Computer Security Capstone

Project 4: Capture The Flag (CTF)

Chi-Yu Li (2024 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 for all the tasks except Task I-4
  - Solving Task I-4 locally
- You are NOT allowed to team up: one student one team
  - Discussions are allowed between teams, but any collaboration is prohibited
- TA: Yen-Chiu Lee

How to Proceed?

- Connecting to each CTF server: nc <ip> <port>
  - ip: 140.113.24.241
  - 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
- You can use python with pwntools, too

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}
- You will need to submit the programs
  - run the programs when you demo

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 (80%)
- Task II: CTF beginners (20%)
- Download all given executable and source files from e3
  - CTF Server using ubuntu 22.04 (for some problem to calculate address)

Task I: Basic CTF Problems

- Task I-1: Flag Shop (20%)
- Task I-2: Magic (20%)
- Task I-3: Ret2libc (20%)
- Task I-4: Matryoshka Doll (20%)

Task I-1: Flag Shop

- Goal: Learn how operator and type conversion works in C/C++
- Server port: 30170

- Hints
  - cppreference

Task I-2: Magic

- Goal: learn about the glibc PRNG
- Server port: 30171

- Hints
  - Is the random function really random?

Task I-3: Ret2libc

- Goal: learn to identify basic logic flaw and buffer overflow in source codes
- Server port: 30173

- Hints
  - Inspect the code, where buffer overflow can occur?
  - Stack buffer overflow
  - return to libc example

Task I-4: Matryoshka Doll

- Goal: Learn how a file format is determined by OS

- Hints
  - Does that image have some additional bytes?
  - Magic number for file

Task II: CTF Beginner

- Task II-1: FMT (10%)
- Task II-2: Hello System (10%)

Task II-1: FMT

- Goal: learn to identify dangerous function usage
- Server port: 30172

- Hints
  - How do you use printf normally?
    - Which conversion specifier can modify variable?

Task II-2: Hello System

- Goal: learn how to leak canary and perform buffer overflow
- Server port: 30174
- Hints
  - Notice the difference between scanf and read
  - What are the features of canary?

Important: How to Prepare Your Program?

- Must provide a Makefile which compiles your source codes into six executable file
- You can use any language and library you want
  - Use your environment to demo
  - Do not hardcode the flag in your program
- Test requirements for your program
  - Do not need user interaction to get flag
    - For online tasks, you can only input server IP and port
    - For local tasks, you can only input file path
  - Must print flag to stdout

Important: How to Demo Your Program?

- Download your code from e3
- Download new file for task II-1
  - Given when demo, flag length are the same
- Run make if needed
- Run your executables
- Ask some questions about your code
- Binary file for task I-1, I-2, I-3, I-4, II-2 will not change
  - You can hardcode some symbol address if you need
  - FLAG environment will change to avoid hardcode the flag

Project Submission

- Due date: 6/12 11:55 p.m.
- Submission rules
  - Put all your files into a directory and name it using your student ID(s)
  - Zip the directory and upload the zip file to New e3
  - A sample of the zip file: 1234567.zip
    - 1234567
    - ├ Makefile (if needed)
    - ├ …
    - ├ …
    - └ …
  - If files are not in a directory after unzip, 10 points will be deducted.

Questions?

Useful Info

- command
  - checksec
  - readelf
- pwntools
  - connect to server and control what content will be sent to it
  - generate shellcode, attach gdb …etc.
- gdb
  - normal plugins: pwngdb / gef /peda
  - dynamic analysis of the program

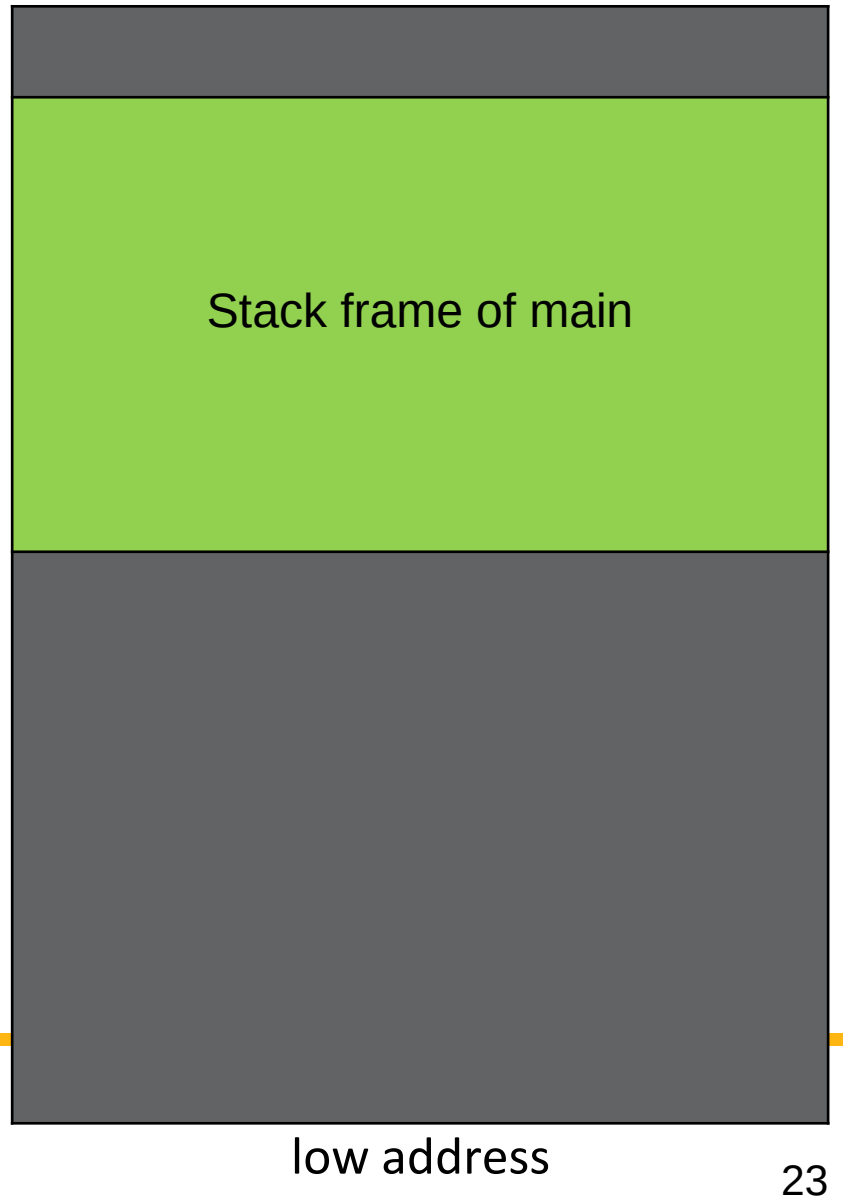
Example: Stack frame during a function call

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

rbp ➔

Call fun = push next_rip
jmp func

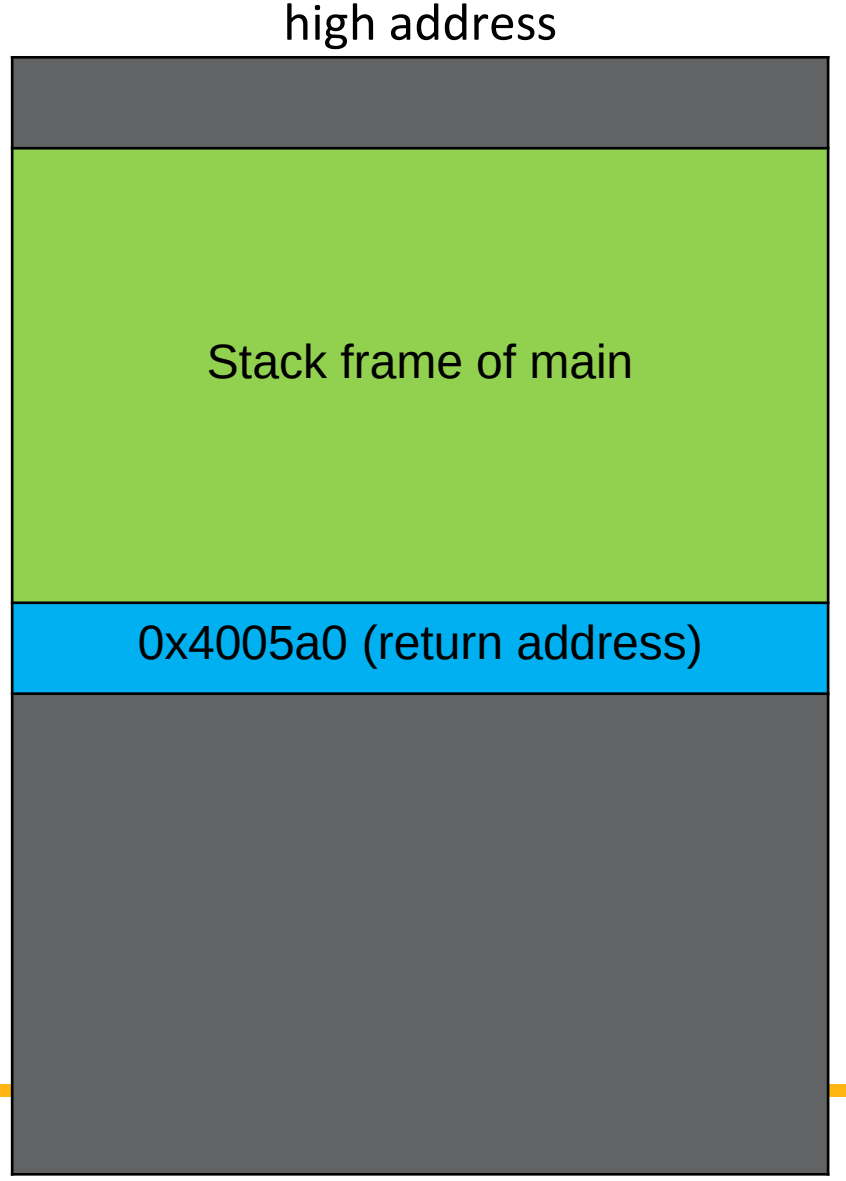
rsp ➔



Example: Stack frame during a function call

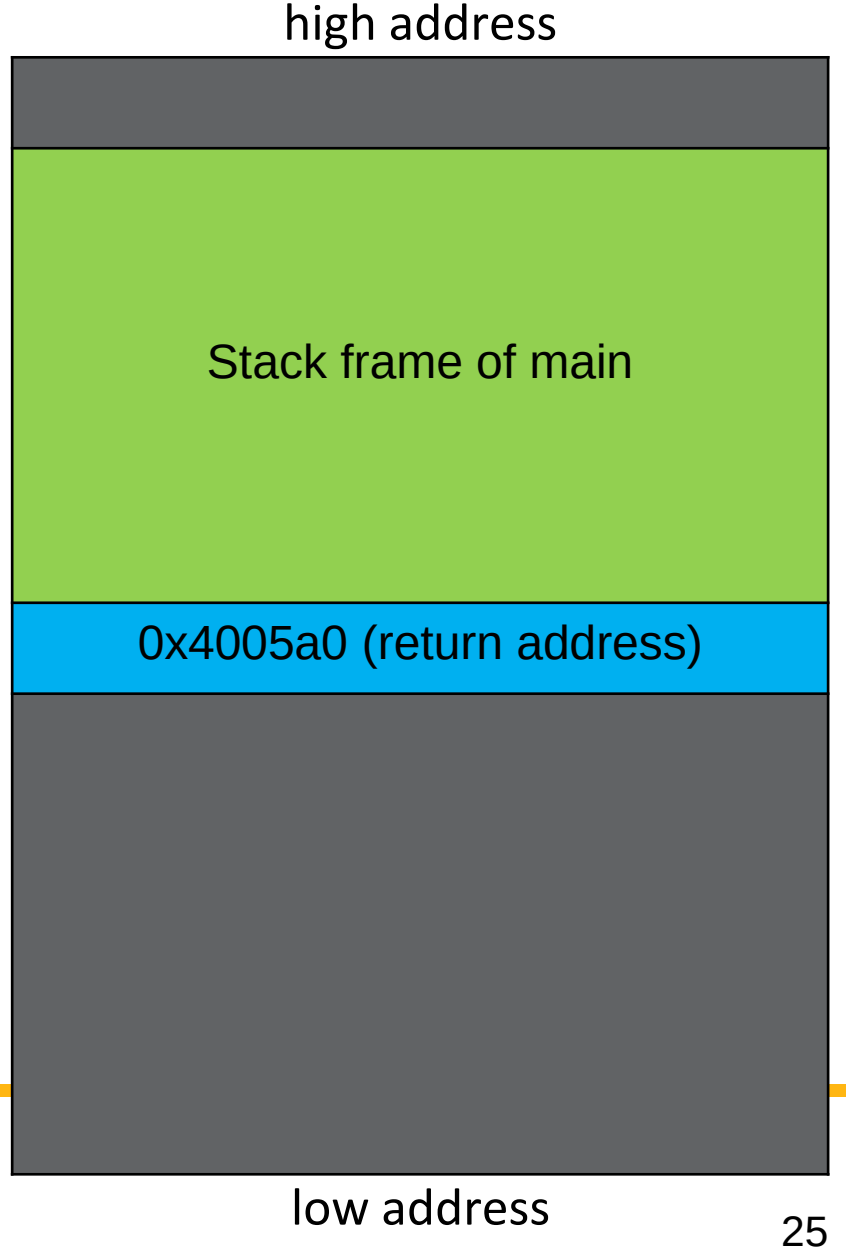
```
func:
        push rbp
        mov rbp, rsp
        sub rsp, 0x30
        …
        move eax, 0x0
        leave
        ret
main:
        …
rip ➔   call func
        mov eax, 0x0 // address 0x4005a0
        …
```

Call fun = push next_rip
jmp func



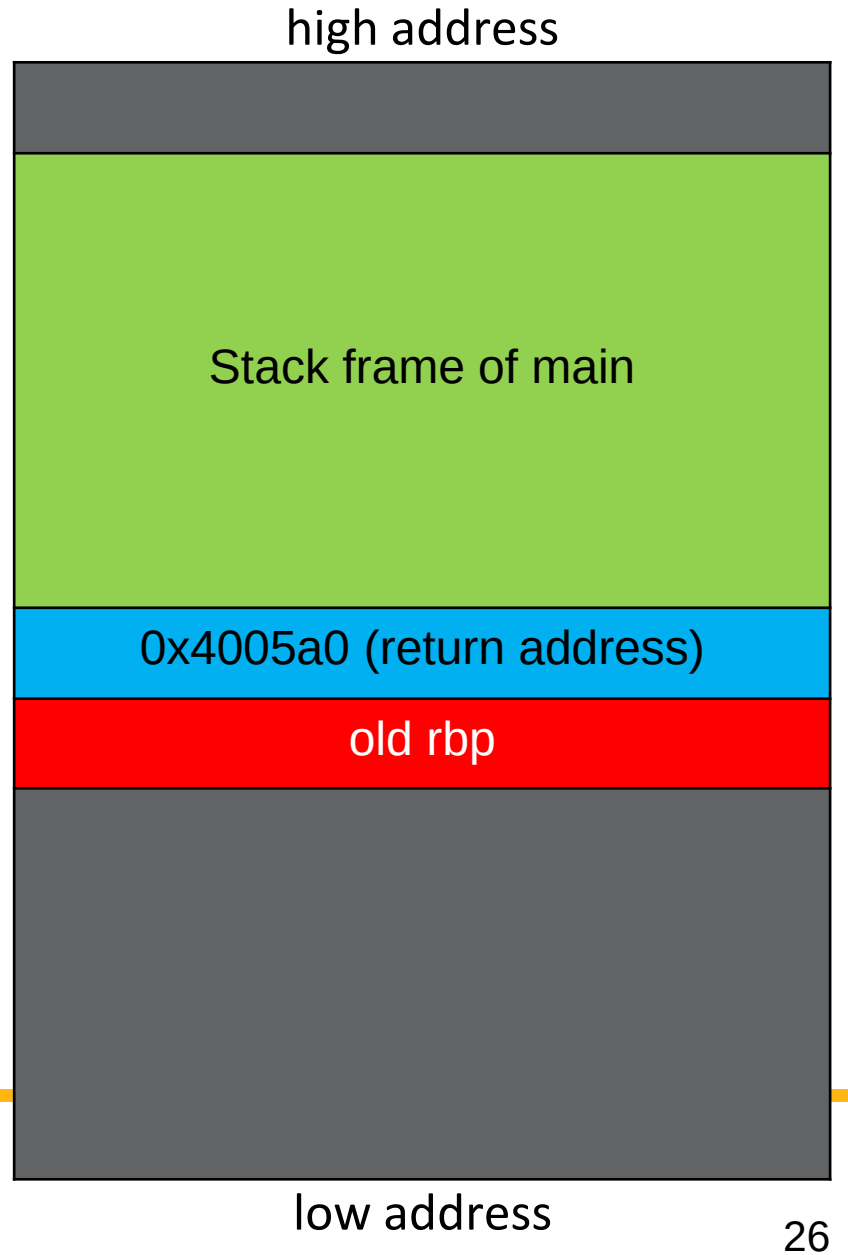
Example: Stack frame during a function call

```
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

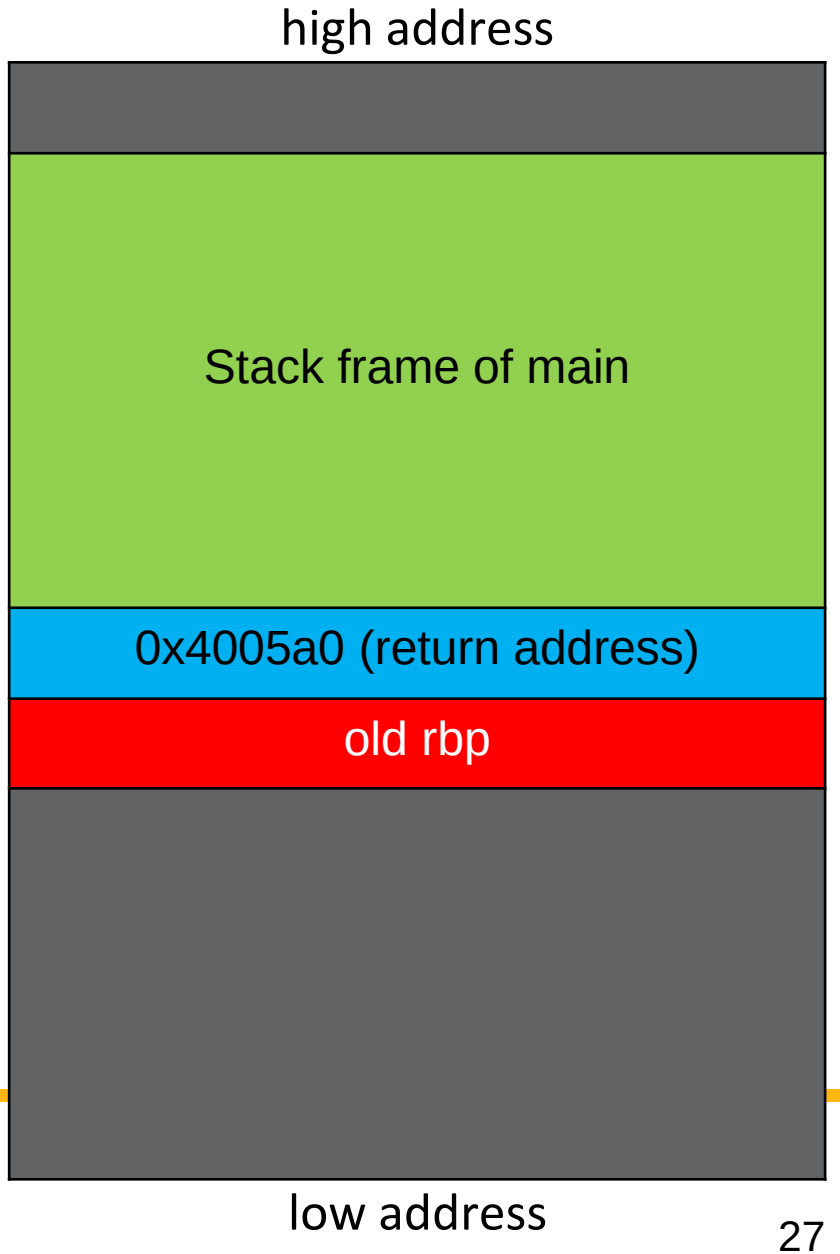
```
func:
        push rbp
rip ➔   mov rbp, rsp
        sub rsp, 0x30
        …
        move eax, 0x0
        leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```



Example: Stack frame during a function call

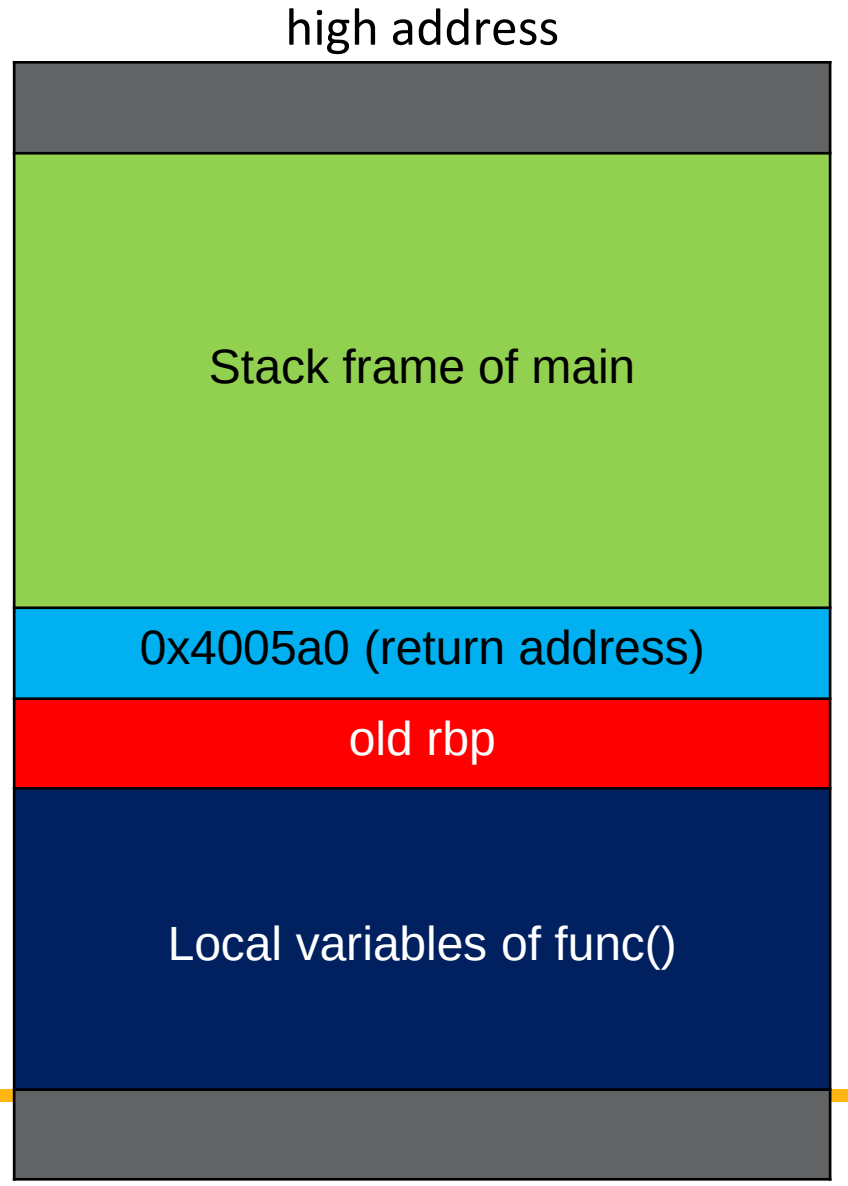
```
func:
        push rbp
        mov rbp, rsp
rip ➔   sub rsp, 0x30
        …
        move eax, 0x0
        leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```

rbp ➔ rsp ➔



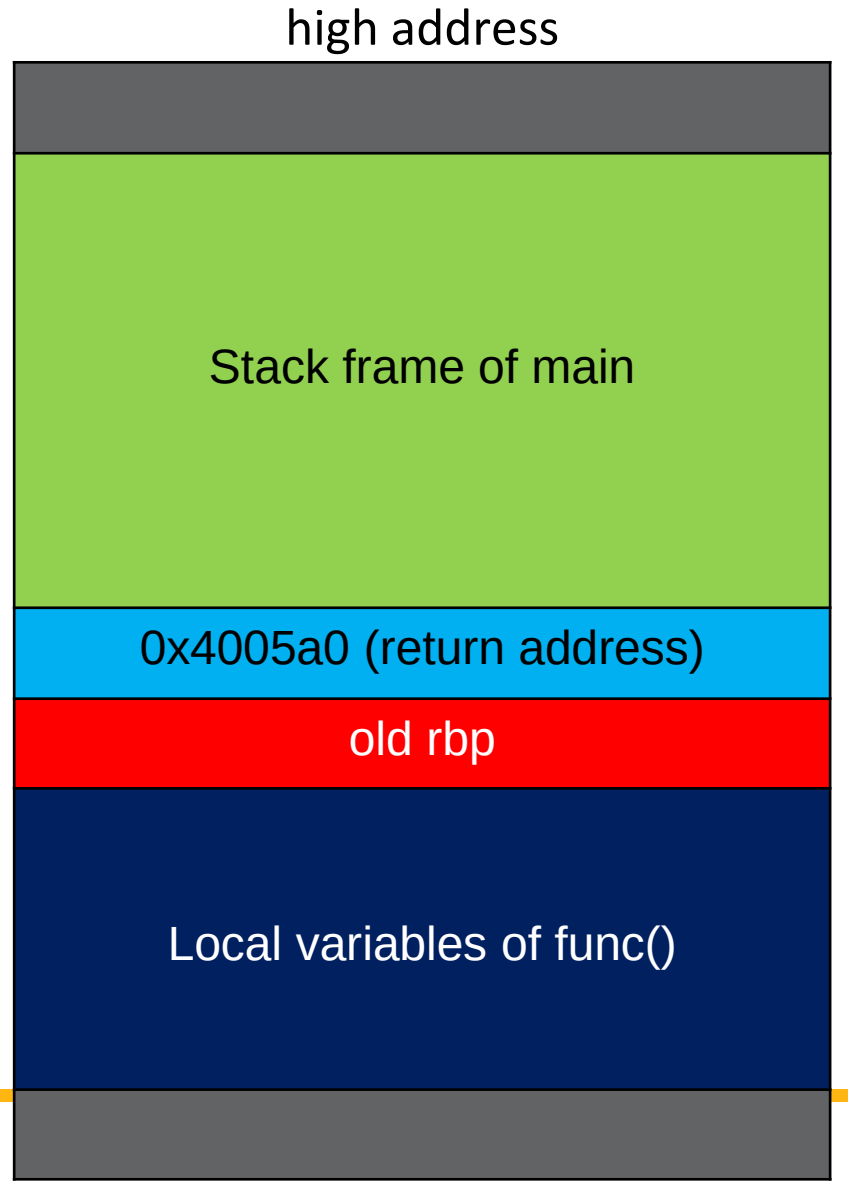
Example: Stack frame during a function call

```
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

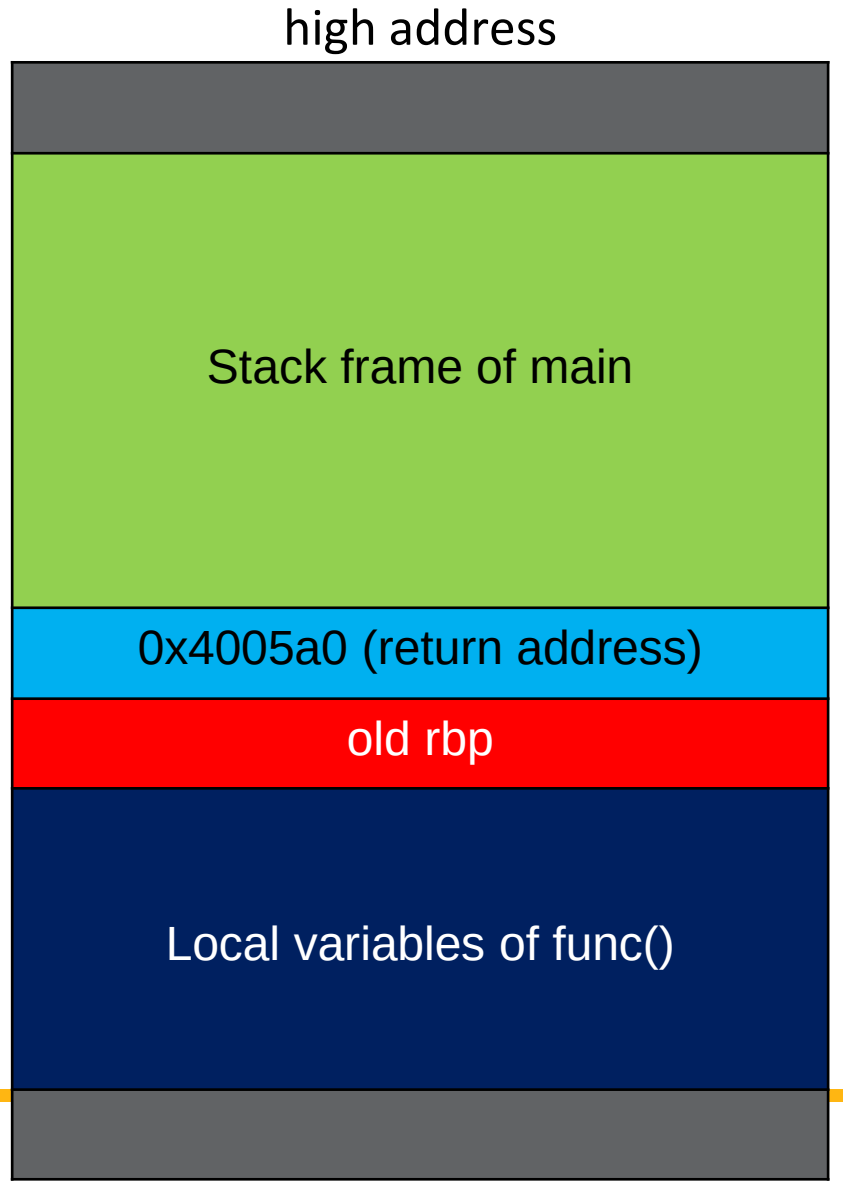
```
func:
        push rbp                    leave = mov rsp, rbp
        mov rbp, rsp                        pop rbp
        sub rsp, 0x30
        …
        move eax, 0x0
rip ➔   leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```



Example: Stack frame during a function call

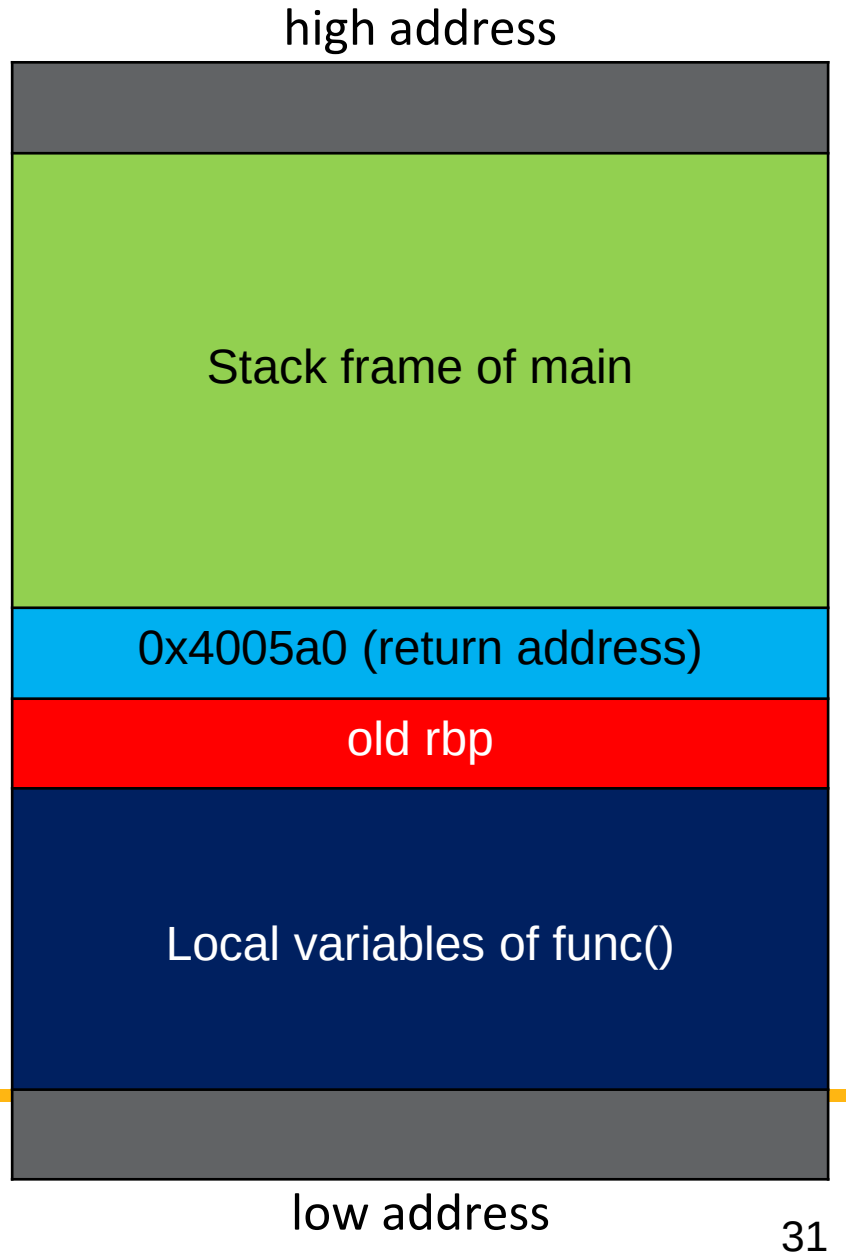
```
func:
        push rbp                    leave = mov rsp, rbp
        mov rbp, rsp                        pop rbp
        sub rsp, 0x30
        …
        move eax, 0x0
rip ➔   leave
        ret
main:
        …
        call func
        mov eax, 0x0 // address 0x4005a0
        …
```

rbp ➔ rsp ➔



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 ➔