Analyzing APT19 malware using a step-by-step method – CYBER GEEKS

Published: 2020-12-26 · Archived: 2026-04-05 13:11:51 UTC

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:

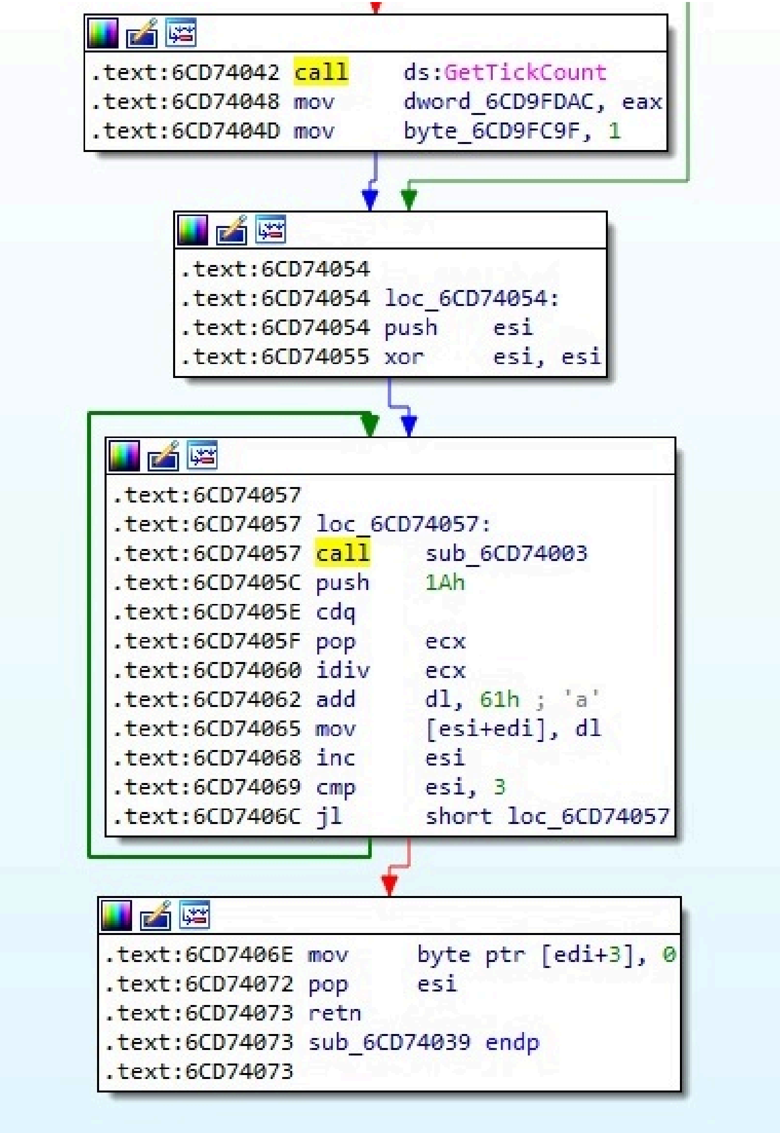

Figure 2

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, 0FFFFFFFFh
.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:

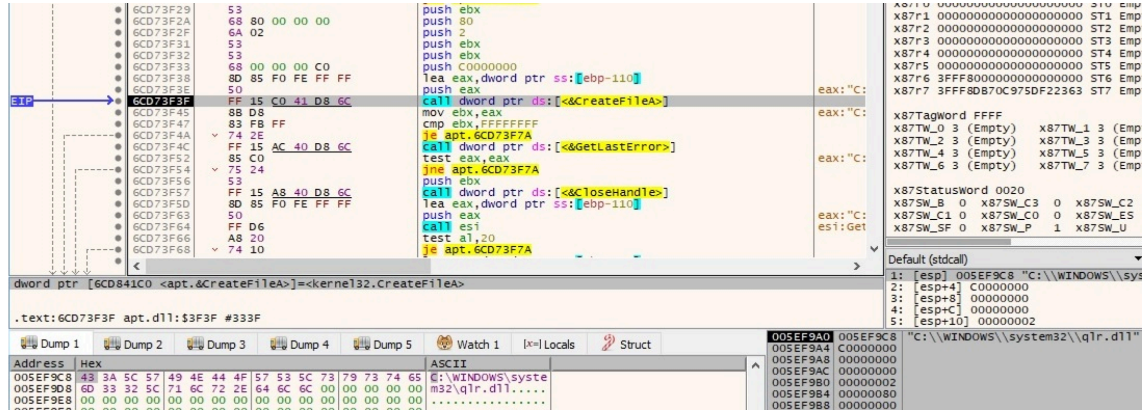

Figure 4

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

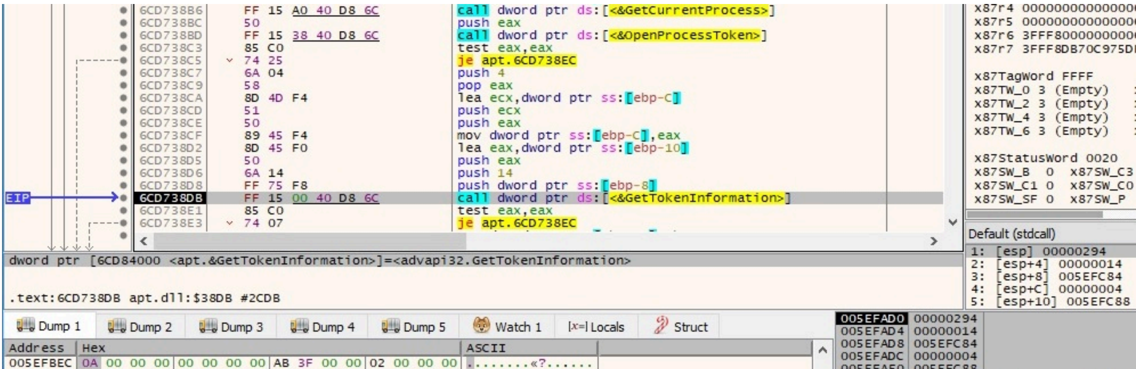

Figure 5

Malware running with admin privileges

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

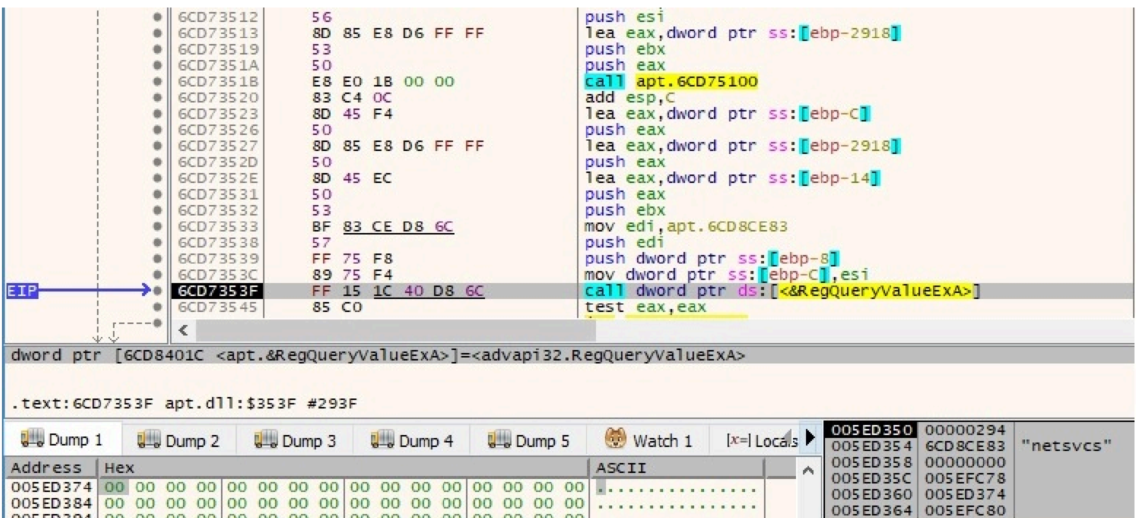

Figure 6

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

Address	Hex	ASCII
005ED374	43 65 72 74 50 72 6F 70 53 76 63 00 53 43 50 6F	CertPropSvc.SCPo
005ED384	6C 69 63 79 53 76 63 00 6C 61 6E 6D 61 6E 73 65	licySvc.lanmanse
005ED394	72 76 65 72 00 67 70 73 76 63 00 69 70 68 6C 70	rver.gpsvc.iphlp
005ED3A4	73 76 63 00 6D 73 69 73 63 73 69 00 73 63 68 65	svc.msiscsi.sche
005ED3B4	64 75 6C 65 00 77 69 6E 6D 67 6D 74 00 53 65 73	dule.winmgmt.Ses
005ED3C4	73 69 6F 6E 45 6E 76 00 46 61 73 74 55 73 65 72	sionEnv.FastUser
005ED3D4	53 77 69 74 63 68 69 6E 67 43 6F 6D 70 61 74 69	SwitchingCompati
005ED3E4	62 69 6C 69 74 79 00 49 61 73 00 49 72 6D 6F 6E	bility.Ias.Irmon
005ED3F4	00 4E 6C 61 00 4E 74 6D 73 73 76 63 00 4E 57 43	.Nla.Ntmssvc.NWC
005ED404	57 6F 72 6B 73 74 61 74 69 6F 6E 00 4E 77 73 61	Workstation.Nwsa
005ED414	70 61 67 65 6E 74 00 52 61 73 61 75 74 6F 00 52	pagent.Rasauto.R
005ED424	61 73 6D 61 6E 00 52 65 6D 6F 74 65 61 63 63 65	asman.Remoteacce
005ED434	73 73 00 53 45 4E 53 00 53 68 61 72 65 64 61 63	ss.SENS.Sharedac
005ED444	63 65 73 73 00 53 52 53 65 72 76 69 63 65 00 54	cess.SRService.T
005ED454	61 70 69 73 72 76 00 57 6D 69 00 57 6D 64 6D 50	apisrv.Wmi.WmdmP
005ED464	6D 53 70 00 77 75 61 75 73 65 72 76 00 42 49 54	mSp.wuauserv.BIT
005ED474	53 00 53 68 65 6C 6C 48 57 44 65 74 65 63 74 69	S.ShellHWDetecti
005ED484	6F 6E 00 4C 6F 67 6F 6E 48 6F 75 72 73 00 50 43	on.LogonHours.PC
005ED494	41 75 64 69 74 00 68 65 6C 70 73 76 63 00 75 70	Audit.helpsvc.up
005ED4A4	6C 6F 61 64 6D 67 72 00 54 6F 6B 65 6E 42 72 6F	loadmgr.TokenBro
005ED4B4	6B 65 72 00 55 73 65 72 4D 61 6E 61 67 65 72 00	ker.UserManager.
005ED4C4	41 70 70 4D 67 6D 74 00 00 00 6F 00 00 00 52 00	AppMgmt...o...R.

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:

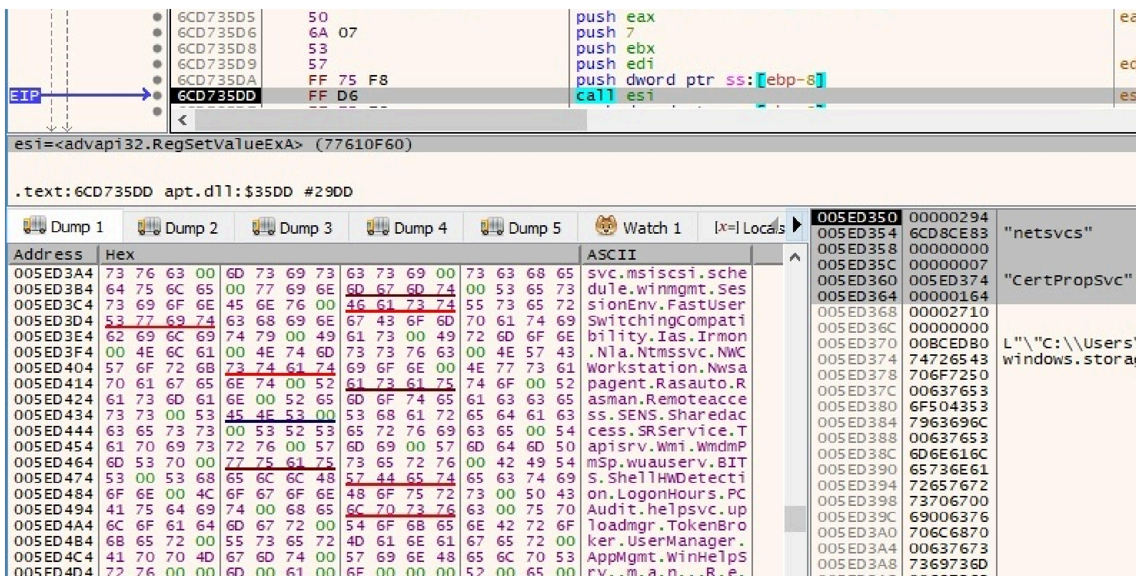

Figure 8

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

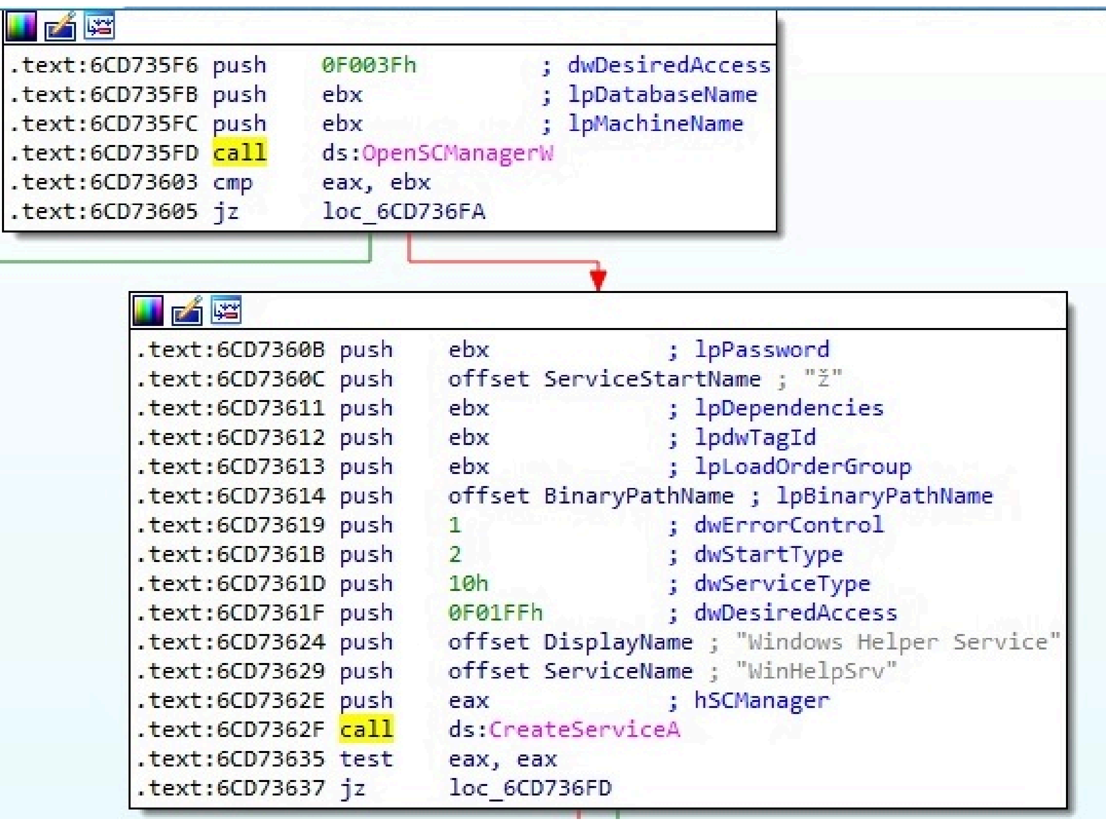

Figure 9

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:

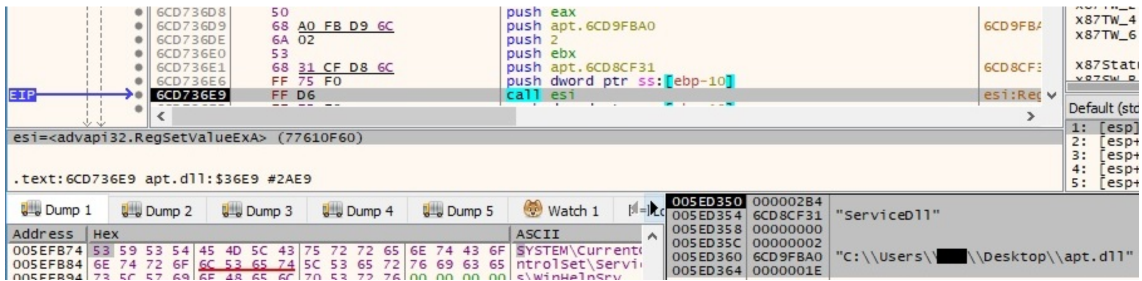

Figure 11

The confirmation that the operation was successful:

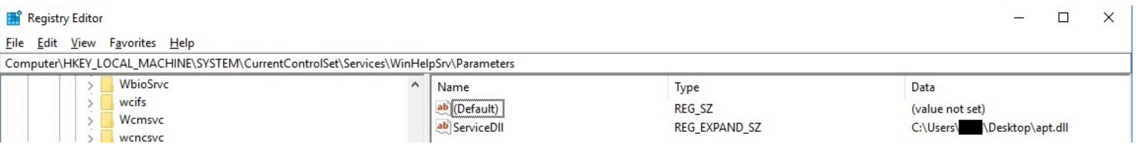

Figure 12

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

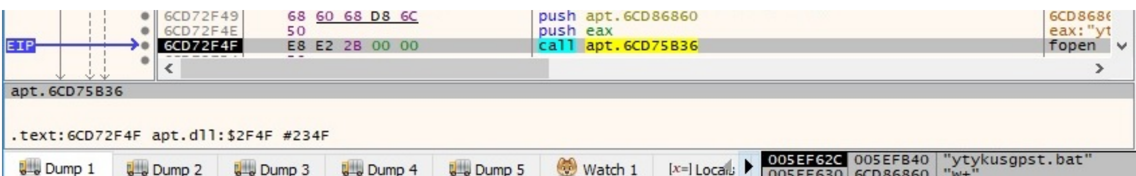

Figure 13

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):

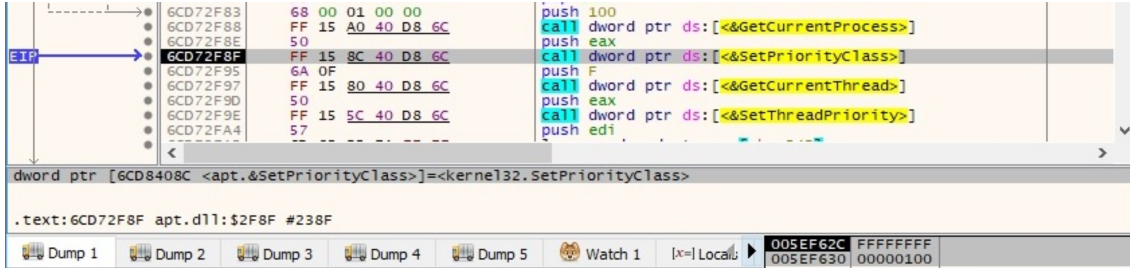

Figure 14

After this operation, there is a call to SetThreadPriority that sets the priority 15 (THREAD_PRIORITY_TIME_CRITICAL) for the current thread:

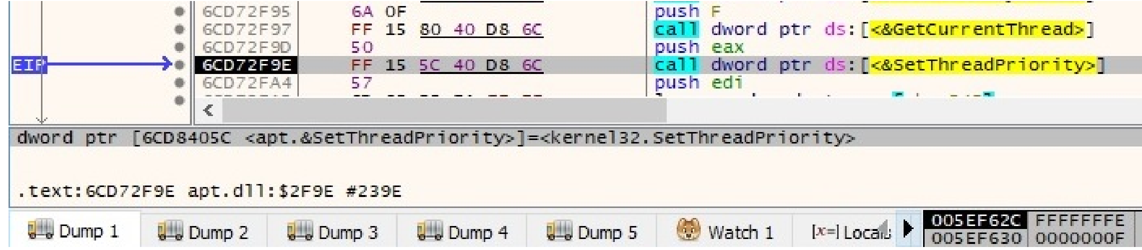

Figure 15

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:

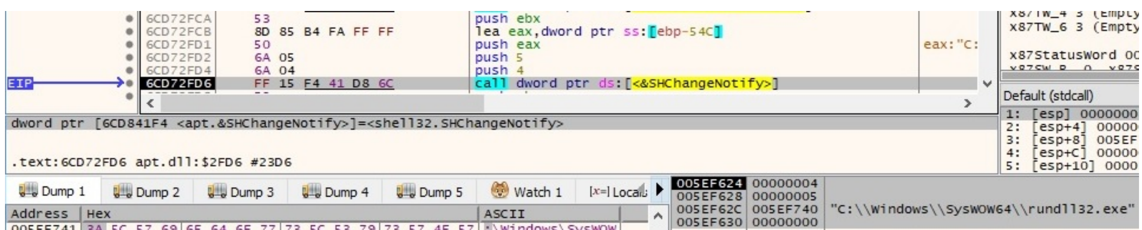

Figure 16

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

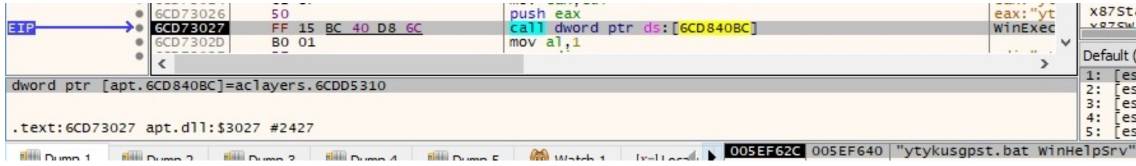

Figure 17

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:

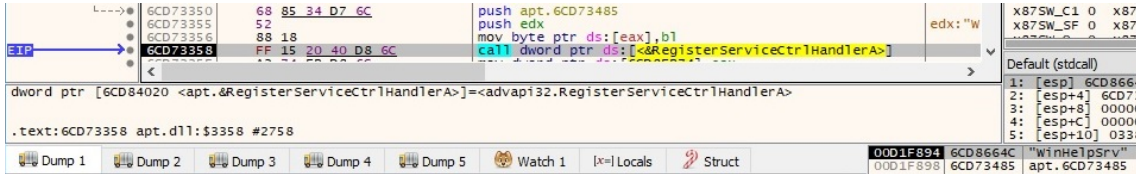

Figure 18

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):

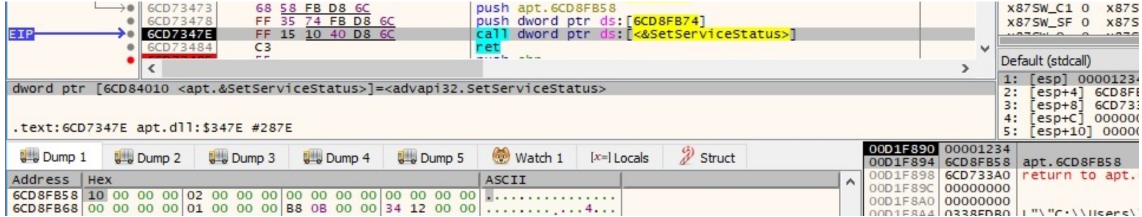

Figure 19

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

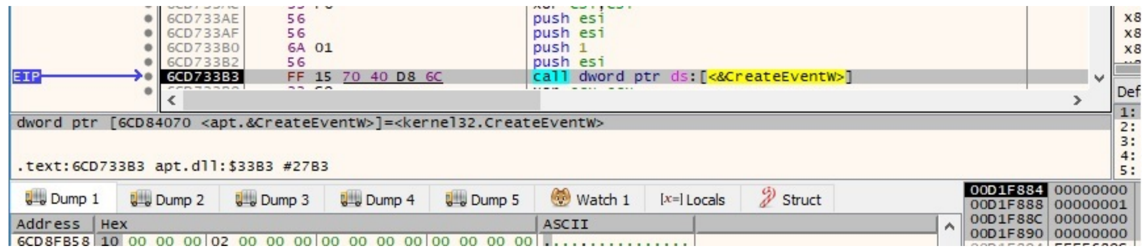

Figure 20

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):

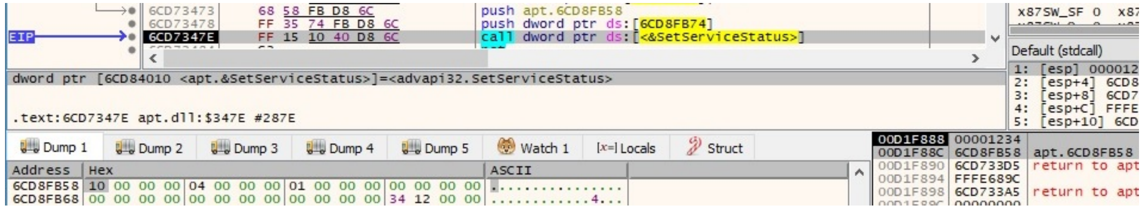

Figure 21

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:

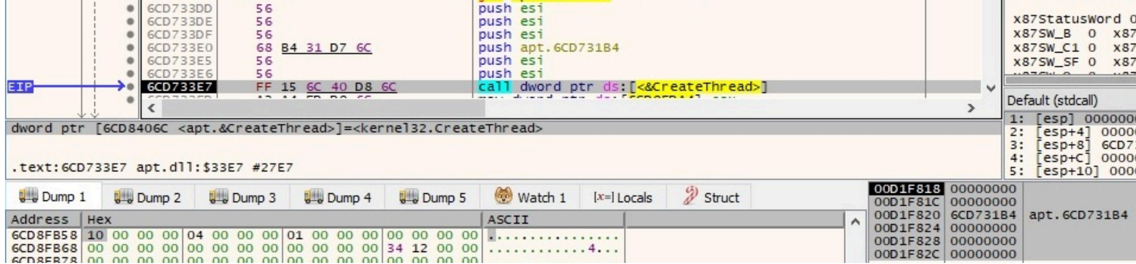

Figure 22

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 ; 'þ'       ; 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:
```

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:

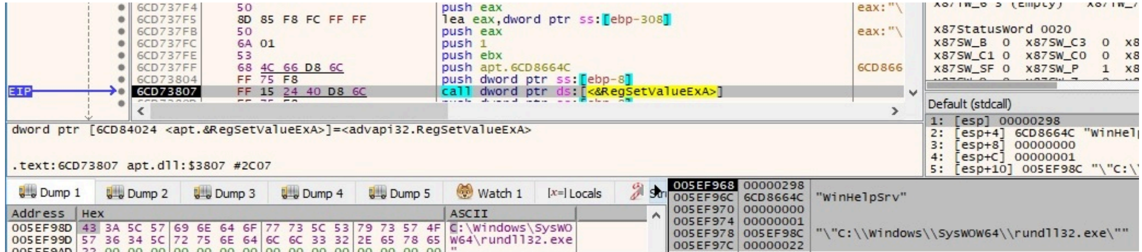

Figure 24

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:

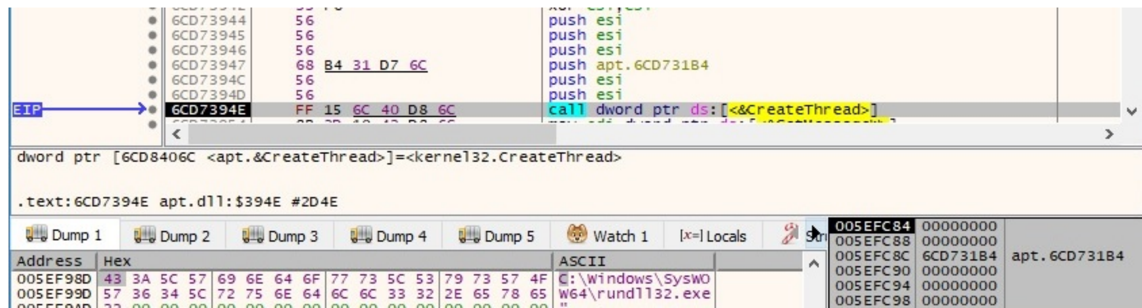

Figure 26

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:

```
.text:6CD73987
.text:6CD73987 loc_6CD73987:                    ; wMsgFilterMax
.text:6CD73987 push    esi
.text:6CD73988 push    esi                      ; wMsgFilterMin
.text:6CD73989 push    esi                      ; hWnd
.text:6CD7398A lea     eax, [ebp+Msg]
.text:6CD7398D push    eax                      ; lpMsg
.text:6CD7398E call    edi ; GetMessageW
.text:6CD73990 test    eax, eax
.text:6CD73992 jnz     short loc_6CD73961

.text:6CD73961
.text:6CD73961 loc_6CD73961:
.text:6CD73961 cmp     [ebp+Msg.message], 10h
.text:6CD73965 jz      short loc_6CD73994

.text:6CD73967 cmp     [ebp+Msg.message], 16h
.text:6CD7396B jz      short loc_6CD73994

.text:6CD7396D cmp     [ebp+Msg.message], 11h
.text:6CD73971 jz      short loc_6CD73994

:6CD73973 lea     eax, [ebp+Msg]
:6CD73976 push    eax                  ; lpMsg
:6CD73977 call    ds:TranslateMessage
:6CD7397D lea     eax, [ebp+Msg]
:6CD73980 push    eax                  ; lpMsg
:6CD73981 call    ds:DispatchMessageW

.text:6CD73994
.text:6CD73994 loc_6CD73994:
.text:6CD73994 xor     eax, eax
.text:6CD73996 pop     edi
.text:6CD73997 inc     eax
.text:6CD73998 pop     esi
```

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:

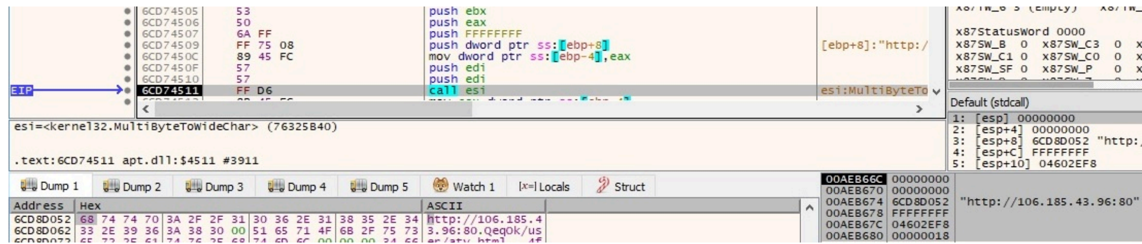

Figure 28

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

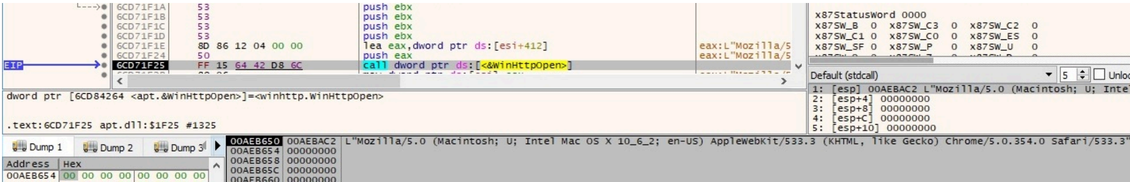

Figure 29

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):

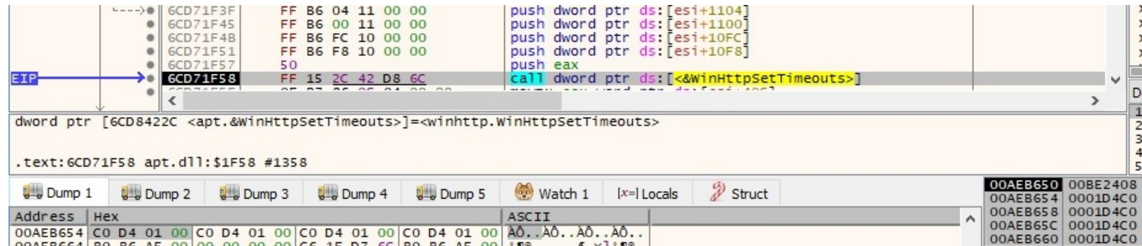

Figure 30

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.

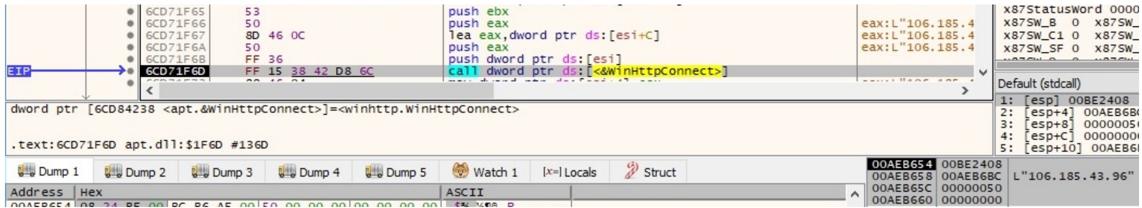

Figure 31

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):

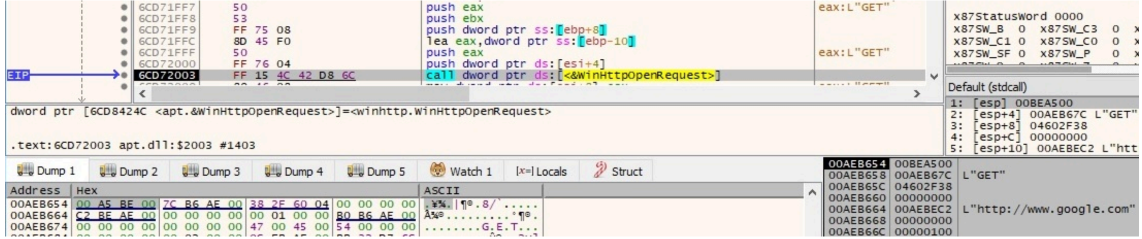

Figure 32

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) AppleWe
bKit/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 WinHttpReceiveResponse function:

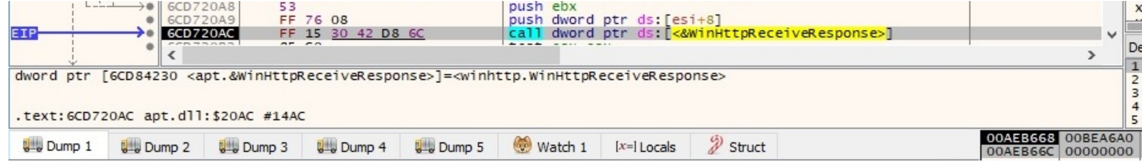

Figure 34

Afterward, the malicious file retrieves header information using WinHttpQueryHeaders API with 0x16 (WINHTTP_QUERY_RAW_HEADERS_CRLF) parameter – receives all the headers returned by the HTTP server:

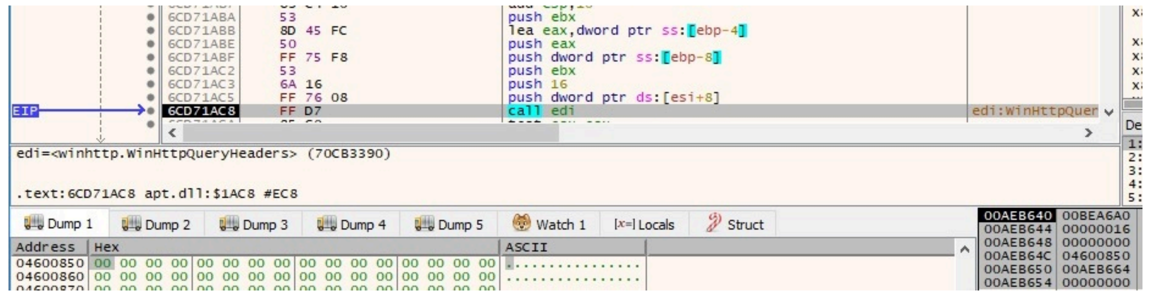

Figure 35

There is a second WinHttpQueryHeaders 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):

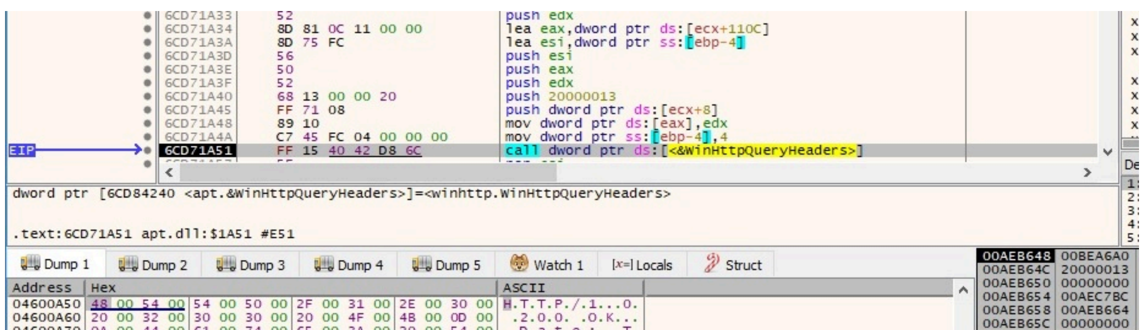

Figure 36

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

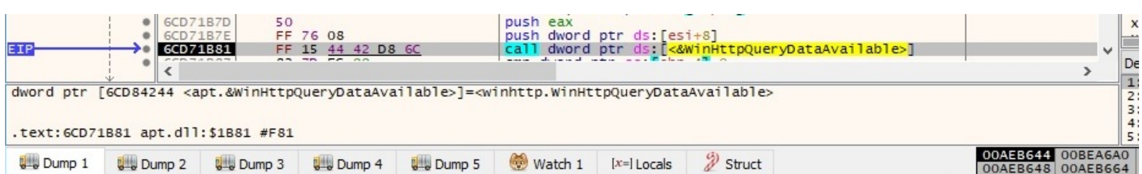

Figure 37

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

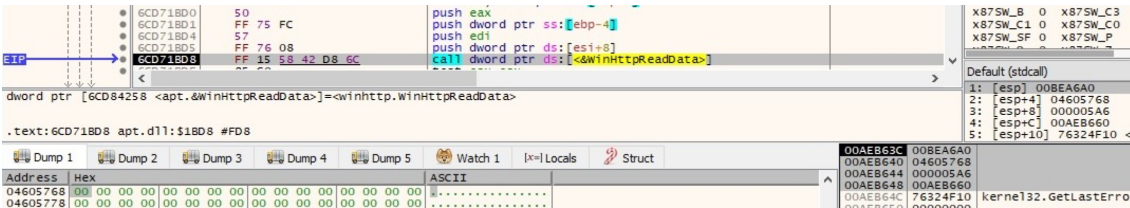

Figure 38

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):

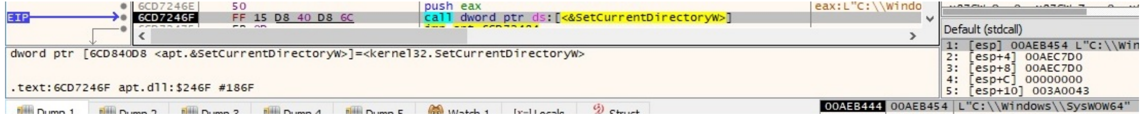

Figure 39

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

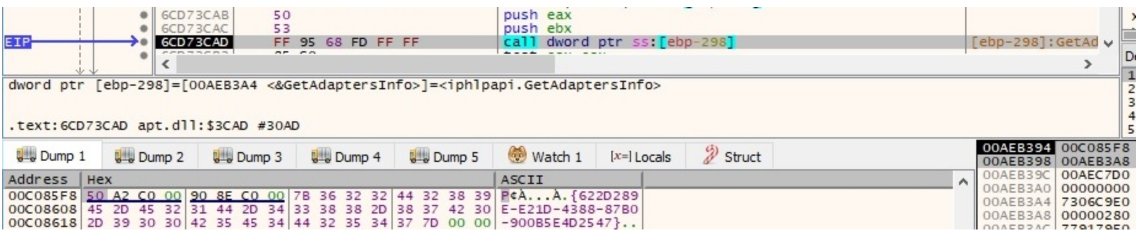

Figure 40

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

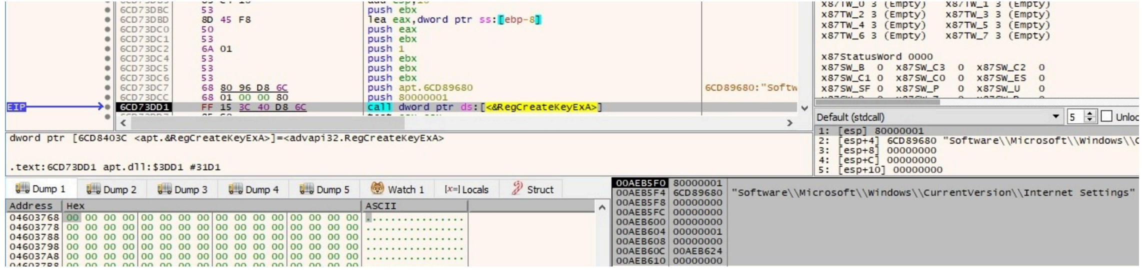

Figure 41

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

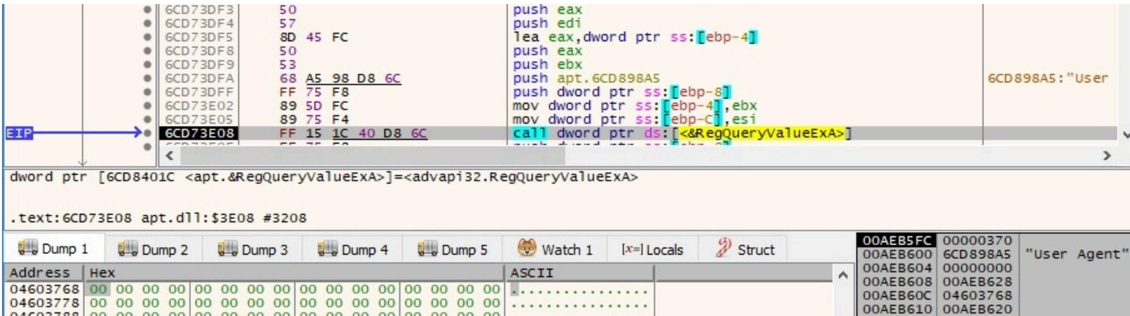

Figure 42

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

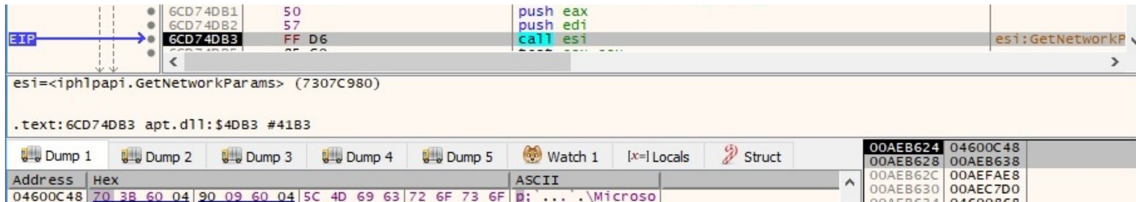

Figure 43

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:

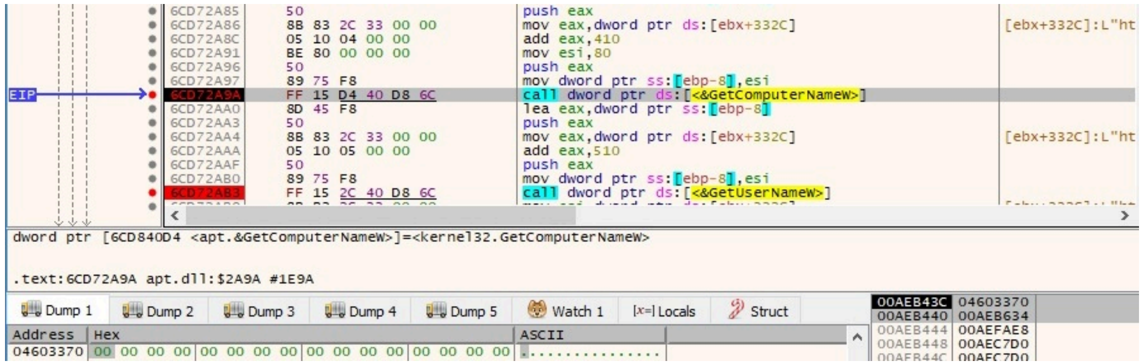

Figure 44

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:

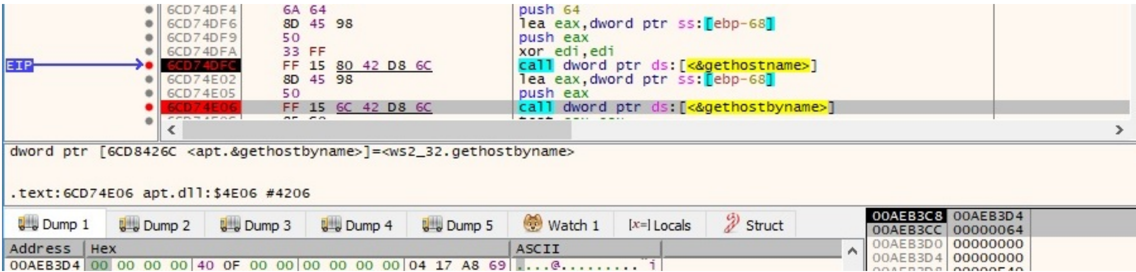

Figure 45

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:

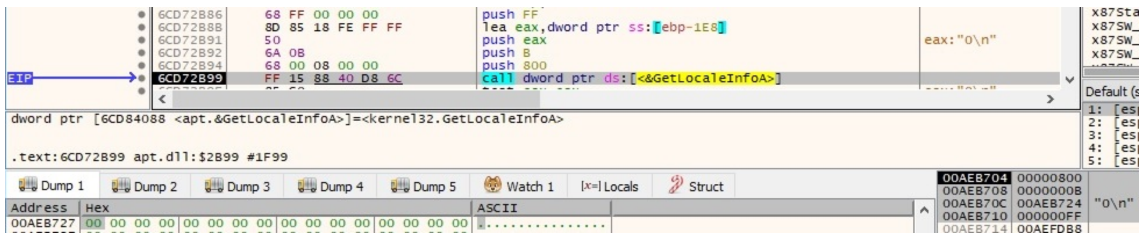

Figure 46

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 cpuid 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?00235284611469 65&data=qgvuclxxlgip". The function WinHttpOpen is called using the user agent extracted earlier from registry, "Mozilla/4.0 (compatible; MSIE 8.0; Win32)":

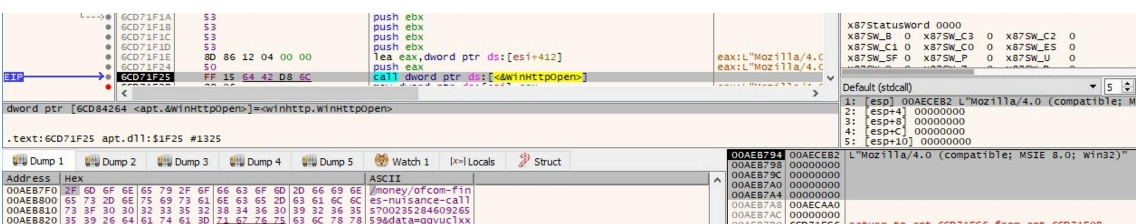

Figure 48

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:

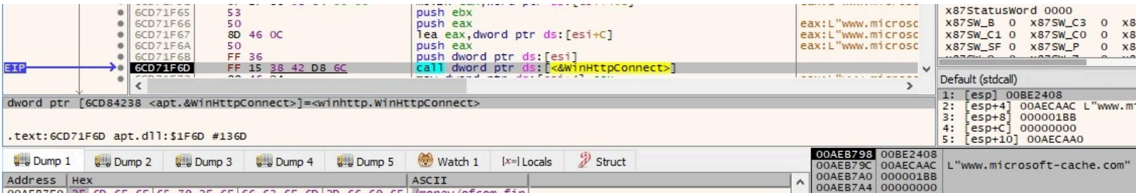

Figure 49

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

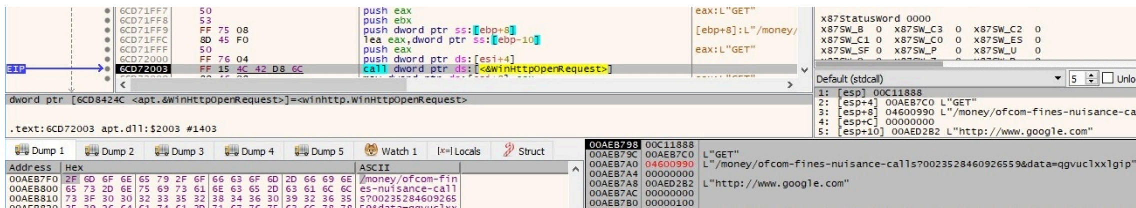

Figure 50

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:

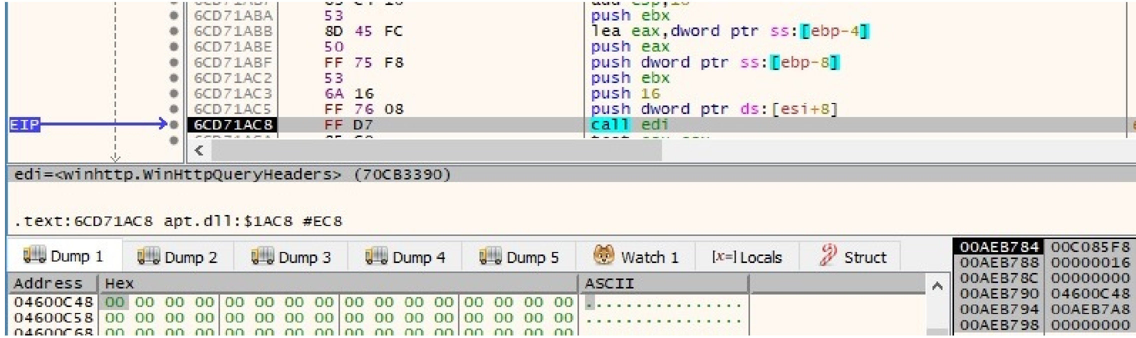

Figure 51

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:

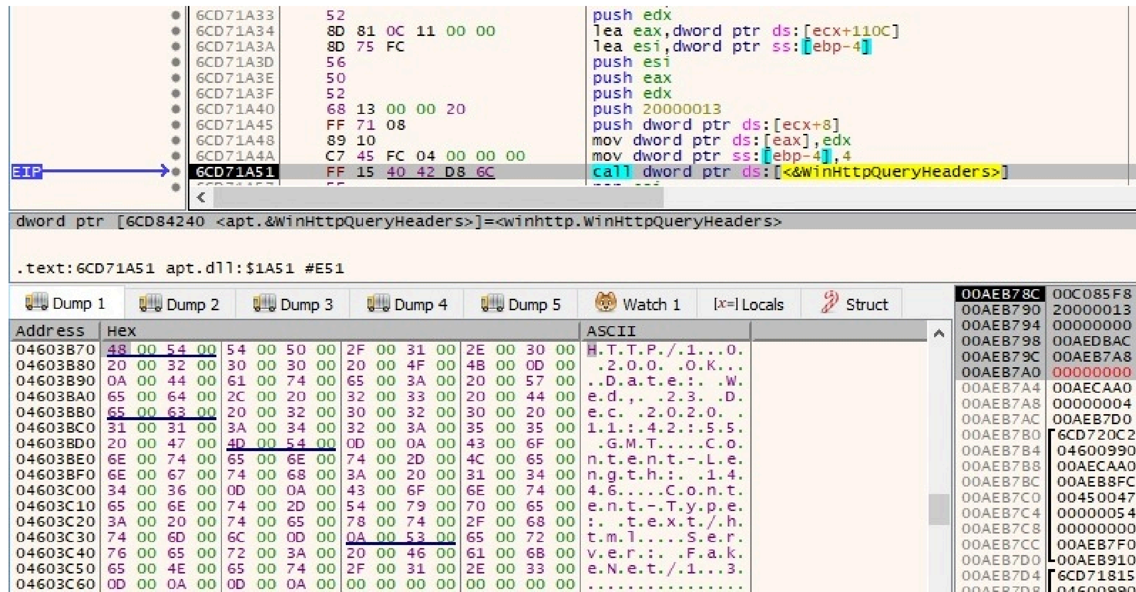

Figure 52

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

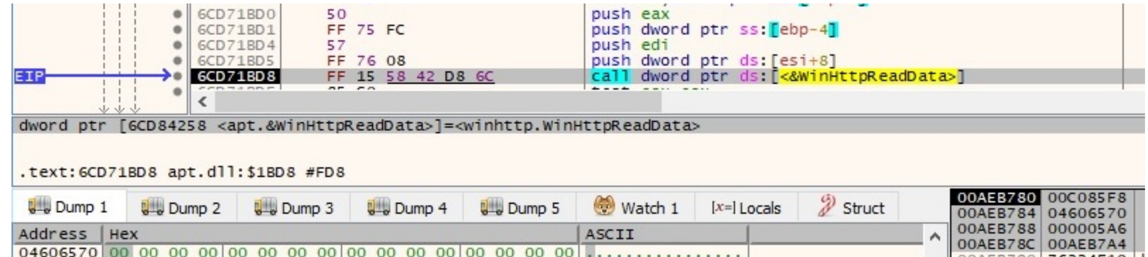

Figure 53

The buffer containing the information that will be exfiltrated is XORed byte-by-byte with a one-byte key. The following information belongs to the buffer: the C2 server address, hostname, username, IP address represented as hex values, 01 constant because the process is running on a 64-bit environment, the result of GetLocaleInfoA call (0x1b5 = 437 in our case), the amount of physical memory represented as a 32-bit value, the amount of physical memory currently available represented as a 32-bit value, the processor name, the width of the screen of the primary display monitor represented as a 32-bit value (0x780 = 1920 in our case) and the height of the screen of the primary display monitor represented as a 32-bit value (0x438 = 1080 in our case):

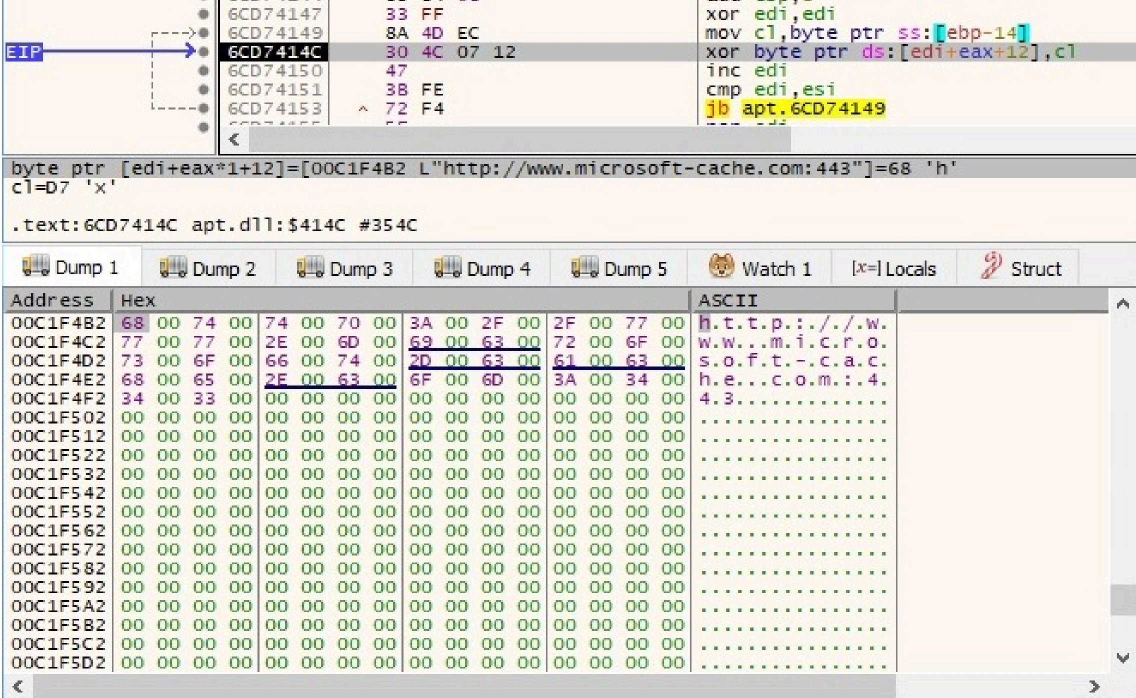

Figure 54

After the operation is complete, the buffer looks like in the following picture:

```
Address   Hex                                              ASCII
00C1F4B2  BF D7 A3 D7 A3 D7 A7 D7 ED D7 F8 D7 F8 D7 A0 D7  ¿×£×£×§×í×ø×ø× ×
00C1F4C2  A0 D7 A0 D7 F9 D7 BA D7 BE D7 B4 D7 A5 D7 B8 D7   × ×ù×º×¾×´×¥×¸×
00C1F4D2  A4 D7 B8 D7 B1 D7 A3 D7 FA D7 B4 D7 B6 D7 B4 D7  ¤×¸×±×£×ú×´×¶×´×
00C1F4E2  BF D7 B2 D7 F9 D7 B4 D7 B8 D7 BA D7 ED D7 E3 D7  ¿×²×ù×´×¸×º×í×ã×
00C1F4F2  E3 D7 E4 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ã×ä×××××××××××××
00C1F502  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F512  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F522  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F532  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F542  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F552  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F562  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F572  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F582  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F592  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F5A2  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F5B2  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F5C2  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
00C1F5D2  D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7 D7  ××××××××××××××××
```

Figure 55

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	41 4E 63 43	62 6B 51 41	42 41 41 41	EggAANcCbkQABAAA
04603D78	41 41 41 41	43 41 41 41	76 39 65 6A	31 36 50 58	AAAACAAAv9ej16PX
04603D88	70 39 66 74	31 2F 6A 58	2B 4E 65 67	31 36 44 58	p9ft1/jX+Neg16DX
04603D98	6F 4E 66 35	31 37 72 58	76 74 65 30	31 36 58 58	oNf517rXvte016XX
04603DA8	75 4E 65 6B	31 37 6A 58	73 64 65 6A	31 2F 72 58	uNek17jXsdej1/rX
04603DB8	74 4E 65 32	31 37 54 58	76 39 65 79	31 2F 6E 58	tNe217TXv9ey1/nX
04603DC8	74 4E 65 34	31 37 72 58	37 64 66 6A	31 2B 50 58	tNe417rX7dfj1+PX
04603DD8	35 4E 66 58	31 39 66 58	31 39 66 58	31 39 66 58	5NfX19fX19fX19fX
04603DE8	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603DF8	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E08	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E18	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E28	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E38	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E48	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E58	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E68	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E78	31 39 66 58	31 39 66 58	31 39 66 58	31 39 66 58	19fX19fX19fX19fX
04603E88	31 39 66 58	31 39 66 58	31 39 66 58	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 WinHttpOpenRequest function (as before, the data parameter contains randomly-generated characters):

Figure 58

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

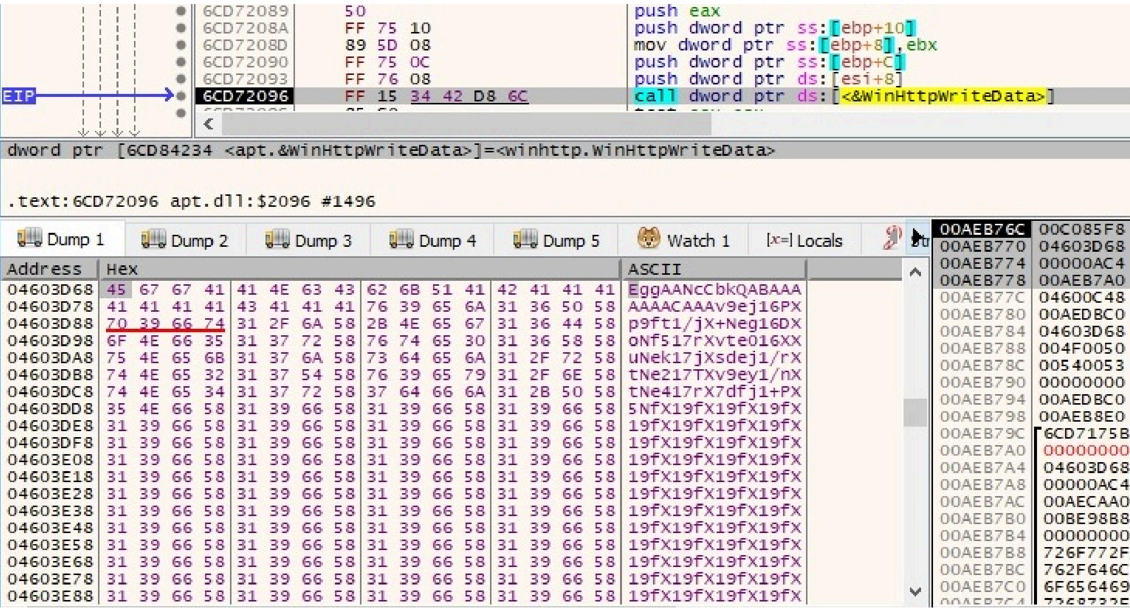

Figure 59

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:

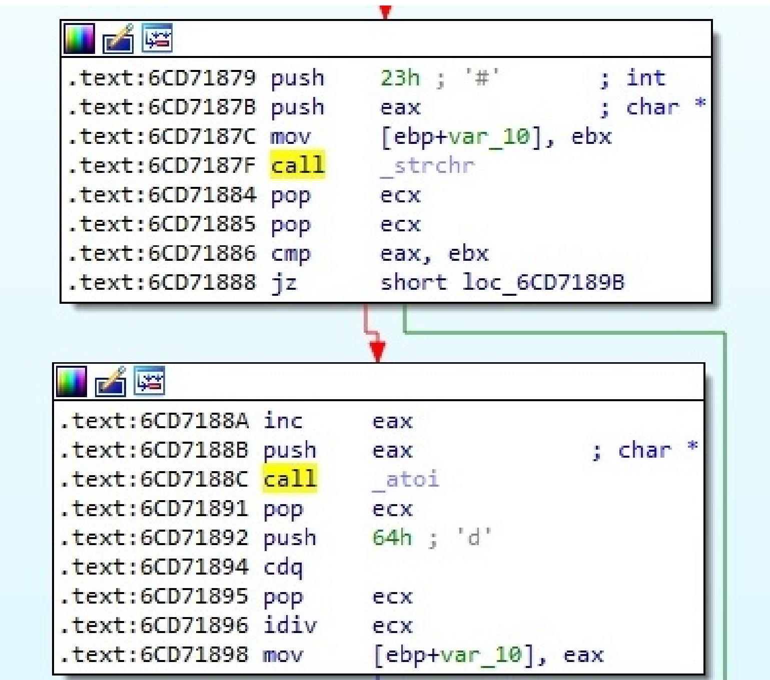

Figure 61

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):

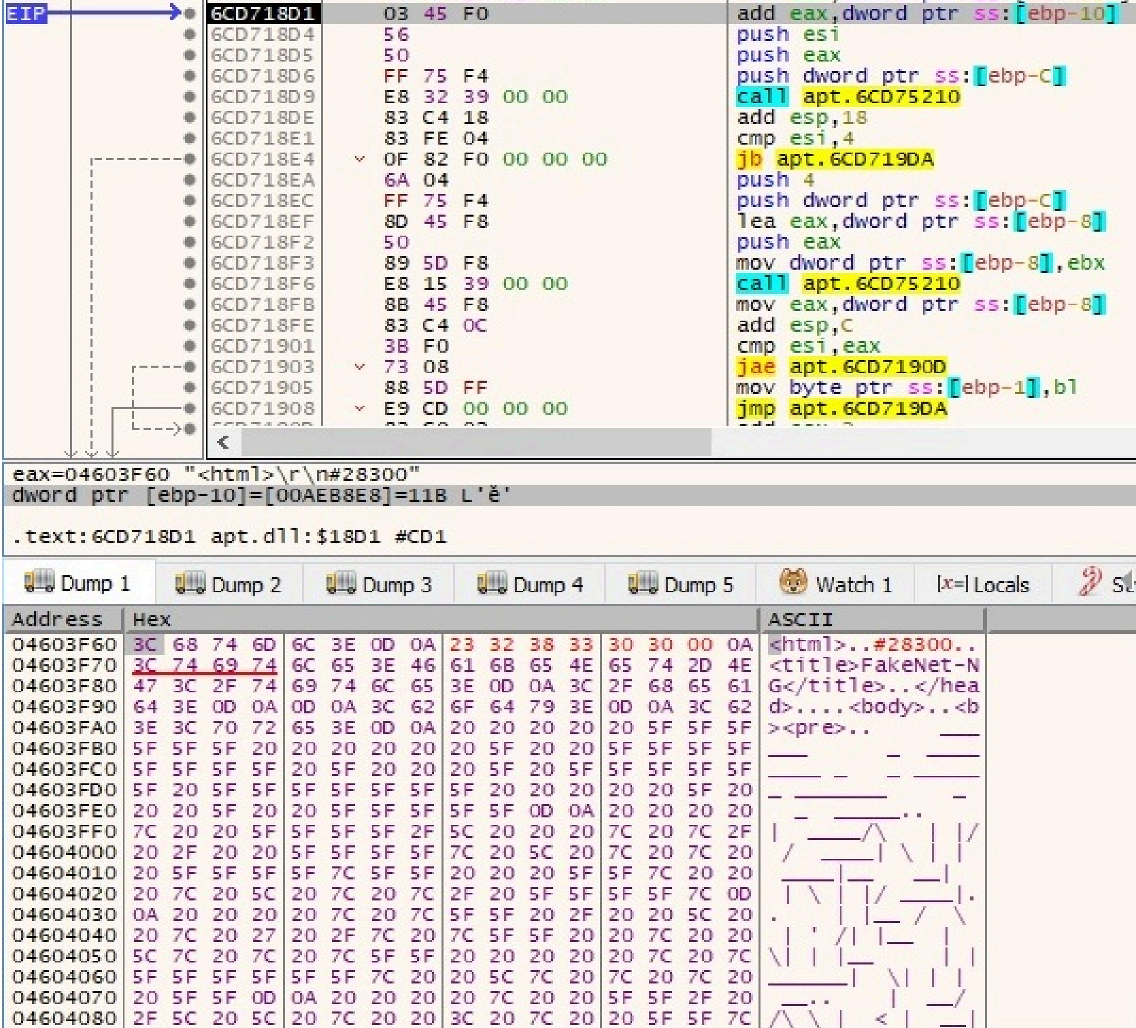

Figure 62

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 ; "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklm"...
.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

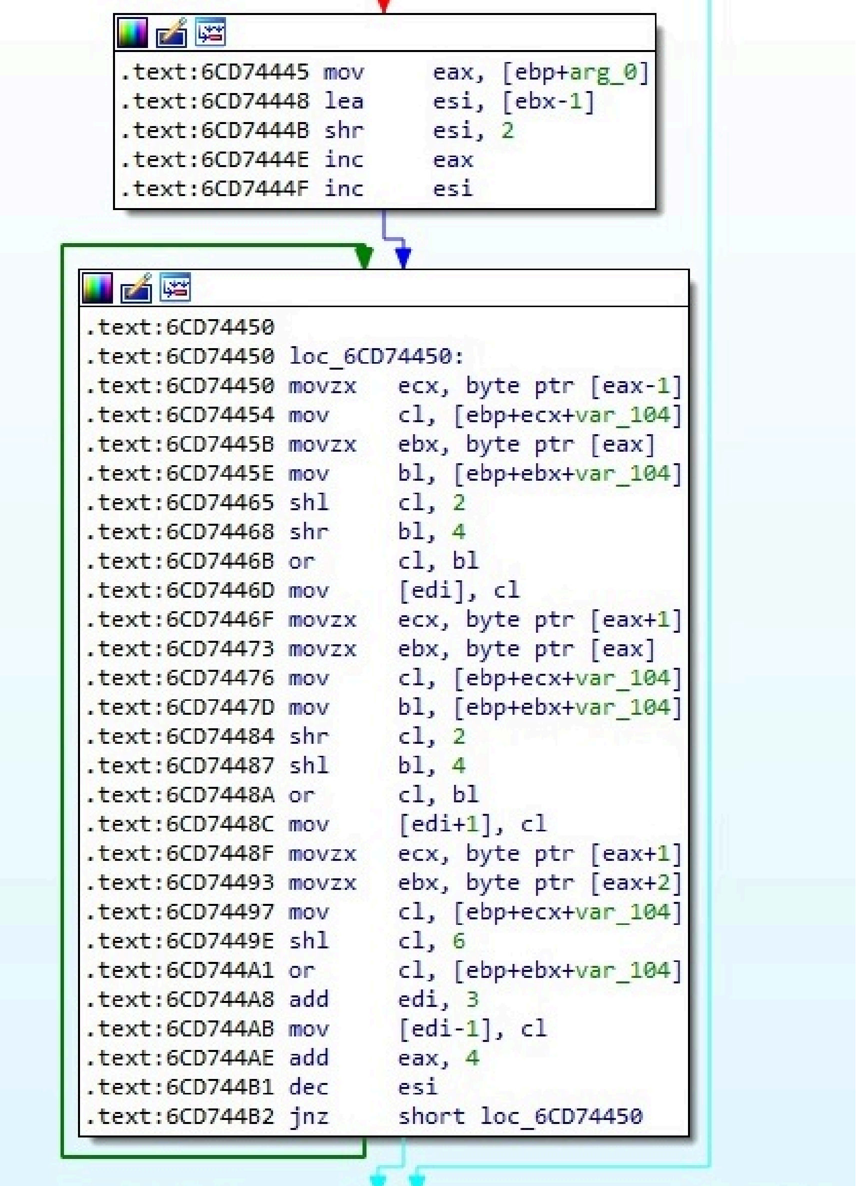

Figure 64

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

.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
```

Figure 65

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:

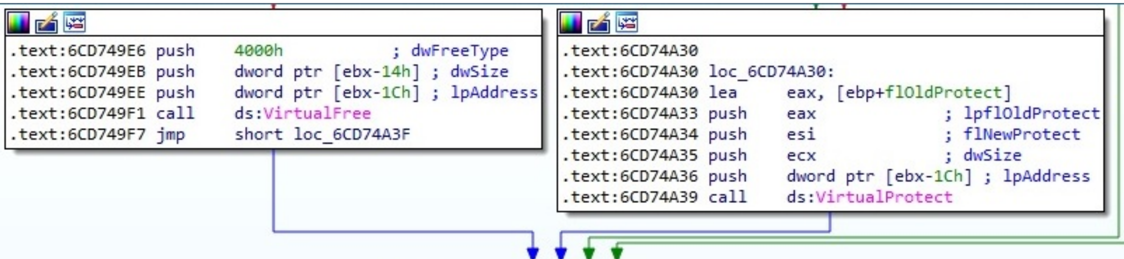

Figure 66

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:

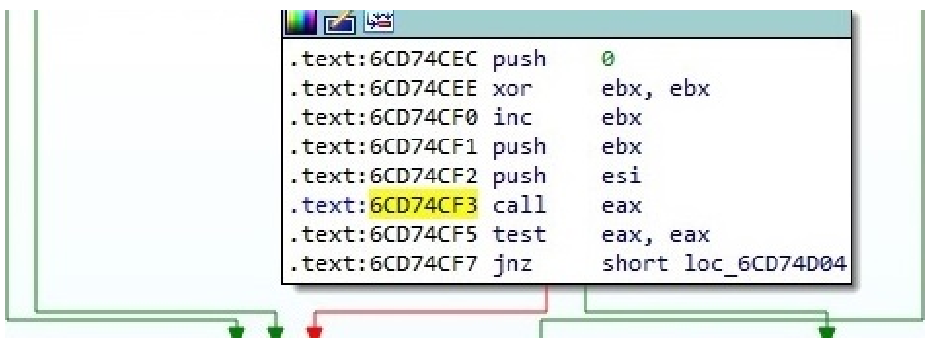

Figure 67

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) Chro
         $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
}
```

Source: https://cybergeeks.tech/analyzing-apt19-malware-using-a-step-by-step-method/