GitHub - bontchev/pcodedmp: A VBA p-code disassembler

By bontchev

Archived: 2026-04-06 03:23:26 UTC

Introduction

It is not widely known, but macros written in VBA (Visual Basic for Applications; the macro programming language used in Microsoft Office) exist in three different executable forms, each of which can be what is actually executed at run time, depending on the circumstances. These forms are:

- *Source code*. The original source code of the macro module is compressed and stored at the end of the module stream. This makes it relatively easy to locate and extract and most free DFIR tools for macro analysis like oledump or olevba or even many professional anti-virus tools look only at this form. However, most of the time the source code is completely ignored by Office. In fact, it is possible to remove the source code (and therefore make all these tools think that there are no macros present), yet the macros will still execute without any problems. I have created a proof of concept illustrating this. Most tools will not see any macros in the documents in this archive it but if opened with the corresponding Word version (that matches the document name), it will display a message and will launch `calc.exe`. It is surprising that malware authors are not using this trick more widely.

- *P-code*. As each VBA line is entered into the VBA editor, it is immediately compiled into p-code (a pseudo code for a stack machine) and stored in a different place in the module stream. The p-code is precisely what is executed most of the time. In fact, even when you open the source of a macro module in the VBA editor, what is displayed is not the decompressed source code but the p-code decompiled into source. Only if the document is opened under a version of Office that uses a different VBA version from the one that has been used to create the document, the stored compressed source code is re-compiled into p-code and then that p-code is executed. This makes it possible to open a VBA-containing document on any version of Office that supports VBA and have the macros inside remain executable, despite the fact that the different versions of VBA use different (incompatible) p-code instructions.

- *Execodes*. When the p-code has been executed at least once, a further tokenized form of it is stored elsewhere in the document (in streams, the names of which begin with `__SRP_`, followed by a number). From there it can be executed much faster. However, the format of the execodes is extremely complex and is specific for the particular Office version (not VBA version) in which they have been created. This makes them extremely non-portable. In addition, their presence is not necessary - they can be removed and the macros will run just fine (from the p-code).

Since most of the time it is the p-code that determines what exactly a macro would do (even if neither source code, nor execodes are present), it would make sense to have a tool that can display it. This is what prompted us to create this VBA p-code disassembler.

Installation

The script will work both in Python version 2.6+ and in Python 3.x. The simplest way to install it is from PyPi with `pip`:

```
pip install pcodedmp -U
```

The above command will install the latest version of `pcodedmp` (upgrading an older one if it already exists), while also installing all the necessary dependencies (currently only `oletools` and `win_unicode_console` but there might be additional ones in the future).

If you would rather install it from the GitHub repository, you can do it like this:

```
git clone https://github.com/bontchev/pcodedmp.git
cd pcodedmp
pip install .
```

Usage

The script takes as a command-line argument a list of one or more names of files or directories. If the name is an OLE2 document, it will be inspected for VBA code and the p-code of each code module will be disassembled. If the name is a directory, all the files in this directory and its subdirectories will be similarly processed. In addition to the disassembled p-code, by default the script also displays the parsed records of the `dir` stream, as well as the identifiers (variable and function names) used in the VBA modules and stored in the `_VBA_PROJECT` stream.

The script supports VBA5 (Office 97, MacOffice 98), VBA6 (Office 2000 to Office 2009) and VBA7 (Office 2010 and higher).

The script also accepts the following command-line options:

`-h`, `--help` Displays a short explanation how to use the script and what the command-line options are.

`-v`, `--version` Displays the version of the script.

`-n`, `--norecurse` If a name specified on the command line is a directory, process only the files in this directory; do not process the files in its subdirectories.

`-d`, `--disasmonly` Only the p-code will be disassembled, without the parsed contents of the `dir` stream or the identifiers in the `_VBA_PROJECT` stream.

`-b`, `--verbose` The contents of the `dir` and `_VBA_PROJECT` streams is dumped in hex and ASCII form. In addition, the raw bytes of each compiled into p-code VBA line are also dumped in hex and ASCII.

`-o OUTFILE`, `--output OUTFILE` Save the results to the specified output file, instead of sending it to the standard output.

For instance, using the script on one of the documents in the proof of concept mentioned above produces the following results:

```
python pcodedmp.py -d Word2013.doc

Processing file: Word2013.doc
================================================================================
Module streams:
Macros/VBA/ThisDocument - 1517 bytes
Line #0:
        FuncDefn (Private Sub Document_Open())
Line #1:
        LitStr 0x001D "This could have been a virus!"
        Ld vbOKOnly
        Ld vbInformation
        Add
        LitStr 0x0006 "Virus!"
        ArgsCall MsgBox 0x0003
Line #2:
        LitStr 0x0008 "calc.exe"
        Paren
        ArgsCall Shell 0x0001
Line #3:
        EndSub
```

For reference, it is the result of compiling the following VBA code:

```
Private Sub Document_Open()
    MsgBox "This could have been a virus!", vbOKOnly + vbInformation, "Virus!"
    Shell("calc.exe")
End Sub
```

Known problems

- Office 2016 64-bit only: When disassembling variables declared as being of custom type (e.g., `Dim SomeVar As SomeType`), the type (`As SomeType`) is not disassembled.
- Office 2016 64-bit only: The `Private` property of `Sub`, `Function` and `Property` declarations is not disassembled.
- Office 2016 64-bit only: The `Declare` part of external function declarations (e.g., `Private Declare PtrSafe Function SomeFunc Lib "SomeLib" Alias "SomeName" () As Long`) is not disassembled.
- Office 2000 and higher: The type of a subroutine or function argument of type `ParamArray` is not disassembled correctly. For instance, `Sub Foo (ParamArray arg())` will be disassembled as `Sub Foo`

`(arg)`.

- All versions of Office: The `Alias "SomeName"` part of external function declarations (e.g., `Private Declare PtrSafe Function SomeFunc Lib "SomeLib" Alias "SomeName" () As Long`) is not disassembled.
- All versions of Office: The `Public` property of custom type definitions (e.g., `Public Type SomeType`) is not disassembled.
- All versions of Office: The custom type of a subroutine or function argument is not disassembled correctly and `CustomType` is used instead. For instance, `Sub Foo (arg As Bar)` will be disassembled as `Sub Foo (arg As CustomType)`.
- If the output of the program is sent to a file, instead of to the console (either by using the `-o` option or by redirecting `stdout`), any non-ASCII strings (like module names, texsts used in the macros, etc.) might not be properly encoded.

I do not have access to 64-bit Office 2016 and the few samples of documents, generated by this version of Office, that I have, have been insufficient for me to figure out where the corresponding information resides. I know where it resides in the other versions of Office, but it has been moved elsewhere in 64-bit Office 2016 and the old algorithms no longer work.

To do

- Implement support of VBA3 (Excel95).
- While the script should support documents created by MacOffice, this has not been tested (and you know how well untested code usually works). This should be tested and any bugs related to it should be fixed.
- I am not an experienced Python programmer and the code is ugly. Somebody more familiar with Python than me should probably rewrite the script and make it look better.

Change log

Version 1.2.6:

- Changed it not to require the `win_unicode_console` module when it is not available - e.g., when not running on a Windows machine or when running under the PyPy implementation of Python, thanks to Philippe Lagadec.

Version 1.2.5:

- Added a sanity check to avoid errors when parsing object declarations
- The functions that produce output now have the output file (default is `stdout`) as a parameter, for better integration with other tools, thanks to Philippe Lagadec.

Version 1.2.4:

- Implemented support for module names with non-ASCII characters in their names. Thanks to Philippe Lagadec for helping me with that.
- Fixed a parsing error when disassembling object declarations.
- Removed some unused variables.
- Improved the documentation a bit.

Version 1.2.3:

- Fixed a few crashes and documented better some disassembly failures.
- Converted the script into a package that can be installed with `pip`. Use the command `pip install pcodedmp`.

Version 1.2.2:

- Implemented handling of documents saved in Open XML format (which is the default format of Office 2007 and higher) - `.docm`, `.xlsm`, `.pptm`.

Version 1.2.1:

- Now runs under Python 3.x too.
- Improved support of 64-bit Office documents.
- Implemented support of some VBA7-specific features (`Friend`, `PtrSafe`, `LongPtr`).
- Improved the disassembling of `Dim` declarations.

Version 1.2.0:

- Disassembling the various declarations (`New`, `Type`, `Dim`, `ReDim`, `Sub`, `Function`, `Property`).

Version 1.1.0:

- Storing the opcodes in a more efficient manner.
- Implemented VBA7 support.
- Implemented support for documents created by the 64-bit version of Office.

Version 1.0.0:

- Initial version.
