Intro to Binary Exploitation/PWN

21 June 2025

by capang

$whoami

- Final Year Computer System and Networking Student in University of Malaya
- PwC Malaysia Cyber Threat Operations, Intern
- CTF player
  - Malaysia Representative for ACS Hacking Contest Vietnam 2024
  - 3rd Place (Malaysia) PwC Hack-A-Day 2024
- Malaysia Cybersecurity Camp 2024 Alumni
- PWN Challenge Creator
  - Girls in CTF 2024
  - BlackBerry CCoE CTF 2025
  - UM Cybersecurity Summit CTF 2025
- Passionate about computers and networking

Blog: https://brocapang.github.io/

LinkedIn: https://www.linkedin.com/in/gnapac/

Disclaimer

This session is intended for educational purposes only.

Please be aware that:

- All activities demonstrated during this session are conducted in a controlled environment with explicit permission.
- Unauthorized access to computer systems is illegal and unethical.
- Ensure that your actions comply with all applicable laws and regulations.
- The information shared should be used to improve your own security posture and not for malicious purposes.
- By participating in this session, you agree to abide by these principles and use the knowledge gained responsibly.

PS: I am a student learning about this also so… feel free to correct any mistakes I make!

What is PWN?

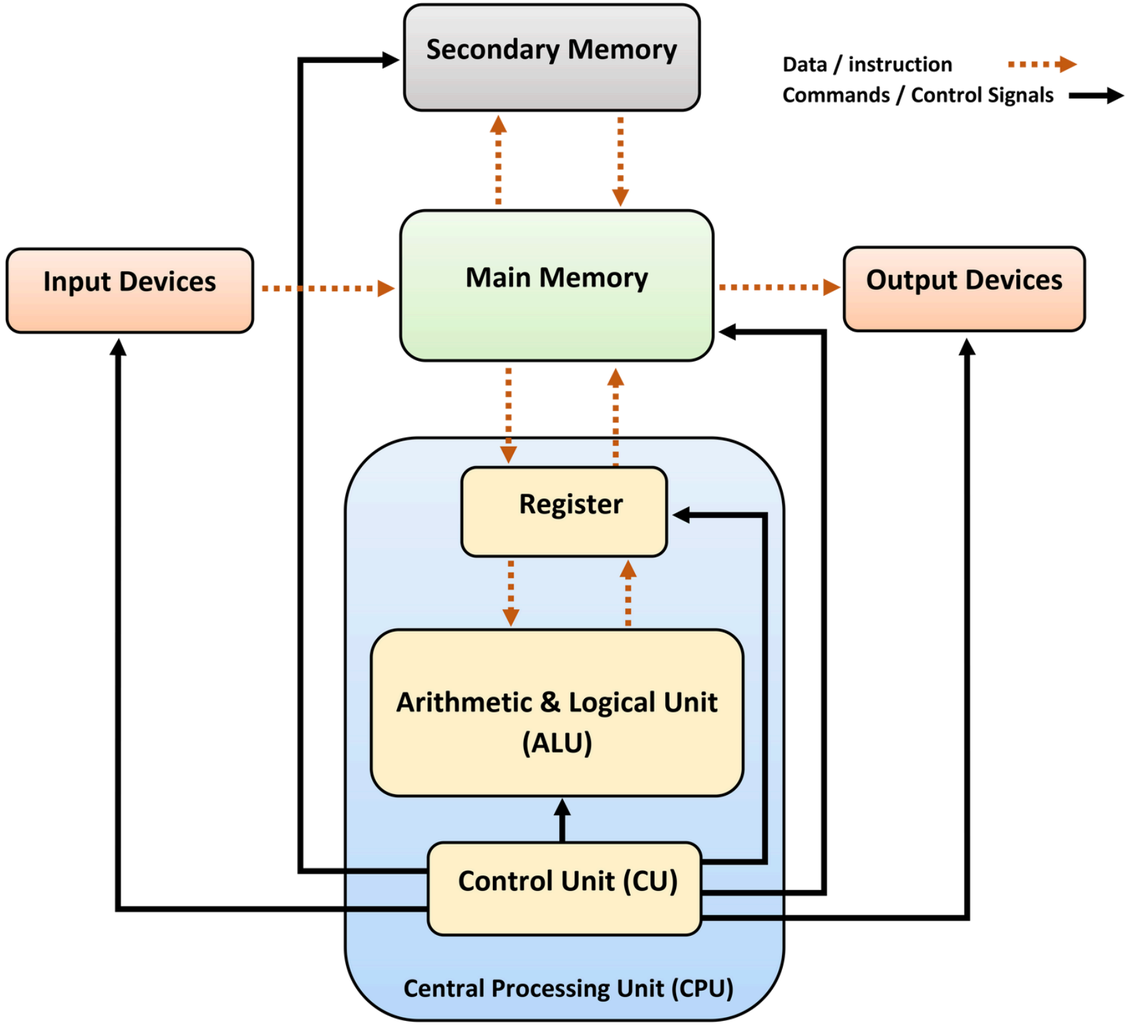
What is PWN?

- PWN = “own”
- Finding a vulnerability in a compiled program
- Using that vulnerability to control the program’s behavior

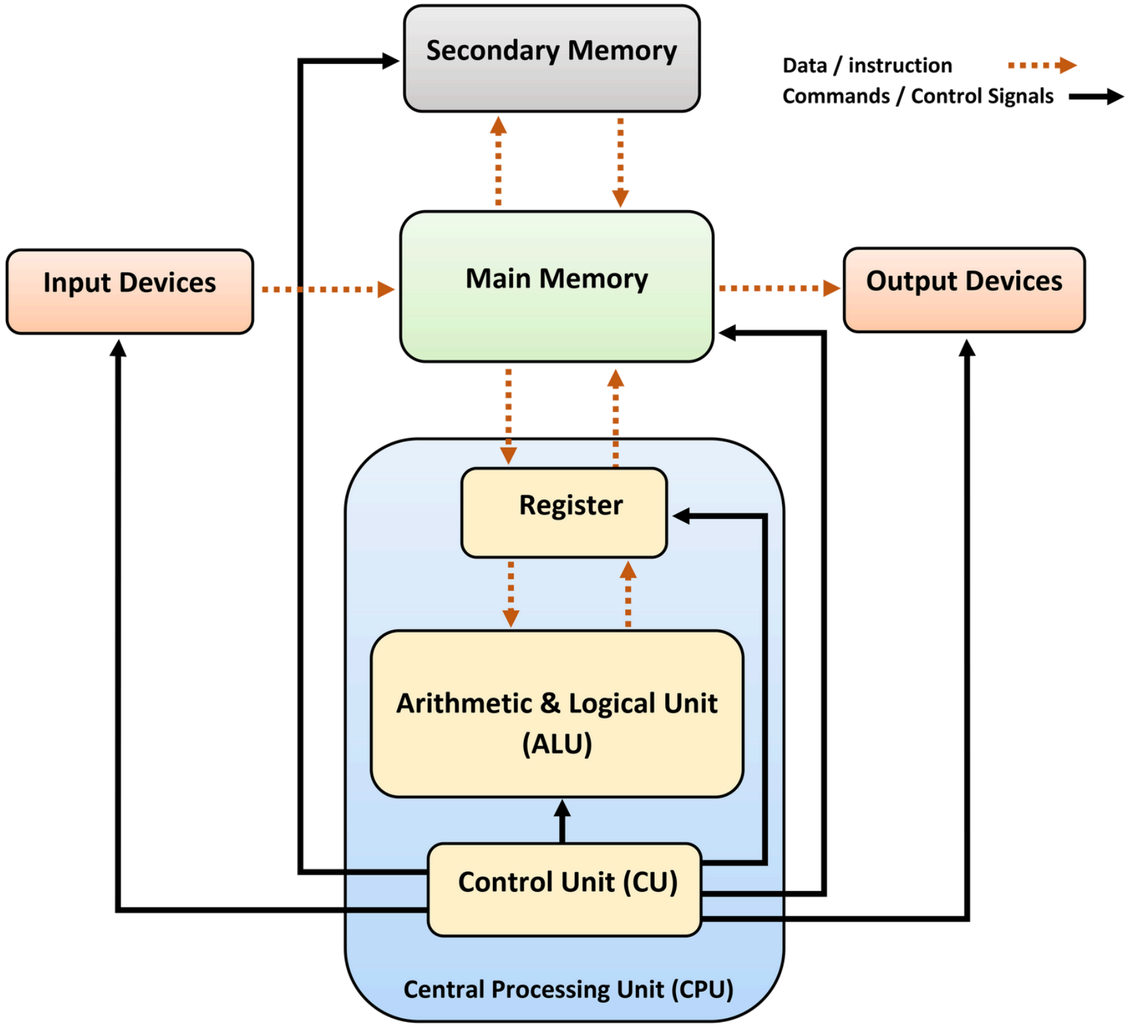
reading code → to crashing a program → run our own commands

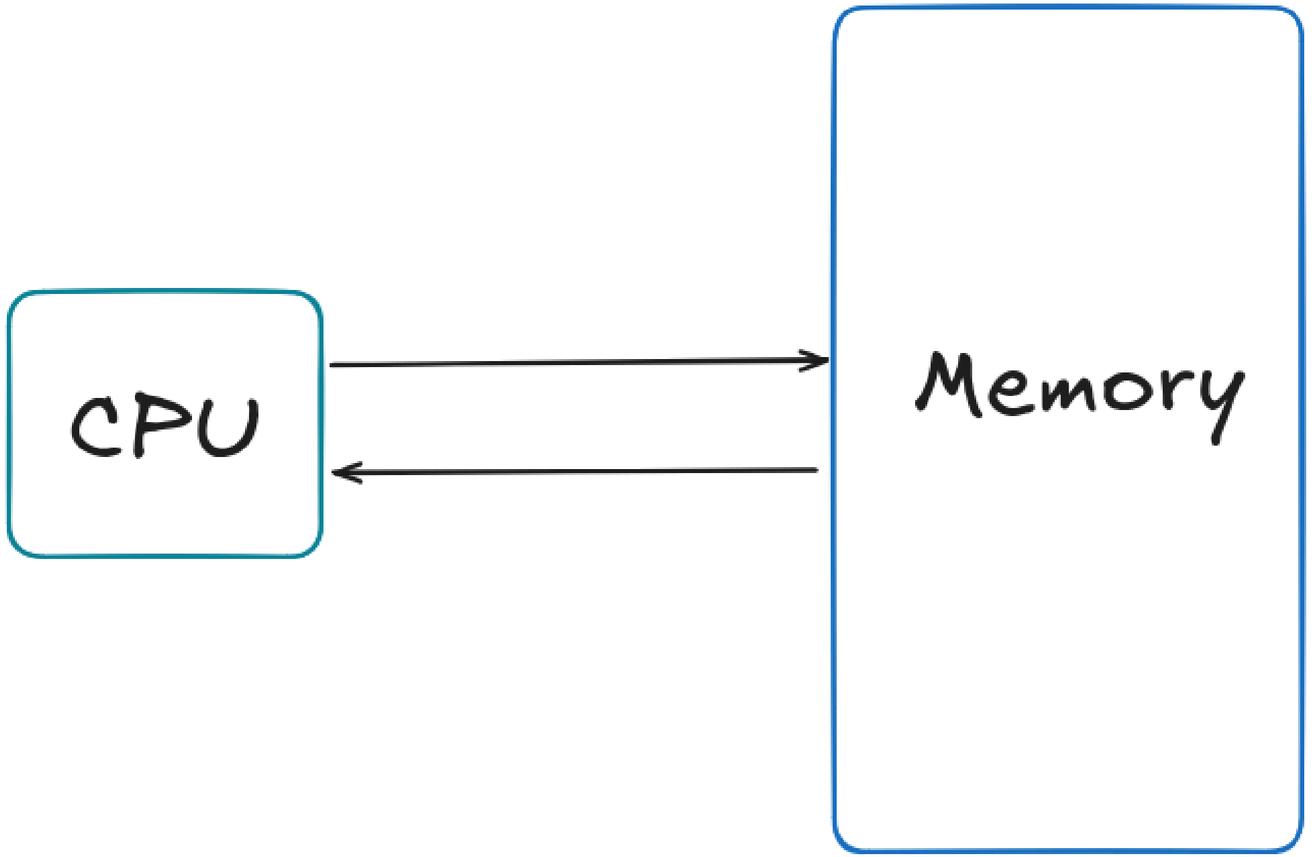
Experience doing PWN

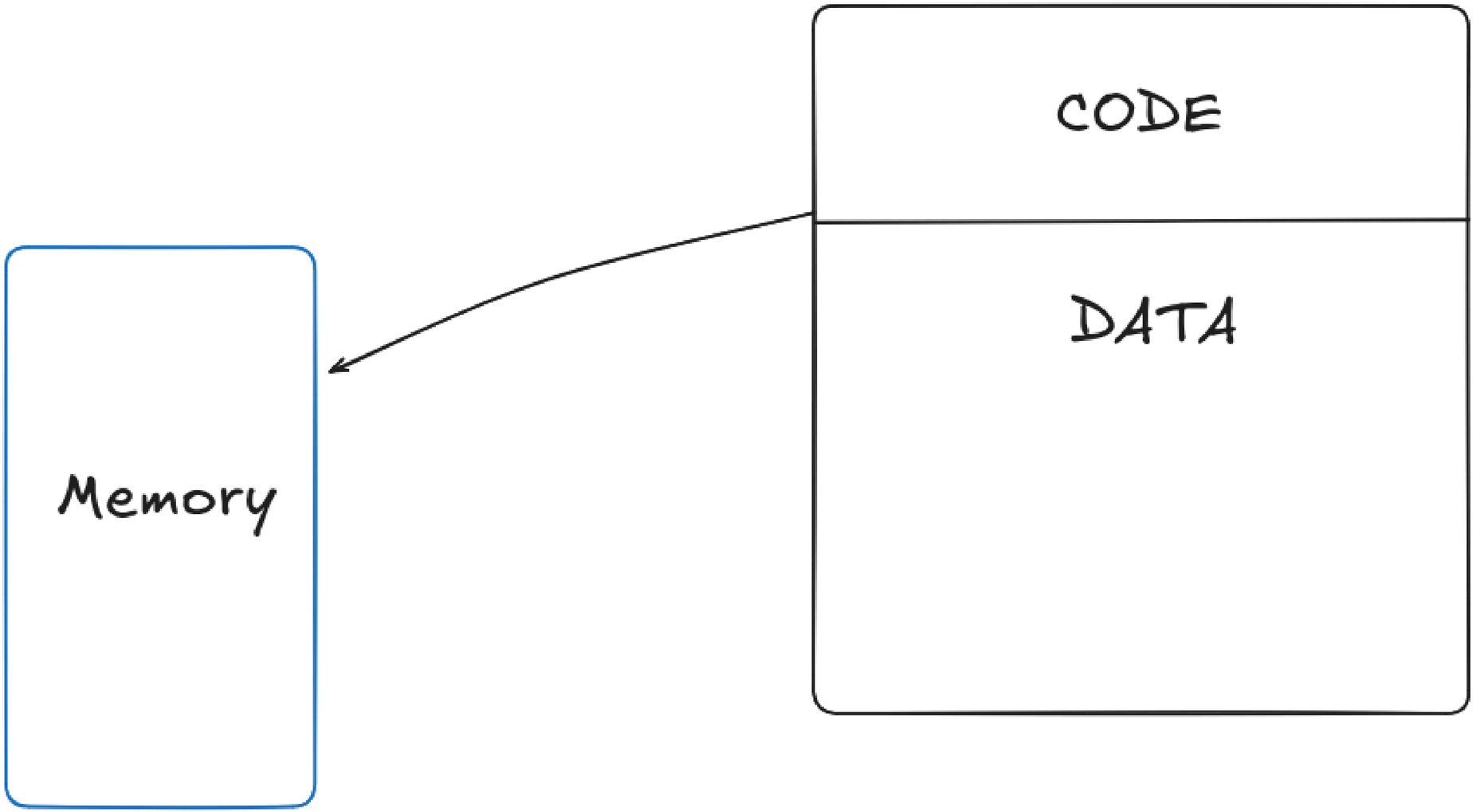
Overview of Computer Architecture for Binary Exploitation

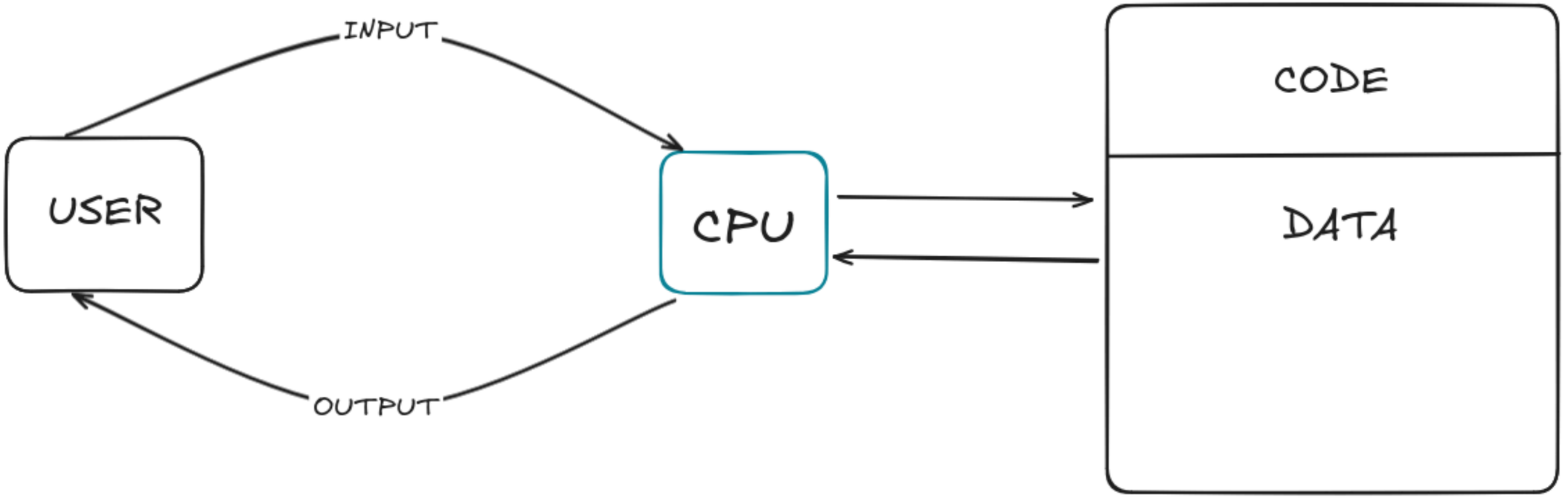

Secondary Memory
Data / instruction
Commands / Control Signals
Input Devices
Main Memory
Output Devices
Register
Arithmetic & Logical Unit
(ALU)
Control Unit (CU)
Central Processing Unit (CPU)

Von Neumann

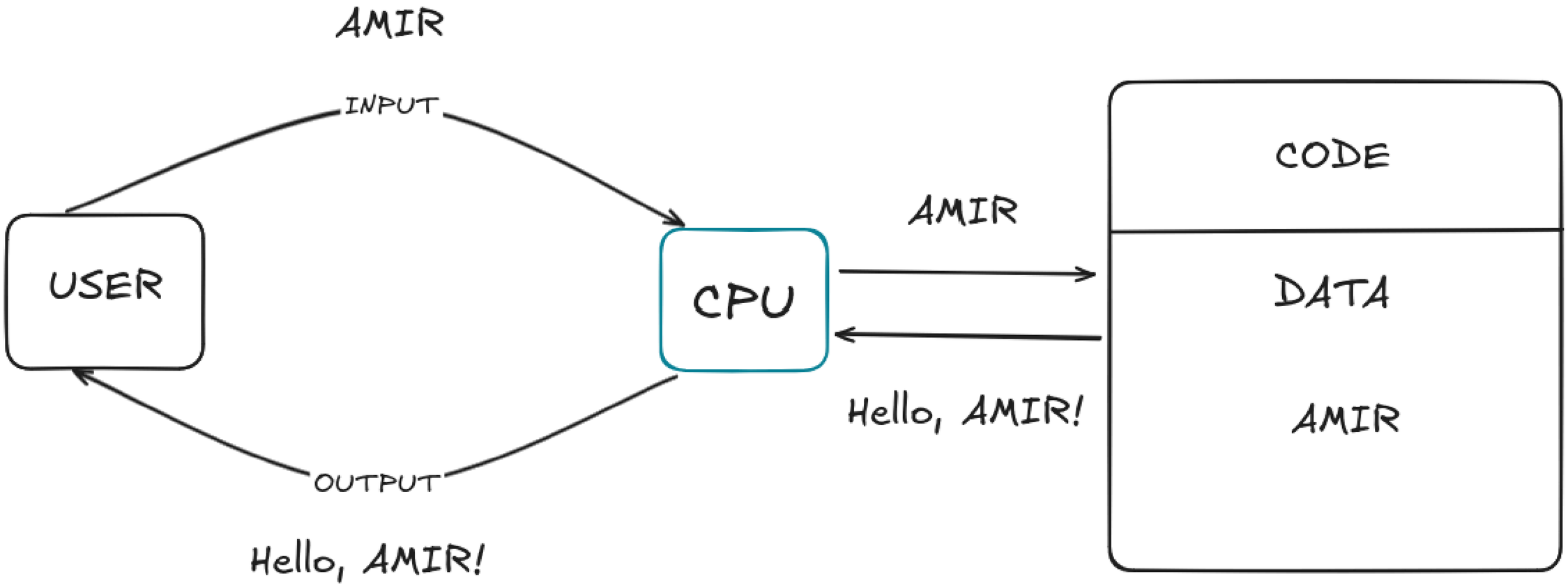

Secondary Memory
Data / instruction
Commands / Control Signals
Input Devices
Main Memory
Output Devices
Register
Arithmetic & Logical Unit
(ALU)
Control Unit (CU)
Central Processing Unit (CPU)


CPU
Memory


CODE
DATA
Memory

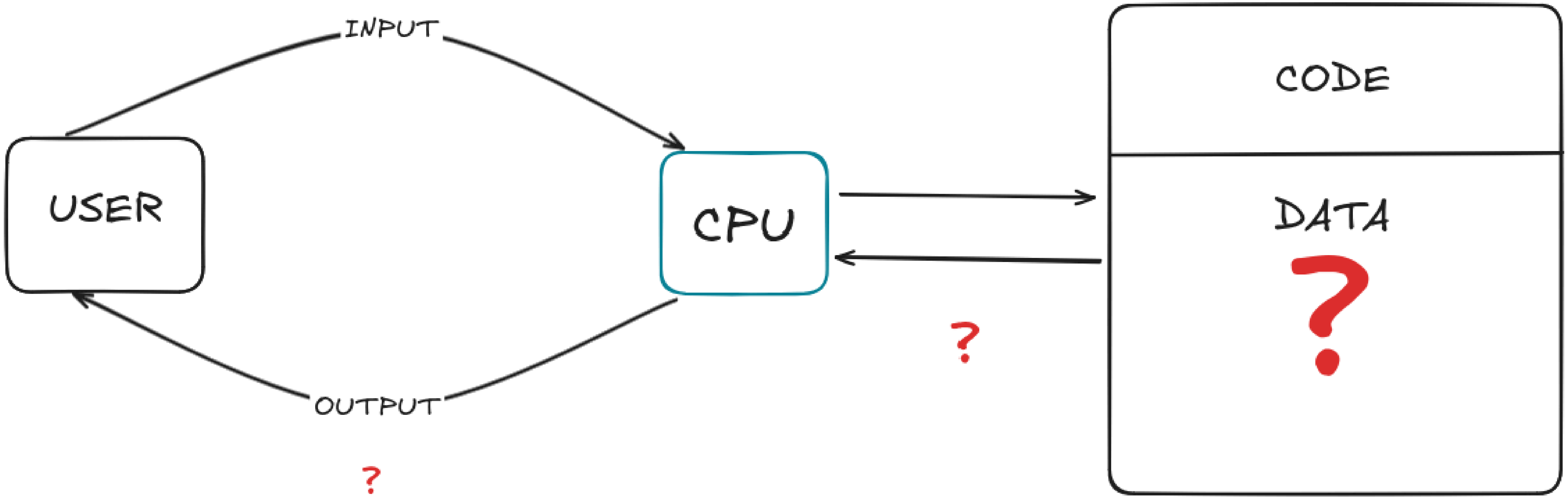

INPUT
USER
CPU
CODE
DATA
OUTPUT

```c
#include <stdio.h>

int main() {
    char name[100];

    printf("Enter your name: ");
    // Read name
    gets(name, sizeof(name), stdin);

    printf("Hello, %s!\n", name);

    return 0;
}

```




```c
#include <stdio.h>

int main() {
    char name[100];

    printf("Enter your name: ");
    // Read name
    gets(name, sizeof(name), stdin);

    printf("Hello, %s!\n", name);

    return 0;
}
```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA




```c
#include <stdio.h>

int main() {
    char name[100];

    printf("Enter your name: ");
    // Read name
    gets(name, sizeof(name), stdin);

    printf("Hello, %s!\n", name);

    return 0;
}
```

The Goal of this Session

- Basics on Low Level Computer Knowledge
- Basics on PWN
  - Understand Buffer Overflow
  - Understand Return Oriented Programming

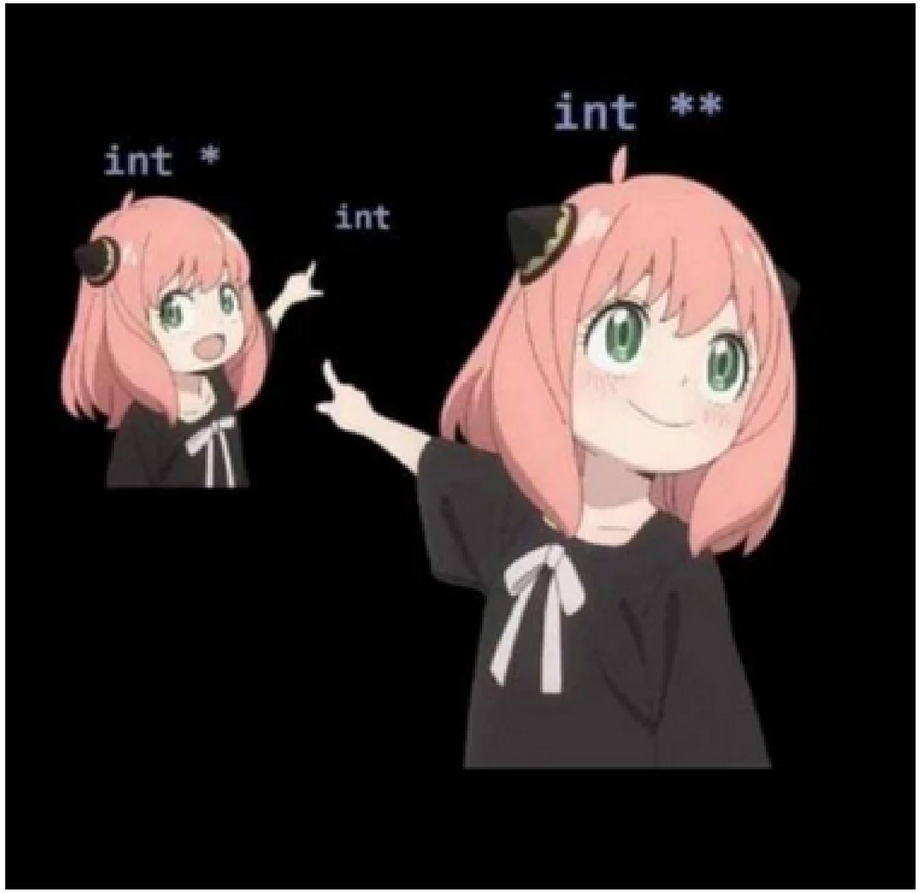
Pointers

What is Pointers?

- A pointer stores a memory address
- It "points to" a location in memory
- Not the data, just the address

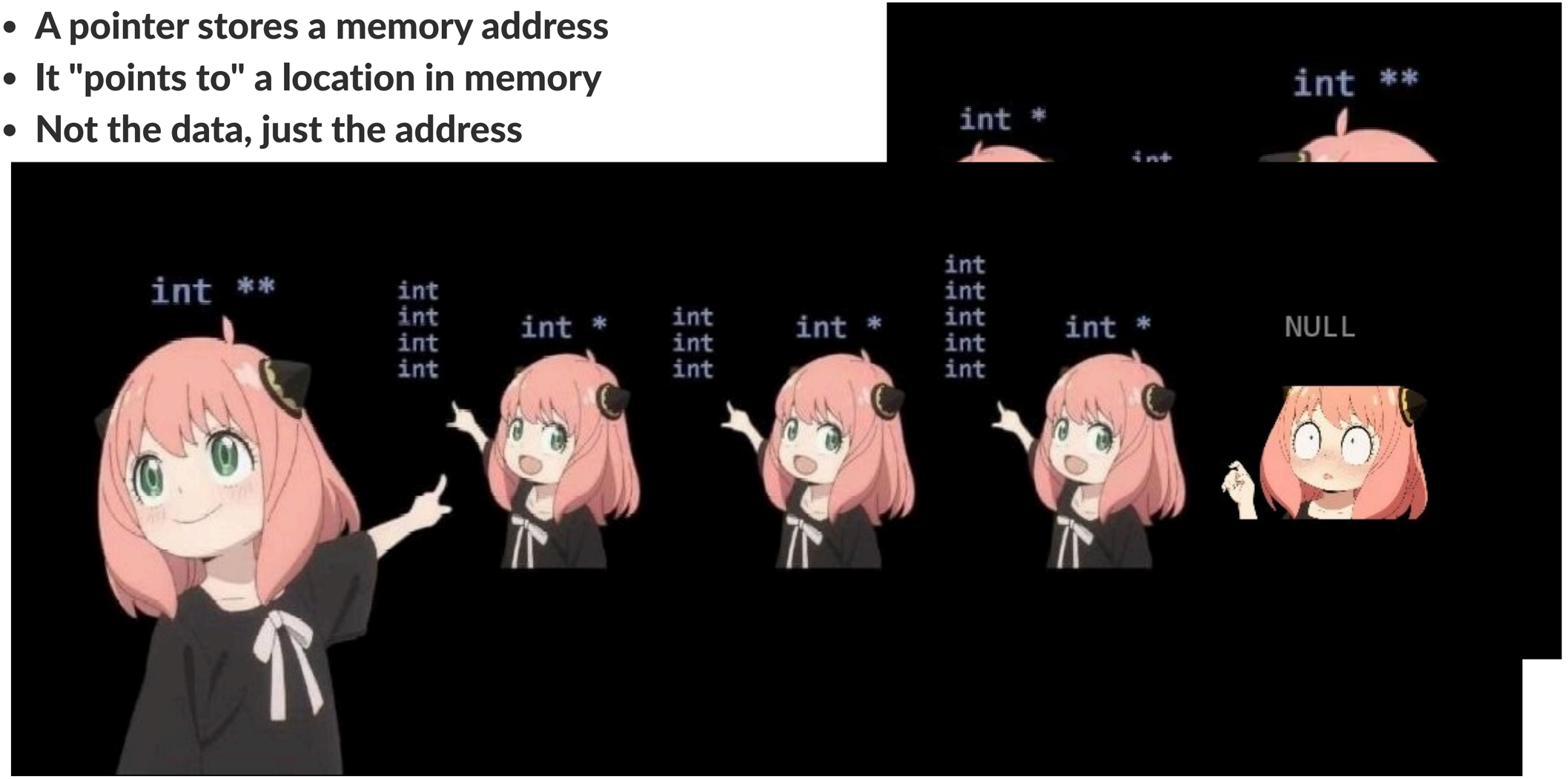
What is Pointers?

- A pointer stores a memory address
- It "points to" a location in memory
- Not the data, just the address



What is Pointers?

- A pointer stores a memory address
- It "points to" a location in memory
- Not the data, just the address



What is Pointers?

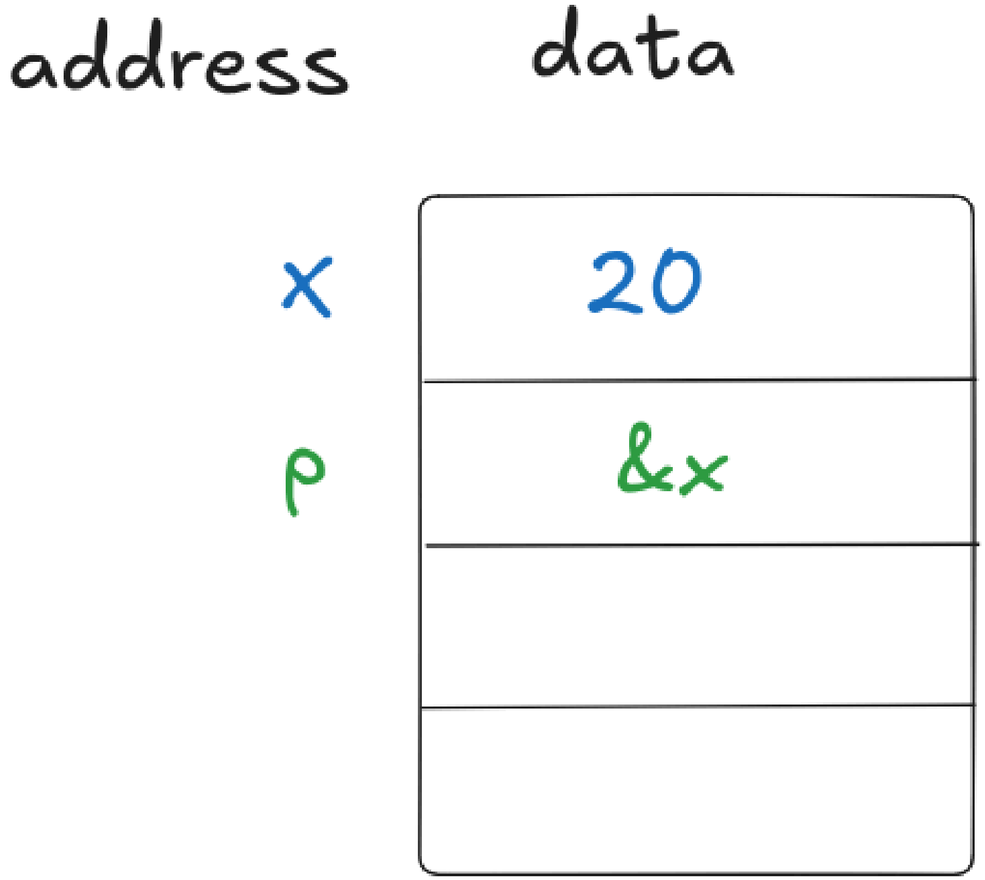
```
int x = 20;

int *p = &x;
```

What is Pointers?

```
int x = 20;

int *p = &x;
```

What is Pointers?

```
int x = 20;

int *p = &x;
```



What is Pointers?

```c
#include <stdio.h>

int main() {

        int x = 20;

        int *p = &x;

        printf("x: %d\n",x);
        printf("&x: %p\n",&x);
        printf("p: %p\n",p);


        return;
}
```

What is Pointers?

```c
#include <stdio.h>

int main() {

        int x = 20;

        int *p = &x;

        printf("x: %d\n",x);
        printf("&x: %p\n",&x);
        printf("p: %p\n",p);


        return;
}
```

```
x: 20
&x: 0x7fffbd64a0bc
p: 0x7fffbd64a0bc
```

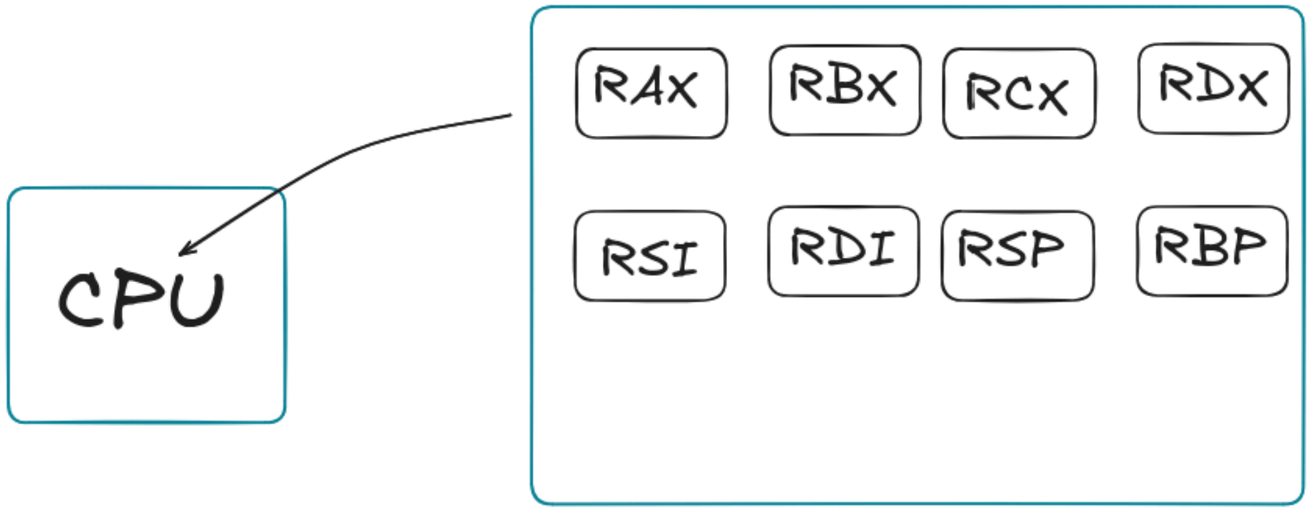
Registers and the CPU

What are Registers?

- Ultra-fast storage built inside the CPU
- Used for temporary data, addresses, control
- Directly manipulated during program execution

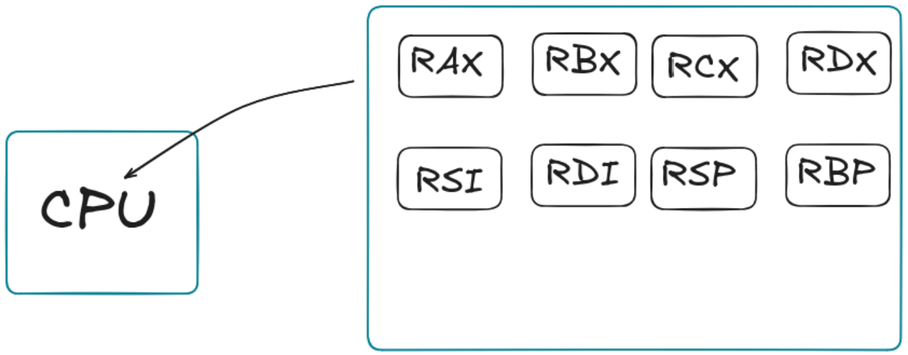
What are Registers?

- Ultra-fast storage built inside the CPU
- Used for temporary data, addresses, control
- Directly manipulated during program execution

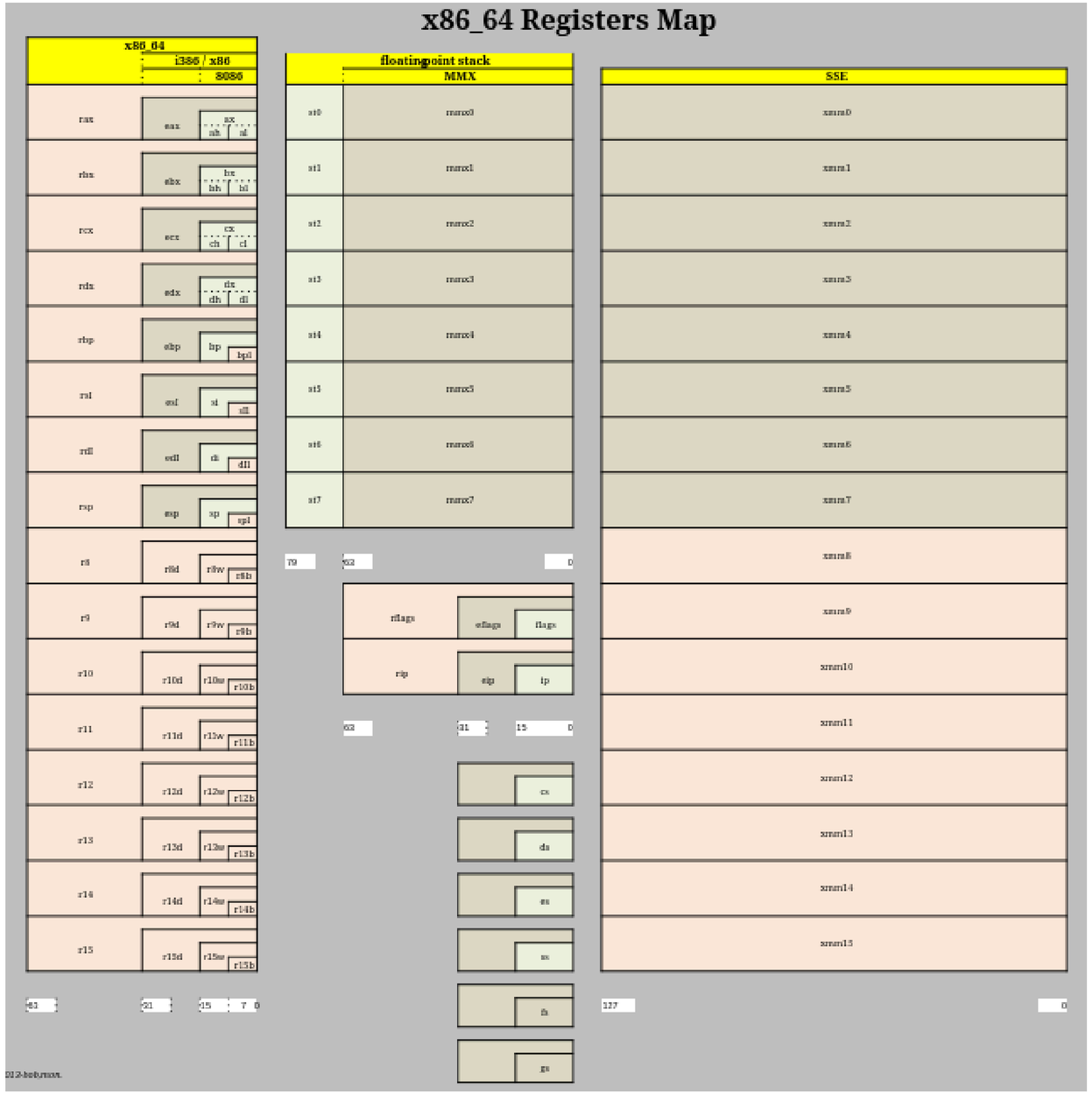


What are Registers?

- Ultra-fast storage built inside the CPU
- Used for temporary data, addresses, control
- Directly manipulated during program execution



q (8 bytes)	l (4 bytes)	w (2 bytes)	b (1 byte)	
%rax	%eax		%ax	accumulate
%rbx	%ebx		%bx	base
%rcx	%ecx		%cx	counter
%rdx	%edx		%dx	data
%rsi	%esi		%si	source index
%rdi	%edi		%di	destination index
%rsp	%esp		%sp	stack pointer
%rbp	%ebp		%bp	base pointer



source: https://upload.wikimedia.org/wikipedia/commons/3/3e/X86_64-registers.svg

Key Registers to take note

- **RSP and RBP: Manages the STACK**
- **RIP: Keeps track of functions called/returned**
- **RDI, RSI, RDX: Function parameters**

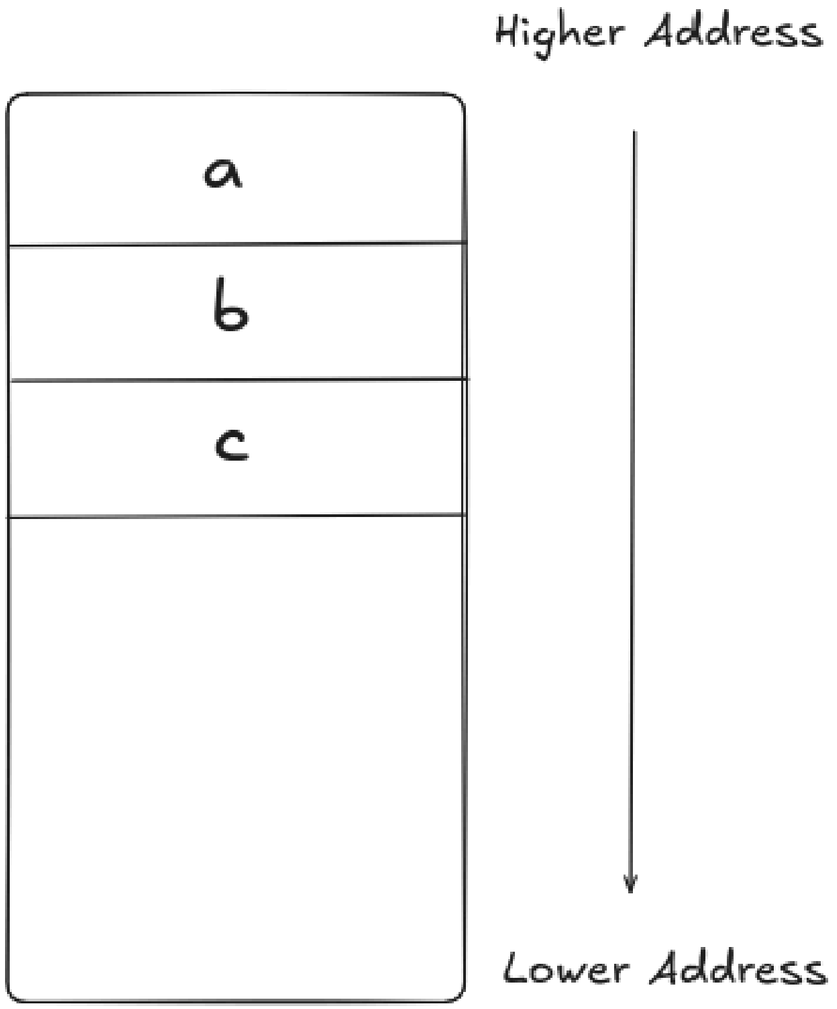
The STACK and Function Calling

What is the Stack?

- One of the “region” of memory for function execution
- Used to store:
  - Function arguments
  - Return addresses
  - Local variables
- Grows downward in memory (from high to low addresses)

What is the Stack?

- One of the “region” of memory for function execution
- Used to store:
  - Function arguments
  - Return addresses
  - Local variables
- Grows downward in memory (from high to low addresses)

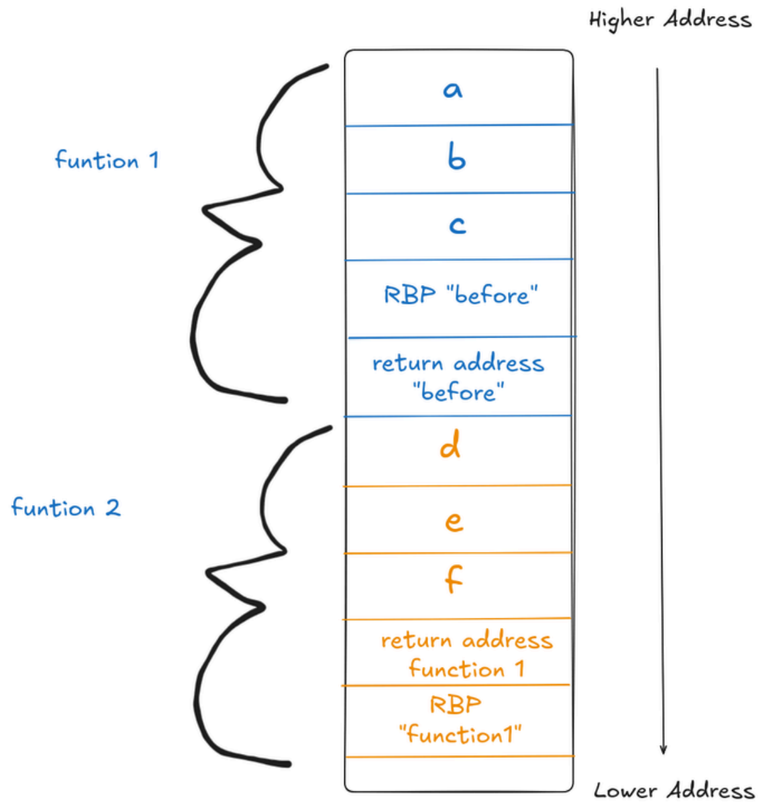


What is the Stack?

```
1   #include <stdio.h>
 1  void function2(){
 2          int d = 400;
 3          int e = 500;
 4          int f = 600;
 5
 6
 7          return;
 8  }
 9
10
11
12  void function1(){
13          int a = 10;
14          int b = 20;
15          int c = 30;
16
17          function2();
18          return;
19
20  }
```

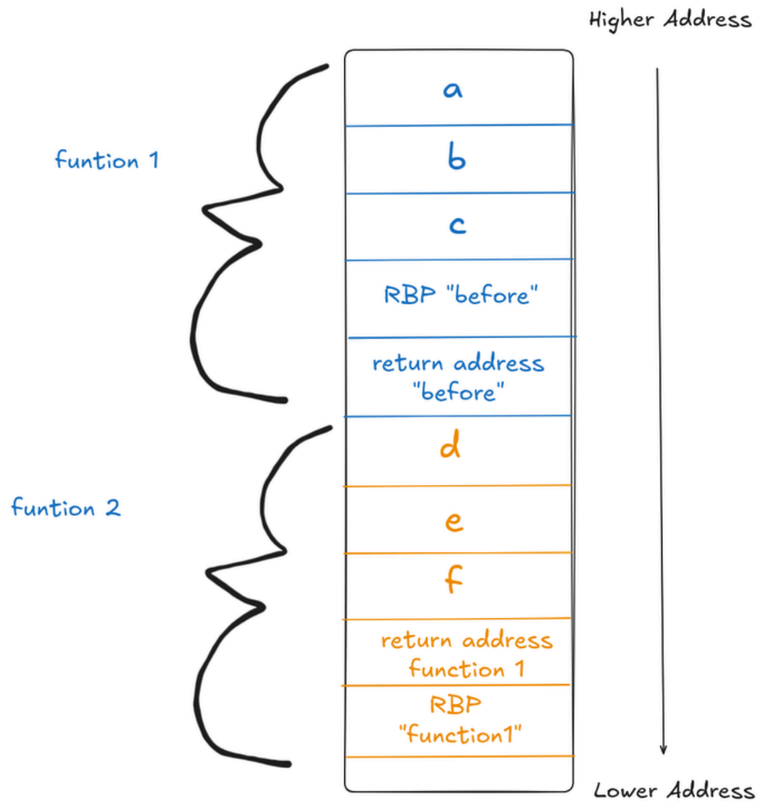
What is the Stack?

```c
#include <stdio.h>
void function2(){
        int d = 400;
        int e = 500;
        int f = 600;


        return;
}



void function1(){
        int a = 10;
        int b = 20;
        int c = 30;

        function2();
        return;

}
```



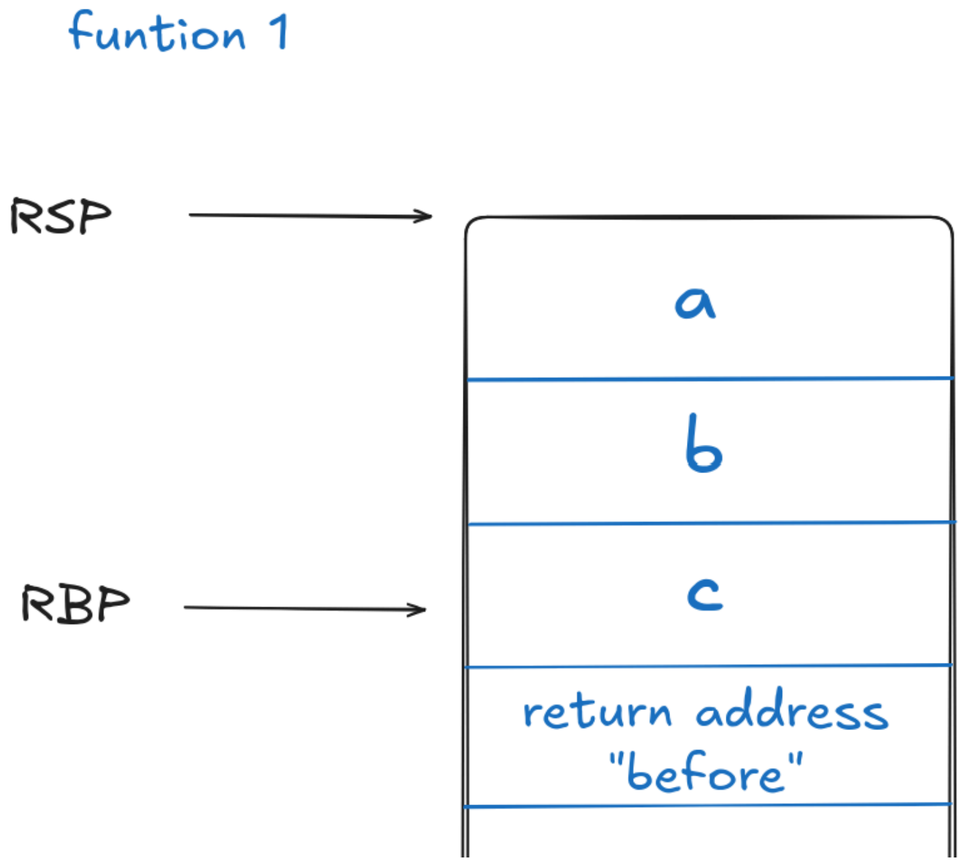
How does the program tracks the size?

```c
#include <stdio.h>
void function2(){
        int d = 400;
        int e = 500;
        int f = 600;


        return;
}



void function1(){
        int a = 10;
        int b = 20;
        int c = 30;

        function2();
        return;

}
```



How does the program tracks the size?

```
12 void function1(){
13         int a = 10;
14         int b = 20;
15         int c = 30;
16
17         function2();
18         return;
19
20 }
```



Function Calling

Function Calling

```
16 void function1(){
15
14          int a = 10;
13          int b = 20;
12
11          function2();
10
 9          return;
 8 }
 7
```

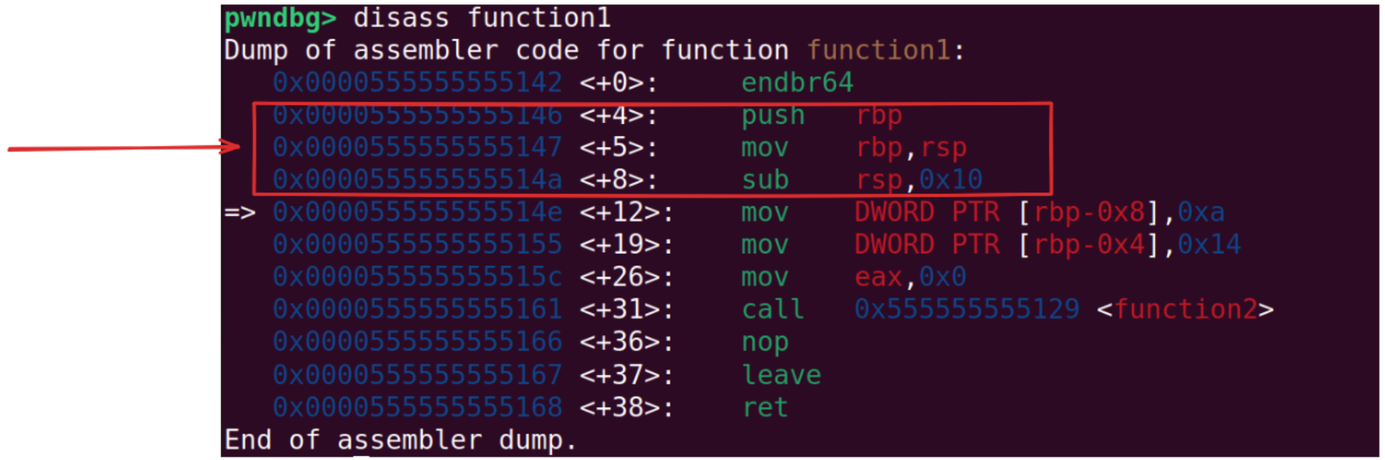
Function Calling

```
16 void function1(){
15
14          int a = 10;
13          int b = 20;
12
11          function2();
10
 9          return;
 8 }
 7
```

```
pwndbg> disass function1
Dump of assembler code for function function1:
   0x0000555555555142 <+0>:     endbr64
   0x0000555555555146 <+4>:     push   rbp
   0x0000555555555147 <+5>:     mov    rbp,rsp
   0x000055555555514a <+8>:     sub    rsp,0x10
=> 0x000055555555514e <+12>:    mov    DWORD PTR [rbp-0x8],0xa
   0x0000555555555155 <+19>:    mov    DWORD PTR [rbp-0x4],0x14
   0x000055555555515c <+26>:    mov    eax,0x0
   0x0000555555555161 <+31>:    call   0x555555555129 <function2>
   0x0000555555555166 <+36>:    nop
   0x0000555555555167 <+37>:    leave
   0x0000555555555168 <+38>:    ret
End of assembler dump.
```

Function Calling

STACK
Space
Allocation

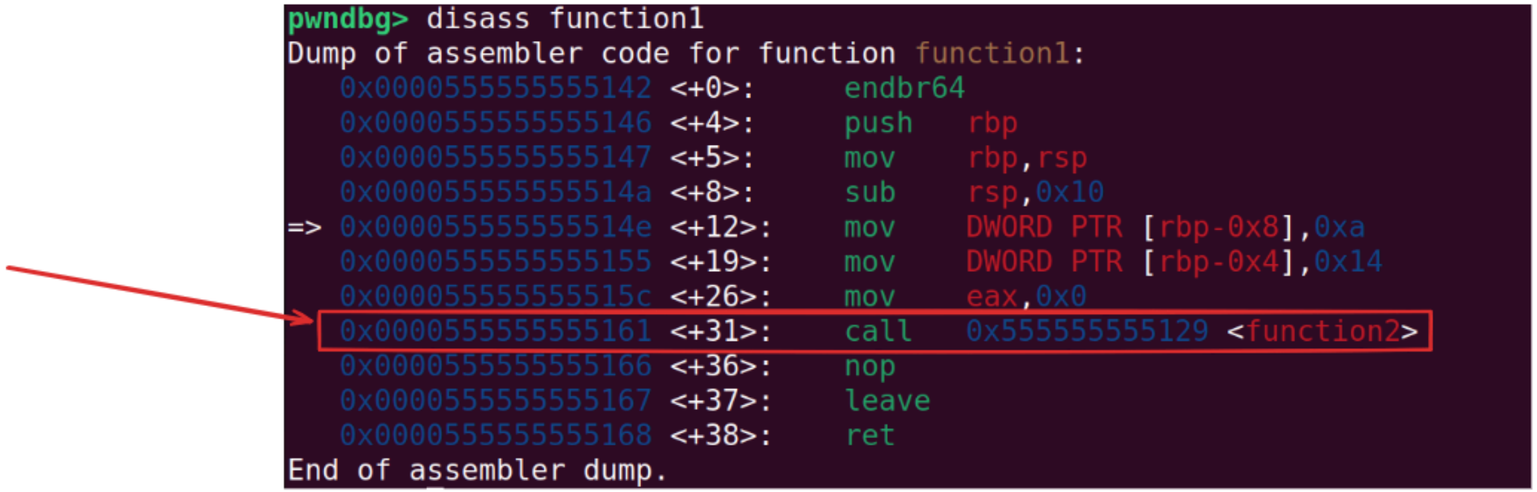


Function Calling

Local Variable

Function Calling

Calling function2



Function Exit?

Function Exit

exit function

Function Exit

exit function

```
pwndbg> disass function1
Dump of assembler code for function function1:
   0x0000555555555142 <+0>:     endbr64
   0x0000555555555146 <+4>:     push   rbp
   0x0000555555555147 <+5>:     mov    rbp,rsp
   0x000055555555514a <+8>:     sub    rsp,0x10
=> 0x000055555555514e <+12>:    mov    DWORD PTR [rbp-0x8],0xa
   0x0000555555555155 <+19>:    mov    DWORD PTR [rbp-0x4],0x14
   0x000055555555515c <+26>:    mov    eax,0x0
   0x0000555555555161 <+31>:    call   0x555555555129 <function2>
   0x0000555555555166 <+36>:    nop
   0x0000555555555167 <+37>:    leave
   0x0000555555555168 <+38>:    ret
End of assembler dump.
```

leave → mov rsp, rbp;
pop rbp;

QnA

No Question = Im a great teacher

No Question = Im a great teacher

No Question = Understand Nothing

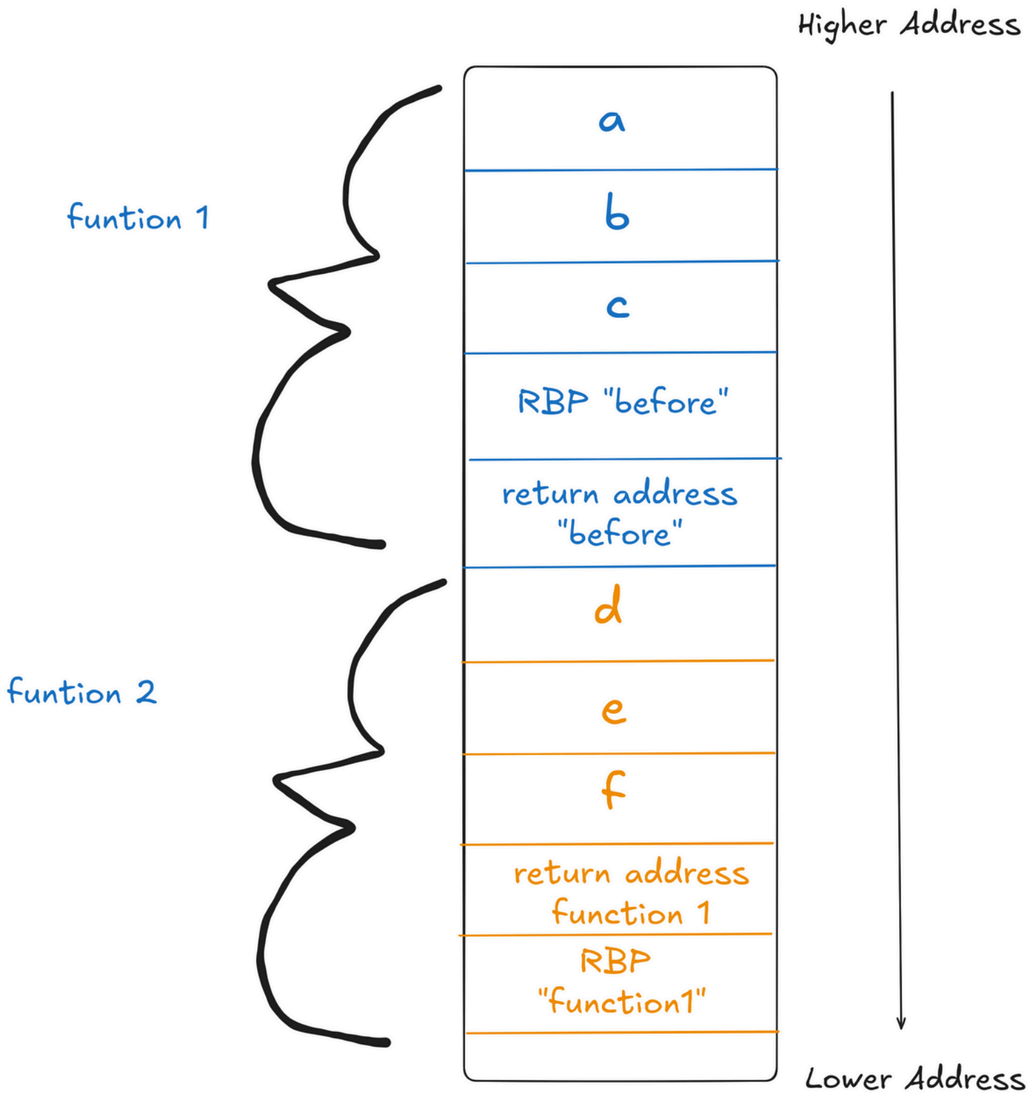
No Question = Understand Nothing

No Question = Understand Nothing

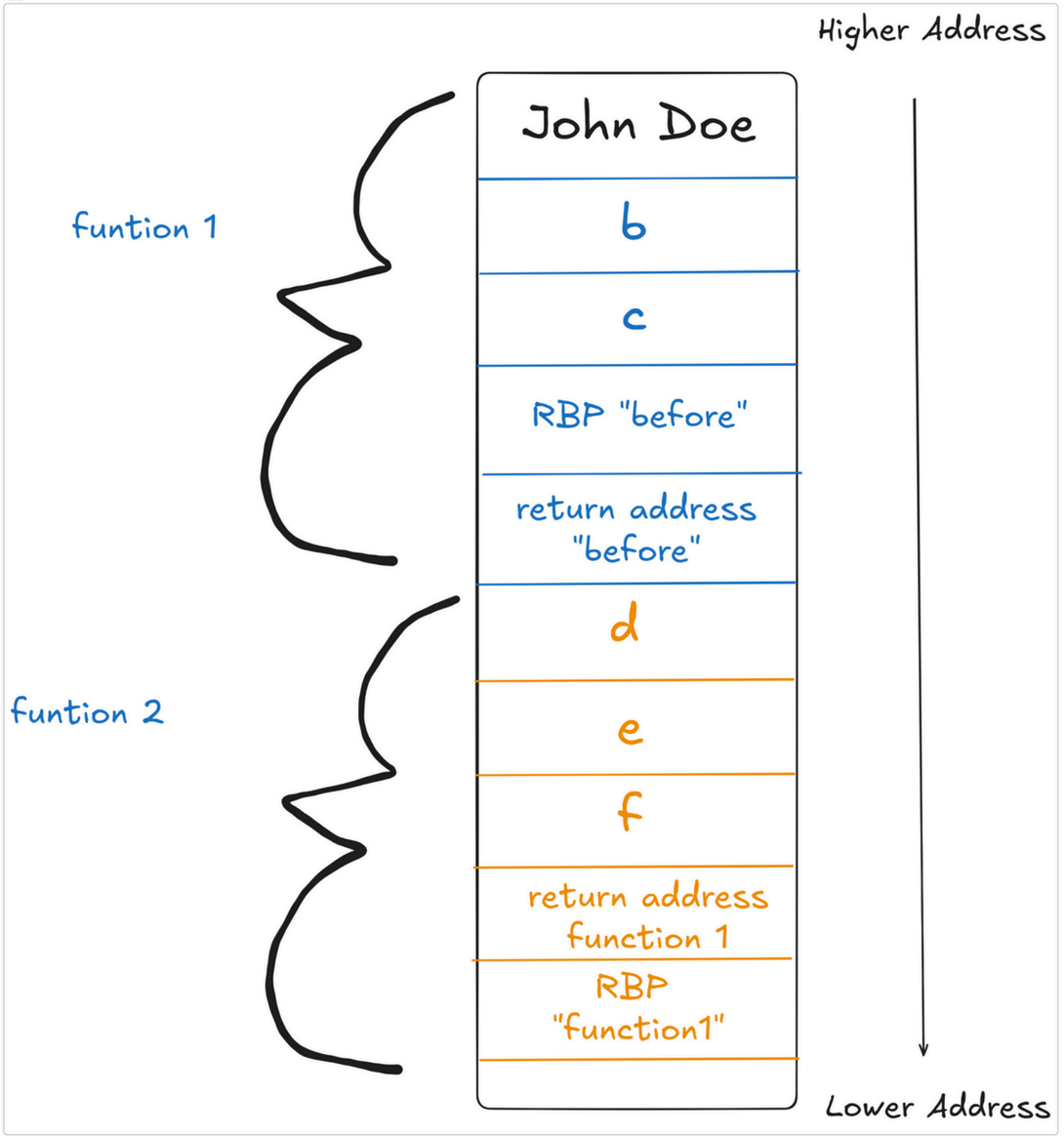
Dont Worry, Just need a lot of homework

Buffer Overflow

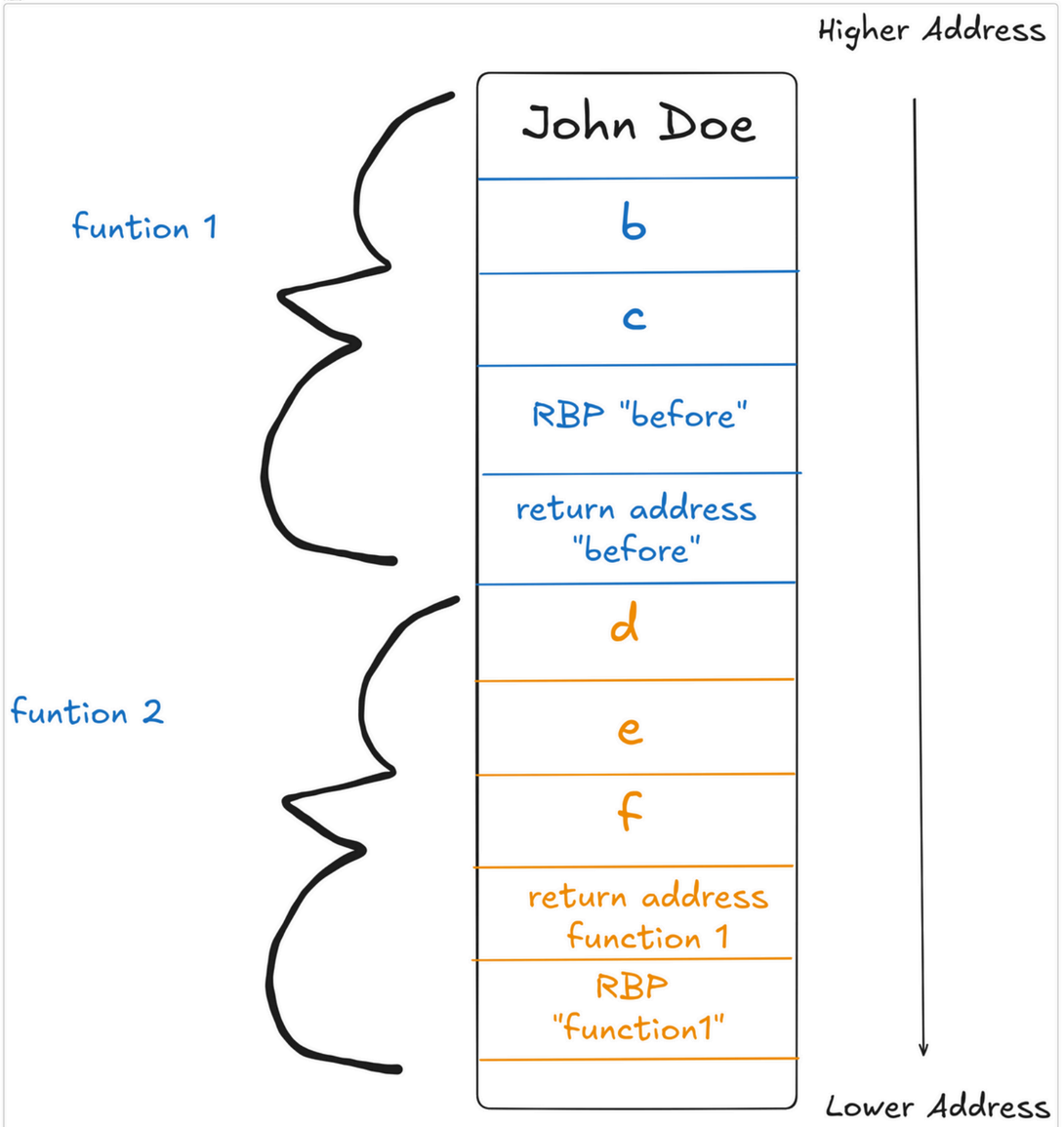
Buffer Overflow



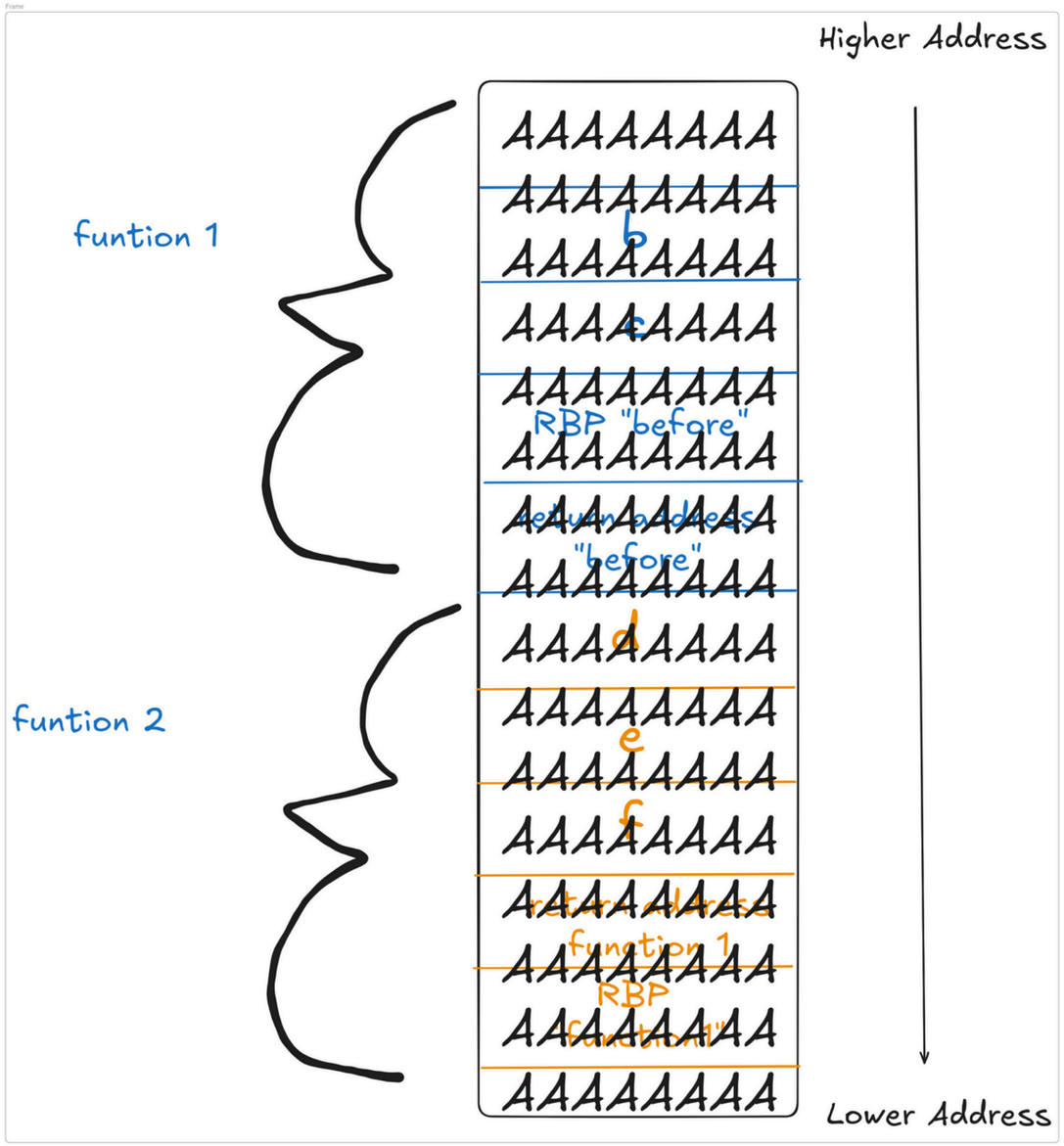
Buffer Overflow



Buffer Overflow

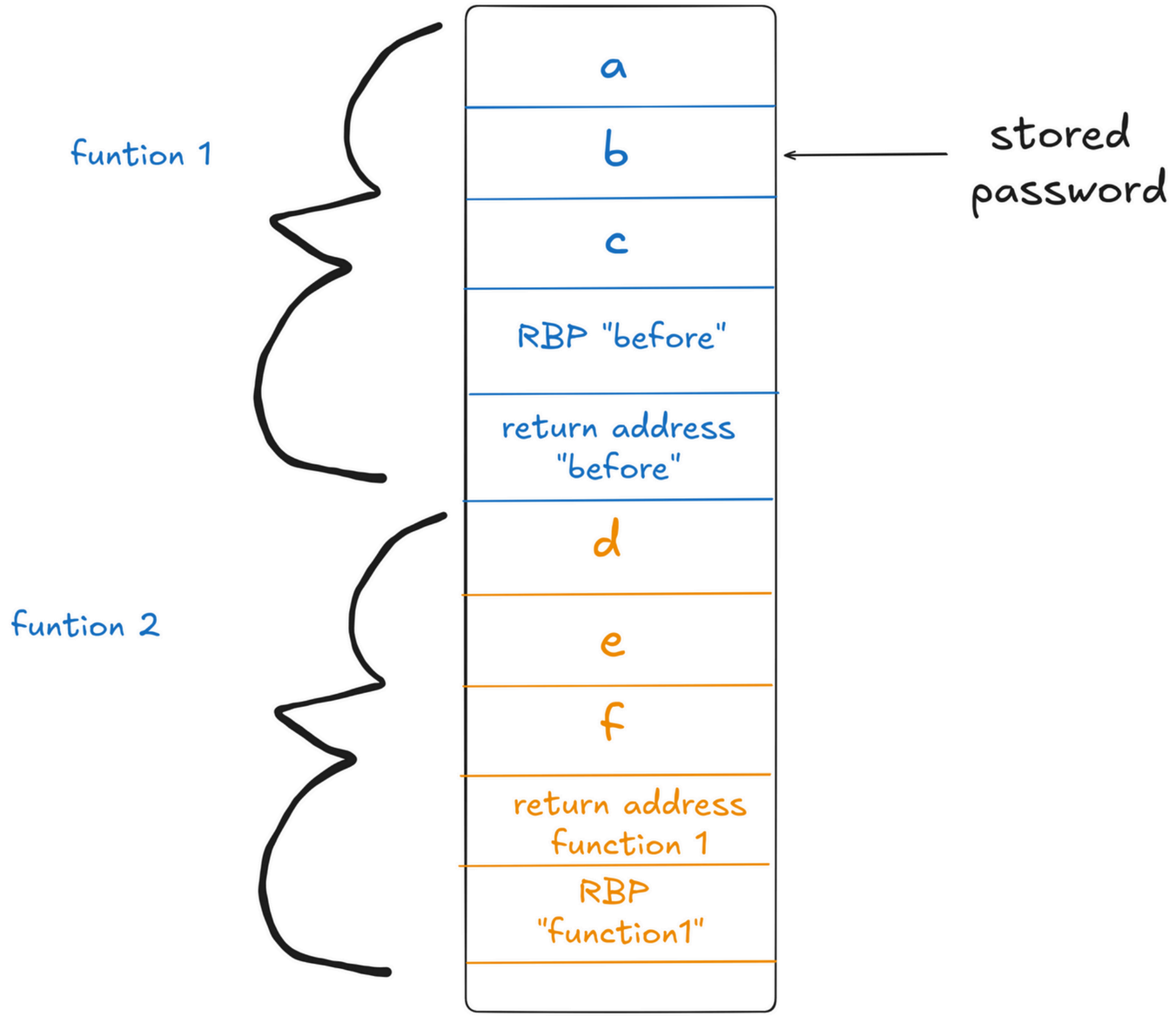


Buffer Overflow

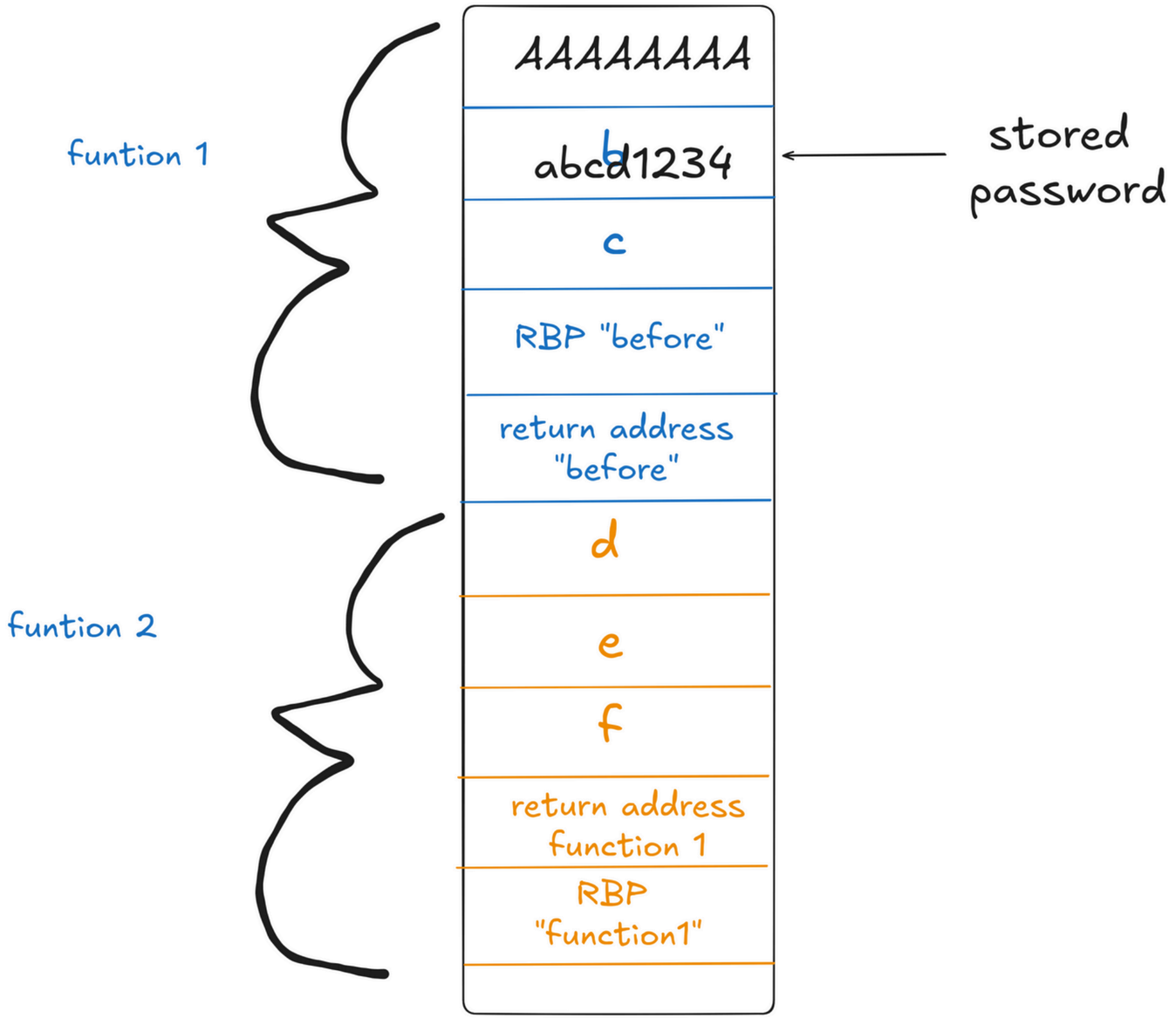


Buffer Overflow

Buffer Overflow

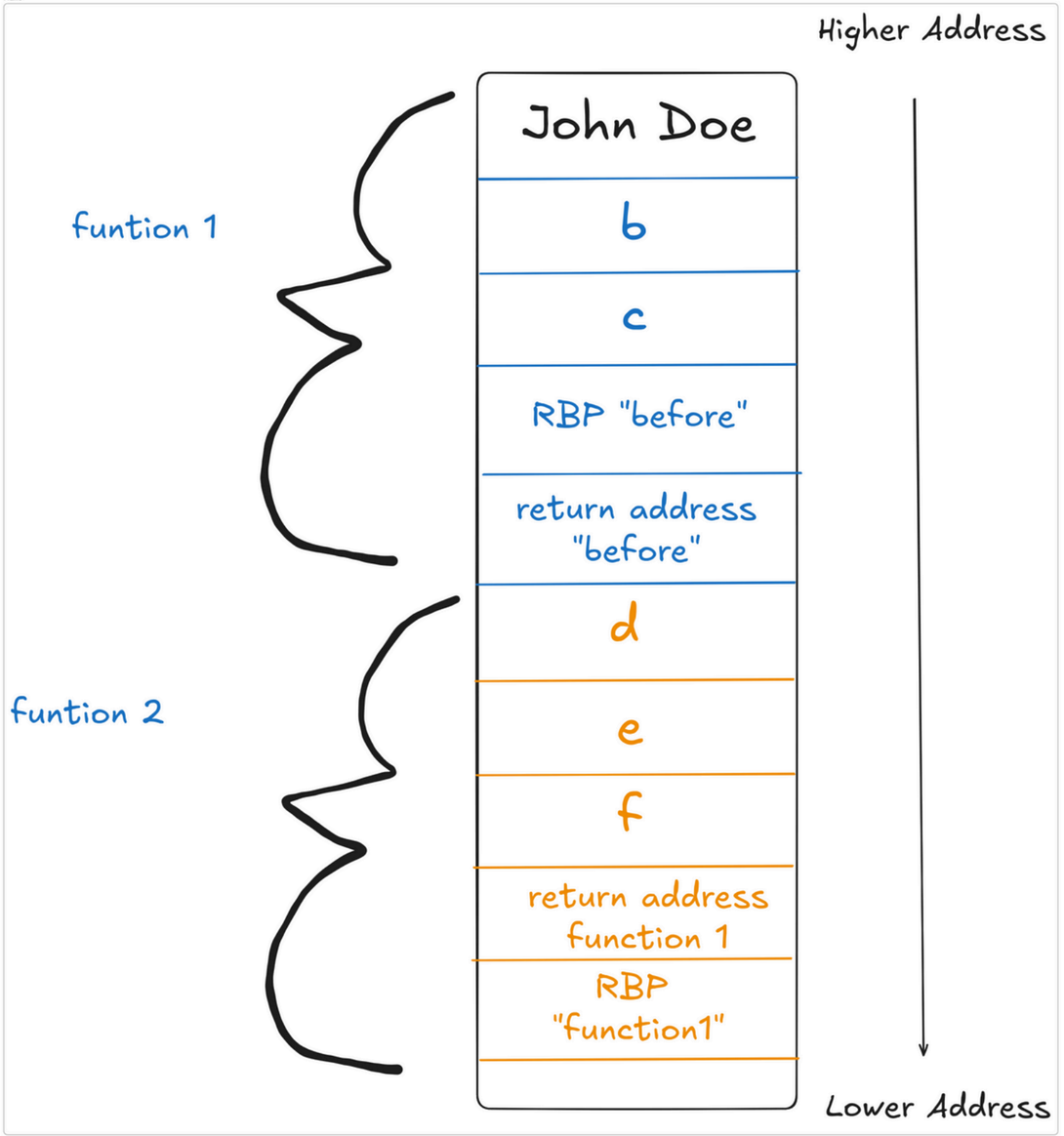


Buffer Overflow

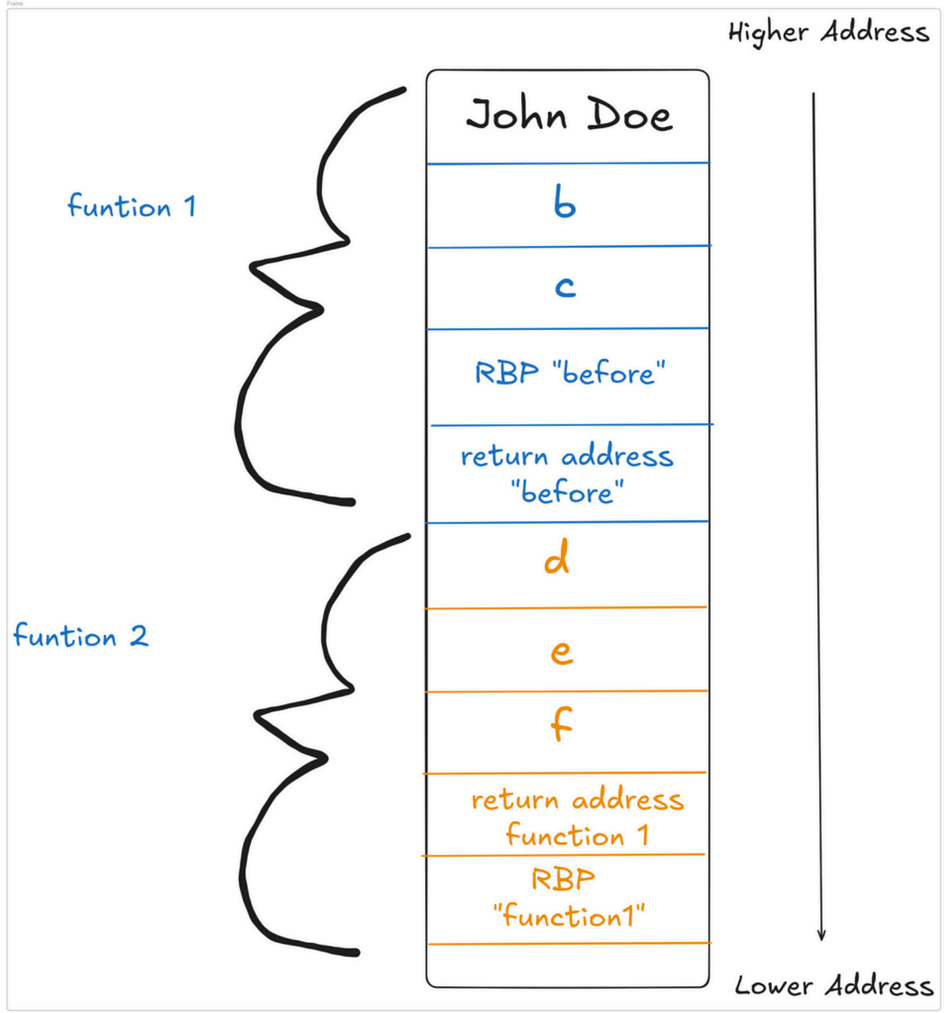


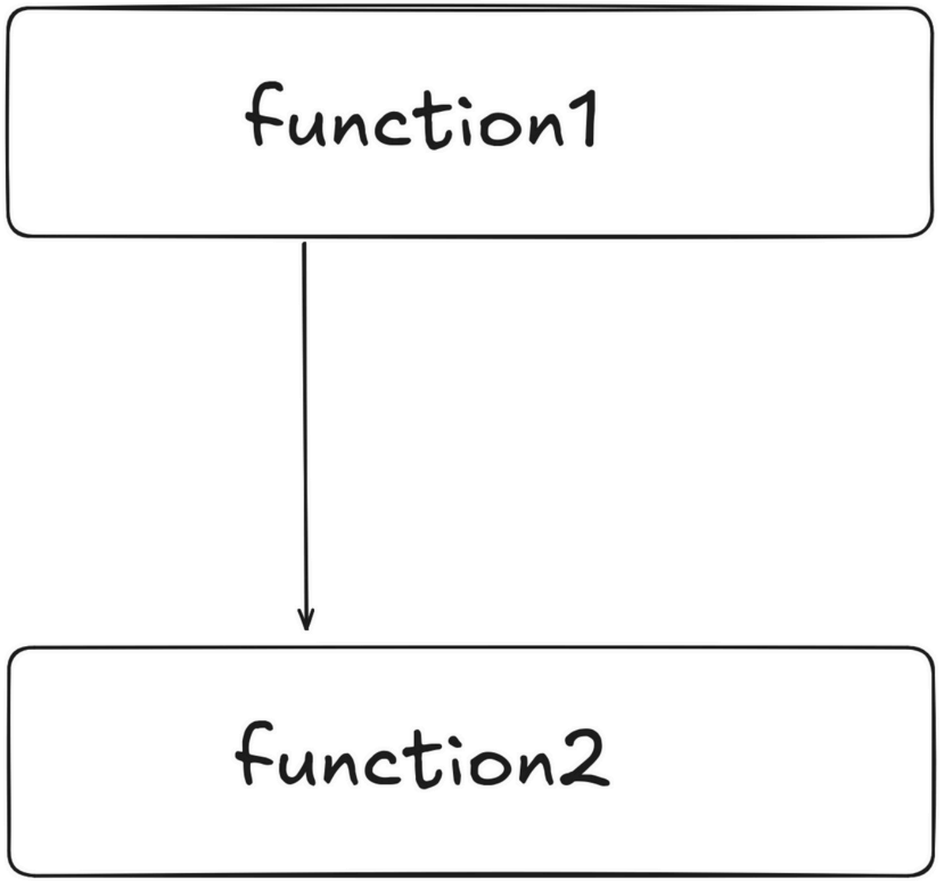
Return Oriented Programming

Return Oriented Programming

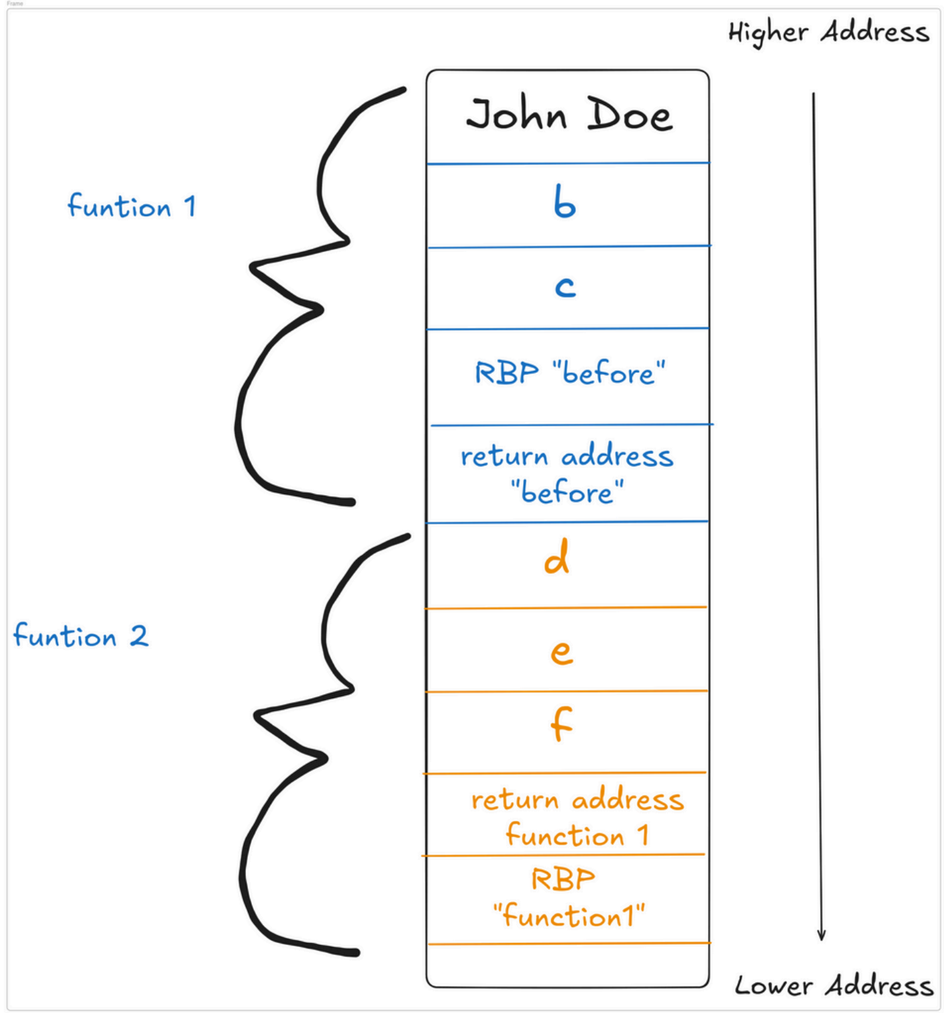


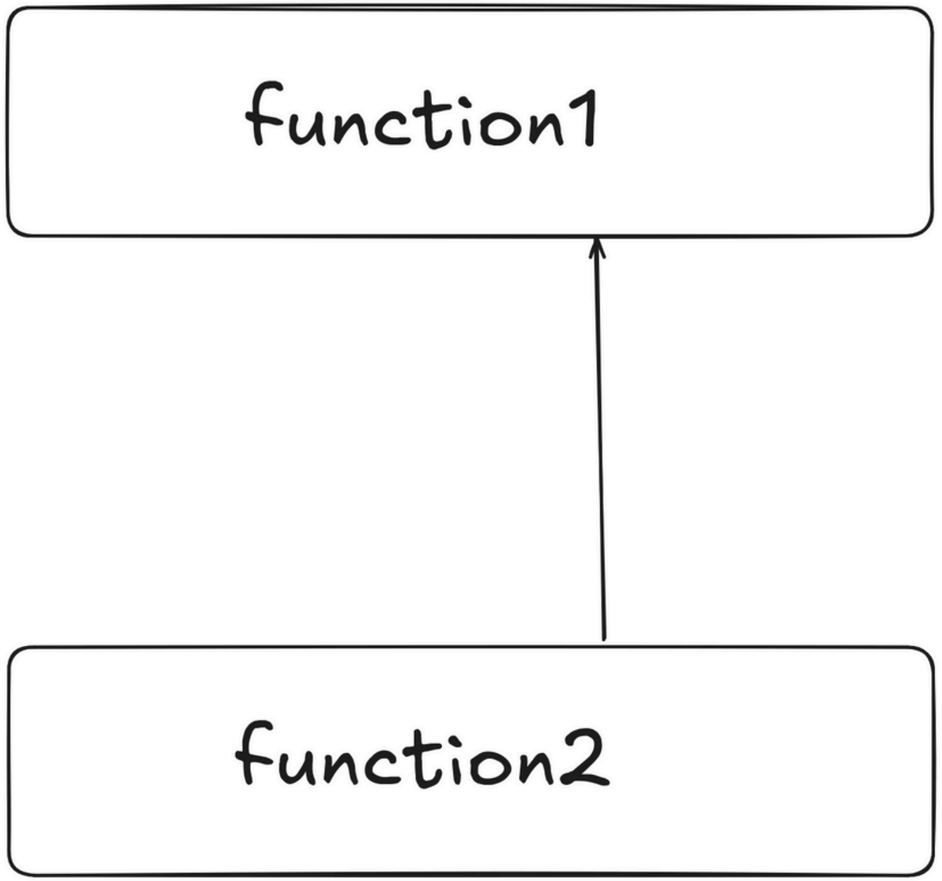
Return Oriented Programming



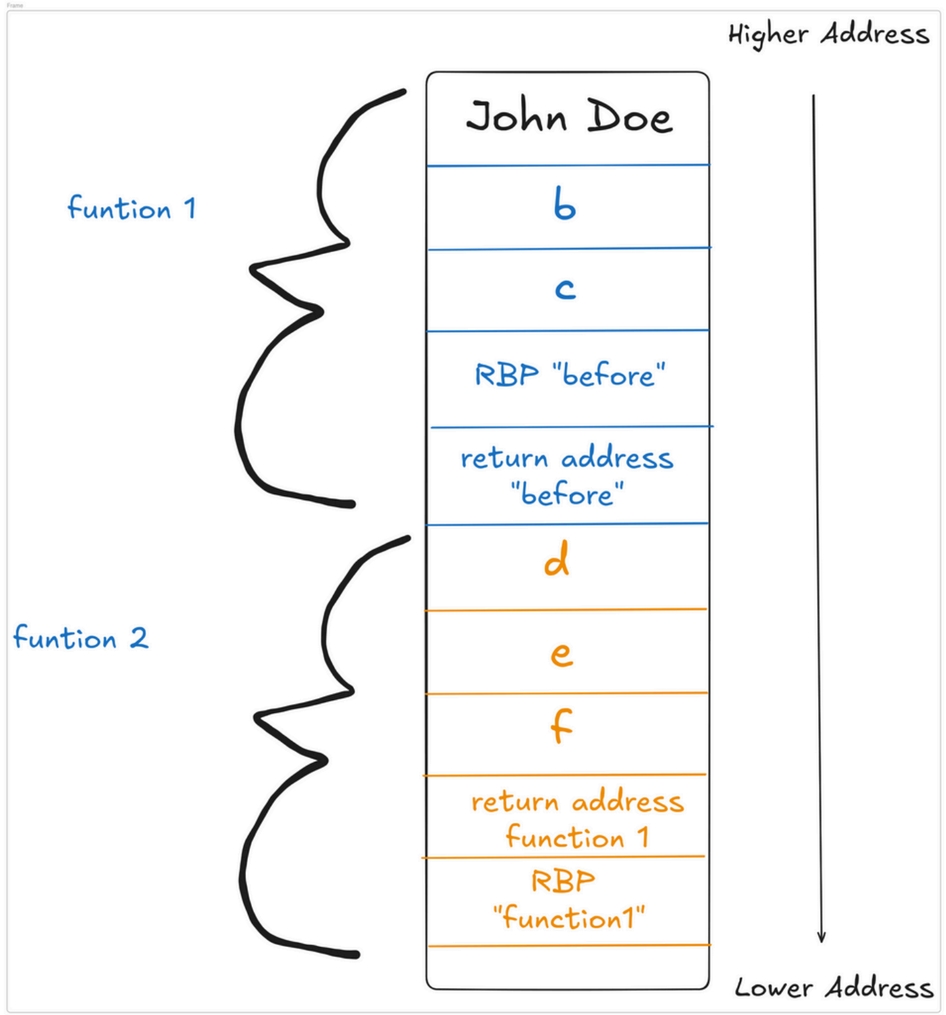


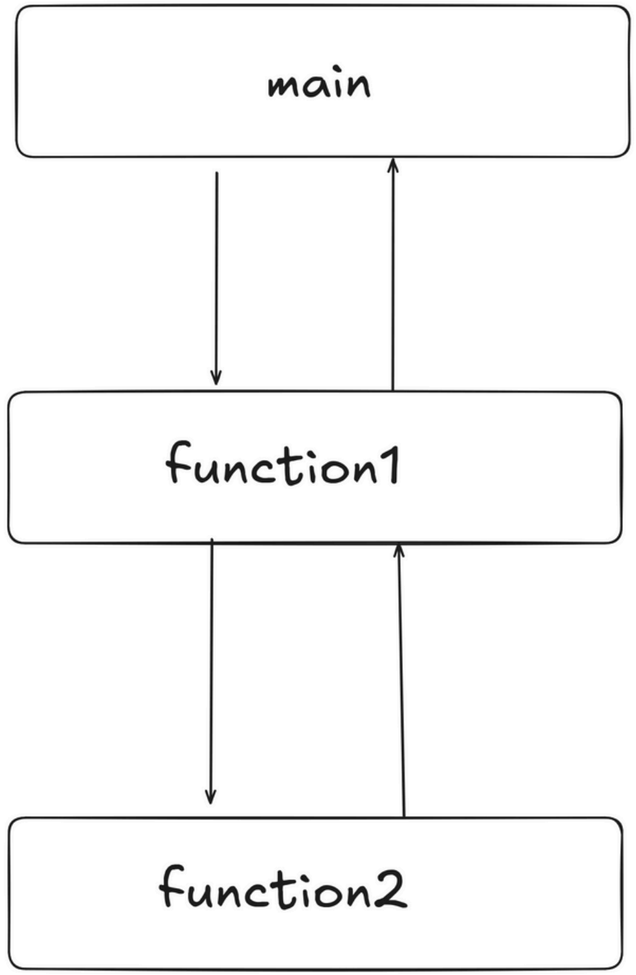
Return Oriented Programming



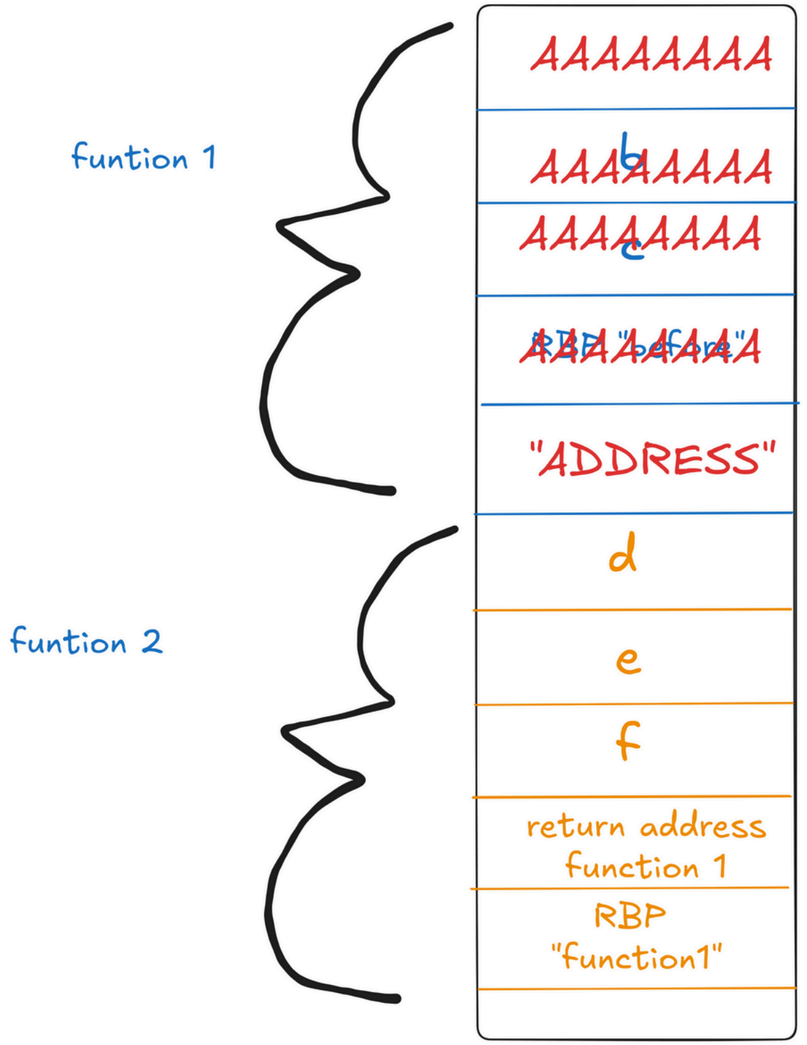


Return Oriented Programming

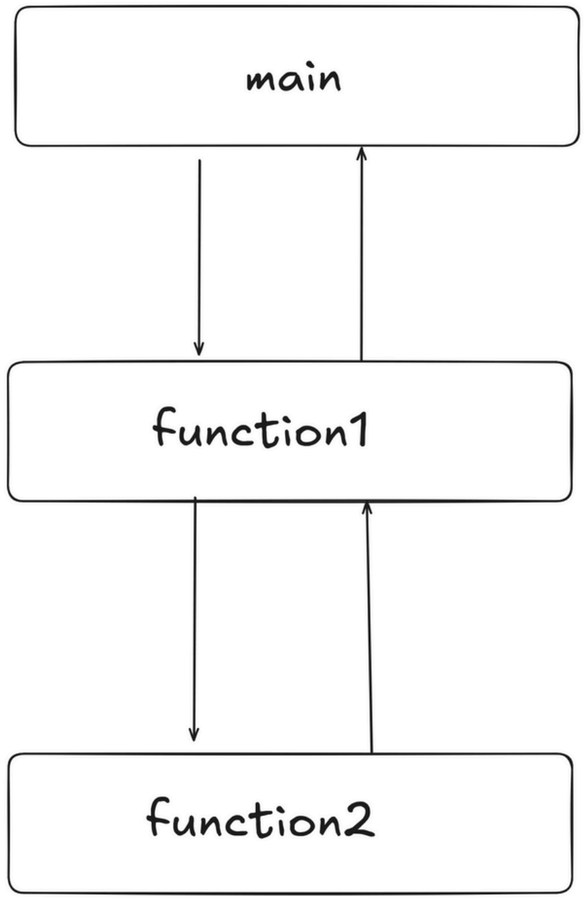


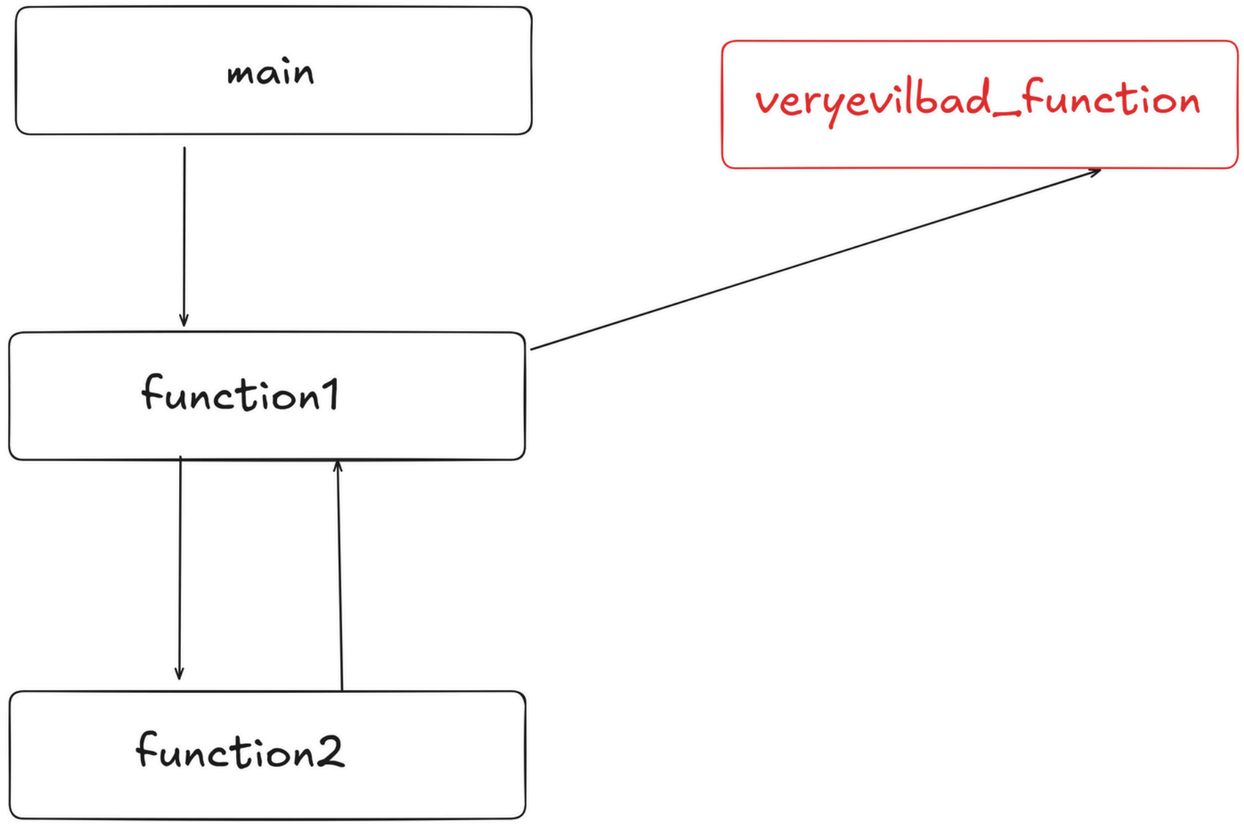


Return Oriented Programming

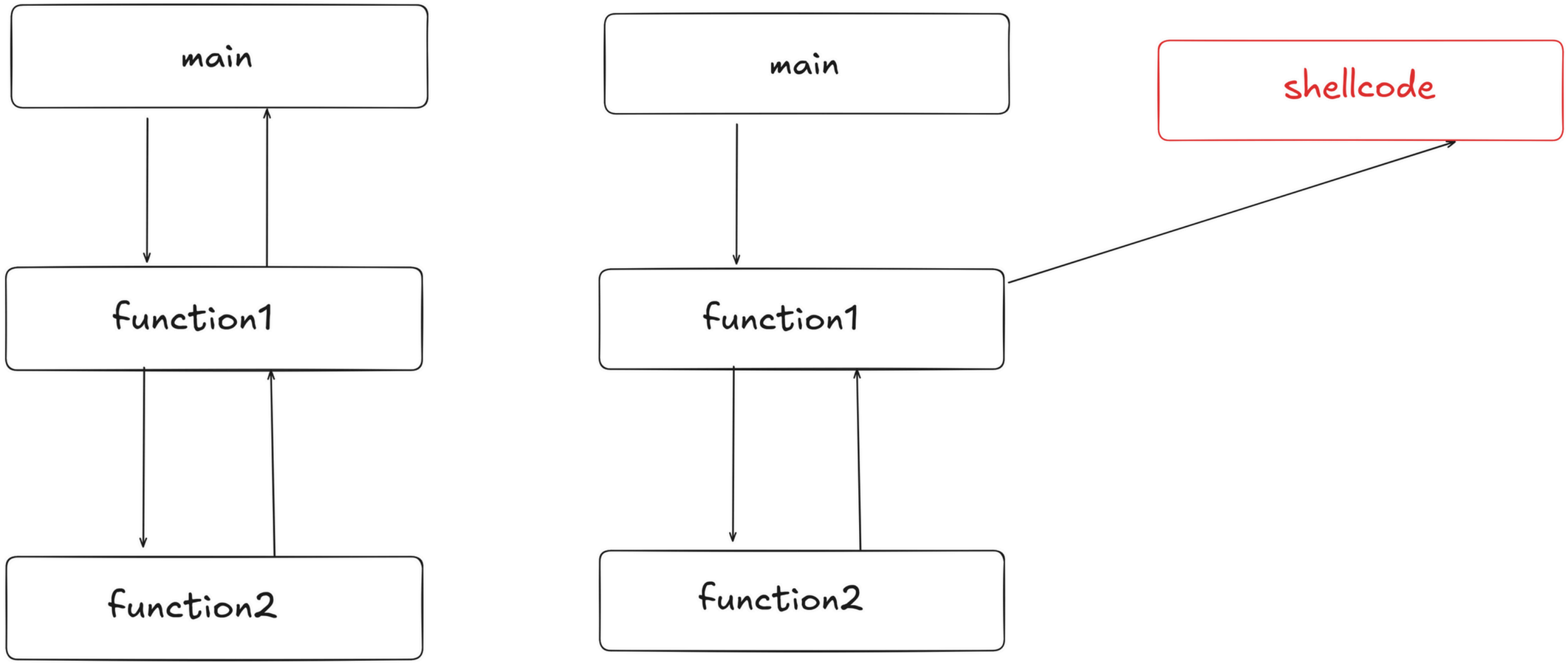


Return Oriented Programming





Return Oriented Programming



ROP Terminology

“gadgets”

A **gadget** is a **small sequence** of **machine instructions/asm code** that already exists somewhere in the binary, usually ending with a **“ret”**.

In exploitation, we weaponize these gadgets together to perform actions we want.

Can use **ROPgadget/objdump** to find these gadgets

```
0x00000000004011fd : pop rbp ; ret
0x0000000000401353 : pop rdi ; ret
0x0000000000401357 : pop rdx ; ret
0x0000000000401355 : pop rsi ; ret
0x0000000000401188 : push -0xffbfc0 ; loopne 0x4011f5 ; nop ; ret
0x000000000040134f : push rbp ; mov rbp, rsp ; pop rdi ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x000000000040105b : sar edi, 0xff ; call qword ptr [rax - 0x5e1f00d]
0x0000000000401361 : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000401360 : sub rsp, 8 ; add rsp, 8 ; ret
0x0000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x0000000000401183 : test eax, eax ; je 0x401190 ; mov edi, 0x404068 ; jmp rax
0x00000000004011c5 : test eax, eax ; je 0x4011d0 ; mov edi, 0x404068 ; jmp rax
0x000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

ROP Terminology

“gadgets”

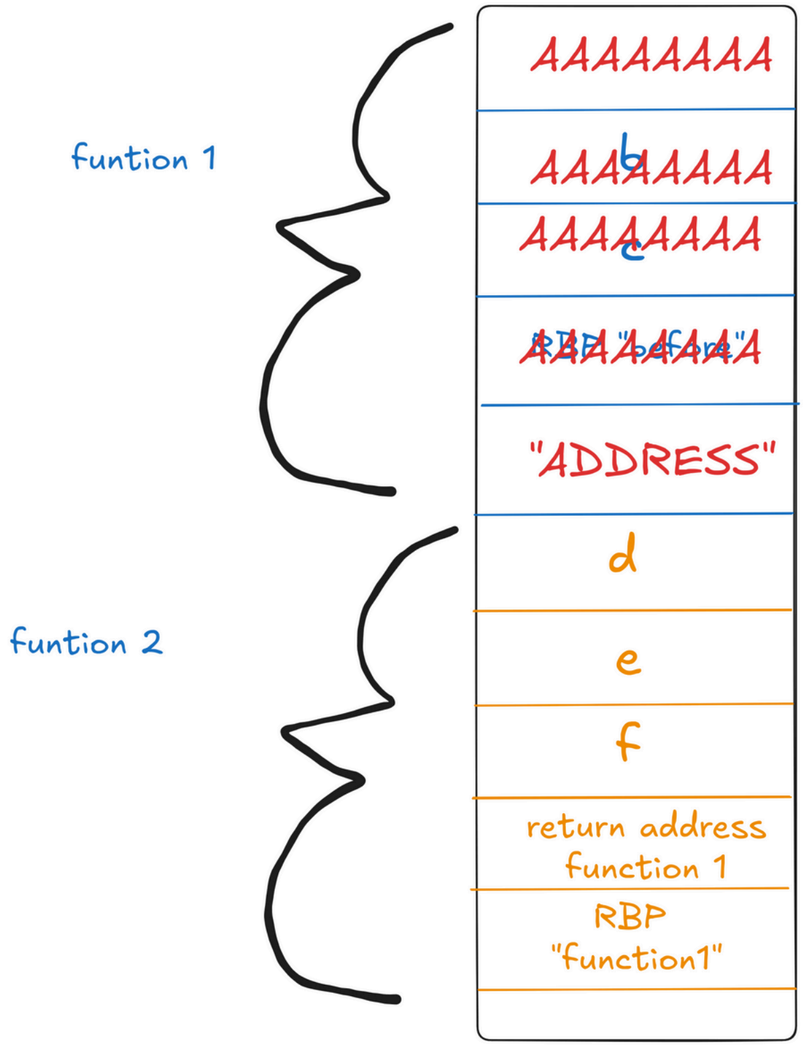
A **gadget** is a **small sequen** lready exists somewhere in the binary, usually ending

In exploitation, we weap s we want.

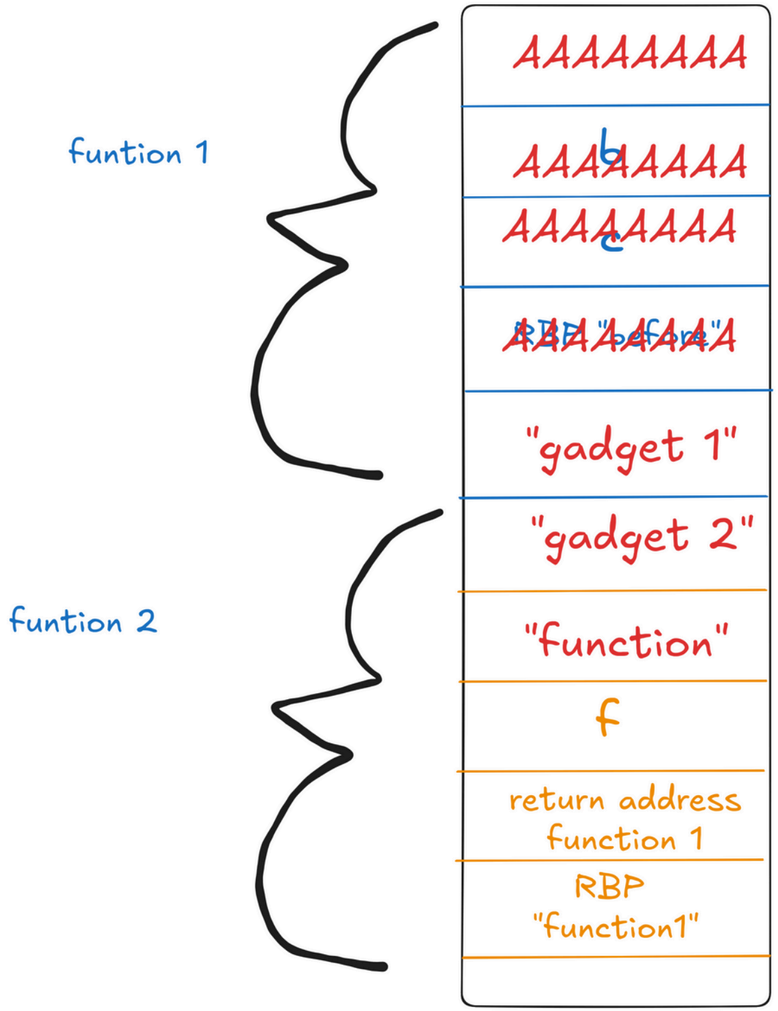
Can use **ROPgadget/objdump** to find these gadgets

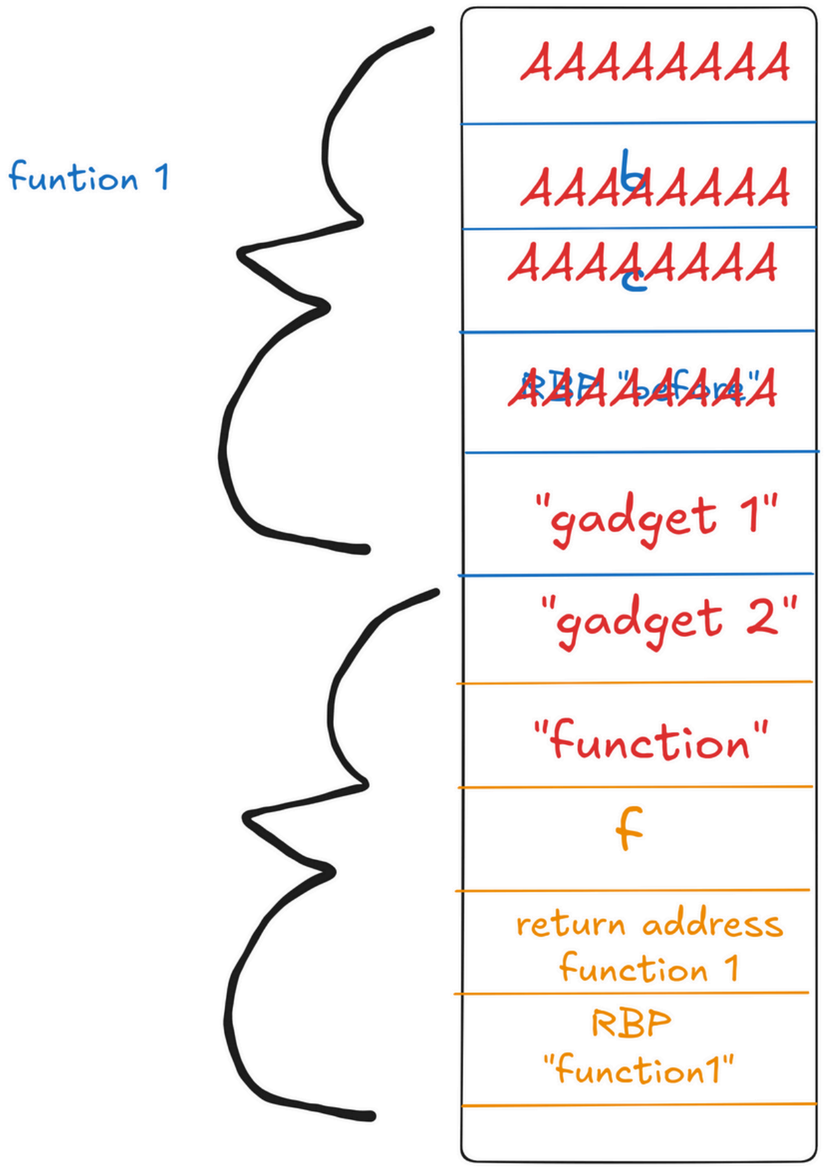
```
                                                    loopne 0x4011f5 ; nop ; ret
                                                    rbp, rsp ; pop rdi ; ret

                                                    [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
                                                    0xff ; call qword ptr [rax - 0x5e1f00d]
                                                    p, 8 ; add rsp, 8 ; ret
                                                    b rsp, 8 ; add rsp, 8 ; ret
                    0000401010 : test eax, eax ; je 0x401016 ; call rax
                    0000401183 : test eax, eax ; je 0x401190 ; mov edi, 0x404068 ; jmp rax
0x00000000004011c5 : test eax, eax ; je 0x4011d0 ; mov edi, 0x404068 ; jmp rax
0x000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

Return Oriented Programming



Return Oriented Programming





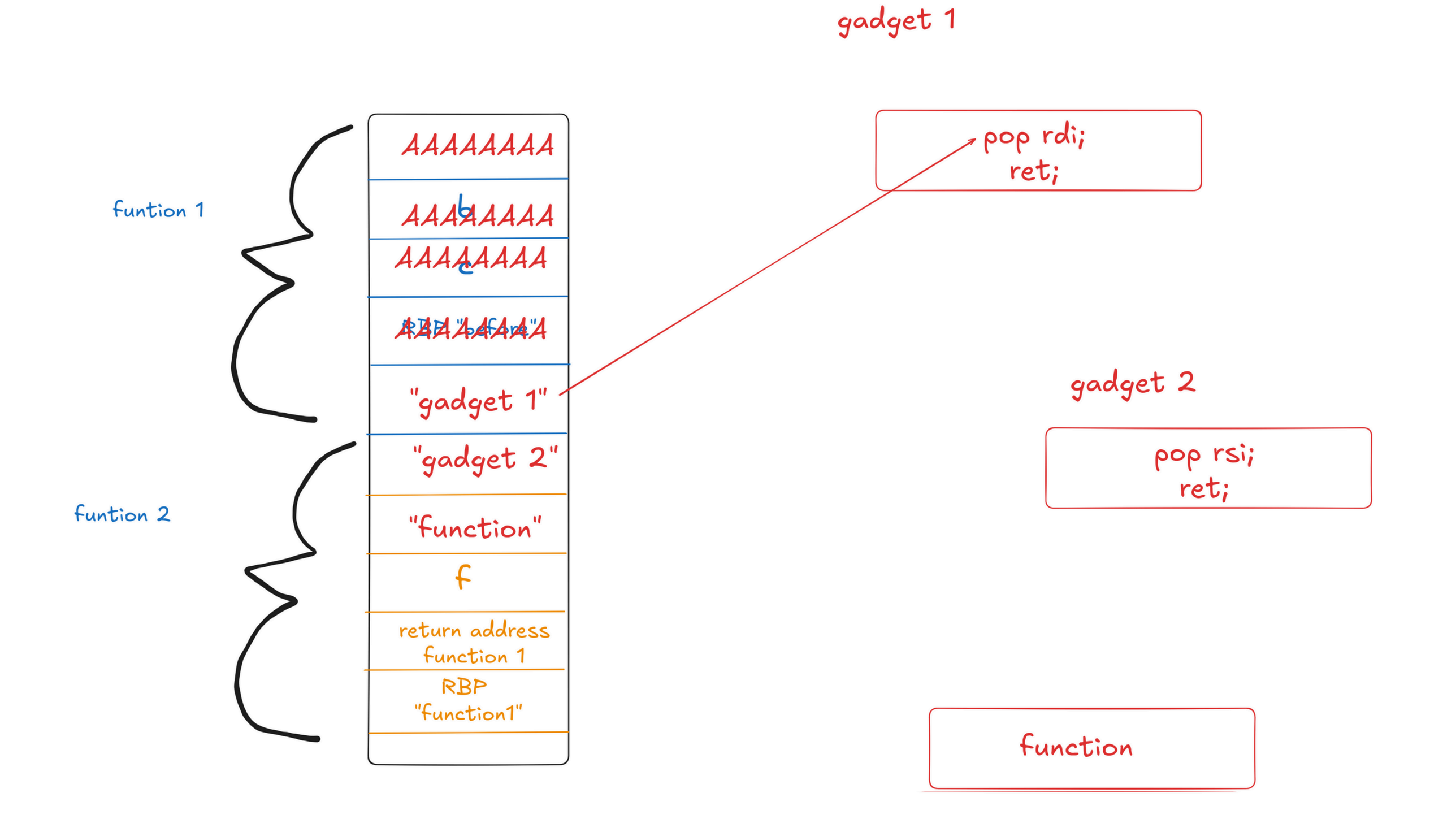
gadget 1

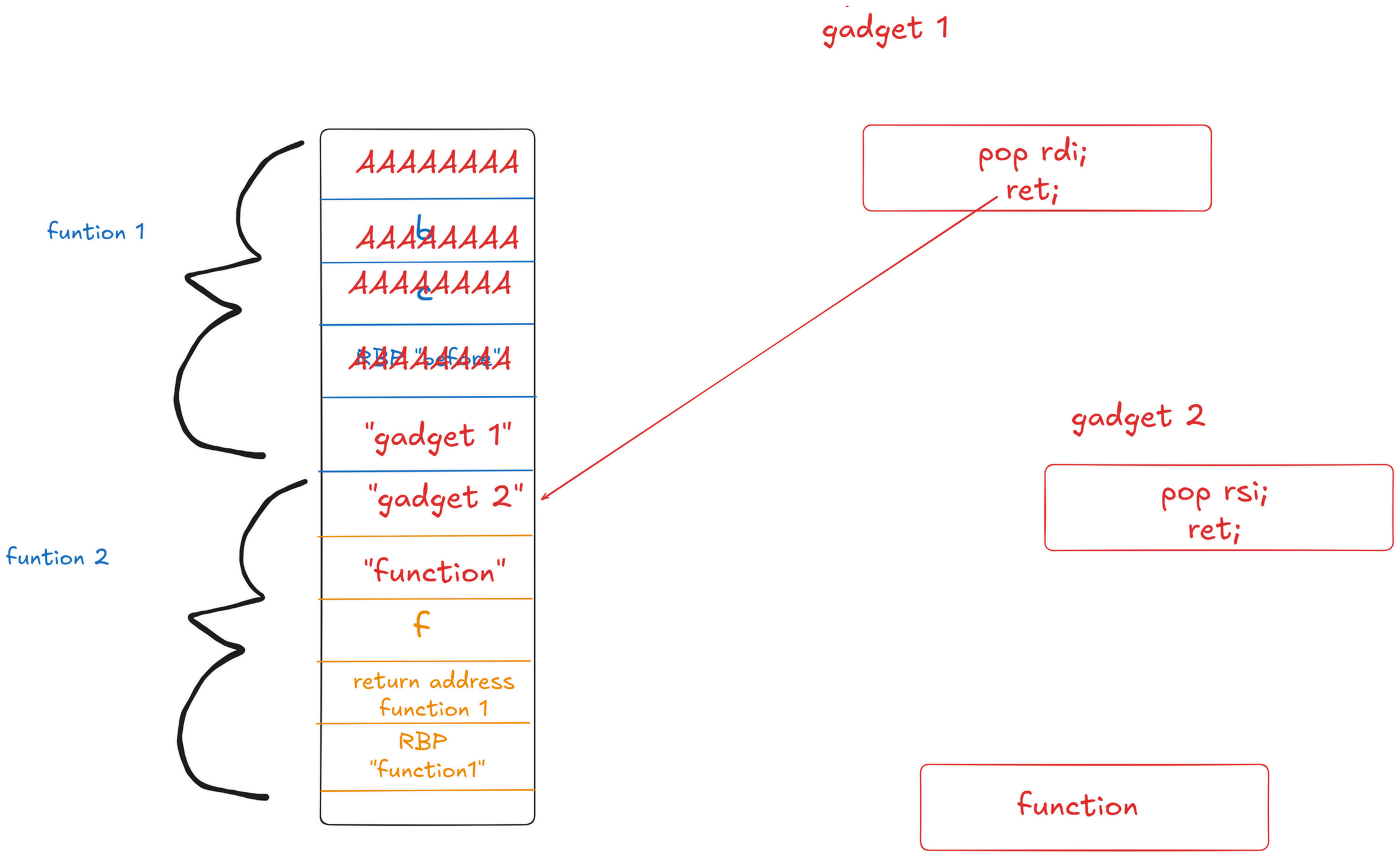
pop rdi;
ret;

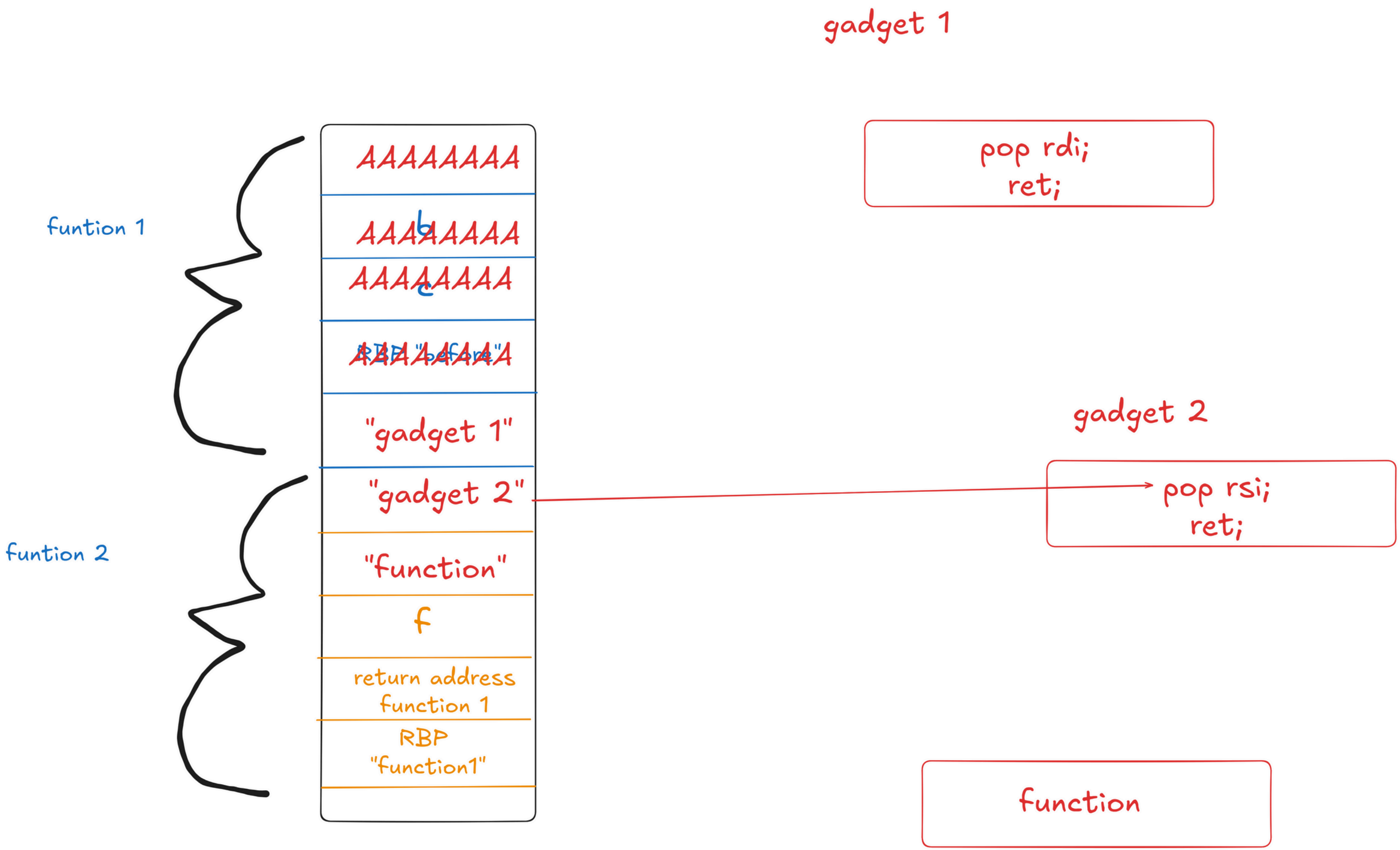
gadget 2

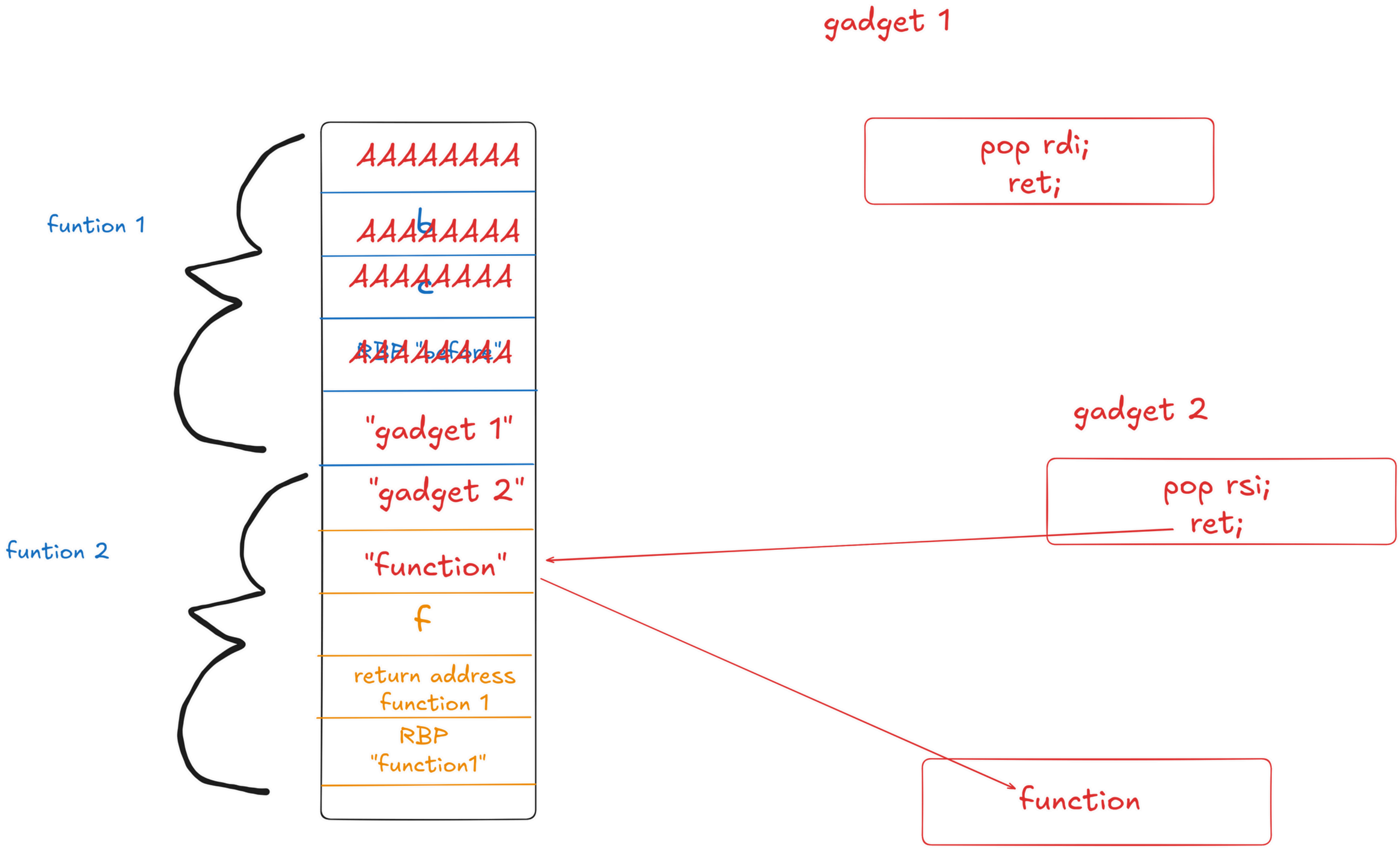
pop rsi;
ret;

function


gadget 1
funtion 1
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
"gadget 1"
"gadget 2"
"function"
f
return address
function 1
RBP
"function1"
pop rdi;
ret;
gadget 2
pop rsi;
ret;
funtion 2
function


gadget 1
funtion 1
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
"gadget 1"
"gadget 2"
"function"
f
return address
function 1
RBP
"function1"
pop rdi;
ret;
gadget 2
pop rsi;
ret;
funtion 2
function


gadget 1
pop rdi;
ret;
funtion 1
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
"gadget 1"
"gadget 2"
gadget 2
pop rsi;
ret;
funtion 2
"function"
f
return address
function 1
RBP
"function1"
function


gadget 1
pop rdi;
ret;
funtion 1
AAAAAAAA
AAAAAAAA
AAAAAAAA
AAAAAAAA
"gadget 1"
"gadget 2"
gadget 2
pop rsi;
ret;
funtion 2
"function"
f
return address
function 1
RBP
"function1"
function

Hands On

QnA

Challenge !!!!

Challenge Walkthrough

THANK YOU

Acknowledgement:

RE:HACK, cyber673, Yappare, Jia Yang ... etc