MACIEJ PAKULSKI

Przejmowanie wiadomości Gadu-Gadu

Stopień trudności

Komunikatory internetowe są jednym z coraz częściej wykorzystywanych sposobów komunikacji międzyludzkiej. Za ich pomocą prowadzimy luźne pogawędki ze znajomymi, załatwiamy sprawy zawodowe, a nawet wyznajemy uczucia drugiej osobie. Czy jednak możemy być pewni tego, że nikt nas nie podsłuchuje?

Jedną z metod przechwycenia wiadomości jest zastosowanie *keyloggera*. W numerze styczniowym w artykule *C#.NET. Podsłuchiwanie klawiatury* przedstawiony został projekt podstawowego *keyloggera* dla systemu Windows. Takie rozwiązanie daje nam jednak możliwość odczytania wyłącznie danych wysyłanych przez klienta komunikatora do serwera. Aby mieć dostęp do danych zarówno wysyłanych, jak i odbieranych z serwera, musimy spróbować przechwycić je z sieci. Komunikacja sieciowa oparta jest na gniazdach. Gniazdo (ang. *socket*) jest abstrakcyjnym dwukierunkowym punktem końcowym procesu komunikacji. Dwukierunkowość oznacza możliwość i odbioru, i wysyłania danych. Pojęcie gniazda wywodzi się od twórców systemu Berkeley UNIX. Istnieje możliwość utworzenia tzw. surowych gniazd (ang. *raw sockets*), dzięki którym mamy dostęp do całości pakietu danych, tzn. wraz z nagłówkiem IP, TCP itd. Uzyskujemy w ten sposób możliwość monitorowania przepływu danych w sieci. Surowe gniazda są w pełni obsługiwane przez system Windows XP SP2. Nowy produkt Microsoftu – Windows Vista – w obecnej wersji ogranicza ich działanie. Można zadać sobie pytanie, czy ograniczenia wprowadzone w Windows Vista są próbą walki z hakerami czy tylko błędem, który zostanie w przyszłości poprawiony. Zwłaszcza, iż według nieoficjalnych zapowiedzi developerów z Redmond, surowe gniazda mają być w pełni dostępne wraz z nadejściem *service packa* dla systemu Windows Vista.

Projekt aplikacji przechwytującej wiadomości komunikatora internetowego zrealizujemy korzystając z platformy .NET, która dostarcza wygodnego zbioru typów pozwalającego w prosty sposób na wykonanie tego zadania. Jako komunikator wykorzystamy klienta sieci Gadu-Gadu.

Gadu-Gadu – wysyłanie i odbieranie wiadomości

Jednym z najpopularniejszych polskich komunikatorów internetowych jest Gadu-Gadu. Jest on dostępny na polskim rynku od 2000 roku. Przez ten czas zyskał sobie wielką popularność i obecnie posiada liczbę użytkowników szacowaną na kilka milionów. Wiele innych komunikatorów umożliwia swoim użytkownikom komunikację nie tylko z własnymi dedykowanymi serwerami, ale również z serwerami sieci Gadu-Gadu.

Klient Gadu-Gadu bazuje na protokole TCP/IP. Połączenie jest realizowane z wykorzystaniem portu 8074 bądź 443. Dane wysyłane są w postaci pakietów, rozpoczynających się od nagłówka, którego postać przedstawia Listing 1.

Pole `type` określa typ wysyłanego pakietu, pole `length` – długość pakietu bez nagłówka, wyrażoną w bajtach. Po nagłówku znajdują się właściwe dane pakietu.

Gdy klient otrzymuje wiadomość, wówczas pole `type` przyjmuje wartość 0x0a. Po nagłówku pakietu wysyłana jest struktura przedstawiona na Listingu 2.

Z ARTYKUŁU DOWIESZ SIĘ

jak protokół Gadu-Gadu obsługuje wysyłanie/odbieranie wiadomości,

jak działają surowe gniazda (ang. raw sockets).

CO POWINIENEŚ WIEDZIEĆ

znać podstawy projektowania zorientowanego obiektowo,

znać podstawy działania sieci komputerowych.

Pole `sender` to numer nadawcy wiadomości, `seq` jest numerem sekwencyjnym, `time` – czasem nadania wiadomości, `class` – klasą wiadomości, a `msg` jest wysyłaną wiadomością. Aby określić długość wiadomości, musimy od pola `length` nagłówka odjąć ilość bajtów zajmowaną przez dane pakietu bez wiadomości (tj. 16 bajtów). Wiadomości są kodowane przy użyciu strony kodowej Windows-1250.

Gdy klient chce wysłać wiadomość, wówczas pole `type` nagłówka przyjmuje wartość 0x0b. Oprócz nagłówka wysyłana jest struktura zdefiniowana na Listingu 3.

Pole `recipient` określa numer odbiorcy, `seq` jest numerem sekwencyjnym, `class` – klasą wiadomości, a `msg` wysyłaną wiadomością.

Protokół IP

Protokół IP wykorzystuje się do sortowania oraz dostarczania pakietów. Pakiety zwane są datagramami. Każdy taki datagram składa się z nagłówka oraz ładunku, zawierającego przeważnie pakiet innego protokołu. Obecnie używa się protokołu IP w wersji 4.

W nagłówku datagramu IPv4 możemy wyróżnić następujące pola:

- *Version* – wersja protokołu,
- *IHL* (*Internet Header Length*) – długość nagłówka, wyrażona jako ilość 32-bitowych słów,
- *TOS* (*Type of Service*) – typ obsługi określający sposób, w jaki powinien być obsłużony pakiet,
- *Total Length* – całkowita długość datagramu IP w bajtach,
- *Identification* – identyfikuje określony datagram IP,
- *Flags* – pole to zawiera 3 bity, jednak wykorzystuje się tylko dwa spośród nich. Jedna z flag określa, czy pakiet może być fragmentowany, druga zaś – czy jest to już końcowy fragment w datagramie, czy może jest jednak więcej fragmentów,
- *Fragment Offset* – określa pozycję fragmentu w stosunku do oryginalnego ładunku IP,
- *TTL* (*Time to Live*) – określa czas (w sekundach), przez jaki datagram pozostanie w sieci, zanim zostanie odrzucony,
- *Protocol* – określa typ protokołu będącego ładunkiem datagramu IP,
- *Header Checksum* – suma kontrolna nagłówka, wykorzystywana w celu sprawdzenia jego poprawności,
- *Source IP Address* – źródłowy adres IP,
- *Destination IP Address* – docelowy adres IP.

Protokół TCP

Protokół TCP jest jednym z najczęściej używanych protokołów komunikacyjnych w sieciach komputerowych. Jest to protokół połączeniowy, w którym to przed rozpoczęciem komunikacji wymagane jest zainicjowanie połączenia przez jedną ze stron. Dane przesyłane są w postaci segmentów składających się każdorazowo z nagłówka oraz danych.

Nagłówek TCP możemy podzielić na następujące pola:

- *Source Port* – port źródłowy,
- *Destination Port* – port docelowy,
- *Sequence Number* – służy do identyfikacji pierwszego bajtu danych w danym segmencie,
- *Acknowledgment Number* – określa numer sekwencyjny bajtu oczekiwanego przez nadawcę,
- *Data Offset* – długość nagłówka mierzona jako ilość 32-bitowych słów,
- *Reserved* – pole zarezerwowane do przyszłego wykorzystania,

Listing 1. *Nagłówek pakietu Gadu-Gadu*

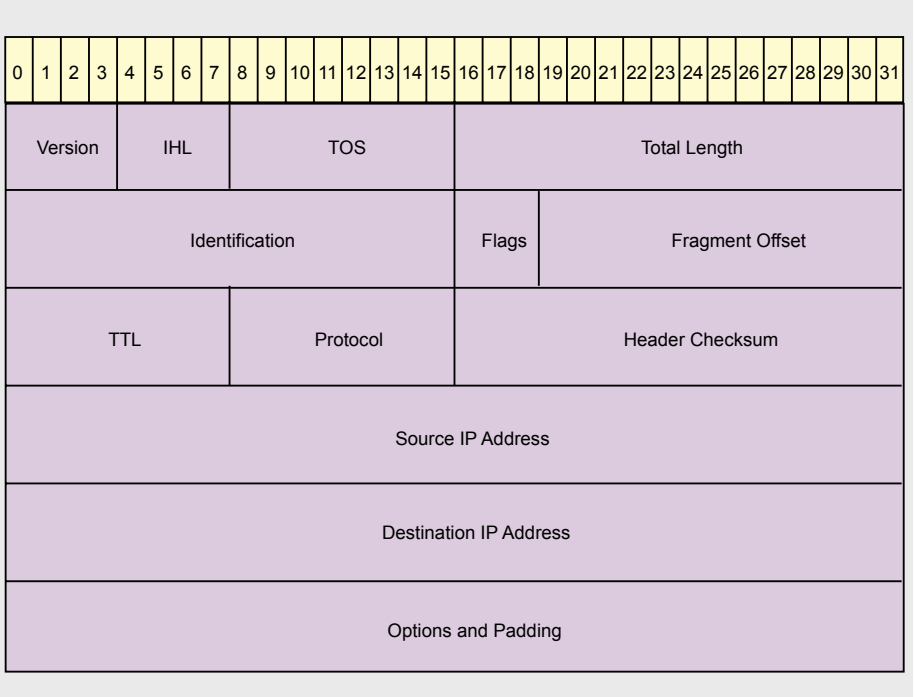
```
struct GGHeader
{
        int type;
        int length;
   }
;
```

Listing 2. *Struktura reprezentująca wiadomość odbieraną przez klienta Gadu-Gadu*

```
struct RecvMsg
{
        int sender;
        int seq;
        int time;
        int class;
        string msg;
};
```

Listing 3. *Struktura reprezentująca wiadomość wysyłaną przez klienta Gadu-Gadu*

```
struct SendMsg
{
        int recipient;
        int seq;
        int class;
        string msg;
};
```



Rysunek 1. *Nagłówek IPv4*

- *Flags* – flagi zawierające informację o przeznaczeniu segmentu,
- *Window* – określa ilość danych, jaką nadawca jest gotów przyjąć,
- *Checksum* – suma kontrolna wykorzystywana do sprawdzenia poprawności transmisji,
- *Urgent Pointer* – pole wykorzystywane do wyróżnienia szczególnie ważnych wiadomości.

Listing 4. *Pola klasy IPHeader*

```
private byte versionAndHeaderLength;
private byte typeOfService;
private ushort totalLenth;
private ushort identification;
private ushort flagsAndOffset;
private byte tTL;
private byte protocolType;
private short checkSum;
private uint sourceAddress;
private uint destinationAddress;
private byte headerLength;
private byte[] ipData = new byte[4096];
```

Listing 5. *Konstruktor klasy IPHeader*

```
public IPHeader(byte[] dataReceived, int received)
{
   try
   {
      MemoryStream sm = new MemoryStream(dataReceived, 0, received);
      BinaryReader br = new BinaryReader(sm);
      versionAndHeaderLength = br.ReadByte();
      typeOfService = br.ReadByte();
      totalLenth = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      identification = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      flagsAndOffset = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      tTL = br.ReadByte();
      protocolType = br.ReadByte();
      checkSum = IPAddress.NetworkToHostOrder(br.ReadInt16());
      sourceAddress = (uint)(IPAddress.NetworkToHostOrder(br.ReadInt32()));
      destinationAddress = (uint)IPAddress.NetworkToHostOrder(br.ReadInt32());
      headerLength = versionAndHeaderLength;
      headerLength = (byte)((headerLength & 0x0f) * 4);
      Array.Copy(dataReceived, headerLength, ipData, 0, totalLenth - headerLength);
   }
   catch (Exception exc){}
}
```

Listing 6. *Właściwości klasy IPHeader*

```
public byte[] Data
{
   get{
      return ipData;
   }
}
public EProtocolType TypeOfProtocol
{
   get
   {
      if (protocolType == 6)
         return EProtocolType.TCP;
      return EProtocolType.OTHER;
   }
}
public int MessageLength
{
   get
   {
      return totalLenth - headerLength;}
}
```

Tworzymy klasę przechwytującą pakiety GG

Projekt aplikacji umożliwiającej przechwytywanie pakietów Gadu-Gadu stworzymy używając języka C# oraz darmowego środowiska Visual C# 2005 Express Edition. Rozpoczynamy od stworzenia nowego projektu *Windows Forms*. Do projektu dodajemy nową klasę i nadajemy jej nazwę `IPHeader`. Klasa ta będzie reprezentować nagłówek IP. Najpierw dodamy odpowiednie pola do naszej klasy (Listing 4).

Pola klasy, oprócz dwóch ostatnich, są polami nagłówka IP. Pole `headerLength` to całkowita długość nagłówka IP. Tablica `ipData` jest ładunkiem datagramu IP.

Możemy teraz przejść do napisania konstruktora klasy. Będzie on pobierał dwa parametry: tablicę przechwyconych bajtów oraz ich ilość. Dodajemy kod z Listingu 5.

Na początku tworzymy strumień poprzez stworzenie nowego obiektu klasy `MemoryStream` z przestrzeni nazw `System.Io`. Przeciążony konstruktor tej klasy wymaga następujących parametrów: tablicy bajtów, z której chcemy utworzyć strumień, pozycji w tablicy, od której zacznie się tworzenie strumienia, a także długości strumienia. Następnie tworzymy obiekt klasy `BinaryReader`, dzięki któremu możliwe jest odczytanie typów prostych ze strumienia. Konstruktor przyjmuje referencję do strumienia, z którego chcemy czytać. Możemy teraz odczytać interesujące nas dane. Warto tu zwrócić uwagę na dwie rzeczy. Metody odczytujące z klasy `BinaryReader` automatycznie zmieniają pozycję w strumieniu po wykonaniu operacji odczytu, a dane odebrane z sieci zapisane są w kolejności *big endian* (najbardziej znaczący bajt jest umieszczany jako pierwszy), podczas gdy na komputerach PC dane są przeważnie zapisywane w kolejności *little endian* (najmniej znaczący bajt umieszczany jest jako pierwszy). W związku z tym, aby zmienić kolejność bajtów, wykorzystujemy statyczną metodę `NetworkToHostOrder` klasy `IPAddress`. Aby móc jej użyć, musimy dodać przestrzeń nazw `System.Net`. Następnie odczytujemy długość odebranego nagłówka IP. Jest ona zapisana jako ilość 4-bajtowych słów

i znajduje się w młodszej części bajtu `versionAndHeaderLength`. Aby nie utracić danych z tej zmiennej, wykorzystujemy pomocniczą zmienną `headerLength`. Wykonujemy operację AND, aby wyzerować starszą część bajtu oraz mnożymy uzyskaną wielkość przez 4 w celu uzyskania właściwej długości nagłówka. Wartość tę wykorzystujemy do określenia ilości bajtów zajmowanych przez ładunek w datagramie IP. Używając statycznej metody `Copy` klasy `Array`, kopiujemy te dane do wcześniej zadeklarowanej tablicy, którą wykorzystamy do odczytu danych segmentu TCP.

Ostatnim krokiem jest zdefiniowanie właściwości. Nie będziemy jednak robić tego dla każdego pola naszej klasy, lecz wyłącznie dla tych wykorzystywanych przy analizie segmentu TCP. Dodajemy kod z Listingu 6.

Dodatkowo zdefiniowaliśmy typ wyliczeniowy z Listingu 7., który należy dodać przed definicją klasy. Służy on do określenia typu protokołu, którego pakiet jest ładunkiem w datagramie IP. Dla nas interesujący jest tylko protokół TCP, tak więc typ wyliczeniowy zawiera tylko dwa pola: `TCP` (dla protokołu TCP) oraz `OTHER` (dla innego typu protokołu).

Przystępujemy teraz do napisania klasy reprezentującej nagłówek TCP. Dodajemy nowa klasę i nadajemy jej nazwę `TCPHeader`. Zaczniemy od dodania pól do naszej klasy – dopisujemy kod z Listingu 8.

Pola klasy, z wyjątkiem dwóch ostatnich, to pola nagłówka TCP, `headerLength` jest długością nagłówka w bajtach, a tablica `tcpData` to dane segmentu.

Przechodzimy teraz do zdefiniowania konstruktora dla naszej klasy. Podbiera on dwa parametry: tablicę bajtów, będącą ładunkiem datagramu IP, oraz ich ilość. Dodajemy kod z Listingu 9.

Na początku tworzymy obiekty `MemoryStream` oraz `BinaryReader`, wykorzystywane do odczytu poszczególnych pól nagłówka TCP. Następnie odczytujemy długość nagłówka. Jest ona zapisana – ponownie jako ilość 4-bajtowych słów – w 4 najstarszych bitach zmiennej `flagsAndOffset`. Wykonujemy więc przesunięcie o 12 pozycji w lewo oraz mnożymy otrzymaną wartość przez 4 w celu uzyskania właściwej ilości bajtów zajmowanych przez nagłówek TCP. Długość tę wykorzystujemy do określenia miejsca początku danych segmentu.

Na końcu dodajemy właściwości dla wybranych pól naszej klasy, dopisując kod z Listingu 10.

Po utworzeniu klas opisujących nagłówki IP oraz TCP, przystępujemy do zdefiniowania klasy przechwytującej dane z sieci. Tworzymy nową klasę i nadajemy jej nazwę `GGListener`. Pierwszym krokiem powinno być dodanie niezbędnych przestrzeni nazw: `System.Net`, `System.Net.Sockets`, `System.IO`, `System.Text`. Do klasy dodajemy pola z Listingu 11.

Pola `ipHeader` oraz `tcpHeader` są odpowiednio referencjami do obiektów `IPHeader` oraz `TCPHeader`. Pole `s` jest obiektem klasy `Socket`, reprezentującej gniazdo. Pole `filePath` jest ścieżką do pliku, w którym będziemy zapisywali przechwycone wiadomości Gadu-Gadu.

Możemy teraz przejść do napisania metod naszej klasy. Przedstawia je kod z Listingu 12.

Pierwsza metoda – konstruktor – jako parametr przyjmuje ścieżkę do pliku, w którym zapiszemy przechwycone wiadomości. Metoda `IsGGPort` sprawdza, czy numer portu, określony przekazywanym do niej parametrem, jest numerem portu Gadu-Gadu. Za pomocą metody `MachineAddress` uzyskujemy referencję do obiektu klasy `IPAddress` reprezentującej adres IP. Sercem klasy jest metoda `StartToListen`, za pomocą której rozpoczynamy przechwytywanie danych. Na początku tworzymy nowy obiekt klasy `Socket`. Konstruktor klasy przyjmuje następujące parametry:

- typ wyliczeniowy `AddressFamily` reprezentuje rodzinę protokołów do obsługi gniazda – każda stała, przechowywana przez ten typ wyliczeniowy, określa sposób, w jaki klasa `Socket` będzie określała adres. `InterNetwork` określa użycie adresu IP w wersji 4,
- typ wyliczeniowy `SocketType`, określający typ gniazda,
- typ wyliczeniowy `ProtocolType`, określający typ protokołu.

Listing 7. *Typ wyliczeniowy EProtocolType*

```
enum EProtocolType
{
   TCP,
   OTHER
}
```

Listing 8. *Pola klasy TCPHeader*

```
private ushort sourcePort;
private ushort destinationPort;
private uint sequenceNumber;
private uint acknowledgmentNumber;
private ushort dataOffsetAndFlags;
private ushort window;
private short checkSum;
private ushort urgentPointer;
private byte headerLength;
private byte[] tcpData = new
                      byte[4096];
```

Listing 9. *Konstruktor klasy TCPHeader*

```
public TCPHeader(byte[] data, int received)
{
   try
   {
      MemoryStream sm = new MemoryStream(data, 0, received);
      BinaryReader br = new BinaryReader(sm);
      sourcePort = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      destinationPort = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      sequenceNumber = (uint)IPAddress.NetworkToHostOrder(br.ReadInt32());
      acknowledgmentNumber = (uint)IPAddress.NetworkToHostOrder(br.ReadInt32());
      dataOffsetAndFlags = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      window = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      checkSum = (short)(IPAddress.NetworkToHostOrder(br.ReadInt16()));
      urgentPointer = (ushort)IPAddress.NetworkToHostOrder(br.ReadInt16());
      headerLength = (byte)(dataOffsetAndFlags >> 12);
      headerLength *= 4;
      Array.Copy(data, headerLength, tcpData, 0, received - headerLength);
   }
   catch (Exception exc){}
}
```

Po stworzeniu gniazda musimy je powiązać z pulą adresów uwzględnianych przy nasłuchu. Służy do tego metoda `Bind`. Jako parametr przyjmuje ona referencję do obiektu klasy `EndPoint`. Jednakże klasa ta jest abstrakcyjna, tak więc musimy skorzystać z jednej z klas pochodnych. Tworzymy więc nowy obiekt klasy `IPEndPoint`. Konstruktor tej klasy wymaga dwóch parametrów: referencji do obiektu klasy `IPAddress` uzyskiwanej przez wywołanie wcześniej zdefiniowanej metody `MachineAddress` oraz numeru portu (dla *surowych* gniazd ustawiamy wartość 0, gdyż nie korzystają one z portów). Następnym krokiem jest określenie trybu operacji niskopoziomowych wykonywanych przez gniazdo poprzez wywołanie metody `IOControl`. Metoda ta przyjmuje 3 parametry:

- typ wyliczeniowy `IOControlCode`, określający kod kontrolny operacji do wykonania. Stała `ReceiveAll` oznacza, że będą odbierane wszystkie pakiety IPv4,
- tablicę bajtów z parametrami wejściowymi. Zgodnie z dokumentacją *Platform SDK* ten parametr powinien być typu `BOOL` (4 bajty) i mieć wartość `TRUE` – tak więc przekazujemy tablicę { 1, 0, 0, 0 },
- tablicę bajtów z danymi wyjściowymi.

Metoda ta jest analogiczna do funkcji *WinApi* `WSAIoctl`.

Następnie rozpoczynamy asynchroniczne odbieranie danych poprzez wywołanie metody `BeginReceive`, przyjmującej następujące parametry:

- tablicę bajtów, w której zostaną umieszczone odebrane dane,
- indeks tablicy, od którego rozpocznie się zapisywanie danych w tablicy,
- ilość bajtów, jaką można maksymalnie odebrać,
- flagi,
- delegację `AsyncCallback` określającą metodę, wywoływaną w momencie ukończenia asynchronicznej operacji,
- obiekt klasy `Object`, dzięki któremu możemy przekazać dane do metody wywoływanej po zakończeniu asynchronicznej operacji.

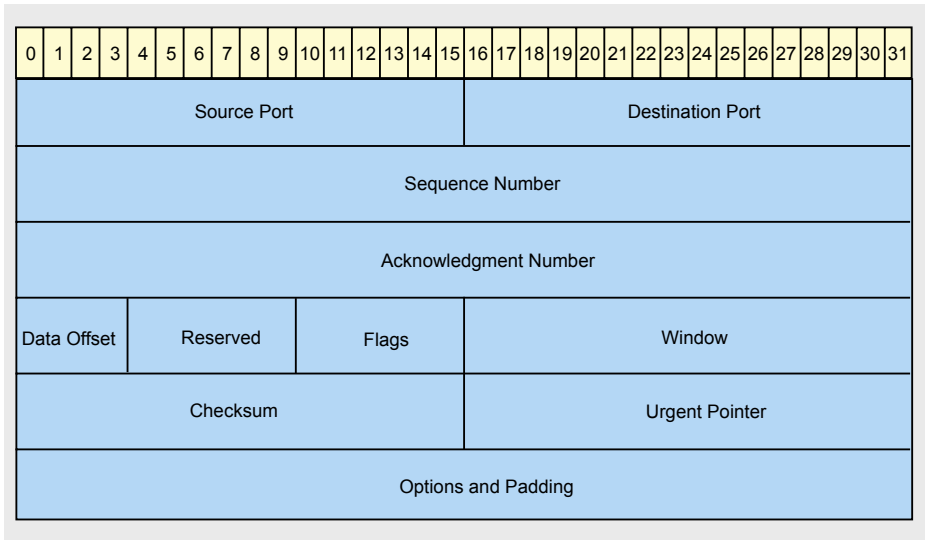
W naszym projekcie delegacja `AsyncCallback` jest implementowana przez metodę `AsyncDataReceived`. Tworzy ona nowy obiekt klasy `IPHeader` i sprawdza, czy ładunek w datagramie IP zawiera segment TCP. Jeżeli tak się dzieje, wywoływana jest metoda `SaveInfo`. Na końcu ponownie rozpoczynamy asynchroniczne odbieranie danych.

Metoda `SaveInfo` rozpoczyna działanie od stworzenia nowego obiektu klasy `TCPHeader`. Następnie sprawdza, czy port źródłowy bądź port docelowy są portami Gadu-Gadu. Jeżeli nie, kończy działanie. W przeciwnym wypadku następuje sprawdzenie, czy przechwyciliśmy wychodzącą/przychodzącą wiadomość, czy też inny typ pakietu. W przypadku, gdy pakiet Gadu-Gadu jest wiadomością wysyłaną do/z klienta Gadu-Gadu, interesujące nas dane są odczytywane i następnie zapisywane do pliku.

Gdy nasza klasa jest już gotowa,

W Sieci

- *http://ekg.chmurka.net/docs/protocol.html* – opis protokołu Gadu-Gadu,
- *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, właściwości i metod, jakie zawiera platforma .NET wraz z przykładowymi programami.



Rysunek 2. *Nagłówek TCP*

Listing 10. *Właściwości klasy TCPHeader*

```
public byte[] TcpData
{
   get
   {
      return tcpData;
   }
}
public ushort SourcePort
{
   get
   {
      return sourcePort;
   }
}
public ushort DestinationPort
{
   get
   {
      return destinationPort;
   }
}
```

Listing 11. *Pola klasy GGListener*

```
private IPHeader ipHeader;
private TCPHeader tcpHeader;
private byte[] data;
private Socket s;
   private string filePath;
```

Listing 12. *Metody klasy GGListener*

```
// konstruktor
public GGListener(string _fileParh)
{
   filePath = _fileParh;
}
private bool IsGGPort(ushort port)
{
   return (port == 8074 || port == 443);
}
public void StartToListen()
{
   try
   {
      s = new Socket(AddressFamily.InterNetwork,
                     SocketType.Raw, ProtocolType.IP);
      s.Bind(new IPEndPoint(MachineAddress(), 0));
      byte[] optionInValue = new byte[4] { 1, 0, 0, 0 };
      byte[] optionOutValue = new byte[4];
      s.IOControl(IOControlCode.ReceiveAll, optionInValue,
                     optionOutValue);
      data = new byte[4096];
      s.BeginReceive(data, 0, data.Length, SocketFlags.None,
                     new AsyncCallback(AsyncDataReceived),
                     null);
   }
   catch (Exception exc){}
}
private IPAddress MachineAddress()
{
   string hostName = Dns.GetHostName();
   IPHostEntry ipHostEntry = Dns.GetHostByName(hostName);
   return ipHostEntry.AddressList[0];
}
private void AsyncDataReceived(IAsyncResult result)
{
   try
   {
      int nReceived = s.EndReceive(result);
      ipHeader = new IPHeader(data, nReceived);
      if (ipHeader.TypeOfProtocol == EProtocolType.TCP)
      {
         SaveInfo();
      }
      data = new byte[4096];
      s.BeginReceive(data, 0, data.Length, SocketFlags.None,
                     new AsyncCallback(AsyncDataReceived),
                     null);
   }
   catch(Exception exc){}
}
private void SaveInfo() {
try
{
   tcpHeader = new TCPHeader(ipHeader.Data,
                  ipHeader.MessageLength);
   if (!(IsGGPort(tcpHeader.DestinationPort) || IsGGPort(tc
                  pHeader.SourcePort)))
      return;
   if (BitConverter.ToUInt32(tcpHeader.TcpData, 0) == 0x0b
                  || BitConverter.ToUInt32(tcpHeader.TcpDa
                  ta, 0) == 0x0a)
   {
      int nMsgLength = BitConverter.ToInt32(tcpHeader.TcpD
                  ata, 4);
      int nStartingByte = 0;
      if (!File.Exists(filePath))
         using(File.Create(filePath));
      using (StreamWriter sw = File.AppendText(filePath))
      {
         string msgType = " ";
         sw.Write("\r\n++++++++++++++++++++++++++++++++++
                  +++++\r\n");
         if (tcpHeader.DestinationPort == 8074 || tcpHeader
                  .DestinationPort == 443 )
         {
            sw.Write("Wiadomość wychodząca\r\n");
            msgType = "Numer odbiorcy ";
            nStartingByte = 20;
            nMsgLength -= 12;
         }
         if (tcpHeader.SourcePort == 8074 ||
                  tcpHeader.SourcePort == 443)
         {
            sw.Write("Wiadomość przychodząca\r\n");
            msgType = "Numer nadawcy ";
            nStartingByte = 24;
            nMsgLength -= 16;
         }
         sw.Write("Port źródłowy " + tcpHeader.SourcePort +
                  "\r\n");
         sw.Write("Port docelowy " + tcpHeader.DestinationP
                  ort + "\r\n");
         sw.Write(msgType + BitConverter.ToUInt32(tcpHeader
                  .TcpData, 8) + "\r\n");
         Encoding e = Encoding.GetEncoding("windows-1250");
         sw.Write("Wiadomość " + e.GetString(tcpHeader.TcpD
                  ata, nStartingByte, nMsgLength));
      }
   }
}
catch (Exception exc){}
}
```

możemy przejść do widoku formy i utworzyć nowy obiekt klasy oraz rozpocząć przechwytywanie danych wywołując metodę `StartToListen`.

Podsumowanie

Podstawowym celem artykułu było pokazanie, jak potężnym narzędziem są *surowe* gniazda i w jak łatwy sposób pozwalają na przechwytywanie danych przesyłanych w sieci. W sposób podobny do prezentowanego w artykule możemy zaimplementować aplikacje przechwytujące e-maile, a także monitorujące adresy odwiedzanych przez użytkownika stron internetowych. Hakerzy wykorzystują je do przeprowadzenia ataków typu *Denial of Service* (odmowa usługi) lub *IP address spoofing* (fałszowanie adresu IP). Jednakże nie wolno zapomnieć o pozytywnych aspektach użycia *surowych* gniazd. Niewątpliwie przydają się wszędzie tam, gdzie wymagany jest pełny wgląd do danych przesyłanych w sieci, np. w celu wysłania/odbioru niestandardowych pakietów.

Maciej Pakulski
Absolwent studiów inżynierskich oraz aktywny członek koła naukowego .NET Wydziału Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Mikołaja Kopernika w Toruniu. Obecnie na studiach magisterskich. Programowaniem zajmuje się od 2004. Potrafi programować biegle w językach C/C++, Java, VHDL. Programowaniem w języku C# i platformą .NET zajmuje się od 2006 roku. Jest autorem szeregu publikacji z zakresu programowania oraz bezpieczeństwa IT.
Kontakt z autorem: *mac_pak@interia.pl*