LIFARS

your digital world, **secured**

MALWARE ANALYSIS- UNPACKING OF EGREGOR RANSOMWARE

January 2021

TABLE OF CONTENTS

LIFARS
your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

EXECUTIVE SUMMARY

In this case study, we describe malware analysis and unpacking of a newly emerged ransomware Egregor. It is an extremely targeted ransomware that tries to extort big companies. The sample that we analyzed was obtained by our colleagues during an incident response at our client's organization.

We reverse engineered and debugged the sample, thus we managed to overcome two loaders and fully unpack the payload. The initial sample consisted of one DLL (named clang.dll) which executed itself in three stages. The DLL loaded a second DLL, which loaded a third DLL containing the actual payload.

One of our key findings is that the execution of the initial malicious DLL had to be invoked with a specific parameter, otherwise the payload was not unpacked. This secret parameter started with –p and it served as a password to correctly decrypt the payload and the attacker had to type it in the command line to detonate the ransomware.

INTRODUCTION

We used Hiew, capa, and IDA for static analysis and reverse engineering, x32dbg for debugging and we ran the malware in a sandbox and examined it with Process Hacker.

This whitepaper is structured in the following way:

1. The First Reflective DLL Loader

   a) Hiew and capa Analysis
   b) IDA Analysis
   c) x32dbg Analysis

2. The Second Reflective DLL Loader

   a) Hiew and capa Analysis
   b) IDA Analysis (and the –p parameter)
   c) x32dbg Analysis

3. Payload

   a) Hiew and capa Analysis
   b) Highlights from IDA Analysis
   c) Dynamic Analysis
   d) OSINT

In sections 1. and 2. we describe reverse engineering of the two nested Reflective DLL Loaders and our subsequent debugging which resulted in obtaining an unpacked payload from the memory. In section 3. we write about basic traits of ransomware payload.

LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

Stage 1: clang.dll (1st Reflective DLL Loader)

Hiew and capa analysis

As the first step of our analysis, we ran a tool called **capa** against clang.dll. capa has a collection of rules created by the cybersecurity community through which it can detect potentially malicious capabilities of an executable file and assign MITRE ATT&CK techniques to them. This is what a capa outcome looked like:

```
+------------------------+------------------------------------------------------------------------+
| ATT&CK Tactic          | ATT&CK Technique                                                       |
|------------------------+------------------------------------------------------------------------|
| DEFENSE EVASION        | Process Injection [T1055]                                              |
|                        | Virtualization/Sandbox Evasion::System Checks [T1497.001]              |
| DISCOVERY              | File and Directory Discovery [T1083]                                   |
| EXECUTION              | Shared Modules [T1129]                                                 |
+------------------------+------------------------------------------------------------------------+

+----------------------------+--------------------------------------------------------------------+
| MBC Objective              | MBC Behavior                                                       |
|----------------------------+--------------------------------------------------------------------|
| ANTI-BEHAVIORAL ANALYSIS   | Virtual Machine Detection::Instruction Testing [B0009.029]         |
+----------------------------+--------------------------------------------------------------------+

+-------------------------------------------------------+-----------------------------------------+
| CAPABILITY                                            | NAMESPACE                               |
|-------------------------------------------------------+-----------------------------------------|
| execute anti-VM instructions (2 matches)              | anti-analysis/anti-vm/vm-detection      |
| hash data using FNV                                   | data-manipulation/hashing/fnv           |
| contains PDB path                                     | executable/pe/pdb                       |
| accept command line arguments (2 matches)             | host-interaction/cli                    |
| query environment variable (2 matches)                | host-interaction/environment-variable   |
| set environment variable (4 matches)                  | host-interaction/environment-variable   |
| enumerate files via kernel32 functions (2 matches)    | host-interaction/file-system/files/list |
| write file (8 matches)                                | host-interaction/file-system/write      |
| print debug messages (5 matches)                      | host-interaction/log/debug/write-event  |
| allocate thread local storage (2 matches)             | host-interaction/process                |
| get thread local storage value (2 matches)            | host-interaction/process                |
| set thread local storage value (2 matches)            | host-interaction/process                |
| allocate RWX memory (2 matches)                       | host-interaction/process/inject         |
| terminate process (4 matches)                         | host-interaction/process/terminate      |
| terminate process via fastfail (6 matches)            | host-interaction/process/terminate      |
| create thread (3 matches)                             | host-interaction/thread/create          |
| link function at runtime (4 matches)                  | linking/runtime-linking                 |
| parse PE header (6 matches)                           | load-code/pe                            |
+-------------------------------------------------------+-----------------------------------------+
```

The following capabilities were worth noticing:

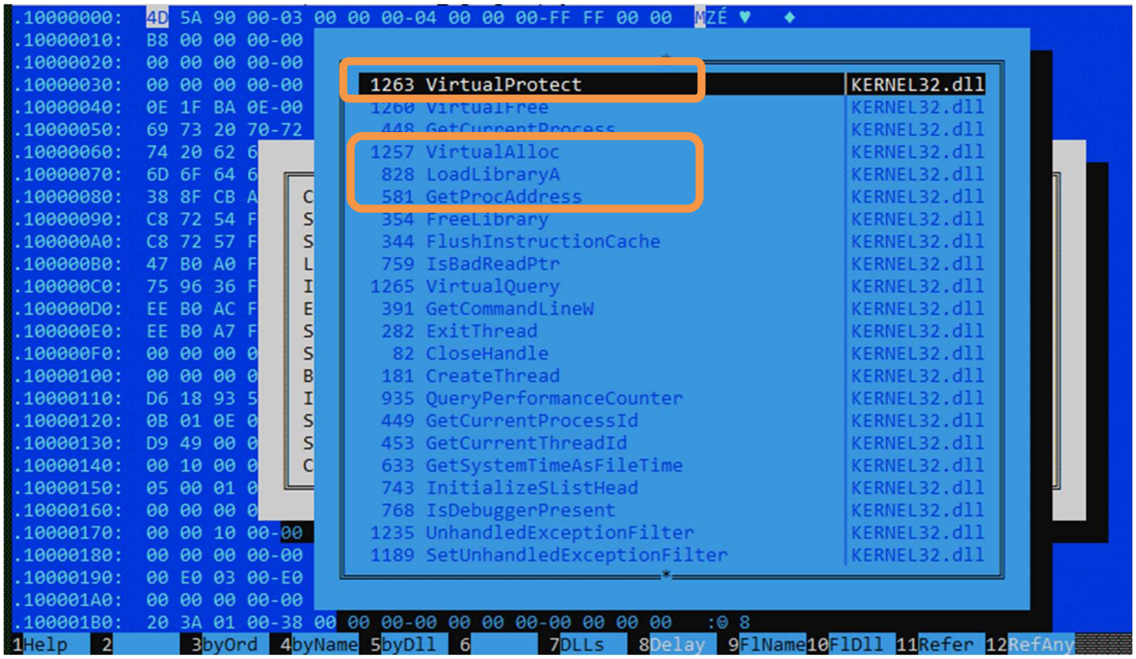
- allocate RWX memory
- parse PE header
- link function at runtime

These capabilities indicated that the DLL could be a loader. Such loader allocates memory with read, write, and execute permissions in the first step. (Usually, a harmless executable does not allocate memory with both write and execute permissions at the same time.) In its next steps, a loader generally unpacks (decodes and decrypts) a previously packed code, parses its PE header, and links the necessary functions.

The fact that clang.dll was a loader, was also supported by the imports we found when we examined it in **Hiew**. *VirtualAlloc* and *VirtualProtect* implied possible allocation of memory with both write and execute

LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

permissions at the same time. *LoadLibrary* and *GetProcAddress* implied that the executable would load necessary DLLs and their functions. This capability was identified by capa as 'link functions at runtime'.

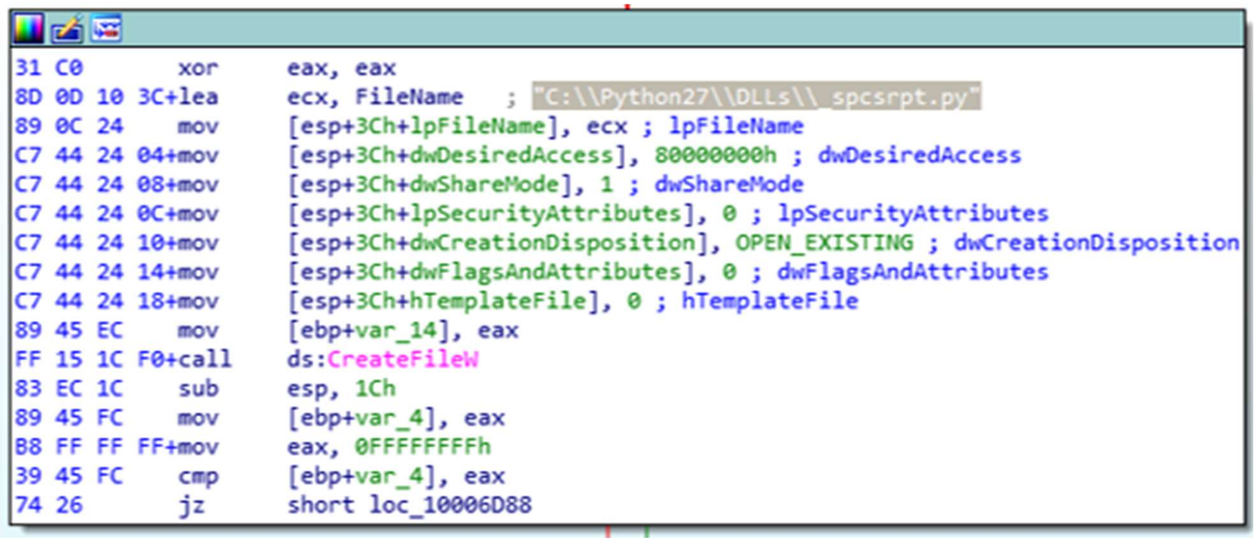


After this initial examination, we opened clang.dll in **IDA**.

IDA analysis

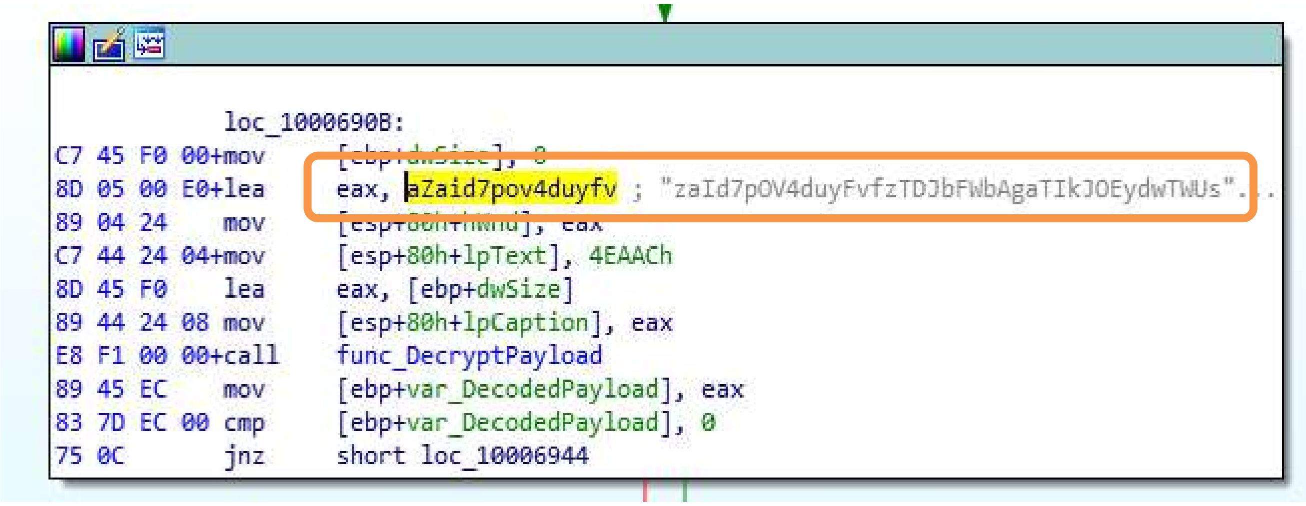
In IDA, firstly we examined the list of used strings and we found a suspicious path: "C:\\Python27\\DLLs_spcsrpt.py". It was used in a block of instructions which suggested that the path was a killswitch. If the particular file was present on the computer, the malware didn't execute.

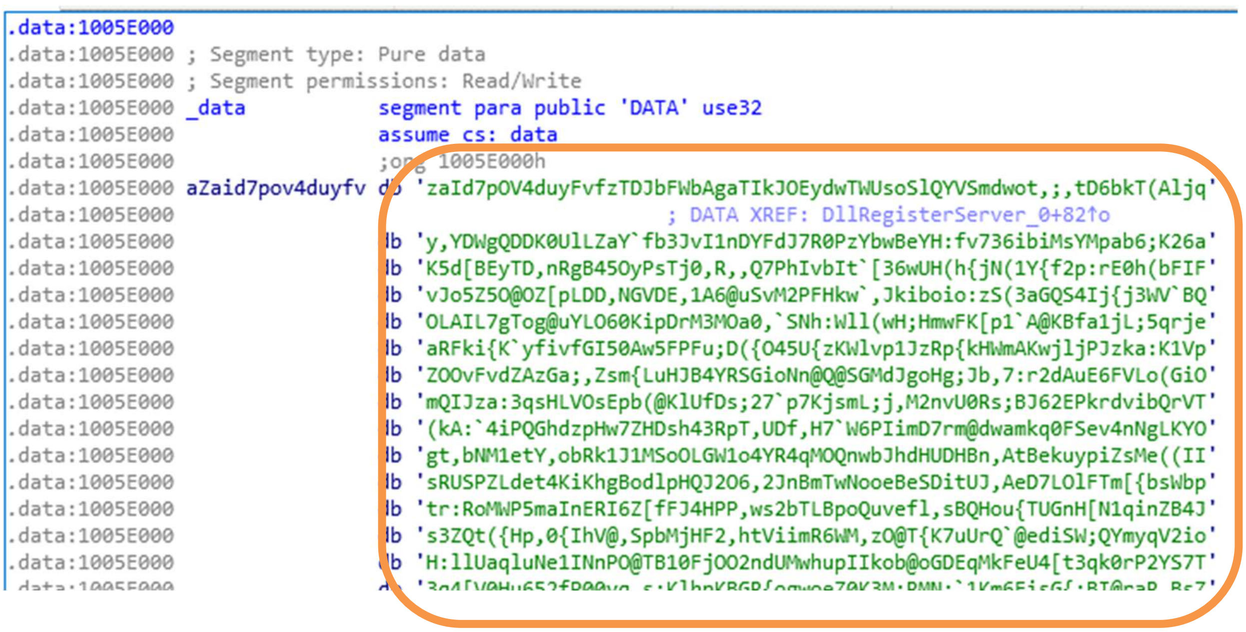
If the security team wanted to protect yet clean machines in the targeted organization from being infected with this particular sample, they could have created a python file within this path and the ransomware would not execute on the machine.



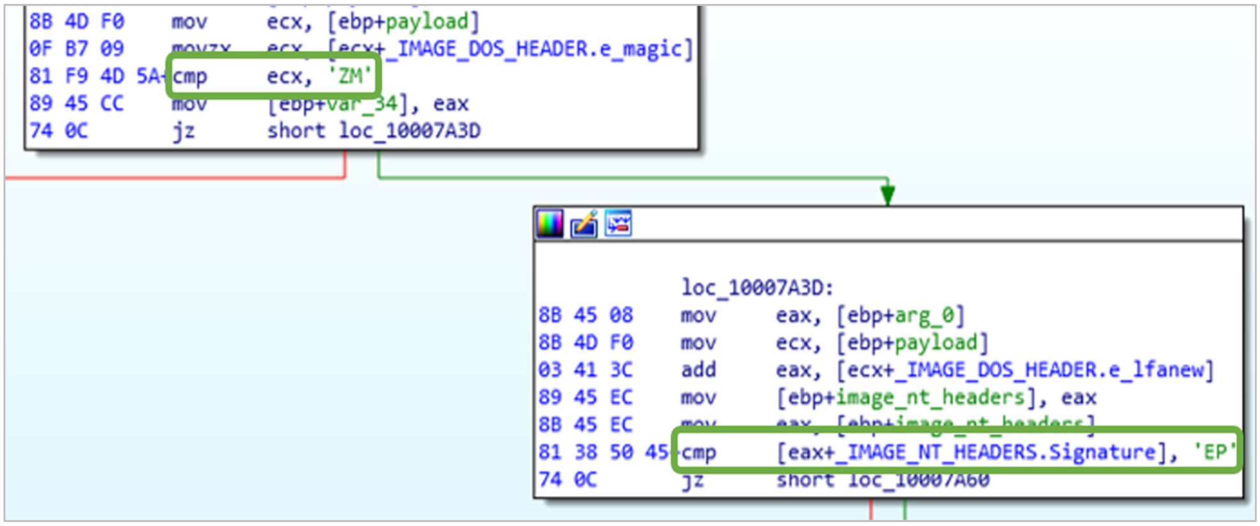
LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

Right after finding a killswitch, we examined one of clang.dll's exported functions – DLLRegisterServer. The thing that caught our eye was a string with the length of a few hundreds of kB whose memory address was saved into one of the registers. This string could be the packed malicious payload.





After examining successive functions, we found instructions that decrypted the payload and then parsed the PE header. You can notice on the screenshot, that the content of the register got compared to values 'ZM' and then 'EP'. If you realize that the order of letters was reversed due to endianness and that the register was in fact compared to the values 'MZ' and 'PE', you come to a conclusion that this particular code corresponded to parsing of PE header.



Another thing that the executable did, was the loading of DLLs and functions from them that were needed for further execution. You can notice such loading in disassembled code as two nested loops – one iterating through all the DLLs and the second one iterating through each function in a DLL.

The behavior we just described corresponds to a technique named Reflective DLL Loading.

Based on Hiew and capa output, our initial assumption was that the DLL was a reflective loader. We have confirmed this assumption through reverse enginnering in IDA.

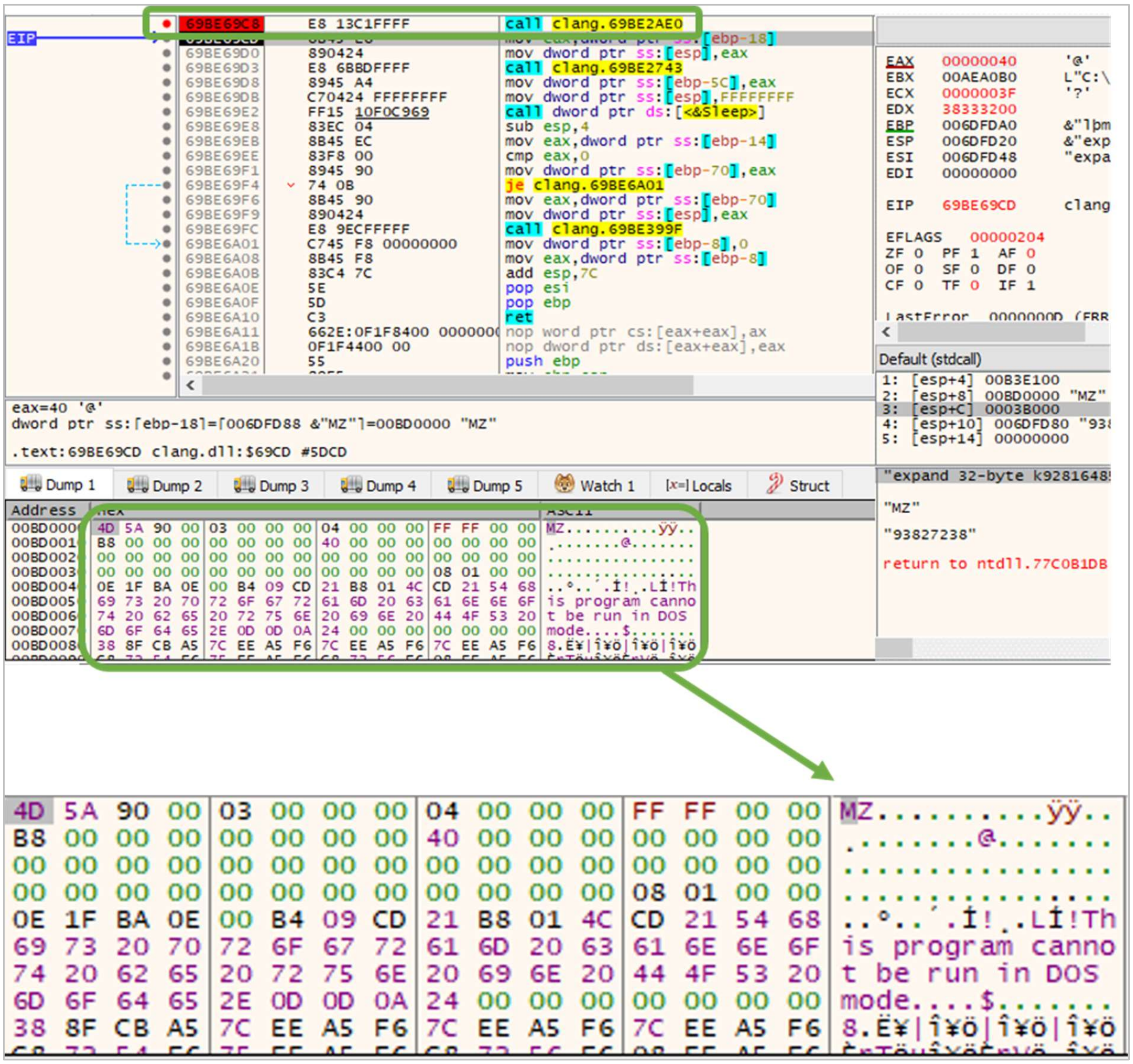
We wanted to get to the payload that it could unpack, load to the memory, and run. To accomplish this, we had to use IDA again and identify the address from which the function to unpack the payload was called. Moreover, we wanted to know to which register the memory address of the result got saved.

244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

X32dbg analysis

After finding the address of instruction that unpacked the payload, and register where the memory address of the result was saved, we opened clang.dll in x32dbg. We set the breakpoint to the correct address and executed the DllRegisterServer function from the malicious DLL until it hit the breakpoint.

Then we looked at the memory address saved in ecx and discovered the unpacked payload in this memory region. Here you can see the breakpoint and the unpacked payload (please notice the header characteristic for EXE and DLL files):



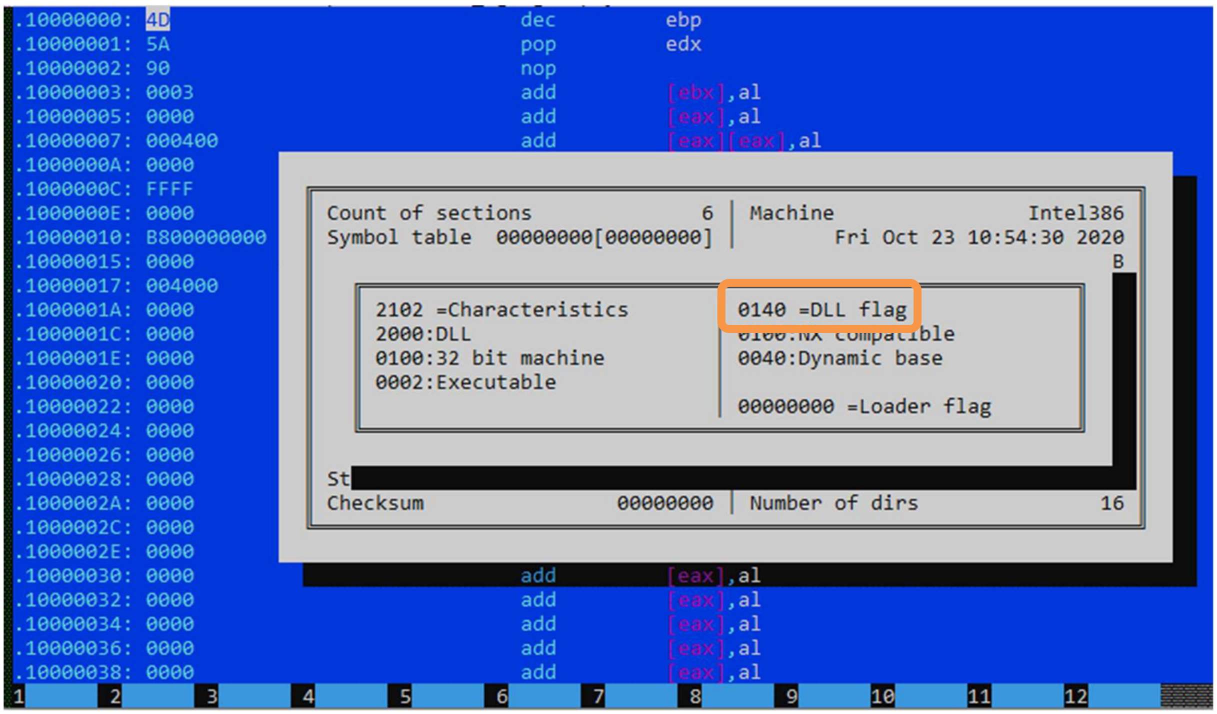
LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

Stage 2: payload1.dll (2nd Reflective DLL Loader)

We copied the extracted payload from the memory and saved it into a file. At that point it was not clear whether the payload was a DLL or an EXE. Moreover, the payload could be another loader or the malicious file-encrypting payload itself. To find answers to these questions, we examined it further through static analysis.

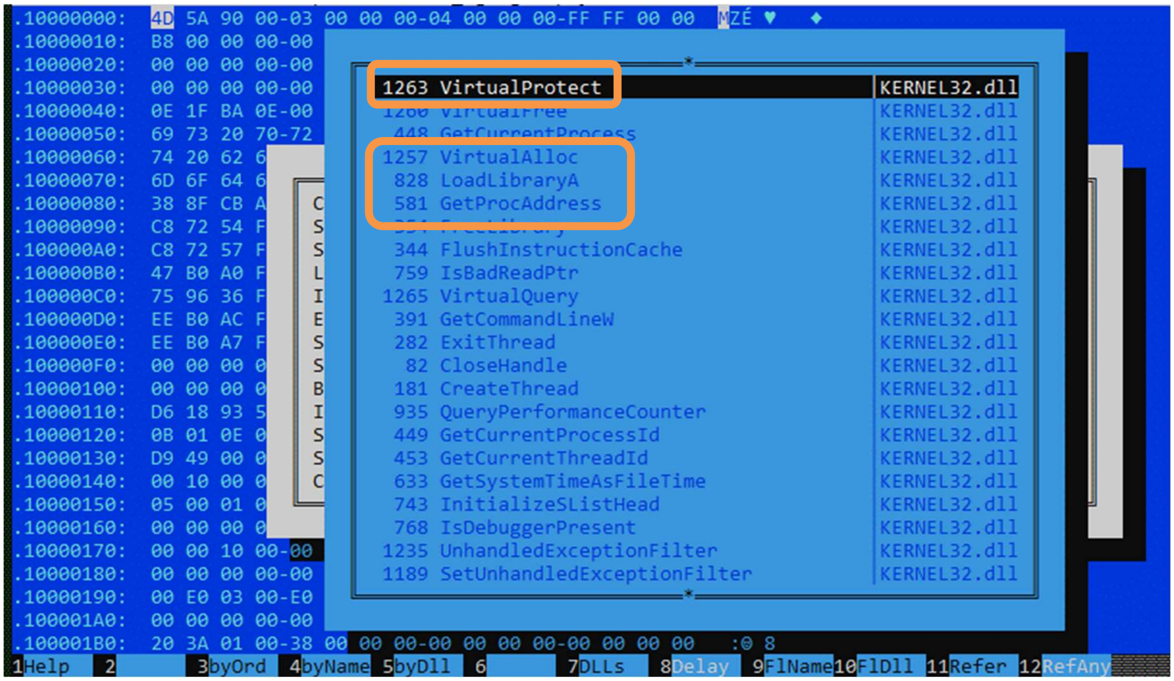
Hiew and capa analysis

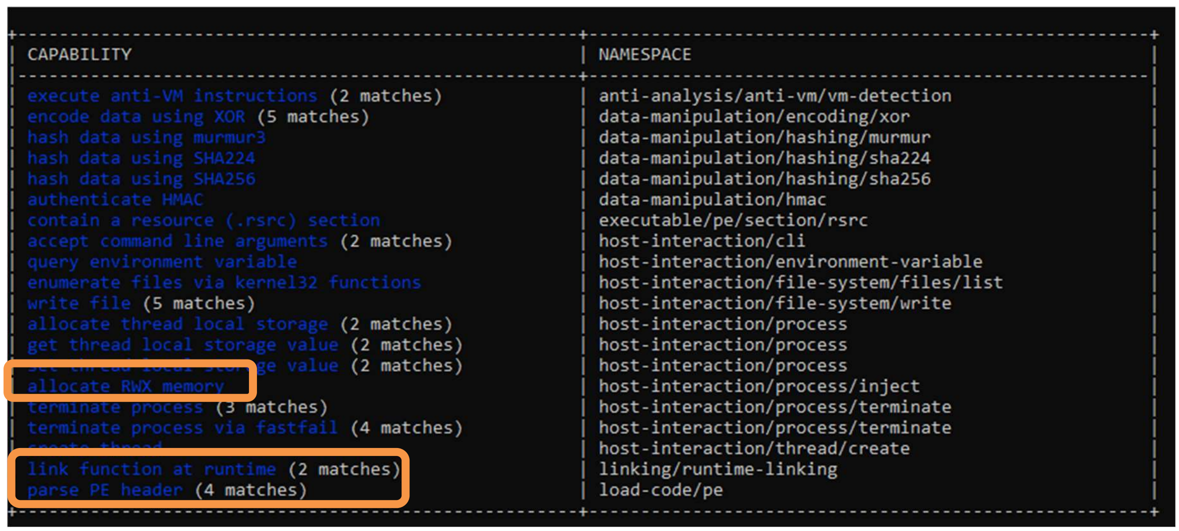
We examined the file in Hiew and saw it was a DLL and not EXE because it had a non-zero DLL flag.



Then we performed a process similar to the one we did with our first loader. We checked the imports of payload1.dll in Hiew and we scanned it with capa. It had imports and capabilities similar to the clang.dll, therefore we suspected that it was another loader.

LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com





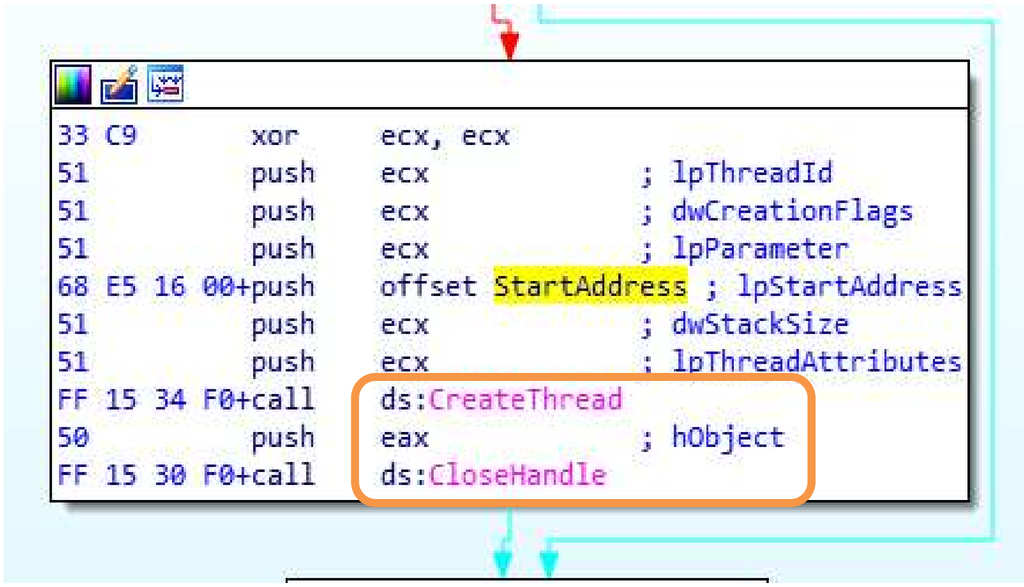
IDA analysis

We opened the DLL in IDA. We didn't notice anything interesting in the exported function DLLEntryPoint. However, we suspected the DLL to be a loader, therefore we moved on to imports, we chose an import GetProcAddress and we searched through its cross references.

We stumbled upon one call of GetProcAddress that had a surrounding code looking like a Reflective DLL Loader. It had two nested loops and it was called from a piece of code that parsed the MZ and PE headers. We do not include screenshots, because they were very similar to the previous loader.

After further examination, we saw, that this piece of code was called from a newly created thread.

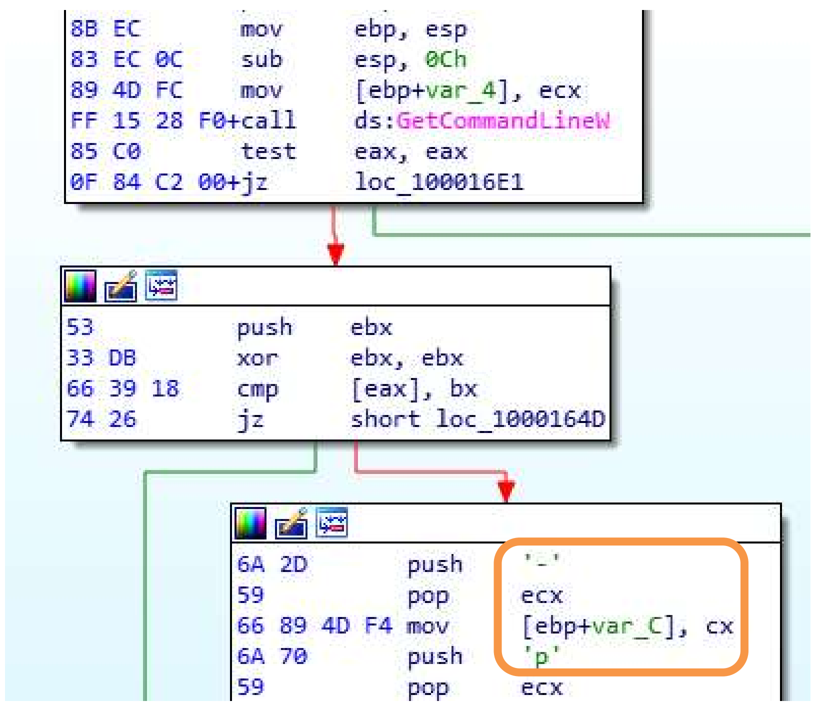
LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com



The –p parameter

However the most interesting thing we noticed was that the code checked if the DLL was called from a commandline with a parameter starting with –p. When we tried to execute the DLL just with parameter -p, it did not work and no payload was unpacked. What was actually needed was a specific word, probably unique for every target. The rest of the word after -p was a password to decrypt the payload.

This feature worked like another layer of protection because it ensured, that the payload was detonated only when the attacker decided to type the parameter into the commandline. We obtained the exact value of the parameter from our colleagues who performed forensic analysis in one of the targeted organizations.



LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

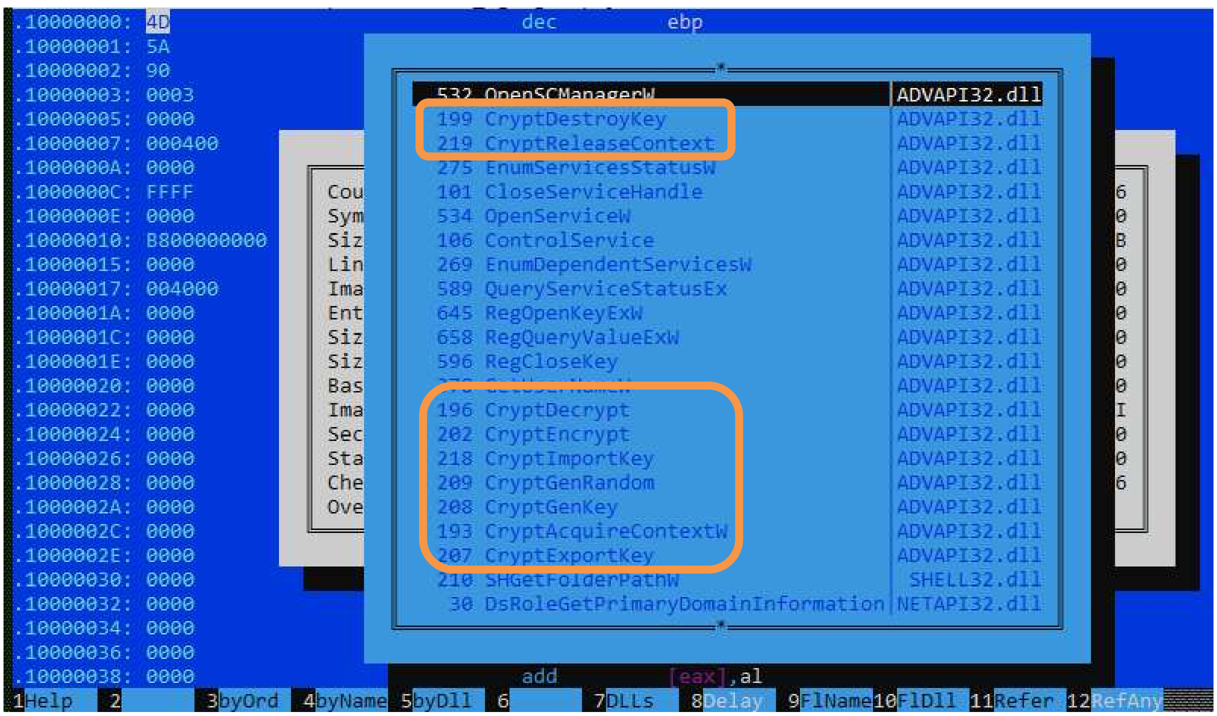
X32dbg analysis

Now that we had the password to decrypt the payload, we wanted to extract this payload from memory with the help of a debugger. We identified the address of a call that got executed right after the unpacked code was loaded in the memory. We proceeded to x32dbg set the breakpoint to this instruction and extracted the unpacked payload from the memory..dll (actual ransomware)

Stage 3: payload2.dll (actual ransomware)

Hiew and capa analysis

At the first sight, this payload looked different than the previous two DLLs. It did not reassemble a loader, but the file-encrypting payload itself. It was capable of using the WinCrypt library, generating keys, encrypting, decrypting, and had many capabilities that implied working with files (see the screenshots below).



LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

CAPABILITY	NAMESPACE
check for time delay via GetTickCount	anti-analysis/anti-debugging/debugger-detection
receive data	communication
connect network resource (2 matches)	communication/http
connect to HTTP server	communication/http/client
create HTTP request (2 matches)	communication/http/client
encode data using XOR (9 matches)	data-manipulation/encoding/xor
encrypt or decrypt via WinCrypt (3 matches)	data-manipulation/encryption
accept command line arguments (5 matches)	host-interaction/cli
get common file path (4 matches)	host-interaction/file-system
delete file (9 matches)	host-interaction/file-system/delete
enumerate files via kernel32 functions (9 matches)	host-interaction/file-system/files/list
get file attributes (5 matches)	host-interaction/file-system/meta
get file size (3 matches)	host-interaction/file-system/meta
set file attributes	host-interaction/file-system/meta
move file	host-interaction/file-system/move
read file (2 matches)	host-interaction/file-system/read
write file (2 matches)	host-interaction/file-system/write
get disk information (5 matches)	host-interaction/hardware/storage
get disk size	host-interaction/hardware/storage
create mutex (2 matches)	host-interaction/mutex
get hostname (6 matches)	host-interaction/os/hostname
get thread local storage value	host-interaction/process
set thread local storage value	host-interaction/process
terminate process (7 matches)	host-interaction/process/terminate
terminate process via fastfail (3 matches)	host-interaction/process/terminate
query registry entry	host-interaction/registry/query
query registry value (3 matches)	host-interaction/registry/query
query service status (2 matches)	host-interaction/service
enumerate services (3 matches)	host-interaction/service/list
get session user name (4 matches)	host-interaction/session
create thread (17 matches)	host-interaction/thread/create
link function at runtime	linking/runtime-linking
parse PE header (5 matches)	load-code/pe

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information [T1027]
DISCOVERY	File and Directory Discovery [T1083]
	Query Registry [T1012]
	System Information Discovery [T1082]
	System Owner/User Discovery [T1033]
	System Service Discovery [T1007]
EXECUTION	Shared Modules [T1129]

MBC Objective	MBC Behavior
ANTI-BEHAVIORAL ANALYSIS	Debugger Detection::Timing/Delay Check GetTickCount [B0001.032]
COMMAND AND CONTROL	C2 Communication::Receive Data [B0030.002]
COMMUNICATION	HTTP Communication::Connect to Server [C0002.009]
	HTTP Communication::Create Request [C0002.012]
	HTTP Communication::Get Response [C0002.017]
CRYPTOGRAPHY	Decrypt Data [C0031]
	Encrypt Data [C0027]
DATA MANIPULATION	Encoding::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]

LIFARS
your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

Highlights from IDA analysis

As the next step we could perform an in-depth analysis of the payload, similar to the analysis from Minerva report. We could examine the configuration of ransomware, possible killswitch, its ability to contact C&C servers, detailed process of generating keys and encrypting files, etc.

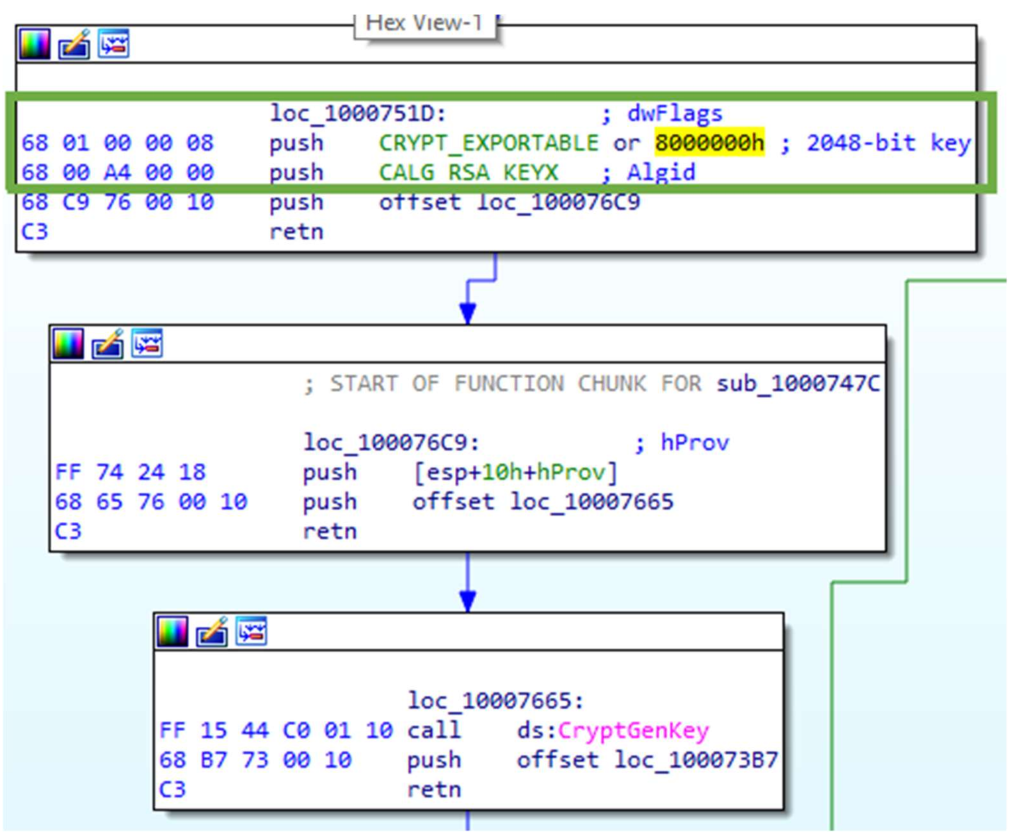
However, the main aim of our report was to examine the process of Reflective DLL Loading and to unpack the payload and extract it from the memory. Therefore, in this section we provide only some highlights from determining one of the used encryption algorithms.

If you would like to learn more details about all encryption algorithms that the ransomware uses, you can read it in the aforementioned Minerva report.

Determining the Encryption Algorithm

To determine the encryption algorithms used by ransomware, we examined cross references of a CryptGenKey function which we found in imports. According to MSDN documentation, the second parameter of the CryptGenKey function should be Algid, which stands for algorithm id. Therefore, we were looking for a parameter that was used in the second push instruction before the call of CryptGenKey.

IDA helped us with this task and added comments containing the names of parameters it expected to find in variables pushed to the stack. The value of Algid parameter was CALG_RSA_KEYX. Moreover, from dwFlags parameter we can see that key size is set to 2048 bits (the upper 16 bits, 0x0800). This value revealed that the ransomware used RSA-2048 in the process of encryption.



LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

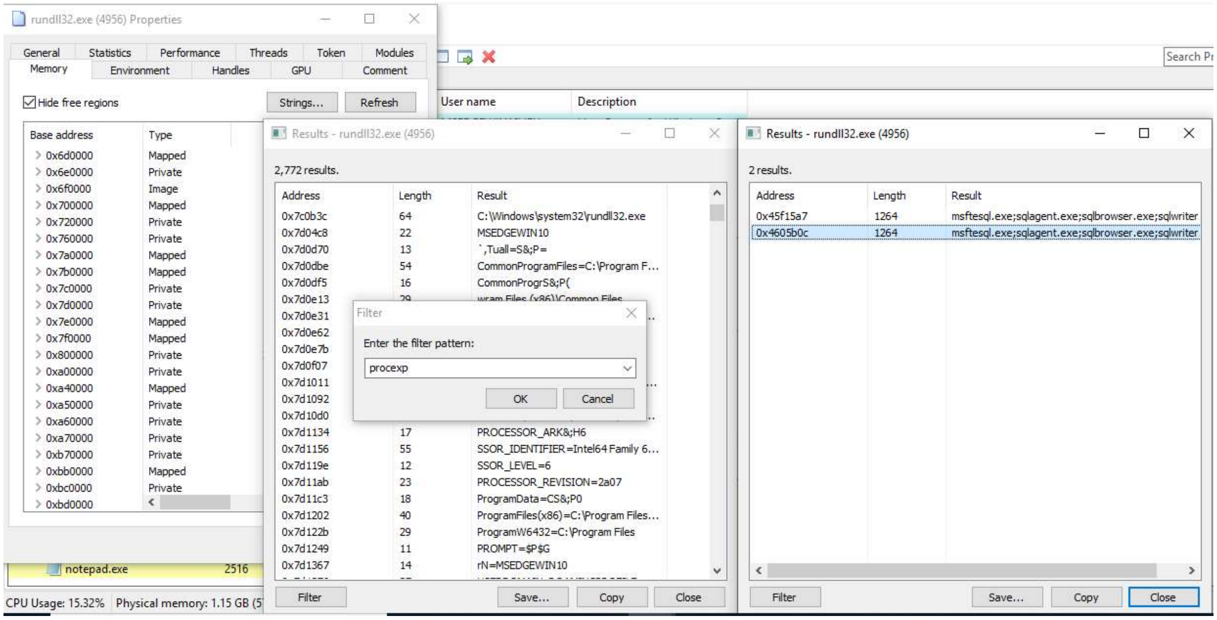
Dynamic Analysis

Finally, we executed the ransomware in a sandbox to perform a dynamic analysis. This is how an Egregor ransom note looked like (we have removed some client-specific details for privacy reasons):

```
------------------
| What happened? |
------------------

Your network was ATTACKED, your computers and servers were LOCKED,
Your private data was DOWNLOADED.

----------------------
| What does it mean? |
----------------------

It means that soon mass media, your partners and clients WILL KNOW about your PROBLEM.

--------------------------
| How it can be avoided? |
--------------------------

In order to avoid this issue,
you are to COME IN TOUCH WITH US no later than within 3 DAYS and conclude the data recovery and breach fixing AGREEMENT.

-------------------------------------------
| What if I do not contact you in 3 days? |
-------------------------------------------

If you do not contact us in the next 3 DAYS we will begin DATA publication.

-----------------------------
| I can handle it by myself |
-----------------------------


--------------------------
| You have convinced me! |
--------------------------

Then you need to CONTACT US, there is few ways to DO that.

I. Recommended (the most secure method)

   a) Download a special TOR browser: https://www.torproject.org/
   b) Install the TOR browser
   c) Open our website with LIVE CHAT in the TOR browser: http://
   d) Follow the instructions on this page.

II. If the first method is not suitable for you

   a) Open our website with LIVE CHAT: https://egregor.top/
   b) Follow the instructions on this page.

Our LIVE SUPPORT is ready to ASSIST YOU on this website.

----------------------------------------
| What will I get in case of agreement |
----------------------------------------

You WILL GET full DECRYPTION of your machines in the network, FULL FILE LISTING of downloaded data,
confirmation of downloaded data DELETION from our servers, RECOMMENDATIONS for securing your network perimeter.

And the FULL CONFIDENTIALITY ABOUT INCIDENT.
```


LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

We wanted to examine the ransomware through procexp. However, procexp was killed immediately after the ransomware started executing. Therefore, we launched the Process Hacker to examine the ransomware. We opened Properties of our ransomware process, opened the Memory tab, extracted the Strings, and searched for "procexp" in these strings. We found a rather long list of programs that this ransomware kills whenever it gets executed. You can see the complete list under the screenshot.



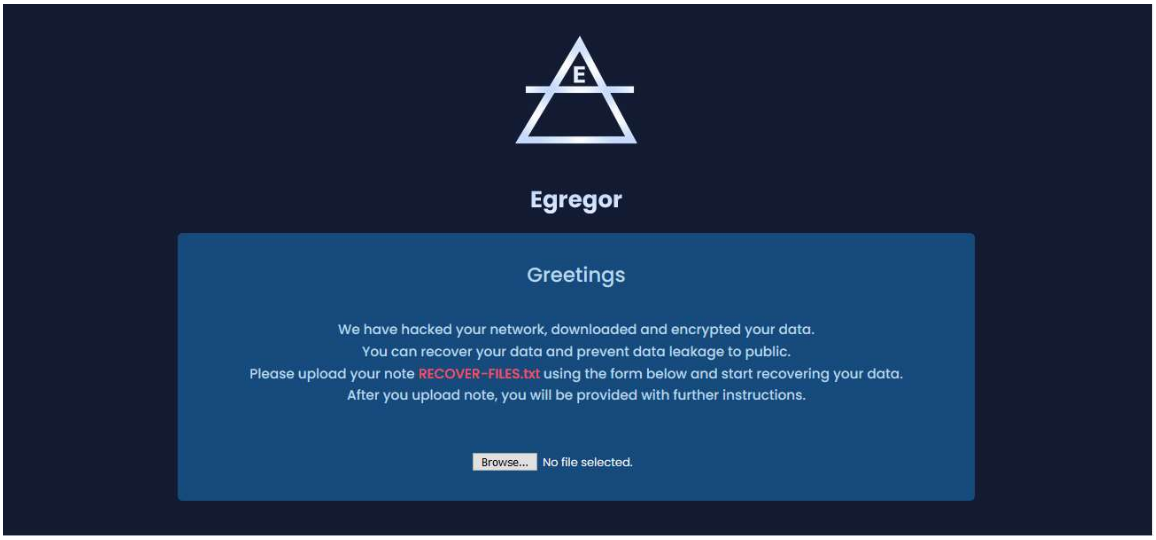

```
msftesql.exe;sqlagent.exe;sqlbrowser.exe;sqlwriter.exe;oracle.exe;ocssd.exe;d
bsnmp.exe;synctime.exe;agntsvc.exe;isqlplussvc.exe;xfssvccon.exe;sqlservr.exe
;mydesktopservice.exe;ocautoupds.exe;encsvc.exe;firefoxconfig.exe;tbirdconfig
.exe;mydesktopqos.exe;ocomm.exe;mysqld.exe;mysqld-nt.exe;mysqld-opt.exe;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


LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

OSINT

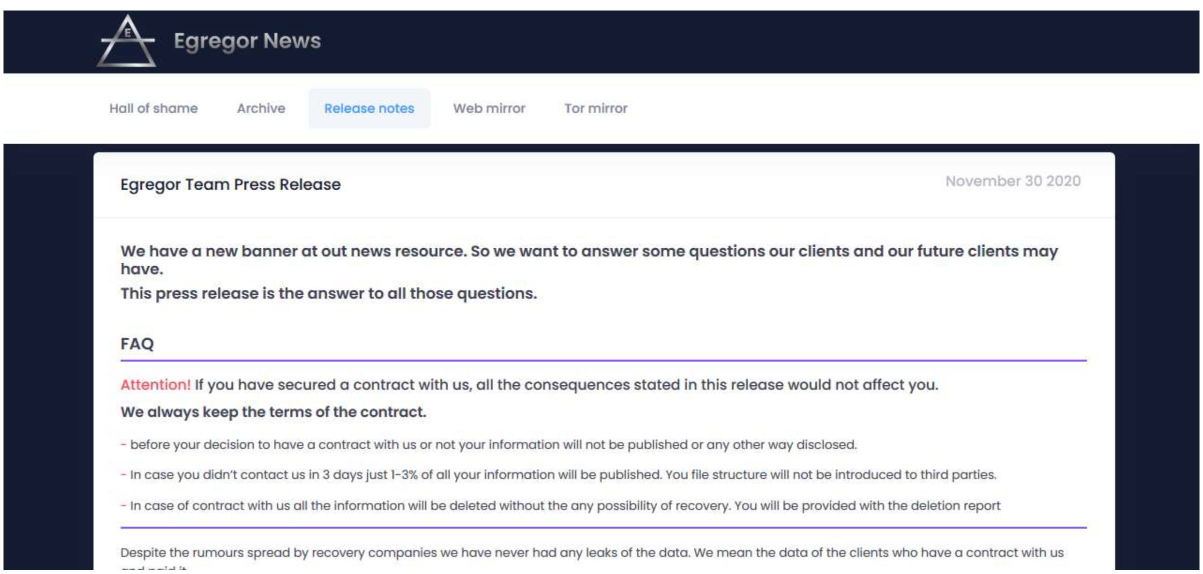
In the ransom note there were two websites with further instructions for the victim. One was an .onion webpage with a hostname specifically crafted for the victim and the second one was a regular webpage in the form hxxps://egregor[.]top hostname-from-onion-webpage.

We visited the webpages from the ransom note. They allowed the victim to upload the ransom note and get instructions from the Egregor creators on how to proceed with the payment. This is how the webpage looked like.



There was also another .onion domain containing a listing of all the companies hacked by Egregor –the list was called a Hall of Shame. Next to every company name, there was information about the percentage of the company's disclosed data. With high probability, this data was exfiltrated during the process of infection with Egregor. The disclosed data was also available for download on this .onion webpage.

Moreover, there was a special category – Hole of the Month – that included two large game companies. Also, a warning was present – a PS notice to think about possible backdoors in products of these companies.



LIFARS
your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com

CONCLUSION

In this case study, we showed the basic concepts of hybrid malware analysis. We used Hiew, capa, and IDA for static analysis and reverse engineering, x32dbg for debugging, and we also ran the malware in a sandbox and performed dynamic analysis.

The malware consisted of three stages – two loaders and one actual payload. Both loaders used a technique called Reflective DLL Loading. Reflective DLL Loader opens the target process with read, write, execute permissions, loads the malicious DLL, and calls its entry point.

The execution of the initial malicious DLL had to be invoked with a specific parameter, otherwise the payload wasn't unpacked. The attacker had to type this parameter into commandline manually. As our reverse engineering revealed, this parameter had to start with –p and the rest of the parameter served as a password to correctly decrypt the payload.

The payload used RSA and ChaCha for encryption. It killed multiple processes – some of those killed processes belonged to different forensic tools and some could protect their data from encryption when they had this data opened at the time the malware executed itself.

We also found a Python file that could be used as a killswitch. However, its name and path is probably unique for every sample.

The Egregor ransomware shows some similarity with Maze ransomware. Both ransomwares use very similar types of obfuscation.

LIFARS your digital world, secured
244 Fifth Avenue, Suite 2035, New York, NY 10001
LIFARS.com (212) 222-7061 info@lifars.com