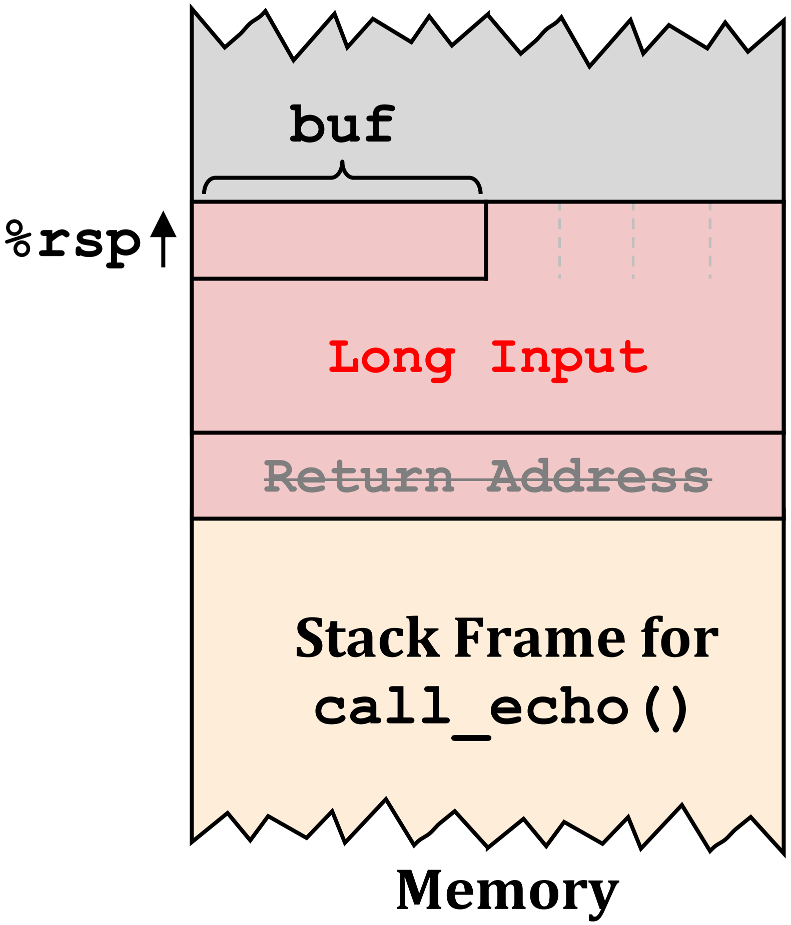
Buffer Overflow

```
/* Echo Line */
void echo(){
    char buf[4];
    gets(buf);   /* similar to scanf */
    puts(buf);   /* similar to printf */
}

void call_echo(){
    echo();
}
```



Buffer Overflow

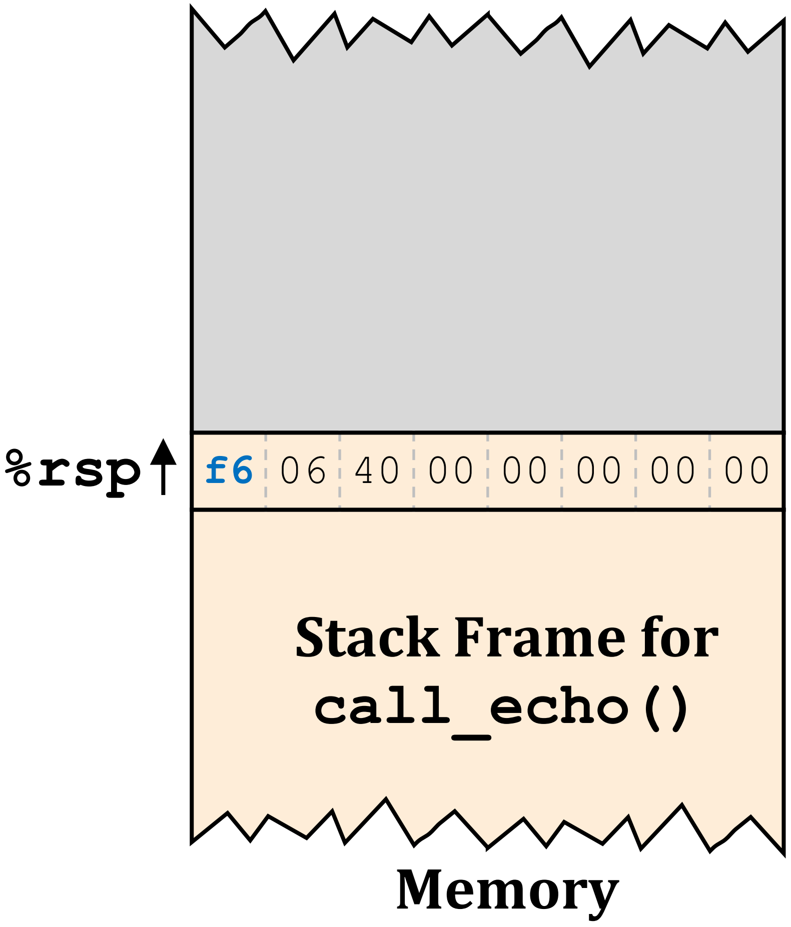
```
/* Echo Line */
void echo(){
    char buf[4]; /* Buffer is too small! */
    gets(buf);
    puts(buf);
}

void call_echo(){
    echo();
}
```



- Unix function gets() has security vulnerability
  - Problem: No limit on the input length
    - Input can exceed buffer and corrupt return address

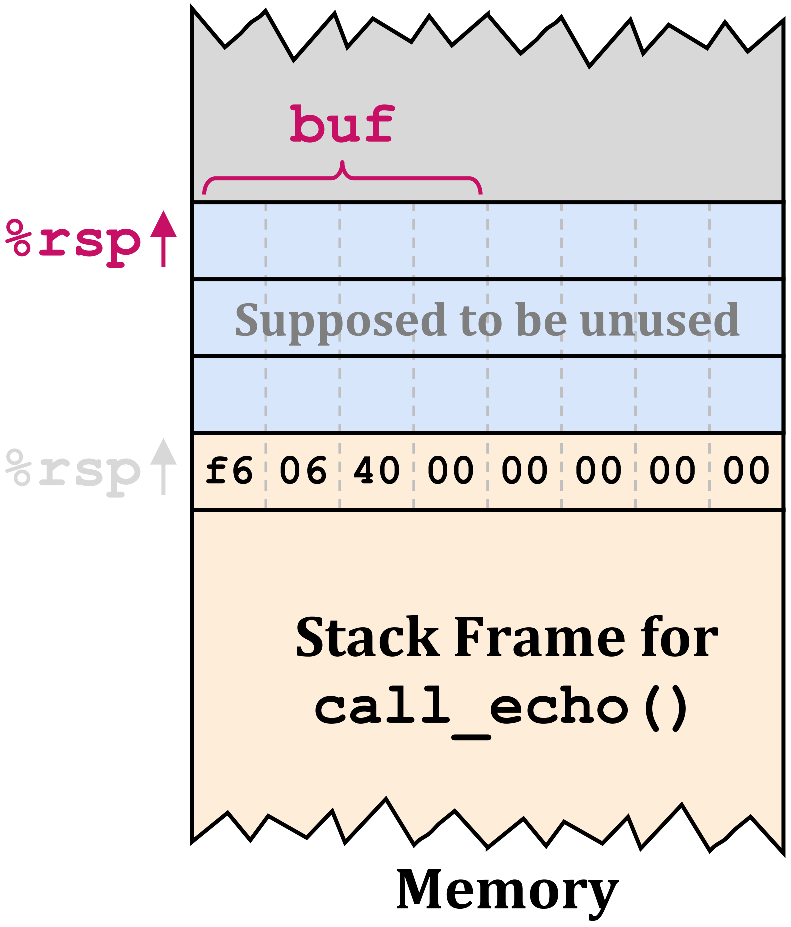
Vulnerable Buffer Code

```
00000000004006cf <echo>:
 4006cf:      48 83 ec 18           sub    $24,%rsp
 4006d3:      48 89 e7              mov    %rsp,%rdi
 4006d6:      e8 a5 ff ff ff        callq  400680 <gets>
 4006db:      48 89 e7              mov    %rsp,%rdi
 4006de:      e8 3d fe ff ff        callq  400520 <puts@plt>
 4006e3:      48 83 c4 18           add    $24,%rsp
 4006e7:      c3                    retq

00000000004006e8 <call_echo>:
 4006e8:      48 83 ec 08           sub    $8,%rsp
 4006ec:      b8 00 00 00 00        mov    $0,%eax
 4006f1:      e8 d9 ff ff ff        callq  4006cf <echo>
 4006f6:      48 83 c4 08           add    $8,%rsp
 4006fa:      c3                    retq
```



Vulnerable Buffer Code

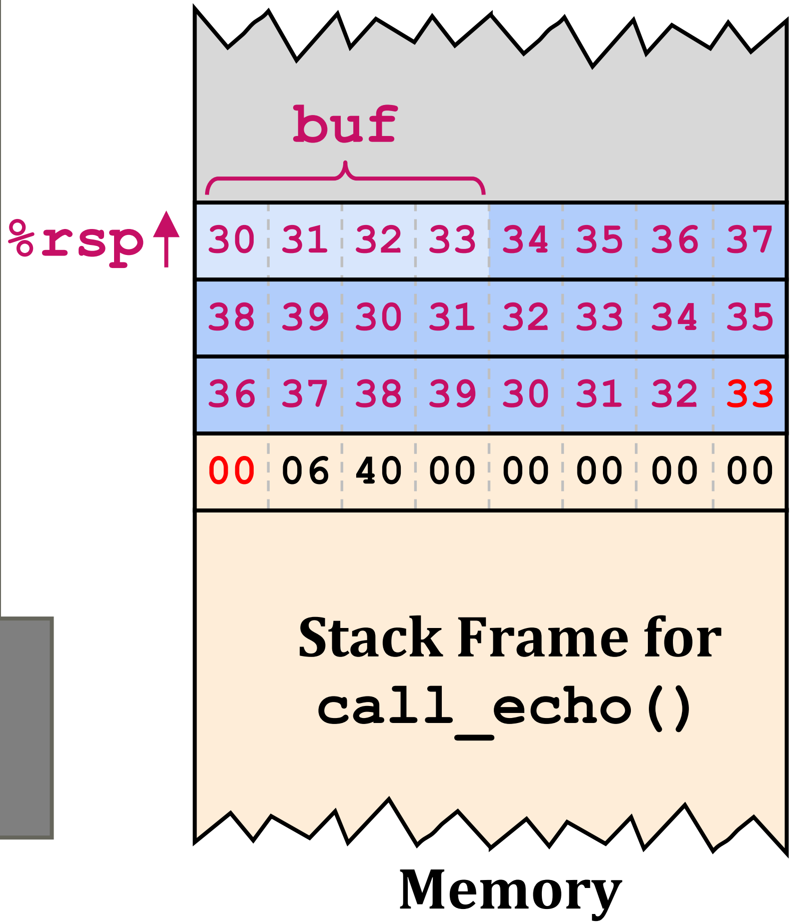
```
00000000004006cf <echo>:
 4006cf:       48 83 ec 18          sub    $24,%rsp
 4006d3:       48 89 e7             mov    %rsp,%rdi
 4006d6:       e8 a5 ff ff ff       callq  400680 <gets>
 4006db:       48 89 e7             mov    %rsp,%rdi
 4006de:       e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:       48 83 c4 18          add    $24,%rsp
 4006e7:       c3                   retq

00000000004006e8 <call_echo>:
 4006e8:       48 83 ec 08          sub    $8,%rsp
 4006ec:       b8 00 00 00 00       mov    $0,%eax
 4006f1:       e8 d9 ff ff ff       callq  4006cf <echo>
 4006f6:       48 83 c4 08          add    $8,%rsp
 4006fa:       c3                   retq
```



Vulnerable Buffer Code

```
00000000004006cf <echo>:
 4006cf:       48 83 ec 18          sub    $24,%rsp
 4006d3:       48 89 e7             mov    %rsp,%rdi
 4006d6:       e8 a5 ff ff ff       callq  400680 <gets>
 4006db:       48 89 e7             mov    %rsp,%rdi
 4006de:       e8 3d fe ff ff       callq  400520 <puts@plt>
 4006e3:       48 83 c4 18          add    $24,%rsp
 4006e7:       c3                   retq

00000000004006e8 <call_echo>:
 4006e8:       48 83 ec 08          sub    $8,%rsp
 4006ec:       b8 00 00 00 00       mov    $0,%eax
 4006f1:       e8 d9 ff ff ff       callq  4006cf <echo>
 4006f6:       48 83 c4 08          add    $8,%rsp
 4006fa:       c3                   retq
```

```
$ ./bufdemo
Type a string:
012345678901234567890123
```



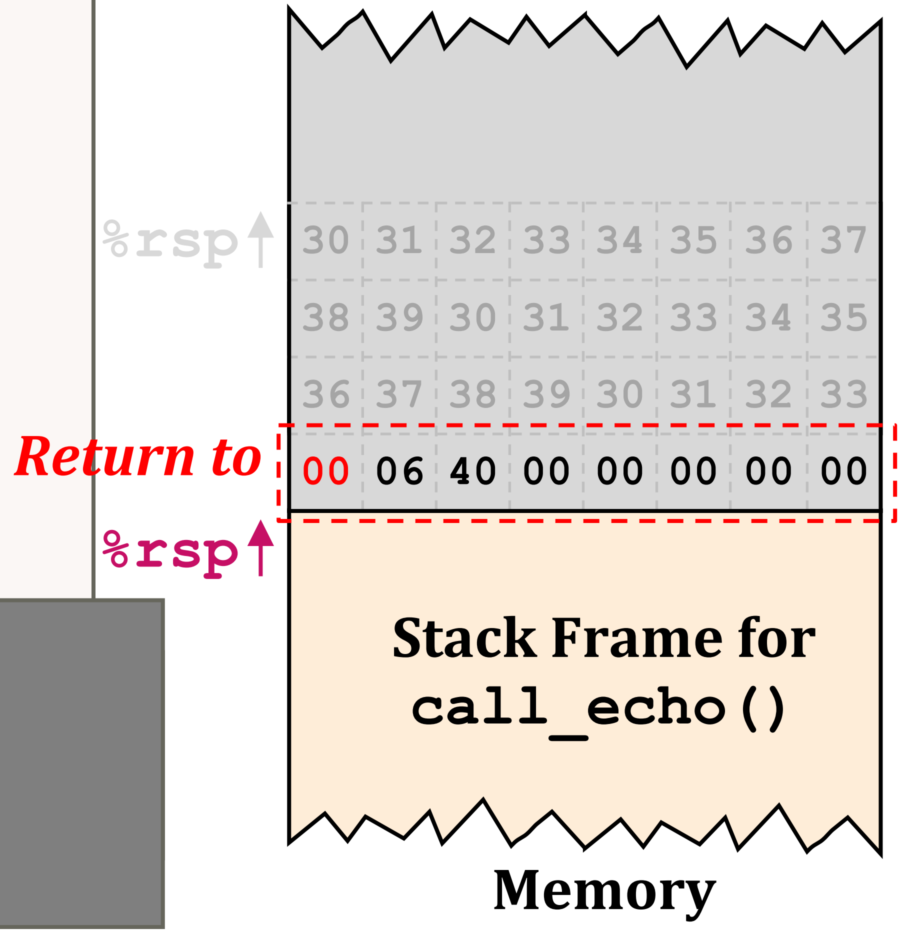
```
30 31 32 33 34 35 36 37
38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33
00 06 40 00 00 00 00 00
```

Vulnerable Buffer Code

```
00000000004006cf <echo>:
 4006cf:     48 83
 4006d3:     48 89
 4006d6:     e8 a5
 4006db:     48 89
 4006de:     e8 3d
 4006e3:     48 83
 4006e7:     c3

00000000004006e8 <ca
 4006e8:     48 83
 4006ec:     b8 00
 4006f1:     e8 d9 ff ff ff          callq   4006cf <echo>
 4006f6:     48 83 c4 08             add     $8,%rsp
 4006fa:     c3                      retq
```

```
register_tm_clones:
    :
  400600:      mov     %rsp,%rbp
  400603:      mov     %rax,%rdx
  400606:      shr     $0x3f,%rdx
  40060a:      add     %rdx,%rax
  40060d:      sar     %rax
  400610:      jne     400614
  400612:      pop     %rbp
  400613:      retq
```

```
$ ./bufdemo
Type a string:
012345678901234567890123
012345678901234567890123
Segmentation Fault
```



```
30 31 32 33 34 35 36 37
38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33
00 06 40 00 00 00 00 00
```

Code Injection Attacks

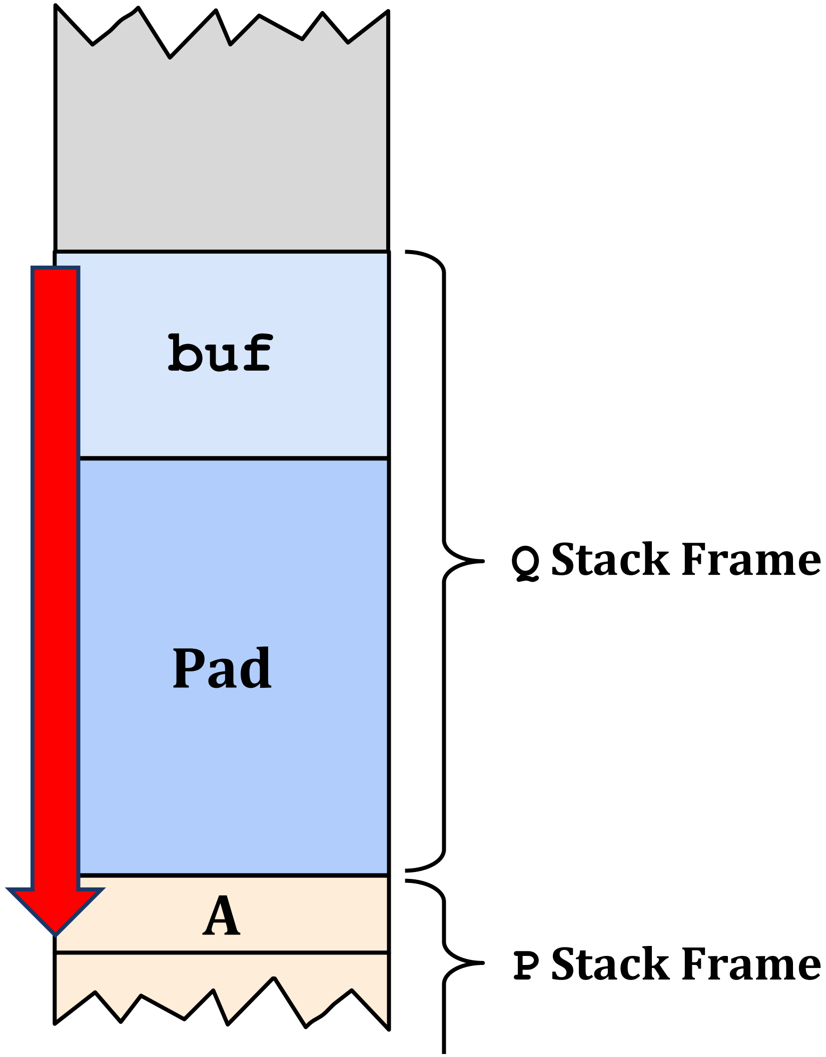
```
void P(){
    Q();
     ⋮
}
```

Return Addr. A

```
int Q(){
    char buf[64];
    gets(buf);
     ⋮
    return val;
}
```

`gets()`:

Overwrite the return address A with the address of buffer (B) + fill the buffer with byte representation of executable code



Stack after Call to `gets()`

Code Injection Attacks

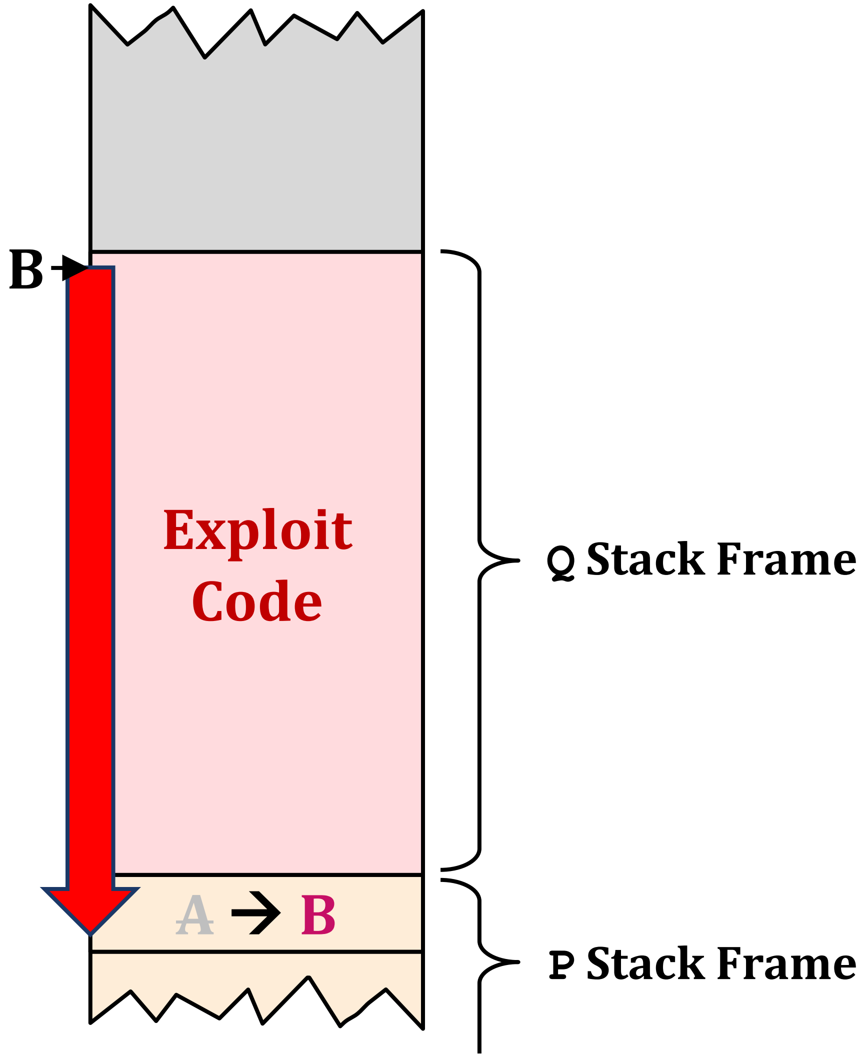
```
void P(){
    Q();
     ⋮
}
```

Return Addr. A

```
int Q(){
    char buf[64];
    gets(buf);
     ⋮
    return val;
}
```

`gets()`:
Overwrite the return address A with the address of buffer (B) + fill the buffer with byte representation of executable code

When `Q` executes `ret`,
will jump to the exploit code



Stack after Call to `gets()`

Hint: Code Injection Attack

- You can use **gcc** and **objdump** to generate byte representation of exploit code
  - Check Appendix B in writeup

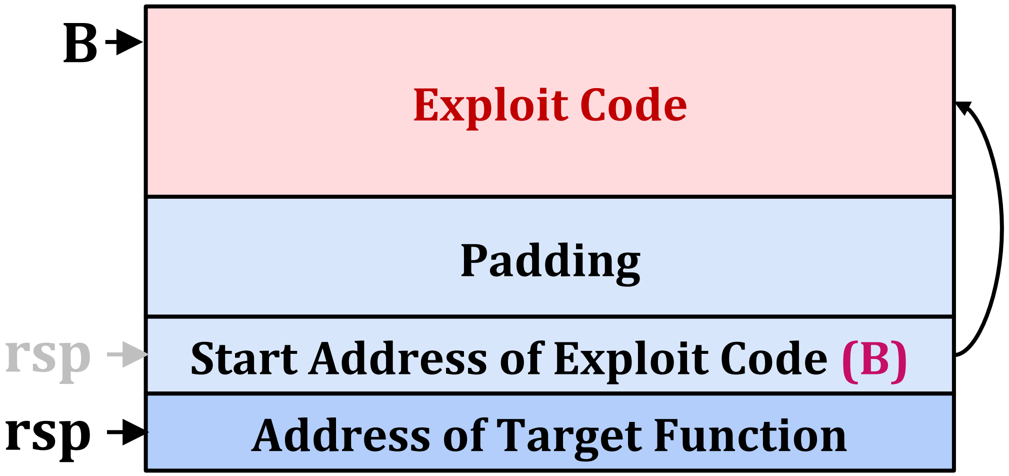
```
[dhgudals1227@programming2 target7]$ vim exploit_code.s
[dhgudals1227@programming2 target7]$ cat exploit_code.s
mov $0x1, %rdi; ret
[dhgudals1227@programming2 target7]$ gcc -c exploit_code.s
[dhgudals1227@programming2 target7]$ objdump -d exploit_code.o > exploit_code.d
[dhgudals1227@programming2 target7]$ cat exploit_code.d

0000000000000000 <.text>:
   0: 48 c7 c7 01 00 00 00 mov    $0x1,%rdi
   7: c3                   retq
```

- You can use **gdb** to obtain start address of exploit code (B)

Hint: Code Injection Attack String

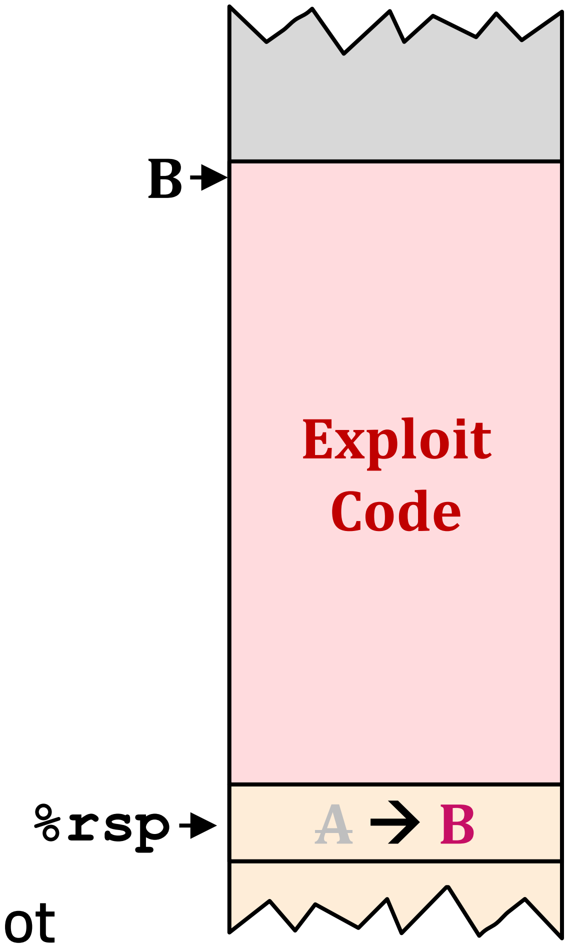
- Focus on a location of `%rsp` after procedure return
  - `%rsp` is pointing just beyond the return address (modified to B)

- Set target function address just beyond exploit code address
  - `ret` instruction in exploit code will jump to target function
  - We can jump into target function after running exploit code



Protection Scheme Against Code Injection Attacks

- Three protection scheme can effectively prevent code injection attacks
  - ASLR, NX, Stack Canary
- ASLR
  - Randomize stack offset → make B unpredictable
- NX
  - Mark stack as a non-executable region
  - Prevent exploit code execution residing in stack
- Stack Canary
  - Memory corruption detection scheme
  - Put a canary just beyond the buffer
  - Before the procedure return, check canary value is changed or not



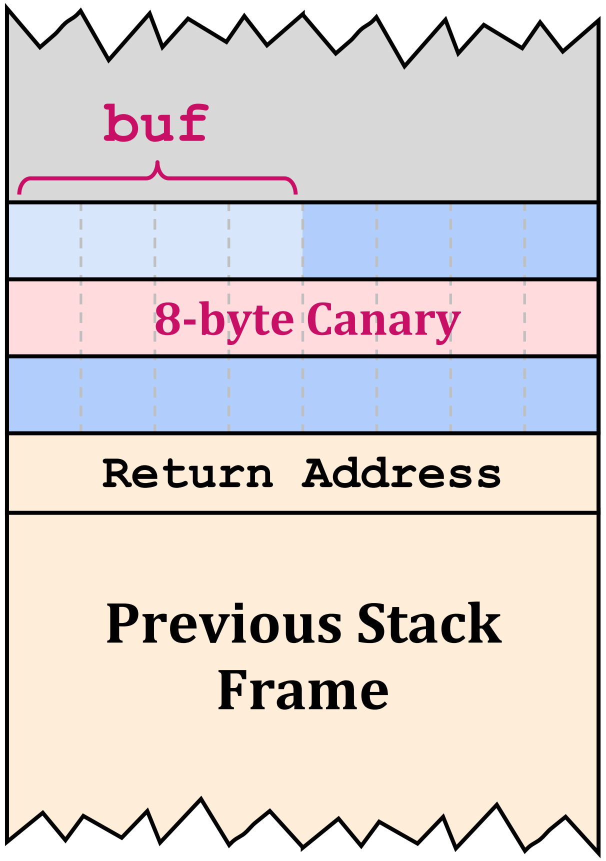
Protection Scheme Against Code Injection Attacks

- Three protection scheme can effectively prevent code injection attacks
  - ASLR, NX, Stack Canary

- ASLR
  - Randomize stack offset → make B unpredictable

- NX
  - Mark stack as a non-executable region
  - Prevent exploit code execution residing in stack

- Stack Canary
  - Memory corruption detection scheme
  - Put a canary just beyond the buffer
  - Before the procedure return, check canary value is changed or not



Return-Oriented Programming (ROP) Attacks

- A more advanced attack technique that can bypass ASLR and NX
  - Cannot bypass stack canary

- Observation
  - Code positions are fixed from run to run
  - Code is executable

- Key Idea
  - Combine original code fragments to build exploit code
    - Each code fragment is called gadget

Gadget Example#1

```
long ab_plus_c(long a, long b, long c){
   return a * b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:   48 0f af fe  imul %rsi,%rdi
  4004d4:   48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:   c3           retq
```

%rax ← %rdi + %rdx

Gadget Address = 0x4004d4

- Use the tail end of existing functions
  - Need `ret` to chain gadgets

Gadget Example#2

```
void setval(unsigned *p){
    *p = 3347663060u;
}
```

```
<setval>:                          Encodes movq %rax, %rdi
   4004d9:   c7 07 d4 48 89 c7   movl   $0xc78948d4,(%rdi)
   4004df:   c3                  retq
```

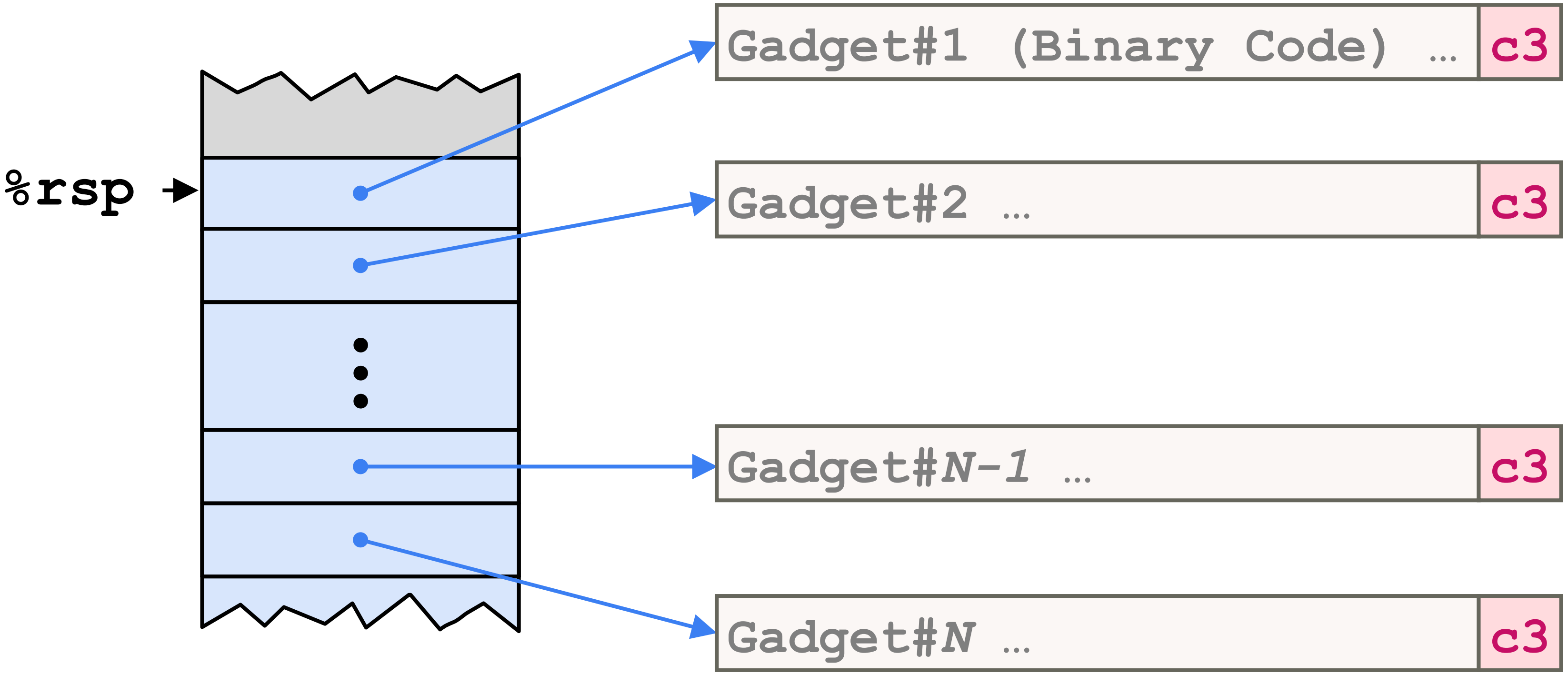
%rdi ← %rax

Gadget Address = 0x4004dc

- Use the tail end of existing functions
  - Need `ret` to chain gadgets

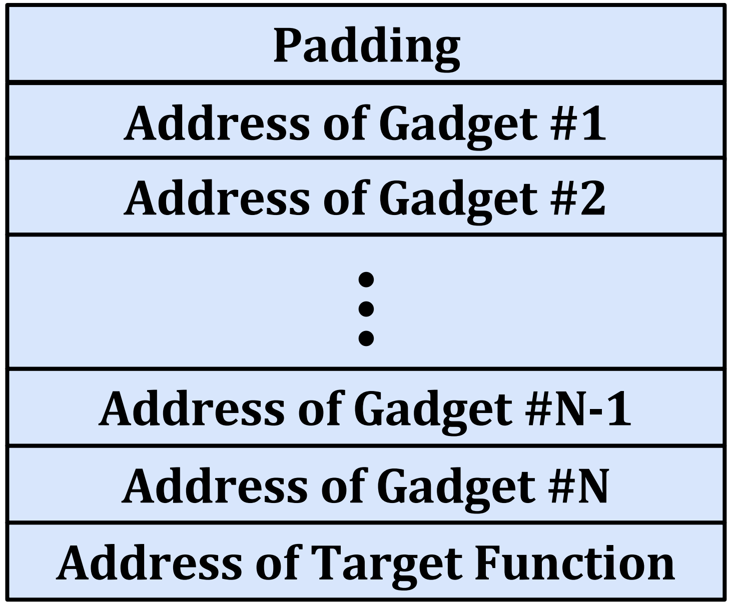
ROP Execution

- Trigger with ret instruction
  - Will start executing Gadget 1
  - `ret` → `pop %rip`
- `ret` in each gadget will start next gadget



Hint: ROP Attack String

- The attack string is composed of three parts
  - Padding to trigger buffer overflow
  - Address of gadgets
    - Do some computations using sequence of existing code fragments
  - Address of target function



Padding

Address of Gadget #1

Address of Gadget #2

⋮

Address of Gadget #N-1

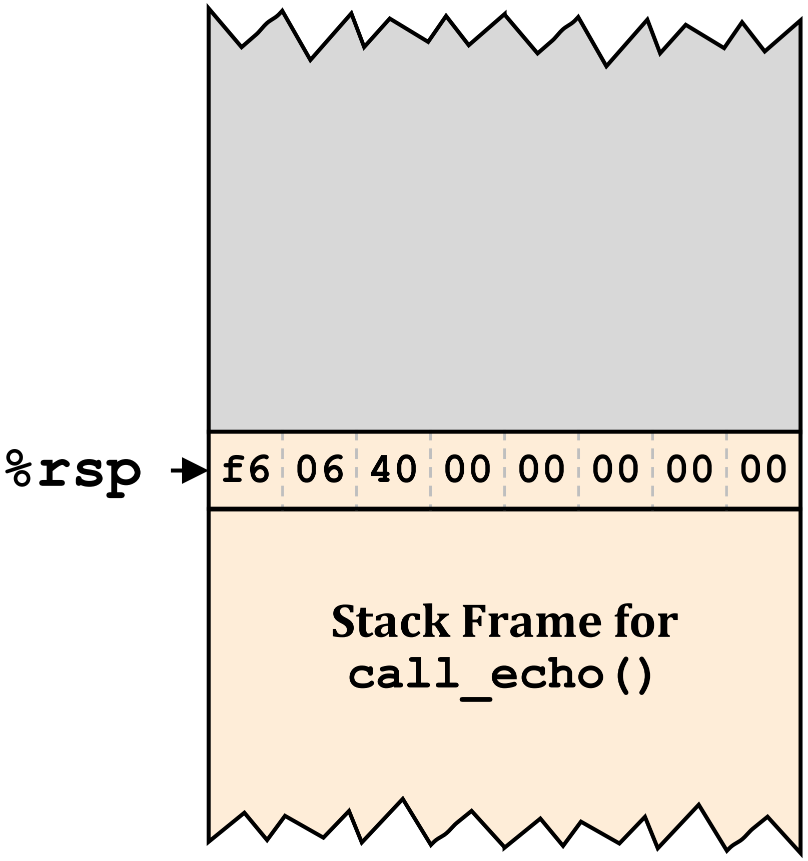
Address of Gadget #N

Address of Target Function

Crafting an ROP Attack String

```
int echo(){
    char buf[4];
    gets(buf);
       ⋮
    return ret
}
```

Attack: Makes `echo()` return `%rdi+%rdx`

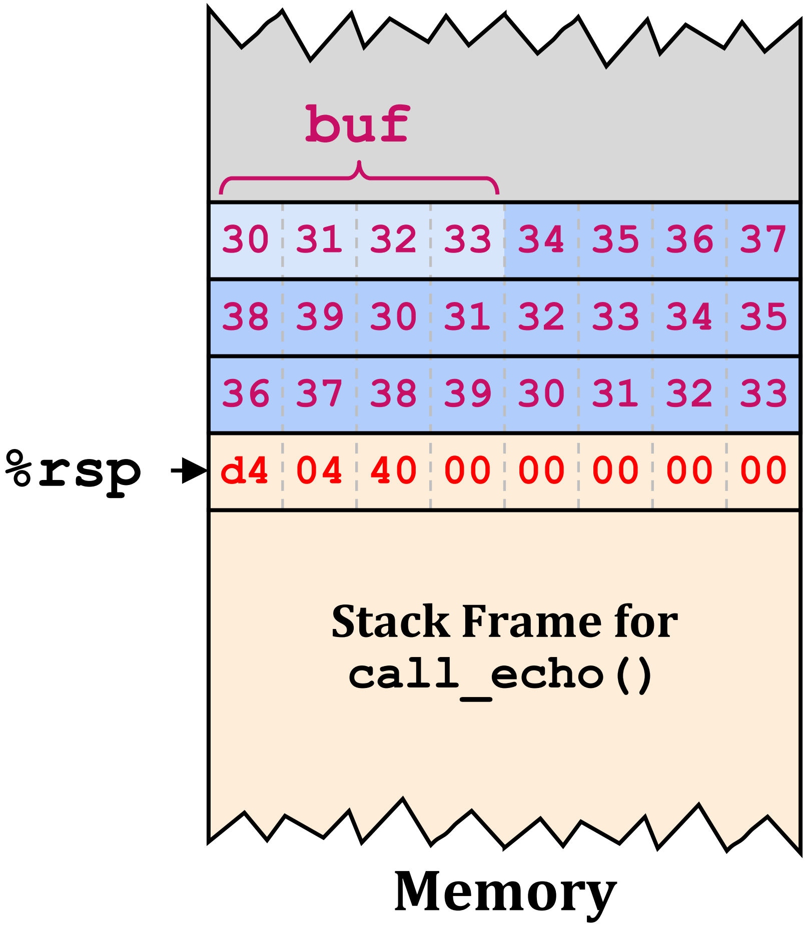


Memory

Gadget: `%rax ← %rdi + %rdx (+ ret)`

```
00000000004004d0 <ab_plus_c>:
  4004d0:   48 0f af fe   imul %rsi,%rdi
  4004d4:   48 8d 04 17   lea (%rdi,%rdx,1),%rax
  4004d8:   c3            retq
```

Crafting an ROP Attack String



```
30 31 32 33 34 35 36 37
38 39 30 31 32 33 34 35
36 37 38 39 30 31 32 33
d4 04 40 00 00 00 00 00
```

```
int echo(){
    char buf[4];
    gets(buf);
      ⋮
    return ret
}
```

Attack: Makes echo() return %rdi+%rdx

Gadget: %rax ← %rdi + %rdx (+ ret)

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

Attack String (HEX)

```
30 31 ... 38 39 30 ... 39 30 ... 33 d4 04 40 00 00 00 00 00
```

Multiple gadgets can corrupt stack upwards

Today’s Agenda

- Background
- Buffer Overflow
    - Vulnerability
    - Protection
- Attack Lab
- Quiz

Attack Lab: Assignment

- Goal: Gain hands-on experience about buffer overflow attacks

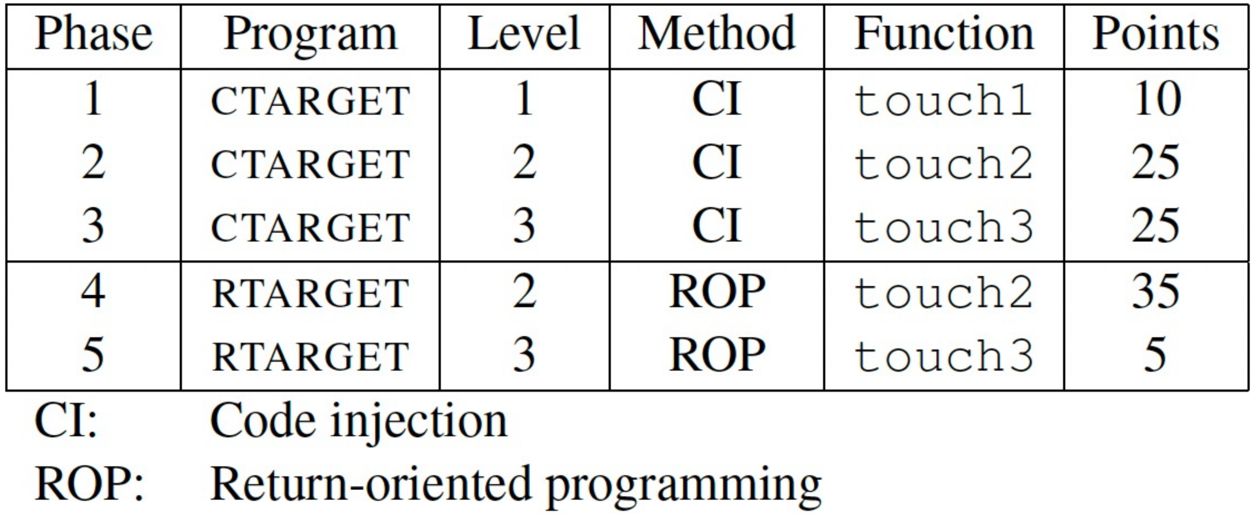
- There are five problems in this lab
  - Three problems are related to code injection attack (CI)
  - Two problems are related to return oriented programming (ROP)

- Due: 11/6 (Wed) 23:59
  - Late submission will not be accepted

- Submit your lab report in pdf
  - Report name: [student id].pdf (e.g., 2023xxxx.pdf)
  - Your report should not exceed 10 pages

Attack Lab: Evaluation

- Score evaluation: Quiz (10%) + Lab report (90%)
  - Lab report evaluation criteria



Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	touch1	10
2	CTARGET	2	CI	touch2	25
3	CTARGET	3	CI	touch3	25
4	RTARGET	2	ROP	touch2	35
5	RTARGET	3	ROP	touch3	5

CI: Code injection
ROP: Return-oriented programming

  - Phase 5 takes only 5% of report score, which will not be critical in overall score
    - If Phase 5 is challenging to you, just leave it (Do not cheat!)

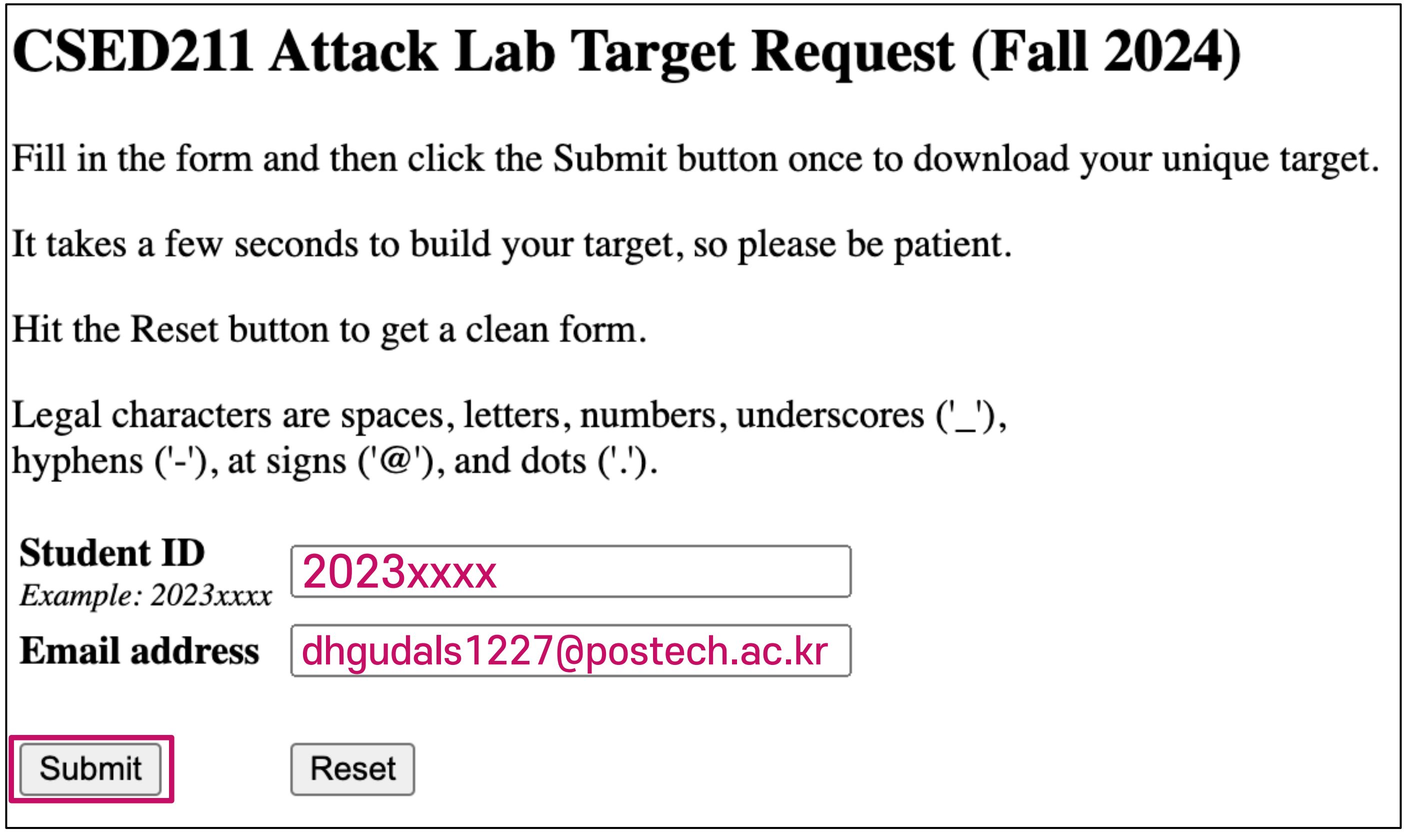
Attack Lab: How to Do

- Connect to the programming server with `SSH` command
  - `$ ssh -p 2022 -L 15213:127.0.0.1:15213 [YourServerID]@programming2.postech.ac.kr`
  - `-p: SSH port number to connect server`
  - `-L: Port tunneling`

- If you are using Xshell, you can use bomblab port forwarding setup

Attack Lab: How to Do

- After login, go to http://127.0.0.1:15213 to download your target,
  - You can access the web server only when the SSH session is alive
  - Enter your information, student ID (학번) and email address



CSED211 Attack Lab Target Request (Fall 2024)

Fill in the form and then click the Submit button once to download your unique target.

It takes a few seconds to build your target, so please be patient.

Hit the Reset button to get a clean form.

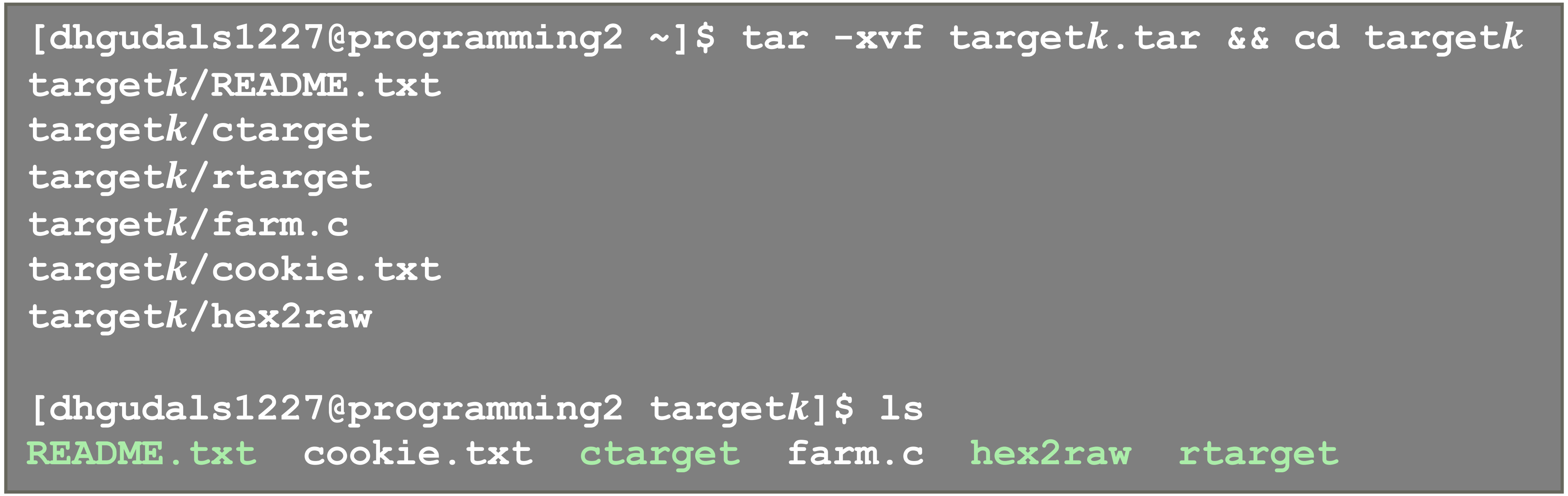
Legal characters are spaces, letters, numbers, underscores ('_'),
hyphens ('-'), at signs ('@'), and dots ('.').

Student ID *Example: 2023xxxx*	2023xxxx
Email address	dhgudals1227@postech.ac.kr

Attack Lab: How to Do

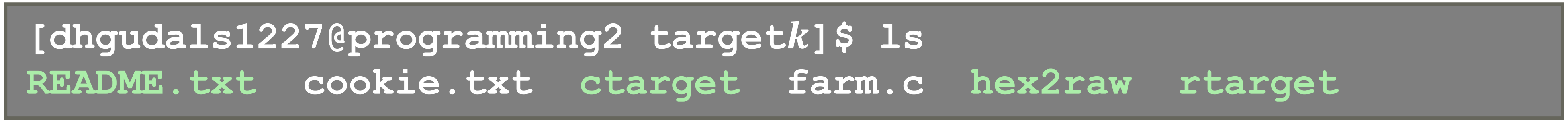
- Upload your target to programming server
  - $ scp -P 2022 [path to target*k*.tar] [povis ID]@programming2.postech.ac.kr:./

- Extract target*k*.tar file and enter the directory
  - $ tar -xvf target*k*.tar && cd target*k*



```
[dhgudals1227@programming2 ~]$ tar -xvf targetk.tar && cd targetk
targetk/README.txt
targetk/ctarget
targetk/rtarget
targetk/farm.c
targetk/cookie.txt
targetk/hex2raw

[dhgudals1227@programming2 targetk]$ ls
README.txt  cookie.txt  ctarget  farm.c  hex2raw  rtarget
```

Attack Lab: How to Do



```
[dhgudals1227@programming2 targetk]$ ls
README.txt  cookie.txt  ctarget  farm.c  hex2raw  rtarget
```

- **`ctarget`**: Program vulnerable to code injection attacks

- **`rtarget`**: Program vulnerable to ROP attacks

- **`hex2raw`**: Tool that can generate raw attack string from .txt file

Attack Lab: How to Do

- Mission: For each problem, design exploit string that can call touch() function
  - Change execution flow of program by triggering buffer overflow

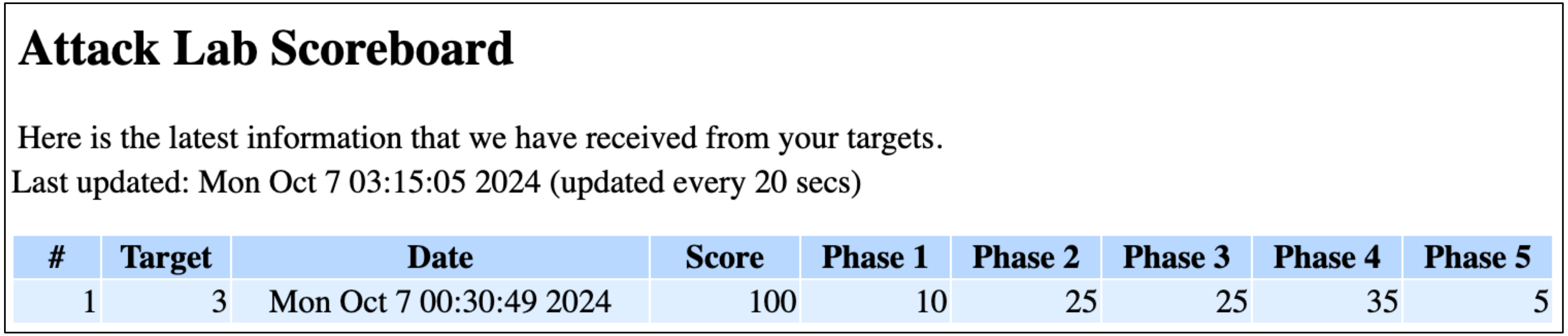
```
[dhgudals1227@programming2 targetk]$ cat ctarget1.txt | ./hex2raw | ./ctarget
Cookie: 0x754e7ddd
Type string: Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

- If exploit is failed, target program will terminate normally (No score penalty)

```
[dhgudals1227@programming2 targetk]$ cat ctarget1.txt | ./hex2raw | ./ctarget
Cookie: 0x754e7ddd
Type string: No exploit. Getbuf returned 0x1
Normal return
```

Attack Lab: How to Do

- Your progress will be automatically uploaded at:
  - http://127.0.0.1:15213/scoreboard
  - You must work on the programming server to properly update scoreboard
    - The score will not be updated if you work in other machines
    - We will evaluate only the problems recorded as solved in the scoreboard for grading your problem



Attack Lab Scoreboard

Here is the latest information that we have received from your targets.
Last updated: Mon Oct 7 03:15:05 2024 (updated every 20 secs)

#	Target	Date	Score	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
1	3	Mon Oct 7 00:30:49 2024	100	10	25	25	35	5

Attack Lab: Tips

- You can get useful information from assembly
  - address of function, size of stack frame, etc.

```
[dhgudals1227@programming2 targetk]$ objdump -d ctarget ctarget.s
[dhgudals1227@programming2 targetk]$ cat ctarget.s

ctarget:     file format elf64-x86-64

Disassembly of section .init:
…
```

Attack Lab: Lab Report Guideline

- Please let me know your target ID (ex. target7 → target ID: 7)

- For each problem, please provide
  1. Screenshot of your attack string

  2. How did you come up with that solution
     - Screenshot of important informations and the way you found it (linux command)
       - function address, stack frame size, start address of exploit code, etc.
       - In ROP, attach screenshots of useful gadgets you found (with their meaning)
     - What was your intention for designing that solution

  3. How your attack string is processed and successfully exploits target
     - Explain detailed control flow and important updates of register value

Cheating Policy

- You can refer
  - Attack lab writeup, lab slides, lecture slides
  - Internet sources that doesn’t involve answers or codes about attack lab

- You should not refer
  - ChatGPT
  - Code and report from a senior who has already taken this course
  - Blogs or github repository that contain solution codes
  - Every other references that violate POSTECHIAN’s Honor

Quiz

- Go to the PLMS, and start the quiz!

[CSED211] Introduction to Computer Software Systems

Lab 4: Attack Lab

Hyeongmin Oh

COMPUTER ARCHITECTURE &
OPERATING SYSTEMS LABORATORY

2024.10.17