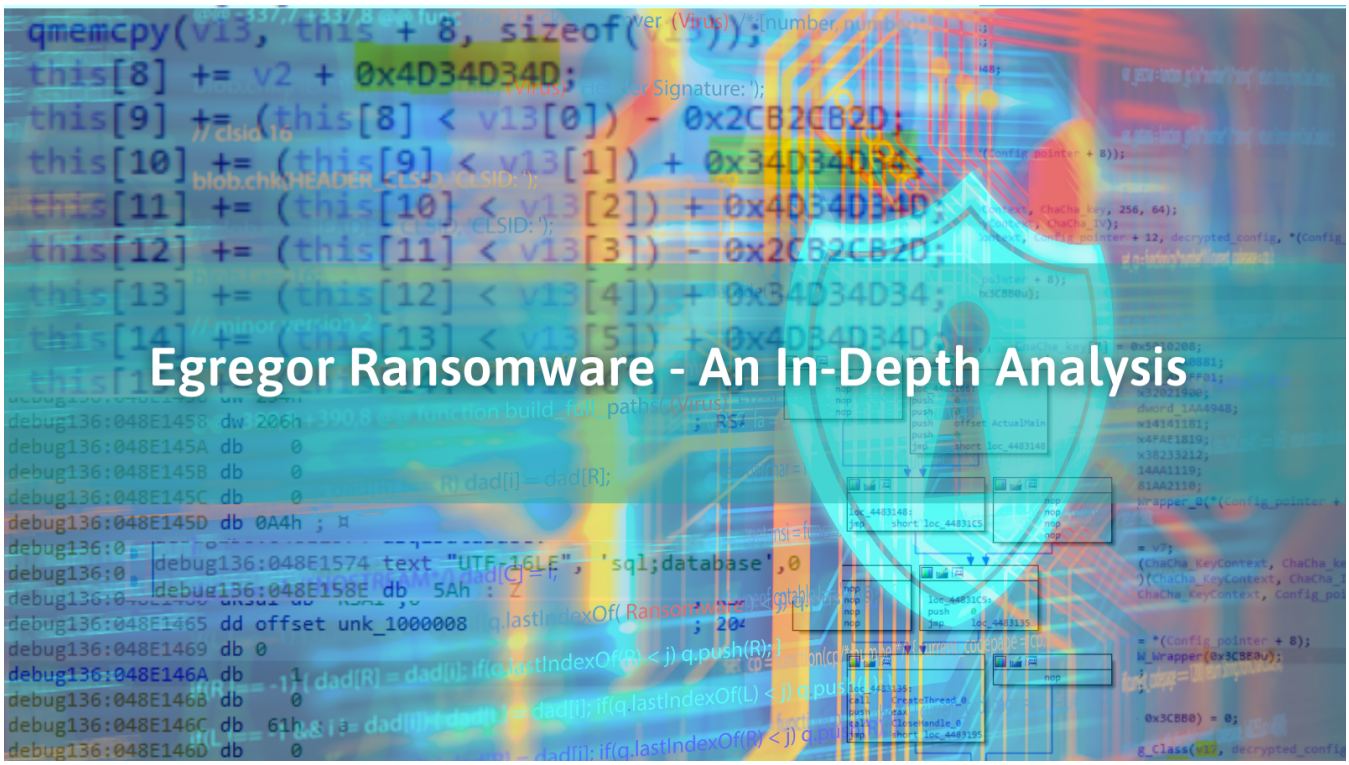
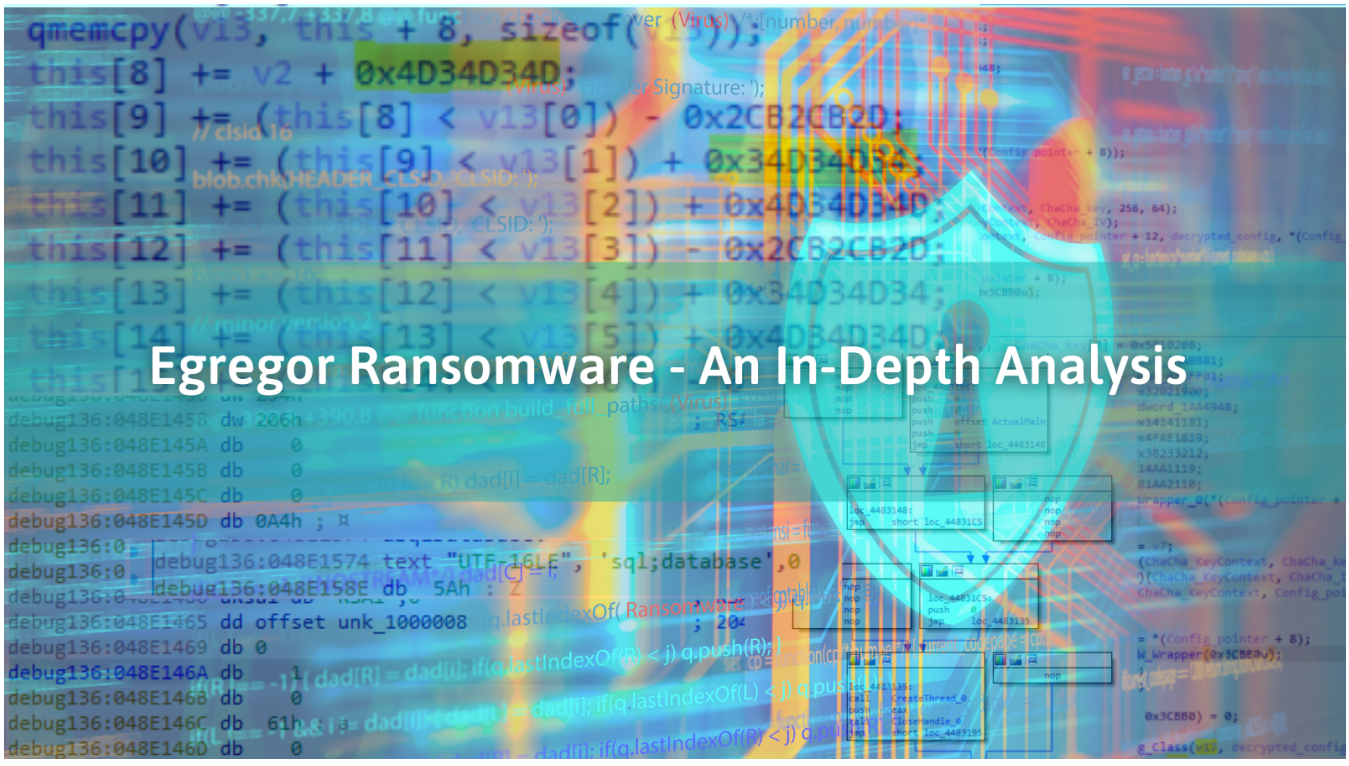
Egregor Ransomware - An In-Depth Analysis

blog.minerva-labs.com/egregor-ransomware-an-in-depth-analysis





- Tweet
-

Minerva Labs undertook a detailed research of the Egregor ransomware, with the goal of providing an in-depth analysis of how it works to infect a target. Better knowledge of threat actor's techniques can help security experts detect and mitigate novel threats, which is especially important considering the recent evolution of ransomware. In the scope of this research we have tried to determine what evasive techniques the malware uses.

The recent surge in Egregor ransomware and the code similarity to Sekhmet and Maze ransomware strains leads us to believe that they probably share the same code base. In addition, similar code obfuscation techniques are used by Maze and Egregor, which slow the analysis process and hinder researchers.

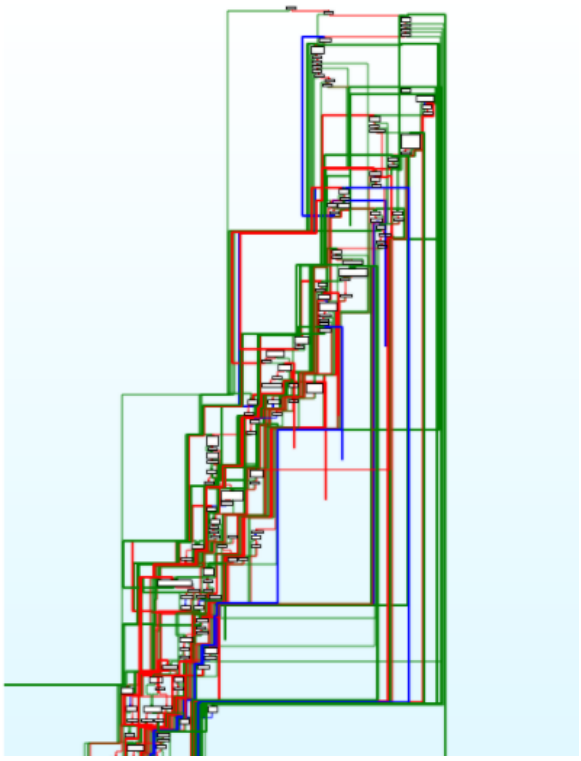
Blog Posts about Egregor detail varying degrees of obfuscation. In our case both the loader and the actual ransomware were highly obfuscated, which forced us to write deobfuscation scripts that ease the analysis process.

The Loader

The ransomware we encountered is a DLL file named b.dll. The file was executed manually using the following command line:

A look at the disassembly of the code indicates that it is highly obfuscated with compiler-based techniques, which makes static analysis of the code quite time consuming.

For example, see below the function DllRegisterServer, which the attacker uses to launch the ransomware, in IDA's graph view:



The obfuscation Egregor uses is similar to the one used in Maze ransomware. We were able to modify Blueliv's Maze deobfuscation script (the blogpost and the original script can be found here) to fit Egregors obfuscation patterns, which allowed for easier analysis of the ransomware.

The loader checks for the command line "--nop" and exits if it exists.

As for further unpacking, a large blob of data is decrypted with the following steps:

- The blob is xor decoded with a hardcoded key (0x4 in our sample).
- The xor'ed data is then Base64 decoded using the windows API function CryptStringToBinaryA.

- A hardcoded key and IV is initialized for the ChaCha20 algorithm, which is then used for the final decryption of the payload. The malware authors decided to change the number of rounds of key rotations from the default of 20 to only 4.

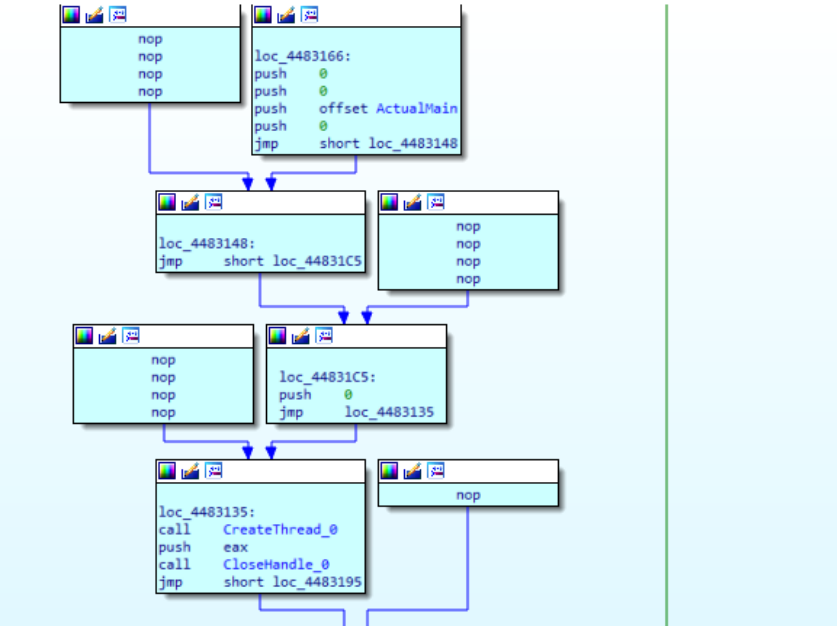
After decrypting the second payload, a DLL file, it is copied to a new allocation that is created using VirtualAlloc with the page permissions RWX.

The last stage of the initial loader is the preparation of the payload in memory. The malware reflectively loads the decrypted payload and uses the function CreateThread to transfer execution to its next stage.

The next stage parses the command line, looking specifically for the parameter -p, which contains a password that is used for the decryption of the ransomware binary. The ransomware is decrypted using a stream cipher that shares some of its constants with Rabbit cipher:

```
qmemcpy(v13, this + 8, sizeof(v13));
this[8] += v2 + 0x4D34D34D;
this[9] += (this[8] < v13[0]) - 0x2CB2CB2D;
this[10] += (this[9] < v13[1]) + 0x34D34D34;
this[11] += (this[10] < v13[2]) + 0x4D34D34D;
this[12] += (this[11] < v13[3]) - 0x2CB2CB2D;
this[13] += (this[12] < v13[4]) + 0x34D34D34;
this[14] += (this[13] < v13[5]) + 0x4D34D34D;
this[15] += (this[14] < v13[6]) - 0x2CB2CB2D;
this[16] = this[15] < v13[7];
```

Ransomware Code:
The ransomware is compiled as a DLL file with only one export named “DllEntryPoint”. The function creates a thread that executes the main subroutine of the ransomware:



Before starting the ransomware’s malicious procedure, a function is called to determine the locale of the workstation. The ransomware uses three different Windows API functions to make sure it is not encrypting a computer located in Russia or any other CIS country:

```
LangId_1 = GetUserDefaultUILanguage();
LangId_2 = GetSystemDefaultLangID();
LangId_3 = GetUserDefaultLangID();
if ( LangId_3 == 0x419
  || LangId_1 == 0x419
  || LangId_2 == 0x419
  || LangId_3 == 0x422
  || LangId_1 == 0x422
  || LangId_2 == 0x422
  || LangId_3 == 0x423
  || LangId_1 == 0x423
  || LangId_2 == 0x423
  || LangId_3 == 0x428
  || LangId_1 == 0x428
  || LangId_2 == 0x428
  || LangId_3 == 0x42B
  || LangId_1 == 0x42B
  || LangId_2 == 0x42B
  || (unsigned __int16)LangId_3 == 0x42C
  || LangId_1 == 0x42C
  || LangId_2 == 0x42C
  || (unsigned __int16)LangId_3 == 0x437
  || LangId_1 == 0x437
  || LangId_2 == 0x437
  || (unsigned __int16)LangId_3 == 0x43F
  || LangId_1 == 0x43F
  || LangId_2 == 0x43F
```

Egregor will terminate if any of the following locales are found:

Locale Code	Country
0x843	Uzbek - Cyrillic
0x819	Russian - Moldova
0x440	Kyrgyz - Cyrillic
0x442	Turkmen
0x82C	Azerbaijani
0x423	Belarusian
0x42B	Armenian
0x443	Uzbek - Latin
0x43F	Kazakh
0x437	Georgian
0x42C	Azerbaijani
0x818	Romanian - Moldova
0x444	Tatar
0x428	Tajik

After the locale check, the ransom configuration will be decrypted from a buffer located in the data section of the executable. The first 8 bytes of the encrypted configuration starts with a PNG header which is skipped by the parser before its decryption. The subsequent DWORD contains the size of the configuration to decrypt. Starting from offset 12, the configuration will be decrypted using round-modified ChaCha20 and a hardcoded key and IV.

The encrypted configuration in-memory:

```
debug121:04932044 unk_4932044 db  89h ; ‰                ; DATA XREF: sub_4912510+1AAto
debug121:04932044                                        ; redundant PNG header
debug121:04932045 db  50h ; P
debug121:04932046 db  4Eh ; N
debug121:04932047 db  47h ; G
debug121:04932048 db  0Dh
debug121:04932049 db  0Ah
debug121:0493204A db  1Ah
debug121:0493204B db  0Ah
debug121:0493204C dd 1A8Ch                               ; size of the configuration
debug121:04932050 db 0A7h ; §                            ; Start of the actual encrypted config
debug121:04932051 db 0E3h ; ã
debug121:04932052 db 0BDh ; ½
debug121:04932053 db  19h
debug121:04932054 db  63h ; c
debug121:04932055 db  64h ; d
```

Decompilation of the configuration class initialization function:

```
ChaCha_key[7] = 0x5010208;
ChaCha_key[6] = &unk_3AA0881;
ChaCha_key[5] = &unk_433FF01;
ChaCha_key[4] = 0x32021900;
ChaCha_key[3] = &dword_1AA4948;
ChaCha_key[2] = 0x14141181;
ChaCha_key[1] = 0x4FAE1819;
ChaCha_key[0] = 0x38233212;
ChaCha_IV[1] = 0x14AA1119;
ChaCha_IV[0] = 0x81AA2110;
v7 = Allocate_RW_Wrapper_0(*(Config_pointer + 8));
if ( !v7 )
  break;
decrypted_config = v7;
(ChaCha_Init_Key)(ChaCha_KeyContext, ChaCha_key, 256, 64);
(ChaCha20_init_IV)(ChaCha_KeyContext, ChaCha_IV);
(ChaCha_Encrypt)(ChaCha_KeyContext, Config_pointer + 12, decrypted_config, *(Config_pointer + 8));
if ( v9 || !v9 )
{
  size_of_config = *(Config_pointer + 8);
  v6 = Allocate_RW_Wrapper(0x3CBB0u);
  *(v14 + 4) = v6;
  v8 = 0xFFFC3450;
  do
    *(v6 + v8++ + 0x3CBB0) = 0;
  while ( v8 );
  initilize_config_Class(v17, decrypted_config, size_of_config);
```

The configuration contains several interesting settings:

- The ransom note.
- List of processes to terminate.
- Blacklisted keywords for the services termination algorithm.
- A hardcoded RSA 2048-bit public key which is used for the file encryption scheme.
- Flags for the presence of remote addresses.

In order to create a fingerprint of the encrypted workstation Egregor uses several API functions to extract information about the machine:

```
debug168:04C4001C dd offset dword_4C90000
debug168:04C40020 dd offset aWorkgroup_0                 ; "WORKGROUP"
debug168:04C40024 db    0
debug168:04C40025 db    0
debug168:04C40026 db    0
debug168:04C40027 db    0
debug168:04C40028 dd offset aCF164540204218              ; "|C:F_164540/204218|D:C_0/820|"
debug168:04C4002C dd offset aTimi                        ; "timi"
debug168:04C40030 dd offset aWindowsDefende              ; "Windows Defender;"
debug168:04C40034 db    0
debug168:04C40035 db    0
debug168:04C40036 db    0
debug168:04C40037 db    0
debug168:04C40038 db    0
debug168:04C40039 db    0
debug168:04C4003A db    0
debug168:04C4003B db    0
debug168:04C4003C dd offset aWindows10Enter              ; "Windows 10 Enterprise N"
debug168:04C40040 db    0
```

The ransomware uses the API functions GetLogicalDriveStrings and GetDiskFreeSpace to identify the names and types of the logical disks connected to the device in addition to the amount of free space available in them.

The ransomware RSA public key in memory, stored in the encrypted configuration:

```
debug136:048E1458 dw 206h                               ; RSA PRIVATE KEY
debug136:048E145A db    0
debug136:048E145B db    0
debug136:048E145C db    0
debug136:048E145D db 0A4h ; ¤
debug136:048E145E db    0
debug136:048E145F db    0
debug136:048E1460 aRsa1 db 'RSA1',0                     ; RSA Magic
debug136:048E1465 dd offset unk_1000008                 ; 2048 bit
debug136:048E1469 db 0
debug136:048E146A db    1
debug136:048E146B db    0
debug136:048E146C db  61h ; a
debug136:048E146D db    0
```

For each execution, a pair of private and public keys are generated. The public key is used for encrypting the symmetrical keys that would later be used for encrypting each file. A unique symmetrical key is generated for every file to be encrypted.

Egregor's key generation scheme is as follows:

- A 2048-bit RSA key pair is generated using CryptGenKey – this is the session key.
- The key is then exported using the API CryptExportKey.
- The exported private key is encrypted with ChaCha using a randomly generated key and IV.
- The ChaCha keys are encrypted using the function CryptEncrypt and the configuration-embedded RSA public key.
- The encrypted ChaCha key and the encrypted session key are saved to disk in a hardcoded path, which in our case is %ProgramData%\dtb.dat.

It is worth noting that the ransomware encrypts the session key with the same protocol that is used to decrypt the ransomware payload (Rabbit Cipher).

The ransomware will stop certain processes and services before encrypting the machine. A list of hardcoded process names is stored in the encrypted configuration file and the malware uses NtQuerySystemInformation to enumerate the running processes and terminates them using the function NtTerminateProcess.

The list of processes that will be terminated, in our sample (a list will also be available in the IOCs section):

```
debug136:035F1592 db    0
debug136:035F1593 aSftesqlExeSqla:
debug136:035F1593 text "UTF-16LE", 'sftesql.exe;sqlagent.exe;sqlbrowser.exe;sqlwriter.e'
debug136:035F1593 text "UTF-16LE", 'xe;oracle.exe;ocssd.exe;dbsnmp.exe;synctime.exe;agn'
debug136:035F1593 text "UTF-16LE", 'tsvc.exe;isqlplussvc.exe;xfssvccon.exe;sqlservr.exe'
debug136:035F1593 text "UTF-16LE", ';mydesktopservice.exe;ocautoupds.exe;encsvc.exe;fir'
debug136:035F1593 text "UTF-16LE", 'efoxconfig.exe;tbirdconfig.exe;mydesktopqos.exe;oco'
debug136:035F1593 text "UTF-16LE", 'mm.exe;mysqld.exe;mysqld-nt.exe;mysqld-opt.exe;dben'
debug136:035F1593 text "UTF-16LE", 'g50.exe;sqbcoreservice.exe;excel.exe;infopath.exe;m'
debug136:035F1593 text "UTF-16LE", 'saccess.exe;mspub.exe;onenote.exe;outlook.exe;power'
debug136:035F1593 text "UTF-16LE", 'pnt.exe;sqlservr.exe;thebat.exe;steam.exe;thebat64.'
debug136:035F1593 text "UTF-16LE", 'exe;thunderbird.exe;visio.exe;winword.exe;wordpad.e'
debug136:035F1593 text "UTF-16LE", 'xe;QBW32.exe;QBW64.exe;ipython.exe;wpython.exe;pyth'
debug136:035F1593 text "UTF-16LE", 'on.exe;dumpcap.exe;procmon.exe;procmon64.exe;procex'
debug136:035F1593 text "UTF-16LE", 'p.exe;procexp64.exe',0
debug136:035F1A83 db 60h
```

As for the service stopping algorithm, the ransomware configuration contains a list of strings that will be used to determine which service should be stopped. Services names will be enumerated using the API function EnumServicesStatus. Any service name that contains the blacklisted strings will be stopped using windows Service Control Manager API.

The list of services keywords in the configuration:

```
debug136:048E1574 text "UTF-16LE", 'sql;database',0
debug136:048E158E db  5Ah ; Z
```

Egregor has the capability to contact hardcoded HTTP URLs. If the offset 0x3a31e and 0x32fb in the configuration does not contain 0, the ransomware will contact IP address/DNS names (which are also embedded in the configuration), and decode their content using the same modified-ChaCha20/Base64 combination used before.

The IDAPython deobfuscation script can be found here.

IOCs:

Hashes:

b9b71eb04d255b21e3272eef5f4c15d1c208183748dfad3569efd455d87879c6 (Egregor loader)

8d5ad342ea9fde48920a926780be432236d074d34f791b5c96ec3a418a1bbbd5

(unpacked ransomware from memory)

Files:

%ProgramData%\dtb.dat

RECOVER-FILES.TXT

Terminated Processes:

msftesql.exe	agntsvc.exe	tbirdconfig.exe	excel.exe	thebat.exe	procmon.exe	procexp.exe
sqlagent.exe	isqlplussvc.exe	mydesktopqos.exe	infopath.exe	steam.exe	procmon64.exe	procexp64.exe
sqlbrowser.exe	xfssvccon.exe	ocomm.exe	msaccess.exe	thebat64.exe	ipython.exe	wpython.exe
sqlwriter.exe	sqlservr.exe	mysqld.exe	mspub.exe	thunderbird.exe	python.exe	dumpcap.exe
oracle.exe	dbeng50.exe	mysqld-nt.exe	onenote.exe	visio.exe	QBW64.exe	synctime.exe
ocssd.exe	ocautoupds.exe	mysqld-opt.exe	outlook.exe	winword.exe	firefoxconfig.exe	sqlservr.exe
dbsnmp.exe	encsvc.exe	mydesktopservice.exe	powerpnt.exe	wordpad.exe	sqbcoreservice.exe	QBW32.exe

References:

- ChaCha20 - https://en.wikipedia.org/wiki/Salsa20
- Rabbit Cipher - https://en.wikipedia.org/wiki/Rabbit_(cipher)
- Blueliv's maze deobfuscation - https://www.blueliv.com/cyber-security-and-cyber-threat-intelligence-blog-blueliv/escape-from-the-maze/
- Minerva Labs Sekhmet - https://blog.minerva-labs.com/minervalabs-vs-sekhmet