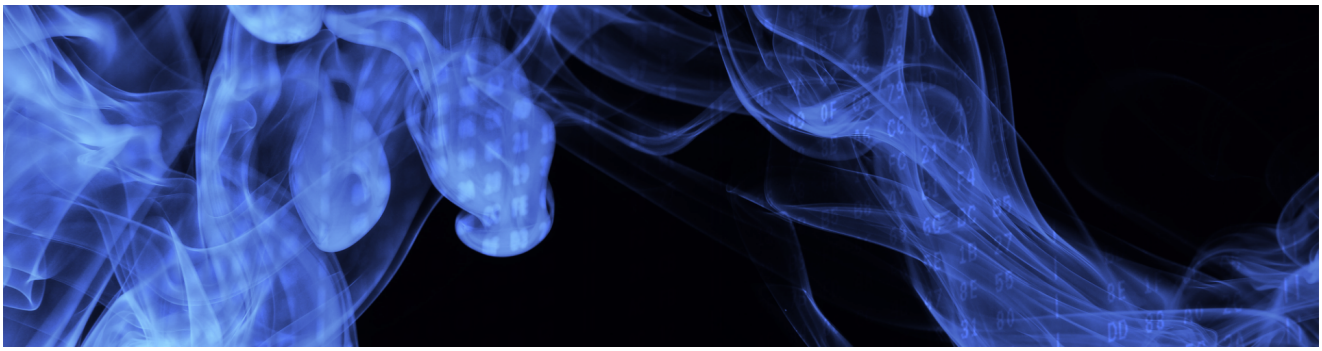
VB6 P-Code Obfuscation

decoded.avast.io/davidzimmer/vb6-p-code-obfuscation/

April 28, 2021



by David ZimmerApril 28, 20216 min read

Code obfuscation is one of the cornerstones of malware. The harder code is to analyze the longer attackers can fly below the radar and hide the full capabilities of their creations.

Code obfuscation techniques are very old and take many many forms from source code modifications, opcode manipulations, packer layers, virtual machines and more.

Obfuscations are common amongst native code, script languages, `.NET IL`, and `Java byte code`.

As a defender, it's important to be able to recognize these types of tricks, and have tools that are capable of dealing with them. Understanding the capabilities of the medium is paramount to determine what is junk, what is code, and what may simply be a tool error in data display.

On the attackers side, in order to develop a code obfuscation there are certain prerequisites required. The attacker needs tooling and documentation that allows them to craft and debug the complex code flow.

For binary implementations such as `native code` or `IL`, this would involve specs of the target file format, documentation on the opcode instruction set, disassemblers, assemblers, and a capable debugger.

One of the code formats that has not seen common obfuscation has been the `Visual Basic 6 P-Code` byte streams. This is a proprietary opcode set, in a complex file format, with limited tooling available to work with it.

In the course of exploring this instruction set certain questions arose:

- Can `VB6 P-Code` be obsfuscated at the byte stream layer?
- Has this occurred in samples in the wild?

- What would this look like?
- Do we have tooling capable of handling it?

Background

Before we continue, we will briefly discuss the `VB6 P-Code` format and the tools available for working with it.

`VB6 P-Code` is a proprietary, variable length, binary instruction set that is interpreted by the `VB6` Virtual Machine (`msvbvm60.dll`).

In terms of documentation, Microsoft has never published details of the `VB6` file format or opcode instruction set. The opcode handler names were gathered by reversers from the debug symbols leaked with only a handful of runtimes.

At one time there was a reversing community, vb-decompiler.theautomaters.com, which was dedicated to the `VB6` file format and P-Code instruction set. Mirrors of this message board are still available today [1].

On the topic of tooling the main disassemblers are `p32Disasm` , `VB-Decompiler` , `Semi-Vbdecompiler` and the `WKTVBDE` P-Code debugger.

Of these only Semi-Vbdecompiler shows you the full argument byte stream, the rest display only the opcode byte. While several private P-Code debuggers exist, `WKTVBDE` is the only public tool with debugging capabilities at the P-Code level.

In terms of opcode meanings. This is still widely undocumented at this point. Beyond intuition from their names you would really have to compile your own programs from source, disassemble them, disassemble the opcode handlers and debug both the native runtime and P-Code to get a firm grasp of whats going on.

As you can glimpse, there is a great deal of information required to make sense of P-Code disassembly and it is still a pretty dark art for most reversers.

Do VB6 obfuscators exist?

While doing research for this series of blog posts we started with an initial sample set of `25,000` P-Code binaries which we analyzed using various metrics.

Common tricks `VB6` malware uses to obfuscate their intent include:

- junk code insertion at source level
- inclusion of large bodies of open source code to bulk up binary
- randomized internal object and method names
    - mostly commonly done at pre-compilation stage
    - some tools work post compilation.

- all manner of encoded strings and data hiding
- native code blobs launched with various tricks such as `CallWindowProc`

To date, we have not yet documented P-Code level manipulations in the wild.

Due to the complexity of the vector, P-Code obsfuscations could have easily gone undetected to date which made it an interesting area to research. Hunting for samples will continue.

Can VB P-Code even be obfuscated and what would that look like?

In the course of research, this was a natural question to arise. We also wanted to make sure we had tooling which could handle it.

Consider the following `VB6` source:

```
Sub main()
    Dim s As String
    s = "th"
    s = s & "is "
    s = s & "ev"
    s = s & "il"
    MsgBox s, vbCritical
End Sub
```

The default P-Code compilation is as follows:

```
4014C4 Module1.Sub Main:
4014C4    1B 00 00              LitStr str_401224='th'
4014C7    43 78 FF              FStStrCopy var_88
4014CA    6C 78 FF              ILdRf [var_88]
4014CD    1B 01 00              LitStr str_401230='is '
4014D0    2A                    ConcatStr
4014D1    31 78 FF              FStStr var_88
4014D4    6C 78 FF              ILdRf [var_88]
4014D7    1B 02 00              LitStr str_40123C='ev'
4014DA    2A                    ConcatStr
4014DB    31 78 FF              FStStr var_88
4014DE    6C 78 FF              ILdRf [var_88]
4014E1    1B 03 00              LitStr str_401248='il'
4014E4    2A                    ConcatStr
4014E5    31 78 FF              FStStr var_88
4014E8    27 08 FF              LitVar_Missing var_F8
4014EB    27 28 FF              LitVar_Missing var_D8
4014EE    27 48 FF              LitVar_Missing var_B8
4014F1    F5 10 00 00 00        LitI4 0x10
4014F6    04 78 FF              FLdRfVar var_88
4014F9    4D 68 FF 08 40        CVarRef var_98  0x4008
4014FE    0A 04 00 14 00        ImpAdCallFPR4 rtcMsgBox
401503    36 06 00 [6 bytes]    FFreeVar var_B8 var_D8 var_F8
40150C    14                    ExitProcI4
```

An obsfuscated sample may look like the following:

```
4014C4 Module1.Sub Main:
4014C4    1B 00 00                    LitStr str_401224='th'
4014C7    1E 0C 00                    Branch loc_4014D0
4014CA    FE 67 4C FF F5 08           ForCy var_B4 loc_401DB9
4014D0    43 78 FF                    FStStrCopy var_88
4014D3    F6 A4 02 BF 76 32 1E 1C 00  LitCy 791450143732.5988
4014DC    1E 15 00                    Branch loc_4014D9
4014DF    54 FC 8B FC 8B              FMemStStrCopy
4014E4    6C 78 FF                    ILdRf [var_88]
4014E7    C3                          NotI4
4014E8    C3                          NotI4
...
401516    F4 F7                       LitI2_Byte 247
401518    FB 19                       OrI2
40151A    FC 8B                       PopAd
40151C    6C 78 FF                    ILdRf [var_88]
...
401538    F6 90 E5 FD 57 9A 1E 82 00  LitCy 3662539522243.9312
401541    1E 7A 00                    Branch loc_40153E
401544    5A                          Erase
401545    29 FC 8B                    FFreeAd  [Invalid size]
401548    FC 8B                       PopAd
40154A    31 78 FF                    FStStr var_88
40154D    1E 8E 00                    Branch loc_401552
401550    94 42 6C 78 FF              FMemLdR4
401555    1E 9A 00                    Branch loc_40155E
401558    7C 06 4A                    ImpAdStCy %c (Error)
40155B    88 95 46                    IStFPR8 arg_4695
40155E    1B 03 00                    LitStr str_401248='il'
```

From the above we see multiple opcode obfuscation tricks commonly seen in native code.

It has been verified that this code runs fine and does not cause any problems with the runtime. This mutated file has been made available on `Virustotal` in order for vendors to test the capabilities of their tooling [2].

To single out some of the tricks:

Jump over junk:

```
4014C4  1B 00 00           LitStr str_401224='th'  ; real code
4014C7  1E 0C 00           Branch loc_4014D0       ; jmp over junk
4014CA  FE 67 4C FF F5 08  ForCy var_B4 loc_401DB9 ; junk
4014D0  43 78 FF           FStStrCopy var_88       ; real code
```

Jumping into argument bytes:

```
4014D3    F6 A4 02 BF 76 32 1E 1C 00   LitCy 791450143732.5988
        ; embed a long stream of data as opcodes, push stack
4014DC    1E 15 00                     Branch loc_4014D9
        ; jmp into the previous opcode stream as instructions
4014DF    54 FC 8B FC 8B               FMemStStrCopy
        ; improperly disassembled showing garbage
4014E4    6C 78 FF                     ILdRf [var_88]
```

At runtime what executes is:

```
4014D3    F6 A4 02 BF 76 32 1E 1C 00   LitCy 791450143732.5988
        ; embed a long stream of data in opcode, push 8 bytes data onto stack
4014DC    1E 15 00                 Branch loc_4014D9
        ; jmp into the previous opcode stream as instructions
4014D9    1E 1C 00                 Branch loc_4014E0
        ; inside currency args, jmp over branch into FMemStStrCopy args
4014E0    FC 8B                    PopAd
        ; remove 4 bytes from stack (cleanup from LitCy)
4014E2    FC 8B                    PopAd
        ; remove 4 bytes from stack (cleanup from LitCy)
4014E4    6C 78 FF                 ILdRf [var_88]
```

Do nothing sequences:

```
4014E7    C3       NotI4           ; negate top stack value
4014E8    C3       NotI4           ; restore its original value

401516    F4 F7    LitI2_Byte 247  ; push 2 bytes onto stack
401518    FB 19    OrI2            ; random math on them
40151A    FC 8B    PopAd           ; pop them from stack
```

Invalid sequences which may trigger fatal errors in disassembly tools:

```
401538    F6 90 E5 FD 57 9A 1E 82 00   LitCy 3662539522243.9312
401541    1E 7A 00                     Branch loc_40153E
401544    5A                           Erase
401545    29 FC 8B                     FFreeAd  [Invalid size]

401555    1E 9A 00                     Branch loc_40155E    ;jump over junk
401558    7C 06 4A                     ImpAdStCy %c (Error)
40155B    88 95 46                     IStFPR8 arg_4695
40155E    1B 03 00                     LitStr str_401248='il'
```

Detection

The easiest markers of P-Code obfuscation are:

- jumps into the middle of other instructions
- unmatched for/next opcodes counts
- invalid/undefined opcodes
- unnatural opcode sequences not produced by the compiler
- errors in argument resolution from randomized data

Some junk sequences such as `Not Not` can show up normally depending on how a routine was coded.

This level of detection will require a competent, error-free, disassembly engine that is aware of the full structures within the `VB6` file format.

Conclusion

Code obfuscation is a fact of life for malware analysts. The more common and well documented the file format, the more likely that obfuscation tools are wide spread in the wild.

This reasoning is likely why complex formats such as `.NET` and `Java` had many public obfuscators early on.

This research proves that `VB6 P-Code` obfuscation is equally possible and gives us the opportunity to make sure our tools are capable of handling it before being required in a time constrained incident response.

The techniques explored here also grant us the insight to hunt for advanced threats which may have been already using this technique and had flown under the radar for years.

We encourage researchers to examine the mutated sample [`2`] and make sure that their frameworks can handle it without error.

References

[1] vb-decompiler.theautomaters.com mirror
http://sandsprite.com/vb-reversing/vb-decompiler/

[2] Mutated P-Code sample SHA256 and VirusTotal link
a109303d938c0dc6caa8cd8202e93dc73a7ca0ea6d4f3143d0e851cd39811261

Tagged ashidding, malware, obfuscation, P-Code, Research, series, VB