Ingeniería Inversa
Clase 0

Presentación

martin.uy
Open by default.

¡Hola!

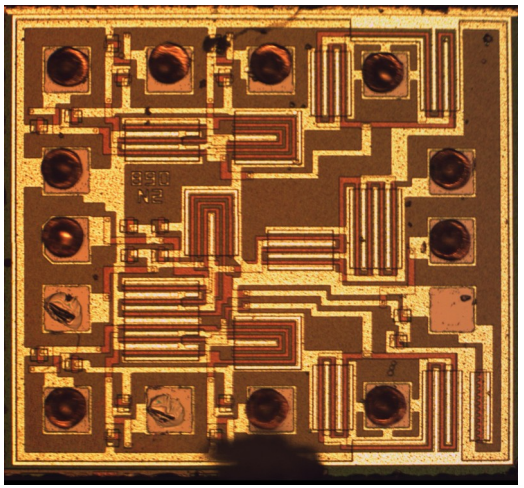
- ¿Nombre?
- ¿Intereses profesionales?
  - ¿Lenguajes?
  - ¿Tecnologías?
- ¿Trabajo?
- ¿Proyectos de tiempo libre?
- ¿Expectativas para el curso?

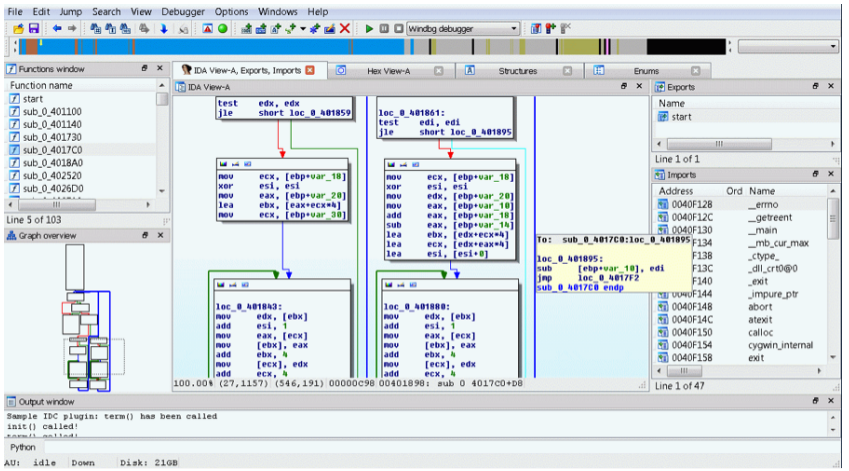
Ingeniería Inversa

*“estudiar o analizar (un dispositivo, como un microchip para computadoras) para **aprender** los detalles del diseño, construcción y operación, y tal vez para producir una copia o una **versión mejorada**” **

* Random House Dictionary, 2017

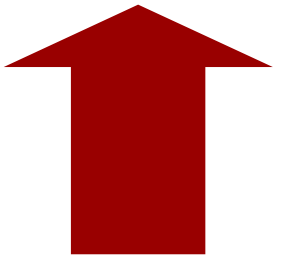
Ingeniería Inversa





Hardware

Software



Trabajos

- Consultor de Seguridad
- Analista de Malware
- Investigador de Seguridad
- Red Team
- Reverse Engineer
- Exploit Writer

Objetivos del curso

- Reversear binarios ejecutables
- Analizar malware binario
- Buscar vulnerabilidades
- Explotar vulnerabilidades

- Aprender sobre APIs, ABIs, formatos binarios, técnicas de reversing, debugging, lenguajes de implementación de sistemas (C/C++), herramientas y ambientes de trabajo.

Nice-to-haves

- Conocimientos deseados
  - C/C++
  - Sistemas operativos (Windows, Linux)
  - Arquitecturas x86 y x86_64
  - Debuggers
- Habilidades blandas
  - Metodología, sistematicidad y perseverancia
  - Motivación
  - Preparación de ambientes de trabajo adecuados
  - Heurística e intuición

Organización del curso

- 1 clase de introducción
- 10 clases teórico-prácticas
- 4 clases de proyecto (a elección)
  - CTFs / Crackmes binarios
  - Análisis o desarrollo de malware
  - Desarrollo de un fuzzer
  - ¿Otra idea?

Organización del curso (2)

- Fechas importantes
  - Elección del proyecto
  - Entrega del proyecto
  - Fin del curso

Syllabus

- Módulo 1: Binarios ejecutables (3 clases)
  - ELF, PE, análisis estático y dinámico
- Módulo 2: Análisis de malware (2 clases)
  - Desarrollo, unpacking e inyección en procesos
- Módulo 3: Bug hunting (2 clases)
  - Fuzzing, instrumentación binaria y análisis dinámico

Syllabus (2)

- Módulo 4: Explotación binaria (3 clases)
  - Stack overflow, integer overflow, use-after-free y ROP chain

Materiales

- Virtual Box VM (Linux)
  - Provisto por el curso
- Windows 7 (virtual o físico)
  - Visual Studio Express
  - IDA Pro demo
  - API Monitor
  - CFF Explorer
  - Wireshark

Canales de comunicación

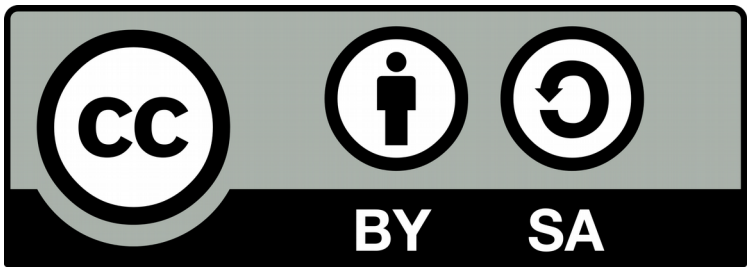
- Web
  - martin.uy/reverse
    - Slides actualizados

- Mail

m@rtin.uy

Licencia

- Todos los materiales del curso están licenciados con **Creative Commons Attribution-ShareAlike International 4.0**
  - creativecommons.org



- Bienvenidos los aportes :-)

Software libre

- Libertad de uso
- Libertad de estudiar y modificar
- Libertad de distribuir
- Libertad de mejorar

FSF

FREE SOFTWARE FOUNDATION

¿Por qué unirse a un proyecto de software libre?

open source initiative

fsf.org | opensource.org

We have cookies!

- Curso Ingeniería Inversa
- Grupo de desarrollo de software libre
  - Glibc
  - OpenJDK
- Proyectos finales de carrera

Linux VM Lab Work

Presentación de la VM del curso
Virtual Box

Linux VM Lab Work

- Fedora 25 – x86_64
  - 4 GB RAM mínimo
  - 100 GB HDD máximo
  - 2+ CPUs recomendado
  - Credenciales: user/1234
- Ambiente de desarrollo, deploy y debugging
  - Linux kernel
  - Glibc
- Ver documento “README_VM”

Linux VM Lab Work

- Virtual Machine Manager (qemu)
  - Linux_VM_Lab_Target
    - Fedora 25 (x86_64)
    - IP: 192.168.122.2
    - Credenciales: test/1234
  - Binary translation → lento para una interfaz gráfica pero suficiente para command line

C

- Dennis Ritchie
  - 1941 – 2011
  - Ph.D. Harvard University
  - Co-creador de Unix (Bell Labs)
  - Turing award 1983
- The C Programming Language
  - Dennis Ritchie & Brian Kernighan
  - 1a edición 1978
  - Lectura recomendada

C

- Lenguaje estandarizado
  - ISO/IEC
  - C89, C90, C95, C99, C11
  - Portabilidad (múltiples plataformas)
  - Componentes
    - Lenguaje (sintáxis y semántica)
    - Librerías

C

- Lenguaje imperativo, estructurado, estáticamente tipado
- De propósito general y relativamente de “bajo nivel”
  - Implementación de sistemas
  - Sistemas operativos
  - Compiladores
  - Máquinas virtuales (ej. CPython)
  - “La mayor parte del código importante está en C” (*)

(*) Dr. Thomas Schwarz

C

- Simple y fácil pero potente
- Multiplataforma (con cierta precaución)
- Se compila a código nativo de la arquitectura (generalmente)
- No hay garbage collector: el programador gestiona la memoria (así como otros recursos)

(*) Thomas Schwarz

C

- Estructura
  - Headers (.h)
    - Declaración de variables, funciones y tipos de datos externos (de otros objetos o librerías compartidas)
  - Implementación (.c)
    - Declaración de variables, funciones y tipos de datos internos del objeto (criterio de encapsulamiento)
    - Definición e inicialización de variables exportadas
    - Implementación de funciones exportadas
  - Al final del día, los headers (.h) son texto que se incluye en los archivos de implementación (.c)

C

- Macros de pre-procesamiento
  - Edición a nivel texto, antes de la compilación

```
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#define CONST_1 1

/* … */

#endif // HEADER_H
```

C

- Algunos operadores (expresiones)
  - Aritméticos
    - +, *, /, -, % (binarios) y ++, - -, (unarios)
  - Booleanos
    - && (AND), || (OR), ! (NOT), == (EQ), != (NEQ), >=, <=
  - Bits
    - ^ (XOR), | (OR), ~ (NOT), & (AND), << y >> (shift)
  - Condicional
    - (condición) ? caso-verdadero : caso-falso
  - Asignación (=, +=, -=, *=, %=, etc.)

C

- Algunos operadores (expresiones)

```
int a = 0x0;
int b = 0xFFFFFFFF;
a |= (1 << 2);
b &= ~(1 << 2);
```

¿Qué se está haciendo con a?

¿Qué se está haciendo con b?

C

- Algunos operadores (expresiones)

```
int a = 0x0;
int b = 0xFFFFFFFF;
a |= (1 << 2);
b &= ~(1 << 2);
```

a = setear un 1 en el bit 3 (desde la derecha)

b = setear un 0 en el bit 3 (desde la derecha)

C

- Constantes
  - Long
    - 1L
  - Unsigned
    - 1U
  - Unsigned long
    - 1UL
  - Float
    - 1.0f, 1e-2
  - Hex
    - 0x1

C

- Constantes
  - Octal
    - 01
  - Caracteres
    - ‘0’ (valor en ASCII), ‘\n’, ‘\t’, ‘\0’, ‘\x...’ (# byte), etc.
  - String
    - “abc”
    - ¿Cuál es la diferencia entre “x” y ‘x’?

C

- Tipos de datos
  - long
  - int
  - short
  - char
  - float / double
  - struct abc {

    …

    }

C

- Tipos de datos
  - void(*)(void) / void*
  - enum abc { … }
  - typedef type_1 type_2

```
typedef struct a {
    int m1;
} a_t;
```

C


```
struct a {
    int a_1;
};
```

Agregación de datos

```
union b {
    int b_1;
    char b_2;
};
```

Tamaño del miembro de mayor tamaño. Se usan asociados a un contexto que permite conocer cuál es el tipo válido de la unión.

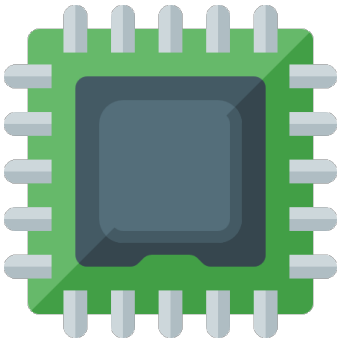
```
enum c {
    c_1 = 0,
};
```

El tipo del enum lo decide el compilador en su implementación. Ejemplo: int.

C


```
enum a_e { A = 1, B, C } a;
struct b {
    int a;
    int b;
} b;
union c {
    char d;
    int e;
} c;

b.a = (int)A;
b.b = 2;
c.d = 60;
c.e = 61;
```

ASM (x86_64)



```
0804840b <main>:
 804840b:       8d 4c 24 04                   lea    0x4(%esp),%ecx
 804840f:       83 e4 f0                      and    $0xfffffff0,%esp
 8048412:       ff 71 fc                      pushl  -0x4(%ecx)
 8048415:       55                            push   %ebp
 8048416:       89 e5                         mov    %esp,%ebp
 8048418:       51                            push   %ecx
 8048419:       83 ec 14                      sub    $0x14,%esp
 804841c:       c7 45 f0 01 00 00 00          movl   $0x1,-0x10(%ebp)
 8048423:       c7 45 f4 02 00 00 00          movl   $0x2,-0xc(%ebp)
 804842a:       c6 45 ec 3c                   movb   $0x3c,-0x14(%ebp)
 804842e:       c7 45 ec 3d 00 00 00          movl   $0x3d,-0x14(%ebp)
 8048435:       83 ec 08                      sub    $0x8,%esp
 8048438:       6a 01                         push   $0x1
 804843a:       68 14 85 04 08                push   $0x8048514
 804843f:       e8 9c fe ff ff                call   80482e0 <printf@plt>
```

C


```
printf("sizeof(long): %d\n", sizeof(long));
printf("sizeof(int): %d\n", sizeof(int));
printf("sizeof(short): %d\n", sizeof(short));
printf("sizeof(char): %d\n", sizeof(char));
printf("sizeof(double): %d\n", sizeof(double));
printf("sizeof(float): %d\n", sizeof(float));

printf("sizeof(struct a): %d\n", sizeof(struct a));
printf("sizeof(union b): %d\n", sizeof(union b));
printf("sizeof(enum c): %d\n", sizeof(enum c));
```

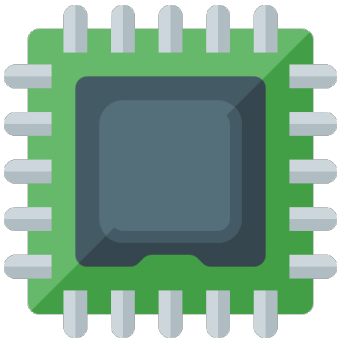
¿Tenemos información suficiente para decidir cuáles son los tamaños de estos tipos de datos?

C


```
64 bits
sizeof(long): 8
sizeof(int): 4
sizeof(short): 2
sizeof(char): 1
sizeof(double): 8
sizeof(float): 4
sizeof(void*): 8
sizeof(struct a): 4
sizeof(union b): 4
sizeof(enum c): 4
```

```
32 bits
sizeof(long): 4
sizeof(int): 4
sizeof(short): 2
sizeof(char): 1
sizeof(double): 8
sizeof(float): 4
sizeof(void*): 4
sizeof(struct a): 4
sizeof(union b): 4
sizeof(enum c): 4
```

ASM (x86_64)



```
void* d = (void*)-1;

long e = -1L;

int f = -1;

short g = -1;

char h = -1;
```

```
nop
movq   $0xffffffffffffffff,-0x8(%rbp)
nop
movq   $0xffffffffffffffff,-0x10(%rbp)
nop
movl   $0xffffffff,-0x14(%rbp)
nop
movw   $0xffff,-0x16(%rbp)
nop
movb   $0xff,-0x17(%rbp)
```

C

- Declarar (funciones y variables)
  - Antes de usarlas
  - Especificar tipos (ej. int a)
- Inicializar variables
  - Asignar valor (ej. a = 1)
  - Variables globales: 0 o NULL por defecto
  - Variable locales: basura por defecto
- Se puede declarar e inicializar variables al mismo tiempo (ej. int a = 1)

C

- Scope
  - Local (a una función)
  - Objeto (static)
  - Global
- Estructuras de control de flujo (if, for, while, do-while, switch, break, goto, return)

C

- Const correctness

```
const int a = 1;

const int *b = &a;

char *c = "abc";

a = 2; // Se puede?

*b = 3; // Se puede?

b = (int*)0x0; // Se puede?

c[0] = 'b'; // Se puede?
```

C

- Const correctness

```
const int a = 1;

const int *b = &a;

char *c = "abc";

a = 2; // Se puede?

*b = 3; // Se puede?

b = (int*)0x0; // Se puede?

c[0] = 'b'; // Se puede?
```

Compila

Ejecuta

C

- Const correctness

```
const int *d = (const int*)0x1;

const int *const e = (const int*)0x1;

int *const f = d; // Se puede?

int *g = d; // Se puede?

*e = 2; // Se puede?

e = (const int*)2; // Se puede?

*f = 2; // Se puede?
```

C

- Const correctness

```
const int *d = (const int*)0x1;

const int *const e = (const int*)0x1;

int *const f = d; // Se puede?
```

Se descarta el calificador “const”

```
int *g = d; // Se puede?
```

Se descarta el calificador “const”

```
*e = 2; // Se puede?
```

```
e = (const int*)2; // Se puede?
```

```
*f = 2; // Se puede?
```

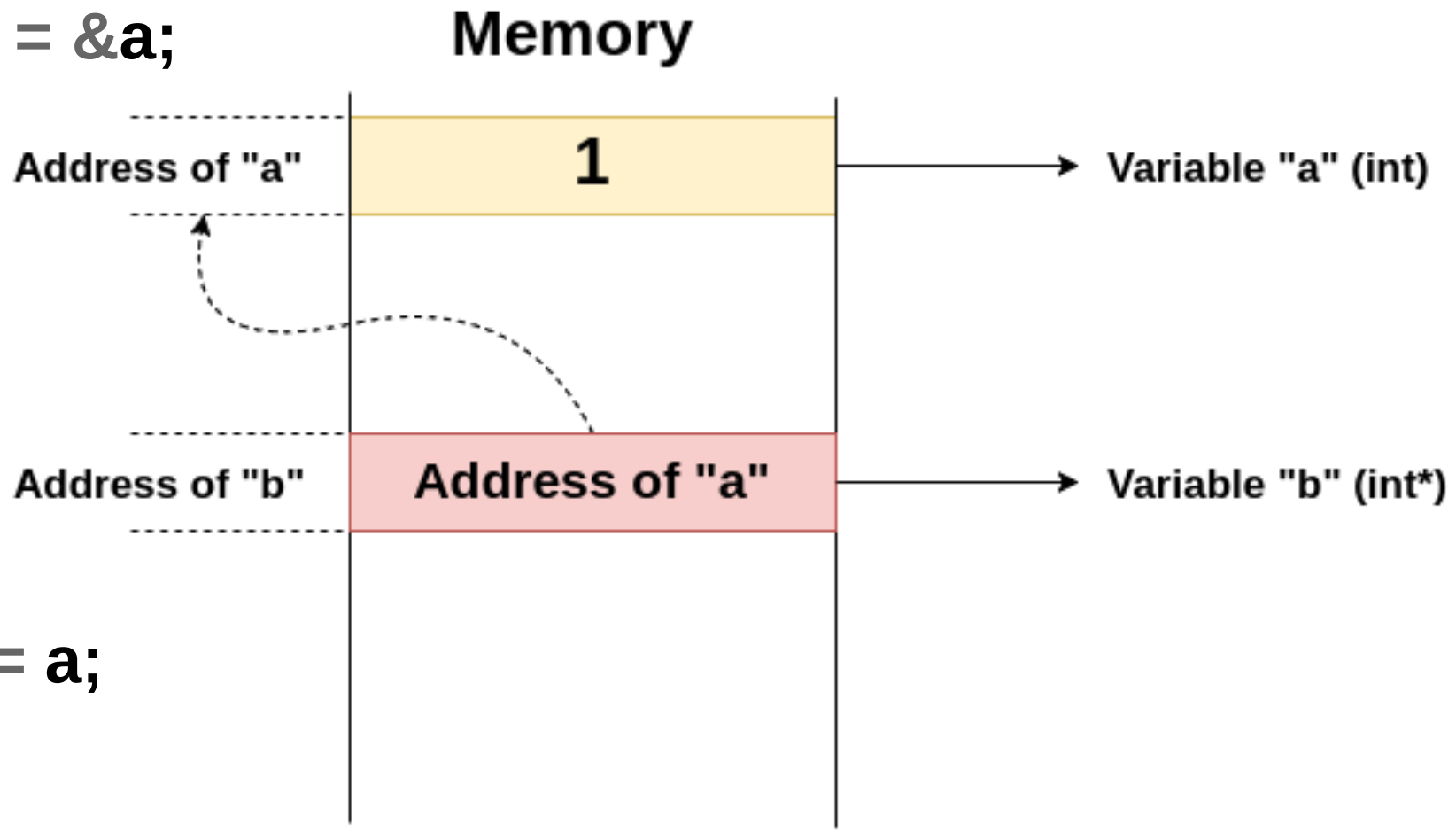
Compila

Ejecuta

C

- Punteros

```
int a = 1;
int *b = &a;
```




```
*b == a;
```

C

- Punteros

```
int a = 1;
int *b = &a;
a = 2;

printf("a: %d, b: %d\n", a, *b);

*b = 3;
printf("a: %d, b: %d\n", a, *b);

b = (int*)0x4;
printf("b: %d\n", *b);
```

C

- Punteros

```
int a = 1;
int *b = &a;
a = 2;

printf("a: %d, b: %d\n", a, *b);

*b = 3;
printf("a: %d, b: %d\n", a, *b);

b = (int*)0x4;
printf("b: %d\n", *b);
```

```
a: 2, b: 2
a: 3, b: 3
Segmentation fault (core dumped)
```

C

- Operadores para punteros

```
struct a {
    int m1;
};

struct a v1;
struct a *v2 = &v1;

v1.m1 = 0;
v2->m1 = 1; // Equivalente a (*v2).m1 = 1;
```

C

- Aritmética de punteros

```
int *a = (int*)0x0;
short *b = (short*)0x0;
int *c = (int*)0x0;

a = a + 1;
b = b + 1;
c = (int*)((char*)c + 1);

printf("a: %p, b: %p, c: %p\n", a, b, c);
```

C

- Aritmética de punteros

```
int *a = (int*)0x0;
short *b = (short*)0x0;
int *c = (int*)0x0;

a = a + 1;
b = b + 1;
c = (int*)((char*)c + 1);

printf("a: %p, b: %p, c: %p\n", a, b, c);
```

a + sizeof(int)

```
a: 0x4, b: 0x2, c: 0x1
```

C

- Casting

```
char a = -1;
unsigned char b = -1;

printf("(int)a: %d, (int)b: %d\n", (int)a, (int)b);

printf("(unsigned int)a: %u, (unsigned int)b: %u\n",
(unsigned int)a, (unsigned int)b);
```

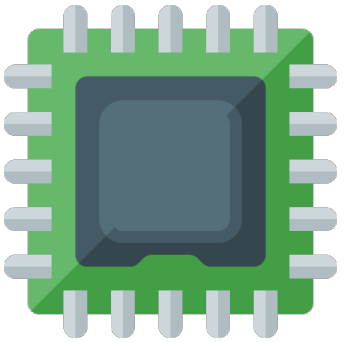
C

- Casting

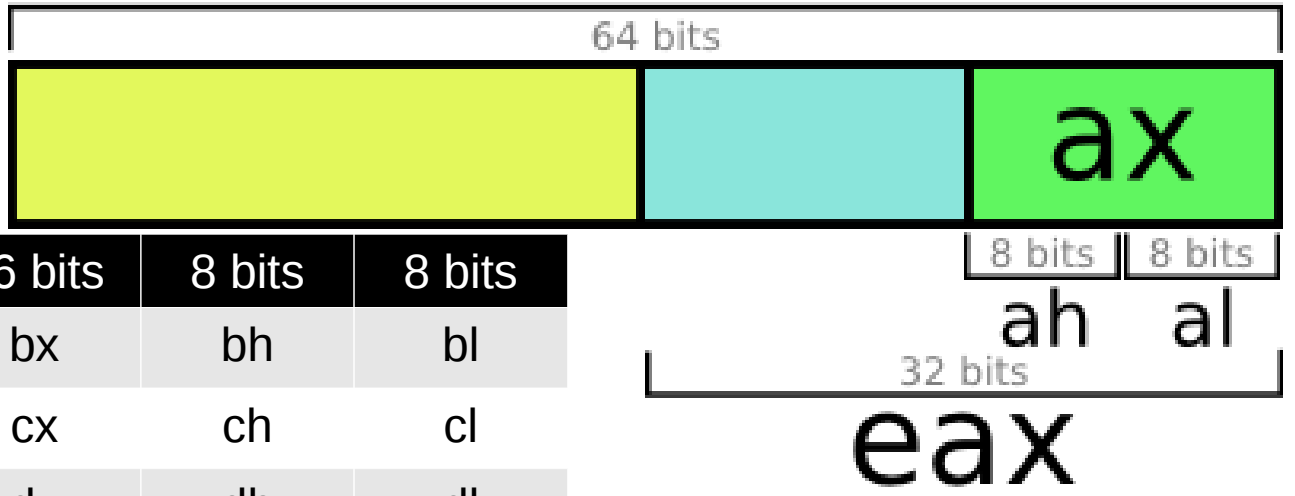
```
char a = -1;
unsigned char b = -1;

printf("(int)a: %d, (int)b: %d\n", (int)a, (int)b);

printf("(unsigned int)a: %u, (unsigned int)b: %u\n",
(unsigned int)a, (unsigned int)b);
```

```
(int)a: -1, (int)b: 255
(unsigned int)a: 4294967295, (unsigned int)b: 255
```

ASM (x86_64)



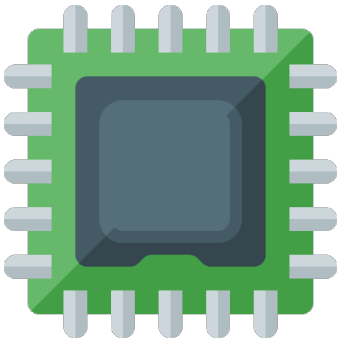
rax



64 bits	32 bits	16 bits	8 bits	8 bits
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rbp	ebp	bp	-	-
rsp	esp	sp	-	-
rsi	esi	si	-	-
rdi	edi	di	-	-
r8	r8d	r8w	-	r8b
...	...	...	...	...

Imagen de http://nullprogram.com/blog/2015/05/15/

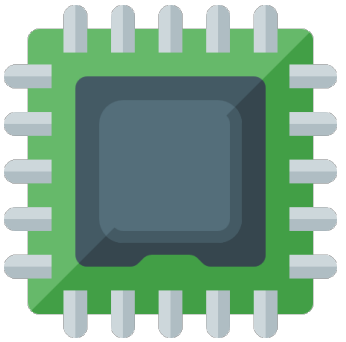
ASM (x86_64)



```
char a = -1;

short b = (short)a;

int c = (short)a;

long d = (long)a;

long e = (long)b;

long f = (long)c;
```

```
nop
movb   $0xff,-0x1(%rbp)
nop
movsbw -0x1(%rbp),%ax
mov    %ax,-0x4(%rbp)
nop
movsbl -0x1(%rbp),%eax
mov    %eax,-0x8(%rbp)
nop
movsbq -0x1(%rbp),%rax
mov    %rax,-0x10(%rbp)
nop
movswq -0x4(%rbp),%rax
mov    %rax,-0x18(%rbp)
nop
mov    -0x8(%rbp),%eax
cltq
mov    %rax,-0x20(%rbp)
nop
```

ASM (x86_64)



```
unsigned char a = 255U;

unsigned int b = (unsigned int)a;

printf("b: %d\n", b);
```

```
nop
movb   $0xff,-0x1(%rbp)
nop
movzbl -0x1(%rbp),%eax
mov    %eax,-0x8(%rbp)
nop
```

b: 255

C

- Arrays

```
int a[2] = {0x1, 0x2};

printf("a[0]: %d\n", a[0]);
printf("a[1]: %d\n", a[1]);
printf("a[-1]: %d\n", a[-1]);
printf("*(a+1): %d\n", *(a+1));
```

C

- Arrays

```
int a[2] = {0x1, 0x2};

printf("a[0]: %d\n", a[0]);
printf("a[1]: %d\n", a[1]);
printf("a[-1]: %d\n", a[-1]);
printf("*(a+1): %d\n", *(a+1));
```

```
a[0]: 1
a[1]: 2
a[-1]: 0
*(a+1): 2
```

C

- Arrays

```
int b[] = {0x1}; // ¿es posible?

int *c = b; // ¿es posible?

char *d = "abcde"; // ¿es posible?

char e[] = "abcde"; // ¿es posible?

char *f = d; // ¿es posible?

char g[] = d; // ¿es posible?
```

C

- Arrays

```
int b[] = {0x1};
```

```
int *c = b;
```

```
char *d = "abcde";
```

```
char e[] = "abcde";
```

```
char *f = d;
```

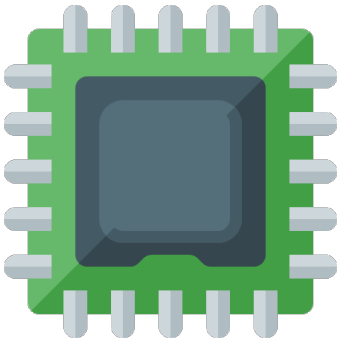
```
char g[] = d;
```

C

- ¿Cuál es la diferencia?

```
char *d = "abcde";

char e[] = "abcde";
```

ASM (x86_64)

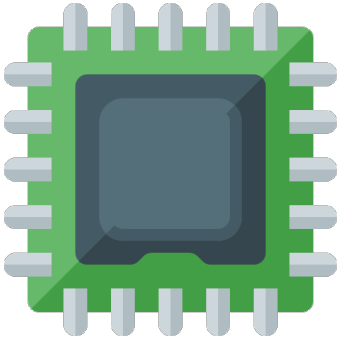


- ¿Cuál es la diferencia?

```
char *d = "abcde";

char e[] = "abcde";
```

```
90                      nop
48 c7 45 f0 20 06 40    movq   $0x400620,-0x10(%rbp)
00
90                      nop
c7 45 c0 61 62 63 64    movl   $0x64636261,-0x40(%rbp)
66 c7 45 c4 65 00       movw   $0x65,-0x3c(%rbp)
90                      nop
```

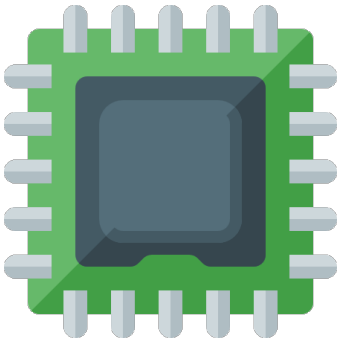
ASM (x86_64)



- Storage (strings, ints y punteros)

```
char *a = "abc\xEA\x9F\xB9";

int b = 0x01020304;

int *c = &b;
```

ASM (x86_64)



- Storage (strings, ints y punteros)

```
char *a = "abc\xEA\x9F\xB9";

int b = 0x01020304;

int *c = &b;
```

String codificado en UTF-8, terminado en NULL

```
0x4005d0:       0x61    0x62    0x63    0xea    0x9f    0xb9    0x00
(gdb) x/4xb ($rbp - 0x14)
0x7fffffffdd5c: 0x04    0x03    0x02    0x01
(gdb) x/8xb ($rbp - 0x10)
0x7fffffffdd60: 0x5c    0xdd    0xff    0xff    0xff    0x7f    0x00    0x00
```

Arquitectura little-endian: valores “invertidos” en memoria

C

- Llamada a funciones

```
struct a {
    int m1;
};

struct a v1;

f ( &v1 );

void f ( struct a *arg1 ) {
    arg1->m1 = 0;
}
```

¿Los parámetros se pasan por copia o por referencia?

C

- Llamada a funciones

```
struct a {
    int m1;
};

struct a v1;

f ( &v1 );

void f ( struct a *arg1 ) {
    arg1->m1 = 0;
}
```

¿Los parámetros se pasan por copia o por referencia?

En C, por copia únicamente

C

- Llamada a funciones

```
void f1 ( struct a arg1 );

struct a f2 ( void );

void f3 ( char arg1[] );

char[] f4 ( void );

char* f5 ( char* arg1 );
```

¿Es válido?

C

- Llamada a funciones

```
void f1 ( struct a arg1 );
```

```
struct a f2 ( void );
```

```
void f3 ( char arg1[] );
```

```
char[] f4 ( void );
```

```
char* f5 ( char* arg1 );
```

Lab

Ejercicio 0.1

- Crear programa en espacio de usuario que imprima “hola mundo” en *stdout*
  - Linkear con master *glibc*
- Debuggear función *printf* (*glibc*)
- Debuggear syscall *sys_write* (kernel)

Lab

Ejercicio 0.2

- Crear un intérprete de bytecodes (Java) en C que soporte las siguientes familias de instrucciones:
  - iconst, istore, iload, bipush, iinc, dup, iand, ixor, ior, ineg, irem, idiv, iadd, imul, isub, pop, nop, swap
- El intérprete recibe como parámetro (argv[1]) una secuencia de bytecodes en hexa
- Nombre del binario ejecutable: bytecode_interpreter
- Ejemplo: ./bytecode_interpreter 043C053D1B1C60…

Lab

Ejercicio 0.2

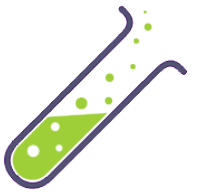
- Validar secuencias recibidas como input y retornar: -1 en caso de error, 0 en caso de éxito
  - Instrucciones válidas
  - Stack debe quedar vacío al final de la secuencia
  - No utilizar variables no inicializadas
  - Instrucciones deben tener operandos suficientes en el stack
  - Largo del stack <= 100
  - Largo de la secuencia <= 200
  - 5 variables locales como máximo
  - División por 0 no permitida
  - ¿Otras validaciones?

Lab

Ejercicio 0.2

- Printear en *stdout* bytecodes assembly cuando se compile en modo “debug” (#ifdef DEBUG). Ej:

```
0: iconst_1
1: istore_1
2: iconst_2
3: istore_2
4: iload_1
5: iload_2
6: iadd
7: istore_3
```

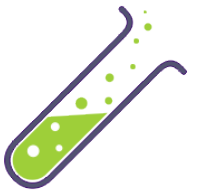


Lab

Ejercicio 0.2

- Printear en *stdout* el valor de las variables locales al final de la ejecución. Representar con el caracter “N” las variables no inicializadas. Ej:

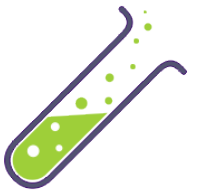
`0:150,1:90,2:12,3:9,4:N,5:N`



Lab

Ejercicio 0.2

- Crear un script con casos de test unitarios que tengan secuencias válidas e inválidas. Llamar al intérprete y verificar en *stdout*: 1) el código de retorno y, 2) variables locales
- Compartir los casos de test unitarios con otros grupos



Referencias

- Secure Coding in C and C++
  (2nd Edition, 2013) – Robert C. Seacord
- The C Programming Language
  - Dennis Ritchie & Brian Kernighan