ROP Dance

Mi Baile en tu RAM

Contexto

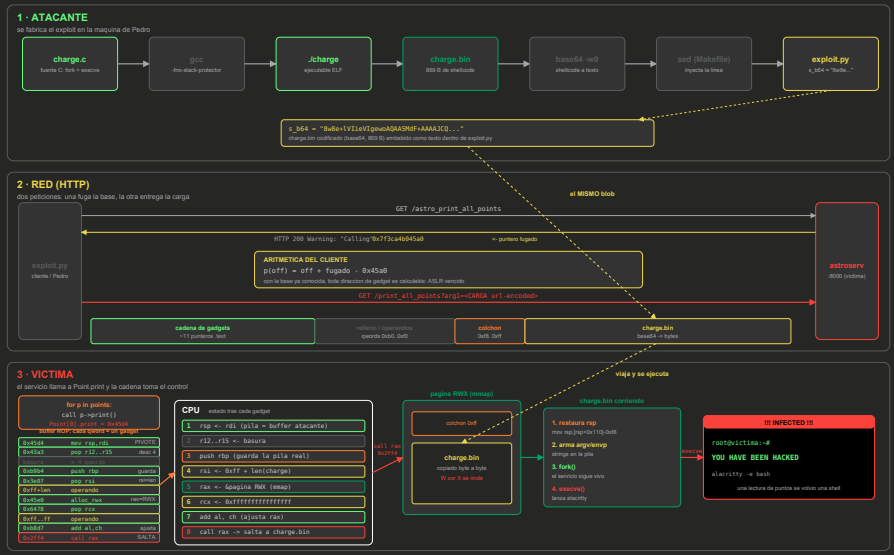
ROP Dance

Charla independiente de una hora.

- Sobre la Programación orientada al retorno una tecnica de ciberseguridad ofensiva.
- Destinada a informáticos avanzados: alumnos del DUOC Viña del Mar y sus amigos.
- Presentada por un ciberatacante legal de la empresa SEK.

Contenido

Resumen del baile

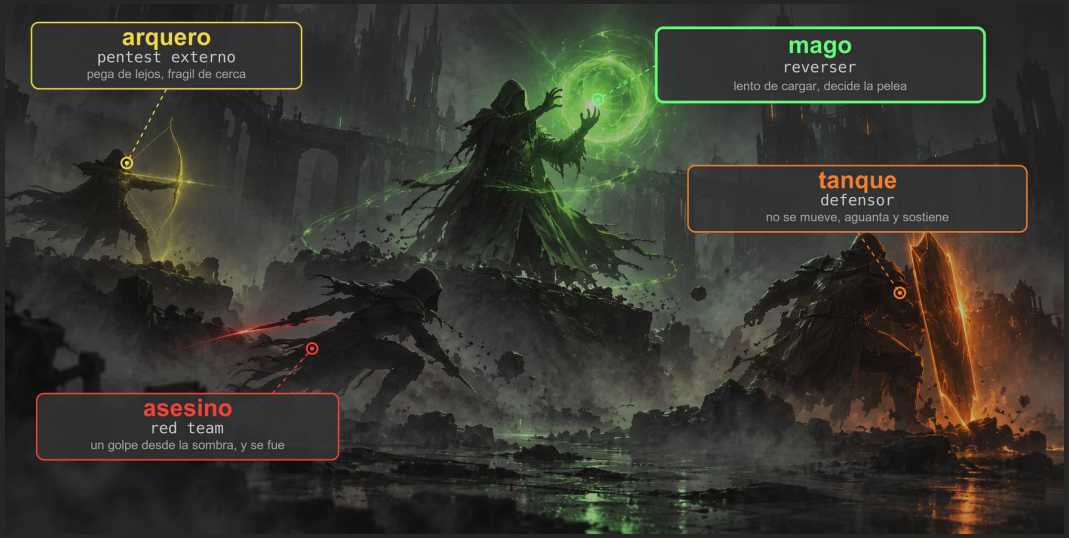


Ingeniería inversa

Ingeniería inversa de código

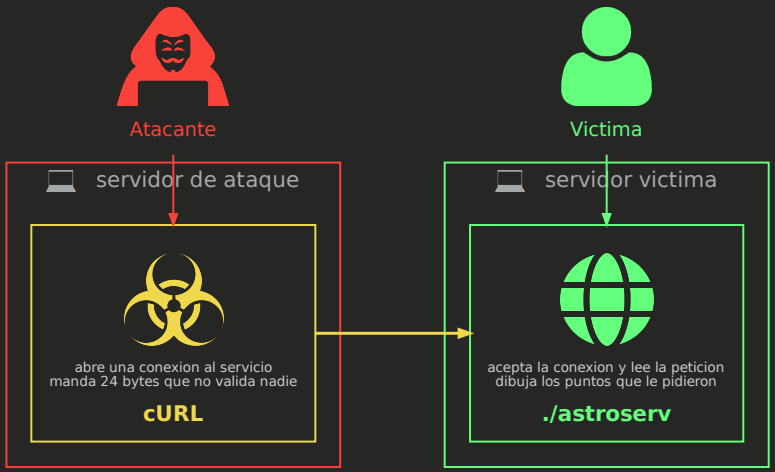
Todo es código abierto si sabes ensamblador.

El reverser

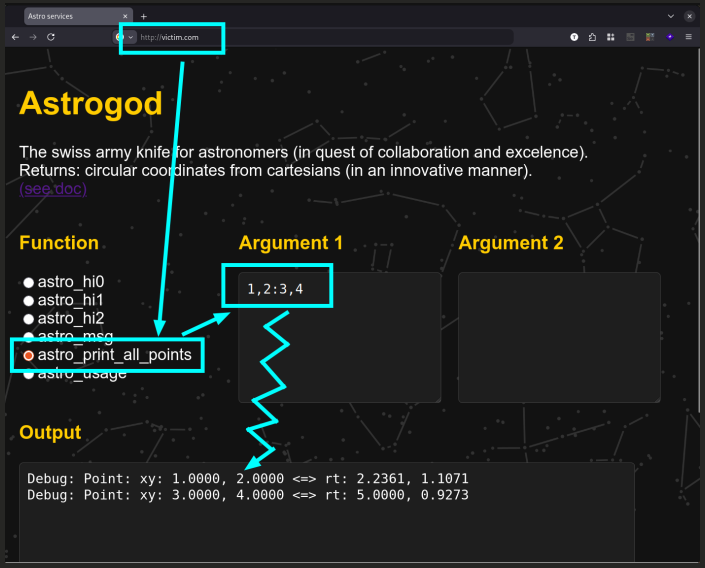

arquero
pentest externo
pega de lejos, fragil de cerca
mago
reverser
lento de cargar, decide la pelea
tanque
defensor
no se mueve, aguanta y sostiene
asesino
red team
un golpe desde la sombra, y se fue

2/4 Escenario

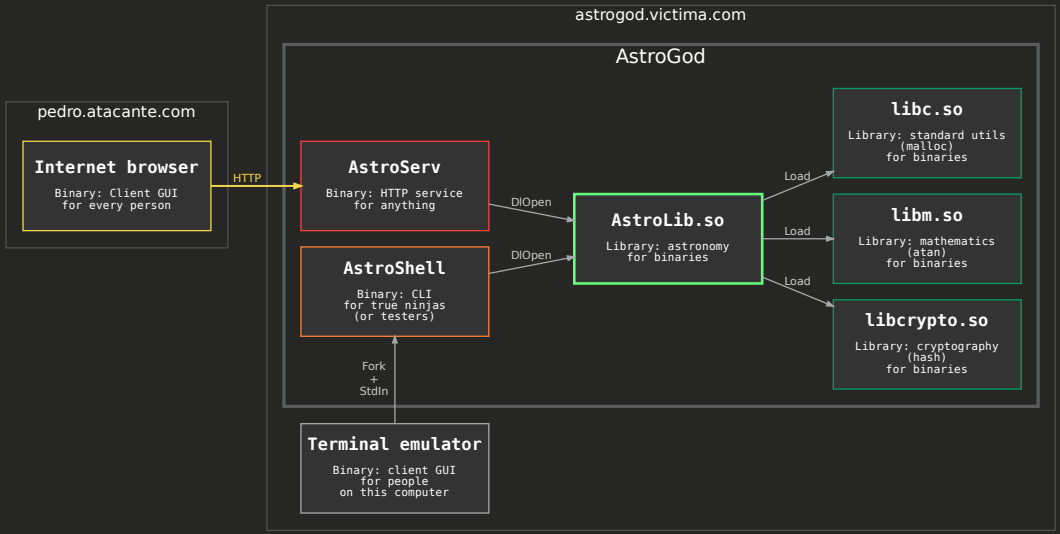
Diagrama del ataque



Applicación HTTP vulnerable



Arquitectura

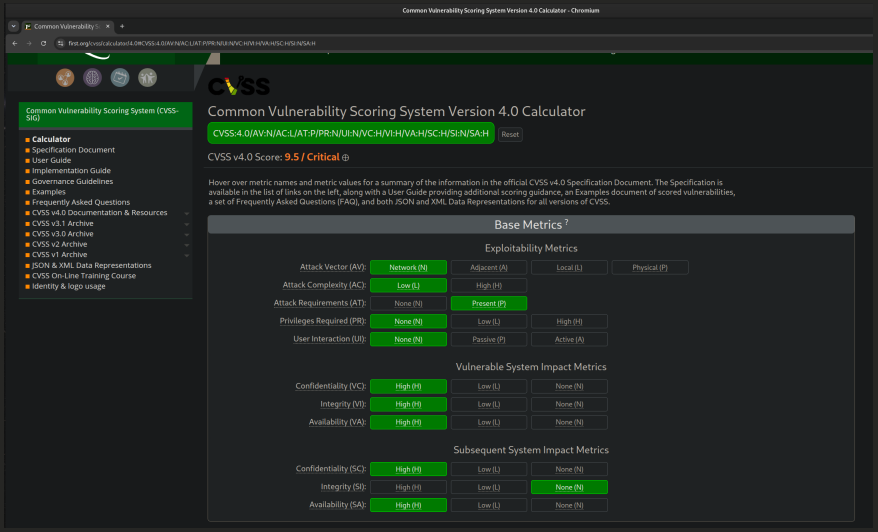


La vulnerabilidad

¿Cuántas formas hay de romper un programa?

De las mil combinaciones de un puntaje a la única línea de código que falla.

Vulnerabilidad: CVSS: 1296 impactos



Vulnerabilidad: Lista Pentest: 40+ nombres

Binary: Memory
Buffer Overflow
Stack Buffer Overflow
Heap Overflow
Integer Overflow
Off-by-One Error
Out-of-Bounds Read
Out-of-Bounds Write
Null Pointer Deref
Uninitialized Variable
Memory Corruption

Binary: Exploit
Use-After-Free
Double Free
Dangling Pointer
Type Confusion
Format String
Return-Oriented Prog.
Return-to-libc
Shellcode Injection
Arbitrary Code Exec.
Race Condition

Web: Injection
SQL Injection
Cross-Site Scripting
Command Injection
Path Traversal
SSRF
XXE Injection
LDAP Injection
Template Injection
Open Redirect
Insecure Deserial.

Access & Logic
Cross-Site Req. Forgery
Broken Access Control
Authentication Bypass
Privilege Escalation
IDOR
Session Hijacking
Clickjacking
Business Logic Flaw
Information Disclosure
Hardcoded Credentials

Vulnerabilidad: RWX: 3 poderes

Cuarenta clases de vulnerabilidad, **tres** consecuencias:

R

Leer
Espionaje

W

Escribir
Sabotaje

X

Ejecutar
Control total

Vulnerabilidad: La última verdad: 1 definición

Un **ciberataque** se realiza mediante la **explotación** de una **vulnerabilidad informática**.

Vulnerabilidad: La última verdad: 1 definición

Un **ciberataque** se realiza mediante la **explotación** de una **vulnerabilidad informática**.

Una **vulnerabilidad informática** es un aspecto cuya explotación permite realizar un **ciberataque**,

Vulnerabilidad: La última verdad: 1 definición

Un **ciberataque** se realiza mediante la **explotación** de una **vulnerabilidad informática**.

Una **vulnerabilidad informática** es un aspecto cuya explotación permite realizar un **ciberataque**, es decir un efecto malicioso que los usuarios de un sistema no habían contemplado.

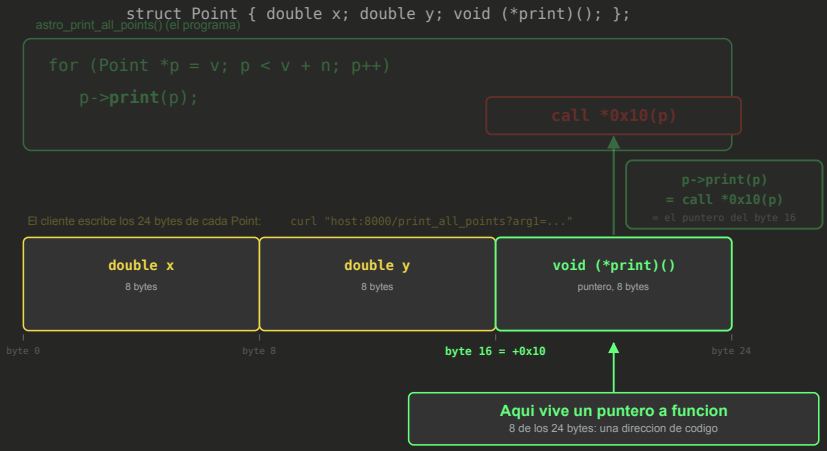
Vulnerabilidad: La última verdad: 1 definición

Un **ciberataque** se realiza mediante la **explotación** de una **vulnerabilidad informática**.

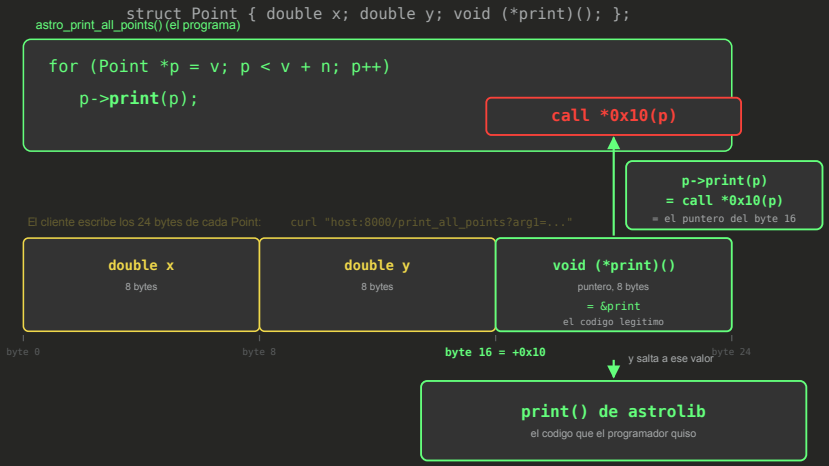
Una **vulnerabilidad informática** es un aspecto cuya explotación permite realizar un **ciberataque**, es decir un efecto malicioso que los usuarios de un sistema no habían contemplado.

Un ciberataque es lo que se realizó mediante computador (ciber) y duele (ataque).

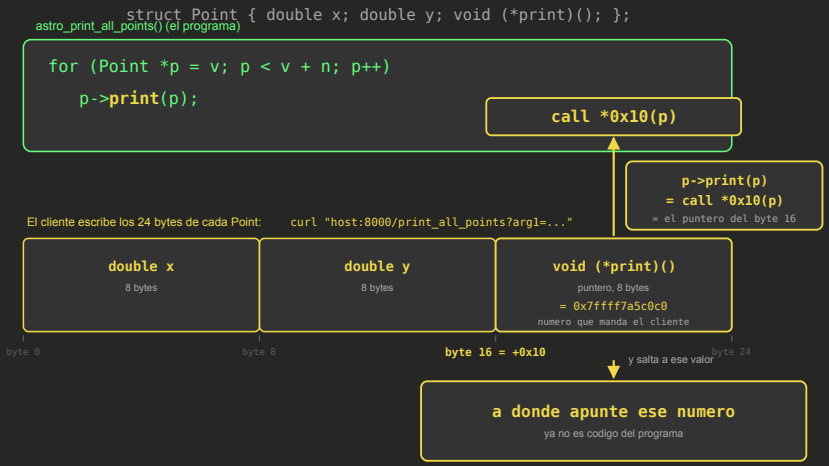
Confusión de tipo: un Point son 24 bytes



Confusión de tipo: el bucle llama al puntero

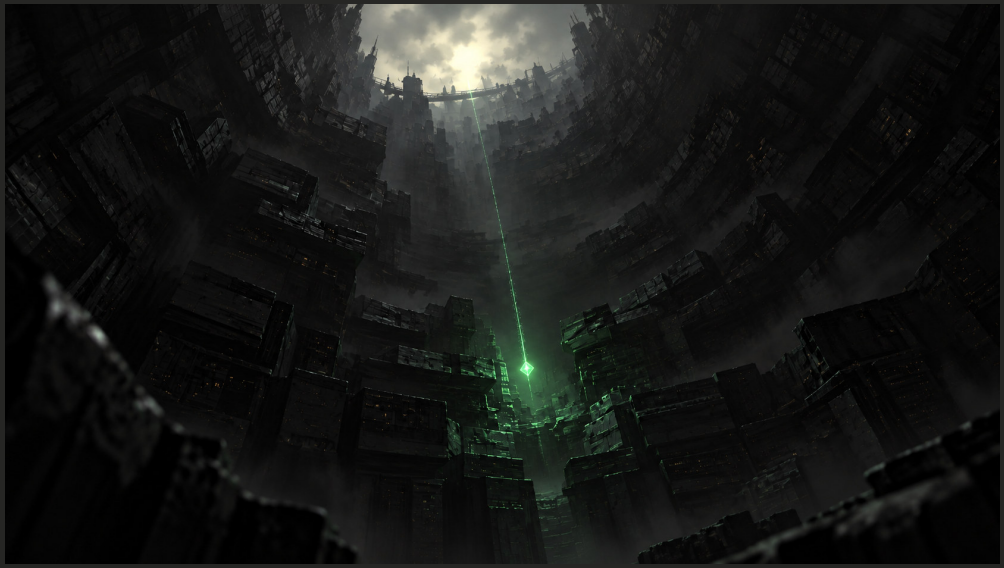


Confusión de tipo: esos bytes vienen de la red

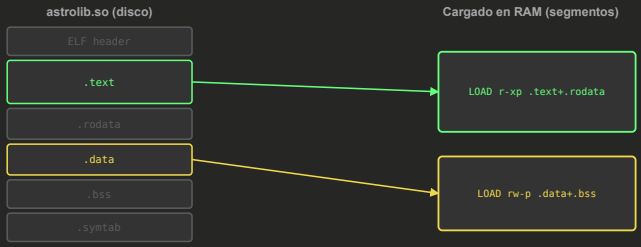


Confusión de tipo: call anywhere

3/4 Inframundo



Proceso: El cargador

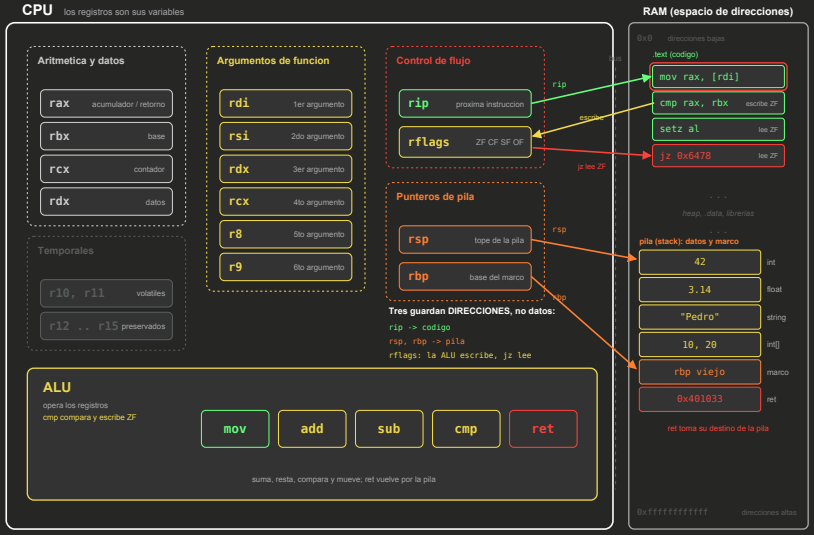


.symtab y .comment no se cargan: solo viven en disco.

Memoria

```
$ cat /proc/1420198/maps
5e4975cf4000-5e4975cf6000 r--p  ~/astroserv
5e4975cf6000-5e4975d06000 r-xp  ~/astroserv
5e4975d06000-5e4975d19000 r--p  ~/astroserv
5e4975d19000-5e4975d1a000 r--p  ~/astroserv
5e4975d1a000-5e4975d1b000 rw-p  ~/astroserv
5e4975d1b000-5e4975d1c000 rw-p
5e49786cd000-5e49786ee000 rw-p  [heap]
70716f000000-70716f0b3000 r--p  .../libcrypto.so.3
70716f0b3000-70716f3e7000 r-xp  .../libcrypto.so.3
70716f3e7000-70716f4b2000 r--p  .../libcrypto.so.3
70716f4b2000-70716f50e000 r--p  .../libcrypto.so.3
70716f50e000-70716f511000 rw-p  .../libcrypto.so.3
70716f511000-70716f514000 rw-p
70716f517000-70716f527000 r--p  .../libm.so.6
70716f527000-70716f5a6000 r-xp  .../libm.so.6
70716f5a6000-70716f5fe000 r--p  .../libm.so.6
70716f5fe000-70716f5ff000 r--p  .../libm.so.6
70716f5ff000-70716f600000 rw-p  .../libm.so.6
70716f600000-70716f628000 r--p  .../libc.so.6
70716f628000-70716f7b1000 r-xp  .../libc.so.6
70716f7b1000-70716f800000 r--p  .../libc.so.6
70716f800000-70716f804000 r--p  .../libc.so.6
70716f804000-70716f806000 rw-p  .../libc.so.6
70716f806000-70716f813000 rw-p
70716f88d000-70716f890000 rw-p
70716f89e000-70716f8a1000 r--p  ~/astrolib.so
70716f8a1000-70716f8ae000 r-xp  ~/astrolib.so
70716f8ae000-70716f8b2000 r--p  ~/astrolib.so
70716f8b2000-70716f8b3000 r--p  ~/astrolib.so
70716f8b3000-70716f8b4000 rw-p  ~/astrolib.so
70716f8b4000-70716f8b7000 rw-p
70716f8b7000-70716f8b8000 r--p  .../ld-linux-x86-64.so.2
70716f8b8000-70716f8e3000 r-xp  .../ld-linux-x86-64.so.2
70716f8e3000-70716f8ed000 r--p  .../ld-linux-x86-64.so.2
70716f8ed000-70716f8ef000 r--p  .../ld-linux-x86-64.so.2
70716f8ef000-70716f8f1000 rw-p  .../ld-linux-x86-64.so.2
7ffd08ab6000-7ffd08ad9000 rw-p  [stack]
7ffd08b71000-7ffd08b75000 r--p  [vvar]
7ffd08b75000-7ffd08b77000 r-xp  [vdso]
ffffffffff600000-ffffffffff601000 --xp
```

CPU: Registros



ASM

x86-64 POR LO QUE HACE

mover y calcular, operar, comparar y saltar: el hilo son las BANDERAS. Sintaxis Intel (destino, origen).

ROP Dance
Mi baile en tu RAM

1 MOVER Y CALCULAR

mov copia lo que hay en la dirección; lea entrega la dirección misma

La escalera del operando de memoria: cada peldaño agrega una pieza. Formato completo [base + indice*escala + desplazamiento]

1. inmediato	`mov rax, 5`	rax <- 5 el valor, sin tocar memoria
2. registro	`mov rax, rbx`	rax <- rbx
3. directo	`mov rax, [0x404018]`	rax <- Mem[0x404018]
4. indirecto	`mov rax, [rbx]`	rax <- Mem[rbx] dereferencia
5. + desplazamiento	`mov rax, [rbx+16]`	rax <- Mem[rbx+16] campo de un struct
6. indexado escalado	`mov rax, [rbx+rcx*8+16]`	arr[i] de 8 bytes

lea load effective address

`lea rax, [rbx+rcx*8+16]`

rax <- rbx+rcx*8+16 la dirección, NO su contenido

Calcula en la unidad de direcciones (AGU), no en la ALU: por eso NO toca banderas.

ninguna: ademas sirve de suma y multiplicacion gratis

2 ARITMETICA Y LOGICA

cada tarjeta muestra las banderas que deja (leyenda abajo)

ARITMETICA

`add rax, rbx` suma: rax += rbx — CF ZF SF OF PF AF

`sub rax, rbx` resta: rax -= rbx — CF ZF SF OF PF AF

`imul rax, rbx` multiplica con signo — CF SF

`inc rax` no toca CF — rax += 1 — ZF SF OF PF AF

`dec rax` no toca CF — rax -= 1 — ZF SF OF PF AF

`neg rax` rax = -rax (dos comp.) — CF ZF SF OF PF AF

LOGICA DE BITS

`and rax, rbx` AND bit a bit — 0 ZF SF 0 PF

`or rax, rbx` OR bit a bit — 0 ZF SF 0 PF

`xor rax, rax` pone rax en 0 (clásico) — 0 ZF SF 0 PF

`not rax` no toca banderas — invierte los bits

DESPLAZAR BITS

`shl rax, 4` x16; mete ceros abajo — CF ZF SF PF

`shr rax, 4` divide /16 sin signo — CF ZF SF PF

`sar rax, 4` divide /16 con signo — CF ZF SF PF

La línea de banderas: ZF letra encendida = se fija · 0 se fuerza a 0 · queda indefinida · punto = sin tocar

CF ZF SF OF : las cuatro que deciden saltos | PF AF : las dos que casi nunca cerrificen

3 COMPARAR: cmp y test

no guardan el resultado: existen solo para dejar banderas

`cmp a, b` hace a - b y TIRA el resultado — `cmp rax,rbx = 48 39 d8`

`cmp rax, rbx` = sub rax, rbx pero sin escribir rax — ej. cmp rax, 5

Deja las seis banderas, igual que sub. El salto que sigue las lee. — ZF=1 sii rax vale 5

CF ZF SF OF PF AF — deja estas banderas

`test a, b` hace a AND b y TIRA el resultado — `test rax,rax = 48 85 c0`

`test rax, rax` ZF=1 sii rax==0: el chequeo de cero — ej. test al, al

Limpia CF y OF a 0; deja ZF SF PF. El clásico para "es cero?". — ZF=1 sii al == 0

0 ZF SF 0 PF — deja estas banderas

las instrucciones de arriba ESCRIBEN estas banderas

4 BUS DE BANDERAS

CF	ZF	SF	OF	PF	AF
acarreo / sin signo	resultado cero	resultado negativo	desborde / con signo	paridad	ajuste BCD

los saltos de abajo las LEEN

5 SALTOS CONDICIONALES

el mnemonico depende de si el numero es con signo o sin signo

CON SIGNO leen SF, OF y ZF (l / g = less / greater)

`jl / jnge`	7c	`jle / jng`	7e
SF != OF	menor	ZF=1 o SF!=OF	menor o igual
`jg / jnle`	7f	`jge / jnl`	7d
ZF=0 y SF=OF	mayor	SF = OF	mayor o igual

SIN SIGNO leen CF y ZF (b / a = below / above)

`jb / jc`	72	`jbe / jna`	76
CF = 1	debajo	CF=1 o ZF=1	debajo o igual
`ja / jnbe`	77	`jae / jnc`	73
CF=0 y ZF=0	encima	CF = 0	encima o igual

COMUNES (leen ZF): `je / jz` ZF=1 igual 74 · `jne / jnz` ZF=0 distinto 75 — los con signo ocupan el bloque de opcodes contiguo 7c..7f

EL PAR QUE MAS SE CONFUNDE el mismo byte, distinto salto: por eso jl y jb NO son intercambiables

`mov al, 0xFF` `cmp al, 1` el byte 0xFF vale -1 con signo, 255 sin signo

CON SIGNO: -1 < 1 -> jl SALTA
lee SF != OF. Verificado: jl tomado.

SIN SIGNO: 255 > 1 -> ja SALTA
lee CF=0 y ZF=0. Verificado: jb NO tomado, ja tomado.

La regla: si el numero puede ser negativo se usan **j / jg** (con signo); si es un tamano, un indice o un puntero se usan **jb / ja** (sin signo).

Convertir C a NASM

```c
// Step 1: Write hello.c
#include <stdio.h>
int main(){
  printf( "Hello World \n" );
  return 0;
}
```

```sh
# Step 2: Create the object file
gcc -fno-asynchronous-unwind-tables \
  -s -c -o hello.o hello.c

# Step 3: Disassemble the object file
objconv -fnasm hello.o

# Step 4: Assemble
nasm -f elf hello.asm
gcc hello.o -o hello
./hello    # out: hello world
```

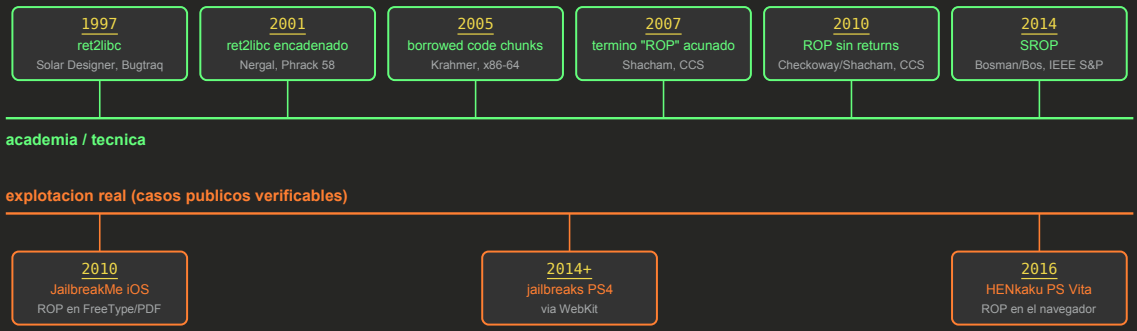
```
; Disassembly of file: hello.o
SECTION .text   align=4 execute

main:   ; Function begin
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0H
    sub     esp, 16
    mov     dword [esp], ?_001
    call    puts
    mov     eax, 0
    leave
    ret

SECTION .rodata align=1 noexecute

?_001:
    ; 0000 _ Hello Wo
    db 48H, 65H, 6CH, 6CH, 6FH,
     ↪  20H, 57H, 6FH
    ; 0008 _ rld .
    db 72H, 6CH, 64H, 20H, 00H
```

ROP: Historia



Pila

LA PILA: memoria, marco y gadgets

x86-64 - sintaxis Intel (destino, origen)
la pila crece hacia abajo: en el dibujo, altas arriba

1 LA PILA EN MEMORIA: una pila de marcos

La pila es una PILA de marcos de funcion.
Direcciones ALTAS arriba, como en un calcado de gdb.

Cada call apila un marco nuevo hacia abajo;
al volver, ret lo descarta y rsp sube.
El heap crece al reves, desde direcciones bajas hacia arriba

2 PUSH y POP: la pila se mueve sola

```
push rax   rsp -= 8 ; [rsp] = rax        (baja 8)   50
pop rax    rax = [rsp] ; rsp += 8        (sube 8)   58
```

cada celda es un qword: 8 bytes, una direccion.

La pila crece hacia ABAJO, a direcciones mas bajas.
En el dibujo las altas van arriba, como un calcado de gdb.

3 CALL y RET: push y pop del rip

```
call f   = push rip ; jmp f              (baja 8)   e8 / ff d0
ret      = pop rip                       (sube 8)   c3
```

la dir de retorno es la instruccion que sigue al call, no vuelve ahi.

ret confia ciegamente en la pila.
quien escribe la pila, escribe el rip.

4 EL MARCO DE PILA: prologo y epilogo

PROLOGO

```
push rbp          55        0x7fffffffe828
                            0x7fffffffe820
mov rbp, rsp      48 89 e5  0x7fffffffe818
                            0x7fffffffe810
sub rsp, N        48 83 ec N
```

rbp = base actual del marco
reserva N bytes de locales

EPILOGO

```
leave   = mov rsp, rbp ; pop rbp      c9
ret     = pop rip                     c3
```

deshace el marco de un golpe
vuelve a quien llamo

La pila queda alineada a 16 bytes justo antes de cada call.
Red zone: 128 bytes bajo rsp usables sin reservar (no en Windows).

el rbp guardado encadena con el marco anterior

EL BLANCO DE ROP: un desbordamiento de un local pisa la dir de retorno.
ret salta a donde diga la pila, no a donde el programa quiso volver.

5 EL REMATE: el catalogo de gadgets, y por que todo acaba en ret

CARGA (pop: valor de la pila -> registro)

```
pop rdi ; ret          rdi = [rsp]; rsp += 8
5f c3
pop rsi ; ret          rsi = [rsp]; rsp += 8
5e c3
pop rdx ; ret          rdx = [rsp]; rsp += 8
5a c3
pop rax ; ret          rax = [rsp]; rsp += 8
58 c3
mov [rdi], rax ; ret   Memory[rdi] = rax
48 89 07 c3
```

ARITMETICA

```
add rax, rbx ; ret     rax += rbx
48 01 d8 c3
xor rax, rax ; ret     rax = 0
48 31 c0 c3
```

PIVOTE DE PILA

```
xchg rsp, rax ; ret    rsp <-> rax: pivota la pila
48 94 c3
leave ; ret            rsp = rbp; pop rbp
c9 c3
```

CONTROL DE FLUJO

```
call rax               salta a la direccion en rax
ff d0
jmp rax                salta a la direccion en rax
ff e0
```

KERNEL

```
syscall ; ret          entra al kernel
0f 05 c3
```

syscall entra al kernel; rax elige cual. execve = 59.
Que registros lleva cada arg va en el poster de REGISTROS.

EL GADGET QUE NADIE ESCRIBO

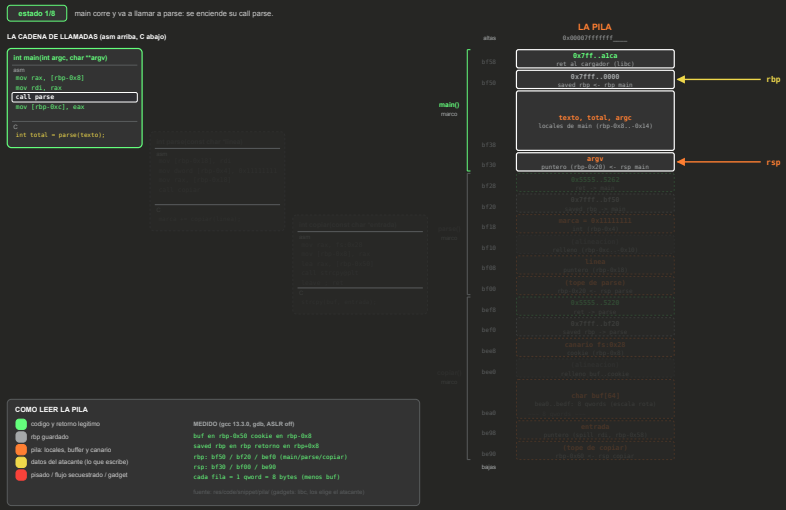
```
mov eax, 0xc35f   =   b8 5f c3 00 00
```

`leido un byte mas adelante:` `5f c3` `=` `pop rdi ; ret`

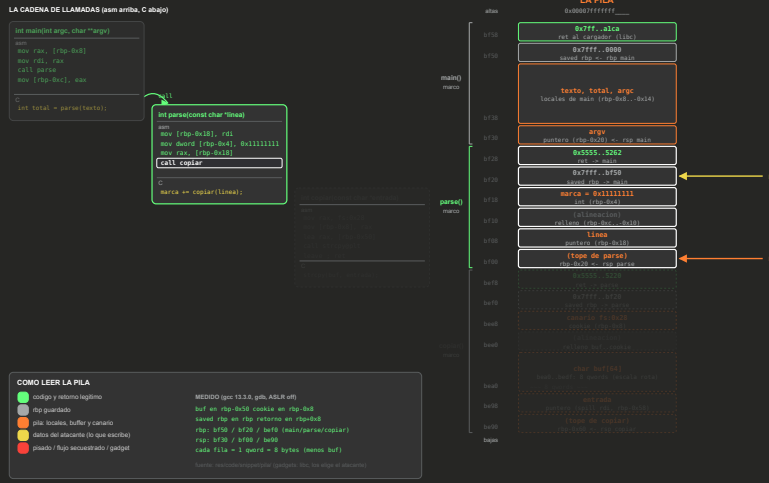
Todo byte cf en el texto es un posible final de gadget: el compilador no lo pone, le lectura desalineada lo saca.

ret = pop rip: cada gadget devuelve el control a la pila.
Encadenar mas convierte la pila en el programa.

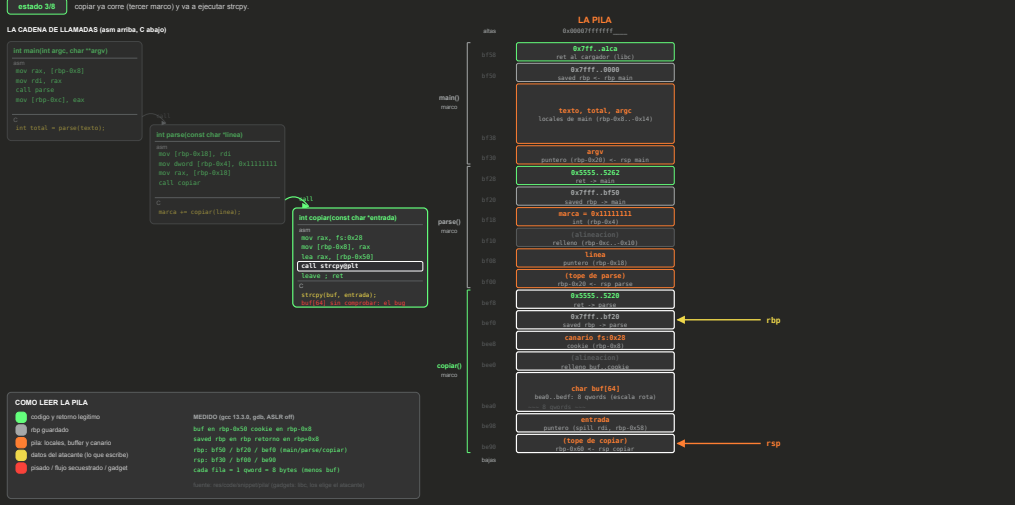
La pila 1/8: solo el marco de main



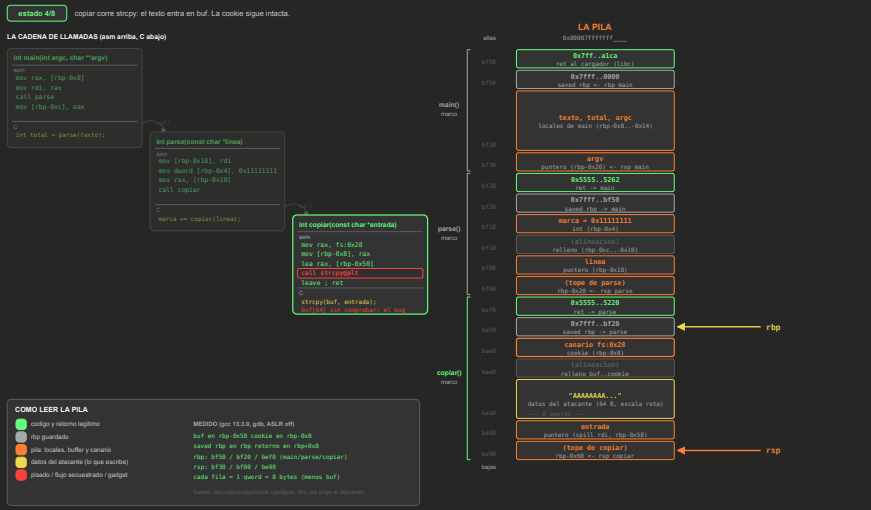
La pila 2/8: parse corre, un escalón más arriba



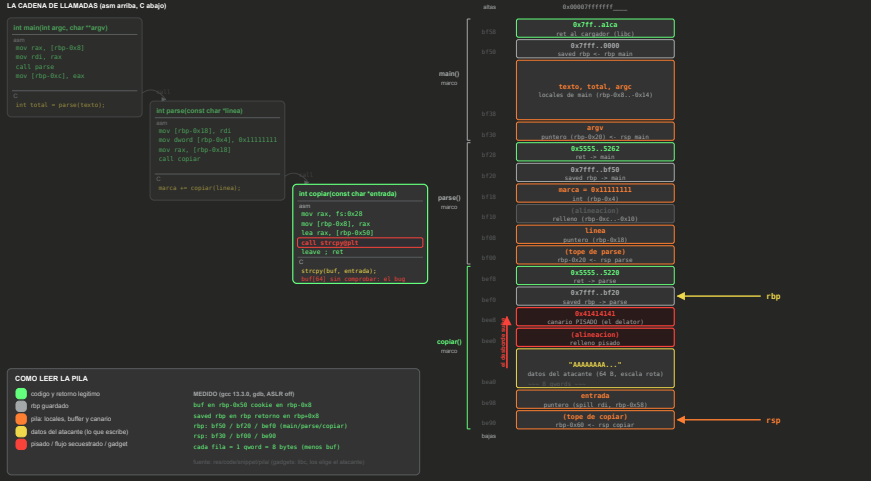
La pila 3/8: copiar corre, con buf[64] sin comprobar



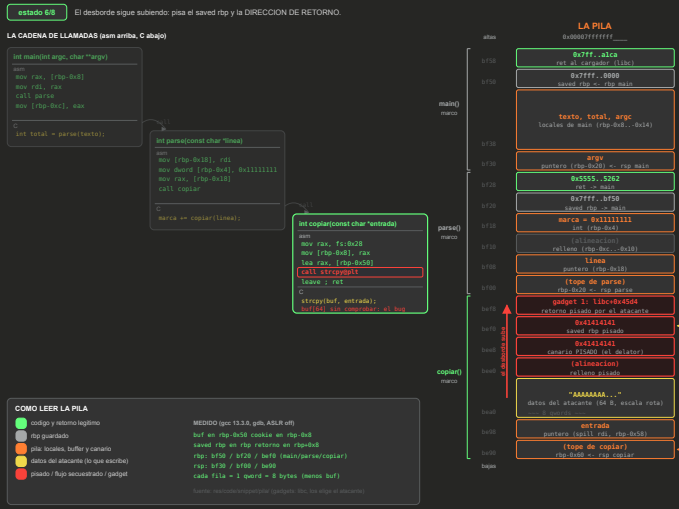
La pila 4/8: mientras cabe, no pasa nada



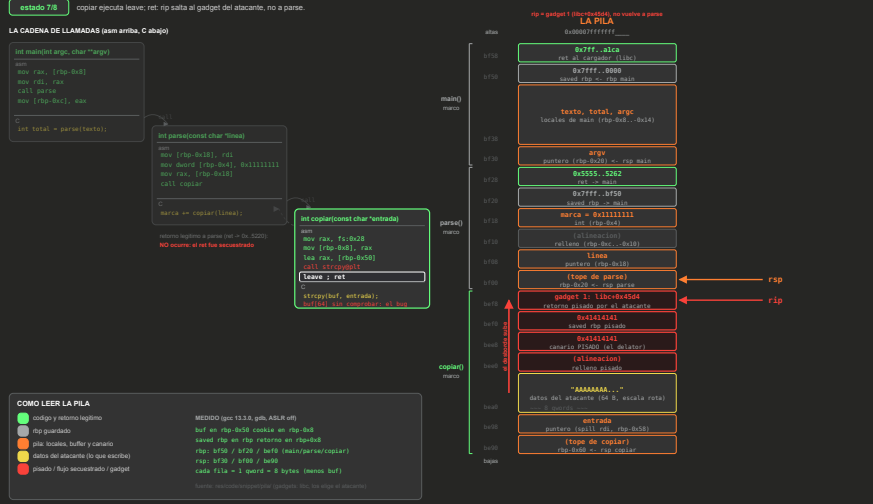
La pila 5/8: pasa de 64 bytes y pisa la cookie



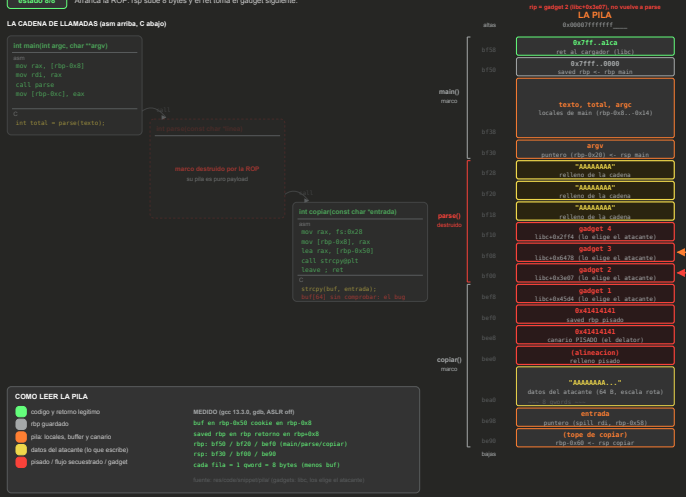
La pila 6/8: el desborde alcanza el retorno



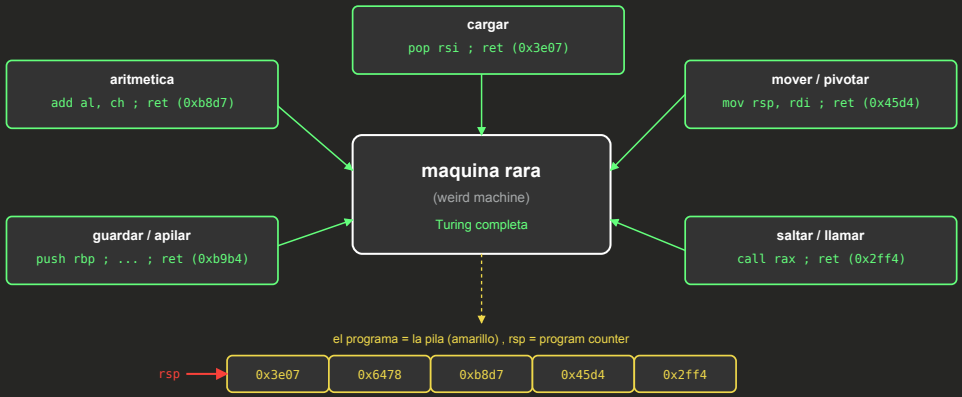
La pila 7/8: leave; ret mete la dirección pisada en rip



La pila 8/8: sigue la cadena, gadget a gadget

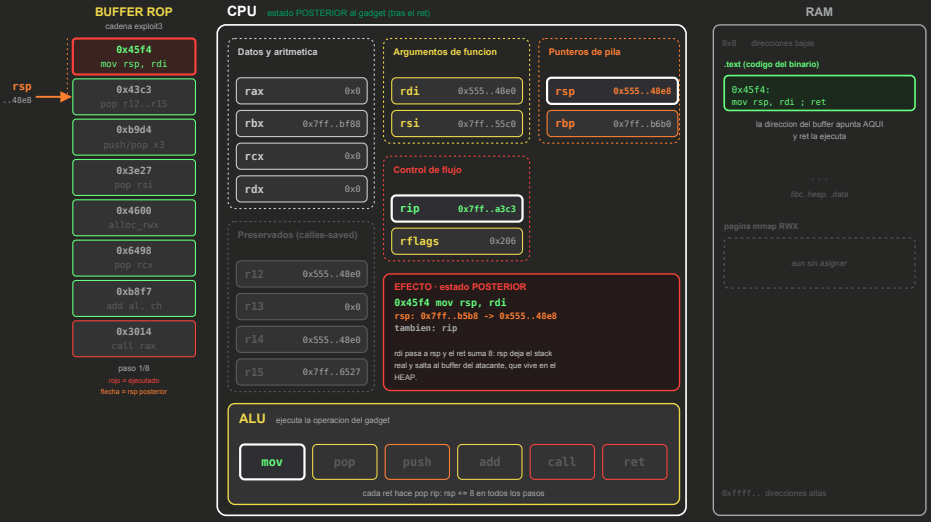


La coreografia, en una maquina rara

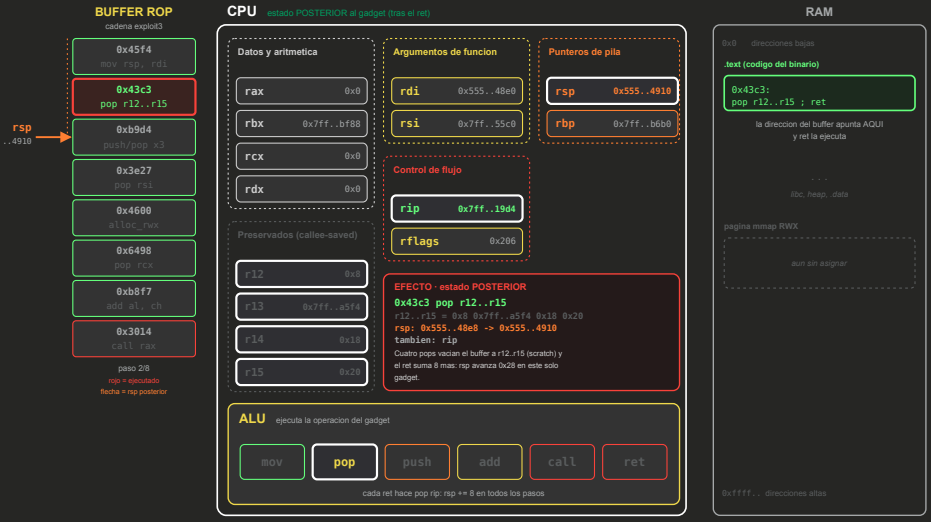


con un salto condicional (aritmetica + carry, como add al, ch) se cierra la completitud

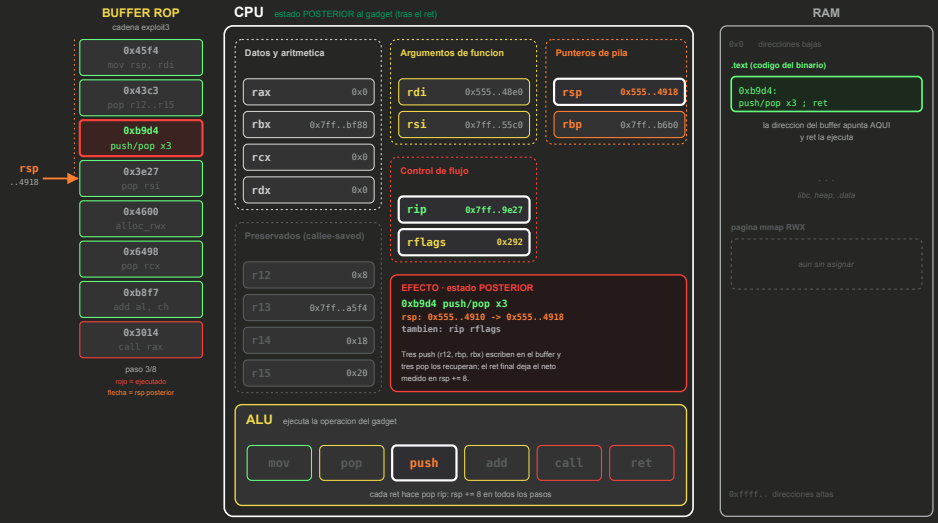
La coreografía, paso a paso



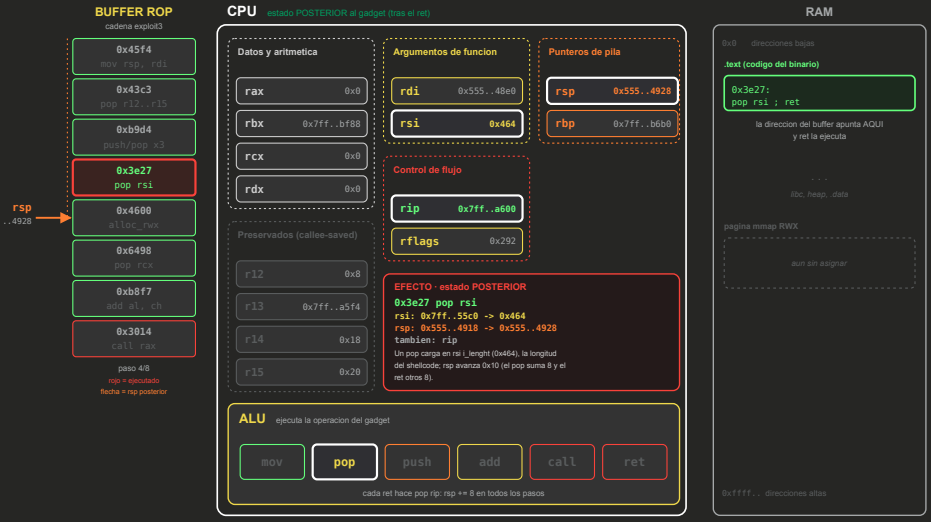
La coreografía, paso a paso



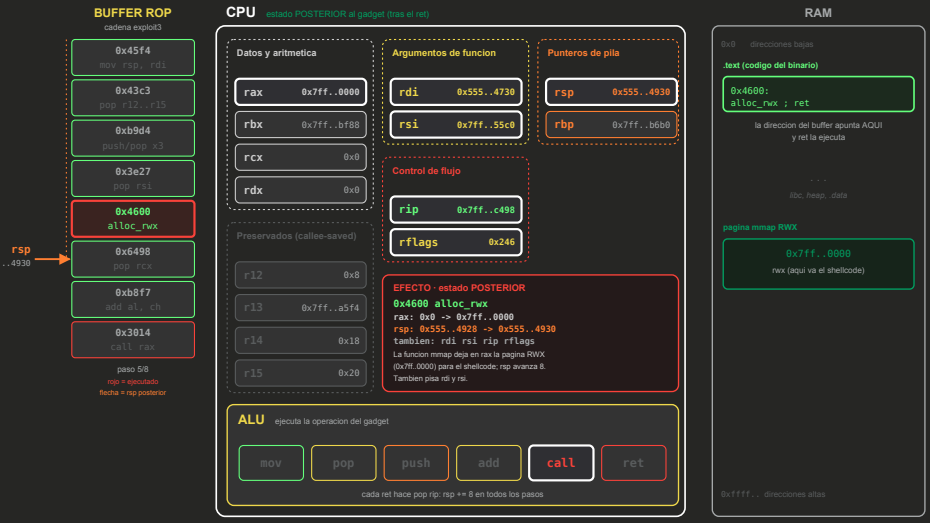
La coreografía, paso a paso



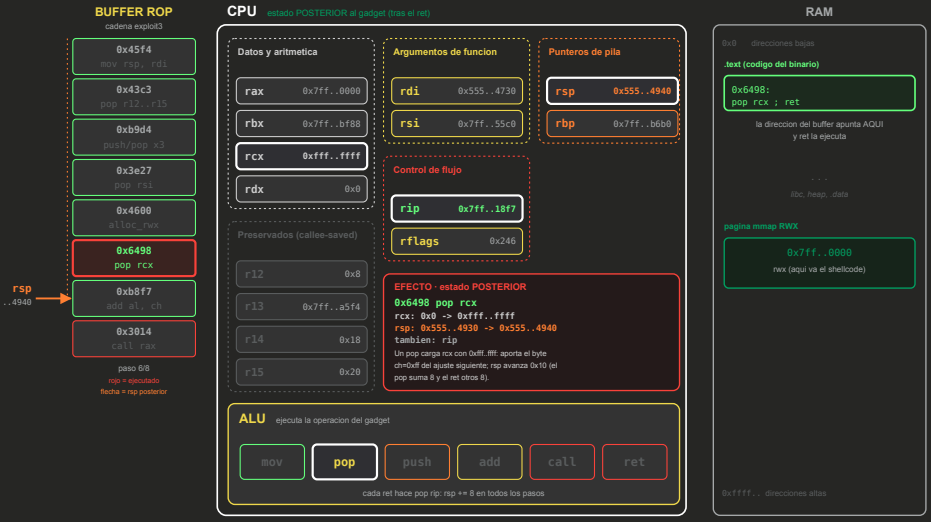
La coreografía, paso a paso



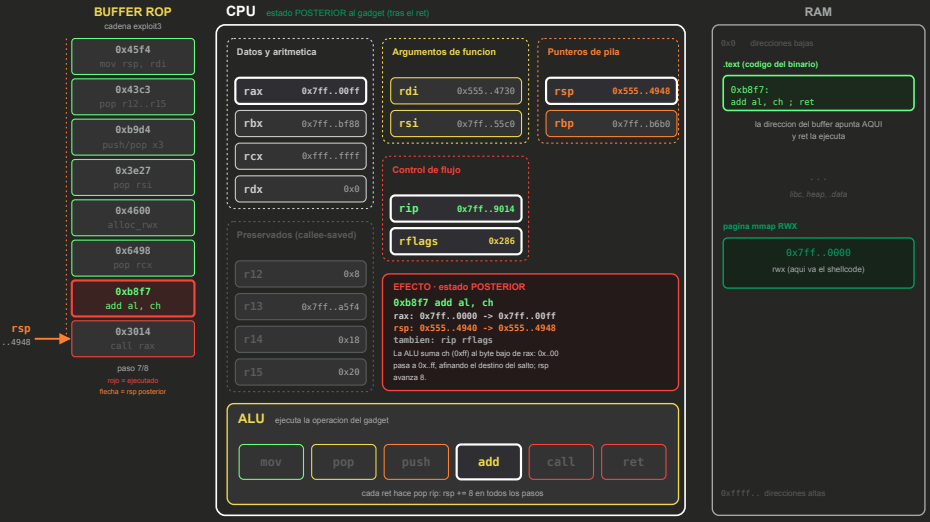
La coreografía, paso a paso



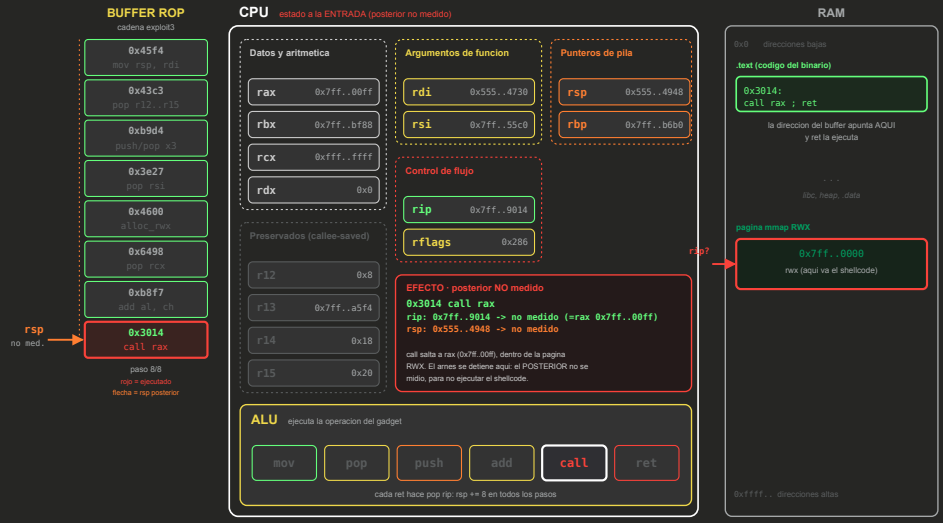
La coreografía, paso a paso



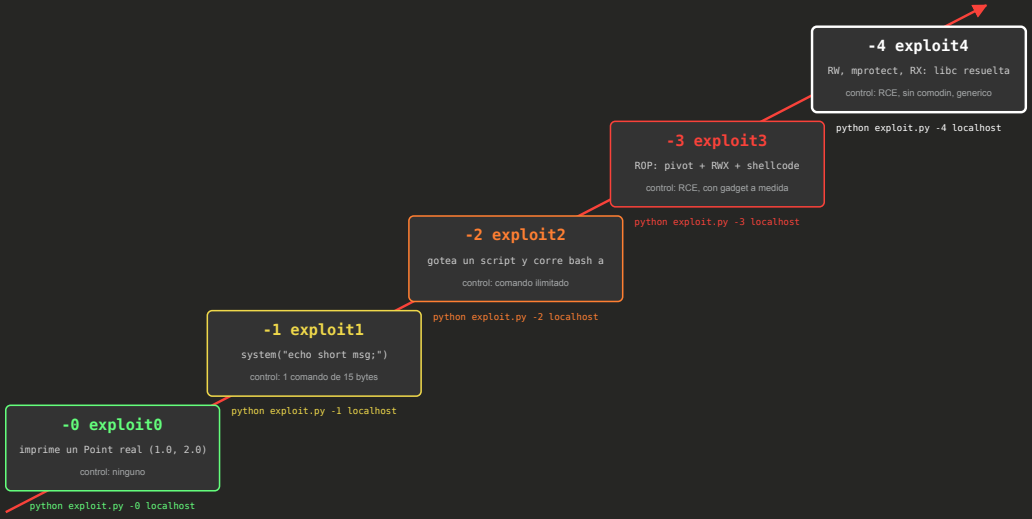
La coreografía, paso a paso



La coreografía, paso a paso



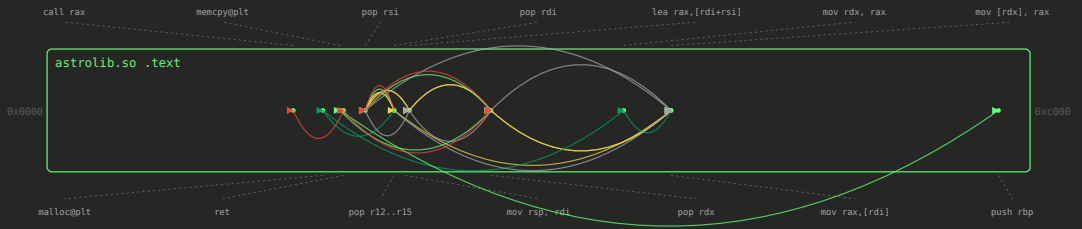
Desarollo agíl de *exploit*



Appendix

Conclusión: ROP Dance

Mi baile en tu RAM



1. pivote	2. scratch	3. malloc: RW	4. memcpy	5. GOT: libc	6. mprotect: RX	7. salto
1 gadgets	10 gadgets	4 gadgets	5 gadgets	6 gadgets	5 gadgets	1 gadgets

malloc: la pagina nace RW
se escribe charge.bin

nunca RWX
mprotect resuelto desde GOT[malloc] + delta

mprotect: la misma, RX
ya no se puede escribir

Conclusión: Más alla

dentro del tunel va lo que se ejecuta en la victima; el leak y la compilacion del shellcode pasan en la maquina del atacante y por eso quedan fuera del casco.
la linea en rojo entrega el control y de ahi sale la flecha. Las lineas largas van plegadas y los comentarios llevan sus espacios compactados; ningun token se retoca.

Recordar: lo que vimos, peldaño a peldaño

1. El mapa: la cadena de ataque entera, antes de explicar nada.
2. El bug: 24 bytes de un Point, y 8 son la dirección que salta.
3. El embudo: 1296 vectores, 40 clases, 3 poderes (RWX), 1 verdad.
4. El inframundo: el ELF en la RAM, la memoria, los registros, el ASM.
5. La escalera: la historia y el exploit ágil, peldaño a peldaño.

Cada lámina que viste fue un peldaño: el bug, siempre el mismo.

Recordar: Más alla

Teoría	Práctica
La caza y el encadenado de gadgets	Resolver los módulos de ROP de pwn.college
El ASLR y la fuga que lo vence	Fugar una dirección de libc y recalcular la base en cada corrida
Las mitigaciones de hoy	Comparar un binario propio con `checksec`: NX, PIE, canario, RELRO
La imagen del proceso en RAM	Estudiar Anatomy of a program in memory
El mapa de memoria en vivo	Leer `/proc/PID/maps` de un proceso propio y ubicar pila, heap y libc
La escritura del exploit	Subir Phoenix, de stack-zero a stack-five
La automatización de la cadena	Rehacer un exploit del taller con `pwntools` en vez de `curl`
Hola mundo sin libc	Compilar el mismo programa en NASM y en GNU as: `make nasm gas`
El texto armado en la pila	Armar el string con inmediatos, sin sección de datos: `hola-pila.s`
El ida y vuelta del desensamblado	Reensamblar y ver dónde se rompe en silencio: `make roundtrip roto`

Los tres últimos viven en `script/asm/`, con un Makefile que se verifica solo.

Referencias: de dónde sale la técnica

Nada de esto es inédito: el paper más reciente es de 2012

Tipo	Autor	Año	Título
Paper	Aleph One	1996	Smashing the Stack for Fun and Profit
Read	Solar Designer	1997	Getting around non-executable stack
Paper	Nergal	2001	Advanced return-into-lib(c) exploits
Paper	Shacham	2007	The Geometry of Innocent Flesh on the Bone
Paper	Checkoway et al.	2010	Return-Oriented Programming without Returns
Paper	Bratus et al.	2011	Exploit Programming: Weird Machines
Paper	Roemer et al.	2012	ROP: Systems, Languages, and Applications
Spec	x86-64 psABI	2024	System V ABI, AMD64 Processor Supplement

Referencias: con qué se hace

Herramientas de código abierto y un binario de fabricación propia

Tipo	Autor	Año	Título
Tool	Gallopsled	2013	pwntools
Tool	Salwan	2011	ROPgadget
Tool	pwndbg	2015	pwndbg
Tool	slimm609	2009	checksec.sh
Tool	NSA	2019	Ghidra
Read	ROP Emporium	2026	Ocho retos de ROP con binarios
Read	ir0nstone	2026	Binary Exploitation Notes
Ley	Congreso de Chile	2022	Ley 21.459 sobre delitos informáticos

Referencias: ensamblador

Type	Author	Date	Title
Ref	Felix Cloutier	2023	Assembly mnemonics
Ref	Intel	2023	The Intel Software Developer's Manual
Read	David Evans	2006	x86 Assembly Guide
Code	Keystone Engine	2023	The Keystone Framework (github)
Ref	Wikipedia	2023	x86 Instruction Listings
Read	Tom's Hardware	2000	Intel Pentium 4 Architecture
Read	William Mahoney	2019	Enumerating x86-64
Code	Web Archive	2012	NASM example
Read	Paul Carter	2019	Assembly Book