Purgalicious VBA: Macro Obfuscation With VBA Purging

fireeye.com/blog/threat-research/2020/11/purgalicious-vba-macro-obfuscation-with-vba-purging.html

Threat Research

Andrew Oliveau, Alyssa Rahman, Brett Hawkins

Nov 19, 2020

9 mins read

Threat Research

Malicious Office documents remain a favorite technique for every type of threat actor, from red teamers to FIN groups to APTs. In this blog post, we will discuss "VBA Purging", a technique we have increasingly observed in the wild and that was first publicly documented by Didier Stevens in February 2020. We will explain how VBA purging works with Microsoft Office documents in Compound File Binary Format (CFBF), share some detection and hunting opportunities, and introduce a new tool created by Mandiant's Red Team: OfficePurge.

MS-OVBA File Format

Before diving into VBA Purging, it is important to understand certain components of Microsoft's specifications on VBA macros (MS-OVBA). We focus on MS-OVBAs in Microsoft Office 97 documents that use the CFBF file format, instead of the modern Open Office XML (OOXML) format used by Microsoft Excel ".xlsx" and Microsoft Word ".docx" documents.

MS-OVBA's file structure stores all VBA data in a hierarchy, which consists of structured storage that contain different types of streams. VBA code in an Office document is stored in various module streams that consists of two parts: the PerformanceCache (also known as P-code), and the CompressedSourceCode. The PerformanceCache section is an array of bytes that contains compiled VBA code. The CompressedSourceCode section contains VBA source code that is compressed with Microsoft's proprietary algorithm. The boundary between the two sections is determined by a MODULEOFFSET, which is stored in the dir stream. A diagram of a module stream is shown in Figure 1.

Figure 1: Module stream diagram

When a VBA macro is added to a document, the VBA engine saves a compiled version in the PerformanceCache section of the relevant module stream to increase performance. However, an Office application will only access the PerformanceCache if its version and architecture match what was used to compile the original VBA code. This version and implementation information is stored in the _VBA_PROJECT and __SRP_# streams. If the versions do not match, the compressed source code is decompressed, compiled, and run instead.

VBA Purging vs VBA Stomping

In 2018, research by the Walmart security team brought a technique known as "VBA Stomping" to wider public awareness. Originally identified by Dr. Vesselin Bontchev in 2016, this technique allows attackers to remove compressed VBA code from Office documents and still execute malicious macros without many of the VBA keywords that AV engines had come to rely on for detection. For an example of VBA stomping in-the-wild, check out "STOMP 2 Dis: Brilliance in the (Visual) Basics".

VBA stomping takes advantage of how module streams are interpreted and exchanges malicious CompressedSourceCode with non-malicious VBA source code, leaving the PerformanceCache untouched. However, the success of this technique is Office-version dependent, implying that an attacker would have to do additional recon on their target and be aware of their victims' deployed Office versions.

VBA purging modifies the module streams in the opposite way. Instead of changing the CompressedSourceCode, VBA purging completely removes the PerformanceCache data from the module stream and the _VBA_PROJECT stream, changes the value of the MODULEOFFSET to 0, and removes all SRP streams (this is necessary because the _VBA_PROJECT and SRP streams contain version-dependent PerformanceCache data that will result in a runtime error when there is no PerformanceCache in the module stream). This removes strings usually found in PerformanceCache that many AV engines and YARA rules depend on for detection. Once removed, attackers are able to use more standard methodologies and execute suspicious functions (i.e. CreateObject) without being detected.

Figure 2 shows the OLE streams for a normal and a purged document, extracted using oledump. In the original document, the Module1 PerformanceCache is 1291 bytes, while it is 0 bytes in the VBA purged document. The purged document has no SRP streams and the _VBA_PROJECT stream has been reduced to 7 bytes.

Figure 2: Analyzing VBA purged document with oledump

Testing the Effectiveness of VBA Purging

Mandiant's Red Team created a command line, C# tool called OfficePurge to test this technique. OfficePurge supports Microsoft Office Word, Excel and Publisher documents that follow the CFBF file format. In the following examples, we used OfficePurge and a VBA payload from the public toolkit Unicorn to test the effectiveness of VBA purging a Microsoft Office Word document that contained a Base64 encoded PowerShell payload (Figure 3).

Figure 3: Macro payload generated with Unicorn

The strings output (Figure 4) for the original Word document shows Unicorn's Base64 encoded PowerShell payload, which is detected by many security products. On the other hand, the output for the VBA purged document does not fully show the Base64 encoded payload because the PerformanceCache is removed. The CompressedSourceCode still contains the Base64 encoded payload, but Microsoft's custom compression algorithm splits the strings, making it harder for static analysis to detect it.

Figure 4: Strings output with a purged and non-purged document

Both documents were submitted to online sandboxes to test detection capabilities of various products. VirusTotal's detection rate of the original document (36/60) dropped by 67% after it was VBA purged (12/61). VirusTotal also categorized the non-purged document as "create-ole", "doc", and "macros", whereas the purged document was only categorized as "doc".

Figure 5: VirusTotal detection results for non-purged Word document

Figure

6: VirusTotal detection results for purged Word document

Detection and Hunting Opportunities

With OfficePurge, we have the ability to quickly erase compiled VBA code and reduce security product detections in public sandboxes, but why stop there? Using this test data, our next step is to build conditional detection logic in formats such as YARA rules, which can identify VBA purged documents and allow us to hunt for previously undetected malicious documents. Under the “sample-data” folder in the OfficePurge GitHub repo, we have added original and purged documents for each supported file type with a macro that will spawn calc.exe. SHA256 hashes are included at the end of this post.

As mentioned before, this technique involves removing PerformanceCache data from the _VBA_PROJECT stream. MSDN documentation shows that the minimum length for the _VBA_PROJECT stream is 7 bytes to fit the required fields in the stream header. The following YARA rule searches for CFBF files with a 7 byte _VBA_PROJECT stream:

```
rule FEYE_OLE_VBAPurged_1 {
    meta:
        author = "Alyssa Rahman (@ramen0x3f)"
        description = "This file has a _VBA_PROJECT stream that has been cleared. This is evidence of VBA purging, a technique where the p-code (PerformanceCache data) is removed from Office files that have an embedded macro."
    strings:
        $vba_proj = { 5F 00 56 00 42 00 41 00 5F 00 50 00 52 00 4F 00 4A 00 45 00 43 00 54 00 00 00 00 00 00 00 00 00 }
    condition:
        uint32(0) == 0xe011cfd0 and ( uint32(@vba_proj[1] + 0x78) == 0x07 )
}
```

Searching with this logic on VirusTotal reveals a large number of malicious documents, meaning this is very prevalent in the wild and in use by attackers. This rule should identify most publicly documented examples of VBA purging, such as 9fd864e578d8bb985cf71a24089f5e2f (HornetSecurity). However, it may also identify some false positives. As previously identified by Didier Stevens, some public libraries such as EPPlus may generate benign documents without PerformanceCache data and appear to be purged.

Another important limitation of this rule is that the _VBA_PROJECT stream data doesn’t have to be completely removed. So while the stream size is 7 in all publicly documented examples of this technique, it doesn’t have to be exactly 7.

One solution to this is to compare the compressed and compiled versions of a document’s macros and look for unexpected variations. Another potential option is a YARA rule that searches the _VBA_PROJECT stream for keywords or bytes, which should appear if the p-code is valid.

But let’s take the easy path first and look for anomalies within OfficePurge. There’s a section within the code that overwrites the _VBA_PROJECT stream with a static header:

```
// Remove performance cache in _VBA_PROJECT stream. Replace the entire stream with _VBA_PROJECT header.
byte[] data = Utils.HexToByte("CC-61-FF-FF-00-00-00");
```

A little bit of Googling shows this header was built based on Microsoft’s specifications. But if we compare a purged and unpurged document, it looks like that header actually varies from specifications in practice (Figure 7).

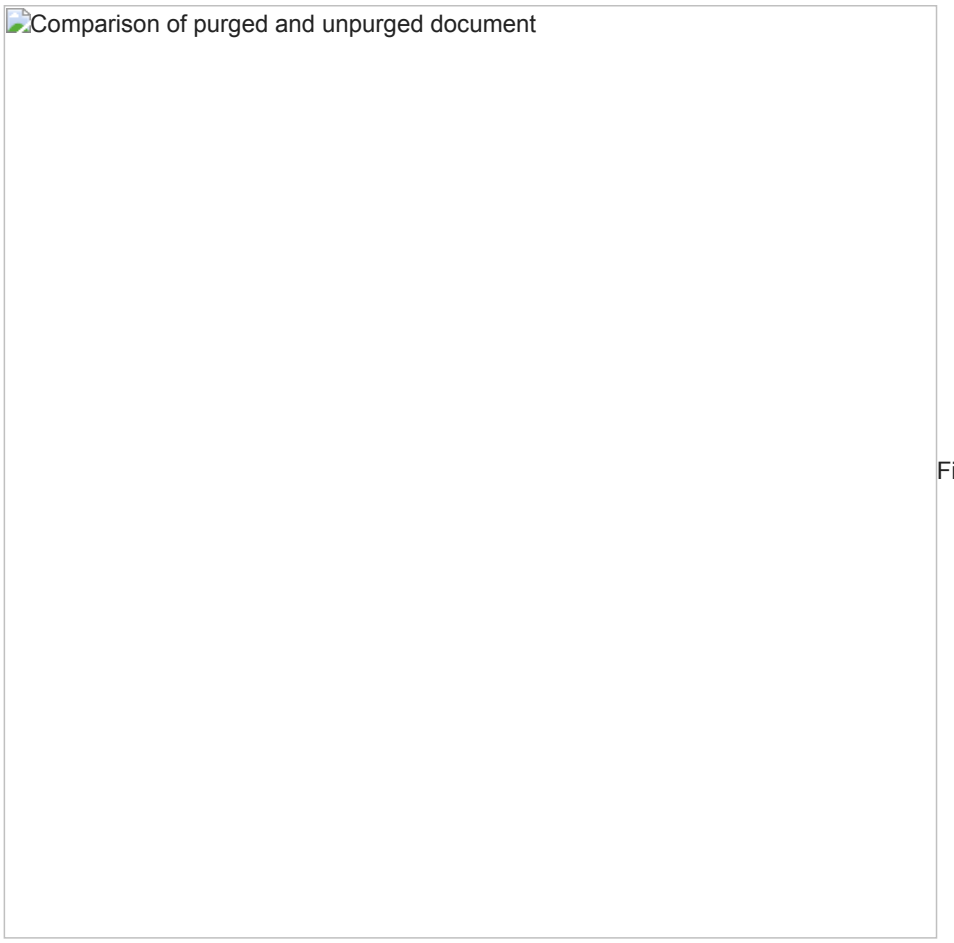

Figure 7: Comparison of purged and unpurged document

This header isn't necessarily proof that a document is malicious or was created with OfficePurge, but it could be a good indicator that the document was created programmatically versus with Office products. With anomalies such as this, we can start building a rule similar to the following, which will search for documents with a "small" _VBA_PROJECT stream and this suspicious stream header:

```
rule FEYE_OLE_VBAPurged_2 {
    meta:
        author = "Michael Bailey (@mykill), Jonell Baltazar, Alyssa Rahman (@ramen0x3f), Joseph Reyes"
        description = "This file has a suspicious _VBA_PROJECT header and a small _VBA_PROJECT stream. This may be
evidence of the VBA purging tool OfficePurge or a tool-generated document."
    strings:
        $vba_proj = { 5F 00 56 00 42 00 41 00 5F 00 50 00 52 00 4F 00 4A 00 45 00 43 00 54 00 00 00 00 00 00 00 00 00 }
        $cc61 = {CC 61 FF FF 00 00 00}
    condition:
        uint32(0) == 0xe011cfd0 and ( uint32(@vba_proj[1] + 0x78) >= 0x07 ) and ( uint32(@vba_proj[1] + 0x78) < 0xff ) and
$cc61
}
```

Searching with the two rules shared here reveals a wide range of threat actors and malware types leveraging VBA purging, or at least some type of automated document generation. On VirusTotal, you're likely to see a number of Emotet payloads caught by this rule, which is understandable given how heavily it relies on malicious email attachments. Another top offender we observed was AgentTesla.

Since these rules both turn up benign documents as well, they aren't ready for a production environment; however, they could be useful as "weak signals" for more manual threat hunting. Many static detection engines may struggle for accuracy when identifying the VBA purging technique. Dynamic analysis techniques, such as those used by FireEye's MVX engine, will still detonate the malicious document properly and be detected even if the VBA is purged.

Conclusion

For as long as companies use Office documents, attackers will be trying to smuggle malicious macros into them. VBA purging represents a recent example of how threat actors continually invent new ways to evade defenders. The artifacts discussed in this blog post should serve as a starting point for detecting VBA purging, and hopefully the tooling and indicators we have shared will help you hunt for additional anomalies in malicious Office documents. Check out OfficePurge today!

Indicators of Compromise

File Name	Description	SHA256 Hash
test.doc	Unicorn macro payload in Word document without VBA purging	f4431f02fe1e624fdb7bf2243bb72f1899d7eccb1ed7b2b42ed86e001e8bff28
test2.doc	Unicorn macro payload in Word document with VBA purging	98bd119f928e8db4ed45f5426f2c35c5f6d6ccc38af029e7ab4b9cfcc1447c53
excel_calc.xls	Sample document in OfficePurge's "sample-data folder"	de6583d338a8061bb1fc82687c8f5bff9a36ba1e2a87172e696ffaeca32567af
excel_calc_PURGED.xls	Sample document in OfficePurge's "sample-data folder"	914a6cf78fe98e80b1dee87347adbc8f8b37a1dfe672aa5196885daa447e9e73
publisher_calc.pub	Sample document in OfficePurge's "sample-data folder"	4bce7c675edde20a3357bc1d0f25b53838ab0b13824ab7a5bbc09b995b7c832f
publisher_calc_PURGED.pub	Sample document in OfficePurge's "sample-data folder"	36bdfaaf3ea228844507b1129b6927e1e69a2cd5e8af99d507121b1485d85e1e
word_calc.doc	Sample document in OfficePurge's "sample-data folder"	23fa4b77c578470c1635fe20868591f07662b998716c51fbb53d78189c06154f
word_calc_PURGED.doc	Sample document in OfficePurge's "sample-data folder"	a7eac98b3477fc97ccfe94f1419a859061ca944dc95372265e922992bd551529