Memory corruption

REMIND: RCE Vulnerability

- **Remote Command Execution:**
  Attacker can execute **any action** from **remote**
- Only constraint: **privilege** level of vulnerable program

- **Any** action:
  - Word could start encrypting your disk
  - Powerpoint could launch a remote shell server
  - A web server could create a new user
  - ...

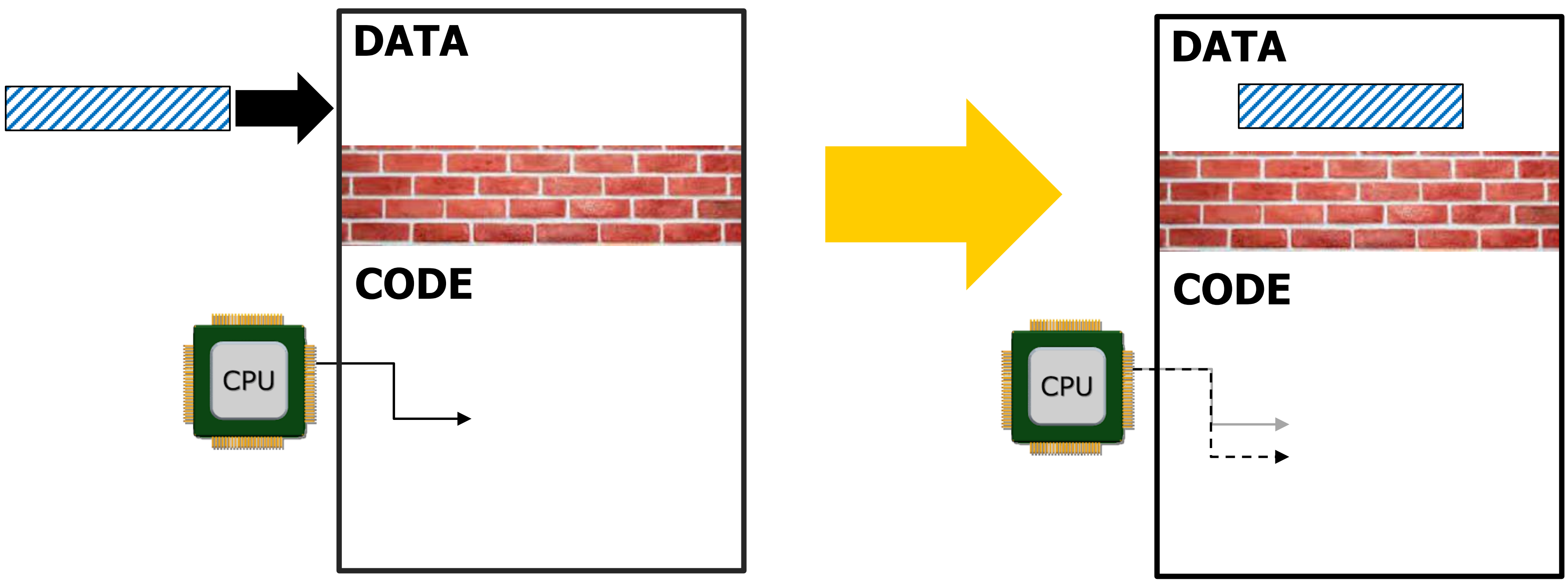
REMIND: RCE Vulnerability

- **Remote Command Execution:**
  Attacker can execute **any action** from **remote**
- Only constraint: **privilege** level of vulnerable program

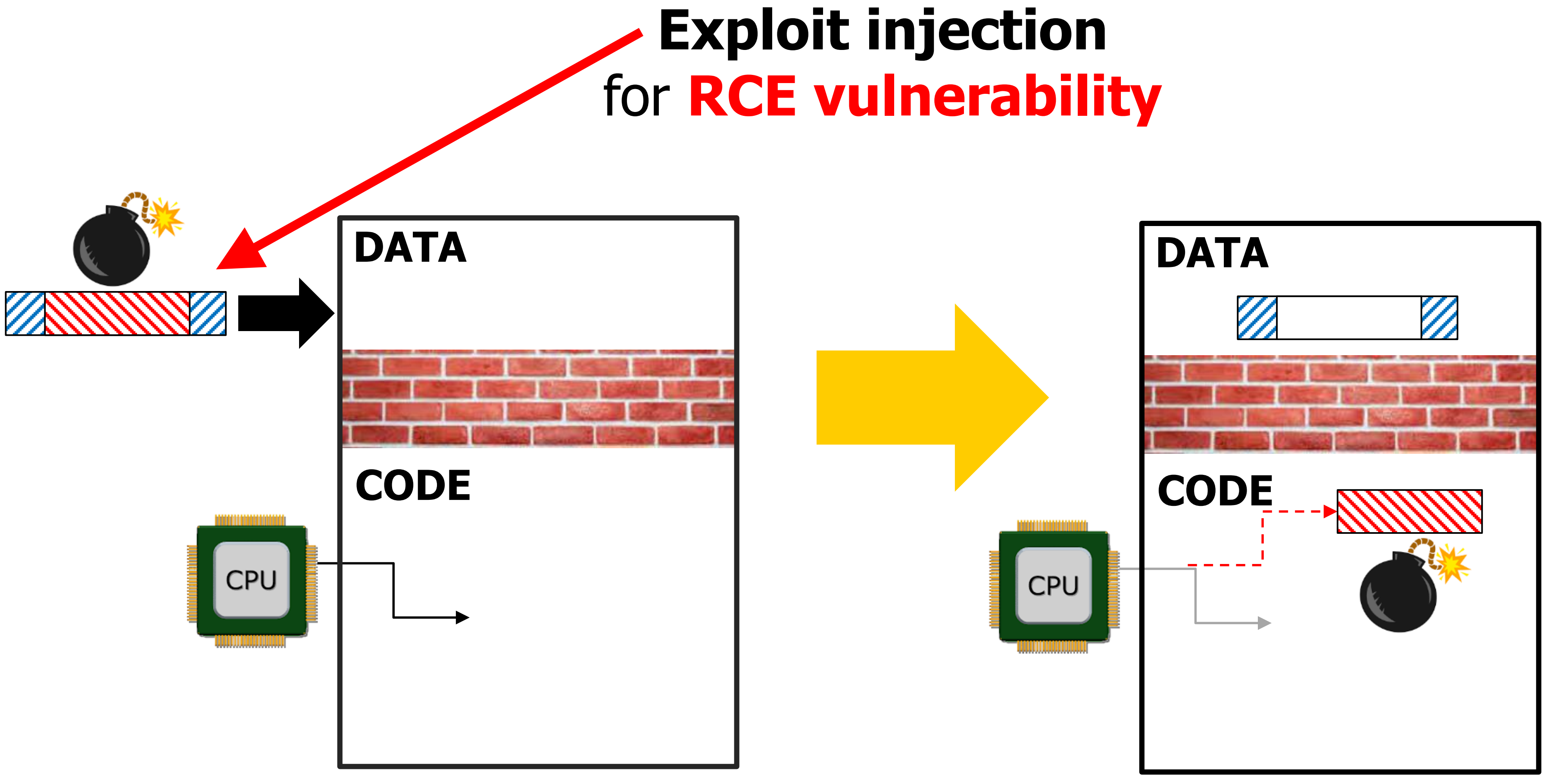
- Command injection
  - Adversary injects
    a string that will be executed as a shell command
- Memory corruption
  - Adversary injects
    a byte sequence that will be executed as CPU instructions

RCE – Memory Corruption (REMIND) (I)

What should **always** happen



RCE – Memory Corruption (REMIND) (II)



Credits

- **Part** of what follows in this file is based on material from:
  - Computer Security Course – Berkeley CS161
  - Software Security Course – Radboud University

- Any possible mistakes/inaccuracies are mine

Memory Corruption

- Memory corruption bug: program accesses memory "incorrectly"
- Most common example: Out-of-bounds write (**buffer overflow**)

- Memory corruption bugs can be exploited by attackers (**vulnerability**)
  - Alter program behavior
  - Take **full control** of program

- One of the **oldest problems** in computer security

Memory Corruption vs Memory Safety

- Memory corruption bug: program accesses memory "incorrectly"
- Terminology that is becoming prevalent: memory **safety**
  - Memory **safe** language
    (language that prevents memory corruption bugs)
  - Memory ~~corruption~~ **safety** bug
    (bug that occurs because the memory is non accessed safely)
  - ...
- Basically the very same concepts
- I will tend to use:
  - Memory corruption bug
  - Memory safe (or memory unsafe) language

Basic Fact

- To make a long story short:
  - **More than 70%** of vulnerabilities are caused by **memory corruption**
  - Statistics and estimates
  - All platforms
  - Similar stats for 0-days (i.e., **exploited in the wild**)
- Oversimplified for ease of description

Depressing fact

The CWE Top 25

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0

Key reason (and current trend)

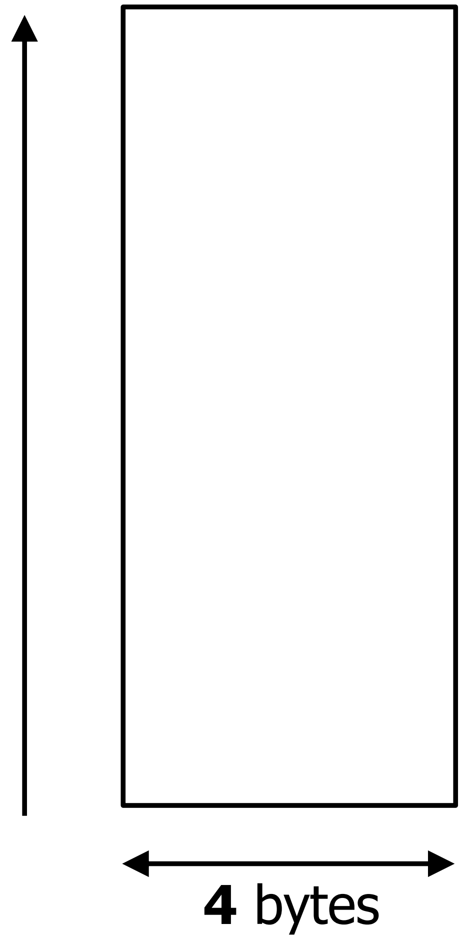
- Most memory corruption vulns are a consequence of usage of **memory unsafe programming languages** (C / C++)

- Strong trend toward abandoning memory unsafe languages and **switch to memory safe** languages
(Rust, Go, Python, Java)

- "Android 13 is the first Android release where a majority of new code added to the release is in a memory safe language (Rust)”
- “There have been zero memory safety vulnerabilities discovered in Android’s Rust code."

Memory Management (in a nutshell) Part 1

Address Space: Our drawings



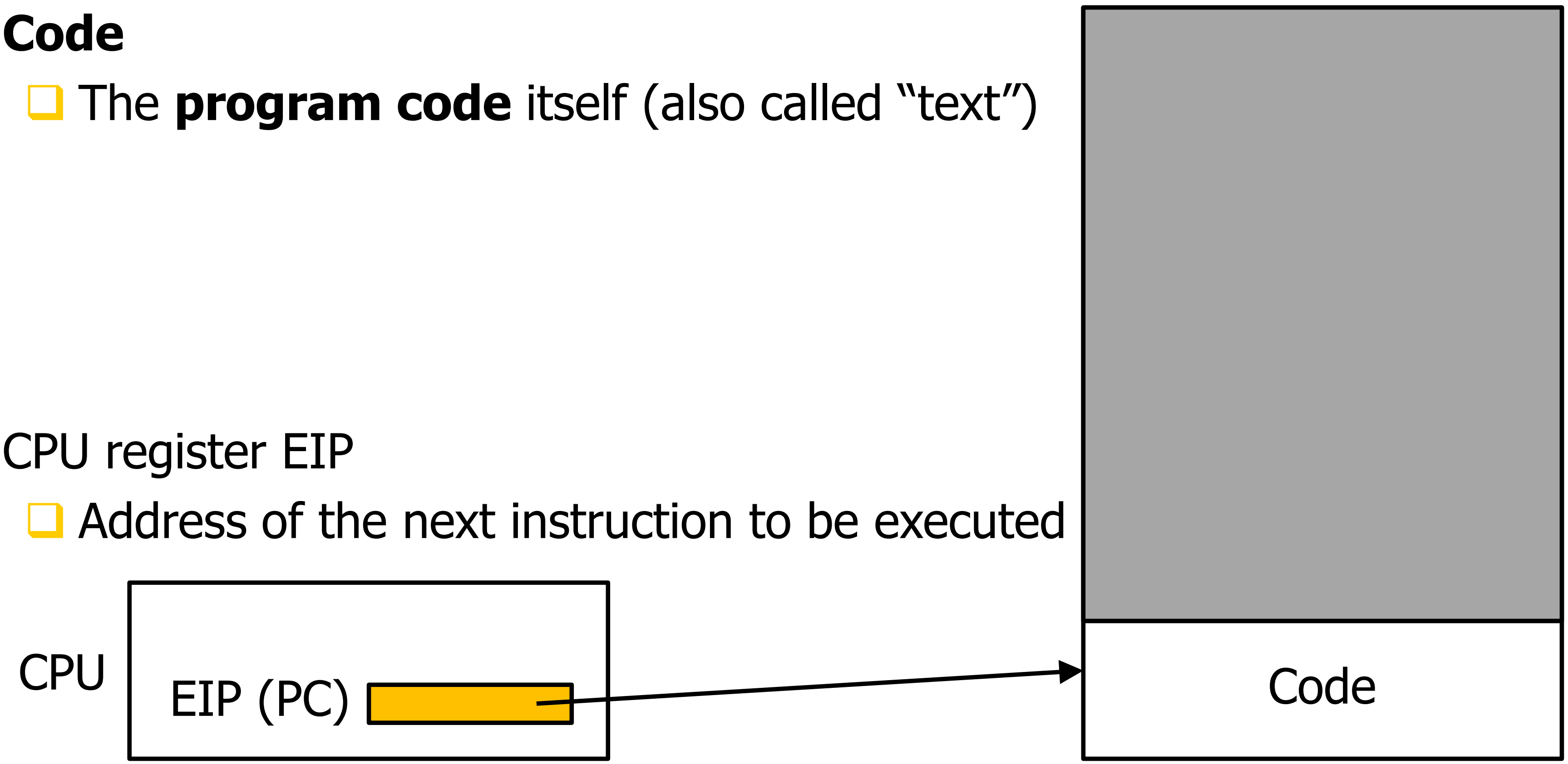
0x FFFFFFFF FFFFFFFF
0x FFFFFFFF FFFFFFFC
0x FFFFFFFF FFFFFFF8
0x FFFFFFFF FFFFFFF4

64 bit addresses
(8 bytes)

0x 00000000 0000000C
0x 00000000 00000008
0x 00000000 00000004
0x 00000000 00000000

Code

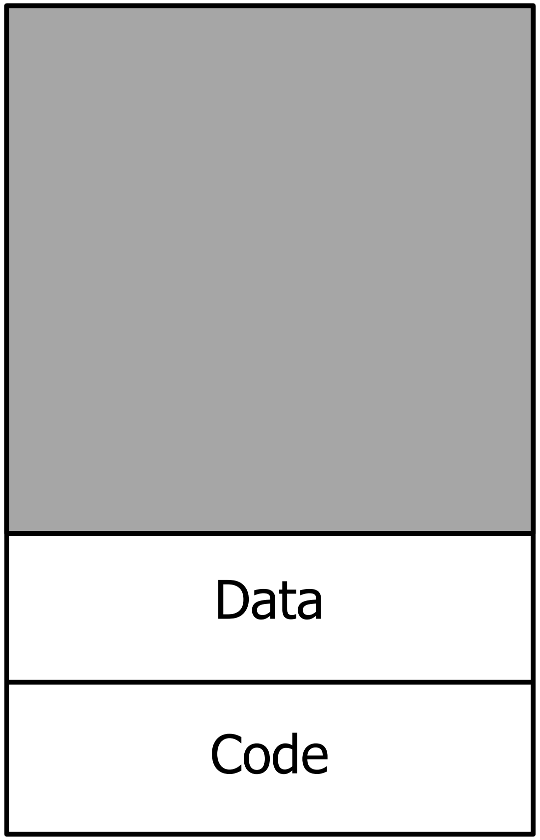
- **Code**
  - The **program code** itself (also called “text”)
- CPU register EIP
  - Address of the next instruction to be executed



Extended **Instruction Pointer**
(Program Counter)

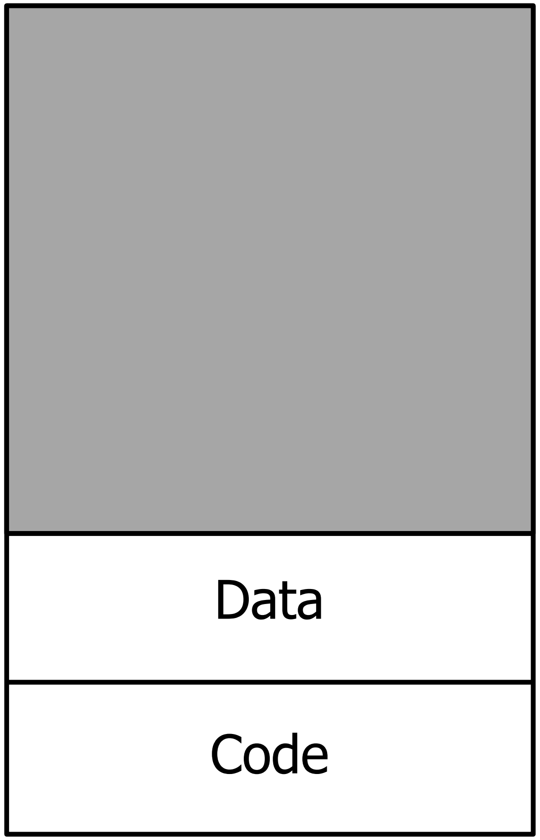
Data

- Code
- **Data**
  - **Static variables**: those that exist for the **entire lifetime** of the program
  - Allocated when the program is started



Remark

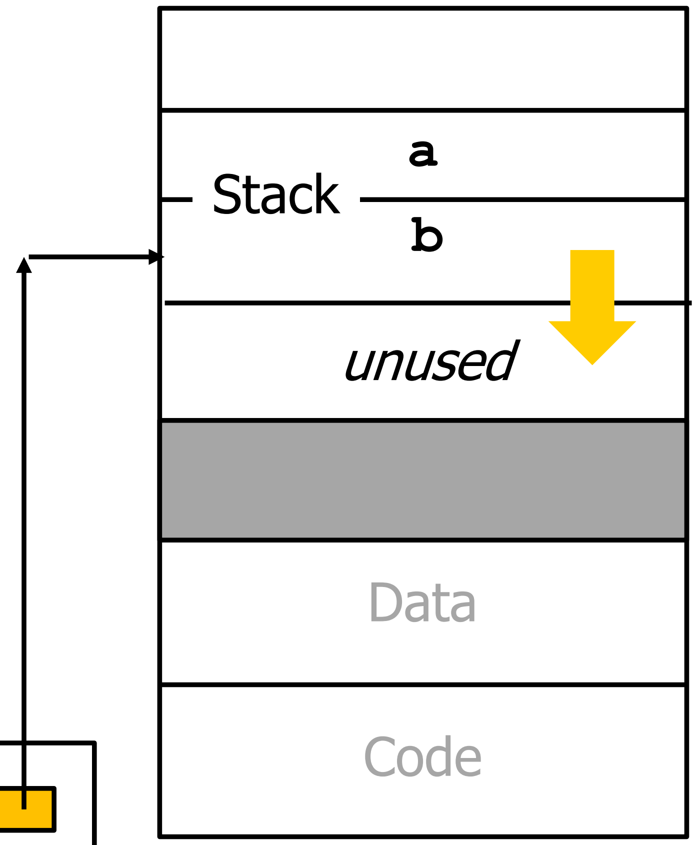
- Starting address of Code is **not** 0
- It is chosen by the O.S. when launching the program



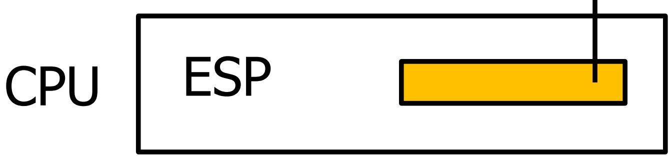
Stack

```
void func(void) {
    int a;
    int b;
    ...
}
```

- **Stack**:
  - **Local variables** and …
  - As you make
    deeper and deeper function calls,
    it grows **downwards**



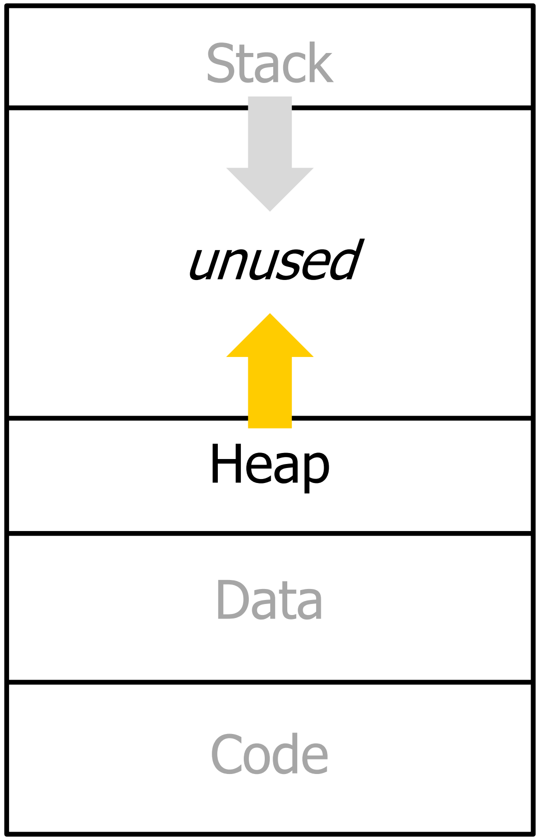
- CPU register ESP
  - Address of the top of the stack



Extended **Stack Pointer**

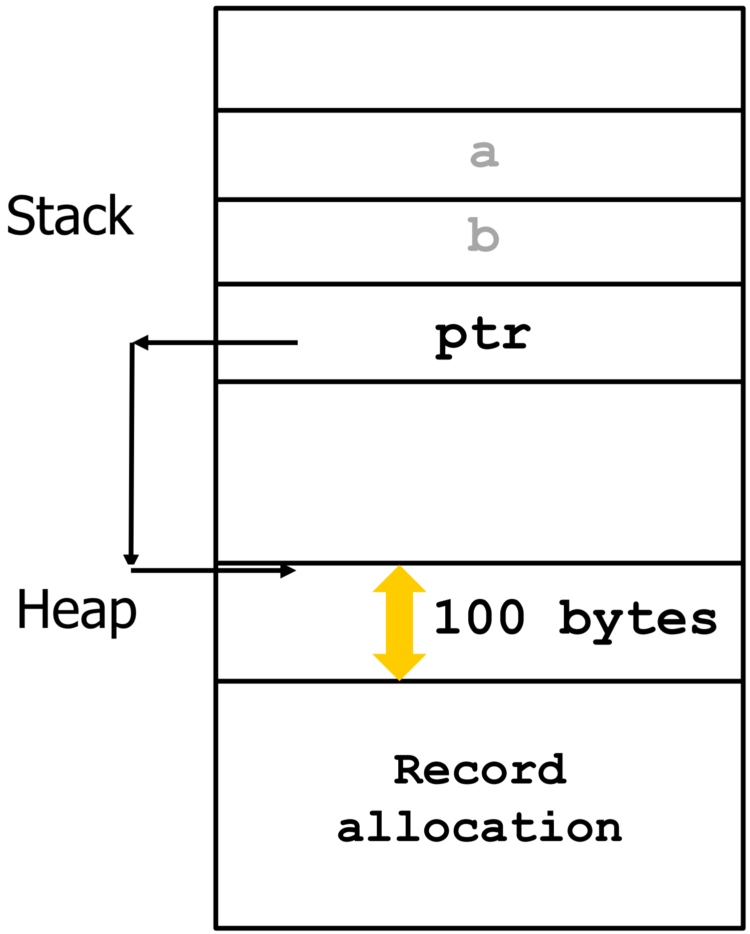
Heap

- Code
- Data
- Stack *(later)*
- **Heap**
  - **Dynamically** allocated memory
    - C language: **malloc** and **free**
    - "Modern languages" = "Objects"
  - As more and more memory is allocated, it grows **upwards**



Heap vs Stack

```
void func(void) {
    int a;
    int b;
    int *ptr;
    ...
    ptr = malloc(100);
    ...
}
```



Memory Management (in a nutshell of a nutshell)

- The address space of a running program is subdivided in **4 regions**

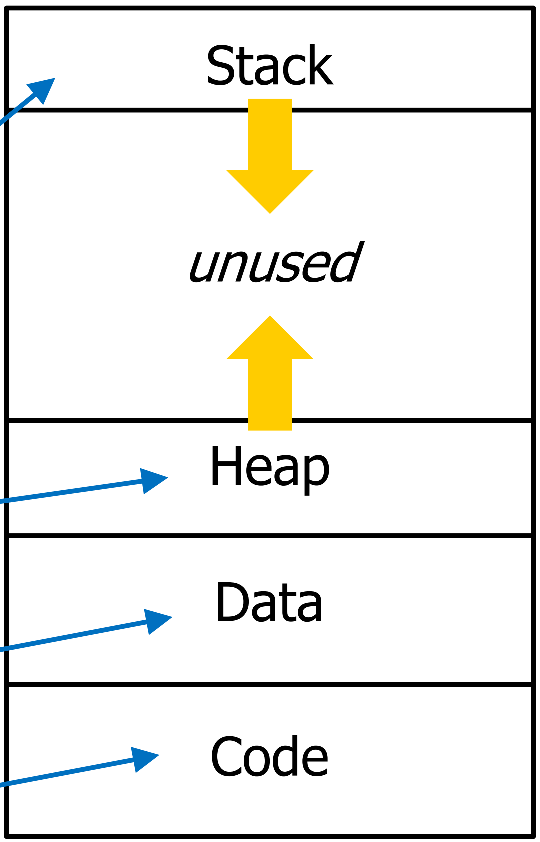
- First (and very rough) approximation:

Local variables + Function arguments

Dynamically created variables ("objects")

Global variables

Program instructions



Memory Corruption Vulnerabilities

Memory Corruption BUG: Example (I)

```
...
char name[4];
...
void function(...) {
      ...
    name[4]='a' // bug: array indexes start from 0
    ...
}
```

- Overwrites a memory region in the **data** area

Memory Corruption BUG: Example (II)

```
void function(...) {
    char name[4];
    ...
    name[4]='a' // bug: array indexes start from 0
    ...
}
```

- Overwrites a memory region in the **stack**

Memory Corruption BUG: Example (III)

```
void function(...) {
    char *name = malloc(4);
    ...
    name[4]='a' // bug: array indexes start from 0
    ...
}
```

- Overwrites a memory region in the **heap**

Bugs vs Vulns

Bugs Error or flaw that causes the sw
- to produce an **incorrect** result, or
- to behave in **unintended ways**.

Vulnerabilities

- ...with **violation** of some **security** property

Vulns vs Exploitable Vulns

Bugs

Error or flaw that causes the sw

- to produce an **incorrect** result, or
- to behave in **unintended ways**.

Vulnerabilities

- ...with **violation** of some **security** property

Exploitable Vuln.

Next slides

- Hypothetical examples (for ease of description)
- Memory corruption **bug** that is also a **vulnerability**
  - Provoked by (now deprecated) library function `gets()`

- **Same bug, different impacts**
  - Access control bypass
  - Configuration change (access control bypass)
  - Command injection

gets() (I)

NAME top

gets - get a string from standard input (DEPRECATED)

SYNOPSIS top

```
#include <stdio.h>

char *gets(char *s);
```

DESCRIPTION top

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with a null byte ('\0').

gets() (II)

NAME top

gets - get a string from standard input (DEPRECATED)

SYNOPSIS top

```
#include <stdio.h>

char *gets(char *s);
```

DESCRIPTION top

Never use this function.

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

- ❑ Library function **deprecated**
- ❑ We use it **only** for illustratory purposes

Hypothetical Example 1

```
...
int authenticated=0;
...
// Complex program
// Access Control based on the value of authenticated
// (set to 1 only upon successful authentication)
...
```

Access Control Bypass

```
...
char name[20];
int authenticated=0;
...
void vulnerable {
    ...
    gets(name);   // reads from input until '\n'
    ...
}
```

- IF input contains more than 20 bytes before `'\n'`
- THEN input overwrites **`authenticated`** with **arbitrary value chosen from the outside**

Hypothetical Example 2

```
...
char dns_address="8.8.8.8";
...
int setConfiguration (...) {
    ...
    // write dns_address in IP configuration
}
...
```

Configuration Change (Access Control bypass)

```
char name[20];
char dns_address="8.8.8.8";
...
int setConfiguration(...) { ... /* use dns_ */ ... }
..
void vulnerable {
    ...
    gets(name);    // reads from input until '\n'
    ...
}
```

- IF input contains more than 20 bytes before `'\n'`
- THEN input overwrites `dns_address` with **arbitrary value chosen from the outside**

Hypothetical Example 3

```
...
char cmd="/usr/bin/ls";
...
int someFunc(...) {
    ...
    execve(cmd);  // execute program (replace
                  //  code, data and clear heap)
}
...
```

Command Injection

```
char name[20];
char cmd="/usr/bin/ls";
...
int someFunc(...) { ... /* use cmd */ ... }
..
void vulnerable {
    ...
    gets(name);    // reads from input until '\n'
    ...
}
```

- IF input contains more than 20 bytes before `'\n'`
- THEN input overwrites `cmd` with **arbitrary value chosen from the outside**

Remark: Impact

- Three different possible impacts for the **same** bug
  - Access control bypass
  - Configuration change (access control bypass)
  - Command injection
- Impact:
  - Depend on **program structure** and **vulnerability**
  - **Not** chosen by the attacker arbitrarily

Remark: Exploitability

```
char name[20];
...
...
...
...
...
char cmd="/usr/bin/ls";
```

- Exploit injection may overwrite many variables in addition to the one of interest
- Program behavior must remain useful to the attacker

↓

- Vulnerability may or may **not** be "practically exploitable"

Useful point of view (I)

- Exploitation based on **overwrite**:
  - Write **attacker-controlled VALUE** at **attacker-controlled LOCATION**

- Access control bypass
  - Alter control flow (if-then-else)
- Configuration change (access control bypass)
  - Alter configuration
- Command injection
  - Alter invocation parameters

Useful point of view (II)

- Overflow on data region $\Rightarrow$ Overwrite on data region
- May or may not be exploited

- We will see that:
  - Overflow on a region (data/stack/heap) may overwrite a **different** region
  - Overflow on **stack** region may **certainly** be exploited
    - This is why there are compiler / o.s. / hw defenses for preventing exploitation

Hhmmm...

- How can the Adversary determine that there is a **bug**?
- ...that is also a **vulnerability**?

- How can the Adversary determine the **impact**?
- How can the Adversary determine how the **exploit** must be structured?

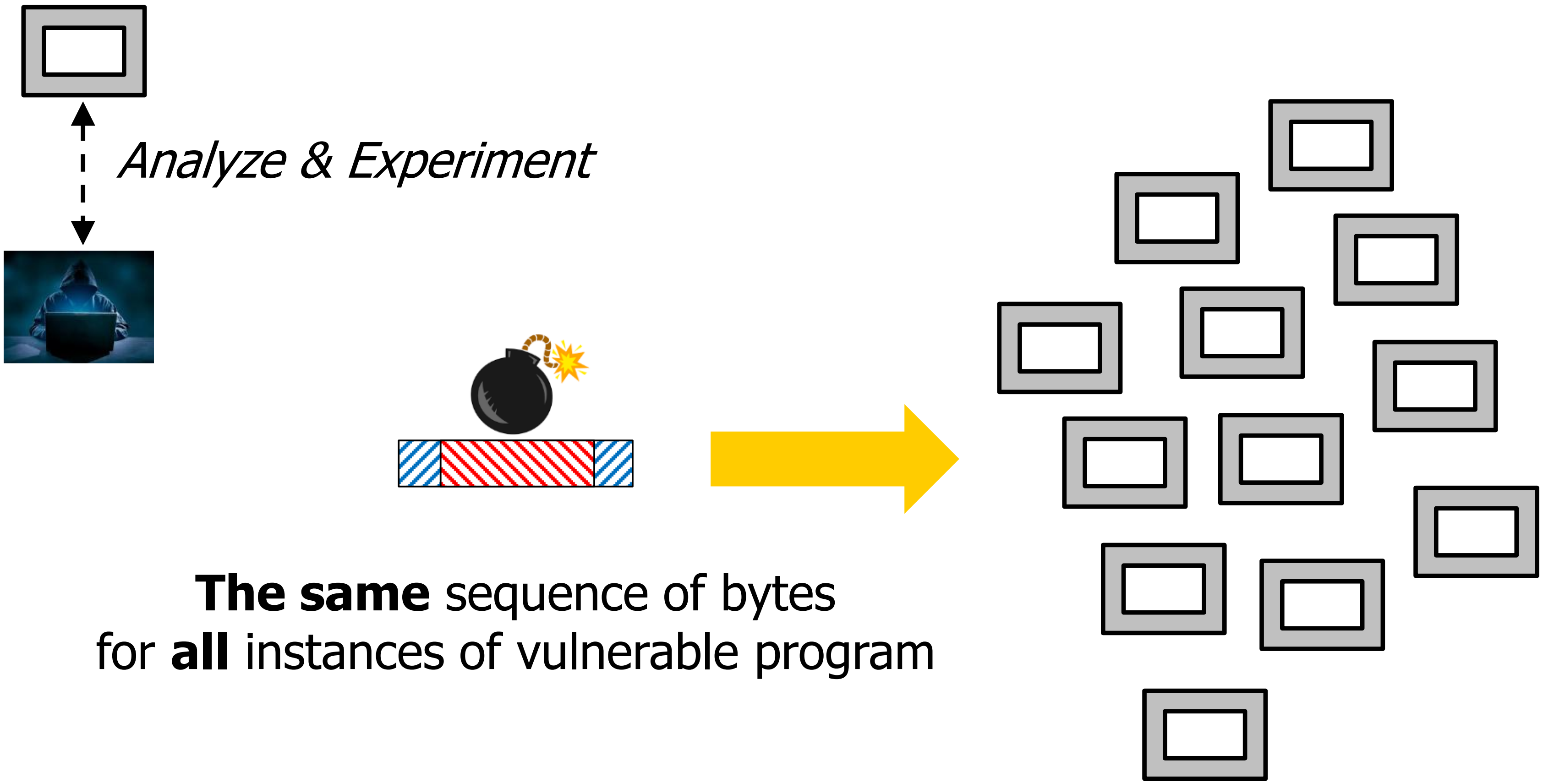
Threat Model: Vulnerability DISCOVERY

- Attacker:
  1. Has target code locally available
  2. Can reconstruct source
     - Open source
     - Decompiler / Disassembler
     - ...

- Source not necessarily identical to the original one...
  ...but in a form "sufficient" for reasoning about its behavior

- Vulnerability discovery: not exploitation

Exploit Development



Buffer Overflow

Buffer Overflow

- Very important and common memory corruption bug:
  - **Write past the end** (or **before the beginning**) of the intended buffer
- All the previous examples are vulnerabilities resulting from:
  - **Input** operations (`gets()` does not check size of destination buffer)

Safe INPUT Libraries

- **Input** operations
  (`gets()` does not check size of destination buffer)
- They **never overflow destination buffer**
  - Files:
    ~~`char gets(char *str);`~~
    `char *fgets(char *str, int n, FILE *stream)`
  - Sockets:
    `size_t recv(int sockfd, void *buf, size_t len,...);`
  - ...

Very Optimistic Assumption

- Files:
  `char *fgets(char *str, int n, FILE *stream)`
- Sockets:
  `size_t recv(int sockfd, void *buf, size_t len,...);`
- ...

- We only use:
  - Input libraries that **never overflow destination buffer**

- Are we safe from buffer overflows?
- Spoiler: no

Fact #1

- Every program has some variables whose values derive from (part of) some input
- Dependency chain potentially "very long"

```
...
char* base_url = ... // Obtained from (part of) input
...
char* full_url = base_url;
...
memcpy(buf, full_url, 12);
...
func(a, full_url);
...
sendto(sock, buf,...);
...
```

Fact #2

- Every input could be provided by an Adversary
  - Adversary may control the value of **many** variables indirectly
  - Even **"far away" from input operations**

```
...
char* base_url = ... // Obtained from (part of) input
...
char* full_url = base_url;
...
memcpy(buf, full_url, 12);
...
func(a, full_url);
...
sendto(sock, buf,...);
...
```

Keep in mind

- Every program has some variables that may be controlled by an Adversary
  - Even "far away" from input operations
  - Dependency chain may be long and complex

- True in every programming language

- Not a problem in a correct program

Why safe input libraries might not be enough (I)

- Code writes in memory buffer `b`
- Number of bytes / offset controlled by variable `var` (very "far away" from input operations)
- `var` is controlled by Adversary

- Code might write **outside** of `b`

Why safe input libraries might not be enough (II)

- Code writes in memory buffer `b`
- Number of bytes / offset controlled by variable `var` (very "far away" from input operations)
- `var` is controlled by Adversary

- Very common source of bugs/vulns:
  - **String manipulation**

Example

- A string is a sequence of chars **terminated** by a NULL byte (`'\0'`)

```
#define MAX_BUF 256
...
char dst[MAX_BUF];
...
// src attacker-controlled
strcpy(dst, src);    // what if strlen(src)> MAX_BUF?
...
```

- Attacker may provide input such that `strlen(src) > MAX_BUF`
  ⇒ **buffer overflow** when writing on `dst`
  ⇒ overwrite something useful

Good practice: Prevent overflowing `dst`

```
#define MAX_BUF 256
...
char dst[MAX_BUF];
...
short len = strlen(src);
if (len < MAX_BUF) strcpy(dst, src);
...
}
```

Not easy!

```
#define MAX_BUF 256
...
char dst[MAX_BUF];
...
short len = strlen(src);
if (len < MAX_BUF) strcpy(dst, src);
...
}
```

- Bug: `short` assigned to `int`

`strlen(src) > 32K`

⇒ `len` takes a negative value (**integer overflow)**

⇒ `if` condition satisfied

⇒ **buffer overflow** when writing on `dst`

⇒ overwrite something useful

Safe STRING Libraries

~~`char *strcpy(char *dest, const char *src);`~~

```
char *strncpy(char *dest, const char *src, size_t n);
```

- Copies up to `n` chars from source to destination
- By making sure `n` = size of destination **we never overflow the destination**

Even more
Very Optimistic Assumption

- We only use:
  - **Input** libraries that never overflow destination buffer
  - **String** libraries that **never overflow destination string**

- Are we safe from buffer overflows?
- Spoiler: no

Great!

```
#define MAX_BUF 256
...
char dst[MAX_BUF];
...
// src attacker-controlled
strncpy(dst, src, MAX_BUF);
...
```

No overflow on `dst`!

Let's continue...

```
#define MAX_BUF 256
...
char dst[MAX_BUF];
...
// src attacker-controlled
strncpy(dst, src, MAX_BUF);
...
#define LEN 512
char path[LEN];
...
memcpy(path, dst, strlen(dst))
...
```

`MAX_BUF < LEN` ⇒ no overflow on `path`

Opss...

```
#define MAX_BUF 256
...
char dst[MAX_BUF];
...
// src attacker-controlled
strncpy(dst, src, MAX_BUF);
...
#define LEN 512
char path[LEN];
...
memcpy(path, dst, strlen(dst))
...
```

Adversary may provoke `dst` not NULL-terminated
⇒ `strlen(dst) > MAX_BUF`

Maybe `strlen(dst) > LEN`
(it depends on where a `\0` byte will be found)
⇒ overflow on `path`

Other impact: Information Disclosure

```
#define MAX_BUF 256
...
char dst[MAX_BUF];
...
// src attacker-controlled
strncpy(dst, src, MAX_BUF);
...
send(socket1, dst, strlen(dst));
...
```

Adversary may provoke `dst` not NULL-terminated
⇒ `strlen(dst) > MAX_BUF`

Out of bound read from `dst` (until finding a `\0` byte)

Safer STRING libraries

size_t
strlcpy(*char *dst, const char *src, size_t size*);

size_t
strlcat(*char *dst, const char *src, size_t size*);

- They are designed to be safer, more consistent, and less error prone replacements for strncpy(3) and strncat(3).
- These **guarantee** to **NULL-terminate the result**.

- Note that a byte for the NUL should be included in *size*.
- Also note that for **strlcpy**() *src* must be NUL-terminated and for **strlcat**() both *src* and *dst* must be NUL-terminated.

Even more Even more
Very Optimistic Assumption

- We only use:
  - **Input** libraries that never overflow destination buffer
  - **String** libraries that **never overflow destination string**
  - **String** libraries **that always terminate destination string**

- Are we safe from buffer overflows?
- Spoiler: no

Real Example (ListServ 2024)

```
smtpHostname = getenv("SMTP_HOSTNAME");
if ( smtpHostname )
{
  if ( !*smtpHostname )
    smtpHostname = name;
}
else
{
  smtpHostname = name;
}
sprintf(dest, "HELO %s\n", smtpHostname);
```

- Environment variable `SMTP_HOSTNAME` may be **longer** than `dest`
  $\Rightarrow$ overflow

Another Example

```
...
int* ptr = malloc(1000);
...
// idx is attacker-controlled
ptr[idx] = val;
...
```

- Buffer overflow:
  - Write `val` (which may or may not be Adversary-controlled)
  - Outside of the allocated buffer
- It may or may not be a vulnerability

Real Example (I)

Talos Vulnerability Report

TALOS-2023-1734

Microsoft Office Excel WebCharts out-of-bounds write vulnerability

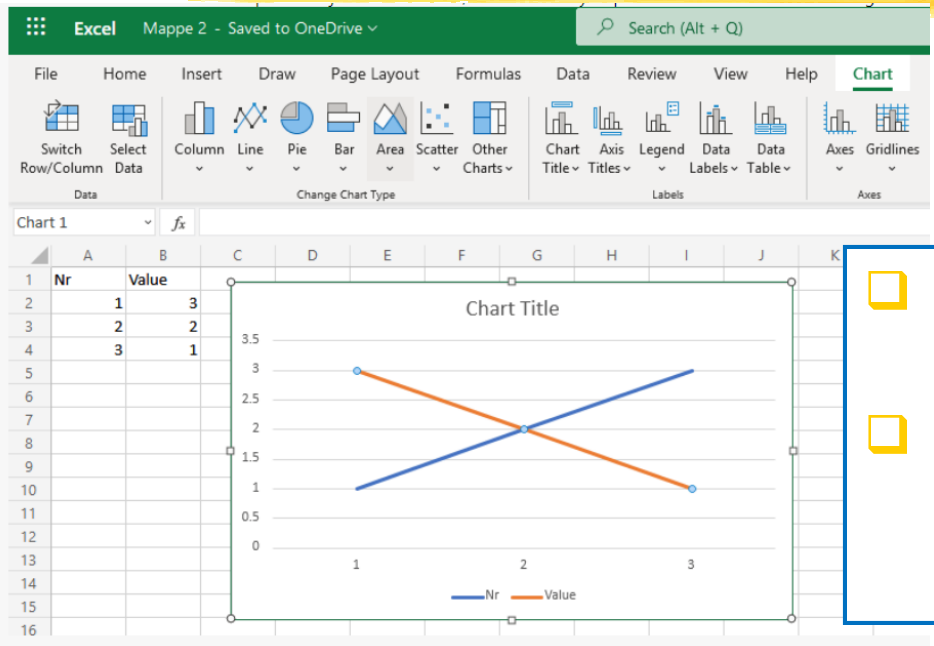
JUNE 13, 2023

CVE NUMBER

CVE-2023-33133

An access violation vulnerability exists in the WebCharts functionality of Microsoft Office Excel 2019 Plus version 2302 build 16130.20332. A specially crafted malformed file can lead to a heap buffer overflow. An attacker can use arbitrary code execution to trigger this vulnerability.

Real Example (II)



- Each web chart stored in an element of `webCharts` array
- This is contained in a dynamically allocated memory starting at address `rootXML`

```
memset(&rootXML->webCharts[156 * index], 0, 156u);
```

Real Example (III)

```
int __fastcall HrInitCHISD2(PBYTE a1, _DWORD *a2)
{

  rootXML = a1 + 17944;
  if ( FHpAllocCore((Mso::Memory *)0x144B0, rootXML, (void **)1) )
    (...)
```

The allocation is made for a `FIXED` size memory `0x144B0`. This means that the `webCharts` array always has slots for a max of `14 elements`. [redacted]

Our testcase contains 0x4d (77) embedded WebCharts elements, which significantly exceeds the acceptable amount. An attacker controlling the amount of `WebChart` elements might control the index for the `webCharts` table, and in that way achieve a precise out-of-bounds write primitive. With a proper heap grooming, this might lead to arbitrary write and finally to arbitrary code execution.

Takeaway 1

- Making sure that "**input never overflows**" is **not** enough for preventing buffer overflows

- **Overflows may happen potentially anywhere**
- **Many** other risky operations
  - String processing
  - ...many others

Takeaway 2

- Unsafe libraries can and should be replaced, but:

1. **Not** a **complete** solution
    1. Memory safety bugs may be in "our code" (not only in libraries)

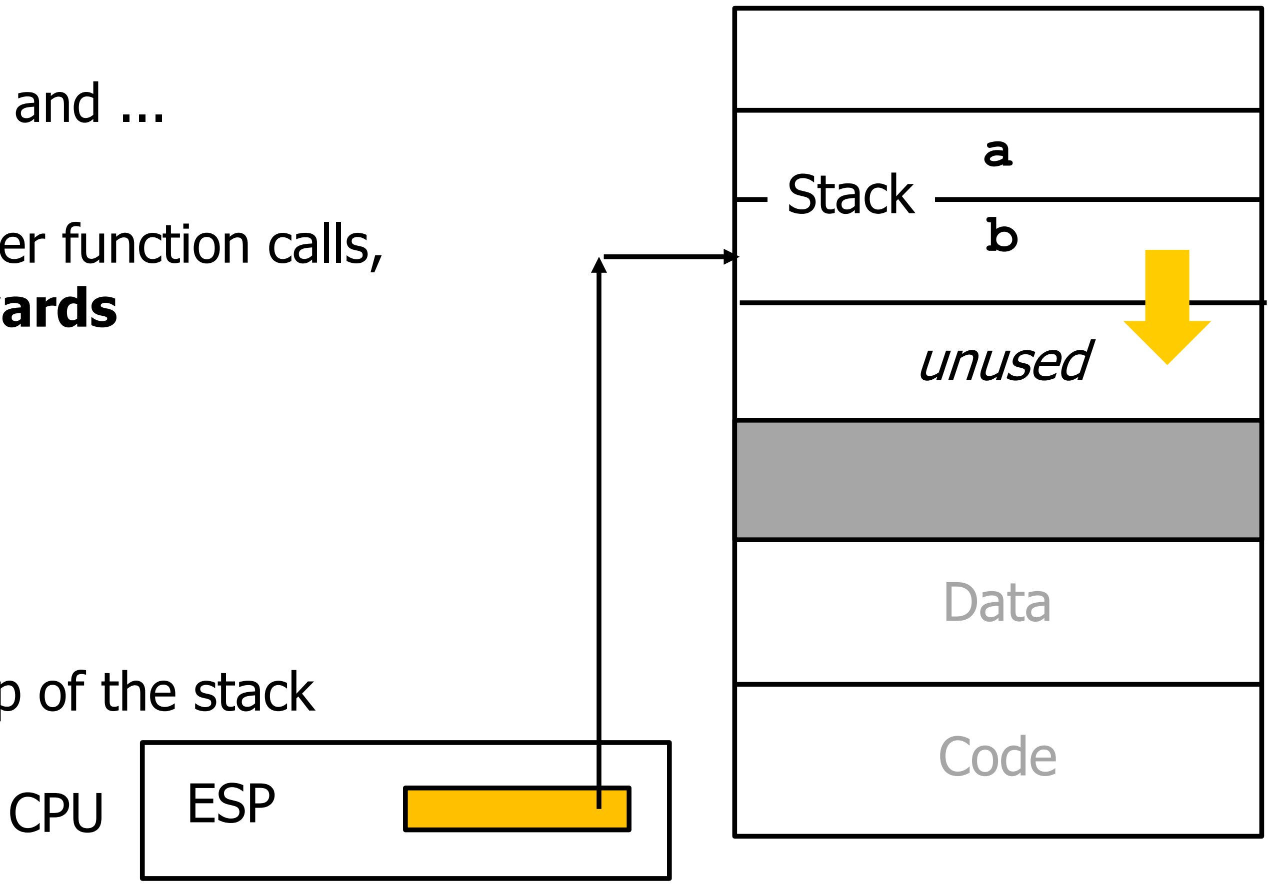
2. **LOT** of **existing** code to be modified, tested and shipped
    - Never forget economics:
        - Who **knows** about these problems?
        - Who has sufficient **incentives** to fix them?

Memory Management (in a nutshell) Part 2

Stack (REMIND)

```
void func(void) {
     int a;
     int b;
     ...
}
```

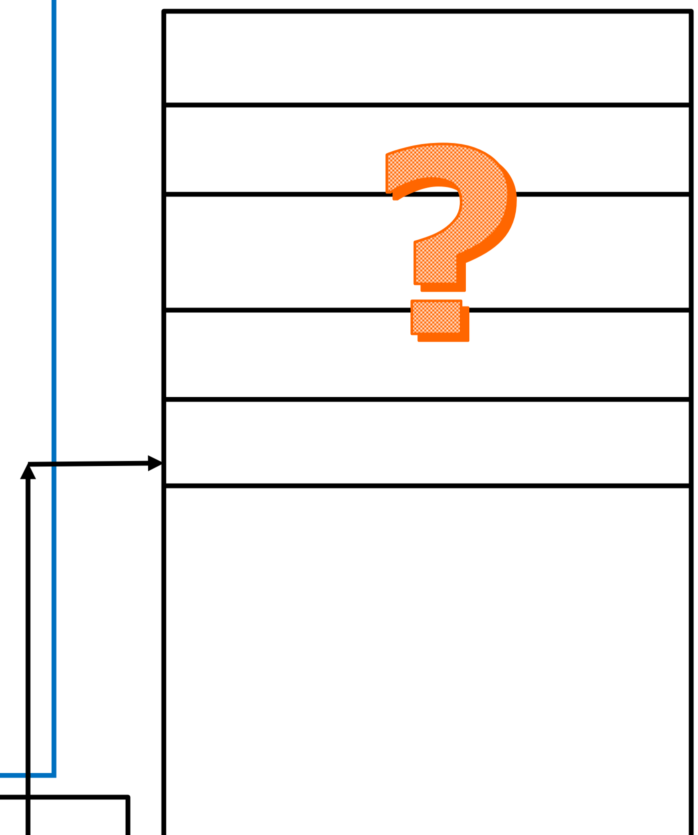
- **Stack**:
  - **Local variables** and ...
  - As you make
    deeper and deeper function calls,
    it grows **downwards**
- CPU register ESP
  - Address of the top of the stack

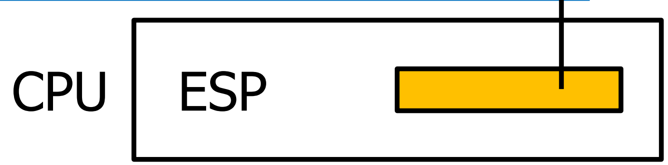


Extended **Stack Pointer**

What is on the Stack?

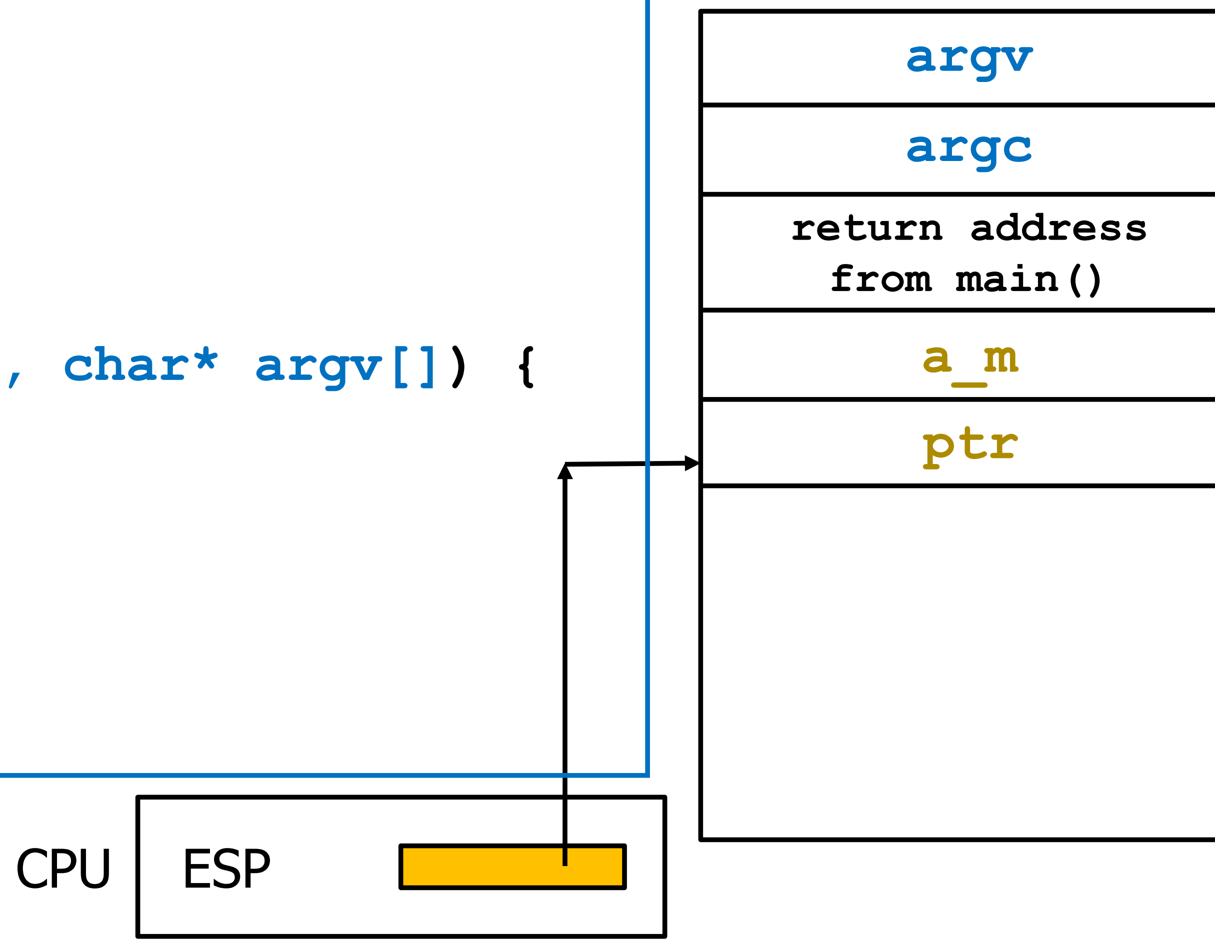
```
void func(int x) {
    int a;
    int b;
    ...
}
void main(int argc, char* argv[]) {
    int a_m;
    int *ptr;
    ...
    func(a_m);
}
```





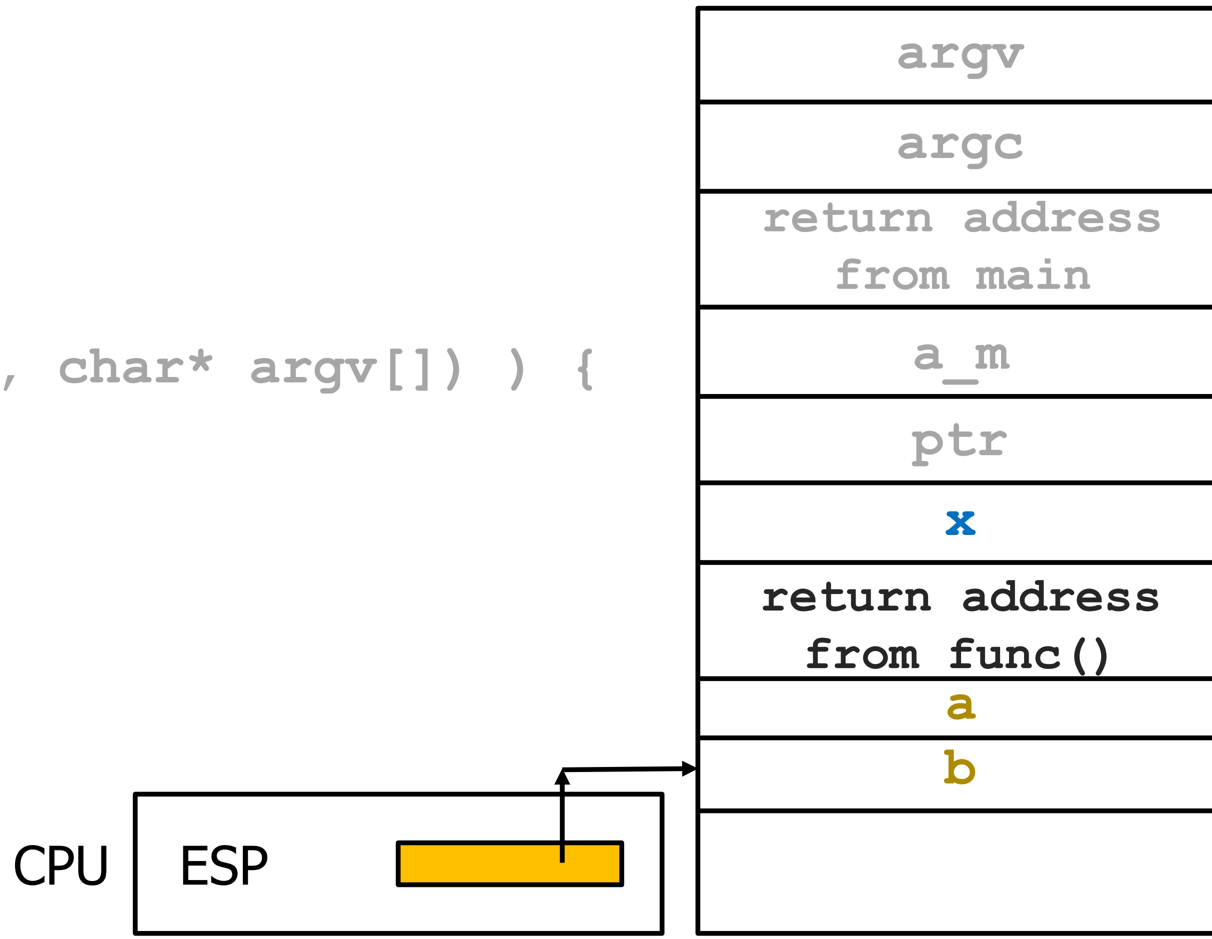
Input Arguments + Local Variables (I)

```
void func(int x) {
    int a;
    int b;

    ...
}
void main(int argc, char* argv[]) {
    int a_m;
    int *ptr;
1   ...
    func(a_m);
}
```



Input Arguments + Local Variables (II)

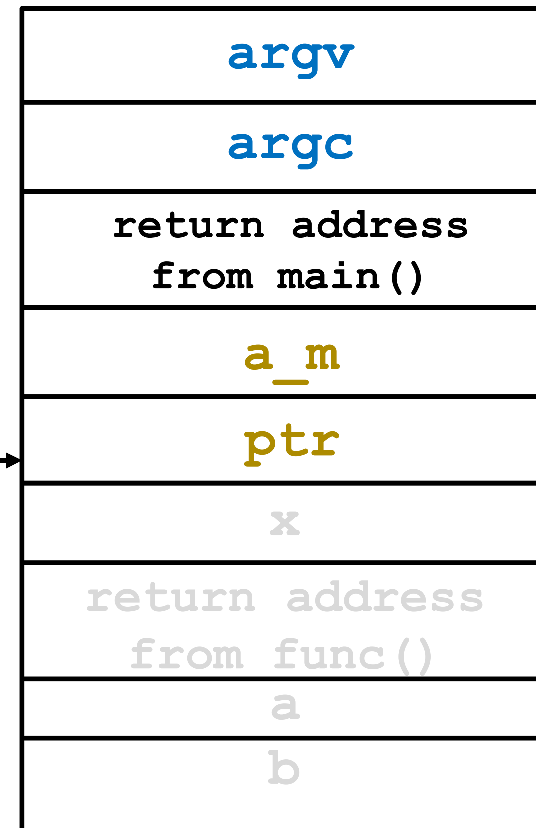
```
void func(int x) {
    int a;
    int b;
2   ...
}
void main(int argc, char* argv[]) ) {
    int a_m;
    int *ptr;
    ...
    func(a_m);
}
```



Input Arguments + Local Variables (III)

```
void func(int x) {
    int a;
    int b;

    ...
}
void main(int argc, char* argv[]) {
    int a_m;
    int *ptr;
    ...
    func(a_m);
}
```

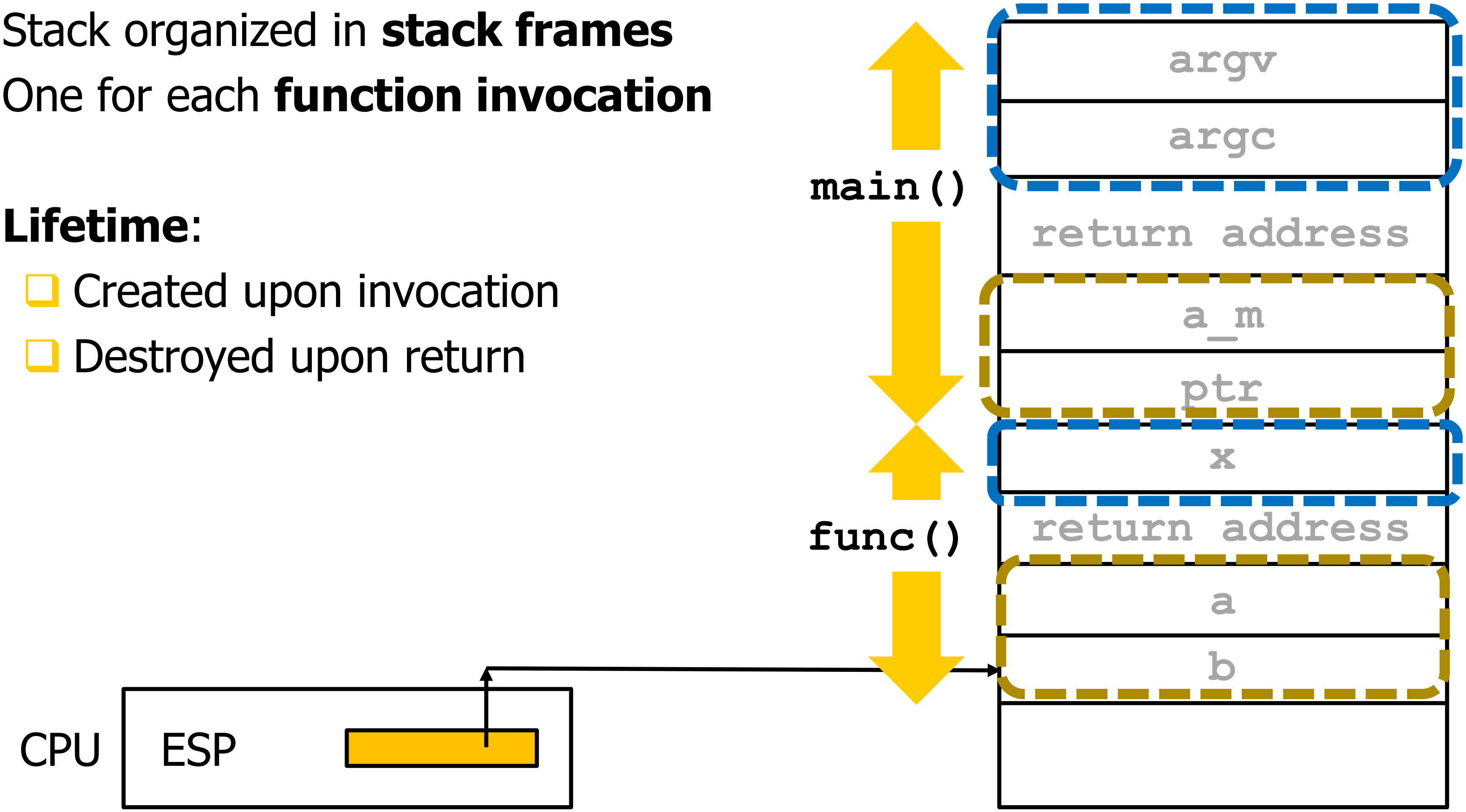
3



Stack Frame (Activation Record)

- Stack organized in **stack frames**
- One for each **function invocation**

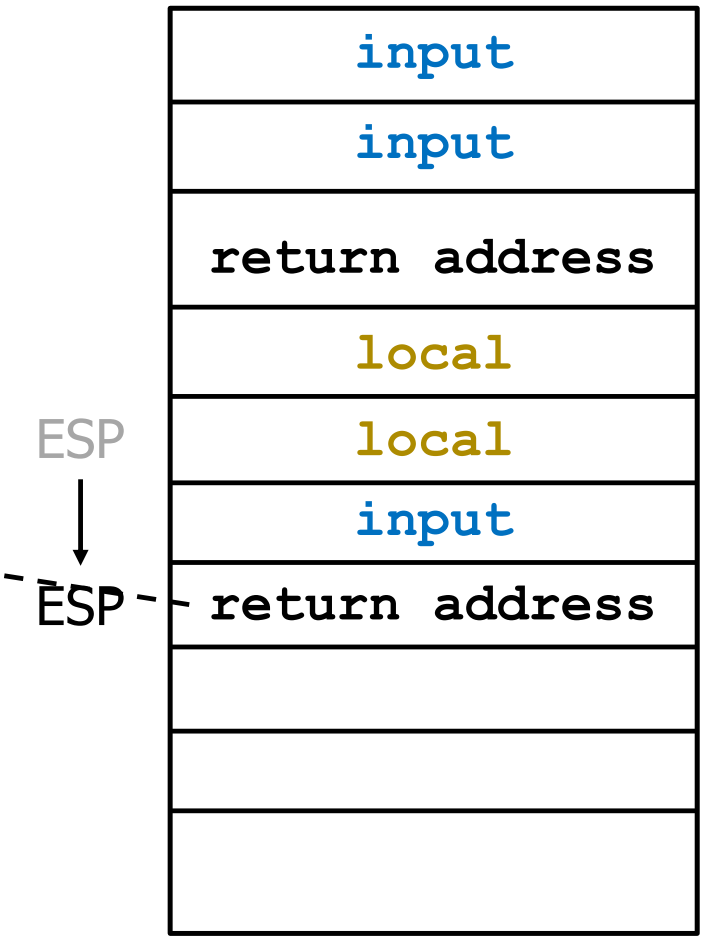
- **Lifetime**:
  - Created upon invocation
  - Destroyed upon return



Who manages Stack Frames? (I)

- Code generated by the compiler

```
caller:
...
push ...      ; input arguments
call callee
```

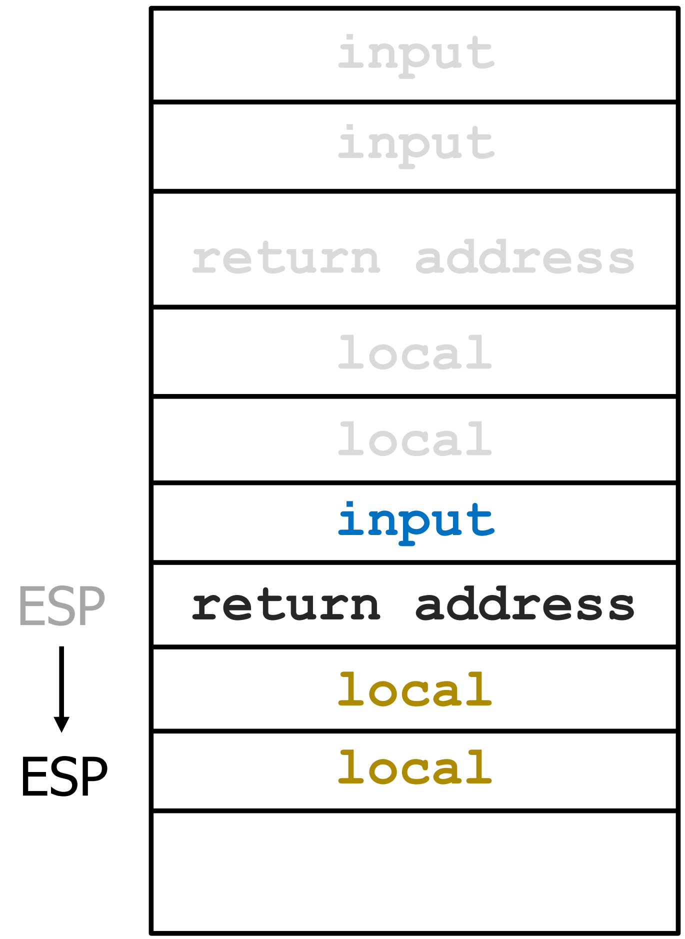


Who manages Stack Frames? (II)

- Code generated by the compiler

```
caller:
...
push ...      ; input arguments
call callee
```

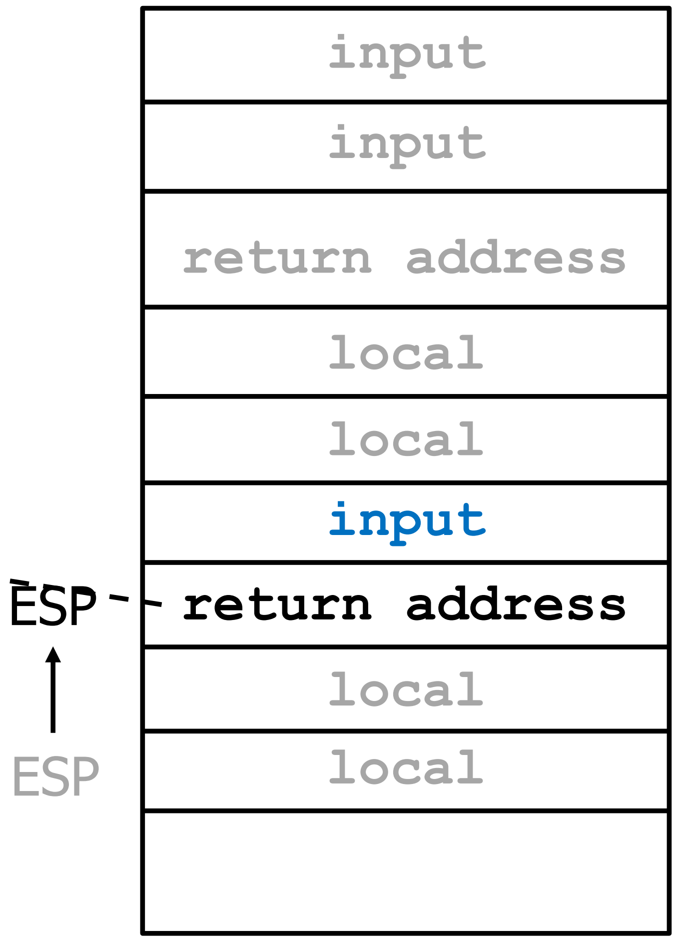
```
callee:
push ...      ; local variables
...           ; function execution
```



Who manages Stack Frames? (III)

- Code generated by the compiler

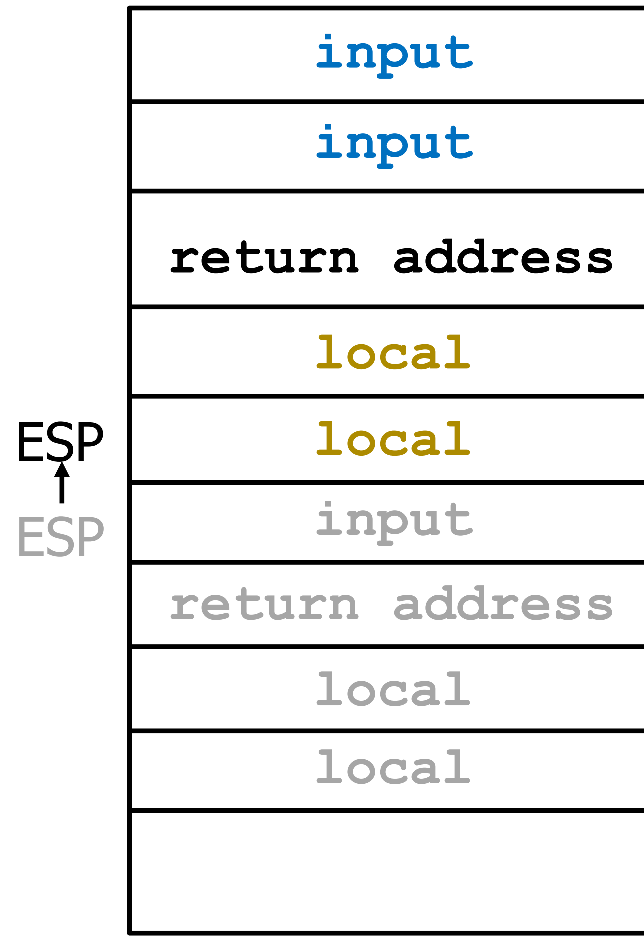
```
caller:
...
push ...      ; input arguments
call callee


callee:
push ...      ; local variables
...           ; function execution
pop ...       ; free local variables
ret
```



Who manages Stack Frames? (IV)

- Code generated by the compiler

```
caller:
...
push ...      ; input arguments
call callee
pop ...       ; free input arguments
...

callee:
push ...      ; local variables
...           ; function execution
pop ...       ; free local variables
ret
```



Who manages Stack Frames? (V)

- Code generated by the compiler

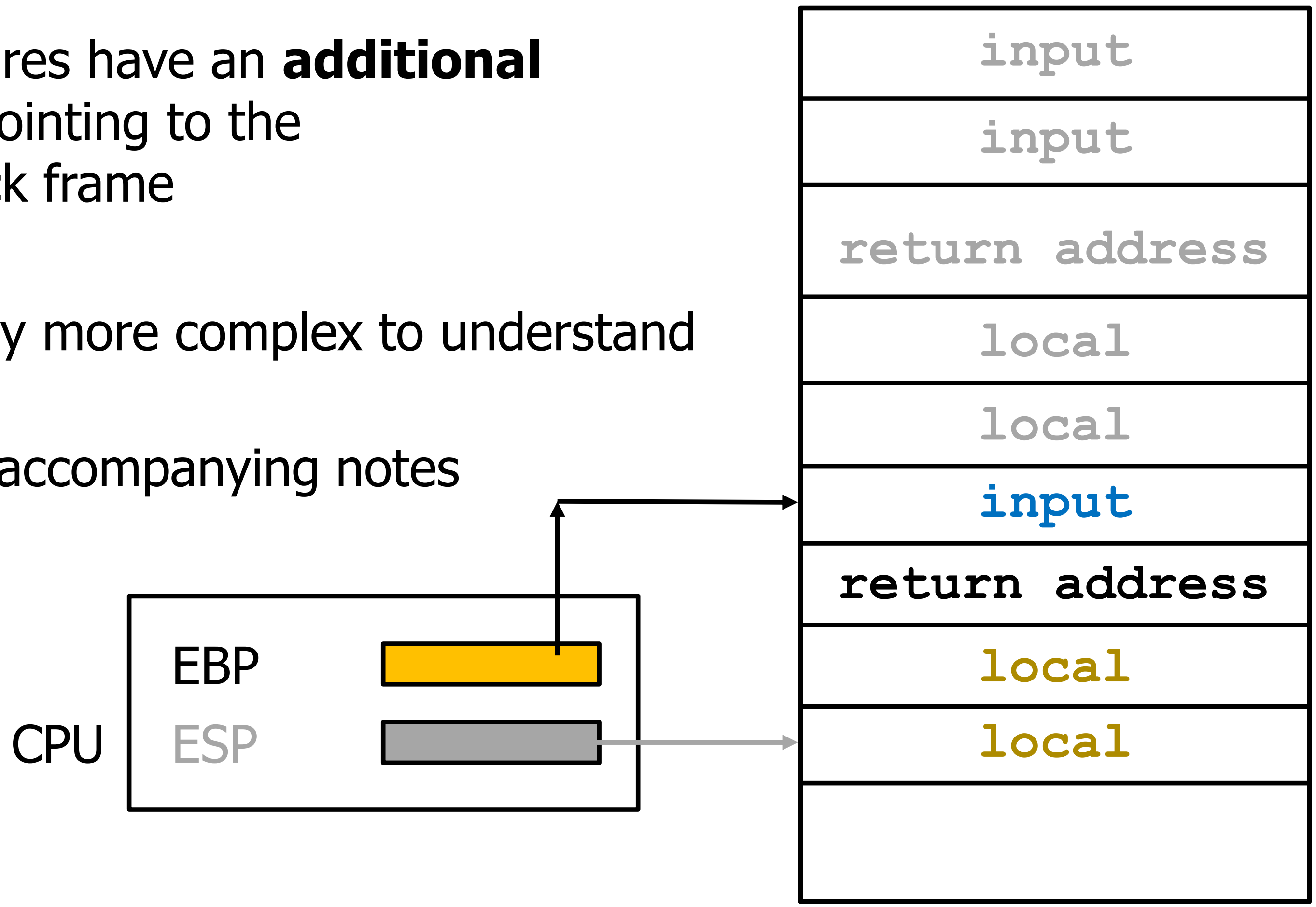
```
caller:
...
push ...      ; input arguments
call callee
pop ...       ; free input arguments
...

callee:
push ...      ; local variables
...           ; function execution
pop ...       ; free local variables
ret
```

Every **invocation** prepares input args before and frees them after

Every **function** has a **prologue** and an **epilogue**

Remark

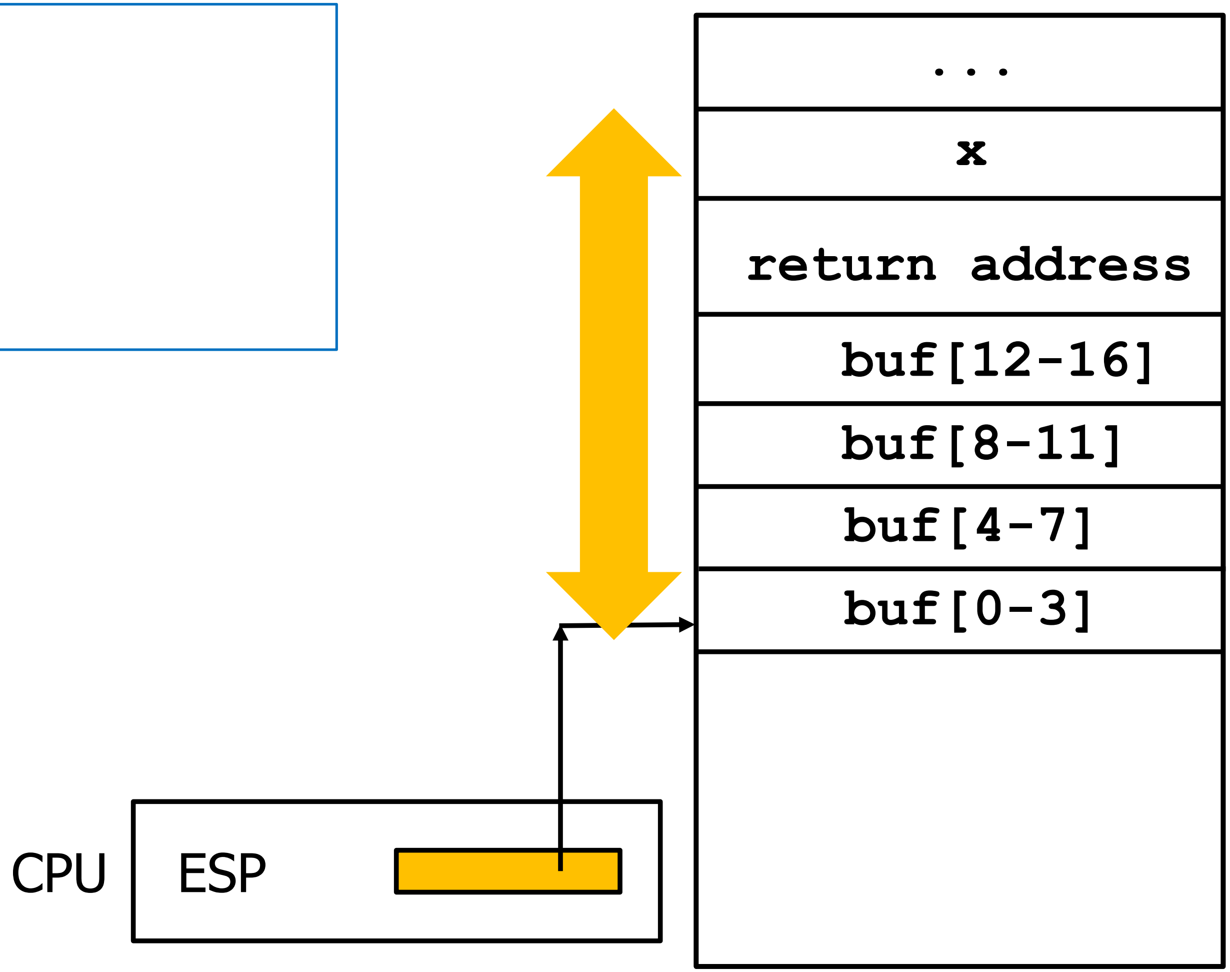
- Modern architectures have an **additional** CPU register for pointing to the **base** of each stack frame
- Real details slightly more complex to understand
- Full details in the accompanying notes



Memory Corruption: Stack Smashing

Hypothetical Example

```
void f(int x) {
  char[16] buf;
   ...
}
...
```



Terminology

- Buffer overflow on the stack
  - Overflow on a variable allocated on the stack

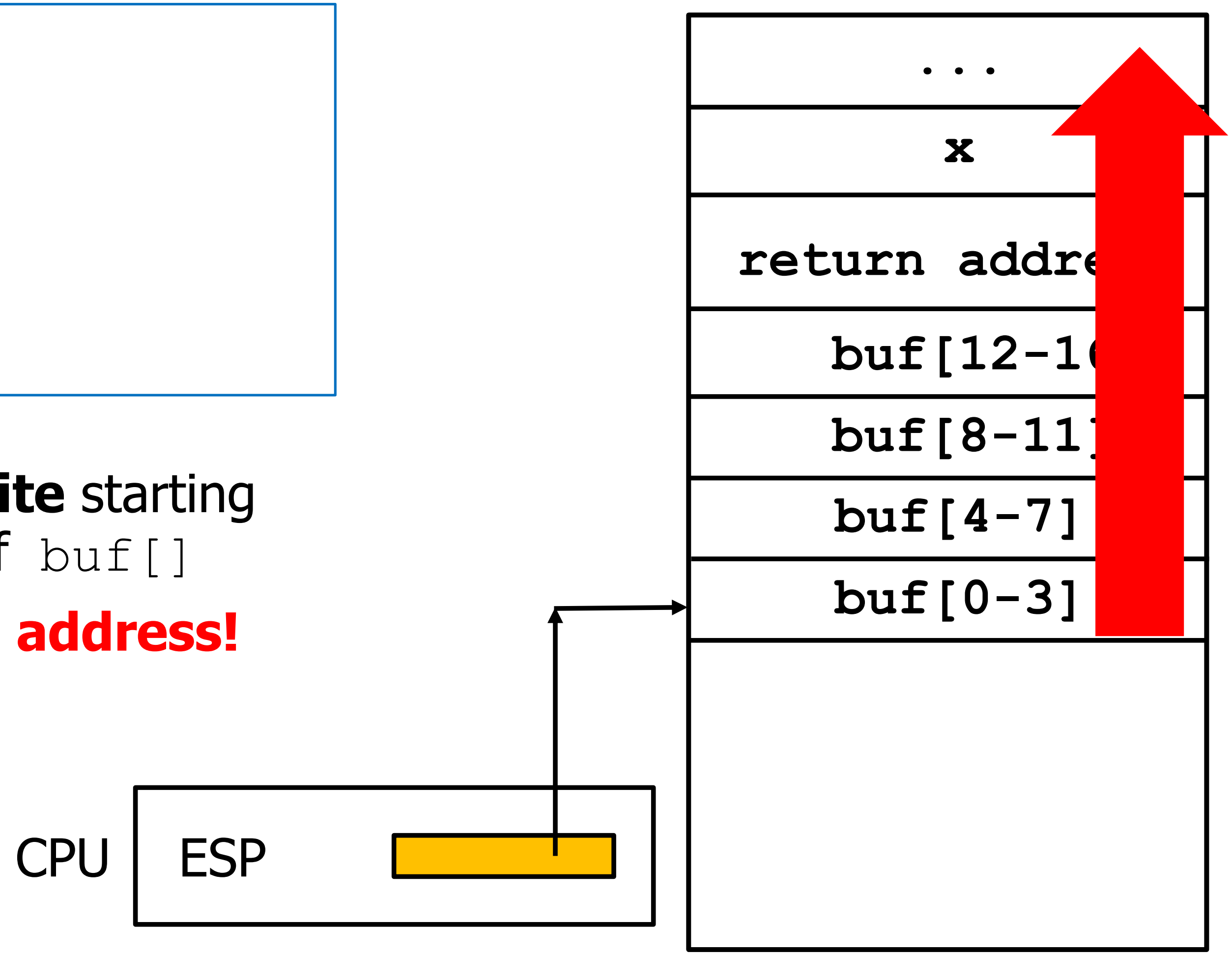
- Stack overflow
- Stack smashing

- IF stack overflow on adversary-controlled variable
- THEN **systematic** techniques for code **injection** / code **reuse**

Stack Overflow

```
void f(int x) {
  char[16] buf;
  gets(buf);
  ...
}
...
```

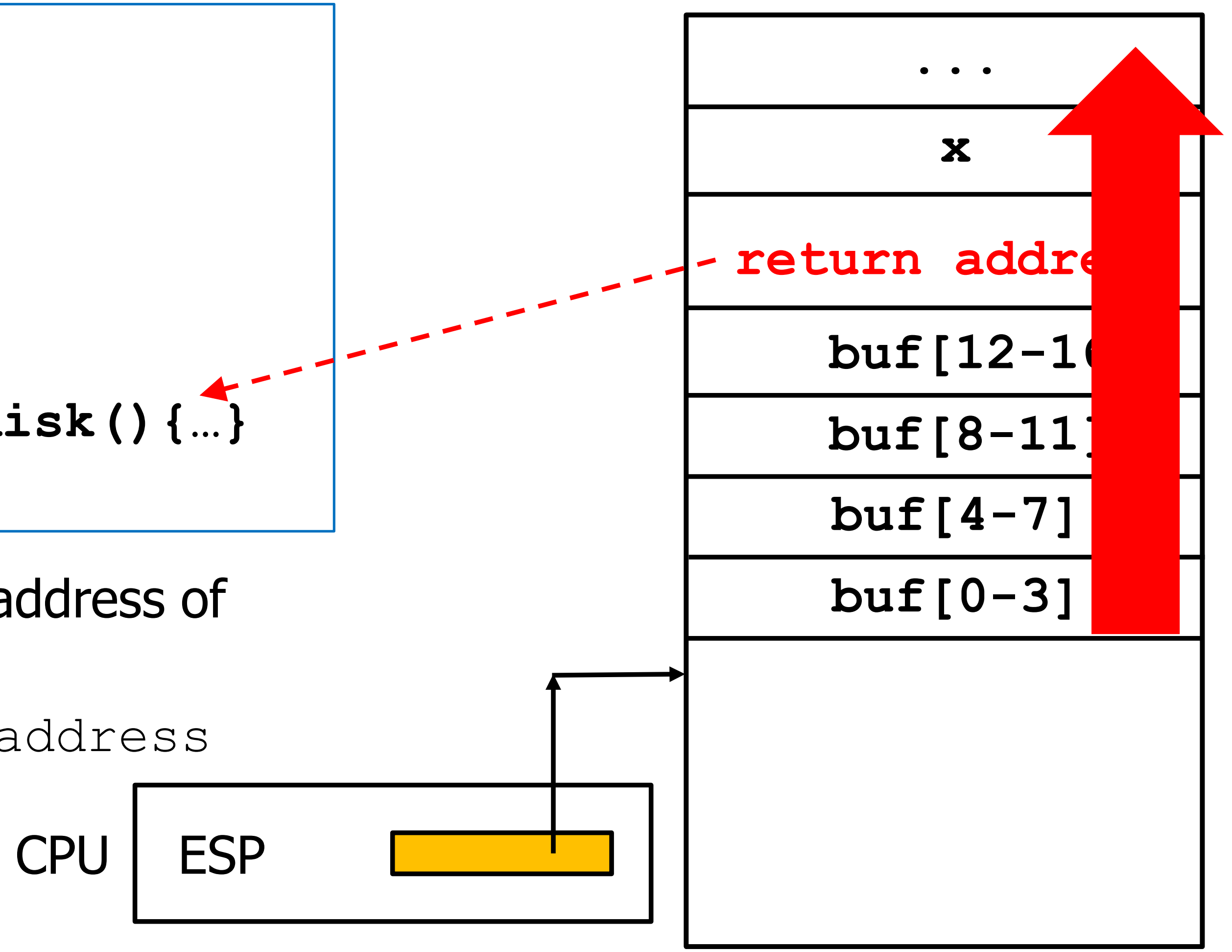
- Attacker can **overwrite** starting from the beginning of `buf[]`
- **...it controls return address!**



Exploit:
Code Reuse (I)

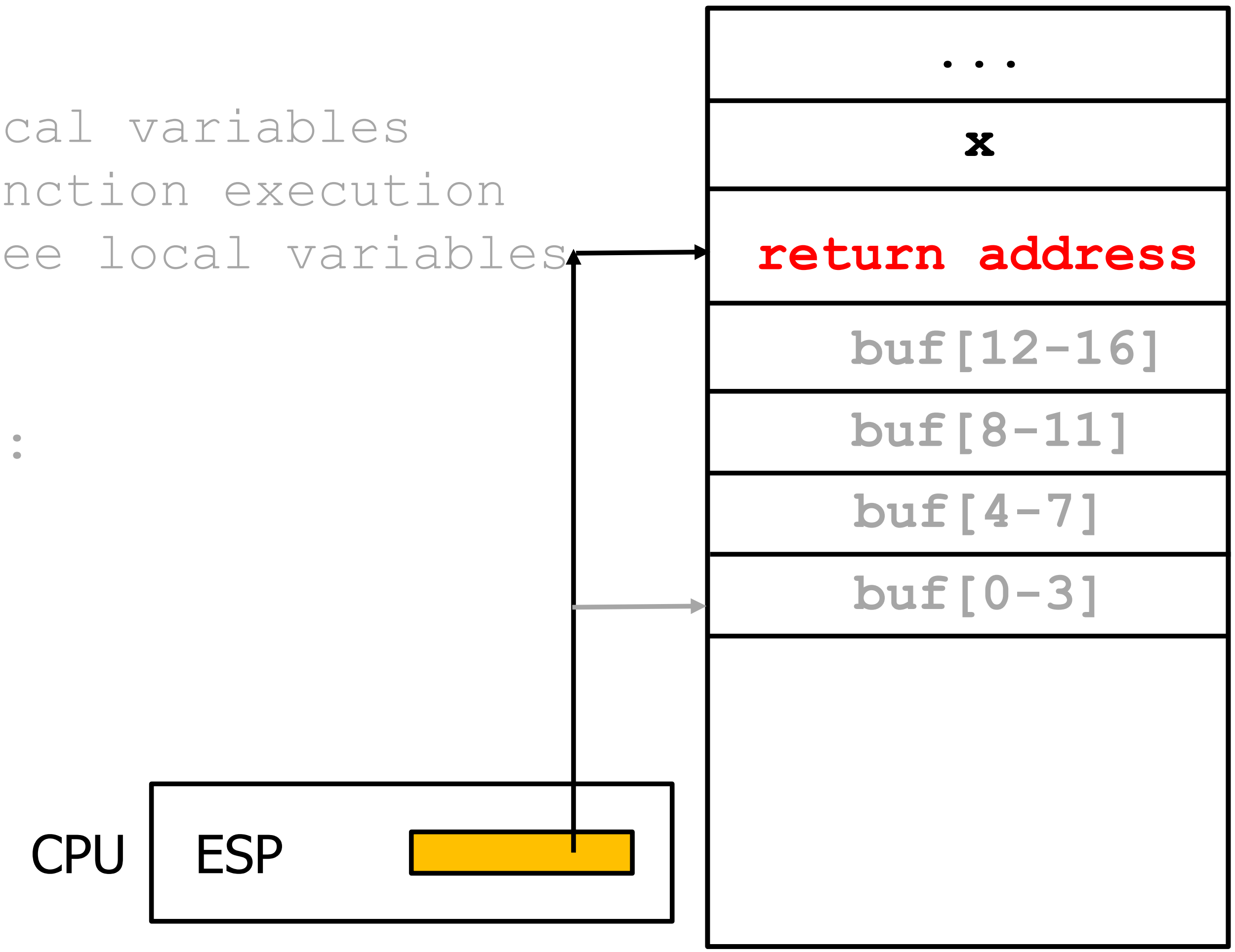
```
void f(int x) {
  char[16] buf;
  gets(buf);
   ...
}
...
void format_hard_disk(){…}
...
```

1. Attacker determines address of function of interest
2. Overwrites `return address` with that address



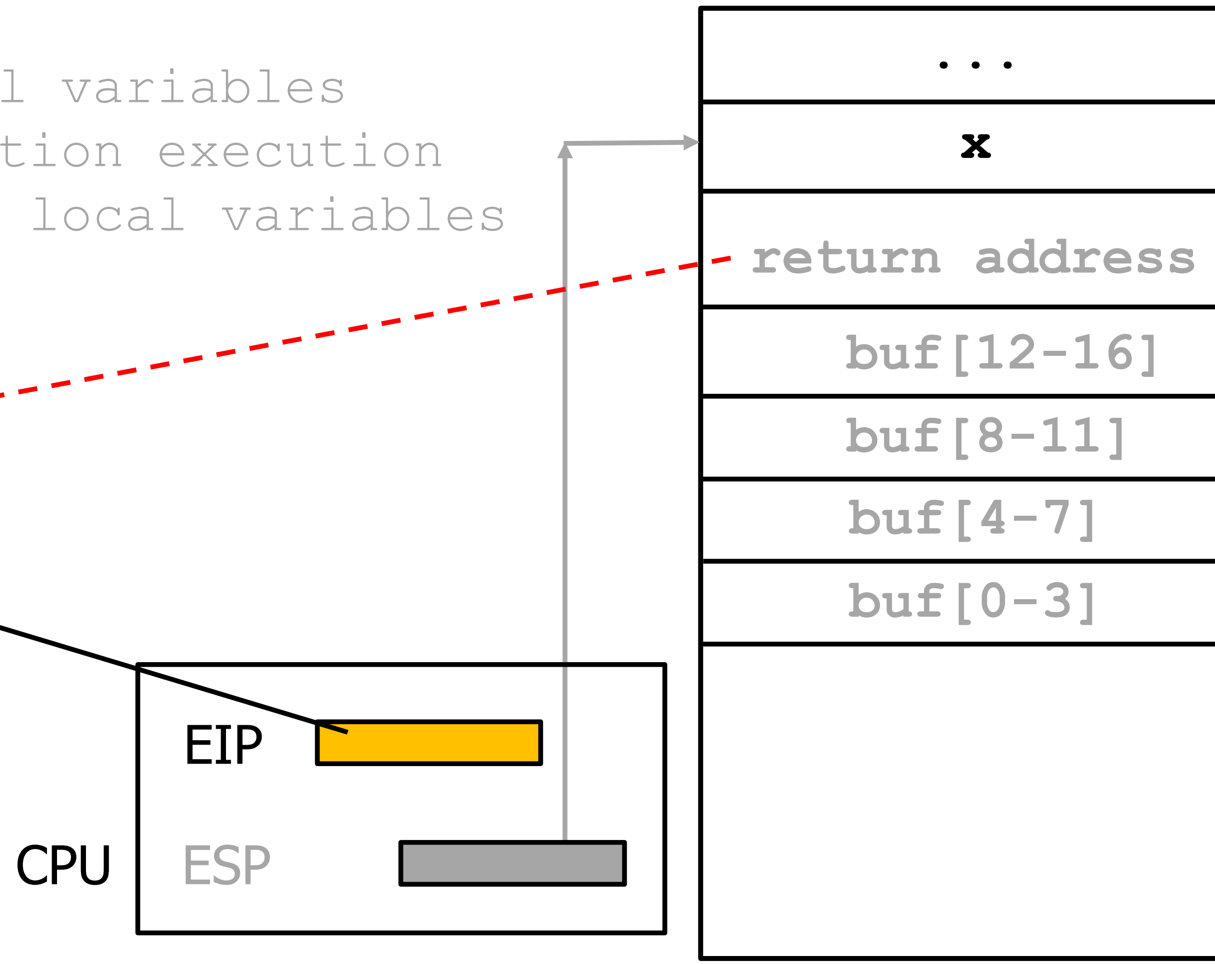
Exploit:
Code Reuse (II-a)

```
callee:
push ...        ; local variables
...             ; function execution
pop ...         ; free local variables
ret
...
format_hard_disk:
...
```



Exploit: Code Reuse (II-b)

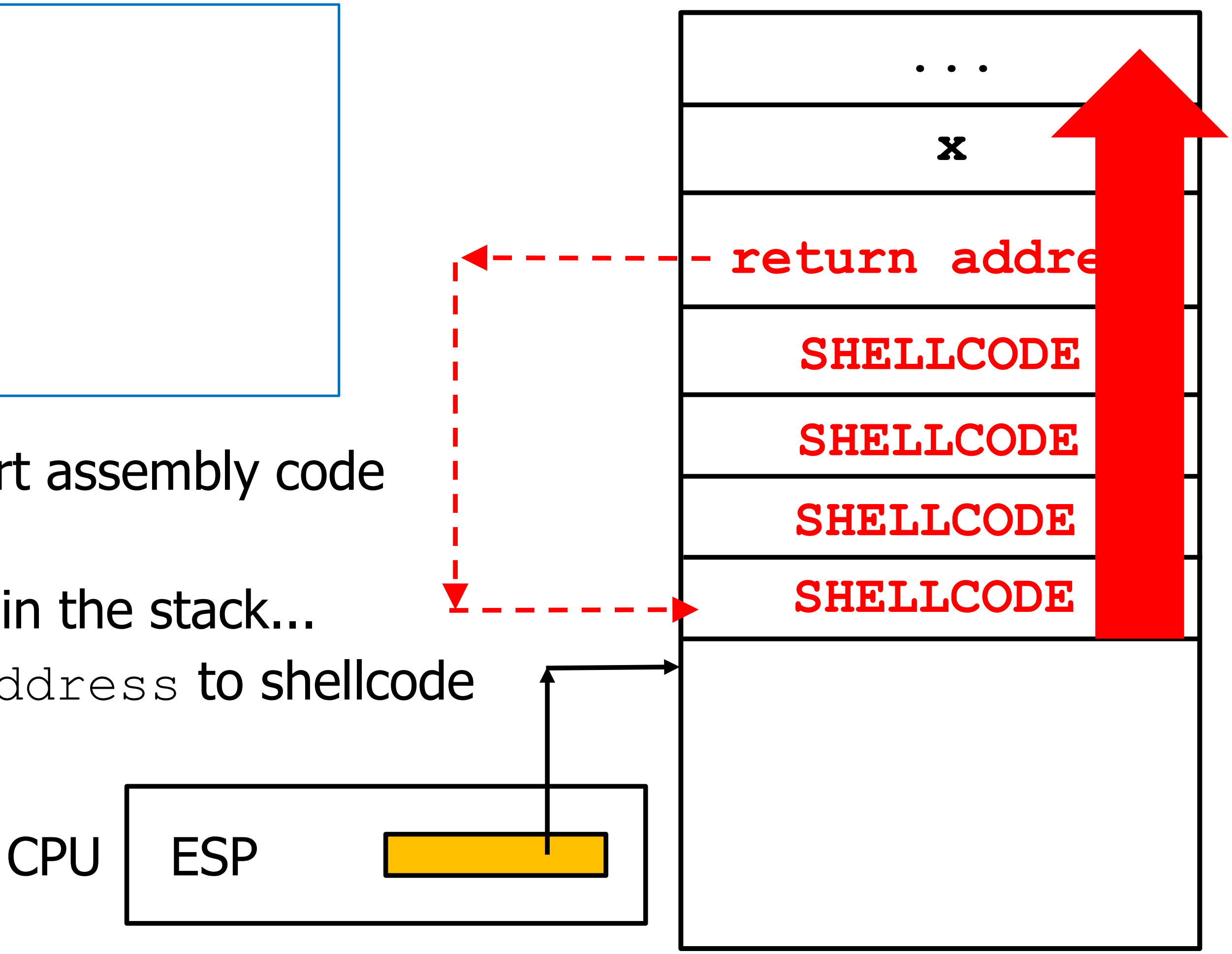
```
callee:
push ...        ; local variables
...             ; function execution
pop ...         ; free local variables
ret
...
format_hard_disk:
...
```



Exploit:
Code Injection (I)

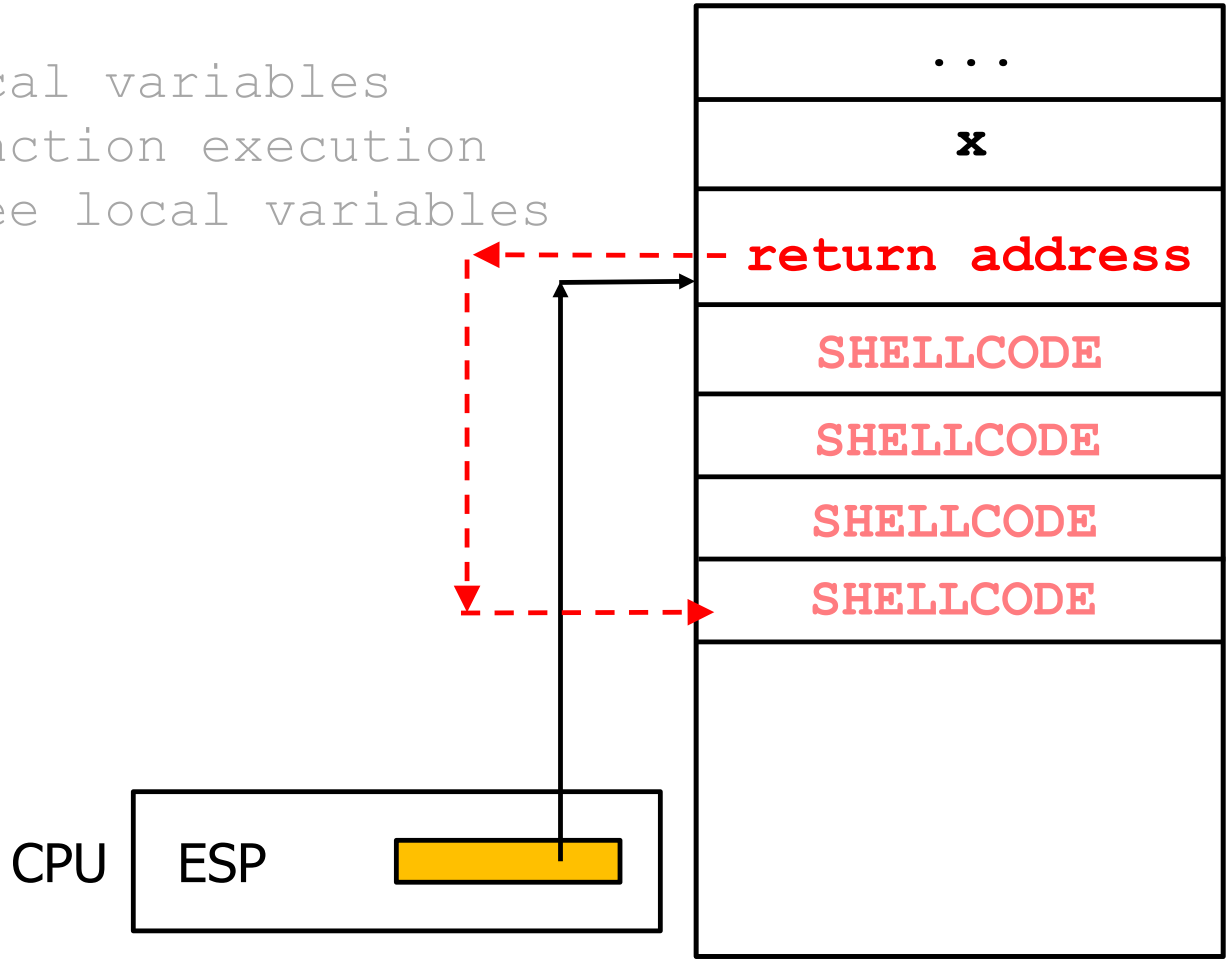
```
void f(int x) {
  char[16] buf;
  gets(buf);
  ...
}
...
```

1. Attacker writes a short assembly code ("**shellcode**")
2. Overwrites shellcode in the stack...
3. ...and set `return address` to shellcode

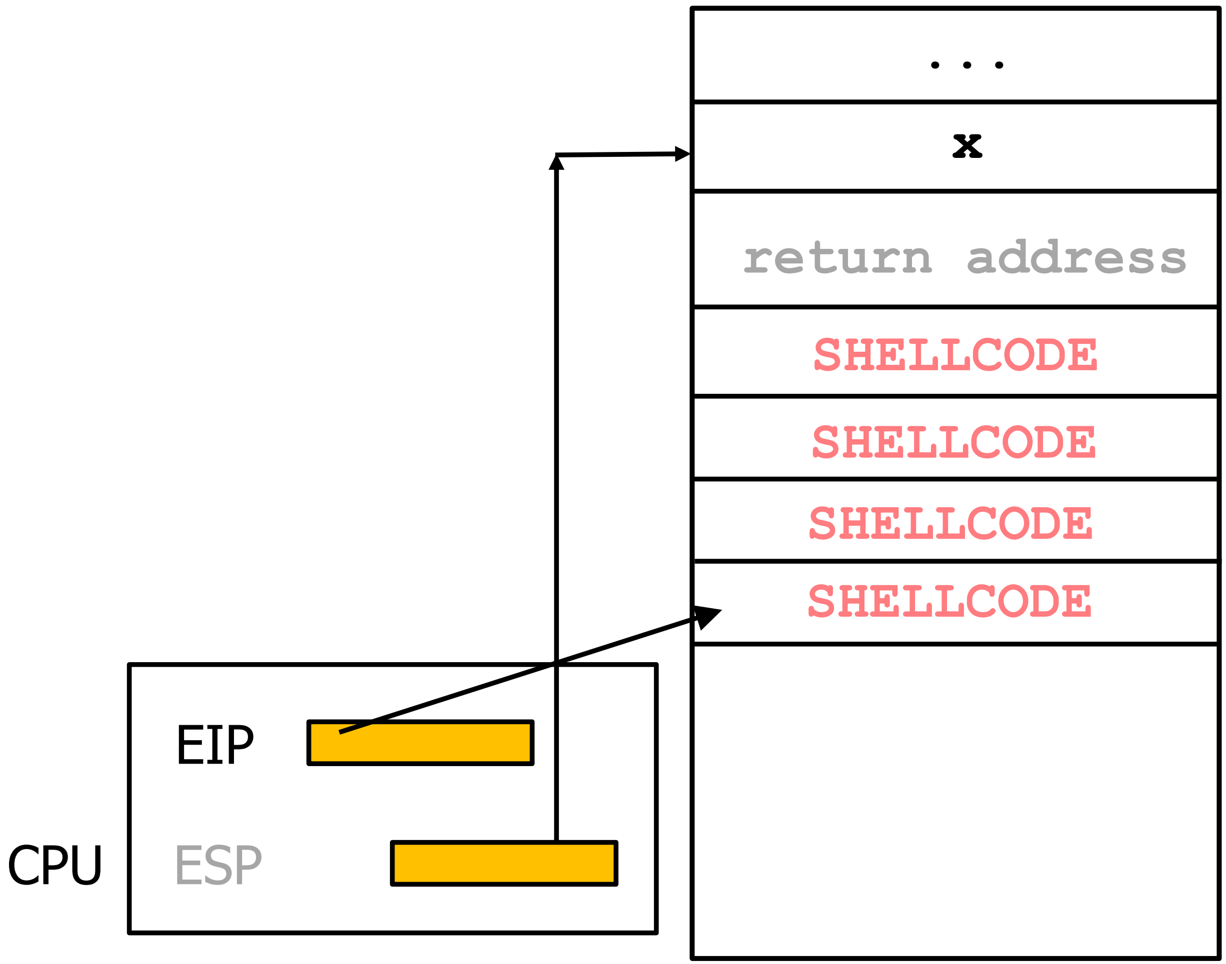


Exploit:
Code Injection (II-a)

```
callee:
push ...      ; local variables
...           ; function execution
pop ...       ; free local variables
ret
```



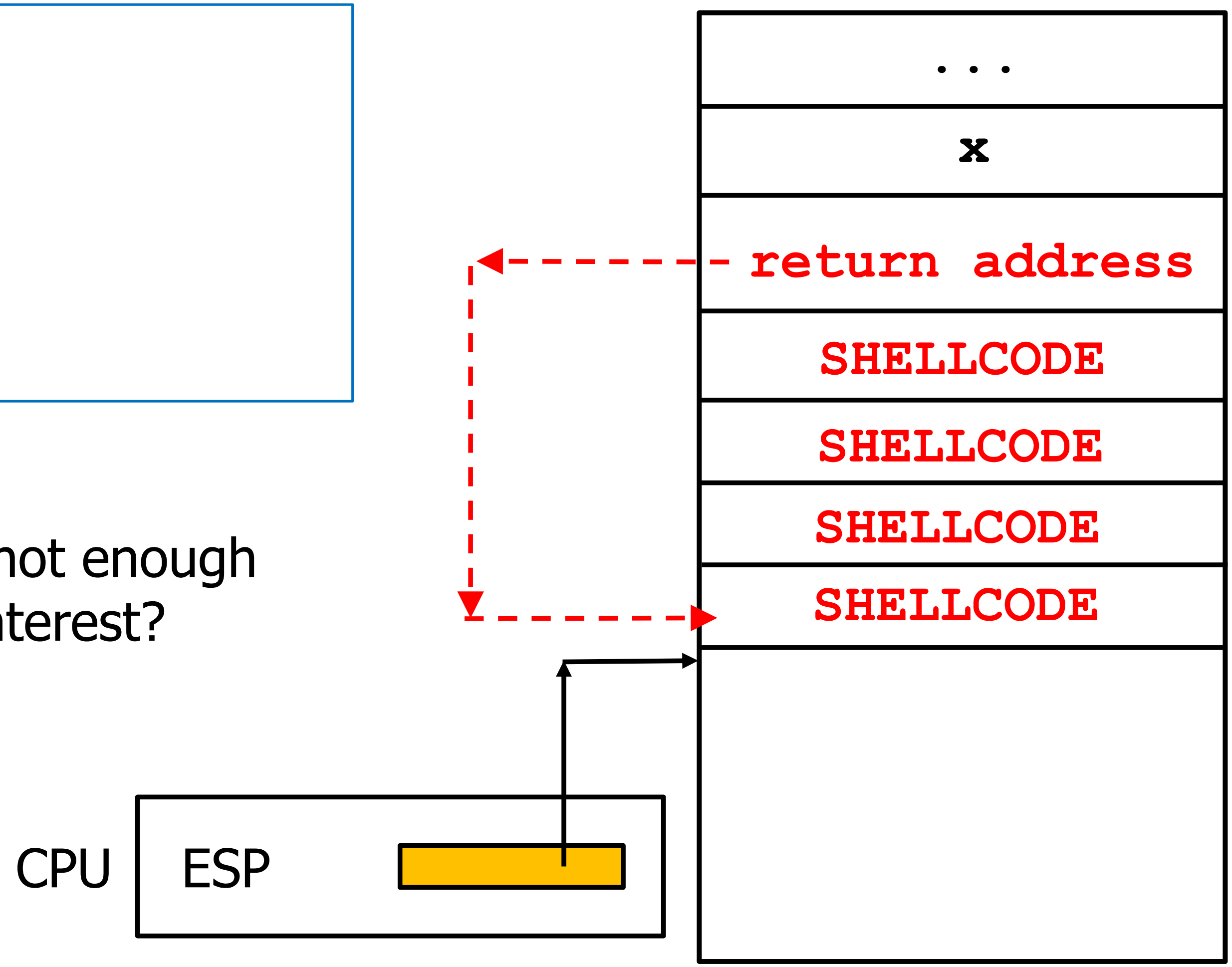
Exploit: Code Injection (II-b)



Hhmmm...

```
void f(int x) {
   char[16] buf;
   gets(buf);
   ...
}
...
```

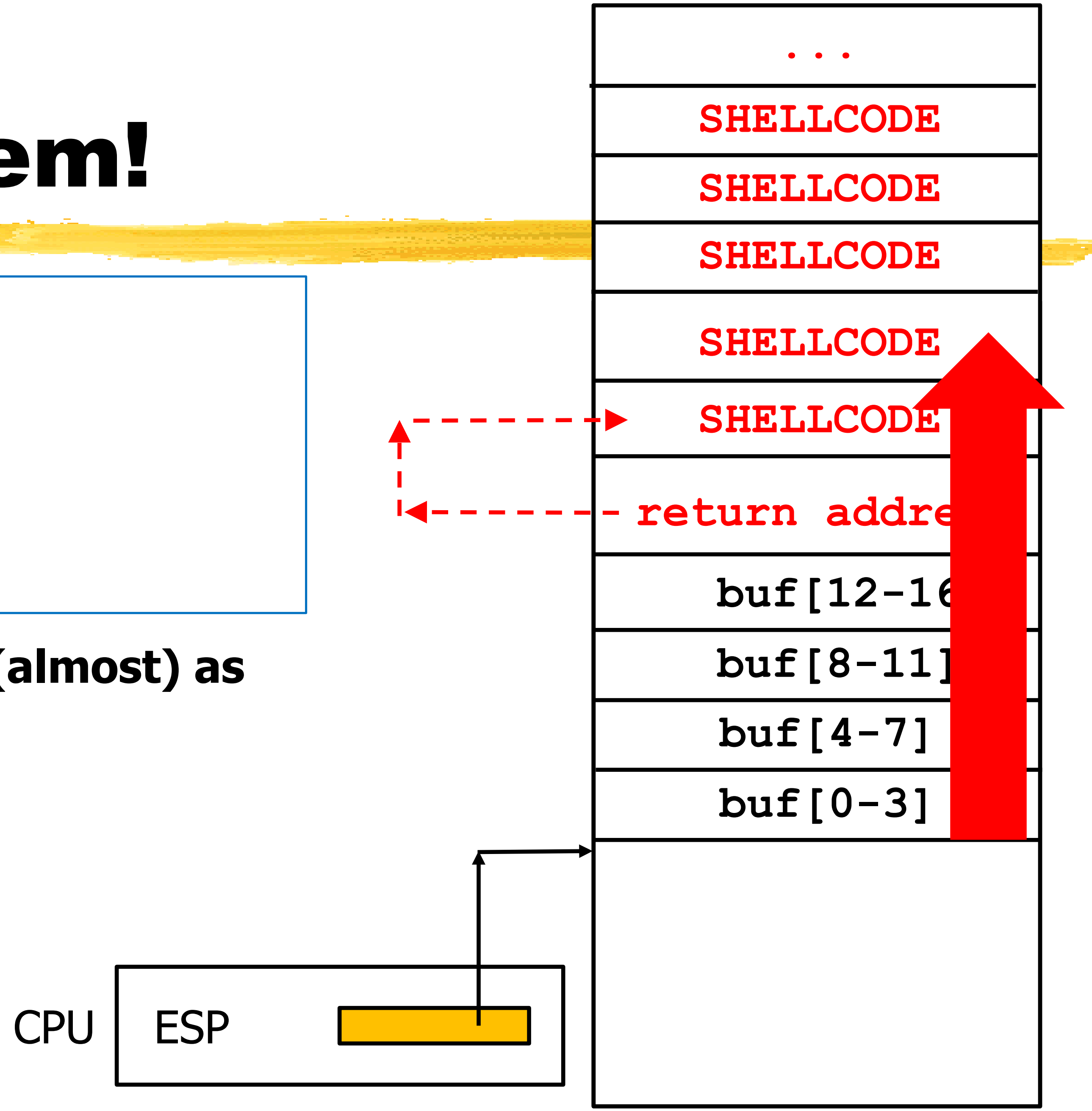
- What if 16 bytes are not enough for the shellcode of interest?



No problem!

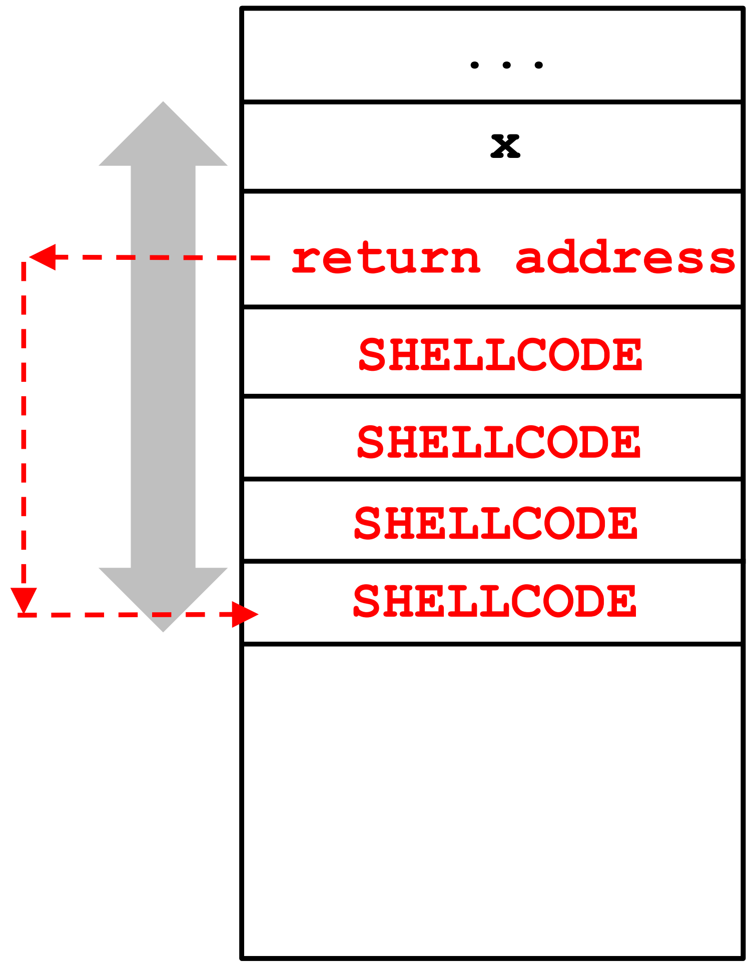
```
void f(int x) {
   char[16] buf;
   gets(buf);
   ...
}
...
```

- We can overwrite (almost) as much as we need!



Buffer Overflow on the Stack

- Can be exploited **systematically**:
  - Code **reuse**:
    overwrite return value
  - Code **injection**:
    owewrite shell code and return value



More Implications of Buffer Overflows

Buffer Overflow on the Heap / Data

- Can it be exploited for code injection / reuse?
- Short answer: it **may** be possible
  - More difficult
  - Not systematic

- **Necessary** condition: ability to **overwrite a function pointer**

Hypothetical Example: Data (I)

```
...
int (*fn_ptr)(void);
...
int someFunc(...) {
    ...
    a = (*fn_ptr)();    // invoke pointed function
}
...
```

Hypothetical Example: Data (II)

```
char vect[...];
...
int (*fn_ptr)(void);
...
int someFunc(...) { ... /* invoke (*fn_ptr)() */ ... }
...
```

- IF overflow on `vect[]` with attacker-controlled value
- THEN code re-use
  (overwrite `fn_ptr` with address of existing function)

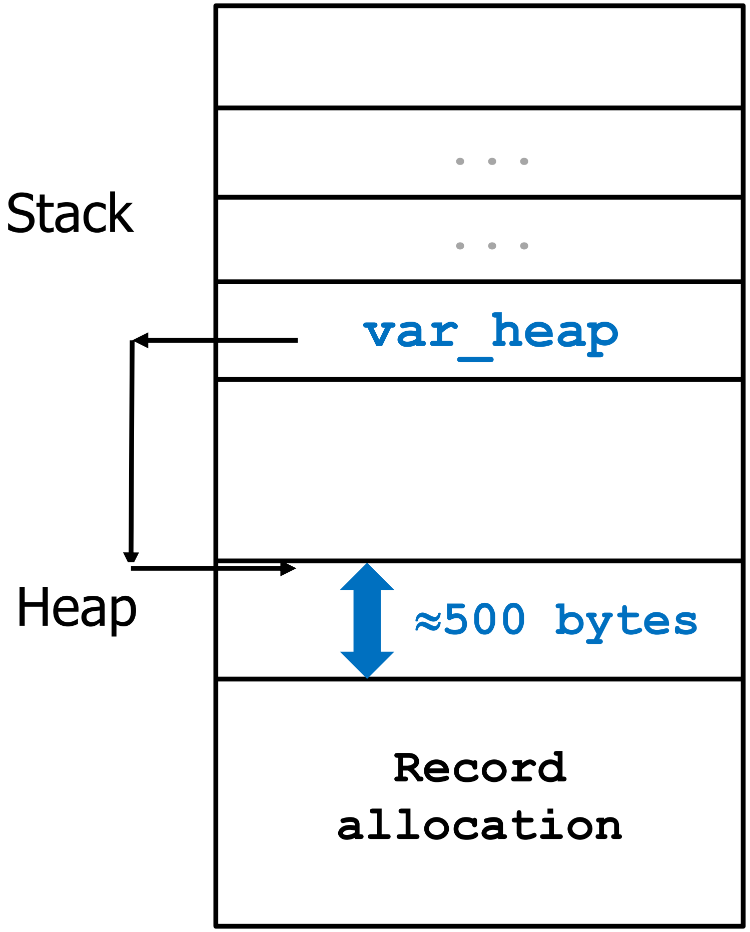
  code injection
  (overwrite `vect[]` and surroundings with shellcode)
  +
  (overwrite `fn_ptr` with address of shellcode)

Hypothetical Example: Heap (I-a)

```
...
typedef struct {
int buf[500];
int (*funcPtr)(int);
} Data;
...
int main() {
   struct Data* var_heap = (Data*)malloc(sizeof(Data));
   var_heap->funcPtr = someFunction;
   var_heap->(*funcPtr)(47);
   var_heap->buf[81] = 7;
   ...
```

- `var_heap` is a local variable of type pointer (resides on the **stack**)
- Its value is an **address in the heap**

Hypothetical Example: Heap (I-b)




```
typedef struct {
int buf[500];
int (*funcPtr)(int);
} Data;
```

Hypothetical Example: Heap (II)

```
...
typedef struct {
int buf[500];
int (*funcPtr)(int);
} Data;
...
```

- IF overflow on `var_heap->buf` with attacker-controlled value
- THEN code re-use / injection
  (overwrite `var_heap->funcPtr` ...)

Exploitation by Overwrite: Recap

- Overflows on the **stack** ⇒ Overwrite on the **stack**
- Overflows on the **heap** ⇒ Overwrite on the **heap**
- Overflows on the **data** ⇒ Overwrite on the **data**

- **Could** be exploited
  - Code re-use / injection
  - Write other variables
    ⇒ More attacker-controlled variables
    (beyond the intended program flow)

More Exploitation by Overwrite (I)

- Overflows on the **stack** $\Rightarrow$ Overwrite on the **stack**
- Overflows on the **heap** $\Rightarrow$ Overwrite on the **heap**
- Overflows on the **data** $\Rightarrow$ Overwrite on the **data**

- Can an overflow on a **region** allow writing in **another** region?

- That would give **much more freedom** for exploitation
  (many more opportunities for **controlling** variables)
  - Overflow on the **stack** $\Rightarrow$ Overwrite function pointer on the **heap**
    Overwrite variable on the **data**
    ...

More
Exploitation by Overwrite (II)

- Can an overflow on a **region** allow writing in **another** region?
- That would give **much more freedom** for exploitation
(many more oppportunities for **controlling** variables)

- Short answer: it **may** be possible
- **Necessary** condition: ability to control value of a (data) **pointer**

Hypothetical Example: Data (I)

```
...
int val;
...
int someFunc(...) {
    ...
    int *ptr_data = &val;
    ...
    // write value a at address val
    *ptr_data = a;
    ...
}
...
```

Hypothetical Example: Data (II-a)

```
...
int val;
...
int someFunc(...) {
    ...
    int *ptr_data = &val;
    ...
    // write value a at address val
    *ptr_data = a;
    ...
}
...
```

- IF stack overflow allows controlling `ptr_data`
- THEN write `a` at **any** address of choice

- Write specified value at arbitrary address

Hypothetical Example: Data (II-b)

```
...
int val;
...
int someFunc(...) {
    ...
    int *ptr_data = &val;
    ...
    // write value a at address val
    *ptr_data = a;
    ...
}
...
```

- IF adversary also controls `a`
- THEN Write arbitrary value at arbitrary address

Hypothetical Example: Heap

```
...
int someFunc(...) {
    ...
    int *ptr_heap = malloc(100);
    ...
    // write value a at address ptr_heap
    *ptr_heap = a;
...
}
...
```

- IF stack overflow allows controlling `ptr_heap`
- THEN write `a` at **any** address of choice

- Write specified value at arbitrary address

Hypothetical Example: Heap

```
...
int someFunc(...) {
    ...
    int *ptr_heap = malloc(100);
    ...
    *ptr_heap = a;
...
}
...
```

- IF overflow on stack allows controlling `ptr_heap`
- THEN write `a` at **any** address of choice

- IF attacker also controls `a`
- THEN attacker also controls written value

Memory Corruption Recap

Memory Corruption (I)

- Memory corruption bug: program accesses memory "incorrectly"
- Memory corruption bugs can be exploited by attackers (**vulnerability**)
- One of the **oldest problems** in computer security

- **#1 weakness in C / C++**
  - These languages are **not memory-safe**
  - Programmer is responsible for memory management
- Typical cause: **arrays**, **pointers**, **strings**, **dynamic memory**
- Tricky to spot and prevent

Memory Corruption (II)

- Out-of-bound write is just **one** of several memory corruption issues
  - Out-of-bound read
  - Use after free
  - Format string attack
  - ...
  - *(out of scope of this course)*

More Insecurity

- **Many additional** sources of insecurity
  - Remember example **integer overflow**

- Bug: `strlen(`**`to_be_copied_to_d`**`) > 32K`

  ⇒ `len` takes a negative value (**integer overflow)**

  ⇒ `if` condition satisfied

  ⇒ **buffer overflow** when writing on **`d`**

  ⇒ overwrite something useful

Absence of Language-Level Security

In a **safer** programming language than C/C++,
the programmer would **not** have to worry about

- **Writing past array bounds**
  (because you'd get an IndexOutOfBoundsException instead)
- **Strings not having a null terminator**
  (because terminators would be inserted by the compiler/interpreter)
- **Integer overflow**
  (because you'd get an IntegerOverflowException instead)
- ...

Design principles of ALGOL 60

Tony Hoare, Turing Award lecture **1980**

- “The first principle of Algol **60** was **security** : ... **every subscript was checked at run time against both the upper and the lower declared bounds of the array**.
- Many years later we asked our customers whether they wished an option to switch off these checks in the interests of efficiency. Unanimously, they urged us not to - they knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous.
- I note with fear and horror that even in **1980**, language designers and users have not learned this lesson. **In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law**.”

Defending against Memory Corruption vulns

Credits

- **A lot** of what follows is based on material from:
  1. Computer Security Course – Berkeley CS161
  2. Software Security Course – Radboud University
  - Mostly from 1

- Any possible mistakes/inaccuracies are mine

Strategies

Vuln prevention:

1. Use safer programming **languages**
2. **Learn to write** memory-safe code
3. Use **tools** for analyzing and patching insecure code

Exploit prevention:

4. Add **mitigations** that make it harder to exploit common vulnerabilities

- Prevention **attempts**

Use safer programming languages

Memory-safe languages

- **Memory-safe languages** are designed to check bounds and prevent undefined memory accesses
  - Examples: Java, Python, C#, Go, **Rust**
  - Most languages besides C, C++, and Objective C

- Memory-safe languages are **not** vulnerable to memory safety vulnerabilities
- **Only** way to stop 100% of memory safety vulnerabilities

Why not used? (I)

- Most commonly cited reason: **performance**

- To make a long story short: **No longer an issue**
- Performance penalty of memory safety is **insignificant**

- Only **possible** exceptions:
  - o.s.
  - certain embedded systems
  - certain gaming platforms

Why not used? (II)

- Real reason: **legacy**

- **Huge** existing code bases are written in C
- Building on existing code is easier than starting from scratch

- Key example: iPhone
- Developed in 1989, we still use Objective C today

Keep in mind

- Programmer time is costly and scarce
  - Writing code in a memory unsafe language tends to take more time

- Memory safe languages often have libraries based on **fast** and **secure** C libraries
  - Python is memory safe
  - Lot of Python app uses NumPy (that internally uses C)

Learn to write + Tools

Learn to write + Tools

Vuln prevention:

1. ...
2. **Learn** to write memory-safe code.
3. Use **tools** for analyzing and patching insecure code.

- Lot of things to say
- Just a couple or remarks

Learn to write memory-safe code (I)

- **Only use libraries that are deemed safe** ("functions that check bounds")

  - Programmer discipline

  +

  - Automatic tools

Learn to write memory-safe code (II)

- **Set of "defensive rules"**
  - Always check a pointer is not null before dereferencing it
  - Always constrain and check data from untrusted sources
  - ...

- Programmer discipline

+

- Automatic tools

- Clearly more difficult to follow systematically

Remark

- **Set of "defensive rules"**
  - ...
  - Always constrain and check data from untrusted sources
  - ...

- Certain defensive rules are crucial
  even with memory-safe languages

- Beyond of scope here

Use tools for analyzing

- **Bug-finding / Code-smelling tools**
  - Look for "common bad practices"
  - Very effective
  - Problem of false positives

- **Fuzzing tools**
  - Inject lot of random inputs
  - Look for a crash (or other unexpected behavior)
  - It is becoming very effective

Mitigations

Mitigations: Basic Idea

- **Make it harder to exploit common vulnerabilities**

- Compiler + O.S.
- Common exploits $\Rightarrow$ program crash
  - **Crashing safer than exploitation**

- Low / Insignificant runtime overhead

Mitigations: Key techniques

- Non-executable pages
  - W^X (Write or Execute)
  - DEP (Date Execution Prevention)
- Address Space Layout Randomization (ASLR)
- Stack Canary
- Pointer authentication

- **More** mitigations exist
  - Out of scope (see links if interested)

Remarks (I)

- Make it **harder** to exploit: **Not** impossible
- Many techniques for trying to **circumvent** mitigations
  - Out of scope: we will just outline the basic ideas

Remarks (II)

- Writing exploits for memory corruption vulns on modern platforms is **VERY DIFFICULT**

1. **Much effort** for circumventing defenses
2. Usually they require **chaining multiple vulns**

O.S.-based Mitigations

O.S. Mitigations

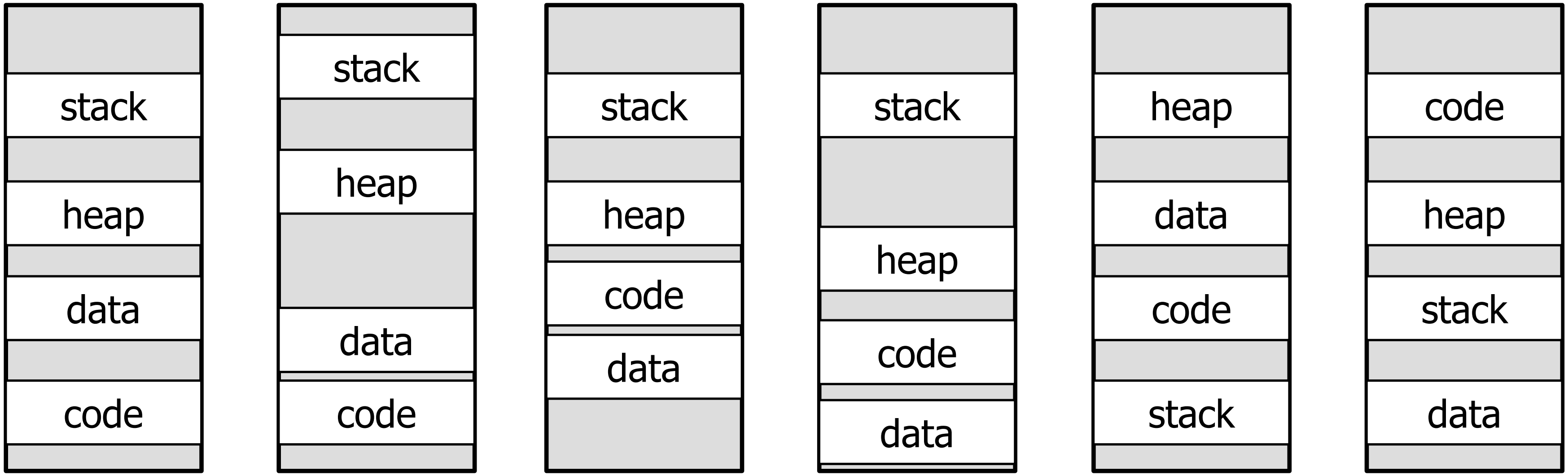
1. Non-executable pages
2. Address Space Layout Randomization (ASLR)

- O.S. support
  - 1 requires also HW support, but this support is ubiquitous

- **Do not require recompilation**
  - **Very important**
    - All libraries and executables unchanged
    - Just switch an option when executing

Mitigation: ASLR

- Place each memory segment in a **different location** each time the program is run



- Attacker cannot prepare exploits with correct addresses

Circumvention IDEA

- Shellcode obtains address of a variable whose **relative address** to shellcode is known
(and then shellcode computes its own address)

- **Brute-force** segment locations
(and then try to obtain other addresses)
  - Randomization usually on "memory page" boundaries
(placed at multiple of 4KB)
  - 32-bit architectures: 2^20 values $\Rightarrow$ can be brute forced
  - 64-bit architectures: 2^36 values $\Rightarrow$ hhmmm

Mitigation: Non-Executable Pages (I)

- Fact: "All" programs do **not** need memory that is **both** written to and executed

- Very powerful defense:
  - Each memory page is **either executable or writable**
  - Mandatory access control (hw + o.s.)

  - Code: Executable // shellcode cannot be **written** here
  - Stack: Writable // shellcode written here cannot **exec**!
  - Data: Writable // ...not even here
  - Heap: Writable // ...not even here

Mitigation: Non-Executable Pages (II)

- Very powerful defense:
  - Each memory page is **either executable or writable**
  - Mandatory access control (hw + o.s.)

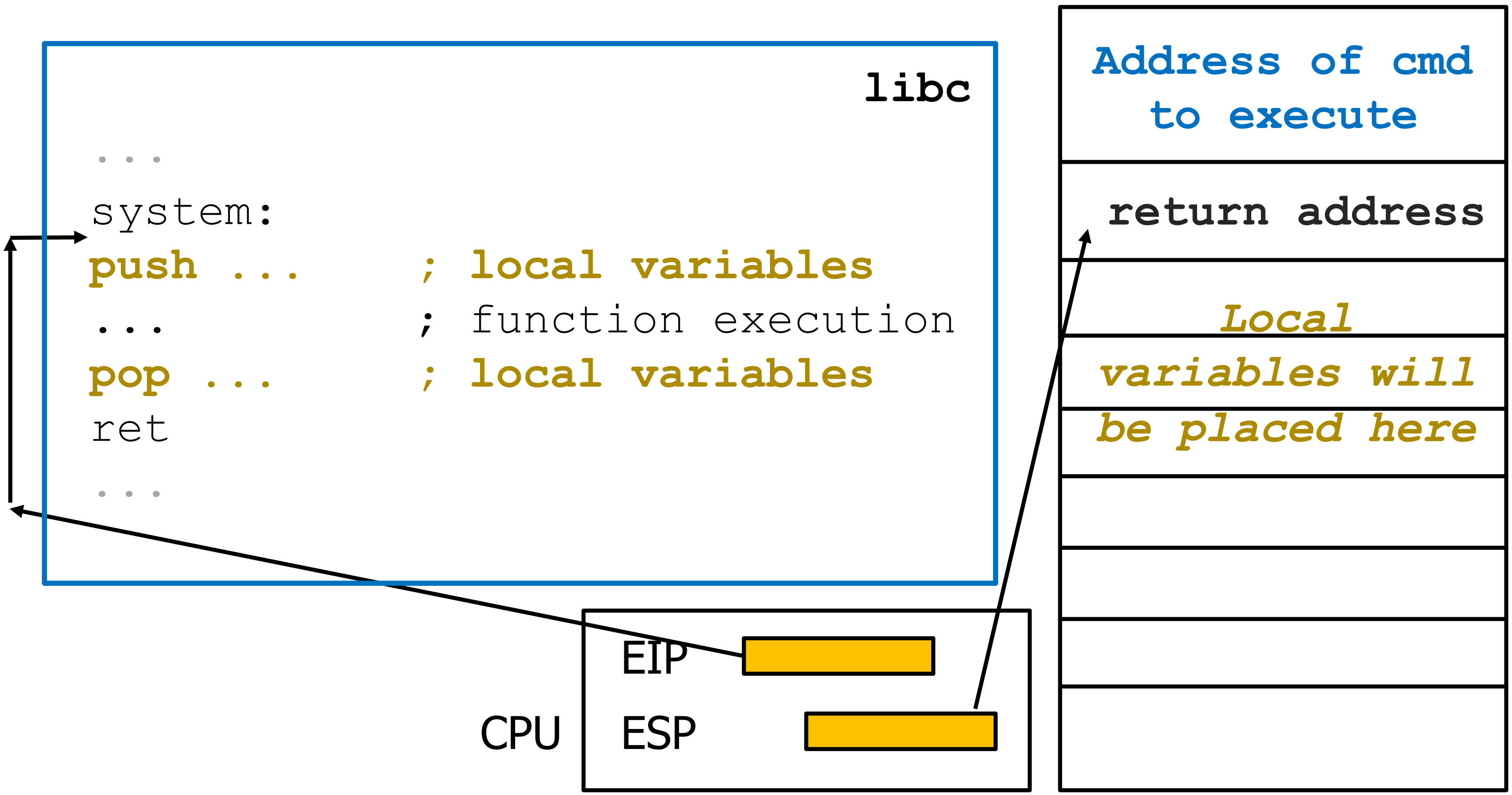
- Common names:
  - **W^X** (Write or Execute)
  - **DEP** (Data Execution Prevention)

Circumvention IDEA: Code reuse: Return to `libc`

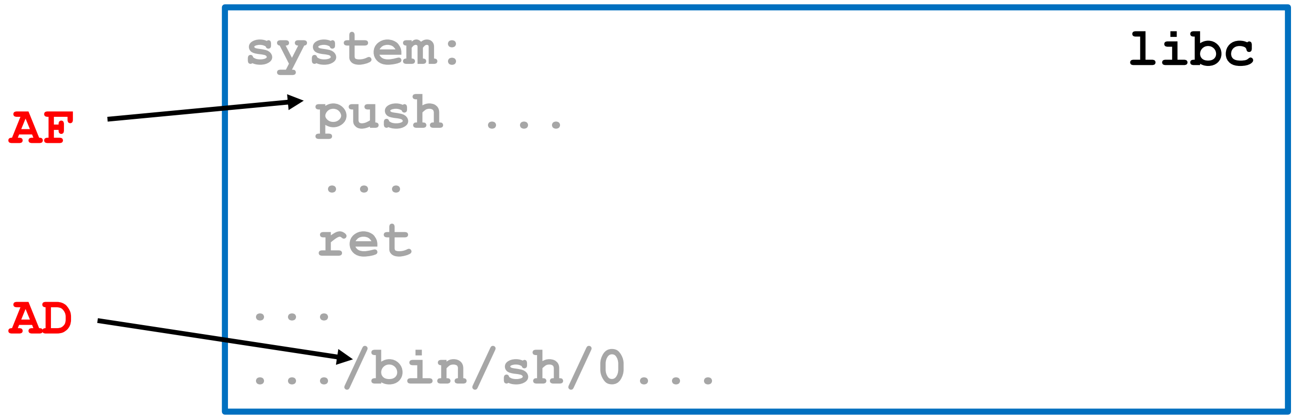
1. Identify potentially useful **function** and **data** that **already exist** in memory (typically in `libc`)
2. Overwrite stack so that:
   - Vulnerable function returns to **existing function**
   - **Existing function** takes **existing data** as input argument

- Example: Adversary wants to invoke `system("/bin/sh")`
  - AF = `address of system in libc`
  - AD = `address of string /bin/shNUL in libc`
  - Overwrite AF and AD at the "right places" in the stack (next slides)

Stack Frame expected by `system` function



Exploit writing prelimiaries

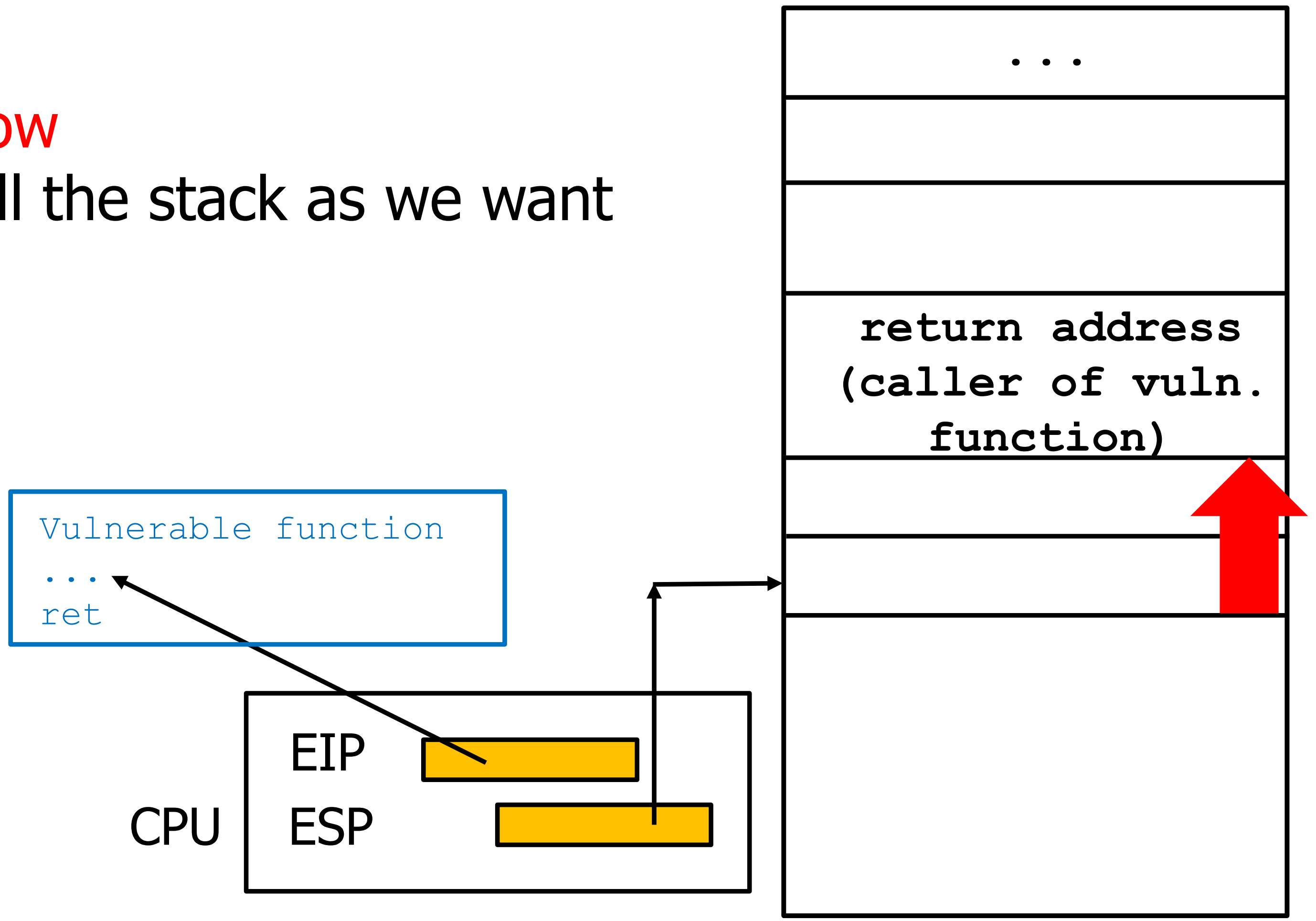


- Analyze `libc`

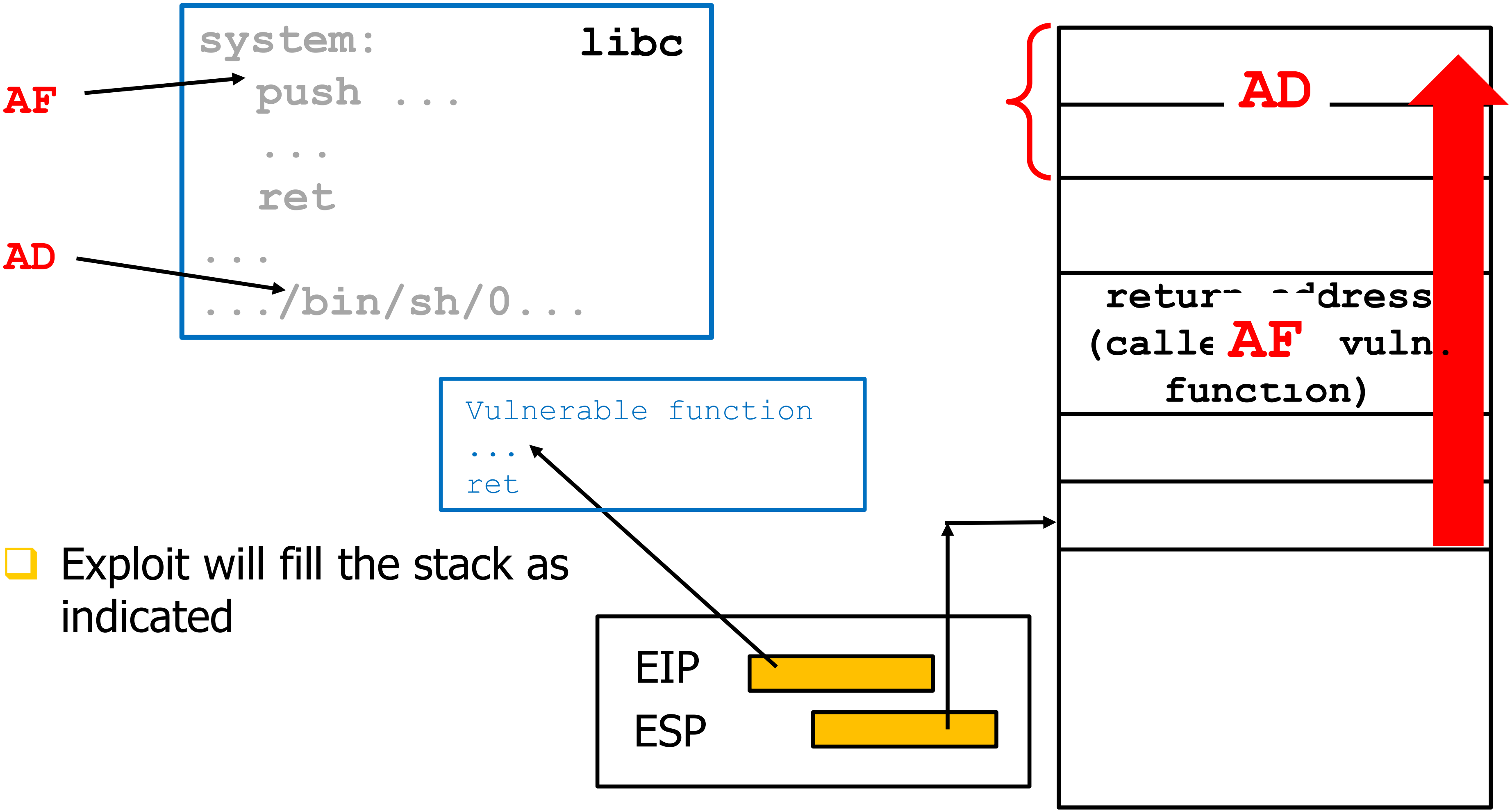
- Determine address of `system` function (**AF**)
- Search `/bin/shNUL` and determine its address (**AD**)
  - If this string does not exist in `libc` (unlikely) then another string must be used

Vulnerable function (I-a)

- Stack overflow
  ⇒ We can fill the stack as we want

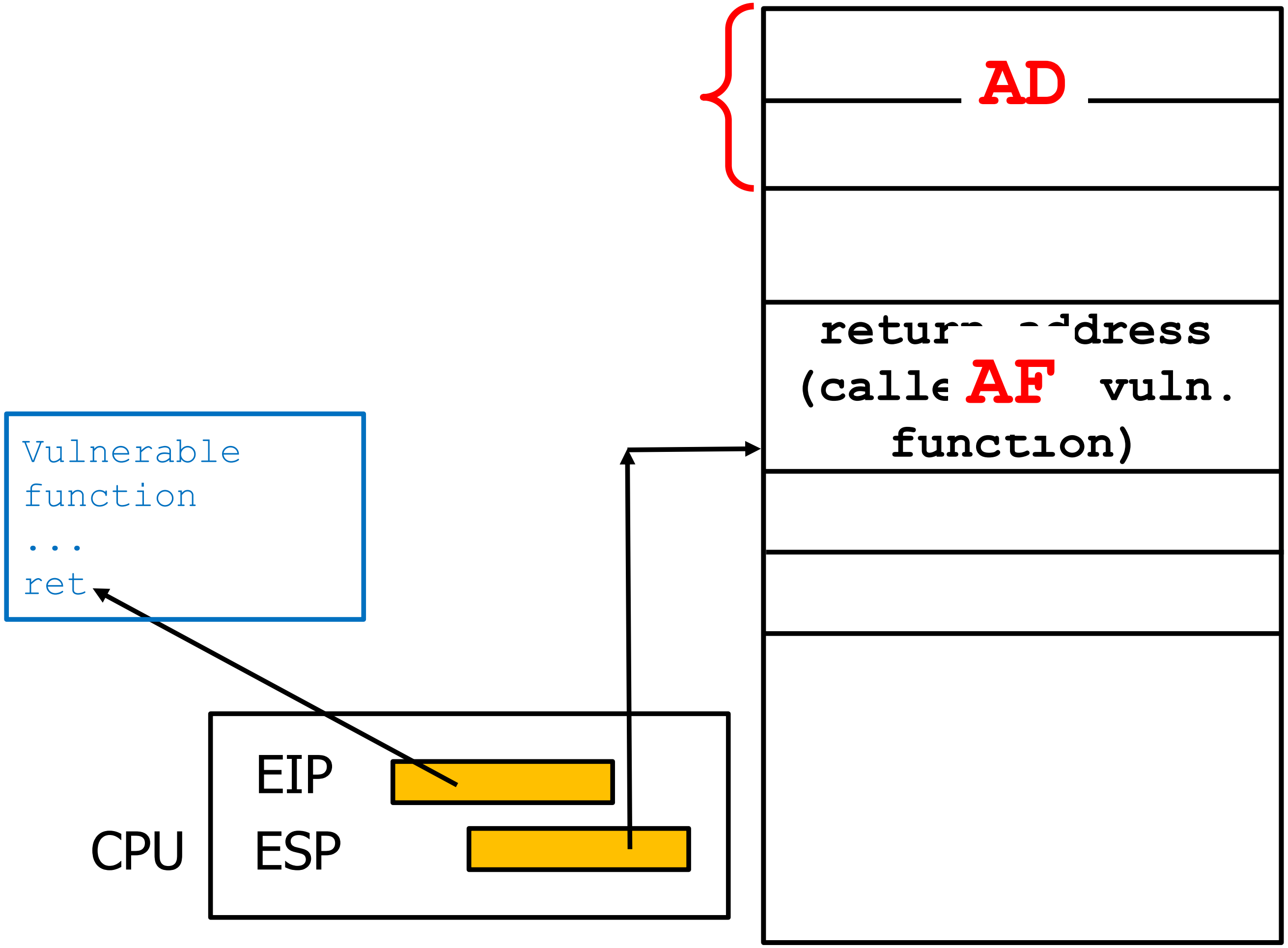


Vulnerable function (I-b)

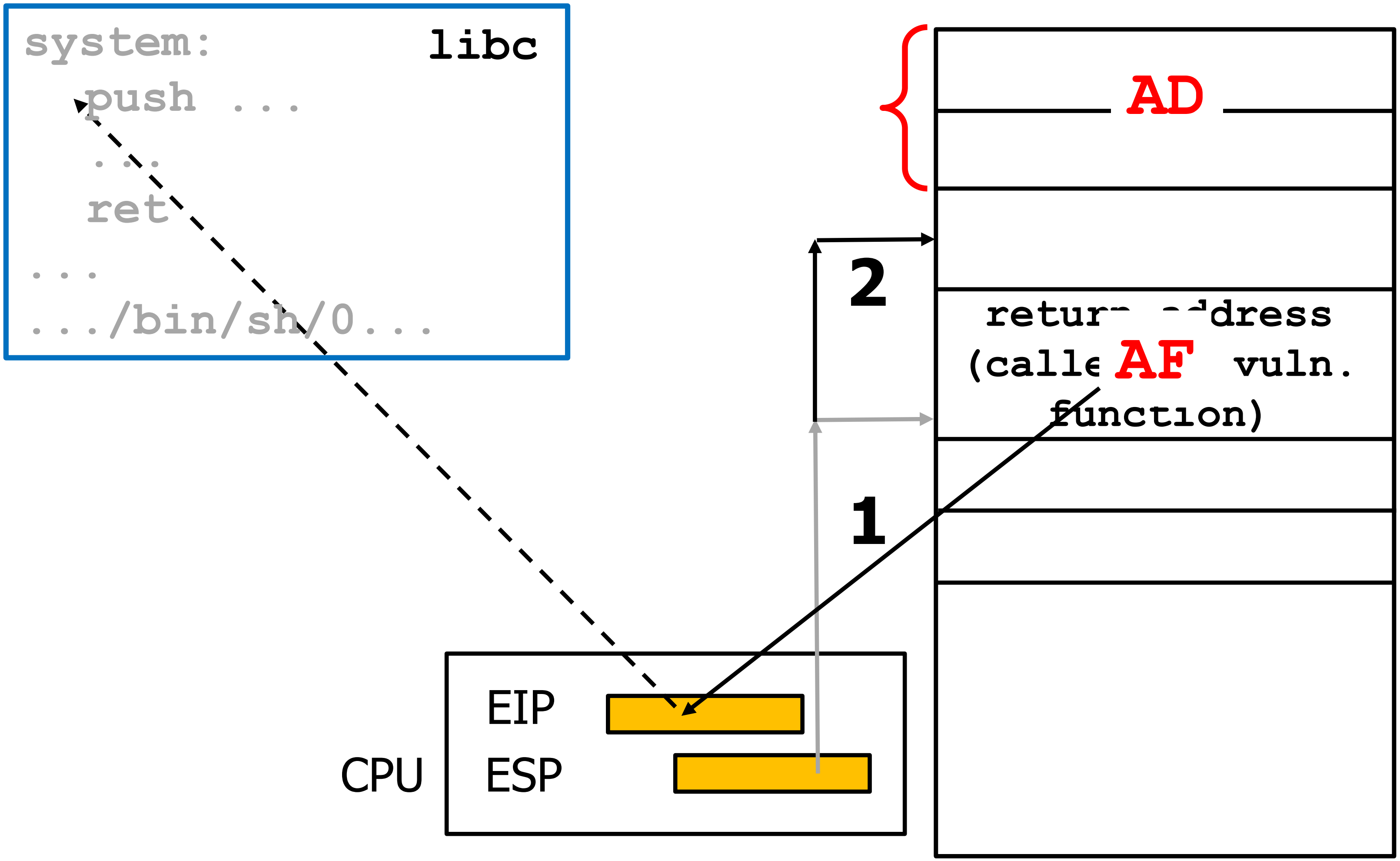


- Exploit will fill the stack as indicated

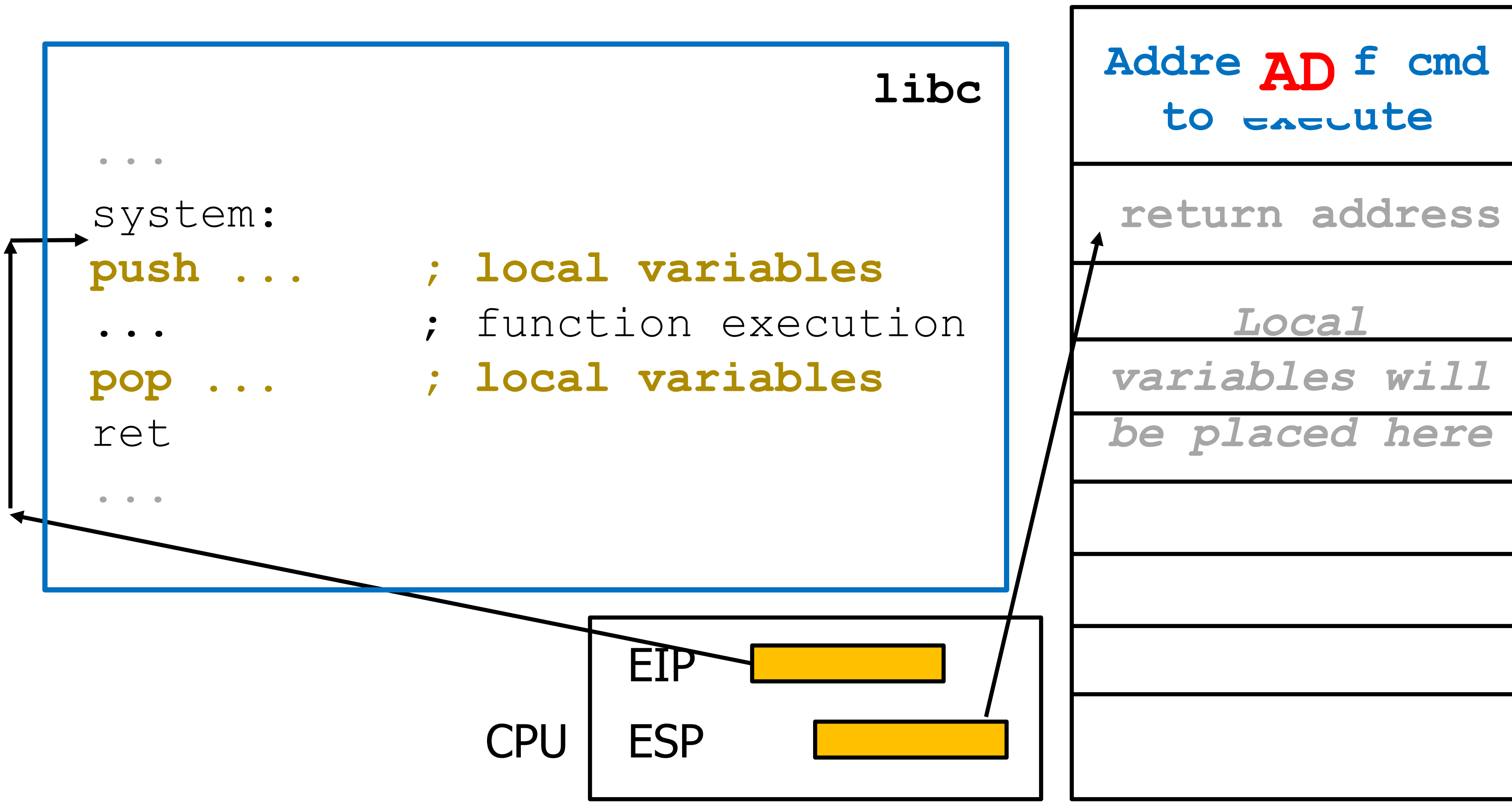
Last instruction of vuln. function: BEFORE



Last instruction of vuln. function: AFTER



BINGO!



libc
...
system:
push ... ; local variables
... ; function execution
pop ... ; local variables
ret
...
Addre AD f cmd
to
return address
Local
variables will
be placed here
EIP
CPU
ESP

Return to `libc` summary

1. Identify potentially useful **function** and **data** that **already exist** in memory (typically in `libc`)
2. Overwrite stack so that:
   - Vulnerable function returns to **existing function**
   - **Existing function** takes **existing data** as input argument

- Vulnerable function (stack overflow) returns
  - EIP will **not** go the Caller of the vulnerable function
- EIP will go the library function prologue
- ...with **input arguments on the stack**
  (as if the library function had been invoked as usual)

Circumvention IDEA: Code reuse: ROP (I)

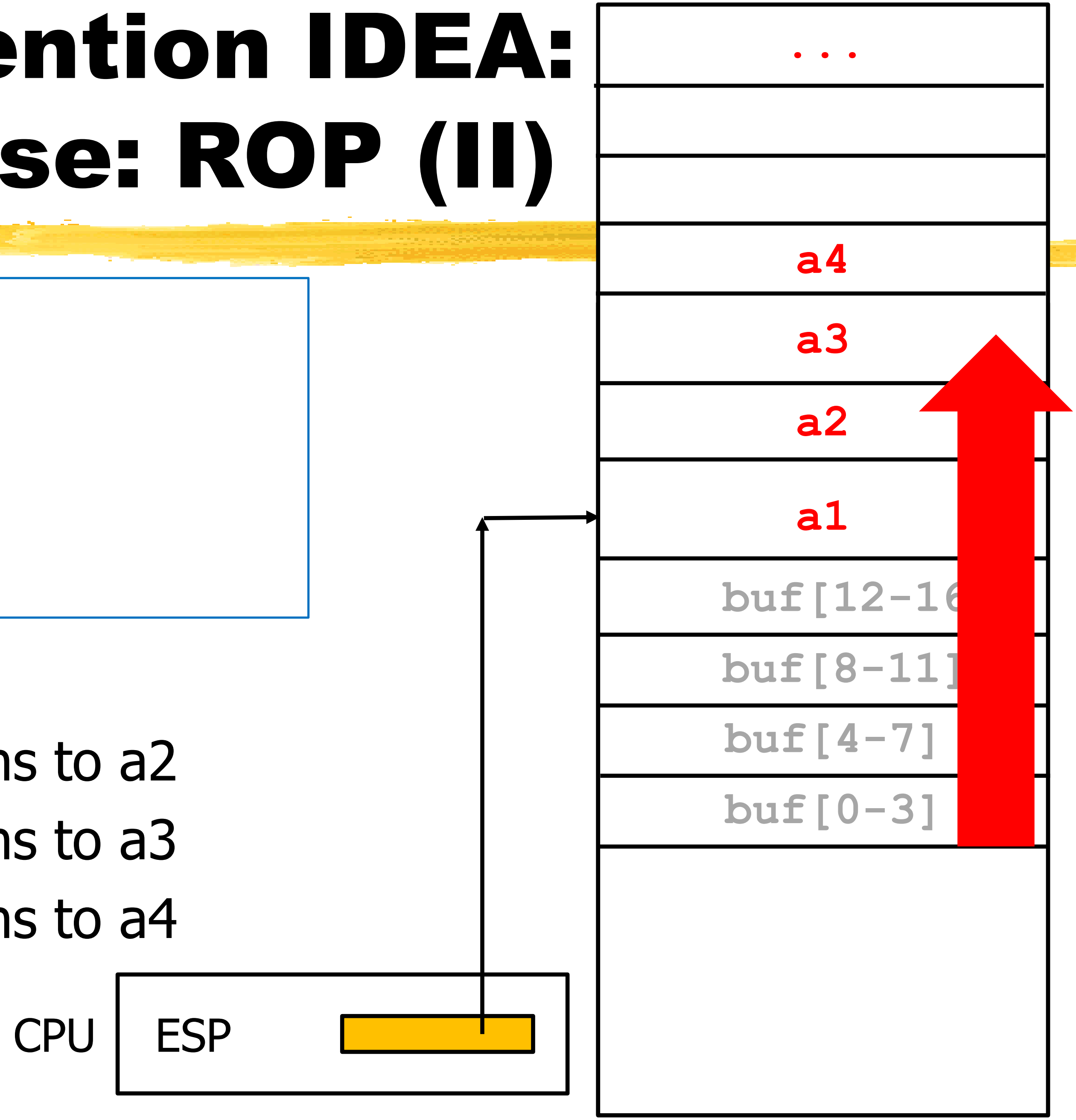
- **Return-oriented programming (ROP)**

- Identify potentially useful segments of code that already exist in memory and terminate with `ret` (**gadgets**)
  - "≈Library functions not from their beginning"
- Overwrite the stack so that:
  1. Vulnerable function returns to `gadget1`
  2. `gadget1` returns to `gadget2`
  3. `gadget2` returns to `gadget3`
  4. …

Circumvention IDEA: Code reuse: ROP (II)

```
void f(int x) {
   char[16] buf;
   gets(buf);
   ...
}
...
```

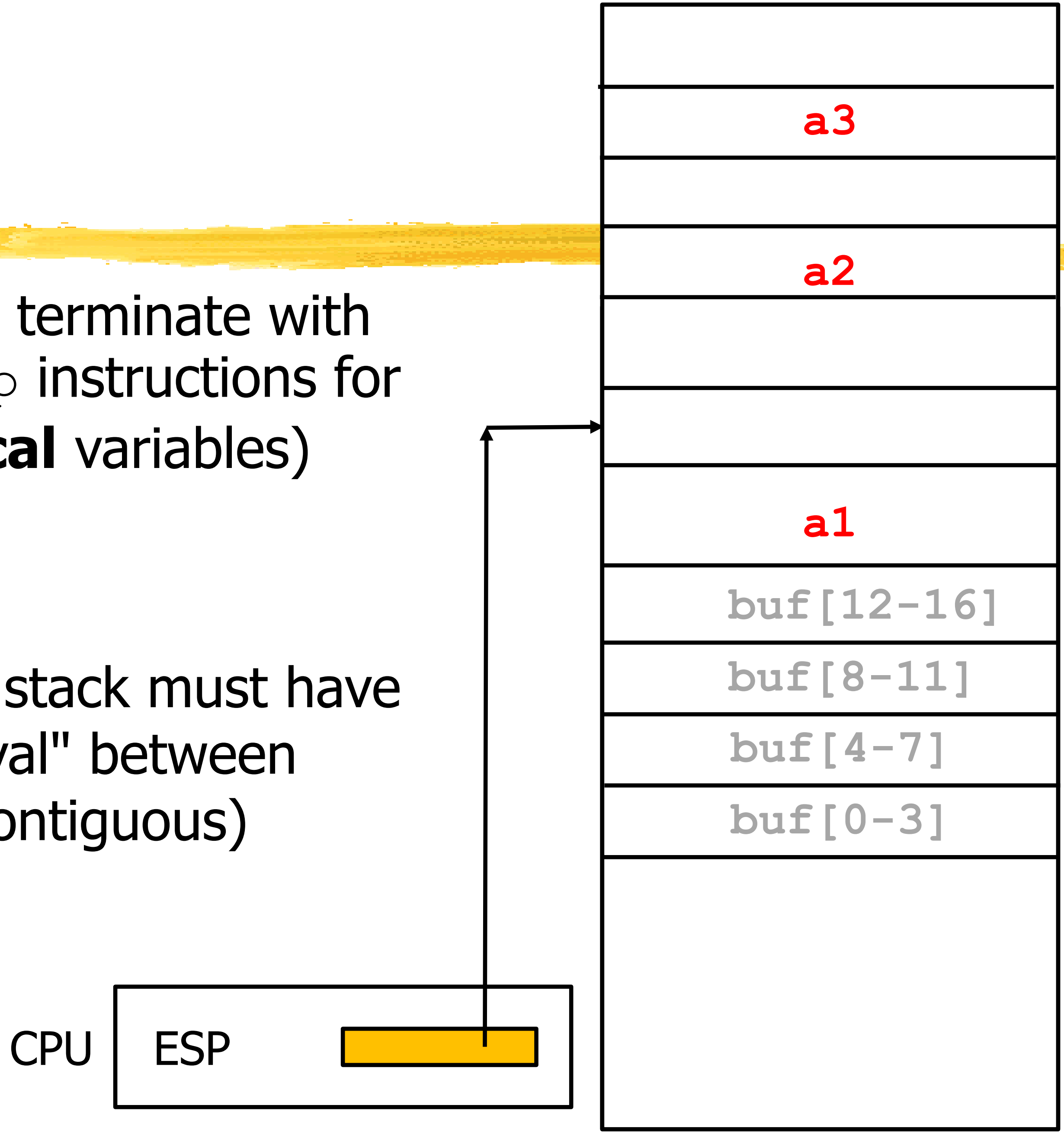
- `f` returns to a1
- Which then returns to a2
- Which then returns to a3
- Which then returns to a4



Remark

- Invoked functions terminate with an **epilogue** (`pop` instructions for dropping their **local** variables)

- Addresses on the stack must have the "correct interval" between each other (not contiguous)



Compiler-based Mitigations: Stack Canary

Compiler-based Mitigations

1. Stack canary
2. Pointer authentication

- Compiler support
  - Hw support for 2 necessary
- **Recompilation required**
  - Costly
    - All libraries and executables have to be recompiled
    - ...and **redistributed**

Stack Canary: When it works

- Effective on vulns that:
  - Write to **consecutive** and **increasing** addresses on the **stack**
  - **Very common** (overflow on local variables)

- **Not** effective on vulns that:
  - Write to memory in other ways (and possibly at attacker-chosen positions on the stack)

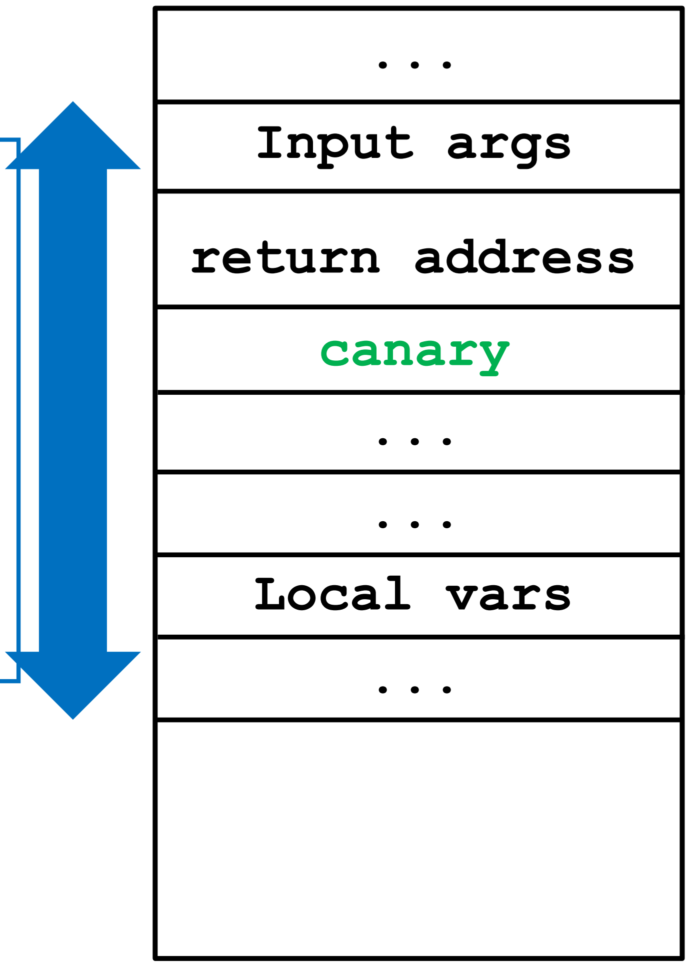
Stack Canary: How it works (I)

- Code generated by the **compiler**

- When a program starts:
  1. Generate a **random** value (**canary**)
  2. Store it at a predefined position on the stack
- Every function **prologue**:
  - Insert canary value on the stack
- Every function **epilogue**:
  - Compare canary value on the stack to expected value
  - If different then crash

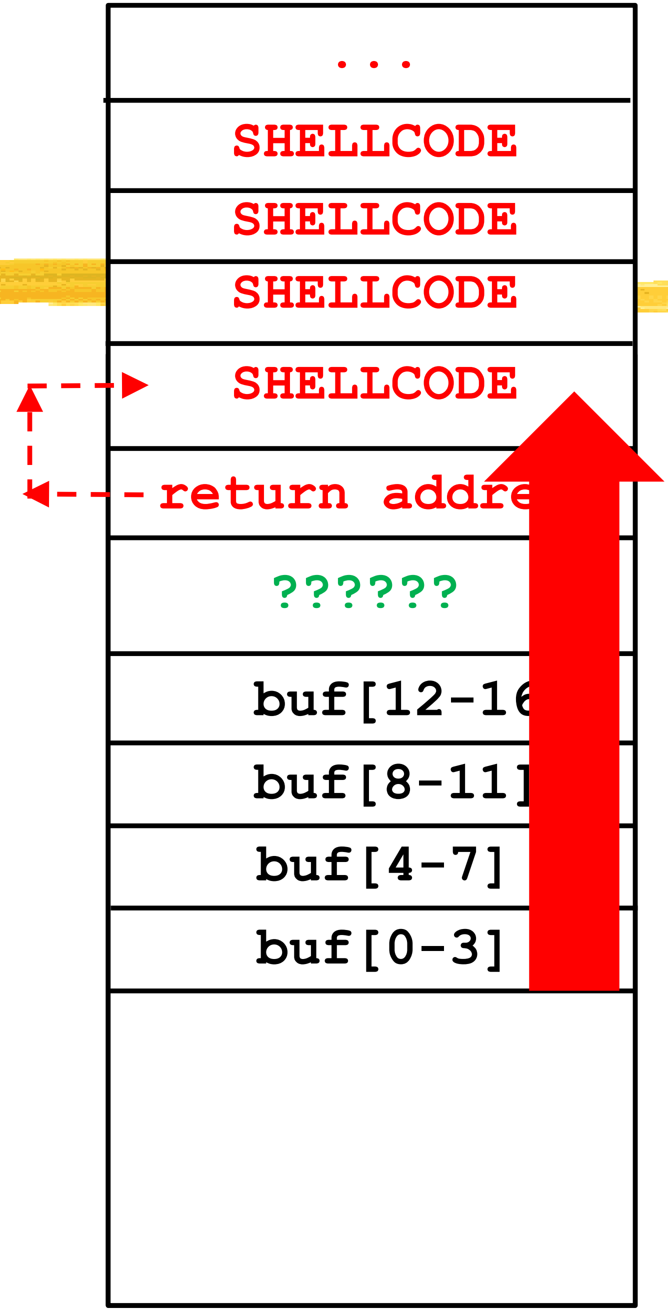
Stack Canary: How it works (II)

```
callee:
push ...      ; push canary@stack
push ...      ; local variables
...           ; function execution
pop ...       ; free local variables
...     ; if canary@stack <> canary
...     ; then jmp crash
ret
```



Stack Canary: How it works (III)

```
callee:
push ...        ; push canary@stack
push ...        ; local variables
...             ; function execution
                  (overflow!)
pop ...         ; free local variables
...     ; if canary@stack <> canary
...     ; then jmp crash
ret
```



- Overwriting return address requires **overwriting the canary**
- ...but the exploit cannot know its value!

Circumvention IDEA

- **Guess** the canary value
  - Repeat injection for every possible canary value
  - Feasibility depends on range size
  - First byte of canary is always '\0'
    (to mitigate possible string-based attacks)

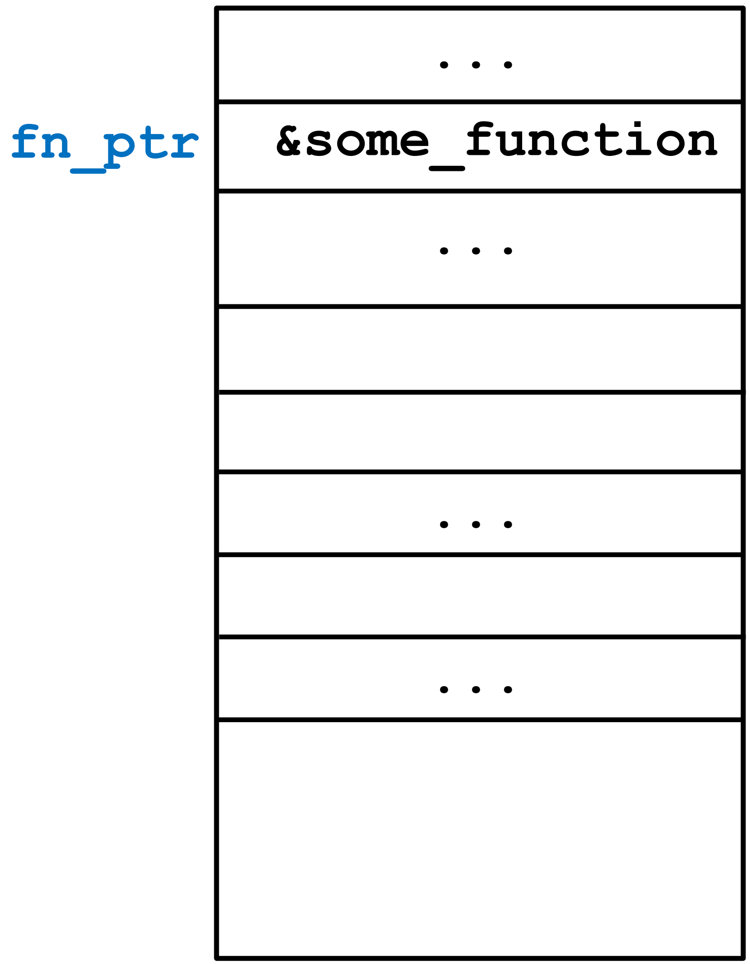
- **Leak** (and then use) the canary value
  - Exploit vulnerabilities that allow **reading** the full stack

Compiler-based Mitigations: Pointer Authentication

Function Pointers (I)

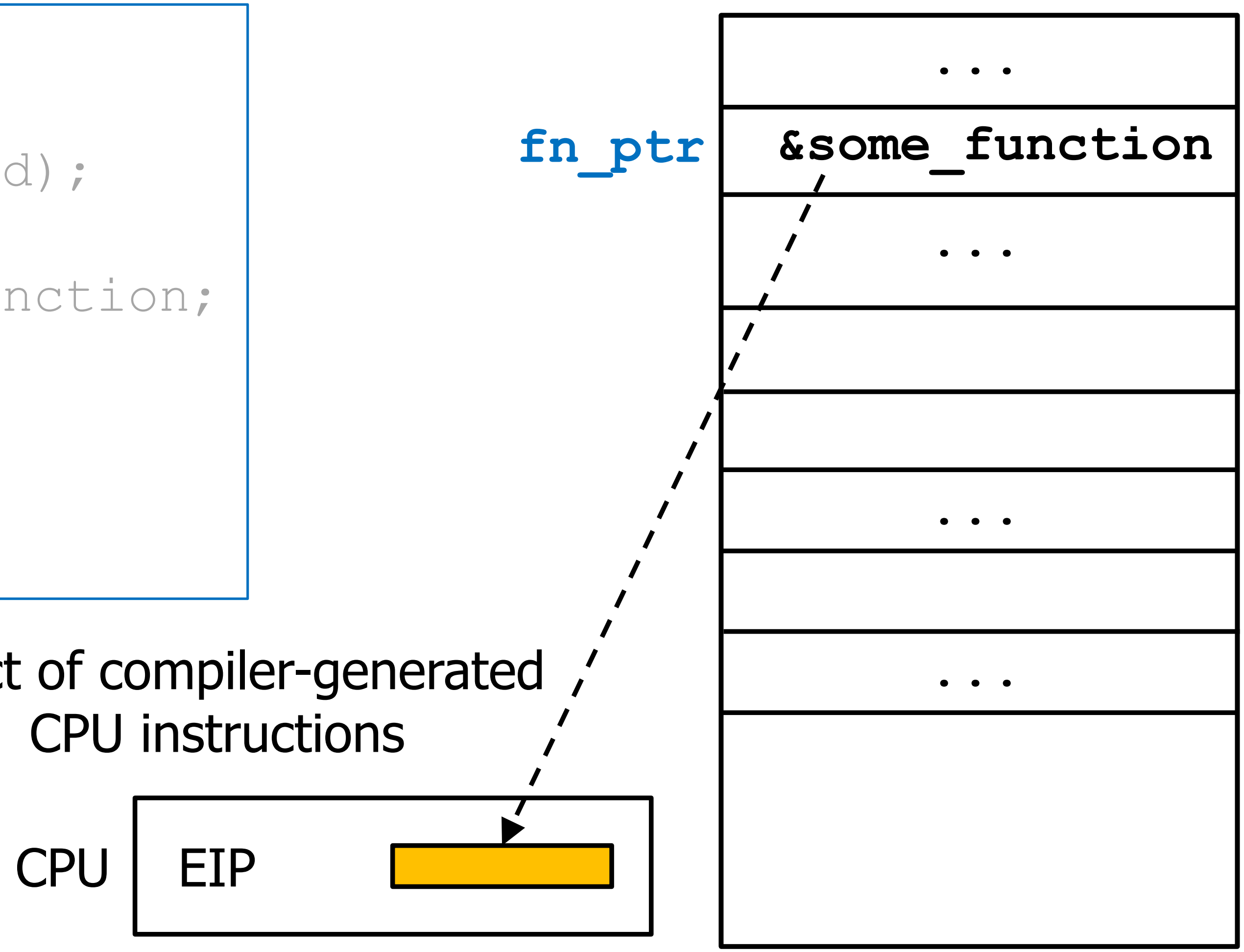
```
...
...
int (*fn_ptr)(void);
...
fn_ptr = &some_function;
...
...
a = (*fn_ptr)();
...
```

Common pattern



Function Pointers (II)

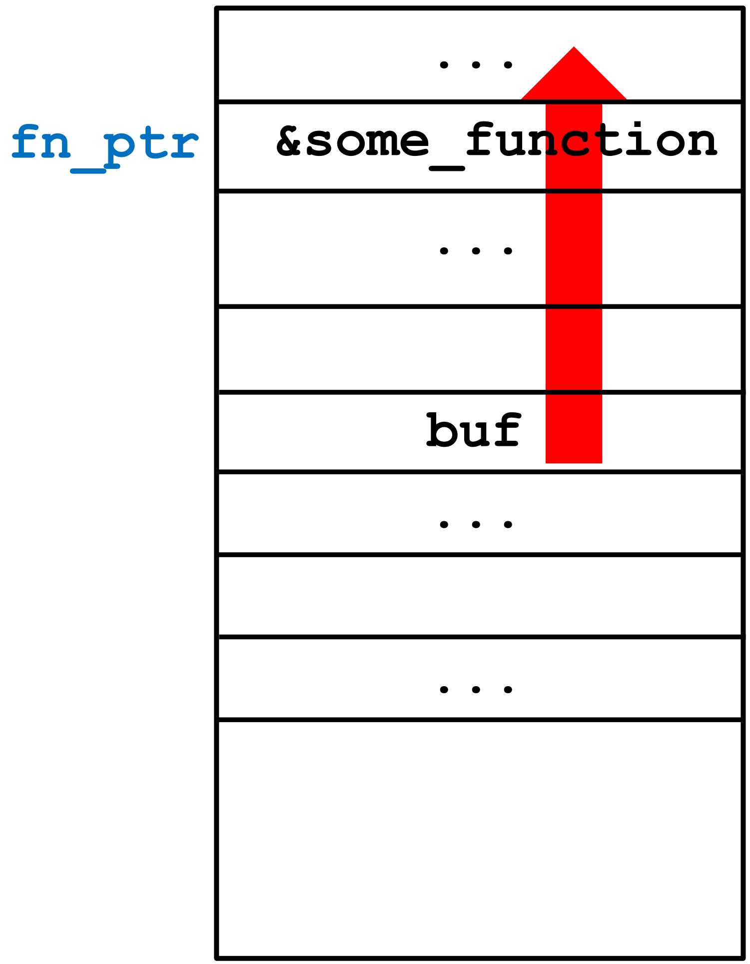
```
...
...
int (*fn_ptr)(void);
...
fn_ptr = &some_function;
...
...
a = (*fn_ptr)();
...
```



Function Pointers: Overwriting

```
int buf[10];
...
int (*fn_ptr)(void);
...
fn_ptr = &some_function;
...
...
a = (*fn_ptr)();
...
```

Overflow triggered here

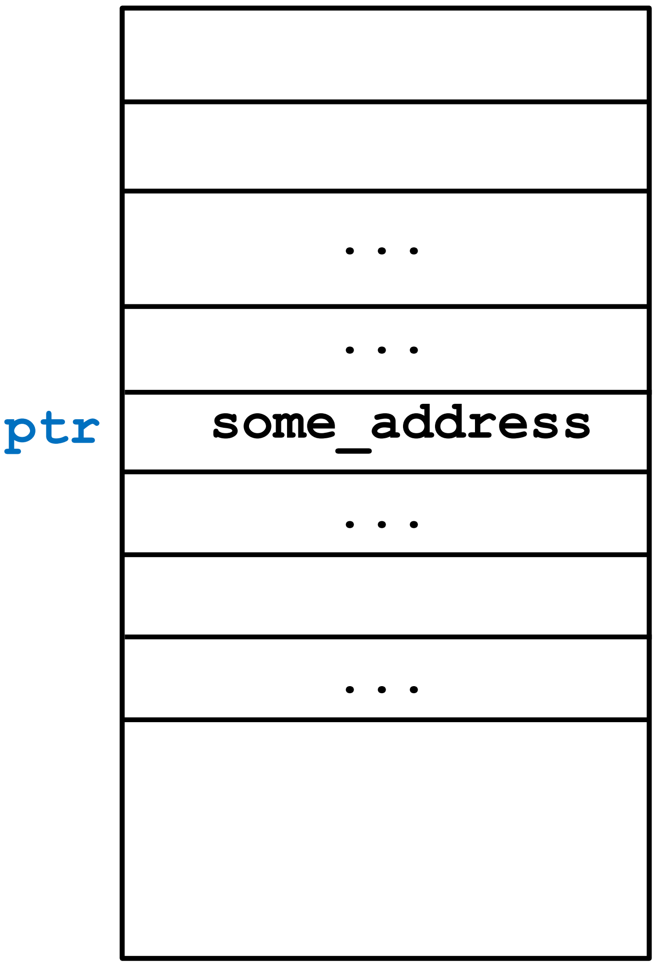


Execution flow diverges to **attacker-controlled address**

Data Pointers (I)

```
...
...
int *ptr;
...
ptr = &some_address;
...
...
*ptr = some_val;
...
```

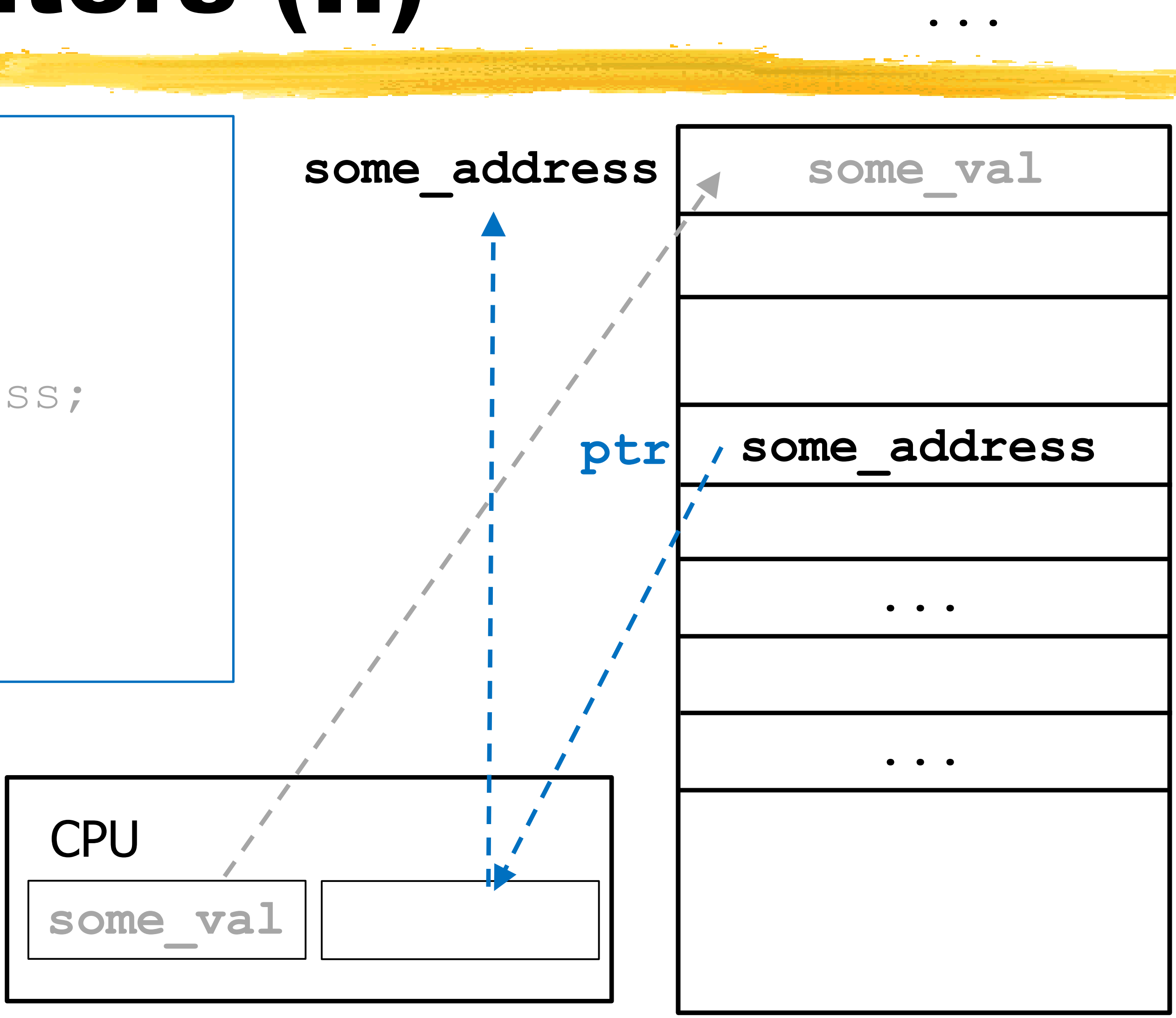
Common pattern



Data Pointers (II)

```
...
...
int *ptr;
...
ptr = &some_address;
...
...
*ptr = some_val;
...
```

Effect of
compiler-generated
CPU instructions

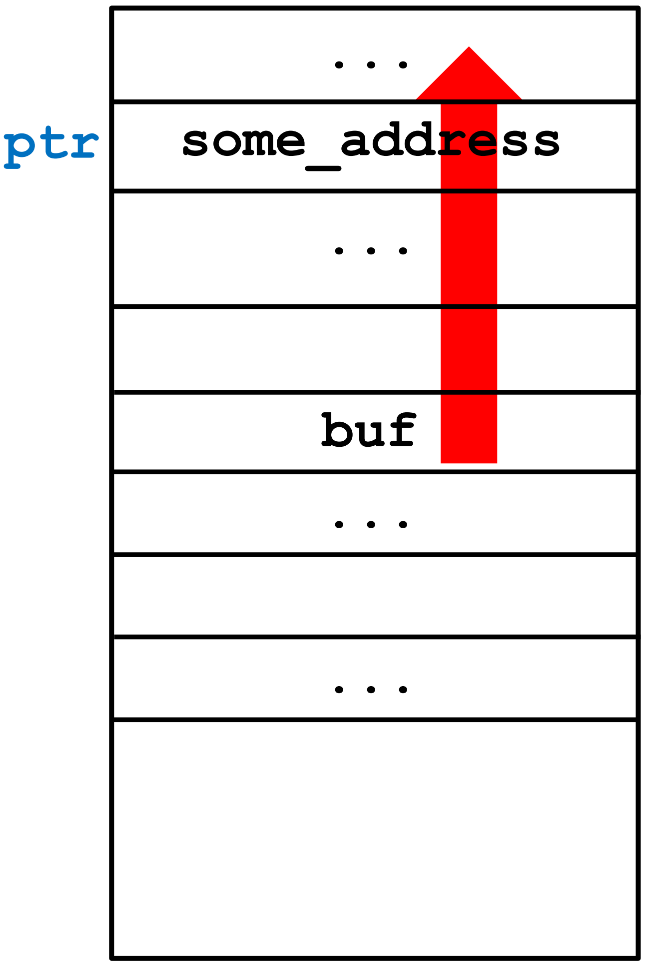


Data Pointers: Overwriting

```
int buf[10];
...
int *ptr;
...
ptr = &some_address;
...
...
*ptr = some_val;
...
```

Overflow triggered here

`some_val` written to
attacker-controlled address



Fact

Overwriting pointer values in memory
is a **crucial** step of (almost) every exploit
for memory safety vulnerabilities

Modern Architectures

- Modern CPU architectures:
  - Process memory N1= 64 bits
  - **Physical** memory N2 **< N1** bits
    - N2 = 48 ⇒ 2^16*2^32 = 64K * G
  - Mapping from **virtual** memory (of each process) to **physical** memory done by hw+o.s.

- Every address in a program has **more bits than necessary**

Pointer Authentication Code (PAC) – SIMPLIFIED (I)

- Several (modern) CPUs have:
  - Secret key `K` in a protected CPU register
  - CPU instruction for
    - **Signing** its operand with `K`
    - **Verifying** signature of operand with `K`

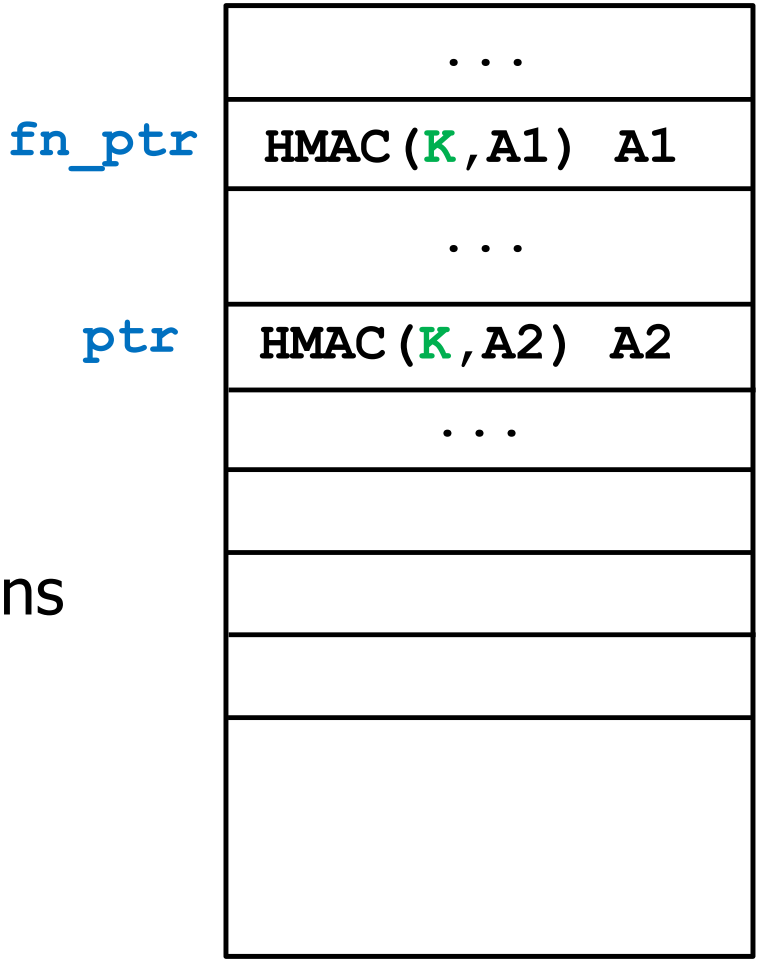
  - Signature stored in N1-N2 most significant bits of operand
    - Before signature `V`
    - After signature `HMAC(K,V) V`

Pointer Authentication Code (PAC) – SIMPLIFIED (II)

- Compiler behavior:
  - **Write pointer** to memory
    - Sign pointer
    - Store signed pointer in memory
  - **Read pointer** from memory
    - Load pointer from memory
    - Verify signature
    - If not valid then error

Writing Pointers to Memory

```
...
fn_ptr = &some_function;
...
ptr = &some_address;
...
```

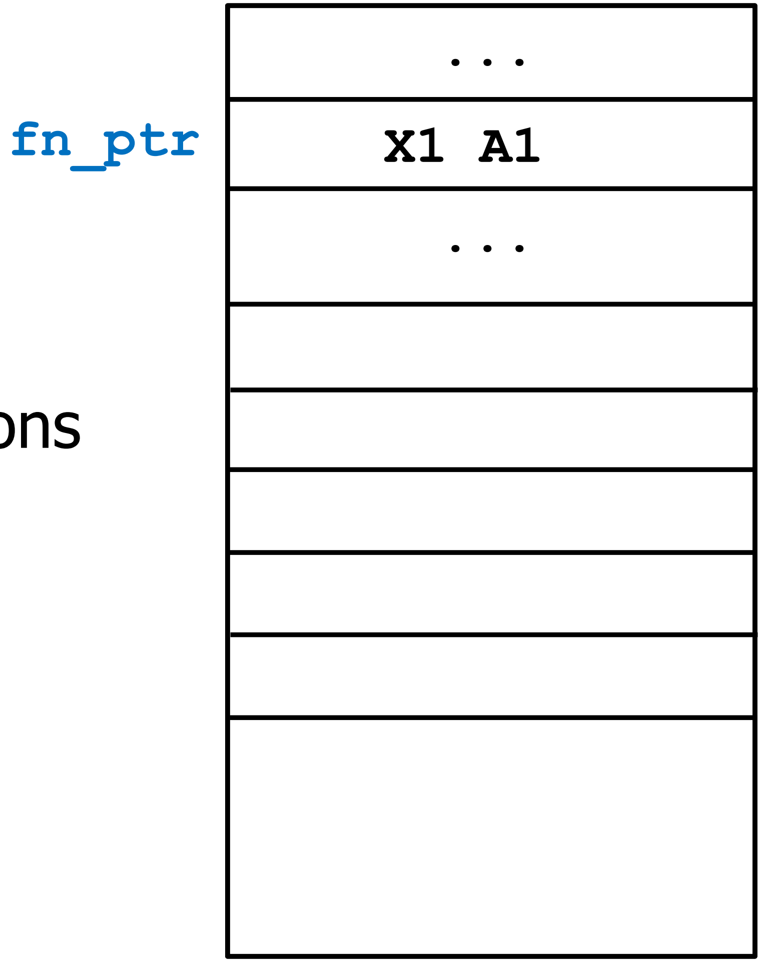


Compiler-generated CPU instructions

- ❑ Sign pointer
- ❑ Store signed pointer in memory

Reading Pointers from Memory (I)

```
...
result = (*fn_ptr)();
...
```

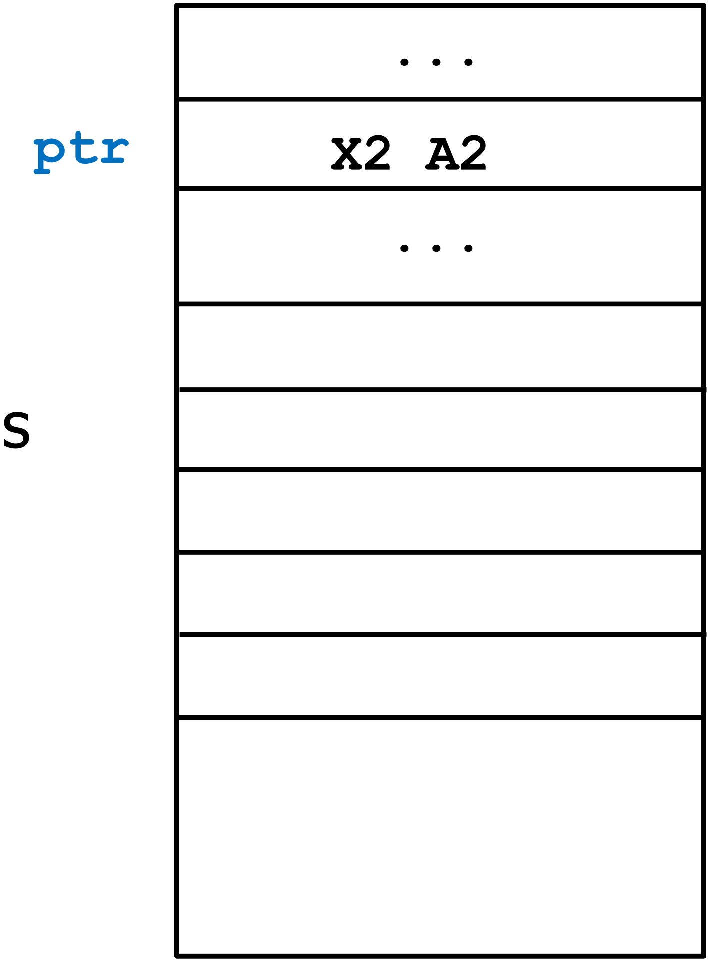


Compiler-generated CPU instructions

- Load pointer from memory
- Verify signature
- If not valid then error
- `call A1`

Reading Pointers from Memory (II)

```
...
*ptr = some_val;
...
```

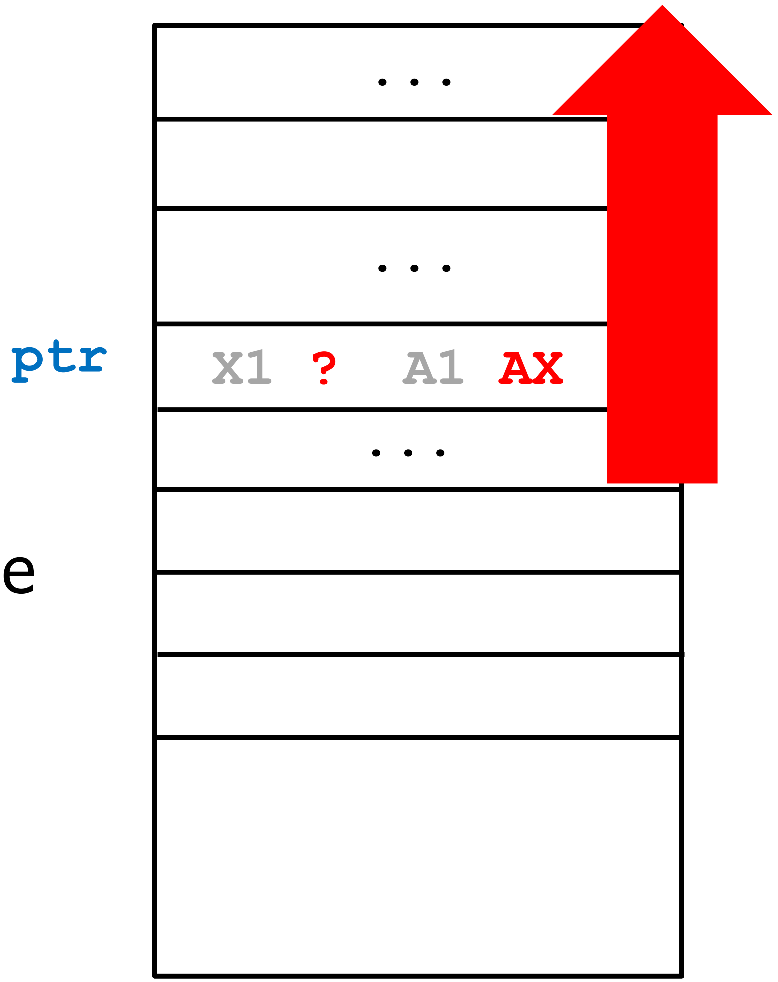


Compiler-generated CPU instructions

- Load pointer from memory
- Verify signature
- If not valid then error
- `some_val → address A1`

Defensive Power

- Overwriting `A1` with `AX` requires writing `HMAC(K,AX)`
- The exploit cannot "know" this value



Circumvention IDEA

- PAC() generation is **deterministic**:
  - Force CPU to generate PAC() for addresses of choice and **reuse** them
  - Copy PAC() generated by the CPU and try to **reuse** them

- **Brute force**
  - It may or may not be possible (it depends on key length / PAC length)

- **Vulnerability** that forces CPU or o.s. to **expose** key

Mitigations In Practice: Final remarks

Defense in depth

- Non-executable pages
- Address Space Layout Randomization (ASLR)
- Stack Canary
- Pointer authentication

- Excellent example of **defense in depth** (**multiple** and **independent** layers)
- Bypassing a layer **does not save the effort** of bypassing the next layer

- More defenses exist

Usage (I)

- Available on most modern platforms
- Compiler flags / O.S. flags

- **"Stack corruption is essentially dead" (Microsoft 2019)**
- Other memory unsafety (and language-based) issues **are not**

Usage (II)

- Pay attention to the **default**!

- **Cisco** Adaptive **Security** Appliance (ASA) 2016
  - Authenticated remote code execution
  - Two buffer overflows
  - No non-executable pages
  - No ASLR
  - No stack canary

- October 2025: have a look at yIKES in the companion website!

BIG headache

- Available on most modern platforms
- Compiler flags / O.S. flags

- **IoT / embedded systems** often do **not** have key mitigations
- ...and run memory-unsafe code

- Writing exploits for those platforms tends to be easy