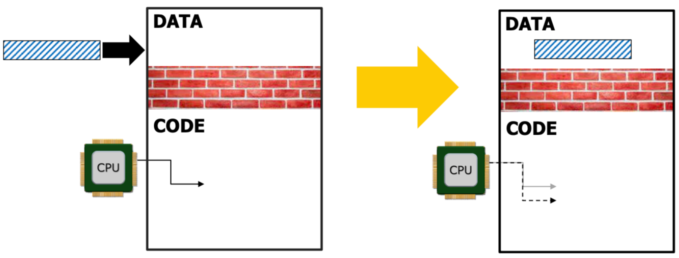
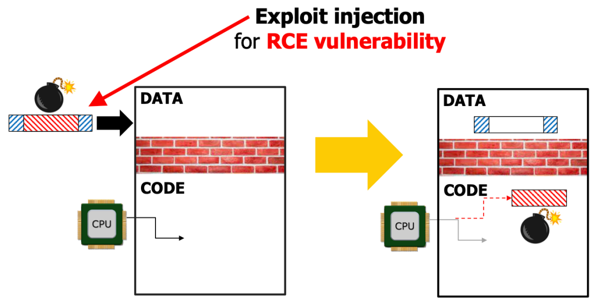
Memory corruption

- Remind on RCE vulnerability:
  - Remote command execution: Attacker can execute any action from remote, the only constraint is the privilege level of the vulnerable program
  - Any action can be done: word could start encrypting your disk, powerpoint could launch a remote shell server, a web server could create a new user …
- How is that done?
- This is what should always happen   this is what happen when there is an RCE vulnerability





- Payload: ciò che c'è dentro l'exploit, that is injected in the vulnerable program

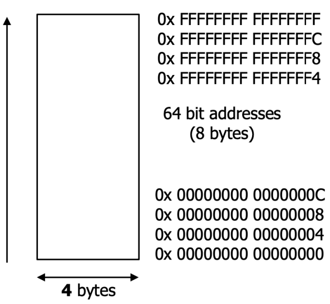
---

Memory corruption

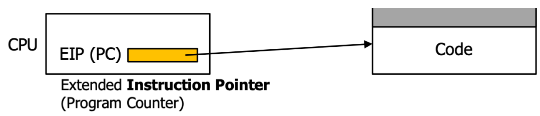
- Memory corruption bug: program accesses memory incorrectly
- Most common example is the out of bounds write (buffer overflow)
- Memory corruption bugs can be exploited by attackers (vulnerability): alter the program behavior, take full control of program
- It is one of the oldest problems in computer security
- Memory corruption vs memory safety:
  - The terminology that is becoming prevalent is memory safety:
    - A memory safe language is a language that prevents memory corruption bugs
    - Memory safety bugs are the bugs that occur because the memory is non accessed safely
  - Basically they are the same concepts (we will use memory corruption bug and memory safe (or memory unsafe) language)
- Basic fact: more than 70% of vulnerabilities are caused by memory corruption (statistic and estimates), in all the platforms
- Out of bounds write is the number one raked in the CWE top 25 of 2022
  - The key reason (and current trend) is that most memory corruption vulns are consequence of usage of memory unsafe programming languages (C/C++), there is a strong trend toward abandoning memory unsafe languages and switch to memory safe languages (rust, go, python, java)

Memory management pt.1

- Address space



-Code: the program code itself (also called text)

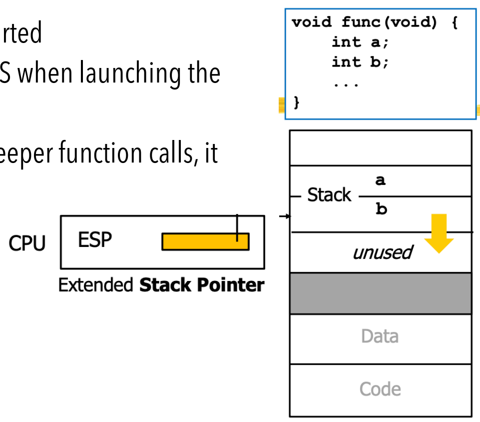


-The CPU register EIP: address of the next instruction to be executed

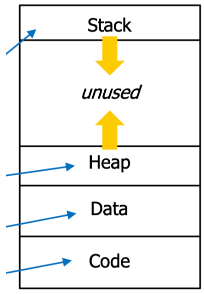
-Data: static variables are those that exist for the entire lifetime of the program, they are allocated when the program is started

-Remark: the starting address of code is not 0, it is chosen by the OS when launching the program

-Stack: contains the local variables and, as you make deeper and deeper function calls, it grows backwards (CPU registers EPS, address of the top of the stack)



- Heap: dynamically allocated memory (C language malloc and free, in modern languages there



Memory corruption

are objects), as more and more memory is allocated, it grows upwards

- The address space of a running program is subdivided in 4 regions
  - First and very rough approximation: there are local variables and function arguments in the stack, then dynamically created variables (objects) in the heap, global variables in data and program instructions in code
- Heap vs stack (?) non so a cosa serva

Memory corruption vulnerabilities

- Threat model: **vulnerability DISCOVERY**
  - Attacker: has a target locally available, 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 remind: is the same sequence of bytes for all instances of the vulnerable program

-Memory corruption bug example: overwrites a memory region in the data area

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

- Overwrites a memory region in the stack

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

-Overwrites a memory region in the heap

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

- gets():
  - Get a string from standard input (DEPRECATED)
  - 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. There is not check for buffer overrun is performed
  - Never use this function, this library function is deprecated, it is only to use for illustrators purposes
- Bug → Vulnerability
  - Memory corruption bug that is also a vulnerability, in this case the bug is provoked by gets
  - A seguire: three different possible impacts for the same bug
    - Access control bypass
    - Configuration change (access control bypass)
    - Command injection
- Threat model: Vulnerability EXPLOITATION
  - Attacker inject input of his choice to vulnerable program
  - Many possible scenarios that depend on the specific vulnerability:
    - Network message (remote injection)
    - File downloaded from attacker controlled location (remote injection)
    - File at some position in local filesystem (local injection)
    - Environment variable of shell (local injection)
  - Non ci interessa (?)
- Hypothetical example →

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

Memory corruption

- Access control bypassed:

```
. . .
char name [20];                    //if input contains more than 20 bytes, the
int authenticated = 0;             //input overwrites authenticated with
. . .                              //arbitrary value chosen from outside
void vulnerable {
      . . .
      gets(name); // reads from input until '\n'
      . . .
}
```

- hypothetical example 2 →
- Configuration change (access control bypass)

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

```
char name[20];
char dns_address = "8.8.8.8";
. . .
int setConfiguration(. . .){
      /* use dns_ */
}
. . .
void vulnerable {                                   //if input contains more than 20
      . . .                                         //bytes before '\n', input
      gets(name); // reads from input until '\n'    //overwrites dns_address with
      ...                                           //arbitrary value chosen from
}                                                   //the outside
```

- Hypothetical example 3 →
- Command injection

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

```
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', input overwrites cmd with arbitrary value chosen from the outside

- Impact: depend on program structure and vulnerability, not chosen by the attacker arbitrarily
- Remark:

```
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

    → the vulnerability may or may not be practically exploitable (ricordiamoci che solo un sottoinsieme dei bug sono effettivamente delle vulnerabilità, a loro volta solo un sottoinsieme delle vulnerabilità è effettivamente exploitable
- Useful point of view: exploitation based on write attacker-controlled value at attacker-controlled location
  - Access control bypass: alter control flow (if-then-else)

Memory corruption

- Configuration change (access control bypass): alter configuration
- Command injection: alter invocation parameters

Buffer overflow

- It is a 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 the size of the destination buffer)
- The solution is use library function that never overflow destination buffer
  - Files: `char *fgets(char *str, int n, FILE *system)`
  - Sockets: `size_t recv (int sockfd, void *buf, size_t len, . . .);`
- It is a very optimistic assumption: if we use input libraries that never overflow destination buffer, we are not safe from buffer overflows anyway
- Fact 1: every input could be provided by an attacker, every program has some variables whose values derive from (part of) some input

→ every program has some attacker-controlled variable

  - It is not a problem in correct programs, it has nothing to do with buffer overflows, it is true in every programming language

- Fact 2: buffer overflows may occur on destination buffer very far away from input operations because of source buffers that are attacker-controlled (many possible causes)

```
...
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,...);
...
```

- Attacker may **indirectly** control **many** variables
- Dependency chain potentially "very long"

- Example

```
. . .
char dst[64];
. . .
// buf attacker - controlled
strcpy(dst, buf);
. . .
```

- if the attacker provide `strlen(buf) > 64` → buffer overflow when writing on `dst`, it may be overwrite something useful

- A good practice to prevent overflowing `dst`

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

- Not easy:
  - Bug `strlen(to_be_copied_to_d) > 32K` → len takes a negative value (integer overflow), if the if condition is satisfied → buffer overflow when writing on d

- Safe string functions:

```
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 do not overflow

- So, if we only use input libraries that never overflow destination buffer and string libraries that never overflow destination string, are we safe from buffer overflows? NO
- Bug: `strlen(base_url) > LEN` → `dst` non null-teminated

→ buffer overflow when writing on path

→ out of bound read from `dst`

Impact: information disclosure

```
#define LEN 64
char dst[LEN];
...
// base_url attacker-controlled
strncpy(dst, base_url, LEN);
...
char path[LEN];
...
memcpy(path, dst, strlen(dst))
...
```

Memory corruption

- Safer string functions

```
size_t → strlcpy(char *dst, const char *scr, size_t size);
size_t → strlcat(char *dst, const char *scr, size_t size);
```

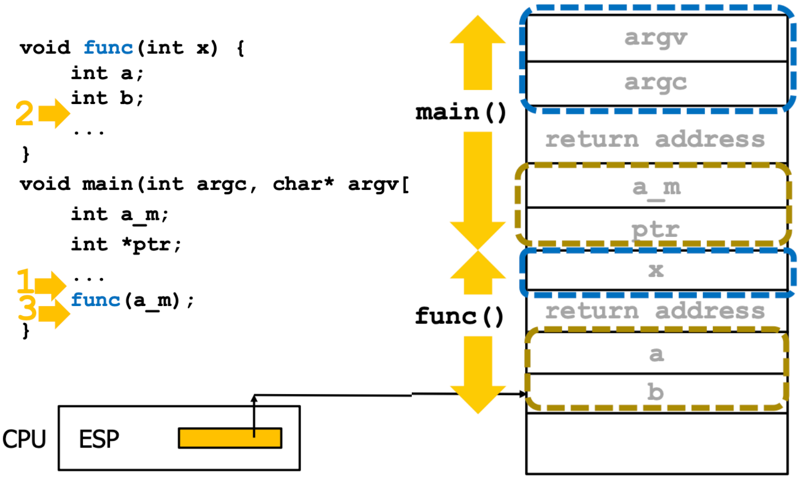
  - They are designed to be safer, more consistent and less error prone replacements for strncpy(3) and strncat(3)
  - There guarantee to NUL-terminate the result
  - Note that a byte for the NUL should be included in size
  - Note that fro strlcpy() src must be NUL-teminated and for strlcat() both src and dst must be NUL-terminated
- So, if we only use input libraries that never overflow destination buffer, string libraries that never overflow destination string and that always terminate destination string, are we safe from buffer overflows? NO
- Example

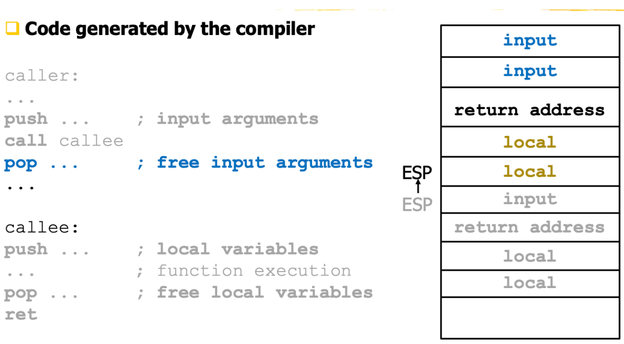
```
. . .
int* ptr = malloc(1000);
// idx is attacker-controlled and negative
prt[idx] = val;
. . .
```

- Memory corruption bug, may or may not be a vulnerability
- Real examples:
  - Talos vulnerability report
  - ListServ 2024
  - Takeaways
    - Unsafe libraries can and should be replaced, but
      1. It is not a complete solution: memory safety bugs may be in our code (not only in libraries)
      2. Lot of existing code has to be modified, tested and shipped, never forget economics (who knows about these problems? Who has sufficient incentives to fix them?
    - Making sure that input never overflows is not enough for preventing buffer overflows
      - There are many other risky operations like string processing and many others
    - Exploitation based on write attacker - controlled value at attacker - controlled location, may happen potentially anywhere (very far away from input operations)

Memory management pt.2

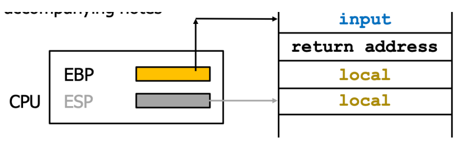
- Stack: remind
  - Contains local variables and as you make deeper and deeper function calls, it grows downwards
  - CPU register ESP (extended stack pointer), address of the top of the stack
- What is on the stack? Input arguments + local variables
- The stack is organized in stack frames, one for each function invocation
  - Lifetime: frames created upon the invocation, destroyed upon the return
- Who manages the stack frames?





-Every invocation prepares input args before and frees them after, every function has a prologue and an epilogue

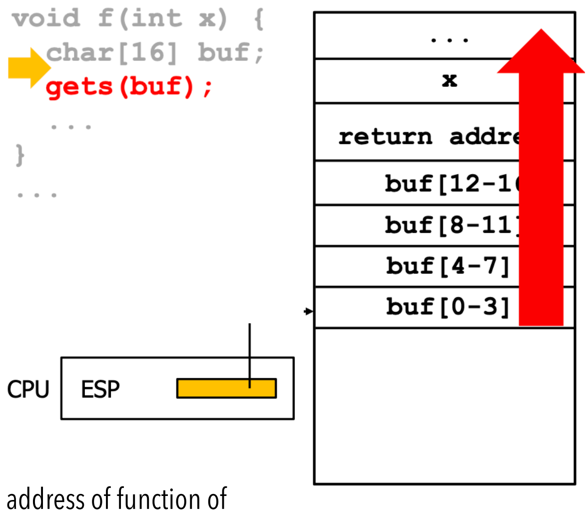
-Remark: modern architectures have an additional CPU register (EBP) for pointing to the base of each stack frame, real details slightly more complex to understand



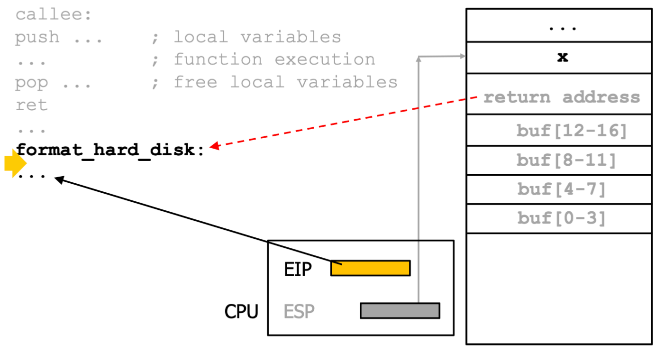
Memory corruption: stack smashing

- Hypothetical example →
    - Attacker can overwrite starting from the beginning of `buf[]`
    - He can control the return address
- **Exploit: code reuse**

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



-The attacker determines address of function of interest, overwrites return address with that address

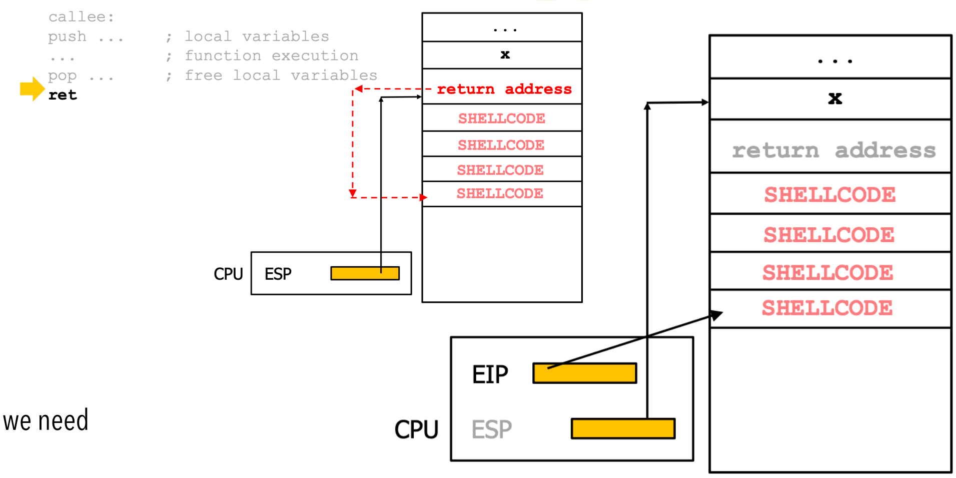


- **Exploit: code injection**

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

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

- What if 16 bytes are not enough for the shellcode od interest?
    - No prob: we can overwrite (almost) how much we need



Memory corruption recap

- Memory corruption bug: program access memory incorrectly, those bugs can be exploited by attackers (vulnerability). It is one of the oldest problems in computer security
- Number one weakness in C/C++
    - These languages are not memory safe, programmer is responsible for memory management
    - Typical cause are arrays, pointers, strings, dynamic memory. Tricky to spot and prevent
- Out of bound write is just one of several memory corruption issues: out of bounds read, use after free, format string attack, …
- More insecurity: there are many additional sources of insecurity (remember example of integer overflow)
- 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 and index out of bounds exception instead), strings not having a null terminator (because terminators would be inserted by the compiler/interpreter), integer overflow (because you'd get an integer overflow exception instead)

Memory corruption

- 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 costumers 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 subscripts errors occur o 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 law"

More implications of buffer overflow

- Buffer overflow on the stack: can be exploited systematically
    - Code reuse (overwrite return value) or code injection (overwrite shell code and return value)
- 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 of buffer overflow on the data

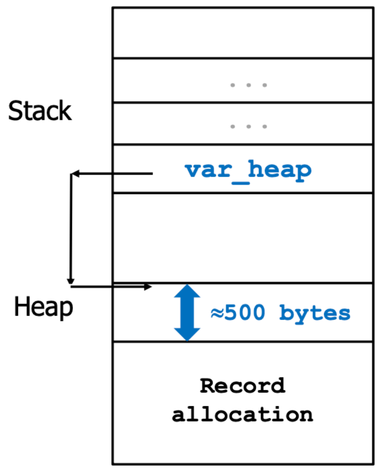
```
      char vect[. . .];
      . . .
      int(*fn_ptr)(void) {
      . . .
      int someFunc(. . .) {
            . . .
            a = (*fn_ptr) (); //invoke
pointed function
      }
      . . .
```

- 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 of buffer overflow on the heap

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



- **Exploitation by overwrite**
    - 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 reuse/injection
        - Write other variables →more attacker-controlled variable (beyond the intended program flow)
- 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)
    - Can overflow on the stack → overwrite function pointer on the heap, overwrite variable on the data, …(?)
    - So, the answer is that it may be possible, the necessary condition is the ability to control value of a (data) pointer

Memory corruption

- Example for data and for heap

```
. . .
int val;
. . .
int someFunc(. . .) {
      . . .
      int *ptr_data = &val;
      . . .
      *ptr_data = a;
      . . .
}
. . .
```

- If overflow on stack allows controlling `ptr_data`, then write a at any address of choice
- If attacker also controls a, then attacker also controls written value

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

- If overflow on stack allows controlling `ptr_heap`, then write a in any address of choice
- If attacker also controls a, then attacker also controls the written value

---

Defending against memory corruption vulns

- Strategies:
  - Vuln prevention: use safer programming languages, learn to write memory safe code and use tools for analyzing and patching insecure code
  - Exploit prevention: add mitigations that make it harder to exploit common vulnerabilities
  - Prevention attempts

Use safer programming languages

- Memory safe languages are designed to check bounds and prevent undefined memory access, examples of memory safe languages are Java, Python, C#, Go, Rust (most languages, besides C, C++, objective C)
- These languages are not vulnerable to memory safety vulnerabilities, the only way to stop 100% of vulnerabilities
- Reasons why they are not used
  - Most commonly cited reason: performance, but this is no longer an issue, the performance penalty of memory safety is insignificant
    - Only possible exceptions are O.S., certain embedded systems, certain gaming platforms
  - Real reason: legacy
    - Huge existing code bases are written in C, building on existing code is easier than starting from scratch

NB programmer time is costly and scarce → writing code in 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 and lot of python apps uses NumPy, that internally uses C)

Learn to write + tools

- Learn to write memory safe code: only use libraries that are deemed safe (with functions that check bounds)
  - Programmer discipline + automatic tools
  - There is a set of defensive rules like always check that a pointer is not null before derefencing it, alway contain and check data from untrusted sources, …
  - It is clearly difficult programming following all the rules
  - Certain defensive rules are crucial, even with memory safe languages
- Use tools for analyzing
  - Bug-finding / code-smelling tools: Look for common bad practices, very effective, but there is the 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

- Basic idea: make it harder to exploit common vulnerabilities
    - Compiler + O.S.
    - Make the program crash with common exploits as input → crashing is safer than exploitation
    - Low / insignificant runtime overhead
- Key techniques: (non molto chiaro, ma dopo si vedono nello specifico alcune di queste tecniche)
    - non executable pages: W^X (write or execute), DEP (Data Execution Prevention)
    - Address space layout randomization (ASLR)
    - Stack canary
    - Pointer authentication
    - More mitigations exist (non ci interessa, in caso ci sono dei link)
- Remark:
    - make it harder to exploit, not impossible
    - there are many techniques for trying to circumvent mitigations
    - Writing exploits for memory corruption vulns on modern platforms is very difficult: much effort for circumventing defenses, usually they require chaining multiple vulns

---

O.S. based mitigations

1. Non executable pages
2. Address space layout randomization (ASLR)

- O.S support: hw support for 1 is ubiquitous
- Do not require recompilation → very important: all libraries and executables unchanged, just switch an option when executing
- **Mitigation: Non executable pages:**
    - Fact: all programs do not need memory that is both written to and executed
    - A very powerful defense is: each memory page is either execrable or writable, mandatory access control (hw + os) → command names of this mitigations are W^X (write or execute) and DEP (data execution prevention)
        - Code: executable → shellcode cannot be written here
        - Stack, data, heap: writable → shellcode written here cannot be executed
- **Circumvention idea, code reuse:**

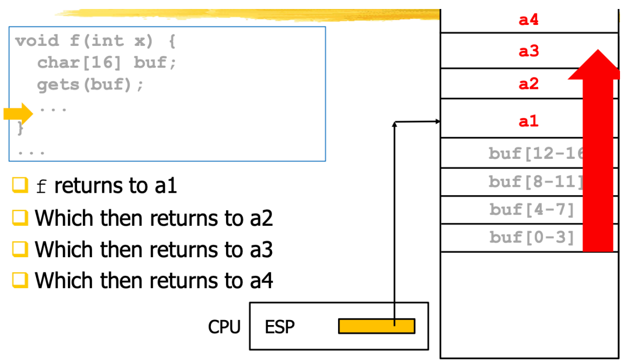
1. return to libc
    1. Identify potentially useful functions that already exist in memory (es `int system(const char *command;`)
    2. Overwrite stack so that return address is the library function, input arguments are injected
2. Return oriented programming (ROP)
    1. Identify potentially useful segments of code that already exist in memory and terminate with ret (gadgets) (es library functions not from their beginning)
    2. …

    - Example:
    - Code segments that already exist in memory (es in C library)

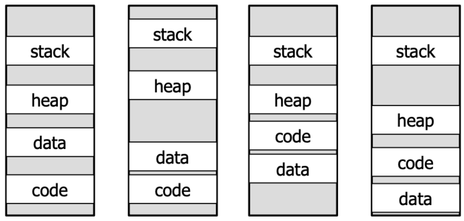
1. Address **a1**: Short code segment that terminates with `ret`
2. Address **a2**: Short code segment that terminates with `ret`
3. Address **a3**: Short code segment that terminates with `ret`
4. Address **a4**: Short code segment that terminates with `ret`

-Their concatenation achieves something useful to the attacker →

-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 (non contiguous)



Memory corruption



- **Mitigation: ASLR**
    - Place each memory segment in a different location each time the program runs → the attacker cannot prepare exploits with correct addresses
- **Circumventing idea:**
    - shellcode obtains address of a variable whose relative address to shellcode is know (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 → can be brute forced
        - 64 bit architectures: 2^36 values → forse anche no

Compiler-based mitigations:

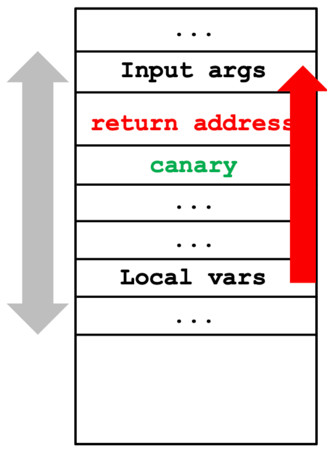
- Compiler based mitigations
1. Snack canary
2. Pointer authentication
- Compiler support: hw support for 2 necessary
- Recompilation required: costly → all the libraries and executables have to be recompiled and redistributed

---

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 effettive on vulns that write to memory in other ways (and possibly at attacker-chosen positions on the stack)
- How it works:
    - Code generated by the compiler
    - When the 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
    - Overflow on local variable
    - overwriting return address requires overwriting the canary, but the exploit cannot know its value

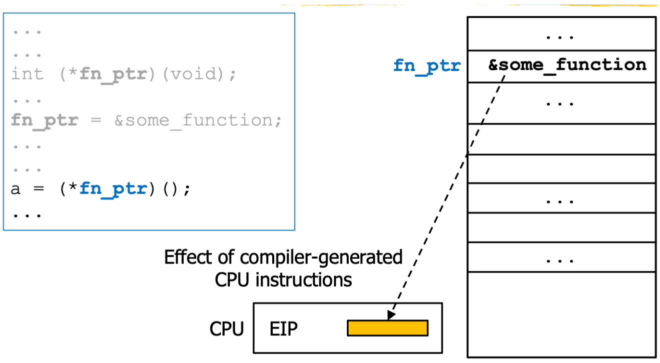
```
callee:
push ...      ; push canary@stack
push ...      ; local variables
...           ; function execution
pop ...       ; free local variables
...    ; if canary@stack <> canary
...    ; then jmp crash
ret
```

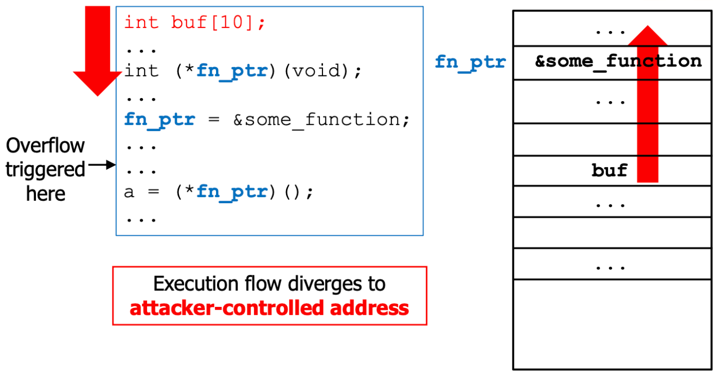


- **Circumvention idea**
    - Guess the canary value, repeat the injection for every possible canary value, feasibility depends on the range size. First byte of the canary is always '\0' (to mitigate possible string-based attacks)
    - Leak (and then use) the canary value: exploit vulnerabilities that allows reading the full stack

Pointer authentication

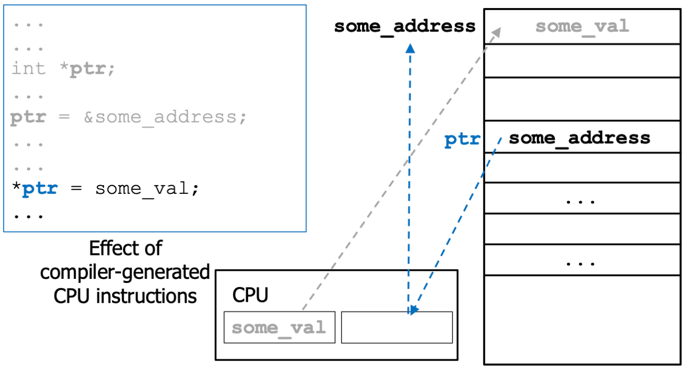
- Function pointers: Overwriting:

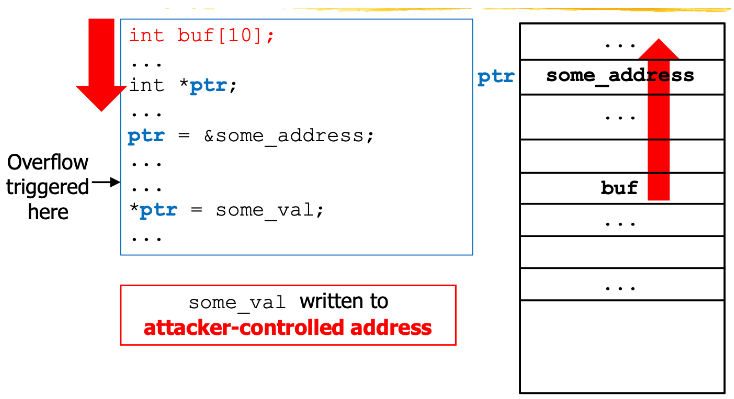




- Data pointers:

Overwriting:





- Fact: overwriting pointer values in memory is a crucial step of (almost) every exploit for memory safety vulnerabilities
- Modern CPU architectures
  - Process memory N1 = 64 bits
  - Physical memory N2 < N1
  - Mapping from virtual memory (of each process) to physical memory is done by hw + os

    → every address in program has more bits than necessary
- **Pointer Authentication Code (PAC)**
  - Every modern CPU has a secret key K in a protected hw register and hw instructions for computing HMAC(K,V) with N1-N2 bits output length (unused address bits)
  - The compiler must generate these instructions whenever it is necessary to write a pointer value to memory or read a pointer value from memory
  - Writing pointers to memory: compiler generated instructions for writing A to memory:
    - Compute HMAC(K,A), then write <HMAC(K,A), A>
  - Reading pointers for memory: compiler generated instructions for using (X,A) read from memory
    - It X = HMAC (K,A), then proceed, else exception
  - Overwriting A1 with AX requires PAC(AX) witch cannot be computed (immagino il motivo sia che di base l'attaccante non ha la chiave per calcolare l'HMAC)
- **Circumvention ideas:**
  - PAC() generation is deterministic:
    - force CPU to generate a PAC() for addresses of choice and reuse them

Memory corruption

    - 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 OS to expose the key

Mitigation in practice

- 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:
  - Available on most modern platforms (chissà a cosa si riferisce)
  - Compiler flags / OS flags
  - Stack corruption is essentially dead (microsoft 2019) → other memory unsafely (and language based) issues are not
  - Pay attention to the default configurations
  - Cisco adaptive security appliance (ASA) 2016
    - Authenticated remote code execution
    - Two buffer overflows
    - No non executable pages
    - No ASLR
    - No stack canary
- Big headache: IoT / embedded systems often do not have key mitigations and run memory unsafe code
  - Writing exploits for those platforms tends to be easy