Computer Security Capstone

Project IV: Capture The Flag (CTF)

Chi-Yu Li (2021 Spring)

Computer Science Department

National Yang Ming Chiao Tung University

Goal

- Understand the exploitation of basic programming bugs, Linux system knowledge, and reverse-engineering

- You will learn about
  - ☐ Solving basic CTF problems
  - ☐ Investigating C/Linux functions deeply instead of simply using them
  - ☐ What buggy codes are and how they can be exploited

What is CTF?

From Wikipedia

- A traditional outdoor game
  - Two teams each have a flag
  - Objective: to capture the other team's flag
- In computer security, it is a type of cryptosport: a computer security competition
  - Giving participants experience in securing a machine
  - Required skills: reverse-engineering, network sniffing, protocol analysis, system administration, programming, etc.
  - How?
    - A set of challenges is given to competitors
    - Each challenge is designed to give a "Flag" when it is countered

A CTF Example

- A toy CTF

$ python -c 'v = input(); ***print("flag:foobar") if v == "1"*** else print("failed")'

- You should enter “1” to pass the *if* statement and get the flag (flag:foobar)
- Otherwise, “failed” is obtained

Requirements

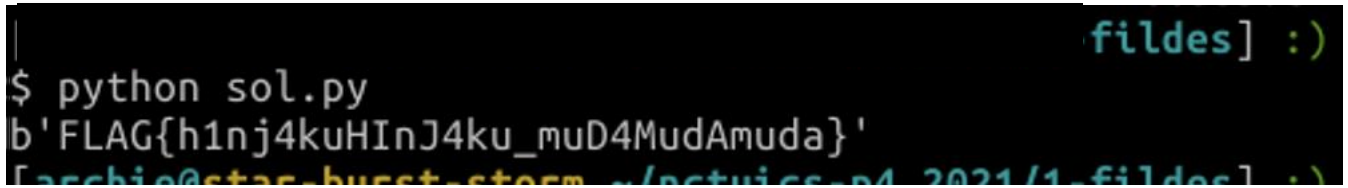
- Linux/Unix environment is required
  - ☐ Connecting to our CTF servers using ‘nc’ for all the tasks except Task I-2
  - ☐ Solving Task I-2 locally
- You are NOT allowed to team up: one student one team
  - ☐ Discussions are allowed between teams, but any collaboration is prohibited
- TA: Chi-Shiuan Liu

How to Proceed?

- Connecting to each CTF server: nc <ip> <port>
  - IP: 140.113.207.240
  - Port is given at each problem
  - The program of each problem runs as a service at the server
  - You can do whatever you are allowed to do

How to Proceed? (Cont.)

- For each CTF problem, you should
  - Analyze its given executable files or source code files
  - Interact with the server to get a flag
  - The flag format: FLAG{xxx}



What If Get Stuck?

- Learn to use “man” in UNIX-like systems
  - If you don’t know something, ask “man”
  - e.g., what is man?
    - $ man man

- Learn to find answers with FIRST-HAND INFORMATION/REFERENCE
  - Google is your best friend (Using ENGLISH KEYWORDS!!)
  - First-hand information: Wikipedia, cppreference.com, devel mailing-list, etc.
  - First-hand reference: papers, standards, spec, man, source codes, etc.
  - Second-hand information: blog, medium, ptt, reddit, stackoverflow post, etc.

Two Tasks

- Task I: Basic CTF problems (70%)
- Task II: CTF beginners (30%)
- Download all given executable and source files from the following link
  - http://140.113.207.240:9820

Task I: Basic CTF Problems

- Task I-1: Fildes (20%)
- Task I-2: Time will stop (20%)
- Task I-3: Translator (30%)

Task I-1: Fildes (20%)

- Goal: Learn about Linux fd & standard I/O streams
- Description
  - Read the code carefully and use your knowledge about *read()* function
  - Read the *read()* man page and find a way to solve this
- Server port: 8831
- Hints
  - $ man stdin
  - $ man 2 read
  - $ man 2 atoi

Task I-2: Time-will-stop (20%)

- Goal: Learn to use tools to inspect binary file
- Description
  - The FLAG is embedded in the binary file “time_will_stop”
  - You can start from having a look at the objects inside the binary file
- No server port; solving it locally
- Recommended tools
  - objdump: display object information from a binary file
  - strings: print strings of printable characters from a binary file
  - GDB - PEDA:
    - Python Exploit Development Assistance for GDB (https://github.com/longld/peda)

Task I-3: Translator (30%)

- Goal: Learn reverse engineering from a binary file to its source code
- Description:
  - The translator encrypts user input and shows the result with printable characters
- Server port: 8833

- Hints
  - You can use code reversing tools like Ida_pro and Ghira
  - You can try to observe what input can cause the flag to be outputted

Task II: CTF Beginner

- Task II-1: Teleportation (10%)
- Task II-2: GOT (10%)
- Task II-3: Secret (10%)

Task II-1: Teleportation (10%)

- Goal: Learn to identify buffer overflow in source codes and overwrite function pointers stored on stack
- Description: Want to know how to teleport to any place you want? Then you got to try this! Give me some spell and the address you want to go, let the magic bring you there!

- Server port: 8834

Task II-1: Teleportation (cont.)

- Recommended tools
  - pwntools (pip install pwntools): a useful python module for pwn
  - GDB - PEDA:
    - Python Exploit Development Assistance for GDB (https://github.com/longld/peda)
    - (gdb-peda) info functions CAN HELP

- Hints
  - No canary
  - No PIE: the address observed in the executable binary file is the virtual address of the process when it's executed

Task II-2: GOT (10%)

- Goal: Learn the vulnerability of format string and redirect code flow with GOT (Global Offset Table) and PLT (Procedure Linkage Table)
- Description: Format is a double-edged sword. It can be really convenient, but it may also leak some secret.
- Server port: 8835
- Recommended tools
  - pwntools (pip install pwntools): a useful python module for pwn
  - GDB - PEDA:
    - Python Exploit Development Assistance for GDB

Task II-2: GOT (cont.)

- Hints
  - No canary
  - No PIE
  - $ man 3 printf
  - Note: If you cannot receive the same amount of data on server version
    - 1. Try to receive data more than one time within one execution.
    - 2. Adjust your mtu. Just for reference, my mtu is 9000 (set your mtu with command $ ifconfig <iface> mtu <amount>)

Example: GOT and PLT

When you call write function	`call 0x4010b0 <write@plt>`

Let's disassemble ***write***(0x4010b0); the second statement will jump to the GOT table of ***write***, which is 0x404028	`gdb-peda$ disassemble 0x4010b0`
	`Dump of assembler code for function write@plt:`
	`0x00000000004010b0 <+0>: endbr64`
	`0x00000000004010b4 <+4>: bnd jmp QWORD PTR [rip+0x2f6d] # 0x404028 <write@got.plt>`
	`0x00000000004010bb <+11>: nop DWORD PTR [rax+rax*1+0x0]`
	`End of assembler dump.`

Let's examine the address; we can see the real address that stored in the GOT table	`gdb-peda$ x 0x404028`
	`0x404028 <write@got.plt>: 0x00401050`
	`gdb-peda$`

Task II-3: Secret (10%)

- Goal: learn to run shellcode with buffer overflow
- Description: Like I said, format is a double-edged sword, did you see any in the code again? If you want to get the secret, you will have to pass my test! Give me your words and tell me what you want me to do, I might let you pass the test.

- Server port: 8836

Task II-3: Secret

- Hints#1
  - No canary
  - NX (No-eXecute) is disabled: executing instruction on memory for data storage is possible
  - No PIE: the address observed in the executable binary file is the virtual address of the process when it’s executed

Task II-3: Secret (Cont.)

- Recommended tools
  - pwntools (pip install pwntools): useful python module for pwn
  - objdump: display information in object files
  - GDB - PEDA:
    - Python Exploit Development Assistance for GDB (https://github.com/longld/peda)
- Hints#2
  - Read the source code carefully, can you see something familiar with the previous challenge?
  - Can you find out the address of the beginning of the buf(see the source code)?
  - Another option to generate shellcode, please check the following website
    - http://shell-storm.org/shellcode/ , the version of my machine is linux/x86-64

Task II-3: Secret (Cont.)

- An example: run shellcode using pwntools to get a flag
  - ☐ cd sample-shellcode
  - ☐ python3 sol.py
  - ☐ cat flag

File: **sol.py**

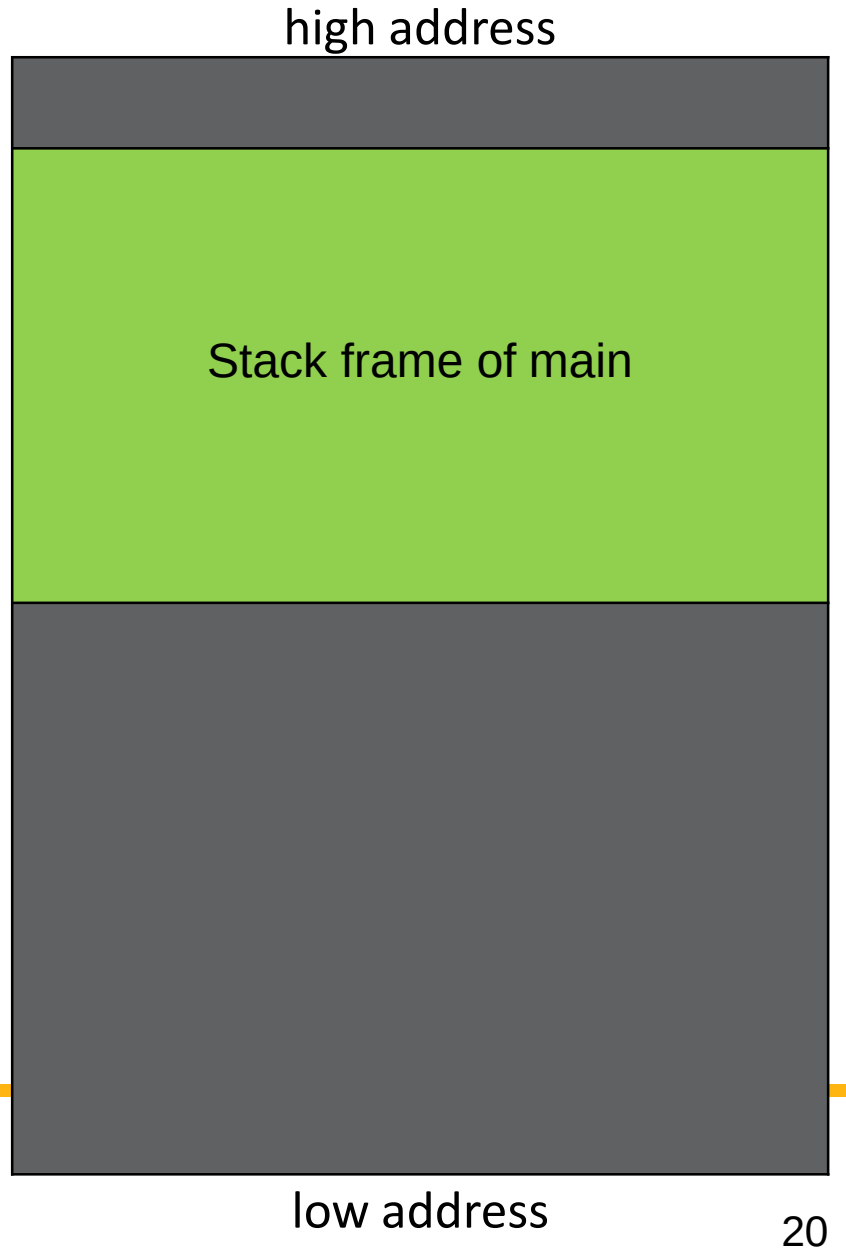
```
1  from pwn import *
2  context.arch = 'amd64'
3  p = process('./shellcode')
4  # To connect to tcp server
5  # p = remote('ip', port)
6  shellcode = asm(shellcraft.amd64.linux.sh())
7  p.send(shellcode)
8  p.interactive()
```

Machine code

Assembly

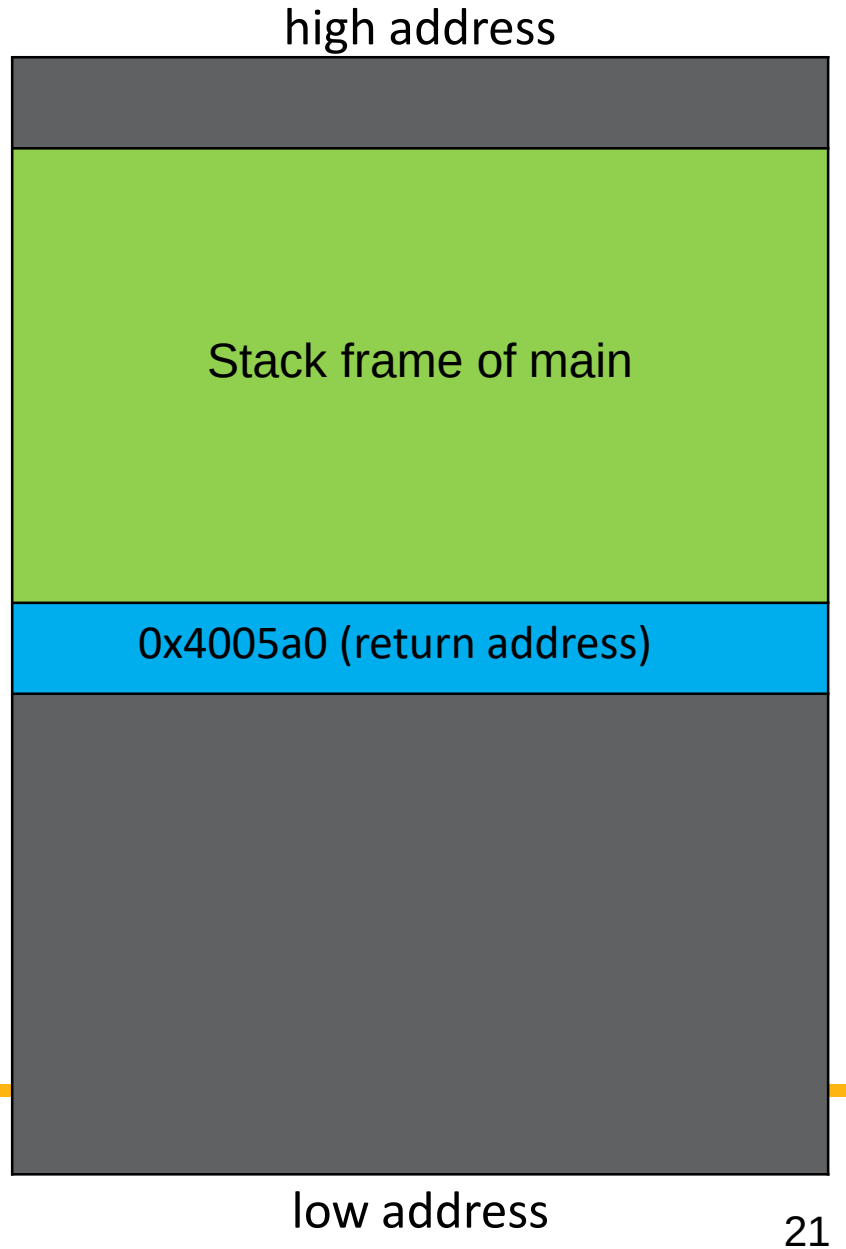
Example: Stack frame during a function call

```
                                                    rbp →
func:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30          Call fun = push next_rip
        …                                 jmp func
        move eax, 0x0                               rsp →
        leave
        ret
main:
        …
rip →   call func
        mov eax, 0x0 // address 0x4005a0
        …
```



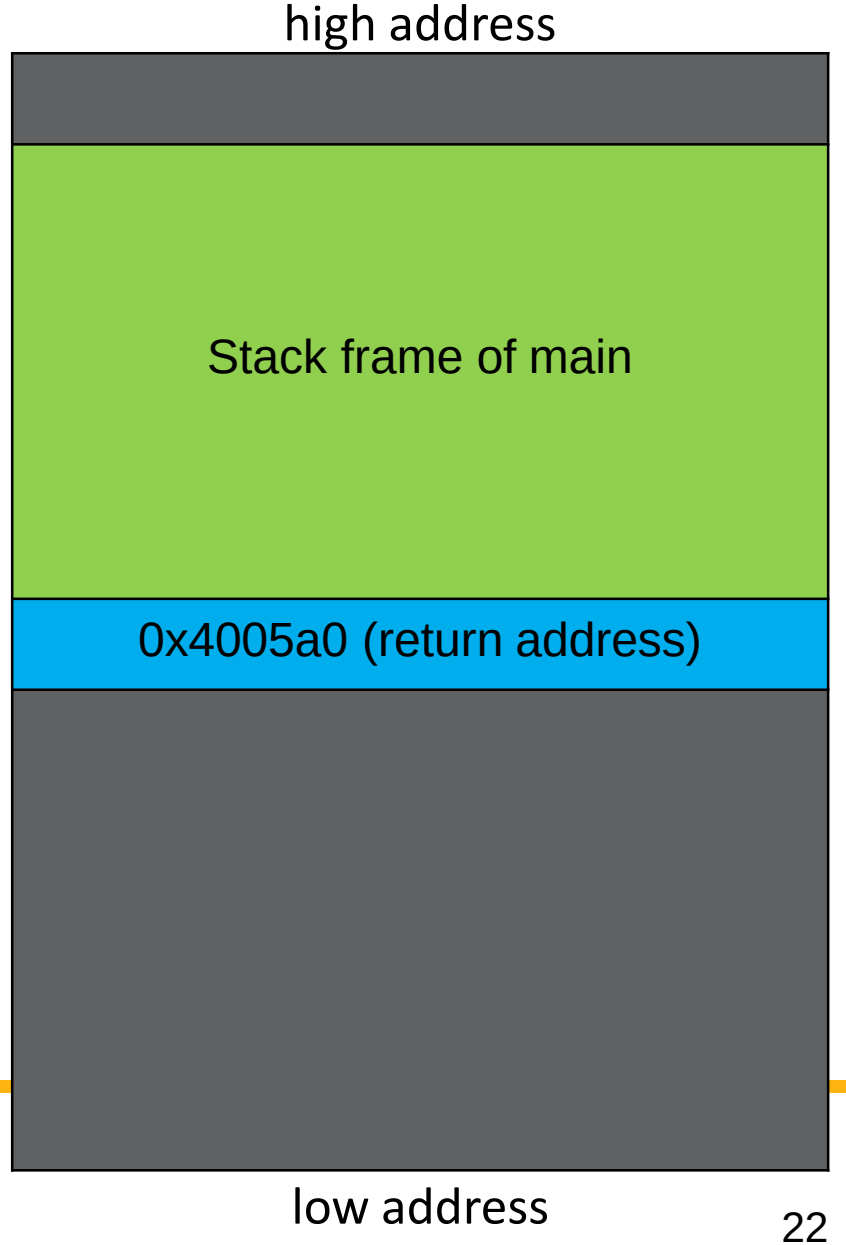
Example: Stack frame during a function call

```
                                                    rbp →
func:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30          Call fun = push next_rip
        …                                jmp func
        move eax, 0x0                               rsp →
        leave                                       rsp →
        ret
main:
        …
rip →   call func
        mov eax, 0x0 // address 0x4005a0
        …
```



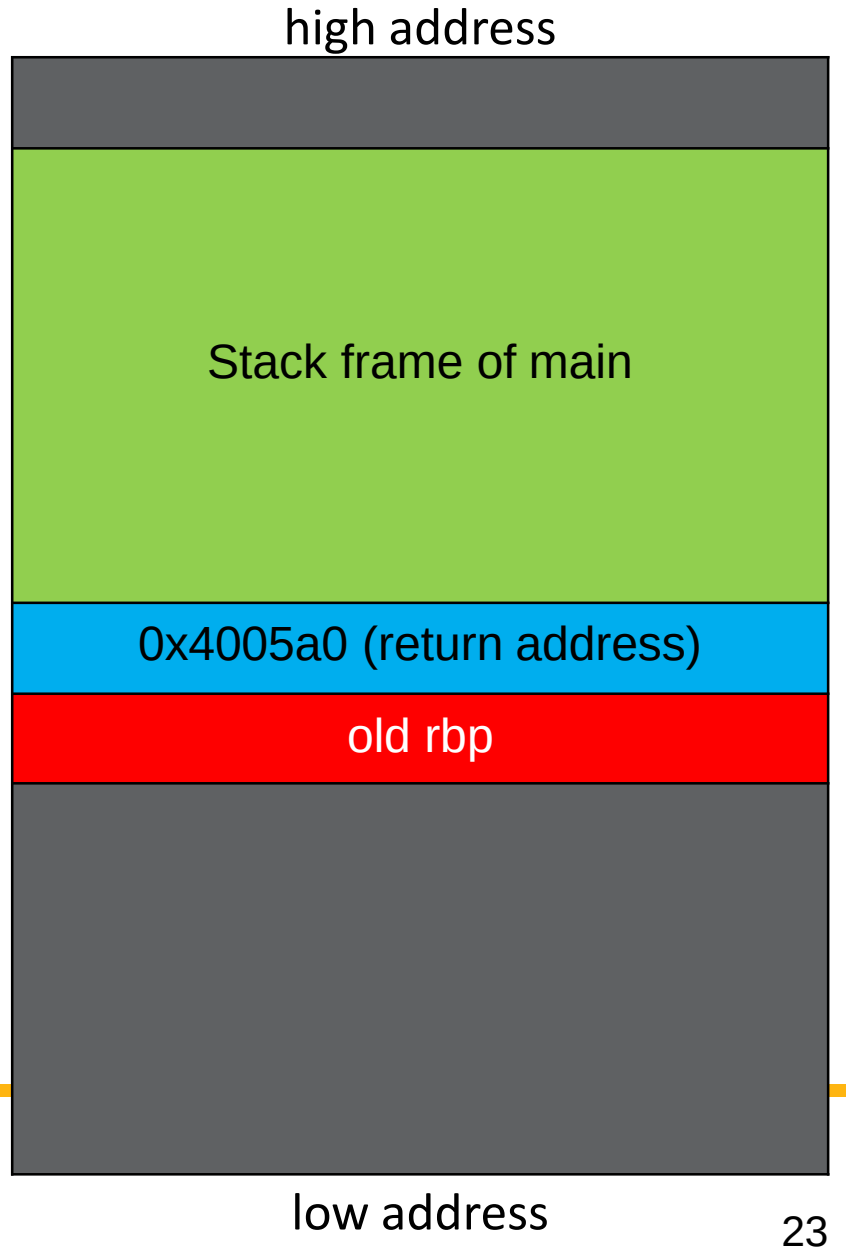
Example: Stack frame during a function call

```
func:
rip →   push rbp
        mov rbp, rsp
        sub rsp, 0x30
        …
        move eax, 0x0
        leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```



Example: Stack frame during a function call

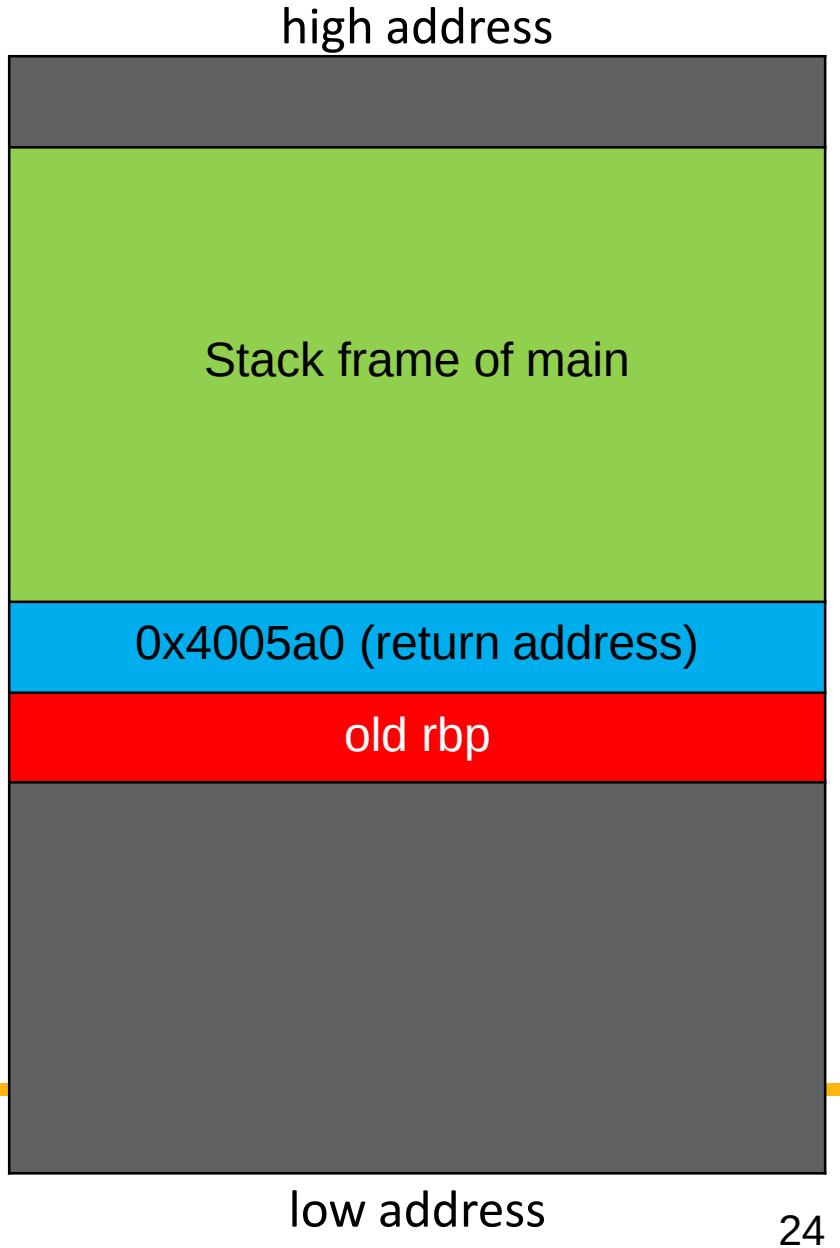
```
                                    rbp →
func:
        push rbp
rip →   mov rbp, rsp
        sub rsp, 0x30
        …
        move eax, 0x0
        leave
        ret                         rsp →
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```



Example: Stack frame during a function call

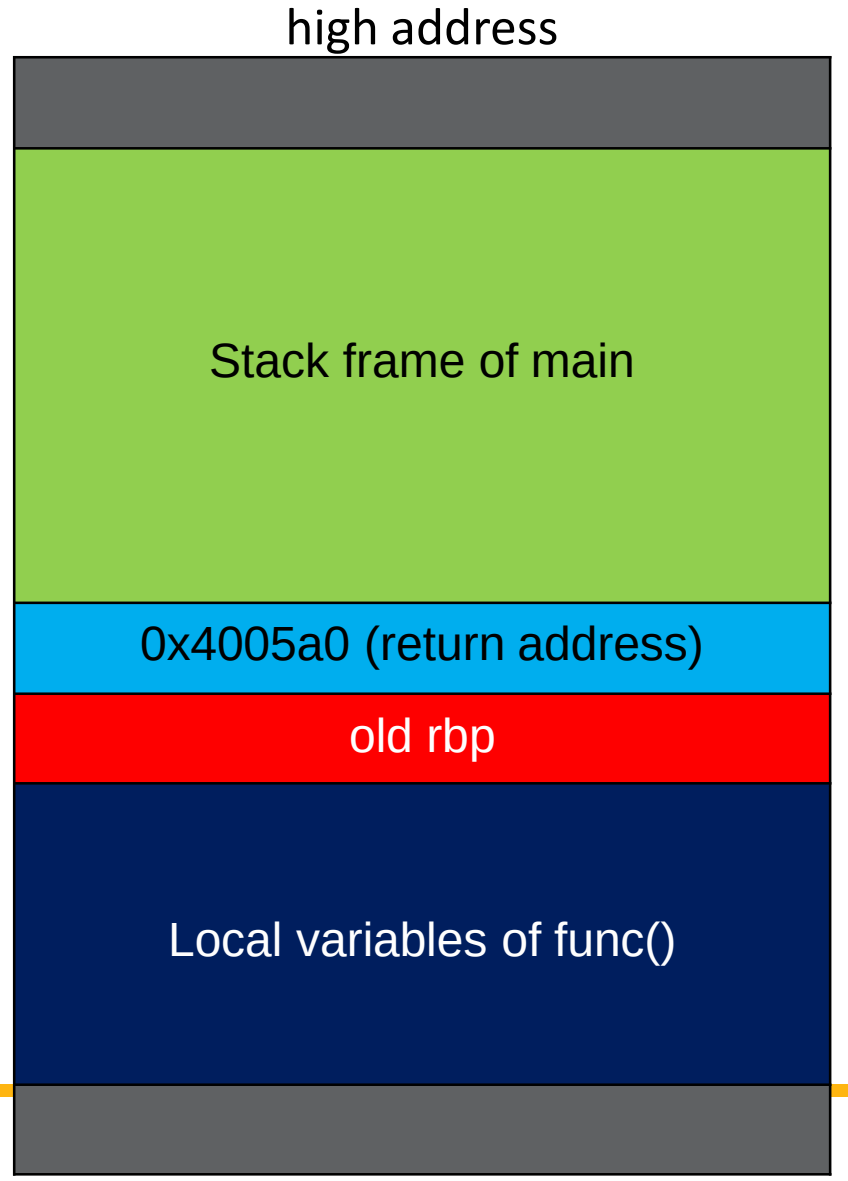
```
func:
        push rbp
        mov rbp, rsp
rip →   sub rsp, 0x30
        …
        move eax, 0x0
        leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```

rbp → rsp →



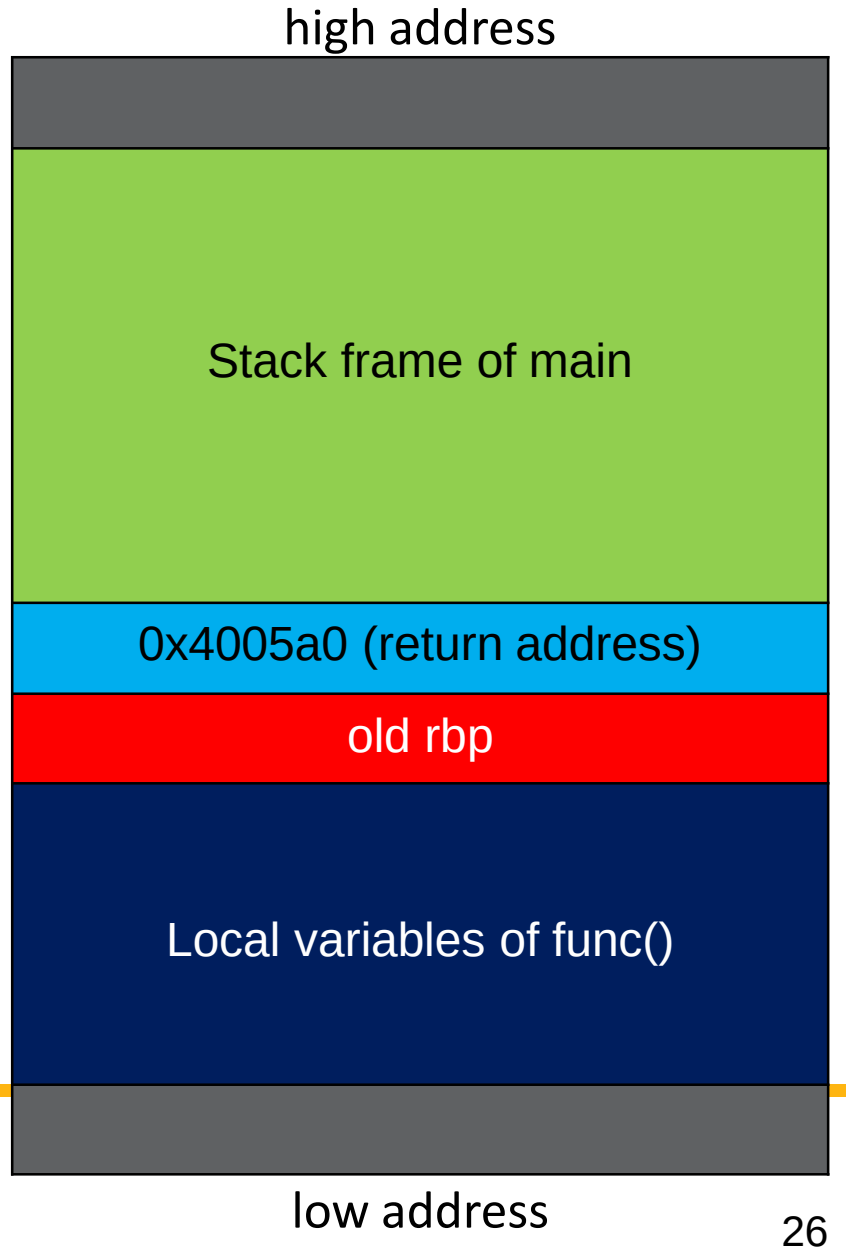
Example: Stack frame during a function call

```
func:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30
rip →   ...
        move eax, 0x0
        leave
        ret
main:
        ...
        call func
        mov eax, 0x0 // address 0x4005a0
```



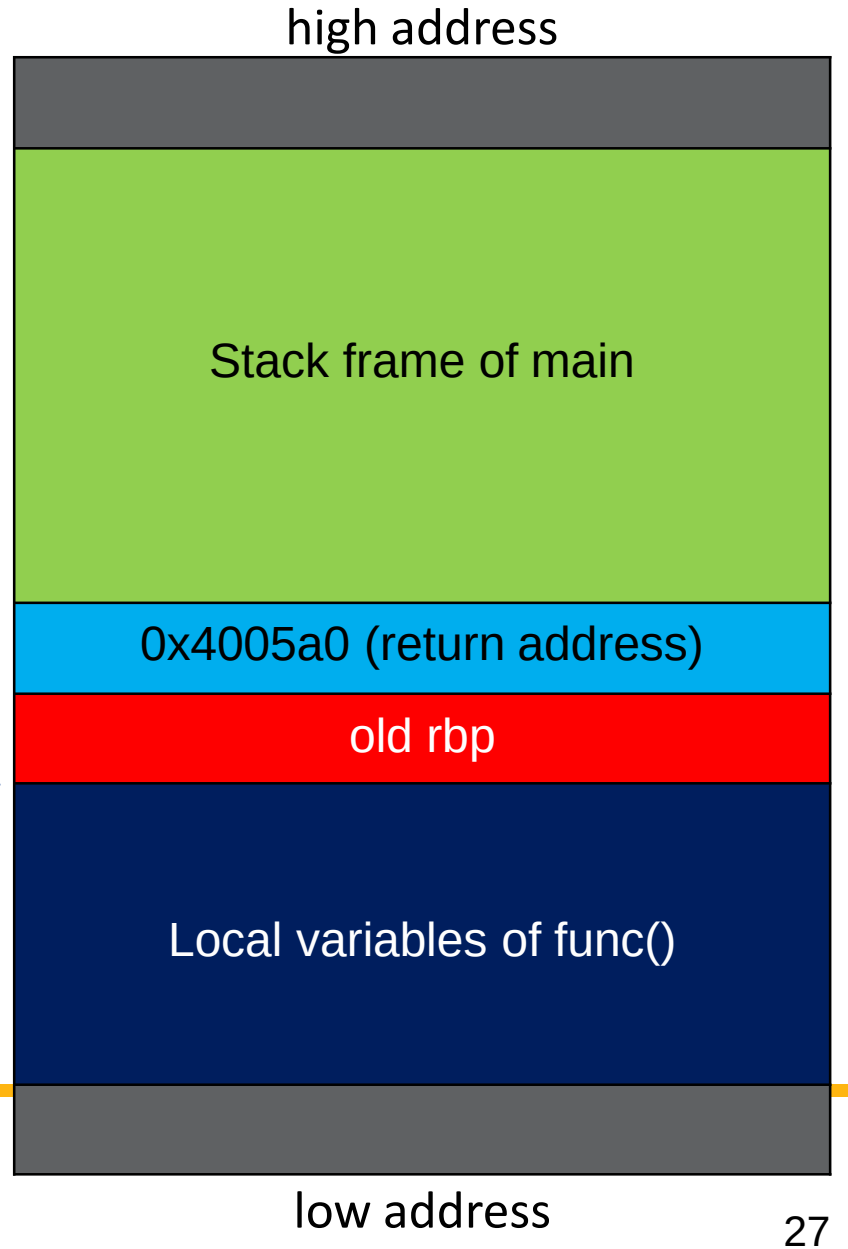
Example: Stack frame during a function call

```
func:
        push rbp                    leave = mov rsp, rbp
        mov rbp, rsp                        pop rbp
        sub rsp, 0x30
        …
        move eax, 0x0
rip →   leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
```



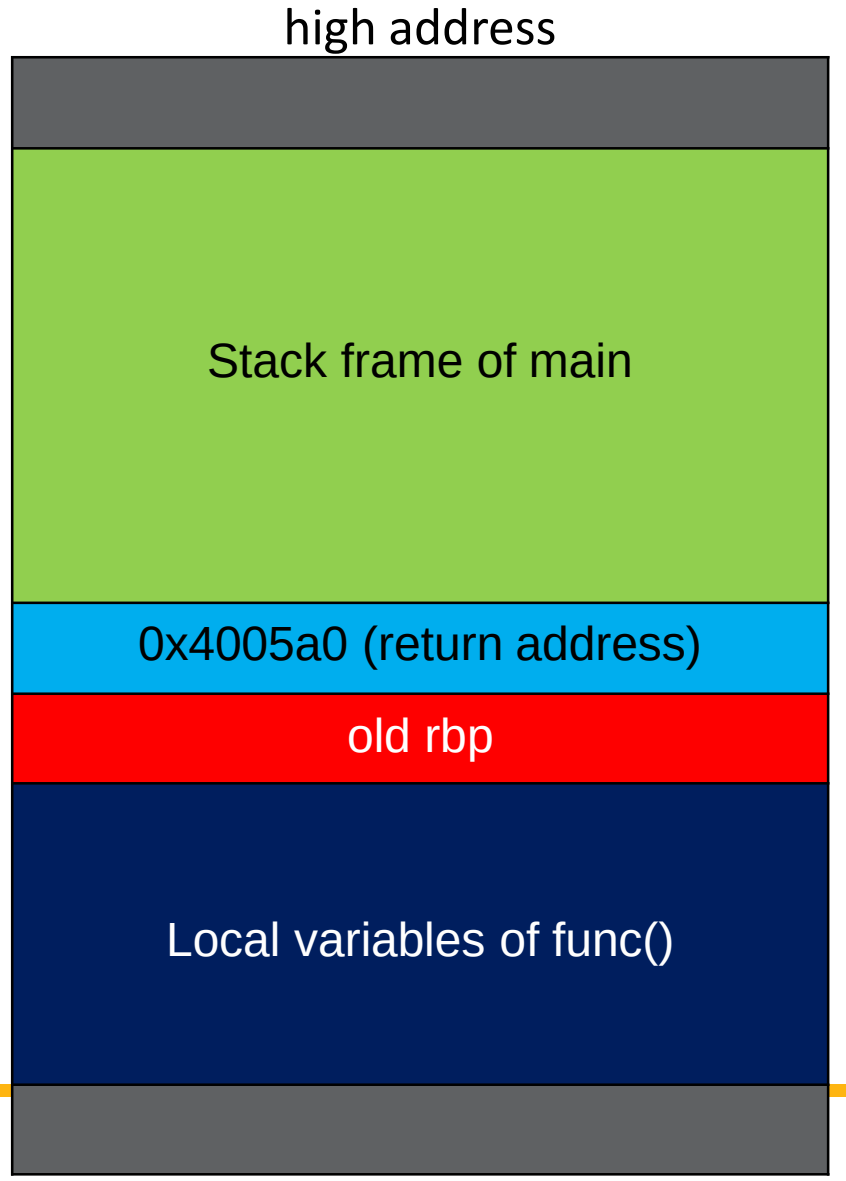
Example: Stack frame during a function call

```
func:
        push rbp                    leave = mov rsp, rbp
        mov rbp, rsp                pop rbp
        sub rsp, 0x30
        …
        move eax, 0x0
rip →   leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
```



Example: Stack frame during a function call

```
                                                    rbp →
func:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30
        …                          ret = pop rip
        move eax, 0x0
        leave                                       rsp →
rip →   ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```

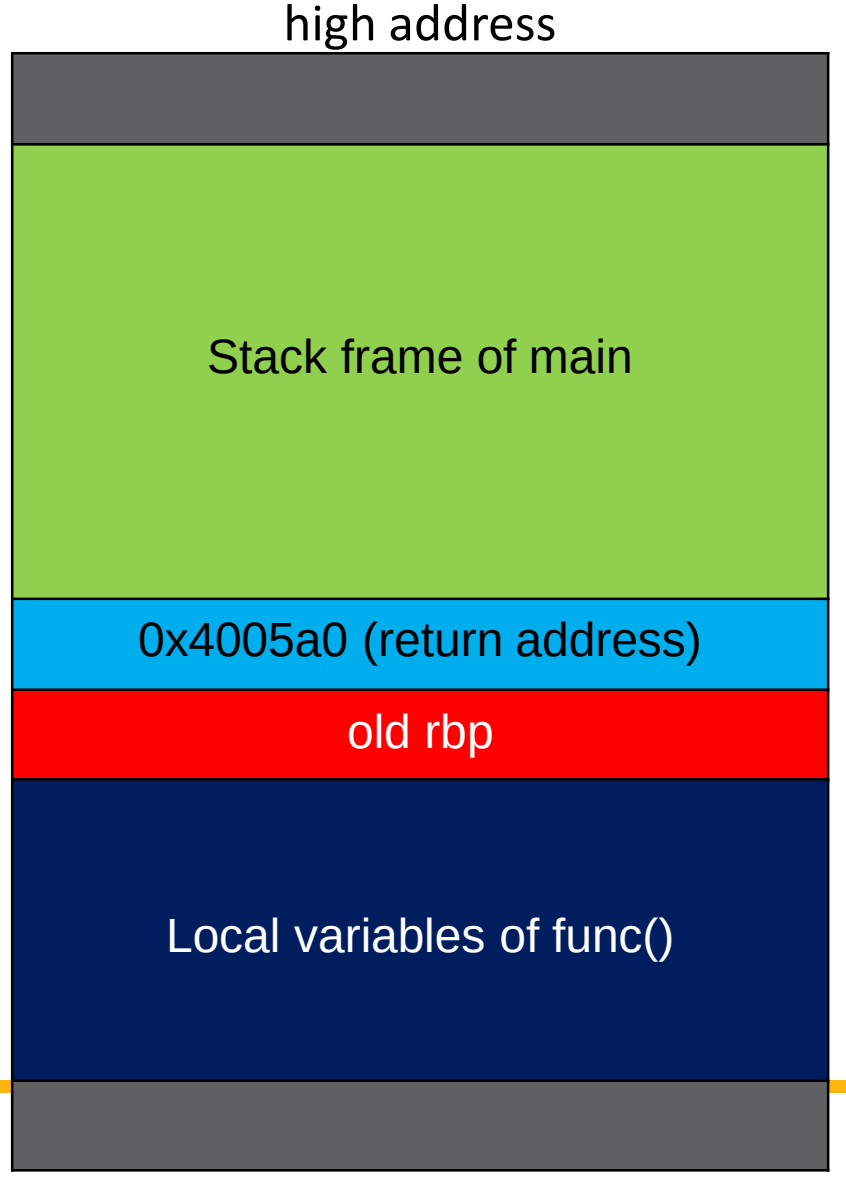


Example: Stack frame during a function call

```
func:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30
        …
        move eax, 0x0
        leave
        ret
main:
        …
        call func
rip →   mov eax, 0x0 // address 0x4005a0
        …
```

rbp →

rsp →



Project Submission

- Due date: 6/16 11:59 p.m.
- Makeup submission (75 points at most): TBA (After the final)
- Submission rules
  - Please create a folder for each task and put all the files/scripts needed to get the task's flag into the folder
  - Please put all the folders into a submission folder; zip the submission folder with your student ID and upload the zip file to New e3
  - Sample zip file: 309551234.zip
    - Fildes
      - exp.py
      - …
    - Time-will-stop
      - exp.py
      - …
    - ….

Demo Procedure

- Date: 6/17
- Location: Online with Zoom
- We will run your programs/scripts with you online and see whether flags can be obtained
  - ☐ All the flags will be changed
- Demo schedule and links will be released later

Questions?