Binary exploitation 101

RecursionFairies@UNITN - 27/02/2019

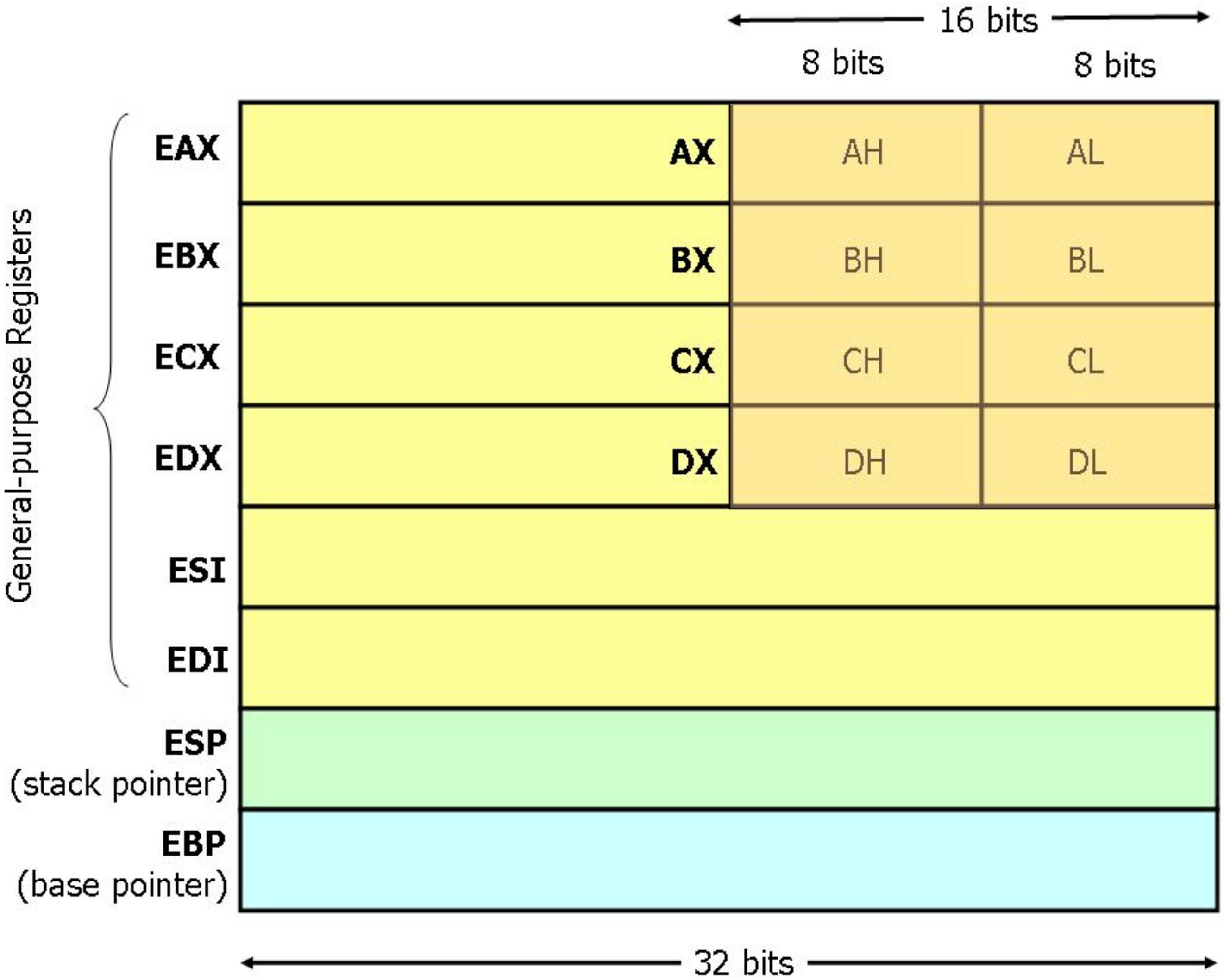
https://goo.gl/kgc9n7

Outline

- Intro to Intel x86 assembly
- Program execution simulation
- Disassembling binary
- Buffer overflow 101
- Buffer overflow 102 - return to shellcode
- Buffer overflow mitigations

Intro to Intel x86 assembly

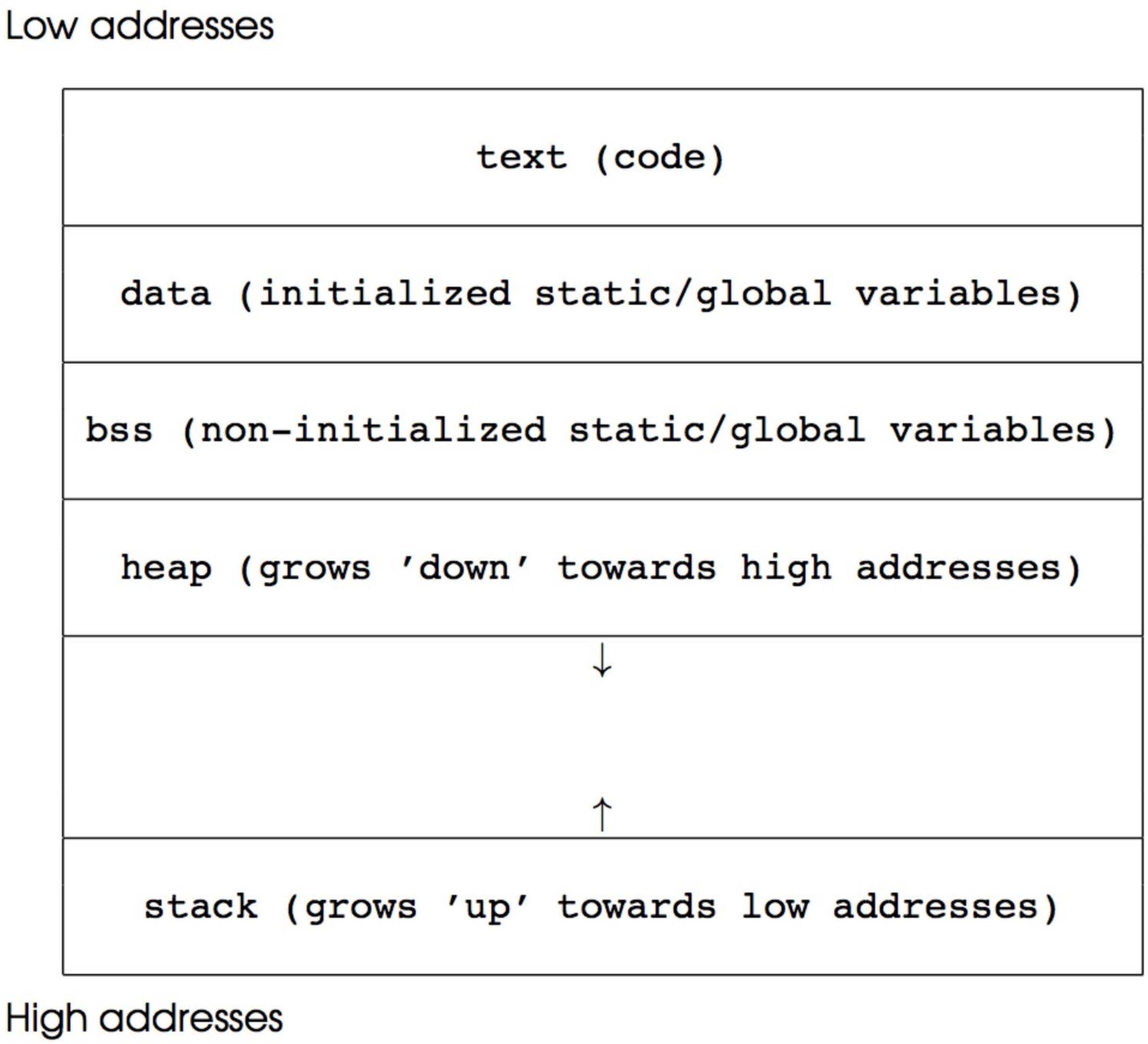
x86 registers



x86 registers 2

- **EAX, EBX, ECX, EDX:** General purpose registers (Accumulator, Base, Counter, Data)
- **ESI, EDI:** Source and destination Indexes, used in array and string copying
- **ESP:** Stack pointer, "top" of the current stack frame
- **EBP:** Base pointer, "bottom" of the current stack frame
- **EIP:** Instruction pointer, pointer to the next instruction to be executed by the CPU
- **EFLAGS:** stores flag bits
  - ZF: zero flag, set when result of an operation equals zero
  - CF: carry flag, set when the result of an operation is too large/small
  - SF: sign flag, set when the result of an operation is negative

C program memory map



Low addresses

text (code)

data (initialized static/global variables)

bss (non-initialized static/global variables)

heap (grows 'down' towards high addresses)

↓

↑

stack (grows 'up' towards low addresses)

High addresses

Stack

- Region of memory where local variables are stored
- Supports *push* and *pop* operations
- Grows towards lower memory addresses → higher values have lower address
- When a function is called a stack frame is set up
  - EBP contains the address of the base of the current stack frame
  - ESP contains the address of the top element of the current stack frame
- Each function call pushes the arguments and the return address to the stack

x86 Instructions

- Moving data and arithmetic operations
- Stack manipulation
- Control flow

NOTE: There are different syntaxes (i.e. two different way to write):

- mov eax,1 → Intel syntax
- movl $1,%eax → AT&T syntax

Moving data and arithmetic operations

- **mov** <*dst*>*,* <*src*> : copies the value <*src*> to <*dst*>
- **lea** <*dst*>*,* <*src*> : loads the address of <*src*> into <*dst*>
- **add** <*dst*>*,* <*src*> : adds the value in <*src*> to <*dst*>
- **sub** <*dst*>*,* <*src*> : subtracts the value in from
- **and** <*dst*>*,* <*src*> : performs a logical and between <*src*> and <*dst*>, placing the result in <*dst*>
- **cmp** <*dst*>*,* <*src*> : compares <*src*> with <*dst*>. This is done by subtracting <*src*> from <*dst*> and updating flags in the flag register

NOTE: There are various addressing modes!

- mov DWORD PTR [ebp - 4], eax
- mov eax, 10

Stack manipulation

- **push** <*val*> : pushes the value <*val*> in to the stack
- **pop** <*addr*> : pops a value from the stack into <*val*>

Control flow

- **jle** <*addr*> : jumps to the address <*addr*> if the <*dst*> in the last *cmp* was less or equal to the <*src*>
- **jge** <*addr*> : jumps to the address <*addr*> if the <*dst*> in the last *cmp* was greater or equal to the <*src*>
- **jmp** <*addr*> : jumps to the address <*addr*>
- **call** <*addr*> : calls the function at <*addr*>. Before the jump is taken, the address of the next instruction is pushed to the stack as the return address of the called function
- **ret** : pops the return address off the stack and returns control to that location
- **nop** : does nothing (will become useful later)

Program execution simulation

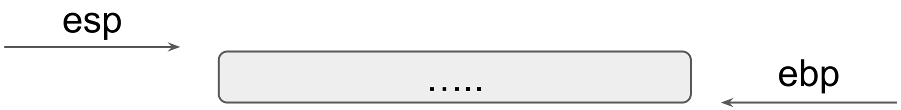
Stack frame

C:

```
...
fun(10);
...
```

asm:

```
push 10
call func    // push next
             // instr addr
             // and jmp to func
```



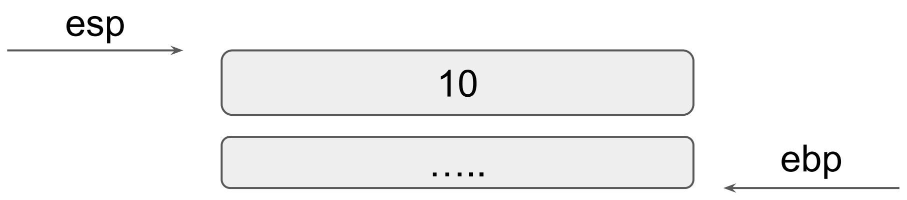
Stack frame

C:

```
...
fun(10);
...
```

asm:

```
→ push 10
  call func    // push next
               // instr addr
               // and jmp to func
```



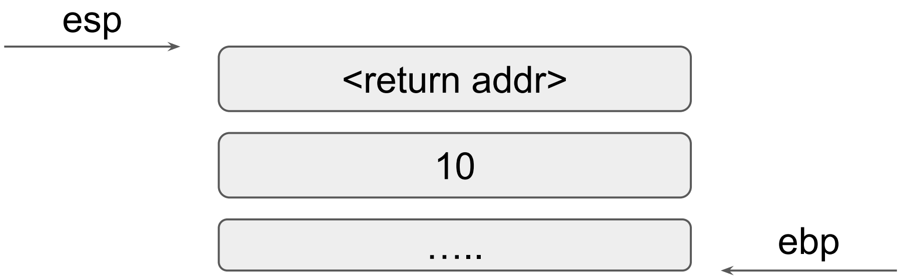
Stack frame

C:

```
...
fun(10);
...
```

asm:

```
push 10
call func    // push next
             // instr addr
             // and jmp to func
```



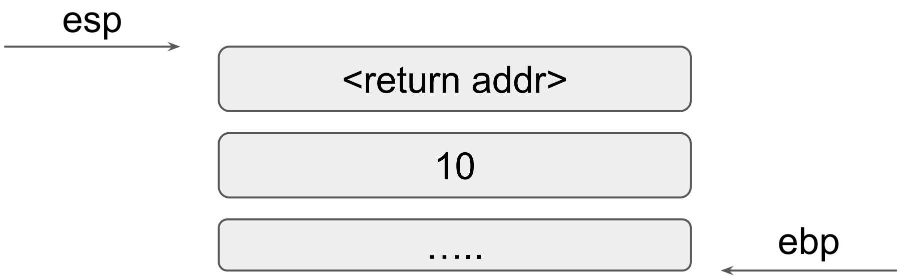
Stack frame

C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
```



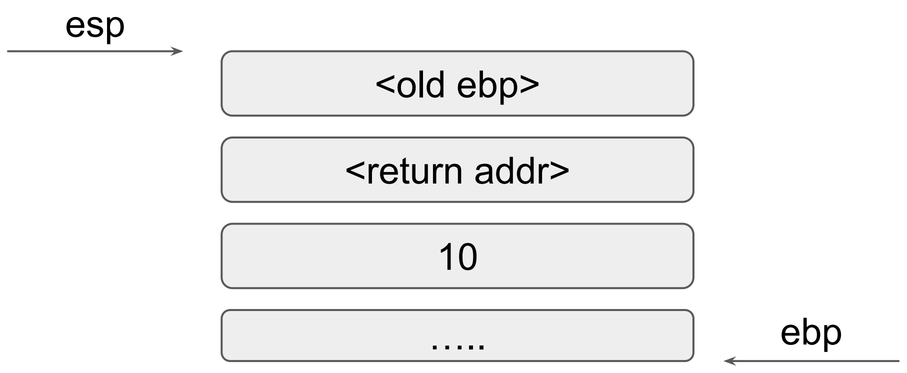
Stack frame

C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
push ebp
mov ebp, esp
sub esp, 8
mov DWORD PTR [ebp - 4], 0
mov eax, DWORD PTR [ebp + 8]
mov DWORD PTR [ebp - 8], eax
```



esp

<old ebp>

<return addr>

10

.....

ebp

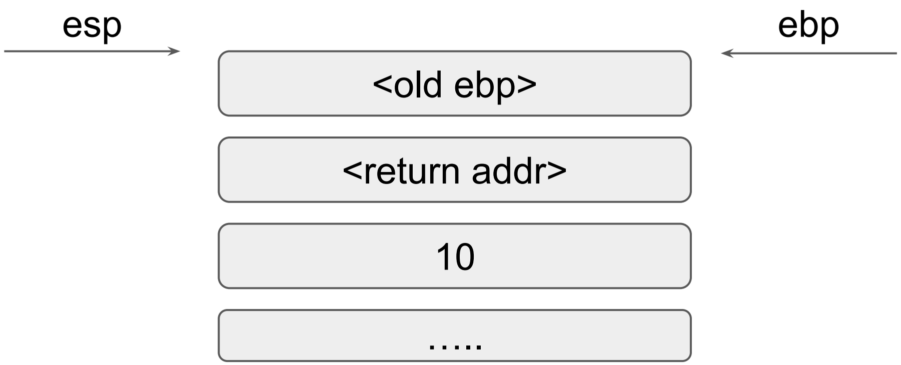
Stack frame

C:

```
int func (int x) {
    int a = 0;
    int b = x;
    ...
}
```

asm:

```
  push ebp
→ mov ebp, esp
  sub esp, 8
  mov DWORD PTR [ebp - 4], 0
  mov eax, DWORD PTR [ebp + 8]
  mov DWORD PTR [ebp - 8], eax
```



esp

ebp

<old ebp>

<return addr>

10

.....

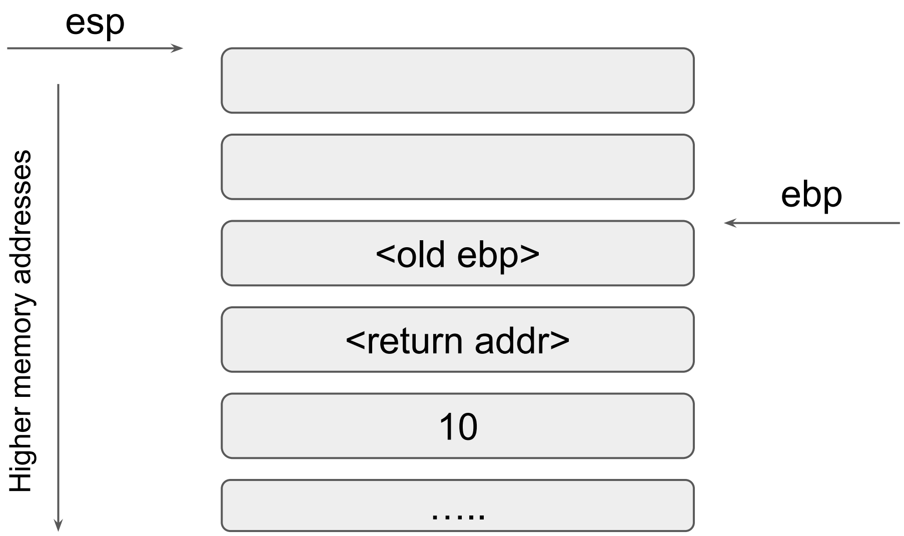
Stack frame

C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
  push ebp
  mov ebp, esp
→ sub esp, 8
  mov DWORD PTR [ebp - 4], 0
  mov eax, DWORD PTR [ebp + 8]
  mov DWORD PTR [ebp - 8], eax
```



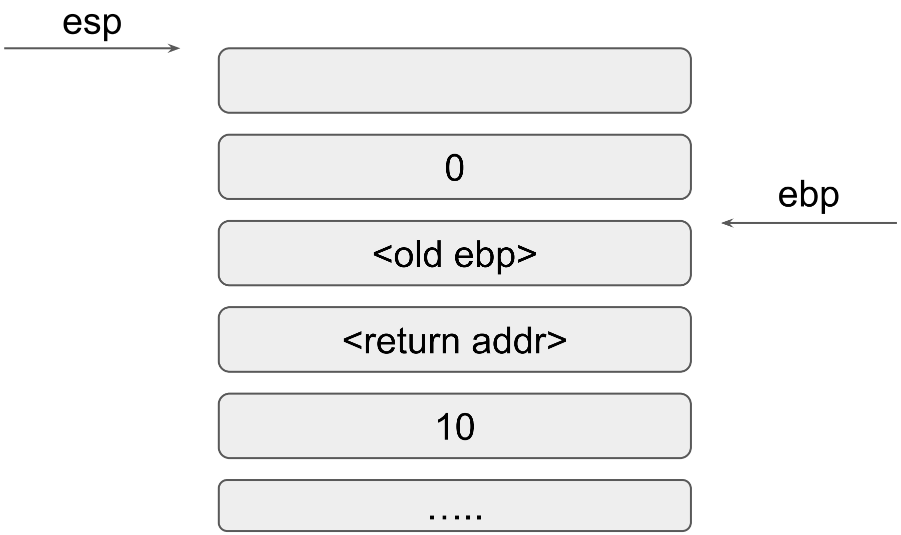
Stack frame

C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
    push ebp
    mov ebp, esp
    sub esp, 8
 →  mov DWORD PTR [ebp - 4], 0
    mov eax, DWORD PTR [ebp + 8]
    mov DWORD PTR [ebp - 8], eax
```



esp

0

ebp

<old ebp>

<return addr>

10

.....

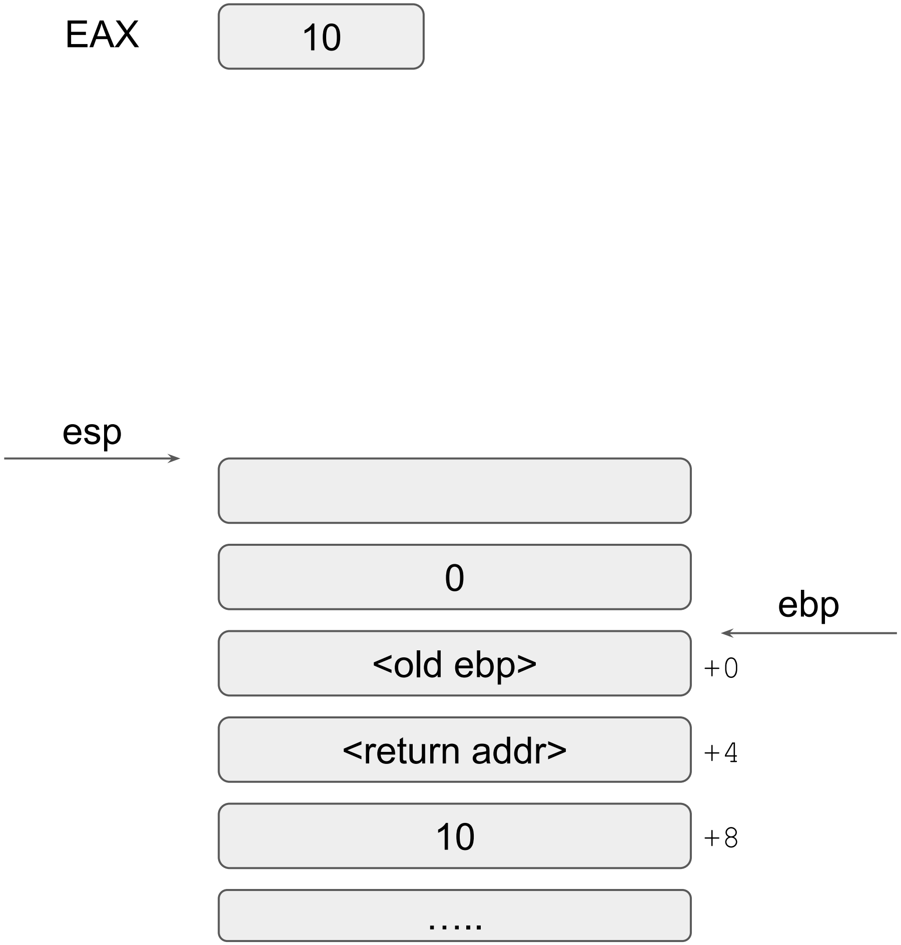
Stack frame

C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
  push ebp
  mov ebp, esp
  sub esp, 8
  mov DWORD PTR [ebp - 4], 0
→ mov eax, DWORD PTR [ebp + 8]
  mov DWORD PTR [ebp - 8], eax
```



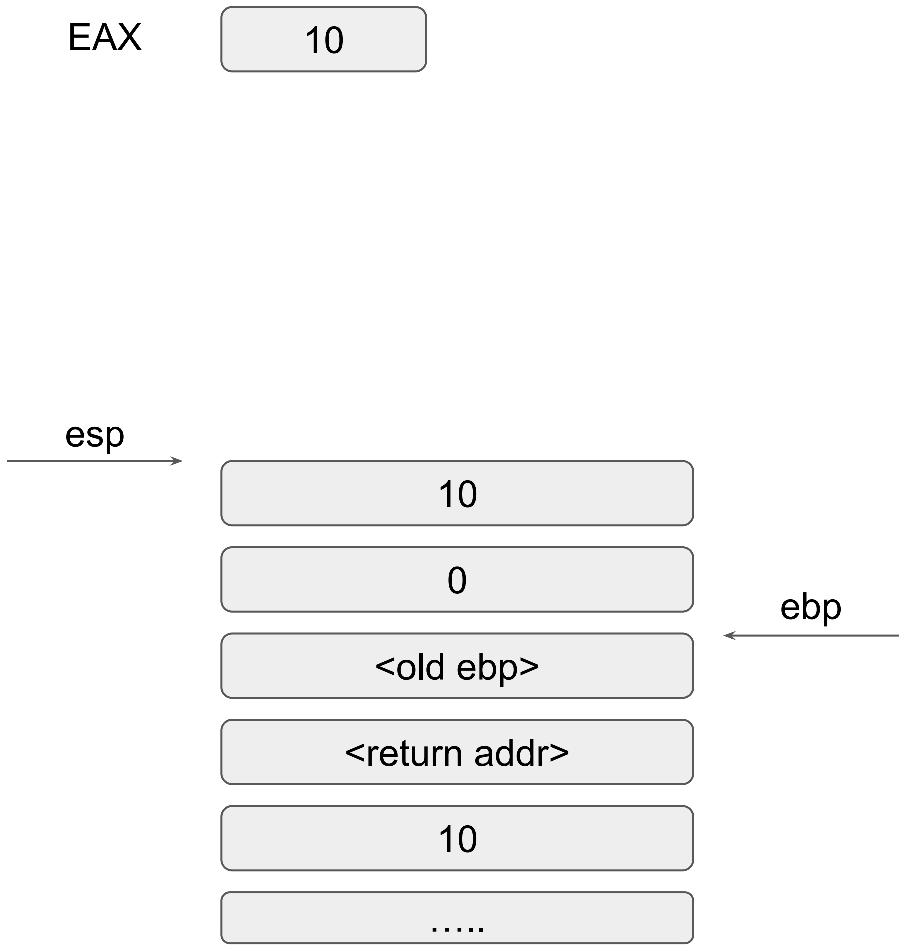
Stack frame

C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
   push ebp
   mov ebp, esp
   sub esp, 8
   mov DWORD PTR [ebp - 4], 0
   mov eax, DWORD PTR [ebp + 8]
→  mov DWORD PTR [ebp - 8], eax
```



EAX

10

esp

10

0

ebp

<old ebp>

<return addr>

10

.....

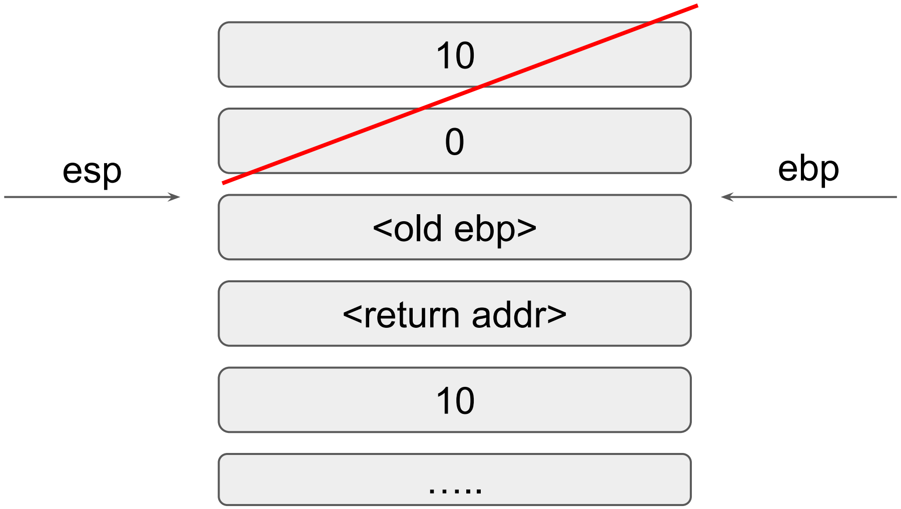
Stack frame

C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
     ...
→    mov esp, ebp  ┐
     pop ebp       ┘→ leave
     ret
```



Stack frame

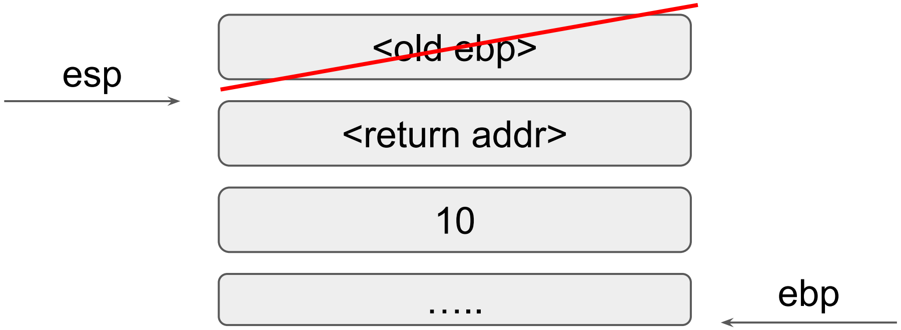
C:

```
int func (int x) {
     int a = 0;
     int b = x;
     ...
}
```

asm:

```
...
mov esp, ebp
pop ebp
ret
```

leave



esp

~~<old ebp>~~

<return addr>

10

.....

ebp

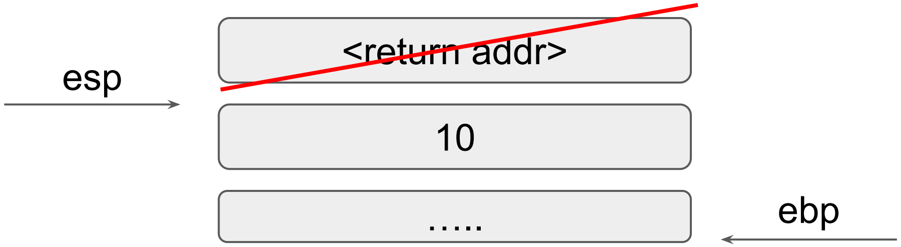
Stack frame

C:

```
int func (int x) {
    int a = 0;
    int b = x;
    ...
}
```

asm:

```
  ...
  mov esp, ebp
  pop ebp
→ ret
```



Disassembling a binary

Tools

Objdump:

```
$ objdump -M intel -D ./mybinary
```

gdb:

```
$ gdb ./mybinary
(gdb) set disassembly-flavor intel
(gdb) disassemble main
```

radare2:

```
$ r2 ./mybinary
r2> aaa
r2> pdf main
```

IDA:

GUI tool, not free neither cheap

Buffer overflow 101

Definition

- Ability to overrun a buffer's boundary and overwrites adjacent memory locations
- Often leads to DoS, remote code execution and / or privilege escalation

UNSAFE:

```
char mystr[50];
scanf("%s", mystr);

char input[50];
gets(input);


char in[50], buf[20];
scanf("%49s", in);
strcpy(buf, in);
```

SAFE:

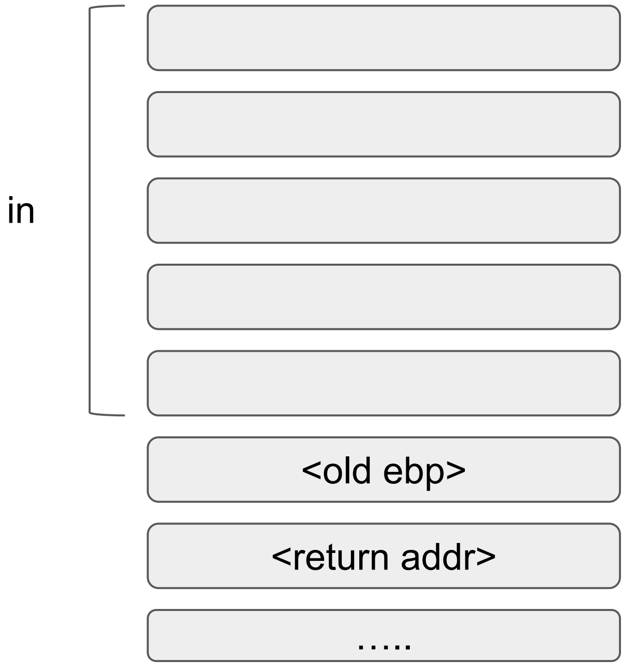
```
char mystr[50];
scanf("%49s", mystr);

char input[50];
fgets(input, sizeof(input),
stdin);

char in[50], buf[20];
scanf("%49s", in);
strncpy(buf, in, sizeof(buf));
```

Example

```
char in[20];
scanf("%s", mystr);
```

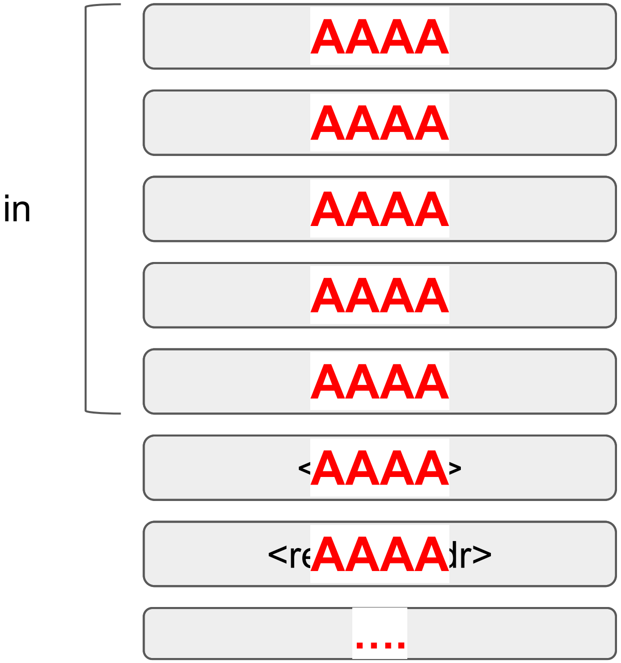


Example

```
char in[20];
scanf("%s", mystr);
```

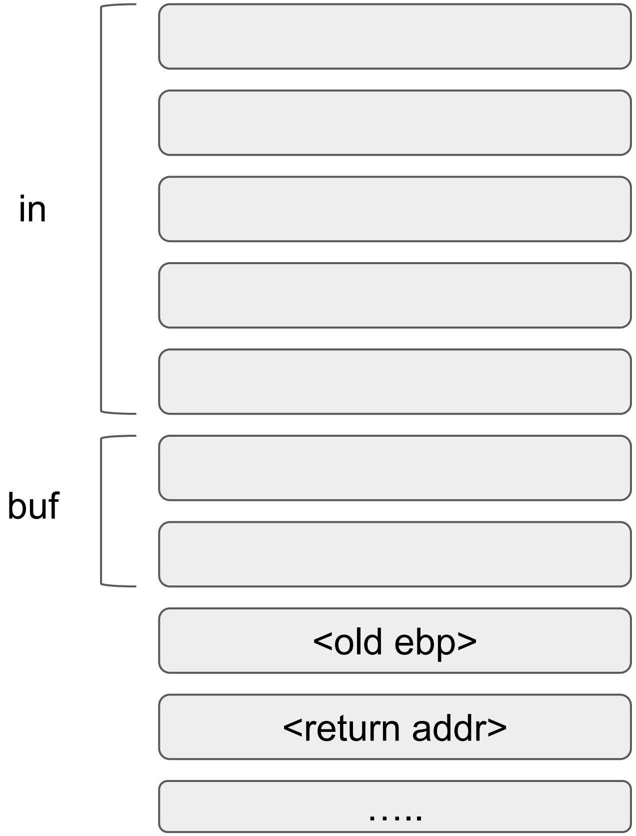
Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

= 30*A

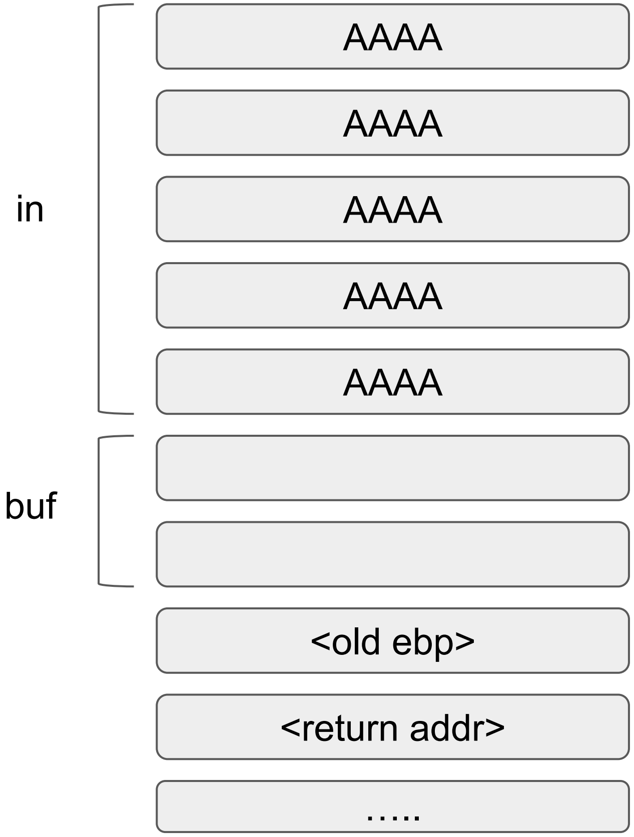


Example 2 - Indirect BOF

```
char in[20], buf[8];
scanf("%19s", in);
strcpy(buf, in);
```



Example 2 - Indirect BOF



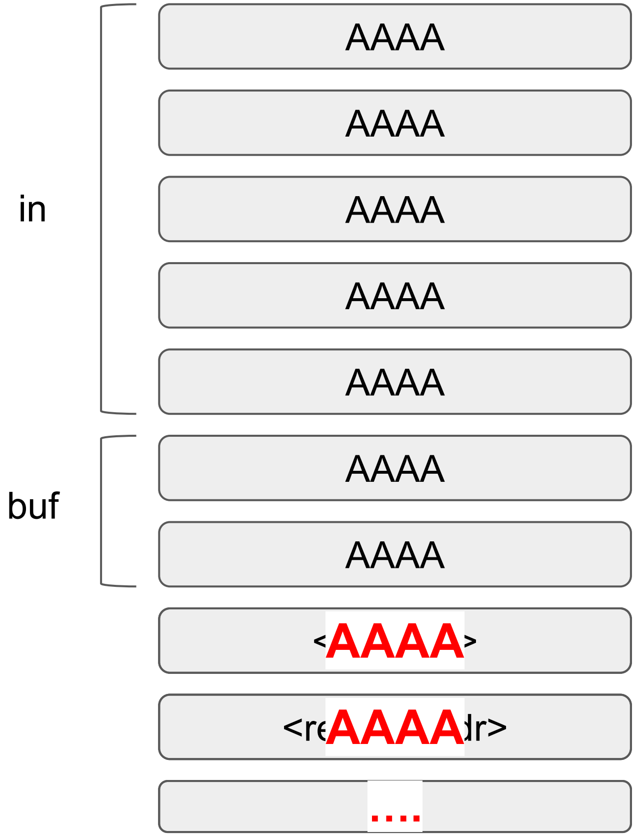
```
char in[20], buf[8];
scanf("%19s", in);
strcpy(buf, in);
```

Input: AAAAAAAAAAAAAAAAAAA

= 20*A

Example 2 - Indirect BOF

```
char in[20], buf[8];
scanf("%19s", in);
strcpy(buf, in);
```

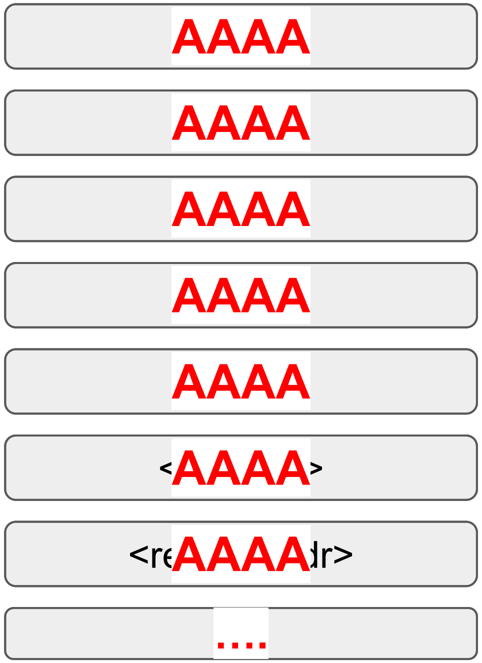


Buffer overflow 102

Now what?

Denial of Service

Arbitrary code execution ← shellcode



AAAA

AAAA

AAAA

AAAA

AAAA

<AAAA>

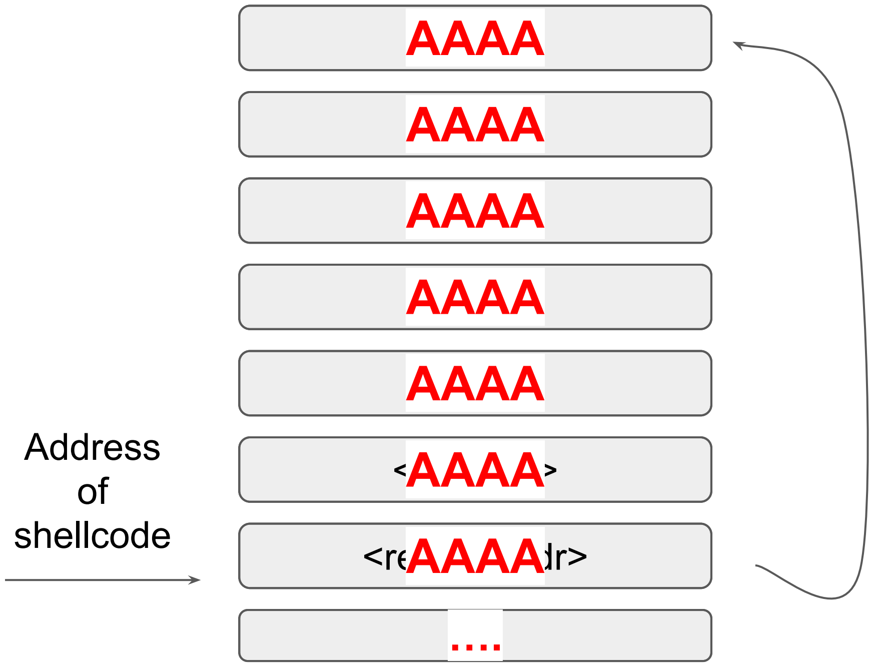
<reAAAAdr>

....

Now what?

Denial of Service

Arbitrary code execution ← shellcode



Shellcode

Sequences of bytes which represents assembled code.

Code: `execve("/bin/sh", ["/bin/sh"], NULL)`

Shellcode:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x8
0\x31\xc0\x40\xcd\x80
```

which corresponds to:

```
31 c0                   xor     %eax,%eax
50                      push    %eax
68 2f 2f 73 68          push    $0x68732f2f
68 2f 62 69 6e          push    $0x6e69622f
89 e3                   mov     %esp,%ebx
89 c1                   mov     %eax,%ecx
89 c2                   mov     %eax,%edx
b0 0b                   mov     $0xb,%al
cd 80                   int     $0x80
31 c0                   xor     %eax,%eax
40                      inc     %eax
cd 80                   int     $0x80
```

Should not contain byte \x00 (null byte). Why?

Mitigations

- ASLR (Address Space Layout Randomization)
- Stack canary
- DEP (Data Execution Prevention) and N^X stack

What’s next?

- Format Strings and informations leakages
- Return Oriented Programming
- Heap Exploitation

References

x86 Assembly guide - http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

Shellcodes - http://shell-storm.org/shellcode

radare2 cheat-sheet - https://github.com/zxgio/r2-cheatsheet/blob/master/r2-cheatsheet.pdf

Linux syscalls - https://syscalls.kernelgrok.com/

Live Overflow yt channel - https://www.youtube.com/playlist?list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN

Challenge’s writeups - https://ctftime.org/writeups

Changelog

27/02/19 - init

01/03/19 - added few more references

- fixed wrong shellcode with http://shell-storm.org/shellcode/files/shellcode-811.php
- Added stack canary mitigation