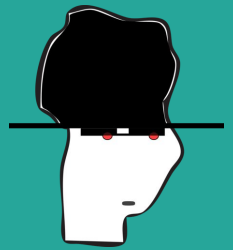
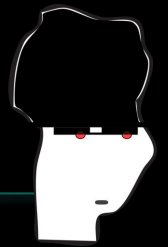


Protecciones contra Explotación

Seguridad Ofensiva

by Joshep Cortez S.
OLAPIC - FAMAF - IUA

x86: Buffer Overflow (ej1)

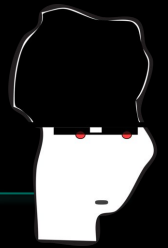


```
int main() {
   int cookie;
   char buf[80];

   gets(buf); //Lee hasta el primer ...
   if (cookie == 0x41424344)
      printf("Ganaste!\n");
}
```

x86: Format String

```
int main (void){
   int i = 1;
   char buf[N];

   fgets(buf, N, stdin);
   printf(buf);

   if (i==0x10)
      puts("YOU WIN!\n");

   return 0;
}
```

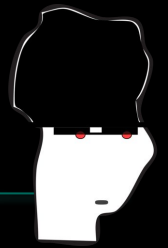
x86: Integer Overflow



```
int main(){
    int size = 0;        /* signed */
    int buf[1024];

    read(0,&size,sizeof(size));
    assert (size <= 1024);
    read(0,buf,size*sizeof(int));

    if (size > 0 && size < 100 && buf[999] == 'B')
        printf("YOU WIN!\n");

    return 0;
}
```

```
gdb-peda$ p /x 0x8000000 * 4
$3 = 0x20000000
```

x86: Control



Sobre/escribir datos:

- **variables: Suele permitir controlar parcialmente el flujo de la ejecución.**
- **RET: Suele permitir controlar EIP directamente y controlar arbitrariamente el flujo de la ejecución.**
- **GOT: Suele permitir controlar EIP directamente y controlar arbitrariamente el flujo de la ejecución.**

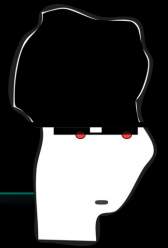
x86: Protecciones

Los sistemas operativos (kernel) y compiladores tienen distintos mecanismos para minimizar el impacto y reducir las posibles vulnerabilidades de corrupción de memoria:

- DEP / NX
- ASLR
- Stack Canaries

```
joe@zoidberg:~/Seg/Rev/bof$ ~/Soft/checksec.sh/checksec --file=01-challenge
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols       FORTIFY Fortified
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   65) Symbols     No    0
```

x86: DEP



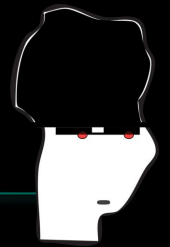
DATA EXECUTION PREVENTION

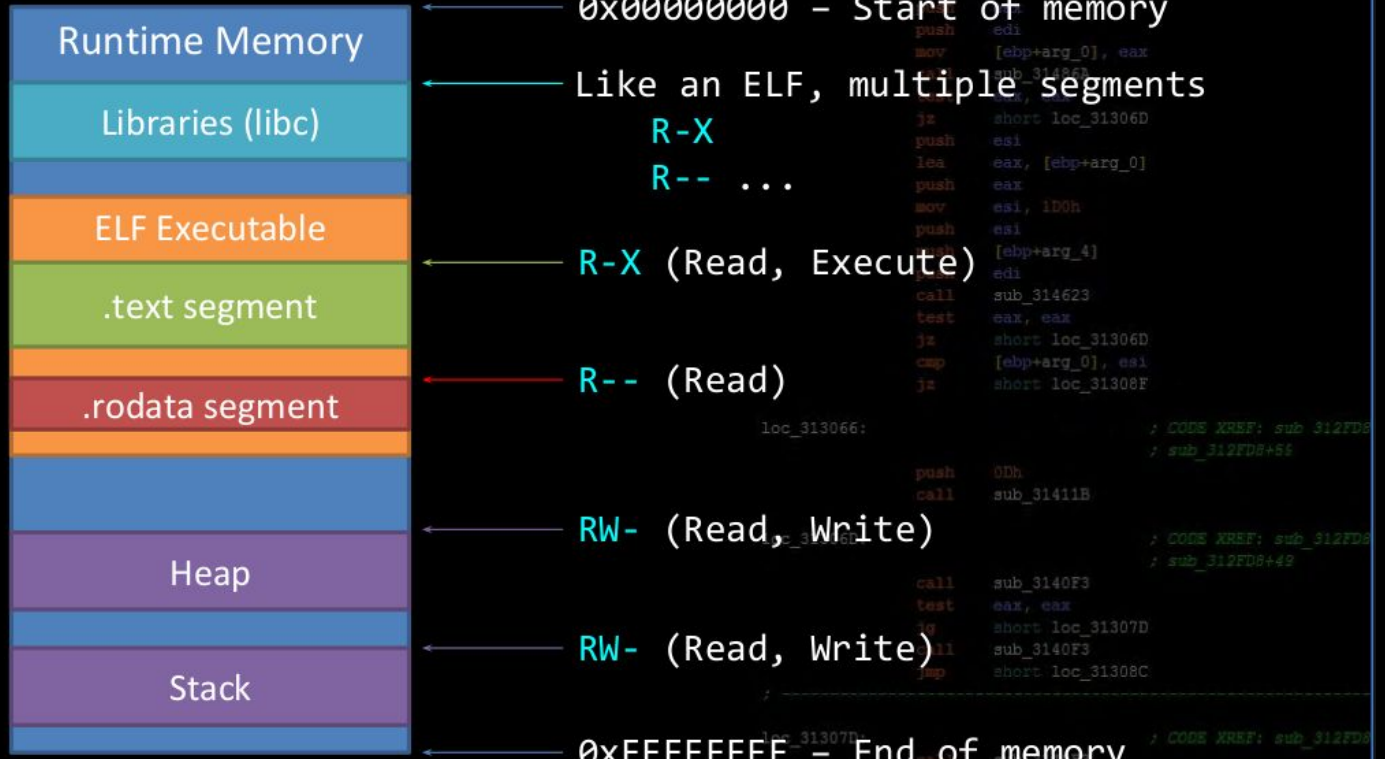
- Una técnica/protección usada para asegurar que solo un segmento de código (.text por ej) tenga permiso de ejecución.
- Intenta mitigar la inyección de código / shellcode payloads.
- Otros nombres con los que se refiere a DEP: NX, XN, XD, W^X
  - Agosto, 2004 - Linux Kernel 2.6.8
  - Agosto, 2004 - Windows XP SP2
  - Junio , 2006 - Mac OSX 10.5

x86: ~~DEP~~

https://github.com/RPISEC/MBE

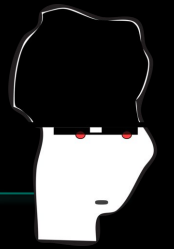
x86: DEP





https://github.com/RPISEC/MBE

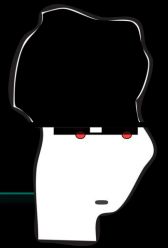
x86: Fundamentos de DEP



- Ningún segmento de memoria deberia ser nunca escribible y ejecutable al mismo tiempo. (W^X)

- Secciones de datos:
  Stack (local vars, contexto de llamadas)
  Heap (Memoria dinámica) <- alloc
  .bss (datos no inicializados)
  .ro (constantes estaticas)
  .data (datos inicializados)

x86: Fundamentos de DEP

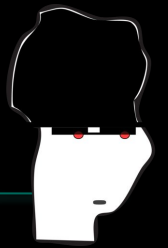


- **Secciones de código:**
  - **.text**
  - **.plt**

```
localhost challenges # cat 06-challenge.c
#include<stdlib.h>
void
win(){
    printf("YOU WIN!\n");
}

int main(int argc, char*argv[]){
    void * buf01 = malloc(1024);
    char buf02[384];

    gets(buf02);
    strcpy(buf01,buf02);

    exit(-1);
}
```

```
Disassembly of section .plt:

08048380 <gets@plt-0x10>:
 8048380:       ff 35 f8 9f 04 08       pushl  0x8049ff8
 8048386:       ff 25 fc 9f 04 08       jmp    *0x8049ffc
 804838c:       00 00                   add    %al,(%eax)
        ...

08048390 <gets@plt>:
 8048390:       ff 25 00 a0 04 08       jmp    *0x804a000
 8048396:       68 00 00 00 00          push   $0x0
 804839b:       e9 e0 ff ff ff          jmp    8048380 <_init+0x1c>

080483a0 <strcpy@plt>:
 80483a0:       ff 25 04 a0 04 08       jmp    *0x804a004
 80483a6:       68 08 00 00 00          push   $0x8
 80483ab:       e9 d0 ff ff ff          jmp    8048380 <_init+0x1c>

080483b0 <malloc@plt>:
 80483b0:       ff 25 08 a0 04 08       jmp    *0x804a008
 80483b6:       68 10 00 00 00          push   $0x10
 80483bb:       e9 c0 ff ff ff          jmp    8048380 <_init+0x1c>
```

x86: Fundamentos de DEP



- **Secciones de código:**
  - **.text**
  - **.plt**

```
localhost challenges # readelf --relocs 06-challenge

Relocation section '.rel.dyn' at offset 0x324 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff0  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x32c contains 7 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a000  00000107 R_386_JUMP_SLOT   00000000   gets
0804a004  00000207 R_386_JUMP_SLOT   00000000   strcpy
0804a008  00000307 R_386_JUMP_SLOT   00000000   malloc
0804a00c  00000407 R_386_JUMP_SLOT   00000000   puts
0804a010  00000507 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a014  00000607 R_386_JUMP_SLOT   00000000   exit
0804a018  00000707 R_386_JUMP_SLOT   00000000   __libc_start_main
```

x86: Fundamentos de DEP

- **Que ocurriria si tratara de ejecutar código en un segmento de datos marcado como no ejecutable?**

  SEGFAULT at 0xffff2a30

x86: DEP

DEP es una de las principales medidas de mitigación en tecnologías modernas que se debe bypassear para lograr explotación en programas actuales.

x86: Bypaseando DEP

DEP intenta frenar a un atacante que controla el registro EIP.

Se asume que el atacante va a intentar ejecutar código "inyectado" en una sección con permisos de RW (un shellcode).

Bypass: Si no podemos ejecutar código inyectado, tendremos que reusar código existente

x86: ROP

ROP: RETURN ORIENTED PROGRAMMING

Técnica de explotación de binarios que reusa gadgets (porciones) de código binario existente con el fin de bypassear DEP.

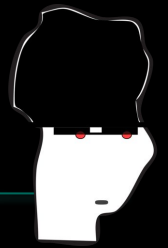
¿Gadgets?

- **Secuencia de instrucciones interesantes típicamente terminada en RET.**
- **Múltiples gadgets se encadenan juntos para lograr dar el comportamiento deseado**

x86: ROP Gadgets

```
joe@zoidberg:~/Seg/Rev/bof$ python3 ~/Soft/ROPgadget/ROPgadget.py --binary 01-challenge
Gadgets information
============================================================
0x080490da : adc al, 0x68 ; and al, al ; add al, 8 ; call eax
0x08049052 : adc al, 0xc0 ; add al, 8 ; push 0x10 ; jmp 0x8049029
0x08049042 : adc al, al ; add al, 8 ; push 8 ; jmp 0x8049029
0x08049126 : adc byte ptr [eax + 0x68], dl ; and al, al ; add al, 8 ; call edx
0x08049057 : adc byte ptr [eax], al ; add byte ptr [eax], al ; jmp 0x8049024
0x080490e4 : adc cl, cl ; ret
0x08049158 : add al, 8 ; add ecx, ecx ; ret
0x080490de : add al, 8 ; call eax
0x0804912b : add al, 8 ; call edx
0x08049034 : add al, 8 ; push 0 ; jmp 0x8049027
0x08049054 : add al, 8 ; push 0x10 ; jmp 0x8049027
0x08049044 : add al, 8 ; push 8 ; jmp 0x8049027
0x0804921f : add bl, al ; mov ebp, dword ptr [esp] ; ret
0x08049037 : add byte ptr [eax], al ; add byte ptr [eax], al ; jmp 0x8049024
0x080491b0 : add byte ptr [eax], al ; add byte ptr [eax], al ; mov ebx, dword ptr [ebp - 4] ; leave ; ret
0x0804916b : add byte ptr [eax], al ; add byte ptr [eax], al ; nop ; jmp 0x8049105
0x080490ea : add byte ptr [eax], al ; add byte ptr [eax], al ; nop ; ret
0x080490eb : add byte ptr [eax], al ; add byte ptr [esi - 0x70], ah ; ret
0x08049235 : add byte ptr [eax], al ; add esp, 8 ; pop ebx ; ret
0x08049039 : add byte ptr [eax], al ; jmp 0x8049022
0x080491b2 : add byte ptr [eax], al ; mov ebx, dword ptr [ebp - 4] ; leave ; ret
0x0804916d : add byte ptr [eax], al ; nop ; jmp 0x8049103
0x080490ec : add byte ptr [eax], al ; nop ; ret
0x08049167 : add byte ptr [ebp + 0x26b4], cl ; add byte ptr [eax], al ; nop ; jmp 0x8049109
0x08049075 : add byte ptr [ebp - 0x2ddf7d], cl ; call dword ptr [eax - 0x73]
0x080490ed : add byte ptr [esi - 0x70], ah ; ret
0x0804916a : add byte ptr es:[eax], al ; add byte ptr [eax], al ; nop ; jmp 0x8049106
0x080490e9 : add byte ptr es:[eax], al ; add byte ptr [eax], al ; nop ; ret
0x08049155 : add eax, 0x804c020 ; add ecx, ecx ; ret
0x0804915a : add ecx, ecx ; ret
0x080490e2 : add esp, 0x10 ; leave ; ret
0x08049215 : add esp, 0xc ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804901b : add esp, 8 ; pop ebx ; ret
```

x86: ROP CHAINS



"ROP CHAINS" se arman con GADGETS:

```
xor ebx, ebx
ret
```

ebx = 0
eip = [esp] ; esp = esp+4

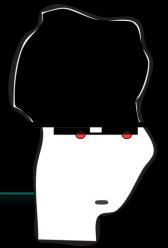
```
pop edx
pop eax
ret
```

edx = [esp] ; esp = esp+4
eax = [esp] ; esp = esp+4
eip = [esp] ; esp = esp+4

```
add eax, ebx
ret
```

eax = eax + ebx
eip = [esp] ; esp = esp+4

x86: Ejemplo ROP



Casi siempre se puede crear un lógica equivalente a un shellcode desde una cadena ROP

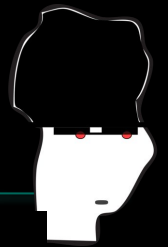
exit(0) - shellcode

```
xor     eax, eax
xor     ebx, ebx
inc     eax
int     0x80
```

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

x86: Ejemplo ROP

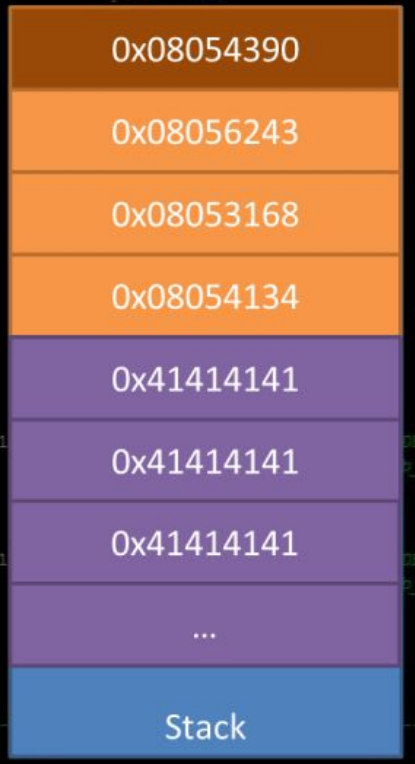


exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

EIP

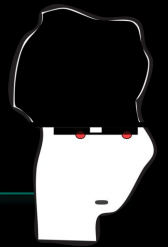
ESP



ROP chain

Stack Growth

x86: Ejemplo ROP

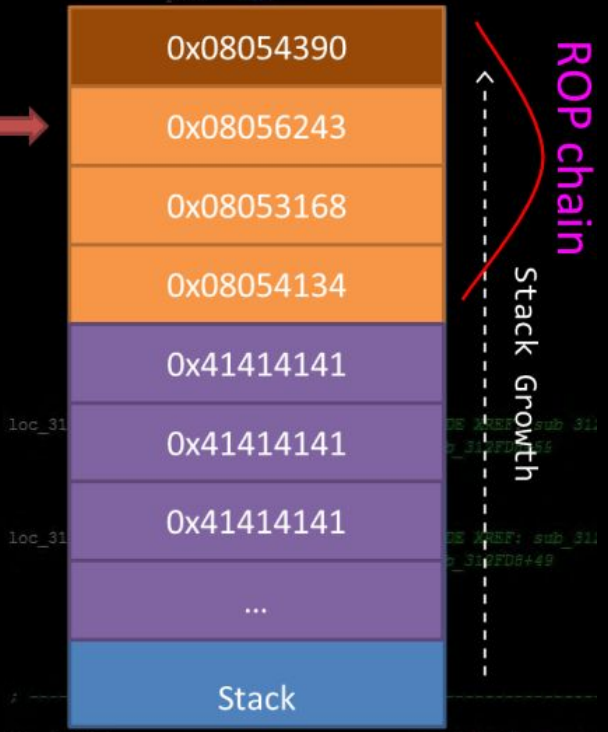


exit(0) - ROP chain

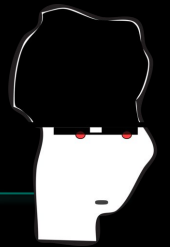
```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

ESP

EIP



x86: Ejemplo ROP

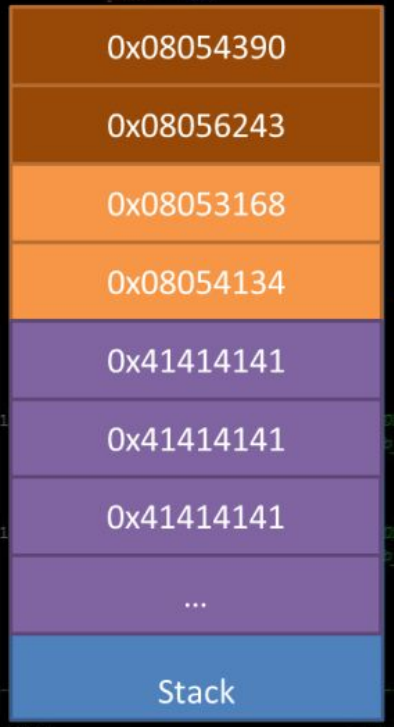


exit(0) - ROP chain

```
xor     eax, eax
ret
.....................................
xor     ebx, ebx
ret
.....................................
inc     eax
ret
.....................................
int     0x80
```

ESP

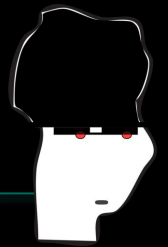
EIP



ROP chain

Stack Growth

x86: Ejemplo ROP

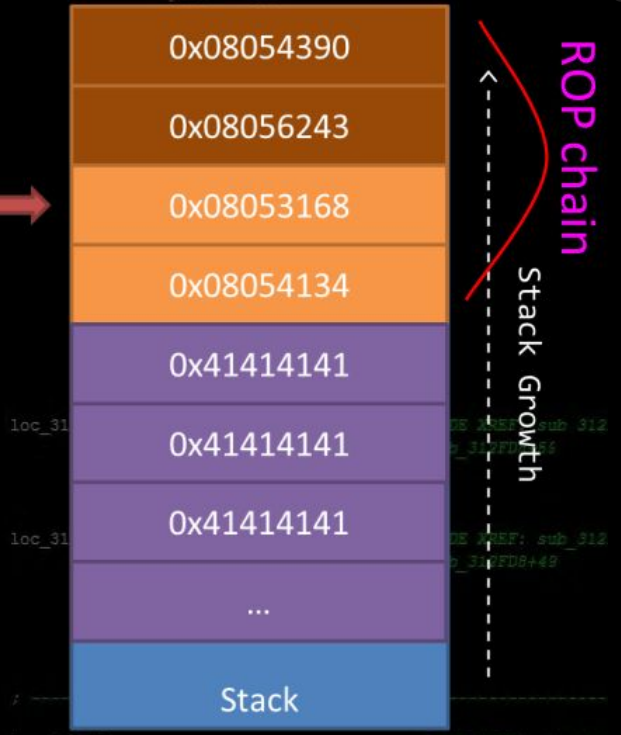


exit(0) - ROP chain

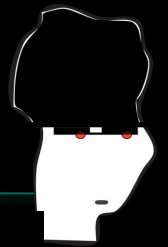
```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

ESP

EIP



x86: Ejemplo ROP



exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

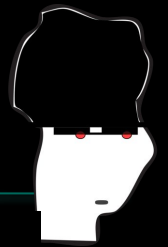
EIP

ESP

ROP chain

Stack Growth

x86: Ejemplo ROP

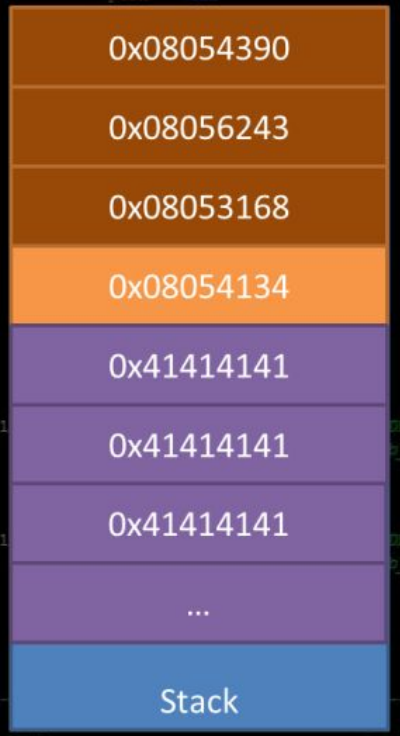


exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```

ESP

EIP



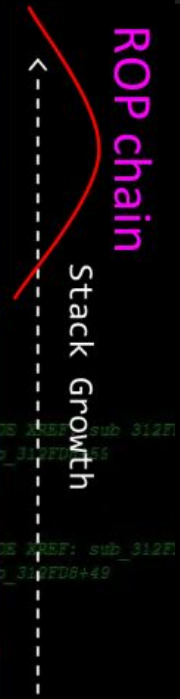
ROP chain

Stack Growth

x86: Ejemplo ROP

exit(0) - ROP chain

```
xor     eax, eax
ret
xor     ebx, ebx
ret
inc     eax
ret
int     0x80
```



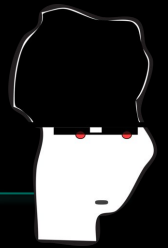
x86: Conclusion ROP

a) Alcanzamos la ejecución de codigo deseado (una especie de shellcode para llamar a *exit*) sin inyectar código.

b) Estas cadenas de ROP pueden ser costosas de encontrar y a veces hay que ser creativos para conseguir gadgets de interés.

c) A veces hace falta hacer stack pivoting

x86: Ret2libc

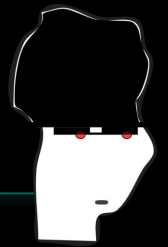


Es un "caso particular" de ROP en el que se salta a direcciones de funciones de la libc (si existen) en lugar de usar gadgets.

Requiere conocer las direcciones a las cuales se puede saltar.

Si es aplicable, resulta más fácil que ROP

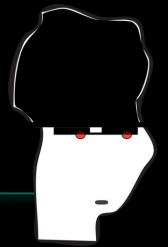
x86: Ret2libc



Funciones tipicas:

- system()
  - ejecuta comandos.
  - system("cat /etc/passwd")
- (f) open(), read(), write()
  - operaciones sobre archivos

x86: Ret2libc



```
0x08045430:  ret        <-- EIP
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

system() --->

system() --->

ret address --->

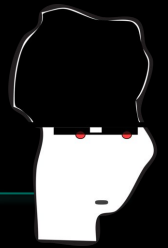
first arg --->
"cat flag.txt"

```
0x08045430:  ret
```

EIP

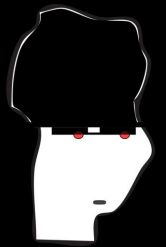
```
.....................................
system()
0xb7e65190:  push   ebx
0xb7e65191:  sub    esp, 8
0xb7e65194:  mov    eax, DWORD PTR
  [esp+0x10]
...
```

x86: Conclusion ROP



a) Alcanzamos la ejecución de codigo deseado (una especie de shellcode para llamar a *exit*) sin inyectar código.

b) Estas cadenas de ROP pueden ser costosas de encontrar y a veces hay que ser creativos para conseguir gadgets de interés.



“Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations”

-Dino Dai Zovi