How to dynamically add data to an executable file (part 2 of 5)

Payload footer record

Payload footer record

In the overview we noted that the payload footer has three purposes:

1. To identify that a payload is present.
2. To record the size of the payload data.
3. To record the size of the original executable file.

Listing 1 defines a Pascal record that records all the required information.

```
type
  TPayloadFooter = packed record
    WaterMark: TGUID;
    ExeSize: LongInt;
    DataSize: LongInt;
  end;
```

Listing 1

The purpose of the fields is as follows:

- The *Watermark* field is a "magic number" that is used to identify the fact that a payload is present. A GUID is used to try to ensure that the watermark can't be present by accident. This field is always set to the same fixed value. In a while we will see how this field is used to detect a payload.
- The *ExeSize* field stores the size of the original executable file before the payload was appended. The field also gives the offset of the start of the payload data in the file, since the payload immediately follows the executable code.
- The *DataSize* field stores the size of the payload itself. This is in fact redundant information – it's value can be deduced from the value of the *ExeSize* field, the size of the file and the size of the *TPayloadFooter* record. However by providing this field we can simplify subsequent code.

We will often need to create a new, blank, footer record that contains the correct watermark. *Listing 2* illustrates a simple helper procedure that initializes such a blank footer.

```
const
  cWaterMarkGUID: TGUID =
    // arbitrary watermark constant: must not be all-zeroes
    '{9FABA105-EDA8-45C3-89F4-369315A947EB}';

procedure InitFooter(out Footer: TPayloadFooter);
begin
  FillChar(Footer, SizeOf(Footer), 0);
  Footer.WaterMark := cWaterMarkGUID;
end;
```

Listing 2

The routine simply zeroes the footer record and stores the required watermark in it.

Let us now consider how to check for the presence of a payload in a file. This is done by reading the final `SizeOf(TPayloadFooter)` bytes from the file into a *TPayloadFooter* record and checking if the *Watermark* field contains the expected magic number. If this is the case it is safe to assume we have a payload and that the record provides valid information about the size of the payload and executable file.

Listing 3 shows the code of a helper routine that both checks for presence of a payload on an open file and gets the footer if so. The routine operates on a standard Pascal un-typed file.

```
function ReadFooter(var F: File;
  out Footer: TPayloadFooter): Boolean;
var
  FileLen: Integer;
begin
  // Check that file is large enough for a footer
  FileLen := FileSize(F);
  if FileLen > SizeOf(Footer) then
  begin
    // Big enough: move to start of footer and read it
    Seek(F, FileLen - SizeOf(Footer));
    BlockRead(F, Footer, SizeOf(Footer));
  end
  else
    // File not large enough for footer: zero it
    // .. this ensures watermark is invalid
    FillChar(Footer, SizeOf(Footer), 0);
  // Return if watermark is valid
  Result := IsEqualGUID(Footer.WaterMark, cWaterMarkGUID);
end;
```

Listing 3

ReadFooter first gets the size of the file and checks if it is large enough to contain a payload footer. If the file is large enough it moves the file pointer to `SizeOf(TPayloadFooter)` bytes back from the end of the file and reads in the the footer record. If the file is too small the routine fills a footer record with zeros, making it invalid (i.e. the watermark is zero). Finally the routine checks if the record's *Watermark* field contains the required *GUID* and returns the result. The footer record is passed out as a parameter.

Watermark warning

Ensure watermarks are non-zero since the code uses zero values to indicate that no footer is present.

We now have enough information to enable us to move on to develop a class that helps us to manage payloads, which is what we do in the *next part*.