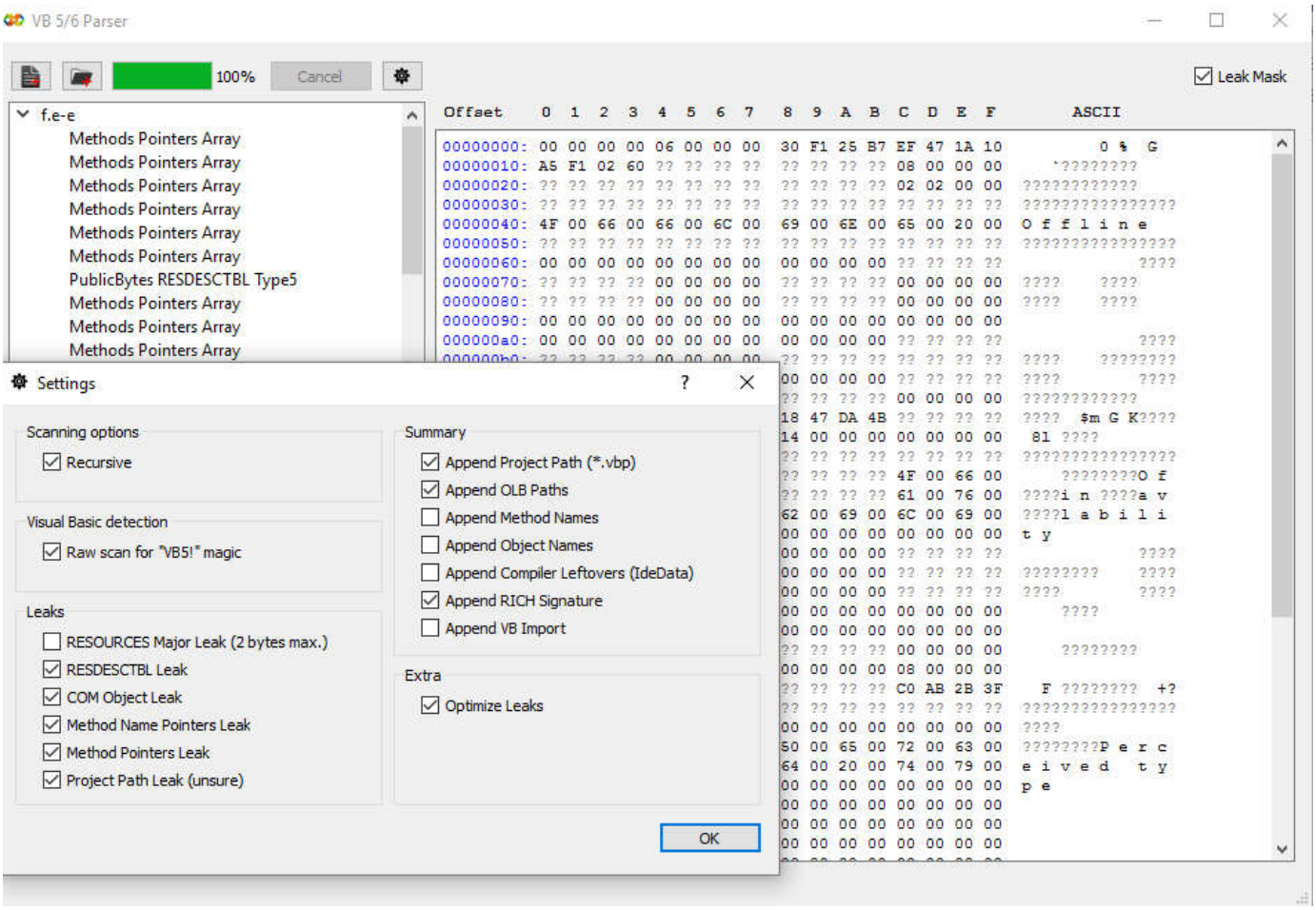
sysenter-eip.github.io

Visual Basic 5/6 compiler memory leak inside created executable files



SRC: GITHUB BINARY: RELEASES

Intro

It's the 2019 year now but there's still something to be discovered about Visual Basic 5/6. From the realistic point of view, official compilers were discontinued a long time ago. Maybe the authors of malware cryptors or open source projects (**PDFStreamDumper**) will disagree, but I think the platform is now dead. Still, you can write programs and they will most likely work normally under latest Windows 10 builds.

This article was created when I was asked «to identify the source of some strange information in VB6 binaries and to extract it reliably». This task was given by professor **Igor Yurin** from the university I was studying/working at (**Saratov State University**). So he was the first who noticed this anomaly.

(I was unable to find any other research on this topic, so if you know about any other – contact me

and I will add links here)

The research itself was done about three years ago and remained unpublished due to being incomplete and not useful practically. Until the moment *delete* or *publish* happened. I decided to summarize all I have.

I organized some information about internal structures of VB5/6 files, it is not very important for detecting the main topic, but I still include everything. If you are not interested just read next section and jump to the end (Vulnerability section).

Sorry for the grammar (please, contact me on twitter @sysenter_eip if something needs to be fixed).

TL;DR

While VB5/6 compiler is doing his job (compiling and stuff), it uses memory from the process heap to build objects that later are to be saved into output binary. This memory is not zeroed hence some objects can contain uninitialized values. This memory can be statically extracted and it can contain some interesting data (like source code and environment variables).

Compiled binary for windows and its source code can be found HERE.

Code quality is low, but it's working. I will probably make a console utility for Linux/Windows so researchers with large malware collection can scan their files to see if anything interesting can be extracted. (once done link will appear HERE)

Information sources

First of all we need to acknowledge important research on this topic. First one is an old research by Alex Ionescu [1]. Second good source of information is the source code of Semi-VBDecompiler [2].

I was also able to obtain some debug symbols for Virtual Machine and compiler. It looks like it was supplied with Visual Basic 6.0 Enterprise Edition.

Name	Version	Size	md5
msvbvm60.dbg	6.0.8268	dbg : 2 826 996 dll : 1 409 024	dbg: 8D4E57DC2A426CA2FB79BC1900F7B544 dll : 8D4E57DC2A426CA2FB79BC1900F7B544

Name	Version	Size	md5
vb6.dbg	6.0.8176	dbg :3 747 408 exe: 1 880 064	dbg: C84ADEFDE7428C6C31F1BF780DA87A4C exe: 037C4B5B4CD2809DB33A09BBB37CC693
vba6.dbg	6.0.8169	dbg :1 179 760 exe: 1 701 648	dbg: 48552645E93A2FD65F8683147822AA5E exe: C3D0CA107B96837748373563088B73E8

Table 1. Debug symbols

I was not able to use vba6.dbg file (most likely it doesn't contain anything useful at all), while other two were pretty helpful since they contain function names with prototypes.

In this article I will be referring to the names from debug information if possible. If no symbol are present then use the decompiler [2] notation.

Most things described here are for Visual Basic 6, but it should be also applicable to VB5.

Sometimes if no symbols are present then I will use VA (Virtual Address) as if file was loaded at default ImageBase (no ASLR, so it is more convenient).

Visual Basic 5/6 File Format information

File detection and locating EXEPROJECTINFO

All programs created by VB5/6 compiler are 32-bit PE format applications. No compilers were created for 64-bit, since platform was deprecated long before 64 become popular.

The first important difference from the usual PE files (Unmanaged VisualC++, Delphi) is the import table. It usually contains only one library (MSVBVM60.DLL for VB6 or MSVBVM50.DLL for VB5) – the runtime and virtual machine. This library contain functions actively used from VB program. Besides it will also do an environment initialization and pass the control to program's real entry point (think of it as Main) if binary is *Native*. For the *P-Code* this runtime will be also interpreting the bytecode.

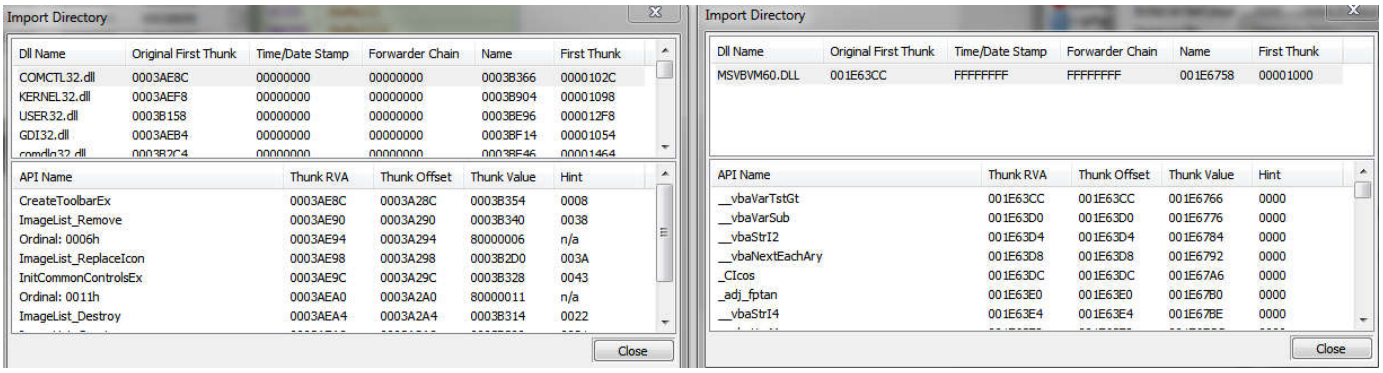


Figure 1. On the left import of the program compiled with MSVC++, on the right – VB6.

Every VB5/6 file has a structure **EXEPROJECTINFO**, which is essential for the runtime to do an initialization part.

Locating this structure can be extremely hard in general, but if no modifications were made to the executable this task is simple.

For exe files pointer to this structure can be found at the EntryPoint. It always looks like shown on the Figure 2.

```
.text:004015F0                  ; ---------------------------------------------------------------------------
.text:004015F0
.text:004015F0                                  public start
.text:004015F0                  start:
.text:004015F0 68 38 17 40 00                   push    offset structEXEPROJECTINFO
.text:004015F5 E8 F0 FF FF FF                   call    ThunRTMain
.text:004015F5                  ; ---------------------------------------------------------------------------
```

Figure 2. EntryPoint of VB6 application. Highlighted bytes can be used as signature for locating EXEPROJECTINFO at EP.

A little different with VB5/6 libraries (dll files). At the EntryPoint there are no pointers to this structure. However such library will always have at least 4 exported functions: DllUnregisterServer, DllGetClassObject, DllRegisterServer, DllCanUnloadNow. Each one of them have a PUSH with desired pointer.

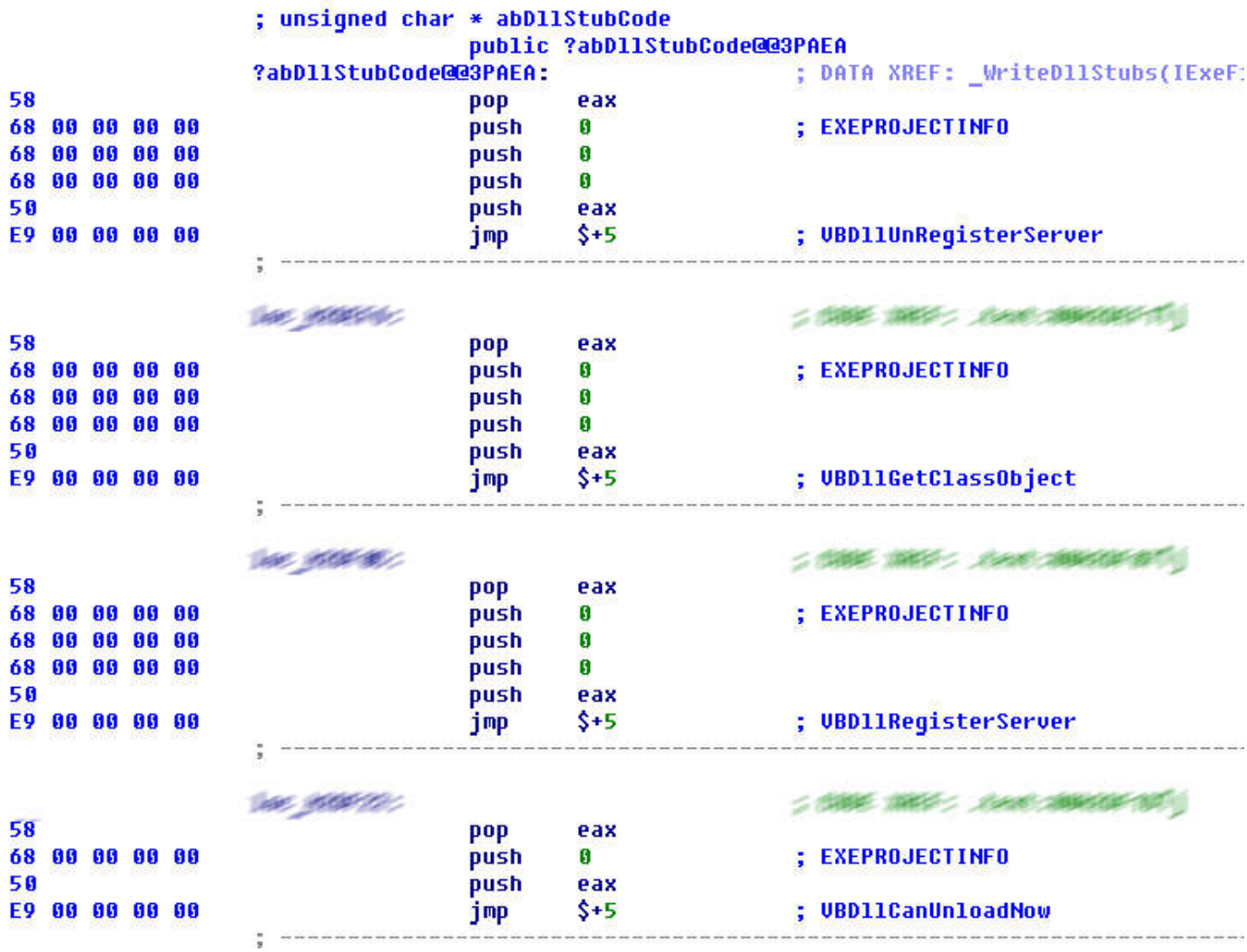


Figure 3. This is actually a code template compiler is using while building the libraries. Take a note that this template is continuous and could be found with exactly same layout in the output file. This means we could use it as a pattern for signature search to locate **EXEPROJECTINFO** without parsing an export table.

Compiler also have templates for the EntryPoint of exe and dll.

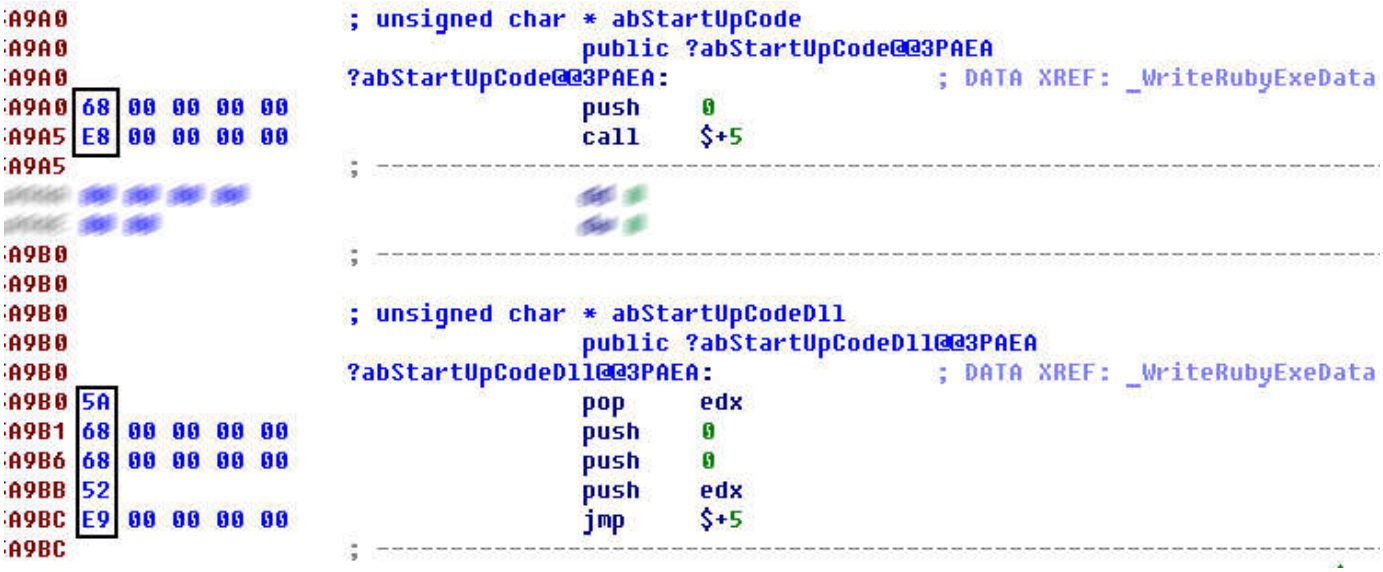


Figure 4. Template of the EntryPoint for exe and dll.

Nothing is stopping user from obfuscating the EntryPoint making extraction of this structure a very hard task in static (body of the structure itself can also be decrypted only in runtime). This leads us to the fact that the only way to locate it generically is by hooking the runtime itself. This is too complicated so we will rely only on signature scanning (which will work in most cases).

Next sections will describe structures which compiler is writing inside VB executable files. Each has a lot of field, some of them completely useless, some required to move deeper. I will be giving short summary of fields to highlight the most useful ones.

EXEPROJECTINFO

EXEPROJECTINFO (**VBHeader** unofficially) is the first main file structure (starting point), it contains links to other structures and also some information related to VB project (like user specified project name and description). Generation code is located in *vb6.exe :: WriteRubyExeData*.

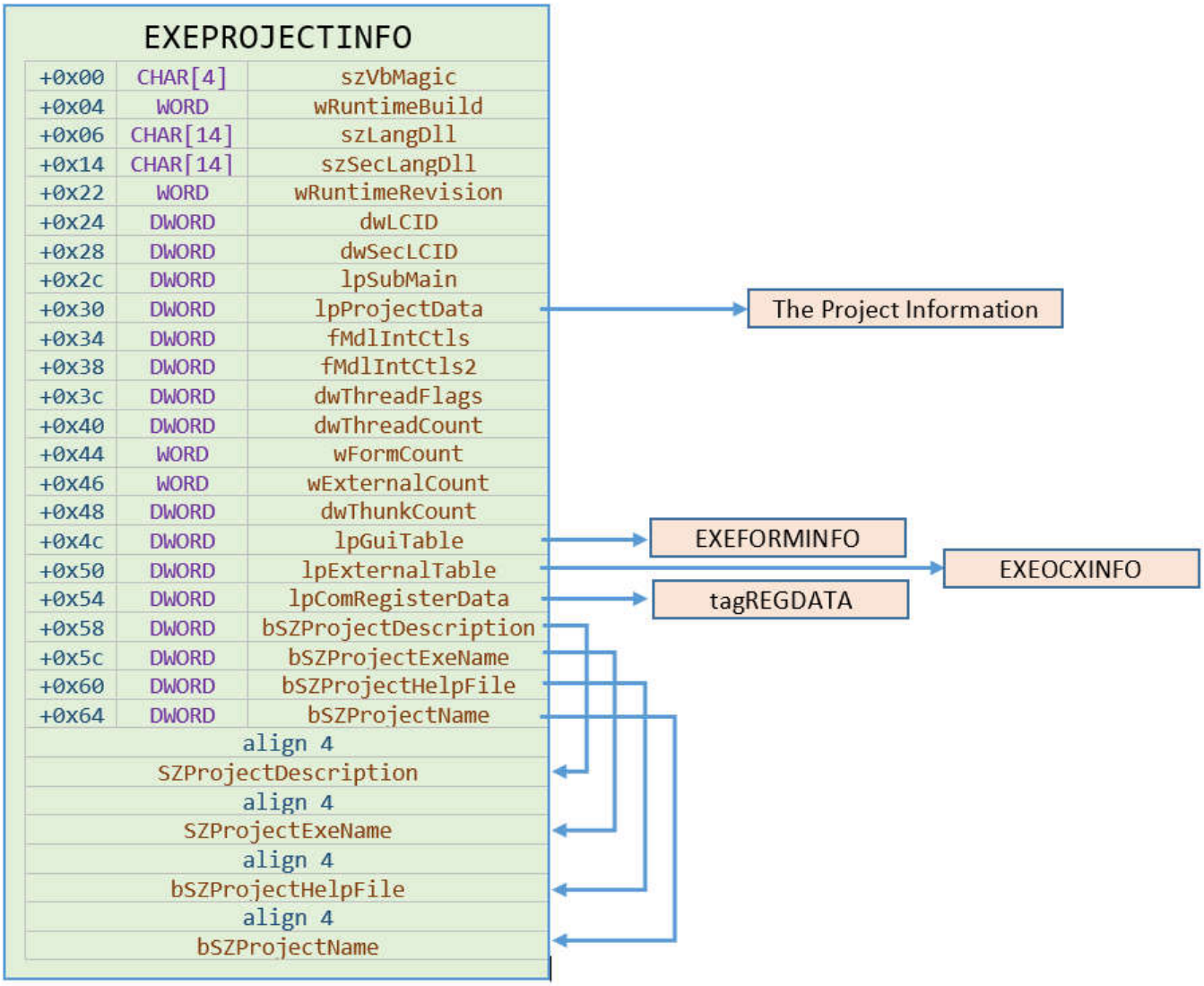


Figure 5. EXEPROJECTINFO

szVbMagic usually contains magic signature «*VB5!*» which is 0x21354256 in HEX. This marker can be used to statically detect this structure (official VB 5/6 compiler always fills this field), but runtime never actually checks for it, so in practice it can contain anything.

Another interesting field **wRuntimeBuild** contains version identifier of runtime used to build this file (it unique identifies compiler's vb6.exe). This allows to identify exact compiler version. For example, value 9782 (decimal) corresponds to latest Visual Basic 6 SP6.

```
:0045A2F7                mov     eax, [ebp+var_94]
:0045A2FD                mov     esi, offset _GUID_NULL
:0045A302                xor     eax, [ebp+arg_24]
:0045A305                lea     edi, [ebp+var_68]
:0045A308                mov     [ebp+EXEPROJECTINFO.szVbMagic], '!5BV'
:0045A312                mov     word ptr [ebp+var_B0+2], 0Ah
:0045A31B                and     eax, 1
:0045A31E                mov     [ebp+EXEPROJECTINFO.wRuntimeBuild], 8176
:0045A327                xor     [ebp+var_94], eax
:0045A32D                and     [ebp+var_94], 0FFFFFFDFh
```

Figure 6. Field **wRuntimeBuild** is set inside WryteRubyExeData.

szLangDll – byte array of size 14. In case the first byte is not «****» *(0x2a), field contains name of the language library (for example, «vb6ko.dll*» or«*VB6ES.DLL»). Can be used to identify localization of the system.

szSecLangDll – another 14 bytes array. This field contains either «*****» or «~». It might contain something besides mentioned, but no real samples were found.

lpProjectData is a pointer to structure **The Project Information**, original name is missing from debug symbols (unofficially: **tProjectInfo**).

lpGuiTable – pointer to an array of **EXEFORMINFO** (**tGuiTable**) structures. **EXEFORMINFO** is stored one after another in count of **wFormCount**.

lpExternalTable – pointer to the first structure of **EXEOCXINFO** (**tComponent**), which contains information about OCX modules used by this program. Count is at **wExternalCount**. Next **EXEOCXINFO** structure can be found at offset **EXEOCXINFO::dwStructSize** counting from the current one.

lpComRegisterData – pointer to **tagREGDATA** structure, which describes used COM objects.

bSZ* this group of fields store an offset to NULL terminated string (base at the beginning of the described structure). Each of the fields have self-describing names. For example ***SZProjectExeName contains name of the file without extension which compiler used to save output file. You should be careful while parsing this strings, since it's single byte encoding there can be chars which are specific to user's locale (ex. cp1251 charset for Russian locale).

Some of bit fields can be used to restore compiler's flags. Sources at [2] will be helpful in that.

The Project Information

Filled at *vba6.dll :: 0x0FB11783*.

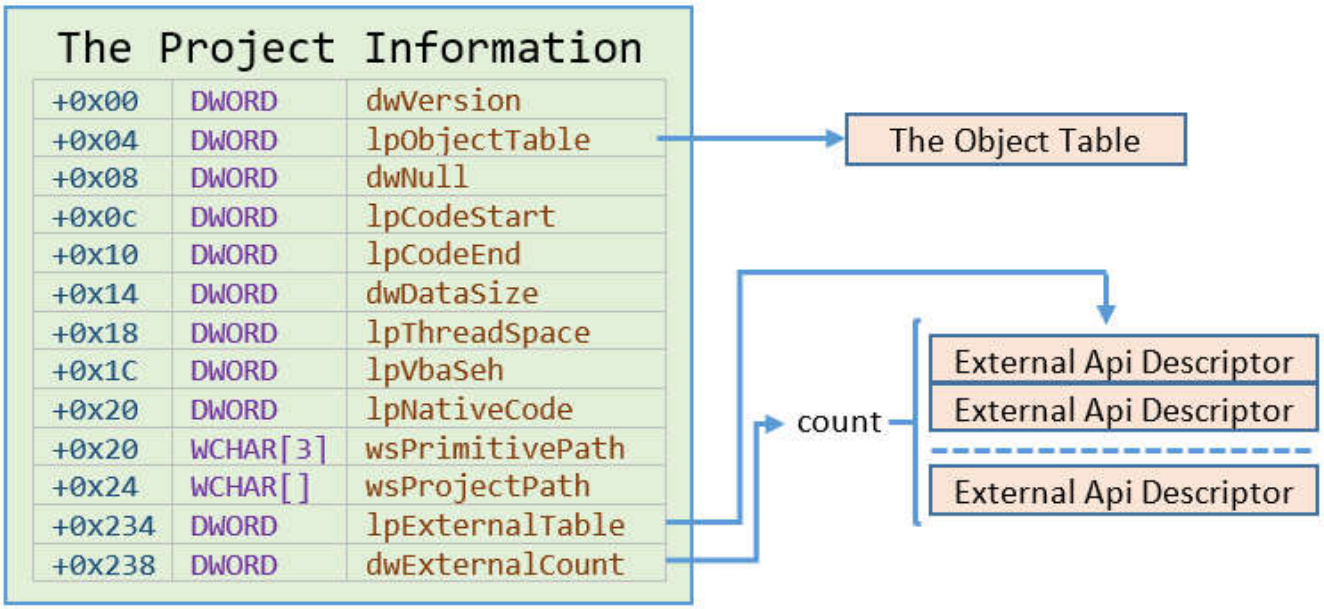


Figure 7. The Project Information

lpObjectTable – pointer to a table of program objects (**The Object Table**).

lpNativeCode – if this field != 0 then the project is Native, other way it is P-Code.

lpExternalTable – pointer to array of **External API Descriptor**, which describe WinAPI functions used by the app.

Next fields are ignored by runtime, but always filled by compiler.

dwVersion – contains 500 (0x1F4).

lpCodeStart – pointer to marker - DWORD, value 0xE9E9E9E9.

lpCodeEnd – pointer to marker - DWORD, value 0x9E9E9E9E.

wsPrimitivePath – originally part of **wsPathInformation**. It determines what data is stored in **wsProjectPath**. In all files I had checked it either contain zeroes (\0), making next field invalid, or contain UNICODE string «*\A». Code that was writing values to this field was removed in SP6 update.

wsProjectPath – Full path to *.vbp file, if wsPrimitivePath is valid.

EXEFORMINFO

By parsing this structure you can extract compiled **FORMS** from application. Field **lpFormBody** points to a blob of data. Decompilation of this data is possible by using «vb3 binary to text» tool [2]. Blob size is stored in **dwSizeOfFormBody**.

Filled inside vba6.exe :: WriteFormExeData.

EXEFORMINFO

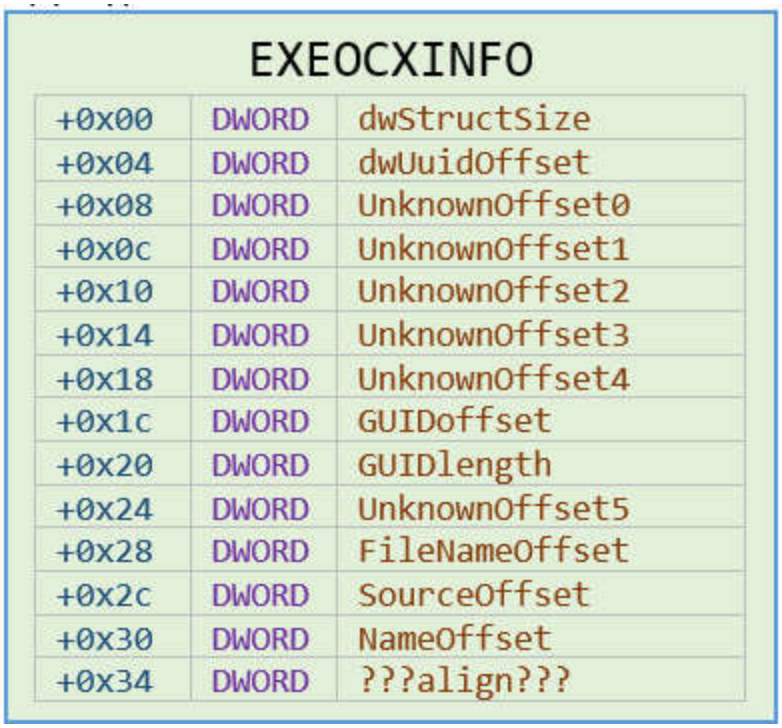
+0x00	DWORD	dwStructSize
+0x04	UUID	uuidObjectGUI
+0x14	DWORD	dwUnknown1
+0x18	DWORD	dwUnknown2
+0x1c	DWORD	dwUnknown3
+0x20	DWORD	dwUnknown4
+0x24	DWORD	lObjectID
+0x28	DWORD	dwUnknown5
+0x2c	DWORD	fOLEMisc
+0x30	UUID	uuidObject
+0x40	DWORD	dwSizeOfFormBody
+0x44	DWORD	dwUnknown7
+0x48	DWORD	lpFormBody

Figure 8. EXEFORMINFO

EXEOCXINFO

Descriptor of OCX files used. Purpose of the binary blobs contained in this structure are unknown.

You can find how compiler is working with it at *msvbvm60 :: CreateOcxDefFromExe*.



EXEOCXINFO

+0x00	DWORD	dwStructSize
+0x04	DWORD	dwUuidOffset
+0x08	DWORD	UnknownOffset0
+0x0c	DWORD	UnknownOffset1
+0x10	DWORD	UnknownOffset2
+0x14	DWORD	UnknownOffset3
+0x18	DWORD	UnknownOffset4
+0x1c	DWORD	GUIDoffset
+0x20	DWORD	GUIDlength
+0x24	DWORD	UnknownOffset5
+0x28	DWORD	FileNameOffset
+0x2c	DWORD	SourceOffset
+0x30	DWORD	NameOffset
+0x34	DWORD	???align???

Figure 9. EXEOCXINFO

tagREGDATA, tagRegInfo, The Designer Info

All offsets are from the base of **tagREGDATA**.

tagREGDATA::bRegInfo – offset to **tagRegInfo**.

tagREGDATA::bSZ*** – offsets to strings.

tagRegInfo::bNextObject – offset to next **tagRegInfo**, last object will have it zeroed.

tagRegInfo::fObjectType – one of the values from Table 2.

tagRegInfo::fIsDesigner – reserved on VB5.

tagRegInfo::bDesignerData – offset to **The Designer Info**, if **tagRegInfo::fIsDesigner** != 0. Reserved on VB5.

Value	Name	Description
0x2	Designer	A Visual Basic Designer for an Add-In
0x10	Class Module	A Visual Basic Class
0x20	User Control	A Visual Basic Active X User Control (OCX)
0x80	User Document	A Visual Basic User Document

Table 2. fObjectType values

The Designer Info::cbStructSize – size of some fields of **The Designer Info** (look at Figure 12 for details).

You can notice that at Figure 12 information about **Addin Specific Data** is missing. This structure was not required for this research, so I only determined the conditions on which it gets filled.

Window that can be used to put some values into structure can be seen at Figure 10 (numbers are showing the order of windows to be opened).

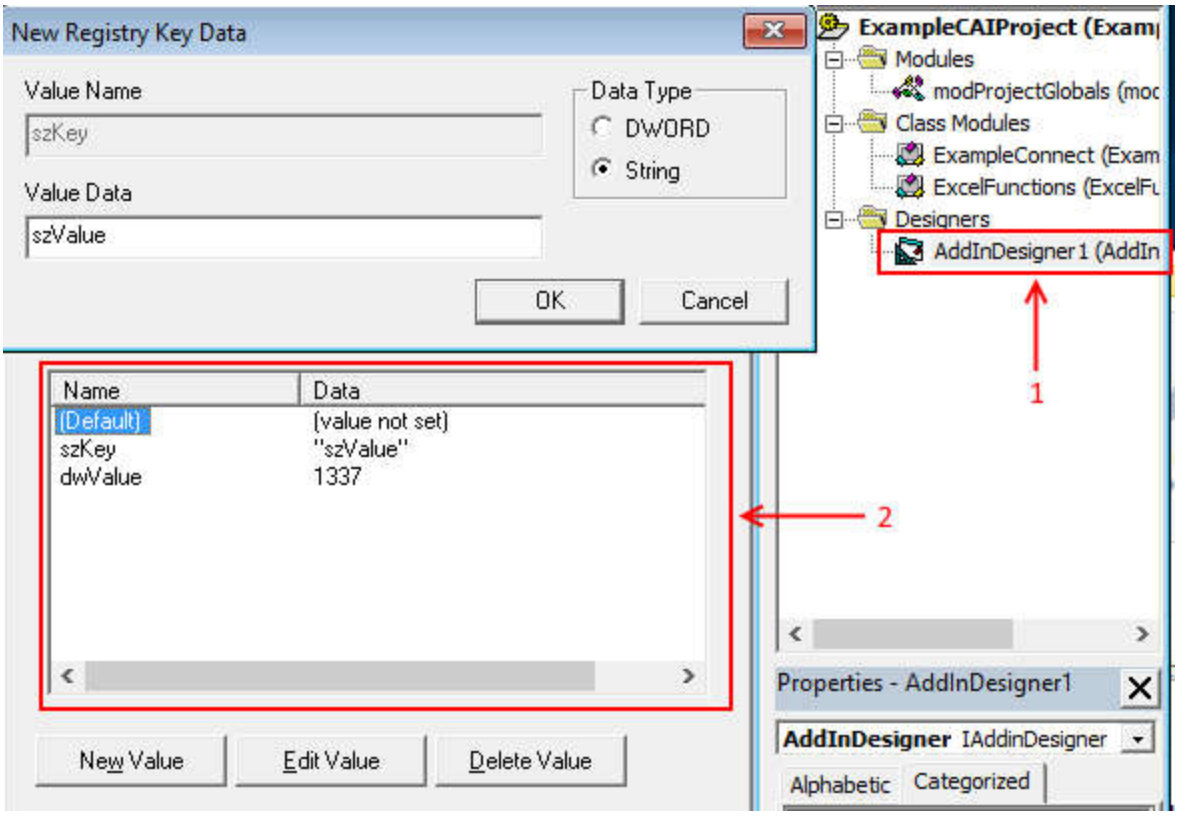


Figure 10. **Addin Specific Data** in GUI

After compilation contents of **Addin Specific Data** can be found in the way like at Figure 11.

```
00000100: 69 74 69 6F 6E 61 6C 2F  61 64 64 69 6E 2F 64 61  itional/addin/da
00000110: 74 61 00 01 00 00 00 01  00 00 00 01 00 00 00 01  ta
00000120: 00 00 00 06 00 00 00 73  7A 4B 65 79 00 08 00 00          szKey
00000130: 00 73 7A 56 61 6C 75 65  00 08 00 00 00 64 77 56   szValue      dwV
00000140: 61 6C 75 65 00 39 05 00  00 80 80 80 80 80 41 64  alue 9         Ad
00000150: 64 49 6E 44 65 73 69 67  6E 65 72 31 00 80 67 75  dInDesigner1  gu
```

Figure 11. **Addin Specific Data** in raw

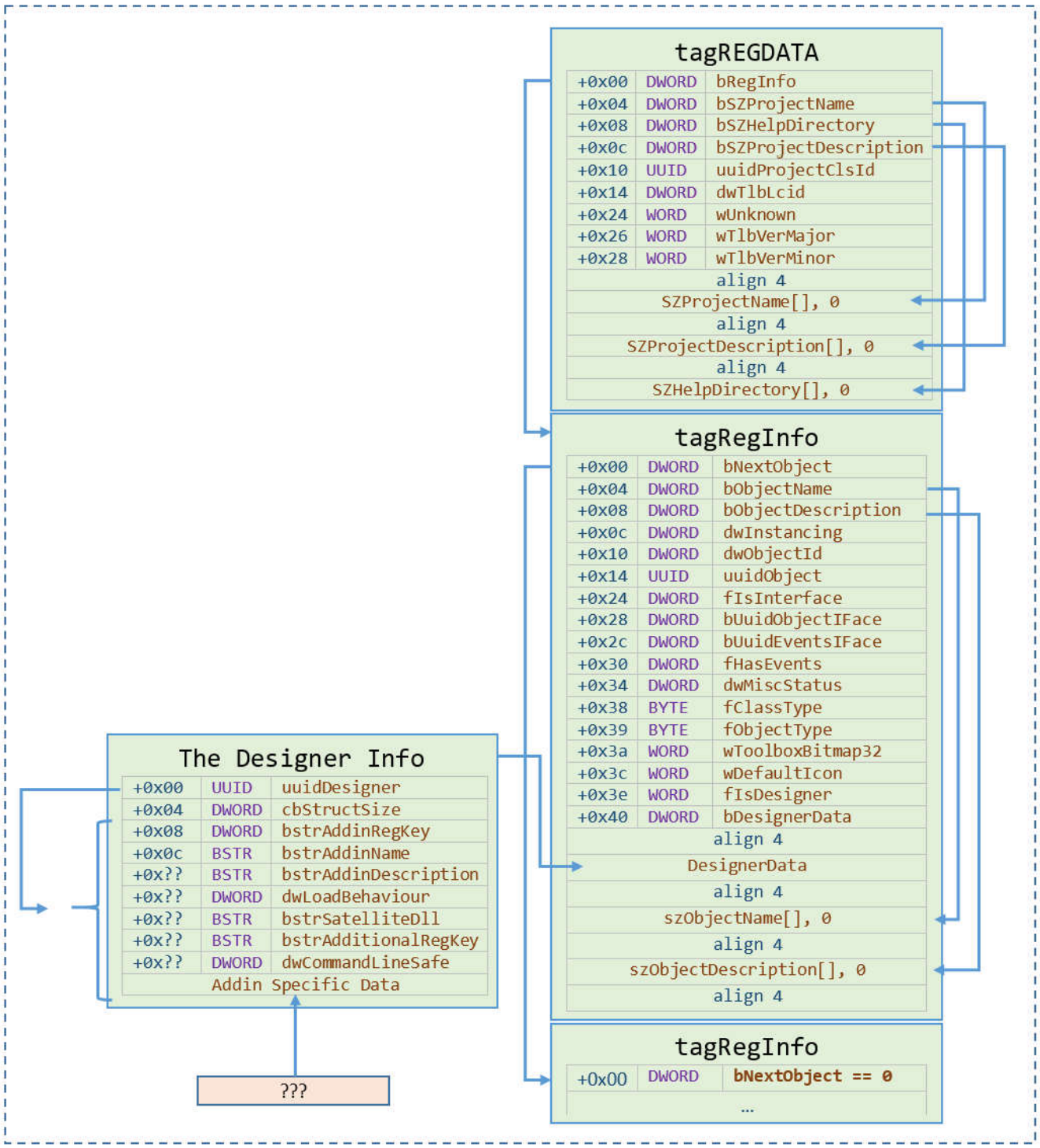


Figure 12. Schema

The Object Table

lpProjectInfo2 – pointer to **The Secondary Project Information**.

dwTotalObjects – count of objects in **lpObjectArray**.

dwCompiledObjects – value >= **dwTotalObjects**.

dwObjectsInUse – same value as **dwTotalObjects**.

lpObjectArray – pointer to array of **The Public Object Descriptor**. Depending on the source different size fields are used for this field. Most likely correct one is **dwTotalObjects**. Usage of **dwCompiledObjects** usually leads to errors.

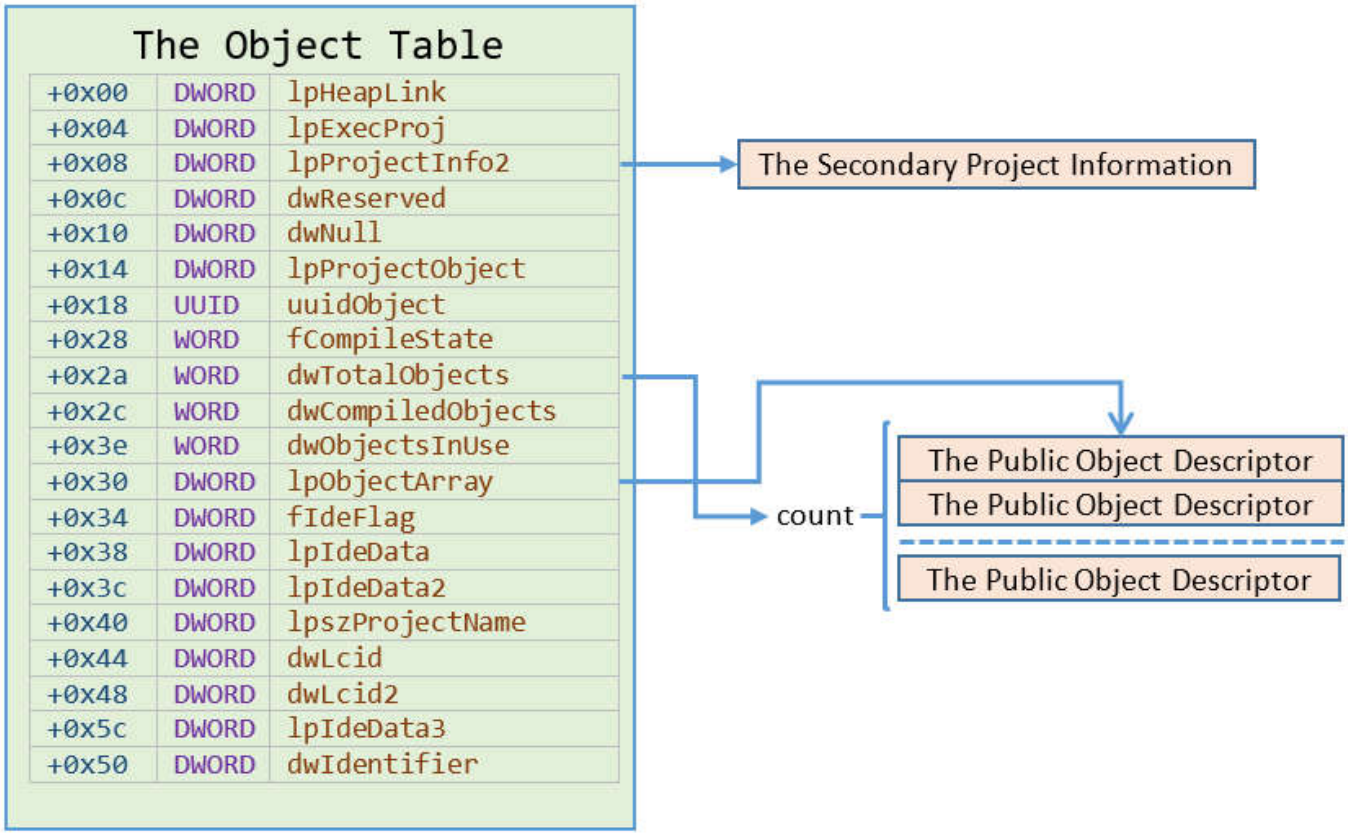


Figure 13. The Object Table

The Secondary Project Information

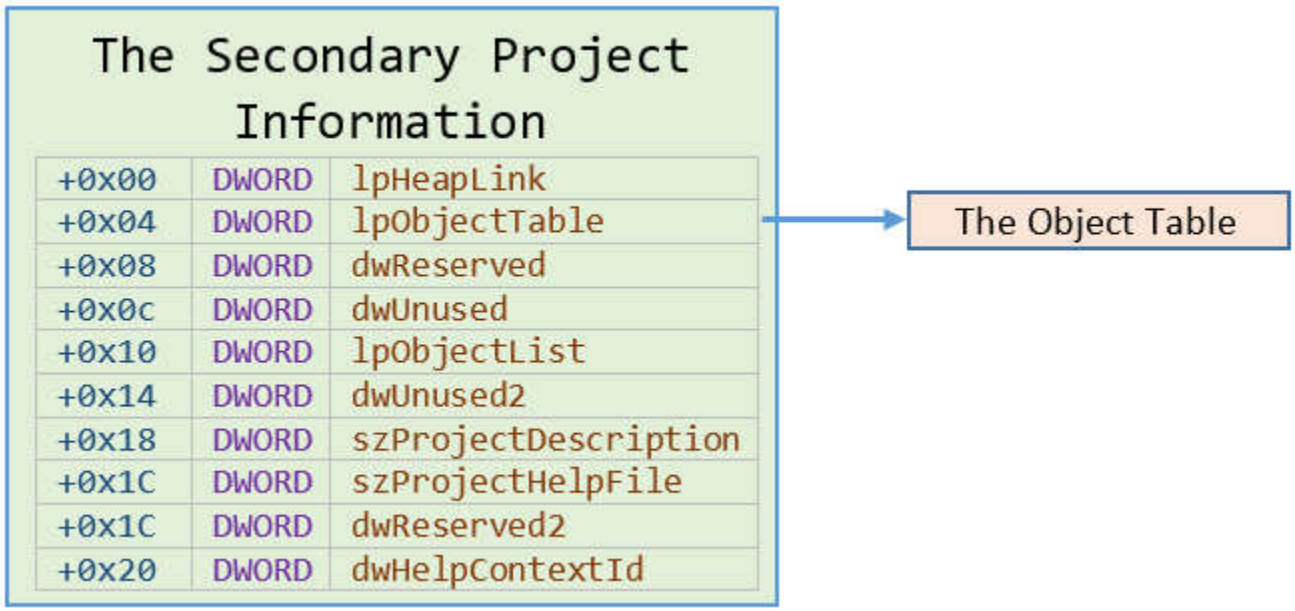


Figure 14. The Secondary Project Information

szProjectDescription and **szProjectHelpFile** are pointer to strings with self-explanatory contents. It is common to find this fields filled with unique information (different from the one at **EXEPROJECTINFO**).

The Public Object Descriptor

lpObjectInfo – pointer to **The Object Info**

lpszObjectName – pointer to C-string of the object name.

lpMethodNames – pointer to array of method names . Count is stored in **dwMethodCount**.

lpPublicBytes – pointer to **RESDESCTBL**.

lpStaticBytes – pointer to **RESDESCTBL** (different from previous one, but with the same format).

fObjectType – bit field, describes type of the object. For us it's important to check the bit 1 (**HasOptInfo**), if it is set we can find **The Optional Object Info** after **The Object Info**.

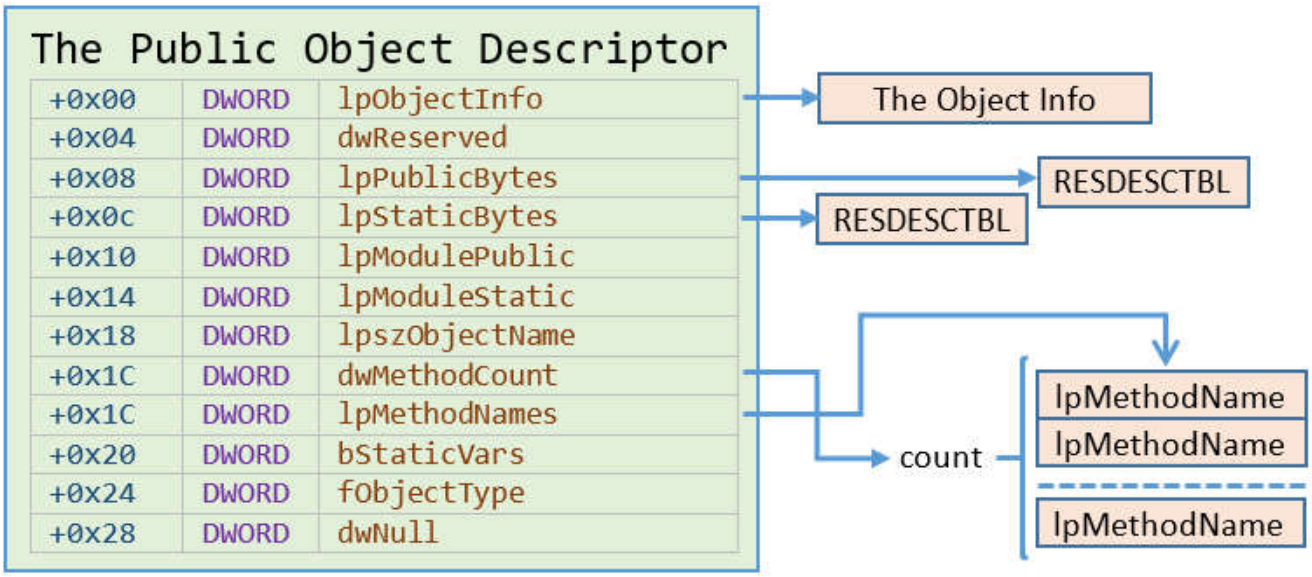


Figure 15. The Public Object Descriptor

```
VB_PublicObjectDescr.fObjectType Properties
###########################################################
form:               0000 0001 1000 0000 1000 0011 --> 18083
                    0000 0001 1000 0000 1010 0011 --> 180A3
                    0000 0001 1000 0000 1100 0011 --> 180C3
module:             0000 0001 1000 0000 0000 0001 --> 18001
                    0000 0001 1000 0000 0010 0001 --> 18021
class:              0001 0001 1000 0000 0000 0011 --> 118003
                    0001 0011 1000 0000 0000 0011 --> 138003
                    0000 0001 1000 0000 0010 0011 --> 18023
                    0000 0001 1000 1000 0000 0011 --> 18803
                    0001 0001 1000 1000 0000 0011 --> 118803
usercontrol:        0001 1101 1010 0000 0000 0011 --> 1DA003
                    0001 1101 1010 0000 0010 0011 --> 1DA023
                    0001 1101 1010 1000 0000 0011 --> 1DA803
propertypage:       0001 0101 1000 0000 0000 0011 --> 158003
                      | ||     |  |    | |    |
[moog]                | ||     |  |    | |    |
HasPublicInterface ---+ ||     |  |    | |    |
HasPublicEvents --------+|     |  |    | |    |
IsCreatable/Visible? ----+     |  |    | |    |
Same as "HasPublicEvents" -----+  |    | |    |
[aLfa]                         |  |    | |    |
usercontrol (1) ---------------+  |    | |    |
ocx/dll (1) ----------------------+    | |    |
form (1) -------------------------------+ |    |
vb5 (1) ----------------------------------+    |
HasOptInfo (1) --------------------------------+
                                               |
module(0) --------------------------------------+
```

Figure 16. The Public Object Descriptor :: fObjectType values from [2]

The Object Info

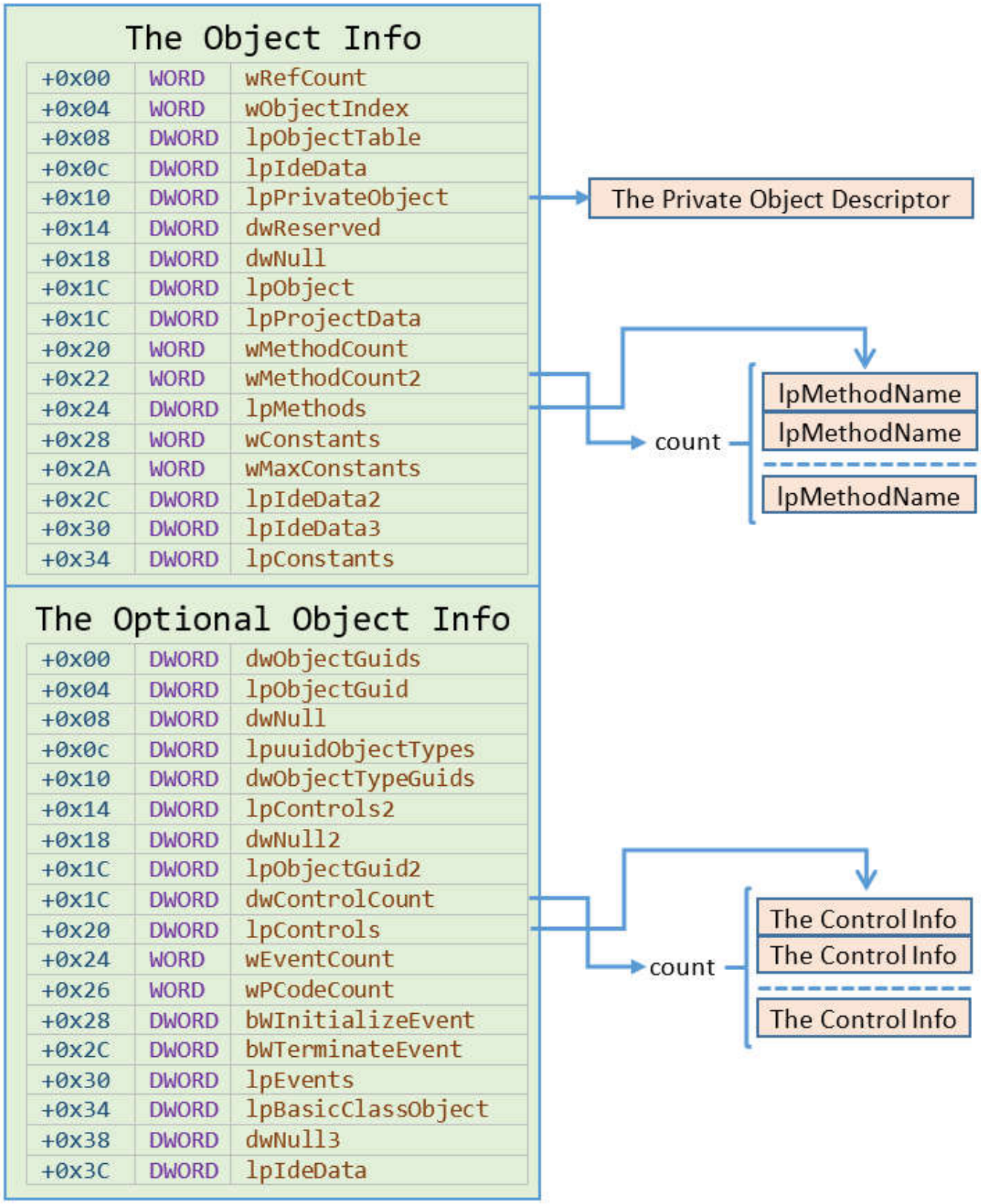


Figure 17. The Object Info

The Method Info

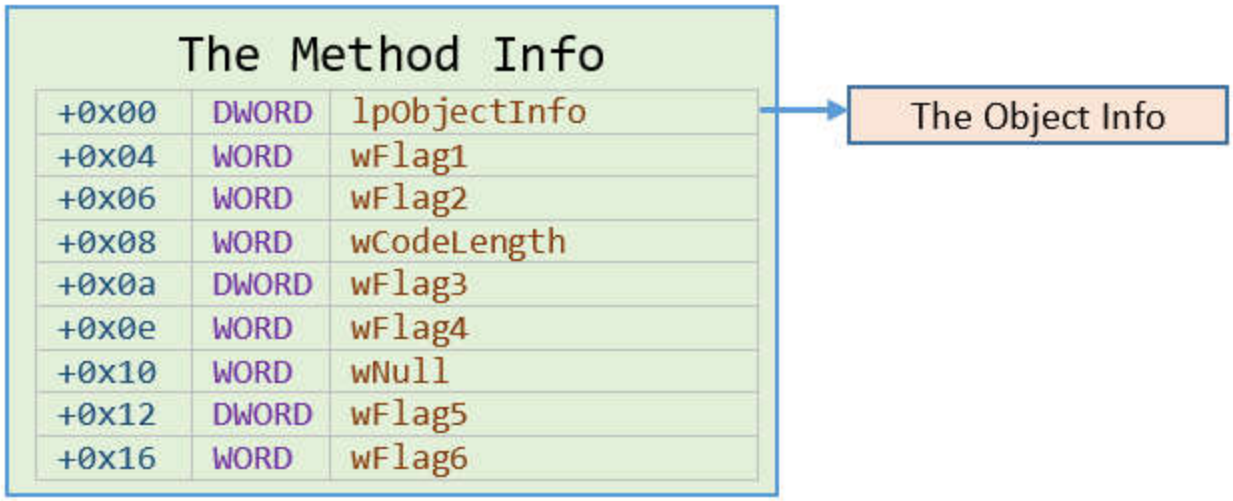


Figure 18. The Method Info

The Private Object Descriptor

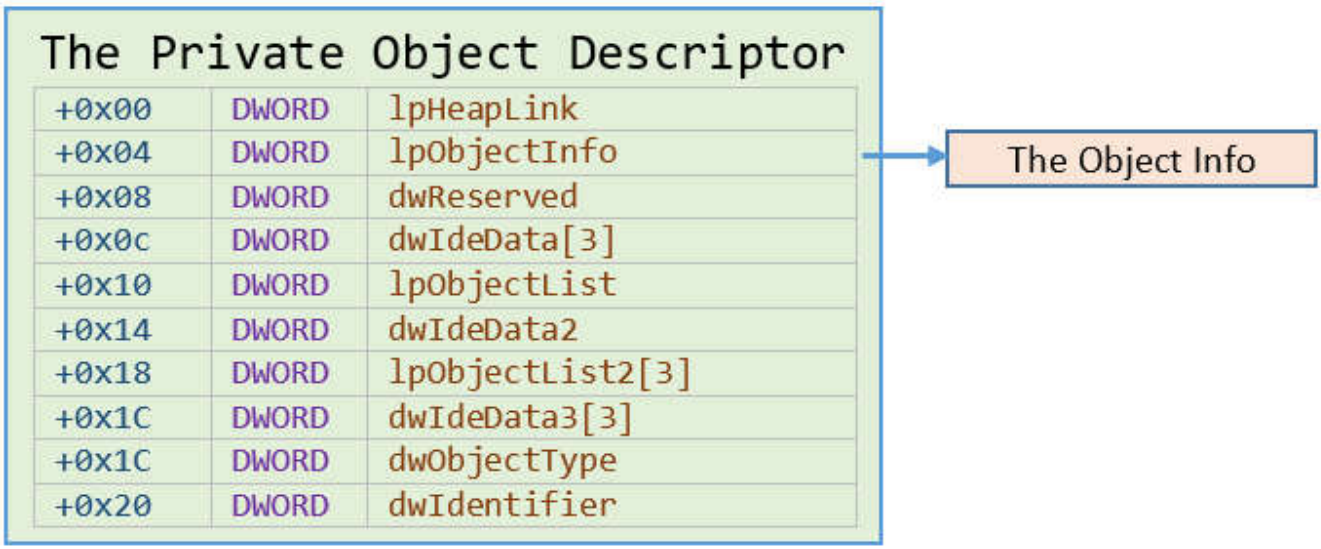


Figure 19. The Private Object Descriptor

The Control Info

lpGuid – pointer to GUID of the current control.

lpszName – name of the control.

lpEventTable – pointer to table of control's methods. Description can be found in IDC script for IDA [3], code is simple but quite big (a lot of different event names).

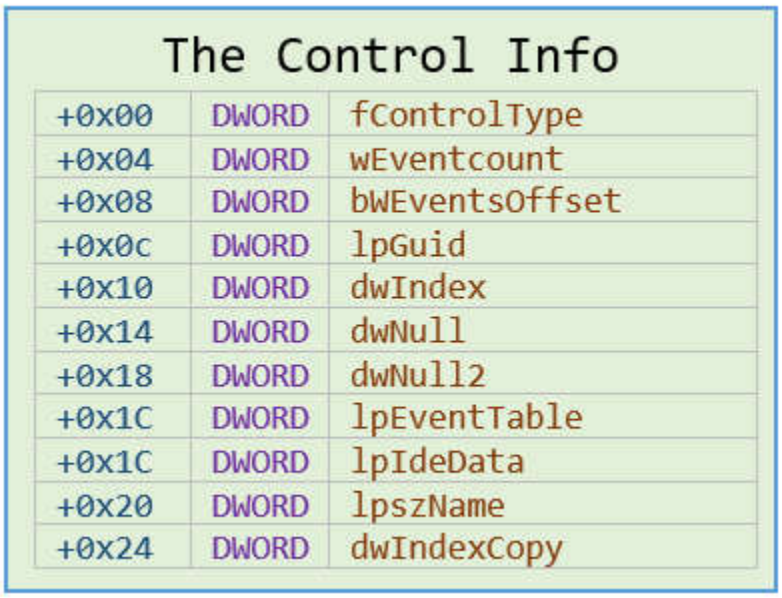


Figure 20. The Control Info

External API Descriptor

Despite the fact that VB5/6 files have no PE import entries besides runtime library, any program can freely use WinAPI functions it wants. This implemented using another internal structure.

As was already said before there are **External API Descriptor** structures present in the file. **dwType** of which can be one of two values:

- 6 – imported by GUID.
- 7 – import by library + function name (the most common one for PE files).

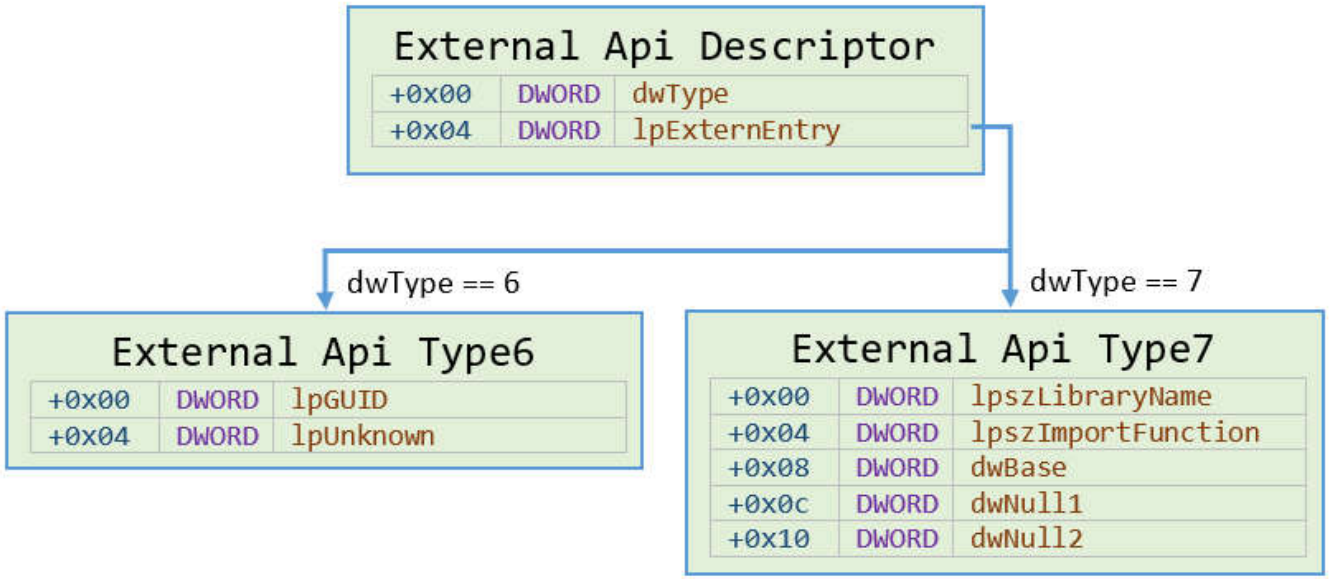


Figure 21. External API Descriptor

RESDESCTBL / RESDESC

Used for «packed» storage of different types of variables. **RESDESCTBL** is a header (fixed size), followed by multiple **RESDESC** (variable size).

In VBParser source code you can find a function which is able to calculate size of this structure. It was recovered from the runtime code.

By looking at this function we can tell that last 4 bits of field **wTypeFlags** identifies the type. Later we will be referring to this numbers as Type 5 and Type 9.

Total block size can be found at **RESDESCTBL::wTotalBytes**. Use it to check the correctness of parsing.

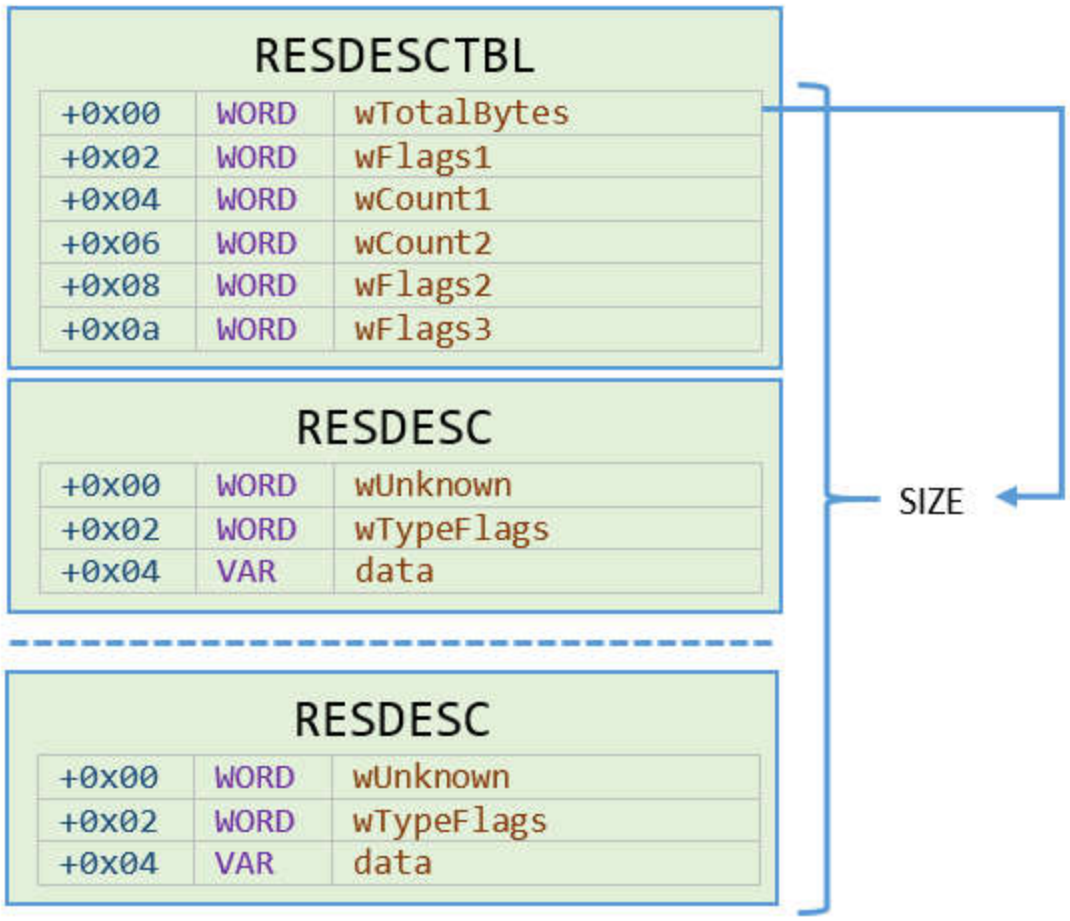


Figure 22. RESDESCTBL / RESDESC

File system paths inside binary

If you ever looked at VB files in HEX view you may notice that it sometimes contain path for files with *.olb, *.tlb, *.oca and other extensions.

These files contains information about external types used by the program.

- .ocx - ActiveX Control
- .oca - Extended type library/custom control cache file that goes along with a .ocx
- .tlb - COM interface definition.
- .olb - A Microsoft Object Library file that contains information referenced by Microsoft Office components.

We would probably want to detect all these paths without use of regular expressions (as text strings can have unexpected values, which we want to detect, too). As far as I know there are at least two structures that are pointing to this paths. But both have zero references leading to them (one of the fields – "ideData" is pointing to External Library Header, but this link is valid only during compilation).

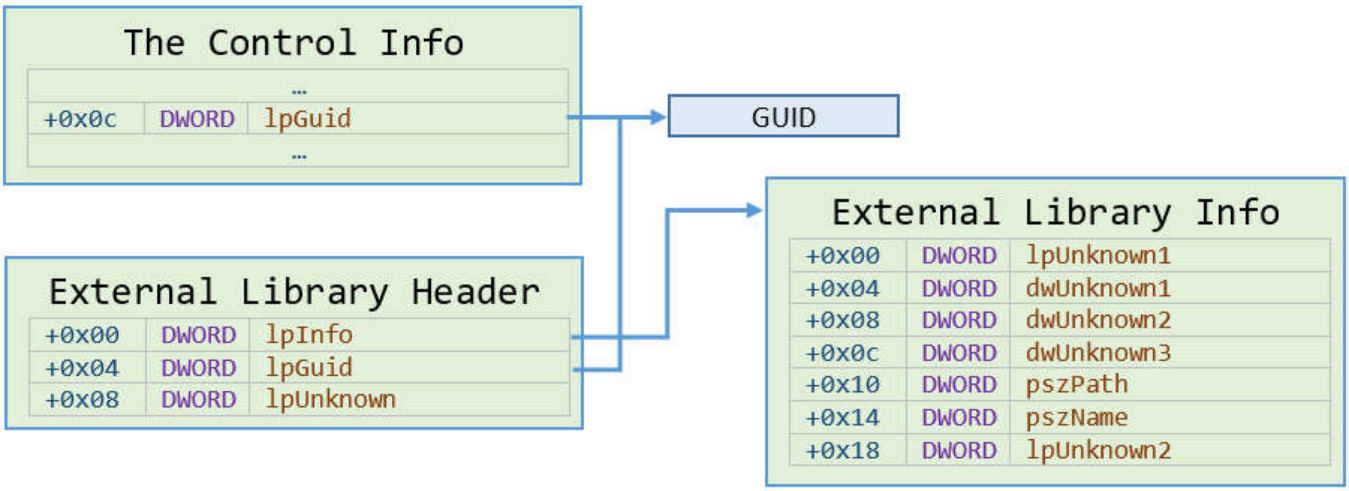


Figure 23. GUID references

It was noticed that this structure can be located indirectly. There is a GUID which is referenced by both desired structure and some other (**The Control Info**). This means that we could collect all GUIDs from the control description and then find all places in the program which have a reference for each of these GUIDs (results will contain among the others pointers from **External Library Header** structure).

Described method is working in most cases, but in general it can miss. GUID which is referenced by desired structure may not be present in the control list. In this case there will be links to it from the program code itself (as an argument for VB specific functions).

Vulnerability: Compiler memory leak inside executable files

While reversing VB5/6 programs we noticed that in some cases executable files contain parts of file system path. We all know that something similar is possible with informative assert strings or PDB paths. But in this case we were not able to tell what is it exactly. Lack of file format documentation and public researches forced us to dig into compiler.

Figure 24. One of the first discovered samples

First of all we downloaded a bunch of source codes and started to compile it one by one, checking every generated file for such anomaly. Quickly we got first results which revealed us that not only path can appear in file, but also environment variables and even source code.

It became obvious that such trash is part of the uninitialized memory and compiler is simply forgot to memset it. There were few possibilities of what memory is leaking (stack and different ways of heap allocation).

After some tests, a special library was created that hooks RtlAllocateHeap and fills all allocated memory with predictable values. I also made code to hook every function prolog (push ebp, mov ebp, esp) in suspected module to memset negative stack the same way.

Finally there were 5 different leaks found on heap. I will describe each one of them in the next sections. Stack leaks were not found.

RESDESC

As were said before this structure stores some packed values. Format is quite complicated, so instead of making a complete parser I made a few masks for each of possible cases to extract leaked memory.

Type 9. According to size calculation function, this structure occupy 4 (header) + 24 bytes. But usually only 6 bytes are used. Remaining 18 will most likely leak some memory.

```
HEADER
WORD wUnused1; // leak
WORD wUnknown; // usually 0xffff
DWORD lpSubResDscrTbl;
WORD wUnused2; // leak
WORD wUnused3; // leak
WORD wUnused4; // leak
WORD wUnused5; // leak
WORD wUnused6; // leak
```

```
WORD wUnused7; // leak
WORD wUnused8; // leak
WORD wUnused9; // leak
```

lpSubResDscrTbl – value is actually a pointer to another **RESDESCRTBL**. Recursion occurs, so it does make sense to go deeper searching for more leaks.

So we have the mask (1 letter = 1 byte, X – Leak, ? - Valid): **X X ? ? ? ? ? ? X X X X X X X X X X X X X X**

```
00000000: 00 00 FF FF C0 49 40 00  43 00 3A 00 5C 00 57 00      I@ C : \ W
00000010: 49 00 4E 00 58 00 50 00                          I N X P
```

Figure 25. Grayed out bytes are valid, other are suspected leak.

Type 5. Let's also require condition **wTypeFlags** == 0x0005. If satisfied, following structure can be used:

```
HEADER
WORD wUnused1; // leak
WORD wUnused2; // leak
RESDESCFLAGS wType2;
WORD wUnused3; // leak
WORD wUnused4; // leak
WORD wUnused5; // leak
BYTE SaBase1[16]; // SAFEARRAY
BYTE SaBase2[16]; // SAFEARRAY
```

Mask for structure with size < 38:

X X X X ? ? X X X X X X ? ? ? ? ? ? ? ? ? ? ? ...

Mask for structure with size >= 38:

X X X X ? ? X X X X X X ? X X

Notice: We found a file that contains unreferenced RESDESCTBL, which includes Type 9 leak. It's not possible to tell if this is a result of file modification or compiler can generate this by itself.

Pointers to this structure can be found in fields **The Public Object Descriptor::PublicBytes** and **The Public Object Descriptor::StaticBytes** for each of the programs objects (object in Visual Basic sense).

Figure 26. Type 5 leak

COM Objects

All COM objects descriptions were build in continuous memory block, which was allocated once, so all leaks should be counted as single one.

Most leaks are located on align bytes of structures and strings.

There are also some fields that is not filled depending on the object type.

Please refer to the source code in vbfile.cpp for details, as I left no detailed notes about detection of this type.

lpMethods

First of all determine whether the file was built as P-Code or Native.

If its Native, all method pointers are leaks.

For P-Code you need to check each of the pointers and see if address it points to present in the file. If yes, you need to check also if **lpObjectInfo** points to a parent (**The Object Info**).

Everything that is not valid on above conditions - leaks.

MethodNames

This buffer is allocated once, so it essential to determine its boundaries. To archive that we will crawl all the objects and record lowest and highest possible addresses of the array of pointers to object methods. After that we get the global minimum and maximum address.

Next step is to detect valid pointers, this could be done by checking if address exists in the binary. All other can be marked as leaks.

Resource Directory Major

This leak is 2 bytes at most, so it completely useless. Still, it deserves a mention

At the build time **Resource Directory** structure's field **Major** (WORD) will remain unset hence we have 2 bytes leak from original buffer.

Fun fact: this structure is duplicated many times (for each of **Resource Directory**). This means that leaked 2 bytes are the same for all such structures in file. This can be used to detect tampering.

Visual Basic 5/6 PE Format specifics

A small remark about specifics of VB5/6 files in the matter of PE file format.

As far as I can tell, VB5/6 is the only compiler that fills “Timestamp” field in **Resource Directory**. This can help to determine real compilation time, since “Timestamp” in **FileHeader** is the first thing to tamper with.

Starting unknown version, compiler includes **RICH Signature** in output files. Together with the field **wRuntimeBuild** of **EXEPROJINFO** it may help to detect file tampering and exact version of compiler environment used.

Results and Examples

As a result of research I created a GUI tool to extract interesting information from VB5/6 files. This includes leaks and some interesting fields.

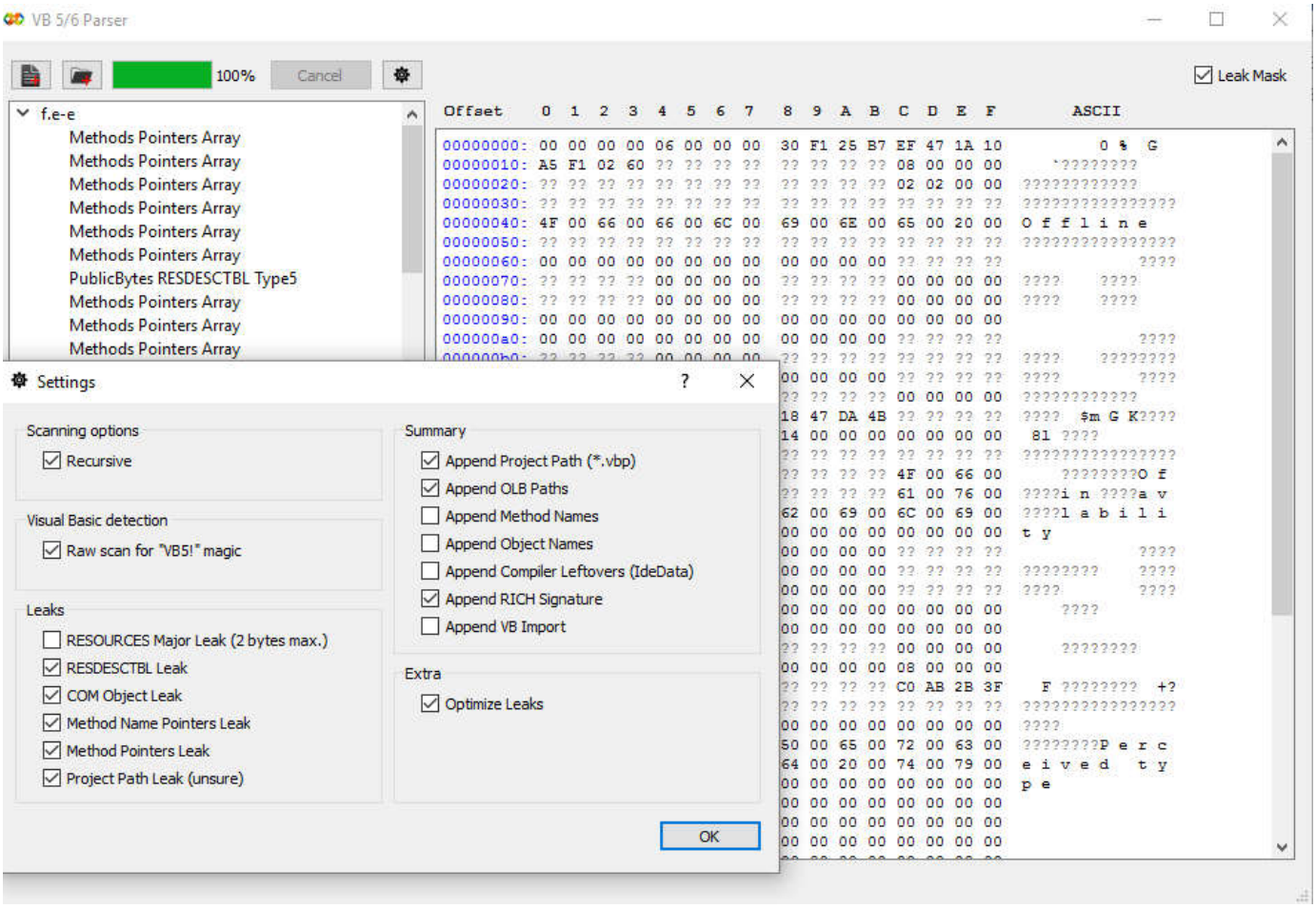


Figure 27. VBParser interface.

Let's look at examples I was able to find in the wild.

Figure 28 shows us username (Administrator) and computer name (WWWFOX-NET) of the machine used to build this sample.

==== Methods Pointers Array====

```
00000000: 5C 00 41 00 64 00 6D 00  69 00 6E 00 69 00 73 00  \ A d m i n i s
00000010: 74 00 72 00 61 00 74 00  6F 00 72 00 00 00 4C 00  t r a t o r   L
00000020: 4F 00 47 00 4F 00 4E 00  53 00 45 00 52 00 56 00  O G O N S E R V
00000030: 45 00 52 00 3D 00 5C 00  5C 00 57 00 57 00 57 00  E R = \ \ W W W
00000040: 46 00 4F 00 58 00 2D 00  4E 00 45 00 54 00 00 00  F O X - N E T
00000050: 4E 00 55 00 4D 00 42 00  45 00 52 00 5F 00 4F 00  N U M B E R _ O
00000060: 46 00 5F 00 50 00 52 00  4F 00 43 00 45 00 53 00  F _ P R O C E S
00000070: 53 00 4F 00 52 00 53 00  3D 00 31 00 00 00 4F 00  S O R S = 1   O
00000080: 53 00 3D 00 57 00 69 00  6E 00 64 00 6F 00 77 00  S = W i n d o w
00000090: 73 00 5F 00 4E 00 54 00  00 00 4F 00 73 00 32 00  s _ N T   O s 2
000000a0: 4C 00 69 00                                      L i
```

Figure 28. Environment variables leak

Sometimes when PATH environment variables leaks, we are able to see which programs were installed. For example Figure 29 shows that user had GnuPG and OpenVPN on his system.

```
00000000: 31 00 2E 00 30 00 5C 00  ?? ?? ?? ?? 3A 00 5C 00  1 . 0 \ ????: \
00000010: 50 00 72 00 6F 00 67 00  72 00 61 00 6D 00 20 00  P r o g r a m
00000020: ?? ?? ?? ?? ?? ?? ?? ??  ?? ?? ?? ?? 57 00 69 00  ????????????W i
00000030: 6E 00 64 00 6F 00 77 00  73 00 20 00 4C 00 69 00  n d o w s   L i
00000040: 76 00 65 00 5C 00 53 00  68 00 61 00 72 00 65 00  v e \ S h a r e
00000050: 64 00 3B 00 43 00 3A 00  5C 00 50 00 72 00 6F 00  d ; C : \ P r o
00000060: 67 00 72 00 61 00 6D 00  20 00 46 00 69 00 6C 00  g r a m   F i l
00000070: 65 00 73 00 5C 00 47 00  4E 00 55 00 5C 00 47 00  e s \ G N U \ G
00000080: 6E 00 75 00 50 00 47 00  5C 00 70 00 75 00 62 00  n u P G \ p u b
00000090: 3B 00 43 00 3A 00 5C 00  50 00 72 00 6F 00 67 00  ; C : \ P r o g
000000a0: 72 00 61 00 6D 00 20 00  46 00 69 00 6C 00 65 00  r a m   F i l e
000000b0: 73 00 5C 00 4F 00 70 00  65 00 6E 00 56 00 50 00  s \ O p e n V P
000000c0: 4E 00 5C 00 62 00 69 00  6E 00 00 00 07 10 00 17  N \ b i n
000000d0: DF 5B 0A 00 10 6A 0C 06  00 94 D6 02 00 00 00 00   [   j
000000e0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
000000f0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000100: 00 00 00 00 00 10 01 19  83 5B 0A 08 58 2A 0E 06           [  X*
00000110: E8 2C E1 02 0F 00 00 00  00 00 00 00 D7 E5 A7 46  ,              F
00000120: 43 1D 00 80 D8 45 0E 76  10 45 0E 76 98 50 0E 76  C    E v E v P v
00000130: 10 43 0E 76 D0 55 0E 76  58 55 0E 76 1C 55 0E 76   C v U vXU v U v
00000140: 78 50 0E 76 08 55 0E 76  68 50 0E 76 F4 54 0E 76  xP v U vhP v T v
00000150: 84 4B 0E 76 30 4D 0E 76  00 00 00 00 00 00 00 00   K v0M v
00000160: 00 00 00 00 80 51 0E 76  00 57 0E 76 00 00 00 00       Q v W v
00000170: 00 00 00 00 00 00 00 00  01 00 00 00 E4 54 0E 76                T v
00000180: E8 4A 0E 76 F8 52 0E 76  24 4B 0E 76 14 4B 0E 76   J v R v$K v K v
00000190: D8 4A 0E 76 C8 4F 0E 76  B8 4F 0E 76 70 54 0E 76   J v O v O vpT v
000001a0: EC 56 0E 76 00 00 00 00  00 00 00 00 F0 53 0E 76   V v          S v
000001b0: 42 08 00 00                                      B
```

Figure 29. PATH leaked

Next example gives us information about installed devices. PCI pair of “VendorID-DeviceID” corresponds to “USB host controller”.

```
==== Methods Pointers Array====

00000000: 12 00 26 00 E8 28 BE 77  00 00 00 00 00 00 00 00    &  ( w
00000010: 01 00 61 00 3F 01 00 00  00 02 00 00 00 00 00 00    a ?
00000020: 10 11 1C 00 2D 00 31 00  31 00 64 00 30 00 2D 00      - 1 1 d 0 -
00000030: 61 00 33 00 63 00 63 00  2D 00 30 00 30 00 61 00  a 3 c c - 0 0 a
00000040: 30 00 63 00 39 00 32 00  32 00 33 00 31 00 39 00  0 c 9 2 2 3 1 9
00000050: 36 00 7D 00 5C 00 74 00  6F 00 70 00 6F 00 6C 00  6 } \ t o p o l
00000060: 6F 00 67 00 79 00 00 00  00 00 00 00 00 00 00 00  o g y
00000070: 00 00 00 00 00 00 00 00  5C 00 5C 00 3F 00 5C 00            \ \ ? \
00000080: 70 00 63 00 69 00 23 00  76 00 65 00 6E 00 5F 00  p c i # v e n _
00000090: 38 00 30 00 38 00 36 00  26 00 64 00 65 00 76 00  8 0 8 6 & d e v
000000a0: 5F 00 32 00 34 00 63 00  35 00 26 00 73 00 75 00  _ 2 4 c 5 & s u
000000b0: 62 00 73 00 79 00 73 00  5F 00 30 00              b s y s _ 0
```

Figure 30. PCI device info leaked

Next few samples had system drive UUID leaked. Maybe it even can uniquely identify the computer used to build this file.

==== Methods Pointers Array====

```
00000000: 12 00 26 00 E8 28 BB 77  00 00 00 00 00 00 00 00    &  ( w
00000010: 01 00 36 00 87 01 00 00  00 02 00 00 00 00 00 00    6
00000020: 80 75 21 00 26 00 66 00  64 00 23 00 7B 00 36 00   u! & f d # { 6
00000030: 39 00 39 00 34 00 61 00  64 00 30 00 34 00 2D 00  9 9 4 a d 0 4 -
00000040: 39 00 33 00 65 00 66 00  2D 00 31 00 31 00 64 00  9 3 e f - 1 1 d
00000050: 30 00 2D 00 61 00 33 00  63 00 63 00 2D 00 30 00  0 - a 3 c c - 0
00000060: 30 00 61 00 30 00 63 00  39 00 32 00 32 00 33 00  0 a 0 c 9 2 2 3
00000070: 31 00 39 00 36 00 7D 00  5C 00 77 00 61 00 76 00  1 9 6 } \ w a v
00000080: 65 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  e
00000090: 00 00 00 00 5C 00 5C 00  3F 00 5C 00 72 00 6F 00      \ \ ? \ r o
000000a0: 6F 00 74 00 23 00 73 00  79 00 73 00 74 00 65 00  o t # s y s t e
000000b0: 6D 00 23 00 30 00 30 00  30 00 30 00              m # 0 0 0 0
```

Figure 31. System drive UUID leaked #1

==== Methods Pointers Array====

```
00000000: 12 00 26 00 E8 28 BE 77  00 00 00 00 00 00 00 00    &  ( w
00000010: 01 00 65 00 4A 01 00 00  00 02 00 00 00 00 00 00    e J
00000020: 48 02 1A 00 5C 00 5C 00  3F 00 5C 00 72 00 6F 00  H   \ \ ? \ r o
00000030: 6F 00 74 00 23 00 73 00  79 00 73 00 74 00 65 00  o t # s y s t e
00000040: 6D 00 23 00 30 00 30 00  30 00 30 00 23 00 7B 00  m # 0 0 0 0 # {
00000050: 36 00 39 00 39 00 34 00  61 00 64 00 30 00 34 00  6 9 9 4 a d 0 4
00000060: 2D 00 39 00 33 00 65 00  66 00 2D 00 31 00 31 00  - 9 3 e f - 1 1
00000070: 64 00 30 00 2D 00 61 00  33 00 63 00 63 00 2D 00  d 0 - a 3 c c -
00000080: 30 00 30 00 61 00 30 00  63 00 39 00 32 00 32 00  0 0 a 0 c 9 2 2
00000090: 33 00 31 00 39 00 36 00  7D 00 5C 00 7B 00 32 00  3 1 9 6 } \ { 2
000000a0: 66 00 34 00 31 00 32 00  61 00 62 00 35 00 2D 00  f 4 1 2 a b 5 -
000000b0: 65 00 64 00 33 00 61 00  2D 00 34 00              e d 3 a - 4
```

Figure 32. System drive UUID leaked #2

Another interesting one, looks like someone was building his project in the Virtual Machine and has File Sharing enabled.

==== Methods Pointers Array====

```
00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000010: 5C 00 5C 00 56 00 62 00  6F 00 78 00 73 00 76 00  \ \ V b o x s v
00000020: 72 00 5C 00 45 06 44 06  41 06 27 06 2A 06 00 00  r \ E D A ' *
```

Figure 33. Virtual Box path has leaked.

Sometimes we might be interested in user locale. It's common to get different paths leaks that contain non Latin characters ("Sık Kullanılan» = «Favorites» in Turkish).

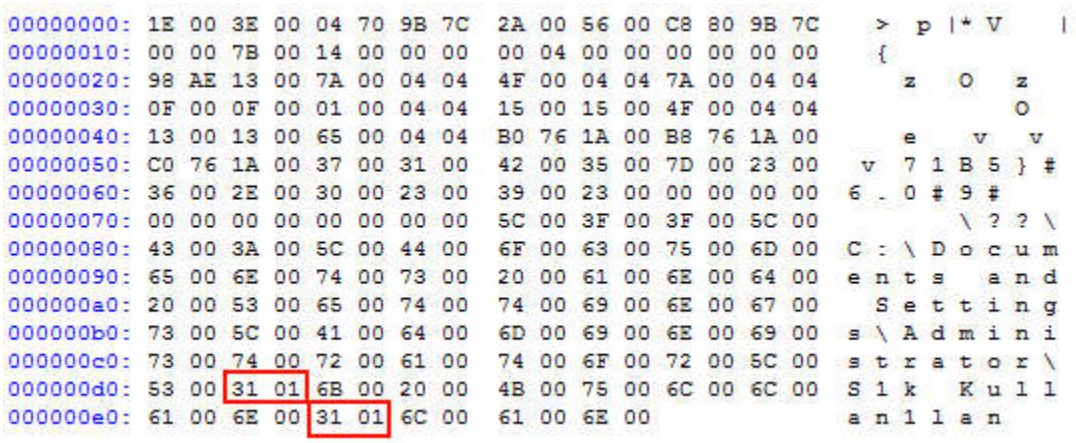


Figure 34. Locale specific path

And even parts of the source code can be found (maybe comments too?).

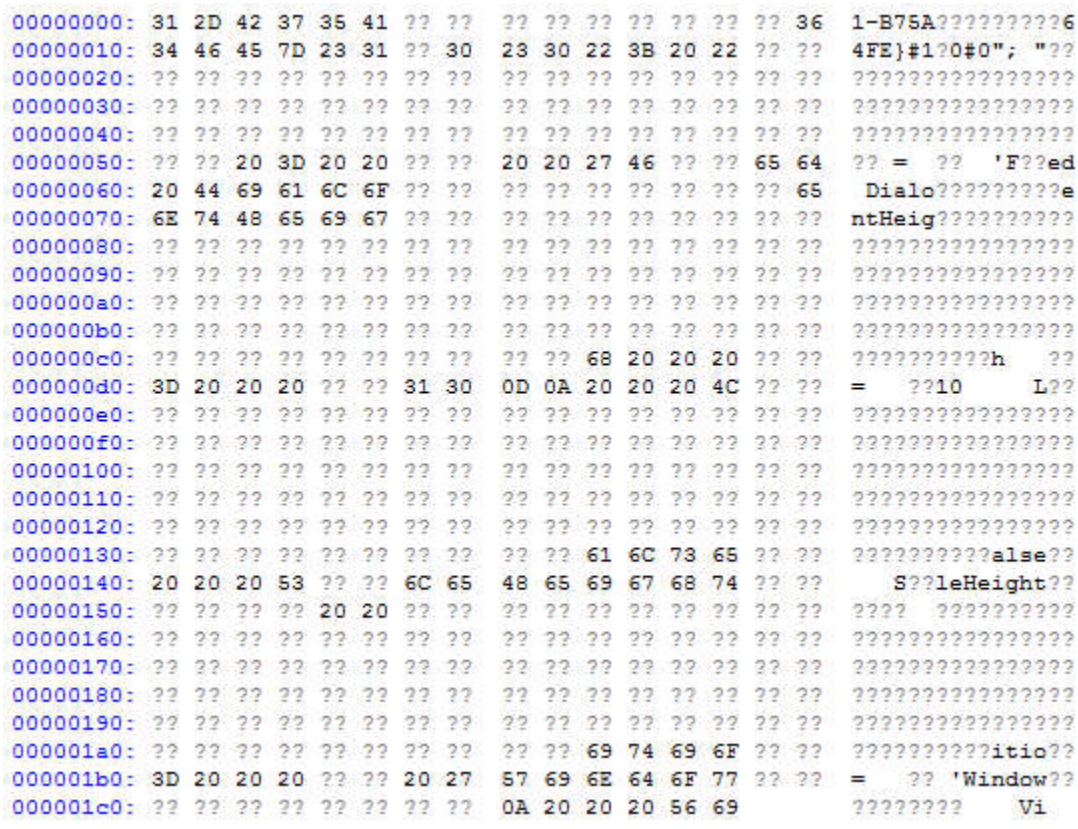


Figure 35. Form source code can be visible inside detected leak.

Final words

I hope this article will make you look again on your personal collection of VB5/6 malware and check if anything fun can be found in there. Share you findings!

Links

1. Alex Ionescu - Visual Basic Image Internal Structure Format (http://sandsprite.com/vb-reversing/files/Alex_Ionescu_vb_structures.pdf)
2. https://github.com/VBGAMER45/Semi-VB-Decompiler/
3. Vb Reversing Info and Links - http://sandsprite.com/vb-reversing/