decoded.avast.io

Recovery of function prototypes in Visual Basic 6 executables - Avast Threat Labs

David Zimmer

10-13 minutes

I was recently probing into the Visual Basic 6 format trying to see how the `IDispatch` implementation resolved vtable offsets for functions.

While digging through the code, I came across several structures that I had not yet explored. These structures revealed a trove of information valuable to reverse engineers.

As we will show in this post, type information from internal structures will allow us to recover function prototypes for public object methods in standard VB6 executables.

This information is related to `IDispatch` internals and does not require a type library like those found with ActiveX components.

For our task of malware analysis, a static parsing approach will be of primary interest.

Background

Most VB6 code units are COM objects which support the

`IDispatch` interface. This allows them to be easily passed around as generic types and used with scripting clients.

`IDispatch` allows functions to be called dynamically by name. The following VB6 code will operate through `IDispatch methods`:

```
Public Function myFunc(arg1 As Long, arg2 As Long) As Long
    myFunc = (arg1 + 2) * arg2
End Function

Private Sub Form_Load()
    Dim obj As Object
    Set obj = Me 'calls now late bound
    Text1 = obj.myFunc(arg2:=2, arg1:="5")  'shows 14
End Sub
```

This simple code demonstrates several things:

- VB6 can call the proper method from a generic object type
- named arguments in the improper order are correctly handled
- values are automatically coerced to the correct type if possible (here the string “5” is converted to long)

If we add a DebugBreak statement, we can also see that the string “myFunc” is passed in as an argument to the underlying `__vbaLateMemNamedCallLd` function. This allows `IDispatch::Invoke` to call the proper function by string name.

From this we can intuit that the `IDispatch` implementation must know the complete function prototype along with argument names and types for correct invocation to occur.

We can further test this by changing an argument name, adding unexpected arguments, or setting an argument type to something that can not be coerced too long.

Each of these conditions raises an error before the function is

called.

Note that there is no formal type library when this code is compiled as a standard executable. This type information is instead embedded somewhere within the binary itself.

So where is this type information stored, and how can we retrieve it?

The Hunt

If we start probing the runtime at `BASIC_CLASS_Invoke` we will encounter internal functions such as `FuncSigOfMember`, `EpiGetInvokeArgs`, `CoerceArg`, and so on. As we debug the code, we can catch access to known VB6 structures and watch where the code goes from there.

Before we get deeper, I will give a quick high-level overview of the Visual Basic 6 file format.

VB6 binaries have a series of nested structures that layout all code units, external references, forms etc. They start with the `VBHeader` and then branch out, covering the various aspects of the file.

In the same way that the Windows loader reads the PE file to set up proper memory layout, the VB runtime (msvbvm60.dll) reads in the VB6 file structures to ready it for execution within its environment.

Good references for the VB6 file format include:

- Visual Basic Image Internal Structure Format – Alex Ionescu
- Semi-VBDecompiler source code – VBGamer45
- vb-decompiler.theautomaters.com message board

The structure and fields names I use in this document primarily

come from the Semi-VBDecompiler source. A useful structure browser is the free vbdec disassembler: http://sandsprite.com/vbdec/

For our work, we will start with the `ObjectTable` found through the `VBHeader->ProjectInfo->ObjectTable`.

The `ObjectTable->ObjectArray` holds the number of Object structures as defined by its `ObjectCount` field. An example is shown below:

```
CMYClass_Object dd offset CMYClass_ObjInfo; aObjectInfo
                                          ; DATA XREF: .text:CMYClass
                dd 0FFFFFFFFh             ; Const1
                dd 401EFCh                ; aPublicBytes
                dd 0                      ; aStaticBytes
                dd 0                      ; aModulePublic
                dd 0                      ; aModuleStatic
                dd 401B90h                ; aObjectName
                dd 5                      ; ProcCount
                dd offset CMYClass_ProcNamesAry; aProcNamesArray
                dd 0FFFFh                 ; oStaticVars
                dd 118003h                ; ObjectType
                dd 0                      ; Null3
Class1_Object   dd offset Class1_ObjInfo; aObjectInfo
                                          ; DATA XREF: .text:Class1_O
                dd 0FFFFFFFFh             ; Const1
```

This is the top-level structure for each individual code object. Here we will find the `ProcNamesArray` containing `ProcCount` entries. This array reveals the public method names defined for the object.

```
CMYClass_ProcNamesAry dd offset aMysub  ; DATA XREF: .text:CM
                                        ; "mySub"
                      dd offset aMyfunc ; "myFunc"
                      dd 0
                      dd offset aMyprop ; "myProp"
                      dd offset aMyfunc2; "myFunc2"
```

The Meat

The `Object->ObjInfo` structure will also lead us to further information of interest. `ObjInfo->PrivateObject` will lead us to the main structure we are interested in for this blog post.

I could not find public documentation on this structure or those

below it. What follows is what I have put together through analysis.

Development of this information had me working across four different windows simultaneously.

- disassembly of the vb runtime
- disassembly of the target executable
- debugger stepping through the code and syncing the disasm views
- specialized tool to view and search vb6 structures

The following is my current definition of the `PrivateObj` structure:

```
CMYClass_PrivateObj dd 0                    ; nul1
                    dd offset CMYClass_ObjInfo; lpParentLink
                    dd 0FFFFFFFFh           ; unk1
                    dd 0                    ; nul2
                    dw 1                    ; cntPublicVars
                    dw 2                    ; cntEvents
                    dd 0                    ; nul3
                    dd offset CMYClass_Priv_FuncTypInf; lpFuncTypeInfo
                    dd 0                    ; nul4
                    dd offset CMYClass_Priv_PubVars; lpPublicVars
                    dd offset CMYClass_Priv_EvtTypInf; lpEventsTypeInfo
                    dd offset nullVal_3     ; lpNull
                    dd 0                    ; nul5
                    dd 0                    ; nul6
                    dd 0                    ; nul7
                    dd 4Ch                  ; unk3
                    dd 104h                 ; unk4
```

In the debugger, I saw vb runtime code accessing members within this structure to get to the vtable offsets for an Invoke call. I then compiled a number of source code variations while observing the effects on the structure. Some members are still unknown, but the primary fields of interest have been identified.

The `PrivateObj` structure leads us to arrays describing each code object's public function, variable, and event type information.

```
CMYClass_Priv_FuncTypInf dd offset CMYClass_PubFuncTypDesc_mySub
                         dd offset CMYClass_PubFuncTypDesc_myFunc
                         dd 0
                         dd offset CMYClass_PubFuncTypDesc_myProp
                         dd offset CMYClass_PubFuncTypDesc_myFunc2
```

Counts for the event and variable arrays are held directly within the `PrivateObj` itself. To get the count for the public functions, we have to reference the top-level `Object->ProcCount` field.

Note that there is a null pointer in the `FuncType` array above. This corresponds to a private function in the source code. If you look back, you will also see that same null entry was found in the `ProcNames` array listed earlier.

Each one of these type information structures is slightly different. First, we will look at what I am calling the `FuncTypDesc` structure.

I currently have it defined as follows:

```
CMYClass_PubFuncTypDesc_mySub_Sub db 8  ; argSize
                db 0                    ; bFlags
                dw 2Dh                  ; vOff
                dw 0FFFFh               ; constFFFF
                dw 0                    ; nul1
                dd 0                    ; optionalVals
                dd 60030002h            ; memberID
                dd offset off_4020C4    ; lpAryArgNames
                dd 0                    ; lpFuncDesc
                dd 0                    ; nul3
                dd 0                    ; helpID
                db 1Eh                  ; unk1
                db    8
                db  2Fh ; /
                db    0

off_4020C4      dd offset aArg1
                dd offset aV
```

The structure displayed above is for the following prototype:

```
Sub mySub(ByVal arg1 As Long, v As Variant)
```

This structure is where we start to get into the real meat of this article. The number of types defined for the method is held within the `argSize` field. The first three bits are only set for property types.

- 111 is a property Set
- 010 is a property Let

- 001 is a property Get (bFlags bit one will additionally be set)

  The type count will be the remaining bits divided by four.

  If the `bFlags` bit one is set, the last entry represents the method's return value. In all other scenarios, the types represent the arguments from one to arg count. VB6 supports a maximum of 59 user arguments to a method.

  An `argSize` of 0 is possible for a sub routine that takes no arguments and has no return value.

  The `vOff` field is the vtable offset for the member. The lowest bit is used as a flag in the runtime and cleared before use. Final adjusted values will all be 32-bit aligned.

  The `optionalVals` field will be set if the method has optional parameters which include default values.

  `LpAryArgNames` is a pointer to a string array. It is not null-terminated, so you should calculate the argument count before walking it.

  After the structure, we see several extra bytes. These represent the type information for the prototype. This buffer size is dynamic.

  A mapping of these values was determined by compiling variations and comparing output for changes. The following values have been identified.

```
'had to manually map these out watching variations
'internal to vb not variant types
Enum eVBInternal_VarTypes
    epvT_bool = 3
    epvT_byte = 5
    epvT_int = 6
    epvT_long = 8
    epvT_single = &HA
    epvT_double = &HB
    epvT_date = &HC
    epvT_currency = &HD
```

```
    epvT_variant = &HF
    epvT_String = &H10
    epvT_object = &H1B
    epvT_comIFace = &H1C  '\
    epvT_comobj = &H1D    ' \__ these add 32bit pointer
    epvT_internal = &H13  ' /
    epvT_hresult = &H1E
End Enum

Public Function initFromRawVal(b As Byte)

    '0x08 = 00001000  'long
    '0x28 = 00101000  'byref long
    '0x68 = 01101000  'array byref long     (0x40 -> 01000000)
    '0xA8 = 10101000  'optional byref long  (0x80 -> 10000000)

    Dim tmp As Byte

    Me.rawVal = b
    tmp = b

    If (tmp And &H80) = &H80 Then
        Me.isOptional = True
        tmp = tmp Xor &H80
    End If

    If (tmp And &H40) = &H40 Then
        Me.isAry = True
        tmp = tmp Xor &H40
    End If

    If (tmp And &H20) = &H20 Then
        Me.isByRef = True
        tmp = tmp Xor &H20
    End If

    Me.baseVal = tmp

End Function
```

The above code, shows that `0x20`, `0x40`, and `0x80` bits have been set to represent `ByRef`, `Array`, and `Optional` specifiers. These are combined with the base types identified in the enumeration. These do not align with the standard `VARIANT VARENUM` type values.

The comobj and internal flags are special cases that embed an additional 32-bit value after the type specifier byte.

This will link to the targets `ObjInfo` structure for internal objects.

If the `comobj` type is encountered, an offset to another structure will be found that specifies the objects library guid, clsid, library name, and dll path. We will show this structure later when we cover public variables.

It is important to note that these offsets appear to always be 32-bit aligned. This can introduce null padding between the type byte and the data offset. Null padding has also been observed between individual type bytes as well. Parsers will have to be tolerant of these variations.

Below is the type definition for the following prototype:

```
Function myFunc(myClass As Class1) As Byte
```

```
CMYClass_PubFuncTypDesc_myFunc db 8                     ; argSize
                               ; DATA XREF: .text:00402040↑o
                 db 1                   ; isFunc ; "myClass"
                 dw 2Dh                 ; vOff
                 dw 0FFFFh              ; const0FFFF
                 dd 0                   ; nul1
                 dw 0                   ; nul2
                 dd 60030002h           ; memberID
                 dd offset off_402004   ; lpAryArgNames
                 dd 0                   ; lpFuncDesc
                 dd 0                   ; nul3
                 dd 0                   ; helpID
                 db 1Eh                 ; unk
                 db  33h ; 3            ; epvT_internal 0x13 | 0x20 Byref
                 db    0                ; padding to 32bit align
                 db    0
                 dd offset Class1_ObjInfo ; pointer to internal class ObjInfo
                 db  25h ; %            ; ByRef Byte
                 db    0
                 db    0
                 db    0
```

Next up comes a quick look at the `PubVarDesc` structure. Here is a structure for the following prototype:

```
Public myPublicVar As Scripting.FileSystemObject
```

```
CMYClass_PubVarTypDesc_0 dd offset aMypublicvar  ; lpName
                 dd 0                   ; nul1 ; "myPublicVar"
                 dd 0                   ; nul2
                 dw 0                   ; index
                 db 3                   ; unk1
```

```
                db 40h                  ; unk2
                dw 2                    ; unk3
                dw 1Ch                  ; unk4
                dd 3Ch                  ; varOffset
                dd 1Dh
                dd offset structCOMObj

structCOMObj    dd offset ppLibGUID     ; DATA XREF: .text:00402088↑o
                dd offset CLSID         ; {2A0B9D10-4B87-11D3-A97A-00104B365C9F}

ppLibGUID       dd offset pLibGuid      ; DATA XREF: .text:structCOMObj↓o
                                        ; {420B2830-E718-11CF-893D-00A0C9054228}
                dd 0
                dd 1
                dd 0
                dd offset aCWindowsSyswow_0 ; "C:\\Windows\\SysWOW64\\scrrun.dll"
                dd offset aScripting    ; "Scripting"
```

This example shows flag `0x1D` to declare an external COM object type. An offset then follows this to a structure which defines the details of the COM object itself.

The value `0x3c` in the `varOffset` field is where the data for this variable is stored. For VB6 COM objects the `ObjPtr()` returns a structure where the first member is a pointer to the objects `Vtable`. The areas below that contain class instance data. Here `myPublicVar` would be found at `ObjPtr()+0x3c`.

Finally, we will look at an `EventDesc` structure for the following prototype:

```
Event MyEvent2(eventArg1 As Long, eventArg2 As Byte)
```

```
CMYClass_PubEventTypDesc_2 db 8         ; argSize
                db 0                    ; isFunc ;
                dd 0FFFFFFFFh           ; constFFFF
                dd 0                    ; nul1
                dw 0                    ; nul2
                dd 1                    ; memberID
                dd offset off_401FF4    ; lpAryArgNames
                dd 0                    ; lpFuncDesc
                dd 0                    ; nul3
                dd 0                    ; helpID
                db 0                    ;
                db  28h ; (
                db  25h ; %
                db    0

off_401FF4      dd offset aEventarg1    ; "eventArg1"
                dd offset aEventarg2    ; "eventArg2"
```

This structure closely follows the layout of the `PubFuncDesc` type. One thing to note, however, is that I have not currently been able to locate a link to the Event name strings embedded within the compiled binaries.

In practice, they are embedded after the strings of the `ProcNamesArray`.

Note that the class can raise these events to call back to the consumer. There is no implementation of an event routine within the code object that defines it.

Conclusion

In this post, we have detailed several structures which will allow analysts to parse the VB object structures and extract function prototypes for public object members.

This type information is included as part of the standard IDispatch plumbing for every user-generated VB6 form, class, user control, etc. This does not apply to functions in BAS code modules as they are not COM objects internally.

It is also interesting to note that VB6 embeds this type data right along with the other internal structures in the .text section of the PE file. No type library is required, and this feature can not be disabled.

While the structures portion of the .text section can contain various compiler-generated native code stubs, all user code falls into the memory range as defined by the `VBHeader.ProjectInfo.StartOfCode` and `EndOfCode` offsets.

The framework required to analyze a VB6 binary and resolve the information laid out in this posting can be fairly complex. Luckily

open-source implementations already exist to help ease the burden.

Structure and prototype extraction routines have already been included in the free vbdec disassembler. This tool can also generate IDC scripts to apply the appropriate structure definitions to a disassembly.

VB6 binaries are typically considered hard to analyze. Extracting internal function prototypes will be a very welcome addition for reverse engineers.