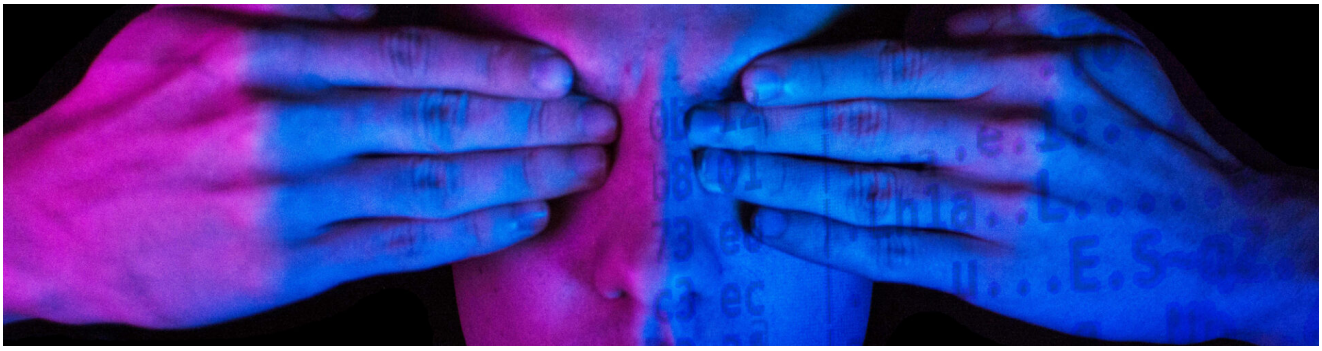
Binary Data Hiding in VB6 Executables

decoded.avast.io/davidzimmer/binary-data-hiding-in-vb6-executables

April 22, 2021



Overview

This is part one in a series of posts that focus on understanding `Visual Basic 6.0 (VB6)` code, and the tactics and techniques both malware authors and researchers use around it.

Abstract

This document is a running tally covering many of the various ways `VB6` malware can embed binary data within an executable.

There are 4 main categories:

- string based encodings
- data hidden within the actual opcodes of the program
- data hidden within parts of the `VB6` file format
- data in or around normal `PE` structures

Originally I was only going to cover data hidden within the file format itself but for the sake of documentation I decided it is worth covering them all.

Data held within the file format is a special case which I find the most interesting. This is because it can be interspersed within a complex set of undocumented structures which would require advanced knowledge and intricate parsing to detect. In this scenario it would be hard to determine where the data is coming from or to even recognize that these buffers exist.

Resource Data

The first technique is the standard built into the language itself, namely loading data from the resource section. `VB6` comes with an add-in that allows users to add a `.RES` file to the project. This file gets compiled into the resource section of the executable and allows for

binary data to be easily loaded.

```
Dim b() As Byte
b() = LoadResData(101, "CUSTOM")
```

This is a well known and standard technique.

Appended Data

This technique is very old and has been used from all manner of programming language. It will be mentioned again for thoroughness and to link to a public implementation [1] that allows for simplified use.

```
Dim b() As Byte
Set cEmb = New CEmbeddedFiles
cEmb.Initialize App.hInstance
For i = 0 To cEmb.FilesCount - 1
    b() = cEmb.FileData(cEmb.FileName(i))
```

Hex String Buffers

It is very common for malware to build up a string of hex characters that are later converted back to binary data. Conversion commonly includes various text manipulations such as decryption or stripping junk character sequences. Extra character sequences are commonly used to prevent automatic recognition of the data as a hex string by AV.

In the context of VB6, there are several limitations. The IDE only allows for a total of `1023` characters to be on a single line. VB's line continuation syntax of `&_` is also limited to only `25` lines. For these reasons you will often see large blocks of data embedded in the following format:

```
Dim x as String
x = "9090eb15"
x = x & "CCCCCCCC"
```

In a compiled binary each string fragment is held as an individual chunk which is easily identifiable. A faster variant may hold each element in a string array so conglomeration only occurs once.

This is a well known and standard technique. It is commonly found in `VBA`, `VB6` and malware written in many other languages. Line length limitations can not be bypassed through command line compilation.

Binary Data Within Images

There are multiple ways to embed lossless data into image formats. The most common will be to embed the data directly within the structure of a `BITMAP` image. Bitmaps can be held directly within `VB6` Image and Picture controls. Data embedded in this manner will be held in the `.FRX` form resource file before compilation. Once compiled it will be held in a binary property field for the target form element. Images created like this can be generated with a special tool, and then embedded directly into the form using the IDE.

The following is a public sample[2] of data being extracted from such a bitmap

```
Sub BMP2Array(bmp As IPicture, Data() As Byte)
   Dim bi As BITMAPINFO, l As Long
   bi.bmiHeader.biSize = Len(bi.bmiHeader)
   GetDIBits Form1.hDC, bmp.handle, 0, 0, ByVal 0, bi, 0
   ReDim Data(bi.bmiHeader.biWidth * Abs(bi.bmiHeader.biHeight) * 3 - 1)
   bi.bmiHeader.biBitCount = 24
   bi.bmiHeader.biCompression = 0
   bi.bmiHeader.biSizeImage = 0
   GetDIBits Form1.hDC, bmp.handle, 0, Abs(bi.bmiHeader.biHeight), Data(0), bi, 0
   CopyMemory l, Data(0), 4
   CopyMemory Data(0), Data(4), l
   ReDim Preserve Data(l - 1)
End Sub
```

Extracted images will display as a series of colored blocks and pixels of various colors. Note that this is not stenography.

Many tools understand how to extract embedded images from binary files. Since the image data still contains the `BITMAP` header, parsing of the `VB6` file format itself is not necessary. This technique is public and in common use. The data is often decrypted after it is extracted.

Chr Strings

Similar to obfuscations found in C malware, strings can be built up at runtime based on individual byte values. A common example may look like the following:

```
a = Chr(&H90) & Chr(&HEB) & Chr(&H15)
```

At the asm level, this serves to break up each byte value and puts it inline with a bunch of opcodes preventing automatic detection or display with strings. For native VB6 code it will look like the following:

```
4019AF 8D 55 9C           lea edx, [ebp+var_64]
4019B2 6A 15              push 15h
4019B4 52                 push edx
4019B5 FF D7              call edi ; Chr()
4019B7 8B 3D 58 10 40 00  mov edi, ds:vbaVarCat
```

In P-Code it will look like the following:

```
4017B4 F5 90 00 00 00  LitI4 0x90
4017B9 04 68 FF        FLdRfVar var_98
4017BC 0A 00 00 08 00  ImpAdCallFPR4 rtcVarBstrFromAnsi
4017C1 04 68 FF        FLdRfVar var_98
4017C4 F5 EB 00 00 00  LitI4 0xEB
4017C9 04 58 FF        FLdRfVar var_A8
4017CC 0A 00 00 08 00  ImpAdCallFPR4 rtcVarBstrFromAnsi
4017D1 04 58 FF        FLdRfVar var_A8
4017D4 FB EF 48 FF     ConcatVar var_B8
```

This is a well known and standard technique. It is commonly found in VBA as well as VB6 malware.

Numeric Arrays

Numeric arrays are a fairly standard technique in malware that are used to break up the binary data amongst the programs opcodes. This is similar to the Chr technique but can hold data in a more compact format. The most common data types used for this technique are `4`

`byte longs`, and `8 byte currency` types. The main advantage of this technique is that the data can be easily manipulated with math to decrypt it on the fly.

```
Dim x(1) as Long
x(0) = 1953719668
x(1) = 828862057
```

Native:

```
40198F FF 15 30 10 40 00     call ds:vbaAryConstruct2
401995 8B 4D E0              mov ecx, [ebp+var_20]
401998 C7 01 74 65 73 74     mov dword ptr [ecx], 74736574h
40199E 8B 55 E0              mov edx, [ebp+var_20]
4019A1 C7 42 04 69 6E 67 31  mov dword ptr [edx+4], 31676E69h
```

P-Code:

```
40179C F5 74 65 73 74  LitI4 0x74736574
4017A1 F5 00 00 00 00  LitI4 0x0
4017A6 04 64 FF        FLdRfVar var_9C
4017A9 A3              Ary1StI4
4017AA F5 69 6E 67 31  LitI4 0x31676E69
4017AF F5 01 00 00 00  LitI4 0x1
4017B4 04 64 FF        FLdRfVar var_9C
4017B7 A3              Ary1StI4
```

```
Dim x(0) as Currency
x(0) = 355993542966400.754@ 'Note type specifier is required
```

Native:

```
40198F FF 15 30 10 40 00     call ds:vbaAryConstruct2
401995 8B 4D E0              mov ecx, [ebp+var_20]
401998 C7 01 74 65 73 74     mov dword ptr [ecx], 74736574h
40199E 8B 55 E0              mov edx, [ebp+var_20]
4019A1 C7 42 04 69 6E 67 31  mov dword ptr [edx+4], 31676E69h
```

P-Code:

```
40179C F6 74 65 73 74 69 6E 67 31  LitCy 355993542966400.754
4017A5 F5 00 00 00 00              LitI4 0x0
4017AA 04 64 FF                    FLdRfVar var_9C
4017AD A4                          Ary1StCy
```

This technique is not as popular as the others, but does have a long history of use. I think the first place I saw it was in Flash ActionScript exploits.

Form Properties

Forms and embedded GUI elements can contain compiled in data as part of their properties. The most common attributes used are `Form.Caption`, `Textbox.Text`, and any element's. Tag property.

Since all of these properties are typically entered via the IDE, they are usually found to contain ASCII only data that is later decoded to binary.

Developers can however embed binary data directly into these properties using several techniques.

While there is way to hexedit raw data in the `.FRX` form resource file, this comes with limitations such as not being able to handle embedded nulls. Another solution is inserting the data post compilation. With this technique a large buffer is reserved consisting of ASCII text that has start and end markers. An embedding tool can then be run on the compiled executable to fill in the buffer with true binary data.

Using form element properties to house text based data is a common practice and has been seen in `VBA`, `VB6`, and even `PDF` scripts. Binary data embedded with a post processing step has been observed in the wild. In both P-Code and Native, access to these properties will be through `COM` object `VTable` calls.

From the Semi-VBDecompiler source, each different control type (including `ActiveX`) has its own parser for these compiled in property fields. Results will vary based on tool used if they can display the data. Semi-Vbdecompiler has an option to dump property blobs to disk for manual exploration. This may be required to reveal this type of embedded binary data.

UserControl Properties

A special case for the above technique occurs with the built in `UserControl` type. This control is used for hosting reusable visual elements and in `OCX` creation. The control has two events which are passed a `PropertyBag` object of its internal binary settings. This binary data can be easily set in the IDE through property pages. This mechanism can be used to store any kind of binary data including entire file systems. A public example of this technique is available[3]. Embedded data will be held per instance of the `UserControl` in its properties on the host form.

```
Sub UserControl_ReadProperties(ByRef pb As PropertyBag)
    Dim bContent() As Byte
    bContent = pb.ReadProperty("Content", bContent())
    Deserialize bContent
End Sub

Sub UserControl_WriteProperties(ByRef pb As PropertyBag)
    pb.WriteProperty "Content", Serialize()
End Sub
```

Binary Strings

Compiled `VB6` executables store internal strings with a length prefix. Similar to the form properties trick, these entries can be modified post compilation to contain arbitrary binary data. In order to discern these data blobs from other binary data, in depth understanding and complex parsing of the `VB6` file format would have to occur.

The longest string that can be embedded with this technique is limited by the line length in the IDE which is `2042` bytes (`(1023 bytes – 2 for quotes) *2 for unicode`).

`VB6` malware can access these strings normally with no special loading procedure. As far as its concerned the source was simply `str = “binary data”` .

The IDE can handle a number of unicode characters which can be embedded in the source for compilation. Full binary data can be embedded using a post processing technique.

Error Line Numbers

VB6 allows for developers to embed line numbers that can be accessed in the event of an error to help determine its location. This error line number information is stored in a separate table outside of the byte code stream.

The error line number can be accessed through the `Erl()` function. VB6 is limited to `0xFFFF` line numbers per function, and line number values must be in the `0-0xFFFF` range. Since the size of the embedded data is limited with this technique, short strings such as passwords and web addresses are the most likely use.

When the code below is run, it will output the message "secret"

```
Dim x As Long, i(2) As Integer, b() As Byte
On Error Resume Next
29541 x = "a"      'throws error wrong type
i(0) = htons(Erl) 'byte swap error line # could avoid w/ diff number
25458 x = "a"      'you can enter &h hex line numbers
i(1) = htons(Erl)
25972 x = "a"
i(2) = htons(Erl)
ReDim b((UBound(i) + 1) * 2)
CopyMemory ByVal VarPtr(b(0)), ByVal VarPtr(i(0)), (UBound(i) + 1) * 2
MsgBox StrConv(b, vbUnicode, &H409)
```

Advanced knowledge of the `VB6` file format would be required in order to discern this data from other parts of the file. Embedded data is sequential and readable if not encoded in some other way.

Function Bodies

The `AddressOf` operator allows `VB6` easy runtime access to the address of a public function in a module. It is possible to include a dummy function that is filled with just placeholder instructions to create a blank buffer within the `.text` section of the executable. This buffer can be easily loaded into a byte array with a `CopyMemory` call. A simple post compilation embedding could be used to fill in the arbitrary data.

```
Dim b(1000) As Byte
CopyMemory VarPtr(b(0)), AddressOf dummyFunc, 1000
```

For P-Code compiles, `AddressOf` returns the offset of a loader stub with a structure offset. P-Code compiles would require several extra steps but would still be possible.

References

[1] Embedded files appended to executable – theTrik:
https://github.com/thetrik/CEmbeddedFiles

[2] Embedding binary data in Bitmap images – theTrik:
http://www.vbforums.com/showthread.php?885395-RESOLVED-Store-binary-data-in UserControl&p=5466661&viewfull=1#post5466661

[3] UserControl binary data embedding – theTrik:
https://github.com/thetrik/ctlBinData

One specific malware family emphasizes how easy it can be to lose your cryptocurrency coins. It is called HackBoss - a simple yet very effective malware that has possibly stolen over $560,000 USD from the victims so far. And it's mainly being spread...

One of the goals of malware authors is to keep their creation undetected by antivirus software. One possible solution for this are crypters. A crypter encrypts a program, so it looks like meaningless data and it creates an envelope for this...