Master Thesis
Computer Science
Thesis no: MSC 2004:9
06 – 2004

# Localization of Spyware in Windows Environments

**Authors: Fredrik Bergstrand**
**Johan Bergstrand**
**Håkan Gunnarsson**

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

**Contact Information:**
Authors:
Fredrik Bergstrand
E-mail: cfb@home.se

Johan Bergstrand
E-Mail: jb78@home.se

Håkan Gunnarsson
E-Mail: hakan.gunnarsson@klostersfalad.se

University advisors:
Martin Boldt
Bengt Carlsson
Department of Software Engineering and Computer Science

Internet: www.tek.bth.se
Phone: +46 457 38 50 00
Fax: 46 457 271 25

# Abstract

This is a thesis about different methods that can be used to detect spyware. Methods included are Layered Service Provider, Internet Protocol Helper API, TDI filtering and API hooking. Some firewall testing applications, leak tests, that use methods that can be used by real spyware program to penetrate firewalls have also been examined. The goal was to develop a Windows 2000/XP program that is able to detect as many of our examined leak tests as possible. Our program uses the methods TDI filtering and API hooking for detection of spyware because our study showed that these methods were the best. To evaluate the program it was tested against our examined leak test programs. Our program managed to detect all leak tests except one.

**Keywords:** spyware, firewall, leak test, API hooking

# Contents

**Summary** .......... **2**
**1. Introduction** .......... **3**
**2. Windows 2000 Network Architecture** .......... **4**
2.1 User Mode .......... 4
2.1.1 Windows Socket 2.0 .......... 4
2.2 Kernel Mode .......... 5
2.2.1 Transport Driver Interface .......... 5
2.2.2 TCP/IP Driver .......... 5
2.2.3 Network Driver Interface Specification .......... 5
**3. Methods to monitor network traffic** .......... **7**
3.1 Layered Service Provider .......... 7
3.2 Internet Protocol Helper API .......... 7
3.2.1 How to retrieve the TCP connection table .......... 7
3.2.2 How to bind a TCP connection to a process and a parent process .......... 8
3.3 TDI Filtering .......... 8
**4. Leak tests** .......... **9**
4.1 Tooleaky .......... 9
4.2 Firehole .......... 9
4.3 Thermite .......... 9
4.4 Copycat .......... 9
4.5 Yalta .......... 9
4.6 Wallbreaker .......... 9
4.7 PC Audit and PC Audit2 .......... 9
4.8 Ghost .......... 10
**5. DLL and Code Injection** .......... **11**
5.1 Common injection techniques .......... 11
5.2 Code Injection .......... 12
**6. Program design** .......... **13**
6.1 Spyware detection .......... 13
6.2 Stealth spyware detection .......... 13
6.3 Log viewer .......... 14
**7. Test results** .......... **16**
**8. Discussion and Conclusion** .......... **18**
**9. Further work** .......... **20**
**10. References** .......... **21**

**Appendix 1.** .......... **1**
Definitions .......... 1

# Summary

The demand for protection against applications, that disclose information about the behavior of users connected to the Internet, residing on the own computer has become more evident. This kind of applications is called spyware.

To identify and prevent applications based on their network activity in real time, the responsibility falls heavily on the firewall vendors. To prove the weaknesses in this area a growing number of firewall testing applications, leak tests, have been developed. These leak tests use methods that can be used by real spyware programs to penetrate firewalls.

This master thesis provides an overview of different methods that can be used to detect spyware. The methods that are included are Layered Service Provider, Internet Protocol Helper API, TDI filtering and API hooking. Some firewall testing applications, leak tests, have also been examined. The goal was to develop a Windows 2000/XP program that is able to detect as many of the examined leak tests as possible. This program has been the main part of this thesis and the most time-consuming. The program uses the methods TDI filtering and API hooking for detection of spyware because our study showed that these methods were the best. To evaluate the program it was tested against the leak test programs we have examined. Our program managed to detect all leak tests except one.

# 1. Introduction

Spyware and adware (see Appendix 1) that is monitoring people’s activity at their workstations is a problem that has got more and more attention in the last years. The things that these kinds of software have in common are that they collect information about the activity that has taken place at the workstation where it has been installed. What kind of information that is collected varies but it is possible for spyware applications to collect passwords, email addresses, web browsing history, online buying habits, your computer's hardware and software information. Apart from the fact that these applications invade the users privacy they can also make the system unstable and more able to crash due to poor implementation of such software. Furthermore, spyware and adware can lead to higher network load when it sends and especially when it receives information across the network between the local computer and the destination server.

Personal firewalls (see Appendix 1) have been criticized for merely protecting against hackers trying to break into the system and not being able to detect and prevent applications from accessing the Internet silently from within the system. To prove this fact a number of leak test programs (see Chap. 4) have been developed [28].

We have tested one of the most well known application monitoring programs, System Safety Monitor [32], against nine leak tests.

The purpose with this thesis is to look at different methods that could be used to localize spyware (see Chap. 3 and 5). The goal is to use one or more methods to develop a program (see Chap. 6) in C/C++ for Microsoft Windows 2000/XP platforms, which will detect spyware. This program will be the main part of this thesis. To evaluate our program, it will be tested (see Chap. 7) against some leak test programs.

In this thesis we will discuss methods for localization of spyware programs that use the TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) protocols. How to clean a computer from spyware will not be included. Microsoft Windows 2000/XP will be the target operating systems.

There are not many books on this topic that we could use, so our main source of information has been the Internet and Microsoft’s documentation MSDN. We decided to use open source projects found on the Internet to base our monitoring application on. The reason for this is that it would take too long to develop a whole monitoring system from scratch.

# 2. Windows 2000 Network Architecture

To understand how it is possible to monitor network traffic (see Chap. 3) it is necessary to take a general look at the Windows 2000 network architecture. It is divided into layers that have specific tasks to perform and within each layer more than one component can perform a similar task. The components that will be described in this chapter are Winsock 2.0 (see 1.1 – 1.8 in Fig. 1), the TDI (Transport Driver Interface, see 2 in Fig. 1), the TCP/IP Driver (see 3 in Fig. 1) and NDIS (Network Driver Interface Specification, see 4 in Fig. 1) because they are interesting by means of monitoring network traffic.

## 2.1 User Mode

A process, for example a user application or an environmental subsystem, that runs in this mode does not have full system access or privileges. The process is dependent on APIs to acquire system access and runs with privileges to access its own memory area. Win32 is an example of environmental subsystem that provides an API (see Windows API in Appendix 1) for operating system services, GUI capabilities, and functions to control all user input and output. [35]

### 2.1.1 Windows Socket 2.0

The first version of Winsock (see 1.1 in Fig. 1) became an industry standard and supports one well-defined interface to the TCP/IP suite of protocols. To allow backward compatibility Winsock 1.1 layer commands are converted to Winsock 2.0 layer commands. A Windows Sockets 16-bit 1.1 application calls the file Winsock.dll (Windows 3.1) and a 32-bit application calls Wsock32.dll (Windows 95). For more information about DLL files see Appendix 1.

Winsock 2.0 (see 1.2 in Fig. 1) extends the Winsock 1.1 interface to provide access to networks using protocols other than from the TCP/IP suit, such as NetWare and AppleTalk. It also provides an interface that applications can use to access many different namespaces, such as DNS (Domain Name System) and NDS (Novell Directory Services). Windows 32-bit network applications call the main Winsock 2.0 file Ws2_32.dll (Windows 98 and later). This file takes network requests from applications and sends those requests to the Winsock 2.0 SPI (Service Provider Interface, see 1.3 in Fig. 1). The file also provides traffic management. Between the Winsock 2.0 API and Winsock 2.0 SPI there are two other files, Mswsock.dll and Ws2help.dll. Mswsock.dll is an API that includes Microsoft extensions to Winsock and supplies services that are not part of Winsock. Ws2help.dll includes platform-specific utilities and supplies operating system-specific code that is not part of Winsock. [8]

The Winsock 2.0 SPI provides access to transport service providers and name space providers. Any number of service provider files can be linked dynamically into the main Winsock 2.0 file at run-time, allowing a WinSock 2.0 application to access services from one or more transport protocols simultaneously. [1]

There are two types of transport service providers, Layered Service Provider (see 1.4 in Fig. 1) and Base Service Provider (see 1.5 in Fig. 1). The Layered Service Provider extends the underlying protocol stack by providing additional services such as authentication, encryption, or proxy server services. The Base Service Provider exposes a Winsock interface to transport service DLL files (see Appendix 1), Helper DLLs, which directly implement different protocols. For example the file Wshtcpip.dll provides the TCP and UDP protocols.

Name Space DLLs (see 1.6 in Fig. 1) enable server and client applications to use an interface for several name services. Services register with the Winsock DLL, and client applications send requests for the names of those services to the Winsock DLL. The Winsock DLL manages registration and loading of name resolution providers and sends name resolution operations to the correct provider. Finally, the provider implements an interface with existing name services, such as DNS. [8]

The file Msafd.dll (see 1.7 in Fig. 1) includes a Winsock interface to kernel mode and the file Afd.sys (see 1.8 in Fig. 1) provides Winsock kernel interface to TDI transport protocols (see Chap. 2.2.1).

## 2.2 Kernel Mode

In this mode a process, for example a driver, runs with full privileges of system access. Any process running in this mode is not restricted to any specific memory space. [35]

### 2.2.1 Transport Driver Interface

The TDI (Transport Driver Interface, see 2 in Fig.1) provides a standard interface between network protocols and clients (applications, network redirectors or networking APIs) of these protocols. The task is to provide greater flexibility and functionality than existing interfaces such as Winsock. The TDI specification describes the set of functions and call mechanisms by which transport drivers and TDI clients communicate. [8]

### 2.2.2 TCP/IP Driver

The network protocols provide services, which allow applications or clients to send data over a network. Microsoft uses TCP/IP (see 3 in Fig.1) as the standard transport protocol for Windows 2000. One factor to the success of TCP/IP is that data packets can be routed to a different subnet by use of the packet's destination address. This ability gives TCP/IP greater fault tolerance. If a network error occurs packets are transported by a different route. Another factor to the success of TCP/IP is that it is the standard protocol for the Internet. [8]

### 2.2.3 Network Driver Interface Specification

NDIS (Network Driver Interface Specification, see 4 in Fig. 1) is a specification for network driver architectures that allow transport protocols such as TCP/IP to communicate with an underlying network adapter or another hardware device. NDIS acts as a boundary layer and permits protocol components to be independent of the network adapter by providing a standard interface to the network protocols. Windows 2000 network architecture supports NDIS, which requires that network adapter drivers follow the NDIS specification. Network driver developed after this specification is called NDIS-miniport. Windows 2000 NDIS includes both a standard connectionless interface and a connection-oriented interface.

Windows 2000 NDIS is implemented by the NDIS wrapper, which is a file called Ndis.sys. It contains code that surrounds all NDIS device drivers. The NDIS wrapper provides a standardized interface between protocol drivers and NDIS device drivers, and contains routines that make it easier to develop NDIS drivers. The NDIS wrapper defines the way a network adapter communicates with the protocols, which make them independent from the network adapter.

NDIS intermediate drivers are located between the transport protocols and the miniport drivers. Intermediate drivers translate packets coming and going to the protocol layer above, into packets that can be sent over a different network media. An intermediate driver can also be used as a filter driver. An example of this is a packet scheduler that schedules each packet passed down from the protocol layer for transmission and each packet received by a miniport driver based on its priority. Another possibility is to use an intermediate driver as a load-balancing driver. It exposes one virtual network adapter to the transport protocols but sends packets across more than one network adapter.

NDIS Hooking Filter Drivers can be used to filter network traffic. This driver intercept some NDIS functions, which makes it possible to trace registration of all installed protocols and which network interface they open. [8]

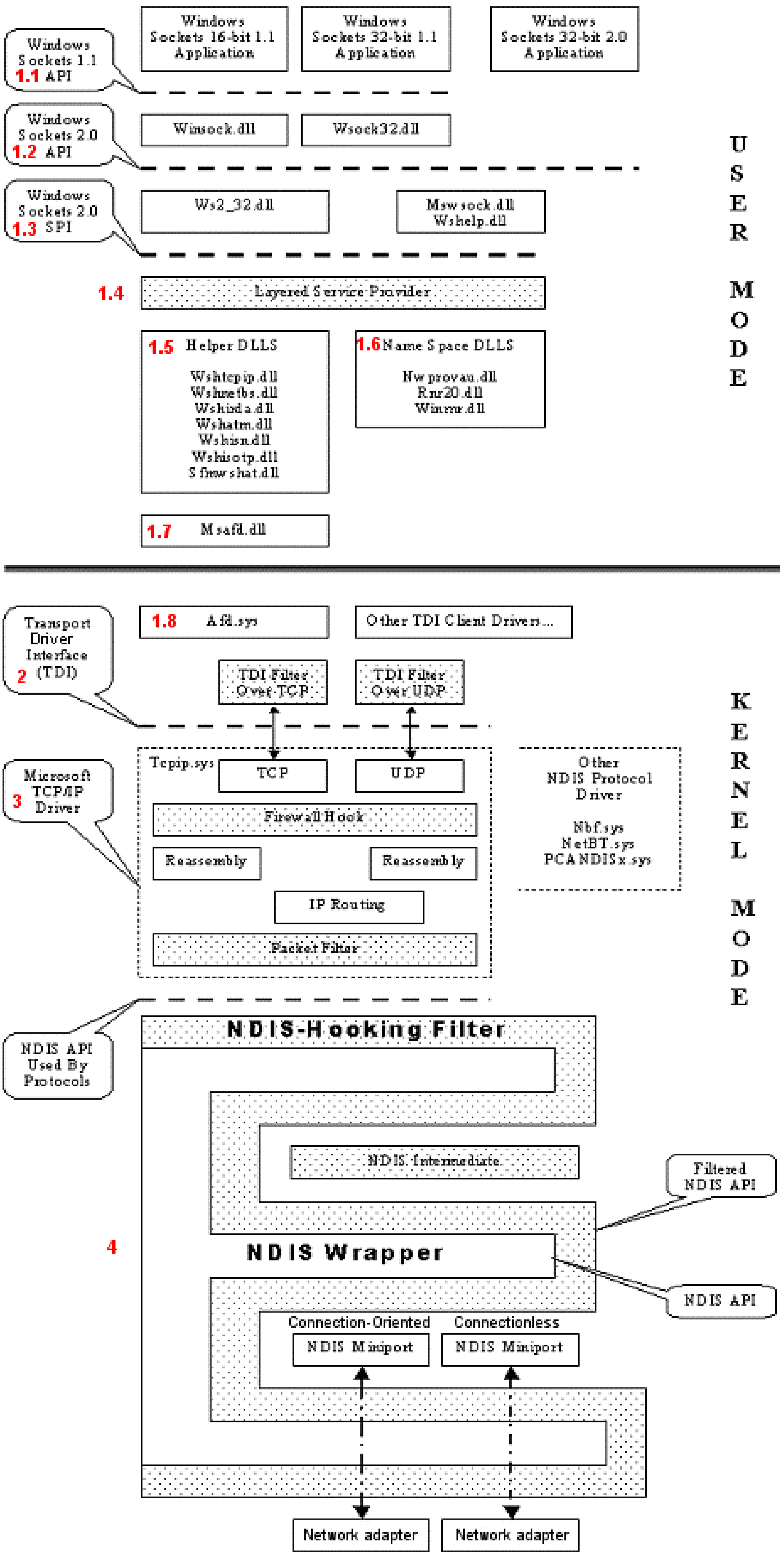


**Figure 1.** Windows 2000 network architecture diagram [9].

# 3. Methods to monitor network traffic

Two user mode methods, LSP (Layered Service Provider) and IP (Internet Protocol) Helper API, and a kernel mode method, TDI Firewall, will be examined in this chapter. These methods have been chosen because they are all located over the kernel mode TCP/IP driver (see 3 in Fig. 1). It is necessary to filter above the kernel mode TCP/IP driver to bind network operations to the associated Windows process. Filters below do not have visibility to process information because it is first at the TDI level (see 2 in Fig. 1) that the network traffic is connected to a process. [9]

## 3.1 Layered Service Provider

Hooking events such as window creation, keystrokes etc has been provided by the windows API from the dawn of the windows age. To monitor a user's activity on the Internet are a bit trickier. There is an API function, GetTcpTable, that will return all open ports on a system, but it does not tell anything about which applications have which ports open or the traffic that is flowing at the moment.

Winsock 2.0 gave birth to a framework that allows application developers to hook their code into the chain of network events. That makes it possible to spy on all Winsock network traffic. Winsock LSP is one method you can use which will run on almost all windows platforms. It is possible to get access to every network request from web browsers like Internet Explorer or any other major Internet aware application.

The LSP architecture does provide a good solution for simple URL filtering or Internet monitoring programs but it is important to remember that it only detects traffic through the Winsock layer. For example it will not detect when someone connects to a shared folder on your PC. Neither do it detect calls directly to the TDI layer. [11]

## 3.2 Internet Protocol Helper API

The IP) Helper API makes it possible to get and modify network configuration settings for a local computer. This API is designed for use in the C/C++ programming language and is supported on Microsoft operating systems but not all operating systems support all functions. The IP Helper API consists of the DLL file iphlpapi.dll that includes functions that can retrieve information about protocols, for example TCP and UDP. The next two chapters will describe how to retrieve the TCP connection table and how to bind each connection to a process. In the same way it is possible to retrieve a connection table for another protocol, for example UDP, and bind each UDP connection to a process. [29]

### 3.2.1 How to retrieve the TCP connection table

The file iphlpapi.dll includes a function called AllocateAndGetTcpExTableFromStack, which retrieves the TCP connection table. Compared to the original function GetTcpTable [29] this function includes the process id for each connection. This extended information is available only on XP and higher.

The first parameter is a pointer to a buffer that receives the TCP connection table as a MIB_TCPEXTABLE structure:

```
typedef struct
{
    DWORD          dwNumEntries;
    MIB_TCPEXROW   table[1];
}MIB_TCPEXTABLE, *PMIB_TCPEXTABLE;
```

This structure consists of a table of TCP connections. The first member is the number of entries in the table. The second member is a pointer to the table of TCP connections implemented as an array of MIB_TCPEXROW structures:

```
typedef struct
{
    DWORD   dwState;         // state of the connection
    DWORD   dwLocalAddr;     // address on local computer
    DWORD   dwLocalPort;     // port number on local computer
    DWORD   dwRemoteAddr;    // address on remote computer
    DWORD   dwRemotePort;    // port number on remote computer
    DWORD   dwProcessId;     // process id of the connection
}MIB_TCPEXROW, *PMIB_TCPEXROW;
```

The MIB_TCPEXROW structure contains information for a TCP connection. The first member dwState specifies the state of the connection, for example listening or established. The last member dwProcessId contain the process id of the connection. [12]

### 3.2.2 How to bind a TCP connection to a process and a parent process

With the function CreateToolhelp32Snapshot [30] (not a IP Helper function) it is possible to get a process snapshot. With the member TH32CS_SNAPPROCESS the function includes all processes in the system in the snapshot:

```
hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0)
```

Then it is possible to retrieve information about the first process encountered in the system snapshot with the function Process32First. To retrieve information about next process, use the function Process32Next:

```
Process32First(hProcessSnap, &pe32)
Process32Next(hProcessSnap, &pe32)
```

If a process from the snapshot belongs to a TCP connection (see Chap. 3.2.1) it is possible to get the name of the process and the parent process id:

```
if (pe32.th32ProcessID == tcpExTable->table[i].dwProcessId)
{
    strMsg.Format("%s %d", pe32.szExeFile, pe32.th32ParentProcessID);
}
```

Then a spyware program starts Internet Explorer the spyware program becomes the parent process to Internet Explorer. By comparing the parent id of the Internet Explorer process with every process id from a snapshot it is possible to get the name of the parent process, i.e. the spyware program. Note that it is not a guarantee to match a process id against a process name because a process could have exited and the process id gotten reused between our TCP connection and process snapshot. [12]

## 3.3 TDI Filtering

TDI Filtering can be done by intercepting the calls that are destined for the TCP/IP driver. In a device driver (see Appendix 1) you can use a function called IOAttachDevice on devices created by the TCP/IP driver. Thereby the driver gets the network traffic first and the filtering can be done by passing or not passing this on to the TCP/IP driver. An alternate approach is to replace the TCP/IP drivers dispatch table, an array of function pointers, with addresses to the TDI filter drivers' functions. This kind of technique is used in several commercial products, for example Outpost Firewall. [13]

# 4. Leak tests

Several applications have been developed to prove the weaknesses of personal firewalls (see Appendix 1). With most of the firewalls the user is asked whether applications are trusted or not to connect to the Internet. To avoid getting this question every time the web browser is started, the user can often configure the firewall to always trust the applications the user has selected. Except Yalta, all of the following leak tests try to bypass firewalls by using the trusted applications.

## 4.1 Tooleaky

Tooleaky was released by Bob Sundling in may 2001. His own words to why he wrote this leaktest is: "*an attempt to better educate users about how useless ALL of these firewall programs really are.*" By that time Tooleaky penetrated every firewall on the market. This was done in a remarkably easy way. Tooleaky just uses the WinExec API to spawn Internet Explorer in a hidden window, sends its information to the server and then retrieves a string from the server and displays it as the web page's title. It then closes the window and displays a message if it succeeded or not.[15]

## 4.2 Firehole

Robin Keir's firehole was the first leak test to perform DLL Injection (see Chap. 5). At first Firehole starts Internet Explorer with the ShellExecute API. Then it uses the SetWindowsHookEx API to load a DLL into the same process space as Internet Explorer. [22]

## 4.3 Thermite

Like Firehole, Thermite injects a DLL into Internet Explorer. But Thermite accomplishes this by creating an additional thread within that process. [18]

## 4.4 Copycat

The user is asked to choose which process Copycat should attack and then Copycat injects its code into that process. What the injected code does is to download a text file from a server. [19]

## 4.5 Yalta

Yalta (Yet another leak test application) has two different kinds of tests, a classic and an advanced one. The first tries to send UDP packets using standard sockets access. The second more advanced test goes below the TCP/IP layer to send UDP packets (Win95/98/Me only). [16]

## 4.6 Wallbreaker

Wallbreaker consist of three tests. The first one uses the fact that if you run explorer with a parameter that is an url, explorer will start Internet Explorer. By using this technique it can get pass firewalls that protect against applications that use the Internet directly or do it by launching a process that starts the web browser and then transmits the information. The second test starts Internet Explorer directly by using a ShellExecute API.

The third test adds another layer to test 1. Wallbreaker writes "explorer http://someurl" into a bat file that it executes, i.e cmd starts explorer that starts Internet Explorer. [20]

## 4.7 PC Audit and PC Audit2

PC Audit [17] does a DLL Injection to inject its code into a trusted application. PC Audit2

[21] uses a different way than the previous version to bypass firewalls with DLL protection.

### 4.8 Ghost

Ghost starts the default web browser, shuts itself down and restarts itself and continues to send data. The reason for this is that a firewall asks the user if the application that starts the web browser is a trusted application. The firewall gets the process ID and name and freezes the application until the user has decided what to do. Through shutting down and restart, the leak test application gets a different process ID. [14]

# 5. DLL and Code Injection

As seen in Table 1, some of the leak tests use DLL Injection, i.e. they inject a DLL Trojan into a trusted application. Let us take a deeper look at this technique.

All applications in Windows use functions in Dynamic Link Libraries. DLL Trojans take advantage of this, when they are injected into applications. Even though the Trojan DLL just is a module of the host process, it can be just as malicious as any other stand-alone trojan executable. The difference is that the process that has been infected will get the blame for whatever the DLL Trojan is up to.

The ability to inject DLL Trojans into trusted applications, gives the DLL Trojans the capability to sneak through routers and port-based firewalls as well as personal firewalls with application filtering rules.

There are two types of firewall tunnelling DLL Trojans, server DLLs and client DLLs. A Server DLL may be injected into a web server or any other application that the firewall has rules to accept incoming calls to. A tight firewall rule set may prevent server DLLs from abusing trusted server applications. For example a web server, which is only permitted to listen on a single port, is relatively safe because generally the DLL Trojan is unable to share the permitted port with the infected application. However many server applications, like file sharing tools, cannot have that tight rules.

Instead of listening for incoming connections, client DLLs establishes outgoing connections. Client DLLs are usually injected into trusted client applications, which are allowed to connect to the Internet. For example a client DLL can abuse the iexplore.exe process in order to connect to a website. Applications which want to establish outgoing connections in the windows OS get dynamically assigned to a local port. That makes it impossible to restrict outgoing connections to a single port and no firewall rule can prevent this kind of attack.

The fact that the DLL Trojan is not a stand-alone process makes it stealth. The Windows Task Manager merely lists processes but not modules and even with a tool that monitors the modules loaded by a process it is hard to detect a DLL Trojan. There can be hundreds of modules to inspect and there is no easy way to determine whether a DLL is malicious or not.

It is hard for firewalls to detect DLL Trojans and even if the firewall exposes a DLL Trojan the ordinary user may think that it is just a harmless “phone home” originating from a trusted application.

If you finally detect a DLL Trojan it can be very hard to get rid of it. They may have been injected into a critical system process, like the DLL Trojan Beast 2 that injects itself into the winlogon.exe process, and you cannot just kill a system process. Other Trojans complicate their removal by infecting almost every process they can find. [24]

## 5.1 Common injection techniques

There are two common ways to inject a DLL Trojan into a trusted host application, dynamic and static DLL Injection.

Dynamic DLL Injection is the most frequently used technique to activate DLL Trojans. The DLL Trojan has a loader application, which injects the Trojan DLL into a host application that was already running or which was previously started by the loader.

The loader application accomplishes this by allocating a memory region inside the host process. Then it writes to that memory region and forces the host process to load the Trojan DLL. The key functions to perform such an injection are VirtualAllocEx, WriteProcessMemory and CreateRemoteThread.

When the code required for loading the Trojan DLL is patched directly into the host, it is called Static DLL Injection. Unlike Dynamic DLL Injection this is a permanent infection and the Trojan DLL will load each time the infected application is started. Another advantage that

Static DLL Injection has is that it is not necessary to use suspicious functions like VirtualAllocEx and CreateRemoteThread. The reason to why these functions look suspicious is that applications, except some system level applications, have generally no legit reason to hijack other applications.  When performing static DLL Injection the standard LoadLibrary function can be used to load the DLL. To detect if a “new” application, for example a downloaded instant messenger, has been infected by means of static DLL Injection you have to compare it with the original, hopefully, uninfected version. A firewall that uses MD5 signature checks on executables and DLL files will detect if an existing application has been infected by means of static DLL Injection. [24]

## 5.2 Code Injection

Code injection is in many ways just like dynamic DLL Injection, i.e. a loader application allocates a memory region, by using VirtualAllocEx, inside the attacked process. Then it uses WriteProcessMemory to write to that memory region. But instead of forcing the attacked process to load a DLL, the malicious code is injected directly into the host process.

This technique has the obvious advantage that there is no need of an additional DLL. But it is also more complicated and riskier than DLL Injection. The host application will crash very easily and it is very tricky to inject just a few instructions. [7]

# 6. Program design

The program is developed in C/C++ for Microsoft Windows 2000/XP and is divided into three parts, one for spyware detection (see Chap. 6.1), one for stealth spyware detection (see Chap. 6.2) and one for viewing the log files (see Chap. 6.3). The first part monitors and filters all in- and outgoing TCP/UDP traffic from the local computer and binds it to the process that receives or sends the data. That makes it possible to detect if a spyware program or a usual program, for example Internet Explorer, receives or sends data. All information is saved to a log file. The second part of the program detects stealth spyware, like Firehole [22], by monitoring APIs. The third part views and monitors the log file in real-time.

## 6.1 Spyware detection

We have chosen to use an open source TDI firewall called "Simple TDI Based Open Source Personal Firewall" [13] in our program to monitor and filter network traffic. We have changed the way information is written to the log file to a format the log viewer can read. The firewall consists of two parts, a Windows service and a device driver.

The service connects to the driver, opens a channel for communication with it and listens for information about network events to be returned by the driver. It reads a configuration file containing the rules that states how the filtering should be done by the driver, which type of traffic is allowed, what events should be logged and these rules are then passed on to the driver. The service application handles the logging of events to a log file or to the event log depending on the settings in the configuration file. Communication with the driver is performed with the help of control codes that tells the driver what action to take. These are passed along with input buffer for input parameters and an output buffer for the return values from the driver in a function called DeviceIoControl. When it comes to notifying the service application about the network events an event object is used.

The device driver filters the network traffic according to the rules provided by the Windows service. It intercepts all calls to the TCP/IP driver and handles the network events in callback functions.

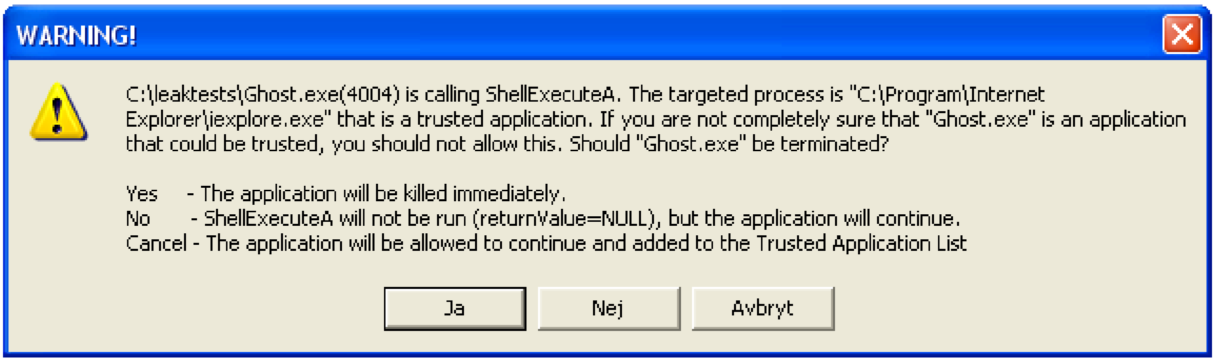
All in- and outgoing TCP/UDP traffic is monitored and saved to a log file. We have changed the format in which the data is written to the log file to a format the log viewer can read. Each entry in the log file now looks like this: *Timestamp, Deny/Allow, Protocol, Direction (In/Out), From IP address, To IP address, Process ID, Process name*.

## 6.2 Stealth spyware detection

The TDI firewall makes sure that only trusted applications are allowed to make network calls. However this is no guarantee for blocking spyware. For example a spyware can use a shell command, like ShellExecute, to launch Internet Explorer with an url that can take any arguments. As described in chapter 5, sophisticated spyware can use techniques like DLL and Code Injection to hijack trusted applications. To prevent this kind of attacks we have implemented an API Monitor that monitors functions that can be used to hijack an application. Our API Monitor has been based on Ivo Ivanov's "HookTool" Application [34].

The API Monitor itself uses DLL Injection, i.e it injects a DLL into the application that should be monitored. Then it is the DLL that initiates the monitoring.

When a new application is started our DLL should be injected inside it. That means that we must detect when a process is created. We have done this by implementing a device driver that uses the Kernel mode (see Chap. 2.2) function PsSetCreateProcessNotifyRoutine(). This function allows adding a callback function, that is called whenever a process is created or deleted. That gives us an opportunity to inject our DLL into the newly created application. By using the functions CreateRemoteThread, VirtualAllocEx and WriteProcessMemory our DLL is injected and loaded into the process space of that application. Now the procedure starts to do the actual function monitoring. The approach is to replace every function call that we want to monitor, with a copy of our own version of that function, this is called hooking. It is not enough to just hook the function in the process itself. The function call that should be hooked could also reside inside a DLL, which the process refers to. Because of that we have to iterate through all modules (DLLs) for a particular process, and make sure that all these modules call our function, instead of the original library's function. But we can still not be sure of that our function will be called instead of the original one. As described in Appendix 1, DLLs can be loaded in runtime. Because of that we have to hook all versions of the LoadLibrary functions (LoadLibraryA, LoadLibraryW, LoadLibraryEx). When they are called, i.e when our copies of the functions are called, we hook the functions in the requested DLL before returning the process handle. Another function that must be hooked is GetProcAddress. If GetProcAddress requests a hooked function, we should return the address to our copy of the requested function.



WARNING!

C:\leaktests\Ghost.exe(4004) is calling ShellExecuteA. The targeted process is "C:\Program\Internet Explorer\iexplore.exe" that is a trusted application. If you are not completely sure that "Ghost.exe" is an application that could be trusted, you should not allow this. Should "Ghost.exe" be terminated?

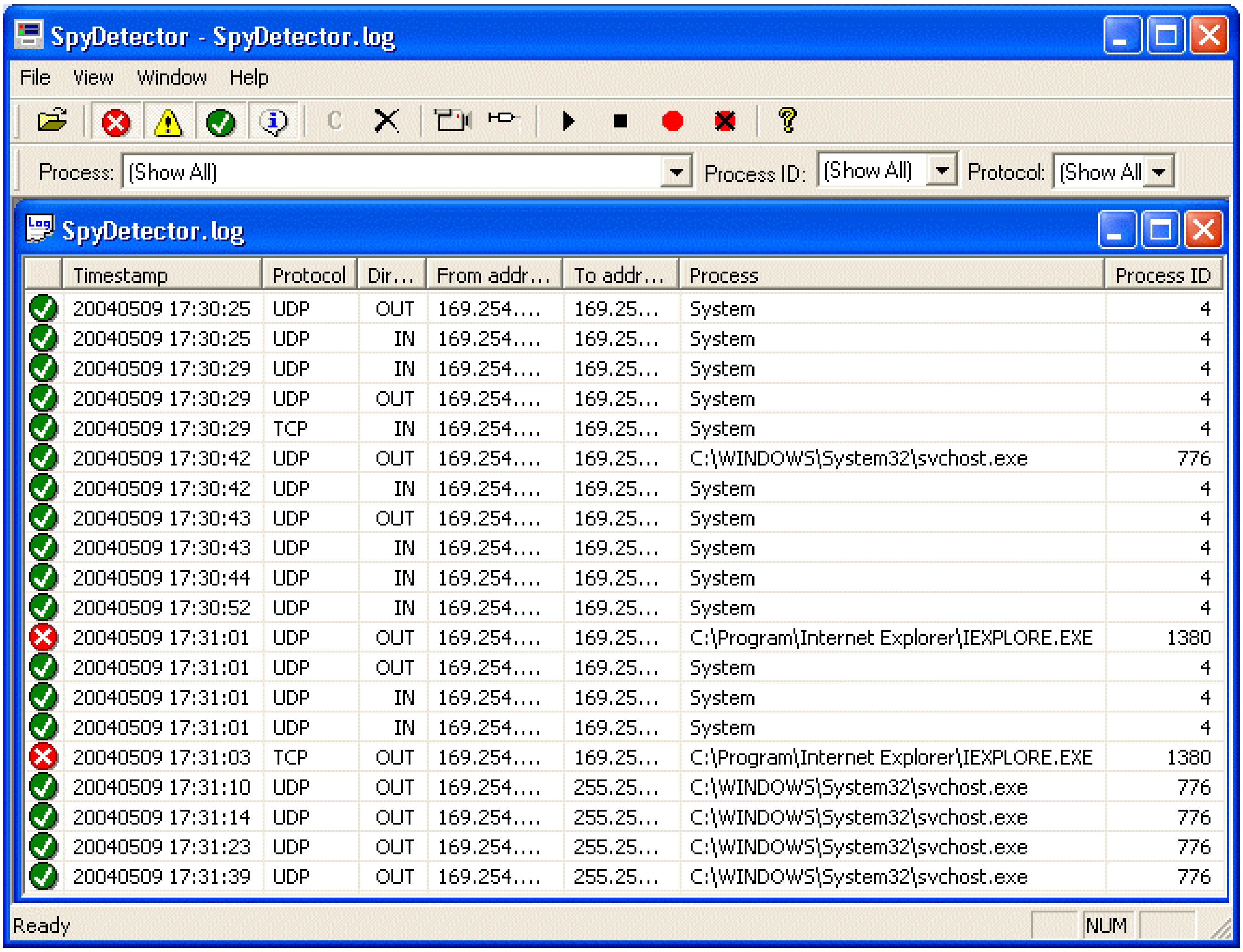
Yes    - The application will be killed immediately.
No     - ShellExecuteA will not be run (returnValue=NULL), but the application will continue.
Cancel - The application will be allowed to continue and added to the Trusted Application List

**Figure 2.** The API monitor warns the user for the leak test ghost.

The program monitors (hooks) functions that are used when performing DLL and Code Injection. It also monitors APIs like ShellExecute that are used to open applications. When a hooked function is called, for example ShellExecute, the program checks if the targeted application is one of the applications in the "trusted applications list". If it is not, the event will be written to a log file, but the user wont get noticed. Otherwise, i.e if the targeted application is a trusted application, the user gets a warning message (see Fig. 2) that shows the name of the application that is calling ShellExecute and which application it is targeting. Then it is up to the user to either allow or deny the function call. If the user allows the function call, and in that way adds the application to the trusted applications list, the log file will be searched to make sure that no there has not occurred any earlier manipulation to the newly added trusted application.

## 6.3 Log viewer

To view the log files we have chosen to use a current MDI (Multi Document Interface) project [31] that we have adapted to view our log files in real-time. In other words when a new entry is added to the opened log file by the TDI firewall, the opened log file is automatically updated and the new entry is shown. By naming the current log file *SpyDetector.log* and an aged log file *SpyDetector.yyyymmdd.log* the program knows which file to monitor for new entries.



Process: (Show All) Process ID: (Show All) Protocol: (Show All

| | Timestamp | Protocol | Dir... | From addr... | To addr... | Process | Process ID |
|---|---|---|---|---|---|---|---|
| ✓ | 20040509 17:30:25 | UDP | OUT | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:25 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:29 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:29 | UDP | OUT | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:29 | TCP | IN | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:42 | UDP | OUT | 169.254.... | 169.25... | C:\WINDOWS\System32\svchost.exe | 776 |
| ✓ | 20040509 17:30:42 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:43 | UDP | OUT | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:43 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:44 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:30:52 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✗ | 20040509 17:31:01 | UDP | OUT | 169.254.... | 169.25... | C:\Program\Internet Explorer\IEXPLORE.EXE | 1380 |
| ✓ | 20040509 17:31:01 | UDP | OUT | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:31:01 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✓ | 20040509 17:31:01 | UDP | IN | 169.254.... | 169.25... | System | 4 |
| ✗ | 20040509 17:31:03 | TCP | OUT | 169.254.... | 169.25... | C:\Program\Internet Explorer\IEXPLORE.EXE | 1380 |
| ✓ | 20040509 17:31:10 | UDP | OUT | 169.254.... | 255.25... | C:\WINDOWS\System32\svchost.exe | 776 |
| ✓ | 20040509 17:31:14 | UDP | OUT | 169.254.... | 255.25... | C:\WINDOWS\System32\svchost.exe | 776 |
| ✓ | 20040509 17:31:23 | UDP | OUT | 169.254.... | 255.25... | C:\WINDOWS\System32\svchost.exe | 776 |
| ✓ | 20040509 17:31:39 | UDP | OUT | 169.254.... | 255.25... | C:\WINDOWS\System32\svchost.exe | 776 |

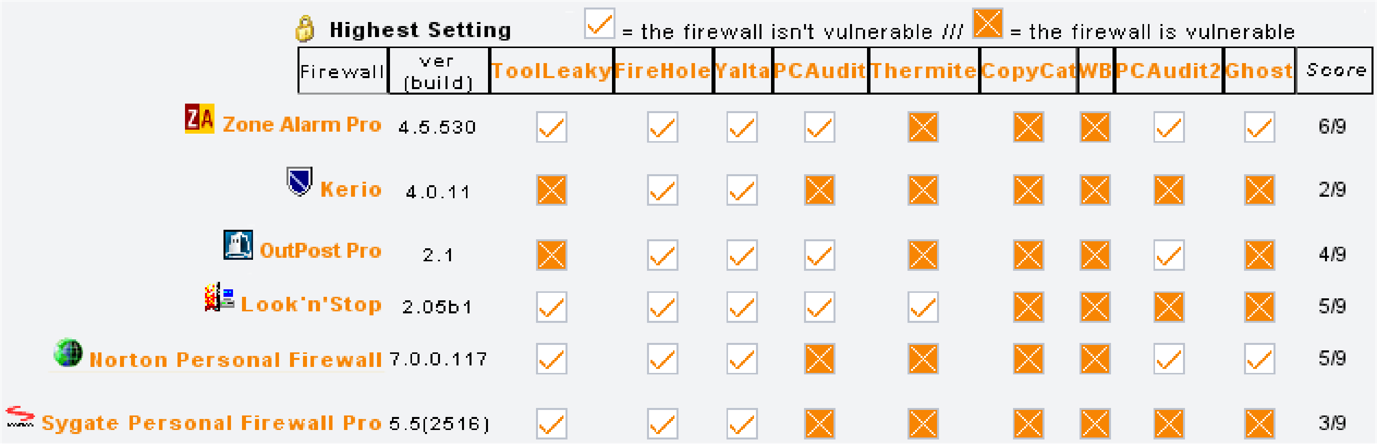
**Figure 3.** The GUI of our program SpyDetector.

When a log file is opened (see Fig. 3) the first column shows, with an icon, if a process is allowed or denied to communicate. By just clicking on a specific icon in the toolbar, allowed or denied processes are shown in the view. With the combo boxes it is possible to just view entries with a specific process, process id or protocol. In the toolbar we have added some buttons that call our events for the firewall and the API monitor. The camera button starts or stops the API monitor i.e. the method to detect stealth spyware. By pressing the button with a sprayer you get a question if you want to inject our DLL file that does the API-hooking in each of the running processes. To install the firewall as a Windows service press the button with a red dot and to uninstall the service press the button with a red dot with a black cross over. The play button starts the service and the stop button stops the service.

We have added a new functionality to the log viewer. All programs that have permission to send or receive data are included in the firewall's configuration file. If a program, that is not included in the configuration file, tries to send or receive data it is denied to communicate and the user gets a question if he wants to add the program's path to the configuration file. If the user answer yes the path is added to the configuration file. If the answer is no, the path will be added to a file containing programs that do not have permission to send or receive data.

## 7. Test results

The tests shown in figure 4 are done by Guillaume Kaddouch [28]. The newest versions of the most common firewalls have been tested against nine leak tests (see Chap. 4).



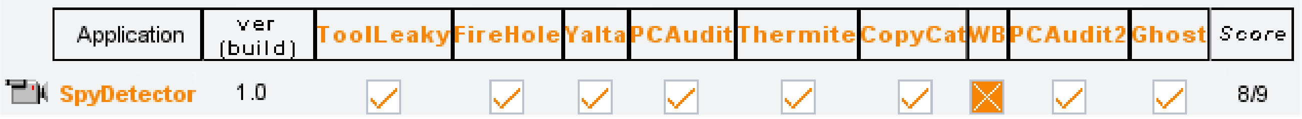
Highest Setting ✓ = the firewall isn't vulnerable /// ☒ = the firewall is vulnerable

| Firewall | ver (build) | ToolLeaky | FireHole | Yalta | PCAudit | Thermite | CopyCat | WB | PCAudit2 | Ghost | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Zone Alarm Pro | 4.5.530 | ✓ | ✓ | ✓ | ✓ | ☒ | ☒ | ☒ | ✓ | ✓ | 6/9 |
| Kerio | 4.0.11 | ☒ | ✓ | ✓ | ☒ | ☒ | ☒ | ☒ | ☒ | ☒ | 2/9 |
| OutPost Pro | 2.1 | ☒ | ✓ | ✓ | ✓ | ☒ | ☒ | ☒ | ✓ | ☒ | 4/9 |
| Look'n'Stop | 2.05b1 | ✓ | ✓ | ✓ | ✓ | ✓ | ☒ | ☒ | ☒ | ☒ | 5/9 |
| Norton Personal Firewall | 7.0.0.117 | ✓ | ✓ | ✓ | ☒ | ☒ | ☒ | ☒ | ✓ | ✓ | 5/9 |
| Sygate Personal Firewall Pro | 5.5(2516) | ✓ | ✓ | ✓ | ☒ | ☒ | ☒ | ☒ | ☒ | ☒ | 3/9 |

**Figure 4.** Test results for the most common firewalls.

The ☒ symbol means that the leak test went through the firewall's outbound protection unnoticed. The ✓ symbol says that the firewall detected the leak test and asked the user if the leak test should be able to send data. Although highest security settings none of the firewalls protects against all the threats demonstrated by the leak tests. With "out of the box", i.e default settings, the firewalls perform even worse.

The test results for our program, SpyDetector, are shown in figure 5.



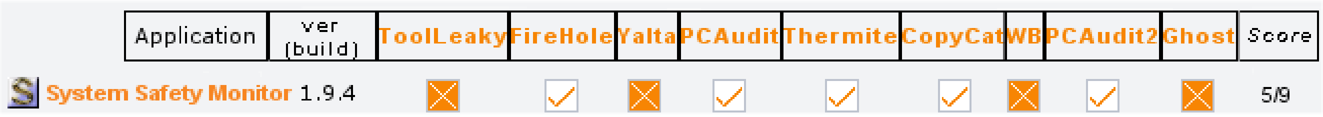
| Application | ver (build) | ToolLeaky | FireHole | Yalta | PCAudit | Thermite | CopyCat | WB | PCAudit2 | Ghost | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SpyDetector | 1.0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ☒ | ✓ | ✓ | 8/9 |

**Figure 5.** Test results for our program.

The firewall detects Yalta's first test as it tries to send UDP packets. Firehole and Ghost are detected by the API Monitor when they try to launch Internet Explorer by calling the ShellExecute API. TooLeaky is detected in the same way when it uses the Winexec API. PC Audit and PC Audit2 are both detected when they try to do DLL Injection by using the SetWindowsHookEx API. Thermite and CopyCat are detected when they call OpenProcess API (CopyCat uses NtOpenProcess API see Chap. 8) to get the handle to Internet Explorer that is its target for the Code Injection (see Chap. 5). Wallbreaker's third test (WB->cmd->explorer->iexplore) is not detected by our program.

SSM, System Safety Monitor, is an application monitor software that has got good critic. SSM uses sandbox measures (see Appendix 1) but it also monitors the programs that the user has allowed to start. Like our program, it hooks certain APIs to prevent "hijacking" of other applications. It is that functionality that we have tested, i.e we allowed the leak tests to run and to pass the test SSM had to popup a warning message when the leak tests did the actual "hijacking".

SSM detects every leak test that tries to do DLL or code injection (see Fig. 6). But SSM



| Application | ver (build) | ToolLeaky | FireHole | Yalta | PCAudit | Thermite | CopyCat | WB | PCAudit2 | Ghost | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| System Safety Monitor | 1.9.4 | ☒ | ✓ | ☒ | ✓ | ✓ | ✓ | ☒ | ✓ | ☒ | 5/9 |

**Figure 6.** Test results for System Safety Monitor.

does not monitor the APIs that Tooleaky, Ghost and Wallbreaker use to start Internet Explorer. Of course it does not detect when Yalta sends the UDP packets because SSM does not filter any network

# 8. Discussion and Conclusion

To detect if it is a trusted or a non-trusted application, for example a spyware, that communicates it is necessary to bind in- and outgoing network traffic on the local computer to an associated process. To do this it is necessary to filter above the kernel mode TCP/IP driver. Filters above the kernel mode TCP/IP driver (see 3 in Fig. 1) can correlate network operations with the associated Windows process. Filters below do not have visibility to process information because it is first at the TDI level (see 2 in Fig. 1) that the network traffic is connected to a process.

At the beginning we just wanted to monitor network traffic and bind it to an associated process. This can be done from user mode with for example the IP Helper API (see Chap. 3.2). But after a while we realised that we also needed to filter network traffic to make it easier to detect spyware programs (see Chap. 6.1). Because when the user has to add the applications that should be able to access the network to a "trusted application list", we know which applications that should be protected from "hijacking". It is possible to filter network traffic from user mode with no need of writing any drivers in kernel mode. But spyware programs can bypass Winsock and use TDI directly, which makes it more secure to filter in kernel mode with a TDI filter driver. Therefore we have chosen to use a TDI firewall that uses a TDI filter driver. There are limitations to this method as well. It is still possible for an application to bypass the TCP/IP stack in order to send information. But we have not found any application that does that and because of that we will not examine this more in detail.

In addition to this we also wanted to detect Stealth spyware. At first we tried to accomplish this by monitoring the parent processes to the trusted applications. By doing that we were able to detect leak tests that opens the trusted application with a shell command. But we had no chance to detect leak tests that use DLL Injection or Code Injection. A method to detect that is to monitor when the APIs, which are used to do DLL and Code Injection, are called. After some studying about DLL Injection we found out that an excellent way to monitor any application is to inject a DLL into it. Now we could hook different API:s, i.e replace any function with our own function. That makes it possible to discover when an application is preparing to attack another application.

As shown in figure 4, very simple leak tests, like TooLeaky, is able to sneak through some firewalls outbound protection. That despite the fact that TooLeaky was released for more than three years ago, May 2001. TooLeaky could be detected very easily by just making sure that the parent process to Internet Exlorer is a trusted application. It is easy to get the impression that firewall developers do not use their big guns when it comes to outbound protection.

To use sandbox measures can seem to be the ultimate solution to avoid spyware, i.e the user tells which applications that should be allowed to run and the rest is blocked. But in that case the user has to make a rule for every single application that should be run and the ordinary user may "mistakenly" allow a malicious program to run. Therefore it is good to have a program like System Safety Monitor that also monitors the trusted applications. However one disadvantage with programs like SSM is that the warning messages never seem to end.

To prevent spyware, we only need to warn the user if a trusted application that is allowed to use the network is "attacked". When using our solution the only programs that need to be trusted are those that should be able to make network calls. Then it is up to our API Monitor to protect these applications from being "hijacked".

As seen in figure 5, our program detects all the tested leak tests except Wallbreaker. Wallbreaker's trick is to add several layers before executing Internet Explorer. This is done by using a "feature" in Windows. When explorer is started with a parameter that is an url, Internet Explorer is started and connects to that url. To add yet another layer, Wallbreaker writes the line "explorer http://UrlWithAnyArguments" into a bat file. When that bat file is executed cmd starts, then cmd starts explorer that starts Internet Explorer. When explorer executes Internet Explorer, cmd has already terminated and the connection to Wallbreaker is broken. One approach to detect that it was Wallbreaker that initiated the network call, is to save all the steps between the parent- and child applications. It would then be possible to warn the user if the initial parent process is not an application that should be allowed to make network calls.

One disadvantage with the API hooking method is that it is hard to know which APIs that should be hooked. This problem became obvious when we tried to find out which API that the leak test CopyCat used to do the Code Injection. We knew that copycat had to open the targeted process to be able to perform the injection and finally we found out that it uses NtOpenProcess that is an undocumented function in Ntdll.dll. Undocumented functions may be changed between one release of Windows to the next and possibly even between service packs for each release. That is why they are undocumented, developers should not be encouraged to use them. However nothing stops spyware developers from using that kind of functions. This means that an application that hooks APIs, like ours do, has to be regularly updated with the newest APIs that could be used to hijack applications.

As discussed in this thesis firewalls lack in their outbound protection and spyware can by rather simple means penetrate the firewalls currently on the market. Because of that we think that it is necessary to use an additional program that does application monitoring to be well protected against spyware.

# 9. Further work

As mentioned in the discussion, one approach to detect spyware that acts like Wallbreaker could be to save a list that includes all steps between a parent and the child application. By iterating through that list it would then be possible to make sure that the initial parent process, to the application that wants to connect to the network, is a trusted application. If that is correctly implemented it may not longer be necessary to hook APIs that is used to start other applications, like ShellExecuteEx. That would be great because there are many different ways in which these API:s could be used to start another program and it is very hard to foresee every way that these APIs could be used.

When a “new” application tries to access the network our firewall blocks it and then asks if it should be allowed or blocked the next time. Instead of blocking the process the first time, it would be better to suspend it while asking the user if it should be blocked or allowed.

When our DLL is injected into a process, that process should also be suspended until the API hooking is completed. That would make sure that an API that should be hooked could not be called before it has been hooked. The SystemProcessInformation class can be used to get information about the process. Then OpenThread could be used to return a handle for the threads in the process. These handles could then be suspended by using the SuspendThread API.

# 10. References

/1/ Jones, A., Ohlund, J. (2002). *Network Programming for Microsoft Windows* (2nd ed.). Washington: Microsoft Press.

/2/ Tanenbaum, A. S. (1996). *Computer Networks* (3rd ed.). New Jersey: Prentice Hall.

/3/ Comer, D. E. (2001). *Computer Networks and Internet* (3rd ed.). New Jersey: Prentice Hall.

/4/ Whatis.com, (Mar 18, 2004). *spyware.*
http://searchcio.techtarget.com/gDefinition/0,294236,sid19_gci214518,00.html

/5/ Whatis.com, (May 18, 2004). *adware*.
http://whatis.techtarget.com/definition/0,289893,sid9_gci521293,00.html

/6/ Shank, D. (2001, March, 12). Office VBA and the Windows API. Microsoft Developer Network (MSDN)

/7/ Kuster, Robert. (July, 2003). *Three Ways to Inject Your Code into Another Process.*
http://www.codeproject.com/threads/winspy.asp

/8/ Microsoft Windows 2000 Server Resource Kit. (2002). *TCP/IP Core Networking Guide.*

/9/ NDIS.com, (March 13, 2003). Windows Network Data and Packet Filtering.
http://www.ndis.com/papers/winpktfilter.htm

/10/ sockaddr.com. (March 2003). *WinSock Version Differences.*
http://www.sockaddr.com/VersionDifferences.html

/11/ Mitcell, Eric. (March 2003). *A Winsock Layered Service Provider.*
http://www.ericmitchell.net/code/winsock_lsp.shtml

/12/ Russinovich, M. (May 19, 2004). *TCPView.*
http://www.sysinternals.com/ntw2k/source/tcpview.shtml

/13/ Staricyn, R. (October 24, 2003). *Project: tdi_fw: Summary.*
http://sourceforge.net/projects/tdifw

/14/ Guillaume Kaddouch, (December 2003). *Ghost.*
http://perso.wanadoo.fr/jugesoftware/firewallleaktester/eng/leaktest13.htm

/15/ Sundling, Bob. (November 5, 2001). *LeakTest.* http://tooleaky.zensoft.com

/16/ soft4ever. (2000-2001). *YALTA.* http://www.soft4ever.com/security_test/En/index.htm

/17/ Guillaume Kaddouch, (December 2003). *PC Audit.*
http://perso.wanadoo.fr/jugesoftware/firewallleaktester/eng/leaktest6.htm

/18/ Guillaume Kaddouch, (December 2003). *Thermite.* http://perso.wanadoo.fr/jugesoftware/firewallleaktester/eng/leaktest8.htm

/19/ Guillaume Kaddouch, (December 2003). *CopyCat.* http://perso.wanadoo.fr/jugesoftware/firewallleaktester/eng/leaktest9.htm

/20/ Guillaume Kaddouch, (December 2003). *Wallbreaker.* http://perso.wanadoo.fr/jugesoftware/firewallleaktester/eng/leaktest11.htm

/21/ Guillaume Kaddouch, (December 2003). *PC Audit2.* http://perso.wanadoo.fr/jugesoftware/firewallleaktester/eng/leaktest12.htm

/22/ Keir, Robin. (25 March 2002). *FireHole.* http://keir.net/firehole.html

/23/ Microsoft Platform SDK: DLLs. Processes, and Threads, October 2002. About Dynamic-Link Libraries. (MSDN)

/24/ ntl. (10 August 2003). Wolves In Sheep's Clothing: Malicious DLLs Injected Into Trusted Host Applications. http://home.arcor.de/scheinsicherheit/dll.htm

/25/ Shaw, G, 2003. "SpyWare & Adware: the Risks facing Business". Network Security, Issue 9, pages 12-14

/26/ Radcliff, Deborah, 2004. "Spyware". Network World, Issue 4, pages 51-52

/27/ Whatis.com. (Jul 31, 2001). *personal firewall***.** http://searchsecurity.techtarget.com/sDefinition/0,,sid14_gci331881,00.html

/28/ Guillaume Kaddouch, (19 May 2004). *Firewall Leak Tester.* http://perso.wanadoo.fr/jugesoftware/firewallleaktester/eng/index.html

/29/ Microsoft Platform SDK: Internet Protocol Helper, October 2002. (MSDN)

/30/ Microsoft Platform SDK: Performance Monitoring, October 2002. (MSDN)

/31/ Manderson, B. (25 February 2004). *A realtime logfile viewer.* http://www.codeproject.com/cpp/logviewer.asp

/32/ Maxcomputing. (1 December 2003). *System Safety Monitor*. http://maxcomputing.narod.ru/ssme.html

/33/ Viega, J., McGraw, G. (2001). *Building Secure Software*. Boston: Addison-Wesley.

/34/ Ivano, I.,(6 May 2004). www.infoprocess.biz

/35/ Allan, M., (6 May 2004). http://fismat.umich.mx/ls/win/win2kguide.pdf

# Appendix 1.

## Definitions

### Adware

Adware is software that displays advertising banners when it is being run. Shareware vendors are often sponsored by advertising companies to add additional code to display these ads. If the user wants to get rid of the ads, he must buy the software. There are also adware that gathers information about the user and sends it to third parties. The difference is that software considered as spyware does not inform the user about its activities and for what purpose the information is gathered. [5]

### Device drivers

Device drivers are programs that control devices that are attached to the computer. These devices can be for example printers, diskette drives and network interface cards. The driver converts general input/output requests to messages that the device can understand. Communication between the application and the driver is done by using control codes (ioctls) through which the application can control the driver.

### DLL

DLLs, Dynamic Link Libraries, can be described as code libraries, which contain functions that can be called from any application running in Windows. There are two different ways for calling the functions inside the DLLs.

In load-time dynamic linking you have to link the application with the import library for the DLL that contains the functions. The import library contains information needed to load the DLL and to locate the exported DLL functions when the application is loaded.

In run-time dynamic linking, the LoadLibrary function is used to load the DLL at run time. When the DLL is loaded the GetProcAddress is called to get the addresses of the exported DLL functions. [23]

### Personal Firewall

Personal firewalls are software that runs on a single computer connected to the Internet and filters inbound and outbound network traffic. When suspicious network activities occur the application alerts the user and asks him to make a decision if the activity should be blocked or allowed. [27]

### Sandbox

When a system runs in a sandbox the behaviour of all processes is restricted based on a security policy that the user sets. For example the user can set a security policy that forces the user to either allow or deny a process to be started by another process. [33]

### Spyware

Spyware is software that gathers information from a computer without the owners' knowledge and consent. The information is then relayed to advertisers and other interested parties. The software often gets into computers as computer virus or is installed with shareware. [4]

### Windows API

An API (application programming interface) is a software interface for components, applications, and operating system just like a number keypad is an interface for a calculator.

A Windows API consists of one or more DLL files that provide some specific functionality. For example, the common dialog boxes (Open, Save, etc.), the Windows shell, operating system settings, and even windows themselves are provided by the Windows API. [6]

**Winsock**

WinSock (Windows Sockets) is an API that provides an interface between Windows based applications and network transport protocols. Through this programming interface applications can communicate using popular networks protocol, such as TCP/IP (Transmission Control Protocol/Internet Protocol) and IPX (Internetwork Packet Exchange). Winsock supports both connection-oriented and connectionless protocols. [1]