

=====[Invisibility on NT boxes]
=====

How to become unseen on Windows NT

Author: Holy_Father <holy_father@phreaker.net>
Version: 1.2 english
Date: 05.08.2003

=====[1. Contents]
=====

1. Contents
2. Introduction
3. Files
 - 3.1 NtQueryDirectoryFile
 - 3.2 NtVdmControl
4. Processes
5. Registry
 - 5.1 NtEnumerateKey
 - 5.2 NtEnumerateValueKey
6. System services and drivers
7. Hooking and spreading
 - 7.1 Rights
 - 7.2 Global hook
 - 7.3 New processes
 - 7.4 DLL
8. Memory
9. Handle
 - 9.1 Naming handle and getting type
10. Ports
 - 10.1 Netstat, OpPorts on WinXP, FPort on WinXP
 - 10.2 OpPorts on Win2k and NT4, FPort on Win2k
11. Ending

=====[2. Introduction]
=====

This document is about technics of hiding objects, files, services, processes etc. on OS Windows NT. These methods are based on hooking Windows API

functions which are described in my document "Hooking Windows API".

Everything here was get from my own research during writing rootkit code, so there is a chance it can be written more effectively or it can be written much more easily. This also involve my implementation.

Hiding arbitrary object in this document mean to change some system functions which name this object in the way they would skip its naming. In the case this object is only return value of that function we would return value as the object does not exist.

Basic method (excluding cases of telling different) is that we would call original function with original arguments and then we would change its output.

In this version of this text are described methods of hiding files, processes, keys and values in registry, system services and drivers, allocated memory and handles.

=====[3. Files]

There are serveral possibilities of hiding files in the way OS would not see it. We would aim only changing API and leave out technics like those which play on features of filesystem. It also is much easier because we dont need to know how particular filesystem works.

=====[3.1 NtQueryDirectoryFile]

=====

Looking for a file on wNT in some directory is based on searching in all its files and files in all its subdirectories. For file enumeration is used function NtQueryDirectoryFile.

```
NTSTATUS NtQueryDirectoryFile(  
    IN HANDLE FileHandle,  
    IN HANDLE Event OPTIONAL,  
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,  
    IN PVOID ApcContext OPTIONAL,  
    OUT PIO_STATUS_BLOCK IoStatusBlock,  
    OUT PVOID FileInformation,  
    IN ULONG FileInformationLength,  
    IN FILE_INFORMATION_CLASS FileInformationClass,  
    IN BOOLEAN ReturnSingleEntry,  
    IN PUNICODE_STRING FileName OPTIONAL,  
    IN BOOLEAN RestartScan  
);
```

Important parameters for us are FileHandle, FileInformation and FileInformationClass. FileHandle is a handle of directory object which can be get from NtOpenFile. FileInformation is a pointer on allocated memory, where this function write wanted data to. FileInformationClass determines type of record written in FileInformation.

FileInformationClass is varied enumerative type, but we need only four values which are used for enumerating directory content:

```
#define FileDirectoryInformation 1  
#define FileFullDirectoryInformation 2  
#define FileBothDirectoryInformation 3  
#define FileNamesInformation 12
```

structure of recoed written in FileInformation for FileDirectoryInformation:

```
typedef struct _FILE_DIRECTORY_INFORMATION {
    ULONG NextEntryOffset;
    ULONG Unknown;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_DIRECTORY_INFORMATION, *PFILE_DIRECTORY_INFORMATION;
```

for FileFullDirectoryInformation:

```
typedef struct _FILE_FULL_DIRECTORY_INFORMATION {
    ULONG NextEntryOffset;
    ULONG Unknown;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaInformationLength;
    WCHAR FileName[1];
} FILE_FULL_DIRECTORY_INFORMATION,
*PFILE_FULL_DIRECTORY_INFORMATION;
```

for FileBothDirectoryInformation:

```
typedef struct _FILE_BOTH_DIRECTORY_INFORMATION {
    ULONG NextEntryOffset;
    ULONG Unknown;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
```

```
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
        ULONG FileAttributes;
        ULONG FileNameLength;
        ULONG EaInformationLength;
        UCHAR AlternateNameLength;
        WCHAR AlternateName[12];
        WCHAR FileName[1];
    } FILE_BOTH_DIRECTORY_INFORMATION,
*PFILE_BOTH_DIRECTORY_INFORMATION;
```

and for FileNamesInformation:

```
typedef struct _FILE_NAMES_INFORMATION {
    ULONG NextEntryOffset;
    ULONG Unknown;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_NAMES_INFORMATION, *PFILE_NAMES_INFORMATION;
```

This function writes a list of these structures in FileInformation.

Only three variables are important for us in any of these structure types.

NextEntryOffset is the length of particular list item. First item can be found on address FileInformation + 0. So the second item is on address FileInformation + NextEntryOffset of first one. Last item has NextEntryOffset set on zero.

FileName is a full name of the file.

FileNameLength is a length of file name.

If we want to hide a file, we need to tell apart these four types and for each returned record we need to compare its name with the one which we want to hide. If we want to hide first record, we have to move following

structures by the size of the first. This will cause the first record would be rewritten. If we want to hide another record, we can easily change the value of NextEntryOffset of previous record. New value of NextEntryOffset would be zero if we want to hide the last record, otherwise the value would be the sum of NextEntryOffset of the record we want to hide and of previous record.

Then we should change the value of Unknown of previous record which is proolly an index for next search. The value of Unknown of previous record should have a value of Unknown of the record we want hide.

If no record which should be seen was found, we will return error STATUS_NO_SUCH_FILE.

```
#define STATUS_NO_SUCH_FILE 0xC000000F
```

```
=====[ 3.2 NtVdmControl ]
```

```
=====
```

From unknown reason DOS emulation NTVDM can get a list of files also with function NtVdmContol.

```
NTSTATUS NtVdmControl(  
    IN ULONG ControlCode,  
    IN PVOID ControlData  
);
```

ControlCode specifies the subfunction which is applied on data in ControlData buffer. If ControlCode equals to VdmDirectoryFile this function does the same as NtQueryDirectoryFile with FileInformationClass set on FileBothDirectoryInformation.

```
#define VdmDirectoryFile 6
```

Then ControlData is used like FileInformation. The only difference here is that we do not know the length of this buffer. So we have to count it manually. We have to add NextEntryOffset of all records and FileNameLength of the last record and 0x5E as a length of the last record excluding the name of the file. Hiding methods are the same as in NtQueryDirectoryFile then.

====[4. Processes]

=====

Various system info is available using NtQuerySystemInformation.

```
NTSTATUS NtQuerySystemInformation(  
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,  
    IN OUT PVOID SystemInformation,  
    IN ULONG SystemInformationLength,  
    OUT PULONG ReturnLength OPTIONAL  
);
```

SystemInformationClass specifies the type of information which we want to get, SystemInformation is a pointer to the function output buffer, SystemInformationLength is the length of this buffer and ReturnLength is number of written bytes.

For the enumeration of running processes we use SystemInformationClass set on SystemProcessesAndThreadsInformation.

```
#define SystemInformationClass 5
```

Returned structure in SystemInformation buffer is:

```
typedef struct _SYSTEM_PROCESSES {
    ULONG NextEntryDelta;
    ULONG ThreadCount;
    ULONG Reserved1[6];
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER UserTime;
    LARGE_INTEGER KernelTime;
    UNICODE_STRING ProcessName;
    KPRIORITY BasePriority;
    ULONG ProcessId;
    ULONG InheritedFromProcessId;
    ULONG HandleCount;
    ULONG Reserved2[2];
    VM_COUNTERS VmCounters;
    IO_COUNTERS IoCounters; // Windows 2000 only
    SYSTEM_THREADS Threads[1];
} SYSTEM_PROCESSES, *PSYSTEM_PROCESSES;
```

Hiding processes is similiar as in the case of hiding files. We have to change NextEntryDelta of previous record of that we want to hide. Usually we will not want to hide the first record here because it is Idle process.

=====[5. Registry]

Windows registry is quite big tree structure containing two important types of records for us which we could want to hide. First type is registry keys, second is values. Owing to registry structure hiding registry keys is not as trivial as hiding file or process.

=====[5.1 NtEnumerateKey]

Owing to its structure we are not able to ask for a list of all keys in the specific part of registry. We can get only information about one key specified by its index in some part of registry. This provides NtEnumerateKey.

```
NTSTATUS NtEnumerateKey(  
    IN HANDLE KeyHandle,  
    IN ULONG Index,  
    IN KEY_INFORMATION_CLASS KeyInformationClass,  
    OUT PVOID KeyInformation,  
    IN ULONG KeyInformationLength,  
    OUT PULONG ResultLength  
);
```

KeyHandle is a handle to a key in which we want to get information about a subkey specified by Index. Type of returned information is specified by KeyInformationClass. Data are written to KeyInformation buffer which length is KeyInformationLength. Number of written bytes is returned in ResultLength.

The most important think we need to perceive is that if we hide a key, indexes of all following keys would be shifted. And because we are able to get information about a key with higher index with asking for key with lower index we always have to count how many records before were hidden and then return the right one.

Let's have a look on the example. Assume we have some keys called A, B, C, D, E and F in any part of registry. Indexing starts from zero which mean index 4 match E key. Now if we want to hide B key and the hooked application call NtEnumerateKey with Index 4 we should return information about F key because there is an index shift. The problem is that we don't know

that there is a shift. And if we didn't care about shifting and return E instead of F when asking for key with index 4 we would return nothing when asking for key with index 1 or we would return C. Both cases are errors. This is why we have to care about shifting.

Now if we counted the shift by recalling the function for each index from 0 to Index we would sometimes wait for ages (on 1GHz processor it could take up to 10 seconds with standard registry which is too much). So we have to think out more sophisticated method.

We know that keys are (except of references) sorted alphabetically. If we neglect references (which we don't want to hide) we can count the shift by following method. We will sort alphabetically our list of key names which we want to hide (RtlCompareUnicodeString can be used), then when application calls NtEnumerateKey we will not recall it with unchanged arguments but we will find out the name of the record specified by Index.

```
NTSTATUS RtlCompareUnicodeString(  
    IN PUNICODE_STRING String1,  
    IN PUNICODE_STRING String2,  
    IN BOOLEAN CaseInsensitive  
);
```

String1 and String2 are strings which will be compared, CaseInsensitive is True if we want to compare with neglecting character case.

Function result describes relation between String1 and String2:

```
result > 0:    String1 > String2  
result = 0:    String1 = String2  
result < 0:    String1 < String2
```

Now we have to find a border. We will compare alphabetically the name of the key specified by Index with the names in our list. The border would be the last lesser name from our list. We know that the shift is at most the number of the border in our list. But not all items from our list have to be a valid key in the part of registry we are in. So we have to ask for all items from our list up to border if they are in this part of the registry. This can be done using NtOpenKey.

```
NTSTATUS NtOpenKey(  
    OUT PHANDLE KeyHandle,  
    IN ACCESS_MASK DesiredAccess,  
    IN POBJECT_ATTRIBUTES ObjectAttributes  
);
```

KeyHandle is a handle of superordinate key. We will use the value from NtEnumerateKey for it. DesiredAccess are access rights. KEY_ENUMERATE_SUB_KEYS is the right value for it. ObjectAttributes describes subkey which we want to open (including its name).

```
#define KEY_ENUMERATE_SUB_KEYS 8
```

If the result of NtOpenKey is 0 opening was successful which mean this key from our list exists. Opened key should be closed via NtClose.

```
NTSTATUS NtClose(  
    IN HANDLE Handle  
);
```

For each call of NtEnumerateKey we will count the shift as a number of keys from our list which exist in the given part of registry. Then we will add

this shift to Index argument and finally call the original NtEnumerateKey.

For getting name of the key specified by Index we will use the value KeyBasicInformation as a KeyInformationClass.

```
#define KeyBasicInformation 0
```

NtEnumerateKey returns this structure in KeyInformation:

```
typedef struct _KEY_BASIC_INFORMATION {
    LARGE_INTEGER LastWriteTime;
    ULONG TitleIndex;
    ULONG NameLength;
    WCHAR Name[1];
} KEY_BASIC_INFORMATION, *PKEY_BASIC_INFORMATION;
```

Only thing we need here is Name and its length NameLength.

If there is no entry for shifted Index we will return error STATUS_EA_LIST_INCONSISTENT.

```
#define STATUS_EA_LIST_INCONSISTENT 0x80000014
```

```
=====[ 5.2 NtEnumerateValueKey ]
```

```
=====
```

Registry values are not alphabetically sorted. Luckily the number of values in one key is quite small, so we can use recall method to get the shift. API for getting info about one value is called NtEnumerateValueKey.

```
NTSTATUS NtEnumerateValueKey(
    IN HANDLE KeyHandle,
    IN ULONG Index,
    IN KEY_VALUE_INFORMATION_CLASS
KeyValueInformationClass,
    OUT PVOID KeyValueInformation,
    IN ULONG KeyValueInformationLength,
    OUT PULONG ResultLength
);
```

KeyHandle is again a handle of superordinate key. Index is an index to the list of values in given key. KeyValueInformationClass describes a type of information which will be stored into KeyValueInformation buffer which is long KeyValueInformationLength bytes. Number of written bytes is returned in ResultLength.

Again we have to count the shift but according to the number of values in one key we can recall this function for all indexes from 0 to Index. The name of the value can be get when KeyValueInformationClass is set to KeyValueBasicInformation.

```
#define KeyValueBasicInformation 0
```

Then we will get following structure in KeyValueInformation buffer:

```
typedef struct _KEY_VALUE_BASIC_INFORMATION {
    ULONG TitleIndex;
    ULONG Type;
    ULONG NameLength;
    WCHAR Name[1];
} KEY_VALUE_BASIC_INFORMATION, *PKEY_VALUE_BASIC_INFORMATION;
```

Again we are interested only in Name and NameLength.

If there is no entry for shifted Index we will return error STATUS_NO_MORE_ENTRIES.

```
#define STATUS_NO_MORE_ENTRIES 0x8000001A
```

```
=====[ 6. System services and drivers ]
=====
```

System services and drivers are enumerated by four independent API functions. Their connections is different in each Windows version. That's why we have to hook all four functions.

```
BOOL EnumServicesStatusA(  
    SC_HANDLE hSCManager,  
    DWORD dwServiceType,  
    DWORD dwServiceState,  
    LPENUM_SERVICE_STATUS lpServices,  
    DWORD cbBufSize,  
    LPDWORD pcbBytesNeeded,  
    LPDWORD lpServicesReturned,  
    LPDWORD lpResumeHandle  
);
```

```
BOOL EnumServiceGroupW(  
    SC_HANDLE hSCManager,  
    DWORD dwServiceType,  
    DWORD dwServiceState,  
    LPBYTE lpServices,  
    DWORD cbBufSize,  
    LPDWORD pcbBytesNeeded,  
    LPDWORD lpServicesReturned,  
    LPDWORD lpResumeHandle,  
    DWORD dwUnknown  
);
```

```
BOOL EnumServicesStatusExA(  
    SC_HANDLE hSCManager,  
    SC_ENUM_TYPE InfoLevel,  
    DWORD dwServiceType,  
    DWORD dwServiceState,  
    LPBYTE lpServices,  
    DWORD cbBufSize,  
    LPDWORD pcbBytesNeeded,  
    LPDWORD lpServicesReturned,  
    LPDWORD lpResumeHandle,  
    LPCTSTR pszGroupName  
);
```

```
BOOL EnumServicesStatusExW(  
    SC_HANDLE hSCManager,  
    SC_ENUM_TYPE InfoLevel,  
    DWORD dwServiceType,  
    DWORD dwServiceState,  
    LPBYTE lpServices,  
    DWORD cbBufSize,  
    LPDWORD pcbBytesNeeded,  
    LPDWORD lpServicesReturned,  
    LPDWORD lpResumeHandle,  
    LPCTSTR pszGroupName  
);
```

The most important here is `lpServices` which points on the buffer where the list of services would be stored. And also `lpServicesReturned` pointing on the number of records in result is important. Structure of data in the output buffer depends on the type of function. For functions `EnumServicesStatusA` and `EnumServicesGroupW` is returned structure

```
typedef struct _ENUM_SERVICE_STATUS {  
    LPTSTR lpServiceName;  
    LPTSTR lpDisplayName;  
    SERVICE_STATUS ServiceStatus;  
} ENUM_SERVICE_STATUS, *LPENUM_SERVICE_STATUS;
```

```
typedef struct _SERVICE_STATUS {  
    DWORD dwServiceType;  
    DWORD dwCurrentState;  
    DWORD dwControlsAccepted;  
    DWORD dwWin32ExitCode;  
    DWORD dwServiceSpecificExitCode;  
    DWORD dwCheckPoint;  
    DWORD dwWaitHint;  
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

for `EnumServicesStatusExA` a `EnumServicesStatusExW` it it

```
typedef struct _ENUM_SERVICE_STATUS_PROCESS {
```

```
        LPTSTR lpServiceName;
        LPTSTR lpDisplayName;
        SERVICE_STATUS_PROCESS ServiceStatusProcess;
} ENUM_SERVICE_STATUS_PROCESS, *LPENUM_SERVICE_STATUS_PROCESS;

typedef struct _SERVICE_STATUS_PROCESS {
    DWORD dwServiceType;
    DWORD dwCurrentState;
    DWORD dwControlsAccepted;
    DWORD dwWin32ExitCode;
    DWORD dwServiceSpecificExitCode;
    DWORD dwCheckPoint;
    DWORD dwWaitHint;
    DWORD dwProcessId;
    DWORD dwServiceFlags;
} SERVICE_STATUS_PROCESS, *LPSERVICE_STATUS_PROCESS;
```

We are interested only in lpServiceName which is the name of system service. Records have static size, so if we want to hide one we will move all following records by its size. Here we have to differentiate between the size of SERVICE_STATUS and SERVICE_STATUS_PROCESS.

====[7. Hooking and spreading]

=====

To get the desiderative efect we have to hook all running processes and also all processes which would be created later. New processes should be hooked before running their first instruction of their own code otherwise they would be able to see our hidden objects in the time before they would be hooked.

====[7.1 Rights]

=====

At first it is good to know that we need at least administrators rights to get access to all running processes. The best possibility is to run our process as system service which run on user SYSTEM. To install the service we also need special rights.

Also getting SeDebugPrivilege is very useful. This can be done using API OpenProcessToken, LookupPrivilegeValue and AdjustTokenPrivileges.

```
BOOL OpenProcessToken(  
    HANDLE ProcessHandle,  
    DWORD DesiredAccess,  
    PHANDLE TokenHandle  
);  
  
BOOL LookupPrivilegeValue(  
    LPCTSTR lpSystemName,  
    LPCTSTR lpName,  
    PLUID lpLuid  
);  
  
BOOL AdjustTokenPrivileges(  
    HANDLE TokenHandle,  
    BOOL DisableAllPrivileges,  
    PTOKEN_PRIVILEGES NewState,  
    DWORD BufferLength,  
    PTOKEN_PRIVILEGES PreviousState,  
    PDWORD ReturnLength  
);
```

Neglecting errors the code can look like this:

```
#define SE_PRIVILEGE_ENABLED    0x0002  
#define TOKEN_QUERY            0x0008  
#define TOKEN_ADJUST_PRIVILEGES 0x0020  
  
HANDLE hToken;  
LUID DebugNameValue;
```

```
TOKEN_PRIVILEGES Privileges;  
DWORD dwRet;
```

```
OpenProcessToken(GetCurrentProcess(),  
                TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,  
hToken);
```

```
LookupPrivilegeValue(NULL, "SeDebugPrivilege", &DebugNameValue);  
Privileges.PrivilegeCount=1;  
Privileges.Privileges[0].Luid=DebugNameValue;  
Privileges.Privileges[0].Attributes=SE_PRIVILEGE_ENABLED;  
AdjustTokenPrivileges(hToken, FALSE, &Privileges, sizeof  
(Privileges),  
                    NULL, &dwRet);
```

```
CloseHandle(hToken);
```

=====[7.2 Global hook]

=====

Enumeration of processes is done by already mentioned API function NtQuerySystemInformation. There are few native processes in the system, so we will use the method of rewriting first instructions of the function to hook them. For each running process we will do the same. We will allocate a part of memory in target process where we will write our new code for functions we want to hook. Then we will change the first five bytes of these functions with jmp instruction. This jmp will redirect the execution to our code. So the jmp instruction will be executed immediately when the hooked function is called. We have to save first instructions of each function which is rewritten. We need them to call original code of the hooked function. Saving instructions is described in chapter 3.2.3 in the document "Hooking Windows API". At first we have to open target process via NtOpenProcess and get the handle. This will fail if we don't have enough rights.

```
NTSTATUS NtOpenProcess(  
    OUT PHANDLE ProcessHandle,  
    IN ACCESS_MASK DesiredAccess,  
    IN POBJECT_ATTRIBUTES ObjectAttributes,  
    IN PCLIENT_ID ClientId OPTIONAL  
);
```

ProcessHandle is a pointer on a handle where the result will be stored.
DesiredAccess should be set on PROCESS_ALL_ACCESS. We will set PID of target process to UniqueProcess part of ClientId structure, UniqueThread should be 0.
Open handle can be always closed via NtClose.

```
#define PROCESS_ALL_ACCESS 0x001F0FFF
```

Now we are going to allocate the part of memory for our code. This can be done using NtAllocateVirtualMemory.

```
NTSTATUS NtAllocateVirtualMemory(  
    IN HANDLE ProcessHandle,  
    IN OUT PVOID BaseAddress,  
    IN ULONG ZeroBits,  
    IN OUT PULONG AllocationSize,  
    IN ULONG AllocationType,  
    IN ULONG Protect  
);
```

ProcessHandle is the one from NtOpenProcess. BaseAddress is a pointer on a pointer on the beginning where we want to allocate. Here will be stored the address of the allocated memory. Input value can be NULL.
AllocationSize is a pointer on number of bytes we want to allocate. And again it is also used as output value for the real number of allocated bytes. It is good to set AllocationType to MEM_TOP_DOWN in addition to MEM_COMMIT because the memory

would be allocated on the highest possible address near DLLs.

```
#define MEM_COMMIT      0x00001000
#define MEM_TOP_DOWN   0x00100000
```

Then we can write our code there using `NtWriteVirtualMemory`.

```
NTSTATUS NtWriteVirtualMemory(
    IN HANDLE ProcessHandle,
    IN PVOID BaseAddress,
    IN PVOID Buffer,
    IN ULONG BufferLength,
    OUT PULONG ReturnLength OPTIONAL
);
```

`BaseAddress` will be that address returned by `NtAllocateVirtualMemory`.
`Buffer` points on bytes we want to write, `BufferLength` is number of bytes we want to write.

Now we have to hook single functions. Only library which is loaded to all processes is `ntdll.dll`. So we have to check if function we want to hook is imported to the process if it is not from `ntdll.dll`. But the memory where would this function (from another DLL) be could be allocated, so rewriting bytes on its address could easily cause error in target process. This is why we have to check whether library (where function we want to hook is) is loaded to target process.

We need to get PEB (Process Environment Block) of target process via `NtQueryInformationProcess`.

```
NTSTATUS NtQueryInformationProcess(
    IN HANDLE ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID ProcessInformation,
```

```
        IN ULONG ProcessInformationLength,  
        OUT PULONG ReturnLength OPTIONAL  
    );
```

We will set ProcessInformationClass to ProcessBasicInformation. Then the PROCESS_BASIC_INFORMATION structure would be returned to ProcessInformation buffer which size is given by ProcessInformationLength.

```
#define ProcessBasicInformation 0  
  
typedef struct _PROCESS_BASIC_INFORMATION {  
    NTSTATUS ExitStatus;  
    PPEB PebBaseAddress;  
    KAFFINITY AffinityMask;  
    KPRIORITY BasePriority;  
    ULONG UniqueProcessId;  
    ULONG InheritedFromUniqueProcessId;  
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;
```

PebBaseAddress is what we were looking for. On PebBaseAddress +0x0C is address PPEB_LDR_DATA. This would be get calling NtReadVirtualMemory.

```
NTSTATUS NtReadVirtualMemory(  
    IN HANDLE ProcessHandle,  
    IN PVOID BaseAddress,  
    OUT PVOID Buffer,  
    IN ULONG BufferLength,  
    OUT PULONG ReturnLength OPTIONAL  
);
```

Parameters are similar like in NtWriteVirtualMemory.

On PPEB_LDR_DATA+0x1C is address InInitializationOrderModuleList. It is the list of libraries loaded to the process. We are interested only in a part of this structure.

```
typedef struct _IN_INITIALIZATION_ORDER_MODULE_LIST {  
    PVOID Next,  
    PVOID Prev,
```

```
        DWORD ImageBase,  
        DWORD ImageEntry,  
        DWORD ImageSize,  
        ...  
    );
```

Next is a pointer on next record, Prev on previous, last record points on first. ImageBase is an address of module in the memory, ImageEntry is the EntryPoint of the module, ImageSize is its size.

For all libraries in which we want to hook we will get their ImageBase (e.g. using GetModuleHandle or LoadLibrary). This ImageBase we will compare with ImageBase of each entry in InInitializationOrderModuleList.

Now we are ready for hooking. Because we are hooking running processes there is a possibility that the code we would be executed in the moment we will be rewriting it. This can cause error, so at first we will stop all threads in target process. The list of its threads can get via NtQuerySystemInformation with SystemProcessesAndThreadsInformation class. Result of this function is described in chapter 4. But we have to add the description of SYSTEM_THREADS structure where the information about thread is.

```
typedef struct _SYSTEM_THREADS {  
    LARGE_INTEGER KernelTime;  
    LARGE_INTEGER UserTime;  
    LARGE_INTEGER CreateTime;  
    ULONG WaitTime;  
    PVOID StartAddress;  
    CLIENT_ID ClientId;  
    KPRIORITY Priority;  
    KPRIORITY BasePriority;  
    ULONG ContextSwitchCount;  
    THREAD_STATE State;  
    KWAIT_REASON WaitReason;
```

```
} SYSTEM_THREADS, *PSYSTEM_THREADS;
```

For each thread we have to get its handle using NtOpenThread.
We will use ClientId for it.

```
NTSTATUS NtOpenThread(  
    OUT PHANDLE ThreadHandle,  
    IN ACCESS_MASK DesiredAccess,  
    IN POBJECT_ATTRIBUTES ObjectAttributes,  
    IN PCLIENT_ID ClientId  
);
```

The handle we want will be stored to ThreadHandle. We will set DesiredAccess to THREAD_SUSPEND_RESUME.

```
#define THREAD_SUSPEND_RESUME 2
```

ThreadHandle will be used for calling NtSuspendThread.

```
NTSTATUS NtSuspendThread(  
    IN HANDLE ThreadHandle,  
    OUT PULONG PreviousSuspendCount OPTIONAL  
);
```

Suspended process is ready for rewriting. We will proceed as it is described in chapter 3.2.2 in "Hooking Windows API". Only difference will be in using functions for other processes.

After a hook we will revive all process threads calling NtResumeThread.

```
NTSTATUS NtResumeThread(  
    IN HANDLE ThreadHandle,  
    OUT PULONG PreviousSuspendCount OPTIONAL  
);
```

====[7.3 New processes]

=====

Infection of all running processes does not affect processes which would be run later. We could get the process list and after a while get a new one and compare them and then infect those processes which are in second list but not in first. But this method is very unreliable.

Much better is to hook function which is always called when new process starts. Because of hooking all running processes on the system we can't miss any new with this method. We can hook NtCreateThread but it is not the easiest way. We will hook NtResumeThread which is also called everytime after the new process is created. It is called after NtCreateThread.

The only problem with NtResumeThread is that it is called not only when new process starts. But we can easily get over this. NtQueryInformationThread will give us an information about which process owns the specific thread. The last thing we have to do is to check whether this process is already hooked or not. This can be done by reading first byte of any function we are hooking.

```
NTSTATUS NtQueryInformationThread(  
    IN HANDLE ThreadHandle,  
    IN THREADINFOCLASS ThreadInformationClass,  
    OUT PVOID ThreadInformation,  
    IN ULONG ThreadInformationLength,  
    OUT PULONG ReturnLength OPTIONAL  
);
```

ThreadInformationClass is information class and it should be set in our case to ThreadBasicInformation. ThreadInformation is the buffer for result which size is ThreadInformationLength bytes.

```
#define ThreadBasicInformation 0
```

For class ThreadBasicInformation is this structure returned:

```
typedef struct _THREAD_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PNT_TIB TebBaseAddress;
    CLIENT_ID ClientId;
    KAFFINITY AffinityMask;
    KPRIORITY Priority;
    KPRIORITY BasePriority;
} THREAD_BASIC_INFORMATION, *PTHREAD_BASIC_INFORMATION;
```

In ClientId is the PID of which owns the thread.

Now we have to infect the new process. The problem is that the new process has only ntdll.dll in its memory. All others modules are loaded immediately after calling NtResumeThread. There are several ways how to handle this problem. E.g. we can hook API called LdrInitializeThunk which is called during process init.

```
NTSTATUS LdrInitializeThunk(
    DWORD Unknown1,
    DWORD Unknown2,
    DWORD Unknown3
);
```

At first we will run original code and then we will hook all functions we want in this new process. But it will be better to unhook LdrInitializeThunk because it is called many times later and we don't want to rehook all functions again. Everything here is done before execution of the first instruction of hooked application. That's why there is no chance it would call any of hooked functions before we hook it.

The hooking in itself is the same as when hooking running

process
but here we don't care about running threads.

=====[7.4 DLL]

=====

In each process in the system is the copy of ntdll.dll. That mean we can hook any function from this module in the process init. But how about functions from other modules like kernel32.dll or advapi32.dll? And there are also several processes which has only ntdll.dll. All other modules can be loaded dynamically in the middle of the code after the process hook. That's why we have to hook LdrLoadDll which loades new modules.

```
NTSTATUS LdrLoadDll(  
    PWSTR szcPath,  
    PDWORD pdwLdrErr,  
    PUNICODE_STRING pUniModuleName,  
    PHINSTANCE pResultInstance  
);
```

The most important for us here is pUniModuleName which is the name of the module. pResultInstance will be filled with its address if the call is successful.

We will call original LdrLoadDll and then hook all functions in loaded module.

=====[8. Memory]

=====

When we are hooking a function we modify its first bytes. Via calling NtReadVirtualMemory anyone can detect that a function is hooked. So

we have to
hook NtReadVirtualMemory to prevent detecting.

```
NTSTATUS NtReadVirtualMemory(  
    IN HANDLE ProcessHandle,  
    IN PVOID BaseAddress,  
    OUT PVOID Buffer,  
    IN ULONG BufferLength,  
    OUT PULONG ReturnLength OPTIONAL  
);
```

We have changed bytes on the beginning of all functions we hooked and we have also allocated memory for our new code. We should check whether caller reads some of these bytes. If we have our bytes in the range from BaseAddress to BaseAddress + BufferLength we have to change some bytes in Buffer.

If one ask for bytes from our allocated memory we should return empty Buffer and an error STATUS_PARTIAL_COPY. This value says not all requested bytes were copied to the Buffer. It is also used when asking for unallocated memory. ReturnLength should be set to 0 in this case.

```
#define STATUS_PARTIAL_COPY 0x8000000D
```

If one ask for first bytes of hooked function we have to call original code and than we should copy original bytes (we have saved them for original calls) to Buffer.

Now the process is not able to detect he is hooked via reading its memory. Also if you debug hooked process debugger will have a problem. It will show original bytes but it will execute our code.

To make hiding perfect we can also hook NtQueryVirtualMemory. This function is used to get information about virtual memory. We can hook it to

prevent detecting our allocated memory.

```
NTSTATUS NtQueryVirtualMemory(  
    IN HANDLE ProcessHandle,  
    IN PVOID BaseAddress,  
    IN MEMORY_INFORMATION_CLASS MemoryInformationClass,  
    OUT PVOID MemoryInformation,  
    IN ULONG MemoryInformationLength,  
    OUT PULONG ReturnLength OPTIONAL  
);
```

MemoryInformationClass specifies the class of data which are returned.

First two types are interesting for us.

```
#define MemoryBasicInformation 0  
#define MemoryWorkingSetList 1
```

For class MemoryBasicInformation is returned this structure:

```
typedef struct _MEMORY_BASIC_INFORMATION {  
    PVOID BaseAddress;  
    PVOID AllocationBase;  
    ULONG AllocationProtect;  
    ULONG RegionSize;  
    ULONG State;  
    ULONG Protect;  
    ULONG Type;  
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Each memory section has its size RegionSize and its type Type. Free memory has type MEM_FREE.

```
#define MEM_FREE 0x10000
```

If a section before ours has type MEM_FREE we should add the size of our section to its RegionSize. If the following section is also MEM_FREE we should add following section size again that RegionSize.

If a section before ours has another type we will return MEM_FREE

for our section. Its size is counted again according to following section.

For class MemoryWorkingSetList is returned structure:

```
typedef struct _MEMORY_WORKING_SET_LIST {
    ULONG NumberOfPages;
    ULONG WorkingSetList[1];
} MEMORY_WORKING_SET_LIST, *PMEMORY_WORKING_SET_LIST;
```

NumberOfPages is the number of items in WorkingSetList. This number should be decreased. We should find ours section in WorkingSetList and move following records over ours. WorkingSetList is an array of DWORDs where higher 20 bits specifies higher 20 bits of section address and lower 12 bits specifies flags.

=====[9. Handle]

=====

Calling NtQuerySystemInformation with SystemHandleInformation class gives us array of all open handles in _SYSTEM_HANDLE_INFORMATION_EX strucure.

```
#define SystemHandleInformation 0x10

typedef struct _SYSTEM_HANDLE_INFORMATION {
    ULONG ProcessId;
    UCHAR ObjectTypeNumber;
    UCHAR Flags;
    USHORT Handle;
    PVOID Object;
    ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

typedef struct _SYSTEM_HANDLE_INFORMATION_EX {
    ULONG NumberOfHandles;
```

```
        SYSTEM_HANDLE_INFORMATION Information[1];
    } SYSTEM_HANDLE_INFORMATION_EX,
*PSYSTEM_HANDLE_INFORMATION_EX;
```

ProcessId specifies the process which owns the handle.

ObjectTypeNumber

is handle type. NumberOfHandles is number of records in Information array.

Hiding one item is trivial. We have to remove all following records by one

and decrease NumberOfHandles. Removing all following is needed because handles

in array are grouped by ProcessId. That mean all handles from one single

process are together. And for one process the number Handle is growing.

Now remember structure _SYSTEM_PROCESSES which is returned by this

function with SystemProcessesAndThreadsInformation class. Here we can see that

each process has an information about its number of handles in HandleCount.

If we want to be perfect we should modify HandleCount owing to how many handles

we hide when calling this function with

SystemProcessesAndThreadsInformation

class. But this correction would be very time-consuming. There are many handles

opening and closing in very short time during normal system running.

So it can

easily happend that number of handles is changed in between two calls of this

function and we don't need to change HandleCount.

====[9.1 Naming handle and getting type]

=====

Handle hiding is trivial but find out which handle to hide is

little

bit harder. If we have e.g. hidden process we should hide all its handles and

all handles which are connected with it. Hiding handles of this

process is again trivial. We are only comparing ProcessId of handle and PID of our process and when they equals we hide it. But handles of other processes have to be named before we can compare something. The number of handles in the system is usually very big, so the best we can do is to compare handle type first before trying to name it. Naming types will save a lot of time for handles we are not interested in.

Naming handle and handle type can be done via calling NtQueryObject.

```
NTSTATUS ZwQueryObject(  
    IN HANDLE ObjectHandle,  
    IN OBJECT_INFORMATION_CLASS ObjectInformationClass,  
    OUT PVOID ObjectInformation,  
    IN ULONG ObjectInformationLength,  
    OUT PULONG ReturnLength OPTIONAL  
);
```

ObjectHandle is a handle we want to get info about, ObjectInformationClass is the type of information which will be stored into ObjectInformation buffer which is ObjectInformationLength bytes long.

We will use class ObjectNameInformation and ObjectAllTypesInformation. ObjectNameInformation class will fill the buffer with OBJECT_NAME_INFORMATION structure, ObjectAllTypesInformation class with OBJECT_ALL_TYPES_INFORMATION structure then.

```
#define ObjectNameInformation 1  
#define ObjectAllTypesInformation 3  
  
typedef struct _OBJECT_NAME_INFORMATION {  
    UNICODE_STRING Name;  
} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;
```

Name determines the name of the handle.

```
typedef struct _OBJECT_TYPE_INFORMATION {
    UNICODE_STRING Name;
    ULONG ObjectCount;
    ULONG HandleCount;
    ULONG Reserved1[4];
    ULONG PeakObjectCount;
    ULONG PeakHandleCount;
    ULONG Reserved2[4];
    ULONG InvalidAttributes;
    GENERIC_MAPPING GenericMapping;
    ULONG ValidAccess;
    UCHAR Unknown;
    BOOLEAN MaintainHandleDatabase;
    POOL_TYPE PoolType;
    ULONG PagedPoolUsage;
    ULONG NonPagedPoolUsage;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;
```

```
typedef struct _OBJECT_ALL_TYPES_INFORMATION {
    ULONG NumberOfTypes;
    OBJECT_TYPE_INFORMATION TypeInformation;
} OBJECT_ALL_TYPES_INFORMATION,
```

```
*POBJECT_ALL_TYPES_INFORMATION;
```

Name determines the name of type object which immediately follows each OBJECT_TYPE_INFORMATION structure. The next OBJECT_TYPE_INFORMATION structure follows this Name, starting on the first four-byte boundary.

ObjectTypeNumber from SYSTEM_HANDLE_INFORMATION structure is an index to TypeInformation array.

Harder is to get the name of handle from other process. There are two possibilities how to name it. First is to copy the handle via NtDuplicateObject to our process and then to name it. This method will fail for some specific types of handles. But it will fail only for few, so we can stay calm

and use
this.

```

NtDuplicateObject(
    IN HANDLE SourceProcessHandle,
    IN HANDLE SourceHandle,
    IN HANDLE TargetProcessHandle,
    OUT PHANDLE TargetHandle OPTIONAL,
    IN ACCESS_MASK DesiredAccess,
    IN ULONG Attributes,
    IN ULONG Options
);

```

SourceProcessHandle is a handle of process which owns SourceHandle which is the handle we want to copy. TargetProcessHandle is handle of process where to copy. This will be handle to our process in our case. TargetHandle is the pointer on handle where to save a copy of original handle. DesiredAccess should be set to PROCESS_QUERY_INFORMATION, Attributes and Options to 0.

Second naming method which works with any handle is to use system driver. Source code for this is available in OpHandle project on my site <http://rootkit.host.sk>.

====[10. Ports]
 =====

The easiest way to enumerate open ports is to use functions called AllocateAndGetTcpTableFromStack and AllocateAndGetUdpTableFromStack, and or AllocateAndGetTcpExTableFromStack and AllocateAndGetUdpExTableFromStack from iphlpapi.dll. The Ex functions are available since Windows XP.

```
typedef struct _MIB_TCPROW {
    DWORD dwState;
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
    DWORD dwRemoteAddr;
    DWORD dwRemotePort;
} MIB_TCPROW, *PMIB_TCPROW;

typedef struct _MIB_TCPTABLE {
    DWORD dwNumEntries;
    MIB_TCPROW table[ANY_SIZE];
} MIB_TCPTABLE, *PMIB_TCPTABLE;

typedef struct _MIB_UDPROW {
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
} MIB_UDPROW, *PMIB_UDPROW;

typedef struct _MIB_UDPTABLE {
    DWORD dwNumEntries;
    MIB_UDPROW table[ANY_SIZE];
} MIB_UDPTABLE, *PMIB_UDPTABLE;

typedef struct _MIB_TCPROW_EX
{
    DWORD dwState;
    DWORD dwLocalAddr;
    DWORD dwLocalPort;
    DWORD dwRemoteAddr;
    DWORD dwRemotePort;
    DWORD dwProcessId;
} MIB_TCPROW_EX, *PMIB_TCPROW_EX;

typedef struct _MIB_TCPTABLE_EX
{
    DWORD dwNumEntries;
    MIB_TCPROW_EX table[ANY_SIZE];
} MIB_TCPTABLE_EX, *PMIB_TCPTABLE_EX;

typedef struct _MIB_UDPROW_EX
{
    DWORD dwLocalAddr;
```

```
        DWORD dwLocalPort;
        DWORD dwProcessId;
    } MIB_UDPROW_EX, *PMIB_UDPROW_EX;

typedef struct _MIB_UDPTABLE_EX
{
    DWORD dwNumEntries;
    MIB_UDPROW_EX table[ANY_SIZE];
} MIB_UDPTABLE_EX, *PMIB_UDPTABLE_EX;

DWORD WINAPI AllocateAndGetTcpTableFromStack(
    OUT PMIB_TCPTABLE *pTcpTable,
    IN BOOL bOrder,
    IN HANDLE hAllocHeap,
    IN DWORD dwAllocFlags,
    IN DWORD dwProtocolVersion;
);

DWORD WINAPI AllocateAndGetUdpTableFromStack(
    OUT PMIB_UDPTABLE *pUdpTable,
    IN BOOL bOrder,
    IN HANDLE hAllocHeap,
    IN DWORD dwAllocFlags,
    IN DWORD dwProtocolVersion;
);

DWORD WINAPI AllocateAndGetTcpExTableFromStack(
    OUT PMIB_TCPTABLE_EX *pTcpTableEx,
    IN BOOL bOrder,
    IN HANDLE hAllocHeap,
    IN DWORD dwAllocFlags,
    IN DWORD dwProtocolVersion;
);

DWORD WINAPI AllocateAndGetUdpExTableFromStack(
    OUT PMIB_UDPTABLE_EX *pUdpTableEx,
    IN BOOL bOrder,
    IN HANDLE hAllocHeap,
    IN DWORD dwAllocFlags,
    IN DWORD dwProtocolVersion;
);
```

There is another way to do this stuff. When program creates a socket and starts listening it surely has an open handle for it and for open port. We can enumerate all open handles in the system and send them special buffer via NtDeviceIoControlFile to find out whether the handle is for open port or not. This will also give us information about the port. Because there are a lot of open handles we will test only handles which type is File and name is \Device\Tcp or \Device\Udp. Open ports have only this type and name.

When we look to the code of iphlpapi.dll functions above we find out that these functions also calls NtDeviceIoControlFile and sends special buffer to get a list of all open ports in the system. That mean only functions we need to hook for hiding ports is NtDeviceIoControlFile.

```
NTSTATUS NtDeviceIoControlFile(  
    IN HANDLE FileHandle  
    IN HANDLE Event OPTIONAL,  
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,  
    IN PVOID ApcContext OPTIONAL,  
    OUT PIO_STATUS_BLOCK IoStatusBlock,  
    IN ULONG IoControlCode,  
    IN PVOID InputBuffer OPTIONAL,  
    IN ULONG InputBufferLength,  
    OUT PVOID OutputBuffer OPTIONAL,  
    IN ULONG OutputBufferLength  
);
```

Interesting agruments for us now are FileHandle which specify a handle of device to communicate with, IoStatusBlock which points to a variable that receives the final completion status and information about the requested

operation, IoControlCode that is a number specifying type of the device, method, file access and a function. InputBuffer contains input data that are InputBufferLength bytes long and similarly OutputBuffer and OutputbufferLength.

====[10.1 Netstat, OpPorts on WinXP, FPort on WinXP]
=====

Getting a list of all open ports is the first way which is used by e.g.

OpPorts and FPort on Windows XP and also Netstat.

Programs calls here NtDeviceIoControlFile twice with IoControlCode

0x000120003. OutputBuffer is filled after a second call. Name of FileHandle is

here always \Device\Tcp. InputBuffer differs for different types of call:

1) To get an array of MIB_TCPROW InputBuffer looks as follows:

first call:

0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
0x00 0x00 0x00 0x00

second call:

0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00
0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
0x00 0x00 0x00 0x00

2) To get an array of MIB_UDPROW:

first call:

0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00

```
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
0x00 0x00 0x00 0x00
```

second call:

```
0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00
0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
0x00 0x00 0x00 0x00
```

3) To get an array of MIB_TCPROW_EX:

first call:

```
0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
0x00 0x00 0x00 0x00
```

second call:

```
0x00 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00
0x02 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
0x00 0x00 0x00 0x00
```

4) To get an array of MIB_UDPROW_EX:

first call:

```
0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00
0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
0x00 0x00 0x00 0x00
```

second call:

```
0x01 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x01
0x00 0x00
0x02 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00
```

0x00 0x00 0x00 0x00

You can see the buffers are different in few bytes only. We can lucidly recapitulate these:

Calls we are interested in have InputBuffer[1] set to 0x04 and mainly InputBuffer[17] on 0x01. Only after these input data we get filled OutputBuffer with desiderative tables. If we want to get info about TCP ports we set InputBuffer[0] on 0x00, or on 0x01 for information about UDP. If we want extended output tables (MIB_TCPROW_EX or MIB_UDPROW_EX) we use Inputbuffer[16] in second call set to 0x02.

If we find out the call with these parameters we can change the output buffer. To get number of rows in output buffer simply divide Information from IoStatusBlock by size of the row. Hiding of one row is easy then. Just rewrite it with following rows and delete last row. Don't forget to change OutputBufferLength and IoStatusBlock.

====[10.2 OpPorts on Win2k and NT4, FPort on Win2k]
=====

We use NtDeviceIoControlFile with IoControlCode 0x00210012 to determine if the handle of File type and name \Device\Tcp or \Device\Udp is the handle of open port.

So at first we compare IoControlCode and then a type and the name of the handle. If it is still interesting then we compare the length of input buffer which should be equal to the length of struct

TDI_CONNECTION_IN. This length is 0x18. OutputBuffer is TDI_CONNECTION_OUT.

```
typedef struct _TDI_CONNECTION_IN
{
    ULONG UserDataLength,
    PVOID UserData,
    ULONG OptionsLength,
    PVOID Options,
    ULONG RemoteAddressLength,
    PVOID RemoteAddress
} TDI_CONNECTION_IN, *PTDI_CONNECTION_IN;

typedef struct _TDI_CONNECTION_OUT
{
    ULONG State,
    ULONG Event,
    ULONG TransmittedTsdus,
    ULONG ReceivedTsdus,
    ULONG TransmissionErrors,
    ULONG ReceiveErrors,
    LARGE_INTEGER Throughput
    LARGE_INTEGER Delay,
    ULONG SendBufferSize,
    ULONG ReceiveBufferSize,
    ULONG Unreliable,
    ULONG Unknown1[5],
    USHORT Unknown2
} TDI_CONNECTION_OUT, *PTDI_CONNECTION_OUT;
```

Concrete implementation of how to determine the handle is open port is available in source code of OpPorts on <http://rootkit.host.sk>. We are interested in hiding specific port now. We already compared InputBufferLength and IoControlCode. Now we have to compare RemoteAddressLength. This is always 3 or 4 for open port. The last we have to do is to compare ReceivedTsdus from OutputBuffer which contains the port in network form and our list of ports we

want to hide. Differentiate between TCP and UDP is done according to the name of the handle. By deleting OutputBuffer, changing IoStatusBlock and returning the value of STATUS_INVALID_ADDRESS we will hide this port.

=====[11. Ending]

=====

Concrete implementation of described techniques will be available with the source code of Hander defender rootkit in version 1.0.0 on its homepage <http://rootkit.host.sk> and on <http://www.rootkit.com>.

It is possible I will add some more information about invisibility on Windows NT in the future. New versions of this document could also contain improvement of described methods or new comments.

Special thanks to Ratter who give me a lot of knowhow which was necessary to write this document and to code project Hacker defender.

Send all remarks to holy_father@phreaker.net or to the board on <http://rootkit.host.sk>.

=====[End]

=====