

# Analyzing APT19 malware using a step-by-step method

[cybergeeks.tech/analyzing-apt19-malware-using-a-step-by-step-method/](http://cybergeeks.tech/analyzing-apt19-malware-using-a-step-by-step-method/)

## Summary

In this blog post we're presenting a full analysis of a DLL backdoor also reported publicly as Derusbi. This particular piece of malware is associated with the actor known as APT19 (Codoso, C0d0so, Sunshop Group).

APT19, also known as C0d0so or Deep Panda, is allegedly a Chinese-based threat group that targeted a lot of industries in the past. FireEye reported that APT19 was active in 2017 when they used 3 different methods to compromise targets: CVE-2017-0199 vulnerability, macro-enabled Microsoft Excel (XLSM) documents and an application whitelisting bypass to the XLSM documents.

The malware registers itself as a service if it has run with administrator privileges, otherwise, it establishes persistence via the "Run" registry key. The main purpose of the malicious DLL is to gather information about the victim's environment such as username, hostname, IP address of the host, the CPU architecture, the default language for the local system, the amount of physical memory, the amount of physical memory currently available, the processor name, the width and the height of the screen of the primary display monitor. The exfiltrated data is encrypted using a XOR operation (the 1-byte key seems to be randomly-chosen), and then encoded using the Base64 algorithm. There is a lot of network communication performed by the malware, however, due to the fact that the C2 server seems to be sinkholed now, we were not able to retrieve the file that was intended to be downloaded by the process.

## Technical analysis

SHA256:

DE33DFCE8143F9F929ABDA910632F7536FFA809603EC027A4193D5E57880B292

The file analyzed in this blog post is a DLL that has the following export functions:



Name	Address	Ordinal
DebugConnect	6CD7399B	1
DebugCreate	6CD7399B	2
ServiceMain	6CD73311	3
DllEntryPoint	6CD76892	[main entry]

Figure 1

DebugCreate and DebugConnect entries have the same address and represent the starting point of the malicious activity. The process computes a random string of 3 characters using GetTickCount API calls and the algorithm shown in figure 2:

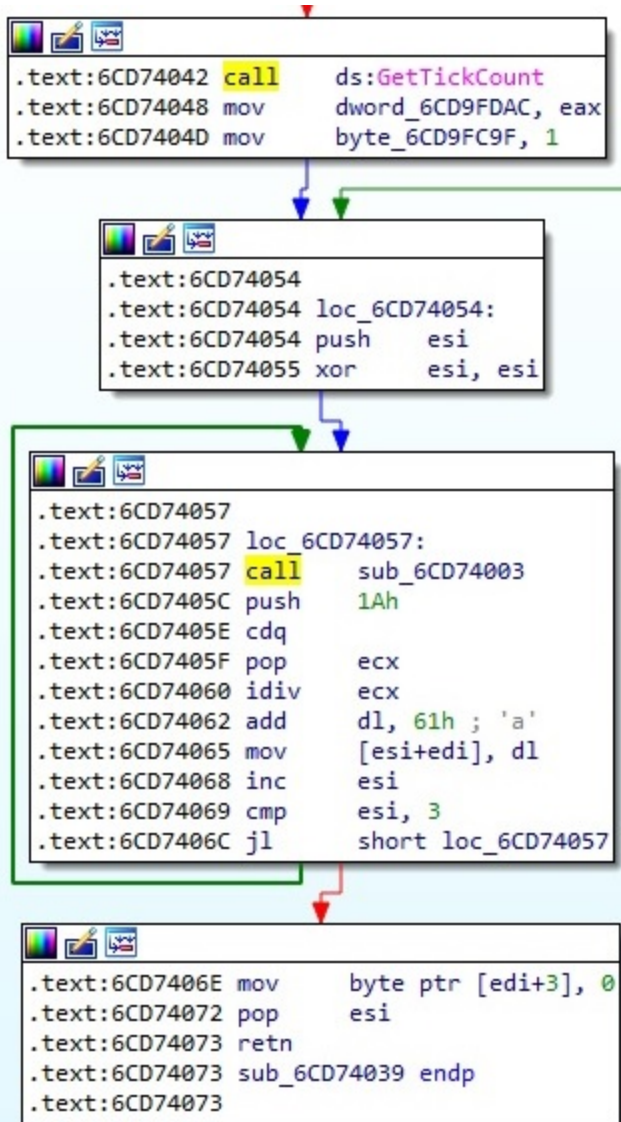


Figure 2

It tries to delete a file/directory called <3 random chars generated earlier>.dll from System32 directory as shown below:

```

.text:6CD73ED1 mov     ecx, eax
.text:6CD73ED3 shr     ecx, 2
.text:6CD73ED6 mov     esi, edx
.text:6CD73ED8 rep movsd
.text:6CD73EDA mov     ecx, eax
.text:6CD73EDC lea     eax, [ebp+pMore]
.text:6CD73EDF push   eax             ; pMore
.text:6CD73EE0 lea     eax, [ebp+pszPath]
.text:6CD73EE6 and     ecx, 3
.text:6CD73EE9 push   eax             ; pszPath
.text:6CD73EEA rep movsb
.text:6CD73EEC call   ds:PathAppendA
.text:6CD73EF2 push   ebx             ; dwErrCode
.text:6CD73EF3 call   ds:SetLastError
.text:6CD73EF9 mov     edi, ds>DeleteFileA
.text:6CD73EFF lea     eax, [ebp+pszPath]
.text:6CD73F05 push   eax             ; lpFileName
.text:6CD73F06 call   edi ; DeleteFileA
.text:6CD73F08 lea     eax, [ebp+pszPath]
.text:6CD73F0E push   eax             ; lpPathName
.text:6CD73F0F call   ds:RemoveDirectoryA
.text:6CD73F15 mov     esi, ds:GetFileAttributesA
.text:6CD73F1B lea     eax, [ebp+pszPath]
.text:6CD73F21 push   eax             ; lpFileName
.text:6CD73F22 call   esi ; GetFileAttributesA
.text:6CD73F24 cmp     eax, 0FFFFFFFh
.text:6CD73F27 jnz     short loc_6CD73F7A

```

Figure 3

Because the file doesn't exist at this time, it's created using CreateFileA API and then deleted using DeleteFileA API. This technique is used to confirm that it has enough rights to write files in the System32 directory:

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code includes instructions for pushing registers, calling CreateFileA, GetLastErrorMessage, and CloseHandle. The memory dump shows the contents of a system32\q1r.dll file, with addresses ranging from 005EF9C8 to 005EF9B8.

Figure 4

The malicious process retrieves process privilege details by calling GetTokenInformation with parameter type 0x14 (TokenElevation):

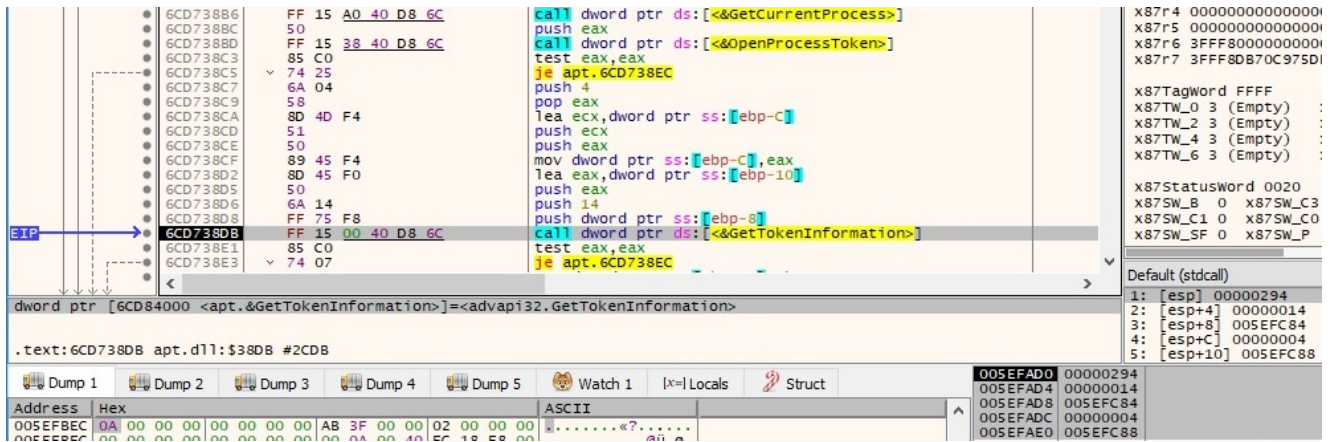


Figure 5

Malware running with admin privileges

Now it queries the “HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows NT\CurrentVersion\Svchost\netsvcs” registry value using RegQueryValueExA function:

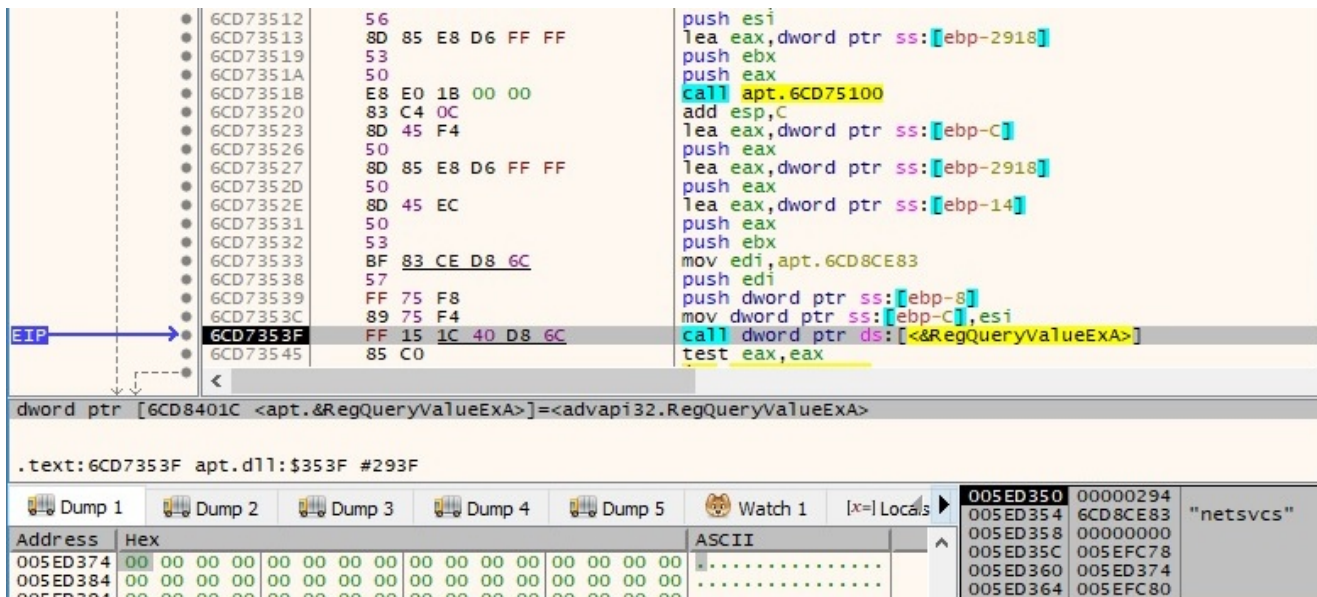


Figure 6

The list of services retrieved earlier is shown in the next figure:

Address	Hex	ASCII
005ED374	43 65 72 74	50 72 6F 70
005ED384	6C 69 63 79	53 76 63 00
005ED394	72 76 65 72	6C 61 6E 6D
005ED3A4	73 76 63 00	61 6E 73 65
005ED384	64 75 6C 65	70 68 6C 70
005ED3C4	73 69 6F 6E	73 63 68 65
005ED3D4	53 77 69 74	00 77 69 6E
005ED3E4	62 69 6C 69	6D 67 6D 74
005ED3F4	00 4E 6C 61	00 53 65 73
005ED404	57 6F 72 68	55 73 65 72
005ED414	70 61 67 65	70 61 74 69
005ED424	61 73 6D 61	61 73 00 49
005ED434	73 73 00 53	72 6D 6F 6E
005ED444	63 65 73 73	00 4E 74 6D
005ED454	61 70 69 73	73 73 76 63
005ED464	6D 53 70 00	00 4E 57 43
005ED474	53 00 53 68	4E 77 73 61
005ED484	6F 6E 00 4C	6E 74 00 52
005ED494	41 75 64 69	61 73 00 49
005ED4A4	6C 6F 61 64	61 73 76 63
005ED484	68 65 72 00	00 4E 6F 6E
005ED4C4	41 70 70 4D	4D 61 6E 61

Figure 7

There is another service called WinHelpSrv that is added to this list. The “netsvcs” value is modified to reflect the change by calling RegSetValueExA API:

The screenshot shows a debugger window with assembly code and a memory dump. The assembly code is as follows:

```

6CD735D5 50          push  eax
6CD735D6 6A 07      push  7
6CD735D8 53          push  ebx
6CD735D9 57          push  edi
6CD735DA FF 75 F8   push  dword ptr ss:[ebp-8]
6CD735DD FF D6      call  esi

```

The memory dump shows the following data:

Address	Hex	ASCII
005ED3A4	73 76 63 00	6D 73 69 73
005ED384	64 75 6C 65	6D 67 6D 74
005ED3C4	73 69 6F 6E	45 6E 76 00
005ED3D4	53 77 69 74	46 61 73 74
005ED3E4	62 69 6C 69	63 68 69 6E
005ED3F4	00 4E 6C 61	67 43 6F 6D
005ED404	57 6F 72 68	70 61 74 69
005ED414	70 61 67 65	61 73 00 49
005ED424	61 73 6D 61	72 6D 6F 6E
005ED434	73 73 00 53	00 4E 57 43
005ED444	63 65 73 73	4E 77 73 61
005ED454	61 70 69 73	6E 74 00 52
005ED464	6D 53 70 00	61 73 00 49
005ED474	53 00 53 68	61 73 76 63
005ED484	6F 6E 00 4C	00 4E 6F 6E
005ED494	41 75 64 69	4D 61 6E 61
005ED4A4	6C 6F 61 64	65 6C 70 73
005ED484	68 65 72 00	65 6E 42 72
005ED4C4	41 70 70 4D	67 65 72 00
005ED4D4	72 76 00 00	6D 00 61 00

The memory dump also shows a list of services with their addresses and names:

Address	Value	Name
005ED350	00000294	
005ED354	6CD8CE83	"netsvcs"
005ED358	00000000	
005ED35C	00000007	
005ED360	005ED374	"CertPropSvc"
005ED364	00000164	
005ED368	00002710	
005ED36C	00000000	
005ED370	008CED80	L"\\C:\\Users\\
005ED374	74726543	windows.stora
005ED378	706F7250	
005ED37C	00637653	
005ED380	6F504353	
005ED384	7963696C	
005ED388	00637653	
005ED38C	6D6E616C	
005ED390	65736E61	
005ED394	72657672	
005ED398	73706700	
005ED39C	69006376	
005ED3A0	706C6870	
005ED3A4	00637673	
005ED3A8	7369736D	

Figure 8

The file creates a new service named WinHelpSrv (Windows Helper Service) as follows:

```

.text:6CD735F6 push 0F003Fh ; dwDesiredAccess
.text:6CD735FB push ebx ; lpDatabaseName
.text:6CD735FC push ebx ; lpMachineName
.text:6CD735FD call ds:OpenSCManagerW
.text:6CD73603 cmp eax, ebx
.text:6CD73605 jz loc_6CD736FA

.text:6CD7360B push ebx ; lpPassword
.text:6CD7360C push offset ServiceStartName ; "z"
.text:6CD73611 push ebx ; lpDependencies
.text:6CD73612 push ebx ; lpdwTagId
.text:6CD73613 push ebx ; lpLoadOrderGroup
.text:6CD73614 push offset BinaryPathName ; lpBinaryPathName
.text:6CD73619 push 1 ; dwErrorControl
.text:6CD7361B push 2 ; dwStartType
.text:6CD7361D push 10h ; dwServiceType
.text:6CD7361F push 0F01FFh ; dwDesiredAccess
.text:6CD73624 push offset DisplayName ; "Windows Helper Service"
.text:6CD73629 push offset ServiceName ; "WinHelpSrv"
.text:6CD7362E push eax ; hSCManager
.text:6CD7362F call ds:CreateServiceA
.text:6CD73635 test eax, eax
.text:6CD73637 jz loc_6CD736FD

```

Figure 9

The description of the service is set to “This is windows helper service. Include windows update and windows error”:

Figure 10

The malicious DLL is registered as a service by adding the “ServiceDll” value that points to its location to the newly created service registry keys:

Figure 11

The confirmation that the operation was successful:

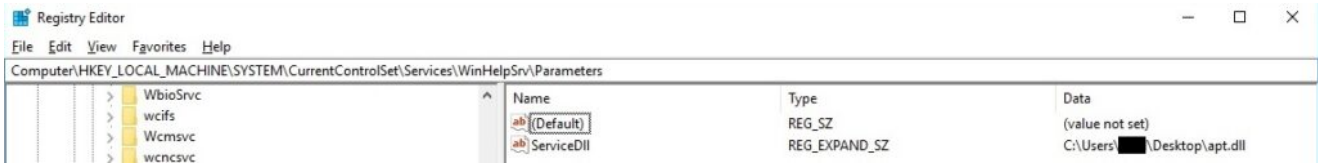


Figure 12

The process creates a batch file called <10 random chars>.bat (the same algorithm utilized before to generate the random letters is used):

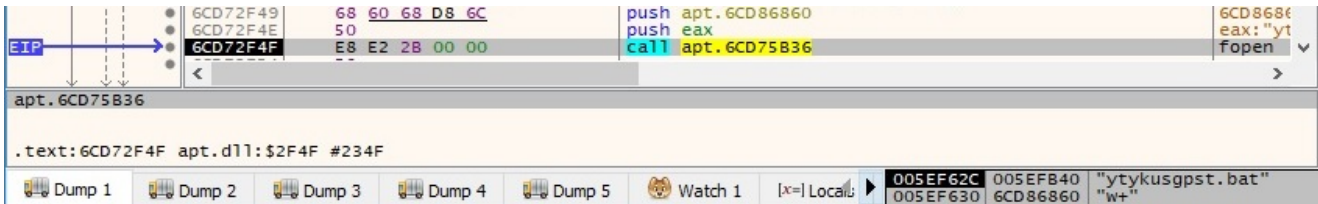


Figure 13

The content of the .bat file is presented below:

```
@echo off
net start %1
del %0
```

The malicious file sets the priority class 0x100 (REALTIME\_PRIORITY\_CLASS) for the current process (this means that the current process has the highest possible priority):

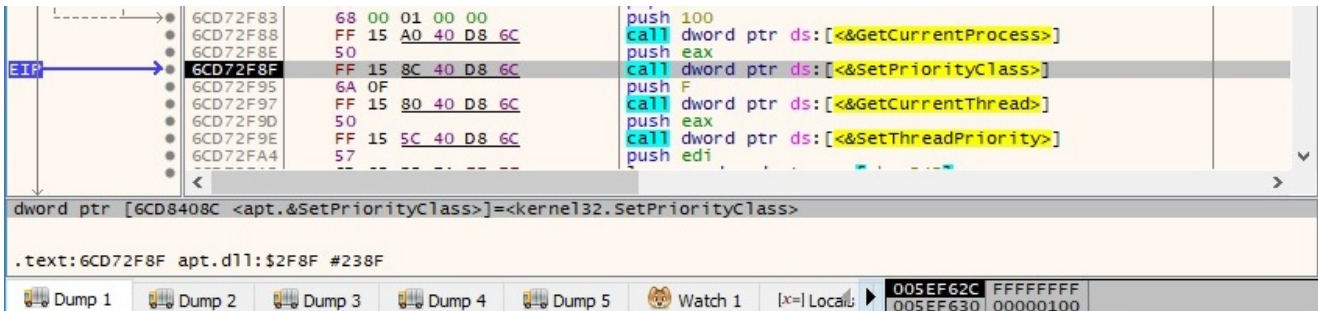


Figure 14

After this operation, there is a call to SetThreadPriority that sets the priority 15 (THREAD\_PRIORITY\_TIME\_CRITICAL) for the current thread:

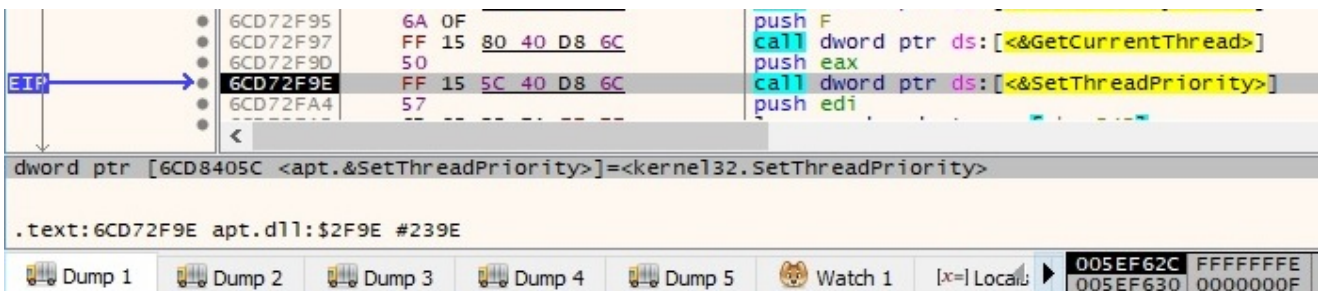


Figure 15

Now there are 2 SHChangeNotify API calls with the following parameters: 0x4 (SHCNE\_DELETE), 0x5 (SHCNF\_PATH), the 3rd parameter is the path to rundll32.exe (because the dll was run using rundll32) and the name of the batch file, respectively, and the 4th parameter is 0. These calls have the purpose of notifying the system if rundll32.exe or the batch file is deleted:

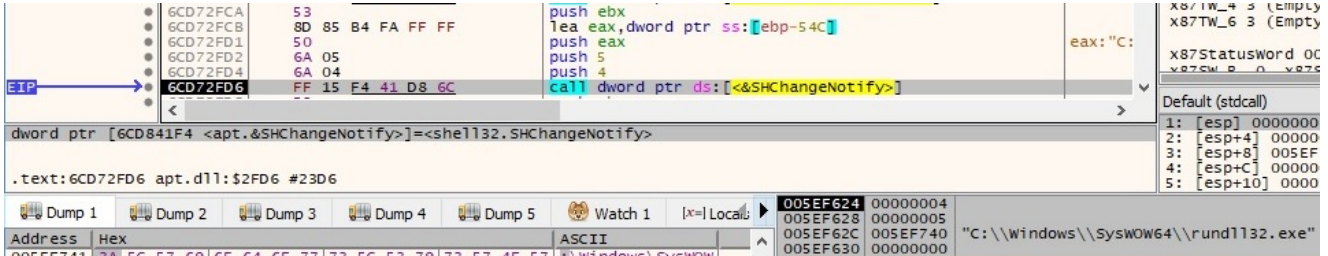


Figure 16

The batch file is executed using the WinExec function. Basically, it starts the WinHelpSrv service, and then the batch file is deleted:

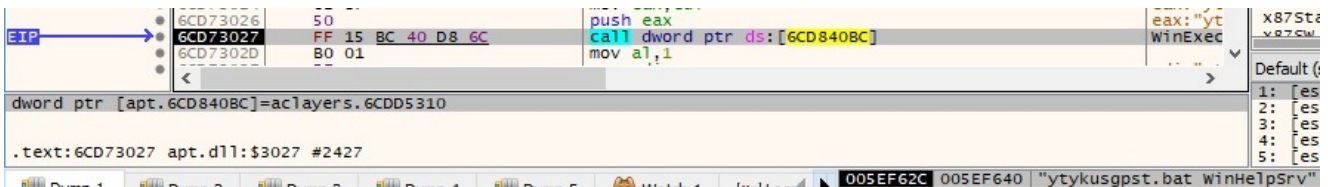


Figure 17

Now we'll talk a bit about the ServiceMain export function that is called when the new service starts. The process registers a function to handle service control requests by calling the RegisterServiceCtrlHandlerA function:

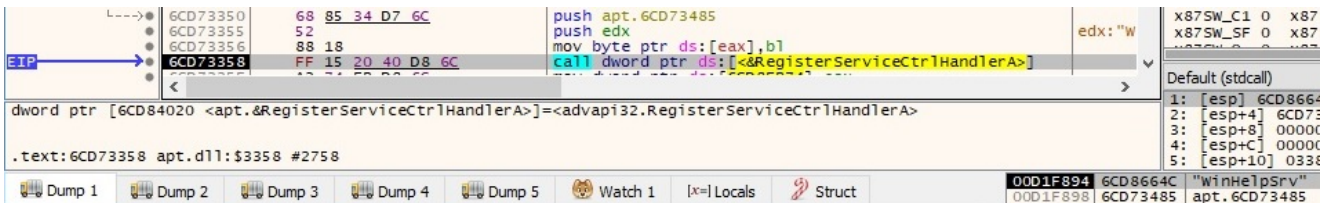


Figure 18

There is a call to SetServiceStatus function using the following SERVICE\_STATUS structure: 0x10 (SERVICE\_WIN32\_OWN\_PROCESS), 0x2 (SERVICE\_START\_PENDING), 0 (no controls are accepted), 0 (dwWin32ExitCode), 0 (dwServiceSpecificExitCode), 0x1 (dwCheckPoint) and 0xbb8 (3000 ms, the amount of time that the service expects an operation to take before the next status update):

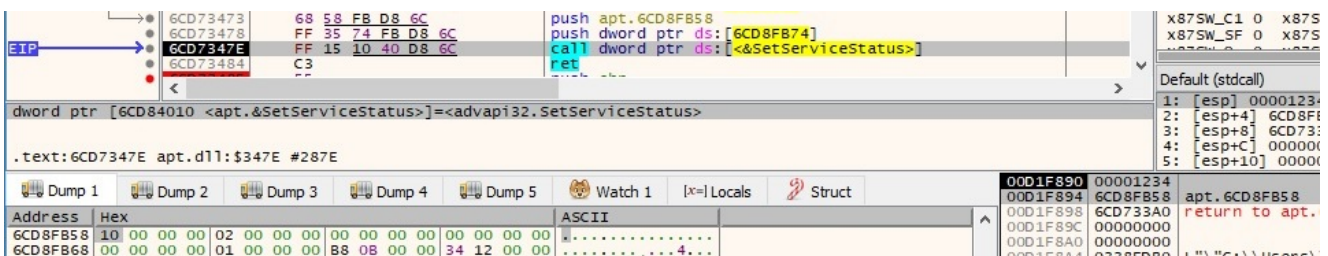


Figure 19



The malicious process creates an unnamed event object by calling the CreateEvent function:

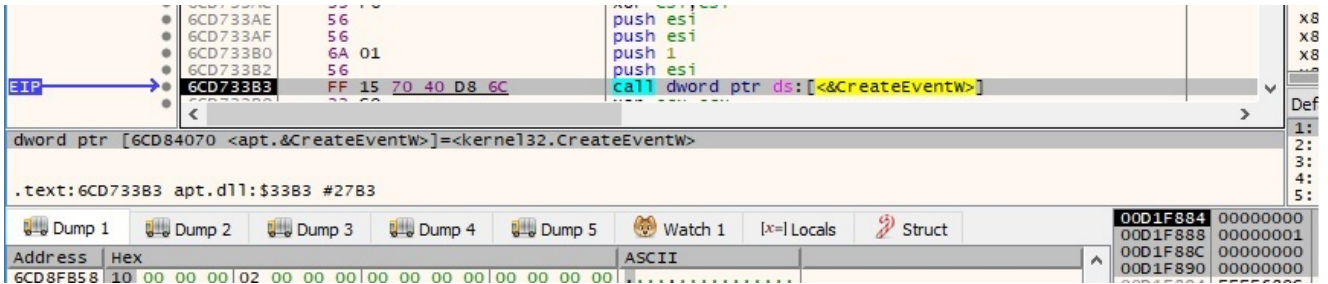


Figure 20

Now it follows another SetServiceStatus call by using the following SERVICE\_STATUS structure: 0x10 (SERVICE\_WIN32\_OWN\_PROCESS), 0x4 (SERVICE\_RUNNING), 0x1 (SERVICE\_ACCEPT\_STOP), 0 (dwWin32ExitCode), 0 (dwServiceSpecificExitCode), 0 (dwCheckPoint) and 0 (dwWaitHint):

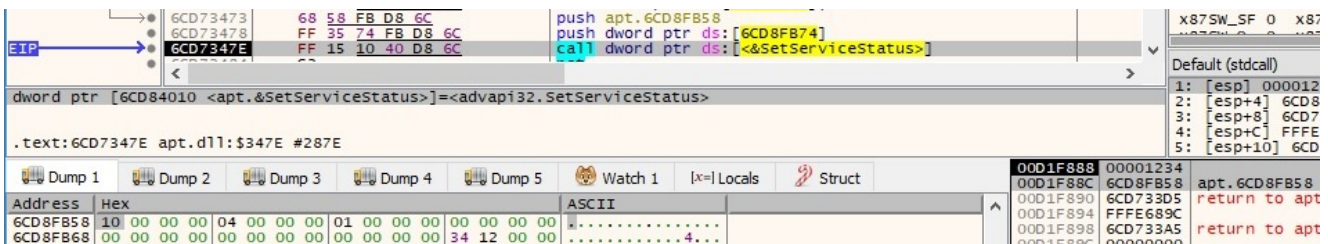


Figure 21

The final operation of this section is to create a new thread using the CreateThread function. The same action will be performed even if the process hasn't run with admin privileges, as we'll see later on:

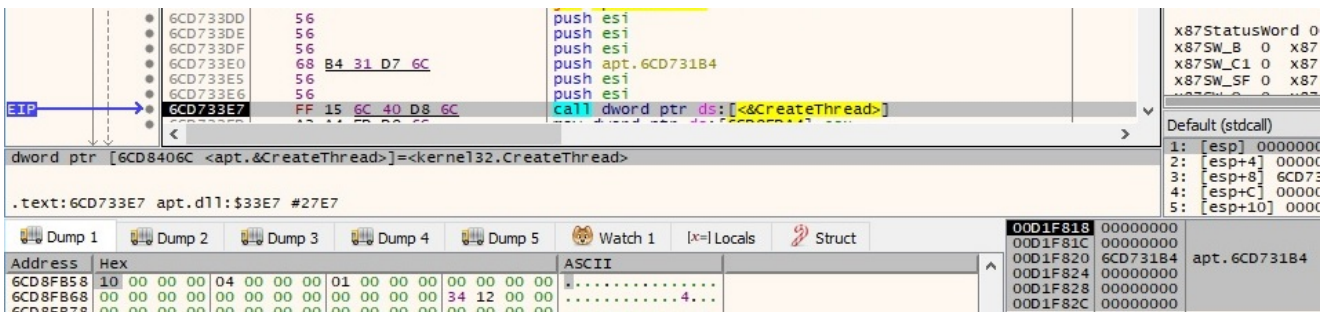


Figure 22

### Malware running without admin privileges

The malware uses an anti-analysis technique by comparing the image path of the executable with rundll32.exe. It is done to ensure that the file is not executed by a sandbox/analyst (it exits if that's the case):

```

.text:6CD7374B
.text:6CD7374B loc_6CD7374B:
.text:6CD7374B xor     ebx, ebx
.text:6CD7374D push   0FEh ; 'p' ; size_t
.text:6CD73752 lea   eax, [ebp+var_107]
.text:6CD73758 push   ebx ; int
.text:6CD73759 push   eax ; void *
.text:6CD7375A mov   [ebp+var_1], bl
.text:6CD7375D mov   [ebp+Filename], bl
.text:6CD73763 call  _memset
.text:6CD73768 add   esp, 0Ch
.text:6CD7376B push   0FFh ; nSize
.text:6CD73770 lea   eax, [ebp+Filename]
.text:6CD73776 push   eax ; lpFilename
.text:6CD73777 push   ebx ; hModule
.text:6CD73778 call  ds:GetModuleFileNameA
.text:6CD7377E lea   eax, [ebp+Filename]
.text:6CD73784 push   offset aRundll32Exe ; "rundll32.exe"
.text:6CD73789 push   eax ; char *
.text:6CD7378A call  _strstr
.text:6CD7378F pop   ecx
.text:6CD73790 pop   ecx
.text:6CD73791 test  eax, eax
.text:6CD73793 jz   short loc_6CD7379C

.text:6CD73795 push   ebx ; uExitCode
.text:6CD73796 call  ds:ExitProcess

.text:6CD7379C
.text:6CD7379C loc_6CD7379C:
.text:6CD7379C lea   eax, [ebp+var_107]

```

Figure 23

The malware is made persistent by adding a new value called WinHelpSrv under the “Run” registry key. In our case, this value points to the location of rundll32.exe because the DLL was run using this executable:

```

005EF98D 43 3A 5C 57 69 6E 64 6F 77 73 5C 53 79 73 57 4F  C:\Windows\SysOW64\rundll32.exe
005EF990 57 36 34 5C 72 75 6E 64 6C 6C 33 32 2E 65 78 65  "C:\Windows\SysOW64\rundll32.exe"
005EF995 73 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

Figure 24

The confirmation that the persistence was successfully established:

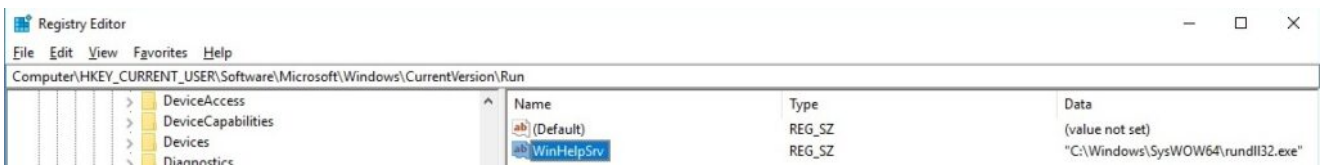


Figure 25

As written before, a new thread is created to execute the same function mentioned when the malware has run with administrator privileges. CreateThread API call is displayed in the next picture:

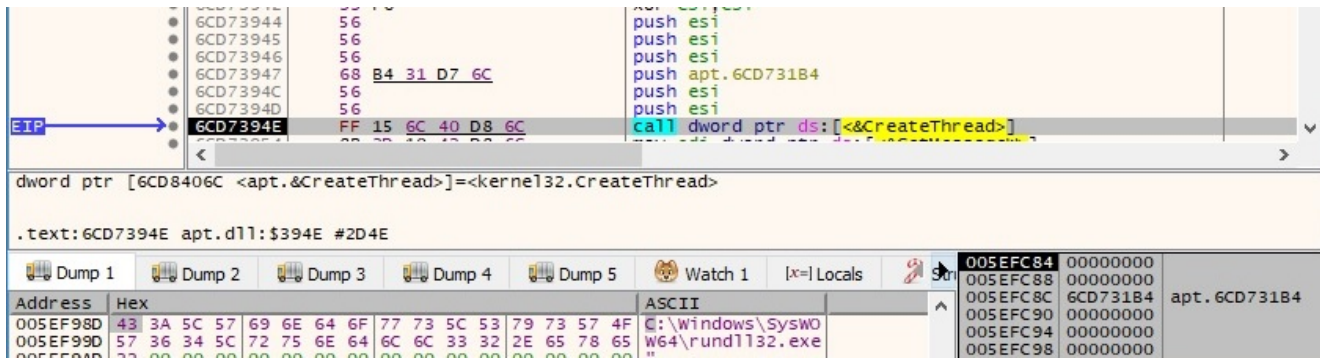
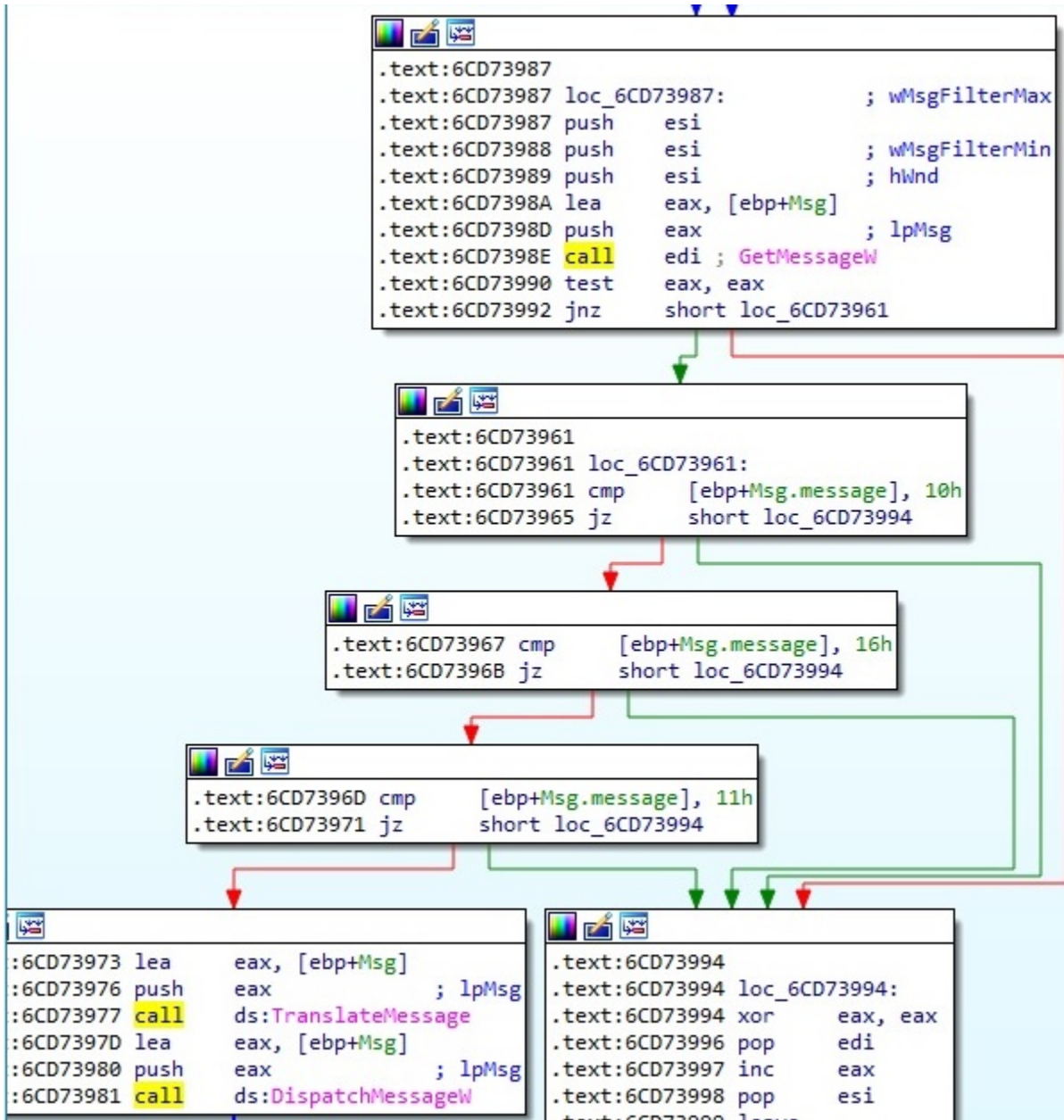


Figure 26

There is a call to GetMessage API to retrieve messages from the thread's message queue. If the message is 0x10 (WM\_CLOSE), 0x11 (WM\_QUERYENDSESSION) or 0x16 (WM\_ENDSESSION) the current function terminates its execution:



Figure

27

Thread activity – StartAddress address

During the entire execution, the internet is emulated using Fakenet. We've observed multiple MultiByteToWideChar function calls used to convert character strings to UTF-16 (wide character) strings. One such call is shown below:

```

    6CD74505 53      push ebx
    6CD74506 50      push eax
    6CD74507 6A FF   push FFFFFFFF
    6CD74509 FF 75 08 push dword ptr ss:[ebp+8]
    6CD7450C 89 45 FC mov dword ptr ss:[ebp-4],eax
    6CD7450F 57      push edi
    6CD74510 57      push edi
    6CD74511 FF D6   call esi
  
```

esi=kernel32.MultiByteToWideChar (76325840)

.text:6CD74511 apt.d11:\$4511 #3911

Address	Hex	ASCII
6CD8D052	68 74 74 70 3A 2F 2F 31 30 36 2E 31 38 35 2E 34	http://106.185.4
6CD8D062	33 2E 39 36 3A 38 30 00 51 65 71 4F 6B 2F 75 73	3.96:80.Qeqok/us

Figure 28

The malware uses the WinHttpOpen function to initialize the use of WinHTTP functions. The user agent is hardcoded in the DLL file:

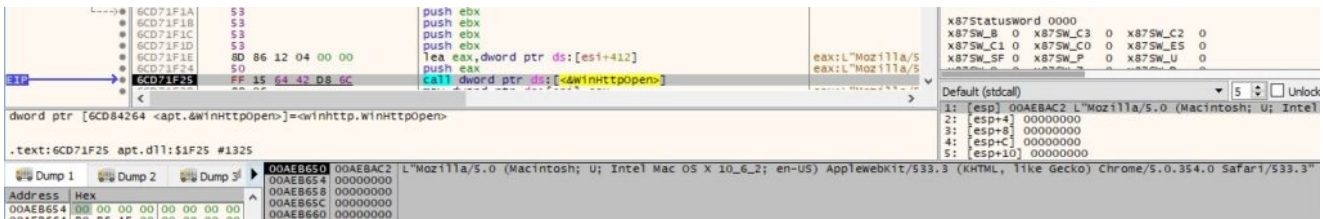


Figure 29

There is a call to WinHttpSetTimeouts function in order to set time-outs involved in HTTP transactions. nResolveTimeout, nConnectTimeout, nSendTimeout and nReceiveTimeout are set to 0x1D4C0 (120.000ms = 120 seconds):

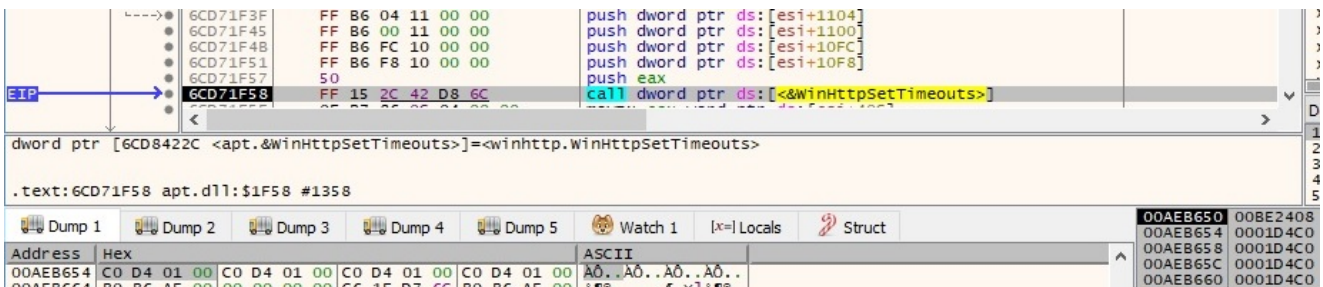


Figure 30

The initial target server of an HTTP request is set to 106.185.43.96 on port 0x50 (80). The WinHttpConnect API call is displayed in figure 31.

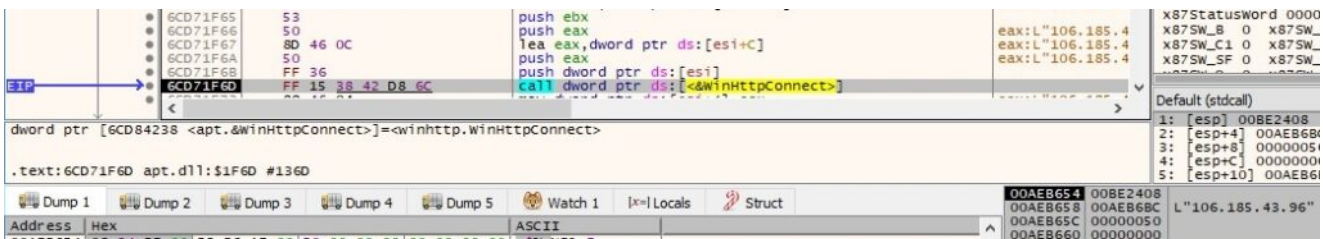


Figure 31

The process performs a GET request to the server mentioned above, with the target resource being /user/atv.html. The pwszReferrer parameter is set to "http://www.google.com" and dwFlags is set to 0x100 (WINHTTP\_FLAG\_BYPASS\_PROXY\_CACHE):



Figure 32

After the WinHttpOpenRequest call there is a WinHttpSendRequest function call. The HTTP request is intercepted by Fakenet, and it replies with a fake response:

```

12/22/20 02:50:22 PM [Diverter] rundll32.exe (5600) requested TCP 106.185.43.96:80
12/22/20 02:50:22 PM [HTTPListener80] GET /user/atv.html HTTP/1.1
12/22/20 02:50:22 PM [HTTPListener80] Cache-Control: no-cache
12/22/20 02:50:22 PM [HTTPListener80] Connection: Keep-Alive
12/22/20 02:50:22 PM [HTTPListener80] Pragma: no-cache
12/22/20 02:50:22 PM [HTTPListener80] Referer: http://www.google.com
12/22/20 02:50:22 PM [HTTPListener80] User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-US) AppleWebKit/533.3 (KHTML, like Gecko) Chrome/5.0.354.0 Safari/533.3
12/22/20 02:50:22 PM [HTTPListener80] Host: 106.185.43.96

```

Figure 33

Now the process is awaiting a response to the HTTP request by calling the WinHttpRequestReceiveResponse function:

Figure 34

Afterward, the malicious file retrieves header information using WinHttpRequestQueryHeaders API with 0x16 (WINHTTP\_QUERY\_RAW\_HEADERS\_CRLF) parameter – receives all the headers returned by the HTTP server:

Address	Hex	ASCII
04600850	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
04600860	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
04600870	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

Figure 35

There is a second WinHttpRequestQueryHeaders API call with 0x20000013 (WINHTTP\_QUERY\_FLAG\_NUMBER|WINHTTP\_QUERY\_STATUS\_CODE) parameter – the status code returned by the HTTP server. It expects a status code of 200 (OK):

Address	Hex	ASCII
04600A50	48 00 54 00 54 00 50 00 2F 00 31 00 2E 00 30 00	H.T.T.P./1..0.
04600A60	20 00 32 00 30 00 30 00 20 00 4F 00 48 00 0D 00	.2.0.0..0.K..
04600A70	0A 00 44 00 61 00 74 00 6E 00 2A 00 70 00 54 00	.A. .4. .a. .r. .n. .* .v. .T.

Figure 36

The process uses the WinHttpRequestQueryDataAvailable function to see how many bytes are available to be read with WinHttpRequestReadData:



Figure 37

Next, there is a call to the WinHttpRequestReadData function that is used to read data returned by the server:

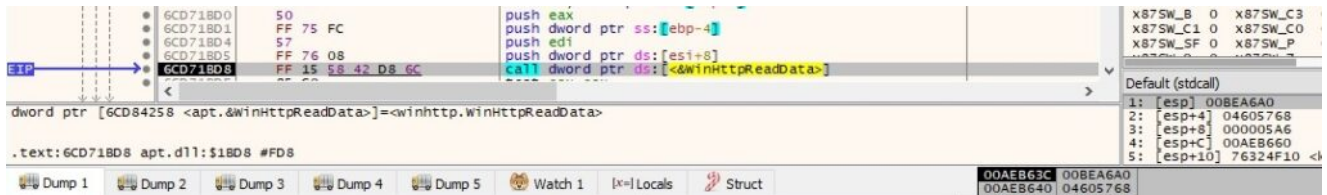


Figure 38

The malicious process uses the WSAStartup function with 0x202 parameter (wVersionRequired) in order to use the Winsock DLL. The current directory for the process is changed to the location of the current executable (rundll32.exe):

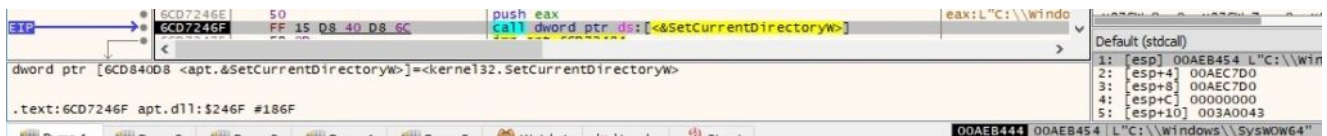


Figure 39

GetAdaptersInfo API is utilized to find adapter information for the local machine. The function call is presented in the next figure.

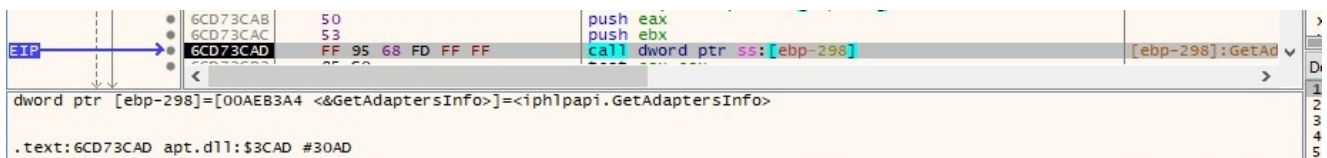


Figure 40

The malware opens the “Software\Microsoft\Windows\CurrentVersion\Internet Settings” registry key by calling the RegCreateKeyExA function:



Figure 41

Now the user agent is extracted from the local host by calling the RegQueryValueExA function, as follows:

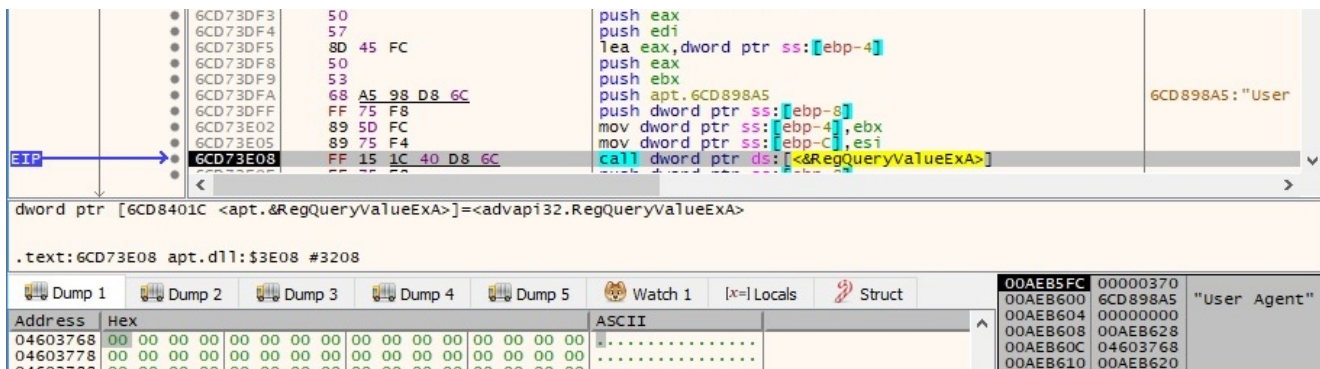


Figure 42

The GetNetworkParams function is utilized to obtain network parameters for the local machine. This information will be exfiltrated as we'll see later on:

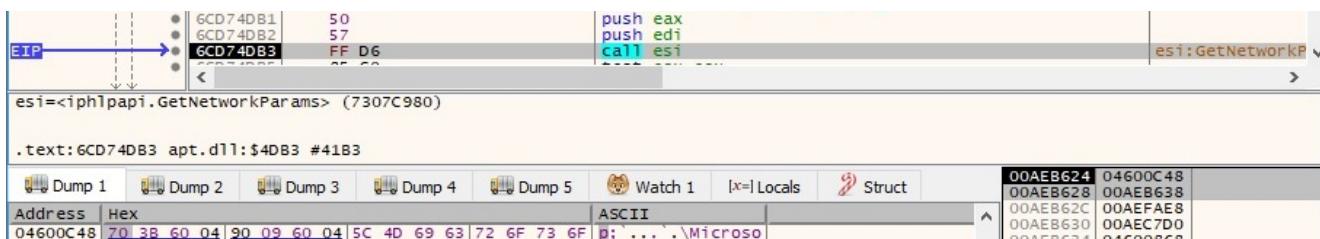


Figure 43

GetComputerNameW and GetUserNameW APIs are used to retrieve the NetBIOS name of the local computer and the name of the user associated with the thread, respectively:

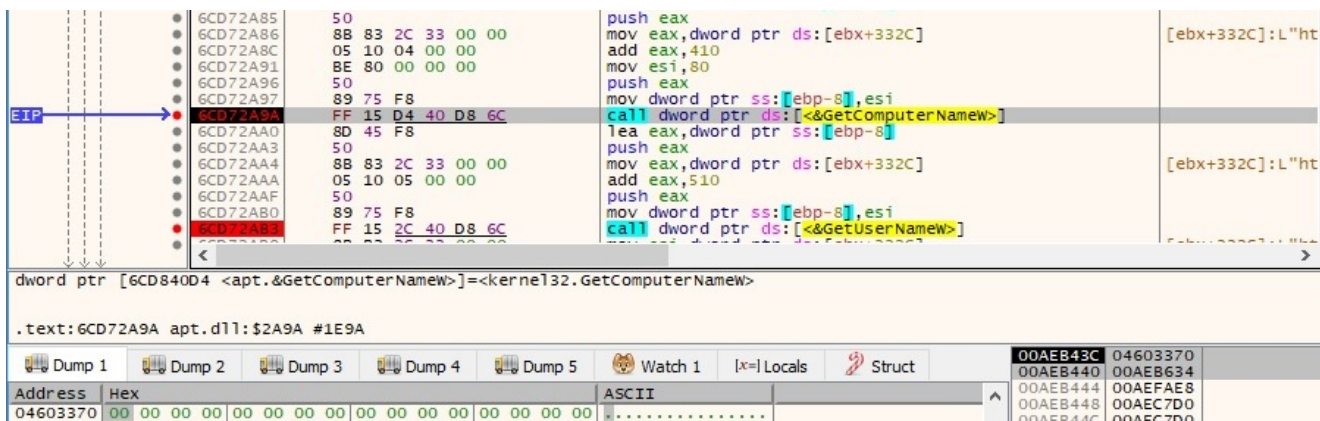


Figure 44

gethostname and gethostbyname functions are used to get the standard host name for the local machine and host information corresponding to the local host, respectively:



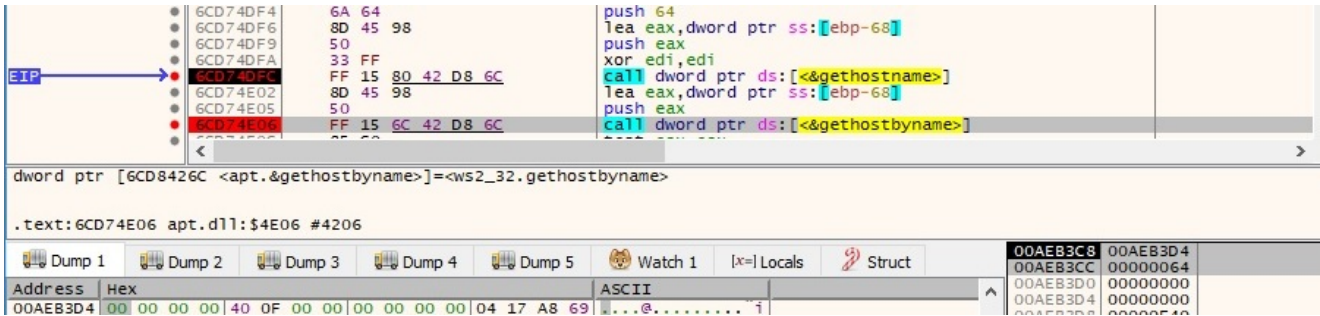


Figure 45

The process verifies the operating system version by calling `GetVersionExA` function and then it checks if the process is running on a 64-bit machine by calling `GetCurrentProcess` and `IsWow64Process` APIs (this information is stored in the buffer along with the hostname and username). The malware retrieves the default locale for the OS by calling `GetLocaleInfoA` function with the following parameters: `0x800` (`LOCALE_SYSTEM_DEFAULT`), `0xb` (`LOCALE_IDEFAULTCODEPAGE`). The result is OEMCP 437 for English ( United States ) that is converted to hex and copied in the buffer that will be exfiltrated:

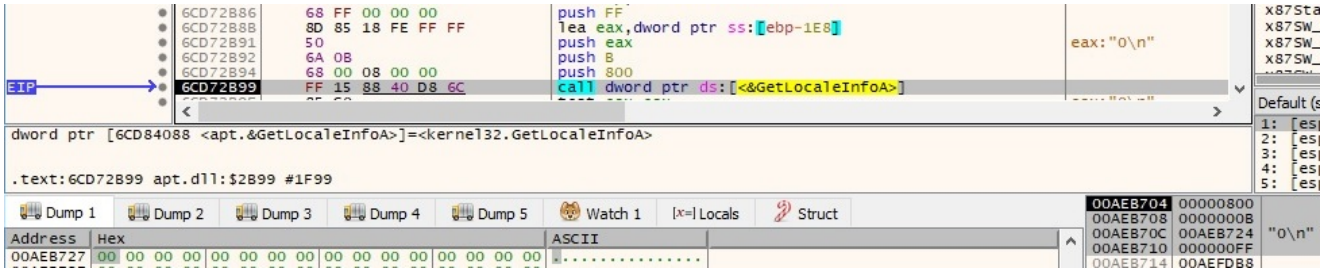


Figure 46

There is a call to the `GlobalMemoryStatusEx` function in order to retrieve information about the physical and virtual memory. The amount of physical memory and the amount of physical memory currently available are saved as 32-bits values to the buffer which will be exfiltrated. Also, the processor name is retrieved using a few `cuid` instructions (“AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx”) and then copied to the same buffer. The malicious process extracts the width and the height of the screen of the primary monitor (in pixels) via 2 `GetSystemMetrics` calls, as follows (these are copied to the same buffer as before):



Figure 47

Again 12 random chars are generated via the same algorithm as presented before, and then the following URI is constructed (data=12 random chars): “/money/ofcom-fines-nuisance-calls?0023528461146965&data=qgvuclxxlgip”. The function WinHttpOpen is called using the user agent extracted earlier from registry, “Mozilla/4.0 (compatible; MSIE 8.0; Win32)”:



Figure 48

As before, the file calls the WinHttpSetTimeouts function using the parameters set as 120 seconds, and then it tries to connect to the C2 server (www.microsoft-cache[.]com) on port 443:



Figure 49

The process performs a GET request using WinHttpOpenRequest and WinHttpSendRequest APIs:

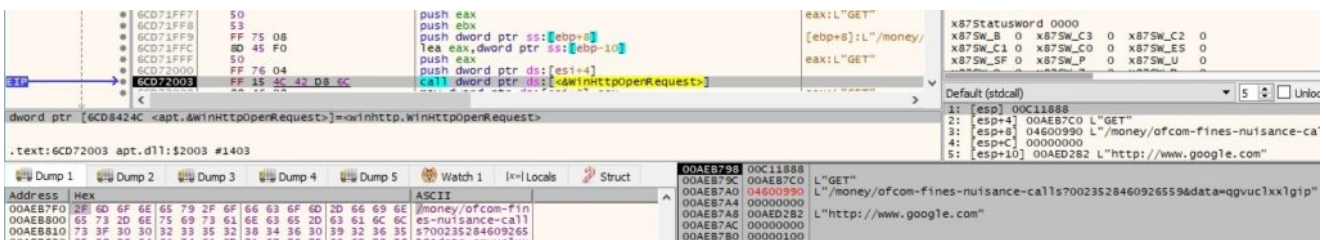


Figure 50

If the request is not successful, the process sleeps for 180 seconds, and then it tries again. The process retrieves header information by calling WinHttpQueryHeaders with 0x16 (WINHTTP\_QUERY\_RAW\_HEADERS\_CRLF) parameter:

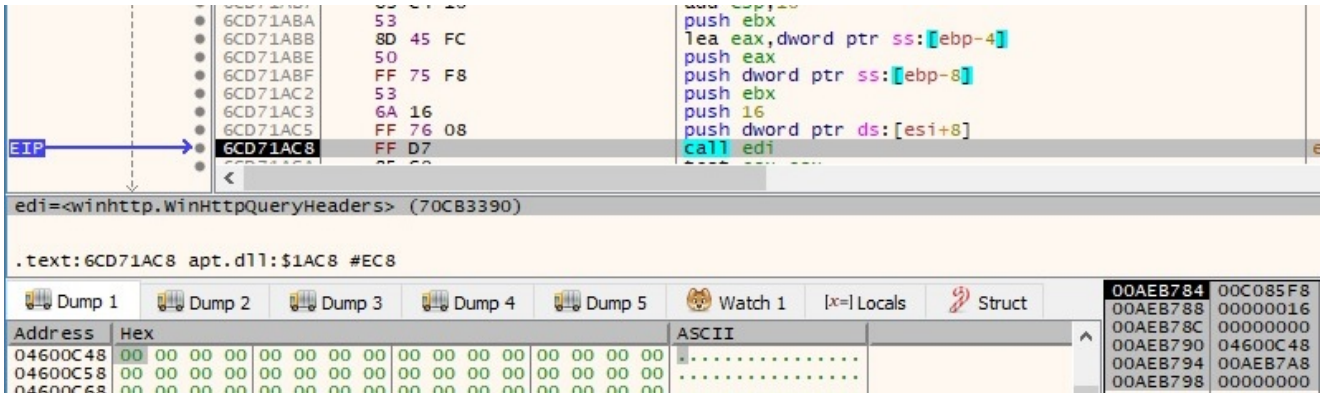


Figure 51

As before, the malware extracts the status code and checks if it's equal to 200 by calling WinHttpQueryHeaders API with 0x20000013 (WINHTTP\_QUERY\_FLAG\_NUMBER|WINHTTP\_QUERY\_STATUS\_CODE) parameter:

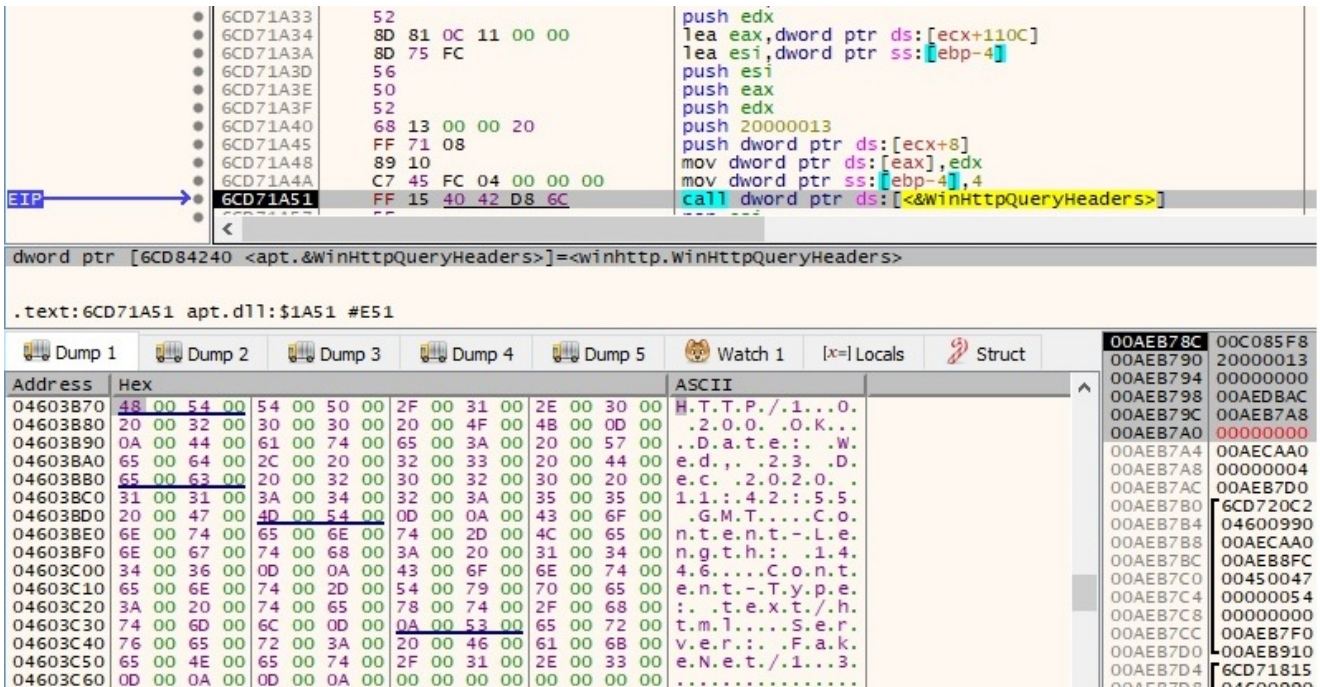


Figure 52

Now there is a call to the WinHttpQueryDataAvailable function, and then it reads the data returned by the C2 server using WinHttpReadData API:

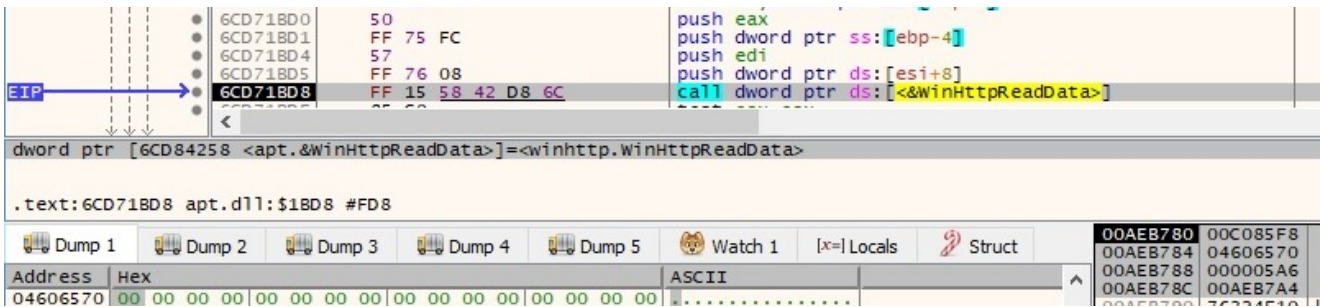


Figure 53



The malware developers have written their implementation of the Base64 algorithm rather than relying on Windows APIs. The following picture presents a part of the assembly code corresponding to it:

```
.text:6CD74268
.text:6CD74268 loc_6CD74268:
.text:6CD74268 mov     [ecx], eax
.text:6CD7426A mov     eax, edi
.text:6CD7426C cdq
.text:6CD7426D push   3
.text:6CD7426F pop    ecx
.text:6CD74270 idiv   ecx
.text:6CD74272 mov     ecx, edi
.text:6CD74274 xor     eax, eax
.text:6CD74276 xor     esi, esi
.text:6CD74278 sub     ecx, edx
.text:6CD7427A mov     [ebp+var_4C], ecx
.text:6CD7427D test    ecx, ecx
.text:6CD7427F jle     short loc_6CD742ED
```

```
.text:6CD74281
.text:6CD74281 loc_6CD74281:
.text:6CD74281 mov     edi, [ebp+arg_0]
.text:6CD74284 lea    ecx, [edi+esi]
.text:6CD74287 movzx  edx, byte ptr [ecx]
.text:6CD7428A shr     edx, 2
.text:6CD7428D mov     dl, [ebp+edx+var_48]
.text:6CD74291 mov     [ebx+eax], dl
.text:6CD74294 movzx  ecx, byte ptr [ecx]
.text:6CD74297 shl     ecx, 4
.text:6CD7429A lea    edi, [edi+esi+1]
.text:6CD7429E movzx  edx, byte ptr [edi]
.text:6CD742A1 shr     edx, 4
.text:6CD742A4 add     edx, ecx
.text:6CD742A6 and     edx, 3Fh
.text:6CD742A9 mov     cl, [ebp+edx+var_48]
.text:6CD742AD mov     [ebx+eax+1], cl
.text:6CD742B1 mov     ecx, [ebp+arg_0]
.text:6CD742B4 movzx  edi, byte ptr [edi]
.text:6CD742B7 lea    ecx, [ecx+esi+2]
.text:6CD742BB movzx  edx, byte ptr [ecx]
.text:6CD742BE shr     edx, 6
.text:6CD742C1 lea    edx, [edx+edi*4]
.text:6CD742C4 and     edx, 3Fh
.text:6CD742C7 mov     dl, [ebp+edx+var_48]
.text:6CD742CB mov     [ebx+eax+2], dl
.text:6CD742CF movzx  ecx, byte ptr [ecx]
.text:6CD742D2 and     ecx, 3Fh
.text:6CD742D5 mov     cl, [ebp+ecx+var_48]
.text:6CD742D9 mov     [ebx+eax+3], cl
.text:6CD742DD mov     ecx, [ebp+var_4C]
.text:6CD742E0 add     esi, 3
.text:6CD742E3 add     eax, 4
.text:6CD742E6 cmp     esi, ecx
.text:6CD742E8 jl      short loc_6CD74281
```

Figure 56

The encrypted buffer is encoded with the Base64 algorithm:

Address	Hex	ASCII
04603D68	45 67 67 41	EggAANcCbKQABAAA
04603D78	41 41 41 41	AAAACAAav9ej16PX
04603D88	70 39 66 74	p9ft1/jX+Neg16DX
04603D98	6F 4E 66 35	oNf517rXvte016XX
04603DA8	75 4E 65 68	uNek17jXsdej1/rX
04603DB8	74 4E 65 32	tNe217Txv9ey1/nX
04603DC8	74 4E 65 34	tNe417rX7dfj1+PX
04603DD8	35 4E 66 58	5NfX19fX19fX19fX
04603DE8	31 39 66 58	19fX19fX19fX19fX
04603DF8	31 39 66 58	19fX19fX19fX19fX
04603E08	31 39 66 58	19fX19fX19fX19fX
04603E18	31 39 66 58	19fX19fX19fX19fX
04603E28	31 39 66 58	19fX19fX19fX19fX
04603E38	31 39 66 58	19fX19fX19fX19fX
04603E48	31 39 66 58	19fX19fX19fX19fX
04603E58	31 39 66 58	19fX19fX19fX19fX
04603E68	31 39 66 58	19fX19fX19fX19fX
04603E78	31 39 66 58	19fX19fX19fX19fX
04603E88	31 39 66 58	19fX19fX19fX19fX

Figure 57

As before, there is a WinHttpOpen API call (same user agent as the last time) followed by a WinHttpSetTimeouts function call, and then it tries to connect to www.microsoft-cache[.]com on port 443 using WinHttpConnect API. The malware performs a POST request by calling the WinHttpRequest function (as before, the data parameter contains randomly-generated characters):

The screenshot shows assembly code with the instruction `call dword ptr ds:[<winHttpRequest>]` highlighted. The register window shows `eax:L"POST"` and `ecx:[ebp+8]:L"/world/`. The dump window shows a hex dump of the data parameter.

Figure 58

The encrypted + encoded buffer is exfiltrated to the C2 server via a WinHttpWriteData function call, as shown below:

The screenshot shows assembly code with the instruction `call dword ptr ds:[<winHttpWriteData>]` highlighted. The register window shows `eax` and `ecx:[ebp+10],ebx`. The dump window shows a hex dump of the data parameter.

Figure 59

The malicious process performs 2 WinHttpQueryHeaders function calls: 1st one has 0x16 (WINHTTP\_QUERY\_RAW\_HEADERS\_CRLF) parameter and the 2nd one has 0x20000013 (WINHTTP\_QUERY\_FLAG\_NUMBER|WINHTTP\_QUERY\_STATUS\_CODE) parameter. It checks out the status code and ensures that it's 200. The thread continues by calling WinHttpQueryDataAvailable and WinHttpReadData APIs to retrieve the server's response. The malware performs another GET request to the C2 server:



Figure 60

The same steps as before are repeated one more time: 2 WinHttpQueryHeaders calls followed by WinHttpQueryDataAvailable and then WinHttpReadData in order to read the data sent by the server. As mentioned in the Unit42 article at <https://unit42.paloaltonetworks.com/new-attacks-linked-to-c0d0s0-group/>, the server's response should contain a "background-color" parameter followed by "#" and an offset. The offset is read, converted to an integer using the atoi function, and then divided by 100, as shown in figure 61:

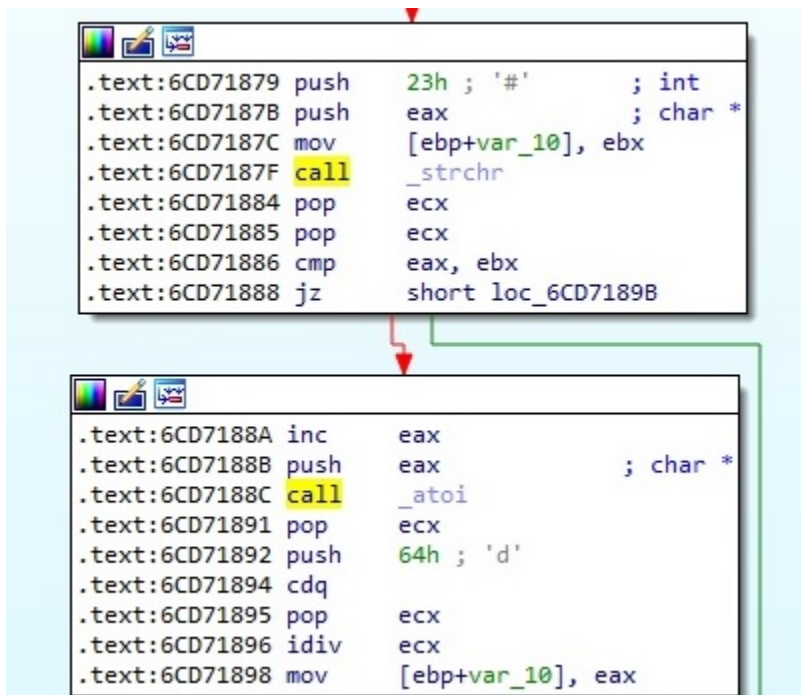
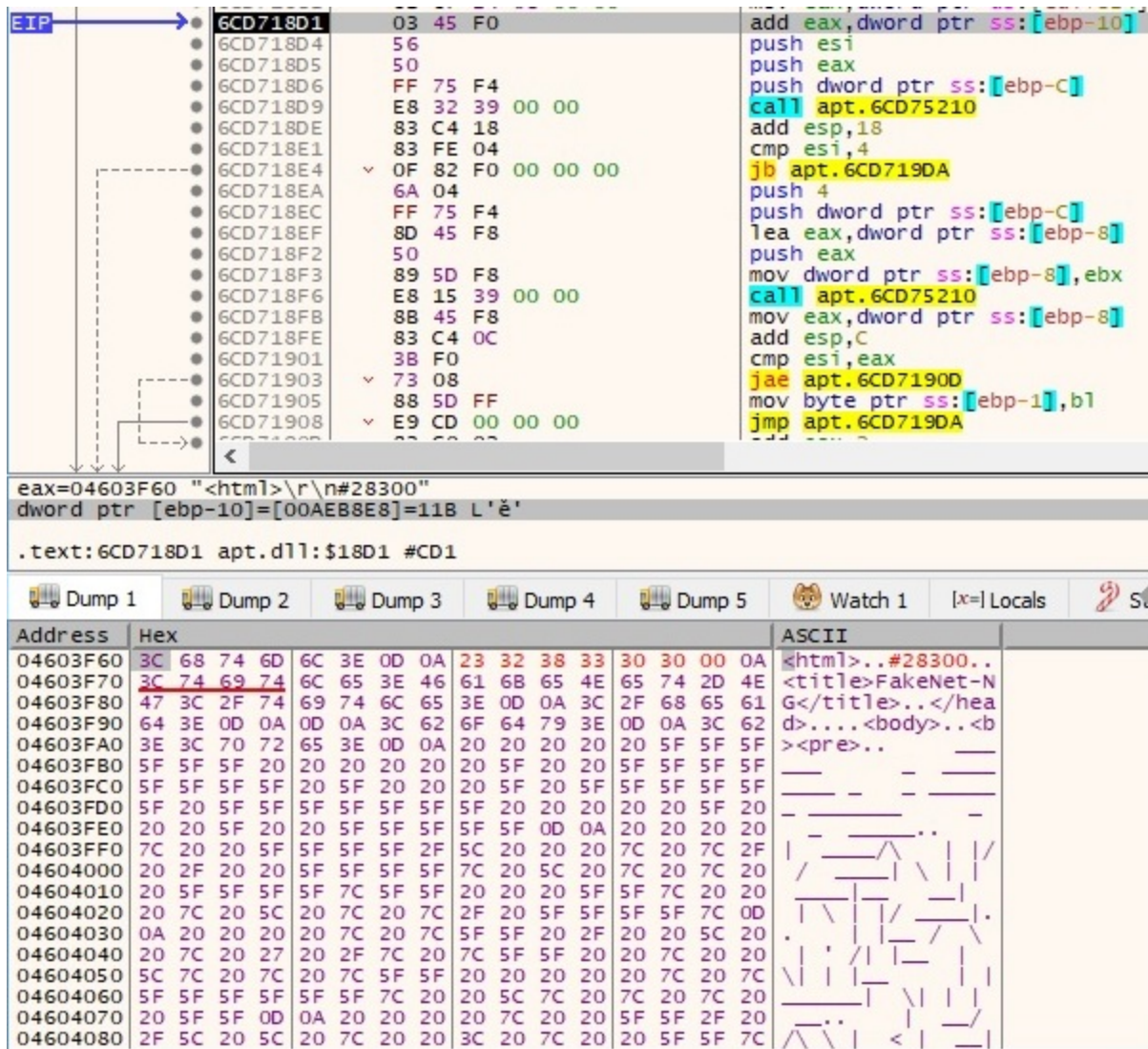


Figure 61

The idea is that the malware reads the data found at the position equal to offset/100. In our case, we've modified the response to contain "#28300" which translates to an offset of 28300 (the position will be 28300/100 = 283). The following picture reveals the fact that the process reads the data found at that specific position (0x11b = 283):



Figure

62

According to the same article, the first 4 bytes represent the total length, and the remaining data would be Base64-encoded. Indeed we were able to identify the function where the server's response is Base64-decoded:



```

.text:6CD74375 var_104= byte ptr -104h
.text:6CD74375 var_103= byte ptr -103h
.text:6CD74375 var_4= dword ptr -4
.text:6CD74375 arg_0= dword ptr 8
.text:6CD74375 arg_4= dword ptr 0Ch
.text:6CD74375 arg_8= dword ptr 10h
.text:6CD74375
.text:6CD74375 push    ebp
.text:6CD74376 mov     ebp, esp
.text:6CD74378 sub     esp, 150h
.text:6CD7437E mov     eax, ___security_cookie
.text:6CD74383 xor     eax, ebp
.text:6CD74385 mov     [ebp+var_4], eax
.text:6CD74388 mov     eax, [ebp+arg_4]
.text:6CD7438B push   ebx
.text:6CD7438C push   esi
.text:6CD7438D push   edi
.text:6CD7438E push   10h
.text:6CD74390 mov     ebx, ecx
.text:6CD74392 pop     ecx
.text:6CD74393 mov     esi, offset aAbcdefghijklmn ; "ABCDEFGHIIJKLMNOPQRSTUVWXYZabcdefghijklmnop..."
.text:6CD74398 lea    edi, [ebp+var_148]
.text:6CD7439E rep    movsd
.text:6CD743A0 mov     [ebp+var_150], eax
.text:6CD743A6 mov     eax, [ebp+arg_8]
.text:6CD743A9 push   0FFh ; size_t
.text:6CD743AE mov     [ebp+var_14C], eax
.text:6CD743B4 lea    eax, [ebp+var_103]
.text:6CD743BA push   0 ; int
.text:6CD743BC push   eax ; void *
.text:6CD743BD movsb
.text:6CD743BE mov     [ebp+var_104], 0
.text:6CD743C5 call   _memset
.text:6CD743CA add     esp, 0Ch
.text:6CD743CD mov     esi, ebx
.text:6CD743CF test   ebx, ebx
.text:6CD743D1 jle    short loc_6CD743E2

```

Figure 63

```
.text:6CD74445 mov     eax, [ebp+arg_0]
.text:6CD74448 lea     esi, [ebx-1]
.text:6CD7444B shr     esi, 2
.text:6CD7444E inc     eax
.text:6CD7444F inc     esi

.text:6CD74450
.text:6CD74450 loc_6CD74450:
.text:6CD74450 movzx  ecx, byte ptr [eax-1]
.text:6CD74454 mov     cl, [ebp+ecx+var_104]
.text:6CD74458 movzx  ebx, byte ptr [eax]
.text:6CD7445E mov     bl, [ebp+ebx+var_104]
.text:6CD74465 shl     cl, 2
.text:6CD74468 shr     bl, 4
.text:6CD7446B or      cl, bl
.text:6CD7446D mov     [edi], cl
.text:6CD7446F movzx  ecx, byte ptr [eax+1]
.text:6CD74473 movzx  ebx, byte ptr [eax]
.text:6CD74476 mov     cl, [ebp+ecx+var_104]
.text:6CD7447D mov     bl, [ebp+ebx+var_104]
.text:6CD74484 shr     cl, 2
.text:6CD74487 shl     bl, 4
.text:6CD7448A or      cl, bl
.text:6CD7448C mov     [edi+1], cl
.text:6CD7448F movzx  ecx, byte ptr [eax+1]
.text:6CD74493 movzx  ebx, byte ptr [eax+2]
.text:6CD74497 mov     cl, [ebp+ecx+var_104]
.text:6CD7449E shl     cl, 6
.text:6CD744A1 or      cl, [ebp+ebx+var_104]
.text:6CD744A8 add     edi, 3
.text:6CD744AB mov     [edi-1], cl
.text:6CD744AE add     eax, 4
.text:6CD744B1 dec     esi
.text:6CD744B2 jnz    short loc_6CD74450
```

Figure 64

At the time of analysis, no live response has been provided by the C2 server. According to the Unit42 article, the server would respond with a DLL file with 4 exports: StartWorker, StopWorker, WorkerRun and DllEntryPoint. Even if we didn't receive a valid response from the server, we were able to find out that the malicious process allocates a new memory area in order to write the DLL code inside:

```

.text:6CD74C1C
.text:6CD74C1C loc_6CD74C1C:
.text:6CD74C1C push    esi
.text:6CD74C1D mov     esi, ds:VirtualAlloc
.text:6CD74C23 push    edi
.text:6CD74C24 push    4           ; flProtect
.text:6CD74C26 mov     edi, 2000h
.text:6CD74C2B push    edi           ; flAllocationType
.text:6CD74C2C push    dword ptr [ebx+50h] ; dwSize
.text:6CD74C2F push    dword ptr [ebx+34h] ; lpAddress
.text:6CD74C32 call    esi ; VirtualAlloc
.text:6CD74C34 mov     [ebp+lpAddress], eax
.text:6CD74C37 test    eax, eax
.text:6CD74C39 jnz    short loc_6CD74C4F

```

Figure 65

```

.text:6CD74C3B push    4           ; flProtect
.text:6CD74C3D push    edi           ; flAllocationType
.text:6CD74C3E push    dword ptr [ebx+50h] ; dwSize
.text:6CD74C41 push    eax           ; lpAddress
.text:6CD74C42 call    esi ; VirtualAlloc
.text:6CD74C44 mov     [ebp+lpAddress], eax
.text:6CD74C47 test    eax, eax
.text:6CD74C49 jz     loc_6CD74D00

```

The new area of memory has to be executable because the potential DLL has to run, and that's why the malware uses VirtualProtect in order to change the protection of the area:

<pre> .text:6CD749E6 push    4000h       ; dwFreeType .text:6CD749EB push    dword ptr [ebx-14h] ; dwSize .text:6CD749EE push    dword ptr [ebx-1Ch] ; lpAddress .text:6CD749F1 call    ds:VirtualFree .text:6CD749F7 jmp     short loc_6CD74A3F </pre>	<pre> .text:6CD74A30 .text:6CD74A30 loc_6CD74A30: .text:6CD74A30 lea    eax, [ebp+flOldProtect] .text:6CD74A33 push    eax           ; lpflOldProtect .text:6CD74A34 push    esi           ; flNewProtect .text:6CD74A35 push    ecx           ; dwSize .text:6CD74A36 push    dword ptr [ebx-1Ch] ; lpAddress .text:6CD74A39 call    ds:VirtualProtect </pre>
---	--

Figure 66

After the malicious code would be written in the new memory location, the process would pass the execution flow to the new DLL file, as shown in the figure below:

```

.text:6CD74CEC push    0
.text:6CD74CEE xor     ebx, ebx
.text:6CD74CF0 inc     ebx
.text:6CD74CF1 push    ebx
.text:6CD74CF2 push    esi
.text:6CD74CF3 call    eax
.text:6CD74CF5 test    eax, eax
.text:6CD74CF7 jnz    short loc_6CD74D04

```

Figure 67

References

Unit42 report: <https://unit42.paloaltonetworks.com/new-attacks-linked-to-c0d0s0-group/>

VirusTotal link:

<https://www.virustotal.com/gui/file/de33dfce8143f9f929abda910632f7536ffa809603ec027a4193d5e57880b292/detection>

MSDN: <https://docs.microsoft.com/en-us/windows/win32/api/>

Fakenet: <https://github.com/fireeye/flare-fakenet-ng>

FireEye: <https://www.fireeye.com/current-threats/apt-groups.html#apt19>

## INDICATORS OF COMPROMISE

C2 domain: www.microsoft-cache[.]com

C2 IP address: 106.185.43.96

SHA256:

DE33DFCE8143F9F929ABDA910632F7536FFA809603EC027A4193D5E57880B292

URLs: 106.185.43.96/user/atv.html

www.microsoft-cache[.]com:443/money/ofcom-fines-nuisance-calls?  
0023528460592137&data=<12 random chars>

www.microsoft-cache[.]com:443/world/video/shrien-dewani-arrives-uk-murder-trial-collapses-  
video?0023528461146965&data=<12 random chars>

www.microsoft-cache[.]com:443/lifeandstyle/marmalade-paddington-sales-up-making-  
drinking?0023528460592137&data=<12 random chars>

## Yara rules for detecting the threat

```
rule APT19_1 {
  meta:
    author = "CyberMasterV"
    Date = "2020-12-26"

  strings:
    $s1 = "http://www.google.com" wide ascii
    $s2 = "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_2; en-US)
AppleWebKit/533.3 (KHTML, like Gecko) Chrome/5.0.354.0 Safari/533.3" wide ascii
    $s3 = "%s?%016I64d&data=%s"
    $s4 = "DebugCreate"
    $s5 = "DebugConnect"
  condition:
    4 of them
}
```

```
rule APT19_2 {
  meta:
    author = "CyberMasterV"
    Date = "2020-12-26"

  strings:
    $s1 = "DbgEng.Dll" wide ascii
    $s2 = "Windows Helper Service"
    $s3 = "WinHelpSrv"
    $s4 = "KBKBKBKBKBKB"
  condition:
    3 of them
}
```