

# Informe Código Dañino

## CCN-CERT ID-09/20

---

**Guloader**



Abril 2020



Edita:



© Centro Criptológico Nacional, 2019

Fecha de Edición: abril de 2020

#### **LIMITACIÓN DE RESPONSABILIDAD**

El presente documento se proporciona de acuerdo con los términos en él recogidos, rechazando expresamente cualquier tipo de garantía implícita que se pueda encontrar relacionada. En ningún caso, el Centro Criptológico Nacional puede ser considerado responsable del daño directo, indirecto, fortuito o extraordinario derivado de la utilización de la información y software que se indican incluso cuando se advierta de tal posibilidad.

#### **AVISO LEGAL**

Quedan rigurosamente prohibidas, sin la autorización escrita del Centro Criptológico Nacional, bajo las sanciones establecidas en las leyes, la reproducción parcial o total de este documento por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares del mismo mediante alquiler o préstamo públicos.



## ÍNDICE

<b>1. SOBRE CCN-CERT, CERT GUBERNAMENTAL NACIONAL.....</b>	<b>4</b>
<b>2. RESUMEN EJECUTIVO.....</b>	<b>4</b>
<b>3. DETALLES GENERALES .....</b>	<b>5</b>
<b>4. PROCESO DE INFECCIÓN.....</b>	<b>6</b>
4.1 LOADER VISUAL BASIC.....	6
4.2 SHELLCODE .....	7
4.2.1 ANTI-ANÁLISIS .....	8
4.2.2 IMPORTACIÓN DE FUNCIONES.....	10
4.2.3 INYECCIÓN .....	10
4.3 SHELLCODE INYECTADO .....	12
4.3.1 PERSISTENCIA .....	12
4.3.2 COMUNICACIÓN .....	13
4.3.3 EJECUCIÓN DEL BINARIO ADICIONAL.....	15
<b>5. DESINFECCIÓN .....</b>	<b>15</b>
<b>6. REGLAS DE DETECCIÓN.....</b>	<b>16</b>
6.1 REGLA YARA .....	16
<b>7. INDICADORES DE COMPROMISO .....</b>	<b>16</b>



## 1. SOBRE CCN-CERT, CERT GUBERNAMENTAL NACIONAL

El CCN-CERT es la Capacidad de Respuesta a incidentes de Seguridad de la Información del Centro Criptológico Nacional, CCN, adscrito al Centro Nacional de Inteligencia, CNI. Este servicio se creó en el año 2006 como **CERT Gubernamental Nacional español** y sus funciones quedan recogidas en la Ley 11/2002 reguladora del CNI, el RD 421/2004 de regulación del CCN y en el RD 3/2010, de 8 de enero, regulador del Esquema Nacional de Seguridad (ENS), modificado por el RD 951/2015 de 23 de octubre.

Su misión, por tanto, es contribuir a la mejora de la ciberseguridad española, siendo el centro de alerta y respuesta nacional que coopere y ayude a responder de forma rápida y eficiente a los ciberataques y a afrontar de forma activa las ciberamenazas, incluyendo la coordinación a nivel público estatal de las distintas Capacidades de Respuesta a Incidentes o Centros de Operaciones de Ciberseguridad existentes.

Todo ello, con el fin último de conseguir un ciberespacio más seguro y confiable, preservando la información clasificada (tal y como recoge el art. 4. F de la Ley 11/2002) y la información sensible, defendiendo el Patrimonio Tecnológico español, formando al personal experto, aplicando políticas y procedimientos de seguridad y empleando y desarrollando las tecnologías más adecuadas a este fin.

De acuerdo a esta normativa y la Ley 40/2015 de Régimen Jurídico del Sector Público es competencia del CCN-CERT la gestión de ciberincidentes que afecten a cualquier organismo o empresa pública. En el caso de operadores críticos del sector público la gestión de ciberincidentes se realizará por el CCN-CERT en coordinación con el CNPIC.

## 2. RESUMEN EJECUTIVO

Se ha tenido constancia de una campaña de malware en España y Portugal, en la que usando de gancho una falsa vacuna contra el **COVID-19**, se insta al destinatario a abrir un archivo adjunto, llamado 'COVID- 19.exe' dentro de 'COVID- 19.tar', que contiene una versión comprimida de **Guloader**.

El presente documento recoge el análisis de la muestra de código dañino correspondiente a la familia de *downloaders* **Guloader**, identificada por la firma SHA256 5d91ff8d079c5d890da78adb8871e146749872911efe2ebf22cfd02c698ed33d.

El objetivo del binario es cargar en memoria un *shellcode* que descargará de un servidor remoto un *payload* adicional para su ejecución. **Guloader** apareció en la escena en diciembre de 2019, pero el pico observado en abril de 2020 por parte de actores interesados en causar infecciones con herramientas de acceso remoto, ha motivado el desarrollo de este informe. **Guloader** debe su nombre al juego de palabras



formado por “Google loader”, pues en sus inicios la tendencia del código dañino era descargar el *payload* adicional de Google Drive. El proyecto se encuentra en constante desarrollo y la ofuscación que utiliza presenta un problema para su detección.

Hoy en día los *downloaders* son un pilar fundamental en la distribución de código dañino. Una herramienta que sea difícil de detectar por parte de soluciones de seguridad presenta un atractivo especial para los actores que tratan de incrementar sus *botnets*. Si bien se trata de amenazas que no suelen persistir en el tiempo, otras familias de *downloaders* como **SmokeLoader** o **Emotet** demuestran lo contrario.

En los siguientes puntos del documento se entra en detalles técnicos sobre el comportamiento del binario inicial, así como del *shellcode* que carga en memoria. Además, se proporciona una regla YARA e indicadores de compromiso con los que identificar el binario analizado.

### 3. DETALLES GENERALES

El componente *loader* que desencadena el proceso de infección se identifica con la firma SHA256 que se muestra a continuación.

Fichero	SHA256
guloader.exe	5d91ff8d079c5d890da78adb8871e146749872911efe2ebf22cfd02c698ed33d

Se trata de un ejecutable para sistemas Windows de 32-bit, desarrollado en Visual Basic, cuya fecha de compilación muestra un valor falso, indicando a 2013.

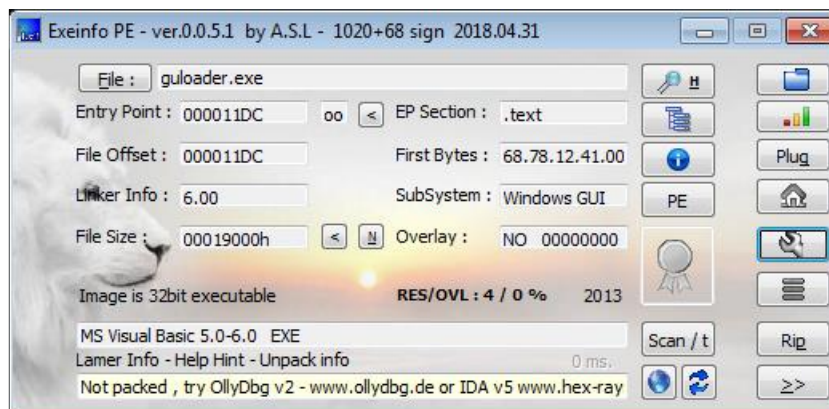


Figura 1. Propiedades del binario inicial

Buscando referencias al hash del mismo, se puede observar que la primera fecha de la que se tiene constancia es del 16 de abril de 2020.

El ejecutable no se encuentra protegido por un *packer*, sin embargo, la capa de ofuscación que presenta, con el añadido de que el esquema de ofuscación cambia para cada muestra, determina el bajo índice de detección que presenta por parte de los motores anti-virus.



## 4. PROCESO DE INFECCIÓN

Debido a la naturaleza del código dañino, el proceso de infección se divide en tres etapas:

- *Loader*, encargado de cargar en memoria el siguiente proceso.
- *Shellcode*, con una comprobación inicial de que no se ha detectado la ejecución del mismo, importación de librerías e inyección en un proceso legítimo del sistema.
- Descarga de binario desde servidor remoto e intenta lograr persistencia en el sistema.

definiendo así la estructura del presente apartado.

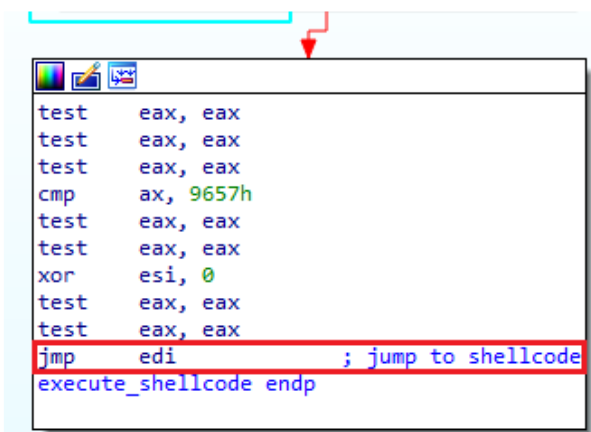
### 4.1 LOADER VISUAL BASIC

El binario inicial actúa como *loader* para el fragmento de *shellcode* que define las posteriores etapas del proceso de infección. Aunque se trata de un binario desarrollado en Visual Basic, el código encargado de cargar en memoria, realizar el *decoding* y ejecutar el *shellcode*, se puede analizar con cualquier desensamblador.

```
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
```

Figura 2. Rutina de *decoding* del *shellcode*

El fragmento de *shellcode* se encuentra en la sección **.text** del binario inicial. Tras copiarlo a un nuevo buffer en memoria reservado a tal efecto y realizar el proceso de *decoding* mediante una operación **XOR** con una clave de 4 bytes, se continúa la ejecución desde el inicio del *shellcode*, aún dentro del espacio del proceso inicial.

Figura 3. Salto incondicional al inicio del *shellcode*

El código responsable de la ejecución del *shellcode* presenta una técnica de ofuscación que consiste en la inserción de código (instrucciones basura), añadiendo de tal manera una capa de complejidad al análisis mientras se mantiene la semántica original del código. En las figuras 3 y 4, se remarcan las instrucciones de interés para el análisis, mientras que el resto se corresponden con las instrucciones basura que se anticipaban. Este método de ofuscación se emplea tanto en el componente *loader* como en el *shellcode* analizado en los siguientes apartados.

## 4.2 SHELLCODE

La segunda etapa del proceso de infección comienza con la primera ejecución del *shellcode*.

```

seg000:005B0000      ; Segment type: Pure code
seg000:005B0000      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      cld
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
  
```

Figura 4. Punto de entrada del *shellcode*

Además del método de ofuscación mencionado en el apartado anterior, el *shellcode* implementa una serie de técnicas destinadas a impedir su análisis, principalmente enfocadas a romper la ejecución dentro del *debugger*.



### 4.2.1 ANTI-ANÁLISIS

La primera de las técnicas destinadas a concluir la ejecución de forma prematura, se basa en enumerar las ventanas en el sistema mediante la función **EnumWindows**. Si el valor obtenido es inferior a lo que el código dañado espera, la ejecución finaliza en este punto.

```

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    0FFFFFFFh
seg000:005B04B8             call     dword ptr [ebp+98h] ; TerminateProcess
seg000:005B04BE             cmp     ebx, ebx
seg000:005B04C0             nop
  
```

Figura 5. El proceso es finalizado si el valor obtenido es menor que 12

Con el objetivo de impedir que un *debugger* pueda interceptar la ejecución del *shellcode*, el código dañado implementa *hooks* en las funciones **DbgBreakPoint** y **DbgUiRemoteBreakin**, pues son las encargadas de tomar el control sobre el proceso cuya ejecución se trata de interrumpir.

El *opcode* **0xCC** del código de la función **DbgBreakPoint** se sustituye por un **NOP**.

ANTES		DESPUÉS	
ntdll.dll:7735000C	ntdll_DbgBreakPoint:	ntdll.dll:7735000C	ntdll_DbgBreakPoint proc near
ntdll.dll:7735000C	int 3	ntdll.dll:7735000C	90 nop
ntdll.dll:7735000D	retn C3	ntdll.dll:7735000D	retn C3
ntdll.dll:7735000D		ntdll.dll:7735000D	ntdll_DbgBreakPoint endp

Figura 3. Hook en la función DbgBreakPoint

Por su parte, en la función **DbgUiRemoteBreakin** se sustituyen las primeras instrucciones con el objetivo de redirigir la ejecución y causar una excepción si esta función es llamada.

ANTES		DESPUÉS	
ntdll.dll:773DF7EA	ntdll_DbgUiRemoteBreakin proc near	ntdll.dll:773DF7EA	ntdll_DbgUiRemoteBreakin proc near
ntdll.dll:773DF7EA	push 8	ntdll.dll:773DF7EA	push 0
ntdll.dll:773DF7EC	offset unk_7736BA30	ntdll.dll:773DF7EC	mov eax, offset BAADF00D
ntdll.dll:773DF7F1	call near ptr unk_7736DEB4	ntdll.dll:773DF7F1	call eax ; BAADF00D
ntdll.dll:773DF7F6	mov eax, large fs:18h	ntdll.dll:773DF7F3	retn 4
ntdll.dll:773DF7FC	mov eax, [eax+30h]	ntdll.dll:773DF7F3	
ntdll.dll:773DF7FF	cmp byte ptr [eax+2], 0	ntdll.dll:773DF7F6	db 64h ; dd BAADF00D
ntdll.dll:773DF803	jnz short loc_773DF80E	ntdll.dll:773DF7F7	db 0A1h ;
ntdll.dll:773DF805	test byte_7FE0204, 2	ntdll.dll:773DF7F8	db 18h
ntdll.dll:773DF80C	jz short loc_773DF836	ntdll.dll:773DF7F9	db 0
ntdll.dll:773DF80E		ntdll.dll:773DF7FA	db 0

Figura 4. Hook en la función DbgUiRemoteBreakin

Siguiendo en la línea de las excepciones provocadas para concluir ejecución al tratar de correr el *shellcode* en un *debugger*, el código dañado trata de ocultar el hilo





de ejecución mediante la llamada a la función **NtSetInformationThread** con el valor correspondiente a **ThreadHideFromDebugger** como parámetro.

```

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      cld
seg000:005B0540      push    0
seg000:005B0542      push    11h ; ThreadHideFromDebugger
seg000:005B0544      nop
seg000:005B0545      push    0FFFFFFFh
seg000:005B0547      nop
seg000:005B0548      call    eax ; NtSetInformationThread
  
```

Figura 5. Ocultación del hilo de ejecución al *debugger*

Si tras efectuar mencionada llamada el flujo de ejecución da con un *breakpoint*, se produciría una excepción que finalizaría el proceso de ejecución.

Además de los métodos descritos para evitar que la muestra corra en un *debugger*, determinadas funciones no se llaman directamente por parte del código dañino, sino que se comprueba en detalle que ningún tipo de *breakpoint* pueda ceder el control a un analista antes de efectuar la llamada.

```

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

Figura 6. Comprobación de software y hardware *breakpoints*

La función encargada de la comprobación tratará de asegurar que ningún *breakpoint*, tanto de hardware mediante la comprobación de los registros **DR0-DR7**, como tanto de software, mediante la comprobación de los *opcodes* **0xCC**, **0x3CD** y **0xB0F**, se encuentren presentes en el momento de la solicitud de ejecución de la función a comprobar. En caso de encontrar uno, se produciría una excepción que finalizaría la ejecución del *shellcode*.



### 4.2.2 IMPORTACIÓN DE FUNCIONES

Como se observa en la figura 8, antes de realizar la llamada a **NtSetInformationThread**, se obtiene la dirección de la función de forma dinámica. Esta técnica de obtener las funciones en tiempo de ejecución presenta otra capa de protección frente al análisis estático, pues de primeras se desconoce que funciones de Windows se importan por el código dañino y donde serán llamadas.

Las funciones a importar se identifican por el valor del hash **dbj2** calculado sobre el nombre de la propia función. De esta manera para importar una función, se sitúa en el registro EDX el valor del hash y se recorren los *exports* de la librería adecuada calculando el hash **dbj2** de cada uno hasta encontrar una coincidencia.

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

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

Figura 10. Implementación de la función djb2

### 4.2.3 INYECCIÓN

Si la ejecución no ha sido interrumpida hasta este punto, el último paso del *shellcode* que está corriendo desde el espacio del proceso del binario inicial consiste en inyectarse en un binario legítimo del sistema. En el caso de la muestra de Guloader analizada, la inyección se realiza en el instalador de complementos de Internet Explorer.

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

La inyección consiste en crear el proceso suspendido en primera instancia.



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

Figura 7. Proceso ieinstal.exe suspendido



En el siguiente paso, se mapea la librería **msvbvm60.dll** en la dirección de memoria 0x00400000.

```

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
  
```

Figura 12. Mapeo de la librería msvbvm60.dll en el proceso suspendido

Finalmente, mediante la función **NtWriteVirtualMemory** se escribirá en el espacio del nuevo proceso el *shellcode* que está siendo ejecutado aún desde el binario inicial.

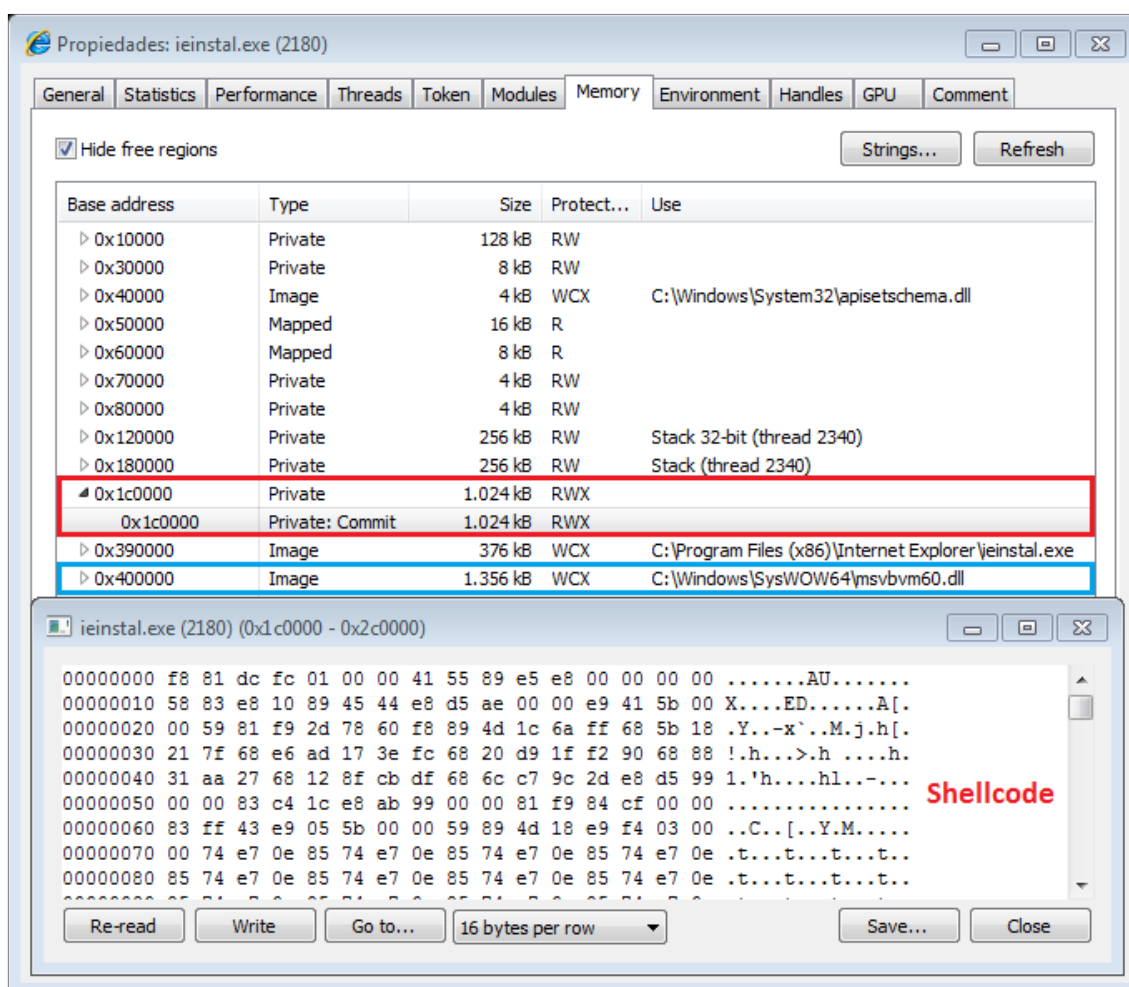


Figura 83. Shellcode inyectado en ieinstal.exe



Se puede observar que los *opcodes* de los que se compone el *shellcode* coinciden con los que se muestran en el punto de entrada del *shellcode* de la figura 4.

La primera ejecución del *shellcode* concluye tras la inyección, continuando desde el nuevo proceso y tomando una vía alternativa en el flujo de ejecución.

### 4.3 SHELLCODE INYECTADO

En esta nueva ejecución, se muestra la finalidad del *shellcode*, descargar un binario desde un servidor remoto para ejecutarlo, y lograr persistencia en el equipo para seguir cargando en memoria ese binario, aunque el sistema se reinicie.

#### 4.3.1 PERSISTENCIA

En cuanto a la persistencia, antes de escribir en el registro de Windows el valor que garantiza su ejecución a través de reinicios del equipo, primero se crea el directorio de instalación.

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

Tanto el nombre del directorio como el nombre del binario se encuentran definidos en el código del *shellcode*. El valor **IMPATIENT** en la clave de registro **Software\Microsoft\Windows\CurrentVersion\Run** apuntará al binario en su directorio de instalación, asegurando de esta manera la persistencia del código dañino.

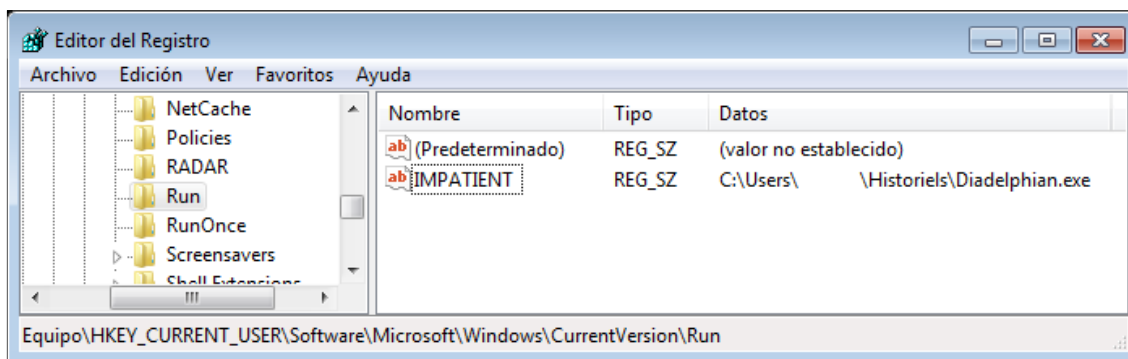


Figura 94. Persistencia del código dañino



### 4.3.2 COMUNICACIÓN

Antes de ejecutar el *payload* final alojado en un servidor remoto, primero se debe descargar dicho *payload*. La muestra de Guloader analizada presenta una URL desde la que descargar el binario a ejecutar, y una segunda URL que no ha sido utilizada en esta ocasión, mostrando lo que se podría considerar como un ejemplo definido por el *builder* del código dañino.

URLs con las descargas adicionales
<a href="https://www.seashotbin[.]com/Lord/Glx_encrypted_3277CA0.bin">https[://www.seashotbin[.]com/Lord/Glx_encrypted_3277CA0.bin</a>
<a href="http://myurl/myfile.bin">http://myurl/myfile.bin</a>

Para descargar el binario se realizará una petición GET a la URL incluida en el *shellcode* con el **user-agent** definido en también en el *shellcode*, característico de Internet Explorer 11.

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

Los servidores en los que se alojan los binarios a descargar por Guloader suelen permitir listar su contenido y el nombre de los binarios suele seguir el patrón **[aleatorio]\_encrypted\_[A-F0-9]{7}.bin**.



Figura 105. Directorio con el fichero a descargar por el código dañino en el servidor remoto

Una curiosa característica observada en el código de Guloader es que las muestras de esta familia de malware se crean para descargar un *payload* específico, comprobando su tamaño antes de proceder al *decoding* y ejecución del mismo.

```

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, 18h
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

```

Figura 11. Comprobación del tamaño del *payload* descargado



Si el tamaño concuerda, se procede a realizar un XOR *decoding* sobre el binario descargado de forma previa a su ejecución. Analizando el binario con el *encoding*, se observa una cabecera con formato [a-f0-9]{64}, que no forma ningún papel en la rutina de *decoding* para obtener el ejecutable final. En la instrucción 0x005B0AF6 de la figura 16 se puede observar cómo se descartan estos 64 (0x40) bytes para la comprobación del tamaño.

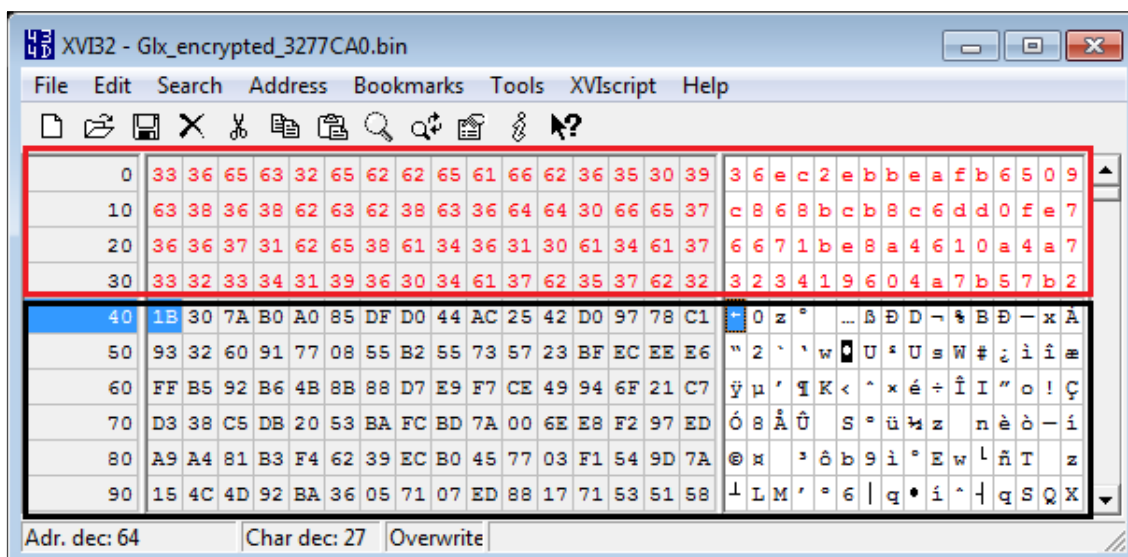


Figura 12. Payload descargado del servidor remoto

A fecha del 20 de abril de 2020, los actores parecen haber decidido dejar atrás el formato de la cabecera mencionado y en lugar de [a-f0-9]{64}, se amplía el rango a cualquier valor resultando en [0x00-0xFF]{64}. Pese al cambio en el formato, la cabecera sigue siendo descartada.

Para su *decoding*, el shellcode aloja la clave para la operación **XOR**, que para este caso es de una longitud de 575 bytes. La muestra de código dañino a ejecutar por la muestra de Guloader analizada se corresponde con la herramienta de acceso remoto **Netwire**.

Fichero	SHA256
Netwire	af0b56fffc1e8df83dc104e0afe91f8921ecfc66fb5599214189fdc90ec1a4d



### 4.3.3 EJECUCIÓN DEL BINARIO ADICIONAL

En lugar de crear un nuevo proceso con el binario descargado, Guloader mapeará el nuevo ejecutable en la dirección 0x00400000 (donde antes estaba la librería **msvbvm60.dll**) y lo correrá como un nuevo hilo de ejecución, ocultándolo también a los *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 ; ':'

Figura 18. Ejecución del binario descargado en un nuevo hilo

Tras lanzar la nueva muestra de código dañino, la parte de Guloader habrá concluido y su hilo ejecución termina en este punto.

## 5. DESINFECCIÓN

De cara a desinfectar un equipo en el que se haya instalado la muestra de Guloader analizada, se proponen los siguientes puntos.

1. Eliminar el directorio de instalación indicado en el apartado de persistencia y su contenido.
2. Eliminar la clave de registro apuntando al directorio de instalación indicada en el apartado de persistencia.
3. Reiniciar el equipo.

Dada la naturaleza de la muestra analizada y debido a su capacidad de recibir ficheros adicionales, no se puede garantizar una limpieza completa del equipo tras desinstalar el binario objeto de análisis.



## 6. REGLAS DE DETECCIÓN

### 6.1 REGLA YARA

```
rule guloader
{
  meta:
    date = "2020-04-20"
    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. INDICADORES DE COMPROMISO

Fichero	SHA256
Loader Visual Basic	5d91ff8d079c5d890da78adb8871e146749872911efe2ebf22cfd02c698ed33d
Netwire	af0b56fffc1e8df83dc104e0afe91f8921ecfc66fb5599214189fdc90ec1a4d

Directorio de instalación
C:\Users\[User]\Historiels\Diadelphian.exe

Localización del payload adicional
<a href="https://www.seashotbin[.]com/Lord/Glx_encrypted_3277CA0.bin">https://www.seashotbin[.]com/Lord/Glx_encrypted_3277CA0.bin</a>

Clave de registro	Valor de la clave
HKCU\Software\Microsoft\Windows\CurrentVersion\Run\IMPATIENT	C:\Users\[User]\Historiels\Diadelphian.exe