VB6 Variable Initialization and Tear Down

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

Introduction:

In this article we will be looking at how VB6 initializes and cleans up module level variables. This query into the inner operations of the Runtime opens the door to some interesting new capabilities for binary analysis.

This includes:

- enumerating all live VB6 classes
- discovery of private variables and types
- extracting data from live class instances
- gaining arbitrary script access to any live class

Background:

Every VB6 code module, regardless of type, can hold its own module level variables.

- If a class goes out of scope and is being destroyed, how does the VB Runtime assure that all of its resources are automatically cleaned up?

- If a module contains an array of predefined size, how does the Runtime allocate the memory so its ready for first time use?

For instanced code modules, such as Forms and Classes, module level variables are stored on a per instance basis. The value returned by ObjPtr() actually returns a data block that holds a multitude of settings such as reference count, VTable pointer, local variables etc.

For BAS modules, where there is no instancing, the variables are held at static offsets starting at `Object.aModulePublic`.

If we want to explore how resource cleanup occurs, we can hold an instance of one of our classes in a BAS level variable and place a MsgBox in the class terminate code. This gives us a convenient place to attach a debugger and examine the call stack.

Consider the following code:

```
'Module 1
Dim a As Class1, b As Object, c As Object

Sub Main()
    Set a = New Class1
    Set b = New Class1
    Set c = New Class1
    a.name = "dog"
    b.name = "cat"
    c.name = "bird"
    InputBox "", , Hex(ObjPtr(a)) & " " & Hex(ObjPtr(b)) & " " & Hex(ObjPtr(c))
End Sub

'Class 1 code
Public name As String

Private Sub Class_Terminate()
    MsgBox "terminating " & name
End Sub
```

Attaching a debugger at the class terminate MsgBox gives us the following call stack:

```
Address    Stack      Procedure / arguments
0019FA64   66028116   MSVBVM60.RUN_INSTMGR::ExecuteInitTerm
0019FA9C   66027F83   MSVBVM60.RcmResetModulesPrepass
0019FAB0   66027F04   MSVBVM60.RcmShutDownProject
0019FAC8   66027B3C   MSVBVM60.RcmResetProject
0019FD44   66051B59   MSVBVM60.66027AC0
0019FD58   66027A39   MSVBVM60.EbResetProjectNormal
0019FD70   66027D28   MSVBVM60.CThreadPool::ResetProject
0019FD8C   66051AEC   MSVBVM60.DbgReset
0019FDAC   6600B1B5   MSVBVM60.DbgResetIfDoneRunning
```

As we examine these routines we discover that VB6 keeps a linked list of all class instances. The run time data we received in form load showed the following ObjPtrs for the 3 class instances:

```
objptr 1 = 0x565280
objptr 2 = 0x565C88
objptr 3 = 0x565CE8
```

As we step through the code in `RcmResetModulesPrepass`, we find it referencing the following VB6 structure:

```
Class1.ObjInfo.lpProjectData = 402018
```

As we analyze the code, we find that this offset is a pointer to a `RUN_INSTMGR` structure

```
00402018  005650C8  *RUN_INSTMGR

005650C8  00565CE8  <-- objptr 3
005650CC  00000000
005650D0  004011E8  Class1.objinfo
005650D4  00000003  <-- live instances count
```

Following the ObjPtr() to the next live instance, we see a linked list emerge at ObjPtr() +4 to the next instance

```
00565CE8  00402348  Project1.00402348  class vtable
00565CEC  00565C88  <-- objptr 2

   00565C88  00402348  Project1.00402348 class vtable
   00565C8C  00565280  <-- objptr 1

       00565280  00402348  Project1.00402348 class vtable
       00565284  00000000  <-- no next instance
```

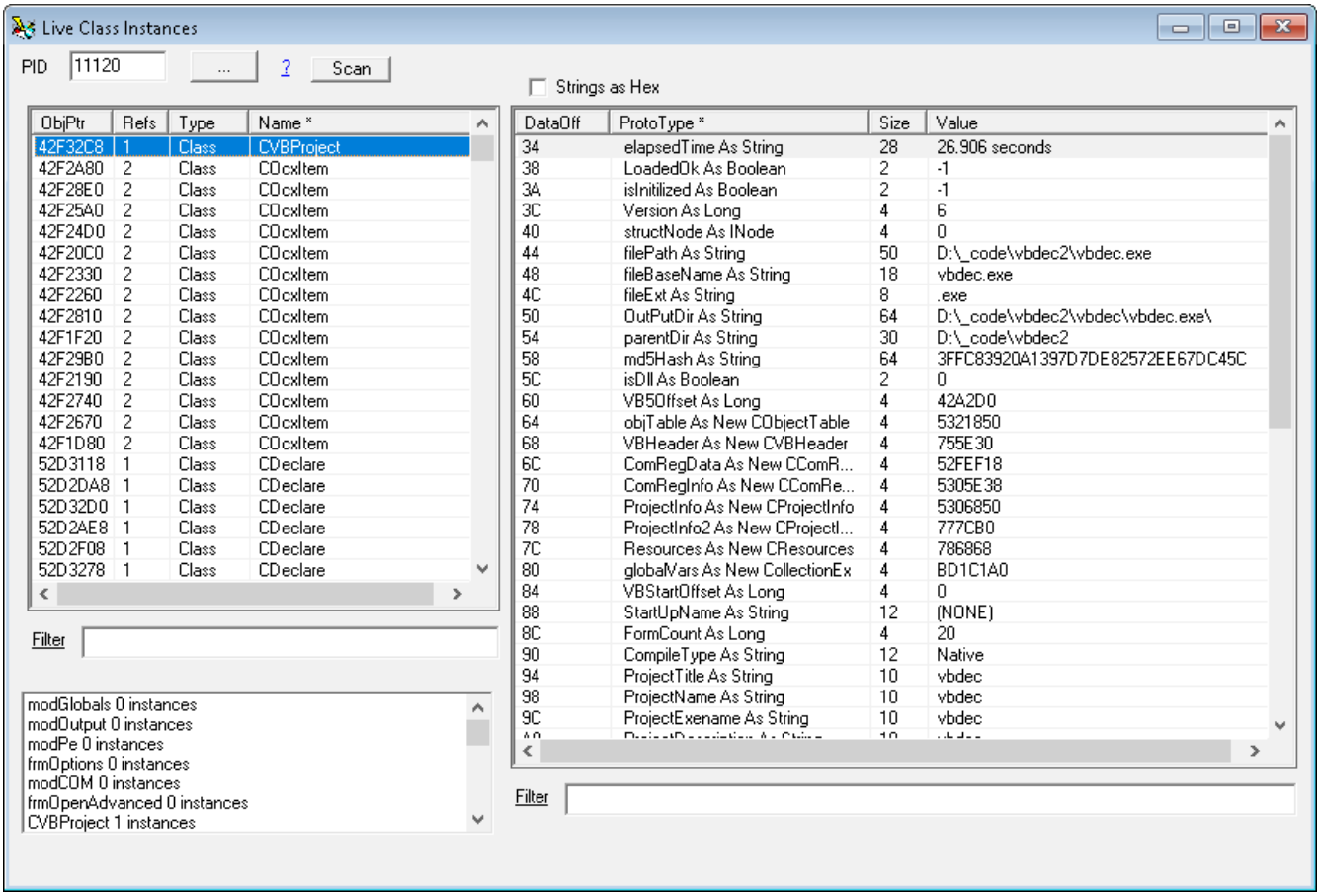
This is an interesting bit of trivia, but is it really useful? Well its more interesting than you might think!

Implications:

At this point, we can now scan a running a process and enumerate all of its live class instances. What would make this really interesting is if we knew what local variables were held at the class level, their offset, and type. This would allow us to remotely view every classes configuration state dumping live data without even attaching a debugger. Is that possible?

Yuuuup.

If you have been following along with my VB internals series there is an article titled Recovery of function prototypes in Visual Basic 6 executables. In this paper we detail how to parse the IDispatch type information to get public variable names, types, and offsets. When you combine these two techniques, you now get a pretty interesting information dumper that can be implemented using only ReadProcessMemory.



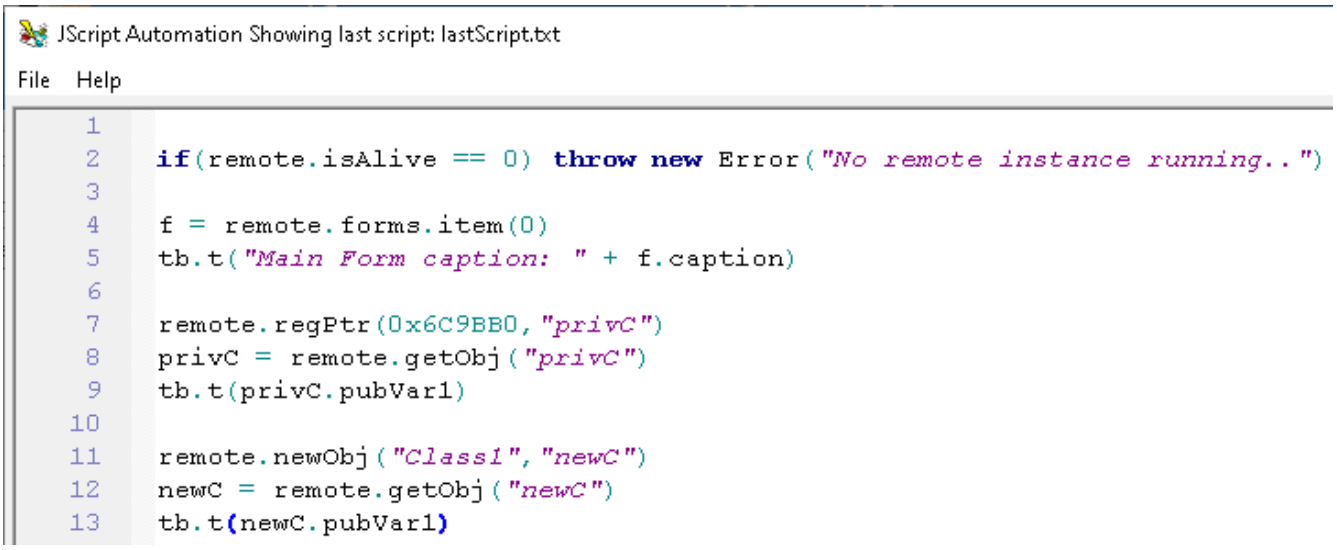
This new form has already been added to Vbdec and is available from the `Explore --> Live Class Instances` menu item.

Ok, neat trick. With a live class instance pointer what else can we do? Can we call methods on these classes from the ObjPtr ?

Why yes we can. Here I will refer you to a previous paper titled Scripting Arbitrary VB6 Applications. Building on our previous work we are now able to access any live class instance through its ObjPtr.

```
int HandleIPCMsg(char* m_msg){

    vector<string> cmd = split(m_msg, ":");

    if(cmd[0] == "regptr"){ //regptr:hex_objptr:ROTNAME
        if(cmd.size()!=3) return -1;

        objPtr = strtoul(cmd[1].c_str(), NULL, 16);
        if(objPtr < 0x100) return -2;

        wstring rotName(A2W(cmd[2].c_str()));
        IUnknown *unk = (IUnknown*)objPtr;

        if( CreateFileMoniker(rotName.c_str(), &mon) == S_OK){
            hr = rot->Register(ROTFLAGS_REGISTRATIONKEEPSALIVE, unk, mon, &appRotToken);
            if(hr == S_OK) rotTokens.push_back(appRotToken);
```

Once openscript is active by launching a process through `Explore --> Script Remote Application` these new features can be accessed using the new “remote” object in the vbdec `Script Automation` form.



Here again, our knowledge keeps compounding upon itself allowing us to do new and magical things.

Going back:

Ok, so the live class instances list is interesting (and a bit unexpected) but there are still questions about that whole object tear down process. We know how the VB Runtime does its internal class cleanup. How does it handle the cleanup of other resources such as external COM objects?

For this query, I added a VB Collection object to our test module, and set a breakpoint on its `VBACollection::Release` method in the Vb Runtime. After some probing of the call stack I came across the following function:

```
.text:66061E42 void RESDESCTBL::DestructItem(
   RESDESCTBL *a1,
   void *a2,
   struct RESDESC *a3,
   unsigned int *a4
)
```

This function is pretty interesting. It reveals that the VB Runtime knows about private variables offsets and types to enable clean tear down. Where is this information held and what does it represent?

As we start analyzing this function, we find that the `RESDESCTBL` pointer corresponds to the `Object.aPublicBytes` field. Let us consider the following code in a module:

```
Private a As String
Private b As Long
Private c As New Collection

Sub Main()
    a = "test"
    b = 3
    c.Add a
End Sub
```

```
4017B8  008   aPublicBytes     401914
4017C0  010   aModulePublic    402024 - data section

00402024 (+0)  004FF90C  UNICODE "test"
00402028 (+4)  00000003
0040202C (+8)  004FF930 -> vtable for collection object

.text:00401914                dw 14h |--;total data size
.text:00401916                dd 10h |  ;size in mem of data block
.text:0040191A                dd 2   |  ;entries
.text:0040191E                dw 0   |__;unknown
.text:00401920                dw 0   ;data offset 1
.text:00401922                dw 1   ;type id = string
.text:00401924                dw 8   ;data offset 2
.text:00401926                dw 3   ;type id = object
```

Since this code is in a non-instanced BAS module, the compiler had to define a data area for variable storage. This is where the `Object.aModulePublic` field comes into play. The `Object.aPublicBytes` field then points to a `RESDESCTBL` structure that defines the variables for the code unit.

Looking at our sample data we have 3 private variables, but only two entries in our `RESDESCTBL`. This is because of how this data is intended to be used. The `Private b as long` is not visible in this table because it does not require any cleanup on termination.

This table can reveal basic type information and data offsets for private variables, but unfortunately it is not as rich as the IDispatch information we extracted before. As we start to probe the format of this type information we find that it aware of Variants, Strings, Arrays, Objects and UDT types (*if the UDT contains a subtype requiring cleanup)

Here is a closer look at my current working definition for the header structure:

```
Private Type tResDescTbl
      size As Integer
      memDataSz As Integer  'end marker for class data block = ABABABAB
      reqAlloc As Integer
      entries As Long
      unk1 As Integer
End Type
```

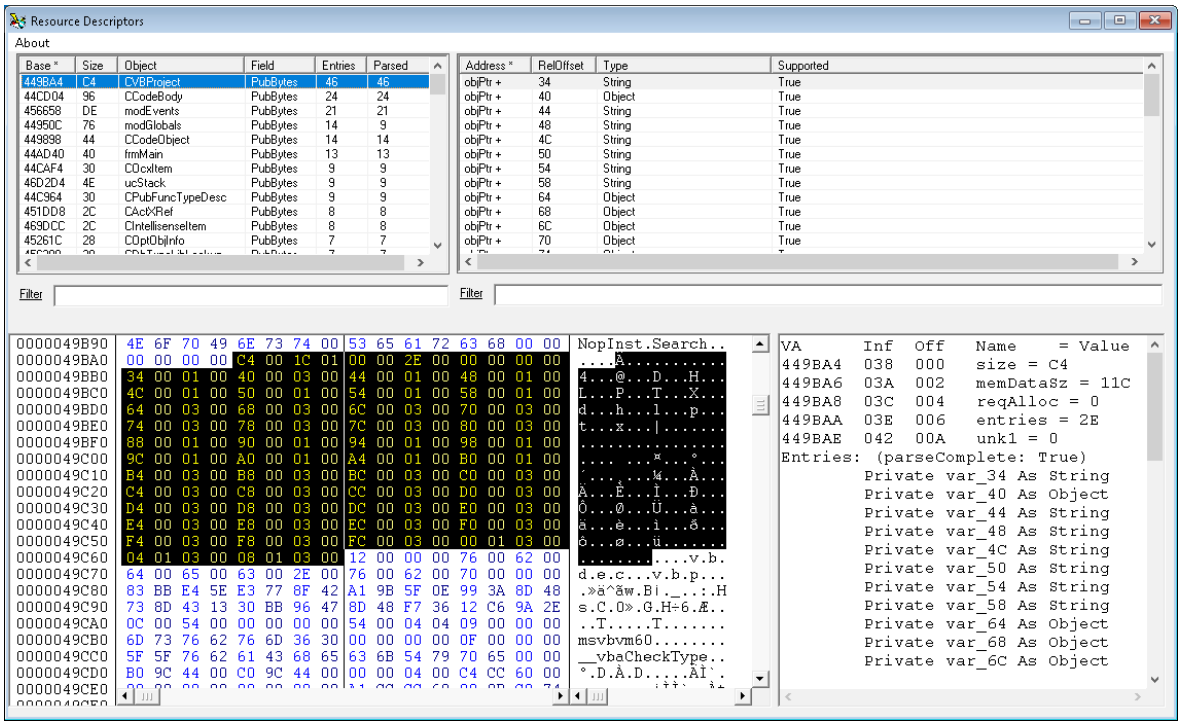
The type data that follows this header contains dynamic field types and can get rather complex.

We also find that this table gets used during code initialization through `RESDESCTBL::ConstructItem`. This is how the Runtime can ensure that pre-dimensioned arrays are ready for first use. The following data defines an array for pre-execution initialization

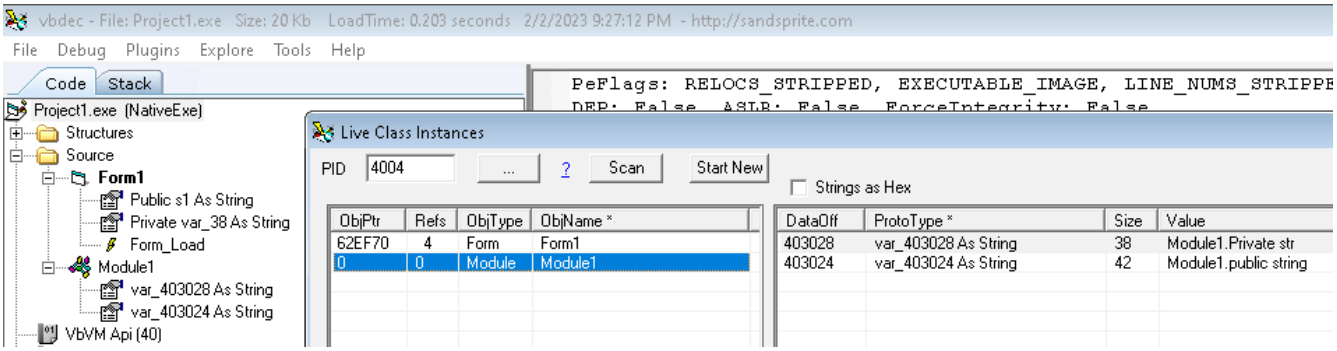
```
Dim x(&H11223344 To &H22334455) As Integer '11111112 elements
```

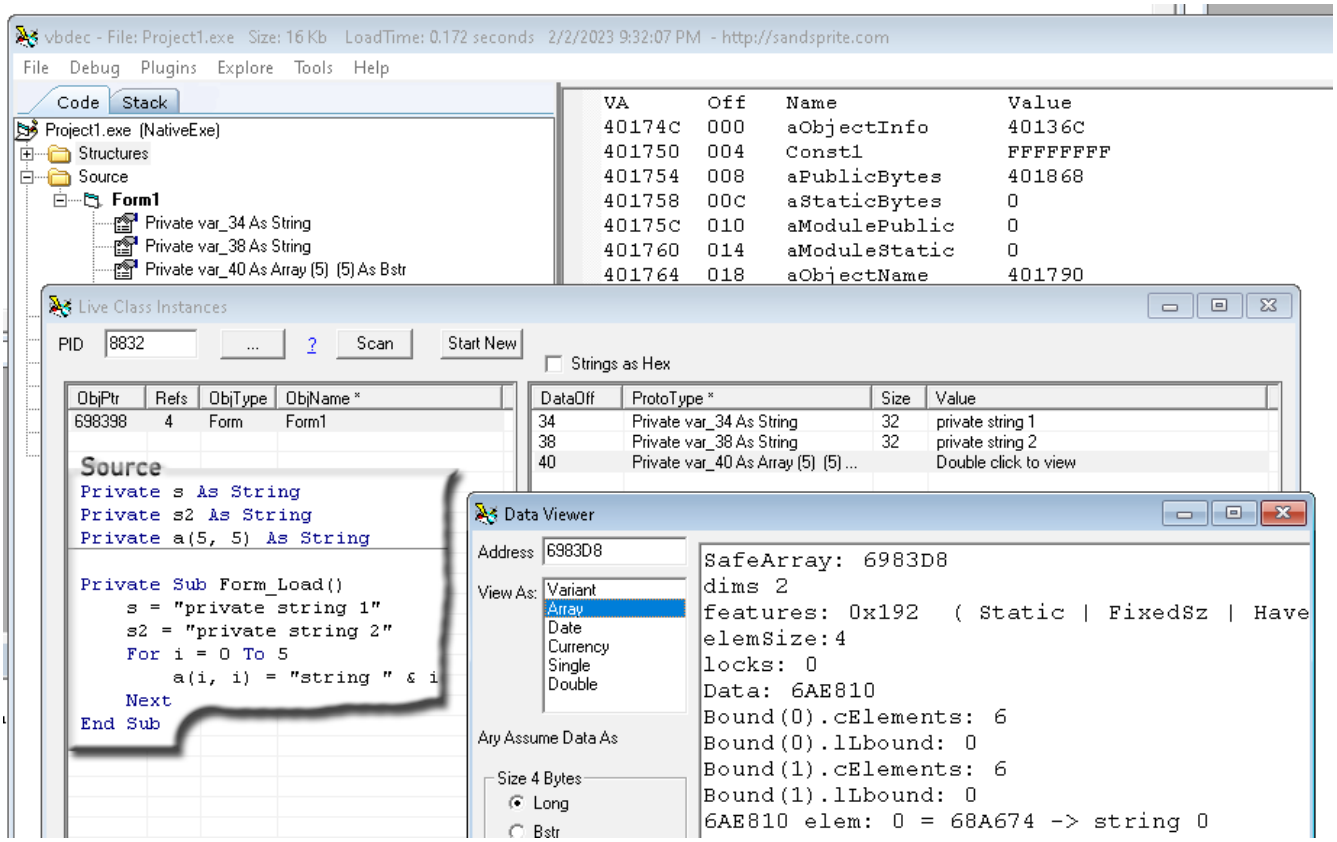
```
.text:0040165C                  dw 38h
.text:0040165E                  dw 20h
.text:00401660                  dw 1  ; 660625AD cmp [esi+6], ax
.text:00401662                  dw 1
.text:00401664                  dw 0
.text:00401666                  dw 0  ; RescDesctbl ends here
.text:00401668                  dw 4  ; data offset - 660625B1 lea edi, [esi+0Ch]
.text:0040166A                  dw 5  ; ary flag 6605606D MOV AX,WORD PTR DS:[ESI+2]
.text:0040166C                  dw 0  ; array pre-def struct
.text:0040166E                  dw 0
.text:00401670                  dw 0  ; 660560F2  TEST BYTE PTR DS:[ECX+8],60
.text:00401672                  dw 0
.text:00401674                  dw 0
.text:00401676                  dw 0  end of struct 2 (size based on 401670 flag)
.text:00401678                  dw 1  raw safe array struct - dims
.text:0040167A                  dw 92 features FADF_HAVEVARTYPE | FIXEDSIZE | STATIC
.text:0040167C                  dd 2  element size (would be 4 for long)
.text:00401680                  dd 0  locks
.text:00401684                  dd 0  pvdata
.text:00401688                  dd 11111112h ; SafeArrayBound.cElements
.text:0040168C                  dd 11223344h ; SafeArrayBound.lbound
.text:00401690                  dw 2 VT_I2 (if long value is 3 - VT_I4)
```

To gain quick visibility into these structures a new form has been added to vbdec under the `Explore --> Resource Descriptors` menu item.



While not every type of entry is fully supported yet, results so far are pretty good. With this extra information we are now able to start discovering private variables. This information can then be bubbled back into the main UI as well in the `Live Class Instances` form.

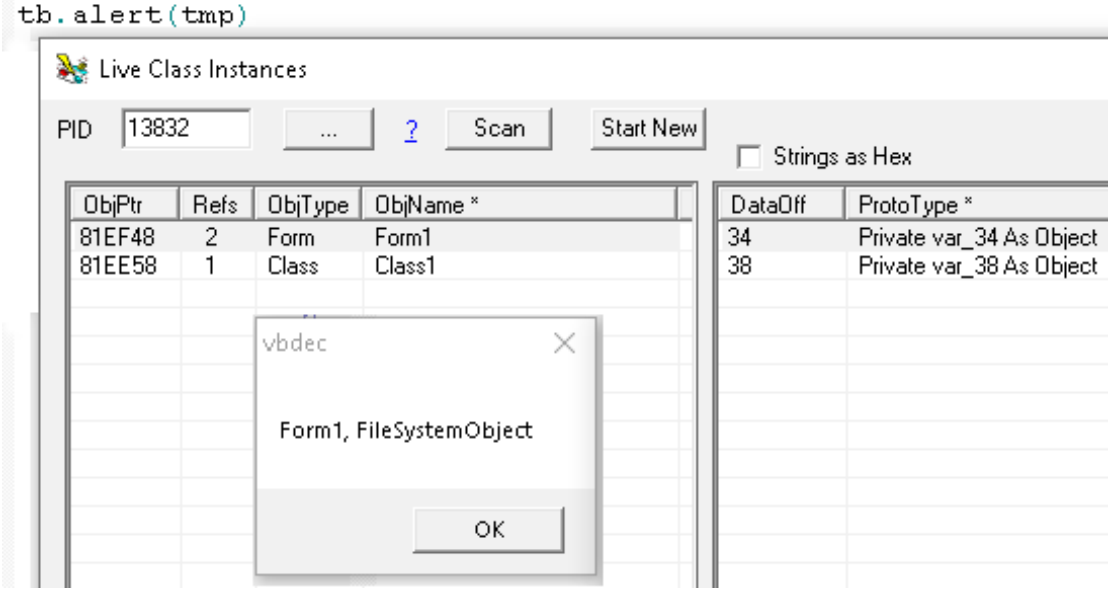




Not as comprehensive as the other data, but still a welcome addition. The next step is to start scanning the disasm to infer more types and data offsets. Knowing about the existence and basic type of private variables could also be coupled with more invasive run time data collection.

I am currently experimenting with a technique to call `rtcTypeName` through openscript to query object types dynamically. This many also be possible through ReadProcessMemory calls for internal objects.

```
tmp = remote.typName(0x81EF48)
p = remote.readLong(0x81EF48 + 0x38);
tmp += ", " + remote.typName(p);
tb.alert(tmp)
```



Conclusion:

In this article we discovered how the VB6 Runtime tracks live class instances and certain types of private variables for initialization and cleanup.

This allowed us to enumerate all live class instances in a running process and dump known data offsets for external viewing.

Building on previous work, the revealed class ObjPtr() also allowed us to remotely script public functions on any arbitrary class instance.

The ability to call private class functions is now also within our reach. This same technique will also allow us to call arbitrary module functions remotely as well. The largest barrier at this point will be determining prototypes for the private functions.

While the type information held within the `RESDESCTBL` is not as rich as the IDispatch information, it is still a welcome addition and reveals new information to us. We may also be able to further refine the data it gives us through probing the live process.

At this point we still have gaps and more to explore but are making good progress none the less.

Our path has proven quite sinuous, but also very interesting. It appears that I am hooked on the intrigue and discovery.