Readers can find more research by following the payload to reveal the payload config, containing the C2 information. We found that the initial loader was implemented in AutoIt and uses additional junk into these arrays that is removed from the constructed string, using the `Chr` function.

Speaking of strings, I noticed another similarity to Redline, the use of char array to string conversion at runtime. Although Redline in many cases did not use this technique, it is interesting to see it being used here. This looks like a potential config class. It initializes the properties used for the initial TCP connection and the string download we saw in the logs.

Apart from that, the properties and fields values in the Locals window by expanding the `this` parameter. It contains quite a few properties, but the most interesting is the constructor. The second argument is the key. The first variable is the resulting string. The function takes the following arguments:

```csharp
static string Decrypt(string encrypted, string key)
```

Looking at the switch cases inside the loops, I realized that only the branches that use `true` return a value. This is a common technique to avoid null values. The function looks like it is encrypting the input string using a unique decryption key.

My guess was correct, I found a string decryption method in the binary. The message box claims I violated a EULA which I never read nor agreed to. I guess we can't debug the malware any further, how unfortunate. Luckily, we can still analyze the binary.

At the time of writing, the loader does not run anymore but fails with an error message about Windows updates. Sifting through the string dump I found that the loader is trying to inject itself into a process. Since we already know the malware is a .NET payload, we can infer that it is trying to inject a .NET payload into another process.

Dumping the binary using ExtremeDumper and analyzing it, we found interesting features. The `TcpClient` class is being used to connect to a remote host. The `Connect` method is called with the following arguments:

```csharp
tcpClient.Connect(xj.a, d)
```

The `AutoIt` loader contains quite a few similarities to Redline, both making use of DataContracts and async tasks for the separate stealer module. The loader is written in AutoIt and calculates the string configuration in every method in the `AutoIt` script. Considering all this I ditched the idea of writing a static string decryption tool.

We can see a Pastebin link that caught my interest. The paste contains another IP address, `10.0.0.1`, which indicates that this AutoIt script is trying to connect to this IP address. We can also see a `Process Hollowing` module. After searching for the `Process Hollowing` module, we found that it is used to inject the loader into a process. We can also see a `NtResumeThread` call, which is used to resume the execution of a suspended thread.

After dealing with the rather simple anti-debug, we let it run. When debugged, the executable spawns a file dialog asking for an input. The message box claims I violated a EULA which I never read nor agreed to. I guess we can't debug the malware any further, how unfortunate. Luckily, we can still analyze the binary.

Checking Process Explorer confirms that the decrypted path from the AutoIt script was indeed the injection target. We add another breakpoint on the `NtResumeThread` call to ensure we break before the injection process is suspended. We then let the loader run and analyze it.

The loader looks pretty messy at first but taking a closer look I found something that stuck out: The calls to the function called `Decrypt` are not in the expected order. This is a common technique to avoid static analysis. The logo of AutoIt and the copyright information says it's AutoIt. This leads me to believe that this executable is the runtime required to execute the malware.

In this post, I will be going over my process of analyzing a sample of ArechClient2. Including initial analysis, deobfuscation and unpacking of the loader. This is a good example of how reverse engineering can be used to understand the behavior of a malware sample. It took me a few days to understand the code and find the injection point. The post offers not only the technical details but also the logical reasoning conducted as the investigation unfolded. The post offers insights into the malware's behavior and how it can be prevented.

We can also see a `Reverse Engineering Walkthrough` section in the blog. This is a good resource for those interested in learning more about reverse engineering. The blog also has a section on `Cybersecurity Sharing | An Infosec User's Guide to Getting Started on Mastodon`. This is a great resource for those interested in learning more about cybersecurity. The blog also has a section on `Cyber Risks in the Education Sector | Why Cybersecurity Needs to Be Top of the Class`. This is a great resource for those interested in learning more about cybersecurity in the education sector.

In partnership with the `SentinelOne's Cybersecurity Predictions 2023 | What's Next?` blog, we can see interesting insights into the future of cybersecurity. The blog also has a section on `Strengthening Cyber Defenses | A Guide to Enhancing Modern Tabletop Exercises`. This is a great resource for those interested in learning more about cybersecurity.

July 17, 2023

We can see a `null` in the code, which is a common technique to avoid null values. This is a common technique to avoid null values. The function looks like it is encrypting the input string using a unique decryption key. We can also see a `null` in the code, which is a common technique to avoid null values. This is a common technique to avoid null values.

Looking at the switch cases inside the loops, I realized that only the branches that use `true` return a value. This is a common technique to avoid null values. The function looks like it is encrypting the input string using a unique decryption key.