BabylonRAT-Malware-Analysis/Reports/BabylonRAT-EN.md at main · Yavuzhanzgen/BabylonRAT-Malware-Analysis

By Yavuzhanzgen

Archived: 2026-09-15 02:00:46 UTC

> **Legal Disclaimer**
>
> This repository is intended solely for malware analysis, reverse engineering research, and educational purposes. All analyses were conducted in an isolated laboratory environment.


MALWARE CLASSIFICATION
REMOTE ACCESS TROJAN
PLATFORM
WINDOWS
COMMUNICATION
HTTP / HTTPS
CAPABILITIES
C2
SCREEN CAPTURE
FILE OPERATIONS
PROCESS MANAGEMENT
PERSISTENCE
ANTIANALYSIS
PE HEADER (HEX)
4D 5A 90 00 03 00 00 00
B8 00 00 00 00 00 00 00
00 00 00 00 40 00 00 00
00 00 00 00 00 00 00 00
E0 00 0F 01 0B 01 09 00
PE32+
WINDOWS EXECUTABLE
C2 INFRASTRUCTURE
ACTIVE NODE
POTENTIAL NODE
CONNECTION
BABYLONRAT
TECHNICAL ANALYSIS REPORT
FAMILY BABYLONRAT
TYPE REMOTE ACCESS TROJAN
PLATFORM WINDOWS
ANALYSIS STATIC / DYNAMIC
REPORT TECHNICAL ANALYSIS
CASE ID
BR-2026-001
REPORT VERSION
1.0
DATE
24 MAY 2026

BabylonRAT Technical Analysis Report

Static Analysis • Dynamic Analysis • Reverse Engineering • YARA • MITRE ATT&CK

Overview

Babylon RAT is an open-source Remote Access Trojan (RAT) family that provides various capabilities for remote access, information collection, and data theft on compromised systems.

The malware has been observed in attack campaigns conducted during 2023–2024, including operations targeting public institutions and political figures in Malaysia.

Its capabilities include system and user information collection, keylogging, clipboard monitoring, credential and browser data theft, file and process management, remote command execution, and C2 (Command and Control) communication. Some variants have also been reported to include network propagation, SOCKS proxy functionality, and DDoS capabilities.

Firefox.exe Analysis

Name	Firefox
MD5	72999C0DA0454511B066547206ED7A58
SHA256	2f61950c3baa58573fac514205cead5afd6e27e84bb2b91b233a9f045d9b3f4c
File Type	PE32/EXE

Static Analysis

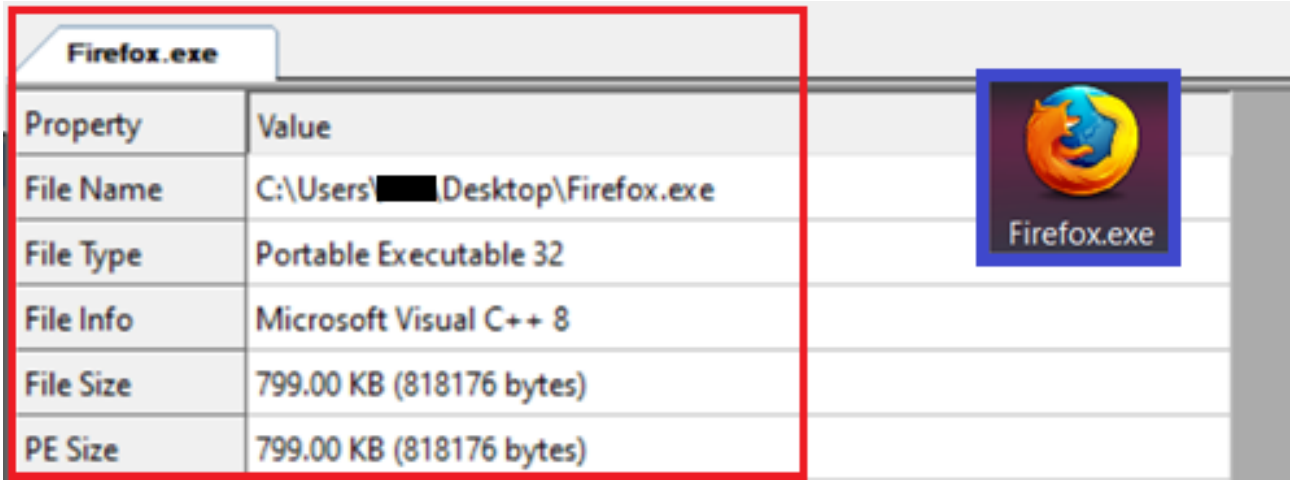


Figure 2. Basic File Properties of the Malware

The malware is identified as a **PE32** executable with an approximate file size of **799 KB**. Based on the filename and icon, the sample is designed to impersonate the legitimate **Mozilla Firefox** browser. This type of masquerading is commonly used as a social engineering technique to make the file appear trustworthy and increase the likelihood of execution.

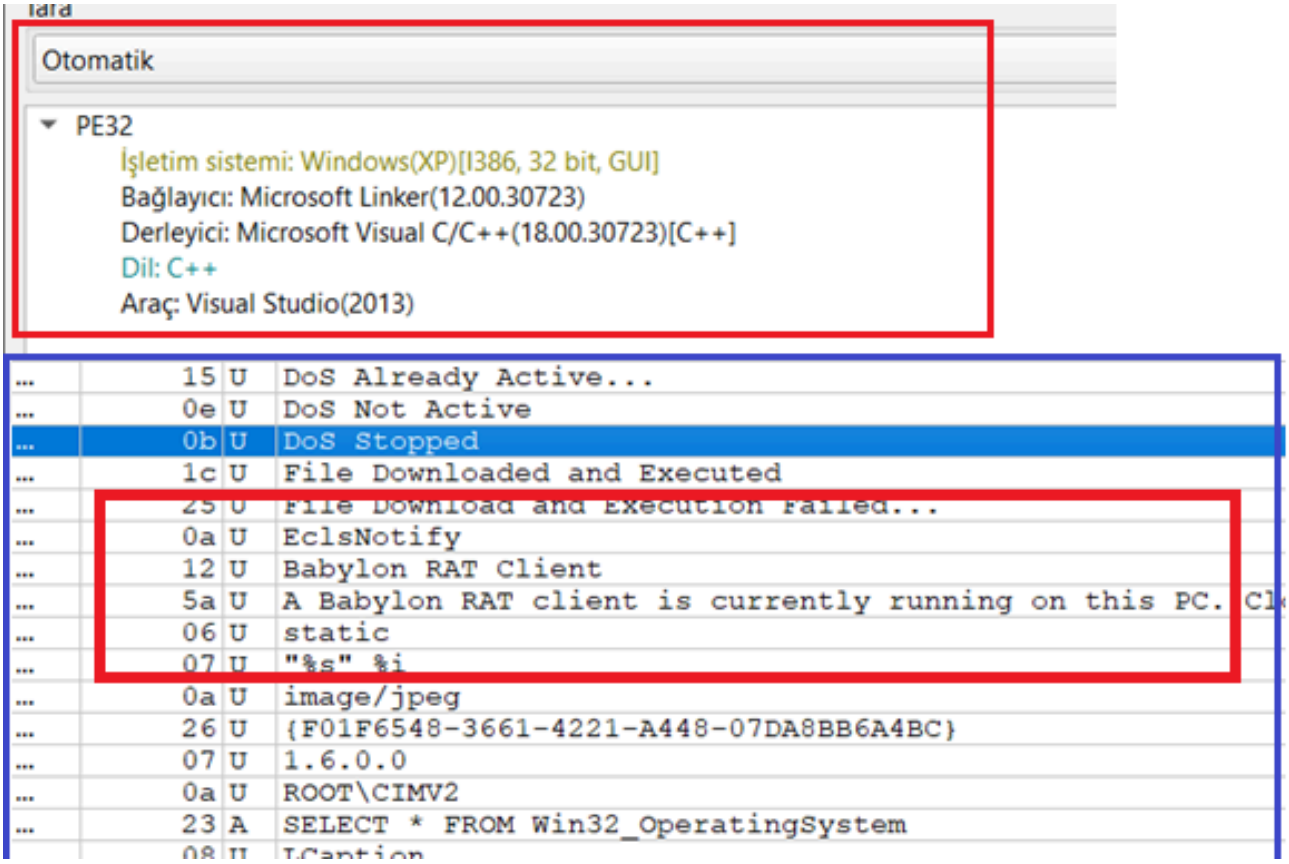


Figure 3. Strings Found Within the Malware

Based on the compilation information, the file appears to have been developed targeting **Windows XP 32-bit** systems. Analysis of the strings contained within the file reveals notable entries such as **"Babylon RAT Client", "Babylon RAT client is currently running on this PC", and "File Downloaded and Executed"**. These strings indicate that the malware is designed as a **Remote Access Trojan (RAT)** and possesses remote management capabilities such as downloading and executing files.

Dynamic Analysis



Figure 4. Dynamic API Resolution

Instead of using a **static import table**, the malware dynamically resolves Windows API functions at runtime through **GetProcAddress**, implementing a dynamic API resolver mechanism. During the resolution process, API names are provided as strings through the stack, and each API pointer is stored in a global table. These pointers are subsequently passed through an **XOR-based obfuscation layer** before being stored.



Figure 5. GetFileAttributesW API

The malware uses the **GetFileAttributesW** API to check the existence and attribute information of a file.

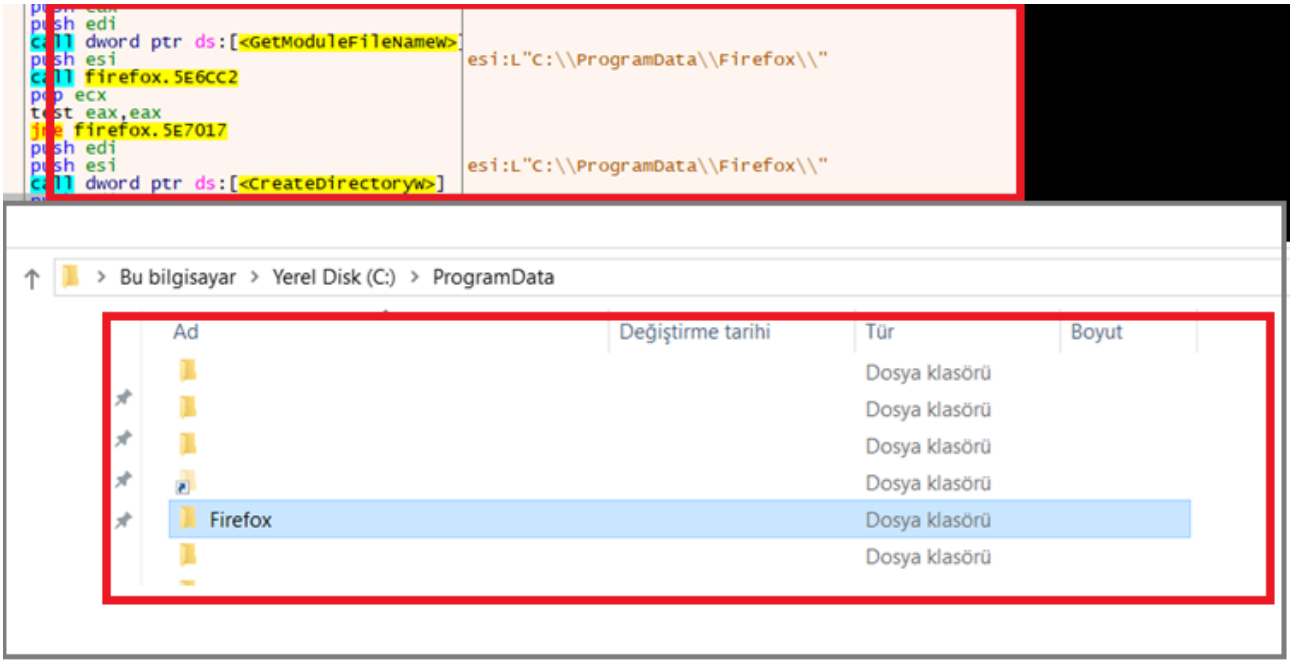


Figure 6. Creation of the Firefox Directory

The **CreateDirectoryW** API is used to create a **Firefox** directory under **ProgramData**.

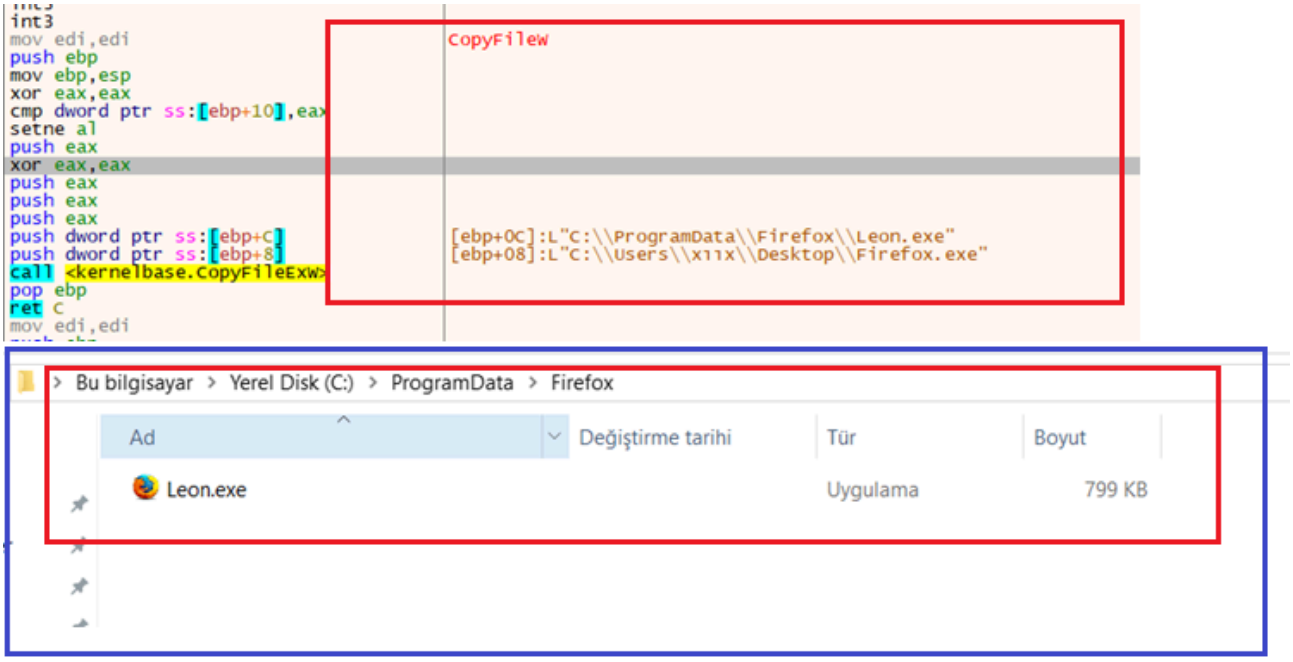


Figure 7. CopyFileW API Call

During the subsequent execution flow, the malware uses the **CopyFileW** API to copy its own executable to the target directory. The original file path is used as the source, and a new copy named **Leon.exe** is created under the previously created fake **“Firefox”** directory.

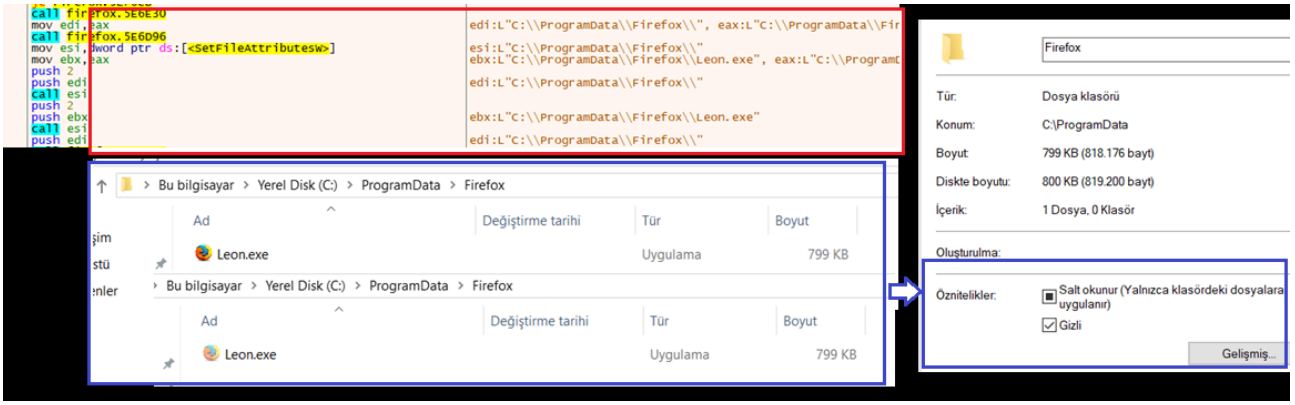


Figure 8. Hiding the Directory and File

The malware uses the **SetFileAttributesW** API to apply the **FILE_ATTRIBUTE_HIDDEN (0x2)** attribute to both the target directory **(C:\ProgramData\Firefox)** and the copied executable file **(Leon.exe)**. As a result, the directory and file are hidden from Windows Explorer, reducing the likelihood of being noticed by the user.

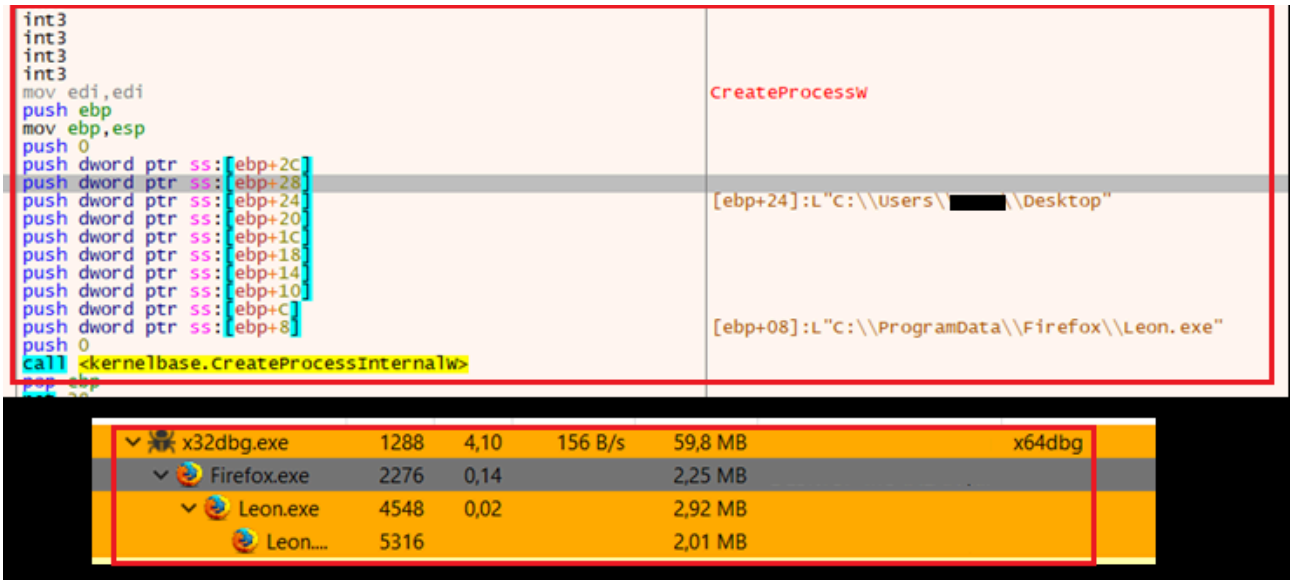


Figure 9. Execution of Leon.exe Under the Firefox.exe Process

During the execution flow, the malware uses the **CreateProcessW** API to launch the target executable. Analysis of the code shows that the same function is called using two different paths. The first call uses an executable path obtained from the current location, while the second call is performed using the previously created persistent path **C:\ProgramData\Firefox\Leon.exe**.

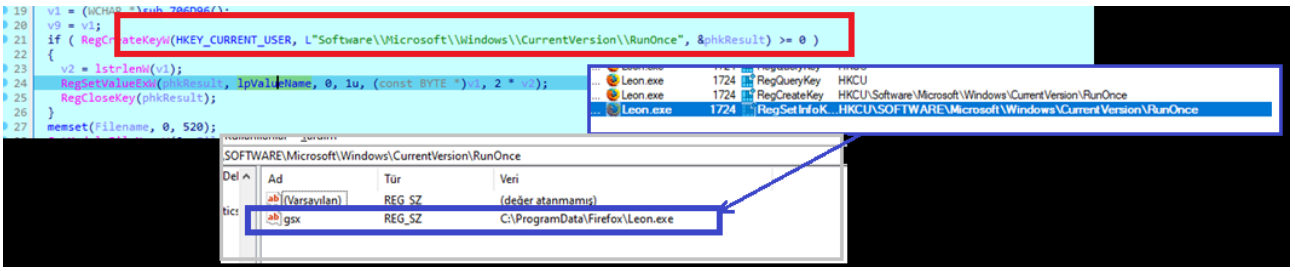


Figure 10. Registry Operation

During the subsequent execution phase, the malware writes the path of **Leon.exe** to the **RunOnce** registry key using the value name **"gsx"**. This ensures that **Leon.exe** is automatically executed during the next user logon.

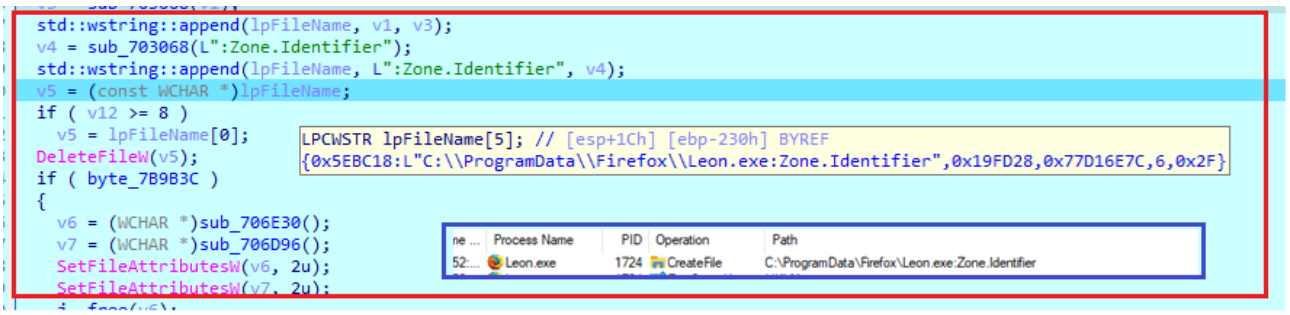


Figure 11. Removal of Zone.Identifier Data

The malware reads its own file and removes the **Zone.Identifier** data associated with it. This behavior is intended to remove the Mark-of-the-Web information associated with files downloaded from the Internet, potentially reducing security warnings and related detection mechanisms.



Figure 12. Mutex Creation

The malware creates a mutex using the **CreateMutexW** API to prevent multiple instances of itself from running simultaneously.

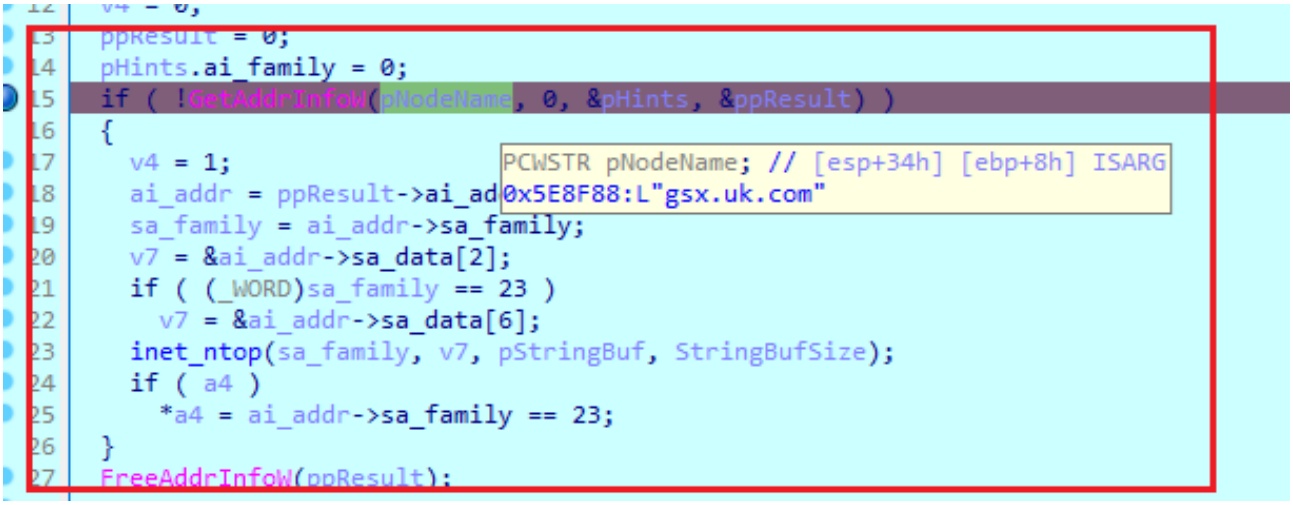


Figure 13. C2 Server Address Used by the Malware

The malware attempts to connect to the dynamically resolved **gsx[.]uk[.]com** server. Since the original server was unavailable, the malware remains in a loop waiting for a response and commands from the server. To continue the analysis, the relevant address was redirected to a local analysis server, allowing the program's communication flow to be observed.



Figure 14. Directory Creation Through a Thread

The malware starts a thread using **CreateThread**. The purpose of this thread is to create the **ConfigEx** directory under **AppData\Roaming** by calling the **CreateDirectoryW** API through the associated function.

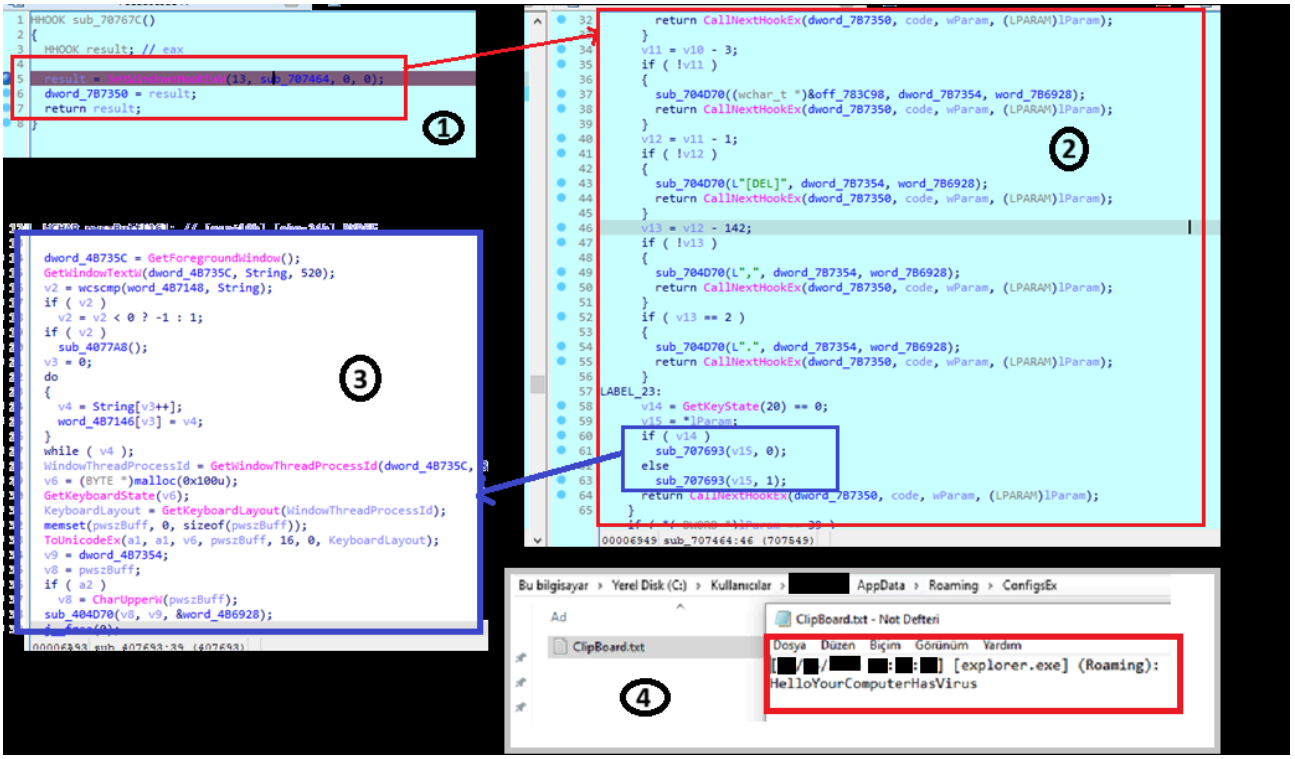


Figure 15. Keyboard Hook Installation and Keystroke Capture

The malware uses the **SetWindowsHookEx** API to install a low-level **keyboard hook** system-wide. The hook function captures user keystrokes, converts virtual-key codes into characters, processes special keys, and stores the collected data in **ClipBoard.txt**.



Figure 16. Operating System Information Collection via WMI

The program queries the **Win32_OperatingSystem** class through **WMI** and retrieves the operating system name from the **Caption** field.

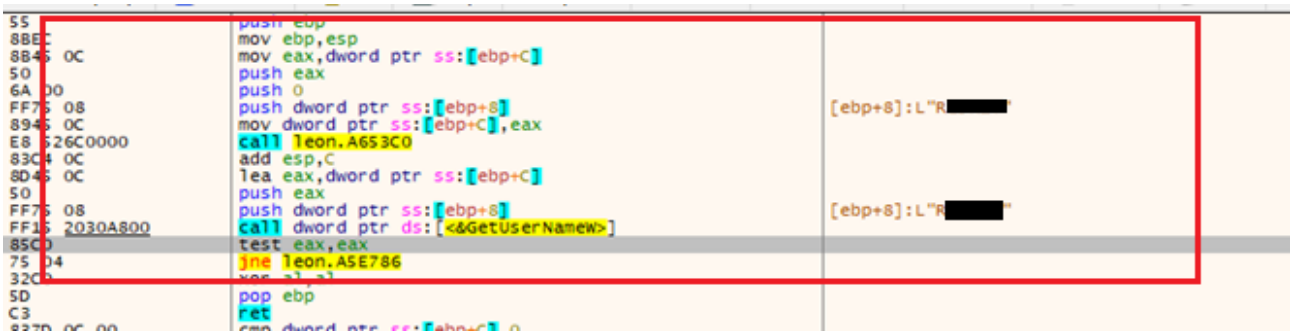


Figure 17. GetUserName API Call

The malware uses the **GetUserName** API to retrieve the username of the current system user.

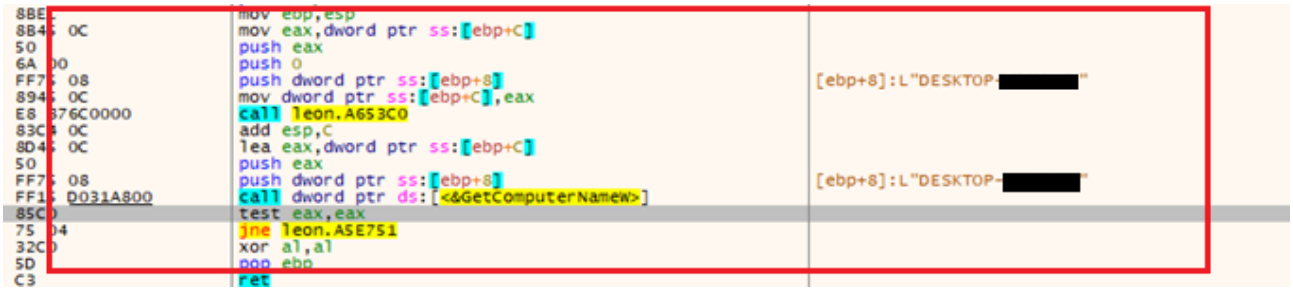


Figure 18. GetComputerNameW API Call

The **GetComputerNameW** API is used to retrieve the computer name.

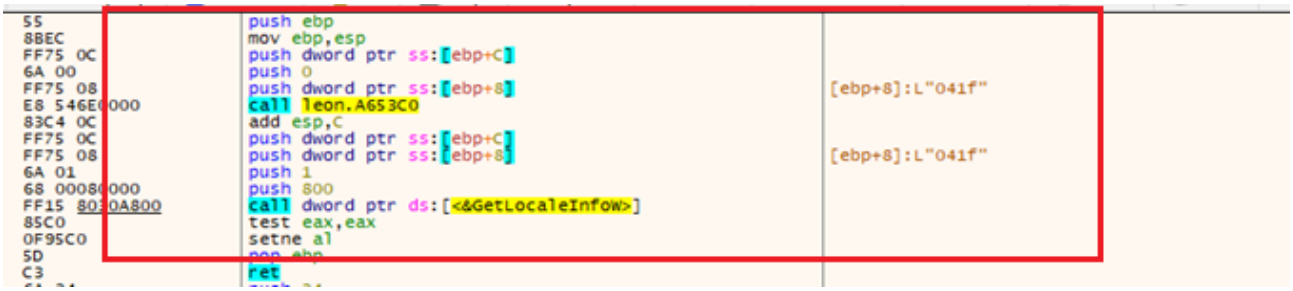


Figure 19. GetLocaleInfoW API Call

The **GetLocaleInfoW** API is used to retrieve the system locale and language information.

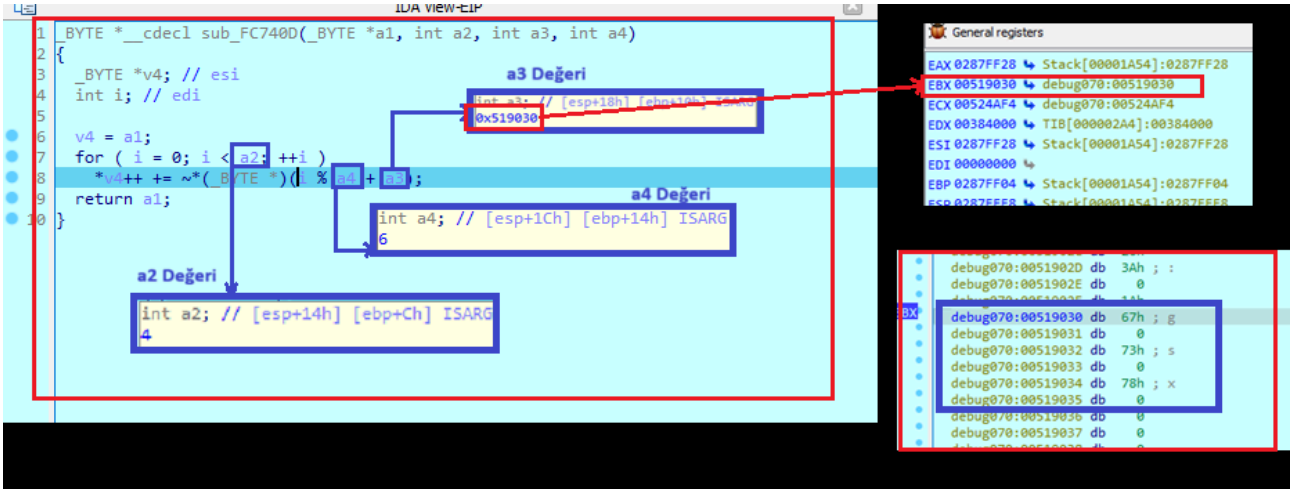


Figure 20. Data Encryption Routine

The **sub_40740D** function performs a simple byte-level transformation on the provided data buffer. For each byte within the length specified by the **a2** parameter, the function adds the bitwise-NOT (`~`) of the corresponding key byte to the current data byte. The key is retrieved from the address specified by **a3** and used cyclically according to the key length **a4** through the **i % a4** operation.

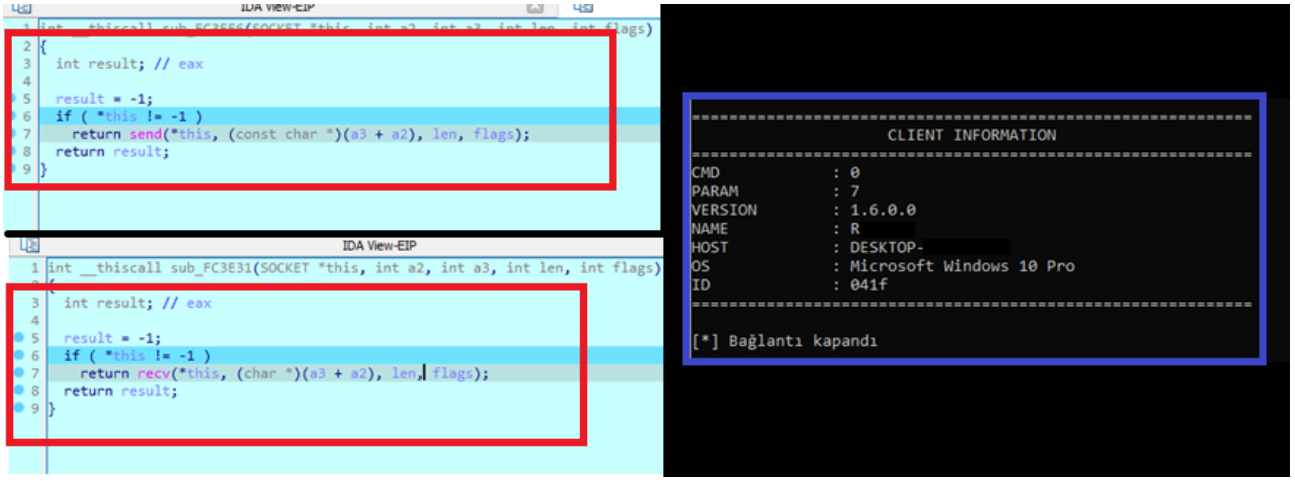


Figure 21. Sending Data

The malware transmits the previously collected system name and system information to the server.

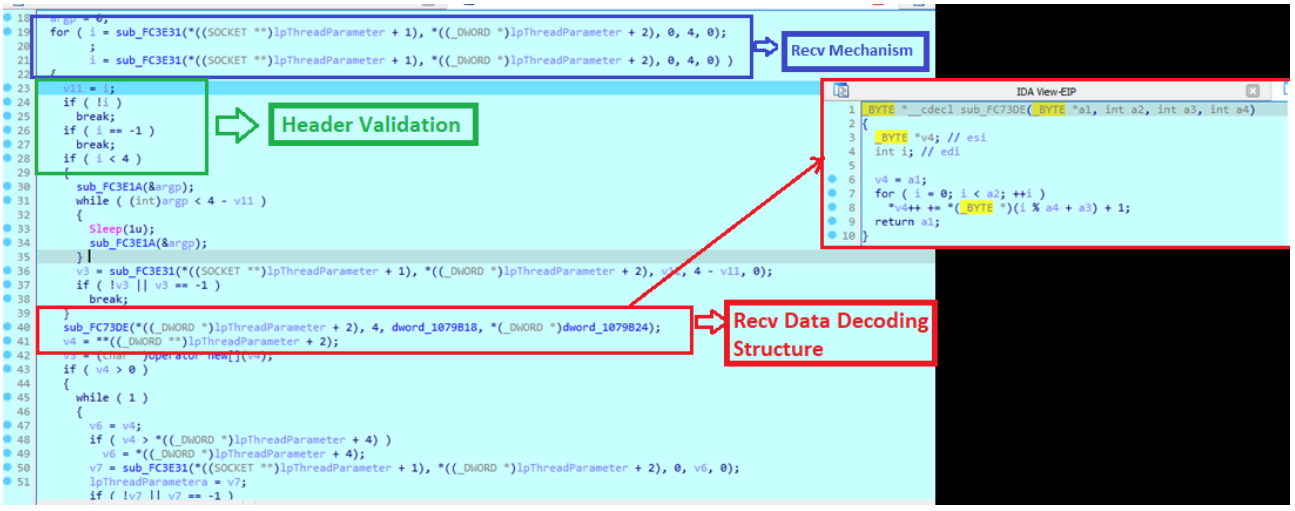


Figure 22. Decoder Structure

sub_FC3E31 invokes **recv()** to receive **4 bytes** of data. If the received data is incomplete, the function continues reading by waiting for short intervals until the required number of bytes has been received.

The received **4-byte** segment is passed to the **sub_FC73DE** function, where it undergoes a byte-level decoding operation. For each byte, a cyclic key is used and the **data += key + 1** operation is applied. The decoded 4 bytes are then used to determine the length of the payload to be received.

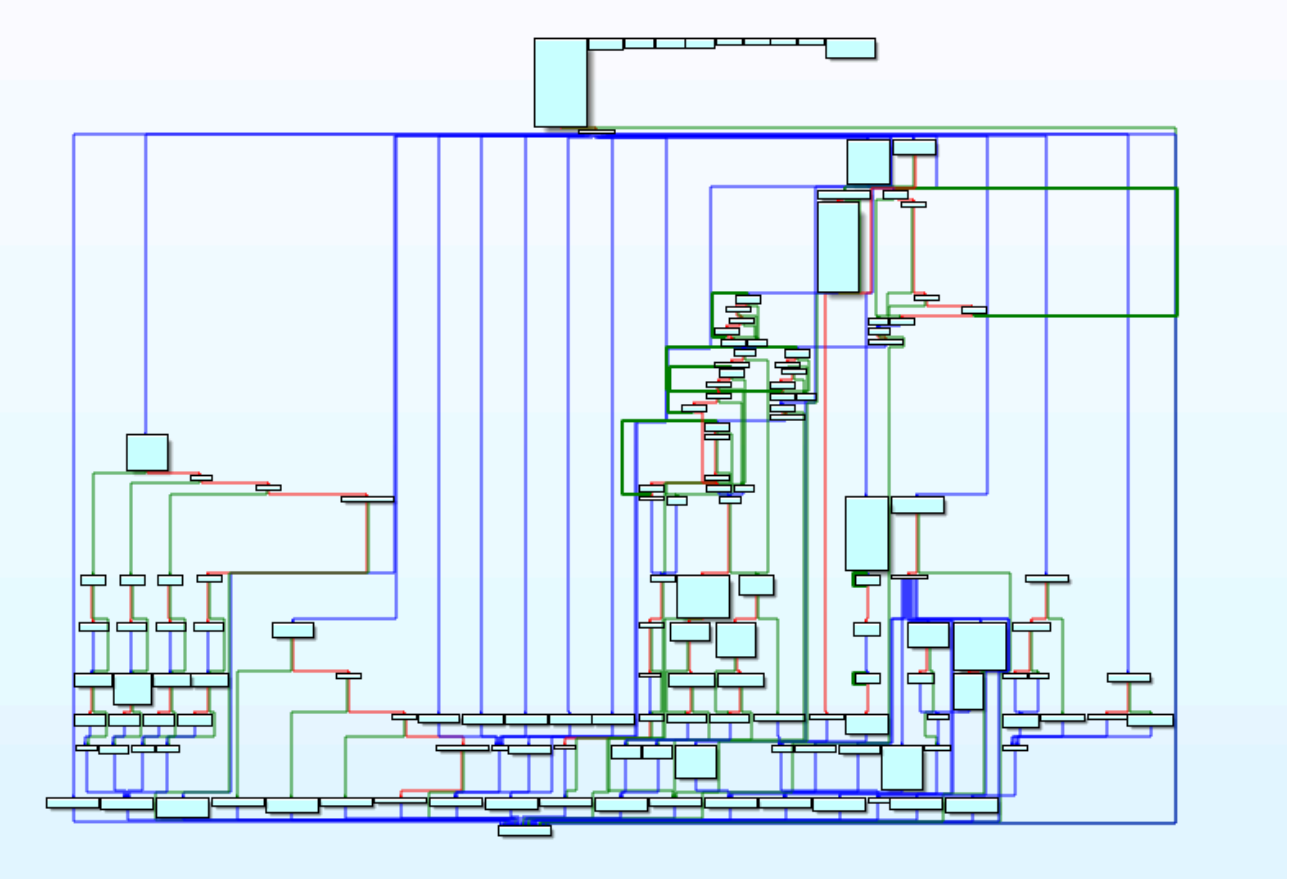


Figure 23. Malicious Command Section

The malware uses a centralized command manager containing **20** different commands to process commands received from the **C2** server. The corresponding case block is executed based on the **Command ID** value received from the server.

Command	Data	Sub-Parameter	Command	Data	Sub-Parameter
1	ScreenShot	-	11	Self Update	-
2	File Manager	-	12	Connection Shutdown	-
3	MessageBox	-	13	File Download	-
4	Keylogger	-	14	DoS Attack	-
5	Monitoring	1,2,3,4	15	Stop DoS Attack	-
6	Monitoring Stop	1,2,3,4	16	Browser Credential Theft	-
8	System Control	0,1,2,3,4,5,6,7,8	18	Remote Shell	-

Command	Data	Sub-Parameter	Command	Data	Sub-Parameter
9	Display Enumeration	-	19	Hosts File Management	-
10	Relaunch	-	20	Information Collection	-

Table 1 — Malware Control Commands

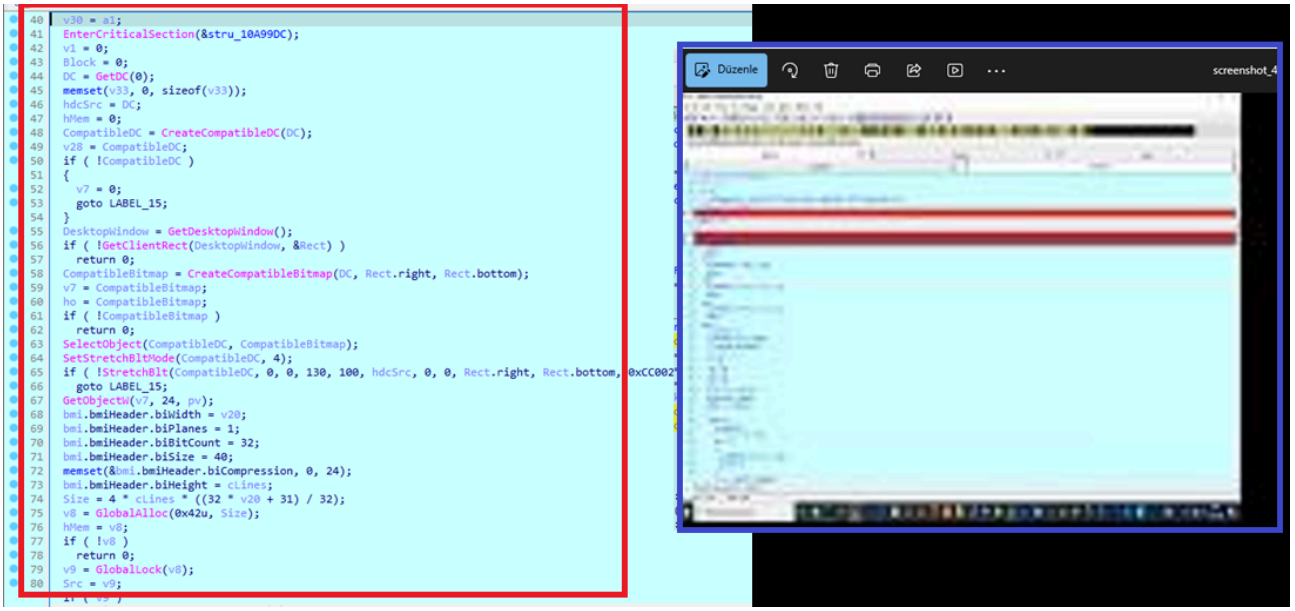


Figure 24. Capturing a Screenshot

When the **v40** value is set to **1**, the corresponding **case 1** condition is executed. In this case, the **sub_45917F()** function is called to capture a screenshot of the system. The function uses the **GetDC** and **CreateCompatibleDC** APIs to capture the screen and then uses **StretchBlt** to resize the image to **130×100 pixels**.



Figure 25. System Drive Discovery

When the **case 2** condition is triggered, the malware first prepares a directory structure similar to **%APPDATA%\Microsoft\Configs//** and a file path named **sqlite3.dll**, then checks the existence and size of the specified file. In addition, **GetLogicalDrives()** is used to enumerate the available drives from **A:–Z:**. Various information is collected for each drive, and **GetDriveTypeW()** is used to classify the drives into types such as fixed disk, network drive, CD/DVD drive, RAM disk, and others.

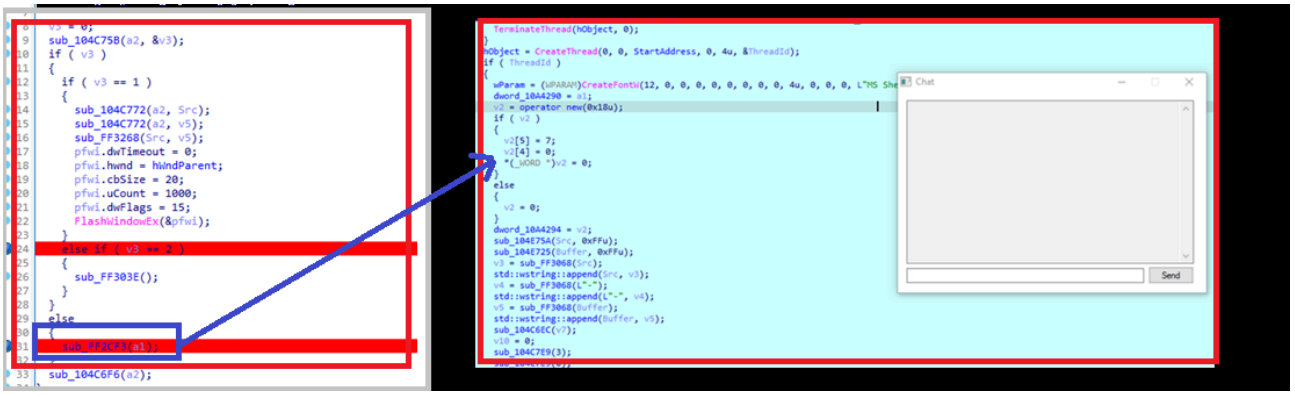


Figure 26. Creation of the Message Display Interface

When **command 3** is sent to the malware, it controls the message display interface according to the specified sub-parameter value. When the value **0** is received, a new message display interface is initialized. If an existing

window or thread is present, it is terminated, a new thread is created, and the required interface components and font are initialized before the window is displayed.

When the value **1** is received, two text values are obtained and displayed in the existing message window. The texts are combined, the display is updated, and the window flashes to attract the user's attention.

When the value **2** is received, the currently open message window is closed, the window state is cleared, and the running interface is terminated.

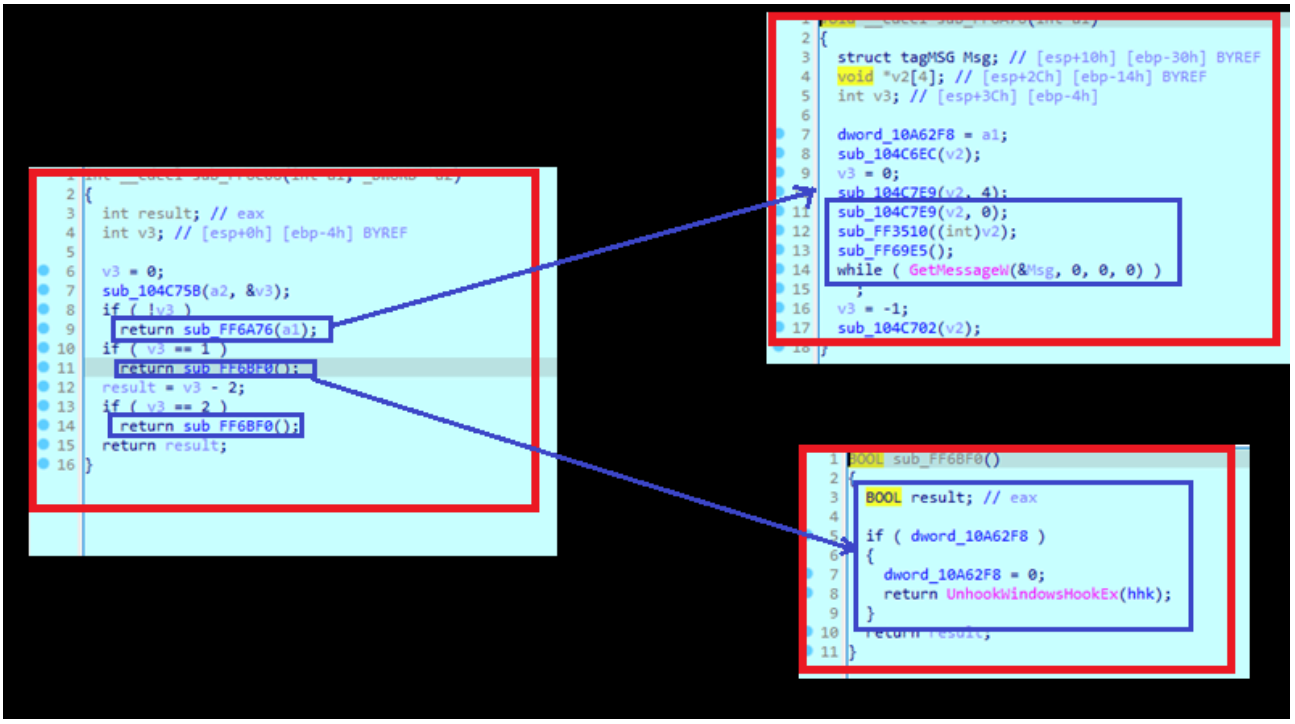


Figure 27. Initialization of the Windows Hook Mechanism

The **sub_FF6C06** function is a command handler that controls the Windows hook mechanism according to the received sub-parameter value. When the value **0** is received, a new thread and message loop are created to initialize the hook. The required structures are prepared, and the Windows message queue is continuously monitored.

When the value **1** or **2** is received, the existing hook is removed and the global hook state is cleared.

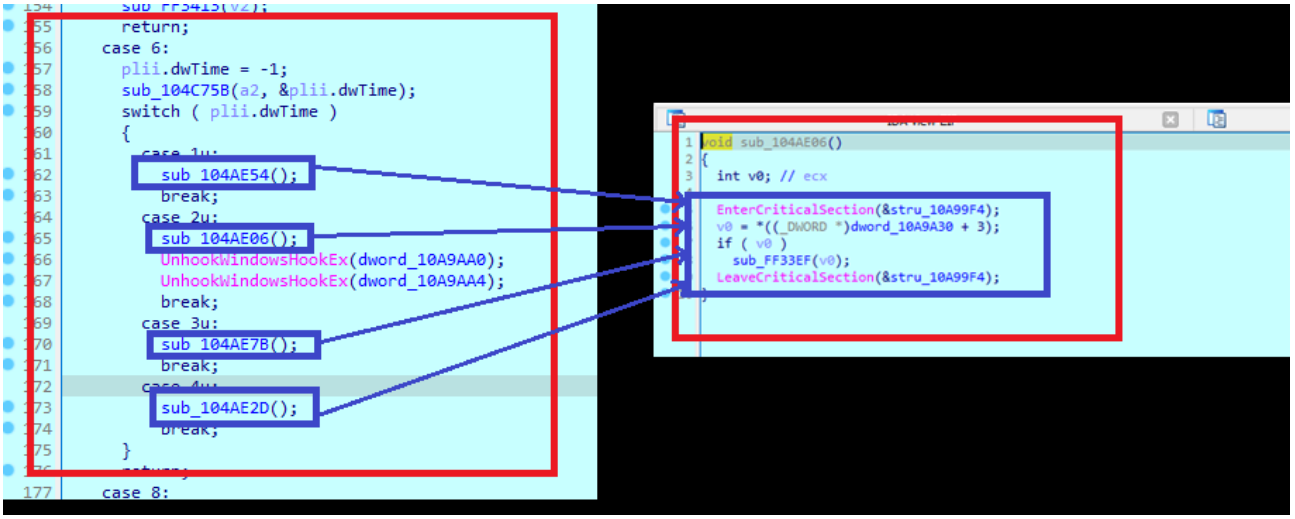


Figure 28. Starting the Windows Hook and Related Modules Based on Sub-Parameters

Command 6 controls the hook and related module operations according to the received sub-parameter value. When the value **1** is received, an operation that starts the monitoring or hook mechanism is performed. When the value **2** is received, the corresponding module is terminated and active Windows hook connections are removed.

When the value **3** is received, another control or monitoring component is initialized. When the value **4** is received, a stop or termination operation is performed on the corresponding component.

This structure functions as a control block responsible for initializing, managing, and terminating hook-based monitoring mechanisms on the system.



Figure 29. System Information Collection, Session Management, and Power State Operations

Command 8 is a command block that controls power management, session operations, and process information collection on the system.

When the value **0** is received, information about running processes is collected. When the value **1** is received, an additional system information collection operation is performed based on the specified parameter, and the collected data is returned.

When the value **3** is received, **ExitWindowsEx** is used to terminate the user's session. When the value **4** is received, the system is placed into sleep mode. When the value **5** is received, the system power state is changed and a standby/sleep operation is performed.



Figure 30. Retrieving the System Display and Screen Configuration

Command 9 is an information-gathering block used to collect **display/screen** configuration information from the system. When this command is executed, display information is first retrieved, and the available display devices or monitor configurations on the system are enumerated. For each display component, resolution information, position data, and related display parameters are collected and added to the packet.

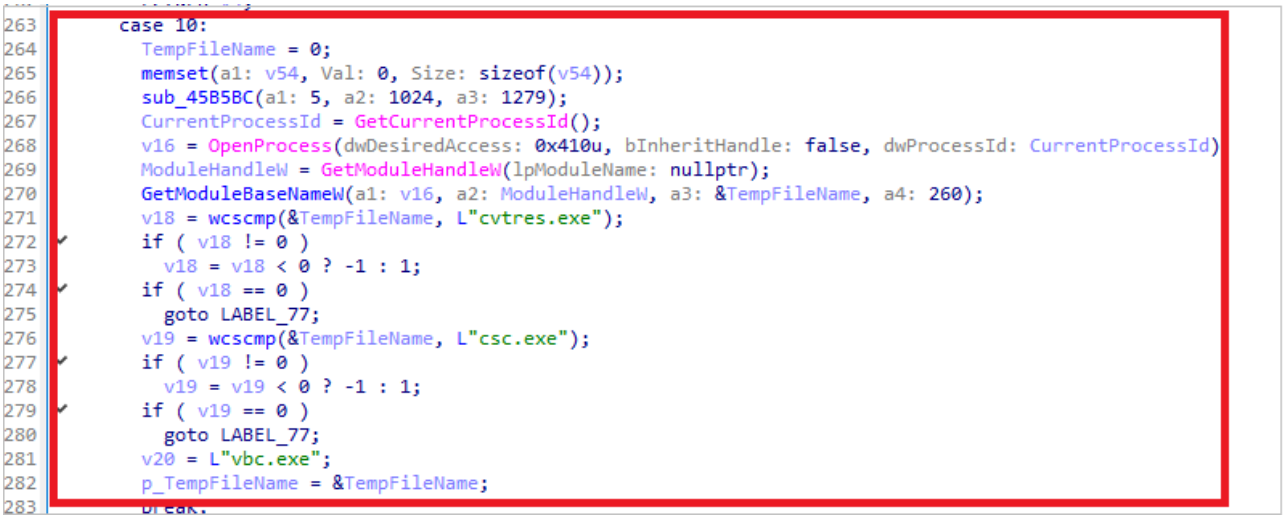


Figure 31. Process Name Check

Command 10 checks the name of the currently running process to identify specific execution environments. The PID of the current process is obtained using **GetCurrentProcessId**, while the name of the executed module is retrieved using **GetModuleHandleW** and **GetModuleBaseNameW**. The process name is then compared against specific executable names such as **cvtres.exe, csc.exe**, and **vbc.exe**.

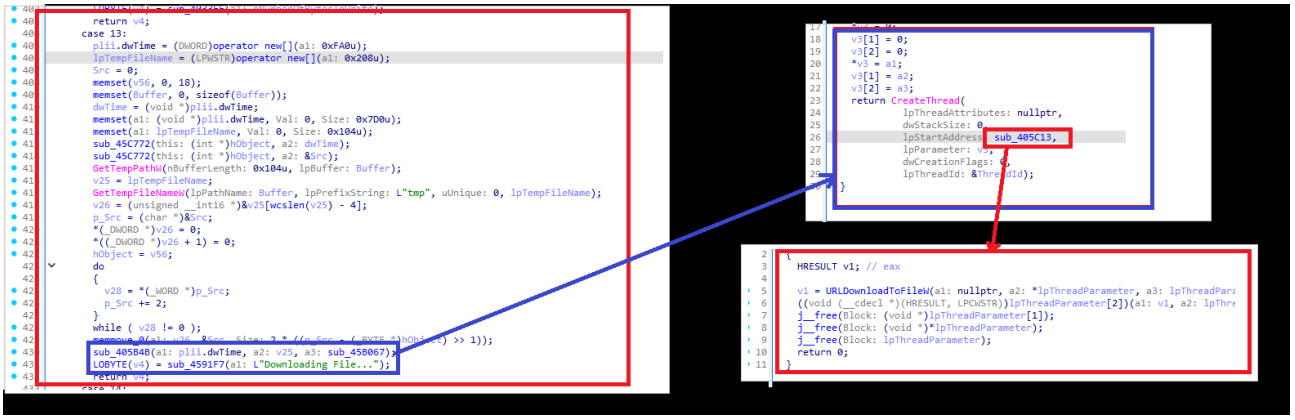


Figure 32. File Download and Temporary File Creation

Command 13 is the command block responsible for downloading a file from a remote source. It allocates memory for the received data and file path, determines the temporary directory using **GetTempPathW**, and creates a temporary file using **GetTempFileNameW**. It then calls the **sub_405B4B()** function to write the received data to this file and generates the **“Downloading File...”** message upon completion.

```
394       LOBYTE(v4) = sub_EB91F7(L"Downloading File...");
395       return v4;
396     case 14:
397       if ( dword_F19A3C )
398       {
399         LOBYTE(v4) = sub_EB91F7(L"DoS Already Active...");
400       }
401       else
402       {
403         v37 = (_WORD *)operator new(2);
404         if ( v37 )
405           *v37 = 0;
406         else
407           v37 = 0;
408         dword_F19A3C = v37;
409         sub_E65C7E(a2);
410         LOBYTE(v4) = sub_EB91F7(L"DoS Active...");
411       }
412       return v4;
413     case 15:
414       if ( dword_F19A3C )
415       {
416         sub_E65D97();
417         j__free(dword_F19A3C);
418         dword_F19A3C = 0;
419         LOBYTE(v4) = sub_EB91F7(L"DoS Stopped");
420       }
421       else
422       {
423         LOBYTE(v4) = sub_EB91F7(L"DoS Not Active");
424       }
425       return v4;
```

Figure 33. Starting and Stopping the DoS Operation

Case 14 is the command block responsible for starting the DoS operation. If an active DoS state already exists, a new operation is not started. Otherwise, the required structure is initialized, **sub_405C7E()** is called, and the **“DoS Active...”** message is generated.

Case 15 stops the active DoS operation by calling **sub_405D97()**, releases the allocated memory, and returns the **“DoS Stopped”** message.

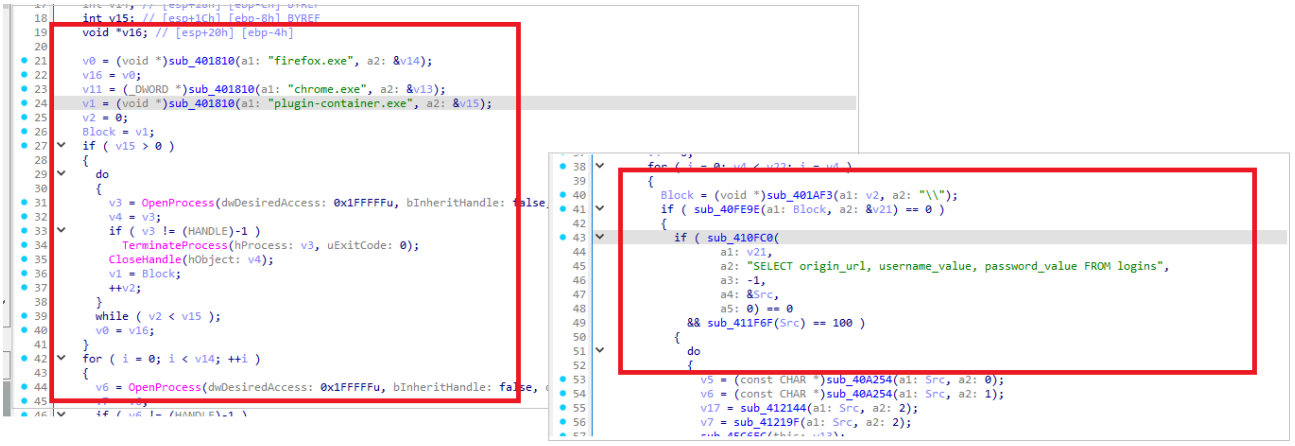


Figure 34. Browser Credential Collection

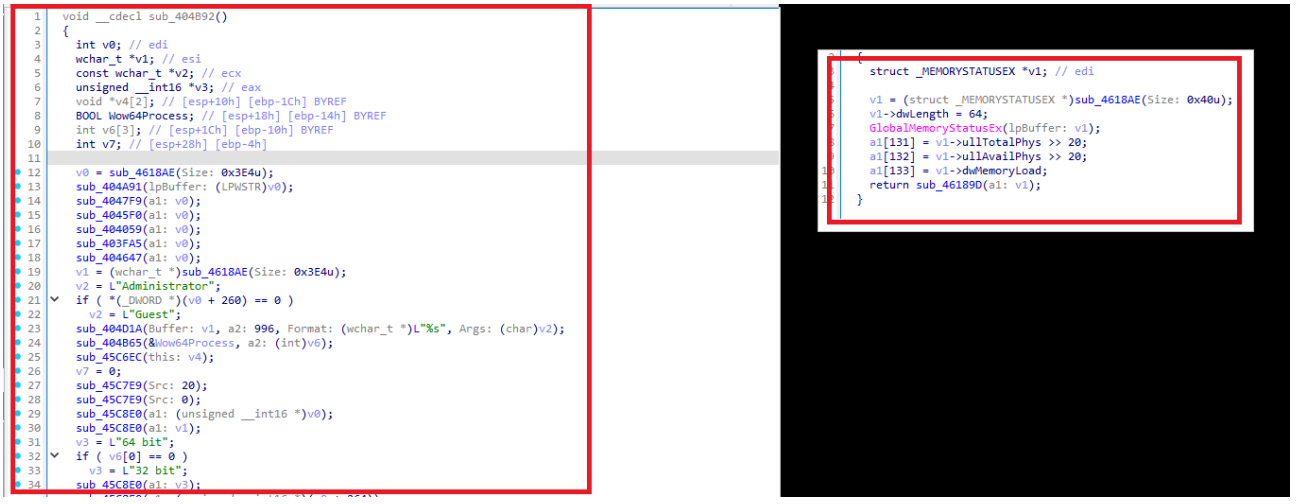
Command 16 targets stored credentials within **Google Chrome** and **Mozilla Firefox** profiles. The **sub_401E64** function terminates the relevant browser processes, while **sub_401909** retrieves URL, username, and stored password data from the **SQLite logins** table. The password data is then processed for decryption and prepared for transmission.



```
    {
      GetSystemDirectoryA(lpBuffer: String1, uSize: 0x104u);
      lstrcatA(lpString1: String1, lpString2: "\\drivers\\etc\\HOSTS");
      SetFileAttributesA(lpFileName: String1, dwFileAttributes: 0x80u);
      FileA = CreateFileA(
                lpFileName: String1,
                dwDesiredAccess: 0x80000000,
                dwShareMode: 1u,
                lpSecurityAttributes: nullptr,
                dwCreationDisposition: 3u,
                dwFlagsAndAttributes: 0,
                hTemplateFile: nullptr);
      FileSize = GetFileSize(hFile: FileA, lpFileSizeHigh: nullptr);
      v7 = operator new[](a1: FileSize);
      ReadFile(
        hFile: FileA,
        lpBuffer: v7,
        nNumberOfBytesToRead: FileSize,
        lpNumberOfBytesRead: &NumberOfBytesRead,
        lpOverlapped: nullptr);
      sub_45C6EC(this: v13);
      v18 = 0;
      sub_45C7E9(Src: 19);
      sub_45C7E9(Src: 0);
      sub_45C7E9(Src: FileSize);
      sub_45C86B(Src: v7, Size: FileSize);
      sub_403510(a1: (int)v13);
      CloseHandle(hObject: FileA);
      v18 = -1;
      sub_45C702(this: v13);
      v2 = v11;
      goto LABEL_11;
    }
    if ( v12 == 1 )
    {
      sub_45C75B(this: a2, a2: &nNumberOfBytesToWrite);
      for ( i = 0; i < (int)nNumberOfBytesToWrite; ++i )
        sub_45C745(this: a2, a2: &v15[i]);
      GetSystemDirectoryA(lpBuffer: Buffer, uSize: 0x104u);
      lstrcatA(lpString1: Buffer, lpString2: "\\drivers\\etc\\HOSTS");
      SetFileAttributesA(lpFileName: Buffer, dwFileAttributes: 0x80u);
      v4 = CreateFileA(
```

Figure 35. Reading and Modifying the HOSTS File

Command 19 manages the Windows **HOSTS** file. When the value **0** is received, the existing HOSTS file contents are read and collected. When the value **1** is received, the HOSTS file is rewritten using the supplied data.



```
void __cdecl sub_404892()
{
  int v0; // edi
  wchar_t *v1; // esi
  const wchar_t *v2; // ecx
  unsigned __int16 *v3; // eax
  void *v4[2]; // [esp+10h] [ebp-1Ch] BYREF
  BOOL Wow64Process; // [esp+18h] [ebp-14h] BYREF
  int v6[3]; // [esp+1Ch] [ebp-10h] BYREF
  int v7; // [esp+28h] [ebp-4h]

  v0 = sub_4618AE(Size: 0x3E4u);
  sub_404A91(lpBuffer: (LPWSTR)v0);
  sub_4047F9(a1: v0);
  sub_4045F0(a1: v0);
  sub_404059(a1: v0);
  sub_403FA5(a1: v0);
  sub_404647(a1: v0);
  v1 = (wchar_t *)sub_4618AE(Size: 0x3E4u);
  v2 = L"Administrator";
  if ( *(_DWORD *)(v0 + 260) == 0 )
    v2 = L"Guest";
  sub_404D1A(Buffer: v1, a2: 996, Format: (wchar_t *)L"%s", Args: (char)v2);
  sub_404B65(&Wow64Process, a2: (int)v6);
  sub_45C6EC(this: v4);
  v7 = 0;
  sub_45C7E9(Src: 20);
  sub_45C7E9(Src: 0);
  sub_45C8E0(a1: (unsigned __int16 *)v0);
  sub_45C8E0(a1: v1);
  v3 = L"64 bit";
  if ( v6[0] == 0 )
    v3 = L"32 bit";
  sub_45C8E0(a1: v3);
```

```
{
  struct _MEMORYSTATUSEX *v1; // edi

  v1 = (struct _MEMORYSTATUSEX *)sub_4618AE(Size: 0x40u);
  v1->dwLength = 64;
  GlobalMemoryStatusEx(lpBuffer: v1);
  a1[131] = v1->ullTotalPhys >> 20;
  a1[132] = v1->ullAvailPhys >> 20;
  a1[133] = v1->dwMemoryLoad;
  return sub_46189D(a1: v1);
}
```

Figure 36. System Information Collection

Command 20 performs comprehensive system information collection. Data including the username and privilege level, processor model and core count, RAM usage, region and time zone, display information, Windows version,

disk space, and system architecture are collected and transmitted to the server.

MITRE ATT&CK

Category	MITRE Techniques
Discovery	T1082 – System Information Discovery T1033 – System Owner/User Discovery T1057 – Process Discovery T1083 – File and Directory Discovery T1614 – System Location Discovery T1120 – Peripheral Device Discovery T1016 – System Network Configuration Discovery
Execution	T1059 – Command and Scripting Interpreter
Persistence	T1547.001 – Registry Run Keys / Startup Folder
Defense Evasion	T1564.001 – Hide Artifacts: Hidden Files and Directories T1553.005 – Mark-of-the-Web Bypass
Credential Access	T1555.003 – Credentials from Web Browsers T1056.001 – Input Capture: Keylogging
C&C	T1071 – Application Layer Protocol T1001 – Data Obfuscation T1105 – Ingress Tool Transfer
Exfiltration	T1041 – Exfiltration Over C2 Channel

Source: https://github.com/Yavuzhanzgen/BabylonRAT-Malware-Analysis/blob/main/Reports/BabylonRAT-EN.md