A detailed analysis of Lazarus APT malware disguised as Notepad++ Shell Extension

cybergeeks.tech/a-detailed-analysis-of-lazarus-malware-disguised-as-notepad-shell-extension

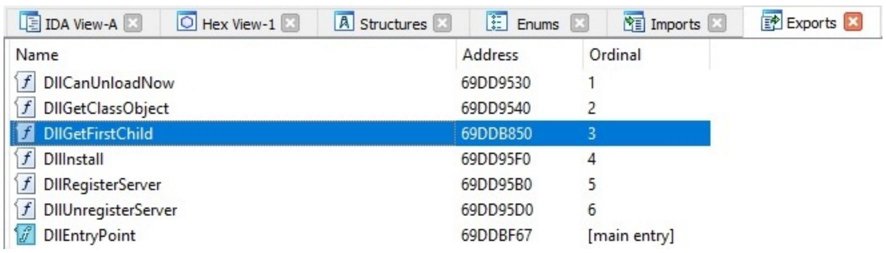
Summary

Lazarus has targeted its victims using job opportunities documents for companies such as LockHeed Martin, BAE Systems, and Boeing. In this case, the threat actor has targeted people that are looking for jobs at Boeing using a document called Boeing BDS MSE.docx (https://twitter.com/ShadowChasing1/status/1455489336850325519). The malware extracts the hostname, username, network information, a list of processes, and other information that will be exfiltrated to one out of the four C2 servers. The data targeted for exfiltration is compressed, XOR-encrypted and then Base64-encoded before being transmitted to the C2 server. The Trojan implements four actions that include downloading and executing a .exe or .dll file, loading a PE (Portable Executable) into the process memory, and executing shellcode.

Technical analysis

SHA256: 803dda6c8dc426f1005acdf765d9ef897dd502cd8a80632eef4738d1d7947269

The file is a DLL that has 7 exports. Only one of these functions implements malicious activity (DllGetFirstChild):



Name	Address	Ordinal
DllCanUnloadNow	69DD9530	1
DllGetClassObject	69DD9540	2
DllGetFirstChild	69DDB850	3
DllInstall	69DD95F0	4
DllRegisterServer	69DD95B0	5
DllUnregisterServer	69DD95D0	6
DllEntryPoint	69DDBF67	[main entry]

Figure 1

The malware retrieves the User Agent by calling the ObtainUserAgentString function. There is also a User Agent that is hardcoded in the binary "Mozilla / 5.0 (Windows NT 10.0; WOW64; Trident / 7.0; rv:11.0) li", which is Internet Explorer on Windows 10:

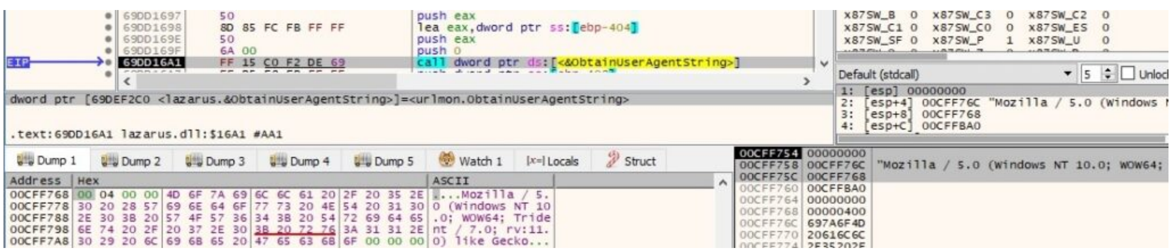


Figure 2

The binary extracts the current system date and time using the GetSystemTimeAsFileTime API:

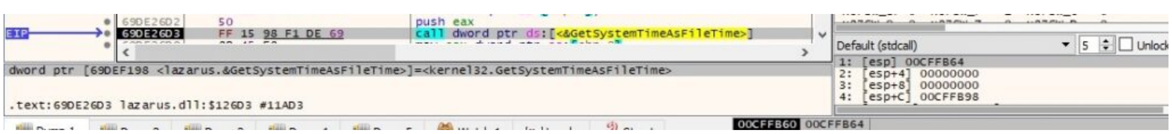


Figure 3

GetModuleHandleW is utilized to retrieve a module handle for ntdll.dll:

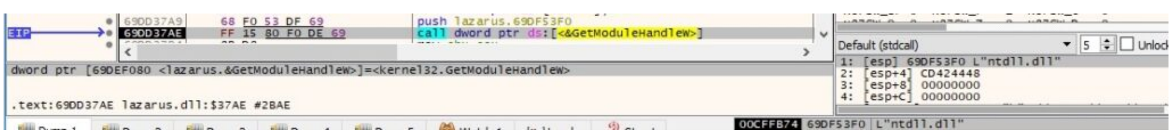


Figure 4

The process gets the address of the following export functions using the GetProcAddress routine: "RtlGetCompressionWorkSpaceSize", "RtlCompressBuffer", "RtlDecompressBuffer", "RtlGetVersion". An example of a function call is shown in figure 5:

Figure 5

The NetBIOS name of the local computer is extracted via a function call to GetComputerNameW:

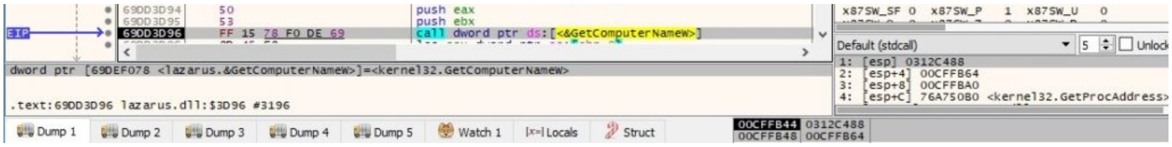


Figure 6

The GetAdaptersInfo API is used to retrieve adapter information for the local machine:

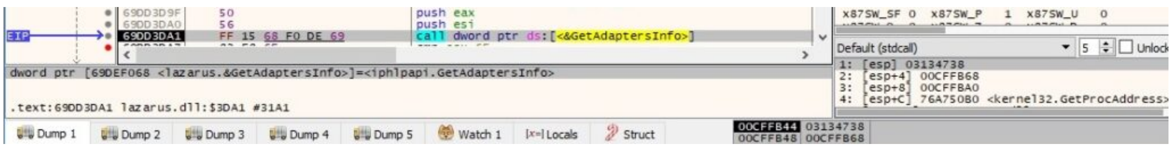


Figure 7

The MAC address extracted above is written to a buffer:

Address	Hex	ASCII
03134520	30 00 30 00 30 00 63 00 32 00 39 00 32 00 35 00	0.0.0.c.2.9.2.5.
03134530	36 00 36 00 31 00 35 00 30 00 30 00 30 00 30 00	6.6.1.5.0.0.0.0.

Figure 8

The file extracts the command-line string for the current process:

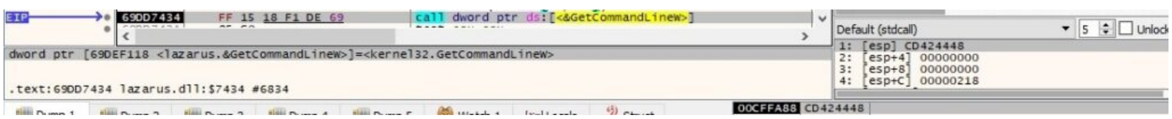


Figure 9

CommandLineToArgvW is utilized to extract an array of pointers to the command-line arguments, along with a count of arguments (similar to argv and argc):

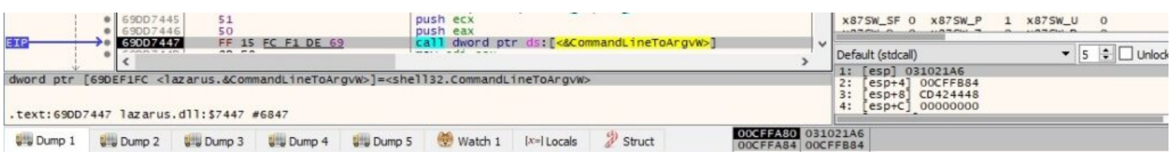


Figure 10

According to an article published at https[:]//zhuanlan.zhihu.com/p/453894016, the malware is supposed to run with the following parameters:

"NTPR P6k+pR6iIKwJpU6oR6ZilgKPL7IxsitJAnpIYSx2KldSSRFFyUIzTBVFAwgzBkI2PS/+EgASBik/GgYBwBbRNy7pP+Xq4uTsxOXU6NPmudaEz7Xy5

The binary decrypts the above parameter using a custom algorithm displayed in figure 11. The list of resulting strings contains multiple C2 servers:

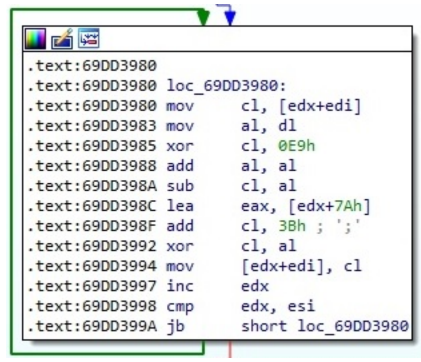


Figure 11

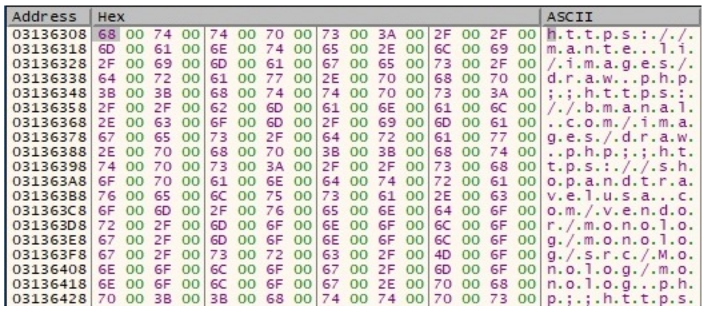


Figure 12

The following URLs have been decrypted:

https[:]//mante.li/images/draw.php

https[:]//bmanal.com/images/draw.php

https[:]//shopandtravelusa.com/vendor/monolog/monolog/src/Monolog/monolog.php

https[:]//industryinfostructure.com/templates/worldgroup/view.php

The GetNetworkParams routine is used to retrieve network parameters for the local computer:

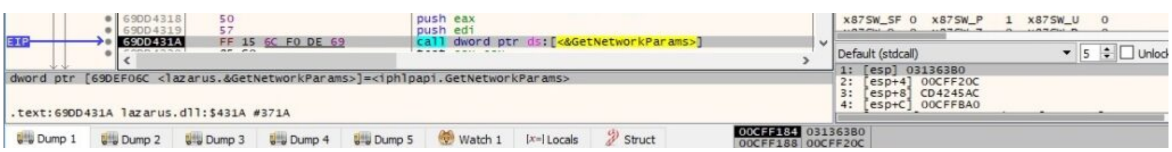


Figure 13

The malicious process extracts the name of the DNS domain assigned to the local host (0x2 = **ComputerNameDnsDomain**):

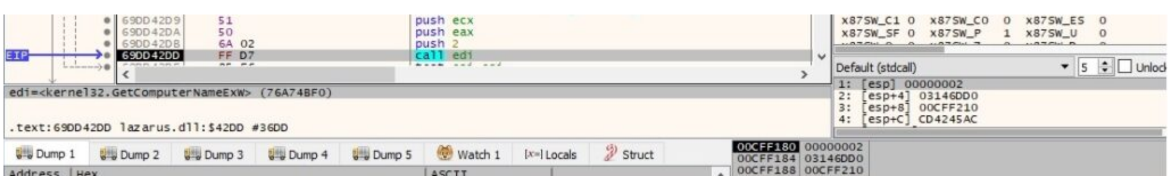


Figure 14

The following network information is written to a temporary buffer:

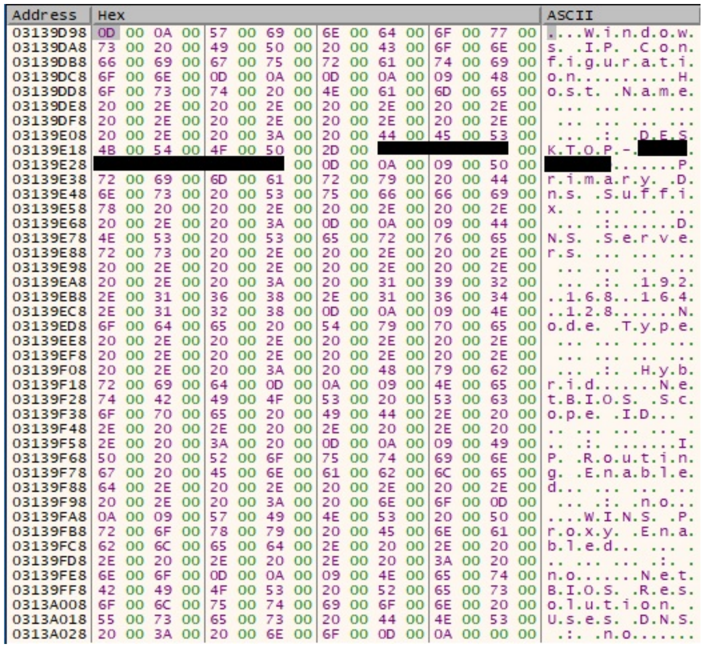


Figure 15

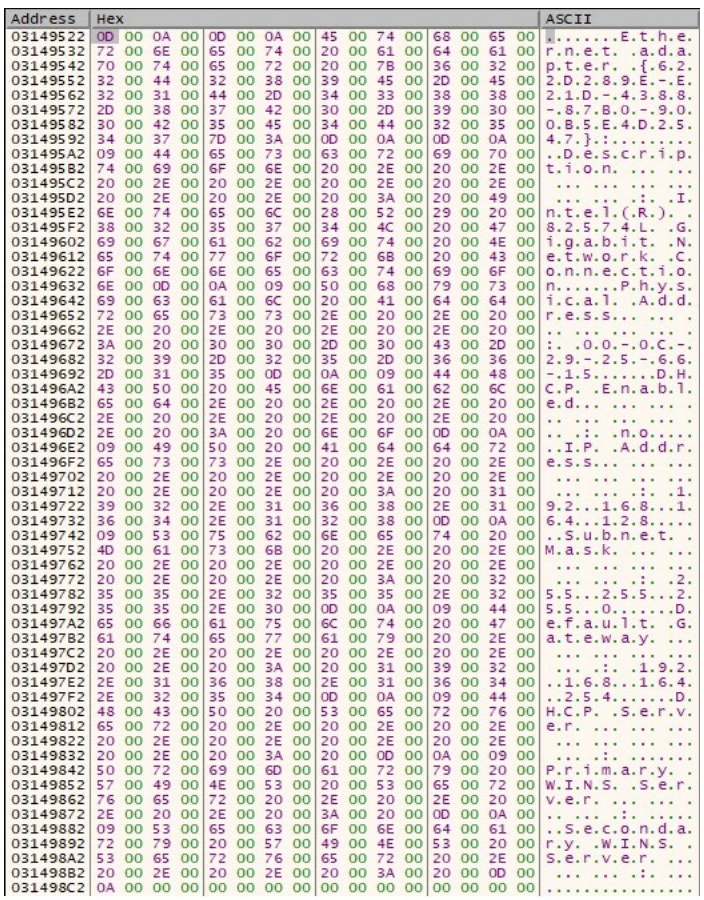


Figure 16

The process gets the username associated with the current thread by calling the GetUserNameW function:

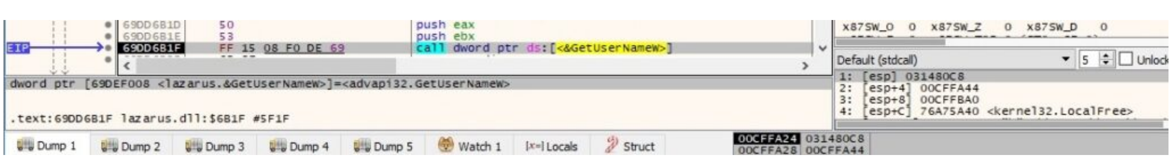


Figure 17

The binary takes a snapshot of all processes in the system using the CreateToolhelp32Snapshot API (0x2 = **TH32CS_SNAPPROCESS**):

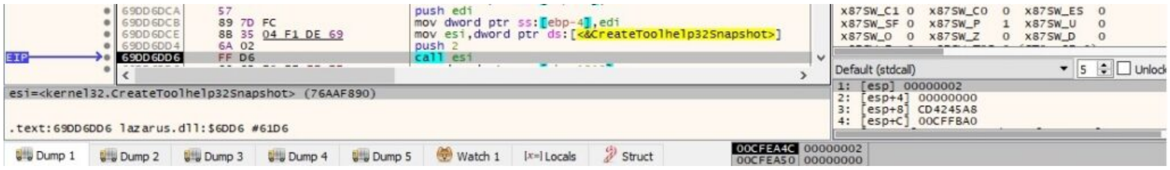


Figure 18

The file extracts information about the first process from the snapshot via a call to Process32FirstW:

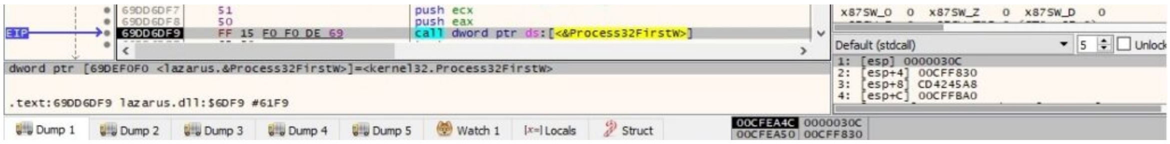


Figure 19

The malicious binary opens the process object using the OpenProcess routine (0x410 = **PROCESS_QUERY_INFORMATION** | **PROCESS_VM_READ**):

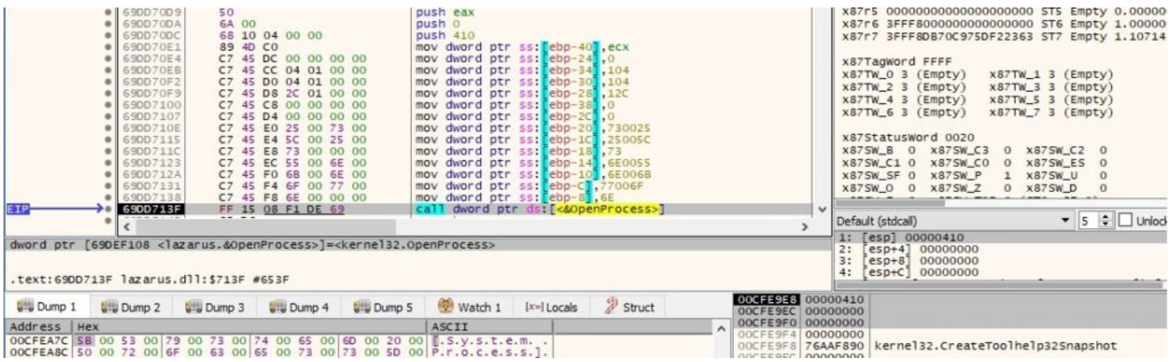


Figure 20

Whether the file doesn't have enough rights to open a process, it copies "Unknown" along with the process name to a temporary buffer.

The binary takes a snapshot of the current process along with all its modules using the CreateToolhelp32Snapshot API (0x8 = **TH32CS_SNAPMODULE**):

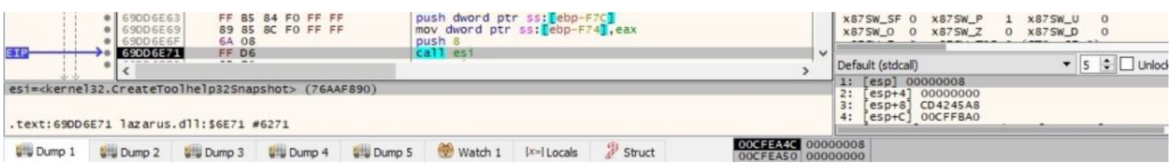


Figure 21

Module32FirstW is utilized to retrieve information about the first module associated with the current process:

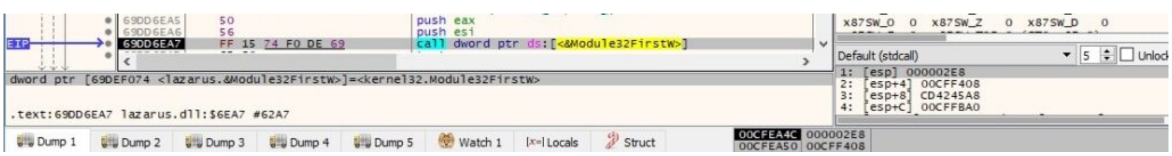


Figure 22

The malicious DLL gets information about the next process recorded in the snapshot:

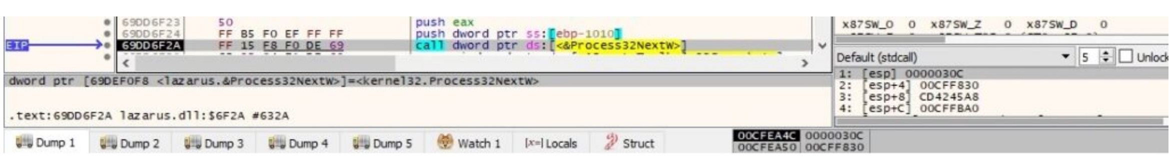


Figure 23

The OpenProcessToken routine is used to open the access token associated with a process (0x8 = **TOKEN_QUERY**):

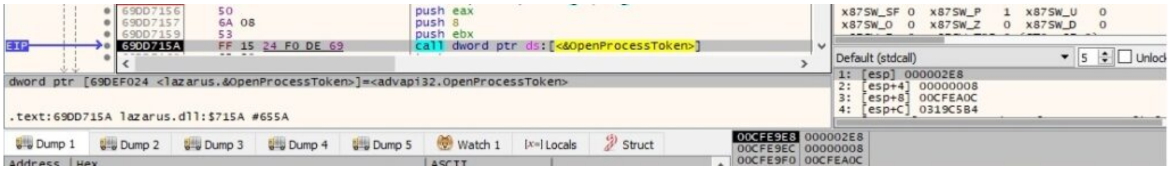


Figure 24

GetTokenInformation is utilized to extract the user account of the token (0x1 = **TokenUser**):

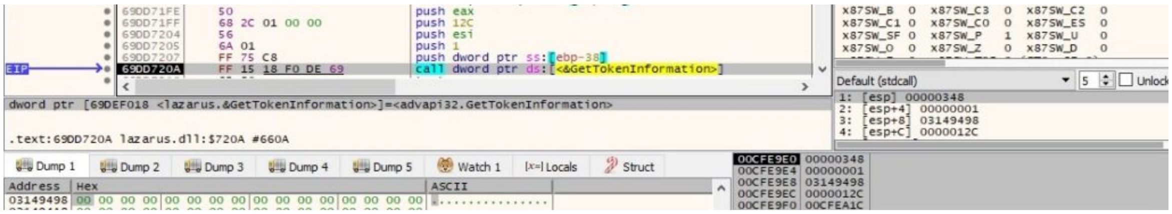


Figure 25

The process retrieves the name of the account for a SID and the name of the first domain on which the SID is found via a function call to LookupAccountSidW:

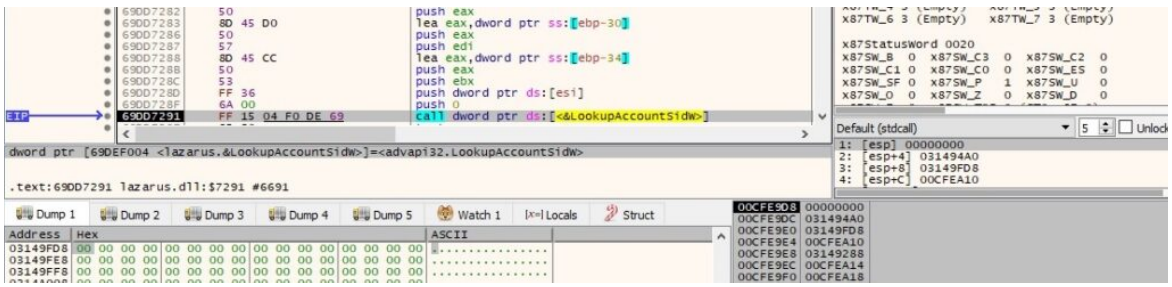


Figure 26

GetTokenInformation is utilized to extract the Terminal Services session identifier associated with the token (0xC = **TokenSessionId**):

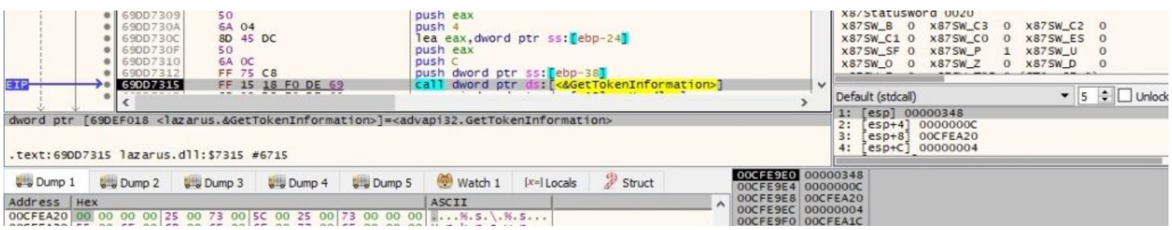


Figure 27

The RtlGetCompressionWorkSpaceSize API is used to determine the correct size of the WorkSpace buffer for the RtlCompressBuffer function (0x102 = **COMPRESSION_FORMAT_LZNT1** | **COMPRESSION_ENGINE_MAXIMUM**):

Figure 28

The process compresses the buffers from figures 15 and 16 using the RtlCompressBuffer function (0x102 = **COMPRESSION_FORMAT_LZNT1** | **COMPRESSION_ENGINE_MAXIMUM**):

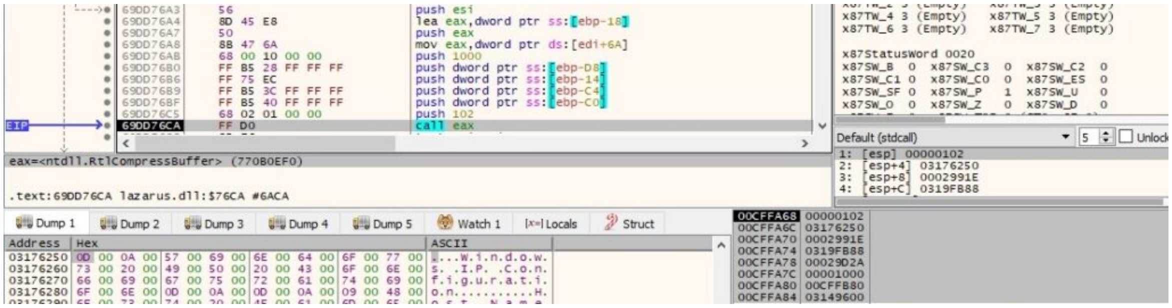


Figure 29

The DLL randomly chooses a C2 server from the list of four. It initializes the application's use of the WinINet functions via a call to InternetOpenW:

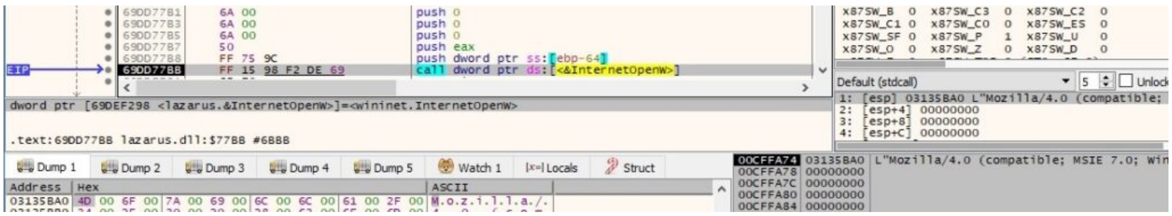


Figure 30

InternetCanonicalizeUrlW is used to canonicalize the URL:

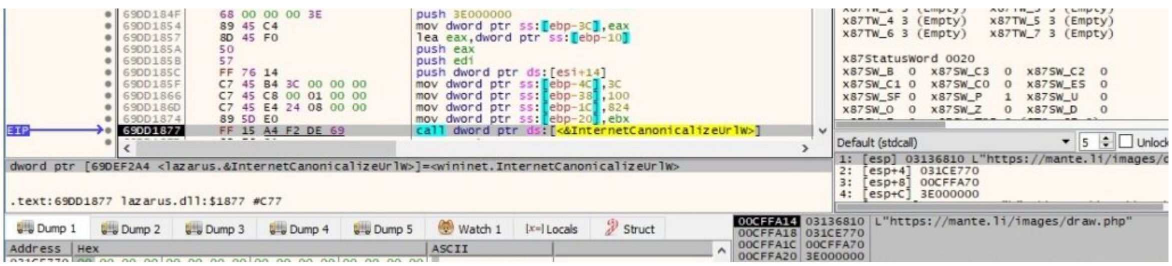


Figure 31

The malware cracks the URL into its component parts by calling the InternetCrackUrlW API:

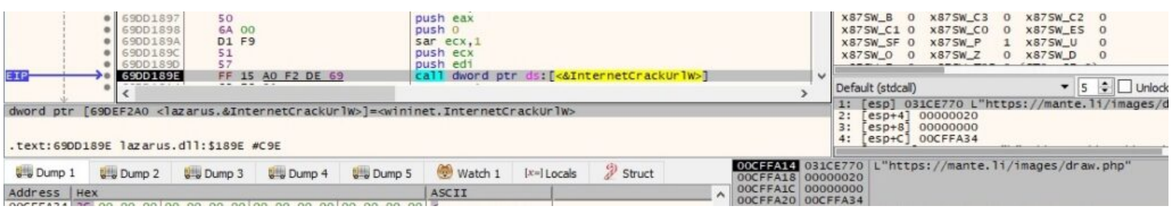


Figure 32

The connect, send and receive timeouts are set to 150s using the InternetSetOptionW routine (0x2 = **INTERNET_OPTION_CONNECT_TIMEOUT**, 0x5 = **INTERNET_OPTION_SEND_TIMEOUT**, 0x6 = **INTERNET_OPTION_RECEIVE_TIMEOUT**):

Figure 33

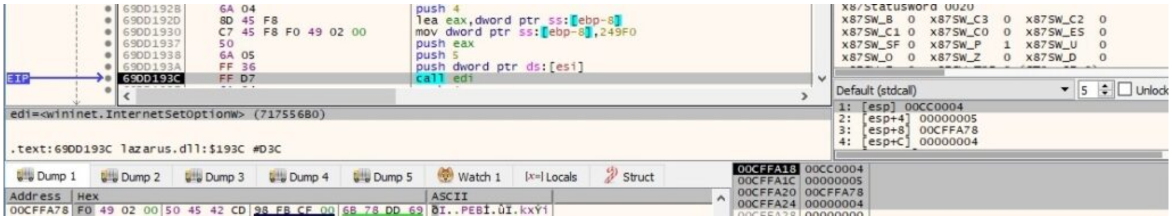


Figure 34

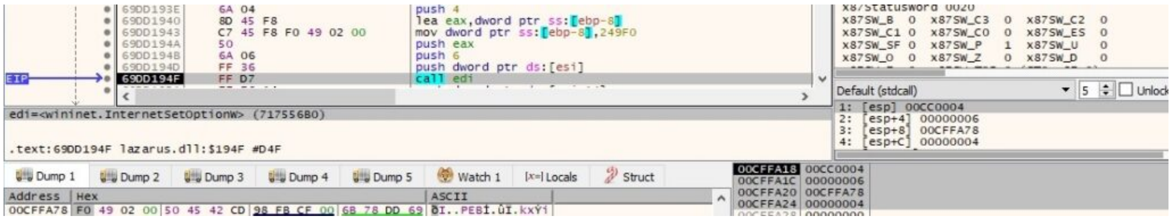


Figure 35

The DLL opens an HTTP session to the C2 server on port 443 (0x3 = **INTERNET_SERVICE_HTTP**):

Figure 36

The binary creates a POST request handle to the URI extracted from the specified URL:

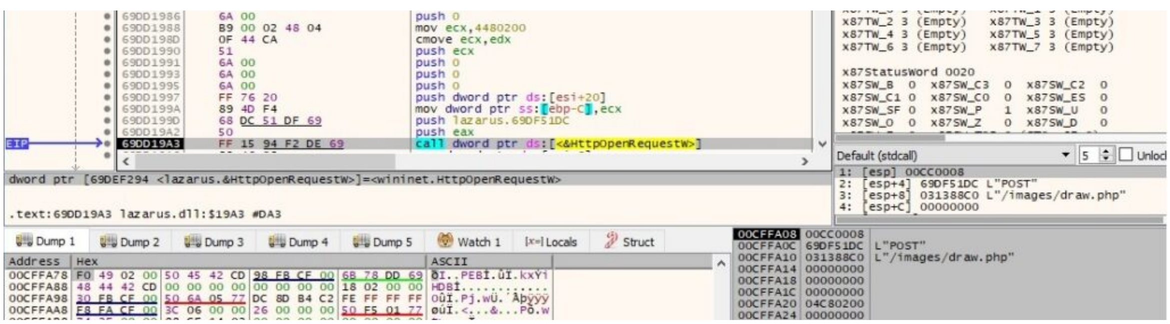


Figure 37

The security flags for the handle are set using the InternetSetOptionW API (0x1F = **INTERNET_OPTION_SECURITY_FLAGS**, 0xF180 = **SECURITY_FLAG_IGNORE_REVOCATION** | **SECURITY_FLAG_IGNORE_UNKNOWN_CA** | **SECURITY_FLAG_IGNORE_CERT_CN_INVALID** | **SECURITY_FLAG_IGNORE_CERT_DATE_INVALID** | **SECURITY_FLAG_IGNORE_REDIRECT_TO_HTTP** | **SECURITY_FLAG_IGNORE_REDIRECT_TO_HTTPS**):

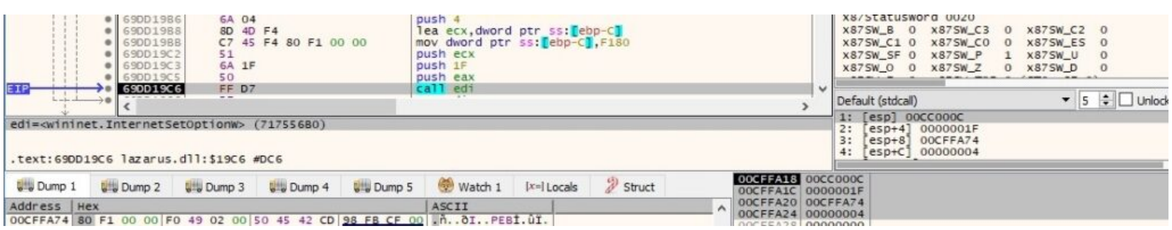


Figure 38

The buffer (concatenation of two buffers) that was compressed earlier is encrypted using XOR (key = 32-byte array):

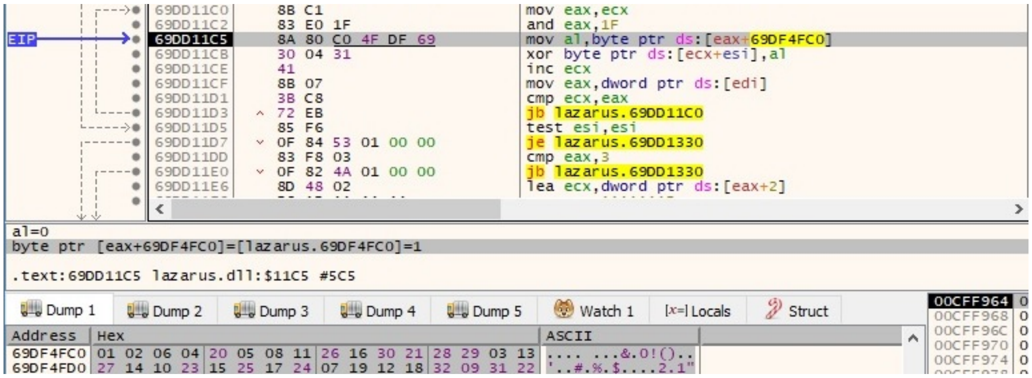


Figure 39

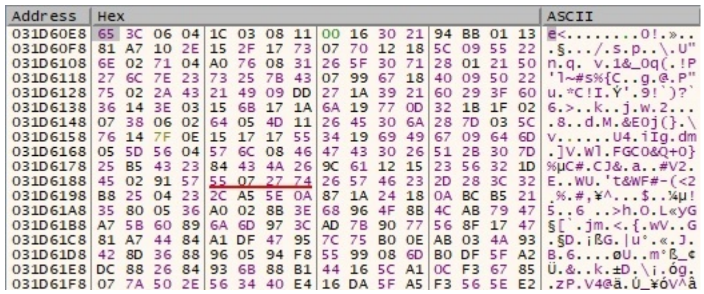


Figure 40

The encrypted buffer from above is encoded using Base64:

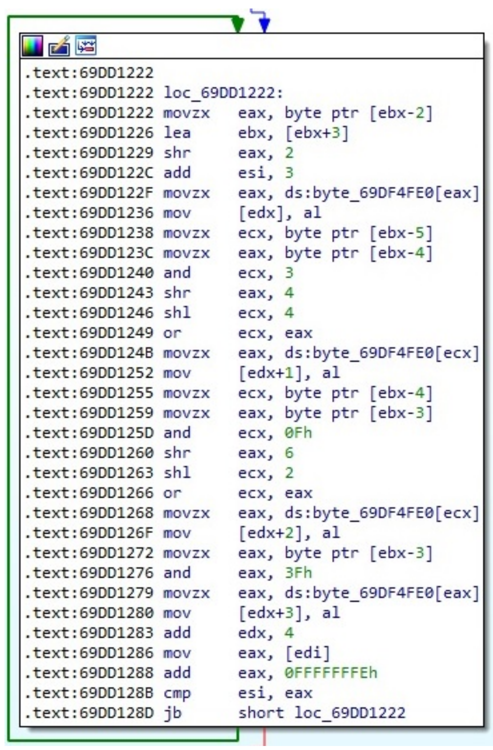
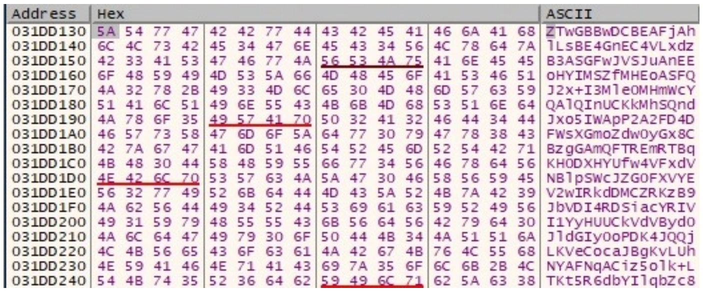


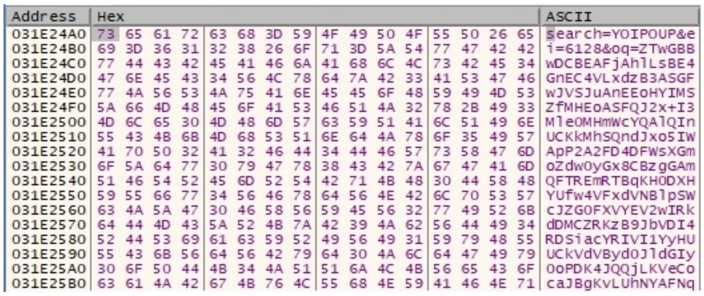
Figure 41



Address	Hex				ASCII
031DD130	5A 54 77 47	42 42 77 44	43 42 45 41	46 6A 41 68	ZTwGBBwDCBEAFjAh
031DD140	6C 4C 73 42	45 34 47 6E	45 43 34 56	4C 78 64 7A	lLsBE4GnEC4VLxdz
031DD150	42 33 41 53	47 46 77 4A	56 53 4A 75	41 6E 45 45	B3ASGFwJVSJuAnEE
031DD160	6F 48 59 49	4D 53 5A 66	4D 48 45 6F	41 53 46 51	oHYIMSZfMHEoASFQ
031DD170	4A 32 78 2B	49 33 4D 6C	65 30 4D 48	6D 57 63 59	J2x+I3Mle0MHmWcY
031DD180	51 41 6C 51	49 6E 55 43	4B 6B 4D 68	53 51 6E 64	QAlQInUCKkMhSQnd
031DD190	4A 78 6F 35	49 57 41 70	50 32 41 32	46 44 34 44	Jxo5IWApP2A2FD4D
031DD1A0	46 57 73 58	47 6D 6F 5A	64 77 30 79	47 78 38 43	FWsXGmoZdw0yGx8C
031DD1B0	42 7A 67 47	41 6D 51 46	54 52 45 6D	52 54 42 71	BzgGAmQFTREmRTBq
031DD1C0	4B 48 30 44	58 48 59 55	66 77 34 56	46 78 64 56	KH0DXHYUfw4VFxdV
031DD1D0	4E 42 6C 70	53 57 63 4A	5A 47 30 46	58 56 59 45	NBlpSWcJZG0FXVYE
031DD1E0	56 32 77 49	52 6B 64 44	4D 43 5A 52	4B 7A 42 39	V2wIRkdDMCZRKzB9
031DD1F0	4A 62 56 44	49 34 52 44	53 69 61 63	59 52 49 56	JbVDI4RDSiacYRIV
031DD200	49 31 59 79	48 55 55 43	6B 56 64 56	42 79 64 30	I1YyHUUCkVdVByd0
031DD210	4A 6C 64 47	49 79 30 6F	50 44 4B 34	4A 51 51 6A	JldGIy0oPDK4JQQj
031DD220	4C 4B 56 65	43 6F 63 61	4A 42 67 4B	76 4C 55 68	LKVeCocaJBgKvLUh
031DD230	4E 59 41 46	4E 71 41 43	69 7A 35 6F	6C 6B 2B 4C	NYAFNqACiz5olk+L
031DD240	54 4B 74 35	52 36 64 62	59 49 6C 71	62 5A 63 38	TKt5R6dbYIlqbZc8

Figure 42

The binary constructs the following parameters “search=YOIPOUP&ei=6128&oq=<Base64-encoded buffer>”:



Address	Hex				ASCII
031E24A0	73 65 61 72	63 68 3D 59	4F 49 50 4F	55 50 26 65	search=YOIPOUP&e
031E24B0	69 3D 36 31	32 38 26 6F	71 3D 5A 54	77 47 42 42	i=6128&oq=ZTwGBB
031E24C0	77 44 43 42	45 41 46 6A	41 68 6C 4C	73 42 45 34	wDCBEAFjAhlLsBE4
031E24D0	47 6E 45 43	34 56 4C 78	64 7A 42 33	41 53 47 46	GnEC4VLxdzB3ASGF
031E24E0	77 4A 56 53	4A 75 41 6E	45 45 6F 48	59 49 4D 53	wJVSJuAnEEoHYIMS
031E24F0	5A 66 4D 48	45 6F 41 53	46 51 4A 32	78 2B 49 33	ZfMHEoASFQJ2x+I3
031E2500	4D 6C 65 30	4D 48 6D 57	63 59 51 41	6C 51 49 6E	Mle0MHmWcYQAlQIn
031E2510	55 43 4B 6B	4D 68 53 51	6E 64 4A 78	6F 35 49 57	UCKkMhSQndJxo5IW
031E2520	41 70 50 32	41 32 46 44	34 44 46 57	73 58 47 6D	ApP2A2FD4DFWsXGm
031E2530	6F 5A 64 77	30 79 47 78	38 43 42 7A	67 47 41 6D	oZdw0yGx8CBzgGAm
031E2540	51 46 54 52	45 6D 52 54	42 71 4B 48	30 44 58 48	QFTREmRTBqKH0DXH
031E2550	59 55 66 77	34 56 46 78	64 56 4E 42	6C 70 53 57	YUfw4VFxdVNBlpSW
031E2560	63 4A 5A 47	30 46 58 56	59 45 56 32	77 49 52 6B	cJZG0FXVYEV2wIRk
031E2570	64 44 4D 43	5A 52 4B 7A	42 39 4A 62	56 44 49 34	dDMCZRKzB9JbVDI4
031E2580	52 44 53 69	61 63 59 52	49 56 49 31	59 79 48 55	RDSiacYRIVI1YyHU
031E2590	55 43 6B 56	64 56 42 79	64 30 4A 6C	64 47 49 79	UCkVdVByd0JldGIy
031E25A0	30 6F 50 44	4B 34 4A 51	51 6A 4C 4B	56 65 43 6F	0oPDK4JQQjLKVeCo
031E25B0	63 61 4A 42	67 4B 76 4C	55 68 4E 59	41 46 4E 71	caJBgKvLUhNYAFNq

Figure 43

The User Agent extracted earlier is added to the HTTP request handle using the HttpAddRequestHeadersW routine (0xA0000000 = **HTTP_ADDREQ_FLAG_REPLACE | HTTP_ADDREQ_FLAG_ADD**):

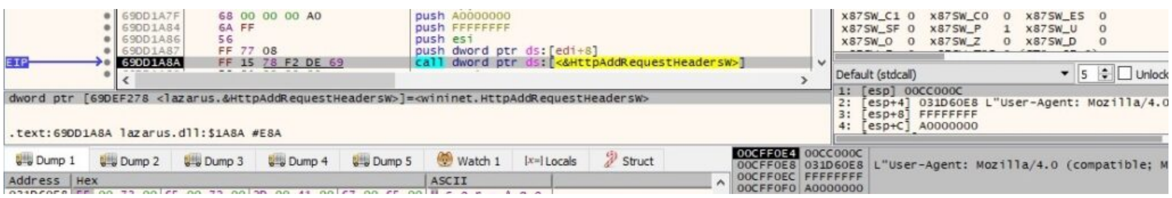


Figure 44

HttpSendRequestW is used to exfiltrate data to the C2 server:

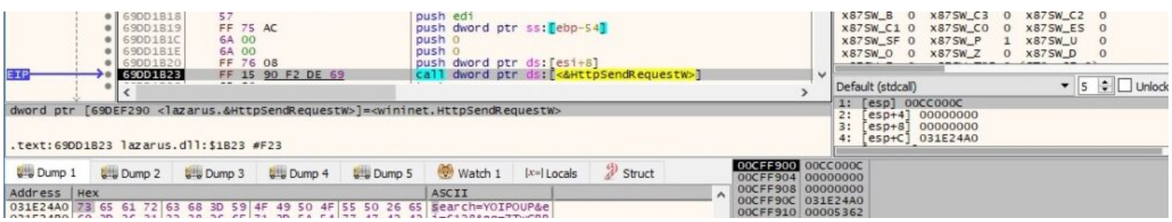


Figure 45

It’s worth mentioning that all C2 servers were down during our analysis. We’ve emulated network connections using FakeNet.

The size of the C2 response is retrieved by calling the HttpQueryInfoW routine (0x5 = **HTTP_QUERY_CONTENT_LENGTH**):

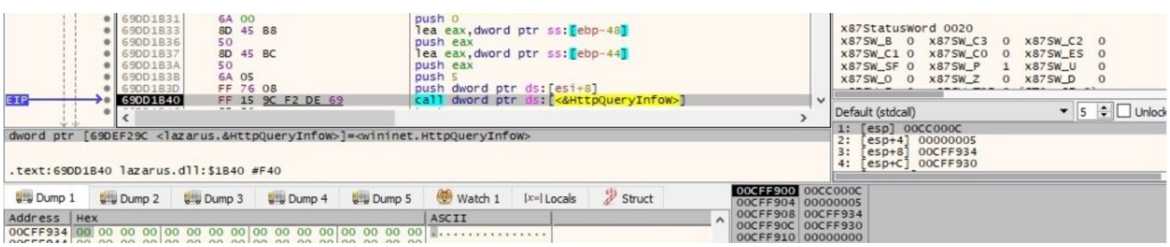


Figure 46

The binary copies the C2 response to a buffer via a function call to InternetReadFile:

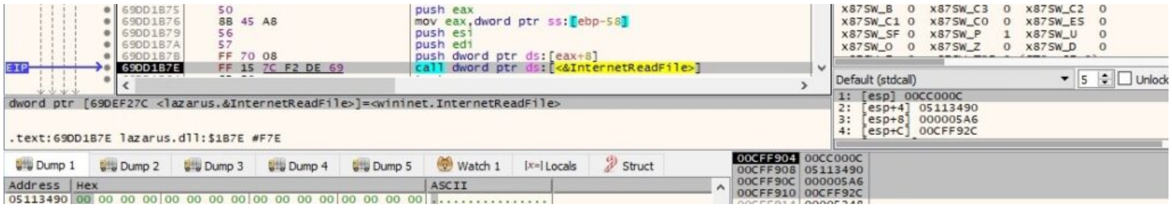


Figure 47

The malicious process parses the data between the “<html></html>” and “<div></div>” tags:

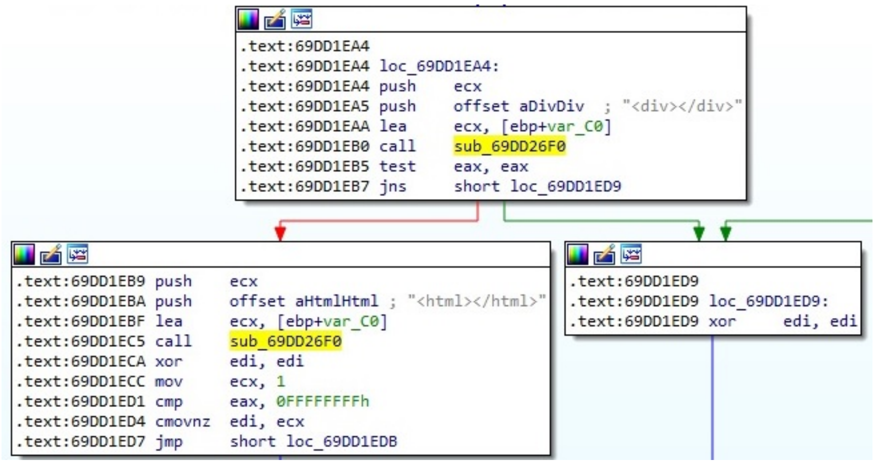


Figure 48

The malware performs a similar POST request with different parameter values “search=DOWPANY&ei=6128”:

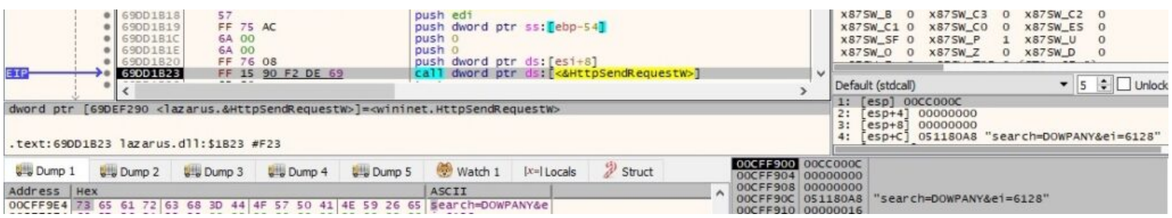


Figure 49

The C2 response is decoded using Base64, and then XOR decrypted. The malware implements 4 different actions that will be explained based on the EAX register value:

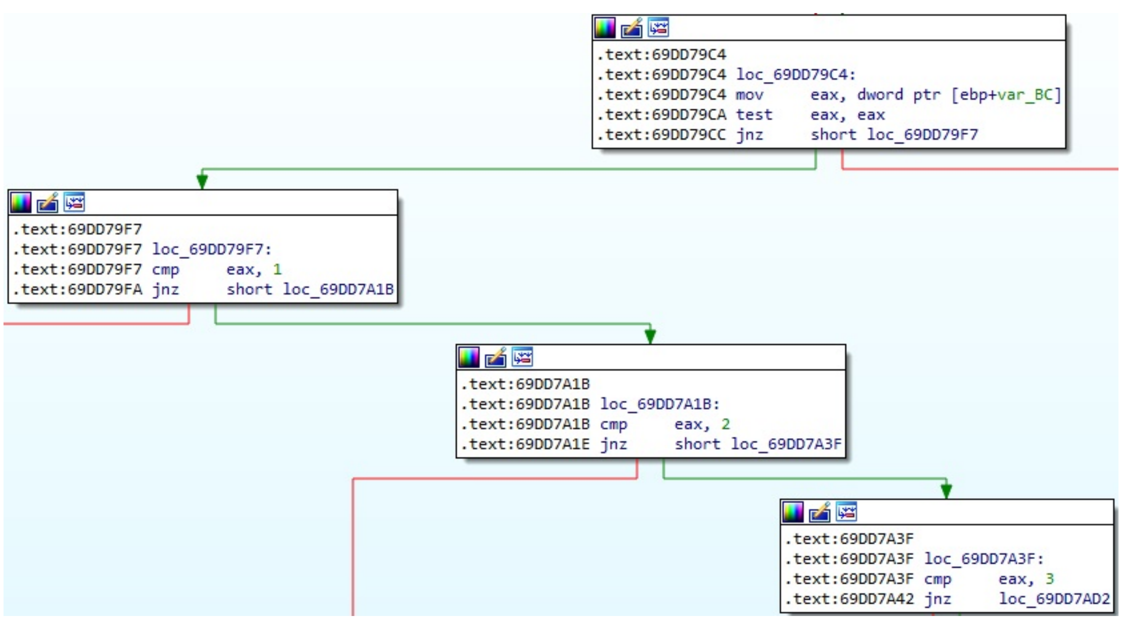


Figure 50

EAX = 0 – load a PE into the current process memory

GetNativeSystemInfo is utilized to retrieve information about the current system:

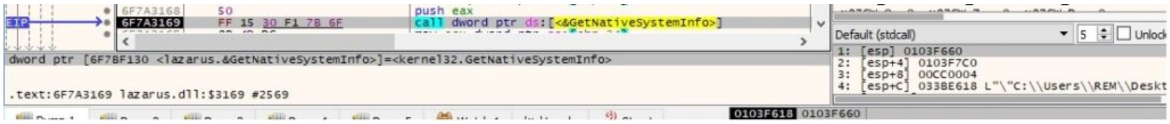


Figure 51

The DLL performs multiple VirtualAlloc function calls that will allocate memory for the new executable (0x3000 = **MEM_COMMIT** | **MEM_RESERVE**, 0x4 = **PAGE_READWRITE**):

Figure 52

The malware changes the memory protection depending on the segment (for example, the code segment's memory protection is set to 0x20 = **PAGE_EXECUTE_READ**):

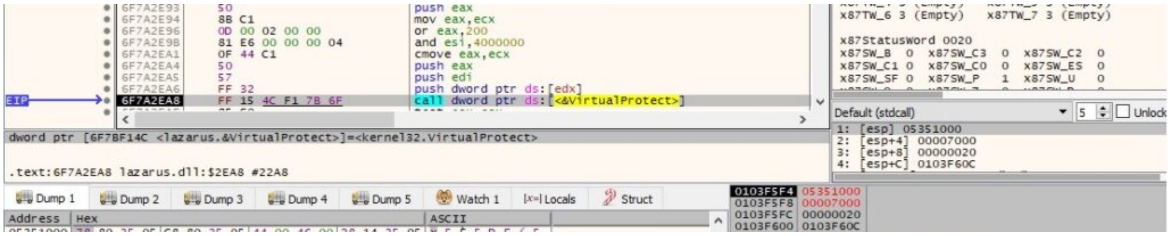


Figure 53

After a few more operations, the process passes the control flow to the new PE.

EAX = 1 – download and execute a .exe file

The binary gets the AppData folder path by calling the SHGetFolderPathW routine (0x1c = **CSIDL_LOCAL_APPDATA**):

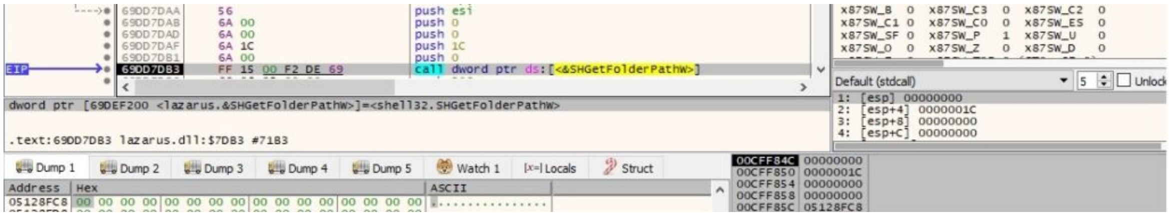


Figure 54

GetTickCount is used to extract the number of milliseconds that have elapsed since the system was started:

Figure 55

The malware creates a file based on the above value (0x40000000 = **GENERIC_WRITE**, 0x1 = **FILE_SHARE_READ**, 0x2 = **CREATE_ALWAYS**, 0x80 = **FILE_ATTRIBUTE_NORMAL**):

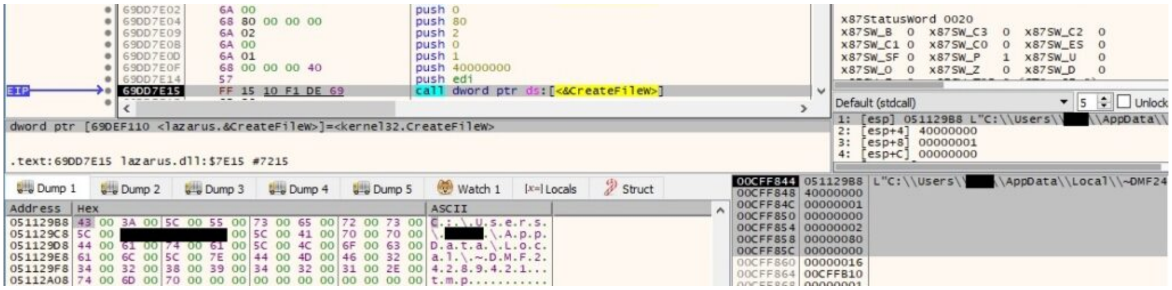


Figure 56

The newly created file is populated with content that is supposed to be transmitted by the C2 server:

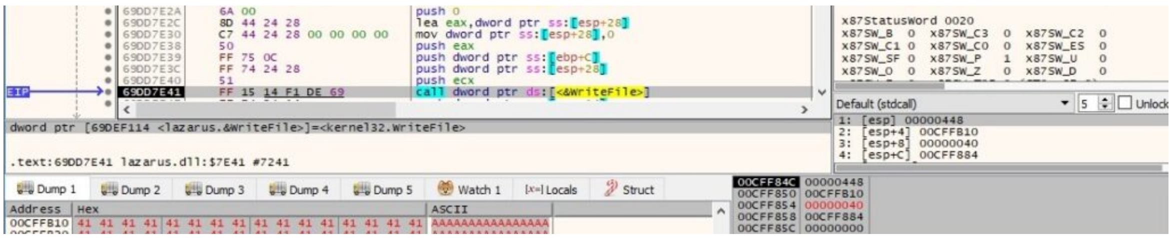


Figure 57

The malicious binary executes the file by calling the CreateProcessW API:

```
.text:69DD7F06
.text:69DD7F06 loc_69DD7F06:
.text:69DD7F06 lea     eax, [esp+90h+ProcessInformation]
.text:69DD7F0A push    eax             ; lpProcessInformation
.text:69DD7F0B lea     eax, [esp+94h+StartupInfo]
.text:69DD7F0F push    eax             ; lpStartupInfo
.text:69DD7F10 push    0               ; lpCurrentDirectory
.text:69DD7F12 push    0               ; lpEnvironment
.text:69DD7F14 push    0               ; dwCreationFlags
.text:69DD7F16 push    0               ; bInheritHandles
.text:69DD7F18 push    0               ; lpThreadAttributes
.text:69DD7F1A push    0               ; lpProcessAttributes
.text:69DD7F1C push    esi             ; lpCommandLine
.text:69DD7F1D push    0               ; lpApplicationName
.text:69DD7F1F call    ds:CreateProcessW
.text:69DD7F25 mov     [esp+90h+var_80], eax
.text:69DD7F29 call    ds:GetLastError
.text:69DD7F2F mov     ecx, [esp+90h+var_78]
.text:69DD7F33 mov     [ecx], eax
.text:69DD7F35 test    esi, esi
.text:69DD7F37 jz      short loc_69DD7F50
```

Figure 58

EAX = 2 – download and execute a .dll file

The execution flow is similar to the above case, and we only highlight the difference. Rundll32.exe is used to execute the DLL file (an export function can also be specified in the command line):

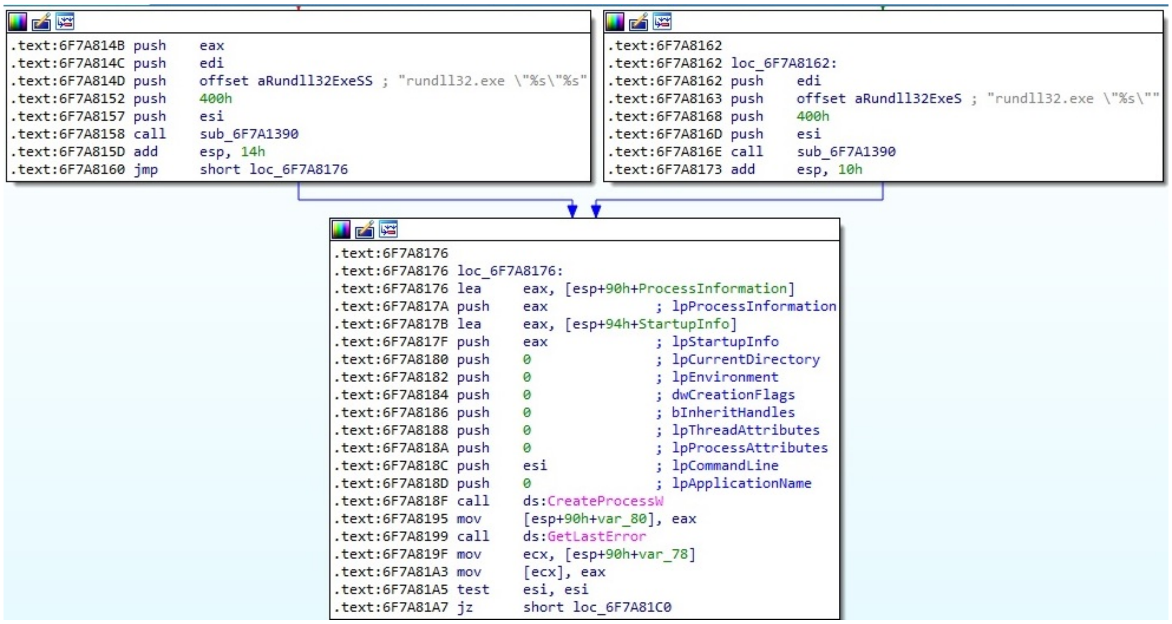


Figure 59

EAX = 3 – copy and execute shellcode

The process allocates memory using the VirtualAlloc routine (0x1000 = **MEM_COMMIT**, 0x40 = **PAGE_EXECUTE_READWRITE**):

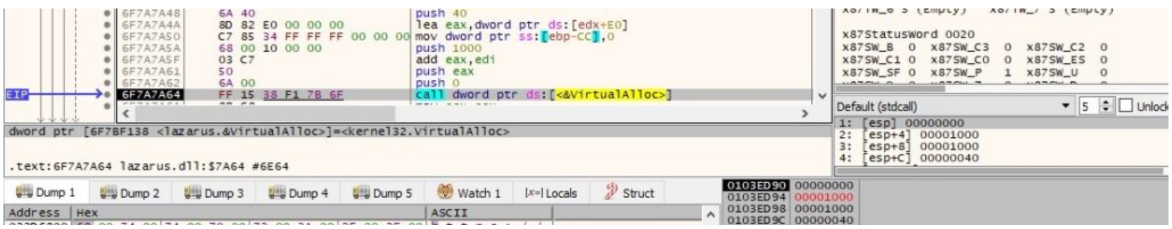


Figure 60

The DLL implements an anti-analysis check. It calls the isProcessorFeaturePresent API in order to determine whether _fastfail() is available. If this feature is not supported, the current process is terminated by calling the GetCurrentProcess and TerminateProcess functions (0x17 = **PF_FASTFAIL_AVAILABLE**):

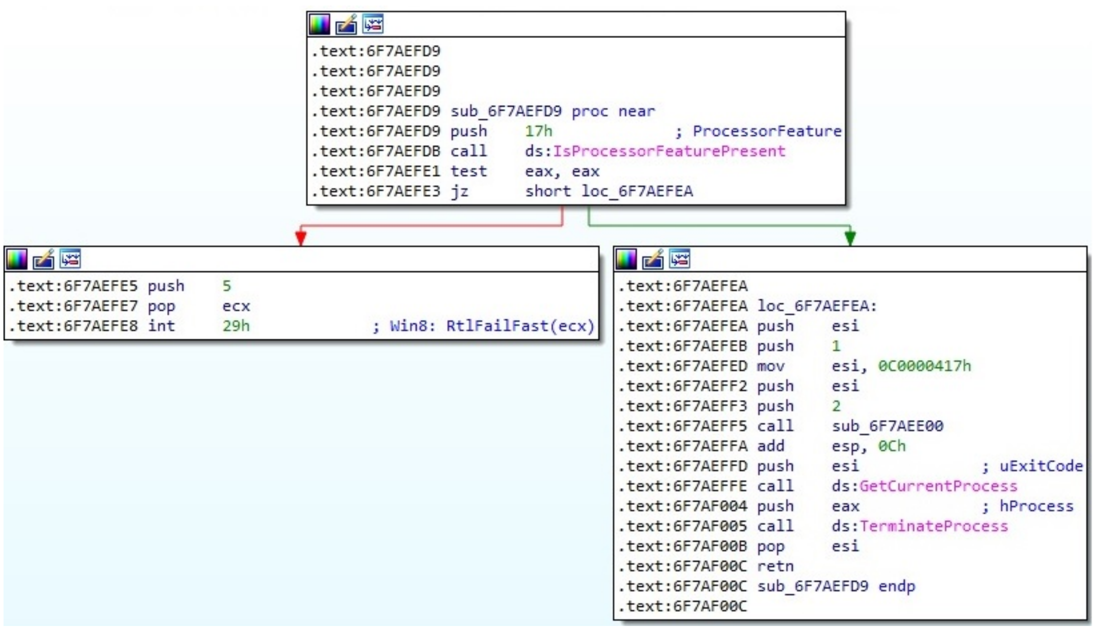


Figure 61

The malware jumps to the shellcode and then frees the memory area allocated earlier:

```
.text:6F7A7A95 call    sub_6F7A3680
.text:6F7A7A9A push    [ebp+Buffer]
.text:6F7A7AA0 call    esi
.text:6F7A7AA2 add     esp, 0Ch
.text:6F7A7AA5 push    8000h           ; dwFreeType
.text:6F7A7AAA push    0               ; dwSize
.text:6F7A7AAC push    [ebp+lpAddress] ; lpAddress
.text:6F7A7AB2 call    ds:VirtualFree
.text:6F7A7AB8 mov     [ebp+var_CC], 1
.text:6F7A7AC2 call    ds:GetLastError
```

Figure 62

As we mentioned at the beginning of the analysis, the threat actor only added the export function explained above, and the others are legitimate.

We've studied a legitimate Notepad++ shell extension (SHA256: f3e2e6f9e7aa065e89040a0c16d1f948489b3751e5eb5efac8106d5f7d65d98d 64-bit) and compared the export functions between the 2 files. As we can see below, the functions are very similar:

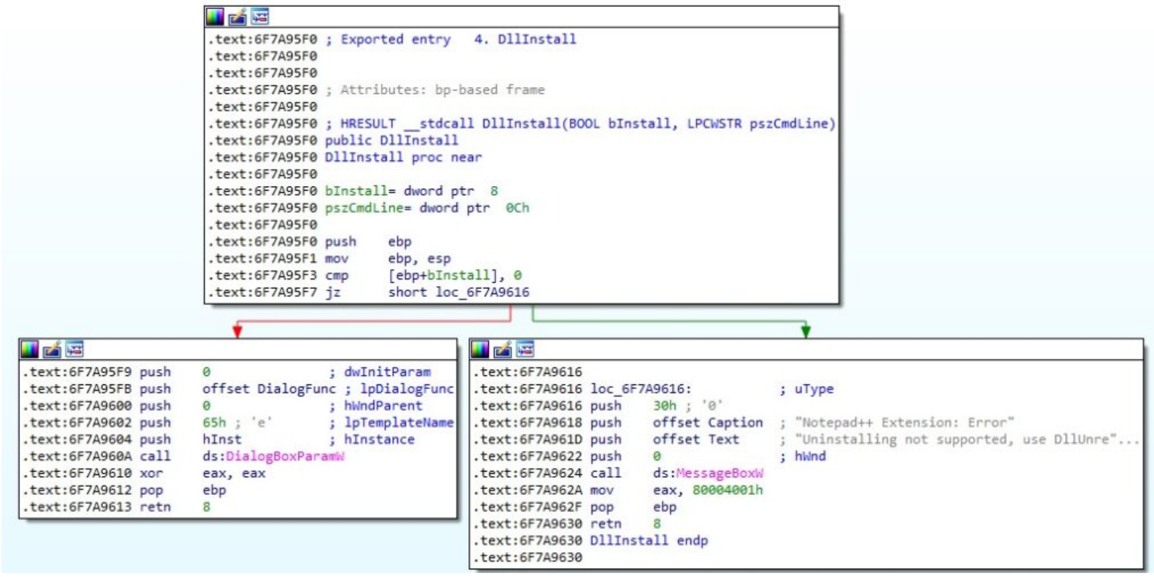


Figure 63

Figure 64

References

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

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

VirusTotal: https://www.virustotal.com/gui/file/803dda6c8dc426f1005acdf765d9ef897dd502cd8a80632eef4738d1d7947269

MalwareBazaar: https://bazaar.abuse.ch/sample/803dda6c8dc426f1005acdf765d9ef897dd502cd8a80632eef4738d1d7947269/

INDICATORS OF COMPROMISE

C2 domains:

mante.li

bmanal.com

shopandtravelusa.com

industryinfostructure.com

SHA256: 803dda6c8dc426f1005acdf765d9ef897dd502cd8a80632eef4738d1d7947269

URLs:

- https[:]//mante.li/images/draw.php
- https[:]//bmanal.com/images/draw.php
- https[:]//shopandtravelusa.com/vendor/monolog/monolog/src/Monolog/monolog.php
- https[:]//industryinfostructure.com/templates/worldgroup/view.php