Parsing VB6 Form Controls

Author: David Zimmer <dzzie@yahoo.com>
Site: http://sandsprite.com

This article is not going to be overly formal, but will be jam packed with in depth info. My understanding of this topic is still in flux and it is very complicated. I am mostly documenting this for my own solidification, but it will also be handy to any poor soul who wades into these dark waters.

Our voyage starts with the main VBHeader structure: its GuiTable field at offset 0x4c, and FormCount member at offset 0x44.

After loading the vbHeader we load the GUI Tables. There is one for each component which has a visual display surface such as a Form, User Controls, etc.

```
    If FormCount > 0 Then
         fPointer = aGuiTable
         Seek fhandle, VaToFileOffset(aGuiTable) + 1
         For i = 0 To FormCount - 1
             Set gui = New CGuiTable
             gui.LoadSelf fhandle, i, fPointer
             fPointer = fPointer + gui.lStructSize
             GuiObjects.Add gui
         Next
     End If
```

The structure for the GUI table still has a bunch of unknown fields. It is currently defined as follows:

```
Private Type tGuiTable 'more: http://sandsprite.com/vb-reversing/vb-decompiler/viewtopic9937.html
         lStructSize          As Long ' 0x00 always 0x50?
         uuidObjectGUI(15)    As Byte ' 0x04 UUID of Object GUI
         Unknown1             As Long ' 0x14
         Unknown2             As Long ' 0x18
         Unknown3             As Long ' 0x1C
         Unknown4             As Long ' 0x20
         lObjectID            As Long ' 0x24 Current Object ID in the Project
         Unknown5             As Long ' 0x28
         fOLEMisc             As Long ' 0x2C OLEMisc Flags
         uuidObject(15)       As Byte ' 0x30 UUID of Object
         dataSize             As Long ' 0x40
         Unknown7             As Long ' 0x44
         aFormPointer         As Long ' 0x48 Pointer to GUI Object Info
         Unknown8             As Long ' 0x4C
End Type
```

The aFormPointer and dataSize are the two key elements. The aFormPointer will lead us to another data blob of dataSize bytes. We start parsing this blob at offset 0x5E (see http://sandsprite.com/vb-reversing/vb-decompiler/viewtopic121f.html for header).

Processing each of these tables looks about like so:

```
Dim cControlHeader As ControlHeader
Dim cArrayHeader As ControlArrayHeader
Dim gui As CGuiTable, b as Byte

  For Each gui In vbp.VBHeader.GuiObjects 'top level structs

            'start new gui object (form etc)
            Seek f, VaToFileOffset(gui.aFormPointer) + &H5E

            Do 'extract each control's config blob

                fPos = loc(f)
                Get f, fPos + 4, b

                If b = &H80 Then
                    Get #f, fPos + 1, cArrayHeader
                    cControlHeader = ArrayToControlHeader(cArrayHeader, arrayIndex)
                Else
                    Get #f, fPos + 1, cControlHeader
                End If

                'save control object

             Loop Until Not FindControlEndPos(f, fPos, cControlHeader, cp)

             gui.lastOffsetProcessed = loc(f)
    Next
```

The ControlHeader, and ControlArrayHeader structures look like the following:

```
Public Type ControlHeader
    length As Long
    cId As Byte
    cname As String
    null As Byte
    cType As Byte
End Type

Private Type ControlArrayHeader 'this changed see update at end...
    length As Integer
    un1 As Byte
    arrayflag As Byte  'must be 0x80
    cId As Byte
    aryIndex As Integer
    cname As String    'ex: 00 05 Text1
    null As Byte       'string null terminator
    cType As Byte
End Type
```

Some example data may look like the following:

```
Offset       0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000   29 00 00 80 01 F4 01 05  00 54 65 78 74 31 00 02   )..€.ô...Text1..
00000010   01 F4 01 04 78 00 78 00  BF 04 EF 01 0B 05 00 54   .ô..x.x.¿.ï....T
```

```
00000020   65 78 74 31 00 12 00 00  FF                        ext1....ÿ

Public Type ControlArrayHeader
    length As Integer           ` 00 29
    un1 As Byte                 ` 00
    arrayflag As Byte           ` 80
    cId As Byte                 ` 01
    aryIndex As Integer         ` 1f4 (500)
    cname As String             ` 0005 Text1
    null As Byte                ` 00 – string null terminator
    cType As Byte               ` 02 – vbTextBox (def a byte type)
End Type
```

The cType byte is defined as follows:

```
'see 0x66110A30 at runtime for "gaps"
Public Enum ControlTypes
    vbPictureBox = 0
    vbLabel = 1
    vbTextBox = 2
    vbFrame = 3
    vbCommandButton = 4
    vbCheckBox = 5
    vbOptionButton = 6
    vbComboBox = 7
    vbListBox = 8
    vbHScroll = 9
    vbVScroll = 10
    vbTimer = 11
    vbForm = 13
    vbDriveListBox = 16
    vbDirectoryListBox = 17
    vbFileListBox = 18
    vbMenu = 19
    vbMDIForm = 20
    vbShape = 22
    vbLine = 23
    vbImage = 24
    vbData = 37
    vbOLE = 38
    vbUserControl = 40
    vbPropertyPage = 41
    vbUserDocument = 42
    vbExternal = &HFF
End Enum
```

In the Semi-VBDecompiler source, the top level function is frmMain.ProcessControls:

https://github.com/VBGAMER45/Semi-VB-Decompiler/blob/master/Semi%20VB%20Decompiler/frmMain.frm#L3789

The function ends at line 4536 and is 747 lines of code. Sub functions within it will also alter the file pointer along the way.

I have rewritten this function several times trying to understand everything that is going on.

Each `gui.aFormPointer` blob contains a series of structures representing all of the controls placed onto the control surface. The control header.length field will take you to the end of that individual structure, however there are a dynamic number of control codes embedded after each structure.

In the Semi-VBDecompiler source this is checked here:

https://github.com/VBGAMER45/Semi-VB-Decompiler/blob/master/Semi%20VB%20Decompiler/frmMain.frm#L4405

Below that you will see it gobble up a series of control codes based on what it finds (line 4424). If it goes to far, then it will rewind the file pointer by one (4438) and start a new control.

The control codes are defined as follows:

```
Enum FormPropCtrlCodes
    fpcc_BeginNewChild = 1                  ' New child control, in existing parent
    fpcc_EndCurrentControlParent = 2        ' End parent control and its children
    fpcc_ContinueCurrentControlParent = 3 ' Another child control follows
    fpcc_FinalEndConfigBlock = 4
    fpcc_StartOfMenus = 5
End Enum
```

You can find a discussion on these in the vbdecompiler.theautomators.com message board:

http://sandsprite.com/vb-reversing/vb-decompiler/viewtopic1df7.html

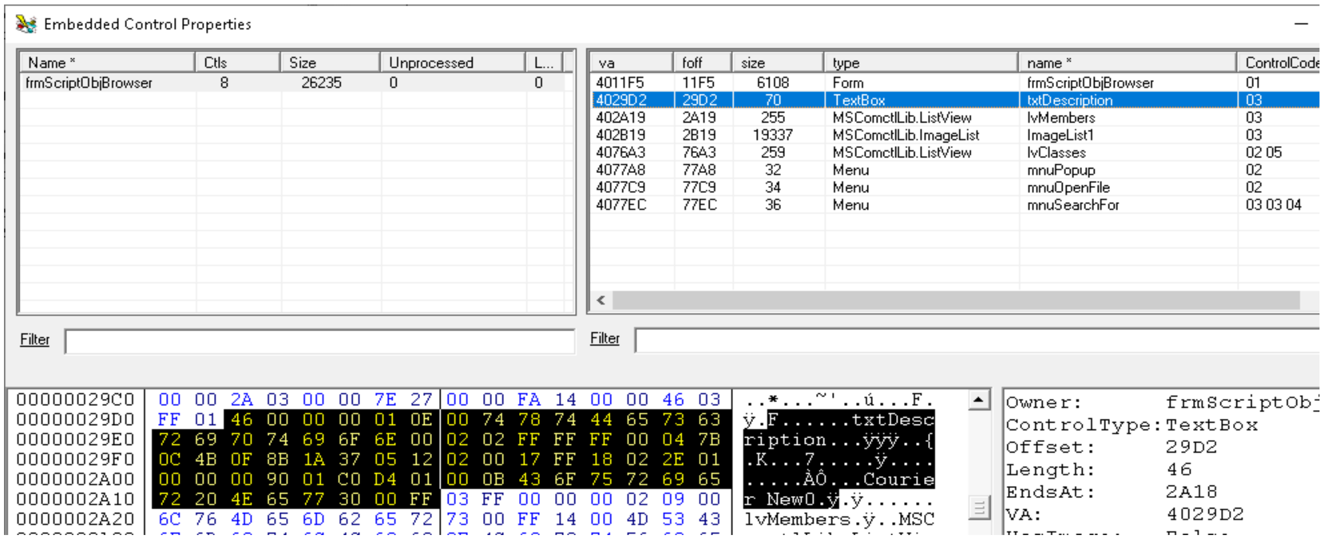
My understanding of the rules for reading these dynamically sized control codes is still in progress. Valid sequences differ for the normal form controls and the menus. If the logic is off, it will start gobbling up bytes that are actually part of the next control header and the extraction will be broken from there on. This can easily happen if the control header length contains bytes which would be valid `FormPropCtrlCodes` codes.

This situation could be handled heuristically with something like:

```
 For Each gui In vbp.VBHeader.GuiObjects

       startPos = VaToFileOffset(gui.aFormPointer)

       Do 'extract each control config blob

tryAgain:
           fPos = loc(f)
           'load control header here

           If isBadLength(cControlHeader.length, startPos, fPos, gui.dataSize) Then
               lateRewinds = lateRewinds + 1 'track error recoveries
               Seek f, fPos                  'move file pointer back 1
               GoTo tryAgain
           End If
```

This does work, but feels sloppy.

Current versions of vbdec have taken to tracking everything in an attempt to shed light on all of the variations possible. Below is the new `Form Object Properties` form:


Embedded Control Properties

Name *	Ctls	Size	Unprocessed	L...
frmScriptObjBrowser	8	26235	0	0

va	foff	size	type	name *	ControlCode
4011F5	11F5	6108	Form	frmScriptObjBrowser	01
4029D2	29D2	70	TextBox	txtDescription	03
402A19	2A19	255	MSComctlLib.ListView	lvMembers	03
402B19	2B19	19337	MSComctlLib.ImageList	ImageList1	03
4076A3	76A3	259	MSComctlLib.ListView	lvClasses	02 05
4077A8	77A8	32	Menu	mnuPopup	02
4077C9	77C9	34	Menu	mnuOpenFile	02
4077EC	77EC	36	Menu	mnuSearchFor	03 03 04

Filter

Filter

```
000029C0  00 00 2A 03 00 00 7E 27 00 00 FA 14 00 00 46 03  ..*...~'..ú...F.
000029D0  FF 01 46 00 00 00 01 0E 00 74 78 74 44 65 73 63  ÿ.F......txtDesc
000029E0  72 69 70 74 69 6F 6E 00 02 02 FF FF FF 00 04 7B  ription...ÿÿÿ..{
000029F0  0C 4B 0F 8B 1A 37 05 12 02 00 17 FF 18 02 2E 01  .K...7.....ÿ....
00002A00  00 00 00 90 01 C0 D4 01 00 0B 43 6F 75 72 69 65  .....ÀÔ...Courie
00002A10  72 20 4E 65 77 30 00 FF 03 FF 00 00 00 02 09 00  r New0.ÿ.ÿ......
00002A20  6C 76 4D 65 6D 62 65 72 73 00 FF 14 00 4D 53 43  lvMembers.ÿ..MSC
```

```
Owner:       frmScriptObj
ControlType: TextBox
Offset:      29D2
Length:      46
EndsAt:      2A18
VA:          4029D2
```

All of this information is also available to the script automation interface. Scripts can be run on the command line to bulk process samples. The following except from ./scripts/gui_scan.js will export all of this data to a database for bulk analysis.

Example vbdec cmdline: `vbdec.exe "sample_path" /js gui_scan.js /term /min /nodisasm`

```
for(i=1; i <= vbp.vBHeader.GuiObjects.count(); i++){

      gui = vbp.vBHeader.GuiObjects(i)
      cc = gui.cco.ControlProps.count
      bnp = gui.bytesNotProcessed

      sql = "Insert into tblGUI (pid,name,ctlcount,bytesNotProcessed,hash)" +
            "Values('"+pid+"','"+gui.name+"','"+cc+"','"+bnp+"','"+hash+"')"

      cn.execute(sql)

      for(j=1; j <= gui.cco.ControlProps.count; j++){

          cp = gui.cco.ControlProps.baseCollection(j)

          sql = "Insert into tblCtls (pid,name,ctlCodes,ctlCodesDesc,hash," +
                "rewinds,length,bytesNotProcessed,controlType) " +
                "Values('"+pid+"','"+cp.owner+"','"+cp.GetControlCodes() +
                "','"+cp.GetControlCodes(1)+"','"+hash+"','"+cp.rewindCount +
                "','"+cp.length+"','"+bnp+"','"+cp.controlType+"')"

          cn.execute(sql)
      }
}
```

After building such a database you can quickly list all valid and invalid control code sequences with sql queries such as:

```
SELECT DISTINCT ctlCodes FROM tblCtls where bytesNotProcessed = 0
SELECT DISTINCT ctlCodes FROM tblCtls where bytesnotprocessed <> 0
```

To build a corpus of vb6 samples you can download the old Planet Source Code archive and batch compile them from the command line:

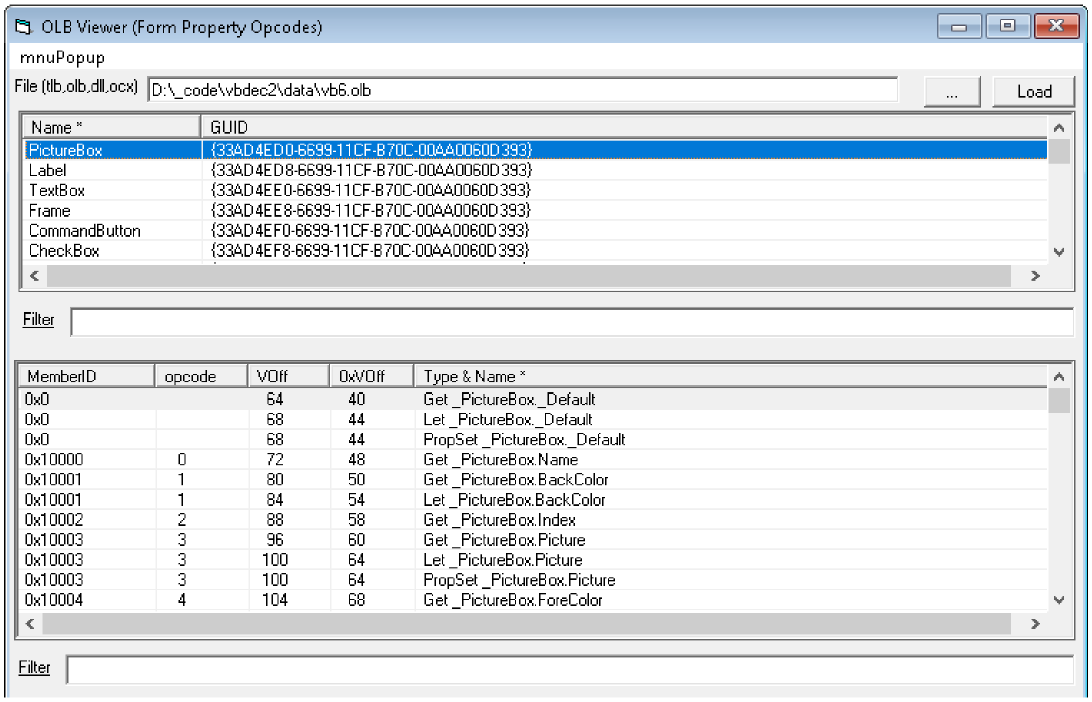
https://gist.github.com/dzzie/3a03cbd6025a2a76eda50246be111896
https://github.com/Planet-Source-Code/PSCIndex/blob/master/ByWorld/visual-basic.md

I am currently in the process of trying to fish out invalid control code sequences and stress testing the control extraction routines. Disassembling vbdec itself was leading to errors on one of its forms due to the cause mentioned above. Overall its a grueling process but unfortunately is a required task.

After accurately extracting all of the individual control property blobs, we have not yet even begun to start extracting the individual control properties such as height and width. In Semi-VBDecompiler you can find these routines individualized for each control type. This is what makes up the first 2000 lines of modControls.bas

https://github.com/VBGAMER45/Semi-VB-Decompiler/blob/master/Semi%20VB%20Decompiler/modControls.bas#L1987

I have completely ditched the form rebuilding and property extraction routines in vbdec now. Form reconstruction is not a priority for me and it was a boat load of code. I am however still going to work on understanding these property blobs as time permits (eg. 0x66065EFD PropLoadProp). In current versions of vbdec a new `OLB Viewer` form has been added.



File (tlb,olb,dll,ocx) D:_code\vbdec2\data\vb6.olb

Name *	GUID
PictureBox	{33AD4ED0-6699-11CF-B70C-00AA0060D393}
Label	{33AD4ED8-6699-11CF-B70C-00AA0060D393}
TextBox	{33AD4EE0-6699-11CF-B70C-00AA0060D393}
Frame	{33AD4EE8-6699-11CF-B70C-00AA0060D393}
CommandButton	{33AD4EF0-6699-11CF-B70C-00AA0060D393}
CheckBox	{33AD4EF8-6699-11CF-B70C-00AA0060D393}

MemberID	opcode	VOff	0xVOff	Type & Name *
0x0		64	40	Get _PictureBox._Default
0x0		68	44	Let _PictureBox._Default
0x0		68	44	PropSet _PictureBox._Default
0x10000	0	72	48	Get _PictureBox.Name
0x10001	1	80	50	Get _PictureBox.BackColor
0x10001	1	84	54	Let _PictureBox.BackColor
0x10002	2	88	58	Get _PictureBox.Index
0x10003	3	96	60	Get _PictureBox.Picture
0x10003	3	100	64	Let _PictureBox.Picture
0x10003	3	100	64	PropSet _PictureBox.Picture
0x10004	4	104	68	Get _PictureBox.ForeColor

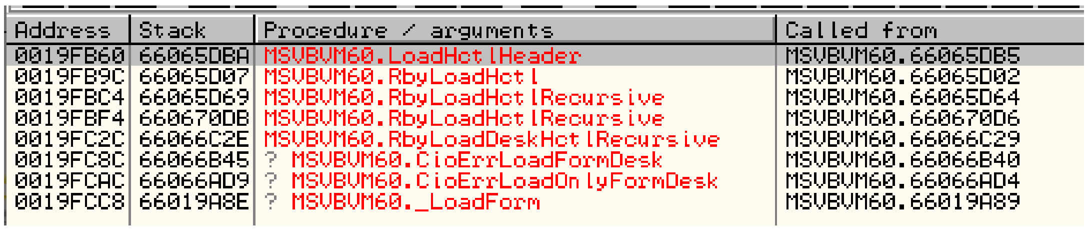
The opcode embedded within the config blob is the memberID – 0x10000. This is displayed in the OlbViewer to help manually read the hex blobs. I will need to see if I can create a fully dynamic way to process the blobs. As always there is much more work to do.

I must eventually extract properties such as .tag, .text, and .caption in the future. Unfortunately the opcode changes with every control and you must fully process a control and its variable size fields to get to the one property you do want :(

For now I codgered together an image extractor routine and added an “Extract Strings” menu item to the form properties viewer. That plus the OlbViewer should get you through in a pinch.

If you want to see how the runtime itself processes these fields, grab a reference copy of the vb runtime with symbols: http://sandsprite.com/vb-reversing/files/msvbvm60.zip

Place those files in the same folder as your target exe and start in OllyDbg. You can use offsets that vbdec shows you to place hardware breakpoints on access to data of interest. For example setting a breakpoint on the 0x80 byte of a textbox array’s header we hit a break on access at 0x66065E71 with the following call stack:



```
Address  Stack    Procedure / arguments                      Called from
0019FB60 66065DBA MSVBVM60.LoadHctlHeader                    MSVBVM60.66065DB5
0019FB9C 66065D07 MSVBVM60.RbyLoadHctl                       MSVBVM60.66065D02
0019FBC4 66065D69 MSVBVM60.RbyLoadHctlRecursive              MSVBVM60.66065D64
0019FBF4 660670DB MSVBVM60.RbyLoadHctlRecursive              MSVBVM60.660670D6
0019FC2C 66066C2E MSVBVM60.RbyLoadDeskHctlRecursive          MSVBVM60.66066C29
0019FC8C 66066B45 ? MSVBVM60.CioErrLoadFormDesk              MSVBVM60.66066B40
0019FCAC 66066AD9 ? MSVBVM60.CioErrLoadOnlyFormDesk          MSVBVM60.66066AD4
0019FCC8 66019A8E ? MSVBVM60._LoadForm                       MSVBVM60.66019A89
```

In this manner we can start to trace out where the control codes are used.

In practice dumping large amounts of control codes from actual binaries and looking for patterns which break the parser is easier though, so that’s where I am starting ! Its not a fun task, and completely unpaid, so I will take an easy win if I can find it (plus there is like 10 people in the world who are actually reading this)

Anyway...this is the end of my late night brain dump for now, more to come... enjoy.

Update:
So it took about 4 days to implement all the control code tracking stuff (framework, UI display, and bulk processing script).

Then I ran an automated scan on a bunch of PSC submissions to find problems. After another 2 days of data analysis 4-5 tricky bugs were uncovered in the control processing code. I believe everything should be solid now (fingers crossed).

I am not really sure how much time I have into rewriting the full 750 lines of the ProcessControls routine. It has to be over a month and a half of full time labor though. Changes used to require careful testing. The new Object Properties form now includes enough stats to make it almost trivial.

If you find a sample which still displays Unprocessed <> 0 feel free to submit it to me as a bug. I did implement the late rewind capability but that should consistently show as 0 now too.

Lessons learned:

- tracking control codes, bytes left, total control count, rewinds, and late rewinds really helped
- Export to database and bulk analysis also really helped.
- the command line script support for bulk runs keeps proving its worth
- debug messages littered within the code were to dense to visualize even with a filter
- making data visible and easily digestible is the name of the game
- the late rewind heuristic was actually very effective but hopefully not necessary any more.

- 03 01 was never a valid control code
- 03 03 is valid for a menu but not other controls
- 03 04 can be valid, but only if bytesLeft = 0
- the 0x80 byte to check for an array header has a short story:

```
Public Type ControlHeader
    length As Long
    cId As Byte

Private Type ControlArrayHeader
    length As Integer
    un1 As Byte
    arrayflag As Byte  'must be 0x80
    cId As Byte
```

This structure always bothered me a little bit. Why is length so much shorter for array elements. First error I saw I had to treat the integer as unsigned for it to be right. Then I noticed that un1 was actually part of the length. A 3 byte length with the 4th byte reserved as a flag? C doesnt have a 3 byte type.

Then it finally dawned on me that they reserve the 0x80000000 bit for the isArray flag and keep the other 31 bits for the length. Since the 32nd bit is the sign bit which would turn the value negative, technically it would be unused, and would be free as a flag.

Seems simple once you hear it but when you are only looking at data variations and an evolution of bugs it can take longer to have that moment of realization. Final code now looks like:

```
Private Type ControlArrayHeader
    length As Long

Get f, fPos + 4, b 'still just grab high byte to check flag
If (b And &H80) = &H80 Then 'sign bit is set its an array
      Get #f, fPos + 1, cArrayHeader
      cControlHeader = ArrayToControlHeader(cArrayHeader)

Private Function ArrayToControlHeader(a As ControlArrayHeader) As ControlHeader
    Dim h As ControlHeader
    h.length = (a.length And &H7FFFFFFF) 'remove the sign bit we want the other 31
```

90’s developers were probably more prone to these tricks from asm days. Another fun example here:

http://sandsprite.com/blogs/index.php?uid=7&pid=514&year=2022