Special Interest Group in
Security and Information Integrity (SIG^2)
http://www.security.org.sg

SIG^2 G-TEC Secure Code Study Research Paper

Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration

Author: Chew Keong TAN (chewkeong@security.org.sg)

8 July 2004 (Updated 3 October 2004)

Introduction

Win32 Kernel Rootkits modify the behaviour of the system by Kernel Native API hooking. This technique is typically implemented by modifying the entries within the kernel's System Service Dispatch Table. Such modification ensures that a hook function installed by the rootkit is called prior to the original native API. The hook function usually calls the original native API and modifies the output before returning the results to the user-space program. This technique allows kernel rootkits to hide files, processes, and prevent termination of malicious processes.

This paper gives a short introduction to the technique of Kernel Native API hooking, and proposes a technique for defeating kernel rootkits that hook native APIs by System Service Dispatch Table modification. The proposed technique restores the System Service Dispatch Table directly from user-space and do not require a kernel driver to be loaded.

Kernel Native API Hooking by System Service Dispatch Table Modification

In Windows, user-space applications request for system services by calling APIs that are exported by various DLLs. For example, to write data to an open file, pipe or device, the WriteFile API that is exported by kernel32.dll is usually used. Within kernel32.dll, the implementation of WriteFile API in turn calls the ZwWriteFile native API that is exported by ntdll.dll. The work done by ZwWriteFile is actually performed in kernel-space. Hence, the implementation of ZwWriteFile in ntdll.dll contains only minimal code to transit into kernel-space using interrupt 0x2E. The disassembly of ZwWriteFile on Win2K is shown below.

```
1- MOV EAX, 0ED
2- LEA EDX, DWORD PTR SS:[ESP+4]
3- INT 2E
4- RETN 24
```

The magic number 0xED in line 1 is the Service Number for ZwWriteFile in Win2K. It is used as an index into the kernel's System Service Dispatch Table (SSDT) to locate the address of the service function that implements the actual code for writing to files, pipes or devices. The address of SSDT can be found within the Service Descriptor Table (SDT).

The SDT can be referenced using the KeServiceDescriptorTable symbol, which is exported by ntoskrnl.exe. It is a structure with the following definition.

```
typedef struct ServiceDescriptorTable {
        SDE ServiceDescriptor[4];
} SDT;

typedef struct ServiceDescriptorEntry {
      PDWORD KiServiceTable;
      PDWORD CounterTableBase;
      DWORD  ServiceLimit;
      PBYTE  ArgumentTable;
} SDE;
```

The first member of the structure, SDT.*ServiceDescriptor[0].KiServiceTable,* contains a pointer to the SSDT of the system services implemented by ntoskrnl.exe. As mentioned earlier, the SSDT contains an array of function pointers to the service functions that handle native API calls. The *ServiceLimit* member gives the number of entries in the SSDT.

The DWORD value at KiServiceTable[0xED] is a function pointer to NtWriteFile, which contains the actual code to write to files, pipes or devices. Hence, to modify the behaviour of the user-space WriteFile API, one simply needs to write a hook (replacement) function, load it into kernel-space as a driver, and modify KiServiceTable[0xED] to point to the hook function. The hook function needs to keep a copy of the original function pointer (original value of KiServiceTable[0xED]), so that the original function can be called to perform its intended task.

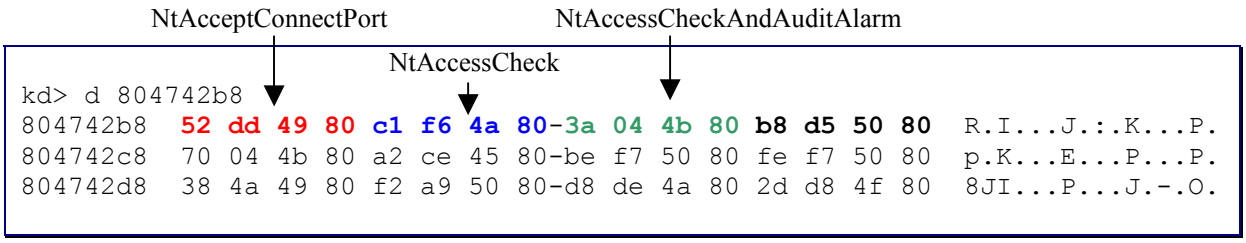
The following screen dump from WinDbg shows the contents of KeServiceDescriptorTable and KeServiceDescriptorTable.KiServiceTable.

```
kd> d KeServiceDescriptorTable
8046dfa0  b8 42 47 80 00 00 00 00-f8 00 00 00 9c 46 47 80  .BG.........FG.
```

Address of KiServiceTable

Number of Entries in KiServiceTable

Dump of KeServiceDescriptorTable on Win2K



Dump of KeServiceDescriptorTable.KiServiceTable on Win2K

The following examples illustrate how Kernel Native API hooking can be used to modify the behaviour of certain APIs.

Example One - Process Hiding by Hooking ZwQuerySystemInformation

User-space programs can use the APIs exported by the ToolHelp DLL to obtain a list of all running processes. The APIs in turn calls the ZwQuerySystemInformation native API exported by ntdll.dll to obtain the list of running processes by specifying (SystemProcessesAndThreadsInformation) as its first parameter. To hide processes, a Win2K kernel-space rootkit, which is loaded as a driver, can modify the function pointer at KiServiceTable[0x97] (ZwQuerySystemInformation) to redirect the call to a hook function.

The hook function first calls the original ZwQuerySystemInformation API to obtain an array containing information of all running process. The returned array is then modified to remove the entry containing the process to be hidden. Finally, the modified result is returned to the user-space program. This effectively prevents the user-space program from "seeing" the hidden process.

Example Two - Driver/Module Hiding by Hooking ZwQuerySystemInformation

User-space programs can obtain a list of all loaded drivers using the ZwQuerySystemInformation native API, specifying SystemModuleInformation as its first parameter. As mentioned earlier, ZwQuerySystemInformation is exported by ntdll.dll and can be called directly by user-space programs. In kernel-space, the ZwQuerySystemInformation native API obtains the list of loaded drivers by traversing the PsLoadedModuleList.

A Win2K kernel-space rootkit can manipulate the results returned by ZwQuerySystemInformation by modifying KiServiceTable[0x97] (ZwQuerySystemInformation) to point to a hook function. The hook function will first call the original ZwQuerySystemInformation to get an array of all loaded drivers. The driver to be hidden (i.e. the rootkit) is then removed from the array. This manipulated array is returned to the user-space program.

Example Three – File Hiding by Hooking ZwQueryDirectoryFile

User-space programs use the FindFirstFile and FindNextFile APIs exported by kernel32.dll to obtain a listing of all files in a directory. These APIs ultimately calls the ZwQueryDirectoryFile native API to retrieve the required file listing. A kernel-space rootkit can manipulate the output of ZwQueryDirectoryFile to remove any entries containing the file to be hidden before returning the results to the user-space program.

Restoring the System Service Dispatch Table

From the above examples, it should be obvious that if we could restore the System Service Dispatch Table to its original state, we would be able to disable any kernel rootkits that modifies system behaviour by hooking entries within the System Service Dispatch Table. The following sections describe in detail how this could be done. A proof-of-concept (POC) rootkit-defense tool, ***SDTrestore***, was developed to illustrate the techniques described in this paper. This POC tool can be downloaded from the following URL.

http://www.security.org.sg/code/sdtrestore.html

Modifying the System Service Dispatch Table from User-Space

The System Service Dispatch Table (SSDT) exists in kernel-space and normally, to modify entries in the SSDT, the rootkit must load itself into the running kernel as a driver. However, it is possible for a user-space program to modify the SSDT entries by writing directly to kernel memory using \device\physicalmemory.

Mark Russinovich from Sysinternals first used \device\physicalmemory in his Physmem tool to allow the viewing of physical memory [3]. An excellent article that describes in detail how to read and write to kernel memory using \device\physicalmemory can be found at [2]. An interesting code that shows how to hide process by direct manipulation of kernel memory via \device\physicalmemory can be found at [4].

The following sequence of steps describes how a user-space program that runs with Administrator privilege can gain read/write access to kernel memory via \device\physicalmemory.

1. Use NtOpenSection native API (exported by ntdll.dll) with SECTION_MAP_READ | SECTION_MAP_WRITE access flags to get a handle to \device\physicalmemory. This will usually fail since the Administrator does not have SECTION_MAP_WRITE access rights to \device\physicalmemory.

2. Use NtOpenSection native API with READ_CONTROL | WRITE_DAC access flag to get a handle to \device\physicalmemory. This allows a new DACL to be added to the \device\physicalmemory object.

3. Add a DACL to \device\physicalmemory, granting SECTION_MAP_WRITE access to the Administrator account.

4. Try to get a handle to \device\physicalmemory again using NtOpenSection native API with SECTION_MAP_READ | SECTION_MAP_WRITE access flags.

After performing the above sequence of steps, the user-space program would have successfully obtained a handle to \device\physicalmemory. In order to write to physical memory, the program must first map the physical memory page into its virtual address space. This can be done using the NtMapViewOfSection native API as shown below.

```
ntStatus = _NtMapViewOfSection(
      hPhyMem,                        // Handle to \device\physicalmemory
      (HANDLE)-1,
      virtualAddr,       // OUT - Virtual memory where the physical memory
                                 is mapped to.
      0,
      *length,
      &viewBase,         // IN/OUT - Physical memory address to map in
      length,            // IN/OUT - Size of the mapped physical memory
      ViewShare,
      0,
      PAGE_READWRITE     // Map for READ/WRITE access
      );
```

After mapping the physical memory pages into its virtual memory space, a user-space program can then read and write to them like any allocated memory. The output parameter *virtualAddr*, gives the virtual memory address where the physical memory pages are mapped to.

Locating the Memory Address of the System Service Dispatch Table

In order for a user-space program to modify the System Service Dispatch Table entries, it must first determine its physical memory address and map the page into its virtual memory space. The address of the SSDT can be found in the *KiServiceTable* member of the *KeServiceDescriptorTable* structure. This means that we must first locate *KeServiceDescriptorTable* before we can get the address of the SSDT. However, the memory address of *KeServiceDescriptorTable* differs across the different kernel Service Pack versions. Despite this, it is still possible for a user-space program to reliably determine the address of *KeServiceDescriptorTable* since this symbol is exported by ntoskrnl.exe. To obtain this address, a user-space program first loads ntoskrnl.exe into memory with proper memory alignment. The offset address of *KeServiceDescriptorTable* is then determined by searching for its symbol in the export table of ntoskrnl.exe.

The offset address of *KeServiceDescriptorTable* is then converted to physical memory address and the corresponding physical memory page is mapped into the virtual memory space of the user-space program. To convert the offset address of *KeServiceDescriptorTable* to physical memory address, we must first determine the kernel's base-address in protected-

mode virtual memory. This is easily done by calling ZwQuerySystemInformation with SystemModuleInformation as its first parameter. With the kernel base-address, the physical memory address containing *KeServiceDescriptorTable* can be calculated as follows.

```
PhyMemAddrKeServiceDescriptorTable =
       KernelVirtualBaseAddr + OffsetAddrKeServiceDescriptorTable – 0x80000000
```

In this case, we assume that protected-mode virtual memory starts at 0x80000000.

After mapping the physical memory page containing *KeServiceDescriptorTable* (using \device\physicalmemory), we can determine the address of the System Service Dispatch Table by reading its first structure element, *ServiceDescriptor[0].KiServiceTable*. The address that was read must be converted to physical memory address before it can be used to map the page containing the System Service Dispatch Table. This address is easily calculated as follows, assuming that protected-mode virtual memory starts at 0x80000000.

```
PhyMemAddrServiceTable = VirtualMemAddrServiceTable – 0x80000000
```

The virtual address of *KiServiceTable* is also used to locate the original copy of System Service Dispatch Table within ntoskrnl.exe. The offset location of the original SSDT in the disk image of ntoskrnl.exe is easily calculated as follows.

```
OffsetAddrServiceTable = VirtualMemAddrServiceTable – KernelVirtualBaseAddr
```

Shortly after the release of our SDTrestore tool, 90210 suggested on rookit.com an improved technique of locating *KiServiceTable* [10]. This technique is based on the observation that *KeServiceDescriptorTable* is initialized in the *KiInitSystem* function with the following instruction.

```
mov     ds:KeServiceDescriptorTable, offset KiServiceTable
```

It is possible to locate this instruction within ntoskrnl.exe by scanning its relocation table for references to instruction that corresponds to the form "*mov KeServiceDescriptorTable, imm32*". Using the relocation table to assist in the scanning is more efficient and reliable than scanning the entire code section of ntoskrnl.exe for the above instruction. Once this instruction is located, the offset address of *KiServiceTable* can then be determined.

Restoring Modified Entries in the System Service Dispatch Table

After the physical memory page containing the running kernel's System Service Dispatch Table was mapped, a loop will compare all entries of the running kernel's SSDT with the original SSDT stored within the disk image of ntoskrnl.exe. Each entry within the running

kernel's SSDT is actually a function pointer with absolute virtual address. This must be converted to offset address before it can be compared against its corresponding entry from the original System Service Dispatch Table. The conversion is done as follows.

```
OffsetAddrOfFuncPtr = VirtualMemAbsAddrOfFuncPtr – KernelVirtualBaseAddr
```

Any discrepancies will indicate that the particular native API has been hooked, and any hooked SSDT entries can be restored using the original values obtained from the disk image (ntoskrnl.exe). Prior to restoration, the original values obtained from the disk image must first be converted from offset address to absolute virtual address.

Disabling He4Hook's Kernel Native API Hooks by SSDT Restoration

He4Hook is a kernel rootkit that uses Kernel Native API hooking as one of the ways to hide and protect files/directories. Using our SDTrestore rootkit-defense tool, we found that He4Hook hooks the following native API when its file-system hooking feature is enabled using the **–hk:1** option.

```
C:\>he4hookcontrol -hk:1

He4HookControl v2.03 - control utility for He4HookInv
Copyright (C) 2000 He4 developers team
He4Dev@hotmail.com

He4HooInv device installed -
     Version: 20001005
     Base: 8121D000
File system - hooked

C:\>sdtrestore
SDTrestore Version 0.1 Proof-of-Concept by SIG^2 G-TEC (www.security.org.sg)

KeServiceDescriptorTable               8046DFA0
KeServiceDecriptorTable.ServiceTable   804742B8
KeServiceDescriptorTable.ServiceLimit  248

ZwCreateFile              20 --[hooked by unknown at 81222476]--
ZwOpenFile                64 --[hooked by unknown at 812224A8]--
ZwQueryDirectoryFile      7D --[hooked by unknown at 812224D2]--

Number of Service Table entries hooked = 3

WARNING:  THIS IS EXPERIMENTAL CODE.  FIXING THE SDT MAY HAVE GRAVE
CONSEQUENCES, SUCH AS SYSTEM CRASH, DATA LOSS OR SYSTEM CORRUPTION.
PROCEED AT YOUR OWN RISK.  YOU HAVE BEEN WARNED.

Fix SDT Entries (Y/N)? :
```

ZwQueryDirectoryFile was hooked by He4Hook to hide files and directories from directory listings. Hooking ZwCreateFile and ZwOpenFile allows He4Hook to restrict the type of assess on protected files and directories. Using SDTrestore, we were able to restore the SSDT to its original state. Restoration of the System Service Dispatch Table effectively disables the file and directory protection feature of He4Hook when it is used with the –hk:1 option. This is illustrated by the screen dump below.

```
C:\>dir se*
 Volume in drive C has no label.
 Volume Serial Number is BC03-1AEF

 Directory of C:\

06/29/2004  10:15p      <DIR>          secret
               0 File(s)              0 bytes
               1 Dir(s)   3,151,261,696 bytes free

C:\>he4hookcontrol -a:c:\secret -c:R

He4HookControl v2.03 - control utility for He4HookInv
Copyright (C) 2000 He4 developers team
He4Dev@hotmail.com

He4HooInv device installed -
     Version: 20001005
     Base: 8121D000

Protected files list:
c:\secret (R)

C:\>dir se*
 Volume in drive C has no label.
 Volume Serial Number is BC03-1AEF

 Directory of C:\

File Not Found

C:\>sdtrestore
SDTrestore Version 0.1 Proof-of-Concept by SIG^2 G-TEC (www.security.org.sg)

KeServiceDescriptorTable                8046DFA0
KeServiceDecriptorTable.ServiceTable    804742B8
KeServiceDescriptorTable.ServiceLimit   248

ZwCreateFile               20 --[hooked by unknown at 81222476]--
ZwOpenFile                 64 --[hooked by unknown at 812224A8]--
ZwQueryDirectoryFile       7D --[hooked by unknown at 812224D2]--

Number of Service Table entries hooked = 3

WARNING:  THIS IS EXPERIMENTAL CODE.  FIXING THE SDT MAY HAVE GRAVE
CONSEQUENCES, SUCH AS SYSTEM CRASH, DATA LOSS OR SYSTEM CORRUPTION.
PROCEED AT YOUR OWN RISK.  YOU HAVE BEEN WARNED.

Fix SDT Entries (Y/N)? : y

[+] Patched SDT entry 20 to 80497EF9
[+] Patched SDT entry 64 to 80498755
[+] Patched SDT entry 7D to 80498541

C:\>dir se*
 Volume in drive C has no label.
 Volume Serial Number is BC03-1AEF

 Directory of C:\

06/29/2004  10:15p      <DIR>          secret
               0 File(s)              0 bytes
               1 Dir(s)   3,151,261,696 bytes free
```

Security Tools that use Kernel Native API Hooking

The technique of Kernel Native API hooking by System Service Dispatch Table manipulation is not used solely by rootkits. Using our KProcCheck tool [5], we found that several security tools also use this technique for a variety of purposes. The following are some of the security tools that use Kernel Native API hooking.

- DiamondCS Process Guard (v2.000)
- Kerio Personal Firewall 4 (v4.0.16)
- Sebek (v2.1.5)

DiamondCS™ Process Guard v2.000

Process Guard is a Win32 security system that protects both system and security processes (as well as user-defined processes) from attacks by other processes, services, drivers, and other forms of executing code on a system. It can protect a process against termination, suspension and prevents loading of malicious kernel drivers.

Using KProcCheck, we found that Process Guard works by hooking the following native APIs.

```
KProcCheck Version 0.1 Proof-of-Concept by SIG^2 (www.security.org.sg)

Checks SDT for Hooked Native APIs

ZwCreateFile               20 \??\C:\WINNT\System32\drivers\procguard.sys [F7392D8A]
ZwCreateKey                23 \??\C:\WINNT\System32\drivers\procguard.sys [F7391F98]
ZwCreateThread             2E \??\C:\WINNT\System32\drivers\procguard.sys [F73924FC]
ZwOpenFile                 64 \??\C:\WINNT\System32\drivers\procguard.sys [F7392C62]
ZwOpenKey                  67 \??\C:\WINNT\System32\drivers\procguard.sys [F7391F64]
ZwOpenProcess              6A \??\C:\WINNT\System32\drivers\procguard.sys [F739289E]
ZwOpenThread               6F \??\C:\WINNT\System32\drivers\procguard.sys [F73926F8]
ZwRequestWaitReplyPort     B0 \??\C:\WINNT\System32\drivers\procguard.sys [F7390AE6]
ZwSetValueKey              D7 \??\C:\WINNT\System32\drivers\procguard.sys [F739224E]
ZwWriteVirtualMemory       F0 \??\C:\WINNT\System32\drivers\procguard.sys [F7392A40]

Number of Service Table entries hooked = 10
```

Further testing reveals that by restoring the System Service Dispatch Table with our SDTrestore rootkit-defense tool, we were able to disable the protection offered by Process Guard. In other words, the processes that were being protected by Process Guard can now be easily terminated using Windows Task Manager.

Kerio Personal Firewall 4 (v4.0.16)

Kerio Personal Firewall (KPF) is a state-of-the-art personal firewall that helps users restrict how their computers exchange data with other computers on the Internet or local network. KPF has a System Security feature that allows the user to control the execution of programs on his system. KPF prevents malicious code from spawning processes on the user's system

by prompting the user for action whenever an unknown/new or modified program is being executed.

The System Security feature works by hooking the following native APIs.

```
KProcCheck Version 0.1 Proof-of-Concept by SIG^2 (www.security.org.sg)

Checks SDT for Hooked Native APIs

ZwCreateFile               20 \SystemRoot\system32\drivers\fwdrv.sys [BFBD3830]
ZwCreateProcess            29 \SystemRoot\system32\drivers\fwdrv.sys [BFBD3380]
ZwCreateThread             2E \SystemRoot\system32\drivers\fwdrv.sys [BFBD35E0]
ZwResumeThread             B5 \SystemRoot\system32\drivers\fwdrv.sys [BFBD3630]

Number of Service Table entries hooked = 4
```

By restoring the System Service Dispatch Table with our SDTrestore rootkit-defense tool, we were able to disable the System Security feature of KPF4. With the feature disabled, KPF4 will no longer prompt the user for actions when an unknown/new or modified program is being executed.

Sebek (v2.1.5)

Sebek is a data capture tool designed to capture the attackers activities on a honeypot without the attacker knowing it. It has two components. The first is a client that runs on the honeypots, its purpose is to capture all of the attackers activities (keystrokes, file uploads, passwords) then covertly send the data to the server. The second component is the server that collects data from the honeypots.

Sebek prevents itself from being detected by hooking several native APIs in kernel-space. Hooking is performed in the module SEBEK.sys by replacing entries within the SDT ServiceTable. Sebek logs all console events by hooking ZwReadFile and ZwWriteFile. Hooking these two native APIs allows Sebek to trap any read/write request to the console and to send them to the logging server.

Using KProcCheck, we were able to determine that Sebek hooks the following native APIs.

```
C:\>kproccheck -t
KProcCheck Version 0.1 Proof-of-Concept by SIG^2 (www.security.org.sg)

Checks SDT for Hooked Native APIs

ZwClose                     18 SEBEK.sys [F729A092]
ZwCreateFile                20 SEBEK.sys [F729A98C]
ZwCreateKey                 23 SEBEK.sys [F729AD10]
ZwEnumerateKey              3C SEBEK.sys [F729AE02]
ZwEnumerateValueKey         3D SEBEK.sys [F729AA50]
ZwOpenFile                  64 SEBEK.sys [F729A8E6]
ZwOpenKey                   67 SEBEK.sys [F729AD88]
ZwQueryDirectoryFile        7D SEBEK.sys [F729A4CC]
ZwQuerySystemInformation    97 SEBEK.sys [F729A5F0]
ZwReadFile                  A1 SEBEK.sys [F7299CF0]
ZwRequestWaitReplyPort      B0 SEBEK.sys [F7299F14]
ZwSecureConnectPort         B8 SEBEK.sys [F7299FE6]
ZwWriteFile                 ED SEBEK.sys [F7299D48]

Number of Service Table entries hooked = 13
```

By restoring the System Service Dispatch Table entries, we were able to disable the console logging and anti-detection ability of Sebek.

Conclusion

In this paper, we have given a short introduction to the technique of Kernel Native API hooking by System Service Dispatch Table manipulation. This technique has been used by kernel rootkits to modify the behaviour of the system. We have shown that it is possible for a user-space program to disable such rootkits by restoring the SSDT to its original state by writing directly to protected memory via \device\physicalmemory.

In our research, we also found that several security tools use similar hooking technique to implement some of their security features. We have shown that it is possible for a user-space program to disable their security features by restoring the SSDT. Hence, we recommended that such security tools should take additional steps to prevent the restoration of the System Service Dispatch Table entries, which could lead to the disabling of their security features. For example, by preventing user-space programs from loading kernel drivers and blocking write access to \device\physicalmemory.

References

[1] Greg Hoglund, "NT Rootkit – The original and first public NT ROOTKIT". http://www.rootkit.com

[2] crazylord, "Playing with Windows /dev/(k)mem", Phrack Volume 0x0b, Issue 0x3b, Phile #0x10 of 0x12, Jul 2002. http://www.phrack.org/phrack/59/p59-0x10.txt

[3] Mark Russinovich, "Physmem", Systems Internals. http://www.sysinternals.com/files/physmem.zip

[4] 90210, "Process Hide", 29A#7 magazine, VX Heavens, Jan 2004. http://vx.netlux.org/vx.php?id=ep12

[5] Tan Chew Keong, "Win2K Kernel Hidden Process/Module Checker 0.1 (Proof-Of-Concept)", May 2004. http://www.security.org.sg/code/kproccheck.html

[6] He4Hook, http://www.rootkit.com/vault/hoglund/He4Hook215b6.zip

[7] fuzen_op, "FU Rootkit", https://www.rootkit.com/vault/fuzen_op/FU_Rootkit.zip

[8] joanna, "klister", http://www.rootkit.com/vault/joanna/klister-0.4.zip

[9] firew0rker, "Kernel-mode backdoors for Windows NT", Phrack 62, Volume 0x0b, Issue 0x3e, Phile #0x06 of 0x10, July 2004.

[10] 90210, "A more stable way to locate real KiServiceTable", http://www.rootkit.com/newsread.php?newsid=176

[11] David A. Soloman and Mark E. Russinovich, "Inside Microsoft Windows 2000 Third Edition"

[12] Sven B. Schreiber, "Undocumented Windows 2000 Secrets, A Programmer's Cookbook"

Acknowledgement

The author would like to thank the SIG^2 G-TEC Lab (http://www.security.org.sg/webdocs/g-tec.html) for supporting this research.