Safer Sockets

by Primoz Gabrijelcic

Let's face it: Delphi's VCL has bugs. It is just too big and complex to be completely cleaned up. While some bugs are just irritating, others can be very instructive. I'll show you one such bug and then discuss some ways of working around it (fixing it, as you'll see, won't always be possible).

Blocking Sockets

Today's culprit is TWinSocketStream, a streaming class introduced in Delphi 3. It hardly changed in Delphi 4: some integers were replaced with DWORDs. If you have the VCL source code, you'll find it in the ScktComp unit, which is in SOURCE\INTERNET for Delphi 3 or SOURCE\VCL for Delphi 4. If you don't have the source don't worry: you should still find some interesting facts and ideas in this article.

It all starts with sockets. Sockets are a standardised way of accessing TCP/IP communication layers from a program. In other words, if you develop internet applications, you need sockets. A socket represents two-way communication: you write something to the socket and then you get a reply.

There are two ways of accessing the socket, non-blocking and blocking. In the former scenario, a socket generates events when it needs data or it has data ready. So you must build a state machine to implement a simple data exchange algorithm. If that seems too much trouble, you can always access a socket in the blocking mode. You just Write to the socket and the program stops until all data is written. Similarly with Read. Just like using a file.

Are there drawbacks? Well, I've already said it: when you access a socket in blocking mode, the program *blocks* until the operation is finished. Also, sockets have no built-in timeout mechanism, so you must either watch the communication with another thread or use a special streaming class for reading and writing... you guessed it, TWinSocketStream!

Enter TWinSocketStream

TWinSocketStream is derived from TStream. It overrides the Read, Write and Seek functions, and adds WaitForData (Listing 1). Also changed is the constructor, which takes two arguments: the socket that the instance of TWinSocketStream is bound to and the timeout value (in milliseconds). This is used in the Read and Write functions: they timeout after the specified time and return 0 (that is, 0 bytes were received or transmitted).

WaitForData returns when data is ready to be read or when the specified timeout is exceeded. Seek just returns 0, as seeking cannot be implemented on sockets.

How It Should Work

The Read function takes just two parameters, Buffer and Count. A maximum of Count bytes will be read into Buffer and then Read will return. Read will return after some time (the timeout value is defined in TWinSocketStream.Create) even if not enough data was read. The function result is the number of bytes read, or 0 if it times out.

The implementation of TWinSocketStream.Read is very neat and effective (see Listing 2). First, the Overlapped memory structure is prepared. Then ReadFile is called. We already know the Buffer and Count parameters. Result returns the number of bytes read and Overlapped specifies that the read operation is asynchronous (note that asynchronous operations with sockets are supported even in Windows 95). The program will continue execution and the event FEvent (specified in Overlapped.hEvent) will be fired after all the data is read. An exception is raised if ReadFile fails.

Read then waits at least FTimeOut milliseconds (that is, the timeout value specified in the constructor) for FEvent to fire. If a timeout occurs, 0 is returned, otherwise GetOverlappedResult is called to

```
TWinSocketStream = class(TStream)
private
  FSocket: TCustomWinSocket;
  FTimeout: Longint;
  FEvent: TSimpleEvent;
public
  constructor Create(ASocket: TCustomWinSocket; TimeOut: Longint);
  destructor Destroy; override;
  function WaitForData(Timeout: Longint): Boolean;
  function Read(var Buffer; Count: Longint): Longint; override;
  function Write(const Buffer; Count: Longint): Longint; override;
  function Seek(Offset: Longint; Origin: Word): Longint; override;
  property TimeOut: Longint read FTimeout write FTimeout;
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
function TWinSocketStream.Read(var Buffer; Count: Longint): Longint;
var
  Overlapped: TOverlapped;
  ErrorCode: Integer;
begin
  FSocket.Lock;
  try
    FillChar(OVerlapped, SizeOf(Overlapped), 0);
    Overlapped.hEvent := FEvent.Handle;
    if not ReadFile(FSocket.SocketHandle, Buffer, Count, DWORD(Result),
      @Overlapped) and (GetLastError <> ERROR_IO_PENDING) then begin
      ErrorCode := GetLastError;
      raise ESocketError.CreateFmt(sSocketIOError, [sSocketRead, ErrorCode,
        SysErrorMessage(ErrorCode)]);
    end;
    if FEvent.WaitFor(FTimeOut) <> wrSignaled then
      Result := 0
    else begin
      GetOverlappedResult(FSocket.SocketHandle, Overlapped, DWORD(Result),
        False);
      FEvent.ResetEvent;
    end;
  finally
    FSocket.Unlock;
  end;
end;
```

retrieve the result of the ReadFile operation and that result is returned. FEvent is then reset.

Oh, and everything is of course wrapped into a critical section (FSocket.Lock to FSocket.Unlock).

Why It Doesn't

So where is the problem? It usually occurs when communication is extremely slow (eg slow modem lines with lots of line noise). In that case, FEvent.WaitFor may timeout and TWinSocketStream.Read returns 0, but that does not cancel out a pending asynchronous ReadFile request. Some time later all the data may arrive. When that happens, ReadFile tries to put the received data into Buffer. But Buffer may not be there any more!

Examine the (intentionally oversimplified) example in Listing 3. Now imagine how this program may execute: Buffer is allocated, Sockstream.Read is called, Read timeouts and sockstream.Read returns with result 0, but ReadFile is still waiting for data. Buffer is freed and then reallocated, very likely from the same memory area as before. During a long computation ReadFile manages to collect the data and stores it into Buffer. But hey, wait, that is not the same Buffer anymore! Data already there will be destroyed.

So, using TWinSocketStream.Read can cause memory overwrite, data corruption, program malfunction Access Violations and more.

TWinSocketStream.Write is implemented in a similar manner and can cause the same problems. Furthermore, Write can send random data to the socket and as such presents a security threat.

And How We Can Fix It

There are two workarounds, each with a different problem.

The first option is to add one line to TWinSocketStream.Read. Just before returning 0 (Result := 0), the ReadFile operation can be cancelled by CancelIO. Instead of:

```
if FEvent.WaitFor(FTimeOut) <>
  wrSignaled then
  Result := 0
else
```

```
program BadRead(sockstream: TWinSocketStream);
var
  buffer: pointer;
  count: longint;
begin
  GetMem(buffer,1024);
  sockstream.Read(buffer,1024);
  FreeMem(buffer);
  GetMem(buffer,1024);
  { ... some long computation involving buffer }
  FreeMem(buffer);
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
function TSafeWinSocketStream.Read(var Buffer; Count: longint): longint;
var numb,lread: integer;
begin
  if swsFailed then
    Result := 0
  else begin
    try
      numb := 0;
      lread := 1;
      while (lread > 0) and (numb < count) do begin
        lread := count-numb;
        if lread > CSmallBlockSize then
          lread := CSmallBlockSize;
        if not inherited WaitForData(TimeOut) then
          lread := 0
        else
          lread := inherited Read(swsBuffer^,lread);
        if lread > 0 then begin
          Move(swsBuffer^,pointer(integer(@buffer)+numb)^,lread);
          numb := numb + lread;
        end;
      end;
      Result := numb;
    except
      Result := 0;
    end;
    swsFailed := (Result <> count);
  end;
end; { TSafeWinSocketStream.Read }
```

we now have:

```
if FEvent.WaitFor(FTimeOut) <>
  wrSignaled then begin
  CancelIO(FSocket.SocketHandle);
  Result := 0
end else
```

Instead of fixing the ScktComp unit it is probably better to create a derived class, override the Read method, paste code from ScktComp and make this small modification. Choose your way but beware: CancelIO is not available in Windows 95, only in 98 and NT 4.

A more complicated solution is to defer freeing Buffer until Socket is destroyed (destroying a socket cancels all pending ReadFile operations). However, this solution is hard to implement. So, I have created TSafeWinSocketStream, a safe wrapper around TWinSocketStream that executes all read/write operations with a buffer that will be freed only after the socket is destroyed. It has its own drawbacks too: after a failed operation, recovery is not possible and all subsequent calls to Read or Write fail without trying to read or write.

TSafeWinSocketStream.Read (Listing 4) reads data in small blocks of size CSmallBlockSize (defined in the SafeWS unit) and moves successfully read data into the main buffer. If a read operation fails, an internal flag is set to indicate failure and the number of successfully read bytes is returned. The internal buffer is freed in TSafeWinSocketStream.Destroy, after the socket itself is destroyed. TSafeWinSocketStream.Write is fixed in the same manner.

Using this derived class is simple: replace all occurrences of TWinSocketStream with TSafeWinSocketStream. Your program will work as before, just better. No more fear of data overwrites or Access Violations. At least not the ones caused by TWinSocketStream.

Primoz Gabrijelcic (actually, Primo\v{z} Gabrijel\v{c}i\v{c} for all \TeX nicians out there) is R&D Manager at FA Bernhardt GmbH. He would like to be able to do on PCs what he could do on VAXes over 10 years ago. Meanwhile, email him at gp@fab-online.com