SunBurst: the next level of stealth

blog.reversinglabs.com/blog/sunburst-the-next-level-of-stealth

Threat Research | December 16, 2020

Blog Author
Tomislav Peričin, Chief Software Architect & Co-Founder at ReversingLabs. Read More...

Executive summary

ReversingLabs:

- shows conclusive details that Orion software **build and code signing infrastructure was compromised.**
- discloses compilation artifacts confirming that Orion **source code was directly modified** to include a malicious backdoor.
- discloses software delivery artifacts confirming that a **backdoored Orion software patch was delivered through its existing software release management system.**
- proposes a **novel approach to detect and prevent future software supply chain attacks.**

REVERSINGLABS
Could your software build process release the next major breach?
Are 3rd party software threats going unchecked?
LEARN HOW WE CAN HELP

Summary

SolarWinds, a company that makes IT monitoring and management solutions, has become the latest target of a sophisticated supply chain attack. Multiple SolarWinds Orion software updates, released between March and June 2020, have been found to contain backdoor code that enables the attackers to conduct surveillance and execute arbitrary commands on affected systems.

ReversingLabs' research into the anatomy of this supply chain attack unveiled conclusive details showing that Orion software build and code signing infrastructure was compromised. The source code of the affected library was directly modified to include malicious backdoor code, which was compiled, signed and delivered through the existing software patch release management system.

While this type of attack on the software supply chain is by no means novel, what is different this time is the level of stealth the attackers used to remain undetected for as long as possible. The attackers blended in with the affected code base, mimicking the software developers' coding style and naming standards. This was consistently demonstrated through a significant number of functions they added to turn Orion software into a backdoor for any organization that uses it.

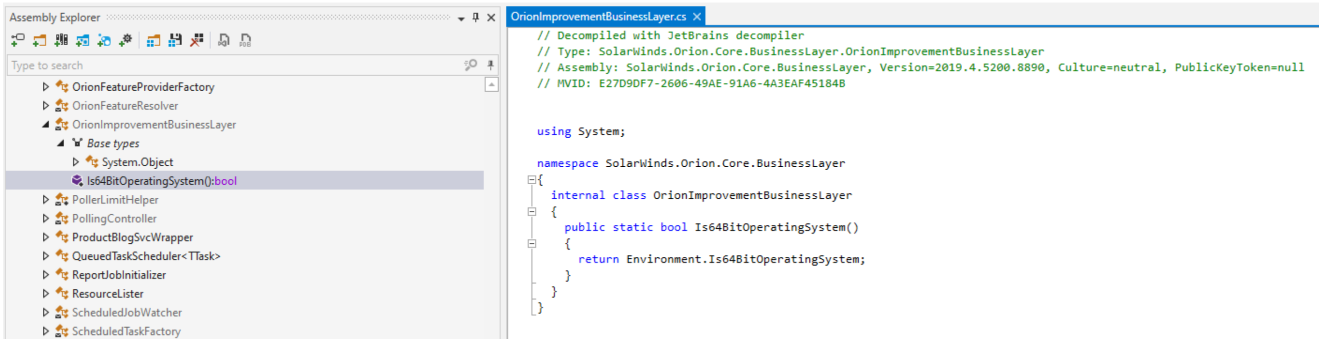
Hiding from software developers

Piecing together a story from the outside of the incident is difficult. However, the trail of breadcrumbs left behind is sufficient to glean some insight into the methods the attackers used to compromise the Orion software release process.

Such an investigation typically starts with what's known, which in this case is the list of backdoored software libraries. A file named SolarWinds.Orion.Core.BusinessLayer.dll within the Orion platform software package update SolarWinds-Core-v2019.4.5220-Hotfix5.msp is the first version known to contain the malicious backdoor code. That library has been thoroughly analyzed in FireEye's technical blog, which describes the backdoor behavior very well.

However, we can draw further conclusions about the attackers' patience, sophistication and the state of Orion software build system from the analysis of metadata.

While the first version to contain the malicious backdoor code was 2019.4.5200.9083, as outlined by the FireEye blog, there was a previous version that was tampered with by the attackers: version 2019.4.5200.8890, from October 2019, and this version had only been slightly modified. While it doesn't contain the malicious backdoor code, it does contain the .NET class that will host it in the future.



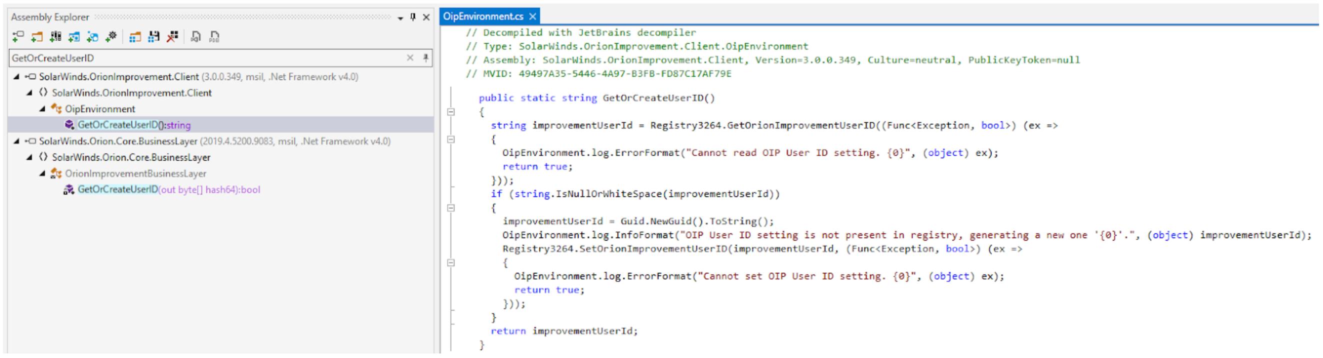
```
// Decompiled with JetBrains decompiler
// Type: SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
// Assembly: SolarWinds.Orion.Core.BusinessLayer, Version=2019.4.5200.8890, Culture=neutral, PublicKeyToken=null
// MVID: E27D9DF7-2606-49AE-91A6-4A3EAF45184B


using System;

namespace SolarWinds.Orion.Core.BusinessLayer
{
  internal class OrionImprovementBusinessLayer
  {
    public static bool Is64BitOperatingSystem()
    {
      return Environment.Is64BitOperatingSystem;
    }
  }
}
```

Figure 1. - Empty .NET class prior to backdoor code addition [ver. 2019.4.5200.8890]

This first code modification was clearly just a proof of concept. Their three step action plan: Compromise the build system, inject their own code, and verify that their signed packages are going to appear on the client side as expected. Once these objectives were met, and the attackers proved to themselves that the supply chain could be compromised, they started planning the real attack payload.

The name of the class, **OrionImprovementBusinessLayer**, had been chosen deliberately. Not only to blend in with the rest of the code, but also to fool the software developers or anyone auditing the binaries. That class, and many of the methods it uses, can be found in other Orion software libraries, even thematically fitting with the code found within those libraries. This implies not only the intent to remain stealthy, but also that the attackers were highly familiar with the code base.

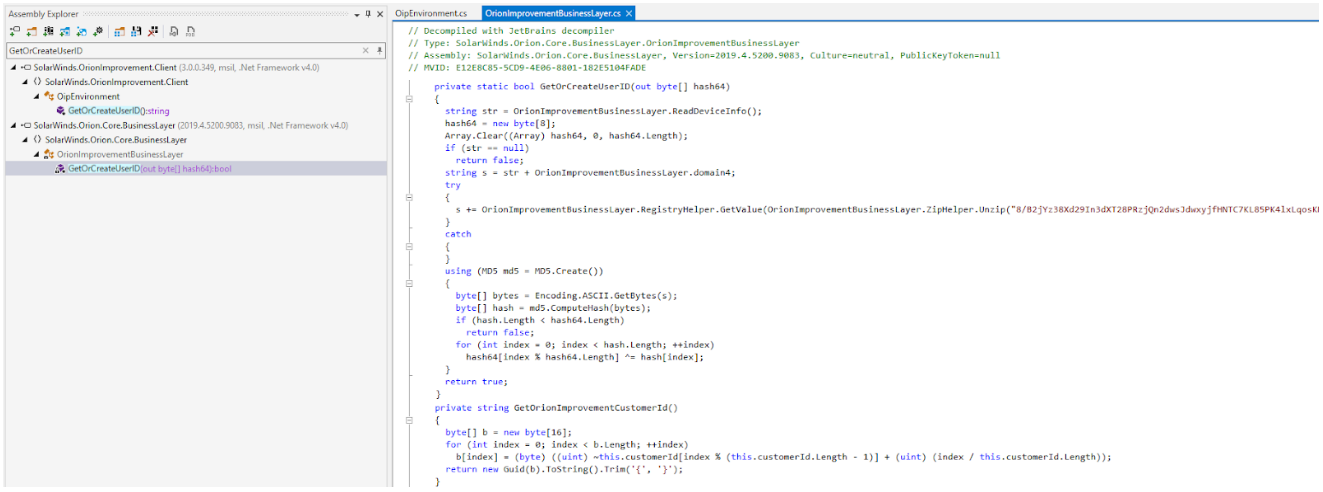
Compare, for instance, the functions that compute the UserID. In the Orion Client code, this function tries to read the previously computed value from the registry, or creates a new GUID for the user.



```
// Decompiled with JetBrains decompiler
// Type: SolarWinds.OrionImprovement.Client.OipEnvironment
// Assembly: SolarWinds.OrionImprovement.Client, Version=3.0.0.349, Culture=neutral, PublicKeyToken=null
// MVID: 49497A35-5446-4A97-B3FB-FD87C17AF79E

public static string GetOrCreateUserID()
{
  string improvementUserId = Registry3264.GetOrionImprovementUserID((Func<Exception, bool>) (ex =>
  {
    OipEnvironment.log.ErrorFormat("Cannot read OIP User ID setting. {0}", (object) ex);
    return true;
  }));
  if (string.IsNullOrWhiteSpace(improvementUserId))
  {
    improvementUserId = Guid.NewGuid().ToString();
    OipEnvironment.log.InfoFormat("OIP User ID setting is not present in registry, generating a new one '{0}'.", (object) improvementUserId);
    Registry3264.SetOrionImprovementUserID(improvementUserId, (Func<Exception, bool>) (ex =>
    {
      OipEnvironment.log.ErrorFormat("Cannot set OIP User ID setting. {0}", (object) ex);
      return true;
    }));
  }
  return improvementUserId;
}
```

Figure 2. - GetOrCreateUserID in Orion Client [ver. 3.0.0.349]

Mimicking that, the attackers created their own implementations of these functions to also compute the UserID, and named them the same way. Their functions are even using the same GUID format for the ID type later on.



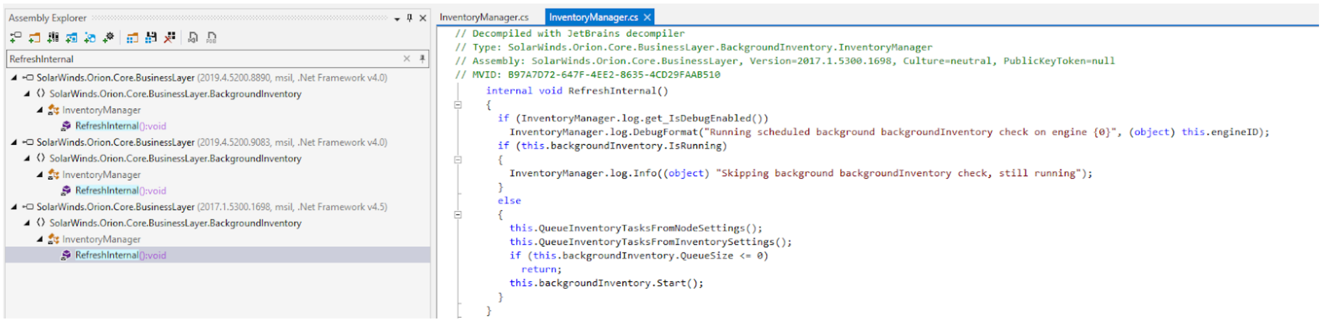
```
// Decompiled with JetBrains decompiler
// Type: SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
// Assembly: SolarWinds.Orion.Core.BusinessLayer, Version=2019.4.5200.9083, Culture=neutral, PublicKeyToken=null
// MVID: E12E8C85-5CD9-4E06-8801-182E5104FADE

private static bool GetOrCreateUserID(out byte[] hash64)
{
  string str = OrionImprovementBusinessLayer.ReadDeviceInfo();
  hash64 = new byte[8];
  Array.Clear((Array) hash64, 0, hash64.Length);
  if (str == null)
    return false;
  string s = str + OrionImprovementBusinessLayer.domain4;
  try
  {
    s += OrionImprovementBusinessLayer.RegistryHelper.GetValue(OrionImprovementBusinessLayer.ZipHelper.Unzip("8/B2jYz38Xd29In3dXT28PRzjQn2dwsJdwxyjfHNTC7KL85PK4lxLqosKI
  }
  catch
  {
  }
  using (MD5 md5 = MD5.Create())
  {
    byte[] bytes = Encoding.ASCII.GetBytes(s);
    byte[] hash = md5.ComputeHash(bytes);
    if (hash.Length < hash64.Length)
      return false;
    for (int index = 0; index < hash.Length; ++index)
      hash64[index % hash64.Length] ^= hash[index];
  }
  return true;
}
private string GetOrionImprovementCustomerId()
{
  byte[] b = new byte[16];
  for (int index = 0; index < b.Length; ++index)
    b[index] = (byte) ((uint) ~this.customerId[index % (this.customerId.Length - 1)] + (uint) (index / this.customerId.Length));
  return new Guid(b).ToString().Trim('{', '}');
}
```

Figure 3. - GetOrCreateUserID in backdoor class [ver. 2019.4.5200.9083]

While not spot on, this code performs a similar function as the original. The pattern of naming classes, members, and variables appropriately is visible everywhere in the backdoored code.

There really is a method called **CollectSystemDescription** and **UploadSystemDescription** used by Orion Client library code. Just like there was an **IOrionImprovementBusinessLayer** interface the attackers mimicked for the name of the class in which they placed the backdoor code.

However, any code added to the library doesn't just magically execute itself. The attackers still need to call it somehow. And the way that was done tells us that the build system itself was compromised.


Assembly Explorer

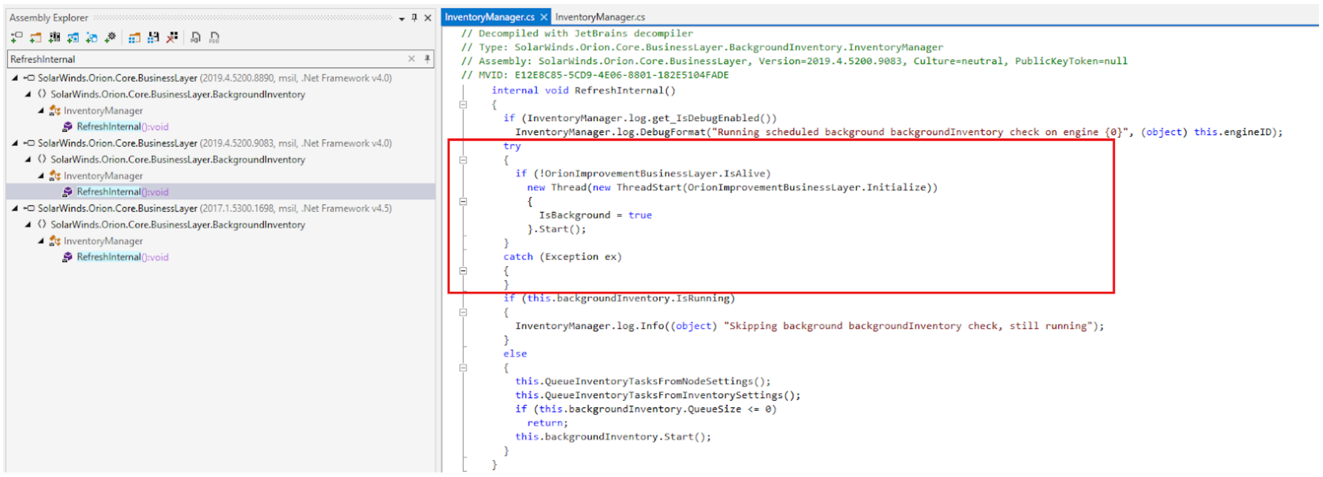
RefreshInternal

```
◢ SolarWinds.Orion.Core.BusinessLayer (2019.4.5200.8890, msil, .Net Framework v4.0)
  ◢ SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory
    ◢ InventoryManager
        RefreshInternal():void
◢ SolarWinds.Orion.Core.BusinessLayer (2019.4.5200.9083, msil, .Net Framework v4.0)
  ◢ SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory
    ◢ InventoryManager
        RefreshInternal():void
◢ SolarWinds.Orion.Core.BusinessLayer (2017.1.5300.1698, msil, .Net Framework v4.5)
  ◢ SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory
    ◢ InventoryManager
        RefreshInternal():void
```

InventoryManager.cs | InventoryManager.cs

```
// Decompiled with JetBrains decompiler
// Type: SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory.InventoryManager
// Assembly: SolarWinds.Orion.Core.BusinessLayer, Version=2017.1.5300.1698, Culture=neutral, PublicKeyToken=null
// MVID: B97A7D72-647F-4EE2-8635-4CD29FAAB510
    internal void RefreshInternal()
    {
      if (InventoryManager.log.get_IsDebugEnabled())
        InventoryManager.log.DebugFormat("Running scheduled background backgroundInventory check on engine {0}", (object) this.engineID);
      if (this.backgroundInventory.IsRunning)
      {
        InventoryManager.log.Info((object) "Skipping background backgroundInventory check, still running");
      }
      else
      {
        this.QueueInventoryTasksFromNodeSettings();
        this.QueueInventoryTasksFromInventorySettings();
        if (this.backgroundInventory.QueueSize <= 0)
          return;
        this.backgroundInventory.Start();
      }
    }
```

Figure 4. - RefreshInternal in the clean software version [ver. 2017.1.5300.1698]


Assembly Explorer

RefreshInternal

```
◢ SolarWinds.Orion.Core.BusinessLayer (2019.4.5200.8890, msil, .Net Framework v4.0)
  ◢ SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory
    ◢ InventoryManager
        RefreshInternal():void
◢ SolarWinds.Orion.Core.BusinessLayer (2019.4.5200.9083, msil, .Net Framework v4.0)
  ◢ SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory
    ◢ InventoryManager
        RefreshInternal():void
◢ SolarWinds.Orion.Core.BusinessLayer (2017.1.5300.1698, msil, .Net Framework v4.5)
  ◢ SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory
    ◢ InventoryManager
        RefreshInternal():void
```

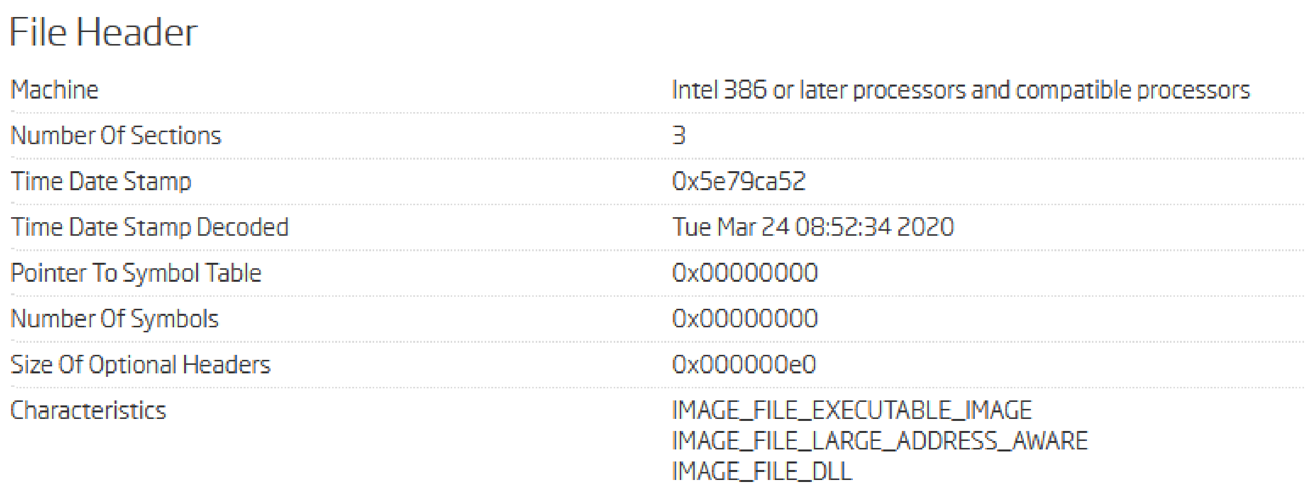
InventoryManager.cs | InventoryManager.cs

```
// Decompiled with JetBrains decompiler
// Type: SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory.InventoryManager
// Assembly: SolarWinds.Orion.Core.BusinessLayer, Version=2019.4.5200.9083, Culture=neutral, PublicKeyToken=null
// MVID: E12E8C85-5CD9-4E06-8801-182E5104FADE
    internal void RefreshInternal()
    {
      if (InventoryManager.log.get_IsDebugEnabled())
        InventoryManager.log.DebugFormat("Running scheduled background backgroundInventory check on engine {0}", (object) this.engineID);
      try
      {
        if (!OrionImprovementBusinessLayer.IsAlive)
          new Thread(new ThreadStart(OrionImprovementBusinessLayer.Initialize))
          {
            IsBackground = true
          }.Start();
      }
      catch (Exception ex)
      {
      }
      if (this.backgroundInventory.IsRunning)
      {
        InventoryManager.log.Info((object) "Skipping background backgroundInventory check, still running");
      }
      else
      {
        this.QueueInventoryTasksFromNodeSettings();
        this.QueueInventoryTasksFromInventorySettings();
        if (this.backgroundInventory.QueueSize <= 0)
          return;
        this.backgroundInventory.Start();
      }
    }
```

Figure 5. - RefreshInternal in the backdoored library [ver. 2019.4.5200.9083]

Code highlighted in red is the additional functionality the attackers put in. This small block of code creates a new thread that runs the backdoor while Orion software is performing its background inventory checks. Such a location is perfect for this kind of code to be added, as the original code is already dealing with the long-running background tasks. So like the rest of the attacker-injected code, it just blends in.

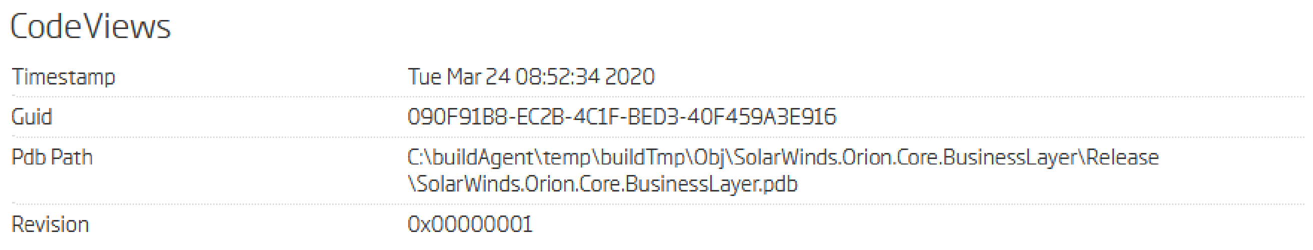
While there are techniques to decompile the .NET code, inject something new, and recompile the code afterwards, this wasn't the case here. The **InventoryManager** class was modified at the source code level, and the file was ultimately built with the regular Orion software build system.
This can be confirmed by looking at the timestamps for the backdoored binary, other libraries within the same package, and the patch file that delivers them.



File Header

Machine	Intel 386 or later processors and compatible processors
Number Of Sections	3
Time Date Stamp	0x5e79ca52
Time Date Stamp Decoded	Tue Mar 24 08:52:34 2020
Pointer To Symbol Table	0x00000000
Number Of Symbols	0x00000000
Size Of Optional Headers	0x000000e0
Characteristics	IMAGE_FILE_EXECUTABLE_IMAGE IMAGE_FILE_LARGE_ADDRESS_AWARE IMAGE_FILE_DLL

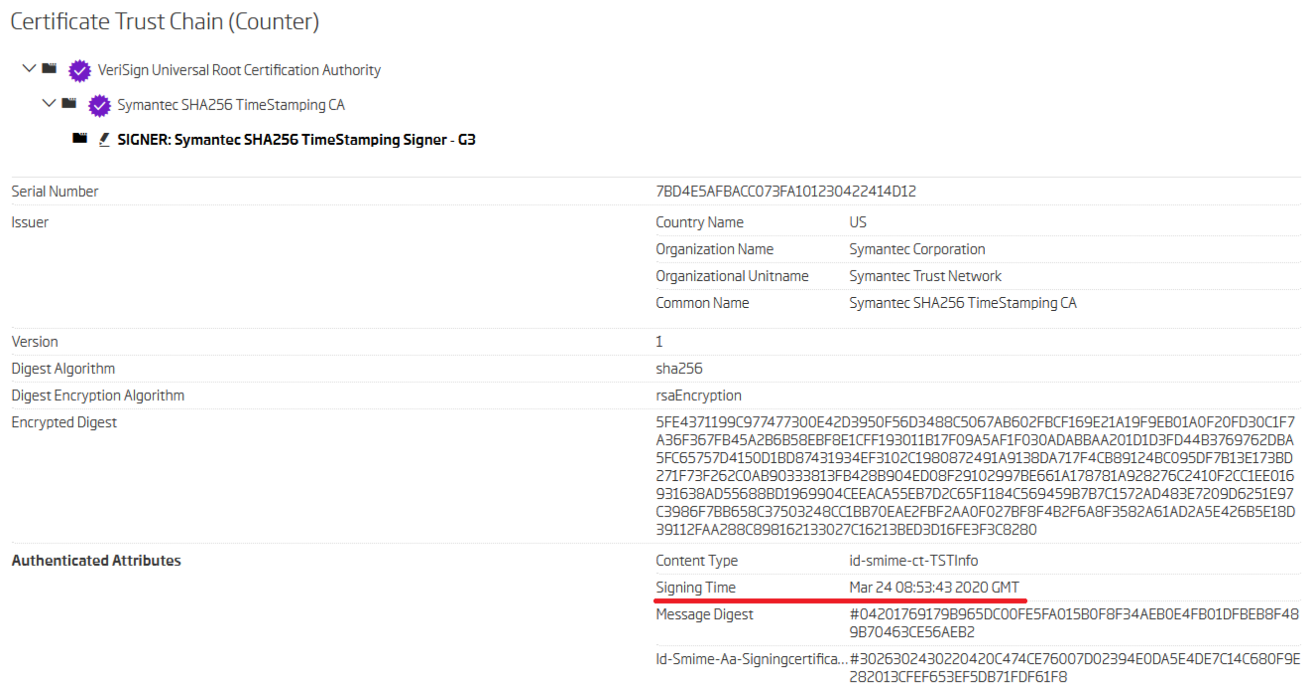
Figure 6. - Backdoored library compile time [ver. 2019.4.5200.9083]



CodeViews

Timestamp	Tue Mar 24 08:52:34 2020
Guid	090F91B8-EC2B-4C1F-BED3-40F459A3E916
Pdb Path	C:\buildAgent\temp\buildTmp\Obj\SolarWinds.Orion.Core.BusinessLayer\Release \SolarWinds.Orion.Core.BusinessLayer.pdb
Revision	0x00000001

Figure 7. - Backdoored library PDB symbols time [ver. 2019.4.5200.9083]



Certificate Trust Chain (Counter)

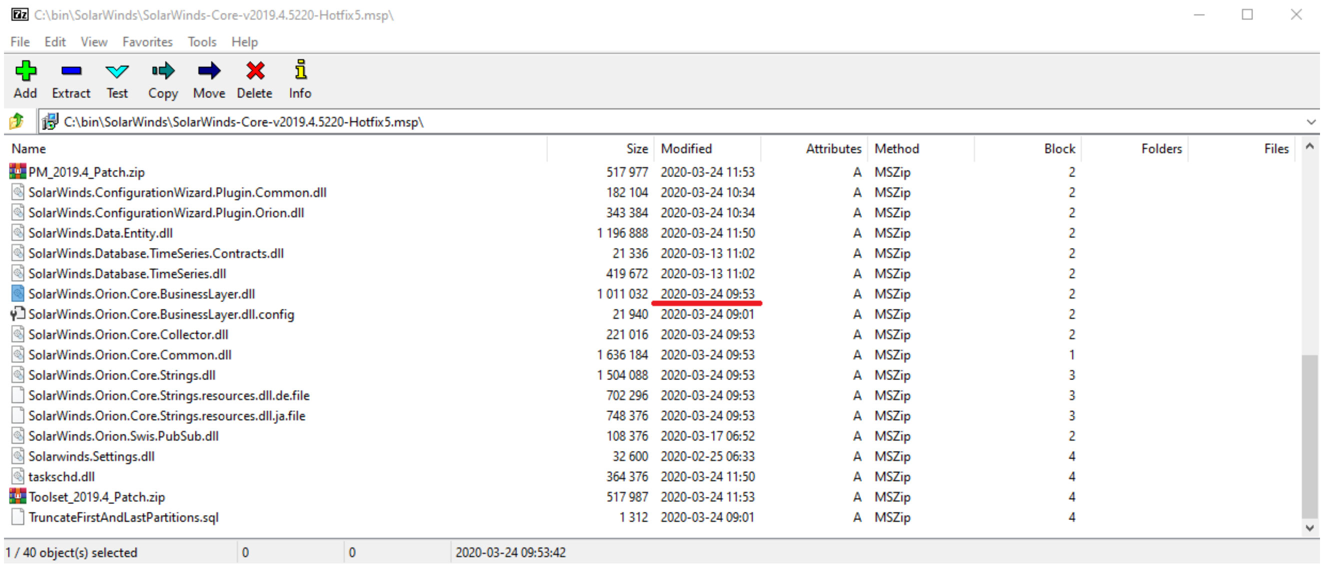
- VeriSign Universal Root Certification Authority
  - Symantec SHA256 TimeStamping CA
    - **SIGNER: Symantec SHA256 TimeStamping Signer - G3**

Serial Number	7BD4E5AFBACC073FA101230422414D12	
Issuer	Country Name	US
	Organization Name	Symantec Corporation
	Organizational Unitname	Symantec Trust Network
	Common Name	Symantec SHA256 TimeStamping CA
Version	1	
Digest Algorithm	sha256	
Digest Encryption Algorithm	rsaEncryption	
Encrypted Digest	5FE4371199C977477300E42D3950F56D3488C5067AB602FBCF169E21A19F9EB01A0F20FD30C1F7 A36F367FB45A2B6B58EBF8E1CFF193011B17F09A5AF1F030ADABBAA201D1D3FD44B3769762DBA 5FC65757D4150D1BD87431934EF3102C1980872491A9138DA717F4CB89124BC095DF7B13E173BD 271F73F262C0AB90333813FB428B904ED08F29102997BE661A178781A928276C2410F2CC1EE016 931638AD55688BD1969904CEEACA55EB7D2C65F1184C569459B7B7C1572AD483E7209D6251E97 C3986F7BB658C37503248CC1BB70EAE2FBF2AA0F027BF8F4B2F6A8F3582A61AD2A5E426B5E18D 39112FAA288C898162133027C16213BED3D16FE3F3C8280	
Authenticated Attributes	Content Type	id-smime-ct-TSTInfo
	Signing Time	Mar 24 08:53:43 2020 GMT
	Message Digest	#04201769179B965DC00FE5FA015B0F8F34AEB0E4FB01DFBEB8F48 9B70463CE56AEB2
	Id-Smime-Aa-Signingcertifica...	#3026302430220420C474CE76007D02394E0DA5E4DE7C14C680F9E 282013CFEF653EF5DB71FDF61F8

Figure 8. - Backdoored library signing time [ver. 2019.4.5200.9083]

Timestamps between the PE file headers and the CodeViews match perfectly. That, with the revision number set to one, means that the file was compiled only once – or that it was a clean build. Since the file was signed, and cross-signed for timestamping, the timestamps within the headers can be reliably validated. The cross-signing timestamp is controlled by a remote server that is outside of the build environment, and can't be tampered with.

Signing occurred within a minute of library compilation. That leaves no time for the attackers to be able to monitor the build system, replace the binary and change the metadata to match this perfectly. The simplest way for all these timestamp artifacts to align perfectly is to have the attackers' code injected directly into the source, and then have the existing build and signing system perform the compilation and release processes as defined by the Orion software developers.


C:\bin\SolarWinds\SolarWinds-Core-v2019.4.5220-Hotfix5.msp\
File Edit View Favorites Tools Help
Add Extract Test Copy Move Delete Info
C:\bin\SolarWinds\SolarWinds-Core-v2019.4.5220-Hotfix5.msp\

Name	Size	Modified	Attributes	Method	Block	Folders	Files
PM_2019.4_Patch.zip	517 977	2020-03-24 11:53	A	MSZip	2		
SolarWinds.ConfigurationWizard.Plugin.Common.dll	182 104	2020-03-24 10:34	A	MSZip	2		
SolarWinds.ConfigurationWizard.Plugin.Orion.dll	343 384	2020-03-24 10:34	A	MSZip	2		
SolarWinds.Data.Entity.dll	1 196 888	2020-03-24 11:50	A	MSZip	2		
SolarWinds.Database.TimeSeries.Contracts.dll	21 336	2020-03-13 11:02	A	MSZip	2		
SolarWinds.Database.TimeSeries.dll	419 672	2020-03-13 11:02	A	MSZip	2		
SolarWinds.Orion.Core.BusinessLayer.dll	1 011 032	2020-03-24 09:53	A	MSZip	2		
SolarWinds.Orion.Core.BusinessLayer.dll.config	21 940	2020-03-24 09:01	A	MSZip	2		
SolarWinds.Orion.Core.Collector.dll	221 016	2020-03-24 09:53	A	MSZip	2		
SolarWinds.Orion.Core.Common.dll	1 636 184	2020-03-24 09:53	A	MSZip	1		
SolarWinds.Orion.Core.Strings.dll	1 504 088	2020-03-24 09:53	A	MSZip	3		
SolarWinds.Orion.Core.Strings.resources.dll.de.file	702 296	2020-03-24 09:53	A	MSZip	3		
SolarWinds.Orion.Core.Strings.resources.dll.ja.file	748 376	2020-03-24 09:53	A	MSZip	3		
SolarWinds.Orion.Swis.PubSub.dll	108 376	2020-03-17 06:52	A	MSZip	2		
Solarwinds.Settings.dll	32 600	2020-02-25 06:33	A	MSZip	4		
taskschd.dll	364 376	2020-03-24 11:50	A	MSZip	4		
Toolset_2019.4_Patch.zip	517 987	2020-03-24 11:53	A	MSZip	4		
TruncateFirstAndLastPartitions.sql	1 312	2020-03-24 09:01	A	MSZip	4		

```
1 / 40 object(s) selected    0    0    2020-03-24 09:53:42
```

Figure 9. - Backdoored library file modification time [ver. 2019.4.5200.9083]

Finally, the MSP patch file contains a CAB archive that preserves the local last modified time for the library. Which, assuming the build system is running in the GMT+1 time zone, also confirms that the file was last modified during signing.

The files surrounding the backdoored library that belong to the same namespace were also compiled at the same time. Since they don't depend on each other, they wouldn't be built at the same time unless the build system was not running a complete build.

Since the MSP patch file is signed, and its signing time matches the contents of the package, this confirms that the patch file was created on the same machine as the rest of the build.

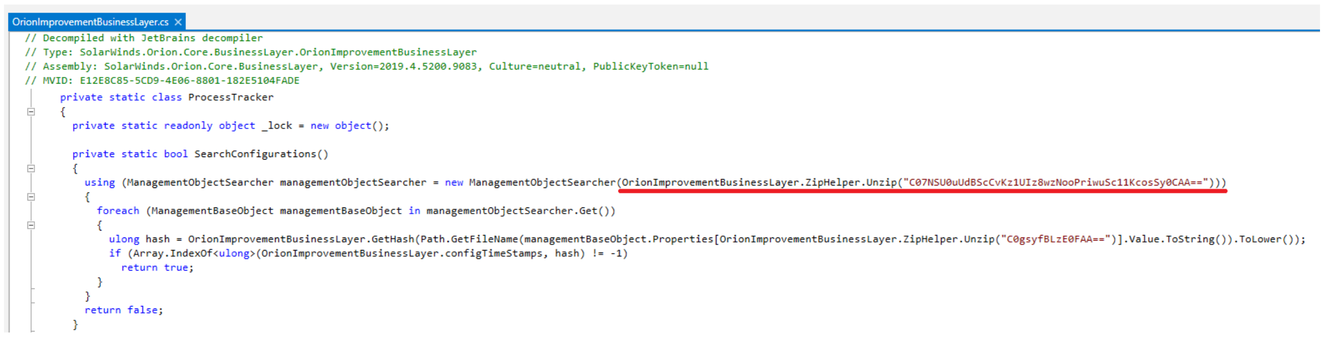
The big question is: was source control compromised, or was the attackers' code just placed on the build machine?

Unfortunately, that is something the metadata can’t reveal. There are no such artifacts that get preserved during software compilation. But the attackers went through a lot of trouble to ensure that their code looks like it belongs within the code base. That was certainly done to hide the code from the audit by the software developers.

What is certain is that the build infrastructure was compromised. In addition, the digital signing system was forced to sign untrusted code. While there’s no evidence at the moment that SolarWinds certificates were used to sign other malicious code, that possibility should not be excluded. And, as a precaution, all certificates and keys used on that build system should be revoked.

Hiding from security analysts

Consider for a second the type of customer that runs Orion software within their environment. For a software supply chain attack like this to work, the attackers need to keep under the radar and evade millions of dollars of security investment. They need to fool the highly specialized detection software, the people that run it to detect threats, and use it to proactively hunt for anomalies – for months. To pull that trick off, the attackers need to strike the right balance between staying hidden and achieving their objective.



```
// Decompiled with JetBrains decompiler
// Type: SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
// Assembly: SolarWinds.Orion.Core.BusinessLayer, Version=2019.4.5200.9083, Culture=neutral, PublicKeyToken=null
// MVID: E12E8C85-5CD9-4E06-8801-182E5104FADE
    private static class ProcessTracker
    {
      private static readonly object _lock = new object();

      private static bool SearchConfigurations()
      {
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher(OrionImprovementBusinessLayer.ZipHelper.Unzip("C07NSU0uUdBScCvKz1UIz8wzNooPriwuSc11KcosSy0CAA==")))
        {
          foreach (ManagementBaseObject managementBaseObject in managementObjectSearcher.Get())
          {
            ulong hash = OrionImprovementBusinessLayer.GetHash(Path.GetFileName(managementBaseObject.Properties[OrionImprovementBusinessLayer.ZipHelper.Unzip("C0gsyfBLzE0FAA==")].Value.ToString()).ToLower());
            if (Array.IndexOf<ulong>(OrionImprovementBusinessLayer.configTimeStamps, hash) != -1)
              return true;
          }
        }
        return false;
      }
```

Figure 10. - Backdoored library obfuscated strings [ver. 2019.4.5200.9083]

Large security budgets come with quite a lot of perks. Being able to do internal threat hunting is certainly one of them. And there’s nothing more threat hunters like to look for than anomalies in their data. YARA rules are just one way of finding odd things just laying about.

The string “Select * From Win32_SystemDriver” is probably found in quite a few of them. That is why the attackers chose to hide all such noisy strings with a combination of compression and Base64 encoding. Such a two step approach was necessary because there are also quite a few hunting rules out there that look for Base64 variants of aforementioned string.

By reversing those steps, C07NSU0uUdBScCvKz1UIz8wzNooPriwuSc11KcosSy0CAA== found above becomes “Select * From Win32_SystemDriver”. And all the threat hunting rules stay none the wiser.

Such string obfuscation is repeated throughout the code. And that’s the balance between standing out in a software developer review and fooling the security systems, a gamble that has paid off for the attackers.

Preventing supply chain attacks

Very few security companies are focused on securing the software supply chain. For most, talking about reducing the risks that these types of attacks pose is far-off. In many ways, we’re still in the problem awareness phase. And, as unfortunate as they are, incidents like this help draw attention to this multifaceted problem, one that equally affects those that ship software and those that consume it.

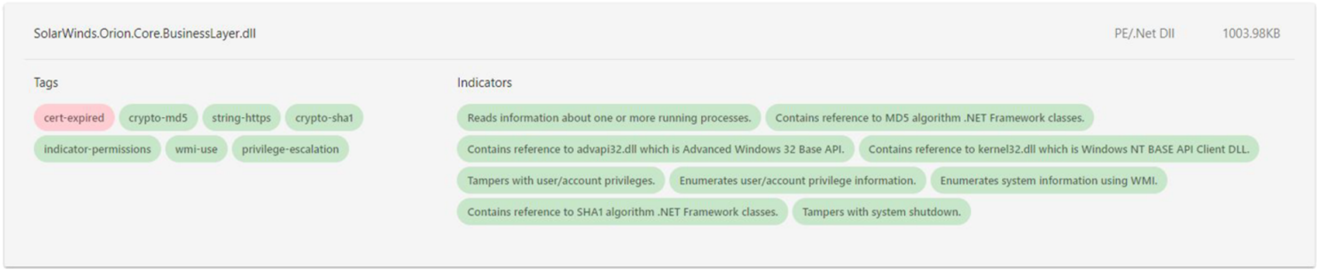
ReversingLabs research and development teams pride themselves with thinking about such big problems before they become widespread concerns. To that goal, we built many prototypes of products and solutions to address such problems.

Software supply chain protection is certainly a huge problem waiting to be solved. And internally, we've defined product strategies about protecting both sides of the equation - the developer and the user.

We envisioned a system able to scan “gold” software release images prior to their release or consumption. This system is purposely built to look for software tampering, digital signing, and build quality issues. It is ingrained into the continuous software development and release cycle, with the aim to bring these issues to the surface and provide guidance in eliminating them.

One key aspect of such a system is the ability to pinpoint behavioral differences between compiled software versions. Dubbed static behavioral indicators, these descriptions translate the underlying code actions into the effects they could have on the machine that runs them.

When layed out as a difference between added (green) and removed (red) code, the effects of software behavior changes become apparent. For the backdoored SolarWinds binary, this raises a number of security alarms that would have made it possible to catch this supply chain attack much sooner.



SolarWinds.Orion.Core.BusinessLayer.dll PE/.Net Dll 1003.98KB

Tags

🟥 cert-expired
🟩 crypto-md5
🟩 string-https
🟩 crypto-sha1
🟩 indicator-permissions
🟩 wmi-use
🟩 privilege-escalation

Indicators

🟩 Reads information about one or more running processes.
🟩 Contains reference to MD5 algorithm .NET Framework classes.
🟩 Contains reference to advapi32.dll which is Advanced Windows 32 Base API.
🟩 Contains reference to kernel32.dll which is Windows NT BASE API Client DLL.
🟩 Tampers with user/account privileges.
🟩 Enumerates user/account privilege information.
🟩 Enumerates system information using WMI.
🟩 Contains reference to SHA1 algorithm .NET Framework classes.
🟩 Tampers with system shutdown.

Figure 11. - Static behavior diff between ver. 2019.4.5200.8890 and ver. 2020.2.5300.12432

The following list highlights important static code behavior changes between the first tampered version and the one which contains the malicious backdoor code.

1. Reads information about one or more running processes
Having an application suddenly become aware of other running processes in the environment is highly unusual. For mature code bases, this functionality is typically added in major releases. There's typically a big feature planned behind this kind of code. And there's usually a good reason for the addition: some type of inter-process communication, or a desire to control running processes. In any other scenario, such an unplanned addition would be a cause for concern.

2. Contains references to MD5/SHA1 algorithm .NET Framework classes
While not highly unusual, hashing algorithms like MD5 and SHA1 are typically implemented to solve a specific problem. It's either some sort of content validation, authentication, or uniqueness check. Each of these can usually be mapped to a high-level requirement and tracked back to a feature modification request or a similar development task.

3. Contains references to kernel32.dll / advapi32.dll native Windows API
Referencing native Windows APIs from .NET library all of a sudden is very unusual. While the underlying code that interacts with the system is a necessity, even for managed applications, there are better ways of doing it. For example, provided language runtimes can typically achieve the same effect as what most developers require from native functions, but without having to deal with type uncertainty. By itself, regardless of the supply chain attack context, this is what developers refer to as code smell.

4. Enumerates system information using WMI
Windows Management Instrumentation (WMI) is a set of system functions that enable the application to get information on the status of local and remote computer systems. IT administrators use these functions to manage computer systems remotely. Understanding why such functionality is added suddenly is crucial. It is unlikely that the scope of the application has changed so dramatically that the interaction between remote computer systems has become a part of its key tasks. And if the goal is to retrieve something from the local system, there might already be code that has that information.

5. Enumerates and tampers with user/account privileges
Looking up user or account privileges is typically the first step in having them elevated. Running code at elevated privileges is done to perform a limited action, like copying files to restricted folders, manipulating running processes, changing system setting, etc. These are all actions that must have a firm reason behind them, and adding them to a mature code base is at least questionable. A developer should be made aware of this type of thing, and should have to sign off on it.

6. Tampers with system shutdown
Sticking with the theme of unnecessary privileges for an application, we have a big red flag

at the end. Being able to shutdown or reboot a computer isn't something that's added to code unexpectedly. That is a feature that takes coordination between multiple code components, and is usually implemented at a single location within the application. Having it appear elsewhere is definitely cause for concern.
Regardless of the side of the software deployment process one finds themselves on, a report about the impact of software code changes is an invaluable piece of information. For software developers, it can lead to informed decisions about the underlying code behavior. And for software consumers, it can ensure detection of anomalous code additions. Either way, the impact of such a system is transformative to software deployment processes. It serves as a verification barrier that can make it harder for these kinds of software supply chain attacks to recur.

New Control Mechanisms Needed

SUNBURST illustrates the next generation of compromises that thrive on access, sophistication and patience. For companies that operate valuable businesses or produce software critical to their customers, inspecting software and monitoring updates for signs of tampering, malicious or unwanted additions must be part of the risk management process. This type of tampering exploits software distributions that are trusted by the traditional security software stack, which is unique in comparison to known malicious implants. The distributions could not be easily inspected, if at all, by any perimeter control. Hiding in plain sight behind a globally known software brand or a trusted business-critical process, gives this method access that a phishing campaign could only dream to achieve.

Most cyber security frameworks such as NIST CSF document the need for continuous risk management and inspection of data and software. This, in turn, includes the need that all third party and open source software, whether built internally or externally, be continually inspected for tampering, malicious content, or any unwanted characteristics that clash with an organization's acceptable policies.

ReversingLabs is always thinking about the big challenges that lay ahead. We'd be happy to discuss our viewpoints and offer solutions towards reducing organizational software supply chain risks. Please get in touch so that we can solve these problems together.

Referenced files:

File name	SolarWinds.Orion.Core.BusinessLayer.dll
Version	2019.4.5200.8890
TimeStamp	Thu Oct 10 13:26:39 2019
Hash	5e643654179e8b4cfe1d3c1906a90a4c8d611cea
Note	File contains placeholder OrionImprovementBusinessLayer class.

File name	SolarWinds.Orion.Core.BusinessLayer.dll
Version	2019.4.5200.9083
TimeStamp	Tue Mar 24 08:52:34 2020
Hash	76640508b1e7759e548771a5359eaed353bf1eec
Note	First known instance of the backdoored library.

File name	SolarWinds.Orion.Core.BusinessLayer.dll
Version	2020.2.5200.12394
TimeStamp	Tue Apr 21 14:53:33 2020
Hash	2f1a5a7411d015d01aaee4535835400191645023
Note	Contains malicious backdoor code.

File name	SolarWinds.Orion.Core.BusinessLayer.dll
Version	2020.2.5300.12432
TimeStamp	Mon May 11 21:32:40 2020
Hash	d130bd75645c2433f88ac03e73395fba172ef676
Note	Contains malicious backdoor code.

File name	SolarWinds-Core-v2019.4.5220-Hotfix5.msp
Version	2019.4.5220
TimeStamp	Tue March 24 10:57:09 2020
Hash	1b476f58ca366b54f34d714ffce3fd73cc30db1a
Note	HotFix patch containing the first known backdoor instance.

MORE BLOG ARTICLES