Reverse Engineering Dridex and Automating IOC Extraction

appgate.com/blog/reverse-engineering-dridex-and-automating-ioc-extraction

Dridex[1] is a major banking trojan that appeared somewhere around 2011, continually evolving ever since.
The APT (Advanced Persistence Threat) known as TA505[2] is associated to Dridex, as well as with other infamous malware such as TrickBot and Locky ransomware.

Once installed, Dridex can download additional files to provide more functionality to the trojan. Simply put, there are four main components:

Downloader	Infected Microsoft Office documents that downloads or drops Dridex's Loader
Loader	Responsible for downloading and installing the main module.
Bot	This is the main module, which contains most of Dridex's functionalities.
Modules	Additional files that provides more functionalities, downloaded by the Loader.

In the last few days, our team detected recent Dridex samples through our live hunting process and, after analyzing them, we've decided to publish an analysis of the main features implemented by this threat that makes detection and analysis difficult.

Therefore, in this post, we will focus on the second layer (Loader), showing the technical details regarding:

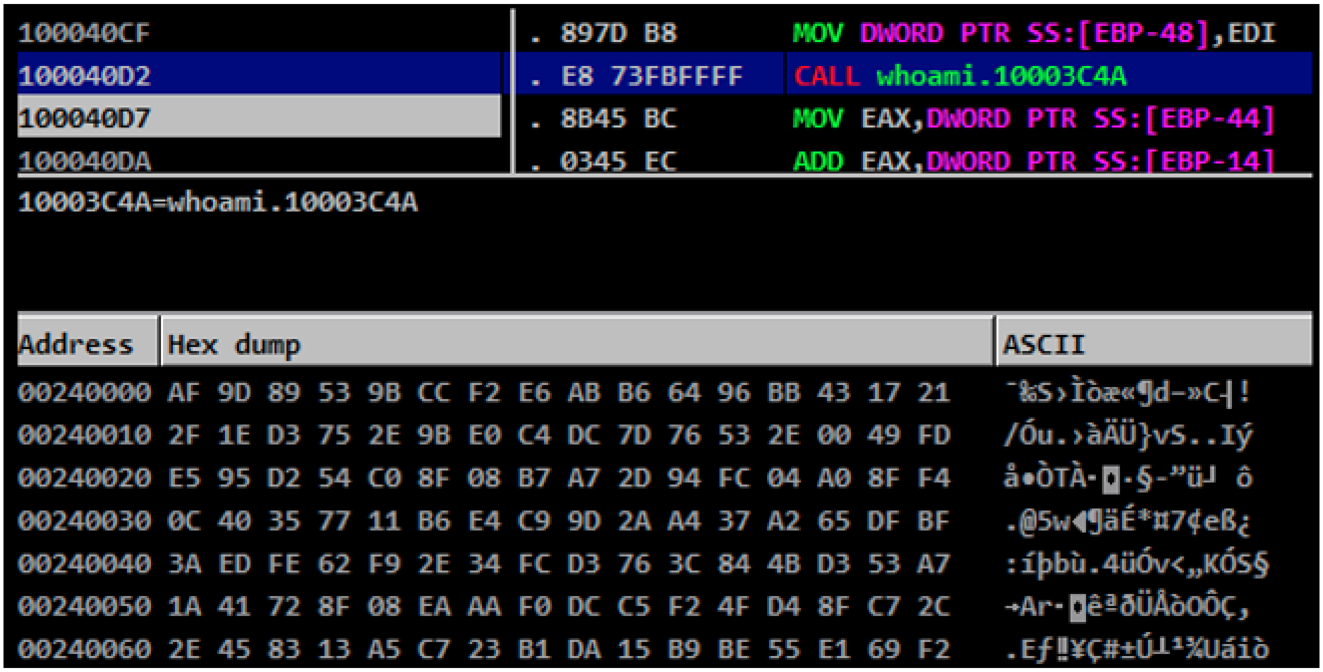
1. Unpacking;
2. Dynamic API Calls;
3. Encrypted Strings;
4. C2 Network Communication.

In addition, we are publishing a tool that statically extracts IOCs from the latest Dridex binaries, as we believe that this can enable organizations to reduce analysis time spent during incidents or prevent the malware family altogether.

1. Unpacking

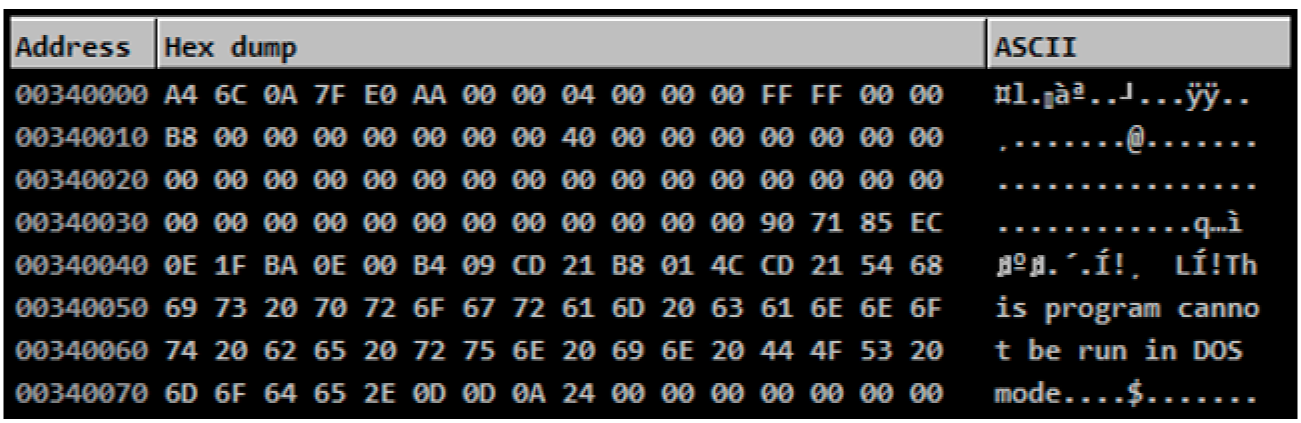
Unpacking is an important step because Dridex doesn't write the unpacked payload to disk, instead, the custom packer loads it directly to memory and there it stays. First, to analyze and extract the IOCs, we must get the unpacked sample, and this is not a difficult task.

You can easily execute the packed sample and then run the amazing PE-sieve[3] tool, from hasherezade, which will extract the payload from memory. However, if you are curious like us and wants to understand how it works, the first step in the unpacking process is the allocation of an encrypted content into memory:



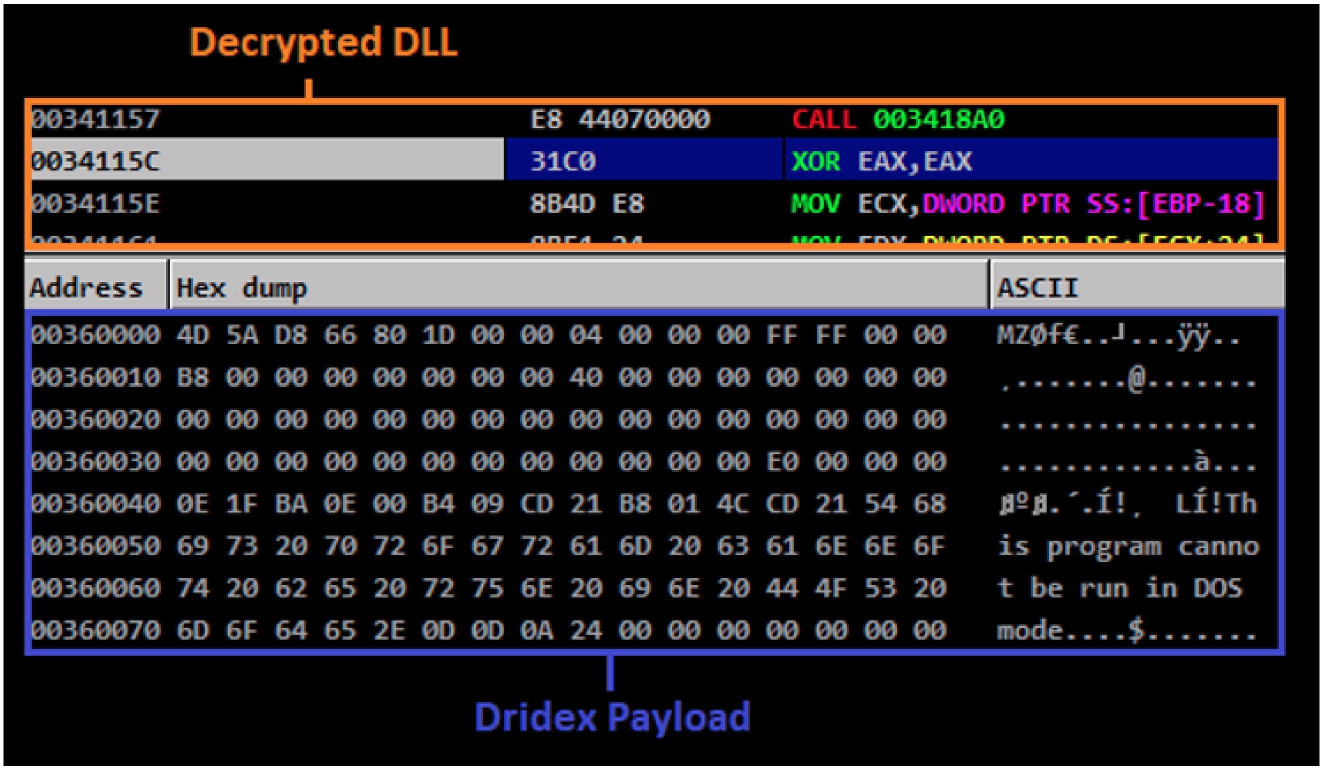
Encrypted DLL in Memory.

This data is an encrypted small DLL that is responsible for unpacking the Dridex loader. Also, we noticed that the magic bytes (MZ) for the MS-DOS header was not present, probably to avoid automatic filetype identifications.



Decrypted DLL in Memory.

After the decryption process, the main executable transfers the execution to the allocated DLL, which unpacks the Dridex Loader and then replaces the main executable code with the payload's code.

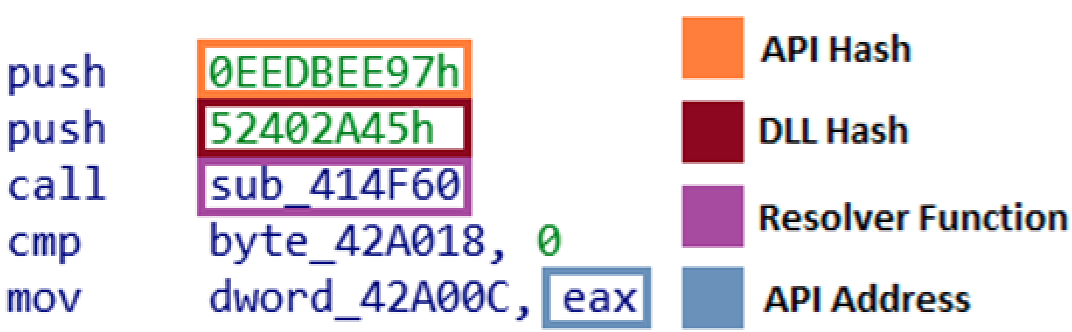


Dridex Payload in Memory.

2. Dynamic API Calls

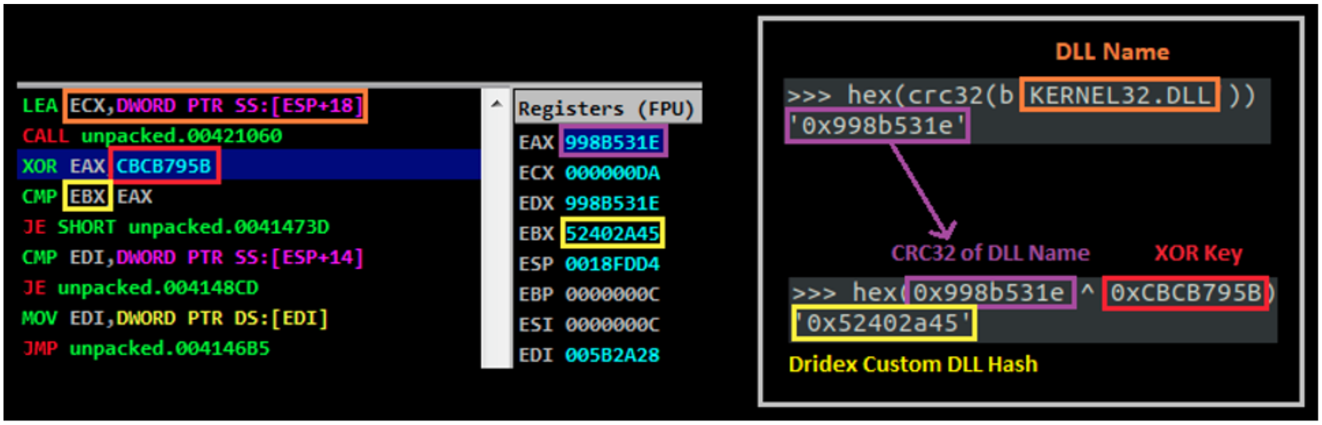
Dridex doesn't have an import table like regular PE files. Instead, all the API calls are dynamically resolved by the malware using a custom technique to avoid detection by APIs and to make reverse engineering more difficult.

To resolve an API, it calls a "resolver" function passing two parameters, both custom hashes. The first one is a hash for the DLL name that contains the API, and the second one is the hash of the API name that Dridex needs to resolve.



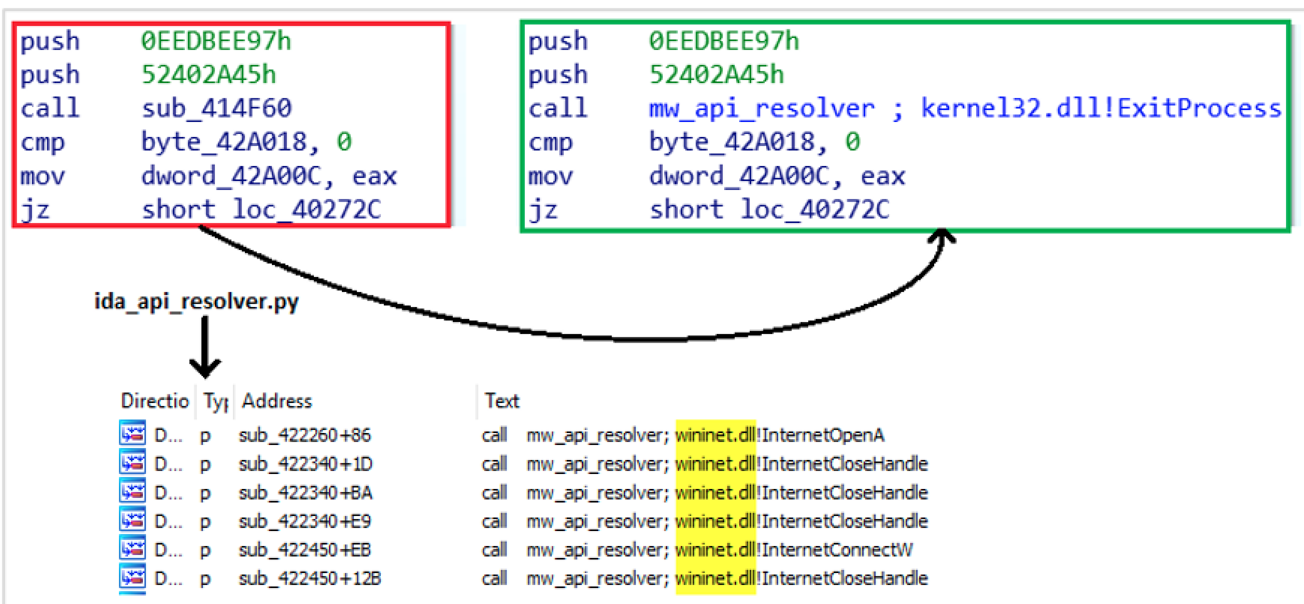
The way the API is resolved isn't trivial, but in summary, if the API wasn't already resolved by the malware, it parses the DLL linked list, located in the process' PEB (Process Environment

Block), generating a CRC32 hash for each DLL and API name and compares that with the ones that was passed into the function. Aside from the CRC32 hash, the value is also “xored” with a custom key, making the values unique on each Dridex sample.



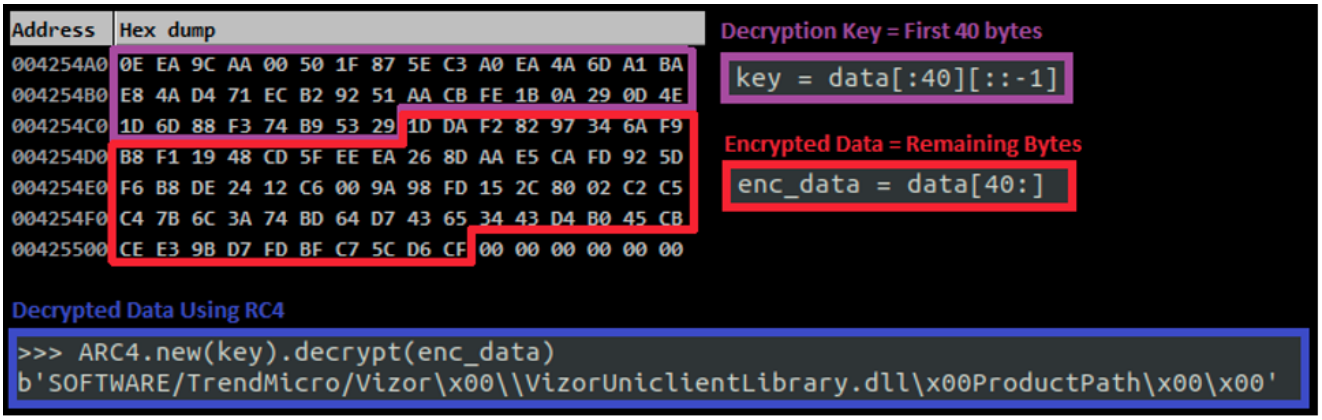
Dridex Searching for DLL Using Custom Hash.

Therefore, in the example above, the first hash stands for “kernel32.dll”. Using this same logic, we created a small IDA script which resolves API calls automatically and inserts a comment where the function is called, so we can easily search where in the code certain DLLs or APIs are being used.



Block with Encrypted Strings.

Each chunk contains one or more strings that are encrypted with RC4, where the key is the first 40 bytes (in little-endian format) of each block.



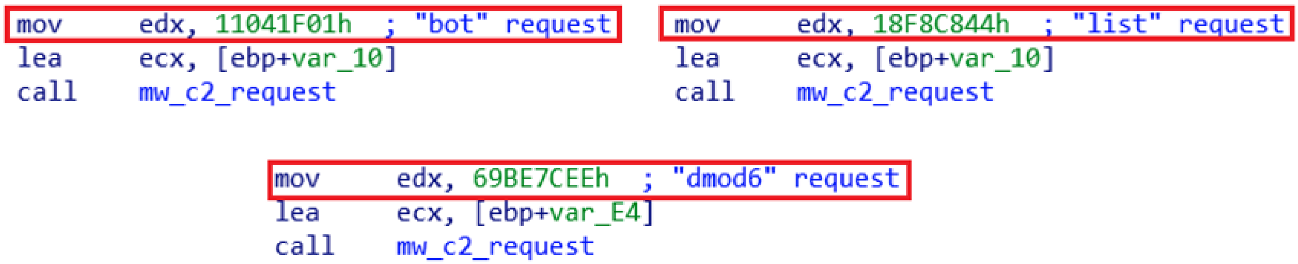
Decrypting Dridex Strings.

Aside from the encrypted strings, we also found plain text strings in this recent Loader that were not present in older versions, which can be used for identification (Yara rule):

- bot
- vnc
- socks
- uacme
- list
- cve-2015-0057 (mod5)
- TrendMicro (mod9)
- NetChecker (mod10)
- rep: DllLoaded (dmod5)
- rep: DllStarted (dmod6)
- rep: NetGood (dmod7)
- rep: NetFail (dmod8)
- rep: NetPart (dmod9)
- rep: StartedInLo (dmod10)
- rep: StartedInHi (dmod11)

4. C2 Network Communication

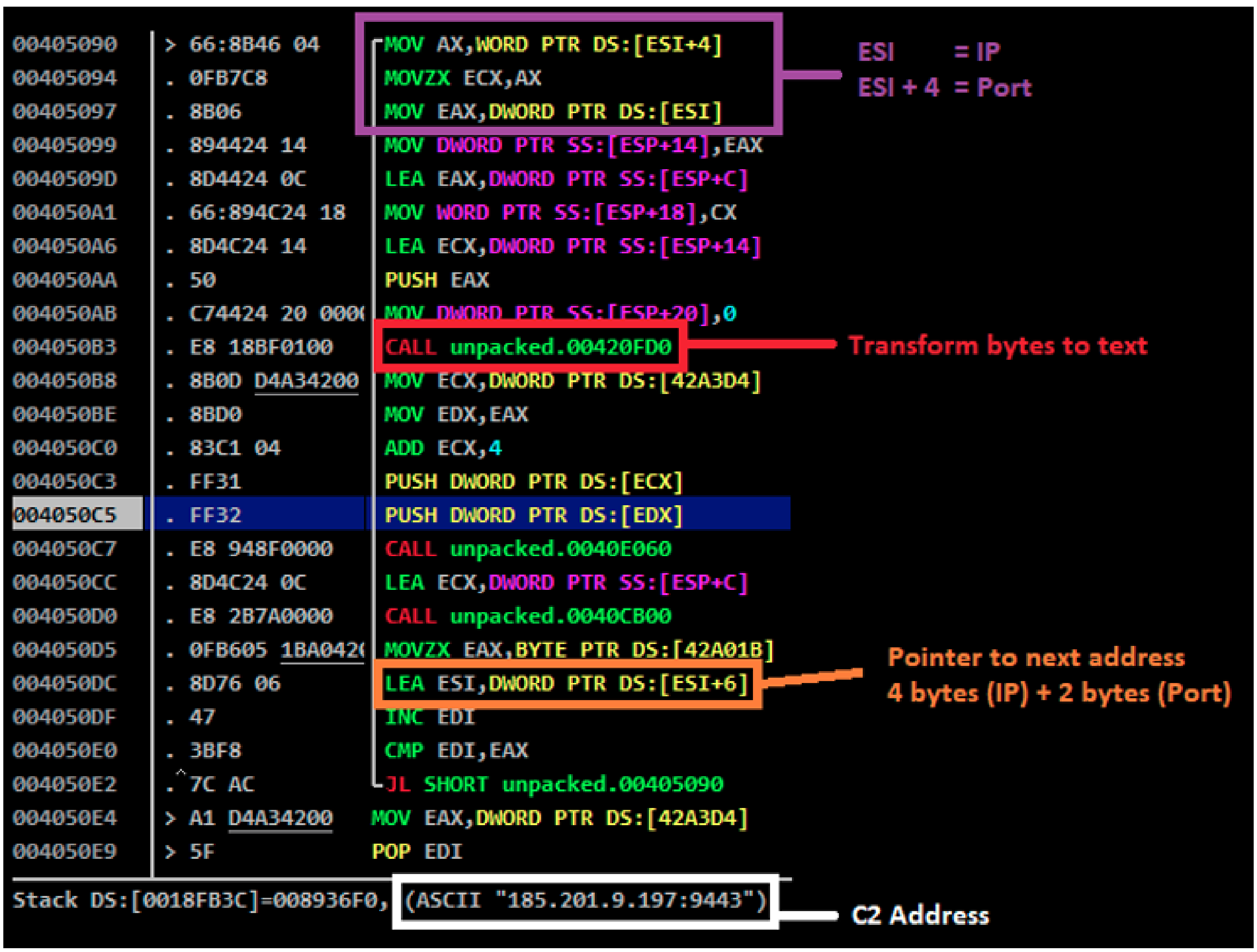
The loader is responsible for the initial C2 communication and for downloading additional files, such as the bot and the modules. Analyzing the function that does such communication, we could see that the command sent to the C2 is passed as a parameter, and it's the CRC32 hash of the command string.



Function for C2 Communication.

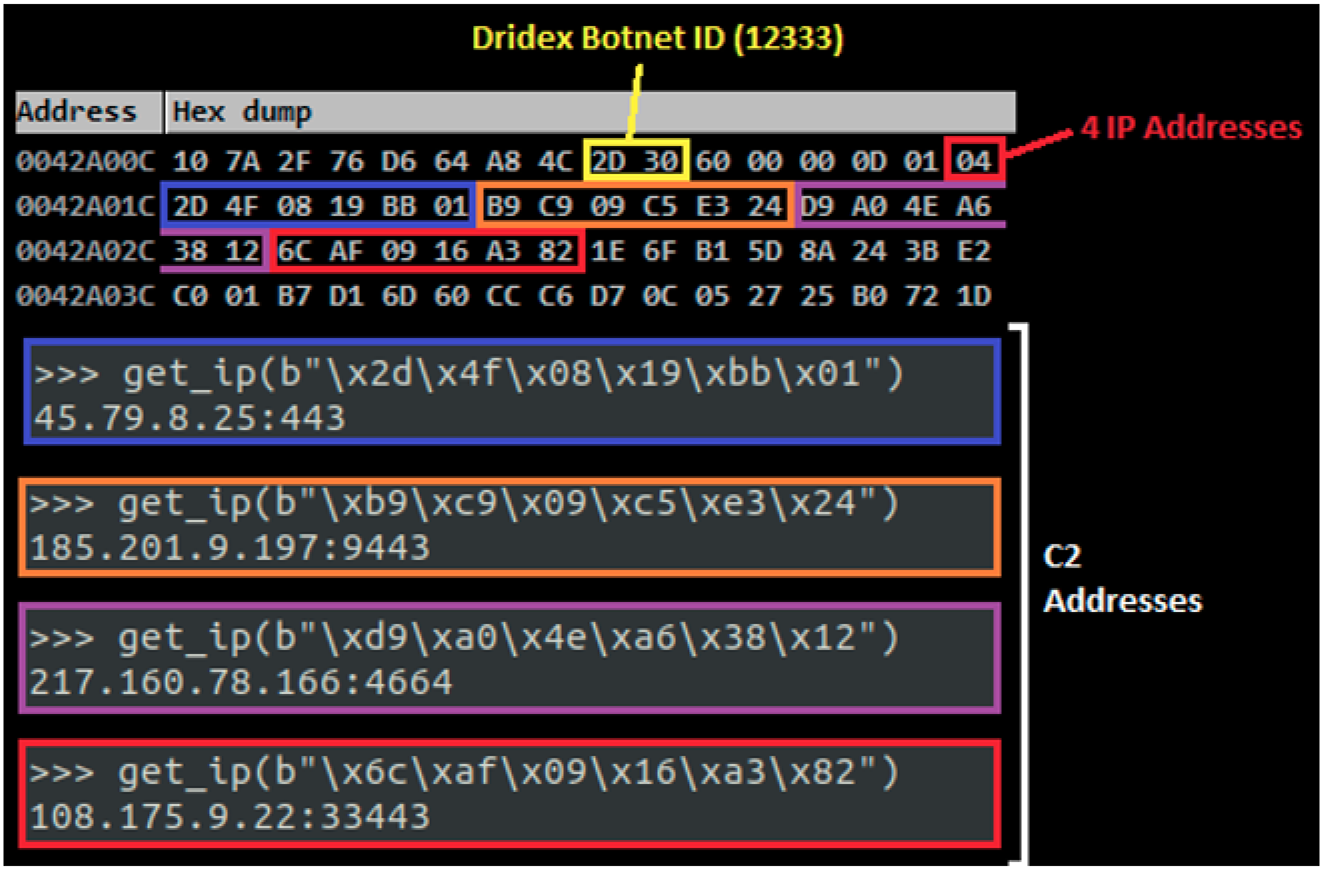
Curiously though, that despite using the hashes of the commands in the communication, this specific sample has the same commands in the strings, as mentioned above, such as the "list" that requests a list of IPs to connect or the "bot" that is the command to download the next stage.

Before doing the request, the botnet ID and the C2 addresses are parsed.



Dridex Parsing Command & Control IPs

This information is stored in the PE “.data” section, represented in bytes, along with the Dridex botnet ID.

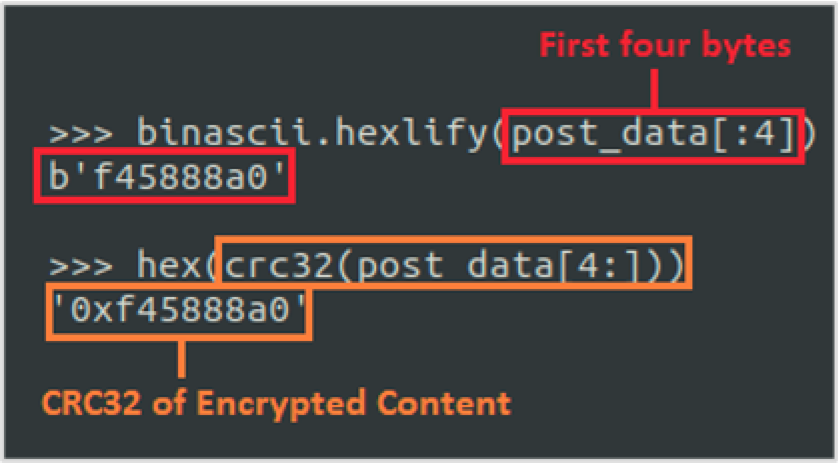


Dridex C2 IPs.

Once these addresses are parsed, Dridex then sends a POST request to one of the C2 addresses and, bypassing the SSL, we could see that the data is encrypted. If the server doesn’t respond as expected, Dridex continues to send the same content to the other IPs.

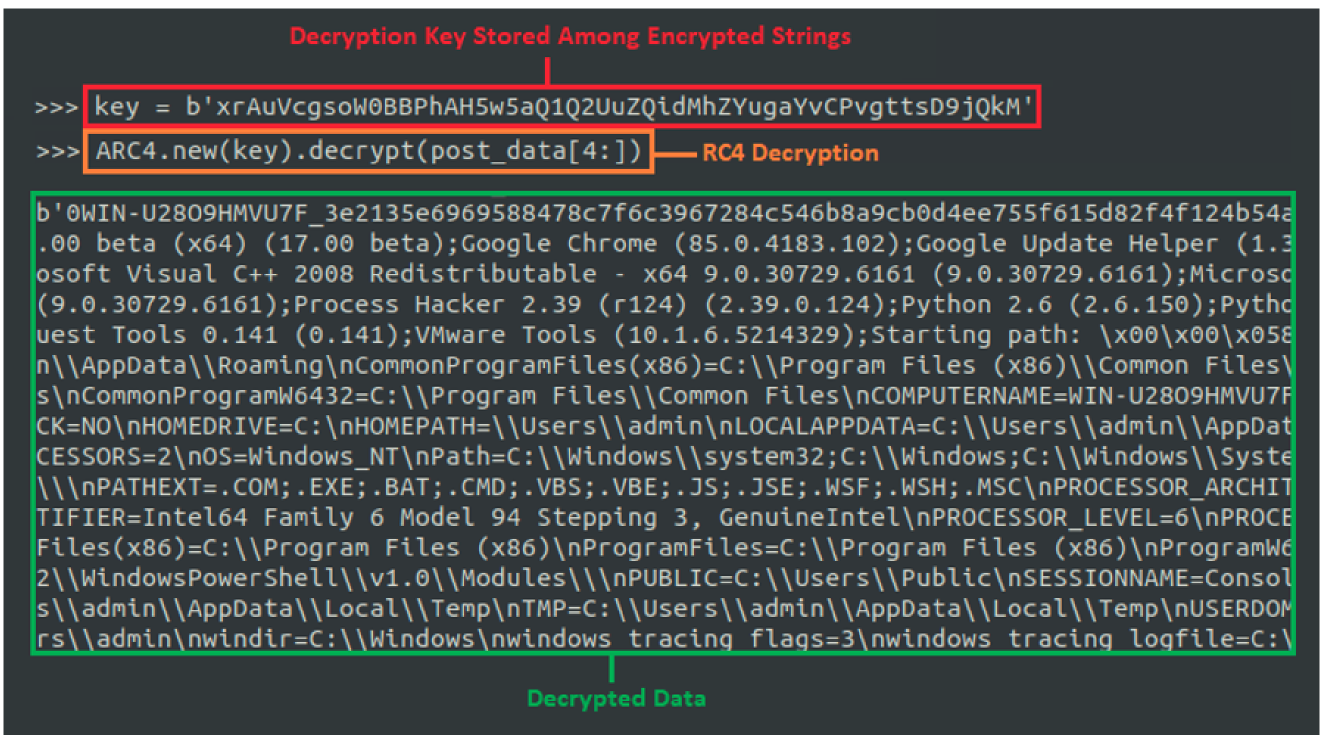
Dridex POST Request to C2.

After analyzing the function, we found that the malware uses the first 4 bytes as a checksum for the encrypted bytes, with CRC32 hash.



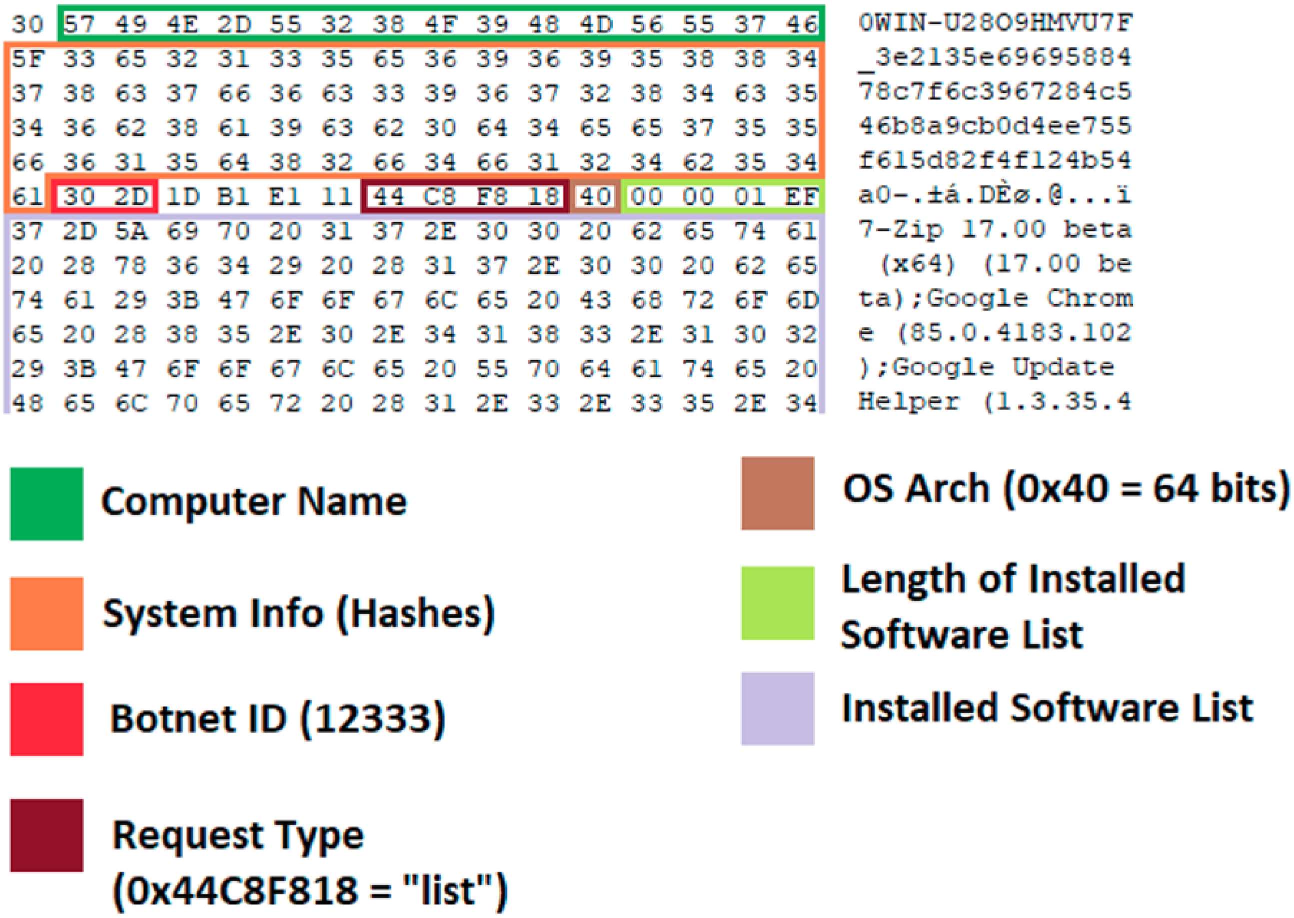
Encrypted Network Data Checksum

If the checksum matches, the data can be decrypted correctly. The algorithm used by Dridex is RC4 and the encryption/decryption key is stored among Dridex decrypted strings.



Decrypted Network Data.

The image below illustrates some of the main fields used in the first POST request made by this Dridex sample.



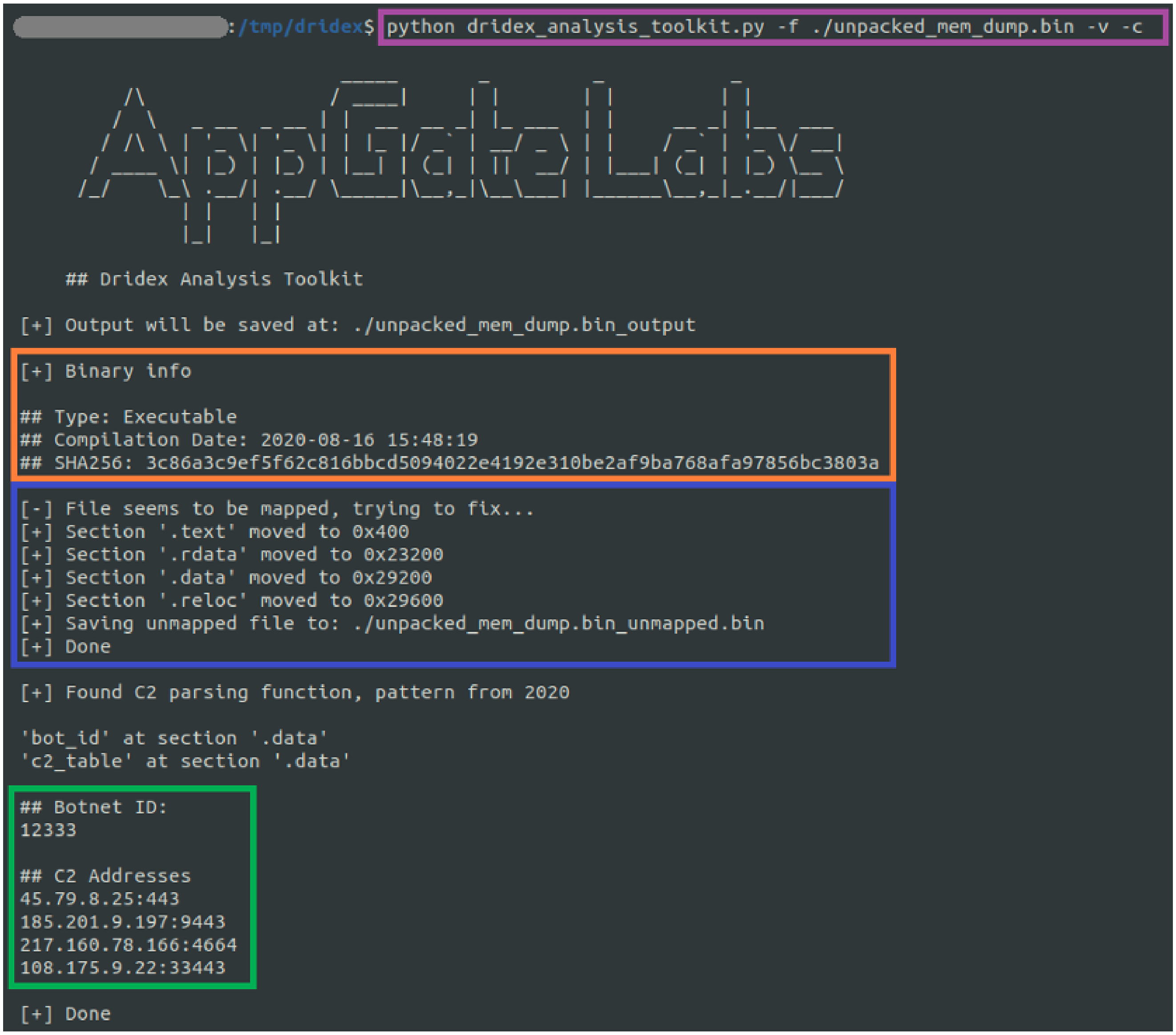
Data Sent by Dridex to C2.

5. Automating IOC Extraction

Along with this blog post, we are releasing a python script that automates the IOC extraction from the Dridex loader. These are the main features implemented by our “Dridex Analysis Toolkit”:

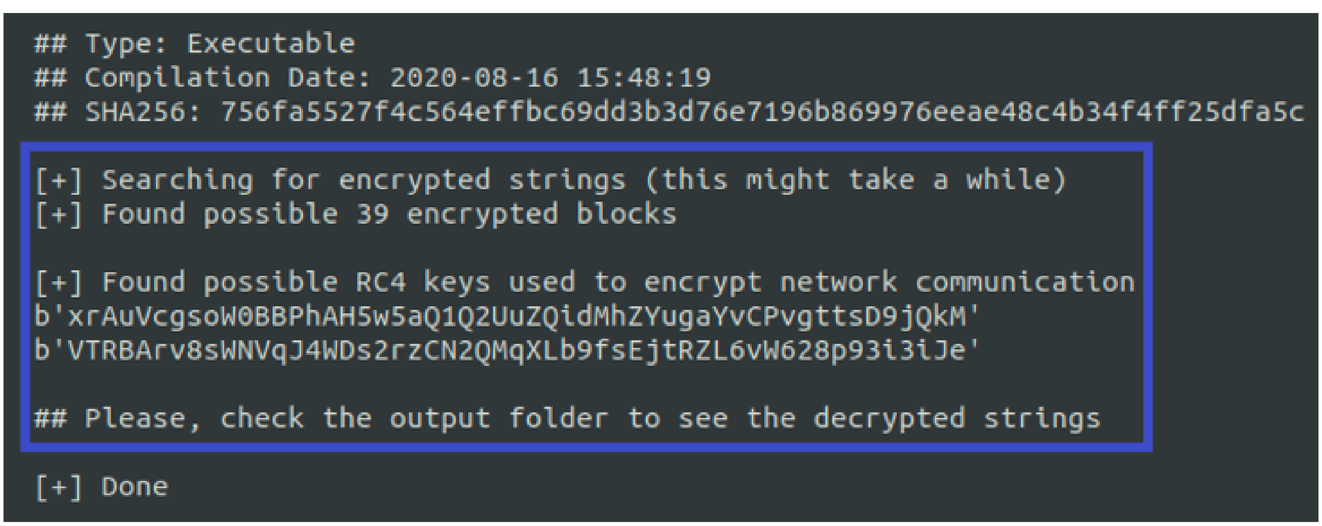
- Extract Botnet ID;
- Extract C2 IP Addresses;
- Decrypt Strings;
- Decrypt Network Communication.

Furthermore, since most Dridex payloads comes from memory dumps, the script also tries to unmap the PE file to disk, so it can get the right offsets to parse, example:



Extracting C2 Addresses from Dridex Payload Automatically.

By using the “-s” option, the script searches and decrypts the payload strings and prints any possible RC4 keys:

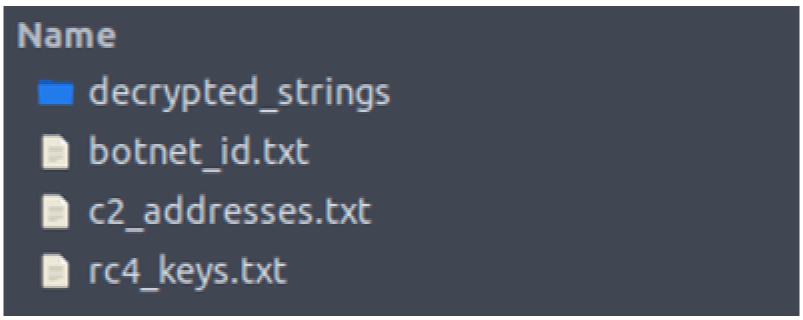


RC4 Keys Found by the Script.

You can then use these keys to decrypt any network communication by using the “-n” option:

Decrypted Network Data.

The script also writes the output data into a folder:



Script Output.

Conclusion

In this post we show the main technical characteristics that makes Dridex a difficult malware to detect and analyze. By publishing this analysis and the automation script, our intention is to help analysts understand how key parts of Dridex work and to help organizations detect and extract Dridex IOCs as early as possible, so that appropriate actions are taken faster.

IOCs

Packed Dridex Loader

d506f18f771ec417c27a6528c17f08ee9d180d40a0a9c6b6ef93b7a39304b96a

Unpacked Dridex Loader

756fa5527f4c564effbc69dd3b3d76e7196b869976eeae48c4b34f4ff25dfa5c

C2 Addresses

45.79.8[.]25:443

185.201.9[.]197:9443

217.160.78[.]166:4664

108.175.9[.]22:33443

Botnet ID

12333

RC4 Keys

xrAuVcgsoW0BBPhAH5w5aQ1Q2UuZQidMhZYugaYvCPvgttsD9jQkM

VTRBArv8sWNVqJ4WDs2rzCN2QMqXLb9fsEjtRZL6vW628p93i3iJe

[1] https://malpedia.caad.fkie.fra...

[2] https://malpedia.caad.fkie.fra...

[3] https://github.com/hasherezade...