Inline Hooking in Windows

6 September 2011

Brian MARIANI
Senior Security Consultant

SOME IMPORTANT POINTS

- This document is the second of a series of five articles relating to the art of hooking.
- As a test environment we will use an english Windows Seven SP1 operating system distribution.

WHAT IS AN INLINE HOOK

- When an inline hook is implemented **it will overwrite the first bytes codes** of a windows api in order to redirect code flow.

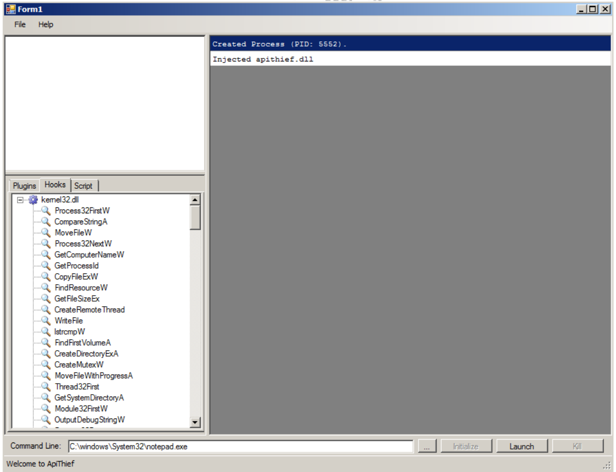
- This kind of technique can be used in ring 3 or ring 0 modes.

CHARACTERISTICS

- Inline hooking **is more robust** than import address table (IAT) hooks.
- They do not have problems related to dll binding at runtime.
- They **can trap any function call**, not just system calls.
- No device driver is required.
- They are **widely used** in many renowned professional applications and well-known rootkits.

SOME GOOD UTILITIES

- At the **Defcon 17** Nick Harbour has presented a good tool named **apithief**.
- The tool launches a process in a suspended state.
- It then injects a DLL and hook Win32 API functions.
- The tool can then monitor the api behavior.



SOME BAD UTILITIES

- Well-known and widespread malware take advantage of this kind of technique to hook key api windows components **in order to spy and steal sensitive information** from the user.

- One of these famous malwares is **Zeus**, also known as **Zbot**, PRG, **Wsnpoem** or **Gorhax**.

```
0:000> u 71ab6233
ws2_32!WSASend:
71ab6233 e985790c8f        jmp     00b7dbbd
71ab6238 51                push    ecx
71ab6239 51                push    ecx
71ab623a 813d2840ac714894ab71 cmp dword ptr [ws2_32!PrologPointer
71ab6244 56                push    esi
71ab6245 0f847f540000      je      ws2_32!WSASend+0x14 (71abb6ca)
71ab624b 8d45f8            lea     eax,[ebp-8]
71ab624e 50                push    eax
```

ws2_32!**WSASend** API hooked

Implementing an inline hook

- We inject our dll into the process we want to hijack. (Please see Userland Hooking in Windows)
- **The first bytes of the target api are saved**, these are the bytes that will be overwritten.
- We **place a jump, call** or **push-ret** instruction.
- The jump **redirects the code flow** to our function.
- The hook can later **call the original api using the saved bytes** of the hooked function.
- The original function **will return control to our function** and then **data can be easily tampered**.

WHAT ARE WE GOING TO OVERWRITE

- It is **very important to determine** what and where we are going to overwrite the bytes codes.
- Knowing exactly how much space we have at our disposal **is really important**, otherwise we can **destroy the logic** of the api, or write beyond his borders.
- An unconditional jump will require 5 bytes.
- We **need to disassemble** the beginning of the target api.

WHERE ARE WE GOING TO OVERWRITE

- The start of a windows api function is usually the same.
- After XP-SP2 Microsoft decided to change the windows api preamble:

```
MOV EDI,EDI
PUSH EBP
MOV EBP,ESP
```

- This allow for hot-patching, inserting a new code without the need to reboot the box.
- We can use these five bytes **to insert our own set of instructions.**

WHERE ARE WE GOING TO OVERWRITE

- Another reason why we select the start of the function is because **the more deeper into the function the hook is located, the more we must be careful with the code**.
- If you hook deeper into the function, you can create unwanted issues.
- Things become more complex.
- Keep it simple and stupid.

AN UNSUCCESSFUL HOOK

- If we try to hook every function in the same way we can modify the logic of the function.
- In the picture below we have inserted an unconditional jump that destroy the logic of the function.

```
773203E2   33C9           XOR ECX,ECX
773203E4   83E9 01        SUB ECX,1
773203E7   C3             RETN
773203E8   90             NOP
773203E9   90             NOP
773203EA   8BFF           MOV EDI,EDI
773203EC   55             PUSH EBP
773203ED   8BEC           MOV EBP,ESP
773203EF   6A 38          PUSH 38
```

```
773203E2   E9 E90D6EF4    JMP 6BA011D0
773203E7   C3             RETN
773203E8   90             NOP
773203E9   90             NOP
773203EA   8BFF           MOV EDI,EDI
773203EC   55             PUSH EBP
773203ED   8BEC           MOV EBP,ESP
773203EF   6A 38          PUSH 38
```

A SUCCESSFUL HOOK

- In this particular example the prolog of **httpsendrequest** API has been hooked with a jump to our function.

```
7731EE9B   8BFF          MOV EDI,EDI
7731EE9D   55            PUSH EBP
7731EE9E   8BEC          MOV EBP,ESP
7731EEA0   83EC 40       SUB ESP,40
7731EEA3   53            PUSH EBX
```

```
7731EE9B   E9 30236EF4   JMP 6BA011D0
7731EEA0   83EC 40       SUB ESP,40
7731EEA3   53            PUSH EBX
```

- We have used **the first 5 bytes** of the target function to place our **jmp** instruction.

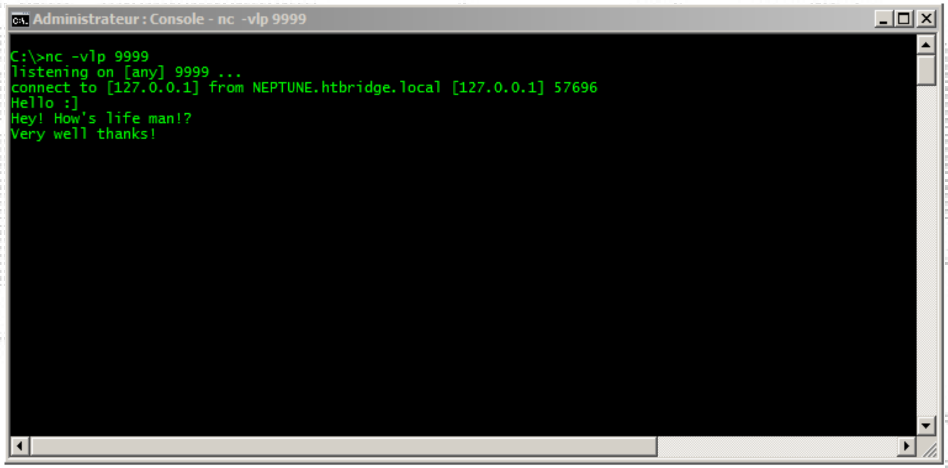
- In this practical example our target is a simple chat system using nc.exe, but every program using WS2_32.DLL!Send Api could be used.
- We want to hijack the sent messages from one of two users.
- To accomplish this task we inject our DLL into nc.exe process and we hook the aforementioned api.
- Then we jump into our function and we change the stack parameters, then we adjust our stack to avoid incovenients and finally we jump again to WS2_32.DLL!Send api.
- This hook example can be used to create more complex scenarios.

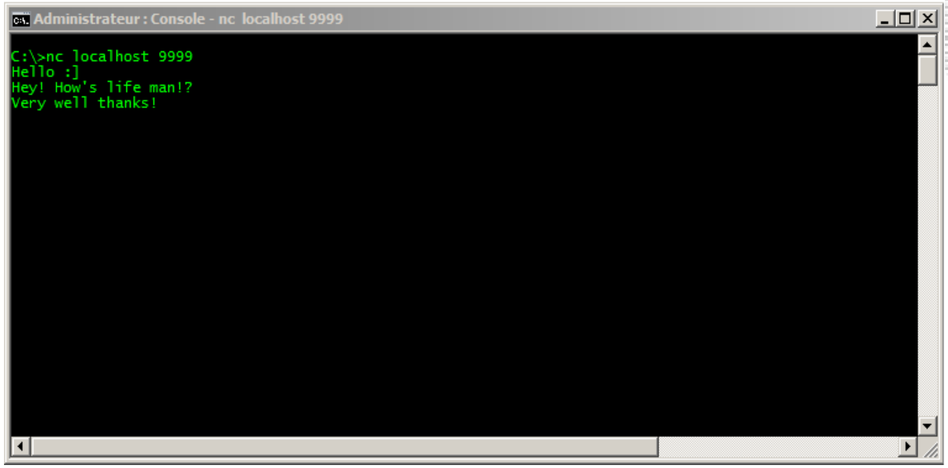
- Before hooking the WS2_32.dll!send API we attach with our debugger to the nc.exe process which is already listening on port 9999. We can see the unhook prolog.

```
7651C4C8   8BFF                 MOV EDI,EDI
7651C4CA   55                   PUSH EBP
7651C4CB   8BEC                 MOV EBP,ESP
7651C4CD   83EC 10              SUB ESP,10
7651C4D0   56                   PUSH ESI
7651C4D1   57                   PUSH EDI
7651C4D2   33FF                 XOR EDI,EDI
7651C4D4   813D 48705376 3(     CMP DWORD PTR DS:[76537048],WS2_32.7651:
7651C4DE   75 7B                JNZ SHORT WS2_32.7651C55B
7651C4E0   393D 70705376        CMP DWORD PTR DS:[76537070],EDI
7651C4E6   74 73                JE SHORT WS2_32.7651C55B
7651C4E8   FF35 44705376        PUSH DWORD PTR DS:[76537044]
7651C4EE   FF15 48125176        CALL DWORD PTR DS:[<&API-MS-Win-Core-Pr( kernel32.TlsGetValue
7651C4F4   8945 F8              MOV DWORD PTR SS:[EBP-8],EAX
7651C4F7   3BC7                 CMP EAX,EDI
7651C4F9   74 60                JE SHORT WS2_32.7651C55B
7651C4FB   897D FC              MOV DWORD PTR SS:[EBP-4],EDI
7651C4FE   FF75 08              PUSH DWORD PTR SS:[EBP+8]
7651C501   E8 6A6BFFFF          CALL WS2_32.76513070
7651C506   8BF0                 MOV ESI,EAX
7651C508   3BF7                 CMP ESI,EDI
7651C50A   0F84 C92A0000        JE WS2_32.7651EFD9
7651C510   8B4D F8              MOV ECX,DWORD PTR SS:[EBP-8]
7651C513   8B45 10              MOV EAX,DWORD PTR SS:[EBP+10]
7651C516   53                   PUSH EBX
7651C517   83C1 08              ADD ECX,8
7651C51A   8D55 FC              LEA EDX,DWORD PTR SS:[EBP-4]
7651C51D   52                   PUSH EDX
7651C51E   51                   PUSH ECX
7651C51F   57                   PUSH EDI
```

WS2_32!Send

- Server is listening for inbound connections (upper image) and the client connects (lower image). All is working normally and users are discussing without any issues.





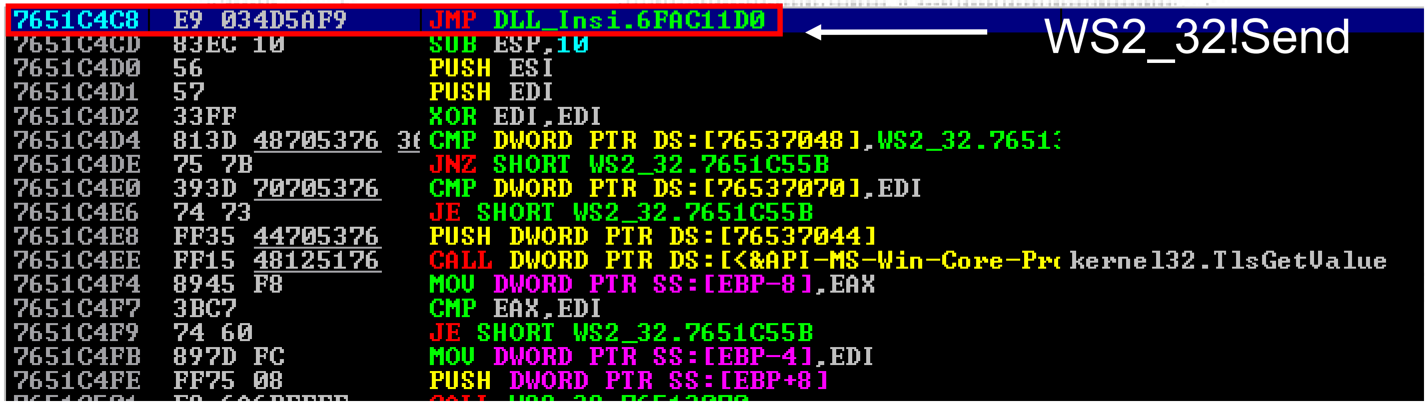
- At this time we **inject** our DLL into the **nc.exe process**.

```
C:\Users\bmariani\Desktop\DLLInsidious>injector.exe nc.exe c:\Users\bmariani\Desktop\DLLInsidious\DLL_Ins
Pid is -> 3640
-> Opening the target process...
-> Memory Allocation...
-> Writing DLL in memory...
-> Creating the Thread...
-> DLL injected :]
```

INLINE HOOKING PRACTICAL EXAMPLE (5)

- Let’s check using the our debugger how’s things have changed.

- The API preamble has been modified. It has been replaced with a jump to our evil function.

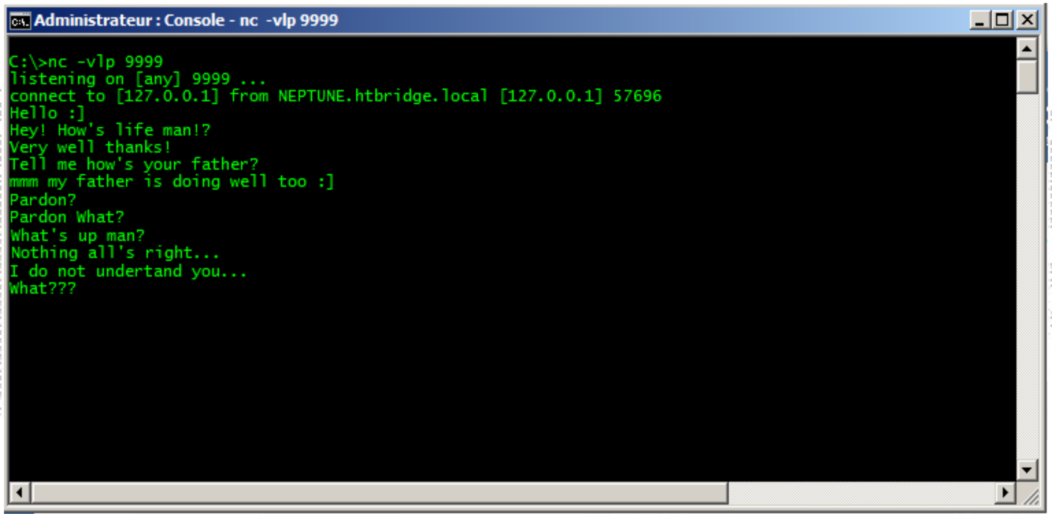


INLINE HOOKING PRACTICAL EXAMPLE (6)

- Let's check our function at address 0x6FAC11D0.

```
6FAC11D0  9C              PUSHFD
6FAC11D1  8BEC            MOV EBP,ESP
6FAC11D3  83EC 08         SUB ESP,8
6FAC11D6  B8 0030AC6F     MOV EAX,DLL_Insi.6FAC3000                    ASCII "You friend
6FAC11DB  8845 FF         MOV BYTE PTR SS:[EBP-1],AL
6FAC11DE  C745 F8 2800000 MOV DWORD PTR SS:[EBP-8],28
6FAC11E5  8945 0C         MOV DWORD PTR SS:[EBP+C],EAX
6FAC11E8  58              POP EAX
6FAC11E9  8945 10         MOV DWORD PTR SS:[EBP+10],EAX
6FAC11EC  58              POP EAX
6FAC11ED  9D              POPFD
6FAC11EE  89FF            MOV EDI,EDI
6FAC11F0  55              PUSH EBP
6FAC11F1  89EC            MOV ESP,EBP
6FAC11F3  90              NOP
6FAC11F4  90              NOP
6FAC11F5  90              NOP
6FAC11F6  90              NOP
6FAC11F7  90              NOP
6FAC11F8  90              NOP
6FAC11F9  90              NOP
6FAC11FA  90              NOP
6FAC11FB  90              NOP
6FAC11FC  90              NOP
6FAC11FD  90              NOP
6FAC11FE  90              NOP
6FAC11FF  90              NOP
6FAC1200  90              NOP
6FAC1201  90              NOP
6FAC1202  E9 C6B2A506     JMP WS2_32.7651C4CD
```

- The behavior of the chat has changed too :]



DETECTING INLINE HOOKING (1)

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- Detecting inline hooks is pretty simple.

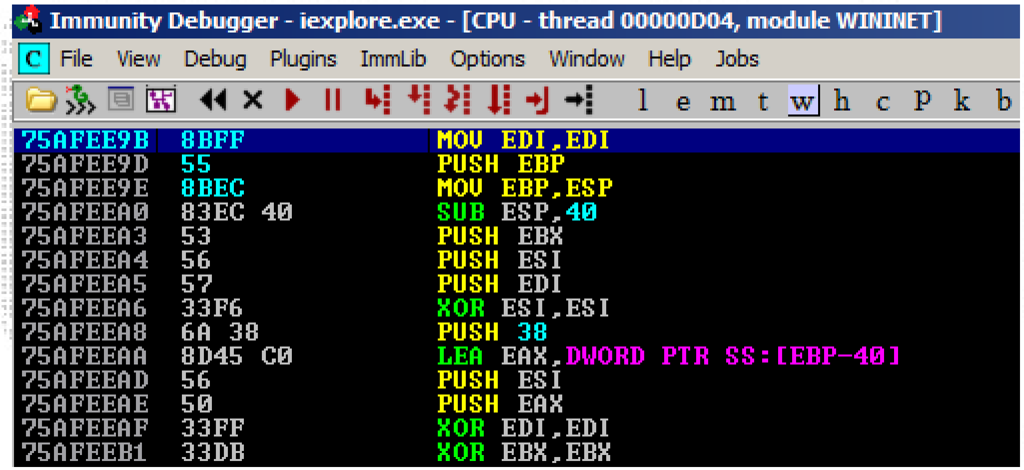
- Rootkits detectors like **gmer**, or **BlackLight** from Fsecure do a good job.

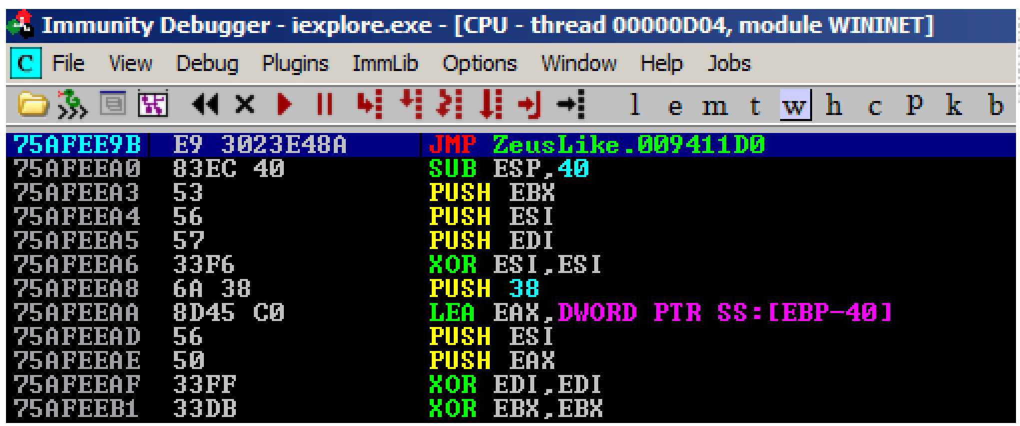
- Let's do an injection test into **Internet Explorer 8.0 in Windows 7** and hook **wininet!HttpSendRequestW** Api.

```
#include <windows.h>
#include <stdio.h>
#include <winsock.h>

__declspec(naked) MyHttpSendRequest()
{
   __asm("nop");
   __asm("nop");
   __asm("nop");
   __asm("nop");
   __asm("nop");
   __asm("nop");
   __asm("nop");
}

BOOL APIENTRY DllMain (HINSTANCE hInst,DWORD reason,LPVOID reserved)
{
char  JmpOpcode[1] =      "\xE9";
char  SavesOpcodes[5] =  "\x90\x90\x90\x90\x90";
DWORD lpflOldProtect  = 0;
HMODULE HandleModule;
DWORD AddressAPI;
DWORD AddresseFakeApi;
DWORD calculateJMP,JMP_TO,Trampoline;


    switch (reason)
    {
      case DLL_PROCESS_ATTACH:
      HandleModule = GetModuleHandle(TEXT("wininet.dll"));
      AddressAPI = GetProcAddress(HandleModule,"HttpSendRequestW");
      AddresseFakeApi = (LPDWORD)&MyHttpSendRequest;

      memcpy(SavesOpcodes,AddressAPI,0x5);
      calculateJMP = AddresseFakeApi - AddressAPI;
      JMP_TO = calculateJMP - 5;

      VirtualProtect(AddressAPI,0x8,PAGE_READWRITE,&lpflOldProtect);
      memcpy(AddressAPI,JmpOpcode,0x1);
      memcpy(AddressAPI+1,&JMP_TO,0x4);

      __asm("int3");
      break;
    }
    return TRUE;
}
```

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- The hook is in place :]





DETECTING INLINE HOOKING (3)

- But it has been successfully detected by **gmer anti-rootkit**.

GMER 1.0.15.15641

Rootkit/Malware | >>>

Type	Name	Value
.text	ntkrnlpa.exe!KiDispatchInterrupt + 5A2	82AB9092 19 Bytes [E0, 0F, BA, F0, 07, 73, 09, ...] {LOOPNZ 0x11; MOV EDX, 0x97307f0; MOV CR4, EAX..
.text	ntkrnlpa.exe!RtlSidHashLookup + 230	82AC0890 4 Bytes [C6, 6A, D2, 8D]
.text	ntkrnlpa.exe!RtlSidHashLookup + 2C8	82AC0928 4 Bytes [98, 72, D2, 8D]
.text	ntkrnlpa.exe!RtlSidHashLookup + 2EC	82AC094C 4 Bytes [C0, 76, D2, 8D] {SAL BYTE [ESI-0x2e], 0x8d}
.text	ntkrnlpa.exe!RtlSidHashLookup + 308	82AC0968 4 Bytes [8C, B4, D2, 8D]
.text	ntkrnlpa.exe!RtlSidHashLookup + 32C	82AC098C 4 Bytes [8C, 69, D2, 8D]
.text	...	
?	C:\Windows\System32\Drivers\SafeBoot.sys	Le processus ne peut pas accéder au fichier car ce fichier est utilisé par un autre processus.
.text	autochk.exe	002D11E8 4 Bytes [9C, 37, CC, 32]
.text	autochk.exe	002D11EF 3 Bytes [80, 6F, 02]
.text	autochk.exe	002D1204 4 Bytes [00, 00, 00, 00] {ADD [EAX], AL; ADD [EAX], AL}
.text	autochk.exe	002D120C 1 Byte [00]
.text	autochk.exe	002D1210 1 Byte [00]
.text	...	
.text	C:\Program Files\Mozilla Firefox\firefox.exe[4664] ntdll.dll!LdrLoadDll	7761F5B5 5 Bytes JMP 013D13F0 C:\Program Files\Mozilla Firefox\firefox.exe (Firefox/Mozilla Corporation)
.text	C:\Program Files\Internet Explorer\iexplore.exe[5500] WININET.dll!HttpSendRequestW	75AFEE9B 5 Bytes JMP 009411D0 c:\Users\bmariani\Desktop\ZeusLike\ZeusLike.dll

CONCLUSION

- Inline hooks can be very useful when trying to reverse malware.
- Paradoxically nowadays malcode use this technique to change the behavior of the operating system in order to steal personal information.
- By understanding and identifying hooks, thought, you will be able to detect most public rootkits and malwares.
- We hope this document help you understand and master inline hooks.
- Demo video: [Inline Hooking in Windows]

TO BE CONTINUED

- In future documents we will introduce Kernel Hooking.

REFERENCES

- Subverting the Windows Kernel (Greg Hoglund & James Butler)
- Professional Rootkits (Ric Vieler)
- Programming application for Microsoft Windows fourth edition (Jeffrey Ritchter)
- http://msdn.microsoft.com/en-us/magazine/cc301805.aspx
- Programming Windows Security (Keith Brown)
- http://support.microsoft.com/kb/197571
- http://www.youtube.com/watch?v=2RJyR9igsdI
- http://community.websense.com/blogs/securitylabs/archive/tags/Zeus/default.aspx
- http://www.ibm.com/developerworks/linux/library/l-ia/index.html
- http://www.gmer.net/

Thank-you for reading

Thank you for reading

Your questions are always welcome!

Brian.mariani@htbridge.ch