Beating the System: Putting The Squeeze On

by Dave Jewell

In last month's column I spent some time looking at the various ways in which it's possible to obfuscate the IDE-detection code in your shareware components, thus making it much more difficult for a would-be hacker to convert the trial version of a control into something that's usable without restriction. This time round, I'm going to continue the same theme, but here the emphasis will be on how to protect your finished applications, rather than protecting individual components.

Letting It All Hang Out, Revisited

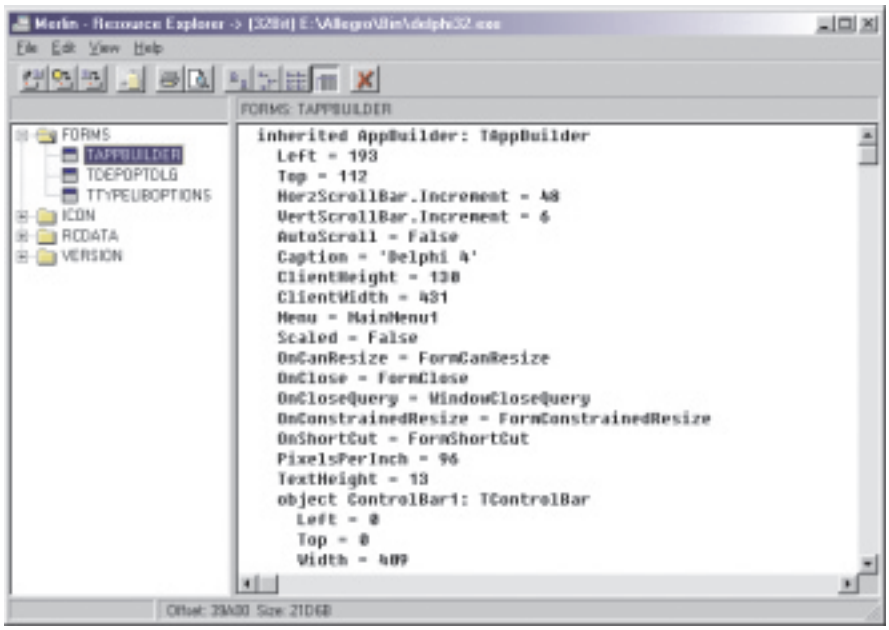
You might well ask why it's necessary to protect a deployed EXE file? After all, once a component has been linked into an executable, it's no longer easily extractable from the surrounding program code. That's certainly true, but we're not simply talking about components here, we're really talking about the wider issues of intellectual property and trade secrets. As I mentioned last month, I've recently seen a Delphi component which goes to great lengths to hide the way in which it implements direct disk access, presumably to prevent other shareware authors from bringing out competing products!

Again, let's suppose that you've developed a new Pascal compiler that provides an extraordinary degree of optimisation in the generated code. Or maybe you've come up with some clever new algorithms for compression, fast manipulation of graphic images, or whatever. Or perhaps you simply prefer to distribute your shareware application with certain restrictions built-in, and you don't want prospective hackers to easily be able to remove those limitations. All of these are good reasons for protecting the innards of an executable against casual viewing, and that's true whether it be an EXE file, package, ActiveX control, or whatever.

In case you're thinking that it really doesn't matter if one or two dedicated individuals succeed in patching the trial version of your application, then think again. Hackers tend to like showing off what they've done to like-minded individuals, and the various 'cracker' websites are full of patches and passwords which can be used to remove protection from hundreds of shareware programs. These sites are known not just to hackers, but also to many others on the lookout for a free lunch. Once your program is listed on these sites, human nature dictates that many people who would otherwise have registered your software won't bother to do so. Over the years, I've talked to dozens of shareware authors about their experiences, and the overwhelming consensus is that making a program difficult to crack can significantly improve your shareware revenues. Such is human nature.

Right then; having hopefully convinced you that protecting your software is potentially a good idea, let me illustrate just how easy it is to peek around inside a Delphi application (much the same techniques can be applied to C++Builder programs as well). Please understand that I'm showing you these techniques specifically so that you'll realise how vulnerable an unprotected executable can be, and for no other reason.

➤ *Figure 1: Merlin's Resource Viewer makes it easy to examine the embedded form resources contained within a compiled executable. Here, we're looking at the main application form within the Delphi 4 IDE itself.*



Merlin: Gone But Not Forgotten...

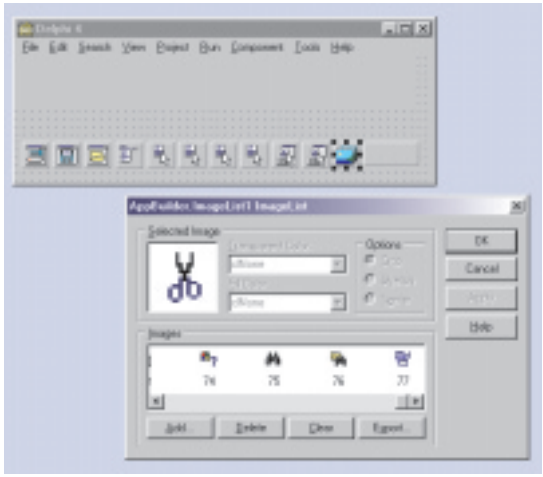
Back in 1997, two well-known Delphi programmers, Mike Scott and John Howe, got together and developed Merlin: an innovative set of COM-based Delphi add-ins.

Sadly, since that time, there does not seem to have been any further development work put into Merlin, one of the reasons being competition from GExperts and CodeRush. Be that as it may, you can still download a shareware version from the official Merlin website at www.boots.com/merlin/.

Although Merlin is getting somewhat long in the tooth now, I still use it a lot, mainly because it's possible to configure Merlin such that the various goodies like its Resource Viewer and Executable Viewer are available on Windows Explorer's context menu. With Merlin installed, I can just right click an EXE file in the shell, and select Resource Viewer from the resulting context menu. You can see a copy of the Resource Viewer running in Figure 1. In this particular case, the Resource Viewer is peeking inside the DELPHI32.EXE program that ships with Delphi 4, the main IDE program. You can see that it contains three `TForm` resources, the first of which, `TAppBuilder`, corresponds to the main IDE window which we all know and love. This particular form resource is being displayed in human readable form in the right-hand pane.

What can we do with this? Well, for starters, you could use the `Save Resource...` option on Merlin's `File` menu. Using this facility, the currently selected resource can be saved as a .RES file to disk. In the case of a Delphi form file, Merlin could save the illustrated `TAppBuilder` resource as a .DFM file. Bear in mind that, at the Windows API level, a .DFM file is simply a .RES file containing a single named `RT_RCDATA` resource where the resource name corresponds directly with the type of the form; `TAppBuilder` in this case.

But there are more tricks up Merlin's copious sleeve. Try this: select all the text in the right-hand pane of the Resource Viewer and then copy it to the Windows clipboard. Next, fire up a copy of Delphi 4 and start a new, empty project. Rename the form so that it has the same name as what you can see in Merlin. In the case of Figure 1, you'd want to change the name of the form to `AppBuilder`). This will cause Delphi to automatically change the form type to `TAppBuilder`. Next, right click on the form window and choose `View as Text` from the context menu. Are you getting the idea? Now delete everything in the resulting code editor view and do a paste to copy the incoming DFM resource data from the clipboard. Finally, choose `View as Form` from the code editor's context menu to return to a conventional form-based view.



➤ *Figure 2: With a little bit of Merlin Magic, (and the Windows clipboard), it's possible to transfer an embedded form resource back into the Delphi IDE for viewing in the usual way. Again, this is the form that corresponds to the Delphi IDE's main form, viewed from inside the IDE..., if you see what I mean.*

Hey presto, with a wave of his magic wand, Merlin has allowed us to grab an embedded form resource from the IDE's own executable and copy it into the development environment where we can see what makes it tick. If you now try double clicking on any of the various `TMainMenu` and `TPopupMenu` components on the form, you'll find that it's possible to view their menu data structures in the usual way. Moreover, you can open the `TImageList` control and browse through the list of glyphs used by the IDE (see Figure 2).

Some months back, I found occasion to use this very technique during a somewhat... umm... *spirited* debate with a Borland (sorry, Inprise) representative who was endeavouring to persuade the denizens of the CIX Borland conference that fixing the dreaded Delphi 4 STB imagelist corrupted icon problem was not the company's responsibility. The false claim was made that putting over 254 bitmaps into an imagelist was the root cause of the problem and we were confidently told that the Delphi 4 IDE went over this limit. By peeking at the various forms in Delphi 4 using the technique described above, I was able to definitely establish that this wasn't the case. In fact, you can clearly see from Figure 2 that there are only 77 bitmaps stored in `ImageList1`. Needless to say, at the time of writing, Borland have just released Update 3 for Delphi 4, which it's claimed now fixes the problem...

Incidentally, if you have Merlin, and you followed the steps I gave earlier, you'll have noticed that the IDE complained about certain component types not being found. Just press the `Ignore All` button when this dialog appears. The reason it happens is because the IDE uses a number of proprietary components (for example `TDockPanel` and `TDockToolBar`) which aren't available to us ordinary mortals. At least, not officially...☺. For similar reasons, you won't be able to save the 'kidnapped' form without lots of complaints from the IDE. That's because all we've done is retrieve the resource data, but the corresponding component declarations are missing from the `TAppBuilder` form declaration, and the various event handlers referenced in the resource data are also absent.

Is there a way around this? Well, there is if you have a clever little utility called EXE2DPR. If you don't have it, you can download a restricted shareware version from the author's website at www.cdc.net/~dmitri/. As the name suggests, EXE2DPR takes an existing

EXE file (it only works for program files, not for packages) and reconstructs the DPR file. But that's not all, it also scans the EXE file for form resources and generates all the missing DFM files. Finally, it also creates a skeletal version of the .PAS file for each form, including all the overridden event handlers that have been referenced from the .DFM resource data: see Figure 3. It's a very cute little utility.

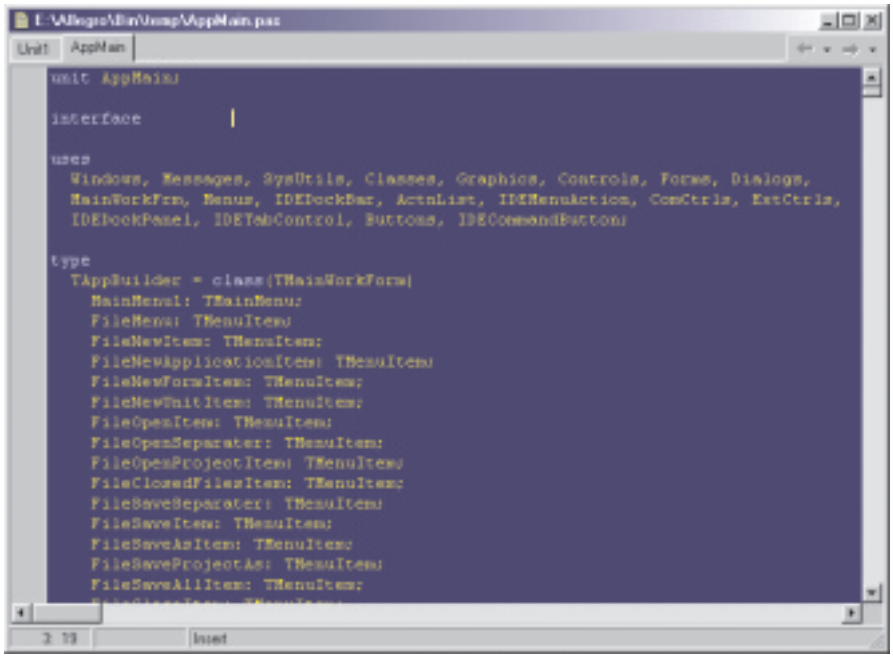
Incidentally, this is probably a good place to emphasise that there are always legitimate reasons for reverse engineering an executable or using tools such as EXE2DPR. If you're in the embarrassing position of having accidentally fed your source code to the dog (or whatever) then such tools can save you a great deal of time in reconstructing, if not all the code, then at least the essential framework of your application.

Fun With UpdateResource

By now, you should hopefully be starting to feel somewhat nervous about the possibility of releasing unprotected Delphi-authored trial software into the field. If so then great, that's exactly my objective. Of course I'm not saying that Delphi applications are easier to reverse engineer than Java code. Most folks know that thanks to utilities such as SourceAgain, it's often possible to recover *virtually all the Java source code* from a Java applet with the greatest of ease (as I've been writing about in my *Java Column* in *Developers Review*, www.itecuk.com). There's no simple way of doing that with a Delphi application, but on the other hand, it's much easier to reverse engineer a Delphi program than one written in C or C++.

If you come from a traditional Windows SDK programming background, you might be forgiven for thinking that a Delphi program is relatively secure. After all, there are no conventional menu resources, dialog template resources or bitmap resources. The Delphi equivalent of all those things is safely hidden away inside the resource date of your form file, where a conventional resource editor (such as Borland's Resource Workshop of fond memory) can't touch it. But as I've tried to show you above, this doesn't mean that it's completely inaccessible.

➤ *Figure 3: Utilities such as EXE2DPR allow you to take things even further. This is part of the source code for the APPMAIN.PAS unit, extracted from the Delphi 4 IDE. Obviously, this utility can't recover the source of individual methods, but it does an excellent job of recovering the form declaration.*



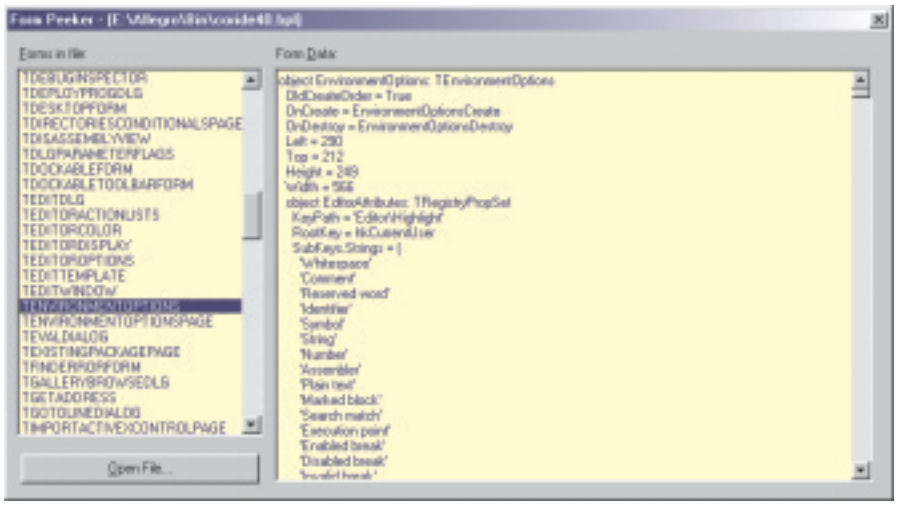
Suppose you've got a trial program which has no functionality restrictions, but simply displays a background bitmap with the words 'Unregistered Version' displayed prominently in red. Typically, you would store that bitmap as part of the form data of a `TImage` component, or possibly in an imagelist. Either way, once you've extracted the DFM data into a form that can be manipulated by the Delphi IDE, the bitmap can easily be changed. This raises the interesting question of how does one then put the modified form resource back into the original executable? Again, this is quite easy thanks to a routine called `UpdateResource` that's only available under Windows NT. Using `UpdateResource`, it's possible to permanently edit the resources contained within a compiled, linked executable. Thus, using techniques similar to those described here, it would actually be quite straightforward to write a Delphi application which 'reaches into' the heart of another executable, streams an existing form resource into memory, makes changes to (for example) `TImage` and `TImageList` items in the usual way, and then streams the result back out again, writing it to the EXE file with `UpdateResource`.

And Even More Fun With ObjectBinaryToText...

Maybe you're wondering how a utility such as Merlin's Resource Viewer works its magic? There's actually very little work involved. Just to prove the point, take a look at the code for my own little Form Peeker shown in Listing 1: you can see the program running in Figure 4 and (as ever) full source is included on this month's disk. The operation of this program is very straightforward: when the user clicks on the `Open File` button, a filename is obtained from the common dialog Open File box in the usual way, and the `GetResourceInfo` routine is then called. Within `GetResourceInfo`, I use the `LoadLibraryEx` routine to load the designated executable into memory. Somewhat coun-

```
unit UMain;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, StdCtrls, ComCtrls;
type
  TPeekForm = class(TForm)
    FormList: TListBox;
    OpenDialog: TOpenDialog;
    Label1: TLabel;
    OpenButton: TButton;
    Label2: TLabel;
    FormData: TRichEdit;
    procedure FormDestroy(Sender: TObject);
    procedure OpenButtonClick(Sender: TObject);
    procedure FormListClick(Sender: TObject);
  private
    hMod: THandle;
    procedure Clear;
    procedure GetResourceInfo;
  public
  end;
var
  PeekForm: TPeekForm;
implementation
{$R *.DFM}
function EnumResProc (hMod: THandle; ResType, ResName:
  PChar; Self: TPeekForm): Boolean; stdcall;
var
  h: THandle;
  p: pDWord;
begin
  // OK - we've got a rc_Data resource, but is it a DFM?
  h := LoadResource(hMod, FindResource(hMod, ResName,
    ResType));
  p := LockResource (h);
  if p^ = $30465054 then
    Self.FormList.Items.Add (ResName);
  Result := True;
end;
procedure TPeekForm.Clear;
begin
  if hMod > 0 then
    FreeLibrary (hMod);
  FormData.Lines.Clear;
  FormList.Clear;
end;
procedure TPeekForm.GetResourceInfo;
var
  hTemp: THandle;
begin
  hTemp := LoadLibraryEx (PChar (OpenDialog.FileName), 0,
    Load_Library_As_DataFile);
  if hTemp <> 0 then begin
    Clear;  hMod := hTemp;
    Caption := Format('Form Peeker - [%s]',
      [OpenDialog.FileName]);
    EnumResourceNames(hMod, rt_rcData, @EnumResProc,
      Integer(Self));
    if FormList.Items.Count > 0 then begin
      FormList.ItemIndex := 0;
      FormListClick (Self);
    end;
  end;
end;
procedure TPeekForm.FormDestroy (Sender: TObject);
begin
  Clear;
end;
procedure TPeekForm.OpenButtonClick(Sender: TObject);
begin
  if OpenDialog.Execute then GetResourceInfo;
end;
procedure TPeekForm.FormListClick(Sender: TObject);
var
  sText: TMemoryStream;
  sRes: TResourceStream;
begin
  with FormList do if ItemIndex <> -1 then begin
    sRes := TResourceStream.Create(hMod, Items [ItemIndex],
      rt_rcData);
    try
      sText := TMemoryStream.Create;
      try
        ObjectBinaryToText (sRes, sText);
        sText.Position := 0;
        FormData.Lines.LoadFromStream (sText);
      finally
        sText.Free;
      end;
    finally
      sRes.Free;
    end;
  end;
end;
end.
```

➤ *Listing 1*



➤ *Figure 4: Here's my little form peeker program, the source code for which is given in Listing 1. As you can see, there's really very little to it. This utility can be used to examine the innards of all 32-bit Delphi and C++ Builder executables, as well as .DPL and .BPL packages.*

ter-intuitively, the `LoadLibraryEx` routine will happily load EXE files as well as DLLs, meaning that you can use it to examine both applications and packages.

If the executable loads successfully, then the left-hand listbox and the right-hand rich-edit control are both cleared and the API-level `EnumResourceNames` routine is called to enumerate all the resources contained within the designated module. More specifically, it enumerates all resources of type `RT_RCDATA`, because of course we're looking specifically for form resources. This causes Windows to repeatedly call our enumeration callback routine, `EnumResProc`, passing the name of each resource found in the `ResName` argument.

I know I've said this before, but let me emphasise *once again* that when working with an API-level enumeration routine, you must be sure to use the `stdcall` specifier when defining the call-back routine. If, on the other hand, you use a VCL enumeration routine (such as `EnumModules`, defined in the `SYSTEM` unit) then you should *not* use the `stdcall` specifier. If you get it wrong, you'll soon know about it!

For each encountered resource, the `EnumResProc` routine loads it into memory and examines the first four bytes of the resource to determine whether or not this is actually a form. In Delphi, every form is a `RT_RCDATA` resource, but every `RT_RCDATA` resource ain't necessarily a form! To see what I mean, use Merlin's Resource Viewer to

take a peek at CORIDE40.BPL, the package which implements the core IDE code. In there, you'll find a heck of a lot of forms, but you'll also see a number of other RT_RCDATA resources which don't correspond to forms, such as the various key-mapping resources. If EnumResProc is happy that it's dealing with a resource, then the name of the resource is added to the FormList listbox.

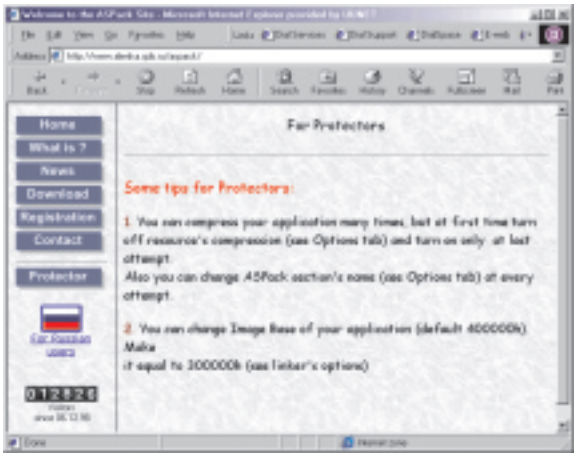
And that's about it, apart from the FormListClick routine where the interesting stuff happens. Inside there, the clicked-on resource data is copied into a stream using the seriously useful TResourceStream class. Once that's done, the even more useful ObectBinaryToText routine is called to convert the binary stream data into a human readable representation of the form. This is written to a memory stream object, and subsequently copied into the Lines property of the rich-text edit control using the familiar LoadFromStream method.

In case you're wondering why I bothered with a rich-text control, suffice it to say that some of those form resources are *big*, especially when they contain a large number of embedded bitmaps. In Figure 4, you can see my little Form Peeker program examining the TEnvironmentOptions form that's used to implement the tabbed environment options dialog in the Delphi and C++Builder IDEs. If you were to replace the rich-text control with, say, a TMemo control, then you'd end up exceeding the capacity of the control when selecting this particular form resource.

This program could clearly be taken much further. You could parse the human-readable text data and build up a list of used components and event handlers. You could even do what EXE2DPR does and recreate a compilable declaration for the form itself!

Here Come The (Compressed) Cavalry!

At this point, your nervousness should have matured into a deep despondency. If it's really this easy to 'massage' Delphi software, then maybe I should switch to C++, right? Wrong! Nothing could be *that* bad! The fact is, it's very easy to protect your EXE files, DLLs and packages against the sort of invasion of privacy that I've been discussing up until now. You can do it by making use of one of the various executable compressor utilities that are on the market at the present time.



➤ *Figure 5: Some EXE compression tool authors have thought seriously about anti-hacking issues. The creator of ASPack advocates the use of multiple levels of compression and also changing the section name used by the decompression code.*

If you're not familiar with EXE compressors, then here's how they work. A compressor applies some data compression algorithm to the code contained within your executable file, and (optionally) to the resource data, and import/export tables contained within the file. You're probably aware of the fact that machine code cannot be compressed to the same extent as plain text, but nevertheless most compressors are able to reduce the size of a file by something like half.

Using a compressor has several benefits. Firstly, it means that the EXE file will typically load more rapidly since disk I/O is, in terms of speed, a much more expensive operation than in-memory decompression. Secondly, the end-user of your program will benefit from reduced disk space requirements and thirdly (most important of all from the viewpoint of this discussion) is the anti-hacking angle. It ought to be obvious that when you're dealing with a compressed executable it becomes difficult, if not impossible, to patch the program code. Not only can the hacker not 'see' the original code, but any patch that's applied must fit right into the compressed bit stream. Since the decompression code in a compressed executable will almost certainly perform some sort of checksum validation on the decompressed data, you can see that the chances of successfully applying a patch 'in situ' are exceedingly slim.

I mentioned earlier that my Form Peeker utility was included on this month's cover disk. What I didn't mention was that I've compressed it. The initial program was only 13Kb long because I chose to build the program using packages. (This is a Delphi 3 application, by the way, so if you don't have the Delphi 3 runtime packages installed, you'll have to rebuild the EXE under your own preferred version of Delphi before running it.) Using a shareware compressor called ASPack, I then shaved a further 3Kb away, resulting in an EXE file that's a mere 9728 bytes long! But here's the cunning part: if you try to use PEEKER.EXE to look inside its own executable, you won't find any form resources at all. Those resources are still there, but they're compressed, and therefore the first four bytes are not recognised as a valid signature by the code inside EnumResProc.

When using various programmers' utilities to peek inside a compressed file, the results will differ according to how the utility was written and what specific compressor was used to crunch the file. Merlin will happily display the form resources within PEEKER.EXE, but it won't recognise them as forms because it can't find the magic four-byte signature, all you'll see is a hex dump of the com-

pressed data. If you compress your executable using a different compressor, Merlin will typically crash altogether, and we're more than happy that it should do so, since we're trying to prevent any form of peeking from going on!

For your information, Table 1 gives a quick round-up of compressors I have known and loved.

A couple of other compressors that spring to mind are NeoLite (www.neoworx.com/neolite/) and WWPack32 (www.webmedia.pl/wwpack32/index.html) although I have no experience with these two particular products.

If security is your primary consideration when compressing your program executables, then I'd have to recommend that you do not use Shrinker or PkLite. I hasten to add that there's nothing wrong with these products, but utilities are available on a number of hacker sites which can decompress an executable that's been crunched by one of these products. In principle, there's nothing to prevent decompressors being written for *any* of these tools, but most of the compressor authors are aware of the problem and seeking to present a difficult (and possibly moving!) target to anyone who fancies having a go at cracking the decompression scheme that's used.

Conclusions

In this month's column, I've tried to demonstrate how easy it is to peek inside (and ultimately, modify) Delphi-authored applications, firstly by making use of a couple of commercially available tools, and then by creating a utility of my own. Finally, I've demonstrated how to largely avoid this problem by making use of one of the compressors that are currently available. If you're a shareware author who releases trial/limited functionality software, I hope I've convinced you that investing in a compressor is a worthwhile thing to do.

No discussion on compression technology (and anti-hacking techniques in general) would be worthwhile without giving you a few other assorted tips.

Firstly, don't use packages unless you really have to. A program that's linked with Delphi's runtime packages must necessarily expose a lot of runtime type information in order for the dynamic-link magic to work. At the same time, a custom package must expose even more information since, by default, every non-private method of every object will be exported. If you've ever wondered how the Delphi IDE works, then look no further than Delphi version 4.0 which exposes pretty well everything that's going. Enough said...

Another tip is to remove the `PACKAGEINFO` resource before compressing your executable. This resource gives the prospective hacker a big hint when trying to decompile your program, because it indicates the name of every unit that's been linked into the EXE, this information is used by (for example) Merlin's Executable Viewer, cousin to the Resource Viewer that I've discussed earlier. Be aware that a few applications object violently to having their `PACKAGEINFO` resource removed, so you will have to suck it and see. Also, you should never remove resources from an executable *after* it has been decompressed because this will typically upset the decompressor, which thinks it's being short-changed. Instead, remove any unwanted resources before compressing the executable.

Finally, bear in mind that it's possible to compress the same executable more than once. The authors of ASPack specifically advocate this as an anti-hacking technique. Some compressors won't allow you to compress an application multiple times because they specifically check for the presence of their own decompression signature. However, you can always achieve the same effect by compressing the same executable twice using different compressors. You will probably appreciate that compressing a file that's already been compressed once will often result in a slight *increase* in file size, but that's immaterial, what we're trying to do is make life more difficult for the would-be hacker.

➤ *Table 1*

Shrinker	www.blinkinc.com A relatively expensive, but well-respected commercial compressor. This is the only one which works with 16- and 32-bit executables. Unfortunately, a 'de-shrinker' utility has recently appeared on various hacker websites.
Petite	www.icl.ndirect.co.uk/petite/ An excellent British product, enthusiastically supported by its creator, Ian Luck. Ian is aware that some decompressors have been compromised and is keen to make life as difficult as possible for the would-be hacker.
ASPack	www.alenka.spb.ru/aspack/ Another very nice Russian compressor (written in Delphi) that has some innovative features such as the ability to change the section name containing the decompression code, and to try out the compressed executable before committing.
PkLite	www.pkware.com/download.html For 16-bit executables only. Somewhat dated. A number of hackers' utilities are available for decompressing executables created with PkLite.

Dave Jewell is a freelance consultant, programmer and technical journalist specialising in system-level Windows and DOS work. He is Technical Editor of *Developers Review*, also published by iTec. You can reach Dave by email at Dave@HexManiac.com