CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenturesecurity

aizoon AUSTRALIA EUROPE USA
TECHNOLOGY CONSULTING

B5

EY Building a better working world

eni

exprivia | ITALTEL

IBM

KPMG

LEONARDO

NTT DATA
Trusted Global Innovator

NUMERA
SISTEMI E INFORMATICA S.p.A.

Telsy

SPONSOR GOLD

bip.

cisco

MONTE DEI PASCHI DI SIENA
BANCA DAL 1472

negg

NOVANEXT
connecting the future

pwc

SPONSOR SILVER

DiGi ONE
the leading digital company

ICT CYBER CONSULTING

Software Security 2
Memory Management and Allocation

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Outline

- Memory Management
- Memory Layout
- Stack
- Heap
- Bad stories

Memory management

- Many of the modern programming languages allow programmers to use data types without having to know how they are represented.
- Similarly, programmers often ignore where data is stored (allocated)…
  - compiler makes decisions, however at runtime allocation is under the control of Operating System and CPU.

Memory management

- In some programming languages, like C, memory management can be controlled by programmers:
  - memory can be dynamically allocated and deallocated;
  - memory address of variables can be obtained (pointers).
- If *x* is a variable, *&x* denotes the pointer to x, i.e., the memory address where x is stored.

Memory allocation…

Let us consider the following simple C program:

```c
#include <stdio.h>

int main() {
    int i;
    char c;
    short s;
    long l;

    printf("i is allocated at %p\n", &i);
    printf("c is allocated at %p\n", &c);
    printf("s is allocated at %p\n", &s);
    printf("l is allocated at %p\n", &l);
}
```

We can assume that:

- *int* needs 4 bytes
- *char* needs 1 byte
- *short* needs 2 bytes
- *long* needs 8 bytes.

Memory allocation…

Let us consider the following simple C program:

```c
#include <stdio.h>

int main() {
    int i;
    char c;
    short s;
    long l;

    printf("i is allocated at %p\n", &i);
    printf("c is allocated at %p\n", &c);
    printf("s is allocated at %p\n", &s);
    printf("l is allocated at %p\n", &l);
}
```

Variable declatations

We can assume that:

- *int* needs 4 bytes
- *char* needs 1 byte
- *short* needs 2 bytes
- *long* needs 8 bytes.

Memory allocation…

Let us consider the following simple C program:

```c
#include <stdio.h>

int main() {
    int i;
    char c;
    short s;
    long l;

    printf("i is allocated at %p\n", &i);
    printf("c is allocated at %p\n", &c);
    printf("s is allocated at %p\n", &s);
    printf("l is allocated at %p\n", &l);
}
```

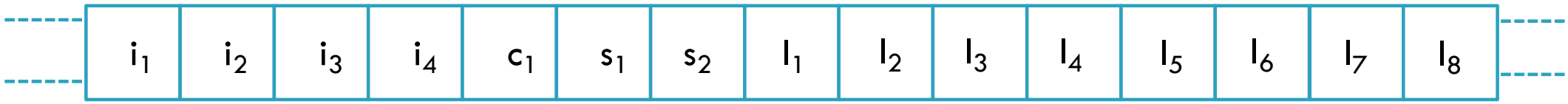
Variable addresses

We can assume that:

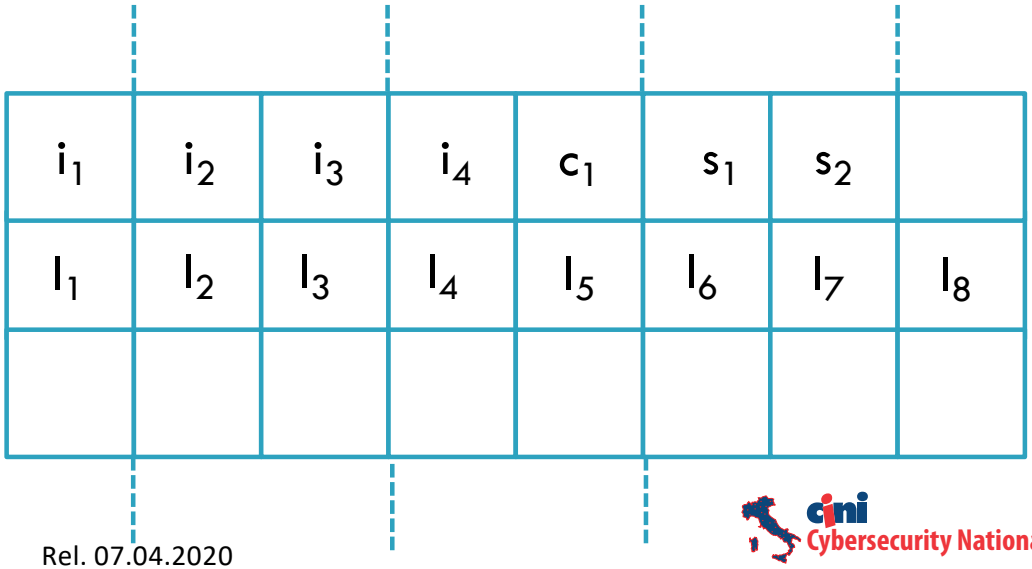
- *int* needs 4 bytes
- *char* needs 1 byte
- *short* needs 2 bytes
- *long* needs 8 bytes.

Memory allocation…

Memory is usually represented as a sequence of bytes:



In a n·8 architecture, bytes are arranged in groups of n:



Memory allocation…


```c
#include <stdio.h>

int main() {
    int i;
    char c;
    short s;
    long l;

    printf("i is allocated at %p\n", &i);
    printf("c is allocated at %p\n", &c);
    printf("s is allocated at %p\n", &s);
    printf("l is allocated at %p\n", &l);
}
```

We can run this simple program to observe how data are allocated in memory.

Memory allocation…

Different allocation policies can be used by *gcc*:

```
[CC> gcc -o alignment alignment.c
[CC> ./alignment
i is allocated at 0x7ffee117e7cc
c is allocated at 0x7ffee117e7cb
s is allocated at 0x7ffee117e7c8
l is allocated at 0x7ffee117e7c0
[CC>
```

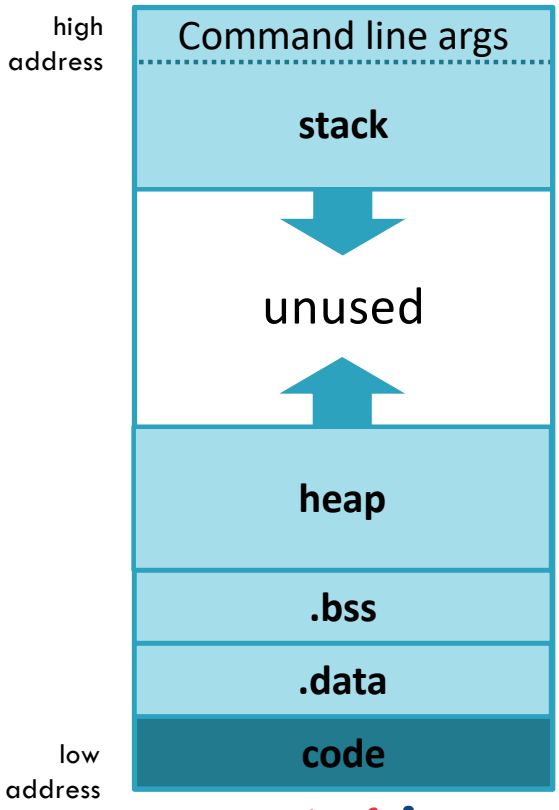
```
[CC> gcc -o alignment alignment.c -O2
[CC> ./alignment
i is allocated at 0x7ffee4dbb7c8
c is allocated at 0x7ffee4dbb7cf
s is allocated at 0x7ffee4dbb7cc
l is allocated at 0x7ffee4dbb7c0
[CC>
```


Data alignment

- Compilers may introduce padding or change the order of data in memory
- There are trade-offs between speed and memory usage.
- *C* compilers provide many optional optimization.

Memory segments

- Memory is allocated for each process (a running program) to store data and code.
- This allocated memory consists of different segments:
  - *stack:* for local variables
  - *heap:* for dynamic memory
  - *data segment:*
    - *global uninitialized variables (.bss)*
    - *global initialized variables (.data)*
  - *code segment*

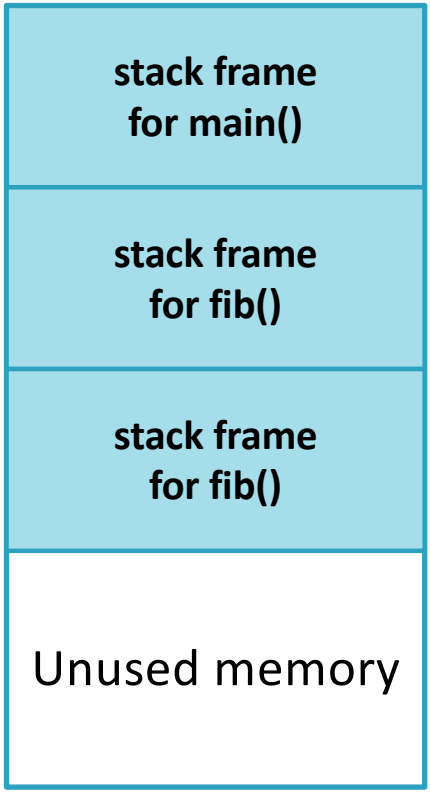


The stack

- The stack consists of a sequence of stack frames (or activation records), each for each function call:
  - allocated on *call*;
  - de-allocated on *return*.

```
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```

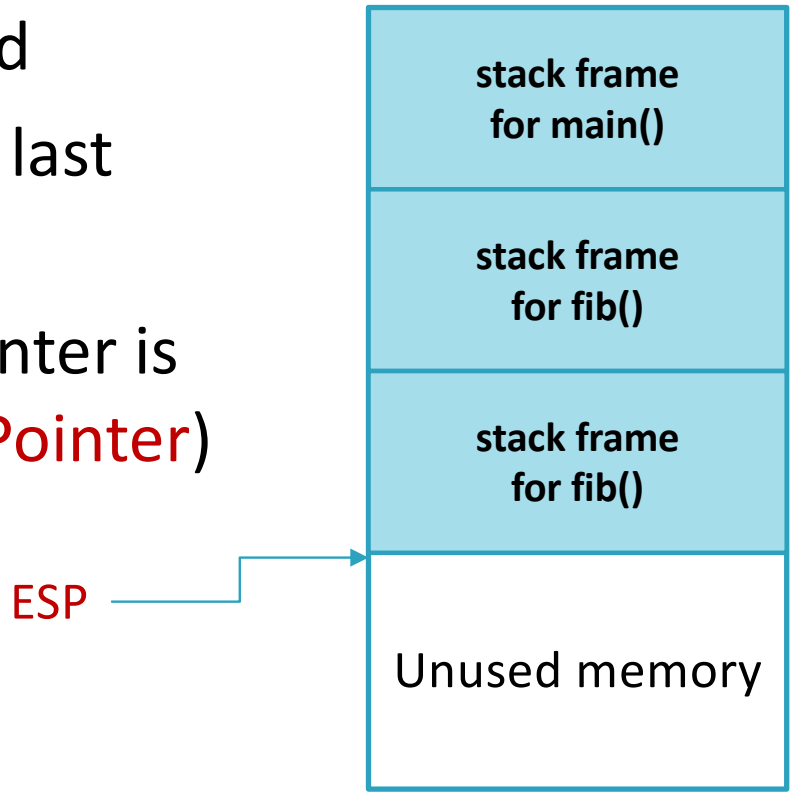


The stack

- The precise structure and organization of the stack depends on system architecture, operating system, and compilers we are using.
- For the sake of simplicity, in this lecture we will focus on *x86* architectures (32 and 64 bits) and on *gcc* compiler on *Linux*.
- More detailed information are available at the following links:
  - https://docs.microsoft.com/en-us/cpp/cpp/argument-passing-and-naming-conventions
  - http://refspecs.linuxbase.org/

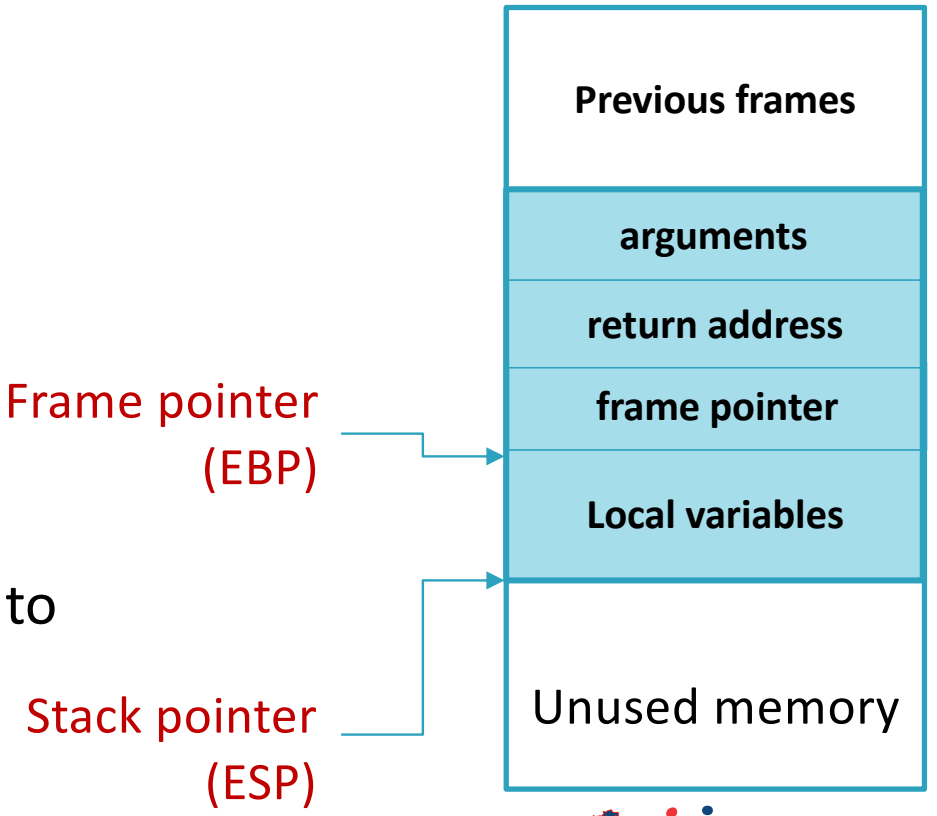
The stack

- Typically the stack grows downward
- The stack pointer (SP) refers to the last element on the stack
- On *x86* architectures, the stack pointer is stored in the ESP (Extended Stack Pointer) register.



Stack frame (for x86)

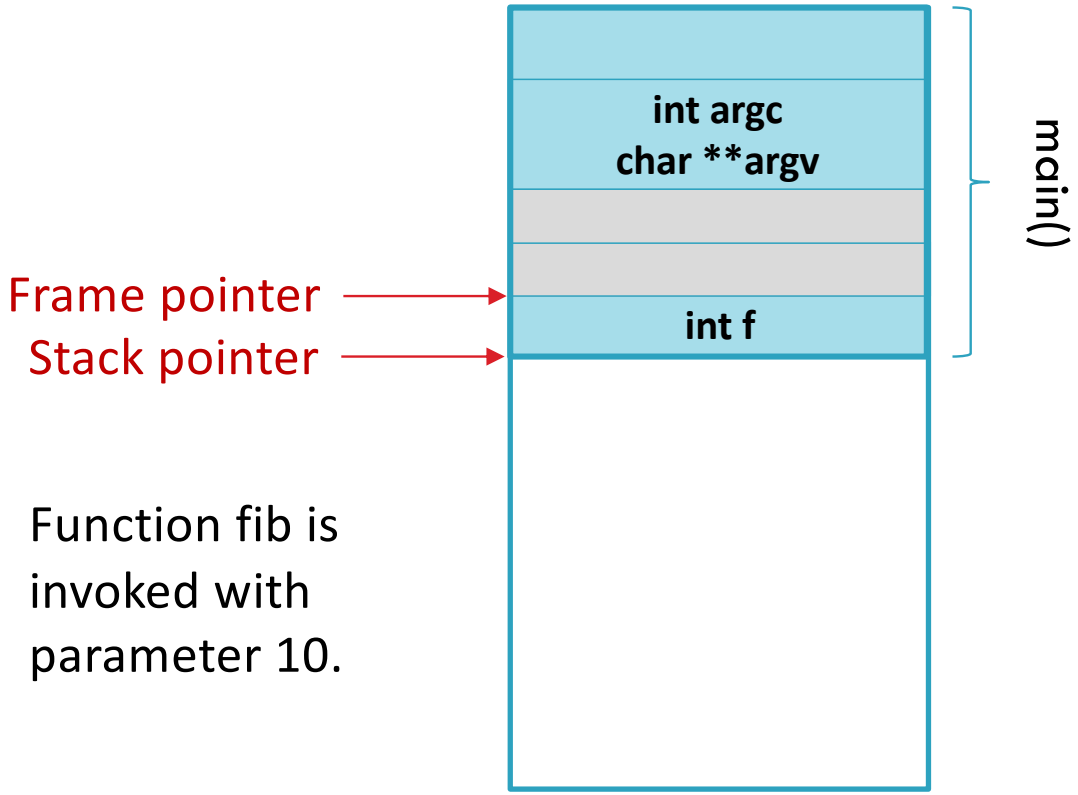
- In *x86* architecture, each stack frame contains:
  - Function arguments;
  - Local variables;
  - Copies of registries that must be restored:
    - return address
    - previous frame pointer
- Frame pointer, named Extended Base Pointer (EBP), provides a starting point to local variables.



Stack frame: example


```
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```



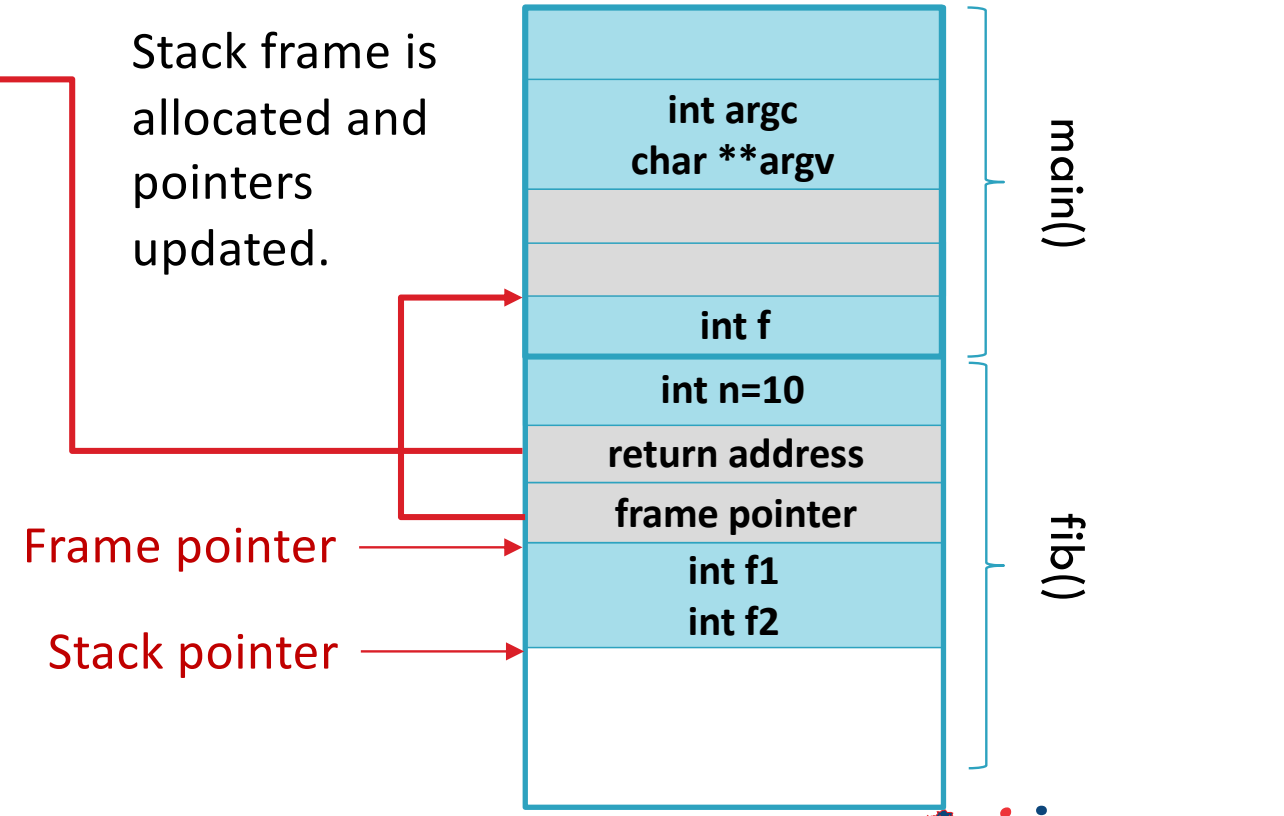
Function fib is invoked with parameter 10.

Stack frame: example


```
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```

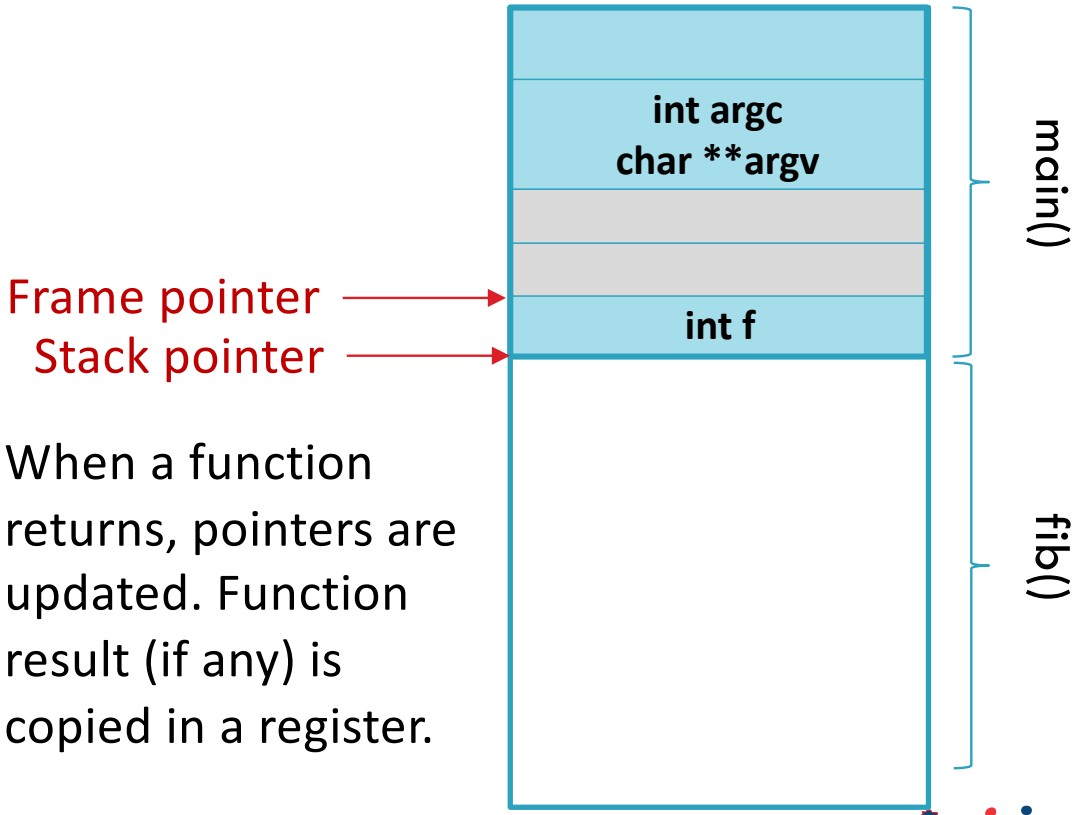
Stack frame is allocated and pointers updated.



Stack frame: example


```
int main(int argc, char **argv) {
  int f = fib( n: 10);
  printf("FIB(10)=%d\n",f);
}

int fib(int n) {
  int f1;
  int f2;
  if (n<=2) {
      return 1;
  } else {
      f1 = fib( n: n-1);
      f2 = fib( n: n-2);
      return f1+f2;
  }
}
```



When a function returns, pointers are updated. Function result (if any) is copied in a register.

Stack frame (for x86-64)

- In the *64-bit* version of *x86:*
  - Registers are extended to 64 bits;
  - A new set of 8 registers are added.
- The arguments are not all in the stack:
  - The first 6 are passed via registers;
  - The remaining are placed in the stack (like for *x86*).
- Pointers *EBP* and *ESP* are named *RBP* and *RSP.*
- A *red zone* of 128 bytes is placed in the stack just under RSP
  - This can be used to store extra local variables and is not modified by interrupt/exception/signal handlers.

C declaration: cdecl

- The cdecl (C declaration) is a calling convention used in many compilers for *x86*
  - A calling convention describes how functions receive their parameters from caller and how they return they results.
- In cdecl arguments are passed on the stack, while values are returned via registers.
- Function arguments are pushed on the stack in the *right-to-left order.*
- The caller *cleans* the stack after the function call returns.

Security issues

- In the stack, variables do not have any default initialization:
  - uninitialized local variables can be used to read values from previous function calls;
- The space assigned to the stack is finite and a function call may fail because there is no more memory:
  - We must guarantee that this does not happen or handle it in a safe way.
- Program and control data are mixed in the stack:
  - By accessing over buffers limit may corrupt the return address!

The heap

- Memory allocation and de-allocation in the stack is very fast
  - However, this memory cannot be used after a function returns
- The heap is used to store dynamically allocated data that outlive function calls:
  - This area is under programmer's responsibility.

Memory management functions

- Basic *C* functions for memory management are:
  - *malloc(int),* given an integer *n* allocates an area of *n* (continuous) byes and returns a pointer to that area;
  - *free(void*)*, deallocates the memory associated with a pointer.

Stack and Heap pointers

- We have always to preserve reference to allocated areas in the heap
  - Otherwise we cannot use them!
- These references (pointers) can be stored in the stack or in the heap.
- We could store pointers to the stack in the heap
  - This can be dangerous! Referenced memory could be released!

Memory security problems

- In *C* memory safety is not guaranteed and malicious or insecure code can access data anywhere on stack and heap:
  - via pointer arithmetic;
  - by overrunning array bounds.
- Security problems with memory are due to:
  - Running out of memory;
  - Lack of initialization of memory;
  - Bugs in program code.

List of common errors (1/3)

- Dereferencing an uninitialized or freed address:

```
int *p;
int z = *p;
```

```
int *p = malloc(sizeof(int));
...
free(p)
...
int z = *p;
```

In both the cases we are going to use an invalid memory location. The result is unpredictable.

List of common errors (2/3)

- Freeing an already freed pointer or a pointer that was not dynamically allocated :

```
int *p = malloc(sizeof(int));
...
free(p)
...
free(p)
```

```
int z = 10;
int *p = &z;
...
free(p)
```

In both the cases the result is unpredictable. It may cause abnormal termination, or memory corruption.

List of common errors (3/3)

- Attempt to access to memory beyond the allocated buffer:

```
int *p1 = malloc(100*sizeof(int));
int *p2 = malloc(200*sizeof(int));

p2=p1;

p2[150]=10;
```

We have an overflow: a memory area outside the buffer is changed!

Good practices

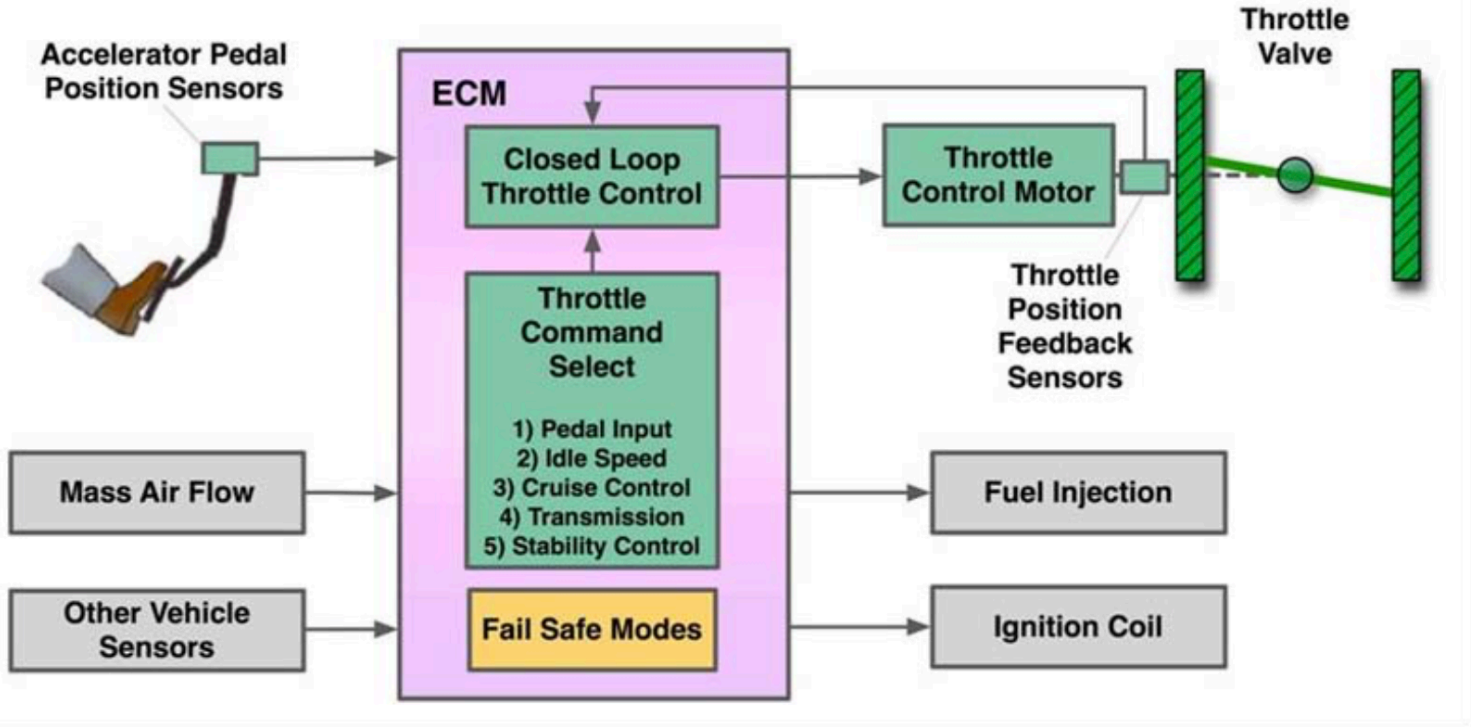
- Every *malloc* should have an associated *free*.
- Pointers should be initialized when defined (either to a valid address or to *NULL)*.
- Pointers should be assigned to *NULL* after being freed.
- Test if a pointer is *NULL* before using it (defensive programming).

Bad stories…

- Bugs in memory allocation can be the source of serious software vulnerabilities.
- Two well known examples are:
  - Toyota unintended acceleration (UA) cases;
  - Heartbleed bug in OpenSSL.

Unintentionally Acceleration

- NASA team investigates UA (2010-2011)



Unintentionally Acceleration

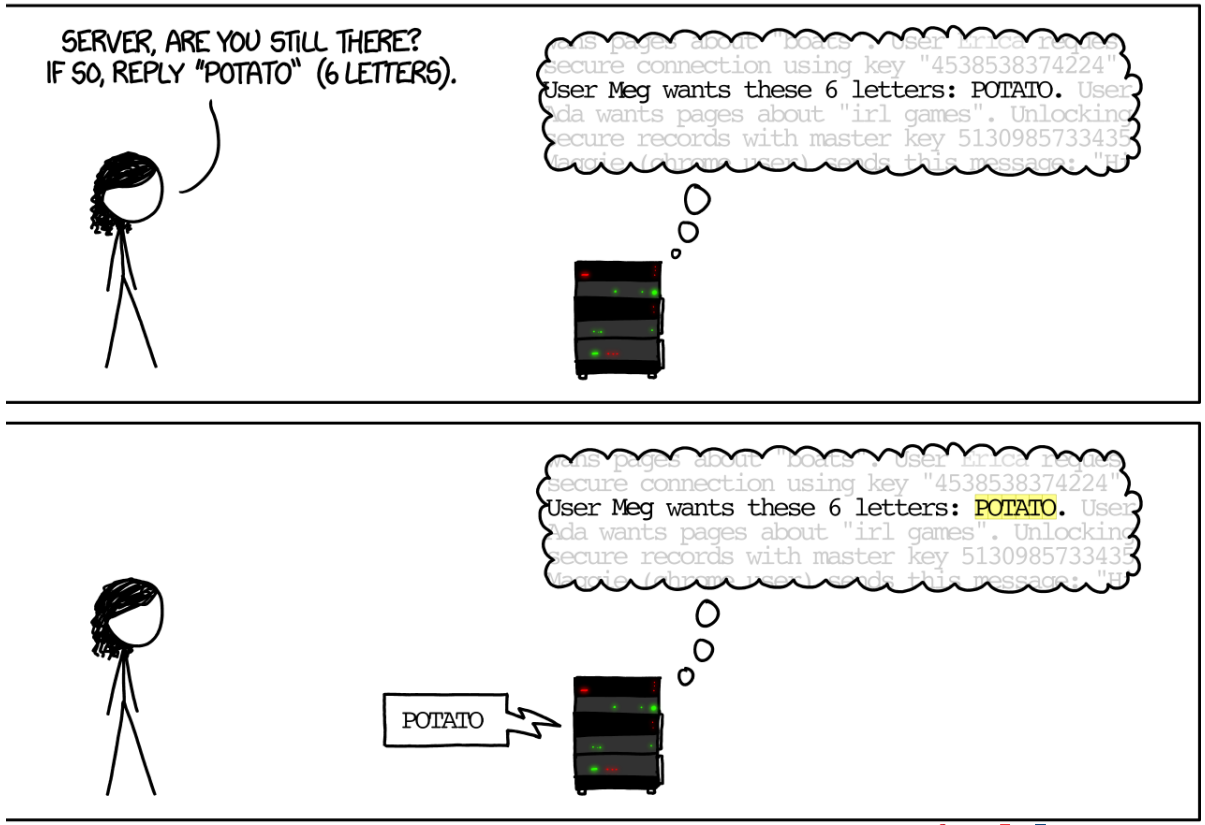
- The bug originated from a stack overflow that may corrupt critical variables of the Operating System.

Heartbleed bug

- The heartbeat functionality in SSL can be used to check if a connection is still alive.
- Malformed input can be provided to OpenSSL to let it print a large part of the stack (possibly containing private keys).
- The same problem effects some TLS implementations (see CVE-2016-9244).

Heartbleed bug

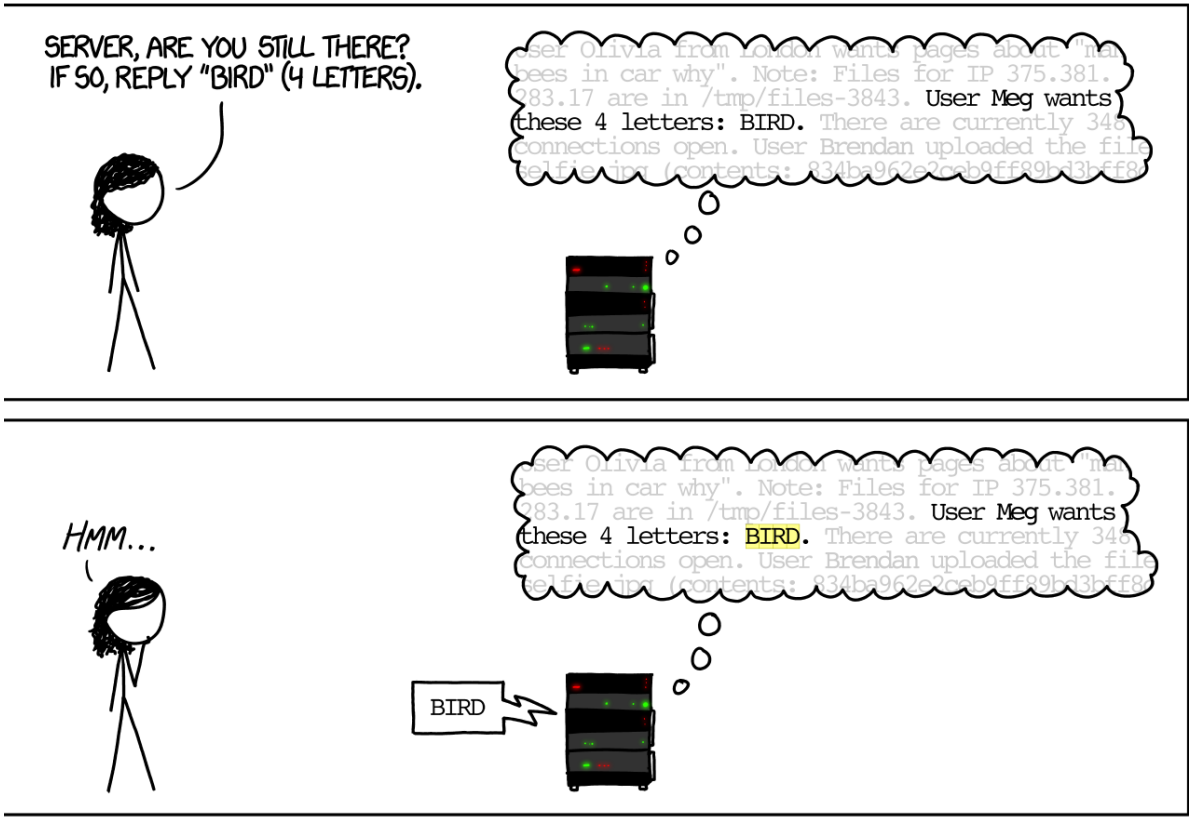
Xkcd provides a nice explanation how the bug works:



https://xkcd.com/1354/

Heartbleed bug

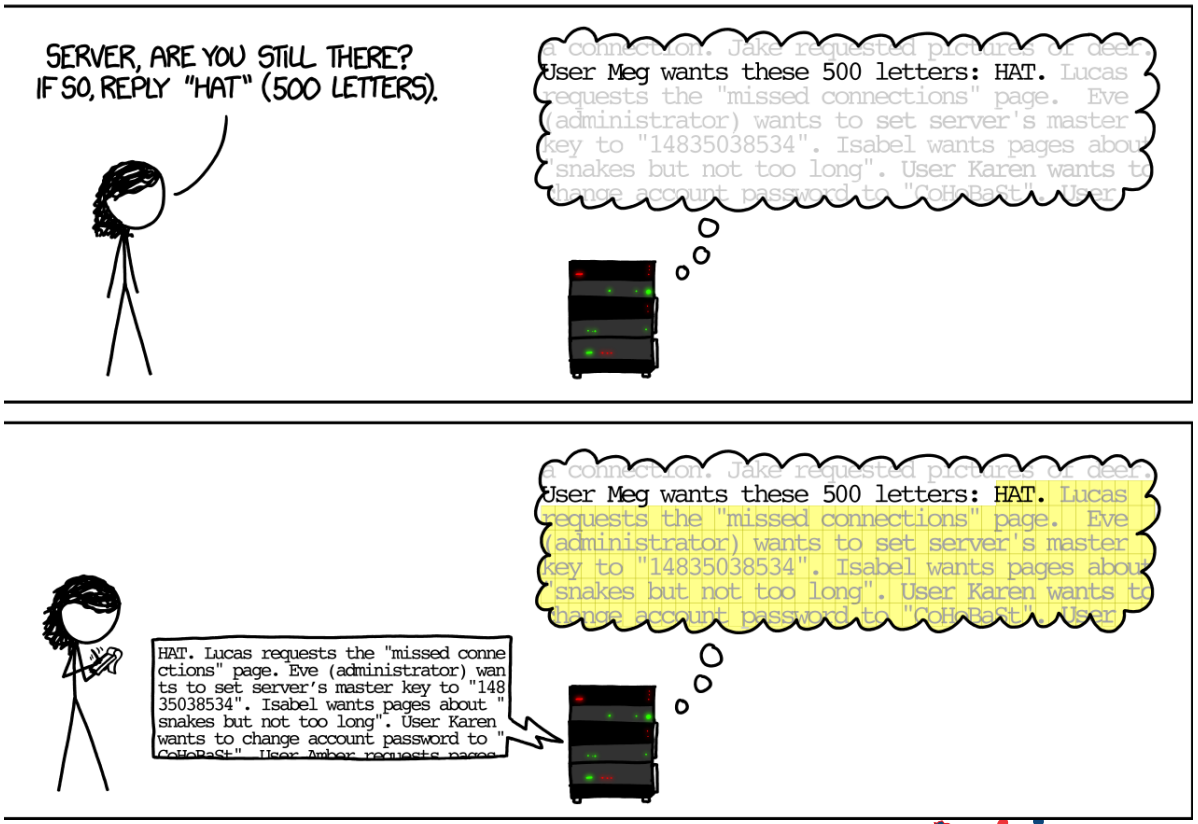
Xkcd provides a nice explanation how the bug works:



https://xkcd.com/1354/

Heartbleed bug

Xkcd provides a nice explanation how the bug works:



https://xkcd.com/1354/

Software Security 2
Memory Management and Allocation

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

https://cybersecnatlab.it

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenturesecurity

aizoon AUSTRALIA EUROPE USA
TECHNOLOGY CONSULTING

B5

EY Building a better working world

eni

exprivia | ITALTEL

IBM

KPMG

LEONARDO

NTT DATA
Trusted Global Innovator

NUMERA
SISTEMI E INFORMATICA S.p.A.

Telsy

SPONSOR GOLD

bip.

cisco

MONTE DEI PASCHI DI SIENA
BANCA DAL 1472

negg

NOVANEXT
connecting the future

pwc

SPONSOR SILVER

DiGi ONE
the leading digital company

ICT CYBER CONSULTING