Memory corruption problems: summary

- access outside array bounds
- buggy pointer arithmetic
- dereferencing null pointer
- using a dangling pointer
- forgetting to check for failures in allocation
- forgetting to de-allocate, causing a memory leak

Summary

```
1000  …
1001  void f (){
1002    char* buf,*buf1;
1003    buf = malloc(100);
1004    buf[0] = 'a';
        ...
2001    free(buf1);
2002    buf[0] = 'b';
        ...
3001    free(buf);
3002    buf[0] = 'c';
3003    buf1 = malloc(100);
3004    buf[0] = 'd'
3005  }
```

possible null dereference
(if malloc failed)

potential use-after-free
if buf & buf1 are aliased

use-after-free; buf[0] points
to de-allocated memory

memory leak; pointer `buf1`
to this memory is lost &
memory is never freed

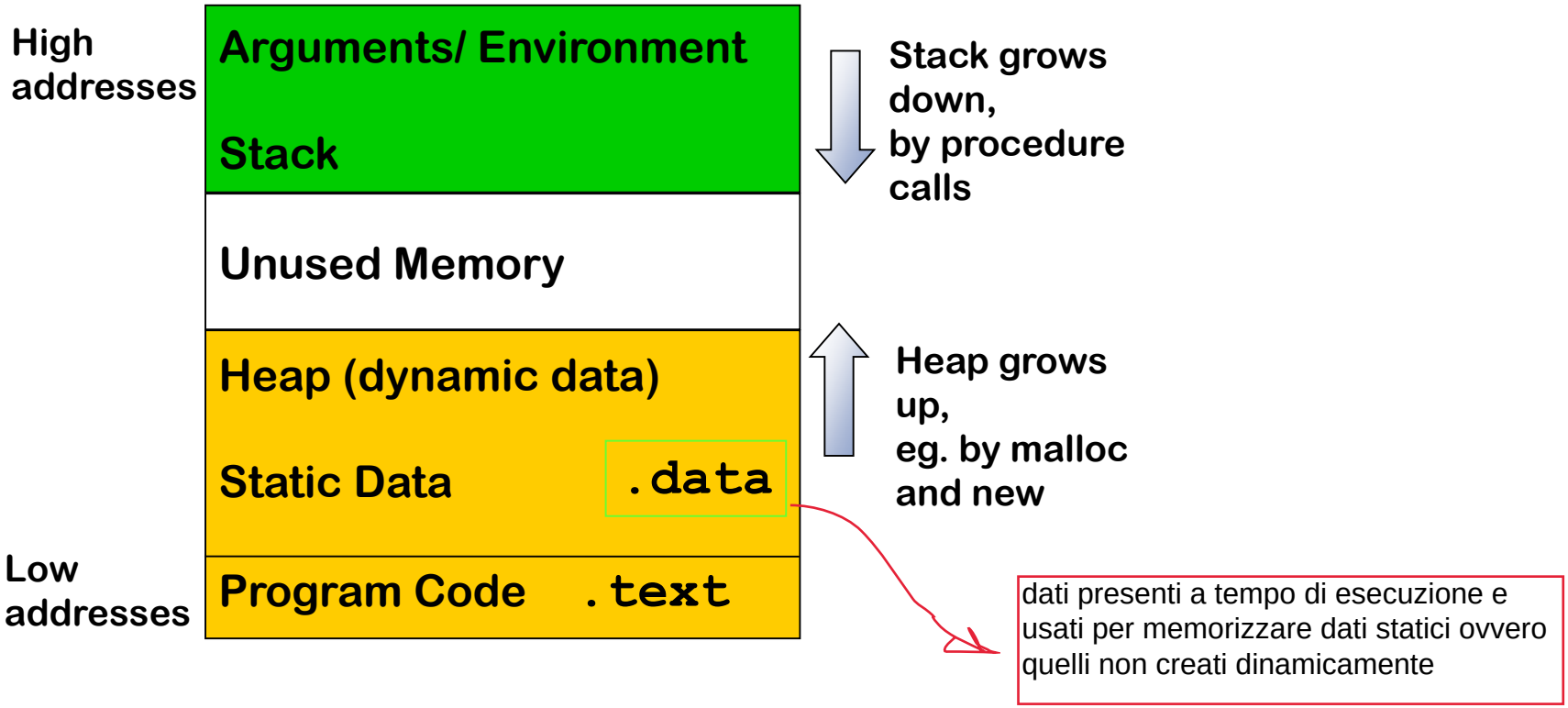
use-after-free, but now buf[0]
might point to memory that
has now been re-allocated

Reading

- **Chapter 3: Lecture Notes on Language Based Security, by E. Poll**
- **Available on TEAMS**

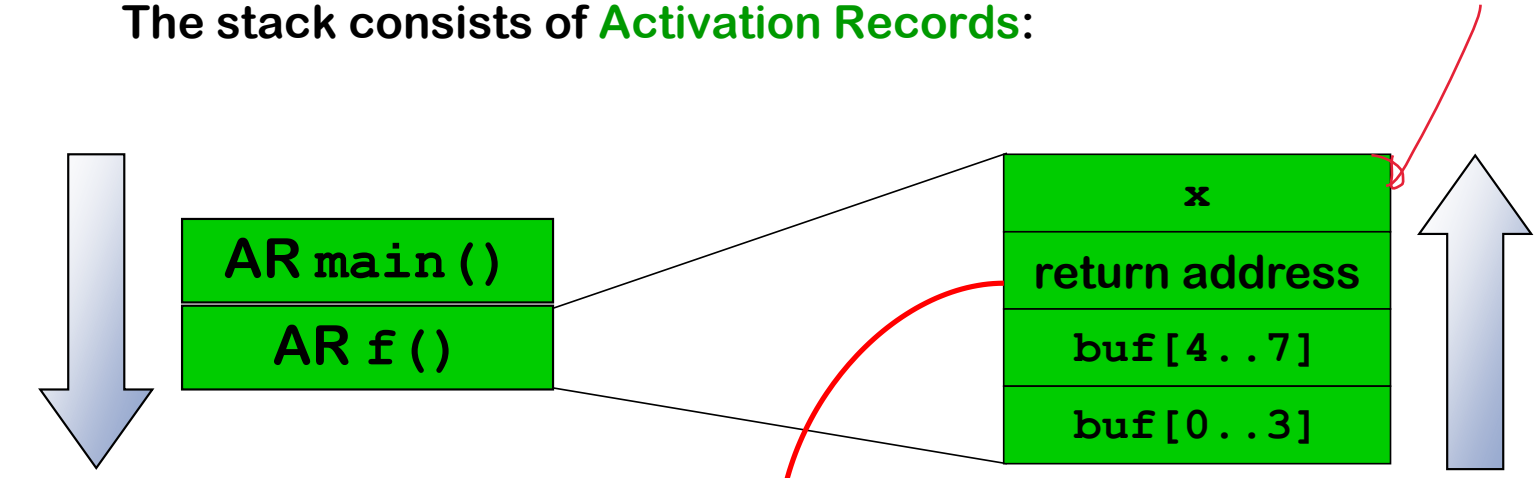
Run time organization
smashing the stack

Programming Language Implementation: memory layout



stiamo immaginando di avere 32 bit per memorizzare una parola etc..

The stack consists of Activation Records:



Stack grows downwards

```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); …
}
void format_hard_disk(){…}
```

Buffer grows upwards

C(C++)


SOME ISSUES

Note

The compiler must determine, at **compile- time**, the layout of activation records and the generated code that, when executed at run- time, correctly **accesses locations** in those activation records.

The compiler must determine, at **compile- time**, the layout of data heap and generate code that, when executed at run- time, correctly **accesses locations** in those data.

Data layout, and date accesses are designed together with the compiler!!!

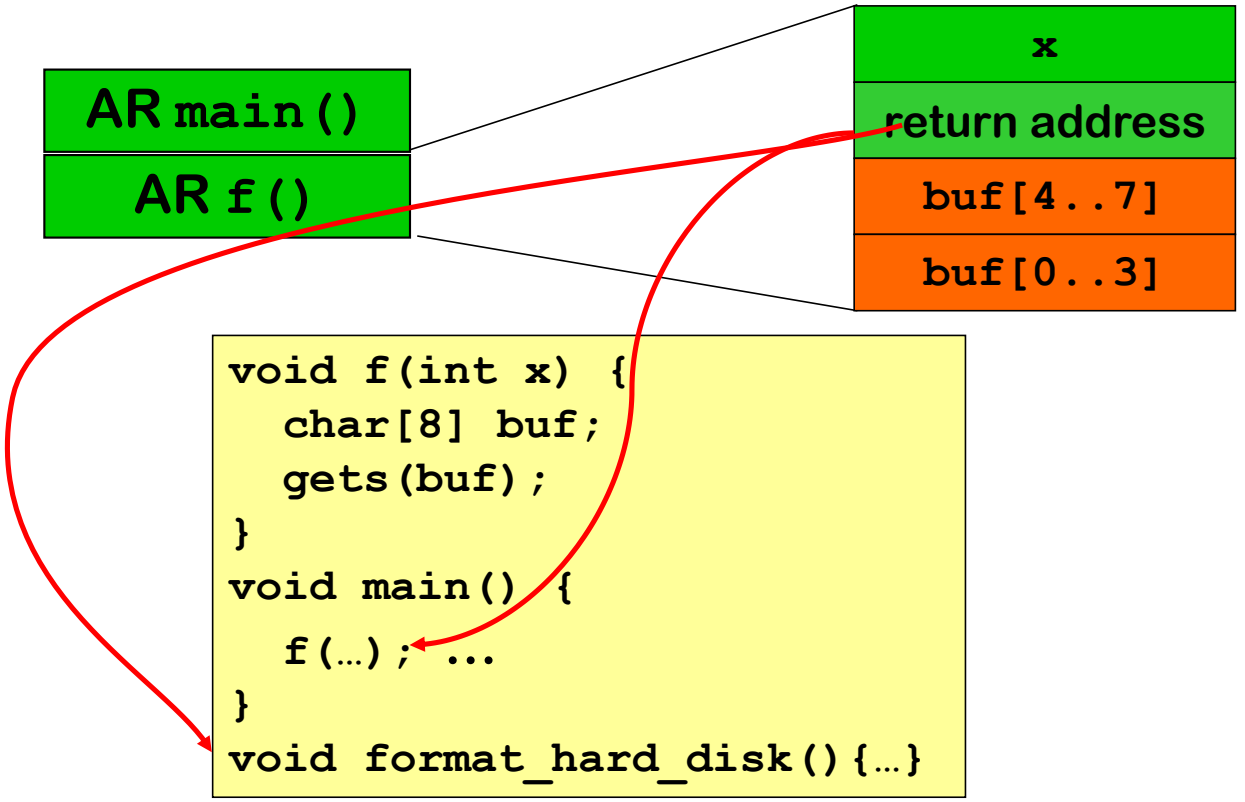
si intende la dimensione del blocco, la posizione degli elementi all'interno del blocco e i meccanismi di accesso all'interno del blocco di offset...questi aspetti sono determinati assieme al progetto del compilatore

C Library functions

- **C** library function - **gets**()
- The **C** library function **gets**(char *str) reads a line from stdin and stores it into the string pointed to by str. It stops when either the newline character is read or when the end-of-file is reached, **whichever comes first**

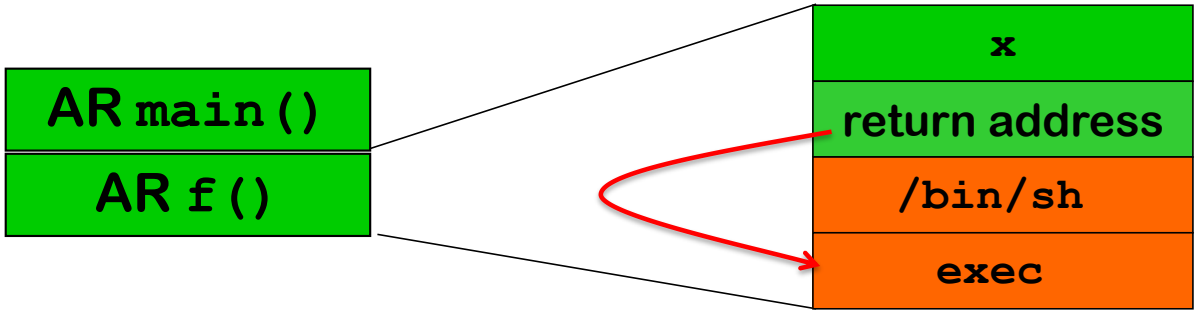
What if `gets()` reads more than 8 bytes ?

Attacker can jump to abitrary point in the code!

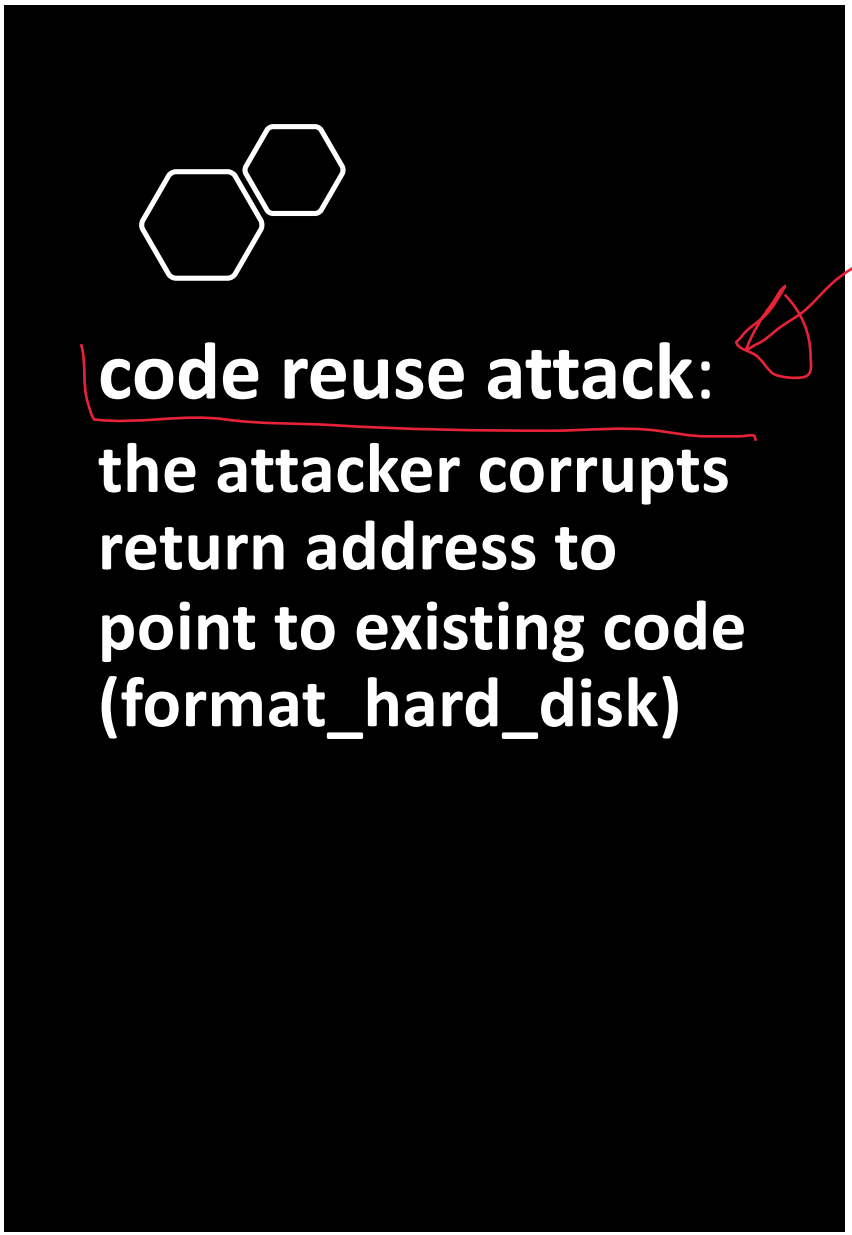


What if `gets()` reads more than 8 bytes ?

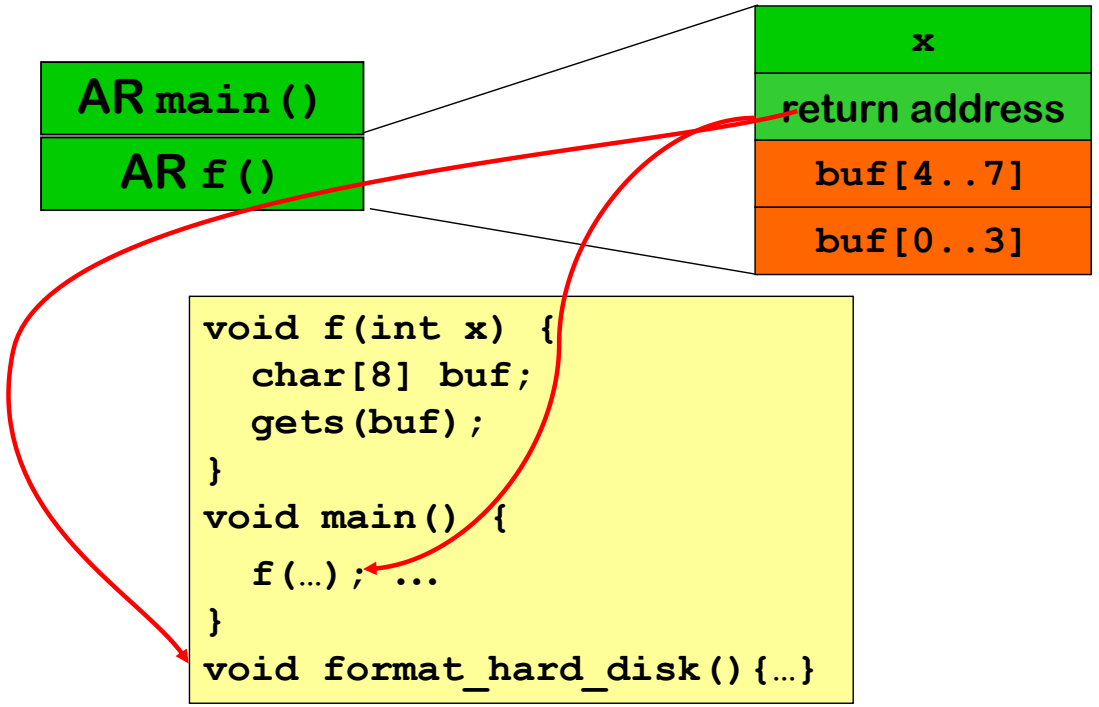
Attacker can jump to his own code (aka shell code)

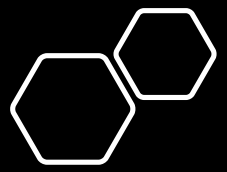



```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); …
}
void format_hard_disk(){…}
```



code reuse attack:
the attacker corrupts return address to point to existing code (format_hard_disk)

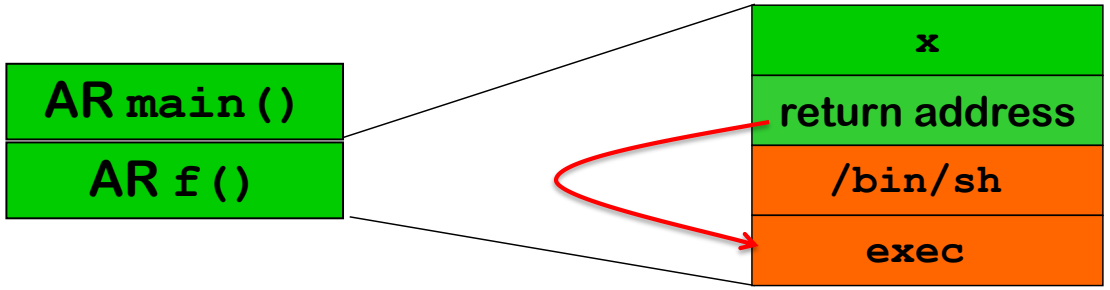




code injection attack

The attacker inserts his own shell code in a buffer and corrupts

return addresss to point to his code (**exec('/bin/sh')**)




```
void f(int x) {
   char[8] buf;
   gets(buf);
}
void main() {
   f(…); ...
}
void format_hard_disk(){…}
```

GETS HAS BEEN REMOVED FROM C STANDARD SINCE 2011

GOOD NEW


What about
run-time support?

… Lots of details (of the run-time organization) to get right!

knowing precise location of return address
other data on stack,
knowing address of code to jump to,
….

Discussion

Activation Record Layout: the advantage of defining the order of elements in AR is that accesses are obtained by a standard and efficient mechanism: **Base address + offset**

There is nothing magic about this organization: it improves execution speed and simplifies code generation

But the are are a number of vulnerabilities

DEFENCES

Changing software toolchain (e.g., compilers) and **run-time mechanisms** to prevent exploitation of vulnerabilities.

Changing software toolchain: extra work on designer & developers,

Chanting run-time support: increses run-time overhead

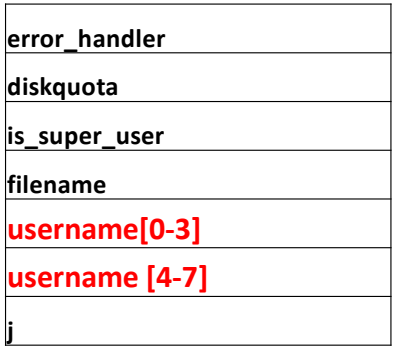
More fun on the stack = more attacks

```
void f(void(*error_handler)(int),...) {
  int diskquota = 200;
  bool is_super_user = false;
  char* filename = "/tmp/scratchpad";
  char[8] username;
 int j = 12; ...
}
```

vediamo come è fatto sulla destra il record di attivazione quando viene messe sullo stack. Per semplicità viene messo sul record di attivazione solamente le informazioni relativamente alle variabili locali che sono presenti nel record di attivazione.

More fun on the stack = more attacks

```
void f(void(*error_handler)(int),...) {
  int diskquota = 200;
  bool is_super_user = false;
  char* filename = "/tmp/scratchpad";
  char[8] username;
 int j = 12; ...
}
```



The attacker has enough knowledge on run-time data layout and accesses to run-time data: the threat model

The attacker can overflow **username**

More fun on the stack = more attacks

```
void f(void(*error_handler)(int),...) {
  int diskquota = 200;
  bool is_super_user = false;
  char* filename = "/tmp/scratchpad";
  char[8] username;
 int j = 12; ...
}
```

error_handler
diskquota
is_super_user
filename
username[0-3]
username [4-7]
j

The attacker has enough knowledge on run-time data layout and accesses to run-time data: the threat model
The attacker can overflow **username**

Corrupting the return address, might lead to corrupt

- pointers (e.g. filename)

- other data on the stack (e.g. is_super_user, diskquota)

- function pointers (e.g. error_handler)

But not j, unless the compiler chooses to allocate variables in a different order, which the compiler is free to do.

questa operazione di corruzione dipende da come il compilatore progetta il layout in fase di esecuzione delle funzioni.

Also more fun on the heap

```
struct BankAccount {
  int number;
  char username[20];
  int balance;
}
```

Suppose attacker can overflow username

More fun on the heap

```
struct BankAccount {
  int number;
  char username[20];
  int balance;
}
```

Suppose attacker can overflow username

This can corrupt other fields in the struct.

Which field(s) can be corrupted depends on the order of the fields in memory


… it is just an issue of
run-time data structures?

Data representation (C Reminder)

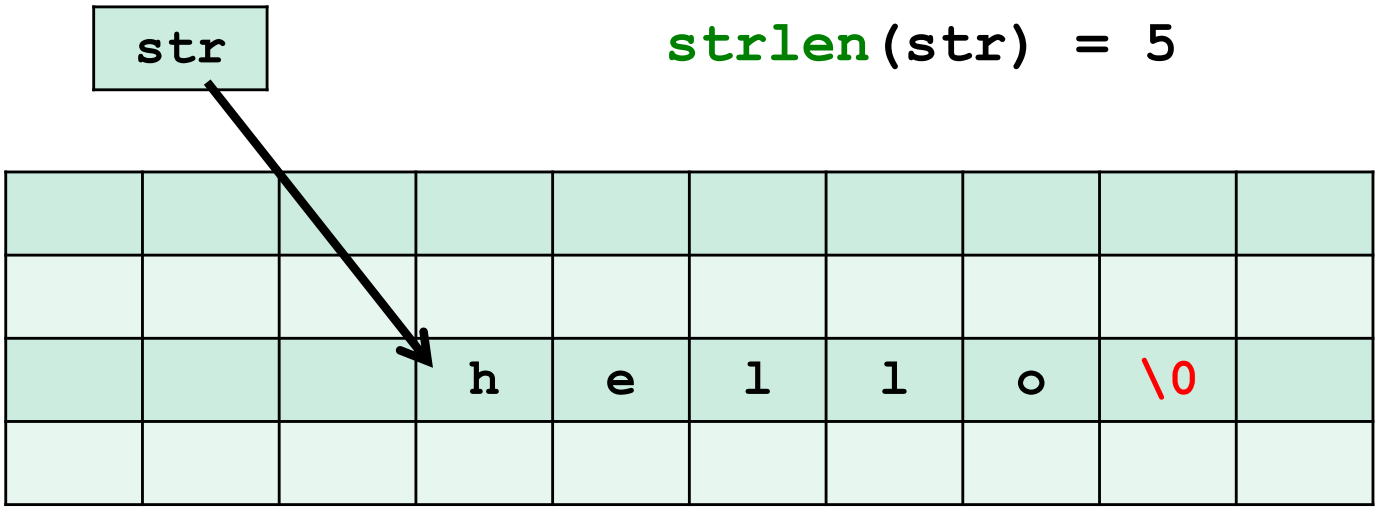
A char in C is always exactly one byte

A string is a sequence of chars terminated by a NULL byte

String variables are pointers of type char*

```
char* str = "hello";   // a string str
```

```
strlen(str) = 5
```



```
char buf[20];
gets(buf); // read user input until
           // first EoL or EoF character
```

- *Never* use `gets`
  - `gets` has been removed from the C library
- Use `fgets(buf, size, file)` instead

```
char dest[20];
strcpy(dest, src); // copies string src to dest
```

- `strcpy` assumes `dest` is long enough and `src` is null-terminated
- Use `strncpy(dest, src, size)` instead

```
char dest[20];
strcpy(dest, src); // copies string src to dest
                   // strcpy assumes dest is long enough
                   // and assumes src is null-terminated
```

Suggestion: strncpy(dest, src, size) is better (padding null char)

```
char dest[20];

// Beware of difference between sizeof and strlen

sizeof(dest) = 20 // size of an array

strlen(dest) = number of chars up to first null byte
// length of a string
```

```
char src[9];
char dest[9];
char* base_url = "www.di.it";
strcpy(src, base_url, 9);
// copies base_url to src
strcpy(dest, src);
// copies src to dest
```

base_url is 10 chars long,
incl. its null terminator,
so src will not be null-terminated

```
char src[9];
char dest[9];
char* base_url = "www.di.it";
strncpy(src, base_url, 9);
// copies base_url to src
strcpy(dest, src);
// copies src to dest
```

quando vado a copiare con strncpy "base_url" in source comincio a violare una delle proprietà di strncpy ovvero src non sarà null-terminated.

Don’t replace

```
strcpy(dest, src)
```

with

```
strncpy(dest, src, sizeof(dest))
```

but with

```
strncpy(dest, src, sizeof(dest)-1)
dst[sizeof(dest)-1] = '\0';
```

if `dest` should be null-terminated!

Moral

MA PERCHE' ALLORA NON PROGETTARE DEI LINGUAGGI DI PROGRAMMAZIONE IN MODO TALE CHE TUTTI GLI ASPETTI DI GESTIONE DELLE PROPRIETA' DEI DATI PRIMITIVI SIANO DEFINITI DA UN COMPILATORE E CHE SIA IL LINGUAGGIO AD AMMETTERE DELLE GARANZIE CHE SE C'E' LA PROPRIETA' INVARIANTE CHE NEL LINGUAGGIO LA STRINGA DEVE ESSERE NULL-TERMINATED ALLORA TUTTE LE STRINGHE SARANNO NULL-TERMINATED?

a strongly typed programming language would guarantee that strings are always null-terminated, without the programmer having to worry about this…

```
char *buf;
int len;
...
buf = malloc(MAX(len,1024)); // allocate buffer
read(fd,buf,len); // read len bytes into buf
```

```
char *buf;
int len;
...
buf = malloc(MAX(len,1024)); // allocate buffer
read(fd,buf,len); // read len bytes into buf
```

What happens if `len` is negative?

The length parameter of `read` system call is unsigned!
So negative `len` is interpreted as a big positive one!

```
char *buf;
int len; ...
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(MAX(len,1024));
read(fd,buf,len);
```

A remaining problem may be that `buf` is not null-terminated; we ignore this for now.

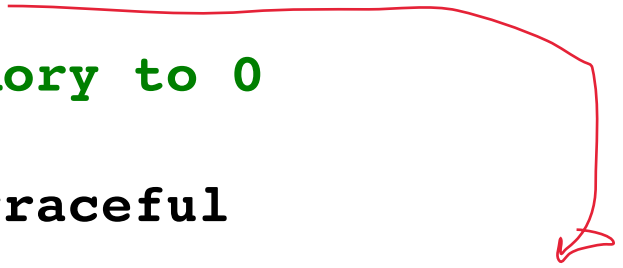
```
char *buf;
int len; ...
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(MAX(len,1024));
read(fd,buf,len);
```

What if the malloc() fails?
(because we are out of memory)

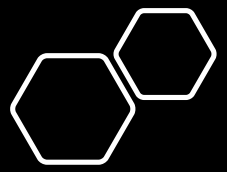
```
char *buf;
int len;
...
if (len < 0)
    {error ("negative length"); return; }
buf = malloc(MAX(len,1024));
if (buf==NULL) { exit(-1);}
// or something a bit more graceful
read(fd,buf,len);
```

```
char *buf;
int len;
...
// better solution

if (len < 0) {error ("negative length"); return; }
buf = calloc(MAX(len,1024));
//to initialise allocate memory to 0
if (buf==NULL) { exit(-1);}
// or something a bit more graceful
read(fd,buf,len);
```



la calloc mette tutto in un valore di default



Defensive Programming

We look out for trouble before it happens, anticipate the unexpected, and never put ourselves into a position from which we can’t extricate ourselves

Design by Contract: clients and suppliers must agree on rights and responsibilities.

The Pragmatic Programmer

From Journeyman to Master

Andrew Hunt
David Thomas

```
#define MAX_BUF 256

void BadCode (char* in) {
 short len;
 char buf[MAX_BUF];
 len = strlen(in);
 if (len < MAX_BUF) strcpy(buf,in);
}
```

```
#define MAX_BUF 256

void BadCode (char* in)
{    short len;
     char buf[MAX_BUF];

     len = strlen(in);

     if (len < MAX_BUF) strcpy(buf,in);
}
```

What if `in` is longer than 32K ?

len may be a negative number, due to integer overflow

hence: potential buffer overflow

The integer overflow is the root problem,
the (heap) buffer overflow it causes makes it exploitable

See https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer+overflow


Safe Programming
Languages

safe programming languages

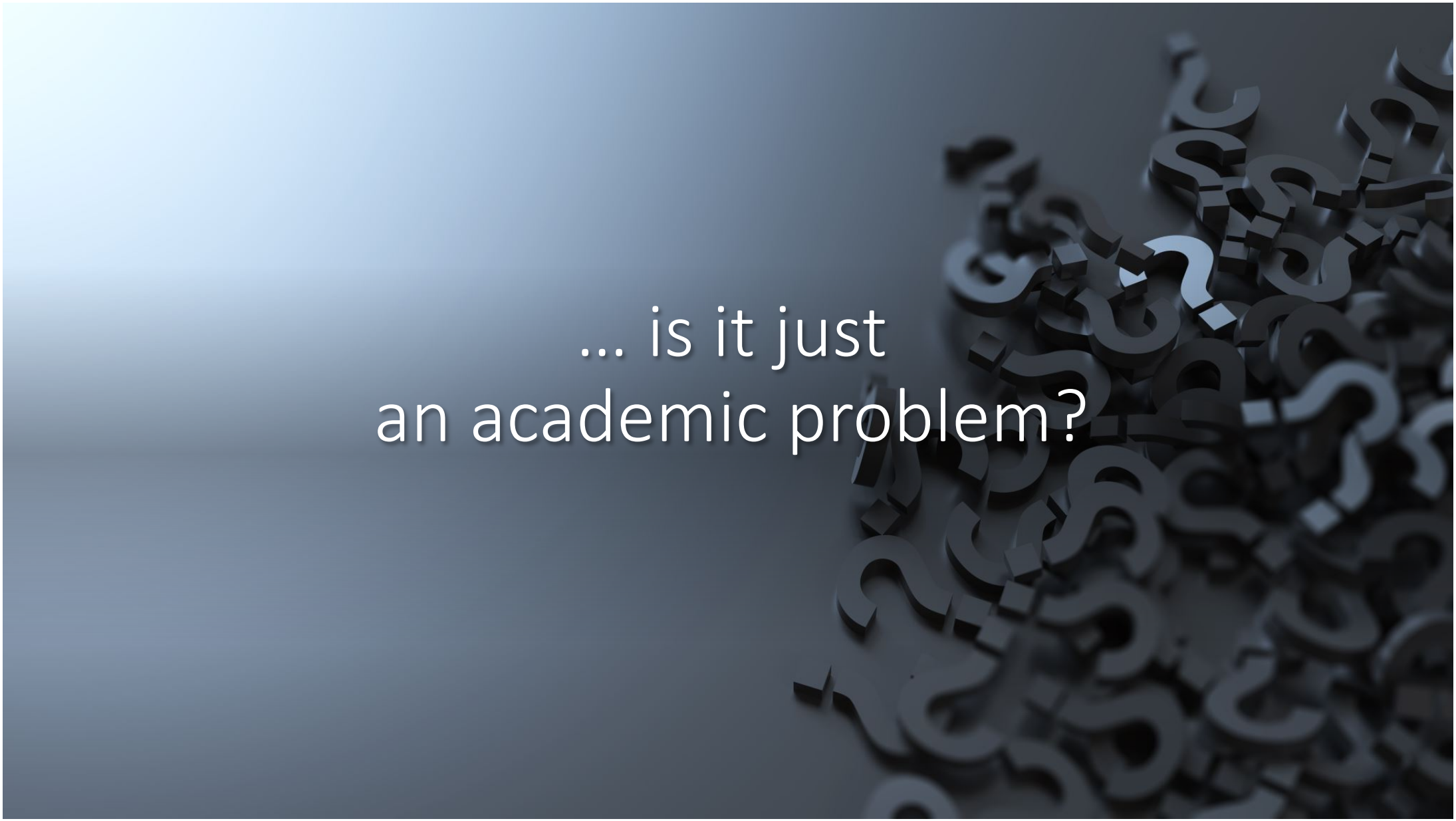
- In a safe programming language, the programmer would not have to worry about writing past array bounds
  - The design of the language includes suitable constructs to programmdo design and program the raise of exception.like IndexOutOfBoundsException instead
- Suitable type systems and implicit conversion mechanisms from signed to unsigned integers
  - the type system/compiler would forbid this
- Malloc possibly returning null & malloc not initialising memory
  - The language always ensures default initialisation
- Integer overflow
- …

Unsafe programming languages

- Applications written in programming languages like C or C++ are prone to ***Memory Corruption*** bugs .
- The lack of memory safety (or type safety) in such languages enables attackers to exploit memory bugs by maliciously altering the program's behavior or even taking full control over the control-flow.

```
1. void* f(int start) {
2. if (start+100 < start) return SOME_ERROR;
// line 2 checks for overflow
3. for (int i=start; i < start+100; i++) {
4. . . . // i will not overflow
5. }
}
```

1) But one cannot assume that overflow produces a negative number;
   I. line 2 is not a good check for integer overflow.
2) Worse still, if integer overflow occurs, behaviour is undefined, and ANY compilation is ok
   i. So compiled code can do anything if `start+100` overflows
   ii. So compiled code can do nothing if `start+100` overflows This means the compiler may remove line 2


… is it just
an academic problem?

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)  return; /* len too large */
if (buf + len < buf) return;
/* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

questo è un codice noto in letteratura preso da un articolo accademico, qui il problema è che len è unsigned...qui l'istruzione buf+len < buf per il compilatore è un istruzione che sarà sempre verificata per il compilatore dunque la elimina, questa cosa è detta CONSTANT PROPAGATION, ovvero ovunque vi è quella espressione viene messo false e si fanno le ottimizzazioni relative. Perchè in questo caso da un punto di vista logico, il valore A a cui sommo un valore B NON PUO' ESSERE MINORE DI A, dunque elimina il controllo e si può avere un overflow a tempo di esecuzione.

similar checks are found in a number of systems, including the the Linux kernel and the Python interpreter!!!

C standard: in a correct program, pointer addition will not yield a pointer value outside of the same object.

This allows gcc to simply assume that no pointer overflow ever occurs on *any* architecture.

Under this assumption, buf + len must be larger than buf, and thus the "overflow" check always evaluates to *false*.

gcc removes the check, paving the way for an attack to the system

A better code

affrontare il problema in questo modo sarebbe la cosa corretta ovviamente e non può eliminare l'istruzione perchè per poterla eliminare deve avere il valore attuale di len e il valore attuale di BUFLEN, ma questa operazione staticamente non può essere sostituita perchè dipende dal valore effettivo associato alle variabili.

```
if (len >= BUFLEN) return
/* buffer overflow attempt started */
```

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun) return POLLERR;
/* write to address based on tun */
```

A defect in the Linux kernel, where the programmer incorrectly placed the dereference tun->sk before checking tun != null

facendo questo controllo dopo lo standard del C dice: se il puntatore viene dereferenziato vuol dire che deve essere non-null e dato che tun è non-null allora questo controllo può essere tolto e il compilatore può sapere, avendo fatto un operazione di deferenziazione ed ha avuto successo, a quel punto il controllo successivo sul fatto che debba essere diverso da null si può togliere e se si può togliere ho un programma che in realtà può operare con strutture in particocolare accedere a un socket che in realtà non esiste

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun) return POLLERR;
/* write to address based on tun */
```

when gcc first sees the dereference tun->sk, it concludes that the pointer tun must be non-null (C standard states that dereferencing a null pointer is undefined)

Since tun is non-null, gcc further determines that the null pointer check is unnecessary and eliminates the check, creating a vunerability

A defect in the Linux kernel, where the programmer incorrectly placed the dereference tun->sk before checking tun != null

Discussion

Compiler optimization may create security vulnerabilities

… but ptimizations are important for achieving good performance; many optimizations fundamentally rely on the precise semantics of the C language,

Challenge: distinguish legal yet complex optimizations from an optimization that goes too far and create security holes!!

```
#include <stdio.h>

int main(int argc, char* argv[]) {
 if(argc>1)
    printf(argv[1]); return 0;
}
```

This program is vulnerable to format string attacks, where calling the program with strings containing special characters can result in a buffer overflow attack.

Format strings and memory corruption

Strings can contain special characters, eg `%s` in

```
printf("Cannot find file %s", filename);
```

Such strings are called format strings

What happens if we execute the code below?

```
printf("Cannot find file %s");
```

What can happen if we execute

```
printf(string)
```

where `string` is user-supplied ? And it contains special characters, eg %s, %x, %n, %hn?

Attacks

Suppose attacker can feed malicious input string `s` to the `printf(s)` statement.

Take1: The format string: `%x` reads and prints bytes from stack,

Assume s =
`%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x`
`%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x`
`%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x...`

`The execution of print(s)`
dumps the stack ,including passwords, keys,... stored on the stack

Attacks (2): Corrupting the stack

The format string

`%n` writes the number of characters printed to the stack,

Assume to input the string `12345678%n`

The execution of the statement writes value 8 to the stack

Attack (3): read arbitrary memory

a format string of the form
`\xEF\xCD\xCD\xAB %x%x...%x%s`

print string at memory address `ABCDCDEF`

Preventing attacks

- Always replace **printf(str)** with **printf("%s", str)**
- **Compiler** or **static analysis tool** could warn if the number of arguments does not match the format string,
  - printf ("x is %i and y is %i", x);
  - gcc has (far too many?) command line options for this: -Wformat –Wformat-no-literal –Wformat-security ...
- If the format string is not a compile-time constant, we cannot decide this at compile time, so compiler has to give false positives or false negatives
- **See https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string to see how common format strings still are**

Recap

- Buffer overflow is a major weakness in C and C++ programs because these language are not memory-safe
  - Tricky to spot
  - Typical cause: programming with arrays, pointers, and strings

Related attacks

- Format string attack: another way of corrupting stack
- Integer overflows: often a stepping stone to getting a buffer to overflows
  - but just the integer overflow can already have a security impact; eg think of banking software

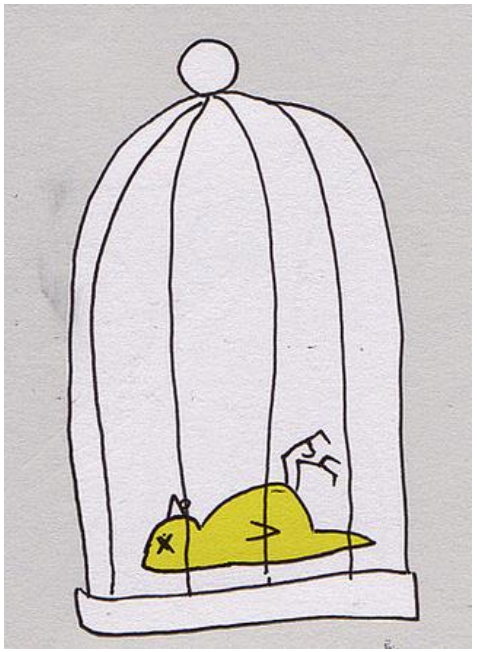

Platform
Level
Defenses

Platform level defenses

- Defenses the compiler, hardware, OS,... can take, **without the programmer having to know**
- Some defenses may need OS & hardware support
- Some defenses cause **overhead**
  - if the overhead is unacceptable in production code, we can still use it when testing
  - Some defenses may **break binary compatibility**
    - if a compiler adds extra book-keeping & checks, then all libraries may need to be re-compiled with that compiler

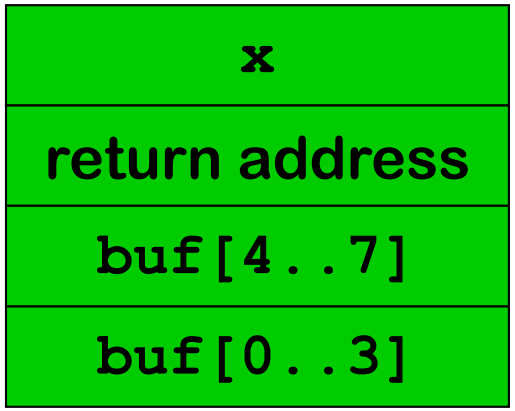
Defenses (#1): Stack Canaries

- A dummy value - **stack canary or cookie** - is written on the stack in front of the return address and checked when function returns
- A careless stack overflow will overwrite the canary, which can then be detected
  - **first introduced in as StackGuard in gcc**
  - **only very small runtime overhead**

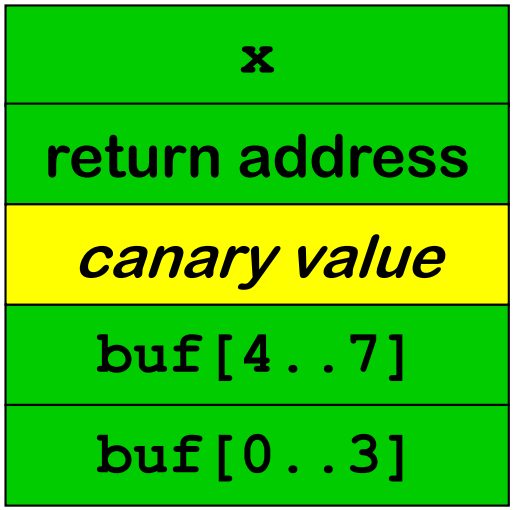


l'effetto netto non è solo quello di modificare il layout del record di attivazione ma anche di una modifica dell'epilogo ovvero del codice che viene associato all'epilogo della funzione non è quello che avevamo prima.

Stack without canary



Stack with canary



Il compilatore deve modificare la struttura del runtime, i layout del record di attivazione
Continua a usare il meccanismo di indirizzo di base + offset per poter accedere agli elementi dunque efficiente
Il prezzo da pagare è che il codice dell'epilogo e anche il codice del prologo sono diversi in quanto il codice del prologo quando vado ad inserire sullo stack del record di attivazione deve mettere il valore del canary, il codice dell'epilogo invece prima di restituire il controllo al chiamante deve fare il controllo del canary e controllare che non sia stato modificato.

Improvements

Il valore del canary può essere scelto in diversi modi:

More variation in canary values: not a fixed values hardcoded in binary but a random values chosen for each execution

XOR the return address into the canary value

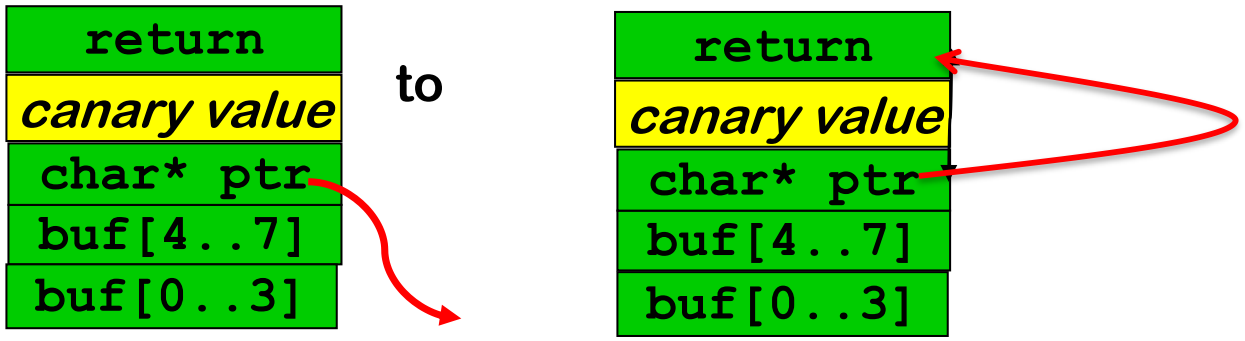
Include a null byte in the canary value, because C string functions cannot write nulls inside strings

But

A careful attacker can still defeat canaries, by

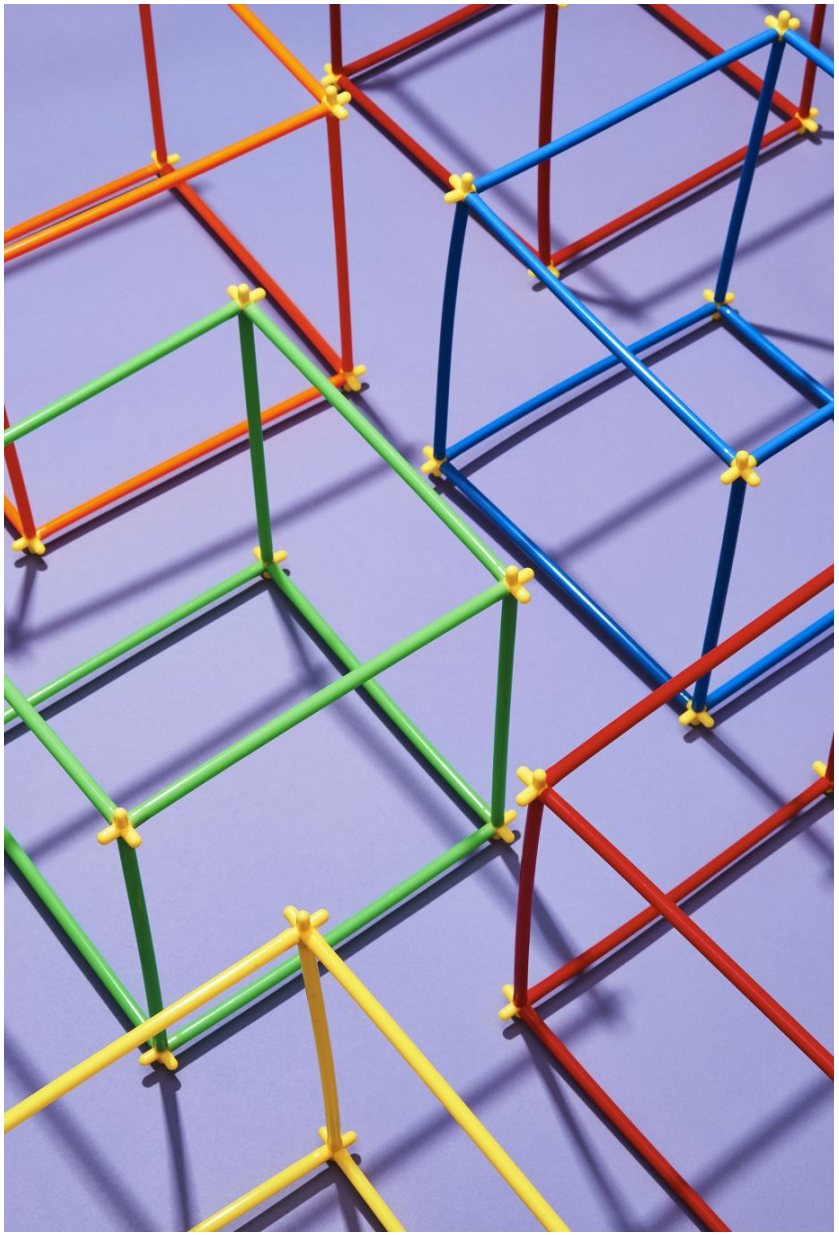
- overwriting the canary with the correct value
- corrupting a pointer to point to the return address to then change the return address without killing the canary

eg changing



Countermeasures

RUN-TIME STRUCTURES



IBM ProPolice

- Re-order elements on the stack to reduce the potential impact of overruns
  - swapping parameters
- Safer strategy: allocated array buffers **above** all other local variables

Further improvements (cont.)

- A separate shadow stack with copies of return addresses, used to check for corrupted return addresses
- The attacker should not be able to corrupt the shadow stack

lo stack nascosto contiene esattamente una copia magari cifrata dei punti di ritorno. A questo il prologo mette sullo stack il record di attivazione della funzione che viene invocata, mette nello stack il punto di ritorno e poi copia nello stack nascosto il valore del punto di ritorno, mentre l'epilogo prende sullo stack le informazioni presenti sul punto di ritorno e va a controllare se quelle informazioni coincidono con le informazioni memorizzare nello stack nascosto. Se non coincidono significa che c'è stato un attacco. Questo comporta una modifica del runtime, della generazione del codice, perchè lo stack è nascosto.

A questo punto si potrebbe pensare di randomizzare tutta la struttura del layout non solo dello stack ma anche dell'heap e anche su come vengono memorizzati i dati globali. Tutto questo aspetto è randomizzato e dunque la visione del layout di tutta la struttura a runtime che è valida per un esecuzione non è valida per quella successiva, per cui se l'attaccante opera su quella immagine non riesce a replicarla su quella successiva

Address Space Layour Randomization

- Attacker needs detailed info about memory layout
  - to jump to specific piece of code
  - to corrupt a pointer at known position on the stack
- Attacks become harder if compilers randomise the memory layout every time we start a program
  - change the offset of the heap, stack, etc, in memory by some random value
- Attackers can still analyse memory layout on their own laptop, but will have to determine the offsets used on the victim’s machine to carry out an attack

Run-time organization and code generation

Stack & Heap

- Stack Canaries
- Reorder layout of AR and Heap elements (randomization)
- Shadow stack

Code optimization: safer startegies

- More later!!

EXAMPLE

Windows 2003 Stack Protection

Nice example of the ways in which things can go wrong…

- Enabled with /GS command line option in Visual Studio
- When canary is corrupted, control is transferred to an exception handler
- Exception handler information is stored …

  on the stack!
- Attacker can corrupt the exception handler info on the stack, in the process corrupt the canaries, and then let Stack Protection mechanism transfer control to a malicious exception handler

  [http://www.securityfocus.com/bid/8522/info]
- Countermeasure: only allow transfer of control to registered exception handlers