Protecting the Oracle

–

A proof of concept for a Delphi obfuscator

Table of Contents

Illustration 1: DFM tree before and after obfuscation

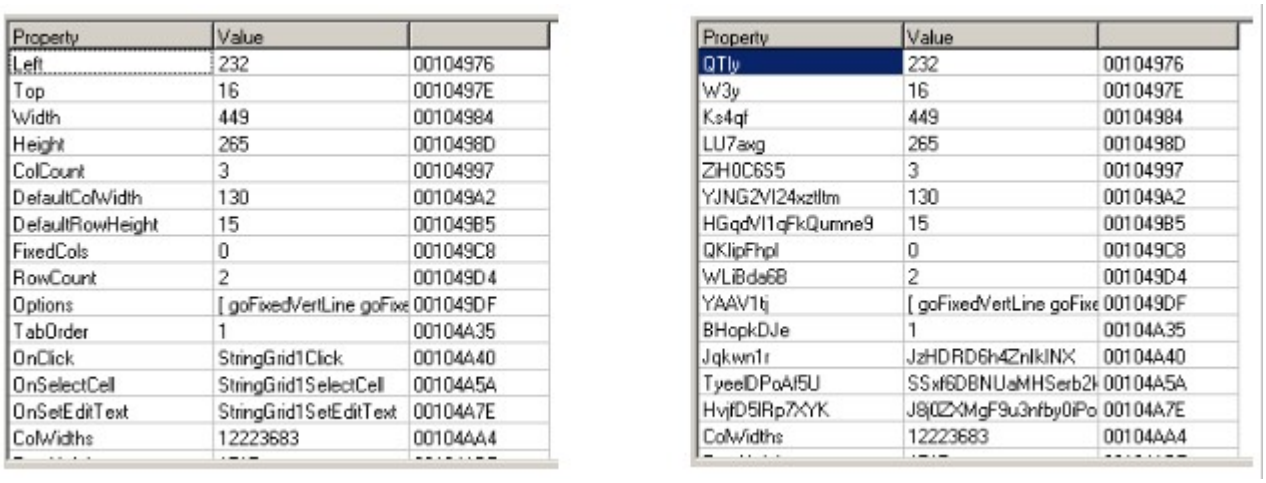

Illustration 2: Properties of a StringGrid control before and after obfuscation

0. Introduction

The advent of DeDe, the self-proclaimed Delphi Decompiler, in 1999 started a new era in reverse engineering programs compiled with Borland Delphi. In case you're not aware of this tool, the term decompiler is actually misleading to describe the functionality of DeDe as it implies that the tool can reconstruct Delphi source code from binary files. This is definitely not the case though.
What DeDe really is is a class browser that allows the user to browse the meta-data of all classes derived from TObject that are used in the binary file (that's equivalent to the classes of the VCL). It's basically an enhanced version of the Delphi object inspector that's used during Delphi development but it works for binary files instead of Delphi source files.
DeDe also comes with other features like a built-in disassembler but these other features were neither new nor as revolutionary as DeDe's core functionality.

1. What exactly is the problem?

At first glance the insight DeDe offers doesn't seem to be much of a problem. After all there's no critical data in the properties of the components you set in the object inspector when you implement your software. This line of thinking is fundamentally flawed though. There actually is critical data in these properties, and lots of it. Otherwise DeDe would have never had such an impact on reverse engineering Delphi binaries.
The critical data I'm talking about in the context of reverse engineering Delphi files are the names of the properties and some of their values.

Imagine a shareware program that can be registered using a standard name/password combination the user enters in a special dialog. The password can be entered in edit fields and the user confirms his input by clicking a button. Now what's more helpful to a person who wants to gain access to the important parts of the registration logic with the goal to crack the software? A resource named RegisterDialog or a resource named G5gAQLRICMZPIU?
What about a button named RegButton with a corresponding OnClick property called RegButtonClick? This is certainly a dead give-away for any potential attacker and because DeDe can resolve the addresses where these events can be found in the file and perform an instant disassembly of the code there the attacker could locate the critical code in less than three seconds.
Would a button named HoiDDdf4 with a property named KLF442E and a value like JHIogeeGEIffdF be just as valuable to the attacker, especially when it's buried in hundreds if not thousands of equally meaningless names? I think not.
And that's exactly what this paper intends to show you: How to make DeDe completely useless by obfuscating the data DeDe reads from Delphi binary files.

2. An explanation of how DeDe works and an overview of what's coming

Delphi binaries contain two different sets of data which DeDe takes and combines to display interesting information.
The first of these two sets is the Delphi formular data (hereafter referred to as DFM data). This is the data you might be familiar with from your own Delphi development. It's nothing but a textual representation of all the component and control names of the project and their properties that can be accessed and changed through the object inspector during development.
The other dataset (hereafter called VMT data) is the data that can be found in the so called virtual method tables (VMTs) of all classes that are either direct or indirect child classes of

TObject. These virtual method tables are the meta-data which describe the entire class hierarchy of the classes derived from TObject. This includes but is not limited to the names and types of the class properties, the methods belonging to the classes and a whole lot of other data. An in-depth overview of this VMT data is given in chapter 5.
The tool I'm going to introduce as a proof of concept for how I'm suggesting to combat DeDe and other tools of it's kind is named Pythia after the priestess of the oracle of Delphi. It can modify the DFM and VMT data of existing binary files in such a way that it's rendered useless for any person attempting to reverse engineer the software while the program itself will keep working just like the unobfuscated program. Pythia does basically exactly what DeDe does but instead of presenting the data from Delphi executable files it'll obfuscate it.
In the coming two sections I'm going to introduce the data structures used to hold the DFM and VMT data in Delphi executables as far as I've succesfully reverse-engineered them myself during the development of Pythia. Afterwards I'm going to give an overview of the process of connecting and obfuscating DFM and VMT data and the problems that come with that process.
As far as I know the information I'm going to present is valid for Delphi executables compiled with Delphi 5 or later. Coincidentally it also applies to C++ Builder files compiled with BCB 5 or later. This is because C++ Builder like Delphi uses the VCL (Visual Components Library) for all forms development. Consequently Pythia should work with executables compiled with version 5 of higher of Delphi or C++ Builder.

3. The DFM data

The Delphi form data is the less complicated of the two datasets used by DeDe and Pythia. It's basically a straight-forward translation of the textual representation of controls of a Delphi project into a binary representation. The textual representation can be found in the DFM files of a project during development.
Locating DFM data is easy, it can be found in the resource directory of Delphi executables. DFM data resources are always of type 0x0A (RT_RCDATA) and they can easily be distinguished from other resources of the same type by checking the first four bytes of the resource. DFM resources always begin with the four bytes long string „TPF0“.
After the „TPF0“ identifier the real DFM data begins. It's structured in a hierarchical tree that represents the placement of controls on a form. The root element of a DFM resource is always the form itself. This form can contain controls like memos, labels or panels. Panels and other container controls can again contain other controls. This is a recursive process, the height of the DFM tree is arbitrary.
Every time a control is defined in the DFM resource it's type and name is given followed by it's properties and the definitions of it's sub-controls.

Let's have a closer look at the exact format of the DFM data now. Following the „TPF0“ string comes a Pascal-style string (PString) that identifies the name of the class of the form in that resource. Unlike C-style strings PStrings are not zero-terminated. To be able to determine the length of these strings they begin with a single byte that contains that length. The string data follows right after this first byte. On few occasions the length can also be given by a four bytes long integer number. This is the case when dealing with so called LongStrings. In this document the term PString always refers to strings of a maximum size of 255 characters, the use of LongStrings is mentioned explicitly wherever applicable.

Once in a while control definitions do not instantly begin with with the PString that gives the name of the class of the control. They begin with a single byte of the form 0xFX instead.

DeDe translates this byte by adding „inherits“ to the definition of the control. I haven't worked out the exact purpose of this byte but it appears to be irrelevant for obfuscation. While testing Pythia with random programs I've encountered values of 0xF1, 0xF2 and 0xF4 for this byte. It's likely that more of these values, especially 0xF3, exist. Pythia just skips over them.

After the string that specifies the type of a control (root-control or not) there's another PString that gives the name of the control object. Then the definition of the properties of that control begins.
Every property definition is of the same form. First there's a PString that contains the name of the property. This is followed by a single byte that indicates the type of the property. The third and last element of a property definition is the the value of the property. The format of the value depends on the type of the property. A list of property definitions goes on until a terminating zero byte is found instead of a new property name. At this point the definition of a new control begin or the resource ends.

Here's a list of the structures of the type specific data of the property values I've managed to find and figure out. The first column gives the value of the byte that identifies the property type. The second column contains the name of the type and the last column gives a brief description of what I've found out about these types.

ID	*Name*	*Description*
0x00	Variant	I've only encountered this property once during the development of Pythia. Unfortunately the value of the property was null then so I haven't been able to research the structure of this type thoroughly. When I looked at it that one time though it was a two bytes long value.
0x01	Array	An array is a linear list of types. All arrays I've seen so far contained Strings of resource type 0x06 but it's likely that the type of array elements can be any DFM type. The length of an array value depends on the elements of the array. Arrays are closed by a terminating zero-byte.
0x02	Byte	As the name implies byte values are exactly one byte long.
0x03	Word	Word values are two bytes long.
0x04	Dword	Dword values are four bytes long.
0x05	Double	I'm not really sure if this is really a double value. Judging from the names of the properties where I've seen this property type it's very likely though. The size of this resource type seems to be 10 bytes.
0x06	String	Values of type 0x06 are PStrings with a size of up to 0xFF bytes (the length of the string is given by a single byte).
0x07	Enumeration	The value of an enumeration in the DFM resource is the string representation of the enumeration value, for example the string “bsToolWindow” for the property BorderStyle. The structure of values of type 0x07 is identical to those of type 0x06.
0x08	Boolean false	The length of values of this type is zero because the type is already the value. Type 0x08 denotes boolean properties of value false.
0x09	Boolean true	The opposite of the boolean false value. Also 0 bytes long.
0x0A	Image	An image resource is a chunk of bytes that contains the data of an image. These resources begin with a four bytes long value that contains the size of the actual image data. There's an important fact to remember about these resources: The data of some of them (not all) begins with the classname of the type of the image resource. I've seen TBitmap and TIcon so far but there might be others. This needs to be kept in mind when these two classes are obfuscated later. Otherwise these images won't be loaded anymore.
0x0B	Set	The structure of a set can be compared to the structure of enumerations. The only difference is that while only one value of an enumeration can be selected as the value of a property an arbitrary amount of values from a set can be selected. Like arrays sets are zero-terminated.

ID	*Name*	*Description*
0x0C	LongString?	Possibly equivalent in structure to 0x14. I don't have any concrete example for this type of resource here but some of my older source codes indicate that I used to treat this type like resources of type 0x14.
0x0D	Nil	Another resource type of size 0. 0x0D represents the value nil.
0x0E	Record	This is probably the most complicated resource type. A record can contain elements and these elements are described by their property definitions inside the record. Records are basically trees that are comparable in structure to the entire DFM tree. The main difference is that records can only contain unnamed elements of one type. That means the elements inside the record are described by nothing more than their properties. Every element of the record is zero-terminated and the entire record itself is also zero-terminated.
0x12	Unicode String	A LongString where every character is 2 bytes long. If the size of the string is 20 the size of the actual resource is 2 * 20 plus the four bytes that contain the length.
0x14	LongString	A PString where the length is given in a 32bit value instead of an 8bit value.

Table 1: DFM types

Now that I've given a brief overview of the DFM structure as far as I've uncovered it let me give a practical example of one of these structures.

```
0005DE4C 5450 4630 0654 466F 726D 3105 466F 726D TPF0.TForm1.Form
0005DE5C 3104 4C65 6674 03C0 0003 546F 7002 6B05 1.Left....Top.k.
0005DE6C 5769 6474 6803 B802 0648 6569 6768 7403 Width....Height.
0005DE7C E001 0743 6170 7469 6F6E 0605 466F 726D ...Caption..Form
0005DE8C 3105 436F 6C6F 7207 0963 6C42 746E 4661 1.Color..clBtnFa
0005DE9C 6365 0C46 6F6E 742E 4368 6172 7365 7407 ce.Font.Charset.
0005DEAC 0F44 4546 4155 4C54 5F43 4841 5253 4554 .DEFAULT_CHARSET
0005DEBC 0A46 6F6E 742E 436F 6C6F 7207 0C63 6C57 .Font.Color..clW
0005DECC 696E 646F 7754 6578 740B 466F 6E74 2E48 indowText.Font.H
0005DEDC 6569 6768 7402 F509 466F 6E74 2E4E 616D eight...Font.Nam
0005DEEC 6506 0D4D 5320 5361 6E73 2053 6572 6966 e..MS Sans Serif
0005DEFC 0A46 6F6E 742E 5374 796C 650B 000E 4F6C .Font.Style...Ol
0005DF0C 6443 7265 6174 654F 7264 6572 080D 5069 dCreateOrder..Pi
0005DF1C 7865 6C73 5065 7249 6E63 6802 600A 5465 xelsPerInch.`.Te
0005DF2C 7874 4865 6967 6874 020D 0005 544D 656D xtHeight....TMem
0005DF3C 6F05 4D65 6D6F 3104 4C65 6674 0270 0354 o.Memo1.Left.p.T
0005DF4C 6F70 0270 0557 6964 7468 03B9 0006 4865 op.p.Width....He
0005DF5C 6967 6874 0259 0D4C 696E 6573 2E53 7472 ight.Y.Lines.Str
0005DF6C 696E 6773 0106 0E54 6869 7320 6973 2061 ings...This is a
0005DF7C 2074 6573 7406 2174 6F20 6465 6D6F 6E73  test.!to demons
0005DF8C 7472 6174 6520 7468 6520 4446 4D20 7374 trate the DFM st
0005DF9C 7275 6374 7572 652E 0008 5461 624F 7264 ructure...TabOrd
0005DFAC 6572 0200 0000 0754 4275 7474 6F6E 0742 er.....TButton.B
0005DFBC 7574 746F 6E31 044C 6566 7403 9800 0354 utton1.Left....T
0005DFCC 6F70 03D8 0005 5769 6474 6802 4B06 4865 op....Width.K.He
0005DFDC 6967 6874 0219 0743 6170 7469 6F6E 0602 ight...Caption..
0005DFEC 4F4B 0854 6162 4F72 6465 7202 0107 4F6E OK.TabOrder...On
```

```
0005DFFC 436C 6963 6B07 0C42 7574 746F 6E31 436C Click..Button1Cl
0005E00C 6963 6B00 0000                          ick...
```

This DFM resource defines a form named Form1 of type TForm1 with two controls: A TMemo object named Memo1 and a TButton object named Button1. Several different property types are used for the properties. How to interpret the values of these properties is described in the table above.

4. The VMT data

The second dataset, the virtual method table data, is a bit more complicated. It's harder to locate than the DFM data and it's structure is more complex. VMT data is used to describe the classes that derive from TObject at one point. There's exactly one VMT in the executable file for every class used by the program.

The most important of all the structures I'm going to describe in this section is the so called VMT table. The following table shows the layout of a VMT structure. It's been taken from Brian Long's paper „Debugging With More Than Watches And Breakpoints“[1] which he presented at BorConUK 2001. I've marked the parts which are important for DeDe and Pythia.

Constant name	*Offset*	*Description*
vmtSelfPtr	-76	Address of first VMT entry if any, or of classname
vmtIntfTable	-72	Address of implemented interface table
vmtAutoTable	-68	Address of automated class section (Delphi 2)
vmtInitTable	-64	Address of table of fields requiring initialization
vmtTypeInfo	-60	Address of RTTI
vmtFieldTable	-56	Address of published field table
vmtMethodTable	-52	Address of published method table
vmtDynamicTable	-48	Address of DMT
vmtClassName	-44	Address of classname string
vmtInstanceSize	-40	Number of bytes of instance data required by object
vmtParent	-36	Address of ancestor class VMT
vmtSafeCallException	-32	Address of virtual method, SafeCallException
vmtAfterConstruction	-28	Address of virtual method, AfterConstruction
vmtBeforeDestruction	-24	Address of virtual method, BeforeDestruction
vmtDispatch	-20	Address of virtual method, Dispatch
vmtDefaultHandler	-16	Address of virtual method, DefaultHandler
vmtNewInstance	-12	Address of virtual method, NewInstance
vmtFreeInstance	-8	Address of virtual method, FreeInstance
vmtDestroy	-4	Address of virtual destructor, Destroy

Table 2: Structure of virtual method tables

1 http://www.blong.com/Conferences/DCon2001/Debugging/Debugging.htm

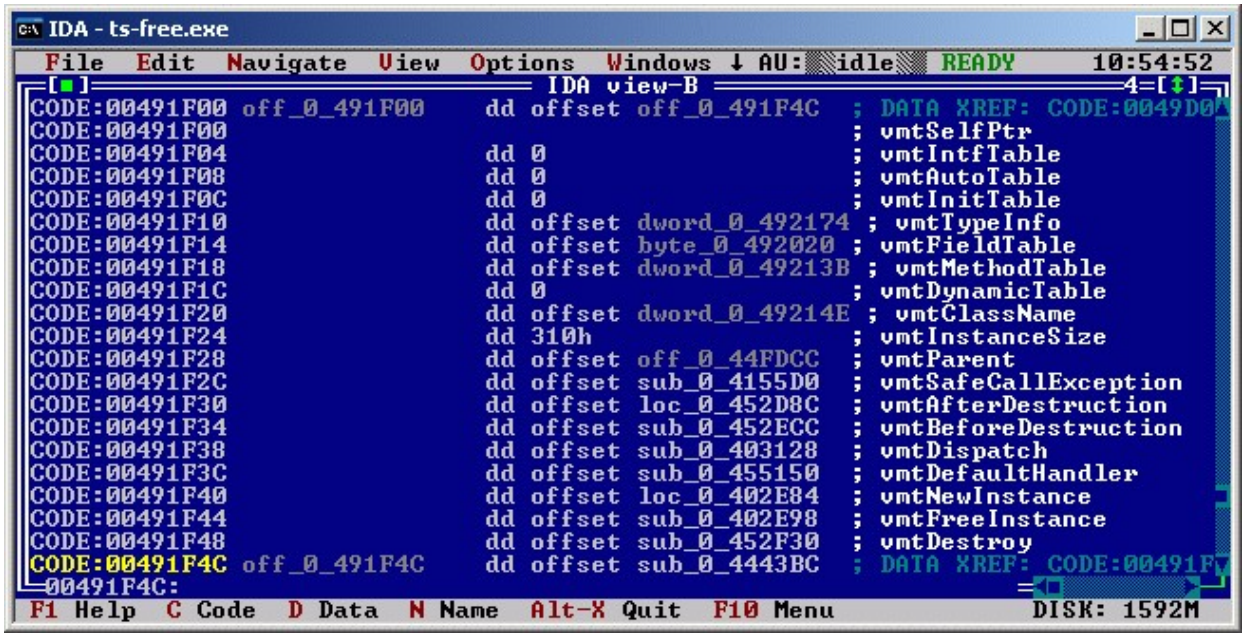

Illustration 3: Example of a VMT Table in a disassembled file.

An entire VMT structure is 76-bytes long. It begins with a pointer (called vmtSelfPtr) that points to the virtual address that's 76 bytes after the virtual address of the beginning of the VMT (that's the byte right after the end of a VMT table). This is how Pythia recognizes VMTs. Pythia searches through the entire file and attempts to locate offsets X that contain a value that's the virtual address of the virtual address of X + 76. Some more validity checking is performed and when this also passes succesfully a VMT was recognized.

Another simple entry in the VMT structure is the field called called vmtClassName. This field contains the virtual address of a PString that's the classname of the class the VMT belongs to.

vmtParent contains the virtual address of the vmtTypeInfo structure of the parent class of a class.

The first of the more complicated elements of the VMT structure is vmtTypeInfo. vmtTypeInfo contains the virtual address of the TypeInfo structure of a class (this would also be the value of the vmtParent entry of sub-classes of that class).
TypeInfo structures contain the meta-data that defines the properties of a class. Here's the general layout of these structures.

Offset	*Size in bytes*	*Description*
0	1	The purpose of the first byte is unknown.
1	X	A PString that contains the name of the class the TypeInfo structure belongs to.
X + 2	4	Equals the value of the vmtSelfPtr value of that class.
X + 6	4	Appears to be a pointer to a pointer to the TypeInfo structure of the parent class.
X + 10	2	Another word value of unknown purpose.
X + 12	Y	The name of the package the current class belongs to.
X + Y + 13	2	The number of PropertyInfo blocks to follow.

Table 3: Layout of a vmtTypeInfo structure

This block is immediately followed by the number of PropertyInfo blocks specified in the last 2-byte value of the TypeInfo structure.

Here's the general layout of a PropertyInfo block[2].

Offset	*Size*	*Description*
0	4	A pointer to pointer to the virtual address of the TypeInfo structure of the value type of that property.
4	4	A pointer to the virtual address of the get-procedure of that property.
8	4	A pointer to the virtual address of the set-procedure of that property.
12	4	A pointer to the virtual address of the stored-procedure of that property.
16	4	Index for array properties.
20	4	Default value
24	4	Index for indexed properties
28	4	Flags describing property procedures.
32	X	A PString that gives the name of the property that's being defined in the current PropertyInfo block.

Table 4: Layout of a PropertyInfo structure

2 See http://www.blong.com/Conferences/BorConUK98/DelphiRTTI/CB140.htm and http://www.freepascal.org/docs-html/rtl/typinfo/tpropinfo.html for more information.

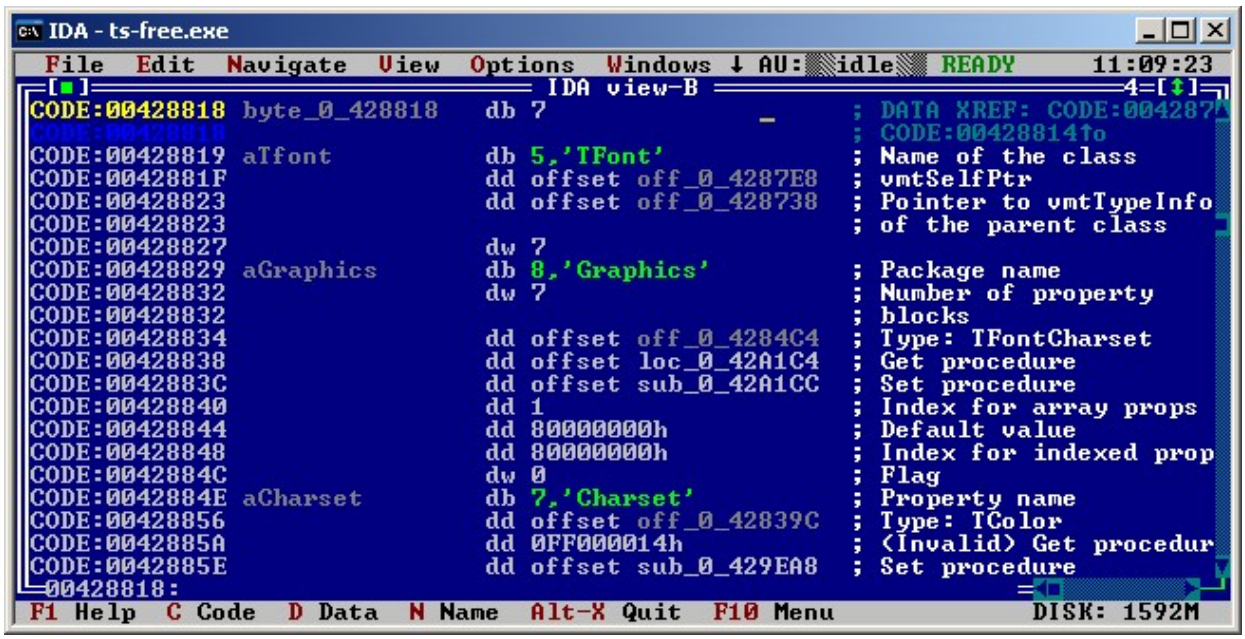

Illustration 4: Disassembled vmtTypeInfo block for class TFont.

The next structure I'm going to introduce now is the FieldTable structure which can be found at the virtual address of the vmtFieldTable value of a VMT.

Offset	*Size*	*Description*
0	2	The number of field blocks to follow.
2	4	A pointer to a list that contains pointers to the TypeInfo blocks of the classes of all fields. The first element of that list is a word value that gives the number of pointers in the list.

Table 5: Layout of a vmtFieldTable structure

Right after the FieldTable structure comes the promised number of field blocks. These are of the following structure:

Offset	*Size*	*Description*
0	4	Unknown purpose
4	2	Unknown purpose
6	X	A PString containing the name of the field.

Table 6: Layout of field block structures.

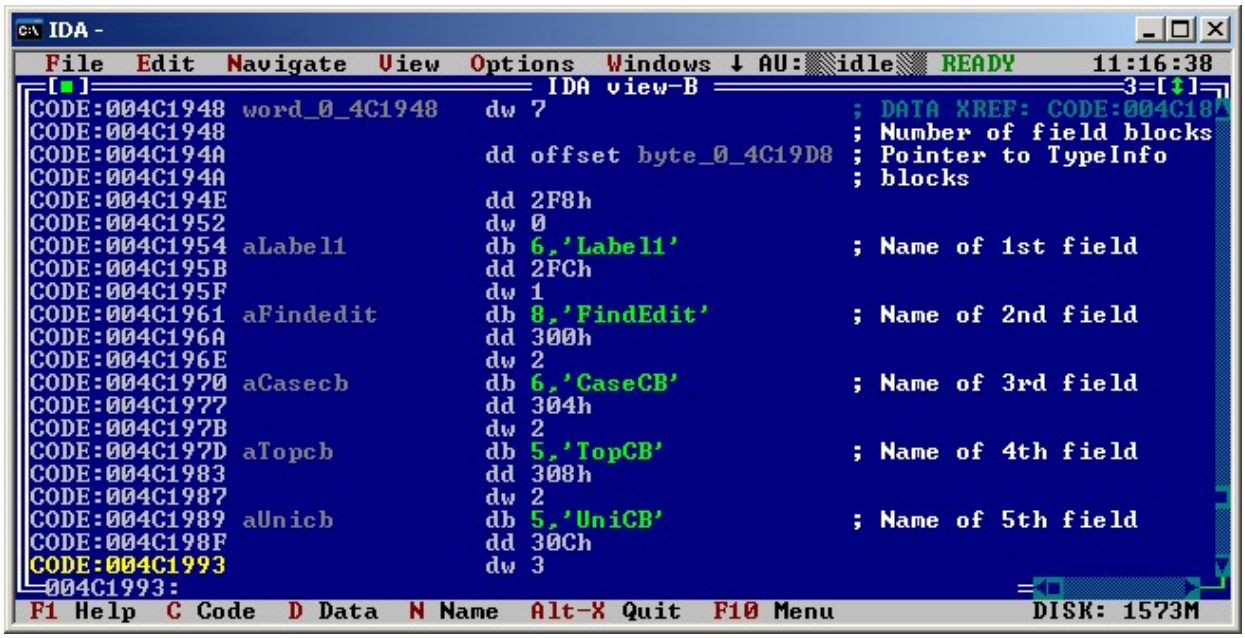

Illustration 5: Disassembled vmtFieldTable block.

The last structure of relevance for Pythia is the MethodTable structure. Information about the methods of a class are stored here. A method table begins with a single word value that contains the number of MethodInfo blocks to follow. MethodInfo blocks have the following structure.

Offset	*Size*	*Description*
0	2	An id for the method.
2	4	The virtual address of the method.
4	X	A PString that contains the name of the method.

Table 7: Layout of vmtMethodTable structures

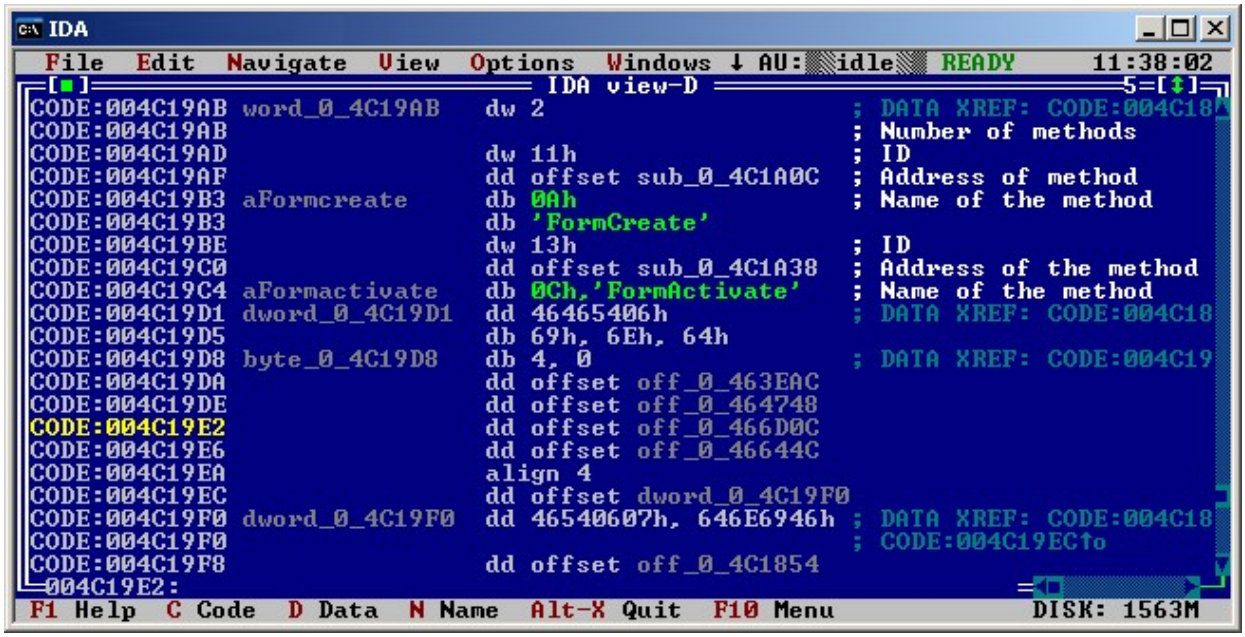

Illustration 6: Disassembled vmtMethodInfo block.

If you recall the example I've given in the introduction you might realize that this is maybe the most important structure of them all. This is where the method identified by DeDe at some offset turns from RegBtnClick to GboeBOIfDD3ge.

5. Combining DFM data and VMT data

Now that the necessary structures for obfuscation have been introduced it's time to explain how they are connected as that's one of the main problems for obfuscation. If the two trees are not perfectly synchronized anymore after obfuscation the program will refuse to start (in the best case) or generate more or less subtle errors (in the worst case).

Pythia stores all string values it can find in DFM data and VMT data in dynamically allocated strings. After reading the two datasets all the structures are connected in the way I'm about to describe. String values are connected by not more than exchanging pointers in these structures so that strings of equal value really point to the the same string data. This simplifies the process of obfuscation tremendously. Otherwise it would have been necessary to keep the DFM and VMT tree synchronized which while certainly possible is quite complicated. As soon as all string values were synchronized like that it's enough to iterate through all members of the VMT tree and to obfuscate the string values in that tree. The DFM tree is updated automatically due to the shared pointers.

Synchronizing the classnames found in the DFM data with those of the VMT data is simple because of the unique nature of classnames in a Delphi file. It's therefore enough to walk through the entire VMT tree until a VMT with that classname was found.

It should be noted that Pythia does not obfuscate the classnames of classes that are found on the top level of the DFM data (TForm, ...). When attempting to do this unresolvable errors occured. The name of the resource that holds the DFM data equals the classname of the top-level control in that resource. Changing the resource name is also necessary when obfuscating classnames of top level classes but it didn't solve the problem in all cases. In some cases renaming the resource too worked but I've also encountered several very weird

cases, for example one resource named TAboutBox. It was possible to rename this resource to TAAXXXXXXX or any letter actually for the Xs. It was not possible to touch the first two letters in any way. And the third letter could be changed from anything between 'a' and 'o' but as soon as I changed it to 'p' or a higher letter the program refused to start. That's very weird behaviour I can not yet explain.

Resolving the names of the controls isn't too tough either. The names of top-level controls have no dependencies, they can be changed at will. They are also not part of the VMT structure which means they are not automatically obfuscated after the entire VMT tree was changed.
Child controls on the other hand are part of the VMT structure. They are defined in the vmtFieldTable block of a VMT. Resolving names of child controls found in the DFM data is therefore merely checking the field table of the right VMT. Which VMT to check is also very straight-forward. Even though child controls can recursively be placed inside other child controls the VMT that contains all field definitions is always the VMT of the top-most control (the form). Checking the VMT of the top-level control is therefore enough.

The last things to obfuscate are the names of the properties and their values. Here's where things get a bit more complicated.
Let's start with the names of the properties. In the easiest case looking up a property name is nothing more but searching the vmtTypeInfo structure of a property with a given name. If the property can't be found there it must be looked up recursively in the VMTs of all parent classes of that class until TObject is reached. If the property can't even be found there it's not part of the VMT tree. This is rare but valid. On very few occasions Delphi does not put properties into the VMT tree. The property TextHeight, for example, is one of these. I don't know the reason for this but properties that can't be looked up in the VMT tree also can't be obfuscated by Pythia.
Another major problem are properties that derive from TCollection. Due to the nature of collections in Delphi it's apparently not possible to find out what elements are inside a collection by merely looking at the VMT tree. Pythia contains a list of known collections which is of course by no means complete. This list knows which elements are inside a collection with a given name. If new collections are found they must be added to this list too.
Another thing to look out for are qualified property names. These names which appear only in DFM data contain a '.' character. An example which appears in basically every Delphi executable is a property named „Font.Charset“. Not a single control has a property of that name but many controls have a property named „Font“ which has a property named „Charset“. And that's how qualified property names have to be looked up. Starting with the first of the period-separated segments of the name all individual segments have to be looked up in the context of what the look-up of the former segment returned. All but one of the property names I've seen during testing consisted of only two segments but that one other property which had 3 segments seems to be sufficient proof that the number of segments is probably arbitrary.

The values of properties are even more difficult to look up and obfuscate. There is no simple link between the values of properties and the VMT although one could probably construct at least a weak link by checking if the type of the property is at least of the same type as the value of the property.
Pythia doesn't do that though. Pythia merely compares strings. If the value of a property matches a field or method found in the VMT it'll be obfuscated. That works well for

property values like acAction1 or FormCreate. This doesn't work nearly as well for property values like Left which might be a completely valid caption for a label and a completely valid name of a property at the same time. Nevertheless Pythia would obfuscate the caption of the label too.
Qualified values are also a possibility. They must be looked up just like qualified property names.

The individual segments of qualified property names and qualified property values can probably be either names of classes, fields, properties or methods. It's noteworthy that case does not seem to be important when comparing strings from the DFM data with strings from the VTM data. At first glance it seems that equality between strings is determined case-insensitively, the strings in the DFM data appear to be case-sensitively equal to the corresponding string in the VMT. I've experienced a single case where this was not true though. In one file I used during testing there was a method called „FilterEditChange“ defined in the VMT data, the string in the DFM data was written „FiltereditChange“ though. This might happen because Delphi is not case-sensitive and the programmer might have made a typo when assigning this method to the OnChange event of a control.

6. Problems and ideas

Some of the problems were already mentioned above but let me reiterate them and add some other problems to think about.

- Some properties can't be looked up. This is not a major problem because it only happens very rarely and I've never seen it happen for an interesting property.
- Top-level classnames can't be obfuscated. This can be a potential problem if classnames like TRegisterDialog are used.
- The obfuscation method only works for Delphi binaries with statically linked packages because dynamically linked packages don't contain the VMT tree (which can then be found in VCLxx.BPL; xx depends on the Delphi version). This is generally a non-issue because basically all Delphi binaries are linked statically. For the few remaining Delphi programs using dynamically linked packages it might be possible to use a package file that was obfuscated just like the EXE file.
- Obfuscating the values of properties is problematic because valid strings might be obfuscated too (see the „Left“ example in 6). This problem could be cured by introducing whitelists or blacklists to the obfuscator where the user can specify property values that should (not) be obfuscated.
- The obfuscation of type names has the potential to make features like error-reporting useless. This is partly true when using Pythia in it's current form. It's easy to produce a log of what exactly was obfuscated and using that log it's possible to determine the original name of an encrypted type. A significantly better idea would be the extension of the aforementioned whitelists and blacklists to cover type names (and all other strings which are potentially obfuscated) too.
- The TCollection issue as already explained in section 6 could be solved better, for example using a config file where the user can add the names of types derived from TCollection and the type of the elements of that collection.
- The effects of obfuscation on RTTI have not been researched.

- Unobfuscated strings like captions still exist and in the face of obfuscated strings they might become a valuable piece of information for any attacker. Obfuscating the TRegDialog is nearly useless if the caption of the dialog still reads „Register...“. This is not a problem of the obfuscator though, it's a problem of the software that's obfuscated.
- The process I've described is most likely not valid for new Delphi.NET executable files. This is not a real problem though as DeDe is probably not going to work with them either.
- Pythia does not obfuscate the string representations of enumeration or set values although this would certainly be possible.
- Pythia only uses a small part of the VMT. Other parts can also be used to fool DeDe or similar tools. For example it might be possible to change class hierarchies at run-time by manipulating VMTs or it might be possible to crash DeDe by creating VMTs that are complete bogus and point to invalid addresses.

7. Conclusion

The paper has shown that it's both possible to render DeDe useless and how to implement an obfuscator that automates the process. It's not complete by any means but it's a good first step for someone who has more time and dedication for such a project than myself. Nevertheless it should not be forgotten that security by obscurity is not really security at all. Obfuscating Delphi binaries the way I've introduced it is merely a small part in the protection of a piece of software.