Botception with Necurs: Botnet distributes script with bot capabilities

blog.avast.com/botception-with-necurs-botnet-distributes-script-with-bot-capabilities-avast-threat-labs

VBScript allows threat actors to steal personal data and make victims vulnerable to keyloggers, banking malware and ransomware

Over the past few days, we have been analyzing a development with the Necurs botnet - a cybercrime operation dating back to 2012 that quickly became one of the largest spam botnets in the world. We reported on the infamous cybergang responsible for the distribution of global malware campaigns such as “Locky” and “GlobeImposter” in two blog posts (here and here) that explained how malware is spread via Necurs. And now we have seen a new link to that chain with attackers serving brand new files via the same botnet. These files are spreading malicious Visual Basic Scripts (VBScripts) and our analysis suggests that the authors are using the services provided by the Necurs botnet to reach more victims. The ultimate goal of the attackers is to make systems vulnerable to attacks with the ability to steal personal data and to infect them with keyloggers, banking malware, and ransomware.

An examination of the source code suggests that the VBScripts are hosting a severe form of malware known as Agony Rootkit used to infect the Master Boot Record (MBR) of computers. This can be particularly destructive. By infecting the MBR, the malware is able to execute before systems boot-up allowing it to bypass security software. It then inserts a backdoor into the operating system which gives the attacker full control of the machine. With these administrative rights, the attacker can install worms, keyloggers and other malicious files. They can also access stored data including personal or financial information, putting users at risk.

A deeper look inside the VBScripts distributed by Necurs

Below are examples of live VBScripts in action. At first glance, it appears to include random numerical combinations that could be discarded as junk, possibly added to alter the code structure in order to evade detection. However, we spent some time investigating the code via the decryption function and spotted some logic behind its construction.

The scripts themselves are around 72KB and are similar in composition except for some small changes in the obfuscation methods.

```
',225,387,426,486,547,609,983,1019,1315,1362,1493,1594,
',90,184,557,619,812,1014,1374,1662,1860,1899,1945,1995
',306,867,1043,1112,1130,1210,1408,1482,1500,3221,3234,
',336,784,946,1110,1128,1208,3611,3854,7261,7582,8236,8
',4958,6026,6479,9596,

',4989,6057,6510,9598,
maxSymb = 125
Dim unCoded(13809)
Dim MamyCo(127)
'On Error Resume Next
incKasp = 0
Randomize
while (r < 999990)
```

```
',238,466,516,562,586,678,758,760,826,906,908,1012,1186,1
',2525,2972,3009,4159,4348,4978,
',3328,6010,6463,7257,

',3330,6041,6494,7288,

Dim unCoded(8144)
maxSymb = 124+1
Dim MyCode(127)
on error resume next
executeglobal chr(round(tan(CDbl("1,55874871694071")))) &
chr(round(tan(CDbl("1,56059259930094")))) & chr(round(tan
chr(round(tan(CDbl("1,56210089378312")))) & chr(round(tan
chr(round(tan(CDbl("1,56253205213525")))) & chr(round(tan
```

A closer look at the code reveals some important information. We noticed that the function uses *ReadLine*

on the file itself to decrypt the control panel (below):

```
MamyCo(i) = Replace(f143346346.ReadLine, "',", "") :: tmp = ""
For j = 1 to Len(MamyCo(i))
```

The script is really easy to deobfuscate; the only issue is comment removal — which is often present in analytical tools — that can mislead the analysts.

After decrypting these comments, in version one, we spotted a nice piece of code which could be used as a modeling example “How to write the control panel for your malware in VBS”. The next version is less self explanatory and readable.

It's probable that the authors of VBScripts connected with the authors of the Necurs botnet because of its scale and potential for ubiquitous email scams. The scripts appear to work as a downloader and control panel for the Agony rootkit, however changing the payload is particularly easy. This could open the door to more severe malware infections for victims in the future, such as ransomware or banking Trojans.

The malware control panel includes unusually beautiful code

The code itself contains very well-named variables and also debugging output to a log file. However, debugging is switched off by default as the variable *isDebug* is set to *false*. The debugging output informs about every important subroutine and contains strings such as:

"Preparing done!" (initialization and installation phase finished)
"Oh, its main cycle! CMD response" & cmd (after receiving a new command)
*"F***! Panel maybe die! I will try to change it..."* (new command has less than 4 characters)
"Unistall [sic] command gotted!" (uninstall command received)
"Oh, its ddos command!" (ddos command received)
“ddos finished! Sended "&cnt&" requests!" (ddos command finished)

Interestingly, this makes this code unusually beautiful as one rarely stumbles upon a relatively well-structured and “commented” piece of code with self-documenting names, particularly when the subject is malware.

As well, the code reveals some important information about its creator. The author appears to have a problem with the past tense of irregular verbs such as *to get* or *to send*. In addition, several log entries use incorrect English language sentence construction, such as "Panel maybe die!". It could suggest that the author’s first language is not English, however this may well be staged.

The script initializes its variables such as Command and Control (C&C) addresses, various paths and several objects that are used to interface system functions. Then, the script tries to install itself to the folder *%APPDATA%* under the names *<HWID>.vbs*, *g_<HWID>.vbs_w.vbs (HWID* being a random sequence of letters loaded from registry or generated at startup), and possibly *<static_number_sequence>_log.txt* and relaunch itself from the %APPDATA% directory. After the initial installation, it checks whether another instance of the script is running. Persistence is key, so if only one instance is running, a task named *ChromeUpdate* is created to launch the first file on user log on. Also, multiple registry entries that launch the script at startup are created. Otherwise, the script quits.

Once the script gets over the initialization, it has to ask the C&C servers for commands and process them. That is where a traditional event loop comes in. Verification that the script is located in the %APPDATA% directory has to be passed first, otherwise *agony* is invoked six times before proceeding to the next iteration. Let us assume that this test has passed. Now the script sends a POST request with a rather long GET data string, e.g.:

os=Windows 10 Enterprise&user=Evzen@DEMO-PC&av=Avast AntivirusWindows Defender&7045Mb # Intel(R) Core(TM)CPU E5-2690 v4 @ 2.60GHz # Standard VGA Graphics Adapter&hwid=AABBccddEEFFGGhhiiJJKKLLm&x=64

The only fields that require some discussion are _fw_ and _hwid_. "fw" describes the hardware (specifically *RAM size # CPU info # GPU info*), while "hwid" is a 25-character long random string generated at first launch and saved into the registry at *HKEY_CURRENT_USER\Software\ARRSSS*.

The response contains a command. These commands have a very similar structure and every command triggers a confirmation that is sent again as a POST request to the same address as a command request, however the data string differs:

ok=<zid>&hwid=<hwid> in case of success

error=<zid>&hwid=<hwid> in case of an error

The value <zid> is given by the command string. All available commands are listed below:

Command	Command string	Action
download	download!<url>!<zid>	Download a file from <url> and execute it.
update	update!<url>!<zid>	Download a PE file from <url>, save it to *%TEMP%* and execute it.
plugin	plugin!<url>!<zid>	Download a dll from <url>, save it to *%TEMP%* use RUNDLL32.exe to execute it.
uninstall	uninstall!<data>!<zid>	Replace dropped files and a log by a file with one whitespace.
ddos	ddos!<url>!<count>	Send <count> POST requests to <url>.

The denial-of-service (DDoS) attack that can be caused with the above command initiated by this script also carries some data. It seems that these attacks can be identified by the associated data, as all the scripts collected so far have the POST data hard-coded:

ufgiweugdiqwfgqofwg=32587234678235678642652634986592365 9

Now the script also ends up in several invocations of *agony*. The *agony* function serves three purposes. It checks whether the script is running in the installation directory. The result only affects which copy of the scripts is launched if no instance of *%APPDATA%\<HWID>.vbs* or *g_%APPDATA%\<HWID>.vbs_w.vbs* is running, and the log entry suggests that this is intended as a self-defense. Moreover, if the script is not named <HWID>.vbs and located in *%APPDATA%*, it overwrites its two files by *<static_number_sequence>_log.txt* and adds itself to startup through registry entries:

HKEY_CURRENT_USER\software\microsoft\windows\currentversion\run
HKEY_LOCAL_MACHINE\software\microsoft\windows\currentversion\run
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell

Every *agony* ends up with 500 milliseconds of a blissful *Sleep*, a function suspending the code execution for a given time.

Version progression

A new version, discovered approximately a day after the described script's release, brought several updates. In the new version, the payload is hidden beyond another *.vbs* file that acts as a downloader for the following stage. Moreover, its functions seem to have been refactored as e.g. the *agony* function is now removed in favour of a simple *Sleep* and the detection of an antivirus installed on the system is now hidden in a function called *func15*. Also, the script installs itself only to *%APPDATA%/<HWID>.vbs*, which simplifies checking for running instances, which means that only one function is necessary for checking.

The command structure is simplified to only include the functions *download, update*, and *uninstall*. The command *update* now expects a Visual Basic script, while *update* expects a Portable Executable (PE) file.

The last significant addition is a new function called *psCommand* that, as expected, executes a command in PowerShell. Currently, this is only used in the initialization.

VBS decryptor:
0089A6E7E92B75952F5C2E3A04A7AB65133F4CCA732BC96ECB0A34389D8FC7F4 (v1)

Dae17df6225f05e99bf0e84b3a8438560befc7eb6bd07a7b4d4e451ec33b6a5f (v2)

VBS control panel:

3011126B5210298D843D6D3B84143BE292633A4A7C0D14E947AE6BE11B74CE2F (v1)

676abca2210742e57b432558276b616b1e4e5286c772aed8c63efed230ff2430 (v2)