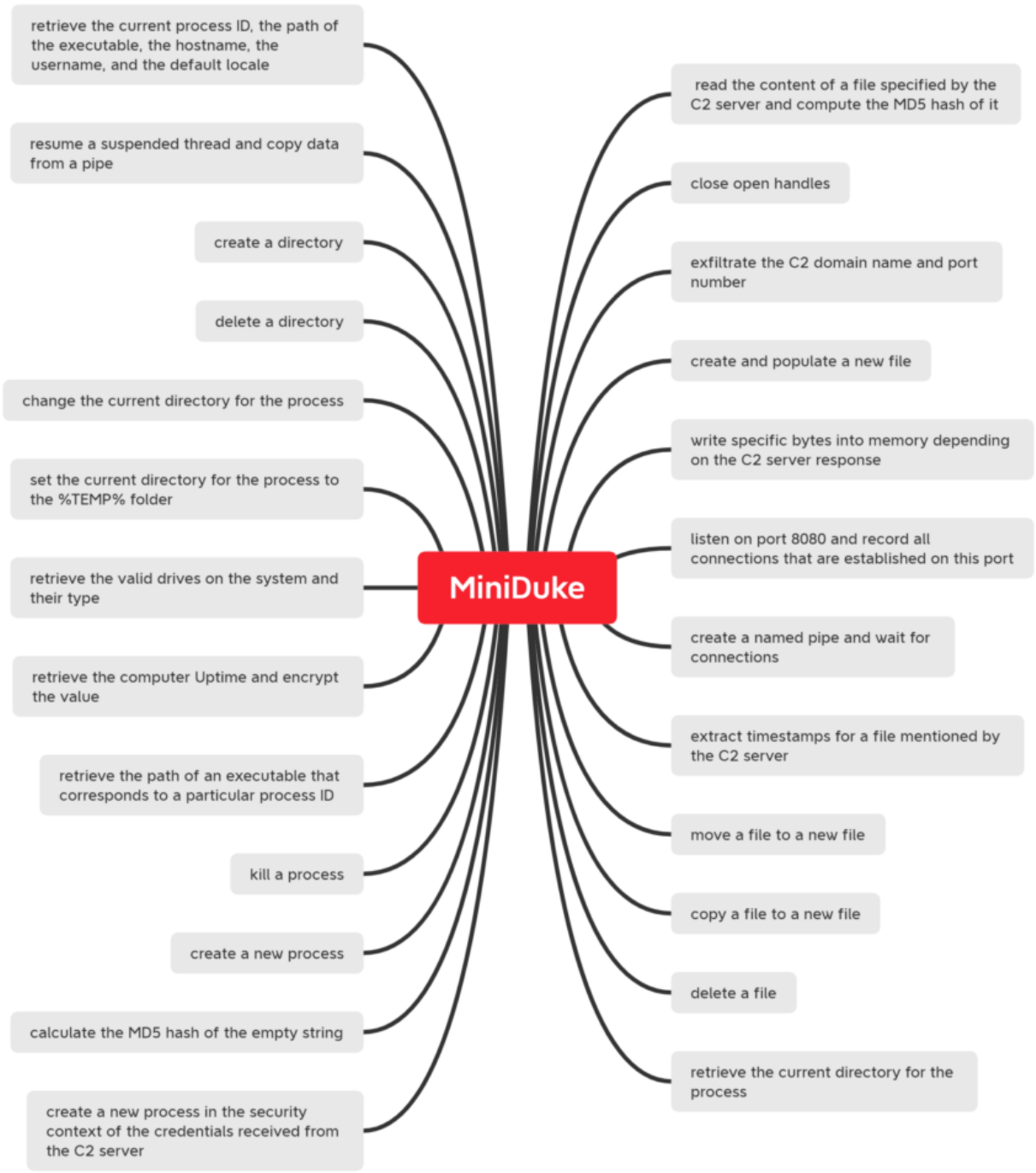
How to defeat the Russian Dukes: A step-by-step analysis of MiniDuke used by APT29/Cozy Bear

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:

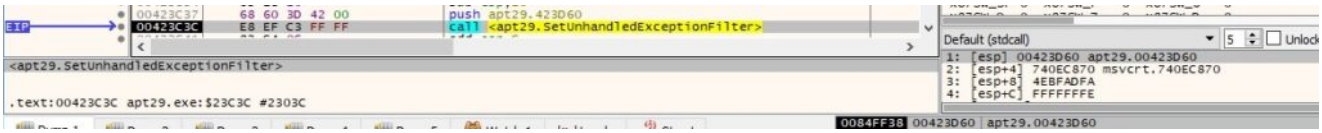


Figure 1

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):

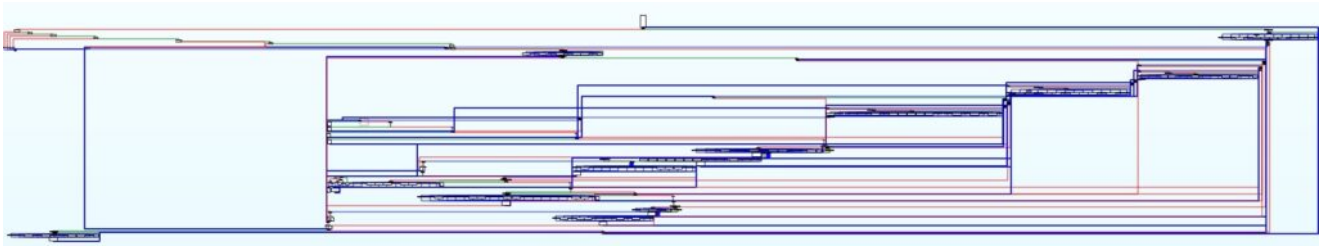


Figure 4

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:

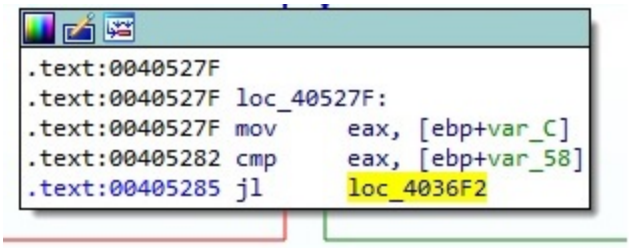


Figure 5

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

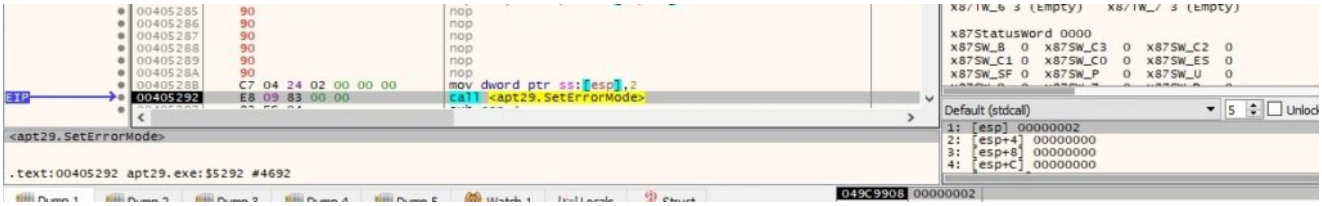


Figure 6

The kernel32.dll, advapi32.dll and winninet.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:

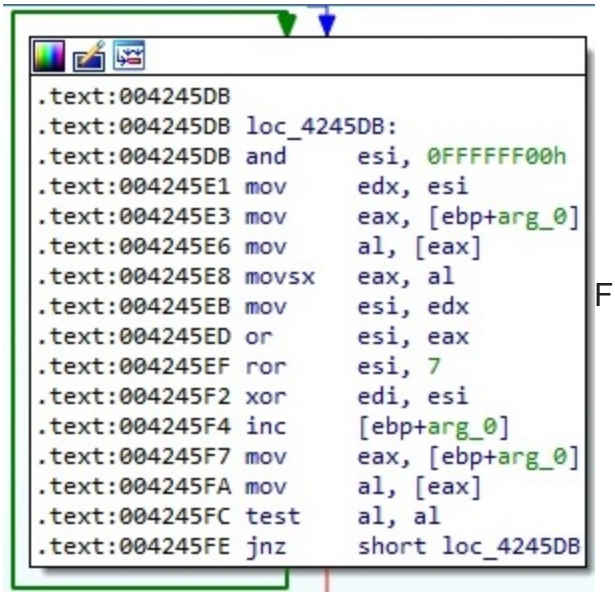


Figure 9

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**):

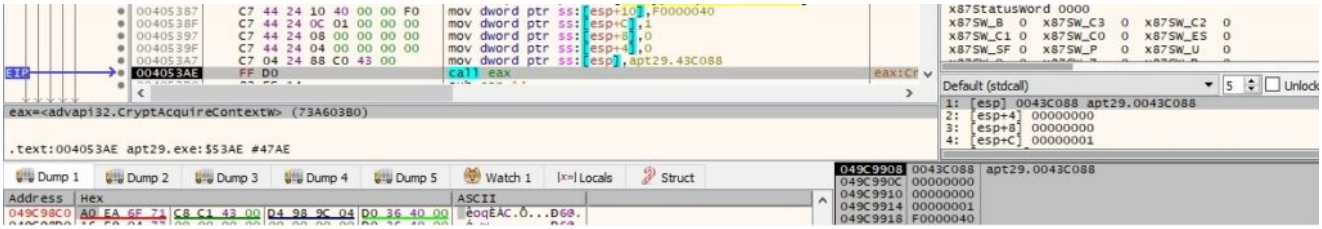


Figure 10

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

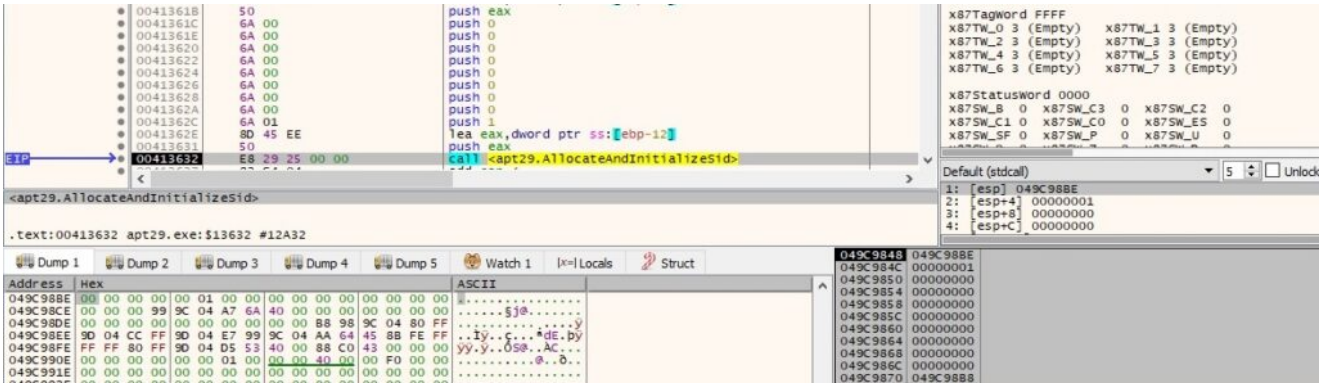


Figure 11

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

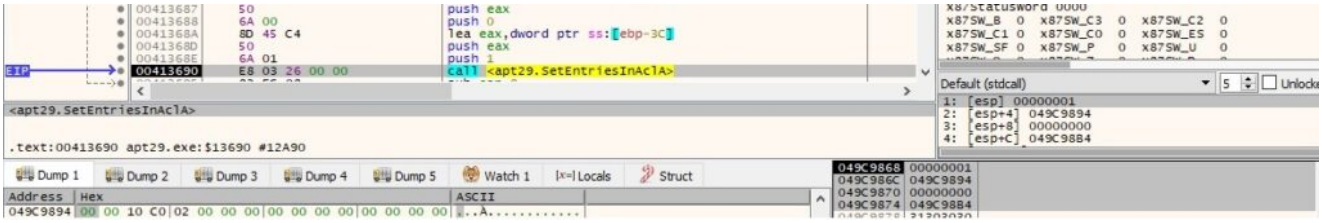


Figure 12

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

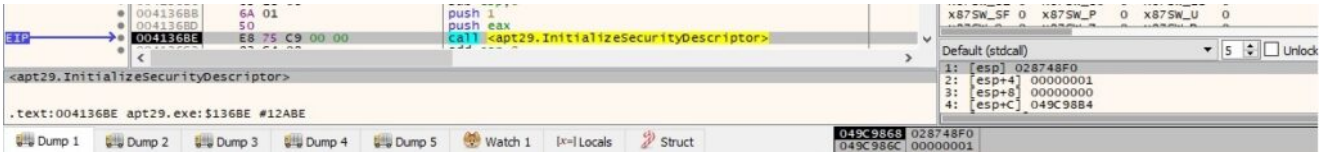


Figure 13

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

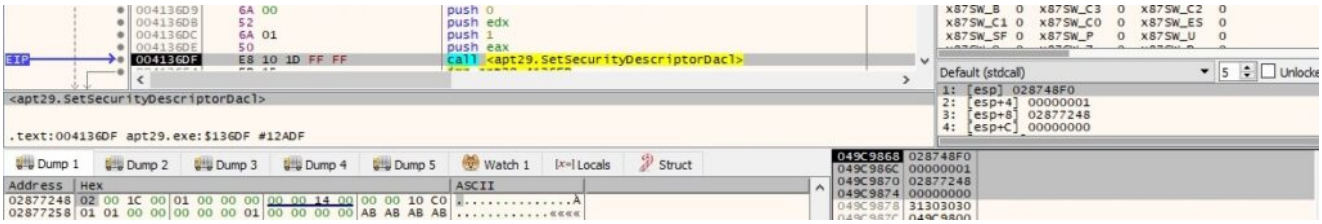


Figure 14

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

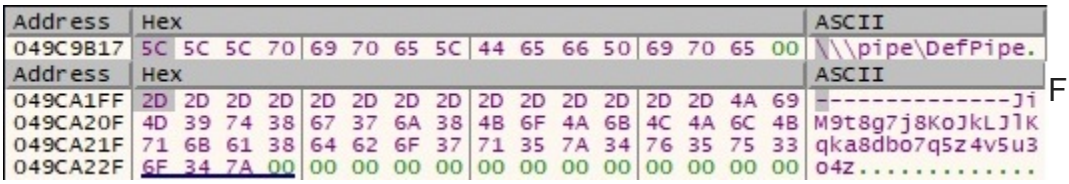


Figure 15

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

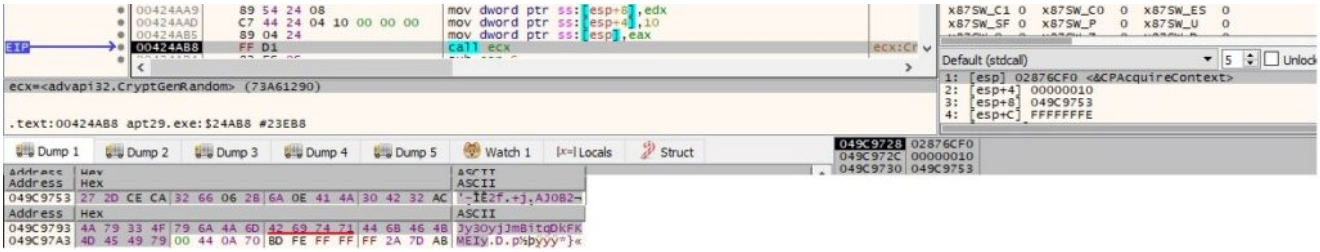


Figure 16

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:

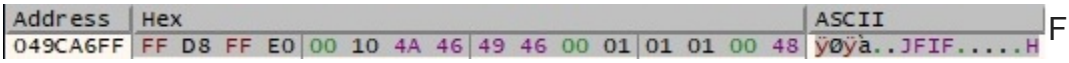


Figure 17

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

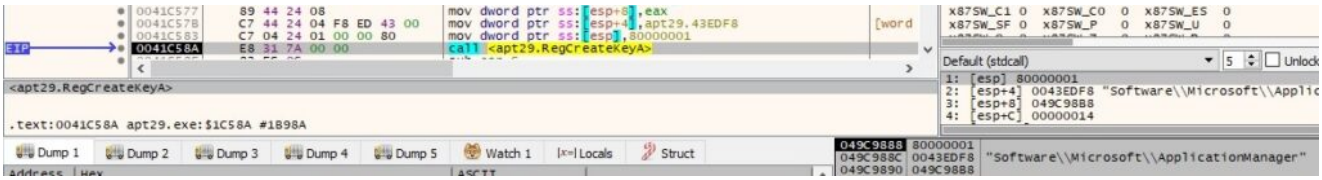


Figure 18

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

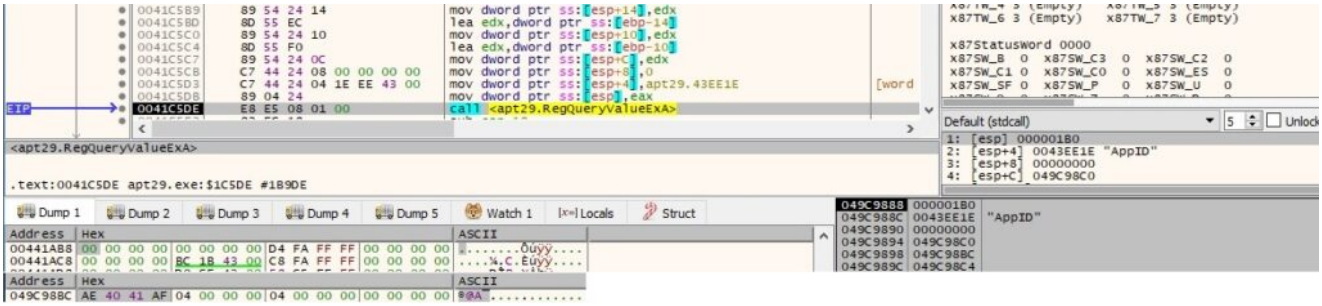


Figure 19

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

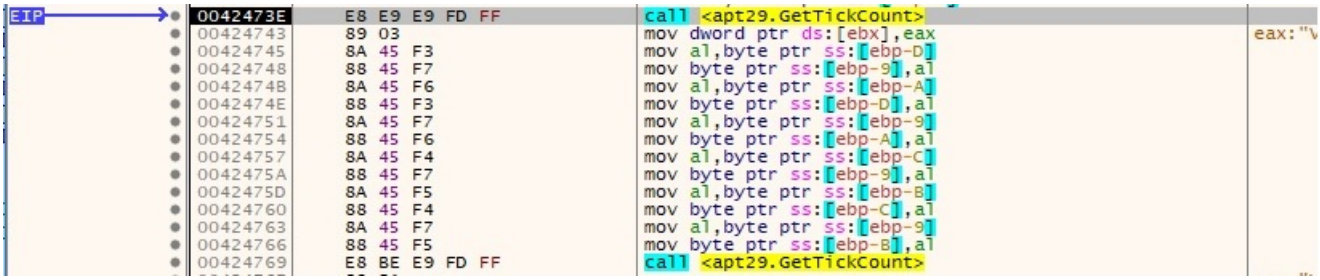


Figure 20

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:

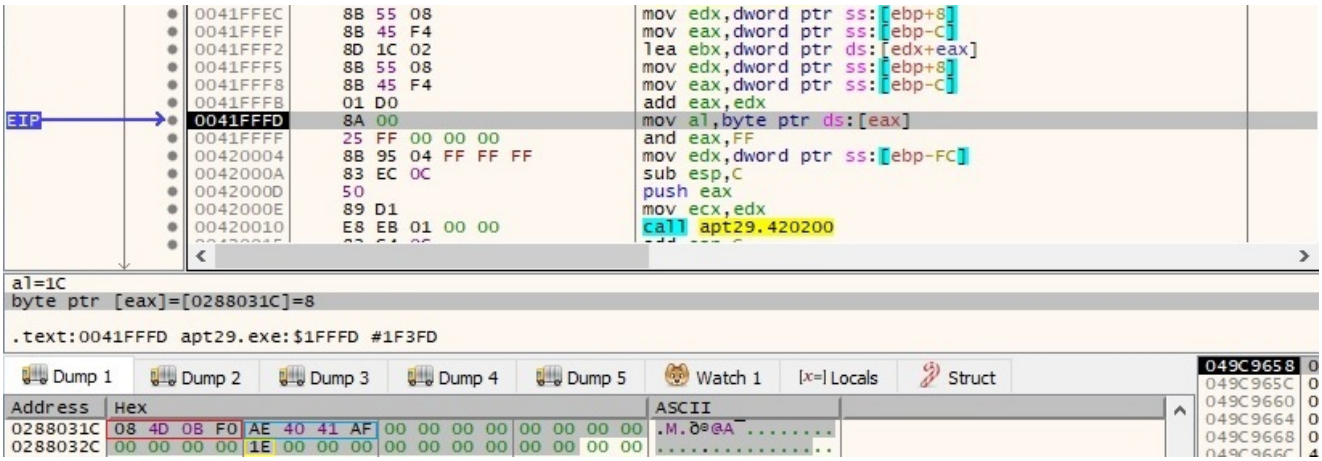


Figure 21

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
```

Address	Hex	ASCII
0288031C	0C A6 39 B4 B9 80 E9 6D 57 11 34 B8 B1 57 AB F8	.¦9´¹.émW.4¸±W«ø
0288032C	86 E3 FC 11 00 AA B2 6B 93 28 00 00 00 00 00 00	.âü..ª²k.(......

Figure 22

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

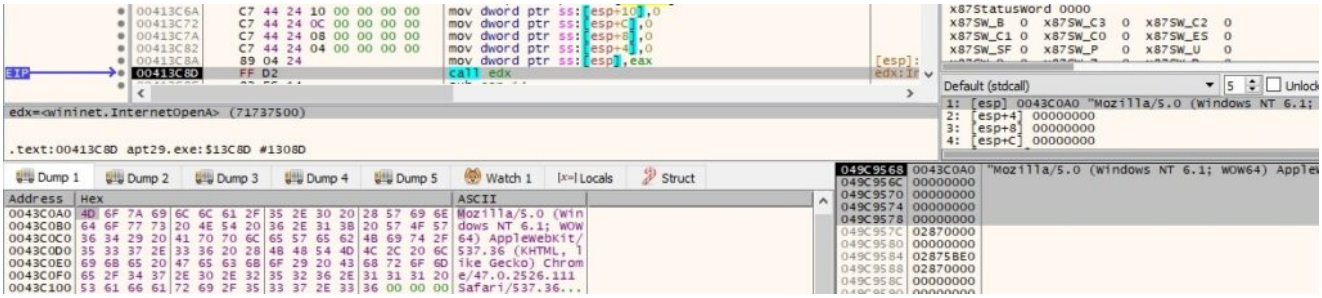


Figure 23

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

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**):

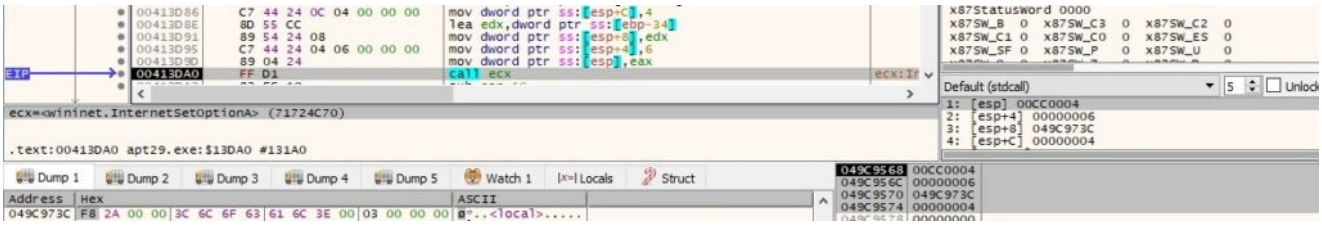


Figure 26

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

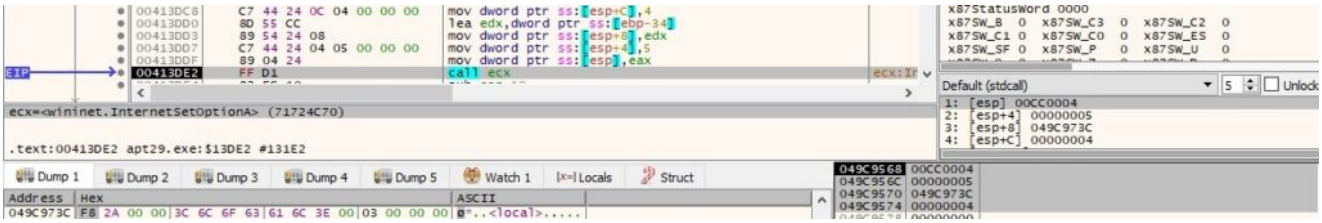


Figure 27

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

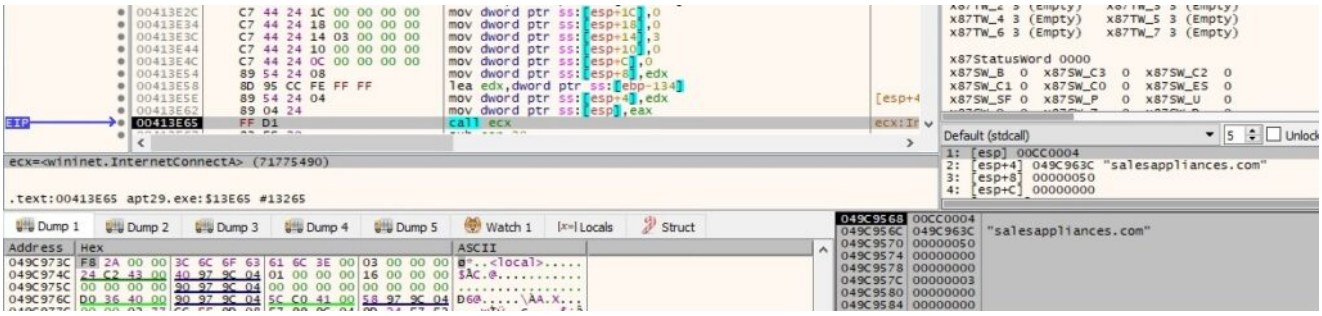


Figure 28

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):

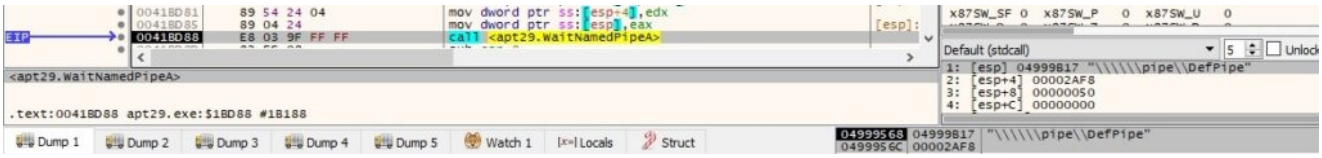


Figure 29

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

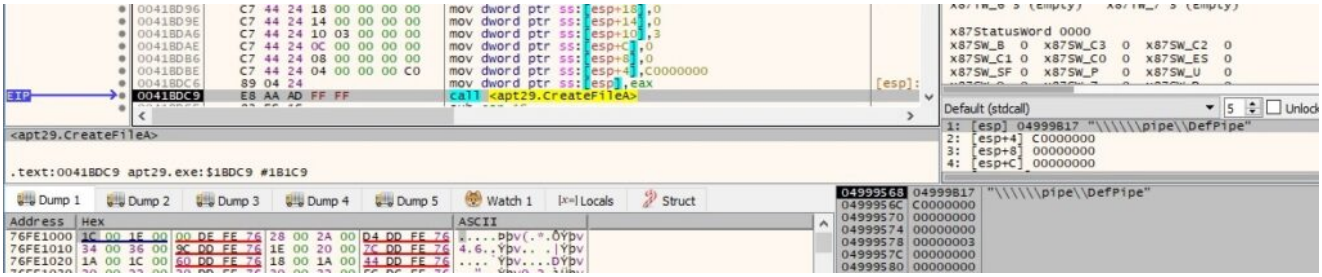


Figure 30

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

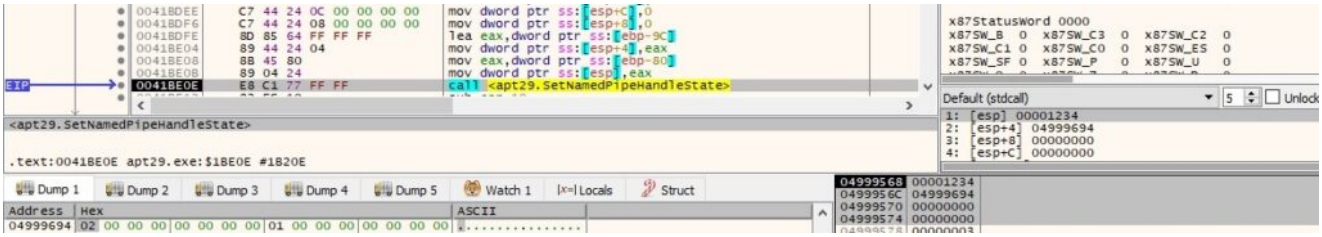


Figure 31

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

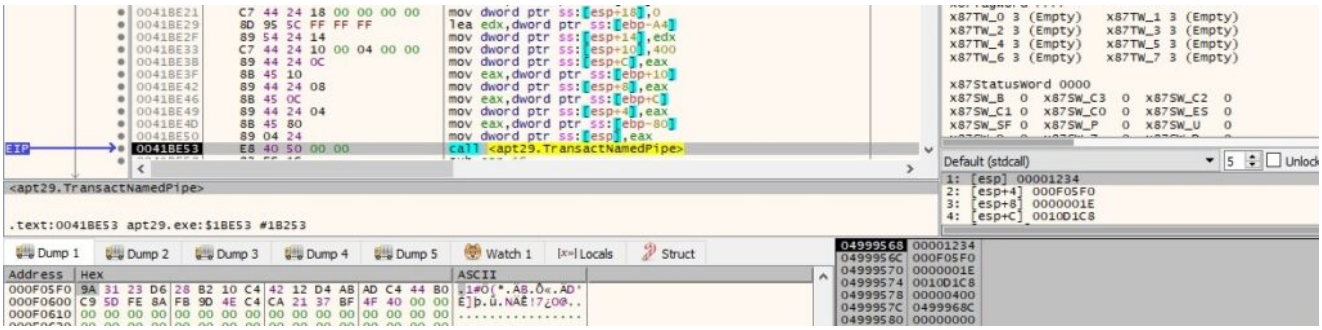


Figure 32

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**):

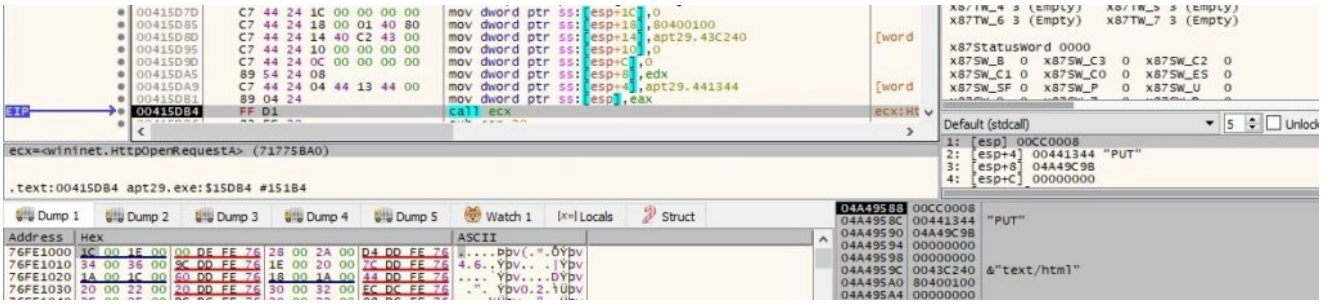


Figure 33

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

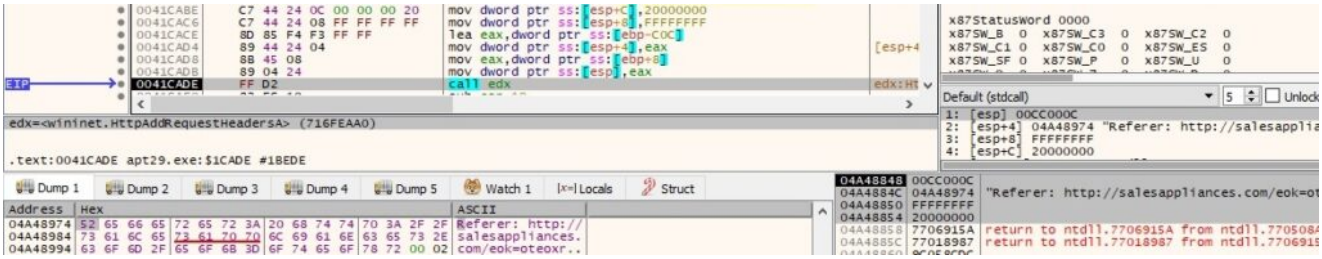


Figure 34

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

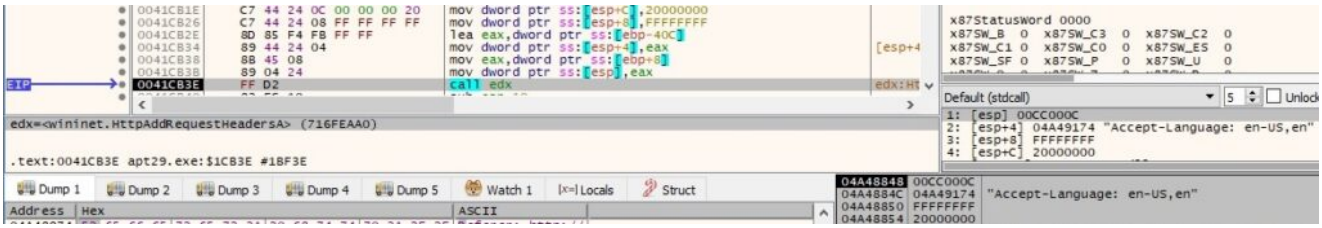


Figure 35

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

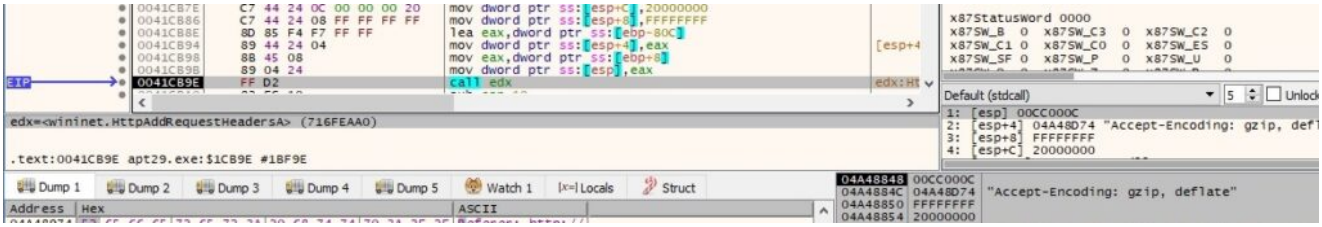


Figure 36

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**):

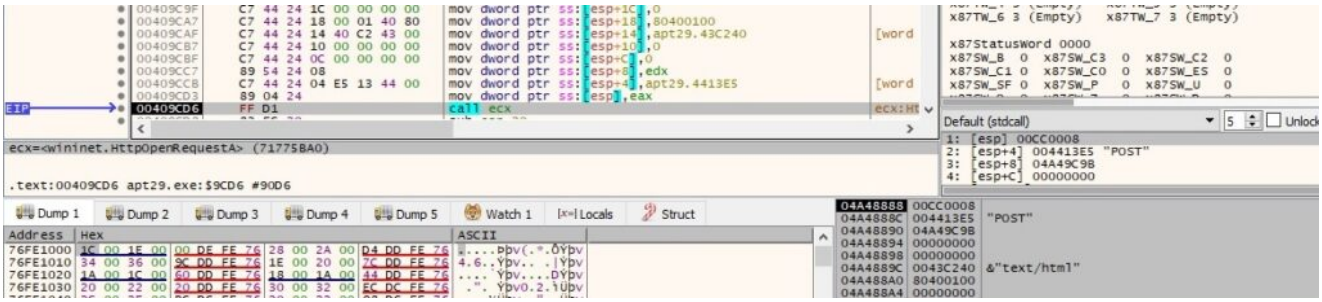


Figure 38

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

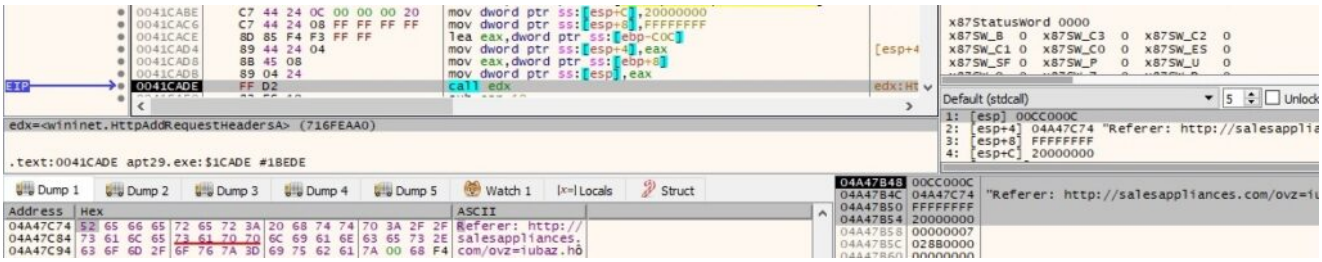


Figure 39

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**):

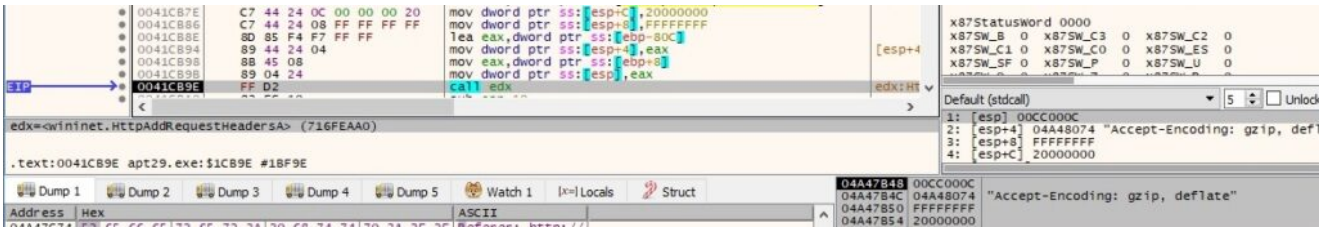


Figure 41

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:

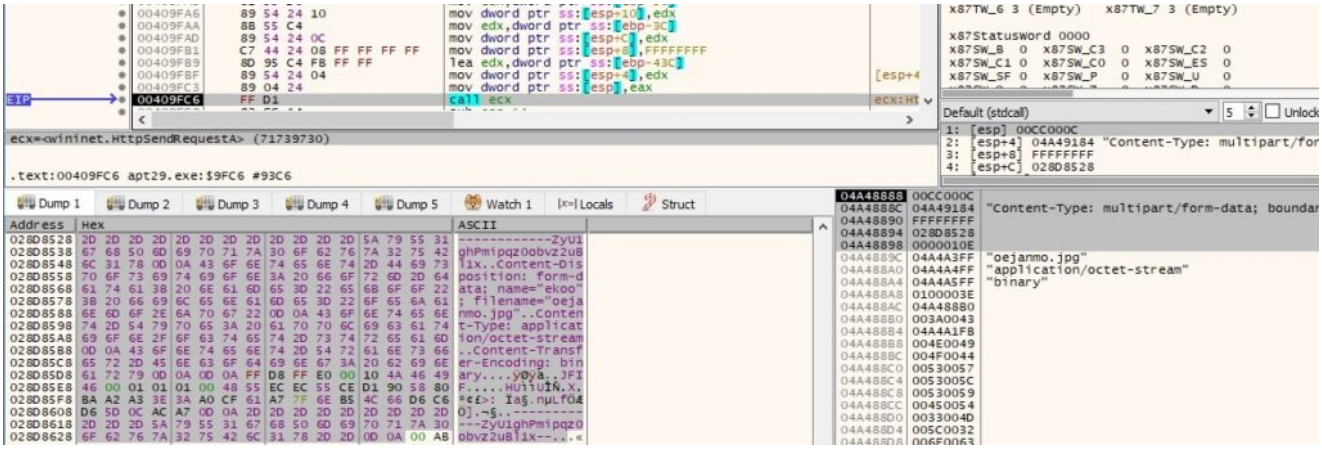


Figure 42

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**):

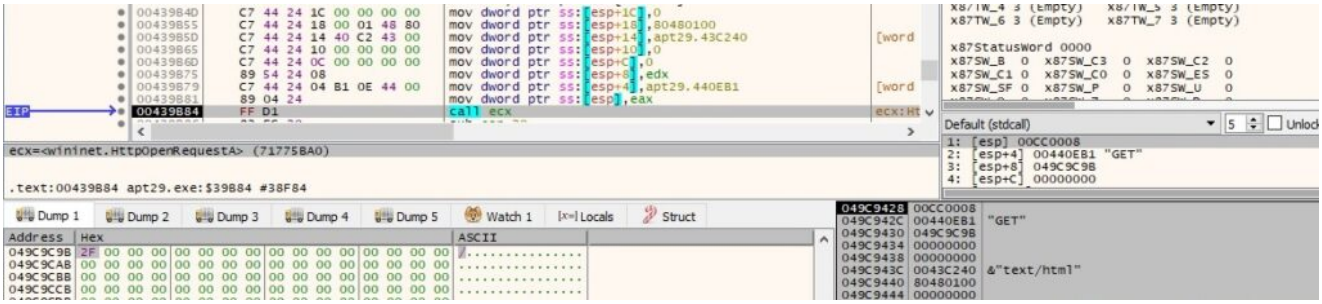


Figure 43

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):

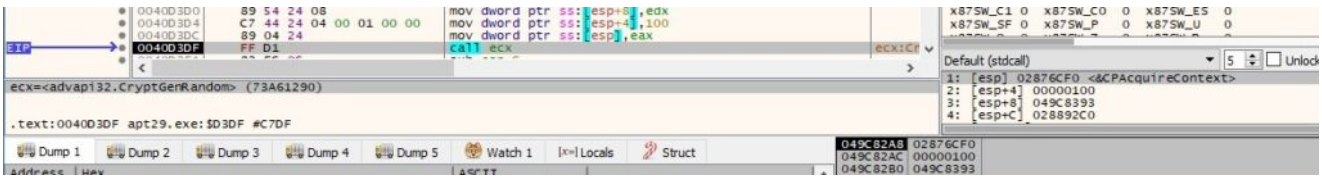


Figure 44

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:

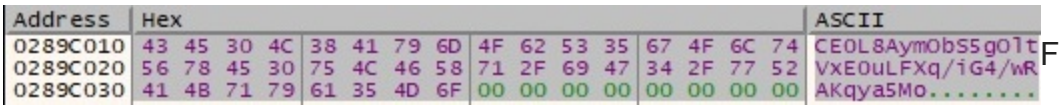


Figure 45

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.

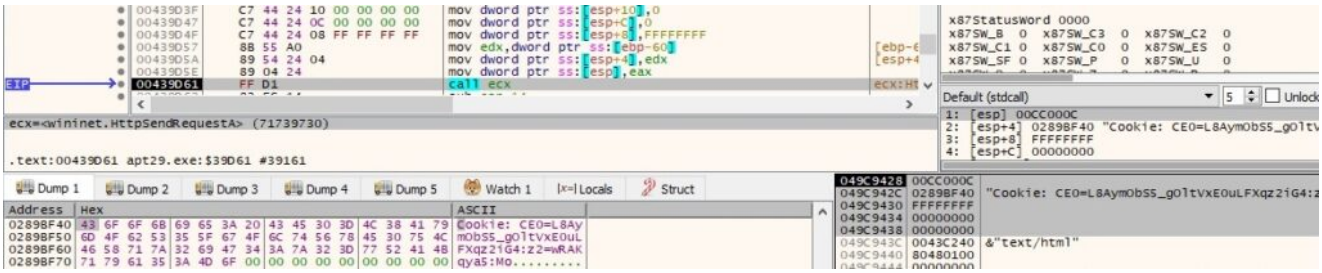


Figure 46

Here is the network request captured by FakeNet:

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:

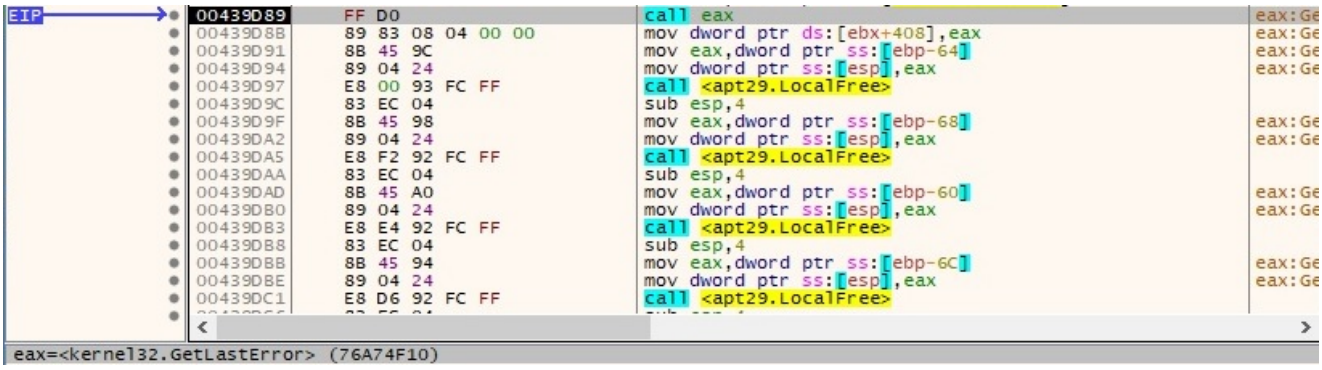


Figure 48

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

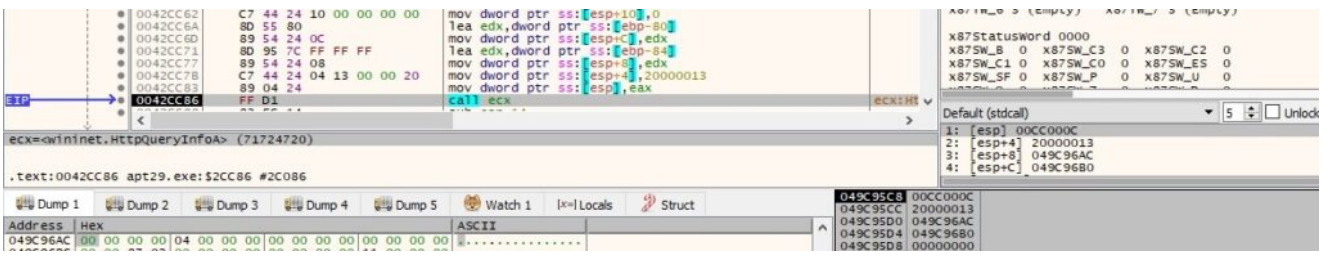


Figure 49

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

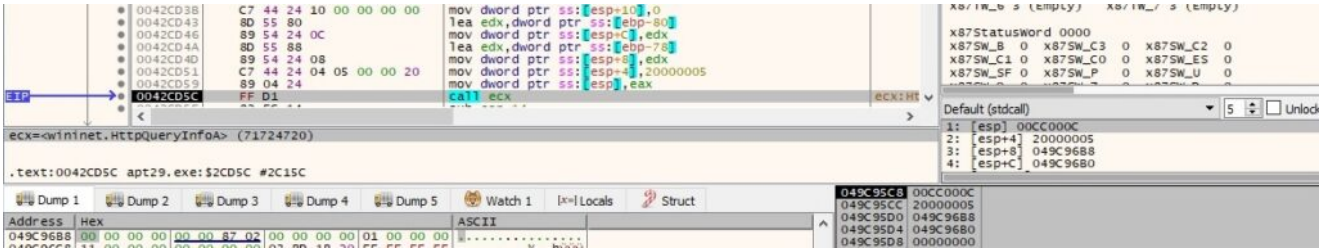


Figure 50

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:

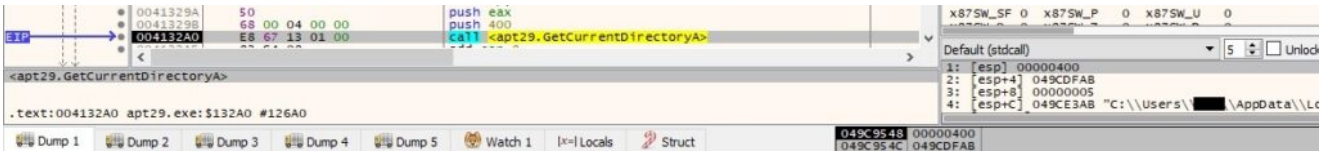


Figure 54

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**):

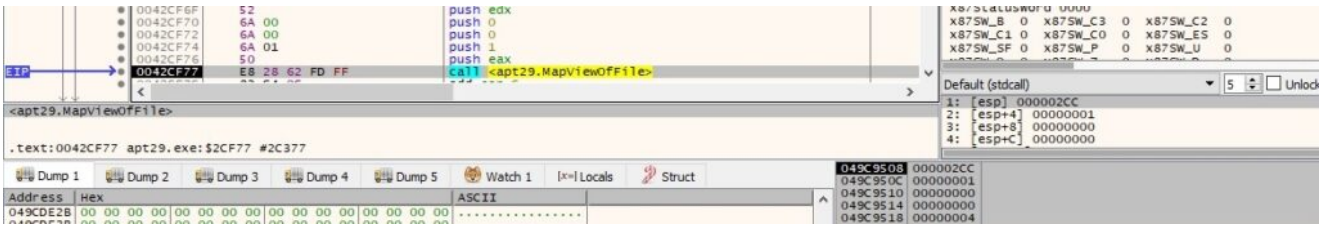


Figure 57

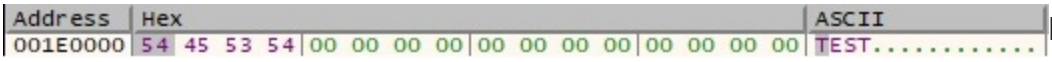


Figure 58

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], 0EFCDAB89h
.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	.;ÙK.hxäðÖDÃÉ^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):

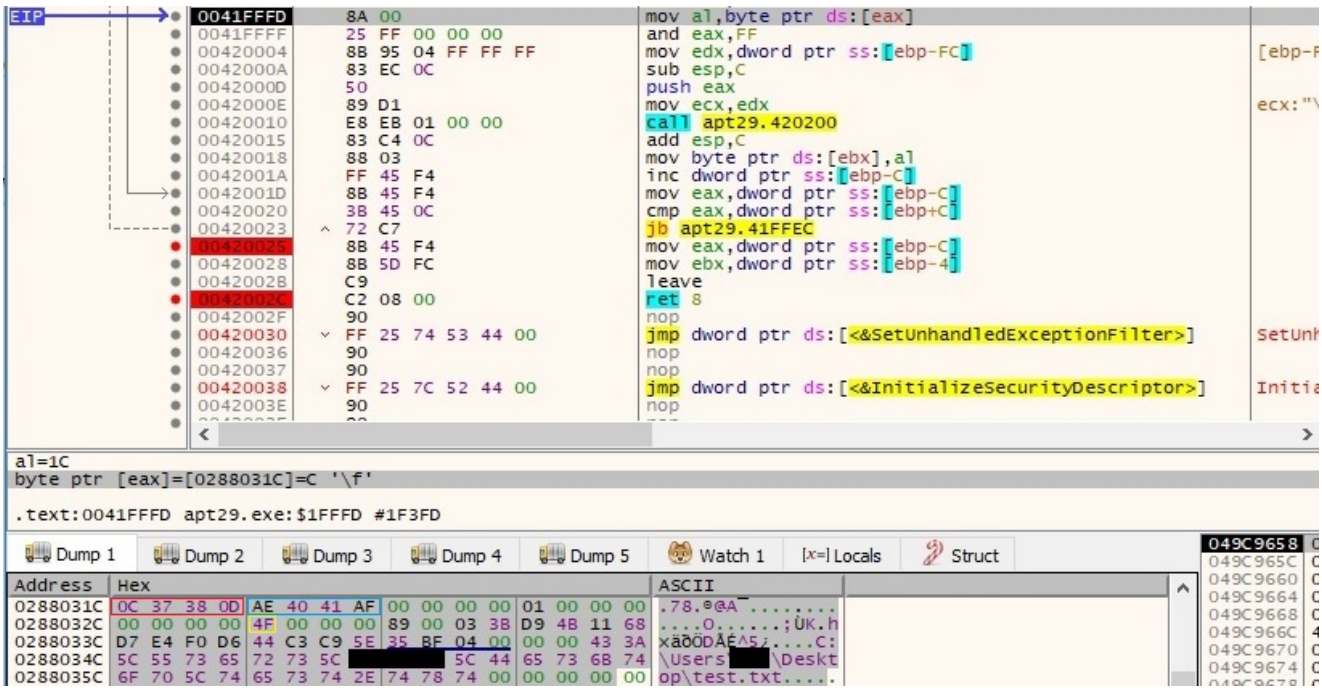


Figure 61

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**):

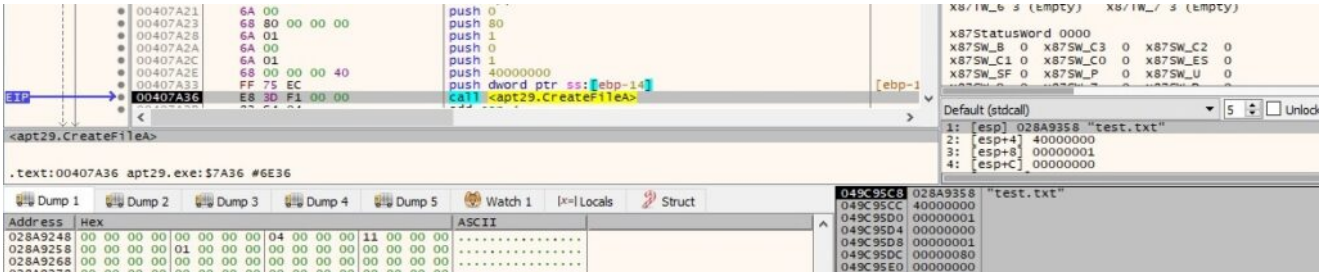


Figure 62

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:

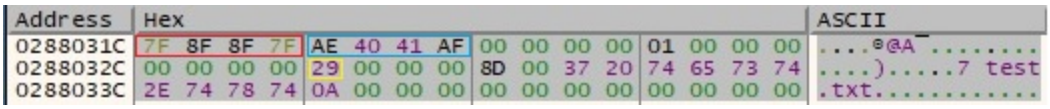


Figure 64

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:

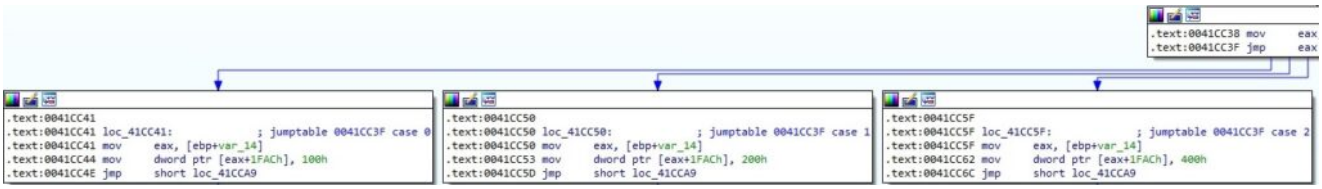


Figure 65

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):

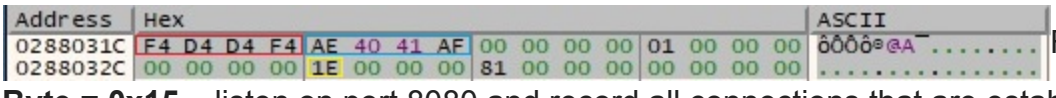


Figure 66

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:

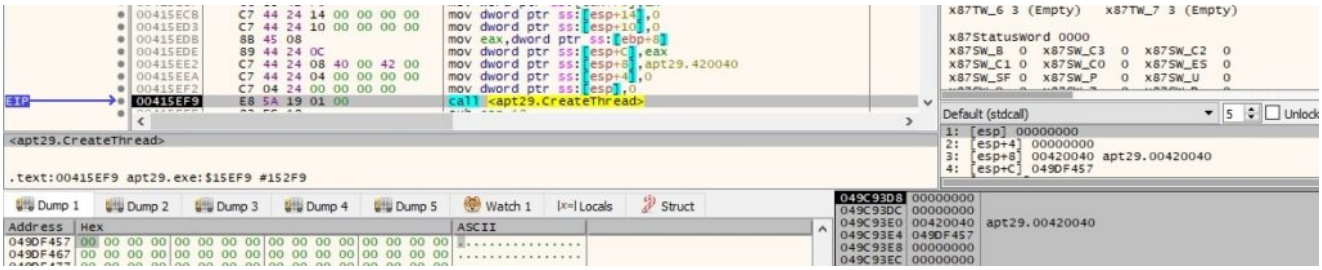


Figure 67

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:

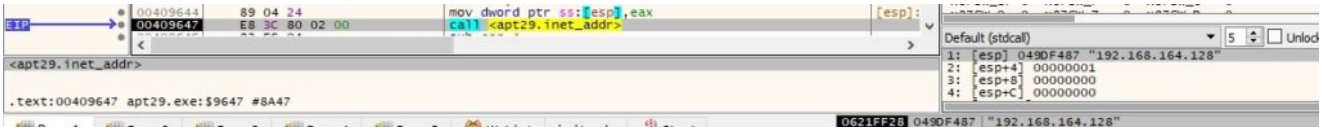


Figure 68

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:

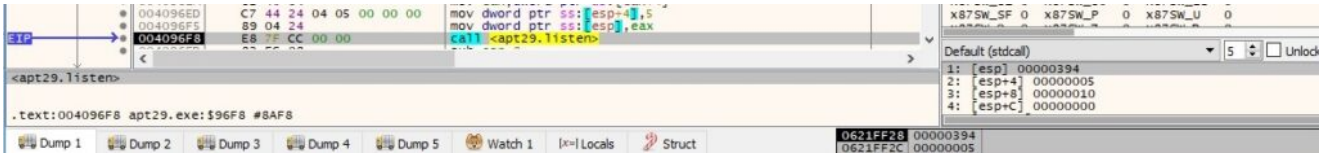


Figure 71

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

Figure 72

For the sake of the analysis, we've emulated an external connection from a remote IP to the local host on port 8080. The getpeername API is utilized to extract the address of the peer to which the socket is connected:

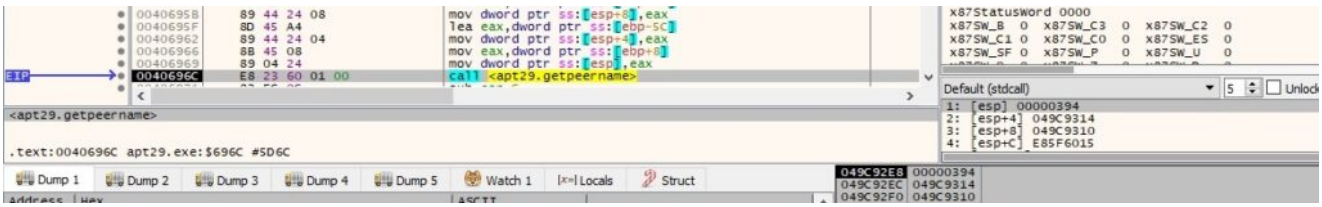


Figure 73

The inet_ntoa routine is the opposite of inet_addr and it's used to convert an IP from a hex form into an ASCII string (dotted-decimal format):

Figure 74

getsockname is utilized to retrieve the local name for the socket:

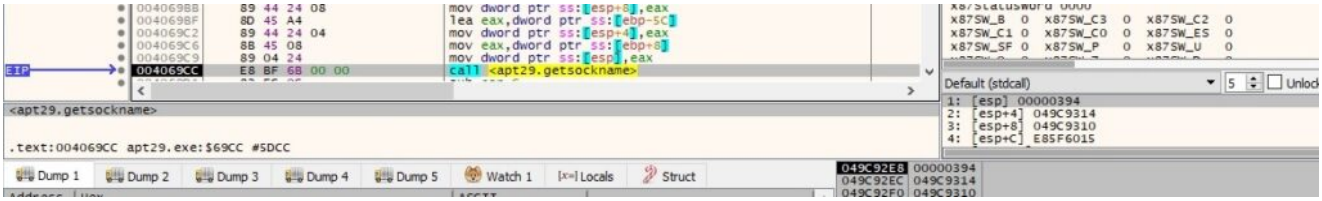


Figure 75

inet_ntoa is used again to convert the IP address from hex to dotted-decimal format, as highlighted in figure 76:

Figure 76

The final buffer that will be exfiltrated contains some details about the network connection (source and destination IPs/ports) and the string "listen":

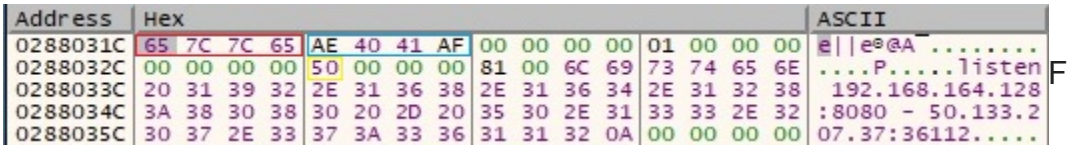


Figure 77

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:

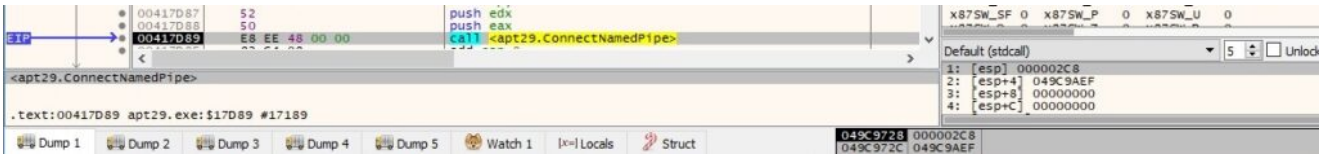


Figure 80

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:

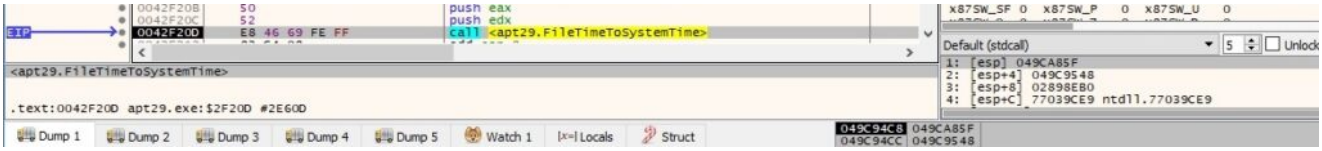


Figure 83

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

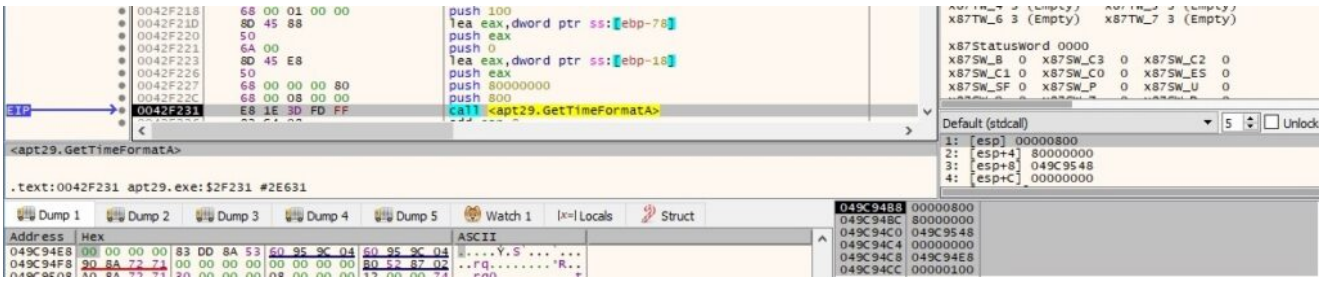


Figure 84

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

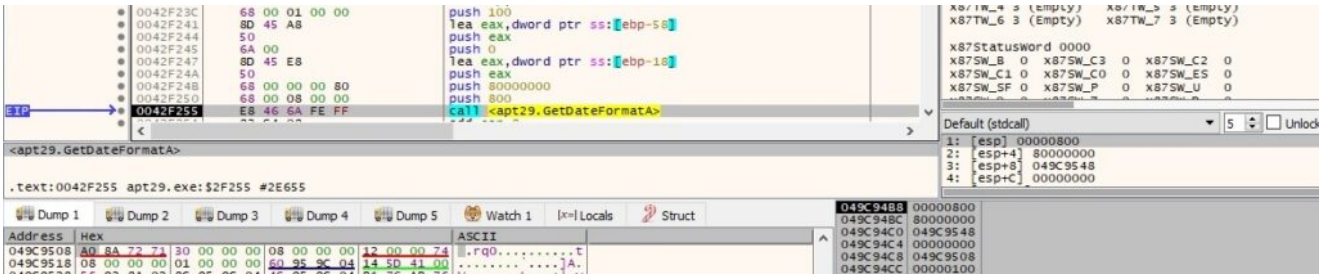


Figure 85

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

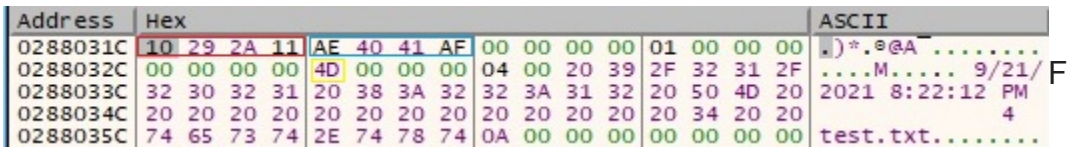


Figure 86

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**):

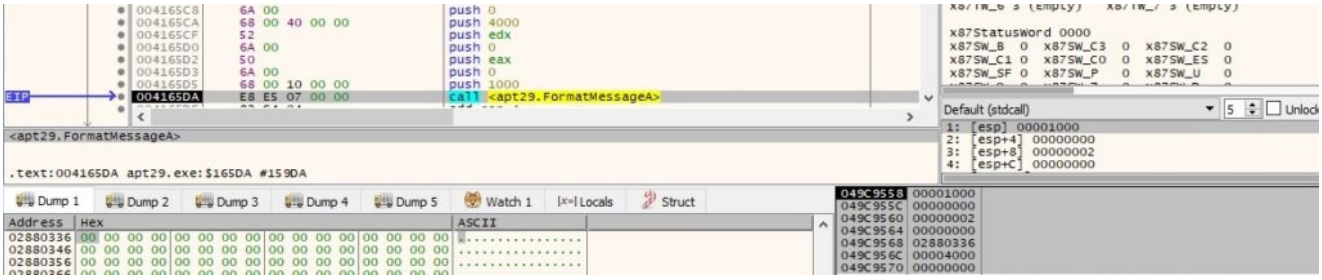


Figure 87

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:

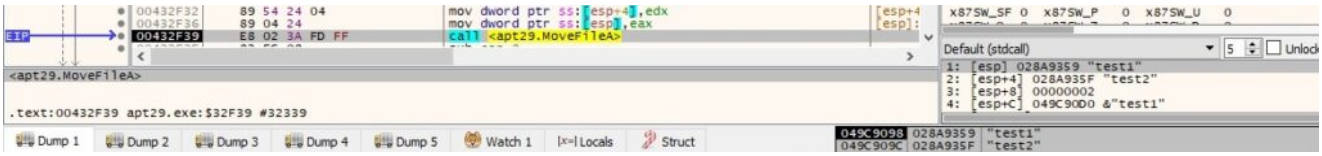


Figure 88

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:

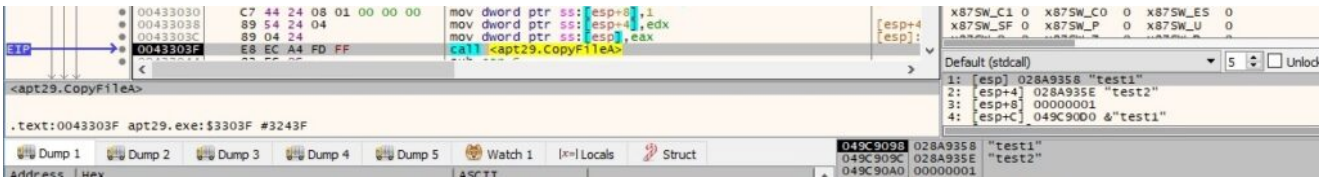


Figure 89

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:

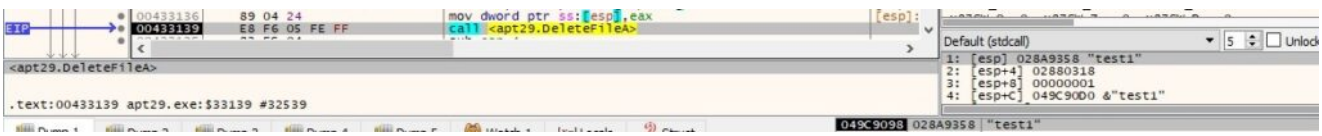


Figure 90

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:

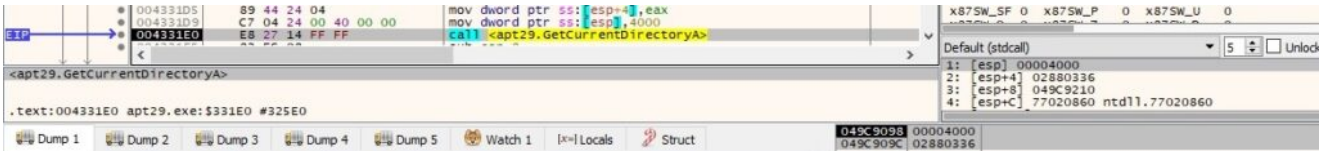


Figure 91

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

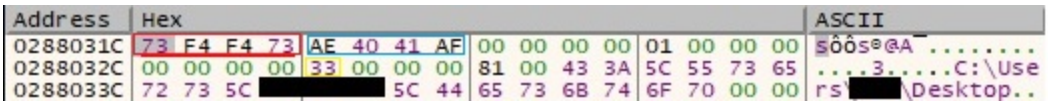


Figure 92

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:

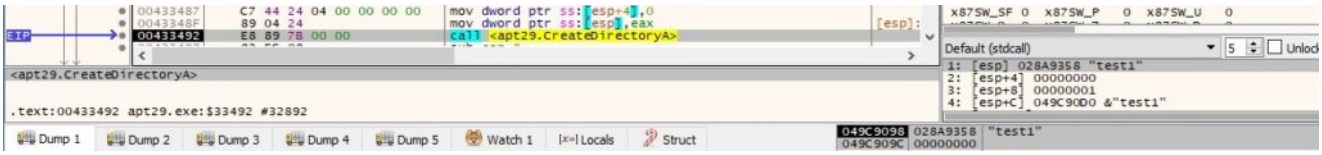


Figure 93

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:

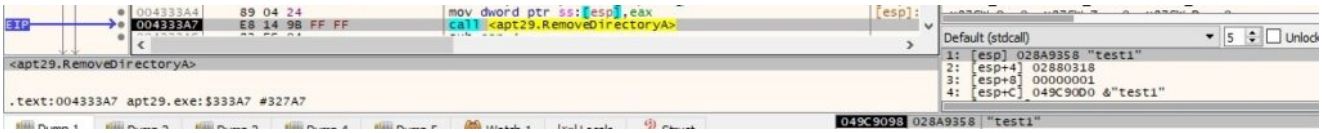


Figure 94

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:

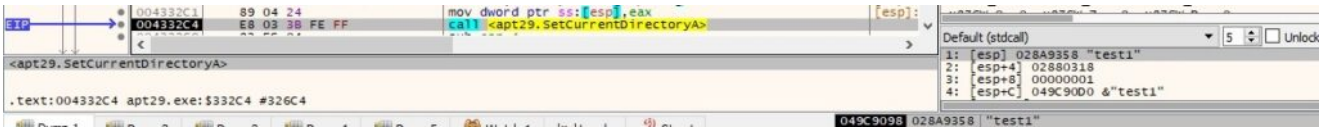


Figure 95

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:

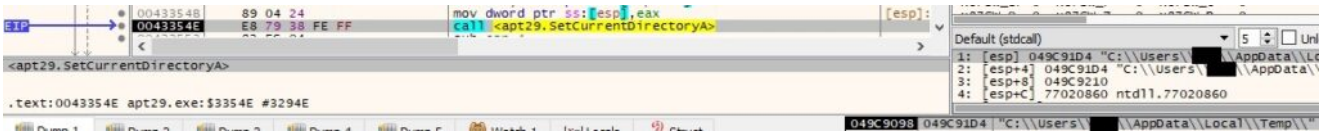


Figure 97

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:

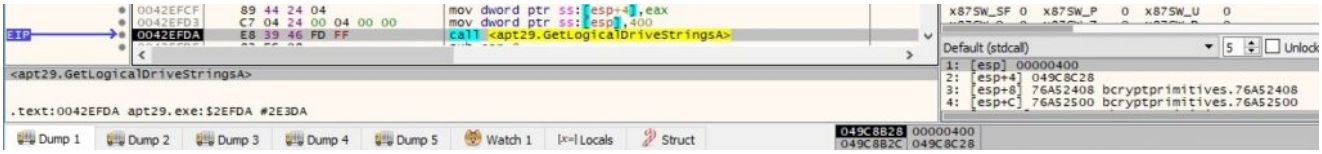


Figure 98

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

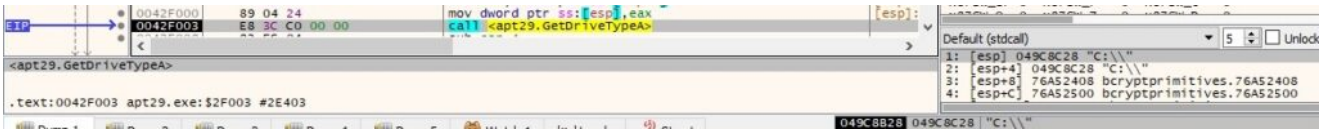


Figure 99

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	iÂÂi®@A¯ ....
0288032C	00 00 00 00 2E 00 00 00 81 00 43 3A 5C 20 66 69	..........C:\ 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:

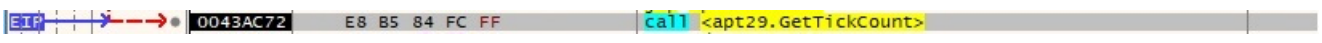


Figure 101

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:

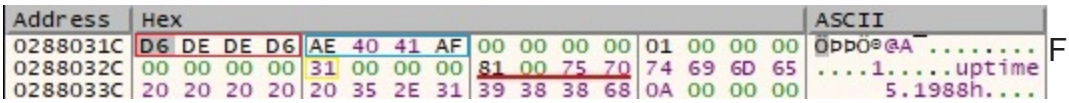


Figure 103

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**):

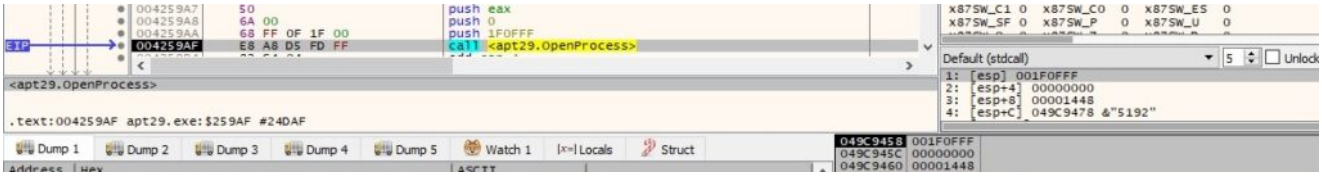


Figure 105

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

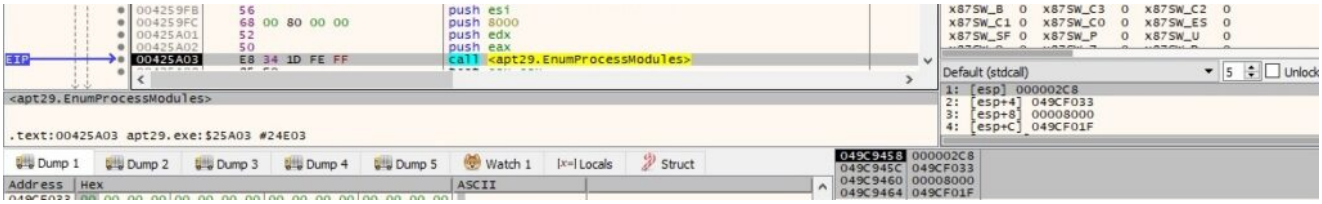


Figure 106

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	¿..À®@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 78 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:

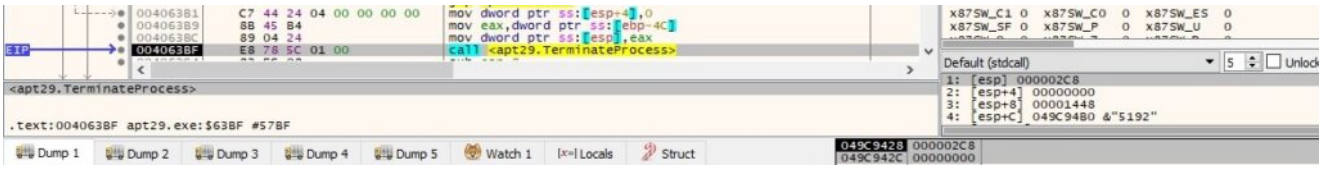


Figure 110

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

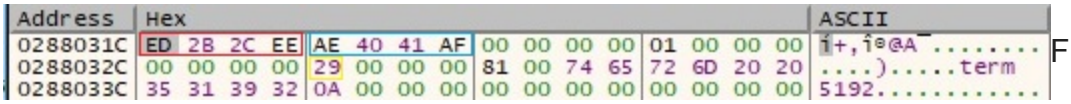


Figure 111

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:

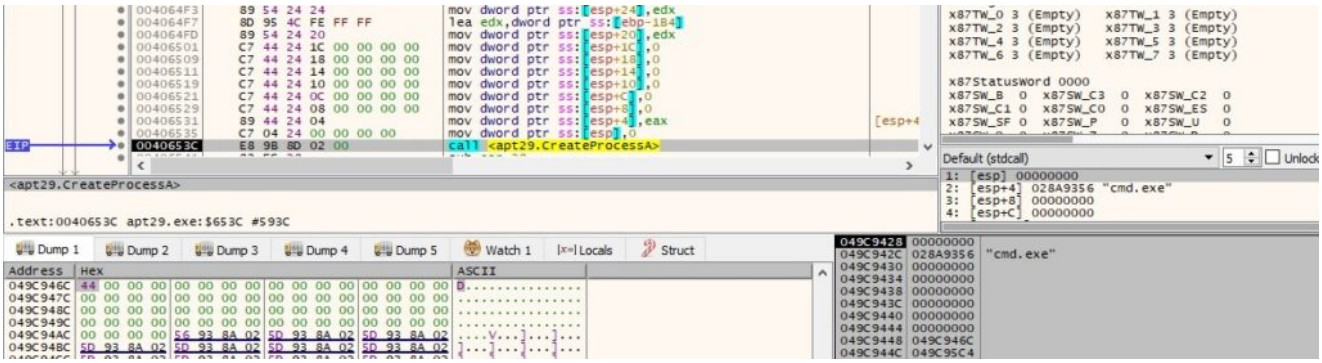


Figure 112

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

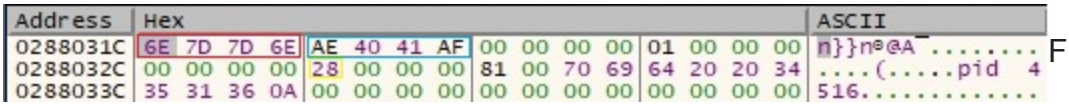


Figure 113

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:

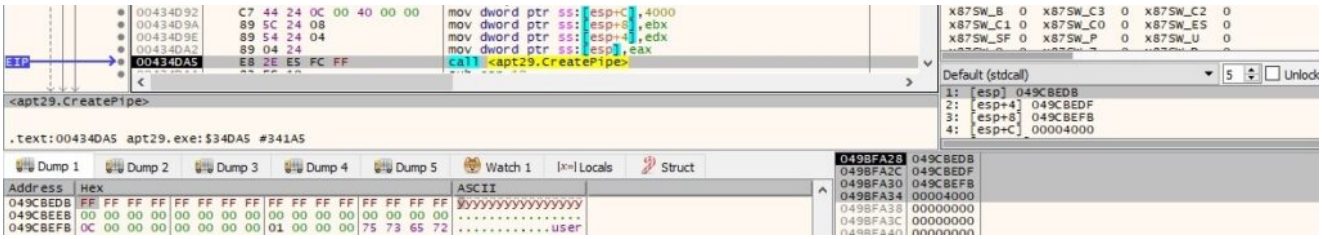


Figure 114

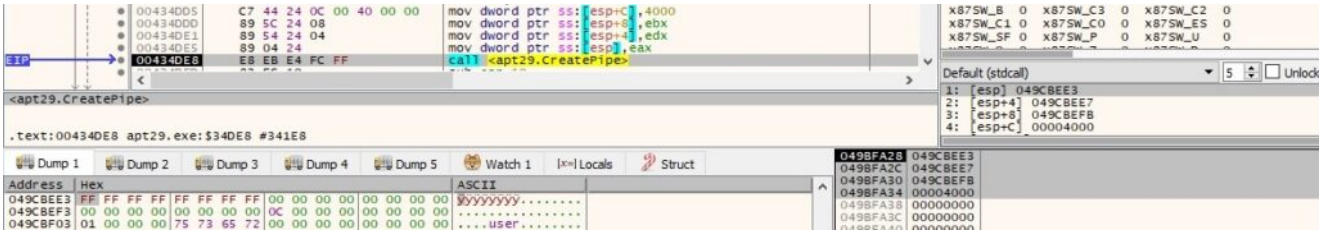


Figure 115

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

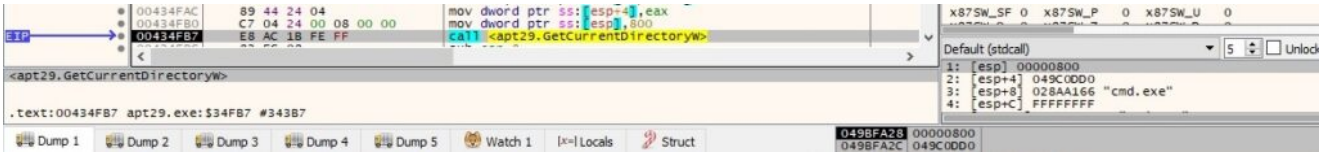


Figure 116

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

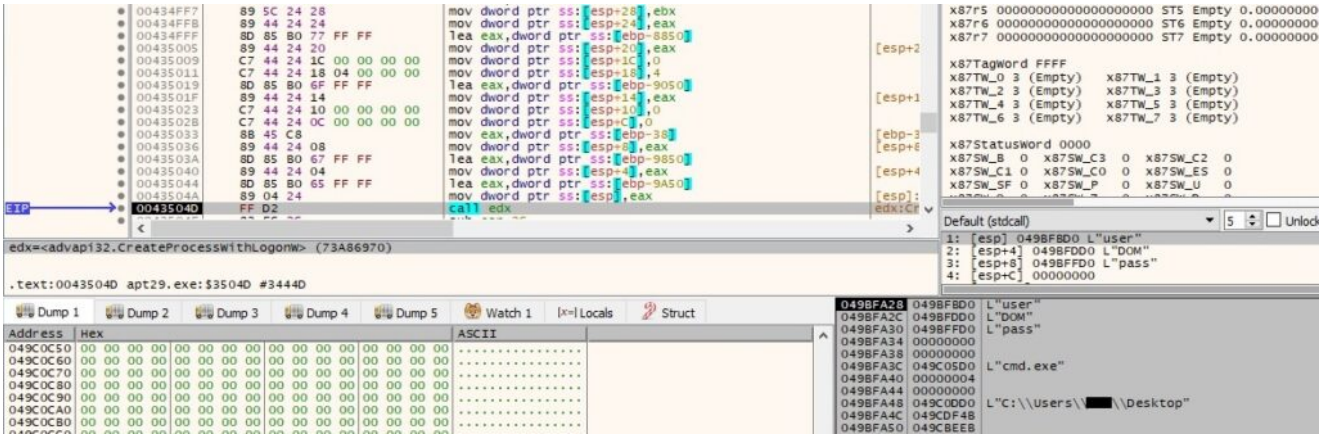


Figure 117

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

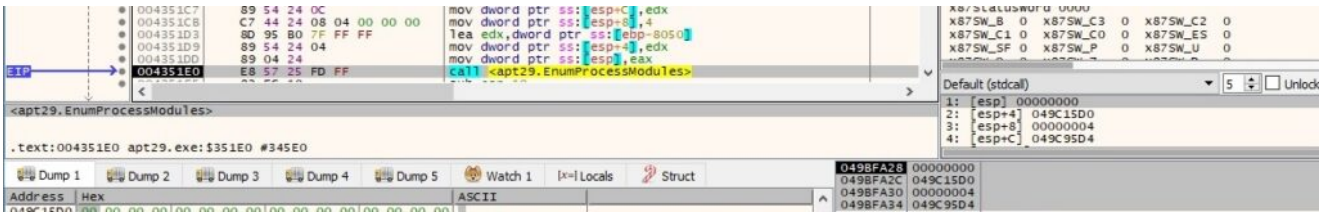


Figure 118

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

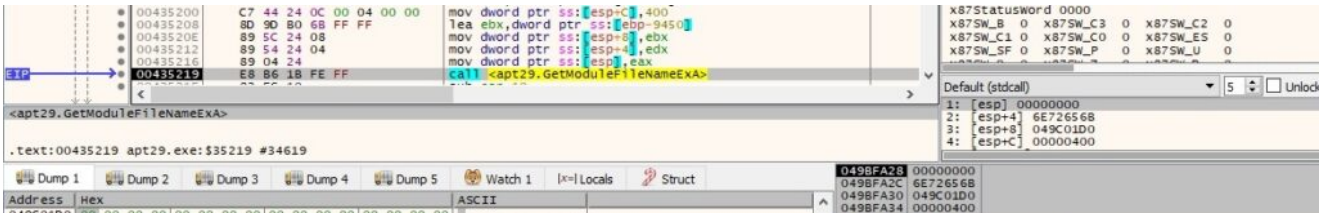


Figure 119

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

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

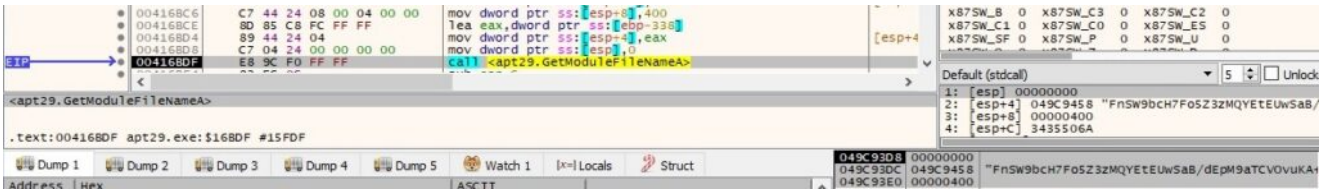


Figure 120

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

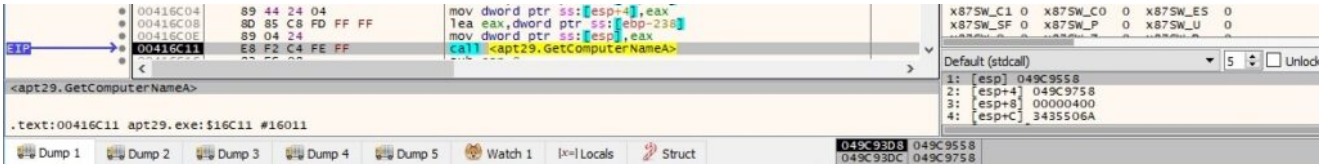


Figure 121

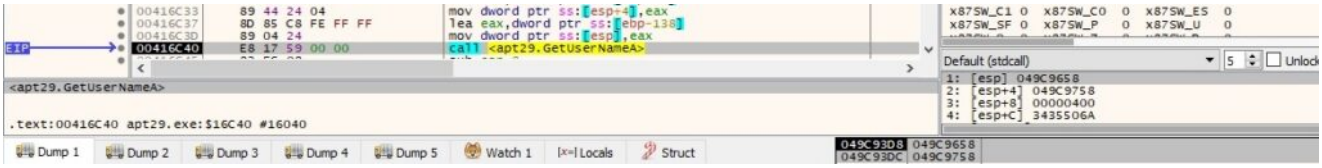


Figure 122

The current process ID is extracted using the GetCurrentProcessId routine:

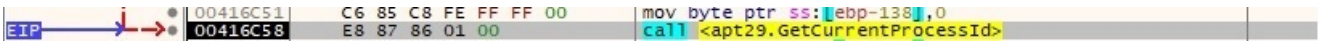


Figure 123

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

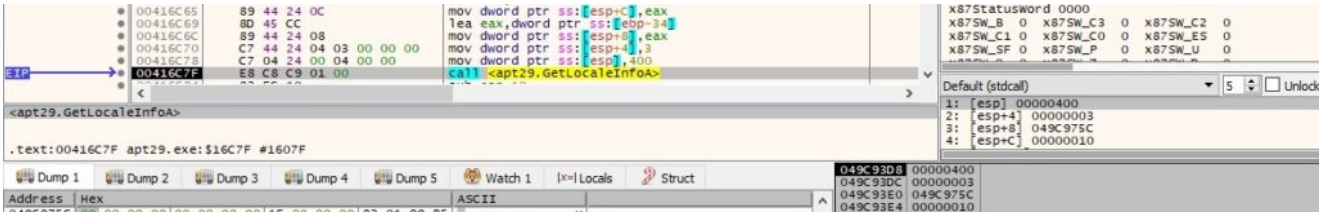


Figure 124

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):

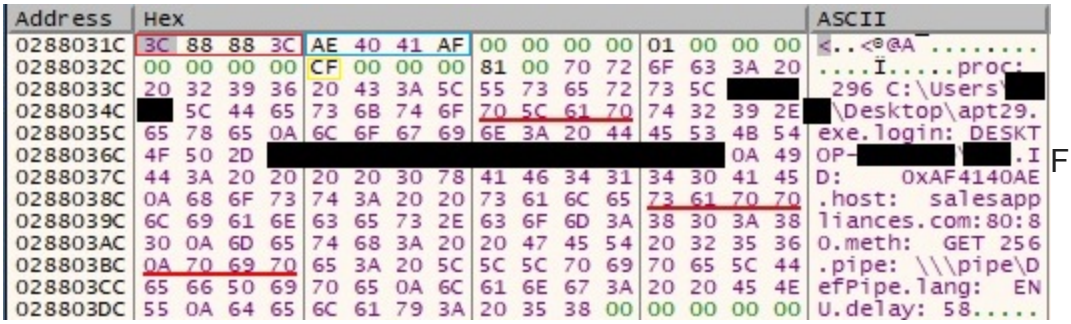


Figure 125

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:

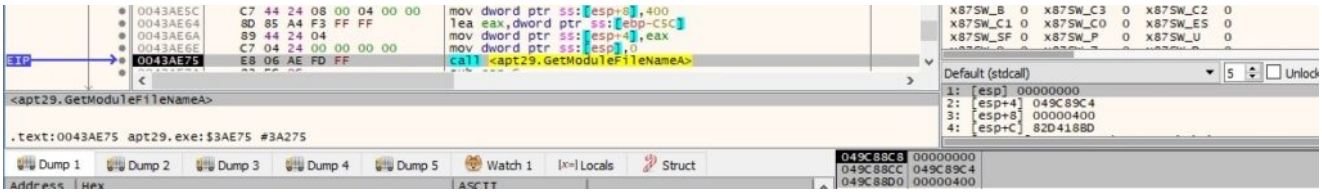


Figure 126

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

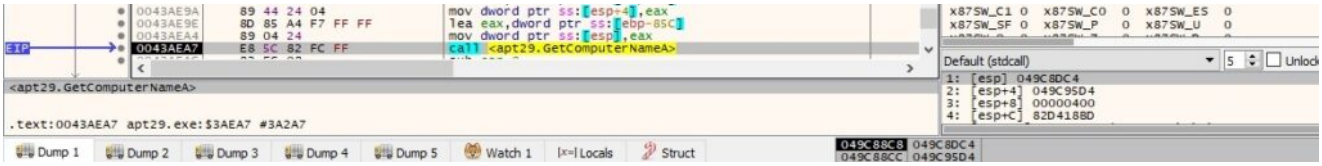


Figure 127

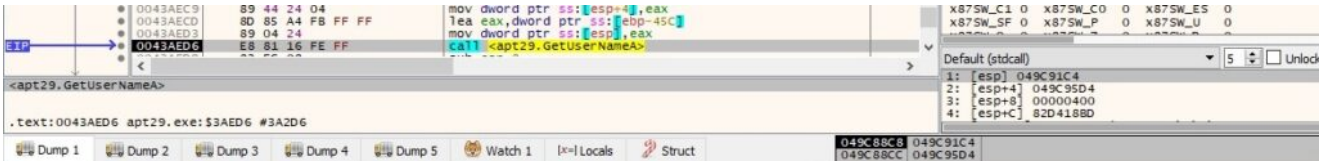


Figure 128

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

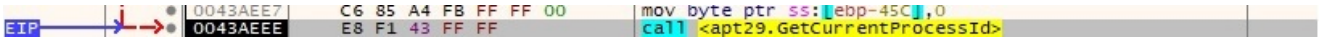


Figure 129

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

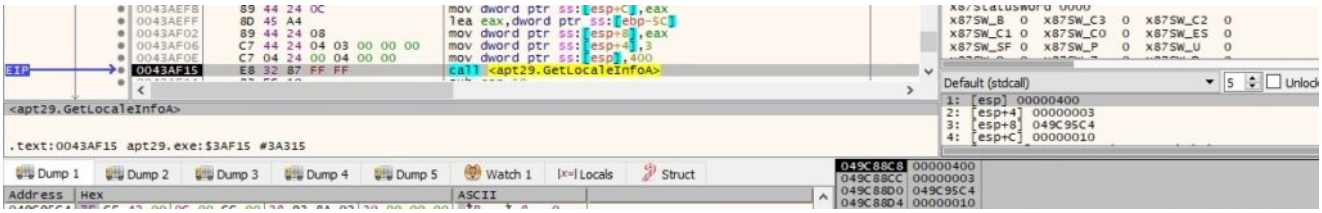


Figure 130

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):

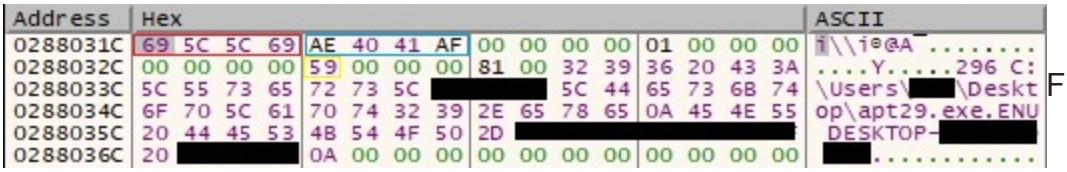


Figure 131

Byte = 0x48 – retrieve the hostname and username

The malware extracts the username and hostname as before:

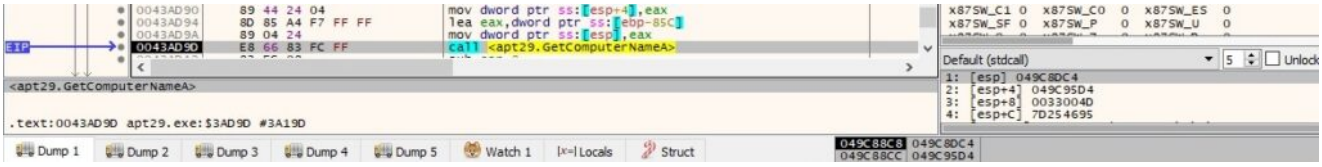


Figure 132

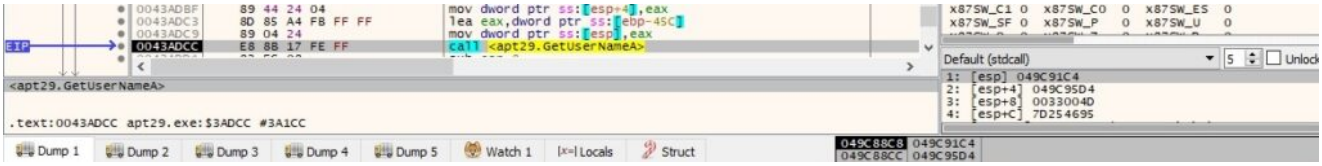


Figure 133

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

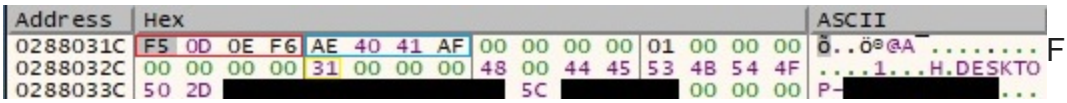


Figure 134

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:

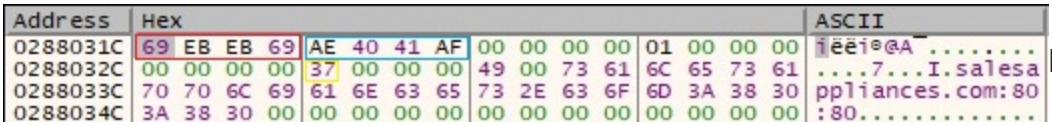


Figure 135

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:

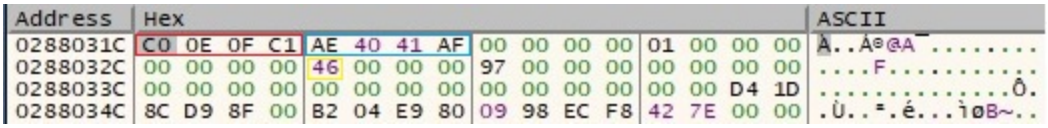


Figure 136

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:

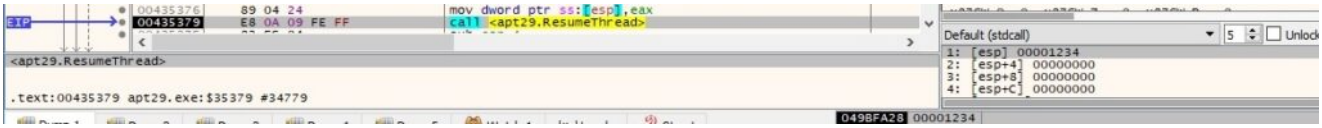


Figure 137

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:

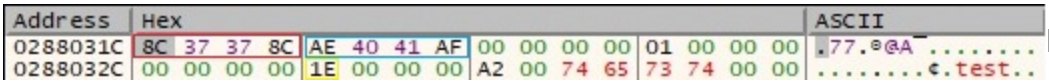


Figure 140

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:

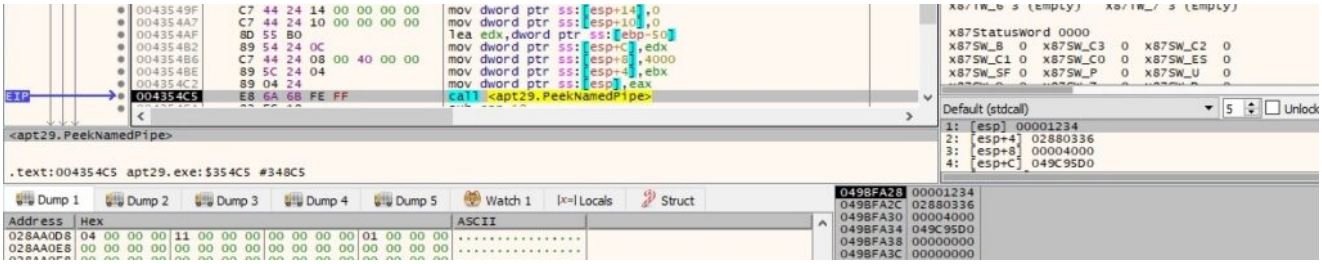


Figure 141

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

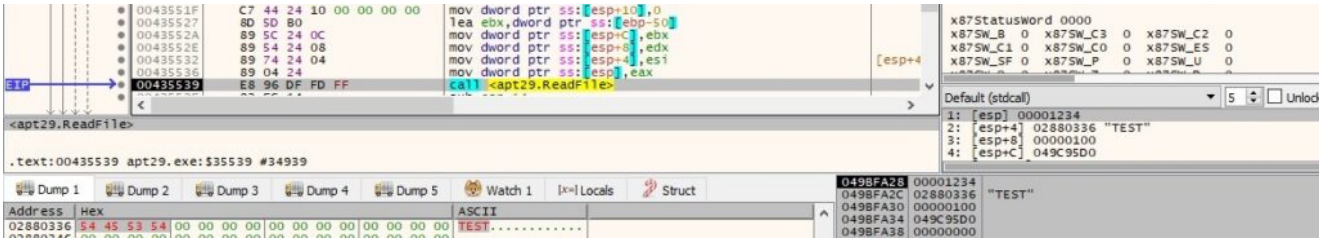


Figure 142

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

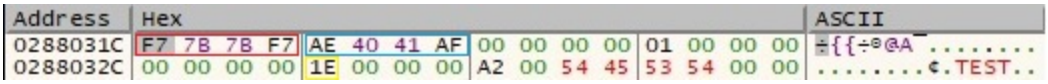


Figure 143

Byte = 0xE3 – kill a process

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

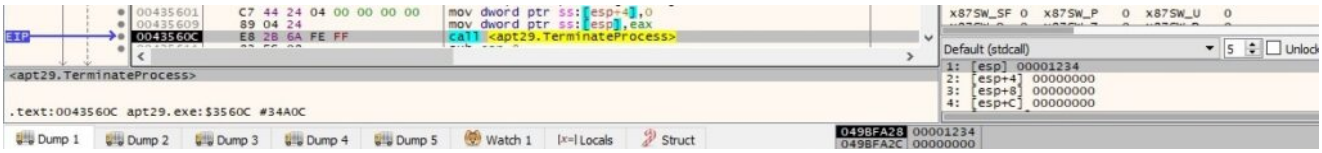


Figure 144

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:

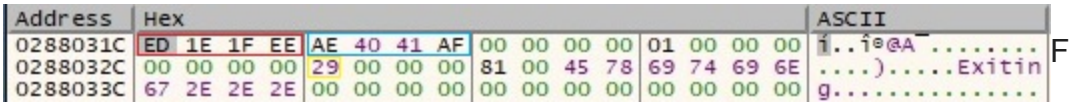


Figure 145

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