Scripting Arbitrary VB6 Applications

Author: David Zimmer <dzzie@yahoo.com> Site: http://sandsprite.com Date: 12.7.22

Introduction:

While doing malware analysis it is often required to interact with running code to discover how it operates. This can be done through techniques such as API hooking, debugging, system monitoring etc.

Some tasks may even require the extraction, reconstitution, or reuse of malware code in order to perform a certain duty. This is common for code such as domain name generation and decryption routines.

Anytime there is an easy way to reuse existing functionality, my interest is piqued. The research presented in this post gives us another powerful mechanism of code reuse.

The technique we will be detailing today is a method in which we can grant ourselves scripting access to any existing Visual Basic 6 (VB6) executable. This technique does not require any source code modifications.

Once implemented, we can access any loaded forms, embedded controls, and public members. This includes access to any class hierarchies held as form level public instances.

With some additional work, any live class instance in the program could be accessed in a similar manner. Manually creating new instances of internal classes has also proven possible.

Similar techniques have been explored in the past. What follows is my run at the problem set.

Background:

All VB6 objects are COM Objects which support the IDispatch interface. This is a core feature of the language and allows for them to be trivially utilized by scripting engines. Since it is so easy, it is common for developers to internally add scripting support to their applications for automation purposes.

To develop a component that allows for external automation, Visual Basic 6 supports the ActiveX Dll and ActiveX Exe project types. If you have ever used a script which calls CreateObject(), you have already worked with ActiveX COM Objects. Support for COM is integrated deeply into the Windows operating system and entails many powerful features.

Knowing that all VB6 COM objects are inherently scriptable, can this functionality be enabled for random executables without source code access?

Probing the runtime:

In the exploration of this question, I started examining the runtime to see how I might easily extract live object references and manually implement scriptable access.

The first place to look, with the largest impact, is the Forms object.

This object holds a reference to the loaded forms for each VB component. Once we have access to an individual form, we then would also have access to all of its embedded controls and public members. This would be a significant first step and where we begin our journey.

If we look at a native executable which uses the global Forms object we will see the following code generated:

```
mov eax, objRef_dataSect
xor edi, edi
cmp eax, edi
jnz short already_live
push offset objRef_dataSect
push offset comdefStruct
call ds:__vbaNew2

ComdefStruct
  dd 2
  dd offset dword_401A90 ;{FCFB3D23-A0FA-1068-A738-08002B3371B5}
  dd offset dword_401AA0 ;{FCFB3D22-A0FA-1068-A738-08002B3371B5}
  dd 0
```

This is easy enough to replicate and should give us a reference to the global Forms object on demand. (Note each VB component gets its own “Global” object. An ActiveX Dll can not access the main executables forms unless explicitly passed a reference)

Before we can test this theory, we first need a method to execute our own code inside the host VB6 process. Dll injection is the obvious answer, however it is not quite as straight forward as you might think.

There are two factors that must be considered when trying to run injection code in a VB6 process.

The first is runtime initialization. A certain number of internal steps must occur before you call any runtime functions. In our previous paper Binary Reuse of VB6 PCode Functions, this is what the call to `CreateIExprSrvObj` accomplished. We could inject into a fully loaded process, however that has some side effects which we will discuss later on. For our purposes we will require injection at process startup.

The second thing we must consider is that all VB6 executables use the single threaded apartment model (STA). To use runtime functions, we need to work within the main VB6 thread. Working from other threads will crash the process as the runtime tries to access Thread Local Storage (TLS) members it expects to exist. While people have devised ways to work in multiple threads with VB6, we will avoid it here.

Our initial criteria are:

- must inject at process startup
- can not call runtime functions until initialized
- must only call runtime functions from main VB6 thread

The first experiment was to inject at startup and then place a hook on the user32.BeginPaint API. This hook is called from the main VB thread during form creation. At this point the runtime is fully initialized and ready for use. I created a quick mockup and gave it a shot.

Here we encounter our first road bump. Apparently vbaNew2 can not actually create an instance of this object and throws a Class not registered error. This is strange because we are literally using code lifted from the compiler itself.

To validate this result I first confirmed my use of the vbaNew2 function with other known good data and then tested it while running from within the hook. Both worked.

I then took a closer look at the sample VB application and found a little surprise. In the disassembly above, we see it check to see if a cached object reference is already alive. If so, then creation is skipped.

The surprise is, even on its very first use in Form_Load, the object reference is already set. The vbaNew2 call is never used and merely a compiler artifact. In fact forcing the call to vbaNew2 leads to the same Class not registered error.

The `Forms` object reference is being set before any user code executes. This indicates it must be a special case set by the runtime somewhere. A hardware breakpoint on the `objRef_dataSect` address quickly confirms this.

The VB runtime has an internal function named `TipRegAppObject`. This function will scan the `VBHeader.ProjectInfo.ExternalTable` looking for an entry of type 6 with a matching CLSID. If this is found, it will manually set the object instance address.

```
VBHeader.ProjectInfo.ExternalTable = 401258 ;array of 8 byte structs

dword_401258 dd 6
             dd offset off_401B24

off_401B24 dd offset 401A90 ;{FCFB3D23-A0FA-1068-A738-08002B3371B5}
           dd offset objRef_dataSect
```

If the application developer never used the `global Forms` class, `TipRegAppObject` has nothing to do and no reference will be set.

We can not formally create the class we want, If the developer did not use it, then it will not be cached anywhere that is easy to grab. We could search for another way to try to find it, but we already have a universal spot where it is guaranteed to appear.

At this point I decided to hook an internal runtime function. Hooking a hardcoded offset has some downsides like locking us to a specific dll version. For a research tool, this is an acceptable tradeoff.

After some more analysis, I decided to hook one layer above `TipRegAppObject` at `CVBApplication::Init(CVBApplication *this)`.

This gives us access to the full `CVBApplication` object. This includes access to more internal classes and a reference to the executable's `VBHeader` structure.

Implementation:

The initial injection routine now sets two hooks. One on `CVBApplication::Init` to grab a reference to internal runtime classes. The second on `BeginPaint` which allows us to trigger in the main VB thread once initialization is complete.

The next thing to consider is how to execute our final payload within the main VB6 thread. One technique is to simply load our own ActiveX dll into the process passing a reference to the main components Forms object. This is basically forcing a plugin model into the application. This should work fine however it is not my first choice.

My primary target is to gain some kind of remote scripting access. This is where the magic of the Running Object Table (ROT) comes into play. The ROT is how ActiveX exes register themselves as available for use with GetObject().

The COM object lives in one executable running as a server. External clients then connect and use it from another process.

While experimenting with manual ROT registration, I noticed that it worked for form objects, but would not allow access to internal classes.

```
Error: 0x62 - A property or method call cannot include a reference to
a private object, either as an argument or as a return value
```

We know that the core implementation of the class fully supports scripting. This is not a problem when using an internal script engine or a plugin model. What difference is there between an ActiveX Exe class, and a standard executable class?

Comparing the two in a structure viewer, we find that the Object.ObjectType of ActiveX Exe classes have bit 0x800 set. If we patch this on disk, we now gain full access to all standard classes. The ObjectType can also be patched in memory, but it must be patched before the class is created to take effect. This is why we must inject at process creation.

Our `CVBApplication::Init` hook is a perfect time to do this since we already have the reference to the `VBHeader` structure and no class instances have been created yet. Our complete `CVBApplication_Init` hook looks like the following:

```
unsigned int __declspec(naked) My_CVBApplication_Init(void *_this){

	_asm{
		//int 3

		//this hook triggers during runtime initialization
		//this is called once for every vb component loaded,
		//each get their own CVBApplication object and forms obj
		//currently we only track the first one for the main exe

		mov eax, vbApp
		cmp eax, 0
		jnz notFirst

		mov vbApp, ecx
```

```
            pushf
            pushad
            call MakeClassesPublic
            popad
            popf

notFirst:
            jmp Real_CVBApplication_Init
      }
}
```

The complete BeginPaint hook is below:

```
HDC __stdcall My_BeginPaint(HWND hWnd, LPPAINTSTRUCT lpPaint){

      //now running in main VB thread, this is hwnd of the main window

       IMoniker *mon = 0;
      char wndClass[256] = {0};

      int sz = GetClassName(hWnd, &wndClass[0], 255);
       HDC ret =  Real_BeginPaint(hWnd,lpPaint);
       msgf("BeginPaint(h=%x) class: %s", hWnd, wndClass);

      //sometimes CompatDesktopWindowReplacement hits first - ignore
      if(strcmp(wndClass, "ThunderRT6FormDC") !=0 &&
         strcmp(wndClass, "ThunderRT6MDIForm") !=0 ) return ret;

      int disabled = DisableHook((ULONG_PTR)Real_BeginPaint);
      msgf("vbApp=0x%x hookDisabled=%d", (int)vbApp, disabled);

      if(prevWndProc == 0){

            HMENU h = GetSystemMenu(hWnd, 0);
             AppendMenu(h, MF_STRING, IDM_MYACTION, "Gnarfle the Garthok");

             prevWndProc = (WNDPROC)SetWindowLongPtr(hWnd, GWL_WNDPROC,
                                    (LONG_PTR)&myNewWndProc);
             msgf("IPC listening on hwnd %x", hWnd);

            if(GetRunningObjectTable(0, &rot) == S_OK){
                  if( CreateFileMoniker(L"remote.forms", &mon) == S_OK){
                        IDispatch *IDisp = (IDispatch*)pPlus(vbApp,0x24);
                        HRESULT hr = rot->Register(
                              ROTFLAGS_REGISTRATIONKEEPSALIVE,
                              IDisp, mon, &appRotToken
                        );
                        if(hr == S_OK){
                              msgf("registered remote.forms in ROT");
                        }else{
                              msgf("ROT registration failed %x", hr);|
                        }
                        mon->Release();
                  }
            }
      }
      return ret;
}
```

In this code we first disable our hook so it will not trigger again. We only needed an initial foot hold in the main VB thread.

For continued access, we register a new System Menu item and subclass the main window.

This allows us to manually trigger new features we might want in the future. With the window subclass in place, I also implemented a basic InterProcess Communications (IPC) server for programmatic access. These additions are not required but are useful for future exploration.

Finally we register the main executable's `global Forms` object in the ROT as "`remote.forms`".

With this complete everything is now up and running!

A Windows Script Host (WSH) Javascript that interacts with our test application is shown below:

```
var o = GetObject("remote.forms")
var f = o.Item(0)
f.List1.Clear()
WScript.Echo(f.text1.text)
f.text1.text = "test from js"
WScript.echo(f.formMeth("remote hi").toString(16))
WScript.echo(f.pubClass.classMeth("remote class hi").toString(16))
```

Even Python can be used:

```
import os, sys
import win32com.client

forms = win32com.client.GetObject("remote.forms")

for form in forms:
     print form.Name
     for c in form.Controls:
          print " " + c.Name
```

Conclusion:

In this post we have covered how to make any VB6 application remotely scriptable using built in features of the language and operating system.

While we did encounter several snags along the way, everything turned out to be manageable and resulted in a successful trial.

For a brief recap of events:

- Target VB6 process is started with an injection dll
- New thread hooks `CVBApplication_Init` and `BeginPaint`
- `CVBApplication_Init` hook:
    - Stores a reference to internal VB objects during runtime initialization
    - Walks `VBHeader.ProjectInfo.ObjectTable` setting all classes public

- `BeginPaint hook:`
  - Runs from main VB6 thread
  - Adds a system menu item and subclasses main window for IPC (optional)
  - Registers internal `Forms` object in the ROT
- IPC allows for bidirectional communications between injector and target process.

Our first inroads are based on the forms collection for simplicity and depth of impact. This technique can be applied to any COM object instance.

In addition to the forms collection, access to other internal objects can be intercepted with hooks on various runtime API such as vbaNew, vbaNew2, vbaFreeObj etc.

Some thoughts on future innovations of this technique:

- track new class instances and add them to an exposed VB Collection in ROT
- injector keeps a visual manager of live objects (using IPC callbacks)
- injector implements its own script host for integration
- Ability to stall host process while accessing transitory objects
- artificially increment reference counts to keep instances alive.
- Trigger arbitrary class/form creation using IPC and vbaNew (already tested)

Full source code for this research is available for download.This includes an injector, dll, test app, and demonstration scripts.

The code is being released as a proof of concept work. Adaptations may be required for specific targets. This research is just the beginning of what is possible with this technique.