



Improving the HideDebugger function

Shub-Nigurrath of ARTeam

Version 1.1 - July 2006

1.	Abstract	2
2.	Intercepted APIs, how are used and relative patches.....	3
3.	Accessing the PEB structure	3
3.1.	Limitation of the NtQueryInformationProcess API documentation	5
3.1.1	Complete PROCESS_BASIC_INFORMATION	5
3.1.2	Complete ProcessInformationClass	5
3.1.3	Complete PEB structure.....	6
3.2.	Manipulating the PEB	7
3.3.	What is the ProcessHeaps and ForceFlags?	9
3.4.	Code to access the PEB elements	11
3.4.1	Access the Thread Selector Entry (TSE)	11
3.4.2	Access the Frame Segment FS.....	12
3.4.3	Manipulating the PEB elements.....	12
3.4.4	Manipulating the ProcessHeaps elements.....	13
3.5.	CheckRemoteDebuggerPresent().....	13
3.6.	ZwQueryInformationProcess()	15
4.	How-To inject Offset Independent Code into a victim, with C	16
4.1.	Patch of ZwQueryInformationProcess()	21
5.	Coding the whole solution	24
6.	Testing the approach	28
7.	References.....	30
8.	Conclusions.....	30
9.	History.....	31
10.	Greetings	31

Keywords

anti-debugging, C



1. Abstract

In the Issue number 1 of this eZine [1] I wrote a paper on improving the anti-debugging features of StraceNT, an API tracing utility which was release as freeware but without support for any type of anti debugging. This lack prevented its usage with protected programs such as AsProtected targets.

I then introduced a Dll having one single export called HavePhun() and explained how to modify the main application so as to let it load the Dll at the right point, just before executing the traced target.

Now I will explain how to write a much more powerful hiding routine which could be simply substituted to the previous Dll to more efficiently hide StraceNT.

The code I will present of course can be used for your loaders to hide them to more aggressive tests the loaded programs might do, it is written to be easily ported to any program you might want to write.

As usual I will provide sample code with this tutorial, and non-commercial sample victims. All the sources have been tested with Win2000/XP and Visual Studio 6.0.
The techniques described here are general and not specific to any commercial applications. The whole document must be intended as a document on programming advanced techniques, how you will use these information will be totally up to your responsibility.



2. Intercepted APIs, how are used and relative patches

First of all there are plenty ways to detect a debugger or detect that a program is debugged. A tutorial on all the possible ways would end up being so much space that is a too big effort to write it. Anyway a first distinction must be done: some anti-debugging tricks detect the presence of Olly or other specific programs, while other tests detects that the program is debugged by a program.

The first type of checks is not really an anti-debugging test, is rather an environment sanity check. I will not worry of these sanity checks because are strictly tied to the fact for example that you are using Olly for debugging or other similar tools. I'm planning to use the code I wrote for custom loaders, which are usually not checked directly.

I will worry instead of the debuggers tests which are implemented fishing information from the system, either through system APIs or accessing to kernel structures or kernel windows objects. There are even in this case plenty ways to detect that a program is being debugged and a function might not hope to hide from any possible test. Execryptor protector is just the most famous example of this type which has several sanity environment checks as well as different debugging and exotic tests..

3. Accessing the PEB structure

We start from IsDebuggerPresent API. This is the most simple to use test and is used in a simple way, something like the following:

```
BOOL bRet=IsDebuggerPresent();  
return ((bRet)?(POSITIVE):(NEGATIVE));
```

The API returns a positive value if a debugger is detected. If you inspect what the API does you will easily see that it access to some system structures where a debugged bit is read and reported to the caller. I already described this behaviour an earlier tutorial [2], but anyway what is accessed is the Process Environment Block (PEB) BeingDebugged field.

The disassembling of the API clearly shows what I told:

7C812E03 64:A1 18000000	MOV	EAX,DWORD PTR FS:[18]	;access the FSBase
7C812E09 8B40 30	MOV	EAX,DWORD PTR DS:[EAX+30]	;access RVApeb
7C812E0C 0FB640 02	MOVZX	EAX,BYTE PTR DS:[EAX+2]	;access BeingDebugged
7C812E10 C3	RET		;EAX contains 1 if debugged

Figure 1, taken from a tutorial of Benina I found on the net, might help to understand the situation.

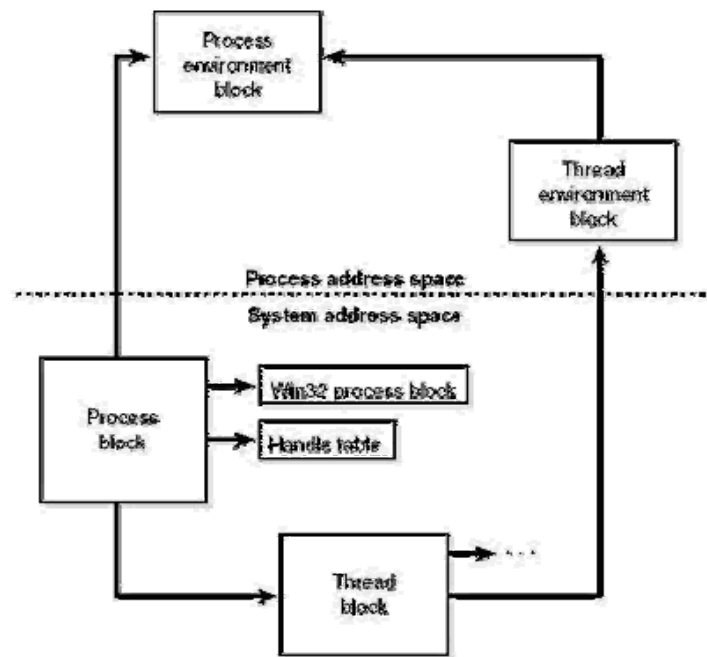


Figure 1 - Data structures associated with processes and threads

The PEB structure contains all *User-Mode* parameters associated by system with current process. The TEB (Thread Environment Block) instead is associated to each thread of a process.

In order to fool this API is enough to zero out the BeingDebugged value, but if with ASM is relatively simple to access the PEB, with C is a little more complex.

The system offers a simple method for the programs to directly access their own PEB to directly check this value (IsDebuggerPresent API); indeed this is not what happens all the times you want to seriously protect you programs: a call to the above API is for sure easily detected by most reversers.

IsBeingDebugged accesses the PEB with a couple of ASM instructions you also can directly use for your own programs. Another way is to use the Win32 API NtQueryInformationProcess¹, a more generic function which can be used to access to the PEB base (see Figure 2).

NtQueryInformationProcess

[NtQueryInformationProcess is available for use in Windows 2000 and Windows XP. It may be altered or unavailable in subsequent versions. Applications should use the alternate functions listed in this topic.]

The NtQueryInformationProcess function retrieves information about the specified process.

```
NTSTATUS NtQueryInformationProcess(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength
);
```

Parameters

ProcessHandle

[in] Handle to the process for which information is to be retrieved.

ProcessInformationClass

[in] Type of process information to be retrieved. This parameter can be one of the following values from the **PROCESSINFOCLASS** enumeration.

ProcessBasicInformation

Retrieves a pointer to a PEB structure that can be used to determine whether the specified process is being debugged, and a unique value used by the system to identify the specified process. It is best to use the [CheckRemoteDebuggerPresent](#) and [GetProcessId](#) functions to obtain this information.

ProcessWow64Information

Determines whether the process is running in the WOW64 environment (WOW64 is the x86 emulator that allows Win32-based applications to run on 64-bit Windows). It is best to use the [IsWow64Process](#) function to obtain this information.

ProcessInformation

[in, out] Pointer to a buffer supplied by the calling application into which the function writes the requested information. The size of the information written varies depending on the value of the *ProcessInformationClass* parameter.

Figure 2 - NtQueryInformationProcess MSDN details

¹ NtQueryInformationProcess is a forwarder to the real API ZwQueryInformationProcess



The code to access the PEB via the NtQueryInformationProcess API would look like following:

```
ULONG bytes=0;
PROCESS_BASIC_INFORMATION ProcessInfo;
NTSTATUS ntstatus = NtQueryInformationProcess(hproc, ProcessBasicInformation,
                                             (PVOID)&ProcessInfo, sizeof(PROCESS_BASIC_INFORMATION), &bytes);
if(ntstatus != STATUS_SUCCESS || bytes<=0 ) {
    <error>
}
ProcessInfo.pebBaseAddress is the pointer the the PEB structure.
```

Where PROCESS_BASIC_INFORMATION is a structure like this, according to MSDN:

```
typedef struct _PROCESS_BASIC_INFORMATION {
    PVOID Reserved1;
    PPEB PebBaseAddress;
    PVOID Reserved2[2];
    ULONG_PTR UniqueProcessId;
    PVOID Reserved3;
} PROCESS_BASIC_INFORMATION;
```

the PebBaseAddress is the pointer to the base of the PEB.

Accessing the PebBaseAddress you can access the BeingDebugged value and test it.

If you need to do these steps for a remote process (not yourself) you can simply obtain the hproc through the OpenProcess API and then call the NtQueryInformationProcess passing hproc like below:

```
HANDLE hproc=OpenProcess(PROCESS_ALL_ACCESS, FALSE, GetProcessId());
```

3.1. Limitation of the NtQueryInformationProcess API documentation

The documentation of the API omits several details for all it's parameters, details that are really lot useful.

3.1.1 Complete PROCESS_BASIC_INFORMATION

The PROCESS_BASIC_INFORMATION's definition above is what MSDN reports in the documentation of the API, but what are those Reserved1, 2, 3 elements? They have a really useful meaning. The complete structure is the following one:

```
typedef struct _tagPROCESS_BASIC_INFORMATION
{
    DWORD ExitStatus;
    DWORD PebBaseAddress;
    DWORD AffinityMask;
    DWORD BasePriority;
    ULONG UniqueProcessId;
    ULONG InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION;
```

Just compare the two definitions and do your own conclusions. The InheritedFromUniqueProcessId is the ID of the process which launched our process².

3.1.2 Complete ProcessInformationClass

MSDN also reports that ProcessInformationClass can have only two possible values:

² Have you ever used Process Explorer from Sysinternals, and have you seen it's hierarchical view of processes?



- ProcessBasicInformation
- ProcessWow64Information

The whole range of possible values is instead much longer. We will use several not documented options in the following of this document..

```
typedef enum _PROCESSINFOCLASS {
    ProcessBasicInformation,
    ProcessQuotaLimits,
    ProcessIoCounters,
    ProcessVmCounters,
    ProcessTimes,
    ProcessBasePriority,
    ProcessRaisePriority,
    ProcessDebugPort,
    ProcessExceptionPort,
    ProcessAccessToken,
    ProcessLdtInformation,
    ProcessLdtSize,
    ProcessDefaultHardErrorMode,
    ProcessIoPortHandlers, // Note: this is kernel mode only
    ProcessPooledUsageAndLimits,
    ProcessWorkingSetWatch,
    ProcessUserModeIoPL,
    ProcessEnableAlignmentFaultFixup,
    ProcessPriorityClass,
    ProcessWx86Information,
    ProcessHandleCount,
    ProcessAffinityMask,
    ProcessPriorityBoost,
    ProcessDeviceMap,
    ProcessSessionInformation,
    ProcessForegroundInformation,
    ProcessWow64Information,
    ProcessImageFileName,
    ProcessLUIDDeviceMapsEnabled,
    ProcessBreakOnTermination,
    ProcessDebugObjectHandle,
    ProcessDebugFlags,
    ProcessHandleTracing,
    MaxProcessInfoClass
} PROCESSINFOCLASS;
```

3.1.3 Complete PEB structure

Also the PEB is not completely documented. MSDN reports a partially hidden structure: this the structure of the PEB it reports:

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[229];
    PVOID Reserved3[59];
    ULONG SessionId;
} PEB, *PPEB;
```

as you can imagine behind that Reserved1,2,3 vectors there are plenty of useful information we can use.

This is the complete PEB definition:

```
typedef struct _PEB {
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged;
    BOOLEAN Spare;
    HANDLE Mutant;
    PVOID ImageBaseAddress;
```



```
PPEB_LDR_DATA LoaderData;
PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
PVOID SubSystemData;
PVOID ProcessHeap;
PVOID FastPebLock;
PPEBLOCKROUTINE FastPebLockRoutine;
PPEBLOCKROUTINE FastPebUnlockRoutine;
ULONG EnvironmentUpdateCount;
PPVOID KernelCallbackTable;
PVOID EventLogSection;
PVOID EventLog;
PPEB_FREE_BLOCK FreeList;
ULONG TlsExpansionCounter;
PVOID TlsBitmap;
ULONG TlsBitmapBits[0x2];
PVOID ReadOnlySharedMemoryBase;
PVOID ReadOnlySharedMemoryHeap;
PPVOID ReadOnlyStaticServerData;
PVOID AnsiCodePageData;
PVOID OemCodePageData;
PVOID UnicodeCaseTableData;
ULONG NumberOfProcessors;
ULONG NtGlobalFlag;
BYTE Spare2[0x4];
LARGE_INTEGER CriticalSectionTimeout;
ULONG HeapSegmentReserve;
ULONG HeapSegmentCommit;
ULONG HeapDeCommitTotalFreeThreshold;
ULONG HeapDeCommitFreeBlockThreshold;
ULONG NumberOfHeaps;
ULONG MaximumNumberOfHeaps;
PPVOID *ProcessHeaps;
PVOID GdiSharedHandleTable;
PVOID ProcessStarterHelper;
PVOID GdiDCAttributeList;
PVOID LoaderLock;
ULONG OSMajorVersion;
ULONG OSMinorVersion;
ULONG OSBuildNumber;
ULONG OSPlatformId;
ULONG ImageSubSystem;
ULONG ImageSubSystemMajorVersion;
ULONG ImageSubSystemMinorVersion;
ULONG GdiHandleBuffer[0x22];
ULONG PostProcessInitRoutine;
ULONG TlsExpansionBitmap;
BYTE TlsExpansionBitmapBits[0x80];
ULONG SessionId;
} PEB, *PPEB;
```

There's much inside the PEB!

3.2. Manipulating the PEB

The PEB.BeingDebugged byte is not the only one that can be used to test if the program is debugged or not. There are other bytes which we are interested to manipulate in order to better hide our debugger:

- PEB.BeingDebugged, already discussed before.
- PEB.NtGlobalFlag
- PEB.ProcessHeaps

Figure 3, Figure 4 reports for example the result of a tool able to show the PEB of a running process.

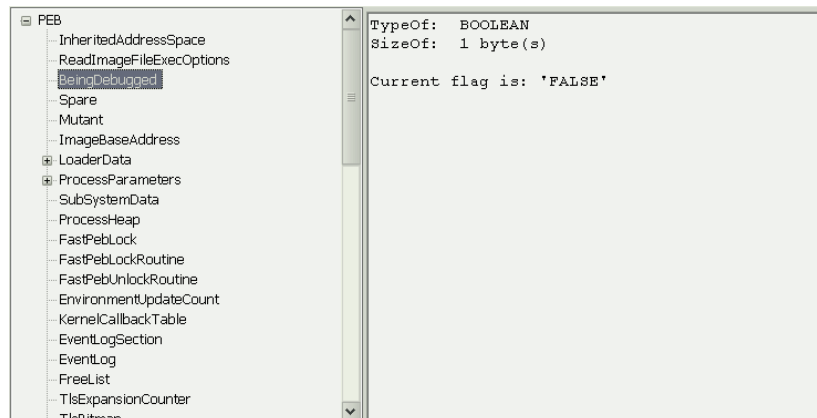


Figure 3 - PEB exploration tool, BeingDebugged value

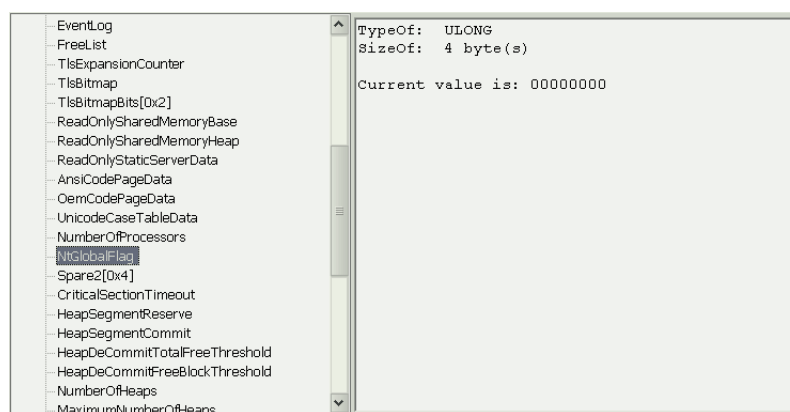


Figure 4 - PEB exploration tool, NtGlobalFlag

The ProcessHeaps is a little different, because it is a pointer to another structure which has to be accessed and modified too (as shown in Figure 5). It by definition represents the First byte after PEB, but holds some information too.

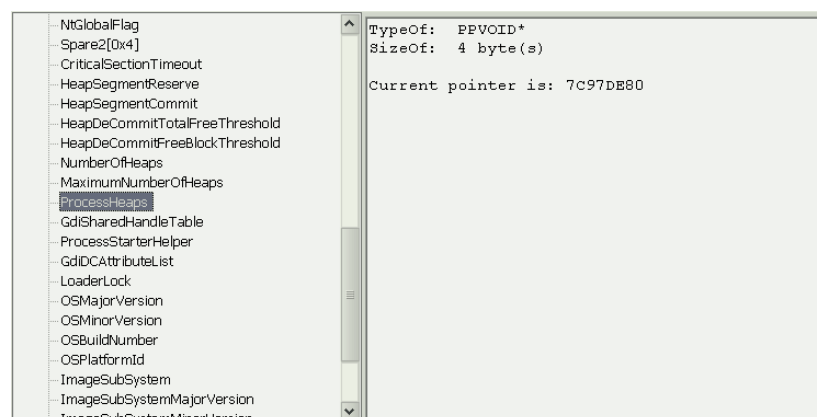


Figure 5 - PEB exploration tool, ProcessHeaps

Note that there's an API which reads the ProcessHeap of the calling process (see Figure 6), but it works only for process from which is called. We want the ProcessHeaps of a remote process and thus we will access to this memory section through the PEB as described before.



GetProcessHeaps

The `GetProcessHeaps` function obtains handles to all of the heaps that are valid for the calling process.

```
DWORD GetProcessHeaps(  
    DWORD NumberOfHeaps,  
    PHANDLE ProcessHeaps  
);
```

Parameters

NumberOfHeaps

[In] Maximum number of heap handles that can be stored into the buffer pointed to by *ProcessHeaps*.

ProcessHeaps

[Out] Pointer to a buffer that receives an array of heap handles.

Figure 6 - `GetProcessHeaps` API

3.3. What is the `ProcessHeaps` and `ForceFlags`?

The `ProcessHeaps` pointed memory structure is not much documented and indeed there's no official documentation on it. Anyway there's something we know. When you are debugging at Ring3, Windows uses a 'debug' form of heap allocation function calls instead of the standard ones. This different behaviour is needed because in debugging mode the system does several additional checks required in order to debug the program. So, debuggee can check this to discover if it is debugged or not. The structure of the `ProcessHeaps` memory is then different if you are debugging or not the program.

If you dump the memory section with WinDBG via the `!heap` command you obtain the following:

```
0:000> !heap  
Index Address Name Debugging options enabled  
1: 00140000 tail checking free checking validate parameters  
2: 00240000 tail checking free checking validate parameters  
3: 00250000 tail checking free checking validate parameters
```

```
0:000> !heap -v 1  
Index Address Name Debugging options enabled  
1: 00140000  
Segment at 00140000 to 00240000 (00003000 bytes committed)  
Flags: 50000062  
ForceFlags: 40000060  
Granularity: 8 bytes  
Segment Reserve: 00100000  
Segment Commit: 00002000  
DeCommit Block Thres: 00000200  
DeCommit Total Thres: 00002000  
Total Free Size: 000000c9  
Max. Allocation Size: 7ffdefff  
Lock Variable at: 00140608  
Next TagIndex: 0000  
Maximum TagIndex: 0000  
Tag Entries: 00000000  
PsuedoTag Entries: 00000000  
Virtual Alloc List: 00140050  
UCR FreeList: 00140598  
FreeList Usage: 00000000 00000000 00000000 00000000  
FreeList[ 00 ] at 00140178: 001429c0 . 001429c0 (1 block )
```

```
0:000> dd 140000  
00140000 000000c8 0000016b eeffeeff 50000062  
00140010 40000060 0000fe00 00100000 00002000  
00140020 00000200 00002000 000000c9 7ffdefff  
00140030 06080001 00000000 00000000 00000000  
00140040 00000000 00140598 00000017 ffffffff8  
00140050 00140050 00140050 00140640 00000000  
00140060 00000000 00000000 00000000 00000000  
00140070 00000000 00000000 00000000 00000000
```



What is interesting for us is the ForceFlags element which has not zero values when the program is debugged. This value holds all the flag used by the system to allocate memory in the Heap, a value of 0 means normal allocation process, any other value is used to further specify the debugging system behaviour.

All the possible values are these:

```
/* SYSTEM_GLOBAL_FLAG.GlobalFlag constants */
#define FLG_STOP_ON_EXCEPTION          0x00000001
#define FLG_SHOW_LDR_SNAPS             0x00000002
#define FLG_DEBUG_INITIAL_COMMAND      0x00000004
#define FLG_STOP_ON_HUNG_GUI           0x00000008
#define FLG_HEAP_ENABLE_TAIL_CHECK     0x00000010
#define FLG_HEAP_ENABLE_FREE_CHECK     0x00000020
#define FLG_HEAP_VALIDATE_PARAMETERS   0x00000040
#define FLG_HEAP_VALIDATE_ALL          0x00000080
#define FLG_POOL_ENABLE_TAIL_CHECK     0x00000100
#define FLG_POOL_ENABLE_FREE_CHECK     0x00000200
#define FLG_POOL_ENABLE_TAGGING        0x00000400
#define FLG_HEAP_ENABLE_TAGGING        0x00000800
#define FLG_USER_STACK_TRACE_DB        0x00001000
#define FLG_KERNEL_STACK_TRACE_DB      0x00002000
#define FLG_MAINTAIN_OBJECT_TYPELIST    0x00004000
#define FLG_HEAP_ENABLE_TAG_BY_DLL     0x00008000
#define FLG_IGNORE_DEBUG_PRIV          0x00010000
#define FLG_ENABLE_CSRDEBUG            0x00020000
#define FLG_ENABLE_KDEBUG_SYMBOL_LOAD  0x00040000
#define FLG_DISABLE_PAGE_KERNEL_STACKS 0x00080000
#define FLG_HEAP_ENABLE_CALL_TRACING    0x00100000
#define FLG_HEAP_DISABLE_COALESCING     0x00200000
#define FLG_ENABLE_CLOSE_EXCEPTIONS     0x00400000
#define FLG_ENABLE_EXCEPTION_LOGGING    0x00800000
#define FLG_ENABLE_DBGPRINT_BUFFERING   0x08000000
```

Moreover there's an undocumented setting that can also be used to force this value to always be set to 0:

set _NO_DEBUG_HEAP=1

this way the value is kept to 0 by the system, but it's a global behaviour. If you enter the above command in DOS window and dump again the structure like above you will notice that ForceFlags is set to 0.

```
0:000> !heap
Index Address Name           Debugging options enabled
  1:  00140000
  2:  00240000
  3:  00250000
0:000> !heap -v 1
Index Address Name           Debugging options enabled
  1:  00140000
    Segment at 00140000 to 00240000 (00004000 bytes committed)
    Flags:                00000002
    ForceFlags:          00000000
    Granularity:           8 bytes
    Segment Reserve:       00100000
    Segment Commit:        00002000
    DeCommit Block Thres:  00002000
    DeCommit Total Thres:  00002000
    Total Free Size:       000001fe
    Max. Allocation Size:  7ffdefff
    Lock Variable at:      00140608
    Next TagIndex:         0000
    Maximum TagIndex:      0000
    Tag Entries:           00000000
```



```
PsuedoTag Entries:      00000000
Virtual Alloc List:     00140050
UCR FreeList:           00140598
FreeList Usage:         00000000 00000000 00000000 00000000
FreeList[ 00 ] at 00140178: 00143018 . 00143018 (1 block )
0:000> dd 140000
00140000 000000c8 000001a1 eeffeeff 00000002
00140010 00000000 0000fe00 00100000 00002000
00140020 00000200 00002000 000001fe 7ffdefff
00140030 06080001 00000000 00000000 00000000
00140040 00000000 00140598 0000000f ffffffff8
00140050 00140050 00140050 00140640 00000000
00140060 00000000 00000000 00000000 00000000
00140070 00000000 00000000 00000000 00000000
```

A program can use a piece of code like this, to check if it's debugged or not:

```
DWORD dwForceFlags=0;
HANDLE hproc=OpenProcess(PROCESS_ALL_ACCESS,FALSE, GetCurrentProcessId());
if(hproc) {
    HANDLE hpc=GetProcessHeap();
    if(hpc) {
        SIZE_T numread;
        if (!ReadProcessMemory(hproc, (LPCVOID)((DWORD)hpc + 0x10),
            &dwForceFlags, 4, &numread) || numread != 4)
            return UNKNOWN;
    }
}
CloseHandle(hproc);
return ((dwForceFlags==0)?("Not Debugged"):( "Debugged!"));
```

What we can do is to wipe out the ForceFlags value or modify in order to change the Heaps behaviour: we will do a complete wiping.

3.4. Code to access the PEB elements

In order to change the PEB elements you have to access the structure, but we will not follow the official method explained above, we will access things directly so as to skip the incomplete structure definitions Microsoft exposes.

3.4.1 Access the Thread Selector Entry (TSE)

The first API you need is the one shown in Figure 7:

GetThreadSelectorEntry
The **GetThreadSelectorEntry** function retrieves a descriptor table entry for the specified selector and thread.

```
BOOL GetThreadSelectorEntry(
    HANDLE hThread,
    DWORD dwSelector,
    LPLDT_ENTRY lpSelectorEntry
);
```

Parameters
hThread
[in] Handle to the thread containing the specified selector. The handle must have `THREAD_QUERY_INFORMATION` access. For more information, see [Thread Security and Access Rights](#).
dwSelector
[in] Global or local selector value to look up in the thread's descriptor tables.
lpSelectorEntry
[out] Pointer to an [LDT_ENTRY](#) structure that receives a copy of the descriptor table entry if the specified selector has an entry in the specified thread's descriptor table. This information can be used to convert a segment-relative address to a linear virtual address.
Return Values
If the function succeeds, the return value is nonzero. In that case, the structure pointed to by the *lpSelectorEntry* parameter receives a copy of the specified descriptor table entry.
If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Figure 7 – GetThreadSelectorEntry

which needs the HANDLE of the remote thread, the selector value of the remote thread and a pointer to an LDT_ENTRY structure.



The selector is taken directly from the remote process calling the `GetThreadContext` like in the following code:

```
// Set up the victimContext access flag
victimContext.ContextFlags = CONTEXT_SEGMENTS;
// Fill the victim context structure with process data
if (!GetThreadContext(thread, &victimContext))
    return FALSE;

// GetThreadSelectorEntry is only functional on x86-based systems.
// For systems that are not x86-based, the function returns FALSE
// The GetThreadSelectorEntry function fills this structure with
// information from an entry in the descriptor table. You can use
// this information to convert a segment-relative address to a linear virtual address.
// The base address of a segment is the address of offset 0 in the segment.
// To calculate this value, combine the BaseLow, BaseMid, and BaseHi members
LDT_ENTRY sel;
if (!GetThreadSelectorEntry(thread, victimContext.SegFs, &sel))
    return FALSE;
```

3.4.2 Access the Frame Segment FS

Once you have the `LDT_ENTRY` you can use it in order to calculate the FS base address. This address must be calculated since Windows XP SP2, because Microsoft changed the system so as to allow this structure to be relocable. They realized that most viruses were accessing this structure too much easily with a single ASM instruction so they changed it so as each process can allocate it where needed.

The way to go is the following:

```
//Gets the real address of FS
DWORD fsbase = (sel.HighWord.Bytes.BaseHi<<8 | sel.HighWord.Bytes.BaseMid)<<16 | sel.BaseLow;
DWORD RVApeb;
SIZE_T numread;

//Access PTR FS:[30], from that address takes the RVA PEB address.
//This works also for relocated PEBs (WINXP SP2)
if (!ReadProcessMemory(hproc, (LPVOID)(fsbase + 0x30), &RVApeb, 4, &numread) || numread != 4)
    return FALSE;
```

the final step of the above code reads the memory pointed by the FS base address plus 0x30, which is equivalent to reading FS:[30] value. The 30th element of the Frame Selector is the RVA value of the PEB. Remember that we are accessing the thread and process elements of a remote process so we need to read them from the target process's memory.

3.4.3 Manipulating the PEB elements

The last step consists in accessing the PEB elements so as to be able to modify them. We need to modify the following members:

- PEB.BeingDebugged
- PEB.NtGlobalFlag
- PEB.ProcessHeaps

The code is always the same: access to the correct offset where we know from documentation that the value is stored, store and modify it locally, write back the new value.

```
////////////////////////////////////
//PEB.BeingDebugged patch
WORD wBeingDebugged;
if (!ReadProcessMemory(hproc, (LPVOID)(RVApeb + 0x2), &wBeingDebugged, 2, &numread) ||
    numread != 2)
```



```
return FALSE;

wBeingDebugged = 0;
if (!WriteProcessMemory(hproc, (LPVOID)(RVApceb + 0x2), &wBeingDebugged, 2, &numread) ||
    numread != 2)
    return FALSE;

////////////////////////////////////
//NtGlobalFlag patch

//Access the PEB+0x68 which is the NtGlobalFlag, is a ULONG
DWORD dwNtGlobalFlag;
if (!ReadProcessMemory(hproc, (LPVOID)(RVApceb + 0x68), &dwNtGlobalFlag, 4, &numread) ||
    numread != 4)
    return FALSE;

dwNtGlobalFlag=0;
if (!WriteProcessMemory(hproc, (LPVOID)(RVApceb + 0x68), &dwNtGlobalFlag, 4, &numread) ||
    numread != 4)
    return FALSE;
```

3.4.4 Manipulating the ProcessHeaps elements

What we need is to read the ProcessHeaps value and then use it to access the 10th DWORD after (ForceFlags element), which stores another check value used to detect the debugger's presence.

```
////////////////////////////////////
// ProcessHeap

//Access the PEB+0x18 which is the ProcessHeap, is a DWORD
DWORD dwProcessHeap;
if (!ReadProcessMemory(hproc, (LPVOID)(RVApceb + 0x18), &dwProcessHeap, 4, &numread) ||
    numread != 4)
    return FALSE;

DWORD dwForceFlags;
if (!ReadProcessMemory(hproc, (LPVOID)(dwProcessHeap + 0x10), &dwForceFlags, 4, &numread) ||
    numread != 4)
    return FALSE;

dwForceFlags=0;
if (!WriteProcessMemory(hproc, (LPVOID)(dwProcessHeap + 0x10), &dwForceFlags, 4, &numread) ||
    numread != 4)
    return FALSE;

////////////////////////////////////
```

3.5. CheckRemoteDebuggerPresent()

The API is able the equivalent of IsDebuggerPresent , but remote processes, it received an HANDLE to an open process and reports if it's debugged or not.

```
BOOL bVal=FALSE
BOOL bRet=CheckRemoteDebuggerPresent(hProc, &bVal);
If(bRet==TRUE && bVal==TRUE)
    printf ("debugger found!\n");
```

its definition according to MSDN is the one of Figure 8. Note that in order to be debugged you must return some values and set to true the pDebuggerPresent pointed memory cell.



CheckRemoteDebuggerPresent

The CheckRemoteDebuggerPresent function determines whether the specified process is being debugged.

```
BOOL CheckRemoteDebuggerPresent (  
    HANDLE hProcess,  
    PBOOL pbDebuggerPresent  
);
```

Parameters

hProcess
[in] Handle to the process.

pbDebuggerPresent
[in, out] Pointer to a variable that the function sets to TRUE if the specified process is being debugged, or FALSE otherwise.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Figure 8- CheckRemoteDebuggerPresent MSDN definition

I found a reliable way to fool the API which consists on patching it.

This is the API disassembly

```
7C859936      3945 08      CMP DWORD PTR SS:[EBP+8],EAX  
7C859939      0F95C0      SETNE AL                      ; <-- NOP  
7C85993C      8906      MOV DWORD PTR DS:[ESI],EAX  
7C85993E      33C0      XOR EAX,EAX  
7C859940      40      INC EAX                      ; <-- NOP  
7C859941      EB 09      JMP SHORT kernel32.7C85994C  
7C859943      6A 57      PUSH 57  
7C859945      E8 76F9FAFF CALL kernel32.SetLastError  
7C85994A      33C0      XOR EAX,EAX  
7C85994C      5E      POP ESI  
7C85994D      5D      POP EBP  
7C85994E      C2 0800      RETN 8
```

Well, the code following code is far from being optimized and a little redundant, but believe me it's like that just for educational purposes, not my coding skills limitations ^_^

```
////////////////////////////////////  
//CheckRemoteDebuggerPresent in another way because sometimes return TRUE!  
// 7C859936      3945 08      CMP DWORD PTR SS:[EBP+8],EAX  
// 7C859939      0F95C0      SETNE AL                      ; <-- NOP  
// 7C85993C      8906      MOV DWORD PTR DS:[ESI],EAX  
// 7C85993E      33C0      XOR EAX,EAX  
// 7C859940      40      INC EAX                      ; <-- NOP  
// 7C859941      EB 09      JMP SHORT kernel32.7C85994C  
// 7C859943      6A 57      PUSH 57  
// 7C859945      E8 76F9FAFF CALL kernel32.SetLastError  
// 7C85994A      33C0      XOR EAX,EAX  
// 7C85994C      5E      POP ESI  
// 7C85994D      5D      POP EBP  
// 7C85994E      C2 0800      RETN 8  
hKer = GetModuleHandle("KERNEL32");  
addrIDP = GetProcAddress(hKer, "CheckRemoteDebuggerPresent");  
BYTE patch_bytes1[]={0x90, 0x90, 0x90};  
BYTE ori_bytes1[] = {0x0F, 0x95, 0xC0}; //SETNE AL  
BYTE off_bytes1 = 0x37;  
BOOL bApply_patch1=TRUE;  
  
BYTE patch_bytes2[]={0x90};  
BYTE ori_bytes2[] = {0x40}; //INC EAX  
BYTE off_bytes2 = 0x3E;  
BOOL bApply_patch2=TRUE;  
  
BYTE temp_bytes[3]; //stores temporary bytes read to compare with expected..  
int idx=0;  
  
if (addrIDP==NULL)  
    return FALSE;
```



```
//We must change permission to containing page
VirtualProtectEx(hproc, addrIDP, 5, PAGE_EXECUTE_READWRITE, &dwOldProt);

//Patches the SETNE AL
if (!ReadProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes1),
    &temp_bytes, 3, &numread) || numread != 3)
    return FALSE;

for(idx=0; idx<3; idx++)
    if(temp_bytes[idx]!=ori_bytes1[idx])
        bApply_patch1=FALSE;

if(bApply_patch1)
    if (!WriteProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes1),
        &patch_bytes1, 3, &numread) || numread != 3)
        return FALSE;

//Patches the INC EAX
if (!ReadProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes2),
    &temp_bytes, 1, &numread) || numread != 1)
    return FALSE;

for(idx=0; idx<1; idx++)
    if(temp_bytes[idx]!=ori_bytes2[idx])
        bApply_patch2=FALSE;

if(bApply_patch2)
    if (!WriteProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes2),
        &patch_bytes2, 1, &numread) || numread != 1)
        return FALSE;
```

3.6. ZwQueryInformationProcess()

As said before this the same of NtQueryInformationProcess and can be used to access several Windows objects, this is a really generic API and so it's not that easy to patch. Anyway it can be used to get access to the DebugObject and thus as an anti-debug method.

Let see some possible pieces of code a program can use to check if it's debugged or not, using this API. We need to access the ProcessDebugPort. As explained in previous sections this is an undocumented feature of this API.

The code is the following:

```
//Open the process and then open the main thread so as I can read anything inside.
HANDLE hproc=OpenProcess(PROCESS_ALL_ACCESS,FALSE, GetCurrentProcessId());
BOOL bRet=FALSE;

if(hproc) {
    ULONG bytes=0;
    DWORD ProcessInfoBuffer=0;

    NTSTATUS nStatus= ZwQueryInformationProcess(hproc, ProcessDebugPort,
        &ProcessInfoBuffer, 4, &bytes);

    if(nStatus!= STATUS_SUCCESS)
        return "Error";

    if(bytes>0) {
        if(ProcessInfoBuffer!=0)
            bRet=TRUE;
        else
            return "Not Debugged";
    }
    else
        return "Error";

    CloseHandle(hproc);
    return ((bRet)?("Debugged"):( "Not Debugged"));
}
```



If ProcessInfoBuffer returned is not NULL the program is degugged.

4. How-To inject Offset Independent Code into a victim, with C

In order to patch the ZwQueryInformationProcess API we need to inject a new API body into the victim's address space, so as the victim process (and only it) will use the changed API. There's a general technique that can be useful to insert patches inside another process, directly from C++, using Offset Independent Code (see deroko tutorial [2] to understand what OIC is).

The method uses the naked attribute, clearly documented into MSDN.

For functions declared with the naked attribute, the compiler generates code without prolog and epilog code. You can use this feature to write your own prolog/epilog code sequences using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

```
__declspec( naked ) declarator
```

Because the naked attribute is only relevant to the definition of a function and is not a type modifier, naked functions must use extended attribute syntax and the __declspec keyword.

The compiler cannot generate an inline function for a function marked with the naked attribute, even if the function is also marked with the __forceinline keyword.

Examples

This code defines a function with the naked attribute:

```
__declspec( naked ) int func( formal_parameters )
{
    // Function body
}
```

Or, alternatively:

```
#define Naked __declspec( naked )
Naked int func( formal_parameters )
{
    // Function body
}
```

The naked attribute affects only the nature of the compiler's code generation for the function's prolog and epilog sequences. It does not affect the code that is generated for calling such functions. Thus, the naked attribute is not considered part of the function's type, and function pointers cannot have the naked attribute. Furthermore, the naked attribute cannot be applied to a data definition. For example, this code sample generates an error:

```
__declspec( naked ) int i;           // Error--naked attribute not
                                     // permitted on data declarations.
```

The naked attribute is relevant only to the definition of the function and cannot be specified in the function's prototype. For example, this declaration generates a compiler error:

```
__declspec( naked ) int func();      // Error--naked attribute not
                                     // permitted on function declarations
```




For example here's how the same code with naked and not naked attributes looks.

```
#include <stdio.h>
void func1();

void main() {
    printf("Here in the main\n");
    func1();
}

void func1() {
    printf("Here in the func1()\n");
}
```

The result is what is shown in Figure 9

00401000	\$ 55	PUSH EBP		
00401001	. 8BEC	MOV EBP,ESP		
00401003	. 68 30704000	PUSH trial.00407030		ASCII "Here in the main\n"
00401008	. E8 1C000000	CALL trial.00401029		call printf
0040100D	. 83C4 04	ADD ESP,4		
00401010	. E8 02000000	CALL trial.00401017		call func1
00401015	. 5D	POP EBP		kernel32.7C816D4F
00401016	. C3	RETN		
00401017	\$ 55	PUSH EBP		func1
00401018	. 8BEC	MOV EBP,ESP	prolog	
0040101A	. 68 44704000	PUSH trial.00407044		ASCII "Here in the func1()\n"
0040101F	. E8 05000000	CALL trial.00401029	<- function body	call printf
00401024	. 83C4 04	ADD ESP,4		
00401027	. 5D	POP EBP		kernel32.7C816D4F
00401028	. C3	RETN	epilog	
00401029	\$ 53	PUSH EBX		printf
0040102A	. 56	PUSH ESI		

Figure 9 - example func1() with prolog and epilog clearly shown

The prolog and the epilog are simple instructions used to ensure the stack integrity and Olly knows them very well, it is able to easily recognize the func1 boundaries, just using them. Note the the final RETN is inserted by the compiler.

This is instead the same code with the naked attribute for func1

```
#include <stdio.h>
void func1();

void main() {
    printf("Here in the main\n");
    func1();
}

__declspec( naked ) void func1() {
    printf("Here in the func1()\n");
}
```

The Figure 10 shows the result: the func1 body is now only made of the instructions we wrote, a printf of a string that is translated to a PUSH and a call to the inlined printf, no return, no stack integrity additional information, just the ADD ESP,4 to remove the PUSH did at 00401017. All is up to the user responsibility. As you can see OllyDbg is no more able to correctly recognize call to func1.



00401000	\$ 55	PUSH EBP	
00401001	. 8BEC	MOV EBP,ESP	
00401003	. 68 30704000	PUSH trial_n.00407030	ASCII "Here in the main\n"
00401008	. E8 17000000	CALL trial_n.00401024	call printf
0040100D	. 83C4 04	ADD ESP,4	
00401010	. E8 02000000	CALL trial_n.00401017	call func1
00401015	. 5D	POP EBP	kernel32.7C816D4F
00401016	. C3	RETN	
00401017	\$ 68 44704000	PUSH trial_n.00407044	ASCII "Here in the func1()\n"
0040101C	. E8 03000000	CALL trial_n.00401024	call printf
00401021	. 83C4 04	ADD ESP,4	
00401024	\$ 53	PUSH EBX	printf
00401025	. 56	PUSH ESI	
00401026	. BE 90704000	MOV ESI,trial_n.00407090	

Figure 10 - example func1() with naked attribute

So for example to have again the same compiled code we saw in Figure 9 we can do something as following.

```
#include <stdio.h>
void func1();

void main() {
    printf("Here in the main\n");
    func1();
}

__declspec( naked ) void func1() {
    __asm{
        PUSH EBP
        MOV EBP,ESP
    }
    printf("Here in the func1()\n");
    __asm{
        POP EBP
        RETN
    }
}
```

Once said these basic things we can concentrate quite fast on the injection technique. What we want to do is to write into the address space of a target program a piece of ASM or even C code. We want to do this having still the possibility offered by the compiler to write the injection code before compilation. The other possibility is to write an array of bytes (the opcodes) into the destination process, this method has the drawback that it's not easily maintainable.

Suppose we need to Hook or modify a function called FooFunc into the target process (for example a system API or a function imported from a DLL). We need two naked functions:

```
void start_FooFuncHook();
void end_FooFuncHook ();
```

where the start_FooFuncHook() contains the code to be injected in the target process at the beginning of the FooFunc function:

```
__declspec( naked ) void start_FooFuncHook () {
    //the code to be inserted in the target process
}
```

The end_FooFuncHook() is an empty function instead (being naked it will not produce a single bit of code) that must be placed **immediately** after the end of the start_FooFuncHook in the source code.

```
__declspec( naked ) void end_FooFuncHook (void) { }
```

In another section of the program you will need a call WriteProcessMemory



```
FARPROC addrIDP;
HINSTANCE hInst;
//function pointer to the FooFunc, depends on the FooFunc prototypes. This is only
//required to bypass the C compiler type checking: also functions have a type associated,
//that's tied to the function prototype.
//fcnFooFunc is a typedef defined for example like following:
//
//typedef NTSTATUS (NTAPI *fcnZwQueryInformationProcess)(HANDLE, PROCESSINFOCLASS, PVOID,
// ULONG, PULONG);
fcnFooFunc fcn;
DWORD dwOldProt=0;
hInst = GetModuleHandle("targetDll"); //for example NTDLL
addrIDP = GetProcAddress(hKer, "FooFunc");
if (addrIDP==NULL)
    return FALSE;

//Convert function pointers to DWORD and calculate the difference among pointer..
int cbCodeSize=((LPBYTE) end_FooFuncHook - (LPBYTE) start_FooFuncHook);

//Change Page permissions so as to write the patch
dwOldProt=0;
fcn=(fcnFooFunc)addrIDP;

//We must change permission for 5 bytes only, but the system changes the whole memory
//page containing that bytes
if(!VirtualProtectEx(hproc, addrIDP, 5, PAGE_EXECUTE_READWRITE, &dwOldProt))
    return FALSE;

//Allocates remotely the required memory
//the function determines where to allocate memory
LPVOID lpAllocatedMem = VirtualAllocEx( hproc, NULL, cbCodeSize + 5, MEM_COMMIT,
                                       PAGE_EXECUTE_READWRITE);

if(lpAllocatedMem==NULL)
    return FALSE;

//Writes the new function to the allocated memory.
if (!WriteProcessMemory(hproc, lpAllocatedMem, (LPVOID)&start_FooFuncHook,
                        cbCodeSize, &numread) || numread != (SIZE_T)cbCodeSize)
    return FALSE;

//Patch the entry point of the FooFunc to point to the new function.
//Code the JMP at the lpAllocatedMem
// JMP is coded as relative JMP less the instruction width which is 4 for DWORD of the
// JMP plus 1 for the OP-Code
// It's coded as: JMP destination_address - current_address - instruction_length(5 bytes)
DWORD jmp_length=((LPBYTE)lpAllocatedMem-(LPBYTE)addrIDP) - 5;
BYTE jmp_hook[5]={0xE9, 0x00, 0x00, 0x00, 0x00,};
jmp_hook[1]= (BYTE)(jmp_length);
jmp_hook[2]= (BYTE)(jmp_length >> 8);
jmp_hook[3]= (BYTE)(jmp_length >> 16);
jmp_hook[4]= (BYTE)(jmp_length >> 24);

//Writes the JMP to the new patch at the API entrypoint.
if (!WriteProcessMemory(hproc, (LPVOID)addrIDP, &jmp_hook, 5, &numread) || numread != 5)
    return FALSE;

//Restore Old page protection, some packer detect page protection changes..
VirtualProtectEx(hproc, addrIDP, 5, dwOldProt, NULL);
```



The call is somehow completed by some surrounding check, which might not always be useful, but are required for a safer code:

1. get the FooFunc function address through GetProcAddress. This, due to C syntax requires also a definition of the function pointer, like the typedef used fcnFooFunc
2. calculate the size of the bytes to be written as
(LPBYTE) end_FooFuncHook - (LPBYTE) start_FooFuncHook
3. Obtain the address of the function to be Hooked through the function pointers



```
fcn=(fcnFooFunc)addrIDP;
```

4. Change the permission of the page which containing the function (for example if it's in a system dll might be required) using VirtualProtectEx³

```
VirtualProtectEx(hproc, addrIDP, 5, PAGE_EXECUTE_READWRITE, &dwOldProt)
```

5. Remotely allocate the required space for the new injected code through VirtualAllocEx

```
LPVOID lpAllocatedMem = VirtualAllocEx( hproc, NULL, cbCodeSize + 5, MEM_COMMIT,  
                                       PAGE_EXECUTE_READWRITE)
```

6. Write the new code to the remote process using as start address the start_FookFuncHook and the size calculated as step 1.

```
WriteProcessMemory(hproc, lpAllocatedMem, (LPVOID)&start_FooFuncHook, cbCodeSize, &numread)
```

7. Patch the entry point of the FooFunc function to point to the new injected code, for example with a JMP to the lpAllocatedMem.

This step requires a little knowledge of how relative JMPs are coded. The JMPs always jumps to a relative offset. This offset is calculated as following:

```
JMP destination_address - current_address - instruction_length(5 bytes)
```

Moreover the destination address has to be written with LSB logic, from the last to the first byte. So for example a JMP to address 0x01100000 from 0x7C90E01B (entry point of the API) becomes a JMP to the address 0x7B80E016 which is coded as: E9 16E0807B.

```
DWORD jmp_length=((LPBYTE)lpAllocatedMem-(LPBYTE)addrIDP) - 5;  
BYTE jmp_hook[5]={0xE9, 0x00, 0x00, 0x00, 0x00,};  
jmp_hook[1]= (BYTE)(jmp_length);  
jmp_hook[2]= (BYTE)(jmp_length >> 8);  
jmp_hook[3]= (BYTE)(jmp_length >> 16);  
jmp_hook[4]= (BYTE)(jmp_length >> 24);
```

8. Writes the new FooFunc entry point with the coded JMP

```
WriteProcessMemory(hproc, (LPVOID)addrIDP, &jmp_hook, 5, &numread)
```

9. Restore the old memory page protections flags, as found at step 3

```
VirtualProtectEx(hproc, addrIDP, 5, dwOldProt, NULL);
```

Of course I chosen to patch the original function entry point with a simple JMP to the new code just injected, but also a:

```
PUSH lpAllocatedMem  
RETN
```

would have worked fine. This last solution would have been even easier, because there's no need to calculate offsets for the JMP, but I like more complex things..

The interesting thing is that using the naked attribute we were able to force the compiler to not modify the functions bodies and also we were able to calculate the size of the injected function: if we consider that the compilation process is sequential the end_FooFuncHok placed just after the start_FooFuncHook acts as a place holder and does not generate real code. The second extremely big advantage is that we can code the start_FooFuncHook function with C, leaving us a big freedom of expression.

³ 5 bytes or even 1 is enough, because of the VirtualProtectEx changes the permission of the whole memory page.



4.1. Patch of ZwQueryInformationProcess()

What I did to patch this function is to code what follows, using the method described in Section 4. I will follow here the process I did to figure out a possible solution and the limitations of some approaches. First of all I tried this solution that patches the API with some conditions, for example only when there are specific parameters. If not it executes the remaining of the original API, returning the control to the caller from which I patched the JMP (0x7C90E01B, ZwQueryInformationProcess EP). I used the advantages described earlier to mix C code and ASM, but figured out some little problems. I think that discussing this approach here too, might help other to understand errors.

```
//see http://msdn2.microsoft.com/en-US/library/h5w10wxs.aspx for the naked attribute!
//naked tells to compiler to do not create prolog and epilog for this specific function, it's
perfect for ASM functions!
__declspec( naked ) void start_ZwQueryInformationProcessHook () {

    FARPROC hZw;
    hZw = GetProcAddress(GetModuleHandle("NTDLL"), "ZwQueryInformationProcess");

    __asm {
        cmp [esp+8], 7
        je short no_zwqueryinformationprocess
        //executes the normal function as usual
        //address of the ZwQueryInformationProcess entry point on the stack + 5 bytes
        //skipping the first instruction already done
        mov ebx, hZw           //gets the API entry point
        add ebx, 5h           //add 5 to skip the initial instruction (1 byte opcode + 4 byte
                             //for dword operand)
        mov eax, 9Ah          //original instruction of the API overwritten by the JMP at the
                             //API entry point
        jmp ebx               //jump to the original left part of the API into NTDLL

    no_zwqueryinformationprocess:
        mov eax, [esp+0ch]
        cmp eax, 0           //if eax==0 then exit hook without doing anything
        je exit_hook
        mov [eax], 0
        push eax
        push ebx
        mov eax, [esp+18h] //points to how many bytes your API should read
        mov ebx, [esp+1Ch] //point to how many bytes the API read indeed
        cmp ebx, 0           //if the last parameter is null, just skip it.
                             //Otherwise emulate the function
        je exit_hook
        mov [ebx], eax       //these two values must be the same to behave like original

    exit_hook:
        pop ebx
        pop eax
        mov eax, 0           //eax must be set to 0 meaning that all went fine
        ret 14h
    }
}
```

I reported only the naked function; the surrounding code used to inject this code into the target is reported in Section 4 and 5.

The ZwQueryInformationProcess API before the patch looks like the following:

7C90E01B	\$ B8 9A000000	MOV EAX, 9A
7C90E020	. BA 0003FE7F	MOV EDX, 7FFE0300
7C90E025	. FF12	CALL DWORD PTR DS:[EDX]
7C90E027	.- E9 D41F8683	JMP 00170000

Figure 11 - ZwQueryInformationProcess disassembled view



Figure 11 reports the API how it looks before patch. As you can see there's a MOV EAX, 9A which identifies the service asked and then MOV into EDX of the API which will enter the SYSTEM to perform the function [4]:

```

7C90EB8B >/\$ 8BD4          MOV EDX,ESP
7C90EB8D | . 0F34          SYSENTER
7C90EB8F | . 90             NOP
7C90EB90 | . 90             NOP
7C90EB91 | . 90             NOP
7C90EB92 | . 90             NOP
7C90EB93 | . 90             NOP
7C90EB94 >\$ C3             RETN

```

After the patch will look as in Figure 12 and the JMP will point to the new code as in Figure 13.

7C90E01B	- E9 E01F8584	JMP 01160000
7C90E020	BA 0003FE7F	MOV EDX,7FFE0300
7C90E025	. FF12	CALL DWORD PTR DS:[EDX]
7C90E027	.- E9 D41F8683	JMP 00170000

Figure 12 - ZwQueryInformationProcess after the patch

01160000	68 B0144200	PUSH 4214B0	ASCII "ZwQueryInformationProcess"
01160005	68 CC144200	PUSH 4214CC	ASCII "NTDLL"
0116000A	FF15 9CC04100	CALL DWORD PTR DS:[<&KERNEL32.GetModuleHandleA>]	kernel32.GetModuleHandleA
01160010	50	PUSH EAX	
01160011	FF15 A0C04100	CALL DWORD PTR DS:[<&KERNEL32.GetProcAddress>]	kernel32.GetProcAddress
01160017	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
0116001A	807C24 08 07	CMP BYTE PTR SS:[ESP+8],7	
0116001F	74 0D	JE SHORT 0116002E	
01160021	8B5D FC	MOV EBX,DWORD PTR SS:[EBP-4]	
01160024	83C3 05	ADD EBX,5	
01160027	B8 9A000000	MOV EAX,9A	
0116002C	FFE3	JMP EBX	
0116002E	8B4424 0C	MOV EAX,DWORD PTR SS:[ESP+C]	
01160032	83F8 00	CMP EAX,0	
01160035	74 14	JE SHORT 0116004B	
01160037	C600 00	MOV BYTE PTR DS:[EAX],0	
0116003A	50	PUSH EAX	
0116003B	53	PUSH EBX	
0116003C	8B4424 18	MOV EAX,DWORD PTR SS:[ESP+18]	
01160040	8B5C24 1C	MOV EBX,DWORD PTR SS:[ESP+1C]	
01160044	83FB 00	CMP EBX,0	
01160047	74 02	JE SHORT 0116004B	
01160049	8903	MOV DWORD PTR DS:[EBX],EAX	
0116004B	5B	POP EBX	
0116004C	58	POP EAX	
0116004D	B8 00000000	MOV EAX,0	
01160052	C2 1400	RETN 14	

Figure 13 – ZwQueryInformationProcess new allocated buffer

The problem with the solution proposed above, as also shown in Figure 13, is that what we used as local strings ("NTDLL" and "ZwQueryInformationProcess") is instead stored into our process's memory space, and not in the target's address space.

```

01160000    68 B0144200    PUSH 4214B0          ; ASCII "ZwQueryInformationProcess"
01160005    68 CC144200    PUSH 4214CC          ; ASCII "NTDLL"

```

The addresses 4214B0 and 4214CC are local to the loader

This method so, after the final injection would fail because the two initial PUSHes will not find the referenced strings in the target process. A solution would be to allocate also these two strings into the target space, but would take things longer and more complex than required. I so switched to a much more simpler solution, reported below: I replicated the code of the



ZwQueryInformationProcess locally. This makes my patch less resilient because if that API changes (different OSe or SPs) I will not be aware, but it works for all XPs so it's fine too.

```
__declspec( naked ) void start_ZwQueryInformationProcessHook ( ) {  
  
    __asm {  
        cmp [esp+8], 7  
        je short no_zwqueryinformationprocess  
        //executes the normal function as usual  
        //the code comes from the original implementation of the API  
        mov eax,9Ah  
        mov edx,7FFE0300h  
        call [edx]  
        retn 14h  
  
    no_zwqueryinformationprocess:  
        mov eax,[esp+0ch]  
        cmp eax,0           //if eax==0 then exit hook without doing anything  
        je exit_hook  
        mov [eax], 0  
        push eax  
        push ebx  
        mov eax, [esp+18h] //points to how many bytes your API should read  
        mov ebx, [esp+1Ch] //point to how many bytes the API read indeed  
        cmp ebx, 0         //if the last parameter is null, just skip it.  
                           //Otherwise emulate the function  
  
        je exit_hook  
        mov [ebx], eax     //these two values must be the same to behave like original  
  
    exit_hook:  
        pop ebx  
        pop eax  
        mov eax,0         //eax must be set to 0 meaning that all went fine  
        ret 14h  
  
    }  
}
```

This latter solution contains all offset independent references and all references are local to the start_ZwQueryInformationProcessHook function (see Figure 14).

01250000	807C24 08 07	CMP BYTE PTR SS:[ESP+8],7
01250005	74 0F	JE SHORT 01250016
01250007	B8 9A000000	MOV EAX,9A
0125000C	BA 0003FE7F	MOV EDX,7FFE0300
01250011	FF12	CALL DWORD PTR DS:[EDX]
01250013	C2 1400	RETN 14
01250016	8B4424 0C	MOV EAX,DWORD PTR SS:[ESP+C]
0125001A	83F8 00	CMP EAX,0
0125001D	74 14	JE SHORT 01250033
0125001F	C600 00	MOV BYTE PTR DS:[EAX],0
01250022	50	PUSH EAX
01250023	53	PUSH EBX
01250024	8B4424 18	MOV EAX,DWORD PTR SS:[ESP+18]
01250028	8B5C24 1C	MOV EBX,DWORD PTR SS:[ESP+1C]
0125002C	83FB 00	CMP EBX,0
0125002F	74 02	JE SHORT 01250033
01250031	8903	MOV DWORD PTR DS:[EBX],EAX
01250033	5B	POP EBX
01250034	58	POP EAX
01250035	B8 00000000	MOV EAX,0
0125003A	C2 1400	RETN 14
0125003D	0000	ADD BYTE PTR DS:[EAX],AL

Figure 14 – ZwQueryInformationProcess final patch



5. Coding the whole solution

We are not ready to code a whole function for hiding our debugger to different system APIs. Several things have been already discussed above and here's only a reassume. I assume you have read the previous paper [1] and understood how the HideDebugger function there presented is injected into StraceNT.

As usual the entry point of the Dll is the following function, which received a pointer to the structure `PROCESS_INFORMATION` directly from StraceNT.

The characters array "coded", used in the following code, is just a useless thing that allows inserting into compiled code my signature.

```
extern "C" int HavePhun(PROCESS_INFORMATION *pPI) {
    char coded[256];
    sprintf(coded, "Coded by SHub-Nigurath of ARTeam v1.1.");

    return HideDebugger(pPI->hThread, pPI->hProcess);
}
```

The function `HideDebugger` is called from `HavePhun()`.

Here's the code. Of course it's a bit longer than the previous version (also because it's commented) but it does the following hidings:

- Patch `IsDebuggerPresent()`
- Patch `CheckRemoteDebuggerPresent()`
- Wipe `PEB.BeingDebugged`
- Wipe `PEB.ProcessHeap`
- Wipe `PEB.NtGlobalFlag`
- Patch `ZwQueryInformationProcess()`

Try to follow and understand the code on your own and see just after for a brief discussion on it.

```
// -----
// IsDebuggerPresent patching routine
// -----
BOOL HideDebugger(HANDLE thread, HANDLE hproc)
{
    CONTEXT victimContext;

    // This function is used to patch the IsDebuggerPresent API
    // which might be called from debugged program (e.g. ASProtect) in order to detect debugger
    // presence. This function is mainly based on FS:[0] treating.
    // In an x86 environment, the FS register points to the current
    // value of the Thread Information Block (TIB) structure.
    // One element in the TIB structure is a pointer to an EXCEPTION_RECORD
    // structure, which in turn contains a pointer to an exception
    // handling callback function. Thus, each thread has its own exception callback function.
    // The x86 compiler builds exception-handling structures on the stack
    // as it processes functions. The FS register always points to the TIB,
    // which in turn contains a pointer to an EXCEPTION_RECORD structure.
    // The EXCEPTION_RECORD structure points to the exception handler function.

    // EXCEPTION_RECORD structures form a linked list: the new EXCEPTION_RECORD
    // structure contains a pointer to the previous EXCEPTION_RECORD structure,
    // and so on. On Intel-based machines, the head of the list is always pointed
    // to by the first DWORD in the thread information block, FS:[0]

    // 77E5276B > 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
    // 77E52771 8B40 30 MOV EAX,DWORD PTR DS:[EAX+30]
    // 77E52774 0FB640 02 MOVZX EAX,BYTE PTR DS:[EAX+2]
    // 77E52778 C3 RETN

    // Set up the victimContext access flag
```




Improving the HideDebugger function

```
victimContext.ContextFlags = CONTEXT_SEGMENTS;
// Fill the victim context structure with process data
if (!GetThreadContext(thread, &victimContext))
    return FALSE;

// GetThreadSelectorEntry is only functional on x86-based systems.
// For systems that are not x86-based, the function returns FALSE
// The GetThreadSelectorEntry function fills this structure with
// information from an entry in the descriptor table. You can use
// this information to convert a segment-relative address to a linear virtual address.
// The base address of a segment is the address of offset 0 in the segment.
// To calculate this value, combine the BaseLow, BaseMid, and BaseHi members

LDT_ENTRY sel;
if (!GetThreadSelectorEntry(thread, victimContext.SegFs, &sel))
    return FALSE;

//Gets the real address of FS
DWORD fsbase = (sel.HighWord.Bytes.BaseHi<<8 | sel.HighWord.Bytes.BaseMid)<<16 | sel.BaseLow;
DWORD RVApeb;
SIZE_T numread;

//Access PTR FS:[30], from that address takes the RVA PEB address.
//This works also for relocated PEBs (WINXP SP2)
if (!ReadProcessMemory(hproc, (LPVOID)(fsbase + 0x30), &RVApeb, 4, &numread) || numread != 4)
    return FALSE;

////////////////////////////////////
//PEB.BeingDebugged patch
WORD wBeingDebugged;
if (!ReadProcessMemory(hproc, (LPVOID)(RVApeb + 0x2), &wBeingDebugged, 2, &numread) ||
    numread != 2)
    return FALSE;

wBeingDebugged = 0;
if (!WriteProcessMemory(hproc, (LPVOID)(RVApeb + 0x2), &wBeingDebugged, 2, &numread) ||
    numread != 2)
    return FALSE;

////////////////////////////////////
//NtGlobalFlag patch

//Access the PEB+0x68 which is the NtGlobalFlag, is a ULONG
DWORD dwNtGlobalFlag;
if (!ReadProcessMemory(hproc, (LPVOID)(RVApeb + 0x68), &dwNtGlobalFlag, 4, &numread) ||
    numread != 4)
    return FALSE;

dwNtGlobalFlag=0;
if (!WriteProcessMemory(hproc, (LPVOID)(RVApeb + 0x68), &dwNtGlobalFlag, 4, &numread) ||
    numread != 4)
    return FALSE;

////////////////////////////////////
// ProcessHeap

//Access the PEB+0x18 which is the ProcessHeap, is a DWORD
DWORD dwProcessHeap;
if (!ReadProcessMemory(hproc, (LPVOID)(RVApeb + 0x18), &dwProcessHeap, 4, &numread) ||
    numread != 4)
    return FALSE;

DWORD dwForceFlags;
if (!ReadProcessMemory(hproc, (LPVOID)(dwProcessHeap + 0x10), &dwForceFlags, 4, &numread) ||
    numread != 4)
    return FALSE;

dwForceFlags =0;
if (!WriteProcessMemory(hproc, (LPVOID)(dwProcessHeap + 0x10), &dwForceFlags, 4, &numread) ||
    numread != 4)
    return FALSE;

////////////////////////////////////
////////////////////////////////////
//Patch ZwQueryInformationProcess
```



Improving the HideDebugger function

```
FARPROC addrIDP;
HINSTANCE hKer;
fcnzWQueryInformationProcess fcn;
DWORD dwOldProt=0;
hKer = GetModuleHandle("NTDLL");
addrIDP = GetProcAddress(hKer, "ZwQueryInformationProcess");
if (addrIDP==NULL)
    return FALSE;

//Convert function pointers to DWORD and calculate the difference among pointer..
int cbCodeSize=((LPBYTE) end_ZwQueryInformationProcessHook -
                (LPBYTE) start_ZwQueryInformationProcessHook);

//Change Page permissions so as to write the patch
dwOldProt=0;
fcnz=(fcnzWQueryInformationProcess)addrIDP;
//We must change permission for 5 bytes only, but the system changes the whole memory
//page containing that bytes
if(!VirtualProtectEx(hproc, addrIDP, 5, PAGE_EXECUTE_READWRITE, &dwOldProt))
    return FALSE;

//Allocates remotely the required memory
//the function determines where to allocate memory
LPVOID lpAllocatedMem = VirtualAllocEx( hproc, NULL, cbCodeSize + 5, MEM_COMMIT,
                                        PAGE_EXECUTE_READWRITE);

if(lpAllocatedMem==NULL)
    return FALSE;

//Writes the new function to the allocated memory.
if (!WriteProcessMemory(hproc, lpAllocatedMem, (LPVOID)&start_ZwQueryInformationProcessHook,
                        cbCodeSize, &numread) || numread != (SIZE_T)cbCodeSize)
    return FALSE;

//Patch the entry point of the ZwQueryInformationProcess to point to the new function.
//Code the JMP at the lpAllocatedMem
// JMP is coded as relative JMP less the instruction width which is 4 for DWORD of the
// JMP plus 1 for the OP-Code
// It's coded as: JMP destination_address - current_address - instruction_length(5 bytes)
DWORD jmp_length=((LPBYTE)lpAllocatedMem-(LPBYTE)addrIDP) - 5;
BYTE jmp_hook[5]={0xE9, 0x00, 0x00, 0x00, 0x00,};
jmp_hook[1]= (BYTE)(jmp_length);
jmp_hook[2]= (BYTE)(jmp_length >> 8);
jmp_hook[3]= (BYTE)(jmp_length >> 16);
jmp_hook[4]= (BYTE)(jmp_length >> 24);

//Writes the JMP to the new patch at the API entrypoint.
if (!WriteProcessMemory(hproc, (LPVOID)addrIDP, &jmp_hook, 5, &numread) || numread != 5)
    return FALSE;

//Restore Old page protection, some packer detect page protection changes..
VirtualProtectEx(hproc, addrIDP, 5, dwOldProt, NULL);

////////////////////////////////////
//CheckRemoteDebuggerPresent in another way because sometimes return TRUE!
// 7C859936 3945 08 CMP DWORD PTR SS:[EBP+8],EAX
// 7C859939 0F95C0 SETNE AL ; <-- NOP
// 7C85993C 8906 MOV DWORD PTR DS:[ESI],EAX
// 7C85993E 33C0 XOR EAX,EAX
// 7C859940 40 INC EAX ; <-- NOP
// 7C859941 EB 09 JMP SHORT kernel32.7C85994C
// 7C859943 6A 57 PUSH 57
// 7C859945 E8 76F9FAFF CALL kernel32.SetLastError
// 7C85994A 33C0 XOR EAX,EAX
// 7C85994C 5E POP ESI
// 7C85994D 5D POP EBP
// 7C85994E C2 0800 RETN 8
hKer = GetModuleHandle("KERNEL32");
addrIDP = GetProcAddress(hKer, "CheckRemoteDebuggerPresent");
BYTE patch_bytes1[]={0x90, 0x90, 0x90};
BYTE ori_bytes1[] = {0x0F, 0x95, 0xC0}; //SETNE AL
BYTE off_bytes1 = 0x37;
BOOL bApply_patch1=TRUE;

BYTE patch_bytes2[]={0x90};
BYTE ori_bytes2[] = {0x40}; //INC EAX
```



Improving the HideDebugger function

```
BYTE off_bytes2      =0x3E;
BOOL bApply_patch2=TRUE;

BYTE temp_bytes[3]; //stores temporary bytes read to compare with expected..
int idx=0;

if (addrIDP==NULL)
    return FALSE;

//We must change permission to containing page
VirtualProtectEx(hproc, addrIDP, 5, PAGE_EXECUTE_READWRITE, &dwOldProt);

//Patches the SETNE AL
if (!ReadProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes1),
    &temp_bytes, 3, &numread) || numread != 3)
    return FALSE;

for(idx=0; idx<3; idx++)
    if(temp_bytes[idx]!=ori_bytes1[idx])
        bApply_patch1=FALSE;

if(bApply_patch1)
    if (!WriteProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes1),
        &patch_bytes1, 3, &numread) || numread != 3)
        return FALSE;

//Patches the INC EAX
if (!ReadProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes2),
    &temp_bytes, 1, &numread) || numread != 1)
    return FALSE;

for(idx=0; idx<1; idx++)
    if(temp_bytes[idx]!=ori_bytes2[idx])
        bApply_patch2=FALSE;

if(bApply_patch2)
    if (!WriteProcessMemory(hproc, (LPVOID)((DWORD)addrIDP + (DWORD)off_bytes2),
        &patch_bytes2, 1, &numread) || numread != 1)
        return FALSE;

return TRUE;
}

////////////////////////////////////
//see http://msdn2.microsoft.com/en-US/library/h5wl0wxs.aspx for the naked attribute!
//naked tells to compiler to do not create prolog and epilog for this specific function, it's
perfect for ASM functions!
__declspec( naked ) void start_ZwQueryInformationProcessHook () {

    __asm {
        cmp [esp+8], 7
        je short no_zwqueryinformationprocess
        //executes the normal function as usual
        //the code comes from the original implementation of the API
        mov eax,9Ah
        mov edx,7FFE0300h
        call [edx]
        retn 14h

    no_zwqueryinformationprocess:
        mov eax,[esp+0ch]
        cmp eax,0           //if eax==0 then exit hook without doing anything
        je exit_hook
        mov [eax], 0
        push eax
        push ebx
        mov eax, [esp+18h] //points to how many bytes your API should read
        mov ebx, [esp+1Ch] //point to how many bytes the API read indeed
        cmp ebx, 0
        //if the last parameter is null, just skip it.
        //Otherwise emulate the function

        je exit_hook
        mov [ebx], eax      //these two values must be the same to behave like original

    exit_hook:
        pop ebx
        pop eax
    }
```



```
mov eax,0          //eax must be set to 0 meaning that all went fine
ret 14h

}

}

////////////////////////////////////
// This function is just used as a terminating reference for the memory copying of the
// above function to the victim's process's space. The naked attribute guarantees that there are
// no prologs and epilogs in the function, just like writing directly in ASM.
__declspec( naked ) void end_ZwQueryInformationProcessHook (void) { }
```

You also need some additional definitions coming from the Driver Development Kit (DDK) of Windows and already discussed before. I inserted them into the code here presented, just to not be obliged to install the whole DDK to compile this code.

```
typedef LONG NTSTATUS;

typedef enum _PROCESSINFOCLASS {
    ProcessBasicInformation,
    ProcessQuotaLimits,
    ProcessIoCounters,
    ProcessVmCounters,
    ProcessTimes,
    ProcessBasePriority,
    ProcessRaisePriority,
    ProcessDebugPort,
    ProcessExceptionPort,
    ProcessAccessToken,
    ProcessLdtInformation,
    ProcessLdtSize,
    ProcessDefaultHardErrorMode,
    ProcessIoPortHandlers,
    ProcessPooledUsageAndLimits,
    ProcessWorkingSetWatch,
    ProcessUserModeIOPL,
    ProcessEnableAlignmentFaultFixup,
    ProcessPriorityClass,
    ProcessWx86Information,
    ProcessHandleCount,
    ProcessAffinityMask,
    ProcessPriorityBoost,
    ProcessDeviceMap,
    ProcessSessionInformation,
    ProcessForegroundInformation,
    ProcessWow64Information,
    ProcessImageFileName,
    ProcessLUIDDeviceMapsEnabled,
    ProcessBreakOnTermination,
    ProcessDebugObjectHandle,
    ProcessDebugFlags,
    ProcessHandleTracing,
    MaxProcessInfoClass
} PROCESSINFOCLASS;

typedef NTSTATUS (NTAPI *fcntZwQueryInformationProcess)(HANDLE, PROCESSINFOCLASS, PVOID,
    ULONG, PULONG);
void start_ZwQueryInformationProcessHook();
void end_ZwQueryInformationProcessHook ();
```

6. Testing the approach

The result using the function I wrote can be tested using the xADT (eXtensible Anti-Debugger Tool) I released with ARTeam [3]. The results without and with the patches are shown in the Figure 15 and Figure 16⁴. The interesting thing is that some tests fails to understand if the program is debugged or not, the code might be improved but it's an initial result very interesting.

⁴ The xADT released version does not have the checkbox you can see in these snapshots, it's an "internal" release I use for testing purposes ;-)



Improving the HideDebugger function

Figure 17 shows the result reported by OllyDbg with all the known hiding plugins today available: Olly Advanced 1.26 beta 10 by Markus-DJM, Hide OllyDbg 1.01 by xDREAM, Hide Debugger 1.23f by Asterix.

Enable	TestName	Result	Status	Description of Test
<input type="checkbox"/>	IsDebuggerPresent()	Positive	Terminated	Test using IsDebuggerPresent
<input type="checkbox"/>	CheckRemoteDebuggerPresent()	Positive	Terminated	Test using CheckRemoteDebuggerPresent
<input type="checkbox"/>	PEB.BeingDebugged	Positive	Terminated	Controls PEB.BeingDebugged
<input type="checkbox"/>	PEB.ProcessHeap	Positive	Terminated	Controls PEB.ProcessHeap
<input type="checkbox"/>	GetProcessHeap()	Positive	Terminated	Controls PEB.ProcessHeap through GetProcessHeap API
<input type="checkbox"/>	PEB.NtGlobalFlag	Positive	Terminated	Controls PEB.NtGlobalFlag
<input type="checkbox"/>	PEB.NtGlobalFlag2	Positive	Terminated	Controls PEB.NtGlobalFlag via ZwQueryInformationProcess
<input type="checkbox"/>	Debug Registers	Negative	Terminated	Test if any of the Debug Registers is not 0
<input type="checkbox"/>	Single Step	Negative	Terminated	Test if single step bit in EFlags is set
<input type="checkbox"/>	CreateFileDrivers()	Negative	Terminated	Test some drivers using CreateFile
<input type="checkbox"/>	ZwQueryInformationProcess()	Positive	Terminated	Test using ZwQueryInformationProcess
<input type="checkbox"/>	ZwQueryInformationThread()	Unknown/Error	Terminated	Test using ZwQueryInformationThread
<input type="checkbox"/>	FindWindow OllyDbg	Positive	Terminated	Test using FindWindow OllyDbg
<input type="checkbox"/>	Invalid_Handle Exception Test	Positive	Terminated	Test looking if the Invalid_Handle Exception is caught or not
<input type="checkbox"/>	ParentProcess Test	Positive	Terminated	Test looking if the ParentProcess is a debugger
<input type="checkbox"/>	SIDT Test	Negative	Terminated	Test reading SIDT table: 0x01 Debug(Fault/Trap), 0x03 Breakpoint
<input type="checkbox"/>	UnhandledExceptionFilter	Negative	Terminated	Test using UnhandledExceptionFilter
<input type="checkbox"/>	ZwQueryObject DebugObject Test	Positive	Terminated	Test using the DebugObject read through ZwQueryObject

Test: IsDebuggerPresent()
Message from test function: Nothing
Result: Debugger detected

Test: CheckRemoteDebuggerPresent()
Message from test function: Nothing
Result: Debugger detected

Figure 15 – Debugged xADT report, without any patch

Enable	TestName	Result	Status	Description of Test
<input type="checkbox"/>	IsDebuggerPresent()	Negative	Terminated	Test using IsDebuggerPresent
<input type="checkbox"/>	CheckRemoteDebuggerPresent()	Negative	Terminated	Test using CheckRemoteDebuggerPresent
<input type="checkbox"/>	PEB.BeingDebugged	Negative	Terminated	Controls PEB.BeingDebugged
<input type="checkbox"/>	PEB.ProcessHeap	Negative	Terminated	Controls PEB.ProcessHeap
<input type="checkbox"/>	GetProcessHeap()	Negative	Terminated	Controls PEB.ProcessHeap through GetProcessHeap API
<input type="checkbox"/>	PEB.NtGlobalFlag	Negative	Terminated	Controls PEB.NtGlobalFlag
<input type="checkbox"/>	PEB.NtGlobalFlag2	Negative	Terminated	Controls PEB.NtGlobalFlag via ZwQueryInformationProcess
<input type="checkbox"/>	Debug Registers	Negative	Terminated	Test if any of the Debug Registers is not 0
<input type="checkbox"/>	Single Step	Negative	Terminated	Test if single step bit in EFlags is set
<input type="checkbox"/>	CreateFileDrivers()	Negative	Terminated	Test some drivers using CreateFile
<input type="checkbox"/>	ZwQueryInformationProcess()	Negative	Terminated	Test using ZwQueryInformationProcess
<input type="checkbox"/>	ZwQueryInformationThread()	Unknown/Error	Terminated	Test using ZwQueryInformationThread
<input type="checkbox"/>	FindWindow OllyDbg	Negative	Terminated	Test using FindWindow OllyDbg
<input type="checkbox"/>	Invalid_Handle Exception Test	Positive	Terminated	Test looking if the Invalid_Handle Exception is caught or not
<input type="checkbox"/>	ParentProcess Test	Positive	Terminated	Test looking if the ParentProcess is a debugger
<input type="checkbox"/>	SIDT Test	Unknown/Error	Terminated	Test reading SIDT table: 0x01 Debug(Fault/Trap), 0x03 Breakpoint(Trap), 0x05 Exception
<input type="checkbox"/>	UnhandledExceptionFilter	Negative	Terminated	Test using UnhandledExceptionFilter
<input type="checkbox"/>	ZwQueryObject DebugObject Test	Positive	Terminated	Test using the DebugObject read through ZwQueryObject

Test: IsDebuggerPresent()
Message from test function: Nothing
Result: I am NOT Debugged, Test went fine!

Test: CheckRemoteDebuggerPresent()
Message from test function: Nothing
Result: I am NOT Debugged, Test went fine!

Figure 16 - Debugged xADT with patches here discussed applied.

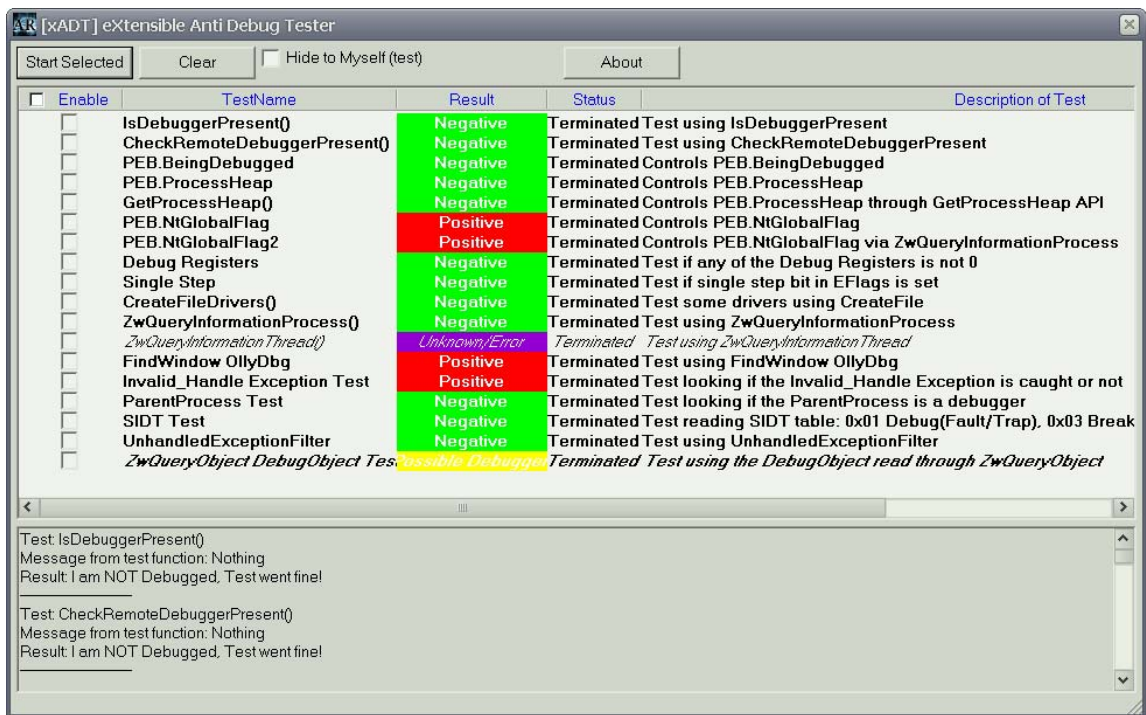


Figure 17 - Debugged xADT inside OllyDbg with all plugins enabled

7. References

[1] “Improving StraceNT:Adding Anti-Debugging Functionality”, Shub-Nigurrath, ARTeam, eZine No1. Vol.1 200, <http://ezine.accessroot.com/>

[2] “Anti-Anti-Dump and nonintrusive tracers”, deroko, ARTeam, <http://tutorials.accessroot.com>

[3] xADT eXtensible Anti-Debug Tester, Shub-Nigurrath, ARTeam, <http://releases.accessroot.com>

[4] “System Call Optimization with the SYSENTER Instruction”, CodeGuru, <http://www.codeguru.com/Cpp/W-P/system/devicedriverdevelopment/article.php/c8223/>

8. Conclusions

Weel this is the end of this document, I tried to introduce a lot of concepts in order to leave anyone being able to arrive at the conclusions. I hope to have been clear enough to let other start wondering on this subject to improve what I started to do. The conclusion of this tutorial is that there’s a lot of additional things to do and might be I’ll update this tutorial too when the function I use will become useless for specific targets.

I suggest as usual to use this tutorial for learning more in deep how the operative system works and to use these examples to evolve your RCE techniques and not to crack programs.

All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don’t use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.



9. History

- Version 1.0 – First public release!
- Version 1.1 – Improved the discussion on ZwQueryInformationProcess's patch

10. Greetings

I wish to tank all the ARTeam members of course and who read the beta versions of this tutorial and contributed,.. and of course you, who are still alive at the end of this quite long and complex document!



<http://cracking.accessroot.com>