# VISUAL BASIC REVERSED - A decompiling approach

Author: AndreaGeddon

**Abstract**

*Keywords: Reverse Code Engineering, Visual Basic, VB, Decompiling*

*Frameworks are getting more and more popular today, Visual Basic is one of them. Personally i hate frameworks, and also most reversers do. So, why this tutorial? We can consider both the light and the dark side of the problem: frameworks usually put a lot of code in the compiled programs, so it becomes hard to find the way among all that jungle. But they also use sets of pre-built objects, so theese objects are always the same and can be recognized, helping the reverser to understand the code itself. In a VB PE you have a lot of information inside the exe, so you can easily extract all the information you need about all components of the program. To analyze a VB application I used this program that was written by a friend of mine (thank you `_d31m0s_`!). It's a sort of name/serial crackme, but we are not interested in serial fishing, we are interested in how it works and how the vb knows how to build the app itself. I asked my friend to write it adding some event handling (colors, on over, etc) and a simple algorithm to check serial. He also wrote the proggy using more source files and making various subs (some null sub too). We also have the source of all, but we will check them later.*

*Let's make some introduction now!*

---

## 1 INTRODUCTION

Before VB5 the VB programs were not truly traduced in assembler, they were coded in Pseudo Code (hehe you all remember those hating pcode exe!), and the VB virtual machine had the task of interpreting the pseudo code and execute it. Those programs were linked to vbrun100, vbrun200, vbrun300, vbrun400 dlls (depending on the version); well thing are a little different because there were variations between 16bit or 32bit modules (pcode were mostrly 16bit apps), but this is not what we are looking for. Today we have version 5 and 6 of VB, they use MSVBVM50.dll and MSVBVM60.dll, and now VB exes are really compiled and traduced in asm. As you all know you can't use usual breakpoints like "GetWindowTextA" when debugging VB programs, infact you should use the apis exported from the VBVM dll (e.g. for a serial i would use `__vbaStrCmp`, or rtcMsgBox), if you want to use theese apis in softice you just have to load the VB dll (in winice.dat or via symbol loader). This will help you debug VB applications.

## 2 ANALISYS

Let's start disassembling the proggy. I am using Ida, i advice you to use it too, but you can use other disassemblers if you want. First of all lets have a general look, so we can have a general idea. You can easily see the following:

```
00401000

 ...       IAT (First Thunk ok apis)

004010F0

 ...       some data

004011A5

 ...       transfer area (declspec(dllimport) style)

0040130E

 ...       lots of data

004023EC

 ...       local transfer area (for internal event handlers)

0040242C

 ...       other data

00402E44

 ...       code

00403D06

 ...       other data
```

Ok we have a general idea of the mapping of the program. Notice that all read only data is in the .text section, the data before and after the code contains names of imported functions (IT original first thunks), however now we begin analisys.

We start from entry point. What we see is:

```
00401310   push  offset  RT_MainStruct
00401315   call  ThunRTMain
```

if you debug this, you execute the call and the program runs. The analisys here is very simple: the entry point simply consists of:

```
ThunRTMain(&RT_MainStruct);
```

this function is the main function of the VBVM (the Thunder Runtime Engine) and has one parameter, the pointer to a complex structure that describes all the application. It is clear that all the data, all the callbacks and so on are described by this structure, so we can extract a lot of information. We just have to check this big structure! Let's go!

```
00401970   RT_MainStruct db 'VB5!'   ;signature

00401974                 db 1Ch ;
00401975   aVb6it_dll    db '#VB6IT.DLL',0  ;italian language
00401980                 dd 0
00401984                 dd 2Ah
00401988                 dd 0
0040198C                 dd 0
00401990                 dd 0A0000h
00401994                 dd 410h
00401998                 dd 409h
0040199C                 dd 0
004019A0                 dd offset  ProjectStruct
004019A4                 dd 30F016h
004019A8                 dd 0FFFFFF00h
004019AC                 dd 8
004019B0                 dd 1
004019B4                 dd 2
004019B8                 dd 0E9h
004019BC                 dd offset  DialogsStruct
004019C0                 dd offset  ExternStruct
004019C4                 dd offset  ProjectInfo
004019C8                 dd 78h
004019CC                 dd 7Eh
004019D0                 dd 84h
004019D4                 dd 85h
004019D8                 dd 0
004019DC                 dd 0
004019E0                 dd 0
004019E4                 dd 0
```

the first field is the signature of the struct itself. Note that the program has been written with VB 6.0, but the signature is "VB5!", probably they didnt change it for cross compatibility? Or just forgot it? Who knows! However we can see various infos, the "VB6IT.dll" that should be the module to load the language (the app has been written with italian version of vb). What should really abtract your attention are the four pointers (ProjectStruct, DialogsStruct, ExternStruct, ProjectInfo). I gave them a name because i know their function, you should see just four addresse in the dasm. However we will analyse theese structs to find infos we need.

```
0040131C  ProjectInfo dd 0 ; DATA XREF: 004019C4o
00401320              dd 30h
00401324              dd 40h
00401328              dd 0
0040132C              dd 0EACF9A13h
00401330              dd 4F93898Bh
00401334              dd 0DF0C5493h
00401338              dd 159AAEEDh
0040133C              dd 0
00401340              dd 10000h
00401344 a020430progetto db 0,0,'020430Progetto1',0
00401356 aC000        db '-C000-'
0040135C              dd 0 ;
```

here we can see various numbers, in particular there is the project name as last field. If we go and search in the source of the app, we can see in the .vbp the following line

```
Reference=*\G{00020430-0000-0000-C000-000000000046}#2.0#0#D:\WINNT\System32\STDOLE2.TLB#OLE Automation
```

so we can think all the data bytes represent the type of project, modules contained etc etc. This isn't very useful, so we go on. We go back to main struct and see another sub structure:

```
004018F8  ExternStruct dd 6   ;flag

004018FC               dd offset InsideImport   ;import data
```

it is a bit harder to figure out what this structure is. This handles other imported functions, they can be inside or outside virtual machine module. The flag indicates the type of import (6 = inside, 7 = outside). If you look at InsideImport you will find:

```
00402B64  InsideImport dd offset  Descr
00402B68               dd offset  Thunk
```

Descr will point to four dwords that seem to be the same in all vb apps. Thunk will contain a pointer to an area where addresses (of some code of the virtual machine) are stored. I analysed an other vb app, its ExternStruct is:

```
//snippet from another vb app

00401A5C  ExternStruct dd 7  ;outside
00401A60               dd offset  ImportData
00401A64               dd 6  ;inside
00401A68               dd offset  InsideImport
//end snippet from another vb app
```

as you can see in that app there was an external function, infact the app used ShellExecute to open the internet browser and link to a site. The ImportData is as follows:

```
//snippet from another vb app

 00402DA4  ImportData dd offset  aShell32  ;modulename
 00402DA8             dd offset  aShellexecutea  ;apiname
 00402DAC             align 8
 00402DB0             dd offset  IAT_Data
//end snippet from another vb app
```

where you can see you have the pointers to the name of the module and the name of the api to import. The `IAT_Data` fields points to thunking data (hInstance of module and api address). This data is also used from the primitive `DLL_Import` of VB, used to thunk to outside apis. Back to our app, we check the DialogsStruct pointer:

```
 00401A00  DialogsStruct[0] dd 50h        ;sizeof struct

 00401A04                   dd 356022C6h
 00401A08                   dd 400E3F28h
 00401A0C                   dd 495240B8h
 00401A10                   dd 0C9491BB6h
 00401A14                   dd 0
 00401A18                   dd 0
 00401A1C                   dd 0
 00401A20                   dd 0
 00401A24                   dd 0
 00401A28                   dd 310h
 00401A2C                   dd 0
 00401A30                   dd 0
 00401A34                   dd 0
 00401A38                   dd 0
 00401A3C                   dd 0
 00401A40                   dd 596h
 00401A44                   dd 0
 00401A48                   dd offset  MainDialog
 00401A4C                   dd 4Ch
 00401A50  DialogsStruct[1] dd 50h         ;sizeofstruct
 00401A54                   dd 78CBBB9Fh
 00401A58                   dd 401EB563h
 00401A5C                   dd 0DB80B296h
 00401A60                   dd 7EFFD31Ah
 00401A64                   dd 0
 00401A68                   dd 0
 00401A6C                   dd 0
 00401A70                   dd 0
 00401A74                   dd 3
 00401A78                   dd 100h
 00401A7C                   dd 0
 00401A80                   dd 0
 00401A84                   dd 0
 00401A88                   dd 0
```

```
00401A8C                dd 0
00401A90                dd 1A3h
00401A94                dd 0
00401A98                dd offset  AboutDialog
00401A9C                dd 9Ch
```

we have an array of dialog descriptors. Each descriptor has various data values, in particular has in the 19th field a pointer to a struct were the resource info are stored. Theese structure have variable size, because it depends on the data contained by the resources. Lets go and see MainDialog; i will not paste all the data but only the important things.

```
004013C2  aForm1          db 5,0,'Form1',0

 004013CC  aLeimcrackme    db 0Bh,0,'Leimcrackme',0



 004013E2  IconData        db 6,3,0,0,6Ch,74h,0,0,0FEh...

 004016FA                 dw 2Dh   ; clientleft
  004016FE                 dw 14Ah  ; clienttop
  00401702                 dw 1248h ; clientwidth
  00401706                 dw 0B7Ch ; clientheight


  00401715  aText2          db 5,0,'Text2',0

 00401731  aText1          db 5,0,'Text1',0

 0040174D  aCommand2       db 8,0,'Command2',0
  0040175A  aCheck          db 7,0,'&Check!',0

 00401777  aCommand1       db 8,0,'Command1',0
  00401784  aAbout          db 6,0,'&About',0
  004017A2  aLabel2         db 6,0,'Label2',0
 004017AD  aSerial         db 7,0,'Serial:',0
  004017D1  aLabel2_0       db 6,0,'Label2',0
 004017DC  aName           db 5,0,'Name:',0
  004017FC  aLabel1         db 6,0,'Label1',0
  00401807  aWhoeverTriesTh db '',0,'Whoever tries...
```

you can see all the components of the crackme, their data, etc etc. IconData is the raw data of the icon of the main dialog.

If we look at the source we have:

```
Begin VB.Form Form1
  BorderStyle = 3 'Fixed Dialog
  Caption = "Leimcrackme"
  ClientHeight = 2940
  ClientLeft = 45
  ClientTop = 330
  ClientWidth = 4680
  Icon = "andre.frx":0000
  LinkTopic = "Form1"
```

you can see the icon data is encoded in the .frx file, which usually have big data. So the Icon field links to andre.frx file and 0000 is the offset of the starting data. Infact also the last label is linked as follows:

```
  Caption = $"andre.frx":030A
```

infact in the frx file after the raw icon data there is at offset 30Ah the string of that label. Of course in the compiled app all the infos (.frm and .frx) are built in the dialog sutrcture. In the same way you can see the infos about the second dialog (AboutDialog). Now we go and check the most important structure:

```
 00401AA0  ProjectStruct dd 1F4h    ;signature?

 00401AA4                dd offset  Tree
  00401AA8                dd 0
  00401AAC                dd offset  StartOfCode
  00401AB0                dd offset  EdnOfCode
  00401AB4                dd 1238h
  00401AB8                dd offset  DataVar1
  00401ABC                dd offset  vbaExceptHandler
  00401AC0                dd offset  StartOfData
  00401AC4                dd  84h dup(0)
  00401CD4                dd offset  ExternStruct
  00401CD8                dd 1
```

Let's see the fields: there is a pointer to the exetern imports, we already covered this structure. There are StartOfCode and EndOfCode vars, they indicate where the executable code starts and where it ends. The code is delimited by two signatures (E9E9E9E9h starting, 9E9E9E9E ending) and some 0xCC padding. There is also a field that is the pointer to the base per-thread exception handler, that is `__vbaExceptHandler` (but of course the code will install others handerls). There are the pointers to StartOfData and another pointer that points to StartOfData + 8, it seems theese values are common for all applications. In the 84h dup(0) space you can probably find infos about path of the project etc, usually you see here some unicode strings, nothing interesting. The remaining field, Tree, is a descriptor of code modules as they were in the source (and as they are oranized in the compiled exe).

So we are going to check this struct; it is a bit complex, so pay attention:

```
00402434  Tree dd 0

00402438      dd offset  VB_Func
 0040243C      dd offset  TreeData
 00402440      dd 0FFFFFFFFh
 00402444      dd 0
 00402448      dd offset  UnkVar1
 0040244C      dd 1F1CB8D4h
 00402450      dd 42793AE6h
 00402454      dd 51A97A97h
 00402458      dd 41E1033h
 0040245C      dd 4000Ah
 00402460      dd 40004h
 00402464      dd offset  ModulesList  ;ptr to ModulesList[0]
 00402468      dd 0
 0040246C      dd 0
 00402470      dd 0
 00402474      dd offset  aProgetto1_0 ; "Progetto1"
 00402478      dd 409h
 0040247C      dd 410h
 00402480      dd 0
 00402484      dd 2
```

`VB_Func` is a pointer to a location that is filled at runtime with some address of the vbvm, there are other fields such as the project name and an unknown var, but the interesting things are TreeData and ModuleList. Let's see TreeData:

```
00402DE0  TreeData dd 0
00402DE4          dd offset  Tree   ;back pointer
00402DE8          dd 0FFFFFFFFh
00402DEC          dd 0
00402DF0          dd offset  FormList
00402DF4          dd 0
00402DF8          dd 0
00402DFC          dd 0
00402E00          dd 0FFFFFFFFh
00402E04          dd 0
00402E08          dd offset  ProjectInfo2
00402E0C          dd offset  RawData1
00402E10          dd offset  R_UnkVar1
00402E14          dd offset  ProjectInfo2
00402E18          dd offset  RawData2
00402E1C          dd offset  R_UnkVar2
00402E20          dd offset  ProjectInfo2
00402E24          dd offset  RawData3
00402E28          dd offset  R_UnkVar3
00402E2C          dd offset  ProjectInfo2
00402E30          dd offset  RawData4
00402E34          dd offset  R_UnkVar4
```

```
00402E38          dd 0
00402E3C          dd offset  RawData5
00402E40          dd offset  R_UnkVar5
```

what we got here? A back pointer to tree struct, some raw data and ather infos about the project (you can see at projectInfo2 some data and usually you will find the string 'C:\Programmi\Microsoft Visual Studio\VB98\VB6.OLB'), the important pointer is FormList. Let's see:

```
00402D18  FormList dd offset Form[0]  ;form1
00402D1C          dd 0FFFFFFFFh      ;module1
00402D20          dd 0FFFFFFFFh      ;module2
00402D24          dd offset Form[1]  ;form2
infact if we see the .vbw file we find:
```

```
Form1 = 186, 224, 986, 702, , 207, 76, 652, 524, C
Module1 = 236, 214, 1036, 662,
Module2 = 166, 196, 966, 644,
Form2 = 100, 283, 900, 731, , 173, 118, 973, 566, C
```

so the FormList simply points to a list of tied dialogs to the forms/modules of the project. Note that modules correspond to .bas files, forms to .frm files (which include .frx for raw resources). Let's go on:

```
00402D60 Form[0] dd 0

00402D64         dd offset FormDescriptor[0]
00402D68         dd 0FFFFFFFFh
00402D6C         dd 0
00402D70         dd 0
00402D74         dd 0
00402D78         dd offset  FlagList
00402D7C         dd 0
00402D80         dd offset  UnkData
00402D84         dd offset  UnkData
00402D88         dd offset  UnkData
00402D8C         dd 0
00402D90         dd 0
00402D94         dd 0
00402D98         dd 58h
00402D9C         dd 4
```

i will not paste Form[1] because its identical to Form[0]. We see some pointers to some data (mostly they point to null values), and then a pointer to FormDescriptor. Here we stop for now, we have to go back to check Tree.ModulesList structure (from there we will reach again FormDescriptor structs):

```
00402488  ModulesList[0] dd offset FormDescriptor[0]
 0040248C                dd 0FFFFFFFFh
 00402490                dd offset  Flags_0
 00402494                dd 0
 00402498                dd 0
 0040249C                dd 0
 004024A0                dd offset  aForm1_0 ; "Form1"
 004024A4                dd 5
 004024A8                dd offset  OptionalData1
 004024AC                dd 0FFFFh
 004024B0                dd 18083h          ;Flags_1
 004024B4                dd 0
 004024B8 ModulesList[1]  dd offset ModuleDescriptor[0]
 004024BC                dd 0FFFFFFFFh
 004024C0                dd offset  Flags_1
 004024C4                dd 0
 004024C8                dd offset unk_modvar1
 004024CC                dd 0
 004024D0                dd offset  aModule1 ; "Module1"
 004024D4                dd 7
 004024D8                dd 0
 004024DC                dd 0FFFFh
 004024E0                dd 18001h
 004024E4                dd 0
 004024E8 ModulesList[2] dd offset ModuleDescriptor[1]
 004024EC                dd 0FFFFFFFFh
 004024F0                dd offset  Flags_1
 004024F4                dd 0
 004024F8                dd offset unk_modvar2
 004024FC                dd 0
 00402500                dd offset  aModule2 ; "Module2"
 00402504                dd 3
 00402508                dd 0
 0040250C                dd 0FFFFh
 00402510                dd 18001h
 00402514                dd 0
 00402518 ModulesList[3] dd offset FormDescriptor[1]
 0040251C                dd 0FFFFFFFFh
 00402520                dd offset  Flags_2
 00402524                dd 0
 00402528                dd 0
 0040252C                dd 0
 00402530                dd offset  aForm2 ; "Form2"
 00402534                dd 2
 00402538                dd offset OptionalData2
 0040253C                dd 0FFFFh
 00402540                dd 18083h
 00402544                dd 0
```

voil. We can easily see names and order of the forms/modules of the project. Note that the first field is a descriptor for the Module/Form, they are different descriptors, so they have a different structure. We can go and see ModuleDescriptors:

```
00401938 ModuleDescriptor[0] dd 10001h
0040193C                     dd offset  Tree  ;back pointer
00401940                     dd 0
00401944                     dd 0FFFFFFFFh
00401948                     dd 0FFFFFFFFh
0040194C                     dd 0
00401950                     dd offset ModulesList[1]  ;back pointer
00401954                     dd offset  MD0_UnkVar
00401958                     dd 0
0040195C                     dd 7D63150h
00401960                     dd 0
00401964                     dd 0
00401968                     dd 0
0040196C                     dd offset  RT_MainStruct ; ptr to following address
```

nothing of interest here. The other module descriptor is like this so i won't paste it. Let's see form descriptors:

```
00401F48 FormDescriptor[0] dd 1
00401F4C                   dd offset  Tree      ;back pointer
00401F50                   dd 0
00401F54                   dd offset Form[0]   ;back pointer
00401F58                   dd 0FFFFFFFFh
00401F5C                   dd 0
00401F60                   dd offset  ModulesList  ;back pointer
00401F64                   dd offset  DataVar1
00401F68                   dd 0
00401F6C                   dd 7D98E18h
00401F70                   dd 0
00401F74                   dd 0
00401F78                   dd 0
00401F7C                   dd offset  FD0_Raw1
00401F80                   dd 1
00401F84                   dd offset  FD0_Raw2
00401F88                   dd 0
00401F8C                   dd offset  FD0_Raw1
00401F90                   dd 1
00401F94                   dd offset  FD0_ControlsList
00401F98                   dd 0
00401F9C                   dd offset  FD_Raw3
00401FA0                   dd 7            ;number of controls in list
00401FA4                   dd offset  FD0_ControlsList
00401FA8                   dd 1B70005h
00401FAC                   dd 6C0068h
00401FB0                   dd offset  FD0_Dispatcher
00401FB4                   dd offset  FD0_UnkVar1
00401FB8                   dd 0
00401FBC                   dd 1324FCh
```

we see some unknown vars and some back pointers, what we care of is `FD0_COntrolsList` and `FD0_Dispatcher`. We see there are 7 controls in the list, so let's see the list:

```
00401FC8  FD0_ControlsList[0] dd 180040h    ;control type
00401FCC                      dd 34h        ;ID1
00401FD0                      dd offset  RawData1
00401FD4                      dd 30005h     ;ID2
00401FD8                      dd 0
00401FDC                      dd 0
00401FE0                      dd offset LocalDispatcher[0]
00401FE4                      dd 7DC1BF0h
00401FE8                      dd offset  aText2_0 ; "Text2"
00401FEC                      dd 30005h
00401FF0  FD0_ControlsList[1] dd 180040h    ;control type
00401FF4                      dd 38h        ;ID1
00401FF8                      dd offset RawData1
00401FFC                      dd 30004h     ;ID2
00402000                      dd 0
00402004                      dd 0
00402008                      dd offset LocalDispatcher[1]
0040200C                      dd 7DC1BF0h
00402010                      dd offset  aText1_0 ; "Text1"
00402014                      dd 30004h
etc...
```

we have all the components of the dialog, which is fundamental in event tracking. You can see names for the controls, some strange values (ID1 and ID2, semms like they are internally used as resource id) and LocalDispatcher. This is the point of all! LocalDispatcher infact points to a structure that contains all event handlers of the controls! Let's see LocalDispatcher[0]:

```
004020F4 LocalDispatcher[0] dd 0
004020F8                    dd offset  FD0_ControlsList[0]
004020FC                    dd offset FormDescriptor[0]
00402100                    dd offset  i_EVENT_SINK_QueryInterface
00402104                    dd offset  i_EVENT_SINK_AddRef
00402108                    dd offset  i_EVENT_SINK_Release
0040210C                    dd 18h dup(0)
```

the first field is always zero. The second is a backpointer to the parent structure in the controls list, the third is a backpointer to parent FormDescriptor structure. After we find some basic handlers which are present in all controls, then we have no other handlers. This is the dispatcher of a "label" control, so it has no other handlers!

So we can choose another control in the list:

```
00402040 ControlsList[3] dd 110040h
00402044                 dd 40h
00402048                 dd offset  RawData3
0040204C                 dd 30002h
00402050                 dd 0
00402054                 dd 0
00402058                 dd offset LocalDispatcher[3]
0040205C                 dd 7DC1C10h
00402060                 dd offset  aCommand1_0 ; "Command1"
00402064                 dd 30002h
```

as before we go and see the LocalDispatcher[3]:

```
00402278 LocalDispatcher[3] dd 0
0040227C                    dd offset  ControlsList_3_
00402280                    dd offset  FormDescriptor_0_
00402284                    dd offset  i_EVENT_SINK_QueryInterface
00402288                    dd offset  i_EVENT_SINK_AddRef
0040228C                    dd offset  i_EVENT_SINK_Release
00402290                    dd offset onClickAbout

00402294                    dd 10h dup(0)
```

voil, what we were searching for! We have the onClickAbout pointer, that points to:

```
004023FD onClickAbout:
004023FD     sub dword ptr [esp+4], 3Fh
00402405 onClickAbout1:
00402405     jmp onClickAboutRoutine
```

that is a transfer area (internal function dispatcher) for local event handlers. So now we know what routine is executed when "About" button is pressed. The problem is, how did I know that it is a onClick handler? It could be a onOver, onMove etc etc? To answer this question let's see the Label1 dispatcher, we know that it has a onOver handler, we can see it at runtime (and we know it is label1 from line 004017FC remember?)

```
00402330 LocalDispatcher[5] dd 0
00402334                    dd offset ControlsList[5]
00402338                    dd offset FormDescriptor[0]
0040233C                    dd offset  i_EVENT_SINK_QueryInterface
00402340                    dd offset  i_EVENT_SINK_AddRef
00402344                    dd offset  i_EVENT_SINK_Release
00402348                    dd 0
0040234C                    dd 0
00402350                    dd 0
00402354                    dd 0
00402358                    dd 0
0040235C                    dd 0
00402360                    dd 0
00402364                    dd 0
00402368                    dd 0
0040236C                    dd offset onOverLabel1
00402370                    dd 8 dup(0)
```

we see the common handler and backpointers, then far away the onOver handler pointer. So my idea is that every field of the LocalDispatch structure is a pointer to a given event handler (00402348 would be onClick etc), in addition the LocalDIspatch structure seems to be not exactly the same for all control types, so if you want to map all controls handlers you should write an app which uses all possible handlers and see where they are placed in this struct. Now we can go back to FormDescriptor[0] and check the last pointer, that was FD0Dispatcher, it points to

```
004020E0  FD0_Dispatcher dd offset  onClickAboutPre1
004020E4                 dd offset  onClickAbout1
004020E8                 dd offset  onClickCheck1
004020EC                 dd offset  OnOverForm1
004020F0                 dd offset  onOverLabel11
```

a simple list to all import transfers addresses (they point directly to the jumps, the pointers in local dispatchers instead point to previous line!). As for this form, we can go in ModulesList[3] and check FormDescriptor[1], we will find the controls list and associated event handlers.

In particular, in this form there is a moving button, if we check the LocalDispatcher of this button we see:

```
00401E70 FD1_LocalDispatcher[1] dd 0
00401E74                        dd offset FD1_ControlsList[1]
00401E78                        dd offset  FormDescriptor_1_
00401E7C                        dd offset  i_EVENT_SINK_QueryInterface
00401E80                        dd offset  i_EVENT_SINK_AddRef
00401E84                        dd offset  i_EVENT_SINK_Release
00401E88                        dd offset onClickOk
00401E8C                        dd 0
00401E90                        dd 0
00401E94                        dd 0
00401E98                        dd 0
00401E9C                        dd 0
00401EA0                        dd 0
00401EA4                        dd 0
00401EA8                        dd 0
00401EAC                        dd offset onOverOk
00401EB0                        dd 7 dup(0)
```

the other controls in the list just have default handlers.

We have mapped all the resource and relative event handlers just by examining data structures (and using a bit of zen!).

Now we can work on the code. Let's say we want to find a correct serial for a given name. We know that the routine called when the Check button is pressed is at address 00402FD0, so we start from there:

```
-standard function initialization

 00402FD0  push ebp         ;allocate private stackframe
 00402FD1  mov ebp, esp
 00402FD3  sub esp, 0Ch

 00402FD6  push offset vbaExceptHandler   ;installing default seh
 00402FDB  mov eax, large fs:0
 00402FE1  push eax
 00402FE2  mov large fs:0, esp
 00402FE9  sub esp, 74h
 00402FEC  push ebx         ;save registers area
 00402FED  push esi
 00402FEE  push edi

-loading destructors
 00402FEF  mov [ebp-0Ch], esp
 00402FF2  mov dword ptr [pDestruct], offset  Destructors_2

-allocating dynamic resource
 00402FF9  mov esi, [ebp+8]
 00402FFC  mov eax, esi
 00402FFE  and eax, 1
 00403001  mov [ebp-4], eax
```

```
00403004  and esi, 0FFFFFFFEh
00403007  push esi
00403008  mov [ebp+8], esi
0040300B  mov ecx, [esi]
0040300D  call dword ptr [ecx+4] ;  Zombie_AddRef
```

the parent object (the form) is passed as parameter (ebp+8, sorry i forgot to resolve ebp based frame function in ida!) and the COM tecnology uses AddRef to increment reference count of the object (instantiation). For those of you that don't know this, COM objects are responsible for their lifetime, the resources they use are allocated until the reference count is ¿0, when it reaches 0 the objects enter zombie state and can be deallocated to free resources (well, things are a little more complex, see COM object management documentation to know more on this topic).

-background color changing

```
00403010  xor eax, eax
00403012  lea edx, [ebp-24h]
00403015  mov ebx, 80020004h
0040301A  mov edi, 0Ah
0040301F  mov [ebp-24h], eax   ;zero vars
00403022  push edx
00403023  mov [ebp-34h], eax
00403026  mov [ebp-44h], eax
00403029  mov [ebp-1Ch], ebx
0040302C  mov [ebp-24h], edi
0040302F  call ds:rtcRandomNext     ;get random fp values
00403035  fstp dword ptr [COlor_R]
00403038  lea eax, [ebp-34h]
0040303B  mov [ebp-2Ch], ebx
0040303E  push eax
0040303F  mov [ebp-34h], edi
00403042  call ds:rtcRandomNext
00403048  fstp dword ptr [Color_G]
0040304B  lea ecx, [ebp-44h]
0040304E  mov [ebp-3Ch], ebx
00403051  mov ebx, [esi]
00403053  push ecx
00403054  mov [ebp-44h], edi
00403057  call ds:rtcRandomNext    ;color_B not stored, used directly eax
0040305D  fmul ds:_0255            ;multiply each rand fpu * 255
00403063  mov edi, ds:__vbaR8IntI2
00403069  fnstsw ax
0040306B  test al, 0Dh
0040306D  jnz loc_40312F           ;fpexception
00403073  call edi ;  __vbaR8IntI2  ;cast from _fpu real 8 bytes_ to _integer 2 bytes_
00403075  fld dword ptr [ebp-7Ch]
00403078  fmul ds:_0255
0040307E  push eax
0040307F  fnstsw ax
00403081  test al, 0Dh
00403083  jnz loc_40312F           ;fpexception
00403089  call edi ;  __vbaR8IntI2
```

```
 0040308B  fld dword ptr [ebp-78h]
 0040308E  fmul ds:_0255
 00403094  push eax
 00403095  fnstsw ax
 00403097  test al, 0Dh
 00403099  jnz loc_40312F           ;fpexception
 0040309F  call edi ;  __vbaR8IntI2
 004030A1  push eax
 004030A2  call ds:rtcRgb
 004030A8  push eax                 ;rgb resulting from previous calculus
 004030A9  push esi                 ;form object instance
 004030AA  call dword ptr [ebx+64h] ;  MSVBVM_UnkFunc2 (Set back form color)
 004030AD  test eax, eax
 004030AF  fnclex
 004030B1  jge short  loc_4030C2     ;taken

 004030B3  push 64h
 004030B5  push offset dword_402590
 004030BA  push esi
 004030BB  push eax
 004030BC  call ds:__vbaHresultCheckObj

 004030C2  lea edx, [ebp-44h]
 004030C5  lea eax, [ebp-34h]
 004030C8  push edx
 004030C9  lea ecx, [ebp-24h]
 004030CC  push eax
 004030CD  push ecx
 004030CE  push 3      ;number of objects
 004030D0  call ds:__vbaFreeVarList   ;free vars used in previous rtcRandomNext

-name/serial check
 004030D6  add esp, 10h
 004030D9  call  VoidCheck
 004030DE  test ax, ax
 004030E1  jnz short  loc_4030E8
 004030E3  call SerialValidation
 004030E8  mov dword ptr [ebp-4], 0
 004030EF  Destr_2_0:
 004030EF  wait
 004030F0  push offset  loc_403110
 004030F5  jmp short EndRoutine
```

SerialValidation is the routine that checks the serial, so we will check it (see later). VoidCheck is the routine that checks if text fields are void, if so display the error message then exit.

```
-destructors and ending stuff

 004030F7  Destr_2_1:               ;called in case of error
 004030F7  lea edx, [ebp-44h]
 004030FA  lea eax, [ebp-34h]
 004030FD  push edx
 004030FE  lea ecx, [ebp-24h]
 00403101  push eax
 00403102  push ecx
 00403103  push 3    ;number of vars to free
 00403105  call ds:__vbaFreeVarList
 0040310B  add esp, 10h
 0040310E  retn
 0040310F  EndRoutine
 0040310F  retn         ;goes to 00403110
 00403110  mov eax, [ebp+8]         ;ptr to form object
 00403113  push eax
 00403114  mov edx, [eax]
 00403116  call dword ptr [edx+8]   ;Zombie_Release (decrease reference count
 00403119  mov eax, [ebp-4]         ;for form object)
 0040311C  mov ecx, [ebp-14h]
 0040311F  pop edi                  ;save registers area
 00403120  pop esi
 00403121  mov large fs:0, ecx      ;restore exception handler
 00403128  pop ebx
 00403129  mov esp, ebp             ;delete private stackframe
 0040312B  pop ebp
 0040312C  retn 4

 0040312F  jmp loc_4011AC    ; __vbaFPException
```

are you beginning to feel the VB framework? It's really easy as you can see, the structure of the code is always the same.

Again let's see all the code, so you can understand how easy is code analisys:

```
SerialValidation

 004032C0 push ebp               ;allocate private stackframe
 004032C1 mov ebp, esp
 004032C3 sub esp, 8
 004032C6 push offset  vbaExceptHandler  ;allocate exception handler
 004032CB mov eax, large fs:0
 004032D1 push eax
 004032D2 mov large fs:0, esp
 004032D9 sub esp, 158h
 004032DF push ebx               ;registers save area
 004032E0 push esi
 004032E1 push edi

-destructors allocation and initialization
 004032E2 mov [ebp+var_8], esp
 004032E5 mov [ebp+var_4], offset Destructors_5
 004032EC mov eax, Form1Instance
 004032F1 xor edi, edi
 004032F3 cmp eax, edi            ;is form1 instanced?
 004032F5 mov [ebp+var_20], edi   ;zero vars
 004032F8 mov [ebp+var_30], edi
 004032FB mov [ebp+var_40], edi
 004032FE mov [ebp+var_50], edi
 00403301 mov [ebp+var_54], edi
 00403304 mov [ebp+var_58], edi
 00403307 mov [ebp+var_5C], edi
 0040330A mov [ebp+var_6C], edi
 0040330D mov [ebp+var_7C], edi
 00403310 mov [ebp+var_8C], edi
 00403316 mov [ebp+var_9C], edi
 0040331C mov [ebp+var_AC], edi
 00403322 mov [ebp+var_BC], edi
 00403328 mov [ebp+var_CC], edi
 0040332E mov [ebp+var_DC], edi
 00403334 mov [ebp+var_EC], edi
 0040333A mov [ebp+var_FC], edi
 00403340 mov [ebp+var_10C], edi
 00403346 mov [ebp+var_11C], edi
 0040334C mov [ebp+var_13C], edi
 00403352 mov [ebp+var_14C], edi
 00403358 mov [ebp+var_15C], edi
 0040335E jnz short  loc_403375          ;jump if form1 istanced
 00403360 push offset Form1Instance
 00403365 push offset FormDescriptor_0_
 0040336A call ds:__vbaNew2             ;if not instanced the form wuold
 00403370 mov eax, Form1Instance        ;have been created here
-copy vars from form instance to local vars
 00403375 mov ecx, [eax]
 00403377 push eax                  ;form object instance
```

```
00403378 call dword ptr [ecx+308h] ;MSVBVM_UnkFunc (some sort of addref)
0040337E mov esi, ds:__vbaVarMove
00403384 mov ebx, 9
00403389 lea edx, [ebp+var_6C]
0040338C lea ecx, [ebp+Text1]      ;note, text1 object, not string!
0040338F mov [ebp+var_64], eax     ;(parameter in ecx, fastcall style)
00403392 mov [ebp+var_6C], ebx
00403395 call esi ;  __vbaVarMove   ;copy text1 in local var
00403397 mov eax, Form1Instance
0040339C cmp eax, edi
0040339E jnz short loc_4033B5      ;avoid allocation of form if it exists
004033A0 push offset Form1Instance
004033A5 push offset FormDescriptor_0_
004033AA call ds:__vbaNew2
004033B0 mov eax, Form1Instance
004033B5 mov edx, [eax]
004033B7 push eax
004033B8 call dword ptr [edx+30Ch] ; MSVBVM_UnkFunc (as above)
004033BE lea edx, [ebp+var_6C]
004033C1 lea ecx, [ebp+Text2]      ;note, text2 object, not string!
004033C4 mov [ebp+var_64], eax     ;(parameter in ecx, fastcall style)
004033C7 mov [ebp+var_6C], ebx
004033CA call esi ;  __vbaVarMove   ;copy text2 in local var
```

as you can see COM tecnology strikes again: in the form object there is the list of controls, each one with his own data (text controls have a simple string). So what does the code do? It simply copies the data from those object in local variables (Text1 and Text2), it uses `__vbaVarMove` to copy data. Attention: it does not copy only the string, but the whole text object! The parameter is passed via ecx register (fastcall convention), and is a pointer to the memory space that receives object data. So if you want to know the data of the object (the text in this case), after `__vbaVarMove` go to data pointed by Text1 (ebp-50h), and see the third dword: it is a pointer to the unicode string for name inserted. So the structure should be as follows:

```
RT_Text_Object:

 +00 SizeOf(RT_Text_Object)

 +04 Method1

 +08 TextPointer

 +0C etc (other values or methods)
```

ok, we now know that the proggy has copied locally the data of text objects, we also know the addresses of this objects, so we can easily track all movements on theese strings.

```
-check length of name

 004033CC mov eax, 1
 004033D1 lea ecx, [ebp+Text1]
 004033D4 mov [ebp+var_104], eax
 004033DA mov [ebp+var_F4], eax
 004033E0 lea eax, [ebp+var_10C]
 004033E6 lea edx, [ebp+var_6C]
 004033E9 push eax
 004033EA mov esi, 2
 004033EF push ecx              ;object (Text1)
 004033F0 push edx              ;var object that receives length
 004033F1 mov [ebp+var_10C], esi
 004033F7 mov [ebp+var_FC], esi
 004033FD mov [ebp+var_114], edi
 00403403 mov [ebp+var_11C], esi
 00403409 call ds:__vbaLenVar   ;get length of text1 (name)
 0040340F push eax              ;var object for target value of subtraction
 00403410 lea eax, [ebp+var_FC]
 00403416 lea ecx, [ebp+var_7C]
 00403419 push eax              ;var object for value to subtract (1)
 0040341A push ecx              ;var object for result of subtraction
 0040341B call ds:__vbaVarSub
```

some easy function here, the first function gets the Text1 and gets its length. As before keep in mind that the pointers points to OBJECTS, so vbaLenVar does no return the length of the string, but the object that contains the length. The object is as before:

```
RT_Var_Object:

 +00 SizeOf(RT_Text_Object)

 +04 Method1

 +08 Data     (length of string)

 +0C etc (other values or methods)
```

so you must look the third dword at the memory pointed by the result pointer of vbaLenVar. Same thing for all other `__vba***`. So here it gets length of string and decrements it by one.

```
-main for() cycle

 00403421 push eax                ;(strlen(Text1)-1) object
 00403422 lea edx, [ebp+var_11C]
 00403428 lea eax, [ebp+var_15C]
 0040342E push edx
 0040342F lea ecx, [ebp+var_14C]
 00403435 push eax
 00403436 lea edx, [counter]
 00403439 push ecx
 0040343A push edx                ;counter

 0040343B call ds:__vbaVarForInit ;prepare for() cycle
 00403441 mov edi, ds:__vbaVarMul
 00403447 mov ebx, ds:__vbaVarAdd


-loop here
 0040344D test eax, eax           ;is cycle finished? (true = loop, false = exit)
 0040344F jz  EndCheck             ;then check ends
 00403455 mov eax,  Form1Instance
 0040345A test eax, eax
 0040345C jnz short  loc_403473       ;is form1 instanced?
 0040345E push offset  Form1Instance  ;create instance if not
 00403463 push offset FormDescriptor[0]
 00403468 call ds:__vbaNew2
 0040346E mov eax,  Form1Instance
 00403473 mov ecx, [eax]             ;use current instance
 00403475 push eax                   ;form1 instance
 00403476 call dword ptr [ecx+308h]  ;MSVBVM_UnkFunc
 0040347C lea edx, [ebp+var_5C]
 0040347F push eax
 00403480 push edx
 00403481 call ds:__vbaObjSet        ;set object var to [ebp+var_5C] ptr
 00403487  mov eax, 1
 0040348C  lea ecx, [counter]
 0040348F  mov [ebp+var_F4], eax      ;set vars to true
 00403495  mov [ebp+var_A4], eax
 0040349B  mov [ebp+var_114], eax
 004034A1  mov eax, [ebp+var_5C]
 004034A4  mov [ebp+var_94], eax
 004034AA  lea eax, [ebp+var_AC]
 004034B0  push eax                   ;value to add (1)
 004034B1  lea edx, [ebp+var_11C]
 004034B7  push ecx                   ;target value of addiction (counter)
 004034B8  lea eax, [ebp+var_8C]
 004034BE  push edx                   ;value to add
 004034BF push eax                   ;target
 004034C0 mov [ebp+var_104], 64h
 004034CA mov [ebp+var_10C], esi
 004034D0 mov [ebp+var_FC], esi
 004034D6 mov [ebp+var_AC], esi
```

```
004034DC mov [ebp+var_11C], esi
004034E2 mov [ebp+var_5C], 0
004034E9 mov [ebp+var_9C], 9
004034F3 call ebx ;  __vbaVarAdd
004034F5 push eax                    ;result object
004034F6 call ds:__vbaI4Var          ;convert result object to int_4_bytes
004034FC lea ecx, [ebp+var_9C]
00403502 push eax                    ;position
00403503 lea edx, [ebp+var_BC]
00403509 push ecx                    ;length
0040350A push edx                    ;target string (name)
0040350B call ds:rtcMidCharVar       ;extract one char from name at counter+1 position
00403511 lea eax, [ebp+var_BC]
00403517 lea ecx, [ebp+var_54]
0040351A push eax                    ;extracted char object
0040351B push ecx                    ;name string
0040351C call ds:__vbaStrVarVal
00403522 push eax                    ;extracted char value
00403523 call ds:rtcAnsiValueBstr    ;get ansi value from extracted value
00403529 push eax                    ;ansi string of value of mid
0040352A call ds:__vbaStrI2          ;convert it to int_2_bytes
```

this code increments the for() counter and gets the char at counter+1 position with mid function. Then it obtains the numeric value of the extracted char, then at the end it converts it into a unicode string representing that char in decimal number. Example: at iteration x it extracts Name[x+1] char, let's assume it is an A (0x41). Then it gets the numeric value (0x41) and then the unicode string 65 (dec for 0x41), unicode means the string will be (0x36 0x00 0x35 0x00).

```
00403530 mov edx, eax
00403532 lea ecx, [ebp+var_58]  ;ptr to unicode int2bytes
00403535 call ds:__vbaStrMove   ;move unicode decimal number in ebp+var48
0040353B push eax               ;unicode decimal number
0040353C call ds:rtcR8ValFromBstr  ;convert unicode decimal number string

                                ;in floating point value
00403542 call ds:__vbaFpI4         ;convert previous fp number in int4bytes
```

here we have the numeric value of the extracted char in eax

```
00403548 cdq
00403549 mov ecx, 0Ah              ;divisor
0040354E mov [ebp+var_13C], 3
00403558 idiv ecx                  ;int4bytes % 10
0040355A lea eax, [ebp+var_10C]    ;trash result
00403560 lea ecx, [counter]        ;trash divisor
00403563 mov [rest], edx    ;save rest of division
00403569 lea edx, [ebp+var_40]
0040356C push edx                  ;result string
0040356D push eax                  ;value 100 (for multiplication)
0040356E lea edx, [ebp+var_FC]
00403574 push ecx                  ;target (counter)
00403575 lea eax, [total]
00403578 push edx                  ;value to add (1)
00403579 push eax                  ;target (result)
0040357A call ebx ;  __vbaVarAdd    ;increment counter
0040357C lea ecx, [ebp+var_7C]
0040357F push eax
00403580 push ecx
00403581 call edi ;  __vbaVarMul    ;100 * (counter+1)
00403583 push eax
00403584 lea edx, [ebp+var_13C]
0040358A lea eax, [ebp+var_CC]
00403590 push edx
00403591 push eax
00403592 call edi ;  __vbaVarMul    ;(100*(counter+1)) * previous calculus

                                   ;on extracted char
00403594 lea ecx, [ebp+var_DC]
0040359A push eax
0040359B push ecx
0040359C call ds:__vbaVarInt       ;cast to int the result of all calculus
004035A2 lea edx, [ebp+var_EC]
004035A8 push eax                  ;int numeric value
004035A9 push edx                  ;string that gets total
004035AA call ds:__vbaVarCat       ;cat unicode string of int numeric value

                                   ;to unicode string of total
004035B0 mov edx, eax
004035B2 lea ecx, [ebp+var_40]
004035B5 call ds:__vbaVarMove      ;copy string of total to [ebp+var_40]
004035BB lea eax, [ebp+var_58]
004035BE lea ecx, [ebp+var_54]
004035C1 push eax
004035C2 push ecx
004035C3 push esi                  ;esi = 2 (number of vars)
004035C4 call ds:__vbaFreeStrList  ;free temp vars
004035CA add esp, 0Ch
004035CD lea ecx, [ebp+var_5C]
004035D0 call ds:__vbaFreeObj      ;free temp object
```

```
004035D6 lea edx, [ebp+var_BC]
004035DC lea eax, [ebp+var_AC]
004035E2 push edx
004035E3 lea ecx, [ebp+var_8C]
004035E9 push eax
004035EA lea edx, [ebp+var_9C]
004035F0 push ecx
004035F1 lea eax, [total]           ;temp total
004035F4 push edx
004035F5 push eax
004035F6 push 5
004035F8 call ds:__vbaFreeVarList   ;free 5 temp vars
004035FE add esp, 18h
00403601 lea ecx, [ebp+var_15C]
00403607 lea edx, [ebp+var_14C]
0040360D lea eax, [counter]
00403610 push ecx                   ;maximum value of counter
00403611 push edx                   ;incremental step
00403612 push eax                   ;actual counter
00403613 call ds:__vbaVarForNext    ;if actual counter < maximum then return true
00403619 jmp loc_40344D             ;else return false
```

this is the loop for serial calculus

```
0040361E EndCheck: ; CODE XREF: Check+18Fj
0040361E call nullsub_1             ;null calls
00403623 call nullsub_1             ;I asked _d31m0s_ to add them
00403628 call nullsub_1             ;just to check some things
0040362D call nullsub_1
00403632 call nullsub_1
00403637 call nullsub_1
0040363C lea ecx, [ebp+var_40]
0040363F lea edx, [ebp+var_30]
00403642 push ecx                   ;calculated serial (text object)
00403643 push edx                   ;inserted serial  (text object)
00403644 call ds:__vbaVarTstEq      ;compare the two string objects
0040364A test ax, ax                ;true = equal, false = different
0040364D jz short  Error             ;if strings are different then error message
0040364F call  OkMessage             ;else ok message
00403654 wait                       ;useless waste...
00403655 push offset loc_4036FE
0040365A jmp short Ending
0040365C
0040365C Error:
0040365C call  ErrorMessage
00403661 wait
00403662 push offset loc_4036FE
00403667 jmp short Ending
00403669
00403669 Destructor:                ;called in case of error
00403669 lea eax, [ebp-58h]
0040366C lea ecx, [ebp-54h]
```

```
0040366F push eax
00403670 push ecx
00403671 push 2
00403673 call ds:__vbaFreeStrList
00403679 add esp, 0Ch
0040367C lea ecx, [ebp-5Ch]
0040367F call ds:__vbaFreeObj
00403685 lea edx, [ebp-0ECh]
0040368B lea eax, [ebp-0DCh]
00403691 push edx
00403692 lea ecx, [ebp-0CCh]
00403698 push eax
00403699 lea edx, [ebp-0BCh]
0040369F push ecx
004036A0 lea eax, [ebp-0ACh]
004036A6 push edx
004036A7 lea ecx, [ebp-9Ch]
004036AD push eax
004036AE lea edx, [ebp-8Ch]
004036B4 push ecx
004036B5 lea eax, [ebp-7Ch]
004036B8 push edx
004036B9 lea ecx, [ebp-6Ch]
004036BC push eax
004036BD push ecx
004036BE push 9
004036C0 call ds:__vbaFreeVarList
004036C6 add esp, 28h
004036C9 retn
004036CA Ending:
004036CA lea edx, [ebp+var_15C]  ;free vars
004036D0 lea eax, [ebp+var_14C]
004036D6 push edx
004036D7 push eax
004036D8 push 2
004036DA call ds:__vbaFreeVarList
004036E0 mov esi, ds:__vbaFreeVar
004036E6 add esp, 0Ch
004036E9 lea ecx, [counter]
004036EC call esi ;  __vbaFreeVar
004036EE lea ecx, [ebp+var_30]
004036F1 call esi ;  __vbaFreeVar
004036F3 lea ecx, [ebp+var_40]
004036F6 call esi ;  __vbaFreeVar
004036F8 lea ecx, [ebp+var_50]
004036FB call esi ;  __vbaFreeVar
004036FD retn                 ;return to 004036FE
004036FE mov ecx, [ebp-10h]
00403701 pop edi              ;save register area
00403702 pop esi
00403703 mov large fs:0, ecx  ;restore seh handler
0040370A pop ebx
```

```
0040370B mov esp, ebp         ;delete private stackframe
0040370D pop ebp
0040370E retn
```

ok, now we discovered how the serial is built, in particular if you see the `__vbaVarTestEq` function, the two parameters passed are two string objects, one for serial inserted and one for the correct serial, so you can do serial fishing here without calculating. The algorithm we reversed is:

```
nser = nser & Int(100 * (i + 1) * (Val(Asc(Mid(Form1.Text1, i + 1, 1))) Mod 10))
```

so debugging vb applications is really easy, just keep in mind that functions use Objects instead of direct values. Once you know this, COM jungle will not be a problem!

```
Serial for "AndreaGeddon" is 5000016005004200700800011000, find your own!
```

## 3 SUMMARY

I hope that at the end of this tutorial you will have learned how to debug visual basic applications. As you can see it is really easy, you do not even need SmartCheck. The exe itself is full of precious infos, you can easily find all event handlers. Then you just have to analyse the code to understand what the program does. Remember that COM tecnology is object based, so when you see functions and you analyse their parameters and return values, you know that you are considering OBJECTS, not the values directly. The values you search (strings, numbers, etc) will be encoded in the object. We encountered a lot of rtc* and `__vba*` apis here, they always use objects, infact when there is an addition you see `__vbaVarAdd` instead of a simple "add" asm instruction, this is because the function adds two number objects! Naturally the code uses also asm instructions for direct value arithmetics, this happens when there is some casting such as `__vbaI4Var` function etc etc.

I wrote this tute because my intent is vb full decompiling, now that you know the structure of the compiled exes you can understand that decompiling is possible, and is relatively easy. Hope I will write a proggy about this one day!

GREETS AND THANKS

Thanks to `_d31m0s_` who wrote the vb app for this tutorial (i will kill him for lame messages in it!), greets to all RET friends and great reversers! Greets to all UIC members an to all #crack-it people, see you all guys!

GoodBye

| | | |
|---|---|---|
| [AndreaGeddon] | andreageddon@hotmail.com | my mail |
| | www.andreageddon.8m.com | my lame italian site |
| [RET] | www.reteam.org | RET's great site |
| [UIC] | www.quequero.org | italian university of cracking |