A Technical Look At Dyreza | Malwarebytes Labs

By hasherezade

Published: 2015-11-03 · Archived: 2026-04-05 15:43:11 UTC

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

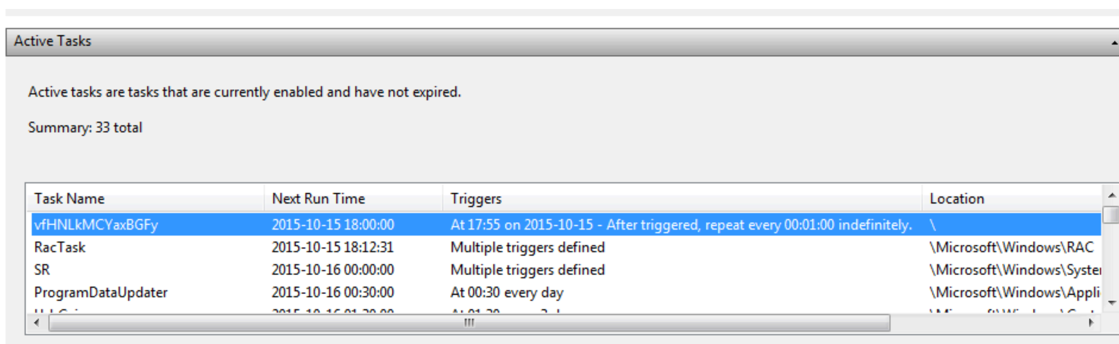
- ff3d706015b7b142ee0a8f0ad7ea2911 – **Dyreza** executable**-** a persistent botnet agent, carring DLLs with the core malicious activities
  - 5a0e393031eb2accc914c1c832993d0b – Dyreza DLL (32bit)
  - 91b62d1380b73baea53a50d02c88a5c6 – Dyreza DLL (64 bit)

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 *explorer*, *svchost*, *taskeng*… 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 *vfHNLkMCYaxBGFy.exe*

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 (*svchost*, *explorer*) communicates with the C&C:

<non-existent>	3160	TCP	testmachine	49160	141.8.226.14	http	CLOSE_WAIT
svchost.exe	600	TCP	testmachine	49183	83.241.176.230	4443	CLOSE_WAIT
svchost.exe	600	TCP	testmachine	49206	83.241.176.230	4443	CLOSE_WAIT

Process	PID	Protocol	Local ...	Remote Address	Remote Port	State	Se...	Sent Bytes	Rcvd Pa...	Rcvd B...
System	4	UDP	netbios-ns	*	*		2 070	103 932	1 929	96 882
svchost.exe	1448	UDP	52678	*	*		16	15 848		
explorer.exe	392	TCP	49679	197.231.198.234	4443	CLOSE_WAIT	3	579	6	1 745
explorer.exe	392	TCP	49680	197.231.198.234	4443	CLOSE_WAIT	3	579	6	1 729

Checking on 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.

```
00294D20 $ 55              PUSH EBP
00294D21 . 8BEC            MOV EBP,ESP
00294D23 . 83EC 2C         SUB ESP,2C
00294D26 . 56              PUSH ESI
00294D27 . 50              PUSH EAX
00294D28 . 64:A1 3000000   MOV EAX,DWORD PTR FS:[30]
00294D2E . 8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
00294D31 . 58              POP EAX
00294D32 . 8B45 FC         MOV EAX,DWORD PTR SS:[EBP-4]
00294D35 . 8378 64 02      CMP DWORD PTR DS:[EAX+64],2           is CPU count < 2?
00294D39 . 0F82 81000000   JB z_payloa.00294DC0                  crash the app!
00294D3F . E8 0CF4FFFF     CALL z_payloa.00294150                load imports
00294D44 . 8BF0            MOV ESI,EAX
```

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.

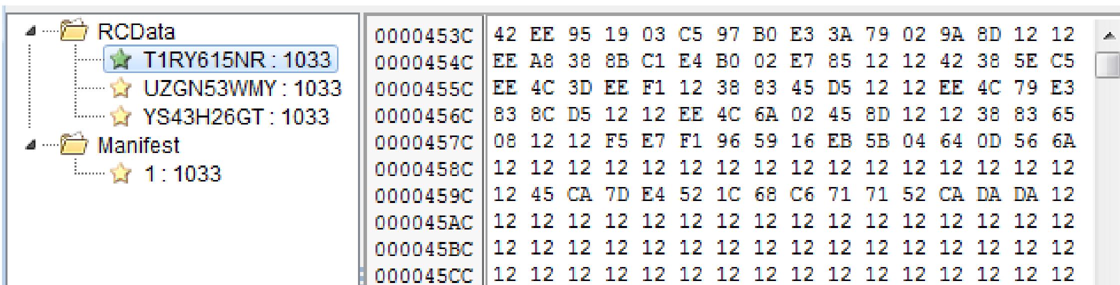
It checks, if it is deployed under debugger – using function *LookupPrivilegeValue* with argument *SeDebugPrivilege* – if it returns non-zero value, execution is terminated.

```
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  .v  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
013D18D6  .   TEST EAX,EAX
013D18D8  .v  JE SHORT vfHNLkMC.013D18F5
013D18DA  .   MOV EAX,[ARG.1]
013D18DD  .   MOV ECX,DWORD PTR DS:[ESI+0x54]        advapi32.AdjustTokenPrivileges
013D18E0  .   PUSH EDI
013D18E1  .   PUSH EDI

DS:[00020020]=7620CDCF (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.

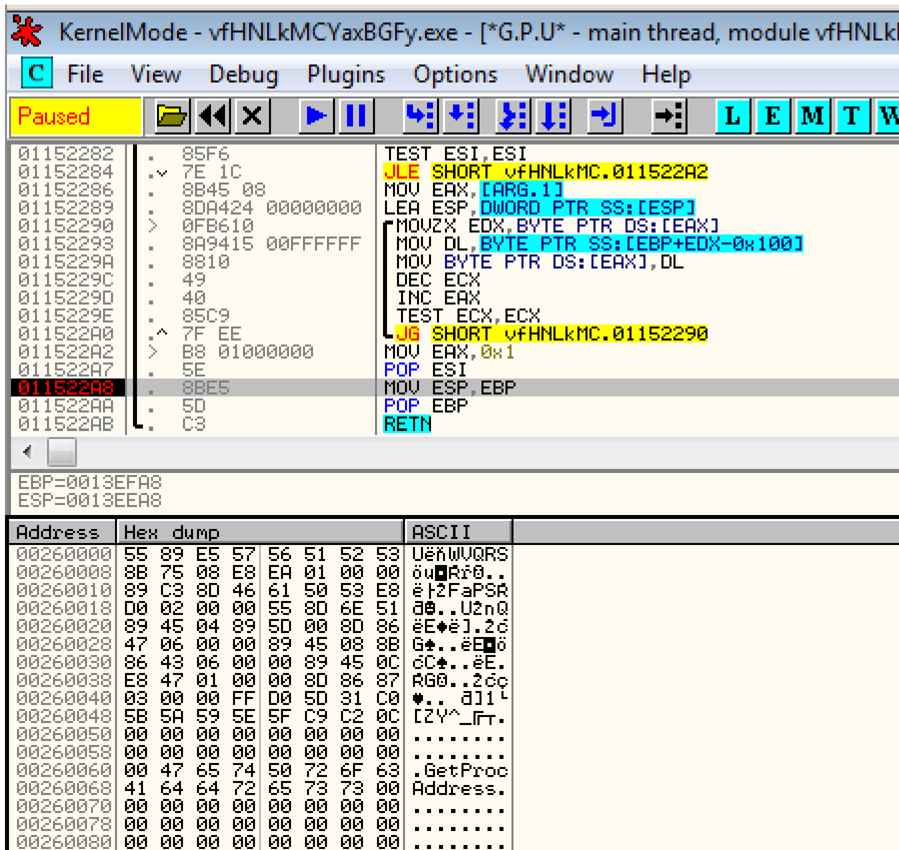
If it is deployed from the valid path and the initial argument passed validation, it performs another check – verifying if it is deployed for the first time. It is achieved by creating a specific Global mutex (it's name is a hash of Computer name and OS Version – fetched by functions: *GetComputerName*, *RtlGetVersion*).

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.



T1RY615NR – encrypted 32 bit code, UZGN53WMY – the key, YS45H26GT – encrypted 64bit code

Unpacking:



The unpacking algorithm is pretty simple – *key_data* contains values and *data* – list of indexes of the values in *key_data*. We process the list of indexes and read the corresponding values:

[code language="python"] def decode(data, key_data): decoded = bytearray() for i in range(0, len(data)): val_index = data[i] decoded.append(key_data[val_index]) return decoded [/code]

This script decrypts dumped resources:

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 *IsWow64Process*.

```
013D3222  .   LEA EDX,[LOCAL.8]
013D3225  .   PUSH EDX                              Arg2 = 770070B4
013D3226  .   PUSH ESI                              Arg1 = 00020000
013D3227  .   CALL vfHNLkMC.013D1830                find resource #1
013D322C  .   ADD ESP,0x18
013D322F  .   TEST EAX,EAX
013D3231  .v  JE SHORT vfHNLkMC.013D327C
013D3233  .   MOV EAX,DWORD PTR DS:[ESI+0x20]       kernel32.GetCurrentProcess
013D3236  .   PUSH vfHNLkMC.013D5040
013D323B  .   CALL EAX
013D323D  .   MOV ECX,DWORD PTR DS:[ESI+0x108]      kernel32.IsWow64Process
013D3243  .   PUSH EAX
013D3244  .   CALL ECX                              vfHNLkMC.013D5040
013D3246  .   TEST EAX,EAX
013D3248  .v  JE SHORT vfHNLkMC.013D327C
013D324A  .   CMP DWORD PTR DS:[0x13D5040],0x0
013D3251  .   MOV EAX,DWORD PTR DS:[ESI+0x174]      vfHNLkMC.013D3C80
013D3257  .   LEA EDX,[LOCAL.8]
013D325A  .   PUSH EDX                              ntdll.KiFastSystemCallRet
013D325B  .v  JE SHORT vfHNLkMC.013D3283
013D325D  .   PUSH 0x7
013D325F  >   CALL EAX
013D3261  .   PUSH vfHNLkMC.013D5044                Arg4 = 013D5044
013D3266  .   PUSH vfHNLkMC.013D5038                Arg3 = 013D5038
013D326B  .   LEA ECX,[LOCAL.8]
013D326E  .   PUSH ECX                              Arg2 = 013D5040
013D326F  .   PUSH ESI                              Arg1 = 00020000
013D3270  .   CALL vfHNLkMC.013D1830                find resource #2
013D3275  .   ADD ESP,0x18
```

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.

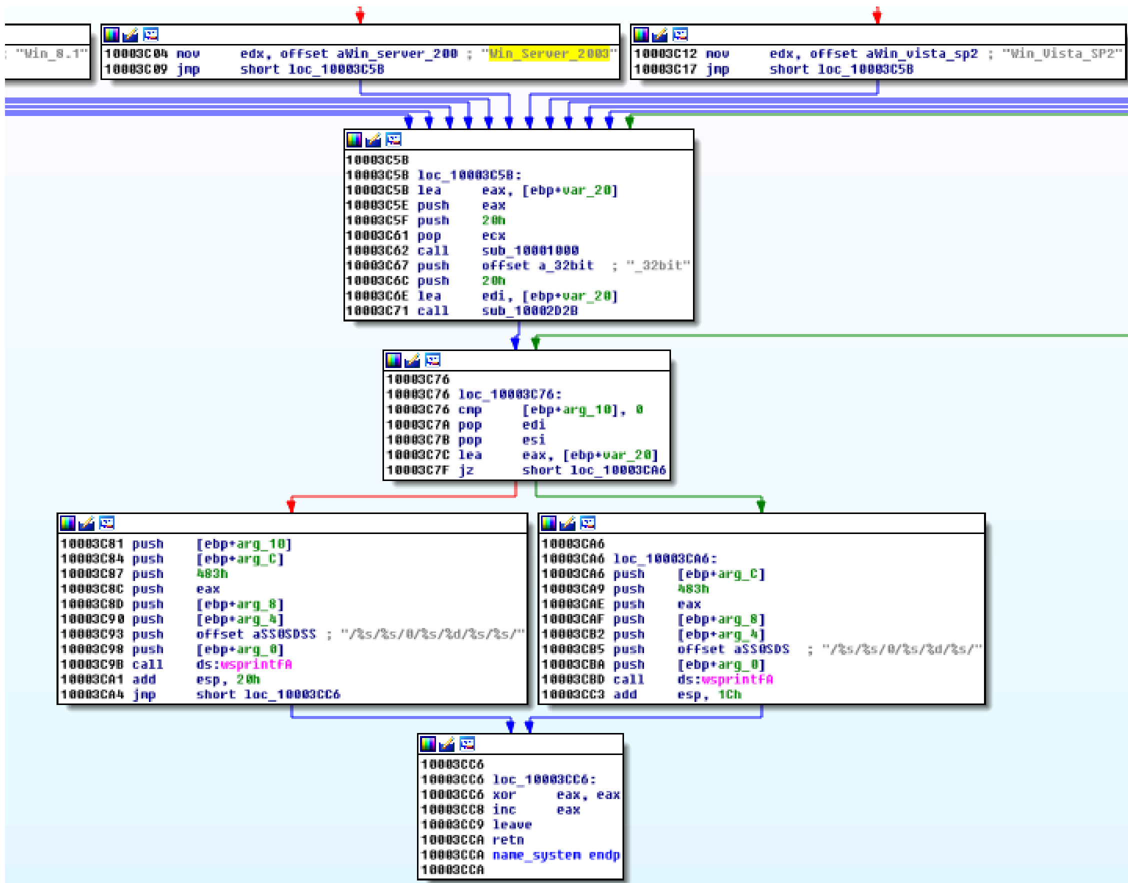
We can see the strings that are used for communication with the C&C:

Both – 32 and 64 bit DLLs have analogical functionality. Only architecture-related elements and strings are different.

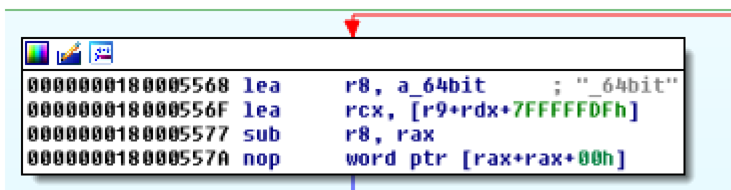
The agent identifies the system:

```
0FD23BA3  .  TEST EAX,EAX                              kernel32.BaseThreadInitThunk
0FD23BA5  .v JE test_5rs.0FD23C76
0FD23BAB  .  MOV EAX,[LOCAL.76]
0FD23BB1  .  CMP EAX,0x1DB0                            Switch (cases A28..2580)
0FD23BB6  .v JNZ SHORT test_5rs.0FD23BC2
0FD23BB8  .  MOV EDX,test_5rs.0FD3138C                 ASCII "Win_7"; Case 1DB0 of switch 0FD23BB1
0FD23BBD  .v JMP test_5rs.0FD23C5B
0FD23BC2  >  CMP EAX,0x1DB1
0FD23BC7  .v JNZ SHORT test_5rs.0FD23BD3
0FD23BC9  .  MOV EDX,test_5rs.0FD31394                 ASCII "Win_7_SP1"; Case 1DB1 of switch 0FD23BB1
0FD23BCE  .v JMP test_5rs.0FD23C5B
0FD23BD3  >  CMP EAX,0xA28
0FD23BD8  .v JNZ SHORT test_5rs.0FD23BE1
0FD23BDA  .  MOV EDX,test_5rs.0FD313A0                 ASCII "Win_XP"; Case A28 of switch 0FD23BB1
0FD23BDF  .v JMP SHORT test_5rs.0FD23C5B
0FD23BE1  >  CMP EAX,0x23F0
0FD23BE6  .v JNZ SHORT test_5rs.0FD23BEF
0FD23BE8  .  MOV EDX,test_5rs.0FD313A8                 ASCII "Win_8"; Case 23F0 of switch 0FD23BB1
0FD23BED  .v JMP SHORT test_5rs.0FD23C5B
0FD23BEF  >  CMP EAX,0x2580
0FD23BF4  .v JNZ SHORT test_5rs.0FD23BFD
0FD23BF6  .  MOV EDX,test_5rs.0FD313B0                 ASCII "Win_8.1"; Case 2580 of switch 0FD23BB1
0FD23BFB  .v JMP SHORT test_5rs.0FD23C5B
0FD23BFD  >  CMP EAX,0xECE
0FD23C02  .v JNZ SHORT test_5rs.0FD23C0B
0FD23C04  .  MOV EDX,test_5rs.0FD313B8                 ASCII "Win_Server_2003"; Case ECE of switch 0FD23BB1
0FD23C09  .v JMP SHORT test_5rs.0FD23C5B
0FD23C0B  >  CMP EAX,0x1772
0FD23C10  .v JNZ SHORT test_5rs.0FD23C19
0FD23C12  .  MOV EDX,test_5rs.0FD313C8                 ASCII "Win_Vista_SP2"; Case 1772 of switch 0FD23BB1
0FD23C17  .v JMP SHORT test_5rs.0FD23C5B
0FD23C19  >  CMP EAX,0x1770
0FD23C1E  .v JNZ SHORT test_5rs.0FD23C27
0FD23C20  .  MOV EDX,test_5rs.0FD313D8                 ASCII "Win_Vista"; Case 1770 of switch 0FD23BB1
0FD23C25  .v JMP SHORT test_5rs.0FD23C5B
0FD23C27  >  CMP EAX,0x1771
0FD23C2C  .v JNZ SHORT test_5rs.0FD23C35
0FD23C2E  .  MOV EDX,test_5rs.0FD313E4                 ASCII "Win_Vista_SP1"; Case 1771 of switch 0FD23BB1
0FD23C33  .v JMP SHORT test_5rs.0FD23C5B
0FD23C35  >  LEA ECX,DWORD PTR DS:[EAX-0x275A]
0FD23C3B  .  CMP ECX,0xA5
0FD23C41  .v JA SHORT test_5rs.0FD23C4A
0FD23C43  .  MOV EDX,test_5rs.0FD313F4                 ASCII "Win_10_IP"
0FD23C48  .v JMP SHORT test_5rs.0FD23C5B
0FD23C4A  >  MOV EDX,test_5rs.0FD31400                 ASCII "Win_10_TH1"
0FD23C4F  .  CMP EAX,0x2800
0FD23C54  .v JE SHORT test_5rs.0FD23C5B
0FD23C56  .  MOV EDX,test_5rs.0FD3140C                 ASCII "unknown"
0FD23C5B  >  LEA EAX,[LOCAL.8]
0FD23C5E  .  PUSH EAX                                 rArg1 = 77203C33
0FD23C5F  .  PUSH 0x20                                |
0FD23C61  .  POP ECX                                  |kernel32.77203C45
0FD23C62  .  CALL test_5rs.0FD21000                   Ltest_5rs.0FF41000
0FD23C67  .  PUSH test_5rs.0FD31414                    ASCII "_32bit"
0FD23C6C  .  PUSH 0x20
0FD23C6E  .  LEA EDI,[LOCAL.8]
0FD23C71  .  CALL test_5rs.0FD22D2B
```

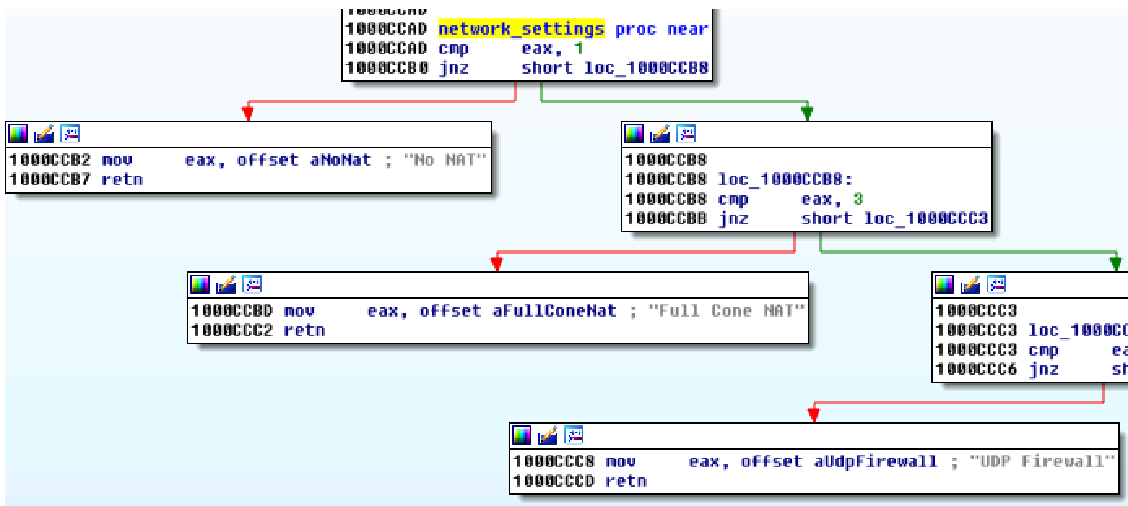
and then – include this data in information sent to the C&C:



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



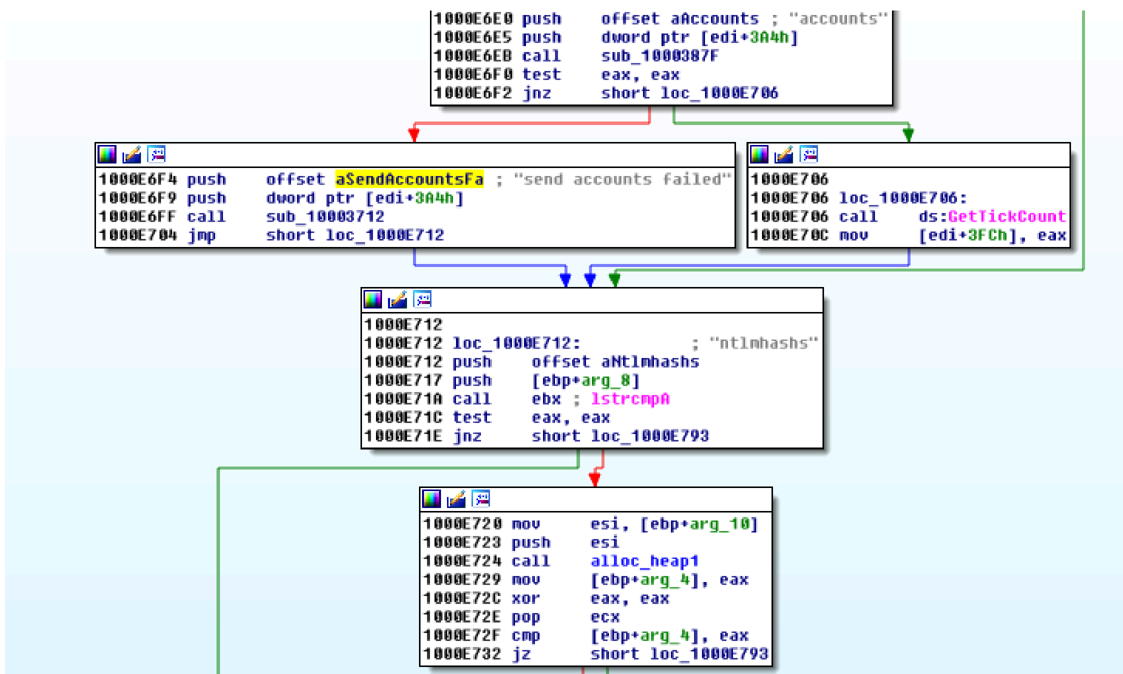
Also, network settings are examined (to verify and inform the C&C whether the client can establish back connection – command : AUTOBACKCONN)



It targets following browsers:

```
0F9C024F  .  8B35 04129C0F   MOV ESI,DWORD PTR DS:[<&SHLWAPI.StrStrIW]  shlwapi.StrStrIW
0F9C0255  .  68 442D9C0F     PUSH test_5rs.0F9C2D44                     Pattern = "chrome.exe"
0F9C025A  .  57              PUSH EDI                                   String = NULL
0F9C025B  .  C745 FC 01000000 MOV [LOCAL.1],0x1
0F9C0262  .  FFD6            CALL ESI                                   StrStrIW
0F9C0264  .  85C0            TEST EAX,EAX
0F9C0266  .v 75 24           JNZ SHORT test_5rs.0F9C028C
0F9C0268  .  68 5C2D9C0F     PUSH test_5rs.0F9C2D5C                     Pattern = "firefox.exe"
0F9C026D  .  57              PUSH EDI                                   String = NULL
0F9C026E  .  FFD6            CALL ESI                                   StrStrIW
0F9C0270  .  85C0            TEST EAX,EAX
0F9C0272  .v 75 18           JNZ SHORT test_5rs.0F9C028C
0F9C0274  .  68 742D9C0F     PUSH test_5rs.0F9C2D74                     Pattern = "iexplore.exe"
0F9C0279  .  57              PUSH EDI                                   String = NULL
0F9C027A  .  FFD6            CALL ESI                                   StrStrIW
0F9C027C  .  85C0            TEST EAX,EAX
0F9C027E  .v 75 0C           JNZ SHORT test_5rs.0F9C028C
0F9C0280  .  68 902D9C0F     PUSH test_5rs.0F9C2D90                     Pattern = "microsoftedge"
0F9C0285  .  57              PUSH EDI                                   String = NULL
0F9C0286  .  FFD6            CALL ESI                                   StrStrIW
```

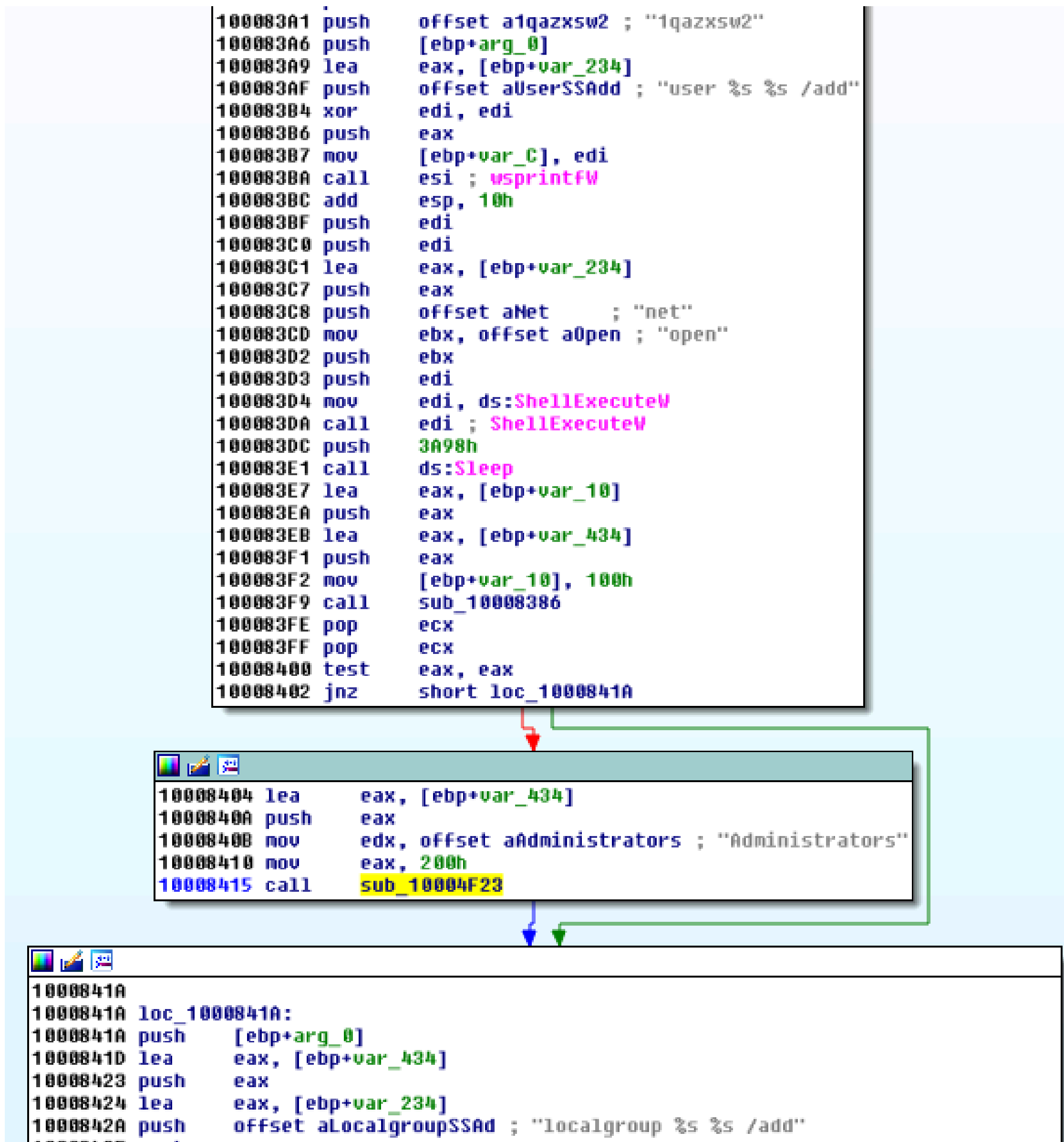
Below – attempt to send stolen account credentials:



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:

```
1000DC4E mov     esi, offset aGeneral ; "--General--\r\n"
1000DC53 push    esi
1000DC54 call    edi ; lstrlenW
1000DC56 push    esi
1000DC57 add     eax, eax
1000DC59 lea     esi, [esp+5Ch+var_3C]
1000DC5D call    sub_1000752A
1000DC62 call    sub_10005B79
1000DC67 mov     ebx, eax
1000DC69 mov     [esp+58h+var_40], ebx
1000DC6D test    ebx, ebx
1000DC6F jz      short loc_1000DC83

1000DC71 push    ebx
1000DC72 call    edi ; lstrlenW
1000DC74 add     eax, eax
1000DC76 push    ebx
1000DC77 call    sub_1000752A
1000DC7C push    ebx
1000DC7D call    heap_free
1000DC82 pop     ecx

1000DC83
1000DC83 loc_1000DC83:                ; "\r\n"
1000DC83 mov     ebx, offset asc_100119EC
1000DC88 push    ebx
1000DC89 push    4
1000DC8B pop     eax
1000DC8C lea     esi, [esp+5Ch+var_3C]
1000DC90 call    sub_1000752A
1000DC95 mov     esi, offset aUsers ; "--Users--\r\n"
1000DC9A push    esi
1000DC9B call    edi ; lstrlenW
1000DC9D push    esi
1000DC9E add     eax, eax
1000DCA0 lea     esi, [esp+5Ch+var_3C]
1000DCA4 call    sub_1000752A
1000DCA9 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:



Trying to add a user with administrative privileges

```
10005856 call     adjust_shutdown_priviledges
1000585B 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
```

Shutdown system on command (AUTOKILLOS)

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:

(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)

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: *AirOS*, *MicroTik*).

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://techhelplist.com/maltlqr/ – by [@Techhelplistcom](https://twitter.com/Techhelplistcom) (list of C&Cs from the current sample: https://techhelplist.com/maltlqr/reports/01oct-20oct-status.txt)
- **Scripts** used in this post: https://github.com/hasherezade/malware_analysis/tree/master/dyreza

About the author

Unpacks malware with as much joy as a kid unpacking candies.

Source: https://blog.malwarebytes.com/threat-analysis/2015/11/a-technical-look-at-dyreza/