

# Code: analysis, bugs, and security

supported by Bitdefender

Marius Minea

`marius@cs.upt.ro`

28 September 2016

# Course goals

improve skills: write robust, secure code

understand program internals

learn about security vulnerabilities, detection, prevention

use tools to reverse engineer and analyze code

perhaps in the future: analyze and counter malware

## Code, data, stack, ...

We know the basics:

logically, the program has different memory areas:

- code

- (global) data

- stack (for function calls)

- heap (for dynamic allocation)

What can we find out about them by running a program ?

(look at various addresses printed by `progsegs.c`)

## Program addresses, at first sight

Addresses are in different numeric ranges

Recursive call: new copies for each instance  
can determine size of *stack frame*

Total address range (from code to stack) is HUGE  
orders of magnitude more than computer memory  
⇒ these are *logical* (virtual), not physical addresses

Running the program repeatedly, addresses differ

*Address Space Layout Randomization*

estimate: how many bits vary ?

protects against attacks that need to know address values

# Typical memory layout of C programs

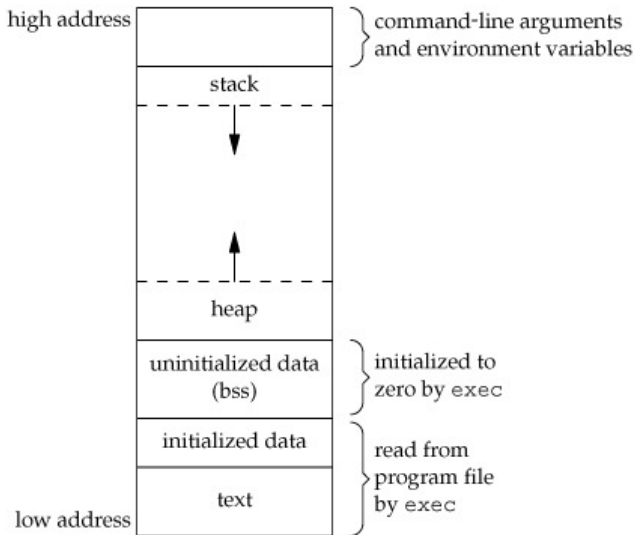
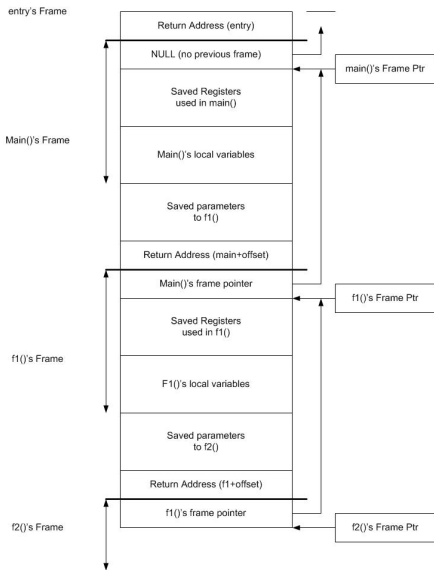


Figure: <http://www.geeksforgeeks.org/memory-layout-of-c-program/>

# Typical stack frame layout

```
void f2(...) {  
  int f1(...) {  
    f2(...);  
  }  
  int main() {  
    ... = f1(...);  
  }  
}
```



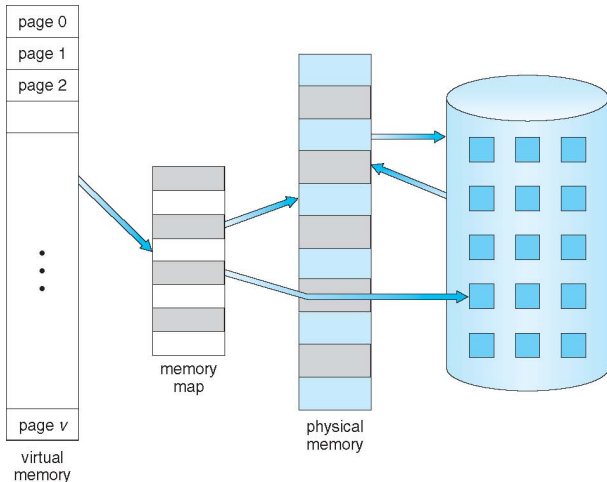
# Virtual memory in a nutshell

A mapping from logical to physical addresses supported by processor hardware (memory management unit) and operating system

- provides *abstraction* (program need not worry about size and usage of physical memory)
  - virtual address space can be larger than physical memory
  - memory *pages* transferred to/from secondary memory (disk) as needed
- provides *protection*
  - can set up *permissions* for memory segments
  - memory space of one process protected from another
  - but: can also set up *sharing*



# Virtual Memory That is Larger Than Physical Memory





# Address Translation

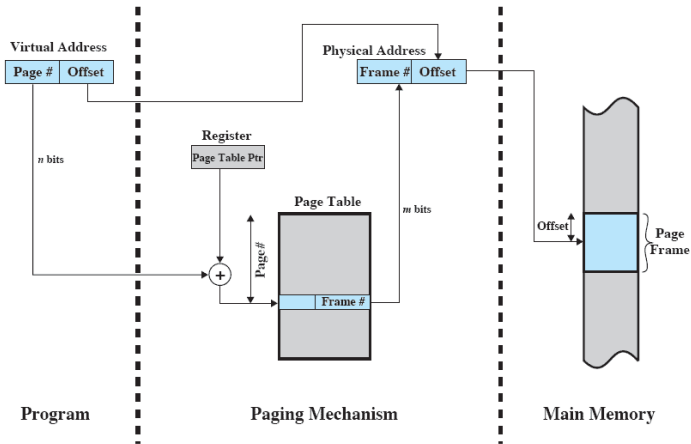


Figure 8.3 Address Translation in a Paging System

## Arrays and pointers

What difference (if any) is there between

`char s[] = "test";`      and      `char *p = "test";` ?

## Arrays and pointers

What difference (if any) is there between

`char s[] = "test";`      and      `char *p = "test";` ?

*Array*: `char s[] = "test";`      `s[0]` is 't', `s[4]` is '\0' etc.

`s` is a *constant address* (`char *`), not a variable in memory

CANNOT assign `s = ...` but may assign `s[0] = 'f'`

`sizeof(s)` is `5 * sizeof(char)`

`&s` is `s`, but different type, address of 5-char array: `char (*)[5]`

<code>sizeof (entire array) is not <code>strlen</code> (up to '\0')</code>
--

## Arrays and pointers

What difference (if any) is there between

`char s[] = "test";`      and      `char *p = "test";` ?

*Array:* `char s[] = "test";`      `s[0]` is 't', `s[4]` is '\0' etc.

`s` is a *constant address* (`char *`), not a variable in memory

CANNOT assign `s = ...` but may assign `s[0] = 'f'`

`sizeof(s)` is `5 * sizeof(char)`

`&s` is `s`, but different type, address of 5-char array: `char (*) [5]`

<code>sizeof (entire array) is not <code>strlen</code> (up to '\0')</code>
--

*Pointer:* `char *p = "test";`      `p[0]` is 't', `p[4]` is '\0' (same)

`p` is a *variable of address type* (`char *`), has a memory location

CANNOT assign ~~`p[0] = 'f'`~~ ("`test`" is a string *constant*)

can do `p = s;` then `p[0] = 'f'`;      can assign `p = "ana";`

`sizeof(p)` is `sizeof(char *)`      `&p` is NOT `p`

⇒ WRONG: ~~`scanf("%4s", &p);`~~      RIGHT: `scanf("%4s", p);`

(if `p` is valid address and has room)

# Arrays and pointers

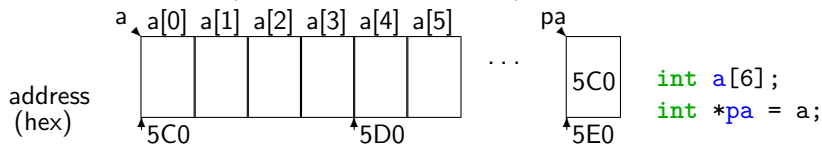
The *name of an array* is a *constant address*

Can declare `int a[LEN], *pa;` and assign `pa = a;`

Similar: `a` and `pa` have same type: `int *`

But: `pa` is a *variable*  $\Rightarrow$  uses memory; *can assign* `pa = addr`

`a` is a *constant* (array has fixed address) *can't assign* ~~`a = addr`~~



`*a` and `*pa`: indirections with different operations in machine code:

- `*a` references object from *constant* address (*direct* addressing)

- `*pa` must first get *value* of variable `pa` (an address), loading it from the *constant* address `&pa`) *then* dereference it (*indirect* addressing)

# Binary data representation

Suppose we want to process a bitmap file

## Bitmap file header

This block of bytes is at the start of the file and is used to identify the file. A typical application reads this block first to ensure that the file is actually a BMP file and that it is not damaged. The first 2 bytes of the BMP file format are the character "B" then the character "M" in ASCII encoding. All of the integer values are stored in little-endian format (i.e. least-significant byte first).

Offset hex	Offset dec	Size	Purpose
00	0	2 bytes	The header field used to identify the BMP and DIB file is 0x42 0x4D in hexadecimal, same as BM in ASCII. The following entries are possible: <ul style="list-style-type: none"><li>▪ <b>BM</b> – Windows 3.1x, 95, NT, ... etc.</li><li>▪ <b>BA</b> – OS/2 struct bitmap array</li><li>▪ <b>CI</b> – OS/2 struct color icon</li><li>▪ <b>CP</b> – OS/2 const color pointer</li><li>▪ <b>IC</b> – OS/2 struct icon</li><li>▪ <b>PT</b> – OS/2 pointer</li></ul>
02	2	4 bytes	The size of the BMP file in bytes
06	6	2 bytes	Reserved; actual value depends on the application that creates the image
08	8	2 bytes	Reserved; actual value depends on the application that creates the image
0A	10	4 bytes	The offset, i.e. starting address, of the byte where the bitmap image data (pixel array) can be found.

## Bitmap file format (cont'd)

Offset (hex)	Offset (dec)	Size (bytes)	Windows BITMAPINFOHEADER <sup>[1]</sup>
0E	14	4	the size of this header (40 bytes)
12	18	4	the bitmap width in pixels (signed integer)
16	22	4	the bitmap height in pixels (signed integer)
1A	26	2	the number of color planes (must be 1)
1C	28	2	the number of bits per pixel, which is the color depth of the image. Typical values are 1, 4, 8, 16, 24 and 32.
1E	30	4	the compression method being used. See the next table for a list of possible values
22	34	4	the image size. This is the size of the raw bitmap data; a dummy 0 can be given for BI_RGB bitmaps.
26	38	4	the horizontal resolution of the image. (pixel per meter, signed integer)
2A	42	4	the vertical resolution of the image. (pixel per meter, signed integer)
2E	46	4	the number of colors in the color palette, or 0 to default to $2^n$
32	50	4	the number of important colors used, or 0 when every color is important; generally ignored

To work with ints that are *exactly* 2 bytes, 4 bytes, etc., need *fixed-width integers*: `stdint.h` (since C99)  
`int8_t`, `int16_t`, `int32_t`, `int64_t`,  
`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

## Big-endian and little-endian

BMP specification: “all integers are stored in little-endian format”

little-endian: least-significant byte first

0x12345678 is stored as 0x78 0x56 0x34 0x12

Intel x86

big-endian: most-significant byte first

0x12345678 is stored as 0x12 0x34 0x56 0x78

Mac, PPC, Sun, Internet (also called 'network byte order')

Make sure values are read/written from/to file in correct byte order



## We'll use: Program analysis infrastructures

Allow program representation and manipulation  
at source or binary level

Built-in analyses + API to write your own

LLVM: one of the most widely used, complete compiler toolchain

BAP (D. Brumley, CMU): OCaml + Python bindings  
team won DARPA Cyber Grand Challenge 2016

angr (UC Santa Barbara): Python framework

CIL (G. Necula, Berkeley): OCaml + Perl  
outputs instrumented C code

## Example of source instrumentation with CIL

Analysis library provides a data type to represent statements

```
type stmtkind =  
  | Instr of instr list      (* straight-line instructions *)  
  
  | Return of exp option * location (* The return statement. *)  
  
  | Goto of stmt ref * location   (* A goto statement. *)  
  
  | Break of location  (* break to end of nearest loop/switch *)  
  
  | Continue of location (* continue to start of nearest loop *)  
  
  | If of exp * block * block * location (* A conditional. *)  
  
  | Switch of exp * block * (stmt list) * location  
  
  | Loop of block * location * (stmt option) * (stmt option)  
  (* a while(1) loop with continue and break *)  
  | Block of block          (* block of statements. *)
```

## C representation in CIL (cont'd.)

```
type instr =
  | Set    of lval * exp * location (* An assignment. *)
  | Call  of lval option * exp * exp list * location (* fct

and lval = lhost * offset (* base address + field/index *)

and lhost =
  | Var    of varinfo      (* base is a variable *)
  | Mem    of exp          (* access through a pointer *)
```

To instrument code, traverse statements (control flow graph), identify interesting statements, insert new ones.

e.g. can log all/some memory writes

## Fast compiler-supported instrumentation

Address sanitizer (with recent clang / gcc versions)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *p = malloc(20);
    strcpy(p, "test");
    puts(p);
    free(p);
    p[1] = 'a'; // wrong
}
```

```
% gcc -fsanitize=address usefree.c
```

```
% ./a.out
```

```
==31741==ERROR: AddressSanitizer: heap-use-after-free on
address 0x60300000efe1 at pc 0x0000004008c6 bp
0x7ffeef2227b0 sp 0x7ffeef2227a8
```

```
WRITE of size 1 at 0x60300000efe1 thread T0
```

```
#0 0x4008c5 in main /home/marius/curs/bitdef/usefree.c:11
```

## Software security: increasingly automated

automated vulnerability detection + exploit generation

comparison of old (buggy) + patched program versions

⇒ exploit generation

'compilers' for return-oriented programming exploits

A good read (insights into research advances):

G. Vigna et al., (State of) The Art of War: Offensive Techniques in Binary Analysis, IEEE Security & Privacy, 2016