Analyzing APT19 malware using a step-by-step method

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:

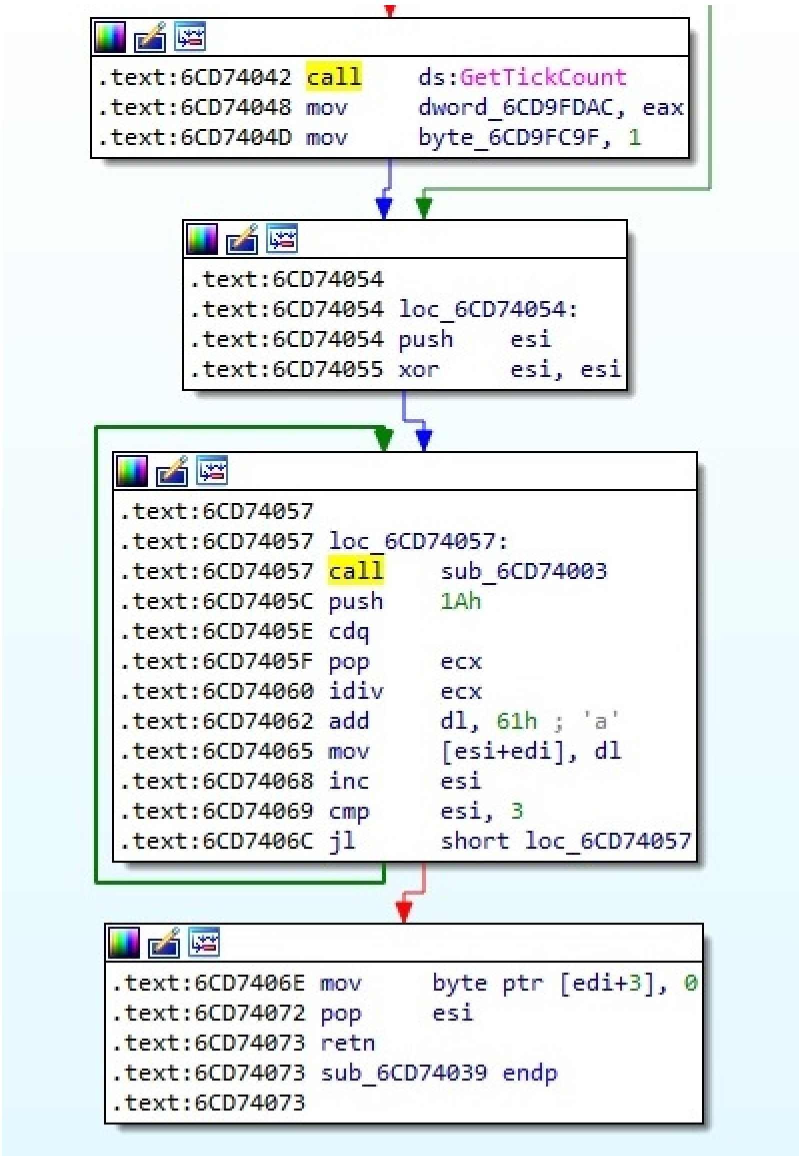

Figure 2

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, 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:

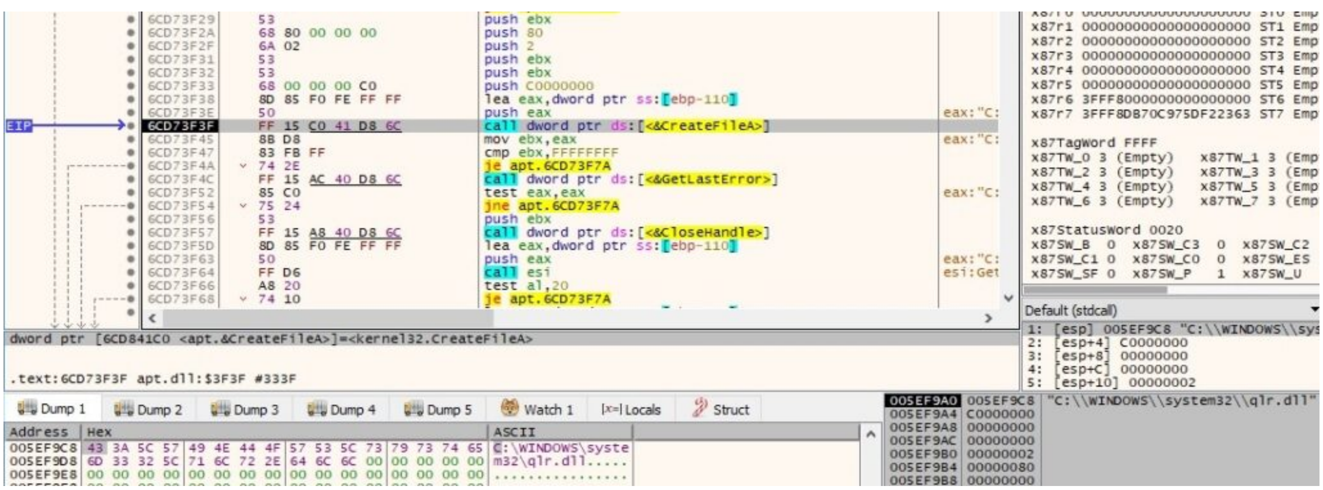

Figure 4

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

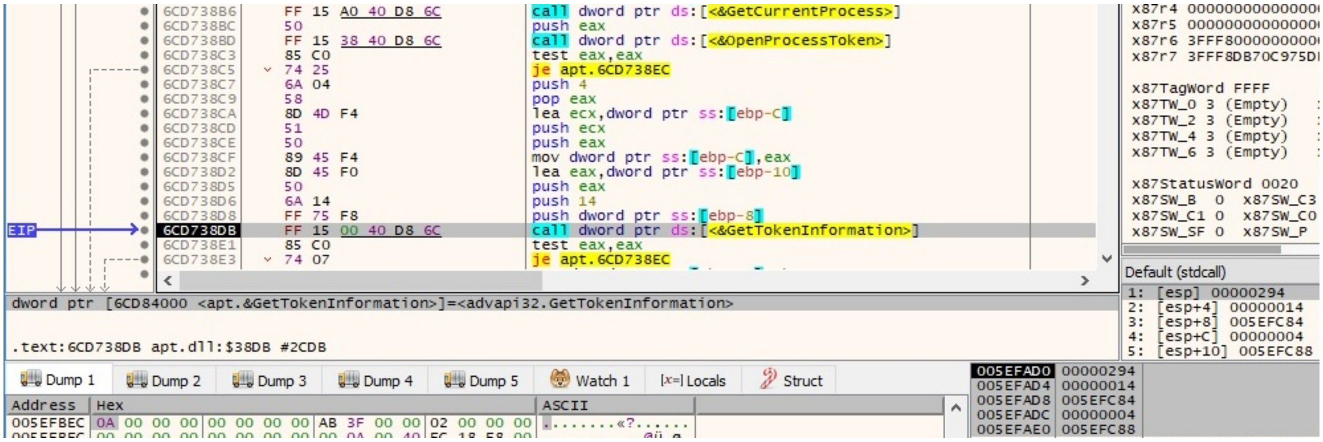

Figure 5

Malware running with admin privileges

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

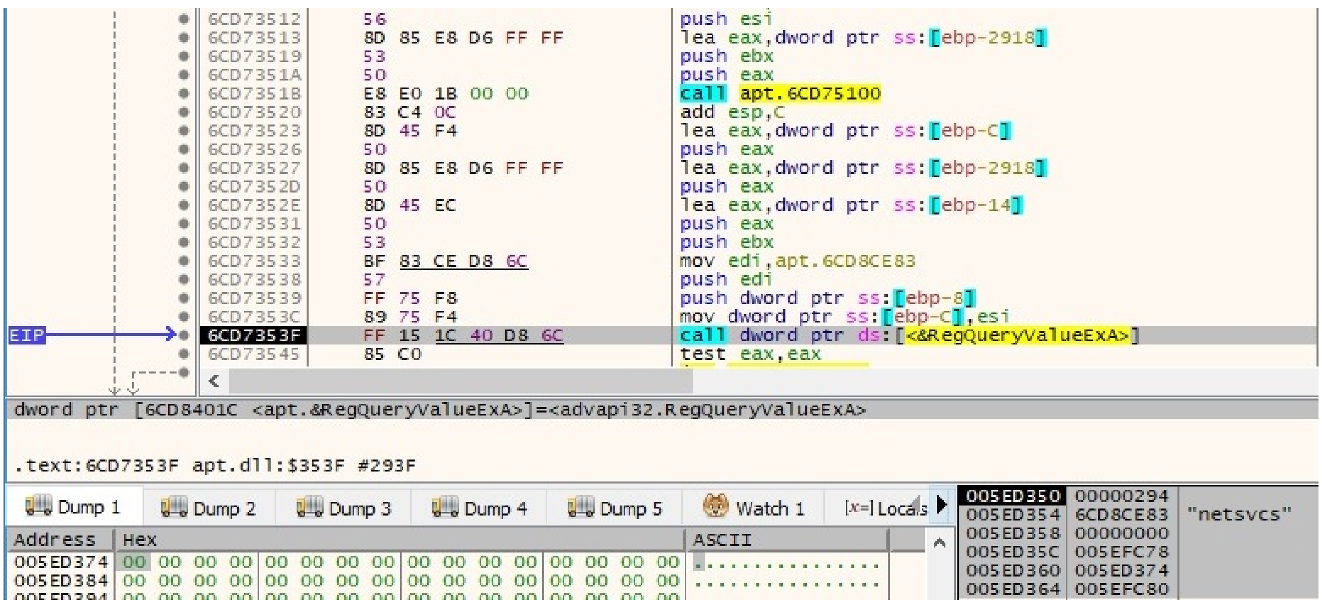

Figure 6

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

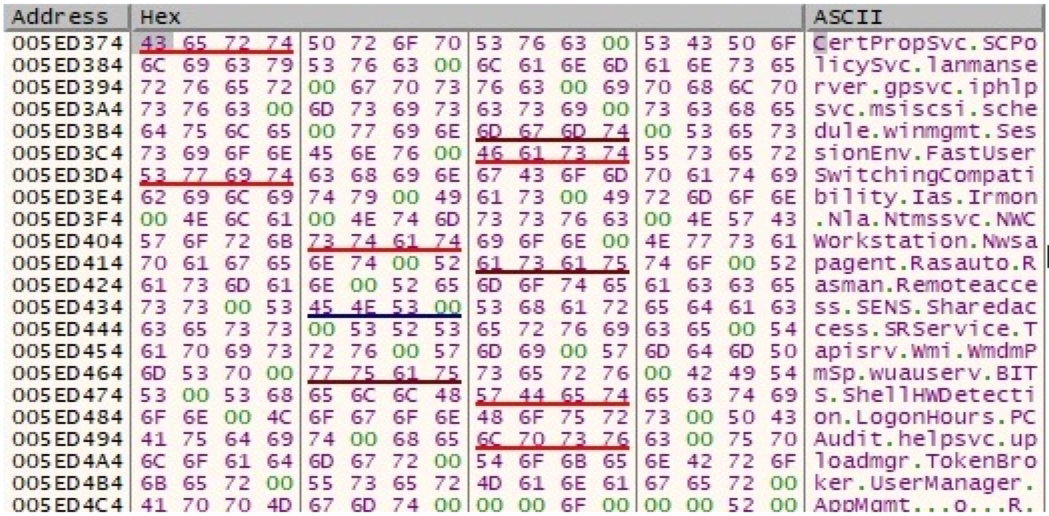

Figure 7

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:

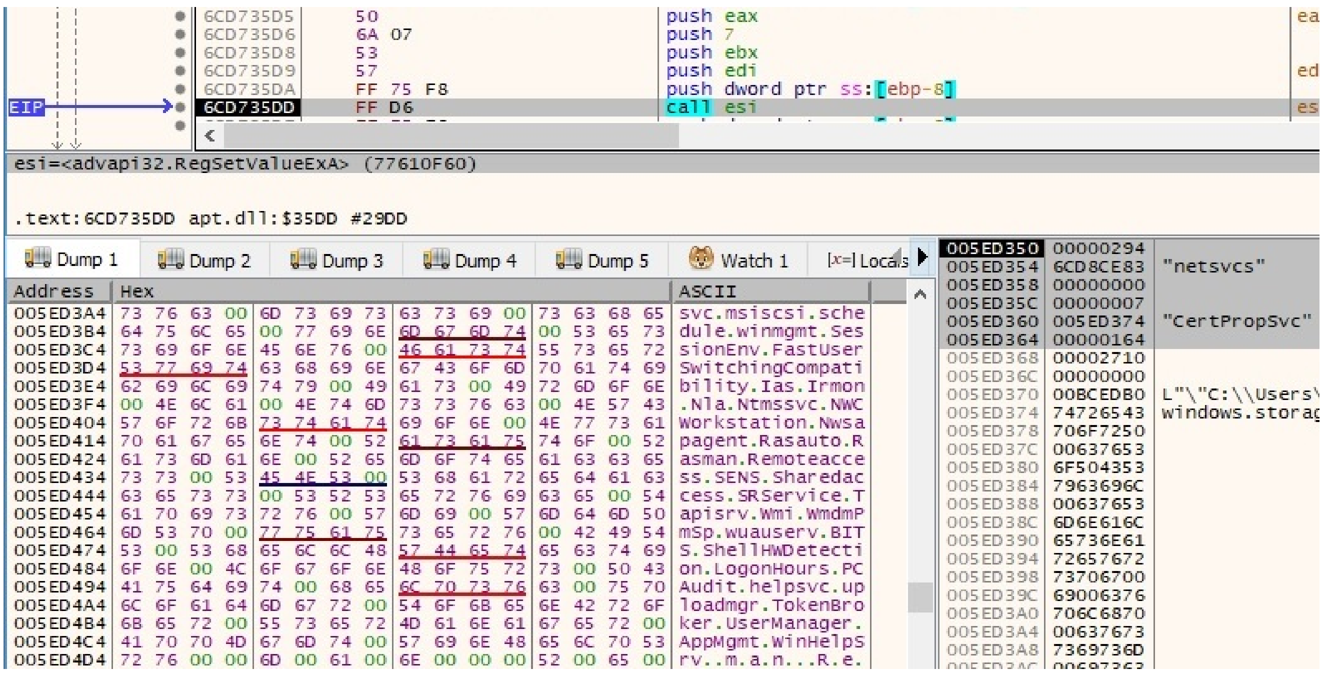

Figure 8

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

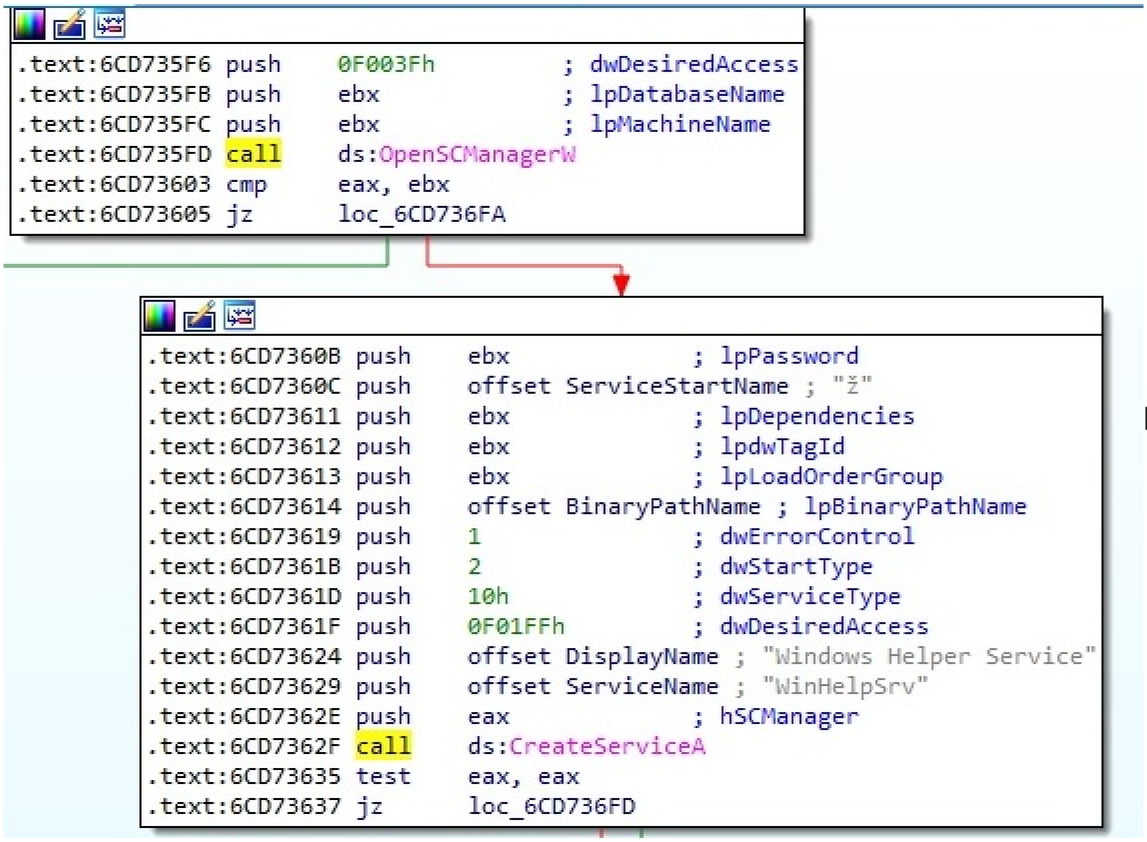

Figure 9

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:

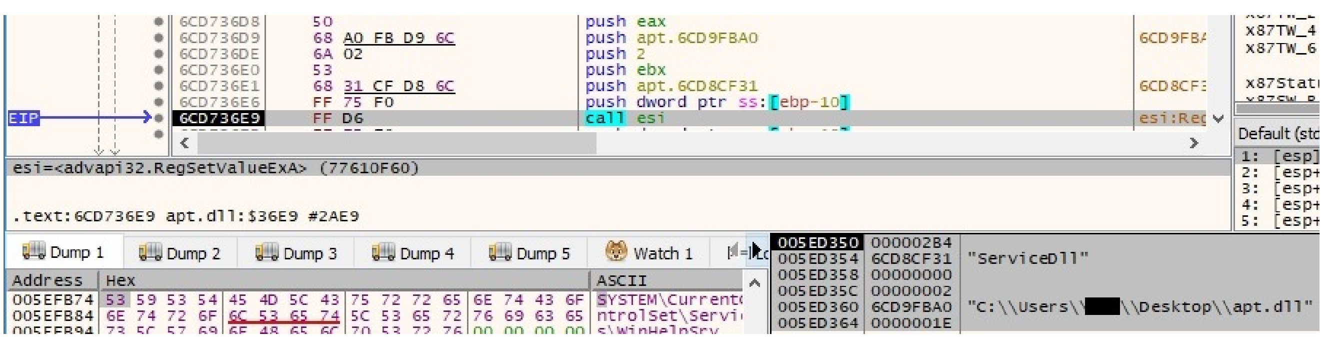

Figure 11

The confirmation that the operation was successful:

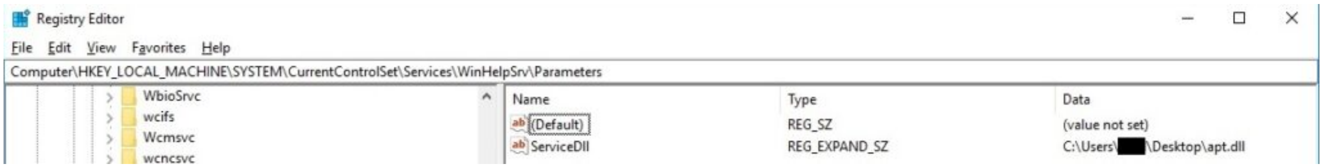

Figure 12

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

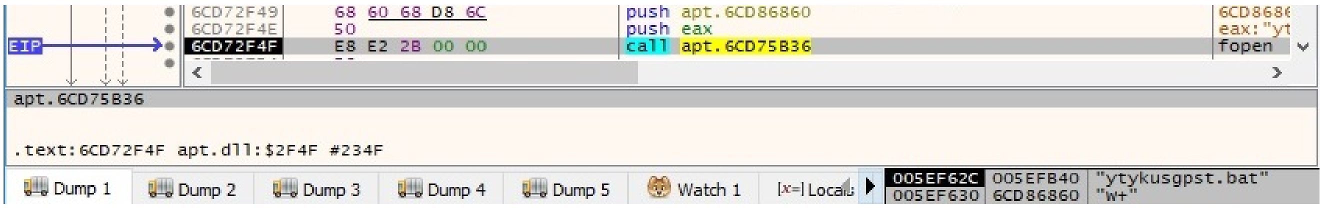

Figure 13

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):

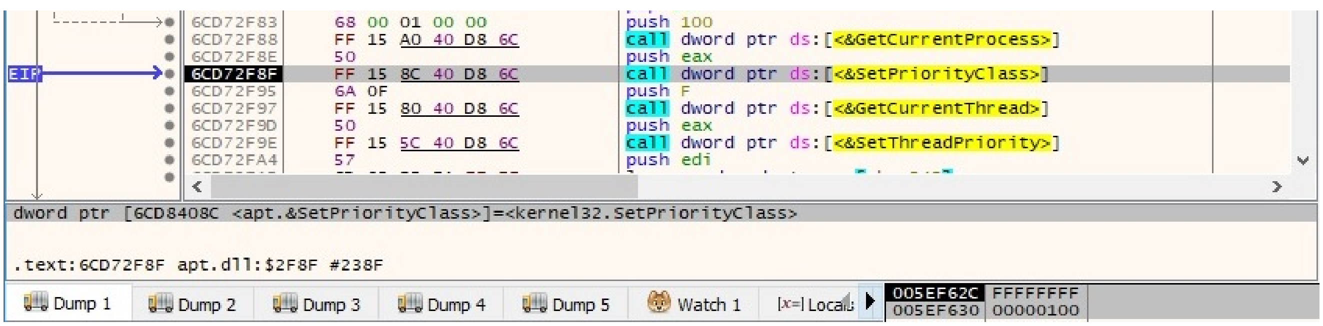

Figure 14

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

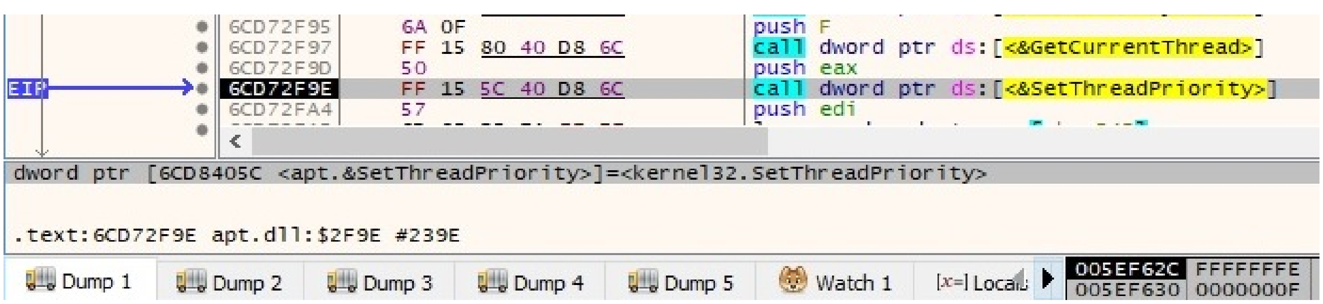

Figure 15

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:

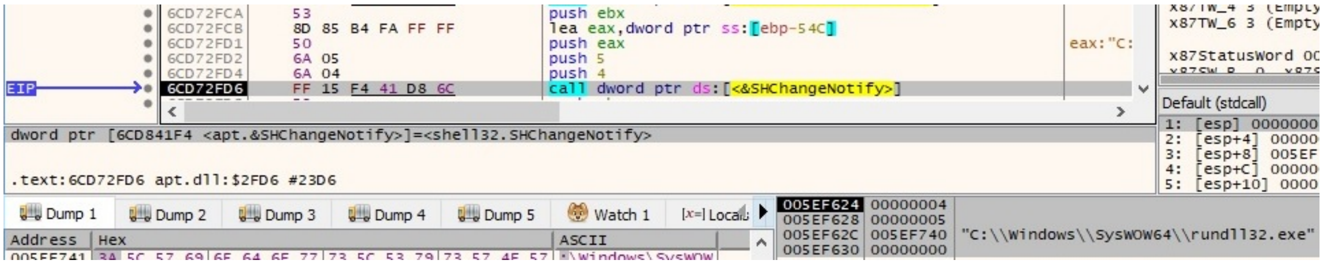

Figure 16

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

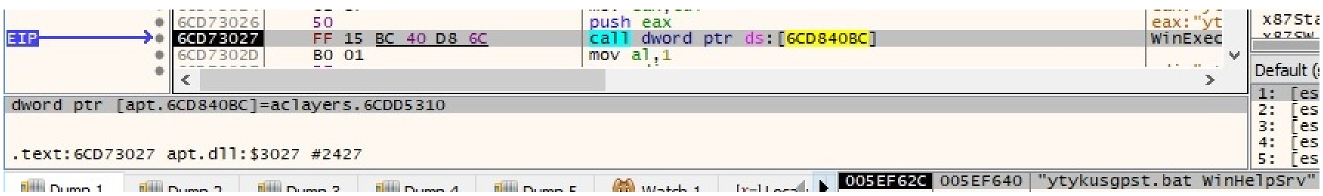

Figure 17

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:

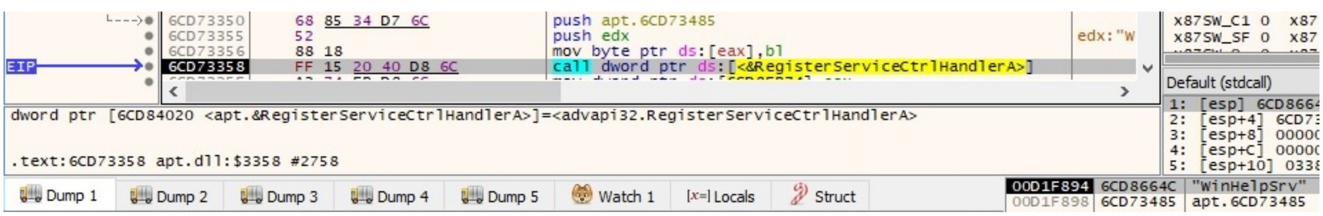

Figure 18

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):

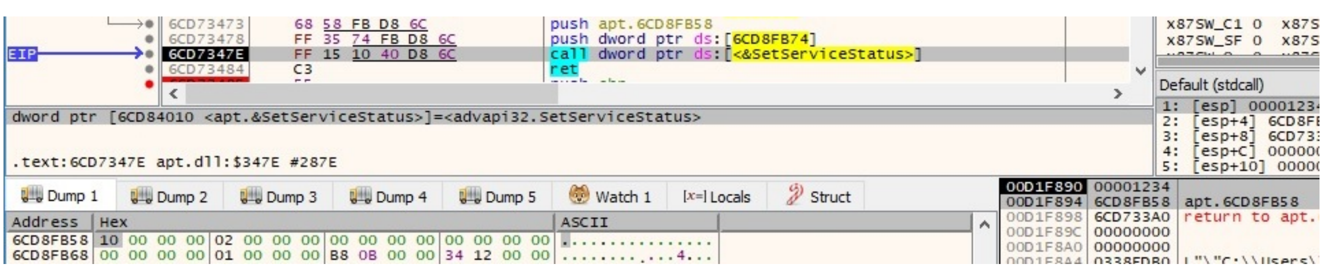

Figure 19

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

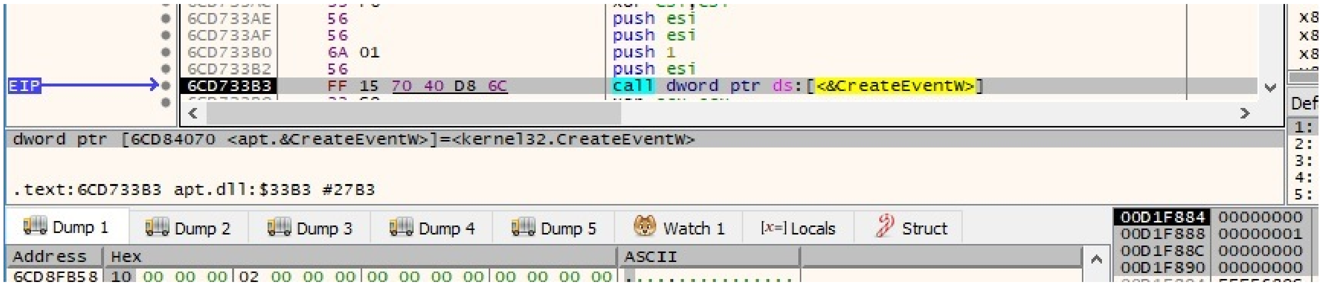

Figure 20

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):

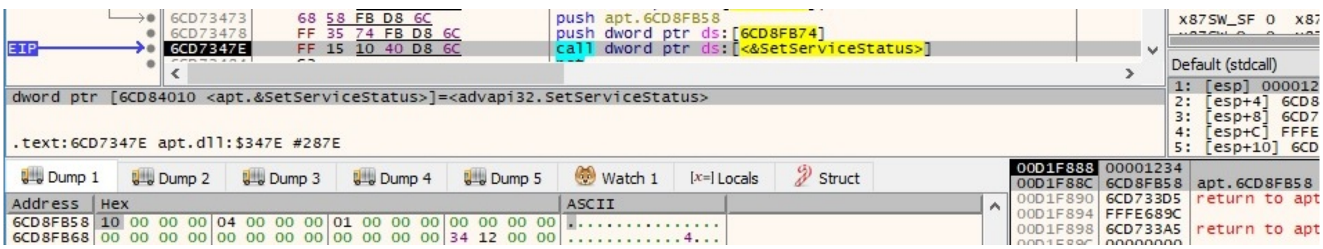

Figure 21

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:

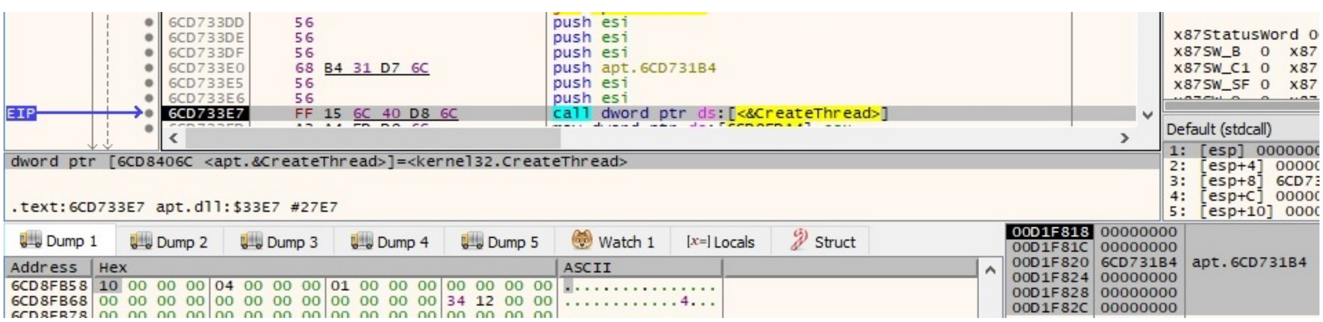

Figure 22

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):

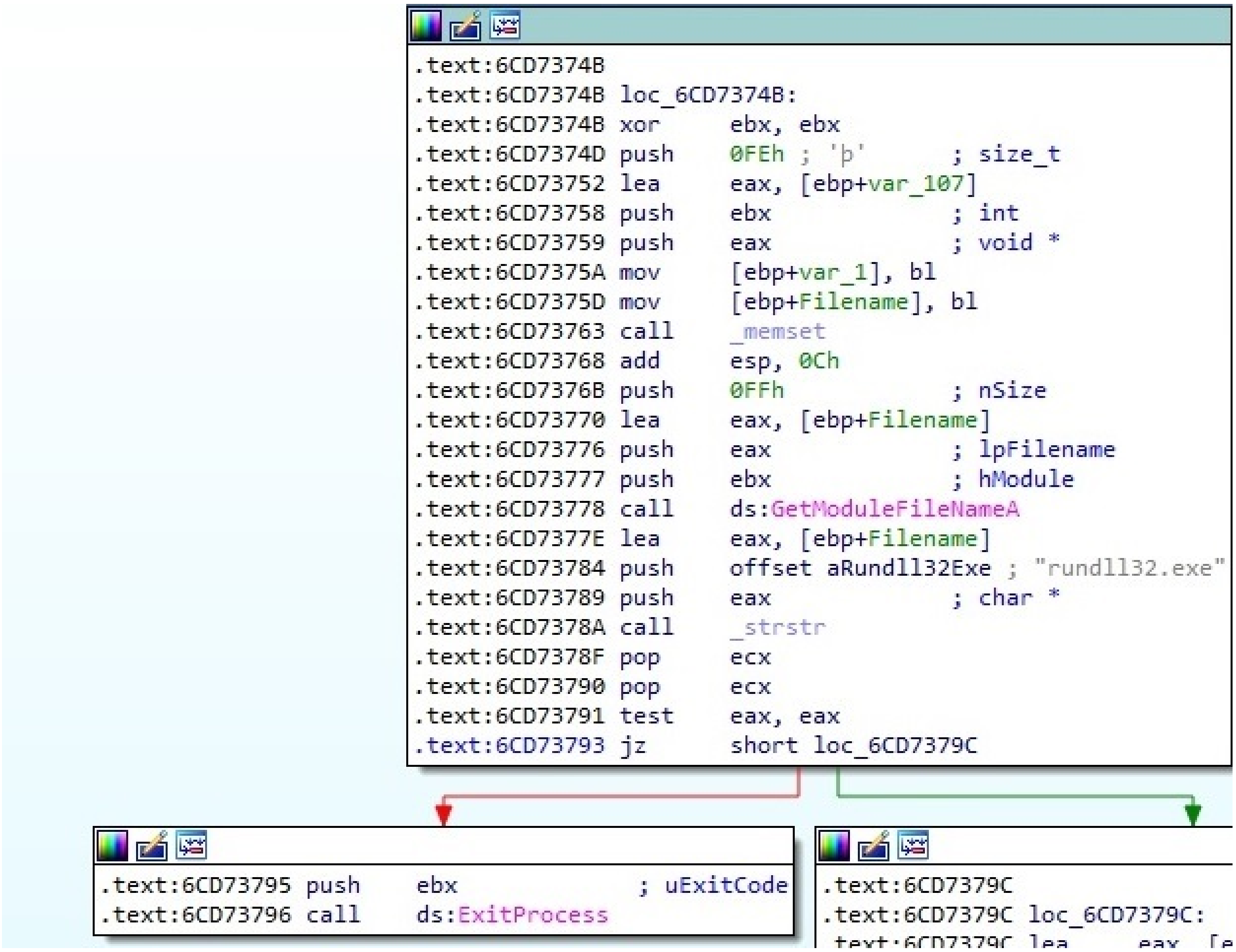

Figure 23

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:

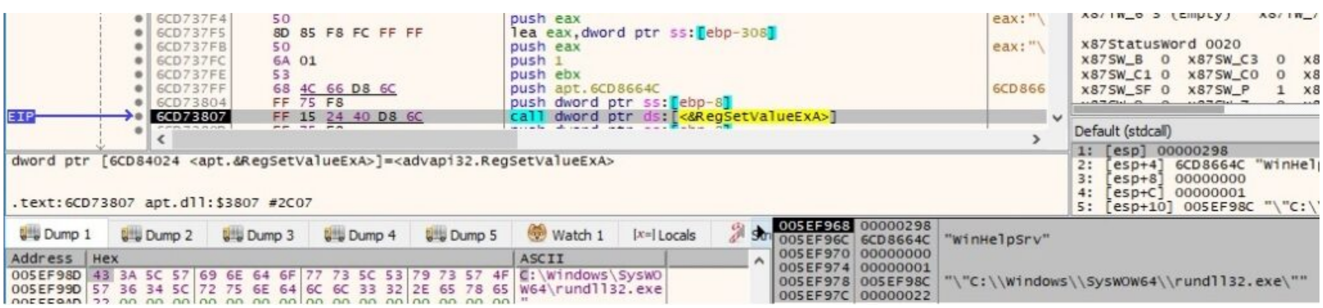

Figure 24

The confirmation that the persistence was successfully established:

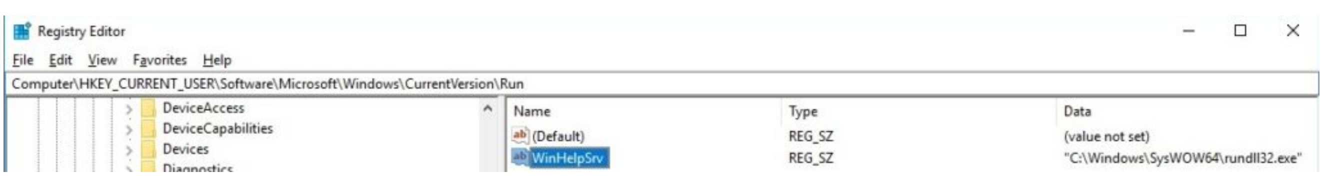

Figure 25

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:

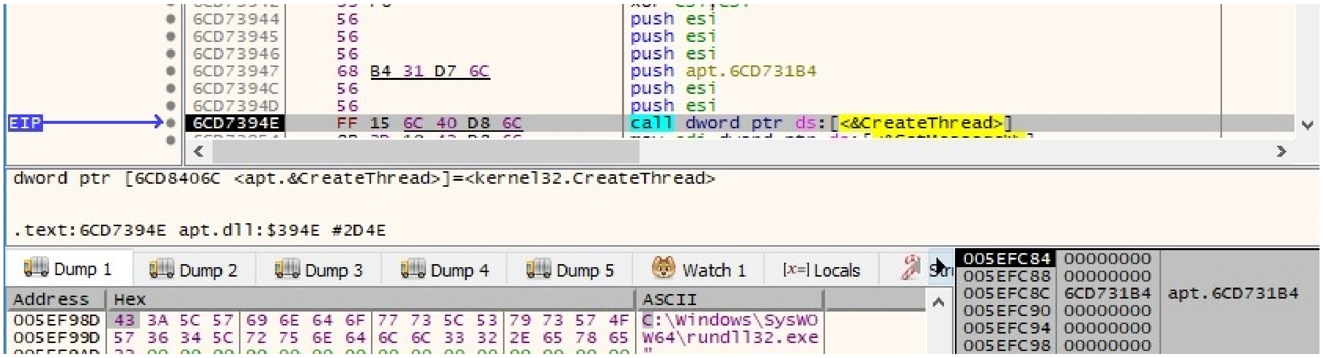

Figure 26

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:

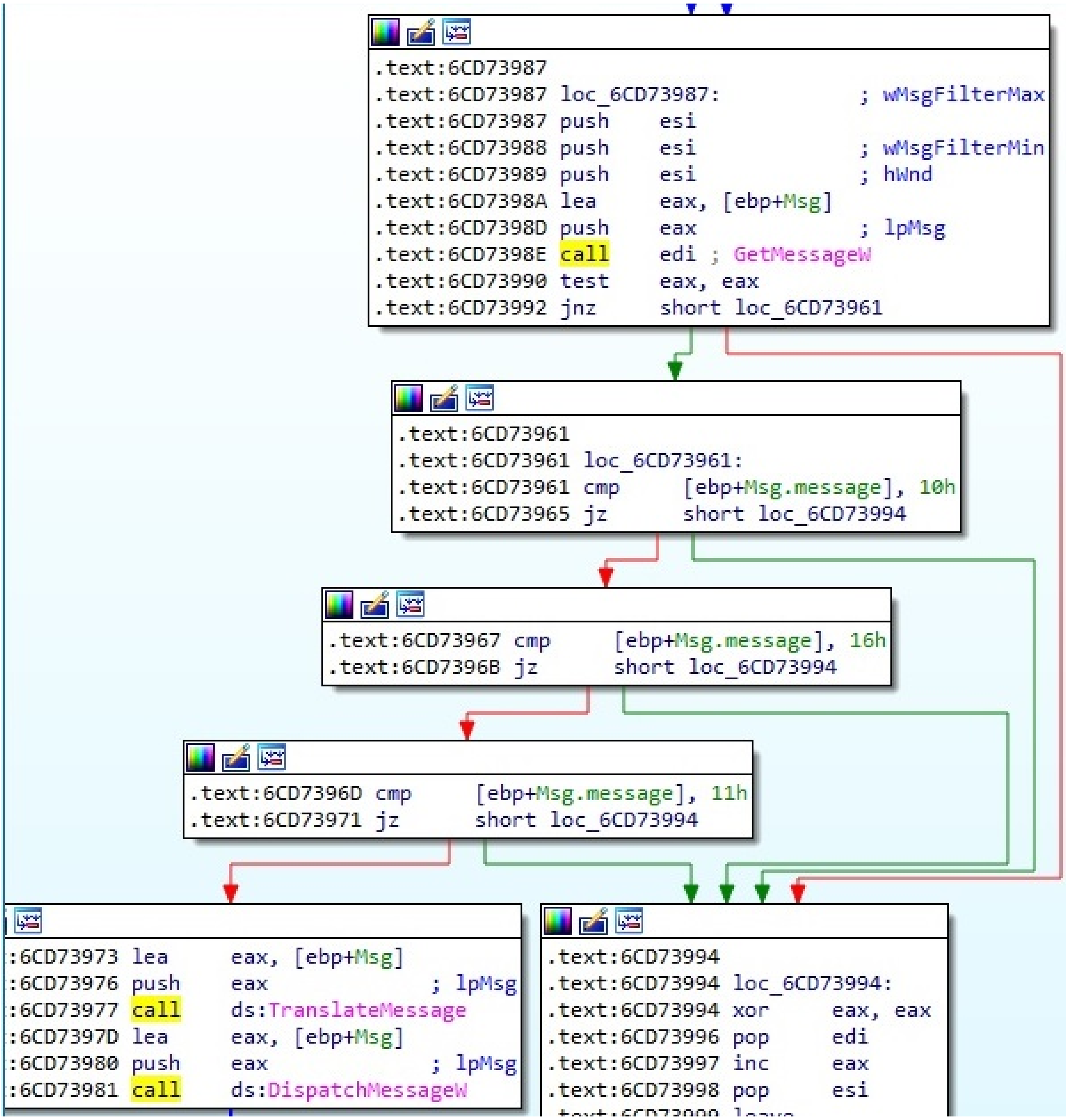

Figure 27
Thread activity – StartAddress address

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:

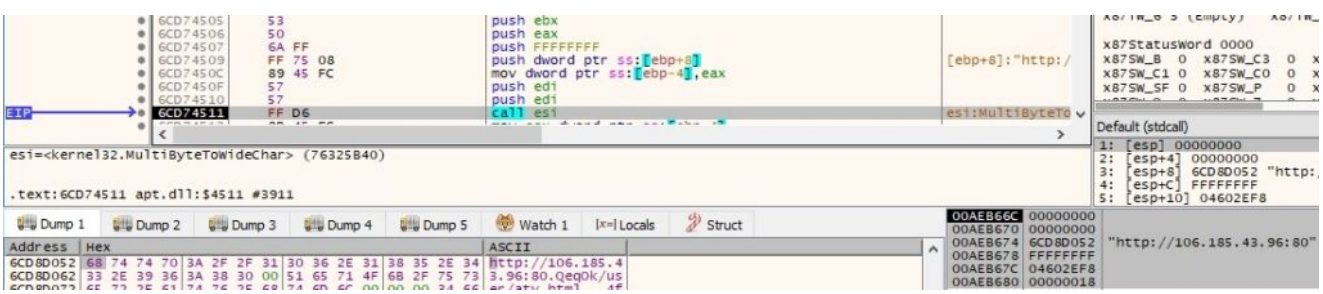

Figure 28

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

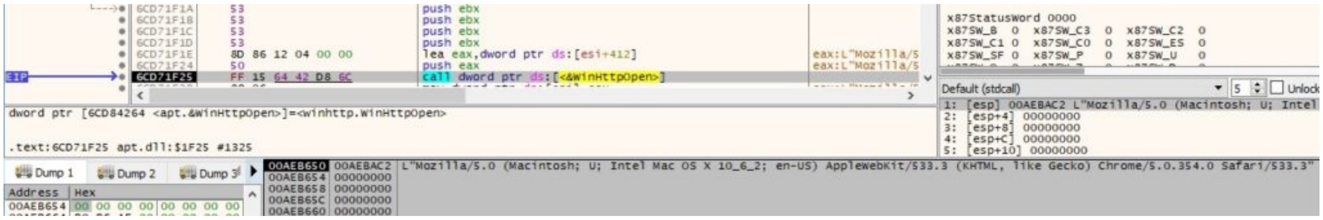

Figure 29

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):

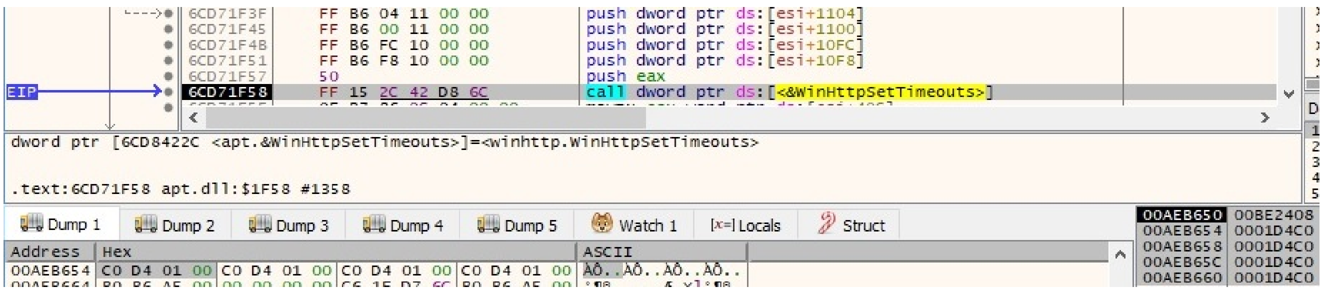

Figure 30

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.

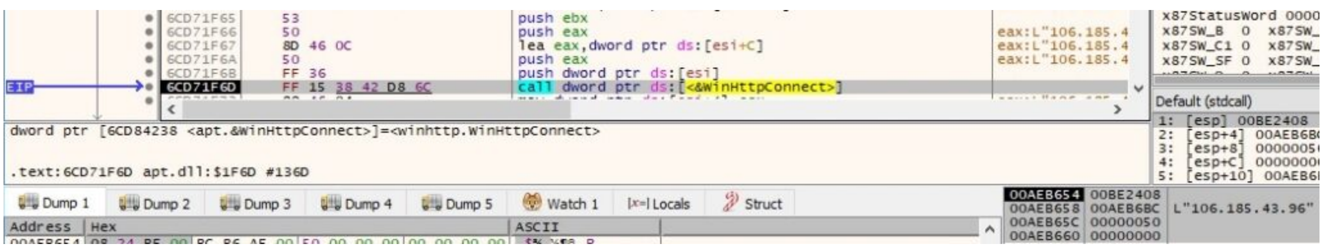

Figure 31

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:

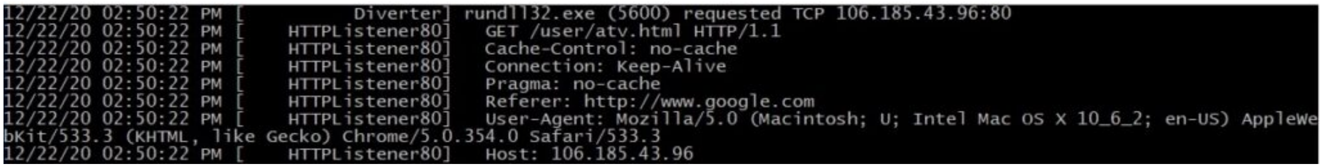

Figure 33

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

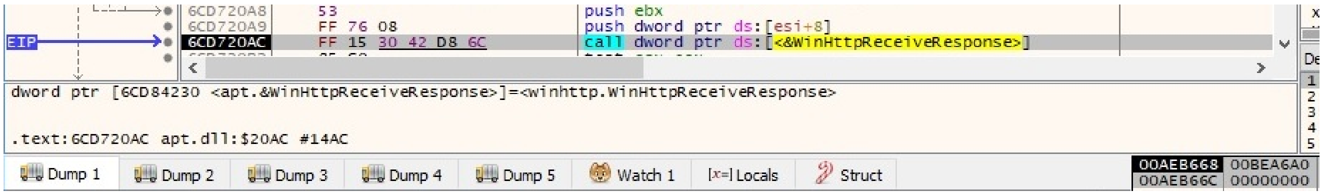

Figure 34

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:

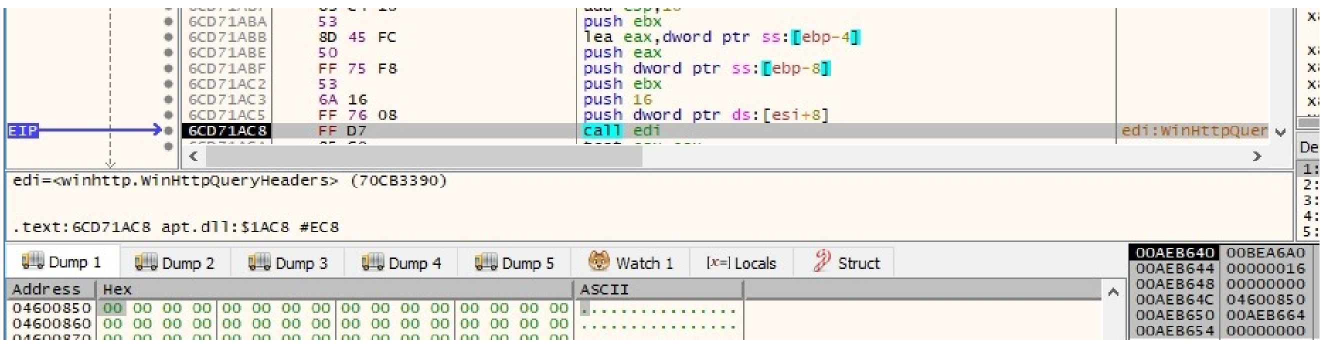

Figure 35

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):

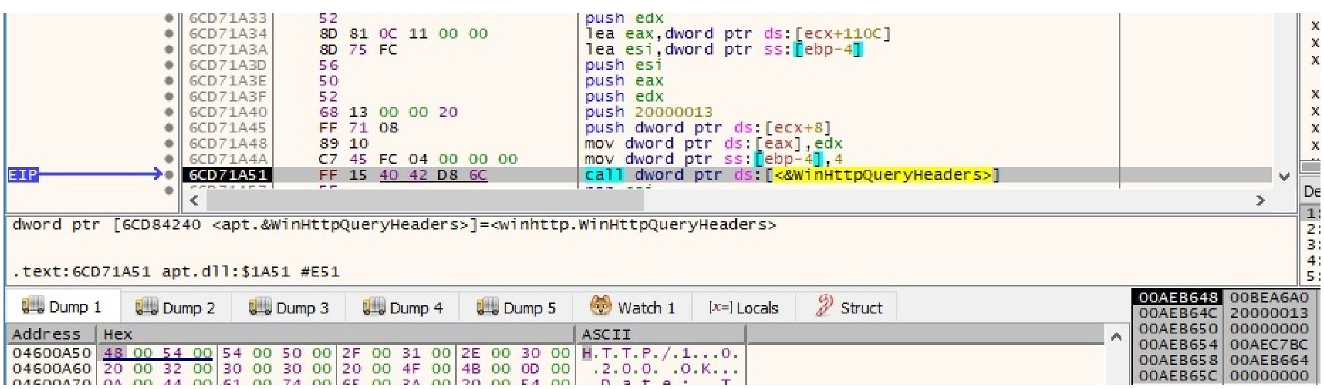

Figure 36

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

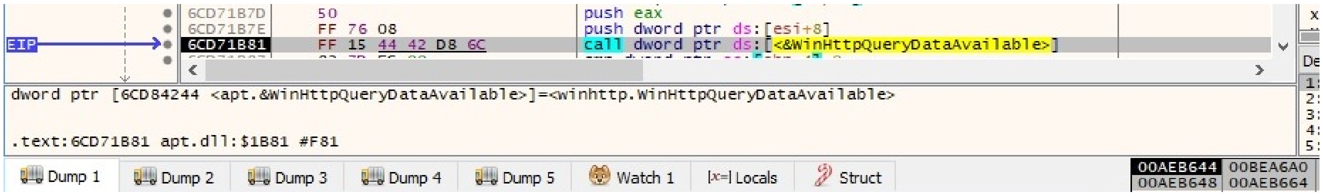

Figure 37

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

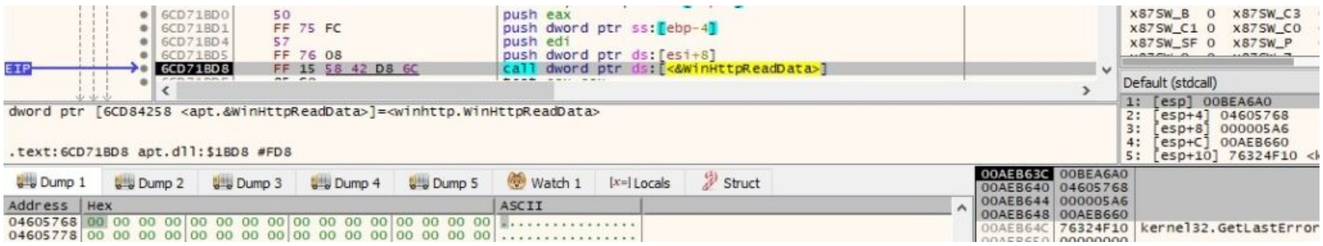

Figure 38

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):

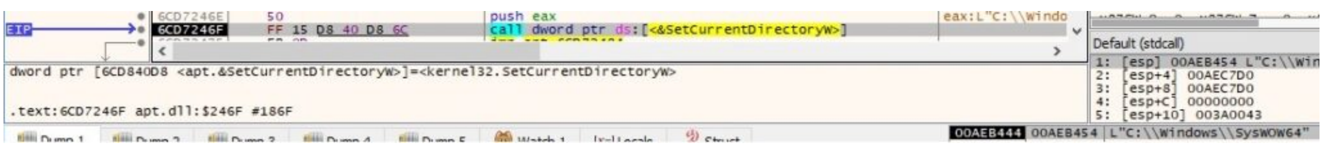

Figure 39

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

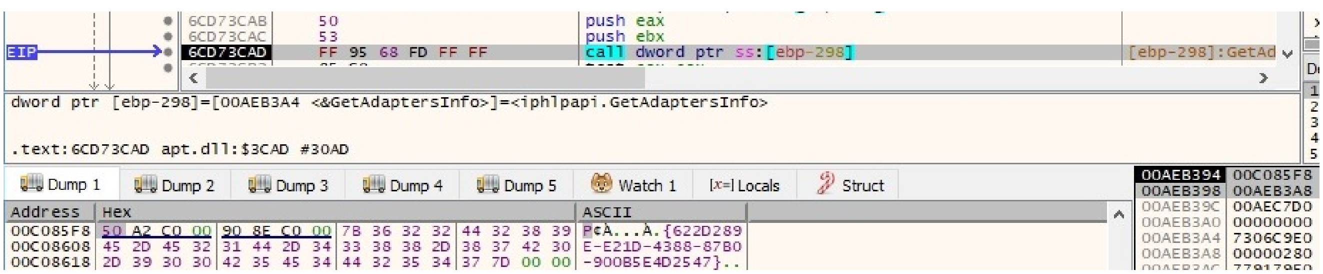

Figure 40

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

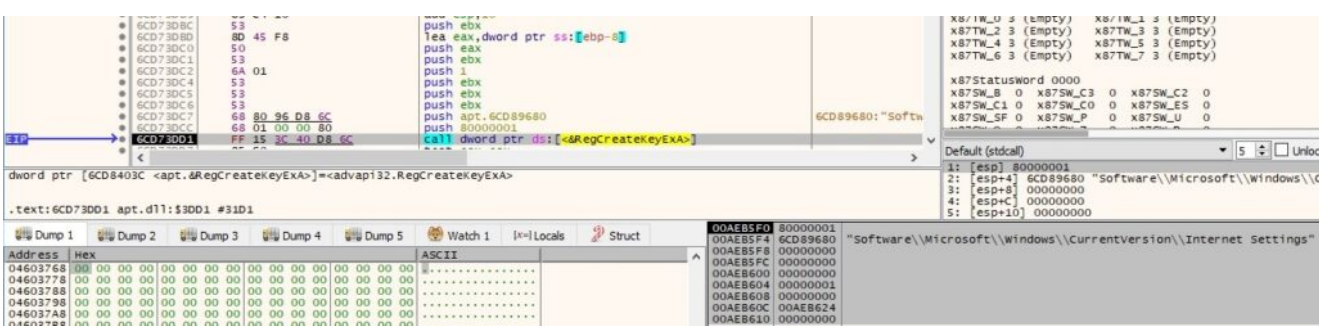

Figure 41

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

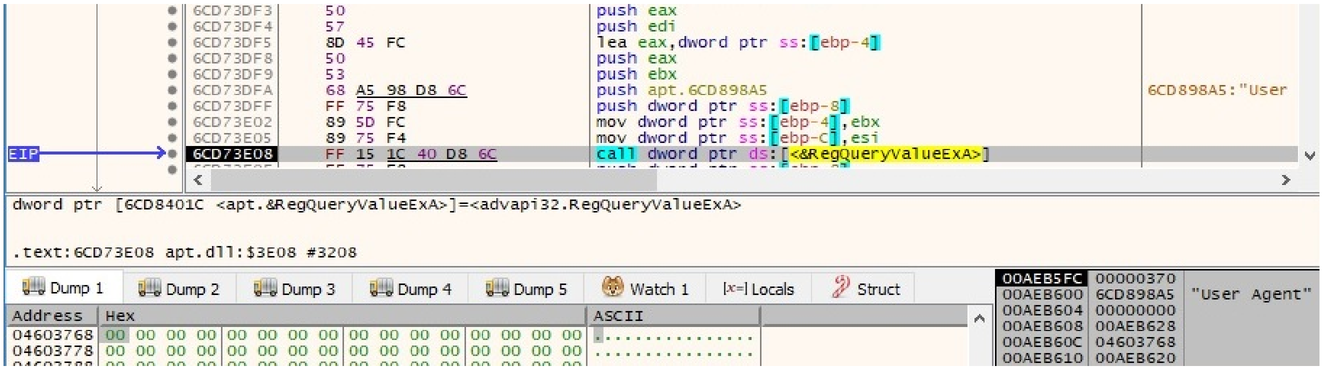

Figure 42

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

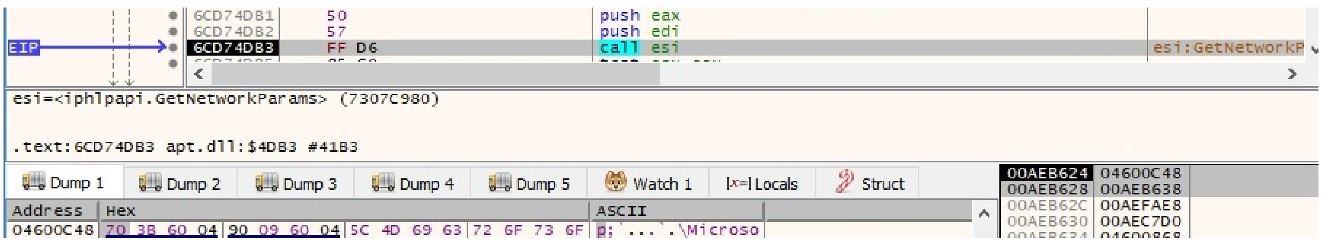

Figure 43

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:

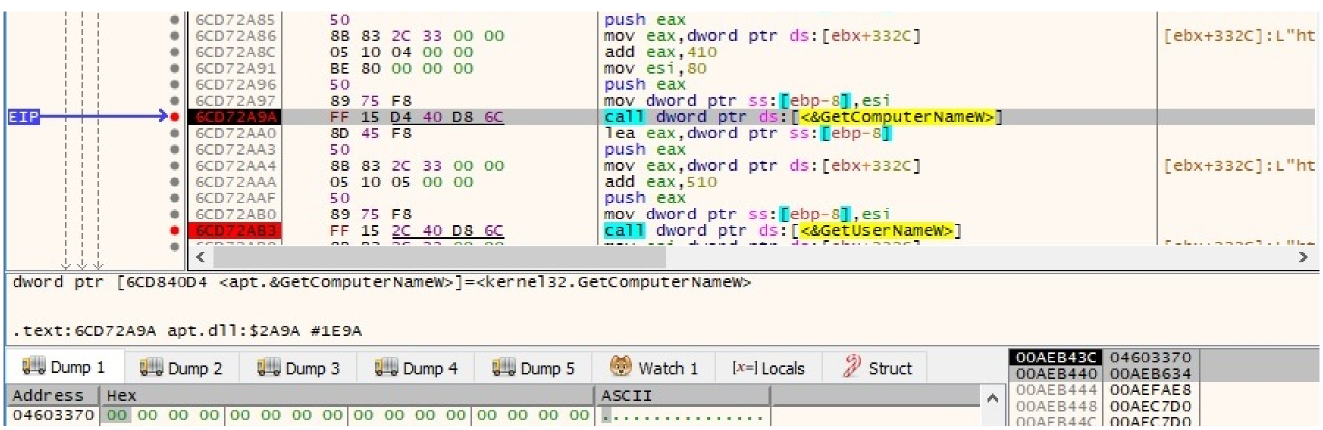

Figure 44

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:

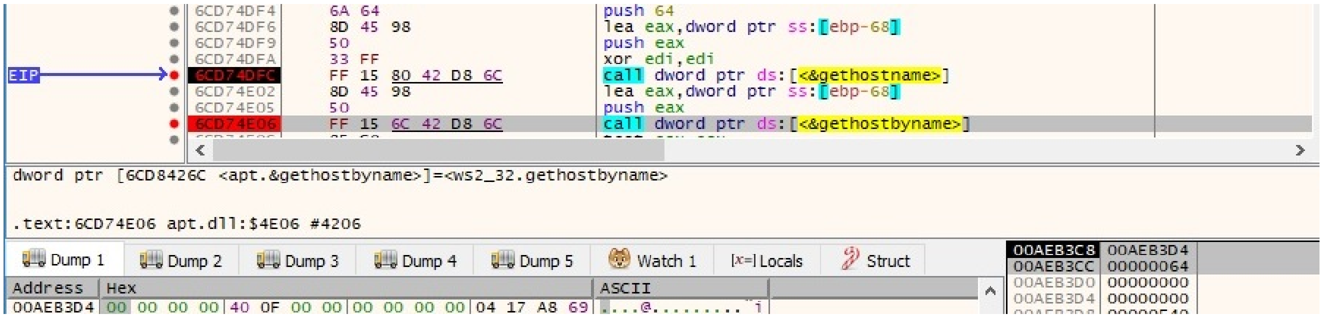

Figure 45

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:

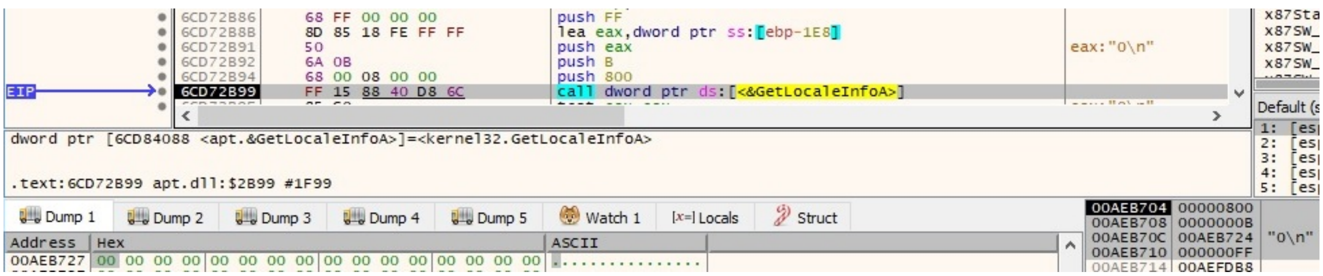

Figure 46

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?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:

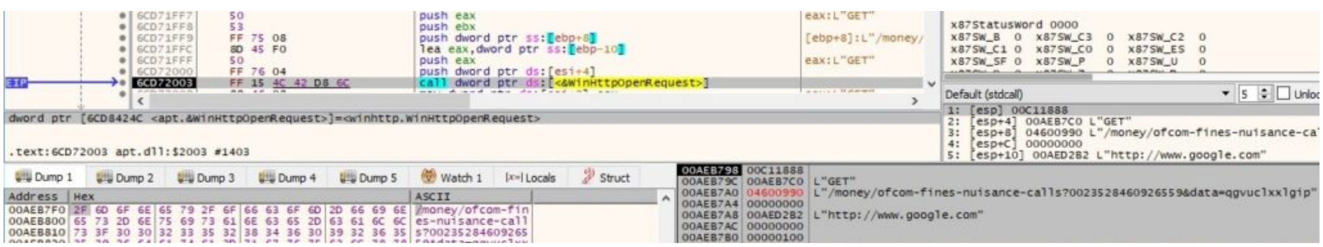

Figure 50

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:

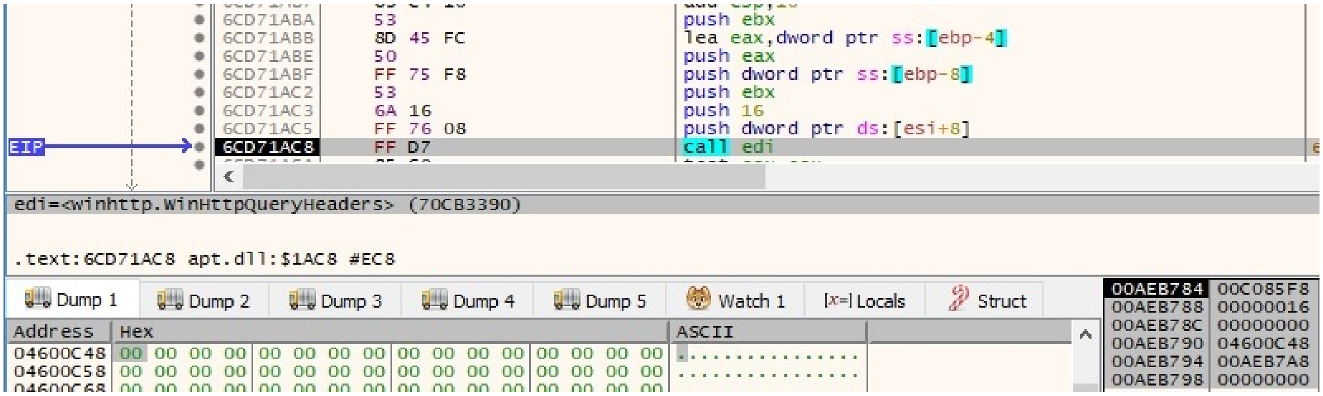

Figure 51

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:

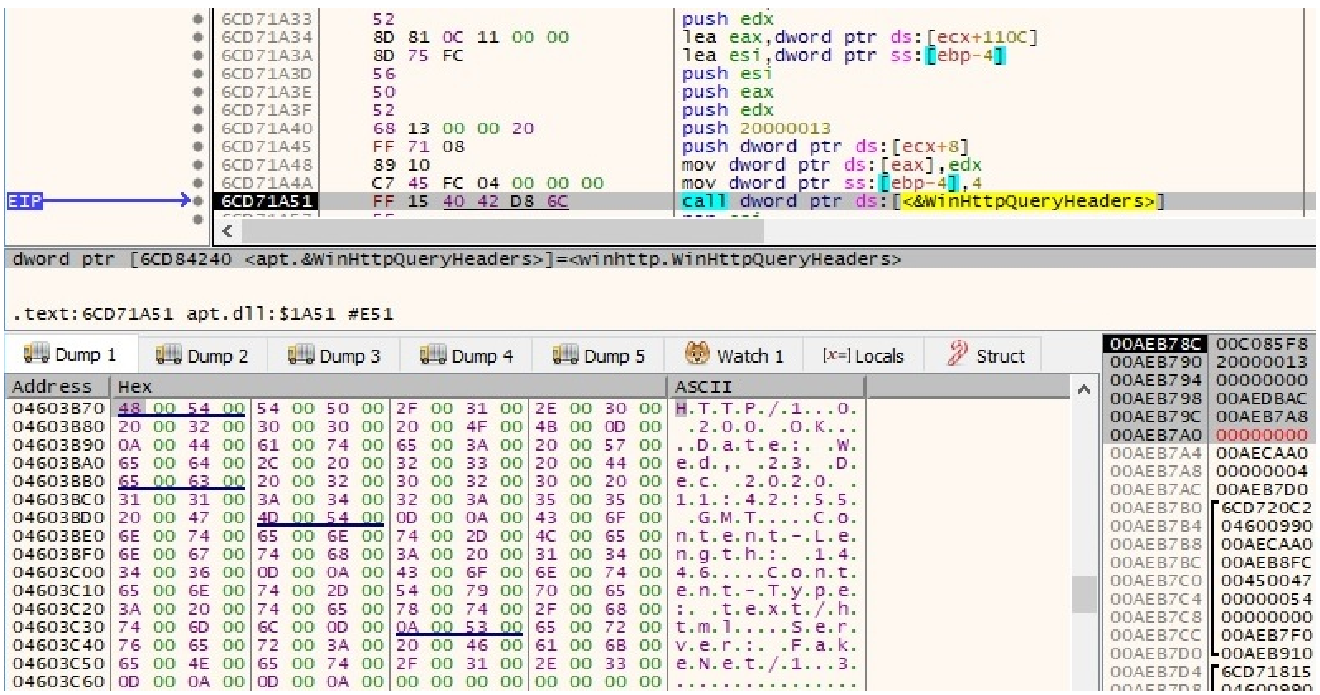

Figure 52

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

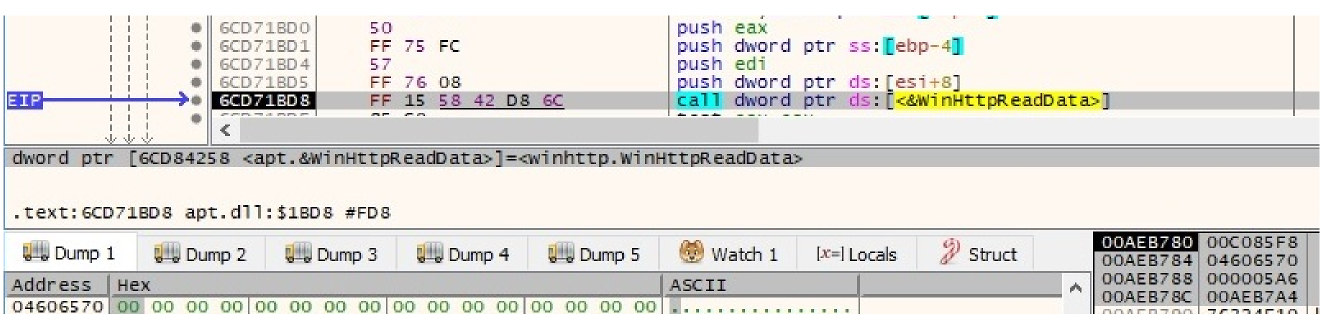

Figure 53

The 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):

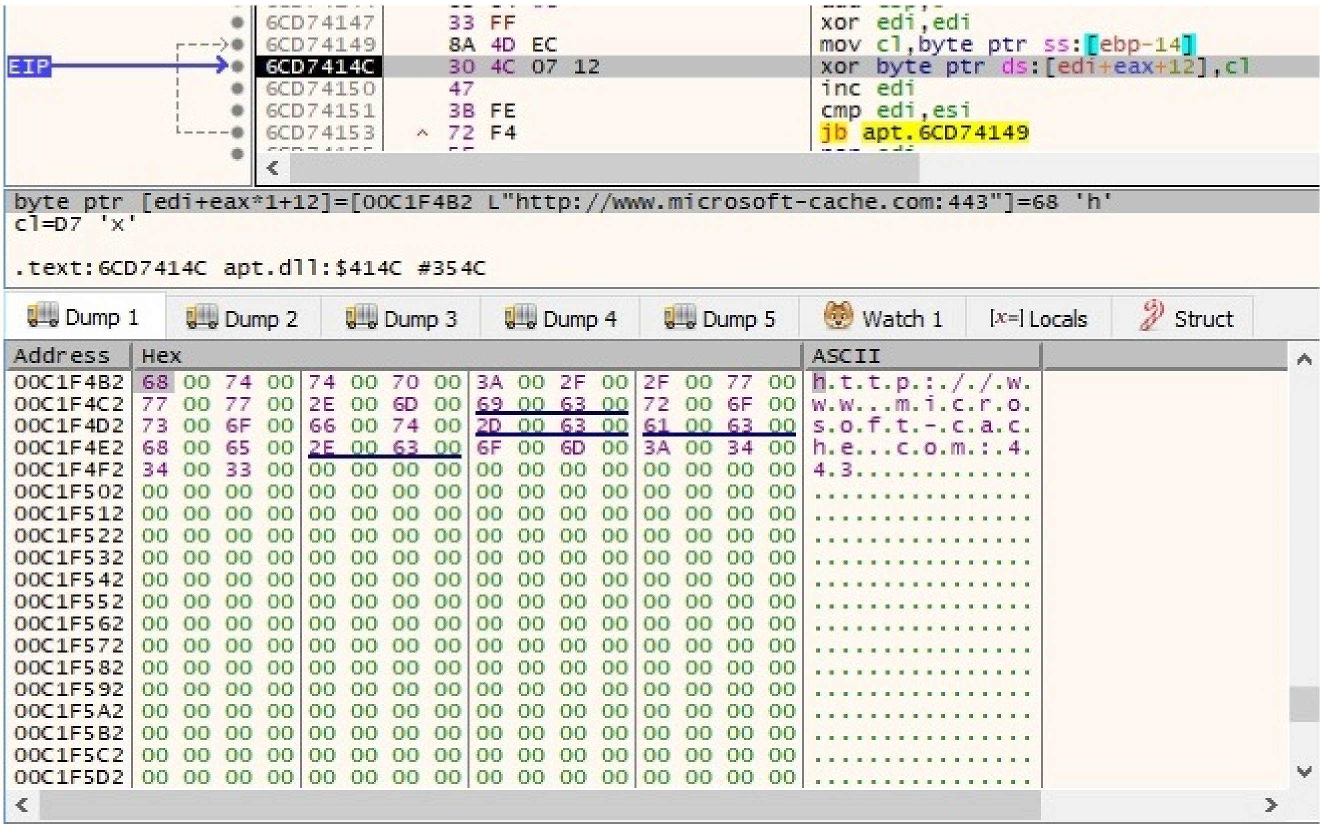

Figure 54

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

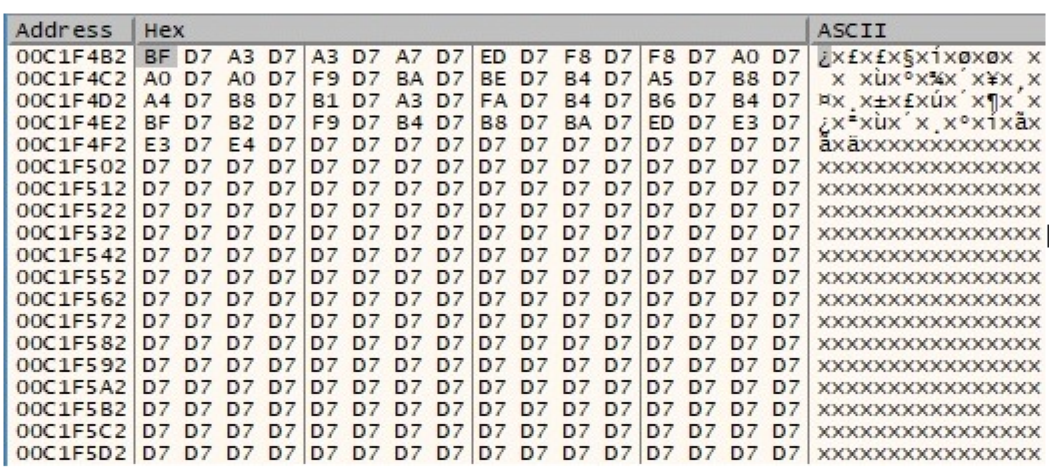

Figure 55

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:

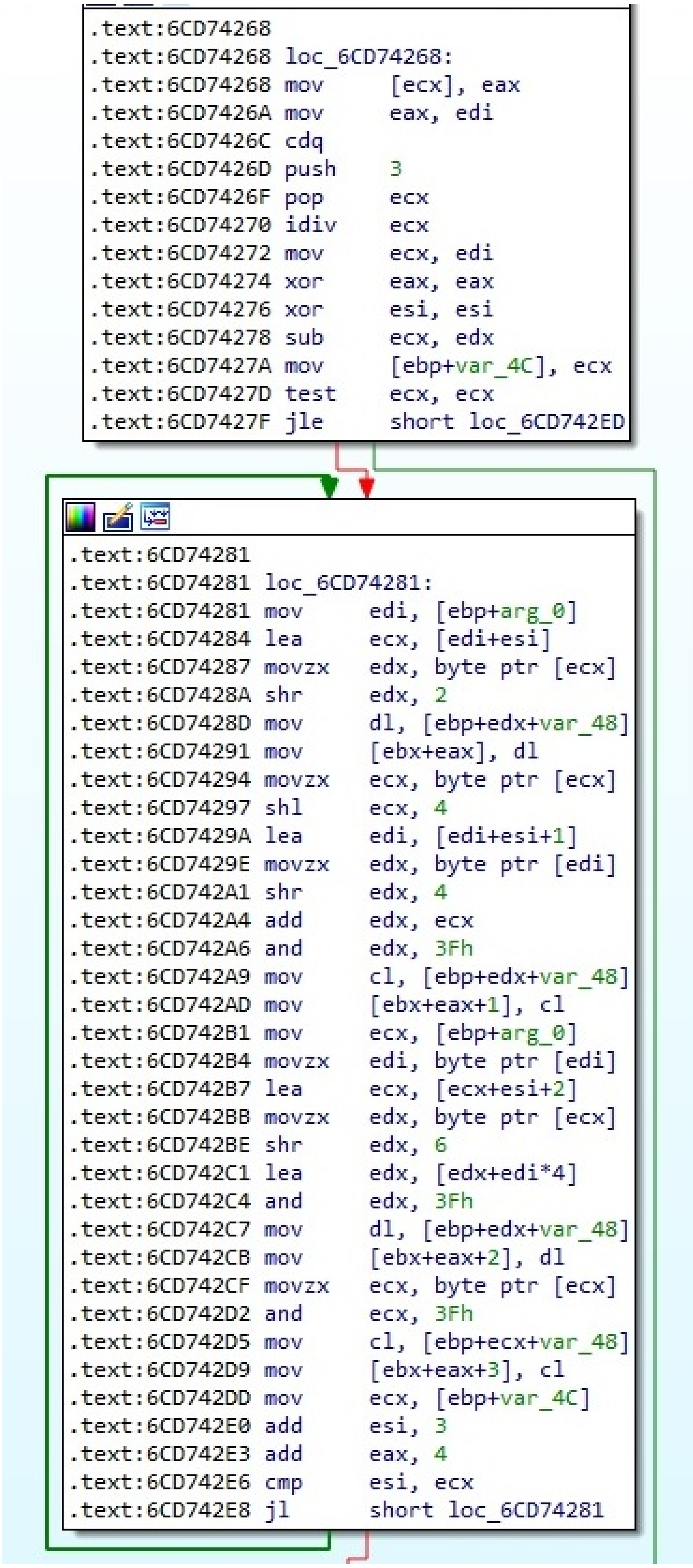

Figure 56

The encrypted buffer is encoded with the Base64 algorithm:

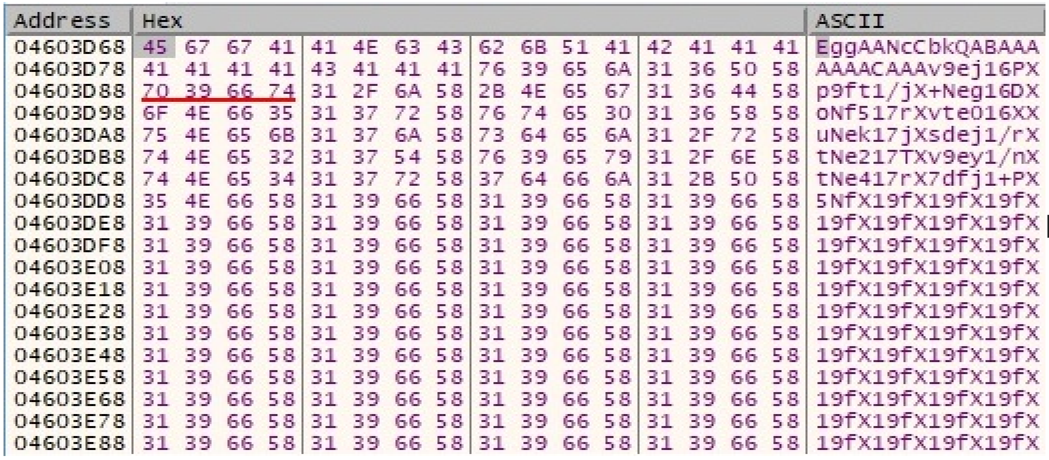

Figure 57

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:

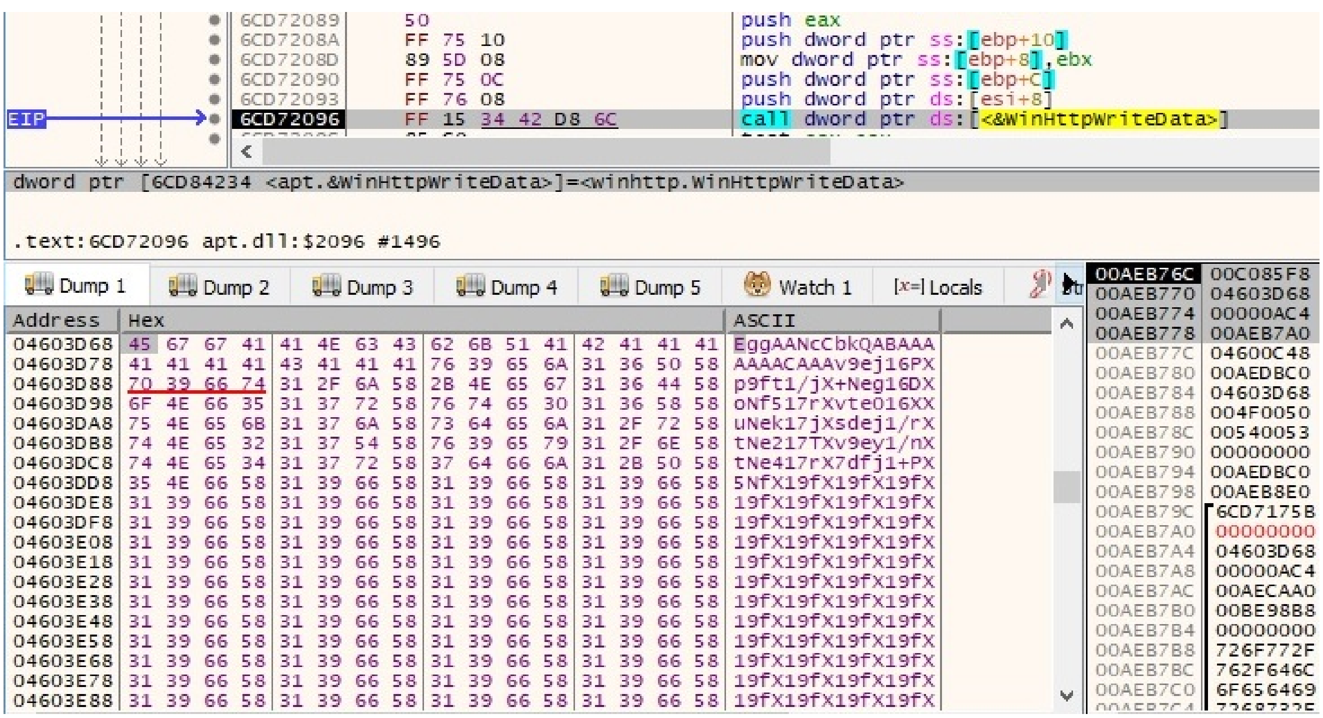

Figure 59

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:

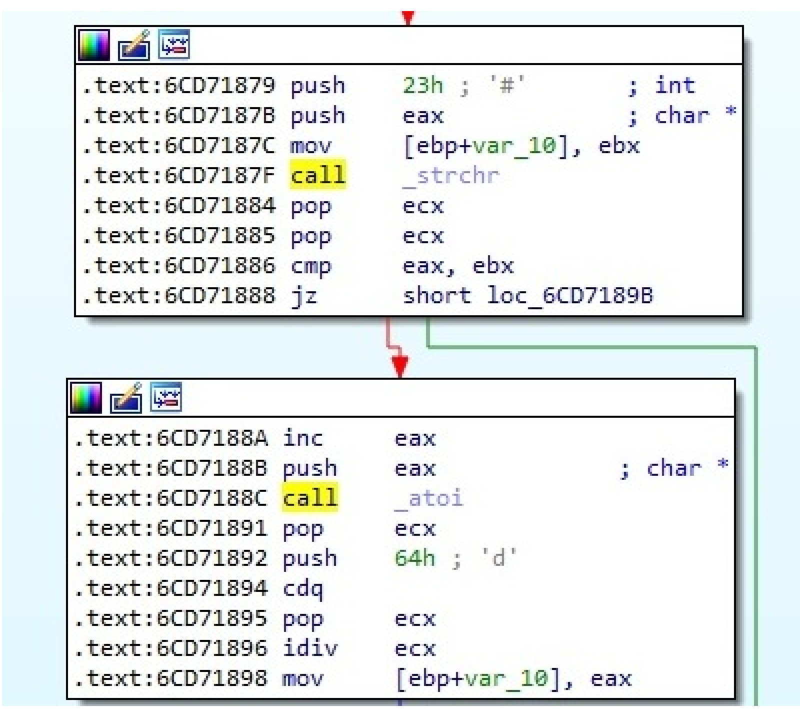

Figure 61

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):

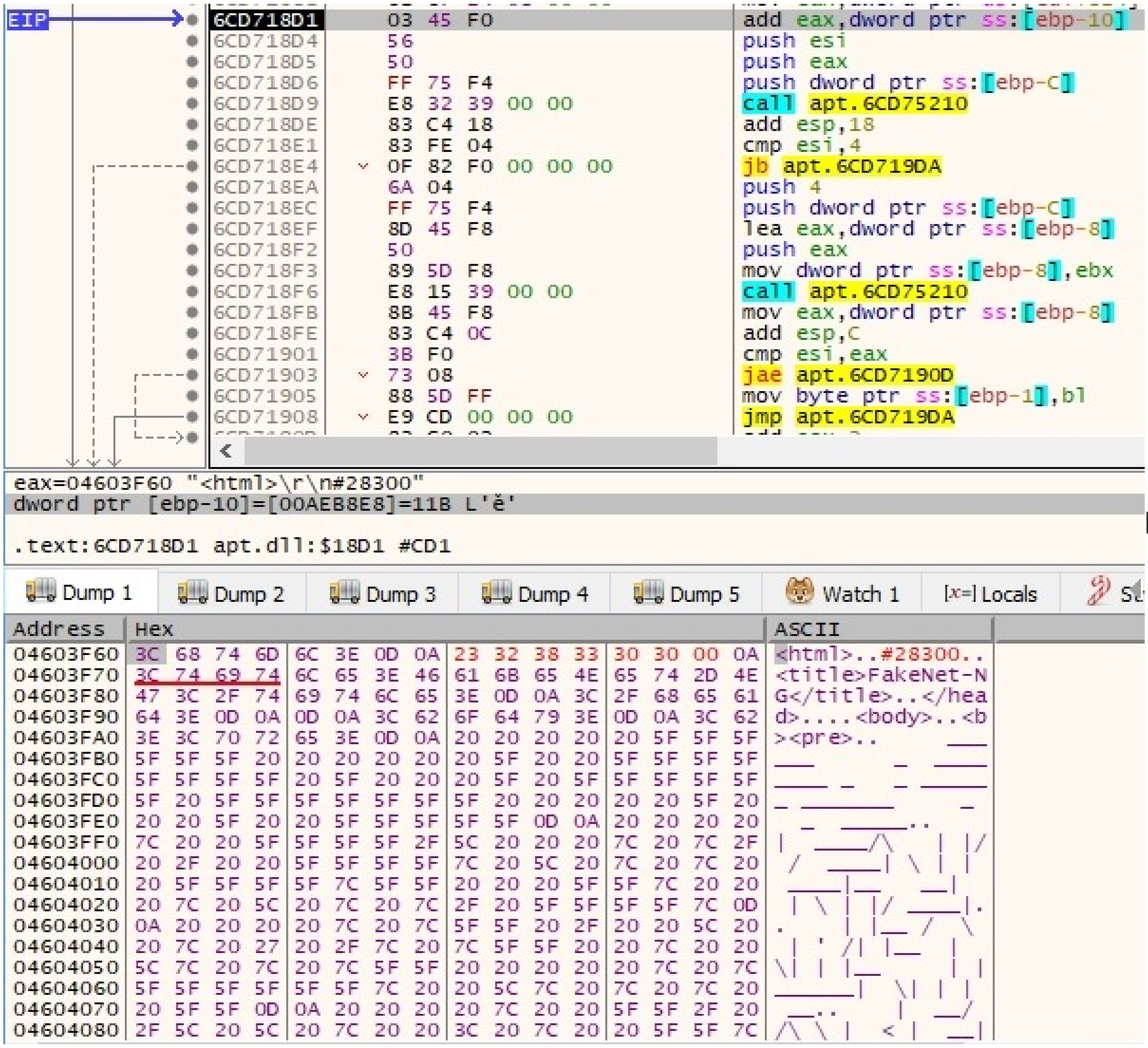

Figure 62

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

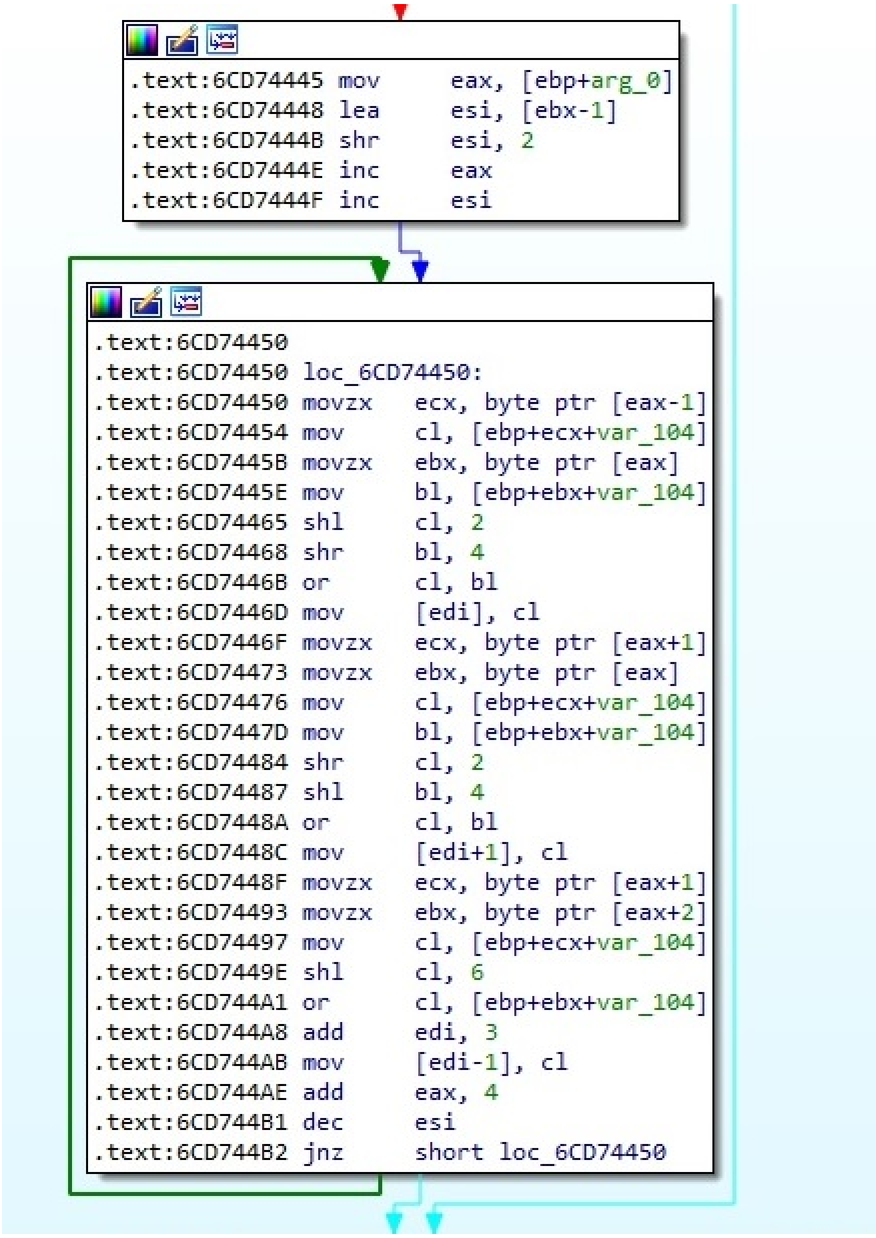

Figure 64

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:

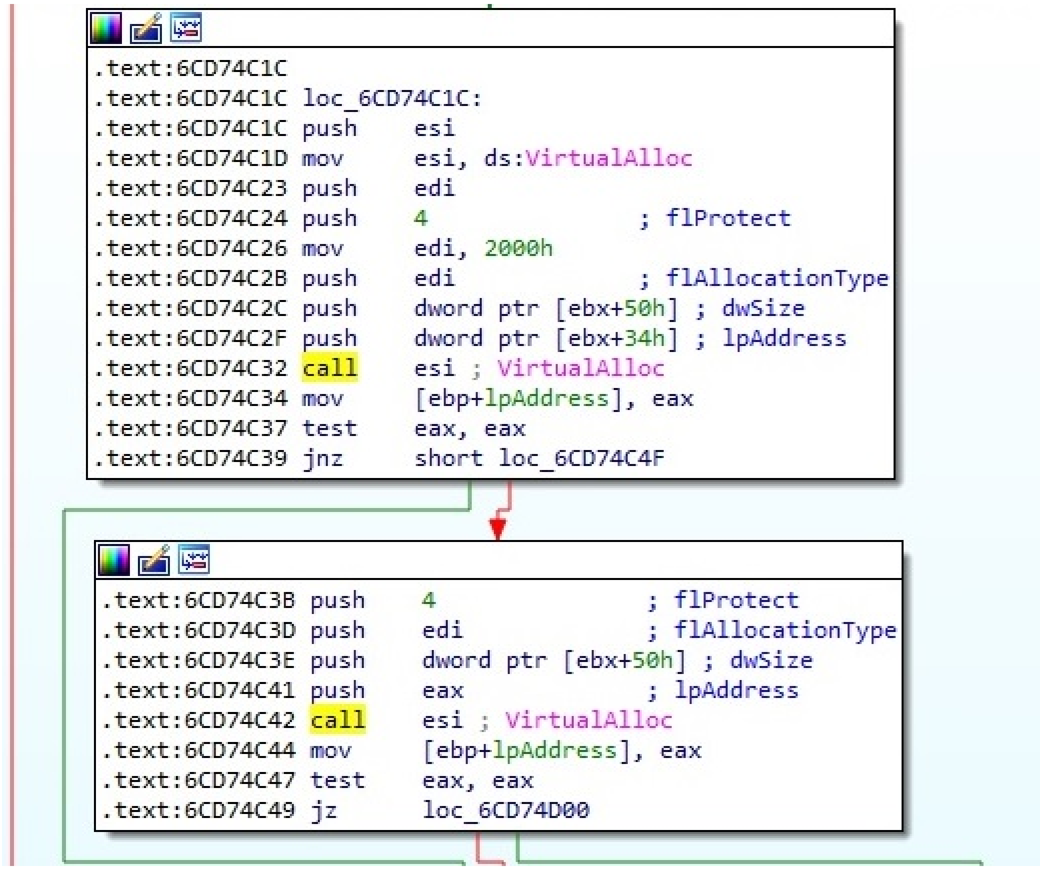

Figure 65

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:

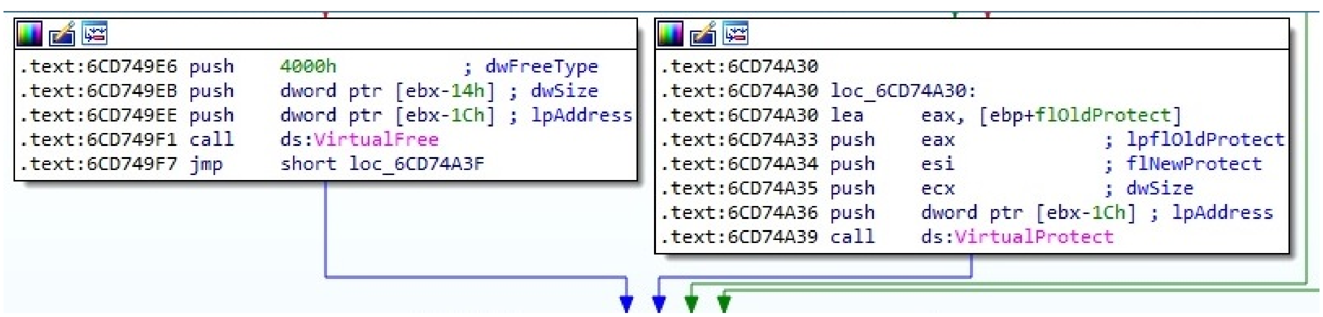

Figure 66

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:

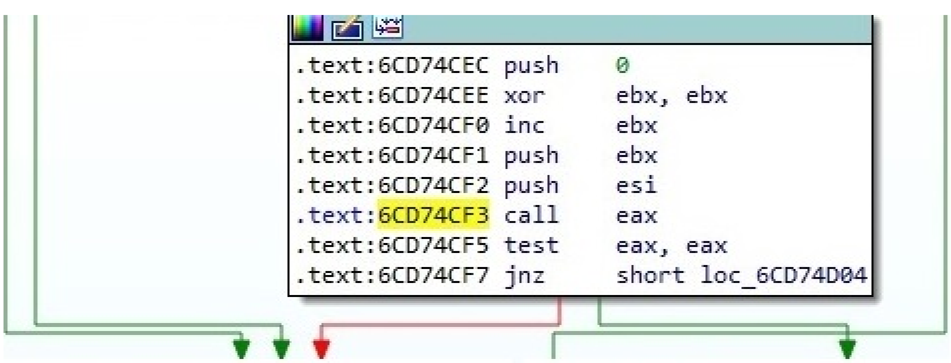

Figure 67

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) 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
}
```